Minimal OpenRISC System on Chip

Author: Raul Fajardo
rfajardo@gmail.com

Rev. 1.1

September 23, 2010

Copyright (C) 2010 Raul Fajardo

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license should be included with this document. If not, the license may be obtained from www.gnu.org, or by writing to the Free Software Foundation.

This document is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

History

Rev.	Date	Author	Description
1.0	02/01/10	Raul Fajardo	First Draft
1.1	09/23/10	Raul Fajardo	First Revision

Table of Contents

INTRODUCTION 5
1.1 System Overview 5
1.2 System Features 6
MINSOC ARCHITECTURE 8
2.1 OR1200 OpenRISC Implementation 9
2.2 Wishbone Interconnect 10
2.2.1 Configuring the Interconnect 10
2.2.2 Attaching Modules to the System 12
2.3 Start-Up Modules 12
FIRMWARE 14
3.1 Bootstrapping: a tiny bootloader 14
3.2 Communication to SoC Modules 15
3.3 Interrupt Processing 16
3.3.1 Interrupt Processing on MinSoC 17
SIMULATION 18
4.1 Initializing Memory 19
4.2 Bus Functional Models 19
CONCLUSION & FUTURE STEPS 21

1 Introduction

MinSoC consists only of the minimal requirements for an implementation using the OpenRISC processor. This documentation aims to support its use and applicability as a base for custom projects. As an open source project, every part of it can be uncovered and analyzed. However, without guidelines and explanation of what is intended with its design, this task can be very difficult. Therefore, the following document gives an overview of the project and explains its design goals and major details in order to allow its user to adapt and extend MinSoC to fit his needs.

1.1 System Overview

The Minimal OpenRISC System on Chip is a system on chip (SoC) implementation with standard IP cores available at OpenCores. This implementation consists of a standard project comprehending the standard IP cores necessary for a SoC embedding the OpenRISC implementation OR1200.

This project idea is to offer a synthesizable SoC which can be uploaded to every FPGA and be compatible with every FPGA board without the requirement of changing its code. In order to deliver such a project, the project has been based on a standard memory implementation and the Advanced Debug System, which allows system debug and software upload with the same cables used for FPGA configuration.

The adaptation of the project to a target board is made in 2 steps maximum. First, the “minsoc_defines.v” file has to be adjusted, generally one has to only uncomment his FPGA manufacturer and FPGA model definitions. After that, a constraint file for your specific pinout has to be created. Constraint files for standard boards can be found in the backend directory of the project.

Furthermore, the project offers working testbench and firmwares for its SoC. The current testbench can be run out of the box using Icarus Verilog v. 9.1. The firmwares are nearly the same of those of orpsocv2 [BJ2009]. The differences are for now, that the

known UART "hello world" example now runs with interrupts and a new Ethernet example has been added to it.

To complete, an on-chip memory instance is provided to embed the CPU's firmware. The size of this memory can be adapted defining its address width inside of the same minsoc_defines.v file, affecting simulation and synthesis equally. This enables the customization of the SoC to the available resources of the target FPGA, for general purposes, or to the memory amount required by the target firmware, for custom implementation, e.g. ASIC.

An overview about the complete SoC and its external connections is on Figure 1.

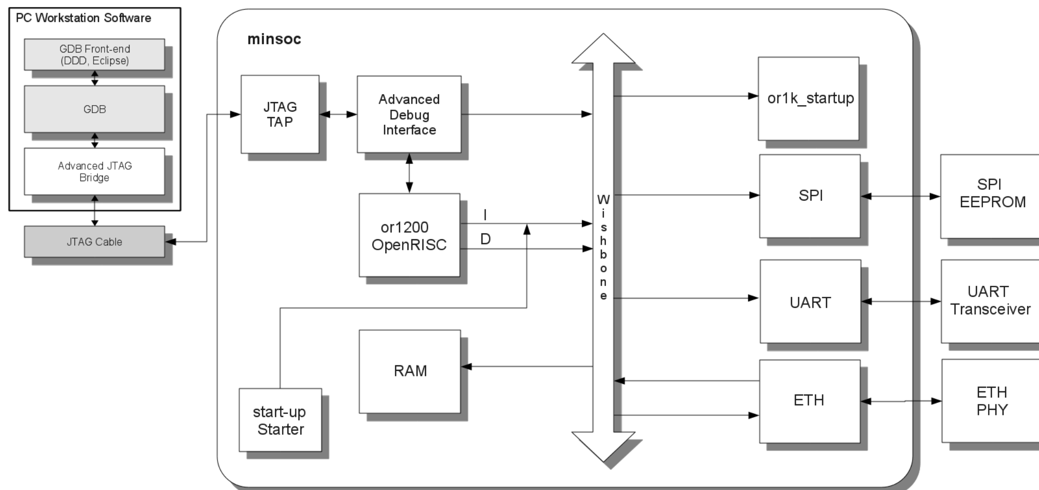

Figure 1: Overview of MinSoC: arrows leaving a module signalize a module's wishbone master interface, arrows pointing to a module signalize a wishbone slave interface. Double-sided arrows are different connections.

1.2 System Features

- OR1200 OpenRISC implementation
- Resizable on-chip memory
- System frequency selection
- JTAG debug featuring a multitude of cables

- Start-up option to automatically load your firmware on start-up from an external SPI memory
- UART and Ethernet modules
- FPGA generic and specific code (Xilinx & Altera) for memory, clock adaptation (PLLs and DCMs) and JTAG tap
- System configuration in a single definition file
- Example firmwares using UART and Ethernet
- Testbench included, simulating target software and system

2 MinSoC Architecture

MinSoC has been designed so that minimal changes are necessary in order to synthesize it for FPGAs of varied vendors. At the design stage, it has been decided that RAM blocks present in the FPGA will serve as SoC memory for the CPU's firmware, RAM module of Figure 1. Reasons for that decision are the availability of memory blocks on every FPGA and not requiring external memory to provide a working SoC.

Instantiation of on-chip memories is different from device to device, making them implementation dependent. A generic implementation would imply on the synthesizer using logic blocks to create memory, however there are mostly not enough logic blocks to support so much memory. Therefore, MinSoC includes implementations of most Altera and Xilinx FPGA memory blocks. The FPGA manufacturer and model configuration of the “minsoc_defines.v” file selects the correct implementation of the on-chip memory.

The JTAG tap and an included clock divider can be optionally selected to be target specific, having the advantage of higher speed and smaller size. The selection between FPGA generic or specific instances can be made through the project definition file, “minsoc_defines.v”. By selecting the FPGA specific modules, hardware modules already implemented on the FPGA chip will be used instead of logic created ones. This has generally the advantage of higher speed and smaller size. Moreover, a FPGA specific JTAG tap allows the upload and debug of software through the same connection used to configure the FPGA, while the generic JTAG requires the assignment of extra FPGA pins, which must be then connected to the JTAG cable. Lastly, the included generic clock divider can only divide the clock by even numbers. FPGA specific clock adaptation on the other side does not only divide the clock by any number, but also allows for cleaner clock signals.

The extra modules UART, Ethernet (ETH) and Start-Up (start-up Starter, or1k_startup and SPI) can be excluded from MinSoC by commenting their definitions on the file “minsoc_defines.v”. Start-up and Ethernet are commented by default.

2.1 OR1200 OpenRISC Implementation

The OR1200 implementation [OR1200] is the core of the Minimal OpenRISC System on Chip. The CPU is a 32-bit scalar RISC with Harvard microarchitecture Figure 2. Therefore, it has primarily two memory interfaces, an instruction interface and a data interface. Through the instruction interface, the CPU fetches instructions from memory to be executed, while the data interface stores/loads data on/from memory or sets/retrieves registers on/from other modules. These are 32-bit wide Wishbone interfaces [HR2002].

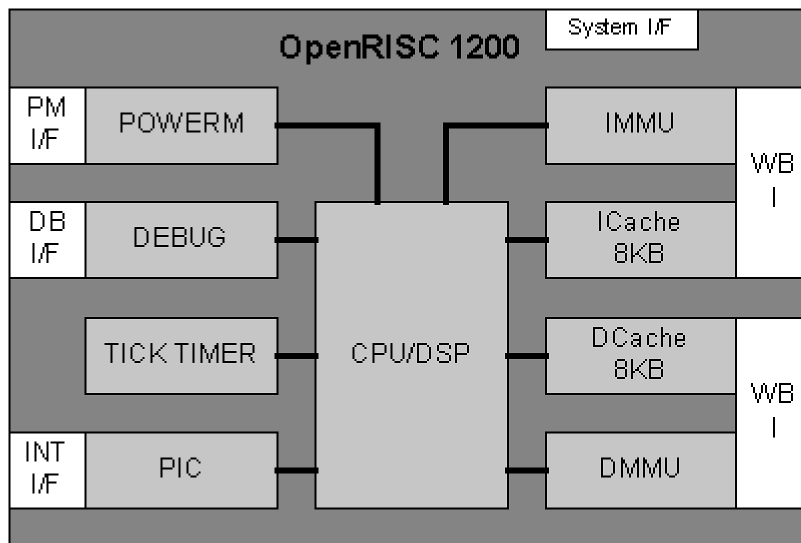

Figure 2: OpenRISC 1200 Architecture

Inputs of the Programmable Interrupt Controller (PIC) interface are connected to interrupt signals from interface controllers of the SoC, specifically UART and Ethernet controllers. This way, software can be programmed to hold on interrupt occurrence, process interrupt and resume afterwards.

The Debug interface (DEBUG) allows control of software execution for OR1200. This interface is connected to the Advanced Debug Interface [YN2008], which enables debug of software using many JTAG cables. Because the Advanced Debug Interface is also connected to the Wishbone bus, it can upload Software to system memory.

The Power Management interface (POWERM) can be used to sleep the OR1200 and reduce the power consumption, but it is left unconnected by MinSoC.

2.2 Wishbone Interconnect

OR1200, the CPU of MinSoC, communicates with the world via Wishbone interfaces following its protocol [HR2002]. The majority of the IP cores in OpenCores also provides a Wishbone interface, thus being compatible with this CPU. First, the CPU must be connected to memory so it can execute software. Furthermore, different extra modules might be necessary to fulfill a task. General Purpose IO is probably the best example of a module to be applied to a system. GPIOs are used to inform states by enabling a LED, to control external modules by software bit-banging, etc. For a computer style standard text IO, UART modules are generally applied. Using a terminal program on a computer, text can be sent from the CPU to the computer and vice-versa.

The connection of diverse modules to only two Wishbone interfaces or of two interfaces to a single module requires an interconnect. An interconnect fulfill two purposes, routing and arbitrating. A master requires connection to a slave by signalizing the slave address on the interconnect. Only slave interfaces have addresses on the interconnect. Master interfaces do not respond to transactions, they initiate them. The interconnect grants the master a connection to the slave module if the slave connection is idle and if there is a free bus to forward the connection.

There are two types of wishbone interconnects, shared bus and crossbar switch. Two OpenCores projects implement each of these; shared bus [J2003] and crossbar switch [UR2001]. A shared bus interconnect only allows one master to communicate with one slave at the same time, while a crossbar switch may allow N masters connect to N slaves at the same time, according to the number of implemented buses.

On MinSoC, the implemented interconnect “minsoc_tc_top.v” is an updated version of the interconnect used on orpsoc project version 1. It is a double shared bus interconnect. It can comply 8 masters/initiators and 9 slaves/targets. The slave addresses can be given as parameters upon instantiation. Slave module 0 has its own bus and thus can be accessed in parallel to any other target module. However, targets 1 to 8 share a bus and cannot be accessed in parallel.

The modules connected to this interconnect can be seen on Figure 1. The symbol for the interconnect is the thick arrow labeled Wishbone. Target 0 is the RAM module. On upcoming versions it will be replaced with optionally wb_conbus [J2003] or wb_conmax [UR2001].

2.2.1 Configuring the Interconnect

The addresses for the targets on the interconnect can be defined as parameters on minsoc_tc_top's instantiation. To define an address, its width and value must be given. They both refer to the highest order bits of the 32-bit address. The default width for all modules on “minsoc_defines.v” is 8-bits and the address for UART is 0x90 for example.

This way, every access to addresses starting from 0x90000000-0x90FFFFFF are going to be forwarded to the UART module. Targets 0 and 1 may have any addresses on the interconnect. Targets 2 to 8's MSB are predefined by the 6th parameter (t28_addr). 0x9 is the default on MinSoC, thus targets 2 to 8 might have any addresses from 0x90- to 0x9F- but no other. The reason for this is a check for valid target address on the second bus arbiter. The arbiter only grants connection to the initiator, if the initiator has the address to a valid target. Instead of checking all target addresses, it checks only two; the address of the first target and the MSB address of targets 2-8.

Parameter	Parameter Position	Target Number	Used Definition	Default Value	Purpose	Connected Module
t0_addr_w	1		APP_ADDR_DEC_W	8	Target 0's address width	
t0_addr	2	0	APP_ADDR_SRAM	0x00	Target 0's address	RAM
t1_addr_w	3		APP_ADDR_DEC_W	8	Target 1's address width	
t1_addr	4	1	APP_ADDR_FLASH	0x04	Target 1's address	or1k_startup
t28c_addr_w	5		APP_ADDR_DECP_W	4	Address width of MSB for targets 2-8	
t28_addr	6	2-8 MSB	APP_ADDR_PERIP	0x9	MSB address of targets 2-8	
t28i_addr_w	7		APP_ADDR_DEC_W	8	Complete address width of MSB Target 2-8	
t2_addr	8	2	APP_ADDR_SPI	0x97	Target 2's address	SPI EEPROM
t3_addr	9	3	APP_ADDR_ETH	0x92	Target 3's address	ETH Slave
t4_addr	10	4	APP_ADDR_AUDIO	0x9D	Target 4's address	NC
t5_addr	11	5	APP_ADDR_UART	0x90	Target 5's address	UART
t6_addr	12	6	APP_ADDR_PS2	0x94	Target 6's address	NC

t7_addr	13	7	APP_ADDR_RES1	0x9E	Target 7's address	NC
t8_addr	14	8	APP_ADDR_RES2	0x9F	Target 8's address	NC

2.2.2 Attaching Modules to the System

To attach a new module to the system, the file “minsoc_top.v” has to be edited and interconnect slots must be available. Most modules are controlled by a CPU and have thus a slave interface. These modules are connected to a target slot on the interconnect. A master interface, on the other hand, can be connected to the an initiator slot. Furthermore, the module's Wishbone interface must be 32-bit wide so that the module can be directly connected to the interconnect.

The UART module [MI2001] is connected to MinSoC. It exemplifies the connection of external modules to the interconnect. In order to attach it to the system, its Wishbone signals have to be connected to a target slot from the interconnect, clock and reset to the correspondent system signals and the RX and TX signals forwarded as SoC outputs. Since the UART module does not comprehend a Wishbone error signal, the error input from the target slot must be assigned to 0. Please refer to “minsoc_top.v” lines 942-950 for the interconnect target slot, lines 666-697 for the UART module instantiation, line 308 for the zero assignment of the error signal and lines 248-265 for the wire declarations. Notice that the external wires have the same name as the MinSoC outputs for UART, line 20, implying their connection.

2.3 Start-Up Modules

The modules, start-up Starter, or1k_startup [UM2009] and SPI [S2002] comprehend the Start-Up circuitry. Its goal is the automatic upload of software from an external SPI memory to the SoC main memory upon circuit power up. The Start-Up circuit is triggered by the SoC reset signal. Although FPGA on-chip memory can be synthesized to contain a software, this solution cannot be applied to external memory.

Depending on FPGA and board, different strategies are used to upload a configuration file deployed on external memory on power-up. After the upload a reset signal is triggered. This signal triggers then the Start-Up circuitry.

The start-up Starter module is coded directly on the “minsoc_top.v” file, lines 338-378. When triggered by the reset signal, it switches the data input signal from the OR1200 instruction interface (wb_dat_i) from the interconnect to itself. A tiny program, lines 365-374, sets the CPU program counter to or1k_startup module address. When the

CPU tries to fetch instructions from or1k_startup, the start-up Starter module switches the data input signal from the OR1200 instruction interface back to the interconnect.

The or1k_startup module contains a small program which controls the SPI interface to read data from the externally connected SPI EEPROM and write it to the main memory. The or1k_startup module expects the first 4 bytes of the SPI memory to inform the size of the program. After the whole program has been written to the main memory, the or1k_startup module sets the CPU program counter to address 0x100, which is the hardwired software reset for the OR1200. The MinSoC included software creates hex files which contains the program size on its first 4 bytes. These hex files can be written to EEPROMs and are compatible with the start-up circuit.

3 Firmware

MinSoC contains two firmwares, a “Hello World” application using the UART interface and an Ethernet interface example, which sends packets received over Ethernet through UART. Both firmwares handle interrupts. The “Hello World” firmware responds to character receive over UART by sending back the the next alphabetical character over the same interface. The Ethernet firmware processes the Ethernet receive interrupt and respond to it by sending the received packet through UART.

The two projects are Makefile projects, they are based on the MinSoC support library, which is a slightly modified version of the support library found on the orpsocv2 project [BJ2009]. The support library includes the register definitions for the CPU itself, register handling macros, CPU initialization code or reset function and interrupt processing methods. Furthermore the Makefiles use a linker script from this library which defines memory spaces (reset, interrupt handlers, program, stack, etc) and selects the code to be attached to them.

Compiling the firmwares by calling “make all” creates for each firmware two executable files (or32 extensions) and two hex files (hex extensions). Two files contain the initialization of data and instruction cache and the CPU initialization (project-icdc names), while the two other comprehend the CPU initialization only (project-nocache names). If the OR1200 implementation has been synthesized without data and instruction caches, the latter should be used. The hex files are used to program memories and for the Testbench (RTL simulation), while the executable file can be used for the OpenRISC simulator or by the OpenRISC gdb to upload the firmware and control its execution.

3.1 Bootstrapping: a tiny bootloader

The necessity of a bootstrapping your program is generally unclear, because commonly the operating system takes care of it. Therefore standard “Hello World” programs are compiled without further care and uploaded to system memory, although it

will not work. There are essentially three mistakes here; the printf function of "stdio.h" is not defined for the OpenRISC uClibc, the resulting executable does not have information about its memory mapping and registers and stack are not initialized. So basically the CPU will software reset to address 0x100, find some chunks of code and run it maybe overwriting the very code while filling the stack. On operating systems, all this is solved upon process creation (e.g. process fork).

The OpenRISC software reset is located at memory address 0x100, so there the CPU initialization will take place (i.e. registers and stack initialization). Furthermore, a memory space between 0x200 and 0x1000 is reserved for interrupt handling, since the CPU jumps to those addresses when interrupt occurs. Finally, the program main memory can be declared starting from the end of the interrupt handler address space until the end of the available memory. The stack is placed on the main memory as well. This memory space is created using a linker script, dividing exceptions from code and so on. On MinSoC the memory space is defined on orp.ld under the "sw/support".

Since C does not have direct access to registers, an assembly code, "reset.S", is required to initialize the registers (supervision register and general purpose registers) and stack. The assembly code also contains and attaches the code to the defined memory spaces. After the initializations, lines 109-111, the assembly instructs the CPU to jump to the C main function. The C main function is defined then under the target project. The stack size is defined on the "board.h" file of the support library under same directory.

3.2 Communication to SoC Modules

Hardware modules are either controlled by registers or pins. Most IP cores have a communication interface which allows a CPU access to its registers and thus its functionality. Different registers can be addressed by the CPU through the communication interface. Since all modules communicates using the same protocol and over an interconnect, a register access from the CPU can be made directly through physical address access from the software.

The address is composed of the module's address and the register address. For instance, to access the Modem Status register of the UART module through software, a variable must be assigned to the content of the address 0x90000006. The register Modem Status is located on address 6 of the UART module, while the UART module is addressed by the leading address 0x90 on the interconnect.

The header file "support.h" of the support library include 3 macros to access register of different width; REG8(address), REG16(address), REG32(address) (i.e. "char tmp = REG8(0x90000006);"). According to these macros, OR1200 instructions which affects only the indicated width are asserted. For example, to send a character over UART, a write to the Transmitter Holding Register address 0x00 is enough (e.g. "REG8(0x9000000000) = 'H';").

For clarity, the addresses of the modules on interconnect are defined on “board.h” file. Also the internal registers of every module are defined by names for later use, their names imply their functionality. The UART registers are found on “sw/support/uart.h” and the Ethernet registers on “sw/eth/eth.h”. In MinSoC, the register accesses are implemented then by “REG_W(IF_BASE+REG_ADDR)” (e.g. “REG8(UART_BASE+UART_TX) = c;”). Although an addition operation is explicitly declared for register accesses, it does not imply performance penalty. The compiler recognizes that the two elements are defined constants and automatically replaces operation and operands with the operation's result.

The way the module's registers control the module operation is defined by the implementation. In order to write a driver for an IP core its documentation and possibly its source code have to be examined. Their analysis points out how the module functionality can be controlled through its registers.

3.3 Interrupt Processing

Interrupts are used by firmwares to interrupt normal operation and respond to other requests. The interrupt mechanism is hardwired on CPUs. Upon interrupt occurrence, the program counter is set immediately to a certain memory address. It has the advantage not to require the software to poll for occurrence of the event and respond to the incoming request immediately.

An interrupt service routine can be divided in two steps, the interrupt handler and the event processing. The interrupt handler takes care of the mechanics of the interrupt and assures safe software operation, while the event processing takes care of the meaning of the event, such as reacting to an external signal.

Because the software jumps into a specific memory address upon interrupt, a part of the code for the interrupt must be placed on that address; the interrupt handler. The interrupt handler saves the CPU state by filling up the stack with the values of the registers; sets the return address to another small function, which reloads the register values from stack and resumes the interrupt; and finally jumps the execution to a C event processing function. Since C does not have direct access to CPU registers, the interrupt handler must be programmed in assembly.

A C function can be used then to execute instructions requested by the interrupt event (e.g. read data on packet reception interrupt). This C function is free to care only about the real event, since the lower level CPU requisites have been covered by the interrupt handler. As soon as the C function ends, the CPU executes the register and stack state reload and resume the interrupt, jumping back to the program exactly where it has been left.

External interrupts are connected to the OR1200 through the PIC (Programmable Interface Controller) interface, all external interrupts are then bundled together to one internal interrupt; the exception handled on address 0x800. In order to find out which external interrupt occurred, the status register of PIC is retrieved. The asserted bits of the status register indicate which interrupt events occurred. The support library includes a mechanism to hide this task from the end user, so that external interrupt processing functions can be programmed regardless also of this requisite.

3.3.1 Interrupt Processing on MinSoC

On MinSoC the interrupt handlers are located under “sw/support/except.S”, two labels for the register storage and end of exception functions are defined. They are located on lines 211-275. To understand the CPU instructions used on “except.S” and the task of the CPU general purpose registers, refer to “OpenRISC 1000 Architecture Manual”; CPU instructions on pages 34-251, CPU general purpose register usage on pages 333-335.

Regular interrupt processing functions are defined on the top source of the target firmware (“sw/eth/eth.c”, “sw/uart/uart.c”, lines 11-28). None of them has implementation, but any code inserted in these functions would be executed upon interrupt occurrence, since the interrupt handlers call them. However, the CPU will only process exceptions, if the exception handling of the CPU is active.

The function called by the interrupt handler to process external interrupts is defined in the support library on “support.c” (hpint_except()). This first function calls then int_main(), which finally retrieves the status of the PIC and calls registered interrupt processing functions according to the asserted bits of the status register.

In order to use the external interrupt mechanism, the external interrupt mechanism has to be initialized by the function “int_init()”, this function also enables the exception handling of the CPU. Then, interrupt processing functions have to be registered (“sw/uart/uart.c”, line 98-99). Furthermore, external interrupt processing functions are allowed to have any name.

The bit order of the status register for the PIC is given by the OR1200 “pic_ints_i” input on the RTL project (“minsoc_top.v” line 589). For instance, this signal is connected to the UART module interrupt signal through its 2nd bit (“minsoc_top.v” line 683). Then, in order for the external interrupt mechanism to call a given function on the occurrence of this interrupt, the first variable of the registering function “int_add” has to accord to the bit position of the correspondent interrupt signal on the “pic_ints_i” input of OR1200 (e.g. int_add(2,&uart_interrupt);). Through the definition file “minsoc_defines.v” lines 84-89, the position of the interrupts on the PIC status register can be easily changed.

4 Simulation

Simulation is the standard method to verify a hardware design. For Register Transfer Level design, testbenches are created. Testbenches exercise the design stimulating its inputs and monitoring its outputs. A testbench can be complex or simple, depending on the design complexity and abstraction; and on how extensive the target functionality or feature will be tested. Quality of a test is measured on code coverage, which points out how extensively all design functionality and features are tested. Due to the concurrency of hardware, tests might have to check if one feature can still respond, while the design is busy performing other tasks. In order to narrow the testing down, engineers design tests based on specification, which mostly includes use cases. Their task is to inspect if the design behaves correctly, according to its specification.

MinSoC is a base platform for many purposes. Depending on the applied firmware or included modules, the design will have a different task, which must be verified by simulation. For this simulation, the project includes the main elements necessary for every platform, so the user can concentrate on testing his platform only.

The first major task of MinSoC's testbench is to bring the target firmware to the main memory. This way, the simulated design will execute the same target firmware which will run later on chip. Then, external interfaces must be stimulated by the testbench, as if they were connected to real external devices. To accomplish this, implemented bus functional models comprehend send/receive tasks, which can be called from within the testbench main routine in order to imitate connected devices. Although the use of these test tools do not enable the test of every corner case of the chip implementation, they allow a complete functional test of the design. Moreover, the software being run on the very simulation can be debugged using the same OpenRISC gdb port and the Advanced Debug System connecting software used for software debugging on chip.

The included "bench/verilog/minsoc_bench.v" file contains the MinSoC testbench. The behavior of the test environment or testbench routine is found on lines 141-170. The rest of the file defines the test environment, instantiates the MinSoC and

initializes the memory with the target firmware so the testbench routine can be easily created.

The testbench routine stimulates the system for both example firmwares, UART “Hello World!” and Ethernet. It sends the character 'A' through UART to test the interrupt behavior of the “Hello World!” application and sends an Ethernet packet through Ethernet to test the Ethernet application. UART data sent to the world by the SoC will be displayed by the simulation on the simulating console, as if it were a terminal program.

4.1 Initializing Memory

To initialize the SoC memory, a testbench internal memory, a Verilog double array called “program_mem” is pre-loaded with the content of a hex file generated by the firmware Makefile project. This is accomplished by the Verilog command “$readmemh()”, “minsoc_bench.v” line 96. Then, two different strategies to initialize the SoC main memory are implemented, the applied one depends on the definitions of “bench/verilog/minsoc_bench_defines.v”. Either the memory will be initialized previous to simulation start (INITIALIZE_MEMORY_MODEL) or after simulation start (START_UP). The latter option models the real circuit behavior using the SoC start-up circuitry, but takes considerable longer to start executing the firmware. The former option copies the content of the hex file directly to a memory model and starts the firmware execution already on simulation start.

Technically, with the START_UP option, the testbench simulates a SPI EEPROM serially sending the complete firmware content. Though, this is actually initiated by the SoC and not by the testbench. The SoC (SPI master) asserts read requests on the interface to acquire the data from the slave, which is then simulated by the testbench.

The INITIALIZE_MEMORY_MODEL option is more convenient, because it is a lot faster. Besides, the START_UP design is already verified. Therefore, there is no necessity of further exercising this part of the code. However, the testbench has to use a memory model, four 8-bit memory arrays, for the INITIALIZE_MEMORY_MODEL option, instead of the synthesizable memory. This memory model has the disadvantage of not being equal to the memory for synthesis.

The synthesizable memory is implemented using vendor specific memory blocks with fixed width and depth. To implement a resizable memory using these blocks, several blocks are instantiated depending on the user input, using the Verilog generate statement. In the testbench, the memory is filled in a loop, where the different memory blocks, instances, are addressed by a loop variable. However, instances generated by the generate statement are not allowed to be addressed by loop variables.

4.2 Bus Functional Models

BFMs model the bus behavior, implementing specified bus tasks, as arbitration or drive and sample of low-level signal behavior according to the bus protocol. They provide tasks (Hardware Description Language functions), which permit the control of transactions on the respective bus. With these tasks, data can be send/received to/from the design by the testbench.

On MinSoC, Ethernet functional send and receive tasks are implemented to exchange information between SoC and testbench ("minsoc_bench.v" lines 338-517). In this model the incidence of collisions is ignored. Therefore, the model is not appropriated for simulation of multiple concurrent Ethernet nodes.

The UART BFM consists of only the sending task ("minsoc_bench.v" lines 277-293), since the reception of data from UART is forwarded to the simulating terminal output by the "uart_decoder" task ("minsoc_bench.v" lines 297-327).

5

Conclusion & Future Steps

An implementation of a system embedding the OpenRISC requires a variety of external elements covered here. Most importantly, the memory, the debug interface (firmware upload) and the Wishbone interconnect are included and functional. Using an already validated system reduces the final bug count, implementation effort and allows the focus to be set on the final design task. The possibility of extension of the project and update of existing elements is essential to cover the multitude of tasks required by the embedded world.

To port most applications to MinSoC, the design of a custom firmware and inclusion of new modules is sufficient. Though, module updates, port of new technologies and possibly custom modules are sometimes necessary. To accomplish these tasks, not only the comprehension of the very tasks and of the MinSoC design and its interfaces is of big importance, but also the analysis of its source code.

Bibliography

OR1200: Damjan Lampret, OpenRISC 1200 IP Core Specification, 2001, http://www.opencores.org/openrisc,or1200

HR2002: Richard Herveille, SoC Interconnection: Wishbone, 2002, http://www.opencores.org/opencores,wishbone

YN2008: Nathan Yawn, Advanced Debug System, 2008, http://www.opencores.org/project,adv_debug_sys

J2003: John, WISHBONE Conbus IP Core, 2003, http://www.opencores.org/project,wb_conbus

UR2001: Rudolf Usselmann, WISHBONE Conmax IP Core, 2001, http://www.opencores.org/project,wb_conmax

MI2001: Igor Mohor, UART 16550 core, 2001, http://www.opencores.org/project,uart16550

UM2009: Michael Unnebäck, OR1k Start-up, 2009, http://www.opencores.org/openrisc,startup

S2002: Simons, SPI controller core, 2002, http://www.opencores.org/project,spi

BJ2009: Julius Baxter, ORPSoC - OpenRISC Reference Platform SoC and Test Suite, 2009, http://www.opencores.org/openrisc,orpsocv2