国立大学法人
電気通信大学
The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

「Course」
RISC-V Computer System Integration

「Lecture 09」 Course Summary

Phạm Công Kha
Hoàng Trọng Thức

Tháng 9/2023

Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

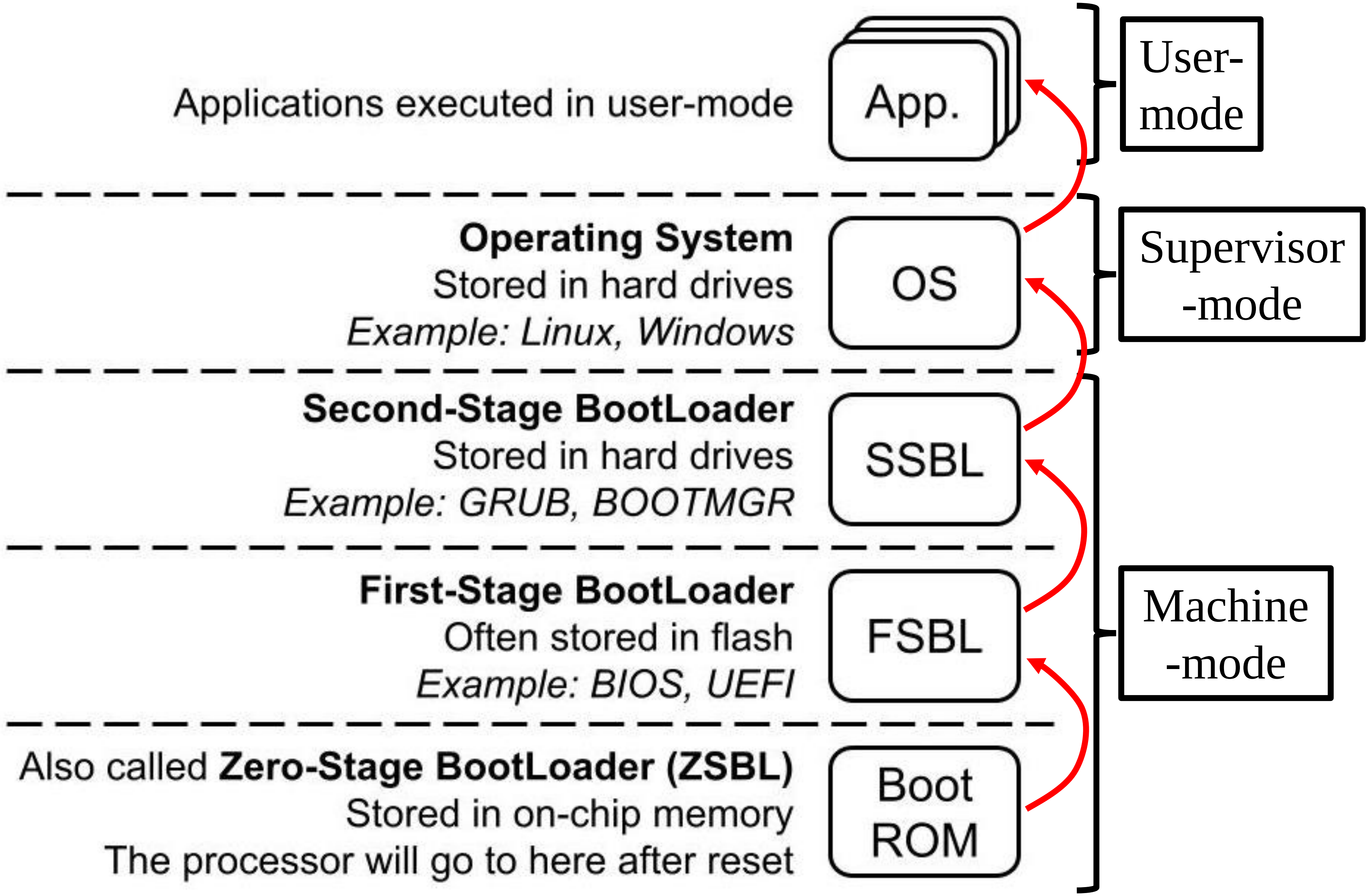
国立大学法人 電気通信大学 The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

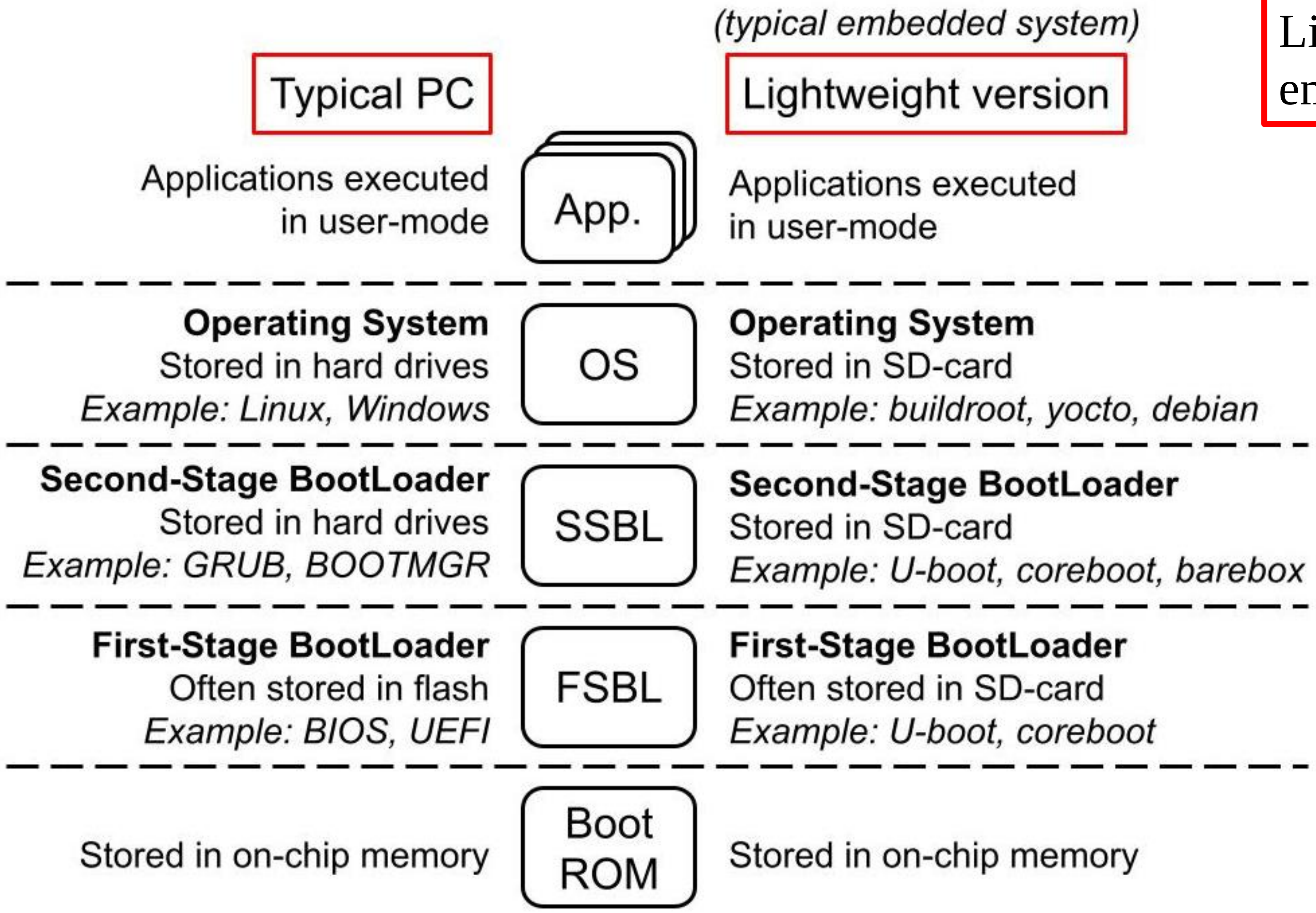
1. Boot sequence (1/11) Generic boot flow



A typical PC boot sequence

1. **Boot ROM** *(ZSBL)*: stores device tree; the reset point of processor(s).
2. **FSBL**: checks hardware based on the device tree; preparing SSBL's environment.
3. **SSBL**: prepares OS's environment, allocates memory, and installing drivers.

1. Boot sequence (2/11) Embedded boot flow



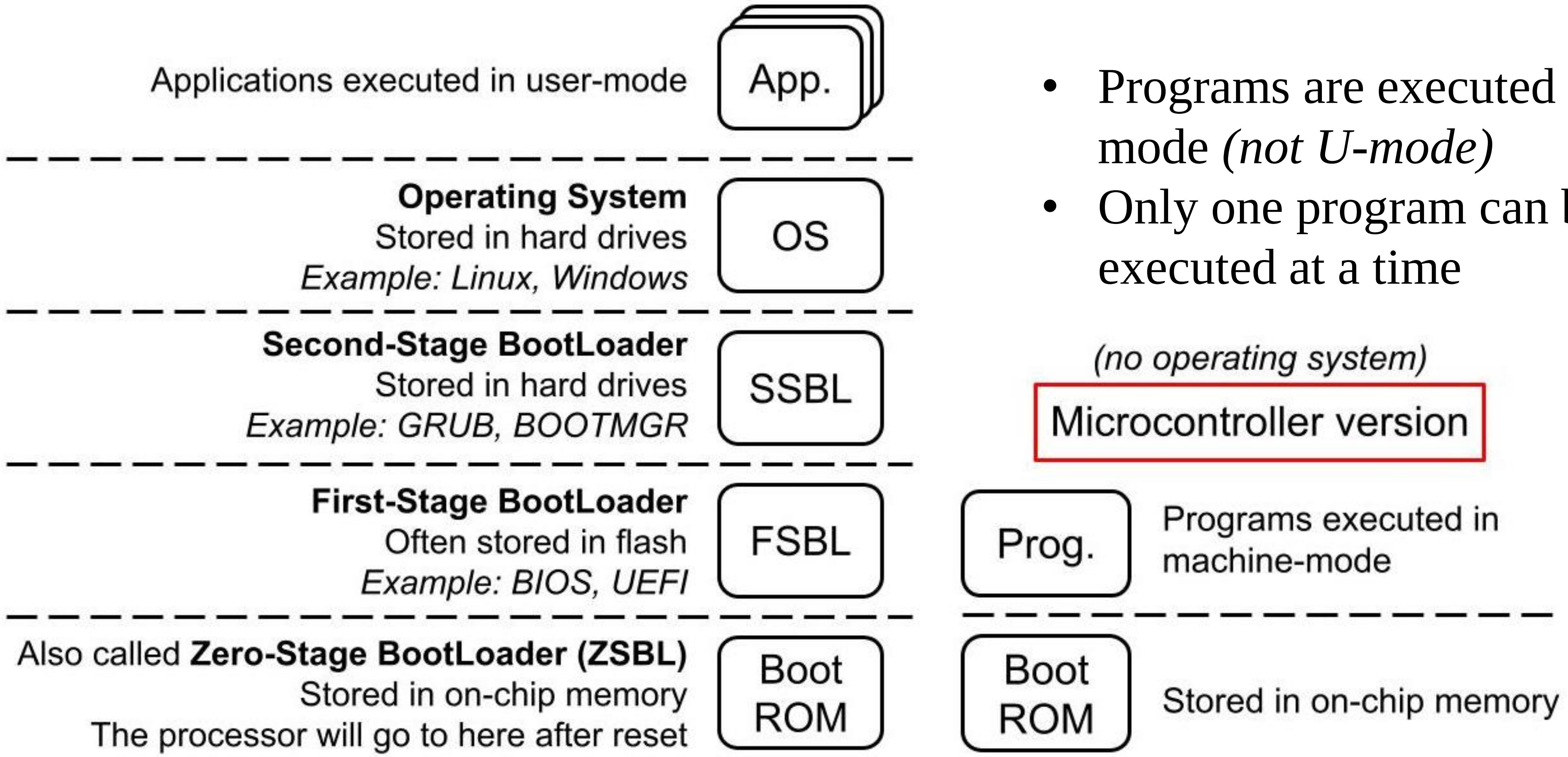
Lightweight version used in embedded systems

In lightweight:

- Compact OSes
- Simpler bootloaders
- Bootloaders and OS are often stored in SD-card

1. **Boot sequence** (3/11) MCU boot flow

Microcontroller (MCU) version with no operating system

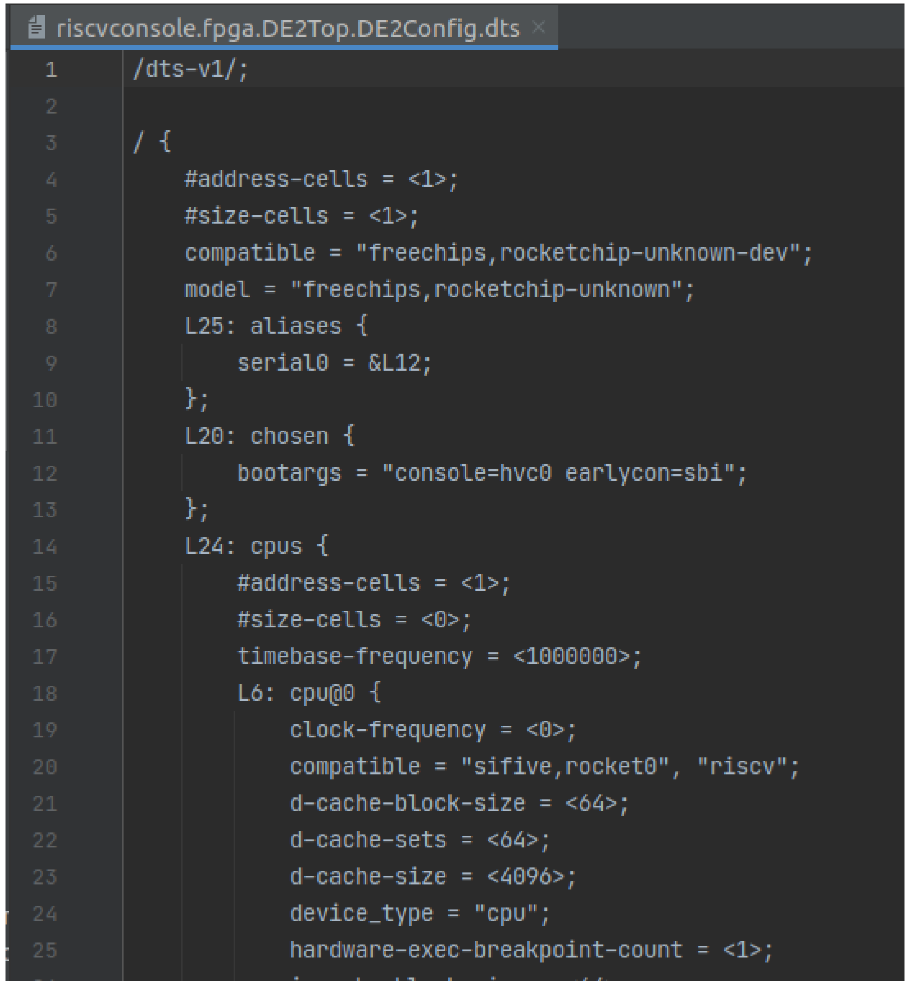


- Programs are executed at M-mode *(not U-mode)*
- Only one program can be executed at a time

1. Boot sequence (4/11) Device tree

Device tree (**.dts**) and its binary (**.dtb**) files are the *declaration* files *from hardware to software*. Device tree contains all the information of hardware devices.

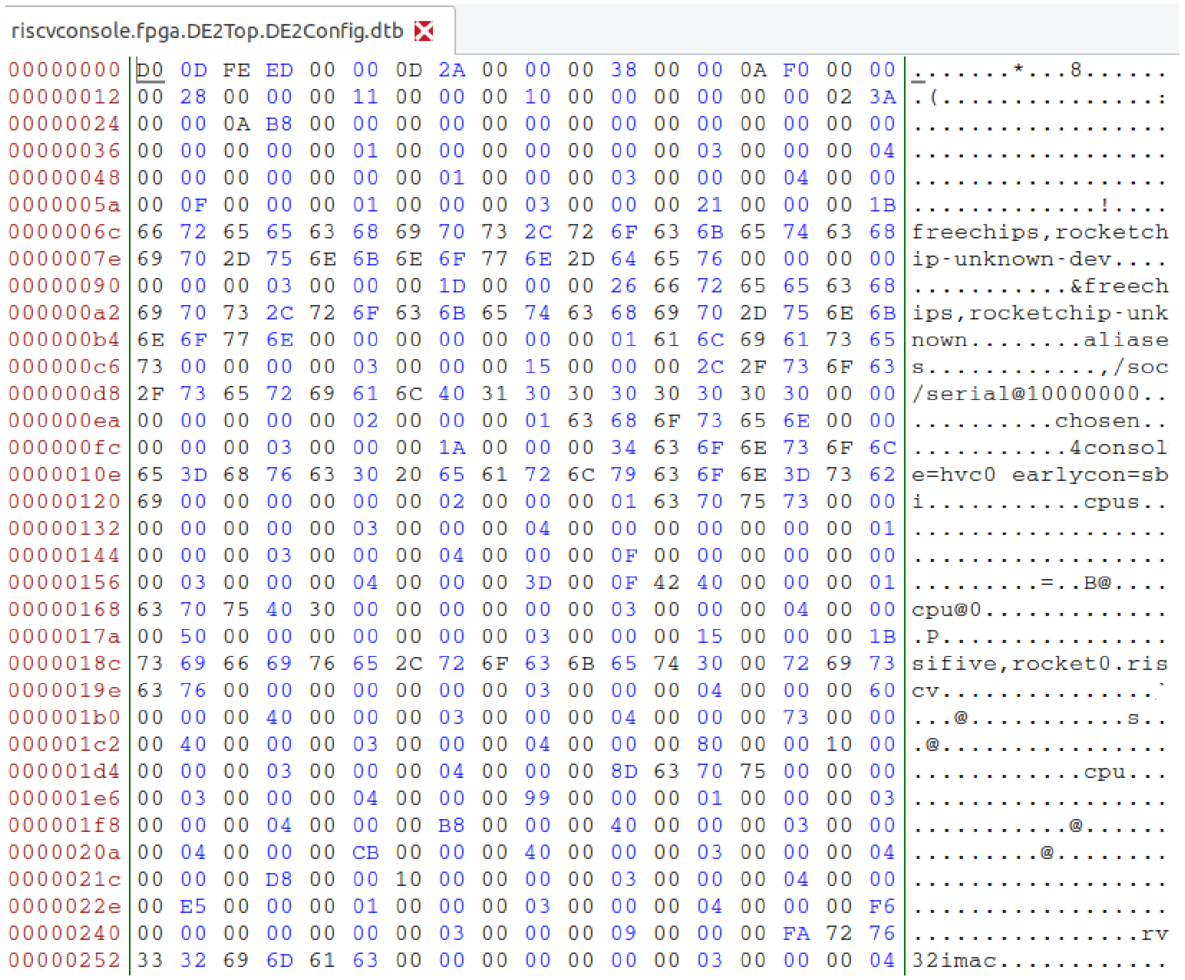
The **.dts** file:



riscvconsole.fpga.DE2Top.DE2Config.dts

```
/dts-v1/;

/ {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "freechips,rocketchip-unknown-dev";
    model = "freechips,rocketchip-unknown";
    L25: aliases {
        serial0 = &L12;
    };
    L20: chosen {
        bootargs = "console=hvc0 earlycon=sbi";
    };
    L24: cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        timebase-frequency = <1000000>;
        L6: cpu@0 {
            clock-frequency = <0>;
            compatible = "sifive,rocket0", "riscv";
            d-cache-block-size = <64>;
            d-cache-sets = <64>;
            d-cache-size = <4096>;
            device_type = "cpu";
            hardware-exec-breakpoint-count = <1>;
```

The **.dtb** file:

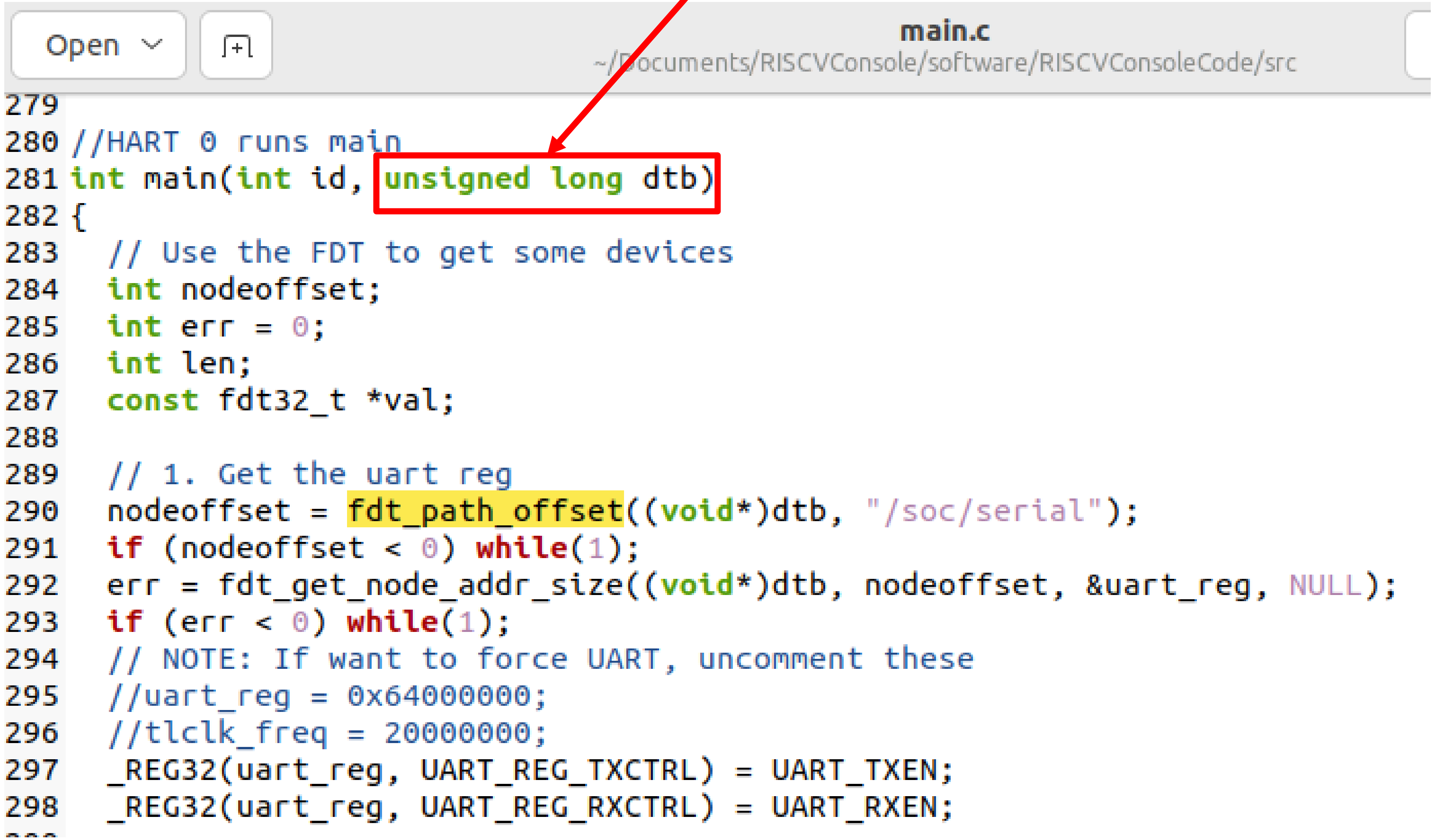


riscvconsole.fpga.DE2Top.DE2Config.dtb

```
00000000 D0 0D FE ED 00 00 0D 2A 00 00 00 38 00 00 0A F0 00 00 .......*...8......
00000012 00 28 00 00 00 11 00 00 00 10 00 00 00 00 00 00 02 3A .(...............:
00000024 00 00 0A B8 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..................
00000036 00 00 00 00 00 01 00 00 00 00 00 00 00 03 00 00 00 04 ..................
00000048 00 00 00 00 00 00 00 01 00 00 00 03 00 00 00 04 00 00 ..................
0000005a 00 0F 00 00 00 01 00 00 00 03 00 00 00 21 00 00 00 1B .............!....
0000006c 66 72 65 65 63 68 69 70 73 2C 72 6F 63 6B 65 74 63 68 freechips,rocketch
0000007e 69 70 2D 75 6E 6B 6E 6F 77 6E 2D 64 65 76 00 00 00 00 ip-unknown-dev....
00000090 00 00 00 03 00 00 00 1D 00 00 00 26 66 72 65 65 63 68 ...........&freech
000000a2 69 70 73 2C 72 6F 63 6B 65 74 63 68 69 70 2D 75 6E 6B ips,rocketchip-unk
000000b4 6E 6F 77 6E 00 00 00 00 00 00 00 01 61 6C 69 61 73 65 nown........aliase
000000c6 73 00 00 00 00 03 00 00 00 15 00 00 00 2C 2F 73 6F 63 s...........,/soc
000000d8 2F 73 65 72 69 61 6C 40 31 30 30 30 30 30 30 30 00 00 /serial@10000000..
000000ea 00 00 00 00 00 02 00 00 00 01 63 68 6F 73 65 6E 00 00 ..........chosen..
000000fc 00 00 00 03 00 00 00 1A 00 00 00 34 63 6F 6E 73 6F 6C ...........4consol
0000010e 65 3D 68 76 63 30 20 65 61 72 6C 79 63 6F 6E 3D 73 62 e=hvc0 earlycon=sb
00000120 69 00 00 00 00 00 00 02 00 00 00 01 63 70 75 73 00 00 i...........cpus..
00000132 00 00 00 00 00 03 00 00 00 04 00 00 00 00 00 00 00 01 ..................
00000144 00 00 00 03 00 00 00 04 00 00 00 0F 00 00 00 00 00 00 ..................
00000156 00 03 00 00 00 04 00 00 00 3D 00 0F 42 40 00 00 00 01 .........=..B@....
00000168 63 70 75 40 30 00 00 00 00 00 00 03 00 00 00 04 00 00 cpu@0.............
0000017a 00 50 00 00 00 00 00 00 00 03 00 00 00 15 00 00 00 1B .P................
0000018c 73 69 66 69 76 65 2C 72 6F 63 6B 65 74 30 00 72 69 73 sifive,rocket0.ris
0000019e 63 76 00 00 00 00 00 00 00 03 00 00 00 04 00 00 00 60 cv...............`
000001b0 00 00 00 40 00 00 00 03 00 00 00 04 00 00 00 73 00 00 ...@...........s..
000001c2 00 40 00 00 00 03 00 00 00 04 00 00 00 80 00 00 10 00 .@................
000001d4 00 00 00 03 00 00 00 04 00 00 00 8D 63 70 75 00 00 00 ............cpu...
000001e6 00 03 00 00 00 04 00 00 00 99 00 00 00 01 00 00 00 03 ..................
000001f8 00 00 00 04 00 00 00 B8 00 00 00 40 00 00 00 03 00 00 ...........@......
0000020a 00 04 00 00 00 CB 00 00 00 40 00 00 00 03 00 00 00 04 .........@........
0000021c 00 00 00 D8 00 00 10 00 00 00 00 03 00 00 00 04 00 00 ..................
0000022e 00 E5 00 00 00 01 00 00 00 03 00 00 00 04 00 00 00 F6 ..................
00000240 00 00 00 00 00 00 00 03 00 00 00 09 00 00 00 FA 72 76 ................rv
00000252 33 32 69 6D 61 63 00 00 00 00 00 00 00 03 00 00 00 04 32imac............
```

1. **Boot sequence** (5/11) Call **.dtb** in `main()`

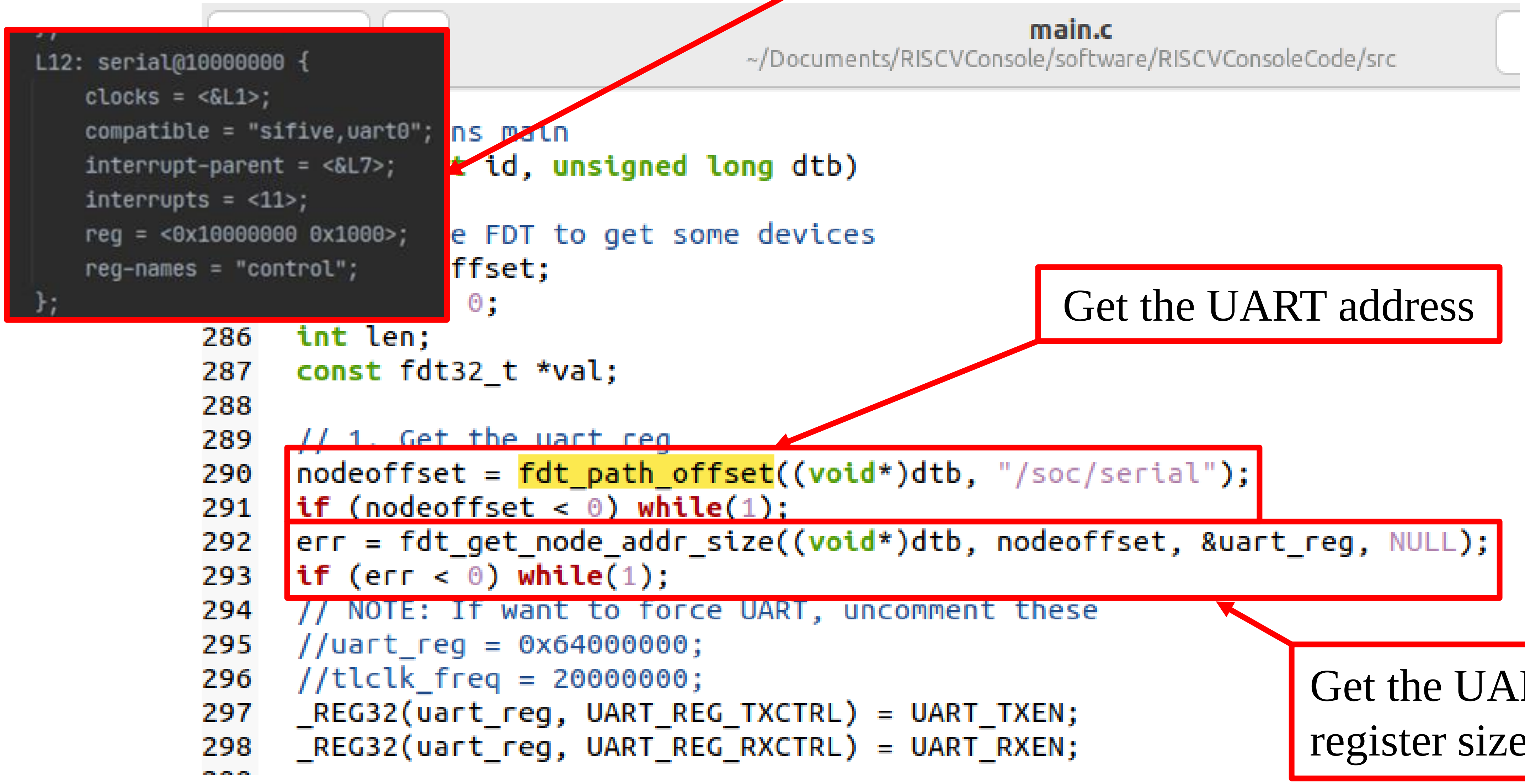
The pointer of **.dtb** will be passed to the `main()` by the bootloader.



```
//HART 0 runs main
int main(int id, unsigned long dtb)
{
  // Use the FDT to get some devices
  int nodeoffset;
  int err = 0;
  int len;
  const fdt32_t *val;

  // 1. Get the uart reg
  nodeoffset = fdt_path_offset((void*)dtb, "/soc/serial");
  if (nodeoffset < 0) while(1);
  err = fdt_get_node_addr_size((void*)dtb, nodeoffset, &uart_reg, NULL);
  if (err < 0) while(1);
  // NOTE: If want to force UART, uncomment these
  //uart_reg = 0x64000000;
  //tlclk_freq = 20000000;
  _REG32(uart_reg, UART_REG_TXCTRL) = UART_TXEN;
  _REG32(uart_reg, UART_REG_RXCTRL) = UART_RXEN;
```

1. Boot sequence (6/11) Call .dtb in `main()`

For example: to use the UART from the device-tree



```
L12: serial@10000000 {
    clocks = <&L1>;
    compatible = "sifive,uart0";
    interrupt-parent = <&L7>;
    interrupts = <11>;
    reg = <0x10000000 0x1000>;
    reg-names = "control";
};
```

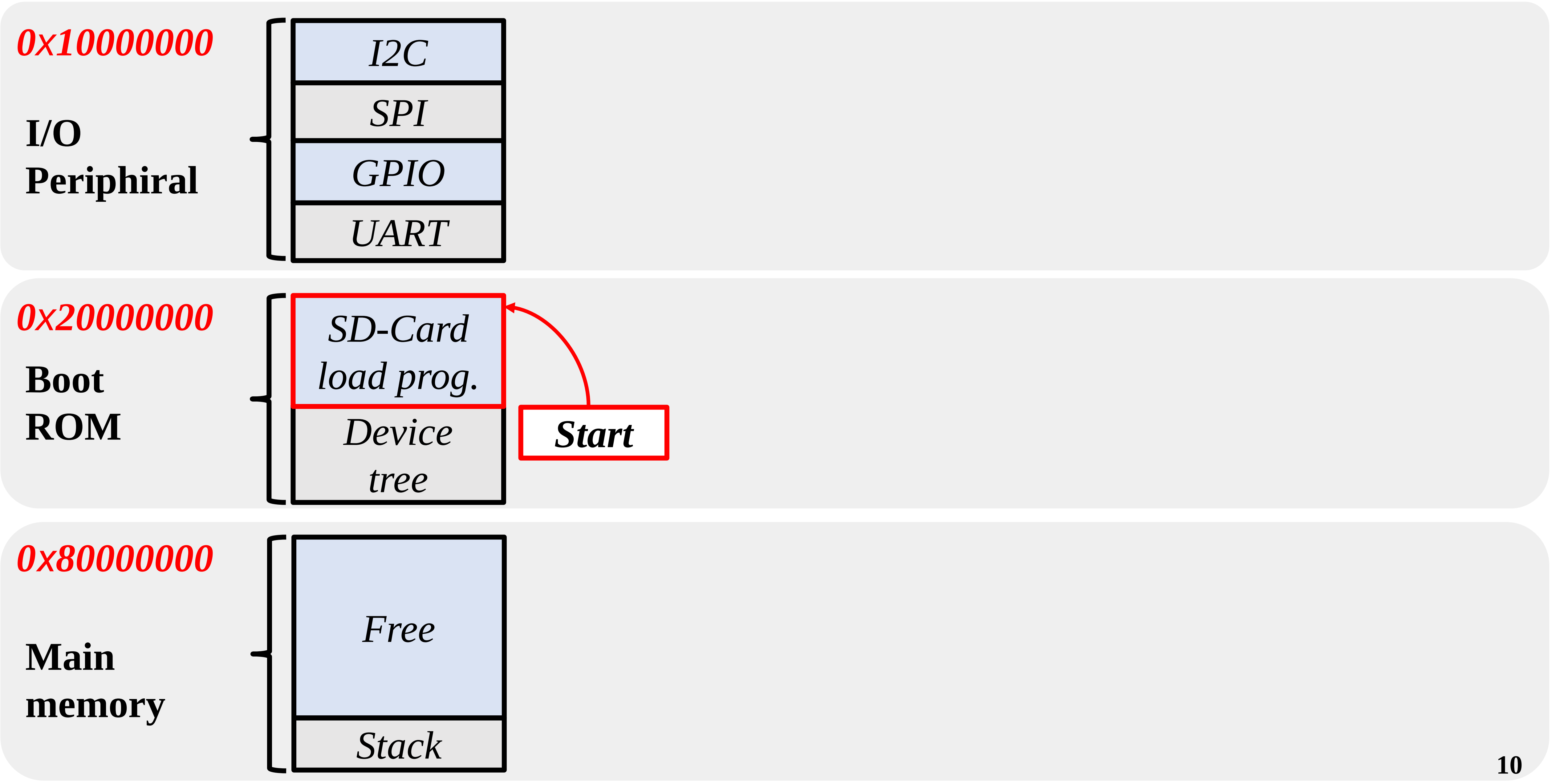
main.c
~/Documents/RISCVConsole/software/RISCVConsoleCode/src

```
ns main
 id, unsigned long dtb)

e FDT to get some devices
ffset;
 0;
    int len;
    const fdt32_t *val;

    // 1. Get the uart reg
    nodeoffset = fdt_path_offset((void*)dtb, "/soc/serial");
    if (nodeoffset < 0) while(1);
    err = fdt_get_node_addr_size((void*)dtb, nodeoffset, &uart_reg, NULL);
    if (err < 0) while(1);
    // NOTE: If want to force UART, uncomment these
    //uart_reg = 0x64000000;
    //tlclk_freq = 20000000;
    _REG32(uart_reg, UART_REG_TXCTRL) = UART_TXEN;
    _REG32(uart_reg, UART_REG_RXCTRL) = UART_RXEN;
```

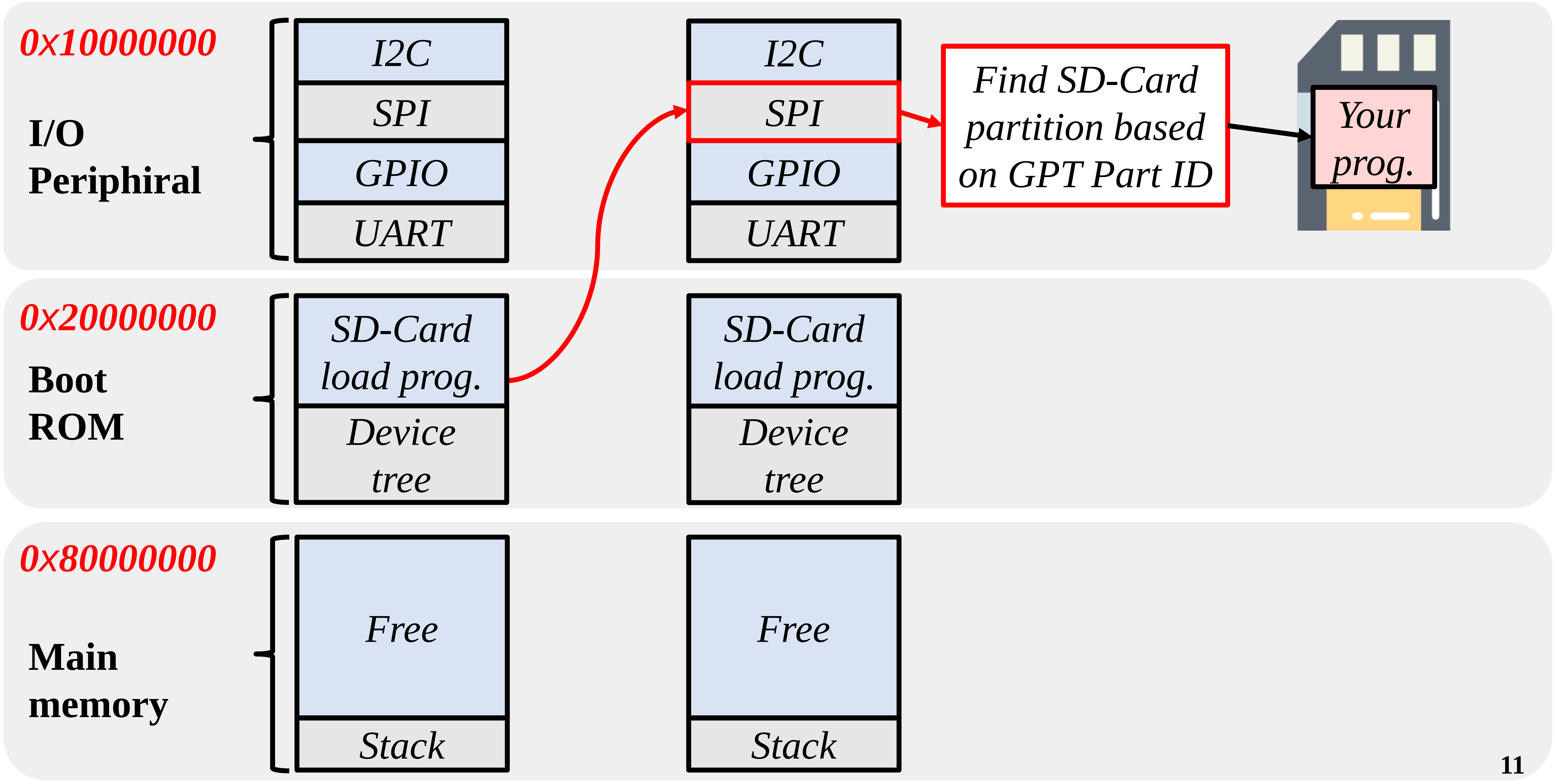
Get the UART address

Get the UART register size

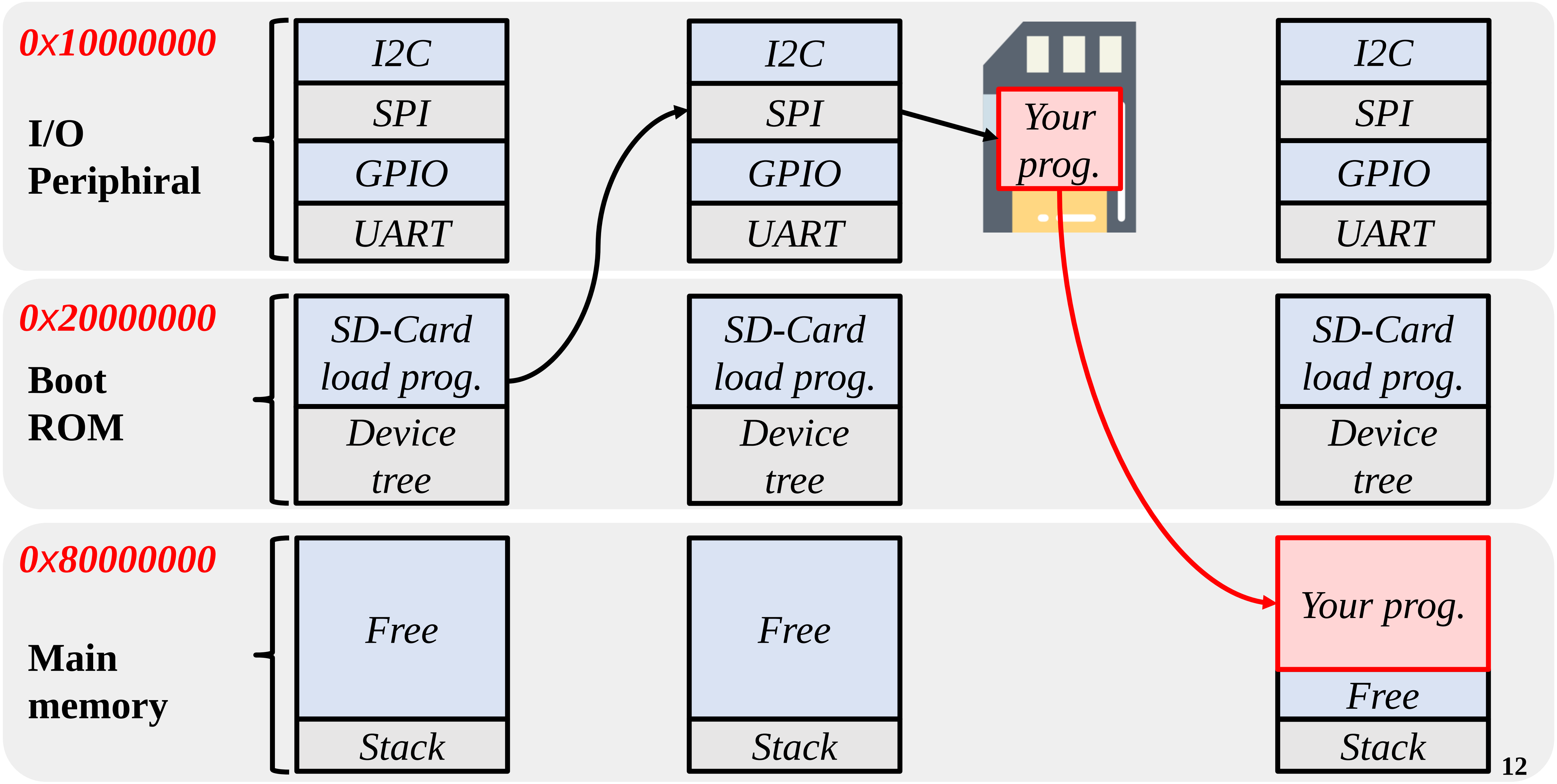
1. **Boot sequence** (7/11) Our boot flow: start at boot ROM



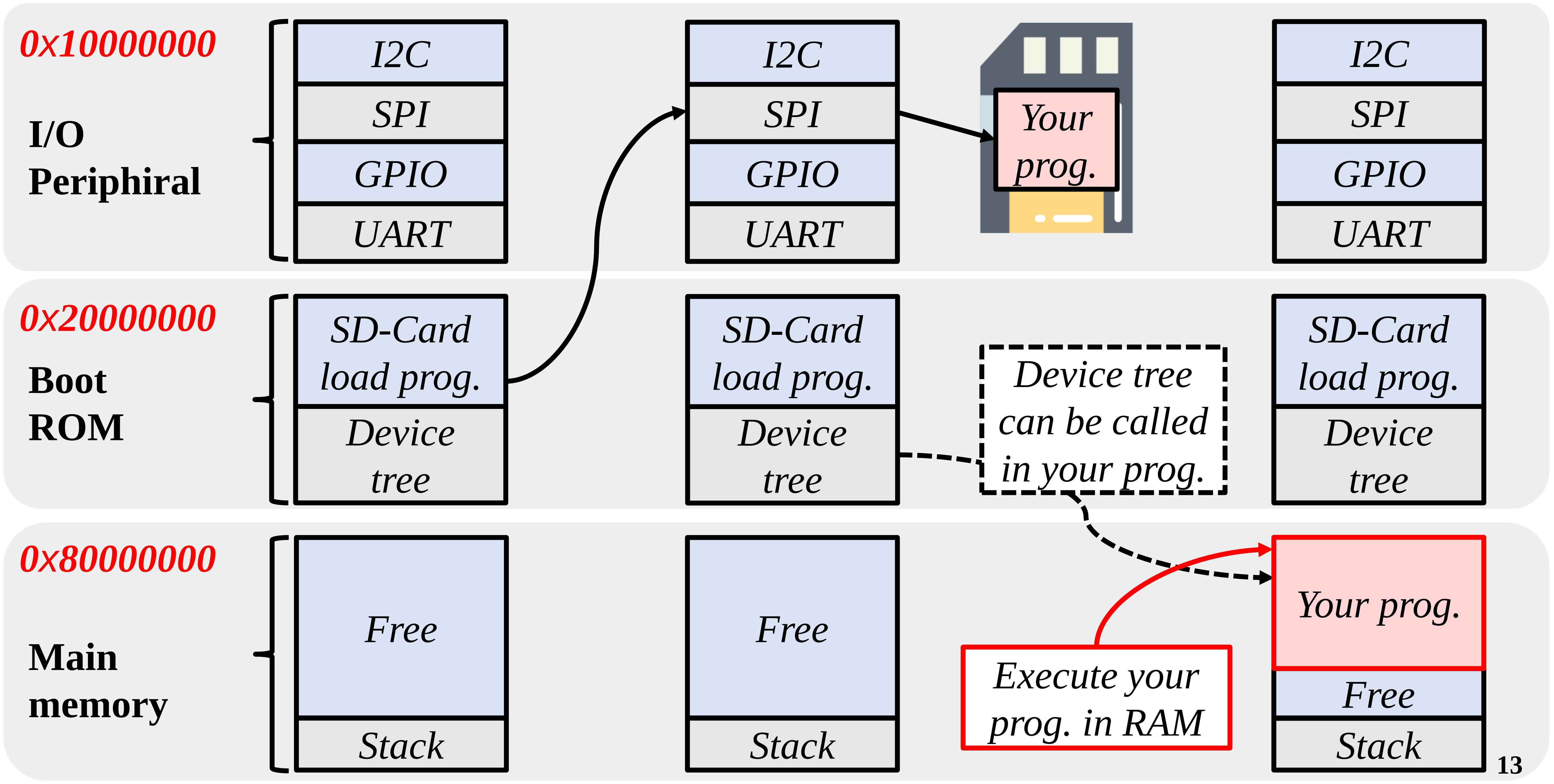
1. Boot sequence (8/11) Our boot flow: find SD-card partition



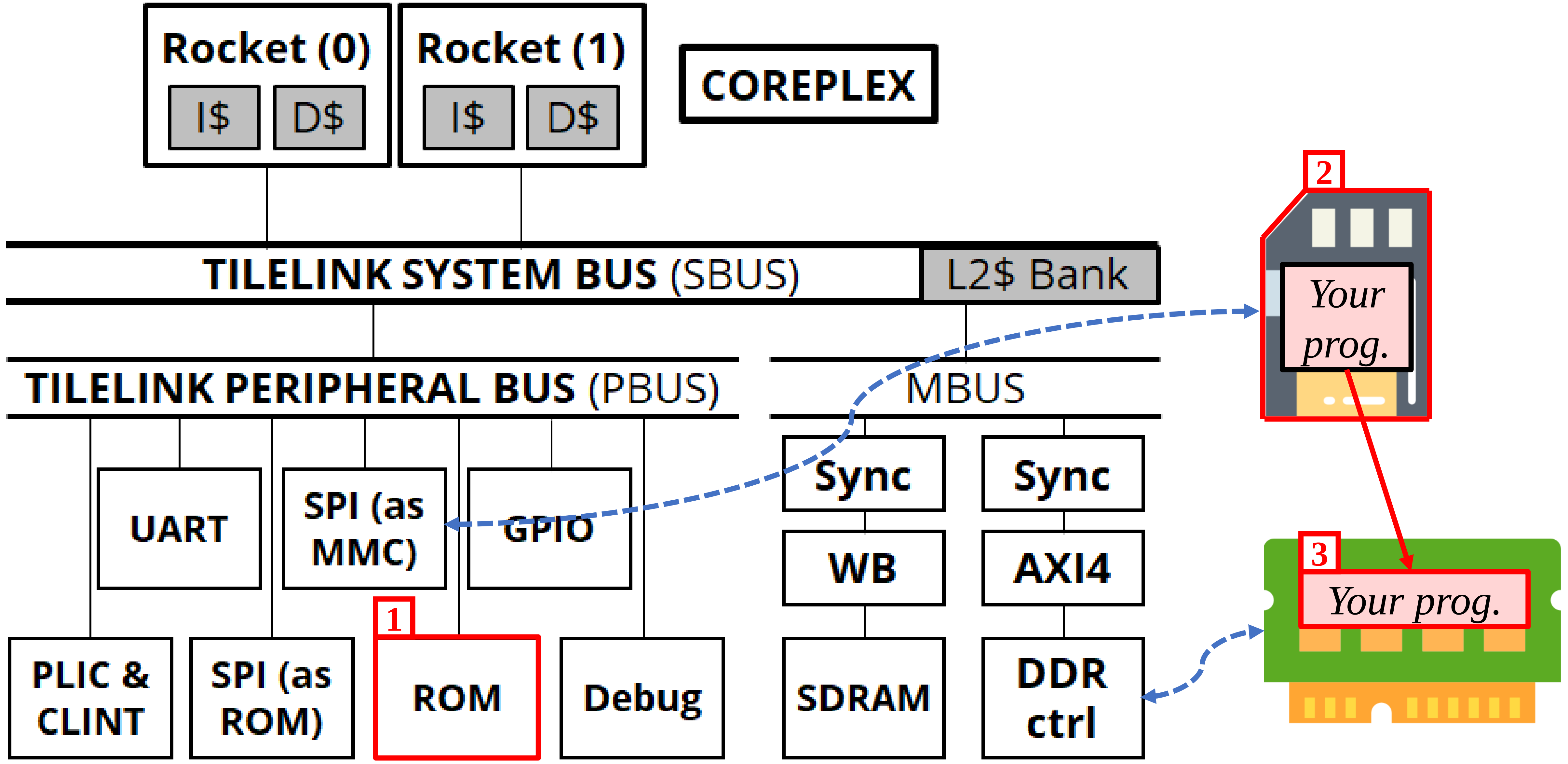
1. Boot sequence (9/11) Our boot flow: copy to RAM



1. Boot sequence (10/11) Our boot flow: execute in RAM



1. **Boot sequence** (11/11) Our boot flow: architecture view



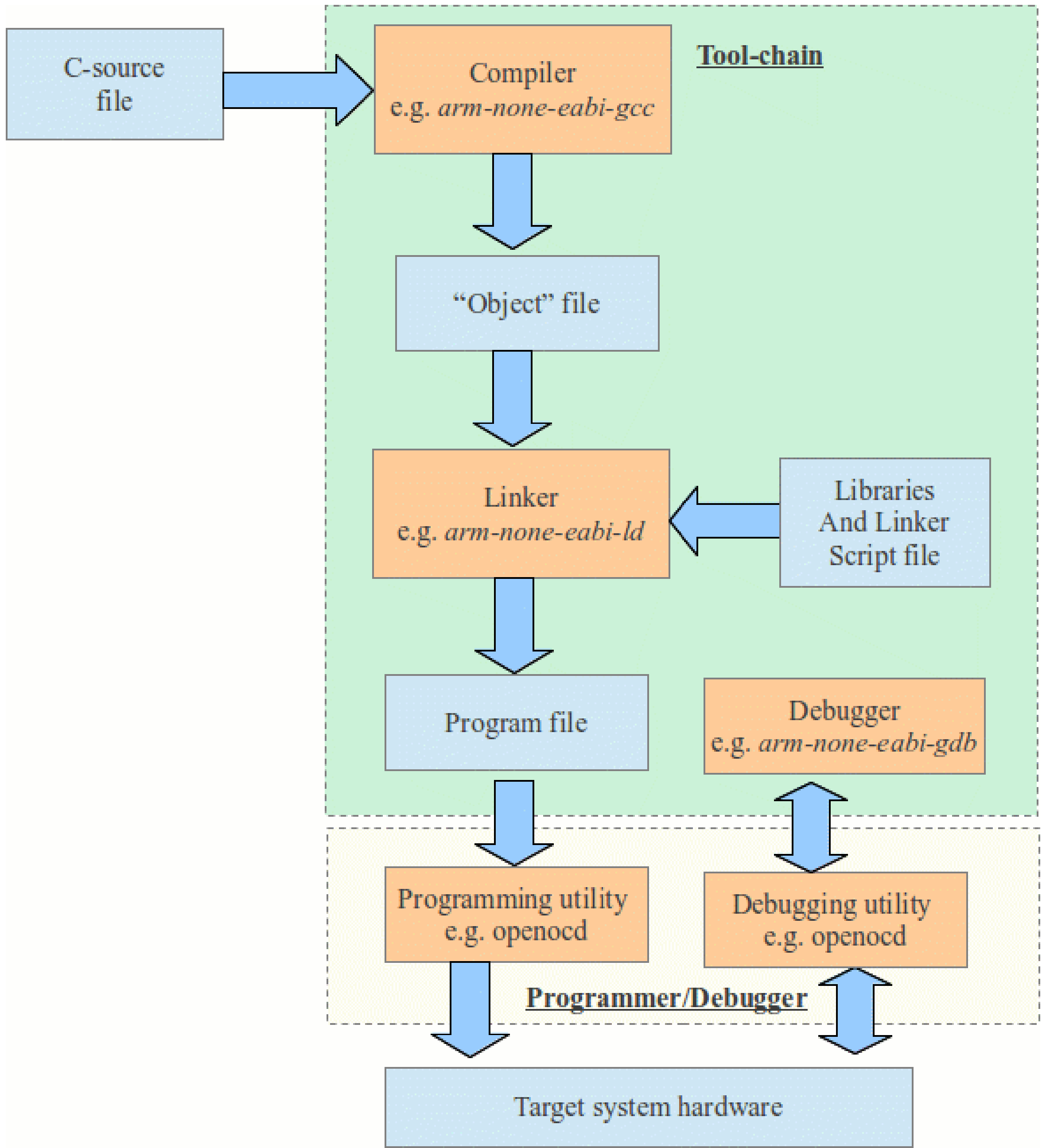
国立大学法人 電気通信大学 The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

2. RISC-V & RISC-V ISA (1/7) Toolchain


C-source file
Compiler e.g. arm-none-eabi-gcc
Tool-chain
"Object" file
Linker e.g. arm-none-eabi-ld
Libraries And Linker Script file
Program file
Debugger e.g. arm-none-eabi-gdb
Programming utility e.g. openocd
Debugging utility e.g. openocd
Programmer/Debugger
Target system hardware

- To compile the software, we need the toolchain.
- Toolchain comes with its **I**nstruction **S**et **A**rchitecture (**ISA**).
- *Each ISA has its own toolchain.*

Three most important tools in any toolchain

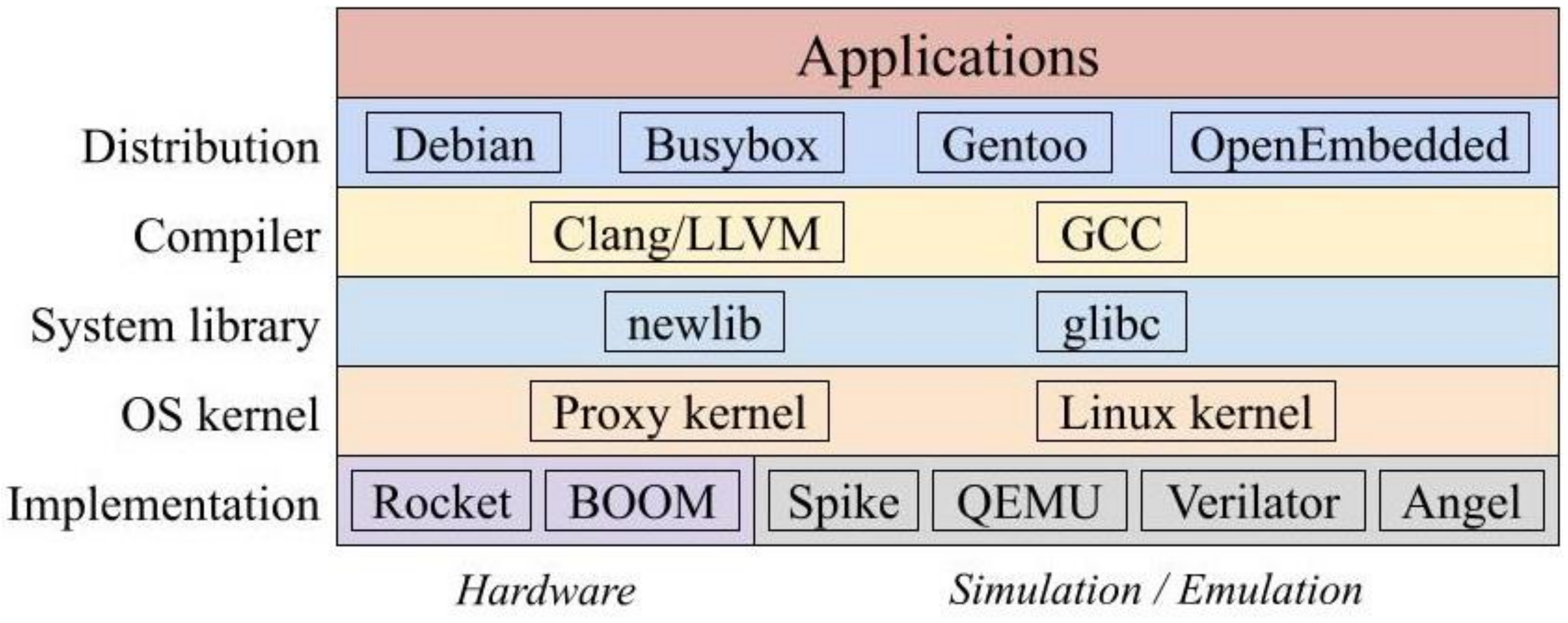
- **GCC:** *(cross C compiler)* makes a C code into assembly code
- **LD:** *(linker)* links standard libraries into the build; also links between multiple C files
- **GDB:** *(debugger)* debug the hardware/simulator/emulator

RISC-V is an ISA.

Other common ISAs: i386, amd64, ARM 32/64, AVR, MIPS, etc.

2. **RISC-V & RISC-V ISA** (2/7) RISC-V toolchain

RISC-V toolchain and its ecosystem



(User's) software in the top
OS file system
Compiler
Standard libraries
OS kernel
Hardware in the bottom

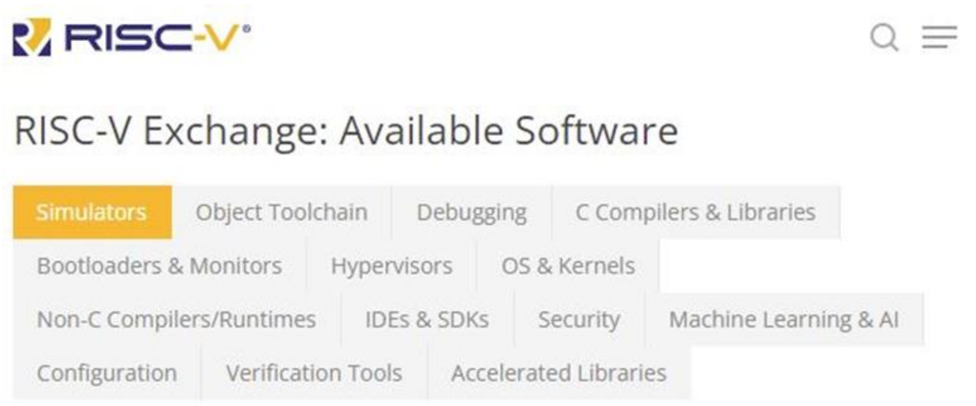
Top-down explanation:

User's applications on the top are operated in an OS file system, which then compiled by a compiler based on multiple standard libraries. After compiled, the execution file is run on the OS kernel that manages the hardware in the bottom.

2. RISC-V & RISC-V ISA (3/7) RISC-V ISA

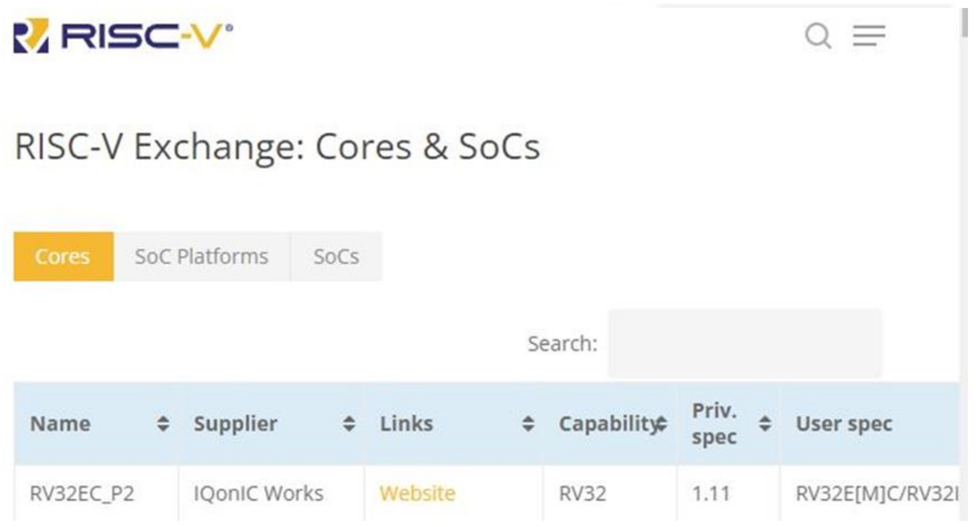
Open-source **RISC-V** means open-source **ISA**, no more, no less.

RISC-V Foundation: https://riscv.org/



RISC-V Exchange: Available Software

Simulators | Object Toolchain | Debugging | C Compilers & Libraries | Bootloaders & Monitors | Hypervisors | OS & Kernels | Non-C Compilers/Runtimes | IDEs & SDKs | Security | Machine Learning & AI | Configuration | Verification Tools | Accelerated Libraries



RISC-V Exchange: Cores & SoCs

Cores | SoC Platforms | SoCs

Search:

Name	Supplier	Links	Capability	Priv. spec	User spec
RV32EC_P2	IQonIC Works	Website	RV32	1.11	RV32E[M]C/RV32I

- Official released ISA specification
- Many cores, SoCs, & software are available for free
- Developers can reuse each other designs & tools
→ significantly reducing R&D time and effort

License free:

- RISC-V ISA
- RISC-V toolchain

License depends on authors/developers:

- RISC-V processors
- RISC-V software applications
- RISC-V-related products

2. RISC-V & RISC-V ISA (4/7) RISC-V extensions

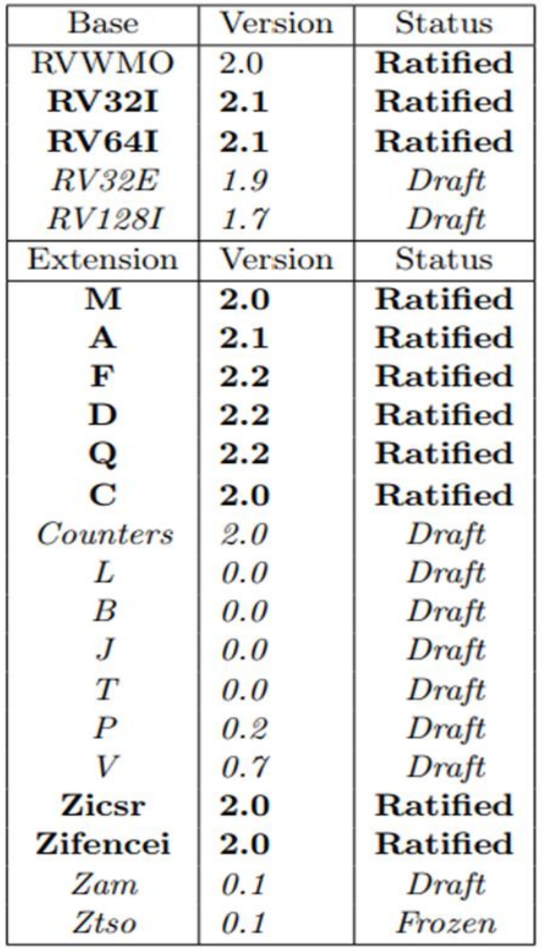
What makes **RISC-V** different: <u>its modular mindset</u>

Int	Mul	Atomic	Float	Double	Compress

Base instruction set: **I**nteger

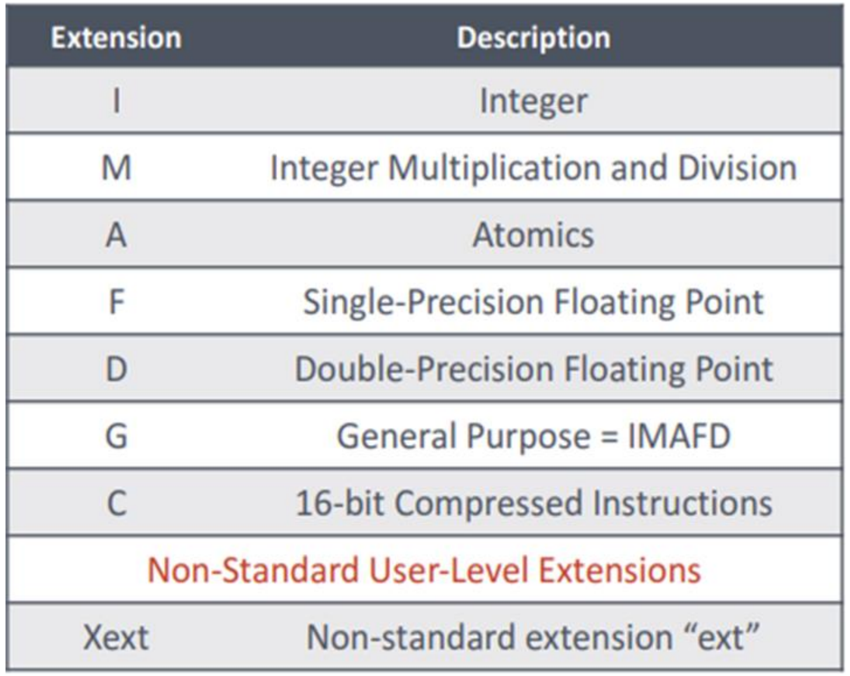
Extended instruction set: *the rest*

There are also <u>a lot more</u> than just **IMAFDC :**



Base	Version	Status
RVWMO	2.0	Ratified
RV32I	**2.1**	**Ratified**
RV64I	**2.1**	**Ratified**
RV32E	*1.9*	*Draft*
RV128I	*1.7*	*Draft*
Extension	Version	Status
M	**2.0**	**Ratified**
A	**2.1**	**Ratified**
F	**2.2**	**Ratified**
D	**2.2**	**Ratified**
Q	**2.2**	**Ratified**
C	**2.0**	**Ratified**
Counters	*2.0*	*Draft*
L	*0.0*	*Draft*
B	*0.0*	*Draft*
J	*0.0*	*Draft*
T	*0.0*	*Draft*
P	*0.2*	*Draft*
V	*0.7*	*Draft*
Zicsr	**2.0**	**Ratified**
Zifencei	**2.0**	**Ratified**
Zam	*0.1*	*Draft*
Ztso	*0.1*	*Frozen*

modular architecture helps fine-tune the performance based on developer's needs



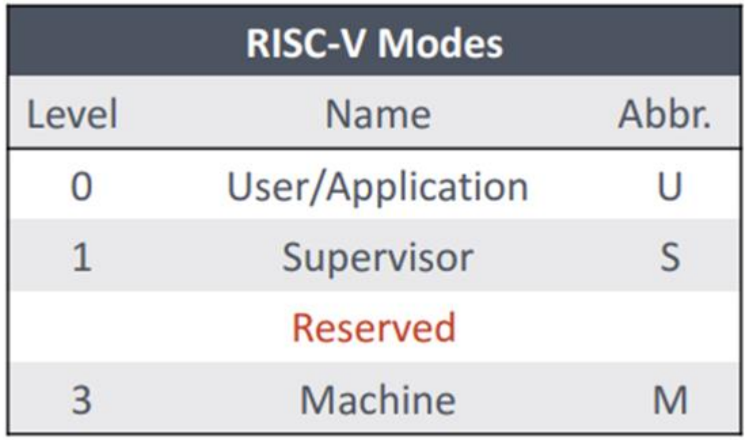
Extension	Description
I	Integer
M	Integer Multiplication and Division
A	Atomics
F	Single-Precision Floating Point
D	Double-Precision Floating Point
G	General Purpose = IMAFD
C	16-bit Compressed Instructions
Non-Standard User-Level Extensions	
Xext	Non-standard extension "ext"

The most common extensions: **IMAFDC** *(also known as* ***GC****)*

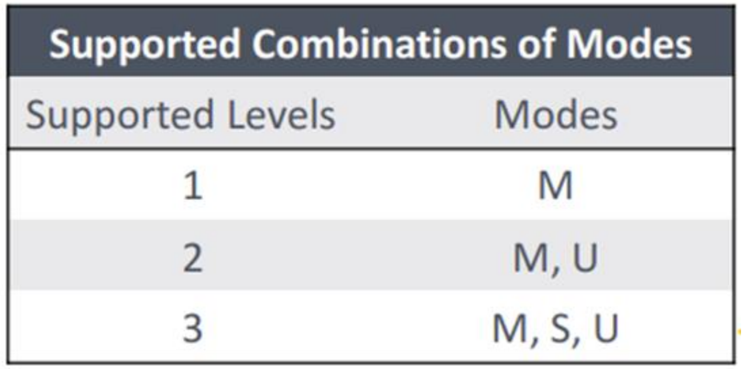
2. RISC-V & RISC-V ISA (5/7) RISC-V OS stack

To support an Operating System (OS), the ISA has to support the OS stack *or the **M-/S-/U-mode**.*

RISC-V privileged architecture:

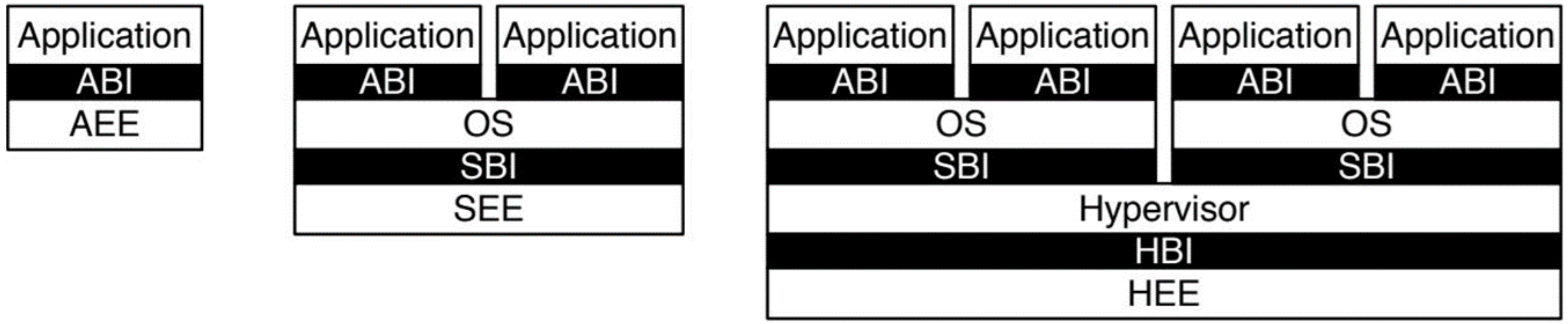


RISC-V Modes		
Level	Name	Abbr.
0	User/Application	U
1	Supervisor	S
	Reserved	
3	Machine	M



Supported Combinations of Modes	
Supported Levels	Modes
1	M
2	M, U
3	M, S, U

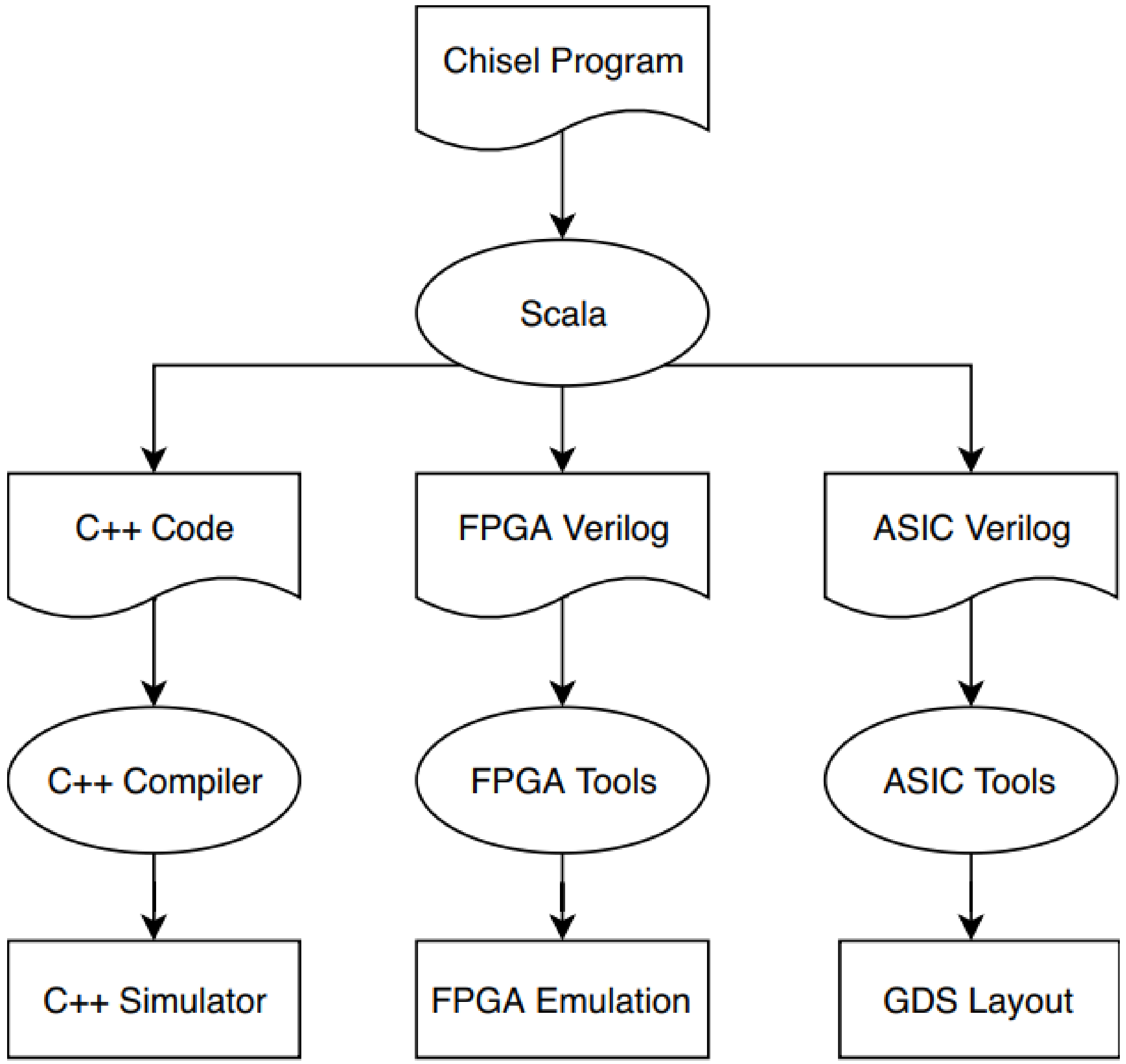
Different scenarios of utilizing the OS stack:



RISC-V ISA not only supports the OS stack, but also provides a **privileged architecture**.

→ Better security scheme by having the hardware recognize different codes executed at different modes.

2. RISC-V & RISC-V ISA (6/7) CHISEL



Chisel is a library.
Scala is a language.

- **Scala** itself is a *high-level object-oriented programming language*
  → It is not designed for "hardware coding."

- **Chisel** is a library attached to Scala to define a set of coding rules.
  → It is designed for "hardware coding."

- From **Scala** to **Verilog**:
  Scala → Java → FIRRTL → Verilog
  1st arrow: Scala compiler named SBT
  2nd arrow: executing Java
  3rd arrow: FIRRTL compiler

2. RISC-V & RISC-V ISA (7/7) RISC-V key contributions

RISC-V revolutionizes Computer System Design

1. **Modular at heart:**
   customizable ISA and customizable hardware
   → fine-tune the system to your specific needs.

2. **Open-source community:**
   license-free ISA, open cores and SoCs, open-source libraries, open-source software, etc. → reuse other developers' designs → save time and effort for R&D

3. **CHISEL** *(**C**onstructing **H**ardware **I**n **S**cala **E**mbedded **L**anguage)***:**
   a new way to "coding" hardware circuits. When compiled, it will generate a true RTL Verilog code.
   → a "meta-programming" language for hardware developers with parameters and sub-designs that can be overridden or extended.
   → easy to develop "object-oriented" hardware library for reuse purpose.

国立大学法人
電気通信大学
The University of Electro-Communications

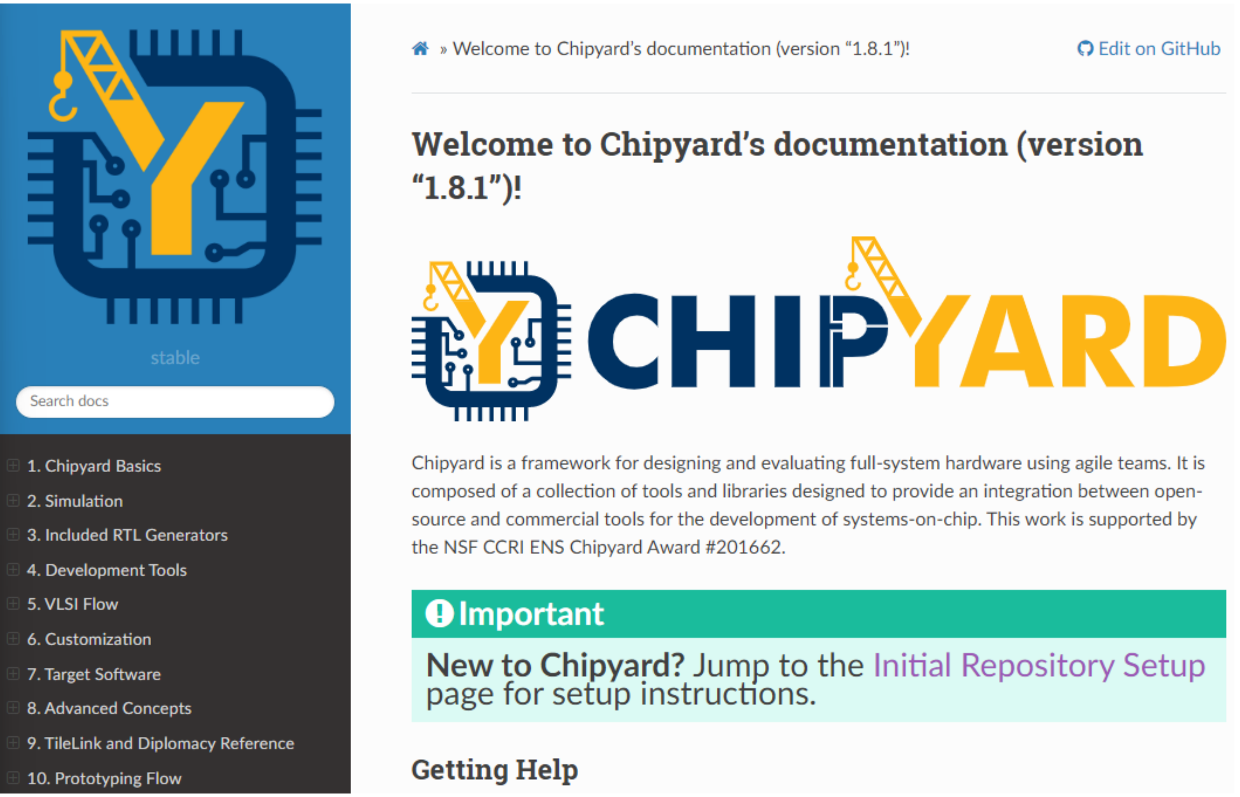
Pham Laboratory
Integrated circuit design laboratory

Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

3. Our RISC-V computer system (1/10) Libraries

Our hardware was built using the **Chipyard** *library*.
And the core *processor* was the **Rocket**.



1. Chipyard Basics
2. Simulation
3. Included RTL Generators
4. Development Tools
5. VLSI Flow
6. Customization
7. Target Software
8. Advanced Concepts
9. TileLink and Diplomacy Reference
10. Prototyping Flow

» Welcome to Chipyard's documentation (version "1.8.1")! Edit on GitHub

Welcome to Chipyard's documentation (version "1.8.1")!

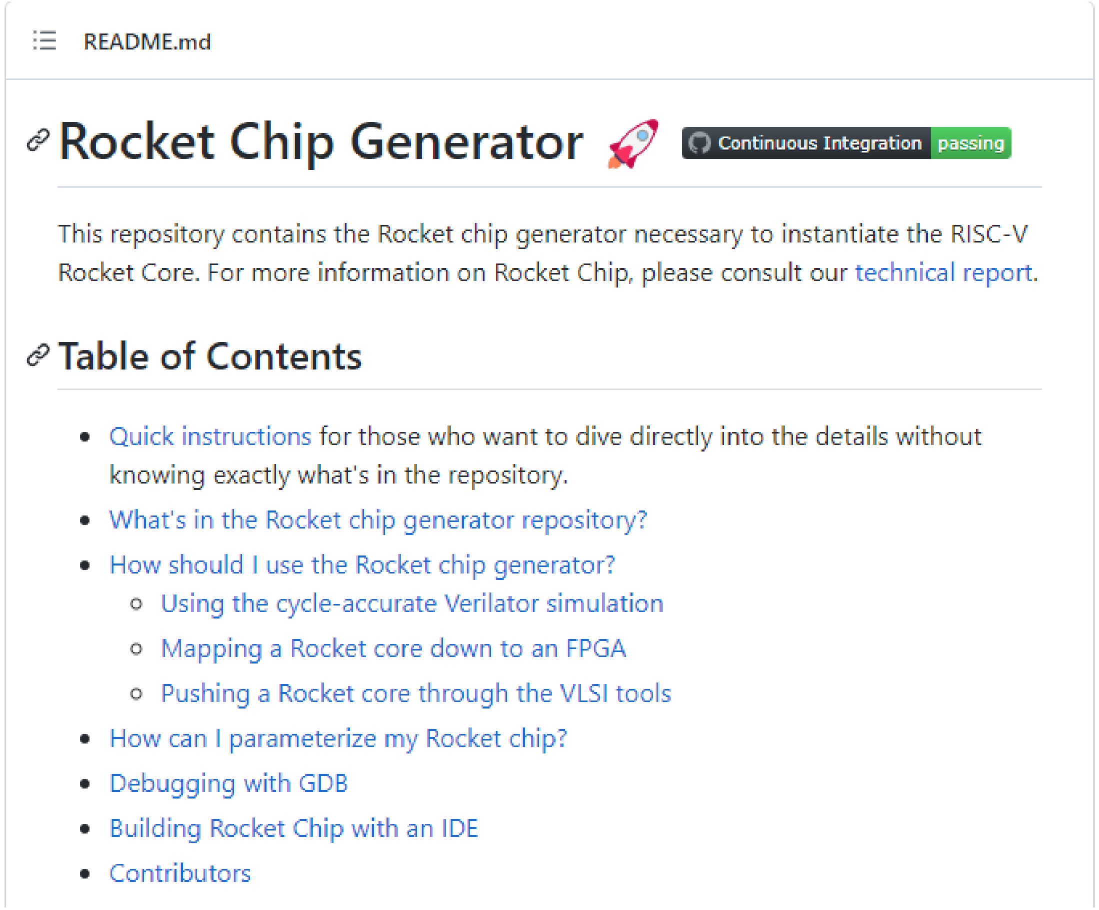
Chipyard is a framework for designing and evaluating full-system hardware using agile teams. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of systems-on-chip. This work is supported by the NSF CCRI ENS Chipyard Award #201662.

Important

New to Chipyard? Jump to the Initial Repository Setup page for setup instructions.

Getting Help

- Github: https://github.com/ucb-bar/chipyard


README.md

Rocket Chip Generator

This repository contains the Rocket chip generator necessary to instantiate the RISC-V Rocket Core. For more information on Rocket Chip, please consult our technical report.

Table of Contents

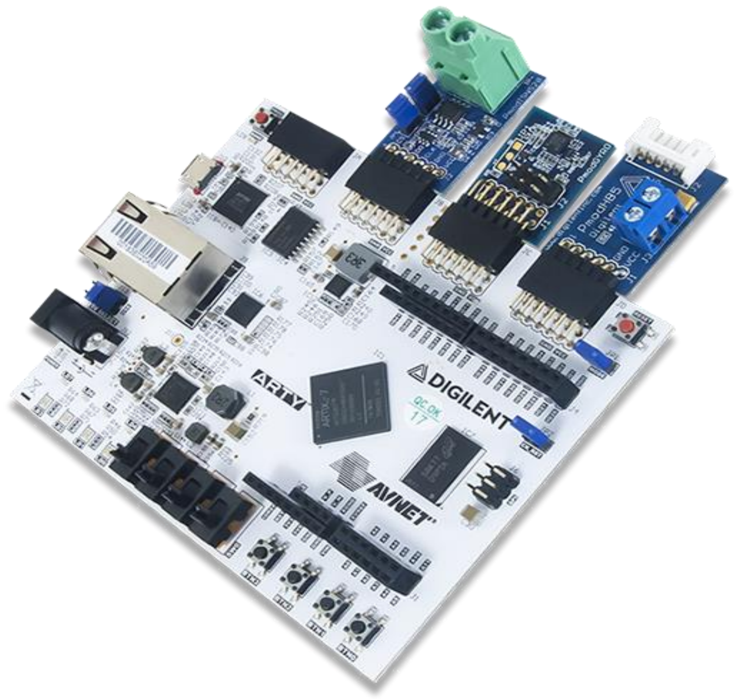
- Quick instructions for those who want to dive directly into the details without knowing exactly what's in the repository.
- What's in the Rocket chip generator repository?
- How should I use the Rocket chip generator?
  - Using the cycle-accurate Verilator simulation
  - Mapping a Rocket core down to an FPGA
  - Pushing a Rocket core through the VLSI tools
- How can I parameterize my Rocket chip?
- Debugging with GDB
- Building Rocket Chip with an IDE
- Contributors

- Github: https://github.com/chipsalliance/rocket-chip

3. Our RISC-V computer system (2/10) FPGA

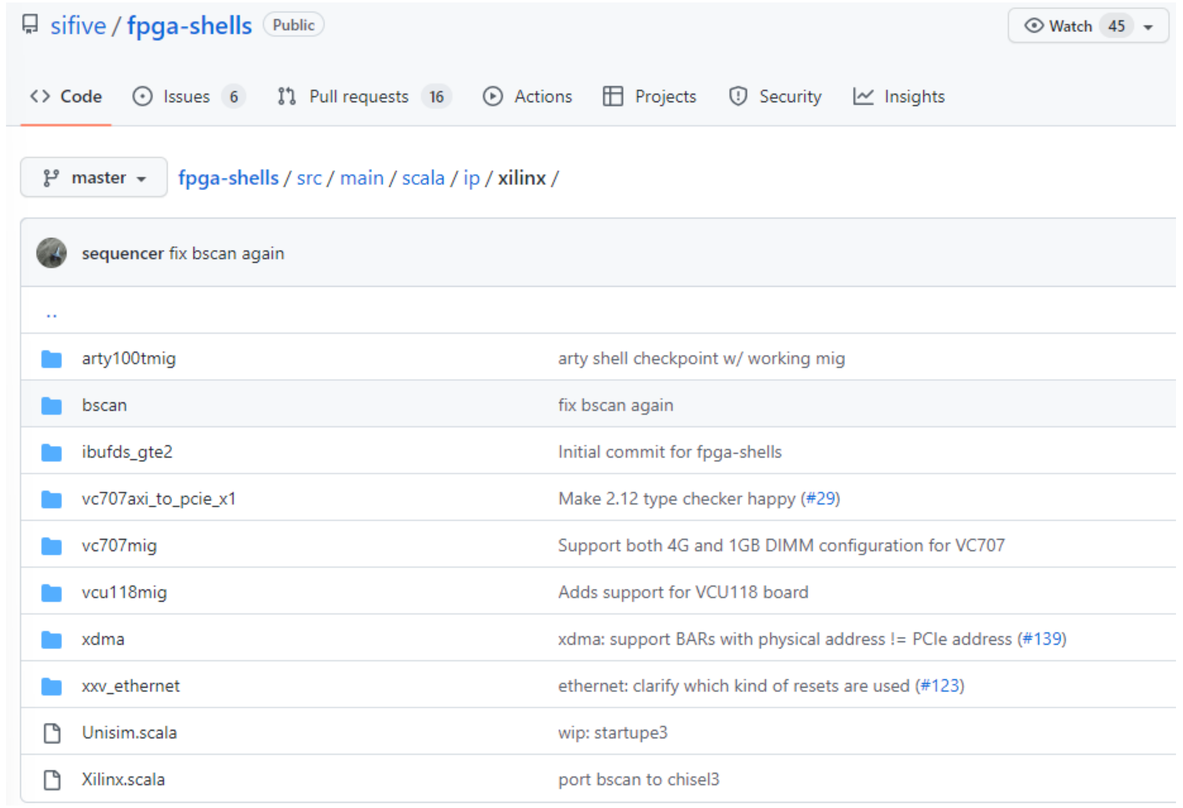
Arty-A7 was the *FPGA* chosen to be our example implementation.
Its IP was utilized by using the **fpga-shells** *library* from SiFive.

Arty-A7 FPGA



- Link: https://digilent.com/reference/programmable-logic/arty-a7/start

fpga-shells library



sifive / fpga-shells Public

Watch 45

Code | Issues 6 | Pull requests 16 | Actions | Projects | Security | Insights

master

fpga-shells / src / main / scala / ip / xilinx /

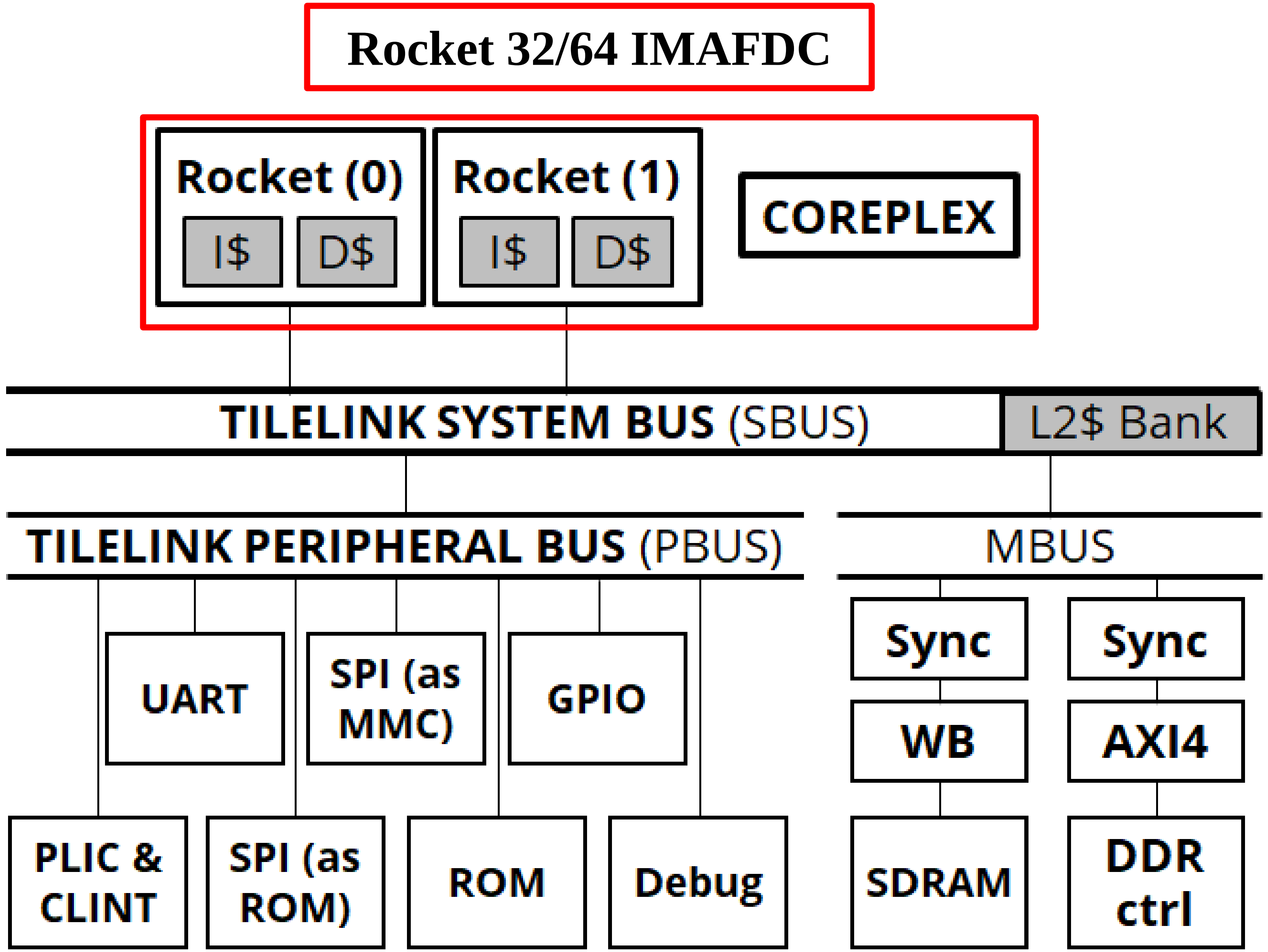
sequencer fix bscan again

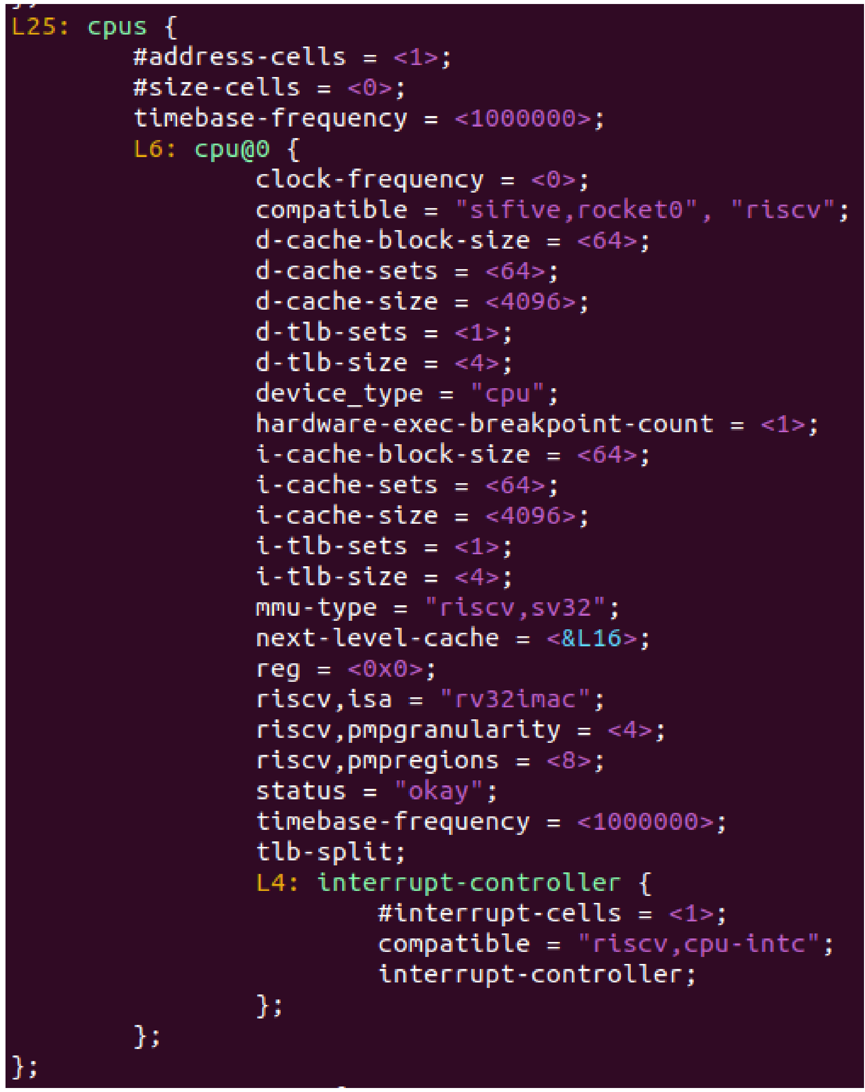
..

arty100tmig	arty shell checkpoint w/ working mig
bscan	fix bscan again
ibufds_gte2	Initial commit for fpga-shells
vc707axi_to_pcie_x1	Make 2.12 type checker happy (#29)
vc707mig	Support both 4G and 1GB DIMM configuration for VC707
vcu118mig	Adds support for VCU118 board
xdma	xdma: support BARs with physical address != PCIe address (#139)
xxv_ethernet	ethernet: clarify which kind of resets are used (#123)
Unisim.scala	wip: startupe3
Xilinx.scala	port bscan to chisel3

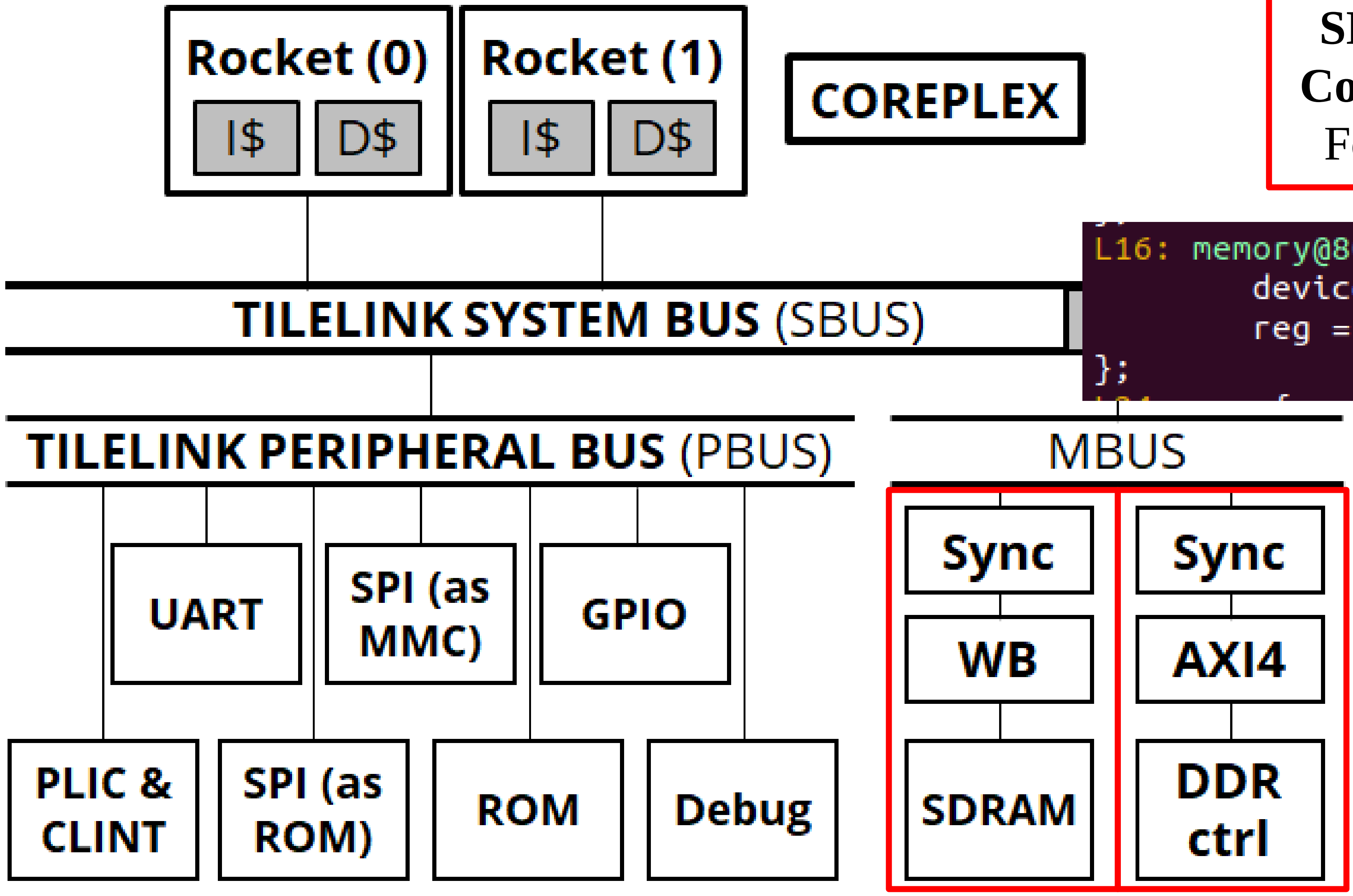
- Github: https://github.com/sifive/fpga-shells

3. Our RISC-V computer system (3/10) Processor

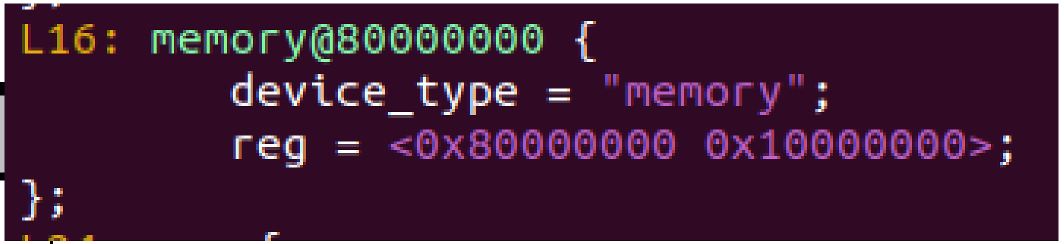



```
L25: cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        timebase-frequency = <1000000>;
        L6: cpu@0 {
                clock-frequency = <0>;
                compatible = "sifive,rocket0", "riscv";
                d-cache-block-size = <64>;
                d-cache-sets = <64>;
                d-cache-size = <4096>;
                d-tlb-sets = <1>;
                d-tlb-size = <4>;
                device_type = "cpu";
                hardware-exec-breakpoint-count = <1>;
                i-cache-block-size = <64>;
                i-cache-sets = <64>;
                i-cache-size = <4096>;
                i-tlb-sets = <1>;
                i-tlb-size = <4>;
                mmu-type = "riscv,sv32";
                next-level-cache = <&L16>;
                reg = <0x0>;
                riscv,isa = "rv32imac";
                riscv,pmpgranularity = <4>;
                riscv,pmpregions = <8>;
                status = "okay";
                timebase-frequency = <1000000>;
                tlb-split;
                L4: interrupt-controller {
                        #interrupt-cells = <1>;
                        compatible = "riscv,cpu-intc";
                        interrupt-controller;
                };
        };
};
```

3. Our RISC-V computer system (4/10) Memory

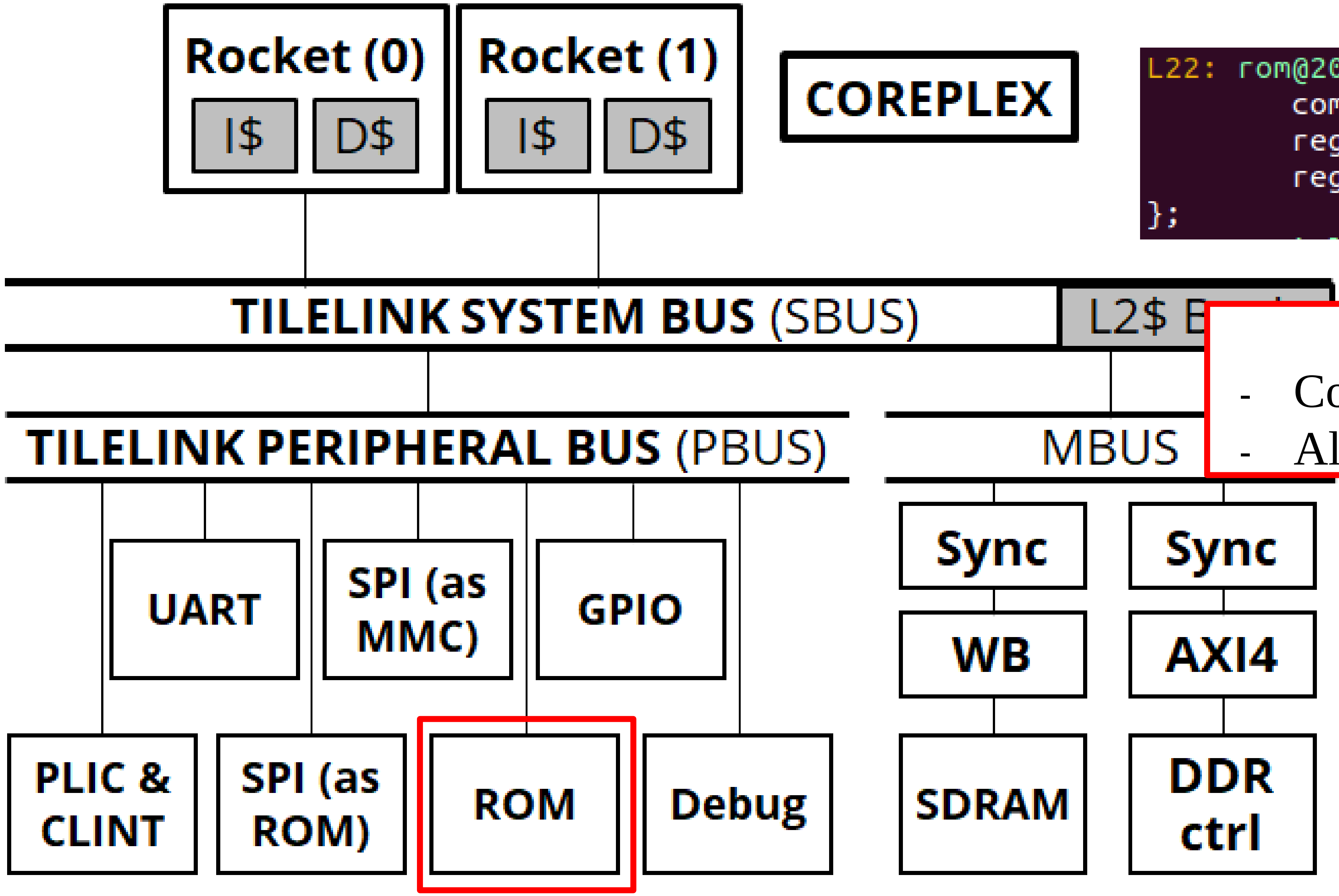


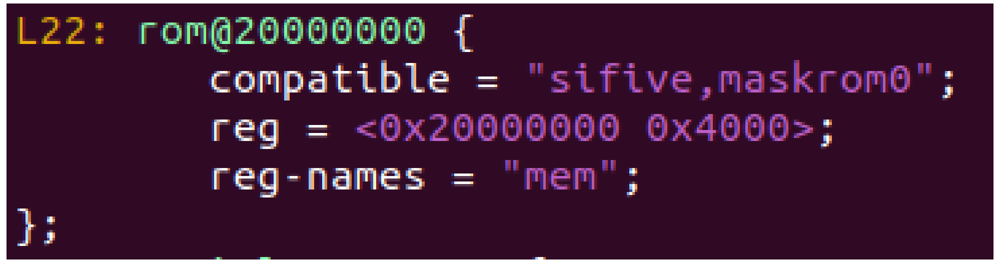
SDRAM Controller For DE2	**DDR3 Controller** For ArtyA7



```
L16: memory@80000000 {
        device_type = "memory";
        reg = <0x80000000 0x10000000>;
};
```

3. Our RISC-V computer system (5/10) Boot ROM



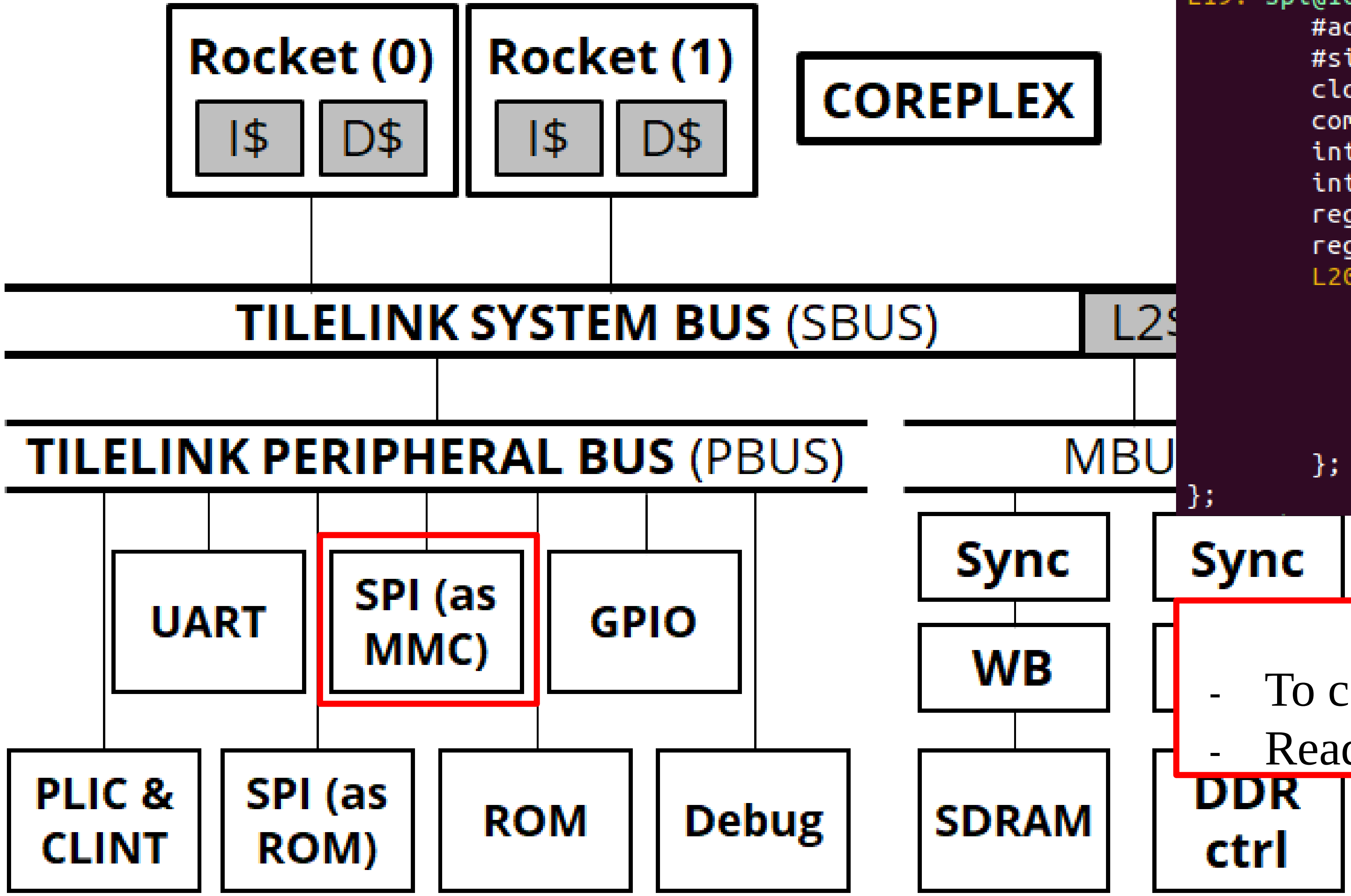


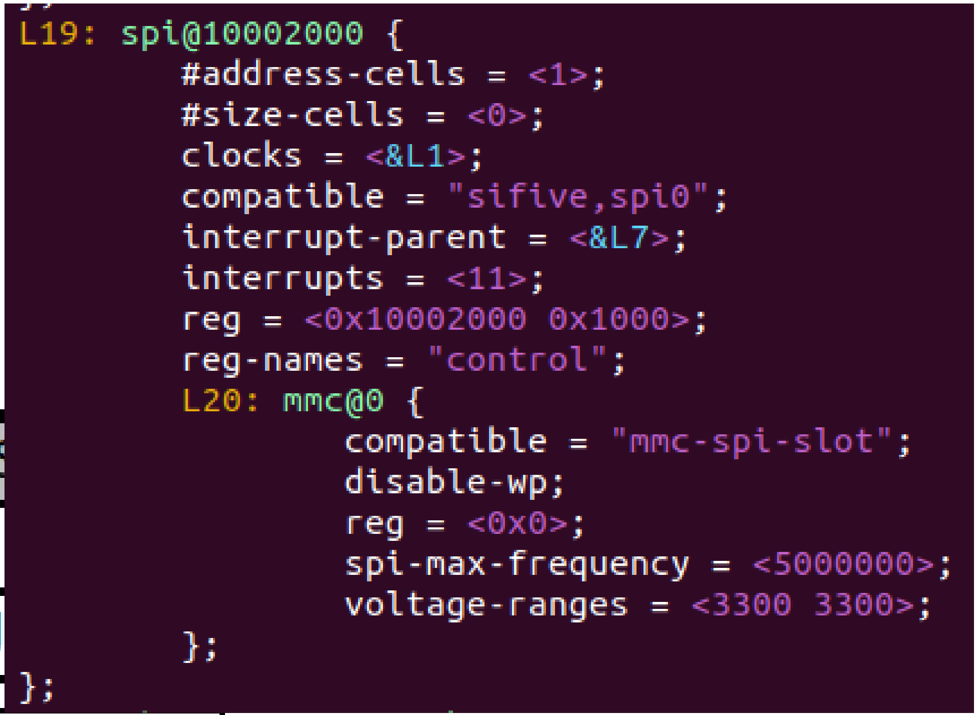
```
L22: rom@20000000 {
        compatible = "sifive,maskrom0";
        reg = <0x20000000 0x4000>;
        reg-names = "mem";
};
```

ROM

- Contains the SD load boot
- Always fixed

3. Our RISC-V computer system (6/10) SD-card

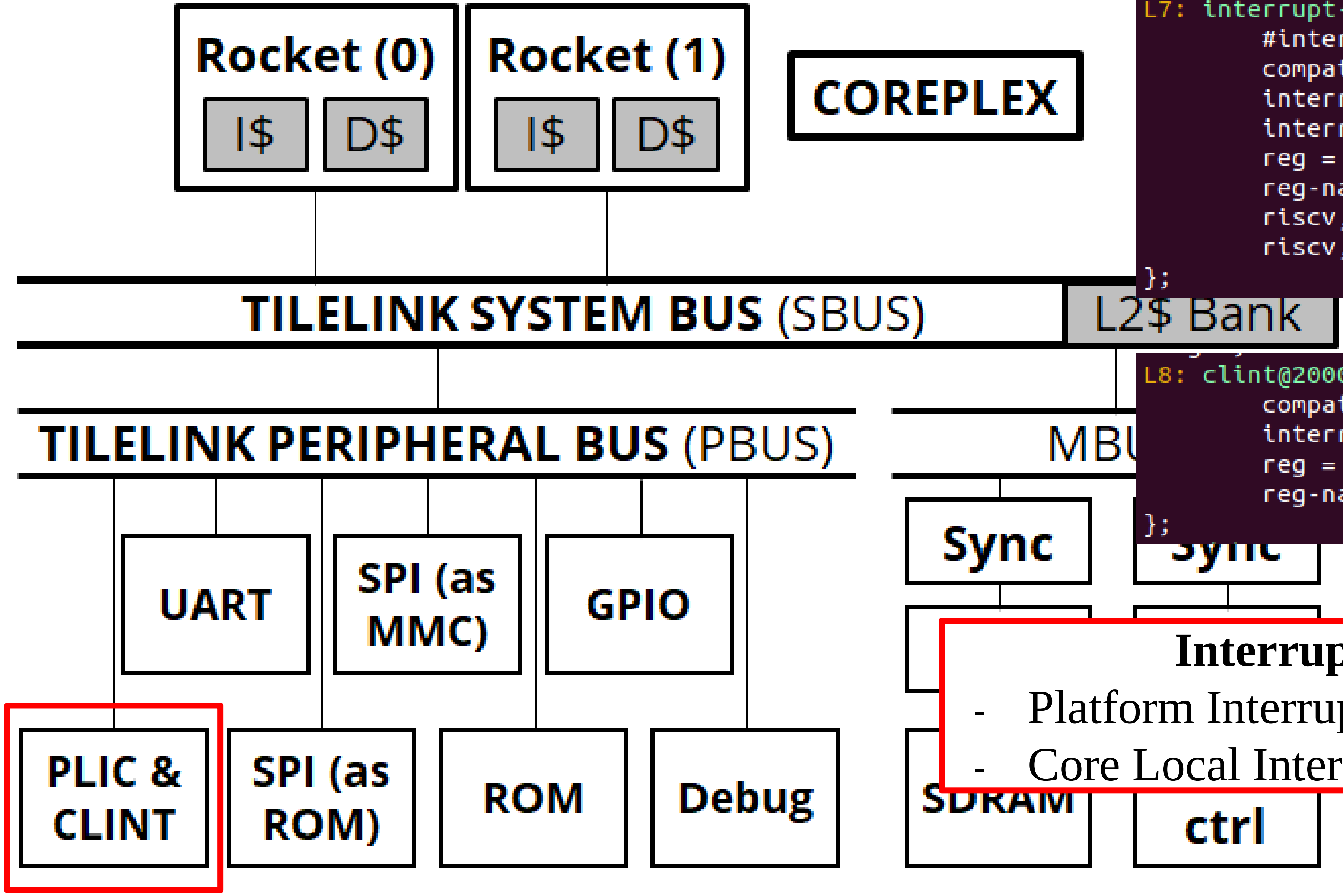


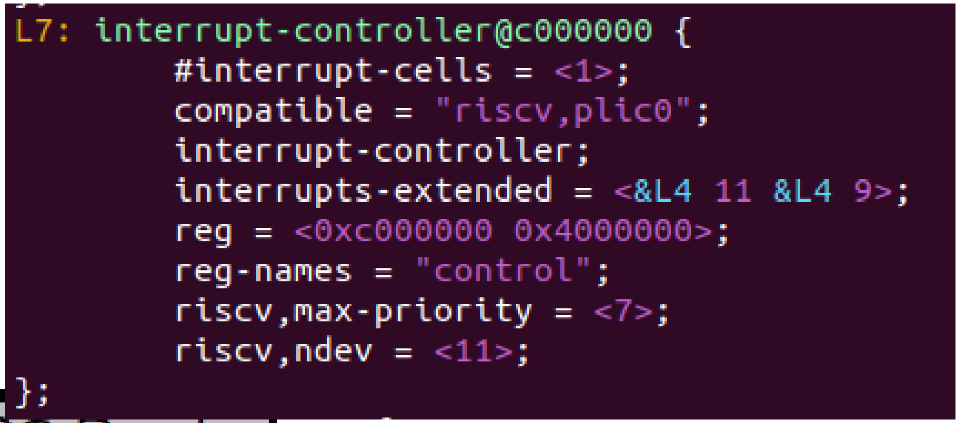

```
L19: spi@10002000 {
        #address-cells = <1>;
        #size-cells = <0>;
        clocks = <&L1>;
        compatible = "sifive,spi0";
        interrupt-parent = <&L7>;
        interrupts = <11>;
        reg = <0x10002000 0x1000>;
        reg-names = "control";
        L20: mmc@0 {
                compatible = "mmc-spi-slot";
                disable-wp;
                reg = <0x0>;
                spi-max-frequency = <5000000>;
                voltage-ranges = <3300 3300>;
        };
};
```

SPI Controller

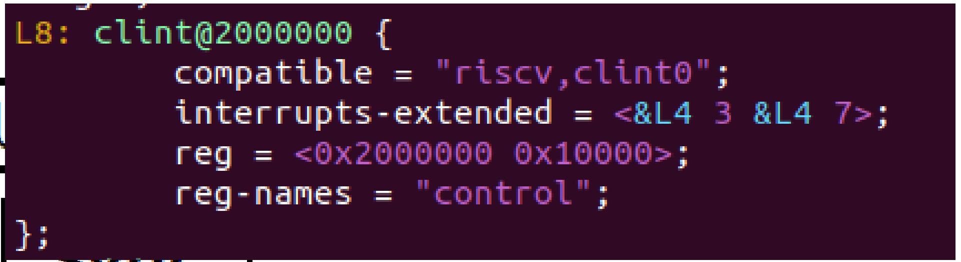
- To communicate with SD cards
- Read the partition and execute

3. Our RISC-V computer system (7/10) Interrupt





```
L7: interrupt-controller@c000000 {
        #interrupt-cells = <1>;
        compatible = "riscv,plic0";
        interrupt-controller;
        interrupts-extended = <&L4 11 &L4 9>;
        reg = <0xc000000 0x4000000>;
        reg-names = "control";
        riscv,max-priority = <7>;
        riscv,ndev = <11>;
};
```

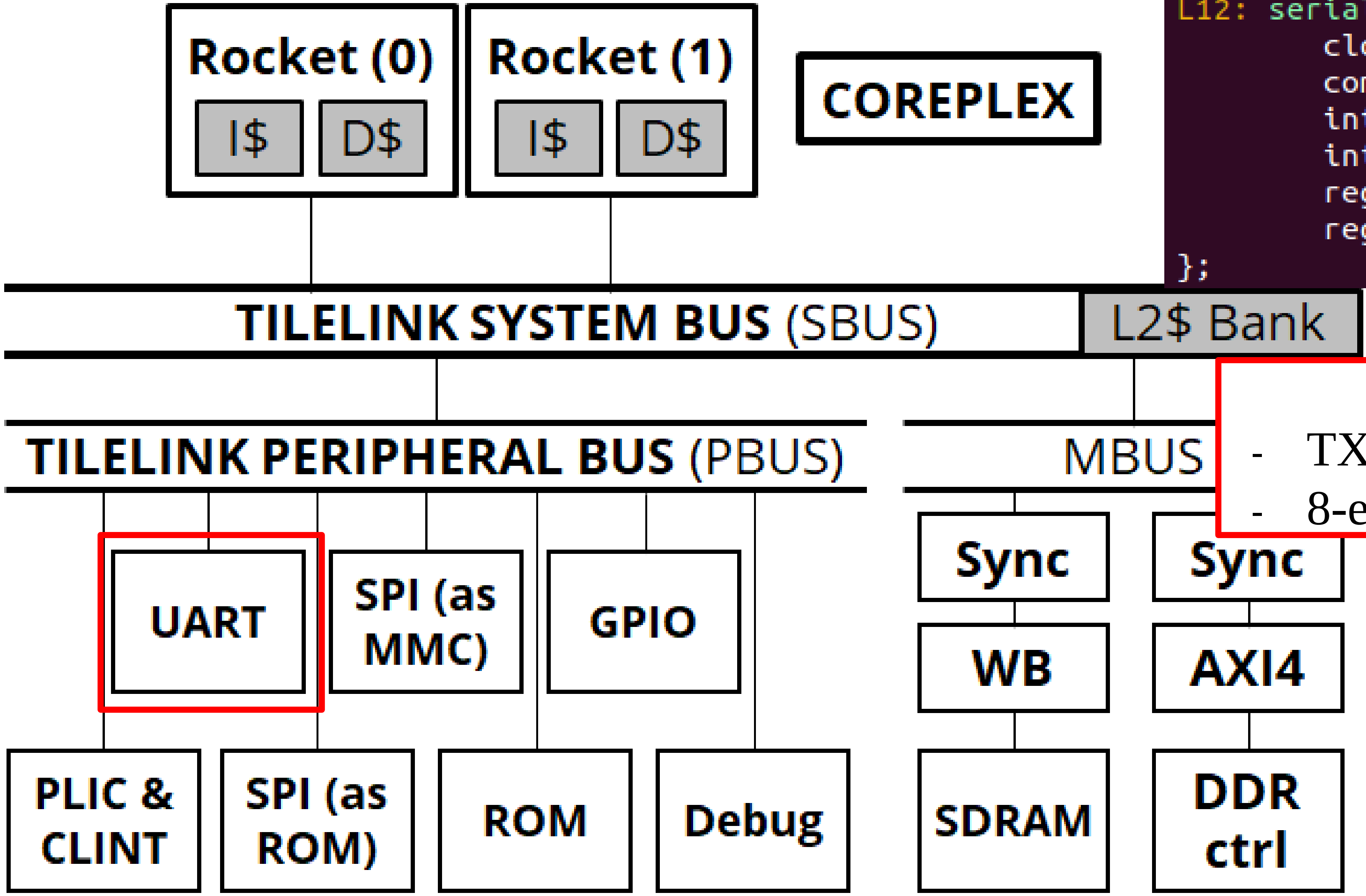


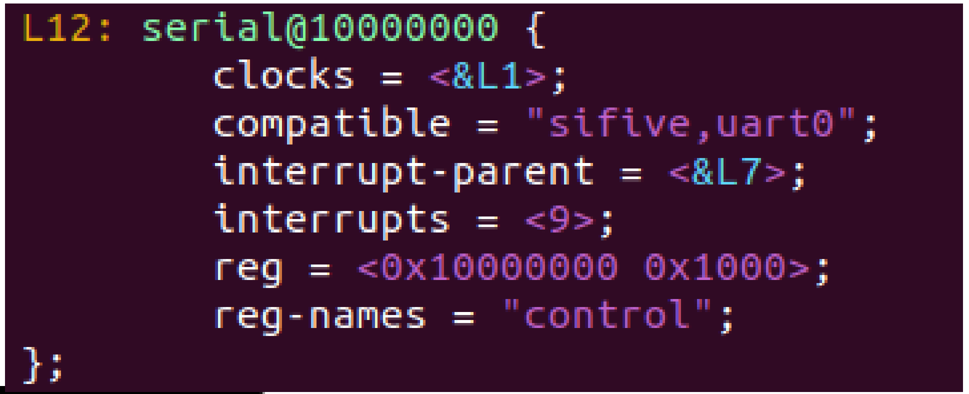
```
L8: clint@2000000 {
        compatible = "riscv,clint0";
        interrupts-extended = <&L4 3 &L4 7>;
        reg = <0x2000000 0x10000>;
        reg-names = "control";
};
```

Interrupt Controllers

- Platform Interrupts (External)
- Core Local Interrupts (Time & Software)

3. Our RISC-V computer system (8/10) UART



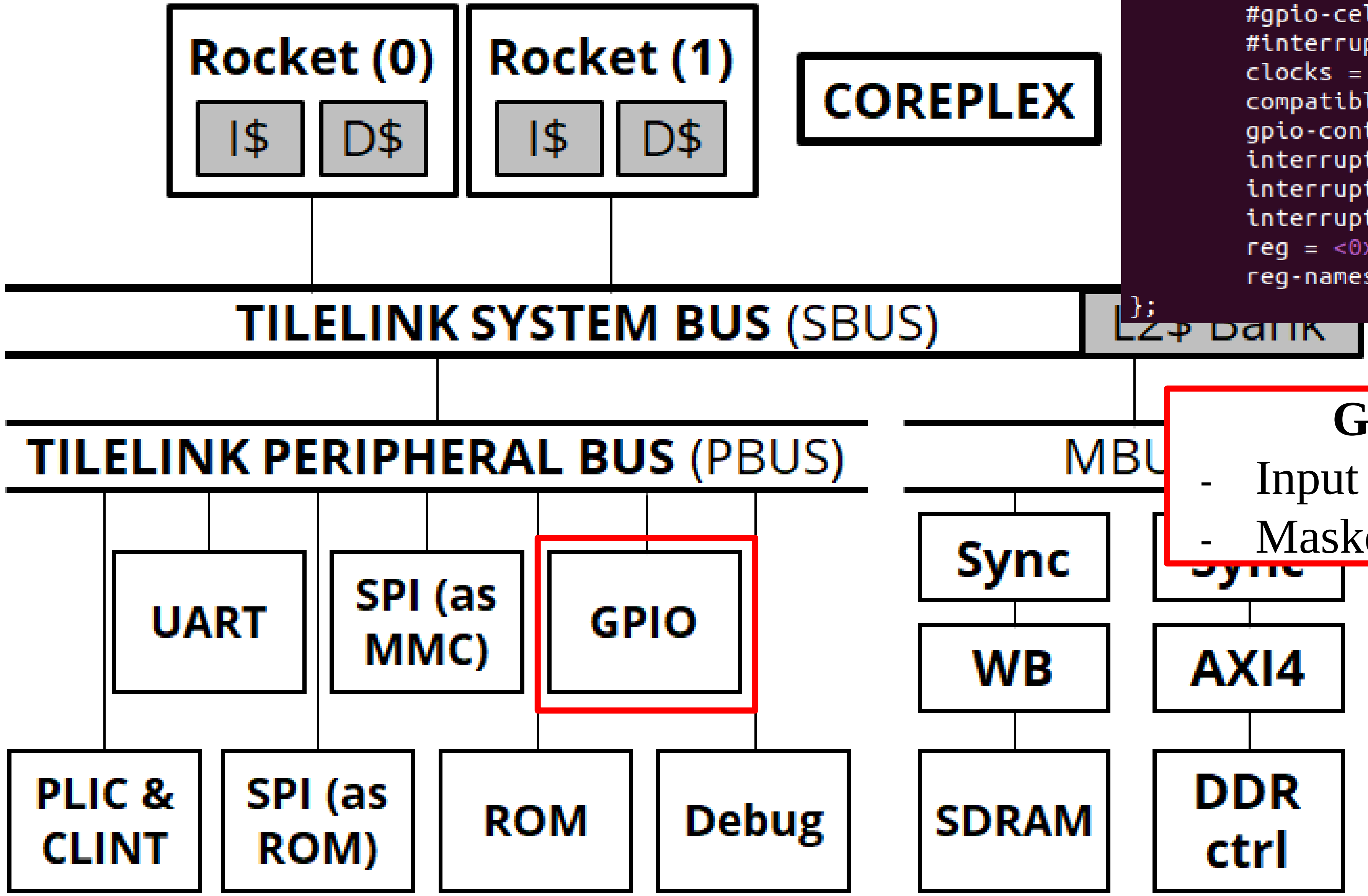


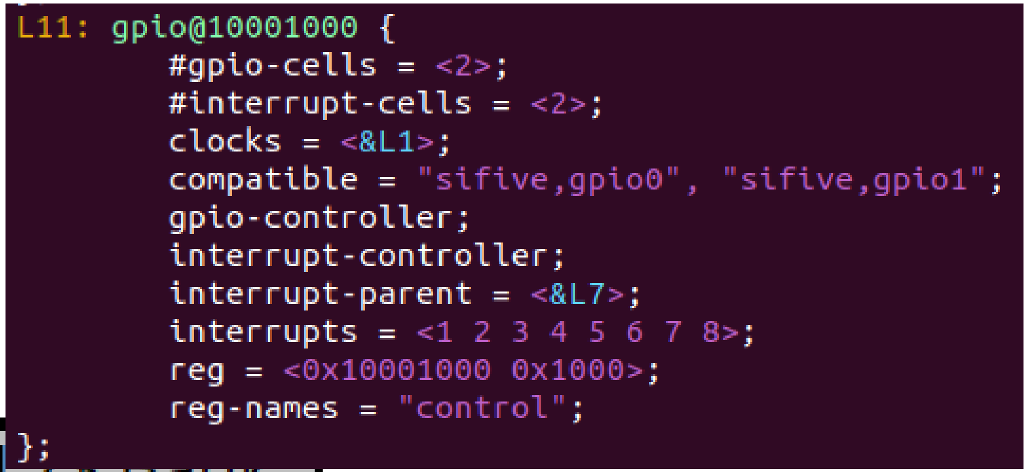
```
L12: serial@10000000 {
        clocks = <&L1>;
        compatible = "sifive,uart0";
        interrupt-parent = <&L7>;
        interrupts = <9>;
        reg = <0x10000000 0x1000>;
        reg-names = "control";
};
```

UART Controller

- TX and RX channels
- 8-entry FIFO with interrupts

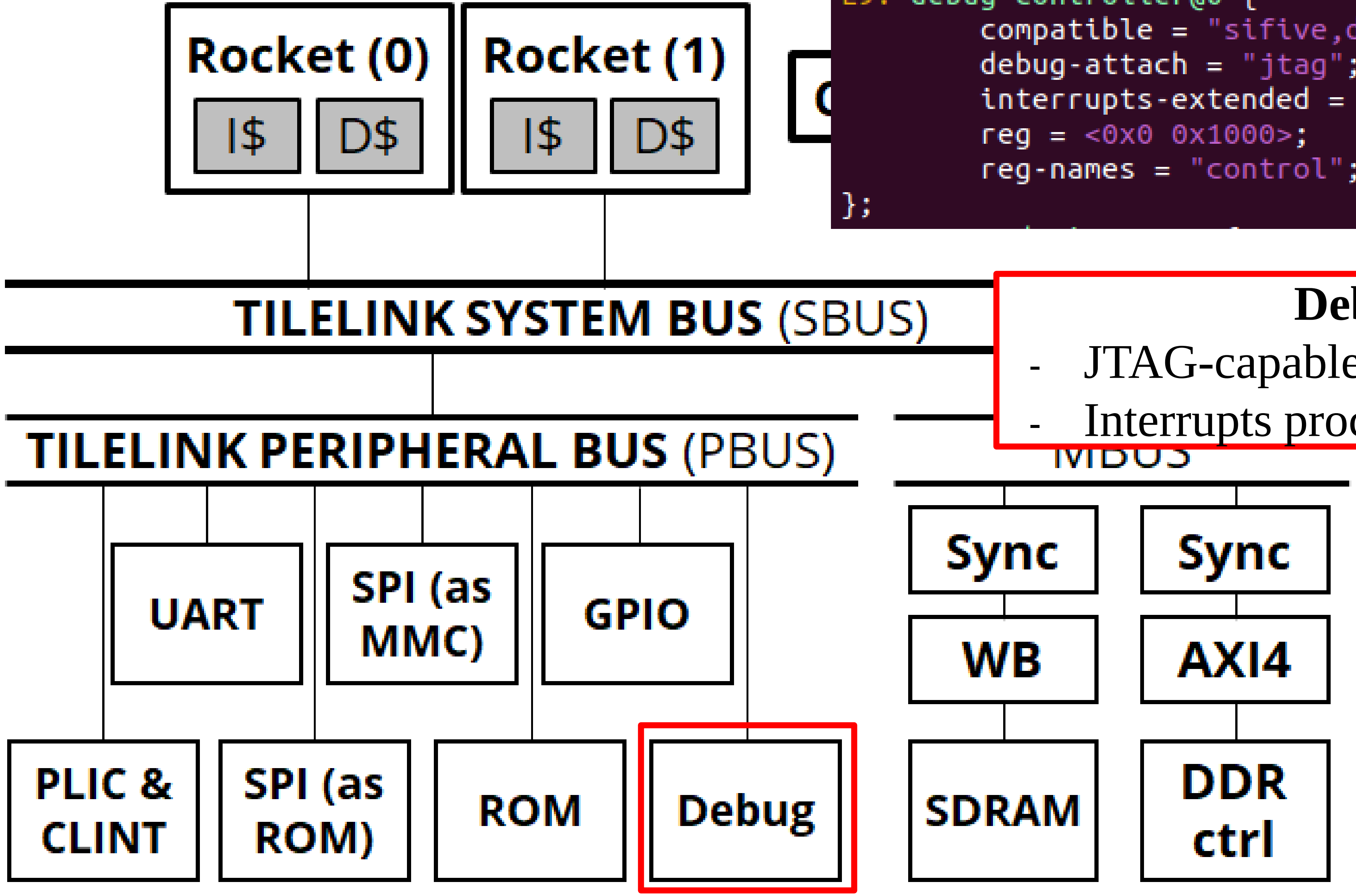
3. Our RISC-V computer system (9/10) GPIO

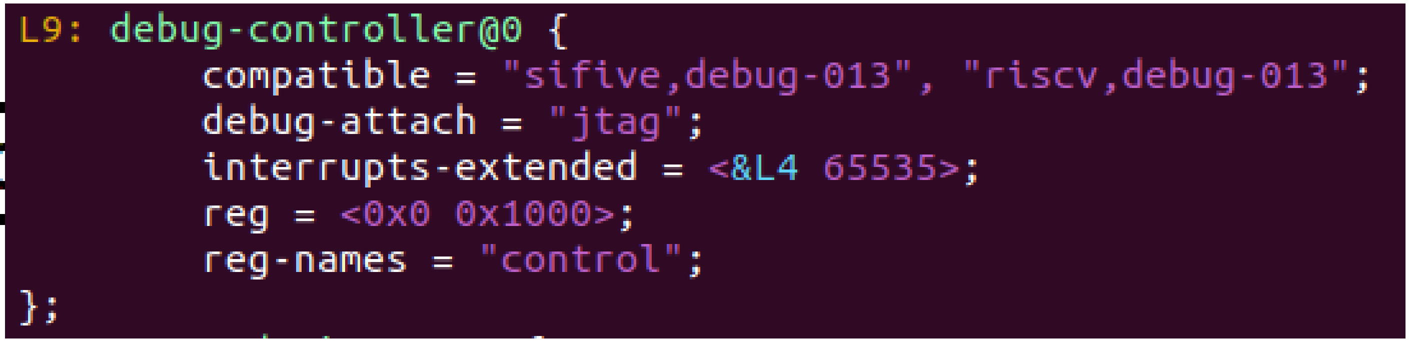



```
L11: gpio@10001000 {
        #gpio-cells = <2>;
        #interrupt-cells = <2>;
        clocks = <&L1>;
        compatible = "sifive,gpio0", "sifive,gpio1";
        gpio-controller;
        interrupt-controller;
        interrupt-parent = <&L7>;
        interrupts = <1 2 3 4 5 6 7 8>;
        reg = <0x10001000 0x1000>;
        reg-names = "control";
};
```

GPIO Controller
- Input and Output registers
- Masked interrupts supported

3. Our RISC-V computer system (10/10) Debug module





```
L9: debug-controller@0 {
        compatible = "sifive,debug-013", "riscv,debug-013";
        debug-attach = "jtag";
        interrupts-extended = <&L4 65535>;
        reg = <0x0 0x1000>;
        reg-names = "control";
};
```

Debug

- JTAG-capable debug module
- Interrupts processor and debug

国立大学法人 電気通信大学 The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

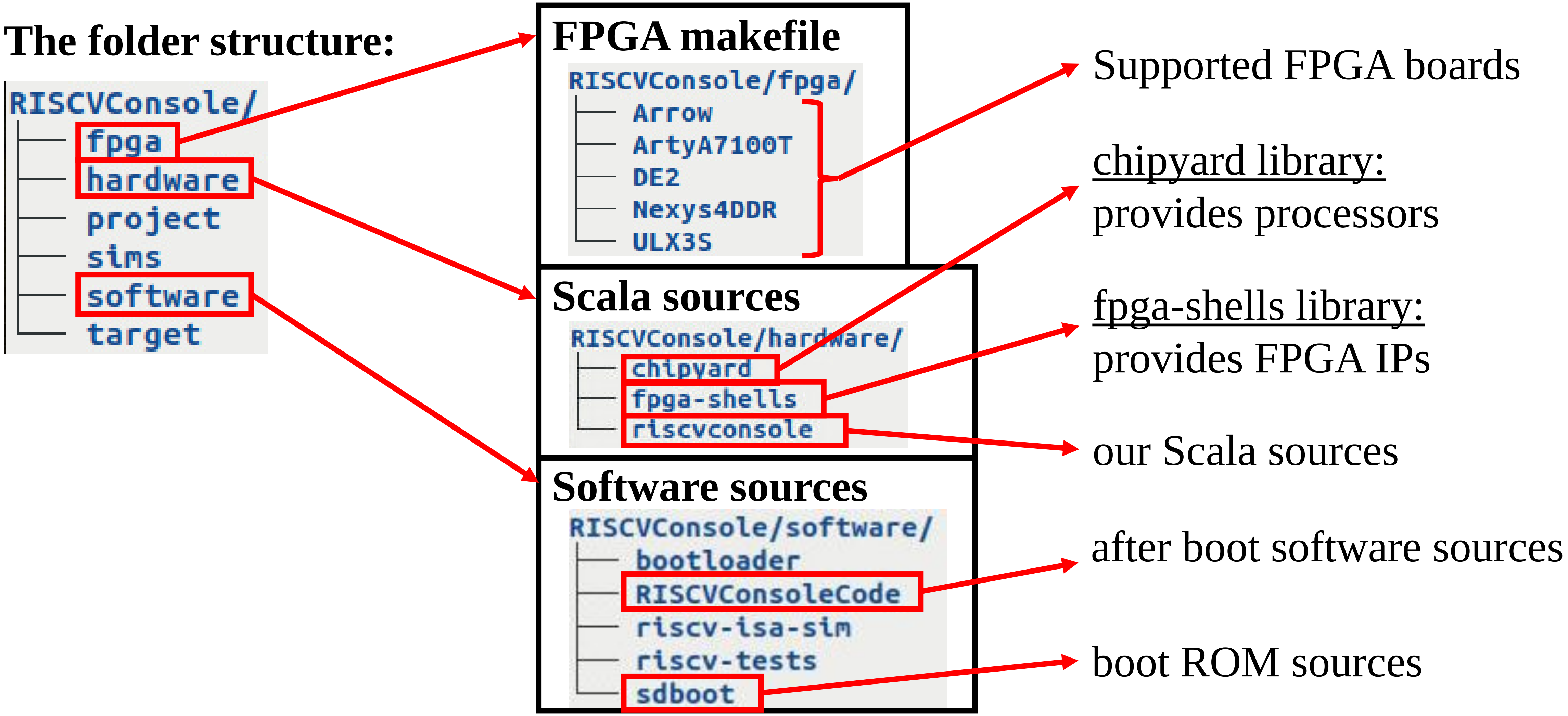
Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

4. **Hardware make flow** (1/10) Folder structure

To clone the project:

```
$ git clone https://github.com/uec-hanken/RISCVConsole.git
$ cd RISCVConsole/
$ ./update.sh
```



The folder structure:

```
RISCVConsole/
├── fpga
├── hardware
├── project
├── sims
├── software
└── target
```

FPGA makefile

```
RISCVConsole/fpga/
├── Arrow
├── ArtyA7100T
├── DE2
├── Nexys4DDR
└── ULX3S
```

Scala sources

```
RISCVConsole/hardware/
├── chipyard
├── fpga-shells
└── riscvconsole
```

Software sources

```
RISCVConsole/software/
├── bootloader
├── RISCVConsoleCode
├── riscv-isa-sim
├── riscv-tests
└── sdboot
```

Supported FPGA boards

chipyard library:
provides processors

fpga-shells library:
provides FPGA IPs

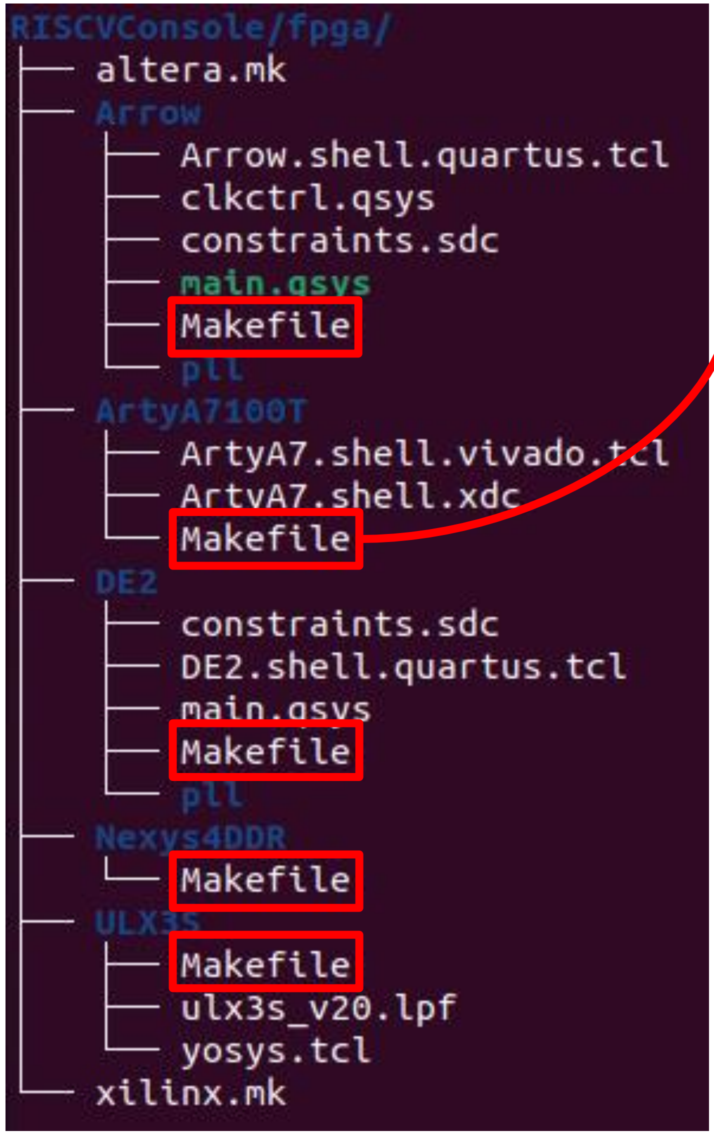
our Scala sources

after boot software sources

boot ROM sources

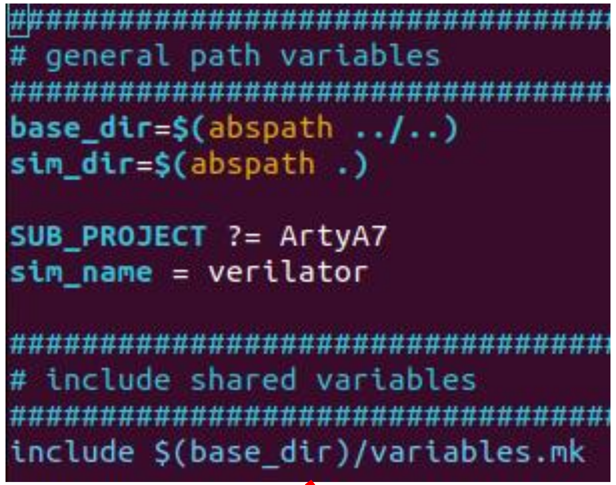
4. Hardware make flow (2/10) Makefile in **fpga** folder

$$\begin{tikzcd}\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT } (\textit{.scala})} \arrow[r] & \boxed{\text{FIRRTL } (\textit{.fir})} \arrow[r] & \boxed{\text{FPGA } (\textit{.v})} \arrow[r] & \boxed{\text{FPGA } (\textit{.bit})}\end{tikzcd}$$



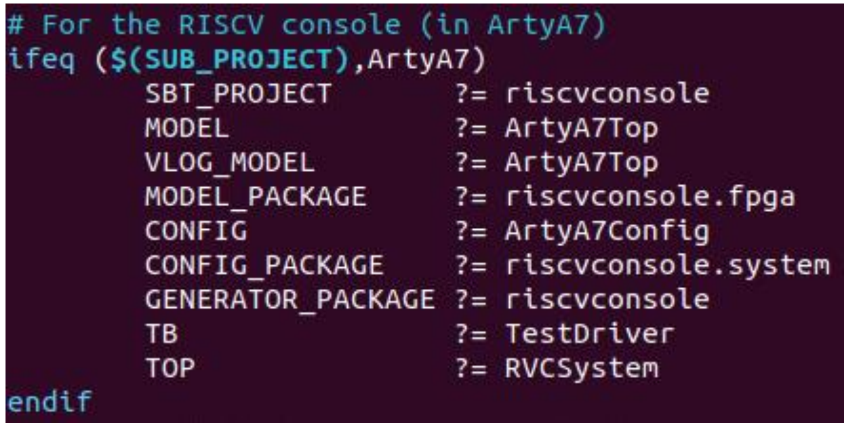
```
RISCVConsole/fpga/
├── altera.mk
├── Arrow
│   ├── Arrow.shell.quartus.tcl
│   ├── clkctrl.qsys
│   ├── constraints.sdc
│   ├── main.qsys
│   ├── Makefile
│   └── pll
├── ArtyA7100T
│   ├── ArtyA7.shell.vivado.tcl
│   ├── ArtyA7.shell.xdc
│   └── Makefile
├── DE2
│   ├── constraints.sdc
│   ├── DE2.shell.quartus.tcl
│   ├── main.qsys
│   ├── Makefile
│   └── pll
├── Nexys4DDR
│   └── Makefile
├── ULX3S
│   ├── Makefile
│   ├── ulx3s_v20.lpf
│   └── yosys.tcl
└── xilinx.mk
```

At `RISCVConsole/fpga/ArtyA7100T/Makefile`:



```
#################################
# general path variables
#################################
base_dir=$(abspath ../..)
sim_dir=$(abspath .)

SUB_PROJECT ?= ArtyA7
sim_name = verilator

#################################
# include shared variables
#################################
include $(base_dir)/variables.mk
```

At `RISCVConsole/variables.mk`:

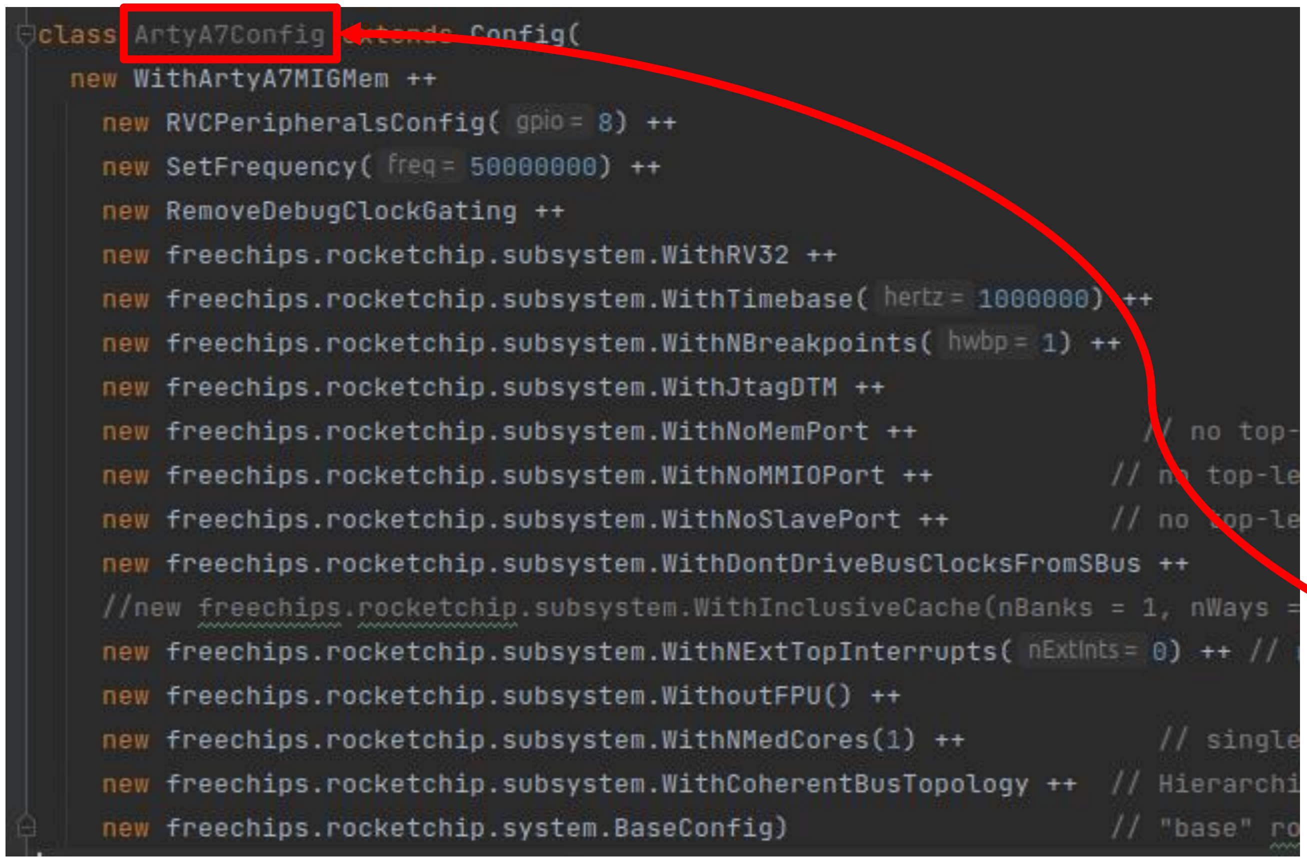


```
# For the RISCV console (in ArtyA7)
ifeq ($(SUB_PROJECT),ArtyA7)
	SBT_PROJECT       ?= riscvconsole
	MODEL             ?= ArtyA7Top
	VLOG_MODEL        ?= ArtyA7Top
	MODEL_PACKAGE     ?= riscvconsole.fpga
	CONFIG            ?= ArtyA7Config
	CONFIG_PACKAGE    ?= riscvconsole.system
	GENERATOR_PACKAGE ?= riscvconsole
	TB                ?= TestDriver
	TOP               ?= RVCSystem
endif
```

4. **Hardware make flow** (3/10) Equivalent in **Scala** config

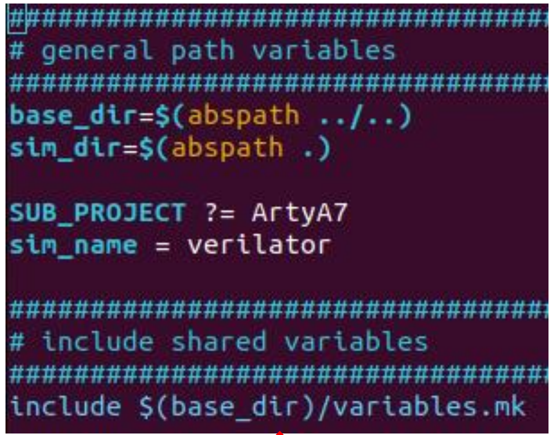
$$\begin{tikzcd}\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT (\textit{.scala})}} \arrow[r] & \boxed{\text{FIRRTL (\textit{.fir})}} \arrow[r] & \boxed{\text{FPGA (\textit{.v})}} \arrow[r] & \boxed{\text{FPGA (\textit{.bit})}}\end{tikzcd}$$

At `RISCVConsole/hardware/riscvconsole/src/main/scala/riscvconsole/RVCConfig.scala`:



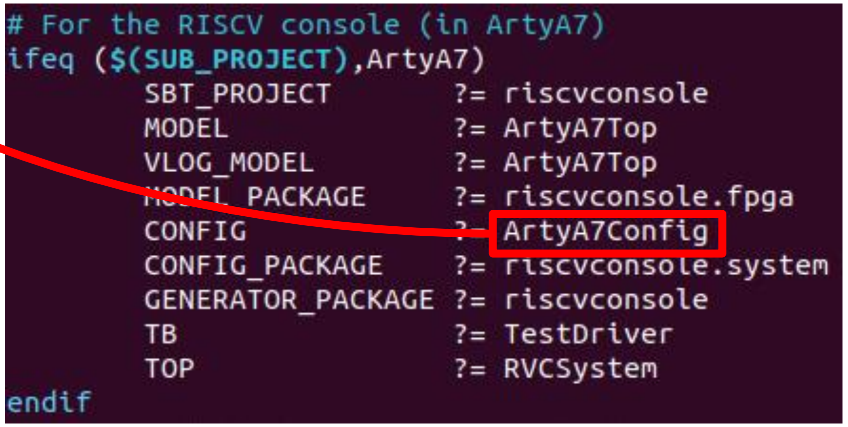
```
class ArtyA7Config extends Config(
  new WithArtyA7MIGMem ++
    new RVCPeripheralsConfig( gpio = 8) ++
    new SetFrequency( freq = 50000000) ++
    new RemoveDebugClockGating ++
    new freechips.rocketchip.subsystem.WithRV32 ++
    new freechips.rocketchip.subsystem.WithTimebase( hertz = 1000000) ++
    new freechips.rocketchip.subsystem.WithNBreakpoints( hwbp = 1) ++
    new freechips.rocketchip.subsystem.WithJtagDTM ++
    new freechips.rocketchip.subsystem.WithNoMemPort ++                // no top-
    new freechips.rocketchip.subsystem.WithNoMMIOPort ++              // no top-le
    new freechips.rocketchip.subsystem.WithNoSlavePort ++             // no top-le
    new freechips.rocketchip.subsystem.WithDontDriveBusClocksFromSBus ++
    //new freechips.rocketchip.subsystem.WithInclusiveCache(nBanks = 1, nWays =
    new freechips.rocketchip.subsystem.WithNExtTopInterrupts( nExtInts = 0) ++ //
    new freechips.rocketchip.subsystem.WithoutFPU() ++
    new freechips.rocketchip.subsystem.WithNMedCores(1) ++             // single
    new freechips.rocketchip.subsystem.WithCoherentBusTopology ++  // Hierarchi
    new freechips.rocketchip.system.BaseConfig)                     // "base" ro
```

At `RISCVConsole/fpga/ArtyA7100T/Makefile`:



```
##################################
# general path variables
##################################
base_dir=$(abspath ../..)
sim_dir=$(abspath .)

SUB_PROJECT ?= ArtyA7
sim_name = verilator

##################################
# include shared variables
##################################
include $(base_dir)/variables.mk
```

At `RISCVConsole/variables.mk`:



```
# For the RISCV console (in ArtyA7)
ifeq ($(SUB_PROJECT),ArtyA7)
	SBT_PROJECT       ?= riscvconsole
	MODEL             ?= ArtyA7Top
	VLOG_MODEL        ?= ArtyA7Top
	MODEL_PACKAGE     ?= riscvconsole.fpga
	CONFIG            ?= ArtyA7Config
	CONFIG_PACKAGE    ?= riscvconsole.system
	GENERATOR_PACKAGE ?= riscvconsole
	TB                ?= TestDriver
	TOP               ?= RVCSystem
endif
```

4. Hardware make flow (4/10) **Scala** structure

$$\begin{tikzcd}\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT } \textit{(.scala)}} \arrow[r] & \boxed{\text{FIRRTL } \textit{(.fir)}} \arrow[r] & \boxed{\text{FPGA } \textit{(.v)}} \arrow[r] & \boxed{\text{FPGA } \textit{(.bit)}}\end{tikzcd}$$

FPGA Shell folder

GPIO Pins
UART Pins
SPI Pins
I2C Pins
DDR Ports
SDRAM Ports
CODEC Ports
JTAG Pins
Other Ports

FPGA folder

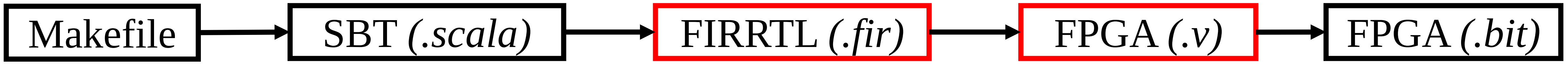
RVC. System

- General Purpose IO
- UART
- SPI Flash
- I2C
- SDRAM/DDR
- TL Serial (For simulations)
- FFT/CODEC
- Any additional peripherals

RVC. Subsystem

- TileLink Buses
- Debug Module
- Boot ROM
- Core Local Interrupts
- Platform Level Interrupt Controller
- Coreplex
  - Rocket
  - BOOM

4. Hardware make flow (5/10) `$ make default`



Now, to make the system, from the RISCVConsole, go to the Arty build folder:

```
$ cd fpga/ArtyA7100T/
```

Export the RISC-V toolchain to the **PATH**:

```
$ export RISCV=/opt/riscv
$ export PATH=$RISCV/bin/:$PATH
```

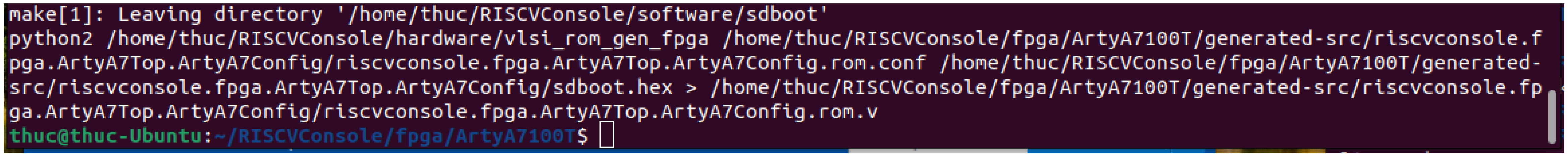
Export *vivado* to the **PATH**:

```
$ export PATH=/opt/xilinx/Vivado/2021.1/bin/:$PATH
```

For the compilation:

```
$ make default
```

This will compile Scala to Verilog *(also compile the boot ROM C/C++ sources)*

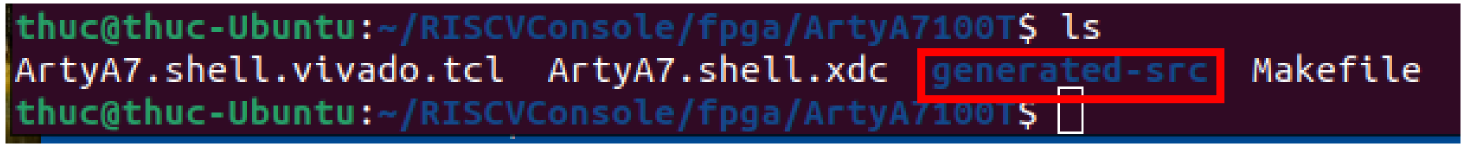


```
make[1]: Leaving directory '/home/thuc/RISCVConsole/software/sdboot'
python2 /home/thuc/RISCVConsole/hardware/vlsi_rom_gen_fpga /home/thuc/RISCVConsole/fpga/ArtyA7100T/generated-src/riscvconsole.f
pga.ArtyA7Top.ArtyA7Config/riscvconsole.fpga.ArtyA7Top.ArtyA7Config.rom.conf /home/thuc/RISCVConsole/fpga/ArtyA7100T/generated-
src/riscvconsole.fpga.ArtyA7Top.ArtyA7Config/sdboot.hex > /home/thuc/RISCVConsole/fpga/ArtyA7100T/generated-src/riscvconsole.fp
ga.ArtyA7Top.ArtyA7Config/riscvconsole.fpga.ArtyA7Top.ArtyA7Config.rom.v
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ 
```

4. Hardware make flow (6/10) Generated files

$$\begin{tikzcd}
\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT } (\textit{.scala})} \arrow[r] & \boxed{\text{FIRRTL } (\textit{.fir})} \arrow[r] & \boxed{\text{FPGA } (\textit{.v})} \arrow[r] & \boxed{\text{FPGA } (\textit{.bit})}
\end{tikzcd}$$

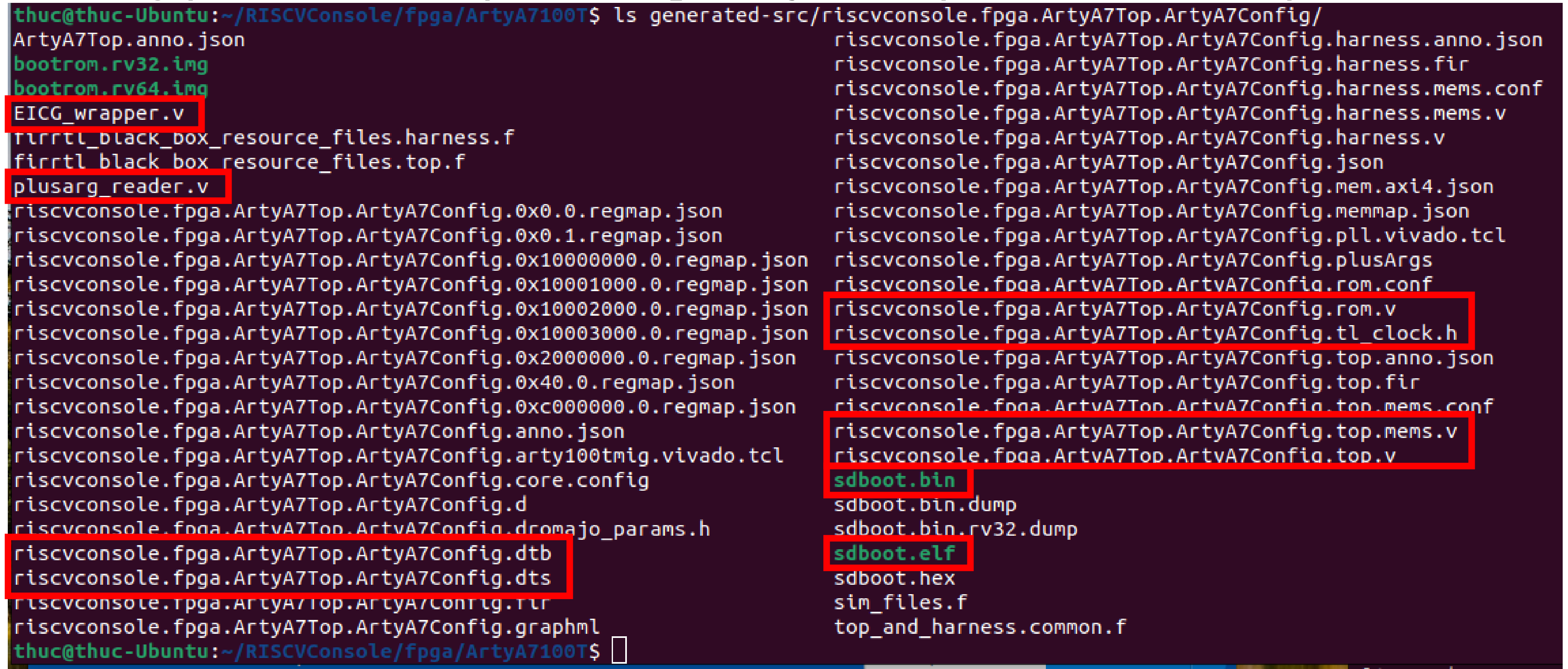
After `$ make default`, the **generated-src** folder is created:



```
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ ls
ArtyA7.shell.vivado.tcl  ArtyA7.shell.xdc  generated-src  Makefile
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ 
```

Inside the **generated-src** folder, there are many files:

Verilog files, FIRRTL files, temporary Java files, boot ROM files, device tree, etc.

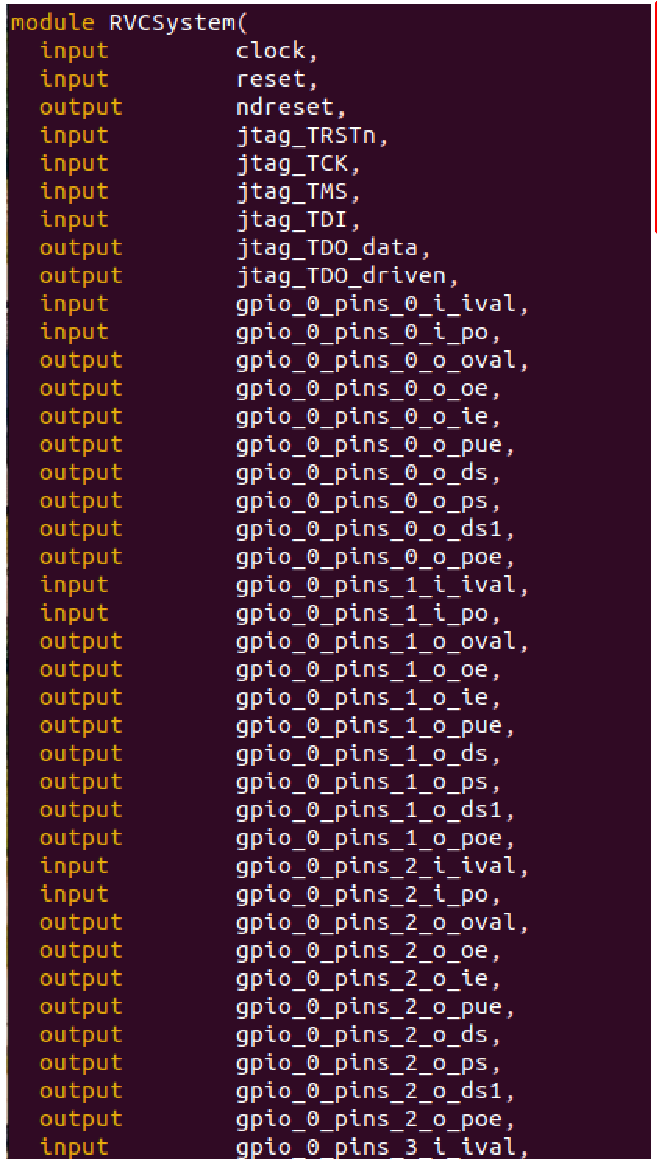


```
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ ls generated-src/riscvconsole.fpga.ArtyA7Top.ArtyA7Config/
ArtyA7Top.anno.json                                            riscvconsole.fpga.ArtyA7Top.ArtyA7Config.harness.anno.json
bootrom.rv32.img                                               riscvconsole.fpga.ArtyA7Top.ArtyA7Config.harness.fir
bootrom.rv64.img                                               riscvconsole.fpga.ArtyA7Top.ArtyA7Config.harness.mems.conf
EICG_wrapper.v                                                 riscvconsole.fpga.ArtyA7Top.ArtyA7Config.harness.mems.v
firrtl_black_box_resource_files.harness.f                      riscvconsole.fpga.ArtyA7Top.ArtyA7Config.harness.v
firrtl_black_box_resource_files.top.f                          riscvconsole.fpga.ArtyA7Top.ArtyA7Config.json
plusarg_reader.v                                               riscvconsole.fpga.ArtyA7Top.ArtyA7Config.mem.axi4.json
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x0.0.regmap.json     riscvconsole.fpga.ArtyA7Top.ArtyA7Config.memmap.json
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x0.1.regmap.json     riscvconsole.fpga.ArtyA7Top.ArtyA7Config.pll.vivado.tcl
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x10000000.0.regmap.json  riscvconsole.fpga.ArtyA7Top.ArtyA7Config.plusArgs
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x10001000.0.regmap.json  riscvconsole.fpga.ArtyA7Top.ArtyA7Config.rom.conf
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x10002000.0.regmap.json  riscvconsole.fpga.ArtyA7Top.ArtyA7Config.rom.v
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x10003000.0.regmap.json  riscvconsole.fpga.ArtyA7Top.ArtyA7Config.tl_clock.h
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x2000000.0.regmap.json   riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.anno.json
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0x40.0.regmap.json    riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.fir
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.0xc000000.0.regmap.json   riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.mems.conf
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.anno.json             riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.mems.v
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.arty100tmig.vivado.tcl    riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.v
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.core.config           sdboot.bin
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.d                     sdboot.bin.dump
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.dromajo_params.h      sdboot.bin.rv32.dump
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.dtb                   sdboot.elf
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.dts                   sdboot.hex
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.fir                   sim_files.f
riscvconsole.fpga.ArtyA7Top.ArtyA7Config.graphml               top_and_harness.common.f
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ 
```

Some important files

4. Hardware make flow (7/10) Generated Verilog

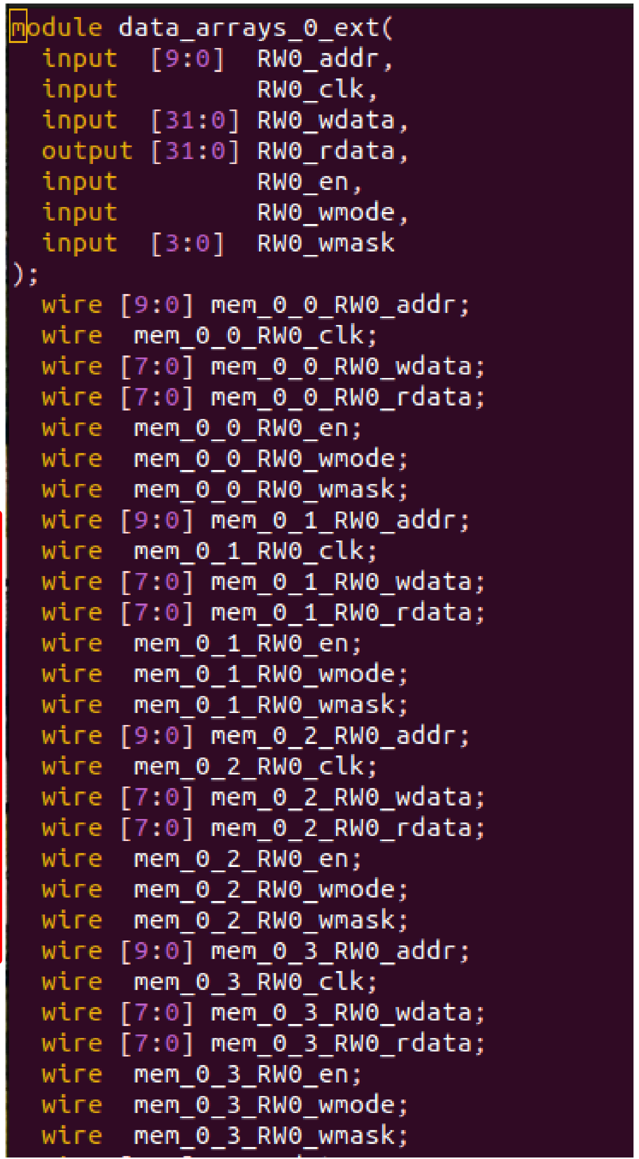
$$\begin{tikzcd}
\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT (\textit{.scala})}} \arrow[r] & \boxed{\text{FIRRTL (\textit{.fir})}} \arrow[r] & \boxed{\text{FPGA (\textit{.v})}} \arrow[r] & \boxed{\text{FPGA (\textit{.bit})}}
\end{tikzcd}$$



```
module RVCSystem(
  input         clock,
  input         reset,
  output        ndreset,
  input         jtag_TRSTn,
  input         jtag_TCK,
  input         jtag_TMS,
  input         jtag_TDI,
  output        jtag_TDO_data,
  output        jtag_TDO_driven,
  input         gpio_0_pins_0_i_ival,
  input         gpio_0_pins_0_i_po,
  output        gpio_0_pins_0_o_oval,
  output        gpio_0_pins_0_o_oe,
  output        gpio_0_pins_0_o_ie,
  output        gpio_0_pins_0_o_pue,
  output        gpio_0_pins_0_o_ds,
  output        gpio_0_pins_0_o_ps,
  output        gpio_0_pins_0_o_ds1,
  output        gpio_0_pins_0_o_poe,
  input         gpio_0_pins_1_i_ival,
  input         gpio_0_pins_1_i_po,
  output        gpio_0_pins_1_o_oval,
  output        gpio_0_pins_1_o_oe,
  output        gpio_0_pins_1_o_ie,
  output        gpio_0_pins_1_o_pue,
  output        gpio_0_pins_1_o_ds,
  output        gpio_0_pins_1_o_ps,
  output        gpio_0_pins_1_o_ds1,
  output        gpio_0_pins_1_o_poe,
  input         gpio_0_pins_2_i_ival,
  input         gpio_0_pins_2_i_po,
  output        gpio_0_pins_2_o_oval,
  output        gpio_0_pins_2_o_oe,
  output        gpio_0_pins_2_o_ie,
  output        gpio_0_pins_2_o_pue,
  output        gpio_0_pins_2_o_ds,
  output        gpio_0_pins_2_o_ps,
  output        gpio_0_pins_2_o_ds1,
  output        gpio_0_pins_2_o_poe,
  input         gpio_0_pins_3_i_ival,
```

Top file:
`riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.v`

File that contains all the memories used in the system:
`riscvconsole.fpga.ArtyA7Top.ArtyA7Config.top.mems.v`

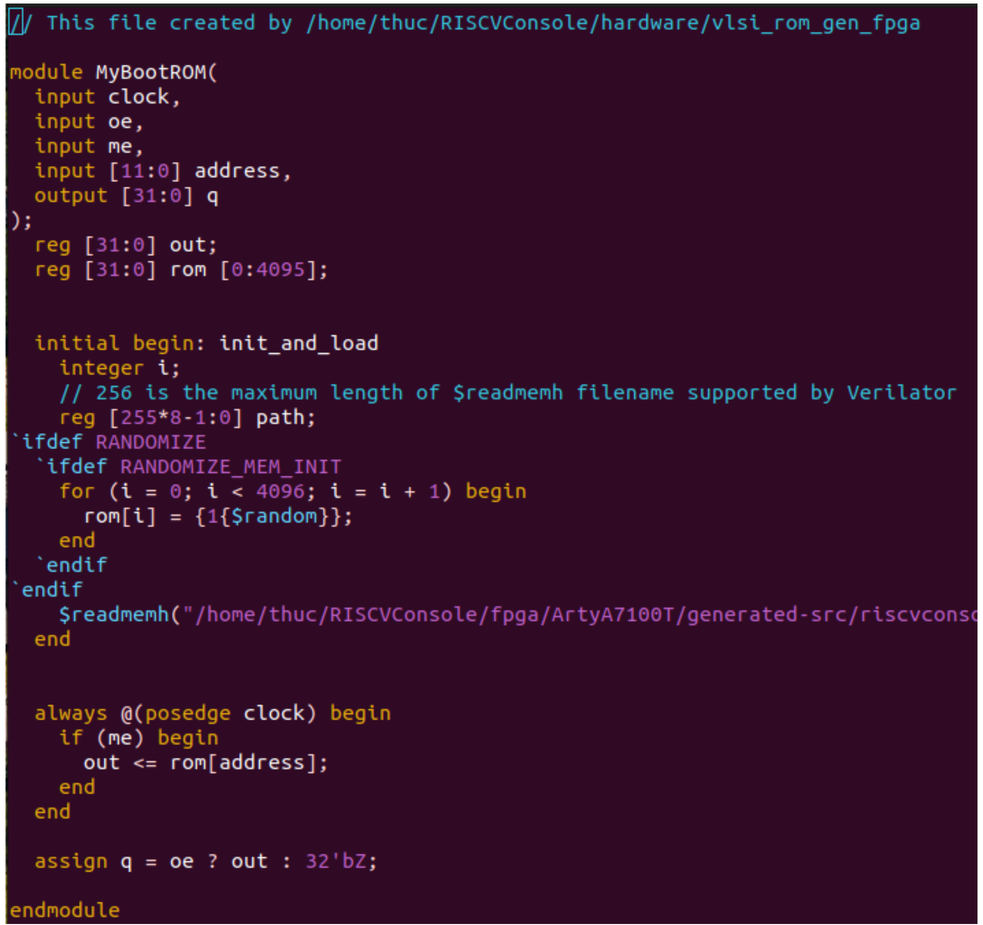


```
module data_arrays_0_ext(
  input  [9:0]  RW0_addr,
  input         RW0_clk,
  input  [31:0] RW0_wdata,
  output [31:0] RW0_rdata,
  input         RW0_en,
  input         RW0_wmode,
  input  [3:0]  RW0_wmask
);
  wire [9:0] mem_0_0_RW0_addr;
  wire  mem_0_0_RW0_clk;
  wire [7:0] mem_0_0_RW0_wdata;
  wire [7:0] mem_0_0_RW0_rdata;
  wire  mem_0_0_RW0_en;
  wire  mem_0_0_RW0_wmode;
  wire  mem_0_0_RW0_wmask;
  wire [9:0] mem_0_1_RW0_addr;
  wire  mem_0_1_RW0_clk;
  wire [7:0] mem_0_1_RW0_wdata;
  wire [7:0] mem_0_1_RW0_rdata;
  wire  mem_0_1_RW0_en;
  wire  mem_0_1_RW0_wmode;
  wire  mem_0_1_RW0_wmask;
  wire [9:0] mem_0_2_RW0_addr;
  wire  mem_0_2_RW0_clk;
  wire [7:0] mem_0_2_RW0_wdata;
  wire [7:0] mem_0_2_RW0_rdata;
  wire  mem_0_2_RW0_en;
  wire  mem_0_2_RW0_wmode;
  wire  mem_0_2_RW0_wmask;
  wire [9:0] mem_0_3_RW0_addr;
  wire  mem_0_3_RW0_clk;
  wire [7:0] mem_0_3_RW0_wdata;
  wire [7:0] mem_0_3_RW0_rdata;
  wire  mem_0_3_RW0_en;
  wire  mem_0_3_RW0_wmode;
  wire  mem_0_3_RW0_wmask;
```

4. Hardware make flow (8/10) Generated Verilog

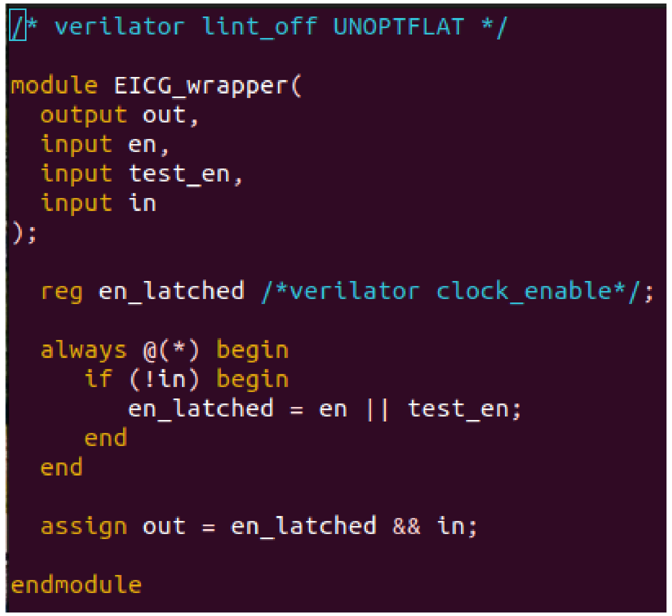
$$\begin{tikzcd}\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT } (\textit{.scala})} \arrow[r] & \boxed{\text{FIRRTL } (\textit{.fir})} \arrow[r] & \boxed{\text{FPGA } (\textit{.v})} \arrow[r] & \boxed{\text{FPGA } (\textit{.bit})}\end{tikzcd}$$

Boot ROM file:
`riscvconsole.fpga.ArtyA7Top.ArtyA7Config.rom.v`

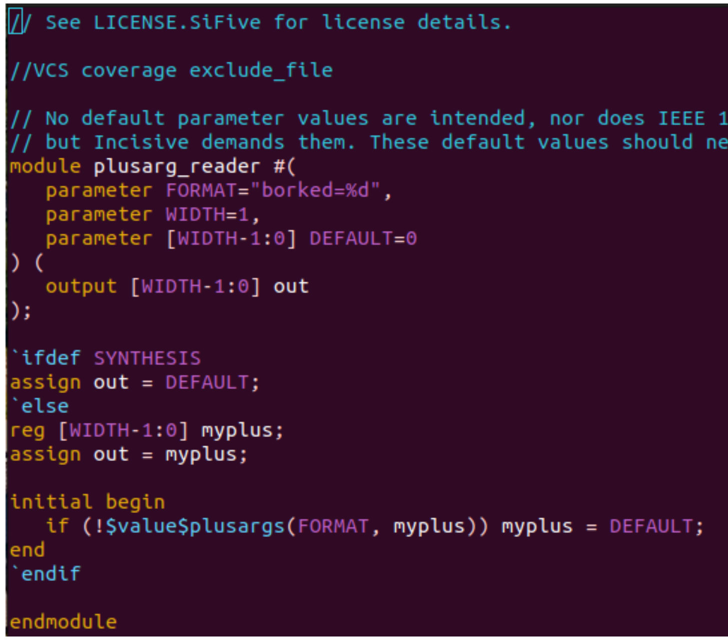


```verilog
// This file created by /home/thuc/RISCVConsole/hardware/vlsi_rom_gen_fpga

module MyBootROM(
  input clock,
  input oe,
  input me,
  input [11:0] address,
  output [31:0] q
);
  reg [31:0] out;
  reg [31:0] rom [0:4095];


  initial begin: init_and_load
    integer i;
    // 256 is the maximum length of $readmemh filename supported by Verilator
    reg [255*8-1:0] path;
`ifdef RANDOMIZE
  `ifdef RANDOMIZE_MEM_INIT
    for (i = 0; i < 4096; i = i + 1) begin
      rom[i] = {1{$random}};
    end
  `endif
`endif
    $readmemh("/home/thuc/RISCVConsole/fpga/ArtyA7100T/generated-src/riscvconso
  end


  always @(posedge clock) begin
    if (me) begin
      out <= rom[address];
    end
  end

  assign q = oe ? out : 32'bZ;

endmodule
```

Other Verilog files:
- `EICG_wrapper.v`
- `plusarg_reader.v`



```verilog
/* verilator lint_off UNOPTFLAT */

module EICG_wrapper(
  output out,
  input en,
  input test_en,
  input in
);

  reg en_latched /*verilator clock_enable*/;

  always @(*) begin
     if (!in) begin
        en_latched = en || test_en;
     end
  end

  assign out = en_latched && in;

endmodule
```



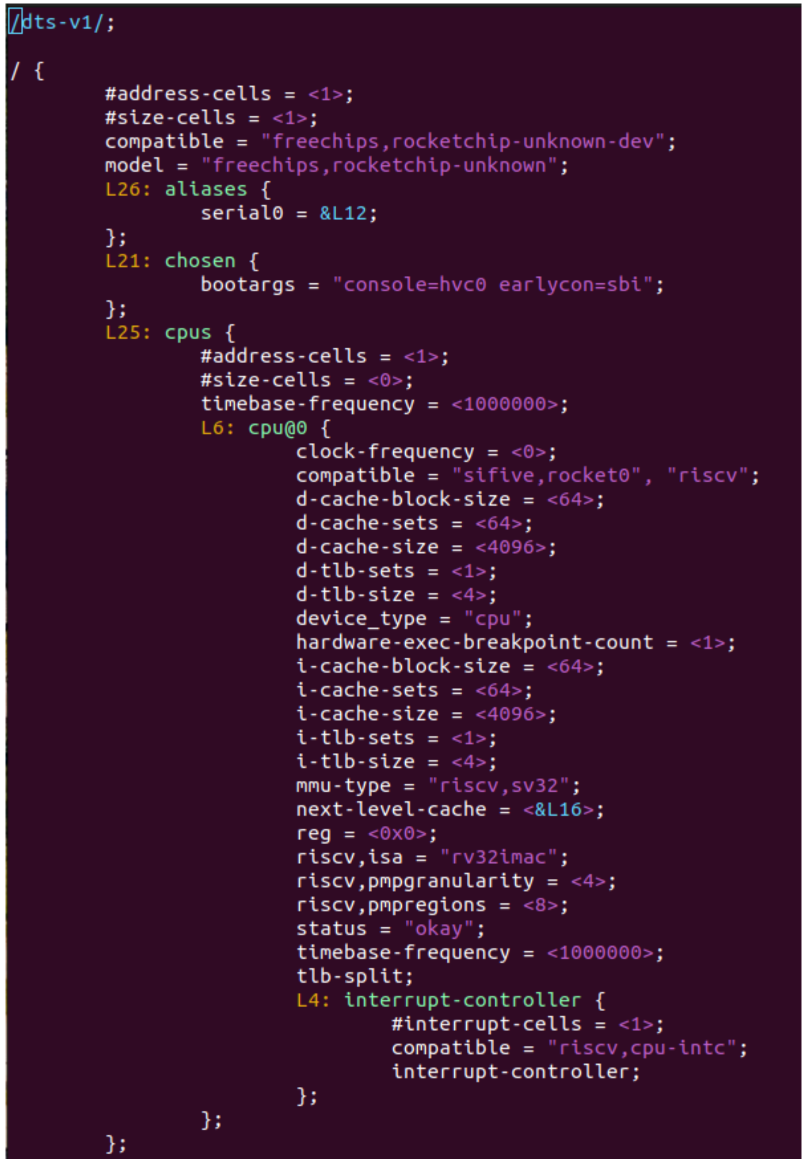
```verilog
// See LICENSE.SiFive for license details.

//VCS coverage exclude_file

// No default parameter values are intended, nor does IEEE 1
// but Incisive demands them. These default values should ne
module plusarg_reader #(
   parameter FORMAT="borked=%d",
   parameter WIDTH=1,
   parameter [WIDTH-1:0] DEFAULT=0
) (
   output [WIDTH-1:0] out
);

`ifdef SYNTHESIS
assign out = DEFAULT;
`else
reg [WIDTH-1:0] myplus;
assign out = myplus;

initial begin
   if (!$value$plusargs(FORMAT, myplus)) myplus = DEFAULT;
end
`endif

endmodule
```

4. Hardware make flow (9/10) Generated device tree

$$\begin{tikzcd}\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT } (\textit{.scala})} \arrow[r] & \boxed{\text{FIRRTL } (\textit{.fir})} \arrow[r] & \boxed{\text{FPGA } (\textit{.v})} \arrow[r] & \boxed{\text{FPGA } (\textit{.bit})}\end{tikzcd}$$

Device tree file:
(needed for software)
`riscvconsole.fpga.ArtyA7Top.ArtyA7Config.dts`

Its binary version:
`riscvconsole.fpga.ArtyA7Top.ArtyA7Config.dtb`



```
/dts-v1/;

/ {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "freechips,rocketchip-unknown-dev";
        model = "freechips,rocketchip-unknown";
        L26: aliases {
                serial0 = &L12;
        };
        L21: chosen {
                bootargs = "console=hvc0 earlycon=sbi";
        };
        L25: cpus {
                #address-cells = <1>;
                #size-cells = <0>;
                timebase-frequency = <1000000>;
                L6: cpu@0 {
                        clock-frequency = <0>;
                        compatible = "sifive,rocket0", "riscv";
                        d-cache-block-size = <64>;
                        d-cache-sets = <64>;
                        d-cache-size = <4096>;
                        d-tlb-sets = <1>;
                        d-tlb-size = <4>;
                        device_type = "cpu";
                        hardware-exec-breakpoint-count = <1>;
                        i-cache-block-size = <64>;
                        i-cache-sets = <64>;
                        i-cache-size = <4096>;
                        i-tlb-sets = <1>;
                        i-tlb-size = <4>;
                        mmu-type = "riscv,sv32";
                        next-level-cache = <&L16>;
                        reg = <0x0>;
                        riscv,isa = "rv32imac";
                        riscv,pmpgranularity = <4>;
                        riscv,pmpregions = <8>;
                        status = "okay";
                        timebase-frequency = <1000000>;
                        tlb-split;
                        L4: interrupt-controller {
                                #interrupt-cells = <1>;
                                compatible = "riscv,cpu-intc";
                                interrupt-controller;
                        };
                };
        };
```

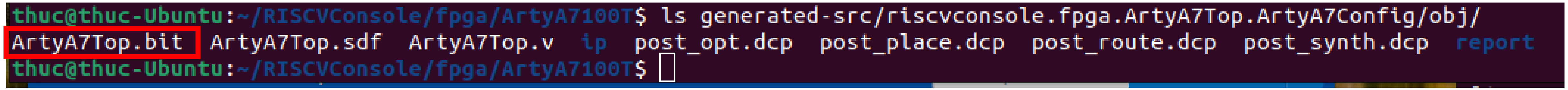
4. Hardware make flow (10/10) `$ make bit`

$$\begin{tikzcd}
\boxed{\text{Makefile}} \arrow[r] & \boxed{\text{SBT } (\textit{.scala})} \arrow[r] & \boxed{\text{FIRRTL } (\textit{.fir})} \arrow[r] & \boxed{\text{FPGA } (\textit{.v})} \arrow[r] & \boxed{\text{FPGA } (\textit{.bit})}
\end{tikzcd}$$

The `$ make default` is just for generating Verilog.
Now, to compile the FPGA:

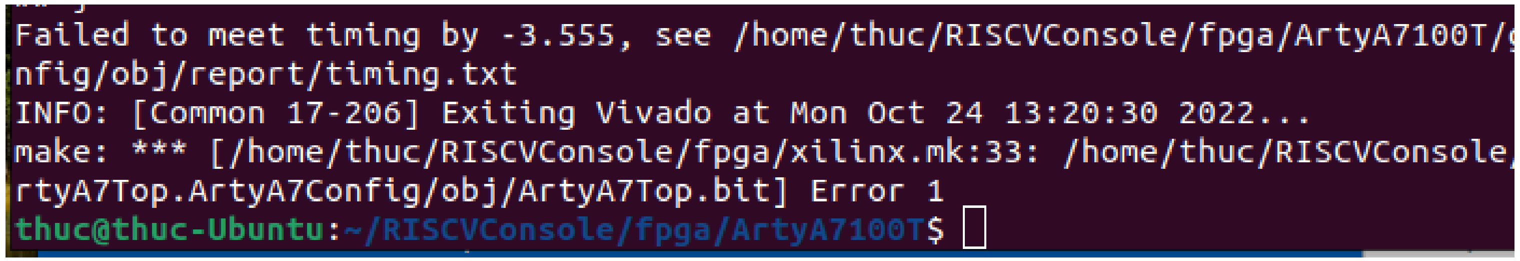
```
$ make bit
```
This will compile Verilog to **.bit** file for programming the FPGA

After `$ make bit`, you can find the **.bit** file for programming the FPGA in:
`generated-src/riscvconsole.fpga.ArtyA7Top.ArtyA7Config/obj/`



```
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ ls generated-src/riscvconsole.fpga.ArtyA7Top.ArtyA7Config/obj/
ArtyA7Top.bit  ArtyA7Top.sdf  ArtyA7Top.v  ip  post_opt.dcp  post_place.dcp  post_route.dcp  post_synth.dcp  report
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ 
```

****Note:*** if the `$ make bit` has an error related to timing, it is fine as long as the **.bit** file was generated.



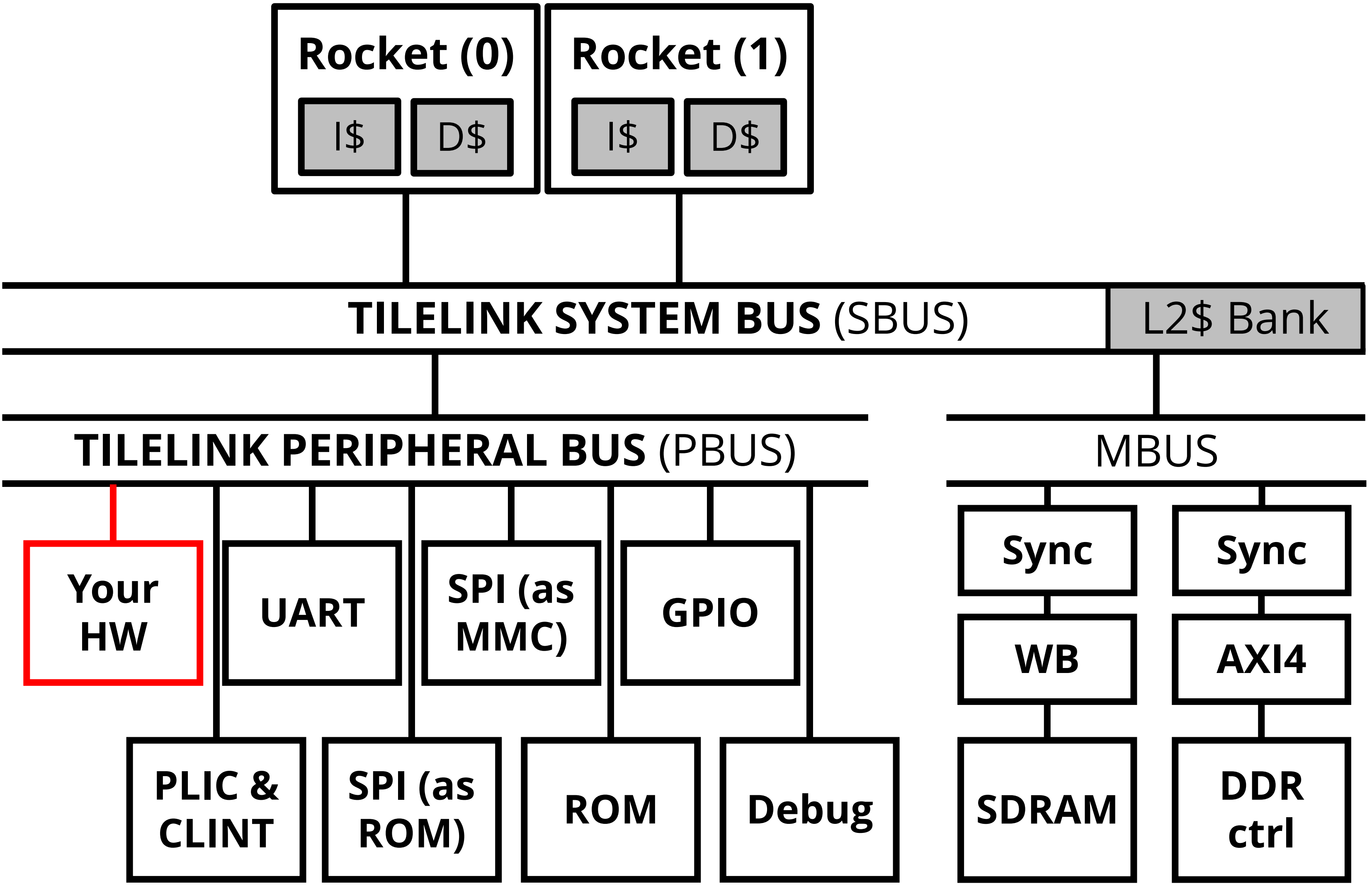
```
Failed to meet timing by -3.555, see /home/thuc/RISCVConsole/fpga/ArtyA7100T/g
nfig/obj/report/timing.txt
INFO: [Common 17-206] Exiting Vivado at Mon Oct 24 13:20:30 2022...
make: *** [/home/thuc/RISCVConsole/fpga/xilinx.mk:33: /home/thuc/RISCVConsole/
rtyA7Top.ArtyA7Config/obj/ArtyA7Top.bit] Error 1
thuc@thuc-Ubuntu:~/RISCVConsole/fpga/ArtyA7100T$ 
```

Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

5. Add custom hardware (1/10) Final goal

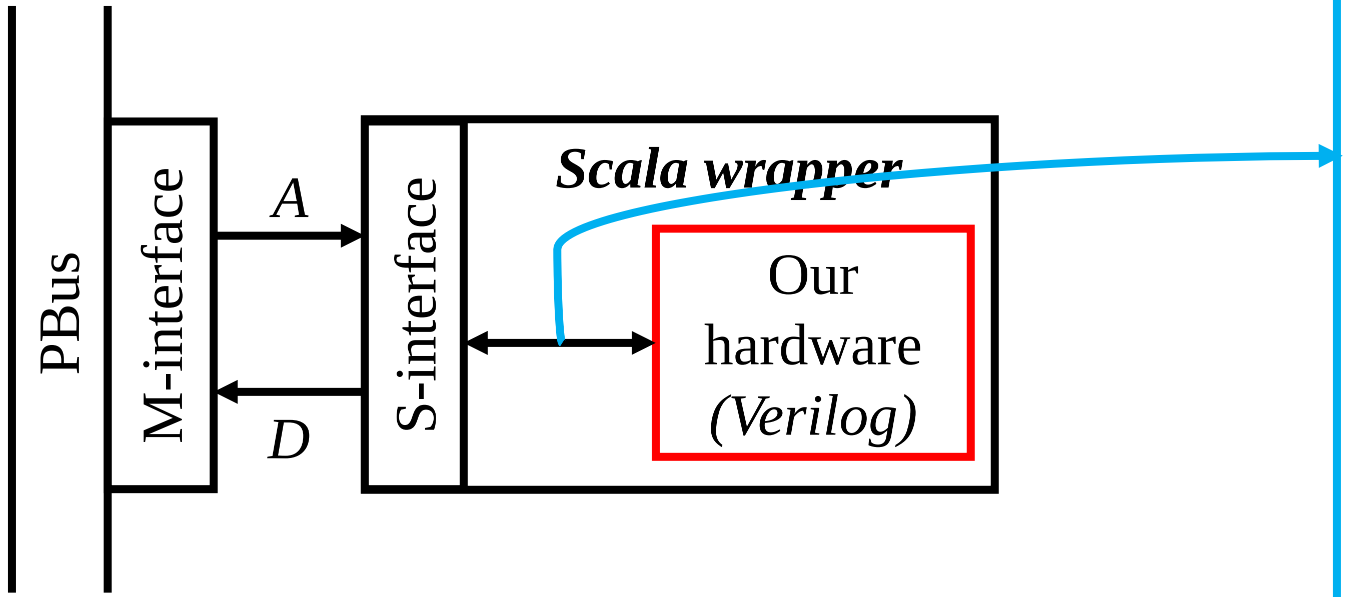
Adding custom hardware to the **Scala** system and then controlling it on software is the *core* knowledge of this course.



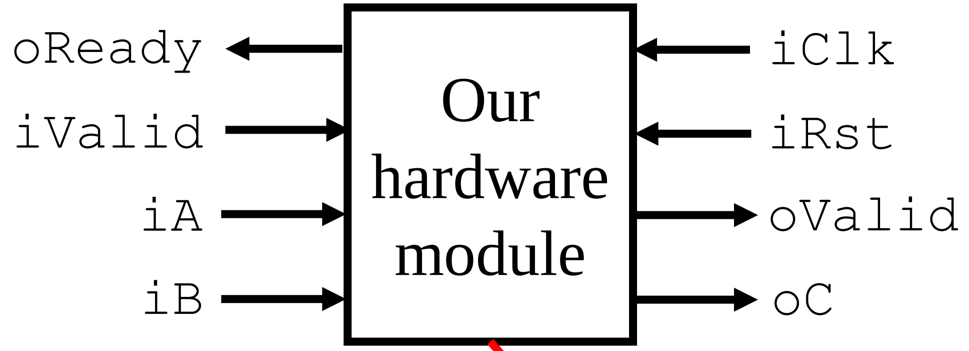
To do that, we have to:

1. Understand the bus protocol (*TileLink*) and memory-mapped communication.
2. Prepare a custom hardware in Verilog (GCD, Greatest Common Divisor, will be used as an example circuit in this lecture).
3. Attach the Verilog module to the Scala system and then regenerate the system.
4. Finally, learn to control the custom hardware in the software after boot.

5. Add custom hardware (2/10) Custom hardware's signals



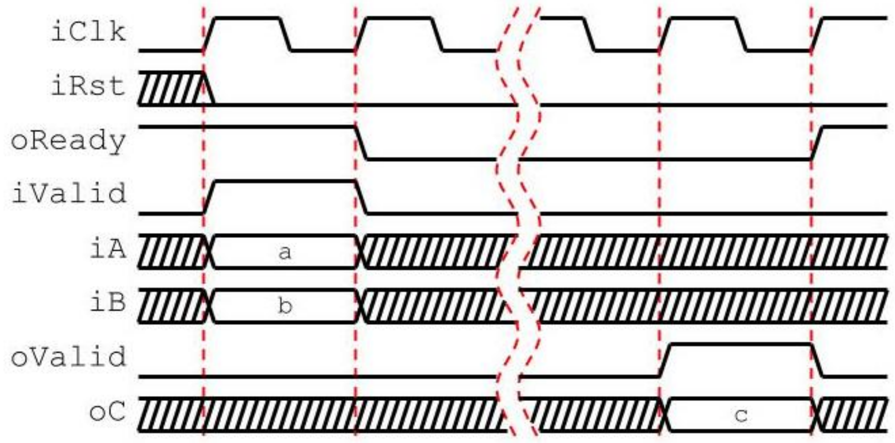
For a very simple case, iA and iB as inputs and oC as output. Then, our hardware's signals will look like this:



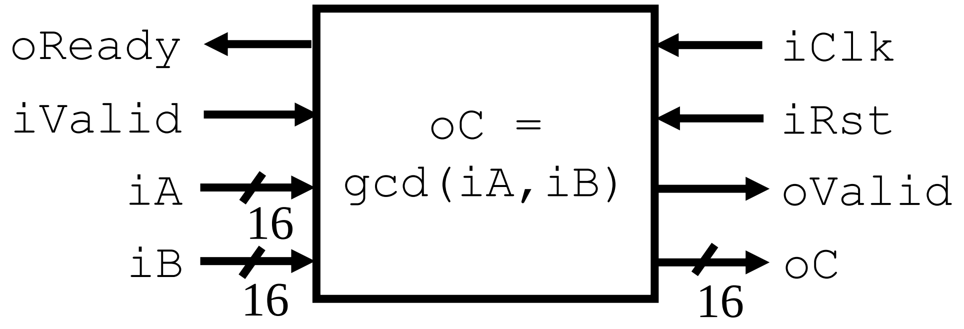
GCD pseudo-code:

```
Function: c=gcd(a,b)
BEGIN
  while(b!=0); do
    if(a>b)   swap(a,b)
    else      b=b-a
  done
  c=a
END
```

GCD waveform:



GCD was chosen to be the example:



5. **Add custom hardware** (3/10) GCD register-map

With the GCD module, the memory-register map can be chosen like this:

#Byte (total: 4-Byte *or* 32-bit per register)

Register names

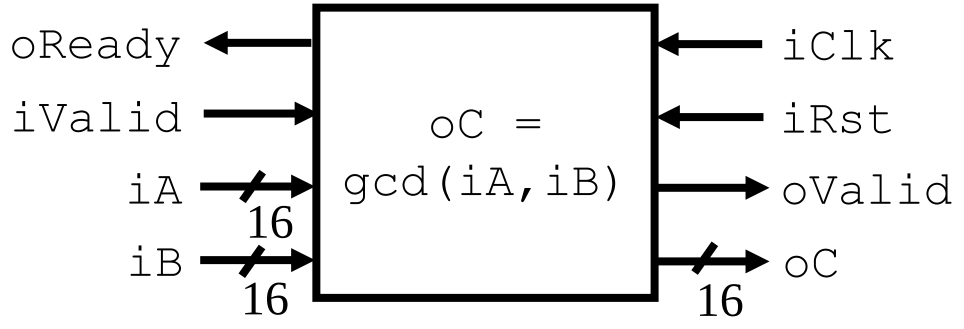
	3	2	1	0
trigger	not use	not use	iRst	iValid
data_a	not use	not use	iA	iA
data_b	not use	not use	iB	iB
data_c	not use	not use	oC	oC
status	not use	not use	not use	oReady

Offset

0x0	*p
0x4	*(p+1)
0x8	*(p+2)
0xC	*(p+3)
0x10	*(p+4)

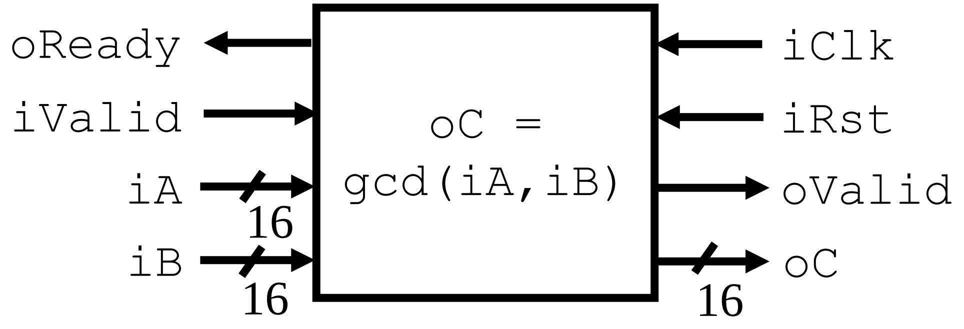
Hardware's point-of-view

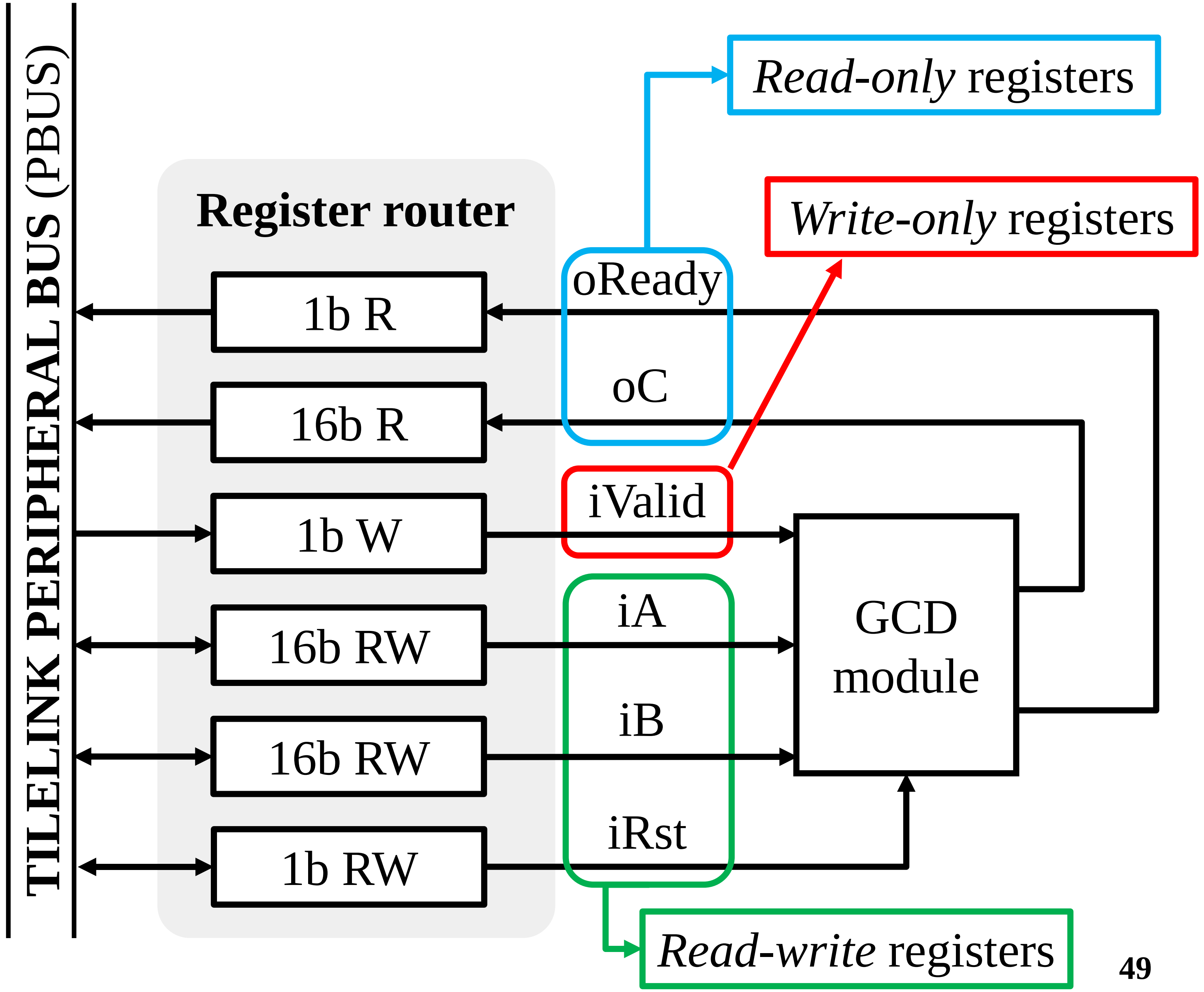
Software's point-of-view



5. Add custom hardware (4/10) GCD register-router

	3	2	1	0
trigger	▨	▨	iRst	iValid
data_a	▨	▨	iA	
data_b	▨	▨	iB	
data_c	▨	▨	oC	
status	▨	▨	▨	oReady

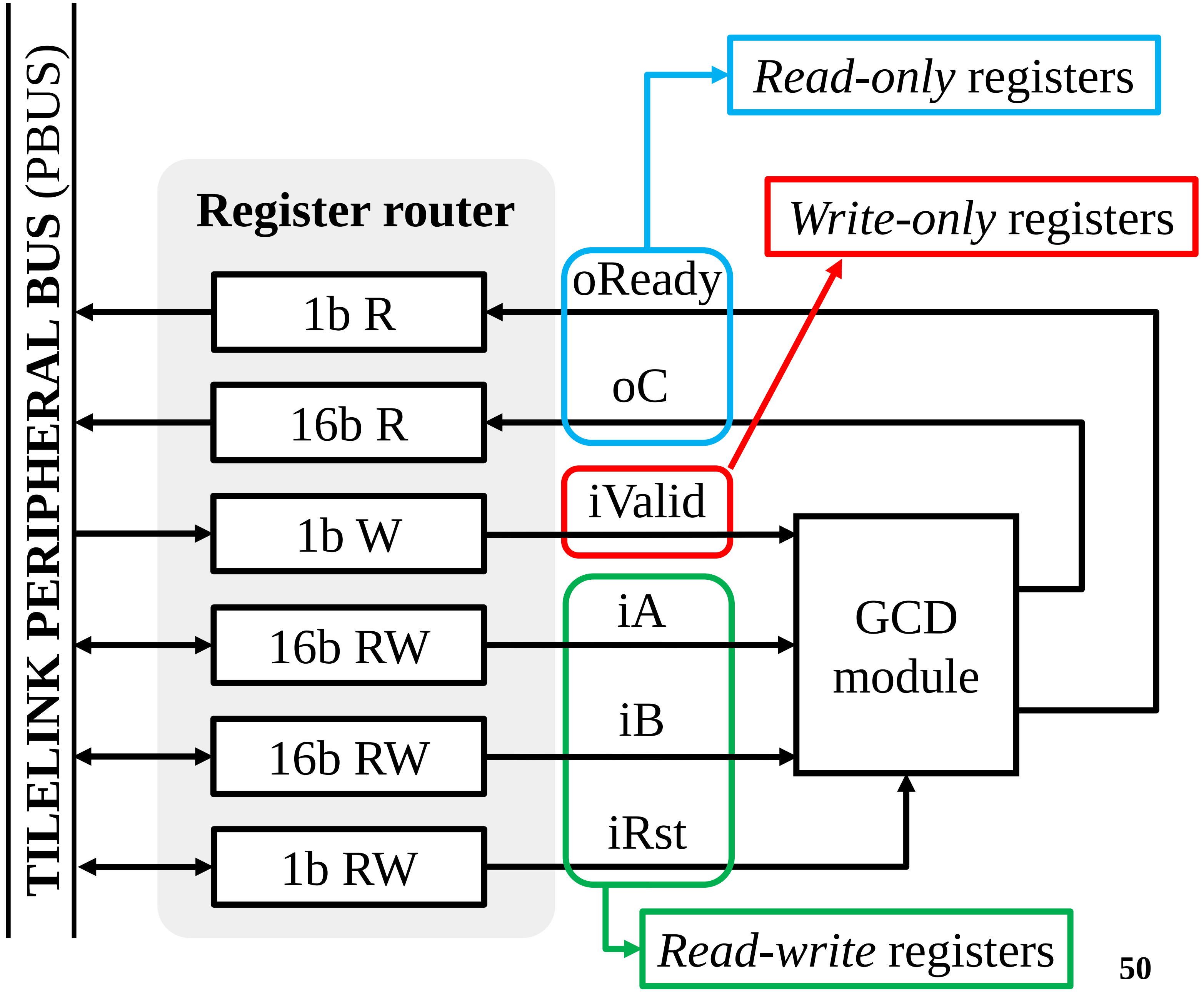




5. Add custom hardware (5/10) GCD software example

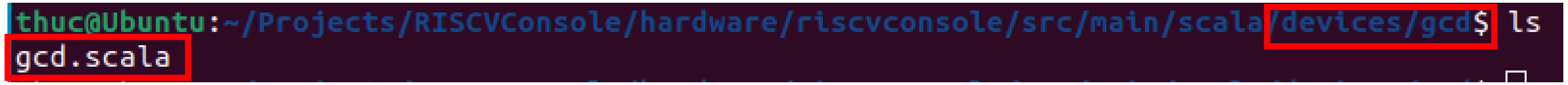
Software example

```
// Reset the hardware
_REG32(gcd_reg, GCD_TRIGGER) =
0x00000100;
_REG32(gcd_reg, GCD_TRIGGER) =
0x00000000;

// Write A & B values
_REG32(gcd_reg, GCD_DATA_A) = 7;
_REG32(gcd_reg, GCD_DATA_B) = 3;

// Trigger and wait
_REG32(gcd_reg, GCD_TRIGGER) =
0x00000001;
while(!(_REG32(gcd_reg,
GCD_STATUS) && 0x00000001));

// Get and print the result
int c = _REG32(gcd_reg,
GCD_DATA_C);
kprintf("GCD of 7 and 3 is %d\n",
c);
```



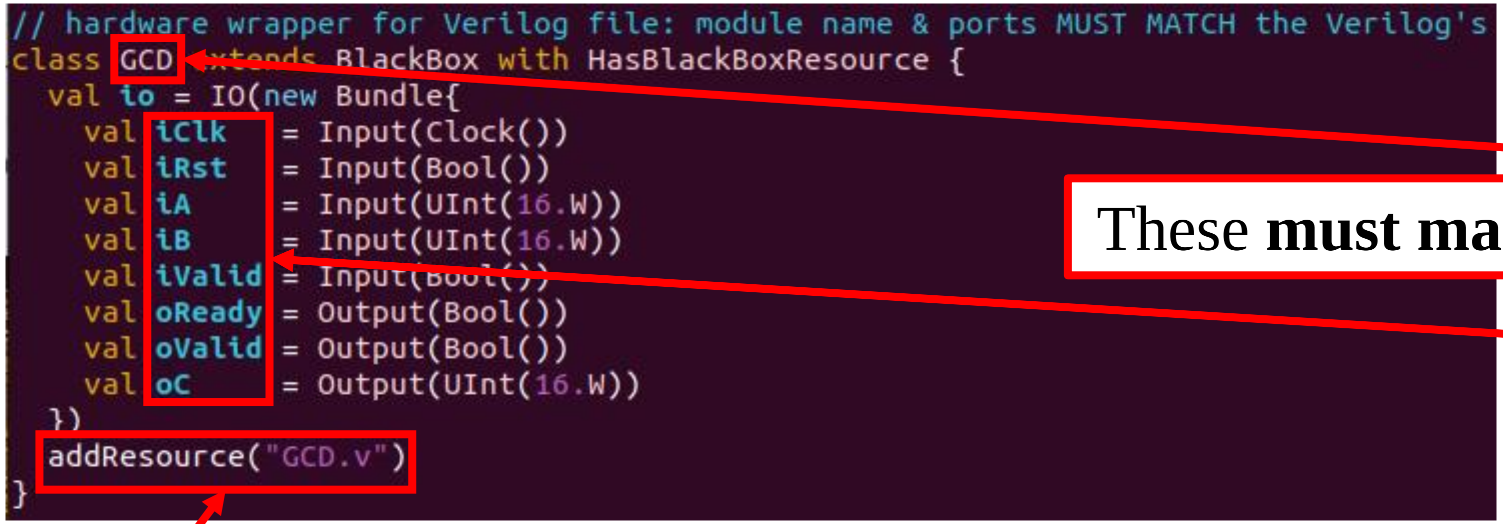
5. Add custom hardware (6/10) GCD Scala file

The ***Scala wrapper file*** must be created under the `devices/` folder:



```
thuc@Ubuntu:~/Projects/RISCVConsole/hardware/riscvconsole/src/main/scala/devices/gcd$ ls
gcd.scala
```

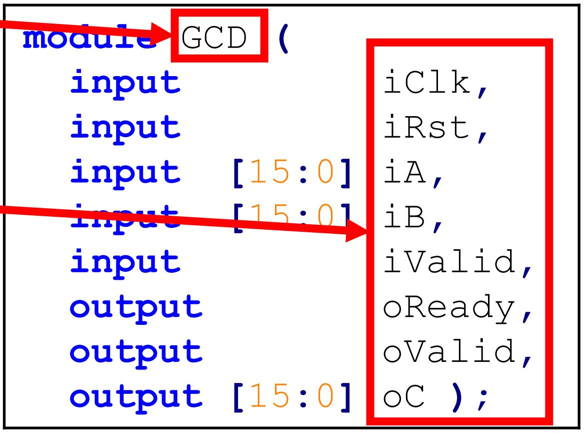
In the **`gcd.scala`** file, make a “fake” class that contains only the ports:



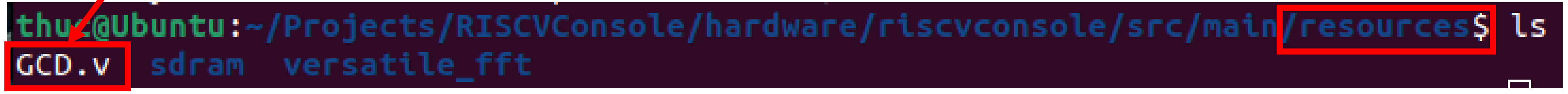
```
// hardware wrapper for Verilog file: module name & ports MUST MATCH the Verilog's
class GCD extends BlackBox with HasBlackBoxResource {
  val io = IO(new Bundle{
    val iClk   = Input(Clock())
    val iRst   = Input(Bool())
    val iA     = Input(UInt(16.W))
    val iB     = Input(UInt(16.W))
    val iValid = Input(Bool())
    val oReady = Output(Bool())
    val oValid = Output(Bool())
    val oC     = Output(UInt(16.W))
  })
  addResource("GCD.v")
}
```

These **must match**

In **Verilog** file:



```
module GCD (
  input           iClk,
  input           iRst,
  input   [15:0]  iA,
  input   [15:0]  iB,
  input           iValid,
  output          oReady,
  output          oValid,
  output  [15:0]  oC );
```

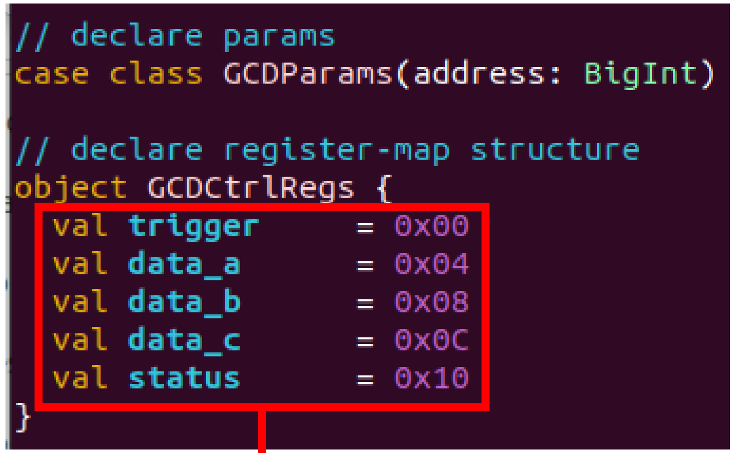


```
thuc@Ubuntu:~/Projects/RISCVConsole/hardware/riscvconsole/src/main/resources$ ls
GCD.v  sdram  versatile_fft
```

The `addResource()` will automatically find the given file-name in the `resources/` folder

5. Add custom hardware (7/10) Wires & regs declaration

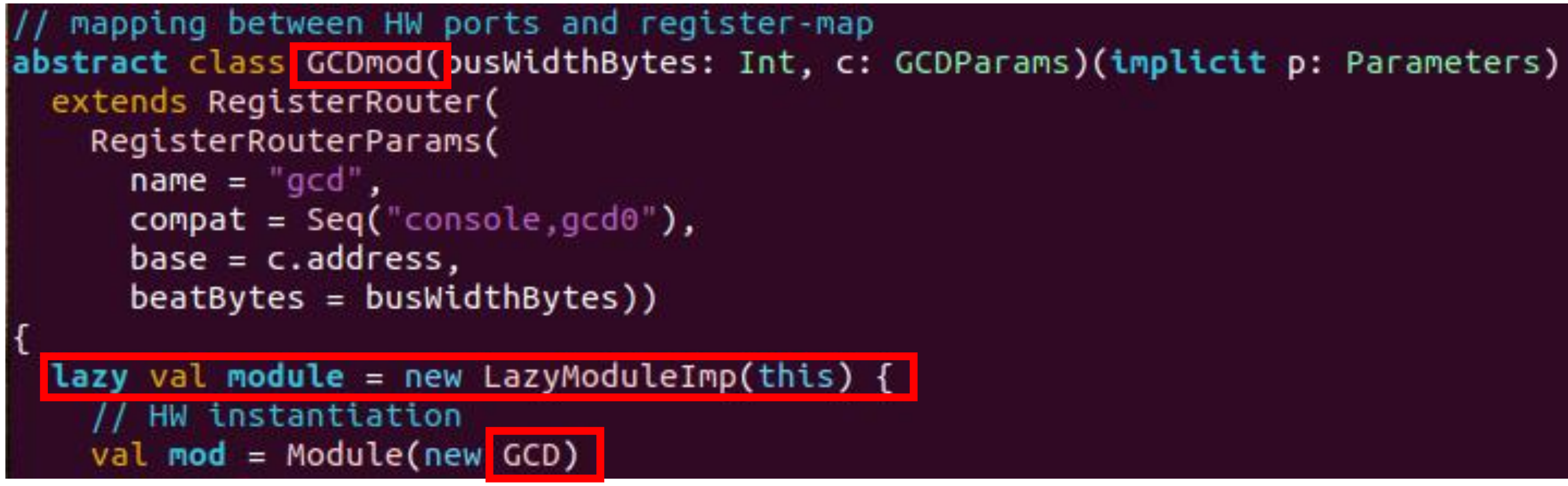
In the `gcd.scala` file:



```
// declare params
case class GCDParams(address: BigInt)

// declare register-map structure
object GCDCtrlRegs {
  val trigger      = 0x00
  val data_a       = 0x04
  val data_b       = 0x08
  val data_c       = 0x0C
  val status       = 0x10
}
```

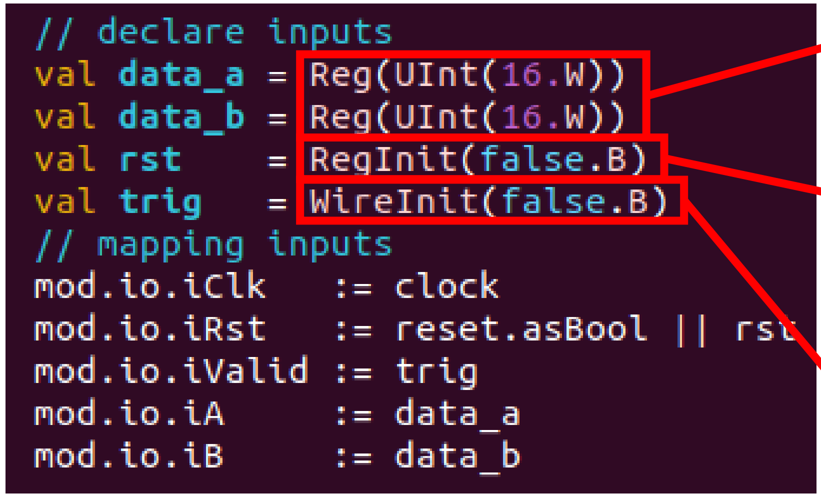
Declare registers with its offset.

Create a <u>*Scala wrapper*</u> called **GCDmod** that includes the previous **GCD** class as sub-module.



```
// mapping between HW ports and register-map
abstract class GCDmod(busWidthBytes: Int, c: GCDParams)(implicit p: Parameters)
  extends RegisterRouter(
    RegisterRouterParams(
      name = "gcd",
      compat = Seq("console,gcd0"),
      base = c.address,
      beatBytes = busWidthBytes))
{
 lazy val module = new LazyModuleImp(this) {
   // HW instantiation
   val mod = Module(new GCD)
```

Declare the *wires* and *regs* in the `GCDmod` and connect them to the `GCD` sub-module:



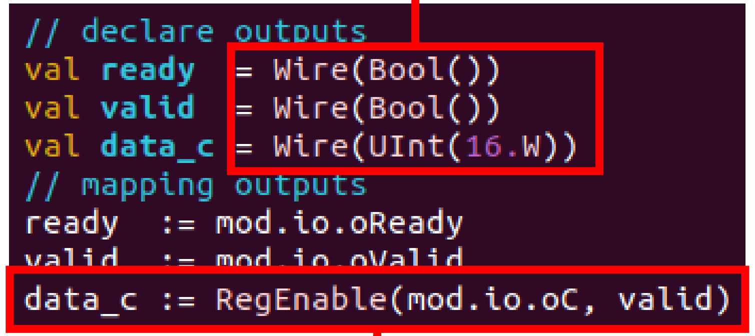
```
// declare inputs
val data_a = Reg(UInt(16.W))
val data_b = Reg(UInt(16.W))
val rst    = RegInit(false.B)
val trig   = WireInit(false.B)
// mapping inputs
mod.io.iClk    := clock
mod.io.iRst    := reset.asBool || rst
mod.io.iValid  := trig
mod.io.iA      := data_a
mod.io.iB      := data_b
```

Normal ***regs***

reg with init value of `0`

wire with init value of `0`

All outputs must be ***wires***



```
// declare outputs
val ready  = Wire(Bool())
val valid  = Wire(Bool())
val data_c = Wire(UInt(16.W))
// mapping outputs
ready  := mod.io.oReady
valid  := mod.io.oValid
data_c := RegEnable(mod.io.oC, valid)
```

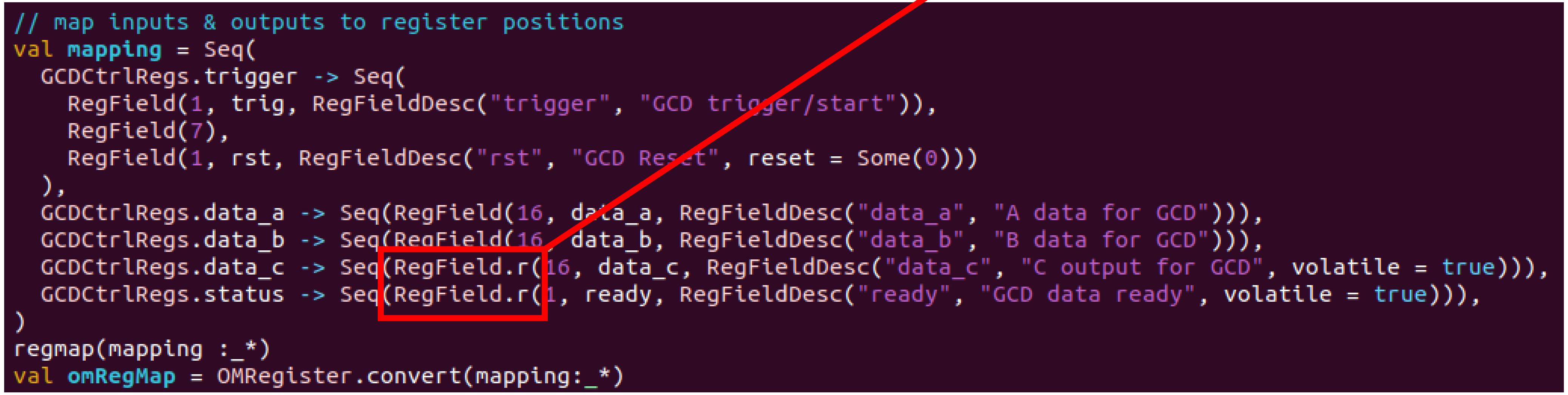
`data_c` captures `oC` at `valid`

5. Add custom hardware (8/10) Map to registers

In the `gcd.scala` file:

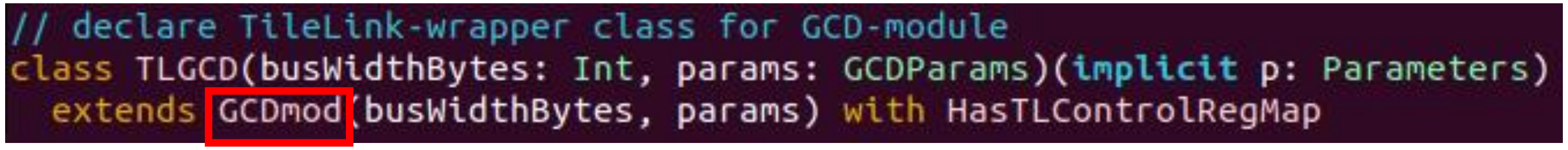
`RegField.r` instead of `RegFiled` means **read-only**.

Mapping between signals and registers:



```
// map inputs & outputs to register positions
val mapping = Seq(
  GCDCtrlRegs.trigger -> Seq(
    RegField(1, trig, RegFieldDesc("trigger", "GCD trigger/start")),
    RegField(7),
    RegField(1, rst, RegFieldDesc("rst", "GCD Reset", reset = Some(0)))
  ),
  GCDCtrlRegs.data_a -> Seq(RegField(16, data_a, RegFieldDesc("data_a", "A data for GCD"))),
  GCDCtrlRegs.data_b -> Seq(RegField(16, data_b, RegFieldDesc("data_b", "B data for GCD"))),
  GCDCtrlRegs.data_c -> Seq(RegField.r(16, data_c, RegFieldDesc("data_c", "C output for GCD", volatile = true))),
  GCDCtrlRegs.status -> Seq(RegField.r(1, ready, RegFieldDesc("ready", "GCD data ready", volatile = true))),
)
regmap(mapping :_*)
val omRegMap = OMRegister.convert(mapping:_*)
```

Finally, create a *TileLink wrapper* called **TLGCD** that extends from the previous **GCDmod** class:

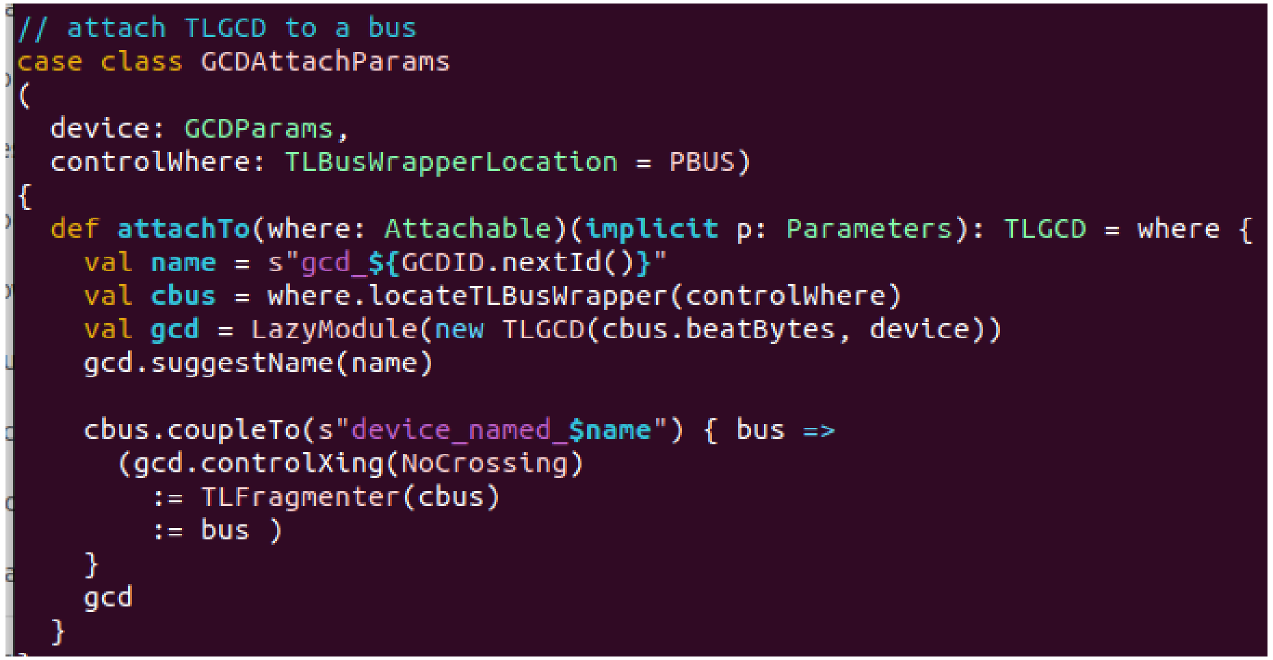


```
// declare TileLink-wrapper class for GCD-module
class TLGCD(busWidthBytes: Int, params: GCDParams)(implicit p: Parameters)
  extends GCDmod(busWidthBytes, params) with HasTLControlRegMap
```

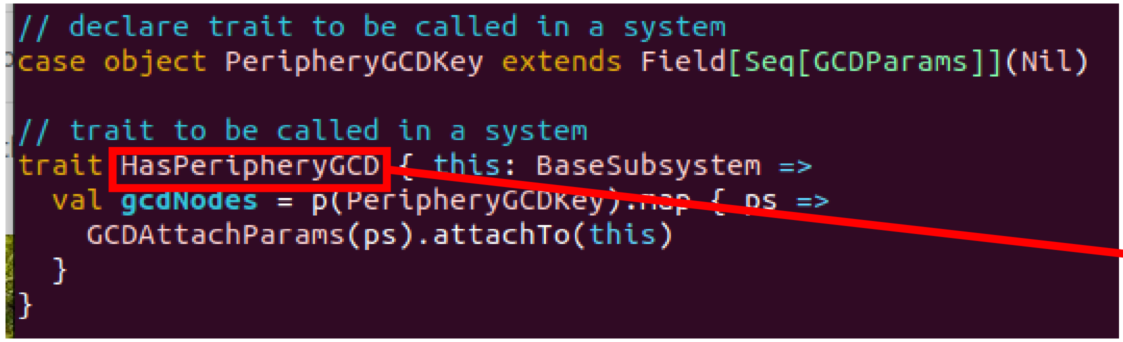
5. Add custom hardware (9/10) System traits

In the `gcd.scala` file:

Create a **trait** to be called later in the RISCVConsole System:



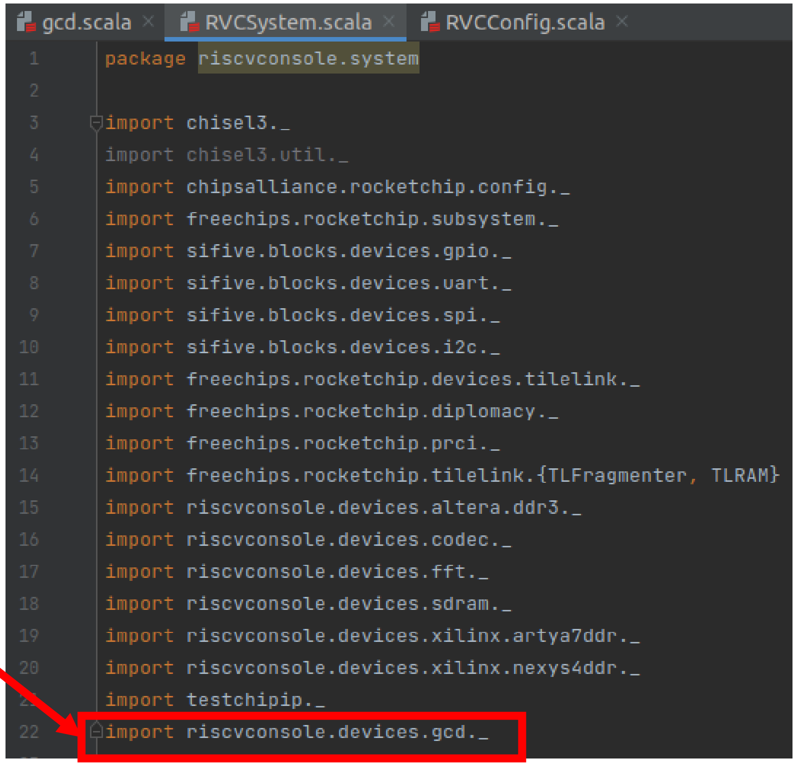
```
// attach TLGCD to a bus
case class GCDAttachParams
(
  device: GCDParams,
  controlWhere: TLBusWrapperLocation = PBUS)
{
  def attachTo(where: Attachable)(implicit p: Parameters): TLGCD = where {
    val name = s"gcd_${GCDID.nextId()}"
    val cbus = where.locateTLBusWrapper(controlWhere)
    val gcd = LazyModule(new TLGCD(cbus.beatBytes, device))
    gcd.suggestName(name)

    cbus.coupleTo(s"device_named_$name") { bus =>
      (gcd.controlXing(NoCrossing)
        := TLFragmenter(cbus)
        := bus )
    }
    gcd
  }
```



```
// declare trait to be called in a system
case object PeripheryGCDKey extends Field[Seq[GCDParams]](Nil)

// trait to be called in a system
trait HasPeripheryGCD { this: BaseSubsystem =>
  val gcdNodes = p(PeripheryGCDKey).map { ps =>
    GCDAttachParams(ps).attachTo(this)
  }
}
```

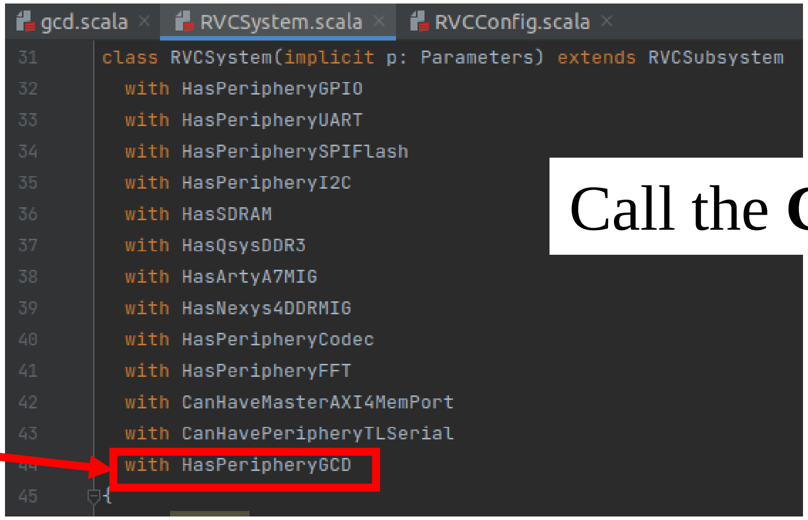
In the `RVCSystem.scala` file:

Import the **GCD package**:



gcd.scala × RVCSystem.scala × RVCConfig.scala ×

```
package riscvconsole.system

import chisel3._
import chisel3.util._
import chipsalliance.rocketchip.config._
import freechips.rocketchip.subsystem._
import sifive.blocks.devices.gpio._
import sifive.blocks.devices.uart._
import sifive.blocks.devices.spi._
import sifive.blocks.devices.i2c._
import freechips.rocketchip.devices.tilelink._
import freechips.rocketchip.diplomacy._
import freechips.rocketchip.prci._
import freechips.rocketchip.tilelink.{TLFragmenter, TLRAM}
import riscvconsole.devices.altera.ddr3._
import riscvconsole.devices.codec._
import riscvconsole.devices.fft._
import riscvconsole.devices.sdram._
import riscvconsole.devices.xilinx.artya7ddr._
import riscvconsole.devices.xilinx.nexys4ddr._
import testchipip._
import riscvconsole.devices.gcd._
```



gcd.scala × RVCSystem.scala × RVCConfig.scala ×

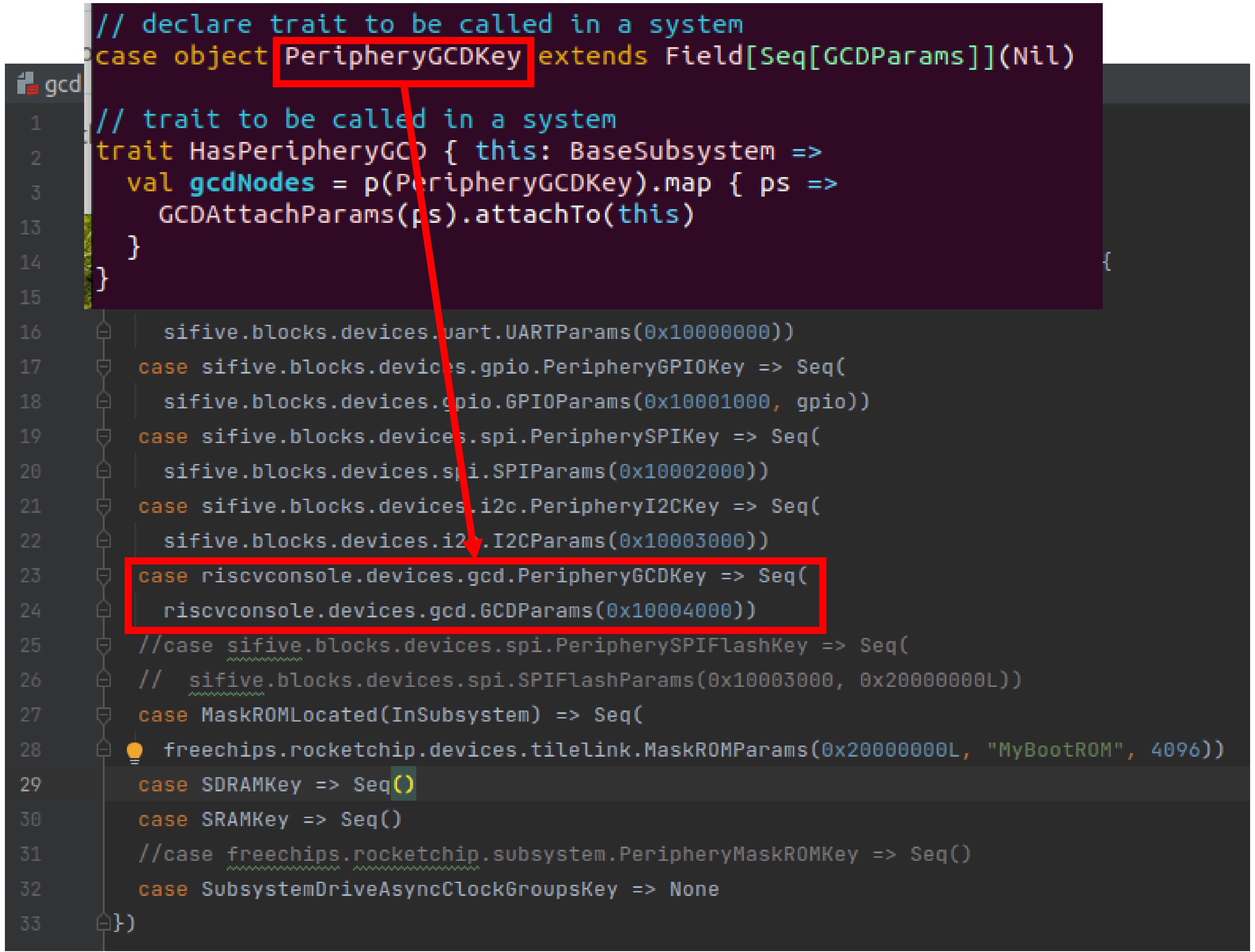
```
class RVCSystem(implicit p: Parameters) extends RVCSubsystem
  with HasPeripheryGPIO
  with HasPeripheryUART
  with HasPeripherySPIFlash
  with HasPeripheryI2C
  with HasSDRAM
  with HasQsysDDR3
  with HasArtyA7MIG
  with HasNexys4DDRMIG
  with HasPeripheryCodec
  with HasPeripheryFFT
  with CanHaveMasterAXI4MemPort
  with CanHavePeripheryTLSerial
  with HasPeripheryGCD
{
```

Call the **GCD trait**:

5. Add custom hardware (10/10) Assign the address

In the `RVCConfig.scala` file:

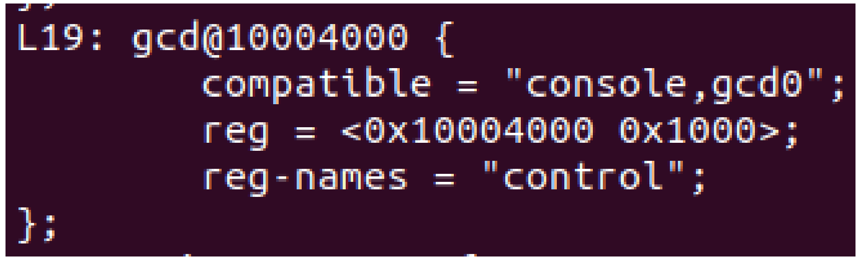
Put the **GCDKey** and assign an address.:



```
// declare trait to be called in a system
case object PeripheryGCDKey extends Field[Seq[GCDParams]](Nil)

// trait to be called in a system
trait HasPeripheryGCD { this: BaseSubsystem =>
  val gcdNodes = p(PeripheryGCDKey).map { ps =>
    GCDAttachParams(ps).attachTo(this)
  }
}
```

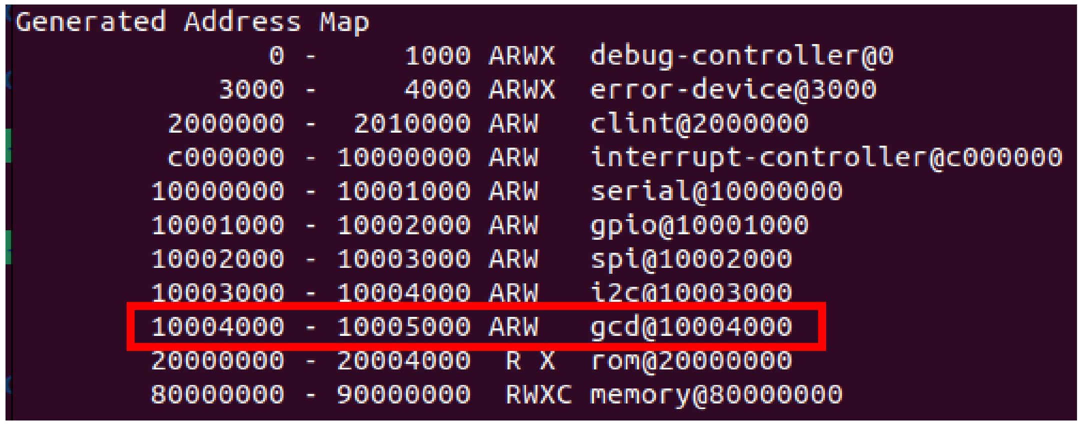
```
    sifive.blocks.devices.uart.UARTParams(0x10000000))
  case sifive.blocks.devices.gpio.PeripheryGPIOKey => Seq(
    sifive.blocks.devices.gpio.GPIOParams(0x10001000, gpio))
  case sifive.blocks.devices.spi.PeripherySPIKey => Seq(
    sifive.blocks.devices.spi.SPIParams(0x10002000))
  case sifive.blocks.devices.i2c.PeripheryI2CKey => Seq(
    sifive.blocks.devices.i2c.I2CParams(0x10003000))
  case riscvconsole.devices.gcd.PeripheryGCDKey => Seq(
    riscvconsole.devices.gcd.GCDParams(0x10004000))
  //case sifive.blocks.devices.spi.PeripherySPIFlashKey => Seq(
  //  sifive.blocks.devices.spi.SPIFlashParams(0x10003000, 0x20000000L))
  case MaskROMLocated(InSubsystem) => Seq(
    freechips.rocketchip.devices.tilelink.MaskROMParams(0x20000000L, "MyBootROM", 4096))
  case SDRAMKey => Seq()
  case SRAMKey => Seq()
  //case freechips.rocketchip.subsystem.PeripheryMaskROMKey => Seq()
  case SubsystemDriveAsyncClockGroupsKey => None
})
```

Next time, `$ make default` will see a new **gcd** module is included in the device-tree:



```
L19: gcd@10004000 {
        compatible = "console,gcd0";
        reg = <0x10004000 0x1000>;
        reg-names = "control";
};
```

The **gcd** is added in the address map:



```
Generated Address Map
               0 -     1000 ARWX  debug-controller@0
            3000 -     4000 ARWX  error-device@3000
         2000000 -  2010000 ARW   clint@2000000
         c000000 - 10000000 ARW   interrupt-controller@c000000
        10000000 - 10001000 ARW   serial@10000000
        10001000 - 10002000 ARW   gpio@10001000
        10002000 - 10003000 ARW   spi@10002000
        10003000 - 10004000 ARW   i2c@10003000
        10004000 - 10005000 ARW   gcd@10004000
        20000000 - 20004000  R X  rom@20000000
        80000000 - 90000000  RWXC memory@80000000
```

国立大学法人
電気通信大学
The University of Electro-Communications

Pham Laboratory
Integrated circuit design laboratory

Outline

1. Boot sequence
2. RISC-V and RISC-V ISA
3. Our RISC-V computer system
4. Hardware make flow
5. Add custom hardware
6. Cryptosystem

6. Cryptosystem (1/14) Pillars in cybersecurity

1. Confidentiality:	the data is ciphered → un-authorized party cannot read the data
2. Integrity:	the data is original → un-authorized party cannot modify the data
3. Availability:	authorized parties can access the data anytime without difficulty
4. Authentication:	verify sender and/or reader identification → Reader knows who the sender is, and vice versa
5. Non-repudiation:	the data is sent only to an authorized party → un-authorized party cannot copy the data

****Note:*** number 1, 2, and 3 are also called the CIA triad.

6. **Cryptosystem** (2/14) Pillars in cybersecurity

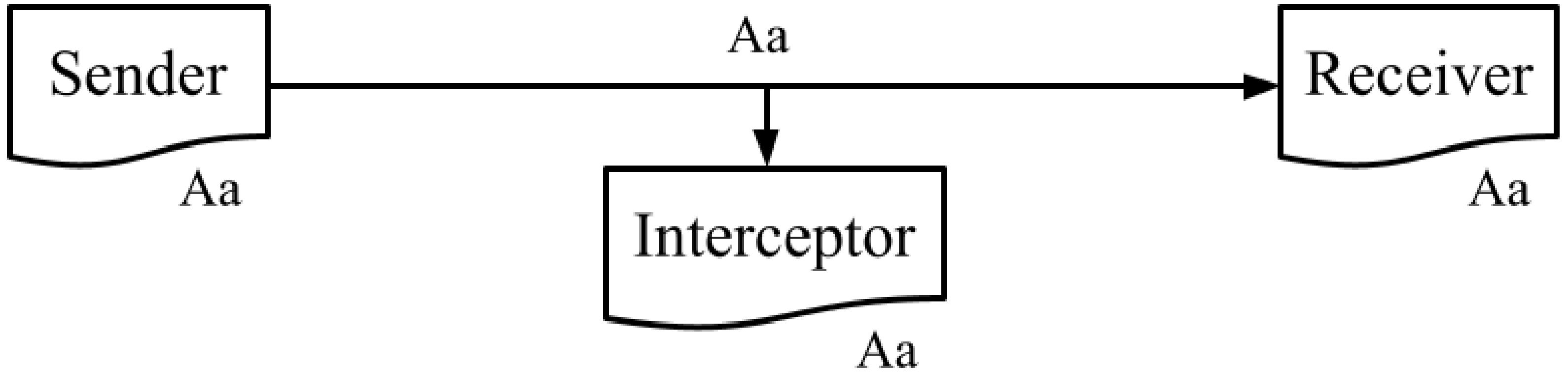
A combination of Secure boot and **TEE** / **TPM** solves the problems of *Authentication*, *Non-repudiation*, and *Availability*.

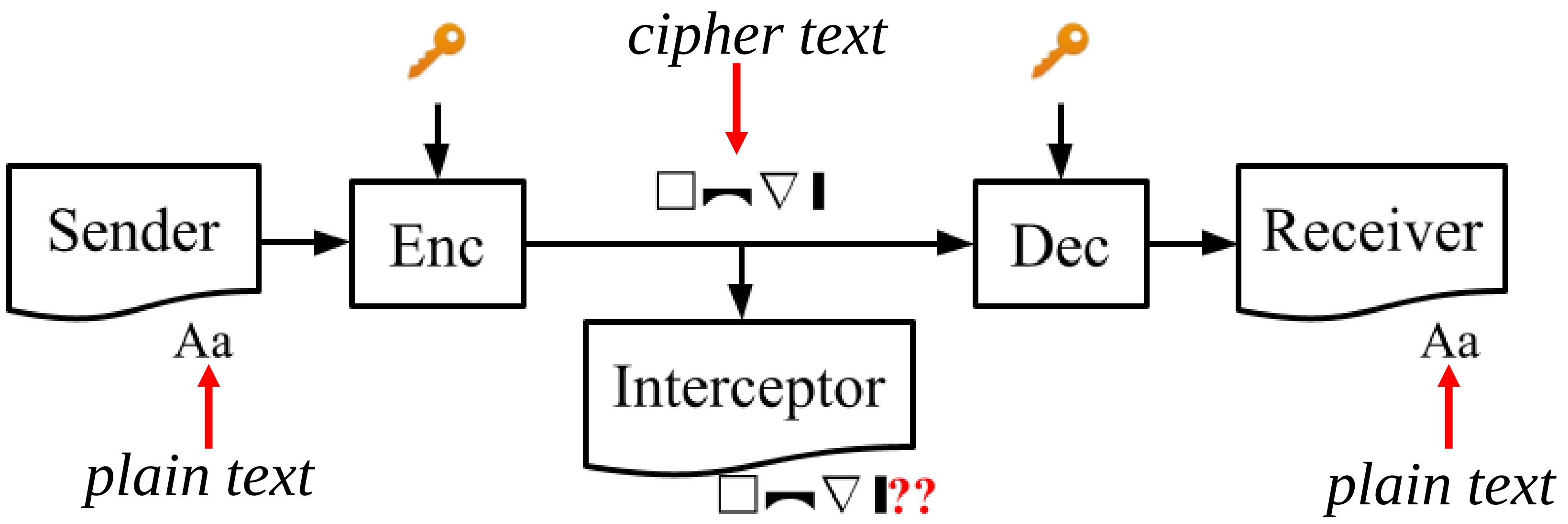
For *Integrity* and *Confidentiality*, Secure boot + **TEE** / **TPM** are not directly solve them, but lay a foundation for other completed solutions.

6. Cryptosystem (3/14) Eavesdropping problem

Every cryptosystem begins with the Eavesdropping problem:

- **Eavesdropping:** a hacker intercepts, deletes, or modifies data that is transmitted between two parties.





The solution is always an encryption mechanism.

6. Cryptosystem (4/14) Eavesdropping problem

We need to transfer data securely over the *untrusted* transmission line.

⇒ Use a cipher algorithm to encrypt/decrypt the data before/after the transmission.

⇒ The key **k** of a cipher algorithm needs to be agreed upon before the transmission.

⇒ We need to transfer the key **k** securely over the *untrusted* transmission line.

This is a classical ***chicken-and-egg*** problem

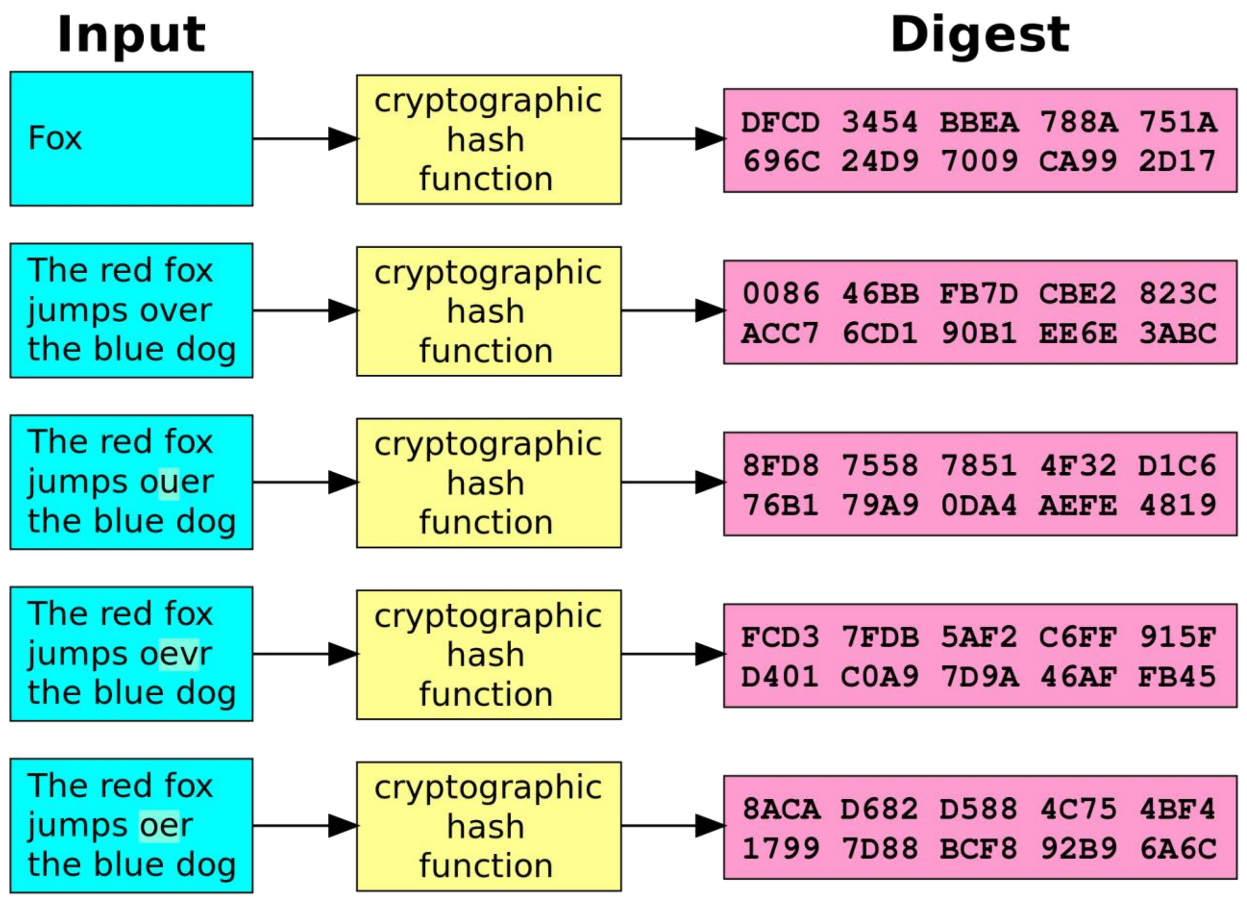
In-a-nut-shell:

Sender and **Receiver** need to share the *key* **k** before transmission: How we can do that?

Solve this problem is the main goal of a cryptosystem

6. Cryptosystem (5/14) Hash function

Hash function, also called **digest** function, is a function that converts a given string *(or any data alike)* with any length to a fixed-length result.



Input
Digest
Fox
The red fox jumps over the blue dog
The red fox jumps ouer the blue dog
The red fox jumps oevr the blue dog
The red fox jumps oer the blue dog
cryptographic hash function
DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

Hash function is a <u>one-way</u> function
⇒ We cannot restore the original data

Imagine the *meat grinder*:

The process is irreversible.

The <u>more chaotic</u> the **result** is, the <u>better</u> the **hash** algorithm.

6. Cryptosystem (6/14) Crypto-key scheme

Crypto-key scheme will use an **asymmetric encryption** algorithm and have <u>three</u> functions: `gen-key()`, `sign()`, and `verify()`

`(k,k′) = gen-key(x)`

- Input `x` : called a seed, usually a random number
- Output `k` and `k′` : called a pair-key, interchangeable, have the same fixed-length

`s = sign(m,k)`

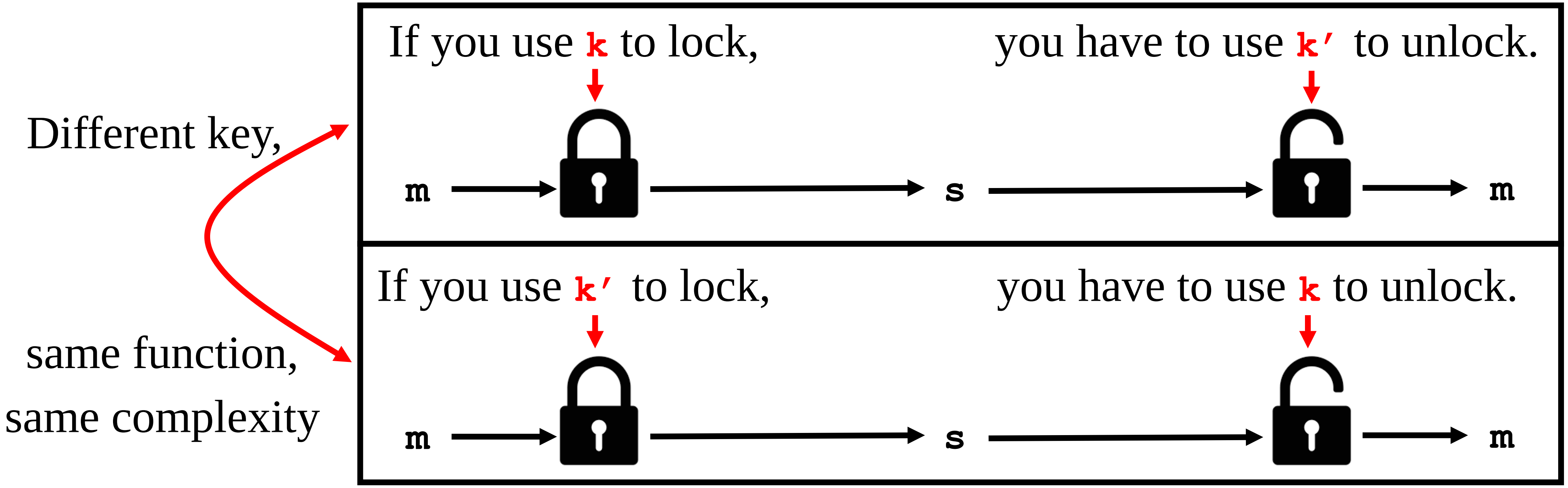
- Input `m` : called a message, could be anything
- Input `k` : called a key, retrieve from `gen-key()`
- Output `s` : called a signature, has a fixed-length

`m′ = verify(s,k′)`

- Input `s` : called a signature, retrieve from `sign()`
- Input `k′` : called a key, retrieve from `gen-key()`
- Output `m′` : called a retrieved-message, suppose to be identical with `m`

6. **Cryptosystem** (7/14) Crypto-key scheme

The **pair-key** is interchangeable



From the pair-key of k and k′, one will be chosen as public key ***P***, and the other as secret key ***S***.

The only different is, after one has been chosen as public key ***P*** *(publish for everyone to know)*, you have to conceal the secret key ***S*** *(only known to yourself)*.

6. **Cryptosystem** (8/14) Completed solution

Back to the initial problem…

Let's both parties have their own pair-keys

Side **A**

$(\boldsymbol{P_A}, \boldsymbol{S_A})$ = **`gen-key`**$(\boldsymbol{a})$

$\boldsymbol{P_A}$ is know by everyone on the internet.
On the internet, only **A** know its $\boldsymbol{S_A}$.

Side **B**

$(\boldsymbol{P_B}, \boldsymbol{S_B})$ = **`gen-key`**$(\boldsymbol{b})$

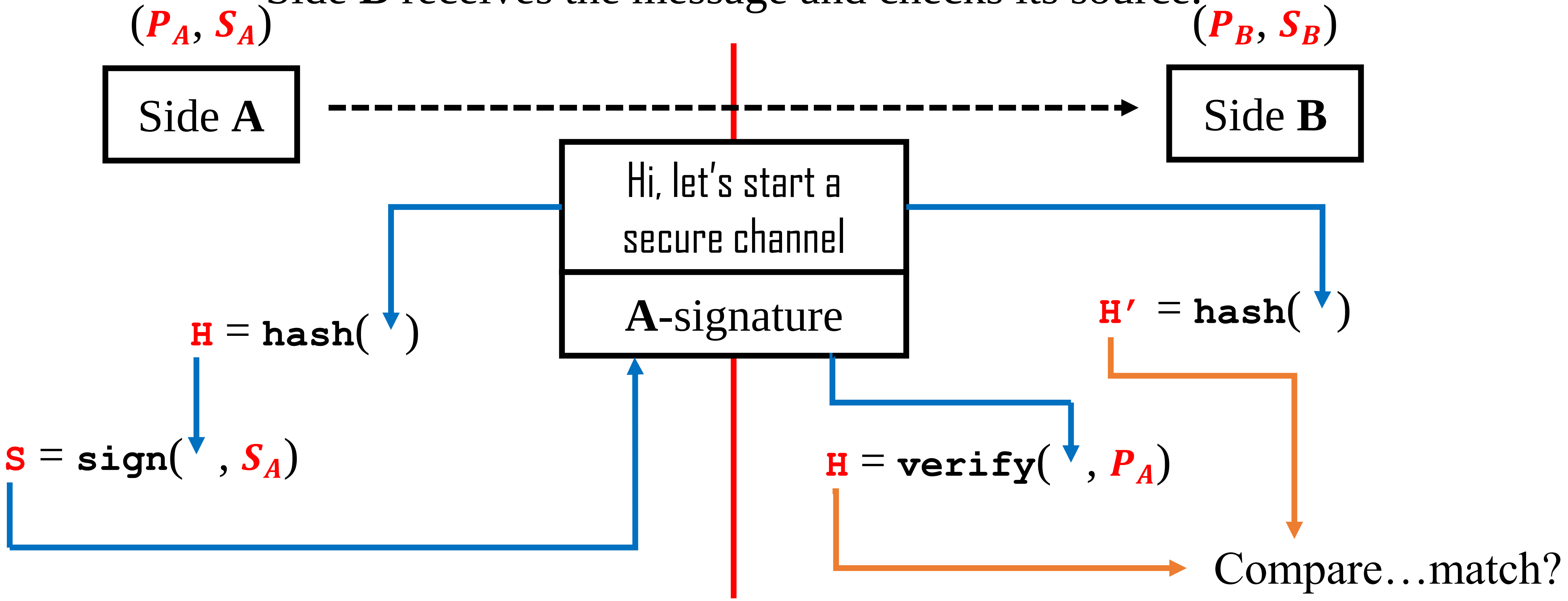
$\boldsymbol{P_B}$ is know by everyone on the internet.
On the internet, only **B** know its $\boldsymbol{S_B}$.

- This step usually run offline with the **highest** *security settings*.
- The generated pair-keys now become their **identities**.

6. Cryptosystem (9/14) Completed solution

Side **A** wants to start the secure channel.

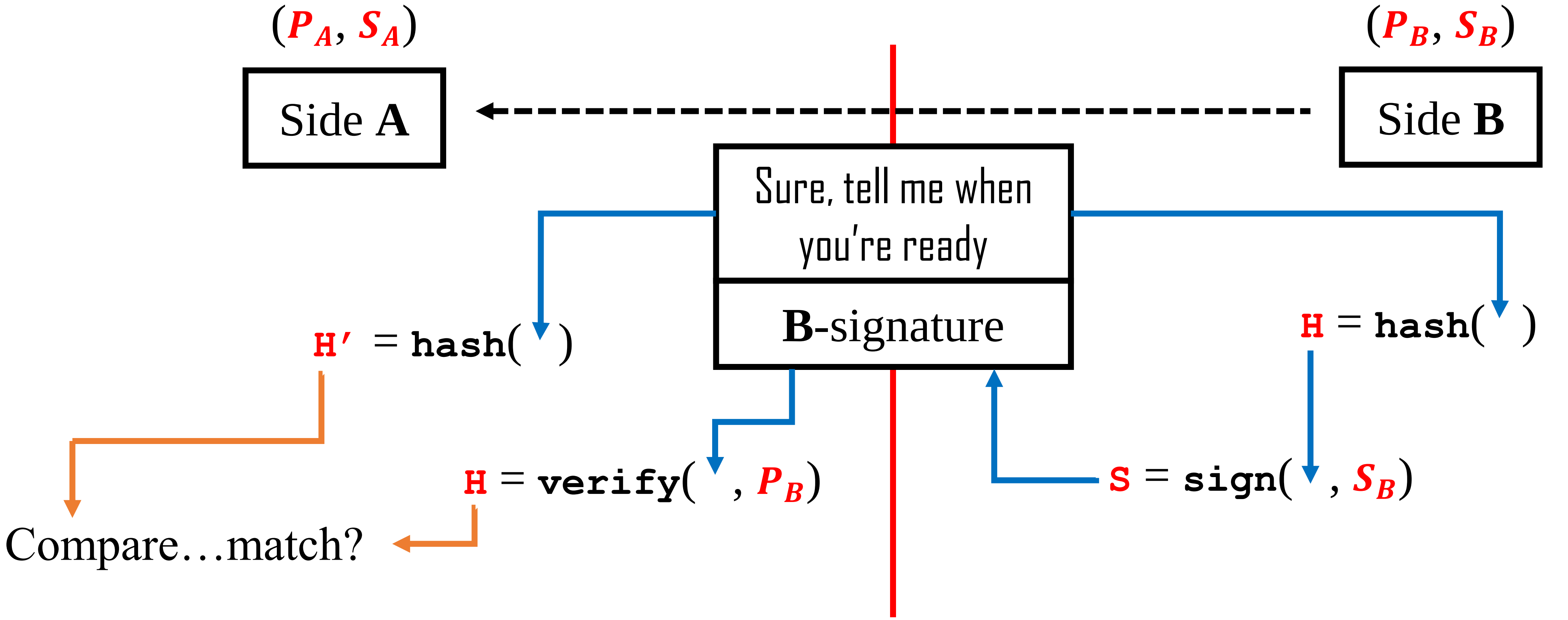
Side **B** receives the message and checks its source.



- The body of message, **m**, is not encrypted *(yet)*.
- **B** will know if *anyone* try to fake **A**'s **identity**.
- **B** will know if *anyone* try to modify the **original message**.

6. Cryptosystem (10/14) Completed solution

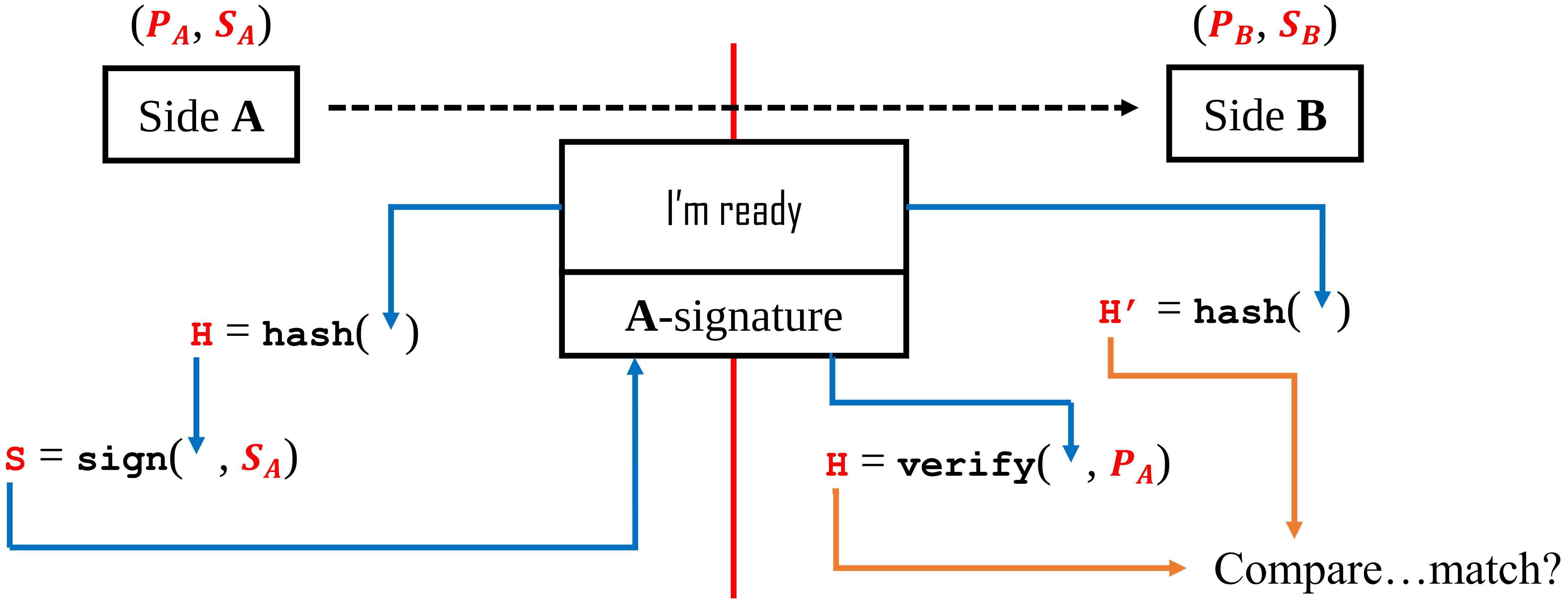
Side **B** acknowledges the initial message.



- Using the same mechanism to send and check data.
- Until now, the body of message, `m`, is still not encrypted *(yet)*.

6. Cryptosystem (11/14) Completed solution

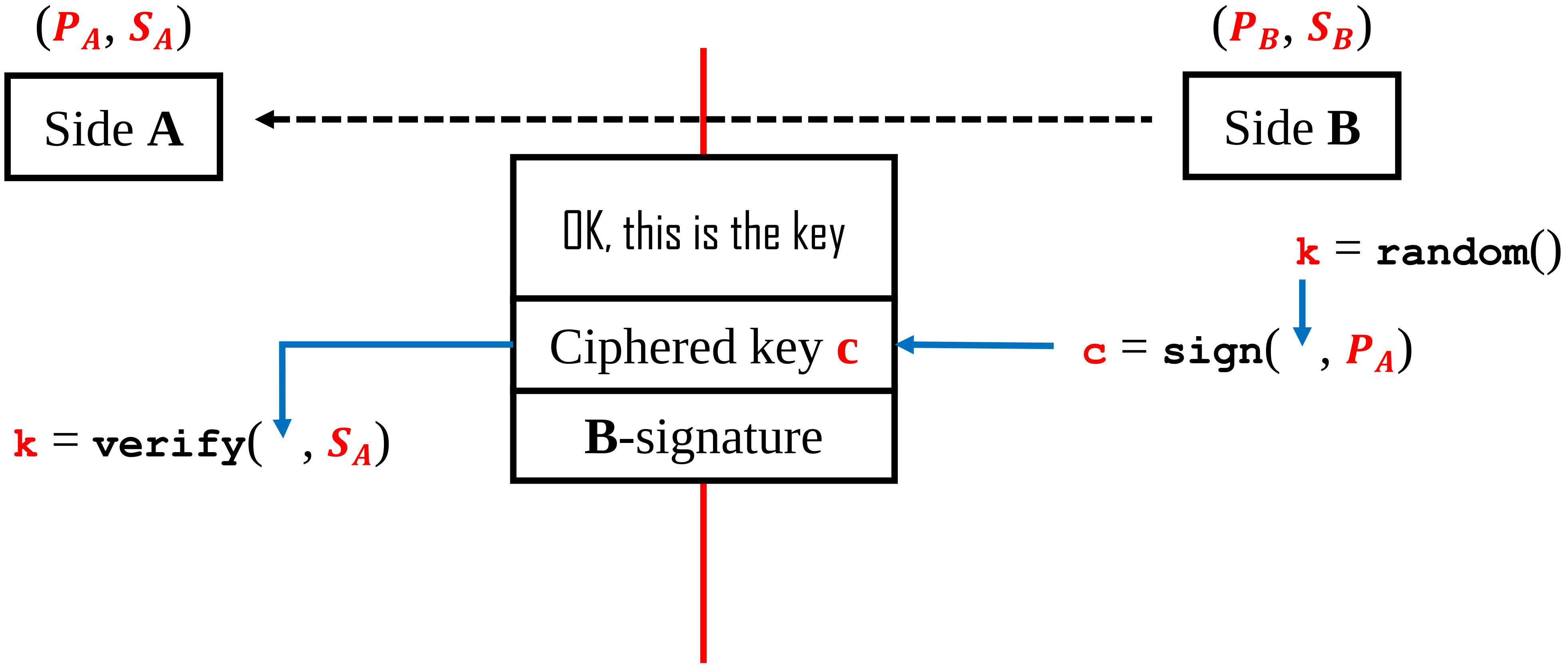
Side **A** ready for the key **k** transmission.



- Using the same mechanism to send and check data.
- Until now, the body of message, `m`, is still not encrypted *(yet)*.

6. Cryptosystem (12/14) Completed solution

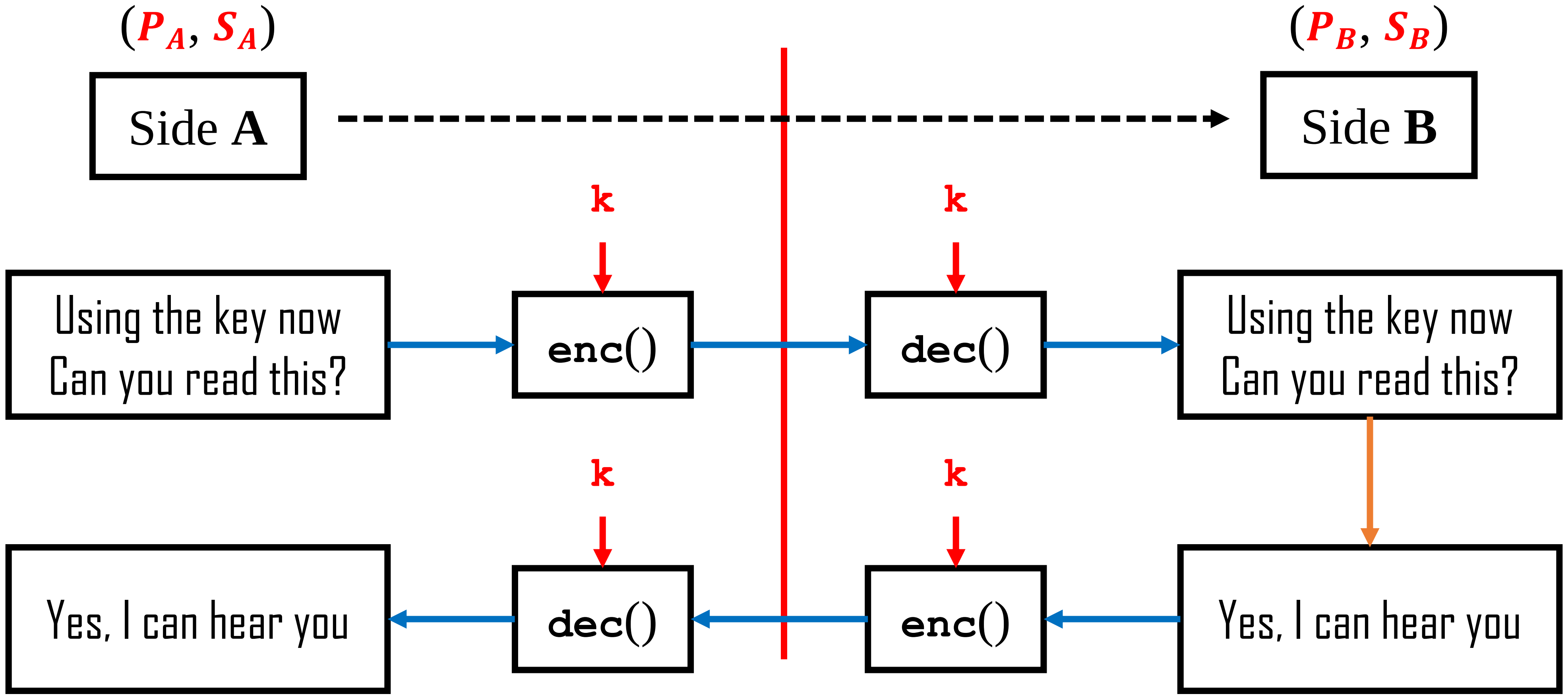
Side **B** sends the key **k** over to **A**.



- From everybody, only **A** can unlock the ciphered key **c** to retrieve the **k**.
- The attached **B**-signature is still needed for checking its source.

6. Cryptosystem (13/14) Completed solution

Side **A** tests the key **k**. Side **B** acknowledges, thus finishes the handshake.



- **A** tests the newly received key **k**.
- Now, the handshake is over. Typical cipher algorithm is now used.

6. Cryptosystem (14/14) Summary

- The primary goal of a cryptosystem is about solving the <u>eavesdropping</u> problem.
  ⇒ Thus, solving the *Integrity* and *Confidentiality* problems.

- A cryptosystem usually assumes that the **sender** and **receiver** themselves are <u>trusted</u> *(i.e., the sender and receiver* ***devices*** *are <u>not compromised</u>)*.
  ⇒ Therefore, a <u>truly completed</u> cryptosystem needs a **TPM/TEE** with *secure boot*.
  Furthermore, **TPM/TEE** + *secure boot* also solve the *Authentication*, *Non-repudiation*, and *Availability* problems.

- A typical cryptosystem needs a *hash*, a *cipher*, and a *crypto-key* scheme.
  If we view cryptosystem as a stage, then *hash*, *cipher*, and *crypto-key* are <u>roles</u> to be played, **not** <u>actors</u>. ⇒ <u>Actors</u> can be **changed**, but the <u>roles</u> are **not**.
  For example: *SHA*, *AES*, and *RSA* are <u>actors</u> to play the <u>roles</u> of *hash*, *cipher*, and *crypto-key*, respectively.

THANK YOU

Tháng 9/2023