Tampereen Teknillinen Yliopisto (TTY)

TAMPERE UNIVERSITY
OF TECHNOLOGY
Institute of Digital and Computer Systems

MILK COPROCESSOR

External version 1.0

Author: **Claudio Brunelli**

rev. June 2007

Contents

List of Figures

List of Tables

Chapter 1

Internal architecture

1.1 Interface and block schemes

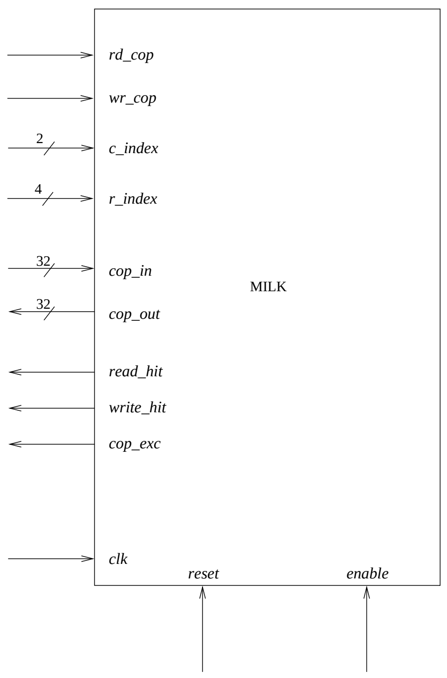

Figure 1.1: Milk coprocessor

Milk (figure 1.1) is a floating-point unit (FPU) which can be used as a coprocessor for a RISC core; in particular it was developed to interface with two cores (even though it could be attached virtually to any core, by modifying its interface or adding a wrapper): the COFFEE RISC core from Tampere University of Technology, and Qrisc core, from the University of Bologna.
Milk is attached to the core via a dedicated, local data bus which is 32-bit wide.

The address bus is split in two parts: one part (made up of signal `c_index`) is used to index Milk among the other coprocessors which could possibly be attached to the core, and the other part (made up of signal `r_index`) is used to access one of its internal registers. Signal `r_index` is also made so that its 3 least significant bits identify one of the general purpose registers of Milk (as long as the MSB, that is `r_index(3)`,is equal to 0); when `r_index(3)` is equal to 1 then we are accessing a special purpose register inside Milk. In particular, the configuration 1000 addresses the control register, the configuration 1001 addresses the status register, the configuration 1010 addresses the register whose MSB contains the result of comparisons (for compatibility with Qrisc core).
The interface of Milk comprises then a set of control signals; they are all described in table 1.1. A description of each of the internal blocks which build up Milk coprocessor will be presented in the next pages.

Table 1.1: milk coprocessor external pins

PIN NAME	DIR	WIDTH	DESCRIPTION
`clk`	in	1	clock signal.
`reset`	in	1	reset signal.
`enable`	in	1	enables the internal registers of the coprocessor to sample their inputs.
`cop_in`	in	32	data bus used by the core to load operands and instruction words into Milk.
`cop_out`	out	32	data bus used by the core to read back results and the contents of the status register of Milk.
`rd_cop`	in	1	read command. When it is set to its “high” logical value the core reads the result of an arithmetic operation (or the contents of the status register) from Milk.
`wr_cop`	in	1	write command. When it is set to its “high” logical value the core writes an operand (or an instruction word) inside Milk.
`c_index`	in	2	coprocessor address. The configuration “01” identifies Milk among the other coprocessors which can possibly be connected to the core.
`r_index`	in	4	register address.
`cop_exc`	out	1	when an exception occurs inside Milk during computation, this signal is pulsed. In particular, it keeps an high value for one clk cycle.
`read_hit`	out	1	this signal goes high when the core reads a data from milk.
`write_hit`	out	1	this signal goes high when the core writes a data or an instruction into milk.
`cop_stall`	out	1	this (pulsed) signal is used to send an interrupt request to the core.

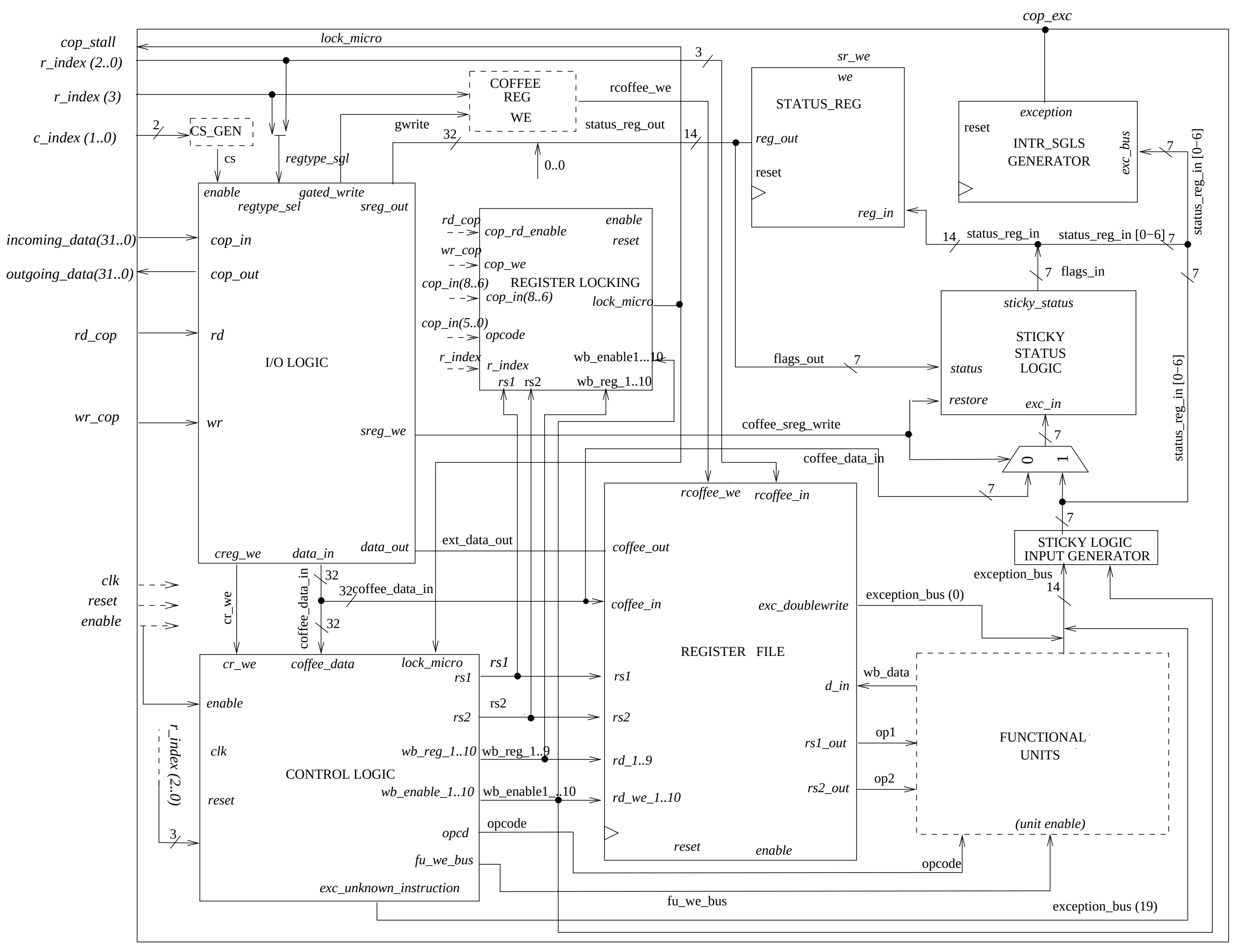

Figure 1.2: Milk coprocessor internal architecture

1.2 I/O logic

The I/O logic is in charge of providing an interface between Milk and the core.

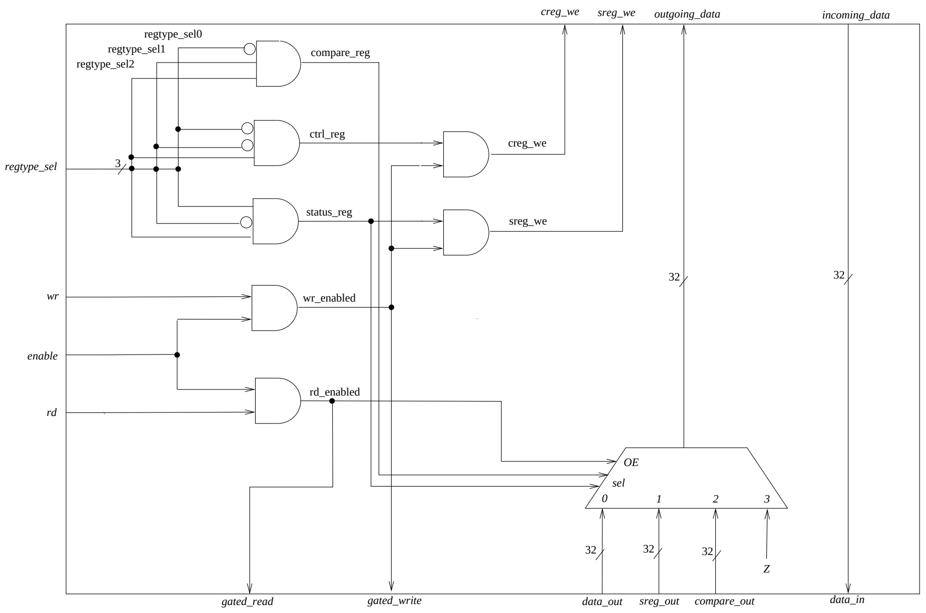

Figure 1.3: Milk coprocessor I/O logic

Table 1.2: IO logic external ports

PORT NAME	DIR	WIDTH	DESCRIPTION
`enable`	in	1	driven by signal `cs`, which is set active when Milk coprocessor is the one currently indexed by the core.
`rd`	in	1	driven directly by port `rd_cop`, which is the control command used by the core to read data from Milk.
`wr`	in	1	driven directly by port `wr_cop`, which is the control command used by the core to send data to Milk.
`regtype_sel`	in	2	driven by signal `regtype_sgl`, which specifies whether the currently indexed register is a general purpose register, or the instruction register, or the status register.
`gated_write`	out	1	drives signal `write_hit_s`, which is used in combination with with signal `r_index(3)` to generate signal `rcoffee_we`, that is the write enable command of the registers in the register file. Signal `rcoffee_we` is connected to port `rcoffee_we` of the `REGISTER_FILE`.
`gated_read`	out	1	drives signal `read_hit_s`, which is connected to the output port `read_hit` of Milk.
`creg_we`	out	1	drives signal `cr_we`, which is connected to port `cr_we` in the `CONTROL LOGIC` and is used to let the current instruction word reach the instruction register.
`sreg_we`	out	1	drives signal `coffee_sreg_write`, which is connected to port `restore` in the `STICKY STATUS LOGIC`, and is used to manually reset the flag bits in the status register.
`outgoing_data`	out	32	Directly connected to port `cop_out`, that is the external outgoing data bus.
`incoming_data`	in	32	Directly connected to port `cop_in`, that is the external incoming data bus.
`data_in`	out	32	drives signal `coffee_data_in`, which is connected to port `coffee_in` in the `REGISTER FILE`, and brings data from the core to the register file, the control register and the status register.
`sreg_out`	in	32	driven by signal `status_reg_out`, which comes from the status register, making available its content to the core.
`data_out`	in	32	driven by signal `ext_data_out`, which comes from port `coffee_out` in the `REGISTER FILE`. It brings to the core the result of computations (stored in the indexed register).

1.3 Register file

The Register file contains 8 registers, each 32-bits wide, and the mux/demux logic to route the desired values to/from the selected registers. They are general purpose registers, used to store the operands and the results of arithmetic computations.

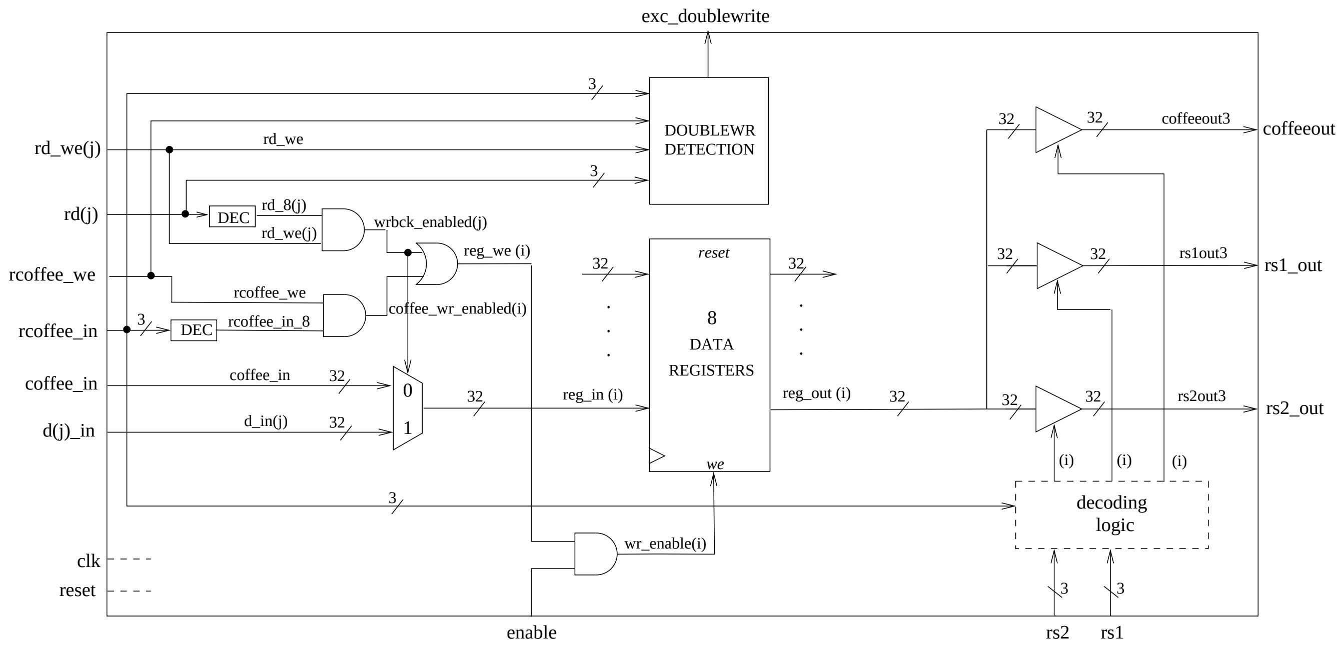

Figure 1.4: Milk coprocessor register file

Table 1.3: register file external ports

PORT NAME	DIR	WIDTH	DESCRIPTION
`clk`	in	1	clock signal
`reset`	in	1	reset signal
`enable`	in	1	driven by an high value.
`rd_we(j)`	in	1	driven by signal `wb_enable`, which is used by the control logic in order to specify that a functional unit is enabled to perform a writeback of its result in the register file.
`rd(j)`	in	3	driven by signal `wb_reg`, which is used by the control logic in order to specify the address of the target register in which the result has to be stored.
`rcoffee_we`	in	1	driven by signal `rcoffee_we`, which is used in order to specify that the core is enabled to write a data in the register file.
`rcoffee_in`	in	3	driven by signal `r_index[2..0]`, which is used in order to specify the address of the target register in which the data from the core has to be stored.
`coffee_in`	in	32	driven by signal `coffee_data_in`, which is the data sent by the core.
`d(j)_in`	in	32	driven by signal `(FU name)_out`, which is the result sent by the selected functional unit.
`coffee_out`	out	32	drives signal `ext_data_out`, that is connected to port `data_out` in the `IO LOGIC` and is then connected to the external outgoing data bus.
`rs1_out`	out	32	drives signal `op1`, which is connected to the port of the first operand of each of the functional unit.
`rs2_out`	out	32	drives signal `op2`, which is connected to the port of the second operand of each functional unit which require two operands.

1.4 Control logic

The control logic rules the execution inside Milk. Inside the control logic there is the control register, which stores the instruction words sent to Milk by the core (the details of the instruction register can be found in chapter 3, special regsiters, while the encoding of the instructions can be found in chapter 4, instruction set). The control logic enables each functional unit (FU) to make the writeback of results only at the right time. Moreover, it controls the flow of data inside Milk, selecting the correct source/destination registers.

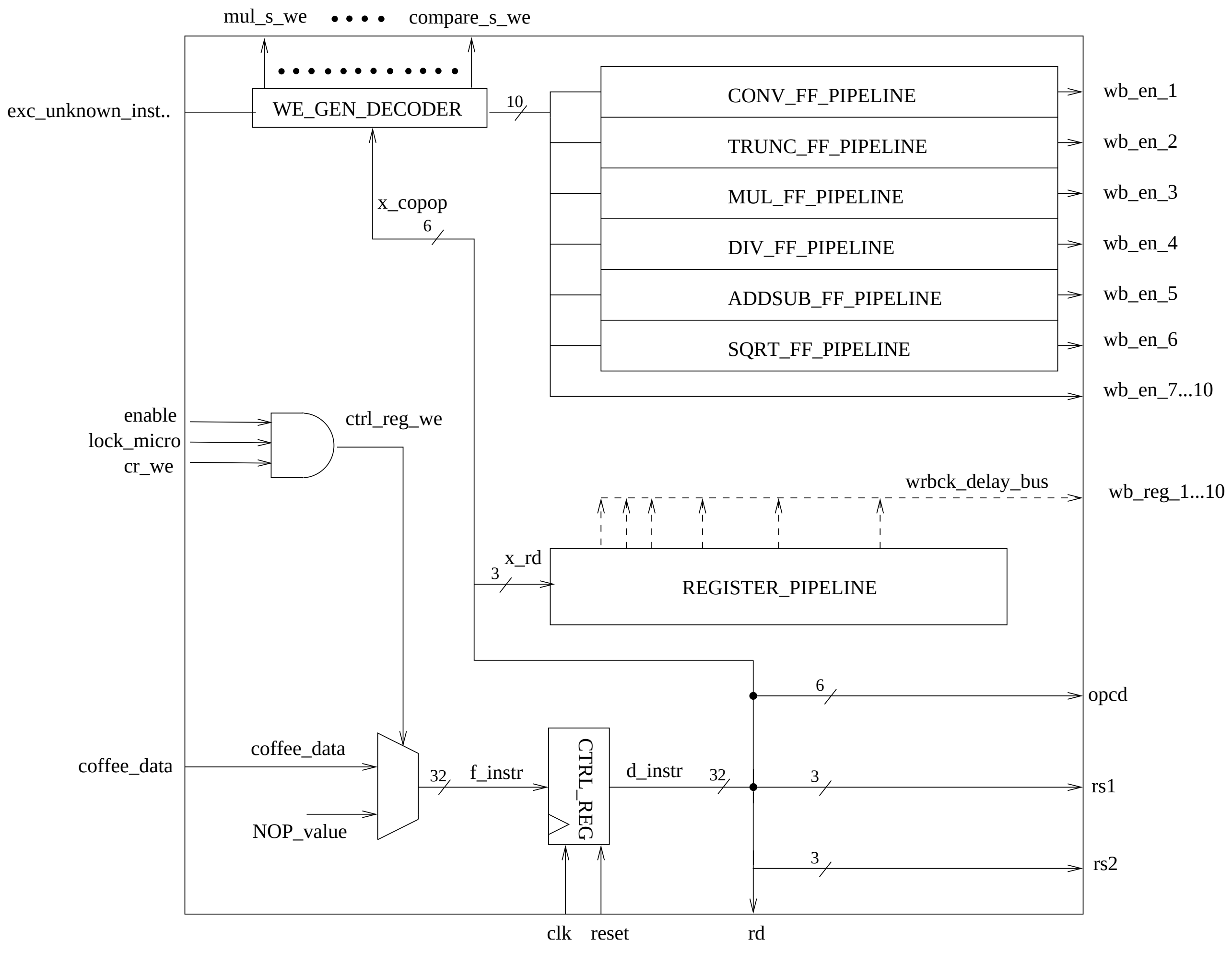

Figure 1.5: Milk coprocessor control logic

Table 1.4: control logic external ports

PORT NAME	DIR	WIDTH	DESCRIPTION
`clk`	in	1	clock signal.
`reset`	in	1	reset signal.
`enable`	in	1	driven by main enable signal sent by the core.
`cr_we`	in	1	driven by signal `cr_we`, which is used by the IO logic in order to specify that the control register has to sample the control word sent by the core.
`coffee_data`	in	32	driven by signal `coffee_data_in`, which is the control word sent by the core.
`cvt_s_we`	out	1	drives signal `cvt_s_we`, which enables the "integer to single precision float" converter to sample the current operands.
`trunc_s_we`	out	1	drives signal `trunc_s_we`, which enables the "single precision float to integer" converter to sample the current operands.
`mul_s_we`	out	1	drives signal `mul_s_we`, which enables the multiplier to sample the current operands.
`add_s_we`	out	1	drives signal `add_s_we`, which enables the adder to sample the current operands.
`sub_s_we`	out	1	drives signal `sub_s_we`, which enables the adder to sample the current operands.
`div_s_we`	out	1	drives signal `div_s_we`, which enables the divider to sample the current operands.
`sqrt_s_we`	out	1	drives signal `sqrt_s_we`, which enables the square root extractor to sample the current operands.
`compare_s_we`	out	1	drives signal `compare_s_we`, which enables the comparator to sample the current operands.

Table 1.5: control logic external ports (continued)

`exc_unknown_instruction`	out	1	drives signal `exception_bus(19)`. It's set to a logical high value when the core writes an instruction word containing an opcode which is not valid for Milk.
`wb_en_(j)`	out	1	drives signal `wb_en_(j)`, that is connected to port `rd_we(j)` in the `REGISTER FILE` and enables the target register to sample the current result from functional unit (j).
`wb_reg_(j)`	out	3	drives signal `wb_reg_(j)`, which is connected to port `rd_(j)` in the `REGISTER FILE`, and specifies the address of the target register.
`rs1`	out	3	drives signal `rs1`, which is connected to port `rs1` in the `REGISTER FILE`, and specifies the address of the first of the current operands.
`rs2`	out	3	drives signal `rs2`, which is connected to port `rs2` in the `REGISTER FILE`, and specifies the address of the second of the current operands.
`opcd`	out	6	drives signal `opcode`, which is connected to port `opcode` in the `comparator`, and specifies the address of the first of the current operands.

1.5 Interrupt request signal generator

It is a very simple logic, which ensures that interrupt signals (sent by Milk to the core) last for one clock cycle.

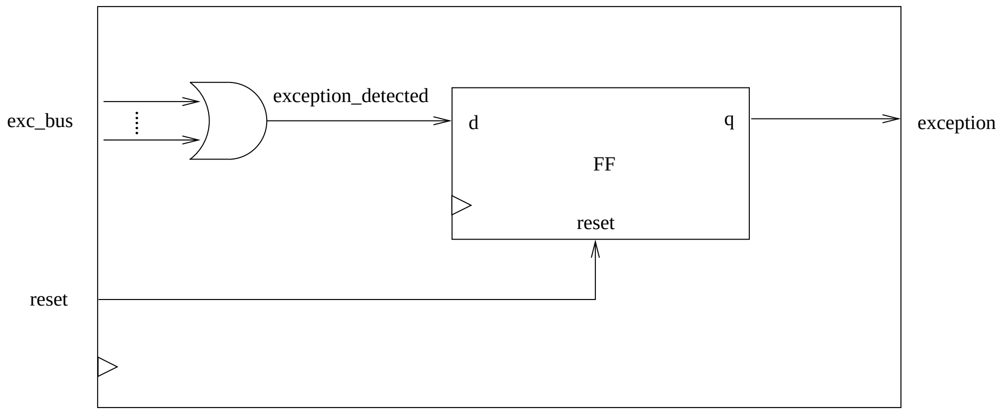

Figure 1.6: Milk coprocessor interrupt request signal generation logic

1.6 Register locking

This block generates a stall signal, which freezes the core when necessary, in order to guarantee the logic consistency of the flow of application programs. In particular, the core is stalled whenever it tries to read a result from Milk when it is not ready yet (the result is still being calculated); the core is also stalled when it tries to issue to Milk an instruction which is dependent on the result(s) of one or more previous instructions, which are still running in Milk's pipeline.

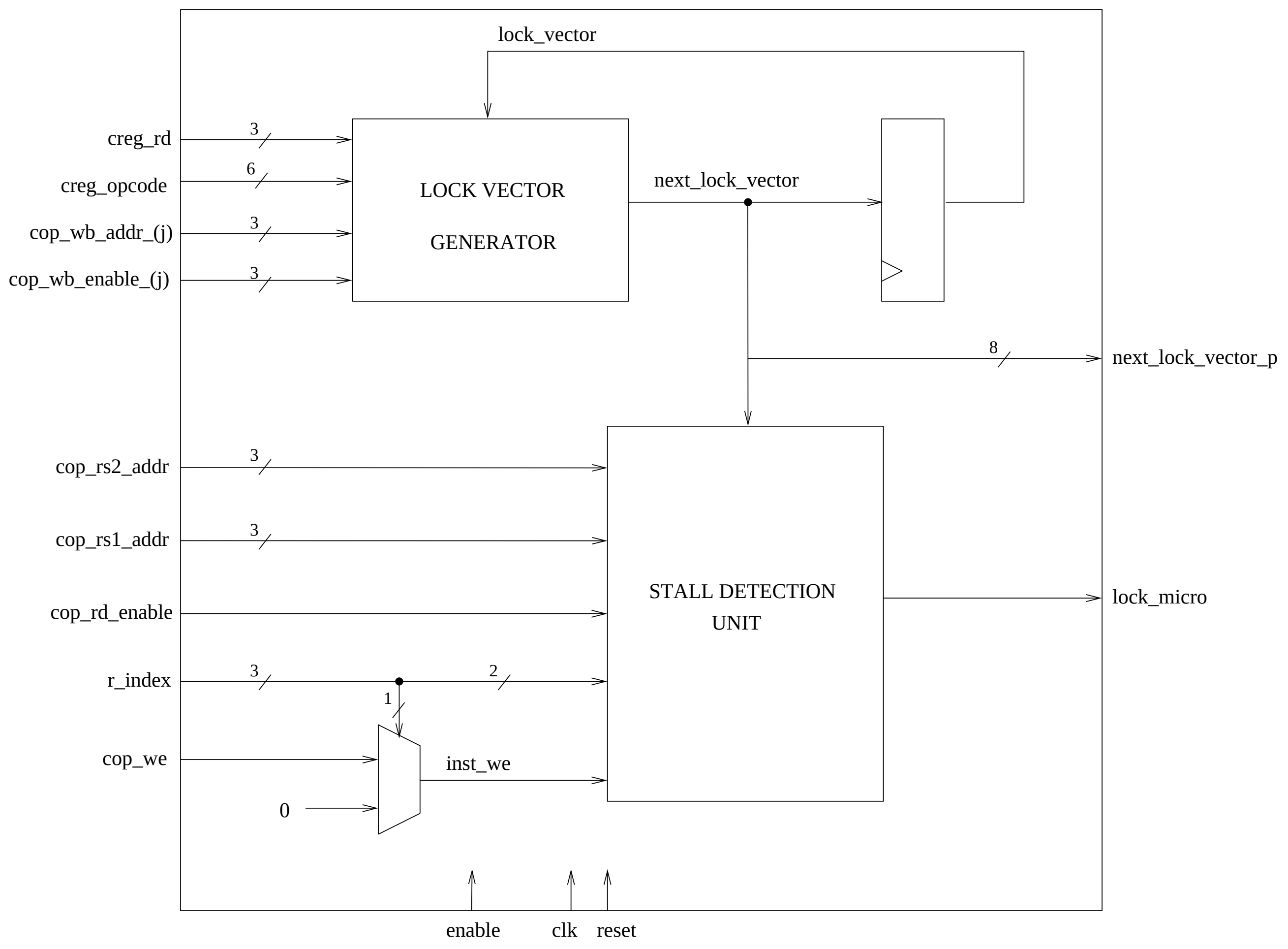

Figure 1.7: Milk coprocessor control logic

Table 1.6: register locking logic: external ports (continued)

PORT NAME	DIR	WIDTH	DESCRIPTION
`clk`	in	1	clock signal.
`reset`	in	1	reset signal.
`enable`	in	1	driven by main enable signal sent by the core.
`cop_we`	in	1	driven by signal `wr_cop`. It's set to a logical high value when the core writes a data or an instruction.
`cop_rd_enable`	in	1	driven by signal `rd_cop`, It's set to a logical high value when the core reads a result from Milk.
`cop_rs1_addr`	in	3	driven by signal `wb_reg_(j)`, which is connected to port `rd_(j)` in the `REGISTER FILE`, and specifies the address of the target register.
`cop_rs2_addr`	in	3	driven by signal `rs1`, which is connected to port `rs1` in the `REGISTER FILE`, and specifies the address of the first of the current operands.
`cop_wb_enable_(j)`	in	1	driven by signal `rs2`, which is connected to port `rs2` in the `REGISTER FILE`, and specifies the address of the second of the current operands.
`cop_wb_addr_(j)`	in	3	driven by signal `wb_reg_(j)`, which is connected to port `wb_reg_(j)` in the control logic, and specifies the address of the first of the destination register.
`lock_micro`	out	1	drives port `cop_stall`.
`creg_rd`	in	1	address of the register in the register file to wich the result is stored.
`creg_opcode`	in	2	it is the most significant part of the opcode.
`next_lock_vector_p`	out	8	it represents the content of the lock vector in the following clock cycle.

1.7 Main functional units

The functional units (FUs) are a set of components, each dedicated to perform the elaboration of a given arithmetic operation (multiplication, division, etc.). Each FU can be freely removed from the architecture just setting to 0 the value of the corresponding VHDL generic, so that the user can save area on chip when he know in advance that he won't need too often a certain operation. In particular, the divider and the square root together take a lot of area (approximately 70in certain cases. The latencies (in terms of clock cycles) of each FU are:

Table 1.7: Clock cycles per instruction

instruction	clk cycles amount
ADD.S	5
SUB.S	5
MUL.S	3
DIV.S	12
SQRT.S	8
ABS.S	1
MOV.S	1
NEG.S	1
NOP.S	1
NEG.S	1
COP.CVT.S	2
COP.CVT.W	2
COP.C.cond	1

Note that by excluding from the implementation the hardware logic which handles the denormal operands, all the latencies are then reduced by one cycle.

1.7.1 Integer to single precision conversion

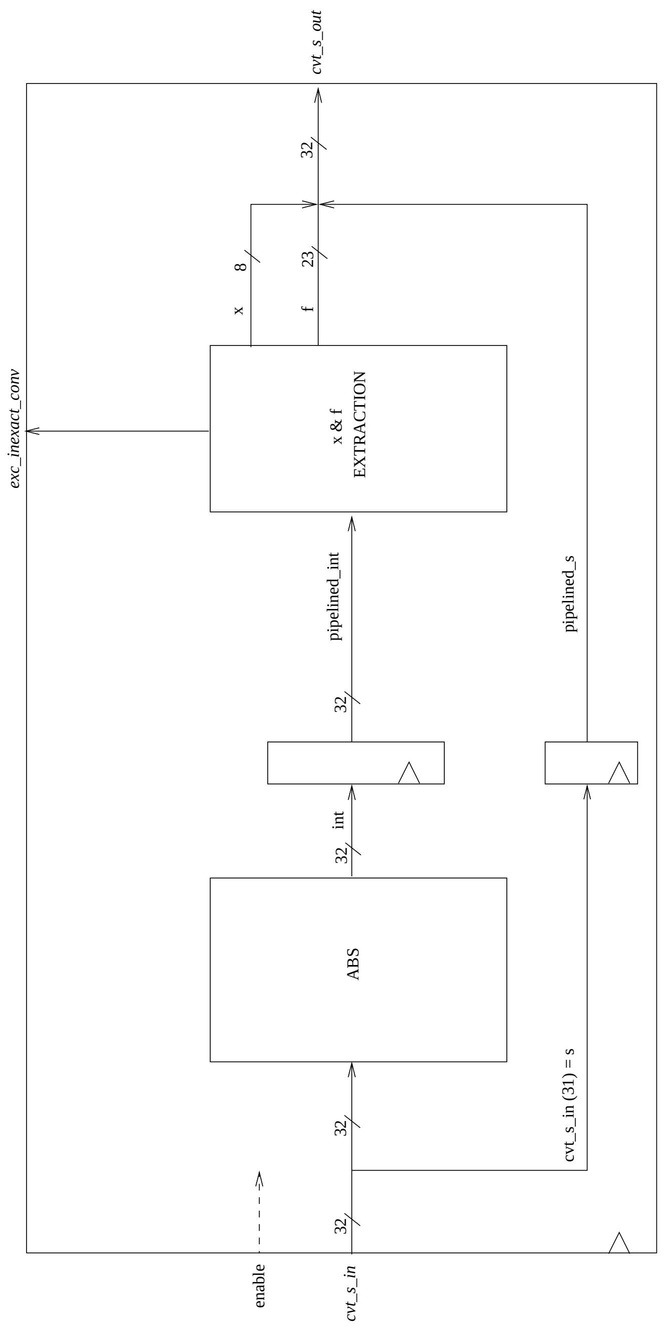

Figure 1.8: Milk coprocessor integer to single FP conversion logic

1.7.2 Single precision to integer conversion

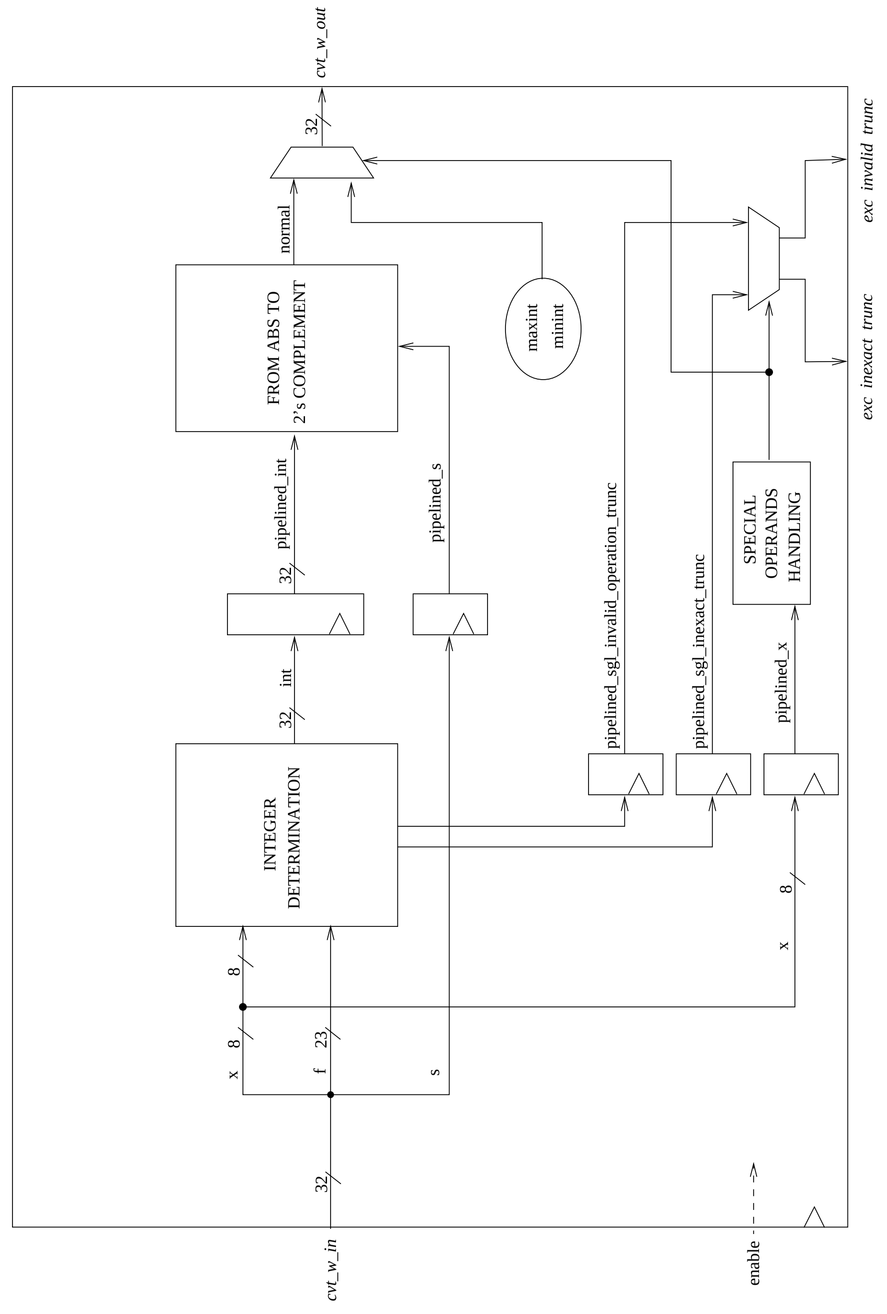

Figure 1.9: Milk coprocessor single FP to integer conversion logic

1.7.3 Floating-point multiplication

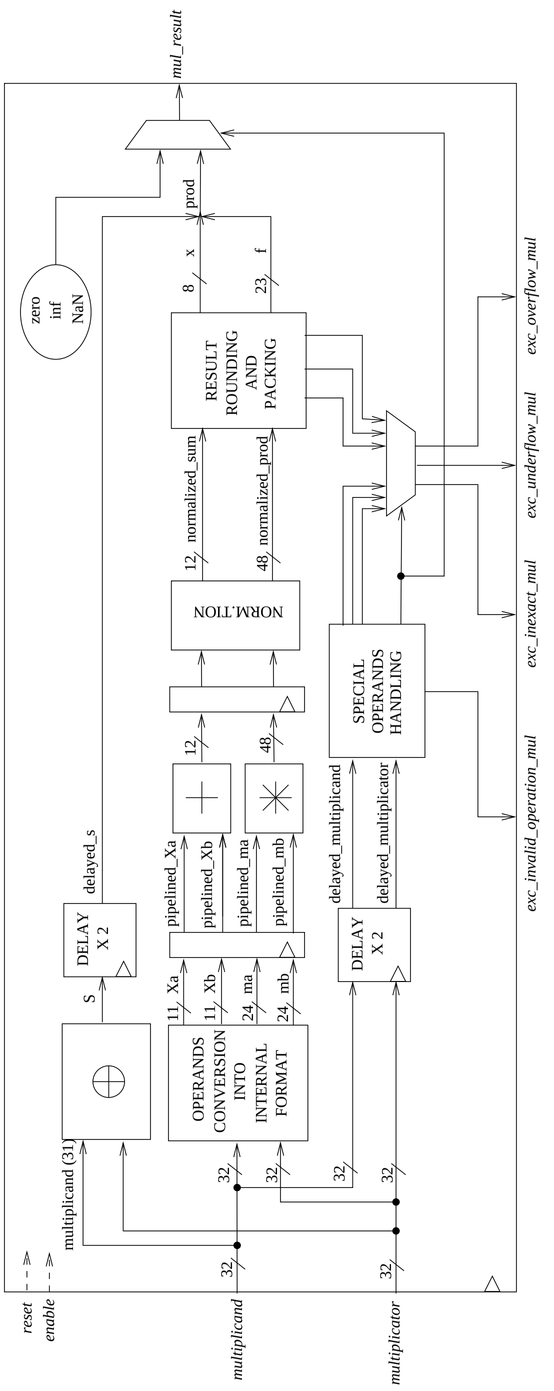

Figure 1.10: Milk coprocessor floating-point multiplier

1.7.4 Floating-point division

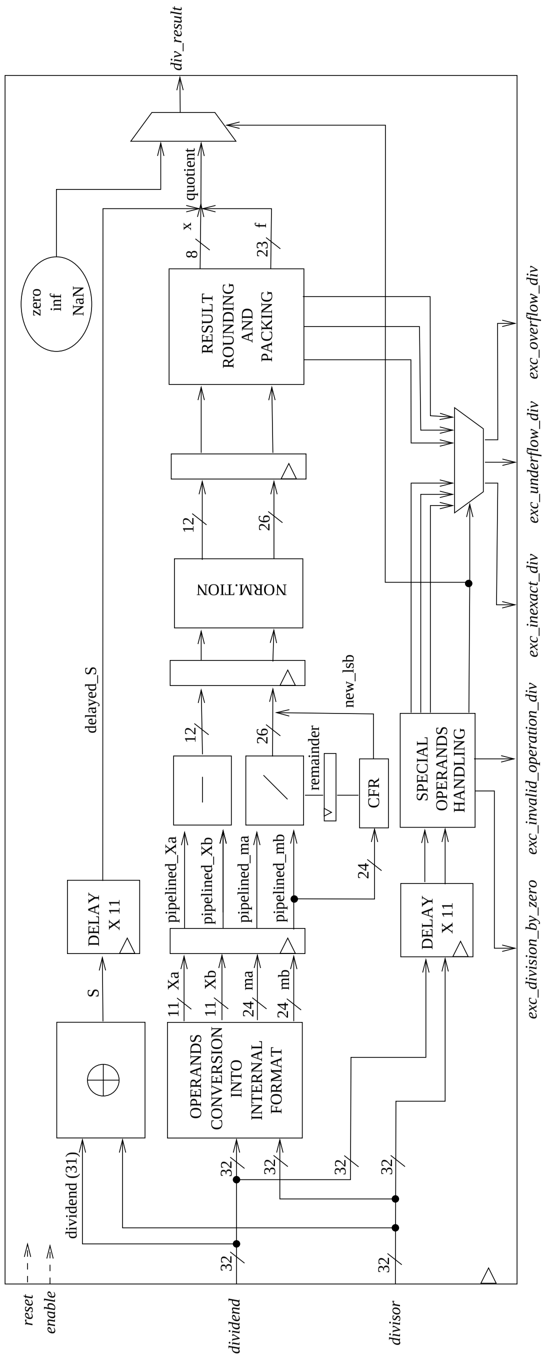

Figure 1.11: Milk coprocessor floating-point divider

1.7.5 Floating-point algebric addition

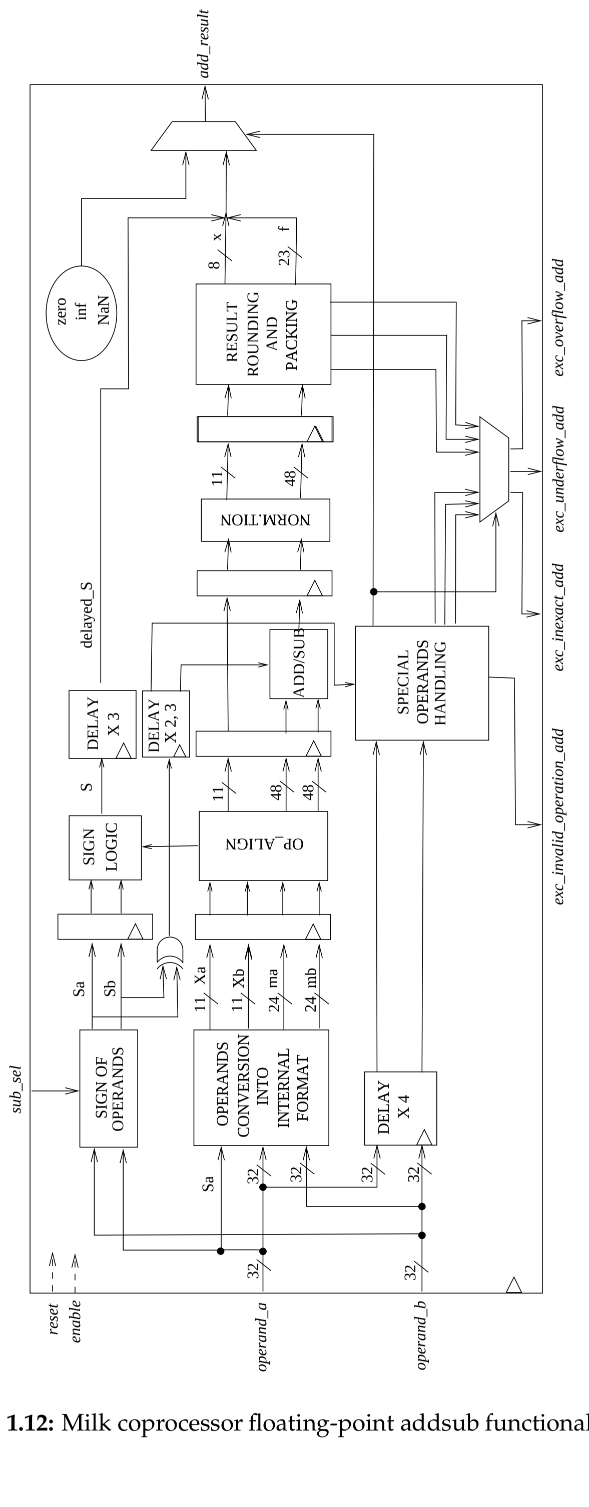

Figure 1.12: Milk coprocessor floating-point addsub functional unit

1.7.6 Floating-point square root

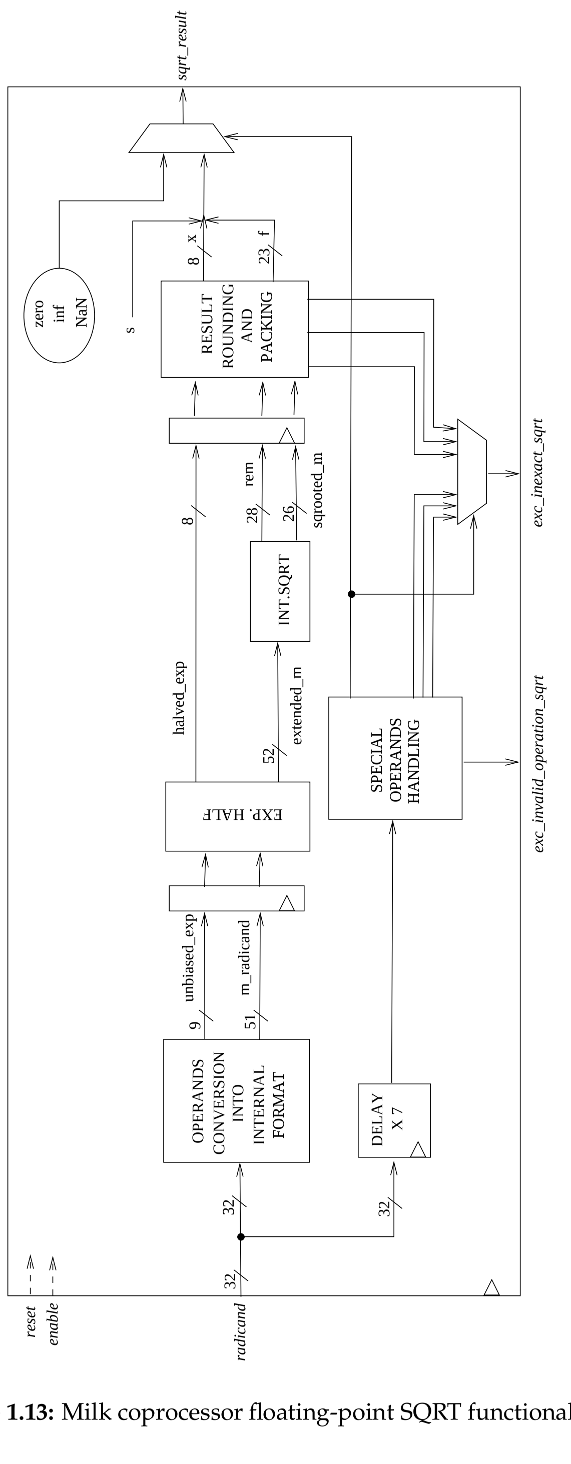

Figure 1.13: Milk coprocessor floating-point SQRT functional unit

Chapter 2

Registers

Milk has 8 general purpose 32-bit registers for arithmetic operands and results storage. Two special purpose registers are present in the achitecture: status register and control register. they are described in the following two subsections.

2.1 Status register

It's a 32-bit register (see figure 2.1). Bits 31..18, bit 7 and bit 0 are not used in the current implementation, and it's assumed they all are zeroes. Bits 17..14 contain the result of the latest comparison instruction: bit 17 is a logical flag that specifies whether the requested comparison (operand a is less than operand b, operand a is equal to operand b, and so forth) returned a TRUE or FALSE result. This result is used by the Qrisc processor. Bits 16..14 instead have respectively the following meaning: equal, less, unordered. They are used by Coffee RISC core to know what kind of relationship holds between the operands of the latest comparison issued. For instance, if bit 16 is set to one while bits 15 and 14 are zero, then it means that operand a is equal to operand b; if operand a is smaller than operand b, then bits 16 and 14 are zero while bit 15 is set to one.
Bits 13..8 are the flag bits related to floating-point exceptions, and they refer to the whole computation since last reset or last writing from user. Bits 6..1 are the same flags, but they refer to the last executed instruction only.
A more accurate description of each flag is listed below:

- **overflow** (OFC, OFA)
  The result, rounded as if the exponent range were unbounded, would be larger in magnitude than the destination format's largest finite number.

31	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
▨	CR	EQ	LT	URD	UIA	DZA	NVA	NXA	UFA	OFA	DWA	UIC	DZC	NVC	NXC	UFC	OFC	DWC

Figure 2.1: Milk coprocessor status register

- **underflow** (UFC, UFA)
  The rounded result is inexact and would be smaller in magnitude than the smallest normalized number in the destination format. Result is tiny and loss of accuracy occurs (may be either a denormalization loss or an inexact result).

- **inexact result** (NXC, NXA)
  The rounded result of an operation differs from the infinitely precise unrounded result.

- **invalid** (NVC, NVA)
  An operand is improper for the operation to be performed.

- **division by zero** (DZC, DZA)
  Division of a subnormal or normalized number by a null divisor. Note that if also dividend is null (invalid operation), DZC is not set.

- **unsupported instruction** (UIC, UIA)
  The current opcode is not recognized by Milk since corresponding instruction is not implemented in hardware.

2.2 Control register

The content of the control register is shown in figure 2.2: Bits 31..26 contains

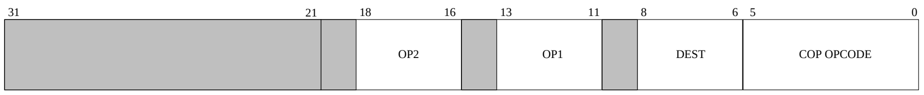

Figure 2.2: Milk coprocessor control word

the encoding of the "cop" instruction of Coffee RISC core.
Bits 25..24 are used to index one among the 4 coprocessors that can be attached to Coffee RISC core.
Bits 23..22 are unused.
Bit 21 specifies the floating-point precision. Only single precision is currently supported by Milk coprocessor, and this bit is always 0.
Bits 20..16 are the address of the second operand's source register. When the current instruction supports only on operand this field is ignored. Note that since only 8 registers are present in the architecture, bits 20 and 19 are not used.
Bits 15..11 are the address of the first operand's source register. Note that since only 8 registers are present in the architecture, bits 15 and 14 are not used.
Bits 10..6 are the address of the destination's register. Note that since only 8 registers are present in the architecture, bits 10 and 9 are not used.
Bits 5..0 are the opcode of the current instruction performed by Milk.
More details and precise examples can be found further in the text, in instruction set describing section.

Chapter 3

Choices and assumptions

Many of the choices about results to be delivered and computation in general are directly following what is stated in IEEE 754-1985 standard for binary floating-point arithmetic and in IEEE 1754-1994 standard for a 32-bit microprocessor architecture.

IEEE 754 doesn't require that all aspects of the standard are implemented in hardware. So privileged-mode software shall emulate any functionality not present in the hardware.

3.1 instructions scheduling

Instructions can be executed in parallel, and shorter operations can be started while longer ones are still running.
Programmer should take care of data consistency inside registers (for example, he should pay attention not to write data to a register that contains a data that still has to be read).
He also should never write a data to a register where a functional unit is writing back its result at the same time; if so, then Milk will allow the functional unit to write the result, and the data from the core will be discarded.

3.2 NaNs handling

NaNs are handled completely by hardware, without trapping to software routines. Anyway it is possible for the user to ”remove” the logic which takes care of normalizing denormal operands just setting to zero the value of the corresponding VHDL generic. This way some area on chip can be saved, and all the latencies of the FUs are shortened (1 clock cycle less).

NaN transformation: when a SNaN is read by a functional unit, the result is the QNaN obtained by setting the most signifcant bit of the fractional part (quiet bit) to 1, and the invalid operation flag is set. when converting to a narrower destination format excess low-order bits of the operand fraction are discarded. When converting to a wider format, excess low-order bits of the result fraction are set to zero.
If the instruction is a MOV, or anyway a non arithmetic instruction, then the invalid operation flag is not set, and the number is copied as it is (change of sign could be performed if specified by the instruction).
The result of an elaboration involving NaN operands depends is instruction dependent:

- **trunc**
  operand is a NaN, infinity, large argument ($\geq$ 2147483648.0 or
  $\leq$ -2147483649.0): invalid exception is raised.
  If operand is positive result is 2147483647, otherwise result is -2147483648.

- **arithmetic operations with one operand**

  - no NaN operands: for all invalid operations, like sqrt(-1), the result will be the QNaN with sign = zero, exponent = all ones, fraction = all ones.
  - QNaN operand: no exception, result is the QNaN operand.
  - SNaN operand: invalid exception, result is the QNaN obtained by NaN transformation.

- **arithmetic operations with two operands**

  - no NaN operands: for all invalid operations, like 0/0, the result will be the QNaN with sign = zero, exponent = all ones, fraction = all ones.
  - both QNaN operands: no exception, result is the QNaN operand loaded from source register 2.
  - both SNaN operands: invalid exception, result is the QNaN operand obtained from NaN transformation of SNaN in source register 2.
  - one SNaN operand: invalid exception, result is the QNaN obtained by NaN transformation.
  - one QNaN operand: no exception, result is the QNaN operand.

3.3 Underflow definition

Underflow occurs if **both** of the following conditions are verified ("tinyness" and "loss of accuracy"):

- the result has magnitude between zero and the smallest normalized nomber in the destination format (result is **tiny**)
- the correctly rounded result in the destination format suffers a **loss of accuracy**. This happens for example after rounding of the exact tiny value.

Note that this implies that if the exact unrounded value is equal to the rounded one (no rounding error), then the result is not inexact and not causes underflow even if it's "tiny". The same way, any tiny result that leads to a rounding error arises both inexact and underflow exceptions, even if rounded result is zero or the smallest normalized number.

3.4 Traps

Up to now architecture doesn't allow the user to choose whether or not to enable traps for exceptions; they are set to disabled by default. Only "unsupported instruction" exceptions causes a trap to the core using the `cop_exc` signal.

3.5 Rounding modes

Milk currently supports only the default rounding mode, that is *Round towards nearest even*.

Chapter 4

Instruction set

4.1 Instructions description

This chapter describes the machine instructions recognized by Milk coprocessor. The following set of instructions is what the assembler should provide; the syntax is made of an instruction mnemonic followed by destination and source registers.
Abbreviations used:

- dr : destination registers, number in the range 0...31
- sr1, sr2 : source registers, number in the range 0...31 (in case that the instruction only needs one operand, it's simply named sr)
- opc: opcode of Milk instructions

Note that each of the supported instructions' mnemonics ends with a number (coprocessor index), for example, fadd0, fadd1, fmul1, etc.
This is due to the fact that Coffee RISC core supports up to 4 coprocessors, so any of them could be a floating-point unit (FPU), and the one who should actually perform the operation is that indexed by the number specified by the number at the end of the mnemonic. This way, fadd0 is a floating-point addiction to be executed by coprocessor number 0, fadd1 is a floating-point addiction to be executed by coprocessor number 1, and so on (for this reason, in the following pages are explained the instructions related to coprocessor number 0, because the instructions related to the other ones are exactly the same for meaning and syntax, and differs only for the coprocessor index).

fadd0

syntax fadd0 dr, sr1, sr2

description Single-precision floating-point (algebraic) addiction to be executed by coprocessor number 0. The contents of the source registers sr1 and sr2 are added together and the result is placed to the destination register dr. Overflow exception is generated if the result's exponent exceeds 127.Underflow exception is generated if the result's exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever both operands are infinites with opposite signs, or when at least one of the operands is a SNaN.

fsub0

syntax fsub0 dr, sr1, sr2

description Single-precision floating-point (algebraic) subtraction to be executed by coprocessor number 0. The contents of the source register sr2 is subtracted by the one in sr1 (meaning that: rd = sr1 - sr2) and the result is placed to the destination register dr. Overflow exception is generated if the result's exponent exceeds 127.Underflow exception is generated if the result's exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever both operands are infinites with opposite signs, or when at least one of the operands is a SNaN.

fmul0

syntax fmul0 dr, sr1, sr2

description Single-precision floating-point multiplication to be executed by coprocessor number 0. The contents of the source registers sr1 and sr2 are multiplied and the result is placed to the destination register dr. Overflow exception is generated if one of the operands is infinite and the other is a finite number, or if the result's exponent exceeds 127. Underflow exception is generated if the result's exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever one operand is infinite and the other one is null, or when at least one of the operands is a SNaN.

fdiv0

syntax	fdiv0 dr, sr1, sr2
description	Single-precision floating-point division to be executed by coprocessor number 0. The contents of the source register sr1 is divided by the one in sr2 and the result is placed to the destination register dr (meaning that: rd = sr1 / sr2). Overflow exception is generated if dividend is infinite and divisor is zero, or if the result's exponent exceeds 127. Underflow exception is generated if the result's exponent exceeds -150. Together with overflow and underflow, also inexact exception occurs. Invalid operation exception occurs whenever both operands are infinite or both are null, or when at least one of the operands is a SNaN. Division by zero exception is raised when a finite non-null number is divided by a null divisor.

fsqrt0

syntax	fsqrt0 dr, sr
description	Single-precision floating-point square-root to be executed by coprocessor number 0. The content of the source register sr is square-rooted and the result is placed to the destination register dr. Invalid operation exception occurs whenever radicand is negative, or when it's a SNaN.

fabs0

syntax	fabs0 dr, sr
description	Single-precision floating-point absolute value (ABS) to be executed by coprocessor number 0. The absolute value of the content of the source register sr is placed to the destination register dr. Invalid operation exception occurs whenever operand is a NaN.

fmov0

syntax	fmov0 dr, sr
description	The operand has to be moved to another register by coprocessor number 0. The value of the content of the source register sr is moved to the destination register dr.

fneg0

syntax	fneg0 dr, sr
description	Single-precision floating-point sign inversion to be executed by coprocessor number 0. The value of the content of the source register sr is inverted in sign and placed to the destination register dr. Invalid operation exception occurs whenever operand is a NaN.

fnop0

syntax	fnop0
description	No operation is executed.

fcvt.s0

syntax	fcvt.s0 dr, sr
description	Integer to single-precision floating-point conversion to be executed by coprocessor number 0. The value of the content of the source register (considered as an integer) sr is converted into signle precision floating-point format and placed to the destination register dr.

fcvt.w0

syntax	fcvt.s0 dr, sr
description	Single-precision floating-point to integer conversion to be executed by coprocessor number 0. The value of the content of the source register sr (considered as a single precision floating-point number) is converted into integer format and placed to the destination register dr. Invalid operation exception is generated if operand is a NaN, infinity, or a large argument ($\geq$ 2147483648.0 or $\leq$ -2147483649.0). Denormal numbers are flattened to zero, and inexact result exception is raised.

fcCONDITION0

syntax	fcCONDITION0 dr, sr1, sr2
description	Comparison to be executed by coprocessor number 0. The contents of the source registers sr1 and sr2 are compared according to the condition specified in the name of the instruction and the result is placed to the destination register dr. Invalid operation exception occurs when at least one of the operands is a NaN and MSB in the opcode is set; result is unordered. NaN compares unordered with everything including itself. Sign of zero is ignored, so +0 = -0.

4.2 Instruction encoding

Here are listed the instruction words sent from the core to Milk coprocessor.
Notes:

- first 6 bits (bits 31..26) of the instruction words correspond to the **cop** instruction of the Coffee RISC core, and are ignored by Milk
- bits 25..24 of the instruction words specify which one among the 4 coprocessors supported by Coffee RISC is the one to be used. They are the same as bits `c_index[1..0]`
- bits 23..22 are ignored by Milk
- bit 21 specifies wether the istruction has to be executed in single-precision or in double-precision. At the moment only single precision is implemented in hardware
- bits 20..16, 15..11 and 10..6 specify the addresses (inside Milk coprocessor's register file) of operands and result of the instruction. They all are 5-bit fields so that up to 32 internal registers are supported, but actually only 8 registers are implemented inside Milk,so only the 3 least significant bits are considered.
- bits 5..0 represent the opcode of one instruction supported by Milk
- milk internal opcodes opc = 010000 and opc = 010001 are RESERVED

fadd0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 000000

fadd1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 000000

fadd2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 000000

fadd3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 000000

fsub0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 000001

fsub1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 000001

fsub2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 000001

fsub3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 000001

fmul0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 000010

fmul1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 000010

fmul2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 000010

fmul3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 000010

fdiv0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 000011

fdiv1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 000011

fdiv2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 000011

fdiv3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 000011

fsqrt0 dr, sr

111100	00	XX	0 = single precision	XXXXX	sr	dr	opc = 000100

fsqrt1 dr, sr

111100	01	XX	0 = single precision	XXXXX	sr	dr	opc = 000100

fsqrt2 dr, sr

111100	10	XX	0 = single precision	XXXXX	sr	dr	opc = 000100

fsqrt3 dr, sr

111100	11	XX	0 = single precision	XXXXX	sr	dr	opc = 000100

fabs0 dr, sr

111100	00	XX	0 = single precision	XXXXX	sr	dr	opc = 000101

fabs1 dr, sr

111100	01	XX	0 = single precision	XXXXX	sr	dr	opc = 000101

fabs2 dr, sr

111100	10	XX	0 = single precision	XXXXX	sr	dr	opc = 000101

fabs3 dr, sr

111100	11	XX	0 = single precision	XXXXX	sr	dr	opc = 000101

fmov0 dr, sr

111100	00	XX	0 = single precision	XXXXX	sr	dr	opc = 000110

fmov1 dr, sr

111100	01	XX	0 = single precision	XXXXX	sr	dr	opc = 000110

fmov2 dr, sr

111100	10	XX	0 = single precision	XXXXX	sr	dr	opc = 000110

fmov3 dr, sr

111100	11	XX	0 = single precision	XXXXX	sr	dr	opc = 000110

fneg0 dr, sr

111100	00	XX	0 = single precision	XXXXX	sr	dr	opc = 000111

fneg1 dr, sr

111100	01	XX	0 = single precision	XXXXX	sr	dr	opc = 000111

fneg2 dr, sr

111100	10	XX	0 = single precision	XXXXX	sr	dr	opc = 000111

fneg3 dr, sr

111100	11	XX	0 = single precision	XXXXX	sr	dr	opc = 000111

fnop0

111100	00	XX	0 = single precision	XXXXX	XXXXX	XXXXX	opc = 001000

fnop1

111100	01	XX	0 = single precision	XXXXX	XXXXX	XXXXX	opc = 001000

fnop2

111100	10	XX	0 = single precision	XXXXX	XXXXX	XXXXX	opc = 001000

fnop3

111100	11	XX	0 = single precision	XXXXX	XXXXX	XXXXX	opc = 001000

fcvt.s0 dr, sr

111100	00	XX	0 = single precision	XXXXX	sr	dr	opc = 100000

fcvt.s1 dr, sr

111100	01	XX	0 = single precision	XXXXX	sr	dr	opc = 100000

fcvt.s2 dr, sr

111100	10	XX	0 = single precision	XXXXX	sr	dr	opc = 100000

fcvt.s3 dr, sr

111100	11	XX	0 = single precision	XXXXX	sr	dr	opc = 100000

fcvt.w0 dr, sr

111100	00	XX	0 = single precision	XXXXX	sr	dr	opc = 100100

fcvt.w1 dr, sr

111100	01	XX	0 = single precision	XXXXX	sr	dr	opc = 100100

fcvt.w2 dr, sr

111100	10	XX	0 = single precision	XXXXX	sr	dr	opc = 100100

fcvt.w3 dr, sr

111100	11	XX	0 = single precision	XXXXX	sr	dr	opc = 100100

fc.f0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110000

fc.f1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110000

fc.f2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110000

fc.f3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110000

fc.un0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110001

fc.un1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110001

fc.un2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110001

fc.un3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110001

fc.eq0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110010

fc.eq1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110010

fc.eq2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110010

fc.eq3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110010

fc.ueq0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110011

fc.ueq1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110011

fc.ueq2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110011

fc.ueq3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110011

fc.olt0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110100

fc.olt1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110100

fc.olt2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110100

fc.olt3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110100

fc.ult0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110101

fc.ult1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110101

fc.ult2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110101

fc.ult3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110101

fc.ole0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110110

fc.ole1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110110

fc.ole2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110110

fc.ole3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110110

fc.ule0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 110111

fc.ule1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 110111

fc.ule2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 110111

fc.ule3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 110111

fc.sf0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111000

fc.sf1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111000

fc.sf2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111000

fc.sf3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111000

fc.ngle0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111001

fc.ngle1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111001

fc.ngle2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111001

fc.ngle3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111001

fc.seq0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111010

fc.seq1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111010

fc.seq2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111010

fc.seq3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111010

fc.ngl0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111011

fc.ngl1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111011

fc.ngl2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111011

fc.ngl3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111011

fc.lt0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111100

fc.lt1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111100

fc.lt2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111100

fc.lt3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111100

fc.nge0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111101

fc.nge1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111101

fc.nge2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111101

fc.nge3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111101

fc.le0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111110

fc.le1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111110

fc.le2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111110

fc.le3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111110

fc.ngt0 dr, sr1, sr2

111100	00	XX	0 = single precision	sr2	sr1	dr	opc = 111111

fc.ngt1 dr, sr1, sr2

111100	01	XX	0 = single precision	sr2	sr1	dr	opc = 111111

fc.ngt2 dr, sr1, sr2

111100	10	XX	0 = single precision	sr2	sr1	dr	opc = 111111

fc.ngt3 dr, sr1, sr2

111100	11	XX	0 = single precision	sr2	sr1	dr	opc = 111111