Integrated Electronics

REDFINv2 IP-Core Data Sheet

DRL:
CI-No.:

	NAME:	FUNCTION:	DATE:
Prepared:	Jakob Lechner	ASIC/FPGA Object Engineer	
Approved:	Martin Auer	Project Manager	

Valid without signatures if stamped “released”.

	Name:	Function:
Co-Authors:	logged in DMS	
Reviewers:	logged in DMS	
Distribution:	logged in DMS	

Number of pages of main document (incl. cover pages, excl. annexes etc.)**: 64**

Number of pages attached (annexes etc.)**: -**

Document Change Log:

Issue	Modified Pages	Description
1		Initial Issue.
2	§7	Fixed remark on address format.
	§9.2.1.7	Fixed description of Frame Pointer register.
	§9.1	Unified register map for all bus width and data widths. Removed data registers from register map (no longer accessible over the bus interface).
	§9.2	Removed data registers from register map (no longer accessible over the bus interface).
	§3, §5.1, §5.3, §6.2.1, §7, §10.2	Removed support for MRegBus.
	§9.1, §9.2, §6.1.4, §6.2.4	Reorganised registers: RfnControlReg split into RfnControl0Reg and RfnControl1Reg, Renamed UEC flag to UME and moved into RfnErrorReg. Added FF field for reporting memory address where EDAC error occurred.
	§9.2.1.3	Removed Active and Cond flags from RfnStatusReg.
	§6.2.3.1.8, §6.2.1	Removed multi-word bus read/write instructions for sake of simplicity.
	§6.2.3.1.7, §6.2.1, §5.1, §6.2.3.1.1	Removed ldfp instruction. Added more generic lds and sts instructions instead.
	§6.2.3.1	Updated instruction opcodes.
	§6.1.4	Updated description of memory scrubber.
	§6.2.3.1.8, §9.2.1.4	Added bus error flag.
	§5.8	Described feature changes in comparison with REDFINv1.
	§10.1.1	Added description of clock signals.
	§10.3.1	Added description of timing of Instruction & Data Memory interface.
	§7	Mention AHB error responses in case of accesses to unmapped addresses.
	§10.1.2	Added requirements for reset synchronization.
	§10.1.1	Added explanation for necessary timing constraints due to different clock domains.
	§5.2	Added section of configuration options.

TABLE OF CONTENTS

T A B L E O F F I G U R E S

1. INTRODUCTION

1.1 PURPOSE

The objective of the IP-Core User Manual (or Data Sheet (ADS)) is to define how the REDFINv2 IP-Core is supposed to be used.

1.2 SCOPE

The ADS describes the REDFINv2 IP-Core from a user point of view. It is therefore addressed to:

- FPGA Verification Engineers
- FPGA Validation Engineers
- Software Designers
- Customer

In the process-oriented Integrated Management System (certified to ISO9001 and ISO14001) of RUAG Space GmbH (RSA), the ADS is part of the ASIC/FPGA Verification, Coding and Synthesis process B30.10.2.

1.3 ACRONYMS

AD	Applicable Documents
ADP	ASIC/FPGA Development and Control Plan
ADS	ASIC/FPGA Data Sheet
AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
ASIC	Application Specific Integrated Circuit
EDAC	Error Detection And Correction
EEPROM	Electrically Erasable Programmable Read Only Memory
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
INT	Interrupt
I/O	In-/Output
ISA	Instruction Set Architecture
LSB	Least Significant Bit
LUT	Look-Up Table
MSB	Most Significant Bit
N/A	Not Applicable
OBC	On-Board Processor
RAM	Random Access Memory
REDFIN	REDuced instruction set for Fixed-point & INteger arithmetics
RSA	RUAG Space GmbH
SRAM	Static Random Access Memory
SoC	System-on-Chip
TBC	To Be Confirmed
TBD	To Be Defined
VHDL	Very High Speed Integrated Circuit Hardware Description Language

2. DOCUMENTS

2.1 APPLICABLE DOCUMENTS

ARS	D-3267786-SPC-RSA	REDFINv2 IP-Core Requirement Specification

2.2 REFERENCE DOCUMENTS

N/A

2.3 REFERENCE STANDARDS

AMBA	ARM IHI 0011A	AMBA Specification (Rev 2.0)

3. FEATURE SUMMARY

- **Main Functions**
  - Memory Controller: Provides access to the Instruction & Data Memory, which is connected to the REDFINv2 IP-Core. Memory words are EDAC-protected. A scrubber periodically reads memory words and corrects single-bit errors.
  - Instruction Handler: Performs all the core tasks of instruction execution
    - Instruction fetch from the memory
    - Instruction decoding
    - Operand fetching (register & memory operands via the Memory Controller)
    - Execution of control instructions (such as jumps)
    - Write back of data words to the register set and the memory
    - Execution of bus master accesses (AMBA AHB)
  - Arithmetic Logic Unit: Executes all arithmetical, logical and shift instructions. Supports both integer and fixed-point operations. The bit width of data words is configurable via a VHDL generic, ranging from 8 to 64 bits.
  - Bus slave interface: Provides read & write access to the IP-Core's registers and the connected Instruction & Data Memory (AMBA AHB slave).
- **Interfaces**

Memory Interface	For connecting Instruction & Data Memory
Bus Master	AMBA AHB
Bus Slave	AMBA AHB
Clk	Clock signal
Reset	asynchronously asserted, synchronously de-asserted

4. DEFINITIONS

4.1 NAMES

The following conventions are used for all names (for signals, registers and BIOS services some extra conventions are defined below):

- A dollar sign ($) in a name is used as a wildcard representing a number. (If the dollar sign ($) is used in a context it must then be defined somewhere else in the document)
- An asterisk (*) in a name is used as a wildcard representing one or more characters.
- Curly brackets are used to list options in names. E.g. *Sync{A,B}*

4.2 BIT NUMBERING

In this document the following conventions are used:

- The most significant bit in a vector has the highest bit number and the leftmost position in a field.
- The least significant bit in a vector has the lowest bit number and the rightmost position in a field.

4.3 SIGNAL NAMES

The following conventions are used for signal names:

- Signal names are written in italics, *SignalName*.
- Active low signals are named *SignalName_N*.

4.4 NUMBER FORMAT

The following conventions are used for writing numbers:

- Binary numbers are indicated either by the subscript "2"; example: 1001_2, or by the postfix “b”, example: 1001b
- Decimal numbers are indicated by no subscript, e.g. 67, 8723 and 47860.
- Hexadecimal numbers are either indicated by the subscript "16"; example: $F00_{16}$, or by the prefix “0x”, example: 0x03FF.
- Unless the radix is explicitly declared as above the number should be considered to be decimal.
- The decimal point is written with a point, “.”, for all numbers. Separation between number groups (if used) is made by spaces.
- Unless otherwise specified, any binary or hexadecimal number shall be interpreted as unsigned integer.
- Signed numbers will be explicitly defined and shall be interpreted using a two’s complement representation.

- X as ‘Value’ in a memory or register field definition means arbitrary data format.

4.5 REGISTERS

The following convention is used for registers and RAM area entries.

- Register names are underlined: <u>RegisterName</u>.
- Flags of a register are indicated by the name of the register and the flag, separated by a point and underlined: <u>RegisterName.Flag</u>. A flag is only 1 bit wide.
- Fields of a register are indicated by the name of the register and the field, separated by a point and underlined: <u>RegisterName.Field</u>. In contrast to a flag, a field is several bits wide. The width must be defined somewhere else.

4.6 TERMINOLOGY

In this document the following terminology is used:

Asserted	Indicates that a signal is driven to its active state.
Deasserted	Indicates that a signal is driven to its inactive state.
Set	The content of a register bit is set to logic 1.
Clear	The content of a register bit is set to logic 0.

5. FUNCTIONAL OVERVIEW

This chapter describes the general functional behaviour of the REDFINv2 core. In §9 the IP-Core registers are described in detail.

This chapter is divided into:

- **General**
  Describes how the REDFINv2 core works from a user perspective
- **Initialisation**
  Describes the initialisation sequence for the REDFINv2 core
- **Operation/Usage**
  Describes how the REDFINv2 core shall be used during normal operation
- **Interrupt Handling**
  Describes the interrupt handling of the REDFINv2 core
- **Error Handling and Correction Procedures**
  Describes how to handle errors in the REDFINv2 core
- **Usage Constraints**
  Describes actions that are not allowed and the resulting consequences

5.1 GENERAL

The REDFINv2 IP-Core is a light-weight processing core mainly designed for evaluation of arithmetic expressions and simple control tasks. The REDFINv2 instruction set architecture offers a configurable bit width for the data path, ranging from 8 to 64 bits. The desired configuration can be set using a VHDL generic. Throughout this document the data width will be denoted by *ABW*. Instruction words, on the other hand, have a fixed width of 16 bits. Figure 1 shows the block diagram of a REDFINv2 core and its integration into a System-on-Chip (SoC) design. As can be seen the REDFINv2 core can be connected to a AMBA AHB bus system. The REDFINv2 core acts both as a bus master and a bus slave. Via the latter interface other bus masters in the SoC can access the registers and the memory of the REDFINv2 core and thus control its execution, via the master interface the core itself is capable to actively control other system components.

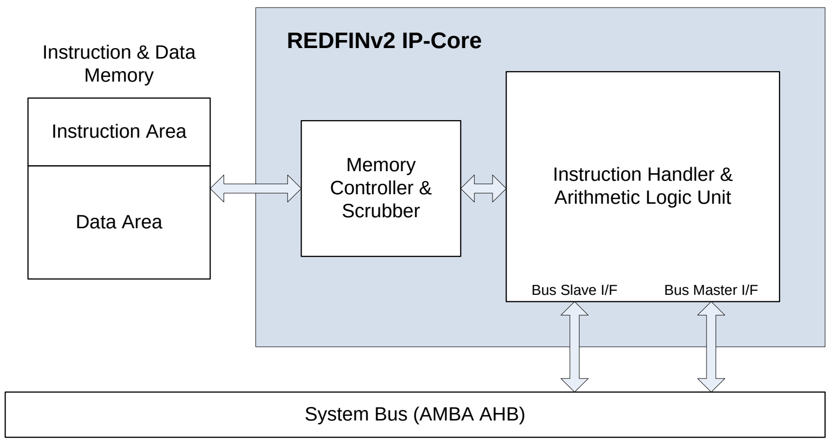
Instruction & Data Memory
Instruction Area
Data Area
REDFINv2 IP-Core
Memory Controller & Scrubber
Instruction Handler & Arithmetic Logic Unit
Bus Slave I/F
Bus Master I/F
System Bus (AMBA AHB)

Figure 1 – REDFINv2 IP-Core block diagram

The REDFINv2 Core interfaces an Instruction & Data Memory, which is organised in two areas, one for instruction words and one for data words. The number of addressable memory words is configurable at compile-time and depends on the number of instruction and data words that shall be available. In the current implementation the REDFINv2 core can address a maximum of 2^{16} instructions in the Instruction Area and up to 2^{ABW} + 255 data words in the Data Area of the memory. The configured number of instruction words will be denoted by *InstrWC* in this document, the number of data words by *DataWC*.

The REDFINv2 provides a register set with 4 *ABW*-bit general purpose registers. As can be seen in Figure 2, the REDFINv2 core is based on a register-memory architecture, i.e. instructions can fetch their operands from register set as well as directly from the Data Area of the connected Instruction & Data Memory.

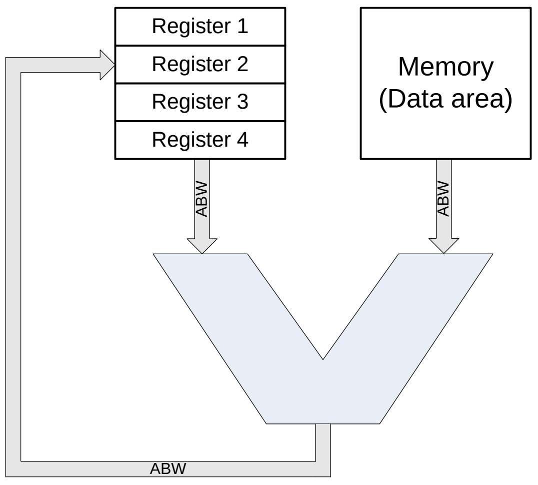

Figure 2 – Register-memory architecture

The instruction set includes the following type of operations:

- Arithmetic operations: add, subtract, multiply and divide. There are arithmetic instructions for integer and fixed-point operations available. In the latter case the operands are interpreted as signed fixed-point numbers, represented in two's complement format. The number of fractional bits can be adjusted via the register field RfnControl1Reg.FracLen. If this field is set to zero, the operands are interpreted as regular signed integer numbers and the fixed-point instructions therefore behave exactly like the integer counterparts.
- An instruction for efficient evaluation of polynomials is provided (pmac). This instruction always assumes fixed-point operands.
- Logical instructions like and, or, xor, etc.
- Shift instructions
- Comparison instructions
- Register initialisation instructions
- Load/store instructions for moving data words between the general purpose registers and the local data memory
- Load/store instructions for moving data values between special-purpose registers and the local data memory

- Bus access instructions for performing read & write accesses on the AMBA AHB bus to which the REDFINv2 core is connected to as a master component.
- Control instructions

5.2 CONFIGURATION

The REDFINv2 IP core offers various configuration parameters, which can be adjusted when integrating the IP core into a larger design. The parameters can be set via VHDL generics, as listed in Table 1.

Generic name	Short identifier[1]	Description	Valid settings	Default
iDataWidth_G	*ABW*	Bit width of data path.	8 to 64	32
iTotalMemDW_G	-	Gross width of data buses of Instruction & Data Memory (with parity bits).	• if iMemDW_G = 16: 22 • if iMemDW_G = 32: 39	22
iMemDW_G	*MemDW*	Net width of data buses of Instruction & Data Memory (without parity bits).	16, 32	16
iMemAW_G	*MemAW*	Width of address bus of Instruction & Data Memory. *MemAW* needs to be set to ceil($\log_2$(*MemSize*)). The overall memory size *MemSize* depends on various other configuration parameters (refer to §8.1).	≥ 1	12
iInstrWC_G	*InstrWC*	Defines the number of 16-bit instruction words that can be stored in the Instruction Area of Instruction & Data Memory.	≥ 1	1024
iDataWC_G	*DataWC*	Defines the number of *ABW*-bit instruction words that can be stored in the Instruction Area of Instruction & Data Memory.	≥ 1	1024
bFastMult_G	-	Use fast or slow multiplier for Sequencers. Fast multipliers require DSP elements in case of an FPGA.	• False: Slow multiplier • True: Fast multiplier	True
bFastShift_G	-	Use fast or slow shifters for Sequencers. Fast shifters require DSP elements in case of an FPGA implementation.	• False: Slow shifter • True: Fast shifter	True

[1] Short names for generics used throughout this document to improve readability.

iScrubberDiv_G	-	Clock divider for adjusting the scrubber period for the Instruction & Data Memory. One *MemDW*-bit memory word is scrubbed per scrubber period. Therefore, the total time for scrubbing the entire memory is equal to *MemSize* * iScrubberDiv_G *Clk* periods. The overall memory size *MemSize* depends on various other configuration parameters (refer to §8.1).	Depends on radiation environment and size of memories. The minimum setting of 100 *Clk* periods.	5000
BusBaseAddr_G	*BBA*	AHB base address of the first register. This parameter needs to be specified as a byte address.	0 to 2^{32} – “number of mapped bus addresses” (refer to §7.1)	0
iBusDW_G	BusDW	Width of the data buses of the AHB bus I/F.	16, 32 (needs to be identical to iMemDW_G).	16
iClk2AHBClkRatio_G	-	*Clk*-to-*AHBClk* ratio: *Clk* and *AHBClk* are expected to be ratiochronous. Hence, the frequency of the *Clk* signal can be an integer multiple of the frequency of the *AHBClk* signal. Refer to §10.1.1 for more details on clocking.	1 to 16	1
bSynPreserveFSM_G	-	Synthesis attribute for preserving one-hot encoding of FSM state registers. This can needs to be enabled for building fault-tolerant state machines.	False, True	True
bSynReplicateFSM_G	-	Synthesis attribute for allowing replication of FSM state vectors. This needs to be disabled for building fault-tolerant FSMs.	False, True	False

Table 1 – REDFINv2 VHDL Generics

5.3 INITIALISATION

After the de-assertion of the reset signal, the Instruction and Data Memory is initialised with data values of 0 by the REDFINv2 core. The duration of this memory initialisation is discussed in §6.1.2. Otherwise, no functional initialisation is performed. To be able to execute subroutines, the Instruction & Data Memory still needs to be initialised by an external controller using the core's bus slave interface (AMBA AHB).

5.4 OPERATION / USAGE

5.4.1 Operation Modes

Figure 3 depicts the operational modes of the REDFINv2 IP-Core. An explanation of these modes is given in the following subsections.

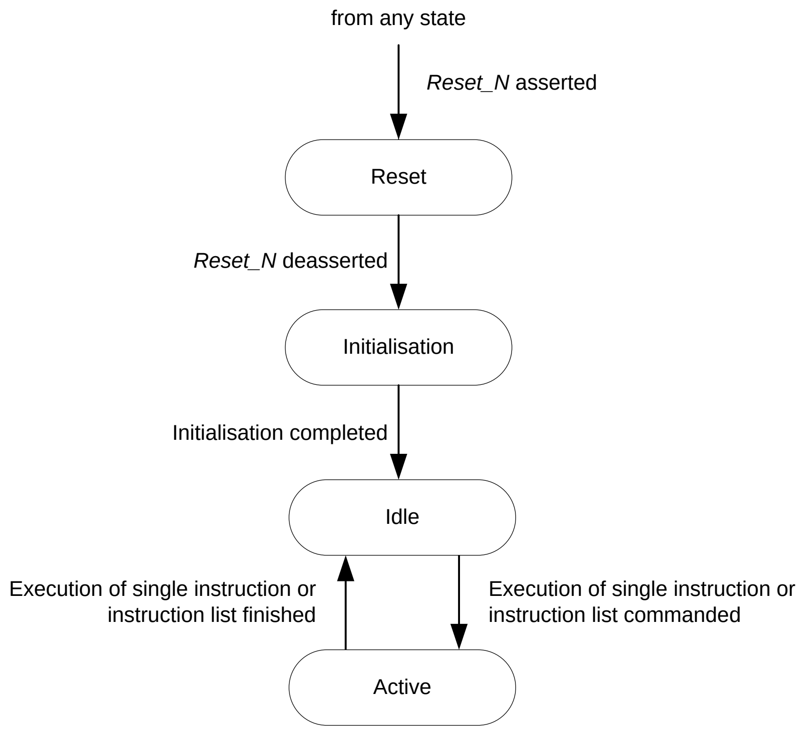
from any state
Reset_N asserted
Reset
Reset_N deasserted
Initialisation
Initialisation completed
Idle
Execution of single instruction or instruction list finished
Execution of single instruction or instruction list commanded
Active

Figure 3 – REDFINv2 core operation modes

5.4.1.1 Reset

After power on, the REDFINv2 core has to be reset to reach a defined state. The Reset mode is entered when *Reset_N* input is asserted. It sets all internal registers and output signals to their default values.

5.4.1.2 Initialisation

After the de-assertion of the reset signal, the Instruction and Data Memory is initialised with data values of 0 by the REDFINv2 core. This ensures that all parity bits are correct and therefore no memory errors will be reported when the memory scrubber is turned on.

5.4.1.3 Idle

After completion of the Instruction & Data Memory initialisation, the REDFINv2 core resides in *Idle* mode and waits for commands by an external controller. Single instructions can be commanded immediately after reset, for execution of subroutines, however, the Instruction Area of the memory first needs to be initialised.

5.4.1.4 Active

When the execution of a single instruction or a subroutine is commanded, the REDFINv2 core enters the active mode and stays in this mode until the execution has been finished, or the execution is pre-maturely aborted by another command.

5.4.2 Clocking and Reset

For information about how to clock the REDFINv2 core, see §10.1.1, for reset see §10.1.2.

5.5 INTERRUPT HANDLING

No interrupt handling is implemented in the REDFINv2 core.

5.6 ERROR HANDLING AND/OR CORRECTION PROCEDURES

For errors related to accesses of the Instruction & Data Memory, please refer to §6.1.4. Errors related to the execution of instructions and subroutines are described in §6.2.4.

5.7 USER CONSTRAINTS

TBD

5.8 FEATURE CHANGES COMPARED TO REDFINV1

- Instruction & Data Memory is no longer internally instantiated inside REDFIN core. The REDFINv2 entity now has a memory interface where a synchronous SRAM memory needs to be attached. During a read access the connected memory block is expected to synchronously sample the address and provide read data in the next clock cycle. Size and structure of connected memory needs to be defined with the following VHDL Generics (refer to §8 for details):
  - iMemAW_G: Width of address bus.
  - iMemDW_G: Width of data bus (16 or 32 bit supported).
  - iInstrWC_G: Number of accessible instruction words.
  - iDataWC_G: Number of accessible data words.
- Sublists are now called subroutines.
- Redesign of bus interfaces:
  - All MRegBus interfaces have been removed (simple slave for register access, complex slave for memory access, master interface).
  - AHB slave interface added for accessing REDFINv2 registers and Instruction & Data Memory.
  - Instruction & Data Memory can be accessed concurrently with instruction/subroutine execution (no more ILMA errors).

    - Bus read/write instructions now implement AHB master interface.
    - Bus read/write instructions simplified: Only a single-word bus access is performed. Data width of transferred word depends on configured bus width (VHDL generic iBusDW_G = 16 or 32).
- Instruction for EEPROM read access removed.
- Frame Pointer register added to extend number of addressable data words. If frame pointer is set to 3, e.g. the instruction “add $R0, 2” would fetch the data word 3+2=5 from memory and add it to register 0.
- Load/store instructions for special purpose registers added (lds, sts).
- Redesign of REDFIN registers:
    - Prefix “Arith” replaced by “Rfn” in register names.
    - RfnControlReg split into RfnControl0Reg and RfnControl1Reg.
    - Cond and Sign flags removed from RfnStatusReg.
    - Global error flag removed from status register
    - Memory Initialisation complete flag added to status register.
    - RfnEDACReportReg split into RfnEDACReport[0,1]Reg.
    - UEC flag renamed to UME and moved to RfnErrorReg.
    - General purpose registers R$ no longer accessible via bus slave interface.
    - ArithRAMAddrReg and ArithRAMDataReg for indirect access to Instruction & Data memory removed.
- Supported data width extended: 8 <= ABW <= 64.
- Fast/Slow multiplier selectable by VHDL generic.
- Fast/Slow shifter selectable by VHDL generic.
- Scrubber rate for Instruction & Data memory configurable by VHDL generic ScrubberDiv_G.
- Abs instruction added (computes absolute value of the number stored in a register).
- Jump absolute instruction added (jumps to absolute instruction address).

6. FUNCTIONAL BEHAVIOUR

This chapter describes the detailed functional behaviour of the REDFINv2 IP-Core modules. In the following subsections each functional module is separately described, and in §9 the IP-Core registers are described in detail.

Most module subsections are divided into:

- **General**
  Describes how the module works from a user perspective.
- **Initialisation**
  Describes the initialisation sequence for the module (if applicable)
- **Operation/Usage**
  Describes how each module shall be used during normal operation
- **Error Handling**
  Describes how to handle internal errors in the module.
- **Usage Constraints**
  Describes actions that are not allowed and the resulting consequences.
- **Examples (optional)**
  Gives a few examples how to perform different tasks using the module. General

6.1 MEMORY CONTROLLER

6.1.1 General

The Memory Controller provides access to the Instruction & Data Memory connected to the REDFINv2 IP-Core. The stored memory words will be protected by a Hamming code, which is capable to correct single-bit errors and detect double-bit errors. A scrubber can be enabled which periodically reads memory locations in order to detect and correct single-bit errors in the memory.

6.1.2 Initialisation

The entire Instruction & Data Memory is autonomously filled with 0 immediately after the reset is de-asserted. Consequently, all memory cells will have correct EDAC after the initialisation process has been finished.

The scrubber should normally be used, but default state after reset is off since it would report false errors until the Instruction & Data Memory has been initialised to 0.

The duration of the memory initialisation depends on the number of the stored instruction & data words and the memory organisation (refer to §8). The following list specifies the memory initialisation times for different IP-Core configurations:

- *MemDW*=16 and 8 ≤ *ABW* ≤ 16: *InstrWC* + *DataWC* + 1 clock cycles
- *MemDW*=16 and 17 ≤ *ABW* ≤ 32: *InstrWC* + *DataWC**2 + 1 clock cycles
- *MemDW*=16 and 33 ≤ *ABW* ≤ 48: *InstrWC* + *DataWC**3 + 1 clock cycles
- *MemDW*=16 and 49 ≤ *ABW* ≤ 64: *InstrWC* + *DataWC**4 + 1 clock cycles
- *MemDW*=32 and 8 ≤ *ABW* ≤ 32: *InstrWC*/2 + *DataWC* + 1 clock cycles
- *MemDW*=16 and 33 ≤ *ABW* ≤ 64: *InstrWC*/2 + *DataWC**2 + 1 clock cycles

6.1.3 Operation/Usage

The Memory Controller is used by the Instruction Handler of the REDFINv2 core. When accessing an external controller wants to access the Instruction & Data Memory the bus slave interface, provided by the REDFINv2 core, needs to be used. In this case the Instruction Handler pass on the bus request to the Memory Controller, making sure that any ongoing instruction execution is not impeded.

The worst-case latency of a bus access to the Instruction & Data Memory is x (TBD) clock cycles.

6.1.4 Error Handling

The memory words stored in the connected RAM are EDAC protected. The EDAC is of a Hamming type, correcting all single bit errors and detecting all double bit errors. Detected EDAC errors are indicated in the registers RfnErrorReg, RfnEDACReport0Reg and RfnEDACReport1Reg, separately for correctable and non-correctable errors. The number of corrected errors, saturated at 15, is provided by the field RfnEDACReport0Reg.CEC. The non-correctable error flags RfnErrorReg.UME as well as the CEC field can be cleared by overwriting the values with 0.

The scrubbing function, meaning reading, checking and possibly correcting SRAM content, can be enabled/disabled via RfnControl0Reg.SCRB and is triggered periodically. Each time the scrubber is triggered a single memory word is fetched and corrected, if necessary. If an instruction/subroutine is currently executed when the scrubber is triggered, the scrubbing activity is deferred until the end of executed instruction. The scrubbing frequency can be determined at synthesis-time with the VHDL generic *iScrubberDiv_G*. This generic configures a clock divider, which allows for setting the frequency of scrubber activations to *f_clock* / *iScrubberDiv_G*, where *f_clock* denotes the frequency of the main clock of the REDFINv2 core. The length of a full scrubber cycle for scrubbing the entire memory depends on the number of the stored instruction & data words and the memory organisation (refer to §8). The scrubber cycle for checking and potentially correcting all memory words can be determined as follows:

- *MemDW*=16 and 8 ≤ *ABW* ≤ 16: (*InstrWC* + *DataWC*) * *iScrubberDiv_G* / *f_clock*
- *MemDW*=16 and 17 ≤ *ABW* ≤ 32: (*InstrWC* + *DataWC**2) * *iScrubberDiv_G* / *f_clock*
- *MemDW*=16 and 33 ≤ *ABW* ≤ 48: (*InstrWC* + *DataWC*3*) * *iScrubberDiv_G* / *f_clock*
- *MemDW*=16 and 49 ≤ *ABW* ≤ 64: (*InstrWC* + *DataWC*4*) * *iScrubberDiv_G* / *f_clock*
- *MemDW*=32 and 8 ≤ *ABW* ≤ 32: (*InstrWC*/2 + *DataWC*) * *iScrubberDiv_G* / *f_clock*
- *MemDW*=16 and 33 ≤ *ABW* ≤ 64: (*InstrWC*/2 + *DataWC**2) * *iScrubberDiv_G* / *f_clock*

Note that memory errors encountered during a scrubber accesses are also reported in register RfnErrorReg and RfnEDACReport$Reg.

The SBTW and DBTW bits in conjunction with the UEAddr field in the RfnEDACInjectReg can be used to force EDAC errors in order to test the error handling functionality. The MSB data input to the SRAM is inverted if the value of SBTW is 1 and the written SRAM address is equal to the specified error address in RfnEDACInjectReg.UEAddr. A later read access from this memory address will then suffer from a single-bit error.

The same procedure applies to the DBTW function where two MSBs of the data vector are inverted when writing to the memory (thus producing a double-bit error at a later read access). The EDAC function shall not be used when scrubber is enabled. Otherwise it might occur that the injected error is corrected unintentionally. Furthermore, it is recommended to trigger the enhanced EDAC test function (by writing the RfnEDACInjectReg) only in case no memory access is ongoing. Otherwise it might occur that the injection operation is quit although not executed. Both the SBTW and the DBTW flag shall be left in default state (0) during normal operation.

6.1.5 Usage Constraints

N/A

6.2 INSTRUCTION HANDLER & ARITHMETIC LOGICAL UNIT

6.2.1 General

The Instruction Handler together with the Arithmetic Logic Unit (ALU) performs all tasks that are necessary to execute the instructions defined by the REDFINv2 instruction. Furthermore, it controls the execution of instruction sequences stored in the memory (subroutines):

- Fetching of instruction words from the Instruction & Data Memory
- Instruction decoding
- Fetching of register and memory operands
- Execution of instructions (with the help of the ALU)
- Write back of data words to the register set and the memory
- Execution of bus master accesses (AMBA AHB master)

Table 2 shows all available instructions in an overview. A detailed description of every instruction can be found in §6.2.3.1.

Instruction	Description
add rX, memoffset	rX = rX + [FP + memoffset]
add_si rX, simm	rX = rX + simm
fadd rX, memoffset	rX = rX + [FP + memoffset]
fadd_si rX, simm	rX = rX + simm
sub rX, memoffset	rX = rX – [FP + memoffset]
sub_si rX, simm	rX = rX – simm
fsub rX, memoffset	rX = rX – [FP + memoffset]
fsub_si rX, simm	rX = rX – simm
mul rX, memoffset	rX = rX * [FP + memoffset]
mul_si rX, simm	rX = rX * simm
fmul rX, memoffset	rX = rX * [FP + memoffset]
fmul_si rX, simm	rX = rX * simm
div rX, memoffset	rX = rX / [FP + memoffset]
div_si rX, simm	rX = rX / simm
fdiv rX, memoffset	rX = rX / [FP + memoffset]
fdiv_si rX, simm	rX = rX / simm
abs rX	rX = abs(rX)

pmac uimm1, uimm2	r1 = r1 * var1^uimm1 * var2^uimm2
	r0 = r0 + r1

and rX, memoffset	rX = rX & [FP + memoffset]
or rX, memoffset	rX = rX | [FP + memoffset]
xor rX, memoffset	rX = rX xor [FP + memoffset]
not rX	rX = ~rX
sl rX, memoffset	rX = rX << [FP + memoffset]
sl_i rX, uimm	rX = rX << uimm
sr rX, memoffset	rX = rX >> [FP + memoffset]
sr_i rX, uimm	rX = rX >> uimm
sra rX, memoffset	rX = (int) rX >> [FP + memoffset]
sra_i rX, uimm	rX = (int) rX >> uimm
cmpeq rX, memoffset	cond = (rX == [FP + memoffset])
cmplt rX, memoffset	cond = (rX < [FP + memoffset])
cmpgt rX, memoffset	cond = (rX > [FP + memoffset])
ld_si rX, simm	rX = (int) simm
ld_i rX, uimm	rX = uimm
ld rX, memoffset	rX = [FP + memoffset]
ldmi rX, memoffset	rX = [[FP + memoffset]]
st rX, memoffset	[FP + memoffset] = rX
stmi rX, memoffset	[[FP + memoffset]] = rX
lds memoffset	rS = [FP + memoffset]
sts rX, memoffset	[FP + memoffset] = rS
brd rX, memoffset	rX(*BusDW*-1...0) = bus([FP + memoffset])
bwr rX, memoffset	bus([FP + memoffset]) = rX(*BusDW*-1...0)
wait uimm	nop for uimm clock cycles
jmp memoffset	InstructionCounter = [FP + memoffset]
jmpi simm	InstructionCounter += simm + 1
jmpi_ct simm	if cond : InstructionCounter += simm + 1
jmpi_cf simm	if ~cond : InstructionCounter += simm + 1
halt	Stop the execution of a subroutine

Table 2 – Instruction Set Overview

6.2.2 Initialisation

When subroutines (see §6.2.3) are used the respective instructions first need to be loaded into the instruction area of the Instruction & Data Memory. Otherwise no initialisation of the REDFINv2 core is required.

6.2.3 Operation/Usage

The REDFINv2 core provides two modes of operation:

1. Execution of single instructions
2. Execution of subroutines, which are sequences of instructions.

Single instructions can be executed by writing the respective 16-bit instruction word to register RfnInstructionReg. This triggers the execution of that instruction, which takes a fixed (data-independent) number of clock cycles. Afterwards, the REDFINv2 Core returns to its idle state and waits for further instructions to be commanded.

When executing a subroutine the REDFINv2 core fetches and executes the subroutine instructions from the Instruction & Data Memory. For triggering the execution of a subroutine the address of the first instruction of the desired subroutine needs to be written to register RfnStartsubReg. Note that a written address is interpreted as an instruction address, i.e. an address value of 0 references the first instruction at the beginning of the Instruction Area of the memory, an address value of 1 references the second instruction, etc.

After each fetched instruction, the Instruction Pointer is incremented by 1 and the next instruction is fetched from the referenced memory location. Jump instructions allow for modification of the instruction pointer and thus implement jumps within the subroutine. The subroutine execution ends when a *halt* instruction is fetched from the memory. If the user fails to terminate a subroutine with a *halt* instruction, instructions might be fetched and executed (if legal) until the last entry of the instruction area in the memory is reached. In this case an unexpected end of subroutine violation is raised (flag RfnErrorReg.UEOS set to ‘1’) and any further processing is aborted.

An ongoing subroutine execution can also be terminated by the external controller when either a new instruction is written to RfnInstructionReg or the start of a new subroutine is triggered by writing to RfnStartsubReg. Writing a *halt* instruction to RfnInstructionReg therefore will always stop any activity of the REDFINv2 Core.

Note: A subroutine may contain arbitrary instructions from the REDFINv2 instruction set.

For simple iterative tasks with a pre-defined constant number of iterations, the REDFINv2 Core can be instructed to execute a subroutine multiple times upon a single RfnStartsubReg write. The number of iterations is defined by the value of register field RfnControl0Reg.SubIter at the time the subroutine execution starts. Note that the field value is incremented by one by the REDFINv2 Core, i.e. with a value of zero a subroutine will be executed once.

Input operands for a subroutine can be *pushed* by an external controller to the REDFINv2 core over the bus slave interface, either by writing the four general purpose registers or by initialising relevant data words in the Instruction & Data Memory. The location of the inputs clearly depends on the specific subroutine that is executed. Alternatively, the subroutine can be programmed to *pull* inputs from external subsystems by accessing relevant bus addresses using the available bus read instructions.

For the provisioning of computation results by the REDFINv2 core similar strategies can be employed: The outputs of a subroutine execution could be stored in the general purpose registers or in the Data Area of the memory. An external controller then needs to *pull* the stored data words from these storage locations, as soon as they are ready. Alternatively, the REDFINv2 core can *push* outputs to external subsystems by writing relevant bus addresses using its bus write instructions.

6.2.3.1 Instruction Set

6.2.3.1.1 Definitions

Depending on the type of the instruction, different instruction formats are used, see Table 3. The most significant 6-bits define the opcode, and the fields REG2 and MEM8 encode 2-bit register addresses and 8-bit memory address offsets, respectively. For most instructions the REG2 field contains a reference to one of the four general purpose registers. In case of the *load special* and *store special* instructions, however, one of the special-purpose registers will be accessed (refer to lds/sts instructions in §6.2.3.1.7).

The SIMMx and UIMMx fields encode signed and unsigned immediate values of various bit widths. Note that in Table 3 opcode prefixes for each instruction format are illustrated with grey background.

	15					10	9 8	7 0
TYPE A	0	0	0	0	0	0	-	
TYPE B[2]	0	X	X	X	X	X	REG2/-	MEM8
TYPE C	1	0	0	X	X	X	REG2	SIMM8
TYPE D	1	0	1	0	X	X	REG2	SIMM8
TYPE E	1	0	1	1	X	X	REG2	UIMM8
TYPE F	1	1	0	X	X	X	UIMM10/SIMM10	
TYPE G	1	1	1	0	X	X	REG2	-
TYPE H	1	1	1	1	1	1	UIMM5	UIMM5

Table 3 – Instruction Formats

The REDFINv2 core uses a frame pointer for addressing of data words in the memory. Therefore, MEM8 address offsets always have to be interpreted relative to the current frame pointer value *FP*. If *FP* = 3, e.g. and the MEM8 field is assigned to 2, the instruction would operate on the 5th *ABW*-bit data word in the Data Area of the Instruction & Data Memory.

Instruction addresses are interpreted relative to the start of the Instruction Area in the memory. A jump to address 0, e.g. continues the subroutine execution from the first instruction at the beginning of the memory, in case of a jump to address 1 the second instruction located in the memory is executed.

Note that the specific physical addresses of instruction and data words depend on the various configuration parameters of the REDFINv2 core and the resulting organisation of the Instruction & Data Memory (refer to §8 for further details).

[2] Type_B format includes all opcodes with a leading 0, except for the all-zero opcode (Type_A format).

The following notations are used throughout the following sections:

- rX: Register reference, X=0,…,3.
- memoffset: Placehoder for a word address offset, relative to the frame pointer, for referencing an *ABW*-bit data word in the data area of the Instruction & Data Memory.
- [FP + memoffset]: Value of the *ABW*-bit data word in the data area of the Instruction & Data Memory, located at the address which is obtained when adding the current frame pointer value with *memoffset*.
- uimm: unsigned immediate integer value
- simm: signed immediate integer value
- bus(*A*): Value of *BusDW*-bit word located at the bus address *A*.

6.2.3.1.2 Arithmetic Instructions

Add	Signed addition of memory word to register value. Operands and result are interpreted as signed integer numbers. add rX, memoffset rX = rX + [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	0	0	1	0	0	rX	memoffset	

fadd	Signed addition of memory word to register value. Operands and result are interpreted as signed fixed-point numbers or signed integer numbers, depending on RfnConfigReg.FracLen. fadd rX, memoffset rX = rX + [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	0	1	1	0	0	rX	memoffset	

add_si	Signed addition of immediate value to register value. The register operand and the encoded immediate value are interpreted as signed integer numbers. add_si rX, simm rX = rX + simm							
Instruction word (Type C):								
15					10	9 8	7	0
1	0	0	0	0	0	rX	simm	

fadd_si	Signed addition of immediate value to register value. The register operand is interpreted as signed fixed-point number, depending on RfnConfigReg.FracLen. The encoded immediate is interpreted as a signed integer value and will be converted to a fixed-point number before the computation. The result will also be stored as fixed-point number (format as determined by RfnConfigReg.FracLen).	
	add_si rX, simm	rX = rX + simm

Instruction word (Type D):

15					10	9 8	7 0
1	0	1	0	0	0	rX	simm

sub	Signed subtraction of memory value from register value. Operands and result are interpreted as signed integer numbers.	
	sub rX, memoffset	rX = rX – [FP + memoffset]

Instruction word (Type B):

15					10	9 8	7 0
0	0	0	1	0	1	rX	memoffset

fsub	Signed subtraction of memory value from register value. Operands and result are interpreted as signed fixed-point numbers, depending on RfnConfigReg.FracLen.	
	fsub rX, memoffset	rX = rX – [FP + memoffset]

Instruction word (Type B):

15					10	9 8	7 0
0	0	1	1	0	1	rX	memoffset

sub_si	Signed subtraction of immediate value from register value. The register operand and the encoded immediate value are interpreted as signed integer numbers.	
	sub_si rX, simm	rX = rX – simm

Instruction word (Type C):

15					10	9 8	7 0
1	0	0	0	0	1	rX	simm

fsub_si	Signed subtraction of immediate value from register value. The register operand is interpreted as signed fixed-point number, depending on RfnConfigReg.FracLen. The encoded immediate is interpreted as a signed integer value and will be converted to a fixed-point number before the computation. The result will also be stored as fixed-point number (format as determined by RfnConfigReg.FracLen). fsub_si rX, simm rX = rX – simm							
Instruction word (Type D):								
15					10	9 8	7 0	
1	0	1	0	0	1	rX	simm	

mul	Signed multiplication of register value and memory word. Operands and result are interpreted as signed integer numbers. mul rX, memoffset rX = rX * [FP + memoffset]						
Instruction word (Type B):							
15					10	9 8	7 0
0	0	0	1	1	0	rX	memoffset

fmul	Signed multiplication of register value and memory word. Operands and result are interpreted as signed fixed-point numbers, depending on RfnConfigReg.FracLen. fmul rX, memoffset rX = rX * [FP + memoffset]						
Instruction word (Type B):							
15					10	9 8	7 0
0	0	1	1	1	0	rX	memoffset

mul_si	Signed multiplication of register value with immediate value. The register operand and the encoded immediate value are interpreted as signed integer numbers. mul_si rX, simm rX = rX * simm						
Instruction word (Type C):							
15					10	9 8	7 0
1	0	0	0	1	0	rX	simm

fmul_si	Signed multiplication of register value with immediate value. The register operand is interpreted as signed fixed-point number, depending on RfnConfigReg.FracLen. The encoded immediate is interpreted as a signed integer value and will be converted to a fixed-point number before the computation. The result will also be stored as fixed-point number (format as determined by RfnConfigReg.FracLen). fmul_si rX, simm — rX = rX * simm							
Instruction word (Type D):								
15					10	9 8	7	0
1	0	1	0	1	0	rX	simm	

div	Signed division of register value by memory word. Operands and result are interpreted as signed integer numbers. div rX, memoffset — rX = rX / [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	0	0	1	1	1	rX	memoffset	

fdiv	Signed division of register value by memory word. Operands and result are interpreted as signed fixed-point numbers, depending on RfnConfigReg.FracLen. fdiv rX, memoffset — rX = rX / [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	0	1	1	1	1	rX	memoffset	

div_si	Signed division of register value by immediate value. The register operand and the encoded immediate value are interpreted as signed integer numbers. div_si rX, simm — rX = rX / simm							
Instruction word (Type C):								
15					10	9 8	7	0
1	0	0	0	1	1	rX	simm	

fdiv_si	Signed division of register value by immediate value. The register operand is interpreted as signed fixed-point number, depending on RfnConfigReg.FracLen. The encoded immediate is interpreted as a signed integer value and will be converted to a fixed-point number before the computation. The result will also be stored as fixed-point number (format as determined by RfnConfigReg.FracLen). fdiv_si rX, simm					rX = rX / simm		
Instruction word (Type D):								
15					10	9 8	7 0	
1	0	1	0	1	1	rX	simm	

abs	Computes absolute value of a register. Both integer and fixed-point values are supported. abs rX					rX = abs(rX)	
Instruction word (Type G):							
15					10	9 8	7 0
1	1	1	0	0	1	rX	-

pmac	Extended multiply-accumulate operation for computation of polynomial terms with two variables. The *coefficient* of the term needs to be provided in register r1, *Variable 1* and *Variable 2* are loaded from the memory data words with addresses FP+0x00 and FP+0x01, respectively. The result of the evaluated term is accumulated in register *r0*. The coefficient, the variables and the result are interpreted as signed fixed-point numbers (format depending on RfnConfigReg.FracLen). pmac uimm1, uimm2					r1 = r1 * var1^uimm1 * var2^uimm2; r0 = r0 + r1	
Instruction word (Type H):							
15					10	9 5	4 0
1	1	1	1	1	1	uimm1	uimm2

6.2.3.1.3 Logical Instructions

and	Bitwise AND of register value and memory word. and rX, memoffset					rX = rX & [FP + memoffset]	
Instruction word (Type B):							
15					10	9 8	7 0
0	0	0	0	0	1	rX	memoffset

or	Bitwise OR of register value and memory word. or rX, memoffset rX = rX | [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	0	0	0	1	0	rX	memoffset	

xor	Bitwise XOR of register value and memory word. xor rX, memoffset rX = rX xor [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	0	0	0	1	1	rX	memoffset	

not	Bitwise negation of register value. not rX rX = ~rX							
Instruction word (Type G):								
15					10	9 8	7	0
1	1	1	0	0	0	rX	-	

6.2.3.1.4 Shift Instructions

sl	Logical left shift of register value by number of bits specified in memory word. sl rX, memoffset rX = rX << [FP + memoffset]							
Instruction word (Type B):								
15					10	9 8	7	0
0	1	1	0	0	0	rX	memoffset	

sl_i	Logical left shift of register value by number of bits specified by unsigned immediate value. sl_i rX, uimm rX = rX << uimm							
Instruction word (Type E):								
15					10	9 8	7	0
1	0	1	1	0	0	rX	uimm	

sr	Logical right shift of register value by number of bits specified in memory word. sr rX, memoffset					rX = rX >> [FP + memoffset]	
Instruction word (Type B):							
15					10	9 8	7 0
0	1	1	0	0	1	rX	memoffset

sr_i	Logical right shift of register value by number of bits specified by unsigned immediate value. sr_i rX, uimm					rX = rX >> uimm	
Instruction word (Type E):							
15					10	9 8	7 0
1	0	1	1	0	1	rX	uimm

sra	Arithmetic right shift of register value by number of bits specified in memory word. sra rX, memoffset					rX = (int) rX >> [FP + memoffset]	
Instruction word (Type B):							
15					10	9 8	7 0
0	1	1	0	1	0	rX	memoffset

sra_i	Arithmetic right shift of register value by number of bits specified by unsigned immediate value. sra_i rX, uimm					rX = (int) rX >> uimm	
Instruction word (Type E):							
15					10	9 8	7 0
1	0	1	1	1	0	rX	uimm

6.2.3.1.5 Comparison Instructions

cmpeq	Compares if register value is equal to value of data word in memory. If this is the case, the condition flag RfnStatusReg.Cond is set. cmpeq rX, memoffset					cond = (rX == [FP + memoffset])	
Instruction word (Type B):							
15					10	9 8	7 0
0	1	0	0	0	1	rX	Memoffset

cmplt	Compares if register value is lower than value of data word in memory. If this is the case, the condition flag RfnStatusReg.Cond is set. cmplt rX, memoffset cond = (rX < [FP + memoffset])						
Instruction word (Type B):							
15					10	9 8	7 0
0	1	0	0	1	0	rX	Memoffset

cmpgt	Compares if register value is greater than value of data word in memory. If this is the case, the condition flag RfnStatusReg.Cond is set. cmpgt rX, memoffset cond = (rX > [FP + memoffset])						
Instruction word (Type B):							
15					10	9 8	7 0
0	1	0	0	1	1	rX	memoffset

6.2.3.1.6 Register Initialisation Instructions

ld_si	Load 8-bit signed immediate value into specified register (sign extension is performed). ld_si rX, simm rX = (int) simm						
Instruction word (Type C):							
15					10	9 8	7 0
1	0	0	1	1	1	rX	simm

ld_i	Load 8-bit unsigned immediate value into specified register. ld_i rX, uimm rX = uimm						
Instruction word (Type E):							
15					10	9 8	7 0
1	0	1	1	1	1	rX	uimm

6.2.3.1.7 Load/Store Instructions

ld	Load *ABW*-bit data word from memory. ld rX, memoffset rX = [FP + memoffset]						
Instruction word (Type B):							
15					10	9 8	7 0
0	0	1	0	0	0	rX	memoffset

ldmi	Load *ABW*-bit data word from memory. The address of the data word is specified by a pointer stored in memory (memory indirect addressing mode). ldmi rX, memoffset rX = [[FP + memoffset]]						
Instruction word (Type B):							
15					10	9 8	7 0
0	0	1	0	1	0	rX	memoffset

st	Store register value to *ABW*-bit data word in memory. st rX, memoffset [FP + memoffset] = rX						
Instruction word (Type B):							
15					10	9 8	7 0
0	0	1	0	0	1	rX	memoffset

stmi	Store register value to *ABW*-bit data word in memory, The address of the data word is specified by a pointer stored in memory (memory indirect addressing mode). stmi rX, memoffset [[FP + memoffset]] = rX						
Instruction word (Type B):							
15					10	9 8	7 0
0	0	1	0	1	1	rX	memoffset

lds	Load special purpose register with value from data word in memory. The following registers are supported: • rS=0: RfnControl1Reg • rS=1: RfnErrorReg • rS=2: RfnFramePointerReg lds memoffset rS = [FP + memoffset]						
Instruction word (Type B):							
15					10	9 8	7 0
0	1	1	0	1	1	rS	memoffset

sts	Store special purpose register value to *ABW*-bit data word in memory. The following registers are supported: • rS=0: RfnControl1Reg • rS=1: RfnErrorReg • rS=2: RfnFramePointerReg sts rS, memoffset [FP + memoffset] = rS						
Instruction word (Type B):							
15					10	9 8	7 0
0	1	1	1	0	0	rS	memoffset

6.2.3.1.8 I/O

brd	Loads *BusDW*-bit word from the configured bus master interface (AMBA AHB). The accessed bus address is defined by the referenced word of the Instruction & Data Memory. If the bit width of the fetched bus word is smaller than *ABW*, the MSBs will be padded by zeros, before the word is loaded into register *rX*. In case the accessed bus slave responds with a bus error, the flag RfnErrorReg.BE is raised. brd rX, memoffset rX(*BusDW*-1..0) = bus([FP + memoffset])						
Instruction word (Type B):							
15					10	9 8	7 0
0	1	0	1	0	0	rX	memoffset

bwr	Writes *BusDW*-bit word to the configured bus master interface (AMBA AHB). The accessed bus address is defined by the referenced word of the Instruction & Data Memory. If the bit width of the bus is smaller than *ABW*, the MSBs of register rX will be truncated. In case the accessed bus slave responds with a bus error, the flag RfnErrorReg.BE is raised. bwr rX, memoffset bus([FP + memoffset]) = rX(*BusDW*-1..0)						
Instruction word (Type B):							
15					10	9 8	7 0
0	1	0	1	0	1	rX	memoffset

6.2.3.1.9 Control Instructions

jmp	Jump instruction, which sets the Instruction Counter to a value loaded from a specified memory address. If the resulting target instruction address is out-of-range of the instruction memory area, the execution will terminate with an ILOP error. Note: This instruction is only useful when executed within a subroutine. If it is executed as a single instruction written to RfnInstructionReg, the instruction has the same effect as a single-cycle nop.							
	jmp memoffset							InstructionCounter = [FP + memoffset]
Instruction word (Type B):								
15					10	9 8		7 0
0	1	0	1	1	0	-		memoffset

jmpi	Immediate jump instruction (relative to Instruction Counter value when instruction is fetched). If the resulting target instruction address is out-of-range of the instruction memory area, the execution will terminate with an ILOP error. Note: This instruction is only useful when executed within a subroutine. If it is executed as a single instruction written to RfnInstructionReg, the instruction has the same effect as a single-cycle nop.						
	jmpi simm						InstructionCounter = InstructionCounter + simm + 1
Instruction word (Type F):							
15					10	9 0	
1	1	0	0	0	0	simm	

jmpi_ct	Conditional immediate jump instruction (relative to Instruction Counter value when instruction is fetched). If the resulting target instruction address is out-of-range of the instruction memory area, the execution will terminate with an ILOP error. Note: This instruction is only useful when executed within a subroutine. If it is executed as a single instruction written to RfnInstructionReg, the instruction has the same effect as a single-cycle nop.						
	jmpi_ct simm						if cond : InstructionCounter = InstructionCounter + simm + 1
Instruction word (Type F):							
15					10	9 0	
1	1	0	0	0	1	simm	

jmpi_cf	Conditional immediate jump instruction (relative to Instruction Counter value when instruction is fetched). If the resulting target instruction address is out-of-range of the instruction memory area, the execution will terminate with an ILOP error. Note: This instruction is only useful when executed within a subroutine. If it is executed as a single instruction written to RfnInstructionReg, the instruction has the same effect as a single-cycle nop. jmpi_cf simm — if ~cond : InstructionCounter = InstructionCounter + simm + 1						
Instruction word (Type F):							
15					10	9	0
1	1	0	0	1	0	simm	

wait	No operation for specified number of clock cycles. wait uimm — nop for uimm clock cycles						
Instruction word (Type F):							
15					10	9	0
1	1	0	0	1	1	uimm	

halt	Stops the execution of a subroutine. If no subroutine is executed, the instruction has the same effect as a single-cycle nop. Halt — Stops the execution of a subroutine						
Instruction word (Type A):							
15					10	9	0
0	0	0	0	0	0	-	

6.2.4 Error Handling

When overflow errors occur during the execution of arithmetic instructions the flag RfnErrorReg.OV is set.

When an illegal instruction is commanded, the flag RfnErrorReg.ILOP is set. An illegal instruction is an instruction with illegal opcode, which is either written to register RfnInstructionReg or fetched as part of a subroutine. Furthermore, the ILOP flag will be asserted when the target address of jump instruction is out-of-range of the Instruction Area of the memory. Whenever the ILOP flag is set, the instruction/subroutine execution is aborted.

Flag RfnErrorReg.UME is set when a read access to the Instruction & Data Memory fails due to an uncorrectable EDAC error (double-bit error). In this case the execution of the current instruction/subroutine is aborted. Single-bit errors are corrected automatically. For every corrected single-bit error field RfnEDACReport0Reg.CEC is incremented by one (see §9.2.1.8). The memory address of the first single-bit or double-bit error is reported by the field RfnEDACReport1Reg.FF (for details refer to §9.2.1.9).

When the ongoing execution of a single instruction or a subroutine is aborted due to another write access to register RfnInstructionReg or RfnStartsubReg, the execution abort error flag RfnErrorReg.EA is asserted.

A division by zero will not be executed, i.e. the result register will not be modified. Instead flag RfnErrorReg.DZ will be asserted.

Furthermore, the status register contains a general error flag: RfnStatusReg.Err. This flag is set whenever any of the errors described above occurs. The general error flag can, e.g. be used as an error status for a subroutine execution. If the flag is cleared at the beginning of a subroutine, the user can check the flag after the subroutine execution. If the flag was set again, the subroutine was not able to compute a valid result.

All of the above flags can be independently cleared by writing zero on the respective register bit. Note that the general error flag always needs to be cleared explicitly; it will not automatically become de-asserted when clearing any other error flag.

6.2.5 Usage Constraints

Writing the four general purpose registers or the frame pointer during an ongoing instruction or subroutine execution does not raise an error. However, this might corrupt the result of the computation and should therefore be avoided. The same consideration applies for accesses of the Instruction & Data Memory via the bus slave interface during instruction or subroutine execution.

7. BUS INTERFACE

The REDFINv2 Core provides an AMBA AHB slave interface to integrate the IP-core into a larger design. The width of the data buses, in this document denoted by *BusDW*, can be configured using a VHDL generic to a value of 16 or 32 bits. The AHB interface of the REDFINv2 Core only supports accesses of the same size as the configured bus width, i.e. in case of *BusDW*=16 the size of the AHB transfer needs to be set to “Halfword”, in case of *BusDW*=32 the transfer size needs to be “Word”. Bus transfers with other transfer sizes will be terminated with an error response. Furthermore, accesses to unmapped addresses will result in an AHB error response.

7.1 BUS ADDRESS MAP

The bus base address *BBA* of the REFFINv2 IP-Core is configurable by a VHDL generic. The address range of the bus is partitioned into one area for the registers of the REDFINv2 Core and one area for the Instruction & Data Memory, as can be seen in Table 4. The organisation of the respective areas, can be found in §8 and §9.

Area	Start address	End address	Size	Comment
Registers	*BBA*	*BBA* + $3FF_{16}$	1024 bytes	Refer to §9.
Instruction & Data Memory	*BBA* + 400_{16}	*BBA* + 400_{16} + “Size of Instruction & Data Memory” - 1	Refer to §8.	Refer to §8.

Table 4 – Bus Address Map

Note: All specified addresses are byte addresses.

8. INSTRUCTION & DATA MEMORY

8.1 MEMORY MAP

The REDFINv2 Core can be configured to work with an Instruction & Data Memory that is capable of storing memory words, which are either 16 or 32 bits wide. The bit width of the stored memory words will be denoted by *MemDW*.

Note that the memory width needs to be equal to the configured data width of the bus interface, hence *MemDW*=*BusDW*.

The instruction and data words stored in the memory are protected by a Hamming code, which is able to detect double-bit errors and correct single-bit errors (SECDED). Since the parity bits will be appended as MSBs to the memory words, the physical bit width of the connected RAM block needs to be greater than the selected *MemDW* value. In case of *MemDW*=16 there are 6 parity bits needed, in case of *MemDW*=32 the number of parity bits is 7. Thus, the total width of the connected RAM block needs to be either 22 (*MemDW*=16) or 39 bits (*MemDW*=32).

The memory organisation depends on the following configuration parameters:

- The width of the Instruction & Data Memory (*MemDW*)
- The number of available instruction words (*InstrWC*).
- The number of available data words (*DataWC*)
- Data width of the REDFINv2 Core (*ABW*).

The *MemDW* setting determines how the 16-bit instruction words can be organised in the memory. In case of *MemDW*=16, one instruction word can be stored per memory location, in case of *MemDW*=32 two instruction words are stored per memory location (refer to §8.2.1 and §8.3.1, respectively).

The organisation of the data area depends on how many memory locations are needed to store the *ABW*-bit data words, which again depends on the memory width. For details refer to the data word entry definitions for *MemDW*=16 and *MemDW*=32 in §8.2 and §8.3.

Table 6 to Table 10 show the memory maps for all possible *MemDW* and *ABW* configurations.

Area	Entries	Start address	End address	Size	Comment
Instruction Area	Instruction Word 0	0	0	*InstrWC**2 bytes	Refer to §8.2.1.
	Instruction Word 1	1	1		
	…	…	…		
	Instruction Word *InstrWC*-1	*InstrWC*-1	*InstrWC*-1		
Data Area	Data Word 0	*InstrWC*	*InstrWC*	*DataWC**2 bytes	Refer to §8.2.2.
	Data Word 1	*InstrWC*+1	*InstrWC*+1		
	…	…	…		
	Data Word *DataWC*-1	*InstrWC*+ *DataWC-1*	*InstrWC*+ *DataWC-1*		

Table 5 – Instruction & Data Memory (*MemDW*=16, 8 ≤ *ABW* ≤ 16)[3]

Area	Entries	Start address	End address	Size	Comment
Instruction Area	Instruction Word 0	0	0	*InstrWC**2 bytes	Refer to §8.2.1.
	Instruction Word 1	1	1		
	…	…	…		
	Instruction Word *InstrWC*-1	*InstrWC*-1	*InstrWC*-1		
Data Area	Data Word 0	*InstrWC*	*InstrWC+1*	*DataWC**4 bytes	Refer to §8.2.2.
	Data Word 1	*InstrWC+2*	*InstrWC+3*		
	…	…	…		
	Data Word *DataWC*-1	*InstrWC*+ *(DataWC-1)*2*	*InstrWC*+ *(DataWC-1)*2+1*		

Table 6 – Instruction & Data Memory (*MemDW*=16, 17 ≤ *ABW* ≤ 32)[3]

[3] All specified addresses are word addresses for *MemDW*-bit memory words.

Area	Entries	Start address	End address	Size	Comment
Instruction Area	Instruction Word 0	0	0	*InstrWC**2 bytes	Refer to §8.2.1.
	Instruction Word 1	1	1		
	…	…	…		
	Instruction Word *InstrWC*-1	*InstrWC*-1	*InstrWC*-1		
Data Area	Data Word 0	*InstrWC*	*InstrWC+2*	*DataWC**6 bytes	Refer to §8.2.4.
	Data Word 1	*InstrWC+3*	*InstrWC+5*		
	…	…	…		
	Data Word *DataWC*-1	*InstrWC+ (DataWC-1)*3*	*InstrWC+ (DataWC-1)*3+2*		

Table 7 – Instruction & Data Memory (*MemDW*=16, 33 ≤ *ABW* ≤ 48)[3]

Area	Entries	Start address	End address	Size	Comment
Instruction Area	Instruction Word 0	0	0	*InstrWC**2 bytes	Refer to §8.2.1.
	Instruction Word 1	1	1		
	…	…	…		
	Instruction Word *InstrWC*-1	*InstrWC*-1	*InstrWC*-1		
Data Area	Data Word 0	*InstrWC*	*InstrWC+3*	*DataWC**8 bytes	Refer to §8.2.5.
	Data Word 1	*InstrWC+4*	*InstrWC+7*		
	…	…	…		
	Data Word *DataWC*-1	*InstrWC+ (DataWC-1)*4*	*InstrWC+ (DataWC-1)*4+3*		

Table 8 – Instruction & Data Memory (*MemDW*=16, 49 ≤ *ABW* ≤ 64)[3]

Area	**Entries**	**Start address**	**End address**	**Size**	**Comment**
Instruction Area	Instruction Word 0/1	0	0	*InstrWC*2* bytes	Refer to §8.3.1.
	Instruction Word 2/3	1	1		
	…	…	…		
	Instruction Word *InstrWC*-2 / *InstrWC*-1	*InstrWC/2*-1	*InstrWC/2*-1		
Data Area	Data Word 0	*InstrWC/2*	*InstrWC/2*	*DataWC**4 bytes	Refer to §8.3.2.
	Data Word 1	*InstrWC/2+1*	*InstrWC/2+1*		
	…	…	…		
	Data Word *DataWC*-1	*InstrWC/2+ DataWC-1*	*InstrWC/2+ DataWC-1*		

Table 9 – Instruction & Data Memory (*MemDW*=32, 8 ≤ *ABW* ≤ 32)[3]

Area	**Entries**	**Start address**	**End address**	**Size**	**Comment**
Instruction Area	Instruction Word 0/1	0	0	*InstrWC*2* bytes	Refer to §8.3.1.
	Instruction Word 2/3	1	1		
	…	…	…		
	Instruction Word *InstrWC*-2 / *InstrWC*-1	*InstrWC/2*-1	*InstrWC/2*-1		
Data Area	Data Word 0	*InstrWC/2*	*InstrWC/2+1*	*DataWC**8 bytes	Refer to §8.3.3.
	Data Word 1	*InstrWC/2+2*	*InstrWC/2+3*		
	…	…	…		
	Data Word *DataWC*-1	*InstrWC/2+ (DataWC-1)*2*	*InstrWC/2+ (DataWC-1)*2+1*		

Table 10 – Instruction & Data Memory (*MemDW*=32, 33 ≤ *ABW* ≤ 64)[3]

8.2 MEMORY ENTRIES DEFINITION (*MEMDW*=16)

Note: The SRAM entries are defined as either R or W as seen from the user, depending on the typical usage of the entry. All entries are however possible to read and write via memory access.

8.2.1 Instruction Word Entries

Word	15	0
0	Instruction Word$(15:0)	
	MSB	LSB

Function Instruction word that is part of a subroutine (a detailed description of the instruction word format can be found in §6.2.3.1).

Fields

Bit	Field	Access	Value	Description
15:0	Instruction	R/W	-	Instruction word.

Timing Read access by REDFINv2 IP-Core is performed during subroutine execution. Read/write access can also be performed by an external controller via the configured bus interface. During an ongoing subroutine execution, it needs to be ensured by the external controller that no instruction words are updated, which might be concurrently accessed by the REDFINv2 core.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.2.2 Data Word Entries (8 ≤ *ABW* ≤ 16)

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Data word$(*ABW*-1:0)															
	MSB															LSB

Function *ABW*-bit data words, which are read by the REDFINv2 IP-Core as operand for an operation, or are written by the Sequencer when a store instruction is executed.

Fields

Bit	Field	Access	Value	Description
ABW-1:0	Data word	R/W	Signed	Data word entry. If *ABW* < 16, the MSBs of Word 0 remain unused (in case of a store instruction they will be set to 0).

Timing Read/write access by the REDFINv2 IP-Core is performed during instruction and subroutine execution. Read/write access of data words can also be performed by an external controller via the configured bus interface. It needs to be ensured by the external controller that data word accesses are performed mutually exclusive.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.2.3 Data Word Entries (17 ≤ *ABW* ≤ 32)

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Data word$(15:0)															
1	Data word$(*ABW*-1:16)															
	MSB															LSB

Function *ABW*-bit data words, which are read by the REDFINv2 IP-Core as operand for an operation, or are written by the Sequencer when a store instruction is executed.

Fields

Bit	Field	Access	Value	Description
ABW-1:0	Data word	R/W	Signed	Data word entry. If *ABW* < 32, the MSBs of Word 1 remain unused (in case of a store instruction they will be set to 0).

Timing Read/write access by the REDFINv2 IP-Core is performed during instruction and subroutine execution. Read/write access of data words can also be performed by an external controller via the configured bus interface. It needs to be ensured by the external controller that data word accesses are performed mutually exclusive.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.2.4 Data Word Entries (33 ≤ *ABW* ≤ 48)

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Data word(15:0)															
1	Data word(31:16)															
2	Data word(*ABW*-1:32)															
	MSB															LSB

Function *ABW*-bit data words, which are read by the REDFINv2 IP-Core as operand for an operation, or are written by the Sequencer when a store instruction is executed.

Fields

Bit	Field	Access	Value	Description
ABW-1:0	Data word	R/W	Signed	Data word entry. If *ABW* < 48, the MSBs of Word 2 remain unused (in case of a store instruction they will be set to 0).

Timing Read/write access by the REDFINv2 IP-Core is performed during instruction and subroutine execution. Read/write access of data words can also be performed by an external controller via the configured bus interface. It needs to be ensured by the external controller that data word accesses are performed mutually exclusive.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.2.5 Data Word Entries (49 ≤ *ABW* ≤ 64)

Word	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Data word$(15:0)															
1	Data word$(31:16)															
2	Data word$(47:32)															
3	Data word$(*ABW*-1:48)															
	MSB															LSB

Function *ABW*-bit data words, which are read by the REDFINv2 IP-Core as operand for an operation, or are written by the Sequencer when a store instruction is executed.

Fields

Bit	Field	Access	Value	Description
ABW-1:0	Data word	R/W	Signed	Data word entry. If *ABW* < 64, the MSBs of Word 3 remain unused (in case of a store instruction they will be set to 0).

Timing Read/write access by the REDFINv2 IP-Core is performed during instruction and subroutine execution. Read/write access of data words can also be performed by an external controller via the configured bus interface. It needs to be ensured by the external controller that data word accesses are performed mutually exclusive.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.3 MEMORY ENTRIES DEFINITION (*MEMDW*=32)

8.3.1 Instruction Word Entries

Word	31 16	15 0
0	Instruction Word 1 (15:0)	Instruction Word 0 (15:0)
	MSB	LSB

Function Each instruction word entry encodes 2 16-bit instruction words, which are part of a subroutine (a detailed description of the instruction word format can be found in §6.2.3.1).

Fields

Bit	Field	Access	Value	Description
15:0	Instruction word 0	R/W	-	First instruction word to be executed if this memory location is fetched.
31:16	Instruction word 1	R/W	-	Second instruction word to be potentially executed if first instruction word is not a halt or a jump instruction, which jumps to another instruction.

Timing Read access by REDFINv2 IP-Core is performed during subroutine execution. Read/write access can also be performed by an external controller via the configured bus interface. During an ongoing subroutine execution, it needs to be ensured by the external controller that no instruction words are updated, which might be concurrently accessed by the REDFINv2 core.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.3.2 Data Word Entries (8 ≤ *ABW* ≤ 32)

Word	31	0
0	Data word$(*ABW*-1:0)	
	MSB	LSB

Function *ABW*-bit data words, which are read by the REDFINv2 IP-Core as operand for an operation, or are written by the Sequencer when a store instruction is executed.

Fields

Bit	Field	Access	Value	Description
ABW-1:0	Data word	R/W	Signed	Data word entry. If *ABW* < 32, the MSBs of Word 0 remain unused (in case of a store instruction they will be set to 0).

Timing Read/write access by the REDFINv2 IP-Core is performed during instruction and subroutine execution. Read/write access of data words can also be performed by an external controller via the configured bus interface. It needs to be ensured by the external controller that data word accesses are performed mutually exclusive.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

8.3.3 Data Word Entries (33 ≤ *ABW* ≤ 64)

Word	31 0
0	Data word$(31:0)
1	Data word$(*ABW*-1:32)
	MSB LSB

Function *ABW*-bit data words, which are read by the REDFINv2 IP-Core as operand for an operation, or are written by the Sequencer when a store instruction is executed.

Fields

Bit	Field	Access	Value	Description
ABW-1:0	Data word	R/W	Signed	Data word entry. If *ABW* < 64, the MSBs of Word 1 remain unused (in case of a store instruction they will be set to 0).

Timing Read/write access by the REDFINv2 IP-Core is performed during instruction and subroutine execution. Read/write access of data words can also be performed by an external controller via the configured bus interface. It needs to be ensured by the external controller that data word accesses are performed mutually exclusive.

Constraints N/A

Reset Value Not initialised by REDFINv2 core.

9. REGISTER

9.1 REGISTER MAP

The registers of the REDFINv2 core can be accessed over its AMBA AHB bus slave interface. Table 11 specifies the mapping of registers to bus addresses. Note that this table provides register offsets, relative to the base address of the bus slave interface (refer to §7.1). To obtain the absolute bus address of a particular register the respective register offset needs to be added to this base address. Furthermore, note that the register addresses are not dependent on the configured data width of the bus. For both *BusDW*=16 and *BusDW*=32 the byte offsets as specified in Table 11 are applicable.

Register Name	**Byte Offset**
RfnControl0Reg	0x0
RfnControl1Reg	0x4
RfnStatusReg	0x8
RfnErrorReg	0xC
RfnStartsubReg	0x10
RfnInstructionReg	0x14
RfnFramePointerReg	0x18
RfnEDACReport0Reg	0x1C
RfnEDACReport1Reg	0x20
RfnEDACInjectReg	0x24

Table 11 – Register Offsets

9.2 REGISTER DESCRIPTION

This chapter defines all REDFINv2 registers and their fields. For each register the following is defined.

- Function
- Field Description
- Timing
- Constraints
- Reset Value

Note: The access type for REDFINv2 registers are shown at each detailed register definition. The following convention is used:

- W register field write access is supported
- R register field read access is supported
- RC register field read access is a timing event (e.g. starts a function or clears a flag)
- WT register field write access is a timing event (e.g. starts a function)

Note: Unused bits are not explicitly described and are read as ‘0’ and a write access to unused bits is ignored.

9.2.1.1 RfnControl0Reg

Function Configures the REDFINv2 Core.

Fields

Bit	Field	Access	Value	Description
0	SCRB	R/W	0	Scrubbing of Instruction & Data Memory inactive.
			1	Scrubbing of Instruction & Data Memory active.
15:1	SubIter	R/W	Unsigned	Number of iterations+1 a subroutine execution is repeated.

Timing None

Constraints None

Reset Value 0x0000

9.2.1.2 RfnControl1Reg

Function Configures the REDFINv2 Core.

Fields

Bit	Field	Access	Value	Description
5:0	FracLen	R/W	Unsigned	If this field is set to 0 all arithmetic operations are performed as regular integer operations. In case field is set to a value > 0, operands from memory or registers will be interpreted as signed fixed-point numbers with *FracLen* fractional bits.

Timing None

Constraints FracLen must be smaller than *ABW*.

Reset Value 0x0000

9.2.1.3 RfnStatusReg

Function Status register of the REDFINv2 Core.

Fields

Bit	Field	Access	Value	Description
0	InitC	R	0	Initialisation of Instruction & Data Memory with 0 values after reset release ongoing.
			1	Initialisation of Instruction & Data Memory completed.
1	Active	R	0	REDFINv2 Core is idle.
			1	Execution of an instruction or subroutine is ongoing.

Timing None

Constraints None

Reset Value 0x0000

9.2.1.4 RfnErrorReg

Function Error register of the REDFINv2 Core.

Fields

Bit	Field	Access	Value	Description
0	OV	R/W	0	No arithmetic overflow occurred.
			1	Arithmetic overflow occurred. An overflow is set by all arithmetic instructions, if the result of the operation exceeds the signed integer range that can be represented under the current configuration. The specific range depends on the setting of register field RfnControl1Reg.FracLen. Flag can be cleared by writing zero on this register bit, writing of ‘1’ is ignored.
1	DZ	R/W	0	No Division-by-zero error occurred.
			1	Division-by-zero error occurred. Flag can be cleared by writing zero on this register bit, writing of ‘1’ is ignored.
2	ILOP	R/W	0	No illegal operation occurred.
			1	Illegal operation occurred, i.e. an instruction with illegal opcode or a *jump* instruction with illegal target address. Flag can be cleared by writing zero on this register bit, writing of ‘1’ is ignored.
3	EA	R/W	0	No execution abort error occurred.
			1	Execution abort error occurred, when RfnInstructionReg or RfnStartsubReg is written during an ongoing instruction or subroutine execution.
4	UEOS	R/W	0	No unexpected end of subroutine occurred.
			1	Unexpected end of subroutine occurred; i.e. last memory address of Instruction Area was reached without proper subroutine termination by Halt instruction. Flag can be cleared by writing zero on this register bit, writing of ‘1’ is ignored.
5	BE	R/W	0	No bus error has occurred.
			1	Bus error has occurred during read access.
6	UME	R/W	0	No uncorrectable EDAC error has occurred since last read access.
			1	At least one uncorrectable EDAC error has occurred since last read access.

Timing None

Constraints None

Reset Value 0x0000

9.2.1.5 RfnStartsubReg

Function Upon a write access to this register the execution of a subroutine, located at the specified address in the instruction area of the Instruction & Data Memory, is started.

Fields

Bit	Field	Access	Value	Description
n-1:0	Addr	R/WT	Unsigned	Subroutine address (specified as instruction index, where address 0 references the first instruction word in the first instruction entry of the Instruction & Data Memory). The bit width *n* of this field is determined as follows: • *MemDW* = 16: *n* = ceil($\log_2$(*InstrWC*)) • *MemDW* = 32: *n* = ceil($\log_2$(*InstrWC / 2*))

Timing None

Constraints None

Reset Value 0x0000

9.2.1.6 RfnInstructionReg

Function Upon a write access to this register the written instruction is executed.

Fields

Bit	Field	Access	Value	Description
15:0	Instr	R/WT	-	Instruction word (a detailed description of the instruction word format can be found in §6.2.3.1).

Timing None

Constraints None

Reset Value 0x0000

9.2.1.7 RfnFramePointerReg

Function Frame pointer registers that provides base address for addressing memory operands.

Fields

Bit	Field	Access	Value	Description
n-1:0	Addr	R/W	Unsigned	Address of frame pointer (*ABW*-bit word address, starting from the first data word in the Data area of the Instruction & Data Memory. A frame pointer value of 0 points to the first *ABW*-bit data word, a frame pointer value of 1 points to the second *ABW*-bit data word). The bit width *n* of the frame pointer field is defined as follows: $n = \mathrm{ceil}(\log_2(\mathit{DataWC}))$

Timing None

Constraints Writing this registers during an ongoing instruction or subroutine execution might corrupt the result of the computation and should be avoided.

Reset Value 0x0000

9.2.1.8 RfnEDACReport0Reg

Function Reports errors related to the EDAC function of the Instruction & Data Memory. The register is cleared by a read access.

Fields

Bit	Field	Access	Value	Description
3:0	CEC(3..0)	R/W	Unsigned	Correctable EDAC error counter. Saturates at 15.

Timing None

Constraints None

Reset Value 0x0000

9.2.1.9 RfnEDACReport1Reg

Function Reports errors related to the EDAC function of the Instruction & Data Memory. The register is cleared by a read access.

Fields

Bit	Field	Access	Value	Description
MemAW-1:0	FF	R	Unsigned	The RAM address of the first detected EDAC error since last read access. The FF field is latched to the RAM address when one of the following events occurs: - CEC goes from 0 to 1 and UEC is 0. - UME goes from 0 to 1. Uncorrectable errors have thus precedence over correctable errors. The FF field shall be ignored in case both CEC and UME are 0.

Timing None

Constraints None

Reset Value 0x0000

9.2.1.10 RfnEDACInjectReg

Function Configures EDAC injection/test function of the Instruction & Data Memory.

Fields

Bit	Field	Access	Value	Description
0	SBTW	R/W	0	No single-bit error forced during write access.
			1	Forcing once a single-bit error in data word when written to address equal to UEAddr (for test purpose).
1	DBTW	R/W	0	No double-bit error forced during write access.
			1	Forcing once a double-bit error in data word when written to address equal to UEAddr (for test purpose).
MemAW+1:2	UEAddr	R/W	Unsigned	Address pointer to specify which data word to be manipulated (word address, starting from the beginning of the Instruction & Data Memory.

Timing -

Constraints -

Reset Value 0x0000

10. INTERFACE

10.1 INTERFACE DESCRIPTION

10.1.1 Clock Interface

The REDFINv2 IP-Core needs to be supplied with two clock signals, one for the Core Clock domain and one for the Bus Clock domain, named *Clk* and *AHBClk*. These two clock signals need to be ratiochronous, i.e. derived from the same clock source and need to have the following frequency relationship:

$$f_{Clk} = n \cdot f_{AHBClk}, \tag{1}$$

where the ratio *n* = iClk2AHBClkRatio_G needs to be an integer number greater than or equal to 1.

Due to the ratiochronous nature of *Clk* and *AHBClk*, the signals crossing between these two domains are not asynchronous and therefore are not synchronized by a multi-stage brute-force synchronizer. However, it is necessary to specify the timing relationship between the two clocks in order to properly constrain the signals crossing the clock domains for the static timing analysis. The typical scenario is that the slower AHB clock is generated from a faster main clock by a clock divider or a PLL. In this case the relationship between the output signal of the clock divider and the main clock can be specified by adding a "create_generated_clock" statement in the SDC file.

If iClk2AHBClkRatio_G is set to 1, both clock inputs (AHBClk, Clk) need to be connected to the same clock signal and only a single clock domain needs to be constrained.

■ *Clk*

This signal clocks the Core Clock domain of the REDFINv2 IP-Core, which contains all functions relevant for instruction and subroutine execution. As described above, the Core Clock domain can be operated with the same frequency as or a higher frequency than the Bus Clock domain, with the only constraint that the frequency ratio is an integer number.

■ *AHBClk*

This signal clocks the Bus Clock domain of the REDFINv2 IP-Core.

10.1.2 Reset Interface

■ *SyncReset_N*

The REDFINv2 IP-Core is reset when *SyncReset_N* is active (minimum reset pulse width is 100 ns). The reset function is performed asynchronously and therefore does not need an active clock. Reset de-assertion, however, should be synchronous to a rising edge of the slower of the two clocks, i.e. the *AHBClk*. This guarantees that both clock domains start with their first clock cycle at the same point in time (there might, however, be a constant phase offset between the clock signals). Since the REDFINv2 IP-Core does not perform synchronisation of the reset signal, this needs to be guaranteed by the higher-level design.

Table 12 summarizes all REDFINv2 core output ports and their reset values.

10.2 INTERFACE SIGNAL RESET VALUES

Signal	Value During/After Reset
IDMemRd	0
IAHB_DMemWr	0
IDMemAddr(MemAW-1..0)	all 0
IDMemWrD(MemDW-1..0)	all 0
AHB_SLAVE.HREADY	0
AHB_SLAVE.HRESP(1..0)	all 0
AHB_SLAVE.HRDATA(31..0)	all 0
AHB_MASTER.HBUSREQx	0
AHB_MASTER.HLOCKx	0
AHB_MASTER.HTRANS(1..0)	all 0
AHB_MASTER.HADDR(31..0)	all 0
AHB_MASTER.HWRITE	0
AHB_MASTER.HSIZE(2..0)	all 0
AHB_MASTER.HBURST(2..0)	all 0
AHB_MASTER.HPROT(3..0)	all 0
AHB_MASTER.HWDATA(31..0)	all 0

Table 12 –REDFINv2 IP-Core output signal reset values

10.3 INTERFACE SIGNAL TIMINGS

10.3.1 Instruction & Data Memory Interface

The expected read/write timings of the Instruction & Data Memory, which is connected to the REDFINv2 IP-Core is illustrated in Figure 4 and Figure 5. These waveforms are consistent with synchronous non-registered operation of block RAMs, as usually available in FPGAs.

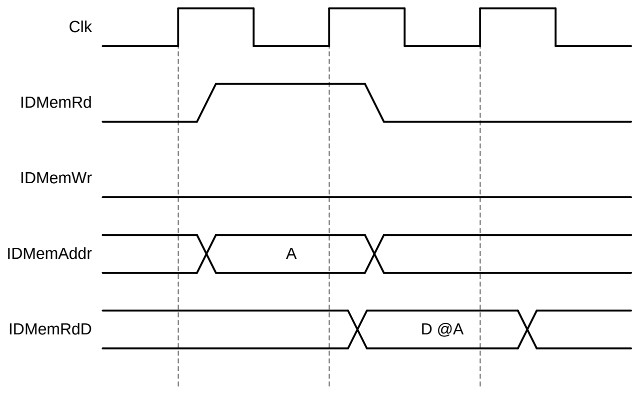

Figure 4 – Read Waveform

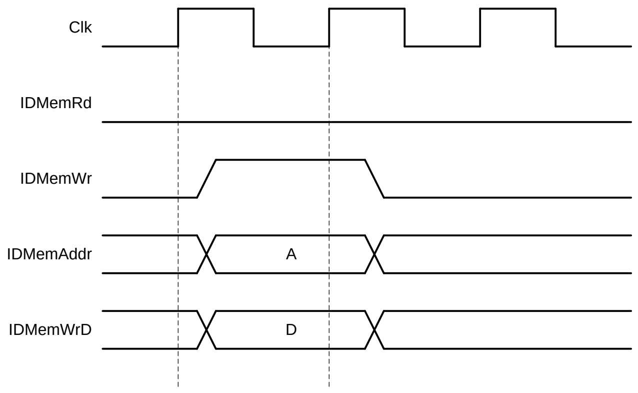

Figure 5 – Write Waveform

11. DEBUG AND TEST FACILITIES

N/A

12. APPENDIX

N/A