Atlas 2k Processor

by Stephan Nolting

WISHBONE COMPATIBLE

Proprietary Notice

License

Disclaimer

This document was created using OpenOffice.

Table of Content

1. Introduction

Welcome to the **Atlas 2k Processor** project!

I've come a long way working with famous processor architectures like ARM, DLX, MIPS, AVR and some - let's call them more 'exotic' - ASIP cores. And during my work with them, I gathered a lot of ideas what a cool processor architecture might look like. But since no processor featured all those ideas, I decided to create my own one.

My first attempt was the STORM Core. It was more like a personal research project for me to get into the basics of processor architecture. But at least for my taste, the STORM Core turned out to be way too clumsy and complicated to use for embedded projects. So I took my positive experiences from that project, combined them with many ideas and approaches and glued them all together to create a CPU, that really measures up to all my – and hopefully someone else's – expectations.

Of course, the specifications of the Atlas 2k processor aren't carved in stone yet... So if you have any cool ideas for it, feel free to drop me a line.

However, have fun with the Atlas 2k processor! ;)

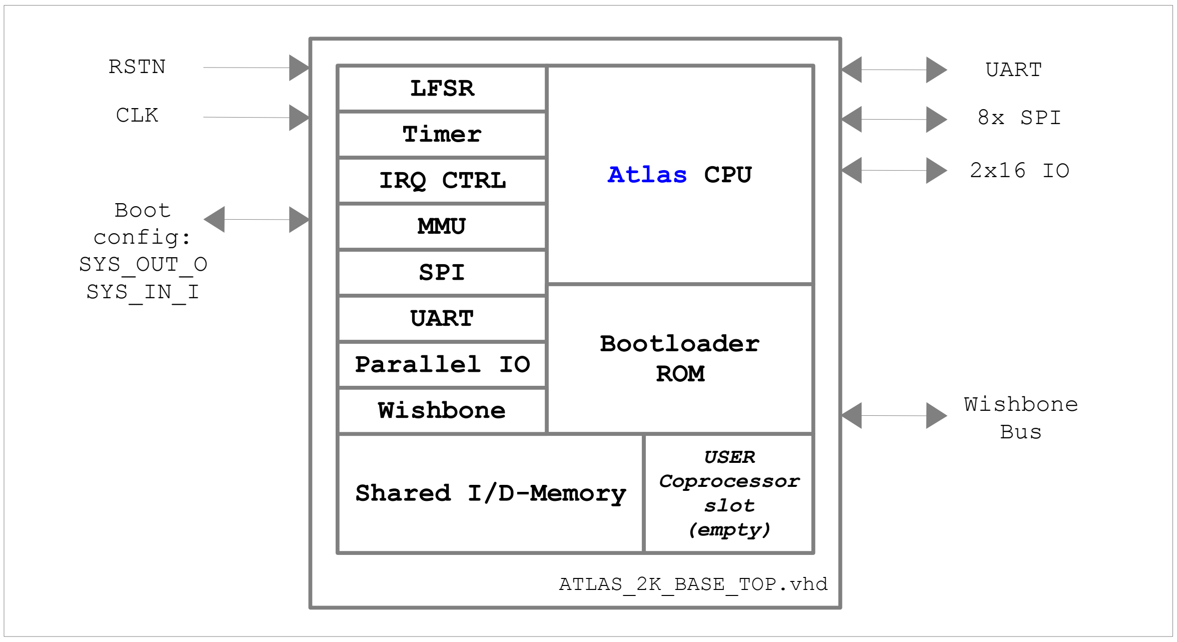


Figure 1: Atlas 2k Processor - Base Setup

NOTE: Above, you see the block diagram of Atlas 2k base setup. If you don't want to take the risk of going crazy by implementing the memory interface by yourself, you should really use this setup instead of the processor alone. This setup already includes the processor coupled with a compatible shared I/D-memory and it is the perfect starting point for your embedded SoC.

1.1. Processor Features

- ✔ 16-bit RISC open source soft-core processor with very small
- ✔ Very small outline
- ✔ Completely described in behavioral, platform-independent **VHDL**
- ✔ Pipelined instruction execution in 5 stages
- ✔ Single cycle execution of all instructions (except for branches and multi-cycle operations)
- ✔ 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 in unprivileged-mode
- ✔ Software traps (system call and unauthorized coprocessor/register bank/ MSR access)
- ✔ Power-saving sleep mode
- ✔ Interrupt pin for critical applications; signal is directly forwarded into the CPU
- ✔ Simple memory interface
- ✔ Integrated bootloader supporting several functions and boot options:
  - ➔ Boot from UART / SPI EEPROM / internal memory / Wishbone device
  - ➔ RAM dump / Wishbone dump / program SPI EEPROM
- ✔ Interface for external "user" coprocessor to extend the processor's functionality and processing speed
- ✔ Integrated system coprocessor:
  - ➔ High precision timer (32-bit)
  - ➔ Memory management unit (supports paging)
  - ➔ Flexible linear-feedback shift register for pseudo-random data
  - ➔ Interrupt controller for up to 8 channels (1 channel for general purpose)
  - ➔ 16+8 bit input and 16+8 bit output parallel IO port
  - ➔ General purpose SPI communication controller with 8 individual ports
  - ➔ Configurable universal asynchronous receiver/transmitter (UART)
  - ➔ Wishbone Bus Adapter (32-bit address, 16-bit data, supports pipelined burst transfers)

1.2. Project Folder Structure

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

- **asm:** This folder contains the Atlas assembler program. The C source files of can be found in the sub-folder "*src*".
- **doc:** The Atlas 2k data sheet (this file) and a copy of the implemented Wishbone bus specifications can be found here.
- **rtl:** All rtl files of the processor are located here.
- **sim:** The sim folder contains a testbench for the Atlas 2k processor base setup and a default Xilinx ISIM© waveform configuration.
- **software:** The software folder contains the assembler source file of the bootloader as well as some example programs.
- **syn:** This folder can be used as project folder for your EDA tool of choice.

1.3. VHDL File Hierarchy

All necessary hardware description files are located in the project's *rtl* folder. The top entity of the processor is "**ATLAS_2K_TOP.vhd**", the top entity of the system setup is "**ATLAS_2K_BASE_TOP.vhd**".

`ATLAS_2K_BASE_TOP.vhd`	**→ Basic system setup top entity**
`- INT_RAM.vhd`	→ Internal memory component
`- ATLAS_2K_TOP.vhd`	**→ Atlas 2k Processor top entity**
`  - BOOT_MEM.vhd`	→ Bootloader memory
`  - MEM_GATE.vhd`	→ Memory gateway
`  - ATLAS_CPU.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
`  - SYSTEM_CP.vhd`	→ Internal system coprocessor
`    - COM_0_CORE.vhd`	→ Communication controller
`    - COM_1_CORE.vhd`	→ Bus adapter (Wishbone)
`    - SYS_0_CORE.vhd`	→ System controller 0
`    - SYS_1_CORE.vhd`	→ System controller 1

Table 1: Project's VHDL file hierarchy

2. Top Entity Signal Description

This chapter give a brief overview of the signal ports of the basic setup's top entity (**ATLAS_2K_BASE_TOP.vhd**) and the processor's top entity (**ATLAS_2K_TOP.vhd**). The type of all signals/generics is **std_logic** or **std_logic_vector**, respectively.

2.1. Atlas 2k Basic Setup Top Entity

When using this setup as initial starting point – which I encourage you to do ;) -, do not forget to set the user configuration constants in the ATLAS_2K_TOP.vhd file.

Signal name	Width (#bits)	Dir	Function
Global Control			
CLK_I	1	IN	Global clock signal, all registers trigger on the rising edge
RSTN_I	1	IN	Global reset signal, synchronized to CLK_I and **low-active**
IO Interface			
UART_RXD_I	1	IN	UART receiver input
UART_TXD_O	1	OUT	UART transmitter output
SPI_MOSI_O	8	OUT	8 SPI data transmitter channel outputs
SPI_MISO_I	8	IN	8 SPI receiver channel inputs
SPI_SCK_O	8	OUT	8 SPI clock line outputs
SPI_CS_O	8	OUT	8 SPI chip select lines (low active)
PIO_OUT_O	16	OUT	16 parallel output ports
PIO_IN_I	16	IN	16 parallel input ports
SYS_OUT_O	8	OUT	Bootloader/system output port (bootloader status)
SYS_IN_I	8	IN	Bootloader/system input port (boot configuration)
Wishbone Bus			
WB_CLK_O	1	OUT	Main bus clock (same as CLK_I)
WB_RST_O	1	OUT	Bus reset, synchronous, high-active
WB_ADR_O	32	OUT	Address output
WB_SEL_O	2	OUT	Byte select (“always”11” → full word transfer)
WB_DATA_O	16	OUT	Write data output
WB_DATA_I	16	IN	Read data input
WB_WE_O	1	OUT	Write enable signal
WB_CYC_O	1	OUT	Valid cycle signal
WB_STB_O	1	OUT	Address/data strobe
WB_ACK_I	1	IN	Acknowledge input
WB_ERR_I	1	IN	Bus error

Table 2: ATLAS 2K BASE SETUP - top entity interface ports

NOTE: The Atlas 2k processor as well as the base setup provide a LOT of IO pins for parallel input/output and SPI ports. If you do not need all of them, you should not remove them from the top entity and the corresponding sub modules. For compatibility reasons with future updates of them, you should instantiate the top entity of the processor or the base setup into another file, where you can assign all the desired IO ports and tie the unused inputs to low and leave all unused outputs 'open'.

2.2. Atlas 2k Processor Top Entity

Signal name	Width (#bits)	Dir	Function
Configuration generics			
CLK_SPEED_G	32	-	Main clock speed in Hz
Global Control			
CLK_I	1	IN	Global clock signal, all registers trigger on the rising edge
RST_I	1	IN	Global reset signal, synchronized to CLK_I and **high-active**
CE_I	1	IN	Global clock enable, high-active, should only change on falling edge of CLK_I
Coprocessor Interface			
CP_EN_O	1	OUT	Coprocessor access enable
CP_ICEEN_O	1	OUT	Coprocessor interface clock enable
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
CP_CMD_O	9	OUT	Coprocessor command, consisting of source/destination register and operation command
CP_DAT_O	16	OUT	Coprocessor write data
CP_DAT_I	16	IN	Coprocessor read data
Memory interface			
MEM_I_PAGE_O	16	OUT	Instruction memory page
MEM_I_ADR_O	16	OUT	Instruction memory address
MEM_I_EN_O	1	OUT	Instruction output enable
MEM_I_DAT_I	16	IN	Instruction word output
MEM_D_EN_O	1	OUT	Data memory enable
MEM_D_RW_O	1	OUT	Data memory read('0')/write access('1')
MEM_D_PAGE_O	16	OUT	Data memory page
MEM_D_ADR_O	16	OUT	Data memory address
MEM_D_DAT_O	16	OUT	Data memory write data
MEM_D_DAT_I	16	IN	Data memory read data
CRITICAL_IRQ_I	1	IN	Critical interrupt request
IO Interface			
UART_RXD_I	1	IN	UART receiver input
UART_TXD_O	1	OUT	UART transmitter output
SPI_MOSI_O	8	OUT	8 SPI data transmitter channel outputs
SPI_MISO_I	8	IN	8 SPI receiver channel inputs
SPI_SCK_O	8	OUT	8 SPI clock line outputs
SPI_CS_O	8	OUT	8 SPI chip select lines (low active)
PIO_OUT_O	16	OUT	16 parallel output ports
PIO_IN_I	16	IN	16 parallel input ports
SYS_OUT_O	8	OUT	Bootloader/system output port (status)
SYS_IN_I	8	IN	Bootloader/system input port (boot configuration)
IRQ_I	1	IN	Interrupt request input → connected to internal IRQ controller channel #7
Wishbone Interface			
WB_CLK_O	1	OUT	Main bus clock (same as CLK_I)
WB_RST_O	1	OUT	Bus reset, synchronous, high-active
WB_ADR_O	32	OUT	Address output

Signal name	Width (#bits)	Dir	Function
WB_SEL_O	2	OUT	Byte select (“always”11” → full word transfer)
WB_DATA_O	16	OUT	Write data output
WB_DATA_I	16	IN	Read data input
WB_WE_O	1	OUT	Write enable signal
WB_CYC_O	1	OUT	Valid cycle signal
WB_STB_O	1	OUT	Address/data strobe
WB_ACK_I	1	IN	Acknowledge input
WB_ERR_I	1	IN	Bus error

Table 3: ATLAS 2K's top entity interface ports

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.

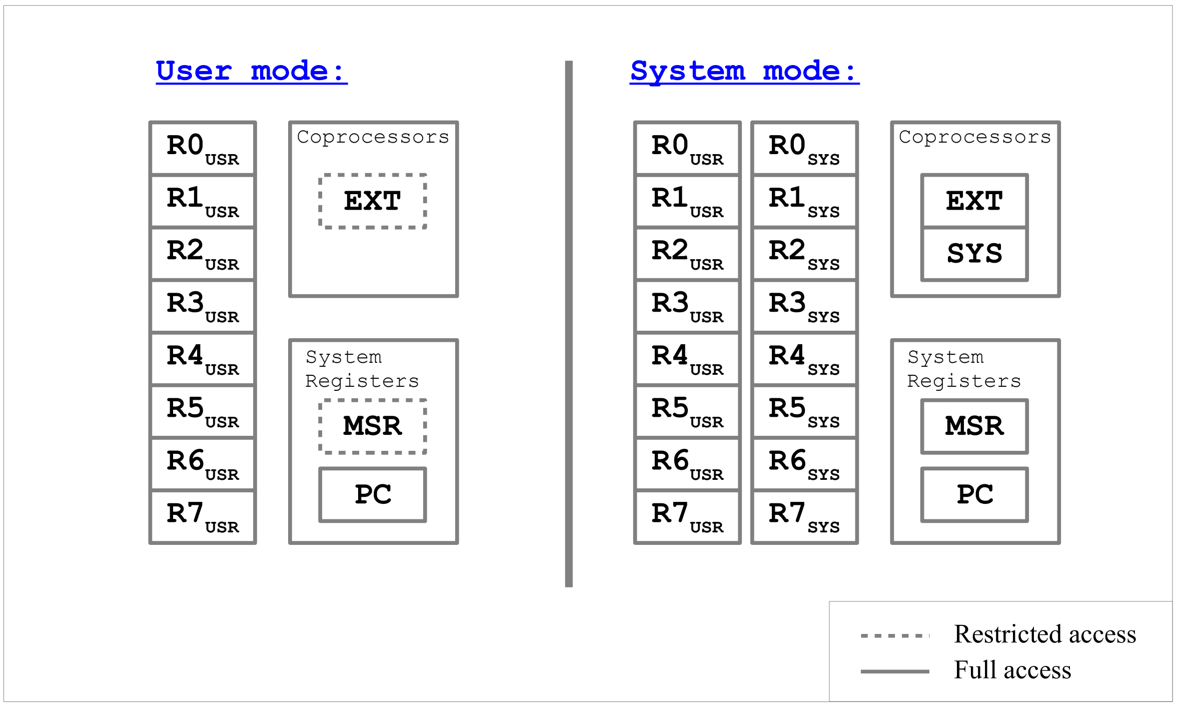


Figure 2: Operation modes and accessible registers

3.1. Operating Modes

Two different operation modes are supported by the Atlas CPU. 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 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 an interrupt trap, called the command error trap. This 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]
1 (highest)	Hardware reset	No	x"0000"
2	Critical error IRQ (EXT_INT_0 of CPU) (external CRITICAL_IRQ_I pin)	Yes	x"0002"
3	Interrupt controller IRQ (EXT_INT_1 of CPU)	Yes	x"0004"
4	Command error trap (undefined instruction or coprocessor / register bank / MSR access violation)	No	x"0006"
5 (lowest)	Software interrupt trap (SYSCALL instruction)	No	x"0008"

Table 4: Interrupt vector addresses (hexadecimal) and priority list

NOTE: The shown interrupts are the interrupts sources of the CPU only. The ATLAS 2K supports additional interrupts, which are processed by the interrupt controller and forwarded to **EXT_INT_1**. For more information about the IRQ signals of the interrupt controller, see the chapter about the internal coprocessor.

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. Also, the global external interrupt flag in the MSR is automatically cleared whenever a valid interrupt or exception is executed. This prevent the interrupt handler to be interrupted again by external interrupt requests. The external interrupt enable flag can be re-set by specific handler termination instructions (like `RETI`).

NOTE: The execution of all instructions – even of the multi-cycle memory access operations - is atomic. Thus, the complete 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.

1 Addresses correspond to the default setup of the CPU "byte-addressing mode".

3.4. Coprocessors

The Atlas CPU supports up to two coprocessors, where coprocessor #1 is already integrated into the Atlas 2K as system coprocessor. This coprocessor includes often used devices like a timer, UART, IO ports, an interrupt controller, a memory management unit and a Wishbone-compatible bus control unit. The coprocessor #0 slot ("external" or "user" coprocessor) can be used by the system designer to attach custom logic to the Atlas CPU. 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 #0 can also be accessed in user mode, but if necessary, the access can be restricted to system mode by setting the protection flag in the machine status register. Any attempt to access a protected coprocessor in user mode will trigger the command error 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 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 command error trap is taken. The different flags and flag sets of the MSR are shown in the figure below.

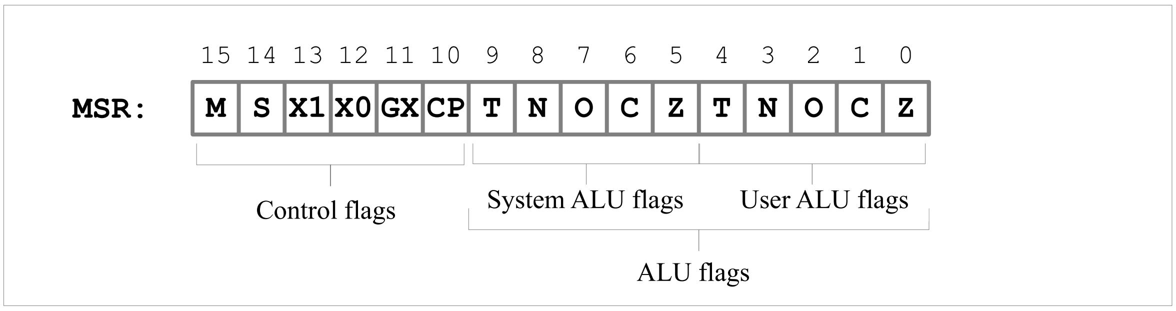


Figure 3: Machine Status Register

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 # for user mode	Bit # for 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 5: 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**	External coprocessor (coprocessor #0) protection	Coprocessor #0 can be accessed in user and system mode	Coprocessor #0 can only be accessed in system mode
11	**GX**	Global interrupt line enable	Disable interrupt lines	Enable interrupt lines
12	**X0**	IRQ controller IRQ mask	Disable IRQ controller IRQ	Enable IRQ controller IRQ
13	**X1**	CRITICAL_IRQ_I mask	Disable CRITICAL_IRQ_I	Enable CRITICAL_IRQ_I
14	**S**	Previous operating mode	Processor was in user mode	Processor was in system mode
15	**M**	Operating mode	Processor is in user mode	Processor is in system mode

Table 6: System control flags – access only in system mode

Bit 10 (**CP**-flag) is used to protect the external "user" coprocessor (coprocessor #0) from being accessed in user mode. An unauthorized access in user mode will trigger the command error 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 = CRITICAL_IRQ_I*, **X1** for *EXT_INT_1 = internal IRQ controller IRQ*) 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) indicates 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.

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).

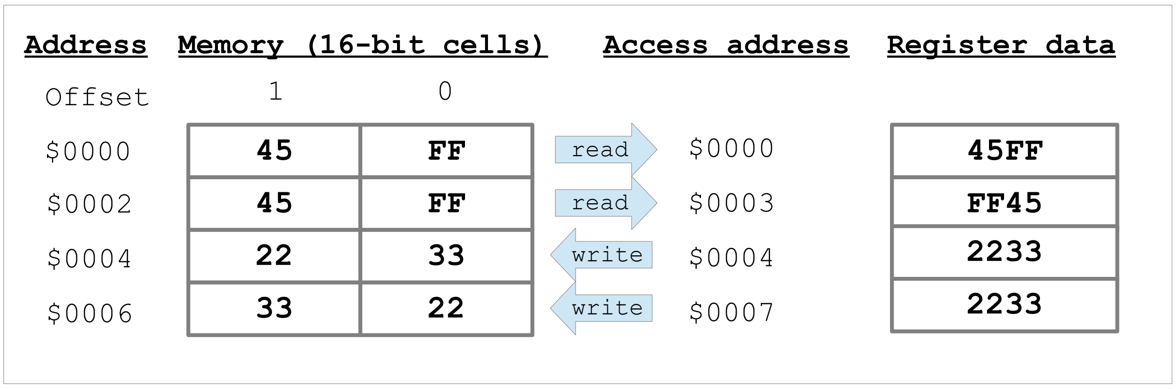


Figure 4: Memory accesses on aligned / unaligned word boundaries (hexadecimal data)

NOTE: Instruction fetch accesses will always be performed on aligned addresses, therefore instruction opcodes must be placed at word boundaries.

3.6.1. Physical Address Extension (Paging)

To extent the accessible memory space, the system coprocessor (coprocessor #1, SYS_1_CORE) of the Atlas 2k 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 and the MMU.

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; direct system calls or command errors), 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. Bit #0 of the program counter will always be zero.

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.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Processing	0	0	CMD				Rd			Ra			S	Rb		
Load MSR to register	0	0	0	1	1	0	Rd			A	B	0	0	0	0	0
Store register to MSR	0	0	0	1	1	1	0	0	0	A	B	0	0	Rb		
Store I. to ALU flags	0	0	0	1	1	1	0	T	N	1	B	1	0	O	C	Z
Load PC to register	0	0	1	1	1	0	Rd			0	0	0	0	0	0	0
Store register to PC	0	0	1	1	0	1	0	0	0	Ra			0	L	I	U
Load reg from user bank	0	0	1	0	0	1	Rd_sys			Ra_usr			S	Ra_usr		
Store reg to user bank	0	0	1	0	0	0	Rd_usr			Ra_sys			S	Ra_sys		
Memory Access	0	1	P	U	W	L	Rd			Ra			I	Offset		
Memory Swap	0	1	1	0	0	0	Rd			Ra			0	Rb		
Branch (and link)	1	0	COND				L	Offset								
Load Immediate	1	1	0	0	M	I	Rd			Immediate						
Bit Manipulation	1	1	0	1	M	S	Rd			Ra			Bit			
Coprocessor Processing	1	1	1	0	0	N	Cd/Cb			Ca			0	CMD		
Coprocessor Transfer	1	1	1	0	1	N	Cd/Rd			Ca/Ra			L	CMD		
Multiplication	1	1	1	1	0	0	Rd			Ra			0	Rb		
Sleep	1	1	1	1	0	1	0	Tag								
Reg-based branches	1	1	1	1	0	1	1	A	L	COND				Rb		
Conditional move	1	1	1	1	1	0	Rd			COND				Rb		
System Call	1	1	1	1	1	1	Tag									
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 5: Instruction set formats

4.1. Data Processing

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

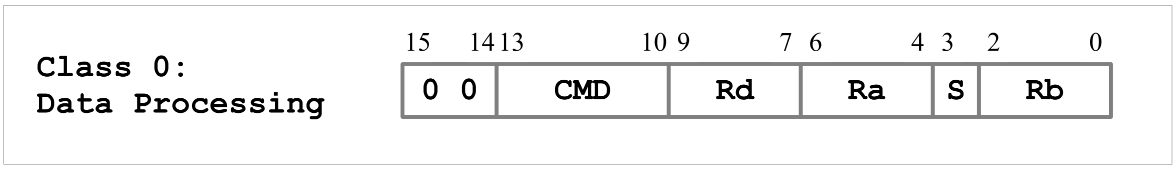


Figure 6: Data processing instructions format

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. There are four different options, which specify how the previous state of the carry and zero flags are taken into account (options for the carry flag: use carry_in-flag or use inverted carry_in-flag; options for the zero-flag: AND current zero-flag with previous zero-flag or OR current zero-flag with previous zero-flag).

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".

A "redundant " `SUB Rd, Ra, Rb` instruction with Ra = Rb would always result zero. Since other instructions exist to clear a register (like `XOR Rd, Ra, Ra`), this type of instruction is used to implement a negation instruction. So a `SUB Rd, Ra, Ra` is interpreted as `NEG Rd, Ra` (which is also accepted by the assembler) and computes Rd = 0 – Ra. The redundant form of the `SUB` instruction should not be used. Furthermore, a redundant version of the `SBC` instruction would be possible. An `SBC` instruction with Ra = Rb is also unlikely and does not really make sense, therefore this instruction also implements a subtraction from 0, but this time with taking the carry flag of a previous computation into account. Actually a redundant SBC instruction (`SBC Rd, Ra, Ra`) is processes Rd = 0 – Ra – Carry and is named `NEC` (`NEC Rd, Ra`).

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 7: 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 8: Shift commands; note that '&' indicates a concatenation

The CPX instruction supports following flag input/output options:

Option	CMD[9:7]	Flag input	Flag output
none | C_ANDZ	000	C_in = C	C = SUB(Rx, Ry); Z = Z_old AND Z(SUB(Rx, Ry))
NOTC_ANDZ	100	C_in = not C	C = SUB(Rx, Ry); Z = Z_old AND Z(SUB(Rx, Ry))
C_ORZ	010	C_in = C	C = SUB(Rx, Ry); Z = Z_old OR Z(SUB(Rx, Ry))
NOTC_ORZ	110	C_in = not C	C = SUB(Rx, Ry); Z = Z_old OR Z(SUB(Rx, Ry))

Table 9: CPX flag generation options

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, TST, TEQ (compare/test operations, no result write-back to a register)
```
<CMP|TST|TEQ> <Ra>, <Rb>
```

3. CPX (compare extended using flags, no result write-back to a register)
```
<CPX> <Ra>, <Rb>{, <C_ANDZ|C_ORZ|NOTC_ANDZ|NOTC_ORZ>}
```

4. 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>
```

5. 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 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 to R1
CMP  R2, R0            ; compare low words first, then
CPX  R3, R4, C_ORZ     ; compare and set Z_new = Z(result) OR Z_old
```

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"
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"
CPX  R3, R4, C_ORZ     = 0b 00.0111.010.011.1.100 = x"1D3C"
```

4.1.1. User Register Bank Access

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

	15 14	13 10	9 7	6 4	3	2 0
Load from user bank	0 0	1 0 0 1	Rd_sys	Ra_usr	S	Ra_usr
Store to user bank	0 0	1 0 0 0	Rd_usr	Ra_sys	S	Ra_sys

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 command error trap.

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.

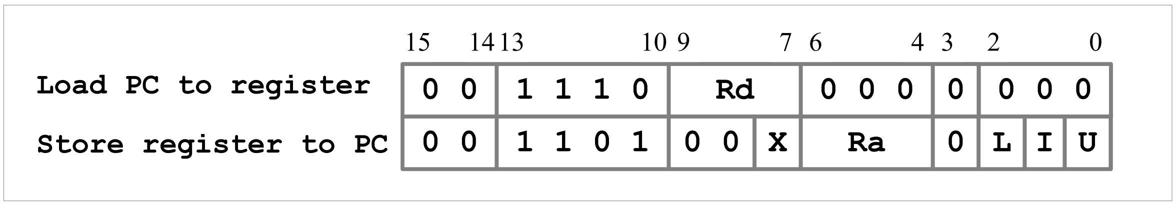


Figure 8: Program counter access instructions subset formats

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 command error 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 instruction-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.

	15	14	13			10	9		7	6		4	3	2		0
Load MSR to register	0	0	0	1	1	0	Rd			A	B	0	0	0	0	0
Store register to MSR	0	0	0	1	1	1	0	0	0	A	B	0	0	Rb		
Store I. to ALU flags	0	0	0	1	1	1	0	T	N	1	B	1	0	O	C	Z

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. The 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 command error 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)	CMD error exception
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}`	Read/write user / system / 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
```

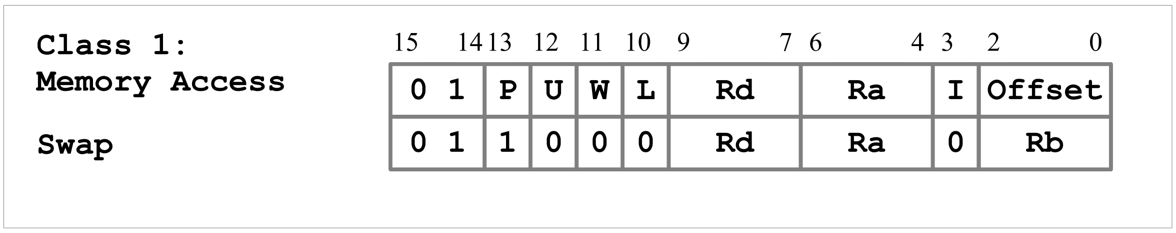
Coding Examples

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.



	15	14	13	12	11	10	9 … 7	6 … 4	3	2 … 0
Class 1: Memory Access	0	1	P	U	W	L	Rd	Ra	I	Offset
Swap	0	1	1	0	0	0	Rd	Ra	0	Rb

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.

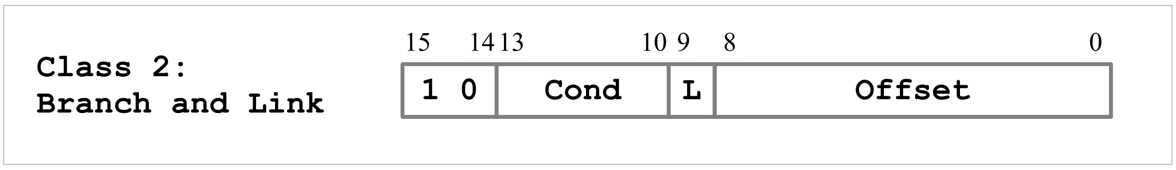


Figure 11: Branch and link instructions format

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	Ra = Rb	Z
NE	0001	Not equal	Ra != Rb	not Z
CS	0010	Unsigned higher or same	Ra =< Rb	C
CC	0011	Unsigned lower	Ra > Rb	not C
MI	0100	Negative	(Ra – Rb) < 0	N
PL	0101	Positive or zero	(Ra – Rb) >= 0	not N
OS	0110	Overflow		O
OC	0111	No overflow		not O
HI	1000	Unsigned higher	Ra < Rb	C and (not Z)
LS	1001	Unsigned lower or same	Ra >= Rb	(not C) or Z
GE	1010	Greater than or equal	Ra =< Rb	N xnor O
LT	1011	Less than	Ra > Rb	N xor O
GT	1100	Greater than	Ra < Rb	(not Z) and (N xnor O)
LE	1101	Less than or equal	Ra >= Rb	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.

NOTE: The presented condition codes are also used for the register-based branches and the conditional move instructions (see later).

Assembler Syntax

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

1. 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.

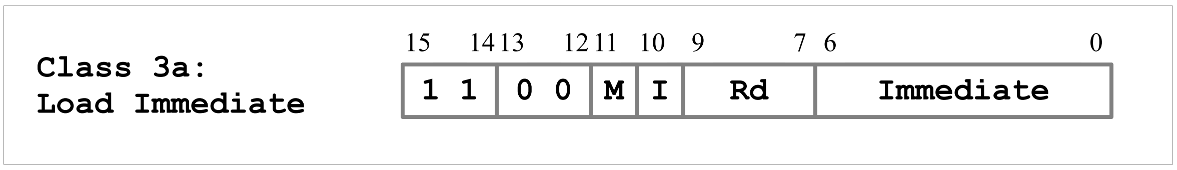


Figure 12: Load immediate instructions format

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.

1. 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.

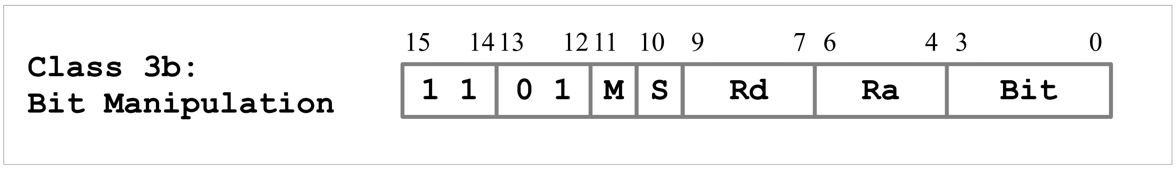


Figure 13: Bit manipulation instructions format

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".

Register-given index for store-to-T-Flag operations

When using the `STB` or `STBI` command, the indexed bit is given by a four-bit immediate. Since bit 9 of the corresponding opcode is not uses for coding the operation itself, it is used to select between the mentioned immediate indexing or a register-based immediate (when bit 9 is set). For the last case, the actual bit index is given by the lowest 4 bit of the second register argument. Use the `STBR` instruction to store the bit, indexed by the lowest 4 bit of the second source register, to the T-flag. Use the `STBRI` instruction to store the inverted bit, indexed by the lowest 4 bit of the second source register, to the T-flag.

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 immediate-indexed bit)
```
<SBR|CBR> <Rd>, <Ra>, <#Imm>
```

2. LDB (load immediate-indexed bit from T-flag)
```
<LDB> <Rd>, <Ra>, <#Imm>
```

3. STB/STBI (store immediate-indexed bit to T-flag / store inverted immediate-indexed 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>
```

5. STBR/STBRI (store register-indexed bit to T-flag / store inverted register-indexed bit to T-flag)
```
<STBR>{I} <Ra>, <Ri>
```

`{I}`	Invert source/parity bit while it is transferred to the T-flag when present.
`<Rd>`	Destination register.
`<Ra>`	Source register.
`<Ri>`	Index register.
`<#Imm>`	4-bit immediate value addressing 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
STBR R7, R4        ; store bit R7[R4(3:0)] 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"
STBR R7, R4       = 0b 11.01.1.1.100.111.0100 = x"DE74"
```

4.6. Coprocessor Data Processing

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

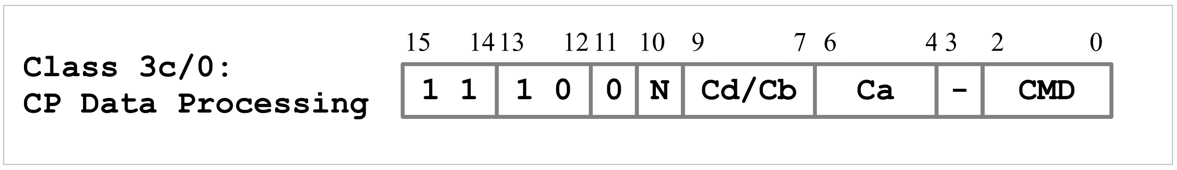


Figure 14: Coprocessor data processing instructions format

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 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.

1. 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.

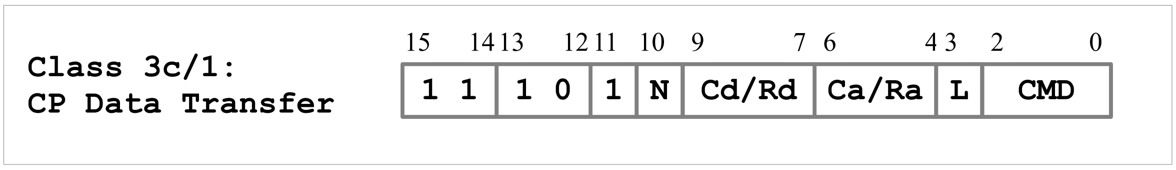


Figure 15: Coprocessor data transfer instructions format

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 (move data from coprocessor to CPU: L = '0', move data from CPU to coprocessor: 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 (move coprocessor- register to CPU-register)
```
<MRC> <#CP>, <Rd>, <Ca>, <#Cmd>
```

2. MCR (move CPU-register to coprocessor-register)
```
<MCR> <#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

The instruction encoding of the multiply instruction is shown in the figure below.

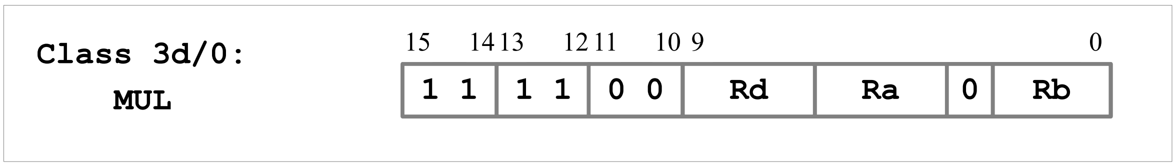


Figure 16: MUL instruction format

The `MUL` instruction will multiply Ra and Rb and places the lowest 16-bit of the result in Rd.
Rd ← (Ra*Rb)[15:0]. This instruction does not perform any kind of flag manipulation.

Assembler Syntax

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

1. MUL (multiply)
```
<MUL> <Rd>, <Ra>, <Rb>
```

`<Rd>`	Destination register.
`<Ra>`	Operand A register.
`<Rb>`	Operand B register.

Assembler Examples

```
MUL R0, R1, R2    ; R0 = 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.

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

4.9. Sleep Command

The instruction encoding of the sleep command is shown in the figure below.

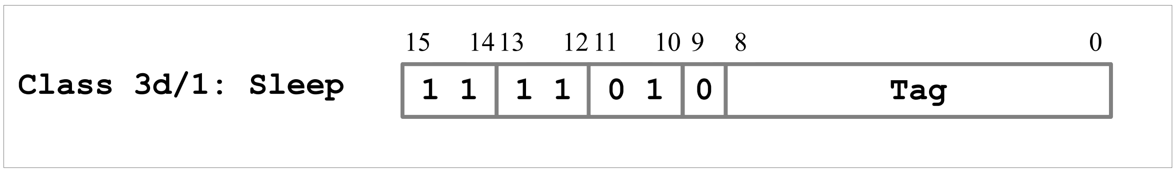


Figure 17: Sleep command

The sleep command will set the CPU in deep sleep mode disabling the pipeline as well as the instruction fetch system. Thus, the power consumption caused by dynamic switching activity can be massively reduced. After entering sleep mode, the CPU is frozen an will only wake up in reset or an incoming interrupt request on then *xirq0* or *xirq1* signal pin (interrupt request from internal coprocessor or via the *critical IRQ* pin). Note, that the corresponding interrupt lines have to be activated (MSR: **X0** and/or **X1** flag have to be set; the state of the global interrupt enabled flag **GX** (MSR) is irrelevant). A 9-bit tag (**Tag**-field) can be applied to the instruction, to specify information about the sleep entering condition, that can be checked by an interrupt handler. If the **GX** flag is cleared, operation resumes right after the sleep command when an interrupt request occurs. When the **GX** flag is set, operation resumes in the corresponding interrupt handler. Both cases require the **X0** and/or **X1** flag to be set.

NOTE: The sleep command can only be executed in SYSTEM mode. If executed in USER mode, the command error trap is taken.

Assembler Syntax

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

1. SLEEP (set CPU to sleep mode)

```
<SLEEP> {#Tag}
```

`{#Tag}`	9-bit immediate value, automatically set to zero if not present; use #-prefix.

Assembler Examples

```
SLEEP #412          ; go to sleep mode with '412' as tag
SLEEP               ; go to sleep mode with no tag (tag-field is all-zero)
```

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.

```
SLEEP #412          = 0b 11.11.01.0.110011100 = x"F59C"
SLEEP               = 0b 11.11.01.0.000000000 = x"F400"
```

4.10. Register-Based Branches

The instruction encoding of the register-based branch instructions is shown in the figure below.

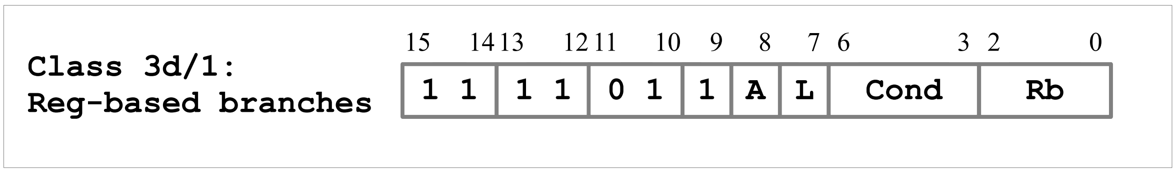


Figure 18: Register-based branches

The register-based branches allow to conditionally branch absolute (bit 8: **A** = '1') to a destination given by a register or relative (bit 8: **A** = '0') to a register-given offset depending on a condition (**COND**-field). The condition codes are the same as the ones from the "branch / branch and link" instructions (see above). It is also possible to save the return address to the link register by setting the link bit (bit 7: **L** = '1').

Assembler Syntax

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

1. RBA, RBR (register-based absolute/relative branch, conditional or unconditional, link-option)

```
<RBA|RBR>{L}{cond} <Rb>
```

`<RBA>`	Absolute branch, register Rb gives branch destination.
`<RBR>`	Relative branch, register Rb gives branch offset.
`{L}`	Store return address to link register when present.
`{cond}`	Condition code (see above). If not present, 'always' (AL) condition is used.
`<Rb>`	Register with branch destination / offset.

Assembler Examples

```
RBA     R4          ; always branch absolute to [R4]
RBREQ   R4          ; if equal, branch to PC+R4
RBRLEQ  R4          ; if equal, branch to PC+R4 and link
```

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.

```
RBA     R4          = 0b 11.11.01.1.1.0.1111.100 = x"F77C"
RBREQ   R4          = 0b 11.11.01.1.0.0.0000.100 = x"F604"
RBRLEQ  R4          = 0b 11.11.01.1.0.1.0000.100 = x"F684"
```

4.11. Conditional Move

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

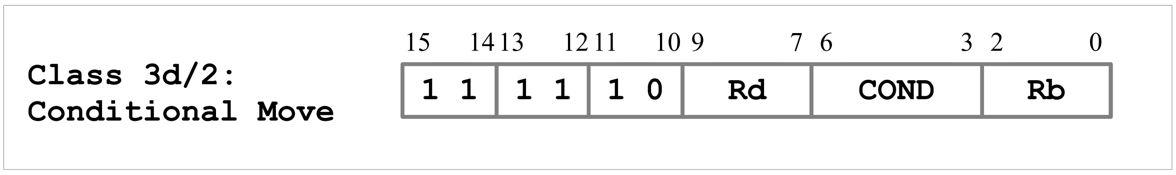


Figure 19: Conditional move instruction format

Conditional move instructions can be used to move data from the source register (**Rb**-field) to the destination register (**Rd**-field) when the specified condition (**COND**-field) is true. The condition codes are the same as for the register-based branch and simple branch instructions (see above).

Assembler Syntax

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

1. MV (conditional move)

```
<MV>{cond} <Rd>, <Rb>
```

`{cond}`	Condition code. If not present, 'always' (AL) condition is used.
`<Rb>`	Source register.
`<Rd>`	Destination register.

Assembler Examples

```
MVEQ  R4, R7      ; R4 <= R7 if equal
MV    R0, R1      ; R0 <= R1 (always)
MVCC  R3, R0      ; R3 <= R0 if carry is cleared
```

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.

```
MVEQ  R4, R7      = 0b 11.11.10.100.0000.111 = x"FA07"
MV    R0, R1      = 0b 11.11.10.000.1111.001 = x"F879"
MVCC  R3, R0      = 0b 11.11.10.011.0011.000 = x"F998"
```

4.12. System Call

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

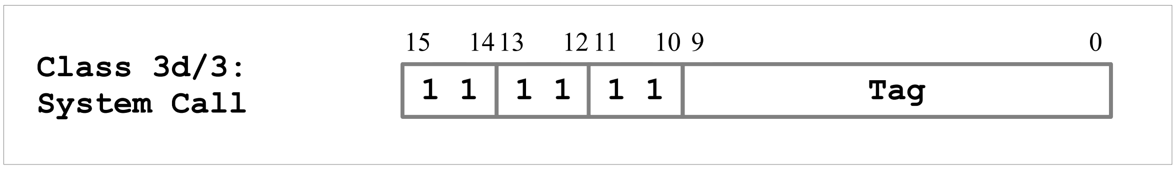


Figure 20: System call instruction format

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**-field) 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 user mode, the instruction will behave like a branch and link instruction to the software interrupt vector, which is executed in system mode. When returning with `RTX` from the software interrupt handler, the original program will be resumed in user mode, since the previous mode (system) has been stored in the MSR.

Assembler Syntax

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

1. 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 (all-zero)
```

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"
```

4.13. Pseudo Instructions

Pseudo instructions are instructions, constructed from actual implemented instructions. These operations can help to make assembler programs more user friendly and easier to edit and read.

Pseudo instruction	Translation	Function
CLR Rx	**EOR** Rx, Rx, Rx	Clear register Rx (Rx ← x"0000")
CLRS Rx	**EORS** Rx, Rx, Rx	Clear register Rx (Rx ← x"0000") and set flags
COM Rx	**NAND** Rx, Rx, Rx	Invert register (Rx ← not Rx)
COMS Rx	**NANDS** Rx, Rx, Rx	Invert register (Rx ← not Rx) and set flags
MOV Rx, Ry	**INC** Rx, Ry, #0	Copy register (Rx ← Ry)
MOVS Rx, Ry	**INCS** Rx, Ry, #0	Copy register (Rx ← Ry) and set flags
NOP	**INC** R0, R0, #0	No operation
PEEK Rx	**LDR** Rx, R6, +#0, pre	Copy word from top of stack (Rx ← MEM[R6])
POP Rx	**LDR** Rx, R6, +#2, pre, !	Pop register Rx from negative growing stack (normal case) (Rx ← MEM[R6+2]; R6 ← R6+2)
POP+ Rx	**LDR** Rx, R6, -#2, pre, !	Pop register Rx from explicit positive growing stack (Rx ← MEM[R6-2]; R6 ← R6-2)
PUSH Rx	**STR** Rx, R6, -#2, post, !	Push register Rx on negative growing stack (normal case) (MEM[R6] ← Rx; R6 ← R6-2)
PUSH+ Rx	**STR** Rx, R6, +#2, post, !	Push register Rx on explicit positive growing stack (MEM[R6] ← Rx; R6 ← R6+2)

Table 15: List of implemented pseudo-instructions

5. Atlas Evaluation Assembler

I've programmed a small assembler, that is capable of assembling the previously explained instructions into an Atlas CPU-compatible binary format. The program is located in the *asm* folder and can be run using the command prompt. The actual assembly program is passed as first argument when calling the assembler. You can specify a 10 character long name for the assembled image as second argument, but this is optional.

For example to assemble the blink demo eyample...:

```
...\asm>atlas_asm ..\software\examples\blink_demo\demo.asm blink_demo
ATLAS 2k Assembler, Version 2014.04.12
by Stephan Nolting (stnolting@gmail.com), Hanover, Germany
www.opencores.org/project,atlas_core


Assembler completed without errors (0 warnings)
Image size:    0x002E (46) bytes
XOR check sum: 0xAA75
Image name:    blink_demo
```

For every assembling process, the assembler will generate several files (in the same folder as the executable). The following table explains the function of this files.

File name	File type	Function
`pre_processor.asm`	`ASM text file`	Intermediate processing file. The line numbers of warning and errors correspond to this file.
`init.vhd`	`VHDL text file`	VHDL file containing a memory signal initialization. Use this file for simulation or direct ROM initialization. Note: The hardware can directly start this image – this image can not be booted by the bootloader!
`boot_init.vhd`	`VHDL text file`	VHDL file for the initialization of memory component, that can be booted by the bootloader (e.g. a Wishbone boot ROM).
`out.bin`	`Binary file`	Binary output image. Use this file for transferring a program via the bootloader (boot from UART / burn EEPROM).

Table 16: Assembler – generated files

NOTE: The assembler program was compiled for a 64-bit Windows machine, maybe you need to recompile the assembler sources to make it work on your system.

NOTE: If you are using instruction constructs, like a redundant `ORR`, which evolves to a user-bank transfer, the assembler will output a warning to inform you.

5.1. Pre-Processor Instructions

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

Instruction	Example	Function

`.equ`	`.equ temp     r4`
	`.equ ctrl_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 registers (r0, …, r7), the coprocessor registers (c0, …, c7) or immediate values (positive integers, 16-bit, decimal/ binary/ hexadecimal representation, introduced with '#'-prefix, see below)

`.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!"`
	`.stringz "With linebreak\nnext line"`
		With the .string instruction you can initialize memory directly with an ASCII string. All "**.stringz**" strings are automatically terminated with one/two zeroes, depending on word-boundaries. A '\n' will include a line break to the string. Do not use '\' alone!

`.include`	`.include "file_name"`	Copy content of file *file_name* to the include instruction's position (file must be in the same directory).

Table 17: Assembler - pre-processor instructions

The assembler/pre-processor supports binary, hexadecimal and decimal representation for numbers. The following table shows the three formats and their usage.

Number representation	Number suffix	ASM Example	Function
`decimal`	*none*	**LDIL** `Rx, #98`	Rx ← 98
`hexadecimal`	`0x`	**LDIL** `Rx, #0x62`	Rx ← 0x62 = 98
`binary`	`0b`	**LDIL** `Rx, #0b01100010`	Rx ← 0b01100010 = 98

Table 18: Assembler - number representations

Also, several different methods for simple number/offset handling are implement:

Method keyword	ASM Example	Function
`low[]`	**`LDIL`** `Rx, low[some_constant]`	Load low byte of constant into Rx. For example the constant is defined as "`.equ some_constant #2400`", which would copy 2400&255 = 96 to Rx. Can be used for direct register initialization.
	`LDIL` `Rx, low[some_label]`	Load low byte of the absolute 16-bit address of label "`some_label`" into Rx. Can be used for register-based absolute branches (e.g. `GT` instruction).
`high[]`	**`LDIH`** `Rx, high[some_constant]`	Load high byte of constant into Rx. For example the constant is defined as "`.equ some_constant #2400`", which would copy (2400>>8)&255 = 9 to Rx. Can be used for direct register initialization.
	`LDIH` `Rx, high[some_label]`	Load high byte of the absolute 16-bit address of label "`some_label`" into Rx. Can be used for register-based absolute branches (e.g. `GT` instruction).
`rlow[]`	**`LDIL`** `Rx, rlow[some_label]`	Load the low byte of the relative distance from this instruction to the label "`some_label`" into Rx. Can be used for register-based relative branches (e.g. `RBR` instruction).
`rhigh[]`	**`LDIH`** `Rx, rhigh[some_label]`	Load the high byte of the relative distance from this instruction to the label "`some_label`" into Rx. Can be used for register-based relative branches (e.g. `RBR` instruction).

Table 19: Assembler - number/offset handling methods

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

The `MUL` instruction features no status flag update. Also, a `MAC` instruction is not included in the Atlas CPU specifications. So, if a `MAC` operation with flag update is required, it is suitable to construct the actual `MAC` operation from 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 absolute 16-bit base address of the branch table
        LDIL R0, low[branch_table]       ; load low address byte of branch table
        LDIH R0, high[branch_table]      ; load high address byte of branch table

        ; multiply offset by 2 by left-shifting one position; this is necessary, since
        ; 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 (R6) while loading or storing data (R0) 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. Of course you can also use the assembler pseudo instructions push/push+/pop/pop+/peek.

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

        LDR R0, R6, +#0, pre             ; peek operation: r0 = top of stack (peek r0)
```

5.2.7. Print Strings

String are used to process data arrays. For instance, these can be arrays of ASCII characters, terminated with a zero byte. The following example shows how to output a string via a serial port (the actual sending subroutine is left blank, see the chapter about the UART core for an actual subroutine example). This code also outputs a line break after the string.

```
       ;print strings + line feed
       ;executed in system mode

       LDIL  R3, low[text_string]       ; load absolute address of string
       LDIH  R3, high[text_string]
       BL    uart_print                 ; call the print routine

       ...

text_string:
.stringz "To boldly go,\nwhere no man has gone before..."  ; zero-terminated text string
                                                           ; with line break by '\n'

       ...

; UART string print subroutine, sends string addressed with r3
; -------------------------------------------------------------------
uart_print:         MOV  R1, LR                        ; save link register

uart_print_loop:    LDR  R2, R3, +#1, post, !          ; get one string 'word'

                    LDIL R0, #0x00                     ; high-byte mask
                    LDIH R0, #0xFF
                    AND  R2, R2, R0                    ; apply mask

                    SFTS R2, R2, #SWP                  ; swap bytes & zero test
                    BEQ  uart_print_end                ; done when r2 is zero

                    BL   uart_sendbyte                 ; send one byte
                    B    uart_print_loop               ; resume loop

uart_print_end:     LDIL R2, #0x0A                     ; send line feed
                    BL   uart_sendbyte
                    LDIL R2, #0x0D                     ; carriage return
                    BL   uart_sendbyte

                    RET  R1                            ; done


; UART sendbyte subroutine, transmits low-byte of R2 via UART
; -------------------------------------------------------------------
uart_sendbyte:      ; send r2 via system UART...
                    RET  LR
```

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

      CLR  R1                   ; 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 means storing r2's parity to the T-flag

      LDSR R2                   ; copy MSR to r2
      LDB  R2, R2, #6           ; copy system-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
```

5.2.10. Interrupt Vector Table

The interrupt vector table contains the five 16-bit addresses of the different interrupt handlers. When the actual handler is out of the range of a simple branch instruction, a branch to an intermediate calling functions must be performed (an example of this is shown using the command error trap "cmd_err_handler").
The complete interrupt vector table must always be at the beginning of the program, thus the branch instruction for the reset-handler must be at PC (program counter) location x"0000", the branch instruction for the external interrupt line 0 handler must be at PC location "x0002" and so on (byte-addressing mode).

```
      ;Interrupt Vector Table
      ;auto-executed in system mode, since it's part of the interrupt handler system

; there MUST NOT be any instruction/data before this part!

reset_vec:          B reset                  ; reset handler
x_int0_vec:         B xint0_handler          ; external interrupt line 0 handler
x_int1_vec:         B xint1_handler          ; external interrupt line 1 handler
cmd_err_vec:        B cmd_err_handler_pre    ; command error handler
swi_vec:            B swi_handler            ; system call handler

...

cmd_err_handler_pre:      ; since the actual 'command error'- handler is out of reach of
                          ; a simple branch instruction, this stopover is necessary
      LDIL R0, low[cmd_err_handler]  ; load low byte of absolute address of handler
      LDIH R0, high[cmd_err_handler] ; load high byte of absolute address of handler
      GT   R0                        ; go to that address

...

cmd_err_handler: ...      ; actual 'command error'- handler
```

5.2.11. Hardware-based OR compare

Sometimes it is necessary to check if a register is equal to at least one element of a list (e.g. if a state varibale is set to "RUN" or "FROZEN"....). The CPX instructions allows a very efficient way to implement such "OR-ed" compare operations.

```
      ;Hardware OR-ed compare of R1 using CPX
      ; executed in system or user mode

      LDIL R0, #0x05      ; symbol 1
      CMP  R0, R1         ; set Z flag for BEQ

      LDIL R0, #0x27      ; symbol 2
      CPX  R0, R1, C_ORZ  ; compare R0 and R1 and OR zero-detector result with Z-flag

      LDIL R0, #0x33      ; symbol 3
      CPX  r0, R1, C_ORZ  ; compare R0 and R1 and OR zero-detector result with Z-flag

      BEQ  some_label     ; execute branch if R1 is equal to at least 1 of the symbols
```

5.2.12. Conditional Execution

If-then constructs are basic component of every program. A simple way of implementing them is to execute always the if-part, then evaluate the condition and skip the else-part if the condition is false. Otherwise the else part is executed, overwriting the results of the then-part. Of course this style requires that the if-part does not perform any damage, since it is always executed, even if the condition is not fulfilled.

```
      ;conditional execution (bad implementation)
      ;executed in system or user mode

      ; if (carry_flag = 1) then r0 = r1+r2 else r0 = 10
      ADD  R0, R1, R2     ; if part                (1 cycle)
      BCS  end            ; branch if carry set    (1/3 cycles)
      LDIL R0, #10        ; else part              (1 cycle)
end:  ...
```

This implementation requires 3 clock cycles if the condition was false and 5 clock cycles if the condition was true resulting in a not predictable execution time. To overcome this issue, you can use partial predication. The Atlas CPU does not support full predicated execution like the ARM architecture – there is just not enough space in the opcodes. However, there are conditional move instructions, which can select the result of two processing paths (if-part and else-part) according to a given condition. Since no branches are used, the execution time for this scenario is always determine (and even shorter for this scenario).

```
      ;conditional execution (good implementation)
      ;executed in system or user mode

      ; if (carry_flag = 1) then r0 = r1+r2 else r0 = 10
      ADD  R3, R1, R2     ; if part                (1 cycle)
      LDIL R0, #10        ; else part              (1 cycle)
      MVCS R0, R3         ; copy if carry set      (1 cycle)
      ...
```

5.2.13. Long Relative Branches

The `B` and `BL` instructions (conditional branch / branch and link) support a 9-bit wide signed word-aligned offset for relative branching. This allows a maximum distance of -256 and +255 instructions / words. But what if we need a conditional relative branch to a label farther away (so with full 16 bit offset)? A possible implementation might look like this (condition is the the carry flag → branch if carry flag is set):

```
rel_branch_offset = absolute_address_of_label - absolute_address_of_branch_instruction
```

```
      ;long realtive branch (bad implementation)
      ;executed in system or user mode

this: LDIL  R0, low[label]
      LDIH  R0, high[label]     ; get absolute address of label "label"
      LDIL  R1, low[this]
      LDIH  R1, high[this]      ; get absolute address of label "this"
      SUB   R2, R0, R1          ; distance between labels "this" and "label"
                                ; = relative offset
      DEC   R2, R2, #7
      DEC   R2, R2, #7          ; subtract the offset for the four LDI, the 2 DEC and
                                ; the SUB instruction = 7*2=14 byte =
                                ; this is the offset between "label" and the branch cmd
      RBRCS R2                  ; branch relative to [r2] if carry set

      ...                       ; more than 255 instructions in between

label: ...                      ; this label is far away...
```

Of course this computation overhead to determine the relative offset is absolutely unacceptable. To overcome this, the assembler tool provides a simple feature to directly load the relative distance of a label from an `LDI` instruction into a register using the `RLOW/RHIGH` options:

```
      ;long realtive branch (good implementation)
      ;executed in system or user mode

      LDIL  R0, rlow[label-4]   ; get low part of distance to "label" minus 4 byte
      LDIH  R0, rhigh[label-2]  ; get high part of distance to "label" minus 2 byte
      RBRCS R0                  ; branch relative to [r2] if carry set

      ...                       ; more than 255 instructions in between

label: ...                      ; this label is far away...
```

The `RLOW` argument gives the low byte of the distance to the given label and `RHIGH` gives the high byte of the distance to the given label. A "-4" for the `LDIL` is required, since the label "label" is 4 byte farther away at this point than from the instruction where we actually add the offset to the PC to perform the branch (the `RBRCS`). Thus, we need to subtract 2 from the offset for the `LDIH` instruction.

NOTE: There must be no spaces inside a label reference. Things like `LDIL R0, RLOW[label_-_4]` are not allowed!!!

NOTE: This kind of long relative branching only works with the `RBR` and `RBRL` instructions!

6. Core Architecture

This chapter takes a closer look at the actual rtl implementation of the CPU core. In the following diagram, you can see the basic layout of the Atlas 2k processor.

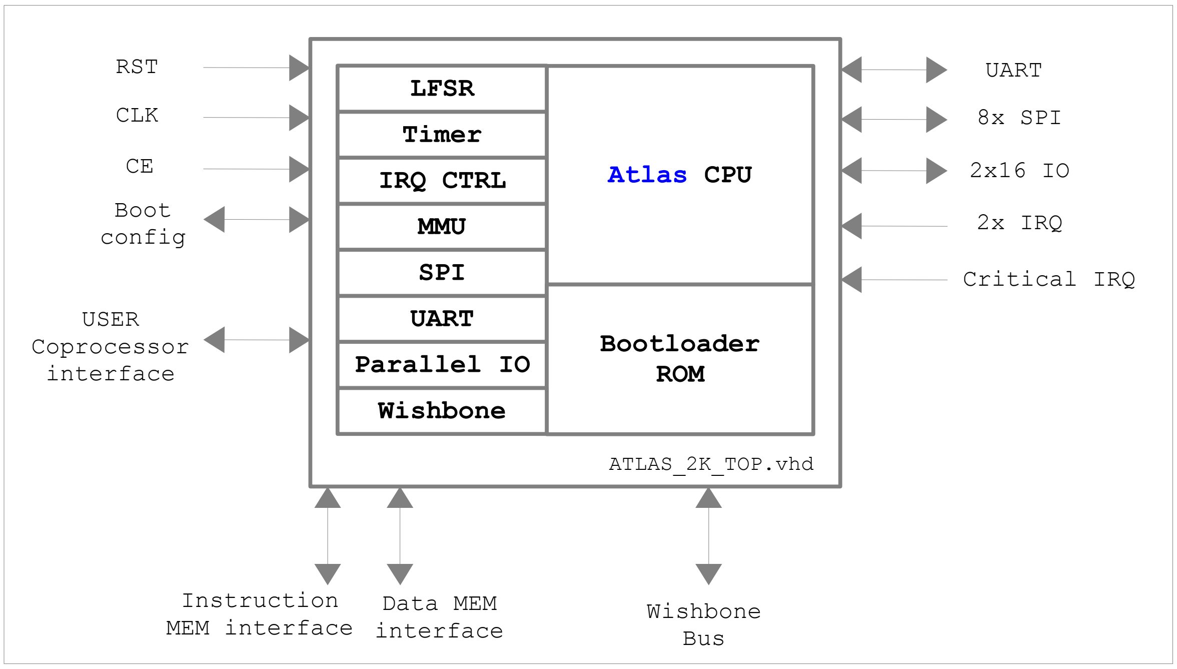


Figure 21: Atlas 2k Processor Overview

6.1. Module Description

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

File name	Functionality
ALU.vhd	The ALU holds the primary arithmetical/logical unit, the coprocessor interface as well as the multiplication unit (if synthesized).
ATLAS_2K_TOP.vhd	**This is the top entity of the Atlas 2k processor.**
ATLAS_CPU.vhd	Top entity of the Atlas CPU.
ATLAS_pkg.vhd	Package file for the Atlas project. All additional configurations are made here.
ATLAS_2K_BASE_TOP.vhd	Top entity of the basic system on chip setup, including the CPU and some RAM.
BOOT_MEM.vhd	ROM with bootloader code.
COM_0_CORE.vhd	Communication controller for PIO, SPI and UART.
COM_1_CORE.vhd	Wishbone communication controller.
CTRL.vhd	This file provides the control “spine” of the processor.
INT_RAM.vhd	Internal RAM component for the basic system setup.
MEM_ACC.vhd	All data memory requests emerge from this unit. Furthermore, processing result routing circuits are located here.
MEM_GATE.vhd	Bootloader ROM / memory system gateway.
OP_DEC_vhd	Opcode decoder. The instruction opcodes are decoded into processor internal control signals in this unit.
REG_FILE.vhd	This file contains the main data register file, organized as 2x16*16-bit memory.
RST_PROTECT.vhd	This unit guarantees a valid reset for the processor.
SYS_0_CORE.vhd	System controller with IRQ control, timer, and LFSR.
SYS_1_CORE.vhd	System controller with MMU.
SYS_REG.vhd	The system register file contains the program counter, the machine status register and the interrupt and context control circuits.
SYSTEM_CP.vhd	The access unit to all system coprocessor cores.
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 20: Atlas 2k VHDL rtl files and description

6.2. Data Path

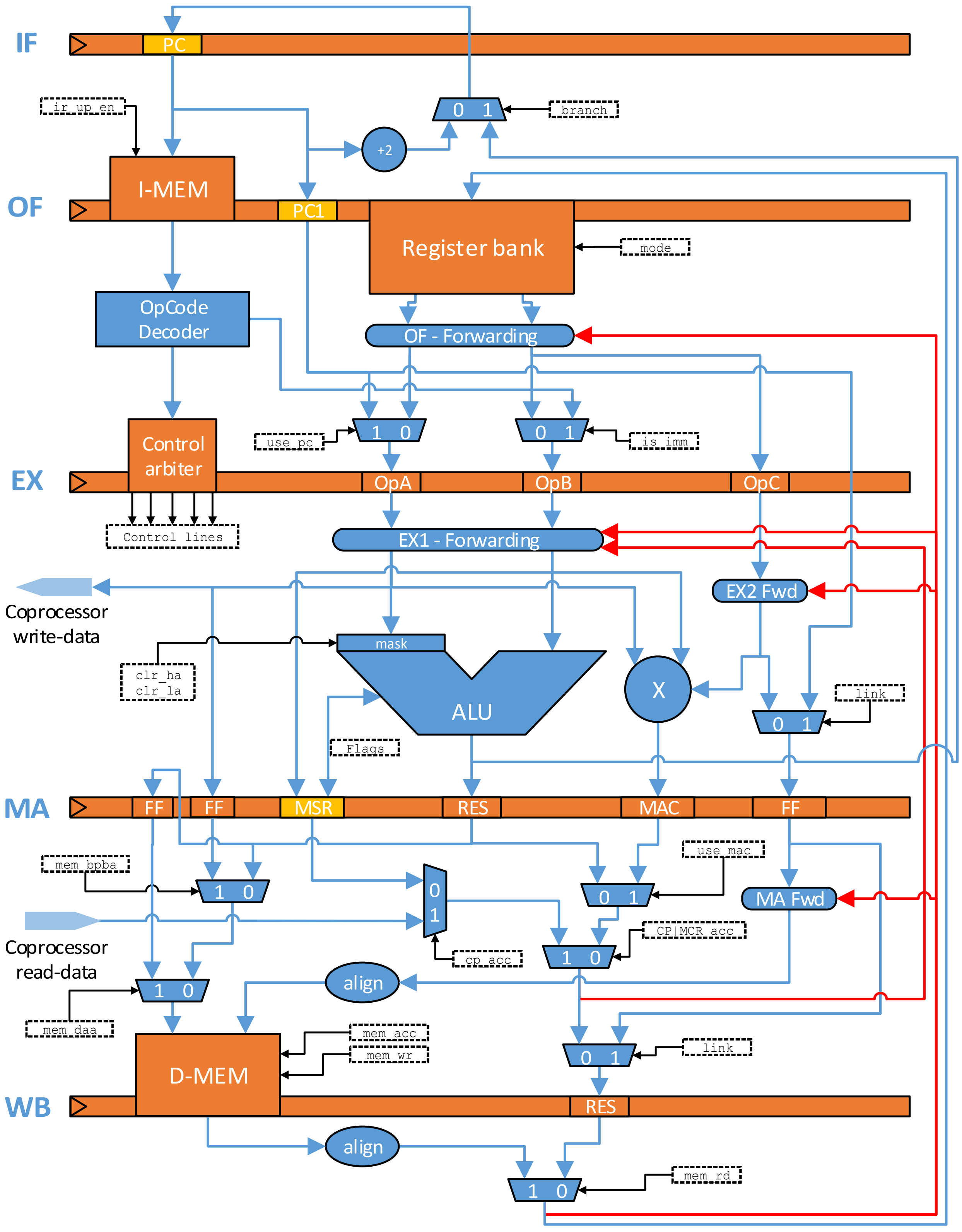


Figure 22: Atlas CPU main data path diagram

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 23: 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 blocks, 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 21: Atlas CPU pipeline stages

6.4.1. Local Pipeline Conflicts

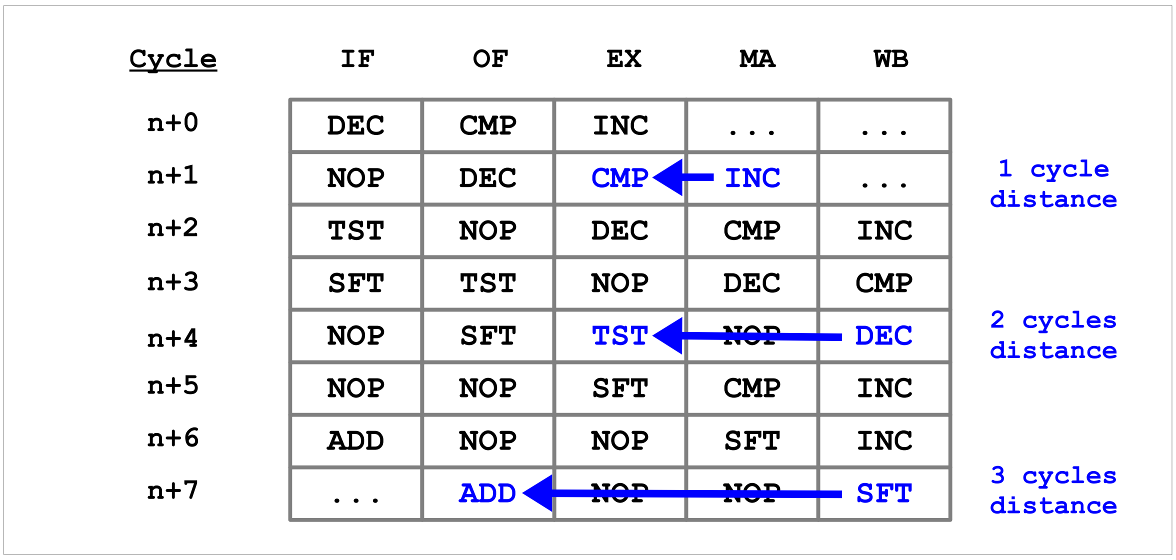
Whenever data is needed, that has already been processed but has not yet reached the end of the pipeline, a local data dependency occurs. For data, that will be processed by the ALU, the source and destination data can be separated by 1, 2 or 3 cycles in the pipeline. The following example program illustrates 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 AND 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).



Cycle	IF	OF	EX	MA	WB	
n+0	DEC	CMP	INC	. . .	. . .	
n+1	NOP	DEC	CMP	INC	. . .	1 cycle distance
n+2	TST	NOP	DEC	CMP	INC	
n+3	SFT	TST	NOP	DEC	CMP	
n+4	NOP	SFT	TST	NOP	DEC	2 cycles distance
n+5	NOP	NOP	SFT	CMP	INC	
n+6	ADD	NOP	NOP	SFT	INC	
n+7	. . .	ADD	NOP	NOP	SFT	3 cycles distance

Figure 24: 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.

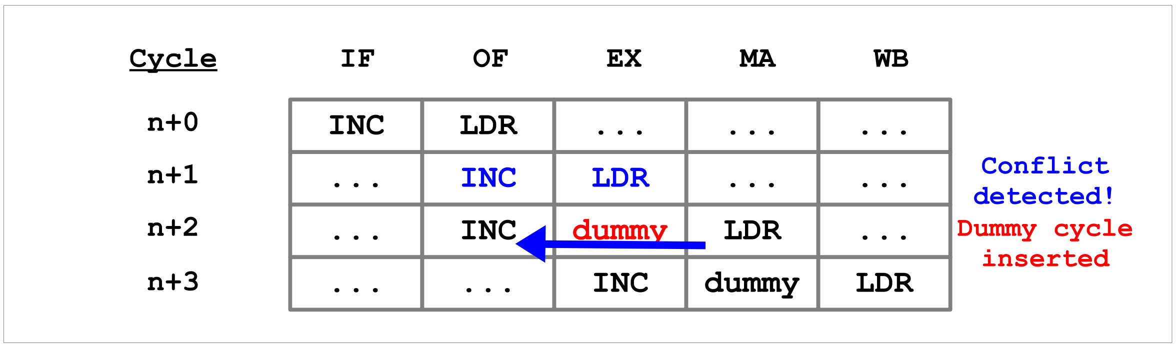


Figure 25: Memory read-data temporal data dependency

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.

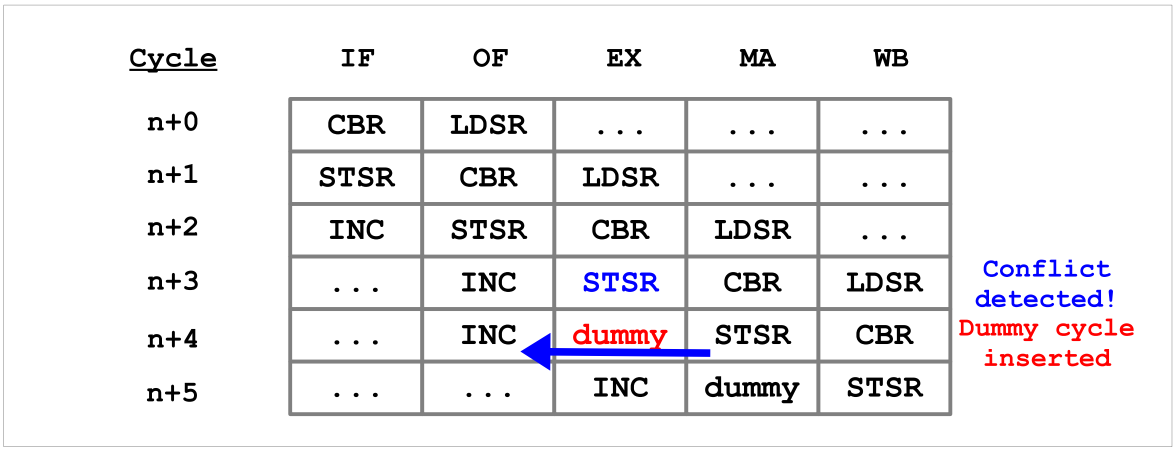
6.4.2.1. MSR Write Access

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.



Cycle	IF	OF	EX	MA	WB
n+0	CBR	LDSR	. . .	. . .	. . .
n+1	STSR	CBR	LDSR	. . .	. . .
n+2	INC	STSR	CBR	LDSR	. . .
n+3	. . .	INC	STSR	CBR	LDSR
n+4	. . .	INC	dummy	STSR	CBR
n+5	. . .	. . .	INC	dummy	STSR

Figure 26: 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 update instructions are very rare in common programs, this issue should not be further relevant.

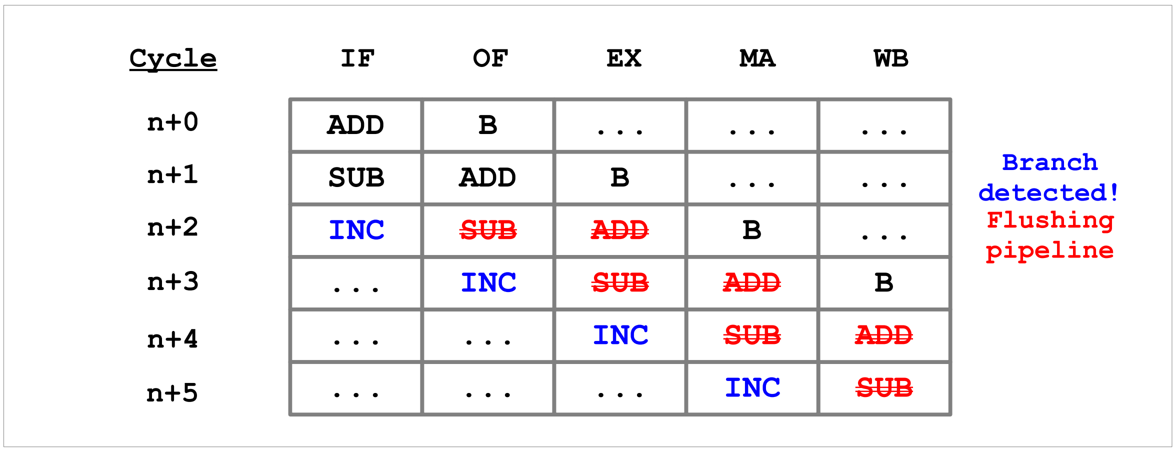
6.4.4. Branches

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”).



Cycle	IF	OF	EX	MA	WB
n+0	ADD	B	...	...	...
n+1	SUB	ADD	B	...	...
n+2	INC	~~SUB~~	~~ADD~~	B	...
n+3	...	INC	~~SUB~~	~~ADD~~	B
n+4	...	...	INC	~~SUB~~	~~ADD~~
n+5	...	...	...	INC	~~SUB~~

Figure 27: 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 interrupted program. While exceptions (system call / undefined instruction / access violations) can only occur synchronous to the pipeline and 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, external interrupts (via the two IRA lines of the CPU) 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 2k provides three different interfaces:

1. The memory interface (data and instructions)
2. The coprocessor interface
3. The peripheral IO interface
4. The Wishbone bus interface

The memory interface is mandatory for the processor to operate. It consists of a read-only instruction memory and a read/write data interface. Both interfaces are 16-bit wide (address and data bus) and the memory system is byte-addressed.

The coprocessor interface can be used to connect a hardware accelerator or an additional communication controller tightly to the CPU to extend the processor functionality and processing power. The peripheral/IO interface directly connects the processor to the outer world. This includes the UART, an 8 channel SPI and a 2x16-bit parallel IO port.

NOTE: The CRITICAL_IRQ_I input can be used for signaling critical system states like memory problems, power failure, etc..

6.5.1. Data / Instruction Memory Interface

The Atlas 2k must be connected to data and instruction memories to operate. When using separated memories for instruction and data, the instruction memory is read-only and need only one read port. The data memory requires for this kind of implementation a read and a write port. When using a common memory structure for instruction and data, the memory requires a single write port and two read ports. Furthermore, it is possible to connect the data and instruction memory interface to a bus unit with or without caches, to access a single read-port and write-port memory. However, the basic setup (**ATLAS_2K_BASE_TOP.vhd**) connects the Atlas 2k processor to a shared instruction/data interface.
Let's start with the instruction fetch interface. This interface is very simple to implement. It consists of the the instruction page (`MEM_I_PAGE_O`), the instruction address (`MEM_I_ADR_O`), the instruction word read back (`MEM_I_DAT_I`) and an enable signal (`MEM_I_EN_O`). The instruction page selects the current instruction memory bank. The instruction address outputs the current value of the CPU's program counter and thus determines the address of the next instruction. 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 (`MEM_I_EN_O`) goes low (inactive), the instruction memory is disabled and it has to hold the last instruction word, since the instruction memory output is also used as instruction register.

Signal name	Size (bit)	Direction	Function
MEM_I_PAGE_O	16	out	Page selection output, generated by the memory management unit (MMU)
MEM_I_ADR_O	16	out	Instruction address output (PC), only word-aligned addresses
MEM_I_EN_O	1	in	Instruction memory output enable. Instruction memory output must not alter when this signal is low!
MEM_I_DAT_I	16	out	Instruction data input

Table 22: Instruction interface

The data interface operates nearly in the same manner. Here, the enable signal (`MEM_D_EN_O`) indicates a valid read or write access to the data memory. Just like the instruction memory, the data memory has to keep the last data output if the enable signal goes low again. The page address (`MEM_D_PAGE_O`) selects the accessed memory page and the address (`MEM_D_ADR_O`) output specifies the actual address for the store/load operation. Write-data (`MEM_D_DAT_O`) is stored when the read/write select signal (`MEM_D_RW_O`) is high. If the signal is low, data is read from the memory and forwarded to the processor (`MEM_D_DAT_I`).

Signal name	Size (bit)	Direction	Function
MEM_D_EN_O	1	out	Data memory enable (valid access)
MEM_D_RW_O	1	out	Read ('0') or write ('1') access
MEM_D_PAGE_O	16	out	Page selection output, generated by the memory management unit (MMU)
MEM_D_ADR_O	16	out	Data address output
MEM_D_DAT_O	16	out	Write data output
MEM_D_DAT_I	16	in	Read data input

Table 23: Data interface

6.5.2. Paging / Memory Layout

Since the Atlas 2k is a 16-bit processor, it can only address 2^{16} bytes = 2^{15} words = 64kB directly. To overcome this memory limit, a paging scheme has been implement. This means, that the actual 16-bit address is extended with another 16-bit address, which specifies the accessed memory page. All in all, this scheme can address up to 2^{32} bytes = 4 GB and also enables the designer to create an operating system, where programs can be run independently in separate pages. Of course, the actual number of page and the page size itself can be modified corresponding to the application. The absolute maximum configuration is: 2^{16} pages (- boot ROM pages = 2^{15} pages) with 64kB memory space each. For more information see the chapter about the MMU.
To modify the number of pages or the page size, the construction of the actual memory address buses (I and D) - consisting of the page addresses (I and D page) and access addresses (also I and D) - must be adapted. When using the **ATLAS_2K_BASE_top.vhd** file as top entity, the number of pages and the page size can be configured via constants. In the following, some examples are presented to illustrate, how to construct the memory buses to setup the memory layout (VHDL syntax, '&' = concatenation).

NOTE: The memory is byte-addressed!

Example 1: 4 pages with 4kB each → total of 16kB memory; page selector width: 2 bit, page address width: 11 bit, memory address width: 13 bit

```
RAM_I_ADR(12:0) <= MEM_I_PAGE_O(1:0) & MEM_I_ADR_O(11:1);
RAM_D_ADR(12:0) <= MEM_D_PAGE_O(1:0) & MEM_D_ADR_O(11:1);
```

Example 2: 16 pages with 64kB each → total of 1MB memory; page selector width: 4 bit, page address width: 15 bit, memory address width: 19 bit

```
RAM_I_ADR(18:0) <= MEM_I_PAGE_O(3:0) & MEM_I_ADR_O(15:1);
RAM_D_ADR(18:0) <= MEM_D_PAGE_O(3:0) & MEM_D_ADR_O(15:1);
```

6.5.3. Coprocessor Interface

The coprocessor interface is dedicated to connected an external coprocessors (abbreviated as CP) directly to the Atlas 2k processor without the need of coupling it via some kind of system bus. This allows to create a small application specific system with a tightly coupled processing device, providing low data latency and thus high data transfer performance. The data communication between the CPU and the coprocessor 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.

Signal name	Size (bit)	Direction	Function
`CP_EN_O`	1	out	Valid access to coprocessor
`CP_ICE_O`	1	out	Coprocessor interface clock enable
`CP_OP_O`	1	out	Data transfer ('1') / data processing ('0')
`CP_RW_O`	1	out	Read ('0') / write ('1') access
`CP_CMD_O`	9	out	2..0: Command from `CDP` instruction 5..3: Operand B / source register address 8..6: Operand A / destination register address
`CP_DAT_O`	16	out	Coprocessor write data
`CP_DAT_I`	16	in	Coprocessor read data

Table 24: Coprocessor interface port of the Atlas 2k processor

The following graphic illustrates the interface architecture of a coprocessor. This interface allows writing and reading data to/from the device using the `MRC` and `MCR` instructions.

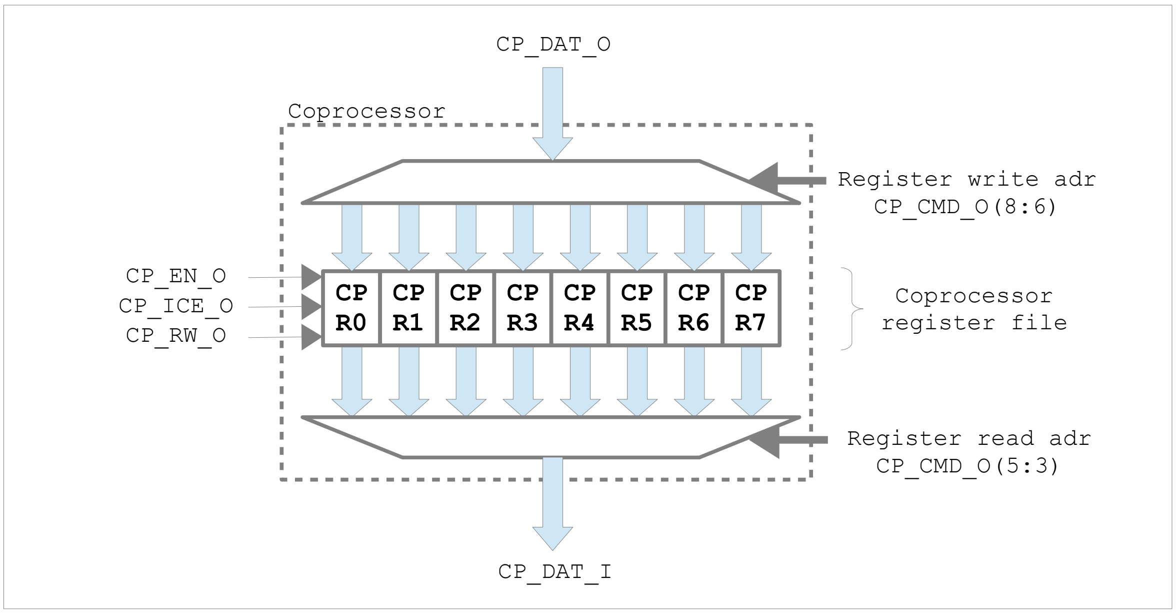


Figure 28: Coprocessor architecture for reading/writing data (transfers)

6.5.4. Wishbone Bus Interface

The Wishbone bus is an open-source bus specification for interconnecting several IP core within a system on chip. A copy of the implemented specifications can be found in the project's *doc* folder.

The Wishbone bus interface of the Atlas 2k processor is **NOT** used for instruction fetch or direct memory access via the memory-access instructions (`LDR`, `STR`). Instead, the Wishbone bus arbiter can be used by a program, which is executed from the internal processor memory, to access processor-external peripheral devices or memories. Since programs cannot be directly executed from a memory attached to the Wishbone bus, the program code has to be transferred to the internal RAM before. For instance, this can be done by a control instance like a minimal operating system.

6.6. Hardware Utilization

Here are some synthesis results for two different FPGA platforms.
→ The synthesis was done for the default Base Setup on 9th of April, 2014
→ Internal memory configuration: 4 pages with 4KB each
→ 64 Bytes Wishbone FIFO size (32 entries)

Xilinx Spartan XC3S400A	Atlas 2k Base Setup	
Number of Slices:	1394 / 3584	38%
Number of 4 input LUTs:	2443 / 7168	34%
Number of Slice Flip Flops:	1180 / 7168	16%
Number of IOs:	157	-
Number of BRAMs:	9 / 20	45%
Number of MULT18X18SIOs:	1 / 20	5%
Maximum Frequency:	**85.252 MHz**	

Table 25: Hardware utilization – Xilinx – Synthesis, speed optimized

Altera Cyclone IV EP4CE22F17C6N	Atlas 2k Base Setup	
Total logic element:	2730 / 22320	12%
Total combinatorial functions:	2508 / 22320	11%
Dedicated logic registers:	1198 / 22320	5%
Total pins:	157	-
Total memory bits:	294912 / 608256	48%
Embedded Multiplier 9-bit elements:	2 / 132	2%
Maximum Frequency:	**97.58 MHz**	

Table 26: Hardware utilization – Altera – Full implementation, slow 1200mV 0C model, no Wishbone devices

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 Manipulation
25	`ctrl_tf_store_c`	Store bit to mode-corresponding transfer flag

Bit #	Signal name	Function
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
		System Register Access
28	`ctrl_cp_acc_c`	Current operation is a coprocessor operation
29	`ctrl_cp_trans_c`	Coprocessor data transfer ('1') or coprocessor data processing operation ('0')
30	`ctrl_cp_wr_c`	Write access to coprocessor
31	`ctrl_cp_id_c`	Coprocessor ID bit ('1' for coprocessor #1, '0' for coprocessor #0)
		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 (SYSCALL instruction)
42	`ctrl_cmd_err_c`	Invalid/ undefined instruction or unauthorized access (command error trap)
43	`ctrl_ctx_down_c`	Switch down to user mode
44	`ctrl_restsm_c`	Restore saved operation mode
		Data Memory Access
45	`ctrl_mem_acc_c`	Perform data memory access
46	`ctrl_mem_wr_c`	Write ('1') or read ('0') access
47	`ctrl_mem_bpba_c`	Use bypassed base address
48	`ctrl_mem_daa_c`	Use delayed base address
		MAC Unit
49	`ctrl_use_mac_c`	Access the multiply-and-accumulate unit (if implemented)
50	`ctrl_load_mac_c`	Load an accumulation value to the MAC buffer
51	`ctrl_use_offs_c`	Use the loaded value to perform the actual MAC operation
		Other
52	`ctrl_sleep_c`	Go to sleep mode
53	`ctrl_cond_wb_c`	Is conditional write back

Table 27: CPU 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
ctrl_alu_cf_opt_c	ctrl_rd_2_c	Carry flag option for CPX (normal/ invert carry_in)
ctrl_alu_zf_opt_c	ctrl_rd_1_c	Zero flag option for CPX (AND/OR zero_in)

Table 28: CPU main control bus, signal reuse during pipeline process

7. Internal Coprocessor

The Atlas 2k includes several peripheral devices, which are combined into an internal coprocessor. Since this coprocessor is connected as **coprocessor #1**, it can only be accessed in system mode. Any other unprivileged access will trigger the undefined instruction trap. The different functional units are mapped together to modules (or “sub-coprocessors”) corresponding to their function in the system. The table below shows the different modules of the system coprocessor.

Coprocessor	Module	Name	Function
#1	c0	sys_0_core	Timer, LFSR, IRQ controller
	c1	sys_1_core	MMU, system information
	c2	com_0_core	UART, SPI, Parallel IO
	c3	com_1_core	Wishbone bus adapter

Table 29: Internal system coprocessor (#1) functional cores

The internal coprocessor can be accessed via the special 'coprocessor data transfer' operations (`MRC` and `MCR`) and each module is addressed by the `MRC/MCR` coprocessor register number (c0, …, c7). The actual register of the module is addressed via the command argument (#0, …, #7). The 'coprocessor data processing' operation (`CDP`) is only implemented for the c3-module yet. Any execution of it on any other module will not have any effect. The following example shows the read and write operations to move data between the coprocessor modules and the CPU:

```
MRC  #1, r4, c2, #7    ; copy parallel input to CPU register r4
MRC  #1, r5, c0, #2    ; copy timer counter register to CPU register r5

MCR  #1, c2, r0, #2    ; copy CPU register r0 to COM/SPI configuration register
MCR  #1, c1, r1, #2    ; copy CPU register r1 to MMU system-I-page register
```

7.1. Module c0 – System Controller 0

The system controller 0 (module 'c0') contains devices, which are mandatory for most applications. These devices are a high precision timer, a linear-feedback shift register and an interrupt controller. All in all, module c0 contains 8 16-bit wide registers, which are used for communication and configuration. The different devices as well as their interface/configuration register are about to be explained in this chapter.

7.1.1. Interrupt Controller

The internal interrupt controller supports 8 IRQ input channels. Each channel can be enabled/disabled and configured to trigger either on a voltage level (high/low) or on an edge (rising/falling). The first 6 IRQ channels are connected to internal devices, number 6 is reserved for future use and channel number 7 is forwarded to the `ATLAS_2K_TOP.vhd` entity (the Base Setup entity `ATLAS_2K_BASE_TOP.vhd` ties this signal to zero). The following table shows the channel mapping, the channel ID and the priorities.

Priority	IRQ channel	Connected device / port
`Highest`	0	Timer match interrupt
	1	Wishbone bus adapter IRQ
	2	UART data received interrupt
.	3	UART data transmission done interrupt
.	4	SPI transfer done interrupt
.	5	PIO input pin change interrupt
.	6	*reserved*
`Lowest`	7	External interrupt request pin "IRQ_I"

Table 30: Interrupt controller channels

Each channel of the IRQ controller can be enabled or disabled via the mask bits in the "irq_sm" register. The actual type of trigger can be set up by the the two config bytes of the "irq_conf" register. The low byte selects between level or edge triggering and the high byte specifies the actual level or edge type for the trigger. When ever a valid trigger occurs, the interrupt request is send via the **EXT_INT_1** pin to the CPU (handler base address 0x0004). The interrupt request handler then must read the "irq_sm" register to acknowledge the interrupt. Bits 2 down to 0 of this register specify the source of the corresponding interrupt request.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c0**	#0	irq_sm	2..0	R	Channel number of interrupt source (0..7), acknowledge IRQ on read access
				15..8	R/W	Enable bit mask for each channel (7..0)
		#1	irq_conf	7..0	R/W	Channel config 0: Level triggered ('1') or edge triggered ('0') channel (7..0)
				15..8	R/W	Channel config 1: High level/rising edge trigger ('1') or low level/falling edge trigger ('0') for channel (7..0)

Table 31: Interrupt controller register map

7.1.1.1. ASM Example – Setting up the IRQ controller

The following example code demonstrates how to set up a rising-edge trigger for the external interrupt line “IRQ_I(0)”. This code must be executed in system mode in order to access the system coprocessor.

```
LDIL r0, #0
LDIH r0, #0b01000000
MCR  #1, c0, r0, #0    ; set enable mask for channel 6 (IRQ_I(0))

LDIL r0, #0b00000000   ; set edge trigger for all channels
LDIH r0, #0b01000000   ; set rising edge for channel 6
MCR  #1, c0, r0, #1    ; set trigger config

LDSR r0
SBR  r0, r0, #11       ; set global IRQ enable flag in MSR
SBR  r0, r0, #13       ; enable IRQs from EXT_IN_1
STSR r0
```

The following example code show how to get the ID of the pending IRQ channel. This code must be executed in system mode in order to access the system coprocessor.

```
MRC   #1, r0, c0, #0   ; ack IRQ and get ID
LDIL  r1, #0x07        ; mask for ID bits
AND   r0, r0, r1       ; the IRQ ID in r0
```

7.1.2. High Precision Timer

The system controller contains a single 2x16-bit high precision counter for timing applications. The main clock speed of the processor drives an internal pre-counter. Whenever this pre-counter reaches the value in the pres-scaler “timer_prsc” register, the timer counter register “timer_cnt” will be incremented. If this counter registers matches the threshold value in the “timer_thr” register, the timer interrupt signal will be high for one cycle and the counter register is set to 0 to start over again.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c0**	#2	`timer_cnt`	15..0	R/W	Timer counter register
		#3	`timer_thr`	15..0	R/W	Timer threshold value
		#4	`timer_prsc`	15..0	R/W	Timer prescaler register

Table 32: High precision timer register map

The timer can be disabled by setting the “timer_thr” register to zero. The timer can be restarted at any time by clearing “timer_cnt”. Also, any write access to the “timer_prsc” or the “timer_thr” register will reset the counter.

NOTE: The timer “threshold match” interrupt goes high for one cycle whenever the TIMER_CNT register matches the TIMER_THR register. This IRQ is connected to the system interrupt controller on channel 0.

The interval for the timer IRQ to trigger can be obtained by the following formula:

$$T_{TimerIRQ} = \frac{TIMER_{PRSC}+1}{MAIN_{CLK}} TIMER_{THR}$$

Example: For a 5 second timer IRQ at a processor clock speed of 50MHz, set $TIMER_{PRSC}$ = 49999 = 0x3C4F and $TIMER_{THR}$ = 5000 = 0x1388.

7.1.3. Linear-Feedback Shift Register (LFSR)

Many applications require some kind of random numbers. With the internal *Galois* linear-feedback shift register, the processor can generate pseudo-random numbers automatically without any software overhead. To adapt the LFSR to the user's application, it is possible to configure the positions of the taps for the XOR computation. This is done by setting the “lfsr_poly” register. Bit 15 of this register configures the LFSR data update strategy: Setting this bit will let the LFSR operate in free-running mode, where new data is generated at the main clock frequency. Clearing this bit will let the LFSR only sample new data after a reading the “lfsr_data” register.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c0**	#5	`lfsr_data`	15..0	R/W	LFSR data register
		#6	`lfsr_poly`	14..0	R/W	LFSR polynomial / tap register
				15	R/W	LFSR update: '1': free running mode, '0': after read-access

Table 33: Linear-feedback shift register register map

NOTE: The “lfsr_poly” register should be set before setting the “lfsr_data” register.

7.2. Module c1 – System Controller 1 (MMU)

The Atlas 2k processor features a memory management unit (MMU). This MMU, implemented as system module c1 of the internal system coprocessor (coprocessor #1), enables the user to access a memory/IO space of up to 2^{32} bytes (4GB). Therefore, the actual data and instruction addresses from the CPU, which are 16-bit wide, are concatenated with another 2x16 bit, determining the accessible data and instruction page, to create 32-bit wide address for memory/IO access.

The MMU is accessed via the coprocessor interface and the coprocessor data transfer instructions (only in system mode). The following table shows all accessible registers of the MMU module.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	c1	#0	mmu_irq_base	15..0	R/W	Interrupt base page
		#1	mmu_sys_i_page	15..0	R/W	Current system mode instruction page
		#2	mmu_sys_d_page	15..0	R/W	Current system mode data page
		#3	mmu_usr_i_page	15..0	R/W	Current user mode instruction page
		#4	mmu_usr_d_page	15..0	R/W	Current user mode data page
		#5	mmu_i_page_link	15..0	R	Previous instruction page before IRQ
		#6	mmu_d_page_link	15..0	R	Previous data page before IRQ
		#7	mmu_sys_info	15..0	R	System information; every 1st read: clock speed high, every 2nd read: clock speed low (value comes from the clock speed configuration generic)

Table 34: MMU register map

7.2.1. Theory of Operation

The resulting 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 (I-page and D-page for user and system mode). Together with the data and instruction address buses from the CPU, which 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.

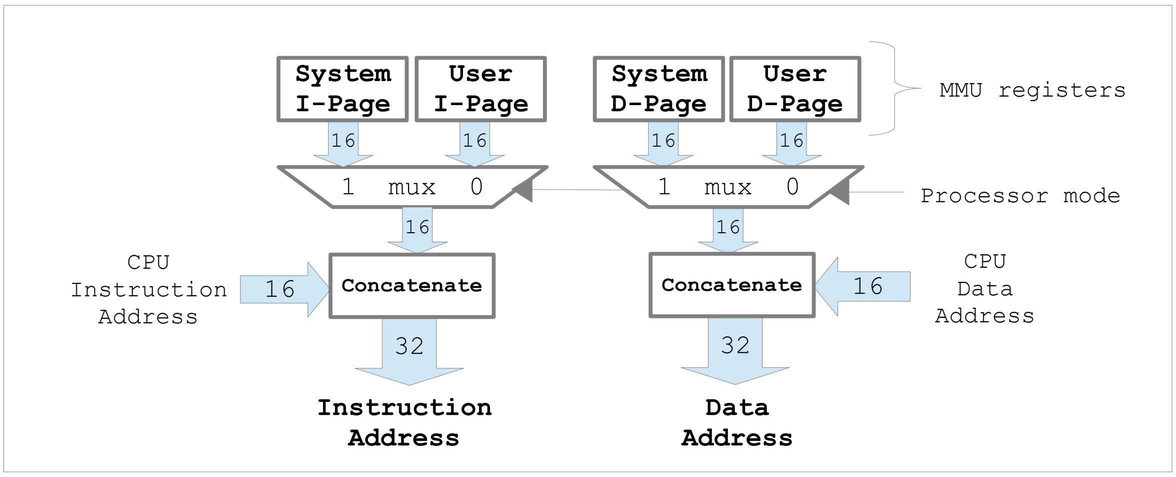


Figure 29: MMU address generation block diagram; the numbers in the arrows refer to the address widths

Whenever an interrupt or exception occurs, the *MMU_IRQ_BASE* register is automatically copied to the system I- and D-page registers (*MMU_SYS_I_PAGE* and *MMU_SYS_D_PAGE*) allowing the interrupt handler, which runs in system mode, to access the specified IRQ base page. Also, the last accessed I- and D-pages are store to the I- and D-link register (*MMU_I_PAGE_LINK* and *MMU_D_PAGE_LINK*). This makes it easy to restore the last accessed pages after an interrupt has been processed. The data of the I/D-link registers have just to be copied back to the system page registers when the interrupt handler has finished.

When writing data to the current I/D-page register (eg. the system mode registers when in system mode), it takes two cycles until the new page numbers affect the MMU output and thus the actual memory address buses. Thus, the I-page register must be set before performing a branch to an address within the new I-page.

7.2.1.1. ASM Example – MMU page and context switch

This examples shows how to start a program (in user mode), which is located at the beginning of data and instruction page "0x10AC".

```
LDIL r1, #0xAC                  ; load low byte of destination page
LDIH r1, #0x10                  ; load high byte of destination page
CLR  r0                         ; start address in new page is zero

MCR  #1, c1, r1, #4             ; MMU's user d-page register
;there must be no delay between the next two instructions!
MCR  #1, c1, r1, #3             ; MMU's user i-page register
GTU  r0                         ; copy entry address to PC → finalize branch
                                ; and switch to user mode
```

7.3. Module c2 - Communication Controller 0

Module c2 of the internal system coprocessor features three different mainstream communication interfaces:

1. Universal asynchronous receiver/transmitter (UART)
2. Serial peripheral interface (SPI)
3. Parallel input/output ports (PIO)

Again, all communication controller register can only be accessed in system mode.

7.3.1. Universal Asynchronous Receiver/Transmitter (UART)

The UART is a standard communication interface for all kind of applications. A simple UART with fixed frame setup and variable Baud value is implemented as part of the communication controller in module c2 of the system coprocessor.

Top Entity Signal	Direction	Size (bit)	Function
UART_RXD_I	Input	1	UART receiver input
UART_TXD_O	Output	1	UART transmitter output

Table 35: UART IO

Before you can use the UART for data receiving/sending, you have to wake up the transceiver from power-down mode by setting bit #6 in the *COM_CTRL* register.

The frame format is fixed to 8 data bits, no parity bit and 1 stop bit (8-N-1). The actual Baud rate can be configured using the $UART_{PRSC}$ register. The prescaler value is computed by the formula below, where $MAIN_{CLK}$ defined the processor clock frequency and *BAUD* the actual used Baud rate.

$$UART_{PRSC} = \frac{MAIN_{CLK}}{BAUD+15} \quad \text{and for the BAUD rate} \quad BAUD = \frac{MAIN_{CLK}}{UART_{PRSC}} - 15$$

The interface of the UART is based on the data register *UART_RTX_SD*. When writing to this register, the lowest 8 bit of the data is send via the UART TX channel. The user can check, if the UART is currently performing a transmission, by reading the transmitter busy flag in the *COM_CTRL* register. Whenever data is received via the UART receiver RX channel, the data ready flag in the *UART_RTX_SD* register is set and the received data can be obtained by reading the same register.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c2**	#0	uart_rtx_sd	7..0	R/W	Read: UART receive data Write: UART transmit data
				15	R	UART receiver data ready flag
		#1	uart_prsc	15..0	R/W	UART baud rate prescaler (UART$_{PRSC}$)
		#2	com_ctrl	5	R	UART transmitter busy flag
				6	R/W	UART activated when '1'
				7	R	UART RX buffer overflow

Table 36: UART controller register map

NOTE: The UART features two interrupt lines, connected to the system interrupt controller. The "data received" interrupt is connected to channel 2 and the "data sending done" interrupt is connected to channel 3. Both IRQs go high for one cycle when triggered.

7.3.1.1. ASM Example – setting UART Baud rate to 2400 @ 50MHz

```
      LDIL r1, #0xE0                    ; baud prescaler is 20704 (rounded)
      LDIH r1, #0x50                    ; = hex 0x50E0
      MCR  #1, c2, r1, #1               ; set UART prescaler

      ; activate UART controller
      MRC  #1, r0, c2, #2               ; get com control register
      SBR  r0, r0, #6                   ; set UART activate bit
      MRC  #1, c2, r0, #2               ; set com control register
```

7.3.1.2. ASM Example – get data from UART receiver, received data in r0

```
uart_receivebyte:
      MRC   #1, r0, c2, #0              ; get uart rx-status/data register
      STBI  r0, #15                     ; copy inverted uart rx_ready flag to T-flag
      BTS   uart_receivebyte            ; nothing received, keep on waiting
      LDIH  r0, #0x00                   ; clear upper byte
      RET   lr                          ; return to calling instance
```

7.3.1.3. ASM Example – send data via UART transmitter, transmitted data in r1

```
uart_sendbyte:
      MRC  #1, r0, c2, #2               ; get com control register
      STB  r0, #5                       ; copy uart tx_busy flag to T-flag
      BTS  uart_sendbyte                ; still set, keep on waiting
      MCR  #1, c2, r1, #0               ; send data
      RET  lr                           ; return to calling instance
```

7.3.2. Serial Peripheral Interface (SPI)

The SPI is a very common interface for connection a large variety of different devices, like SD-cards, EEPROMs, displays or AD/DA converter.

Top Entity Signal	Direction	Size (bit)	Function
SPI_MOSI_O	Output	8	Serial data output (8 channels)
SPI_MISO_I	Input	8	Serial data input (8 channels)
SPI_SCK_O	Output	8	Serial clock output (8 channels)
SPI_CS_O	Output	8	Chip select lines, **low-active** (8 channels)

Table 37: SPI IO

NOTE: The SPI controller also features a "transfer done" interrupt, which goes high for one cycle whenever an SPI transmission is done. The IRQ is connected to the system interrupt controller on channel 4.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c2**	#2	com_ctrl	0	R/W	SPI MSB first ('0'), LSB first ('1')
				1	R/W	SPI clock polarity
				2	R/W	SPI edge offset, '0' first edge, '1' second offset
				3	R	SPI busy flag
				4	R/W	SPI auto apply CS
				5	R	UART transmitter busy flag
				11..8	R/W	SPI data frame length (actual length is [11:8]-1)
				15..12	R/W	SPI clock prescaler (SPI_{PRSC})
		#3	spi_data	15..0	R/W	SPI receive/transmit data, start transfer on write-access
		#4	spi_cs	7..0	R/W	SPI channel chip select (a '1' will set the corresp. CS low)

Table 38: SPI controller register map

The SPI communication interface controller is based on a 16-bit shift register, thus a maximum of 16-bit data can be transferred at once. The actual data size per transaction can be set to a value between 1 and 16 bit. The direction of the transfer shift (left shift = MSB first or right shift = LSB first) ca also be configured via the "com_ctrl" register. There are four different SPI 'modes', which can be setup by the clock polarity and the edge offset. For mode 0 for example, both option must be set to zero. Whenever you are writing data to the "spi_data" register, an SPI transfer is started. By obtaining the SPI busy flag in the "com_ctrl" register, you can check when a transfer is done.

Before the transfer is initiated, the destination device must be selected by writing the corresponding chip select number to the "spi_cs" register, When the CS auto apply feature in the "com_ctrl" register is set, the CS signal will automatically be activated when the transfer is started. If the auto apply feature is disabled, the CS must be manually set and cleared before and after the transfer. This also allows to perform arbitrarily wide transfer operations. Note, that the CS lines are low-active, but will only become enabled, when setting the corresponding bit in the chip select register high. The SPI prescaler (bits 15:12 in "com_ctrl" register, SPI_{PRSC}) defines the clock speed (SPI_{CLK}) of the SPI serial clock line.

$SPI_{PRSC} = \log_2[\frac{MAIN_{CLK}}{2 \cdot SPI_{CLK}} - 1]$ and for the SPI serial clock frequency $SPI_{CLK} = \frac{MAIN_{CLK}}{2 + 2^{SPI_{PRSC}+1}}$

The actual frame size can be configured via the bits 11..8 of the "com_ctrl" register. A binary 0b1111 represents a 16-bit transfer, a 0b0111 an 8-bit transfer and a 0b0000 an 1-bit transfer (just examples). If you want to transfer larger frames, you must use the manual chip select function and break down the actual frame size to max. 16-bit long frames (see examples below).

7.3.2.1. ASM example – perform 8 bit SPI transfer

```
; 8-bit transfer from/to r1 from/to device on CS1
; prsc=3, MSB first, mode 0

LDIL  r0, 0b00010000            ; auto set CS, MSB first, mode 0
LDIH  r0, 0b00110111            ; prsc 3, length = 8 bit
MCR   #1, c2, r0, #2            ; SPI config
LDIL  r0, #2                    ; CS1
MCR   #1, c2, r0, #4            ; set CS

MOV   r0, r1                    ; copy tx data
BL    do_spi_trans              ; perform transfer (see below)
MOV   r1, r0                    ; copy rx data
```

7.3.2.2. ASM example – perform 32 bit SPI transfer

```
; 32-bit transfer from/to r2:r1 from/to device on CS2
; prsc=3, MSB first, mode 0

LDIL  r0, 0b00000000            ; manually set CS, MSB first, mode 0
LDIH  r0, 0b00111111            ; prsc 3, length = 16 bit
MCR   #1, c2, r0, #2            ; set SPI config

LDIL  r0, #4                    ; CS2
MCR   #1, c2, r0, #4            ; assert CS

MOV   r0, r2                    ; copy tx data part 1
BL    do_spi_trans              ; perform transfer (see below)
MOV   r2, r0                    ; copy rx data part 1

MOV   r0, r1                    ; copy tx data part 2
BL    do_spi_trans              ; perform transfer (see below)
MOV   r1, r0                    ; copy rx data part 2

CLR   r0
MCR   #1, c2, r0, #4            ; de-assert CS
```

7.3.2.3. ASM example – SPI transfer subroutine

```
      ; subroutine to initiate SPI transfer
      ; TX data in r0, RX data in r0 afterwards

do_spi_trans:
      MCR  #1, c2, r0, #3            ; set SPI data (r0) → start transfer

      ; wait for transmission end
do_spi_trans_wait:
      MRC  #1, r0, c2, #2            ; get status reg
      STB  r0, #3                    ; busy flag
      BTS  do_spi_trans_wait         ; still set?
      MRC  #1, r0, c2, #3            ; get received data in r0

      RET  lr                        ; return to calling instance
```

7.3.3. Parallel Input/Output Ports (PIO)

The parallel IO port features 16 inputs (*PIO_IN*) and 16 outputs (*PIO_OUT*). Another 8 input and output ports are provided by the system IO port (*SYS_IO*), but this ports should be reserved for the bootloader, since the in-build bootloader uses this port for status lights and boot strap configuration.

Top Entity Signal	Direction	Size (bit)	Function
PIO_OUT_O	Output	16	Parallel output data
PIO_IN_I	Input	16	Parallel input data
SYS_OUT_O	Output	8	Parallel output data for system/bootloader
SYS_IN_O	Input	8	Parallel input data for system/bootloader

Table 39: PIO IO

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c2**	#5	pio_in	15..0	R	Parallel input data (PIO_IN_I port)
		#6	pio_out	15..0	R/W	Parallel output data (PIO_OUT_O port)
		#7	sys_io	7..0	R	System input data (SYS_IN_I port)
				15..8	R/W	System output data (SYS_OUT_O port)

Table 40: PIO controller register map

NOTE: The parallel input port (*PIO_IN*) features a pin change IRQ, that goes high for one cycle whenever an input pin changes it state. This IRQ is connected to the system interrupt controller on channel 5.

7.4. Wishbone Bus Adapter

The Wishbone bus is a very cool open-source standard for on-chip interconnection of several devices. Via the Wishbone bus adapter, a SYSTEM mode program can access an up to 32-bit large address space – enough to add a lot of additional peripherals and large memories, too. The adapter implements the Wishbone b4 specifications (see doc folder) and features only pipelined transfers with a variable burst size.

Top Entity Signal	Direction	Size (bit)	Function
WB_CLK_O	Out	1	Bus main clock
WB_RST_O	Out	1	Bus main reset, synchronous, high-active
WB_ADR_O	Out	32	Address output
WB_SEL_O	Out	2	Byte select (always "11")
WB_DATA_O	Out	16	Write data output
WB_DATA_I	In	16	Read data input
WB_WE_O	Out	1	Write enable
WB_CYC_O	Out	1	Valid bus cycle
WB_STB_O	Out	1	Data/address strobe
WB_ACK_I	In	1	Acknowledge input
WB_ERR_I	In	1	Bus error

Table 41: Wishbone Adapter IO

The communication between the CPU and the Wishbone bus fabric is based on two independent FIFOs. Once the transfer (TX) FIFO is loaded, the actual transfer to the bus slave can be done in the background allowing the CPU to resume other applications. Also, after read-access has been configured and started, the adapter performs the data fetch from the bus net in the background, too.

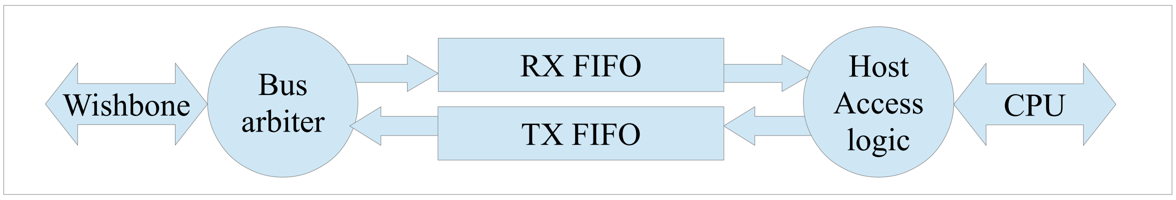


Figure 30: Wishbone bus adapter – functional diagram

A transfer is configured by first setting the burst size in the control register (*ctrl_reg*). The maximum burst size depends on the FIFO's depth, which is 32 words for default (can be changed in the ATLAS_pkg file). Then, the base address is set via the low and high *base_adr* registers. Afterwards, the address increment/decrement is set by the *adr_offs* register. This value is interpreted as 2's complement to allow up/down increment. The register can also be set to zero, for instance if you want to access a FIFO. Finally, the maximum cycle length is configured via the timeout_val register. For write transfers (CPU → Wishbone bus) you can now write data to the adapter's TX FIFO (*rxt_fifo* register). The actual transfer is initiated by executing the corresponding coprocessor command (read_transfer or write_transfer). For read accesses, the RX FIFO can be read when the transfer has finished.

NOTE: No write access to any register can be performed when a transfer is in progress.

To figure out when a transfer has finished, you can either check the busy flag in the adapter's control register *ctrl_reg* or you can enable the transfer done interrupt. This interrupt (connected to port 1 of the internal IRQ controller) will become high whenever a transfer has finished. Two additional interrupt sources are supported by the adapter: Whenever the WB_ERR_I signal becomes high, the current transfer is terminated and the bus error flag becomes active also triggering the adapter's IRQ line. Via the *timeout_val* register the maximum number of cycles for a transfer can be determined. If a transfer exceeds this maximum, the bus timeout flag becomes active, again triggering the adapter's IRQ line. Of course, the corresponding interrupt source has to be enabled via the interrupt enable bits in the control register. All interrupts are acknowledged at once when reading the control register and thus determining the actual interrupt source via the lowest three bits.

Coprocessor	Module	Register	Name	Bit(s)	R/W	Function
#1	**c3**	#0	ctrl	0	R	Transfer done flag (gets cleared on read-access)
				1	R	Bus error flag (gets cleared on read-access)
				2	R	Bus timeout flag (gets cleared on read-access)
				3	R/W	Transfer done interrupt request enable
				4	R/W	Bus error interrupt request enable
				5	R/W	Bus timeout interrupt request enable
				6	R	Busy flag (transfer in progress)
				7	R	Direction of current/last transfer (0: Bus read, 1: Bus write)
				15..8	R/W	Burst size (max size depends on FIFO size)
		#1	base_adr_l	15..0	R/W	Low part of bus base address
		#2	base_adr_h	15..0	R/W	High part of bus base address
		#3	adr_offs	15..0	R/W	Address offset for transfer (2's complement)
		#4	rtx_fifo	15..0	R	Read data from receiver FIFO
					W	Write data to transmitter FIFO
		#5	timeout_val	15..0	R/W	Maximal transfer length in cycles

Table 42: Wishbone bus controller register map

Only this functional core (the Wishbone Bus Adapter) features the execution of coprocessor command. This commands are used to initiate a transfer and also to specify the direction. For the coprocessor data processing, use the module “ID” as source/destination registers (→ **c3**; see example below).

Command code	Function
#0	Start READ transfer (Wishbone → CPU)
#1	Start WRITE transfer (CPU → Wishbone)
#2 .. #7	*reserved*

Table 43: Wishbone Adapter processing commands

Command usage:

```
    CDP   #1, c3, c3, #0       ; init read transfer
    CDP   #1, c3, c3, #1       ; init write transfer
```

7.4.1. ASM Example – 8 words burst read transfer with busy wait

```
      ; Configure Wishbone bus adapter
      LDIL  r0, #0b00000000           ; no interrupts, thanks
      LDIH  r0, #0b00000111           ; burst size = 8
      MCR   #1, c3, r0, #0            ; set CTRL reg

      LDIL  r0, #0xFF
      MCR   #1, c3, r0, #5            ; set timeout value to maximum (so we don't care)

      LDIL  r0, #0x00
      LDIH  r0, #0x0A
      LDIL  r1, #0xEE
      LDIH  r1, #0x20                 ; base address is 0x20EE0A00
      MCR   #1, c3, r0, #1            ; set BASE ADR L
      MCR   #1, c3, r1, #2            ; set BASE ADR H
      LDIL  r0, #2
      MCR   #1, c3, r0, #3            ; set offset = +2

      ; start transfer
      CDP   #1, c3, c3, #0            ; initiate read transfer

      ; wait for transfer to finish (busy wait)
wait: MRC   #1, r0, c3, #0            ; get CTRL reg
      STB   r0, #6                    ; extract busy flag
      BTS   wait                      ; wait until busy flag is cleared

      ; get RX FIFO and store to local buffer indexed by r5
      LDIL  r0, #8                    ; number of words to read = 8
get:  MRC   #1, r1, c3, #4            ; get rx fifo entry
      STR   r1, r5, +#2, post, !      ; store data to local buffer at [r5++]
      DECS  r0, r0, #1                ; decrement loop counter
      BNE   get                       ; repeat until all 8 words are done
```

8. Getting Started

To make things a little bit easier, I suggest to use the “basic setup” of the Atlas 2k processor. This design unit includes the actual Atlas 2k processor together with an internal RAM component. The user coprocessor slot is kept empty – you can fill it with some cool custom logic - and only the boot IO and the peripheral IO is propagated throughout the interface ports. Also, a pre-defined testbench and a waveform configuration file are available for the basic setup.

Of course you can use this basic setup also as starting point for your own SoC design - and yeah, this is what I recommend ;)

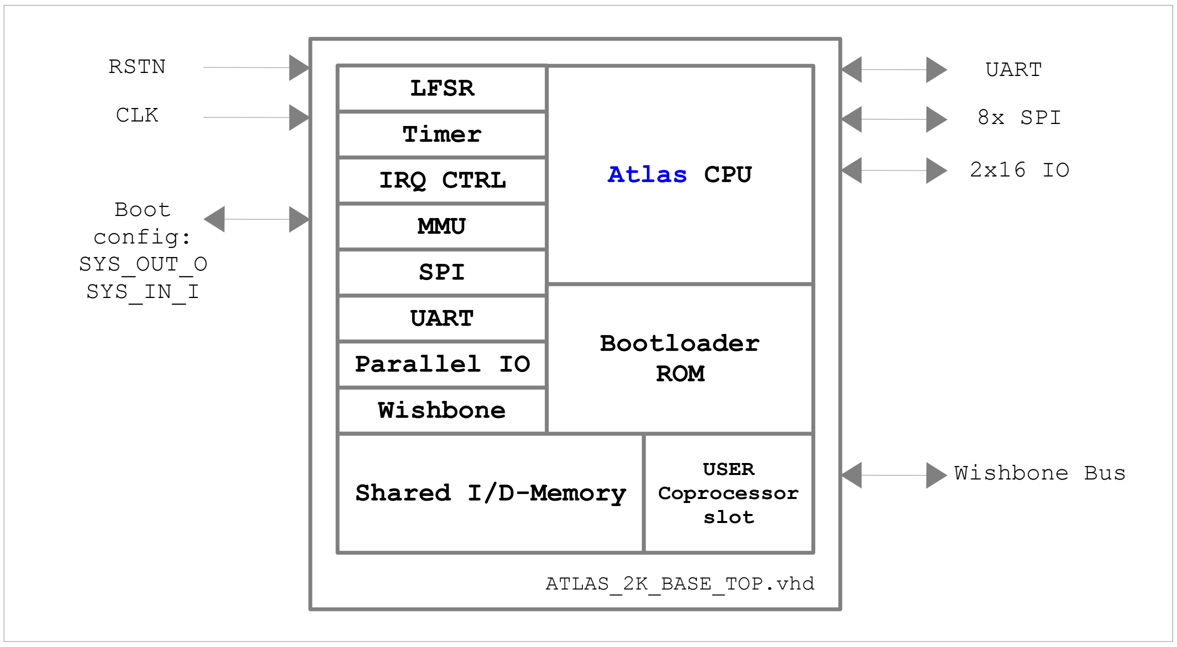


Figure 31: Atlas 2k Base Setup

Before you can use the basic setup for implementation or simulation, you have to configure the design. This is done by setting three VHDL constants (see below).

```
-- *** USER CONFIGURATION ***
-- **********************************************************************************************
   constant clk_speed_c : std_logic_vector(31 downto 0) := x"02FAF080"; -- clock speed in Hz
   constant num_pages_c : natural := 4; -- number of pages (must be a power of 2)
   constant page_size_c : natural := 4096; -- page size in bytes (must be a power of 2)
-- **********************************************************************************************
```

Atlas 2k Base Setup – User Configuration Constants

VHDL configuration constants:

- `clk_speed_c:` Clock speed (in Hz) of the `CLK_I` main clock in hexadecimal representation.
- `num_pages_c:` Number of pages – must be a power of two!
- `page_size_c:` Page size in bytes – must also be a power of two!

The total memory consumption is `num_pages_c * page_size_c`.

8.1. Necessary Files for the Atlas 2k Base Setup

The following table gives information about all the relevant files for the Atlas 2k Base Setup for synthesis and/or simulation

What	Atlas 2k Base Setup
Testbench:	*sim/atlas_2k_base_tb.vhd*
Xilinx ISIM demo waveform configuration:	*sim/xilinx_isim_atlas_2k_base_tb_wave.wcfg*
Top entity:	rtl/*ATLAS_2K_BASE_TOP.vhd*
rtl files	- rtl/ALU.vhd - rtl/ATLAS_2K_BASE_TOP.vhd - rtl/ATLAS_2K_TOP.vhd - rtl/ATLAS_CPU.vhd - rtl/ATLAS_pkg.vhd - rtl/BOOT_MEM.vhd - rtl/COM_0_CORE.vhd - rtl/COM_1_CORE.vhd - rtl/CTRL.vhd - rtl/INT_RAM.vhd - rtl/MEM_ACC.vhd - rtl/MEM_GATE.vhd - rtl/OP_DEC.vhd - rtl/REG_FILE.vhd - rtl/SYS_0_CORE.vhd - rtl/SYS_1_CORE.vhd - rtl/SYS_REG.vhd - rtl/SYSTEM_CP.vhd - rtl/WB_UNIT.vhd

Table 44: Atlas 2k Base Setup simulation / synthesis file guideline

8.2. Simulation

To easily evaluate and simulate a program for the Atlas 2k Base Setup, I have prepared a simple testbench and Xilinx ISIM © compatible pre-defined waveform. The testbench only includes the Atlas 2k Processor together with a reset and clock generator. Of course you can add additional modules for simulation. The clock generator is set at 50Mhz. You can change this value, but then you have to change the clock speed generic of the Atlas 2k, too.

When you want to simulate a program on the Atlas 2k, a big question occurs: Where does the memory image come from? There is no serial EEPROM or UART module implemented in the testbench, so the image has to be already in memory when the simulation starts. To do this, you can initialize the memory signal of the internal RAM (do this only for simulation, since it is not VHDL conform for implementation) with the ***init.vhd*** assembler output, which can be found after assembling in the ***asm*** folder.

```
--      ======================================================================
        signal MEM_FILE : int_mem_file_t;    -- use this for implementation
--      signal MEM_FILE : int_mem_file_t := -- use this for simulation only
--      (
--              others => x"0000" -- replace this with simulation memory content
--      );
--      ======================================================================
```

Cut-out of the INT_RAM.vhd file

To include the assembled program into the simulation environment, you first have to un-comment the second declaration of the `MEM_FILE` signal and comment out the first declaration of it. Then open the *asm/init.vhd* file and copy **all** of the content and paste it between the two brackets of the *MEM_FILE* signal initialization, replacing the red note. The result should look like this (of course the memory initialization image corresponds to your assembled program):

```
--      ======================================================================
--      signal MEM_FILE : int_mem_file_t;    -- use this for implementation
        signal MEM_FILE : int_mem_file_t := -- use this for simulation only
        (
                000000 => x"bc05", -- B
                000001 => x"bc00", -- B
                000002 => x"bc00", -- B
                000003 => x"bc00", -- B
                000004 => x"bc00", -- B
                000005 => x"2800", -- CLR
                000006 => x"ed0f", -- MCR
                000007 => x"c2b2", -- LDIL
                000008 => x"be09", -- BL
                000009 => x"ed0f", -- MCR
                000010 => x"3c00", -- SFT
                000011 => x"0001", -- INC
                000012 => x"c08f", -- LDIL
                others => x"0000"  -- NOP
        );
--      ======================================================================
```

INT_RAM.vhd file with memory init image

Now start the simulation of the testbench. The testbench sets the boot configuration to '11' to start the copied image directly from memory after start up.

8.3. Using the Atlas 2k Bootloader

The Atlas 2k processor contains a ROM with a powerful bootloader, which features several options for booting the processor. To reduce the circuitry of the address decoder, only bit 15 of the page address is checked to decide if the bootloader ROM or the 'normal' memory area is selected. Thus, all pages starting from ***0x8000*** will all access the bootloader page. The loader allows to resume execution of an image in internal RAM (page 0, address 0), to boot from a Wishbone device, the image download via UART or from an SPI EEPROM (CS 0) and even the programming of such an attached EEPROM. For all this operations, the bootloader does not need any RAM at all, so any image data in RAM will not be altered in any way.

NOTE: I suggest to use “Tera Term” or “Hterm” as terminal program.

The bootloader allows the selection of the boot option for the processor via a serial console (UART) or via the boot switch configuration, which is done by setting the lowest two bits of the SYS_IO_I port (bits 1:0).

Terminal console setup:

- 2400 Baud (slow, because of in-fly programming of EEPROM)
- 8 data bits
- no parity bit
- 1 stop bit

Boot configuration pins (**SYS_IO_I(1 downto 0)**):

- `'00'`: Start bootloader console via UART
- `'01'`: Boot from UART
- `'10'`: Boot from EEPROM (SPI EEPROM at CS 0)
- `'11'`: Boot from internal RAM memory (page 0, address 0x0000)

You need an EEPROM, that is compatible to Microchip ® SPI EEPROM like 25LC512 with 16-bit address and a 32-bit transfer frame size.

To access the bootloader via console, connect the processor via a COM port to a computer, configure the terminal (see above), set the boot configuration switch to “00” and reset the processor. The LSB of the `SYS_OUT_O` status output port will light up and the following menu should show up in your terminal:

```
Atlas-2K Bootloader - V20140414
by Stephan Nolting, stnolting@gmail.com
www.opencores.org/project,atlas_core

Boot page: 0x8000
Clock(Hz): 0x02FAF080

cmd/boot-switch:
 0/'00': Restart console
 1/'01': Boot UART
 2/'10': Boot EEPROM
 3/'11': Boot memory
 4: Boot WB
 p: Burn EEPROM
 d: RAM dump
 r: Reset
 w: WB dump
cmd:>
```

Atlas 2k bootloader console

Now you can enter an option of the list to perform the corresponding operation.

Bootloader console commands:

- **0**: Restart console (print menu again)
- **1**: Boot from UART (transfer image via UART directly into internal RAM)
- **2**: Boot from EEPROM (SPI EEPROM at SPI.CS0)
- **3**: Boot from memory (start image from memory system, page 0, address 0)
- **4**: Boot from Wishbone device[2] (specified by 32-bit address)
- **p**: Program EEPROM (program SPI EEPROM at CS 0 via UART)
- **d**: RAM dump (hex dump of any accessible memory page)
- **r**: Reset processor and restart bootloader
- **w**: Dump Wishbone network (by address (32-bit) and number of words)

NOTE: For any kind of image download to the processor or the attached EEPROM, use the “**out.bin**” file, generated by the assembler program (can be found in the same folder). The file must be transferred by the terminal program in RAW BYTE MODE, that means without any protocol or handshake data.

NOTE: The ASM source file of the bootloader can be found in the folder “*software/bootloader*”,

2 Wishbone device must offer a valid **boot image** → content of the assembler's “`boot_init.vhd`” file.

8.4. Let's Get It Started!

In the *software/examples/blink_demo* folder you can find a simple demo program (*blink_demo.asm*), which implements a 4-bit counter, that outputs it's state via the lowest 4 bit of the `SYS_OUT_O` port.

To make this program run, follow the following steps:

1. Create a new project with your EDA tool of choice (Xilinx ISE, Altera Quartus II,...) and add all the rtl files (see table above) of the processor to your project.

2. Select the `ATLAS_2K_BASE_TOP.vhd` as top entity. Also, set all the user configuration constants in this file corresponding to your setup (clock speed and memory layout).

3. Perform the project compilation and assign all input and output pins. Make sure to connect the `SYS_IN_I` port to some switches and the `SYS_OUT_O` to some high-active LEDs. Also do not forget to connect the UART RXD and TXD pins to your serial interface and assign reset and clock signals (the reset input is low-active by default).

4. Download the generated bitstream to your FPGA.

5. Start a terminal program (like HTerm), connect to the corresponding COM port, set the terminal setting to 2400-8-N-1 and open the connection. Make sure, your terminal does not use any kind of frame protocol for receiving/transmitting data (raw byte mode only).

6. Now it is time to assemble the program file (yes, we will assemble it, even if there is already an assembled version in the blink_demo folder ;)). Start a command prompt, navigate to the assembler folder *asm* and start the assembling of the demo program (here, we use "DEMO_PROG1" as final image name):

```
...\trunk\asm>atlas_asm ..\software\examples\blink_demo\blink_demo.asm DEMO_PROG1
```

7. Reset the Atlas 2k processor via the assigned reset pin. Make sure, the lowest two bit of the `SYS_IN_I` port are tied to ground (logical 0). This will tell the bootloader to start the terminal console.

8. The bootloader prompt should show up in your terminal program and the LED connected to the lowest bit of the `SYS_OUT_O` port should light up.

9. In your terminal program, press (and send) an ASCII '1', this will initiate the booting via UART. Now open the *out.bin* file in the *asm* folder (the assembled program) via your terminal program to download it to the Atlas 2k. This transfer must be done in raw byte mode – no command or protocol frame data must be added.

10. When the download has completed, the image is automatically started and you should see some pretty LED flashes.

11. **Troubleshooting:** Encountering any problems? Take a closer look at this chapter again, maybe you have forgotten something... If your setup still does not want to work, write me an E-mail and we will try to figure out the problem together.

8.5. Example Programs

The *software* folder includes sources of the integrated bootloader (*bootloader* folder) as well as as some example (*examples* folder) programs. In each example program folder you can find the assembler source/sources (files ending with ".asm"), an assembled binary file for bootloader upload ("out.bin") and an VHDL memory initialization file for simulation ("init.vhd). When there are several asm files in a folder, the file with the same name as the folder is the main assembler file (e.g. blink_demo.asm in the blink_demo folder).

Example folder	Description
`blink_demo`	4-bit up-counter, displayed on the LSBs of the SYS_OUT port
`random_numbers`	You can init the pseudo-random generator via a terminal and print random data.
`More to come... ;)`	

Table 45: List of example programs

9. Frequently Asked Questions (FAQ)

This chapter presents some questions, that I have been asked about this project as well as some questions, that might occur someday.

- ***Is there a 32-bit version of the Atlas 2k processor?***
  Theoretically, you can change the data size to 32-bit, so all kind of addressing and data computation is done using 32-bit values but instructions will still be 16-bit wide. This option is partly prepared, but not fully implemented/tested yet and therefore might not work. However, this 32-bit ode comes with some 'issues', since the Atlas CPU was designed for 16-bit data.

- ***I don't need all the peripheral interfaces of the Atlas 2k - how can I get rid of them?***
  Right now there aren't any configuration generics, which enable or disabled the actual synthesis of special functional cores. However, if you don't need a some of them, you can disconnect them from the outer world (tie inputs to zero, leave outputs opened). The synthesis tool will then eliminate most of that functional core, except for the host interface stuff – but that shouldn't be a problem since this interface logic only consumes a very small amount of hardware.

10. Appendix

10.1. System Coprocessor Modules

Module		Register		Bit(s)	R/W	Function
c0	sys_0_core	#0	irq_sm	2..0	R	IRQ source (0..7) and ACK IRQ on read-access
				15..8	R/W	Interrupt channel mask (channel 0..7)
		#1	irq_conf	7..0	R/W	IRQ channel config 0: '1' level triggered, '0': edge triggered
				15..8	R/W	IRQ channel config 1: '1' high/rising, '0' low/falling level/edge
		#2	timer_cnt	15..0	R/W	Timer counter register
		#3	timer_thr	15..0	R/W	Timer threshold value
		#4	timer_prsc	15..0	R/W	Timer clock prescaler
		#5	lfsr_data	15..0	R/W	Linear-feedback shift register (LFSR) seed/data
		#6	lfsr_poly	14..0	R/W	LFSR polynomial/tap register
				15	R/W	LFSR update: '1': free-running mode, '0': after every read-access
c1	sys_1_core	#0	mmu_irq_base	15..0	R/W	Interrupt base page
		#1	mmu_sys_i_page	15..0	R/W	System mode instruction page
		#2	mmu_sys_d_page	15..0	R/W	System mode data page
		#3	mmu_usr_i_page	15..0	R/W	User mode instruction page
		#4	mmu_usr_d_page	15..0	R/W	User mode data page
		#5	mmu_i_page_link	15..0	R	Linked instruction page
		#6	mmu_d_page_link	15..0	R	Linked data page
		#7	mmu_sys_info	15..0	R	Various system info (clock speed)
c2	com_0_core	#0	uart_rtx_sd	7..0	R/W	UART send/receive data
				15	R	UART data received flag
		#1	uart_prsc	15..0	R/W	UART baud prescaler
		#2	com_ctrl	0	R/W	SPI MSB first ('0'), LSB first ('1')
				1	R/W	SPI clock polarity
				2	R/W	SPI edge offset, '0': first edge, '1': second edge
				3	R	SPI busy flag
				4	R/W	SPI auto apply CS
				5	R	UART transmitter busy flag
				6	R/W	UART enabled when '1'
				7	R	UART RX buffer overflow
				11..8	R/W	SPI data length (actual transfer frame length is [11:8]+1)
				15..12	R/W	SPI clock prescaler
		#3	spi_data	15..0	R/W	SPI send/receive data
		#4	spi_cs	7..0	R/W	SPI chip select (channels 7..0)
		#5	pio_in	15..0	R	Parallel input data
		#6	pio_out	15..0	R/W	Parallel output data
		#7	sys_io	7..0	R	System/bootloader parallel input (boot config)
				15..8	R/W	System/bootloader parallel output (status)

Module		Register		Bit(s)	R/W	Function
c3	com_1_core	#0	ctrl_reg	0	R	Transfer done flag (gets cleared on read-access)
				1	R	Bus error flag (gets cleared on read-access)
				2	R	Bus timeout flag (gets cleared on read-access)
				3	R/W	Transfer done interrupt request enable
				4	R/W	Bus error interrupt request enable
				5	R/W	Bus timeout interrupt request enable
				6	R	Busy flag (transfer in progress)
				7	R	Direction of last/current transfer (0: Bus → CPU, 1: CPU → Bus)
				15..8	R/W	Burst size (maximal size depends on FIFO size)
		#1	base_adr_l	15..0	R/W	Low part of transfer base address
		#2	base_adr_h	15..0	R/W	High part of transfer base address
		#3	adr_offs	15..0	R/W	Address offset for burst transfer (2's complement)
		#4	rtx_fifo	15..0	R	Read data from Wishbone RX FIFO
					W	Write data to Wishbone TX FIFO
		#5	timeout_val	15..0	R/W	Maximum transfer length in cycles

Table 46: Processor functional cores ***(unlisted registers/bits are read as zero and are reserved for future use)***

10.2. System Coprocessor Commands

This table presents all implemented commands of the functional cores of the internal coprocessor (CP #1). Unlisted commands are reserved for future use and should not be used.

Module		Command (CMD)	Function	ASM Example
c3	com_1_core	#0	Initiate read transfer (CPU ← Wishbone bus)	CDP #1, c3, c3, #0
		#1	Initiate write transfer (CPU → Wishbone bus)	CDP #1, c3, c3, #1

Table 47: Processor functional cores – executable commands

10.3. Interrupt/Exception Vector Map

The following table shows the five basic exception/interrupt vectors of the Atlas 2k and the extended interrupt sources of the internal system IRQ controller (coprocessor #1, module c0).

Name	Base address	Priority	Source	
`reset_vec`	`0x0000`	**1**	Power-up / hardware reset signal	
`x_int0_vec`	`0x0002`	**2**	"critical IRQ" top entity signal	
`x_int1_vec`	`0x0004`	**3**	**CP1.C0: System interrupt controller** (use rising edge trigger for channels 1..6)	
			Channel / Priority	**Source**
			`0`	Timer match
			`1`	Wishbone bus adapter (buss error / bus timeout / transfer done)
			`2`	UART data received
			`3`	UART transmission done
			`4`	SPI transfer done
			`5`	PIO pin change
			`6`	*reserved*
			`7`	External IRQ_I signal of ATLAS_2k_TOP entity, not used in basic setup
`cmd_err_vec`	`0x0006`	**4**	Unauthorized access by an instruction (e.g. coprocessor #1 access in USER mode)	
`swi_vec`	`0x0008`	**5**	`SWI` instruction	

Table 48: Atlas 2k interrupts/exceptions

10.4. Instruction Set Summary

The following table presents all instructions, that are implemented by the Atlas 2k evaluation assembler. The following acronym definitions are used:

`Ra, Rb`	Operand registers (R0, … , R7) from the current register bank (USER/SYSTEM)
`Rd`	Destination register (R0, … , R7) from the current register bank (USER/SYSTEM)
Ra_{USR}, Rb_{USR}	Operand registers (R0, … , R7) from the USER register bank
Rd_{USR}	Destination register (R0, … , R7) from the USER register bank
Ra_{SYS}, Rb_{SYS}	Operand registers (R0, … , R7) from the SYSTEM register bank
Rd_{SYS}	Destination register (R0, … , R7) from the SYSTEM register bank
`Imm3`	3-bit unsigned immediate
`Imm4`	4-bit unsigned immediate
`Imm5`	5-bit unsigned immediate
`Imm8`	8-bit unsigned / signed immediate
`Imm9`	9-bit signed immediate
`Imm10`	10-bit unsigned immediate
`MI`	Memory indexing (pre / post)
`FS`	Flag set (`usr_flags, sys_flags(,` *`alu_flags`*`)` or full, if no argument is present)
`RFS`	Reduced flag set (`usr_flags, sys_flags`)
`STP`	Shift type (`SWP, ASR, ROL, ROR, LSL, LSR, RLC, RRC`)
`CP`	Coprocessor number (ID = #0 or #1)
`Ca, Cb`	Coprocessor register (C0, … , C7)
`CC`	3-bit immediate coprocessor command
`inFlags`	Flags, that are taken into account for computation
`outFlags`	Flags updated by a computation
`MSR`	Machine status register
`MODE`	Corresponds to bit #15 of the MSR
`USR`	System mode (MSR[15] = '1')
`SYS`	User mode (MSR[15] = '0')
`CPX_OPT`	Flag option (`C_ANDZ, NOTC_ANDZ, C_ORZ, NOTC_ORZ`)
`PC`	Program counter
`SP`	Stack pointer (R6 of current register bank)
`LR`	Link register (R7 of current register bank)
`C, N, O, T, Z`	Mode-corresponding ALU flags (**C**arry, **N**egative, **O**verflow, **T**ransfer, **Z**ero)

NOTE: Pseudo instructions, which are instructions constructed from other instructions, are printed in blue.

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
`ADC`	Rd, Ra, Rb	Add two registers with carry	Rd ← Ra + Rb + C	C	-	1
`ADCS`	Rd, Ra, Rb	Add two registers with carry and set flags	Rd ← Ra + Rb + C	C	C, N, O, Z	1
`ADD`	Rd, Ra, Rb	Add two registers	Rd ← Ra + Rb	-	-	1
`ADDS`	Rd, Ra, Rb	Add two registers and set flags	Rd ← Ra + Rb	-	C, N, O, Z	1
`AND`	Rd, Ra, Rb	Logical AND	Rd ← Ra AND Rb	-	-	1
`ANDS`	Rd, Ra, Rb	Logical AND and set flags	Rd ← Ra AND Rb	-	C, N, O, Z	1
`B`, `BAL`	Imm9	Relative branch always to “Imm9”	PC ← PC + Imm9*2	-	-	1\3
`BCS`	Imm9	Relative branch if carry cleared to “Imm9”	PC ← PC + Imm9*2	C	-	1\3
`BCS`	Imm9	Relative branch if carry set to “Imm9”	PC ← PC + Imm9*2	C	-	1\3
`BCSL`	Imm9	Relative branch if carry cleared to “Imm9” and link	PC ← PC + Imm9*2; LR ← PC + 2	C	-	1\3

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
BCSL	Imm9	Relative branch if carry set to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	C	-	1\3
BEQ	Imm9	Relative branch if equal to "Imm9"	PC ← PC + Imm9*2	Z	-	1\3
BEQL	Imm9	Relative branch if equal to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	Z	-	1\3
BGE	Imm9	Relative branch if greater or equal to "Imm9"	PC ← PC + Imm9*2	N, O	-	1\3
BGEL	Imm9	Relative branch if greater or equal to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	N, O	-	1\3
BGT	Imm9	Relative branch if greater than to "Imm9"	PC ← PC + Imm9*2	N, O, Z	-	1\3
BGTL	Imm9	Relative branch if greater than to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	N, O, Z	-	1\3
BHI	Imm9	Relative branch if unsigned higher to "Imm9"	PC ← PC + Imm9*2	C, Z	-	1\3
BHIL	Imm9	Relative branch if unsigned higher to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	C, Z	-	1\3
BIC	Rd, Ra, Rb	Bit clear	Rd ← Ra AND (!Rb)	-	-	1
BICS	Rd, Ra, Rb	Bit clear and set flags	Rd ← Ra AND (!Rb)	-	C, N, O, Z	1
BL, BALL	Imm9	Relative branch always to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	-	-	1\3
BLE	Imm9	Relative branch if less than or equal to "Imm9"	PC ← PC + Imm9*2	C, N, Z	-	1\3
BLEL	Imm9	Relative branch if less than or equal to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	C, N, Z	-	1\3
BLS	Imm9	Relative branch if unsigned lower or same to "Imm9"	PC ← PC + Imm9*2	C, Z	-	1\3
BLSL	Imm9	Relative branch if unsigned lower or same to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	C, Z	-	1\3
BLT	Imm9	Relative branch if less than to "Imm9"	PC ← PC + Imm9*2	N, O	-	1\3
BLTL	Imm9	Relative branch if less than to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	N, O	-	1\3
BMI	Imm9	Relative branch if negative to "Imm9"	PC ← PC + Imm9*2	N	-	1\3
BMIL	Imm9	Relative branch if negative to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	N	-	1\3
BNE	Imm9	Relative branch if not equal to "Imm9"	PC ← PC + Imm9*2	Z	-	1\3
BNEL	Imm9	Relative branch if not equal to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	Z	-	1\3
BOC	Imm9	Relative branch if overflow cleared to "Imm9"	PC ← PC + Imm9*2	O	-	1\3
BOCL	Imm9	Relative branch if overflow cleared to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	O	-	1\3
BOS	Imm9	Relative branch if overflow set to "Imm9"	PC ← PC + Imm9*2	O	-	1\3
BOSL	Imm9	Relative branch if overflow set to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	O	-	1\3
BPL	Imm9	Relative branch if positive to "Imm9"	PC ← PC + Imm9*2	N	-	1\3
BPLL	Imm9	Relative branch if positive to "Imm9" and link	PC ← PC + Imm9*2; LR ← PC + 2	N	-	1\3
BRAGE	Rb	Branch absolute to [Rb] if greater or equal	PC ← Rb	N, O	-	1\3
BRAGT	Rb	Branch absolute to [Rb] if greater than	PC ← Rb	N, O, Z	-	1\3
BRALE	Rb	Branch absolute to [Rb] if less than or equal	PC ← Rb	C, N, Z	-	1\3

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
BRALGE	Rb	Branch absolute to [Rb] if greater or equal, and link	PC ← Rb; LR ← PC + 2	N, O	-	1\3
BRALGT	Rb	Branch absolute to [Rb] if greater than, and link	PC ← Rb; LR ← PC + 2	N, O, Z	-	1\3
BRALLE	Rb	Branch absolute to [Rb] if less than or equal, and link	PC ← Rb; LR ← PC + 2	C, N, Z	-	1\3
BRALLT	Rb	Branch absolute to [Rb] if less than, and link	PC ← Rb; LR ← PC + 2	N, O	-	1\3
BRALT	Rb	Branch absolute to [Rb] if less than	PC ← Rb	N, O	-	1\3
BRALTS	Rb	Branch absolute to [Rb] if transfer set, and link	PC ← Rb; LR ← PC + 2	T	-	1\3
BRATS	Rb	Branch absolute to [Rb] if transfer set	PC ← Rb	T	-	1\3
BRRGE	Rb	Branch relative to [Rb] if greater or equal	PC ← PC + Rb	N, O	-	1\3
BRRGT	Rb	Branch relative to [Rb] if greater than	PC ← PC + Rb	N, O, Z	-	1\3
BRRLE	Rb	Branch relative to [Rb] if less than or equal	PC ← PC + Rb	C, N, Z	-	1\3
BRRLGE	Rb	Branch relative to [Rb] if greater or equal, and link	PC ← PC + Rb; LR ← PC + 2	N, O	-	1\3
BRRLGT	Rb	Branch relative to [Rb] if greater than, and link	PC ← PC + Rb; LR ← PC + 2	N, O, Z	-	1\3
BRRLLE	Rb	Branch relative to [Rb] if less than or equal, and link	PC ← PC + Rb; LR ← PC + 2	C, N, Z	-	1\3
BRRLLT	Rb	Branch relative to [Rb] if less than, and link	PC ← PC + Rb; LR ← PC + 2	N, O	-	1\3
BRRLT	Rb	Branch relative to [Rb] if less than	PC ← PC + Rb	N, O	-	1\3
BRRLTS	Rb	Branch relative to [Rb] if transfer set and link	PC ← PC + Rb; LR ← PC + 2	T	-	1\3
BRRTS	Rb	Branch relative to [Rb] if transfer set	PC ← PC + Rb	T	-	1\3
BTS	Imm9	Relative branch if transfer set to “Imm9”	PC ← PC + Imm9*2	T	-	1\3
BTSL	Imm9	Relative branch if transfer set to “Imm9” and link	PC ← PC + Imm9*2; LR ← PC + 2	T	-	1\3
CBR	Rd, Ra, Imm4	Clear bit in register	Rd ← Ra AND (!(1<<Imm4))	-	-	1
CDP	CP, Ca, Cb, CC	Coprocessor data processing (CC on Ca and Cb)	Ca ← Ca [CC] Cb @ CP	-	-	1
CLR	Rd	Clears a register (Rd = 0)	Rd ← Rd XOR Rd	-	-	1
CLRS	Rd	Clears a register (Rd = 0) and set flags	Rd ← Rd XOR Rd	-	C, N, O, Z	1
CMP(S)	Ra, Rb	Compare two registers	Flags ← Ra - Rb	-	C, N, O, Z	1
COM	Rd, Ra	Logical NOT	Rd ← Ra NAND Ra	-	-	1
COMS	Rd, Ra	Logical NOT and set flags	Rd ← Ra NAND Ra	-	C, N, O, Z	1
CPX(S)	Ra, Rb, CPX_OPT	Compare two registers with flags using flag option	Flags ← Ra – Rb, C, Z [CPX_OPT]	C, Z	C, N, O, Z	1
DEC	Rd, Ra, Imm3	Subtract a three-bit immediate from a register	Rd ← Ra - Imm3	-	-	1
DECS	Rd, Ra, Imm3	Subtract a three-bit immediate from a register and set flags	Rd ← Ra - Imm3	-	C, N, O, Z	1
EOR	Rd, Ra, Rb	Logical EXCLUSIVE OR	Rd ← Ra XOR Rb	-	-	1
EORS	Rd, Ra, Rb	Logical EXCLUSIVE OR and set flags	Rd ← Ra XOR Rb	-	C, N, O, Z	1
INC	Rd, Ra, Imm3	Add three-bit immediate to register	Rd ← Ra + Imm3	-	-	1

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
INCS	Rd, Ra, Imm3	Add three-bit immediate to register and set flags	Rd ← Ra + Imm3	-	C, N, O, Z	1
LDB	Rd, Ra, Imm4	Load T-flag to a register's bit	Rd ← (Ra[Imm4] ← T)	T	-	1
LDIH	Rd, Imm8	Load upper 8 bit with immediate	Rd[15:8] ← Imm8	-	-	1
LDIL	Rd, Imm8	Load and sign extend lower 8 bit with immediate	Rd[7:0] ← Imm8; Rd[15:8] ← Imm8[7]	-	-	1
LDPC	Rd	Move program counter to register	Rd ← PC	-	-	1
LDR	Rd, Ra/Imm3, MI	Load data from memory with indexing pre/post indexing (MI)	Rd ← MEM[Ra+Rb/Imm3]	-	-	1\2
LDSR	Rd, FS	Move machine status register flag set to register	Rd ← MSR[FS]	-	-	1
LDUB	Rd_{SYS}, Ra_{USR}	Load register from user bank	Rd_{SYS} ← Ra_{USR}	-	-	1
LDUBS	Rd_{SYS}, Ra_{USR}	Load register from user bank and set flags	Rd_{SYS} ← Ra_{USR}	-	C, N, O, Z	1
MCR	CP, Cd, Ra, CC	Store data to coprocessor (with command)	CP.Cd ← Ra, [CC]	-	-	1
MOV	Rd, Ra	Copy register Ra to Rd	Rd ← Ra + 0	-	-	1
MOVS	Rd, Ra	Copy register Ra to Rd and set flags	Rd ← Ra + 0	-	C, N, O, C	1
MRC	CP, Rd, Ca, CC	Load data from coprocessor (with command)	Rd ← CP.Ca, [CC]	-	-	1
MUL	Rd, Ra, Rb	Multiply Ra and Rb	Rd ← Ra * Rb	-	-	1
MV, MVAL	Rd, Rb	Copy Rb to Rd	Rd ← Rb	-	-	1
MVCC	Rd, Rb	Copy Rb to Rd if unsigned lower	Rd ← Rb		-	1
MVCS	Rd, Rb	Copy Rb to Rd if unsigned higher or same	Rd ← Rb		-	1
MVEQ	Rd, Rb	Copy Rb to Rd if equal	Rd ← Rb		-	1
MVGE	Rd, Rb	Copy Rb to Rd if greater than or equal	Rd ← Rb		-	1
MVGT	Rd, Rb	Copy Rb to Rd if greater than	Rd ← Rb		-	1
MVHI	Rd, Rb	Copy Rb to Rd if unsigned higher	Rd ← Rb		-	1
MVLE	Rd, Rb	Copy Rb to Rd if less than or equal	Rd ← Rb		-	1
MVLS	Rd, Rb	Copy Rb to Rd if unsigned lower or same	Rd ← Rb		-	1
MVLT	Rd, Rb	Copy Rb to Rd if less than	Rd ← Rb		-	1
MVMI	Rd, Rb	Copy Rb to Rd if negative	Rd ← Rb		-	1
MVNE	Rd, Rb	Copy Rb to Rd if not equal	Rd ← Rb		-	1
MVOC	Rd, Rb	Copy Rb to Rd if no overflow	Rd ← Rb		-	1
MVOS	Rd, Rb	Copy Rb to Rd if overflow	Rd ← Rb		-	1
MVPL	Rd, Rb	Copy Rb to Rd if positive or zero	Rd ← Rb		-	1
MVTS	Rd, Rb	Copy Rb to Rd if transfer flag set	Rd ← Rb		-	1
NAND	Rd, Ra, Rb	Logical NOT-AND	Rd ← Ra NAND Rb	-	-	1
NANDS	Rd, Ra, Rb	Logical NOT-AND and set flags	Rd ← Ra NAND Rb	-	C, N, O, Z	1
NEC	Rd, Ra	Compute negative of register with taking carry flag into account	Rd ← 0 - Ra - C	C	-	1
NECS	Rd, Ra	Compute negative of register with taking carry flag into account; and set flags	Rd ← 0 - Ra - C	C	C, N, O, Z	1
NEG	Rd, Ra	Compute negative of register	Rd ← 0 - Ra	-	-	1
NEGS	Rd, Ra	Compute negative of register and set flags	Rd ← 0 - Ra		C, N, O, Z	1
NOP	-	No operation	R0 ← R0 + 0	-	-	1
ORR	Rd, Ra, Rb	Logical OR	Rd ← Ra OR Rb	-	-	1

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
ORRS	Rd, Ra, Rb	Logical OR and set flags	Rd ← Ra OR Rb	-	C, N, O, Z	1
PEEK	Rd	Copy word from top of stack	Rd ← MEM[SP]	-	-	1
POP	Rd	Pop register from negative growing stack (normal case)	Rd ← MEM[SP+2]; SP ← SP + 2	-	-	2
POP+	Rd	Pop register from explicit positive growing stack	Rd ← MEM[SP-2]; SP ← SP - 2	-	-	2
PUSH	Ra	Push register on negative growing stack (normal case)	MEM[SP] ← Ra; SP ← SP - 2	-	-	1
PUSH+	Ra	Push register on explicit positive growing stack	MEM[SP] ← Ra; SP ← SP + 2	-	-	1
RBA, RBAAL	Rb	Branch always absolute to [Rb]	PC ← Rb	-	-	3
RBACC	Rb	Branch absolute to [Rb] if carry cleared	PC ← Rb	C	-	1\3
RBACS	Rb	Branch absolute to [Rb] if carry set	PC ← Rb	C	-	1\3
RBAEQ	Rb	Branch absolute to [Rb] if equal	PC ← Rb	Z	-	1\3
RBAHI	Rb	Branch absolute to [Rb] if unsigned higher	PC ← Rb	C, Z	-	1\3
RBAL, RBALAL	Rb	Branch always absolute to [Rb], and link	PC ← Rb; LR ← PC + 2	-	-	3
RBALCC	Rb	Branch absolute to [Rb] if carry cleared, and link	PC ← Rb; LR ← PC + 2	C	-	1\3
RBALCS	Rb	Branch absolute to [Rb] if carry set, and link	PC ← Rb; LR ← PC + 2	C	-	1\3
RBALEQ	Rb	Branch absolute to [Rb] if equal, and link	PC ← Rb; LR ← PC + 2	Z	-	1\3
RBALHI	Rb	Branch absolute to [Rb] if unsigned higher, and link	PC ← Rb; LR ← PC + 2	C, Z	-	1\3
RBALLS	Rb	Branch absolute to [Rb] if unsigned lower, and link	PC ← Rb; LR ← PC + 2	C, Z	-	1\3
RBALMI	Rb	Branch absolute to [Rb] if minus, and link	PC ← Rb; LR ← PC + 2	N	-	1\3
RBALNE	Rb	Branch absolute to [Rb] if not equal, and link	PC ← Rb; LR ← PC + 2	Z	-	1\3
RBALPL	Rb	Branch absolute to [Rb] if plus, and link	PC ← Rb; LR ← PC + 2	N	-	1\3
RBALS	Rb	Branch absolute to [Rb] if unsigned lower	PC ← Rb	C, Z	-	1\3
RBALVC	Rb	Branch absolute to [Rb] if overflow cleared, and link	PC ← Rb; LR ← PC + 2	O	-	1\3
RBALVS	Rb	Branch absolute to [Rb] if overflow set, and link	PC ← Rb; LR ← PC + 2	O	-	1\3
RBAMI	Rb	Branch absolute to [Rb] if minus	PC ← Rb	N	-	1\3
RBANE	Rb	Branch absolute to [Rb] if not equal	PC ← Rb	Z	-	1\3
RBAPL	Rb	Branch absolute to [Rb] if plus	PC ← Rb	N	-	1\3
RBAVC	Rb	Branch absolute to [Rb] if overflow cleared	PC ← Rb	O	-	1\3
RBAVS	Rb	Branch absolute to [Rb] if overflow set	PC ← Rb	O	-	1\3
RBR, RBRAL	Rb	Branch always relative to [Rb]	PC ← PC + Rb	-	-	3
RBRCC	Rb	Branch relative to [Rb] if carry cleared	PC ← PC + Rb	C	-	1\3

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
RBRCS	Rb	Branch relative to [Rb] if carry set	PC ← PC + Rb	C	-	1\3
RBREQ	Rb	Branch relative to [Rb] if equal	PC ← PC + Rb	Z	-	1\3
RBRHI	Rb	Branch relative to [Rb] if unsigned higher	PC ← PC + Rb	C, Z	-	1\3
RBRL, RBRLAL	Rb	Branch always relative to [Rb], and link	PC ← PC + Rb; LR ← PC + 2	-	-	3
RBRLCC	Rb	Branch relative to [Rb] if carry cleared, and link	PC ← PC + Rb; LR ← PC + 2	C	-	1\3
RBRLCS	Rb	Branch relative to [Rb] if carry set, and link	PC ← PC + Rb; LR ← PC + 2	C	-	1\3
RBRLEQ	Rb	Branch relative to [Rb] if equal, and link	PC ← PC + Rb; LR ← PC + 2	Z	-	1\3
RBRLHI	Rb	Branch relative to [Rb] if unsigned higher, and link	PC ← PC + Rb; LR ← PC + 2	C, Z	-	1\3
RBRLLS	Rb	Branch relative to [Rb] if unsigned lower, and link	PC ← PC + Rb; LR ← PC + 2	C, Z	-	1\3
RBRLMI	Rb	Branch relative to [Rb] if minus, and link	PC ← PC + Rb; LR ← PC + 2	N	-	1\3
RBRLNE	Rb	Branch relative to [Rb] if not equal, and link	PC ← PC + Rb; LR ← PC + 2	Z	-	1\3
RBRLPL	Rb	Branch relative to [Rb] if plus, and link	PC ← PC + Rb; LR ← PC + 2	N	-	1\3
RBRLS	Rb	Branch relative to [Rb] if unsigned lower	PC ← PC + Rb	C, Z	-	1\3
RBRLVC	Rb	Branch relative to [Rb] if overflow cleared, and link	PC ← PC + Rb; LR ← PC + 2	O	-	1\3
RBRLVS	Rb	Branch relative to [Rb] if overflow set, and link	PC ← PC + Rb; LR ← PC + 2	O	-	1\3
RBRMI	Rb	Branch relative to [Rb] if minus	PC ← PC + Rb	N	-	1\3
RBRNE	Rb	Branch relative to [Rb] if not equal	PC ← PC + Rb	Z	-	1\3
RBRPL	Rb	Branch relative to [Rb] if plus	PC ← PC + Rb	N	-	1\3
RBRVC	Rb	Branch relative to [Rb] if overflow cleared	PC ← PC + Rb	O	-	1\3
RBRVS	Rb	Branch relative to [Rb] if overflow set	PC ← PC + Rb	O	-	1\3
SBC	Rd, Ra, Rb	Subtract two registers with carry	Rd ← Ra - Rb - C	C	-	1
SBCS	Rd, Ra, Rb	Subtract two registers with carry and set flags	Rd ← Ra - Rb - C	C	C, N, O, Z	1
SBR	Rd, Ra, Imm4	Set bit in register	Rd ← Ra OR (1<<Imm4)	-	-	1
SFT	Rd, Ra, STP	Shift with 3-bit type STP	Rd ← shift(Ra, STP)	C	-	1
SFTS	Rd, Ra, STP	Shift with 3-bit type STP and set flags	Rd ← shift(Ra, STP)	C	C, N, O, Z	1
SLEEP	Imm9	Set CPU into sleep mode with 9-bit tag	-	-	-	1
SPR	Ra	Store parity of register to T-flag	T ← parity(Ra)	-	T	1
SPRI	Ra	Store inverted parity of register to T-flag	T ← NOT parity(Ra)	-	T	1
STAF	Imm5, RFS	Store immediate to MSR ALU flag set	MSR[RFS] ← Imm5	-	C, N, O, Z, T	1
STB	Ra, Imm4	Store bit to T-flag	T ← Ra[Imm4]	-	T	1
STBI	Ra, Imm4	Store inverted bit to T-flag	T ← NOT Ra[Imm4]	-	T	1
STBR	Ra, Rb	Store bit to T-flag (register-indexed)	T ← Ra[Rb(3:0)]	-	T	1
STBRI	Ra, Rb	Store inverted bit to T-flag (register-indexed)	T ← NOT Ra[R4(3:0)]	-	T	1

Mnemonic	Operands	Description	Operation	inFlags	outFlags	Cycles
`STPC,` `RET,` `GT`	Ra	Store register to program counter	PC ← Ra	-	-	1
`STPCI,` `RETI,` `GTI`	Ra	Store register to program counter and enable global external interrupts	PC ← Ra; MSR(11) ← '1'	-	-	1
`STPCIL,` `RETIL,` `RETL`	Ra	Store register to program counter, link and enable global external interrupts	PC ← Ra; LR ← PC+2; MSR(11) ← '1'	-	-	1
`STPCL,` `RETL,` `GTL`	Ra	Store register to program counter and link	PC ← Ra; LR ← PC+2	-	-	1
`STPCU,` `RETU,` `GTU`	Ra	Store register to program counter, change to user mode when in system mode	PC ← Ra; MODE ← USR	-	-	1
`STPCUI,` `RETUI,` `GTUI`	Ra	Store register to program counter, change to user mode when in system mode and enable global external interrupts	PC ← Ra; MODE ← USR; MSR(11) ← '1'	-	-	1
`STPCUIL,` `RETUIL,` `GTUIL`	Ra	Store register to program counter, change to user mode when in system mode and link, link and enable global external interrupts	PC ← Ra; MODE ← USR; LR ← PC+2; MSR(11) ← '1'	-	-	1
`STPCUL,` `RETUL,` `GTUL`	Ra	Store register to program counter, change to user mode when in system mode and link	PC ← Ra; MODE ← USR; LR ← PC+2	-	-	1
`STPCX,` `RETX,` `GTX`	Ra	Store register to program counter and switch back to previous mode	PC ← Ra; MODE ← MSR(14)	-	-	1
`STPCXI,` `RETXI,` `GTXI`	Ra	Store register to program counter, switch back to previous mode and enable global external interrupts	PC ← Ra; MODE ← MSR(14); MSR(11) ← '1'	-	-	1
`STPCXIL,` `RETXIL,` `GTXIL`	Ra	Store register to program counter, switch back to previous mode and link, link and enable global external interrupts	PC ← Ra; MODE ← MSR(14); LR ← PC+2; MSR(11) ← '1'	-	-	1
`STPCXL,` `RETXL,` `GTXL`	Ra	Store register to program counter, switch back to previous mode and link	PC ← Ra; MODE ← MSR(14); LR ← PC+2	-	-	1
`STR`	Rd, Ra/Imm3, MI	Store data to memory with indexing pre/post indexing (MI)	MEM[Ra+Rb/Imm3]← Rd	-	-	1\2
`STSR`	Ra, FS	Store register to machine status register flag set	MSR ← Ra[FS]	-	-	1
`STUB`	Rd_{USR}, Ra_{SYS}	Store register to user bank	Rd_{USR} ← Ra_{SYS}	-	-	1
`STUBS`	Rd_{USR}, Ra_{SYS}	Store register to user bank and set flags	Rd_{USR} ← Ra_{SYS}	-	C, N, O, Z	1
`SUB`	Rd, Ra, Rb	Subtract two registers	Rd ← Ra - Rb	-	-	1
`SUBS`	Rd, Ra, Rb	Subtract two registers and set flags	Rd ← Ra - Rb	-	C, N, O, Z	1
`SYSCALL`	Imm10	System call (software interrupt) with 10-bit tag	MODE ← SYS; LR_{sys} ← PC + 2; PC ← x"0008"	-	-	1
`TEQ(S)`	Ra, Rb	Logical AND-test	Flags ← Ra AND Rb	-	C, N, O, Z	1
`TST(S)`	Ra, Rb	Logical OR-test	Flags ← Ra OR Rb	-	C, N, O, Z	1

Table 49: Atlas 2k Instruction Set Reference Card