LAB-μE

FACULTAD DE INGENIERIA
Universidad de Buenos Aires

The X$_{\text{FIRE}}$ Processor Architecture Specification

Rev. 0.1
February 20, 2016

Abstract

This document contains all the functional specifications needed for designing, implementing and verifying the X_{FIRE} Processor Architecture.

Revision history

Rev.	Date	Author/s	Description
1.0	August 31, 2015	Alpago, Octavio Natale, Luciano Luciano Natale Zacchigna, Federico Giordano	First draft version.

Contents

List of Figures

List of Tables

0.1 About This Document

0.1.1 Introduction

Welcome to the X_{FIRE} Processor Architecture! The X_{FIRE} (pronoiced "Crossfire") Processor Architechture is targeted to be synthesized on FPGA and ASICs and its objective is to provide a solution for embedded systems. The processor is designed under the full RISC philosophy and is mainly focused on providing a simple and implementation agnositc architecture.

This manual describes the architechture, the instruction set, the internal registers, the addressing modes, the supported data types, the memory model, the peripheral interface, exceptions and interruptions. This manual does not describe neither implementation nor details such as pipeline, cache and other optimizations, leaving those details to the implementation itself.

0.1.2 List of contributors

By alphabetic order:

- Alpago, Octavio - oalpago@gmail.com
- Natale, Luciano - luchonat@gmail.com
- Sanca, Gabriel - gandres.sanca@gmail.com
- Zacchigna, Federico Giordano - federico.zacchigna@gmail.com

0.1.3 Conventions

Chapter 1

Architecture description

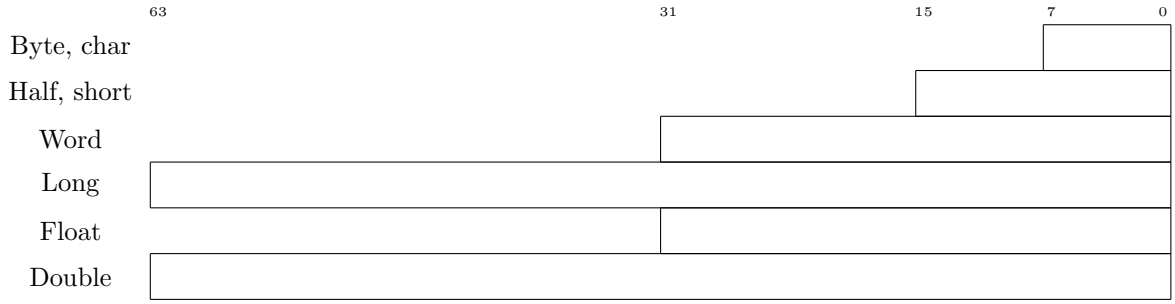

Figure 1.1: Data sizes.

1.1 Data types

Integer supported data types are: *byte* (8 bits), *half* (16 bits), *word* (32 bits), *double word* (64 bits). Floating point supported data types are: *single-precision* (32 bits) and *double-precision* (64 bits). Data sizes are shown in figure 1.1.

- *Byte* is also referred as *char*.
- *Half* is also referred as *short*.
- *Word* is also referred as *integer*.
- *Double word* is also referred as *long*.
- *Single-precision floating point* is also referred as *float*.
- *Double-precision floating point* is also referred as *double*.

1.2 Internal Registers

The reset state of all the registers is 0 including the special purpose registers. Exception to this rule is the L0 register when implemented, whose reset value is 0x800000000.

1.2.1 General Purpose Registers

- There are 32 general purpose registers (GPRs) named R0, R1, ... R31.
- The GPRs can be used as even-odd pairs holding long integer registers named R0, R2, R4, ..., R30.
- Each GPR has 4 flag bits: carry (C), negative (N), overflow (V), zero (Z) which store the ALU operation status. The negative flag is implicit and corresponds to the MSB of the GPR (bit 31).
- When loading data into a GPR from memory, carry (C), overflow(V) and zero (Z) flags are left unmodified; negative (N) flag is implicit in the data loaded in the register.
- When an ALU operation writes back double words on pair of registers, the even register holding the lower bits of the long data has its carry (C), overflow (V) and zero (Z) flags unmodified; negative (N) flag is implicit in the data hold by the register, the odd register holding the upper bits of the long data, has all flags consistent with the operation.
- Byte and half data types are loaded into the low portion of GPRs from memory with either zeros or sign bit replicated to fill the 32 bits depending on the instruction.
- There are 32 single-precision floating point registers (FPRs) named F0, F1, ..., F31.
- The FPRs can be used as even-odd pairs holding double-precision floating point registers named F0, F2, F4, ..., F30.
- Each FPR has 4 flag bits: is-a-number (A), is-infinity (I), negative (N), zero (Z) which store the FPU operation status. The negative flag is implicit and corresponds to the MSB of the FPR (bit 31).
- When loading data into a FPR from memory, is-a-number (A), is-infinity (I), and zero (Z) are left unmodified; negative (N) flag is implicit in the data loaded in the register.
- When an FPU operation writes back double floats on pair registers, the even register holding the lower bits of the double float has its is-a-number (A), is-infinity (I), and zero (Z) flags unmodified; negative (N) flag is implicit in the data loaded in the register, the odd register holding the upper bits of the double float has all its flags consistent with the operation.

1.2.2 Special Purpose Registers

The procesor provides a set of special registers, some of them are accesible with special instructions. Special registers are:

- The config register (CFG) is 6 bits wide and stores the configuration status of the processor. Its content is set and read with special register instructions. Since special moving instructions transfer 32 bits, bits from 31 down to 6 are discarded on write, and return zero on read.

  Bit 0 stores the execution mode. If unset, the processor is running in supervisor mode. If set, it is running in user mode. Read section 1.7 for more information on execution modes.

  Bit 1 stores the configuration support for nested exceptions. If unset, the processor does not provide support for nested exceptions, while if set, it does. Read subsection 1.6.1 for more information on the nested exceptions support.

  Bits 3 down to 2 selects the system tick timer operation mode. Bit 2 selects free running mode. If unset, the system tick timer is in free runing mode. If set is in auto stop mode. When running in auto stop mode, bit 3 works as system tick timer start. This bit is not sticky, meaning that it goes back to '0' one clock cycle after it was written and always returns '0' on read. Read subsection 1.6.3 for more information on the systick exception.

  Bits 5 down to 4 selects the rounding mode for floating point operations. The encoding is as follows:

  - '00': RN - Round to nearest
  - '01': RZ - Round towards zero
  - '10': RP - Round towards plus infinity
  - '11': RM - Round towards minus infinity

- The system timer period register (STP) is a 22 bits register that configures the system timer reload value. Read subsection 1.6.3 for more information on the systick exception.

- The mask register (MR) is 33 bits wide and provides the mechanisms to enable or disable the corresponding source of interruption or cause. The most significant bit is the global mask bit. This bit is not software accesible; this means that instructions cannot read or write to this special bit and its content is set and read automatically by the architecture. When it is set, all exceptions are disabled. Its content is set and read with special register instructions. Read section 1.6 for more information on this register.

- The instruction register (IR) is 32 bits wide and stores the opcode of the instruction that is being executed. Its content is read with special register instructions. A writing operation on this register has no effect.

- The system timer register (TMR) is 32 bits wide and corresponds to a 32 bits down counter that generates the systick exception when reaches zero. The reload value of the system timer can be configured trough the STP register. This register is read only for software: a writing operation has no efect on this register. Read subsection 1.6.3 for more information on this register.

- The peripheral write data register (PWD) is 32 bits wide and provides outgoing communication with peripherals. Its content is set and read with special register instructions.

- The peripheral write address register (PWA) is 32 bits wide and provides addressing for peripherals. Its content is set and read with special register instructions.

- The peripheral read data register (PRD) is 32 bits wide and provides incoming communication with peripherals. Its content is read with special register instructions. A writing operation on this register has no effect.

- The peripheral read address register (PRA) is 32 bits wide and provides incomming identification of peripherals. Its content is read with special register instructions. A writing operation on this register has no effect.
- The program counter (PC) is 30 bits wide addressing a memory space which ranges from `0x00000000` to `0xFFFFFFFC`. This register is not software addressable. The current address is given by $4 * PC$.
- The cause register (CR) is 32 bits wide and stores the cause of exceptions and interrupts. Each bit represents a different exception cause / interrupt source. When an exception or interrupt condition is detected, the corresponding bit must be set, and shall remain set until the event is handled. The hardware is responsible of reseting the corresponding bit when execution of the handler is started. This register is not software addressable. Read section 1.6 for more information on this register.
- The link register stack (LRS) is a set of 32 registers named $L0, L1, \ldots, L31$ which store the return address (PC + 1) value **only** from exceptions. Each register is 36 bits wide. This register file is optional and adds support for nested exceptions. If not supported, then one link register (L0) must be implemented. These registers are not software addressable. Read subsection 1.6.1 for more information.
- The exception stack pointer (ESP) is a 5 bits wide register and points to the top of the LRS. This register is optional and adds support for nested interruptions. Read subsection 1.6.1 for more information.

The structure and organization of registers is shown in Tables **??** and 1.3.

Addressing of Special Registers

Software addressable special registers are set and read with special register instructions called *movi2s* and *movs2i*. For more informtion on this instructions, read section 2.1. Special registers addressing is shown in Table 1.1.

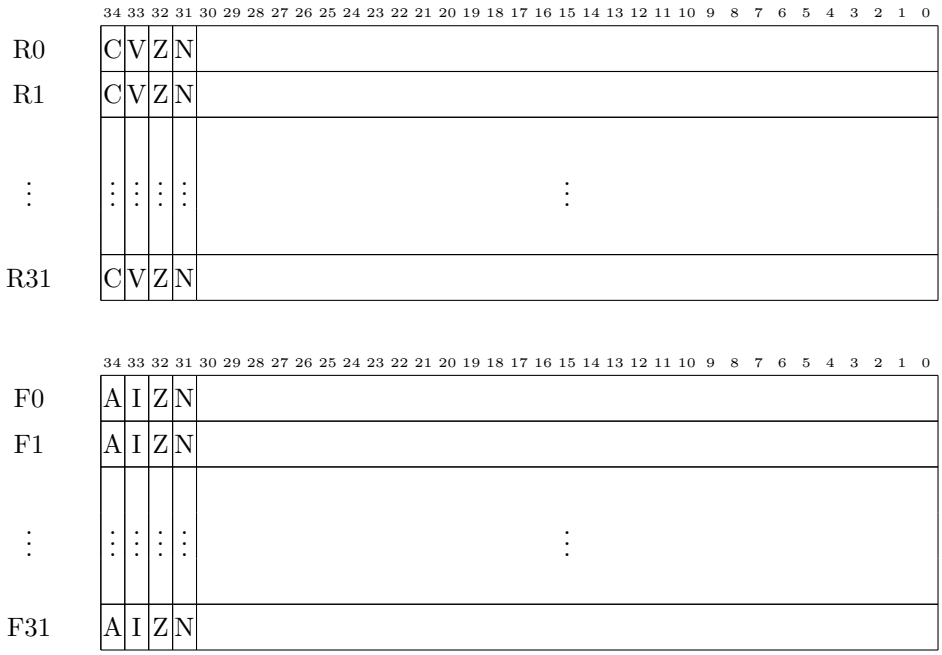

Figure 1.2: General registers structure and organization.

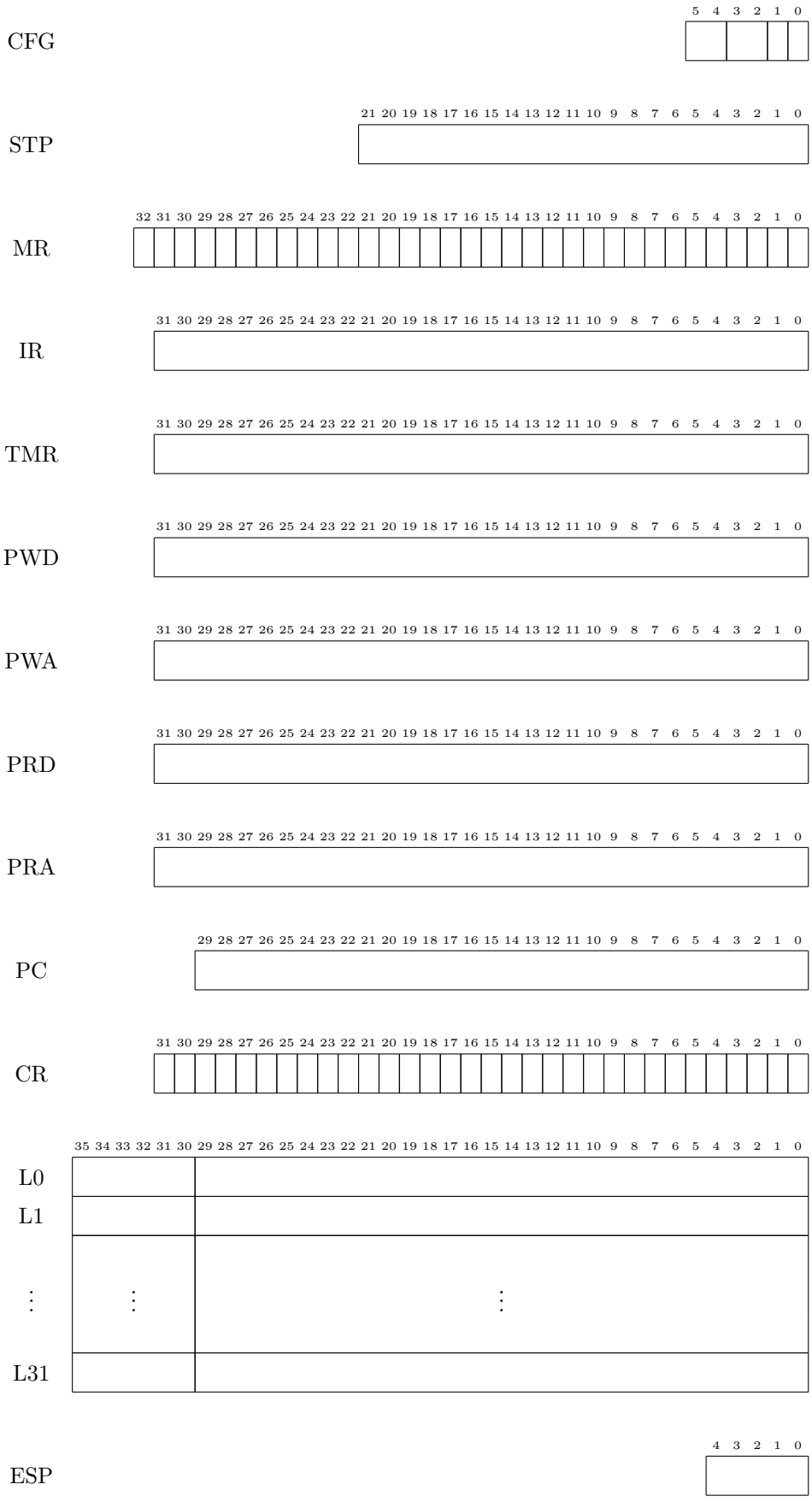

Figure 1.3: Special registers structure and organization.

Register Name	**Mapped Address**	**Hardware Size [bits]**	**Software Size [bits]**	**Hardware Access**	**Software Access**
CFG	0x00	6	32	Read only	Read-Write
STP	0x01	22	32	Read only	Read-Write [1]
MR	0x02	33	32	Read-Write	Read-Write [2]
IR	0x03	32	32	Read-Write	Read only
TMR	0x04	32	32	Read-Write	Read only
PWD	0x05	32	32	Non accessible	Read-Write
PWA	0x06	32	32	Non accessible	Read-Write
PRD	0x07	32	32	Read only	Read only
PRA	0x08	32	32	Read only	Read only
PC	NA	30	0	Read-Write	Non accessible
CR	NA	32	0	Read-Write	Non accessible
L0	NA	36	0	Read-Write	Non accessible
⋮	⋮	⋮	⋮	⋮	⋮
L31	NA	36	0	Read-Write	Non accessible
ESP	NA	5	0	Read-Write	Non accessible

Table 1.1: Special registers addressing.

Port name	Description	Size (Bits)	Type
arst	Asynchronous high level active reset	1	Input
clk	Posedge active clock	1	Input
ena	High level active synchronous enable	1	Input
dm_wr_data	Data memory write data bus	32	Output
dm_rd_data	Data memory read data bus	32	Input
dm_address	Data memory address bus	32	Output
dm_wr_ena	High level active data memory write enable	1	Output
im_rd_data	Instruction memory read data bus	32	Input
im_address	Instruction memory address bus	32	Output
us	User/Supervisor execution mode ('1': user, '0': supervisor)	1	Output
pwd	Peripheral write data	32	Output
pwd_wr_ena	Peripheral write write enable	1	Output
prd	Peripheral read data	32	Input
pwa	Peripheral write address	32	Output
pra	Peripheral read address	32	Output
irq	High level active interruption request	1	Input
iack	High level active interruption acknowledge	1	Output

Table 1.2: Processor I/O description.

1.3 Ports Interface

Data type	Size	Address aligment condition
Byte	8 bits	xx...xxxx
Half	16 bits	xx...xxx0
Word	32 bits	xx...xx00

Table 1.3: Data Memory Address Alignment Condition.

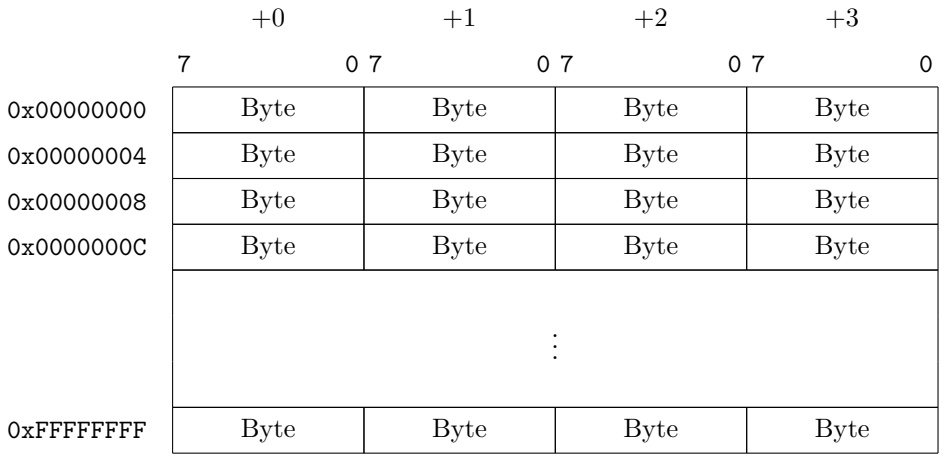

Figure 1.4: Byte data type aligment.

1.4 Memory Organization

1.4.1 Data Memory

Memory interface

- Read/write cycles take only one clock period.
- Memory is organized in big-endian format.
- No long-data transfer (64 bits) is supported.
- Only aligned data is supported:
  - A byte data can be located in any address. See figure 1.4.
  - A half data can only be located in even addresses. See figure 1.5.
  - A word data can only be located in addresses multiple of 4. See figure 1.6.

  Read table 1.3

Addressing Modes

The only one supported addressing mode is displacement with signed 13 bit fields.

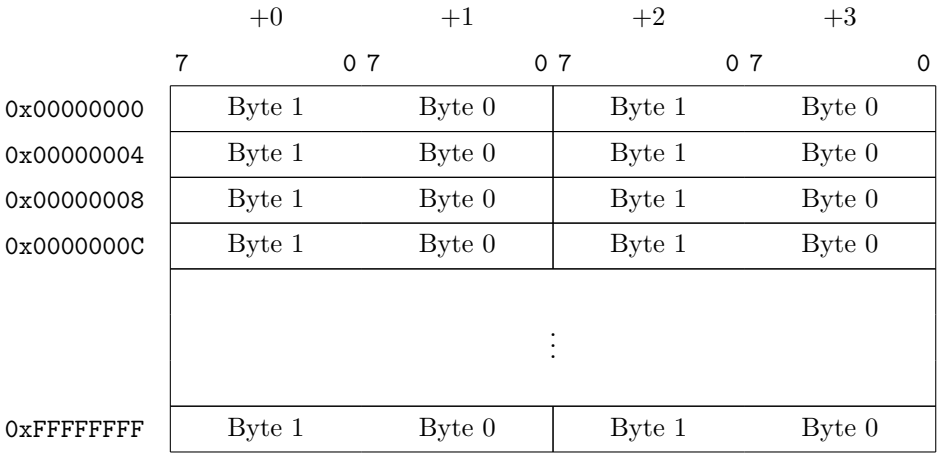

Figure 1.5: Half data type aligment.

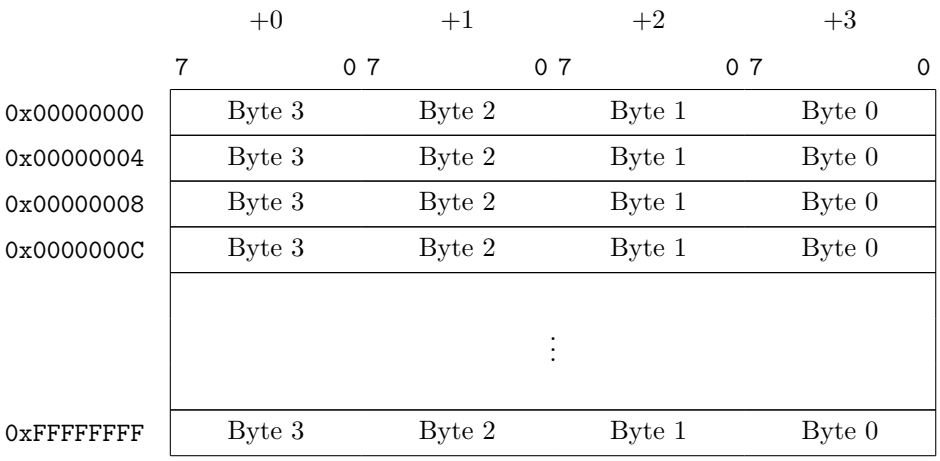

Figure 1.6: Word data type aligment.

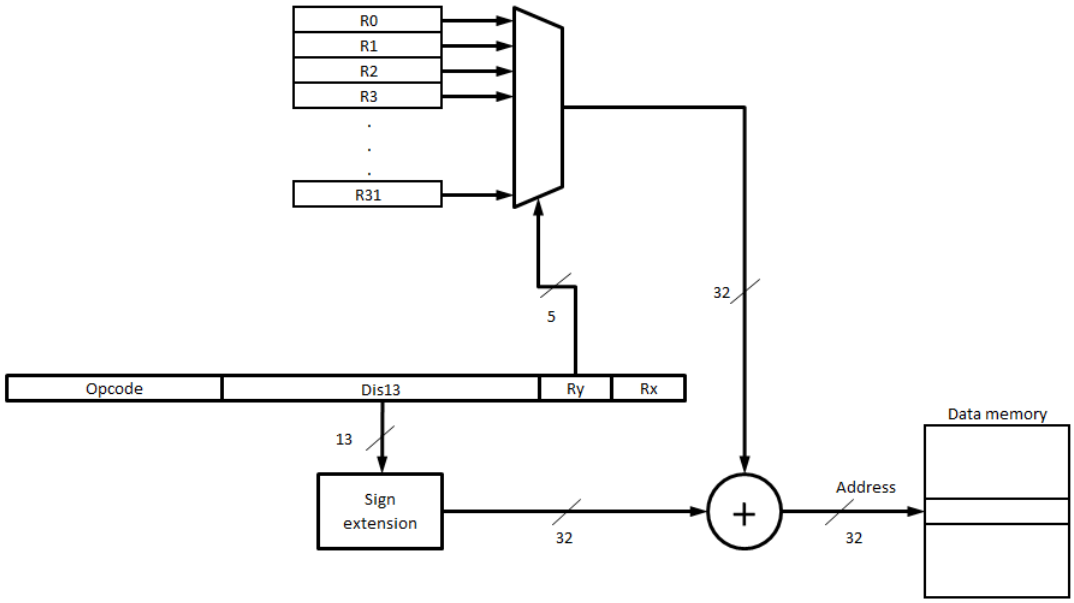

Figure 1.7: Effective data memory address calculation.

1.4.2 Instruction Memory

- Instruction memory is read-only for the procesor. Read cycle takes only one clock period.
- Memory is organized in big-endian format.
- Only 32 bits single-word instructions are supported.
- An instruction can only be located in addresses multiple of 4.
- First 32 adresses (128 bytes) are reserved for exception vectors.

See figure 1.8

Program counter, branch and jump target addresses calculations

With instructions being 4 bytes long and unaligned access to instruction memory being not allowed, the effective instruction memory address when stored in GPR's is given by bits 31 downto 2. Branch or jump instructions that encode the address displacement in the corresponding field, also take advantage of this aligment restriction, the two lower bits of the destination address are implicit (zeroes).

1.4.3 Memory Timing Diagrams

	+0	+1	+2	+3
	7 0	7 0	7 0	7 0
0x00000000	Byte 3	Byte 2	Byte 1	Byte 0
0x00000004	Byte 3	Byte 2	Byte 1	Byte 0
0x00000008	Byte 3	Byte 2	Byte 1	Byte 0
0x0000000C	Byte 3	Byte 2	Byte 1	Byte 0
		⋮		
0xFFFFFFFF	Byte 3	Byte 2	Byte 1	Byte 0

Figure 1.8: Instruction memory organization.

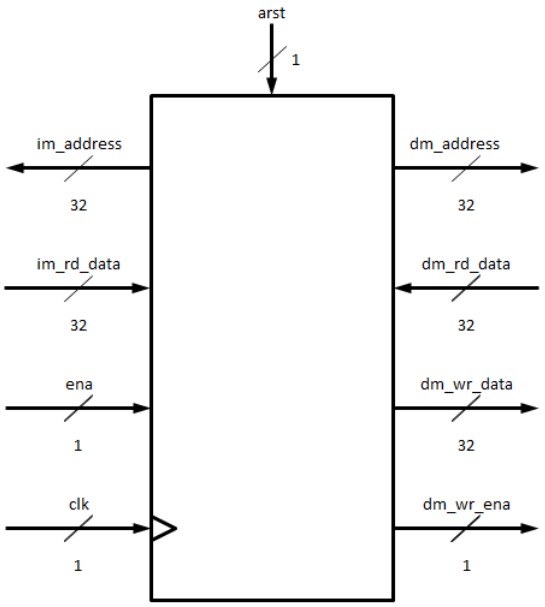

Figure 1.9: Memory Port Interface.

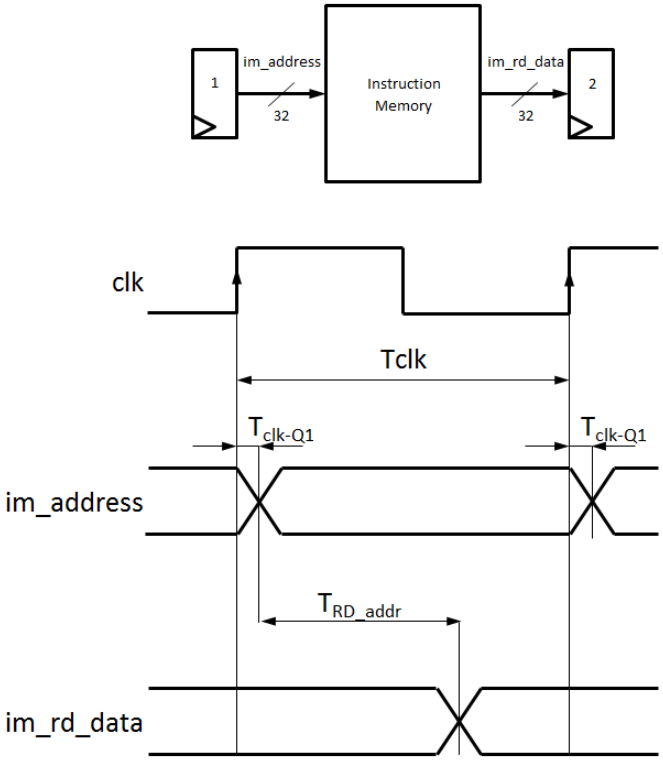

Figure 1.10: Instruction memory read cycle timing diagram.

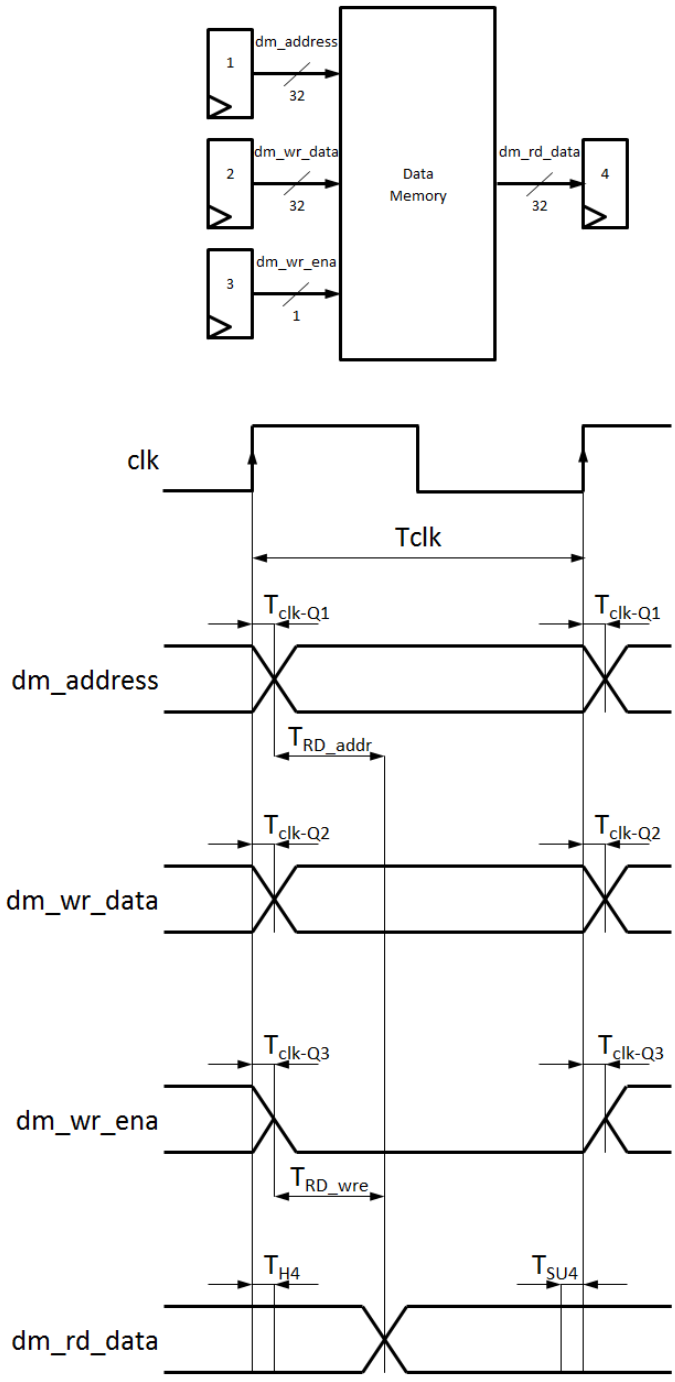

Figure 1.11: Data memory read cycle timing diagram.

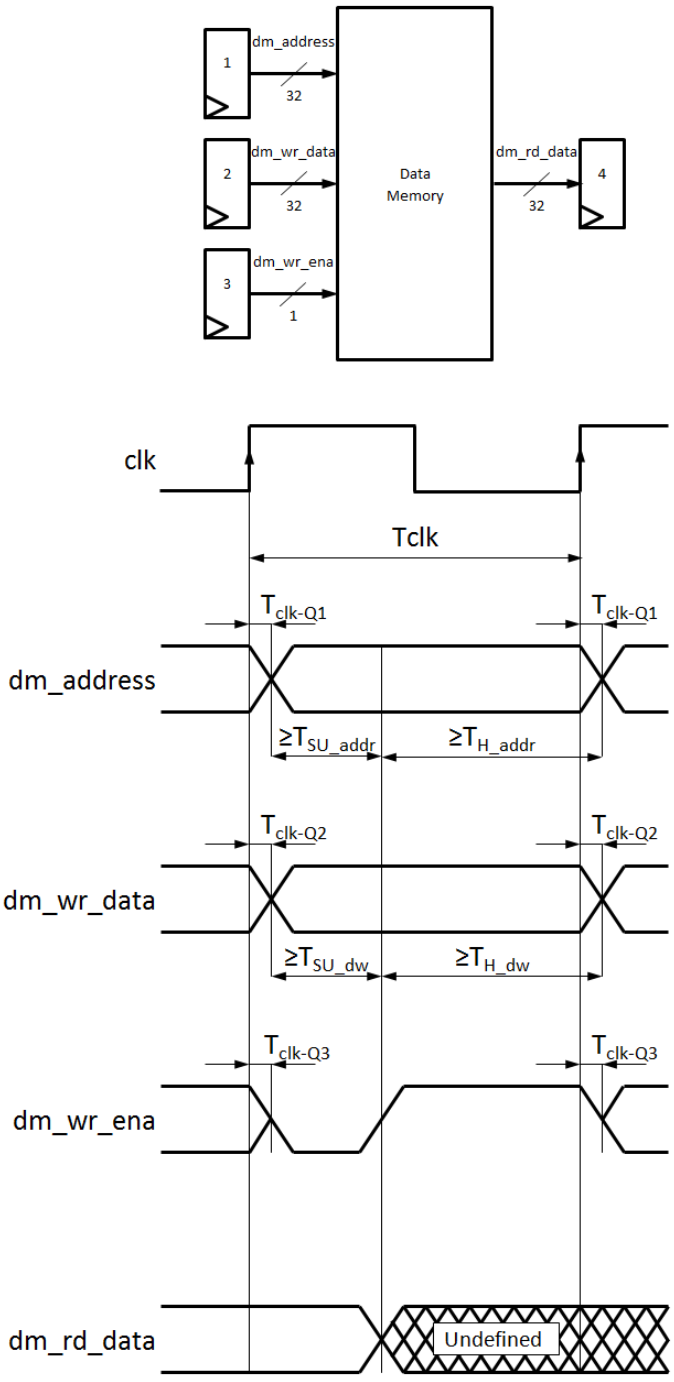

Figure 1.12: Data memory write cycle timing diagram.

Parameter	Name	Description
T_{SU_dw}	Write data setup time	Minimum time to set dm_wr_data to a stable value before the rising edge of dr_wr_ena.
T_{H_dw}	Write data hold time	Minimum time to hold dm_wr_data to a stable value after the rising edge of dr_wr_ena.
T_{SU_addr}	Address setup time	Minimum time to set dm_address to a stable value before the rising edge of dr_wr_ena.
T_{H_addr}	Address hold time	Minimum time to hold dm_address to a stable value after the rising edge of dr_wr_ena.
T_{RD_addr}	Read access time	Maximum time taken by dm_rd_data to become valid after dm_address changes.
T_{RD_wre}	Read access time	Maximum time taken by dm_rd_data to become valid after dm_wr_ena goes low.
T_{clk}	Clock period	Period of time between two consecutive rising edges of clock signal.
T_{clk-Q_i}	Clock to output time	Propagation time of the register i.
T_{SUi}	Register setup time	Setup time of the register i.
T_{Hi}	Register hold time	Hold time of the register i.

Table 1.4: Timing Parameters Description.

Port name	Description	Size (Bits)	Type
clk	Posedge active clock	1	Input
start	High active strobe start execution Note: This signal must be implemented only for a non-pipelined ALU.	1 1	Input Input
rst	High active asynchronous reset	1	Input
srst	High active synchronous reset	1	Input
ena	High active synchronous enable	1	Input
format	Word or long word operands '0': Word '1': Long	2	Input
operation	Selects the operation to be executed	XX	Input
operand_x	First operand	64	Input
operand_y	Second operand	64	Input
flags	Ouptut flags Bit 0: zero Bit 1: overflow Bit 2: carry Note: Negative flag is implicit (MSB of the result)	3	Output
result	Operation result	64	Output
done	Up when finished and until start is asserted Note: This signal must be implemented only for a non-pipelined ALU.	1 1	Output Input

Table 1.5: ALU Port interface

1.5 Execution of instruccions

1.5.1 The concept of 'executing only if resources available'

When decoded, a particular instruccion will only be executed if the resources used by the instruction are available.

Availability of resources means that source and destination operands and the corresponding execution unit are free to perform the instruction execution. Otherwise, the instruction will be stalled till all resources are available. This mechanism ensures that all instruction are always properly executed.

1.5.2 ALU

Esto debería sacarse de acá The integer arithmetic logic unit (ALU) operates on 32 and 64 bits integer data sources and result. It also needs to report carry, overflow, zero and negative flags. The ALU is in charge of all arithmetic and logic operations on GPR's.

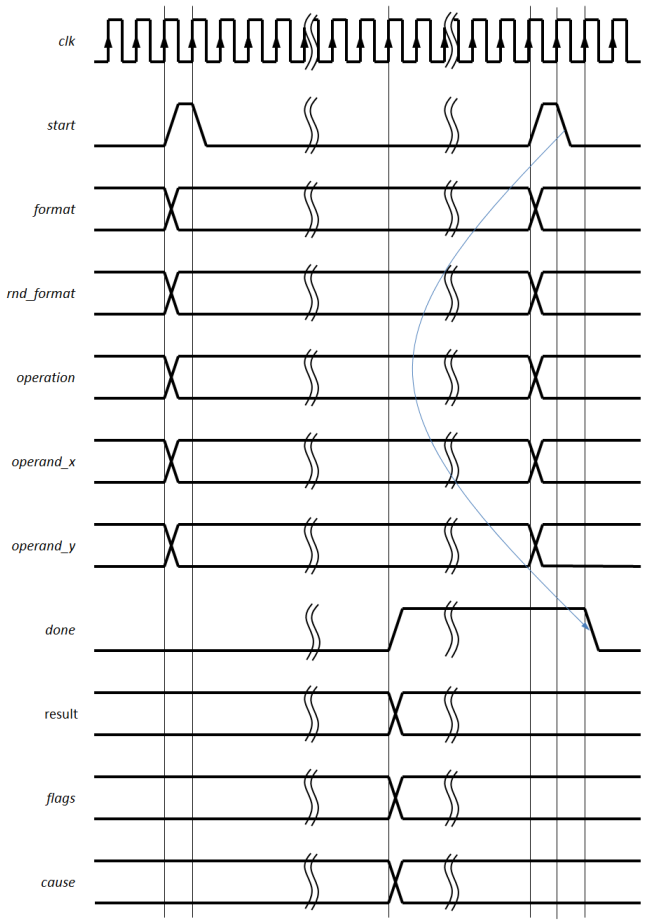

Figure 1.13: FPU timing diagram.

1.5.3 Floating point unit

Esto debería sacarse de acá The floating point unit (FPU) is optional to the design. The FPU is in charge of all arithmetic and logic operations on FPR's operating on 32 and 64 bits formats.

When FPU is not provided, floating point operations must always generate an exception and the corresponding exception mask should not be set. In this case, software support must be provided to handle the exception and returning a consistent result, so the execution of the program can be resumed. Read section 1.6 for more information.

If the optional FPU is present, it must be IEEE 754 compliant and provide the following interface.

1.5.4 FPU timig diagrams

Port name	Description	Size (Bits)	Type
clk	Posedge active clock	1	Input
start	High active strobe start execution Note: This signal must be implemented only for a non-pipelined FPU.	1 1	Input Input
rst	High active asynchronous reset	1	Input
srst	High active synchronous reset	1	Input
ena	High active synchronous enable	1	Input
fp_format	Single or double precission operands '0': Single '1': Double	2	Input
rnd_mode	Establishes rounding mode for the result '00': RN - Round to nearest '01': RZ - Round towards zero '10': RP - Round towards plus infinity '11': RM - Round towards minus infinity	2	Input
operation	Selects the operation to be executed	XX	Input
operand_x	First operand	64	Input
operand_y	Second operand	64	Input
cause	Exception cause Bit 0: Underflow Bit 1: Overflow Bit 2: Division by zero Bit 3: Invalid operation	4	Output
flags	Ouptut flags Bit 0: Zero Bit 1: Infinity Bit 2: A number Note: Negative flag is implicit (MSB of the result)	3	Output
result	Operation result	64	Output
done	Up when finished and until start is asserted Note: This signal must be implemented only for a non-pipelined FPU.	1 1	Output Input

Table 1.6: ALU Port interface

1.6 Exceptions

Exceptions are handled by raising a flag, the offending (or interrupted) instruction address + 1 is stored in an L register. When an exception condition is detected, the PC is loaded with the indexed special address *idx* (exception identification immediate). Reset is treated just as another source of exception. Being the CR 32 bits wide, the architecture provides a space of 32 different sources of exceptions, so the first 32 instruction addresses (from 0x00000000 to 0x0000007C) are reserved for exception handlers.
Exceptions should only be handled if the correspondig bit in MR is unset. As stated in section 1.2, MR has 33 physical bits. The higher bit is a global mask for all exceptions and it is a hardware mechanism to avoid nested exceptions when they are disabled.
Interrupts are handled as exceptions. Instructions stored at the special addresses must provide the handling for the exceptions. Priorities are fixed and they correspond with the order stated in the listing below. Lower ordered exceptions always have higher priority over other sources of exceptions.
The lowest priority is software generated exceptions (*trap* instruction); thus it has the highest number, 31. On the other end, reset has the highest priority (and the lowest number, 0).
The indexed address schema provides space for one instruction to start the handling routine.
Sources of exception not defined in this document are left free for future revisions of the architecture or implementation specific exceptions.

- E0: Reset
- E1: System tick.
- E2: Execute invalid instruction.
- E3: Misaligned data memory access.
- E4: Misaligned long GPR access.
- E5: Misaligned double FPR access.
- E6: Integer division by zero.
- E7: Long integer division by zero.
- E8: Instruction is a FPU operation.
- E9: Instruction is a FPU extendended operation.
- E10: FPU_V flag is set.
- E11: FPU_Z flag is set.
- E12: FPU_O flag is set.
- E13: FPU_U flag is set.
- E30: Interruption request.
- E31: Trap.

Step	Description	Assembly	Registers Status

1.	Exception *idx* is raised		if $(mr[idx] == `0')$ then
			$l_0[29:0] \leftarrow_{30} (pc+1)$
			$pc \leftarrow_{30} idx$
			$mr[32] \leftarrow_1 1$
			$cr[idx] \leftarrow_1 0$
			$cfg[0] \leftarrow_1 0$
			continue with step 2.
			else
			Do not handle the exception
			endif

2.	Jump to the exception handler routine	`j dis26`	$pc \leftarrow_{30} pc + sx_{30}(dis26)$
3.	Save context to be used in handler	User code	
4.	Handler routine	User code	
5.	Recover context	User code	
6.	Return from exception	`rfe`	$pc \leftarrow_{30} l_0[29:0]$ $mr[32] \leftarrow_1 0$ $cfg[0] \leftarrow_1 1$

Table 1.7: Nested Interruptions Disabled Interrupt Sequence

Nested exceptions support is configured by means of the bit 1 of the CFG register. If not supported, the CPU cannot accept a new source of exception until the handling of the current one has finished. On the other hand, when nested interruptions are supported, only a higher priority exception can interrupt the current one.

1.6.1 Exception handling

As stated earlier, the nested exception support can be enabled/disabled through the bit 1 of the CFG register.

Nested insterruptions disabled

In table 1.7 interruption handling with nested interruptions disabled is described; pc is the offending (or interrupted) instruction address.

Nested insterruptions enabled

In table 1.8 interruption handling with nested interruptions disabled is described; pc is the offending (or interrupted) instruction address.

Step	Description	Assembly	Registers Status

1.	Exception *idx* is raised		if $(mr[idx] == `0')$ && $(idx < l_{esp}[35:30])$ then
			$l_{esp}[29:0] \leftarrow_{30} (pc+1)$
			$esp \leftarrow_5 (esp+1)$
			$l_{esp}[34:30] \leftarrow_5 idx$
			$pc \leftarrow_{30} idx$
			$cr[idx] \leftarrow_1 0$
			$cfg[0] \leftarrow_1 0$
			continue with step 2.
			else
			Do not handle the exception
			endif

2.	Jump to the exception handler routine	`j dis26`	$pc \leftarrow_{30} pc + sx_{30}(dis26)$
3.	Save context to be used in handler	User code	
4.	Handler routine	User code	
5.	Recover context	User code	
6.	Return from exception	`rfe`	$esp \leftarrow_5 esp - 1$ $pc \leftarrow_{30} l_{esp}[29:0]$ $cfg[0] \leftarrow_1 1$

Table 1.8: Nested Interruptions Enabled Sequence Detail

1.6.2 Reset exception

As stated before, reset is just another exception, but there are a few differences with the other sources.
The reset exception cannot be masked: altough MR can be written with the lower bit set, the reset exception will not be masked. Also, it will not be masked by the global mask bit, enabling the reset to happen even if an exception is being handled.

Getting out of the reset exception and booting

The reset mechanism will set all registers with zero (except L0), though, the first instruction excecuted by the microprocesor is in address `0x00000000`, and as the CFG register is zeroed, the microprocesor will run in supervisor mode. Thus, the reset address is the vectored address for the reset exception. That instruction should be a jump to the reset exception handler. This handler should implement all the boot code necessary to set up the environment. As L0[29:0] is loaded with 0 (as a consequence of the reset) the code cannot call rfe to get out of the exception (calling rfe will load the PC with the reset address again). So CFG register must be manually set to user mode once initialization is done and everything is ready to start running user code via a *movi2s* instruction.

1.6.3 System tick

System tick is a periodic exception internally generated by the CPU hardware.The system timer is a 32 bits down counter that generates the systick exception when reaches zero. The reload value of the system timer can be configured trough the STP register.

The CFG register (bit 2) sets the system timer behavior. If unset, the timer will be in free running mode, meaning that once the timer reaches zero, it will be automatically reloaded with the value $(1024 * STP) + 1023$ and will continue counting down. On the other hand, if set, the system timer will stop when reaches zero and will reload the value $(1024 * STP) + 1023$ and resume the count down after CFG[3] is written with *movi2s*.

The bit 3 of the CFG register is not persistent, it means that goes back to '0' one clock cycle after it was written. Always returns '0' on read.

1.6.4 Interruptions

The CPU is hardware interrupted by rising the interrupt input port (*irq*) and exception E30 is launched if not masked. When the processor attends the interruption exception, the *iack* output is raised by the processor indicating that the exception was already handled. The *iack* output is set low by the CPU when a new interruption request occurs. Fig. 1.14 shows a timing diagram of the interruption sequence.

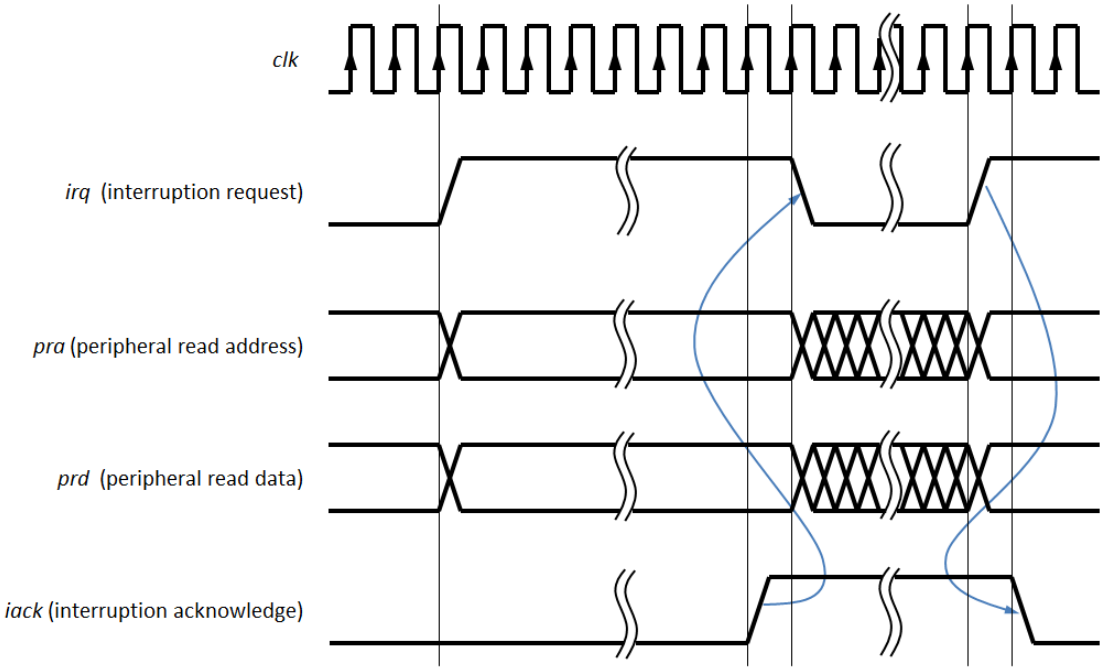

Figure 1.14: Interruption timing diagram.

1.7 Operating modes

The processor has two different operating modes: *supervisor mode* and *user mode*.

In supervisor mode, all instructions can be executed and the CFG[0] register is unset.

In user mode, some instructions can not be executed. Any attempt to execute one of those instructions causes an invalid operation exception (E2). The user mode is indicated by setting CFG[0].

The privileged instructions (those that can only be executed in supervisor mode) are the following:

- `movi2s`
- `movs2i`
- `rfe`

The reset mode is always supervisor mode.

The instruction *trap* is used by user programs to generate software exceptions (providing a mechanism to implement system calls).

The *rfe* returns the processor to user mode. Another way to return the processor to user mode is by setting CFG[0].

1.8 Peripherals

Comunication with peripherals is achieved through interruptions and four special porpouse registers: PWD, PRD, PWA, and PRA.

Whenever a peripheral should be written the sequence that must be followed is:

- PWA must be set using *movi2s* with the peripheral address.
- PWD must be set using *movi2s* instruction to copy the information from one of the GPR's. Automatically, a strobe is generated on the *pwd_wr_ena* output pin signaling the writting operation.

Whenever a peripheral should be read the sequence that must be followed is:

- PRA must be set by external hardware.
- PRD must be set by external hardware.
- The *irq* line is raised.

Since the system is synchronous with the CPU clock, the three events avobe can happen at the same time (i.e. in the same clock cycle).

Chapter 2

Detailed instruction set description

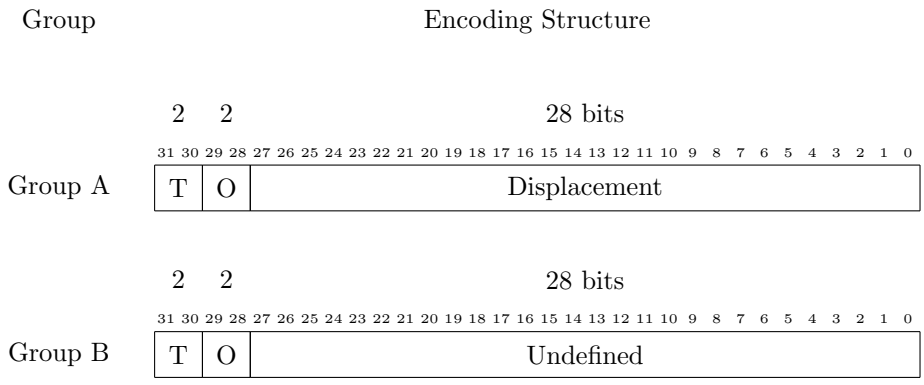

Figure 2.1: Type 0 instruction encoding structure

2.1 Instruction Set Architecture

2.1.1 Instruction format

There are four intruction main types according to how many registers are used as arguments of the instruction, named Type 0, 1, 2 and 3. The type of instruction is identified by the two most significant bits of the opcode. This types of instructions are further organized in groups, which identify if the registers are used as sources or destinations and the role of the remaining bits of the opcode, such as immediate data, displacement, subopcodes or size of the operation (width of the affected data).

Type 0 instructions

Type 0 instructions are encoded with only two bits (bits 28 and 29). Usage of the 28 remaining bits depend on the instruction itself. This type of instructions are further divided in two groups: A and B. Group A has a 28 bit displacement field. Group B does not specify those bits. The enconding structure is shown in figure 2.1.

Type 1 instructions

Type 1 instructions are encoded with four bits (bits 26 to 29) and make use of one register, either as source or destination of the operation. This type of instructions is further divided in four groups: A, B, C and D. Group A uses a source register (addressed by bits 21 to 25) and a twenty one bits displacement field (bits 0 to 20). Group B uses a destination register (addressed by bits 16 to 20) and a 16 bits displacement field (bits 0 to 15), leaving 5 bits unespecified (bits 21 to 25). Group C uses a source register (addressed by bits 21 to 25), a 4 bits subopcode (bits 17 to 20) and a 17 bits displacement field (bits 0 to 16). Group D uses a destination register (addressed by bits 16 to 20) and a 16 bits immediate field (bits 0 to 15), leaving 5 bits unespecified (bits 21 to 25). The encoding structure is shown in figure 2.2.

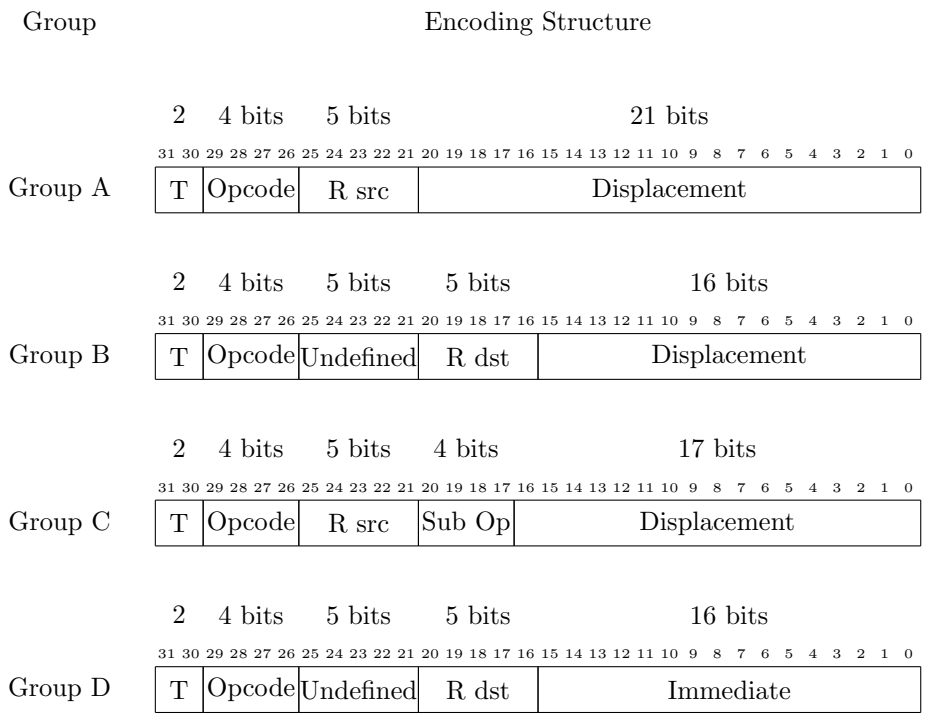

Figure 2.2: Type 1 instruction encoding structure

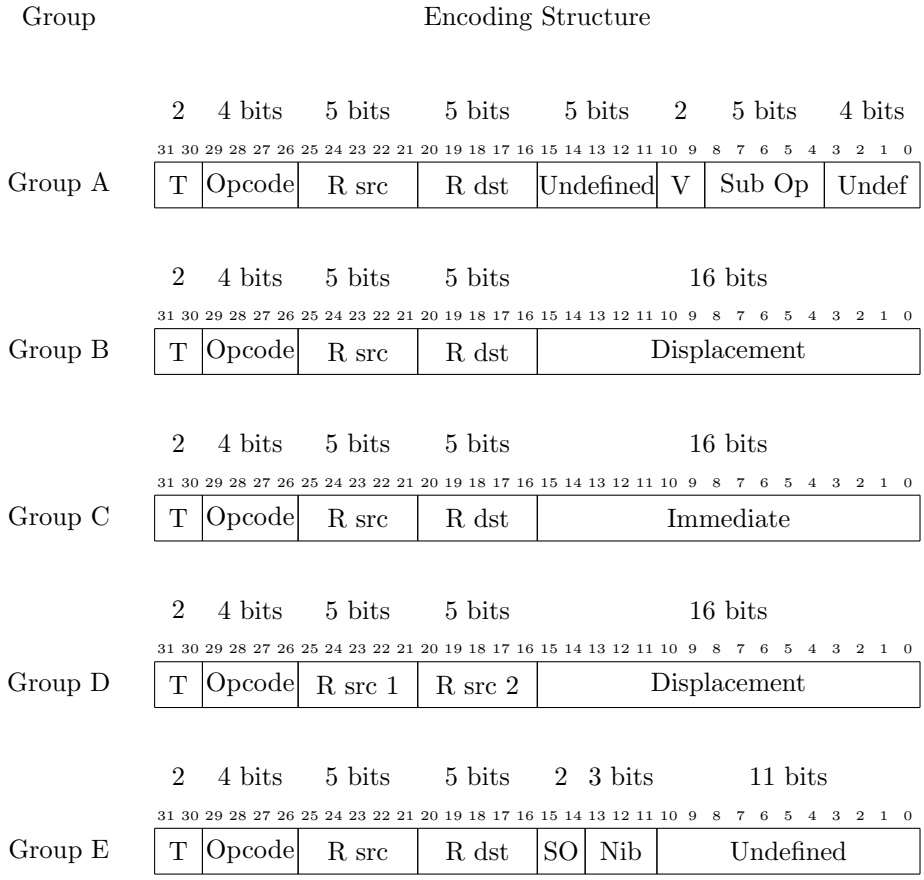

Figure 2.3: Type 2 instruction encoding structure

Type 2 instructions

Type 2 instructions are encoded with four bits (bits 26 to 29) and make use of two registers, one being always source and the other one either source or destination of the operation. This type of instructions are further divided in five groups: A, B, C, D and E. Group A uses a source register (bits 21 to 25), a destination register (bits 16 to 20), a 5 bits subopcode (bits 4 to 8), a 2 bits variation specifier (bits 9 and 10), leaving 9 bits unespecified (bits 0 to 3 and 11 to 15). Group B uses a source register (bits 21 to 25), a destination register (bits 16 to 20) and a 16 bits displacement field (bits 0 to 15). Group C uses a source register (bits 21 to 25), a destination register (bits 16 to 20) and a 16 bits immediate field (bits 0 to 15). Group D uses two source registers (bits 21 to 25 and 16 to 20) and a 16 bits displacement field (bits 0 to 15). Group E uses a source register (bits 21 to 25), a destination register (bits 16 to 20), a two bits subopcode field (bits 14 and 15), a three bits data field (bits 11 to 13), leaving 11 bits unespecified (bits 0 to 10). The encoding structure is shown in figure 2.3.

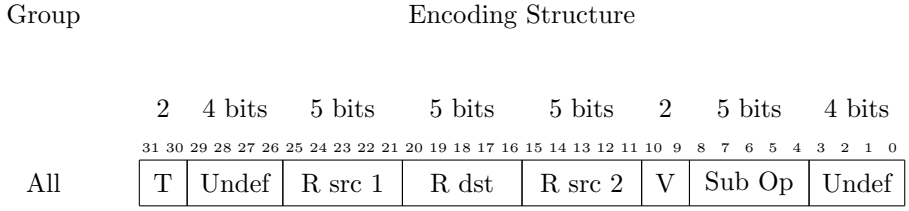

Figure 2.4: Type 3 instruction encoding structure

Mnemonic	Description
`j LABEL`	Jump
`trap`	Trap
`rfe`	Return from exception

Type 3 instructions

Type 3 instructions make use of three registers, two sources and one destination. This type of instructions are not further divided in groups. All three registers are encoded with five bits each (bits 21 to 25 and bits 11 to 15 for source registers and bits 16 to 20 for destination register), a 5 bits subopcode field (bits 4 to 8), a 2 bits variation specifier (bits 9 and 10), leaving 8 bits undefined (bits 0 to 3 and 26 to 29). The encoding structure is shown in figure 2.4.

2.1.2 Full instruction set

Arranged by type

This sections tabulates instructions organized by type.

- Type 0 instructions are summarized in table **??**.
- Type 1 instructions are summarized in table **??**.
- Type 2 instructions are summarized in table **??**.
- Type 3 instructions are summarized in table **??**.

Arranged by function

This section further tabulates instructions according to their functional behavior, listed as follows:

- Register data transfer are summarized in table 2.1.
- Memory data transfer are summarized in table 2.2.

Mnemonic		Description
`mov.w`	`rdst,rsrc`	Move GPR into GPR
`mov.l`	`rdst,rsrc`	Move long GPR into long GPR
`mov.f`	`fdst,fsrc`	Move FPR into FPR
`mov.d`	`fdst,fsrc`	Move double FPR into double FPR
`movf2w`	`rdst,fsrc`	Move FPR into GPR
`movw2f`	`fdst,rsrc`	Move GPR into FPR
`movd2l`	`rdst,fsrc`	Move double FPR into long GPR
`movl2d`	`fdst,rsrc`	Move long GPR into double FPR
`movw2s`	`sdst,rsrc`	Move GPR into SPR
`movs2w`	`rdst,ssrc`	Move SPR into GPR
`movn2fg.g`	`rdst,rsrc,NIB`	Move nibble to GPR flags
`movfg2n.g`	`rdst,rsrc,NIB`	Move GPR flags to nibble
`movn2fg.f`	`fdst,rsrc,NIB`	Move nibble to FPR flags
`movfg2n.f`	`rdst,fsrc,NIB`	Move FPR flags to nibble

Table 2.1: Register data transfer instructions

- Arithmetic are summarized in table 2.4.
- Cordic are summarized in table 2.5.
- Logic are summarized in table 2.6.
- Control are summarized in table 2.7.
- Data are summarized type conversion in table 2.8.

Table 2.5: Cordic instructions

Mnemonic		Description
`cos.w`	`rdst,rsrc`	Cosine of GPR
`cos.l`	`rdst,rsrc`	Cosine of long GPR
`cos.f`	`fdst,fsrc`	Cosine of FPR
`cos.d`	`fdst,fsrc`	Cosine of double FPR
`cosh.w`	`rdst,rsrc`	Hyperbolic cosine of GPR
`cosh.l`	`rdst,rsrc`	Hyperbolic cosine of long GPR
`cosh.f`	`fdst,fsrc`	Hyperbolic cosine of FPR
`cosh.d`	`fdst,fsrc`	Hyperbolic cosine of double FPR
`sin.w`	`rdst,rsrc`	Sine of GPR
`sin.l`	`rdst,rsrc`	Sine of long GPR
`sin.f`	`fdst,fsrc`	Sine of FPR
`sin.d`	`fdst,fsrc`	Sine of double FPR

Continued on next page

– Continued from previous page

Mnemonic		Description
`sinh.w`	`rdst,rsrc`	Hyperbolic sine of GPR
`sinh.l`	`rdst,rsrc`	Hyperbolic sine of long GPR
`sinh.f`	`fdst,fsrc`	Hyperbolic sine of FPR
`sinh.d`	`fdst,fsrc`	Hyperbolic sine of double FPR
`sqrt1msq.w`	`rdst,rsrc`	Square root 1 minus square of GPR
`sqrt1msq.l`	`rdst,rsrc`	Square root 1 minus square of long GPR
`sqrt1msq.f`	`fdst,fsrc`	Square root 1 minus square of FPR
`sqrt1msq.d`	`fdst,fsrc`	Square root 1 minus square of double FPR
`sqrt1psq.w`	`rdst,rsrc`	Square root 1 plus GPR^2
`sqrt1psq.l`	`rdst,rsrc`	Square root 1 plus long GPR^2
`sqrt1psq.f`	`fdst,fsrc`	Square root 1 plus FPR^2
`sqrt1psq.d`	`fdst,fsrc`	Square root 1 plus double FPR^2
`atan.w`	`rdst,rsrc`	Arctangent of GPR
`atan.l`	`rdst,rsrc`	Arctangent of long GPR
`atan.f`	`fdst,fsrc`	Arctangent of FPR
`atan.d`	`fdst,fsrc`	Arctangent of double FPR
`atanh.w`	`rdst,rsrc`	Hyperbolic arctangent of GPR
`atanh.l`	`rdst,rsrc`	Hyperbolic arctangent of long GPR
`atanh.f`	`fdst,fsrc`	Hyperbolic arctangent of FPR
`atanh.d`	`fdst,fsrc`	Hyperbolic arctangent of double FPR
`ln.w`	`rdst,rsrc`	Natural logarithm of GPR
`ln.l`	`rdst,rsrc`	Natural logarithm of long GPR
`ln.f`	`fdst,fsrc`	Natural logarithm of FPR
`ln.d`	`fdst,fsrc`	Natural logarithm of double FPR
`sqrt.w`	`rdst,rsrc`	Square root of GPR
`sqrt.l`	`rdst,rsrc`	Square root of long GPR
`sqrt.f`	`fdst,fsrc`	Square root of FPR
`sqrt.d`	`fdst,fsrc`	Square root of double FPR
`pol2recx.w`	`rdst,rsrc1,rsrc2`	Polar to rectangular x of GPR's
`pol2recx.l`	`rdst,rsrc1,rsrc2`	Polar to rectangular x of long GPR's
`pol2recx.f`	`fdst,fsrc1,fsrc2`	Polar to rectangular x of FPR's
`pol2recx.d`	`fdst,fsrc1,fsrc2`	Polar to rectangular x of double FPR's
`pol2recy.w`	`rdst,rsrc1,rsrc2`	Polar to rectangular src of GPR's
`pol2recy.l`	`rdst,rsrc1,rsrc2`	Polar to rectangular src of long GPR's
`pol2recy.f`	`fdst,fsrc1,fsrc2`	Polar to rectangular src of FPR's
`pol2recy.d`	`fdst,fsrc1,fsrc2`	Polar to rectangular src of double FPR's
`hyp2recx.w`	`rdst,rsrc1,rsrc2`	Hyperbolic to rectangular x of GPR's
`hyp2recx.l`	`rdst,rsrc1,rsrc2`	Hyperbolic to rectangular x of long GPR's
`hyp2recx.f`	`fdst,fsrc1,fsrc2`	Hyperbolic to rectangular x of FPR's

Continued on next page

– Continued from previous page		
Mnemonic		Description
`hyp2recx.d`	`fdst,fsrc1,fsrc2`	Hyperbolic to rectangular x of double FPR's
`hyp2recy.w`	`rdst,rsrc1,rsrc2`	Hyperbolic to rectangular src of GPR's
`hyp2recy.l`	`rdst,rsrc1,rsrc2`	Hyperbolic to rectangular src of long GPR's
`hyp2recy.f`	`fdst,fsrc1,fsrc2`	Hyperbolic to rectangular src of FPR's
`hyp2recy.d`	`fdst,fsrc1,fsrc2`	Hyperbolic to rectangular src of double FPR's
`norm.w`	`rdst,rsrc1,rsrc2`	Norm of GPR's
`norm.l`	`rdst,rsrc1,rsrc2`	Norm of long GPR's
`norm.f`	`fdst,fsrc1,fsrc2`	Norm of FPR's
`norm.d`	`fdst,fsrc1,fsrc2`	Norm of double FPR's
`atan2.w`	`rdst,rsrc1,rsrc2`	Arctangent of GPR's
`atan2.l`	`rdst,rsrc1,rsrc2`	Arctangent of long GPR's
`atan2.f`	`fdst,fsrc1,fsrc2`	Arctangent of FPR's
`atan2.d`	`fdst,fsrc1,fsrc2`	Arctangent of double FPR's

Conventions for semantic descriptions

The following sections gives the detailed specification of the instructions set. For the semantic descrition a high level formal language is adopted. It is worth to mention that as any high level language, there are more than one way to express the same operation. Since that, the instruction descriptions are arbitrary.

Predefined variables

They are used to refer parts of the procesor.

- *pc*: Program counter, stores 32 bits unsigned integers. It is incremented in 4 positions since it indexes instruction words of 32 bits width. It means that two consecutive instructions in the program memory are in the addresses pc and pc + 4. The pc content must be always a multiple of 4.
- $m[]$: Array that represents the data memory. It is addressed by bytes (8 bits) but it can returns bytes (8 bits), halfs (16 bits) and words (32 bits). As mentioned in section 1.4.1, halfs and words must be aligned.
- $im[]$: Array that represents the instruction memory. It is addressed by words (32 bits).
- $alu_x.c$, $alu_x.v$, $alu_x.z$, $alu_x.n$: Repersent ALU operation output flags ($x = 0$ is for low GPR and $x = 1$ is for high GPR).
- r_x, r_y, r_z: Represent any of the 32 general purpose integer registers R0, ..., R31.
- $r_x.c$, $r_x.v$, $r_x.z$, $r_x.n$: Represent the associated flags to register r_x.

Mnemonic		Description
`loadi`	`rdst,IMM`	Load immediate
`loadui`	`rdst,IMM`	Load unsigned immediate
`lhi`	`rdst,IMM`	Load high immediate
`load.b`	`rdst,DIS(rsrc)`	Load byte
`loadu.b`	`rdst,DIS(rsrc)`	Load unsigned byte
`load.h`	`rdst,DIS(rsrc)`	Load half word
`loadu.h`	`rdst,DIS(rsrc)`	Load unsigned half word
`load.w`	`rdst,DIS(rsrc)`	Load word
`load.f`	`fdst,DIS(rsrc)`	Load float
`store.b`	`rsrc1,DIS(rsrc2)`	Store byte
`store.h`	`rsrc1,DIS(rsrc2)`	Sotre half word
`store.w`	`rsrc1,DIS(rsrc2)`	Store word
`store.f`	`fsrc,DIS(rsrc)`	Store float

Table 2.2: Memory data transfer instructions

- $fpu_x.a$, $fpu_x.i$, $fpu_x.z$, $fpu_x.n$: Repersent FPU operation output flags ($x = 0$ is for low FPR and $x = 1$ is for high FPR).
- f_x, f_y, f_z: Represent any of the 32 floating point registers F0, ..., F31.
- $f_x.a$, $f_x.i$, $f_x.z$, $f_x.n$: Represent the associated flags to register f_x.
- imm_n, $disp_n$, nib_n: Represent constants included in instructions. "n" indicates the bit width of the constants.
- $cnv_x.c$, $cnv_x.v$, $cnv_x.z$, $cnv_x.n$, $cnv_x.a$, $cnv_x.i$: Represent CU operation output flags ($x = 0$ is for low GPR or FPR and $x = 1$ is for high GPR or FPR).
- $cordic_x.c$, $cordic_x.v$, $cordic_x.z$, $cordic_x.n$, $cordic_x.a$, $cordic_x.i$: Represent Cordic operation output flags ($x = 0$ is for low GPR or FPR and $x = 1$ is for high GPR or FPR).

Numbers

When the description of the instruction implies the use of a number, it can be expressed in the proper radix in order to clarify the notation. The convention adopted is like in C programming language. The native type is radix 10. Radix 16 is noted by the prefix 0x and binary numbers by 0b.

Literals

Logic values are indicated by the symbols '0' and '1'.

Operators and expressions

- Assignments: The used symbol is "$\longleftarrow_n$", where n represents the number of bits of the assigment.

Mnemonic		Description
addi.w	rdst,rsrc,IMM	Add immediate
addui.w	rdst,rsrc,IMM	Add unsigned immediate
subi.w	rdst,rsrc,IMM	Substract immediate
subui.w	rdst,rsrc,IMM	Substract unsigned immediate
add.w	rdst,rsrc1,rsrc2	Add GPR's
add.l	rdst,rsrc1,rsrc2	Add long GPR's
add.f	fdst,fsrc1,fsrc2	Add FPR's
add.d	fdst,fsrc1,fsrc2	Add double FPR's
sub.w	rdst,rsrc1,rsrc2	Substract GPR's
sub.l	rdst,rsrc1,rsrc2	Substract long GPR's
sub.f	fdst,fsrc1,fsrc2	Substract FPR's
sub.d	fdst,fsrc1,fsrc2	Substract double FPR's
mult.w	rdst,rsrc1,rsrc2	Multiply GPR's
mult.l	rdst,rsrc1,rsrc2	Multiply GPR's into long GPR
mult.f	fdst,fsrc1,fsrc2	Multiply FPR's
mult.d	fdst,fsrc1,fsrc2	Multiply double FPR's
div.w	rdst,rsrc1,rsrc2	Divide GPR's
div.l	rdst,rsrc1,rsrc2	Divide long GPR's
div.f	fdst,fsrc1,fsrc2	Divide FPR's
div.d	fdst,fsrc1,fsrc2	Divide double FPR's
rem.w	rdst,rsrc1,rsrc2	Remainder of GPR's
rem.l	rdst,rsrc1,rsrc2	Remainder of long GPR's
rl.w	rdst,rsrc1,rsrc2	Rotate left GPR
rl.l	rdst,rsrc1,rsrc2	Rotate left long GPR
rr.w	rdst,rsrc1,rsrc2	Rotate right GPR
rr.l	rdst,rsrc1,rsrc2	Rotate right long GPR
sl.w	rdst,rsrc1,rsrc2	Shift left GPR
sl.l	rdst,rsrc1,rsrc2	Shift left long GPR
sra.w	rdst,rsrc1,rsrc2	Shift right arithmetical GPR
sra.l	rdst,rsrc1,rsrc2	Shift right arithmetical long GPR

Table 2.3: Arithmetic instructions

Mnemonic		Description
addi.w	rdst,rsrc,IMM	Add immediate
addui.w	rdst,rsrc,IMM	Add unsigned immediate
subi.w	rdst,rsrc,IMM	Substract immediate
subui.w	rdst,rsrc,IMM	Substract unsigned immediate
add.w	rdst,rsrc1,rsrc2	Add GPR's
add.l	rdst,rsrc1,rsrc2	Add long GPR's
add.f	fdst,fsrc1,fsrc2	Add FPR's
add.d	fdst,fsrc1,fsrc2	Add double FPR's
sub.w	rdst,rsrc1,rsrc2	Substract GPR's
sub.l	rdst,rsrc1,rsrc2	Substract long GPR's
sub.f	fdst,fsrc1,fsrc2	Substract FPR's
sub.d	fdst,fsrc1,fsrc2	Substract double FPR's
mult.w	rdst,rsrc1,rsrc2	Multiply GPR's
mult.l	rdst,rsrc1,rsrc2	Multiply GPR's into long GPR
mult.f	fdst,fsrc1,fsrc2	Multiply FPR's
mult.d	fdst,fsrc1,fsrc2	Multiply double FPR's
div.w	rdst,rsrc1,rsrc2	Divide GPR's
div.l	rdst,rsrc1,rsrc2	Divide long GPR's
div.f	fdst,fsrc1,fsrc2	Divide FPR's
div.d	fdst,fsrc1,fsrc2	Divide double FPR's
rem.w	rdst,rsrc1,rsrc2	Remainder of GPR's
rem.l	rdst,rsrc1,rsrc2	Remainder of long GPR's
rl.w	rdst,rsrc1,rsrc2	Rotate left GPR
rl.l	rdst,rsrc1,rsrc2	Rotate left long GPR
rr.w	rdst,rsrc1,rsrc2	Rotate right GPR
rr.l	rdst,rsrc1,rsrc2	Rotate right long GPR
sl.w	rdst,rsrc1,rsrc2	Shift left GPR
sl.l	rdst,rsrc1,rsrc2	Shift left long GPR
sra.w	rdst,rsrc1,rsrc2	Shift right arithmetical GPR
sra.l	rdst,rsrc1,rsrc2	Shift right arithmetical long GPR

Table 2.4: Arithmetic instructions

Mnemonic		Description
inv.w	rdst,rsrc	Invert GPR
inv.l	rdst,rsrc	Invert long GPR
and.w	rdst,rsrc1,rsrc2	And GPR's
and.l	rdst,rsrc1,rsrc2	And long GPR's
or.w	rdst,rsrc1,rsrc2	Or GPR's
or.l	rdst,rsrc1,rsrc2	Or long GPR's
xor.w	rdst,rsrc1,rsrc2	Xor GPR's
xor.l	rdst,rsrc1,rsrc2	Xor long GPR's
srl.w	rdst,rsrc1,rsrc2	Shift right logical GPR
srl.l	rdst,rsrc1,rsrc2	Shift right logical long GPR

Table 2.6: Logic instructions

Mnemonic		Description
`j`	`LABEL`	Jump
`jr`	`rsrc`	Jump register
`jal`	`rsrc,DIS`	Jump and link
`jral`	`rdst,rsrc`	Jump register and link
`trap`		Trap
`rfe`		Return from exception
`bc`	`rsrc,DIS`	Branch if carry GPR
`bv`	`rsrc,DIS`	Branch if overflow GPR
`bz`	`rsrc,DIS`	Branch if zero GPR
`bn`	`rsrc,DIS`	Branch if negative GPR
`bnc`	`rsrc,DIS`	Branch if not carry GPR
`bnv`	`rsrc,DIS`	Branch if not overflow GPR
`bnz`	`rsrc,DIS`	Branch if not zero GPR
`bnn`	`rsrc,DIS`	Branch if not negative GPR
`bfpan`	`fsrc,DIS`	Branch if is a number FPR
`bfpinf`	`fsrc,DIS`	Branch if is infinite FPR
`bfpz`	`fsrc,DIS`	Branch if zero FPR
`bfpn`	`fsrc,DIS`	Branch if negative FPR
`bfpnan`	`fsrc,DIS`	Branch if is not a number FPR
`bfpninf`	`fsrc,DIS`	Branch if is not infinite FPR
`bfpnz`	`fsrc,DIS`	Branch if not zero FPR
`bfpnn`	`fsrc,DIS`	Branch if negative FPR

Table 2.7: Control instructions

Mnemonic		Description
`cnvf2d`	`fdst,fsrc`	Convert FPR to double FPR
`cnvd2f`	`fdst,fsrc`	Convert double FPR to FPR
`cnvf2w`	`rdst,fsrc`	Convert FPR to GPR
`cnvw2f`	`fdst,rsrc`	Convert GPR to FPR
`cnvd2w`	`rdst,fsrc`	Convert double FPR to GPR
`cnvw2d`	`fdst,rsrc`	Convert GPR to double FPR
`cnvf2l`	`rdst,fsrc`	Convert FPR to long GPR
`cnvl2f`	`fdst,rsrc`	Convert long FPR to FPR
`cnvd2l`	`rdst,fsrc`	Convert double FPR to long GPR
`cnvl2d`	`fdst,rsrc`	Convert long GPR to double FPR

Table 2.8: Data type conversion instructions

- Arithmetic and logic operators: They can be used for representing any operation between predefined variables and GPR's / FPR's. Addition ("+"), subtraction ("−"), multiplication ("∗"), division ("/"), remainder ("%"), logic AND ("&"), logic OR ("|"), logic XOR ("∧"), logic inversion ("∼"), sine ("*sin*"), cosine ("*cos*"), hyperbolic sine ("*sinh*"), hyperbolic cosine ("*cosh*"), arctangent ("*atan*"), natural logarithm ("*ln*"), square root ("*sqrt*"), A? ("*a*?"), B? ("*b*?").

- Comparison: They return "true" or "false" to be used in flow control statements like "if". Equal than ("=="), not equal than("! ="), less than ("<"), less or equal than ("<="), greater than (">"), greater or equal than (">=").

- Logic concatenation: It permits to concatenate any expression of the same type. It is noted by "#".

- Range of bits: V[x:y] represents a range of bits of the variable V from the position x to the position y.

- Particular bit: V[x] represents a particular bit of the variable V in the position x.

- Logic extension: $x_n(V)$ extends the variable V with left side zeros to fit a n bits size.

- Signed extension: $sx_n(V)$ extends the variable V with its most significant bit to fit a n bits size.

- Unsigned casting: $U_n(V)$, the variable V is as an n bits unsigned integer.

- Logic replication: $n(V)$, the variable V is replicated by n at bit level.

- Decision statement: the only one defined statement is "if (<logic_condition>) then <expressions> [else < *e*xpressions> endif]".

2.1.3 Source and destination operands

As a general overview, the operands have the functions described in table 2.9.

2.2 Full instruction set description

This section will give a full description of all the instructions opcodes and their behavior and manipulation of architecture resources. The listing will be ordered by type, subtype and group. As a first step, all executed instructions load it's opcode to the IR. This is noted as $ir \longleftarrow_{30} im[pc]$, and this expression should be prepended to all operations noted within the following description.

Type	Subtype	Group	Operation
0	0	0	$pc \longleftarrow pc + disp$
		1	No arguments
		2	No arguments
1	0	0	$pc \longleftarrow r_x$
		1	$pc \longleftarrow pc + displ$
	1	0	$r_x \longleftarrow imm$
		1	$r_x \longleftarrow pc + 1$
2	0	0	$r_x \longleftarrow r_y$
		1	$r_x \longleftarrow r_y$
		2	$r_x \longleftarrow r_y$
		3	$r_x \longleftarrow r_y \dagger imm$
		4	$r_x \longleftarrow m[r_y + dis]$
		5	$r_x \longleftarrow pc; pc \longleftarrow r_y$
	1	0	$mem[r_y + dis] \longleftarrow r_x$
3	0	0	$r_x \longleftarrow r_y \dagger r_z$

Note: "†" represents any arithmetic or logical operation.

Table 2.9: Operands functions in instructions

j - Jump

Mnemonic	:	`j LABEL`
Coding	:	`00 00 ddddddddddddddddddddddddddddd`
Operation	:	$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{26})$
Assember Validations	:	`DIS range: -33554432:33554431`
CPU Validations	:	`dst range: 0:31`
Affected flags	:	None
Notes	:	Label is translated to displacement from PC.

trap - Trap

Mnemonic	:	`trap`
Coding	:	`00 01 uuuuuuuuuuuuuuuuuuuuuuuuuuuuu`
Operation	:	$cr[31] \longleftarrow_1 1$
Assember Validations	:	`None.`
CPU Validations	:	`None.`
Affected flags	:	$cr[31]$
Notes	:	Trap will be detected in next fetch step. Being the lowest priority exception cannot be used inside a handler.

rfe - Return from exception

Mnemonic	:	`rfe`
Coding	:	`00 10 uuuuuuuuuuuuuuuuuuuuuuuuuuuuuu`
Operation	:	$esp \longleftarrow_5 esp - 1$ $pc \longleftarrow_{30} l_{esp}[29:0]$ $cfg[0] \longleftarrow_1 0$
Assember Validations	:	`None.`
CPU Validations	:	`None.`
Affected flags	:	$cfg[0]$
Notes	:	None.

jr - Jump register

Mnemonic	:	`jr rsrc`
Coding	:	`01 0000 r_src ddddddddddddddddddddd`
Operation	:	$pc \longleftarrow_{30} r_{src}[31:2]$
Assember Validations	:	`None.`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	The two LSB's of r_{src} are discarded. These bits should be zero, but no checking is done.

jal - Jump and link

Mnemonic	:	`jal rsrc,DIS`
Coding	:	`01 0001 uuuuu r_dst ddddddddddddddddd`
Operation	:	$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis21)$ $r_{src} \longleftarrow_{32} (pc+1)\#0b00$
Assember Validations	:	`None.`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bc - Branch if carry GPR

Mnemonic	:	`bc rsrc,DIS`
Coding	:	`01 0010 r_src 0000 ddddddddddddddddd`

Operation	:	if $(r_{src}.c ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bv - Branch if overflow GPR

Mnemonic	:	`bv rsrc,DIS`
Coding	:	`01 0010 r_src 0001 dddddddddddddddd`

Operation	:	if $(r_{src}.v ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bz - Branch if zero GPR

Mnemonic	:	`bz rsrc,DIS`
Coding	:	`01 0010 r_src 0010 dddddddddddddddd`

Operation	:	if $(r_{src}.z ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bn - Branch if negative GPR

Mnemonic	:	`bn rsrc,DIS`
Coding	:	`01 0010 r_src 0011 dddddddddddddddd`

Operation	:	if $(r_{src}.n ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bnc - Branch if not carry GPR

Mnemonic	:	`bnc rsrc,DIS`
Coding	:	`01 0010 r_src 0100 dddddddddddddddd`

Operation	:	if $(r_{src}.c ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bnv - Branch if not overflow GPR

Mnemonic	:	`bnv rsrc,DIS`
Coding	:	`01 0010 r_src 0101 ddddddddddddddddd`

Operation	:	if $(r_{src}.v ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bnz - Branch if not zero GPR

Mnemonic	:	`bnz rsrc,DIS`
Coding	:	`01 0010 r_src 0110 ddddddddddddddddd`

Operation	:	if $(r_{src}.z ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bnn - Branch if not negative GPR

Mnemonic	:	`bnn rsrc,DIS`
Coding	:	`01 0010 r_src 0111 ddddddddddddddddd`

Operation	:	if $(r_{src}.n ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpan - Branch if is a number FPR

Mnemonic	:	`bfpan fsrc,DIS`
Coding	:	`01 0010 r_src 1000 ddddddddddddddddd`

Operation	:	if $(f_{src}.a ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpinf - Branch if is infinite FPR

Mnemonic	:	`bfpinf fsrc,DIS`
Coding	:	`01 0010 r_src 1001 ddddddddddddddddd`

Operation	:	if $(f_{src}.i ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpz - Branch if zero FPR

Mnemonic	:	`bfpz fsrc,DIS`
Coding	:	`01 0010 r_src 1010 ddddddddddddddddd`

Operation	:	if $(f_{src}.z ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpn - Branch if negative FPR

Mnemonic	:	`bfpn fsrc,DIS`
Coding	:	`01 0010 r_src 1011 ddddddddddddddddd`

Operation	:	if $(f_{src}.n ==' 1')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpnan - Branch if is not a number FPR

Mnemonic	:	`bfpnan fsrc,DIS`
Coding	:	`01 0010 r_src 1100 ddddddddddddddddd`

Operation	:	if $(f_{src}.a ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpninf - Branch if is not infinite FPR

Mnemonic	:	`bfpninf fsrc,DIS`
Coding	:	`01 0010 r_src 1101 ddddddddddddddddd`

Operation	:	if $(f_{src}.i ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpnz - Branch if not zero FPR

Mnemonic	:	`bfpnz fsrc,DIS`
Coding	:	`01 0010 r_src 1110 ddddddddddddddddd`

Operation	:	if $(f_{src}.z ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

bfpnn - Branch if negative FPR

Mnemonic	:	`bfpnn fsrc,DIS`
Coding	:	`01 0010 r_src 1111 ddddddddddddddddd`

Operation	:	if $(f_{src}.n ==' 0')$ then
		$pc \longleftarrow_{30} pc + 1 + sx_{30}(dis_{16})$
		endif

Assember Validations	:	`DIS range: -65536:65535`
CPU Validations	:	`src range: 0:31`
Affected flags	:	None
Notes	:	None.

loadi - Load immediate

Mnemonic	:	`loadi rdst,IMM`
Coding	:	`01 0100 uuuuu r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst} \longleftarrow_{32} sx_{32}(imm_{16}[15:0])$ $r_{dst}.n \longleftarrow_{1} r_{dst}[31]$
Assember Validations	:	`IMM range: -32768:32767`
CPU Validations	:	`dst range: 0:31`
Affected flags	:	$r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

loadui - Load unsigned immediate

Mnemonic	:	`loadui rdst,IMM`
Coding	:	`01 0101 uuuuu r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst} \longleftarrow_{32} x_{32}(imm_{16}[15:0])$ $r_{dst}.n \longleftarrow_{1} r_{dst}[31]$
Assember Validations	:	`IMM range: 0:65536`
CPU Validations	:	`dst range: 0:31`
Affected flags	:	$r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

lhi - Load high immediate

Mnemonic	:	`lhi rdst,IMM`
Coding	:	`01 0110 uuuuu r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst}[31:16] \longleftarrow_{16} imm_{16}[15:0]$ $r_{dst}.n \longleftarrow_{1} r_{dst}[31]$
Assember Validations	:	`IMM range: -32768:32767`
CPU Validations	:	`dst range: 0:31`
Affected flags	:	$r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

mov.w - Move GPR into GPR

Mnemonic	:	`mov.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00000 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src}$ $(r_{dst}.c \# r_{dst}.v \# r_{dst}.z \# r_{dst}.n) \longleftarrow_{4} (r_{src}.c \# r_{src}.v \# r_{src}.z \# r_{src}.n)$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

mov.l - Move long GPR into long GPR

Mnemonic	:	`mov.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00000 uuuuu`

Operation	:	if (dst % 2 ==′ 0′ & src % 2 ==′ 0′) then
		(r_{dst+1}#r_{dst}) ⟵$_{64}$ (r_{src+1}#r_{src})
		($r_{dst}.c$#$r_{dst}.v$#$r_{dst}.z$#$r_{dst}.n$) ⟵$_4$ ($r_{src}.c$#$r_{src}.v$#$r_{src}.z$#$r_{src}.n$)
		($r_{dst+1}.c$#$r_{dst+1}.v$#$r_{dst+1}.z$#$r_{dst+1}.n$) ⟵$_4$ ($r_{src+1}.c$#$r_{src+1}.v$#$r_{src+1}.z$#$r_{src+1}.n$)
		else
		cr[4] ⟵$_1$ 1
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if (dst%2 ==′ 0′ & src % 2 ==′ 0′) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.z$
		else
		cr[4]
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

mov.f - Move FPR into FPR

Mnemonic	:	`mov.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 00000 uuuuu`
Operation	:	f_{dst} ⟵$_{32}$ f_{src} ($f_{dst}.a$#$f_{dst}.i$#$f_{dst}.z$#$f_{dst}.n$) ⟵$_4$ ($f_{src}.a$#$f_{src}.i$#$f_{src}.z$#$f_{src}.n$)
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

mov.d - Move double FPR into double FPR

Mnemonic	:	`mov.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 00000 uuuuu`

Operation	:	if (dst % 2 ==′ 0′ & src % 2 ==′ 0′) then
		$f_{dst+1}\#f_{dst}) \longleftarrow_{64} (f_{src+1}\#f_{src})$
		$(f_{dst}.a\#f_{dst}.i\#f_{dst}.z\#f_{dst}.n) \longleftarrow_{4} (f_{src}.a\#f_{src}.i\#f_{src}.z\#f_{src}.n)$
		$(f_{dst+1}.a\#f_{dst+1}.i\#f_{dst+1}.z\#f_{dst+1}.n) \longleftarrow_{4} (f_{src+1}.a\#f_{src+1}.i\#f_{src+1}.z\#f_{src+1}$
		else
		$[5] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if (dst%2 ==′ 0′ & src % 2 ==′ 0′) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, ($f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$)
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

movf2w - Move FPR into GPR

Mnemonic	:	`movf2w rdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00001 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} f_{src}$ $(r_{dst}.c\#r_{dst}.v\#r_{dst}.z\#r_{dst}.n) \longleftarrow_{4} (f_{src}.a\#f_{src}.i\#f_{src}.z\#f_{src}.n)$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c, r_{dst}.v, r_{dst}.z, r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

movw2f - Move GPR into FPR

Mnemonic	:	`movw2f fdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00001 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} r_{src}$ $(f_{dst}.a\#f_{dst}.i\#f_{dst}.z\#f_{dst}.n) \longleftarrow_{4} (r_{src}.c\#r_{src}.v\#r_{src}.z\#r_{src}.n)$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

movd2l - Move double FPR into long GPR

Mnemonic	:	`movd2l rdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 00001 uuuuu`

Operation	:	if (dst % 2 ==′ 0′ & src % 2 ==′ 0′) then
		(r_{dst+1}#r_{dst}) ⟵$_{64}$ (f_{src+1}#f_{src})
		($r_{dst}.c$#$r_{dst}.v$#$r_{dst}.z$#$r_{dst}.n$) ⟵$_4$ ($f_{src}.a$#$f_{src}.i$#$f_{src}.z$#$f_{src}.n$)
		($r_{dst+1}.c$#$r_{dst+1}.v$#$r_{dst+1}.z$#$r_{dst+1}.n$) ⟵$_4$ ($f_{src+1}.a$#$f_{src+1}.i$#$f_{src+1}.z$#$f_{src+1}.n$)
		else if (dst%2 ==′ 1′) then
		cr[4] ⟵$_1$ 1
		else
		cr[5] ⟵$_1$ 1
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if (dst%2 ==′ 0′ & src % 2 ==′ 0′) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else if (dst%2 ==′ 1′) then
		cr[4]
		else
		cr[5]
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

movl2d - Move long GPR into double FPR

Mnemonic	:	`movl2d fdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 00001 uuuuu`

Operation	:	if (dst % 2 ==′ 0′ & src % 2 ==′ 0′) then
		(f_{dst+1}#f_{dst}) ⟵$_{64}$ (r_{src+1}#r_{src})
		($f_{dst}.c$#$f_{dst}.v$#$f_{dst}.z$#$f_{dst}.n$) ⟵$_4$ ($r_{src}.a$#$r_{src}.i$#$r_{src}.z$#$r_{src}.n$)
		($f_{dst+1}.c$#$f_{dst+1}.v$#$f_{dst+1}.z$#$f_{dst+1}.n$) ⟵$_4$ ($r_{src+1}.a$#$r_{src+1}.i$#$r_{src+1}.z$#$r_{src+1}.n$)
		else if (dst%2 ==′ 1′) then
		cr[5] ⟵$_1$ 1
		else
		cr[4] ⟵$_1$ 1
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if (dst%2 ==′ 0′ & src % 2 ==′ 0′) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, ($f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$)
		else if (dst%2 ==′ 1′) then
		cr[5]
		else
		cr[4]
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

movw2s - Move GPR into SPR

Mnemonic	:	`movw2s sdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00010 uuuuu`
Operation	:	$s_{dst} \longleftarrow_{32} r_{src}$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:8; src range: 0:31`
Affected flags	:	None
Notes	:	As not all SPR's are exactly 32 bits wide, exceeding bits from `rsrc` are discarded.

movs2w - Move SPR into GPR

Mnemonic	:	`movs2w rdst,ssrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00010 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} s_{src}$ $r_{dst}.c \longleftarrow_{1} 0$ $r_{dst}.v \longleftarrow_{1} 0$ $r_{dst}.z \longleftarrow_{1} r_{dst}[31] \mid r_{dst}[30] \mid \ldots \mid r_{dst}[0]$ $r_{dst}.n \longleftarrow_{1} r_{dst}[31]$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:8`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	As not all SPR's are exactly 32 bits wide, exceeding bits from `rdst` are zeroed.

cnvf2d - Convert FPR to double FPR

Mnemonic	:	`cnvf2d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00011 uuuuu`

Operation	:	if ($dst\%2 ==' 0'$) then
		$(f_{dst+1}\#f_{dst}) \longleftarrow_{64} double(f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cnv_1.a$
		$f_{dst+1}.i \longleftarrow_1 cnv_1.i$
		$f_{dst+1}.z \longleftarrow_1 cnv_1.z$
		$f_{dst+1}.n \longleftarrow_1 cnv_1.n$
		$f_{dst}.a \longleftarrow_1 cnv_0.a$
		$f_{dst}.i \longleftarrow_1 cnv_0.i$
		$f_{dst}.z \longleftarrow_1 cnv_0.z$
		$f_{dst}.n \longleftarrow_1 cnv_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31`

Affected flags	:	if ($dst\%2 ==' 0'$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

cnvd2f - Convert double FPR to FPR

Mnemonic	:	`cnvd2f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00011 uuuuu`

Operation	:	if ($src\ \%\ 2 ==' 0'$) then
		$fx \longleftarrow_{32} float(f_{src+1}\#f_{src})$
		$f_{dst}.a \longleftarrow_1 cnv_0.a$
		$f_{dst}.i \longleftarrow_1 cnv_0.i$
		$f_{dst}.z \longleftarrow_1 cnv_0.z$
		$f_{dst}.n \longleftarrow_1 cnv_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31 and even`

Affected flags	:	if ($dst\ \%\ 2 ==' 0'$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ is implicit.

cnvf2w - Convert FPR to GPR

Mnemonic	:	`cnvf2w rdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 00011 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} integer(f_{src})$ $r_{dst}.c \longleftarrow_{1} cnv_0.c$ $r_{dst}.v \longleftarrow_{1} cnv_0.v$ $r_{dst}.z \longleftarrow_{1} cnv_0.z$ $r_{dst}.n \longleftarrow_{1} cnv_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

cnvw2f - Convert GPR to FPR

Mnemonic	:	`cnvw2f fdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 00011 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} float(r_{src})$ $f_{dst}.a \longleftarrow_{1} cnv_0.a$ $f_{dst}.i \longleftarrow_{1} cnv_0.i$ $f_{dst}.z \longleftarrow_{1} cnv_0.z$ $f_{dst}.n \longleftarrow_{1} cnv_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

cnvd2w - Convert double FPR to GPR

Mnemonic	:	`cnvd2w rdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00100 uuuuu`

Operation	:	if ($src \ \% \ 2 ==' 0'$) then
		$r_{dst} \longleftarrow_{32} integer(f_{src+1} \# f_{src})$
		$r_{dst}.c \longleftarrow_1 cnv_0.c$
		$r_{dst}.v \longleftarrow_1 cnv_0.v$
		$r_{dst}.z \longleftarrow_1 cnv_0.z$
		$r_{dst}.n \longleftarrow_1 cnv_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31 and even`

Affected flags	:	if ($src \ \% \ 2 ==' 0'$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
		else
		$cr[5]$
		endif

Notes	:	$r_{dst}.n$ is implicit.

cnvw2d - Convert GPR to double FPR

Mnemonic	:	`cnvw2d fdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00100 uuuuu`

Operation	:	if ($dst \ \% \ 2 ==' 0'$) then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} double(r_{src})$
		$f_{dst+1}.a \longleftarrow_1 cnv_1.a$
		$f_{dst+1}.i \longleftarrow_1 cnv_1.i$
		$f_{dst+1}.z \longleftarrow_1 cnv_1.z$
		$f_{dst+1}.n \longleftarrow_1 cnv_1.n$
		$f_{dst}.a \longleftarrow_1 cnv_0.a$
		$f_{dst}.i \longleftarrow_1 cnv_0.i$
		$f_{dst}.z \longleftarrow_1 cnv_0.z$
		$f_{dst}.n \longleftarrow_1 cnv_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31`

Affected flags	:	if ($dst \ \% \ 2 ==' 0'$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

cnvf2l - Convert FPR to long GPR

Mnemonic	:	`cnvf2l rdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 00100 uuuuu`

Operation	:	if (dst % 2 ==′ 0′) then
		(r_{dst+1}#r_{dst}) ⟵$_{64}$ $long(r_{src})$
		$r_{dst+1}.c$ ⟵$_1$ $cnv_1.c$
		$r_{dst+1}.v$ ⟵$_1$ $cnv_1.v$
		$r_{dst+1}.z$ ⟵$_1$ $cnv_1.z$
		$r_{dst+1}.n$ ⟵$_1$ $cnv_1.n$
		$r_{dst}.c$ ⟵$_1$ $cnv_0.c$
		$r_{dst}.v$ ⟵$_1$ $cnv_0.v$
		$r_{dst}.z$ ⟵$_1$ $cnv_0.z$
		$r_{dst}.n$ ⟵$_1$ $cnv_0.n$
		else
		$cr[4]$ ⟵$_1$ 1
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and evel; src range: 0:31`

Affected flags	:	if (dst % 2 ==′ 0′) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

cnvl2f - Convert long FPR to FPR

Mnemonic	:	`cnvl2f fdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 00100 uuuuu`

Operation	:	if (src % 2 ==′ 0′) then
		f_{dst} ⟵$_{32}$ $float(r_{src+1}$#$r_{src})$
		$f_{dst}.a$ ⟵$_1$ $cnv_0.a$
		$f_{dst}.i$ ⟵$_1$ $cnv_0.i$
		$f_{dst}.z$ ⟵$_1$ $cnv_0.z$
		$f_{dst}.n$ ⟵$_1$ $cnv_0.n$
		else
		$cr[4]$ ⟵$_1$ 1
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31 and even`

Affected flags	:	if (src % 2 ==′ 0′) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
		else
		$cr[4]$
		endif

Notes	:	$f_{dst}.n$ is implicit.

cnvd2l - Convert double FPR to long GPR

Mnemonic	:	`cnvd2l rdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00101 uuuuu`

Operation	:	if ($dst \ \% \ 2 ==' 0'$ & $src \ \% \ 2 ==' 0'$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} long(f_{src+1} \# f_{src})$
		$r_{dst+1}.c \longleftarrow_1 cnv_1.c$
		$r_{dst+1}.v \longleftarrow_1 cnv_1.v$
		$r_{dst+1}.z \longleftarrow_1 cnv_1.z$
		$r_{dst+1}.n \longleftarrow_1 cnv_1.n$
		$r_{dst}.c \longleftarrow_1 cnv_0.c$
		$r_{dst}.v \longleftarrow_1 cnv_0.v$
		$r_{dst}.z \longleftarrow_1 cnv_0.z$
		$r_{dst}.n \longleftarrow_1 cnv_0.n$
		else if ($dst\%2 ==' 1'$) then
		$cr[4] \longleftarrow_1 1$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 ==' 0'$ & $src \ \% \ 2 ==' 0'$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$) a definir
		else if ($dst\%2 ==' 1'$) then
		$cr[4]$
		else
		$cr[5]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

cnvl2d - Convert long GPR to double FPR

Mnemonic	:	`cnvl2d fdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00101 uuuuu`

Operation	:	if (dst % 2 ==′ 0′ & src % 2 ==′ 0′) then
		(f_{dst+1}#f_{dst}) ⟵$_{64}$ $double(r_{src+1}\#r_{src})$
		$f_{dst+1}.a$ ⟵$_1$ $cnv_1.a$
		$f_{dst+1}.i$ ⟵$_1$ $cnv_1.i$
		$f_{dst+1}.z$ ⟵$_1$ $cnv_1.z$
		$f_{dst+1}.n$ ⟵$_1$ $cnv_1.n$
		$f_{dst}.a$ ⟵$_1$ $cnv_0.a$
		$f_{dst}.i$ ⟵$_1$ $cnv_0.i$
		$f_{dst}.z$ ⟵$_1$ $cnv_0.z$
		$f_{dst}.n$ ⟵$_1$ $cnv_0.n$
		else if (dst%2 ==′ 1′) then
		$cr[5]$ ⟵$_1$ 1
		else
		$cr[4]$ ⟵$_1$ 1
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if (dst % 2 ==′ 0′ & src % 2 ==′ 0′) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$)
		else if (dst%2 ==′ 1′) then
		$cr[5]$
		else
		$cr[4]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

inv.w - Invert GPR

Mnemonic	:	`inv.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00110 uuuuu`
Operation	:	r_{dst} ⟵$_{32}$ r_{src} $r_{dst}.c$ ⟵$_1$ $alu_0.c$ $r_{dst}.v$ ⟵$_1$ $alu_0.v$ $r_{dst}.z$ ⟵$_1$ $alu_0.z$ $r_{dst}.n$ ⟵$_1$ $alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

inv.l - Invert long GPR

Mnemonic	:	`inv.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 00110 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} r_{src+1}\#r_{src}$
		$r_{dst+1}.c \longleftarrow_{1} alu_1.c$
		$r_{dst+1}.v \longleftarrow_{1} alu_1.v$
		$r_{dst+1}.z \longleftarrow_{1} alu_1.z$
		$r_{dst+1}.n \longleftarrow_{1} alu_1.n$
		$r_{dst}.c \longleftarrow_{1} alu_0.c$
		$r_{dst}.v \longleftarrow_{1} alu_0.v$
		$r_{dst}.z \longleftarrow_{1} alu_0.z$
		$r_{dst}.n \longleftarrow_{1} alu_0.n$
		else
		$cr[4] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

jral - Jump register and link

Mnemonic	:	`jral rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 00111 uuuuu`
Operation	:	$pc \longleftarrow_{30} r_{src}[31:2]$ $r_{dst} \longleftarrow_{32} (pc+1)\#0b00$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	None
Notes	:	The two LSB's of r_{src} are discarded. These bits should be zero, but no checking is done.

cos.w - Cosine of GPR

Mnemonic	:	`cos.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10000 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} cos(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

cos.l - Cosine of long GPR

Mnemonic	:	`cos.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10000 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} cos(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

cos.f - Cosine of FPR

Mnemonic	:	`cos.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10000 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} cos(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

cos.d - Cosine of double FPR

Mnemonic	:	`cos.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10000 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} cos(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

cosh.w - Hyperbolic cosine of GPR

Mnemonic	:	`cosh.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10001 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} cosh(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

cosh.l - Hyperbolic cosine of long GPR

Mnemonic	:	`cosh.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10001 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} cosh(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

cosh.f - Hyperbolic cosine of FPR

Mnemonic	:	`cosh.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10001 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} cosh(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

cosh.d - Hyperbolic cosine of double FPR

Mnemonic	:	`cosh.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10001 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} cosh(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

sin.w - Sine of GPR

Mnemonic	:	`sin.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10010 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} sin(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sin.l - Sine of long GPR

Mnemonic	:	`sin.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10010 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} sin(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sin.f - Sine of FPR

Mnemonic	:	`sin.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10010 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} sin(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

sin.d - Sine of double FPR

Mnemonic	:	`sin.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10010 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} sin(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

sinh.w - Hyperbolic sine of GPR

Mnemonic	:	`sinh.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10011 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} sinh(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sinh.l - Hyperbolic sine of long GPR

Mnemonic	:	`sinh.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10011 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} sinh(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sinh.f - Hyperbolic sine of FPR

Mnemonic	:	`sinh.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10011 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} sinh(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

sinh.d - Hyperbolic sine of double FPR

Mnemonic	:	`sinh.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10011 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} sinh(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

sqrt1msq.w - Square root 1 minus square of GPR

Mnemonic	:	`sqrt1msq.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10100 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} sqrt(1 - r_{src}^2)$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sqrt1msq.l - Square root 1 minus square of long GPR

Mnemonic	:	`sqrt1msq.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10100 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} sqrt(1 - (r_{src+1}\#r_{src})^2)$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sqrt1msq.f - Square root 1 minus square of FPR

Mnemonic	:	`sqrt1msq.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10100 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} sqrt(1 - f_{src}^2)$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

sqrt1msq.d - Square root 1 minus square of double FPR

Mnemonic	:	`sqrt1msq.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10100 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} sqrt(1 - (f_{src+1}\#f_{src})^2)$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

sqrt1psq.w - Square root 1 plus GPR^2

Mnemonic	:	`sqrt1psq.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10101 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} sqrt(1 + r_{src}^2)$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	
Notes	:	

sqrt1psq.l - Square root 1 plus long GPR^2

Mnemonic	:	`sqrt1psq.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10101 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} sqrt(1 + (r_{src+1}\#r_{src})^2)$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`
Affected flags	:	
Notes	:	

sqrt1psq.f - Square root 1 plus FPR^2

Mnemonic	:	`sqrt1psq.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10101 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} sqrt(1 + f_{src}^2)$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	
Notes	:	

sqrt1psq.d - Square root 1 plus double FPR^2

Mnemonic	:	`sqrt1psq.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10101 uuuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} sqrt(1 + (f_{src+1}\#f_{src})^2)$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`
Affected flags	:	
Notes	:	

atan.w - Arctangent of GPR

Mnemonic	:	`atan.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10110 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} atan(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

atan.l - Arctangent of long GPR

Mnemonic	:	`atan.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10110 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} atan(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

atan.f - Arctangent of FPR

Mnemonic	:	`atan.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10110 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} atan(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

atan.d - Arctangent of double FPR

Mnemonic	:	`atan.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 11 10110 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} atan(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

atanh.w - Hyperbolic arctangent of GPR

Mnemonic	:	`atanh.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 10111 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} atanh(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

atanh.l - Hyperbolic arctangent of long GPR

Mnemonic	:	`atanh.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 10111 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} atanh(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

atanh.f - Hyperbolic arctangent of FPR

Mnemonic	:	`atanh.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10111 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} atanh(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

atanh.d - Hyperbolic arctangent of double FPR

Mnemonic	:	`atanh.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 10111 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} atanh(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

ln.w - Natural logarithm of GPR

Mnemonic	:	`ln.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 11000 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} ln(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

ln.l - Natural logarithm of long GPR

Mnemonic	:	`ln.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 11000 uuuuu`

Operation	:	if (dst % 2 == 0 & src % 2 == 0) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} ln(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if (dst % 2 == 0 & src % 2 == 0) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

ln.f - Natural logarithm of FPR

Mnemonic	:	`ln.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 11000 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} ln(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

ln.d - Natural logarithm of double FPR

Mnemonic	:	`ln.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 11000 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} ln(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

sqrt.w - Square root of GPR

Mnemonic	:	`sqrt.w rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 00 11001 uuuuu`
Operation	:	$r_{dst} \longleftarrow_{32} sqrt(r_{src})$ $r_{dst}.c \longleftarrow cordic_0.c$ $r_{dst}.v \longleftarrow cordic_0.v$ $r_{dst}.z \longleftarrow cordic_0.z$ $r_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sqrt.l - Square root of long GPR

Mnemonic	:	`sqrt.l rdst,rsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 01 11001 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{src+1}\#r_{dst} \longleftarrow_{64} sqrt(r_{src+1}\#r_{src})$
		$r_{dst+1}.c \longleftarrow_1 cordic_1.c$
		$r_{dst+1}.v \longleftarrow_1 cordic_1.v$
		$r_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$r_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$r_{dst}.c \longleftarrow_1 cordic_0.c$
		$r_{dst}.v \longleftarrow_1 cordic_0.v$
		$r_{dst}.z \longleftarrow_1 cordic_0.z$
		$r_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4]$
		endif

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sqrt.f - Square root of FPR

Mnemonic	:	`sqrt.f fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 11001 uuuuu`
Operation	:	$f_{dst} \longleftarrow_{32} sqrt(f_{src})$ $f_{dst}.a \longleftarrow cordic_0.a$ $f_{dst}.i \longleftarrow cordic_0.i$ $f_{dst}.z \longleftarrow cordic_0.z$ $f_{dst}.n \longleftarrow cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

sqrt.d - Square root of double FPR

Mnemonic	:	`sqrt.d fdst,fsrc`
Coding	:	`10 0000 r_src r_dst uuuuu 10 11001 uuuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{src+1}\#r_{dst} \longleftarrow_{64} sqrt(f_{src+1}\#f_{src})$
		$f_{dst+1}.a \longleftarrow_1 cordic_1.a$
		$f_{dst+1}.i \longleftarrow_1 cordic_1.i$
		$f_{dst+1}.z \longleftarrow_1 cordic_1.z$
		$f_{dst+1}.n \longleftarrow_1 cordic_1.n$
		$f_{dst}.a \longleftarrow_1 cordic_0.a$
		$f_{dst}.i \longleftarrow_1 cordic_0.i$
		$f_{dst}.z \longleftarrow_1 cordic_0.z$
		$f_{dst}.n \longleftarrow_1 cordic_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5]$
		endif

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

addi.w - Add immediate

Mnemonic	:	`addi.w rdst,rsrc,IMM`
Coding	:	`10 0001 r_src r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst}[31:0] \longleftarrow_{32} r_{src}[31:0] + sx_{32}(imm_{16})$
		$r_{dst}.c \longleftarrow_{1} alu_0.c$
		$r_{dst}.v \longleftarrow_{1} alu_0.v$
		$r_{dst}.z \longleftarrow_{1} alu_0.z$
		$r_{dst}.n \longleftarrow_{1} alu_0.n$
Assember Validations	:	`IMM range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

addui.w - Add unsigned immediate

Mnemonic	:	`addui.w rdst,rsrc,IMM`
Coding	:	`10 0010 r_src r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst}[31:0] \longleftarrow_{32} r_{src}[31:0] + x_{32}(imm_{16})$
		$r_{dst}.c \longleftarrow_{1} alu_0.c$
		$r_{dst}.v \longleftarrow_{1} alu_0.v$
		$r_{dst}.z \longleftarrow_{1} alu_0.z$
		$r_{dst}.n \longleftarrow_{1} alu_0.n$
Assember Validations	:	`IMM range: 0:65536`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

subi.w - Substract immediate

Mnemonic	:	`subi.w rdst,rsrc,IMM`
Coding	:	`10 0011 r_src r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst}[31:0] \longleftarrow_{32} r_{src}[31:0] - sx_{32}(imm_{16})$
		$r_{dst}.c \longleftarrow_{1} alu_0.c$
		$r_{dst}.v \longleftarrow_{1} alu_0.v$
		$r_{dst}.z \longleftarrow_{1} alu_0.z$
		$r_{dst}.n \longleftarrow_{1} alu_0.n$
Assember Validations	:	`IMM range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

subui.w - Substract unsigned immediate

Mnemonic	:	`subui.w rdst,rsrc,IMM`
Coding	:	`10 0100 r_src r_dst iiiiiiiiiiiiiiii`
Operation	:	$r_{dst}[31:0] \longleftarrow_{32} r_{src}[31:0]$–$x_{32}(imm_{16})$ $r_{dst}.c \longleftarrow_{1} alu_0.c$ $r_{dst}.v \longleftarrow_{1} alu_0.v$ $r_{dst}.z \longleftarrow_{1} alu_0.z$ $r_{dst}.n \longleftarrow_{1} alu_0.n$
Assember Validations	:	`IMM range: 0:65536`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

load.b - Load byte

Mnemonic	:	`load.b rdst,DIS(rsrc)`
Coding	:	`10 0101 r_src r_dst dddddddddddddddd`
Operation	:	$r_{dst} \longleftarrow_{32} sx_{32}(M[r_{src} + sx_{32}(dis16)][7:0])$
Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

loadu.b - Load unsigned byte

Mnemonic	:	`loadu.b rdst,DIS(rsrc)`
Coding	:	`10 0110 r_src r_dst dddddddddddddddd`
Operation	:	$r_{dst} \longleftarrow_{32} x_{32}(M[r_{src} + sx_{32}(dis16)][7:0])$
Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

load.h - Load half word

Mnemonic	:	`load.h rdst,DIS(rsrc)`
Coding	:	`10 0111 r_src r_dst ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%2 == 0)$ then
		$r_{dst} \longleftarrow_{32} sx_{32}(M[r_{src} + sx_{32}(dis16)][15:0])$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 2`

Affected flags	:	if $(r_{src} + sx_{32}(dis16)\%2 == 0)$ then
		$r_{dst}.n$
		else
		$cr[3]$
		endif

Notes	:	$r_{dst}.n$ is implicit.

loadu.h - Load unsigned half word

Mnemonic	:	`loadu.h rdst,DIS(rsrc)`
Coding	:	`10 1000 r_src r_dst ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%2 == 0)$ then
		$r_{dst} \longleftarrow_{32} x_{32}(M[r_{src} + sx_{32}(dis16)][15:0])$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 2`

Affected flags	:	if $(r_{src} + sx_{32}(dis16)\%2 == 0)$ then
		$r_{dst}.n$
		else
		$cr[3]$
		endif

Notes	:	$r_{dst}.n$ is implicit.

load.w - Load word

Mnemonic	:	`load.w rdst,DIS(rsrc)`
Coding	:	`10 1001 r_src r_dst ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%4 == 0)$ then
		$r_{dst} \longleftarrow_{32} M[r_{src} + sx_{32}(dis16)][31:0]$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 4`

Affected flags	:	if $(r_{src} + sx_{32}(dis16)\%4 == 0)$ then
		$r_{dst}.n$
		else
		$cr[3]$
		endif

Notes	:	$r_{dst}.n$ is implicit.

load.f - Load float

Mnemonic	:	`load.f fdst,DIS(rsrc)`
Coding	:	`10 1010 r_src r_dst ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%4 == 0)$ then
		$f_{dst} \longleftarrow_{32} M[r_{src} + sx_{32}(dis16)][31:0]$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 4`

Affected flags	:	if $(r_{src} + sx_{32}(dis16)\%4 == 0)$ then
		$r_{dst}.n$
		else
		$cr[3]$
		endif

Notes	:	$f_{dst}.n$ is implicit.

store.b - Store byte

Mnemonic	:	`store.b rsrc1,DIS(rsrc2)`
Coding	:	`10 1011 rsrc1 rscr2 ddddddddddddddd`
Operation	:	$M[r_{src} + sx_{32}(dis16)][7:0] \longleftarrow_{8} r_{dst}[7:0]$
Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	None
Notes	:	None.

store.h - Sotre half word

Mnemonic	:	`store.h rsrc1,DIS(rsrc2)`
Coding	:	`10 1100 rsrc1 rscr2 ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%2 == 0$ then
		$M[r_{src} + sx_{32}(dis16)][15:0] \longleftarrow_{16} r_{dst}[15:0]$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 2`

Affected flags	:	if $(r_{src_1} + sx_{32}(dis16)\%2 == 0)$ then
		None
		else
		$cr[3]$
		endif

Notes	:	None.

store.w - Store word

Mnemonic	:	`store.w rsrc1,DIS(rsrc2)`
Coding	:	`10 1101 rsrc1 rscr2 ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%4 == 0$ then
		$M[r_{src} + sx_{32}(dis16)][31:0] \longleftarrow_{32} r_{dst}[31:0]$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 4`

Affected flags	:	if $(r_{src_1} + sx_{32}(dis16)\%4 == 0)$ then
		None
		else
		$cr[3]$
		endif

Notes	:	None.

store.f - Store float

Mnemonic	:	`store.f fsrc,DIS(rsrc)`
Coding	:	`10 1110 rsrc1 rscr2 ddddddddddddddd`

Operation	:	if $(r_{src} + sx_{32}(dis16)\%4 == 0$ then
		$M[r_{src} + sx_{32}(dis16)][31:0] \longleftarrow_{32} f_{dst}[31:0]$
		else
		$cr[3] \longleftarrow_{1} 1$
		endif

Assember Validations	:	`DIS range: -32768:32767`
CPU Validations	:	`dst range: 0:31; src range: 0:31; address aligned to 4`

Affected flags	:	if $(r_{src_1} + sx_{32}(dis16)\%4 == 0)$ then
		None
		else
		$cr[3]$
		endif

Notes	:	None.

movn2fg.g - Move nibble to GPR flags

Mnemonic	:	`movn2fg.g rdst,rsrc,NIB`
Coding	:	`10 1111 r_src r_dst 00 NNN uuuuuuuuuuu`
Operation	:	$r_{dst}.c\#r_{dst}.v\#r_{dst}.z\#r_{dst}.n \longleftarrow_{4} r_{src}[4 * nib3 + 3 : 4 * nib3]$
Assember Validations	:	`NIB range: 0:7`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	None.

movfg2n.g - Move GPR flags to nibble

Mnemonic	:	`movfg2n.g rdst,rsrc,NIB`
Coding	:	`10 1111 r_src r_dst 01 NNN uuuuuuuuuuu`
Operation	:	$r_{dst}[4 * nib3 + 3 : 4 * nib3] \longleftarrow_{4} r_{src}.c\#r_{src}.v\#r_{src}.z\#r_{src}.n$
Assember Validations	:	`NIB range: 0:7`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	None
Notes	:	None.

movn2fg.f - Move nibble to FPR flags

Mnemonic	:	`movn2fg.f fdst,rsrc,NIB`
Coding	:	`10 1111 r_src r_dst 10 NNN uuuuuuuuuuu`
Operation	:	$f_{dst}.a\#f_{dst}.i\#f_{dst}.z\#f_{dst}.n \longleftarrow_{4} r_{src}[4 * nib3 + 3 : 4 * nib3]$
Assember Validations	:	`NIB range: 0:7`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	None.

movfg2n.f - Move FPR flags to nibble

Mnemonic	:	`movfg2n.f rdst,fsrc,NIB`
Coding	:	`10 1111 r_src r_dst 11 NNN uuuuuuuuuuu`
Operation	:	$r_{dst}[4 * nib3 + 3 : 4 * nib3] \longleftarrow_4 f_{src}.a\#f_{src}.i\#f_{src}.z\#f_{src}.n$
Assember Validations	:	`NIB range: 0:7`
CPU Validations	:	`dst range: 0:31; src range: 0:31`
Affected flags	:	None
Notes	:	None.

add.w - Add GPR's

Mnemonic	:	`add.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00000 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} + r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

add.l - Add long GPR's

Mnemonic	:	`add.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00000 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) + (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

add.f - Add FPR's

Mnemonic	:	`add.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 00000 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} + f_{src_2}$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

add.d - Add double FPR's

Mnemonic	:	`add.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 00000 uuuu`

Operation	:	if (dst % 2 == 0 & src % 2 == 0) then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} (f_{src_1+1} \# f_{src_1}) + (f_{src_2+1} \# f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if (dst % 2 == 0 & src % 2 == 0) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

sub.w - Substract GPR's

Mnemonic	:	`sub.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00001 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} - r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sub.l - Substract long GPR's

Mnemonic	:	`sub.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00001 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) - (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sub.f - Substract FPR's

Mnemonic	:	`sub.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 00001 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} - f_{src_2}$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

sub.d - Substract double FPR's

Mnemonic	:	`sub.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 00001 uuuu`

Operation	:	if (dst % 2 == 0 & src % 2 == 0) then
		$(f_{dst+1}\#f_{dst}) \longleftarrow_{64} (f_{src_1+1}\#f_{src_1}) - (f_{src_2+1}\#f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if (dst % 2 == 0 & src % 2 == 0) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

mult.w - Multiply GPR's

Mnemonic	:	`mult.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00010 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} * r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

mult.l - Multiply GPR's into long GPR

Mnemonic	:	`mult.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00010 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) * (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 ; src1 range: 0:31; src2 range: 0:31`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

mult.f - Multiply FPR's

Mnemonic	:	`mult.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 00010 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} * f_{src_2}$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

mult.d - Multiply double FPR's

Mnemonic	:	`mult.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 00010 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} (f_{src_1+1} \# f_{src_1}) * (f_{src_2+1} \# f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

div.w - Divide GPR's

Mnemonic	:	`div.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00011 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} / r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

div.l - Divide long GPR's

Mnemonic	:	`div.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00011 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) / (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

div.f - Divide FPR's

Mnemonic	:	`div.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 00011 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} / f_{src_2}$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

div.d - Divide double FPR's

Mnemonic	:	`div.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 00011 uuuu`

Operation	:	if $(dst\ \%\ 2 == 0\ \&\ src\ \%\ 2 == 0)$ then
		$(f_{dst+1}\#f_{dst}) \longleftarrow_{64} (f_{src_1+1}\#f_{src_1})/(f_{src_2+1}\#f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst\ \%\ 2 == 0\ \&\ src\ \%\ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

rem.w - Remainder of GPR's

Mnemonic	:	`rem.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00100 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1}\%r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

rem.l - Remainder of long GPR's

Mnemonic	:	`rem.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00100 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) \% (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

and.w - And GPR's

Mnemonic	:	`and.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00101 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} \& r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

and.l - And long GPR's

Mnemonic	:	`and.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00101 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) \& (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

or.w - Or GPR's

Mnemonic	:	`or.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00110 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} \vert r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

or.l - Or long GPR's

Mnemonic	:	`or.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00110 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) \mid (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

xor.w - Xor GPR's

Mnemonic	:	`xor.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 00111 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} \hat{} r_{src_2}$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

xor.l - Xor long GPR's

Mnemonic	:	`xor.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 00111 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) \curlyvee (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

rl.w - Rotate left GPR

Mnemonic	:	`rl.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 01000 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} (r_{src_1}[32 - U_5(r_{src_2}[4:0]) : 0] \# r_{src_1}[31 : 32 - U_5(r_{src_2}[4:0])])$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

rl.l - Rotate left long GPR

Mnemonic	:	`rl.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 01000 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} ((r_{src_1+1} \# r_{src_1})[64 - U_6(r_{src_2}[5:0]) : 0] \# (r_{src_1+1} \# r_{src_1})[63 : 64 - U_6(r_{src_2}[5:0])])$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

rr.w - Rotate right GPR

Mnemonic	:	`rr.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 01001 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} (r_{src_1}[U_5(r_{src_2}[4:0]) - 1 : 0] \# r_{src_1}[31 : U_5(r_{src_2}[4:0]) : 0])$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

rr.l - Rotate right long GPR

Mnemonic	:	`rr.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 01001 uuuu`

Operation	:	if (dst % 2 == 0 & src % 2 == 0) then
		(r_{dst+1}#r_{dst}) $\longleftarrow_{64}$ ((r_{src_1+1}#r_{src_1})[$U_6(r_{src_2}[5:0]) - 1 : 0$]#($r_{src_1+1}$#$r_{src_1}$)[63 :
		$U_6(r_{src_2}[5:0])$])
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if (dst % 2 == 0 & src % 2 == 0) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sl.w - Shift left GPR

Mnemonic	:	`sl.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 01010 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} (r_{src_1}[32 - U_5(r_{src_2}[4:0]) : 0]\#(r_{src_2}[4:0])('0'))$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sl.l - Shift left long GPR

Mnemonic	:	`sl.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 01010 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} ((r_{src_1+1} \# r_{src_1})[64 - U_6(r_{src_2}[5:0]) : 0] \# (r_{src_2}[5:0])('0'))$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

sra.w - Shift right arithmetical GPR

Mnemonic	:	`sra.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 01011 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} (U_5(r_{src_2}[4:0]))(r_{src_1}[31]) \# r_{src_1}[31 : U_5(r_{src_2}[4:0]) : 0])$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

sra.l - Shift right arithmetical long GPR

Mnemonic	:	`sra.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 01011 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} ((U_6(r_{src_2}[5:0]))((r_{src_1+1} \# r_{src_1})[63]) \# (r_{src_1+1} \# r_{src_1})[63:0])])$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

srl.w - Shift right logical GPR

Mnemonic	:	`srl.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 01100 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} (U_5(r_{src_2}[4:0]))('0') \# r_{src_1}[31 : U_5(r_{src_2}[4:0]) : 0])$ $r_{dst}.c \longleftarrow_1 alu_0.c$ $r_{dst}.v \longleftarrow_1 alu_0.v$ $r_{dst}.z \longleftarrow_1 alu_0.z$ $r_{dst}.n \longleftarrow_1 alu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

srl.l - Shift right logical long GPR

Mnemonic	:	`srl.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 01100 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} ((U_6(r_{src_2}[5:0]))('0') \# (r_{src_1+1} \# r_{src_1})[63 : U_6(r_{src_2}[5:0])])$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

pol2recx.w - Polar to rectangular x of GPR's

Mnemonic	:	`pol2recx.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 10000 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} * cos(r_{src_2})$ $r_{dst}.c \longleftarrow_1 cordic_0.c$ $r_{dst}.v \longleftarrow_1 cordic_0.v$ $r_{dst}.z \longleftarrow_1 cordic_0.z$ $r_{dst}.n \longleftarrow_1 cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

pol2recx.l - Polar to rectangular x of long GPR's

Mnemonic	:	`pol2recx.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 10000 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) * cos(r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

pol2recx.f - Polar to rectangular x of FPR's

Mnemonic	:	`pol2recx.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 10000 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} * cos(f_{src_2})$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

pol2recx.d - Polar to rectangular x of double FPR's

Mnemonic	:	`pol2recx.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 10000 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} (f_{src_1+1} \# f_{src_1}) * cos(f_{src_2+1} \# f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0$) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

pol2recy.w - Polar to rectangular src of GPR's

Mnemonic	:	`pol2recy.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 10001 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} * sin(r_{src_2})$ $r_{dst}.c \longleftarrow_1 cordic_0.c$ $r_{dst}.v \longleftarrow_1 cordic_0.v$ $r_{dst}.z \longleftarrow_1 cordic_0.z$ $r_{dst}.n \longleftarrow_1 cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

pol2recy.l - Polar to rectangular src of long GPR's

Mnemonic	:	`pol2recy.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 10001 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) * sin(r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

pol2recy.f - Polar to rectangular src of FPR's

Mnemonic	:	`pol2recy.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 10001 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} * sin(f_{src_2})$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

pol2recy.d - Polar to rectangular src of double FPR's

Mnemonic	:	`pol2recy.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 10001 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} (f_{src_1+1} \# f_{src_1}) * sin(f_{src_2+1} \# f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

hyp2recx.w - Hyperbolic to rectangular x of GPR's

Mnemonic	:	`hyp2recx.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 10010 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} * \hat{}(r_{src_2})$ $r_{dst}.c \longleftarrow_1 cordic_0.c$ $r_{dst}.v \longleftarrow_1 cordic_0.v$ $r_{dst}.z \longleftarrow_1 cordic_0.z$ $r_{dst}.n \longleftarrow_1 cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

hyp2recx.l - Hyperbolic to rectangular x of long GPR's

Mnemonic	:	`hyp2recx.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 10010 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) * \curlyvee (r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

hyp2recx.f - Hyperbolic to rectangular x of FPR's

Mnemonic	:	`hyp2recx.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 10010 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} * \curlyvee (f_{src_2})$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

hyp2recx.d - Hyperbolic to rectangular x of double FPR's

Mnemonic	:	`hyp2recx.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 10010 uuuu`

Operation	:	if (dst % 2 == 0 & src % 2 == 0) then
		$(f_{dst+1}\#f_{dst}) \longleftarrow_{64} (f_{src_1+1}\#f_{src_1}) * \hat{} (f_{src_2+1}\#f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if (dst % 2 == 0&src % 2 == 0) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

hyp2recy.w - Hyperbolic to rectangular src of GPR's

Mnemonic	:	`hyp2recy.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 10011 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} r_{src_1} * sinh(r_{src_2})$ $r_{dst}.c \longleftarrow_1 cordic_0.c$ $r_{dst}.v \longleftarrow_1 cordic_0.v$ $r_{dst}.z \longleftarrow_1 cordic_0.z$ $r_{dst}.n \longleftarrow_1 cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

hyp2recy.l - Hyperbolic to rectangular src of long GPR's

Mnemonic	:	`hyp2recy.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 10011 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} (r_{src_1+1} \# r_{src_1}) * sinh(r_{src_2+1} \# r_{src_2})$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

hyp2recy.f - Hyperbolic to rectangular src of FPR's

Mnemonic	:	`hyp2recy.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 10011 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} f_{src_1} * sinh(f_{src_2})$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

hyp2recy.d - Hyperbolic to rectangular src of double FPR's

Mnemonic	:	`hyp2recy.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 10011 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} (f_{src_1+1} \# f_{src_1}) * sinh(f_{src_2+1} \# f_{src_2})$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

norm.w - Norm of GPR's

Mnemonic	:	`norm.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 10100 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} sqrt(r^2_{src_1} + r^2_{src_2})$ $r_{dst}.c \longleftarrow_1 cordic_0.c$ $r_{dst}.v \longleftarrow_1 cordic_0.v$ $r_{dst}.z \longleftarrow_1 cordic_0.z$ $r_{dst}.n \longleftarrow_1 cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

norm.l - Norm of long GPR's

Mnemonic	:	`norm.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 10100 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} sqrt((r_{src_1+1} \# r_{src_1})^2 + (r_{src_2+1} \# r_{src_2})^2)$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

norm.f - Norm of FPR's

Mnemonic	:	`norm.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 10100 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} sqrt(f^2_{src_1} + f^2_{src_2})$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

norm.d - Norm of double FPR's

Mnemonic	:	`norm.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 10100 uuuu`

Operation	:	if $(dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0)$ then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} sqrt((f_{src_1+1} \# f_{src_1})^2 + (f_{src_2+1} \# f_{src_2})^2)$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if $(dst \ \% \ 2 == 0 \& src \ \% \ 2 == 0)$ then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

atan2.w - Arctangent of GPR's

Mnemonic	:	`atan2.w rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 00 10101 uuuu`
Operation	:	$r_{dst} \longleftarrow_{32} atan(r_{src_1}, (r_{src_2}))$ $r_{dst}.c \longleftarrow_1 cordic_0.c$ $r_{dst}.v \longleftarrow_1 cordic_0.v$ $r_{dst}.z \longleftarrow_1 cordic_0.z$ $r_{dst}.n \longleftarrow_1 cordic_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$
Notes	:	$r_{dst}.n$ is implicit.

atan2.l - Arctangent of long GPR's

Mnemonic	:	`atan2.l rdst,rsrc1,rsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 01 10101 uuuu`

Operation	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$(r_{dst+1} \# r_{dst}) \longleftarrow_{64} atan((r_{src_1+1} \# r_{src_1}), (r_{src_2+1} \# r_{src_2}))$
		$r_{dst+1}.c \longleftarrow_1 alu_1.c$
		$r_{dst+1}.v \longleftarrow_1 alu_1.v$
		$r_{dst+1}.z \longleftarrow_1 alu_1.z$
		$r_{dst+1}.n \longleftarrow_1 alu_1.n$
		$r_{dst}.c \longleftarrow_1 alu_0.c$
		$r_{dst}.v \longleftarrow_1 alu_0.v$
		$r_{dst}.z \longleftarrow_1 alu_0.z$
		$r_{dst}.n \longleftarrow_1 alu_0.n$
		else
		$cr[4] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if ($dst \ \% \ 2 == 0 \ \& \ src \ \% \ 2 == 0$) then
		$r_{dst}.c$, $r_{dst}.v$, $r_{dst}.z$, $r_{dst}.n$, $r_{dst+1}.c$, $r_{dst+1}.v$, $r_{dst+1}.z$, $r_{dst+1}.n$
		else
		$cr[4] \longleftarrow_1 1$

Notes	:	$r_{dst}.n$ and $r_{dst+1}.n$ are implicit.

atan2.f - Arctangent of FPR's

Mnemonic	:	`atan2.f fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 10 10101 uuuu`
Operation	:	$f_{dst} \longleftarrow_{32} atan(f_{src_1}, (f_{src_2}))$ $f_{dst}.a \longleftarrow_1 fpu_0.a$ $f_{dst}.i \longleftarrow_1 fpu_0.i$ $f_{dst}.z \longleftarrow_1 fpu_0.z$ $f_{dst}.n \longleftarrow_1 fpu_0.n$
Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31; src1 range: 0:31; src2 range: 0:31`
Affected flags	:	$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$
Notes	:	$f_{dst}.n$ is implicit.

atan2.d - Arctangent of double FPR's

Mnemonic	:	`atan2.d fdst,fsrc1,fsrc2`
Coding	:	`11 uuuu rsrc1 r_dst rsrc2 11 10101 uuuu`

Operation	:	if (dst % 2 == 0 & src % 2 == 0) then
		$(f_{dst+1} \# f_{dst}) \longleftarrow_{64} atan((f_{src_1+1} \# f_{src_1}), (f_{src_2+1} \# f_{src_2}))$
		$f_{dst+1}.a \longleftarrow_1 fpu_1.a$
		$f_{dst+1}.i \longleftarrow_1 fpu_1.i$
		$f_{dst+1}.z \longleftarrow_1 fpu_1.z$
		$f_{dst+1}.n \longleftarrow_1 fpu_1.n$
		$f_{dst}.a \longleftarrow_1 fpu_0.a$
		$f_{dst}.i \longleftarrow_1 fpu_0.i$
		$f_{dst}.z \longleftarrow_1 fpu_0.z$
		$f_{dst}.n \longleftarrow_1 fpu_0.n$
		else
		$cr[5] \longleftarrow_1 1$
		endif

Assember Validations	:	`None.`
CPU Validations	:	`dst range: 0:31 and even; src1 range: 0:31 and even; src2 range: 0:31 and even`

Affected flags	:	if (dst % 2 == 0 & src % 2 == 0) then
		$f_{dst}.a$, $f_{dst}.i$, $f_{dst}.z$, $f_{dst}.n$, $f_{dst+1}.a$, $f_{dst+1}.i$, $f_{dst+1}.z$, $f_{dst+1}.n$
		else
		$cr[5] \longleftarrow_1 1$

Notes	:	$f_{dst}.n$ and $f_{dst+1}.n$ are implicit.

2.3 TODO's y discusiones

2.3.1 Hablado en Marzo

- Operaciones sobre long: OK y agregado al doc
- Z y N son consistentes con el resultado intermedio: OK y agregado al doc
- ALU con bus de 64 bits: OK y agregado al doc
- Register file con 2 enable de escritura y data_in de 64 bits =¿ ataca a la alu con los 64 bits y ataca a la memoria con 32: Nada que aclarar en este documento. Esto tiene más que ver con la implementación
- Se discutió si utilizar interrupciones sin vectorizar =¿ ganas flexibilidad en el tratamiento de la IRQ pero perdes en flexibilidad de definición de interrupciones. Quedó pendiente

2.3.2 Abril

- Se habló sobre como implementar excepciones: Excepciones -¿ 32 causas distintas. Una de ellas es interrupción. La gestión de interrupción se encargará el periferico correspondiente. La condición de excepción se levanta en la etapa que corresponda, y se transfiere hasta la ultima etapa del pipe, en donde se va a transferir al registro de causa (para evitar que se me levanten varios bits de excepción al mismo tiempo y poder chequear ese registro (andeado con la máscara) y hacerle or a todos los bits del resultado para ver si hay excepción). Ver de vectorizar 32 saltos con 2 instrucciones, una para saltar y otra para hacer el return from exception.
- Mapeo de periféricos: Dos opciones, una es segmentar la memoria (usando el bit mas alto para elegir entre memoria y periféricos). La otra es poner un bit en un registro y en función de ese bit, se elige a quién le hablo. Otro esquema (descartado por complejidad) es el de escribir en un reg especial de 32 bits donde una parte se usa para direccionar los periféricos y otra para pasarle o recibir datos. El problema de esto es que te deja limitado en la cantidad de periféricos desde la definición de los bits de este registro.

Sobre las excepciones, se implementa un un registro de causa de 32 bits (CR) y un registro de máscaras (MR). Una de las excepciones es interrupciones, que se manejan con un periférico (standard_interrupt_controller). El mapa de memoria de datos se divide en dos (mapeados con el MSB de la dirección). La parte baja se usa para direccionar la memoria de datos y la parte alta para los periféricos. Hay un set de periféricos que deben ser implementados sí o sí (obligatorios). Otros se pueden llegar a implementar opcionalmente. Los periféricos deben definirse con tres cosas:

- Registros internos
- Mapeo en memoria (direcciones)
- Puertos

2.3.3 Mayo

Lo propuesto fue:

1. El micro tiene una linea de entrada de interrupcion externa (1 bit) que corresponde a una exception mas y se atiende como cualquier otra excepcion.

2. hay un registro de mascara de excepciones.

3. las excepciones tienen prioridad: hay que determinar cuales son las prioridades (ni recuerdo que excepciones habiamos dicho de poner pero supongo que habra interrupcion, invalid operation, alu error -tipo div by zero- etc). Según lo teníamos definido antes, las prioridades se manejan por soft!

4. Cuando hay una interrupcion externa, se salta a una posicion de memoria determinada (no me acuerdo cual pusiste en el PDF) y se atiende la interrupcion. El address the boot (valor del PC luego del reset) es 0x0000. Cuando hay una excepción se salta a la dir 0x000000F0 y de ahí se hace todo el manejo por soft. Igual esto se podría cambiar

5. el espacio de memoria de programa empieza a partir de 1K (0x0400) Eso es para dejar espacio para código de booteo???

6. Las interrupciones se atienden mediante un controlador de interrupciones programable (PIC): en la rutina de interrupcion el micro consulta al PIC preguntando el address del periferico que interrumpe y el dato. Esto lo dejé planteado para ver como lo resolvemos en la sección de interrupciones

7. El micro tiene 3 registros especiales para interfaceo de perifericos:
   * DPW (data periphal write - 32bits): directamente cableado a un puerto de salida, corresponde al dato que el micro escribe al periferico.
   * DPR (data periphal read - 32bits): directamente cableado a un puerto de entrada, corresponde al dato que el micro lee desde el periferico.
   * PAD (peripheral address - 16 bits): directamente cableado a un puerto de salida, corresponde al address del periferico que el micro direcciona. Este registro de 16 bits genera un espacio de 16K posibles perifericos ya que al igual que las memorias de datos e instrucciones, los perifericos son direccionados de a bytes.
   Estos tres registros son escritos y leidos con las instrucciones movi2s y movs2i. Esto hace que los perifericos sean transparantes para una MMU, la MMU solo se ocupa de memoria de programa + datos y no de perifericos. Ya están agregados estos registros

8. El PIC siempre debe ir en el address 0x0000 direccionada por el PAD. Todo otro periferico se direcciona a partir de la direccion 0x0004 del PAD. Esto es para que el micro le pueda hablar al PIC??? Yo lo que hice es que el PIC provee un set de registros específios que se usan con movi2s y movs2i, para configurarlo... haría falta esto???

9. No hay perifericos obligatorios a excepcion del PIC. Dependen de la aplicacion. El PIC es el unico obligatorio.

2.3.4 Mayo 2

Resolvimos casi todo lo dejado pendiente. Aún sigue pendiente definir de forma urgente cómo se comunican los periféricos con el PIC para mandarle data al CPU.

Falta generar un criterio de prioridad de excepciones y explicitarlo en el documento.

2.3.5 Junio

1. Todavía falta ver lo del protocolo de periféricos y el PIC, definirlos bien y documentarlos
2. Branch delay slot: definir el branch delay slot de las instrucciones de branching y de las excepciones. Pueden ser distintos. Lo que definamos de BDS va a definir en cuanto hay que incrementar el PC!!!
3. Meter las instrucciones del cordic de nacho -¿ esto sería una "isa extension"???
4. Actualizar gráficos de formato de instrucciones
5. Por definición r31 es el return address (las instrucciones de jal y jalr escriben en el 31 y no puede modificarse ese comportamiento. Convención que el stack pointer sea el r30. Esto done tenemos que aclararlo / agregarlo???
6. Agregar modos de ejecución supervisor - user
7. Agregar IR - Instruction Register, es un registro que se carga con el opcode de la instrucción que se está ejecutando.
8. Se agregó un bit de máscara global que es sólo accesible por hardware (o sea, existe y es el bit 32 del registro de máscaras, pero por soft sólo se puede acceder de los bits 0 a 31). Ese bit NO DEBE enmascarar al reset, porque por su funcionamiento, se activa cuando entramos en la ejecución de un handler de excepciones, esto denota que no existen excepciones anidadas, y se desactiva al finalizar la ejecución del handler. Si ese bit enmascarara al reset (que es la excepcion de máxima prioridad), no podría resetearse el micro mientras se está ejecutando un handler, cosa que no es deseable.
9. Resta aclarar cómo se cargan los flags de registros al levantar de memoria los datos, dado que en realidad los flags son resultantes de las operaciones sobre los registros, pero algunos deberían ser siempre coherentes con el dato levantado de memoria (por ejemplo Z).
10. Definir el mecanismo con el que se sale del handler de excepción (rfe) que se hace con los bits de máscara.
11. Definir la excepcion TRAP que genera una excepción por software.
12. Cómo se baja el flag de la excepción que se empieza a atender? Puede ser o bien apenas saltamos al handler, porque si lo hacemos al final (con el rfe) tenemos dos problemas, que no sabemos de qué excepción venimos y no sabemos si efectivamente se había levantado nuevamente esa excepcion (que podría ser el caso de las interrupciones)). Una cosa importante acá también es que es IMPORTANTE que un handler no pueda generar excepciones (o sea, un driver mal escrito podría hacer cagadas...).

En cuanto a los modos de ejecución:

1. necesitamos un stack pointer de supervisor distinto al de user el cual es R31?
2. hay que agregar el registro especial US de un solo bit al mapa de registros

3. en la descripcion del handling de la exception indicar que ademas de que el registro L guarda el program counter, el registro US se pone en '0' y el global mask en '1'. Cuando viene la instrucion RFE al salir de la rutina de exception ser restaura el valor del program counter, el global_mask va a '0' y el US va a '1'.

4. en los I/O del micro debe estar como salida el bit S del registro US, asi la MMU sabe si los accesos a memoria son en modo supervisor o usuario.

Si vemos un poco lo que pasa en MIPS con los system calls, básicamente, se sigue un protocolo (medio similar a la ABI) de en qué registros guardar ciertas cosas y el micro levanta la data de esos registros para operar en las system calls, que creo que se resuelven completamente por soft. O sea, sería, una vez más, en pos de simplicidad, relegar el laburo al soft, como lo venimos haciendo. Octavio dice de meter un campo inmediato para poner el tipo de syscall pero capaz que manejándonos con los registros no hace falta. Ahora lo que sé es que cada micro define sus system calls... por sentido común y / o experiencia, cuales deberíamos meter???