HDU-ITMO Joint Institute
杭州电子科技大学圣光机联合学院

Computer Systems Design

Laboratory work 3

Integration of programmable cores in computer system

Author:

Alexander Antonov

Associate Professor

Lecturer:

Ivan Lukashov

lukashov@itmo.ru

2024

Contents

1. Objectives

- Students can implement embedded processor cores in computer system designs
- Students can integrate custom IP in computer system designs
- Students can integrate custom logic with embedded processor cores using system bus interface
- Students can build and implement embedded software for custom designs
- Students understand how to use Xilinx Vivado Design Suite to implement and debug processor-enabled designs

2. Overview

This laboratory work covers software (firmware) based implementation of functionality using embedded programmable processor core. Using programable processors, through having lower efficiency compared to direct hardware implementation, offers multiple virtues: simplification of programming, faster compilation, software update capability, better availability of engineers, etc. In this Lab, basic open-source MCU with RISC-V central processor unit (CPU) core will be used. RISC-V is an open instruction set architecture being widely used both in academia and industry in recent years.

3. Prerequisites

Prerequisites are the same as for Lab 1 and Lab 2, with two additional points [3, 4]:

1. Xilinx Vivado 2019.1 HLx Edition (free for target board, available at https://www.xilinx.com/support/download.html).
2. ActiveCore baseline distribution (available at https://github.com/AntonovAlexander/activecore)
3. Generated RISC-V CPU HDL sources
4. Working RISC-V GNU toolchain (available at https://github.com/riscv/riscv-gnu-toolchain)

   NOTE: pre-built binaries for various hosts can be downloaded from https://www.sifive.com/software. Do not forget to update `PATH` variable after downloading. Consider using Cygwin (with `make` utility) or WSL for RISC-V software compilation in Windows hosts.
5. (for FPGA prototyping) Digilent Nexys 4 DDR FPGA board (https://store.digilentinc.com/nexys-4-ddr-artix-7-fpga-trainer-board-recommended-for-ece-curriculum/)
6. (for FPGA prototyping) working Python 3 installation with `pyserial` package

4. Task

1. Examine Sigma MCU baseline project
2. (if FPGA board available) Implement Sigma MCU in FPGA device and verify correctness of the baseline
3. Write software implementation of functionality for CPU according to your variant
4. Verify functional correctness in simulation
5. Implement the design and collect metrics of the implementation
6. (if FPGA board available) Upload your program to Sigma MCU and make sure it works correctly
7. Analyze performance of implementations
8. Package your solution and submit to the teacher's email

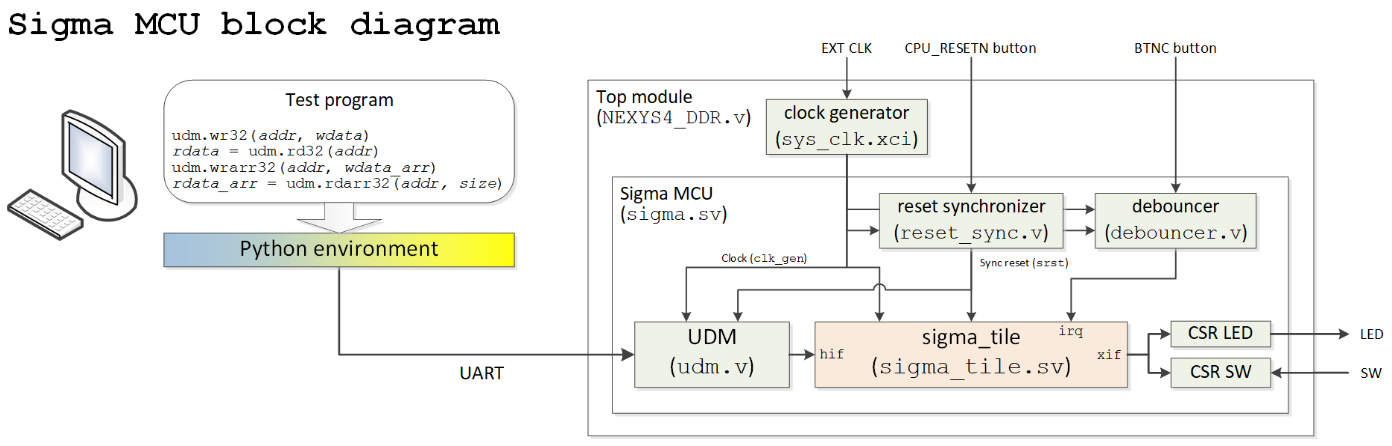
5. GUIDANCE

Detailed guidance will be provided using the example of a program with the same functionality as modules designed in Lab 1 and Lab 2.

1. Examine Sigma MCU baseline project

Sigma is a basic microcontroller unit (MCU) soft core consisting of `sigma_tile` processing module, UDM and general-purpose input/output (GPIO) controller. GPIO controller is mapped on LEDs and switches on FPGA board.

Block diagram of Sigma MCU is shown in Figure 1.



```
udm.wr32(addr, wdata)
rdata = udm.rd32(addr)
udm.wrarr32(addr, wdata_arr)
rdata_arr = udm.rdarr32(addr, size)
```

Figure 1 Sigma MCU block diagram

`Sigma_tile` module contains embedded CPU core with RISC-V (RV32IM) ISA, tightly coupled on-chip RAM with single-cycle delay, several special-function registers (SFRs), Host InterFace (HIF), and eXpansion InterFace (XIF). Multiple `sigma_tile` modules can fit in a single FPGA device. HIF and XIF have the same bus protocol as UDM block (see Lab 1).

Block diagram of `sigma_tile` module is shown in Figure 2.

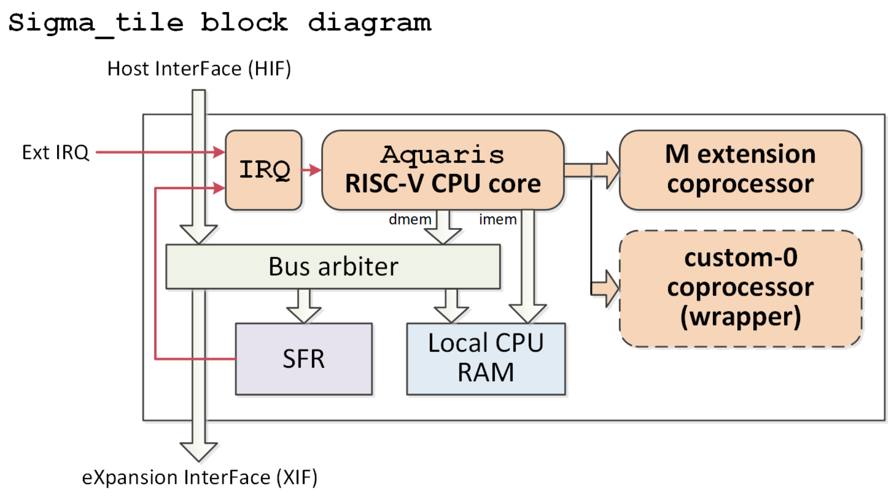
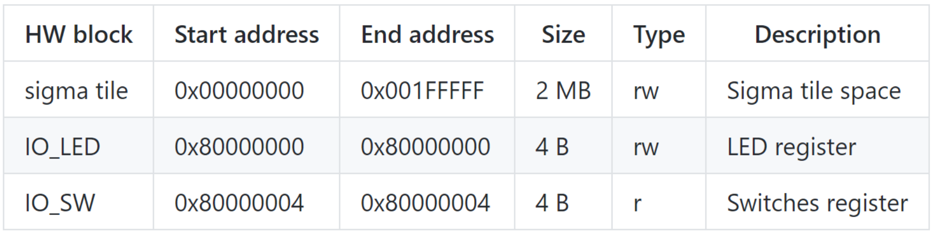


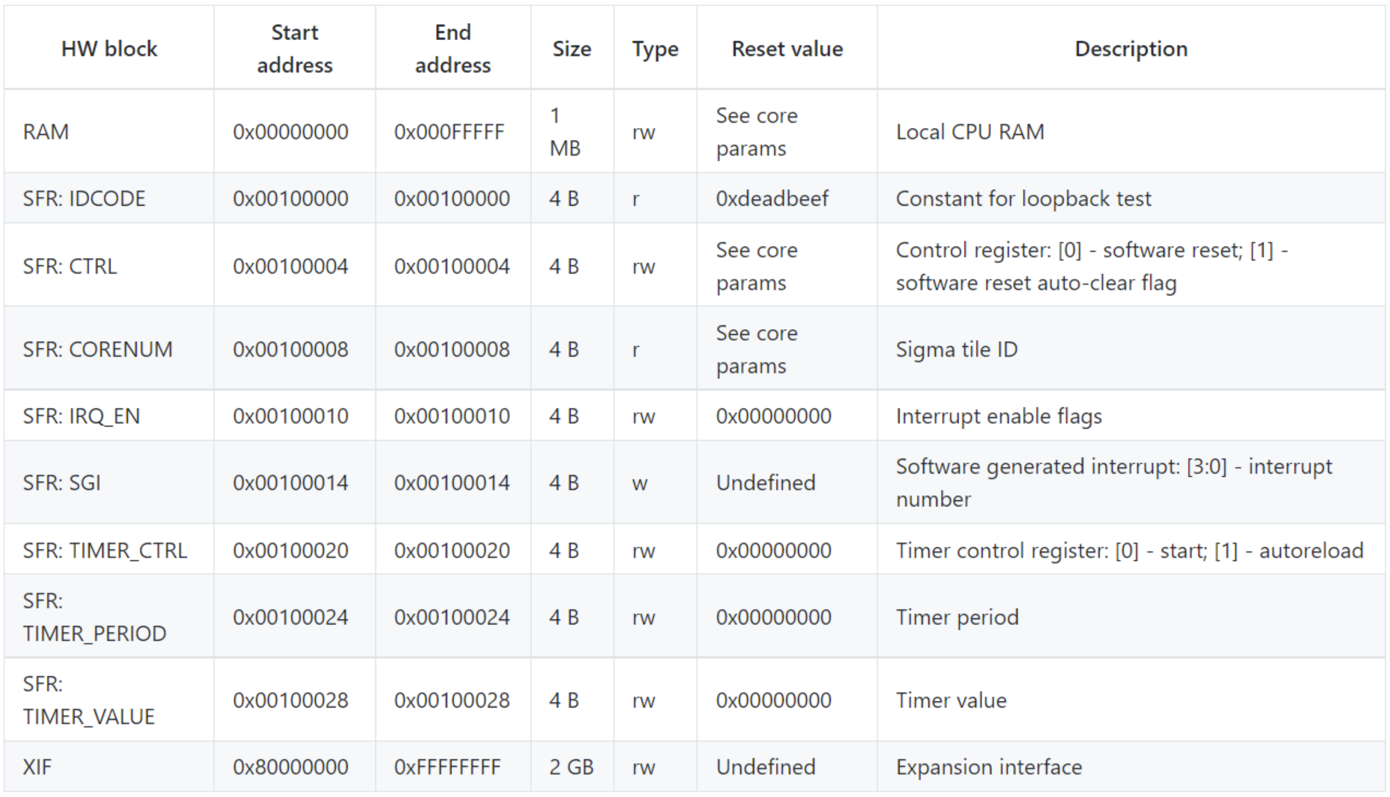
Figure 2 `sigma_tile` block diagram

Address maps of Sigma MCU and `sigma_tile` module are shown in Figure 3 and Figure 4 respectively. Memory maps are identical for UDM and CPU.



HW block	Start address	End address	Size	Type	Description
sigma tile	0x00000000	0x001FFFFF	2 MB	rw	Sigma tile space
IO_LED	0x80000000	0x80000000	4 B	rw	LED register
IO_SW	0x80000004	0x80000004	4 B	r	Switches register

Figure 3 Address map of Sigma MCU



HW block	Start address	End address	Size	Type	Reset value	Description
RAM	0x00000000	0x000FFFFF	1 MB	rw	See core params	Local CPU RAM
SFR: IDCODE	0x00100000	0x00100000	4 B	r	0xdeadbeef	Constant for loopback test
SFR: CTRL	0x00100004	0x00100004	4 B	rw	See core params	Control register: [0] - software reset; [1] - software reset auto-clear flag
SFR: CORENUM	0x00100008	0x00100008	4 B	r	See core params	Sigma tile ID
SFR: IRQ_EN	0x00100010	0x00100010	4 B	rw	0x00000000	Interrupt enable flags
SFR: SGI	0x00100014	0x00100014	4 B	w	Undefined	Software generated interrupt: [3:0] - interrupt number
SFR: TIMER_CTRL	0x00100020	0x00100020	4 B	rw	0x00000000	Timer control register: [0] - start; [1] - autoreload
SFR: TIMER_PERIOD	0x00100024	0x00100024	4 B	rw	0x00000000	Timer period
SFR: TIMER_VALUE	0x00100028	0x00100028	4 B	rw	0x00000000	Timer value
XIF	0x80000000	0xFFFFFFFF	2 GB	rw	Undefined	Expansion interface

Figure 4 Address map of `sigma_tile` module

NOTE: Only 4-byte aligned accesses are supported.

RISC-V CPU supports basic bare metal programming (base RV32I ISA, without FPU, MMU, etc). ActiveCore distribution provides six Sigma MCU projects with different CPU configurations (1-6 pipeline stages). As you learned in Lab 2, longer pipeline can operate on higher frequencies and have better performance, however, consuming more hardware resources and power.

The projects are located at: `activecore/designs/rtl/sigma/syn/syn_`***`X`***`stage/NEXYS4_DDR`

Unpack the `coregen` directory with generated CPU HDL sources (provided by the teacher) in the following directory:

`activecore/designs/rtl/sigma_tile/hw/riscv`

E.g. `riscv_5stage.sv` file should be located at:

`activecore/designs/rtl/sigma_tile/hw/riscv/coregen/riscv_5stage/sverilog`

Open `NEXYS4_DDR.xpr` file using Xilinx Vivado 2019.1.

2. (if FPGA board available) Implement Sigma MCU in FPGA device and verify correctness of the baseline

Go to `activecore/designs/rtl/sigma/sw/apps` directory and build CPU software using `make` command.

Implement the design, generate the bitstream and upload it to FPGA. LEDs should start blinking with variable speed, depending on value on switches.

Find out the name of COM port associated with the board (`COM<number>` on Windows hosts or `tty<number>` on Linux hosts). Open `hw_test.py` test Python script and fill the correct COM port name in line 14:

```
    udm = udm("<correct COM port name>", 921600)
```

Run CPU compliance tests using `hw_test_compliance.py` Python script. The script will upload 52 test programs for CPU and verify correctness of their operation. The last line of console output should be:

```
    Total tests PASSED:  52 , FAILED:  0
```

Run CPU application tests using `hw_test_bechmarks.py` Python script. The script will upload 9 test programs for CPU and verify correctness of their operation. The last line of console output should be:

```
    Total tests PASSED:  9 , FAILED:  0
```

You can type `help(sigma)` and `help(sigma_tile)` in Python console for full API reference of Sigma MCU and `sigma_tile` module respectively.

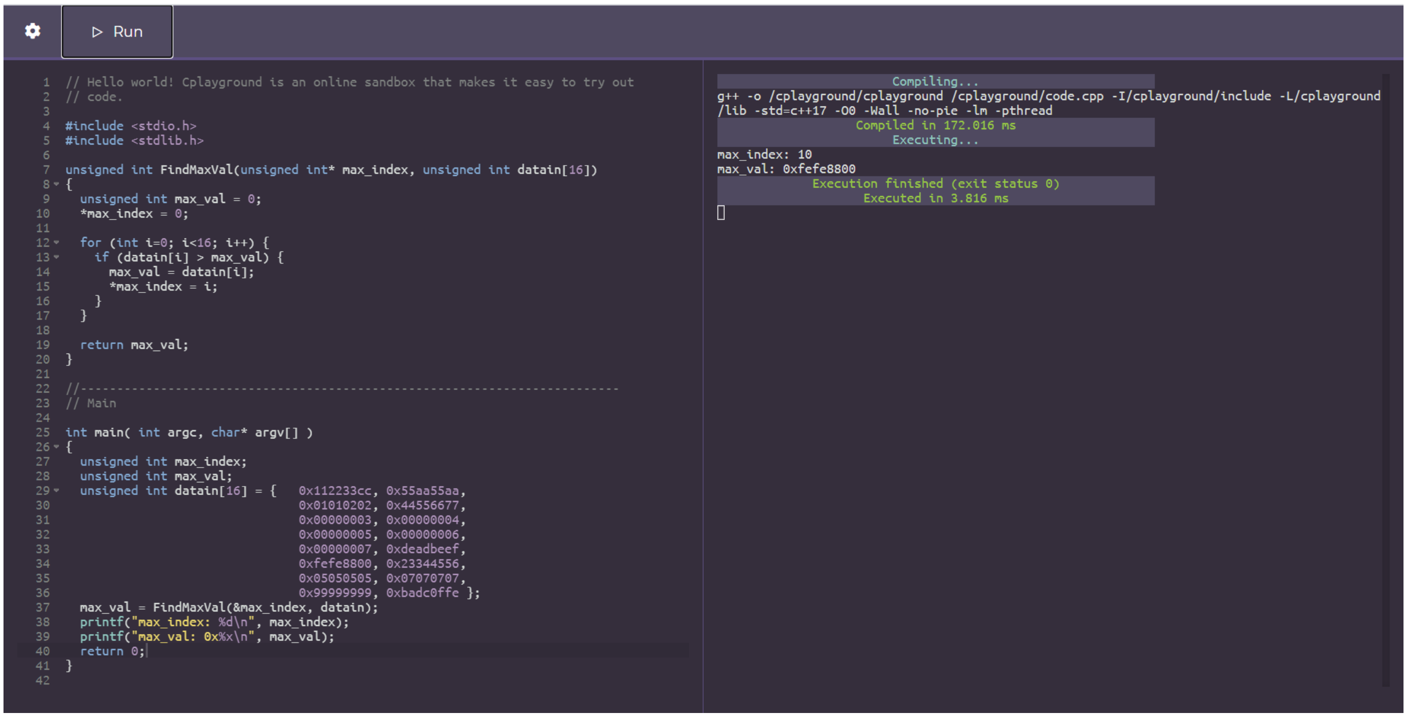
3. Write software implementation of functionality for CPU according to your variant

Sigma MCU distribution provides several demo applications that can be used as reference (see Table 1).

Demo application	**Description**
`heartbeat_variable`	A counter that is output to LED register. The period is continuously read from Switches register. Period is implemented as CPU busy waiting.
`irq_counter`	A counter that is output to LED register. Increment is triggered by interrupt 3 that is mapped on button on FPGA board.
`dhrystone`	Dhrystone synthetic benchmark
`median`	Three-element median filter operating on 400-element array of integers.
`mul_sw`	Software multiplication of two integers producing an integer.
`qsort`	Quick sort operating on 1024-element array of integers.
`rsort`	Radix sort operating on 1024-element array of integers.
`crc32`	CRC32 hash calculation
`md5`	MD5 hash calculation
`timer_test`	A counter that is output to LED register. Utilizes the timer to count the period. The period is read from Switches register on reset.
`bootloader`	Bootloader of programs in binary (ELF) format from the memory buffer

Table 1 Demo applications provided in Sigma MCU distribution

Write software implementation of your functionality and check its correctness. You can use your standard local `gcc` installation or an online service (e.g. https://cplayground.com/ or https://ideone.com/) for this task. Test result for our example is shown in Listing 1.



```
// Hello world! Cplayground is an online sandbox that makes it easy to try out
// code.

#include <stdio.h>
#include <stdlib.h>

unsigned int FindMaxVal(unsigned int* max_index, unsigned int datain[16])
{
  unsigned int max_val = 0;
  *max_index = 0;

  for (int i=0; i<16; i++) {
    if (datain[i] > max_val) {
      max_val = datain[i];
      *max_index = i;
    }
  }

  return max_val;
}

//------------------------------------------------------------------------
// Main

int main( int argc, char* argv[] )
{
  unsigned int max_index;
  unsigned int max_val;
  unsigned int datain[16] = {   0x112233cc, 0x55aa55aa,
                                0x01010202, 0x44556677,
                                0x00000003, 0x00000004,
                                0x00000005, 0x00000006,
                                0x00000007, 0xdeadbeef,
                                0xfefe8800, 0x23344556,
                                0x05050505, 0x07070707,
                                0x99999999, 0xbadc0ffe };
  max_val = FindMaxVal(&max_index, datain);
  printf("max_index: %d\n", max_index);
  printf("max_val: 0x%x\n", max_val);
  return 0;
}
```

```
                    Compiling...
g++ -o /cplayground/cplayground /cplayground/code.cpp -I/cplayground/include -L/cplayground
/lib -std=c++17 -O0 -Wall -no-pie -lm -pthread
               Compiled in 172.016 ms
                    Executing...
max_index: 10
max_val: 0xfefe8800
         Execution finished (exit status 0)
                Executed in 3.816 ms
```

Listing 1 Testing software implementation using cplayground.com

Go to `activecore/designs/rtl/sigma/sw/apps` directory and add new directory for your software. In our example, the new directory is called `findmaxval`.

Create new C source file in the new directory. In our example, the file is called `findmaxval.c`. Write your program in this file. Source code for the example program in shown in Listing 2:

```
#define IO_LED            (*(volatile unsigned int *)(0x80000000))
#define IO_SW             (*(volatile unsigned int *)(0x80000004))

unsigned int FindMaxVal(unsigned int* max_index, unsigned int datain[16])
{
  unsigned int max_val = 0;
  *max_index = 0;

  for (int i=0; i<16; i++) {
    if (datain[i] > max_val) {
      max_val = datain[i];
      *max_index = i;
    }
  }

  return max_val;
}

//------------------------------------------------------------------------
// Main

int main( int argc, char* argv[] )
{
  unsigned int max_index;
  unsigned int max_val;
  unsigned  int  datain[16]  =  {  0x112233cc,  0x55aa55aa,  0x01010202,  0x44556677,
0x00000003,  0x00000004,  0x00000005,  0x00000006,  0x00000007,  0xdeadbeef,  0xfefe8800,
0x23344556, 0x05050505, 0x07070707, 0x99999999, 0xbadc0ffe };
  IO_LED = 0x55aa55aa;
  max_val = FindMaxVal(&max_index, datain);
  IO_LED = max_index;
  IO_LED = max_val;
  while (1) {}        // infinite loop
}
```

Listing 2 C source code in `findmaxval.c`

NOTE: we have output `0x55aa55aa` value to LEDs to mark the end of startup sequence and start of the target function `FindMaxVal`. In the end of the program, we output `max_index` and `max_val` values and send CPU to infinite loop.

NOTE: since Sigma MCU does not have standard output, we use LEDs to output resulting values.

Prepare executable image for CPU. Open `Makefile` in `activecore/designs/rtl/sigma/sw/apps` directory and add the reference to the new directory in `bmarks` variable (added line is highlighted in cyan). Source code for the updated `bmarks` assignment is shown in Listing 3:

```
bmarks = \
  heartbeat_variable \
  median \
  qsort \
  rsort \
  irq_counter \
  findmaxval \
  <commented lines>
```

Listing 3 Source code of the updated `bmarks` assignment in Makefile

Call `make` command from `activecore/designs/rtl/sigma/sw/apps` directory to build the program image.

4. Verify functional correctness in simulation

Open the testbench file `activecore/designs/rtl/sigma/tb/riscv_tb.sv`, set up the CPU configuration, and make `mem_data` parameter of `sigma` instance reference to your program image. For our example, code updates are shown in Listing 4.

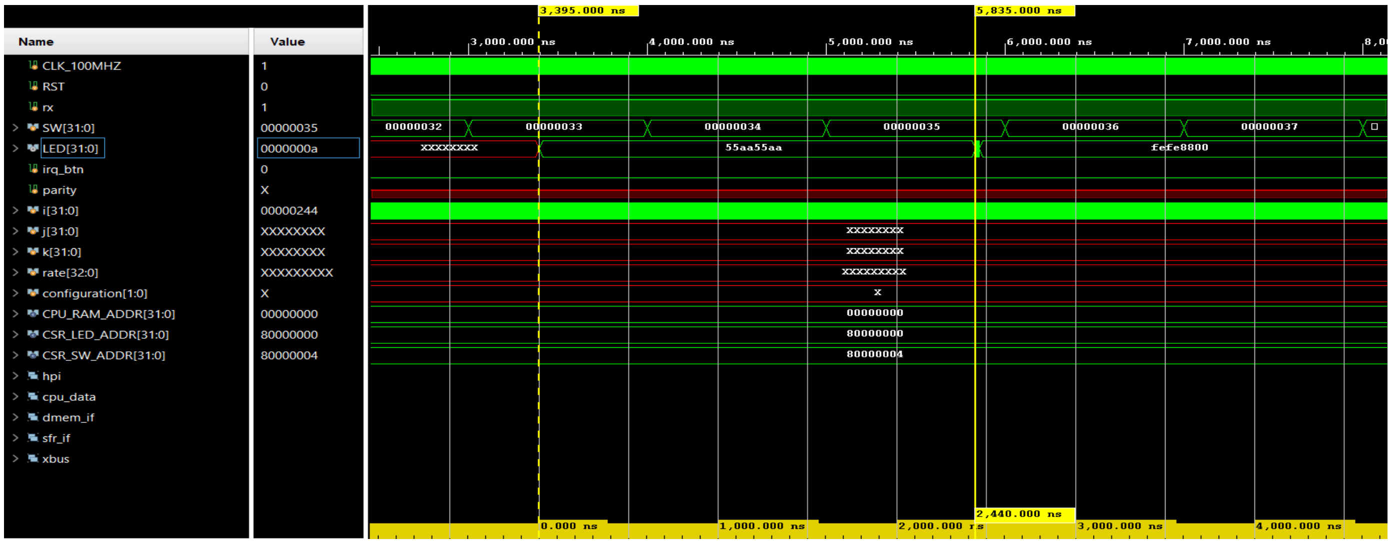
```
sigma
#(
  //.CPU("riscv_1stage")
  //.CPU("riscv_2stage")
  //.CPU("riscv_3stage")
  //.CPU("riscv_4stage")
  .CPU("riscv_5stage")
  //.CPU("riscv_6stage")

  , .UDM_RTX_EXTERNAL_OVERRIDE("YES")
  , .DEBOUNCER_FACTOR_POW(2)
  , .delay_test_flag(0)

  , .mem_init_type("elf")
  ,
.mem_init_data("<PATH_TO_ACTIVECORE>/activecore/designs/rtl/sigma/sw/apps/findmaxval.
riscv")
  , .mem_size(8192)
) sigma
(
  .clk_i(CLK_100MHZ)
  , .arst_i(RST)
  , .irq_btn_i(irq_btn)
  , .rx_i(rx)
  //, .tx_o()
  , .gpio_bi(SW)
  , .gpio_bo(LED)
);
```

Listing 4 Updated module instantiation in `riscv_tb.sv` testbench

Simulation waveform for 5-stage CPU configuration is shown in Figure 5.



Name	Value
CLK_100MHZ	1
RST	0
rx	1
SW[31:0]	00000035
LED[31:0]	0000000a
irq_btn	0
parity	X
i[31:0]	00000244
j[31:0]	XXXXXXXX
k[31:0]	XXXXXXXX
rate[32:0]	XXXXXXXXX
configuration[1:0]	X
CPU_RAM_ADDR[31:0]	00000000
CSR_LED_ADDR[31:0]	80000000
CSR_SW_ADDR[31:0]	80000004
hpi	
cpu_data	
dmem_if	
sfr_if	
xbus	

Figure 5 Simulation waveform

The values on LEDs are correct, the program works as intended.

NOTE: if resulting values do not appear in simulation, check the program is placed in RAM. Compare first several values of `/riscv_tb/sigma/sigma_tile/ram/ram_dual/ram` array to the program binary.

Measure the number of clock cycles needed to execute the program by various CPU configurations. To switch CPU configurations for simulation, uncomment corresponding `CPU` parameter of `sigma` instance in `riscv_tb.sv` testbench from corresponding Vivado project. In the testbench, 100 MHz clock is generated, so 2440 ns equals 244 clock cycles. For our example, results are summarized in Table 2.

CPU configuration	Latency, clock cycles
riscv_1stage	206
riscv_2stage	190
riscv_3stage	217
riscv_4stage	244
riscv_5stage	244
riscv_6stage	271

Table 2 Performance (in clock cycles) of software implementations based on various CPU configurations

5. Implement the design and collect metrics of the implementation

Characteristics of provided `sigma_tile` configurations are shown in Table 3:

CPU configuration	Frequency, MHz	LUTs	FFs
riscv_1stage	75	2504	1706
riscv_2stage	70	1966	1322
riscv_3stage	100	1929	1474
riscv_4stage	140	2330	1741
riscv_5stage	160	2195	1782
riscv_6stage	180	2253	1884

Table 3 Characteristics of provided `sigma_tile` implementations

6. (if FPGA board available) Upload your program to Sigma MCU and make sure it works correctly

To upload your program, add `loadelf` command to the end of `hw_test.py` script. For our example, the line is the following:

```
sigma.tile.loadelf('<PATH_TO_ACTIVECORE>/activecore/designs/rtl/sigma/sw/apps/findmaxva
l.riscv')
```

In our example, the LEDs show `0x8800` (16 least significant bits of `0xfefe8800` value). The program works as intended.

7. Analyze performance of implementations

Now we can analyze the absolute performance values of target functionality implementations based on various CPU configurations. To get these values for each implementation in ns, multiply latency in clock cycles by 10 (10 ns clock period in simulation) and divide by simulation/actual frequency ratio (i.e. multiply latency in clock cycles by 1000 and divide by actual frequency in MHz). For our example, these values are shown in Table 4.

CPU configuration	Latency, ns
riscv_1stage	2747
riscv_2stage	2714
riscv_3stage	2170
riscv_4stage	1743
riscv_5stage	1525
riscv_6stage	1506

Table 4 Absolute performance of target functionality implementations based on various CPU configurations

8. Package your solution and submit to the teacher's email

The package content is equal to Lab 1 and Lab 2, but should also include the program for Sigma MCU (source and binary).

NOTE: if you have issues sending large attachments, consider deleting temporary files from the built projects in NEXYS4_DDR directories (all directories except NEXYS4_DDR.srcs).

6. VARIANTS

Same as for Lab 1.