Atlas Processor Core

by Stephan Nolting

Proprietary Notice

Disclaimer

The **ATLAS Processor Core** was created by Stephan Nolting.
For any kind of feedback, feel free to contact me: stnolting@gmail.com

The most recent version of the ATLAS Processor Core and it's documentary can be found at
http://www.opencores.org/project,atlas_core

Table of Content

1. Introduction

Welcome to the **ATLAS Processor** project!

In contrast to the STORM CORE Processor, the Atlas processor was completely designed on the paper before I wrote the first line of VHDL. Of course, several good ideas evolved during the coding process, but however, the Atlas processor was the first project, where I really intended to start from scratch and create a small and powerful processor core rather than doing a big research project (like the STORM CORE was).

I've come a long way and due to my work with famous processors like ARM (I really love ARM!), DLX and AVR – and of course also with the STORM CORE –, I gathered a lot of ideas what a cool processor architecture might look like. So I combined some of the features from these architectures together with a lot of coffee to create a CPU, that really measures up to all my – and hopefully someone's else – expectations.

I hope you can also feel the beauty of this architecture (yeah, I'm really proud of this processor – even if this sounds strange in a nerdy way...) when working with the CPU ;).

So, have fun with the Atlas processor!

1.1. Core Features

- ✔ 16-bit RISC open source soft-core processor
- ✔ Small outline CPU-only and complex 32-bit addressing main processor implementations available
- ✔ Completely described in behavioral **VHDL**
- ✔ Pipelined instruction execution in 5 stages
- ✔ Single cycle execution of all instructions (except for branches, multi-cycle operations and TDDs[1])
- ✔ Four forwarding units to accelerate internal operand fetch
- ✔ Powerful memory access and bit manipulation instructions
- ✔ Two different operating modes with unique register sets (8 registers each) and privileges
- ✔ Full hardware support for emulating privileged-mode programs (like operating systems) in unprivileged-mode
- ✔ Support for tagged system call operations (software interrupts)
- ✔ Two external interrupt request signals
- ✔ Configurable internal cache* (shared for instructions and data; fully associative)
- ✔ Configurable direct accessed address area to bypass cache (used for shared memories, IO devices, ...)
- ✔ Interface for two external coprocessors to extend the processor's functionality and instruction set
- ✔ Wishbone-compatible pipelined bus interface* supporting burst-transfers
- ✔ Implementation synthesis results (speed optimized) on a Xilinx Spartan-3 XC3S400A FPGA
  - ➔ CPU only: 80Mhz operating frequency at 12% device utilization (~470 slices)
  - ➔ Complete processor (CPU, cache[2], Wishbone bus interface, address extension coprocessor)*: 70Mhz operating frequency at 28% device utilization (~1040 slices)

*) Only available when using the complete Atlas processor (not the CPU-only implementation)

1 TDDs = Temporal data dependencies (processing data, that has not been fetched from the memory yet)
2 Default configuration: 4 cache pages, 64 byte page size, non-cached area starting at $FF000000

1.2. Project Overview

The Atlas project was created to be a general purpose processor for applications, which require minimal hardware resources while providing a maximum of functionality and processing power. Due to it's minimal outlines, it is eminently suited for control applications of any kind

Two different implementation schemes of the CPU are intended: For small applications, the CPU core can be used alone in combination with shared or distributed program and data memories. For this case, any user hardware modules should be connected via the coprocessor interface, to maximize the data transfer bandwidth. Thus, a small microcontroller-like system, that perfectly meets the requirements of the application, can be created. This system setup will be called the Atlas CPU-only or Atlas Micro. Of course, it is also possible to use the CPU alone, connected to your own kind of memory system.

The other implementation scheme, the so-called Atlas Processor, also implements the CPU core itself together with a fully associative shared data and instruction cache, a memory management unit to extend the accessible memory/IO area and a Wishbone-compatible bus interface. All additional hardware modules (user IP cores) can be connected via this bus interface. This setup is very well suited to use the Atlas processor as main processor for larger and more complex system on chips designs, including multi-core structures and large-scale processing arrays.

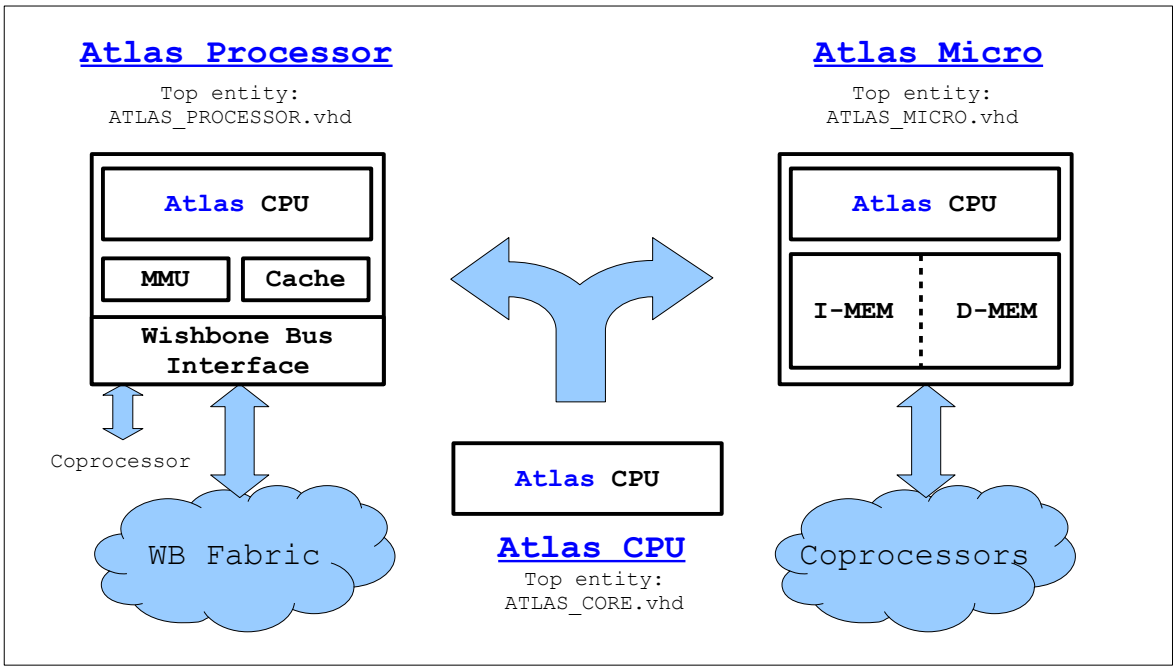

Figure 1: Implementation schemes

You can also use a mixture of both implementation schemes. For example, if the coprocessor interface is sufficient to couple all user IP modules, but if a larger accessible memory is required, a setup only incorporating the CPU core and and the MMU (but without the cache and bus interface) would be feasible.

1.3. Project Folder Structure

The actual project folder contains several folders, which are about to be explained.

- **asm:** This folder contains the Atlas assembler program, the C source file as well as an example folder, providing several example assembler programs.
- **doc:** A copy of the Wishbone bus specifications and the complete Atlas data sheet can be found here.
- **rtl:** All necessary VHDL rtl files for the CPU core and both implementation schemes are located in this folder.
- **sim:** The sim folder contains testbenches for the Atlas Processor and the Atlas Micro together with additional simulation components (like demo memory) and default Xilinx ISIM (c) waveform configurations.
- **syn:** This folder can be used by the synthesis tool for the actual project implementation.

1.4. VHDL File Hierarchy

All necessary hardware description files are located in the project's *rtl* folder. The top entity of the CPU is ATLAS_CORE.vhd. The top entity of the complete processor, including a cache, a memory management unit and the Wishbone bus interface, is ATLAS_PROCESSOR.vhd. The top entity of the micro implementation including the CPU and data/instruction memories is ATLAS_MICRO.vhd.

ATLAS_PROCESSOR.vhd	→ **Atlas Processor** top entity
- BUS_INTERFACE.vhd	→ Cache and Wishbone bus interface
- MMU.vhd	→ Virtual address extension controller
ATLAS_MICRO.vhd	→ **Atlas Micro** top entity, including CPU + D/I memory
- ATLAS_CORE.vhd	→ CPU core top entity
- ATLAS_PKG.vhd	→ Atlas project package file
- ALU.vhd	→ Arithmetical/logical unit, CP interface
- CTRL.vhd	→ CPU control system
- MEM_ACC.vhd	→ Data memory access system
- OP_DEC.vhd	→ Opcode decoder
- REG_FILE.vhd	→ Data register file
- SYS_REG.vhd	→ Machine control register (PC and MSR)
- WB_UNIT.vhd	→ Data write-back unit

Table 1: Project's VHDL file hierarchy

2. Top Entities Signal Description

These chapters give a brief overview of the signal ports of the CPU top entity (ATLAS_CORE.vhd) and the complete processor top entity (ATLAS_PROCESSOR.vhd), which also includes the cache, the bus interface and arbitration control logic. The type of all signals/generics is **std_logic** or **std_logic_vector**, respectively.

2.1. Atlas CPU Interface

The following table presents the interface ports of the Atlas CPU top entity (**ATLAS_CPU.vhd**).

Signal name	Width (#bits)	Dir	Function
BOOT_ADDRESS_G	16	-	Generic for configuring the CPU's boot address
CLK_I	1	IN	Global clock line, all registers trigger on the rising edge, 50% duty cycle
RST_I	1	IN	Global reset signal, synchronized to CLK_I and high-active
HOLD_I	1	IN	Global halt signal, high-active, stops the processor, only change on falling edge of CLK_I
INSTR_ADR_O	16	OUT	New instruction address (= PC)
INSTR_DAT_I	16	IN	New instruction input, must be synchronized to CLK_I
INSTR_EN_O	1	OUT	Instruction update enabled when '1', INSTR_DAT_I must not change when this signal is set to '0'
SYS_MODE_O	1	OUT	Current processor operating mode (0: user mode, 1: system mode)
SYS_INT_O	1	OUT	Asserted when an interrupt has been triggered (internal or external)
MEM_REQ_O	1	OUT	Set to '1' when a data memory access is requested in the next cycle
MEM_RW_O	1	OUT	Memory read ('0') or write ('1') access
MEM_ADR_O	16	OUT	Memory access address
MEM_DAT_I	16	IN	Memory write data
MEM_DAT_O	16	OUT	Memory read data, must be synchronized to CLK_I
CP_CP0_EN_O	1	OUT	Coprocessor 0 select
CP_CP1_EN_O	1	OUT	Coprocessor 1 select
CP_OP_O	1	OUT	Coprocessor processing operation ('0') or data transfer ('1')
CP_RW_O	1	OUT	Coprocessor read ('0') or write ('1') data transfer access
CP_CMD_O	9	OUT	Coprocessor command, consisting of source/destination register and operation command
CP_DAT_O	16	OUT	Coprocessor read data input for both coprocessors (OR-ed)
CP_DAT_I	16	IN	Coprocessor write data
EXT_INT_0_I	1	IN	External interrupt line 0
EXT_INT_1_I	1	IN	External interrupt line 1

Table 2: Processor's CPU top entity interface ports

2.2. Atlas Micro Interface

The following table presents the interface ports of the Atlas Micro top entity (**ATLAS_MICRO.vhd**).

Signal name	Width (#bits)	Dir	Function
Configuration Generics			
MEM_SIZE_G	natural	-	Instruction/data memory size in bytes
SHARED_MEM_G	boolean	-	Shared (=true) or separated (=false) I/D memories
BOOT_ADDRESS_G	16	-	Generic for configuring the CPU's boot address
Global Control			
CLK_I	1	IN	Global clock line, all registers trigger on the rising edge, 50% duty cycle
RST_I	1	IN	Global reset signal, synchronized to CLK_I and high-active
SYS_MODE_O	1	OUT	Current CPU operating mode (0 = user, 1 = system)
Coprocessor Interface			
USR_CP_EN_O	1	OUT	Coprocessor 0 select (user coprocessor)
SYS_CP_EN_O	1	OUT	Coprocessor 1 select (system coprocessor)
CP_OP_O	1	OUT	Coprocessor processing operation ('0') or data transfer ('1')
CP_RW_O	1	OUT	Coprocessor read ('0') or write ('1') data transfer access
CP_CMD_O	9	OUT	Coprocessor command, consisting of source/destination register and operation command
CP_DAT_O	16	OUT	Coprocessor read data input for both coprocessors (OR-ed)
CP_DAT_I	16	IN	Coprocessor write data
Interrupt Line			
IRQ0_I	1	IN	External interrupt line 0
IRQ1_I	1	IN	External interrupt line 1

Table 3: Atlas Micro top entity interface ports

2.3. Atlas Processor Interface

The following table presents the signal specifications of the Atlas processor's top entity interface ports (**ATLAS_PROCESSOR.vhd**). The provided bus interface is compatible to the Wishbone B4 specifications[3].

Signal name	Width	Direction	Function
Configuration Generics			
UC_AREA_BEGIN_G	32	-	First address of not-cached memory/IO area
UC_AREA_END_G	32	-	Last address of not-cached memory/IO area
BOOT_ADDRESS_G	32	-	Processor's boot address
Global Control			
CLK_I	1	IN	Global clock line, all registers trigger on the rising edge, 50% duty cycle
RST_I	1	IN	Global reset signal, synchronized to CLK_I, high-active
User Coprocessor Interface			
CP_EN_O	1	OUT	Access to external (user) coprocessor
CP_OP_O	1	OUT	Data transfer/ processing operation
CP_RW_O	1	OUT	Read/write access
CP_CMD_O	9	OUT	Processing opcode and register addresses
CP_DAT_O	16	OUT	Write data output
CP_DAT_I	16	IN	Read data input
Interrupt Line			
IRQ_I	1	IN	External interrupt request line
Wishbone Bus Interface			
WB_ADR_O	32	OUT	Bus access address
WB_CTI_O	3	OUT	Cycle type identifier
WB_SEL_O	2	OUT	Byte select (always “11” → full 16-bit word data quantity)
WB_TGC_O	1	OUT	Cycle tag indicating the current operating mode
WB_DATA_O	16	OUT	Write data output
WB_DATA_I	16	IN	Read data input
WB_WE_O	1	OUT	Read/write bus access
WB_CYC_O	1	OUT	Valid cycle identifier
WB_STB_O	1	OUT	Data strobe
WB_ACK_I	1	IN	Acknowledge input
WB_HALT_I	1	IN	Hold bus access

Table 4: Processor system's top entity interface ports

3 A copy of the Wishbone specifications can be found in the *core/doc* folder.

3. Programmer's Model

The Atlas processor is a true 16-bit RISC architecture, providing different data register banks and privileges for the two operating modes. The accessible CPU resources according to the operating modes are shown in the figure below.

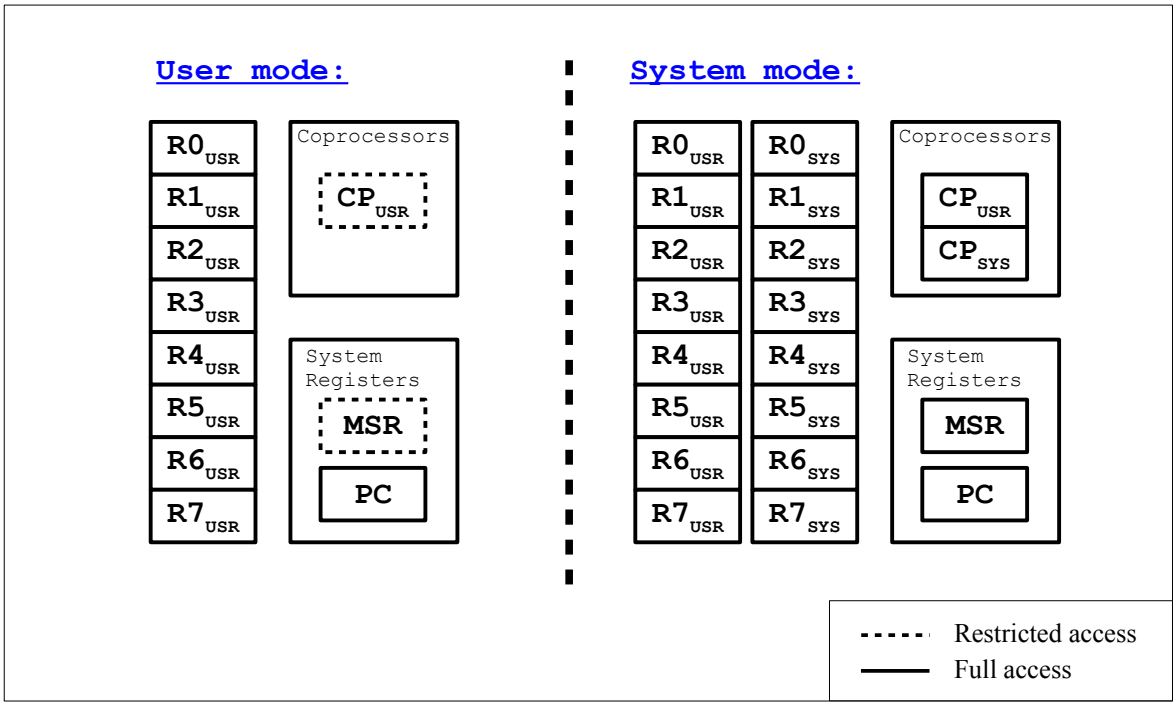

Figure 2: Operation modes and accessible registers

3.1. Operating Modes

Two different operation modes are supported by the Atlas processor. The privileged mode is called “system mode”, where the unprivileged one is called “user mode”. After a hardware reset, the core always starts execution in system mode with full privileges. After program setup, the current processor mode can be switched to user mode to start an application, which requires limited privileges to keep the system's security. The program running in user mode can use system calls to request privileged operations, like direct hardware access. Furthermore, the user program can be interrupted by external interrupts at any time. In this case, the processor automatically switches back to system mode and resumes operation executing the corresponding interrupt handler. Due to hardware features, the context switches from user mode to system mode and back from system mode to user mode do not need any additional software handling.

Note: All instructions and operations, that are allowed in system mode, but are not allowed in user mode (like user bank transfers, accesses to a protected coprocessors or full MSR accesses) will trigger the software interrupt trap (system call alias). These hardware features allow to emulate a system mode program, like an operating system, in user mode. This is very suitable for the implementation of virtual machines, which are able to run complete operating system.

3.2. Exceptions and Interrupts

The Atlas CPU features four different interrupt or exception types. In famous books about computer architecture, “exceptions” refer to all kind of abnormal program interruptions, no matter what source they emerge from. “Interrupts” are a sub group of those exceptions, where the cause emerges from an external signal, like an interrupt request pin. However, in this documentary and in the hardware description files of the CPU, all kinds of abnormal program interruptions are called interrupts. The different types, their priority during execution, their option to be masked and the corresponding addresses of the interrupt handlers are listed in the table below.

Priority	Interrupt source	Mask-able	Handler base address
1 (highest)	Hardware reset	No	x”0000”
2	External interrupt signal 0 (EXT_INT_0_I)	Yes	x”0001”
3	External interrupt signal 1 (EXT_INT_1_I)	Yes	x”0002”
4 (lowest)	Software interrupt (SYSCALL instruction, access violations, undefined instructions)	No	x”0003”

Table 5: Interrupt vector address and priority list

Whenever a valid interrupt condition occurs, the processor stops execution, enters system mode and resumes operation at the corresponding interrupt handler base address. These base addresses are fixed in hardware and only one word separates the different interrupt vectors. Thus, a branch instruction to the final handler, or a branch to an intermediate handler, which loads the address of the final handler) must be inserted into the interrupt vector slots. Furthermore, the return address is automatically stored to the link register.

Note: The execution of all instructions is atomic. Thus, the execution of a single instruction cannot be interrupted by any kind of exception/interrupt.

3.3. Data Registers

Each operating mode has direct access to a mode-depended set of eight 16-bit registers. When changing modes (context switch), no storing of the registers on the stack is necessary, since the hardware changes the accessible register bank corresponding to the new operation mode automatically. When in privileged system mode, all of the 16 register can be accessed, but only 8 of them – the actual system mode registers – can be used for data processing or transfer operations. The remaining 8 user mode registers must be accessed via special instructions and their data has to be moved to a system mode register before performing any data manipulation.

3.4. Coprocessors

Up to two external coprocessors can be attached to the Atlas CPU to extend the functionality and the instruction set of the processor core. By default, coprocessor 1 is already included within the processor system and represents a bus access controller, that is capable of extending the accessible memory space to 4GB. The coprocessor 0 slot can be used by the system designer to attach custom logic to the Atlas processor. This coprocessor slot is disabled by default, so any access will be unsuccessful and triggers the software interrupt. The slot can be enabled via special configuration constants in the CPU's VHDL package file. Both coprocessors can be accessed by special coprocessor instructions. These instructions are separated into two classes: The first classes is used for transferring data from a CPU register to a coprocessor and the other way around. The other class only effects the coprocessor and it's registers and is meant to perform data processing operations directly on the processors. Coprocessor 1 is the “system coprocessor” and thus can only be accessed in system mode. Coprocessor slot 0 can also be accessed in user mode, but if necessary, the access can be restricted to user mode by setting the protection flag in the machine status register. Any attempt to access a protected coprocessor in user mode will trigger the software interrupt trap.

3.5. Machine Status Register

The machine status register, abbreviated as MSR, holds the global control flags as well as the the CPU's ALU flags. The different flags and flag sets of the MSR are shown in the figure below.

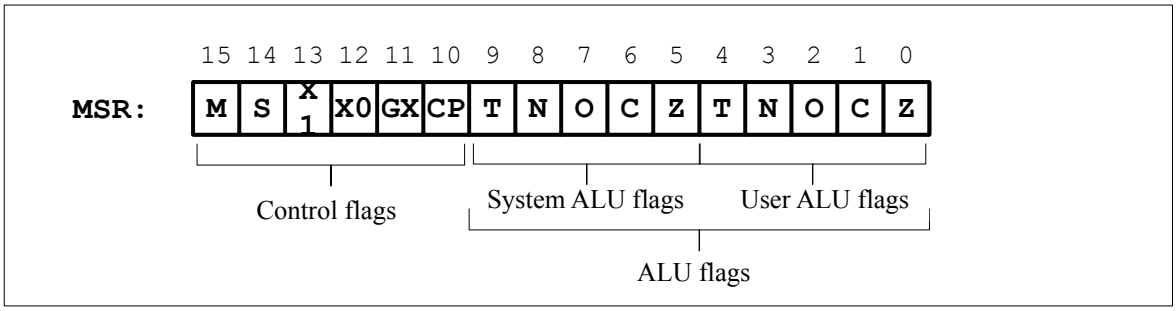

Figure 3: Machine Status Register

The flags, which are used by the arithmetical/logical unit and the condition computing unit, are located in the lowest 10 bit of the machine status register. There are two identical sets of the ALU processing flags. Together they are called “ALU flags” One set is used when in system mode (“system ALU flags”), the other is used by programs in user mode (“user ALU flags”). Each set holds information about the result of the previous data processing operations. These flags can be automatically updated after a data processing operations when using a specific suffix for the corresponding mnemonics. Otherwise, the flags are not altered. The name, location and functionality of the ALU flags is presented in the table below.

Flag name	Bit # in user mode	Bit # in system mode	Function
Z	0	5	Zero flag
C	1	6	Carry flag
O	2	7	Overflow flag
N	3	8	Negative flag (sign)
T	4	9	Transfer flag

Table 6: ALU flags for user / system mode

The zero flag (**Z**-flag) is always set whenever the operation result is zero. The most significant bit of the operation result (= the sign, when using two's complement representation) is copied to the negative flag (**N**-flag). The carry flag (**C**-flag) indicates a carry for an addition and subtraction or a direct data output of the shifter. The overflow flag (**O**-flag) is set whenever a range overflow during a two's complement arithmetical operation takes places. During a shift operation an overflow can occur when the sign bit of Ra gets changed. Logical operations do no alter the overflow or the carry flag. The transfer flag (**T**-flag) is not altered by any data processing operations and is used for bit test and transfer operations. All together, the ALU flag set of the current processor operation mode determines the condition for conditional branches.

The system control flags, located in the highest 6 bits of the MSR, are used to configure general CPU functions. The different flags, their location and their functionality are shown in the table below.

Bit #	Flag name	Function	When set to '0'	When set to '1'
10	**CP**	User coprocessor (coprocessor 0) protection	Coprocessor 0 can be accessed in user and system mode	Coprocessor can only be accessed in system mode
11	**GX**	Global external interrupt enable	Disable all external interrupts	Enable external interrupts
12	**X0**	External interrupt line 0 mask	Disable external interrupt line 0	Enable external interrupt line 0
13	**X1**	External interrupt line 1 mask	Disable external interrupt line 1	Enable external interrupt line 1
14	**S**	Previous operating mode	Was in user mode	Was in system mode
15	**M**	Operating mode	Processor is in user mode	Processor is in system mode

Table 7: System control flags

Bit 10 (**CP**-flag) is used to protect the “user” coprocessor (coprocessor 0) from being accessed in user mode. An unauthorized access in user mode will trigger the software interrupt trap.
The following three bits 11 to 13 (**GX**-, **X0**-, **X1**-flag) configure the two external interrupt lines. A global interrupt is valid and executed when the global interrupt enable flag (**GX**-flag) and the corresponding interrupt line mask flag (**X0** for *EXT_INT_0*, **X1** for *EXT_INT_1*) are set to '1'. Whenever a valid external interrupt request occurs, the execution of the correlated handler is started. The global external interrupt enable flag is then automatically cleared and can be set to '1' again when returning from the interrupt handler routine.
Bit 14 (**S**-flag) indicated the previous operating mode, before a context change has been performed. For example, when executing a interrupt handler from a user mode program, the s-flag is zero. When executing the same handler from a system mode program, the flag is set. As the last bit of the MSR (bit 15), the **M**-flag determines the current operation mode of the CPU. A '1' indicates system mode and a '0' indicates user mode. This flag is automatically updated on context up- (user mode → system mode, exceptions/interrupts) and down-switches (system mode → user mode, e.g. return from exception/interrupt handler). However, it can also be manually set or cleared when operating in system mode.

The MSR can be accessed by special instructions to transfer the MSR content to a register or to store a register's content to the MSR. Also, a direct initialization of either the user mode or the system mode ALU flags with an immediate is possible. In system mode, the complete MSR, only the ALU flags or only the ALU flags of a specific operation mode can be altered. In user mode, only a read or write access to the user mode ALU flags is allowed. When trying to alter or to read other bits (determined by actual read/write option) of the MSR from a user mode program, the software interrupt trap is taken.

3.6. Memory Model

A uniform and linear address space of 2^{16} = 65536 bytes is assumed by the Atlas CPU. However, the memory data bus is 16-bit wide, thus a word of 16 bit is transferred from or to the memory at one time. If a memory system is not capable of presenting a full word at one time, the memory manger has to halt the processor until it has assembled a full 16 bit word.
Data memory accesses can be performed on word boundaries (aligned access) or on unaligned addresses by using any register as pointer. When accessing unaligned addresses, the bytes of the transfer data are swapped. This feature is illustrated in the figure below. Note, that in this example little Endian mode is used. The actual Endianness of the CPU can be modified in the CPU's VHDL package file (default is little Endian).

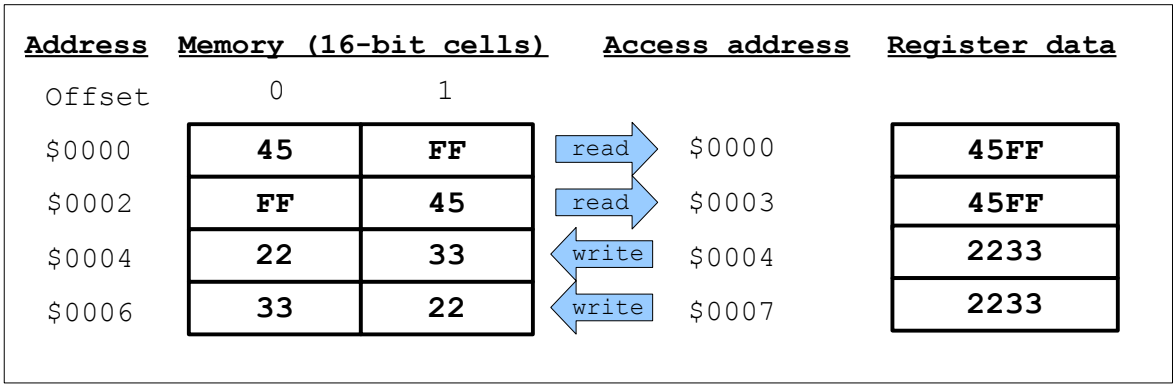

Figure 4: Memory accesses on aligned / unaligned word boundaries (hexadecimal data)

For the CPU-only implementation, the data memory can also be used as program memory implementing a Von-Neumann architecture to share data and instruction memory space. Of course, a Harvard-like architecture with separated memories for instruction and data is also possible. Instruction fetch accesses will always be performed on aligned addresses, therefore instruction opcodes must be placed at word boundaries.

3.6.1. Virtual Address Extension

To extent the accessible memory space, the system coprocessor (coprocessor 1, MMU) of the Atlas processor implementation presents the functionality to separate an address space of 32-bit (4 GB) into 2^{16} blocks of 2^{16} bytes each. The block address (= the most significant 16 bits of the address) is generated by base address registers within the MMU, separated for instruction/data access in user and system mode. It's the task of the system mode program to handle the management of this different memory pages. The chapter about the rtl architecture of the processor will focus on the actual configuration options of the system coprocessor.

3.7. Program Counter

Both operating modes use the same program counter (PC). It can be accessed via special load/store operations. For calling subroutines, register 7 (R7) of the current register bank is used as link register (LR) to store the return address. Furthermore, the link register is used to store the re-entry point (return address) whenever an interrupt or exception occurs. For exceptions (interrupts caused by the software; system calls, undefined instructions or access violations), the return address points to the second instruction after the one, that has caused the exception. For interrupts (external interrupts via the interrupt lines), the link register points to the second instruction after that one, that has completed last before the interrupt occurred. In both cases, the link register has to decremented by two (bytes) to restore the actual return address or re-entry point, respectively.

4. Instruction Set

This chapter introduces the encoding and functional explanation of the implemented instruction set. The complete set is divided into several classes and sub-sets, combining several instructions of one type. All instructions are 16-bit wide and must be placed at word-aligned memory addresses.

A short summary of the Atlas instruction set is shown in the figure below.

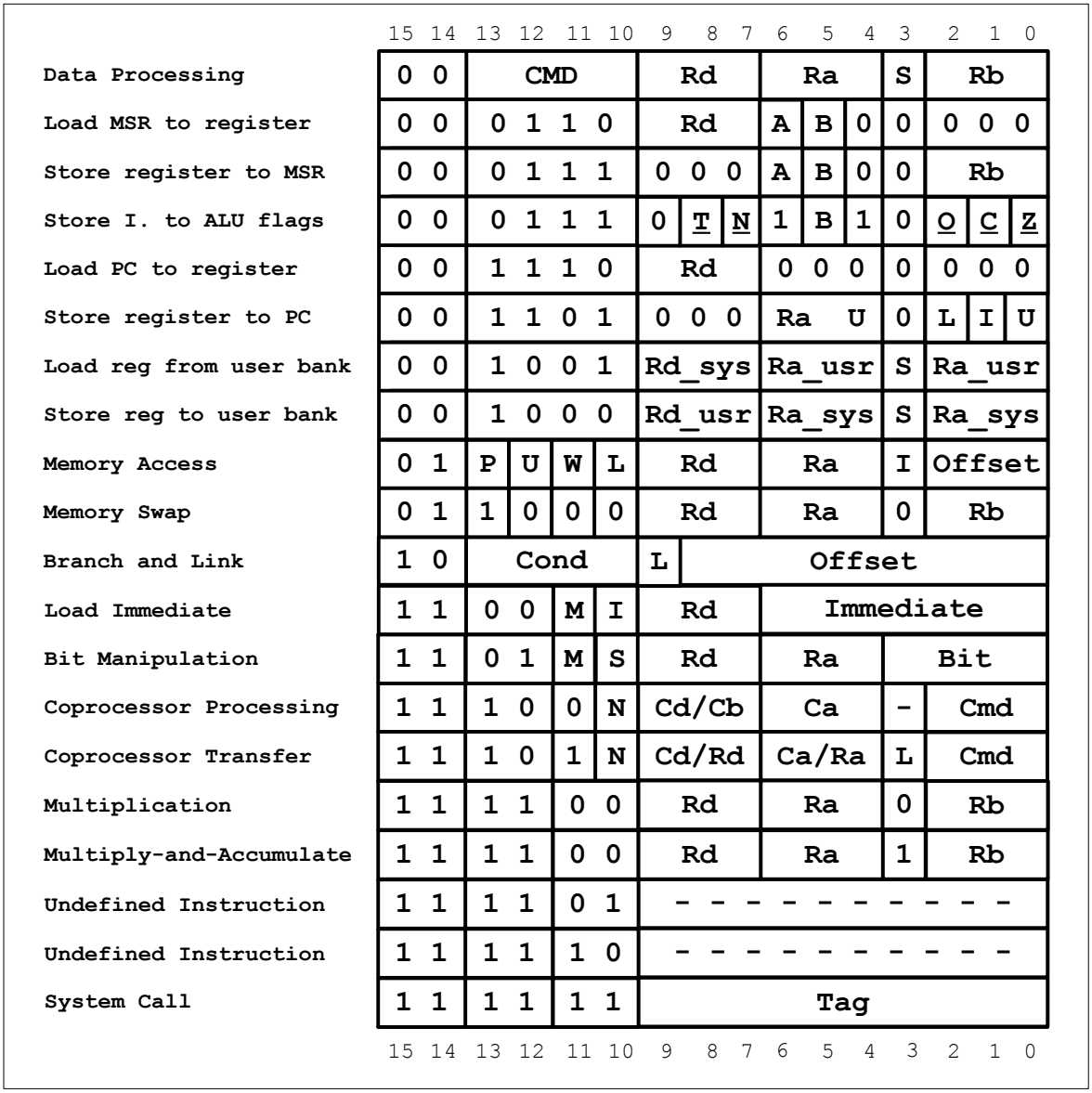

Figure 5: Instruction set formats

4.1. Data Processing

The instruction encoding of the data processing instructions is shown in the figure below.

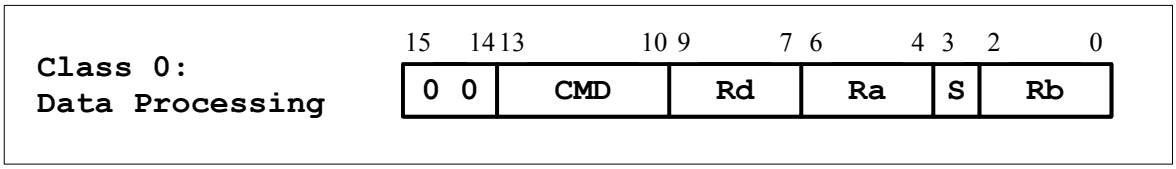

Figure 6: Data processing instructions format

This type of instructions performs an arithmetical or logical operation specified by the CMD bit-field on the two operand registers Ra and Rb and places the result in the destination register Rd (the binary operation codes for the CMD-field are specified in the table below). Some instructions only use register A (Ra) and manipulate it's content by an immediate coded with the three bits of the Rb bit-field. The instructions can be classified as logical (AND, NAND, ORR, EOR, BIC, TEQ, TST), arithmetical (ADD, ADC, SUB, SBC, IND, DEC, CMP, CPX) or shift (SFT) operations.

Whenever the S-bit is set by using an "S" as appendix to a data processing mnemonic, the carry, negative, zero and overflow flags (= the ALU flags corresponding to the current processor mode) are updated corresponding to the computation result. For test and compare instructions (TST, TEQ, CMP, CPX), the S-bit is always set, so the S-appendix is not required for the mnemonics. The assembler will automatically set the S-flag for this instructions. Furthermore, the Rd bit-field is not required for this type of instructions, since no computation data result is generated. Therefore, the Rd bit-field should be filled with zeros.

The extended compare instruction (CPX) can be used to compare larger words than 16-bit. Therefore, the CPX instruction subtracts operand A and operand B but takes also the carry and zero signal of the previous operation into account to compute the actual carry and zero flag result.

Most instructions combine the two operand registers to produce a result. The INC and DEC operations only use operand register A (Ra) and add or subtract a 3-bit immediate, which is encoded in the Rb bit-field. The shift (SFT) command uses this bit-field (Rb) to specify the type of shift operation, that is applied to Ra.

The assembler internal no-operation pseudo instruction (NOP) is formed from an increment on register 0 with a zero immediate and a cleared S-bit, resulting in no actual system state change. Thus, the binary coding of a NOP instruction is x"0000".

Mnemonic	CMD	Action
INC	0000	Rd = Ra + 3-bit-immediate; immediate is formed from the Rb-bits
DEC	0001	Rd = Ra – 3-bit-immediate; immediate is formed from the Rb-bits
ADD	0010	Rd = Ra + Rb
ADC	0011	Rd = Ra + Rb + Carry-Flag
SUB	0100	Rd = Ra - Rb
SBC	0101	Rd = Ra - Rb - Carry-Flag
CMP	0110	Flags = Ra - Rb; result is not written to a register
CPX	0111	Flags = Ra - Rb with old flags; result is not written to a register
AND	1000	Rd = Ra AND Rb
ORR	1001	Rd = Ra OR Rb
EOR	1010	Rd = Ra XOR Rb
NAND	1011	Rd = Ra NAND Rb
BIC	1100	Rd = Ra AND NOT Rb (bit clear)
TEQ	1101	Flags = Ra AND Rb; result is not written to a register
TST	1110	Flags = Ra XOR Rb; result is not written to a register
SFT	1111	Rd = shift(Rb); shift by one position; shift type is specified by Rb-bits

Table 8: Data processing commands

When using the `SFT` (shift) instruction, the Rb bit-field encodes the actual shift functionality by an immediate value. Data of Ra is always shifted by one place in the corresponding direction. The eight different shift types are listed in the table below.

Mnemonic	Rb[2:0]	Function	Data result	Carry result
#SWP	000	Swap bytes	Rd = Ra [7:0] & Ra[15:8]	Carry = Ra[15]
#ASR	001	Arithmetical right shift	Rd = Ra[15] & Ra[15:1]	Carry = Ra[0]
#ROL	010	Rotate left	Rd = Ra[14:0] & Ra[15]	Carry = Ra[15]
#ROR	011	Rotate right	Rd = Ra[0] & Ra[15:1]	Carry = Ra[0]
#LSL	100	Logical left shift	Rd = Ra[14:0] & '0'	Carry = Ra[15]
#LSR	101	Logical right shift	Rd = '0' & Ra[15:1]	Carry = Ra[0]
#RLC	110	Rotate left through carry	Rd = Ra[14:0] & Carry	Carry = Ra[15]
#RRC	111	Rotate right through carry	Rd = Carry & Ra[15:1]	Carry = Ra[0]

Table 9: Shift commands; note that '&' indicates a concatenation

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. INC, DEC (immediate operations)

```
<INC|DEC>{S} <Rd>, <Ra>, <#Imm>
```

2. CMP, CPX, TST, TEQ (compare/test operations, no result write-back to a register)

```
<CMP|CPX|TST|TEQ>{S} <Ra>, <Rb>
```

3. ADD, ADC, SUB, SBC, AND, ORR, NAND, EOR, BIC (arithmetical / logical operations)

```
<ADD|ADC|SUB|SBC|AND|ORR|NAND|EOR|BIC>{S} <Rd>, <Ra>, <Rb>
```

4. SFT (shift operations)

```
<SFT>{S} <Rd>, <Ra>, <#Shift>
```

{S}	Update processing flags corresponding to result when present.
<Rd>	Destination register.
<Ra>	Operand A register.
<Rb>	Operand B register.
<#Imm>	Three bit wide immediate (0...7); with present #-prefix.
<#Shift>	Shift type code, corresponding to the table above; with #-prefix.

Assembler Examples

```
INC  R0, R1, #2   ; increment R1 by 2 and store result to R0
INCS R0, R1, #2   ; increment R1 by 2, set flags and store result to R0
NOP               ; INC R0, R0, #0 = no operation
ADC  R2, R5, R2   ; add R5 and R2 with carry and store result to R2
ORRS R3, R3, R4   ; logical or of R3 and R4, set flags
                  ; and store result back to R3
SFT  R1, R3, #ROL ; rotate left R3 one position and store result to R1

CMP  R2, R0       ; compare low words first, then
CPX  R3, R4       ; compare high words to evaluate a 32-bit comparison
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
INC  R0, R1, #2   = 0b 00.0000.000.001.0.010 = x"0012"
INCS R0, R1, #2   = 0b 00.0000.000.001.1.010 = x"001A"
NOP               = 0b 00.0000.000.000.0.000 = x"0000"
ORRS R3, R3, R4   = 0b 00.1001.011.011.1.100 = x"25BC"
SFT  R1, R3, #ROL = 0b 00.1111.001.011.0.010 = x"3CB2"
```

4.1.1. User Register Bank Access

The instruction encoding of the user register bank access subset instructions is shown in the figure below.

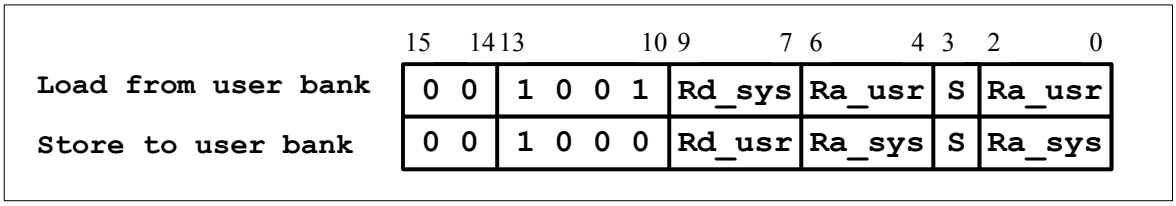

Figure 7: User register bank access instructions subset formats

Since there are no dedicated instructions to access the user register bank from a program in system mode, the access in encoded using a redundant form of the ORR and AND instructions. For Ra = Rb, these instruction are redundant, because the result is always Ra. Therefore the opcodes are reused to encode user bank transfers with the special mnemonics LDUB (load from user bank register) and STUB (store to user bank register).

The LDUB instruction uses the ORR binary format with Ra = Rb (= Ra_usr) to load the user bank register Ra_usr to system bank register Rd_sys. Whereas STSR uses the binary format of AND with Ra = Rb (= Ra_sys) to store the system bank register Ra_sys to the user bank register Rd_sys.

The transfer is only performed when executed in system mode. In user mode the load/store from/to user bank instructions will trigger the software interrupt.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. LDUB (load system bank register from user bank register)

```
<LDUB>{S} <Rd_sys>, <Ra_usr>
```

2. STUB (store system bank register to user bank register)

```
<STUB>{S} <Rd_usr>, <Ra_sys>
```

{S}	Update processing flags corresponding to result when present.
<Rd_sys>	System bank destination register.
<Ra_usr>	User bank source register.
<Rd_usr>	User bank destination register.
<Ra_sys>	System bank source register.

Assembler Examples

```
LDUB  R0, R4      ; load user bank register R4 to system bank register R0
STUB  R3, R2      ; store system bank register R2 to user bank register R3
STUBS R2, R6      ; store system bank register R6 to user bank register R2
                  ; and set flags corresponding to the data in R6
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
LDUB  R0, R4      = 0b 00.1001.000.100.0.100 = x"2444"
STUB  R3, R2      = 0b 00.1000.011.010.0.010 = x"21A2"
STUBS R2, R6      = 0b 00.1000.010.110.1.110 = x"2166"
```

4.1.2. Program Counter Access

The instruction encoding of the program counter access subset instructions is shown in the figure below.

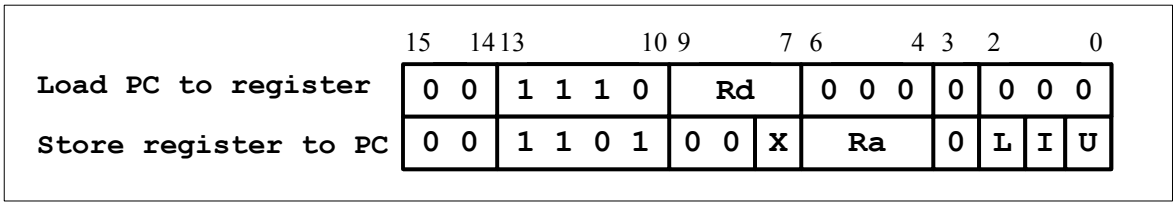

Figure 8: Program counter access instructions subset formats

Since there are no dedicated instructions to access the program counter (PC), the access is coded using the `TEQ` and `TST` instruction with a cleared S-bit. The mnemonics of these special instructions are `LDPC` (load from PC) and `STPC` (store to PC). Not all of the bit-fields are used for the transfer operations. Fill the unused bit-fields with zeros. `STPC` stores Ra to the program counter. This results in a branch to the address stored in Ra. Therefor, this instruction can be used to implement absolute branches. Since Rb is not used in this case, the bit-field of Rb encodes three additional options (X, L, I, U) for storing the new PC value. These options are active when the corresponding bit is set. The different options are presented in the table below.

Bit	Option	Name	Function, when bit is set ('1')
7	X	Mode exchange	Switch to mode, which is stored in MSR's S-flag, only allowed when in system mode!
2	L	Link	Save return address (PC + 2 bytes) to link register (LR = R7)
1	I	G_Interrupt_En	Set global external interrupt enable flag, only allowed when in system mode!
0	U	User Mode	Change operation mode to 'user mode', only allowed when in system mode!

Table 10: PC store options

If bit 0 (**U**) is set, the processor will resume operation in user mode at the address stored in Ra. This functionality can be used to return from a system mode program (e.g. interrupt handler) to restore operation in user mode. When bit 1 (**I**) is set, the global interrupt enable flag will be set. Therefore this option is useful to re-enable external interrupt after an external interrupt handler has finished. Both options will only have an effect when executed in system mode. Otherwise these options are ignored or irrelevant, respectively. Bit 2 (**L**) is set whenever the return address (PC + 2 bytes) shall be stored to the link register. This option is useful for implementing absolute calls to a subroutine. The option presented by bit 7 (**X**) will switch the current operating mode to the previous operating mode, when activated. This Features allows to restore the context after e.g. an interrupt handler, without knowing the actual mode to be restored. The actual mode is stored automatically by the CPU in the S-flag whenever a context change takes place. The X-option will copy the S-flag to the M-flag. Only the **L** option is allowed for programs in user mode. The **X**, **I** and **U** options will trigger the software interrupt trap when executed in user mode.

Note: There are three different mnemonics for the `STPC` (store register to program counter) instruction functionality. All of them perform the same operation and support the previously mentioned options. The three different aliases (`STPC`, `RET`, `GT`) are just used to make the actual intention of an instruction more clear (e.g. `RET` for a return from subroutine...).

The `LDPC` instruction will load the current program counter minus 4 bytes (this corresponds to the actual address of the executed `LDPC` instruction) to register Rd.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. LDPC (load PC to register)

```
<LDPC> <Rd>
```

2. STPC/RET/GT (Three different mnemonics for the same operation: Store register to PC)

```
<STPC|RET|GT>{X|U}{I}{L} <Ra>
```

{X|U}	Change to user mode when 'U' is present or restore saved operating mode ('X'), stored in the s-flag, when present. Only executed when in system mode.
{I}	Set global external interrupt flag when present (and executed in system mode).
{L}	Save return address (PC + 2 bytes) to link register when present.
<Rd>	Destination register.
<Ra>	Source register.

Assembler Examples

```
LDPC  R0    ; copy PC to R0

STPC  R7    ; store R7 to PC (absolute jump to [R7])
RET   R7    ; store R7 to PC (same operation, just another mnemonic)
GT    R7    ; store R7 to PC (same operation, just another mnemonic)

RETU  R7    ; store LR to PC and switch to user mode (e.g. return from
            ; software interrupt handler)
RETUI R7    ; store LR to PC, switch to user mode and set global external
            ; interrupt enable flag (e.g. return from ext. int. handler)

GTX   R2    ; store R2 to PC and restore previous operating mode
GTI   R2    ; store R2 to PC and set global external interrupt flag
GTL   R2    ; store R2 to PC and store return address to LR

GTUL  R3    ; store R3 to PC, change to user mode and store return address
            ; to LR
GTIL  R3    ; store R3 to PC, set global external interrupt flag and store
            ; return address to LR
GTXIL R3    ; store R3 to PC, restore previous operating mode, set
            ; global external interrupt flag and store return addr. to LR
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
LDPC  R0          = 0b 00.1110.000.000.0.000   = x"3800"
RETUI R7          = 0b 00.1101.000.111.0.0.1.1 = x"3473"
```

4.1.3. Machine Status Register Access

The instruction encoding of the machine status register access subset instructions is shown in the figure below.

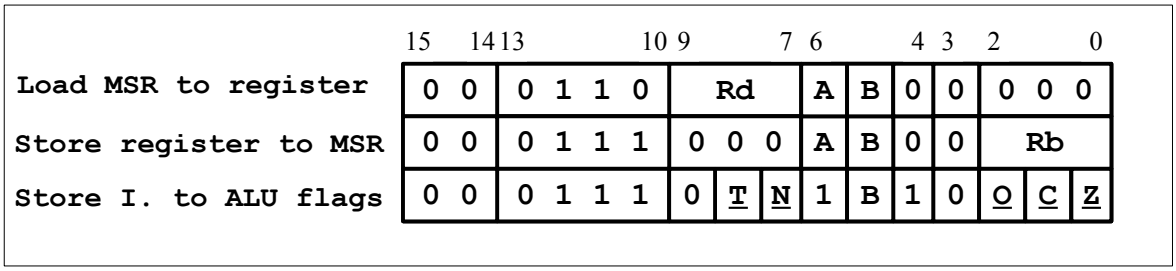

Figure 9: Machine status register access instructions subset formats

Since there are no dedicated instructions to access the machine status register (MSR), the access in encoded using the `CMP` and `CPX` instruction with a cleared S-bit. These mnemonics of these special instructions are `LDSR` (load register from MSR), `STSR` (store register to MSR) and `STAF` (store immediate to MSR's ALU flags). The `LDSR` instruction uses the `CMP` binary format with S='0' and will load the current MSR to Rd. Whereas `STSR` and `STAF` use the binary format of `CPX` with S='0' to store Rb or an immediate to the MSR. Not all of the bit-fields are used for the transfer operations. Fill the unused bit-fields with zeros.

Corresponding to the option bits (A, B), data can be written to the complete MSR, only to the ALU flags (user and system ALU flags), only to the system ALU flags or only to the user ALU flags. In user mode, only the user mode ALU flags can be copied to a register (all other bits are set to zero) and only a store to the user ALU flags can be executed. All other options will trigger the software interrupt when being executed in user mode. In system mode, all different load and store options are allowed. These different options and their behavior in user/system mode when executing `LDSR` or `STSR` instruction are shown in the table below.

A-bit	B-bit	Mode	READ access (LDSR)	STORE access (STSR)	Software Interrupt
0	0	System mode	Read complete MSR	Write complete MSR	No
0	1		Only read all ALU flags	Only write all ALU flags	No
1	0		Only read system ALU flags	Only write system ALU flags	No
1	1		Only read user ALU flags	Only write user ALU flags	No
0	0	User mode	**Unauthorized access!**	**Unauthorized access!**	**Yes!**
0	1		**Unauthorized access!**	**Unauthorized access!**	**Yes!**
1	0		**Unauthorized access!**	**Unauthorized access!**	**Yes!**
1	1		Only read user ALU flags	Only write user ALU flags	No

Table 11: MSR store options and mode corresponding behavior

The `STAF` instruction is used to directly copy an immediate encoded within the instruction either to the system mode ALU flags or to the user mode ALU flags only. The T, N, O, C, Z bit-fields correlate to the new value the user/system mode ALU flags will be set to. Note, that option bit A must be set to '1' for `STAF` operations. Option bit B encodes if the immediate flag data is written to the system mode ALU flags (B = '0') or to the user mode ALU flags (B = '1'). A direct initialization of the system mode ALU flags using the `STAF` instruction is only allowed in system mode.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. LDSR (load register from machine status register)

```
<LDSR> <Rd>, {usr_flags|sys_flags|alu_flags}
```

2. STSR (store register to machine status register)

```
<STSR> <Rb>, {usr_flags|sys_flags|alu_flags}
```

3. STAF (store immediate to system / user ALU flags)

```
<STAF> <#Imm>, <usr_flags|sys_flags>
```

`<Rd>`	Destination register.
`<Rb>`	Source register.
`<#Imm>`	Five bit immediate, loaded to usr/sys ALU flags.
`{usr_flags|sys_flags|alu_flags}`	Write user ALU flags, system ALU flags, all ALU flags or full MSR, when no argument is present.
`<usr_flags|sys_flags>`	Write user ALU flags or system ALU flags.

Assembler Examples

```
LDSR R1, usr_flags      ; load MSR ALU flags to R1
STSR R3                 ; store R3 to MSR (full access)
STSR R4, usr_flags      ; only write R4 to the user mode ALU flags
STSR R4, alu_flags      ; only write R4 to the all ALU flags
STAF #1, usr_flags      ; set zero flag of the user mode ALU flags
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
LDSR R1, usr_flags      = 0b 00.0110.001.110.0.000 = x"18E0"
STSR R3                 = 0b 00.0111.000.011.0.000 = x"1830"
STSR R4, usr_flags      = 0b 00.0111.110.100.0.000 = x"1E40"
STSR R4, alu_flags      = 0b 00.0111.010.100.0.000 = x"1A40"
STAF #1, usr_flags      = 0b 00.0111.000.111.0.001 = x"1A71"
```

4.2. Memory Access

The instruction encoding of the memory access instructions is shown in the figure below.

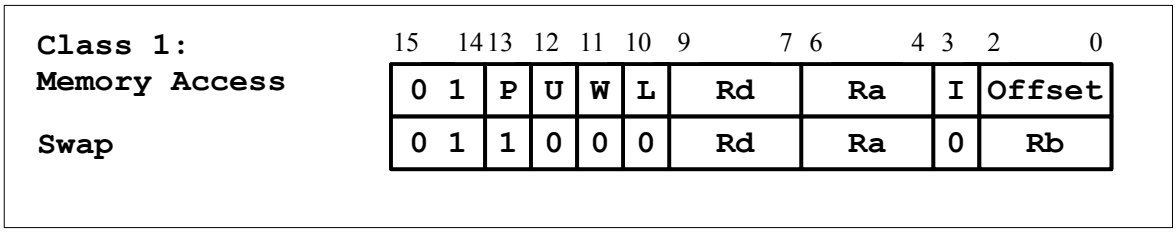

Figure 10: Memory access instructions formats

The memory access instructions allow to move data between a data register and an addressed memory location. Ra always specifies a register, pointing to the accessed memory address. The L-bit determines the data transfer direction. When L is set to '1', the content of Rd is transferred to the memory location addressed (`STR`) by Ra. If the L-bit is set to '0', data from the assigned memory address is loaded into the register (`LDR`), that is specified by the Rd bit-field.

Several different indexing options are implemented. To the memory base address (in Ra), an offset can be added or subtracted (U = '0' subtract, U = '1' add) before or after the actual memory access. Setting the P-bit to '0' will add/subtract the offset before the memory access. When the P-bit is set, the offset will be added/subtracted from or to the base register after the memory access. The result of the operation base +/- offset can be written back to the base register Ra when the W-bit is set. The actual offset can either be a register (I = '0') or a unsigned 3-bit immediate (I = '1').

Bit	Option	Function when set to '0'	Function when set to '1'
13	**P**	Pre-indexing (add/subtract offset to/from base **before** the actual memory access)	Post-indexing (add/subtract offset to/from base **after** the actual memory access)
12	**U**	**Subtract** offset from base register	**Add** offset to base register
11	**W**	Discard result of base+/- offset after memory access	**Write back** the result of base +/- offset to the base register after the actual memory access
10	**L**	**Load** data from memory into a register	**Store** data from a register to memory
3	**I**	Offset is a **register** specified in the offset bit-field	Offset is an unsigned 3-bit **immediate** specified in the offset bit-field

Table 12: Memory access options

One kind of indexing option does not seem logical: A post indexing without a base write back (P = '1' and W = '0'). Here, the post indexing operation is redundant. Therefore, this type of option code is used to specify a new memory access instruction: The atomic memory data swap (`SWP`). This instruction copies the data of the memory location, which is specified by Ra, to Rd and moves afterwards the data of Rb (defined by the Offset bit-field) to the assigned memory location (Rb => M[Ra] => Rd). Hence, a load instruction is followed by a store instruction. Both instructions are tied together (atomic), so no interrupt can be executed before the swap instruction has finished. This is very useful for implementing system semaphores.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. LDR, STR (load/store from/to memory)

```
<LDR|STR> <Rd>, <Ra>, <+|-><Rb|#Imm>, <pre|post>, {!}
```

2. SWP (swap registers with memory)

```
<SWP> <Rd>, <Ra>, <Rb>
```

`<Rd>`	Data register, destination for loads, source for stores.
`<Ra>`	Base address register
`<Rb>`	Source data register for SWP.
`<+|->`	Add or subtracting indexing.
`<Rb|#Imm>`	Offset, register (Rb) or unsigned 3-bit immediate (#Imm).
`<pre|post>`	Pre- (pre) or post- (post) indexing.
`{!}`	Write back indexed base register when present.

Assembler Examples

```
LDR R1, R2, +R3, pre          ; R1 <= M[R2+R3]
LDR R1, R2, +R3, pre, !       ; R1 <= M[R2+R3] and set R2=R2+R3 afterwards
LDR R1, R2, -R3, post, !      ; R1 <= M[R2] and set R2=R2-R3 afterwards
LDR R1, R2, +#2, post, !      ; R1 <= M[R2] and set R2=R2+2 afterwards

STR R4, R5, +#0, pre          ; R4 => M[R5]
STR R4, R5, -R6, pre          ; R4 => M[R5-R6]
STR R4, R5, -#2, pre, !       ; R4 => M[R5-2] and set R5=R5-2 afterwards

SWP R2, R3, R4                ; M[R3] => R2; R4 => M[R3]
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
LDR R1, R2, +R3, pre          = 0b 01.0.1.0.0.001.010.0.011 = x"90A3"
LDR R1, R2, +R3, pre, !       = 0b 01.0.1.1.0.001.010.0.011 = x"98A3"
LDR R1, R2, -R3, post, !      = 0b 01.1.0.1.0.001.010.0.011 = x"68A3"
LDR R1, R2, +#2, post, !      = 0b 01.1.1.1.0.001.010.1.010 = x"78AA"

STR R4, R5, +#0, pre          = 0b 01.0.1.0.1.100.101.1.000 = x"5658"
STR R4, R5, -R6, pre          = 0b 01.0.1.0.1.100.101.0.110 = x"5656"
STR R4, R5, -#2, pre, !       = 0b 01.0.1.1.1.100.101.1.010 = x"5E5a"

SWP R2, R3, R4                = 0b 01.1.0.0.0.010.011.0.100 = x"6134"
```

4.3. Branch and Link

The instruction encoding of the branch and link instructions is shown in the figure below.

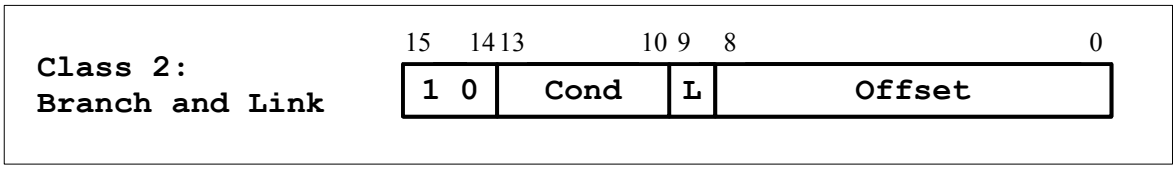

Figure 11: Branch and link instructions format

The branch instruction `B` is used to perform a relative jump to a different location within a range between -256 and +255 words (remember, 1 word = 2 bytes). The offset is stored as two's complement in the offset bit-field. When using the `BL` instruction (with L = '1'), a linked branch is executed. Therefore, the return address (PC + 2 bytes) is stored to the link register LR (= R7). The jump can be conditional when using a specific condition suffix for the `B/BL` instruction from the table below. The different condition suffixes and codes as well as their computation scheme (based on the current state of the ALU flags) are listed in the table below.

ASM Suffix	Cond code	Condition	Condition computation (flags)
EQ	0000	Equal	Z
NE	0001	Not equal	not Z
CS	0010	Unsigned higher or same	C
CC	0011	Unsigned lower	not C
MI	0100	Negative	N
PL	0101	Positive or zero	not N
OS	0110	Overflow	O
OC	0111	No overflow	not O
HI	1000	Unsigned higher	C and (not Z)
LS	1001	Unsigned lower or same	(not C) or Z
GE	1010	Greater than or equal	N xnor O
LT	1011	Less than	N xor O
GT	1100	Greater than	(not Z) and (N xnor O)
LE	1101	Less than or equal	Z or (N xor O)
TS	1110	Transfer flag set	T
AL	1111	Always	1

Table 13: Condition codes

A branch (and link) is only executed if the specified condition is true or when there is no conditional suffix.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

B (branch, conditional or unconditional)

```
<B>{L}{cond} <label>
```

{L}	Store return address to link register when present.
{cond}	Condition code from the table above. If not present, 'always' (AL) condition is used.
<label>	Branch label, relative offset in two's complement (max -256/+255 words).

Assembler Examples

```
      B label_2   ; unconditional branch to "label_2"
label_2:          ; branch destination

      BL subr_1   ; branch to "sub_r" and store return address to LR (=call)
subr_1:           ; this is the subroutine being called
      RET LR      ; return from subroutine

      BCC label_9 ; branch to "label_9" if the carry flag is '0'

      CMP R1, R2  ; compare R1 and R2
      BLEQ subr_3 ; only call "subr_2" if R1 = R2
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
      B label_2         = 0b 10.1111.0.000000001 = x"BA01"
label_2:
      BL subr_1         = 0b 10.1111.1.000000001 = x"BE01"
subr_1:
      BLEQ subr_1       = 0b 10.0000.1.111111111 = x"83FF"
```

4.4. Load Immediate

The instruction encoding of the load immediate instructions is shown in the figure below.

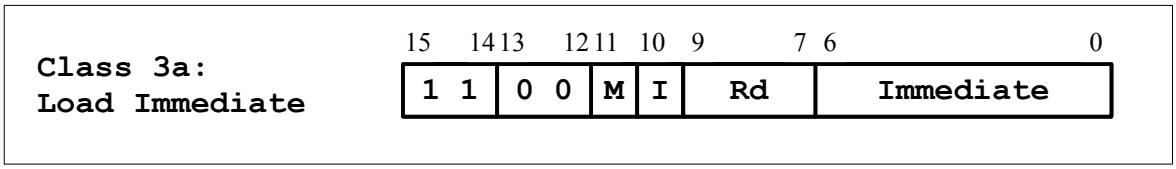

Figure 12: Load immediate instructions format

The load immediate instructions are used to load an 8-bit constant encoded within the instruction to the high byte or sign extended to all bits of the register Rd, respectively. The immediate constant itself is constructed from bit 10 concatenated with bits 6 downto 0 of the instruction word. The `LDIL` (M = '0') mnemonic will load the immediate to the low byte of Rd. All bits of the high byte of Rd will be loaded with the most significant bit of the immediate. This results in a complete load of Rd with the sign (bit 7 of the immediate → bit 10 of the instruction opcode) extended immediate. The `LDIH` (M = '1') mnemonic will load the immediate to the high byte of Rd, leaving the low byte of Rd unchanged. When loading a true 16-bit immediate to register, make sure to load the low byte of it first, otherwise the high byte will be discarded.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

LDIL / LDIH (load immediate 8-bit constant to lower/upper byte)

```
<LDI><L|H> <Rd>, <#Imm>
```

`<L|H>`	Load only high byte of destination register (H) or load whole register with sign extended immediate (L).
`<Rd>`	Destination register.
`<#Imm>`	8-bit “unsigned” immediate value; with present #-prefix.

Assembler Examples

```
(linear execution of all following instructions is assumed)          Register content
LDIL R4, #255     ; load sign extended 255 (= -1) to R4     (R4 = x”FFFF”)
LDIL R4, #2       ; load sign extended 2 to R4              (R4 = x”0002”)
LDIH R4, #7       ; load 7 to the high byte of R4           (R4 = x”0702”)
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x”...”) format, where the dots in the binary format present the different bit-fields.

```
LDIL R4, #255     = 0b 11.00.0.1.100.1111111 = x”C67F”
LDIL R4, #2       = 0b 11.00.0.0.100.0000010 = x”C202”
LDIH R4, #7       = 0b 11.00.1.0.100.0000111 = x”CA07”
```

4.5. Bit Manipulation

The instruction encoding of the bit manipulation instructions is shown in the figure below.

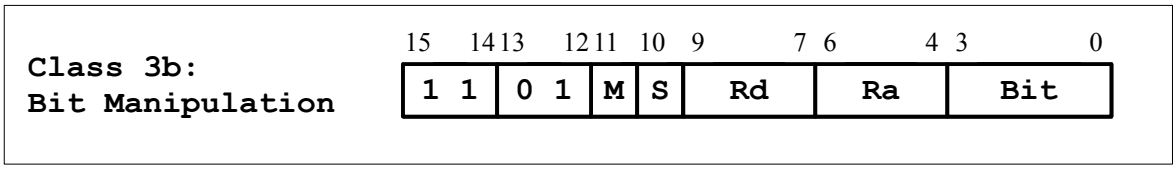

Figure 13: Bit manipulation instructions format

The bit manipulation instruction are used to manipulate a single bit of a register and to store the result to the same or another register, whereas the previous state of the bit is irrelevant. The actual bit is addressed by an 4-bit immediate in the Bit-field.

The `SBR` instruction will set the assigned bit to '1', whereas the `CBR` instruction clears the bit. A store of the assigned bit to the T-flag is possible by using the `STB` instruction. For this case, the Rd bit-field is irrelevant and must be set to “000”. The `LDB` instruction loads the current state of the T-flag to the assigned bit. The different option codes (M and S bits) of the four bit manipulation instructions are shown in the table below.

M	S	Function
0	0	Take data from register Ra, **clear** the assigned bit and store the result to Rd
0	1	Take data from register Ra, **set** the assigned bit and store the result to Rd
1	0	Take data from register Ra, **load** the T-flag to the assigned bit and store the result to Rd
1	1	Take the assigned bit from register Ra and **store** it to the T-flag; no data write back to Rd

Table 14: Bit manipulation operations

Inverted T-Flag transfer

The Atlas CPU only features a T-flag-based branch, that is executed whenever the T-flag is set (`BTS / BLTS`). But for many applications it might be necessary to branch when a bit, stored to the T-flag, is cleared. Therefor, a more efficient way than using two branches have been implemented. The bit of a register, which stored the T-flag, can be inverted during the transfer to adapt to this situations. Then, a `BTS` branch command will execute when the original bit of the register is zero. To invert a bit while it is being transferred to the T-flag, use the “store bit to T-flag and invert” instruction `STBI`. The original source bit of the register is not affected by this instruction. The inverted transfer mode is indicated by setting bit of the unused destination register bit-filed to '1'.

Parity of a Register

The parity of a register is determined by the number of bits, that are set ('1'). An even number of '1's results in a even parity (Parity = 0), an off number of '1' results in an odd parity (Parity = 1). Hence, the actual parity is computed by an XOR of all register bits. The Atlas CPU supports hardware for directly generating the parity result of a register. Use the `SPR` instruction (store parity) to directly store the parity of the source register to the T-flag. To indicate this instruction, the unused bit 8 of the destination field is set. Of course it is also possible to store the inverted parity bit using the `SPRI` instruction to the T-flag (in this case bit 7 and 9 are set). For this instructions the bit-address-filed (bit 3:0) is not used and should be set to “0000”.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. SBR, CBR (set/clear bit from register Ra and write result to Rd)
```
<SBR|CBR> <Rd>, <Ra>, <#Imm>
```

2. LDB (load bit from T-flag and write result to Rd)
```
<LDB> <Rd>, <Ra>, <#Imm>
```

3. STB/STBI (store bit to T-flag / store inverted bit to T-flag)
```
<STB>{I} <Ra>, <#Imm>
```

4. SPR/SPRI (store parity to T-flag / store inverted parity to T-flag)
```
<SPR>{I} <Ra>
```

`{I}`	Invert source/parity bit while it is transferred to the T-flag when present.
`<Rd>`	Destination register.
`<Ra>`	Source register.
`<#Imm>`	4-bit immediate value assigning the desired bit; with present #-prefix.

Assembler Examples

```
SBR  R3, R4, #4   ; set bit 4 of R4's data and store result to R3
CBR  R0, R0, #12  ; clear bit 12 of register R0
STB  R7, #1       ; store bit 1 of R7 to the T-flag
STBI R7, #1       ; store inverted bit 1 of R7 to the T-flag
LDB  R7, R0, #5   ; copy T-flag to bit 5 of R0's data and store result
                  ; to R7
SPR  R7           ; store parity of r7 to the T-flag
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
SBR  R3, R4, #4   = 0b 11.01.0.1.011.100.0100 = x"D5C4"
CBR  R0, R0, #12  = 0b 11.01.0.0.000.000.1100 = x"D00C"
STB  R7, #1       = 0b 11.01.1.1.000.111.0001 = x"DC71"
STBI R7, #1       = 0b 11.01.1.1.001.111.0001 = x"DCF1"
LDB  R7, R0, #5   = 0b 11.01.1.0.111.000.0101 = x"DB85"
SPR  R7           = 0b 11.01.1.1.010.111.0000 = x"DD70"
```

4.6. Coprocessor Data Processing

The instruction encoding of the coprocessor data processing instructions is shown in the figure below.

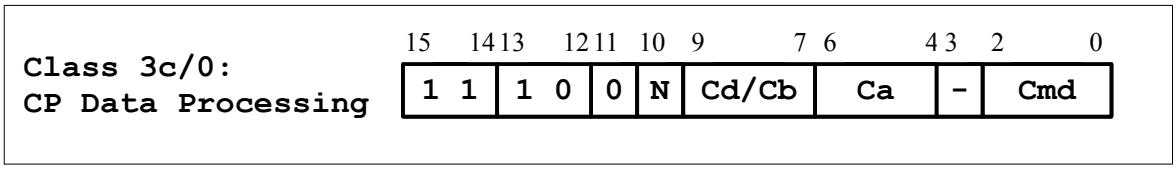

Figure 14: Coprocessor data processing instructions format

The coprocessor data processing instruction `CDP` is used to control one of the two external coprocessor to perform a specific coprocessor-internal operations. The actual functionality of this instruction correspond to the implemented coprocessor. However, it is designed to specify two coprocessor registers, which can be used as source and as source and destination register for operations. A function control can be determined via the three-bit CMD immediate bit-field. Register addresses as well as the command opcode are directly displayed to the coprocessor port. See the coprocessor chapter in the architecture section of this data sheet for more information.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

CDP (coprocessor data processing)

```
<CDP> <#CP>, <Ca>, <Cb>, <#Cmd>
```

`<#CP>`	Coprocessor ID (“#0” or “#1”)
`<Ca>`	Coprocessor operand A / destination register.
`<Cb>`	Coprocessor operand B register.
`<#Cmd>`	3-bit immediate value presenting a coprocessor command.

Assembler Examples

```
CDP #0, C0, C0, #4      ; instruct CP 0 to execute command 4 on registers
                        c0 and c0 and place result in register c0
CDP #1, C7, C3, #1      ; instruct CP 1 to execute command 1 on registers
                        c7 and c3 and place result in register c7
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x”...”) format, where the dots in the binary format present the different bit-fields.

```
CDP #0, C0, C0, #4      = 0b 11.10.0.0.000.000.0.100 = x”E004”
CDP #1, C7, C3, #1      = 0b 11.10.0.1.111.011.0.001 = x”E7B1”
```

4.7. Coprocessor Data Transfer

The instruction encoding of the coprocessor data transfer instructions is shown in the figure below.

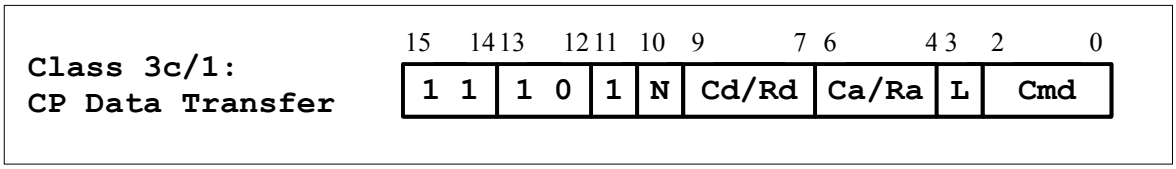

Figure 15: Coprocessor data transfer instructions format

To exchange data between a coprocessor register and an Atlas CPU register, the `MRC` (load data from coprocessor) and `MCR` (store data to coprocessor) instructions are used. Parallel to the data transfer, a command can be specified to trigger additional coprocessor operations. The L-bit determines the transfer direction (load: L = '0', store: L = '1').

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

1. MRC (load data from coprocessor)

```
<MRC> <#CP>, <Rd>, <Ca>, <#Cmd>
```

2. MCR (store data to coprocessor)

```
<MRC> <#CP>, <Cd>, <Ra>, <#Cmd>
```

`<#CP>`	Coprocessor ID (“#0” or “#1”)
`<Cd>`	Coprocessor destination register.
`<Ca>`	Coprocessor source register.
`<Rd>`	CPU destination register.
`<Ra>`	CPU source register.
`<#Cmd>`	3-bit immediate value presenting a coprocessor command.

Assembler Examples

```
MRC #0, R3, C4, #1      ; CP0: R3 <= C4 and execute CMD 1
MCR #1, C7, R3, #0      ; CP1: C7 <= R3 and execute CMD 0
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x”...”) format, where the dots in the binary format present the different bit-fields.

```
MRC #0, R3, C4, #1      = 0b 11.10.1.0.011.100.0.001 = x”E9C1”
MCR #1, C7, R3, #0      = 0b 11.10.1.1.111.011.1.000 = x”EFB8”
```

4.8. Multiply-and-Accumulate

The instruction encoding of the multiply and multiply-and-accumulate instructions are shown in the figure below.

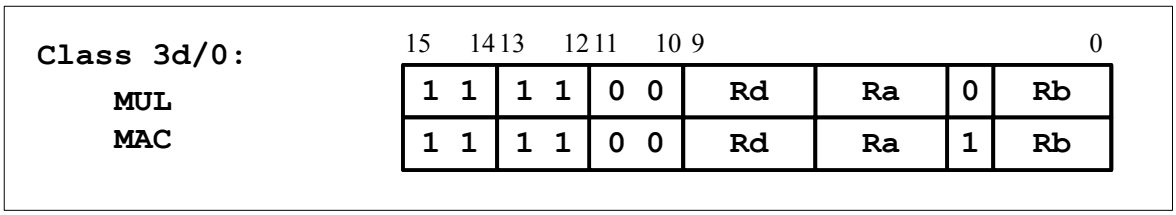

Figure 16: MUL / MAC instruction formats

These instructions provide extended arithmetical functions for multiplication operations. The `MUL` instruction will multiply Ra and Rb and place the lowest 16 result bits in Rd (Rd <= (Ra*Rb)(15:0)). The `MAC` instruction will also multiply Ra and Rb, but will also add the content of Rd to the lowest 16 bits of the multiplication result. The result is then placed in Rd (Rd <= (Ra*Rb)(15:0)+Rd). None of these instruction will perform any flag modification. Since a multiplication with an optional addition requires a lot of area, the actual synthesis of the `MUL` and `MAC` instructions can be enabled or disabled using architecture constants in the Atlas Processor package VHDL file. By default, only the `MUL` instruction will be synthesized. When trying to execute an instruction, that has not been synthesized (in this case the multiply-and-accumulate instruction), the software interrupt trap will be taken.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

MUL, MAC (multiply / multiply-and-accumulate)

```
<MUL|MAC> <Rd>, <Ra>, <Rb>
```

`<Rd>`	Destination register / accumulation source four MAC operations.
`<Ra>`	Operand A register.
`<Rb>`	Operand B register.

Assembler Examples

```
MUL R0, R1, R2    ; R0 = R1 * R2
MAC R0, R1, R2    ; R0 = R1 * R2 + R0
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
MUL R0, R1, R2    = 0b 11.11.00.000.001.0.010 = x"F012"
MAC R0, R1, R2    = 0b 11.11.00.000.001.1.010 = x"F01A"
```

4.9. Undefined Instructions

The instruction encoding of the undefined instructions is shown in the figure below.

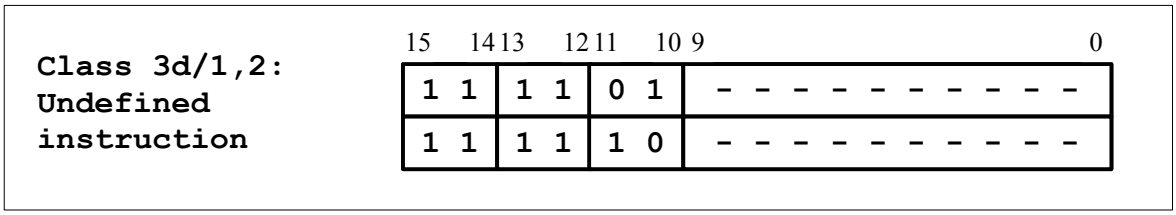

Figure 17: Undefined instruction formats

These instruction types are not implemented yet and are used to keep some space for further instruction set extensions. Therefore, these instructions **should not be used**. However, when executed, the undefined instructions will behave like a system call with a tag corresponding to the lowest 10 bits of the instruction.

4.10. System Call

The instruction encoding of the system call instruction is shown in the figure below.

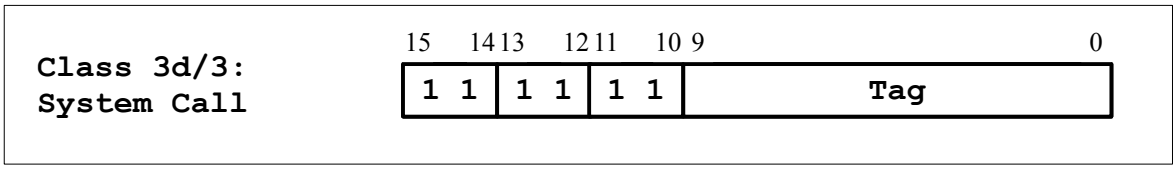

Figure 18: System call instruction format

The system call (`SYSCALL`) instruction is used to enter system mode from a running user program (software interrupt). When executed, program execution will stop, the re-entry point (return address) plus 2 bytes offset will be stored in the system link register, the mode will be changed to system mode and program execution will resume at the software interrupt address. The lowest 10 bits of the instruction can be used to directly transfer an argument (tag) to the software interrupt handler. This tag can be extracted by the handler after loading the system call's causing instruction.

When executing the `SYSCALL` instruction in system mode, the instruction will behave like a branch and link instruction to the software interrupt vector. When returning with RTX from the software interrupt handler, the original program will be resumed in user mode rather than in system mode.

Note: The software interrupt will also be executed for example whenever a user-mode program attempts an unauthorized access to a coprocessor or to restricted bits of the MSR.

Assembler Syntax

Items in { } are optional, whereas items in < > are required. Note the spaces and commas introduced by the lexical rules.

SYSCALL (software interrupt by system call)

```
<SYSCALL> {#Tag}
```

`{#Tag}` 10-bit immediate value, automatically set to zero if not present; use #-prefix.

Assembler Examples

```
SYSCALL #1002     ; trigger software interrupt with '1002' as tag
SYSCALL           ; trigger software interrupt with no tag
```

Coding Examples

The assembled instruction are shown in binary (0b ...) and hexadecimal (x"...") format, where the dots in the binary format present the different bit-fields.

```
SYSCALL #1002     = 0b 11.11.11.1111101010 = x"FFEA"
SYSCALL           = 0b 11.11.11.0000000000 = x"FC00"
```

5. Atlas Evaluation Assembler

I've programmed a small assembler, that is capable of assembling the previously explained instructions into an Atlas CPU-compatible VHDL program memory initialization file. The assembler is still very rudimentary, but it can already be used to write and assemble complex programs. The program is located in the *core/asm* folder and can be run using the Windows command prompt. The actual assembly program is passed as first argument when calling the assembler. A simple example program, which introduces the lexical rules and the basic layout of an assembler program, can also be found in that folder.

To assemble this example file, type and execute this in your Windows command prompt:

```
...\core\asm>atlas_asm ../examples/test.asm
```

In the folder of the assembler executable, the program will generate a "*init.vhd*" file, which contains the data initialization area of a VHDL memory declaration (the program memory). The "*out.bin*" file contains the assembled program in binary format and is dedicated for the future use of a bootloader. All warnings and error will also display the corresponding line. These lines correspond to the "*pre_processor.asm*" file.

Note: The assembler is not case-sensitive.

5.1. Pre-Processor Instructions

The pre-processor instruction can make assembler-life much more easy, since they present different features to create more abstract programs. See the *test.asm* file in the *core/asm* folder for an example assembler program including all the different pre-processor instructions.

Only some rudimentary instructions are supported yet, but hopefully the pre-processor capabilities will grow in future ;)

Instruction	Example	Function

`.equ`	`.equ temp r4`
	`.equ sys_reg c1`
	`.equ de_val #1`
	`.equ mem_size #256`
	`.equ test_b #0b10100011`
	`.equ test_h #0xac`
		This instruction allows to use aliases for the CPU register (r0, …, r7), the coprocessor register (c0, …, c7) or immediate values (positive integers, 16-bit, decimal/ binary/ hexadecimal representation, introduced with '#'-prefix)

`.space`	`.space #4` `.space mem_size`	The space instruction will create an area of a given size, that is initialized with zeroes (x"0000" = NOPs)
`.dw`	`.dw #23432`	The dw instruction can be used to directly initialize the corresponding memory position with a positive, 16-bit immediate (decimal value, introduced with '#'-prefix), with a previously defined .equ-definition or with a branch label address ("[label]")
`.stringz`	`.stringz "Hey there!"`	With the .string instruction you can initialize memory directly with an ASCII string. All ".stringz" strings are automatically terminated with a zero.

Table 15: Pre-processor instructions

5.2. Example Programs

This chapter presents some example program fragments, that illustrate how to use the Atlas assembler mnemonics to create your own application programs. Note, that of course all code fragments need to be included into a 'real' program to run properly.

5.2.1. Bit Test

This is an example of how to use the T-flag to implement bit test operations.

```
      ;Bit-testing operation
      ;executed in system or user mode

      STB R0, #9                ; store bit 9 of r0 to T-flag to test it
      BTS bit_set               ; branch to "bit_set" when r0[9] is '1'
bit_clear: ...                  ; execute this when bit is cleared (redundant label)
bit_set: ...                    ; execute this when bit is set
```

Bit test operations are also very often used to leave a linear program execution. Since the BTS (branch if T-flag is set) instruction only executes, when the T-flag is set, the following implementation of a taken branch whenever a bit is zero seems obvious.

```
      ;Branch when bit is cleared (bad implementation)
      ;executed in system or user mode

      ADD  R0, R4, R3           ; begin of linear program (just an example)
      STB  R0, #9               ; store bit 9 of r0 to T-flag to test it
      BTS  bit_is_set           ; continue linear program execution when bit is set
      B    bit_cleared          ; branch to "bit_clear" when r0[9] is '0'
bit_set:
      SUB  R2, R1, R4           ; end of linear program (just an example)
      ...

bit_clear: ...                  ; execute this when original bit r0[9] is zero
```

But we can do better than that! The bit, which is stored to the T-flag, can be inverted during the transfer. Thus, a true zero-testing branch using also the BTS instruction can be implemented.

```
      ;Branch when bit is cleared (good implementation)
      ;executed in system or user mode

      ADD  R0, R4, R3           ; begin of linear program (just an example)
      STBI R0, #9               ; store inverted bit 9 of r0 to T-flag to test it
      BTS  bit_ clear           ; branch to "bit_clear" when r0[9] is '0'
      SUB  R2, R1, R4           ; end of linear program (just an example)
      ...

bit_clear: ...                  ; execute this when original bit r0[9] is zero
```

5.2.2. Comparing Large Operands

The CPX instructions allows to compare two registers while also taking the zero and carry flags of a previous comparison into account. This is very suitable for implementing a comparison of two arbitrarily wide operands.

```
      ;48-bit comparison
      ;executed in system or user mode

      ; R2, R1, R0 contain 48-bit operand A (r2 most / r0 least significant bits)
      ; R5, R4, R3 contain 48-bit operand B (r5 most / r3 least significant bits)

      CMP R0, R3                ; start to compare the least significant bits
      CPX R1, R4                ; CPX = compare and also take flags into account
      CPX R2, R5                ; finish with comparing the most significant bits

      BEQ equal                 ; go to "equal" when A=B
      BMI a_negative            ; go to "a_negative" when A is negative
      BHI a_uhigher             ; go to "a_uhigher" when A is unsigned higher than B
```

5.2.3. Loop Counters

Conditional loops are one of the basic elements within a program. The following example shows an example of how to implement loops with a small overhead.

```
      ;loop counters
      ;executed in system or user mode

      LDIL R0, #16              ; this is the loop counter → 16 iterations
loop_begin:                     ; beginning of loop
      ...                       ; repeat this 16 times
      DECS R0, R0, #1           ; decrement loop counter and set flags
      BNE  loop_begin           ; branch to "loop_begin" if r0 is not zero
      ...
```

5.2.4. MAC Operation with Flag Update

Neither the MAC nor the MUL instruction features a status flag update. Also, a synthesized MAC instruction requires a lot of additional hardware resources. So, if a MAC instruction with flag update is required, it is suitable to only allow the synthesis of the MUL instruction and construct the actual MAC operation with additional instructions.

```
      ;constructed MAC operation with flag update
      ;executed in system or user mode

      ; compute R0=R1*R2+R3 and set flags corresponding to the result

      MUL  R0, R1, R2     ; R0 = R1 * R2
      ADDS R0, R0, R3     ; R0 = R0 + R3 and set status flags
```

5.2.5. Branch Tables

Branch or call tables are a good method to easily jump to different locations, without the need of comparing a register with immediate values. For example, this kind of value-defined branching can be used to trigger different operation using the system call instruction with a tag, where this tag represents the actual subroutine number, that shall be called. Note, that in the following example, only 16-bit addresses are used. Thus, the subroutine must be in the same page as the branch-table code.

```
      ;branch/call table (subroutine addresses are 16-bit, so in the same page)
      ;executed in system or user mode

      ; R4 presents the number of subroutine to be called
      ; thus, a 2 in R4 would call subroutine_2

      ; first we have to load the 16-bit base address of the branch table
      LDIL R0, #low[branch_table]      ; load low byte of label address
      LDIH R0, #high[branch_table]     ; load high byte of label address

      ; multiply index by two by left-shifting one position; this is necessary, because
      ; each subroutine address in the table is 16-bit wide and the Atlas CPU uses
      ; byte addressing mode by default
      SFT R4, R4, #LSL

      LDR R1, R0, +R4, PRE      ; add offset to base and load address to r1
      GTL R1                    ; goto and link → branch to the loaded address in r1 and
                                ; save return address to the link register
      ...

branch_table:                   ; beginning of branch table
.DW [subroutine_0]              ; absolute 16-bit address of label "subroutine_0"
.DW [subroutine_1]              ; absolute 16-bit address of label "subroutine_1"
.DW [subroutine_2]              ; absolute 16-bit address of label "subroutine_2"
.DW [subroutine_3]              ; absolute 16-bit address of label "subroutine_3"
...
```

5.2.6. Stack Operations

A stack is a common data structure of many applications. The Atlas CPU provides indexing memory access instruction do directly modify the stack pointer while loading or storing data from or to the stack. The following example shows how to implement push and pop operations for positive (from low to high memory addresses) and negative (from high to low memory addresses) growing stacks.

```
      ;stack operation (stack pointer is R6 by default)
      ;executed in system or user mode

      ; positive growing stack
      STR R0, R6, +#2, post, !        ; push r0 on the stack
      LDR R0, R6, -#2, pre, !         ; pop r0 from the stack

      ; negative growing stack
      STR R0, R6, -#2, post, !        ; push r0 on the stack
      LDR R0, R6, +#2, pre, !         ; pop r0 from the stack
```

5.2.7. Strings

String are used to process data arrays. For instance, these can be arrays of ASCII characters. The following example shows how to compare two character strings, which are equal in length.

```
      ;compare two strings
      ;executed in system or user mode

      LDIL R0, #low[string_1]          ; load low byte address of string 1
      LDIH R0, #high[string_1]         ; load high byte address of string 1
      LDIL R1, #low[string_2]          ; load low byte address of string 2
      LDIH R1, #high[string_2]         ; load high byte address of string 2
      LDIL R2, #11                     ; length of strings (11 words)

compare_loop:
      LDR  R3, R0, +#2, post, !        ; load next word of input string 1 into r3
      LDR  R4, R1, +#2, post, !        ; load next word of input string 2 into r4
      CMP  R3, R4                      ; compare two words
      BNE  strings_diff                ; leave loop when not equal
      DECS R2, R2, #1                  ; decrement loop counter
      BNE  compare_loop                ; leave loop when r2 is zero

strings_equal: ...                     ; string are equal when arriving here
strings_diff: ...                      ; string are not equal when arriving here

string_1:
.stringz "This is a demo string"       ; string 1 (21 char bytes + zero → 11 words)
string_2:
.stringz "This is a DEMO string"       ; string 2 (21 char bytes + zero → 11 words)
```

5.2.8. Count Leading Zeros

This example shows how to count the number of leading zeros of a register.

```
      ;count leading zeros of r0, output in r1
      ;executed in system or user mode

      ; load demo data 1478 to r0 → 5 leading zeros
      LDIL R0, #0b11000110       ; load low part of dummy data using binary format
      LDIH R0, #0b00000101       ; load high part of dummy data using binary format

      LDIL R1, #16               ; r1 is 16 if all of r0's bits are zero

      TEQ R0, R0                 ; is r0 already zero?
      BEQ end                    ; skip counting if r0 is zero

start: CLR R1                    ; start of loop: clear counter register r1
loop:  SFTS R0, R0, #LSL         ; shift msb of r0 into carry flag
      BCS end                    ; terminate if a '1' was found
      INC R1, R1, #1             ; increment zero-counter
      B loop

end:   ...                       ; number of leading zeros is in r1
```

5.2.9. LFSR Implementation using Parity of a Register

This example shows how to use the parity hardware to implement a LFSR (linear feedback shift register) for pseudo-random number generation. Therefore, bit 15, 14, 12 and 3 of the LFSR (the taps) are XOR-ed and left-shifted into the LFSR to produce the next value. The actual XOR function of all bits of a register is done by using the parity generation instruction.

```
      ;LFSR implementation
      ;executed in system mode (just for this example)

      LDIL R0, #1               ; load LFSR seed (=1)

      LDIL R1, #0b00001000      ; load low part of LFSR taps (bit 3)
      LDIH R1, #0b11010000      ; load high part of LFSR taps (bits 15, 14, 12)

loop: AND  R2, R0, R1           ; isolate tap bits in r2
      SPR  R2                   ; XOR all bits of r2 and store result to the T-flag,
                                ; this is done by storing r2's parity to the T-flag

      LDSR R2                   ; load MSR to r2
      LDB  R2, R2, #6           ; copy sys-T-flag to the system-mode carry flag
      STSR R2                   ; store r2 to MSR

      SFT  R0, R0, #RLC         ; rotate right and use carry flag as bit #0 input
      B    loop                 ; resume loop
```

6. Core Architecture

This chapter takes a closer look at the actual rtl implementation of the CPU core.

6.1. Module Description

The following table presents the different Atlas VHDL rtl files and their functionality.

File name	Functionality
ATLAS_PROCESSOR.vhd	This is the top entity of the complete processor system, including all CPU core modules together with the MMU and the bus interface.
BUS_INTERFACE.vhd	The bus interface incorporates a shared I/D-cache as well as a Wishbone compatible host controller to communicate with other modules of the SoC.
MMU.vhd	The memory management unit, implemented as system coprocessor, allows to extend the accessible memory space by distributing a 4GB address space into pages of 64kB.
ATLAS_MICRO.vhd	Top entity of the Micro implementation. The module instantiates the CPU core and provides an internal shared/separated instruction and data memory.
ATLAS_pkg.vhd	CPU package file. All architecture constants and system configuration options can be found here. Also, the endianness, synthesized hardware modules and present coprocessor are declared in this file.
ATLAS_CORE.vhd	Top entity of the CPU providing all external interface signals to directly communicate with RAM/ROM; all CPU sub modules are instantiated in this file.
OP_DEC_vhd	Opcode decoder. The instruction opcodes are decoded into processor internalcontrol signals in this unit.
CTRL.vhd	This file provides the control “spine” of the processor. Intermediate control signal computations and the signal buffers for each pipeline stage are located here.
SYS_REG.vhd	The system register file contains the program counter, the machine status register and the interrupt and context control circuits.
REG_FILE.vhd	This file contains the main data register file, organized as 16*16-bit memory.
ALU.vhd	The ALU holds the primary arithmetical/logical unit, the coprocessor interface as well as the multiplication unit (if synthesized).
MEM_ACC.vhd	All data memory requests emerge from this unit. Furthermore, processing result routing circuits are located here.
WB_UNIT.vhd	The write-back unit takes data from the coprocessors, the ALU or the data memory interface and writes it back to the register file.

Table 16: Atlas project VHDL rtl files description

6.2. Data Path

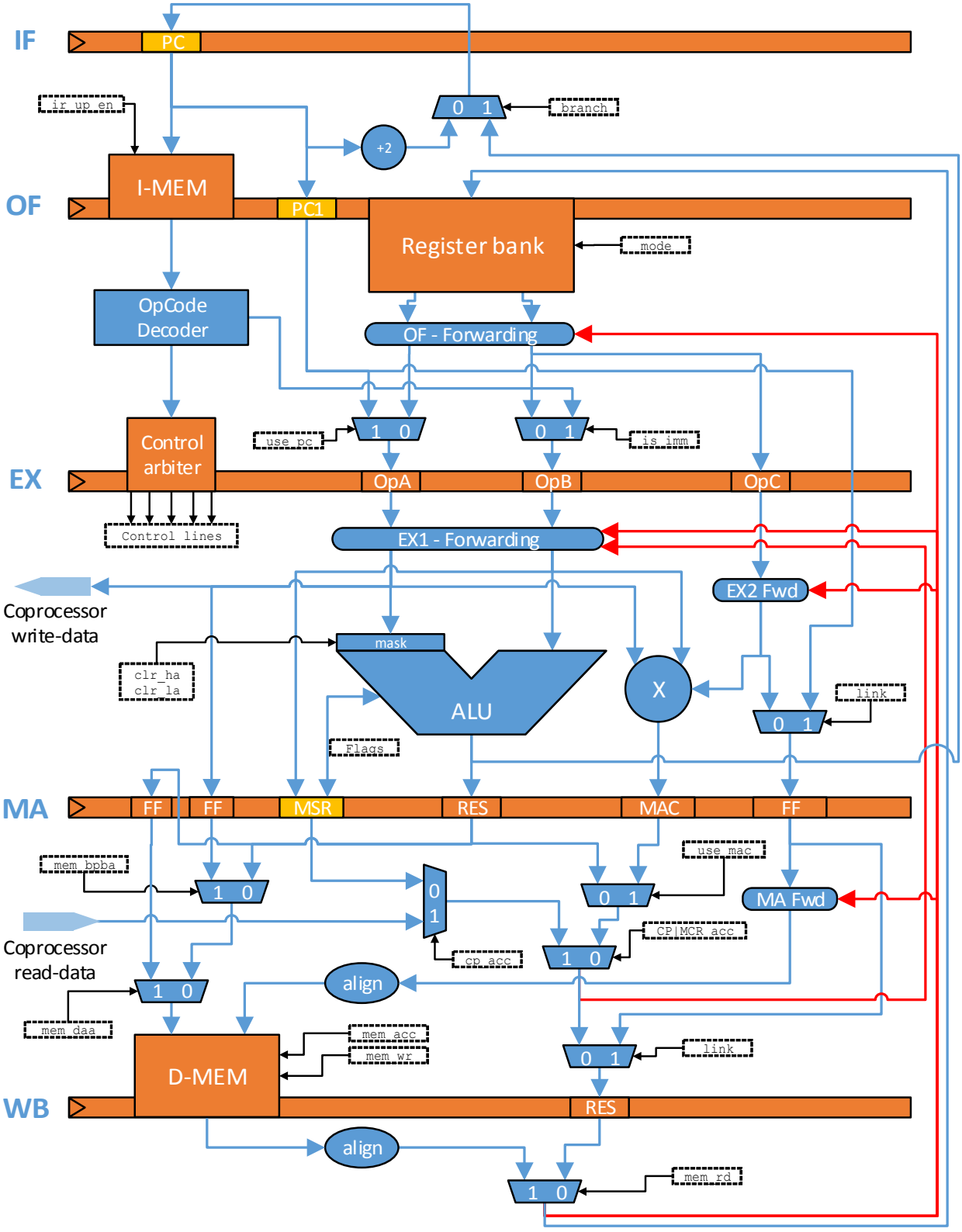

Figure 19: Atlas CPU main data path diagram

6.3. Data Registers

For efficient hardware implementation, the 16 data registers are mapped to a 16x16-bit memory block. The most significant bit of the register address (bit 3) indicates the accessed bank ('0' = user bank, '1' = system bank). The actual register – memory cell mapping is presented in the table below.

0000: User R0	**0100:** User R4	**1000:** System R0	**1100:** System R4
0001: User R1	**0101:** User R5	**1001:** System R1	**1101:** System R5
0010: User R2	**0110:** User R6	**1010:** System R2	**1110:** System R6
0011: User R3	**0111:** User R7	**1011:** System R3	**1111:** System R7

Figure 20: Register mapping to memory block

Note: The register file might be implemented using LUT registers instead of dedicated memory blocks on some FPGAs, since not all FPGA architectures provide dedicated memory, that can be accessed asynchronously when reading data.

6.4. Pipeline

A classical 5-stage pipeline is implemented in the Atlas CPU. Just to clarify the terms of “pipeline stages”, a stage starts always with the update of the register, that drive a specific stage. Also, a cycle starts with the update of a register on a rising edge of the system clock. The table below shows the present pipeline stages of the CPU.

Stage #	Name	Functionality
1: **IF**	Instruction fetch	At the beginning of this stage, the program counter (PC) is updated with the next instruction address. For linear programs, this value for the PC is old_value plus 2 bytes. This address is then applied to the instruction memory.
2: **OF**	Instruction decode and operand fetch	The instruction memory accepts the address and outputs the corresponding instruction on the rising edge of the system clock. The opcode decoder decodes the opcodes an loads operand form the register file and also constructs immediate values.
3: **EX**	Execution	In the execution stage, the main data processing takes place. Furthermore, data is presented to the external coprocessors, the PC and the MSR, depending on the current instruction.
4: **MA**	Memory access	The memory access stage provides write data and the correlated address to the data memory. Also, data read backs from the coprocessor are read in this cycles.
5: **WB**	Write back	The write back stage accepts read data from the memory or any kind of read data from the previous stage (coprocessor, MSR, ALU processing result) and applies it to the register file, whenever a data write back is valid. With the next rising edge, this data is stored to the destination register and thus the execution cycle is completed.

Table 17: Atlas CPU pipeline stages

6.4.1. Local Pipeline Conflicts

Whenever data is needed, that has already been processed but has not yet reached the end of the pipeline, a local data dependency occurs. For data, that will be processed by the ALU, the source and destination data can be separated by 1, 2 or 3 cycles in the pipeline. The following example program illustrates these types of local conflicts (the `NOPs` are only exemplary used to generate the corresponding distances).

```
;1 cycle distance:
inc r4, r1, #1          ; r4 = r1 + 1
cmp r4, r1              ; compare r4 and r1

;2 cycles distance:
dec r5, r1, #1          ; r5 = r1 - 1
nop
tst r5, r5              ; set flags to r5 XOR r5

;3 cycles distance:
sft r6, r1, #swp        ; swap bytes of r1 and store to r6
nop
nop
add r6, r6, r6          ; r6 = r6 * 2
```

Two different forwarding units are used to prevent pipeline stalls whenever these kinds of local data dependencies occur. The first one is located in the OF-stage and can forward data from the WB-stage (data separation by 3 cycles) into the two operand slots of the ALU. The second one is located in the EX-stage and can forward data from the MA-stage (data separation by 1 cycle) and from the WB-stage (data separation by 2 cycles) into the two operand slots of the execution stage (EX).

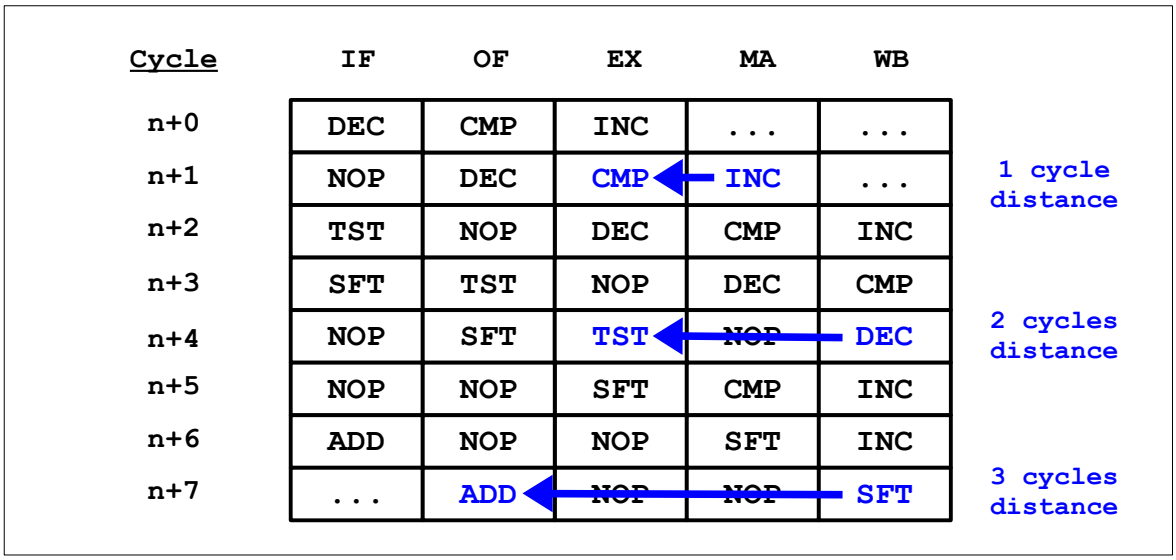

Figure 21: Processing data forwarding

Furthermore, the CPU features two small additional forwarding units to accelerate memory data transfers. The first one is also located in the EX-stage and can forward data from the WB-stage into the ALU bypass operand slot. The second one is located in the MA-stage and can forward data from the WB-stage into the write data port of the data memory.

6.4.2. Temporal Pipeline Conflicts

Temporal data dependencies occur, whenever the operand fetch stage tries to forward data for ALU processing that has not been yet fetched from the data memory. The following example illustrates this kind of data conflict.

```
;memory read-data dependency
ldr r1, r0, +#2, pre      ; r1 = MEM[r0+2], not address pointer update
inc r1, r1, #1            ; r1 = r1 + 1
```

This type of dependency cannot be solved by forwarding alone. The CPU has to insert an empty "dummy cycle" (a NOP) to stop the data processing instruction in the OF-stage until the source data from the memory is available.

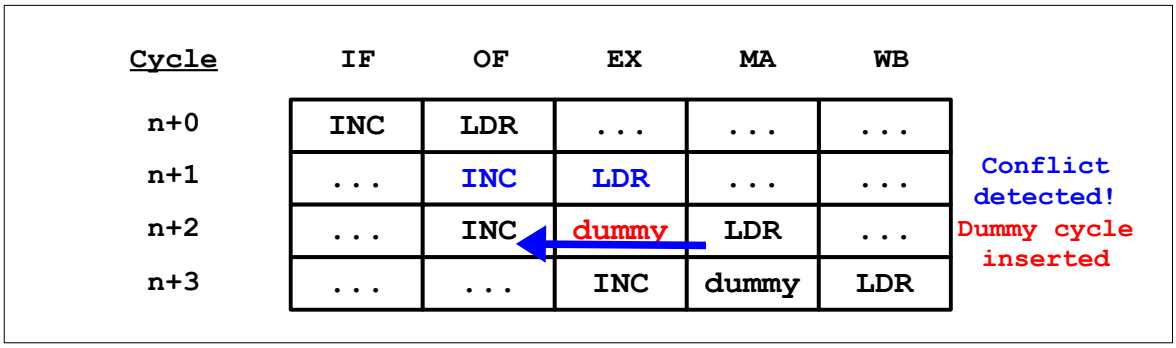

Figure 22: Memory read-data temporal data dependency

While the INC instruction is still in the OF-stage, the memory load instruction (LDR) has reached the MA-stage and the fetched data can be forwarded to the OF-stage.

6.4.2.1. MSR Write Access

Whenever the machine status register (MSR) is updated via the `STSR` (or an alias instruction like `RTX`) instruction, a dummy cycle has to be inserted afterwards. Imagine a system mode program, that clears the M-flag by writing new data to the MSR to switch to user mode.

```
;MSR update flag dependency
ldsr r1              ; r1 = MSR
cbr  r1, r1, #15     ; r1[15] = '0', clear M-flag
stsr r1              ; MSR = r1 (in system mode, switching to user mode)
inc  r4, r4, #1      ; r4 = r4 + 1 (user bank registers)
```

The operand fetch has to wait until this update is completed, because the M-flag determines the most significant bit of the register addresses and thus the actual register bank, where data is taken from. Since the M-flag is cleared now, the new data for the `INC` instruction has to be fetched from the user register bank and not from the system register bank. Therefore a dummy instruction slot is necessary.

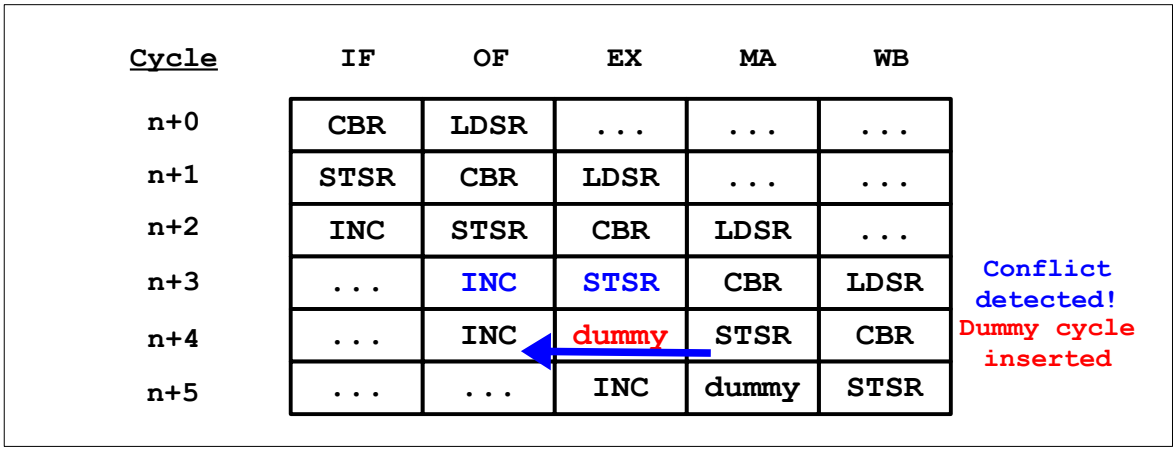

Figure 23: MSR update, status dependency

Even if only the mode (M) and the transfer (T) flags are vulnerable for these kind of conflicts, any kind of manual MSR update causes the system to insert a dummy cycle – this simplification dramatically reduces the hardware overhead. But since MSR updating instructions are very rare in most program codes, this issue should not be further relevant.

6.4.4. Branches

Branches are necessary to leave the linear processing of a program. They occur whenever an unconditional or a conditional branch instruction with fulfilled condition is executed. Also, a manual PC write access via the `STPC` instruction (or any alias instruction like `RET`) will result in a branch to the new address. The Atlas CPU does not use any kind of branch prediction, therefore the strategy is "branches are always taken".

```
      ;Branches
      b label_1           ; go to label_1 (unconditional)
      add r0, r0, r1      ; r0 = r0 + r1 (obsolete!)
      sub r2, r2, r1      ; r2 = r2 - r1 (obsolete!)
      orr r3, r3, r1      ; r3 = r3 | r1 (obsolete!)
label_1:
      inc  r4, r4, #1     ; r4 = r4 + 1
```

When the PC is loaded with a new address, the instructions, which were already loaded after the branch causing instruction into the pipeline, have to be invalidated ("pipeline flush").

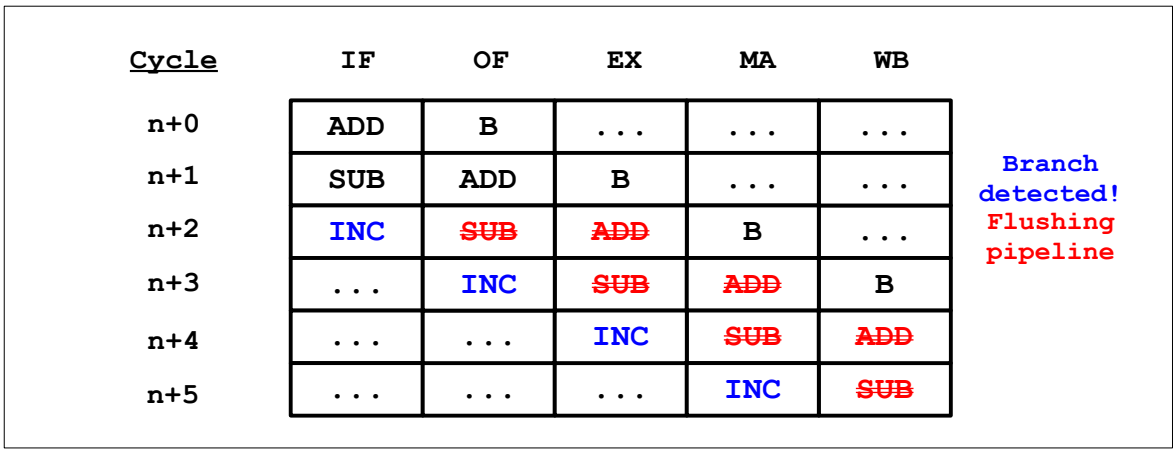

Figure 24: Flushing the pipeline after a taken branch

Since it takes two cycles to fetch a new instruction into the opcode decoding OF-stage after a nonlinear PC update, the two following instructions after the branch are not up-to-date anymore and have to be discarded.

6.4.5. Exceptions and Interrupts

Exceptions and interrupts behave in most ways like branches. Whenever a specific event occurs, for instance the execution of the software interrupt instruction (`SYCALL`), a branch to a corresponding address (address of the software interrupt vector in this case) takes place. An automatic context change is performed by the system to offer a system state, that does not effect the interrupt program. While exceptions (or processor-internal interrupts) can only occur synchronous to the pipeline / instruction flow, external interrupts can occur at every time. Thus, the interrupt-correlated mode changes and branches need to be synchronized to the pipeline. Therefore, this kind of interrupts can only be processed whenever the current instruction in the EX stage can be interrupted and resumed without any problems. Hence, the instruction must not be a multi-cycle operation nor a branch nor an instruction with a temporal data dependency.

6.5. Interfaces

The Atlas CPU requires to be connected to CPU-external data and instruction memories to operate. While the coprocessor interface is optional, the data and instruction interfaces are mandatory. All interfaces are fully synchronous to the CPU's main clock. The different interfaces and implementation schemes are about to be explained in this chapter.

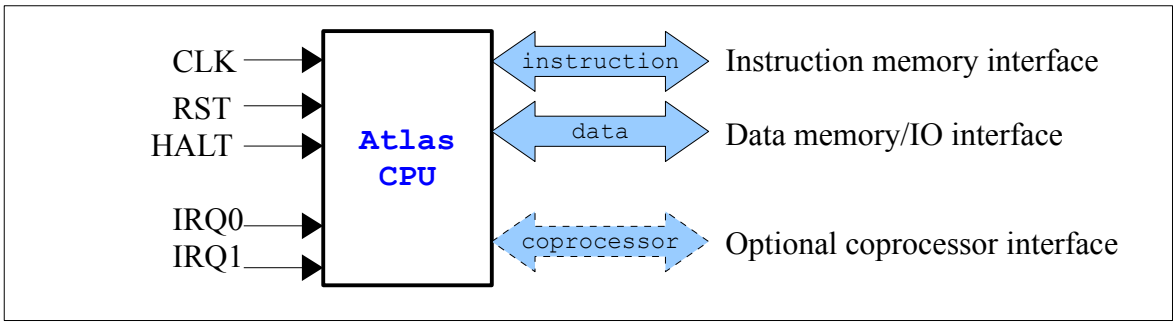

Figure 25: Main interfaces of the Atlas CPU

6.5.1. Memory Interface

This chapter will focus on the “stand alone” (CPU-only) implementation of the Atlas CPU. When in stand-alone mode, the CPU only requires a instruction and data memories together with application-defined hardware modules, connected as coprocessors (exemplary implementation style).

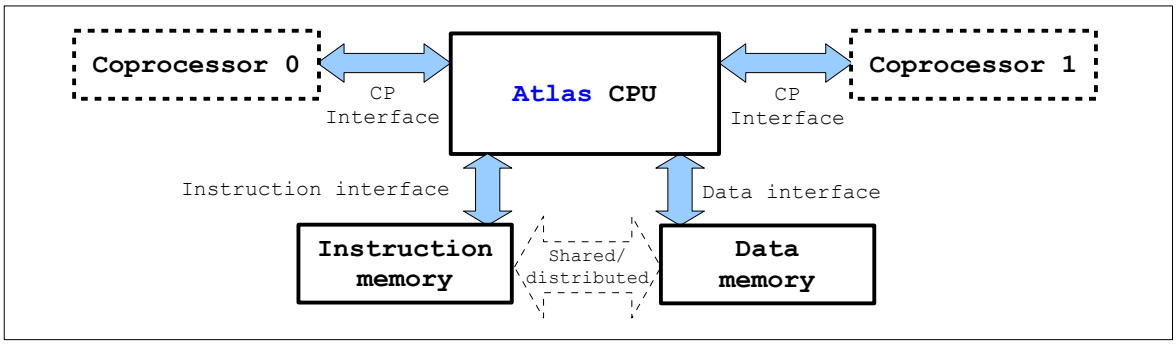

Figure 26: CPU-only implementation block diagram

The data and instruction memories can be separated entities, or a single memory component. Note, that an isolated instruction memory is read-only. Therefore, a shared data and instruction memory needs two read ports, but only one write port. However, separated enable signals for data and instruction output are required for both implementation schemes.

The CPU can either be configured to use different memories or caches for data and instructions (Harvard architecture) or to use a shared memory/cache for data and instructions (Von-Neumann architecture). As an example, this chapter will take a closer look on a Harvard-like implementation.

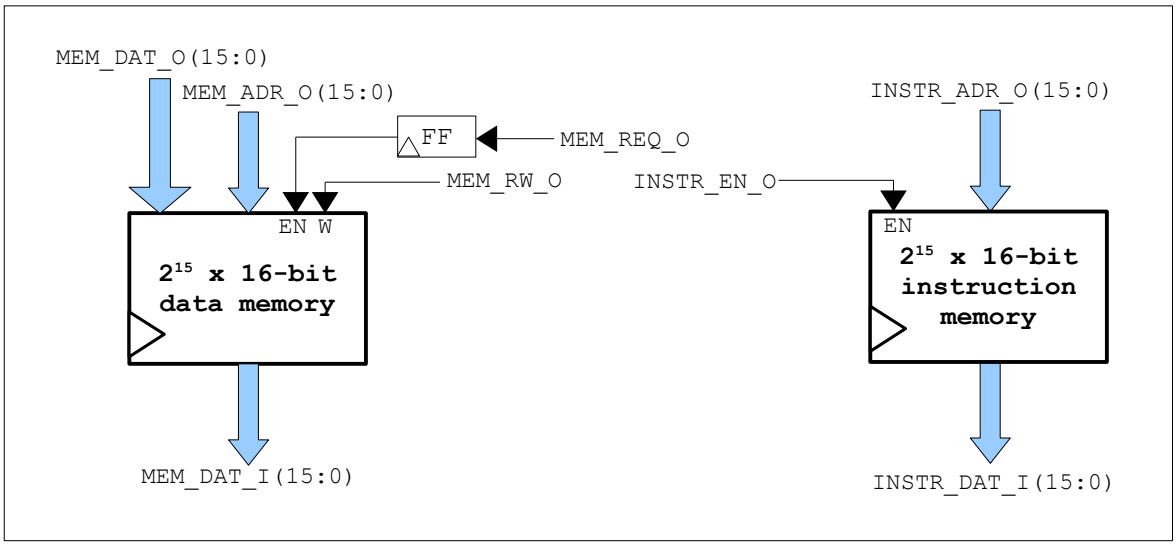

Figure 27: Interface to separated data and instruction memories; signal names correspond to the CPU's instruction/data interface ports

Let's start with the instruction fetch interface. This interface is very simple to implement. It basically consists of the the instruction address (`INSTR_ADR_O`), the instruction word read back (`INSTR_DAT_I`) and an enable signal (`INSTR_EN_O`). The instruction address outputs the current value of the CPU's program counter. On every rising edge of the core clock, the instruction memory outputs the instruction word to the instruction word read back line corresponding to the applied instruction address. Whenever the instruction enable line (`INSTR_EN_O`) goes low, the instruction memory is disabled and it has to hold the last instruction word, since this buffer is used as instruction register.

The data interface operates nearly in the same manner. Here, the enable signal (`MEM_REQ_O`) is applied one cycle before the actual memory access (data and address) takes place. Therefore, it has to be buffered in a flip flop (on the rising edge of the CPU clock) to create the necessary delay. This behavior can be used to tell a memory management system in advance, that the core requests access to the memory. Thus, the delayed enable signal triggers the operation of the memory. Just like the instruction memory, the data memory has to keep the last data output if the enable signal goes low again. Corresponding to the read/write select signal (`MEM_RW_O`), data is stored to the memory (r/w = '1') or read from the memory (r/w = '0'). The access address is presented by the address output port (`MEM_ADR_O`). The store-data comes from the memory write data port (`MEM_DAT_O`). Read-back data from the memory is applied on the rising edge of the CPU clock to the read data port (`MEM_DAT_I`).

6.5.2. Wishbone Interface

For many applications, the direct connection of the CPU to data/instruction memory/memories might be sufficient (CPU-only implementation), but however, many applications require more accessible memory and also some kind of integrated bus to communicate with other SoC modules (like timers, memories, interfaces, other processors, …). The Atlas processor implementation features a Wishbone-compatible bus interface to access other system components via an on chip network fabric (a copy of the Wishbone specifications can be found in the *core/doc* folder).
To allow an efficient use of the bus system, a shared instruction and data cache is connected to the Wishbone bus interface. Furthermore, a memory management unit (MMU) is inserted to extend the accessible memory area to up to 4GB. Of course, the MMU can be bypassed (and therefore removed from the design) when 64kB of addressable memory/IO space within the Wishbone network is sufficient.

The basic structure of the Atlas processor is shown in the figure below. By default, no user coprocessor is implemented within the Atlas processor.

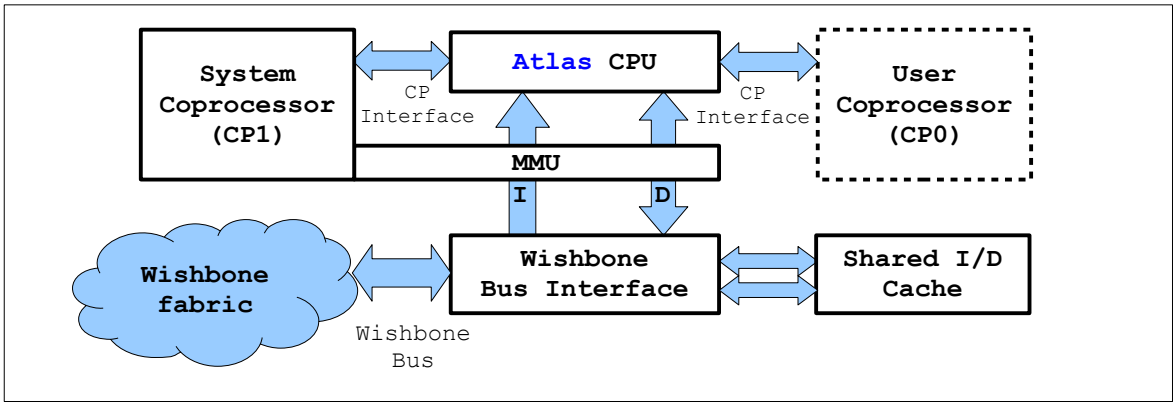

Figure 28: Atlas Processor block diagram

6.5.3. Coprocessor Interface

The coprocessor interface is dedicated to connected up to two external coprocessors (abbreviated as CP) directly to the Atlas CPU without the need of coupling them via some kind of system bus. This feature allows to create a small microprocessor system with two tightly coupled processing devices. The data communication between CPU and a CP is based on direct register transfers between the two entities. Furthermore, direct data manipulation operations specifying two registers of the CP and a command are also implemented. For more information about the transfer and processing instructions, refer to the coprocessor instruction references.

The signal names and their functionality of the Atlas CPU coprocessor interface port are shown in the table below. All CPU output signals of the coprocessor interface are connected to both coprocessors, except for the two enable signals.

Signal name	Size (bit)	CPU-side direction	Unicast/ Broadcast	Function
USR_CP_EN_O	1	out	unicast	Coprocessor 0 (user coprocessor) chip select (active high)
SYS_CP_EN_O	1	out	unicast	Coprocessor 1 (system coprocessor) chip select (active high)
CP_OP_O	1	out	broadcast	Data transfer ('1') or CP data processing ('0') operation
CP_RW_O	1	out	broadcast	Write data to CP ('1') or read data from CP ('0')
CP_CMD_O	9	out	broadcast	Command interface Bit 2 downto 0: *Direct output of the CMD bit-filed of CP instruction* Bit 5 downto 3: *CP operand B address* Bit 8 downto 6: *CP operand A / destination address*
CP_DAT_O	16	out	broadcast	Write data to both coprocessors
CP_DAT_I	16	in	-	OR-ed read data of both coprocessors

Table 18: Coprocessor interface port of the Atlas CPU

All “broadcast signals” are connected to both coprocessor, only the “unicast signals” are private signals for each coprocessor. The data outputs of both coprocessor have to be logically OR-ed to generate the actual data read-back for the CPU's coprocessors read-back port (CP_DAT_I). Because of this read-back style, the coprocessor designer has to ensure, that each coprocessor disables it's data output (set to 0x”0000”) when not select via the USR_CP_EN_O or SYS_CP_EN_O signal, respectively.

Since the data output bus of the coprocessor(s) might be part of the CPU's critical path, which is the execution stage (EX) plus it's forwarding paths, I recommend the use of a buffered coprocessor read-back technique. Therefore, the coprocessor register address, that defines the actual register to be read, should be used to select the corresponding data source already within the EX stage. Thus, the data read from the coprocessor's register file must be buffered with a simple register to have valid read-data in the MA stage. The following figure explains this concept, enhancing the focus on a one coprocessor.

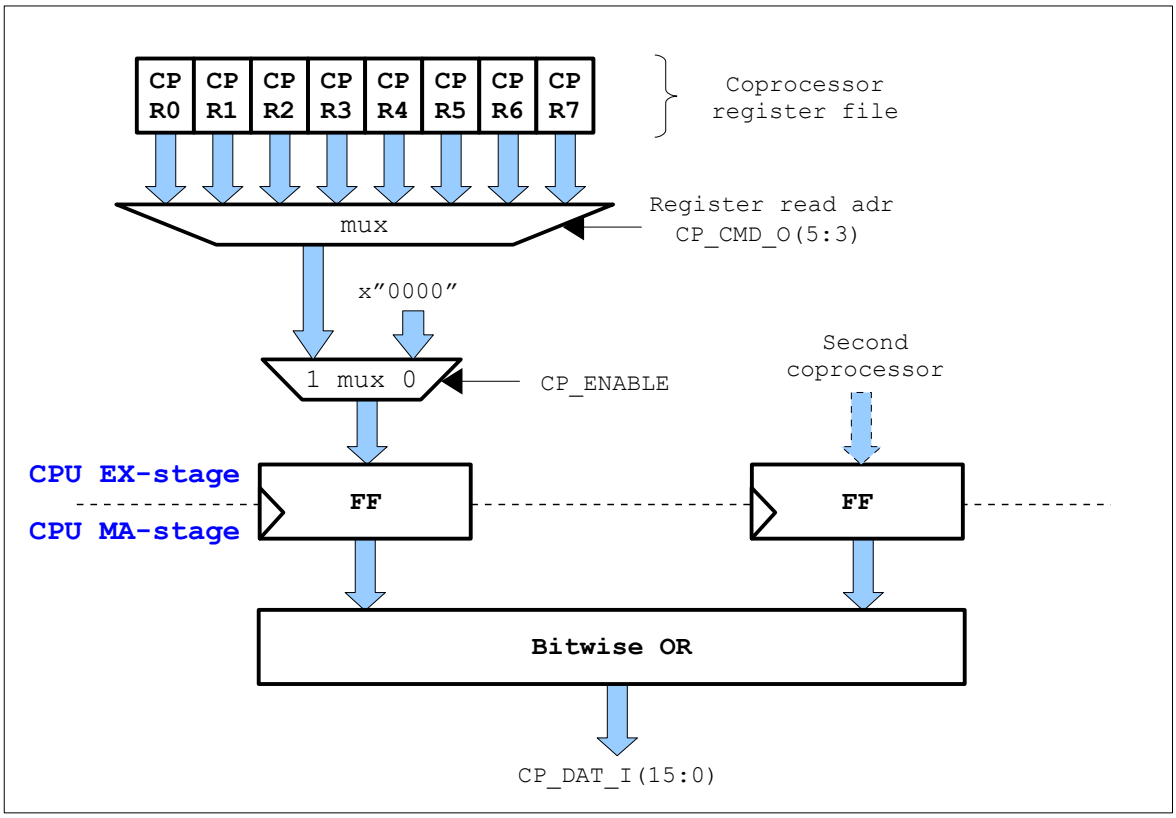

Figure 29: Buffering the data read-back of a coprocessor to shorten critical path

6.6. System Coprocessor (MMU)

By default, the Atlas processor features a memory management unit (MMU). This MMU, implemented as system coprocessor (coprocessor #1), enables the user to access a memory/IO space of up to 2^{32} bytes or 2^{31} words (4GB), respectively. Therefore, the actual data and instruction addresses from the CPU, which are 16-bit wide, are concatenated with another 16 bit, determining the accessible data and instruction page, to create a 32-bit wide address.

6.6.1. Theory of Operation

The complete accessible data space of 2^{32} byte is separated into 2^{16} "pages" of 2^{16} byte each. The actual page is selected via the most significant 16 bits of the final address. These page address bits are taken from page registers, where unique register for instruction and data page access for both operating modes exist. Together with the data and instruction address bits of the CPU, that present the least significant 16 bits of the final address, the final address is constructed. Since the MMU is aware of the current CPU operating mode, an automatic switch between the user and system mode page register is implemented.

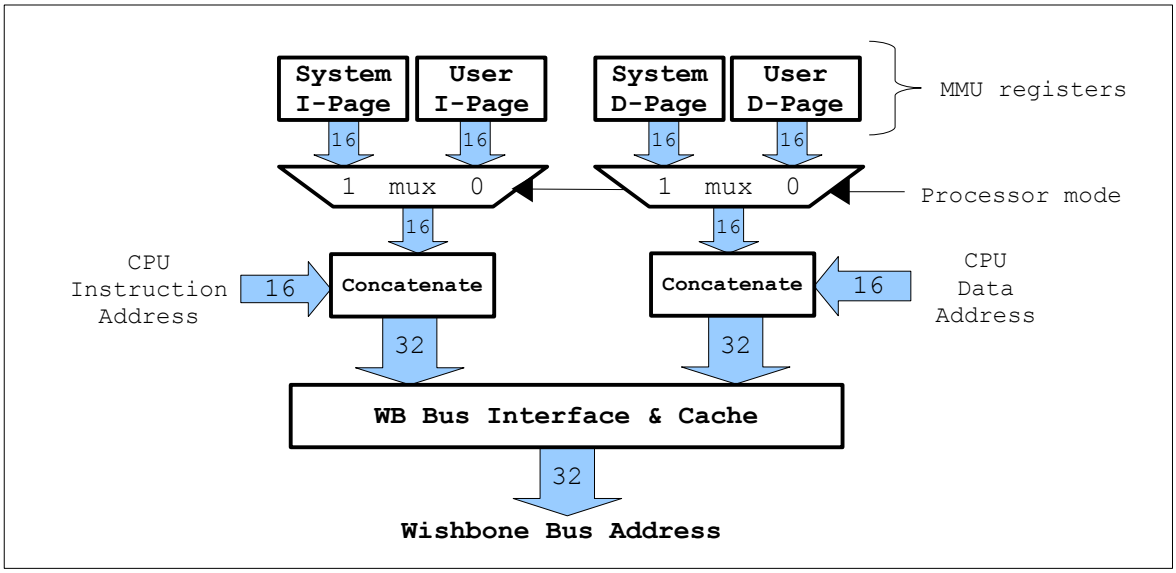

Figure 30: MMU address generation block diagram; the numbers in the arrows refer to the address widths

Whenever an interrupt or exception occurs, the system I- and D-page registers are automatically set to zero (x"0000"), an immediate zero-page output of the MMU is generated and the last accessed I- and D-pages (the last value of the corresponding system page registers) are store to the I- and D-link register. This makes it easy to restore the last accessed pages after an interrupt has been processed. The data of the D/I-link registers have just to be copied back to the system page registers when the interrupt handler has finished.

6.6.2. Interface

The MMU is accessed via the coprocessor interface and the corresponding data transfer and data processing instructions. Remember, that the system coprocessor can only be accessed in system mode. A list of all accessible coprocessor registers is shown below.

Register	Name	Function
c0	MMU_CTRL	MMU control register, see table below
c1	MMU_SCRATCH	Scratch register; free to use
c2	MMU_SYS_I_PAGE	I-Page for system mode
c3	MMU_SYS_D_PAGE	D-Page for system mode
c4	MMU_USR_I_PAGE	I-Page for user mode
c5	MMU_USR_D_PAGE	D-Page for user mode
c6	MMU_I_PAGE-LINK	Last accessed I-Page after an interrupt/exception has occurred
c7	MMU_D_PAGE_LINK	Last accessed D-Page after an interrupt/exception has occurred

Table 19: MMU register map

The functionality of the different control register (MMU_CTRL) bits is explained in the following table.

Bit	Name	R/W	Function
0	cflush	W	Flush cache to sync memory with cache when set to '1'
1	cclr	W	Invalidate all cache entries (reload cache) when set to '1'
2	dda	R/W	Enable direct data access (bypass cache for data requests) when set to '1'
3	csync	R	Cache is sync to memory when '1'
4	bus_err	R/W	Bus error has occurred when '1', write a '1' to acknowledge
5..15	-	*R/W*	*Reserved, should not be altered*

Table 20: MMU control register (MMU_CTRL) bits

Since reading, altering and writing back of the control register takes at least three cycles to complete, most of the relevant control functions can also be controlled by using the coprocessor data processing instruction including a specific command code to trigger the configuration bits of the control register. All implemented control commands are listed in the table below. Using unimplemented commands will not have an effect on the MMU. Furthermore, commands will not have an effect when applied during coprocessor data transfers.

Command	Name	ASM Usage Example	Function
#0	flush_cache	`CDP #1, c0, c0, #0`	Flush cache to synchronize memory with cache
#1	clear_cache	`CDP #1, c0, c0, #1`	Invalidate all cache entries (reload cache)
#2	en_dir_acc	`CDP #1, c0, c0, #2`	Enable direct data access (bypass cache for data requests)
#3	dis_dir_acc	`CDP #1, c0, c0, #3`	Disable direct data access
#4	ack_mmu_irq	`CDP #1, c0, c0, #4`	Acknowledge MMU interrupt (e.g. bus error)
#5	link_copy	`CDP #1, c0, c0, #5`	Copy link registers back to system page registers
#6, #7	*reserved*	-	Currently not implemented, execution has no effect

Table 21: Currently implemented MMU commands

6.6.3. MMU Interrupt

The MMU also features an interrupt request output (connected to IRQ1 of the CPU) to indicate a bus error. These bus errors appear, whenever the bus interface does not receive an acknowledge from an accessed address in the Wishbone network within a specific time (*max_bus_latency_c* constant in the Atlas VHDL package file).

6.6.4. Update-Latency

It takes two cycles until a new value written to a page register has an effect to the address output of the MMU. This ensures an executed branch directly after the MMU register updated instruction will result in a branch to the correct position. There must be no delays between the MMU i-base write instruction and the actual branch instruction to keep the addresses synchronized.

6.6.5. ASM Usage Examples

Below, some examples of how to use the MMU are presented.

```
      ;Absolute branch to label "destination" (within 32-bit address space)
      ;executed in system mode

      LDIL R1, #XLOW[destination]      ; load high address (upper 16 bit) of
      LDIH R1, #XHIGH[destination]     ; label "destination" address
      LDIL R0, #LOW[destination]       ; load lower address (lower 16 bit) of
      LDIH R0, #HIGH[destination]      ; label "destination" address

      ;there must be no delay between the next two instructions!
      MCR  #1, C2, R1, #0              ; load high address to MMU's sys-i-page register
      GTX  R0                          ; load low address to PC → finalize branch
```

```
      ;Flush cache and wait until cache is synchronous to memory
      ;executed in system mode
      ;(bit 3 of the MMU's control register indicates a synchronous cache)

      CDP #1, C0, C0, #0         ; directly execute MMU'S "flush cache" command
                                 ; commands (#0) have no effect during data transfers
get_status:
      MRC  #1, R0, C0, #0        ; read MMU status register to r0
      STBI R0, #3                ; store inverted bit 3 of r0 to T-flag to test it
      BTS  get_status            ; go to beginning of loop when original sync-flag is '0'

cache_sync:                      ; cache is sync when arriving here
```

6.7. Main Control Bus

The following table shows the location and signal names of the main system control bus. All primary control signals, which are emerging from the opcode decoder, are forwarded throughout the complete pipeline are combined within this bus. Even if not all signals are used in every single pipeline stage, all signal are carried out until the end of the processing pipeline. This helps to keep the architecture flexible for future changes.

Bit #	Signal name	Function
Global Control		
0	ctrl_en_c	A '1' indicates a valid operation within the corresponding pipeline stage
1	ctrl_mcyc_c	Multi-cycle/atomic memory operation in progress, no interrupt possible
Operand A		
2	ctrl_ra_is_pc_c	Operand A is the program counter
3	ctrl_clr_ha_c	Set higher byte of operand A to 0
4	ctrl_clr_la_c	Set lower byte of operand A to 0
5	ctrl_ra_0_c	Operand register A address bit 0
6	ctrl_ra_1_c	Operand register A address bit 1
7	ctrl_ra_2_c	Operand register A address bit 2
8	ctrl_ra_3_c	Operand register A address bit 3, indicating source mode
Operand B		
9	ctrl_rb_is_imm_c	Operand B is an immediate
10	ctrl_rb_0_c	Operand register B address bit 0
11	ctrl_rb_1_c	Operand register B address bit 1
12	ctrl_rb_2_c	Operand register B address bit 2
13	ctrl_rb_3_c	Operand register B address bit 3, indicating source mode
Destination Register		
14	ctrl_rd_wb_c	Enable write-back to register file
15	ctrl_rd_0_c	Destination register address bit 0
16	ctrl_rd_1_c	Destination register address bit 1
17	ctrl_rd_2_c	Destination register address bit 2
18	ctrl_rd_3_c	Destination register address bit 3, indicating destination mode
ALU Control		
19	ctrl_alu_fs_0_c	ALU function select bit 0
20	ctrl_alu_fs_1_c	ALU function select bit 1
21	ctrl_alu_fs_2_c	ALU function select bit 2
22	ctrl_alu_usec_c	Use mode-corresponding carry flag for computation
23	ctrl_alu_usez_c	Use mode-corresponding zero flag for computation
24	ctrl_fupdate_c	Update ALU flags after processing

Bit #	Signal name	Function
Bit Manipulation		
25	ctrl_tf_store_c	Store bit to mode-corresponding transfer flag
26	ctrl_tf_inv_c	Invert bit to be stored to T-flag
27	ctrl_get_par_c	Select operand A's parity as T-flag source
28	ctrl_bit_0_c	Bit index bit 0
29	ctrl_bit_1_c	Bit index bit 1
30	ctrl_bit_2_c	Bit index bit 2
31	ctrl_bit_3_c	Bit index bit 3
System Register Access		
32	ctrl_msr_wr_c	Write access to MSR
33	ctrl_msr_rd_c	Read data from MSR
34	ctrl_pc_wr_c	Write access to PC
Branch/Context Control		
35	ctrl_cond_0_c	Condition code bit 0
36	ctrl_cond_1_c	Condition code bit 1
37	ctrl_cond_2_c	Condition code bit 2
38	ctrl_cond_3_c	Condition code bit 3
39	ctrl_branch_c	Current operation is a branch operation
40	ctrl_link_c	Perform link operation (store return address to LR)
41	ctrl_syscall_c	Current operation is some kind of software interrupt
42	ctrl_ctx_down_c	Switch down to user mode
Data Memory Access		
43	ctrl_mem_acc_c	Perform data memory access
44	ctrl_mem_wr_c	Write ('1') or read ('0') access
45	ctrl_mem_bpba_c	Use bypassed base address
46	ctrl_mem_daa_c	Use delayed base address
Coprocessor Access		
47	ctrl_cp_acc_c	Current operation is a coprocessor operation
48	ctrl_cp_trans_c	Coprocessor data transfer ('1') or coprocessor data processing operation ('0')
49	ctrl_cp_wr_c	Write access to coprocessor
50	ctrl_cp_id_c	Coprocessor ID bit ('1' for coprocessor #1, '0' for coprocessor #0)
MAC Unit		
51	ctrl_use_mac_c	Access the multiply-and-accumulate unit (if implemented)
52	ctrl_load_mac_c	Load an accumulation value to the MAC buffer
53	ctrl_use_offs_c	Use the loaded value to perform the actual MAC operation

Table 22: Processor main control bus

As mentioned before, not all signals are used in all pipeline sages. Therefore, some signals are reused with a different name alias when their original purpose is not relevant for further processing anymore. The table below presents this new signals and the reused original signals.

Signal name	Reused signal	Function
ctrl_wb_en_c	ctrl_rd_wb_c	Valid write back
ctrl_rd_mem_acc_c	ctrl_mem_acc_c	True memory access
ctrl_rd_cp_acc_c	ctrl_cp_acc_c	True coprocessor read access
ctrl_cp_msr_rd_c	ctrl_msr_rd_c	True coprocessor or MSR read access
ctrl_cp_cmd_0_c	ctrl_rb_0_c	Coprocessor command bit 0
ctrl_cp_cmd_1_c	ctrl_rb_1_c	Coprocessor command bit 1
ctrl_cp_cmd_2_c	ctrl_rb_2_c	Coprocessor command bit 2
ctrl_cp_ra_0_c	ctrl_ra_0_c	Coprocessor operand A bit 0
ctrl_cp_ra_1_c	ctrl_ra_1_c	Coprocessor operand A bit 1
ctrl_cp_ra_2_c	ctrl_ra_2_c	Coprocessor operand A bit 2
ctrl_cp_rd_0_c	ctrl_rd_0_c	Coprocessor operand A / destination register bit 0
ctrl_cp_rd_1_c	ctrl_rd_1_c	Coprocessor operand A / destination register bit 1
ctrl_cp_rd_2_c	ctrl_rd_2_c	Coprocessor operand A / destination register bit 2
ctrl_re_xint_c	ctrl_rb_1_c	Re-enable global external interrupt flag
ctrl_msr_am_0_c	ctrl_ra_1_c	MSR access mode option bit 0
ctrl_msr_am_1_c	ctrl_ra_2_c	MSR access mode option bit 1

Table 23: Processor main control bus, signal reuse during pipeline process

7. Simulation

For easy simulation and debugging, simple testbenches for both of the implementation schemes are included within the Atlas project. The necessary testbenches as well as additional simulation components can be found in the *core/sim* folder. When you are using Xilinx ISIM © for simulation, you can use the predefined waveform configuration files from the *core/sim/isim_wave* folder. This waveforms contain already all relevant simulation and debugging signals.

The following table presents the file hierarchy for simulating the two designs.

	ATLAS_MICRO	ATLAS_PROCESSOR
Testbench:	sim/testbench_atlas_micro/ *micro_tb.vhd*	sim/testbench_processor_waveform/ *processor_tb.vhd*
Xilinx ISIM demo waveform configuration:	sim/isim_wave/ *MICRO_WAVE.wcfg*	sim/isim_wave/ *PROCESSOR_WAVE.wcfg*
Design top entity:	rtl/*ATLAS_MICRO.vhd*	rtl/*ATLAS_PROCESSOR.vhd*
File dependencies:	- rtl/*ALU.vhd* - rtl/*ATLAS_CORE.vhd* - rtl/*ATLAS_pkg.vhd* - rtl/*CTRL.vhd* - rtl/*MEM_ACC.vhd* - rtl/*OP_DEC.vhd* - rtl/*REG_FILE.vhd* - rtl/*SYS_REG.vhd* - rtl/*WB_UNIT.vhd*	- sim/testbench_processor_waveform/ *TEST_MEM.vhd* - rtl/*ALU.vhd* - rtl/*ATLAS_CORE.vhd* - rtl/*ATLAS_pkg.vhd* - rtl/*BUS_INTERFACE.vhd* - rtl/*CTRL.vhd* - rtl/*MEM_ACC.vhd* - rtl/*MMU.vhd* - rtl/*OP_DEC.vhd* - rtl/*REG_FILE.vhd* - rtl/*SYS_REG.vhd* - rtl/*WB_UNIT.vhd*

Table 24: Simulation file guideline

7.1. Atlas Processor Simulation

To easily evaluate and simulate a program for the Atlas processor, the "*processor_tb.vhd*" testbench in the *core/sim/testbench_processor_system* folder can be used. This testbench includes the top entity of the Atlas processor, together with a Wishbone-compatible demo memory component ("*TEST_MEM.vhd*", in the same folder as the testbench), which is directly connected to Wishbone interface of the processor (without any interconnection fabric).

To include the assembled program into the simulation environment, the content of the init.vhd file, generated by the Atlas assembler, has to be copied to the memory initialization area of the corresponding memory VHDL component. The following code presents a cutout of the "*TEST_MEM.vhd*" component.

```
--- Memory Type ---
type MEM_FILE_TYPE is array (0 to MEM_SIZE-1) of std_logic_vector(data_width_c-1 downto 0);

--- INIT MEMORY IMAGE ---
-------------------------------------------------------
signal MEM_FILE : MEM_FILE_TYPE :=
(
       -- This is where you have to place the "init.vhd" file content --
);
-------------------------------------------------------
```

Open the "*init-vhd*" file from the *asm* folder, copy **all** of the content and paste it between the two brackets of the *MEM_FILE* signal initialization (right there, where the red note is). Afterwards, the cutout of the *TEST_MEM.vhd* file should look somehow like the following example.

```
--- Memory Type ---
type MEM_FILE_TYPE is array (0 to MEM_SIZE-1) of std_logic_vector(data_width_c-1 downto 0);

--- INIT MEMORY IMAGE ---
-------------------------------------------------------
signal MEM_FILE : MEM_FILE_TYPE :=
(
       000000 => x"bc1b", -- B
       000001 => x"bc13", -- B
       000002 => x"bc12", -- B
       000003 => x"bc01", -- B
       000004 => x"07f2", -- DEC
       000005 => x"5078", -- LDR
       000006 => x"2891", -- CLR
       000007 => x"ccfc", -- LDIH
       000008 => x"3001", -- BIC
       000009 => x"c644", -- LDIL
       000010 => x"ca00", -- LDIH
       000011 => x"50c8", -- LDR
       000012 => x"54c8", -- STR
       000013 => x"e400", -- CDP
       000014 => x"ec00", -- MRC
       000015 => x"dc03", -- STB
       others => x"0000"  -- NOP
);
-------------------------------------------------------
```

7.2. Atlas Micro Simulation

Simulating a program on the Atlas Micro is very similar to the simulation of the Atlas Processor. The testbench "*micro_tb.vhd*" of this setup is located in the *core/sim/testbench_atlas_micro* folder. The Atlas Micro design includes the CPU core together with a shared or distributed data and program memory. Since the Atlas CPU and the assembler assume a von-Neumann architecture, I strongly recommend the use of a shared instruction and data memory (default).

The following snippet of the "*micro_tb.vhd*" testbench will create a 1024 byte wide shared instruction/data memory. Make sure, the boot address of the CPU is set to zero to start executing your program right at the beginning.

```
generic map (
               -- Configuration --
               MEM_SIZE_G      => 1024,        -- internal memory size in bytes
               SHARED_MEM_G    => TRUE,        -- shared data/instruction memories
               BOOT_ADDRESS_G  => x"0000"      -- boot address
             )
```

To include the assembled program into the simulation environment, the content of the init.vhd file, generated by the Atlas assembler, has to be copied to the memory initialization area of the corresponding memory VHDL component. The following code presents a cutout of the "*ATLAS_MICRO.vhd*" component.

```
-- Memory Type --
type MEM_FILE_T is array (0 to ((MEM_SIZE_G/2)-1)) of std_logic_vector(data_width_c-1 downto 0);

-- INIT MEMORY IMAGE X --
-- Shared memory: Use this memory image for initializing the DATA and INSTRUCTION memory
-- Separated memories: Use this memory image for initializing the INSTRUCTION memory only
-------------------------------------------------------------------------------------------
signal MEM_FILE_X : MEM_FILE_T :=
(
     others => x"0000" -- Place here the init.vhd file content
);
-------------------------------------------------------------------------------------------

-- INIT MEMORY IMAGE Y --
-- Shared memory: For this case, this component is not used!
-- Separated memories: Use this memory image for optionally initializing the DATA memory
-------------------------------------------------------------------------------------------
signal MEM_FILE_Y : MEM_FILE_T :=
(
     others => x"0000" -- Place here the init.vhd file content
);
-------------------------------------------------------------------------------------------
```

It's the same procedure: Open the "*init-vhd*" file from the *asm* folder, copy **all** of the content and paste it between the two brackets of the *MEM_FILE_X* signal initialization (right there, where the red note is). This signal is used to initialize the data and program memory. Afterwards, the cutout of the ATLAS_MICRO.*vhd* file should look somehow like the following example.

```
-- Memory Type --
type MEM_FILE_T is array (0 to ((MEM_SIZE_G/2)-1)) of std_logic_vector(data_width_c-1 downto 0);

-- INIT MEMORY IMAGE X --
-- Shared memory: Use this memory image for initializing the DATA and INSTRUCTION memory
-- Separated memories: Use this memory image for initializing the INSTRUCTION memory only
-------------------------------------------------------------------------------------------
signal MEM_FILE_X : MEM_FILE_T :=
(
     000000 => x"bc1b", -- B
     000001 => x"bc13", -- B
     000002 => x"bc12", -- B
     000003 => x"bc01", -- B
     000004 => x"07f2", -- DEC
     000005 => x"5078", -- LDR
     000006 => x"2891", -- CLR
     000007 => x"ccfc", -- LDIH
     000008 => x"3001", -- BIC
     000009 => x"c644", -- LDIL
     000010 => x"ca00", -- LDIH
     000011 => x"50c8", -- LDR
     000012 => x"54c8", -- STR
     000013 => x"e400", -- CDP
     000014 => x"ec00", -- MRC
     000015 => x"dc03", -- STB
     others => x"0000"  -- NOP
);
-------------------------------------------------------------------------------------------

-- INIT MEMORY IMAGE Y --
-- Shared memory: For this case, this component is not used!
-- Separated memories: Use this memory image for optionally initializing the DATA memory
-------------------------------------------------------------------------------------------
signal MEM_FILE_Y : MEM_FILE_T :=
(
     others => x"0000" -- not used
);
-------------------------------------------------------------------------------------------
```