STORM CORE Processor System
by Stephan Nolting

Proprietary Notice

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

Welcome to the STORM Core Processor project!

This core started as a personal research project to get into the basic of digital processing circuits. I always wanted to know how a processor works at the basic gate level. The STORM Core is the result of this investigation – hopefully someone out there can learn as much from it as I did ;)

The core itself provides native functionality, operation codes and programmer's models to ARM's famous 32-bit processor family (→ ARM7 / AMR9). See chapter 2 for more information.

1.1 STORM Core Features

- ✔ Opcode and function compatible to ARM's 32-bit instruction set family
- ✔ 32-bit RISC open source soft-core processor
- ✔ Pipelined instruction execution (8 stages)
- ✔ Single cycle execution of all operations (except for branch and multi-cycle memory operations)
- ✔ 7 different operating modes with unique register sets and privileges
- ✔ 4 external interrupt request signals
- ✔ Internal coprocessor for system management
- ✔ Internal 32-bit LFSR
- ✔ System IO port (16x in, 16x out)
- ✔ Completely described in behavioral **VHDL** - no instantiated hardware primitives; coded to allow the synthesis tool 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 pipelined Wishbone bus interface
- ✔ Up to 80Mhz operating frequency (@ 85% device utilization) on a Xilinx Spartan-3 XC3S400A
- ✔ Compatible with arm-elf assembler and WinARM tool chain

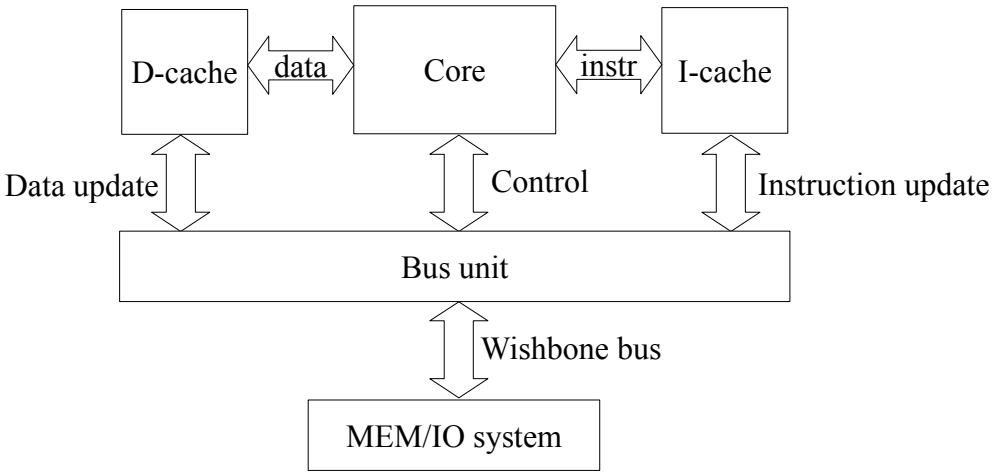
1.2 VHDL File Hierarchy

All needed files are located in the *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 data from/to any MEM/IO location. The number of cache pages as well as the page size and it's coherency strategies can be configured for each cache independently. Together with a bus unit, which connects the the cache memories via a pipelined Wishbone interface to the rest of the system, these four blocks (core, i-cache, d-cache, bus unit) form the STORM_TOP unit.
By default, the processor operates in Big Endian mode. To change to Little Endian mode, set the `USE_BIG_ENDIAN` constant in the core package (*CORE_PKG.vhd*) to `FALSE`.



1.4 STORM_TOP Interface

Generic constant	Generic type	Function
I_CACHE_PAGES	natural	Number of pages in I-Cache
I_CACHE_PAGE_SIZE	natural	I-Cache page size (# of 32-bit words)
D_CACHE_PAGES	natural	Number of pages in D-Cache
D_CACHE_PAGE_SIZE	natural	D-Cache page size (# of 32-bit words)
TIME_OUT_VAL	natural	Maximum Wishbone bus cycle length
BOOT_VECTOR	std_logic_vector(31:0)	Boot vector address
IO_UC_BEGIN	std_logic_vector(31:0)	First address of not cache-able IO area
IO_UC_END	std_logic_vector(31:0)	Last address of not cache-able IO are

Port signal	Signal size	Direction	Function
CORE_CLK_I	1 bit	Input	Core clock signal, triggering on rising edge
RST_I	1 bit	Input	System rest, high-active, sync to rising edge of core clock
IO_PORT_O	16 bit	Output	Direct system output port
IO_PORT_I	16 bit	Input	Direct system input port
WB_ADR_O	32 bit	Output	Wishbone bus address, word-boundary → bits[1..0] = “00”
WB_CTI_O	3 bit	Output	Wishbone bus cycle type
WB_TGD_O	7 bit	Output	Wishbone bus cycle tag
WB_SEL_O	4 bit	Output	Wishbone bus byte select, always set to “1111”
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_ERR_I	1 bit	Input	Wishbone bus abnormal cycle termination
WB_HALT_I	1 bit	Input	Wishbone bus halt
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.

Information about the configuration generics

- Each cache configuration value (number of pages, page size) must be a power of two.
- Any load/store operation from or to an address, which is not cached, will result in an upload or download of the corresponding data page from or to the memory/IO system. Especially for IO devices this might result in inconsistencies and/or bad timing. To avoid this, a specific IO address area can be configured by using the `IO_UC_BEGIN` and `IO_UC_END` generics. When the `CACHED_IO` bit in the system control register 0 is set to zero, any access to a device within this address range will bypass the D-cache and directly access the selected IO device (in single word = 32-bit entry access mode).

! Devices, which are included within the IO area definition (see above) are automatically protected – any access in unprivileged mode will trigger an interrupt and abort the bus transaction. This functionality can be disabled by clearing the `PRTC_IO` bit int the system control register 0.

Information about the interface signals

- All interface signals are `STD_LOGIC` or `STD_LOGIC_VECTOR`.
- Since all components of the STORM Core use a synchronous reset, the RST_I must be kept high for at least one cycle of the core clock to ensure valid reset functionality.
- The `IO_PORT_O` and `IO_PORT_I` signals are processor internal IO ports, controllable via coprocessor registers. They can be used to directly control system functions without implementing IO controller within the Wishbone network.
- The `WB_TGD_O` signal gives information about the current processor mode and the bus access type. Bits 4 downto 0: Current processor mode. Bit 5: '1' instruction transfer, '0' data transfer. Bit 6: '1' dedicated IO access, '0' standard memory access.
- When the `WB_ERR_I` signals is set to '1' during a MEM/IO access, the instruction (when loading the i-cache) or data abort (when loading/flushing the d-cache or when accessing IO) trap is taken.

! The bus unit only supports pipelined Wishbone cycles. Standard (unpipelined) Wishbone slaves must use the `WB_HALT_I` signal to throttle the bus transactions.
Example from the Wishboned specification data sheet:

```
DEVICE_HALT_O <= DEVICE_STB_I and (not DEVICE_ACK_O);
```

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 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 (BX) is implemented, since the processor does not support any short instruction format.
- 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

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"
System, SYS	0x00000000	"11111"
Undefined Instruction, UND	0x00000004	"11011"
Supervisor, SVP	0x00000008	"10011"
(Instruction) Abort, ABT (IAB)	0x0000000C	"10111"
(Data) Abort, ABT (DAB)	0x00000010	"10111"
reserved	0x00000014	-
Interrupt Request, IRQ	0x00000018	"10010"
Fast Interrupt Request, FIQ	0x0000001C	"10001"

2.3 Registers

Each operation mode has a unique register set, including data registers (see table below) implying a link register (LR, always R14), the program counter (PC, always R15), the current machine status register (CMSR (CPSR in ARM)) and a saved machine status register (SMSR_<mode> (SPSR_<mode> in ARM)).

Mode	Accessible data registers	Accessible machine registers
USR	R0, …, R14	PC, CMSR
SYS	R0, …, R14	PC, CMSR, SMSR_SYS
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

Note: User mode (USR) and System mode (SYS) share the same data registers, but System mode has a unique saved machine status registers (SMSR_SYS). Also, System mode is a privileged mode.

Note: Writing 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 memory block addresses 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 executed / respectively an exception trap is taken (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 Core'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 terminated. 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 (and enabled) 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'.

2.5 Machine Status Register

CMSR bit #	Name	Default	Interrupt
0 … 4	SREG_MODE_x	11111	Mode register, SYS after reset
5	*reserved*	0	
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
28	SREG_O_FLAG	0	Overflow flag
29	SREG_C_FLAG	0	Carry flag
30	SREG_Z_FLAG	0	Zero flag
31	SREG_N_FLAG	0	Negative flag

*) 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 components are instantiated, the coding style allows the synthesizing tools to map some modules to dedicated hardware blocks (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/from the machine control registers and to/from the system coprocessor.
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 or from the cache memories 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 fully associative and can be mapped to dedicated memory blocks.
CORE.vhd	The CORE.vhd is the top entity of the STORM processing units.
CORE_PKG.vhd	This file is the main package file, where all necessary modules and parameters are defined.
FLOW_CTRL.vhd	The flow control generates the control signals for each stage and every module within 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 generate the address and the control signals for 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	This “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 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 processing core, data and instruction cache and a Wishbone compatible bus interface.
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 downloaded from the memory/IO system. This can take several cycles, depending on the cache's page size, the speed of the bus system and the speed of the accessed device (e.g. memory).

When the device access takes longer than a maximum value, that can be specified using the system coprocessor, the IAB interrupt is taken (only when the bus unit was fetching instructions (data for i-cache). When it was fetching data (data for the d-cache) the DAB interrupt is taken). Maximum value for max_cycle_length (maximum bus cycle length) = x"FFFF".

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. The page replacement strategy is "least used".

3.3.1 MEM / IO → Cache coherency

If there are other devices than the STORM Core, that can access the memory system, he user has to take care, that the cache has always the recent data from this memory system. For example by using an external interrupt (IRQ), other master devices can show that data within the memory system was changed.
IO devices, such as communication devices (UART, SPI, ...), should not be cached, because this might lead to incoherent data. Use the IO area definition generics (`IO_UC`) to define the address space, where IO devices are located. Nevertheless, IO devices can be cached (`DC_CIO` bit in sys_cp system control register) to support an increase in data transport speed for streaming devices.

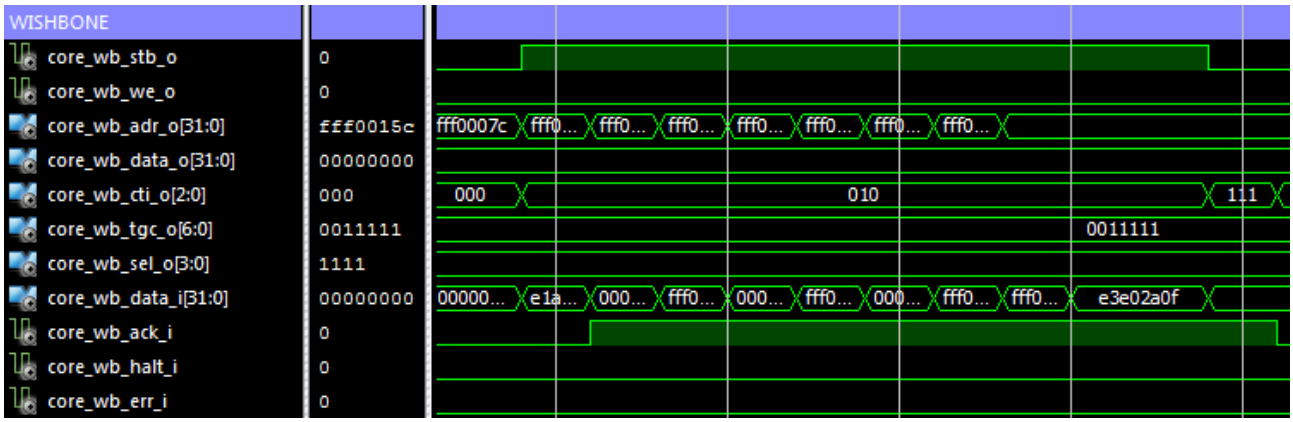
Note: For cache "read-through", set the `IC_AUTOPR/DC_AUTOPR` bit in the sys_cp system control register.

3.3.2 Cache → MEM / IO coherency

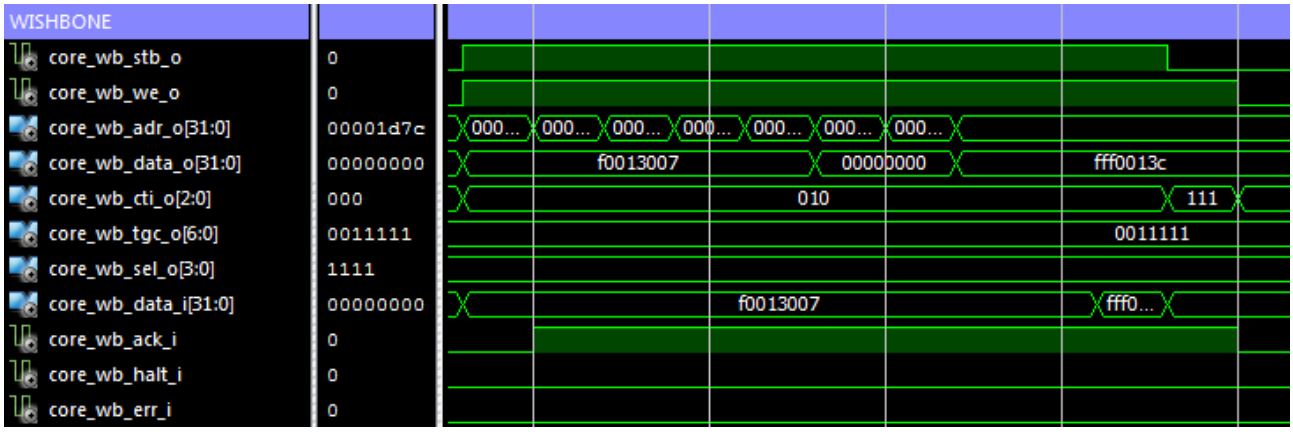
When using "Write-Thru" coherency strategy, any modification of a cache entry (of course only within the d-cache) leads to a write back of the complete corresponding cache page to the memory/IO system. This might cause problems for IO devices (e.g. UART), which trigger their operation on write-access bus cycles. To avoid this problem, IO devices should be mapped to a specific address area, which is defined by the `IO_UC` generics.
Disabling the "Write-Thru" strategy in the system control coprocessor introduces the standard coherency strategy, where a modified cache page is only written back to the memory/IO system when it is going to be replaced by the bus unit.

If there are unused address areas within the valid memory/IO address space, which can be cached, a time-out might occur during a page upload/download, because the bus unit is waiting for an acknowledge from addresses, which are not used. Ensuring a whole-less memory map is crucial. Insert simple dummy registers to close those address gaps within the cache-able address area.

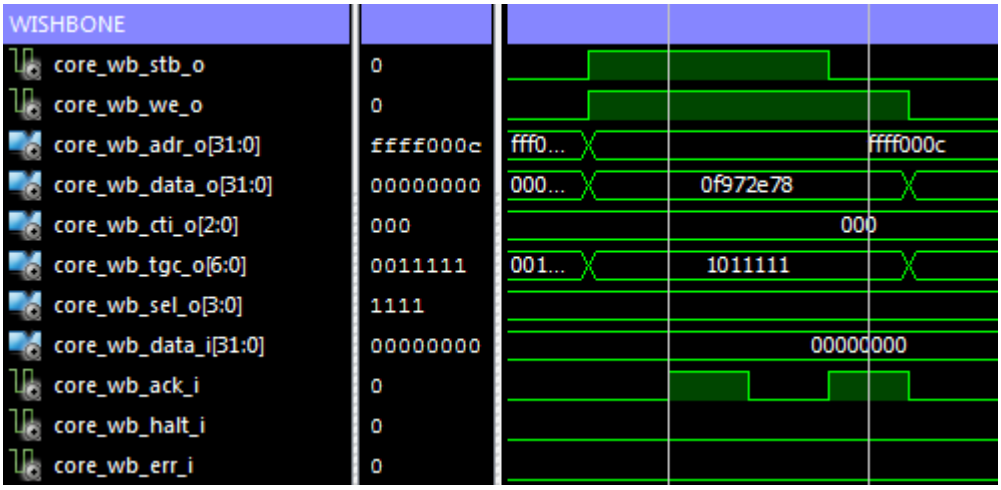
3.4 Example Bus Cycles



D-cache page download, burst transfer (page size = 8)



D-cache page upload, burst transfer (page size = 8)



dedicated IO access

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

3.5.1 Local Pipeline Conflicts

Local pipeline conflicts occur, when data, that is needed for further processing, has not yet reached the end of the pipeline (register file), so it 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 result 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 actual data from R1.

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 within *operand_unit.vhd* file.

3.5.2 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.

3.5.3 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.
(Branch “prediction” is 'always taken'.)

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 fetch the next valid instruction until the OF stage.

3.5.4 Memory-based Branches

Memory-based branches are a special form of (manual) branches. Here, the new value for the program counter does not come from a data register, but from the external memory/IO system.

Example program:

```
LDR PC, [R0]      (PC = MEM[R0])
ADD R3, R0, R1    (obsolete)
EOR R5, R0, R1    (obsolete)
SUB R2, R0, R1    (obsolete)
```

The program counter gets it's update after the *EX* stage. But since the update value comes from the memory system, it is not available until the **LDR** instruction reaches the *WB* stage. The problem is, that a single instruction (the **LDR** in this case) needs data, which is generated at the end of the pipeline, already at an early pipeline stage. This conflict cannot be solved by forwarding or stalling alone: Only the **LDR** is allowed to process within the pipeline, the instruction fetch is halted. So two empty instructions between the **LDR** and the next valid instruction are created. When the **LDR** reaches the *WB* stage, the data is passed via a special bus towards the PC, which can be updated without any conflicts since the instruction in the *EX* stage (and the *MA* stage) is invalid (→ empty instruction = “bubble”).

3.6 Stage Control Bus

The main control bus (CTRL) is generated by 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 recomputed during the pipeline flow.
To keep the design flexible for future changes, all signals are carried throughout the end of the pipeline.

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

Bit #	Signal name	Function
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 data cache read
26	CTRL_MEM_RW	Data cache read ('0') / write ('1') access
27	CTRL_RD_USR	Read data from USER register bank
28	CTRL_WR_USR	Write data to USER register bank
29	CTRL_MREG_ACC	Access machine register file (MREG)
30	CTRL_MREG_M	Access CMSR ('0') / SMSR ('1')
31	CTRL_MREG_RW	MREG read ('0') / write ('1') access
32	CTRL_MREG_FA	Full access ('0') / flag access only ('1')
33	CTRL_CP_ACC	Access coprocessor
34	CTRL_CP_RW	Coprocessor read ('0') / write ('1') access
35	CTRL_CP_REG_0	Coprocessor source / destination register address
36	CTRL_CP_REG_1	
37	CTRL_CP_REG_2	
38	CTRL_CP_REG_3	
39	CTRL_SHIFT_M_0	Barrelshifter shift mode
40	CTRL_SHIFT_M_1	
41	CTRL_SHIFT_V_0	Barrelshifter shift value (immediate)
42	CTRL_SHIFT_V_1	
43	CTRL_SHIFT_V_2	
44	CTRL_SHIFT_V_3	
45	CTRL_SHIFT_V_4	

3.7 Forwarding Bus

A unique forwarding bus is generated by each data processing stage within the pipeline.
So there is one forwarding bus for the *MS* stage, one for the *EX* stage, one for the *MA* stage and one for the *WB* stage. They are used to detect pipeline conflicts. This is done by the operand unit.

Bit #	Signal name	Function
0 ... 31	FWD_DATA_LSB ... FWD_DATA_MSB	Operand data bus
32 ... 35	FWD_RD_LSB ... FWD_RB_MSB	Destination register address
36	FWD_WB	Data value will be written back to register file
37	FWD_MCR_MOD	Machine register file may get modified
38	FWD_FLAG_MOD	Status flags may get modified
39	FWD_MCR_R_ACC	Memory read access
40	FWD_MEM_R_ACC	Machine register file read access
41	FWD_MEM_PC_LD	PC load from memory (memory-based branch)

4. Internal Coprocessor

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 and internal peripherals. 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 will be taken. Also, a write access to the coprocessor, which is not done in privileged mode, triggers the undefined instruction trap.

Program example (flush d-cache):

```
MCR P15, 0, R3, C6, C6
ORR R3, R3, #1
MRC P15, 0, R3, C6, C6, 0
```

Note: The operation bit-fields (here set to 0) 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	Core ID
2	ID_REG_2	r	Core ID
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	ADR_FB	r	Address feedback from bus unit
10	*reserved*	r	*reserved*
11	LFSR_POLY	r/w	Internal LFSR: Polynomial register
12	LFSR_DATA	r/w	Internal LFSR: Data register
13	SYS_IO	r/w	Direct IO register
14	*reserved*	r	*reserved*
15	*reserved*	r	*reserved*

4.1 System Coprocessor Register Set

ID Register 0, 1, 2

This registers present basic information about the STORM Core Processor.

CP Reg	Register	Bits	r/w	default	Function
0	ID_REG_0	31 .. 16	r	2012	Core version update date, year
		15 .. 08	r	3	Core version update date, month
		07 .. 00	r	8	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 0 gives access to *advanced* system configuration options.

CP Reg	Bit(s)	Name	Def	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	DC_WTHRU	0	Enable write-through coherency strategy for D-cache
	4	DC_AUTOPR	0	Auto pre-reload accessed D-cache page (→ “read-through”)
	5	IC_AUTOPR	0	Auto pre-reload accessed I-cache page (→ “read-through”)
	6	CACHED_IO	0	Enable cached IO
	7	PRTC_IO	1	Devices within IO area can only be accessed in privil. modes
	8	DC_SYNC	0	D-Cache is sync when '1' (read-only)
	9	reserved	0	*reserved*
	10	reserved	0	*reserved*
	11	reserved	0	*reserved*
	12	reserved	0	*reserved*
	13	LFSR_EN	0	Enable internal LFSR
	14	LFSR_M	0	New data after core clock ('0') or after data reg read-access ('1')
	15	LFSR_D	0	LFSR shift direction ('0': right, '1': left)
	16..31	MBC	256	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	Bits	Function
8	31 .. 16	D-Cache hit statistics, Hex FFFF is maximum value → D-Cache hit rate is one
	15 .. 00	I-Cache hit statistics, Hex FFFF is maximum value → I-Cache hit rate is one

Bus Unit Address Feedback

Via this register the core can get access to the last used Wishbone address. This can be used to examine the reason for a data / instruction abort (DAB/IAB exception).

CP Reg	Function
9	Last accessed Wishbone address

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

System IO Port

Two 16 bit IO signals are provided by the STORM Core to directly control system functions without using an extra IO controller.

CP Reg	Bits	Function
13	`SYS_IO(31:16)`	Input port (read-only)
	`SYS_IO(15:00)`	Output port

5. Getting Started

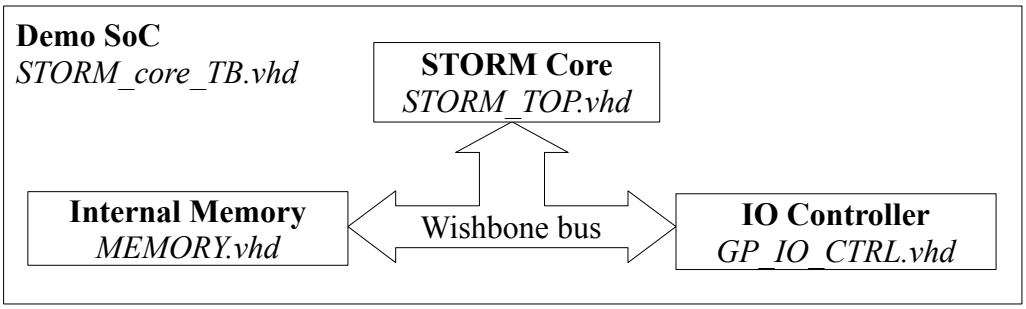
Start your evaluation tool (Xilinx ISE, Altera's Quartus II, Model Sim, etc) and create a new project, adding all files from the project's **rtl** directory. The file names of all needed files are listed in chapter 1.2.

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 interconnection fabric.

Note: For advanced simulation and debugging, you can enable a cache-memory-content signal, which allows an easy (and understandable) representation of the current content of each cache.
Open the **CACHE.vhd** file and uncomment the **GEN_DEBUG_MEM** section.
Do this only when you are simulating, otherwise the synthesis tool might not be able to fit the cache memory blocks into dedicated hardware memory components.

5.1 Demo SoC Setup

A basic setup of a simple SoC, including a compatible Wishbone fabric and bus system, a program/data memory and an IO controller, can be found within the **STORM_core_TB.vhd** file (sim folder). When using Xilinx ISIM, a basic waveform from the "sim/Xilinx ISIM" folder can be used to have a general overview of all important core signals (register, memories, IO, …).
The memory component already contains a simple demo program (→ "software/C/main.c") for testing the Demo SoC. It calculates the first 30 Fibonacci numbers and shows them on the IO controller's output port.



Memory Map			
Address	**Type**	**R/W**	**Device**
x"00000000" ... x"000003FF"	MEM	r/w	1024 bytes of internal program / data memory, preloaded with demo program
x"FFFFE020"	IO	r/w	Parallel output port
x"FFFFE024"	IO	r/w	Parallel input port

5.2 Software Setup Using Assembler (arm-elf)

```
→ arm-elf-as.exe : The arm-elf assembler
→ extract.exe    : The program 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.

To process, execute: `make`

Executing the “make” batch file will assemble the main.asm and all included project files. It generates the *a.out* opcode file, from which the program 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. This file can be used for programming via bootloader.

5.3 Software Setup Using C (WinARM)

```
→ build/STORMcore-RAM.ld : Linker script file
→ storm_extractor.exe    : The program extractor
→ main.c                 : Main program file
→ makefile               : Processing makefile
→ storm_core.h           : STORM Core 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 extractor from the ASM project folder, the storm_extractor from the C project folder will ouput a *storm_program.txt* for direct VHDL memory initialization and a *storm_program.dat* for e.g. bootloader transfer. Use the make file for processing. If you do not use the make file, you have to ensure, the compiler only creates 32-bit ARM opcodes.

To process, execute: `make clean all`

The main.c file contains a simple demo program, which calculates the first 30 Fibonacci numbers. It is intended to be used within the Demo SoC project / testbench.