STORM CORE Processor System
by Stephan Nolting

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 Mentor 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 Processor System and it's documentary can be found at
http://www.opencores.com/project,storm_core

Table of content

1. Introduction
- **1.1** STORM Core Features
- **1.2** VHDL File Hierarchy
- **1.3** System Architecture
- **1.4** STORM_TOP Interface Signals

2. Core Programmer Model
- **2.1** Differences Between ARM and STORM Core
  - **2.1.1** Critical Differences
  - **2.1.2** Noncritical Differences
- **2.2** Operating Modes
- **2.3** Registers
- **2.4** Exceptions / Interrupts

3. Core Hardware
- **3.1** Module Description
- **3.2** Data Flow
- **3.3** Cache Access
  - **3.3.1** IO → Cache Coherency
  - **3.3.2** Cache → IO Coherency
- **3.4** Pipeline conflicts
  - **3.4.1** Local Pipeline Conflicts
  - **3.4.2** Temporal Pipeline Conflicts
  - **3.4.3** Branches
- **3.5** Stage Control Bus

4. Coprocessor Interface
- **4.1** System Control Processor Register Set

5. Getting Started
- **5.1** Hardware Setup / Simulation
- **5.2** Software Setup Using Assembler (arm-elf)
- **5.3** Software Setup Using C (WinARM)

1. Introduction

The STORM Core processor system is a powerful open source soft core processor for FPGA implementation. It is completely described in the hardware description language VHDL and uses no dedicated hardware components, so it can be synthesized for any configurable logic 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 ARM and STORM Core".

1.1 STORM Core Features

- ✔ Opcode and function compatible to ARM's 32-bit instruction set format
- ✔ 32-bit RISC open source soft-core processor
- ✔ 7 different operating modes with unique register sets
- ✔ 4 external interrupt request signals
- ✔ Internal coprocessor for system management
- ✔ Internal 32-bit timer and LFSR
- ✔ Pipelined instruction execution (8 stages)
- ✔ Completely described in behavioral VHDL - no instantiated hardware primitives; coded to make use of dedicated hardware components (multiplier, memory, carry-chain)
- ✔ Configurable I-cache and D-cache as well as D-cache coherency strategy
- ✔ 32-bit Wishbone bus interface
- ✔ Different clock speeds for bus interface and processor core

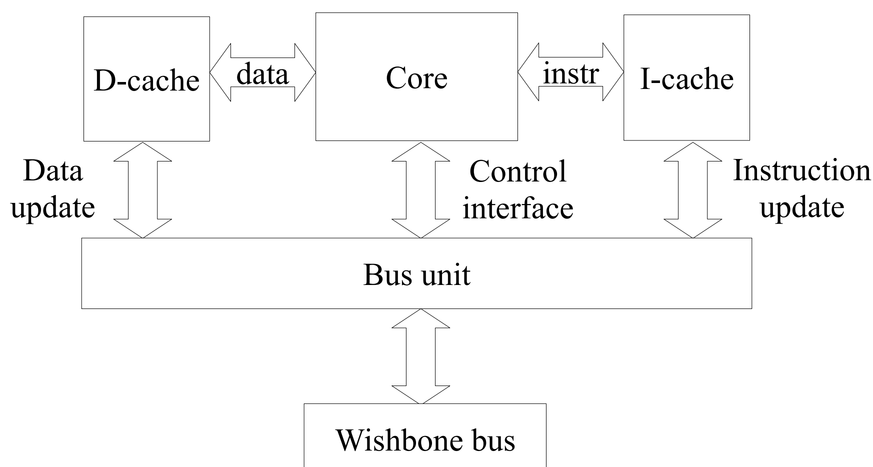
1.2 VHDL File Hierarchy

All needed files are located in the “storm core/rtl” folder.

```
STORM_TOP.vhd
-> BUS_UNIT.vhd
-> CACHE.vhd
-> CORE_PKG.vhd
-> CORE.vhd
   -> OPCODE_DECODER.vhd
   -> FLOW_CTRL.vhd
   -> MC_SYS.vhd
   -> REG_FILE.vhd
   -> OPERAND_UNIT.vhd
   -> MS_UNIT.vhd
      -> MULTIPLY_UNIT.vhd
      -> BARREL_SHIFTER.vhd
   -> ALU.vhd
   -> LOAD_STORE_UNIT.vhd
   -> WB_UNIT.vhd
```

1.3 System Architecture

To increase the performance of the core, the system is equipped with two cache units: A data cache and an instruction cache. Both caches are full associative and can store any data from/to any IO location. The number of cache pages as well as the page size can be configured for each cache independently.
The default coherency strategy for the data cache is “Write-Thru”, so any manipulation of the cache leads to an immediate write back to the IO system.



An example of a compatible Wishbone fabric can be found in the STORM_core_TB.vhd (sim folder).
For more information about the Wishbone architecture, see the Wishbone data sheet in the doc folder.

1.4 STORM_TOP Interface Signals

This are the interface signals and generics of the system's top entity "**STORM_TOP.vhd**".
The type of the core ports are **std_logic** and **std_logic_vector**. All control signals are high-active, so connect all unused inputs to logical zero.

Generic	Generic type	Function
I_CACHE_PAGES	natural	Number of pages in I-Cache
I_CACHE_PAGE_SIZE	natural	I-Cache page size (32-bit words)
D_CACHE_PAGES	natural	Number of pages in D-Cache
D_CACHE_PAGE_SIZE	natural	D-Cache page size (32-bit words)
TIME_OUT_VAL	natural	Maximum Wishbone bus cycle length
BOOT_VECTOR	std_logic_vector(31:0)	Boot vector address

Signal	Signal size	Direction	Function
CORE_CLK_I	1 bit	Input	Core clock signal, triggering on rising edge
BUS_CLK_I	1 bit	Input	Wishbone bus clock signal, triggering on rising edge
RST_I	1 bit	Input	Reset signal, high active, synchronous to core / bus clock
WB_ADR_O	32 bit	Output	Wishbone bus address
WB_CTI_O	3 bit	Output	Wishbone bus cycle type
WB_TGD_O	6 bit	Output	Wishbone bus cycle tag; bits 4 downto 0: current processor mode, bit 5: instruction('1') / data fetch('0')
WB_SEL_O	4 bit	Output	Wishbone bus byte select
WB_WE_O	1 bit	Output	Wishbone bus write enable
WB_DATA_O	32 bit	Output	Wishbone bus data output
WB_DATA_I	32 bit	Input	Wishbone bus data input
WB_STB_O	1 bit	Output	Wishbone bus valid transfer
WB_CYC_O	1 bit	Output	Wishbone bus valid cycle
WB_ACK_I	1 bit	Input	Wishbone bus acknowledge signal
WB_HALT_I	1 bit	Input	Wishbone bus halt request
IRQ_I	1 bit	Input	Interrupt request
FIQ_I	1 bit	Input	Fast interrupt request

For more information about the Wishbone bus, see the Wishbone data sheet, which can also be found in the "doc" folder.

2. Core Programmer Model

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

2.1 Differences between ARM 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 adaptions are necessary in most cases. The critical differences may need a code adaption, when running programs on the STORM Core, which were originally created for an ARM.

2.1.1 Critical Differences

- No multiply-long and multiply-accumulate-long instructions are implemented yet. Executing such an instruction will trigger the undefined instruction trap.
- No branch and exchange instruction in implemented, since the processor does not support any short instruction format.
- Block data transfer instructions do not support the S-bit option, so no user bank transfers or mode changes are possible with this instruction. Setting the S-bit will trigger the undefined instruction trap.
- The prefetch abort interrupt is used as instruction fetch abort interrupt (IAB).
- The data abort interrupt is used as data fetch abort interrupt (DAB).
- When doing shift operations with a register given shift offset, 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.

2.1.2 Noncritical Differences

- There are no restrictions for the use of any register as operand/destination for all instructions (for example all registers in one instruction can be the same; also the PC can be used as operand or destination for any instruction).
- 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 (not intended in ARM code).
- Data bits 8 and 9 of the machine status register are not undefined/reserved, they are used for disabling the DAB and IAB external interrupts (when set to '1').

2.2 Operating Modes

Up to six different operation modes are supported by the STORM Core. After reset, the processor starts operation always in system mode. To change to a different mode, the corresponding *MODE* code has to be written to the lowest 5 bit of the CMSR (CPSR in ARM). This is only possible when the processor is in privileged mode (any other mode than user mode).

Mode	Interrupt base address	Mode code
User, USR	-	"10000"
Undefined Instruction, UND	0x00000004	"11011"
Supervisor, SVP	0x00000008	"10011"
System, SYS	0x00000008	"11111"
Abort, IAB	0x0000000C	"10111"
Abort, DAB	0x00000010	"10111"
reserved	0x00000014	-
Interrupt Request, IRQ	0x00000018	"10010"
Fast Interrupt Request, FIQ	0x0000001C	"10001"

2.3 Registers

The STORM Core provides seven 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), the current machine status register (CMSR) and a saved machine status register (SMSR_<mode> (SPSR in ARM)).

Mode	Accessible data registers	Accessible machine registers
USR	R0, …, R14	PC, CMSR
FIQ	R0, …, R07, R08_FIQ, …, R14_FIQ	PC, CMSR, SMSR_FIQ
IRQ	R0, …, R12, R13_FIQ, R14_FIQ	PC, CMSR, SMSR_IRQ
SVP	R0, …, R12, R13_SVP, R14_SVP	PC, CMSR, SMSR_SVP
ABT	R0, …, R12, R13_ABT, R14_ABT	PC, CMSR, SMSR_ABT
UND	R0, …, R12, R13_UND, R14_UND	PC, CMSR, SMSR_UND
SYS	R0, …, R12, R13_SVP, R14_SVP	PC, CMSR, SMSR_SYS

Note: Supervisor mode (SVP) and System mode (SYS) share the same data registers, but have unique saved machine status registers (SMSR_SVP and SMSR_SYS).

Note: A write-access to R15 (PC) will result in a jump to the written value (address).
When reading from R15, the result is the program counter value (address) of the corresponding operation, which is reading from R15, plus 8 bytes.

All data registers (r0 - r14) are located in the main register file, but only a special set of those is available at one time (depending on the current processor operation mode). The mapping of the data registers to block memory locations is listed below:

```
00: USR32 R00    08: USR32 R08    16: FIQ32 R09    24: ABT32 R13
01: USR32 R01    09: USR32 R09    17: FIQ32 R10    25: ABT32 R14
02: USR32 R02    10: USR32 R10    18: FIQ32 R11    26: IRQ32 R13
03: USR32 R03    11: USR32 R11    19: FIQ32 R12    27: IRQ32 R14
04: USR32 R04    12: USR32 R12    20: FIQ32 R13    28: UND32 R13
05: USR32 R05    13: USR32 R13    21: FIQ32 R14    29: UND32 R14
06: USR32 R06    14: USR32 R14    22: SVP32 R13    30: Dummy Reg
07: USR32 R07    15: FIQ32 R08    23: SVP32 R14    31: Dummy Reg
```

Note: R14 of each mode is used as the corresponding Link Register to store the jump-back address. R13 of each mode is commonly used as Stack Pointer.

Note: Since the PC is not located in the main register file, writing to R15 (PC) will perform a write to a dummy register. Reading the PC will not fetch the value from this dummy registers but will fetch data from the PC directly (plus 8 bytes offset).

2.4 Exceptions / Interrupts

Some processor modes can also be entered by special events (listed below). In this case, an interrupt is taken respectively and exception trap is entered (external interrupts must be enabled in CMSR).

Mode	How to get there
UDI	Execute an undefined instruction
FIQ	Set the FIQ pin to '1'
IRQ	Set the IRQ pin to '1'
ABT	Set the instruction fetch abort pin (I-Abort) or the data fetch abort pin (D-Abort) to '1'
SVP	Execute the “SWI” instruction

Whenever a valid interrupt is taken, the processors does the following operations:

- ➔ Save the jump-back (link) address to the new mode's link register
- ➔ Copy the current machine status register (CMSR) to the corresponding saved machine status register (SMSR) of the new mode
- ➔ If the source of the interrupt is an external pin (IRQ, FIQ, IAB, DAB), disable the corresponding interrupt-enable-bit in the CMSR
- ➔ The processor resumes operation at the corresponding interrupt base address

Internal interrupts, such as software and undefined instruction interrupts, are always triggered by specific opcodes. For example, the SVP trap is entered by executing the SWI instruction. So, such interrupt sources do not need a synchronization into the STORM's pipeline.

External interrupts (DAB, IAB, FIQ, IRQ) can occur at any time and asynchronous to the pipeline.
When a valid external interrupt request appears, the instruction fetch of the core is stopped and the pipeline continuous operation until all instruction, which are currently in the pipeline, have finished. Afterwards, the processor changes the operation mode and executes the branch-and-link operation to jump to the corresponding entry in the interrupt vector table.

If there are several interrupt requests at the same time, the one with the highest priority is executed. All other pending interrupt requests will be stored, so they can be executed after the interrupt handler has finished. The interrupt priority list is listed below:

Priority	Interrupt
1 (highest)	DAB: Data fetch abort
2	FIQ: Fast interrupt request
3	IRQ: Interrupt request
4	IAB: instruction fetch abort
5	UND: Undefined instruction
6 (lowest)	SVP: Software interrupt

External interrupts can be disabled by setting the corresponding interrupt enable bit in the current machine status register (CMSR) to '1'.

CMSR bit #	Bit name	Default	Interrupt
6	SREG_FIQ_DIS	1	Fast interrupt request
7	SREG_IRQ_DIS	1	Interrupt request
8*	SREG_DAB_DIS	1	Data fetch abort
9*	SREG_IAB_DIS	1	Instruction fetch abort

*) Note: This functionality is not ARM-compatible. In ARM processors, these bits are reserved and the corresponding interrupts are always enabled.

3. Core Hardware

This chapter is about the internal RTL structure of the STORM Core processor.
All parts of the architecture are written using behavioral VHDL. Even if no dedicated hardware component are instantiated, the coding style allows the EDA tools to map some modules to dedicated hardware components (e.g. memories, multiplier, adders).

3.1 Module Description

File name	Functional description
ALU.vhd	The ALU holds the primary data operation unit. All address-operations are calculated here (except for the program counter increment). Furthermore it handles the data access to the machine control registers and to the system coprocessor registers.
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 value directly from the opcode or a register value.
BUS_UNIT.vhd	The bus unit presents the Wishbone bus interface. Data and instruction fetch to the caches are coordinated by this unit. It can operate with a different clock, than the core itself.
CACHE.vhd	This is the basic component for the instruction (IC) and data cache (DC). The cache is full-associative and can be mapped to dedicated memory blocks.
CORE.vhd	The CORE.vhd is the top entity of the STORM processor core.
CORE_PKG.vhd	This file is the main package file, where all modules and parameters are defined.
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 instruction arbiter, the cycle arbiter, which solves temporal pipeline conflicts, the branch arbiter and the condition check system are located here.
LOAD_STORE_UNIT.vhd	The load-store unit outputs the address and the control signals to the data cache access port.

File name	Functional description
MC_SYS.vhd	The MC system holds the machine control circuits, which include the program counter, the current and saved machine status register as well as the interrupt handler, the branch system and the context change system. Also the internal system control coprocessor is 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.
MULLTIPLY_UNIT.vhd	The multiply unit calculates a 32x32 bit operation and outputs the lower 32 bits of the result to the ALU data path B.
OPCODE_DECODER.vhd	This unit decodes the ARM-native 32-bit opcodes into processor control signals.
OPERAND_UNIT.vhd	This 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 pipeline data conflict detector and the forwarding system are located here.
REG_FILE.vhd	This unit contains the main data register file. It consists of 32 registers, whereof 16 are accessible at one time, depending on the current operating mode. The registers are mapped to three memory blocks to create three read data read ports while efficiently using the hardware.
STORM_TOP.vhd	This is the top entity of the complete processor system. It inlcudes the processor core, data and instruction cache and a Wishbone compatible bus unit.
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 cache interface.

3.2 Data Flow

The STORM pipeline consists of 8 stages:

```
1. IA:  Instruction access (program counter)
2. IF:  Instruction fetch (I-cache access)
3. ID:  Instruction decode
4. OF:  Operand fetch
5. MS:  Multiplication / Shift
6. EX:  Execution
7. MA:  Memory access
8. WB:  Data write back
```

Stage	Functional description
1. IA	A new instruction cycle starts with the output of the new value for the the program counter, which is old_value + 4, since all instructions are 32 bit wide and have to be aligned.
2. IF	The instruction cache accepts the instruction request and outputs the requested data (if available). If the requested cache line is not available, a new cache page gets updated with the needed data set (see next chapter for more information).
3. ID	In the next cycle, the instruction is loaded into the instruction register and the instruction decoder decodes the applied opcode into internal control signals.
4. OF	The decoded control information loads the needed registers from the register bank. Also the forwarding system takes action in this cycle to fetch operand if there are any data conflicts.
5. MS	In this stage, a multiplication or a shift of the operands can be applied.
6. EX	The following stage is the main execution stage. The arithmetical and logical operations take place in this module. Also, values from the machine status registers or the coprocessors can be loaded here and also the condition check is done in this stage. 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.
7. MA	The next stage performs the memory access and also can update the machine status registers, the PC and the coprocessor registers. The data address and all needed control signals are send to the D-cache. Furthermore the write-data gets aligned if necessary and is also brought to the data memory interface.
8. WB	The final stage is the data write back stage. Read-data from the D-cache is read into this stage, where it gets aligned, depending on the read data quantity and the address offset. Data from the WB stage - either the read memory data or the stage output data of the previous stage - is directly written on the next rising clock edge to the destination register in the data register file. The data flow resumes in the operand fetch stage.

3.3 Cache access

If a requested data entry is not available in a cache memory, a new cache page will be updated with the needed data. This can take several cycles, depending on the cache's page size, the speed of the bus system and the speed of the accessed IO device (e.g. memory).

If the IO access takes more than a maximum value, which can be specified using system coprocessor, the IAB interrupt is taken, when the bus unit was fetching data for the I-cache, or the DAB interrupt is taken, when the bus unit was fetching data for the D-cache.

Re-updating (invalidating all cache entries to get the most recent data from the memory/IO system) and flushing (copying all cache pages to the memory/IO system) the cache manually can be done by using the system control coprocessor.

3.3.1 IO → Cache coherency

The user has to take care, that the cache has always the recent data from the IO/memory system.
For example by using an external interrupt (IRQ) an IO device can show that it's data contents has changed and so it is up to the interrupt handler to re-update the cache content. Of course it is also possible to re-update the cache before any IO access takes place, but this might be inefficient for cache pages sizes bigger than one.

3.3.2 Cache → IO coherency

When using the default “Write-Thru” coherency strategy, any modification of a cache entry leads to a complete write back of the corresponding cache page into the memory/IO system. This might cause problems for IO, which trigger their operation on write-access bus cycles.
Disabling the “Write-Thru” strategy in the system control coprocessor introduces a new coherency strategy, where a modified page is only written back to the IO system when the processor is no longer accessing this page (*feature is BETA version!*).

For example, imagine an UART, which starts transmitting its data register every time, when a write access to this register occur. So if a program has changed a value in a cache page and this UART data register is also part of the cached page, the UART would trigger it's transmission when the cache page is written back to the IO system, even if no UART access was intended.

To solve this problem, you can either set the D-cache page size to 1, or you have to make sure, that no IO device triggers it's operation on writing bus cycles (so use control bits in the IO devices instead).

3.4 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, an arbiter logic is needed, that solves this conflicts. There are two different types of conflicts: Just to differentiate between them, they will be called “local” and “temporal” pipeline conflicts.

3.4.1 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.

3.4.2 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.

3.4.3 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 PC value. The PC gets updated with non-linear data (= when the new PC value is not "old_value + 4") on a rising edge between EX and MA 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 EOR is in the OF stage and the SUB 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 register, so the instruction processing - starting in the IA stage – needs to be disabled for the next 3 cycles, which are necessary to load the next valid instruction until the OF stage.

3.5 Stage Control Bus

The control bus (CTRL) is generated in the opcode decoder and contains all the signals, which are needed to determine the single operations of an instruction. For each pipeline stage, the bus is registered in the FLOW_CTRL. Some signals, like the enable signal, are recalculated during the pipeline flow.
To keep the design flexible for future changes, all signals are propagated through the whole pipeline.

<table>
<tr><th>Bit #</th><th>Signal name</th><th colspan="2">Function</th></tr>
<tr><td>0</td><td>CTRL_EN</td><td colspan="2">Enable signal, all other signals are valid when set to '1'</td></tr>
<tr><td>1</td><td>CTRL_CONST</td><td colspan="2">Second operand is an immediate</td></tr>
<tr><td>2</td><td>CTRL_BRANCH</td><td colspan="2">Is branch operation</td></tr>
<tr><td>3</td><td>CTRL_LINK</td><td colspan="2">Is link operation</td></tr>
<tr><td>4</td><td>CTRL_SHIFTR</td><td colspan="2">Use register value as shift value</td></tr>
<tr><td>5</td><td>CTRL_WB_EN</td><td colspan="2">Enable write-back to register file</td></tr>
<tr><td>6</td><td>CTRL_RD_0</td><td colspan="2" rowspan="4">Destination register address</td></tr>
<tr><td>7</td><td>CTRL_RD_1</td></tr>
<tr><td>8</td><td>CTRL_RD_2</td></tr>
<tr><td>9</td><td>CTRL_RD_3</td></tr>
<tr><td>10</td><td>CTRL_SWI</td><td colspan="2">Is software interrupt instruction</td></tr>
<tr><td>11</td><td>CTRL_UND</td><td colspan="2">Is undefined instruction</td></tr>
<tr><td>12</td><td>CTRL_COND_0</td><td colspan="2" rowspan="4">Condition code</td></tr>
<tr><td>13</td><td>CTRL_COND_1</td></tr>
<tr><td>14</td><td>CTRL_COND_2</td></tr>
<tr><td>15</td><td>CTRL_COND_3</td></tr>
<tr><td>16</td><td>CTRL_MS</td><td colspan="2">Use shifter ('0') or multiplier ('1')</td></tr>
<tr><td rowspan="2">17</td><td>CTRL_AF</td><td rowspan="2">Alter ALU flags</td><td rowspan="10">*) Signals are re-used for the processor operating mode after EX stage</td></tr>
<tr><td>CTRL_MODE_0*</td></tr>
<tr><td rowspan="2">18</td><td>CTRL_ALU_FS_0</td><td rowspan="8">ALU function select</td></tr>
<tr><td>CTRL_MODE_1*</td></tr>
<tr><td rowspan="2">19</td><td>CTRL_ALU_FS_1</td></tr>
<tr><td>CTRL_MODE_2*</td></tr>
<tr><td rowspan="2">20</td><td>CTRL_ALU_FS_2</td></tr>
<tr><td>CTRL_MODE_3*</td></tr>
<tr><td rowspan="2">21</td><td>CTRL_ALU_FS_3</td></tr>
<tr><td>CTRL_MODE_4*</td></tr>
</table>

Bit #	Signal name	Function
22	CTRL_MEM_ACC	Data cache access
23	CTRL_MEM_DQ_0	Transfer data quantity “00” → Word, “01” → Byte, “10”/”11” → Half word
24	CTRL_MEM_DQ_1	
25	CTRL_MEM_SE	Use sign extension for cache read
26	CTRL_MEM_RW	Data cache read ('0') / write ('1') access
27	CTRL_MEM_USER	Access d-cache in user mode
28	CTRL_MREG_ACC	Access machine register file
29	CTRL_MREG_M	Access CMSR ('0') / SMSR ('1')
30	CTRL_MREG_RW	MREG read ('0') / write ('1') access
31	CTRL_MREG_FA	Full access ('0') / flag access only ('1')
32	CTRL_CP_ACC	Access coprocessor
33	CTRL_CP_RW	Coprocessor read ('0') / write ('1') access
34	CTRL_CP_REG_0	Coprocessor source / destination register address
35	CTRL_CP_REG_1	
36	CTRL_CP_REG_2	
37	CTRL_CP_REG_3	
38	CTRL_SHIFT_M_0	Barrelshifter shift mode
39	CTRL_SHIFT_M_1	
40	CTRL_SHIFT_V_0	Barrelshifter shift value
41	CTRL_SHIFT_V_1	
42	CTRL_SHIFT_V_2	
43	CTRL_SHIFT_V_3	
44	CTRL_SHIFT_V_4	

4. Coprocessor Interface

The STORM Core provides no interface for external coprocessors yet. But nevertheless, it is equipped with an internal coprocessor unit to give access to different system control features. This coprocessor is mapped to coprocessor number 15. When trying to access any other coprocessor than CP 15 or if any other coprocessor instruction than coprocessor-register-transfer is executed, the undefined instruction trap is taken. Also, a write access to the coprocessor, which is not done in privileged mode, triggers the undefined instruction trap.

Note: The operation bit-fields in MCR and MRC instructions are ignored by the processor.

Register number	Register name	R/W	Function
0	ID_REG_0	r	Core update date
1	ID_REG_1	r	ID_0
2	ID_REG_2	r	ID_1
3	*reserved*	r	*reserved*
4	*reserved*	r	*reserved*
5	*reserved*	r	*reserved*
6	SYS_CTRL_0	r/w	This register gives access to different system control functions
7	*reserved*	r	*reserved*
8	CSTAT	r	Current cache hit-rate statistics
9	TIME_THRES	r/w	Internal Timer: Threshold value
10	TIME_COUNT	r/w	Internal Timer: Counter register
11	LFSR_POLY	r/w	Internal LFSR: Polynomial register
12	LFSR_DATA	r/w	Internal LFSR: Data register
13	*reserved*	r	*reserved*
14	*reserved*	r	*reserved*
15	*reserved*	r	*reserved*

4.1 System Control Processor Register Set

ID Register 0, 1, 2

This registers present basic information about the STORM Core Processor.

CP Reg	Register	Bit	r/w	default	Function
0	ID_REG_0	31 .. 16	r	2012	Core version update date, year
		15 .. 08	r	1	Core version update date, month
		07 .. 00	r	14	Core version update date, day
1	ID_REG_1	31 .. 00	r	"StNo"	ID_0, 4 ASCII symbols
2	ID_REG_2	31 .. 00	r	"4788"	ID_1, 4 ASCII symbols

System Control Register 0

The system control register gives access to additional system configuration options.
For basic processor operation, no change of this register is necessary.

CP Reg	Bit	Name	Default	Function
6	0	DC_FLUSH	0	Flush (write back) D-Cache, auto-reset to '0' after execution
	1	DC_CLEAR	0	Clear D-Cache (reload cache), auto-reset to '0' after execution
	2	IC_CLEAR	0	Clear I-Cache (reload cache), auto-reset to '0' after execution
	3	C_WTHRU	1	Enable write-through coherency strategy for D-Cache
	4	*reserved*	0	*reserved*
	5	F_RST	0	Force system reset, auto-reset to '0' after execution
	6	FREEZE	0	Shutdown processor until external reset
	7	TIME_EN	0	Internal timer enable
	8	TIME_INT	0	Enable interrupt trigger for internal timer
	9	TIME_M	0	Trigger IRQ ('0') or FIQ ('1') interrupt
	10	LFSR_EN	0	Internal LFSR enable
	11	LFSR_M	0	New data after core clock ('0') or after data reg read-access ('1')
	12	LFSR_D	0	LFSR shift direction ('0': right, '1': left)
	13..14	*reserved*	0	*reserved*
	15..31	MBC	512	Maximum Wishbone bus cycle length

Cache Hit Rate Statistics Register

This register gives basic information about the D/I-cache hit statistics. Every hit access increments the corresponding counter. A miss access resets the corresponding counter.

CP Reg	Bit	Function
8	31 .. 16	D-Cache hit statistics, Hex FFFF is maximum value → Cache hit rate is one
	15 .. 00	I-Cache hit statistics, Hex FFFF is maximum value → Cache hit rate is one

Internal Timer

The STORM Core is equipped with an internal timer system. If enabled (via the TIME_EN bit in the system control register), the TIME_COUNT register gets incremented on every rising edge of the core clock. When the TIME_COUNT register value is equal to the TIME_THRES register value, the TIME_COUNT register is reset to zero. Also an “external” interrupt can be triggered when the TIME_INT bit is set. Via the TIME_M bit a FIQ or an IRQ can be selected.

Note: When the internal timer uses the IRQ/FIQ interrupt trap, the corresponding external interrupt pin is disabled and any signals on that pin are ignored.

CP Reg	Register	Function
9	TIME_THRES	Internal timer threshold register. Timer restarts (and triggers interrupt if enabled) when TIME_COUNT reaches the value of TIME_THRES
10	TIME_COUNT	Internal timer counter register, counting with core clock

Internal Linear Feedback Shift Register (LFSR)

An internal LFSR is also supported by the system coprocessor. LFSR_POLY contain the polynomial for the feedback. LFSR_DATA represents the shifted data f the LFSR. The LFSR is activated by the LFSR_EN bit. It's shift direction can be set by the LFSR_D bit.
An update (next LFSR value) can either be generated on every core clock tick (setting LFSR_M to '0' or after every read-access to the LFSR_DATA register (setting LFSR_M to '1').

CP Reg	Register	Function
11	LFSR_POLY	Polynomial register for internal LFSR
12	LFSR_DATA	Internal LFSR data register

5. Getting Started
5.1 Hardware Setup / Simulation

Start your evaluation tool (Xilinx ISE, Altera's Quartus II, Model Sim, etc) and create a new project, adding all the files from the project's **rtl** directory. All the needed files are listed in chapter 1.2.

The **storm_top.vhd** is the top entity of the complete processor system. Instantiate this component in your design, configure all the generics and connect the ports to a Wishbone compatible switching fabric.

A basic setup of a simple SoC, including a compatible Wishbone fabric and bus system together with a memory, can be found in the STORM_core_TB.vhd file (sim folder). This file can also be used for simulating the core. When using Xilinx ISIM, a basic waveform from the "sim/Xilinx ISIM" can be used to have an overview of all important core signals.

5.2 Software Setup Using Assembler (arm-elf)

```
→ arm-elf-as.exe : The arm-elf assembler
→ extract.exe    : The mnemonic extractor
→ macro.inc      : Assembler macros
→ main.asm       : Main program file
→ make.bat       : Processing batch file
```

The folder "software/ASM" contains the arm-elf-asm assembler. With this tool, assembler programs can directly be converted into ARM-compatible opcodes. For easy software processing, the *make.bat* batch file can be used. The *main.asm* is the main program file. It includes the *macro.inc*, which supports some useful assembler macros.
Executing the "make" batch file will process the main.asm and all included project files. It generates the *a.out* opcode file, from which the mnemonic extractor (*extract.exe*) extracts the binaries for the program memory of the processor core. The *mnemonic.txt* contains the opcodes as VHDL memory initialization construct, which can be directly copied into the memory's vhdl file (→ MEMORY.vhd). The *mnemonic.dat* contains the opcodes in binary format, so this file can be used for programming via bootloader.

5.3 Software Setup Using C (WinARM)

```
→ build/STORMcore-RAM.ld : Linker script file
→ extractor.exe          : The mnemonic extractor
→ main.c                 : Main program file
→ makefile               : Processing batch file
→ storm_core.h           : STORM register definitions
```

The folder "software/C" contains the basic pattern for the setup of a C software project for the STORM Core. If you are using WinARM, simply edit the *main.c* and execute the make file after wards. Just like the mnemonic exdtractor from the ASM project folder, the extractor from the C project folder will ouput a *mnemonic.txt* for direct VHDL memory initialization and a *mnemonic.dat* for e.g. bootloader transfer.

If you do not use the make file, make sure to setup the compiler correctly, so it only creates 32-bit ARM opcodes.