ActiveCore

Laboratory work manual

Using Sigma MCU in FPGA designs

Author:

Alexander Antonov

antonov.alex.alex@gmail.com

2021

Contents

1. TARGET SKILLS

- Implementation of Sigma MCU in hardware projects
- Building and implementation of embedded software for Sigma MCU
- Choosing optimal CPU configuration of Sigma MCU
- Integration of custom logic with Sigma MCU using its expansion interface
- Using Xilinx FPGA and Vivado Design Suite for implementation of Sigma MCU

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

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 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 eCPU 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. (optional) Integrate any UDM-compatible module in Sigma MCU

5. GUIDANCE

Detailed guidance will be provided using the example of a program that searches for the maximum value in 16-element array and returns this value and its index in the array.

1. Examine Sigma MCU baseline project

Sigma MCU is a basic microcontroller unit 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 located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/sigma/doc/sigma_struct.png

`Sigma_tile` module contains embedded CPU (eCPU) core with RISC-V ISA, tightly coupled on-chip RAM with single-cycle delay, interrupt controller, timer, 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. Address maps are identical for UDM and eCPU. Working with UDM can be learned from the corresponding lab work:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/udm/doc/udm_lab_manual.pdf

Block diagram of `sigma_tile` module is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/sigma_tile/doc/sigma_tile_struct.png

Address map of Sigma MCU is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/sigma/doc/sigma_addr_map.md

Address map of `sigma_tile` module is located at:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/sigma_tile/doc/sigma_tile_addr_map.md

Pipeline structures of various RISC-V eCPU configurations can be found here:

https://github.com/AntonovAlexander/activecore/blob/master/designs/rtl/sigma_tile/doc/aquaris_pipeline_structs

RISC-V eCPU supports basic bare metal programming (base RV32I ISA, without FPU, MMU, etc). ActiveCore distribution provides six Sigma MCU projects with different eCPU configurations (1-6 pipeline stages). 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`

Generate RISC-V eCPU HDL sources or unpack the provided `coregen` archive 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.

NOTE: avoid having non-English characters in project location path. Also, avoid very long project location path.

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

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

Implement the design, generate the bitstream and upload it to FPGA device. 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). Go one directory up, open `hw_test_bechmarks.py` test Python script and fill the correct COM port name in line 14:

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

Run eCPU compliance tests using `hw_test_compliance.py` Python script. The script will upload 37 test programs for eCPU and verify correctness of their operation. The last line of console output should be:

```
Total tests PASSED:  37 , FAILED:  0
```

Run eCPU application tests using `hw_test_bechmarks.py` Python script. The script will upload 9 test programs for eCPU 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 eCPU according to your variant

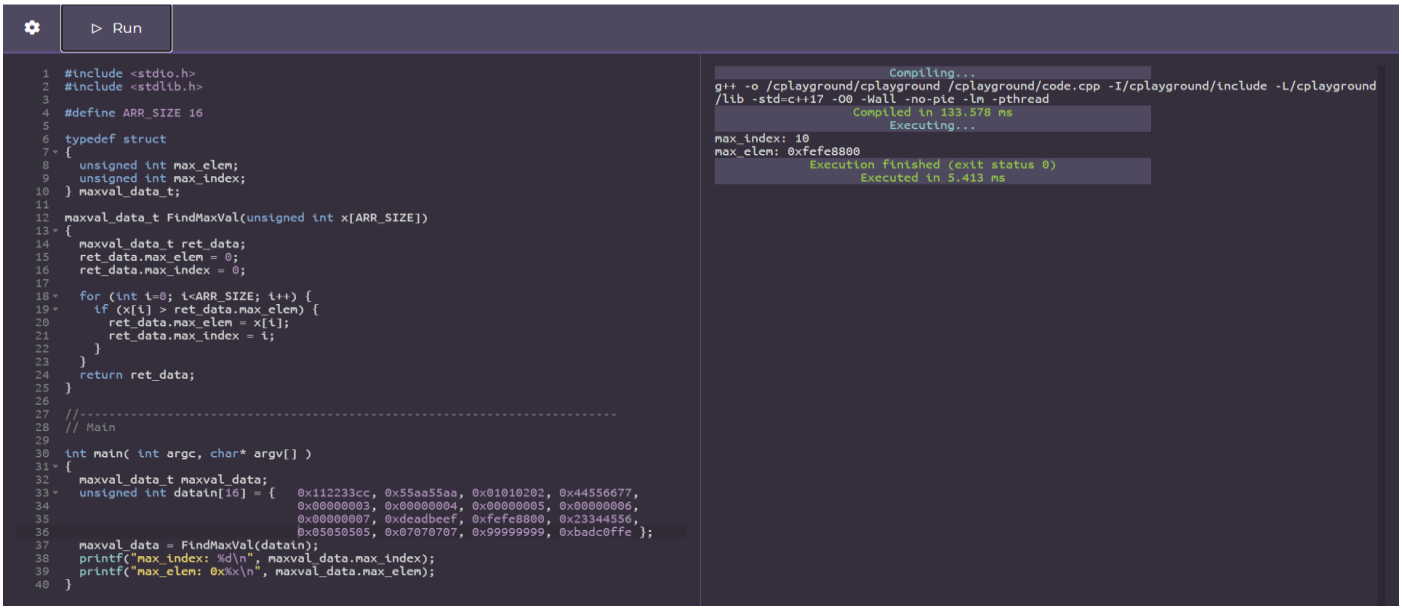
Sigma MCU distribution provides several demo applications that can be used as reference (see Table 1).

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 both local `gcc` installation and an online service (e.g. `cplayground.com`) for this task. Test result for our example is shown in Listing 1.

NOTE: PC and online programming environments will not provide the same peripherals as those included in Sigma MCU. Thus, consider testing only "algorithmic" part of your program in these environments.



Listing 1 Testing software implementation using `cplayground.com`

Go to `activecore/designs/rtl/sigma/sw/benchmarks` 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))

#define ARR_SIZE 16

typedef struct
{
  unsigned int max_elem;
  unsigned int max_index;
} maxval_data_t;

maxval_data_t FindMaxVal(unsigned int x[ARR_SIZE])
{
  maxval_data_t ret_data;
  ret_data.max_elem = 0;
  ret_data.max_index = 0;

  for (int i=0; i<ARR_SIZE; i++) {
    if (x[i] > ret_data.max_elem) {
      ret_data.max_elem = x[i];
      ret_data.max_index = i;
    }
  }
  return ret_data;
}

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

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

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 eCPU to infinite loop.

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

Prepare executable image for eCPU. Open `Makefile` in `activecore/designs/rtl/sigma/sw/benchmarks` 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 = \
   <available applications>
   rsort \
   findmaxval \
   <commented lines>
```

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

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

NOTE: since Sigma MCU does not support hardware multiplication, consider using software one if needed. The example program `mul_sw` is included in ActiveCore distribution.

4. Verify functional correctness in simulation

Open the testbench file `activecore/designs/rtl/sigma/tb/riscv_tb.sv`, choose the CPU configuration, and make `mem_data` parameter of `sigma` instance reference to your ELF 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")

   , .delay_test_flag(0)

   , .mem_type("elf")
   , .mem_data("<PATH_TO_ACTIVECORE>/activecore/designs/rtl/sigma/sw/benchmarks/findmax
val.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 eCPU configuration is shown in Figure 1.

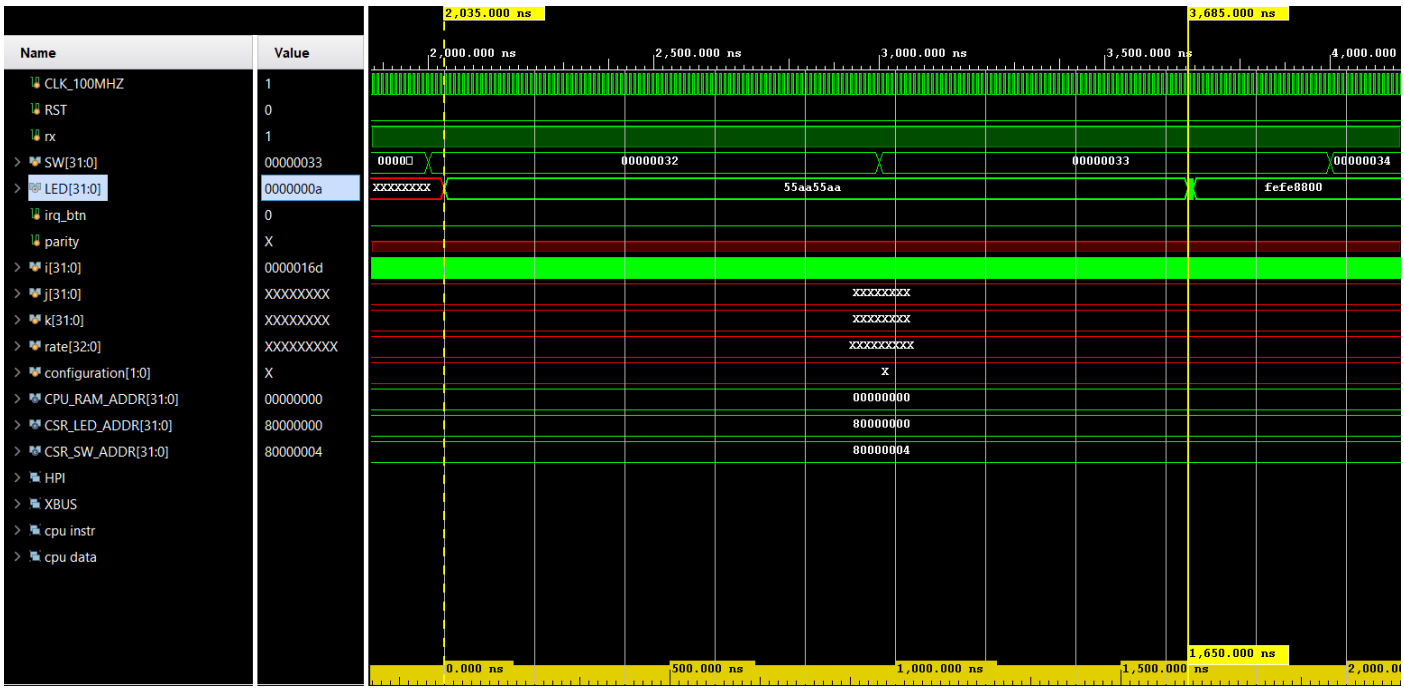


Figure 1 Simulation waveform of program working on eCPU

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

NOTE: if resulting values do not appear in simulation, try the following:

- Check the program is placed in `sigma_tile` RAM. Compare the content of RAM (RAM array is located at `/riscv_tb/sigma/sigma_tile/ram/ram_dual/ram`) to the program binary. Consider specifying absolute path in case the image is not loaded.
- Write intermediate values to LED register.
- Trace program execution.

The program can be traced in simulation using 1-stage eCPU configuration. To switch eCPU configurations for simulation, open corresponding Vivado project and change `CPU` parameter of `sigma` instance in `riscv_tb.sv` testbench. Display the following signals in eCPU (located in `/riscv_tb/sigma/sigma_tile/genblk1.riscv`, see Figure 2):

- `genpstage_EXEC_curinstr_addr` – instruction address
- `genpstage_EXEC_instr_code` – instruction code
- `genpsticky_glbl_regfile` – general-purpose registers

NOTE: you can use the provided `riscv_tb_behav.wcfg` waveform configuration file to display the eCPU state.

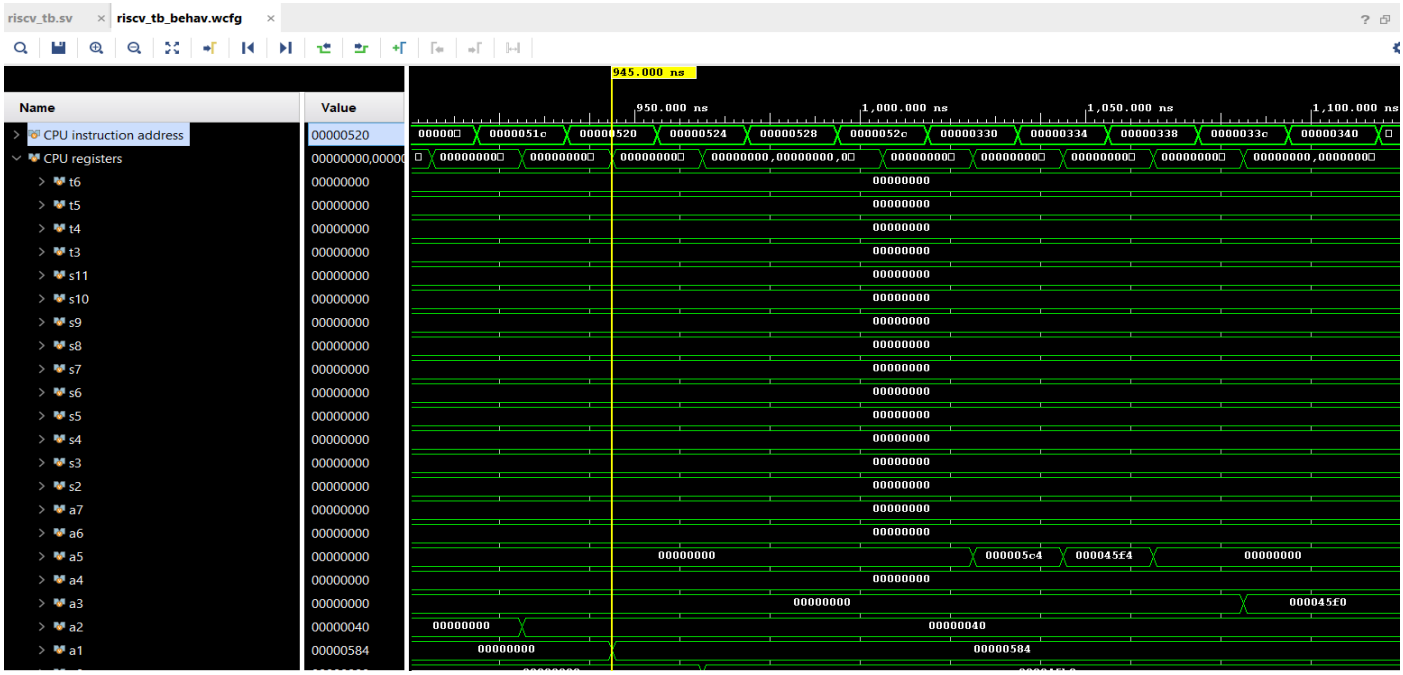


Figure 2 **Tracing program execution using 1-stage eCPU configuration**

```
...
00000518 <main>:
 518: fb010113           addi   sp,sp,-80
 51c: 04000613           li     a2,64
 520: 58400593           li     a1,1412
 524: 00010513           mv     a0,sp
 528: 04112623           sw     ra,76(sp)
...
```

Listing 5 **Fragment of `findmaxval.riscv.dump` program dump file**

Analyze dumped representation of program (`findmaxval.riscv.dump` in our case, see Listing 5) using RISC-V Assembly Programmer's Manual: github.com/riscv/riscv-asm-manual/blob/master/riscv-asm.md. E.g., in our example, instruction at address `0x520` (`li a1,1412`) writes immediate value `1412` (`0x584`) to register `a1`. This operation is marked in Figure 2.

Identify and fix inconsistencies in program execution.

5. Collect performance metrics for various eCPU configurations

Measure the number of clock cycles needed to execute the program by various eCPU configurations. In the testbench, 100 MHz clock is generated, so 2440 ns equals 244 clock cycles. For our example, results are summarized in Table 2.

eCPU configuration	Latency, clock cycles
riscv_1stage	206
riscv_2stage	111
riscv_3stage	155
riscv_4stage	165
riscv_5stage	165
riscv_6stage	192

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

6. Implement the designs and collect metrics of the implementations

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

eCPU configuration	Frequency, MHz	LUTs	FFs
riscv_1stage	70	2144	1180
riscv_2stage	70	2263	1279
riscv_3stage	80	2293	1422
riscv_4stage	140	2284	1686
riscv_5stage	150	2385	1731
riscv_6stage	160	2314	1830

Table 3 Characteristics of provided `sigma_tile` implementations

7. (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/benchmarks/findmaxval.riscv')
```

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

8. Analyze absolute performance of implementations

Now we can analyze the absolute performance values of target functionality implementations based on various eCPU 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.

eCPU configuration	Latency, ns
riscv_1stage	2943
riscv_2stage	1586
riscv_3stage	1938
riscv_4stage	1179
riscv_5stage	1100
riscv_6stage	1200

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

9. (optional) Integrate any UDM-compatible module in Sigma MCU

Since `sigma_tile` XIF protocol is identical to UDM system bus protocol, UDM-compatible modules can be seamlessly integrated in Sigma MCU.

Integrate one of such modules in Sigma MCU (modify `sigma.sv` module) and feed this module with data from eCPU.