

TIMA LABORATORY

—

—

Embedded SoC Design & Integration Project

Authors:

Noureddine Ait Said

Chafik Sohaib

Victor Mesnager

Supervisor:

Prof. Mounir Benabdenbi

January 12, 2021

Contents

I	Introduction to the lowRISC project	1
1	The lowRISC Project	2
1.1	Introduction	2
1.2	lowRISC features	3
1.3	lowRISC resources description	3
1.4	lowRISC hardware architecture overview	4
1.5	lowRISC memory map	5
1.6	Boot sequence	6
1.6.1	Boot sequence in Rocket Chip and Spike	6
1.6.2	Boot sequence in lowRISC	7
1.7	The HW/SW co-design flow using lowRISC	8
1.7.1	HW design flow	8
1.7.2	Software design flow	9
2	Xilinx Nexys 4 DDR FPGA	12
2.1	Board overview	12
2.2	Board configuration	13
3	Basic Demonstration	14
3.1	Demo 1: building the first bitstream with the bootloader	14
3.2	Demo 2: An application running from the DDR memory	22
3.3	Results visualization	26
II	Projects Propositions	28
4	Implementation of a software CNN application on lowRISC	29
4.1	General introduction	29
4.2	Project description and provided resources	29
4.2.1	The CIFAR-10 library	29
4.2.2	CNN inference steps	30
4.3	Project deliverable	32

List of Figures

1.1	lowRISC Chip Hardware Architecture.	5
1.2	lowRISC memory map	6
1.3	lowRISC boot sequence	8
1.4	.elf executable binary generation	9
1.5	ELF format description	10
1.6	Bitstream update with a new bootloader.	11
2.1	Nexys 4 DDR overview	12
2.2	Nexys 4 DDR jumpers description	13
3.1	Excerpt of the <code>make verilog CONFIG=DefaultConfig</code> command output.	15
3.2	Expected Project folder hierarchy.	16
3.3	Vivado GUI after Synthesis and Implementation.	17
3.4	Post-Implementation Resource Utilization on the target FPGA. The target configuration here is <code>DefaultConfig</code>	17
3.5	The output of <code>make boot</code>	18
3.6	The output of <code>make bit-update</code>	19
3.7	The output of <code>make cfgmem-updated</code>	19
3.8	The bootloader launch output	22
3.9	Software debugging using GDB and OpenOCD	24
3.10	FPGA-Host-Output context	26
3.11	Results of the application on the original VGA display.	27
4.1	The CIFAR-10 library	30
4.2	Image preparation	31
4.3	The CNN application design	32

Part I

Introduction to the lowRISC project

In this part the lowRISC project is introduced. The hardware/software environment and co-design flow is explained starting from a Rocket Chip generated hardware, additional peripherals and the memory interface provided by lowRISC all the way to the final FPGA bitstream.

Chapter 1

The lowRISC Project

1.1 Introduction

LowRISC is a free microprocessor project, implementing the risc-v free 64-bit RISC processor architecture. This project is founded by **Robert Mullins**, co-founder of Raspberry Pi and designed in partnership with the University of Cambridge, UK.

This project was created by **Andrew Huang** (MIT, also author of the Novena open platform), **Julius Baxter** (OpenRISC project and Cambridge University), **Michael B. Taylor** (University of California San Diego, UCSD Center for Dark Silicon), **Dominic Rizzo** (Google ATAP) and **Krste Asanović** (Berkley).



The goal is to create an open, low-cost processor, the first model of which should operate at a frequency of 500 MHz, as well as a test platform, both free and to create the necessary tools to facilitate its implementation by third-party builders. Also, promote and support the use of open source hardware at the silicon layer.


The lowRISC platform aim to:

- Provide a high quality, secure, and open IP base for derivative hardware designs.
- Lower the barrier of entry to producing custom silicon for specific applications, meeting growing needs for more specialist chips.
- Establish a vibrant ecosystem around open hardware designs and tools, making them reliable and sustainable.
- Enable great R&D and product development elsewhere, including universities.

The lowRISC foundation is a FOUNDING SILVER member in the RISC-V organization, and they are now working in several project such as:

- **Ibex:** A two-stage pipeline RV32IMC core.
- **LLVM Compiler Infrastructure.**
- **64-bit SoC platform:** this is the project on which this report's work is based.

For further information please refer to the official website: <https://lowrisc.org>

 As of December 25th, the lowrisc.org website does no longer refer to tutorial links. Please refer to <https://www.cl.cam.ac.uk/~jrrk2/docs/docs/> instead.



1.2 lowRISC features

For our project, we are going to use version 0.6¹, which includes:

- A RISC-V CPU written in the Chisel/Scala hardware description language, generated from Rocket-Chip.
- A variety of useful peripherals, UART, PS2 keyboard, MMC/SD-Card controller, Ethernet(100BaseT), VGA compatible screen, keyboard etc.
- A NASTI² interface to proprietary DDR memory controller from Xilinx.
- A bootloader that supports a variety of booting procedures:
 - Loading the application from MMC/SD-Cards to the DDR memory.
 - Loading the application through an Ethernet link.
 - Loading the application using a GNU Debugger (GDB) instance running on the host computer.
- This version does not support only bare-metal applications, but also a minimal Debian Linux OS, compiled for RISC-V, and adapted with drivers to attach to the above peripherals.

In this project, only the bootloader and baremetal applications are considered.

1.3 lowRISC resources description

The resources provided by lowRISC cover different aspects i.e. hardware components, software libraries, tools and scripts etc. However, in this project we will not use everything, the following are the most important folders to be aware of:

- `$LOWRISC_PATH/rocket-chip`

A modified version of Rocket Chip³. It Contains the Rocket Chip core and its sub-systems.

- firrtl: hardware description intermediate language.
- hardfloat: hardware floating-point arithmetic unit.
- torture: tricky tests that stress-test the CPU.
- riscv-tools: the cross-compilation and simulation tool chain.⁴

- `$LOWRISC_PATH/fpga`

Contains FPGA-implementation-related files.

¹Branch : frame-buffer, commit: 5d594a8

²Acronym for “Not A Standard Interface”, a subset of the AXI interface. Used instead of AXI itself to avoid trademark issues. More here <https://stackoverflow.com/questions/28606457/axi4lite-slave-ip>

³This version has been developed to support FPGA workflow. It does not support creating C emulators (using Verilator). This version has been forked from Rocket Chip official repository, commit **84f6ac1**.

⁴Rebuilding the toolchain is not necessary, we will be using the same compilation toolchain as the first labs.



- `bare_metal`
 - `driver`: contains some basic C library functions implementations (`memcpy`, `printf` using the UART driver, ...) ⁵, as well as the linker script (`test.ld`) and the entry file (`crt.S` ⁶).
 - `examples`: contains small bare-metal application examples (`boot.c`, `hello.c`, `jump.c` ...) that can be compiled as a bootloader alone or as an application depending on the linker file's start address. This will be explained later in this document.
- `board/nexys4_ddr`: for the NexysTM4 DDR Artix-7 FPGA Board target
 - `constraint`: pin planning and timing constraints XDC files.
 - `script`: TCL scripts used to generate and configure a VIVADO project. It is recommended not to call these scripts directly, and to use the provided Makefiles.
 - `lowrisc-chip-imp`: the VIVADO project folder. This folder and its files are created upon execution of the `make project` command. Once created, it will contain a standard VIVADO project hierarchy (a `.XPR` vivado project file that can be opened using VIVADO, then a set of other folders: `simulation`, `design runs` ...).

- `$LOWRISC_PATH/src/main/verilog`

Contains all the hardware components of the lowRISC SoC, where some are written in Verilog (*.v) and some in System Verilog (*.sv). The top level code of the lowRISC chip is described in the `chip_top.sv` file. You will also find the hardware description of the available peripherals.

Other folders (`debian-riscv64`, `riscv-linux`, `qemu`, `vsim`) are out of the scope of this project.

1.4 lowRISC hardware architecture overview

The lowRISC chip global architecture is depicted in Fig. 1.1. The top level hardware description can be found at (`$LOWRISC_PATH/src/main/verilog/chip_top.sv`). Inside this top level module you will find the instantiation of several modules:

- A Rocket System: resulting from the CHISEL to Verilog compilation in Rocket Chip. It is composed of one or several Rocket tiles, depending on the compiled architecture, and all the other standard components such as PLIC, CLINT etc.
- TileLink to NASTI bridges: As explained in the previous labs, the busses used in Rocket Chip are based on the TileLink (TL) protocol. However, to communicate with the external peripherals an AXI-compatible interface is needed. lowRISC uses a non-standard subset interface of AXI, called NASTI. The TL to AXI bridges, provided by Rocket Chip, convert memory transactions from TileLink to AXI, which is compatible with the NASTI interface.

Additionally, there is an AXI to a much simpler BRAM interface that converts AXI signals to BRAM signals (`enable`, `write enable`, `address`, `write data`, `read data`). This interface feeds the peripherals subsystem (`periph_soc.sv`), as well as the BRAM that contains the second stage bootloader.

⁵Similar to `syscalls.c` in the previous labs.

⁶`crt.S` is equivalent to `entry.S` in the previous labs.

- Peripherals: all the external peripherals are integrated in one subsystem (`$LOWRISC_PATH/src/main/verilog/periph_soc.v`). Inside this subsystem, several drivers are instantiated, among others you will find: UART (`$LOWRISC_PATH/src/main/verilog/uart.v`), VGA (`$LOWRISC_PATH/src/main/verilog/fstore2.v`), keyboard (`$LOWRISC_PATH/src/main/verilog/ps2-keyboard.v`) etc.

Since the AXI interface is not used inside the peripherals subsystem, a custom address decoding is implemented, and it follows a one hot decoding scheme⁷. More details about that in section 1.5.

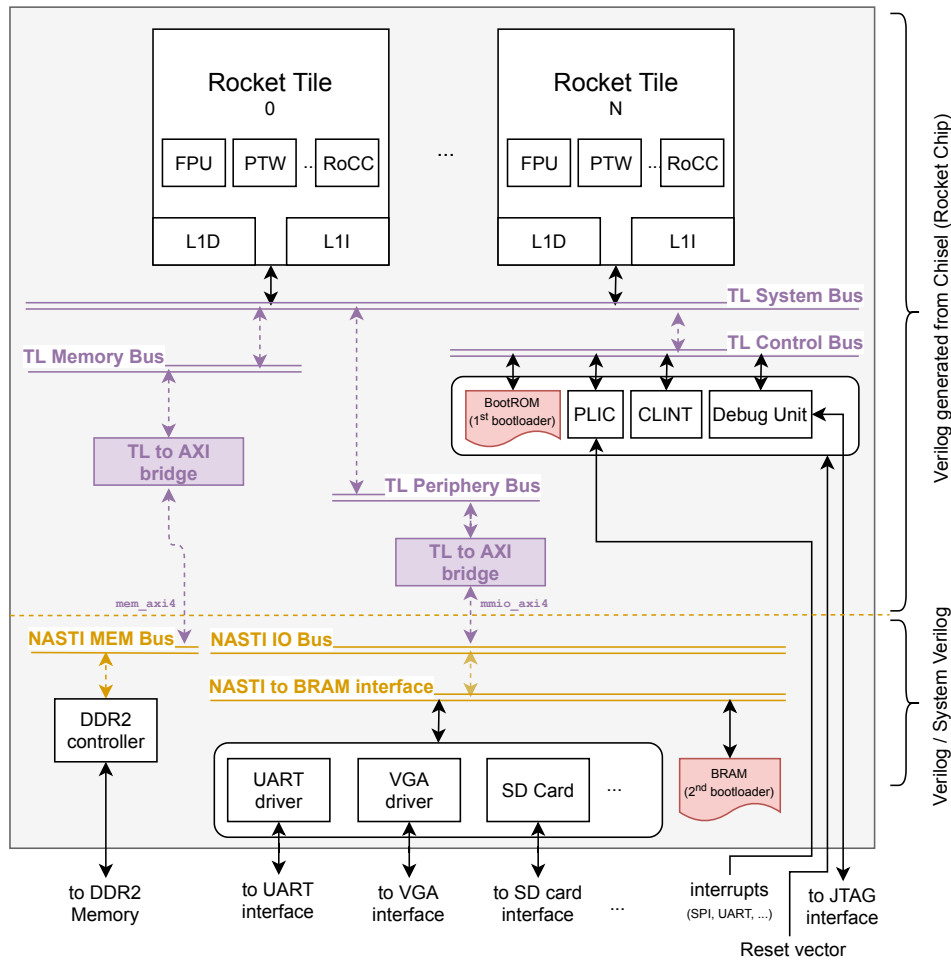


Figure 1.1: lowRISC Chip Hardware Architecture.

1.5 lowRISC memory map

Figure 1.2 depicts the memory map of the system, presented graphically. This memory map can be found at `$LOWRISC_PATH/fpga/bare_metal/driver/lowrisc_memory_map.h`.

```
enum { clint_base_addr = 0x02000000, // CLINT base address
```

⁷One hot decoding <https://en.wikipedia.org/wiki/One-hot>


```

2    plic_base_addr      = 0x0c000000 , // PLIC base address
3    bram_base_addr      = 0x40000000 , // 64KB read-only code region
4    sd_base_addr        = 0x40010000 , // SD card base address
5    sd_bram_addr        = 0x40018000 , // SD card buffer RAM
6    eth_base_addr       = 0x40020000 , // Ethernet base address
7    keyb_base_addr      = 0x40030000 , // Keyboard base address
8    uart_base_addr      = 0x40034000 , // UART base address
9    vga_base_addr       = 0x40038000 , // Older VGA base address
10   new_vga_base_addr    = 0x40080000 , // New VGA driver base address
11   ddr_base_addr        = 0x80000000 // DDR2 Memory base address
12 };

```

Listing 1.1: Memory map

Addresses lower than 0x4000_0000 contain memory mapped registers related to Rocket Chip internal components (debug module, CLINT, PLIC etc.).

Addresses between 0x4000_0000 and 0x4010_0000 are dedicated for MMIO. This memory space is mainly reserved for peripherals as well as the BRAM containing the 2nd stage bootloader. This BRAM is configured to have 64KB.

Addresses greater or equal 0x8000_0000 are reserved for the main memory, and this address space is mapped to the external DDR2 memory, which has a total size of 128MB. It is suitable for running applications or complex OSes such as Linux.

Globally, there are two different address spaces, the MMIO address space (referred to by MMIO_AXI4 in Fig. 1.1). and the External Memory address space (referred to by MEM_AXI4 in Fig. 1.1). mappings are defined in Rocket Chip under \$LOWRISC_PATH/src/main/scala/system/Configs.scala.

1.6 Boot sequence

A boot-loader is the first piece of code executed when booting a system, it tends to be very small and relatively simple.

1.6.1 Boot sequence in Rocket Chip and Spike

In several other RISC-V based implementations, the boot process is composed of two stages. The first stage is usually a three-lines-of-code assembly file, contained in a read-only memory called BootROM. It is usually hard-coded or included in CHISEL (Fig. 1.1) by default at address 0x10000. Usually this bootloader just redirects the PC to another address where the application is mapped (0x8000_0000 for example).

The example depicted in listing 1.2 can be found in \$LOWRISC_PATH/rocket-chip/bootrom/ and it is included at configuration-time in CHISEL in \$LOWRISC_PATH/src/main/scala/Configs.scala. The BootROM also contains the device tree blob (DTB), which describes all the peripherals and devices available and the system. A typical OS such as Linux, would read the DTB to determine the addresses and ranges of the available hardware peripherals.

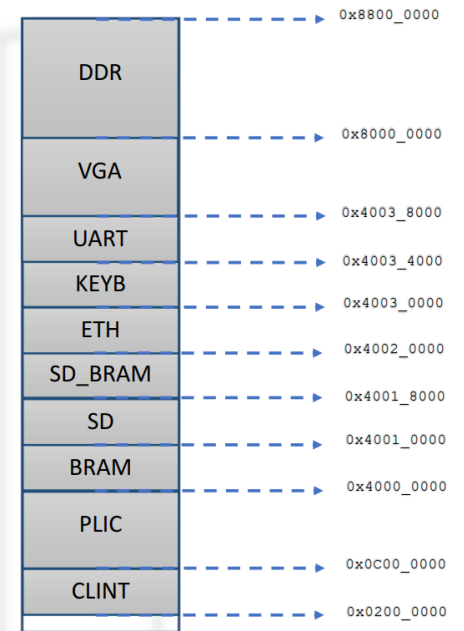


Figure 1.2: lowRISC memory map

This boot process is used in Rocket Chip and Spike. However lowRISC uses a different way of booting to simplify the development process, leaving a variety of booting choices to the developer. The execution does no longer start from the BootROM, but it still contains all the information about the devices available in the system.

```

1 #define DRAMBASE 0x80000000
2
3 .section .text.start, "ax", @progbits
4 .globl _start
5 _start:
6     li s0, DRAMBASE
7     csrr a0, mhartid
8     la a1, _dtb
9     jr s0

```

Listing 1.2: bootrom.S used in Rocket Chip

1.6.2 Boot sequence in lowRISC

In lowRISC, the Rocket Chip system has been modified to expose the reset vector as an input. Using the dip switches available on the FPGA board, a reset vector is assigned a value among 0x4000_0000 (BRAM), 0x8000_0000 (DDR memory), and 0x8020_0000 (DDR memory).

This can be found in the top-level module (\$LOWRISC_PATH/src/main/verilog/chip_top.sv):

```

1 always @* // Combinatorial
2 begin
3     casez (i_dip[1:0])
4         2'b?0: io_reset_vector = 32'h40000000;
5         2'b01: io_reset_vector = 32'h80000000;
6         2'b11: io_reset_vector = 32'h80200000;
7     endcase // casez ()
8 end

```

In this project the bootloader is a C program (defined in \$LOWRISC_PATH/fpga/bare_metal/examples / boot.c), which is compiled and mapped to a 64KB BRAM memory (address 0x4000_0000 in the MMIO address space). The bootloader should be compiled (.elf format), converted to .hex format, and included in the bitstream to initialize the BRAM. More details in 1.7.2

The purpose of BRAM is to allow the FPGA to boot standalone when DDR memory has undefined contents. Clearly it is not possible to rely on the startup contents of DDR as it is volatile. On the other hand the BRAM is initialized from the FPGA configuration bitstream. This emulates the behavior of FLASH memories in embedded systems.

Initially, the dip switch SW0 should be set to 0, in order to point the reset vector to the BRAM memory. When the execution starts, the bootloader initializes the processor and the main peripherals, prints a logo and a welcome message, performs a series of self-tests, and then proceeds to the second step, which is loading the application. The target application can be loaded following two different scenarios depending on the state of the dip switches:

- If (SW[2:0] == "000")

In this case, OpenOCD and GDB should be used to load the application (whose entry address should be set to 0x8000_0000) to DDR memory and the execution resumes from there. GDB automatically sets the PC to the new entry address.

- Else if (SW[2:0] == "010")

In this case, the application (whose entry address is set to 0x8000_0000) will be loaded from the SD card and mapped to the DDR memory. The application should be renamed `boot.bin` because it is hard-coded in the bootloader. You may change it if needed. Once loaded and mapped, a jump is performed by setting the `mepc` CSR register to 0x8000_0000 and calling the `mret` instruction. The application then resumes its execution from the DDR memory.

- Else if (SW[2:0] == "100")

This loads the application from Ethernet. Currently we do not support this mode. Please check the official project for information if needed.

- Else

Otherwise, the boot loader will enter an infinite loop. You may still load the application using GDB in this case, since GDB can interrupt the SoC at any moment.

We highly recommend GDB for development and the SD card for the final demonstration once the development is finished. The boot sequence is depicted in figure 1.3.

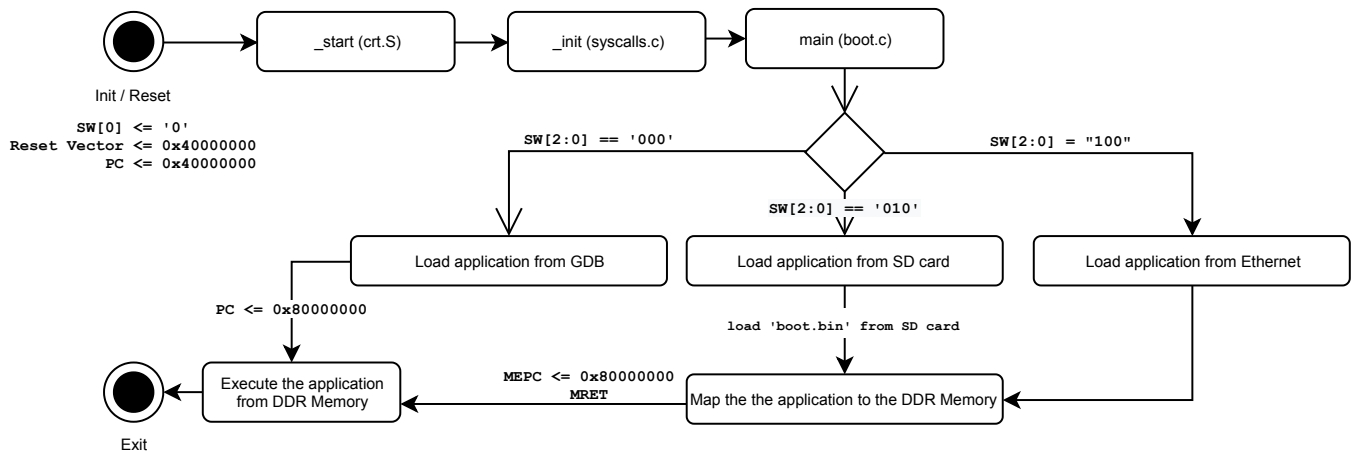


Figure 1.3: lowRISC boot sequence

1.7 The HW/SW co-design flow using lowRISC

1.7.1 HW design flow

As seen previously, lowRISC is composed of several hardware components: auto generated Verilog files from the configurable Rocket Chip as well as the custom peripherals developed by the lowRISC foundation. The HW design flow consists of the following steps:

- Choose the adequate Rocket Chip configuration.

As you already seen in the previous labs, Rocket Chip is highly parameterizable. For instance, you can change the number of cores, the size of the caches, enable/disable the FPU, PTW, virtual memory etc. The only constraint to take in consideration is the resources available in the target FPGA.

In our case, we target a Xilinx Nexys 4 DDR FPGA, and a Default Configuration (RV64IMAFDC ISA) can be supported. Trade-offs should be made if more specific resources are needed. For instance, the L1 data and instruction caches are mainly implemented as BRAM, hence, if more BRAM is needed for a particular application (a frame buffer for a VGA driver for example), then the size of the caches can be reduced by modifying the target Rocket Chip configuration.

- Create a Vivado project, synthesize, and implement.

The project will collect all the necessary source files.

- Generate the bitstream and program the board.

It is fundamental to understand that the generated bitstream is not hardware only. It also contains a BRAM initialized by the bootloader. This is the first piece of code that will be executed after programming the board.

Deployment: section ??

1.7.2 Software design flow

Generating executable binaries in .elf format

The final FPGA bitstream carries both the FPGA hardware infrastructure configuration as well as a BRAM initialization configuration. For that, we first proceed by generating the bootloader in .elf format, using the usual RISC-V compiler. This step is the same whether the target binary is intended to act as a bootloader⁸ or as an application⁹. The only difference is the entry point in the linker file that should be changed accordingly.

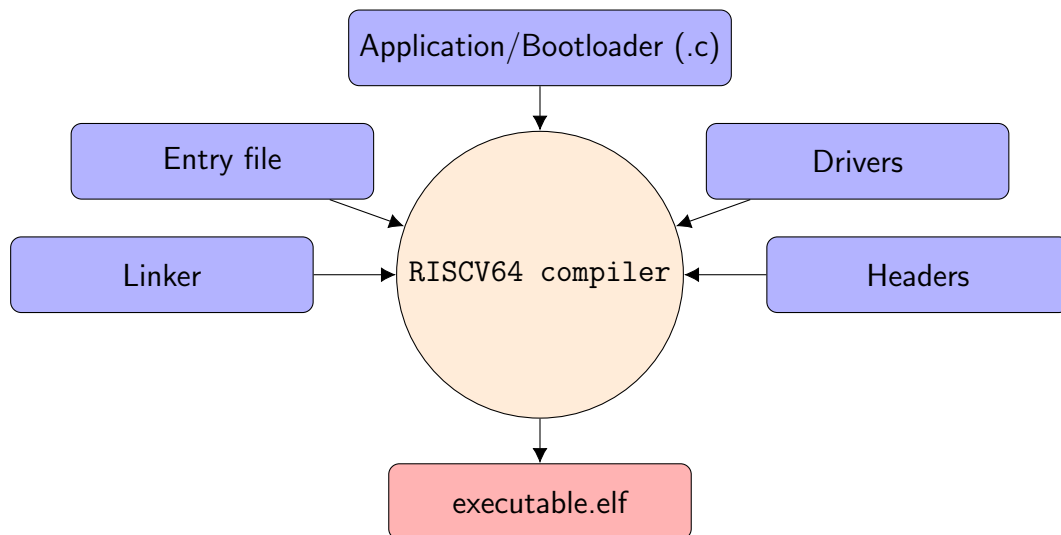


Figure 1.4: .elf executable binary generation

ELF¹⁰ stands for **Executable and Linking Format** is a common, standard format for executable files, object code, and shared libraries. Elf files consists of a symbol look-ups and relocatable

⁸A program that will be mapped to BRAM at address 0x4000_0000

⁹A program that will be mapped to DDR memory at address 0x8000_0000

¹⁰For further information about the elf format please refer to http://www.skyfree.org/linux/references/ELF_Format.pdf

table, that is, it can be loaded at any memory address by the kernel and automatically, all symbols used, are adjusted to the offset from that memory address where it was loaded into. Usually ELF files have a number of sections, such as 'data', 'text', 'rodata', 'bss', to name but a few... organized like described in figure 1.5. It is within those sections where the run-time can calculate where to adjust the symbol's memory references dynamically at run-time.

The RISC-V64 compiler (`riscv64-unknown-elf-gcc`) used to compile the application/bootloader in .elf format using same cross-compiler used in the previous labs. The steps are described in figure 1.4.

- The application/bootloader program ({boot, hello, ...}.c) should be placed in `$LOWRISC_PATH/fpga/bare_metal/examples`.
- The entry file refers to `$LOWRISC_PATH/fpga/bare_metal/driver/crt.S`, it is equivalent to `entry.S` in the previous labs.
- Drivers are located at `$LOWRISC_PATH/fpga/bare_metal/driver`. They contain very basic and minimal support for peripherals and key standard functions (`memcpy`, `printf`, `syscalls.c`, etc.).
- Headers refer mainly to the .h files containing register definitions, memory-mapped registers addresses etc.
- Linker refers to the linker script `$LOWRISC_PATH/fpga/bare_metal/driver/test.ld`.

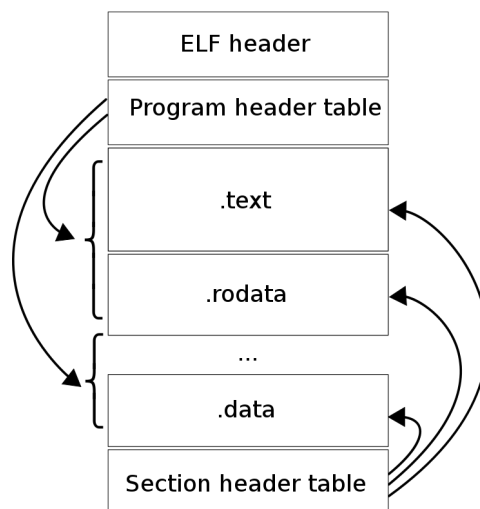


Figure 1.5: ELF format description

! The entry address inside the linker file should be adjusted according to the target. The Makefiles use the same linker for both. Remember to change the address before launching the make command.

```
1 OUTPUT_ARCH( "riscv" )
2 SECTIONS {
3     /* starting address */
4     . = 0x40000000; // Change to 0x80000000 when building an application
5     ...
6 }
```

Generating the bootloader BRAM initialization file

For the bootloader, a step further should be performed, which is the conversion from the .elf format to .hex format in order to initialize the BRAM inside the bitstream.

Hex format is used to store machine language code in hexadecimal form. It is a widely used format to store programs to be transferred to micro-controllers, ROM and EEPROM using burners or JTAG programmers. A hex file describes the raw binary data.

The generated Hex file can be used

- At synthesis-time

As you can see in the top-level module (listing 1.3), using the `readmemh` Verilog directive that initializes memories at synthesis-time with binary/hexadecimal content stored in an external file (Fig. 1.1).

```
1 initial $readmemh("boot.mem", ram);
```

Listing 1.3: BRAM initialization at synthesis-time.

- After the bitstream is generated

It can be used to patch an existing bitstream to change the memory content without re-synthesizing and re-generating the whole bitstream. More details about that in section 1.7.2.

Patching a bitstream with a new bootloader

This step combines the hardware bitstream with the application hex binary and generate a new bitstream that contain the lowRISC hardware description and also the customize application implementable on the FPGA target on bare-metal mode.

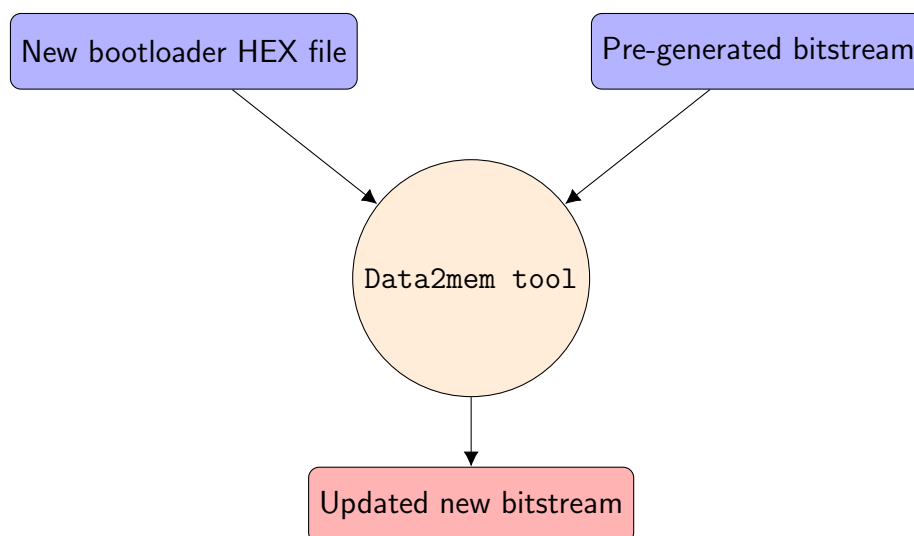


Figure 1.6: Bitstream update with a new bootloader.

Data2mem¹¹ tool: is a Xilinx data translation tool that maps binary content (generated from the bootloader) to block RAM-implemented address spaces on the Xilinx device. In other words, it replaces the BRAM initialization with the new bootloader binary content.

¹¹For more information about this tool https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/data2mem.pdf

Chapter 2

Xilinx Nexys 4 DDR FPGA

2.1 Board overview

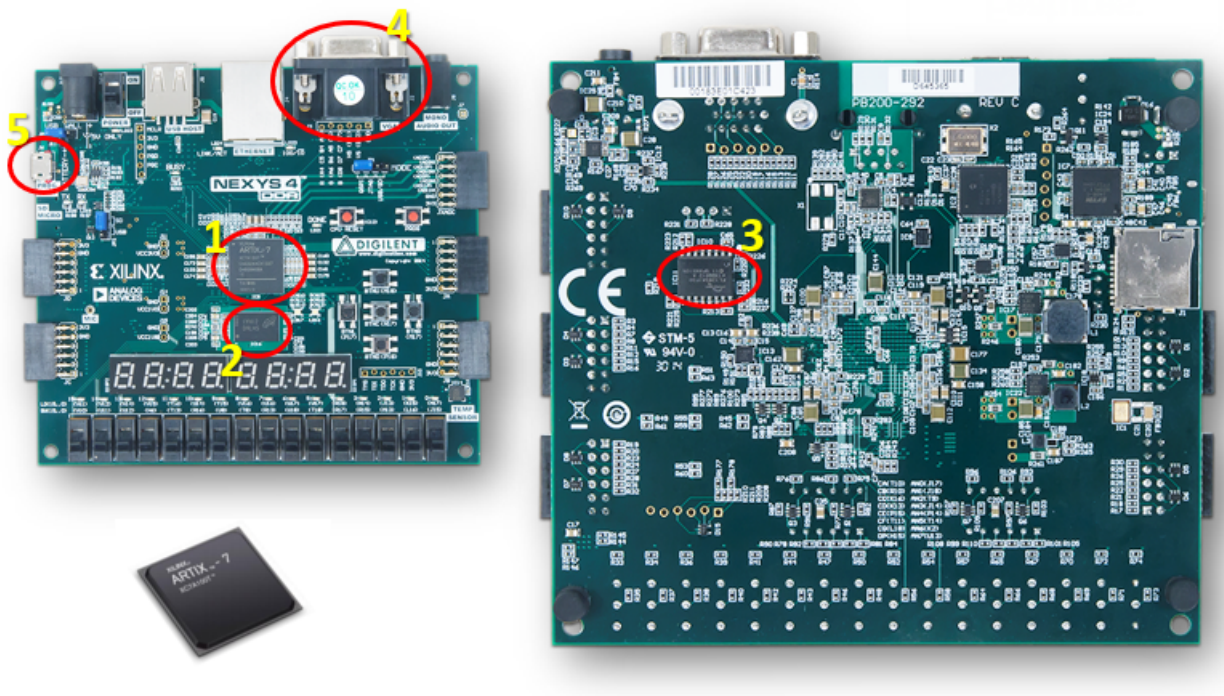


Figure 2.1: Nexys 4 DDR overview

The target device in this project is the Xilinx Nexys4 DDR. It's one of the devices that support the RISC-V ISA, and the test base target of the lowrisc environment.

The Nexys4 DDR board is a complete, ready-to-use digital circuit development platform based on the latest Artix-7 Field Programmable Gate Array (FPGA) from Xilinx (1 in the figure 6.1). With its large, high-capacity FPGA (Xilinx part number XC7A100T-1CSG324C), generous external memories, and collection of USB, Ethernet, and other ports, the Nexys4 DDR can host designs ranging from introductory combinational circuits to powerful embedded processors. Several built-in peripherals, including an accelerometer, temperature sensor, MEMs digital microphone, a speaker amplifier, and several I/O devices allow the Nexys4 DDR to be used for a wide range of designs without needing any other components.

The Artix-7 FPGA is optimized for high performance logic, and offers more capacity, higher performance, and more resources than earlier designs. Artix-7 100T features include:

- Internal clock speeds exceeding 450 MHz.

- 4,860Kbits Block RAM (BRAM).
- 128MB DDR (2 in the figure 6.1).
- 16MB Quad-SPI Flash (3 in the figure 6.1).
- 12-bit VGA output (4 in the figure 6.1).
- Digilent USB-JTAG port for FPGA programming and communication (5 in the figure 6.1).

2.2 Board configuration

In this section, I'm putting some details concerning the jumpers configuration on the board in order to be compatible with the lowrisc requirement.

The lowrisc platform uses the Quad-SPI memory to program the FPGA, so the jumpers configuration should be as follow:

- **JP1**: should be in the SPI Flash position.
- **JP2 and JP3**: Not mandatory when using GDB to load the application. However, when using RISC-V Linux instead of baremetal they should be set to SD/USB position.

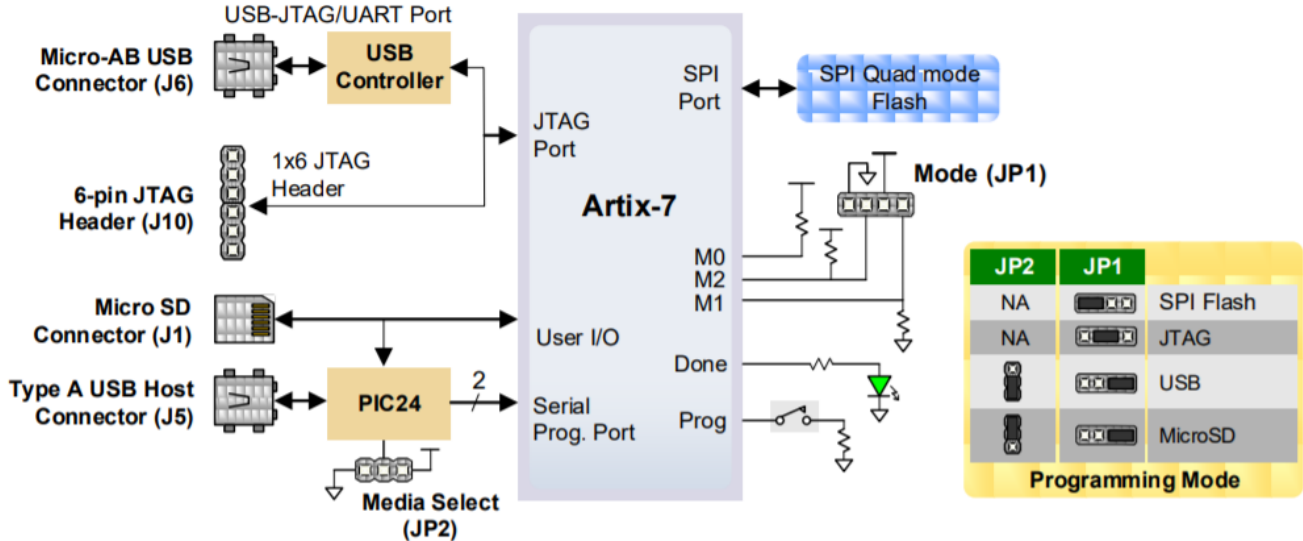


Figure 2.2: Nexys 4 DDR jumpers description

Chapter 3

Basic Demonstration

In this part, you will be provided with the necessary commands in order to put in practice what you learned in the first part.

Terminology: each command in the remaining of this document can be executed either on

- The Host machine only: if the command starts with the tag **[HOST]**; Or
- The Virtual Machine (VM) only: if the command starts with the tag **[VM]**; Or
- Both on the Host and the VM: if nothing is specified.

3.1 Demo 1: building the first bitstream with the boot-loader

1. The main repository of this project is “lowrisc-chip-frame-buffer”.

It contains a special version of the Rocket Chip SoC generator, a set of hardware IPs, and several configuration files. The HW/SW design is also slightly different than the one used in the labs. For example, this flow does not support experimenting with Spike / Verilator. It only supports direct implementation on a Xilinx Nexys 4 DDR (Nexys A7) FPGA.

2. Sourcing tools

```
1 [ VM ]$ cd lowrisc-chip-frame-buffer && source set_env.sh
2 [HOST]$ source <VIVADO 2018.1 configuration file>
```

Make sure that the \$RISCV environment variable still points to the RISCv toolchain's.

Clearly Vivado is going to be used outside the VM.

3. Generating the hardware from Rocket Chip

```
1 $ cd fpga/board/nexys4_ddr # Execute on VM and Host
2 [ VM ]$ make verilog
```

This will generate the default-config-related Verilog files by default in \$LOWRISC_FOLDER/rocket-chip/vsim/generated-src. If the command is executed successfully you will get the following output:

This command calls the makefile available in \$LOWRISC_FOLDER/rocket-chip/vsim with the default parameters (CONFIG=DefaultConfig). If you would like to override the configuration name, use the following

```
1 [ VM ]$ make verilog CONFIG=DefaultSmallConfig
```

```
Generated Address Map
 0 - 1000 ARWX debug-controller@0
3000 - 4000 ARWX error-device@3000
10000 - 20000 R XC rom@10000
2000000 - 2010000 ARW clint@2000000
c000000 - 10000000 ARW interrupt-controller@c000000
40000000 - 40100000 RWX mmio-port-axi4@40000000
80000000 - c0000000 RWXC memory@80000000

[deprecated] Debug.scala:941 (1 calls): $bang$eq is deprecated: "Use \'/=\', which avoids potential p
recedence problems"
[deprecated] DebugTransport.scala:182 (1 calls): $bang$eq is deprecated: "Use \'/=\', which avoids po
tential precedence problems"
[warn] There were 2 deprecated function(s) used. These may stop compiling in a future release, you are
encouraged to fix these issues.
[warn] Line numbers for deprecations reported by Chisel may be inaccurate, enable scalac compiler depr
ecation warnings by either:
[warn]   In the sbt interactive console, enter:
[warn]     set scalacOptions in ThisBuild += Seq("-unchecked", "-deprecation")
[warn]   or, in your build.sbt, add the line:
[warn]     scalacOptions := Seq("-unchecked", "-deprecation")
[info] [14.987] Done elaborating.
[success] Total time: 260 s, completed Dec 13, 2020 6:51:35 PM
```

Figure 3.1: Excerpt of the `make verilog CONFIG=DefaultConfig` command output.

In order to work with lowRISC, custom configurations should extend the base configuration with the new `WithJtagDTMSys` trait, just like the `DefaultConfig` does.

4. In this part, we will create a Vivado project using a TCL script. The created Vivado project will have references to the following:
 - The Verilog file generated from Rocket Chip: contains the CPU core/cores, the caches, MMU, TLBs, TileLink system and peripheral busses ...
 - The hard-coded SystemVerilog/Verilog files: implementing the peripherals drivers, an interface that translates the TileLink bus transactions to AXI-like memory transactions, a DDR RAM controller that connects the AXI-like interface to the actual DDR controller...
 - Board-specific files such as XDC constraints

Create a vivado project using the following command.

```
[HOST]$ make project CONFIG=ConfigNameHere
```

This will create a folder named `lowrisc-chip-imp-ConfigNameHere`. Inside of which you will find a Vivado project file (.xpr) that points to all necessary source files. Here is the expected hierarchy of the project:

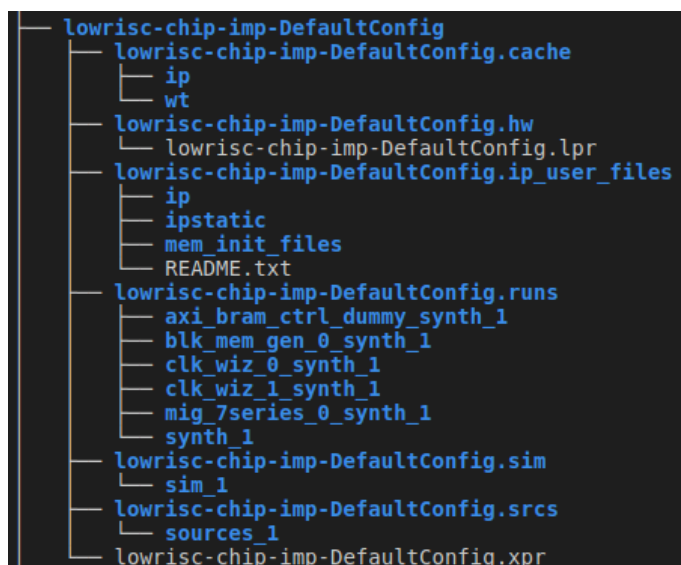


Figure 3.2: Expected Project folder hierarchy.

From this point on, you can either open the project with Vivado and do the next steps (development, synthesis, implementation ...) manually using the GUI. Or instead, you can do that using command line without even opening the Vivado GUI.

5. Generating the first bitstream As explained, you can generate the bitstream using the TCL scripts bundled with the project by issuing the following command:

```
1 [HOST]$ make bitstream CONFIG=ConfigNameHere
```

For information, this step takes a lot of time (e.g., 20min on an Intel(R) Xeon(R) E-2176M CPU @ 2.70GHz), since it compiles, synthesizes, implements the SoC, and generates the FPGA bitstream.

This generates a bitstream file at `lowrisc-chip-imp-ConfigNameHere/lowrisc-chip-imp-ConfigNameHere.runs/impl_1/chip_top.bit`. At the very first time the BRAMs inside the bistream will not be initialized, since the bootloader has not been compiled. These memory will be populated in the next step.

If you prefer the GUI mode, you can launch Vivado (by executing the command `vivado lowrisc-chip-imp-DefaultConfig.xpr` & in a HOST terminal), and then open the `.xpr` file (`fpga/board/nexys4_ddr/lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.xpr`).

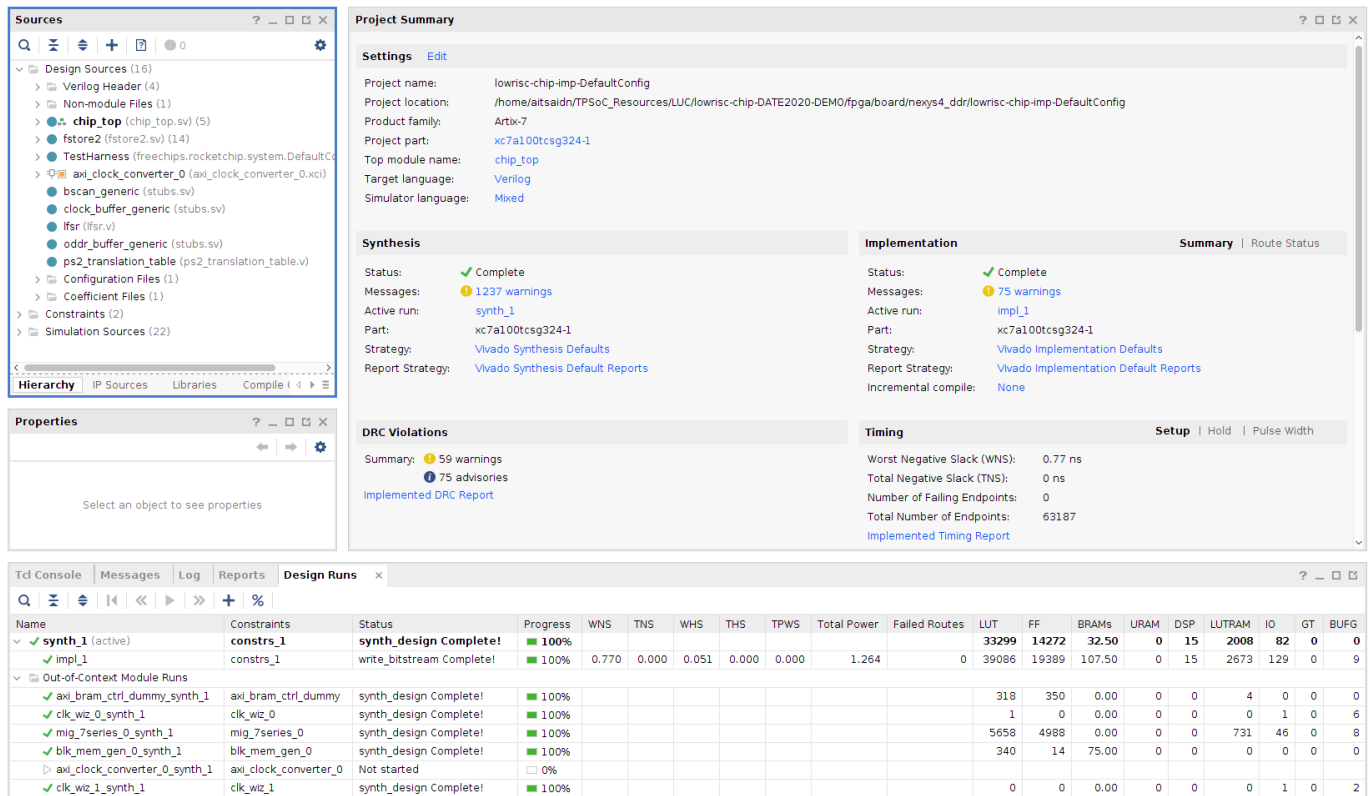


Figure 3.3: Vivado GUI after Synthesis and Implementation.

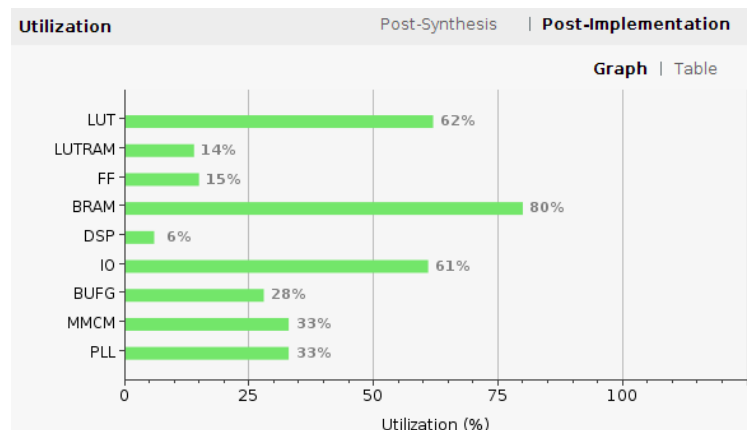


Figure 3.4: Post-Implementation Resource Utilization on the target FPGA. The target configuration here is DefaultConfig

- Now, the generated bitstream only contains the hardware. We need to populate the Bootloader BRAM memory with a bootloader firmware. To do that, we will first locate the bootloader in the software resources, configure it, compile it, then convert it to a .hex file, and then insert it inside the chip_top.bit bitstream and generate a new one chip_top.new.bit.

Locating the bootloader and the linker script:

The Bootloader source code: <Project Folder>/fpga/bare_metal/examples/boot.c.

The linker script: <Project Folder>/fpga/bare_metal/driver/test.ld

Configuring the linker script:

Make sure that the entry address is set to 0x4000_0000 in the linker file (line 26 should be “. = 0x4000_0000;”).

Compiling the bootloader and generating the memory initialization file:

Execute the following commands:

```
1 $ cd <Project Folder>/fpga/board/nexys4_dds
2 [ VM ]$ make boot
3 [HOST]$ make bit-update CONFIG=ConfigNameHere
```

The command `make boot` generates the following output (only the last few lines are depicted):

```
riscv64-unknown-elf-gcc -o boot -g -nostdlib -nostartfiles boot.o blk_legacy.o crt.o ctype.o display_opt
ions.o div64.o elf.o ff.o hid.o logo.o lowrisc_mmc.o memory.o mini-printf.o mmc_legacy.o mmc.o sdhci-min
ion-hash-md5.o string.o strt.o syscalls.o time.o ethlib.o dhcp-client.o random.o -T ../driver/test.ld
make[1]: Leaving directory '/media/user/TPSoC_Resources/TPSoC_3A_New/lowrisc-chip-DATE2020-DEMO/fpga/bare
metal/examples'
riscv64-unknown-elf-objcopy -I elf64-little -O verilog /media/user/TPSoC_Resources/TPSoC_3A_New/lowrisc-
chip-DATE2020-DEMO/fpga/bare_metal/examples/boot examples/cnvmmem.mem
iverilog script/cnvmmem.v -o examples/cnvmmem
(cd examples; ./cnvmmem)
First = 40000000, Last = 04000ffff
mv examples/cnvmmem.hex examples/boot.hex
cp examples/boot.hex src/boot.mem
```

Figure 3.5: The output of `make boot`

If the `make boot` command results in the following error

```
user@phelma:/media/user/TPSoC_Resources/TPSoC_3A_New/lowrisc-chip-DATE2020-DEMO/fpga/board/nexys4_dds$
make boot
mkdir -p examples
ln -sf /media/user/TPSoC_Resources/TPSoC_3A_New/lowrisc-chip-DATE2020-DEMO/fpga/bare_metal/examples/M
akefile examples/Makefile
ln: failed to create symbolic link 'examples/Makefile': Read-only file system
Makefile:246: recipe for target 'examples/Makefile' failed
make: *** [examples/Makefile] Error 1
```

The solution is to provide symbolic link creation permissions to your VM. To achieve that, execute the following command on the HOST:

```
1 [HOST]$ VBoxManage setextradata <VM_Name_Here> VBoxInternal2/
SharedFoldersEnableSymlinksCreate/TPSoC_Resources 1
```

As you can see in the last figure, a hex file is generated from the bootloader (boot.c) and is placed at `src/boot.mem`. This file contains the hexadecimal data of the bootloader firmware. This file will be used to patch the original (hardware-only) bitstream `chip_top.bit`

It is worth noting that, if the synthesis is launched again in Vivado (via command line or GUI), this bootloader will automatically be inserted in the final bitstream.

In our case, instead of re-synthesizing each time we modify the bootloader, we will just patch the actual bitstream with the `boot.mem` file without going through the bitstream generation process. This is achieved by the command `make bit-update`.

The bitstream patch (update) step is achieved using the `data2mem` tool (discussed in section 1.7.2). The generated patched bitstream is named `chip_top.new.bit`, and it contains the hardware as well as the initialized BRAM.



```
vivado -mode batch -source ../../common/script/search_ramb.tcl -tclargs lowrisc-chip-imp-DefaultConfig > search-ramb.log
python ../../common/script/bmm_gen.py search-ramb.log src/boot.bmm 128 65536
data2mem -bm src/boot.mem -bd src/boot.mem -bt lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.bit -o b lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit
```

Figure 3.6: The output of make bit-update

7. Programming the FPGA board

To generate the final .MCS memory configuration file execute the following command. The .MCS file will be flashed into the Quad-SPI flash memory. The benefit of this step is that the FPGA will store its configuration in a non-volatile memory, which means that even if it is powered off, it will restore its configuration after restart.

```
1 [HOST]$ make cfgmem-updated CONFIG=ConfigNameHere
```

```
vivado -mode batch -source ../../common/script/cfgmem.tcl -tclargs "xc7a100t 0" lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit
***** Vivado v2018.1 (64-bit)
**** SW Build 2188600 on Wed Apr  4 18:39:19 MDT 2018
**** IP Build 2185939 on Wed Apr  4 20:55:05 MDT 2018
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

source ../../common/script/cfgmem.tcl
# set bit [lindex $argv 1]
# set device [lindex $argv 0]
# puts "BITSTREAM: $bit"
BITSTREAM: lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit
# puts "DEVICE: $device"
DEVICE: xc7a100t 0
# write cfgmem -force -format mcs -interface spix4 -size 128 -loadbit "up 0x0 $bit" -file "$bit.mcs"
Command: write cfgmem -force -format mcs -interface spix4 -size 128 -loadbit {up 0x0 lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit} -file lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit.mcs
Creating config memory files...
Creating bitstream load up from address 0x00000000
Loading bitfile lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit
Writing file lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit.mcs
Writing log file lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit.prm
=====
Configuration Memory Information
=====
File Format      MCS
Interface       SPIX4
Size            128M
Start Address   0x00000000
End Address     0x07FFFFFF

Addr1      Addr2      Date      File(s)
0x00000000 0x003A607B Dec 13 21:44:42 2020 lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/impl_1/chip_top.new.bit
0 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
write cfgmem completed successfully
INFO: [Common 17-206] Exiting Vivado at Sun Dec 13 21:46:32 2020...
```

Figure 3.7: The output of make cfgmem-updated

Now connect the FPGA to the HOST through USB, and executed the following command to flash the .MCS file into the Quad-SPI memory of the FPGA.

```
1 [HOST]$ make program-cfgmem-updated CONFIG=ConfigNameHere
```

This step takes around 1 minute to complete, and it produces the following output if executed correctly:

```
1 vivado -mode batch -source ../../common/script/program_cfgmem.tcl -tclargs "
  xc7a100t 0" lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.
  runs/impl_1/chip_top.new.bit.mcs
2
3 ***** Vivado v2018.1 (64-bit)
4 **** SW Build 2188600 on Wed Apr  4 18:39:19 MDT 2018
5 **** IP Build 2185939 on Wed Apr  4 20:55:05 MDT 2018
6 ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
```



```
7
8 source ../../common/script/program_cfgmem.tcl
9 # set mcs [lindex $argv 1]
10 # set device [lindex $argv 0]
11 # puts "CFGMEM: $mcs"
12 CFGMEM: lowrisc-chip-imp-DefaultConfig/lowrisc-chip-imp-DefaultConfig.runs/
    impl_1/chip_top.new.bit.mcs
13 # puts "DEVICE: $device"
14 DEVICE: xc7a100t_0
15 # open_hw
16 # connect_hw_server
17 INFO: [Labtools 27-2285] Connecting to hw_server url TCP:localhost:3121
18 INFO: [Labtools 27-2222] Launching hw_server...
19 INFO: [Labtools 27-2221] Launch Output:
20
21 ***** Xilinx hw_server v2018.1
22      **** Build date : Apr  4 2018-18:56:09
23      ** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
24
25
26 # open_hw_target
27 INFO: [Labtoolstcl 44-466] Opening hw_target localhost:3121/xilinx_tcf/Digilent
    /210292A6E910A
28 # current_hw_device [lindex [get_hw_devices] 0]
29 # refresh_hw_device -update_hw_probes false [lindex [get_hw_devices] 0]
30 INFO:
31 ....
32 ....
33 ....
34 Mfg ID : 1    Memory Type : 20    Memory Capacity : 18    Device ID 1 : 0    Device
    ID 2 : 0
35 Performing Erase Operation...
36 Erase Operation successful.
37 Performing Program and Verify Operations...
38 Program/Verify Operation successful.
39 INFO: [Labtoolstcl 44-377] Flash programming completed successfully
40 program_hw_cfgmem: Time (s): cpu = 00:00:00.22 ; elapsed = 00:01:30 . Memory (MB
    ): peak = 1763.488 ; gain = 11.656 ; free physical = 1229 ; free virtual =
    34431
41 # endgroup
42 INFO: [Common 17-206] Exiting Vivado at Sun Dec 13 21:58:56 2020...
```

8. Launching the demo

Once flashed, push the **PROG** button of the FPGA board to launch the execution.

If all dip switches are set to zero, you will notice the bootloader output, either on a VGA screen connected to the FPGA board, or in a terminal.

You can use the following command to connect to the terminal¹:

```
1 [VM || HOST]$ picocom -b 115200 /dev/ttyUSB1 --imap lfcr lf
2 picocom v2.2
3
4 port is          : /dev/ttyUSB1
```

¹ ⚠ In case the terminal is launched from the VM, ensure that the external USB (generally `tttyUSB1`) device is recognized and mounted inside the VM (Devices → USB → Digilent USB Device)



```
5 flowcontrol      : none
6 baudrate is     : 115200
7 parity is       : none
8 databits are    : 8
9 stopbits are    : 1
10 escape is      : C-a
11 local echo is   : no
12 noinit is      : no
13 noreset is     : no
14 noloop is      : no
15 send_cmd is    : sz -vv
16 receive_cmd is : rz -vv -E
17 imap is        : lfcrLf ,
18 omap is        :
19 emap is         : crcrLf ,delbs ,
20
21 Type [C-a] [C-h] to see available commands
22
23 Terminal ready
24 ??lling main with MAC = eee1:e2e3e4e0
25 0: 0
26 1: 1
27 2: 1
28 3: 1
29 800: 1e
30 801: 0
31 802: 0
32 803: 0
33 Selftest iteration 1, next buffer = 0, rx_start = 4000
34 Selftest matches=2/2, delay = 11
35 Selftest iteration 2, next buffer = 1, rx_start = 4800
36 Selftest matches=4/4, delay = 11
37 Selftest iteration 3, next buffer = 2, rx_start = 5000
38 Selftest matches=8/8, delay = 12
39 Selftest iteration 4, next buffer = 3, rx_start = 5800
40 Selftest matches=16/16, delay = 22
41 Selftest iteration 5, next buffer = 4, rx_start = 6000
42 Selftest matches=32/32, delay = 40
43 Selftest iteration 6, next buffer = 5, rx_start = 6800
44 Selftest matches=64/64, delay = 78
45 Selftest iteration 7, next buffer = 6, rx_start = 7000
46 Selftest matches=128/128, delay = 154
47 Selftest iteration 8, next buffer = 7, rx_start = 7800
48 Selftest matches=187/187, delay = 223
49 lowRISC boot program
50
51 =====
52 Hello LowRISC! Sat Dec 28 03:13:09 2019: Turn on SW0 for gdb loading , SW1 for SD
    -card loading , or SW2 for Ethernet loading
```



```

File Edit View Search Terminal Help
eth_read(7d48) returned 0
eth_read(7d50) returned 0
eth_read(7d58) returned 0
eth_read(7d60) returned 0
eth_read(7d68) returned 1
eth_read(7d70) returned 0
eth_read(7d78) returned 0
eth_read(7d80) returned 0
eth_read(7d88) returned 0
eth_read(7d90) returned 0
eth_read(7d98) returned 0
eth_read(7da0) returned deadbeef
eth_read(7da8) returned deadbeef
eth_read(7db0) returned deadbeef
eth_read(7db8) returned 0
eth_read(7dc0) returned 0
eth_read(7dc8) returned 0
eth_read(7dd0) returned 0
Selftest matches=187/187, delay = 0
lowRISC boot program
Version Phelma v0.1 - 2020 - Built at Sun Dec 13 22:24:36 2020
=====
Hello LowRISC! Sun Dec 13 22:24:36 2020: Turn on SW0 for gdb loading, SW1 for SD-card loading, or SW2 for Ethernet loading

```

Figure 3.8: The bootloader launch output

This process can also be applied to small applications, with read-only data. However, our objective is to be able to design more complex applications that will not fit in the 64KB BRAM limit, and will use advanced resources such as FPU² and external peripherals.

3.2 Demo 2: An application running from the DDR memory

There are many advantages of running applications in the DDR memory instead of the limited BRAM space: DDR memory provides 128MB free space, which allows to fit in even more complex applications/OSes with possibly more data inputs/outputs. DDR memory is volatile, readable and writable, which resolves the BRAM read-only restriction. Here are the steps to compile, load, and execute an application in the DDR memory space:

1. Creating a new C program

The application should be created in `$LOWRISC_PATH/fpga/bare_metal/examples`. For instance, create a C file named `hello_phelma.c` and write in the following classic example:

```

1 #include <stdio.h>
2 #include <stdint.h>
3 #include "lowrisc_memory_map.h"
4 #include "mini-printf.h"
5
6 char text[] = "Vafgehpvgvba frgf jnag gb or serr!" ;
7
8 void foo(){
9     int i=0;
10    while(text[i]){
11        char lower = text[i] | 32;
12        if ( lower >= 'a' && lower <= 'm' )
13            text[i] += 13;
14        else if ( lower > 'm' && lower <= 'z' )
15            text[i] -= 13;

```

²In case the application performs floating-point computations, remember to activate. Please refer to Lab2.

```

16     i++;
17 }
18 }
19
20 volatile int wait = 1;
21 int main(int argc , char ** argv) {
22
23     printf("Welcome to this test application written to document the lowrisc
platform:");
24
25     while(wait);
26
27     printf("Old text: %s \n",text);
28     foo();
29     printf("New text: %s \n", text);
30
31     while(!wait);
32
33     return 0;
34 }

```

Listing 3.1: hello_phelma.c

2. Add the executable target to the Makefile.

Inside `$LOWRISC_PATH/fpga/bare_metal/examples/Makefile`, line 34, add the target name `hello_phelma` to the list of possible targets.

```

1 TARGETS = hello_phelma dram hello sdcard trace jump flash tag selftest eth
lowrisc

```

3. Modify the entry address in linker file to `0x8000_0000`
4. Compiling the application

In the same directory, execute the following command

```

1 [ VM ]$ make hello_phelma.riscv

```

This generates a `.elf` file format named `hello_phelma.riscv` and this is the application to be transferred via GDB to DDR memory of the FPGA. No need to convert, GDB will do all the automatically.



Do not forget to add the `.riscv` suffix at the end of the name.

Loading the application using GDB

Figure 3.9 summarizes the debugging scheme, which is very similar to Spike and the C emulator. The only difference is that, in this case, OpenOCD communicates with the external target via UART instead of a TCP socket on the same machine. The used configuration file `openocd-nexys4ddr.cfg` is also different and specific to lowRISC.

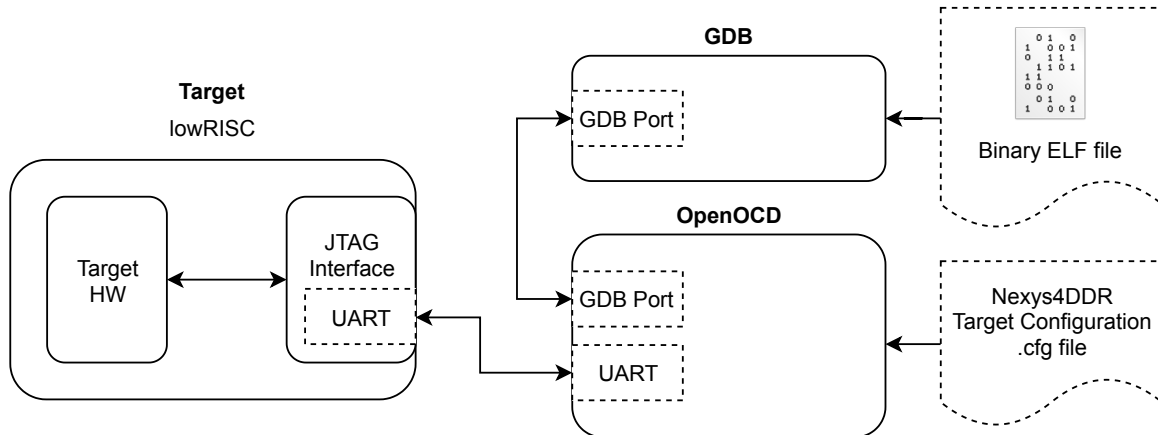


Figure 3.9: Software debugging using GDB and OpenOCD

First, the FPGA board should already be flashed with a valid bootloader (boot.c). It should be launched with the switch SW0 set to ‘1’ in order to activate GDB loading mode.

→ Keep the picocom serial terminal open.

→ In a new terminal, launch OpenOCD:

OpenOCD³ can be found in the resources directory at `TPSoC_3A_New/lowrisc-chip-debugger`

```

1 [ VM ]$ cd lowrisc-chip-debugger
2 [ VM ]$ make debug

```

You get the following log

```

1 ~/lowrisc-chip-debugger$ make debug
2 openocd -f openocd-nexys4ddr.cfg
3 Open On-Chip Debugger 0.10.0+dev-00165-gdc312f5 (2019-06-18-11:37)
4 Licensed under GNU GPL v2
5 For bug reports, read
6 http://openocd.org/doc/doxygen/bugs.html
7 adapter speed: 10000 kHz
8 Info : ftdi: if you experience problems at higher adapter clocks, try the
9 command "ftdi_tdo_sample_edge falling"
10 Info : clock speed 10000 kHz
11 Info : JTAG tap: riscv.cpu tap/device found: 0x13631093 (mfg: 0x049 (Xilinx),
12 part: 0x3631, ver: 0x1)
13 Info : dtmcontrolidle=5, dmi_busy_delay=1, ac_busy_delay=0
14 Info : dtmcontrolidle=5, dmi_busy_delay=2, ac_busy_delay=0
15 Info : dtmcontrolidle=5, dmi_busy_delay=3, ac_busy_delay=0
16 Info : dtmcontrolidle=5, dmi_busy_delay=4, ac_busy_delay=0
17 Info : Disabling abstract command reads from CSRs.
18 Info : Disabling abstract command writes to CSRs.
19 Info : [0] Found 1 triggers
20 Info : Examined RISC-V core; found 1 harts
21 Info : hart 0: XLEN=64, 1 triggers
22 Info : Listening on port 3333 for gdb connections
23 Info : Listening on port 6666 for tcl connections

```

³You can clone it on your machine using the following command

```

git clone --branch refresh-v0.6 --depth 1 --single-branch --recursive https://github.com/lowRISC/
lowrisc-quickstart.git lowrisc-chip-debugger

```



```
22 Info : Listening on port 4444 for telnet connections
23 Info : accepting 'gdb' connection on tcp/3333 # Appears when launching GDB and
    connecting it (next step)
```

→ In another terminal, launch GDB with the compiled file `hello_phelma.riscv`

→ Execute the following commands in GDB in order to load the application to the DDR memory (similar to debugging using the C Emulator):

```
1 $ riscv64-unknown-elf-gdb <PATH.TO.EXECUTABLE.BINARY>
2 (gdb) target remote localhost:3333
3 Remote debugging using localhost:3333
4 warning: Can not parse XML target description; XML support was disabled at
    compile time
5 0x000000004000304c in ?? ()
6 (gdb) load
7 Loading section .text, size 0x313a lma 0x80000000
8 Loading section .text.startup, size 0x5e lma 0x8000313a
9 Loading section .rodata.str1.8, size 0xb30 lma 0x80003198
10 Loading section .rodata, size 0x3100 lma 0x80003cc8
11 Loading section .data, size 0x22 lma 0x80006dc8
12 Loading section .sdata, size 0xa4 lma 0x80006df0
13 Start address 0x80000000, load size 28302
14 Transfer rate: 9 KB/sec, 4717 bytes/write.
15 (gdb) print wait
16 $1 = 1
17 (gdb) print text
18 $2 = "Vafgehpvba frgf jnag gb or serr!"
19 (gdb) print wait = 0
20 $3 = 0
21 (gdb) c
22 Continuing.
23 ^C
24 Program received signal SIGINT, Interrupt.
25 0x0000000080002e66 in main (argc=<optimized out>, argv=<optimized out>)
26 at .../fpga/bare-metal/examples/hello.c:39
27 39 while(!wait)
28 (gdb) print text
29 $4 = "Instruction sets want to be free!"
30 (gdb) print wait
31 $5 = 0
```

Loading the application using the SD card

To load the executable using the SD-card it is enough to rename the executable from **hello.riscv** to **boot.bin** and put the file `boot.bin` (the name is hard-coded in `boot.c`, this can be changed by modifying the code, compiling and merging with the bitstream (`make bit-update`)) in the SD-card then configure the switches in order for the bootloader to read from the SD-card. Insert the SD card in the FPGA board. The bootloader will automatically load any file named `boot.bin` from the SD-card to the FPGA in this configuration case.

The function `sd_main(int sw)` in `boot.c` is responsible for reading the executable named `boot.bin`:

```
1 ...
2 // Open a file
```

```

3 printf("Load boot.bin into memory\n");
4 fr = fopen(&fil , "boot.bin", FA_READ);
5 if (fr) {
6     printf("Failed to open boot!\n");
7     return (int)fr;
8 }
9 ...

```

Listing 3.2: Load boot.bin from SD to memory

3.3 Results visualization

For result output visualization purposes, you can use the `printf()` to print over the UART link, or the VGA to print the image plus an overlay for example. The original version of lowRISC developed by the lowRISC foundation contains a different (more complex) VGA driver. The old one is composed of a palette and a frame buffer which also acts as a basic visual terminal over VGA. The one we developed is more simpler, and which only supports visualizing images. The two drivers are presented in more details in the next part.

In figure 3.11, the result of the test application is visualized using the original version of VGA driver.

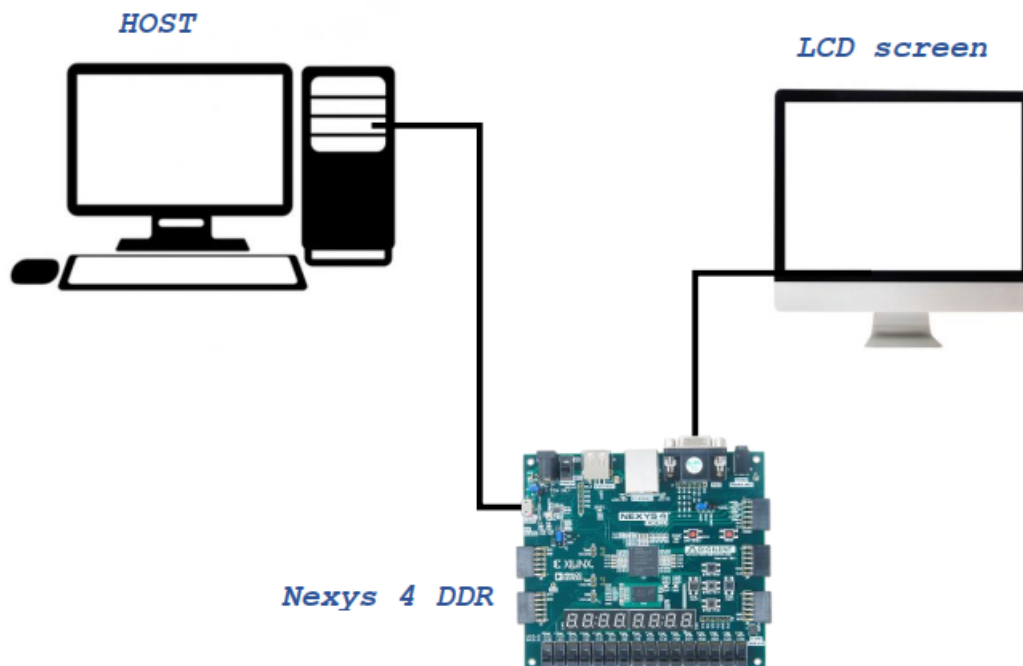


Figure 3.10: FPGA-Host-Output context

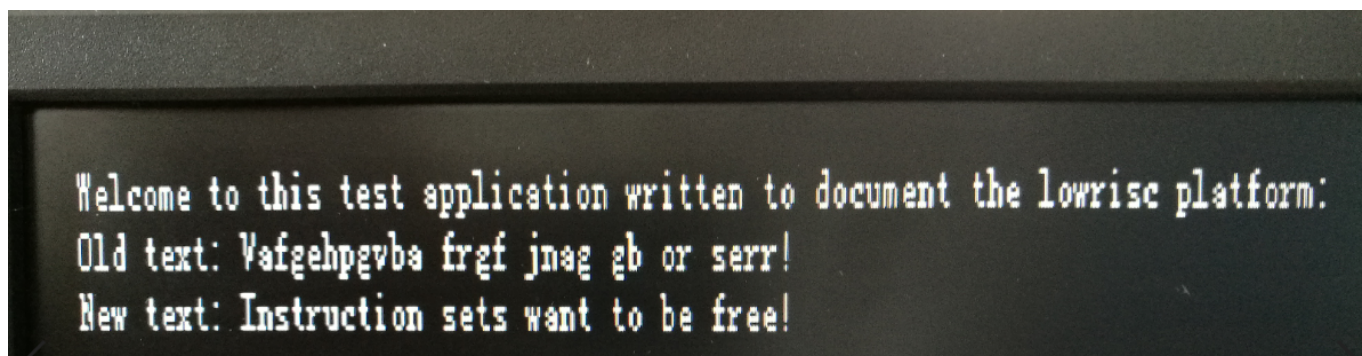


Figure 3.11: Results of the application on the original VGA display.

Part II

Projects Propositions

This part presents the lowRISC-based proposed projects and some guidelines on the implementation and expected deliverables.

Chapter 4

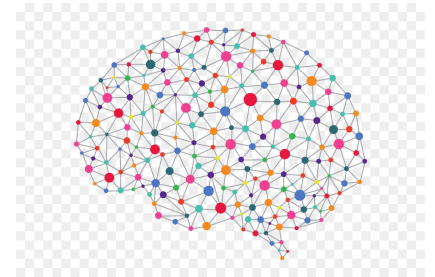
Implementation of a software CNN application on lowRISC

4.1 General introduction

Artificial neural network are networks inspired by the biological brain neural networks and designed using statistical methods. They are a subcategory of **Artificial Intelligence** (AI) algorithms.

A **Convolutional Neural Network** is a very know category of artificial neural networks. They are inspired specifically by the animal visual cortex. Convolutional neural networks are widely applied in image and video processing, object classification and recognition, recommendation systems and natural language processing.

The objective of this project is to implement a CNN classification application in software, on a RISC-V based softcore architecture. The target application implements a classic neural network, trained using the very known CIFAR-10 dataset. The training is done previously on the host machine, and only inference (classification) is considered.



Objectives:

- Acquiring a system-level view of the system: including the hardware architecture (i.e. processor architecture, memory, interconnects, peripherals ...), the software architecture (booting sequence, baremetal core, drivers, debugging ...).
- Understanding the challenges of system-level design (address mapping, performance / resource / power constraints, security issues ...).
- Acquiring basic hands-on experience with HW/SW integration and co-design (integrating a new peripheral, making HW/SW design decisions, writing basic HAL for custom drivers ...).

4.2 Project description and provided resources

4.2.1 The CIFAR-10 library

The CNN application to be developed in this project uses the CIFAR-10 dataset for training. This dataset consists of 60,000 32x32 RGB images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images.

The data set is divided into five training batches and one test batch, each with 10,000 images. The test batch contains exactly 1,000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images

from one class than another. Between them, the training batches contain exactly 5,000 images from each class.

Here are the classes in the data-set, as well as 10 random images from each:

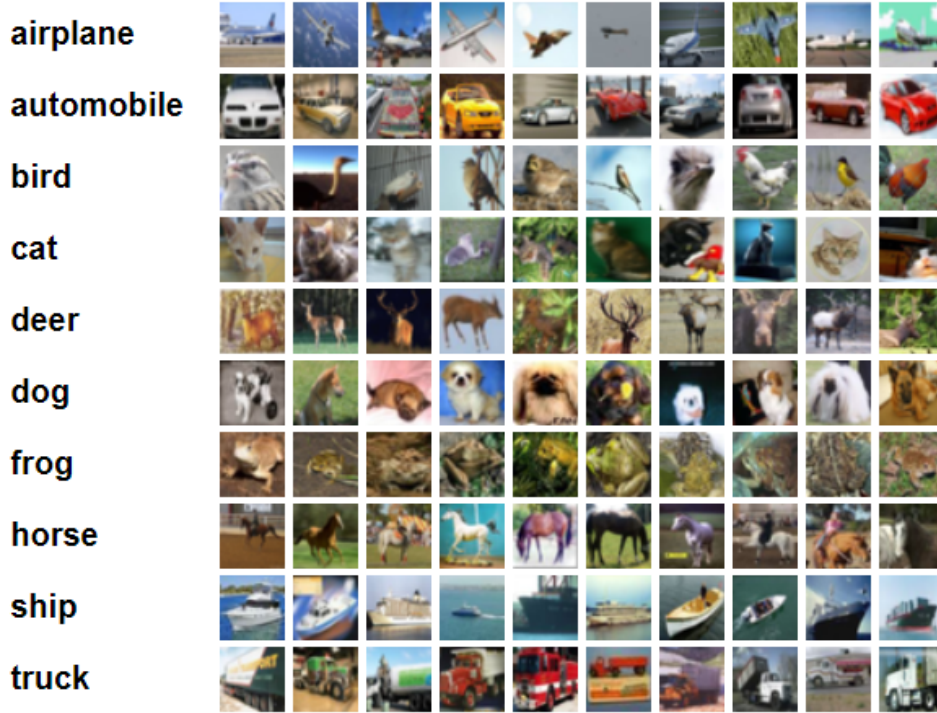


Figure 4.1: The CIFAR-10 library

4.2.2 CNN inference steps

This network takes as input a 24x24 color image, that is to say with three channels of entries that correspond to the colors red, green and blue. It then has a series of convolutions, interlaced with layers of ReLU (max function $(0, x)$) and maxpool (3×3 , stride = 2).

Finally, a step of reshape and a perceptron (fully-connected layer) are applied at the end of the network. At the output of the perceptron, there are 10 values corresponding to the probabilities of belonging to each of the 10 classes of the CIFAR-10 bank.

The network (weights + biases) should be constructed from the CIFAR-10 dataset. If the network is well-trained and correctly implemented, we should get a target success rate of about 70% on the test subset.

The input layer

For this CNN application the images of the CIFAR-10 library are cut in 24x24 pixels image, the cut being centered.

After cutting the image, the pixels values are normalized using the next equation:

- $m'(i,j)$: The pixel value of the result image.
- $m(i,j)$: The pixel value of the original image.

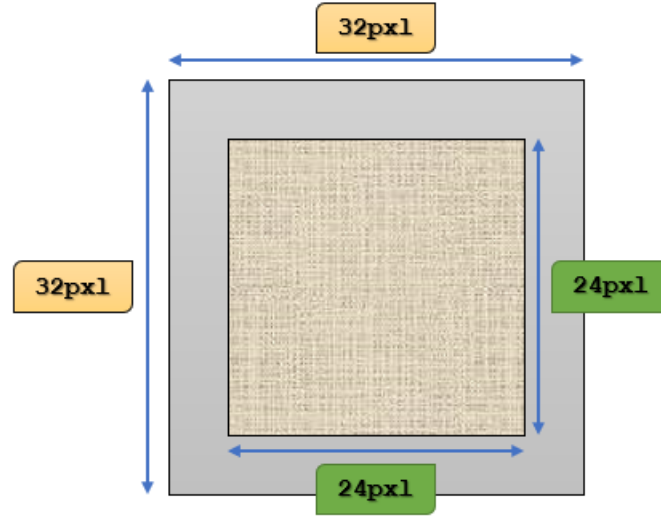


Figure 4.2: Image preparation

$$m'_{i,j} = \frac{m_{i,j} - \mu}{\max(\sigma - \frac{1}{\sqrt{N}})}$$

- μ : The average value of the pixels.
- σ : Standard deviation.
- N : Pixels number.

This leads to centered-reduced values of image pixels, and thus a set of values close to 0 and centered around 0. The results are three layers of 24x24 pixels (R, G, B) which is the formal input of the CNN application. The input is an array of $24 \times 24 \times 3 = 1728$ values.

Intermediate layers and final output

The intermediate layers are mainly implemented as convolutions. They can be implemented using standard math floating-point functions.

The final output is a 10-element vector, containing the probabilities of the image belonging to each of the 10 classes. The element having the maximum probability is most likely the target class. An overall description is depicted in figure 4.3.

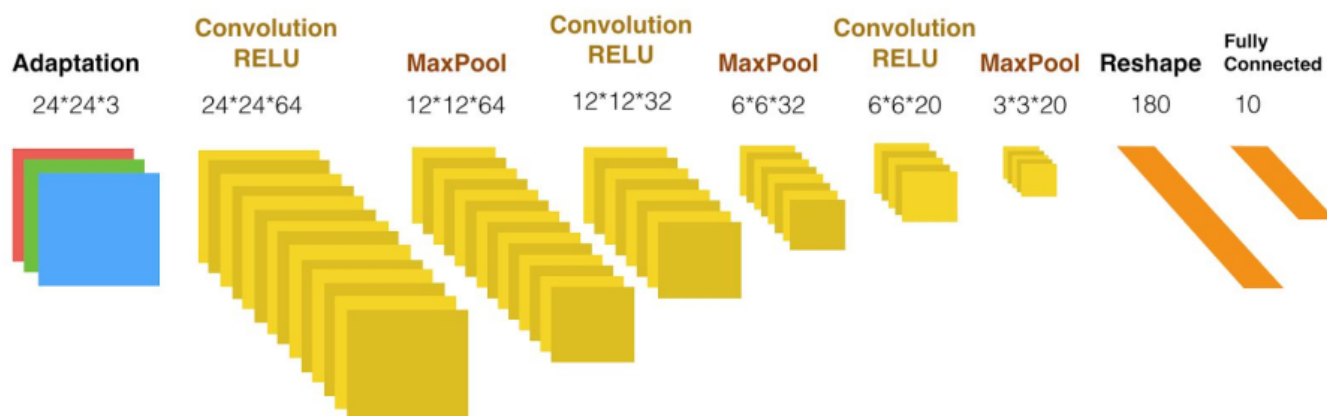


Figure 4.3: The CNN application design

4.3 Project deliverable

Check the presentation joined with this document.

Appendix 2: Tips to speed up development process

Hardware

- Set the `periph_soc.sv` as the top-level module, because synthesizing the whole system will take too much time.
- If only a tiny thing is changed on your design, do not synthesize everything from scratch, create a new design run, and use the “incremental compiling” feature of Vivado.
-

Changelog with regards to the Frame-Buffer version of lowRISC

- The VGA controller has been replaced by a simpler one.
- The new VGA driver needs 25Mhz instead of 120 MHz → `clk_gen` IP is reconfigured to output a 25Mhz clock in the `clk_pixel` output.