STORM CORE Processor System
by Stephan Nolting

General Data Sheet

Proprietary Notice

ARM is a trademark of Advanced RISC Machines Ltd.
Xilinx ISE and Xilinx ISIM are trademarks of Xilinx, Inc.
Quartus is a trademark of Altera corporation.
ModelSim is a trademark of Metor Graphics, Inc.

The **STORM CORE** processor system was created by Stephan Nolting.
Contact: stnolting@googlemail.com, zero_gravity@opencores.org

The most recent version of the STORM core system and it's documentary can be found at
http://www.opencores.com/project,storm_core

Table of content

1. Introduction
- 1.1 VHDL file hierarchy
- 1.2 Module description

2. Interface Signals
- 2.1 CORE.vhd Interface signals
- 2.2 STORM_TOP.vhd Interface signals

3. Core Programmer Model
- 3.1 Operating modes
- 3.2 Registers
- 3.3 Exceptions
- 3.4 Differences between ARM7 and STORM core
  - 3.4.1 Critical differences
  - 3.4.2 Noncritical differences
  - 3.4.3 Differences in STORM mode

4. Core Hardware
- 4.1 Data flow
- 4.2 Pipeline conflicts
  - 4.2.1 Local pipeline conflicts
  - 4.2.2 Temporal pipeline conflicts
  - 4.2.3 Branches
- 4.3 Stage control bus

5. Storm Core System
- 5.1 System Bridge
- 5.2 Internal Memory

6. Getting started
- 6.1 Hardware setup & simulation
- 6.2 Software setup

1. Introduction

The STORM core system is a powerfull open source 32-bit RISC processor system, which is completely described in behavioral VHDL. No dedicated hardware components are instantiated, so it can be synthesized for any FPGA device. The function set, the operation codes as well as the programmer's model are partly native to ARM's famous ARM processor cores (ARM7, ARM9). For further information about compatibility, see the chapter “Programmer's model / Differences between ARM7 and STORM core”.

You can use the processor core on it's own for your project (top entity ***core.vhd***), or you can use a complete system environment, including an internal working memory (RAM) and a Wishbone bus master interface. The top entity for this setup is ***storm_top.vhd***. All other needed files are listed below in the file hierarchy.

1.1 Processor VHDL file hierarchy (Instance name ~ VHDL file)

```
STORM_TOP.vhd
-> PERIPHERAL_UNIT ~ ACCESS_ARBITER.vhd
-> WISHBONE_INTERFACE ~ WISHBONE_IO.vhd
-> WORKING_MEMORY ~ MEMORY.vhd
-> PROCESSOR_CORE ~ CORE.vhd
   -> STORM_CORE.vhd (package file)
   -> Instruction_Decoder ~ X1_OPCODE_DECODER.vhd
   -> Operation_Flow_Control ~ FLOW_CTRL.vhd
   -> Machine_Control_System ~ MCR_SYS.vhd
   -> Register_File ~ REG_FILE.vhd
   -> Operand_Fetch_Unit ~ OPERAND_UNIT.vhd
   -> Multishifter ~ MS_UNIT.vhd
      -> Multiplicator ~ MULTIPLY_UNIT.vhd
      -> Barrelshifter ~ BARREL_SHIFTER.vhd
   -> Operator ~ ALU.vhd
      -> Arithmetical_Core ~ ARITHMETICAL_UNIT.vhd
      -> Logical_Core ~ LOGICAL_UNIT.vhd
   -> Memory_Access ~ LOAD_STORE_UNIT.vhd
   -> Data_Write_Back ~ WB_UNIT.vhd
```

1.2 Module description

REG_FILE.vhd
This unit contains the data register file. It consists of 32 registers, whereof 16 (15 data registers and the PC) are accessible at one time, depending on the current operating mode. The registers are located in three memory blocks (e.g. distributed RAM) to enabled optimal resource use.

OPERAND_UNIT.vhd
The operand unit performs the operand fetch for all the 3 operand-slots. It loads register values from the register file and immediate values from the instruction decoder. Also the data conflict detector and the forwarding system are located here.

MS_UNIT.vhd
The multishifter performs either a multiplication or a barrel shift and outputs the data onto the ALU's secondary data path. Due to the three operand slots, a shift or a multiplication needs no additional data fetch cycles.

BARREL_SHIFTER.vhd
This unit performs the barrel-shifting of the data in ALU data path B.
The shift value can either be an immediate from the opcode or a register value.

MULLTIPLY_UNIT.vhd
The multiply unit calculates a 32x32 bit operation and outputs the lower 32 bit to the ALU data path B.

ALU.vhd
The ALU holds the primary data operation units. All address-operations are executed here (except for the program counter increment). Furthermore it handles the data access to all machine control registers.

ARITHMETICAL_UNIT.vhd
The arithmetical unit performs the dedicated arithmetical add & sub operations
and also the "arithmetical compares" (CMP, CMN).

LOGICAL_UNIT.vhd
The logical unit performs the dedicated logical operations like "BIC" and "OR" and also the "logical compares" (TST, TEQ).

LOAD_STORE_UNIT.vhd
The load-store unit outputs the address and the control signals to the data memory access port.

WB_UNIT.vhd
The write-back unit performs the data write back to the register file and also accepts the read data from the data memory interface.

MCR_SYS.vhd
The MCR system holds the machine control registers, which are the program counter, the saved machine status registers and the current machine status register as well as the interrupt handler and the context change system.

FLOW_CTRL.vhd
The flow control generates the control signals for each stage and every module of the pipeline.
The decoded instruction data is brought to this unit where it triggers all internal operations. Furthermore the cycle arbiter, which solves temporal pipeline conflicts, the branch arbiter and the condition check system are located here.

X1_OPCODE_DECODER.vhd
This unit decodes the ARM-native 32-bit opcodes into processor control signals.

SYSTEM_BRIDGE.vhd
The system bridge is a simple but efficient access arbiter, which can coordinate the access request of two clients (instruction and data interface) to two different resources (internal memory & Wishbone interface).

MEMORY.vhd
The beginning of the IO area (starting at address 0x00000000) can be mapped to the internal working memory. It can be loaded before the synthesis with a user-defined start-up code (e.g. bootloader).

WISHBONE_IO
When not selecting the internal memory, the access arbiter selects the Wishbone bus master interface for IO operations.

STORM_core.vhd
This file is the main package file, where all modules and parameter are defined.

2. Interface Signals

The type of the core inputs and outputs is *std_logic* or *std_logic_vector* (x downto y).
Tie all unused inputs to low potential and leave all unused outputs unconnected (open).

2.1 CORE.vhd interface signals

Signal name	Data size	Direction	Function
CLK	1 bit	Input	Main clock signal, flip-flops trigger on rising edge.
RES	1 bit	Input	Main clock signal, high active, synchronous to CLK.
HALT	1 bit	Input	Disables the CLK signal to freeze the processor when high (only set to high while CLK is low).
MODE	5 bit	Output	Current processor operating mode.
D_MEM_REQ	1 bit	Output	Data interface memory request (high-active).
D_MEM_ADR	32 bit	Output	Data address.
D_MEM_RD_DTA	32 bit	Input	Read data from memory.
D_MEM_WR_DTA	32 bit	Output	Write data to memory.
D_MEM_DQ	4 bit	Output	Data quantity for transfer. “00” = word, “01” = byte, “10”/”11” = half word.
D_MEM_RW	1 bit	Output	Read (low) / write (high) signal.
D_MEM_ABORT	1 bit	Input	Triggers the data abort trap when high.
I_MEM_REQ	1 bit	Output	Instruction interface memory request (high-active).
I_MEM_ADR	32 bit	Output	Instruction address.
I_MEM_RD_DTA	32 bit	Input	Instruction read back from memory.
I_MEM_DQ	4 bit	Output	Data quantity for transfer. “00” = word, “10”/”11” = half word.
I_MEM_ABORT	1 bit	Input	Triggers the prefetch abort trap when high.
IRQ	1 bit	Input	Triggers the IRQ trap when high.
FIQ	1 bit	Input	Triggers the FIQ trap when high.

2.2 STORM_TOP.vhd entity

Signal name	Data size	Direction	Function
CLK_I	1 bit	Input	Main clock signal, flip-flops trigger on rising edge.
RST_I	1 bit	Input	Main clock signal, high active, synchronous to CLK.
WB_DATA_I	32 bit	Input	Wishbone bus data input.
WB_DATA_O	32 bit	Output	Wishbone bus data output.
WB_ADR_O	32 bit	Output	Wishbone bus address output.
WB_ACK_I	1 bit	Input	Wishbone bus acknowledge signal.
WB_SEL_O	4 bit	Output	Wishbone bus byte select. Each bit selects a byte of the data vector (e.g. bit 0 for data bits 7 downto 0).
WB_WE_O	1 bit	Output	Wishbone bus write enable.
WB_STB_O	1 bit	Output	Wishbone bus valid cycle.
WB_CYC_O	1 bit	Output	Wishbone bus valid cycle.
MODE_O	5 bit	Output	Current processor operating mode.
D_ABT_I	1 bit	Input	Triggers the data abort trap when high.
I_ABT_I	1 bit	Input	Triggers the instruction (prefetch) abort when high.
IRQ_I	1 bit	Input	Triggers the IRQ trap when high.
FIQ_I	1 bit	Input	Triggers the FIQ trap when high.

3. Core Programmer model

The STORM core is an ARM native processor system, so you can use most of the ARM's tool chains.
Since the STORM core is *not* intended to be an ARM clone, the programmer's model, the hardware itself and the complete behavior differs in some aspects. Important differences between the original ARM and the STORM core are noted in this chapter.

3.1 Operating modes

Six operating modes are implemented in the STORM core. When resetting the processor, it startes operation always in supervisor mode. To change to a different mode, just write the corresponding *MODE* code to the lowest 5 bit of the CMSR (CPSR) or SMSR (SPSR), when the processor is in a higher privilege mode than “user mode”. The program counter is not affected when using this style of changing the mode. The current mode is also displayed on the MODE port of the STORM processor.
All mode codes are bit compatible to the ARM7.

MODE	Interrupt vector	MODE code
User	0x00000000	“10000”
Undefined Inst.	0X00000004	“11011”
Supervisor	0x00000008	“10011”
Prefetch Abort	0x0000000C	“10111”
Data Abort	0x00000010	“10111”
reserved	0x00000014	
IRQ	0x00000018	“10010”
FIQ	0x0000001C	“10001”

3.2 Registers

The STORM CORE provides six different operating modes, where every mode has it's own register set, including a link register (LR, always r14), the program counter (PC, always R15), a current machine status register (CMSR/CPSR) and a saved machine status register (SMSR_<mode> / SPSR_<mode>)

MODE	Accessible data registers	Machine registers
User	R0, …, R14	PC, CPSR
FIQ	R0, …, R07, R08_FIQ, …, R14_FIQ	PC, CPSR, SPSR_FIQ
IRQ	R0, …, R12, R13_IRQ, …, R14_IRQ	PC, CPSR, SPSR_IRQ
Supervisor	R0, …, R12, R13_SVP, …, R14_SVP	PC, CPSR, SPSR_SVP
Abort	R0, …, R12, R13_ABT, …, R14_ABT	PC, CPSR, SPSR_ABT
Undef. I.	R0, …, R12, R13_UND, …, R14_UND	PC, CPSR, SPSR_UND

When writing to R15 (= PC), a jump to the written value will occur.
When reading from R15, the result is the program counter value (= address) of the corresponding operation, which is reading from R15, plus 8 bytes.

3.3 Exceptions

Some processor modes cannot only be reached by modifying the mode bits of the CMSR
(bits 4 downto 0), but also can be accessed by special events (listed below). In this case, a branch to the corresponding trap address is performed. Furthermore the PC value (= address) of the last executed instruction +4 bytes, (→ return address) before the context change is stored in R14_<new_mode>.

Undefined I. mode:	The processor executes an undefined instruction.
Supervisor mode:	Execute the “software interrupt” instruction or reset the core.
Abort mode:	Take the MEM_ABORT/ PRF_ABORT pin to high level
FIQ/IRQ mode:	Take the corresponding pin (FIQ/IRQ) to high level.

3.5 Differences between ARM7 and the STORM CORE

Since the STORM CORE is a completely new approach of creating an ARM-native processor system, there are some differences. The noncritical ones do not affect the ARM-compatible behavior of the processor, so no code adaption is necessary in most cases. The critical differences may need a code adaption, when running programs on the STORM, which were originally created for an ARM.

3.5.1 Critical differences

- There is no coprocessor interface implemented. When trying to execute a coprocessor operation, the undefined instruction trap is taken.
- No “load/store multiple” instructions exist. When trying to execute LDM/STM instructions, the undefined instruction trap is taken.
- The prefetch interrupt is used as “instruction abort interrupt and” can be requested by the memory system via an external pin, for example when trying to load data as instructions.
- When doing shift operations with a register given shift distance, or when performing MAC operations, no additional data fetch from the register file is necessary. So, if R15 is an operand, it's value will always be the address of the corresponding data processing operation plus 8 bytes.

3.5.2 Noncritical differences

- There are no restrictions for the use of any register as operand (for example all registers in one instruction can be the same; also the PC can be used as operand or destination for any instruction).
- The ARM7 uses one bus system to fetch instructions and data. The STORM core uses two seperate bus systems to access data and instructions, but these ports can be combined by an IO access arbiter.
- When performing single memory access operations, the shift value, which is applied to the offset register value, can also be specified by the content of the data register.

3.5.3 Differences in STORM_MODE

The “STORM_MODE” parameter (→ *storm_core.vhd*) enables additional hardware features. When enabled, the processor is less compatible to the ARM7, but offers new functionality. This different/new functionality is listened below.

- The MVN operation (MVN Rd, Rx ↔ Rd := not Rx) is replaced by the NAND operation (NAND Rd, Rx, Ry ↔ Rd := Rx nand Ry),
- A pure arithmetical shift operation (MOV operation with applied shift) can set the overflow flag, when the sign of the original input data gets changed during the shift.

4. Core Hardware

This chapter is about the internal RTL structure of the STORM processor core.

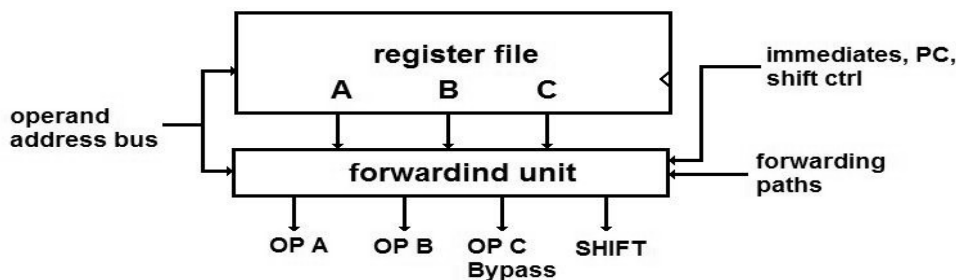
4.1 Data flow

The STORM pipeline consists of 7 internal stages. An external memory system needs at least one additional cycle to provide the data from the memory.
So the resulting eight stages of the instruction processing pipeline are:

```
1. IA:   Instruction access (program counter)
2. IF:   Instruction fetch (external memory )
3. ID:   Instruction decode
4. OF:   Operand fetch
5. MS:   Multiplication / Shift
6. EX:   ALU execution / MCR access
7. MA:   Memory access
8. WB:   Data write back
```

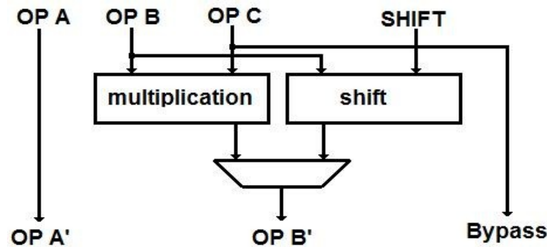
The STORM core processor uses two separate buses to load instruction and to load/store data.
An access arbiter (→ *system_bridge.vhd*) is used to coordinate the access, when both ports are trying to access the same resource, e.g. the internal working memory. A global halt is then applied to the core until both memory requests can be completed.
A valid memory request is indicated via the REQ signals of the data and instruction interface port. While the D_MEM_REQ (data interface) is only high when performing manual data access, the I_MEM_REQ (instruction interface) is always high, if no data dependencies occur and no branch was taken.

A new instruction cycle starts with the new value for the the program counter (→ *mcr_sys.vhd*). For normal 32 bit ARM operations, this value is "old_program_counter + 4 bytes". One clock cycle later, this signal is applied to the memory bus. The memory system accepts the instruction request and outputs it after another clock cycle to the instruction arbiter (→ *flow_ctrl.vhd*). This arbiter can buffer up to 3 instructions plus the current instruction in a small memory, so it allows a continuous instruction flow, even when doing branches or stopping the pipeline for some reason.
Another cycle later, the correct instruction word is applied to the instruction decoder (→ *x1_opcode_decoder.vhd*). In the next cycle, the decoded control signals arrive the first pipeline stage, the operand fetch stage. The operand fetch unit (→ *operand_unit.vhd*) loads register values from the register file, immediate values, current or past program counter values and outputs them to the MS unit. Furthermore, the forwarding circuit is placed in this unit, which is capable of loading already computed data results from later pipeline stages.

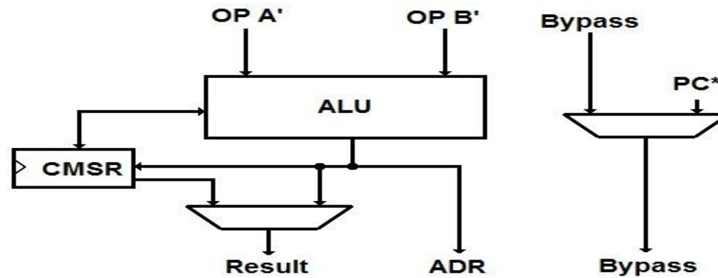


In parallel, the other control signals from the decoder are brought to the flow control, which serves as a large shift register for the control signals.

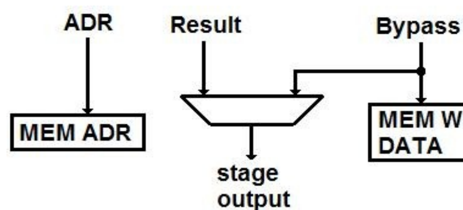
In the next stage, a multiplication or a shift is performed (MS stage).



The following stage is the main execution stage. The arithmetical and logical operations take place in this module. Also the current machine status register is located here. Flags can be read for proper arithmetical/logical operations and also the condition check is done in this stage (→ *flow_ctrl.vhd*). So all instructions, even with a not fulfilled condition code, are valid until this stage, if they were not marked as invalid by the instruction arbiter or the branch control.



The ALU can also perform write & read access to the machine control register file (→ *MCR.vhd*) file. When a rising clock edge appears, the CMSR is updated with the new flag values (arithmetical/logical flags, mode code, …) and also the PC is loaded with it's new value. The data result from the arithmetical/logical unit can either be the true arithmetical/logical calculation result, or the content of a machine status register (CMSR/SMSR). The bypass path is used for memory-write data and also for passing link addresses.



The next stage performs the memory access (MA). The correct address (instruction or data address) is send to the memory interface, all memory control signals are generated and the output data gets aligned in this pipeline stage.

The final stage is the data write back stage (WB). Read-data from an external memory is read into this stage, where it also gets aligned, depending on the read data quantity and the address offset, which is relative to the data transfer quantity (byte, half word, word). Data from the WB stage - either the read memory data or the stage output data of the previous stage - is directly written to the destination register in the register file.

4.2 Pipeline conflicts

When executing linear programs (no branches) without any dependencies between instructions in the pipeline, there are no pipeline conflicts. For all other cases, a logic is needed, that solves this conflicts. There are two different types of conflicts: Just to differentiate between them, I will call them "local pipeline conflicts" and "temporal pipeline conflicts".

4.2.1 Local pipeline conflicts

Local pipeline conflicts just mean, that the needed data for further processing has not yet reached the register file and is still somewhere else in the pipeline.

Program example:

```
ADD R1, R2, #1   (R1 = R2 + 1)
ADC R5, R4, #2   (R5 = R4 + Carry + 2)
SUB R3, R1, #1   (R3 = R1 - 1)
```

The SUB needs the result of the ADD. But when the SUB is in the operand fetch stage, the ADD just has reached the EX stage. Since the ADD instruction needs no further processing, the result is already correct. To avoid wait cycles until the value is written back to the register file, the forwarding unit loads the data directly from the EX stage into the operand fetch unit, where the forwarded result is used instead of the original value from R15.

The forwarding system can forward data from the EX stage, the MA stage and the WB stage, where earlier pipeline stages have higher priority than later ones. The unit itself is based in the *operand_unit.vhd* file.

4.2.2 Temporal pipeline conflicts

Temporal pipeline conflicts occur, when the processor is trying to forward a result, that has not been completely computed yet. So the conflict cannot be solved by forwarding data from some other pipeline stage, since the correct data does not exist yet.

Program example:

```
ADD R1, R2, #1   (R1 = R2 + 1)
SUB R3, R1, #1   (R3 = R1 - 1)
```

When the SUB instruction is in the operand fetch stage, the ADD is in the MS stage, so no addition has taken place yet. The processor can detect this conflict and stalls the instruction fetch for one cycle. That means, the ADD instruction can resume processing in the pipeline, while the SUB instruction is freezed in the OF stage until the needed data is available. The empty "slots" between this instruction (OF: SUB, MS: NOP, EX: ADD) are filled with "NOPs". This "no-operation" instruction does not perform any data manipulation.

Temporal data dependencies can occur in the OF, the MS and the EX stage, when trying to get not yet calculated data. The unit, which solves this conflicts, is the "Temporal Data Dependence Detector" in the *operand_unit.vhd* file, which communicates via the "halt_bus" directly with the instruction cycle arbiter in the *flow_ctrl.vhd* file.

4.2.3 Branches

There are three causes for a non linear change of the program counter:

- unconditional/conditional branches
- interrupts/exceptions
- manual writing to R15

All these operations result in a branch to a new address (new PC value). The PC gets updated with non-linear data (= when the new PC value is not “old_value + 4”) after the EX stage.

Example program:

```
CMP R0, R3         (compare R0 <-> R3)
BEQ subroutine     (branch if equal)
ADD R3, R0, R1     (obsolete)
EOR R5, R0, R1     (obsolete)
SUB R2, R0, R1     (obsolete)
```

When the branch instruction (“BEQ”) reaches the EX stage, the ADD is in the MS stage, the SUB is in the OF stage and the EOR is in ID stage. All the instructions, which are in earlier stages than the BEQ in the EX stage, have to be invalidated by the branch arbiter (“branch cycle arbiter” → *flow_ctrl.vhd* file). Until the processing can resume at the new position, the new address hast to be moved into the PC, send to the memory and the new opcode needs to be stored in the instruction arbiter, so the instruction processing - starting in the IA stage – needs to be disabled for the next 4 cycles.

4.3 Stage control bus

The stage control bus ("CTRL" in the VHDL modules) contains all signals, which determine the function of of a module of each pipeline stage.
Note: All control lines are propagated until the end of the pipeline, even they are not used in later stages. This holds the architecture flexible to changes, but may also result in synthesis warnings – ignore them ;)

Signal name	Data size	Location	Function
EN	1 bit	0	Enable signal, all other signals are valid when set to high.
CONST	1 bit	1	Operand B is an immediate value.
BRANCH	1 bit	2	Is branch operation. After EX stage: Branch taken.
LINK	1 bit	3	Is link operation, insert PC in *bypass* during EX stage.
SHIFTR	1 bit	4	Use shift offset from register.
WB_EN	1 bit	5	Register write back enabled.
RD_0 – RD_3	4 bit	6..9	Destination register address.
SWI	1 bit	10	Is software interrupt operation.
UDI	1 bit	11	Is undefined instruction.
COND_0 – COND_3	4 bit	12..15	Condition code.
MS	1 bit	16	Use shifter (low) or multiplier (high).
AF	1 bit	17	Alter flags when set to high.
ALU_FS_0 – ALU_FS_3	4 bit	18..21	ALU function select. ALU_FS_3 is low for logical operations and high for arithmetical operations.
MEM_ACC	1 bit	22	Perform data memory access.
MEM_DQ_0 – MEM_DQ_1	2 bit	23..24	Data memory transfer quantity. "00" = word, "01" = byte, "10"/"11" = half word.
MEM_SE	1 bit	25	Use sign extension for memory data when set to high.
MEM_RW	1 bit	26	Data memory read (low) / write (high) access.
MEM_USER	1 bit	27	Access data memory in user mode when set to high.
MREG_ACC	1 bit	28	Access internal machine register file.
MREG_M	1 bit	29	Access CMSR (low) / SMSR (high).
MREG_RW	1 bit	30	Machine register file read (low) / write (high) access.
MREG_FA	1 bit	31	Flag access only when set to high.
MODE_0 – MODE_4	5 bit	17..21	Bus locations are re-used after EX stage: Current processor operation mode.

5. Storm Core System

A comfortable way of integrating the STROM core processor into you own project is given by the "STORM Core System" (→ *storm_top.vhd*). This system setup provides an internal working memory and Wishbone bus interface to communicate with other modules.

5.1 System Bridge

The access to the two resources is controlled by an access arbiter (→ *system_bridge.vhd*), which is capable of letting both bus systems of the core (data and instruction interface) access both resources (memory and Wishbone interface). So it is also possible to execute instructions from a memory, that is attached to the Wishbone bus.

Each resource has to acknowledge the access. Both resources have a time out counter in the system bridge, which triggers an interrupt, when no acknowledge has been received for a specific time (configurable in the *storm_top.vhd)*. For example, when the access came from the instruction interface of the core and is has run out of time (no acknowledge received), the instruction abort (= prefetch abort) interrupt is taken.

The **IO_BORDER** constant in the *storm_top.vhd* gives the address border between the internal memory and the Wishbone interface. Also this constant gives the absolute size of the internal memory in 32 bit cells. The address outputs of the core selects the resource to be accessed:

```
CORE_ADR_O <  IO_BORDER : Internal working memory access
CORE_ADR_O >= IO_BORDER : IO access via Wishbone interface
```

5.2 Internal Memory

An internal working memory can be implemented, if the IO_BORDER constant is set to a value higher than zero (zero → internal memory disabled). The memory itself is based on four memory blocks, each holding an eight bit subword of the total 32 bit data word, so it can easily be implemented by the synthesis tool using memory macro blocks (block ram, distributed ram, …). This memory can be configured to have an preloaded image (after downloading the configuration into the FPGA), to have a basic firmware to star up the system.

When simulating the core system, you can use the memory's *DEBUG_MEM* signal to get an easy overview of the current data content.

6. Getting started

This chapter will give a brief introduction of how to setup the STORM core processor on your system. All you need is a VHDL editor and a FPGA synthesis and/or simulation tool. For example you can use Notepad++ and the Xilinx ISE with it's inbuilt ISIM simulator.

6.1 Hardware setup & simulation

First of all download the most recent source code of the STORM core processor from the SVN repository at http://www.opencores.com/project,storm_core. Start your evaluation tool (Xilinx ISE, Altera's Quartus, Model Sim, etc) and create a new project, adding all the files from the core's *"rtl"* directory. All the needed files are listed in chapter 1.1.

The *storm_top.vhd* is the top entity of the complete processor system, including the core itself, inbuilt memory and a Wishbone master interface. You can embed this entity into you own project to create a complete system on chip. Simply connect the ports of the storm_top.vhd to a Wishbone bus fabric and add all the modules you need. Do not panic, if the synthesis tool outputs a lot of warnings telling that there are unconnected signals. These signals are for debugging use and/or keeping the core extendable for future versions.

It is also possible to use the s*torm_top_tb.vhd* as a simple testbench base, if you want to simulate programs on the core system directly. When using the inbuilt simulator of the Xilinx ISE (ISIM), you can use the debug waveform from the *"sim"* folder to get an overview of all the important internal signals.

6.2 Software setup

In the project folder *"arm asm"* you can find the arm-elf-asm assembler. With this tool you can assemble your ARM assembler programs into STORM compatible opcodes. For easy software setup, you can use the compile.bat batch file. It uses the *"test.s"* file as asm input file and creates the mnemonic.txt text file. You can copy the content of this text file into the memory image area in the core's working memory (→ *memory.vhd*). This enables to implement a start-up code into the core's memory, e.g. for bootloader programs (or of course for basic debugging).