

## Integrated Systems Architectures

### Design of a RISC-V-lite processor Description

#### 1 Problem description

In this lab you have to design in VHDL or SystemVerilog a RISC-V-lite processor with 5 pipeline stages. The instructions supported by the RISC-V-lite processor are a subset of the whole RV32I: the ones required to execute the *maxx* application as discussed in Section 2. The RV32I has a 32-bit fixed-width instruction set, where instructions are organized in six classes (R, I, S, B, U and J). The full list of instructions is summarized in the RISC-V reference data card. More details can be found in the RISC-V Instruction Set Manual, Volume I: User-Level/Unprivileged ISA (available at <https://riscv.org/specifications/>) where you can find the detail of opcode values and coding for each RV32I base instruction.

Refer to the slides of the course (Part 3-A Processor architecture) for the details about instructions execution, being aware that the description is not showing the complete architecture to support all the instructions.

#### 2 Compiling the source application

The application you have to run on your RISC-V-lite processor is available on portale as a C model made of 3 files:

1. main.c
2. maxx.h
3. maxx.c

In order to obtain the sequence of instructions to execute, we rely on *gcc*. It is worth noting that, to generate the executable file, the compiler requires informations about the memory organization (linker script) and a preamble to initialize registers, such as the stack pointer-*sp* (*crt0* file). In this lab we rely on very basic linker script and *crt0*<sup>1</sup>.

The simple *crt0* file (*crt0.s*) defines the `_start` symbol, which represents the address of the first instruction to execute (in this example the first instruction in the instruction memory). Then, it loads the global pointer and stack pointer registers (*gp* and *sp*). Finally, it calls the *main* function with jump-and-link instruction (*jal*) and the execution terminates with an infinite loop (*j el*).

**Note** Depending on the

```
.section .init, "ax"
.global _start
_start:
    .cfi_startproc
    .cfi_undefined ra
    .option push
    .option norelax
    la gp, __global_pointer$
    .option pop
    la sp, __stack_top
    add s0, sp, zero
    jal ra, main
el:
    j el
    .cfi_endproc
.end
```

The linker script contains the memory organization: a ROM for the code, which first address is 0x00400000 and a RAM for the data, which first address is 0x10010000. The global pointer is set to 0x10008000 and the stack pointer is placed at address 0x7ffeffc. Then, each segment of the program is assigned to the correct memory type, namely *.init* and *.text* to the ROM and *.data*, *.sdata* and *.bss* to the RAM<sup>2</sup>.

```
OUTPUTFORMAT("elf32-littleriscv", "elf32-littleriscv",
              "elf32-littleriscv")
OUTPUTARCH(riscv)
```

```
MEMORY
{
```

---

<sup>1</sup>Linker script and *crt0* are inspired by the ones you can find at <https://twilco.github.io/riscv-from-scratch/2019/04/27/riscv-from-scratch-2.html>

<sup>2</sup>Note that the instruction memory is a logic ROM, but physically it is a RAM as it needs to be loaded with the code to be executed.

```

/* ISA lab begin */
ROM (rx) : ORIGIN = 0x00400000, LENGTH = 64k
RAM (rwx) : ORIGIN = 0x10010000, LENGTH = 4000M
/* ISA lab end */
}

ENTRY(_start)

SECTIONS
{
/* Read-only sections, merged into text segment: */
PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x10000));
. = SEGMENT_START("text-segment", 0x10000) + SIZEOF_HEADERS;
/* ISA lab begin */
PROVIDE(__stack_top = 0x7ffefffc);
/* ISA lab end */
.init :
{
KEEP (*(SORT_NONE(.init)))
} > ROM
.text :
{
*(.text.unlikely .text.*_unlikely .text.unlikely.*)
*(.text.exit .text.exit.*)
*(.text.startup .text.startup.*)
*(.text.hot .text.hot.*)
*(.text.stub .text.*.gnu.linkonce.t.*)
/* .gnu.warning sections are handled specially by elf32.em. */
*(.gnu.warning)
} > ROM

PROVIDE (__etext = .);
PROVIDE (_etext = .);
PROVIDE (etext = .);
.data :
{
*(.data .data.* .gnu.linkonce.d.*)
SORT(CONSTRUCTORS)
} > RAM
.data1 : { *(.data1) } > RAM
.got : { *(.got.plt) *(.igot.plt) *(.got) *(.igot) } > RAM
/* We want the small data sections together, so single-instruction offsets
can access them all, and initialized data all before uninitialized, so
we can shorten the on-disk segment size. */
.sdata :
{
/* __global_pointer$ = . + 0x800; */
/* ISA lab begin */
__global_pointer$ = 0x10008000;
/* ISA lab end */
*(.srodata.cst16) *(.srodata.cst8) *(.srodata.cst4) *(.srodata.cst2)
*(.srodata .srodata.*)
*(.sdata .sdata.* .gnu.linkonce.s.*)
} > RAM
.edata = .; PROVIDE (edata = .);
. = .;
__bss_start = .;
.sbss :
{
*(.dynsbss)

```

```

        *(.sbss .sbss.* .gnu.linkonce.sb.*)
        *(.scommon)
    } > RAM
    .bss
    {
        *(.dynbss)
        *(.bss .bss.* .gnu.linkonce.b.*)
        *(COMMON)
        /* Align here to ensure that the .bss section occupies space up to
        _end. Align after .bss to ensure correct alignment even if the
        .bss section disappears because there are no input sections.
        FIXME: Why do we need it? When there is no .bss section, we don't
        pad the .data section. */
        . = ALIGN(. != 0 ? 32 / 8 : 1);
    } > RAM
    . = ALIGN(32 / 8);
    . = SEGMENT_START("ldata-segment", .);
    . = ALIGN(32 / 8);
    _end = .; PROVIDE (end = .);
}

```

To compile the source code we call the RISC-V version of *gcc*, that is `riscv32-unknown-elf-gcc` and we specify several parameters, such as the architecture type (RV32I), the optimization level (O3), the absence of external libraries, the linker script and the `crt0` file. This generates the binary executable file in elf format.

Once we have the executable with need to extract:

1. the sequence of instructions with the correct corresponding address;
2. the data

This can be done by relying on a disassembler (*objdump*) able to convert the binary executable file into a text file. In particular, with proper flags one can obtain the content of the program segments and of data segments.

To ease the procedure we provide you with a *makefile* which automatically extracts code and data from the binary executable file by simply issueing:

```

> make clean

> make all

```

This generates two files: `main.hex` and `data.hex`.

Let's consider a simple C program where the *main* calls a function which performs `c=a+b` with `a` and `b` being the first and the second element of a two position array, named `v` (`a=v[0]=5` and `b=v[1]=7`).

Note: this is only an example, your application is *maxx*. The file named `main.hex` looks like the following.

```
main:      file format elf32-littleriscv
```

Disassembly of section `.init`:

```
00400000 <_start>:
  400000:      1fc18197          auipc    gp,0x1fc18
  400004:      00018193          mv       gp,gp
  400008:      7fbff117          auipc    sp,0x7fbff
  40000c:      ff410113          addi     sp,sp,-12 # 7ffefffc <__stack_top>
  400010:      00010433          add      s0,sp,zero
  400014:      008000ef          jal      ra,40001c <main>

00400018 <el>:
  400018:      0000006f          j        400018 <el>
```

Disassembly of section `.text`:

```
0040001c <main>:
  40001c:      10010537          lui      a0,0x10010
  400020:      ff010113          addi     sp,sp,-16
  400024:      00050513          mv       a0,a0
  400028:      00112623          sw       ra,12(sp)
  40002c:      01c000ef          jal      ra,400048 <sum>
  400030:      00c12083          lw       ra,12(sp)
  400034:      100107b7          lui      a5,0x10010
  400038:      00a7a423          sw       a0,8(a5) # 10010008 <c>
  40003c:      00000513          li       a0,0
  400040:      01010113          addi     sp,sp,16
  400044:      00008067          ret

00400048 <sum>:
  400048:      00052703          lw       a4,0(a0) # 10010000 <v>
  40004c:      00452503          lw       a0,4(a0)
  400050:      00a70533          add      a0,a4,a0
  400054:      00008067          ret
```

As it can be observed the code begins with the `_start` label followed by three main columns: the memory address, the instruction in hexadecimal form and the corresponding assembly instruction. The first part is the stating portion of the `crt0`. Then, there is a `jal` instruction to the `main` and when the program returns there is the infinite loop (ending portion of the `crt0`). The `main` program prepares the stack pointer (`sp`) and calls the `sum` routine, which loads `v[0]` and `v[1]` and performs the addition.

The file named `data.hex` looks like the following.

```
main:      file format elf32-littleriscv
```

Disassembly of section `.data`:

10010000	<v>:	
10010000:	0005	c.addi zero,1
10010002:	0000	unimp
10010004:	0007	0x7
10010006:	0000	unimp

As it can be observed the disassembler tries to translate the data into instructions; thus, only the first two columns are relevant: the memory address and the data value in hexadecimal form.

### 3 RISC-V instruction simulator

In order to simulate and validate the architecture, a RISC-V simulator can be very helpful. RARS is a simple and effective RISC-V simulator written in Java

(<https://github.com/TheThirdOne/rars/releases>).

Given that you are working on a machine with Java already installed, RARS can be launched by double-clicking the .jar file or from the command line:

```
java -jar <file.jar>
```

Once you have an assembly file with the code you want to run, you simply open the file via RARS and assemble it (*Run*→*Assemble*). RARS is now ready to run. The instruction to be executed is highlighted in yellow and RARS shows you registers and memory content. The tab *Settings*→*Memory configuration* shows you the current memory configuration, which is compatible with the configuration we specified via the linker script in Section 2.

Simulating an assembly file starting from the .hex files, is rather simple. To ease the process, you are provided with two scripts: *extract\_code.sh* and *extract\_data.sh*. As the name suggests, the former extracts from main.hex the binary executable in hexadecimal format and the corresponding assembly into two separate files. The latter extracts the data from data.hex by removing comments.

Thus, issuing:

```
> ./extract_code.sh main.hex main.txt main.asm
```

you obtain the binary executable in hexadecimal format (main.txt):

```
00400000<_start>:
400000: 1fc18197
400004: 00018193
```

```

400008: 7fbff117
40000c: ff410113
400010: 00010433
400014: 008000ef

```

```

00400018<el>:
400018: 0000006f

```

```

0040001c<main>:
40001c: 10010537
400020: ff010113
400024: 00050513
400028: 00112623
40002c: 01c000ef
400030: 00c12083
400034: 100107b7
400038: 00a7a423
40003c: 00000513
400040: 01010113
400044: 00008067

```

```

00400048<sum>:
400048: 00052703
40004c: 00452503
400050: 00a70533
400054: 00008067

```

and the assembly (main.asm):

```

section .text:

__start:
auipc gp,0x1fc18
mv gp,gp
auipc sp,0x7fbff
addi sp,sp,-12
add s0,sp,zero
jal ra,40001c <main>

el:
j 400018 <el>

main:
lui a0,0x10010
addi sp,sp,-16
mv a0,a0
sw ra,12(sp)
jal ra,400048 <sum>
lw ra,12(sp)
lui a5,0x10010
sw a0,8(a5)
li a0,0
addi sp,sp,16
ret

sum:
lw a4,0(a0)
lw a0,4(a0)
add a0,a4,a0
ret

```

If you want to adjust the .txt file to keep only the instructions by removing labels and addresses you can simply issue:

```
> cat main.txt | sed s/.*:// | sed /^[:space:]]*$/d > main.bin
```

to obtain `main.bin`, which is the content of the instruction memory.

Similarly, you can issue:

```
> ./extract_data.sh data.hex data.txt
```

to obtain

```
v:
0005
0000
0007
0000
```

The assembly file you can simulate with RARS is obtained by joining the .asm files with some minor editing. Indeed, the first section we put in the assembly file is `.data` with the content of `data.asm`, where we re-create 32-bit values by adding the keyword `.word`. Then, we declare the `.text` section, we define the `_start` symbol as global and we include the assembly instructions. The only further modification that is required is to adjust jump instructions such that the destination address becomes a label. In most of the cases the .asm file already provides the name of the label. In some cases you need to check with `main.hex` which instruction corresponds to the destination address and you have to add new labels.

```
.data
v:
.word 0x00000005
.word 0x00000007

.text
.globl _start

_start:
    auipc gp,0x1fc18
    mv gp,gp
    auipc sp,0x7fbff
    addi sp,sp,-12
    add s0,sp,zero
    jal ra,main

el:
    j el

main:
    lui a0,0x10010
```



```

    addi sp,sp,-16
    mv a0,a0
    sw ra,12(sp)
    jal ra,sum
    lw ra,12(sp)
    lui a5,0x10010
    sw a0,8(a5)
    li a0,0
    addi sp,sp,16
    ret

```

sum:

```

    lw a4,0(a0)
    lw a0,4(a0)
    add a0,a4,a0
    ret

```

## 4 Architecture, simulation and validation

Once you have the files explained in the previous sections, you can design the architecture. Note that the designed architecture must be able to run the code as is, so you cannot add/change/remove instructions, but you can add hardware features to the standard architecture (hazard detection, forwarding, branch prediction, ...) as your task is to run the application by maximizing the throughput (reducing stalls and pipe flush). Design your pipeline as a Harvard architecture (separate physical memory for instructions and data) such that you are not making any assumption on the number of clock cycles required to read a value from a memory (either instructions or data).

### 4.1 Step 1

To decouple the pipeline from the memories you need to equip your architecture with a simple request/ready (*req/rdy*) protocol<sup>3</sup>. In this step, keep the memories out of your design. The protocol signals are shown in Table 1.

The protocol relies on:

1. Address phase: a transaction starts in the cycle in which **proc\_req** goes high and completes on the rising clock edge when both **proc\_req** and **mem\_rdy** are high.

---

<sup>3</sup>Despite being general, this protocol is inspired by the Open Bus Interface (OBI) proposed by the OpenHW group (<https://github.com/openhwgroup/obi>) and part of the description is taken from the OBI documentation

Tabella 1: *req/ack* protocol signals

<b>proc_req</b>	out	Request signal, must stay high until <b>mem_rdy</b> is high for one cycle
<b>mem_rdy</b>	in	Ready signal, it states that the request has been accepted.
<b>addr</b>	out	Address, sent together with <b>proc_req</b>
<b>wdata</b>	out	Data to be written to memory. Sent together with <b>proc_req</b> (for data memory only).
<b>rdata</b>	in	Data read from the memory. Only valid when <b>valid</b> is high.
<b>valid</b>	in	This signal will be high for exactly one cycle per request to signal the end of the response phase for both read and write transactions.
<b>we</b>	out	Write Enable, high for writes, low for reads. Sent together with <b>proc_req</b> .

2. Response phase: after a valid address phase, the memory indicates the validity of its response phase signals by setting **valid** high.

In detail: the processor provides a valid address on **addr**, control information for reading/writing on **we**, (as well as write data on **wdata** in case of a writing transaction) and sets **proc\_req** high. The memory sets **mem\_rdy** high as soon as it is ready to serve the request. This may happen at any time, even before the request was sent. After a request has been granted, the address phase signals (**addr**, **we** and **wdata**) may be changed in the next cycle by the processor as the memory is assumed to already have processed and stored that information. After accepting a request, the memory answers with a **valid** set high if **rdata** is valid. This may happen one or more cycles after the request has been accepted. Note that **valid** must also be set high to signal the end of the response phase for a write transaction (although the **rdata** has no meaning in that case). When multiple requests have been accepted, it is assumed that the memory requests will be kept in-order and one **valid** will be signalled for each of them, in the order they were issued. Fig. 1, 2, 3 and 4 show examples of basic writing, basic reading, back to back and slow memory transactions respectively.

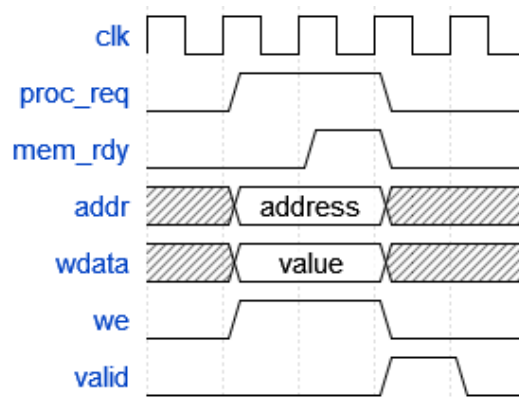


Figure 1: basic writing.

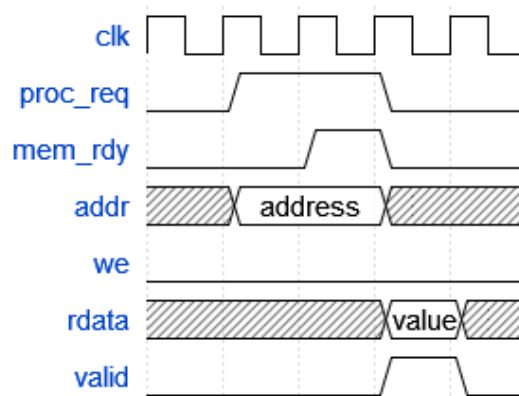


Figure 2: basic reading.

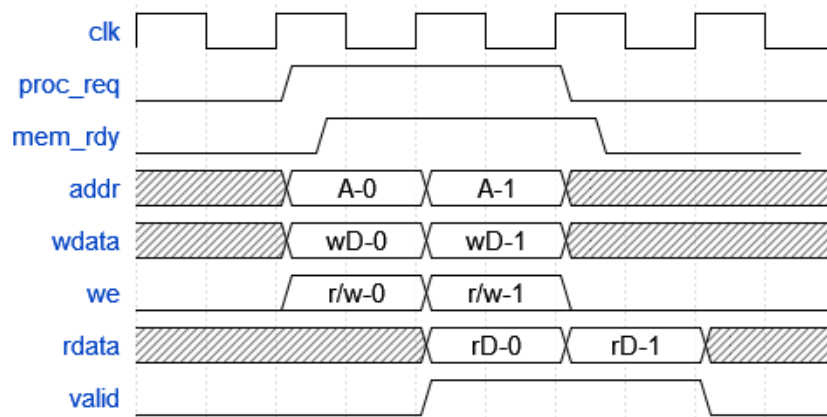


Figure 3: back to back transaction.

Tabella 2: *mem\_wrap\_fake* interface

CLK	clock signal
RSTn	asynchronous (active low)
PROC_REQ	request signal
MEM_RDY	ready signal
ADDR	address bus (32 bit)
WE	write enable
WDATA	input data bus (32 bit)
RDATA	output data bus (32 bit)
VALID	valid signal

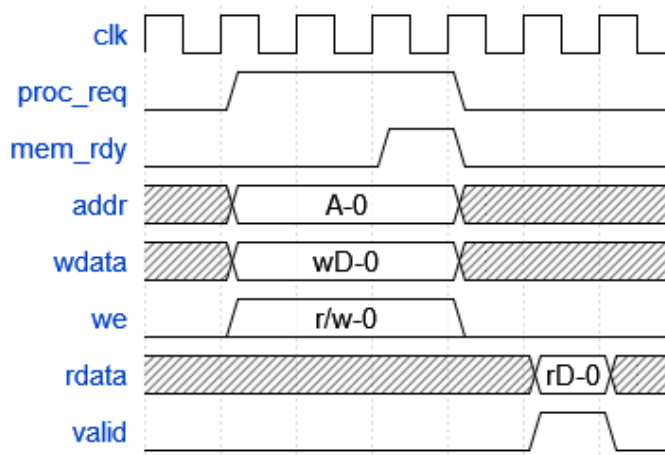


Figure 4: slow memory transaction.

To ease the interaction with the pipeline and the memory, you may consider including in the architecture a fetcher and a Load-Store-Unit (LSU), which implement the protocol described above toward the instruction and data memories, respectively.

Use in the testbench the fake wrap module available on portale to test your implementation of the protocol (processor-side). This module (*mem\_wrap\_fake*) emulates the behaviour of a memory with variable latency by using pseudorandom generators to drive **MEM\_RDY** and **VALID**. The interface of the module is listed in Table 2. The module has also three parameters to be used for customizing the instantiation:

- **CONTENT\_TYPE** (0 code, 1 data) to model the behaviour of the module as code (instruction memory) or data (data memory).
- **tco**, clock to output delay to avoid output signals producing timing violations when simulating a netlist.

- `tpd`, combinational delay to avoid output signals producing timing violations when simulating a netlist.

It assumes you place in the directory where you run the simulation (usually `sim`) the following four files:

- `text_init.hex`, it contains the address of the first instruction in the instruction memory (e.g. 00400000).
- `main.bin`, it contains the sequence of instructions (in hexadecimal) to be executed (e.g. 1fc18197, 00018193, ...).
- `data_init.hex`, it contains the address of the first data in the data memory (e.g. 10010000).
- `data.bin`, it contains the sequence of data (in hexadecimal) (e.g. 00000005, 00000007).

The example comes with a testbench where the time resolution is 1 ps, the clock period is 10 ns and `tco` and `tpd` are 1 ps. The fake wrap module is configured to act as an instruction memory (you can change the `cCONTENT_TYPE` local parameter to configure it as a data memory). Then, the test bench includes a module to generate the clock and the reset, a module to stimulate the fake wrap module and module to write on a file the value on `RDATA` when `VALID` is high. Note that when `VALID` is low and when the operation is a writing one, the value on `RDATA` is undefined. The module to stimulate the fake wrap module is named *request generator* and comes in two versions. The first version (`req_gen`) relies on a FSM that periodically generates a request and waits for the ready from the fake wrap module. The second version (`req_gen_fast`) has the request signal always active, so the fake wrap module answers as soon as possible to the continuous requests.

## 4.2 Step 2

Add to your design an instruction and a data memory by exploiting the already provided SSRAM macro<sup>4</sup>. Insert one or more instances of the SSRAMs into a wrapper such that they expose toward your

---

<sup>4</sup>This macro has been generated using the OpenRAM tool <https://openram.org/>

Tabella 3: SSRAM pinout

clk0	clock signal
csb0	chip select (active low)
web0	write enable (active low)
addr0	address bus (10 bit)
din0	input data bus (32 bit)
dout0	output data bus (32 bit)

processor the interface defined by the protocol descibed in Section 4.1.

The SSRAM is a single-port, synchronous (both in reading and writing) memory with separate input and output ports. Note that in ASIC designs, SSRAM macros are similar to standard cells, but instead of being inferred by the logic synthesizer (as for the logic) they have to be manually instantiated by the designer. You can refer to the available design example to see how to instantiate an SSRAM in your HDL and how to synthesize it (remember to modify the .synopsys\_dc.setup file to include the .db file of the SSRAM). Also ASIC SSRAMs cannot be initialized at design time; thus, you have to include in your design and in the testbench a mechanism (e.g. a multiplexer controlled by the testbench) to first write instructions and data in the SSRAMs and then start the execution.

Table 3 shows the pinout of the macro and Fig. 5 and 6 the timing of simple writing and reading operations.

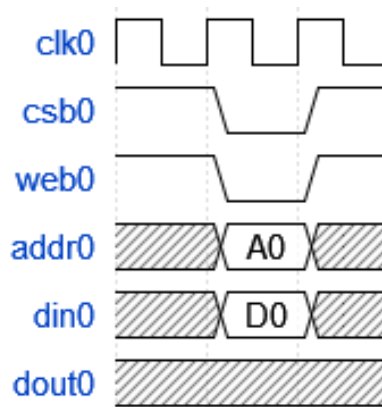


Figure 5: SSRAM writing.

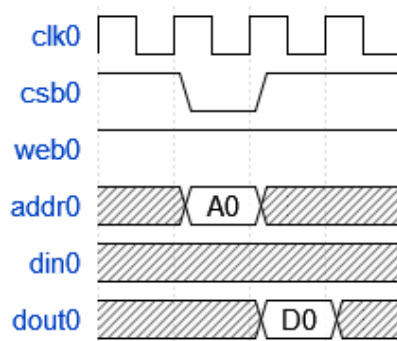


Figure 6: SSRAM reading.

**Note** The macro has some limitations, in particular it does not include sdf file support, so you cannot make power estimation, but you can synthesize it and check the netlist behaviour via simulation. Moreover, the current version has no working lef file, so it does not permit to complete the design flow with correct place and route.