Atlas – X2 Processor

by Stephan Nolting

Proprietary Notice

Disclaimer

For any kind of feedback, feel free to contact me: stnolting@gmail.com

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

Table of Content

1. Project Overview

The Atlas-X2 processor includes all you need to setup a small and powerful system on chip. It is based around the Atlas Processor, which includes the Atlas CPU, a shared instruction/data cache, a memory management unit (MMU) and a Wishbone bus interface. The core itself is a true 16-bit RISC processor but due to the MMU it supports 32-bit bus addresses to access up to 4GB of memory/IO-space. Thus, the X2 is the perfect heart of your next application ;)

1.1. Processor Features

- ✔ Main processor: Atlas Core, 16-bit RISC processor, 5 pipeline stages
- ✔ Configurable shared I/D-cache, fully associative
- ✔ Memory management unit (MMU) to access up to 4GB IO / memory space; attached as coprocessor
- ✔ Multi-peripheral controller including several IO and system cores; attached as coprocessor
  - IO: UART, 16 x SPI, 16 x IN / 16 x OUT parallel IO, 8 channel PWM
  - System: 32-bit timer, interrupt controller, random number generator
  - One slot for custom IP (CIP) block)
- ✔ Wishbone-compatible pipelined bus interface supporting burst-transfers, 32-bit address, 16-bit data
- ✔ Configurable direct accessed address area to bypass cache (for special memories, IO devices, ...)
- ✔ Two external interrupt request signals
- ✔ Open-source design, completely described in behavioral, platform-independent **VHDL**

1.2. Project Folder Structure

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

- ➢ **atlas_x2/doc:** The complete Atlas-X2 data sheet (this document) can be found here.
- ➢ **atlas_x2rtl:** All necessary VHDL rtl files for the Atlas-X2 are located in this folder. Note: The source files of the Atlas Processor are located in the *core/rtl* folder, which need to be included into the X2 project.-
- ➢ **atlas_x2/sim:** The sim folder contains testbenches for the X2 Processor together with additional simulation components (like demo memory) and default Xilinx ISIM (c) waveform configurations.

1.3. VHDL File Hierarchy

The following table presents all necessary VHDL rtl files (+ path) of the Atlas-X2.
The top entity of the complete system is "*ATLAS_X2.vhd*".

Source path	File name	Description
atlas-x2/rtl	**ATLAS_X2.vhd**	→ **Atlas-X2 top entity**
	- MULTI_IO_CTRL.vhd	→ Top entity of the multi-IO controller
	- CIP_CORE.vhd	→ Dummy component, reserved for custom IP
	- INT_CORE.vhd	→ 8 channel interrupt controller
	- LFSR_CORE.vhd	→ Linear feedback shift register, pseudo random source
	- PIO_CORE.vhd	→ 16 x IN and 16 x OUT parallel IO controller
	- PWM_CORE.vhd	→ Pulse-width-modulation controller, 8 channels, 8 bit each
	- SPI_CORE.vhd	→ Serial peripheral interface controller, 16 chip select lines
	- TIMER_CORE.vhd	→ High-precision 32-bit system timer
	- UART_CORE.vhd	→ Universal asynchronous receiver/transmitter
core/rtl	- ATLAS_PROCESSOR.vhd	→ Top entity of the Atlas Processor
	- BUS_INTERFACE.vhd	→ Cache and Wishbone bus interface
	- MMU.vhd	→ Virtual address extension controller
	- ATLAS_CORE.vhd	→ To entity of the Atlas CPU
	- ATLAS_PKG.vhd	→ Atlas project package file
	- ALU.vhd	→ Arithmetical/logical unit, coprocessor 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 (MSR, PC), IRQ control
	- WB_UNIT.vhd	→ A Data write-back unit

Table 1: Project's VHDL files and hierarchy

1.4. Configuration Generics

The generics of the Atlas-X2 top entity are used to configure the main options of the processor. Thus, the system can be adapted to meet the requirements of your specific application.

Signal name	Width/type	Function
UC_AREA_BEGIN_G	32-bit std_logic_vector	First address of un-cached IO area
UC_AREA_END_G	32-bit std_logic_vector	Last address of un-cached IO area
BOOT_ADDRESS_G	32-bit std_logic_vector	Generic for configuring the core's boot address
SYNTH_SPI_G	boolean	Synthesize SPI core when “true”
SYNTH_CIP_G	boolean	Synthesize custom IP module when “true”
SYNTH_UART_G	boolean	Synthesize UART core when “true”
SYNTH_PIO_G	boolean	Synthesize parallel IO core when “true”
SYNTH_PWM_G	boolean	Synthesize PWM core when “true”
SYNTH_INT_G	boolean	Synthesize interrupt controller when “true”
SYNTH_LFSR_G	boolean	Synthesize pseudo-random number generator when “true”
SYNTH_TIME_G	boolean	Synthesize system timer when “true”

Table 2: Processor's top entity configuration generics

The UC_AREA_x generics specify a dedicated memory or IO area, that will always be directly accessed without buffering data in the processor's cache. Thus, additional IO devices or special memories should be mapped to this un-cached address space.

After a hardware reset, the core starts fetching the first instruction from the pre-defined boot address. Therefore, the BOOT_ADDRESS_G generic should determine the address of a non-volatile memory (ROM), containing for a example a bootloader.

Use the SYNTH_x generics to specify which IO / system cores shall be synthesized. Whenever a specific component is not synthesized, it's output ports are automatically tied to zero.

1.5. Interface Ports

The type of all signals/generics is **std_logic** or **std_logic_vector**, respectively. Tie all unused inputs to zero and leave all unconnected outputs 'open'.

Signal name	Width/type	Dir	Function
Global Control			
CLK_I	1-bit	IN	Global clock line, all registers trigger on the rising edge, 50% duty cycle
RST_I	1-bit	IN	Global reset signal, synchronized to CLK_I and high-active
Wishbone Bus			
WB_ADR_O	32-bit	OUT	Address output
WB_CTI_O	3-bit	OUT	Cycle tag identifier
WB_SEL_O	2-bit	OUT	Byte select signal (always “11”)
WB_TGC_O	1-bit	OUT	Cycle tag, indicating current operation mode of the CPU
WB_DATA_O	16-bit	OUT	Data output (write data)
WB_DATA_I	16-bit	IN	Data input (read data)
WB_WE_O	1-bit	OUT	Write enable
WB_CYC_O	1-bit	OUT	Valid cycle signal
WB_STB_O	1-bit	OUT	Data strobe signal
WB_ACK_I	1-bit	IN	Acknowledge signal
WB_HALT_I	1-bit	IN	Halt bus transaction
Peripheral Interface			
UART_RXD_I	1-bit	IN	UART receiver input (UART core)
UART_TXD_O	1-bit	OUT	UART transmitter output (UART core)
SPI_CLK_O	1-bit	OUT	SPI serial clock output (SPI core)
SPI_MOSI_O	1-bit	OUT	SPI serial data output (SPI core)
SPI_MISO_I	1-bit	IN	SPI serial data input (SPI core)
SPI_CS_O	16-bit	OUT	SPI chip select lines (SPI core)
PIO_OUT_O	16-bit	OUT	Parallel data output (PIO core)
PIO_IN_I	16-bit	IN	Parallel data input (PIO core)
PWM_O	8-bit	OUT	Pulse-width-modulated channel (PWM core)
IRQ_I	2-bit	IN	Interrupt request lines (INT core)
RND_I	1-bit	IN	External noise source, tie to '0' if not used (LFSR core)

Table 3: Processor's top entity interface ports

2. System Overview

The Atlas-X2 processor includes all you need to setup a small and powerful system on chip.
The Atlas Processor - including the Atlas CPU, a shared instruction/data cache, a memory management unit (MMU) and a Wishbone bus interface - is the heart of the X2 processor platform. The core itself is a true 16-bit RISC processor but due to the MMU it supports 32-bit bus addresses to access up to 4GB of memory/IO-space.

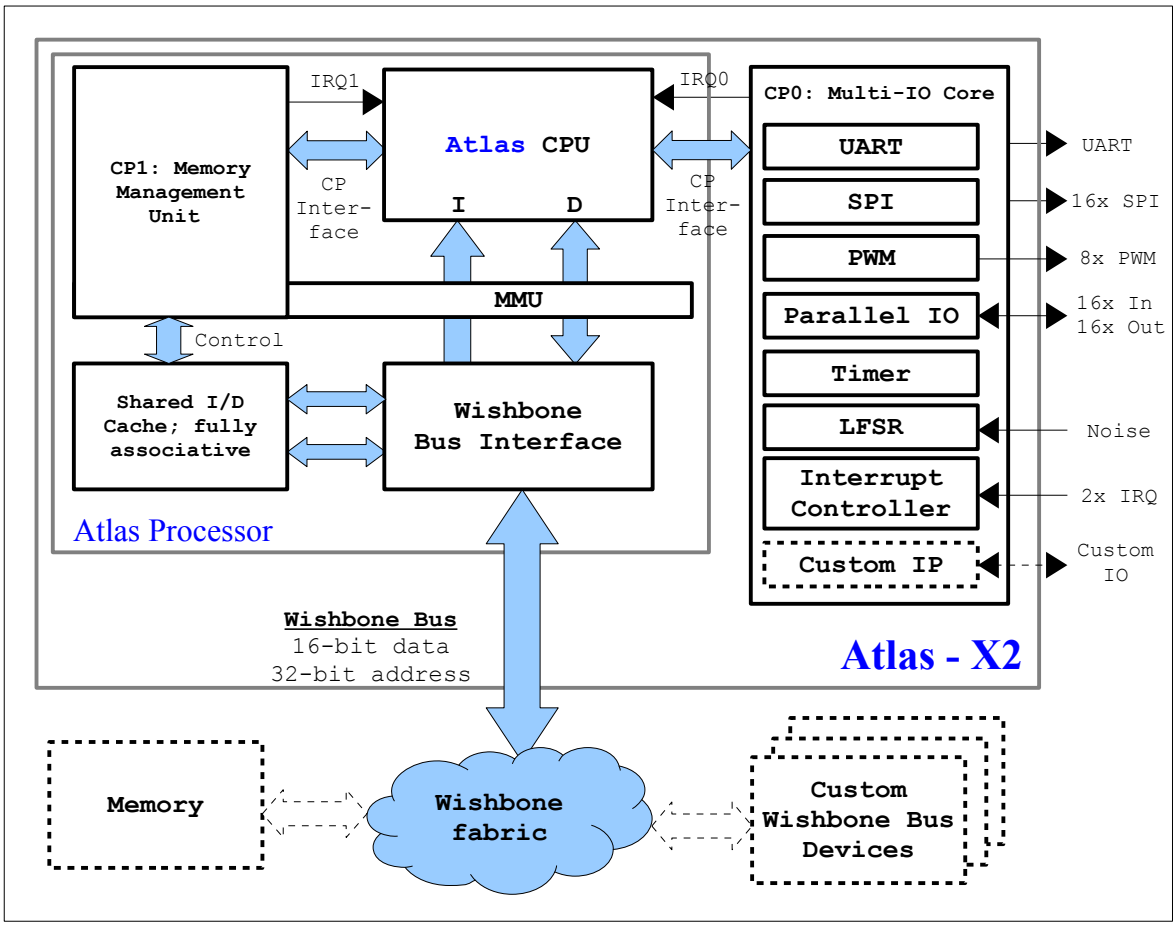

Figure 1: Atlas-X2 block diagram

For more information about the Atlas Processor or the Atlas CPU, please refer to the “Atlas Processor Data Sheet” in the *core/doc* folder.

2.1. Memory Management Unit

The MMU allows to access an IO / memory area of up to 4GB. Therefore, the complete 32-bit address space is separated into 2^{16} pages with 64kb each. The MMU generates the most significant 16 bits of this address by using page registers corresponding to the current operating mode of the CPU. See the Atlas Processor Datasheet for more information regarding the MMU.

The memory management unit is connected as coprocessor #1 to the Atlas CPU. Standard “coprocessor data transfer”- operations can be used to move data between the MMU and the CPU.

Register		Bit(s)	R/W	Function
C0	MMU_CTRL	0	W	Flush cache to sync memory with cache when '1'
		1	W	Clear cache; invalidate all entries when '1'
		2	R/W	Enable direct data access (bypass cache for data requests) when '1'
		3	R	Cache is sync to memory when '1'
		4	R/W	Bus error; acknowledge by writing a '1'
		5	R/W	Enable automatic page switch on CPU context change
		15..6	R/W	*Unused, should not be altered*
C1	MMU_SRATCH	15..0	R/W	Scratch register; free to use
C2	MMU_SYS_I_PAGE	15..0	R/W	Instruction page for system mode
C3	MMU_SYS_D_PAGE	15..0	R/W	Data page for system mode
C4	MMU_USR_I_PAGE	15..0	R/W	Instruction page for user mode
C5	MMU_USR_D_PAGE	15..0	R/W	Data page for user mode
C6	MMU_I_PAGE_LINK	15..0	R/W	Last accessed instruction page before an interrupt has occurred
C7	MMU_D_PAGE_LINK	15..0	R/W	Last accessed data page before an interrupt has occurred

Table 4: MMU Register Map

The MMU supports several commands to accelerate common MMU operations. These commands can be applied to any of the MMU's register, but will only have an effect when used in combination with “coprocessor data operations”. Using these commands in combination with “coprocessor data transfer operations” will not have any effect.

Command	
#0	Flush cache
#1	Clear cache
#2	Enable direct data access
#3	Disable direct data access
#4	Acknowledge MMU interrupt
#5	Copy link registers back to system page registers
#6	*Unused, execution has no effect*
#7	*Unused, execution has no effect*

Table 5: MMU Commands

2.2. Multi-IO-Controller

The Multi-IO-Controller features several common hardware interface like SPI and UART as well as extended IO controllers (parallel IO, PWM). Furthermore, system controllers like a timer, a linear-feedback shift register and an interrupt controller, are included.

The module (or core) is selected via the coprocessor register and the actual module register is selected by the coprocessor command immediate. Each register is 16-bit wide (bits 15..0). All not mentioned registers / bits are not accessible for write-access and will result 0 for read-accesses.

Module		Register		Bit(s)	R/W	Function
C0	**UART**	#0	RTX_DATA	7..0	R	UART receiver register
				7..0	W	UART transmitter register
		#1	STATUS	0	R/W	Enable receiver “data received” interrupt
				1	R/W	Enable transmitter “data send” interrupt
				2	R	Transmitter busy flag
				3	R	Received data flag
		#2	BAUD_PRSC	15..0	R/W	Baud rate prescaler = mainCLK/(BAUD+15)
C1	**SPI**	#0	RTX_DATA	15..0	R	SPI receiver register
				15..0	W	SPI transmitter register
		#1	CS_REG	15..0	R/W	SPI chip select line register
		#2	CONFIG	0	R/W	Enable core when '1'
				1	R/W	'0': MSB first, '1': LSB first
				2	R/W	Clock polarity, '1': idle high, '0': idle low
				3	R/W	Clock phase, '1' first edge, '0': second edge
				4	R/W	Chip select active state, '1': high-active, '0': low-active
				5	R	Transceiver busy flag
				6	R/W	Transmission done interrupt flag
				10..7	R/W	Transmission data length (1..16 bits)
				14..11	R/W	SPI clock prescaler = $\log_2$((mainCLK/(2*SPI_CLK))-1)
C2	**PWM**	#0	PWM_CH0	7..0	R/W	PWM channel 0 duty cycle = (1/255)*100%
		#1	PWM_CH1	7..0	R/W	PWM channel 1 duty cycle = (1/255)*100%
		#2	PWM_CH2	7..0	R/W	PWM channel 2 duty cycle = (1/255)*100%
		#3	PWM_CH3	7..0	R/W	PWM channel 3 duty cycle = (1/255)*100%
		#4	PWM_CH4	7..0	R/W	PWM channel 4 duty cycle = (1/255)*100%
		#5	PWM_CH5	7..0	R/W	PWM channel 5 duty cycle = (1/255)*100%
		#6	PWM_CH6	7..0	R/W	PWM channel 6 duty cycle = (1/255)*100%
		#7	PWM_CH7	7..0	R/W	PWM channel 7 duty cycle = (1/255)*100%

Module		Register		Bit(s)	R/W	Function
C3	**PIO**	#0	OUTPUT	15..0	R/W	Parallel output data
		#1	INPUT	15..0	R/W	Parallel input data
		#2	INT_EN	15..0	R/W	Enable interrupt (bit# corresponds to input pin#)
		#3	INT_CONF0	15..0	R/W	Interrupt type 0 of pin# ('0': level '1': edge triggered)
		#4	INT_CONF1	15..0	R/W	Interrupt type 1 of pin# ('0': low/falling, '1': high/rising)
		#5	INT_SRC	15..0	R	Interrupt source (pin-bit)
C4	**CIP**	#0	CUSTOM_R0	15..0	R/W	Custom IP register 0
		#1	CUSTOM_R1	15..0	R/W	Custom IP register 1
		#2	CUSTOM_R2	15..0	R/W	Custom IP register 2
		#3	CUSTOM_R3	15..0	R/W	Custom IP register 3
		#4	CUSTOM_R4	15..0	R/W	Custom IP register 4
		#5	CUSTOM_R5	15..0	R/W	Custom IP register 5
		#6	CUSTOM_R6	15..0	R/W	Custom IP register 6
		#7	CUSTOM_R7	15..0	R/W	Custom IP register 7
C5	**LFSR**	#0	LFSR_DATA	15..0	R/W	LFSR data register / seed
		#1	POLYNOMIAL	15..0	R/W	Polynomial to determine LFSR taps
		#2	CONTROL	0	R/W	Enable LFSR core
				1	R/W	'1': new data after read-access, '0': “free running mode”
C6	**TIMER**	#0	COUNT	15..0	R/W	Timer counter register
		#1	THRES	15..0	R/W	Threshold value, when COUNT = THRES → toggle IRQ
		#2	PRESCALE	15..0	R/W	COUNT prescaler
C7	**INT**	#0	IRQ_SRC	2..0	R	ID (=IRQ#) of last activated IRQ line, read = IRQ_ACK
		#1	MASK	7..0	R/W	IRQ channel enable
		#2	INT_CONF0	7..0	R/W	Interrupt type 0 of pin# ('0': level '1': edge triggered)
		#3	INT_CONF1	7..0	R/W	Interrupt type 1 of pin# ('0': low/falling, '1': high/rising)

Table 6: Multi-IO Controller Address Map

3. Program Examples

This chapter shows some example codes to explain the usage of the of the IO controller cores. The provided codes are only fragments and might need to be implemented in a “real program” to operate properly. Note, that the examples might be executed in system mode when the IO controller core is protected (MSR configuration).

3.1. UART – Universal Asynchronous Receiver / Transmitter

The UART (module c0) is a common interface for embedded system. The code examples show how to send or receive data via the UART. Even if “TX done” and “RX available” interrupts are available, the following examples use “busy wait” strategies.

3.1.1. Transmit a Byte

```
      ;Transmit low byte in r0 via UART
      ;Required registers: r0, r1

uart_sendbyte:
             mrc  #0, r1, c0, #1       ; get uart status register
             stb  r1, #2               ; copy uart tx_busy flag to T-flag
             bts  uart_sendbyte        ; still set, keep on waiting
             mcr  #0, c0, r0, #0       ; copy r0 to uart transceiver reg
             ret  r7                   ; go back to [r7] = link register
```

3.1.1. Receive a Byte

```
      ;Wait for a byte from UART and return it in r0 (low byte)
      ;Required registers: r0

uart_readbyte:
             mrc  #0, r0, c0, #1       ; get uart status register
             stbi r0, #3               ; copy inverted uart rx_ready flag to T-flag
             bts  uart_readbyte        ; nothing received, keep on waiting
             mrc  #0, r0, c0, #0       ; copy uart transceiver reg to r0
             ret  r7                   ; go back to [r7] = link register
```