Title of the MsC Thesis

Name of author
author.name@ist.utl.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2011

Abstract

Place abstract here. No paragraph breaks.

Keywords: Keyword1, Keyword2, Keyword3, Keyword4, Keyword5

1. Introduction

Motivation and state-of-the-art...

Include relevant references [1].

2. Background

Place text here...

2.1. Sub-section...

A generic CFD design problem can be formally described as

$$\begin{aligned} \text{Minimize} \quad & Y(\alpha, \mathbf{q}(\alpha)) \\ \text{w.r.t.} \quad & \alpha\,, \\ \text{subject to} \quad & \mathcal{R}(\alpha, \mathbf{q}(\alpha)) = 0 \\ & C(\alpha, \mathbf{q}(\alpha)) = 0\,, \end{aligned} \tag{1}$$

where Y is the cost function, α is the vector of design variables and $\mathbf{q}$ is the flow solution, which is typically of function of the design variables, and $C = 0$ represents additional constraints that may or may not involve the flow solution. The flow governing equations expressed in the form $\mathcal{R} = 0$ also appear as a constraint, as the solution $\mathbf{q}$ must always obey the flow physics.

2.2. Sub-section...

More text...

3. Implementation

Place text here...

3.1. Sub-section...

More text...

3.2. Sub-section...

More text...

Project Results In the following chapter, the author will analyse the results obtained from the hardware and software developed in this thesis project. The author's first objective was running the "Hello World!" firmware with the *VexRiscv* CPU. Secondly, he tested the implementation of the interrupt routine software with the developed CLINT hardware. Finally, the candidate successfully executed the minimal Linux OS in real hardware using the developed System on a chip.

All the results obtained in this thesis which communicate with the FPGA board or the SOC testbench, are executing the developed *Console* program. The hardware components comprising the SOC differ in each section of this chapter. The author customises the SOC hardware depending on the software needs. Along this chapter, the developed SOC will be referenced as the *IOb-SoC-Linux*.

In each step, the author studied the simulation with the different logic simulators and the memory resources needed to run the respective firmware. Furthermore, when running the SOC on the FPGA board he examined the required FPGA resources.

4. System Running "Hello World!"

The *IObundle* developers created the "Hello World!" firmware to test the functionality of the *IOb-SoC* template. After the author implemented the *VexRiscv* CPU on the developed SOC, he executed a regression test to verify the correctness of the SOC. The regression test was the execution of the "Hello World!", which was known to work correctly on the *IOb-SoC*.

The "Hello World!" firmware is a program that prints a "Hello World!" message to the user, prints the value of π, which is a floating number, and tests file transferring between the *Console* and *IOb-SoC*. The alterations the author made to the *IOb-SoC* hardware to obtain the results presented in this section was swapping the CPU and removing the *IOb-SoC* external memory L1 cache.

The firmware size dictates the minimal size of the memory on the SOC. The *IOb-SoC-Linux* memory is 32 KB because the "Hello World!" program size is 23964 Bytes. The memory size should always be the closest upper bound power of two.

4.1. Execute in simulation

The author simulated the "Hello World!" program using the *Icarus* simulator and *Verilator*. The "Hello World!" simulation allows the author to make a fair comparison between both logic simulators.

In figure 3, the reader can see the expected output when executing the "Hello World!" firmware. In this example, the simulator executed the firmware using the internal memory of the SOC and considered that the firmware was already on the memory. Considering the firmware was already on the memory, allow the simulator not to execute the firmware transfer between the *Console* and the testbench.

Both open-source logic simulators are capable of executing the "Hello World!" program. The simulation is slower when the baud rate decreases or the memory size increases. The baud rate used in the simulation was 5000000. The baud rate is the number of bits per second transferred to the UART. The author executed the simulations considering the system clock frequency 100MHz. The SOC always stores the bootloader in internal memory. The bootloader memory is 4KB ($2^1$2). Since the bootloader binary is 1508 Bytes, 4KB is enough memory to store the bootloader program. The firmware, by default, is stored in the internal memory, and the memory size is 32KB ($2^1$5). In table 1 the reader can see a timing comparison between the different logic simulators simulating the original *IOb-SoC* and the developed SOC. The "INIT_MEM" flag indicates whether the firmware is already loaded in the FPGA or if the *Console* needs to transfer the firmware to the SOC, the user can set the flag to '1' or '0' respectively. The users can execute the simulations with or without external memory. Furthermore, the firmware can run in internal or external memory. The "make sim-test" command tests the different possible simulations.

	IOb-SoC-Linux		*IOb-SoC*	
Command \Simulator	Icarus	Verilator	Icarus	Verilator
make sim-run INIT_MEM=1	2m 26s	0m 3s	0m 25s	0m 3s
make sim-run INIT_MEM=0	88m 19s	1m 1s	15m 18s	0m 27s
make sim-test	231m 3s	2m 27s	43m 34s	1m 34s

Table 1: Timing the "Hello World!" firmware simulation.

From table 1 engineers are able to conclude the advantage of using *Verilator*. For more complexed systems the *C++* testbench is much faster than the *Verilog* counterpart. The disadvantage of using *Verilator* is that signal values can only be either '0' or '1'. However, the speed-up in the simulation is also due to the signal value limitation. In *Icarus*, the simulation can evaluate the signal as unknown ('x') when they are uninitialised. The author noted that *Verilator* is slower to compile the testbench. However, it is much faster to execute the software. The *IOb-SoC* simulation is faster then the authors SOC simulation because the *PicoRV32* is less complex then the *VexRiscv* CPU.

4.2. Execute in the FPGA Board

In figure 6 the readers can see the output of executing the "Hello World!" firmware in the FPGA. The author synthesised the SOC with the external memory. Furthermore, the firmware is running from external memory.

The tables in 4 are the FPGA implementation results for two FPGA families. The author implemented the developed SOC on the kintex Ultrascale AES-KU040-DB-G board and in the CYCLONE V GT-DK. The kintex Ultrascale has an FPGA more capable than the CYCLONE V.

(a)
h

0.45

	IOb-SoC-Linux	*IOb-SoC*
ALM	10,062	9,280
FF	12150	10020
DSP	8	3
BRAM blocks	234	352
BRAM bits	753,248	779,744

Table 2: Cyclone V GT

(a)
h

0.45

	IOb-SoC-Linux	*IOb-SoC*
LUTs	21226	23003
Registers	23373	22588
DSPs	10	7
BRAM	39.5	34.5

Table 3: Kintex Ultrascale

Table 4: FPGA results for "Hello World!" program using external memory.

The author obtained the values in table 4 while using the SOC with the external memory. When synthesising the SOC, the user can define whether he wants to use external memory with the "RUN_EXTMEM" flag. If "RUN_EXTMEM=1" then a memory controller will be synthesised alongside the developed SOC and loaded onto the FPGA. The memory controller is hardware logic written in Verilog and specific to the FPGA where the SOC is running. In order to better understand resource utilisation, the author decided to compare the resources used when running the "Hello World!" firmware from the internal memory. The resources without the memory controller are in tables 7.

From the tables in 7 the author can confirm the

(a)
h

0.45

	IOb-SoC-Linux	*IOb-SoC*
ALM	3,687	1,542
FF	4707	1214
DSP	8	3
BRAM blocks	56	38
BRAM bits	408,800	296,960

Table 5: Cyclone V GT

(a)
h

0.45

	IOb-SoC-Linux	*IOb-SoC*
LUTs	5457	2072
Registers	4405	1074
DSPs	7	4
BRAM	14	9

Table 6: Kintex Ultrascale

Table 7: FPGA results for "Hello World!" program.

VexRiscv CPU requires more resources than the *PicoRV32*. Comparing tables 4 and 7 the reader can see that there is a big difference in resources due to the utilization of the external memory. Interestingly, in table 2 the *IOb-SoC-Linux* uses less BRAMs than the *IOb-SoC*. The *IOb-SoC-Linux* uses less BRAMs because the L1 cache integrated with the *VexRiscv* CPU is smaller than the L1 *iob-cache* hardware module used by the *IOb-SoC*. In table 7 the *IOb-SoC-Linux* clearly uses more resources because the *VexRiscv* CPU contains the L1 cache and is more complex than the *PicoRV32*. *VexRiscv* is more complex than *PicoRV32* because it supports more instruction extensions and has more Control Status Registers.

5. Interrupt Routines

To test the correct functionality of the interrupts in the SOC, the author executed the developed CLINT testbench. Moreover, to test the complete SOC, he ran the bare-metal firmware created to handle interrupts. The firmware was executed in simulation and implemented in the FPGA Board.

The size of the firmware that tests the interrupt routine is 24364 Bytes. Consequently, the only difference in the SOC used on this section tests is the addition of the CLINT hardware. The memory size is the same since the "Hello World!" program and this firmware have similar sizes, under 32 KB.

5.1. Execute CLINT simulation

After developing the CLINT unit, the author executed its testbench, testing the timer and software interrupts. Figure 7 shows the complete process when running a simulation. The author executed the simulation with *Icarus Verilog*. However, the process is similar with *Verilator*. First, the Makefile script has to compile the testbench software. The [language=bash]make sim-run calls the "build" target of the Makefile in the simulator directory. The Makefile calls the "build" target in figure 7 with the command [language=bash]make -C ./hardware/simulator/icarus build. The build target executes the "iverilog" application which compiles the hardware sources and the testbench into an executable program. Secondly the Makefile calls the "run" target. The "run" target will call the software application that runs the testbench. Figure 7 shows the executable being called with [language=bash]./a.out. After that, the testbench will run until it finishes or the user kills the process.

The author previously discussed the testbench executed in figure 7 in subsection **??**. The testbench first sets the timer interrupt to trigger when the "MTIMER" register counter hits 200. The testbench disables the timer interrupt after the CLINT notifies its occurrence and then triggers the software interrupt. In the clock cycle, right after triggering the software interrupt, the testbench will receive the notification from the CLINT unit and disable the interrupt. The testbench finishes after enough time have passed to allow the interrupts to happen if there is nothing wrong with the hardware. Since the testbench and hardware are , *Icarus* is better then *Verilator* to simulate the CLINT.

5.2. Execute in simulation

The author executed the firmware discussed in subsection **??** on simulation with both *Icarus* and *Verilator*. The figures in 10 show the execution of the firmware. The *IOb-SoC-Linux* implemented does not use external memory.

The author wrote the firmware to set the timer interrupt differently for the simulation with *Icarus* and the simulation with *Verilator*. Since the simulation with *Icarus* is slower than with *Verilator*, the time the firmware is waiting for the interrupt to trigger does not have to be as big for the user to notice it while simulating. The *Icarus* simulation set the timer interrupt to trigger after 0.001 seconds, while the *Verilator* simulation set to 0.01 seconds. When the interrupt handler executes, the firmware prints the current time and the value in the "MTIMER" register. If 0.001 seconds had passed the value of the "MTIMER" register should be $100KHz * 0.001s = 100$. The "MTIMER" register value is not the exact time the interrupt takes to trigger because it also counts the time it takes to execute the rest of the firmware.

5.3. Execute in the FPGA Board

Lastly, the author implemented the *IOb-SoC-Linux* in the FPGA boards. Figure 11 shows the correct execution of the firmware from the *IOb-SoC-Linux* internal memory. The firmware programs the timer interrupt to trigger one second after the firmware starts.

In figure 11, the interrupt handler prints the current time that has passed since the firmware started. The hardware consumed approximately one second waiting for the interrupt to trigger. The firmware uses the extra 0.004 seconds to print the "Hello World" message at the start of the firmware and to execute the interrupt handler. The extra time consumed when executing the *IOb-SoC-Linux* in hardware differs from the simulation's extra time. This difference is due to the baud rate used. Since the hardware baud rate is lower than the simulation baud rate, the messages theoretically take more time to print to the terminal.

Tables 8 and 9 represent how much FPGA resources are consumed by the *IOb-SoC-Linux*. The author executed the firmware from the external memory and the internal memory. The tables in 10 show the resources when the SOC contains an external memory (w/ DDR) and when it does not (w/o DDR).

(a)
h
0.45

	w/o DDR	w/ DDR
ALM	3,883	10,257
FF	4940	12300
DSP	8	8
BRAM blocks	56	234
BRAM bits	408,800	753,248

Table 8: Cyclone V GT

(a)
h
0.45

	w/o DDR	w/ DDR
LUTs	5729	21478
Registers	4580	23545
DSPs	7	10
BRAM	14	39.5

Table 9: Kintex Ultrascale

Table 10: FPGA results for interrupt routine program.

Comparing the table in 10 with the tables in 7 and 4 engineers can see that the CLINT hardware does not use much resources.

6. Boot and use the Linux Operating System

The objective of this thesis project was to run an Operating System in the *IOb-SoC-Linux*. The *IOb-SoC-Linux* used in this section adds the PLIC hardware and substitutes the *iob-UART* for the *iob-UART16550*. The software that comprises the complete OS is the *OpenSBI* bootloader, the device tree blob, the Linux kernel and the root file system.

Table 11 presents how much time it takes to build the complete OS with the command [language=sh]make build-OS. The "real" time is the time that passes since the user executes the command until it finishes. The "user" time is the time the CPU takes while executing operations in the user space. The "user" time is bigger than the "real" time because it counts the time passed in each CPU core. Part of the compilation of the RootFS and the kernel is done in parallel using two cores.

real	4m29,570s
user	8m12,039s
sys	0m56,887s

Table 11: Time it takes to build the OS.

The OS size is to big to run in the FPGA internal memory. The *OpenSBI* bootloader is 90896 Bytes. The device tree blob is 1669 Bytes. The Linux kernel is 4426152 Bytes. Lastly the root file system is 1142733 Bytes. The memory has to have at least 8 MB (2^23) to store all this software. However, the Linux kernel needs a bigger memory where it can store virtual memory pages and execute the different application processes. The device tree source describes the system had 512 MB of available memory. Consequently, the author had to implement the *IOb-SoC-Linux* on the FPGA with access to the external memory. The internal memory could never be as big as 512 MB.

In figure 14 the reader can see the start of the OS simulation with *Verilator*.

Figure 12 shows the initialization of the *Console* program. Furthermore, it shows the instantiation of the *iob-UART16550* and the *iob-PLIC*. The *iob-UART16550* and the PLIC core have an initial block that prints their properties. The synthesis tools do not synthesise the initial block to real hardware, but the simulator executes it. Figure 13 shows the *iob-bootloader* and the start of the *OpenSBI* bootloader. The *iob-bootloader* in figure 13 does not transfer the software to the memory because the author executed the simulation considering that the software was already in the memory.

Figure 15 shows the end of the *OpenSBI* bootloader and the start of the Linux kernel. The first line printed by the Linux kernel indicates the author built the kernel executing, the kernel version and which toolchain he used to compile it.

While figure 15 shows the start of the Linux kernel, figure 16 shows the end of the Linux kernel booting process and the execution of the "init" script. The "init" script , as seen in subsection **??**, is the first program the OS executes after the Linux kernel mounts the RootFS and finishes booting. There exist multiple messages printed to the terminal between the output shown in figure 15 and in 16. Those messages show the progress while the Linux kernel boots. The Linux kernel boot process's last message is Run /init as init process. After that message the SOC executes the "init" program.

The warning /bin/sh: can't access tty; job control turned off that appears at the end of the Linux boot in figure 16 means the shell program is not writing to a *tty*, but a socket. Advanced commands such as Ctrl+Z and Ctrl+C are unavailable when writing to a socket. Furthermore, sh will not support background processes (command &) and the associated bg/fg/disown/jobs commands. However, processes forking themselves and closing their inputs will still work. This way, the system is protected from a race condition that could occur if both the shell and the background process were waiting for user input. This problem happens because the author developed the RootFS to be lightweight for embedded systems. Consequently, it does not implement some Linux files and programs that would enable such features. The "init" script could call the shell program could with [language=sh]sh +m.

Figure 17 shows the developed minimal OS running on an FPGA. The reader can see that the author has suppressed the shell warning. The initial part of the figure shows the final stage of the Linux kernel booting. After booting, the author tested the [language=sh]ls / command that showed the files and directories in the systems' root. Lastly the author executed the [language=sh]cat init command for the OS to print the contents of the "init" script to the terminal. The difference between the "init" script printed and the one presented in listing **??** is that "IObundle" is printed as a design with ASCII characters and the shell program is called with the "+m" argument.

The time the Linux kernel takes to boot in real hardware, figure 17, is almost double what it takes to boot in simulation, figure 17. The time to boot is almost double because the memory module used in the simulation does not have any latency. When the L2 cache fetches data from memory in real hardware, it must wait before receiving the data burst. Using the *CYCLONE V* FPGA board the Linux kernel takes 7.01 seconds to boot. The author expected the boot to take longer since the system clock frequency used with the *CYCLONE V* is 50 MHz. The Kintex Ultrascale was able to run with a frequency of 100 MHz. The *OpenSBI* bootloader and the device tree blob had to be recompiled with the system frequency defined to 50 MHz to run in the *CYCLONE V*.

Table 14 show the resources used by the *IOb-SoC-Linux* in the different FPGAs.

The tables in 14 show that the resources utilization from the *IOb-SoC-Linux* is not much bigger than the *IOb-SoC*. The FPGA still has enough resources to implement hardware accelerators.

(a)
h

0.45

	Resources	FPGA usage %
ALM	11,227	10
DSP	8	3
FF	13725	2
BRAM blocks	234	19
BRAM bits	755,424	9

Table 12: Cyclone V GT

(a)
h

0.45

	Resources	FPGA usage %
LUTs	23126	9.54
Registers	24505	5.05
DSPs	10	0.52
BRAM	39.5	6.58

Table 13: Kintex Ultrascale

Table 14: FPGA results for interrupt routine program.

7. Conclusions

Conclusions, future work and some final remarks...

Acknowledgements

The author would like to thank ...

References

[1] J. Nocedal and S. J. Wright. *Numerical optimization.* Springer, 1999.

0.49
(a)
b

0.49
(a)
b

`start_Hello_sim.png`

`start_hello_fpga.png`

Figure 1: Start of the "Hello World!" firmware.

Figure 4: Start of the "Hello World!" firmware.

0.49
(a)
b

0.49
(a)
b

0.49
(a)
b

icarus_int_sim.png

Figure 8: Interrupt routine firmware with *Icarus*.

0.49
(a)
b

clint_sim.png

Figure 7: CLINT timer and software interrupt simulation.

int_fpga.png

Figure 11: Executing the interrupt routine program on the FPGA.

0.49
(a)
b

start_bootloader_sim.png

Figure 12: *iob-UART16550* and *iob-PLIC* properties.

0.49
(a)
b

start_Linux_sim.png

Figure 15: Start of the Linux kernel boot with *Verilator*.

end_Linux_sim.png

Figure 16: End of Linux kernel boot with *Verilator*.

linux_fpga.png

Figure 17: Linux kernel boot in the FPGA.