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
- Acceleration of Sigma MCU applications using its coprocessor and expansion interfaces
- 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 (with `make` utility) or WSL for RISC-V software compilation in Windows hosts.
5. (for FPGA prototyping) Digilent Nexys A7 FPGA board (https://digilent.com/shop/nexys-a7-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. (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 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 CPU. 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 CPU configurations can be found here:

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

RISC-V CPU supports basic bare metal programming (RV32IM ISA, without FPU, MMU, etc). ActiveCore distribution provides six Sigma MCU projects with different CPU 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 CPU 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 the following directories and build CPU software using `make` command:

- compliance tests: `activecore/designs/rtl/sigma/sw/riscv-compliance`
- demo applications: `activecore/designs/rtl/sigma/sw/apps`

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 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_apps.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. Implement target functionality in pure software

1) Write C application for Sigma MCU

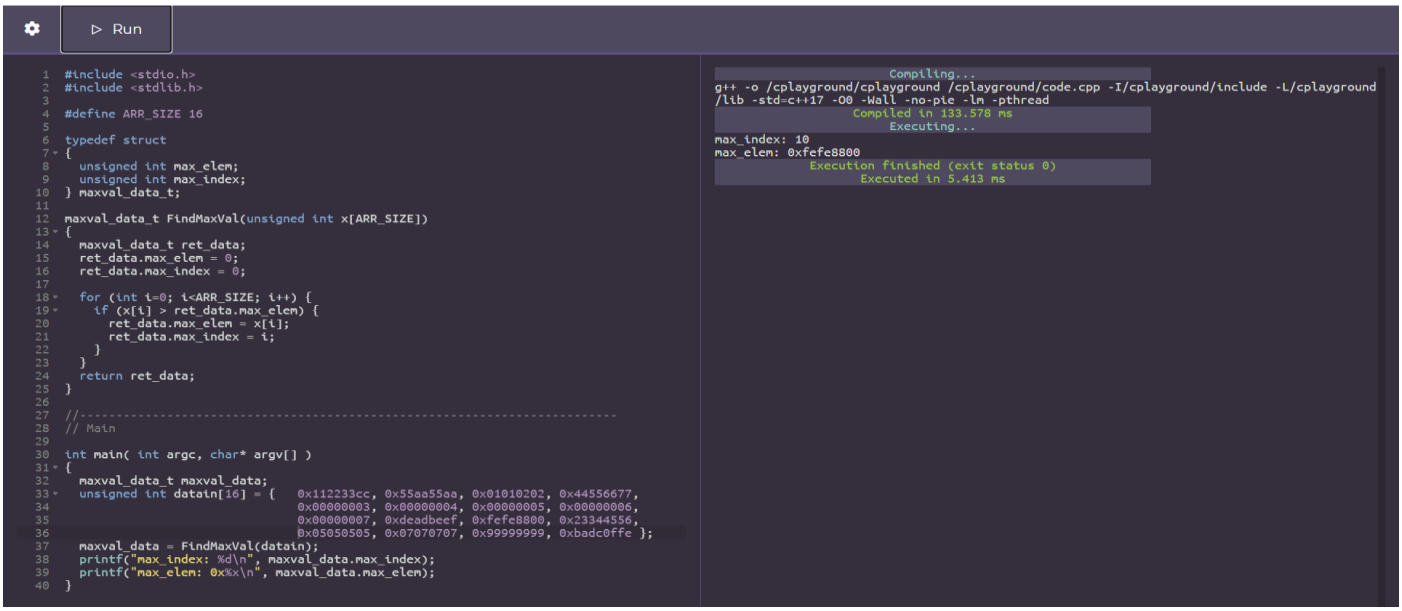
Sigma MCU distribution provides several demo applications that can be used as reference (see Table 1).

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 application for CPU and check its correctness. You can use either 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.

NOTE: PC and online programming environments don't 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/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))

#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 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 = \
   <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.

2) Verify functional correctness in simulation

Open the testbench file `activecore/designs/rtl/sigma/tb/riscv_tb.sv`, select desired clock frequency (needed in Section 7), 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.

```
//`define CLK_HALF_PERIOD                5000          // external 100 MHZ
//`define CLK_HALF_PERIOD                7143          // external 70 MHZ
//`define CLK_HALF_PERIOD                6250          // external 80 MHZ
//`define CLK_HALF_PERIOD                3571          // external 140 MHZ
`define CLK_HALF_PERIOD                3333          // external 150 MHZ
//`define CLK_HALF_PERIOD                3125          // external 160 MHZ

...

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

```
   , .UDM_RTX_EXTERNAL_OVERRIDE("YES")
   , .delay_test_flag(0)

   , .mem_init_type("elf")
   , .mem_init_data("<PATH_TO_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

Once simulation starts, Tcl console should show notification of successful program image upload (see Figure 1).

Figure 1 Notification of successful program image upload

Simulation waveform for 5-stage CPU configuration is shown in Figure 2.

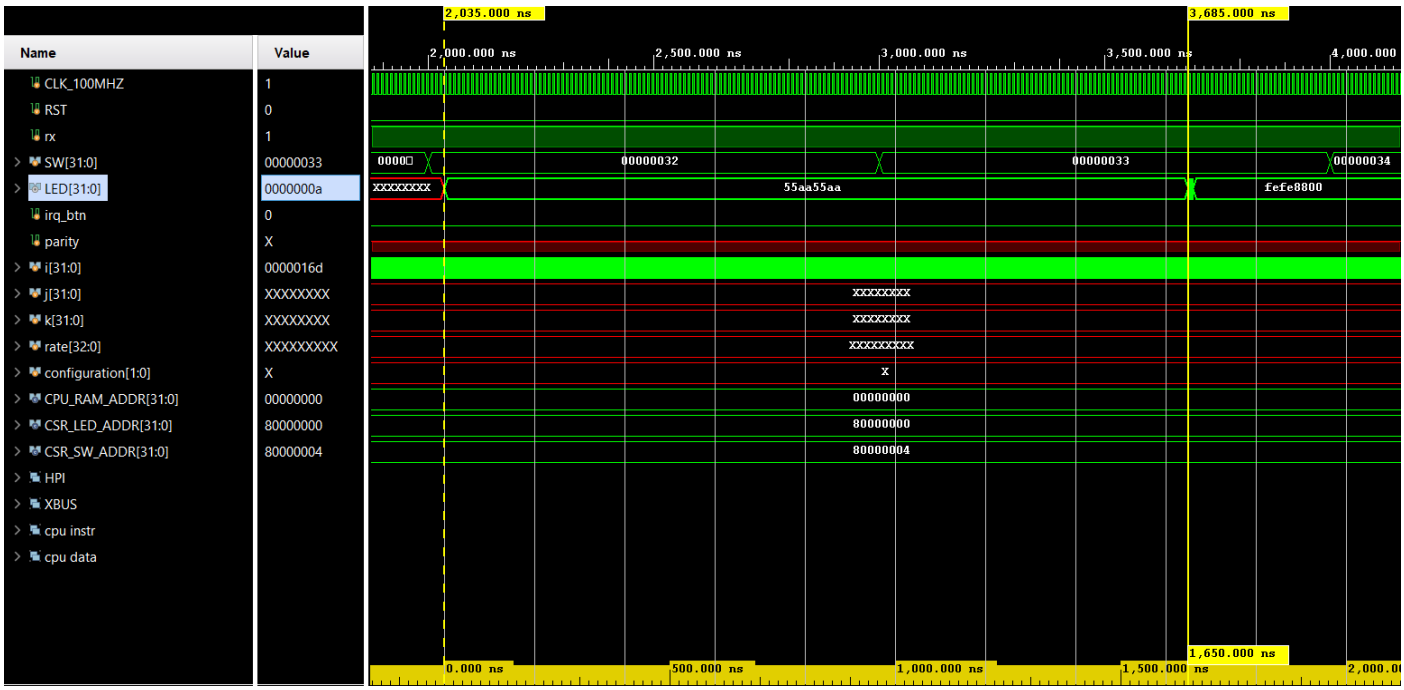


Figure 2 Simulation waveform of program working on CPU

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 CPU configuration. To switch CPU configurations for simulation, open corresponding Vivado project and change `CPU` parameter of `sigma` instance in `riscv_tb.sv` testbench. Display the following signals in CPU (located in `/riscv_tb/sigma/sigma_tile/genblk1.riscv`, see Figure 3):

- `genpstage_EXEC_TRX_LOCAL.curinstr_addr` – instruction address
- `genpstage_EXEC_TRX_LOCAL.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 CPU state.

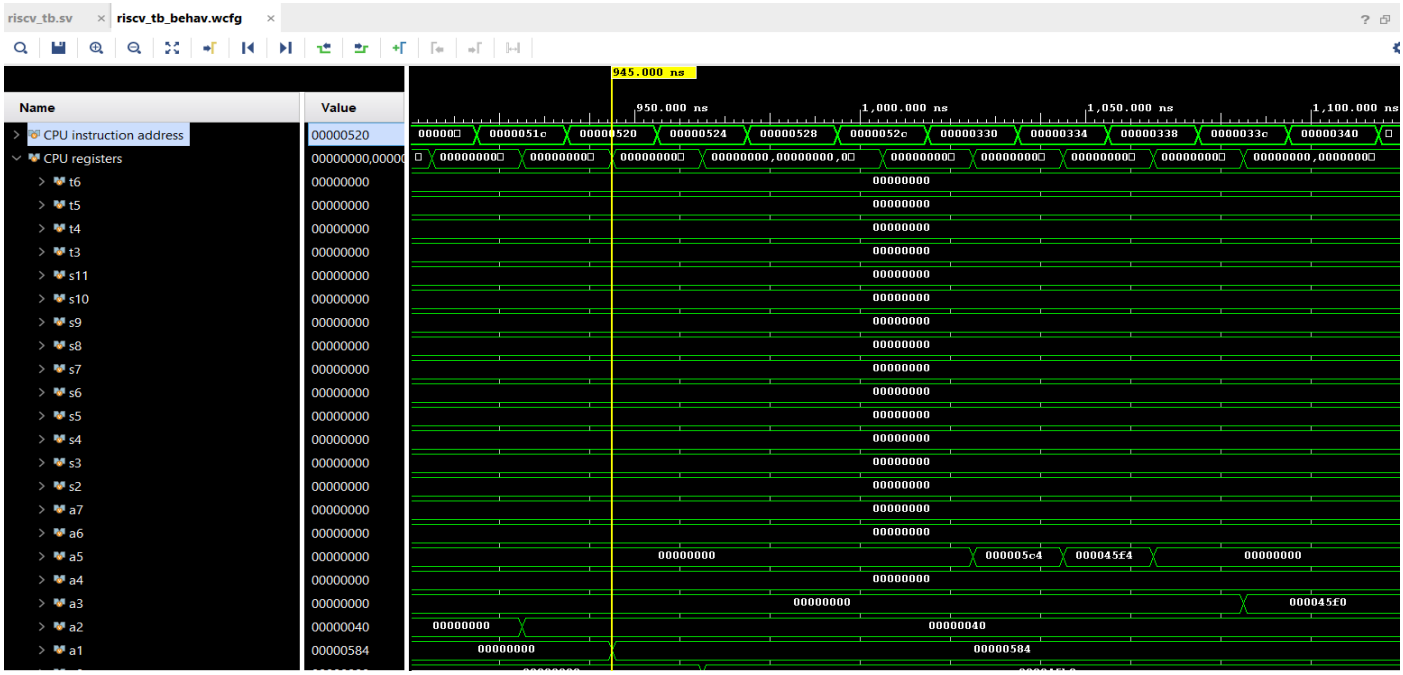


Figure 3 Tracing program execution using 1-stage CPU configuration

```
...
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 3.

Identify and fix inconsistencies in program execution.

3) Implement the designs and collect metrics of the implementations

Characteristics of provided `sigma_tile` configurations are shown in Table 2:

CPU 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 2 Characteristics of provided `sigma_tile` implementations

4) (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>/designs/rtl/sigma/sw/apps/findmaxval.riscv')
```

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

5) Analyze performance for various CPU configurations

Now we can analyze performance values of functionality implementations based on various CPU configurations. Set the actual clock period for each CPU configuration according to Section 4. For our example, these values are shown in Table 3.

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

Table 3 Performance of implementations based on various CPU configurations

4. Accelerate your application in hardware using Sigma MCU coprocessor interface

Sigma MCU provides coprocessor interface, where custom instructions belonging to `custom-0` opcode space (see RISC-V Specification, Vol. 1) are routed. This interface can be useful to accelerate selected, frequently used operations.

1) Write custom coprocessor to accelerate target functionality

By default, CPU coprocessor interface is connected to `coproc_custom0_wrapper` module. Modify this module to implement your coprocessor functionality.

Two operands can be read and one operand written in a single instruction. Beware that execution of these instructions are synchronous to the main CPU pipeline (i.e. response stall will stall the CPU pipeline as well). Coprocessor requests are non-speculative (cannot be killed by the CPU), so this coprocessor can communicate with external modules and have its internal state.

In our example, the coprocessor preserves the index and value of current maximum value. Within each request, the module reads two new values, updates the state, and returns the index of current maximum value. The coprocessor code is shown in Listing 6

```
`include "genexu_MUL_DIV.svh"

module coproc_custom0_wrapper (
   input logic unsigned [0:0] clk_i
   , input logic unsigned [0:0] rst_i
   , output logic unsigned [0:0] stream_resp_bus_genfifo_req_o
   , output resp_struct stream_resp_bus_genfifo_wdata_bo
   , input logic unsigned [0:0] stream_resp_bus_genfifo_ack_i
   , input logic unsigned [0:0] stream_req_bus_genfifo_req_i
   , input req_struct stream_req_bus_genfifo_rdata_bi
   , output logic unsigned [0:0] stream_req_bus_genfifo_ack_o
);

assign stream_req_bus_genfifo_ack_o = stream_req_bus_genfifo_req_i;

logic unsigned [31:0] cur_index, max_index, max_val;
```

```
assign stream_resp_bus_genfifo_wdata_bo = max_index;

always @(posedge clk_i)
  begin
  if (rst_i)
      begin
      stream_resp_bus_genfifo_req_o <= 1'b0;
      cur_index <= 0;
      max_index <= 0;
      max_val <= 0;
      end
  else
      begin
      stream_resp_bus_genfifo_req_o <= 1'b0;
      if (stream_req_bus_genfifo_req_i)
            begin
            if (stream_req_bus_genfifo_rdata_bi.src0_data > max_val)
                  begin
                  max_index <= cur_index;
                  max_val <= stream_req_bus_genfifo_rdata_bi.src0_data;
                  end
            if     ((stream_req_bus_genfifo_rdata_bi.src1_data     >     max_val)     &&
(stream_req_bus_genfifo_rdata_bi.src1_data                                              >
stream_req_bus_genfifo_rdata_bi.src0_data))
                  begin
                  max_index <= cur_index + 1;
                  max_val <= stream_req_bus_genfifo_rdata_bi.src1_data;
                  end
            stream_resp_bus_genfifo_req_o <= 1'b1;
            cur_index <= cur_index + 2;
            end
      end
  end

endmodule
```

Listing 6 Coprocessor design in `coproc_custom0_wrapper` module

2) Write software using the coprocessor

To request the coprocessor, the software should utilize instructions from belonging to `custom-0` opcode space. Add the wrapper for the new instruction using inline assembly and call this wrapper to fire coprocessor requests. Updated software implementation utilizing the coprocessor is shown in Listing 7.

```
#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;

// wrapper for instruction calling the coprocessor
inline unsigned int custom0_instr_wrapper (unsigned int a, unsigned int b)
{
  unsigned int result;
```

```
  asm volatile (".insn r 0x0b, 0x0, 0x0, %0, %1, %2"
    : "=r" (result)
    : "r" (a), "r" (b));
  return result;
}

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=i+2) {
    ret_data.max_index = custom0_instr_wrapper(datain[i], datain[i+1]);
  }
  ret_data.max_elem = x[ret_data.max_index];

  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 7 Updated C source code in `findmaxval.c` utilizing coprocessor request instruction

After compilation, dump file should contain the instruction requesting the coprocessor. For our example, the dump is shown in Listing 8.

```
…
000002a4 <FindMaxVal>:
 2a4: 00052023          sw    zero,0(a0)
 2a8: 00058793          mv    a5,a1
 2ac: 04058613          addi  a2,a1,64
 2b0: 0007a703          lw    a4,0(a5)
 2b4: 0047a683          lw    a3,4(a5)
 2b8: 00d7070b          0xd7070b
 2bc: 00e52023          sw    a4,0(a0)
 2c0: 00878793          addi  a5,a5,8
 2c4: fef616e3          bne   a2,a5,2b0 <FindMaxVal+0xc>
 2c8: 00271713          slli  a4,a4,0x2
 2cc: 00e58733          add   a4,a1,a4
…
```

Listing 8 Fragment of `findmaxval.riscv.dump` program dump file containing instruction calling custom coprocessor

3) Test the updated hardware and software

Repeat the steps 2.2-2.5 to test the updated system in simulation and in hardware. Simulation waveform for our example is shown in Figure 4.

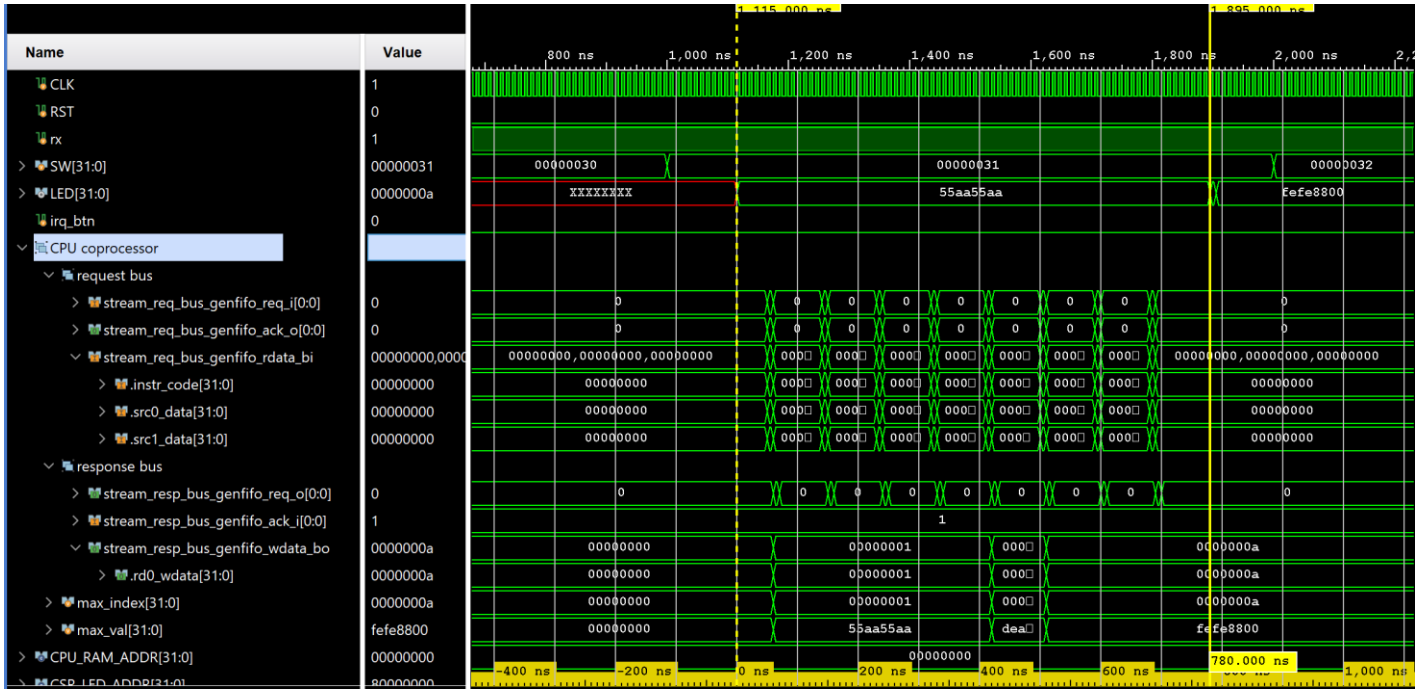


Figure 4 Waveform of CPU requesting custom coprocessor

Note that the new implementation takes 780 ns to complete, compared to 1,650 ns in pure software implementation. So, approximately 2x acceleration has been achieved.

5. Accelerate your application in hardware using Sigma MCU expansion interface

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

NOTE: Beware that XIF address space starts from 0x80000000.

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