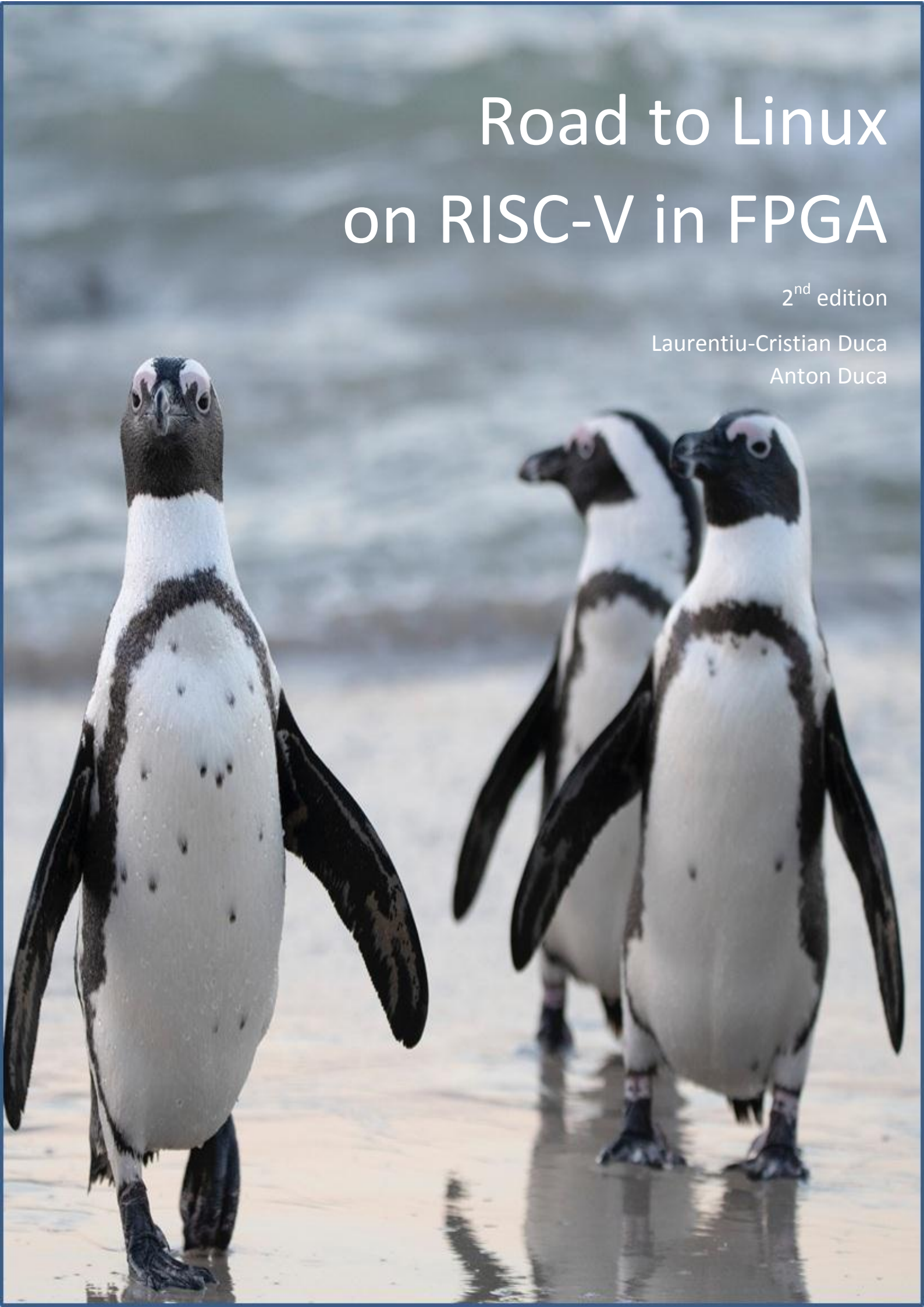
Road to Linux on RISC-V in FPGA

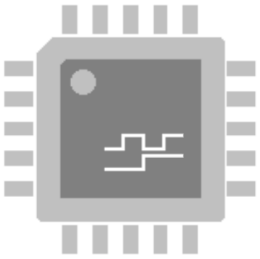
2nd edition

Laurentiu-Cristian Duca
Anton Duca



This page is intentionally left blank

Road to Linux
on RISC-V in FPGA
2nd edition



Laurentiu-Cristian Duca
Anton Duca

Road to Linux on RISC-V in FPGA, 2nd edition

Version history

Date	Description
2024-05-25	True dual-core rlsoc
2023-11	Pseudo dual-core rlsoc
2022-05	Single core rlsoc

Contents

This page was intentionally left blank

1. Introduction

1.1 Before we begin

This book is intended to be used by the students of all ages who want to learn about porting the Linux operating system on the RISC-V32 processor in FPGA.

This book is written after the notes that we took while studying two related projects: the TinyEMU RISC-V emulator [1] and the RVSoC RISC-V32 system implementation in Verilog [2].

As prerequisites, the readers should have programming skills and be familiar with C and Verilog programming languages. The level is intermediate to advanced.

Open the book pdf in multiple browser tabs and study it using the Bootlin's Linux LXR [3] and the v2html Verilog converter [4] ; it will ease up your work!

When building a documentation project using v2html, please note that, depending of the enabled or disabled flags in verilog, some modules may be added or removed from the documentation project.

To download the updated projects presented in this book, please navigate to the following address: https://github.com/laurentiuduca/road-to-linux.

Finally, please notice that if you have any question regarding the material presented here, do not hesitate to write us.

I hope that you will find it useful!

1.2 What this book is about

This book presents an improved fork of the RVSoC project [2] version 53. RVSoC is developed at the Kise Laboratory (Arch Lab) of the Tokyo Institute of Technology (Tokyo Tech), under MIT license. It is a Verilog implementation of the RISC-V32 single core processor with MMU, console and a disk kept in RAM. This project was implemented in the Xilinx Nexys A7 (older name Nexys 4 DDR) and Arty A7 development boards.

In the rlsoc1 version we rewrote part of the memory controller to support all unaligned memory accesses; we also added some new features. The book documents porting and configuring the Linux kernel for this system and also explains the changes needed to successfully make these jobs.

TinyEMU [1] is a RISC-V 32/64/128 emulator developed by Fabrice Bellard under MIT license. It is written in C and because the RVSoC is related to TinyEMU RISC-V simulator, we will take a closer look at this one, too.

We used Ubuntu 20.04 as the development host.

From now on, in this book:

- we will use the rlsoc notation for my version of RVSoC and tinyemu for TinyEMU;

- we will refer to Xilinx and Vivado as xilinx and vivado;

- we will refer to Nexys A7 as nexys4 and to Arty A7 as arty35t.

There is a new chapter that describes a true dual-core version of the rlsoc system – called rlsoc-tn or rlsoc2 - implemented on the $30 tang-nano-20k fpga board.

1.3 Computer architecture shortly

The simplified architecture of a computer machine is presented in Fig. 1-1. Learning how the CPUs, memory and I/O devices work together makes one understand how a computer works.

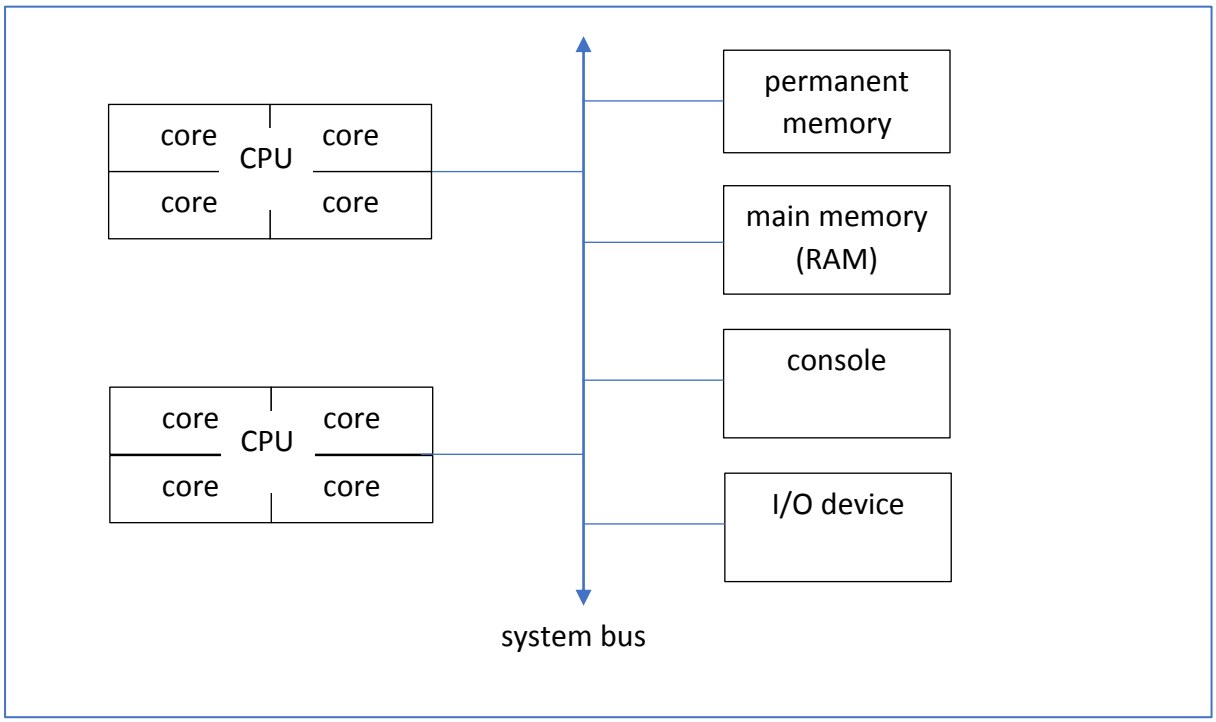


Fig. 1.2-1. Simplified architecture of a computer machine

A program is a set of instructions and data and, in order to be executed, is loaded from permanent memory to main memory and then, parts of it, to the cache memory of the CPU which executes the program.

Instructions and data are organized in addresses; each address corresponds to a piece of data. A CPU accesses RAM, permanent memory and I/O devices by setting specific addresses on the system bus. The total numbers of addresses a CPU can access is called CPU address space. If an I/O device can be accessed at a particular address then it is said that the I/O device it is "mapped" at that address.

The console is a special device which represents the interface between the computer's user and the computer itself. While modern consoles are graphical, the classic console is a text only serial interface and is available in all computer prototypes.

The processor core accesses the main memory via the memory management unit (MMU) in a scheme similar to Fig. 1-2. Between the MMU and main memory we may find a per-core cache and a core-shared cache (last one missing in the figure).

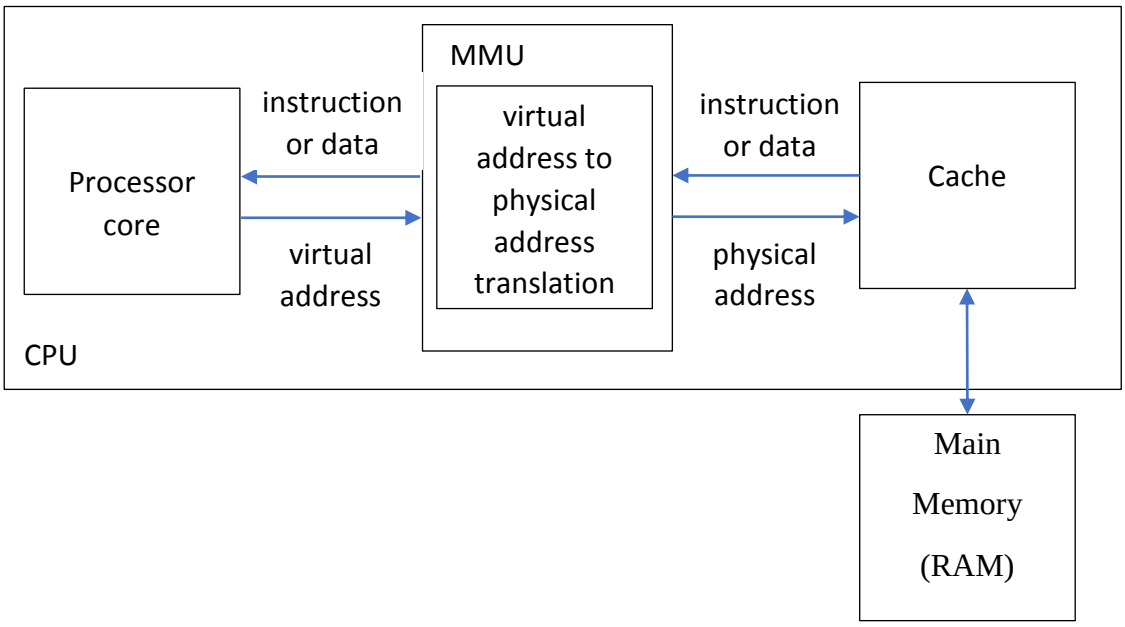


Fig. 1.2-2. Processor core, MMU, cache and main RAM memory

The RAM memory is much faster than the permanent memory; the cache memory is smaller but much faster than the main memory. The cache is bigger but slower than the processor core's register set.

The process that transfers the main memory locations to cache memory is specific to the cache mapping. There are three generally used mapping methods: direct mapping, full associative mapping and set associative mapping.

In full associative mapping, a main memory location can be mapped anywhere in the cache memory. A cache line is the pair [address, data] of the mapped memory location. The disadvantage is that searching the cache is slow.

In direct mapping, multiple blocks of memory can be mapped in the same cache memory block. For example, for a 1024 blocks main memory and a 32 blocks cache memory, there can be mapped 1024/32 = 32 main memory blocks to the same cache memory block. In this case, the address is split into index and tag. The cache line contains contains the tag and the value of the memory location and is indexed by the index part of the address. Example:

- memory addresses 02000 and 03000 have the tags 02 and 03 and the index 000, so can be stored only at cache index 000;
- supposing that the word with value 4567 at address 02000 is stored in cache; then the cache entry 000 contains tag 02 and value 4567.

Directly mapped cache has the advantage that searching if a memory location is present in cache is very fast. The disadvantage is that, when the processor accesses memory locations with different tags but the same index, the cache miss rate has a very high value.

Set associative mapping is a combination between associative and direct mapping. The cache memory is organized in sets. Inside the set, we have associative mapping. If the cache contains only one set, we have associative mapping, while if each set contains only one memory location, then we have direct mapping.

In order to run modern operating systems like Linux, Windows and so on, each CPU core has a MMU attached to it. CPU programs use virtual addresses. Each program has its own virtual to physical mapping. Even if different programs use the same virtual addresses, they access completely different data. A program can not alter the memory used by another program.

Main memory uses pages; for example, in rlsoc, one page has 4096 bytes. Page fault is somehow an equivalent of the cache miss. The operating system (for example, Linux) keeps in main memory the page table for each process. The page table contains an entry for each page of the process. It also a validity bit which gives information if the page is present in main memory.

In order to easier understand how the MMU works let's consider the simplest case where the virtual address is made only by the virtual page number (VPN) and the page offset. VPN is the index in the page table. The physical address contains the physical page number (PPN) and the page offset. The translation between a virtual address to a physical address is shown in the figure below.

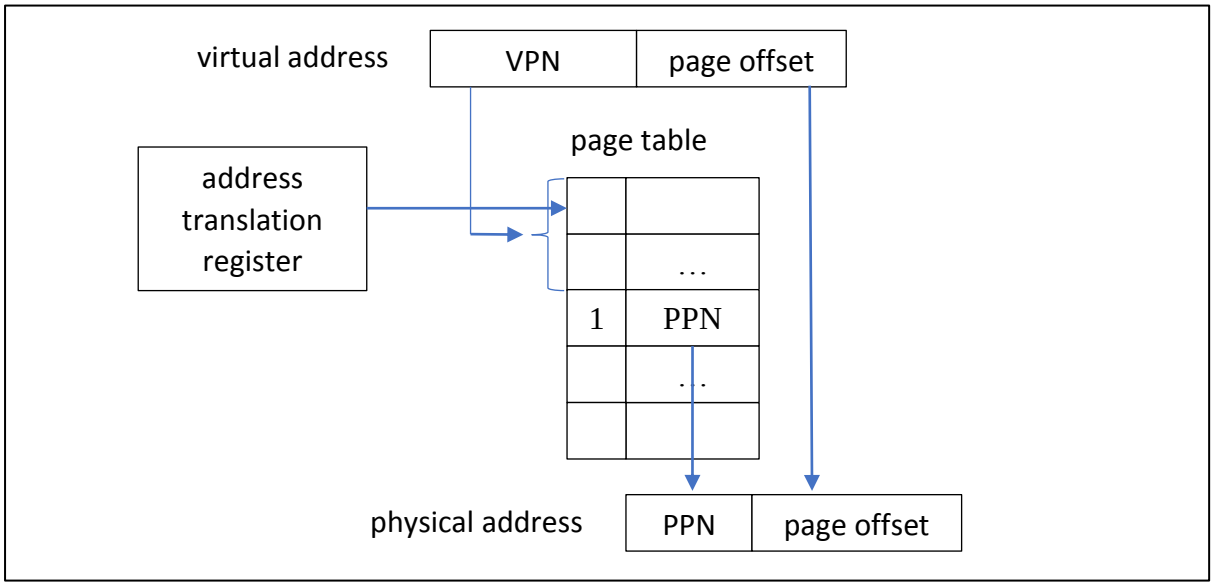


Fig. 1.2-3. Virtual to physical address translation

The address translation register points to the beginning of the page table. The VPN is an index in this table and the MMU will choose the PPN at this index location. When the

valid bit is one, the page is in RAM. The physical address is the concatenation of the PPN and the page offset.

The MMU has its own cache named translation lookaside buffer (TLB) which contains physical addresses that were the result of virtual addresses translation. Usually this cache has over 90% hit rate.

Modern operating systems like Linux require processors with MMU.

2. Building and testing RLSoC

2.1 Testing RLSoC on the FPGA board

The base folder used for testing the rlsoc project is **rvsoc_src_ver053** from the rvsoc_src_ver053-20220521-1310.tgz archive.

We tested rlsoc on the Nexys A7, but we also have built the bit file for arty35t (for which we simulated rlsoc).

The file binary/**initmem.bin** contains BBL (Berkeley Boot Loader), Linux kernel, the initramfs image and the dtb (device tree blob). After programming the FPGA, this file must be sent through the usb-to-serial interface to the board memory, from where it will be executed by rlsoc.

So, to test rlsoc on FPGA do the following:

- use vivado_lab to program the FPGA by selecting **nexys4.bit** for Nexys A7 and **arty35t.bit** for Arty A7 from the **bitstream** folder;

- wait for the led2 on the board to become visible (this announces that the board is ready);

- install and run **gtkterm** (which will show the serial console) configured with 8Mbps and the USB serial port adapter /dev/ttyUSBx where x is your serial device number that your board is connected to; on Windows you can use Tera Term;

- edit **serial_sendfile.py** from the **binary** folder such that it contains the correct /dev/ttyUSBx device id described above; on Windows you can use Tera Term Sendfile;

- run the following command from the **binary** folder, to transfer the Linux system to the board (first install pySerial with the command **pip3 install pyserial**):

```
sudo python3 serial_sendfile.py 8 initmem.bin
```

- led0 will blink while transferring **initmem.bin**;

- when led0 stops blinking the system should boot and the messages are shown in **gtkterm**.

2.2 Simulating RLSoC

The Verilog sources are placed in the **src** folder. The top module for simulation is **m_topsim** and sits in **top.v**. It instantiates the risc-v processor **m_RVCoreM** and the **m_mmu** Verilog modules; **top.v** also implements the **m_dram_sim** module which is a memory simulator. **m_mmu** is the most important module and is contained in **mmu.v**.

I simulated it in verilator and Icarus. It can also be simulated in VCS. To simulate, the **SIM_MODE** flag must be defined in **define.vh** from the **src** folder.

If you want to simulate arty35t then enable the flag **ARTYA7** in **define.vh**. Otherwise disable it.

The simulator uses **init_kernel.txt** which contains the bbl, linux with initramfs, and the dtb, and **init_disk.txt** which should contain the disk image, but, because we use initramfs (as a follow up to the fact that, if used, the disk is also stored in RAM) we set **init_disk.txt** to 0 bytes. More on this we will see in the compiling subchapters. In the **m_topsim** module, there is a code section which loads in the **mem** variable of **m_bu_mem** module the contents of **init_kernel.txt** and **init_disk.txt**. In simulation, this **mem** variable represents the memory.

The simulation shows the system booting bbl and linux. The login in linux is made automatically with user root, because in **mmu.v**, the **cons_fifo** variable is initialized with **root<CR><CR>** and the console takes the input from this variable. After linux login, you can enter the desired command in the **fcmd.txt** file (maximum 16 characters) and then change the command id in the **fid.txt** file in order to instruct the simulator that a new command is available in **fcmd.txt**. The **read_file** module is instantiated in **mmu.v** and reads **fid.txt** and **fcmd.txt**, but initially waits linux to boot – and the time to wait is about **ENABLE_TIMER** + 10000000. **ENABLE_TIMER** is defined in **define.vh**. Please note that linux 5.0 boots about two times faster than linux 5.13.19. In simulation, linux 5.0 boots in about 1 hour on a 2GHz intel i5 processor in verilator.

Verilator is the fastest simulator that we have used. To simulate in verilator, do the following:

- install verilator with the command:

  sudo apt install verilator

- for Verilator 5, in src/Makefile must be appended -Wno-LATCH to VERIFLAGS:

  VERIFLAGS += -Wno-WIDTH -Wno-CASEINCOMPLETE -Wno-COMBDLY -Wno-LATCH

- build the simulator executable by running the following command in the **src** folder:

  make veri

- run the simulator:

  ./simv

To simulate in Icarus, do the following:

- install Icarus Verilog with the command:

sudo apt install iverilog

- build the simulator:

make icarus

- simulate:

make run-i

After simulation, the src folder can be clean with the command:

make clean

2.3. Building RLSoC

2.3.1 Preliminary considerations

For implementing in FPGA the RLSoC project we used Xilinx Vivado ML 2021.2, Standard Edition. A Vivado Webpack license file can be obtained for free from the Xilinx website after registration. If you are unfamiliar with creating a Vivado project and programming the FPGA board we recommend reading [5].

Note 2.3.1-1: if vivado loops indefinitely, close it and rerun it.

The top module for rlsoc synthesis is **m_main** which sits in **main.v**. To be able to build rlsoc, the variables **SIM_MODE** and **SIM_MAIN** from **define.vh** must be disabled.

We show how to build the project from zero and also how to use the already built vivado projects **nexys4.xpr** or **arty35t.xpr**. The projects built in vivado allow using a 100 MHz clock for the rlsoc; the internal clock is 104MHz. For each project there are 16 critical warnings for synthesis and 25 for place and route.

Note 2.3.1-1: Even if vivado reports WNS < 0 and not null TNS for 104MHz clock, the project **nexys4.xpr** works. If you want a WNS > 0 and a null TNS for the project, set the output clock of clk_wiz_1 to 75MHz (see subchapters 2.3.7 and 2.3.8 for instructions).

Note 2.3.1-2: Please make sure that the synthesis process does not have the warning “could not open $readmem data file 'ucimage.hex'”.

If you use a newer version of vivado, then you may have to upgrade the IPs that are used in the project, to a new version.

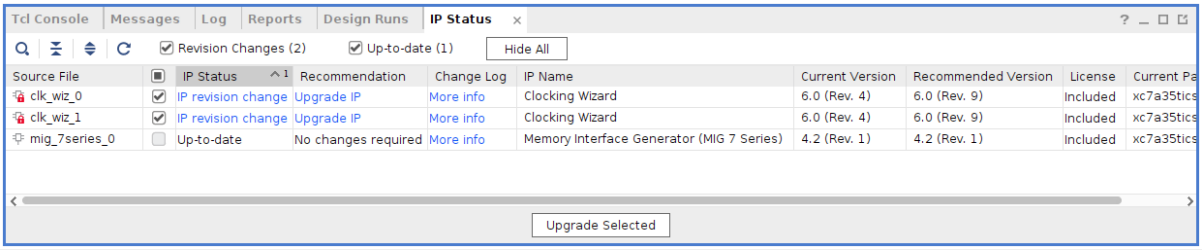


Fig. 2.3.1-1 IP revision change

If you want to build from zero, follow the next instructions.

If you want to build for arty35t then enable the flag **ARTYA7** in **define.vh**. Otherwise disable it.

First, create a new vivado project and specify the device part xc7a100tcsg324-1 for nexys4 and xc7a35ticsg324-1L for arty35t. Add the **src** folder to your project. Then add the constraints file **nexys4.xdc** or **arty35t.xdc** from the **constrs** folder.

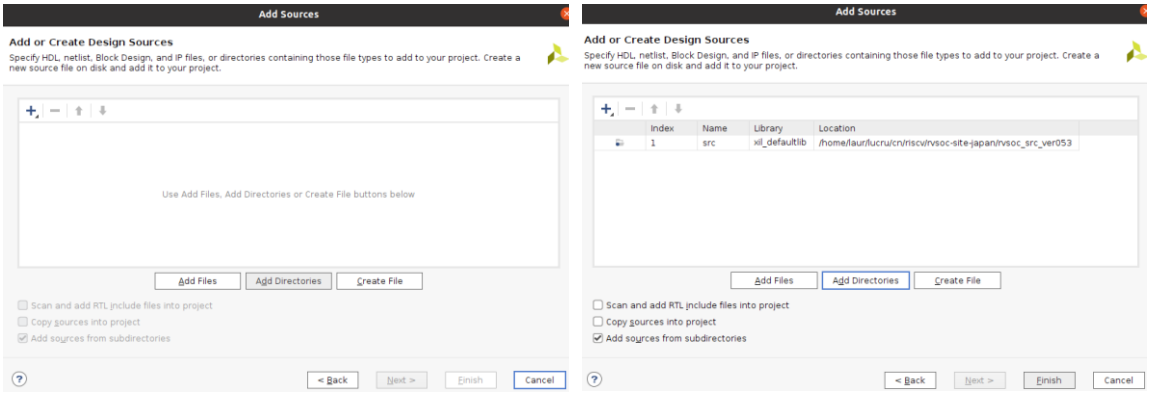


Fig. 2.3.1-2. Add directories in Xilinx Vivado

Then we have to create the memory controller. We will be using the Xilinx MIG memory interface generator 4.2 and Xilinx Clocking Wizard 6.0 available from the Vivado IP catalog. After that you can synthesize, implement design and generate the bit file. You will find the generated bit file in **nexys4.runs/impl_1/m_main.bit** or **arty35t.runs/impl_1/m_main.bit**.

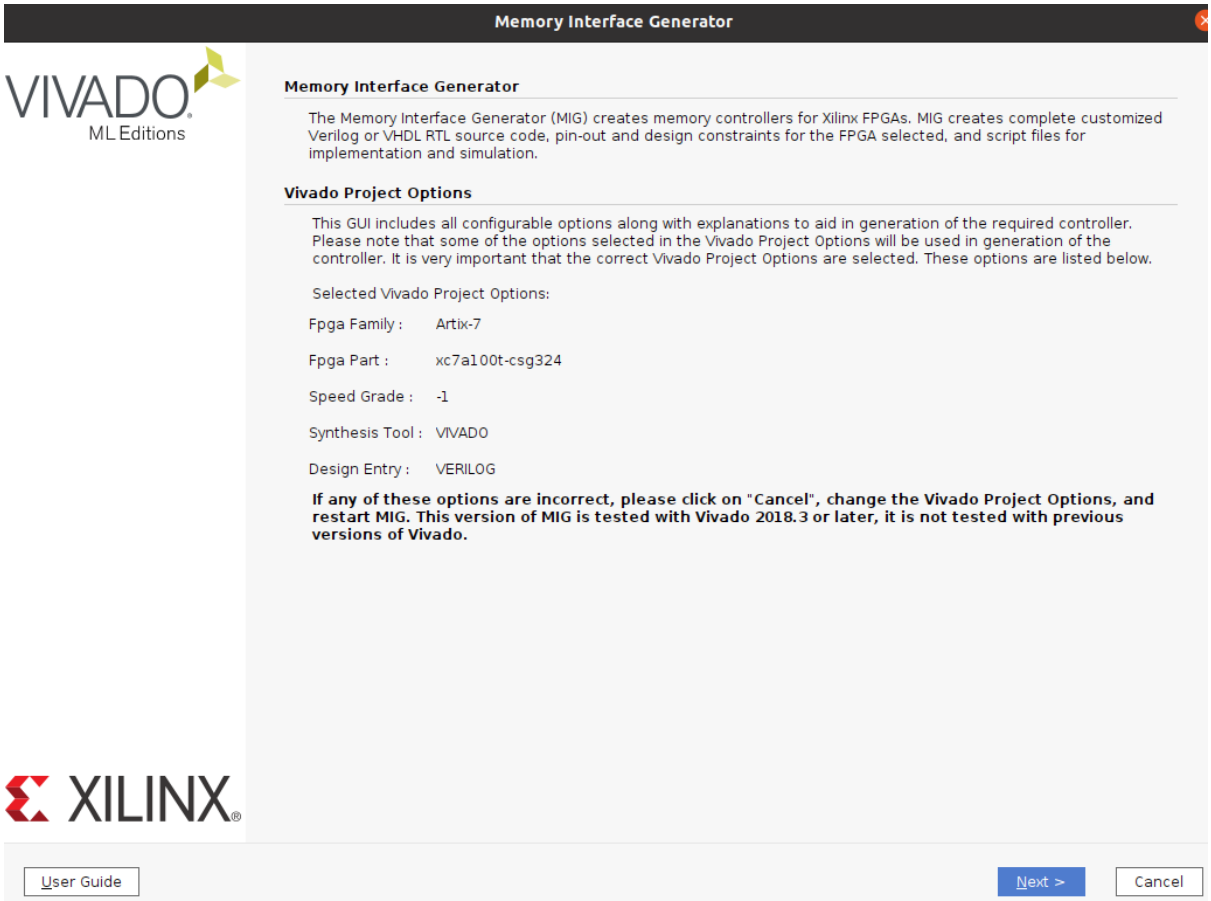
2.3.2 Create MIG for Nexys A7

The settings for the Xilinx MIG for Nexys A7 are shown in its board reference manual [6]. They are presented in the following table.

Memory type	DDR2 SDRAM
Max. clock period	3000ps (667Mbps data rate)
Recommended clock period (for easy clock generation)	3077ps (650Mbps data rate)
Memory part	MT47H64M16HR-25E
Data width	16
Data mask	Enabled
Chip Select pin	Enabled
Rtt (nominal) – On-die termination	50ohms
Internal Vref	Enabled
Internal termination impedance	50ohms

Table 2.3.2 Xilinx MIG settings for Nexys A7

Click on IP-Catalog and search for MIG. We now show the MIG generation step-by-step.



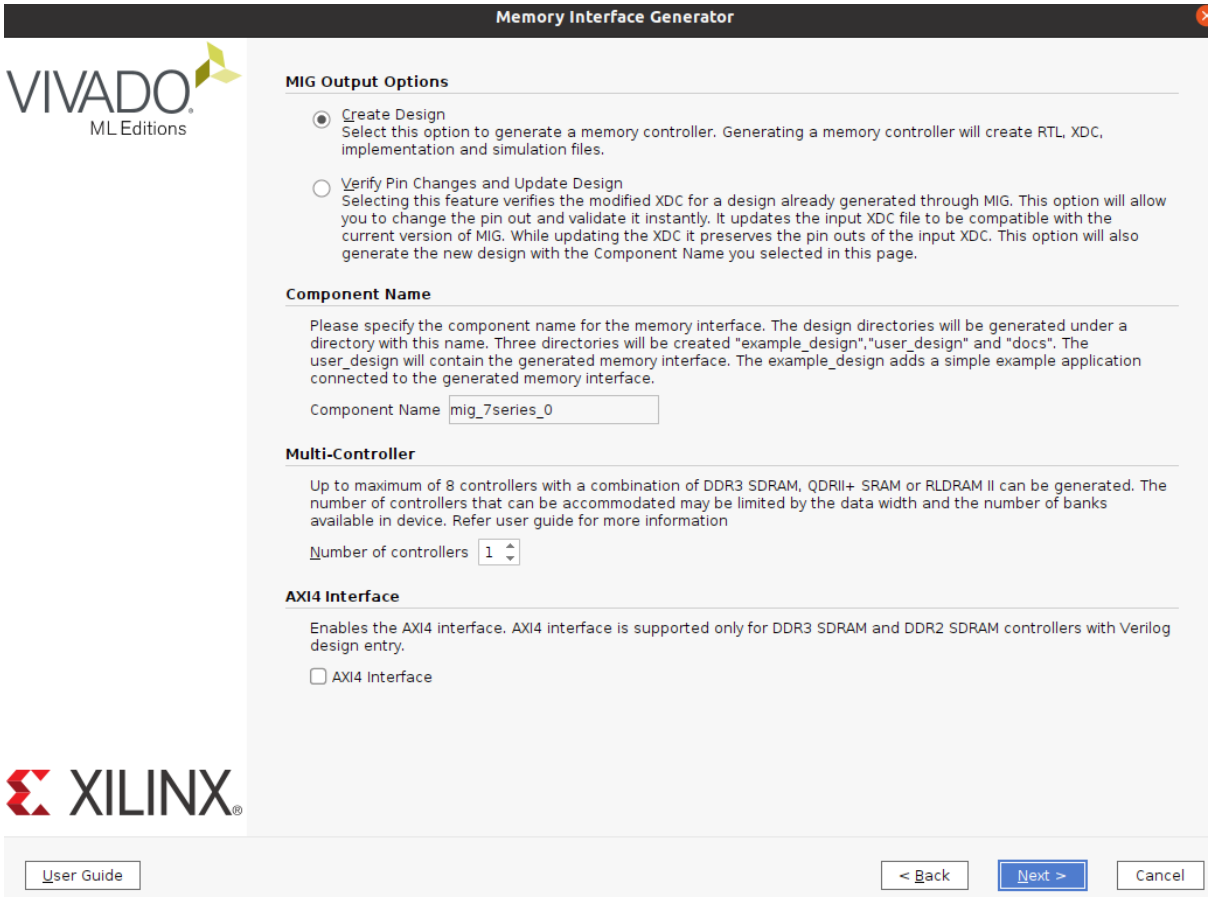


Fig. 2.3.2-1. Nexys A7 MIG generation steps 1a and 1b.

RLSoC does not use the AXI interface; it uses the MIG user interface UI. For more info on MIG UI, see [7].

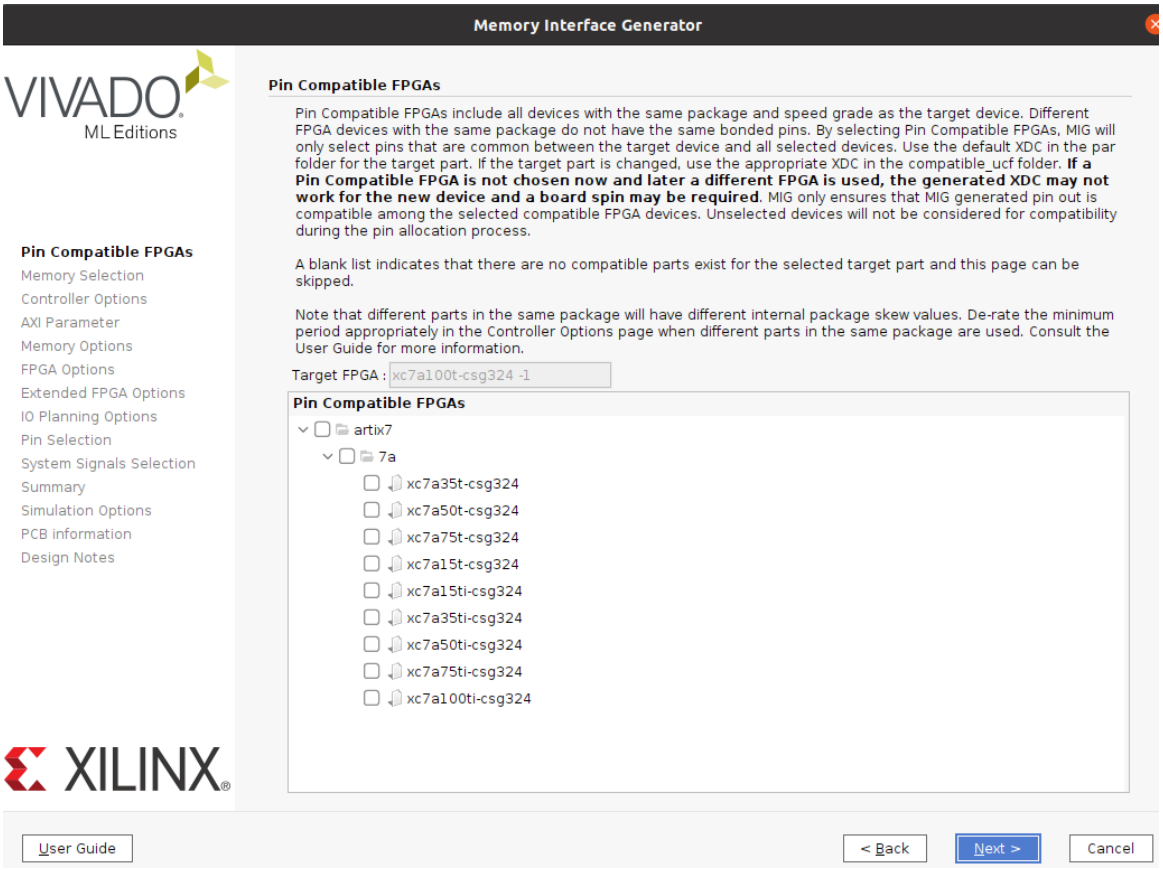


Fig. 2.3.2-2. Nexys A7 MIG generation steps 2,3.

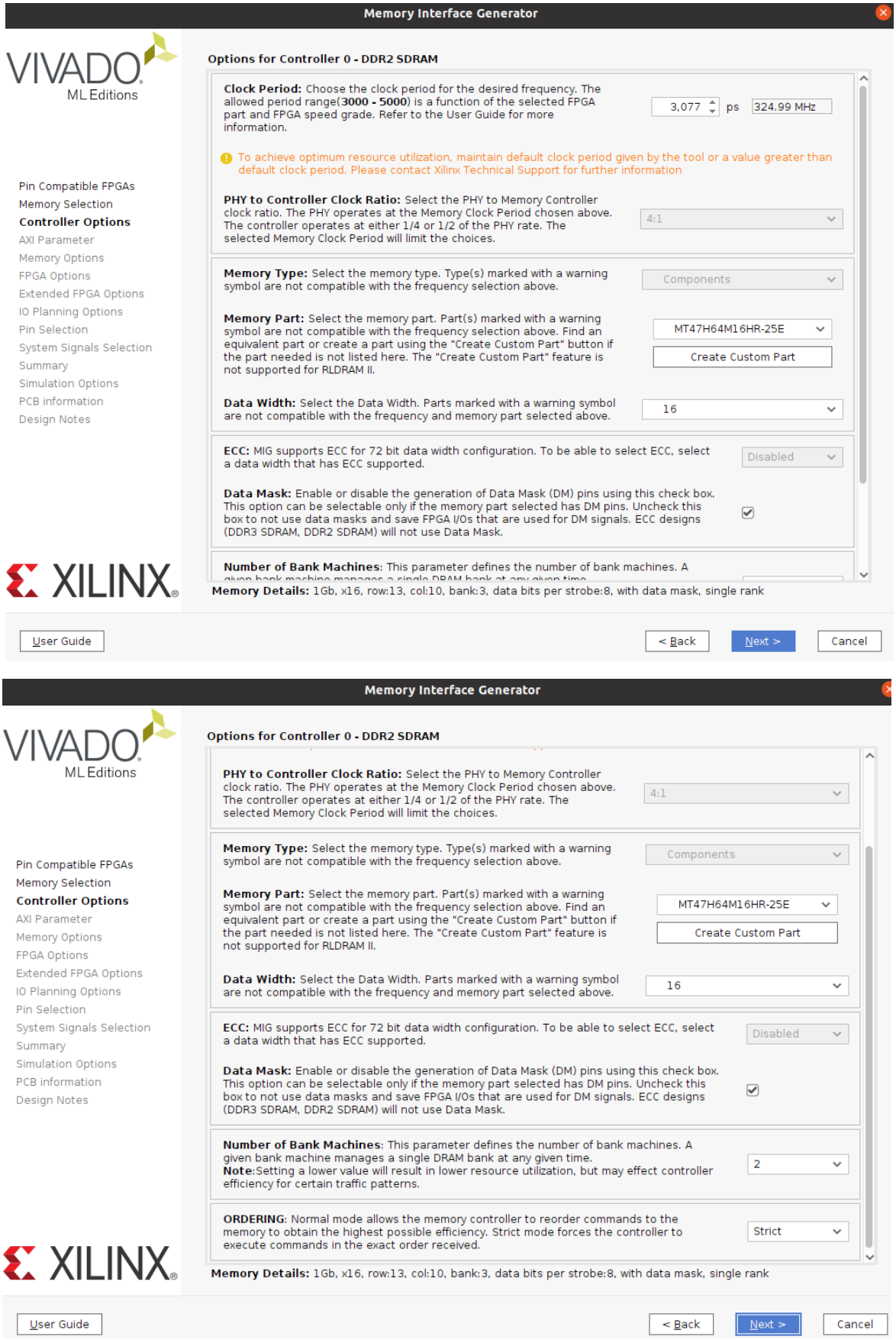


Fig. 2.3.2-3. Nexys A7 MIG generation step 4a and 4b.

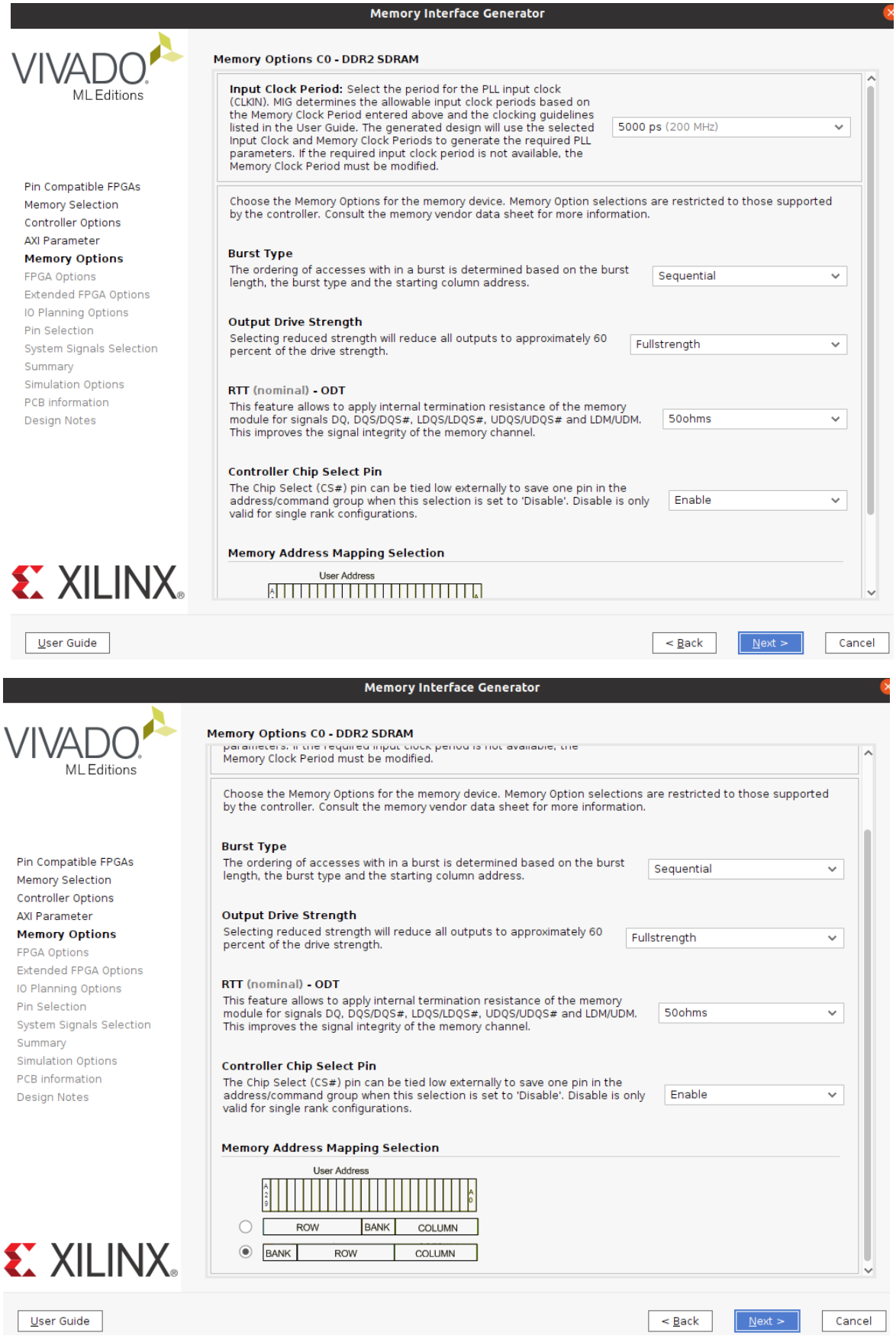
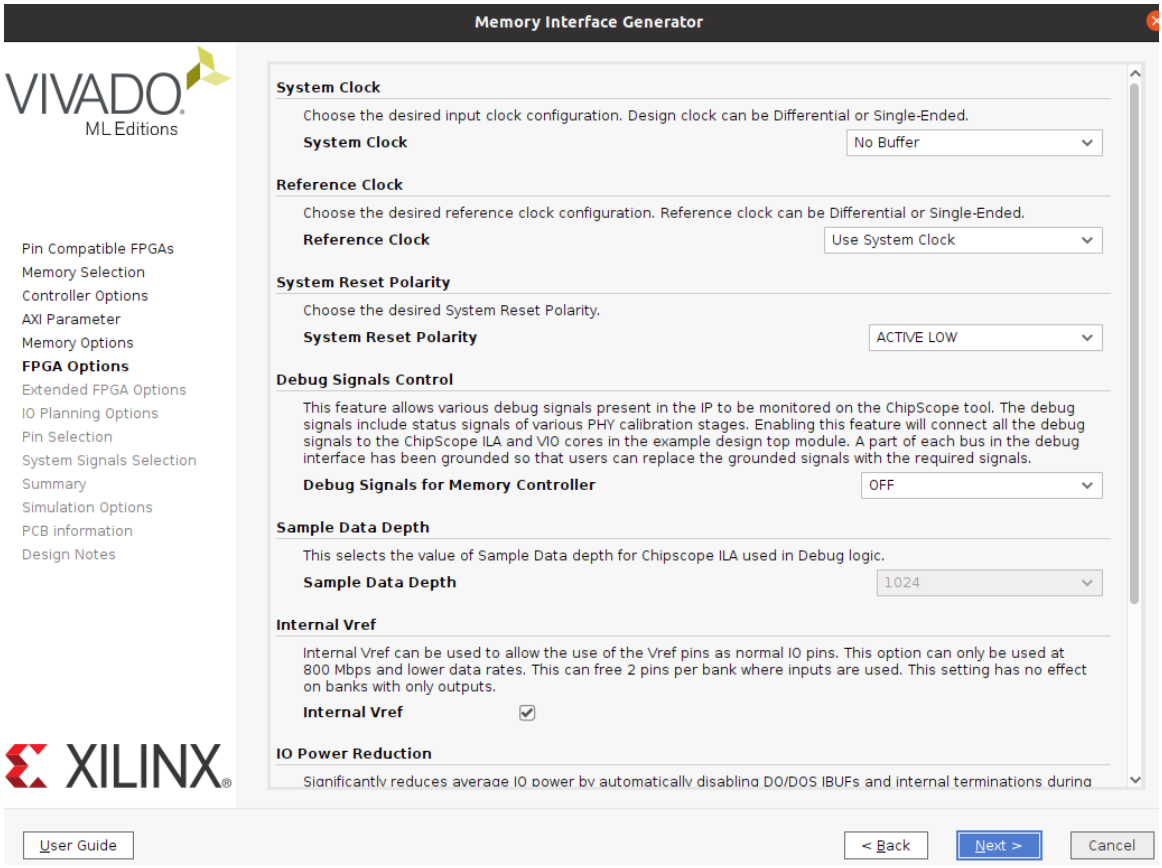


Fig. 2.3.2-4. MIG generation steps 5a and 5b.



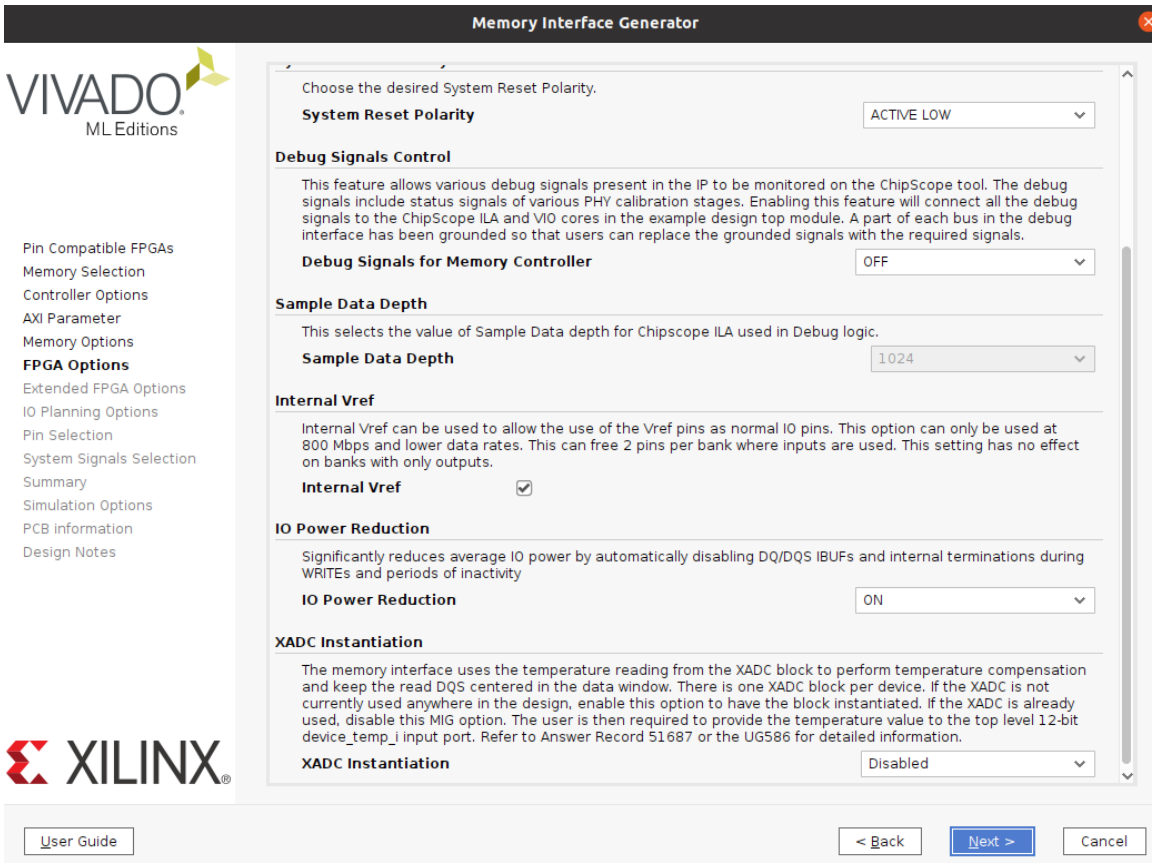
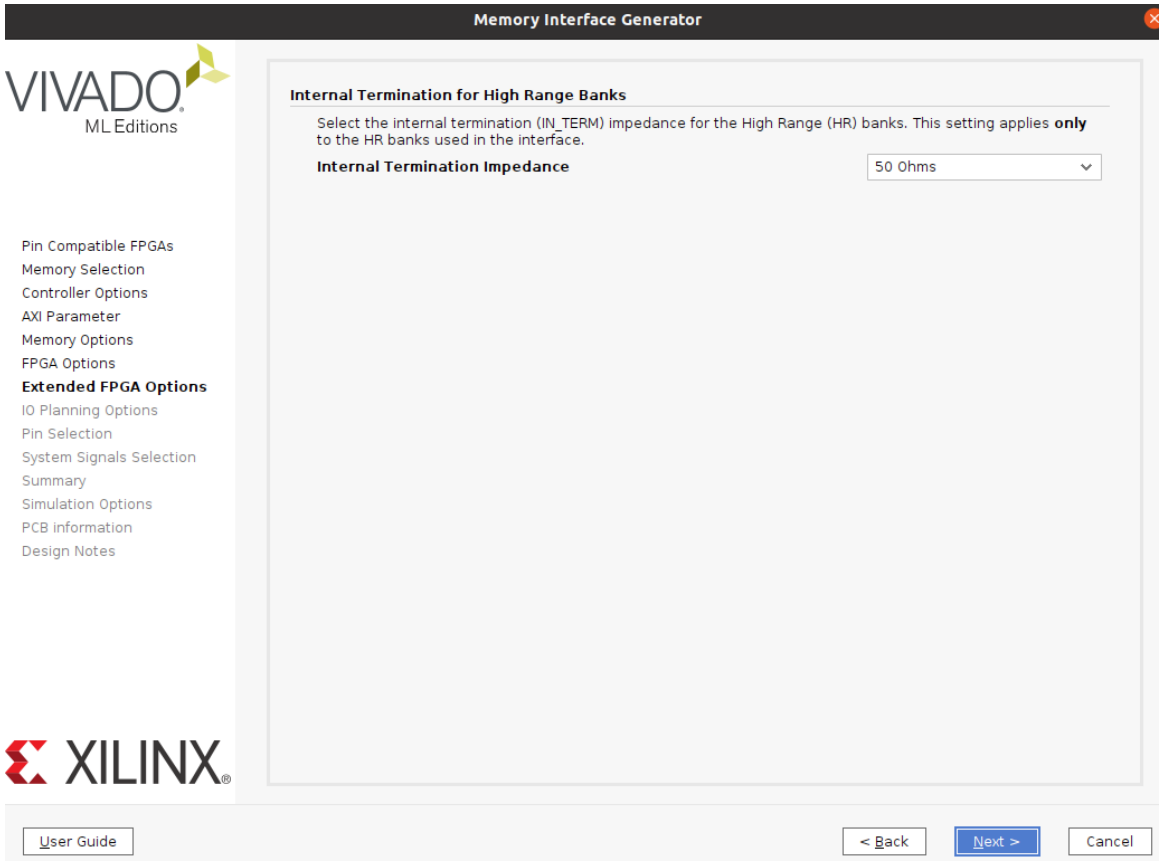


Fig. 2.3.2-5. MIG generation steps 6a and 6b.



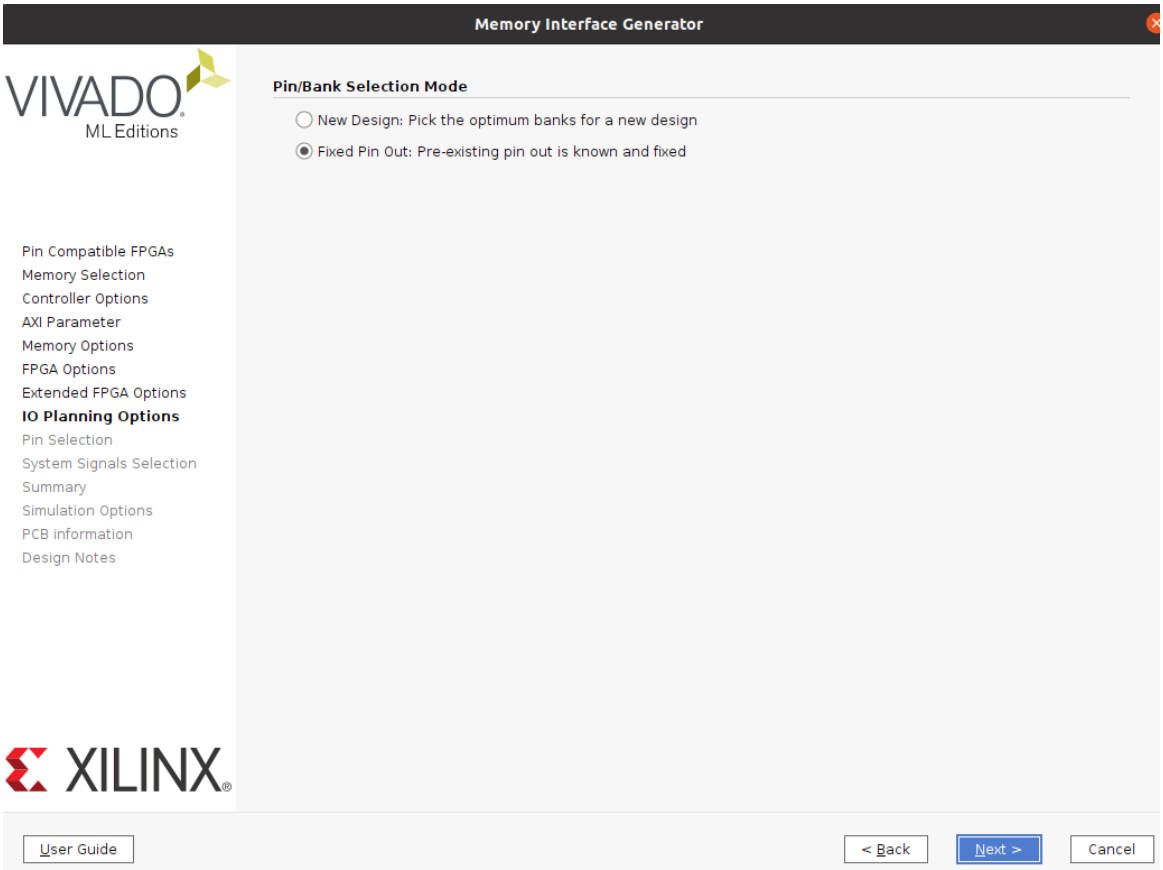


Fig. 2.3.2-6. MIG generation steps 7,8.

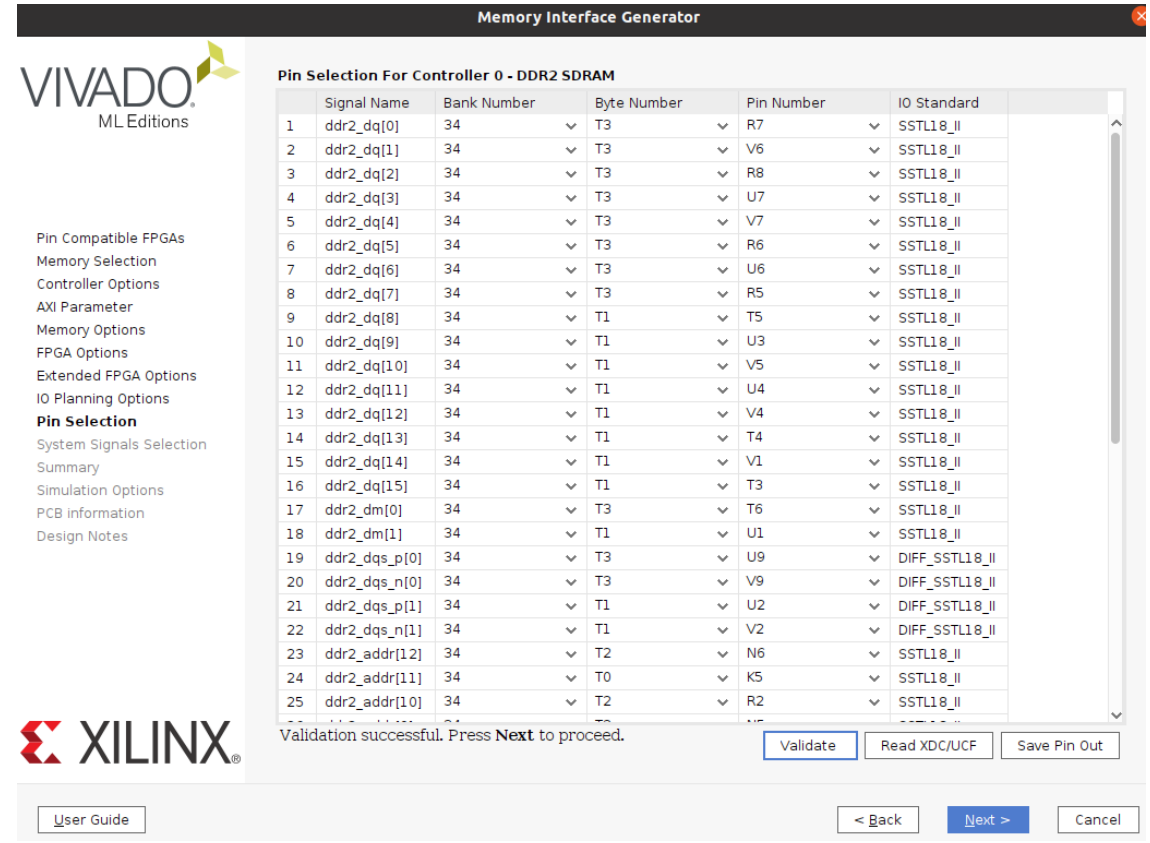


Fig. 2.3.2-7. MIG generation steps 9a and 9b.

Press **Read XDC/UCF** and select **nexys4.ucf** from **constrs** folder. Then click **Validate**.

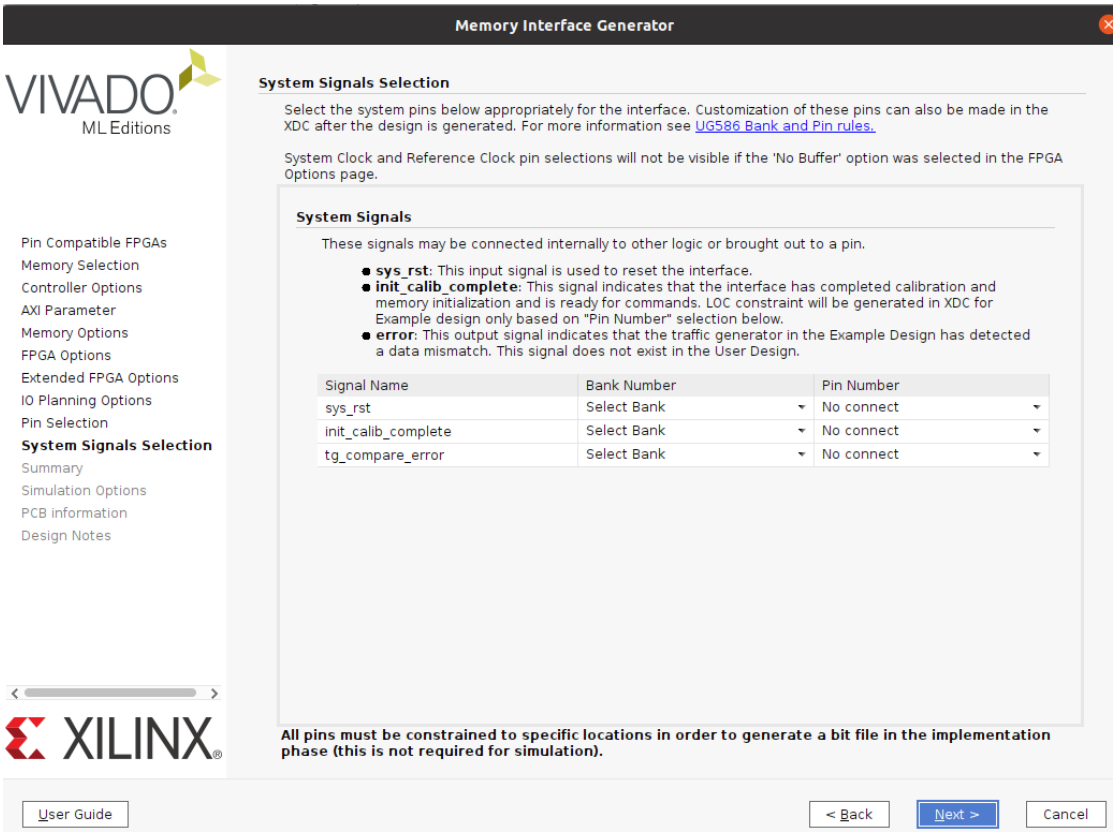
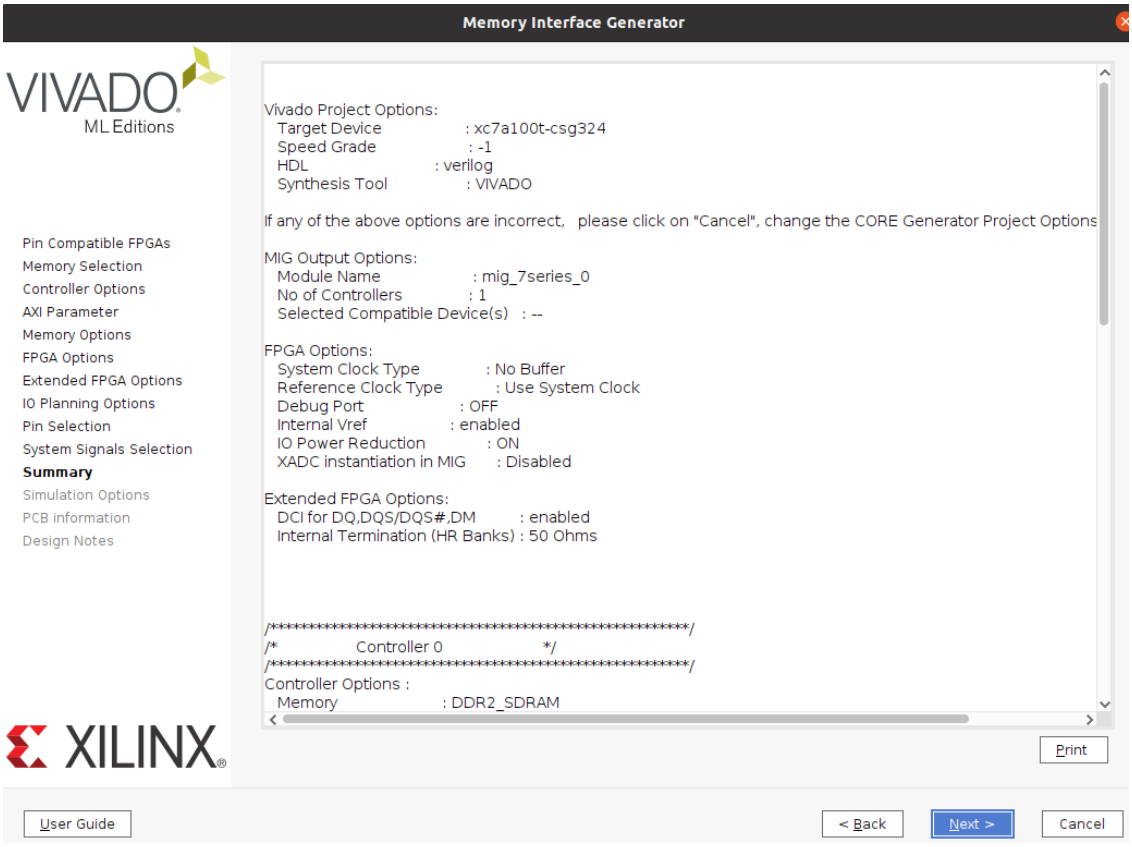


Fig. 2.3.2-8. MIG generation step 10.



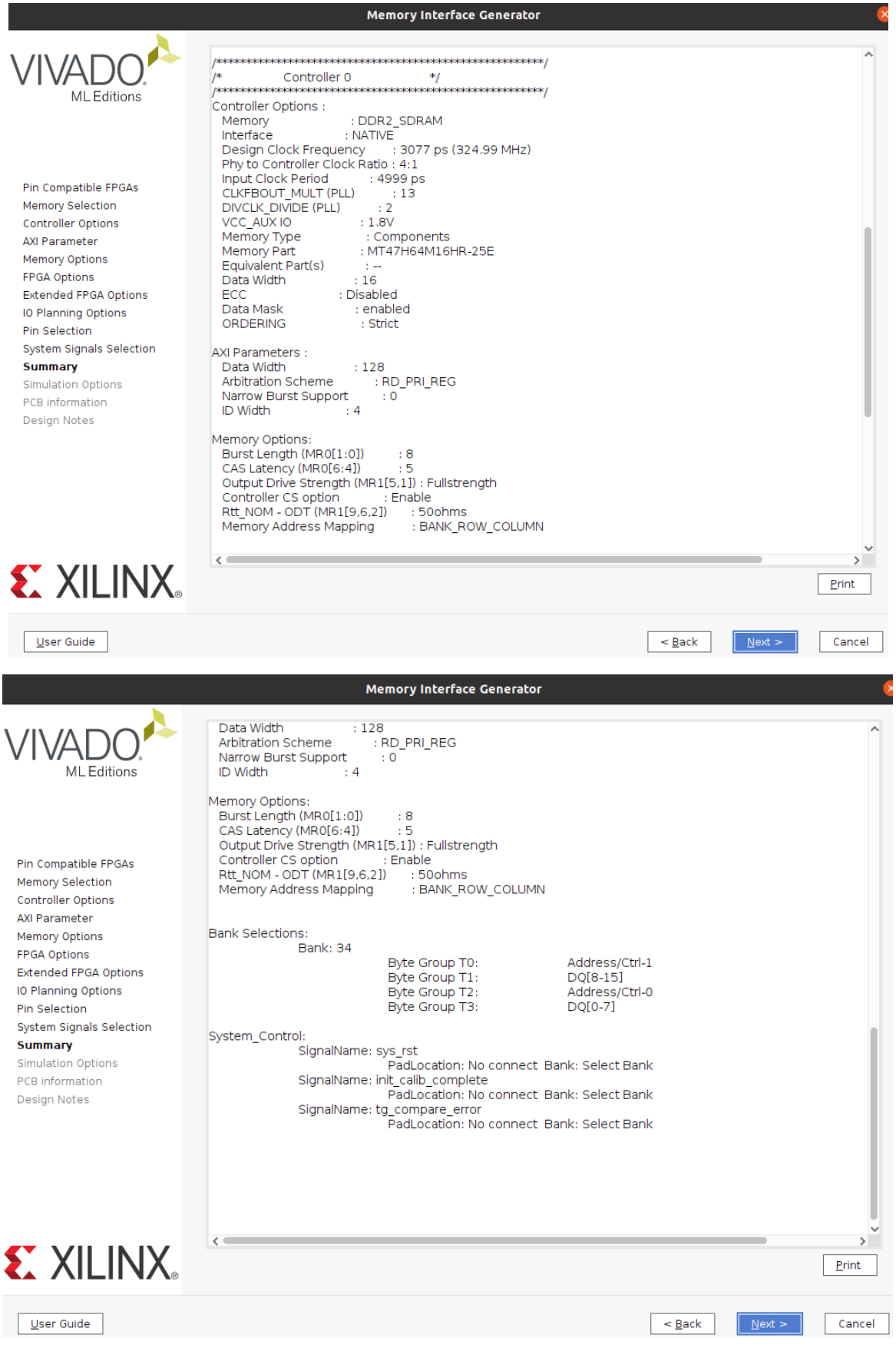


Fig. 2.3.2-9. MIG generation steps 11a, 11b and 11c.

Carefully verify that the settings specified in **Summary** are those that you have chosen in the intermediary steps. For the next windows click Next➔.. ➔Generate. Then select **Out of context per IP** in the **Generate Output Products** window, in order to avoid resynthesizing the IP each time you recompile the project. See Fig. 2.3.2-10.

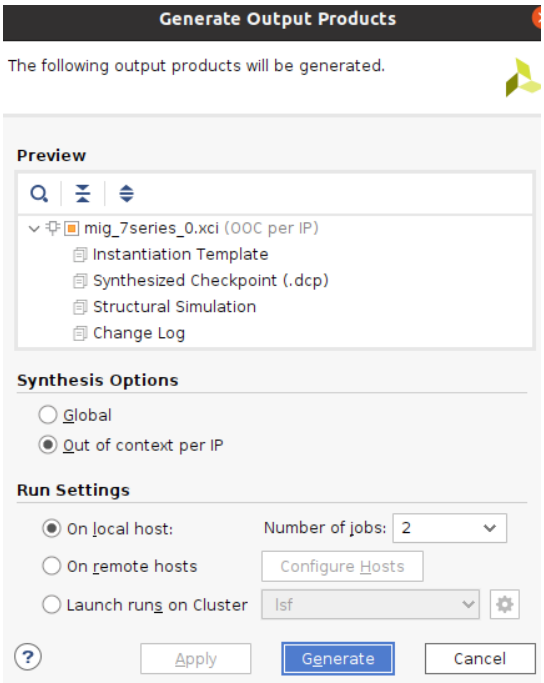


Fig. 2.3.2-10. Generate IP.

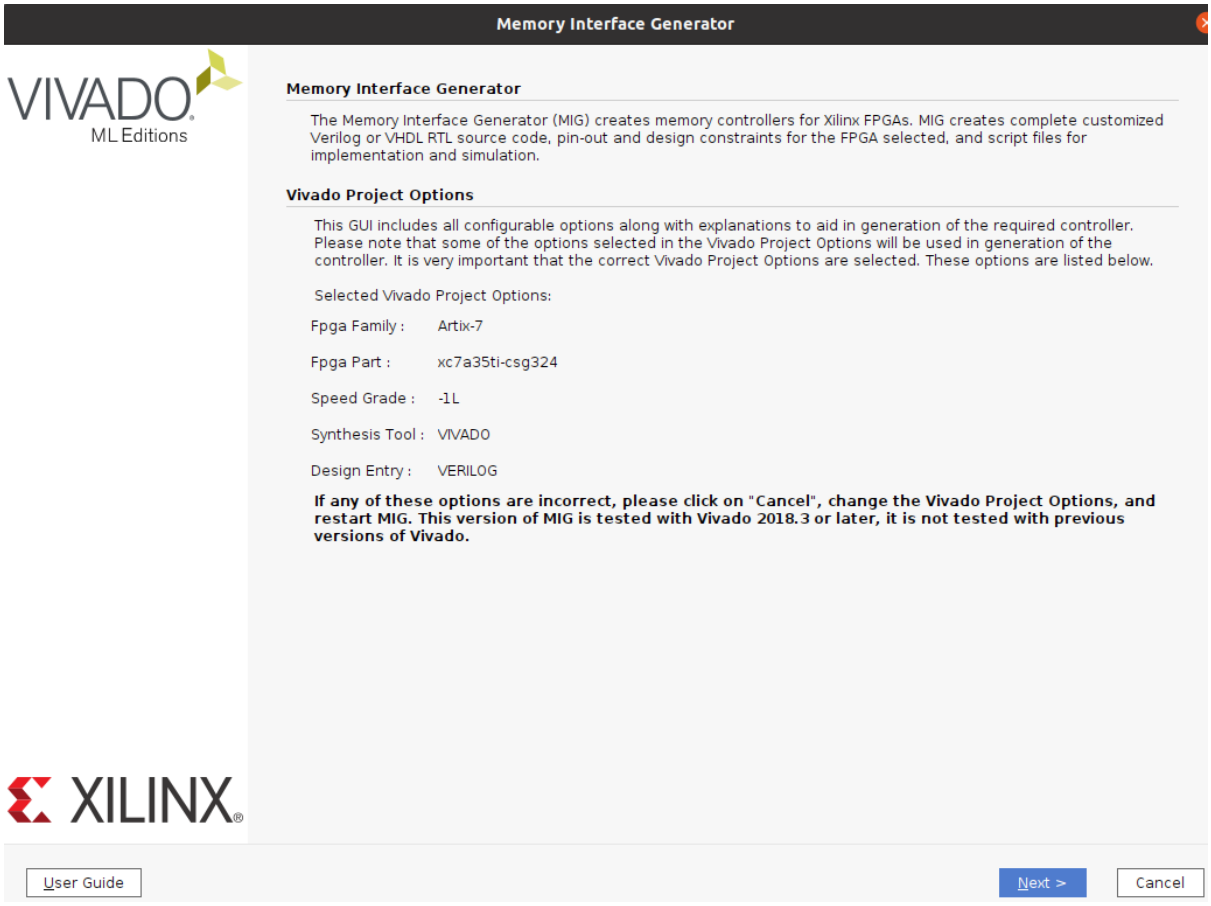
2.3.3 Create MIG for arty35t

The settings for the Xilinx MIG for arty35t are shown in its board reference manual [7]. They are presented in the following table.

Memory type	DDR3 SDRAM
Max. clock period	3000ps (667Mbps data rate)
Memory part	MT41K128M16JT-125
Memory Voltage	1.35V
Data width	16
Data mask	Enabled
Recommended Input Clock Period	6000 ps (166.667 MHz)
Output Driver Impedance Control	RZQ/6
Controller Chip Select pin	Enabled
Rtt (nominal) – On-die termination	RZQ/6
Internal Vref	Enabled
Internal termination impedance	50ohms

Table 2.3.3 Xilinx MIG settings for Arty A7

I now show the MIG generation step-by-step. Please read the notes from the previous subchapter.



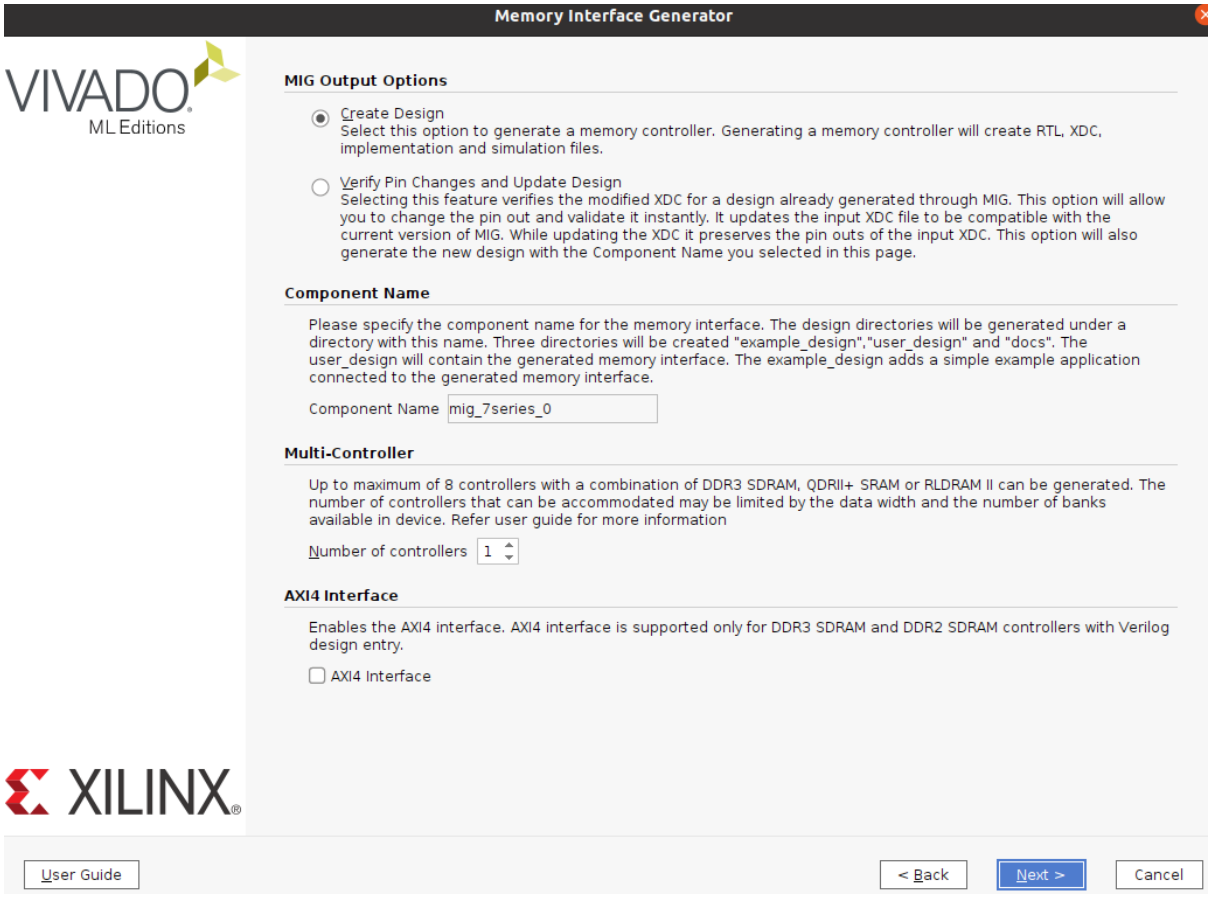


Fig. 2.3.3-1. MIG generation steps 1a and 1b.

RLSoC does not use the AXI interface; it uses the MIG user interface UI. For more info on MIG UI, see [7].

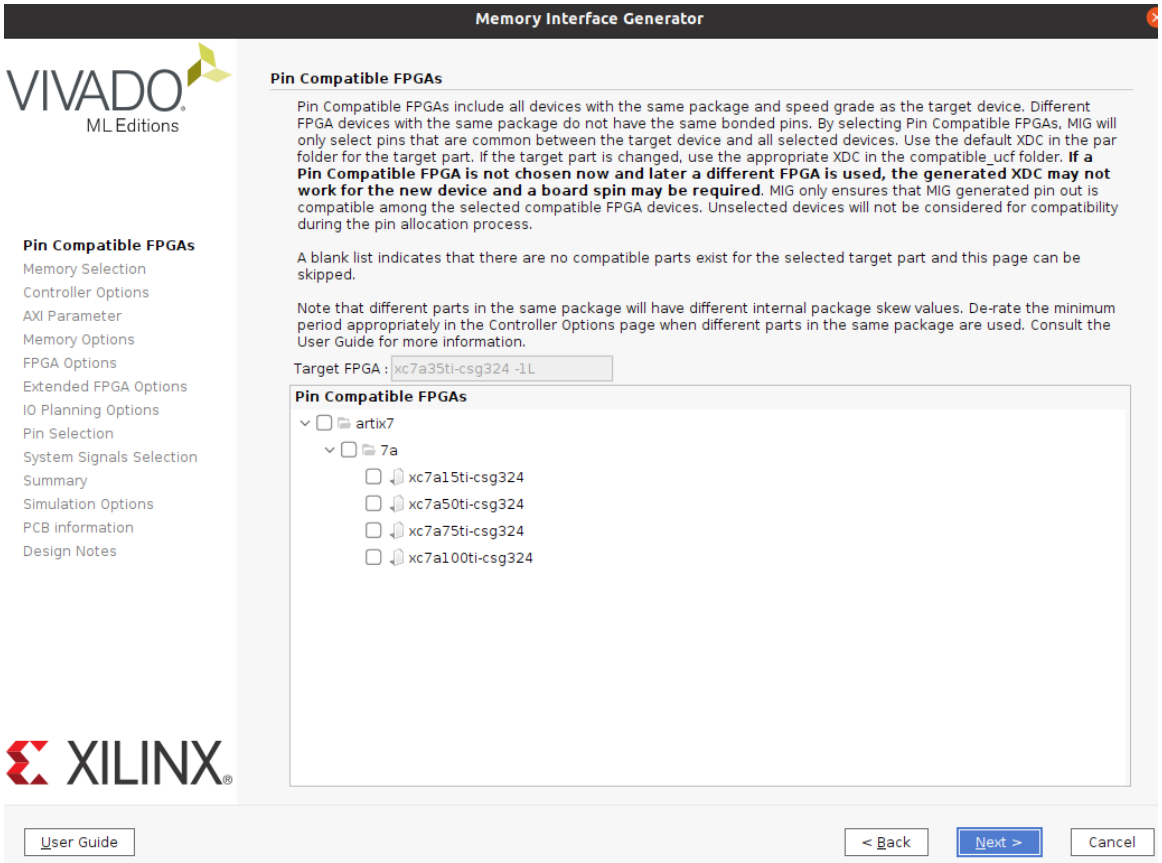
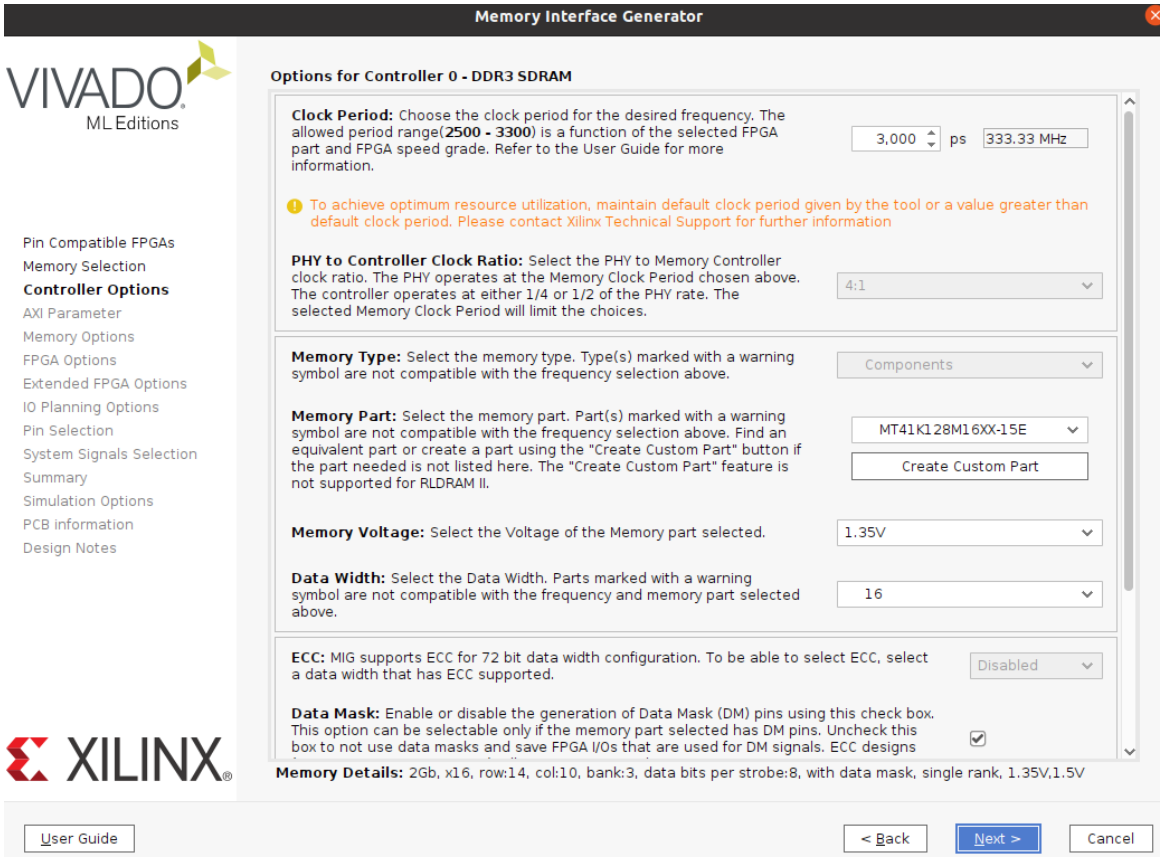


Fig. 2.3.3-2. MIG generation steps 2,3.



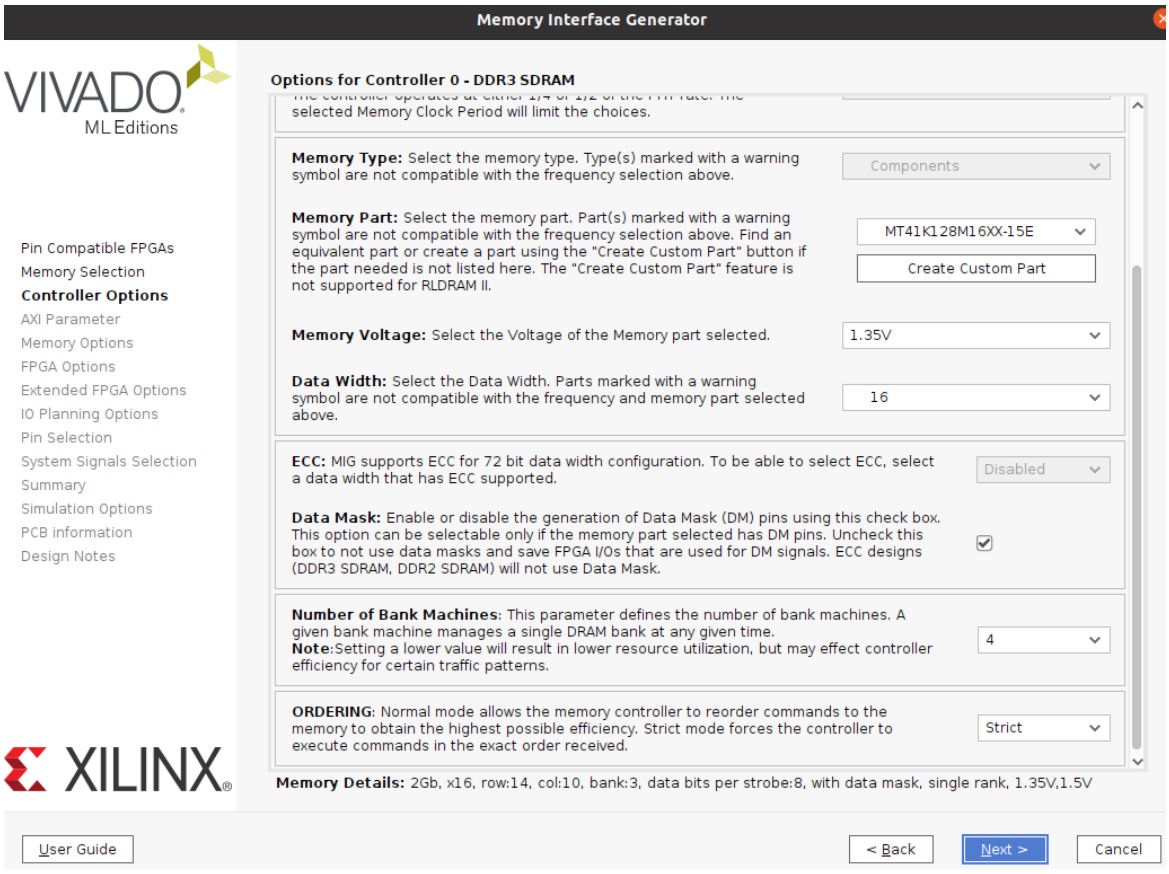
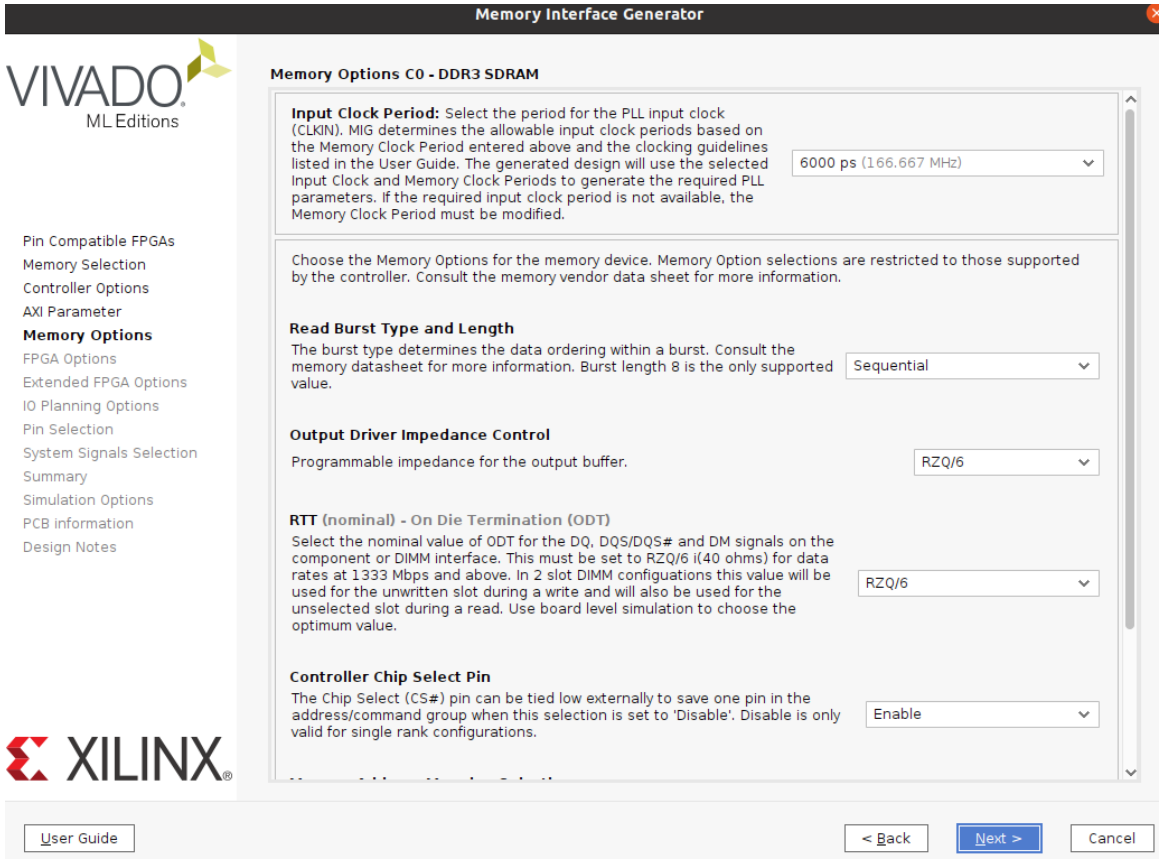


Fig. 2.3.3-3. MIG generation steps 4a and 4b.



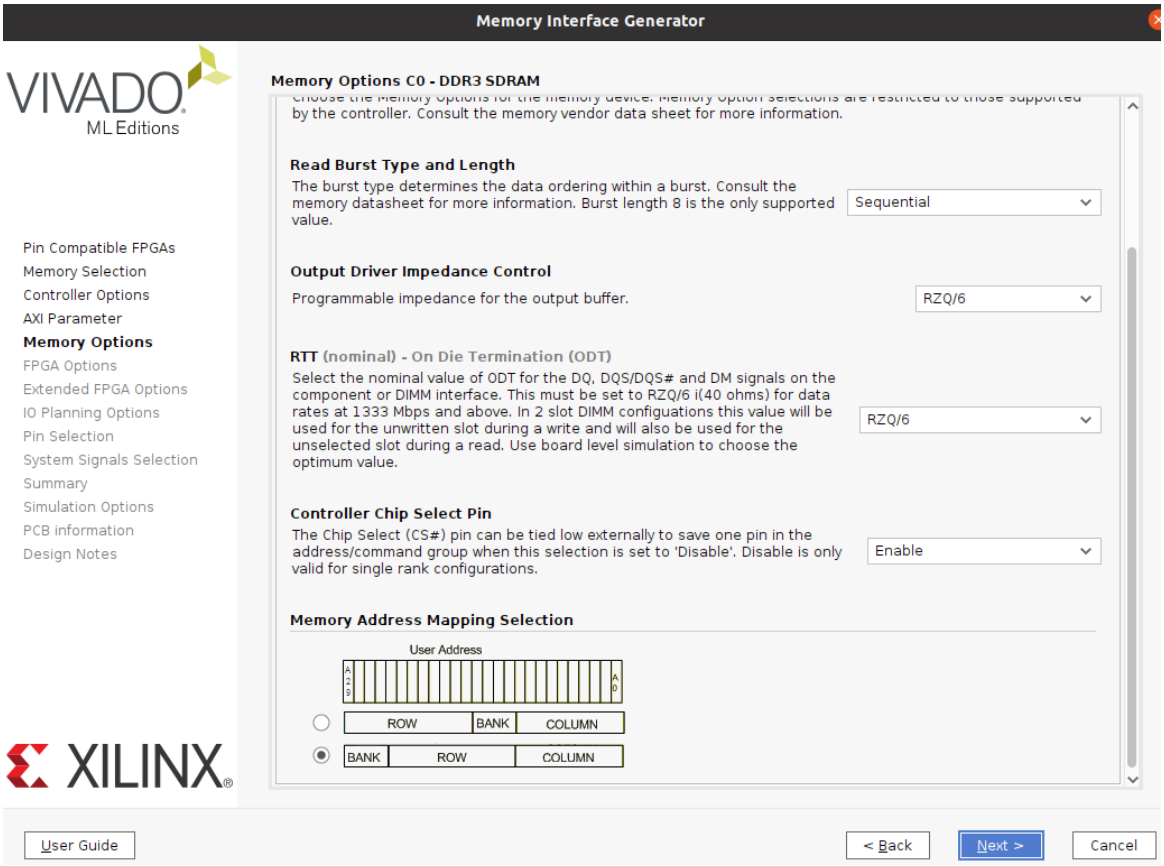


Fig. 2.3.3-4. MIG generation steps 5a and 5b.

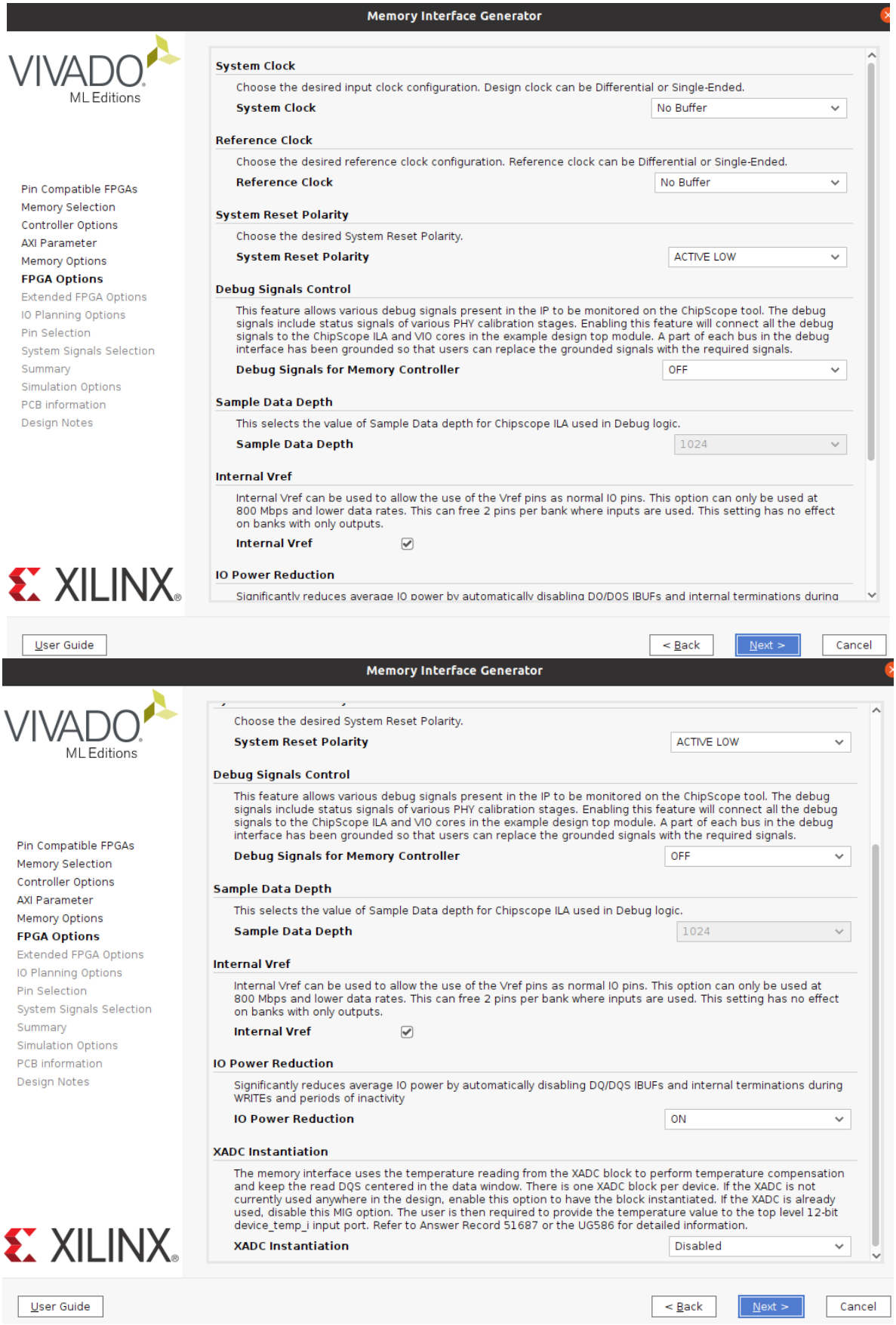


Fig. 2.3.3-5. MIG generation steps 6a and 6b.

Quoting from [7], “If the designs generated from MIG for the No Buffer option are implemented without performing changes, designs can fail in implementation due to IBUFs not instantiated for the ref_clk_i signal. So for No Buffer scenarios, ref_clk_i signal needs to be connected to an internal clock”. The reference clock is used for the IDELAY that calibrates the controller PHY. “An IDELAYCTRL is required in any bank that uses IDELAYs. IDELAYs are associated with the data group (DQ). Any bank/clock region that uses these signals require an IDELAYCTRL. The IDELAYCTRL reference frequency is set by the MIG tool to either 200 MHz, 300 MHz, or 400 MHz depending on memory interface frequency and speed grade of the FPGA” [7]. So, in the clock wizard clk_wiz_0 for arty35t, it was supplied a 200MHz clock to be used as reference clock by the MIG. Please see the clock wizard dedicated subchapters.

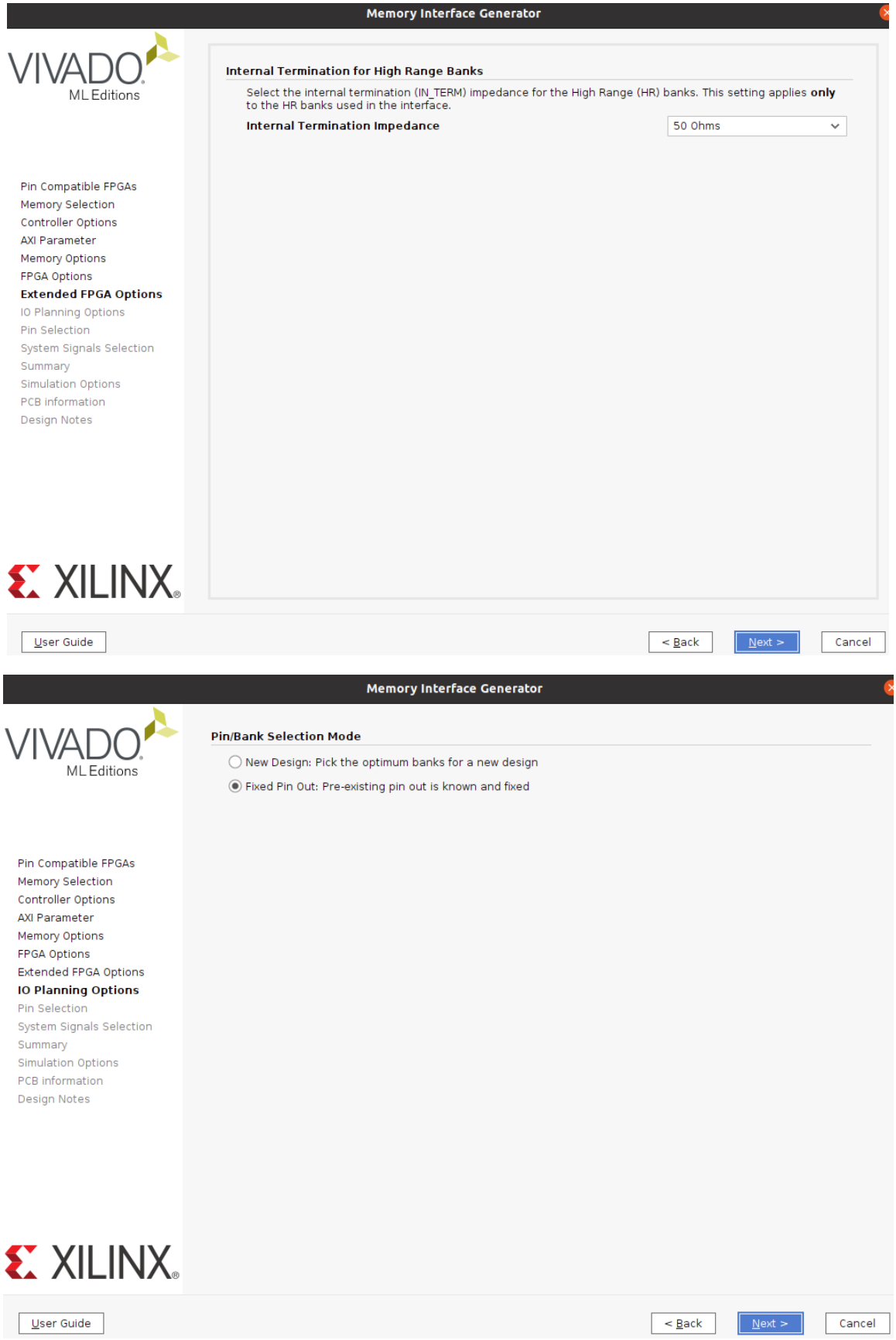
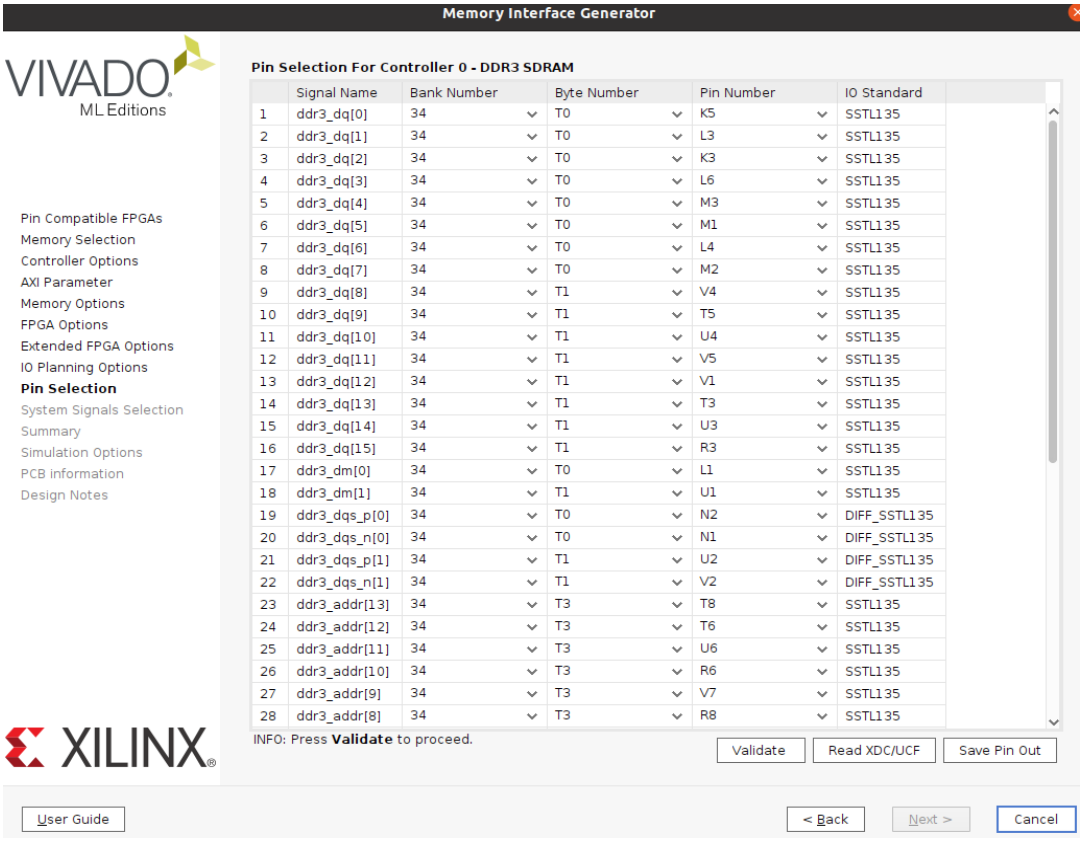


Fig. 2.3.3-6. MIG generation steps 7,8.



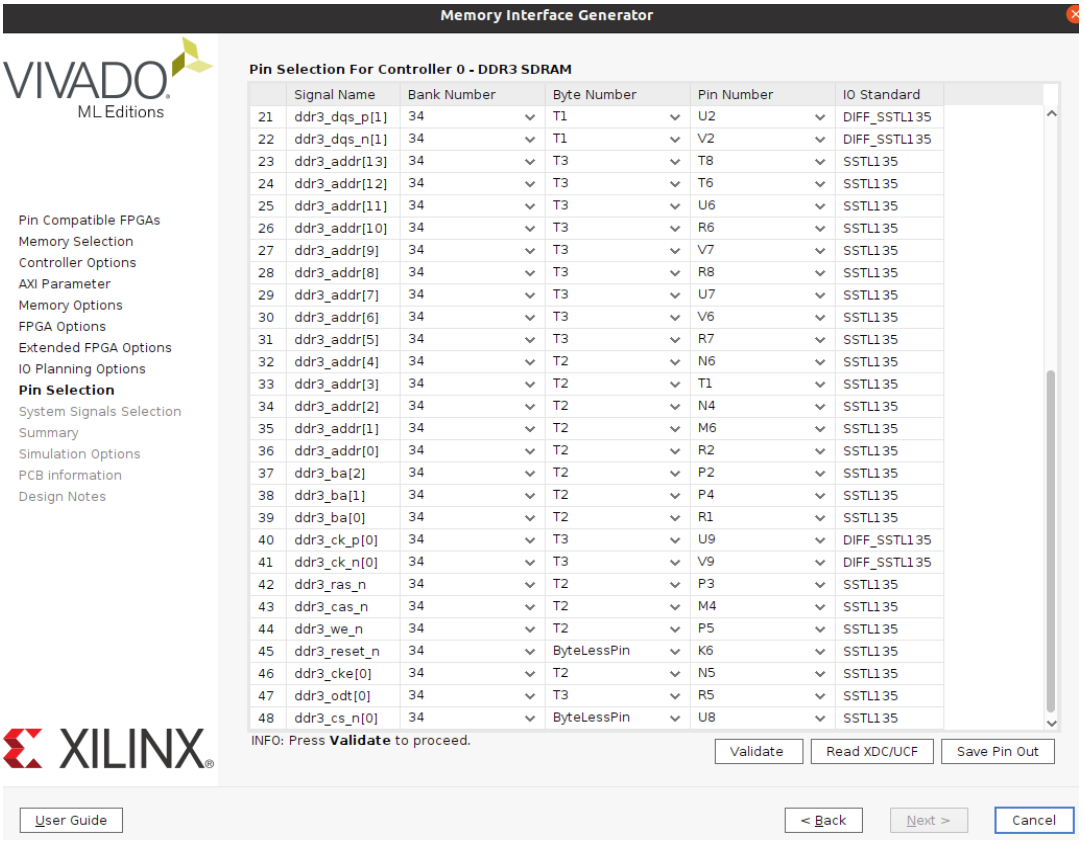


Fig. 2.3.3-7. MIG generation steps 9a and 9b.

Press **Read XDC/UCF** and select **arty35t.ucf** from **constrs** folder. Then click **Validate**.

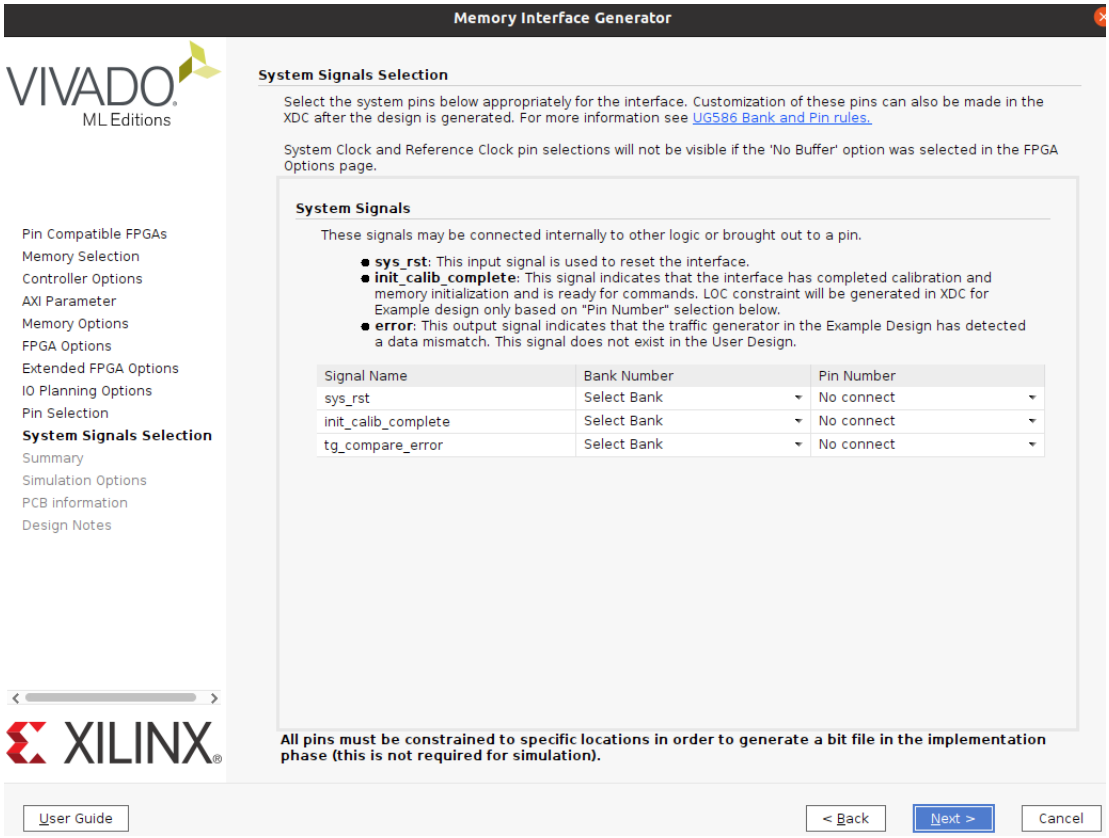


Fig. 2.3.3-8. MIG generation step 10.

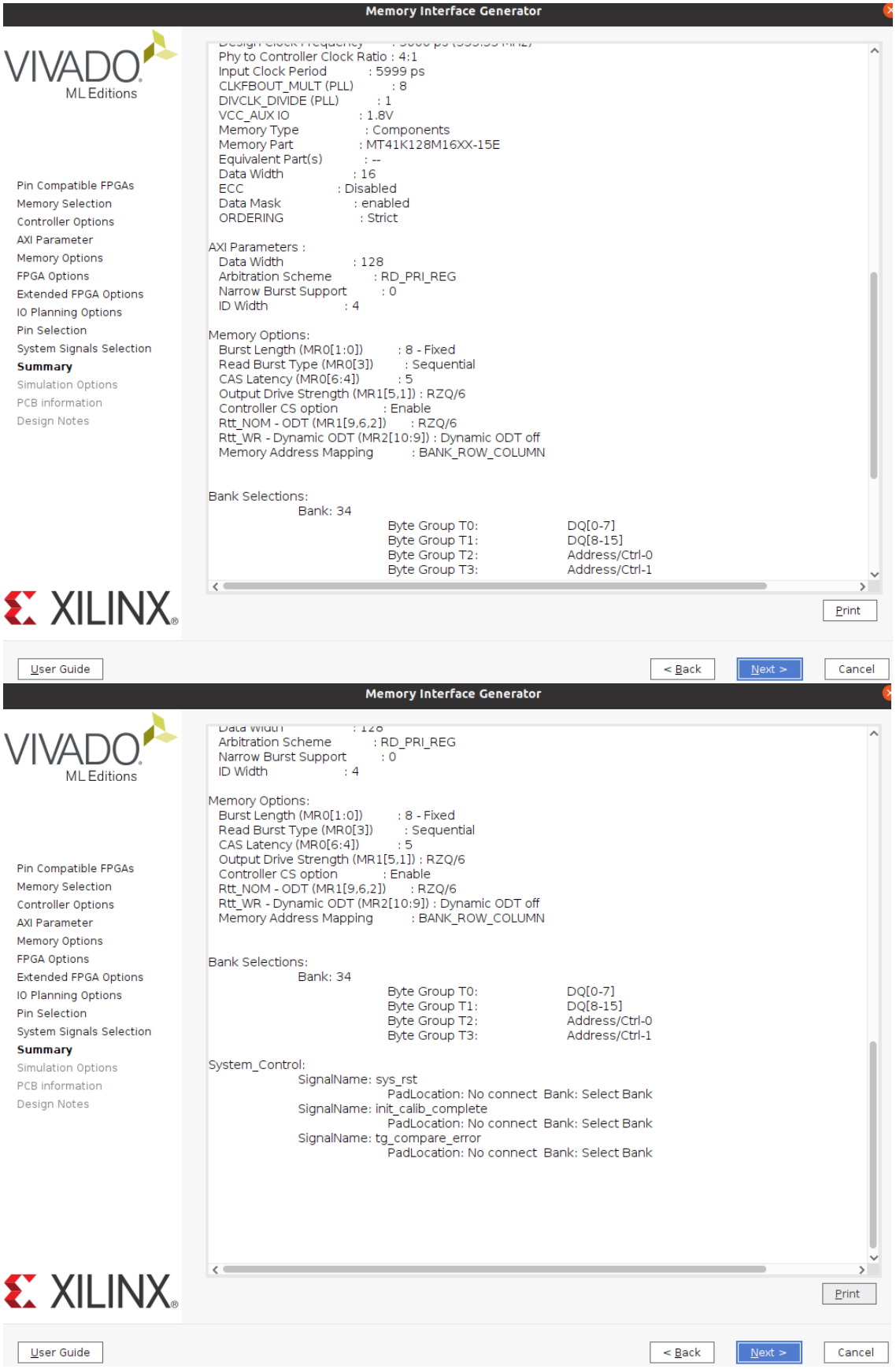


Fig. 2.3.3-9. MIG generation steps 11a, 11b, 11c.

Carefully verify that the settings specified in **Summary** are those that you have chosen in the intermediary steps. For the next windows click Next➔.. ➔Generate. Then select **Out of context per IP** in the **Generate Output Products** window**,** in order to avoid resynthesizing the IP each time you recompile the project. See Fig. 2.3.2-10.

2.3.4 Clocks

Nexys A7 and Arty A7 have 100 MHz board clock which is tied to the **CLK** variable in **main.v**. The module **clk_wiz_0** takes as input **CLK** and outputs the **mig_clk** clock used by the MIG (and also ref_clk used by the arty32t MIG as we have seen in Fig. 2.3.3-5). The module **clk_wiz_1** takes the **mig_ui_clk** (MIG output clock for its user interface) as input and outputs **clk** (an 104 MHz clock) which is used by the other modules as their clock signal. The corresponding code lines are shown in the next listing. Please note that the **locked** signal indicates that MCMM is generating a stable and reliable clock.

```
main.v
// nexys
clk_wiz_0 m_clkgen0 (.clk_in1(CLK), .resetn(RST_X_IN),
                     .clk_out1(mig_clk), .locked(w_locked));
// arty
clk_wiz_0 m_clkgen0 (.clk_in1(CLK), .resetn(RST_X_IN),
                      .clk_out1(mig_clk), .clk_out2(ref_clk), .locked(w_locked));
dram.v
clk_wiz_1 clkgen1 (
          .clk_in1(mig_ui_clk), .resetn(mig_ui_rst_x),
          .clk_out1(clk), .locked(locked));
```

Listing 2.3.4-1. Clock generator modules instantiations.

clk_wiz_0 and **clk_wiz_1** are created using “Clocking Wizard” in IP catalog.

2.3.5 clk_wiz_0 for Nexys A7

I now show the step by step **clk_wiz_0** generation for Nexys A7.

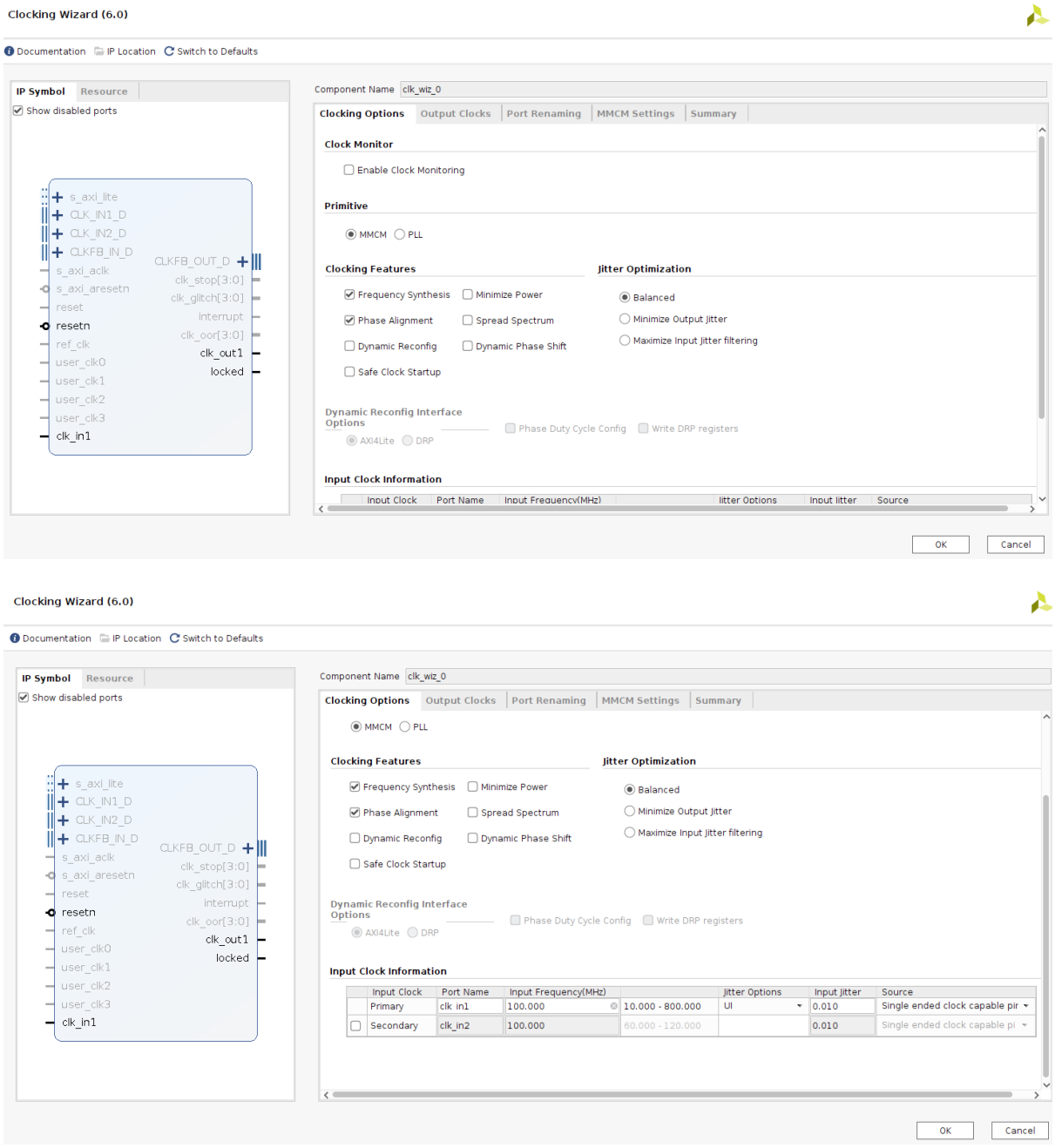
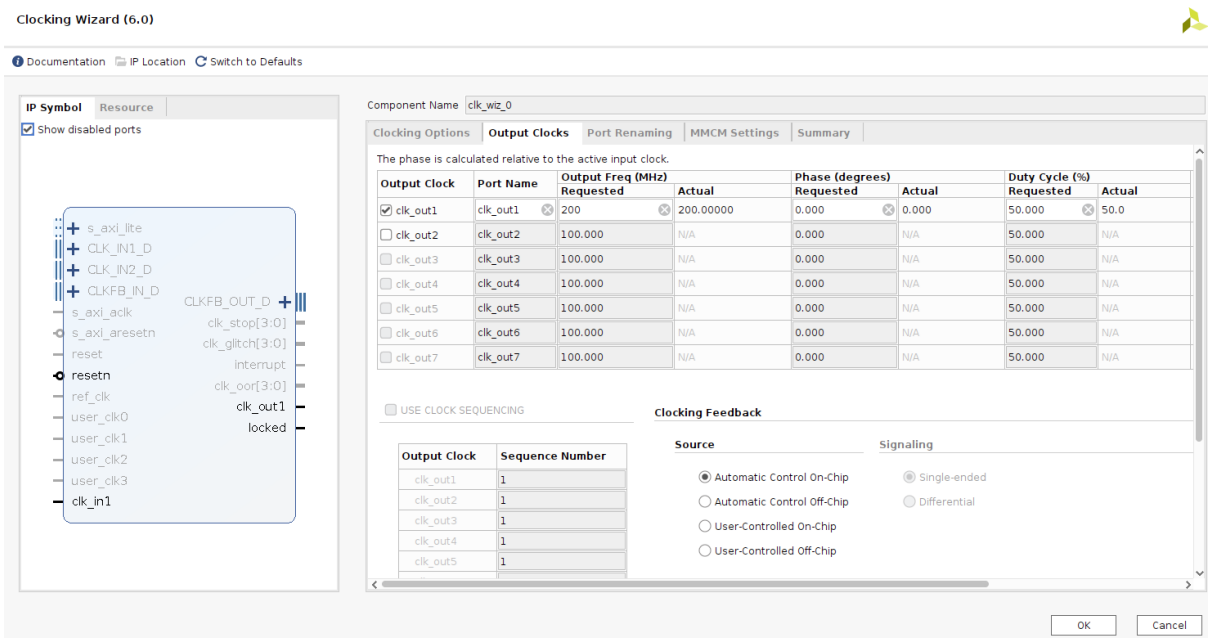
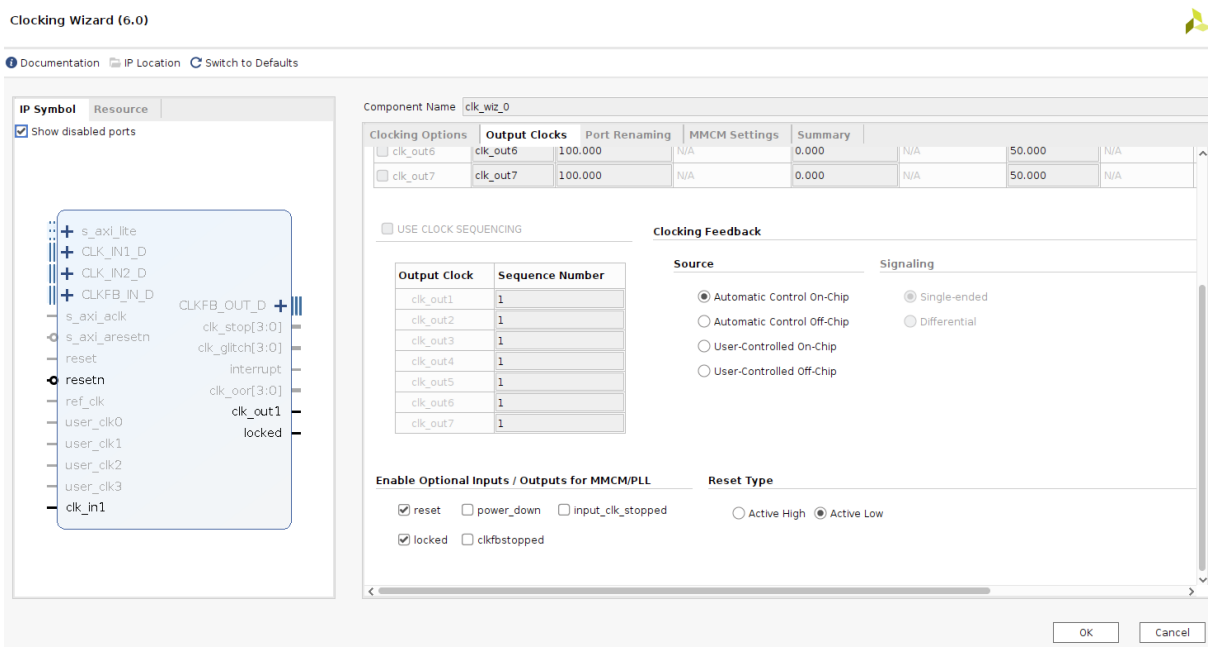


Fig. 2.3.5-1. Nexys A7 clk_wiz_0 generation, steps 1a and 1b.


Clocking Wizard (6.0)
Documentation
IP Location
Switch to Defaults
IP Symbol
Resource
Show disabled ports
s_axi_lite
CLK_IN1_D
CLK_IN2_D
CLKFB_IN_D
s_axi_aclk
s_axi_aresetn
reset
resetn
ref_clk
user_clk0
user_clk1
user_clk2
user_clk3
clk_in1
CLKFB_OUT_D
clk_stop[3:0]
clk_glitch[3:0]
interrupt
clk_oor[3:0]
clk_out1
locked
Component Name
clk_wiz_0
Clocking Options
Output Clocks
Port Renaming
MMCM Settings
Summary
The phase is calculated relative to the active input clock.
Output Clock
Port Name
Output Freq (MHz)
Requested
Actual
Phase (degrees)
Requested
Actual
Duty Cycle (%)
Requested
Actual
clk_out1
clk_out1
200
200.00000
0.000
0.000
50.000
50.0
clk_out2
clk_out2
100.000
N/A
0.000
N/A
50.000
N/A
clk_out3
clk_out3
100.000
N/A
0.000
N/A
50.000
N/A
clk_out4
clk_out4
100.000
N/A
0.000
N/A
50.000
N/A
clk_out5
clk_out5
100.000
N/A
0.000
N/A
50.000
N/A
clk_out6
clk_out6
100.000
N/A
0.000
N/A
50.000
N/A
clk_out7
clk_out7
100.000
N/A
0.000
N/A
50.000
N/A
USE CLOCK SEQUENCING
Output Clock
Sequence Number
clk_out1
1
clk_out2
1
clk_out3
1
clk_out4
1
clk_out5
1
Clocking Feedback
Source
Signaling
Automatic Control On-Chip
Automatic Control Off-Chip
User-Controlled On-Chip
User-Controlled Off-Chip
Single-ended
Differential
OK
Cancel


Clocking Wizard (6.0)
Documentation
IP Location
Switch to Defaults
IP Symbol
Resource
Show disabled ports
s_axi_lite
CLK_IN1_D
CLK_IN2_D
CLKFB_IN_D
s_axi_aclk
s_axi_aresetn
reset
resetn
ref_clk
user_clk0
user_clk1
user_clk2
user_clk3
clk_in1
CLKFB_OUT_D
clk_stop[3:0]
clk_glitch[3:0]
interrupt
clk_oor[3:0]
clk_out1
locked
Component Name
clk_wiz_0
Clocking Options
Output Clocks
Port Renaming
MMCM Settings
Summary
clk_out6
clk_out6
100.000
N/A
0.000
N/A
50.000
N/A
clk_out7
clk_out7
100.000
N/A
0.000
N/A
50.000
N/A
USE CLOCK SEQUENCING
Output Clock
Sequence Number
clk_out1
1
clk_out2
1
clk_out3
1
clk_out4
1
clk_out5
1
clk_out6
1
clk_out7
1
Clocking Feedback
Source
Signaling
Automatic Control On-Chip
Automatic Control Off-Chip
User-Controlled On-Chip
User-Controlled Off-Chip
Single-ended
Differential
Enable Optional Inputs / Outputs for MMCM/PLL
reset
power_down
input_clk_stopped
locked
clkfbstopped
Reset Type
Active High
Active Low
OK
Cancel

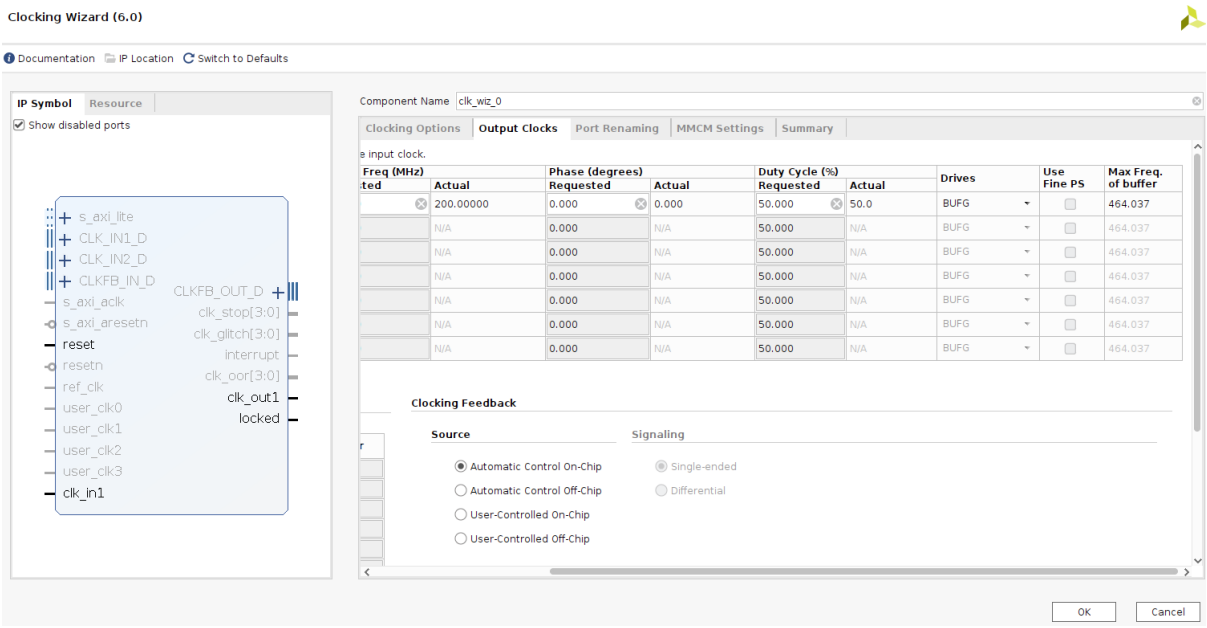


Fig. 2.3.5-2. Nexys A7 clk_wiz_0 generation, steps 2a, 2b and 2c.

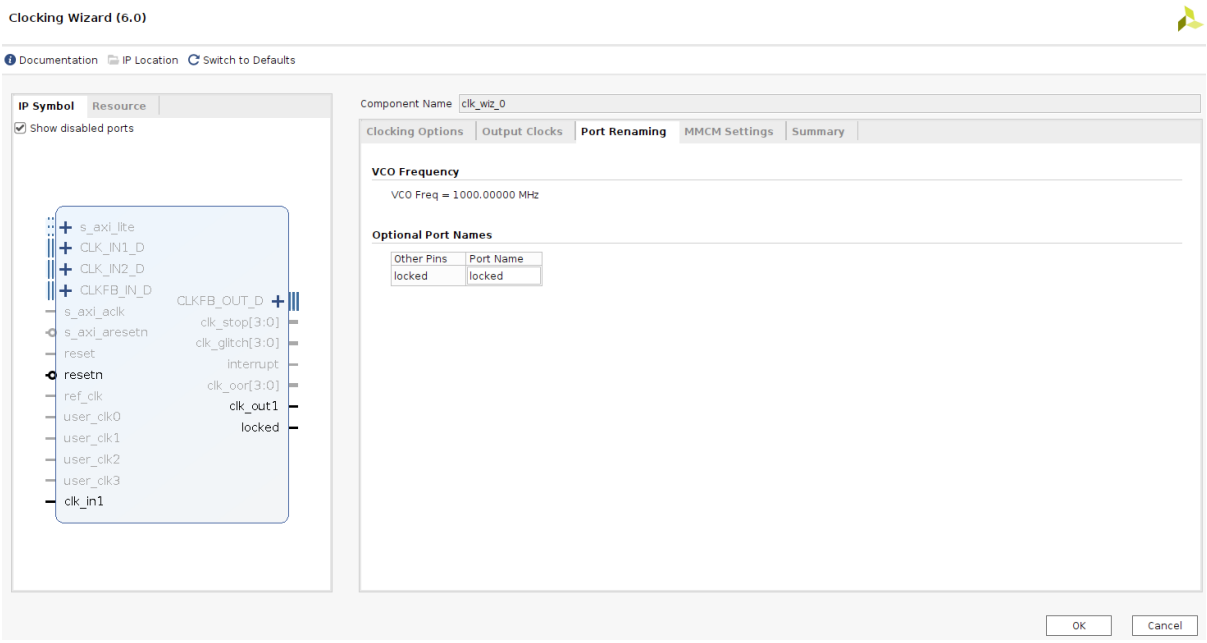


Fig. 2.3.5-3. Nexys A7 clk_wiz_0 generation, step 3.

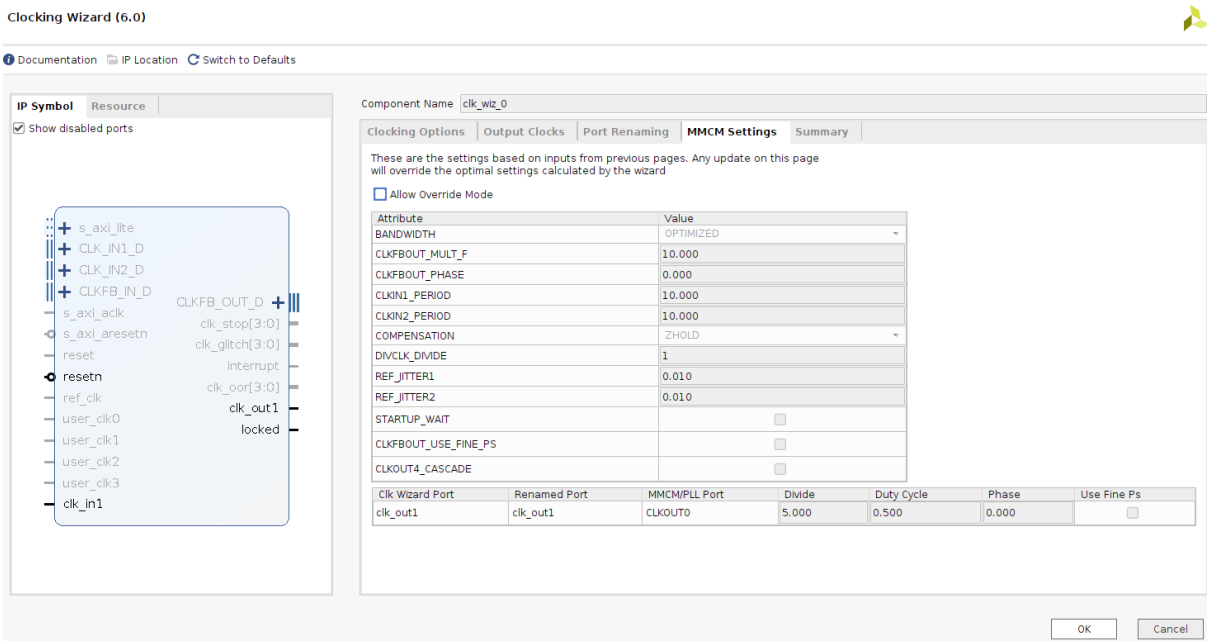


Fig. 2.3.5-4. Nexys A7 clk_wiz_0 generation, step 4.

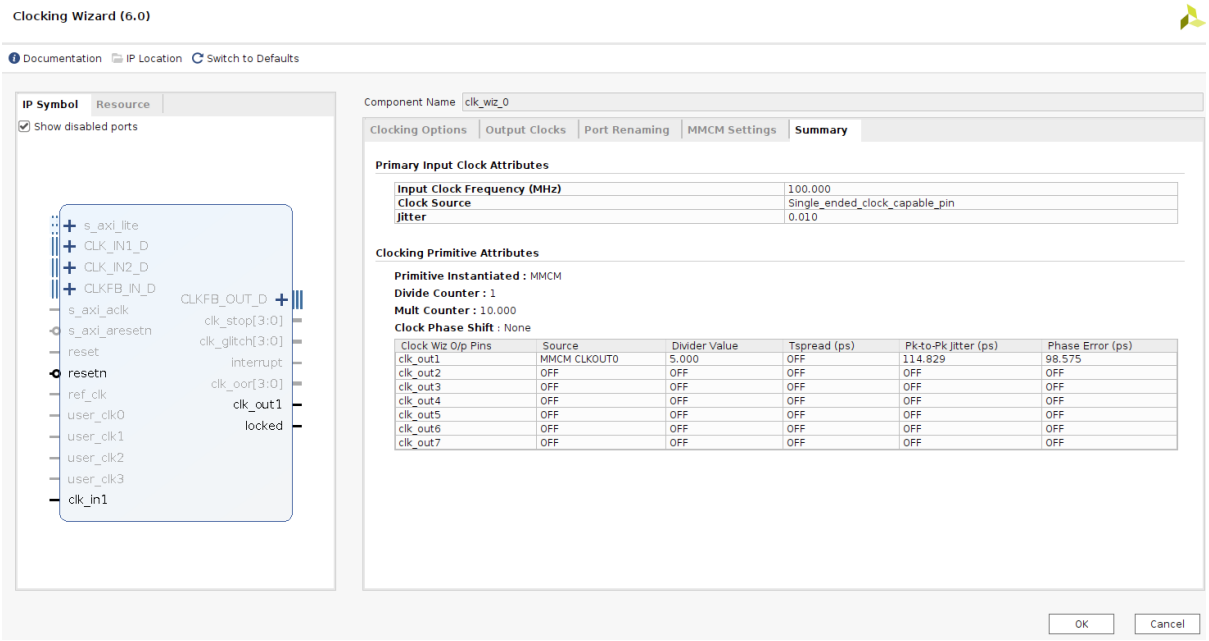


Fig. 2.3.5-5. Nexys A7 clk_wiz_0 generation, step 5.

2.3.6 clk_wiz_0 for Arty A7

I now show the step by step **clk_wiz_0** generation for Arty A7.

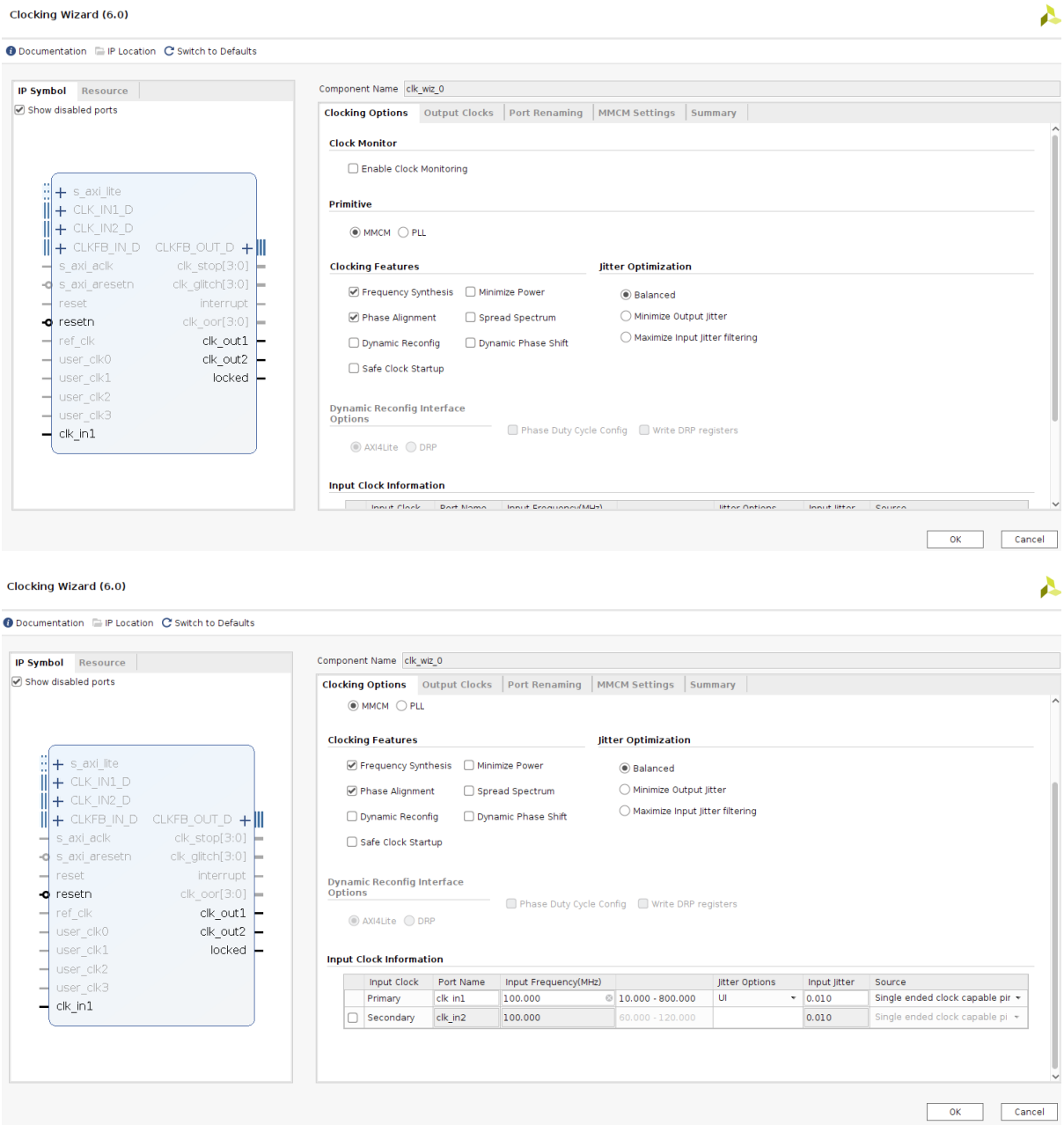
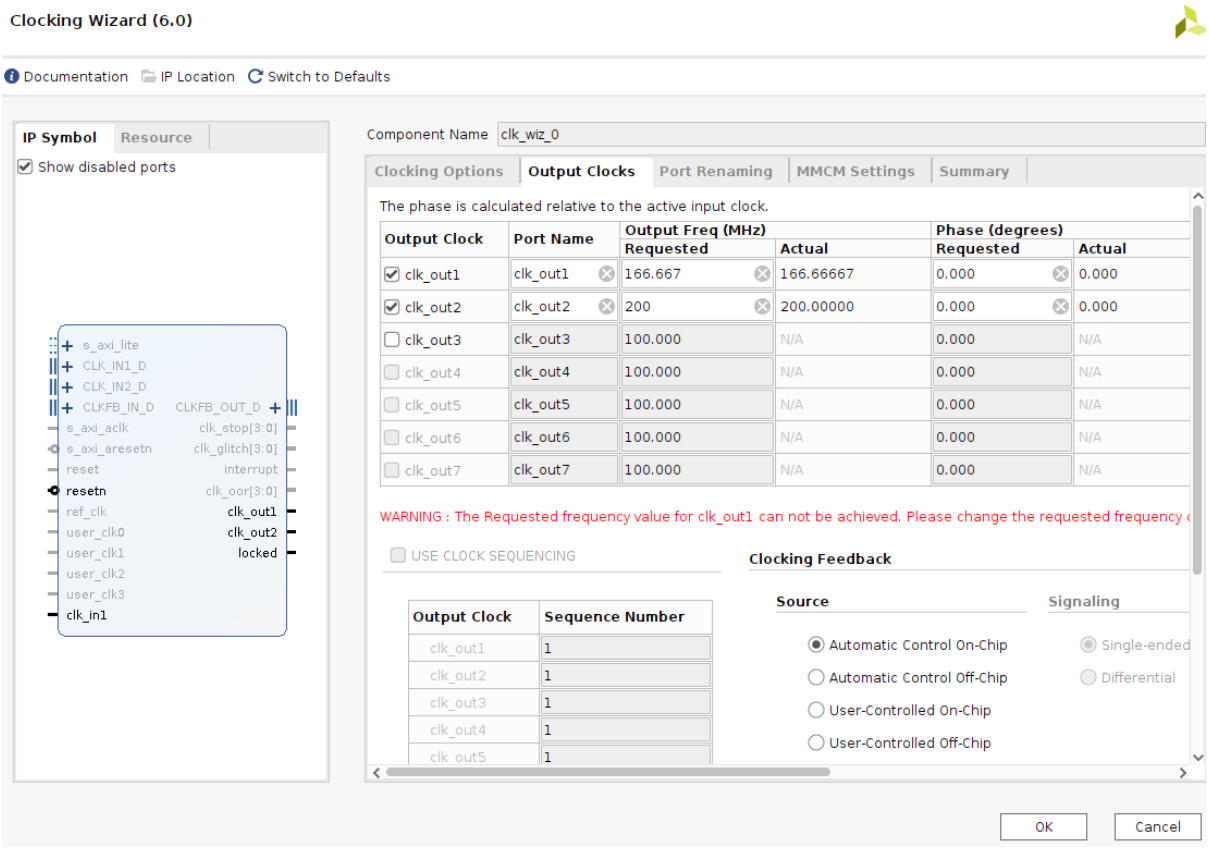
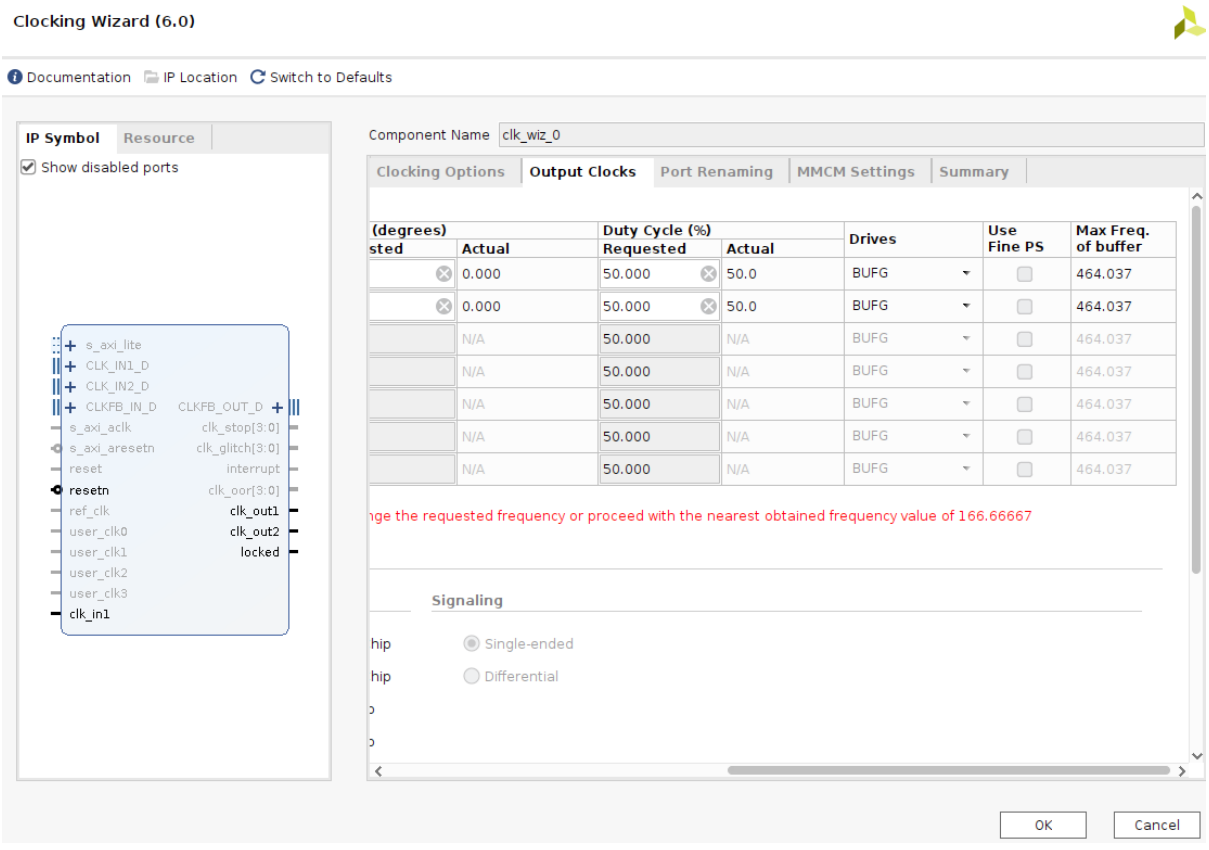


Fig. 2.3.6-1. Arty A7 clk_wiz_0 generation, steps 1a and 1b.


Clocking Wizard (6.0)
Documentation
IP Location
Switch to Defaults
IP Symbol
Resource
Show disabled ports
s_axi_lite
CLK_IN1_D
CLK_IN2_D
CLKFB_IN_D
CLKFB_OUT_D
s_axi_aclk
clk_stop[3:0]
s_axi_aresetn
clk_glitch[3:0]
reset
interrupt
resetn
clk_oor[3:0]
ref_clk
clk_out1
user_clk0
clk_out2
user_clk1
locked
user_clk2
user_clk3
clk_in1
Component Name
clk_wiz_0
Clocking Options
Output Clocks
Port Renaming
MMCM Settings
Summary
The phase is calculated relative to the active input clock.
Output Clock
Port Name
Output Freq (MHz)
Requested
Actual
Phase (degrees)
Requested
Actual
clk_out1
166.667
166.66667
0.000
clk_out2
200
200.00000
clk_out3
100.000
N/A
clk_out4
clk_out5
clk_out6
clk_out7
WARNING : The Requested frequency value for clk_out1 can not be achieved. Please change the requested frequency
USE CLOCK SEQUENCING
Clocking Feedback
Source
Signaling
Automatic Control On-Chip
Automatic Control Off-Chip
User-Controlled On-Chip
User-Controlled Off-Chip
Single-ended
Differential
Sequence Number
OK
Cancel


Clocking Wizard (6.0)
Documentation
IP Location
Switch to Defaults
IP Symbol
Resource
Show disabled ports
Component Name
clk_wiz_0
Clocking Options
Output Clocks
Port Renaming
MMCM Settings
Summary
Duty Cycle (%)
Requested
Actual
Drives
Use Fine PS
Max Freq. of buffer
BUFG
464.037
Signaling
Single-ended
Differential
OK
Cancel

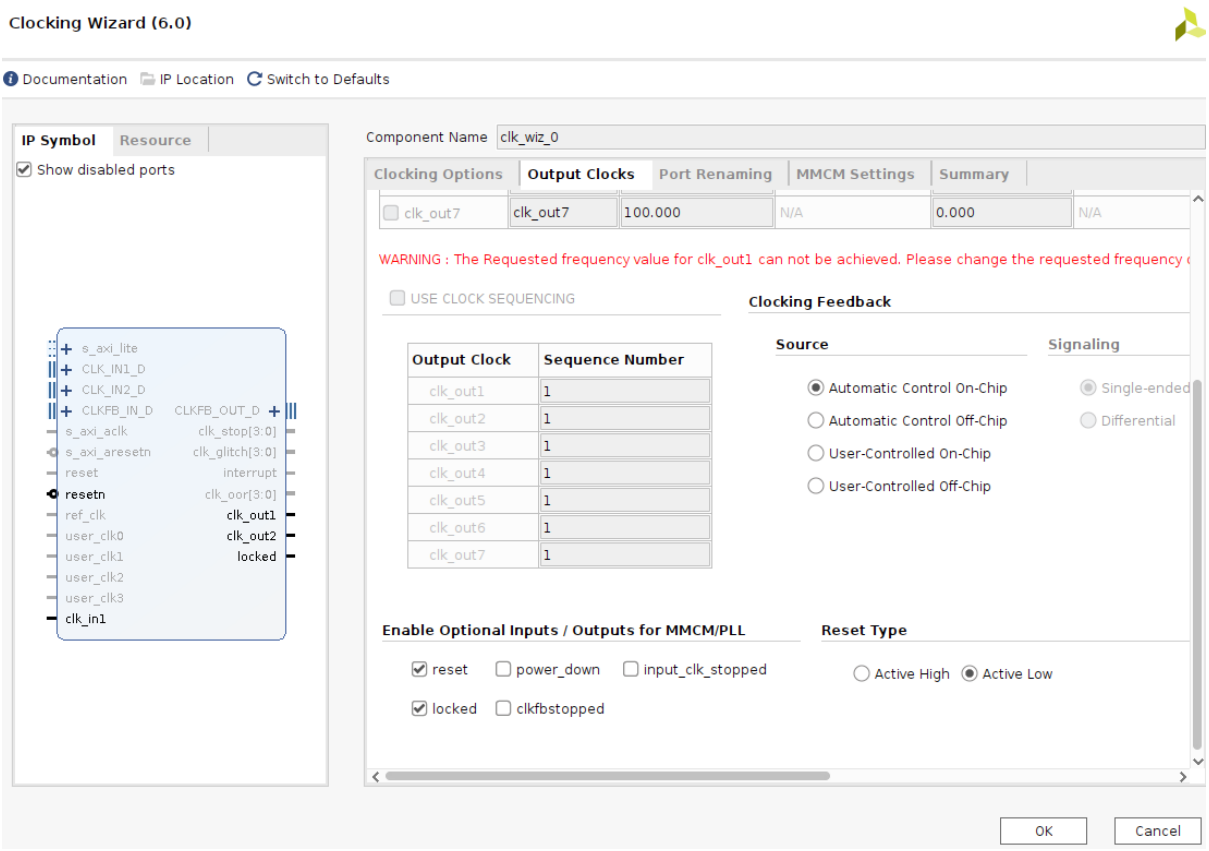


Fig. 2.3.6-2. Arty A7 clk_wiz_0 generation, steps 2a, 2b and 2c.

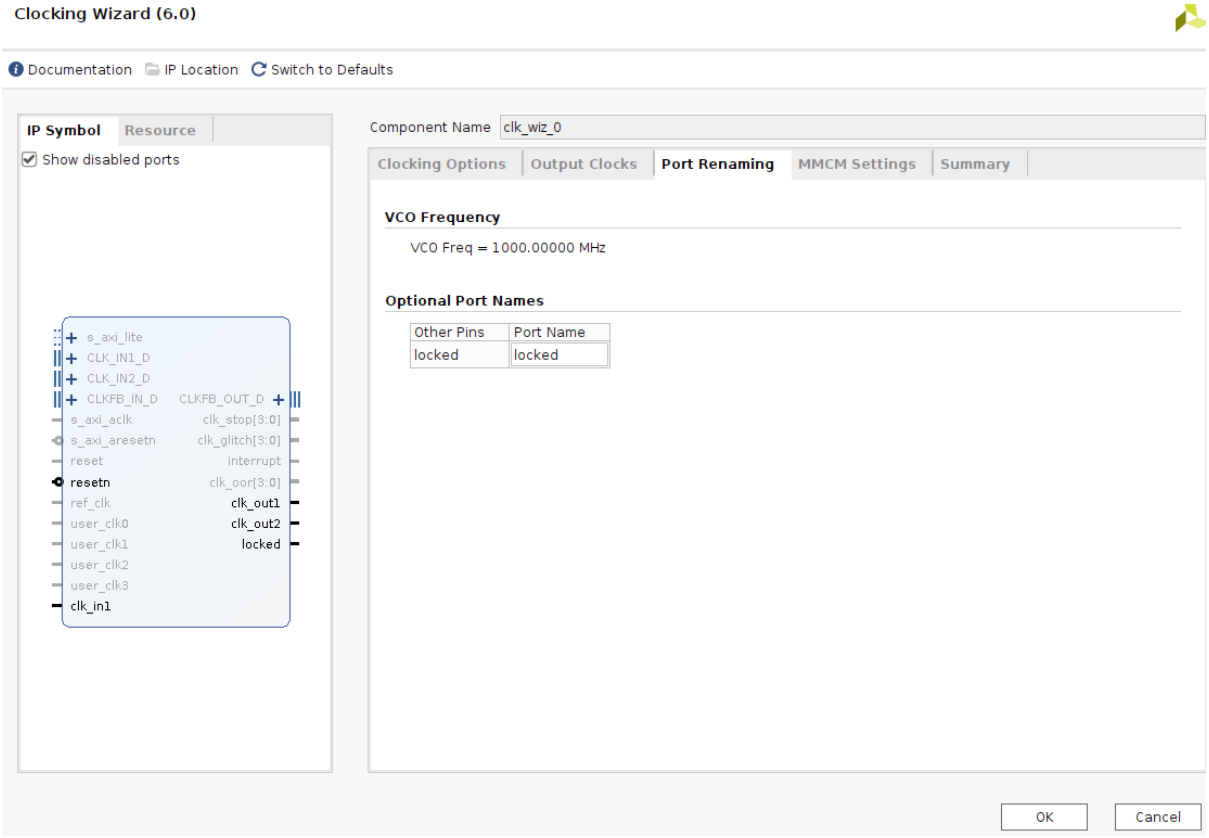


Fig. 2.3.6-3. Arty A7 clk_wiz_0 generation, step 3.

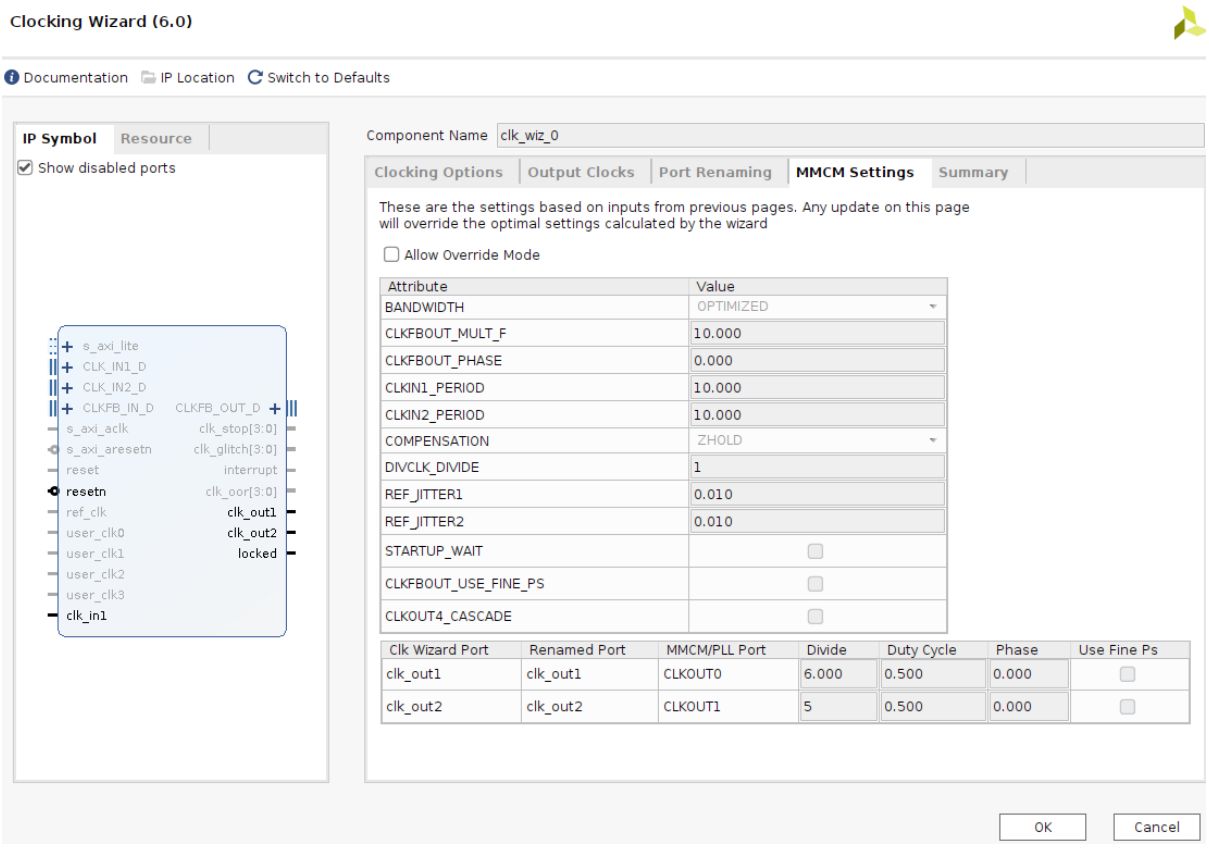


Fig. 2.3.6-4. Arty A7 clk_wiz_0 generation, step 4.

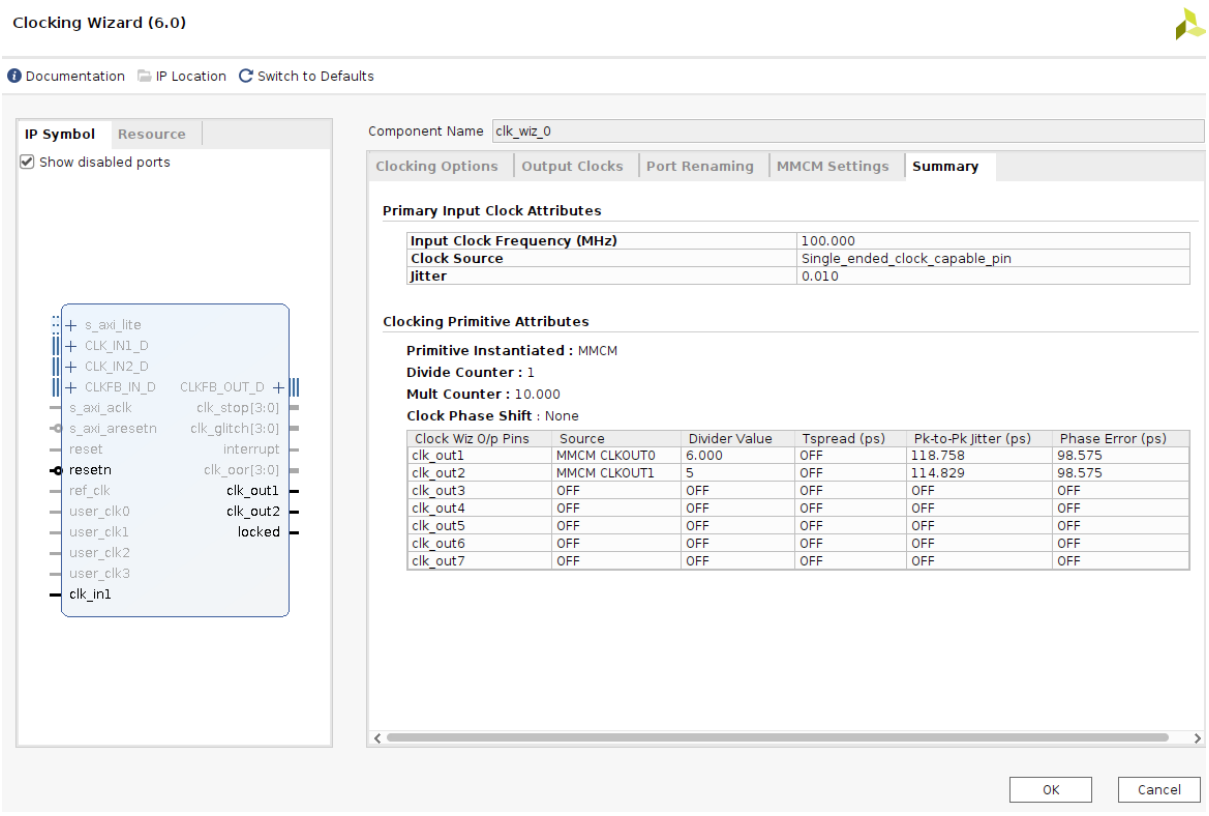


Fig. 2.3.6-5. Arty A7 clk_wiz_0 generation, step 5.

2.3.7. clk_wiz_1 for Nexys A7

I now show the step by step **clk_wiz_1** generation for Nexys A7.

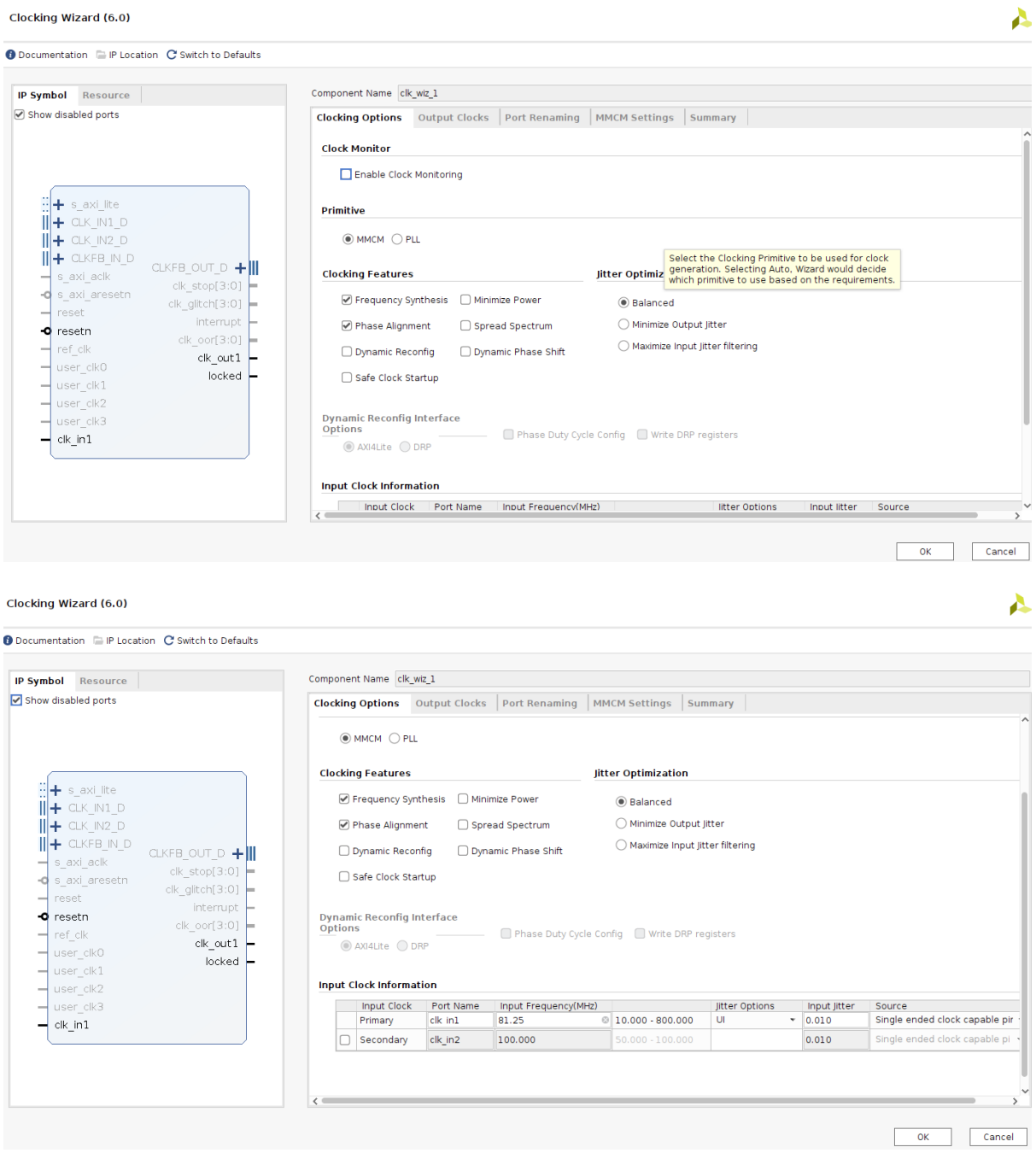
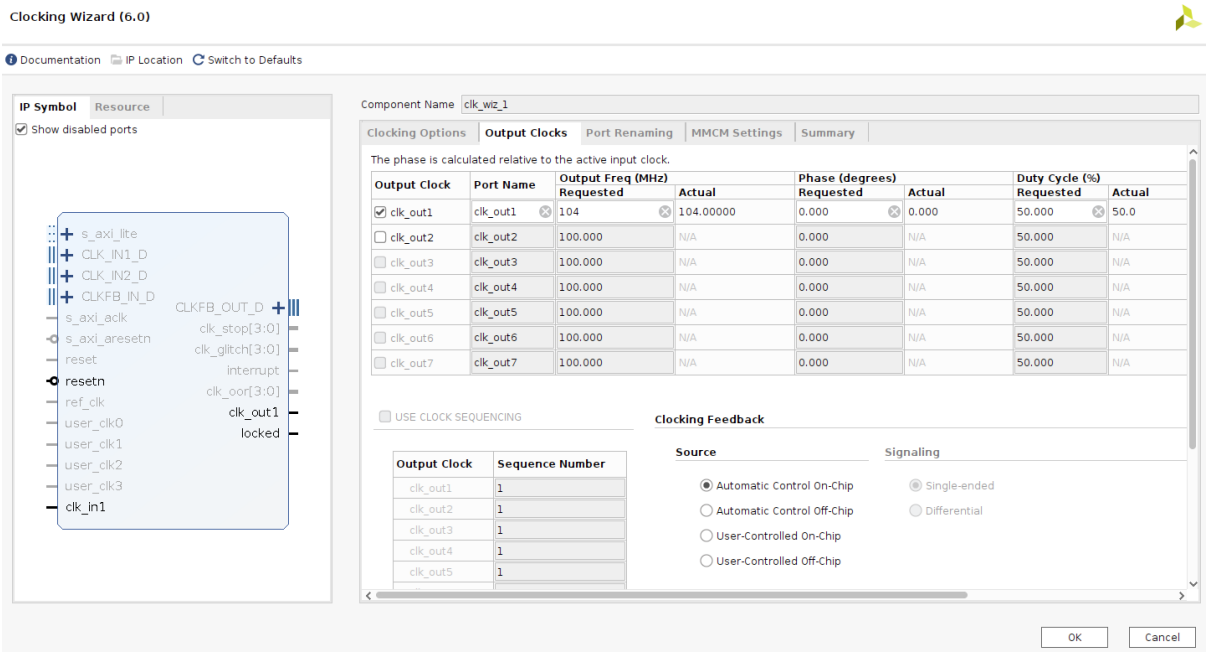
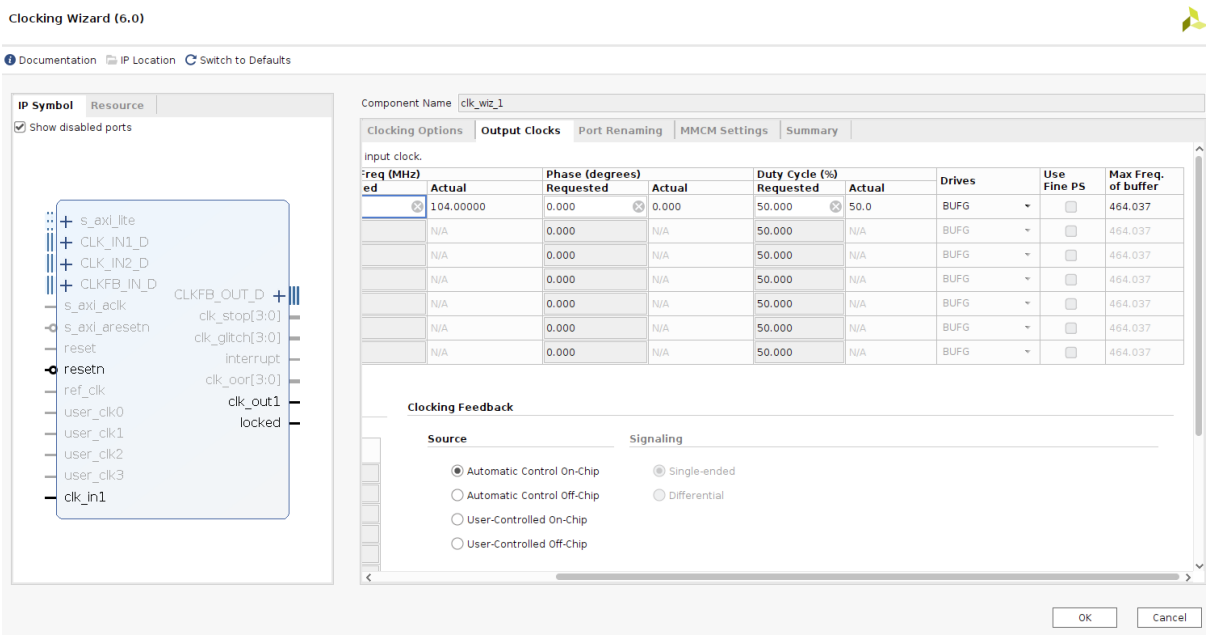


Fig. 2.3.7-1. Nexys A7 clk_wiz_1 generation, steps 1a and 1b.

In Fig. 2.3.2-3, Nexys A7 MIG generation step 4a, because the Phy to controller Ratio is 4:1, and because the memory clock period is 3.077ps (the frequence is 324.99 MHz), then the MIG UI clock frequency is 81.25 (here clk_in1). For more info, see listing 2.3.4-1.


Clocking Wizard (6.0)
Documentation
IP Location
Switch to Defaults
IP Symbol
Resource
Show disabled ports
s_axi_lite
CLK_IN1_D
CLK_IN2_D
CLKFB_IN_D
s_axi_aclk
s_axi_aresetn
reset
resetn
ref_clk
user_clk0
user_clk1
user_clk2
user_clk3
clk_in1
CLKFB_OUT_D
clk_stop[3:0]
clk_glitch[3:0]
interrupt
clk_oor[3:0]
clk_out1
locked
Component Name
clk_wiz_1
Clocking Options
Output Clocks
Port Renaming
MMCM Settings
Summary
The phase is calculated relative to the active input clock.
Output Clock
Port Name
Output Freq (MHz)
Requested
Actual
Phase (degrees)
Requested
Actual
Duty Cycle (%)
Requested
Actual
clk_out1
clk_out1
104
104.00000
0.000
0.000
50.000
50.0
clk_out2
clk_out2
100.000
N/A
0.000
N/A
50.000
N/A
clk_out3
clk_out3
100.000
N/A
0.000
N/A
50.000
N/A
clk_out4
clk_out4
100.000
N/A
0.000
N/A
50.000
N/A
clk_out5
clk_out5
100.000
N/A
0.000
N/A
50.000
N/A
clk_out6
clk_out6
100.000
N/A
0.000
N/A
50.000
N/A
clk_out7
clk_out7
100.000
N/A
0.000
N/A
50.000
N/A
USE CLOCK SEQUENCING
Output Clock
Sequence Number
clk_out1
1
clk_out2
1
clk_out3
1
clk_out4
1
clk_out5
1
Clocking Feedback
Source
Signaling
Automatic Control On-Chip
Automatic Control Off-Chip
User-Controlled On-Chip
User-Controlled Off-Chip
Single-ended
Differential
OK
Cancel


Clocking Wizard (6.0)
Documentation
IP Location
Switch to Defaults
IP Symbol
Resource
Show disabled ports
s_axi_lite
CLK_IN1_D
CLK_IN2_D
CLKFB_IN_D
s_axi_aclk
s_axi_aresetn
reset
resetn
ref_clk
user_clk0
user_clk1
user_clk2
user_clk3
clk_in1
CLKFB_OUT_D
clk_stop[3:0]
clk_glitch[3:0]
interrupt
clk_oor[3:0]
clk_out1
locked
Component Name
clk_wiz_1
Clocking Options
Output Clocks
Port Renaming
MMCM Settings
Summary
input clock.
Actual
Phase (degrees)
Requested
Actual
Duty Cycle (%)
Requested
Actual
Drives
Use Fine PS
Max Freq. of buffer
104.00000
0.000
0.000
50.000
50.0
BUFG
464.037
N/A
0.000
N/A
50.000
N/A
BUFG
464.037
Clocking Feedback
Source
Signaling
Automatic Control On-Chip
Automatic Control Off-Chip
User-Controlled On-Chip
User-Controlled Off-Chip
Single-ended
Differential
OK
Cancel

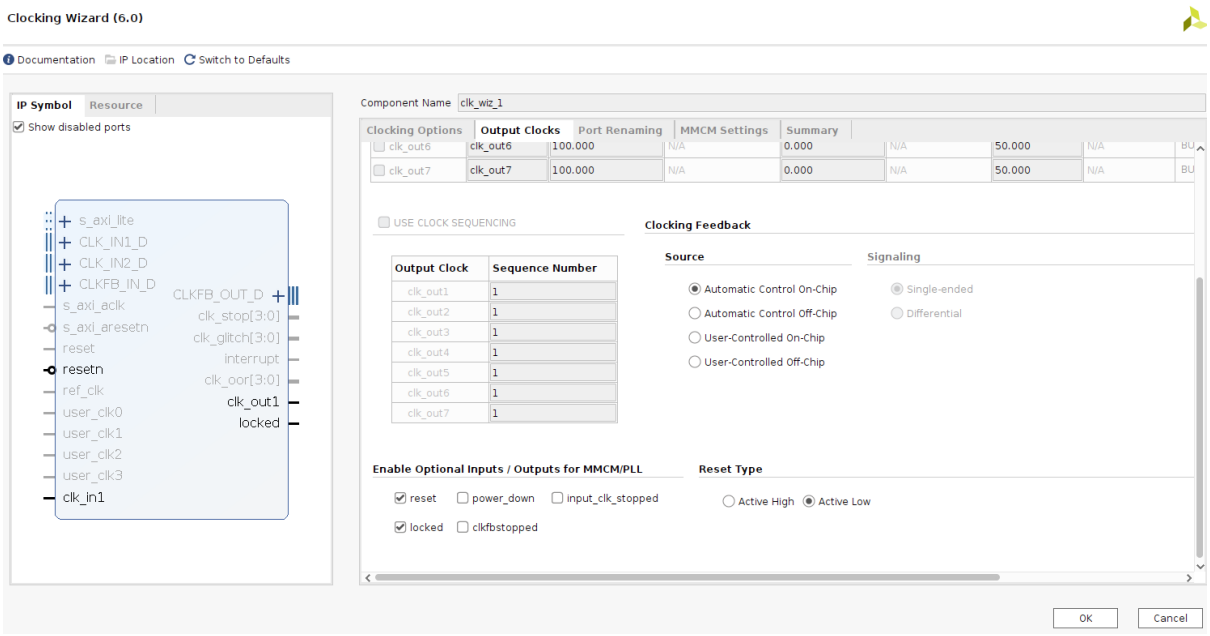


Fig. 2.3.7-2. Nexys A7 clk_wiz_1 generation, steps 2a, 2b and 2c.

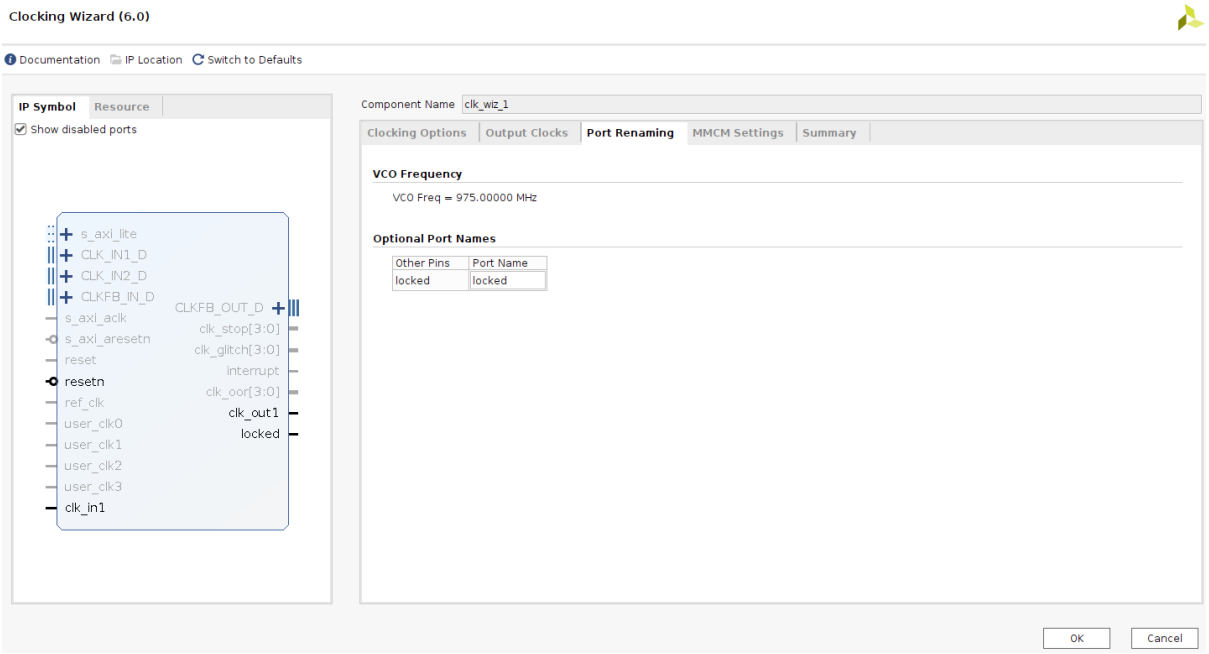


Fig. 2.3.7-3. Nexys A7 clk_wiz_1 generation, step 3.

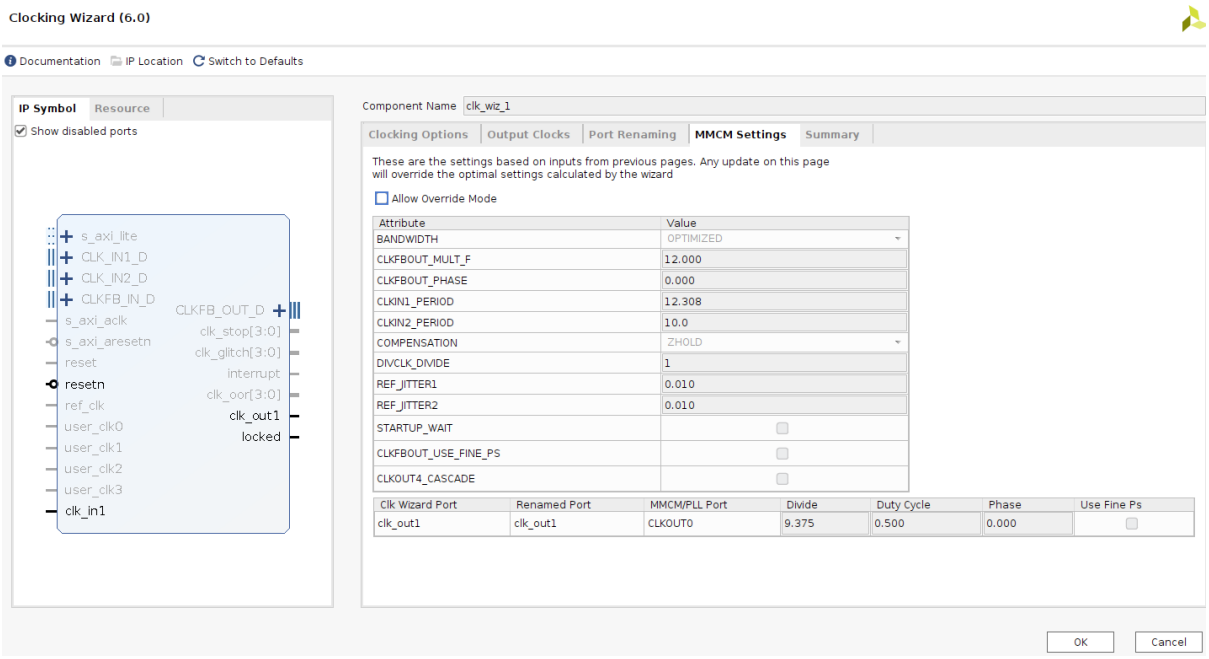


Fig. 2.3.7-4. Nexys A7 clk_wiz_1 generation, step 4.

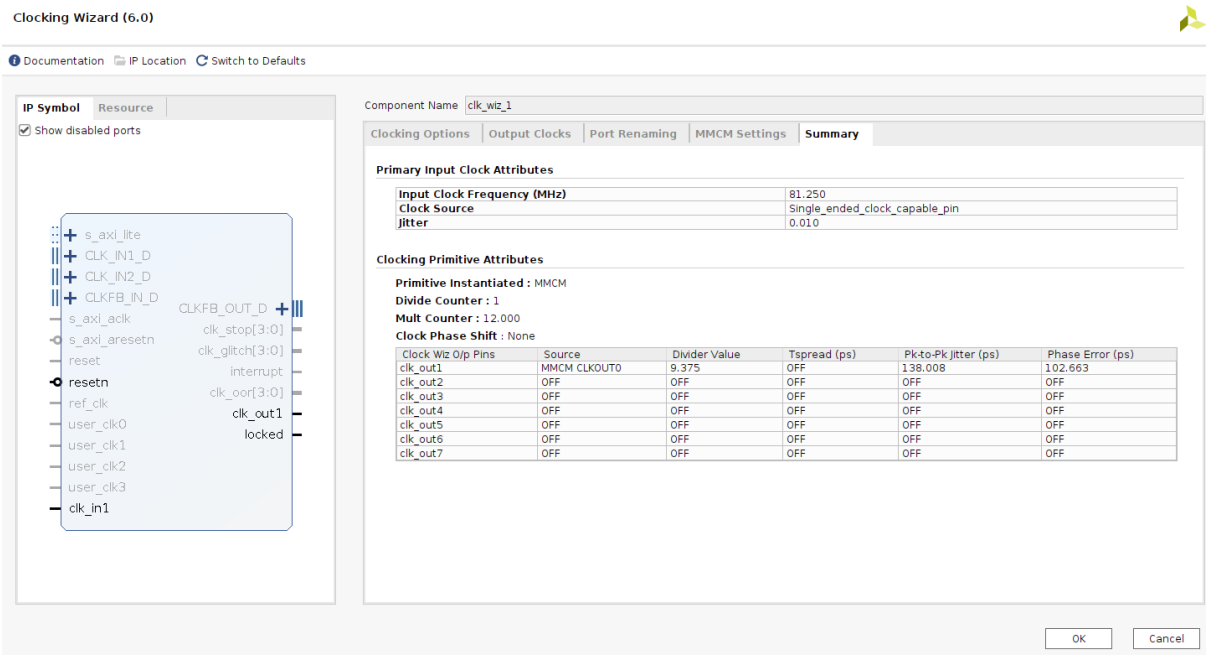


Fig. 2.3.7-5. Nexys A7 clk_wiz_1 generation, step 5.

2.3.8. clk_wiz_1 for Arty A7

I now show the step by step **clk_wiz_1** generation for Arty A7.

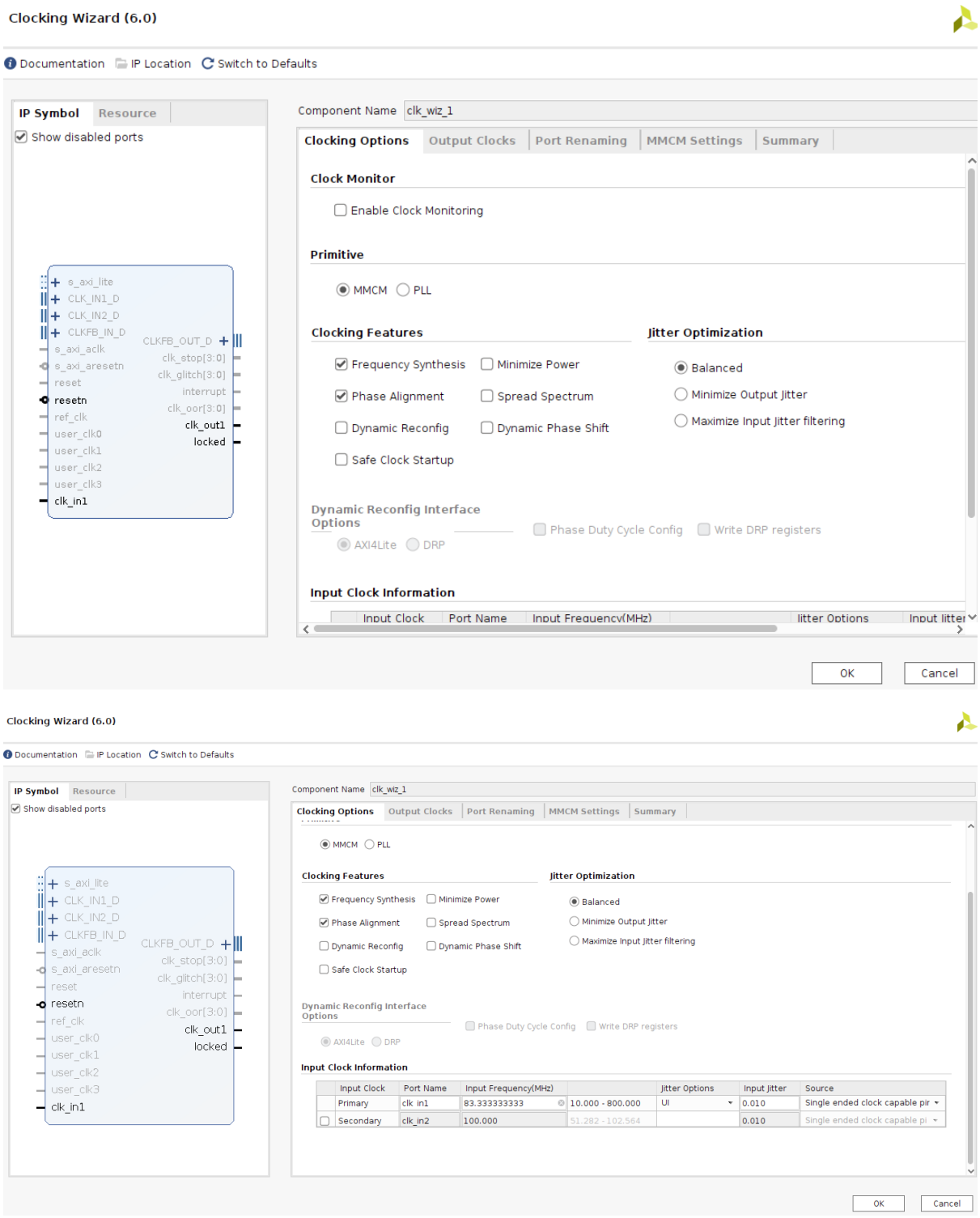
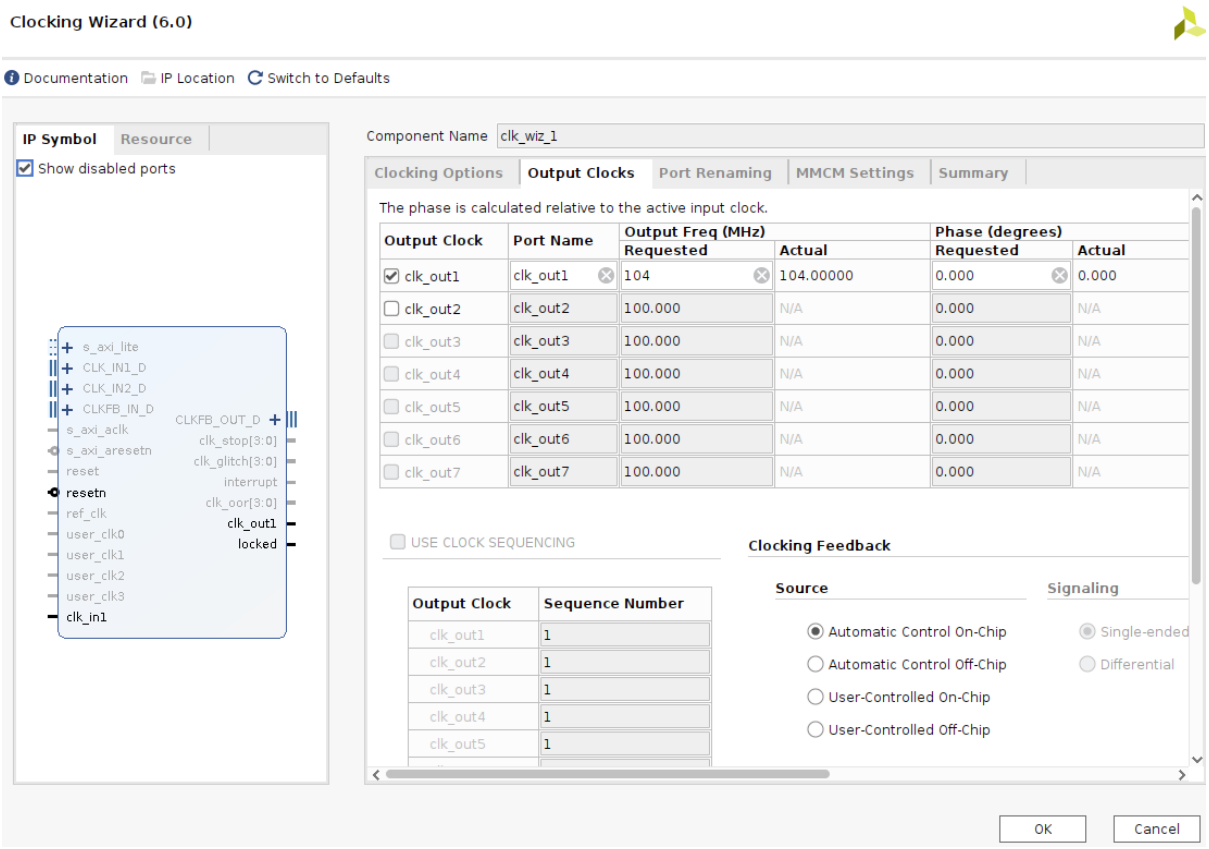
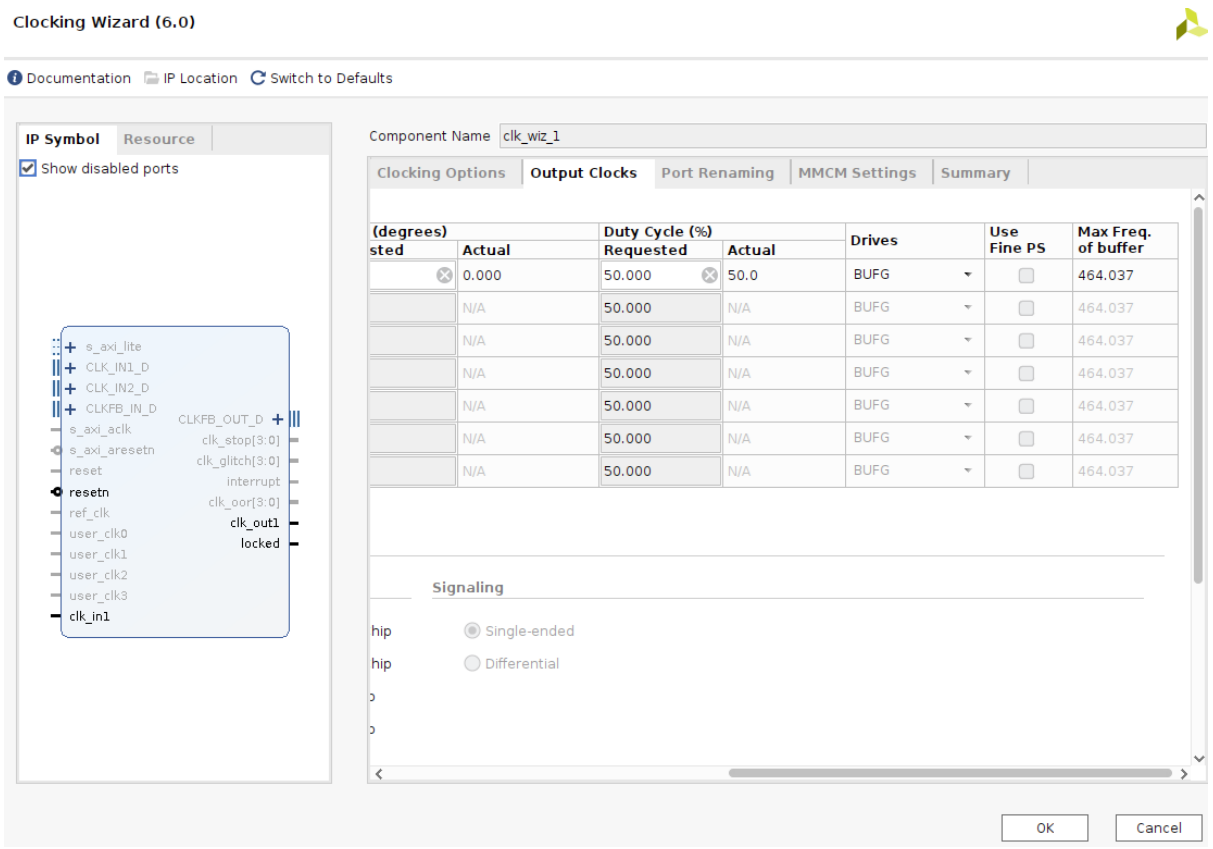


Fig. 2.3.8-1. Arty A7 clk_wiz_1 generation, steps 1a and 1b.

In Fig. 2.3.3-3, Arty A7 MIG generation step 4a, because the Phy to controller Ratio is 4:1, and because the memory clock period is 3ps (the frequence is 333.33 MHz), then the MIG UI clock frequency is 83.33 (here clk_in1). For more info see listing 2.3.4-1.





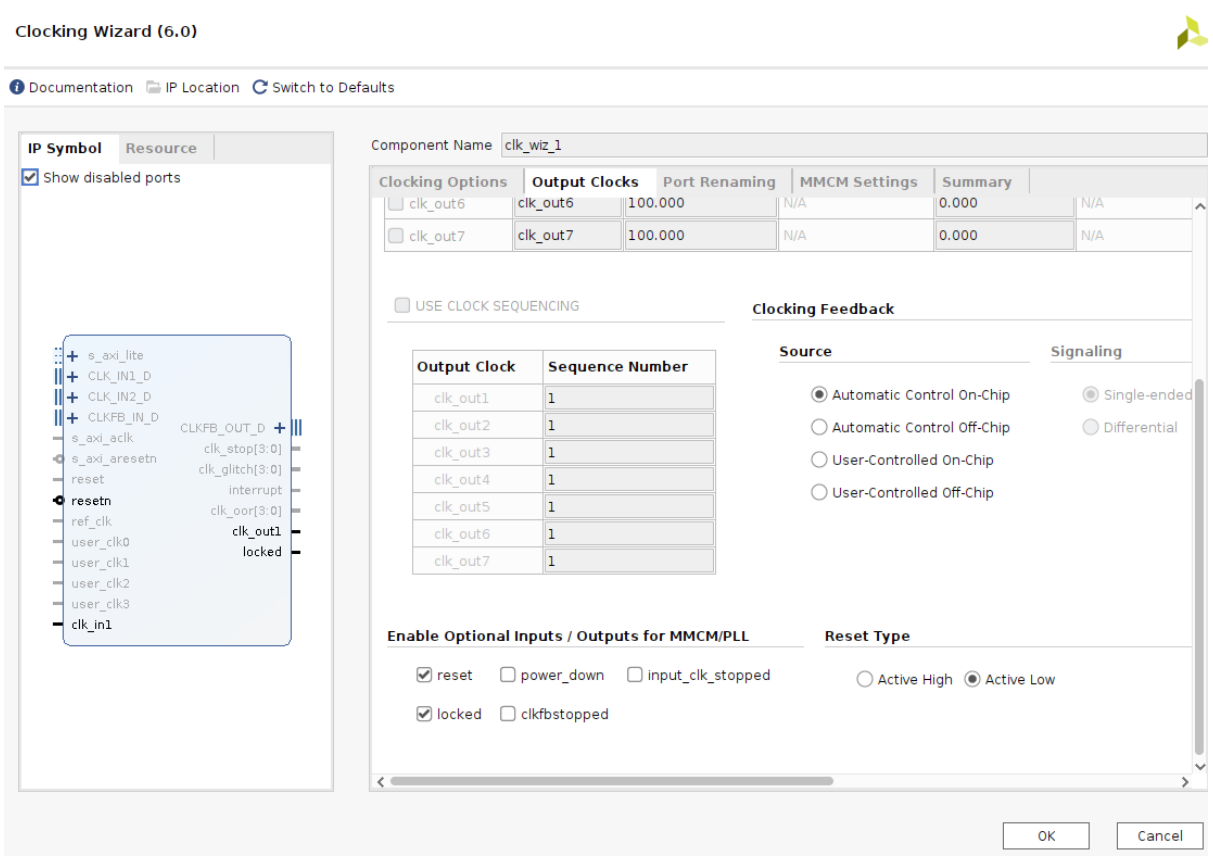


Fig. 2.3.8-2. Arty A7 clk_wiz_1 generation, steps 2a, 2b and 2c.

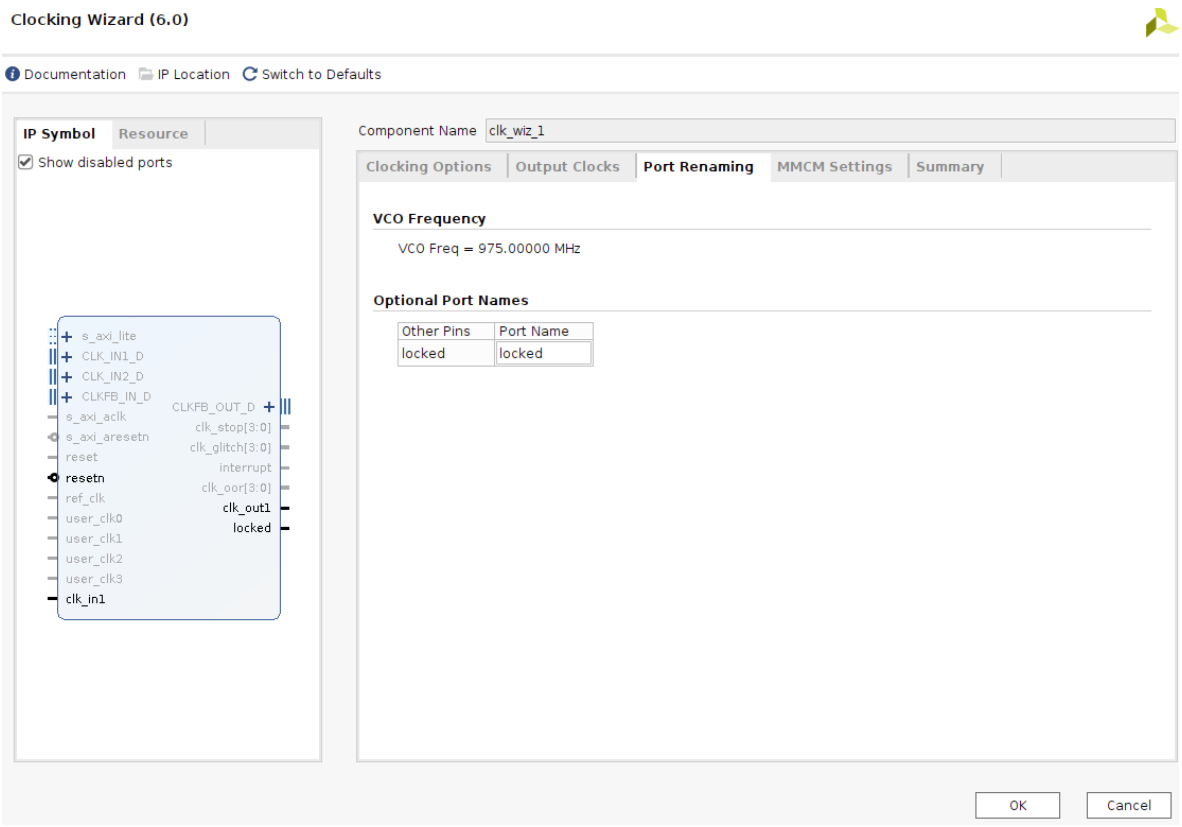


Fig. 2.3.8-3. Arty A7 clk_wiz_1 generation, step 3.

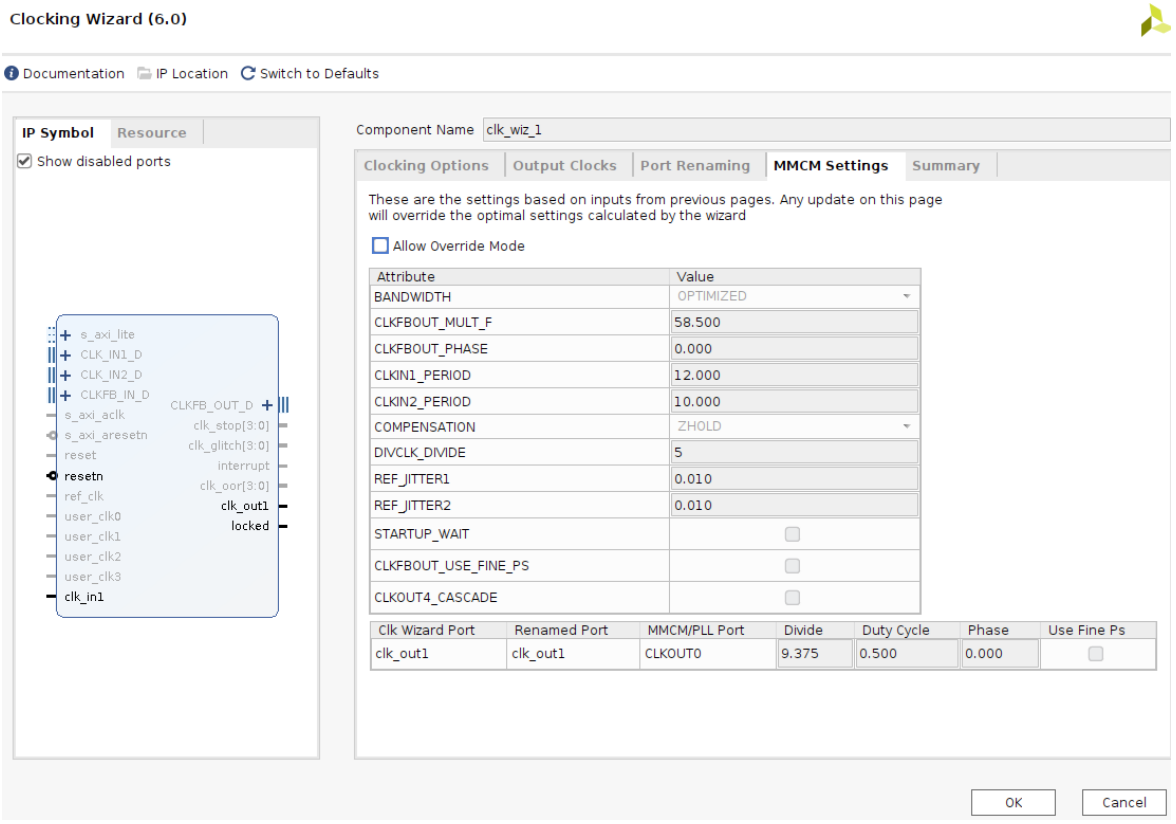


Fig. 2.3.8-4. Arty A7 clk_wiz_1 generation, step 4.

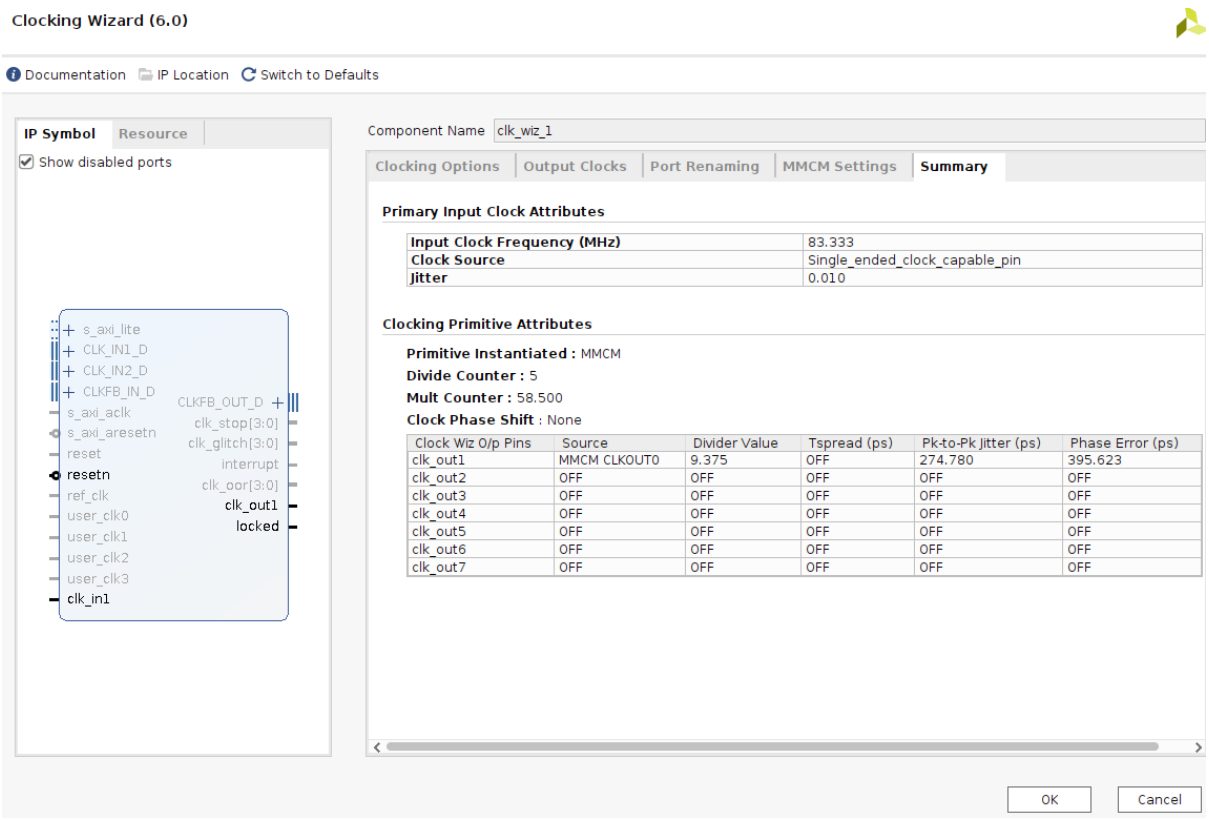


Fig. 2.3.8-5. Arty A7 clk_wiz_1 generation, step 5.

2.4 Building bootloader, Linux, initramfs and device tree blob for RLSoC

In the final software product that will be sent to the rlsoc RAM memory via the serial line, we concatenate the bbl, linux kernel, initramfs and dtb file.

2.4.1 Buildroot

Buildroot is an easy to use linux build system. It can build the linux kernel and the root filesystem in few easy steps. It can be downloaded from [9]. We have included in the folder **rvsoc_patch** the configuration files for buildroot 2019.03 and 2021.08: **buildroot-2019.11.3-5.0-defconfig** and **buildroot-2021.08-5.13.19.defconfig**. One is configured to be used with linux kernel 5.0 and the other with 5.13.19.

We will use buildroot to build the initramfs file system. Initramfs is a file system that resides in the RAM memory; its contents are volatile – meaning that after the system reboot, the changes are lost. Considering that the buildroot folder is in the same directory as **rvsoc_patch**, we can build the initramfs with the following commands:

```
make defconfig BR2_DEFCONFIG=../rvsoc_patch/buildroot-2021.08-5.13.19.defconfig
make
```

The first command sets up the buildroot configuration and the second runs the build process.

If you want to setup the buildroot configuration manually, run the **make menuconfig** command and select the options shown in Listing 2.4.1-1 (tested for buildroot 2021.08):

```
Target options
  Target Architecture (RISCV)
  Target Binary Format (ELF)
  Target Architecture Variant (Custom architecture)
  Instruction Set Extensions
    Integer Multiplication and Division (M)
    Atomic Instructions (A)
    Compressed Instructions (C)
  Target Architecture Size (32-bit)
  Target ABI (ilp32)
Build options
  Enable compiler cache
  [ ]Build code with PIC/PIE
  Stack Smashing Protection (None)
  RELRO Protection (None)
  Buffer-overflow Detection (FORTIFY_SOURCE) (None)
Toolchain
  (buildroot) custom toolchain vendor name
```

```
    C library (glibc)
    Kernel Headers (Linux 5.13.x kernel headers)
    Binutils Version (binutils 2.37)
    GCC compiler Version (gcc 11.x)
    Build cross gdb for the host
System configuration
    Init system (BusyBox)
    /dev management (Dynamic using devtmpfs only)
    Enable root login with password
    ()  Root password
    /bin/sh (busybox' default shell)
Kernel
    [ ] Linux kernel
Target packages
    Games - sl
Filesystem images
    cpio the root filesystem (for use as an initial RAM filesystem)
        Compression method (gzip)
    ext2/3/4 root filesystem
    ext2/3/4 variant (ext2 (rev1))
    (4M) exact size
    [ ] tar the root filesystem
```

Listing 2.4.1-1. Buildroot 2021.08 configuration options.

For buildroot 2019.11.3, the settings that differ from buildroot 2021.08 are:

```
Toolchain
  Kernel Headers (Manually specified Linux version)
    (5.0) linux version
  Custom kernel headers series (5.0.x)
  Binutils Version (binutils 2.31.1)
  GCC compiler Version (gcc 8.x)
```

Listing 2.4.1-2. Buildroot 2019.11.3 configuration options.

It can be seen that the compiler toolchain will be built along with the initramfs file system and the linux kernel will not be built - we will build it separately. The output files are listed in buildroot's **output/images** folder; we are interested in **rootfs.cpio.gz**, the gzipped initramfs cpio archive (which is smaller than 4MB).

We are also interested in the gcc toolchain that buildroot creates. It can be found in **buildroot-2021.08-5.13/output/host/bin** folder and the compiler filenames are started with **riscv32-buildroot-linux-gnu-**. We will use it to compile the linux kernel. Please add it to the PATH variable.

2.4.2 Device tree blob

The device tree blob contains a static description of the hardware system where the software runs on. We expect that this book readers know the basics of device tree blobs; if not, please follow the Thomas Petazzoni's presentation on this subject [10]. The device tree source file that we used is **devicetree_104mhz.dts** and can be found in the **devicetree_104mhz** folder. From the dts we can build the binary dtb with the command:

dtc -I dts -O dtb devicetree_104mhz.dts -o devicetree.dtb

The dtb will be attached to the bbl, linux kernel and initramfs in the final software product.

2.4.3 The Linux kernel

The **rvsoc_patch** folder contains prebuilt linux kernel configurations: **linux-5.13.19.config** and **linux-5.0.config**. The linux kernel 5.13.19 can be built with the following commands (in parenthesis is the gcc toolchain path for building linux 5.0):

```
export    PATH=$PATH:/path/to/buildroot-2021.08-5.13/output/host/bin
(export   PATH=$PATH:/path/to/buildroot-2019.11.3/output/host/bin)
cp rvsoc_patch/linux-5.13.19.config linux-5.13.19/.config
cd linux-5.13.19
make -j5 ARCH=riscv CROSS_COMPILE=riscv32-buildroot-linux-gnu- oldconfig
make -j5 ARCH=riscv CROSS_COMPILE=riscv32-buildroot-linux-gnu- vmlinux
```

If you want to configure manually the linux kernel, the options to be set are shown in Listing 2.4.3-1. In bold are shown specific options. Please note that in the following, the option CONFIG_HVC_RISCV_SBI is not set and options CONFIG_RISCV_SBI, CONFIG_RISCV_SBI_V01 and CONFIG_SERIAL_EARLYCON_RISCV_SBI are set to y.

```
make -j5 ARCH=riscv CROSS_COMPILE=riscv32-buildroot-linux-gnu- tinyconfig
make -j5 ARCH=riscv CROSS_COMPILE=riscv32-buildroot-linux-gnu- menuconfig
General setup
  Automatically append version information to the version string
  Support for paging of anonymous memory (swap)
  Timers subsystem
     Timer tick handling (Periodic timer ticks (constant rate, no dynticks))
  Preemption Model (No Forced Preemption (Server))
  Control Group support
  Namespaces support
  UTS namespace
   PID Namespaces
  Network namespace
  Initial RAM filesystem and RAM disk (initramfs/initrd) support   (/path/to/buildroot-
2021.08-5.13/output/images/rootfs.cpio.gz) Initramfs source
  (0)   User ID to map to 0 (user root)
  (0)   Group ID to map to 0 (group root)
     Support initial ramdisk/ramfs compressed using gzip
  Compiler optimization level (Optimize for size (-Os))
  Configure standard kernel features (expert users)
     Multiple users, groups and capabilities support
     Sysfs syscall support
     Posix Clocks & timers
     Enable support for printk
     BUG() support
     Enable ELF core dumps (NEW)
     Enable full-sized data structures for core
     Enable futex support
```

```
      Enable eventpoll support
      Enable signalfd() system call
      Enable timerfd() system call
      Enable eventfd() system call
      Use full shmem filesystem (NEW)
      Enable AIO support
      Enable IO uring support
      Enable madvise/fadvise syscalls
  Enable membarrier() system call
  Load all symbols for debugging/ksymoops
  Embedded system
  Enable VM event counters for /proc/vmstat
  Enable SLUB debugging support
  Disable heap randomization
  Choose SLAB allocator (SLUB (Unqueued Allocator))
  Allow slab caches to be merged
MMU-based Paged Memory Management Support
CPU errata selection
  RISC-V alternative scheme

Platform type
  Base ISA (RV32I)
  Kernel Code Model (medium low code model)
  Maximum Physical Memory (1GiB)
  Emit compressed instructions when building Linux

Kernel features
  Timer frequency (100 HZ)
   SBI v0.1 support

Boot options
  [ ] UEFI runtime support

General architecture-dependent options
  Enable seccomp to safely execute untrusted bytecode
  Stack Protector buffer overflow detection
  Strong Stack Protector
  Link Time Optimization (LTO) (None)
  (8) Number of bits to use for ASLR of mmap base address
  Provide system calls for 32-bit time_t
  [ ] Make kernel text and rodata read-only
  GCOV-based kernel profiling
  GCC plugins (with nothing)

Enable the block layer
  Partition types
```

```
      Advanced partition selection
      PC BIOS (MSDOS partition tables) support (NEW)

IO Schedulers
  MQ deadline I/O scheduler (NEW)
  Kyber I/O scheduler (NEW)

Executable file formats
  Kernel support for ELF binaries
  Write ELF core dumps with partial segments (NEW)
  Kernel support for scripts starting with #!
  Enable core dump support

Memory Management options
  Memory model (Flat Memory)

Networking support
  Networking options
     Packet socket
     Unix net domain sockets
     TCP/IP networking
  Plan 9 Resource Sharing Support (9P2000)  --->
     9P Virtio Transport
     Debug information
  Netlink interface for ethtool

Device drivers
  [ ] PCI support, see tinyemu config for PCI if you want to enable it
   Generic Driver Options
     Support for uevent helper
     ()    path to uevent helper
     Maintain a devtmpfs filesystem to mount at /dev
     Automount devtmpfs at /dev, after the kernel mounted the rootfs
     Select only drivers that don't need compile-time external firmware
     Disable drivers features which enable custom firmware building
     Firmware loader  --->
     Firmware loading facility
     Allow device coredump
  Device Tree and Open Firmware support ---> nothing
     [ ]   Device Tree overlays
  Block devices
     Virtio block driver
  Network devices support
     Network core driver support
     Virtio network driver
     Failover driver
  Input device support
```

```
    Event interface
    Keyboards
      AT keyboard
    Mice
      All defaults
    Hardware I/O Ports
      Serial I/O support
      Serial port line discipline
      PS/2 driver library
  Character devices
    Enable TTY
    Virtual terminal
    Enable character translations in console
    Support for console on virtual terminal
    Support for binding and unbinding console drivers
    Unix98 PTY support
    Legacy (BSD) PTY support
    (256)   Maximum number of legacy PTY in use
    Automatically load TTY Line Disciplines
    Serial drivers
        [ ] 8250/16550 and compatible serial support
        Early console using RISC-V SBI
    [ ] RISC-V SBI console support
    Virtio console
    /dev/mem virtual device support
  Graphics support
    VGA Arbitration
    (16)  Maximum number of GPUs
    Framebuffer devices
      [ ] Support for frame buffer devices
    Console display driver support
      [ ] VGA text console
      (80) Initial number of console screen columns
      (25) Initial number of console screen rows
      Framebuffer console support
  Virtualization drivers
  Virtio drivers
    Virtio input driver
    Platform bus driver for memory mapped virtio devices
    Memory mapped virtio devices parameter parsing
  VHOST drivers
  [ ] IOMMU Hardware Support
  Common Clock Framework
  IRQ chip support
    RISC-V Local Interrupt Controller
    SiFive Platform-Level Interrupt Controller
```

```
    (CONFIG_IRQ_DOMAIN_HIERARCHY=y selected by SIFIVE_PLIC [=y] && RISCV [=y])

File Systems
  The Extended 4 (ext4) filesystem
  Use ext4 for ext2 file systems (NEW)
  Enable POSIX file locking API
   Enable Mandatory file locking
  Inotify support for userspace
  Pseudo filesystems
    /proc file system support
    Sysctl support (/proc/sys)
    Include /proc/<pid>/task/<tid>/children file
  sysfs file system support
  Tmpfs virtual memory file system support (former shm fs)
  Network file systems
    Plan 9 Resource Sharing Support (9P2000)

Security options
  Harden memory copies between kernel and userspace
  Allow user copy whitelist violations to fallback to object size
  First legacy 'major LSM' to be initialized (Unix Discretionary Access Controls)
  Kernel hardening options -> Memory initialization
    Initialize kernel stack variables at function entry (no automatic initialization (weakest))

Cryptographic API
  Cryptographic algorithm manager
  Disable run-time self tests
  Null algorithms
  Sequence Number IV Generator
  HMAC support
  CRC32c CRC algorithm
  SHA224 and SHA256 digest algorithm
  AES cipher algorithms
  NIST SP800-90A DRBG
  Jitter entropy Non-Deterministic Random Number Generator

Library routines
  CRC16 functions
  CRC32/CRC32c functions
  [ ] XZ decompression support

Kernel hacking
  printk and dmesg options
  Show timing information on printks
  Support symbolic error names in printf
  Compile-time checks and compiler options
    Enable full Section mismatch analysis
```

```
    Make section mismatch errors non-fatal
  Kernel debugging
  Miscellaneous debug code
  Tracers
```

Listing 2.4.3-1. The chosen Linux kernel configuration options

The linux kernel can be built using the command:

```
make -j5 ARCH=riscv CROSS_COMPILE=riscv32-buildroot-linux-gnu- vmlinux
```

2.4.4 Berkeley boot loader for RLSoC

The Berkeley boot loader (bbl) is the program that runs first at the software boot. It is responsible to start linux and pass to it the dtb. BBL sources in the version that we used can be downloaded in the following modes:

```
wget https://github.com/riscv/riscv-pk/archive/v1.0.0.tar.gz

or

git clone https://github.com/riscv/riscv-pk
cd riscv-pk
// list tags
git tag
git checkout v1.0.0
```

BBL must be patched in order to work for rlsoc. Considering that the bbl folder is named **riscv-pk**, then:

```
patch -d riscv-pk -p1 < rvsoc_patch/riscv-pk.patch
patch -d riscv-pk -p1 < rvsoc_patch/riscv-pk-laur.patch
```

Now we can build the BBL:

```
mkdir linux-kernel
cp linux-5.13.19/vmlinux linux-kernel/
cd riscv-pk
mkdir build && cd build
../configure --enable-logo --enable-print-device-tree --host=riscv32-buildroot-linux-gnu --
with-arch=rv32imac --with-payload=../../linux-kernel/vmlinux
make
```

Now we have the **bbl** executable file that contains the BBL and Linux kernel. BBL uses the raw image of the linux kernel computed using riscv32-buildroot-linux-gnu-objcopy applied on vmlinux elf image (see make messages). Please note that the size of bbl+kernel5.0 is half of bbl+kernel5.13.19. Starting with linux kernel 5.7, the raw image of vmlinux is much bigger than the linux kernel 5.6, even if the elf images have nearly the same sizes. That is because of the CONFIG_STRICT_KERNEL_RWX which is enabled by default starting with 5.7 and must be disabled if we want to have a smaller size.

2.4.5 The microcontroller

RLSoC has two RISC-V processors: one that implements priviledged instructions (m_RVCoreM verilog module, referred as “the processor”) and one that implements only unpriviledged instructions (m_RVuc verilog module, referred as “the microcontroller”).

The microcontroller runs raw RISC-V code compiled with the **riscv32-unknown-elf-gcc** compiler which can be found on the internet:

https://github.com/stnolting/riscv-gcc-prebuilt/releases/download/rv32i-2.0.0/riscv32-unknown-elf.gcc-10.2.0.rv32i.ilp32.newlib.tar.gz

Please extract it to an accessible folder; we have extracted it in **/usr/local/share/gcc-riscv32-unknown-elf**.

Then go to the **ucimage** folder and edit the **Makefile** such that the RISCV_PREFIX variable is similar to the one in the listing; please note the folder into which we have extracted the gcc archive.

```
RISCV_PREFIX = /usr/local/share/gcc-riscv32-unknown-elf/bin/riscv32-unknown-elf-
```

Then type at the command prompt the **make** command and copy the resulted **ucimage.hex** file in the **src** folder.

2.4.6 Putting it all together

BBL, the linux kernel, initramfs and dtb must be joined together (after building them using the compiler generated by buildroot), in order to be simulated or run by rlsoc. This is made in the **initmem_gen2** folder, by running:

./run.sh

Please read this simple script to see the commands that it runs. The specific thing to note is that it uses **riscv32-buildroot-linux-gnu-objcopy** to convert elf image of **bbl** to raw image.

Now, **initmem_gen2** contains the following files (please read **initmem_gen2/main.c** to see the simple process that builds them):

- **initmem.bin** – the file to be sent to the rlsoc after programming the FPGA (contains BBL, linux kernel with initramfs, and dtb);

- **init_kernel.txt** – the file used by rlsoc in simulation (contains BBL, linux kernel with initramfs, and dtb);

- **init_disk.txt** – this file size was set to 0 because we use initramfs (see note 2.4.5-2).

Then, in the **rvsoc_src_ver053/src** folder, we can simulate (after enabling SIM_MODE in **define.vh**) the system by running:

```
make veri
./simv
```

We can send the **initmem.bin** file to rlsoc in the FPGA board memory via the serial line (after programming the FPGA):

```
cp initmem_gen2/initmem.bin rvsoc_src_ver053/binary
cd rvsoc_src_ver053/binary
sudo python3 serial_sendfile.py 8 initmem.bin
```

Note 2.4.6-1: Rarely, there are times when errors appear on the serial connection. In this case, please retry the process of programming the FPGA and sending **initmem.bin**. To check if this file was sent successfully, on Nexys A7, when pressing btnu - the checksum of **initmem.bin** computed when received on the serial line is displayed on the board; and, when pressing btnd, the checksum of it as taken from memory. You can verify these checksums with the output shown by **initmem_gen2/run.sh** script. Arty A7 does not have a board display, so it is shown on **led1** if the two checksums are equal (but this does not mean that these are equal to the checksum shown by **initmem_gen2/run.sh**).

Note 2.4.6-2: we did not used the disk facility from rvsoc because it keeps it only in the RAM memory. If you want to test this feature:

- disable initramfs;

- copy **rootfs.ext2** from **buildroot-2021.08-5.13/output/images** as **root.bin** in the **initmem_gen2** folder;
- disable the WITHOUT_DISK option in **initmem_gen2/main.c** and recompile it;
- after that, you will have **init_disk.txt** and **initmem.bin** containing the file system;
- for simulation do the same as shown above;
- for FPGA implementation, define BIN_DISK_SIZE as (16*1024*1024) in **rvsoc_src_ver053/src/define.vh** and rebuild the bit file;

3. Building and testing TinyEMU

TinyEmu is a fast riscv emulator developed by Fabrice Bellard. It boots linux in about 2 seconds. We will build TinyEMU in the **tinyemu** folder of **road-to-linux**.

To build the **temu** binary, which is the TinyEMU main program, execute the following commands:

```
cd tinyemu
wget https://bellard.org/tinyemu/tinyemu-2019-12-21.tar.gz
tar -xf tinyemu-2019-12-21.tar.gz
patch -d tinyemu-2019-12-21 -p1 < tinyemu-laur.patch
cd tinyemu-2019-12-21
make
# go back to the tinyemu folder
cd -
```

Now we must build the bbl for tinyemu. This is **different** than bbl for rlsoc. To build bbl for tinyemu, follow the following steps:

```
git clone https://github.com/riscv/riscv-pk
cd riscv-pk
git checkout ac2c910b18c3e36cfd85080472e78ad2fe484325
cd ..
wget https://bellard.org/tinyemu/diskimage-linux-riscv-2018-09-23.tar.gz
tar -xf diskimage-linux-riscv-2018-09-23.tar.gz
patch -d riscv-pk -p1 < diskimage-linux-riscv-2018-09-23/patches/riscv-pk.diff
patch -d riscv-pk -p1 < ../rvsoc_patch/riscv-pk-laur.patch
cd riscv-pk
mkdir build && cd build
# configure of tinyemu bbl different than rlsoc bbl.
../configure --enable-logo --enable-print-device-tree --host=riscv32-buildroot-linux-gnu --
with-arch=rv32imafdc
make
# go back to tinyemu folder
cd ../..
```

Now that we have the bbl we can run tinyemu. Please execute the commands in Listing 3-1. We can stop tinyemu by pressing Ctrl-C, because we have passed it the **-ctrlc** option.

```
cp riscv-pk/build/bbl .
riscv32-buildroot-linux-gnu-objcopy -S -O binary bbl bbl32.bin
cp ../linux-kernel/vmlinux .
riscv32-buildroot-linux-gnu-objcopy -S -O binary vmlinux kernel-riscv32.bin
cp ../buildroot-2021.08-5.13.19/output/images/rootfs.ext2 ./root-riscv32.bin
```

```
# run tinyemu
./tinyemu-2019-12-21/temu -ctrlc temu-riscv32.cfg
```

Listing 3-1. Running TinyEMU

The same **temu** executable can run images for riscv32 or riscv64, depending on the options in the configuration file. An example of **temu-riscv32.cfg** is shown below.

```
/* VM configuration file */
{
   version: 1,
   machine: "riscv32",
   memory_size: 256,
   bios: "bbl32.bin",
   kernel: "kernel-riscv32.bin",
   cmdline: "console=hvc0 earlycon=sbi root=/dev/vda rw",
   drive0: { file: "root-riscv32.bin" },
/*    eth0: { driver: "user" }, */
}
```

Tinyemu generates by itself a valid dtb. It can be accessed in the **/tmp/riscvemu.dtb** folder. You can inspect it as a dts by running the command:

dtc -I dtb -O dts riscvemu.dtb -o riscvemu.dts

Please note that we have used **riscv32-buildroot-linux-gnu-objcopy** to convert **bbl** and **vmlinux** elf images to raw images. The filenames of these images are specified in **temu-riscv32.cfg**, which is the tinyemu config file. This is a little bit different than **diskimage-linux-riscv-2018-09-23/buildroot-riscv32.cfg** (which is not correctly named) because it tells linux in its command line to print messages early on the sbi console.

The **ctrlc** option passed to **temu** allows stopping temu by pressing Ctrl-C. Tinyemu can also be closed if pressing Ctrl-A and then x.

That's it! You can now use tinyemu and simulate very fast the riscv32 linux based systems.

4. RISC-V internals

4.1 General characteristics

Supported RISC-V architectures are rv32i or rv64i plus standard extensions (a)tomics, (m)ultiplication and division, (f)loat, (d)ouble, or (g)eneral for mafd. RV32e reduces the integer register count to 16 general-purpose registers. Supported ABIs are ilp32 (32-bit soft-float), ilp32d (32-bit hard-float), ilp32f, ilp32d (32-bit hard-float), ilp32e (embedded) and ilp64 ilp64f ilp64d (same but with 64-bit long and pointers).

RLSoC use a RISC-V 32bit processor named RVCoreM without pipeline. RVCoreP is a pipelined riscv32 processor but this won't be discussed here; the interested readers are advised to read [11]. RVCoreM has rv32imac architecture and ilp32 ABI: no floating-point instructions can be generated and no floating-point arguments are passed in registers.

For the tinyemu processor, we specified rv32imafdc architecture to BBL and ilp32 ABI to Linux: hardware floating-point instructions can be generated, but no floating-point arguments will be passed in registers. However, we did not used floating point in this project.

Register	ABI name	Description	Saver
x0	zero	Hard-wired zero	-
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	-
x4	tp	Thread pointer	-
x5	t0	Temporary / alternate link register	Caller
x6-x7	t1-2	Temporaries	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-1	Function arguments / return values	Caller
x12-x17	a2-7	Function arguments	Caller
x18-27	s2-11	Saved registers	Callee
x28-31	t3-6	Temporaries	Caller

Table 4-1. RV32I registers [12].

Instruction bits							Instruction Type
31…25	24…20	19…15	14…12	11…7	6…0		
func7	rs2	rs1	func3	rd	opcode		R-type
imm[11:0]		rs1	func3	rd	opcode		I-type
imm[11:5]	rs2	rs1	func3	imm[4:0]	opcode		S-type
imm[12,10:5]	rs2	rs1	func3	imm[4:1,11]	opcode		B-type
imm[31:12]				rd	opcode		U-type
imm[20,10:1,11,19:12]				rd	opcode		J-type

Table 4-2. RV32I instructions types [12]

Table 4.1 describes the registers and their ABI names for rv32i. Table 4.2 lists the RV32I instructions types.

"A RISC-V-compatible core might support multiple RISC-V-compatible hardware threads, or harts, through multithreading" [12]. The hart in rlsoc is the processor core.

"The FENCE instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output(O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the successor set following a FENCE before any operation in the predecessor set preceding the FENCE" [12].

"At any time, a RISC-V hardware thread (hart) is running at some privilege level encoded as a mode in one or more CSRs (control and status registers). Three RISC-V privilege levels are currently defined" [13] as shown in next table.

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	-
3	11	Machine	M

Table 4-3: RISC-V privilege levels [13].

The environment call (ECALL) instruction is used to transfer control to a higher privilege mode. "Typical use case for Unix-like operating systems is to delegate to S-mode the environment-call-from-U-mode exception" [13].

"The SYSTEM major opcode is used to encode all privileged instructions in the RISC-V ISA. These can be divided into two main classes: those that atomically read-modify-write control and status registers (CSRs), and all other privileged instructions" [13]. CSRs are registers which contain the working state of a riscv processor. The majority of the M-mode CSRs has S-mode equivalents – S-mode CSRs control the state of S-mode and Umode. S-mode CSRs have the mapping of the M-mode CSRs, but without M-mode control bits. We have sstatus, stvec, sip, sie, sepc, scause, etc.

Two famous CSRs are machine ISA register – **misa**, and machine hart id - **mhartid**.

4.2. The Machine status (mstatus) register

"The **mstatus** register keeps track of and controls the hart's current operating state. A restricted view of mstatus appears as the sstatus register in the S-level ISA" [13]. WPRI means reserved field. The fields of the **mstatus** register are shown in the table below.

31	30 23	22	21	20	19	18	17	16 15	14 13
SD	WPRI	TSR	TW	TVM	MXR	SUM	MPRV	XS[1:0]	FS[1:0]
1	8	1	1	1	1	1	1	2	2

12 11	10 9	8	7	6	5	4	3	2	1	0
MPP[1:0]	WPRI	SPP	MPIE	UBE	SPIE	WPRI	MIE	WPRI	SIE	WPRI
2	2	1	1	1	1	1	1	1	1	1

Table 4-1. **mstatus** register fields [13].

The most important fields are described in the next table.

Index	Field	Description
1	SIE	Global interrupt-enable bits, MIE and SIE, are provided for M-mode and S-mode respectively.
3	MIE	
5	SPIE	Machine/Supervisor previous interrupt enable. The state of interrupt enable prior to an interrupt.
7	MPIE	
8	SPP	Machine/Supervisor previous privilege level (the mode before) of the current interrupt.
12:11	MPP	
17	MPRV	Modify PRiVilege
18	SUM	permit Supervisor User Memory access
19	MXR	Make eXecutable Readable

Table 4-2. **mstatus** fields descriptions [13].

"The SPP bit indicates the privilege level at which a hart was executing before entering supervisor mode. When a trap is taken, SPP is set to 0 if the trap originated from user mode, or 1 otherwise. When an SRET instruction is executed to return from the trap handler, the privilege level is set to user mode if the SPP bit is 0, or supervisor mode if the SPP bit is 1; SPP is then set to 0" [13]. So, MPP and SPP can be written in order to enter a lower privilege mode when executing MRET or SRET instructions.

"xPIE holds the value of the interrupt-enable bit active prior to the trap, and xPP holds the previous privilege mode. The xPP fields can only hold privilege modes up to x, so MPP is two bits wide and SPP is one bit wide" [13].

"The SPIE bit indicates whether supervisor interrupts were enabled prior to trapping into supervisor mode. When a trap is taken into supervisor mode, SPIE is set to SIE, and SIE is set to 0. When an SRET instruction is executed, SIE is set to SPIE, then SPIE is set to 1" [13].

"Upon reset, a hart's privilege mode is set to M. The **mstatus** fields MIE and MPRV are reset to 0" [13]. "The MPRV (Modify PRiVilege) bit modifies the privilege level at which

loads and stores execute in all privilege modes" [13]. "When MPRV=0, loads and stores behave as normal" [13]. "When MPRV=1, load and store memory addresses are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP" [13].

"MRET and SRET clear **mstatus**.MPRV when leaving M-mode. An MRET or SRET instruction is used to return from a trap in M-mode or S-mode respectively. When executing an xRET instruction, supposing xPP holds the value y, xIE is set to xPIE; the privilege mode is changed to y; xPIE is set to 1; and xPP is set to U (or M if user-mode is not supported). If xPP≠M, xRET also sets MPRV=0" [13]. Here x can be M or S.

"The SUM (permit Supervisor User Memory access) bit modifies the privilege with which S-mode loads and stores access virtual memory. When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1 in Fig. 4.4-4) will fault. When SUM=1, these accesses are permitted" [13].

"The MXR (Make eXecutable Readable) bit modifies the privilege with which loads access virtual memory. When MXR=0, only loads from pages marked readable (R=1, Fig. 4.4-4) will succeed. When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed" [13].

"Platforms provide a real-time counter, exposed as a memory-mapped machine-mode read-write register, mtime. mtime must run at constant frequency, and the platform must provide a mechanism for determining the timebase of mtime. The mtime register has a 64-bit precision on all RV32 and RV64 systems. Platforms provide a 64-bit memory-mapped machine-mode timer compare register (mtimecmp), which causes a timer interrupt to be posted when the mtime register contains a value greater than or equal to the value in the mtimecmp register. The interrupt remains posted until mtimecmp becomes greater than mtime (typically as a result of writing mtimecmp). The interrupt will only be taken if interrupts are enabled and the MTIE bit is set in the **mie** register" [13].

4.3 Interrupts

WARL is an abbreviation for Write-Any Read-Legal field. A register field that can be written with any value, but returns only supported values when read. WLRL is Write/Read Only Legal Values.

"When a trap is taken into M-mode, **mepc** is written with the virtual address of the instruction that was interrupted or that encountered the exception" [13]. Similar happens for **sepc**.

"When a trap is taken into M-mode, **mcause** is written with a code indicating the event that caused the trap" [13]. Similar happens for **scause**. "The Interrupt bit in the **mcause** register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception" [13]. The **mcause** register format is shown in the next figure.

MXLEN-1	MXLEN-2 0
Interrupt	Exception code

Figure 4.3-1: Machine Cause register **mcause** [13].

mcause values after trap are shown in the next table. Reserved values are not shown.

Interrupt	Exception Code	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
1	1 ≥16	Available for platform use
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	ECALL from U-mode
0	9	ECALL from S-mode
0	11	ECALL from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault
0	16-31	Available for custom use

0	48-63	Available for custom use

Table 4.3-1. Machine cause register (**mcause**) values after trap [13].

To see the synchronous exception priority in decreasing priority order, please consult Table 3.7 from [13].

The supervisor cause register (**scause**) values after trap are shown in the table below.

Interrupt	Exception Code	Description
1	1	Supervisor software interrupt
1	5	Supervisor timer interrupt
1	9	Supervisor external interrupt
1	>= 16	Available for platform use
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store/AMO page fault
0	24-31, 48-63	Available for custom use

Table 4.3-2. Supervisor cause register (**scause**) values after trap [13].

Please note that AMO stands for atomic memory operation instruction. The standard atomic-instruction extension, named “A”, contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space.

Machine interrupt enable (**mie**) and machine interrupt pending (**mip**) CSRs have the same mapping. “Interrupt cause number i (as reported in CSR mcause) corresponds with bit i in both **mip** and **mie**. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are available for platform or custom use”. “An interrupt i will be taken if bit i is set in both **mip** and **mie**, and if interrupts are globally enabled. By default, M-mode interrupts are globally enabled if the hart’s current privilege mode is less than M, or if the current privilege mode is M and the MIE bit in the **mstatus** register is set” [13]. Similar happens to SIP and SIE.

15 12	11	10	9	8	7	6	5	4	3	2	1	0
0	MEIE	0	SEIE	0	MTIE	0	STIE	0	MSIE	0	SSIE	0

Table 4.3-3. Standard portion (bits 15:0) of **mie** [13].

"Bits mip.MEIP and mie.MEIE are the interrupt-pending and interrupt-enable bits for machinelevel external interrupts. MEIP is read-only in **mip**, and is set and cleared by a platform-specific interrupt controller" [13].

"Bits mip.MTIP and mie.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is read-only in **mip**, and is cleared by writing to the memory-mapped machine-mode timer compare register" [13].

"Bits mip.MSIP and mie.MSIE are the interrupt-pending and interrupt-enable bits for machinelevel software interrupts. MSIP is read-only in **mip**, and is written by accesses to memory-mapped control registers, which are used by remote harts to provide machine-level interprocessor interrupts" [13].

Machine Trap Value Register (mtval). "When a trap is taken into M-mode, **mtval** is either set to zero or written with exception-specific information to assist software in handling the trap".

"Machine Trap-Vector Base-Address Register (**mtvec**) is a read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE)".

MXLEN-1 2	1 0
BASE[MXLEN-1:2]	MODE

Fig. 4.3-2. Machine trap-vector base-address register (**mtvec**) [13].

Value	Name	Description
0	Direct	All exceptions set pc to BASE
1	Vectored	Asynchronous interrupts set pc to BASE+(4×cause.ExceptionCode). For example, a machine-mode timer interrupt (see table 4.3-1, timer interrupt is 7) causes the pc to be set to BASE+0x1c.
≥2	•	Reserved

Table 4.3-4. Encoding of mtvec MODE field [13].

"By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction. To increase performance, implementations can provide individual read/write bits within **medeleg** and **mideleg** to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level" [13].

Machine trap (exception and interrupt) delegation register **medeleg**, **mideleg**. "In systems with S-mode, the **medeleg** and **mideleg** registers must exist, and setting a bit in **medeleg** or **mideleg** will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. When a trap is delegated to S-mode, the **scause** register is written with the trap cause" [13].

"An xRET instruction can be executed in privilege mode x or higher, where executing a lower-privilege xRET instruction will pop the relevant lower-privilege interrupt enable and privilege mode stack. In addition to manipulating the privilege stack, xRET sets the **pc** to the value stored in the xepc register" [13].

"The Wait for Interrupt instruction (WFI) provides a hint to the implementation that the current hart can be stalled until an interrupt might need servicing" [13].

"The RISC-V platform-level interrupt controller (PLIC), is an interrupt controller specifically designed to work in the context of RISC-V systems. The PLIC multiplexes various device interrupts onto the external interrupt lines of Hart contexts, with hardware support for interrupt priorities" [14]. We will discuss the PLIC implementation later.

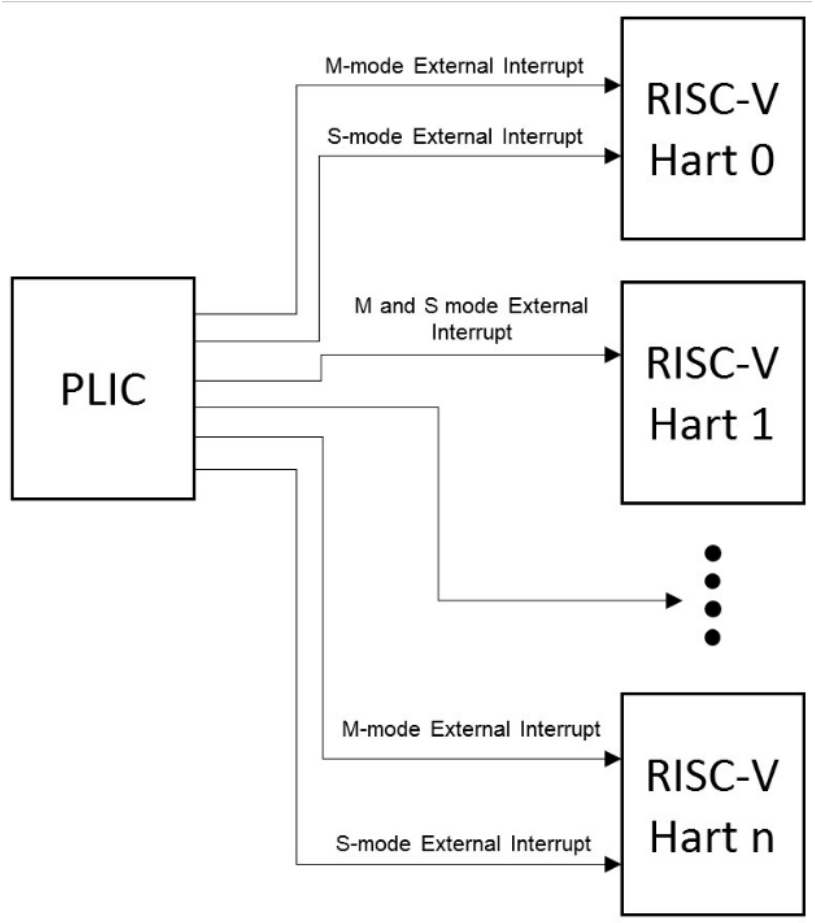


Figure 4.3-3. RISC-V PLIC Interrupt Architecture Block Diagram [14].

4.4 Supervisor Address Translation and Protection (satp) Register

"The satp register is an XLEN-bit read/write register, which controls supervisor-mode address translation and protection. This register holds the physical page number (PPN) of the root page table, i.e., its supervisor physical address divided by 4 KiB; an address space identifier (ASID), which facilitates address-translation fences on a per-address-space basis; and the MODE field, which selects the current address-translation scheme" [13].

31	30 22	21 0
MODE	ASID	PPN

Figure 4.4-1: RV32 Supervisor address translation and protection register **satp** [13].

When MODE is 0 (Bare mode), "supervisor virtual addresses are equal to supervisor physical addresses, and there is no additional memory protection beyond the physical memory protection scheme" [13]. "For RV32, the only other valid setting for MODE is 1, which means **Sv32**, a paged virtual-memory scheme" [13] described later in this chapter.

"The supervisor memory-management fence instruction SFENCE.VMA is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before all subsequent implicit references from that hart to the memory-management data structures" [13]. SFENCE.VMA has the SYSTEM major opcode, func7=7, func3=PRIV, rs2 represents ASID, rs1 is a virtual address. To give a feeling about its behavior note that if rs1=x0 and rs2=x0, the fence orders all reads and writes made to any level of the page tables, for all address spaces. To see the actions taken for the rest of the rs1 and rs2 values, please read subchapter 4.2.1 from [13].

In the following, we describe the **Sv32** Page-Based 32-bit Virtual-Memory Systems. "When **Sv32** virtual memory mode is selected in the MODE field of the satp register, supervisor virtual addresses are translated into supervisor physical addresses via a two-level page table. The 20-bit VPN is translated into a 22-bit physical page number (PPN), while the 12- bit page offset is untranslated. The resulting supervisor-level physical addresses are then checked using any physical memory protection structures, before being directly converted to machine-level physical addresses" [13].

31 22	21 12	11 0
VPN[1]	VPN[0]	Page offset
10	10	12

Figure 4.4-2: **Sv32** virtual address [13].

33 22	21 12	11 0
PPN[1]	PPN[0]	Page offset
12	10	12

Figure 4.4-3: **Sv32** physical address [13].

31 20	19 10	9 8	7	6	5	4	3	2	1	0
PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
12	10	2	1	1	1	1	1	1	1	1

Figure 4.4-4: **Sv32** page table entry [13].

"**Sv32** page tables consist of 2^{10} page-table entries (PTEs), each of four bytes. A page table is exactly the size of a page and must always be aligned to a page boundary. The physical page number of the root page table is stored in the satp register" [13].

"The V bit indicates whether the PTE is valid". "The permission bits, R, W, and X, indicate whether the page is readable, writable, and executable, respectively. When all three are zero, the PTE is a pointer to the next level of the page table; otherwise, it is a leaf PTE" [13].

"Attempting to fetch an instruction from a page that does not have execute permissions raises a fetch page-fault exception. Attempting to execute a load or load-reserved instruction whose effective address lies within a page without read permissions raises a load page-fault exception. Attempting to execute a store, store-conditional (regardless of success), or AMO instruction whose effective address lies within a page without write permissions raises a store page-fault exception" [13].

"The U bit indicates whether the page is accessible to user mode. U-mode software may only access the page when U=1. If the SUM bit in the sstatus register is set, supervisor mode software may also access pages with U=1. However, supervisor code normally operates with the SUM bit clear, in which case, supervisor code will fault on accesses to user-mode pages. Irrespective of SUM, the supervisor may not execute code on pages with U=1" [13].

"The G bit designates a global mapping. Global mappings are those that exist in all address spaces". "The RSW field is reserved for use by supervisor software; the implementation shall ignore this field" [13].

"Each leaf PTE contains an accessed (A) and dirty (D) bit. The A bit indicates the virtual page has been read, written, or fetched from since the last time the A bit was cleared. The D bit indicates the virtual page has been written since the last time the D bit was cleared". "For non-leaf PTEs, the D, A, and U bits are reserved for future standard use and must be cleared by software for forward compatibility" [13]. Two schemes to manage the A and D bits are permitted:

- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, a page-fault exception is raised.
- When a virtual page is accessed and the A bit is clear, or is written and the D bit is clear, the implementation sets the corresponding bit(s) in the PTE. RLSoC implements this option.

"Any level of PTE may be a leaf PTE, so in addition to 4 KiB pages, Sv32 supports 4 MiB megapages. A megapage must be virtually and physically aligned to a 4 MiB boundary" [13].

Next figure show the flow to compute physical address from virtual address in SV32 for two level page table translations (4KiB pages). The translation scheme for 4MiB pages is similar to Fig. 1.2-3.

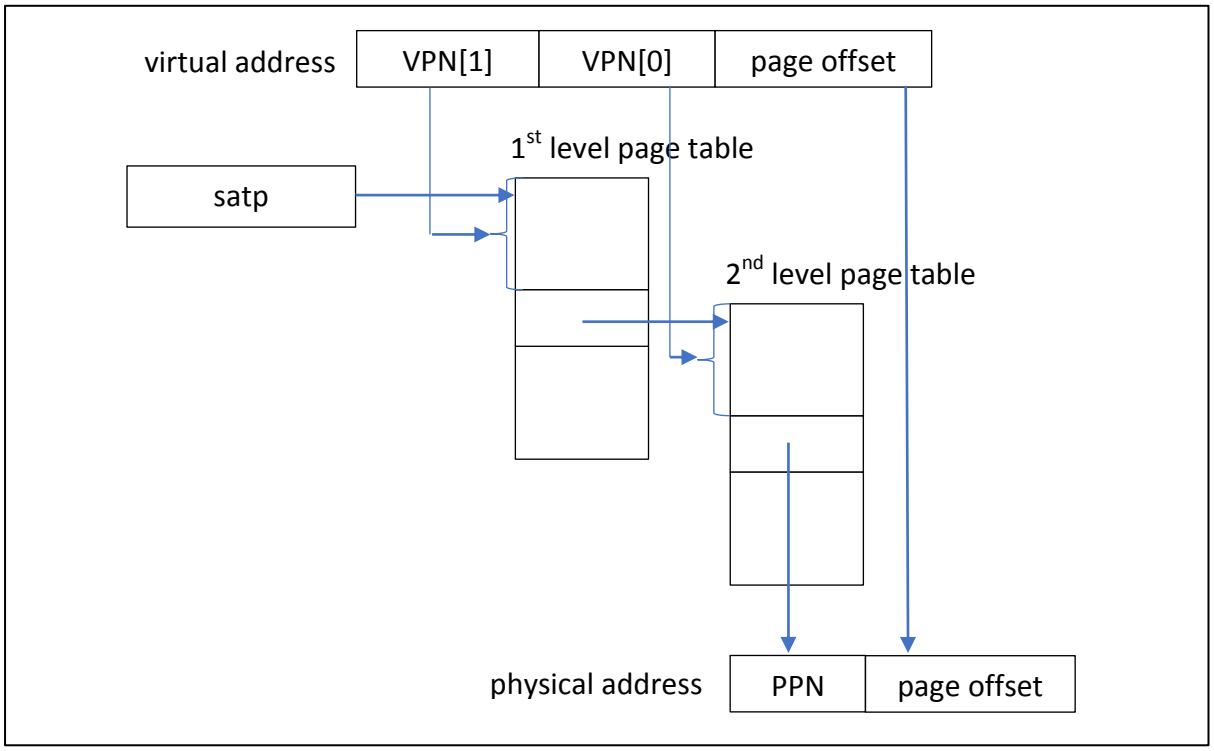


Fig. 4.4-5. Virtual to physical address translation for 4KB pages.

"A virtual address **va** is translated into a physical address **pa** as follows:

1. Let a be satp.ppn × PAGESIZE, and let i = LEVELS − 1. (For Sv32, PAGESIZE=2^{12} and LEVELS=2.)

2. Let pte be the value of the PTE at address a+va.vpn[i]×PTESIZE. (For Sv32, PTESIZE=4.) If accessing pte violates a PMA or PMP check, raise an access-fault exception corresponding to the original access type.

3. If pte.v = 0, or if pte.r = 0 and pte.w = 1, stop and raise a page-fault exception corresponding to the original access type.

4. Otherwise, the PTE is valid. If pte.r = 1 or pte.x = 1, go to step 5. Otherwise, this PTE is a pointer to the next level of the page table. Let i = i − 1. If i < 0, stop and raise a page-fault exception corresponding to the original access type. Otherwise, let a = pte.ppn × PAGESIZE and go to step 2.

5. A leaf PTE has been found. Determine if the requested memory access is allowed by the pte.r, pte.w, pte.x, and pte.u bits, given the current privilege mode and the value of the

SUM and MXR fields of the **mstatus** register. If not, stop and raise a page-fault exception corresponding to the original access type.

6. If i > 0 and pte.ppn[i − 1 : 0] != 0, this is a misaligned superpage; stop and raise a page-fault exception corresponding to the original access type.

7. If pte.a = 0, or if the memory access is a store and pte.d = 0, either raise a page-fault exception corresponding to the original access type, or:

- Set pte.a to 1 and, if the memory access is a store, also set pte.d to 1.
- If this access violates a PMA (physical memory attributes) or PMP (protection) check, raise an access-fault exception corresponding to the original access type.
- This update and the loading of pte in step 2 must be atomic; in particular, no intervening store to the PTE may be perceived to have occurred in-between.

8. The translation is successful. The translated physical address is given as follows:

- pa.pgoff = va.pgoff.
- If i > 0, then this is a superpage translation and pa.ppn[i − 1 : 0] = va.vpn[i − 1 : 0].
- pa.ppn[LEVELS − 1 : i] = pte.ppn[LEVELS − 1 : i] ” [13].

5. RLSoC memory controller

5.1 RLSoC structure

In verilog, the variables which their names are starting with **r_** are of type **reg** or **reg**[] and the variables that start with **w_** are of type **wire** or **wire**[].

The top verilog module used for FPGA implementation is **m_main** from **main.v**. Its structure is depicted in the following figure.

```
module m_main(..)
`ifdef SIM_MAIN
   ddr2_model u_comp_ddr2(..); // for nexys
   ddr3_model u_comp_ddr3(..); // for arty
`endif
   clk_wiz_0 m_clkgen0(..);
   m_mmu c(..);
   m_RVCoreM p(..);
   ..
endmodule
```

Fig. 5.1-1. **m_main** module structure.

m_RVCoreM and **m_mmu** are connected through some common signals.

The structure of the **m_mmu** module is shown in the following figure.

```
module m_mmu(..)

`ifdef SIM_MODE
   read_file rf(..);
`endif

m_tlb TLB_inst_r (..);
m_tlb TLB_data_r (..);
m_tlb TLB_data_w (..);

m_RVuc mc(..);
m_console console(..);
m_disk disk(..);

`ifdef SIM_MODE
   m_dram_sim idbmem(..);
`else
   DRAM_conRV dram_con (..);
`endif

UartTx UartTx0(..);
```

```
PLOADER ploader(..);

endmodule
```

Fig. 5.1-2. **m_mmu** module structure.

5.2 RLSoC initialization

At the very beginning time, the memory is initialized with 0, and, after that, the system waits for the bbl+linux+initramfs file to be sent via the serial line. Let's see how this happens in practice.

In the following listings, the variables may depend on the CLK signal, but we do not show the **always** instructions in order to simplify the listings.

The state of the system at start-up depends on the **r_init_state** variable. This variable is initialized in **mmu.v** as follows:

```
`ifdef SIM_MODE
    reg  [2:0] r_init_state = 5;
`else
    reg  [2:0] r_init_state = 0;
`endif

assign w_init_done = (r_init_state == 5);
```

When the variable **w_init_done** is 0, then the system is in the initialization phase. In this case, the **w_dram_ctrl_t** variable is set to FUNCT3_SW____ which means that when the variable **w_wr_en** is 1, in the RAM memory, will be stored a word of data.

```
wire [2:0]   w_dram_ctrl_t  = (!w_init_done)? `FUNCT3_SW____ : w_dram_ctrl;
`ifdef LAUR_MEM_RB
wire w_wr_en =
                              (r_init_state == 6)? 0 :
                              w_zero_we || w_pl_init_we || w_dram_we_t;
`else
wire w_wr_en =                 w_zero_we || w_pl_init_we || w_dram_we_t;
`endif
```

The **r_zero_done** variable is 0 while the RAM memory is being written with zeroes. As we can see in the below listing, whenever the RAM controller is not busy and the RAM was not fully set to 0, **r_zero_we** is 1.

```
`ifdef SIM_MAIN
        r_zero_we <= 0;
        r_zero_done <= 1;
`else
`ifndef ARTYA7
        if(!w_dram_busy & !r_zero_done) r_zero_we <= 1;
`else
        if(!w_dram_busy & !r_zero_done & calib_done) r_zero_we <= 1;
`endif
        if(r_zero_we) begin
            r_zero_we  <= 0;
            r_zeroaddr <= r_zeroaddr + 4;
        end
        if(r_zeroaddr >= `MEM_SIZE) r_zero_done <= 1;
`endif
```

Let's see now what happens to **r_init_state** variable. We can see from the below listing that r_init_state depends on its previous value and one of**: r_zero_done, r_bbl_done, r_dtree_done, r_disk_done** and **r_mem_rb_done**. Each of these variables becomes 1 when its associated process ends.

```
`ifndef SIM_MODE
    always@(posedge CLK) begin
        r_init_state <= (!RST_X) ? 0 :
                        (r_init_state == 0)                 ? 1 :
                        (r_init_state == 1 & r_zero_done)   ? 2 :
                        (r_init_state == 2 & r_bbl_done)    ? 3 :
                        (r_init_state == 3 & r_dtree_done) ? 4 :
`ifdef LAUR_MEM_RB
                        (r_init_state == 4 & r_disk_done)   ? 6 :
                        (r_init_state == 6 & r_mem_rb_done)  ? 5 :
`else
                        (r_init_state == 4 & r_disk_done)   ? 5 :
`endif
                        r_init_state;
    end
`endif

always@(posedge CLK) begin
if(w_pl_init_we & (r_init_state == 2)) r_initaddr  <= r_initaddr + 4;
if(r_initaddr  >= `BIN_BBL_SIZE)                   r_bbl_done      <= 1;
if(w_pl_init_we & (r_init_state == 3)) r_initaddr3 <= r_initaddr3 + 4;
if(r_initaddr3 >= (`D_INITD_ADDR + `D_SIZE_DEVT))  r_dtree_done    <= 1;
if(w_pl_init_we & (r_init_state == 4)) r_initaddr2 <= r_initaddr2 + 4;
if(r_initaddr2 >= `BBL_SIZE + `BIN_DISK_SIZE)      r_disk_done     <= 1;
end
```

BIN_BBL_SIZE, D_INITD_ADDR, D_SIZE_DEVT, BBL_SIZE and BIN_DISK_SIZE are defined as in the following listing. Please note that SIM_MAIN is defined only when we want to simulate the DDR RAM; for the normal utilization of the system it is disabled. We are especially interested in BIN_BBL_SIZE which is the bbl+linux+initramfs file (eventually appended with zeroes) size. We also are interested in D_INITD_ADDR and D_SIZE_DEVT.

```
`define D_INITD_ADDR    (32*1024*1024)
`define BBL_SIZE        (64*1024*1024)

`ifdef SIM_MAIN
`define BIN_BBL_SIZE   32
`define D_SIZE_DEVT 0
`define BIN_DISK_SIZE 0
`else
`define BIN_BBL_SIZE   (30*1024*1024)
`define D_SIZE_DEVT    (4*1024)
`define BIN_DISK_SIZE 0
`endif
`define BIN_SIZE      (`BIN_BBL_SIZE + `D_SIZE_DEVT + `BIN_DISK_SIZE)
```

The r_initaddr, r_initaddr2, r_initaddr3, r_initaddr6 are the memory addresses where specific data will be written during the corresponding process.

```
wire [31:0]  w_dram_addr_t2 =
                     (r_init_state == 1) ? r_zeroaddr     :
                     (r_init_state == 2) ? r_initaddr     :
`ifdef LAUR_MEM_RB
                    (r_init_state == 6) ? r_initaddr6    :
`endif
                     (r_init_state == 3) ? r_initaddr3    :
                     (r_init_state == 4) ? r_initaddr2    : w_dram_addr_t;

wire [31:0] w_dram_wdata_t = (r_init_state == 1) ? 32'b0 :
                             (r_init_state == 5) ? w_dram_wdata :
                                                   w_pl_init_data;

PLOADER ploader(CLK, RST_X, w_rxd, w_pl_init_addr, w_pl_init_data,
                w_pl_init_we, w_pl_init_done, w_key_we, w_key_data);

DRAM_conRV dram_con (          // user interface ports
                               .i_rd_en(w_dram_le),
                               .i_wr_en(w_wr_en),
                               .i_addr(w_dram_addr_t2),
                               .i_data(w_dram_wdata_t),
                               .o_data(w_dram_odata),
                               .o_busy(w_dram_busy),
                               .i_ctrl(w_dram_ctrl_t),..);
```

We can see that after zeroing the memory, if **r_init_state** != 5, in the memory will be written **w_pl_init_data** which is output of the PLOADER module. PLOADER is implemented in **loader.v**. This file instantiates the **serialc** module which is a classic serial line driver. In PLOADER, the WE variable (which is tied to **w_pl_init_we**) will only be 1 when DATA contains four bytes received from **serialc**. Also in PLOADER, if (waddr >= `BIN_SIZE) then DONE (which is tied to w_pl_init_done) will be set to 1; this means that bbl+linux+initramfs+dtb was successfully received via the serial line. After this, the **serialc** will be used for keyboard input.

Note 5.2-1. In order to have a baudrate of 8Mbps on the serial line, we must define SERIAL_WCNT to 13 for 104 MHz system clock (because 104 MHz / 8 Mbps = 13). This constant is used in the **serialc** module.

5.3. RAM controller

5.3.1 Preliminary considerations

In **mmu.v**, the RAM memory controller is instantiated as in the following listing. **m_dram_sim** is implemented after the **DRAM_conRV** model; the difference is that **m_dram_sim** uses a matrix variable to hold memory data instead of DDR.

```
`ifdef SIM_MODE
    m_dram_sim#(`MEM_SIZE) idbmem(CLK, w_dram_addr_t2, w_dram_odata,
w_dram_we_t, w_dram_le,
                                  w_dram_wdata_t, w_dram_ctrl_t,
w_dram_busy, w_mtime[31:0]);
`else
    DRAM_conRV dram_con (
                                 // user interface ports
                                .i_rd_en(w_dram_le),
                                .i_wr_en(w_wr_en),
                                .i_addr(w_dram_addr_t2),
                                .i_data(w_dram_wdata_t),
                                .o_data(w_dram_odata),
                                .o_busy(w_dram_busy),
                                .i_ctrl(w_dram_ctrl_t),
                                // input clk, rst (active-low)
                                .mig_clk(mig_clk),
                                .mig_rst_x(mig_rst_x),
`ifdef ARTYA7
                                .ref_clk(ref_clk),
`endif
                                // DDR interface ports
                                ..,
                                .o_clk(o_clk),
                                .o_rst_x(o_rst_x),
                                // other
                                .o_init_calib_complete(calib_done)
                                );
`endif
```

DRAM_conRV is implemented in **memory.v**. The memory controller's modules instantiation stack is shown in the following listing.

```
module DRAM_conRV(..)
  state machine ..
  `ifdef SKIP_CACHE
    DRAM_con_witout_cache dram_con_witout_cache (..);
  `else
    cache_ctrl cache_ctrl (..);
  `endif
endmodule

module cache_ctrl(..)
  state machine ..
  DRAM_con_witout_cache dram_con_witout_cache (..);
endmodule
```

```
module DRAM_con_witout_cache(..)
  clk_wiz_1 clkgen1 (from MIG_ui clock to main core clock);
  AsyncFIFO afifo1 (from controller data to MIG);
  AsyncFIFO afifo2 (MIG data to controller);
  DRAMController dc (..);

  state machine ..
endmodule

module DRAMController (..)
  mig_7series_0 mig(DDR2 for nexys or DDR3 for arty interface);
  state machine ..
endmodule
```

The **mig_7series_0** module works with 16 bytes data chunks. When writing data, to DDR, the MIG accepts a 16 bits mask which selects the bytes of the data chunks to be written to memory and which to be skipped. When reading data, the MIG offers 16 bytes as output; that's why the cache line is 16 bytes and **DRAM_conRV** must select, based on the read address, 4 bytes from the 16 offered at MIG output. For memory writing, the most significant 12 bytes of the MIG data are masked (disabled) and only a portion of the first 4 data bytes are written.

5.3.2 DRAM_conRV frontend

The main rule is that **DRAM_conRV** (and other modules too), immediately after a memory request is made (read or write), reports that is busy – until it finishes serving the request.

DRAM_conRV will always send to the backend modules an address which is multiply of 16.

The **DRAM_conRV** output data computation is shown in the following listing.

```
// 16+3 bytes shifted with 0 or 15 bytes.
wire[31:0] w_odata = {r_dram_odata2, r_dram_odata1} >> {r_addr[3:0], 3'b0};
// see if we have LB, LH or LW
assign o_data = (r_ctrl[1:0]==0) ? ((r_ctrl[2]) ? {24'h0, w_odata[7:0]} :
                                     {{24{w_odata[7]}}, w_odata[7:0]}) :
                (r_ctrl[1:0]==1) ? ((r_ctrl[2]) ? {16'h0, w_odata[15:0]} :
                                     {{16{w_odata[15]}}, w_odata[15:0]}) :
                w_odata;
```

In the **DRAM_conRV** state machine, if we have a memory load and the least significant 4 bits of the address are greater than 12, then we must make two 16 byte memory reads: first in **r_dram_odata1** and second in **r_dram_odata2**. That is because DRAM_conRV offers at its output 4 bytes.

At the output, DRAM_conRV must offer byte aligned data; so, in the equation of **w_odata**, we shift {r_dram_odata2, r_dram_odata1} with a multiple of 8 bits.

I said that when writing data, each byte can be masked in order to be written or not. If we have an unaligned memory write request, because DRAM_conRV works with 4 bytes long data, in some cases we need to make two write requests to the backend modules. For example when we have a store word instruction and the least significant two bits of the address are 3, we have first write with r_mask 4'b1000, and second with r_mask 4'b0111 and write data shifted left with 8 bits.

5.3.3 Cache

RLSoC has a direct mapping cache. The cache controller belongs to the **cache_ctrl** module, while the cache memory is implemented in the **m_dram_cache** module. In cache_ctrl, the m_dram_cache module is set up with ADDR_WIDTH of 28 bits, D_WIDTH of 128 bits (16 bytes), and ENTRY of `CACHE_SIZE/16, because a cache line contains 16 bytes. That is why **w_addr** variable of **m_dram_cache** gets the value **c_addr**[31:4] from **cache_ctrl**; this variable is split in {**w_tag**, **w_idx**}.

Please note that **w_flush** and RST_X are disabled in **m_dram_cache** instantiation.

In **m_dram_cache**, **w_maddr** is **w_idx** – the index part of **w_addr,** and **w_mwdata** is {1'b1, **w_tag**, **w_idata**} when **w_we** is 1, and 0, otherwise; these are sent to the **m_bram** module (which is initialized to zero). The **w_modata** output of **m_bram** module is split in {**w_mvalid**, **w_mtag**, **w_mdata**}; **w_oe** is set to (**w_mvalid** && **w_mtag** == **r_tag**), and **w_odata** set to **w_mdata**. When we have a cache hit, **w_oe** is 1; else is 0.

In the cache controller, the state 2'b00 is idle (also the initial cache state), 2'b10 means cache miss and 2'b11 means write to RAM. We have a cache hit when the state is 2'b01 and c_oe is 1. State 2'b10 means cache miss. The system will write to the cache with RAM data when (**r_cache_state** == 2'b10 && !**w_dram_stall**) and the system will clear the cache entry when (**r_cache_state** == 2'b11 && **c_oe**).

The cache controller is busy when **w_dram_stall** is 1 (which means RAM is busy – as a follow up to a RAM read or write command) or **r_cache_state** != 0.

Note 5.3.3-1: The time to boot Linux is 3 times longer when disabling the cache. If you want to disable the cache do the following:

- edit **define.vh** from the **src** folder and enable SKIP_CACHE;

- edit **nexys4.xdc** and **arty35t.xdc** from the **constrs** folder and modify the string **c/dram_con/cache_ctrl/dram_con_witout_cache/clkgen1/inst/mmcm_adv_inst/CLKOUT0** with **c/dram_con/dram_con_witout_cache/clkgen1/inst/mmcm_adv_inst/CLKOUT0**.

- rebuild the vivado projects **nexys4.xpr** or **arty35t.xpr**;

5.3.4 The DRAM_con_witout_cache and DRAMController modules

This module should have been named DRAM_con_without_cache, but a lexicographic mistake was made by the rvsoc authors.

I will explain these modules starting from a simplified version of them which we have created in the **src/dram** folder, namely **dram-no-fifo.v**. In this version, **clk_wiz_1** is not used and the core clock of the rlsoc system is the same as the MIG user interface clock, here named **mig_ui_clk**. Please see note 5.2-1 regarding the uart baudrate set by SERIAL_WCNT.

The **DRAMController** module instantiates the **mig_7series_0** module created in subchapter 2.3. Its most important signals are shown in the table below. To simplify the user logic, **app_wdf_end** and **app_wdf_wren** are tied together in the **mig_7series_0** instantiation, meaning that we have only single write commands.

Signal	Direction	Details
app_addr[]	input	This input indicates the address for the current request.
app_cmd[]	input	This input selects the command for the current request: read or write
app_en	input	This is the active-High strobe for the app_addr[], app_cmd[2:0] and some other not important signals.
app_rdy	output	This output indicates that the UI is ready to accept commands. If the signal is deasserted when app_en is enabled, the current app_cmd and app_addr must be retried until app_rdy is asserted.
app_wdf_data[]	input	This provides the data for write commands.
app_wdf_end	input	This active-High input indicates that the current clock cycle is the last cycle of input data on app_wdf_data[].
app_wdf_mask[]	input	This provides the mask for app_wdf_data[].
app_wdf_rdy	output	This output indicates that the write data FIFO is ready to receive data. Write data is accepted when app_wdf_rdy = 1'b1 and app_wdf_wren = 1'b1.
app_wdf_wren	input	This is the active-High strobe for app_wdf_data[].
app_rd_data[]	output	This provides the output data from read commands.
app_rd_data_valid	output	This active-High output indicates that app_rd_data[] is valid.
ui_clk	output	This UI clock must be a half or quarter of the DRAM clock.
ui_clk_sync_rst	output	This is the active-High UI reset.
init_calib_complete	output	PHY asserts init_calib_complete when calibration is finished.

Table 5.3.4-1. Part of MIG User interface signals [7].

After reset, the **DRAMController** state machine waits for the **init_calib_complete** signal and then enters the idle state. When receives a read or write command, sets up the busy flag and sends the command to the MIG. When de MIG reports that the command is

done via **app_rdy** and **app_wdf_rdy**, the **DRAMController** signals (and **app_rd_data_valid** for read commands) this to the **DRAM_con_witout_cache** module**.**

The **DRAM_con_witout_cache** from **dram-no-fifo.v** is simple. It instantiates the **DRAMController** module and has a straight state machine: if it receives a read or write memory request becomes busy and forwards the request to the **DRAMController**; when this one finishes the request, it will forward the result to the upper control layer and signals not busy.

The **DRAM_con_witout_cache** from **dram-fifo.v** uses **clk_wiz_1** to run the rlsoc system at a higher frequency that the MIG UI clock. It uses two asynchronous FIFOs to communicate with **DRAMController**: **afifo1 AsyncFIFO** for sending data to **DRAMController** and **afifo2 AsyncFIFO** for receiving data from **DRAMController**. Before describing the asynchronous fifo architecture, we must note that **DRAM_con_witout_cache** signals the command to the **DRAMController** as soon as it receives it and does not wait to be finished. It only checks:

- if **afifo1** is full in the case of write commands, and when this happens, waits the **DRAMController** to dequeue a command from **afifo1**;
- if **afifo2** is empty in the case of read commands, and when this happens, waits the **DRAMController** to enqueue the result to **afifo2**.

The **DRAMController** must verify after each “command execution finished” reported by the MIG, if the **DRAM_con_witout_cache** module has enqueued a new command.

Asynchronous FIFOs are used to pass data from one clock domain to the other: here the MIG UI clock and the systems’s core clock generated by **clk_wiz_1**. They use gray code; conversion function from binary code to gray code is a bijective function and vice versa.

The most important thing when working with gray code is that two adjacent values $(v+i)_{gray}$ and $(v+i+1)_{gray}$ differ by a single bit. So, supposing that the faster clock is i times faster than the lower clock, then if the faster pointer transitions from (v+i) to (v+i+1) at the lower speed clock posedge then $(v+i)_{gray}$ and $(v+i+1)_{gray}$ differs by a single bit. So, the lower speed pointer will be $(v+i)_{gray}$ or $(v+i+1)_{gray}$, depending on the value of the bit that differs. If code gray would not be used, then the lower speed pointer may have an invalid value because (v+i) and (v+i+1) may differ by more than 1 bit and some bits will be seen as 0 and others as 1.

Also, we must note that if the **waddr** has wrapped around 1 more time than **raddr2** then the fifo is full. If **waddr2** equals to **raddr**, the fifo is empty.

5.3.5 SIM_MAIN and LAUR_MEM_RB

The SIM_MAIN flag is used when we want just to simulate the RAM memory with its controller to see if it functions correctly. This flag can be used in conjunction with LAUR_MEM_RB flag, which, after rlsoc memory is written at its initialization process – will read back the memory contents. If LAUR_MEM_RB_ONLY_CHECK is defined then the system will print to the verilog simulator console the memory contents. The LAUR_MEM_RB flag also implies computing the checksum of the memory contents.

The last thing to note with LAUR_MEM_RB is that if SIM_MAIN is not defined and if LAUR_MEM_RB_ONLY_CHECK is not defined, the system will send back to the serial line the memory contents until the address (BBL_SIZE + BIN_DISK_SIZE). This address can be changed to the size of a text file if you want to verify that a text file is correctly sent to the memory and read back through the serial line.

Now, we will give a demonstration for the memory address and input data equations used in the verilog code.

```
`ifndef ARTYA7
    assign dram_addr = dout_afifo1_addr[26:1];
    assign dram_din = {{(APP_DATA_WIDTH-32){1'b0}}, dout_afifo1_data};
`else
    assign dram_addr = {2'b0, dout_afifo1_addr[26:4], 3'b0};
    assign dram_din = {4{dout_afifo1_data}};

    wire  [3:0] mask_t = ~data_mask;
    wire [15:0] mask_t2 = mask_t << {dout_afifo1_addr[3:2], 2'b0};
`endif

DRAMController #(..) dc (
`ifndef ARTYA7
        .i_addr({1'b0, dram_addr}),
        .i_mask(data_mask));
`else
        .i_addr(dram_addr),
        .i_mask(~mask_t2));
`endif
```

The Nexys A7 contains one Micron MT47H64M16HR-25:H DDR2 memory component, creating a single rank, 16-bit wide interface [6]. DDR2 MT47H64M16 is 8 Meg x 16 x 8 banks which means 64 Meg x 16 [15]. Refresh count is 8k, row address (a[12:0]) is 8k, bank address (ba[2:0]) is 8 and column address (a[9:0]) is 1k [15]. The address is Rank+Bank+Row+Column = 1+3+13+10 = 27 bits.

The Arty A7 includes one MT41K128M16JT-125 memory component, creating a single rank, 16-bit wide interface [8]. DDR3 MT41J128M16 is 16 Meg x 16 x 8 Banks which means 128 Meg x 16 [16]. Refresh count is 8k, row address (a[13:0]) is 16k, bank address (ba[2:0]) is 8, column address (a[9:0]) is 1k and page size 2KB [16]. The address is Rank+Bank+Row+Column = 1+3+14+10 = 28 bits.

At rlsoc initialization time, if SIM_MAIN is enabled, we will simulate the DDR2 in case of nexys4 or DDR3 for arty35t (see Fig. 5.1-1). The verilog simulation models can be found at [15] for Micron DDR2 and [16] for Micron DDR3.

The simulation consists in sending the memory data contained in the file **src/mem.txt** (which contains 32 bytes) to the memory and then read back. These are the ASCII encoded characters “abcdefghijklmnopponmlkjihgfedcba”; ASCII code for ‘a’ is 8’h61 and for ‘p’ is 8’h70. Recall that when SIM_MAIN is defined, then BIN_BBL_SIZE is 32, D_SIZE_DEVT is 0 and BIN_DISK_SIZE is 0. The simulation can be run with the command **./xsim_run.sh** for nexys and **./xsim_run.sh ARTYA7** for arty, from the **sim** directory. Please note the simulation time in the listings that we show.

Let’s suppose that, for nexys4, the **i_addr** of the **DRAMController** would be **dout_afifo1_addr**[26:0] instead of {1'b0, **dout_afifo1_addr**[26:1]}. In this case, if we simulate using SIM_MAIN, we obtain:

```
m_main.u_comp_ddr2.cmd_task: at time 99238191.0 ps INFO: Precharge All
mem_cnt=       0. sending 61='a'
mem_cnt=       1. sending 62='b'
mem_cnt=       2. sending 63='c'
mem_cnt=       3. sending 64='d'
STATE_IDLE            app_cmd          =          CMD_WRITE,          i_addr=0000000
i_data=00000000000000000000000064636261 i_mask=0
101959730.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000000 data = 6261
101961268.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000001 data = 6463
101962806.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000002 data = 7777
101964345.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000003 data = eeee
101965883.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000004 data = cccc
101967422.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000005 data = 9999
101968960.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000006 data = 2222
101970499.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000007 data = 4444
mem_cnt=       4. sending 65='e'
mem_cnt=       5. sending 66='f'
mem_cnt=       6. sending 67='g'
mem_cnt=       7. sending 68='h'
STATE_IDLE            app_cmd          =          CMD_WRITE,          i_addr=0000004
i_data=00000000000000000000000068676665 i_mask=0
Write    bank 0 col 004, auto precharge 0
102845883.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000004 data = 6665
102847422.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000005 data = 6867
102848960.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000006 data = 2222
102850499.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000007 data = 4444
102852037.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000000 data = 6261
102853576.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000001 data = 6463
102855114.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000002 data = 7777
102856653.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000003 data = eeee
```

So, instead of having 6867 at col 3 and 6665 at col 2, we have them at col 5 and 4. So, i_addr must be 2 instead of 4. In this way, the address must be shifted right with 1 bit. That's why i_addr it is set to {1'b0, **dout_afifo1_addr**[26:1]} for nexys4.

Let's suppose that, for arty35t, we would have **i_addr** of the **DRAMController** **dout_afifo1_addr**[27:0], instead of {2'b0, **dout_afifo1_addr**[26:4], 3'b0} and **dram_din** = **dout_afifo1_data**.

We first send "abcd", "efgh", "ijkl" and "mnop".

```
121644841.0 ps INFO: Precharge All
mem_cnt=       0. sending 61='a'
mem_cnt=       1. sending 62='b'
mem_cnt=       2. sending 63='c'
mem_cnt=       3. sending 64='d'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=0000000
i_data=00000000000000000000000064636261 i_mask=0
124301341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000000 data = 6261
124302841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000001 data = 6463
124304341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000002 data = aaaa
124305841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000003 data = 5555
124307341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000004 data = 5555
124308841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000005 data = aaaa
124310341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000006 data = 9999
124311841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000007 data = 6666
mem_cnt=       4. sending 65='e'
mem_cnt=       5. sending 66='f'
mem_cnt=       6. sending 67='g'
mem_cnt=       7. sending 68='h'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=0000004
i_data=00000000000000000000000068676665 i_mask=0
125189341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000000 data = 6665
125190841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000001 data = 6867
125192341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000002 data = aaaa
125193841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000003 data = 5555
125195341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000004 data = 5555
125196841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000005 data = aaaa
125198341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000006 data = 9999
125199841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000007 data = 6666
mem_cnt=       8. sending 69='i'
mem_cnt=       9. sending 6a='j'
mem_cnt=      10. sending 6b='k'
mem_cnt=      11. sending 6c='l'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=0000008
i_data=00000000000000000000000006c6b6a69 i_mask=0
126077341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000008 data = 6a69
126078841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000009 data = 6c6b
```

```
126080341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000a data = eeee
126081841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000b data = 4444
126083341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000c data = 4444
126084841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000d data = eeee
126086341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000e data = dddd
126087841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000f data = 8888
mem_cnt=      12. sending 6d='m'
mem_cnt=      13. sending 6e='n'
mem_cnt=      14. sending 6f='o'
mem_cnt=      15. sending 70='p'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=000000c
i_data=00000000000000000000000706f6e6d i_mask=0
126965341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000008 data = 6e6d
126966841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000009 data = 706f
126968341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000a data = eeee
126969841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000b data = 4444
126971341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000c data = 4444
126972841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000d data = eeee
126974341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000e data = dddd
126975841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000f data = 8888
```

Now, send “ponmlkjihgfedcba” to the DDR3 module.

```
Mem_cnt=      16. Sending 70=’p’
mem_cnt=      17. sending 6f='o'
mem_cnt=      18. sending 6e='n'
mem_cnt=      19. sending 6d='m'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=0000010
i_data=00000000000000000000000006d6e6f70 i_mask=0
127841341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000010 data = 6f70
127842841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000011 data = 6d6e
127844341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000012 data = eeee
127845841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000013 data = 4444
127847341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000014 data = 4444
127848841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000015 data = eeee
127850341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000016 data = dddd
127851841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000017 data = 8888
mem_cnt=      20. sending 6c='l'
mem_cnt=      21. sending 6b='k'
mem_cnt=      22. sending 6a='j'
mem_cnt=      23. sending 69='i'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=0000014
i_data=00000000000000000000000696a6b6c i_mask=0
128729341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000010 data = 6b6c
128730841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000011 data = 696a
128732341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000012 data = eeee
128733841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000013 data = 4444
```

```
128735341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000014 data = 4444
128736841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000015 data = eeee
128738341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000016 data = dddd
128739841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000017 data = 8888
mem_cnt=      24. sending 68='h'
mem_cnt=      25. sending 67='g'
mem_cnt=      26. sending 66='f'
mem_cnt=      27. sending 65='e'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=0000018
i_data=00000000000000000000000065666768 i_mask=0
129617341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000018 data = 6768
129618841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000019 data = 6566
129620341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001a data = eeee
129621841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001b data = 4444
129623341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001c data = 4444
129624841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001d data = eeee
129626341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001e data = dddd
129627841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001f data = 8888
mem_cnt=      28. sending 64='d'
mem_cnt=      29. sending 63='c'
mem_cnt=      30. sending 62='b'
mem_cnt=      31. sending 61='a'
STATE_IDLE          app_cmd          =          CMD_WRITE,          i_addr=000001c
i_data=00000000000000000000000061626364 i_mask=0
130637341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000018 data = 6364
130638841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000019 data = 6162
130640341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001a data = eeee
130641841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001b data = 4444
130643341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001c data = 4444
130644841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001d data = eeee
130646341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001e data = dddd
130647841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000001f data = 8888
```

The MIG works with 128 bits of data which can be masked. Clearly, the least significant 3 bits of the address are not considered when writing the first 32 bits.

Let's suppose now that we have the rlsoc used equations. Then it will be written the four octets of **dout_afifo1_data** to address (**i_addr**) specified as {**dout_afifo1_addr**[26:4], 3'b0} + ({**dout_afifo1_addr**[3:2], 2'b0} / 2), because this DDR3 memory works with 16 bits pieces of data. To verify this, consider the following.

At the begining, after "abcd" it follows "efgh" at address **dout_afifo1_addr**=4. This address implies **i_addr**=0 and **i_mask**=16'hff0f. Data is **dout_afifo1_data=**{4{32'h68676665}}. So, from col=0 we skip 4 bytes (32 bits) and we arrive

at col=2 and col=3, because this DDR3 memory works with 16 bits pieces of data. So, after “abcdefghijklmnop” we have:

```
126965341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000000 data = 6261
126966841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000001 data = 6463
126968341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000002 data = 6665
126969841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000003 data = 6867
126971341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000004 data = 6a69
126972841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000005 data = 6c6b
126974341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000006 data = 6e6d
126975841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000007 data = 706f
```

The “ponmlkjihgfedcba” ends with “dcba” which means 32’h61626364 at **dout_afifo1_addr**=1c. The values passed to the DRAMController are **i_addr**=0000008, **i_data**=61626364616263646162636461626364 and **i_mask**=0fff. So, from column 8 we skip 12 bytes (3*32 bits) and we arrive at col=0e and col=0f, because this DDR3 memory works with 16 bits pieces of data.

```
130637341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000008 data = 6f70
130638841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 00000009 data = 6d6e
130640341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000a data = 6b6c
130641841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000b data = 696a
130643341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000c data = 6768
130644841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000d data = 6566
130646341.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000e data = 6364
130647841.0 ps INFO: WRITE @ DQS= bank = 0 row = 0000 col = 0000000f data = 6162
```

6. Berkeley Boot Loader (BBL)

6.1 Preliminary considerations

In order to boot the Linux kernel, the system needs to use a bootloader called the Berkeley Boot Loader, or BBL. Shortly, this program runs in machine mode and sets up the trap table, reads the device tree pointed by the a1 register; finally it launches Linux in supervisor mode, which receives in a0 the hart ID which will boot first. Linux detects hardware via the open firmware, sets up virtual memory, loads the kernel modules and starts the init process.

In the **m_regfile** of the **m_RVCoreM** verilog module, mem[11], which represents register a1, is set as `D_INITD_ADDR + `D_START_PC; for rlsoc is (32*1024*1024) + 32'h80000000, when SIM_MAIN is not enabled, and represents the dtb memory address. Also, all other registers are set to 0, so register a0 is set to 0 which represents the hart id that will boot first.

The BBL code is contained in **riscv-pk** folder and is written in the C and assembler language.

In machine/**mcall.h** we have the following defines: #define SBI_SET_TIMER 0, SBI_CONSOLE_PUTCHAR 1, SBI_CONSOLE_GETCHAR 2, SBI_CLEAR_IPI 3, SBI_SEND_IPI 4, SBI_REMOTE_FENCE_I 5, BI_REMOTE_SFENCE_VMA 6, SBI_REMOTE_SFENCE_VMA_ASID 7, SBI_SHUTDOWN 8.

In machine/**mtrap.c** we have the **mcall_trap**() handler which has many functionalities. Here we present the code which prints a character on the system serial console; important is that it uses the **simrv_putc**() function which will be explained later, in the console chapter.

```
#define CMD_PRINT_CHAR 1
volatile int *TOHOST_ADDR = (int *)0x40008000;
void simrv_putc (char c) {
  *TOHOST_ADDR = CMD_PRINT_CHAR << 16 | c;
} ..
static uintptr_t mcall_console_putchar(uint8_t ch) {
  simrv_putc(ch);
  if (uart) uart_putchar(ch); else if (uart16550) uart16550_putchar(ch); else if (htif)
htif_console_putchar(ch); return 0;
}
void mcall_trap(uintptr_t* regs, uintptr_t mcause, uintptr_t mepc) {
  write_csr(mepc, mepc + 4);
  uintptr_t n = regs[17], arg0 = regs[10], arg1 = regs[11], retval, ipi_type;
  switch (n) {
    case SBI_CONSOLE_PUTCHAR:
```

```
            retval = mcall_console_putchar(arg0);
    .. case SBI_SEND_IPI .. case SBI_REMOTE_SFENCE_VMA .. case SBI_REMOTE_FENCE_I ..
```

6.2 mentry.S

To see the BBL entry point, we use the command listed below.

```
riscv32-buildroot-linux-gnu-objdump -d -t -r riscv-pk/build/bbl | less
Disassembly of section .text:
80000000 <_ftext>:
80000000:	2000006f          	j	80000200 <do_reset>
```

We see that the **.text** segment starts at 80000000 and jumps to the **do_reset** function. This function is in the machine/**mentry.S** assembler file. This file starts with the **trap_table** label which contains pointers to **bad_trap, pmp_trap, mcall_trap**, etc. **mcall_trap** is called in bbl mentry.S on mcause=9 or 11 which are ECALL from S_MODE or M_MODE (see the label .Lhandle_trap_in_machine_mode). Note that the **.dc** directive expects expressions separated by commas; the values of the expressions are inserted into the current section. The **.a** suffix emits values of the size of an address on the target system.

A thing to note in assembler is that "to define a local label, write a label of the form `N:' (where N represents any positive integer). To refer to the most recent previous definition of that label write `Nb', using the same number as when you defined the label. To refer to the next definition of a local label, write `Nf'—the `b' stands for "backwards" and the `f' stands for "forwards"" [17].

The **do_reset** function does the following:

- sets to zero the registers from x1 to x31 without x10, x11 which are a0, a1;
- csrw mscratch, x0; mscratch it is used to hold a pointer to a machine-mode hart-local context space and swapped with a user register upon entry to an M-mode trap handler;
- csrw mtvec, trap_vector; trap_vector is a label and we will see it soon;
- la sp, stacks + RISCV_PGSIZE - MENTRY_FRAME_SIZE; sp ← stacks + 4KB - (32*4 + 8*32 + 64); stacks is a label defined at the end as "stacks: .skip RISCV_PGSIZE * MAX_HARTS";
- csrr a3, mhartid; add sp, sp, a3 << RISCV_PGSHIFT
- beqz a3, init_first_hart; boots on the first hart;
- .LmultiHart: # if MAX_HARTS > 1 wait for an IPI (using wfi) to signal that it's safe to boot (calls init_other_hart).

Now we have **mtvec** set to the **trap_vector** label, so we have direct mode. Let's see what the trap_vector code does to handle traps:

- csrrw sp, mscratch, sp /* swap mscratch and sp */
- beqz sp, .Ltrap_from_machine_mode /* if sp is 0, jump to label */
- Store a0, a1 to 10/11*REGBYTES(sp);

- csrr a1, mcause; bgez a1, .Lhandle_trap_in_machine_mode
- This is an interrupt because mcause is < 0, so it has has `CAUSE_INTERRUPT=1<<31 set and this happens for interrupts;
- Is it a machine timer interrupt? -> Simply clear MTIE and raise STIP without CAUSE_INTERRUPT so next time, pc<=stvec.
- .Lmret: # Go back whence we came.
  - restore a0,a1, .. csrrw sp, mscratch, sp;
  - mret # restores state: PC←MEPC, privilege←mstatus.MPP, MIE←mstatus.MPIE
- If it was not a machine timer interrupt then check if is it and IPI? Verify IPI_SOFT, IPI_FENCE_I, IPI_SFENCE_VMA, IPI_HALT; finally: j Lmret.

Now, let's see the code from .Ltrap_from_machine_mode:

- csrr sp, mscratch; addi sp, sp, -INTEGER_CONTEXT_SIZE; store a0,a1
- li a1, TRAP_FROM_MACHINE_MODE_VECTOR /* 13, __trap_from_machine_mode */
- j .Lhandle_trap_in_machine_mode

The code from .Lhandle_trap_in_machine_mode uses:

- the instruction **auipc** which adds a 20-bit upper immediate (the least significant 12 bits are filled to 0 in order to form a 32bit value) to the PC and stores the sum to the destination register.
- the %lo(symbol) and %hi(symbol) which represent the low 12 bits and respective the high 20 bits of absolute address for symbol.
- %pcrel_lo(symbol) and %pcrel_hi(symbol) which represent the low 12 bits and respective the high 20 bits of relative address between pc and symbol (symbol-pc).
- for RISC-V 32, LOG_REGBYTES is 2 and REGBYTES is (1 << LOG_REGBYTES)

This code does the following:

- store registers
- t1 ← trap_table[mcause] /* a1 is mcause or an index to trap_table at the time of the jump to Lhandle_trap_in_machine_mode; the instruction LOAD t1, %pcrel_lo(1b)(t1) uses the least significant 12 bits of the trap_table address because %pcrel_lo is used in conjunction with %pcrel_hi */;
- mv a0, sp; /* a0 ← regs */
- a2 ← mepc; When a trap is taken into M-mode, mepc is written with the virtual address of the instruction that was interrupted or that encountered the exception
- csrrw t0, mscratch, x0; t0 ← user sp
- jalr t1 # Invoke the handler.
- csrw mscratch, sp /* Restore mscratch, so future traps will know they didn't come from M-mode*/;
- restore registers
- mret

6.3 init_first_hart()

This BBL function continues the system initialization. It heavily uses the dtb file to extract hardware info from it.

At its beginning, **init_first_hart()** calls query_uart(dtb), query_uart16550(dtb), and query_htif(dtb), which are not used in rlsoc, although query_htif() is used in tinyemu.

Then the **hart_init()** function is called. It contents are shown in the next listing. The fp_init() function is for floating point initialization and will not be detailed.

```
static void hart_init()
{
  mstatus_init();
  fp_init();
#ifndef BBL_BOOT_MACHINE
  delegate_traps();
#endif /* BBL_BOOT_MACHINE */
  setup_pmp();
}
```

The main things that mstatus_init() does are the following:

- /* Enable software interrupts */ write_csr(mie, MIP_MSIP);
- /* Disable paging if misa reg supports S mode */ if (supports_extension('S')) write_csr(sptbr /* now is named satp */, 0);

The delegate_traps() function will be called because BBL_BOOT_MACHINE is not defined and it does the following:

```
delegate_traps() {
  // send S-mode interrupts and most exceptions straight to S-mode
  if (!supports_extension('S')) return;
          uintptr_t interrupts = MIP_SSIP | MIP_STIP | MIP_SEIP;
          uintptr_t exceptions =
    (1U << CAUSE_FETCH_PAGE_FAULT) |
    (1U << CAUSE_BREAKPOINT) |
    (1U << CAUSE_LOAD_PAGE_FAULT) |
    (1U << CAUSE_STORE_PAGE_FAULT) |
    (1U << CAUSE_USER_ECALL);
  write_csr(mideleg, interrupts);
  write_csr(medeleg, exceptions);
}
```

Finally hart_init() calls setup_pmp(). The PMP defines a finite number of PMP regions which can be individually configured to enforce access permissions to a range of addresses in memory. It sets up a PMP to permit access to all of memory and ignores the illegal-instruction trap if PMPs aren't supported. It selects naturally aligned power-of-2 regions (NAPOT). The NAPOT range selected is 2^(XLEN+3) bytes because pmpaddr0=-1=1..1 and pmpcfg0=PMP_NAPOT | PMP_R | PMP_W | PMP_X.

Then follows the **hls_init(0)** function. HLS is the hart's local storage and is a block of memory at the top of the stack.

```
typedef struct {
  volatile uint32_t* ipi;
  volatile int mipi_pending;

  volatile uint64_t* timecmp;

  volatile uint32_t* plic_m_thresh;
  volatile uintptr_t* plic_m_ie;
  volatile uint32_t* plic_s_thresh;
  volatile uintptr_t* plic_s_ie;
} hls_t;
```

```
hls_t* hls_init(uintptr_t id)
{
  hls_t* hls = OTHER_HLS(id);
  memset(hls, 0, sizeof(*hls));
  return hls;
}

#define MACHINE_STACK_TOP() ({ \
  register uintptr_t sp asm ("sp"); \
  (void*)((sp + RISCV_PGSIZE /*4K*/) & -RISCV_PGSIZE);
}
```

```
#define HLS() ((hls_t*)(MACHINE_STACK_TOP() - HLS_SIZE/*64*/))

#define OTHER_HLS(id)  \
            ((hls_t*)((void*)HLS() + RISCV_PGSIZE * ((id) - read_const_csr(mhartid))))
```

Note: the ipi field of hts_t is initialized for each core to a clint related address.

```
volatile uintptr_t clint_ipi_base=0x60000000;
static void clint_done(const struct fdt_scan_node *node, void *extra)
{
  struct clint_scan *scan = (struct clint_scan *)extra; ..
  clint_ipi_base = scan->reg;

  for (int index = 0; end - value > 0; ++index) { ..
    if (hart < MAX_HARTS) {
      hls_t *hls = OTHER_HLS(hart);
      hls->ipi = (void*)((uintptr_t)scan->reg + index * 4);
```

Then it calls **query_finisher(dtb)** commented as "find the power button early as well so die() works". It commands a search in dtb to find a match between "compatible" and "sifive,test0", which is false in rlsoc.

The **query_mem(dtb)** call containes the following code:

```
  struct fdt_cb cb; struct mem_scan scan;
  cb.open = mem_open; cb.prop = mem_prop; cb.done = mem_done; cb.extra = &scan;
```

```
 mem_size /* global */ = 0;
 fdt_scan(fdt, &cb); /* scans dtb field device_type = “memory” and takes the values from the field reg = <0x0 0x80000000 0x0 0x4000000> for rlsoc. These are start address and size which means 64MB for rlsoc. */
```

The **query_harts(dtb)** function searches in dtb a match like device_type=”cpu” and extracts info like "interrupt-controller", "#interrupt-cells", etc.

The **query_clint(dtb)** does a similar search for clint (Core-Local Interruptor) and sets **mtime** to (void*)((uintptr_t)scan->reg + 0xbff8) where scan->reg is 0x60000000 for rlsoc. Also sets hls->ipi=0x60000000 and hls->timecmp=0x60004000 for hart 0 for rlsoc.

The **query_plic(dtb)** does a similar job for rlsoc, but this is for PLIC:

```
plic_priorities = (uint32_t*)(uintptr_t)scan->reg;
plic_ndevs = scan->ndev; /* = 31 = 1f */
PLIC: prio 50000000 devs 31
interrupts-extended = <0x1 0x9 0x1 0xb>; /* IRQ_S_EXT=9, IRQ_M_EXT=11 */
PLIC scan->reg=50000000 cpu_int=00000009
PLIC scan->reg=50000000 cpu_int=0000000b
hls->plic_m_ie, hls->plic_m_thresh, hls->plic_s_ie, hls->plic_s_thresh:
CPU 0: 50002080 50201000 50002000 50200000
CPU 1-7: all 0
```

The **query_chosen(dtb)** searches in the node “chosen” for kernel-start and kernel-end subnodes. RLSoC dtb does not define these, but tinyemu does: riscv,kernel-start = <0x00 0x80400000>.

wake_harts() does what its name means: it wakes up the risc-v harts.

```
wake_harts() {
   for (int hart = 0; hart < MAX_HARTS; ++hart)
         if ((((~disabled_hart_mask & hart_mask) >> hart) & 1))
               *OTHER_HLS(hart)->ipi = 1; // wakeup the hart
}
```

plic_init() sets plic_priorities[1:31] to 1.

hart_plic_init() does the following:

```
hart_plic_init() {
  // clear pending interrupts
 *HLS()->ipi = 0;  *HLS()->timecmp = -1ULL;  write_csr(mip, 0);
 size_t ie_words = (plic_ndevs + 8 * sizeof(uintptr_t) - 1) / (8 * sizeof(uintptr_t));
 for (size_t i = 0; i < ie_words; i++)
      HLS()->plic_s_ie[i] = ULONG_MAX; // Supervisor not always present
```

```
  *HLS()->plic_m_thresh = 1;
  *HLS()->plic_s_thresh = 0;      // Supervisor not always present
}
```

memory_init() sets mem_size to (mem_size / MEGAPAGE_SIZE * MEGAPAGE_SIZE).

The **boot_loader()** function does the following:

- copies the dtb after the payload (after linux, and it won't touch the original dtb, because for linux 5.13.19 it will place it at address 28MB and in my version the original dtb is placed at 32MB);
- filters some information from the copied dtb;
- prints on the screen the risc-v logo and the filtered dtb;
- sets the **entry_point** to kernel_start (if not null) or _payload_start (if linux is appended to the bbl executable, and it is);
- finally calls **boot_other_hart**(0).

6.4 boot_other_hart()

A very important function is boot_other_hart() and it does the following:

- sets the **entry** variable to **entry_point;**
- sets the **hartid** variable to the mhartid csr value;
- calls **protect_memory**();
- calls **enter_supervisor_mode**(entry, hartid, dtb_output());

The protect_memory() function simply calls setup_pmp(), which we discussed earlier, because rlsoc does not have pmp registers.

The **enter_supervisor_mode()** function code is shown in the following listing. Here:

- REGBYTES is 4;
- MENTRY_HLS_OFFSET is INTEGER_CONTEXT_SIZE (which is 32 * REGBYTES), because SOFT_FLOAT_CONTEXT_SIZE is 0;
- HLS_SIZE is 64;
- MENTRY_FRAME_SIZE is MENTRY_HLS_OFFSET + HLS_SIZE;
- MACHINE_STACK_TOP was described previously.

```
enter_supervisor_mode(fn /*=entry_point*/, arg0 /*=hartid*/, arg1 /*=dtb_output()*/)
{
  mstatus = INSERT_FIELD(mstatus, MSTATUS_MPP, PRV_S);
  mstatus = INSERT_FIELD(mstatus, MSTATUS_MPIE, 0);
  write_csr(mscratch, MACHINE_STACK_TOP() - MENTRY_FRAME_SIZE);
#ifndef __riscv_flen
  uintptr_t *p_fcsr = MACHINE_STACK_TOP() - MENTRY_FRAME_SIZE; // the x0's save slot
  *p_fcsr = 0;
#endif
  write_csr(mepc, fn);
  register uintptr_t a0 asm ("a0") = arg0; register uintptr_t a1 asm ("a1") = arg1;
  asm volatile ("mret" : : "r" (a0), "r" (a1));
}
```

MRET determines what the new operating mode will be according to MPP in **mstatus**, so it will be S-mode. MRET then sets in **mstatus**, MPP=0, MIE=MPIE (here 0), and MPIE=1. Lastly, MRET sets the privilege modes as previously determined, and sets PC to MEPC, which is _payload_start (0x80400000) - the linux kernel start address.

7. Linux, RLSoC and TinyEMU

7.1 Preliminary considerations

Note 7.1-1: we use pseudocode to list functions contents with annotated comments. In the pseudocode there may be present various constructions as:

- f()=g(), means that function f() is identical with a g() call;

- f()→g(), means that function f() calls function g();

- **f(){ g() {..} }**, means that the body of the called function g() is shown next to its call;

- f(a=b) means that function f() is called with parameter **a** having the value of **b**;

To see the linux early boot messages, in dts, we must provide to the linux kernel command line the "earlycon=sbi" option, and in the linux config we enabled CONFIG_SERIAL_EARLYCON_RISCV_SBI; also we must enable CONFIG_RISCV_SBI and CONFIG_RISCV_SBI_V01 to be able to use SBI calls in S-mode.

We have seen that in bbl we can use the **simrv_putc**() function to write to the rlsoc system console. Via the bbl's mcall_trap() function we can call simrv_putc(). In linux 5.13.19, we have the function sbi_console_putchar() that prints a character to the console, by triggering the SBI via the **ecall** risc-v instruction.

```
struct sbiret sbi_ecall(int ext, int fid, unsigned long arg0,
                unsigned long arg1, unsigned long arg2,
                unsigned long arg3, unsigned long arg4,
                unsigned long arg5)
{
    struct sbiret ret;
    register uintptr_t a0 asm ("a0") = (uintptr_t)(arg0);
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg1);
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg2);
    register uintptr_t a3 asm ("a3") = (uintptr_t)(arg3);
    register uintptr_t a4 asm ("a4") = (uintptr_t)(arg4);
    register uintptr_t a5 asm ("a5") = (uintptr_t)(arg5);
    register uintptr_t a6 asm ("a6") = (uintptr_t)(fid);
    register uintptr_t a7 asm ("a7") = (uintptr_t)(ext);
    asm volatile ("ecall"
            : "+r" (a0), "+r" (a1)
            : "r" (a2), "r" (a3), "r" (a4), "r" (a5), "r" (a6), "r" (a7)
            : "memory");
    ret.error = a0;
    ret.value = a1;
```

```
	return ret;
}

void sbi_console_putchar(int ch)
{
	sbi_ecall(SBI_EXT_0_1_CONSOLE_PUTCHAR, 0, ch, 0, 0, 0, 0, 0);
}
```

Via sbi_console_putchar() we can easily build a function that prints a string and a number in hexadecimal. We have used such functions to print the variable names and values on the screen. Here they are listed.

```
void pr_laur(char *s, unsigned long n)
{
	int i=0;
	unsigned char c, v[8];

	if(s)
		while (s[i])
			sbi_console_putchar(s[i++]);
	sbi_console_putchar(' ');
	for(i = 0; i < 8; i++) {
		c = n & 0xf;
		if(c < 10) v[i] = '0'+c;
		else	v[i] = 'a'-10+c;
		n = n >> 4;
	}

	for(i = 7; i >= 0; i--)
			sbi_console_putchar(v[i]);
	sbi_console_putchar('\n');
} //  pr_laur

void pr_laur_regs()
{
	register int t3 asm ("t3");
	register int a1 asm ("a1");
	pr_laur("pr_laur_args a1=", a1);
	pr_laur("pr_laur_args t3=", t3);
}
```

7.2 head.S

We use the compiler to append the linux kernel to the bbl. Linux receives from the bbl the hartid to boot from and the device tree memory address. Its starting point the **_start** entry is implemented in the arch/riscv/kernel/**head.S**. Here, linux makes the switch from running without using the MMU to using the MMU.

By using the following command we can inspect the vmlinux assembly code:

riscv32-buildroot-linux-gnu-objdump -d -t -r vmlinux | less

In the following, we explain a shortened resume of head.S, by using code comments. Remember that linux runs in S-mode.

```
__HEAD
ENTRY(_start)
   j _start_kernel
ENTRY(_start_kernel)
/* Mask all interrupts */
csrw CSR_IE, zero
csrw CSR_IP, zero
/* Load the global pointer */
.option push // saves the current options so that they can be later restored
.option norelax // do not relax AUIPC+LW/SW to a gp relative reference (constant pool). The load of gp obviously can't be relaxed and is fully qualified.
   la gp, __global_pointer$
.option pop
```

Please note that, even if in the vmlinux objdump, we have **c14889c0 g .sdata 00000000 __global_pointer$**, because linux is appended to bbl, if we now print the value of gp will be something like 0x81888b18.

Now linux makes a "hart_lottery" to select a hart to boot linux (the hart which first increment the **hart_lottery** variable will boot the other cores). Because rlsoc has one single hart, this one will be chosen to boot. So we can skip this code.

Then, **_start_kernel** clears BSS for flat non-ELF images and saves the hart ID and DTB physical address:

```
..
/* Save hart ID and DTB physical address */
mv s0, a0;     mv s1, a1
la a2, boot_cpu_hartid // unsigned long boot_cpu_hartid
XIP_FIXUP_OFFSET a2 // .macro XIP_FIXUP_OFFSET reg .endm
REG_S a0, (a2) // sw a0, (a2)
```

After that, linux initializes the page tables and relocates to virtual addresses. Let's see the code and then we will analyze each section:

```
la sp, init_thread_union + THREAD_SIZE
mv a0, s1 // dtb address, because CONFIG_BUILTIN_DTB is not defined
call setup_vm

// CONFIG_MMU is defined, so:
la a0, early_pg_dir
XIP_FIXUP_OFFSET a0
call relocate

call setup_trap_vector

/* Restore C environment */
la tp, init_task # thread pointer struct task_struct init_task;
sw zero, TASK_TI_CPU(tp) // #define TASK_TI_CPU 20 /* offsetof(struct task_struct, thread_info.cpu) */
la sp, init_thread_union + THREAD_SIZE/*8192*/

/* Start the kernel */
call soc_early_init // verified and it does nothing in rlsoc
tail start_kernel
```

Listing 7.2-1. Initialize the page tables and relocate to virtual addresses

In the **la sp, init_thread_union + THREAD_SIZE** instruction:

- THREAD_SIZE is (PAGE_SIZE<<THREAD_SIZE_ORDER), so is 8192;

- in include/linux/sched/**task.h** we have: **extern union thread_union init_thread_union**;

- **init_thread_union** is the address of the top of the stack, and is defined in the **.data** section in arch/riscv/kernel/vmlinux.lds;

- the **thread_union** union is defined as:

```
union thread_union {
        #ifndef CONFIG_ARCH_TASK_STRUCT_ON_STACK // not defined, so taken
                struct task_struct task;
        #endif
        #ifndef CONFIG_THREAD_INFO_IN_TASK // defined, so not taken
        struct thread_info thread_info;
        #endif
        unsigned long stack[THREAD_SIZE/sizeof(long)]; //=8192/4;
};
```

The **setup_vm()** function is defined in arch/riscv/kernel/setup.c, and, mainly, its contents are listed below. It uses the function create_pgd_mapping(.., sz = PGDIR_SIZE, ..) which significantly simplifies its job: it sets up **early_pg_dir** and **trampoline_pg_dir** which will be used to make the switch from physical to virtual addresses.

```
asmlinkage void __init setup_vm(uintptr_t dtb_pa=81c00000) {
        // __PAGETABLE_PMD_FOLDED defined
        load_pa = (uintptr_t)(&_start);//_start= 80400000;
        load_sz = (uintptr_t)(&_end) - load_pa; // 818c2000 - _start = 014c2000
        va_pa_offset = PAGE_OFFSET - load_pa; // 0xc0000000 - pa
        pfn_base = PFN_DOWN(load_pa); //((pa) >> PAGE_SHIFT/*=12*/)
        /*
         * Enforce boot alignment requirements of RV32 and
         * RV64 by only allowing PMD or PGD mappings.
         */
        map_size = PMD_SIZE; // 1<<PMD_SHIFT=PUD_SHIFT=P4D_SHIFT
=PGDIR_SHIFT=22; so map_size is (1 << 22)
        BUG_ON((PAGE_OFFSET % PGDIR_SIZE) != 0); // % (1<<22)
        BUG_ON((load_pa % map_size) != 0);


        pt_ops.alloc_pte = alloc_pte_early; // if it is called, is BUG();
        pt_ops.get_pte_virt = get_pte_virt_early; // inline pte_t *__init get_pte_virt_early(
phys_addr_t pa) { return (pte_t *)((uintptr_t)pa); }
        /* Setup early PGD for fixmap. The corresponding physical address of a fix-mapped
linear address can be set up arbitrarily. A fix-mapped address maps a page frame and it is
treated as a constant pointer. */
        pgd_t early_pg_dir[PTRS_PER_PGD/sizeof(pgd_t) ] __initdata __aligned(PAGE_SIZE);
/* =PAGE_SIZE/4=1024 entries */
        pte_t fixmap_pte[PTRS_PER_PTE  / sizeof(pte_t)]  __page_aligned_bss; // 1024
entries.
        create_pgd_mapping(pgd_t *pgdp=early_pg_dir, uintptr_t va=FIXADDR_START,
phys_addr_t pa =fixmap_pgd_next=fixmap_pte, phys_addr_t sz=PGDIR_SIZE=1<<22,
pgprot_t prot=PAGE_TABLE=__pgprot(_PAGE_TABLE=_PAGE_PRESENT)=(pgprot_t)1) {
                uintptr_t pgd_idx = pgd_index(va); /*= (((va) >> PGDIR_SHIFT=22) &
((PTRS_PER_PGD=1024) - 1))=0x277*/
                if (sz == PGDIR_SIZE) {
                        if (pgd_val(pgdp[pgd_idx])/*=pgdp[pgd_idx].pgd*/ == 0)
                                pgdp[pgd_idx] = pfn_pgd(PFN_DOWN(pa) =pa >> PAGE_SHIFT
=pa>>12, prot); //=__pgd((pfn << _PAGE_PFN_SHIFT=10) | pgprot_val(prot)); = 0x20622801
                                        /* #define __pgd(x)    ((pgd_t) { (x) }) */
                        return;
                }..
        }

        /* Setup trampoline PGD */
        pgd_t swapper_pg_dir[PTRS_PER_PGD] __page_aligned_bss;
        pgd_t trampoline_pg_dir[PTRS_PER_PGD] __initdata __aligned(PAGE_SIZE);
```

```
        create_pgd_mapping(trampoline_pg_dir,
kernel_virt_addr=KERNEL_LINK_ADDR=PAGE_OFFSET, load_pa, PGDIR_SIZE,
PAGE_KERNEL_EXEC = _PAGE_READ | _PAGE_WRITE | _PAGE_PRESENT | _PAGE_ACCESSED
| _PAGE_DIRTY | _PAGE_EXEC);
        /* pgd_idx= 00000300== (0xc0000000 >> 22) & 0x3ff;
trampoline_pg_dir[0x300=768])= pfn_pgd(PFN_DOWN(pa)=pa >> PAGE_SHIFT=12, prot); =
pfn_pgd(0x80400000 >> 12, prot) = (pgd_t) (0x80400000 >> 2 | pgprot_val(prot)) =
201000cf.
        */


        /*
         * Setup early PGD covering entire kernel which will allow
         * us to reach paging_init(). We map all memory banks later
         * in setup_vm_final() below.
         */
        create_kernel_page_table(pg_dir = early_pg_dir,  map_size = 1<<22); {
                uintptr_t va, end_va;
                end_va = kernel_virt_addr /* 0xc0000000 */ + load_sz /* 014c2000 */;
                for (va = kernel_virt_addr; va < end_va; va += map_size)
                        create_pgd_mapping(pgdir, va,
                                load_pa + (va - kernel_virt_addr),
                                map_size=PGDIR_SIZE, PAGE_KERNEL_EXEC);
                        /* => early_pg_dir[0x300 .. 0x305]=201000cf .. 206000cf */
        }
        /* Create two consecutive PGD mappings for FDT early scan */
        pa = dtb_pa /* parameter of setup_vm */ & ~(PGDIR_SIZE - 1); = 0x81c00000
        create_pgd_mapping(early_pg_dir, DTB_EARLY_BASE_VA=PGDIR_SIZE,
                        pa, PGDIR_SIZE, PAGE_KERNEL);
        => early_pg_dir[1]=207000c7
        create_pgd_mapping(early_pg_dir, DTB_EARLY_BASE_VA + PGDIR_SIZE,
                        pa + PGDIR_SIZE, PGDIR_SIZE, PAGE_KERNEL);
        => early_pg_dir[2]=208000c7
        dtb_early_va = (void *)DTB_EARLY_BASE_VA + (dtb_pa & (PGDIR_SIZE - 1)); /* =
PGDIR_SIZE = 0x400000 */
        dtb_early_pa = dtb_pa; /* = 0x81c00000 */
..} // setup_vm
```

The **relocate** label from **head.S** contains the following code.

```
/* a0 is early_pg_dir */
relocate:
                /* Relocate return address */
                la a1, kernel_virt_addr; REG_L a1, 0(a1) /*lw a1, 0(a1) ➔ a1 =
PAGE_OFFSET*/
```

```
                la a2, _start; sub a1, a1, a2; /*a1 = PAGE_OFFSET - _start */
                add ra, ra, a1 //ra is the return addr and was _start+something (0x80401018);
now is PAGE_OFFSET+something (0xc00010ec)
                /* Point stvec to virtual address of instruction after satp write */
                /* 1f means the address of the next label 1: and is c0001040 on my kernel */
                la a2, 1f; add a2, a2, a1; csrw CSR_TVEC, a2
                /* Compute satp for kernel page tables, but don't load it yet */
                srl a2, a0 /*early_pg_dir*/, PAGE_SHIFT /*12*/; satp address is PPN * 4K
                        li a1, SATP_MODE/*0x80000000*/; or a2, a2, a1
                        Note: in virtual, satp = 1 | asid (9 bit, unused) | PPN (22 bit).
                /*
                 * Load trampoline page directory, which will cause us to trap to
                 * stvec if VA != PA, or simply fall through if VA == PA.  We need a
                 * full fence here because setup_vm() just wrote these PTEs and we need
                 * to ensure the new translations are in use.
                 */
                la a0, trampoline_pg_dir; XIP_FIXUP_OFFSET a0;     srl a0, a0, PAGE_SHIFT
                or a0, a0, a1; sfence.vma; csrw CSR_SATP, a0
                /* from now on, satp points to trampoline and because satp[31] is 1 we have
                MMU enabled. The next instr from memory is at virtual (PC=80xxxxxx) which
                generates exception and PC<= stvec. stvec is the virtual address of instruction
                present at next label 1:
                .align 2
                1: /* Set trap vector to spin forever to help debug */
                la a0, .Lsecondary_park; csrw CSR_TVEC, a0
                /* Reload the global pointer */
                .option push; .option norelax; la gp, __global_pointer$; .option pop
                /*
                 * Switch to kernel page tables.  A full fence is necessary in order to
                 * avoid using the trampoline translations, which are only correct for
                 * the first superpage.  Fetching the fence is guaranteed to work
                 * because that first superpage is translated the same way.
                 */
                csrw CSR_SATP, a2 ; //early_pg_dir
                sfence.vma
                ret

                .Lsecondary_park:
                /* We lack SMP support or have too many harts, so park this hart */
                wfi
                j .Lsecondary_park
```

The **setup_trap_vector** label from **head.S** contains the following code:

```
setup_trap_vector:
                /* Set trap vector to exception handler */
                la a0, handle_exception; csrw CSR_TVEC, a0
```

```
/*
 * Set sup0 scratch register to 0, indicating to exception vector that
 * we are presently executing in kernel.
 */
csrw CSR_SCRATCH, zero; // sscratch
ret
```

After that, **head.S** restores the C environment and calls **start_kernel()**.

```
/* Restore C environment */
la tp, init_task # thread pointer to struct task_struct init_task;
sw zero, TASK_TI_CPU(tp) #define TASK_TI_CPU 16 /*is offsetof(struct task_struct,
thread_info.cpu), because CONFIG_THREAD_INFO_IN_TASK=y */
la sp, init_thread_union + THREAD_SIZE/* documented above*/

call soc_early_init // verified and it does nothing in rlsoc
tail start_kernel
```

This was **head.S** in linux 5.13.19.

7.3 RLSoC MMU

In rlsoc, the mmu job is implemented in **mmu.v**. This file also has code for the console, but we'll see about it later.

Until satp[31] becomes 1, rlsoc works in non-MMU mode: all addresses are physical addresses. The memory address format is shown below. Here, **w_instr_addr** comes from the processor. **r_mc_mode** is a flag which gives system control to an auxiliary microcontroller when is not null; it will be discussed later in the console subchapter.

```
wire [31:0] w_insn_paddr = (w_priv == `PRIV_M || w_satp[31] == 0) ?
                                    w_insn_addr : r_tlb_addr;
wire [31:0] w_mem_paddr  = (r_mc_mode != 0) ? w_mc_addr      :
                                  (w_priv == `PRIV_M || w_satp[31] == 0)  ?
                                    w_data_addr : r_tlb_addr;

wire [31:0] w_dram_addr =  (r_mc_mode!=0) ? w_mc_addr :
                                  (w_iscode && !w_tlb_busy) ? w_insn_paddr :
                                  (w_priv == `PRIV_M || w_satp[31] == 0) ?
                                   w_data_addr : (r_tlb_acs && !w_tlb_hit) ?
                                   r_tlb_pte_addr : w_mem_paddr;
wire [31:0] w_dram_addr_t = ((w_dram_addr[31:28]==9) ?
                                 (w_dram_addr & 32'h3fffff) + `BBL_SIZE :
                                  w_dram_addr & 32'h3fffff);
wire [31:0]  w_dram_addr_t2 =
                       (r_init_state == 1) ? r_zeroaddr     :
                       (r_init_state == 2) ? r_initaddr     : ..
                        w_dram_addr_t;
```

There are 3 TLBs: one for instructions, one for data read and one for data write. Each TLB is implemented within the **m_tlb** module which is a direct mapping cache.

```
m_tlb#(20, 22, `TLB_SIZE) TLB_inst_r (CLK, 1'b1, w_tlb_flush,
       w_tlb_inst_r_we, w_insn_addr[31:12], w_insn_addr[31:12],
       w_tlb_wdata, w_tlb_inst_r_addr, w_tlb_inst_r_oe);

m_tlb#(20, 22, `TLB_SIZE) TLB_data_r (CLK, 1'b1, w_tlb_flush,
       w_tlb_data_r_we, w_data_addr[31:12], w_data_addr[31:12],
       w_tlb_wdata, w_tlb_data_r_addr, w_tlb_data_r_oe);

m_tlb#(20, 22, `TLB_SIZE) TLB_data_w (CLK, 1'b1, w_tlb_flush,
       w_tlb_data_w_we, w_data_addr[31:12], w_data_addr[31:12],
       w_tlb_wdata, w_tlb_data_w_addr, w_tlb_data_w_oe);
```

The **r_tlb_addr** address is set in the page walk state machine. Here, **w_tlb_inst_r_addr** is an output of the **TLB_inst_r**, m_tlb instance.

```
// PAGE WALK state
always@(posedge CLK) begin
        if(r_pw_state == 0) begin
            // PAGE WALK START
            if(!w_dram_busy && w_use_tlb) begin
                // tlb miss
                if(!w_tlb_hit) begin
```

```
                    r_pw_state <= 1;
                end else begin
                    r_pw_state <= 7;
                    case ({w_iscode, w_isread, w_iswrite})
                        3'b100 : r_tlb_addr <= {w_tlb_inst_r_addr[21:2],
w_insn_addr[11:0]};
                        3'b010 : r_tlb_addr <= {w_tlb_data_r_addr[21:2],
w_data_addr[11:0]};
                        3'b001 : r_tlb_addr <= {w_tlb_data_w_addr[21:2],
w_data_addr[11:0]}; ..
```

The MMU instruction handling is implemented as follows. We will see how it behaves when simulating the switch from non-MMU to MMU mode, immediately after.

```
// rvcore
r_tlb_req <= (w_state==`S_IF && w_if_state!=1) ? `ACCESS_CODE  :
                    (w_d_en_t)                 ? `ACCESS_READ  :
                    (w_d_we_t)                 ? `ACCESS_WRITE : 2'h3;
// mmu
 wire        w_iscode        = (w_tlb_req == `ACCESS_CODE);
 wire [31:0] v_addr          = w_iscode ? w_insn_addr : w_data_addr;

// Level 1
    wire [31:0] vpn1             = {22'b0, v_addr[31:22]};
    wire [31:0] L1_pte_addr      = {w_satp[19:0], 12'b0} + {vpn1, 2'b0};
    wire  [2:0] L1_xwr           = w_mstatus[19] ? (L1_pte[3:1] |
                                    L1_pte[5:3]) : L1_pte[3:1];
    wire [31:0] L1_paddr         = {L1_pte[29:10], 12'h0};
    wire [31:0] L1_p_addr        = {L1_paddr[31:22], v_addr[21:0]};
    wire        L1_write         = !L1_pte[6] || (!L1_pte[7] && w_iswrite);
    wire        L1_success       = !(L1_xwr ==2 || L1_xwr == 6 ||
         (w_priv == `PRIV_S && (L1_pte[4] && !w_mstatus[18])) ||
         (w_priv == `PRIV_U && !L1_pte[4]) || (L1_xwr[w_tlb_req] == 0));

 // Level 0 is similar
```

The page table entry is read from memory in the page table state machine.

```
        … // Level 1
        else if(r_pw_state == 1 && !w_dram_busy) begin
            L1_pte      <= w_dram_odata;
            r_pw_state  <= 2;
        end … // Level 0 similar …
        // Success?
        else if(r_pw_state == 4) begin
            if(!L1_pte[0]) begin
                physical_addr   <= 0;
                page_walk_fail  <= 1;
            end
            else if(L1_xwr) begin
                physical_addr   <= (L1_success) ? L1_p_addr : 0;
                page_walk_fail  <= (L1_success) ? 0 : 1;
            end
            else if(!L0_pte[0]) begin …

// mmu outputs w_pagefault and cpu inputs it
```

```
assign w_pagefault = !page_walk_fail ? ~32'h0 : (w_iscode) ?
                      `CAUSE_FETCH_PAGE_FAULT : (w_isread) ?
                      `CAUSE_LOAD_PAGE_FAULT  : `CAUSE_STORE_PAGE_FAULT;

wire [21:0] w_tlb_wdata       = {physical_addr[31:12], 2'b0};
```

Now let's see what's happening when linux wants to turn on the MMU and use virtual addresses. For this we will simulate the system with the LAUR_DEBUG_AFTER_CSRW_SATP flag enabled.

Recall that "In systems with S-mode, the medeleg and mideleg registers must exist, and setting a bit in medeleg or mideleg will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. When a trap is delegated to S-mode, the scause register is written with the trap cause" [13]. We must remember that BBL has wrote the **medeleg** register with the CAUSE_FETCH_PAGE_FAULT (0xc) set.

The compiled **relocate** section from **head.S** in **vmlinux** dump looks like in the following listing.

```
c0001000 l     .head.text      00000000 relocate
..
c0001000 <efi_header_end>:
c0001000:      81c18593                addi    a1,gp,-2020 # c1488334 <kernel_virt_addr>
c0001004:      418c            lw      a1,0(a1)
c0001006:      fffff617        auipc   a2,0xfffff
c000100a:      ffa60613        addi    a2,a2,-6 # c0000000 <_start>
c000100e:      8d91            sub     a1,a1,a2
c0001010:      90ae            add     ra,ra,a1
c0001012:      00000617                auipc   a2,0x0
c0001016:      02e60613                addi    a2,a2,46 # c0001040 <efi_header_end+0x40>
c000101a:      962e            add     a2,a2,a1
c000101c:      10561073                csrw    stvec,a2
c0001020:      00c55613                srli    a2,a0,0xc
c0001024:      800005b7                lui     a1,0x80000
c0001028:      8e4d            or      a2,a2,a1
c000102a:      0148a517                auipc   a0,0x148a
c000102e:      fd650513                addi    a0,a0,-42 # c148b000 <trampoline_pg_dir>
c0001032:      8131            srli    a0,a0,0xc
c0001034:      8d4d            or      a0,a0,a1
c0001036:      12000073                sfence.vma
c000103a:      18051073                csrw    satp,a0
c000103e:      0001            nop
c0001040:      00000517                auipc   a0,0x0
c0001044:      03250513                addi    a0,a0,50 # c0001072 <setup_trap_vector+0x12>
c0001048:      10551073                csrw    stvec,a0
c000104c:      01488197                auipc   gp,0x1488
c0001050:      acc18193                addi    gp,gp,-1332 # c1488b18 <__global_pointer$>
c0001054:      18061073                csrw    satp,a2
c0001058:      12000073                sfence.vma
c000105c:      8082            ret
c000105e:      0001            nop
```

After instruction **csrw satp,a0** it will be generated an exception and PC←stvec.

```
pc <= stvec=c0001040. This is a virtual address, and:
va.ppn[1]=va[31:22]=0x300
va.ppn[0]=va[21:12]=0x001
va.pgoff=va[11:0]=0x040

satp, va.ppn[1] => trampoline_pg_dir[0x300] which is 201000cf. This is a pte, and:
pte.ppn[1]=pte[31:20]=0x201
pte.ppn[0]=pte[19:10]=0x000
pte.flags=pte[9:0]=0xcf

0x201000cf=
3322 2222 2222 1111 1111 1100 0000 0000
1098 7654 3210 9876 5432 1098 7654 3210
0010 0000 0001 0000 0000 0000 1100 1111

In mmu.v, we have:
wire [31:0] vpn1          = {22'b0, v_addr[31:22] /* 10 bits */}; = 0x300
wire [31:0] L1_pte_addr    = {w_satp[19:0], 12'b0} + {vpn1, 2'b0}; = &trampoline_pg_dir[0x300];
reg L1_pte     <= w_dram_odata; =201000cf
wire [31:0] L1_paddr       = {L1_pte[29:10], 12'h0}; =0x80400000
wire [31:0] L1_p_addr      = {L1_paddr[31:22], v_addr[21:0]}; /* in case of 4MB pages */
          = {80400000[31:22] , c0001040[21:0]}
          = {10'h10 0000 0001 , 22'b00 0000 0001 0000 0100 0000}
          = 80401040

0x80400000[31:22]=0x201
3322 2222 2222 1111 1111 1100 0000 0000
1098 7654 3210 9876 5432 1098 7654 3210
1000 0000 0100 0000 0000 0000 0000 0000
```

And we have L1_success, because we have create the pte entry with the flags:
PAGE_KERNEL_EXEC = _PAGE_READ | _PAGE_WRITE | _PAGE_PRESENT | _PAGE_ACCESSED | _PAGE_DIRTY | _PAGE_EXEC

So, in conclusion, when we simulate rlsoc we will see an output like the following.

```
stvec write:    time:0000000000062e06 stvec<=c0001040 pc=8040101c
satp write:    time:0000000000062e0f pc=8040103a r_insn_addr=8040103a
w_insn_data=18051073 satp<=8008188b pending_exception=ffffffff stvec=c0001040
mtvec=80000004 w_deleg=00000000 w_busy=0
new insn addr:    time:0000000000062e10 pc=8040103e r_insn_addr=8040103c
w_insn_data=18051073 satp=8008188b pending_exception=ffffffff stvec=c0001040
mtvec=80000004 w_deleg=00000000 w_busy=1
satp write:    time:0000000000062e10 pc=8040103e r_insn_addr=8040103c
w_insn_data=34011173 satp<=8008188b pending_exception=0000000c stvec=c0001040
mtvec=80000004 w_deleg=00000001 w_busy=0
new insn addr:    time:0000000000062e11 pc=c0001040 r_insn_addr=c0001040
w_insn_data=34011173 satp=8008188b pending_exception=ffffffff stvec=c0001040
```

```
mtvec=80000004 w_deleg=00000000 w_busy=1

new insn addr:     time:0000000000062e12 pc=c0001044 r_insn_addr=c0001044
w_insn_data=00000517 satp=8008188b pending_exception=ffffffff stvec=c0001040
mtvec=80000004 w_deleg=00000000 w_busy=1
new insn addr:     time:0000000000062e13 pc=c0001048 r_insn_addr=c0001048
w_insn_data=03250513 satp=8008188b pending_exception=ffffffff stvec=c0001040
mtvec=80000004 w_deleg=00000000 w_busy=1
```

7.4 Basics of TinyEMU

7.4.1 TinyEMU execution flow

Before starting to read this section, please see Note 7.1-1.

The tinyemu execution flow is shown in the following listing. The **main()** function is present in **temu.c**. Important declaration is **p**, a pointer to the **VirtMachineParams** type. main() calls **virt_machine_load_config_file()** from **machine.c** which calls **config_load_file() → load_file(), cb() → config_file_loaded()**; the important parameters that are passed to these functions are shown in the listing. The **vmc** field of **p** is set to **riscv_machine_class** which is defined in **riscv_machine.c**.

Also important calls from main() are **console_init()**, **riscv_machine_init()** – via virt_machine_init() and **riscv_machine_interp()** via virt_machine_run(); we will see them shortly.

```
temu.c::main() {
   VirtMachineParams p_s, *p = &p_s;..
   machine.c::virt_machine_load_config_file(p, path=argv[..], NULL, NULL)→
config_load_file(s, filename, cb=config_file_loaded, s);→
size = load_file(&buf, filename); cb(opaque, buf, size) →
config_file_loaded(VMConfigLoadState *s=opaque) {
   VMConfigLoadState *s = opaque;
   VirtMachineParams *p = s->vm_params;
   virt_machine_parse_config(VirtMachineParams p, (char *)buf, buf_len)→
      p->vmc = struct VirtMachineClass = virt_machine_find_class(p->machine_name =
"riscv32") = virt_machine_list["riscv32"] = riscv_machine.c:: riscv_machine_class;
      p->vmc->virt_machine_set_defaults(p); /* memset(p, 0, sizeof(p)) */
      p->files[VM_FILE_BIOS].filename = vm_get_str_opt(cfg, "bios", &str)
      p->files[VM_FILE_KERNEL].filename = vm_get_str_opt(..);
      p->files[VM_FILE_INITRD].filename = vm_get_str_opt(..);
      p->tab_drive[p->drive_count].filename = str = "root-riscv32.bin";
      p->tab_drive[p->drive_count].device = strdup_null(str);
      .. input_device, ethernet, p->tab_fs[..]="fs0", display ..
   }
   vm_add_cmdline(p, cmdline); // linux cmd line
   fname = get_file_path(p->cfg_filename, p->tab_drive[i].filename); // rootfs
   drive = block_device_init(fname, drive_mode);
   .. fs, eth, ..
   if(!p->display_device) p->console = console_init(allow_ctrlc) { CharacterDevice *dev;
STDIODevice *s; dev->opaque = s; dev->write_data = console_write {fwrite(buf, 1, len,
stdout);}; dev->read_data = console_read; return dev; }
   VirtMachine s = virt_machine_init(p) = p->vmc->virt_machine_init(p); =
riscv_machine_init(const VirtMachineParams *p);
```

```
   for(;;)
      virt_machine_run(s) {..
         virt_machine_interp(m, MAX_EXEC_CYCLE=500000)->s->vmc->virt_machine_interp(s,
max_exec_cycle);=riscv_machine_interp(..);
      }
} // main
```

Listing 7.4-1. Tinyemu main().

The riscv_machine_class variable is defined as follows:

```
const VirtMachineClass riscv_machine_class = {
   "riscv32,riscv64,riscv128",
   riscv_machine_set_defaults,
   riscv_machine_init,
   riscv_machine_end,
   riscv_machine_get_sleep_duration,
   riscv_machine_interp, ..
};
```

Listing 7.4-2. Tinyemu riscv_machine_class variable.

The riscv_machine_init() function is shown below. Its contents are self explanatory. PLIC stands for Platform-Level Interrupt Controller and CLINT for Core Local Interrupter. HTIF (Host Target Interface) provides console emulation for tinyemu. BBL provides HTIF console access via the SBI (Supervisor Binary Interface). **riscv_build_fdt**() builds the dtb.

```
static VirtMachine *riscv_machine_init(const VirtMachineParams *p)
{
   RISCVMachine *s; s->mem_map = iomem.c::phys_mem_map_init(); /* .. s-
>mem_map.register_ram = default_register_ram; .set_ram_addr = default_set_add .. */
   s->cpu_state = (RISCVCPUState*) riscv_cpu_init(s->mem_map, max_xlen=32);
   ..
   cpu_register_ram(s->mem_map, RAM_BASE_ADDR /* 0x80000000 */,
(VirtMachineParams *p)->ram_size, ram_flags=0); //=s->register_ram(s, addr, size,
devram_flags);=default_register_ram();
   cpu_register_ram(s->mem_map, 0x00000000, LOW_RAM_SIZE=0x00010000=64KB, 0);
   ..
   cpu_register_device(s->mem_map, CLINT_BASE_ADDR, CLINT_SIZE, opaque=s, clint_read,
clint_write, DEVIO_SIZE32) {
         PhysMemoryRange *pr;
         pr = &s->phys_mem_range[s->n_phys_mem_range++];
         pr->map = s->mem_map; pr->addr = addr; pr->org_size = size;
         pr->is_ram = FALSE; pr->opaque = opaque;
         pr->read_func = read_func; pr->write_func = write_func; ..
   }
   cpu_register_device(s->mem_map, PLIC_BASE_ADDR, PLIC_SIZE, s,
                                          plic_read, plic_write, DEVIO_SIZE32);
   for(i = 1; i < 32; i++)
         irq_init(&s->plic_irq[i], plic_set_irq, s, i);
```

```
    cpu_register_device(s->mem_map, HTIF_BASE_ADDR, 16,
                            s, htif_read, htif_write, DEVIO_SIZE32);

   /* VIRTIOBusDef vbus_s, *vbus = &vbus_s; */
    s->common.console = p->console;    /* virtio console */ ..
    vbus->mem_map = s->mem_map;
    vbus->addr = VIRTIO_BASE_ADDR; // 0x40010000, in tinyemu dts
    irq_num = VIRTIO_IRQ; // 1
    if (p->console) {
       vbus->irq = &s->plic_irq[irq_num];
       s->common.console_dev = virtio_console_init(vbus, p->console);
       vbus->addr += VIRTIO_SIZE; // 0x1000
       irq_num++;        s->virtio_count++;
    }
    .. add virtio block device, (filesystem, net device, display device, input device)

    copy_bios(s, p->files[VM_FILE_BIOS].buf, p->files[VM_FILE_BIOS].len,
          p->files[VM_FILE_KERNEL].buf, p->files[VM_FILE_KERNEL].len,
          p->files[VM_FILE_INITRD].buf, p->files[VM_FILE_INITRD].len,
          p->cmdline); {
                        ram_ptr = get_ram_ptr(s, 0, TRUE); = phys_mem_get_ram_ptr( s-
                >mem_map, ..);
                        memcpy(ram_ptr, buf, buf_len); // bios (BBL)
                        // we have initramfs appended to kernel
                        memcpy(ram_ptr + kernel_base, kernel_buf, kernel_buf_len);
                        // initrd_buf_len is 0
                        fdt_addr = 0x1000 + 8 * 8; // dtb
                        riscv_build_fdt(s, ram_ptr + fdt_addr,
                                        RAM_BASE_ADDR + kernel_base, kernel_buf_len,
                                        RAM_BASE_ADDR + initrd_base, initrd_buf_len,
                                        cmd_line);
                        /* jump_addr = 0x80000000 */
                        q = (uint32_t *)(ram_ptr + 0x1000);
                        q[0] = 0x297 + 0x80000000 - 0x1000; /* auipc t0, jump_addr */ ..
                        q[4] = 0x00028067; /* jalr zero, t0, jump_addr */
    } // copy_bios

    return (VirtMachine *)s;
}
```

Listing 7.4-3. Tinyemu riscv_machine_init() function.

Regarding the function **riscv_cpu_init**(), in the Makefile, for riscv32, MAX_XLEN is defined as 32, CONFIG_RISCV_MAX_XLEN=64; for riscv64, MAX_XLEN is defined as 64. We must have only one function **riscv_cpu_init**() for both riscv32 and riscv64, and there is a single **temu** executable. **xglue** concatenates two strings to compose one single name.

```
./cutils.h:35:#define xglue(x, y) x ## y
./cutils.h:36:#define glue(x, y) xglue(x, y)

const RISCVCPUClass glue(riscv_cpu_class, MAX_XLEN) = {
        glue(riscv_cpu_init, MAX_XLEN),
        glue(riscv_cpu_end, MAX_XLEN),
        glue(riscv_cpu_interp, MAX_XLEN),
        glue(riscv_cpu_get_cycles, MAX_XLEN),
        glue(riscv_cpu_set_mip, MAX_XLEN),..
};

#if CONFIG_RISCV_MAX_XLEN == MAX_XLEN /* valid only for riscv64 */
RISCVCPUState *riscv_cpu_init(PhysMemoryMap *mem_map, int max_xlen) {
   const RISCVCPUClass *c;
   switch(max_xlen) { case 32: c = &riscv_cpu_class32; .. }
   return c->riscv_cpu_init(mem_map);
}

static RISCVCPUState *glue(riscv_cpu_init,  MAX_XLEN)(PhysMemoryMap *mem_map)
{
       RISCVCPUState *s;
   #ifdef USE_GLOBAL_STATE (is defined)
        s = &riscv_cpu_global_state;
   #else
       s = mallocz(sizeof(*s));
   #endif
        s->common.class_ptr = &glue(riscv_cpu_class, MAX_XLEN);
        s->mem_map = mem_map;
        s->pc = 0x1000;
        s->priv = PRV_M; /*3*/ ..
        return s;
}
```

Listing 7.4-4. Tinyemu riscv_cpu_init() function.

The **riscv_cpu_interp32()** function is shown below. Instructions are read from memory and then executed.

```
void glue(riscv_cpu_interp, MAX_XLEN)(RISCVCPUState *s, int n_cycles)
{
   RISCVCPUClass *c = s->class_ptr;
   c->riscv_cpu_interp(s, n_cycles);=riscv_cpu_class32->glue(riscv_cpu_interp, MAX_XLEN); {
   timeout = s->insn_counter + n_cycles;
   while (!s->power_down_flag && (int)(timeout - s->insn_counter) > 0) {
      n_cycles = timeout - s->insn_counter;
      switch(s->cur_xlen) {
         case 32: riscv_cpu_interp_x32(s, n_cycles);
                = glue(riscv_cpu_interp_x, XLEN) (RISCVCPUState *s, int n_cycles1) {
                  /* check pending interrupts */
```

```
                ...
            s->pending_exception = -1;
            /* we use a single execution loop to keep a simple control flow */
            for(;;) {
                    if (unlikely(code_ptr >= code_end)) {...
                            s->pc = GET_PC(); ..
                    } else {
                            /* fast path */
                    insn = get_insn32(code_ptr);
                    }..
                    opcode = insn & 0x7f;
                    rd = (insn >> 7) & 0x1f;
                    rs1 = (insn >> 15) & 0x1f;
                    rs2 = (insn >> 20) & 0x1f;
                    switch(opcode) {..execute }
        }
}
```

7.4.2 TinyEMU console

The function riscv_machine_init() also initializes the console. Its fragment is shown below. One important thing is that it will call cpu_register_device(.. **virtio_mmio_read, virtio_mmio_write**, ..). These two functions (passed as parameters) from the emulator will be called when the linux kernel uses the console. We will see them in action in the next subchapter.

```
VirtMachine *riscv_machine_init(const VirtMachineParams *p)
{  ..
  /* VIRTIOBusDef vbus_s, *vbus = &vbus_s; */
   s->common.console = p->console;    /* virtio console */ ..
   vbus->mem_map = s->mem_map;
   vbus->addr = VIRTIO_BASE_ADDR; // 0x40010000, in tinyemu dts
   irq_num = VIRTIO_IRQ; // 1
   if (p->console) {
      vbus->irq = &s->plic_irq[irq_num];
      s->common.console_dev = virtio_console_init(VIRTIOBusDef *bus=vbus,
CharacterDevice *cs=p->console) {
         VIRTIOConsoleDevice *s; s = mallocz(..);
        virtio_init(VIRTIODevice *s=&s->common, VIRTIOBusDef *bus=bus, uint32_t
device_id=3, int config_space_size=4, VIRTIODeviceRecvFunc *device_recv
=virtio_console_recv_request) {
            ..if (!bus->pci_bus) { /* MMIO case */
                  s->mem_map = bus->mem_map;
                  s->irq = bus->irq;
                  s->mem_range = cpu_register_device(s->mem_map, bus->addr,
                     VIRTIO_PAGE_SIZE, opaque=s, virtio_mmio_read, virtio_mmio_write,
                                 DEVIO_SIZE8 | DEVIO_SIZE16 | DEVIO_SIZE32);
                  s->get_ram_ptr = virtio_mmio_get_ram_ptr;
            } // if (!pci->bus)
            s->device_id = device_id; // 3=console device
            s->vendor_id = 0xffff;
            s->config_space_size = config_space_size; // 4
            s->device_recv = device_recv = virtio_console_recv_request(VIRTIODevice *s, int
queue_idx, int desc_idx, int read_size, int write_size) /* ← queue_notify(s, queue_idx=val)
← virtio_mmio_write(offset=VIRTIO_MMIO_QUEUE_NOTIFY=0x50) */
            virtio_reset(s);
          } // virtio_init
          s->common.device_features = (1 << 0); /* VIRTIO_CONSOLE_F_SIZE */
          s->common.queue[0].manual_recv = TRUE;
          s->cs = cs;
          return (VIRTIODevice *)s;
      } // virtio_console_init
      vbus->addr += VIRTIO_SIZE; // 0x1000
      irq_num++;
```

```
        s->virtio_count++;
    } // if (p->console)
    ..
}
```

7.5 Linux early console

Note 7.5-1: The linux command line specified in the dtb is the following:

bootargs = "console=hvc0 earlycon=sbi root=/dev/vda rw";

Note 7.5-2: To see the call stack of a specific function you can use **dump_stack**().

So, the early console is named **sbi**. For the early console, in arch/riscv/kernel/**sbi.c** it is used the **sbi_ecall** function in conjunction with SBI_EXT_0_1_CONSOLE_PUTCHAR in order to print a character on the console. The sbi_console_putchar() function is used in sbi_putc() from drivers/tty/serial/**earlycon-riscv-sbi.c** and this in sbi_console_write().

```
void sbi_console_putchar(int ch) {
        sbi_ecall(SBI_EXT_0_1_CONSOLE_PUTCHAR, 0, ch, 0, 0, 0, 0, 0);
}
static void sbi_putc(struct uart_port *port, int c) {
        sbi_console_putchar(c);
}

static void sbi_console_write(struct console *con,
                              const char *s, unsigned n)
{
        struct earlycon_device *dev = con->data;
        uart_console_write(&dev->port, s, n, sbi_putc);
}
static int __init early_sbi_setup(struct earlycon_device *device,
                                  const char *opt) {
        device->con->write = sbi_console_write;
        return 0;
}
EARLYCON_DECLARE(sbi, early_sbi_setup);
```

What EARLYCON_DECLARE does is shown in the following listing. It declares the **__earlycon_sbi** variable of type **struct earlycon_id** with the fields **name** equal to **"sbi"** and **setup** equal to **early_sbi_setup**.

```
#define OF_EARLYCON_DECLARE(_name, compat, fn)                          \
        static const struct earlycon_id __UNIQUE_ID(__earlycon_##_name) \
                EARLYCON_USED_OR_UNUSED __section("__earlycon_table")  \
                __aligned(__alignof__(struct earlycon_id))             \
                = { .name = __stringify(_name),                        \
                    .compatible = compat,                              \
                    .setup = fn }
#define EARLYCON_DECLARE(_name, fn)     OF_EARLYCON_DECLARE(_name, "", fn)
EARLYCON_DECLARE(sbi, early_sbi_setup);
```

For the following please consider that, in assembly:

- the ENTRY(*symbol*) linker script commands the set of the entry point;

- in AT (ldadr), the expression ldadr that follows the AT keyword specifies the load address of the section;

- KEEP mark sections that should not be eliminated.

In **earlycon.c** is used the directive early_param("earlycon", param_setup_earlycon);

let's see what this implies.

```
#define early_param(str, fn) __setup_param(str, fn, fn, 1)

#define __setup_param(str, unique_id, fn, early)                    \
        static const char __setup_str_##unique_id[] __initconst     \
                __aligned(1) = str;                                 \
        static struct obs_kernel_param __setup_##unique_id          \
                __used __section(".init.setup")                     \
                __aligned(__alignof__(struct obs_kernel_param))     \
                = { __setup_str_##unique_id, fn, early }

init/main.c:: extern const struct obs_kernel_param __setup_start[], __setup_end[];

include/asm-generic.h/vmlinux.lds.h
#ifdef CONFIG_SERIAL_EARLYCON
#define EARLYCON_TABLE() . = ALIGN(8);                              \
                         __earlycon_table = .;                      \
                         KEEP(*(__earlycon_table))          \
                         __earlycon_table_end = .;
#else .. #endif

/* init and exit section handling */
#define INIT_DATA                                                   \
        KEEP(*(SORT(___kentry+*)))                                  \
        *(.init.data init.data.*)                                   \
      .. EARLYCON_TABLE()..


#define INIT_SETUP(initsetup_align)                                 \
                . = ALIGN(initsetup_align);                         \
                __setup_start = .;                                  \
                KEEP(*(.init.setup))                                \
                __setup_end = .;

#define INIT_DATA_SECTION(initsetup_align)                          \
        .init.data : AT(ADDR(.init.data) - LOAD_OFFSET) {           \
```

```
                INIT_DATA                                                   \
                INIT_SETUP(initsetup_align)                                 \
                INIT_CALLS                                                  \
                CON_INITCALL                                                \
                INIT_RAM_FS                                                 \
        }

arch/riscv/kernel/vmlinux-xip.lds.S
..
        /* Start of init data section */
        __init_data_begin = .;
        INIT_DATA_SECTION(16)
..

static int __init param_setup_earlycon(char *buf)
{
        .. err = setup_earlycon(buf);
        if (err == -ENOENT || err == -EALREADY)
                return 0;
        return err;
}

int __init setup_earlycon(char *buf)
{
.. for (match = __earlycon_table; match < __earlycon_table_end; match++) {
                if (strncmp(buf, match->name, len))
                        continue;
            ..buf = NULL;
                .. return register_earlycon(buf, match);

}

static struct console early_con = {
        .name =         "uart", /* fixed up at earlycon registration */
        .flags =    CON_PRINTBUFFER | CON_BOOT,
        .index =    0,
};

static int __init register_earlycon(char *buf/*NULL for sbi*/, const struct earlycon_id
*match)
{
        struct uart_port *port = &early_console_dev.port;

        if (buf && !parse_options(&early_console_dev, buf))
                buf = NULL;

        port->uartclk = BASE_BAUD * 16;
```

```
        if (port->mapbase)
                port->membase = earlycon_map(port->mapbase, 64);

        earlycon_init(&early_console_dev, match->name);
        err = match->setup(&early_console_dev, buf);
        earlycon_print_info(&early_console_dev);

        register_console(early_console_dev.con);
        return 0;
}
```

The **start_kernel()** function calls **setup_arch()** and this calls **parse_early_param()**.

```
void __init parse_early_param(void) {
        if (done)
                return;
        strlcpy(tmp_cmdline, boot_command_line, COMMAND_LINE_SIZE);
        parse_early_options(tmp_cmdline);=parse_args("early options", cmdline, NULL, 0,
0, 0, NULL, do_early_param); →parse_one(..) → do_early_param().
    done = 1;
}
static int __init do_early_param(char *param, char *val,
                                 const char *unused, void *arg) {
        const struct obs_kernel_param *p;
        for (p = __setup_start; p < __setup_end; p++) {
                if ((p->early && parameq(param, p->str)) ||
                   (strcmp(param, "console") == 0 &&
                    strcmp(p->str, "earlycon") == 0)
                ) {
                        if (p->setup_func(val) != 0).. // = param_setup_earlycon()
}
```

So, the **register_console()** function is called immediately after bbl gives control to linux. This function call will call twice **try_enable_new_console()**; the first call, try_enable_new_console(newcon, true) returns -ENOENT and the second call, try_enable_new_console(newcon, false) returns 0. So, we have found and registered the early console "sbi". register_console() will set:
- newcon->flags |= CON_ENABLED; newcon->flags |= CON_CONSDEV for early_con (is 1st);
- has_preferred_console is set to 1;
- newcon->next = console_drivers->next; console_drivers->next = newcon;
- exclusive_console = newcon.

Because INIT_DATA_SECTION implies CON_INITCALL, we will have two more calls of register_console():

- one, as start_kernel()→console_init()→con_init()→register_console() and this will call twice try_enable_new_console() and each call will return -ENOENT=-2 for **vt_console_driver**;

- one, as start_kernel()→console_init()→hvc_console_init()→register_console() and this will return -ENODEV=-19 for **hvc_console**, because this was not yet configured.

```
hvc_console.c::static int __init hvc_console_init(void)
{
        register_console(&hvc_console);
        return 0;
}
console_initcall(hvc_console_init);

vt.c::static int __init con_init(void) {..
#ifdef CONFIG_VT_CONSOLE
        register_console(&vt_console_driver);..
}
console_initcall(con_init);

#define CON_INITCALL                                                    \
        __con_initcall_start = .;                                       \
        KEEP(*(.con_initcall.init))                                     \
        __con_initcall_end = .;
#define console_initcall(fn)  ___define_initcall(fn, con /*name id*/, .con_initcall)
#define ___define_initcall(fn, id, __sec)                       \
        __unique_initcall(fn, id, __sec, __initcall_id(fn))
#define __unique_initcall(fn, id, __sec, __iid)                 \
        ____define_initcall(fn,                                 \
                __initcall_stub(fn, __iid, id),                 \
                __initcall_name(initcall, __iid, id),           \
                __initcall_section(__sec, __iid))
#define ____define_initcall(fn, __unused, __name, __sec)  \
        static initcall_t __name __used                         \
                __attribute__((__section__(__sec))) = fn;
```

In the **register_console()** comments we find that “There are two types of consoles - bootconsoles (early_printk) and "real" consoles (everything which is not a bootconsole) which are handled differently:
- any number of bootconsoles can be registered at any time;
- as soon as a "real" console is registered, all bootconsoles will be unregistered automatically;
- once a "real" console is registered, any attempt to register a bootconsoles will be rejected”.

7.6 The hvc0 console

7.6.1 Preliminary considerations

The rlsoc and tinyemu device tree nodes for console are similar but a little bit different. They are shown in the following listing. So, for rlsoc, the console is memory mapped at the 0x40000000 address, and for tinyemu at the 0x40010000 address.

```
// rlsoc console
        virtio@40000000 {
            compatible = "virtio,mmio";
            reg = <0x0 0x40000000 0x0 0x8000000>;
            interrupts-extended = <0x2 0x1>;
        };
// rvsoc disk
        virtio@48000000 {
            compatible = "virtio,mmio";
            reg = <0x0 0x48000000 0x0 0x8000000>;
            interrupts-extended = <0x2 0x2>;
        };
// tinyemu console
        virtio@40010000 {
            compatible = "virtio,mmio";
            reg = <0x00 0x40010000 0x00 0x1000>;
            interrupts-extended = <0x02 0x01>;
        };
// tinyemu disk
        virtio@40011000 {
            compatible = "virtio,mmio";
            reg = <0x00 0x40011000 0x00 0x1000>;
            interrupts-extended = <0x02 0x02>;
        };
```

In the linux kernel **virtio_mmio.c** source, we have the following:

```
static int __init virtio_mmio_init(void) { return platform_driver_register(&virtio_mmio_driver); }
virtio_mmio.c::module_init(virtio_mmio_init);

static struct platform_driver virtio_mmio_driver = {
        .probe          = virtio_mmio_probe, ..
        .driver         = {
                .name  = "virtio-mmio",
                .of_match_table        = virtio_mmio_match, ..
        },
};

static const struct of_device_id virtio_mmio_match[] = {
        { .compatible = "virtio,mmio", },
```

```
        {},
};
MODULE_DEVICE_TABLE(of, virtio_mmio_match);
```

So, **virtio_mmio_probe()** will be called immediately after the dtb is parsed, because is the probe function and in dtb we have the compatible string “virtio,mmio” for console. Let’s see this function in the next listing. We can see important calls that interrogate the hardware asking for version and id. VIRTIO_MMIO_VERSION and VIRTIO_MMIO_DEVICE_ID are defined in include/uapi/linux/**virtio_mmio.h**; other important constants are defined in this file, like VIRTIO_MMIO_QUEUE_DESC_LOW, VIRTIO_MMIO_QUEUE_AVAIL_LOW, VIRTIO_MMIO_QUEUE_USED_LOW, VIRTIO_MMIO_QUEUE_NOTIFY, etc. The id for console is VIRTIO_ID_CONSOLE and is 3.

Please note that the function readl() reads and and **writel**() writes 32 bit (4 bytes) data from / to the memory-mapped I/O space.

```
#define VIRTIO_ID_NET                   1 /* virtio net */
#define VIRTIO_ID_BLOCK                 2 /* virtio block */
#define VIRTIO_ID_CONSOLE               3 /* virtio console */

static int virtio_mmio_probe(struct platform_device *pdev)
{
        struct virtio_mmio_device *vm_dev;
        vm_dev = devm_kzalloc(&pdev->dev, sizeof(*vm_dev), GFP_KERNEL);
        vm_dev->vdev.dev.parent = &pdev->dev;..
        vm_dev->vdev.config = &virtio_mmio_config_ops; /*
        ..vm_dev->pdev = pdev;..
        vm_dev->base = devm_platform_ioremap_resource(pdev, 0);
        ..vm_dev->version = readl(vm_dev->base + VIRTIO_MMIO_VERSION) //= 2;
        vm_dev->vdev.id.device = readl(vm_dev->base + VIRTIO_MMIO_DEVICE_ID);
        ..rc = dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(..)..);
        platform_set_drvdata(pdev, vm_dev);
        rc = register_virtio_device(&vm_dev->vdev); //→ virtio_add_status(dev,
VIRTIO_CONFIG_S_ACKNOWLEDGE); // however, status is not used in rlsoc
}

static const struct virtio_config_ops virtio_mmio_config_ops = {
                .get            = vm_get,               .set            = vm_set,
                .generation     = vm_generation,
                ...find_vqs     = vm_find_vqs,          .del_vqs        = vm_del_vqs,
                .get_features   = vm_get_features,
                .bus_name       = vm_bus_name,
                .get_shm_region = vm_get_shm_region,
};
```

7.6.2 Linux output console

In kernel/printk/**printk.c**, we have the console_unlock() function which is described in the kernel like this: "while the console_lock was held, console output may have been buffered by printk(). If this is the case, console_unlock(); emits the output prior to releasing the lock". This function calls call_console_drivers(ext_text, ext_len, text, len). **call_console_drivers**() does an important thing: "calls the console drivers, asking them to write out log_buf[start] to log_buf[end - 1]":

```
void call_console_drivers(const char *ext_text, size_t ext_len, const char *text, size_t len) {
        for_each_console(con) con->write(con, text, len); ..
}
```

The **hvc_console** variable is defined and used as follows (we are specially interested in the **put_chars()** function):

```
static struct console hvc_console = {
        .name           = "hvc",
        .write          = hvc_console_print,..
        .setup          = hvc_console_setup,..
};

const struct hv_ops *cons_ops[MAX_NR_HVC_CONSOLES];

hvc_console_print(struct console *co, const char *b, unsigned count) {
        ..int index=co->index;..
        cons_ops[index]->put_chars(vtermnos[index], c, i);..
}
```

The hvc console is created as follows. In kernel **virtio_console.c**, we have the following:

```
static struct virtio_driver virtio_console = {..
        .id_table =     id_table,
        .probe =        virtcons_probe, ..
};
static const struct virtio_device_id id_table[] = {
            { VIRTIO_ID_CONSOLE, VIRTIO_DEV_ANY_ID },
            { 0 },
};
static struct bus_type virtio_bus = {
            .name  = "virtio",
            .match = virtio_dev_match,..
            .probe = virtio_dev_probe, ..
};
```

```
virtio_console.c::init()->register_virtio_driver(struct virtio_driver *driver = &virtio_console) {
        driver->driver.bus = &virtio_bus;
        return driver_register(&driver->driver); //→ calls the bus probe function
}
module_init(init);
```

virtcons_probe() is called by **virtio_dev_probe()** and it will setup the hvc console. This is made by calling add_port()→init_port_console()→hvc_alloc()→hvc_check_console(). Finally register_console() is called and it will succeed:

```
static void hvc_check_console(int index) {
        if (hvc_console.flags & CON_ENABLED)
                return;
        if (index == hvc_console.index)
                register_console(&hvc_console);
}
```

Let's dig a bit in **virtcons_probe()** and see how we arrive to **put_chars()**:

```
int virtcons_probe(struct virtio_device *vdev){
  .. // after init_vqs().
  if(!multiport) /* For backward compatibility: Create a console port if we're running on
older host. */ add_port(portdev, id=0) {
        /* If we're not using multiport support, this has to be a console port. */
   err = init_port_console(struct port *port); {
     port->cons.hvc = hvc_alloc(port->cons.vtermno, data=0 /*virtqs have implicit
notifications*/, ops=&hv_ops /*has get/put_chars(),..*/, PAGE_SIZE); {
        struct hvc_struct *hp;
        if (atomic_inc_not_zero(&hvc_needs_init /*-1*/)) { /* branch taken */
                int err = hvc_init(); {
                   struct tty_driver *drv = alloc_tty_driver(HVC_ALLOC_TTY_ADAPTERS);
                   drv->name = "hvc";
                   drv->major = HVC_MAJOR // 229
                   ..tty_set_operations(drv, &hvc_ops);..
                 }
         ..}
        hp->ops = ops; ..
        ..hp->index = i;
        if (i < MAX_NR_HVC_CONSOLES) {
           cons_ops[i] = ops;
           vtermnos[i] = vtermno;
        }
     ..}
   ..}
  }..
}
```

```
struct tty_operations hvc_ops {.. .write=hvc_write()→hvc_push(hp=tty->driver_data); ..}

int hvc_push(struct hvc_struct *hp) {
        int n;
        n = hp->ops->put_chars(hp->vtermno, hp->outbuf, hp->n_outbuf);..
}

/* The operations for console ports. */
static const struct hv_ops hv_ops = {
        .get_chars = get_chars,
        .put_chars = put_chars,..
}
```

Now it's clear that **put_chars()** is used to send characters to virtio console.

virtcons_probe() → init_vqs() → vm_find_vqs() → vm_setup_vq() → vring_create_virtqueue() → vring_create_virtqueue_split() which creates and configures the virtual queues used by linux for the console. vm_setup_vq() will inform rlsoc of the **split.vring** descriptor fields and addresses. The **split** and **vring** fields are shown in the listing below. The most important are **avail**, **used** and **desc** which rlsoc is informed about and uses them; these are created as DMA addressable fields (see vring_create_virtqueue_split() →vring_alloc_queue()→vring_init()), so they can be used by the microcontroller.

The full configuration of the console between linux and tinyemu is shown in appendix 1.

```
struct vring_virtqueue {
        struct virtqueue vq; {
                ..void (*callback)(struct virtqueue *vq);
                struct virtio_device *vdev;..
        }
        /* Is this a packed ring? */ bool packed_ring;..
        /* Last used index we've seen. */      u16 last_used_idx;
        union {
                /* Available for split ring */
                struct {
                        /* Actual memory layout for this queue. */   struct vring vring;
                        #define VRING_DESC_ALIGN_SIZE 16
                        #define VRING_AVAIL_ALIGN_SIZE 2
                        #define VRING_USED_ALIGN_SIZE 4
                        struct vring {    unsigned int num;
                                          vring_desc_t {__virtio64 addr; __virtio32 len;
                                                __virtio16 flags, __virtio16 next; } *desc;
                                          vring_avail_t {__virtio16 flags, idx, ring[];} *avail;
                                          vring_used_t {__virtio16 flags, idx;
```

```
                                        vring_used_elem_t {__virtio32 id, len;} ring[];} *used;
                        };..
                        /* Last written value to avail->idx in guest byte order. */
                        u16 avail_idx_shadow;
                        /* Per-descriptor state. */
                        struct vring_desc_state_split *desc_state;..
                } split;
        ..} // union
        /* How to notify other side. FIXME: commonalize hcalls! */
        bool (*notify)(struct virtqueue *vq);..
};
```

Let's see what **put_chars()** does. Mainly, it fills the queue's descriptor fields (using **virtqueue_add_split**()) and notifies (using virtqueue_notify()) rlsoc that new data is available to be printed on the console.

```
put_chars(u32 vtermno, const char *buf, int count)
{
        /* We turn the characters into a scatter-gather list, add it to the output queue and
        then kick the Host. Then we sit here waiting for it to finish.*/
        if (unlikely(early_put_chars)) // false
                return early_put_chars(vtermno, buf, count);
        port = find_port_by_vtermno(vtermno);
        data = kmemdup(buf, count, GFP_ATOMIC);
        sg_init_one(sg, data, count); /* link sg to data */
        ret = __send_to_port(port, sg, nents=1, count, data, nonblock=false); {
                out_vq = port->out_vq;
                reclaim_consumed_buffers(port);
                virtqueue_add_outbuf(out_vq, sg, num=nents=1, data,
                GFP_ATOMIC);=virtqueue_add(vq, &sg, total_sg=num, out_sgs=1, in_sgs=0,
                data, void *ctx=NULL, gfp_t gfp);=vq->packed_ring ? :
                virtqueue_add_split(_vq, sgs, total_sg, out_sgs, in_sgs, data, ctx, gfp); {
                        struct vring_virtqueue *vq = to_vvq(_vq);
                        if (unlikely(vq->broken)) return -EIO;
                        if (virtqueue_use_indirect(_vq, total_sg)) .. // false.
                        else {
                          desc = vq->split.vring.desc; // important
                          indirect = false, i = head = vq->free_head; descs_used = total_sg;
                        }
                        for (n = 0; n < out_sgs; n++) {
                            for (sg = sgs[n]; sg; sg=sg_next(sg)) {
                                dma_addr_t addr = vring_map_one_sg(vq, sg,
                DMA_TO_DEVICE);
                                desc[i].addr = cpu_to_virtio64(_vq->vdev, addr);
                                desc[i].len = cpu_to_virtio32(_vq->vdev, sg->length);
                                prev = i; i = virtio16_to_cpu(_vq->vdev, desc[i].next);
                            }
                        }
                        for (; n < (out_sgs + in_sgs/*0*/); n++).. // 0 iterations
                        vq->free_head = i;
                        vq->split.desc_state[head].data = data; // important
```

```
                        vq->split.desc_state[head].indir_desc = ctx = NULL;
                        /* Put entry in available array (but don't update avail->idx until they do
                sync). */
                        avail = vq->split.avail_idx_shadow & (vq->split.vring.num - 1);
                        vq->split.vring.avail->ring[avail] = cpu_to_virtio16(_vq->vdev, head);
                        vq->split.avail_idx_shadow++;
                        vq->split.vring.avail->idx = cpu_to_virtio16(_vq->vdev,
                                        vq->split.avail_idx_shadow);
                                vq->num_added++;
                        ..return 0;
                } // virtqueue_add_split
                /* Tell Host to go! */
                virtqueue_kick(out_vq); = bool virtqueue_kick(struct virtqueue *vq) {
                        if (virtqueue_kick_prepare(vq)=virtqueue_kick_prepare_split(_vq)=1)
                                return virtqueue_notify(vq); {
                                        struct vring_virtqueue *vq = to_vvq(_vq);
                                        if (vq->broken) return false;
                                        if (!vq->notify(_vq)=vm_notify { // important
                                                struct virtio_mmio_device *vm_dev =
                                to_virtio_mmio_device(vq->vdev);
                                                writel(vq->index, vm_dev->base +
                                VIRTIO_MMIO_QUEUE_NOTIFY /*0x50*/);
                                        }) {
                                                vq->broken = true; return false;
                                        } // if
                                        return true;
                                } // virtqueue_notify
                           return true;
                } // virtqueue_kick
                // wait till the host ack that it pushed out the data we sent
                while (!virtqueue_get_buf(out_vq, &len) && !virtqueue_is_broken(out_vq))
                        cpu_relax();
        } // __send_to_port
}

static bool virtqueue_kick_prepare_split(struct virtqueue *_vq) {
        if (vq->event // false) { .. } else {
                needs_kick = !(vq->split.vring.used->flags & cpu_to_virtio16(_vq->vdev,
                                VRING_USED_F_NO_NOTIFY)); = // !false, so it's true
        }
        return needs_kick;
}
```

7.6.3 RLSoC output console

RLSoC has two RISC-V processors: one that implements priviledged instructions (m_RVCoreM verilog module, referred as “the processor”) and one that implements only unpriviledged instructions (m_RVuc verilog module, referred as “the microcontroller”).

The microcontroller runs raw RISC-V code compiled with the **riscv32-unknown-elf-gcc** compiler which is distributed along with the book sources. “The processor” uses **riscv32-buildroot-linux-gnu-gcc** build by buildroot.

The verilog variable **r_mc_mode** establishes what processor has access to rlsoc resources:

- if **r_mc_mode** is null, then “the processor” has grant access and the microcontroller waits;
- when **r_mc_mode** is not null, then “the microcontroller” has grant access to the system and the processor waits.

This is achieved by the following:

```
// mmu outputs w_proc_busy to processor's w_busy signal
assign w_proc_busy = w_tlb_busy || w_mc_busy || w_dram_busy || !w_tx_ready;
wire w_mc_busy = (r_mc_mode != 0) ? 1 : 0;

m_RVuc mc(CLK, .RST_X(r_mc_mode!=0), w_dram_busy, w_mc_addr, w_mc_arg,
                w_mc_wdata, w_mc_we, w_mc_ctrl, w_mc_aces);
```

The microcontroller signalizes using **r_mc_done** signal (via writing to `TOHOST_ADDR=32'h40008000 address the value `CMD_POWER_OFF=2) that has done its job and it goes to sleep: **r_mc_mode** becomes 0. See listing below.

When the microcontroller wants to send a character on the uart, it writes at address `TOHOST_ADDR the value (`CMD_PRINT_CHAR << 16 | c).

```
// mmu.v
if(r_tohost[31:16]==`CMD_PRINT_CHAR) begin
            r_uart_we   <= 1;
            r_uart_data <= r_tohost[7:0];..
        end else begin
            r_uart_we   <= 0;
            r_uart_data <= 0;
         end
if(r_tohost[31:16]==`CMD_POWER_OFF)
            r_mc_done <= 1;
r_tohost <= (w_mem_paddr==`TOHOST_ADDR && w_mem_we) ? w_mem_wdata :
                  (r_tohost[31:16]==`CMD_PRINT_CHAR) ? 0 : r_tohost;
```

Listing 7.5.2.3-2. CMD_PRINT_CHAR and CMD_POWER_OFF in mmu.v

In verilog, when one of the variables {w_cons_req, w_disk_req, w_key_req} is 1, then **r_mc_mode** becomes “not null” having a value encoded by the system which corresponds to a job to be executed by the microcontroller.

RLSoC has only two console queues: one for input and other for output. The console code is implemented in **console.v**. Here is what we are interested in for the output console (see the comments in the listing).

```
// mmu.v
wire  [3:0] w_dev       = w_mem_paddr[31:28];// & 32'hf0000000;
wire  [3:0] w_virt      = w_mem_paddr[27:24];// & 32'h0f000000;
wire [27:0] w_offset    = w_mem_paddr & 28'h7ffffff;

wire w_cons_we = (r_mc_mode != 0) ?
                    (w_mem_we && w_mem_paddr[31:12] == 20'h4000a) :
                    (w_mem_we && !w_tlb_busy && w_dev == `VIRTIO_BASE_TADDR
                    /*=4'h4*/ && w_virt == 0);
wire [31:0] w_cons_addr = (r_mc_mode != 0) ? w_mem_paddr : {4'b0,w_offset};

// mmu.v
m_console   console(CLK, 1'b1, w_cons_we, w_cons_addr, w_mem_wdata,
                    plic_pending_irq, w_cons_data,
                    w_cons_irq, w_cons_irq_oe, r_mc_mode,
                    (w_cons_req, w_cons_qnum, w_cons_qsel), w_key_req);

// console.v
module m_console    (CLK, RST_X, w_we, w_addr_t, w_idata,
                    w_iirq, w_odata /*=r_rdata*/,
                    w_oirq, w_oeirq, w_mode,
                    output(w_req, w_qnum, w_qsel), w_keyreq);

wire [7:0] w_addr = w_addr_t[7:0];

/* when w_req=1, (w_cons_req=1) and r_mc_mode becomes 1, r_mc_qnum=w_qnum=1
and r_mc_qsel=w_qsel=1. In this moment, linux made a print request. */
// VIRTIO_MMIO_QUEUE_NOTIFY=50 in tinyemu virtio.c and kernel virtio_mmio.h
assign w_req  = (w_mode==0 && w_we && w_addr == 32'h50 && w_idata == 1);
assign w_qnum = (w_keyreq) ? QueueNum : w_idata; //=1 ← (w_cons_req=1)
assign w_qsel = w_idata; //=1 ← (w_cons_req=1)

reg  [31:0] QueueSel                = 0; /* currently selected queue */
reg  [31:0] Queue[0:(`CONSOLE_QUEUE_NUM_MAX * 9) -1]; // [0:17]

if(w_mode==1 || w_mode==3) begin
   // Microcontroller accesses console
   // $clog2 is the ceiling of the logarithm in base 2.
   /*         Queue[w_addr[5+1:2]]. The first two bits from w_addr are
dropped because Queue[i] is 4 bytes and the microcontroller thinks memory
is byte addressable. */
   r_rdata <= Queue[w_addr[$clog2(`CONSOLE_QUEUE_NUM_MAX*9)+1:2]];
   if(w_we)
     Queue[w_addr[$clog2(`CONSOLE_QUEUE_NUM_MAX*9)+1:2]] <= w_idata;
end else
if(w_mode==0) begin
   // CPU Access ..
   case (w_addr)       // READ
            32'h0    : r_rdata <= MagicValue;
            32'h4    : r_rdata <= Version;
            32'h8    : r_rdata <= DeviceID;
            32'hc    : r_rdata <= VendorID;…
   if(w_we) begin
        case (w_addr)
```

```
            32'h30  : QueueSel                    <= w_idata;
            32'h44  : Queue[QueueSel*9]           <= w_idata;  // Ready
            32'h50  : begin
                        Queue[QueueSel*9+1] <= w_idata;        // Notify
                          InterruptStatus <= InterruptStatus | 1;
            end
            32'h80  : Queue[QueueSel*9+2]         <= w_idata;  // DescLow
            32'h90  : Queue[QueueSel*9+4]         <= w_idata;  // AvailLow
            32'ha0  : Queue[QueueSel*9+6]         <= w_idata;  // UsedLow ..


// mmu.v
always@(*) begin
  case (r_dev /* w_dev */)
  ..                    // r_data_data goes to processor
  `VIRTIO_BASE_TADDR: r_data_data <= (r_virt /* w_virt */ == 0) ?
                                       w_cons_data : w_disk_data;
  default           : r_data_data <= w_dram_odata;
  endcase
end
```

Note that AvailLow, UsedLow and DescLow are 32 bit addresses given by Linux, and in **console.v**, the AvailHigh, UsedHigh and DescHigh are not used because we have a 32 bit processor.

w_qnum is the queue size for the currently selected queue. For RLSoC we have w_qnum=1 to output console and w_qnum=2 for input console (the last one is set by the linux kernel 5.13.19).

Let's see now how the microcontroller behaves. First, the verilog code. Please read the comments in code.

```
// mmu.v
wire [31:0] w_mem_paddr  = (r_mc_mode != 0) ? w_mc_addr : ..
case (w_mem_paddr[31:12])
            20'h40009: begin
                case (w_mem_paddr[3:0])
                    4: r_mc_arg <= r_mc_qnum; // w_mc_qnum
                    8: r_mc_arg <= r_mc_qsel; // w_mc_qsel
                    default: r_mc_arg <= r_mc_mode;
                endcase
            end
            20'h4000a: r_mc_arg <= w_cons_data;
            20'h4000b: r_mc_arg <= w_disk_data;
            20'h4000c: r_mc_arg <= cons_fifo[r_consf_head];
            default:   r_mc_arg <= w_dram_odata;
        endcase
// w_dram_le implies memory read
wire w_dram_le    =  (w_dram_busy)  ? 0 : ..
                     (r_mc_mode!=0) ? (w_mc_aces==`ACCESS_READ &&
                                       w_mc_addr[31:28] != 0) : ..

m_RVuc mc(CLK, (r_mc_mode!=0), w_dram_busy, w_mc_addr, w_mc_arg,
          w_mc_wdata,
          w_mc_we, w_mc_ctrl, w_mc_aces);

// microc.v
```

```
module m_RVuc(CLK, RST_X, w_stall, (out) w_mic_addr, (in) w_data=r_mc_arg,
           (out) w_mic_wdata,
            w_mic_mmuwe, w_mic_ctrl, w_mic_req→w_mc_aces);

// load
assign w_mic_req = (r_state==`MC_IF)                                ?
`ACCESS_CODE   :
                   (r_state==`MC_EX && w_we)                          ?
`ACCESS_WRITE :
                   (r_state==`MC_EX && r_opcode==`OPCODE_LOAD    )   ?
`ACCESS_READ   : 3;

// r_dram_data = w_data = r_mc_arg.
assign w_mem_rdata  = (w_mic_addr[31:28]==0) ? w_lcm_data : r_dram_data;
wire [31:0] w_reg_d  = (r_opcode==`OPCODE_LOAD    ) ? w_mem_rdata : ..;
// w_reg_d will be written in microcontroller's register file m_regfile.

// store
assign w_mic_mmuwe  = w_we && (w_mic_addr[31:28]!=0);
wire           w_we = (r_opcode==`OPCODE_STORE    && r_state==`MC_EX);
```

The microcontroller's program is loaded in its memory:

```
`define D_UC_LM_IFILE    "ucimage.hex"
initial begin
        $readmemh(`D_UC_LM_IFILE, mem); ..
```

Now let's analyze the microcontroller program code written in C and assembler. It is available in the **src/ucimage** folder.

The most important source is **main.c**. Let's see it from the console out point of view.

```
int main(){

   // Mode (1: Console, 2: Disk). This is r_mc_mode in verilog.
   uint32_t *MODE            = (uint32_t*)0x40009000;  uint32_t mode = *MODE;
   uint32_t* QNUM;  QNUM = (uint32_t*)0x40009004;  int qnum = *QNUM;
   uint32_t* QSEL;       QSEL = (uint32_t*)0x40009008;  int idx = *QSEL;

   // Queues for Console
   QueueState *CONS_Q;  CONS_Q = (QueueState *)0x4000a000; ..
   QueueState* tc_queue = (idx==0) ? &CONS_Q[0] : &CONS_Q[1];

   if(mode == 1) {
      // qnum=idx=1 tc_queue = &CONS_Q[1].
      cons_request(0, qnum, tc_queue);
   } ..

   simrv_exit();
}
```

CONS_Q is of type QueueState. This type is defined as follows. So, we have 8 fields of 4 octets and 1 field of 2 octets; however, the gcc compiler will reserve 36 bytes for QueueState. We have 2 queues, each of 9 words of 4 bytes: one for console in which corresponds to CONS_Q[0] and one for console out which corresponds to CONS_Q[1]. Every field from QueueState corresponds to one word (of 4 bytes) of the Queue[0:17] vector from **console.v**.

```
typedef struct QueueState {
   uint32_t Ready;
   uint32_t Notify;
   uint32_t DescLow;
   uint32_t DescHigh;
   uint32_t AvailLow;
   uint32_t AvailHigh;
   uint32_t UsedLow;
   uint32_t UsedHigh;
   uint16_t last_avail_idx;
}QueueState;
```

In **main.c** we also have the **Descriptor** type, which corresponds to the vring_desc_t from linux which we have seen earlier.

```
/* vring_desc_t from linux */
typedef struct Descriptor {
   uint64_t adr;
   uint32_t len;
   uint16_t flags;
   uint16_t next;
}Descriptor;
```

Now let's see the cons_request() function. It loops and at each iteration fills up **desc**, from RAM, based on **qs->AvailLow** and **qs->DescLow** and prints the characters starting at **desc.addr** address. The print is made by the function **simrv_putc()**. Then it calls the **update_descriptor()** function to update the used ring.

The microcontroller uses the addresses set by linux. In order to ease the understanding process, please read the struct vring_virtqueue from linux, especially the field **split.vring** (of struct **vring** type defined in include/uapi/linux/**virtio_ring.h**).

The functions **ram_ld()** and **ram_st()** are used to load and respectively store data words of specified sizes from and into RAM.

```
#define DESC_SIZE 16 /* Descriptor type size is 16 bytes */
/*** console_request for display output                    ****/
void cons_request(uint8_t *mmem/*=0*/, uint32_t q_num/*=1*/, QueueState *qs){
   Descriptor desc;
   uint8_t *p;
```

```
   disk_debug_num++; /* just for debug */

   /* vring_avail_t is struct {__virtio16 flags, idx, ring[];} */
   uint16_t avail_idx = (uint16_t)ram_ld(qs->AvailLow+2, 2, mmem);

   while (qs->last_avail_idx != avail_idx) {

      /* sizeof(__virtio16)=2 and avail.ring[] is of type __virtio16. offsetof(vring_avail_t,
ring)=4. */
     /* qs->last_avail_idx initially is 0, like all fields of Queue[], and is saved in
Queue[QueueSel*9+8] in console.v*/
      uint32_t adr = qs->AvailLow + 4 + (qs->last_avail_idx & (q_num - 1)) * 2;
      uint16_t desc_idx_header = ram_ld(adr, 2, mmem);
      uint32_t desc_adr_header = desc_idx_header * DESC_SIZE + qs->DescLow;

      p = (uint8_t*)&desc;
      for(int i=0; i<DESC_SIZE; i++){ *p = ram_ld(desc_adr_header+i, 1, mmem); p++; }

      for(int i=0; i<(int)desc.len; i++){ /***** write to stdout *****/
         uint8_t d = ram_ld(desc.adr+i, 1, mmem);
         simrv_putc(d);
      }

      update_descriptor(desc_idx_header, 0, q_num, qs, mmem);
      /* Verilog console.v stores this incrementation result. */
      qs->last_avail_idx++;
   }
}
```

The update_descriptor() function is shown below. Please see the comments in code.

```
/*** update the used ring                                                    ****/
void update_descriptor(uint32_t desc_idx, uint32_t desc_len, int q_num,
                  QueueState *qs, uint8_t *mmem){
   /* vring_used_t is struct {__virtio16 flags, idx;  vring_used_elem_t {__virtio32 id, len;}
ring[];}.*/
   uint32_t addr_used_idx = qs->UsedLow + 2;
   uint32_t index = (uint16_t)ram_ld(addr_used_idx, 2, mmem);

   ram_st(addr_used_idx, index+1, 2, mmem);

   // sizeof(vring_used_elem_t)=8. offsetof(vring_used_t, ring)=4.
   uint32_t addr_used_entry = qs->UsedLow + 4 + (index & (q_num - 1)) * 8;
   ram_st(addr_used_entry,   desc_idx, 4, mmem);
   ram_st(addr_used_entry+4, desc_len, 4, mmem);
}
```

The simrv_putc() implemented in **simrv.c** is shown below. Please see listing 7.5.2.3-2 to see how a char is sent to the uart transmitter in **mmu.v**. Note that after giving the command “send the character” to the uart, the microcontroller must wait a given amount of time necessary for the uart to send it on the phy.

```
volatile int *TOHOST_ADDR = (int *)0x40008000;
void simrv_putc (char c) {
   *TOHOST_ADDR = CMD_PRINT_CHAR << 16 | c;
   // wait the uart to put the char on the txd line, for 8Mbps: 10bits*13cycles/bit=130 cpu
cycles
   sleep_fct(100);
}
```

7.6.4 Linux input console

Returning to **virtcons_probe**(), it calls **init_vqs**() which initiates **io_callbacks**[] to **in_intr**() and **out_intr**(); **in_intr**() pushes the pending writes and calls **get_chars**().

Then **init_vqs**() → **virtio_find_vqs**() → **vm_find_vqs**() → **request_irq**(vm_interrupt).

vm_interrupt() clears InterruptStatus of **console.v** and calls **vring_interrupt**() which calls the callbacks **in_intr**() and **out_intr**().

```
virtcons_probe(struct virtio_device *vdev)
{
..err = init_vqs(portdev); {
                vq_callback_t **io_callbacks;
                io_callbacks = kmalloc_array(nr_queues, sizeof(vq_callback_t *),
                /*
                 * For backward compat (newer host but older guest), the host
                 * spawns a console port first and also inits the vqs for port
                 * 0 before others.
                 */
                ..j = 0;
                io_callbacks[j] = in_intr; // void in_intr(struct virtqueue *vq) {
                        port = find_port_by_vq(vq->vdev->priv, vq);
                        port->inbuf = get_inbuf(port);
                        /*Console ports are hooked up with an HVC console and is initialized
                with guest_connected to true.*/ ..
                        if (is_console_port(port)/*==port->cons.hvc*/ &&
                                hvc_poll(port->cons.hvc)  /*__hvc_poll(struct hvc_struct *
                                hp, may_sleep=false); */) {
                                        Uses tty, ..
                                        /* Push pending writes */
                                        if (hp->n_outbuf > 0) written_total = hvc_push(hp);
                                        ..tty = tty_port_tty_get(&hp->port);
                                        /* read data if any */
                                        count = tty_buffer_request_room(&hp->port,
                        N_INBUF);
                                        /* If flip is full, just reschedule a later read */
                                        if (count == 0) {
                                                poll_mask |= HVC_POLL_READ;
                                                goto out;
                                        }
                                        n = hp->ops->get_chars(hp->vtermno, buf, count);
                                        ..return poll_mask;
                                }
                                hvc_kick();
                } // in_intr()
```

```
                    // cont from init_vqs
                    io_callbacks[j + 1] = out_intr; ..
                    // cont from init_vqs
                    ..err = virtio_find_vqs(portdev->vdev, nr_queues, vqs, io_callbacks,
io_names, NULL);=return vdev->config->find_vqs(vdev, nvqs, vqs, callbacks, names, NULL,
desc);=vm_find_vqs(*vdev, nvqs, *vqs[], *callbacks[], names[], const bool *ctx, *desc) {
                            struct virtio_mmio_device *vm_dev= to_virtio_mmio_device(vdev);
                            int irq = platform_get_irq(vm_dev->pdev, 0); // 1=console, 2=disk
                            err = request_irq(irq, vm_interrupt, IRQF_SHARED,
                                        dev_name(&vdev->dev), vm_dev);  vm_interrupt {
                                    /* Read and acknowledge interrupts => clear int from hw */
                                    status = readl(vm_dev->base +
VIRTIO_MMIO_INTERRUPT_STATUS/*0x60*/);
                                    writel(status, vm_dev->base +
VIRTIO_MMIO_INTERRUPT_ACK/0x64*/);
                                    ..if (likely(status & VIRTIO_MMIO_INT_VRING /*1*/)) {
                                            ..list_for_each_entry(info, &vm_dev->virtqueues,
node)
                                                    ret |= vring_interrupt(irq, info->vq); {
                                                        if (vq->vq.callback) /* in_intr(), out_intr()*/
                                                                vq->vq.callback(&vq->vq);
                                                        return IRQ_HANDLED;
                                                    } // vring_interrupt
                                                return ret;
                                    } // if
                            } // vm_interrupt
            ...
}
```

The functionality of **get_chars**() is presented below. It prepares a descriptor to store the incoming data.

```
/* The operations for console ports. */
static const struct hv_ops hv_ops = {
.get_chars = get_chars(u32 vtermno, char *buf, int count) {
   struct port *port = find_port_by_vtermno(vtermno);
   return fill_readbuf(port, (__force char __user *)buf, out_count=count, to_user=false); {
        if (!out_count || !port_has_data(port)) return 0;
        struct port_buffer buf = port->inbuf;
        out_count = min(out_count, buf->len - buf->offset);
        if (to_user) ret = copy_to_user(out_buf, buf->buf + buf->offset, out_count);
        else memcpy((__force char *)out_buf, buf->buf + buf->offset, out_count);
        buf->offset += out_count;
        if (buf->offset == buf->len) {
                /* We're done using all the data in this buffer.
                * Re-queue so that the Host can send us more data.
                add_inbuf(port->in_vq, buf); {
```

```
                        sg_init_one(sg, buf->buf, buf->size); // link sg to buf
                        ret = virtqueue_add_inbuf(vq, sg, num=1, data=buf,
                        GFP_ATOMIC);=virtqueue_add(vq, &sg, total_sg=num, 0, 1, data,
                        NULL, gfp);=virtqueue_add_split(_vq, sgs, total_sg, out_sgs=0,
                        in_sgs=1, data, ctx, gfp);= like put_chars() with the difference {..
                          for (n = 0; n < out_sgs; n++) ..// 0 iterations
                          for (; n < (out_sgs + in_sgs); n++) {
                                for (sg = sgs[n]; sg; sg = sg_next(sg)) {
                                      dma_addr_t addr = vring_map_one_sg(vq, sg,
                                      DMA_FROM_DEVICE);
                                      ..
                                      desc[i].flags = cpu_to_virtio16(_vq->vdev,
                                      VRING_DESC_F_NEXT | VRING_DESC_F_WRITE);
                                      desc[i].addr = cpu_to_virtio64(_vq->vdev, addr);
                                      desc[i].len = cpu_to_virtio32(_vq->vdev, sg->length);
                                      prev = i;
                                      i = virtio16_to_cpu(_vq->vdev, desc[i].next);
                                }
                          }...
                        } // virtqueue_add_split
                        virtqueue_kick(vq); // see it at put_chars() subchapter.
                  } // add_inbuf
            } // if
            /* Return the number of bytes actually copied */     return out_count;
      } // fill_readbuf
}
```

7.6.5 RLSoC input console

When the flag SIM_MODE is enabled, it adds to the following, the **read_file** module; the **read_file** module reads “fid.txt” and when it has a new id, the contents of “fcmd.txt” (stored in **rf.fifo**) will be appended to the **cons_fifo** variable.

Please read the chapter Simulating RLSoC for SIM_MODE.

The rlsoc code that grabs a key that was pressed is presented in the following listing. **w_key_req** depends on **r_consf_en** and **r_mc_mode**.

```
// mmu.v
PLOADER ploader(CLK, RST_X, w_rxd, w_pl_init_addr, w_pl_init_data,
                w_pl_init_we, w_pl_init_done, w_key_we /*!*/, w_key_data);

reg   [7:0] cons_fifo [0:15];

always@(posedge CLK) begin
        if(r_mc_done) begin
            r_mc_mode <= 0;
            r_mc_done <= 0;
            if(r_mc_mode==3) begin
                r_consf_en <= (r_consf_cnts<=1) ? 0 : 1;
                r_consf_head <= r_consf_head + 1;
                r_consf_cnts <= r_consf_cnts - 1;
            end
        end
        else if(r_key_we /*w_key_we*/) begin
            if(r_consf_cnts < 16) begin
                cons_fifo[r_consf_tail] <= r_key_data;
                r_consf_tail            <= r_consf_tail + 1;
                r_consf_cnts            <= r_consf_cnts + 1;
                r_consf_en              <= 1;
            end
        end else begin
            case ({w_cons_req, w_disk_req, w_key_req})
                ..
                3'b001: begin
                    r_mc_mode <= 3;
                    r_mc_qnum <= w_cons_qnum;
                    r_mc_qsel <= 0;
                end
            endcase
        end
end

wire w_key_req = r_consf_en &&
                 (w_mtime > `ENABLE_TIMER + 64'd1000000) &&
                 ((w_mtime & 64'h3ffff) == 0) &&
                 r_mc_mode == 0 && w_init_stage;
// rvcorem.v
always@(posedge CLK) r_init_stage <= (w_state==`S_INI);
assign w_init_stage = r_init_stage;
assign w_state = (!RST_X | r_halt) ? 0       : // `S_INI
                 (w_com) ? `S_COM : (w_busy || w_ex1_busy) ? state  :
                 (state==`S_FIN)? `S_INI :
                 (state==`S_IF  && w_nalign4 && r_if_state!=3) ? `S_IF : ..
```

The microcontroller reads the keys that were pressed with the following logic:

```
// mmu.v
wire [31:0] w_mem_paddr  = (r_mc_mode != 0) ? w_mc_addr : ..
case (w_mem_paddr[31:12])
            ..
            20'h4000c: r_mc_arg <= cons_fifo[r_consf_head];
```

In **console.v**, we have the following regarding the input console interrupts. We will see more about them in the Interrupts subchapter.

```
// mmu.v
wire [31:0] w_virt_irq  = (w_key_req) ? w_cons_irq : (w_isdisk) ?
                                                  w_disk_irq : w_cons_irq;
wire    w_virt_irq_oe   = w_cons_irq_oe | w_disk_irq_oe | w_key_req;


m_console   console(CLK, 1'b1, w_cons_we, w_cons_addr, w_mem_wdata,
                    plic_pending_irq, w_cons_data,
                    w_cons_irq, w_cons_irq_oe, r_mc_mode,
                    (w_cons_req, w_cons_qnum, w_cons_qsel), w_key_req);

// console.v
module m_console    (CLK, RST_X, w_we, w_addr_t, w_idata,
                     w_iirq, w_odata /*=r_rdata*/,
                     w_oirq, w_oeirq, w_mode,
                     output(w_req, w_qnum, w_qsel), w_keyreq);

assign w_qnum = (w_keyreq) ? QueueNum : w_idata;

// w_oeirq is used to clear the interrupt line (used with w_oirq=w_oirq2).
assign w_oeirq = (w_mode==0) && w_we && (w_addr==32'h64
                                         /*=VIRTIO_MMIO_INTERRUPT_ACK*/);
// Console IRQ
    wire [31:0] w_irqmask = 1 << (`VIRTIO_CONSOLE_IRQ /*1*/ -1);
    wire [31:0] w_oirq1   = w_iirq |  w_irqmask;
    wire [31:0] w_oirq2   = w_iirq & ~w_irqmask;
    /* when w_key_req=1, w_irqmask is propagated to w_virt_irq and
                                                    w_virt_irq_oe is 1 */
    assign      w_oirq    = (w_keyreq) ? w_oirq1 : (w_addr==32'h64) ?
                                                    w_oirq2 : w_oirq1;


if(w_keyreq) begin
            InterruptStatus <= InterruptStatus | 1;
end

// CPU Access
if(w_mode==0) begin ..
    if(w_we) begin
        case (w_addr) ..
             32'h64  : begin
                        InterruptAcknowledge <= w_idata;
                        InterruptStatus <= InterruptStatus & ~w_idata;
              end
```

The microcontroller **main**() code will call **kbd_request**() similar to the call of **cons_request**().

```
int main()
{
   // Keyboard input buffer
   uint8_t *CONS_FIFO = (uint8_t*)0x4000c000;
   int cons_fifo = *CONS_FIFO;..

   if(mode == 3) {
      // qnum=2
      kbrd_request(0, qnum, &CONS_Q[0], cons_fifo);
    }..
}
```

The microcontroller code of **kbd_request**() has many similarities to **cons_request**(). The difference is that it stores in the available descriptor the contents of **buf** and then it marks the descriptor as used.

```
void kbrd_request(uint8_t *mmem, uint32_t q_num, QueueState *qs, uint8_t buf){
   Descriptor desc;
   uint8_t *p;

   if (!qs->Ready) return;

   /* vring_avail_t is struct {__virtio16 flags, idx, ring[];} */
   uint16_t avail_idx = (uint16_t)ram_ld(qs->AvailLow+2, 2, mmem);
   if (qs->last_avail_idx == avail_idx) return;

   // sizeof(__virtio16)=2 and ring[] is of type __virtio16. offsetof(vring_avail_t, ring)=4.
   uint32_t adr = qs->AvailLow + 4 + (qs->last_avail_idx & (q_num - 1)) * 2;
   uint16_t desc_idx_header = ram_ld(adr, 2, mmem);
   uint32_t desc_adr_header = desc_idx_header * DESC_SIZE + qs->DescLow;

   p = (uint8_t*)&desc;
   for(int i=0; i<DESC_SIZE; i++){ *p = ram_ld(desc_adr_header+i, 1, mmem); p++; }

   ram_st(desc.adr, (uint32_t)buf, 1, mmem);

   update_descriptor(desc_idx_header, 1, q_num/*2*/, qs, mmem);
   qs->last_avail_idx++;
}
```

7.7 Interrupts

7.7.1 Interrupts in RVCore

In the processor (the m_RVCoreM module), **w_ir** represents the interrupt. Compressed instructions have 2 bytes while normal instructions have 4 bytes. Usually, **w_ir** = **w_insn_data**, which comes from mmu. The various instructions operands are shown in the next listing.

```
// rvcorem.v
wire w_cinsn = (w_ir_org[1:0] != 2'b11); // is it a compressed instruction?
wire [31:0] w_ir = (w_nop) ? `RV32_NOP : (w_cinsn) ? w_ir_t : w_ir_org;
wire [31:0] w_ir_org = (r_if_state == 3) ? (r_ir16_v) ?
  {w_insn_data[15:0], r_ir16} : {w_insn_data[15:0], r_if_irl} :
                                                  w_insn_data;

always @(posedge CLK) if(state == `S_ID) begin
        r_rrs1    <= w_rrs1;      // operand read from register file
        r_rrs2    <= w_rrs2;      // operand read from register file
        r_opcode  <= r_ir[ 6: 0]; // r_ir = w_ir
        r_rd      <= r_ir[11: 7];
        r_rs1     <= r_ir[19:15];
        r_rs2     <= r_ir[24:20];
        r_funct3  <= r_ir[14:12];
        r_funct5  <= r_ir[31:27];
        r_funct7  <= r_ir[31:25];
        r_funct12 <= r_ir[31:20];
        r_imm     <= w_imm;
    end
```

Interrupts are generated from the external devices, while exceptions are signalized as a follow up to an event happening inside the processor-memory system.

The logic behind **pending_exception** and **r_tkn** and **r_jmp_pc** is abstracted in the listing below. When **pending_exception** is ~0 then no exception occurred. The **pending_exception** variable is set when we have a page fault, an interrupt (CAUSE_INTERRUPT = 32'h80000000) or a user ECALL instruction. **r_tkn** and **r_jmp_pc** are used by branch or jump instructions and contain the decision (was the branch taken?) and the new program counter.

```
// mmu.v
assign w_pagefault = !page_walk_fail ? ~32'h0 : (w_iscode) ?
`CAUSE_FETCH_PAGE_FAULT : (w_isread) ? `CAUSE_LOAD_PAGE_FAULT :
`CAUSE_STORE_PAGE_FAULT;

// rvcorem.v
reg  [31:0] pending_exception = ~0;
always @(posedge CLK) begin
        if(w_pagefault != ~0) begin
            pending_exception <= w_pagefault;
        end else if(state == `S_INI) begin
```

```
            pending_exception <= ~0;
            r_tkn             <= 0;
        end else if(state == `S_FIN && !w_busy) begin
            if(w_interrupt_mask != 0) begin
                pending_exception <= `CAUSE_INTERRUPT | irq_num;
            end
        end else if(state == `S_EX1) begin
            case(r_opcode)..
                `OPCODE_JAL      : begin
                    r_tkn        <= 1;
                    r_wb_data    <= (r_cinsn) ? pc+2 : pc+4;
                    r_jmp_pc     <= pc + r_imm;
                end..
                `OPCODE_SYSTEM   : begin
                    if(r_funct3 == `FUNCT3_PRIV  ) begin
                        case (r_funct12)
                            `FUNCT12_ECALL  : begin
                                r_wb_data_csr  <= `CAUSE_USER_ECALL + priv;
                                pending_exception <= `CAUSE_USER_ECALL +
priv;
                            end..
                            `FUNCT12_MRET   : begin
                                r_tkn        <= 1;
                                r_jmp_pc     <= r_rcsr;
                            end..
            endcase
        end
end // always
```

I recall that the machine trap (exception and interrupt) delegation registers **medeleg** and **mideleg** must exist in systems with S-mode, and "setting a bit in **medeleg** or **mideleg** will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. When a trap is delegated to S-mode, the **scause** register is written with the trap cause" [13]. Note that the bbl configures the system to send S-mode interrupts and most exceptions straight to S-mode, by calling the bbl **delegate_traps**() function.

The **cause** variable has **cause**[31]=1 for interrupts. **w_deleg** establishes wheter the trap is delegated to the S-mode.

```
Wire [31:0] pending_interrupts = mip & mie;
wire [31:0] enable_interrupts  = (pending_interrupts) ?
(r_priv_t == `PRIV_M) ? ((w_mstatus_nxt & `MSTATUS_MIE) ? ~mideleg : 0) :
(r_priv_t == `PRIV_S) ? ((w_mstatus_nxt & `MSTATUS_SIE) ? (~mideleg |
mideleg) : ~mideleg) :
(r_priv_t == `PRIV_U) ? ~0 : 0 : 0;

assign w_interrupt_mask = pending_interrupts & enable_interrupts;
// w_irq_t is the first '1' bit of w_interrupt_mask.
assign w_irq_t          = w_interrupt_mask & (~w_interrupt_mask+1);
if(state == `S_COM) begin
            case (w_irq_t)
                32'h00000001: irq_num <= 0;
                32'h00000002: irq_num <= 1;..
            endcase
end
```

```
wire [31:0] cause = (pending_exception != ~0) ? pending_exception :
(`CAUSE_INTERRUPT | irq_num);
wire [31:0] w_deleg = (r_priv_t <= `PRIV_S) ?
((cause & `CAUSE_INTERRUPT) ? (mideleg >> (cause & 32'h1f))  & 1 :
                              (medeleg >> (cause & 32'h1f))) & 1 : 0;
```

Now, let's see how the control is transferred to the trap handler. If a trap has occurred and **w_deleg** = 1, then the processor will handle it in S mode; otherwise the trap will be handled in M mode. Note that **mret** sets MIE to MPIE in **mstatus** (and **sret** similar).

```
if(state == `S_FIN && !w_busy) begin
            if(pending_exception != ~0) begin
                if(w_deleg) begin
                    scause  <= cause;
                    sepc    <= pc;
                    stval   <= pending_tval;
                    mstatus <= w_sstatus_t3; // unsets MSTATUS_SIE
                    priv    <= `PRIV_S;
                end else begin
                    mcause  <= cause;
                    mepc    <= pc;
                    mtval   <= pending_tval;
                    mstatus <= w_mstatus_t3; // unsets MSTATUS_MIE
                    priv    <= `PRIV_M;
                end
            end
            else if(w_interrupt_mask != 0) begin
                if(w_deleg) begin
                    scause  <= cause;
                    sepc    <= (r_tkn) ? r_jmp_pc : (r_cinsn)? pc+2 : pc+4;
                    stval   <= pending_tval;
                    mstatus <= w_sstatus_t3;
                    priv    <= `PRIV_S;
                end else begin
                    mcause  <= cause;
                    mepc    <= (r_tkn) ? r_jmp_pc : (r_cinsn)? pc+2 : pc+4;
                    mtval   <= pending_tval;
                    mstatus <= w_mstatus_t3;
                    priv    <= `PRIV_M;
                end
            end
        end
```

Finaly, the next listing shows the logic to update the program counter. Note that we have described above the variables **r_tkn** and **r_jmp_pc**. Note that RVCore uses direct mode for **stvec** and **mtvec**.

```
if(pending_exception != ~0)  begin pc <= (w_deleg) ? stvec : mtvec; end
else begin
   if(w_interrupt_mask != 0) begin pc <= (w_deleg) ? stvec : mtvec; end
   else begin pc <= (r_tkn) ? r_jmp_pc : (r_cinsn) ? pc + 2 : pc + 4; end
end
```

7.7.2 RLSoC PLIC

PLIC stands for Platform-Level Interrupt Controller.

As we have seen, the input console generates interrupts. At first key pressed after ENABLE_TIMER, **w_key_req**<= 1, **w_virt_irq** <= 1 and **w_virt_irq_oe** <= 1. As a follow up, **w_plic_we** <= 1 and w_wmip <= w_mip | (`MIP_MEIP | `MIP_SEIP). Recall that MIP_MEIP and MIP_SEIP are machine/supervisor-level external interrupts. Linux instructs the processor to read from PLIC_BASE_TADDR + 4, which is **r_plic_odata** and is equal to **w_plic_mask** (in our case VIRTIO_CONSOLE_IRQ).

Linux acknowledges the interrupt by writing to the console base address + VIRTIO_MMIO_INTERRUPT_ACK, so **w_cons_irq_oe** <= 1 and **w_virt_irq** <= 0. Wire **w_plic_pending_irq_next** <= 0. At next clock posedge, **r_plic_pending_irq_t** <= 0 and **r_virt_irq_oe_t** <= 1. So, **wplic_we** = 1, **w_plic_mask_nxt** = 0 and MIP_MEIP and MIP_SEIP will be cleared.

The system can also make use of **plic_served_irq** to clear MIP_MEIP and MIP_SEIP for handled interrupts. More on this will see in the Linux interrupts chapter.

```
// mmu.v
always@(*) begin
        case (r_dev)
            `PLIC_BASE_TADDR  : r_data_data <= r_plic_odata;..

wire [31:0] w_virt_irq  = (w_key_req) ? w_cons_irq : (w_isdisk) ?
                                                    w_disk_irq : w_cons_irq;
wire    w_virt_irq_oe   = w_cons_irq_oe | w_disk_irq_oe | w_key_req;

if(w_virt_irq_oe) plic_pending_irq    <= w_virt_irq;
wire [31:0] w_plic_pending_irq_nxt  =   w_virt_irq_oe ? w_virt_irq :
                                                    plic_pending_irq;
wire [31:0] w_plic_served_irq_nxt   =   w_virt_irq_oe ? plic_served_irq :
                                (w_isread) ? plic_served_irq | w_plic_mask :
                                plic_served_irq & ~(1 << (w_data_wdata-1));
wire        w_isread                = (w_tlb_req == `ACCESS_READ);
wire [31:0] w_plic_mask     = w_plic_pending_irq_nxt & ~plic_served_irq;
wire [31:0] w_plic_mask_nxt = r_plic_pending_irq_t & ~r_plic_served_irq_t;

always@(posedge CLK) if(!w_tlb_busy) begin
            r_virt_irq_oe_t         <= w_virt_irq_oe;
            r_plic_aces_t           <= w_plic_aces;
            r_plic_pending_irq_t    <= w_plic_pending_irq_nxt;
            r_plic_served_irq_t     <= w_plic_served_irq_nxt;
end

wire w_plic_aces = (w_dev == `PLIC_BASE_TADDR/4'h5/ && !w_tlb_busy &&
            ((w_isread && w_plic_mask != 0) ||
             (w_iswrite && w_offset == `PLIC_HART_BASE+4)));

always@(posedge CLK) .. if(w_plic_aces) begin
            r_plic_odata    <= w_plic_mask; // it reduces to w_plic_mask
            plic_served_irq <= w_plic_served_irq_nxt;
end
```

```
assign w_plic_we     = (r_virt_irq_oe_t || r_plic_aces_t);
assign w_wmip        = (w_plic_mask_nxt) ? w_mip | (`MIP_MEIP | `MIP_SEIP) :
                                  w_mip & ~(`MIP_MEIP | `MIP_SEIP);
// w_wmip goes to the processor

// rvcorem.v
assign w_mip       = mip;
if(state == `S_IF) if(mtime > `ENABLE_TIMER) if(w_plic_we) mip <= w_wmip;
if(state == `S_EX2 || state == `S_WB) if(w_plic_we) mip <= w_wmip;
wire [31:0] pending_interrupts = mip & mie;
```

7.7.3 Linux interrupts

PLIC instantiation in the dts has different addresses in rlsoc than in tinyemu.

```
cpus {
 cpu0 {
  ..interrupt-controller {
                #interrupt-cells = <0x1>;
                interrupt-controller;
                compatible = "riscv,cpu-intc";
                phandle = <0x1>;
            };..
soc {..
 // rlsoc
 plic@50000000 {
            #interrupt-cells = <0x1>;
            interrupt-controller;
            compatible = "riscv,plic0";
            riscv,ndev = <0x1f>;
            reg = <0x0 0x50000000 0x0 0x8000000>;
            interrupts-extended = <0x1 0x9 0x1 0xb>;
            phandle = <0x2>;
        };..
 // tinyemu
 plic@40100000 {
            #interrupt-cells = <0x01>;
            interrupt-controller;
            compatible = "riscv,plic0";
            riscv,ndev = <0x1f>;
            reg = <0x00 0x40100000 0x00 0x400000>;
            interrupts-extended = <0x01 0x09 0x01 0x0b>;
            phandle = <0x02>;
        };..
```

Linux uses the macro IRQCHIP_DECLARE to declare riscv_plic0 irqchip driver which associates the dts compatible string “riscv,plic0” to the **plic_init**() function.

```
// drivers/irqchip/irq-riscv-intc.c
IRQCHIP_DECLARE(riscv, "riscv,cpu-intc", riscv_intc_init);

// drivers/irqchip/irq-sifive-plic.c
IRQCHIP_DECLARE(riscv_plic0, "riscv,plic0", plic_init);

// include/linux/irqchip.h
// This macro must be used by the different irqchip drivers to
declare the association between their DT compatible string and their
initialization function.
#define IRQCHIP_DECLARE(name, compat, fn) OF_DECLARE_2(irqchip,
```

```
name, compat, fn)

//include/linux/of.h
#define OF_DECLARE_2(table, name, compat, fn) \
                _OF_DECLARE(table, name, compat, fn, of_init_fn_2)

// this defines __of_table_riscv_plic0 as part of __irqchip_of_table
section (in arch/riscv/kernel/vmlinux.lds)
#if defined(CONFIG_OF) && !defined(MODULE)// true for us
#define _OF_DECLARE(table, name, compat, fn, fn_type)
        static const struct of_device_id __of_table_##name
                __used __section("__" #table "_of_table")          \
                __aligned(__alignof__(struct of_device_id))
                 = { .compatible = compat,                         \
                     .data = (fn == (fn_type)NULL) ? fn : fn  }
#else..

// drivers/irqchip/irqchip.c
extern struct of_device_id __irqchip_of_table[];

void __init irqchip_init(void)
{
        of_irq_init(__irqchip_of_table);
        acpi_probe_device_table(irqchip);
}
```

The function **of_irq_init**() is called in the chain start_kernel()→init_IRQ()→irqchip_init()→of_irq_init(__irqchip_of_table) and it inits the matching interrupt controllers in the device tree.

```
void of_irq_init(struct of_device_id *matches=__irqchip_of_table)
{
     struct of_intc_desc *desc, *temp_desc;
      // Scan and init matching interrupt controllers in DT
      for_each_matching_node_and_match(np, _matches, &match) {
            if (!of_property_read_bool(np, "interrupt-controller") || .. )
                  continue;
            // found: “riscv,cpu-intc”, “riscv,plic0”
            desc = kzalloc(sizeof(*desc), GFP_KERNEL);
            desc->irq_init_cb = match->data; // riscv_intc_init, plic_init
            desc->dev = of_node_get(np);
            desc->interrupt_parent = of_irq_find_parent(np);
      }
      ..ret = desc->irq_init_cb(desc->dev, desc->interrupt_parent);
}
```

riscv_intc_init()→set_handle_irq(&riscv_intc_irq); Their significance is shown in the following listing.

```
#ifdef CONFIG_GENERIC_IRQ_MULTI_HANDLER // y
int __init set_handle_irq(void (*handle_irq)(struct pt_regs *)) {
        handle_arch_irq = handle_irq;
}
```

```
//head.S:
setup_trap_vector: /* Set trap vector to exception handler */
la a0, handle_exception; csrw CSR_TVEC, a0

//riscv/include/asm/csr.h
# define CSR_CAUSE     CSR_SCAUSE

//entry.S:
ENTRY(handle_exception) ..
                csrr s4, CSR_CAUSE
                ..
                  bge s4, zero, 1f
                  la ra, ret_from_exception
                  /* Handle interrupts */
                  move a0, sp /* pt_regs */
                  la a1, handle_arch_irq
                  REG_L a1, (a1)
                  jr a1
            1: // exceptions

// irq-riscv-intc.c
static asmlinkage void riscv_intc_irq(struct pt_regs *regs)
{
        unsigned long cause = regs->cause & ~CAUSE_IRQ_FLAG;

        switch (cause) {
#ifdef CONFIG_SMP
        case RV_IRQ_SOFT:
                /* We only use software interrupts to pass IPIs, so if a
non-SMP system gets one, then we don't know what to do. */
                handle_IPI(regs);
                break;
#endif
        default:
                handle_domain_irq(intc_domain, cause, regs);=
  handle_domain_irq(domain, hwirq, true, regs); {
        struct pt_regs *old_regs = set_irq_regs(regs);
        unsigned int irq = hwirq;
        int ret = 0;

        irq_enter(); // Enter an interrupt context including RCU update

#ifdef CONFIG_IRQ_DOMAIN
                // Find a linux irq from a hw irq number
        if (lookup)
                irq = irq_find_mapping(domain, hwirq); // taken
#endif


        ..
        if (unlikely(!irq || irq >= nr_irqs)) {..
        } else {
                generic_handle_irq(irq); {
                    struct irq_desc *desc = irq_to_desc(irq);
                    ..generic_handle_irq_desc(desc); /*
                      = desc->handle_irq(desc); */
        }..
}
```

The most important init function is **plic_init**(). It starts by reading the plic address, ndevs and extended interrupts.

```
static int __init plic_init(struct device_node *node,
            struct device_node *parent)
{
      struct plic_priv *priv;
      struct plic_handler *handler;

      // Maps the memory mapped IO for a given device_node
      priv->regs = of_iomap(np=node, index=0); {
            of_address_to_resource(np, index, &res);

            return ioremap(res.start, resource_size(&res));
            //res.start=40100000 in tinyemu res.flags=IORESOURCE_MEM
            //res.start=50000000 in rlsoc
      }
      of_property_read_u32(node, "riscv,ndev", &nr_irqs); // =0x1f
nr_contexts = of_irq_count(node); // 2: interrupts-extended = <0x01 0x09
0x01 0x0b>; // 0x9 supervisor external interrupt, 0xb machine external
interrupt.
```

Then it calls **irq_domain_add_linear**() to alloc a domain and set its ops to plic_irqdomain_ops. Note that **plic_irq_domain_alloc**() uses **plic_chip**, which is used by **plic_handle_irq**() – the function that handles plic interrupts.

```
      /* allocs a domain and sets domain->ops = ops; */
      priv->irqdomain = irq_domain_add_linear(node, nr_irqs + 1,
                        &plic_irqdomain_ops, priv);

/*static const struct irq_domain_ops plic_irqdomain_ops = {
      .translate  = irq_domain_translate_onecell,
      .alloc      = plic_irq_domain_alloc, {
          for (i = 0; i < nr_irqs; i++) {
              ret = plic_irqdomain_map(domain, virq + i, hwirq + i); {
                irq_domain_set_info(d, irq, hwirq,
                  &plic_chip /* important */,
                  d->host_data, handle_fasteoi_irq, NULL, NULL);
               ..
              }..
          }..
      }..
};*/
```

Then plic_init()→irq_of_parse_and_map()→irq_create_of_mapping()→ irq_create_fwspec_mapping() → irq_create_mapping() →irq_create_mapping_affinity()→ irq_domain_alloc_descs→ __irq_alloc_descs()→alloc_descs()→irq_insert_desc()→ radix_tree_insert(&irq_desc_tree, irq, desc).

For console, (hwirq=0 and virq=1) it will be called irq_insert_desc() too:
irq_insert_desc() ← alloc_descs() ← __irq_alloc_descs() ← __irq_domain_alloc_irqs() ← irq_domain_alloc_irqs()← irq_create_fwspec_mapping() ← irq_create_of_mapping() ← of_irq_get() ← of_irq_to_resource() ← of_irq_to_resource_table() ← of_device_alloc() ←

of_platform_device_create_pdata() { already OF_POPULATED=1 for plic }
←of_platform_bus_create()←of_platform_populate()←of_platform_default_populate()
← of_platform_default_populate_init() /* arch_initcall_sync(
of_platform_default_populate_init); */, of_platform_sync_state_init() /*
late_initcall_sync(of_platform_sync_state_init); */ ← do_one_initcall() ← do_initcall_level()
← do_initcalls() ← do_basic_setup() ← kernel_init_freeable() ← kernel_init() ← rest_init()
← arch_call_rest_init() ← start_kernel().

So, these write in the **irq_desc_tree** radix tree information that will be extracted by **request_irq**() when the user wants to add his or her own interrupt handler.

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long
flags, const char *name, void *dev)=request_threaded_irq(unsigned int irq,
irq_handler_t handler, irq_handler_t thread_fn=NULL, unsigned long
irqflags, const char *devname, void *dev_id)
{
      desc = irq_to_desc(irq);=radix_tree_lookup(&irq_desc_tree, irq);
      action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
      action->handler = handler;
      action->thread_fn = thread_fn; // null
      action->dev_id = dev_id;
      retval = irq_chip_pm_get(&desc->irq_data);
      retval = __setup_irq(irq, desc, action);
}
```

Finally, **plic_init**() ends with the code in the following listing. The most important is that it sets the **plic_handle_irq**() for IRQ_S_EXT – supervisor external interrupt. Also, important is that it sets the **hart_base** of the handler variable.

```
      for (i = 0; i < nr_contexts; i++) {
            of_irq_parse_one(node, i, &parent);
            // Skip contexts other than external interrupts for our
privilege level.
            if (parent.args[0] != RV_IRQ_EXT=IRQ_S_EXT=9)   continue;
            hartid = riscv_of_parent_hartid(parent.np); // 0
            cpu = riscv_hartid_to_cpuid(hartid); // 0
            /* Find parent domain and register chained handler */
            if (!plic_parent_irq && irq_find_host(parent.np)) {
                  plic_parent_irq = irq_of_parse_and_map(node, i);
                  if (plic_parent_irq) // = 9 = IRQ_S_EXT
                        irq_set_chained_handler(irq=plic_parent_irq,
handle=plic_handle_irq/* important */);
                  }
            // continue from for
            // static DEFINE_PER_CPU(struct plic_handler, plic_handlers);
            handler = per_cpu_ptr(&plic_handlers, cpu);
            cpumask_set_cpu(cpu, &priv->lmask);
            handler->present = true;
            handler->hart_base = priv->regs + CONTEXT_BASE + i *
CONTEXT_PER_HART; // res.start + 0x200000 + i * 0x1000
            handler->enable_base = priv->regs + ENABLE_BASE + i *
ENABLE_PER_HART; // res.start + 0x2000 + i * 0x80
            handler->priv = priv;
            for (hwirq = 1; hwirq <= nr_irqs=31; hwirq++)
                  plic_toggle(handler, hwirq, enable=0);
                  // writel(readl(handler->enable_base) & ~(1 << (hwirq %
```

```
                32))); // in rlsoc are all enabled
                nr_handlers++;
        } // for

        // We can have multiple PLIC instances so setup cpuhp state only when
context handler for current/boot CPU is present.
        handler = this_cpu_ptr(&plic_handlers);
        if (handler->present && !plic_cpuhp_setup_done) {
                cpuhp_setup_state(CPUHP_AP_IRQ_SIFIVE_PLIC_STARTING,
                                  "irqchip/sifive/plic:starting",
                                  plic_starting_cpu, plic_dying_cpu);
                plic_cpuhp_setup_done = true;
        }
} // plic_init
plic_starting_cpu() {
 enable_percpu_irq(plic_parent_irq, irq_get_trigger_type(plic_parent_irq));
→irq_percpu_enable()→riscv_intc_irq_unmask()→csr_set(CSR_IE, BIT(d-
>hwirq)); //d->hwirq = 00000200 = 1<<9 = supervisor external interrupt
}
```

Now it's time to dig into **plic_handle_irq**(). It first get a reference for the **plic_chip,** which has defined **irq_eoi**(). Then sets the hwirq in **plic_served_irq** from verilog and calls the effective interrupt handler. Finally, unsets the hwirq from **plic_served_irq** in the chain_irq_exit() call.

```
static struct irq_chip plic_chip = {
        .name           = "SiFive PLIC",
        .irq_mask       = plic_irq_mask,
        .irq_unmask     = plic_irq_unmask,
        .irq_eoi        = plic_irq_eoi,
#ifdef CONFIG_SMP
        .irq_set_affinity = plic_set_affinity,
#endif
};

/* Handling an interrupt is a two-step process: first you claim the
interrupt by reading the claim register, then you complete the interrupt by
writing that source ID back to the same claim register. This automatically
enables and disables the interrupt, so there's nothing else to do*/
static void plic_handle_irq(struct irq_desc *desc)
{
        struct plic_handler *handler = this_cpu_ptr(&plic_handlers);
        struct irq_chip *chip = irq_desc_get_chip(desc); // plic_chip
        void __iomem *claim = handler->hart_base + CONTEXT_CLAIM/*4*/;

        /*chained_irq_enter() and chained_irq_exit()
        are entry/exit functions for chained handlers
        where the primary IRQ chip may implement either fasteoi
        or level-trigger flow control.*/
        chained_irq_enter(chip, desc); {
                /* FastEOI controllers require no action on entry. */
                if (chip->irq_eoi)
                        return;..
        }

        /* readl sets hwirq in plic_served_irq from verilog */
        while ((hwirq = readl(claim))) {
```

```
            int irq = irq_find_mapping(handler->priv->irqdomain, hwirq);
            /* Invoke the handler for a particular irq */
            generic_handle_irq(irq);
            // in linux 5.0, here was the call: writel(hwirq, claim);
            /* however, vm_interrupt() calls
               writel(status, vm_dev->base + VIRTIO_MMIO_INTERRUPT_ACK);
               which sets w_cons_irq_oe <= 1 and w_virt_irq <= 0 in verilog
               and MIP_MEIP and MIP_SEIP will be cleared - see RLSoC PLIC.
            */
      }

      chained_irq_exit(chip, desc); {
        if (chip->irq_eoi)
          chip->irq_eoi(&desc->irq_data);=plic_irq_eoi(struct irq_data *d){
            struct plic_handler *handler = this_cpu_ptr(&plic_handlers);
            // unsets hwirq from plic_served_irq in verilog by writel
            writel(d->hwirq, handler->hart_base + CONTEXT_CLAIM);
          }..
      }
}
```

7.8 Timer interrupts

7.8.1 RLSoC timer interrupts

The core local interrupter (CLINT) address is different in tinyemu dts than rlsoc dts.

```
// rlsoc
cpus {
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        // 104 * 10^6 Hz = 104 MHz = 0x632ea00
        timebase-frequency = <0x632ea00>;..
}
 soc {
        compatible = "simple-bus";..
        clint@60000000 {
            compatible = "riscv,clint0";
            interrupts-extended = <0x1 0x3 0x1 0x7>;
            reg = <0x0 0x60000000 0x0 0x8000000>;
        };..
}

// tinyemu
cpus {
        #address-cells = <0x01>;
        #size-cells = <0x00>;
        timebase-frequency = <0x989680>;..
}
soc {
        compatible = "ucbbar,riscvemu-bar-soc\0simple-bus";..
        clint@2000000 {
            compatible = "riscv,clint0";
            interrupts-extended = <0x01 0x03 0x01 0x07>;
            reg = <0x00 0x2000000 0x00 0xc0000>;
        };..
}
```

In the bbl **clint_done**() function it is set the **timecmp** field of the HLS() to CLINT_BASE_ADDR + 0x4000: HLS()->timecmp = CLINT_BASE_ADDR + 0x4000.

Also, in the bbl we have the **mcall_set_timer**() function which does the following: it sets the new time value at which the rlsoc will raise a new timer interrupt. The value is on 64 bits.

```
static uintptr_t mcall_set_timer(uint64_t when)
{
```

```
  *HLS()->timecmp = when; // also unsets mip->mtip
  clear_csr(mip, MIP_STIP); // clear stip
  set_csr(mie, MIP_MTIP); // enable mtip
  return 0;
}
```

When bbl writes a new value of 64 bits is at the address CLINT_BASE_TADDR+0x4000, this will be translated in two writes of 32 bits to rlsoc. First write is at offset 0x4000 and the second at offset 0x4004. In rlsoc, the 32 bits values will be concatenated to w_wmtimecmp. Also, the **w_clint_we** will be set in **mmu.v**; this variable is sent to the processor and this one will set the **mtimecmp** register with the new value. The processor will raise timer interrupt when **mtimecmp** < **mtime** (the time counter).

```
// mmu.v
assign w_wmtimecmp  =
(r_dev == `CLINT_BASE_TADDR && w_offset==28'h4000 && w_data_we != 0) ?
          {w_mtimecmp[63:32], w_data_wdata} :
(r_dev == `CLINT_BASE_TADDR && w_offset==28'h4004 && w_data_we != 0) ?
          {w_data_wdata, w_mtimecmp[31:0]} : 0;

assign w_clint_we   = (r_dev == `CLINT_BASE_TADDR && w_data_we != 0);

// rvcorem.v
always@(posedge CLK) begin
        if(state == `S_IF)
            if(mtime > `ENABLE_TIMER)
                if(w_plic_we) mip <= w_wmip;
                else if(mtimecmp < mtime) mip <= mip | `MIP_MTIP;
        ..
        if(state == `S_SD && !w_busy) begin
            if(w_clint_we) begin
                mtimecmp    <= w_wmtimecmp;
                mip         <= mip & ~`MIP_MTIP;
            end
        end..
```

7.8.2 Linux timer interrupts

Linux does not access CLINT directly, but bbl does. Linux accesses CLINT through ECALL instructions which delegate control to bbl in M mode.

```
In linux menuconfig:
  Device drivers → Clock source drivers → CONFIG_CLINT_TIMER[not defined]

arch/riscv/Kconfig:    select CLINT_TIMER if !MMU
arch/riscv/Kconfig.socs:    select CLINT_TIMER if RISCV_M_MODE
arch/riscv/Kconfig.socs:    select CLINT_TIMER if RISCV_M_MODE
drivers/clocksource/Kconfig:config CLINT_TIMER
drivers/clocksource/Makefile:obj-$(CONFIG_CLINT_TIMER)            += timer-clint.o
→ drivers/clocksource/timer-clint.c not compiled (also not defined in linux 5.0)

TIMER_OF_DECLARE(clint_timer, "riscv,clint0", clint_timer_init_dt); // is not compiled
static int __init clint_timer_init_dt(struct device_node *np) {..} // is not called
```

Note that bbl writes **mideleg** to SEIP | STIP | SSIP = 0x222 in **delegate_traps**(). In BBL, machine/**mentry.S**, starting from line 61, when the MTIP interrupt appears, it will be cleared in MIE and STIP will be raised. So, the timer interrupt will be handled by linux.

```
 // bbl mentry.
 csrr a1, mcause
 bgez a1, .Lhandle_trap_in_machine_mode // is and exception because mcause >= 0

 # This is an interrupt.  Discard the mcause MSB and decode the rest.
 sll a1, a1, 1

 # Is it a machine timer interrupt?
 /* IRQ_M_TIMER 7 */
 li a0, IRQ_M_TIMER * 2
 bne a0, a1, 1f

 # Yes.  Simply clear MTIE and raise STIP.
 li a0, MIP_MTIP
 csrc mie, a0
 li a0, MIP_STIP
 csrs mip, a0
```

In timer-riscv.c, we have the declaration shown in the following listing. So, in arch/riscv/kernel/vmlinux.lds, we will have in the section **__timer_of_table**, the variable **__of_table_riscv_timer** of type struct of_device_id which will have the fields { .compatible = "riscv", .data = **riscv_timer_init_dt** } .

```
// drivers/clocksource/timer-riscv.c
//dts: cpu@0 { compatible = "riscv"; .. }
TIMER_OF_DECLARE(riscv_timer, "riscv", riscv_timer_init_dt);
#define TIMER_OF_DECLARE(name, _compat, _fn) \
      OF_DECLARE_1_RET(timer, name, compat, fn)
#define OF_DECLARE_1_RET(table, name, compat, fn) \
            _OF_DECLARE(table, name, compat, fn, of_init_fn_1_ret)
#define _OF_DECLARE(table, name, compat, fn, fn_type)              \
            static const struct of_device_id __of_table_##name          \
                  __used __section("__" #table "_of_table")       \
                  __aligned(__alignof__(struct of_device_id))           \
                   = { .compatible = compat,                      \
                       .data = (fn == (fn_type)NULL) ? fn : fn  }
```

In linux, **start_kernel()→time_init()→timer_probe()**. The last one will call **riscv_timer_init_dt()**.

```
void __init time_init(void) {
      struct device_node *cpu;
      u32 prop;

      cpu = of_find_node_by_path("/cpus");
      of_property_read_u32(cpu, "timebase-frequency", &prop);
      riscv_timebase = prop; // tinyemu timebase-frequency = <0x989680>; =
cycles per second = 10MHz; rlsoc has 104MHz
      lpj_fine = riscv_timebase / HZ /*100*/;

      of_clk_init(NULL);
      timer_probe();
}

void __init timer_probe(void)
{
      // __timer_of_table <- TIMER_OF_DECLARE
      for_each_matching_node_and_match(np, __timer_of_table, &match) {
            if (!of_device_is_available(np))
                  continue;
            init_func_ret = match->data; // riscv_timer_init_dt, see above
            ret = init_func_ret(np);
            ..
      }
}
```

riscv_timer_init_dt() starts by getting the virtual irq of RV_IRQ_TIMER and registering the **riscv_clocksource**.

```
int __init riscv_timer_init_dt(struct device_node *n)
{
      child = of_get_compatible_child(n, "riscv,cpu-intc");
      domain = irq_find_host(child);
      riscv_clock_event_irq = irq_create_mapping(domain, RV_IRQ_TIMER); //= a virq

/*
static struct clocksource riscv_clocksource = {
      .flags            = CLOCK_SOURCE_IS_CONTINUOUS,
```

```
        .read           = riscv_clocksource_rdtime,=get_cycles64();
                                =((u64)get_cycles_hi() << 32) | get_cycles();
                                = csr_read(CSR_TIMEH), csr_read(CSR_TIME);
                                // verilog, rvcorem.v,
                                always@(posedge CLK) begin
                                    `ifdef REAL_MTIME if(RST_X) mtime <= mtime + 1; `endif..
                                always@(*) begin
                                    case(w_csr_addr)
                                    `CSR_TIME      : r_rcsr_t = mtime[31:0];
                                    `CSR_TIMEH     : r_rcsr_t = mtime[63:32];
};
*/
        // install new clocksources
        error = clocksource_register_hz(cs=&riscv_clocksource, int hz=riscv_timebase);
=   clocksource_register_scale(cs, scale=1/*see comments*/, freq=hz);
```

Then, **riscv_timer_init_dt**() finishes by registering the timer interrupt handler and setting the callback **riscv_timer_starting_cpu**() to be called on the present cpus which have reached the CPUHP_AP_RISCV_TIMER_STARTING state.

```
        sched_clock_register(riscv_sched_clock=get_cycles64(), 64, riscv_timebase); {
                // empty because CONFIG_GENERIC_SCHED_CLOCK=n
        }
        error = request_percpu_irq(riscv_clock_event_irq,
                                   handler=riscv_timer_interrupt,
                                   "riscv-timer", dev_id=&riscv_clock_event);

        error = cpuhp_setup_state(CPUHP_AP_RISCV_TIMER_STARTING,
                          "clockevents/riscv/timer:starting",
                          riscv_timer_starting_cpu, riscv_timer_dying_cpu);
}
/*
Let's see what happens for hwirq=5=RV_IRQ_TIMER=IRQ_S_TIMER.
start_kernel→time_init→timer_probe→riscv_timer_init_dt→irq_create_mapping_affinity
→irq_domain_alloc_descs()→ __irq_alloc_descs() → alloc_descs() → irq_insert_desc(); →
radix_tree_insert(&irq_desc_tree, irq, desc);

request_percpu_irq(unsigned int irq, irq_handler_t handler,
                   const char *devname, void __percpu *percpu_dev_id) =
__request_percpu_irq(irq, handler, 0, devname, percpu_dev_id) {
                desc = irq_to_desc(irq);=radix_tree_lookup(&irq_desc_tree, irq);
                action->handler = handler;
                action->flags = flags | IRQF_PERCPU | IRQF_NO_SUSPEND;
                action->name = devname;
                action->percpu_dev_id = dev_id;
```

```
                retval = irq_chip_pm_get(&desc->irq_data);
                retval = __setup_irq(irq, desc, action);
}
*/
```

Regarding the **riscv_clock_event** used by **request_percpu_irq**(), it is used to offer access to the function that sets up the next moment in time when the timer interrupt will arrive.

```
        static DEFINE_PER_CPU(struct clock_event_device, riscv_clock_event) = {..
                .features               = CLOCK_EVT_FEAT_ONESHOT, ..
                .set_next_event = riscv_clock_next_event, ..
        }
        int riscv_clock_next_event(unsigned long delta, struct clock_event_device *ce) {
                csr_set(CSR_IE, IE_TIE);
                sbi_set_timer(get_cycles64() + delta); = __sbi_set_timer(stime_value); {
                        // sbi spec 0.1
                        // Program the timer for next timer event
                        #if __riscv_xlen == 32 // true because CONFIG_ARCH_RV64I=n
                        sbi_ecall(SBI_EXT_0_1_SET_TIMER, 0, stime_value,
                          stime_value >> 32, 0, 0, 0, 0); /* calls the bbl's mcall_set_timer()
                                                             which was described previously */
                }
                return 0;
        }
```

Regarding the timer interrupt handler, it calls **tick_periodic**(), which does the regular timing update job, and also calls the **riscv_clock_next_event**() function which programs rlsoc with the new time value at which will come the next timer interrupt.

```
        static irqreturn_t riscv_timer_interrupt(int irq, void *dev_id) {
                struct clock_event_device *evdev = this_cpu_ptr(&riscv_clock_event);
                csr_clear(CSR_IE=CSR_SIE=0x104, IE_TIE=1<<IRQ_S_TIMER);
                evdev->event_handler(evdev); = tick_handle_periodic(*dev) {
                        tick_periodic(cpu); {
                           do_timer(1);..
                           update_wall_time();..
                        }
                        ..if (!clockevents_program_event(dev, next, false)) {
                                ..rc = dev->set_next_event((unsigned long) clc, dev);
                        }
                } //tick_handle_periodic
        }
```

The **riscv_timer_starting_cpu**() function does two things: sets the event handler of the clock event device to tick_handle_periodic() and enables the timer interrupt line in CSR_SIE.

```
static int riscv_timer_starting_cpu(unsigned int cpu)
{
        struct clock_event_device *ce = per_cpu_ptr(&riscv_clock_event, cpu);
        ce->cpumask = cpumask_of(cpu);
        ce->irq = riscv_clock_event_irq;
        clockevents_config_and_register(dev=ce, riscv_timebase, 100, 0x7fffffff); →
clockevents_register_device(dev);→ tick_check_new_device(dev); → tick_setup_device(td,
newdev, cpu, cpumask_of(cpu)); →tick_setup_periodic(struct clock_event_device *dev, int
broadcast)→tick_set_periodic_handler(struct clock_event_device *dev, int broadcast) {
                dev->event_handler = tick_handle_periodic;
        }
        enable_percpu_irq(riscv_clock_event_irq,
           irq_get_trigger_type(riscv_clock_event_irq))→ irq_percpu_enable()→
riscv_intc_irq_unmask ()→csr_set(CSR_IE, BIT(d->hwirq)); // d->hwirq=00000020
        return 0;
   }
```

7.9 TinyEMU interrupts

As we have seen in the Basics of TinyEMU subchapter, one of the most important function is **riscv_machine_init**(). This one installs using **cpu_register_device**() the various devices of the system. The CLINT and PLIC are registered using their base address, size and callback functions which are called (clint_read, clint_write and plic_read, plic_write) when the system accesses these devices.

```
const VirtMachineClass riscv_machine_class = {
        "riscv32,riscv64,riscv128",
        riscv_machine_set_defaults,
        riscv_machine_init (const VirtMachineParams *p), {..
                cpu_register_device(s->mem_map, CLINT_BASE_ADDR, CLINT_SIZE,
                opaque=s, clint_read, clint_write, DEVIO_SIZE32) {
                        PhysMemoryRange *pr;
                        pr = &s->phys_mem_range[s->n_phys_mem_range++];
                        pr->map = s->mem_map; pr->addr = addr; pr->org_size = size;
                        pr->is_ram = FALSE; pr->opaque = opaque;
                        pr->read_func = read_func; pr->write_func = write_func; ..
                }
                cpu_register_device(s->mem_map, PLIC_BASE_ADDR, PLIC_SIZE, s,
                                        plic_read, plic_write, DEVIO_SIZE32);
                for(i = 1; i < 32; i++)
                        irq_init(&s->plic_irq[i], plic_set_irq, s, i);
                ..
                // console
                s->common.console = p->console;
                memset(vbus, 0, sizeof(*vbus)); // VIRTIOBusDef *vbus;
                vbus->mem_map = s->mem_map;
                vbus->addr = VIRTIO_BASE_ADDR; //0x40010000
                irq_num = VIRTIO_IRQ; // 1
                /* virtio console */
                if (p->console) {
                                vbus->irq = &s->plic_irq[irq_num];
                                s->common.console_dev = virtio_console_init(vbus, p-
                >console);..
        } // riscv_machine_init()
}
```

clint_read() and **clint_write**() are simple functions used to read the time register or read or write the **timecmp** field of the RISCVMachine structure. **timecmp** specifies the moment in time when will arrive the new timer interrupt.

plic_read() and **plic_write**() are functions used to set or clear **plic_served_irq** and read the current pending interrupt.

The function **plic_set_irq**() is used to set or unset the (MIP_MEIP | MIP_SEIP) from the **mip** register. This uses **riscv_cpu_set_mip**() and **riscv_cpu_reset_mip**() which are straight forward.

```
static void plic_set_irq(void *opaque, int irq_num, int state)
{
        RISCVMachine *s = opaque;
        uint32_t mask;
        mask = 1 << (irq_num - 1);
        if (state) s->plic_pending_irq |= mask;
        else      s->plic_pending_irq &= ~mask;
        plic_update_mip(s); {
                RISCVCPUState *cpu = s->cpu_state;
                uint32_t mask;
                mask = s->plic_pending_irq & ~s->plic_served_irq;
                if (mask)  riscv_cpu_set_mip(cpu, MIP_MEIP | MIP_SEIP);
                else        riscv_cpu_reset_mip(cpu, MIP_MEIP | MIP_SEIP);
        }
}
static void glue(riscv_cpu_set_mip, MAX_XLEN)(RISCVCPUState *s, uint32_t mask)
{
   s->mip |= mask;
  /* exit from power down if an interrupt is pending */
   if (s->power_down_flag && (s->mip & s->mie) != 0)
      s->power_down_flag = FALSE;
}
glue(riscv_cpu_reset_mip, MAX_XLEN)(RISCVCPUState *s, uint32_t mask) {
        s->mip &= ~mask;
}
```

In the **riscv_cpu_interp_x32**(), the tinyemu instructions execution loop, first it is verified if any interrupts are pending. If so, **raise_interrupts**() is called.

```
static void no_inline glue(riscv_cpu_interp_x, XLEN)(RISCVCPUState *s,
                                    int n_cycles1)
{
   uint32_t opcode, insn, rd, rs1, rs2, funct3;..
   insn_counter_addend = s->insn_counter + n_cycles1;
   s->n_cycles = n_cycles1;

   /* check pending interrupts */
   if (unlikely((s->mip & s->mie) != 0)) {
      if (raise_interrupt(s)) {
         s->n_cycles--;
         goto done_interp;
      }
```

```
   }

   s->pending_exception = -1;
   /* we use a single execution loop to keep a simple control flow */
    for(;;) {    .. // see TinyEMU execution flow for normal execution of the instructions.
```

The **raise_interrupt**() function is called when the **mip** register contains interrupts that are enabled in the **mie** register and verifies if the global interrupts are enabled in **mstatus** (by calling **get_pending_irq_mask**()); if so, it calls **raise_exception**(). **raise_exception**() checks whether the bit in **mideleg** or **medeleg** corresponding to the **cause** variable is set; if so, it will delegate the trap to the S mode, and, otherwise, to the M mode.

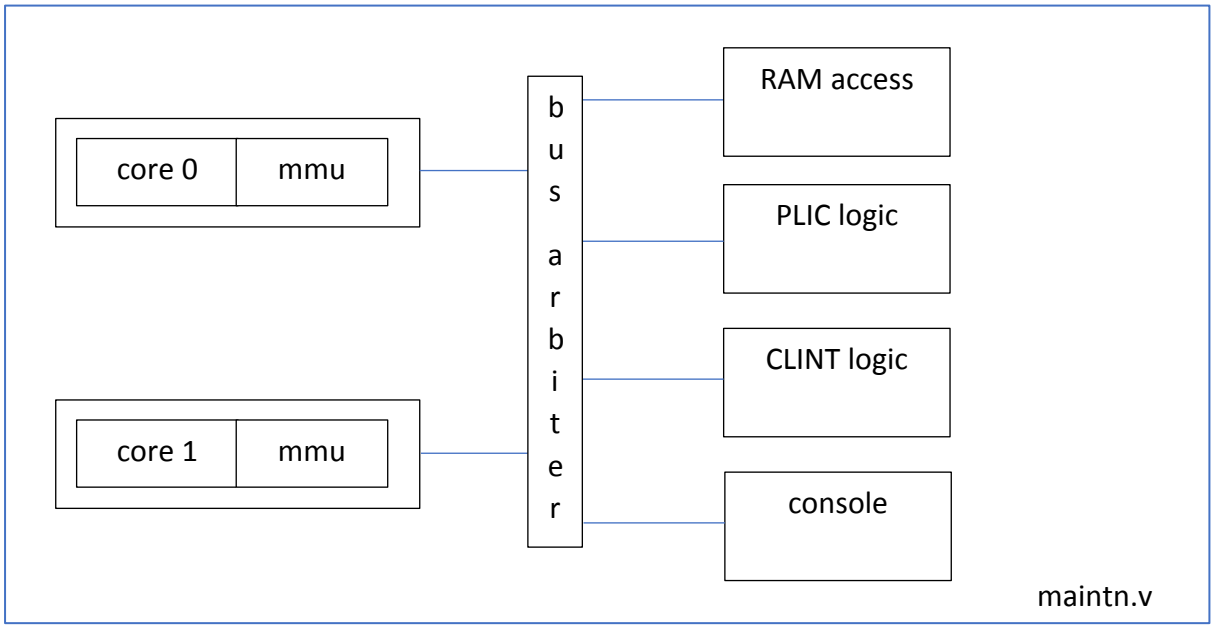
```
raise_interrupt(RISCVCPUState *s)
{
        mask = get_pending_irq_mask(s);
        if (mask == 0)          return 0;
        irq_num = ctz32(mask); // first bit which is 1
        raise_exception(s, cause = irq_num | CAUSE_INTERRUPT); =
raise_exception2(s,cause,tval=0)
/* also called from riscv_cpu_interp_x() when pending_exception >=0,
for example when s->pending_exception = CAUSE_USER_ECALL + s->priv */ {..
                if (s->priv <= PRV_S) {
                        /* delegate the exception to the supervisor priviledge */
                        if (cause & CAUSE_INTERRUPT)
                                deleg = (s->mideleg >> (cause & (MAX_XLEN - 1))) & 1;
                        else
                                deleg = (s->medeleg >> cause) & 1;
                } else deleg = 0;
                ..
                if (deleg) {
                        s->scause = causel;
                        s->sepc = s->pc;..
                        set_priv(s, PRV_S); // privilege level
                        s->pc = s->stvec;
                } else {
                        s->mcause = causel;
                        s->mepc = s->pc;..
                        set_priv(s, PRV_M);
                        s->pc = mtvec;
                }
        }
        return -1;
}
```

8. True dual-core RLSoC

8.1 Dual-core architecture

In order to obtain a dual-core system, changes were made to the m_mmu and m_RVCoreM modules. All virtio logic was deleted from the mmu: m_RVuc (the microcontroller), m_console, m_disk modules were removed. New console logic was addedas a replacement. The init logic, PLIC, CLINT and the new console logic were moved from mmu.v to the top module which is now **maintn.v** for simulation and implementation. The necessary logic from top.v was merged with main.v in maintn.v. See src/run.tcl to a list of verilog files used in rlsoctn.

The new system's architecture is shown in the figure below.



The dts must be changed to have two cores. The cores are identical with the exception that each core has its own reg and phandle ids.

```
cpus {
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        // 27 * 10^6 = 0x19BFCC0
        timebase-frequency = <0x19BFCC0>;

        cpu@0 {
                device_type = "cpu";
                reg = <0x0>;
                status = "okay";
                compatible = "riscv";
```

```
            riscv,isa = "rv32acim";
            mmu-type = "riscv,sv32";
            clock-frequency = <0x19BFCC0>;

            interrupt-controller {
                #interrupt-cells = <0x1>;
                interrupt-controller;
                compatible = "riscv,cpu-intc";
                phandle = <0x1>;
                linux,phandle = <0x01>;
            };
        };

        cpu@1 {
            device_type = "cpu";
            reg = <0x1>;
            status = "okay";
            compatible = "riscv";
            riscv,isa = "rv32acim";
            mmu-type = "riscv,sv32";
            clock-frequency = <0x19BFCC0>;

            interrupt-controller {
                #interrupt-cells = <0x01>;
                interrupt-controller;
                compatible = "riscv,cpu-intc";
                phandle = <0x02>;
                linux,phandle = <0x02>;
            };
        };
    };
```

8.2 Simulating and running on tang nano 20k

8.2.1 Building a minimal linux system

The dual-core system can be simulated in verilator. The sources are packed in the rlsoc-tn-20240526.tgz archive. Please decompress this archive.

BBL can be compiled with the following commands; because it depends on linux, we must build linux first.

```
cd rlsoc-tn/riscv-pk-build
../riscv-pk/configure --enable-logo --enable-print-device-tree --host=riscv32-buildroot-linux-gnu --with-arch=rv32imac --with-payload=../linux-kernel/vmlinux
make
```

Before the true dual-core rlsoc version, it was needed to patch linux arch/riscv/kernel/sbi.c and add the ipi_clear() function as sbi_clear_ipi(). However, now the ipi_clear is done automatically by hardware.

```
static const struct riscv_ipi_ops sbi_ipi_ops = {
	.ipi_inject = sbi_send_cpumask_ipi,
	.ipi_clear  = sbi_clear_ipi
};
```

Then, you must compile linux and busybox and copy vmlinux to **rlsoc-tn/linux-kernel** folder. You can use the linux kernel config and busybox config provided in the archive in this folder. Compile linux after busybox because it needs the initramfs rootfs.cpio.gz archive.

You can study the configs by issuing the menuconfig target and compile linux by issuing the vmlinux target.

```
make ARCH=riscv CROSS_COMPILE=riscv32-linux- menuconfig
make ARCH=riscv CROSS_COMPILE=riscv32-linux- -j5 vmlinux
```

To build a minimal rootfs based on busybox, follow the following steps (find init, rcS and rcK in appendix):

```
git clone https://github.com/mirror/busybox.git
cd busybox && mkdir busybox-static
git checkout 2d4a3d9e6
cp ../linux-kernel/config-busybox-1.37 .config
make ARCH=riscv CROSS_COMPILE=riscv32-linux- -j5
make ARCH=riscv CROSS_COMPILE=riscv32-linux- CONFIG_PREFIX=$(pwd)/busybox-static -j5 install
cd busybox-static
mkdir -p etc/init.d proc sys dev
```

```
cp ../../init .
cp ../../rcS etc/init.d
cp ../../rcK etc/init.d
sudo chown -R root .
sudo chmod 755 -R init etc/init.d/rc*
find . | cpio -o -H newc | gzip > ../../linux-kernel/rootfs.cpio.gz
```

8.2.2 Simulation

After building the linux system we can run the simulation in verilator. In **define.vh** we can select whether to use one or two cores by defining the USE_SINGLE_CORE variable; however, in order to simulate, SIM_MODE must be defined.

run-sim.sh is an easy to understand shell script that builds (by running **make veri**) the simulator executable and launches it (**./simv**).

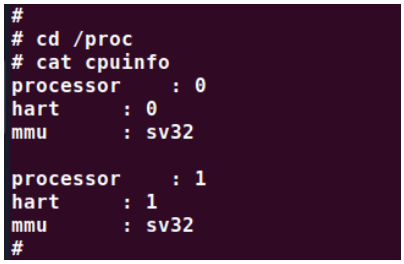
```
cd rlsoc-tn/rvsoc_src_ver053/src
./run-sim.sh
```

This will start rlsoc simulation which boots bbl and the linux system.

Please note that in rlsoc-tn, the **mtime** signal is moved to maintn.v and is incremented at each clock posedge in SIM_MODE too.

When mtime reaches ENABLE_TIMER, the user can enter commands to the linux console via fcmd.txt and fid.txt. Remember that in fid.txt you must write a different id after fcmd.txt is changed.

The simulation takes from 7 minutes on single core to 17 minutes on dual-core at 27MHz in dts.



8.2.3 Running RLSoC on tang nano 20k

Tang nano 20k has a microsd card slot which we will be using to load the linux system into RAM. We work in the src folder and we have to generate the linux binary image. This can be made by running **build.sh** (which in turn uses the tools from the **initmem_gen2** folder). Now insert the microsd card in tang nano and burn initmem.bin on microsd by using the **dd** tool; see example below.

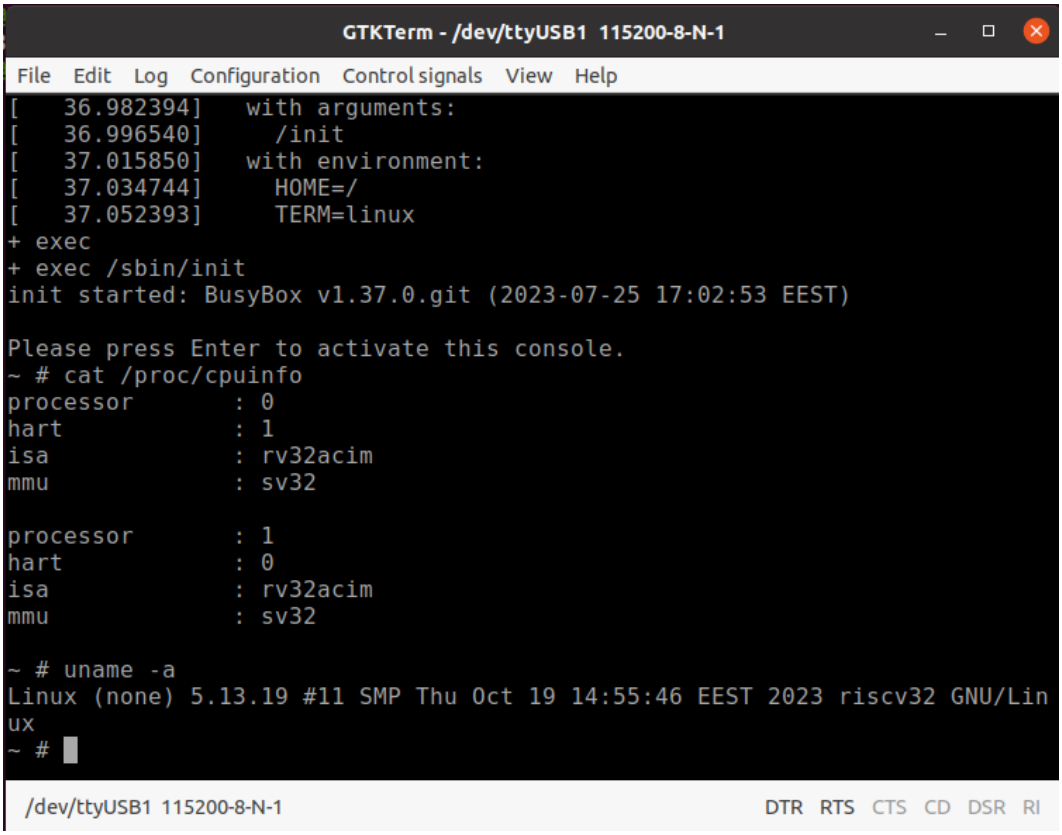
Note 8.2.3-1: please replace /dev/sdb with your microsd drive.

```
./build.sh
sudo dd if=../binary/initmem.bin of=/dev/sdb bs=512 seek=0 && sync
```

In order to build the fpga bitfile, please first select whether to use one or two cores by defining the USE_SINGLE_CORE variable in **define.vh**. SIM_MODE must be undefined. Then you can copy the files from my rlsoc2 github repository to the src folder in order to have the latest verilog implementation. Then use the following command from the src folder (you have to install GoWin IDE first and include gw_sh in your path):

```
make gwsynth
```

Before testing the system, optionally, we may connect a max7219, 7-segment display to tang nano 20k; please see the sdcard section from the **rlsoc.cst** file (which is in the src folder) and the tang nano 20k pinlabel.



Before programming the fpga, launch gtkterm and set the serial port properties as in the figure. To burn the bitfile into the fpga, please power-up the board and run the openFPGAloader command.

```
gtkterm &
openFPGALoader --board tangnano20k impl/pnr/project.fs
```

Five leds (out of six) will be turned on by the hardware implemented first boot loader in order to signalize that linux was correctly transferred from the sdcard to RAM and then the linux boot process will start; please see the **w_led** signal from maintn.v.

In the picture, hart 1 wins the linux hart lottery of the boot process.

Linux boot time is about 1.5 minutes for single core and 3 minutes for double-core.

8.3 Miscelaneous changes

8.3.1 MicroSD

The initmem.bin image has 8MB and would take a lot of time to be transferred via the serial line at a speed of 115200 bps. If we increase the baudrate, errors will appear on the serial line, also the tang nano 20k board will corrupt its serial port at high data rate. So, copying first initmem.bin in a microsd and use the tang nano's microsd to transfer the initmem.bin to the RAM memory is a viable solution.

In maintn.v, in the **sd_loader** module instantiation, we have passed **r_init_state** to the loader **w_main_init_state** signal. When this signal equals 3, we have the process of transferring the microsd image to system RAM. We have also passed **r_sd_state** of maintn.v to microsd **w_ctrl_state** signal. The r_sd_state is driven by maintn logic and is 0 when we want to read a 4 byte word from the microsd and write it to memory. The memory write is ordered by the **r_sd_init_we** signal set by maintn.

The sd_loader module reads data from the microsd in chunks of 512 bytes. For this process, it uses the sd_reader module which was brought by the FPGA-SDcard-Reader project on github.

8.3.2 BBL, linux and device tree

Because BBL requires linux kernel to be aligned, in the binary, at 4MB chunks, and because tang nano 20k fpga has only 8MB RAM, we decided to place linux at address 0. So, in riscv-pk/bbl/bbl.lds we have the payload placed immediately after the memory start address:

```
SECTIONS
{

  . = 0x80000000;
  .payload :
  {
    *(.payload)
  }
```

In riscv-pk/machine/fdt.c, in function mem_done() we have a simplified condition to compute mem_size from the dts. In dts we fouled linux to think that the memory size is 7MB.

```
static void mem_done(const struct fdt_scan_node *node, void *extra)
{
  uintptr_t self = (uintptr_t)mem_done;..
  // laur quick and dirty fix because linux,usable-memory does not work
  if (base <= self/* && self <= base + size*/) { mem_size = size; }
```

The effective bbl code is at address 0x80700000 (7MB after start) which is MEM_START in the riscv-pk/configure file. And in riscv-pk/configure.ac we must have the following intuitive line:

```
AC_SUBST([MEM_START], [0x80700000], [Physical memory start address])
```

In verilog define.vh, the program counter starting point, D_START_PC is at this address 32'h80700000.

When building the binary image, initmem_gen2 will put the dtb at address D_INITD_ADDR=6MB (also D_INITD_ADDR in define.vh) – from where bbl will access it.

BBL will build the dtb based on the dts and will pass the dtb address to the linux kernel. The riscv-pk/bbl/bbl.c dtb_output() function was modified in order to consider the new linux end address instead of a function of MEGAPAGE_SIZE. filter_dtb() puts the dtb at the address returned by dtb_output().

```
static uintptr_t dtb_output()
```

```
{
  uintptr_t end = kernel_end ? (uintptr_t)kernel_end : (uintptr_t)&_payload_end;
  //uintptr_t ee = (end + MEGAPAGE_SIZE - 1) / MEGAPAGE_SIZE * MEGAPAGE_SIZE;
  uintptr_t ee = end;
  return ee;
```

8.3.4 Memory driver

The tang nano 20k uses a GoWin FPGA which has 8MB SDRAM. The SDRAM memory driver is adapted from the sdram-tang-nano-20k project on github. This driver includes a PLL which offers a clock phase shifted with PI radians, which is used by the SDRAM.

The rlsoc uses an intermediate module, namely **memorytn**, to talk to the SDRAM driver. The sdram driver was modified to use 32 bits words; if data is unaligned, memorytn may use two memory read or write commands.

In memorytn, specific refresh logic was added because the raw sdram driver does not include it; if maintn wants to read or write while the refresh is made, these operations are delayed. The refresh is started if the counter has reached the limit; this process is completely transparent to the rest of the system.

```
   task prepare_refresh;
   begin
            state <= 50;
            r_refresh <= 1;
            //r_stall <= 1;
   end
   endtask

always@(posedge clk) begin..
   case(state)
     8'd0: // idle
       ..
      `ifdef TN_DRAM_REFRESH
      else if((r_refreshcnt > `REFRESH_CNT) &&
                !w_busy) begin
         // ram refresh
         prepare_refresh;
         if(r_refresh > `REFRESH_CNT_MAX)
            r_late_refresh <= 1;
      end
```

The memory writes were delayed from 4 to 5 clock cycles in order to avoid data corruption.

In rvcorem.v, when sfence.vma instruction is executed, a TLB flush is ordered.

The memory controller does not use cache.

8.3.5 Known bugs

Many bugs were corrected between rlsoc1 and rlsoc-tn. The most important is that in rlsoc1, the time interrupts generated were only MTIP and no STIP; in our system only linux programs (via the bbl) the CLINT to generate interrupts and these are STIP. Because the timer interrupts appear separately to each processor, the logic for timer interrupt programming which is made via CLINT is moved to the mmu module of each core.

At this time, no bugs are known to be present in rlsoc-tn.

8.4 Bus arbiter

It is important to know that both cores may be active at a given moment of time; but only one has access to memory or I/O at a given moment of time. If both cores want to access these resources simultaneously, then only one has access and the other waits. To satisfy these requirements, a new variable was defined – w_data_busy - which informs the bus arbiter that I/O is or not available in the current moment. It is important to know that the bus arbiter will set w_data_busy to some core only when this core wants to access I/O; if the core does not want to access I/O will have w_data_busy 0.

In maintn.v, r_data_busy is unset only when I/O finishes its job, and this includes the uart tx module.

Any cpu core (which is implemented in m_RVCoreM) can be paused by the arbiter via the mmu by setting its w_busy signal:

```
    assign w_state = (!RST_X | r_halt)? 0       : // `S_INI
                     (w_com) ? `S_COM :
                     ((w_busy && !w_ex1) || w_ex1_busy)? state  :
                     (state==`S_FIN)? `S_INI : ..?
                     : state+1;
```

The core who has access to the bus is selected by the **grant** signal. When a core has **grant** equal to its hart id, the core is connected to the system, while the other is scanned to see if wants access to the system or is working on the internal part of the instruction.

```
        if(state == 0) begin..
            if(r_pending_req1) begin
                grant <= 1;
                state <= 1;
                exec1_aux;
                check_request0;
            end else if(r_pending_req0) begin
                grant <= 0;
                state <= 1;
                exec0_aux;
                check_request1;
            end else if(bus_dram_le1 | bus_dram_we_t1 | bus_data_le1 |
bus_data_we1) begin
                grant <= 1;
                prepare_exec1;
                state <= 1;
                check_request0;..
```

In the bus arbiter state1 the core who has grant had made a system request (memory or I/O) and the bus arbiter waits for the system busy flag – which acknowledges that the system has received the request; after this the bus arbiter goes into state 2 to wait for the system to resolve the request and signalize that is not busy anymore. In every state,

the bus arbiter scans the other hart to see if has made a request to access the system – and if so it will send busy to the that hart.

8.5 The HVC_RISCV_SBI console

The console is mapped into the memory space. A read at HVC_BASE_TADDR address will return the number of characters in the queue and a read at HVC_BASE_TADDR + 4 address will return the first character in the queue:

```
always@(*) begin
    case (r_dev)
        `CLINT_BASE_TADDR : r_data_data <= r_clint_odata;
        `PLIC_BASE_TADDR  : r_data_data <= r_plic_odata;
        `HVC_BASE_TADDR   : if(r_mem_paddr == `HVC_BASE_ADDR) begin
          r_data_data <= {24'h0, 2'h0, r_consf_cnts /*r_consf_en*/};
        end else if(r_mem_paddr == (`HVC_BASE_ADDR + 4)) begin
          r_data_data <= {24'h0, /*cons_fifo[r_consf_head]*/ r_char_value};
        end else ..
```

The queue is now 32 bytes length. In this case, the variable r_data_busy avoids multiple dequeue operations for a single char read:

```
        always@(posedge pll_clk) begin
        if((r_mem_paddr == (`HVC_BASE_ADDR + 4)) && r_consf_cnts &&
r_data_le && (r_data_busy==2)) begin
                r_consf_en <= (r_consf_cnts<=1) ? 0 : 1;
                r_consf_head <= r_consf_head + 1;
                r_consf_cnts <= r_consf_cnts - 1;
                r_char_value <= cons_fifo[r_consf_head];
        end
```

Recall that in SIM_MODE we use the **read_file.v** module to read console commands from the linux user.

To output a character on the console, the software must write to the memory address space at the TOHOST_ADDR address with the command CMD_PRINT_CHAR. Tha variable **r_wait_ready** means that we have waited for the tx ready signal before sending the print command. It avoids writing twice the same character.

```
always@(posedge pll_clk) begin
        if(w_tx_ready)
            r_wait_ready <= 1;
        else
            r_wait_ready <= 0;
        if((r_mem_paddr==`TOHOST_ADDR && r_data_we) &&
(w_data_wdata[31:16]==`CMD_PRINT_CHAR) && w_tx_ready && r_wait_ready) begin
            r_uart_we   <= 1;
            r_uart_data <= w_data_wdata[7:0];
        end else begin
            r_uart_we   <= 0;
            r_uart_data <= 0;
        end
```

In the bbl, the functions that read or write a character are simrv_getc() and simrv_putc(). These are called by mcall_console_getchar() and mcall_console_putchar(), which are called by the bbl when linux uses the ECALL instruction to read or write to the console.

```
void mcall_trap(uintptr_t* regs, uintptr_t mcause, uintptr_t mepc)
{
 write_csr(mepc, mepc + 4);

 uintptr_t n = regs[17], arg0 = regs[10], arg1 = regs[11], retval, ipi_type;

 switch (n)
 {
  case SBI_CONSOLE_PUTCHAR:
    retval = mcall_console_putchar(arg0);
    break;
  case SBI_CONSOLE_GETCHAR:
    retval = mcall_console_getchar();
    break;..
```

For example, the linux call to putchar is:

```
arch/riscv/kernel/sbi.c:75:void sbi_console_putchar(int ch) =
        sbi_ecall(SBI_EXT_0_1_CONSOLE_PUTCHAR, 0, ch, 0, 0, 0, 0, 0);
```

Please remember that CONFIG_HVC_RISCV_SBI must be enabled in the kernel config for the dual-core system; the virtio is not used anymore for the console, so, the PLIC logic can be ignored because the PLIC does not serve any interrupt from now on.

The linux initialization code calls device_initcall(). hvc_sbi_init() is such a call and sets up the hvcd linux thread. This thread periodically calls to __hvc_poll(hp, true); which calls hp->ops->get_chars(hp->vtermno, buf, count).

```
static const struct hv_ops hvc_sbi_ops = {
        .get_chars = hvc_sbi_tty_get, // calls sbi_console_getchar
        .put_chars = hvc_sbi_tty_put, // calls sbi_console_putchar
        .notifier_add remains null
};
drivers/tty/hvc/hvc_riscv_sbi.c: device_initcall(hvc_sbi_init);
hvc_sbi_init(void) { return PTR_ERR_OR_ZERO(hvc_alloc(0, 0, &hvc_sbi_ops, 16)); }

struct hvc_struct *hvc_alloc(uint32_t vtermno, int data,
                             const struct hv_ops *ops,
                             int outbuf_size) {
        if (atomic_inc_not_zero(&hvc_needs_init)) { // true on 5.13
                hvc_init() {
                   tty_set_operations(drv, &hvc_ops);
                   hvc_task = kthread_run(khvcd, NULL, "khvcd");
                }
```

```
        }..
        // finally, the console is registered
        hp->index = i;
        if (i < MAX_NR_HVC_CONSOLES) {
                cons_ops[i] = ops;
                vtermnos[i] = vtermno;
        }..
        hvc_check_console(i)->if (index == hvc_console.index)
                register_console(&hvc_console);
}
```

8.6 Inter processor interrupts

8.6.1 BBL and dual-core boot

The reset_vector is defined as entry in bbl/bbl.lds. In machine/mentry.S, the reset_vector label has a jump to do_reset, which sets mtvec to the trap_vector label.

```
do_reset: ..
  # write mtvec and make sure it sticks
  la t0, trap_vector
  csrw mtvec, t0
  csrr t1, mtvec
1:bne t0, t1, 1b
..
```

The do_reset code also calls init_first_hart() for hart 0 and sets MIP_MSIP in mie for hart 1; then enables MIE in mstatus.

```
..
  /* first hart has mhartid = 0 */
  csrr a3, mhartid
..
  # Boot on the first hart
  beqz a3, init_first_hart

  # set MSIE bit to receive IPI. MIP_MSIP is Machine Software Interrupt = 1 << 3.
  li a2, MIP_MSIP
  csrw mie, a2

  # laur
  # enable interrupts in mstatus
  csrr t1, mstatus
  ori t1, t1, MSTATUS_MIE
  csrw mstatus, t1
..
```

Note that init_first_hart() calls wake_harts() which sends ipi to each hart.

Then do_reset continues the sequence of the hart 1 which waits for an interrupt from hart 0, and when it arrives it will call init_other_hart.

```
.LmultiHart:
#if MAX_HARTS > 1
  # wait for an IPI to signal that it's safe to boot
  wfi
```

```
..
  # laur disable only start if mip is set
  #csrr a2, mip
  #andi a2, a2, MIP_MSIP
  #beqz a2, .LmultiHart
..
  bltu a3, a2, init_other_hart
```

The trap_table label contains __trap_from_machine_mode as the 13rd entry; at this label, the C function trap_from_machine_mode will be called.

```
trap_table:
#define BAD_TRAP_VECTOR 0
  .dc.a bad_trap
  .dc.a pmp_trap ..
#define TRAP_FROM_MACHINE_MODE_VECTOR 13
# this is the 13rd .dc.a
  .dc.a __trap_from_machine_mode

__trap_from_machine_mode:
  jal trap_from_machine_mode
  j restore_regs
```

Because mtvec is loaded with trap_vector, when the ipi comes, in hart 1 will be executed the code at trap_vector label. When the first ipi arrives, mscratch of hart 1 is 0.

```
trap_vector:
  /* swap mscratch and sp */
  csrrw sp, mscratch, sp
  beqz sp, .Ltrap_from_machine_mode
```

At .Ltrap_from_machine_mode, in register a1 is stored the 13rd entry from trap_table; .Lhandle_trap_in_machine_mode will use it.

```
.Ltrap_from_machine_mode:
  csrr sp, mscratch
  addi sp, sp, -INTEGER_CONTEXT_SIZE
  save a0, a1 ..
  /* TRAP_FROM_MACHINE_MODE_VECTOR=13 decimal */
  li a1, TRAP_FROM_MACHINE_MODE_VECTOR
  j .Lhandle_trap_in_machine_mode
```

So, .Lhandle_trap_in_machine_mode will jump to trap_from_machine_mode().

```
.Lhandle_trap_in_machine_mode:
  # Preserve the registers.  Compute the address of the trap handler.
  STORE ra, 1*REGBYTES(sp)
  STORE gp, 3*REGBYTES(sp) ..
1:auipc t0, %pcrel_hi(trap_table)  # t0 <- %hi(trap_table)
  save t1
  sll t1, a1, LOG_REGBYTES          # t1 <- a1 * ptr size
  save t2
  add t1, t0, t1                    # t1 <- %hi(trap_table)[a1]
  save s0
  LOAD t1, %pcrel_lo(1b)(t1)        # t1 <- trap_table[a1] ..
  # Invoke the handler.
  jalr t1
.. restore regs
  mret
```

In trap_from_machine_mode() the mcause is checked and in the case of the IPI, this value is 0x80000003: most significant bit denotes the interrupt and 3 denotes MSIP (ipi is sent as a software interrupt). We also must clear MSIP flag before return.

```
void trap_from_machine_mode(uintptr_t* regs, uintptr_t dummy, uintptr_t mepc)
{
  uintptr_t mcause = read_csr(mcause);

  switch (mcause)
  {
    case CAUSE_LOAD_PAGE_FAULT:
    case CAUSE_STORE_PAGE_FAULT:
    case CAUSE_FETCH_ACCESS:
    case CAUSE_LOAD_ACCESS:
    case CAUSE_STORE_ACCESS:
      return machine_page_fault(regs, mcause, mepc);
    case 0x80000003:
      // mscratch is zero and brought us here from mentry.S. msip=(1 << 3)
      write_csr(mip, read_csr(mip) & ~MIP_MSIP);
      return;
```

After return, .Lhandle_trap_in_machine_mode executes mret and the program counter goes to the instructions following wfi which will call init_other_hart() for hart 1.

After running init_first_hart() and init_other_hart(), the cores enter the linux entry point and will compete at the boot lottery from head.S. After it finishes the base boot sequence, the winner hart will call **smp_init**() to unblock the waiting hart which will execute smp_callin().

8.6.2 RLSoC IPIs

In RLSoC dual core, the order of memory accesses is the same as the program order for load and store operations; so, we can ignore fence instructions.

IPI logic is implemented in maintn.v and rvcorem.v.

In order to send an ipi, the software must write to an offset of the clint base address:

- if data to be written is 0 then software requests clear ipi; if it is 1, it signals ipi from machine mode; if it is 2, it signals ipi from supervisor mode.

- if the offset is 0, then the ipi signals core 0; if the offset is 4, then the ipi signals core 1.

```
if(r_dev == `CLINT_BASE_TADDR && (w_offset==28'h0 || w_offset==28'h4) &&
r_data_we != 0 && (r_data_busy==2)) begin
  if(w_offset==28'h0) begin
    if(w_data_wdata == 32'h0) begin
      r_ipi <= {r_ipi[31:17], 1'b0, r_ipi[15:1], 1'b0};
    end else begin
      // signal core 0
      if(w_data_wdata == 2)
        r_ipi <= {r_ipi[31:17], 1'b1, r_ipi[15:1], 1'b1}; // S-priv
      else
        r_ipi <= {r_ipi[31:17], 1'b0, r_ipi[15:1], 1'b1}; // M-priv
    end
  end else /*if(w_offset == 28'h4)*/ begin ..
```

The maintn.v transmits the r_ipi signal to rvcorem.v:

- this signal has 0 on the hart's bit position if it wants to clear ipi;

- it has 1 on the hart's bit position and 1 on 16 plus the hart's bit position if it wants to send an ipi from supervisor mode;

- it has 1 on the hart's bit position and 0 on 16 plus the hart's bit position if it wants to send an ipi from machine mode;

Recall that the ipi logic of the core transmits ipi by software interrupt: SSIP for supervisor software interrupt pending and MSIP for machine software interrupt pending.

In state S_FIN, if the interrupt to be taken is MSIP or SSIP then the ipi control logic clears mip[3:0].

```
always@(posedge CLK) begin /***** write CSR registers *****/
  if(state == `S_IF) begin
    if(w_ipi & (1<<mhartid)) begin
      if(r_ipi_taken == 0) begin
        if((w_ipi >> 16) & (1<<mhartid)) begin
          mip[3:0] <= mip[3:0] | `MIP_SSIP;
        end else begin
          mip[3:0] <= mip[3:0] | `MIP_MSIP;
        end
```

```
        r_ipi_taken <= 1;
      end
    end else begin
        r_ipi_taken <= 0;
    end
  end else if((state == `S_FIN && !w_busy) && (w_interrupt_mask &&
w_take_int) && (irq_num == `MIP_SSIP_SHIFT || irq_num == `MIP_MSIP_SHIFT))
begin
    mip[3:0] <= 0;
end
```

The **r_ipi_taken** signal avoids sending an ipi multiple times and requires CLINT (maintn.v) to clear the ipi interrupt request.

```
if(w_ipi_taken0) begin
   r_ipi <= {r_ipi[31:17], 1'b0, r_ipi[15:1], 1'b0};
end else if(w_ipi_taken1) begin ..
```

Regarding the **WFI** instruction, while wfi is executed, “if an enabled interrupt is present or later becomes present while the hart is stalled, the interrupt exception will be taken on the following instruction, i.e., execution resumes in the trap handler and mepc = pc + 4.” [13]

“The WFI instruction can also be executed when interrupts are disabled. The operation of WFI must be unaffected by the global interrupt bits in mstatus (MIE and SIE) and the delegation register mideleg (i.e., the hart must resume if a locally enabled interrupt becomes pending, even if it has been delegated to a less-privileged mode), but should honor the individual interrupt enables (e.g, MTIE) (i.e., implementations should avoid resuming the hart if the interrupt is pending but not individually enabled). WFI is also required to resume execution for locally enabled interrupts pending at any privilege level, regardless of the global interrupt enable at each privilege level.” [13]

Here is how the wfi instruction is executed in rvcorem.v.

```
wire w_executing_wfi = ((r_opcode==`OPCODE_SYSTEM   ) && (r_funct3 ==
`FUNCT3_PRIV   ) && (r_funct12== `FUNCT12_WFI    ));
wire w_exit_wfi = (mip & mie & (`MIP_MEIP | `MIP_MTIP | `MIP_MSIP))   |
(r_priv_t <= `PRIV_S && (mip & mie & (`MIP_SEIP | `MIP_STIP |`MIP_SSIP))) |
(r_priv_t <= `PRIV_U && (mip & mie & (`MIP_UEIP | `MIP_UTIP | `MIP_USIP)));

always @(posedge CLK) begin /////// update program counter
  if(!RST_X || r_halt) begin pc <= `D_START_PC; end ..
  else if(state==`S_FIN && !w_busy) begin
    if(pending_exception != ~0) begin
      pc <= (w_deleg) ? stvec : mtvec; // raise exception
    end else if(w_interrupt_mask && w_take_int) begin
      pc <= (w_deleg) ? stvec : mtvec; // interrupt here
    end else if(w_executing_wfi)  begin
      if(w_exit_wfi) begin
        pc <= pc + 4;
      end
    end else begin
      pc <= (r_tkn) ? r_jmp_pc : (r_cinsn) ? pc + 2 : pc + 4;
    end
```

```
   end
end
```

8.6.3 Linux IPIs

Please patch linux arch/riscv/kernel/sbi.c and add the ipi_clear() function as sbi_clear_ipi().

```
static const struct riscv_ipi_ops sbi_ipi_ops = {
        .ipi_inject = sbi_send_cpumask_ipi
        .ipi_clear  = sbi_clear_ipi
};
```

Linux must be compiled with SMP support if we want to use two cores:

```
CONFIG_SMP=y
CONFIG_GENERIC_SMP_IDLE_THREAD=y
```

Linux sends ipi via bbl by using the ECALL instruction. sbi_send_cpumask_ipi() → sbi_send_ipi()→__sbi_send_ipi().

```
if (!sbi_spec_is_0_1()) {
        if (sbi_probe_extension(SBI_EXT_IPI) > 0) {
                __sbi_send_ipi = __sbi_send_ipi_v02;//→ sbi_ecall
(SBI_EXT_IPI, SBI_EXT_IPI_SEND_IPI, ..
        } else {
                __sbi_send_ipi = __sbi_send_ipi_v01;//→ sbi_ecall
(SBI_EXT_0_1_SEND_IPI,..

        }
```

BBL machine/mtrap.c handles linux ecall requests.

```
void mcall_trap(uintptr_t* regs, uintptr_t mcause, uintptr_t mepc)
{
 write_csr(mepc, mepc + 4);
 uintptr_t n = regs[17], arg0 = regs[10], arg1 = regs[11], retval, ipi_type;

 switch (n)
 {..
  case SBI_SEND_IPI:
    ipi_type = IPI_SOFT;
    goto send_ipi;..
send_ipi:
    send_ipi_many((uintptr_t*)arg0, ipi_type);
    retval = 0;
    break;
  case SBI_CLEAR_IPI:
    retval = mcall_clear_ipi();
    break;
```

Finally, bbl sends or clears the ipi. Please note that hls->ipi was set by the bbl while parsing the dtb searching for the clint base address (machine/fdt.c clint_done() function).

```
static void send_ipi(uintptr_t recipient, int event)
{
  if (((disabled_hart_mask >> recipient) & 1)) return;
  //atomic_or(&OTHER_HLS(recipient)->mipi_pending, event);
  mb();
  // laur send 2 to set ssip
  *OTHER_HLS(recipient)->ipi = 2;
}
static uintptr_t mcall_clear_ipi()
{
  // laur also clear the ipi reg
  long hartid = read_csr(mhartid);
  uintptr_t i=0;
  if(hartid == 0)
      *OTHER_HLS(i)->ipi = 0;
  else
      *OTHER_HLS(++i)->ipi = 0;
  return clear_csr(mip, MIP_SSIP) & MIP_SSIP;
}
```

9. Conclusion

In this book we have presented the main aspects of the rlsoc system and tinyemu simulator. We have covered building, simulating and implementation aspects. We had a look at the linux kernel and seen how it manages boot and system interaction.

Clearly, we have not covered all implementation aspects and many improvements can be made to the book. Please now that we did the best when writing this book. We hope you will find it usefull.

Have a great success!

Appendix 1. Linux console configuration on TinyEMU

This listing was created by printing messages on linux kernel using **pr_laur**() and enabling DEBUG_VIRTIO in tinyemu to print debug messages in **virtio_mmio_read()** and **virtio_mmio_write()** functions. Recall that tinyemu console is at address 0x40010000 and tinyemu disk is accessible at address 0x40011000.

```
virtio_mmio_probe
        virtio_mmio_read: offset=0x0 magic val=0x74726976 size=4
        virtio_mmio_read: offset=0x4 version val=0x2 size=4
        virtio_mmio_read: offset=0x8 device-id val=0x3 console-device size=4
        virtio_mmio_read: offset=0xc vendor-id val=0xffff any-or-none-vendor size=4
vm_dev->vdev.id.device =  00000003
vm_dev->version =  00000002
        virtio_mmio_write: offset=0x70 status-reset val=0x0 size=4
        virtio_mmio_read: offset=0x70 VIRTIO_MMIO_STATUS       status val=0x0 size=4
        virtio_mmio_write: offset=0x70 status val=0x1=VIRTIO_CONFIG_S_ACKNOWLEDGE         size=4
        virtio_mmio_probe end
/*virtio_mmio_probe 00000000    vm_dev->vdev.id.device =  00000002 (block) vm_dev->version =  00000002,
no debug prints for it in tinyemu, so all the prints here are for console; end(virtio_mmio_probe).*/

   virtio_mmio_read: offset=0x70 val=0x1 size=4
        virtio_mmio_write: offset=0x70 val=0x3 size=4
   virtio_mmio_write: offset=0x14 DEVICE_FEATURES_SEL  val=0x1 size=4
        virtio_mmio_read: offset=0x10 DEVICE_FEATURES val=0x1 size=4
        virtio_mmio_write: offset=0x14 DEVICE_FEATURES_SEL val=0x0 size=4
        virtio_mmio_read: offset=0x10 DEVICE_FEATURES val=0x1 size=4

   virtio_mmio_write: offset=0x24 VIRTIO_MMIO_DRIVER_FEATURES_SEL val=0x1 size=4
```

```
        virtio_mmio_write: offset=0x20 VIRTIO_MMIO_DRIVER_FEATURES val=0x1 size=4
        virtio_mmio_write: offset=0x24 VIRTIO_MMIO_DRIVER_FEATURES_SEL val=0x0 size=4
        virtio_mmio_write: offset=0x20 VIRTIO_MMIO_DRIVER_FEATURES val=0x1 size=4
   virtio_mmio_read: offset=0x70 val=0x3 size=4
        virtio_mmio_write: offset=0x70 val=0xb size=4
        virtio_mmio_read: offset=0x70 val=0xb size=4

virtcons_probe:
use_multiport(portdev)= 00000000, so is false
nvqs= 00000002
/* Queue selector - Write Only */
        virtio_mmio_write: offset=0x30 VIRTIO_MMIO_QUEUE_SEL;s->queue_sel=0       val=0x0 size=4
/* Ready bit for the currently selected queue - Read Write */
        virtio_mmio_read: offset=0x44 VIRTIO_MMIO_QUEUE_READY;val = s->queue[s->queue_sel=0].ready = 0;       val=0x0 size=4
/* Maximum size of the currently selected queue - Read Only */
        virtio_mmio_read: offset=0x34 VIRTIO_MMIO_QUEUE_NUM_MAX;val = MAX_QUEUE_NUM = 16;       val=0x10 size=4
/* Queue size for the currently selected queue - Write Only */
        virtio_mmio_write: offset=0x38 VIRTIO_MMIO_QUEUE_NUM;s->queue[s->queue_sel].num = val = 16;  val=0x10 size=4
/* Selected queue's Descriptor Table address, 64 bits in two halves */
        virtio_mmio_write: offset=0x80 VIRTIO_MMIO_QUEUE_DESC_LOW;s->queue[s->queue_sel].desc_addr=val;    val=0x821ce000 size=4
        virtio_mmio_write: offset=0x84 VIRTIO_MMIO_QUEUE_DESC_HIGH        val=0x0 size=4
/* Selected queue's Available Ring address, 64 bits in two halves */
        virtio_mmio_write: offset=0x90 VIRTIO_MMIO_QUEUE_AVAIL_LOW;s->queue[s->queue_sel].avail_addr=val;   val=0x821ce100 size=4
        virtio_mmio_write: offset=0x94 VIRTIO_MMIO_QUEUE_AVAIL_HIGH val=0x0 size=4
/* Selected queue's Used Ring address, 64 bits in two halves */
        virtio_mmio_write: offset=0xa0 VIRTIO_MMIO_QUEUE_USED_LOW;s->queue[s->queue_sel].used_addr=val;    val=0x821cf000 size=4
        virtio_mmio_write: offset=0xa4 VIRTIO_MMIO_QUEUE_USED_HIGH        val=0x0 size=4
        virtio_mmio_write: offset=0x44 VIRTIO_MMIO_QUEUE_READY;s->queue[s->queue_sel=0].ready = 1; val=0x1 size=4

        virtio_mmio_write: offset=0x30 VIRTIO_MMIO_QUEUE_SEL;s->queue_sel = 1; val=0x1 size=4
```

```
        virtio_mmio_read: offset=0x44 VIRTIO_MMIO_QUEUE_READY;val = 0 = s->queue[s->queue_sel=1].ready; val=0x0 size=4
        virtio_mmio_read: offset=0x34 VIRTIO_MMIO_QUEUE_NUM_MAX;val = MAX_QUEUE_NUM=16; val=0x10 size=4
                                // please note that rlsoc has `CONSOLE_QUEUE_NUM_MAX=2
        virtio_mmio_write: offset=0x38 VIRTIO_MMIO_QUEUE_NUM;val = s->queue[1].num=16; val=0x10 size=4
/* Selected queue's Descriptor Table address, 64 bits in two halves */
   virtio_mmio_write: offset=0x80 VIRTIO_MMIO_QUEUE_DESC_LOW;s->queue[1].desc_addr=val; val=0x821d0000 size=4
        virtio_mmio_write: offset=0x84 VIRTIO_MMIO_QUEUE_DESC_HIGH val=0x0 size=4
/* Selected queue's Available Ring address, 64 bits in two halves */
        virtio_mmio_write: offset=0x90 VIRTIO_MMIO_QUEUE_AVAIL_LOW;s->queue[1].avail_addr=val;          val=0x821d0100 size=4
        virtio_mmio_write: offset=0x94 VIRTIO_MMIO_QUEUE_AVAIL_HIGH        val=0x0 size=4
/* Selected queue's Used Ring address, 64 bits in two halves */
        virtio_mmio_write: offset=0xa0 VIRTIO_MMIO_QUEUE_USED_LOW;s->queue[1].used_addr=val; val=0x821d1000 size=4
        virtio_mmio_write: offset=0xa4 VIRTIO_MMIO_QUEUE_USED_HIGH val=0x0 size=4
        virtio_mmio_write: offset=0x44 VIRTIO_MMIO_QUEUE_READY;s->queue[1].ready = 1; val=0x1 size=4
        virtio_mmio_read: offset=0x70 status val=0xb size=4
        virtio_mmio_write: offset=0x70 status val=0xf size=4

add_port(portdev, 0)
        virtio_mmio_write: offset=0x50 VIRTIO_MMIO_QUEUE_NOTIFY;queue_notify(s, val=0); qs->manual_recv ---              val=0x0 size=4
        // this is printed 16 times

        virtio_mmio_read: offset=0x60 VIRTIO_MMIO_INTERRUPT_STATUS          val=0x2 size=4
        virtio_mmio_write: offset=0x64 VIRTIO_MMIO_INTERRUPT_ACK    val=0x2 size=4

register_console
preferred_console =  00000000
has_preferred_console =  00000001
try_enable_new_console i is preferred_console =  00000000
c->options =  00000000
```

```
try_enable_new_console(newcon, true) returns  00000000
newcon->index =  00000000
newcon->flags =  00000007
[    1.160891] printk: console [hvc0] enabled
[    1.161310] printk: bootconsole [sbi0] disabled
add_port() end.
virtcons_probe() end.

virtio_mmio_read: addr=0x40010000 offset=0x70 val=0xf size=4

VIRTIO_F_VERSION_1 00000009
nvqs= 00000001 // block
          virtio_mmio_read addr=0x40010000, offset=256 >= VIRTIO_MMIO_CONFIG=256 ***
                    virtio_config_read: val=0xa8
          virtio_mmio_read addr=0x40010000, offset=258 >= VIRTIO_MMIO_CONFIG=256 ***
                    virtio_config_read: val=0x27
//    virtio_mmio_read addr=0x40011000, offset=256 >= VIRTIO_MMIO_CONFIG=256 ***
//                  virtio_config_read: val=0x8000
//        virtio_mmio_read addr=0x40011000, offset=260 >= VIRTIO_MMIO_CONFIG=256 ***
//                  virtio_config_read: val=0x0
[    1.123552] vvirtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=16 write_size=0
irtio_blk virtio virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=16 write_size=0
1: [vda] 32768 5          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=16 write_size=0
12-byte logical           virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=16 write_size=0
blocks (16.8 MB/          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=11 write_size=0
```

```
16.0 MiB)
… other linux boot messages
buildroot login:          virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_read: addr=0x40010000 offset=0x60 VIRTIO_MMIO_INTERRUPT_STATUS val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 VIRTIO_MMIO_INTERRUPT_ACK val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 VIRTIO_MMIO_QUEUE_NOTIFY  val=0x0 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=1 write_size=0
r         virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x0 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=1 write_size=0
o         virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x0 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
          queue_notify: idx=1 read_size=1 write_size=0
o         virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x0 size=4
          virtio_mmio_write: addr=0x40010000 offset=0x50 val=0x1 size=4
```

```
        queue_notify: idx=1 read_size=1 write_size=0
t       virtio_mmio_read: addr=0x40010000 offset=0x60 val=0x1 size=4
        virtio_mmio_write: addr=0x40010000 offset=0x64 val=0x1 size=4
```

Appendix 2. The busybox based rootfs init, rcS and rcK scripts

$ cat init

```
#!/bin/sh

# devtmpfs does not get automounted for initramfs
/bin/mount -t devtmpfs devtmpfs /dev
/bin/mount -t proc proc /proc

if (exec 0</dev/console) 2>/dev/null; then
   exec 0</dev/console
   exec 1>/dev/console
   exec 2>/dev/console
fi

#echo "sbin/init"
exec /sbin/init "$@"
```

$ cat /etc/init.d/rcS

```
#!/bin/sh

# Start all init scripts in /etc/init.d
# executing them in numerical order.
#
for i in /etc/init.d/S??* ;do

        # Ignore dangling symlinks (if any).
        [ ! -f "$i" ] && continue

        case "$i" in
   *.sh)
        # Source shell script for speed.
        (
         trap - INT QUIT TSTP
         set start
         . $i
        )
        ;;
   *)
        # No sh extension, so fork subprocess.
        $i start
        ;;
        esac
done
```

$cat /etc/init.d/rcK

```
#!/bin/sh

# Stop all init scripts in /etc/init.d
# executing them in reversed numerical order.
#
for i in $(ls -r /etc/init.d/S??*) ;do

        # Ignore dangling symlinks (if any).
        [ ! -f "$i" ] && continue

        case "$i" in
  *.sh)
        # Source shell script for speed.
        (
         trap - INT QUIT TSTP
         set stop
         . $i
        )
        ;;
  *)
        # No sh extension, so fork subprocess.
        $i stop
        ;;
        esac
done
```

References

[1] Fabrice Bellard. (2021) TinyEMU RISC-V emulator. [Online]. https://bellard.org/tinyemu

[2] Junya Miura, Hiromu Miyazaki, Kenji Kise. (2020) A portable and Linux capable RISC-V computer system in verilog hdl. [Online]. https://arxiv.org/pdf/2002.03576.pdf

[3] (2021) Elixir Bootlin. [Online]. https://elixir.bootlin.com/linux/v5.13.19/source

[4] Costas Calamvok. (2009) The Verilog to html converter. [Online]. https://burbleland.com/v2html/

[5] Digilent. Getting Started with Vivado. [Online]. https://digilent.com/reference/vivado/getting_started/start

[6] Digilent. Nexys A7 Reference Manual. [Online]. https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual

[7] Xilinx. (2017) Zynq-7000 SoC and 7 series devices memory interface solutions v4.2 User Guide. [Online]. https://www.xilinx.com/support/documentation/ip_documentation/mig_7series/v4_2/ug586_7Series_MIS.pdf

[8] Digilent. Arty A7 reference manual. [Online]. https://digilent.com/reference/programmable-logic/arty-a7/reference-manual

[9] Buildroot - Making Embedded Linux Easy. [Online]. https://buildroot.org/downloads

[10] T. Petazzoni. Device tree 101. [Online]. https://bootlin.com/pub/conferences/2021/webinar/petazzoni-device-tree-101/petazzoni-device-tree-101.pdf

[11] Hiromu Miyazaki, Takuto Kanamori, Md Ashraful Islam, Kenji Kise. (2020, Dec.) RVCoreP : An optimized RISC-V soft processor of five-stage pipelining. [Online]. https://arxiv.org/pdf/2002.03568

[12] Andrew Waterman, Krste Asanovic, Ed., *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191214-draft*.: RISC-V Foundation, Dec. 2019.

[13] Krste Asanovic, Andrew Waterman, Ed., *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.12-draft*.: RISC-V Foundation, 2019.

[14] Drew Barbier, Palmer Dabbelt, Abner Chang. (2021) RISC-V Platform-Level Interrupt Controller. [Online]. https://raw.githubusercontent.com/riscv/riscv-plic-spec/master/riscv-plic.pdf

[1 Micron. 1Gb: x4, x8, x16 DDR2 SDRAM. [Online].
5] https://www.micron.com/products/dram/ddr2-sdram/part-catalog/mt47h64m16hr-25e

[1 Micron. 2Gb: x4, x8, x16 DDR3 SDRAM. [Online].
6] https://www.micron.com/products/dram/ddr3-sdram/part-catalog/mt41j128m16jt-125

[1 Using as - symbol names. [Online]. https://sourceware.org/binutils/docs-
7] 2.24/as/Symbol-Names.html#Symbol-Names

Thank you