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

Computer Systems Design

Laboratory work 4

Using high-level synthesis for hardware accelerator design

Author:

Alexander Antonov

Associate Professor

Lecturer:

Ivan Lukashov

lukashov@itmo.ru

2024

Contents

1. OBJECTIVES

- Student can design the hardware accelerator using HLS methodology
- Students can perform high-level simulation of custom accelerators
- Students can perform C-RTL co-simulation of custom accelerators
- Students can implement custom accelerators using Xilinx Vivado Design Suite
- Students understand how use Xilinx Vivado HLS to accelerate algorithms on FPGAs
- Students can use HLS optimization directives to achieve PPA trade-off

2. OVERVIEW

Laboratory work 4 is an introduction to high-level methods and tools to dedicated hardware design – high-level synthesis (HLS). HLS allows to specify the functional model of hardware using conventional programming languages (usually C/C++/SystemC) and automatically obtain optimized hardware implementations with various PPA ratios. The students learn how to perform basic design procedures for HLS: functional simulation, setting optimization directives, synthesis, C-RTL co-simulation, system integration. HLS has been a hot research topic in recent years and is often considered as replacement for RTL-based design for selected dataflow-dominated applications (signal processing, video, encoding/decoding, etc).

3. PREREQUISITES

Prerequisites are the same as for Lab 1:

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. (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/)
4. (for FPGA prototyping) working Python 3 installation with `pyserial` package

4. TASK

1. Create HLS project in Vivado HLS
2. Design synthesizable implementation for HLS according to your variant
3. Write high-level testbench for your synthesizable function
4. Verify correctness of implementation using functional simulation
5. Verify correctness of implementation using C-RTL co-simulation
6. Integrate the synthesized implementation in UDM baseline project
7. Implement your design, gather and analyze metrics of the implementation
8. (if FPGA board available) Perform HW testing on FPGA board
9. (optional) Obtain alternative implementation and analyze its characteristics
10. Package your solution and submit to the teacher's email

5. GUIDANCE

Detailed guidance will be provided using the example of a module with the same functionality as in Lab 1.

1. Create HLS project in Vivado HLS

Open Vivado HLS and create project with the following settings:

- Project name: put any name (in our example, the project is called `ArrMaxVal`)
- Top function: name of function to be synthesized to hardware. Put the **planned name of your synthesizable function** (in our example: `FindMaxVal`)
- Design and testbench files: leave empty for now
- Solution name: put any name
- Clock period: 10, uncertainty: empty
- Part selection: `xc7a100tcsg324-1`

Press "`Finish`" button

Add source file with synthesizable function to "`Source`" category. You can name it the same as the function. For our example, the name is `FindMaxVal.c`

Add testbench file to "`Test Bench`" category. This file will contain `main` function starting the testbench. Name it `main.c`.

2. Design synthesizable implementation for HLS according to your variant

Write the target function in C language in the source file according to your variant. Use synthesizable constructs only. Source code for the example in shown in Listing 1.

```
#define ARR_SIZE 16

typedef struct
{
	int max_elem;
	int max_index;
} maxval_data;

maxval_data FindMaxVal(unsigned int x[ARR_SIZE])
{
#pragma HLS array_partition variable=x block factor=16
#pragma HLS INTERFACE ap_none port=x
#pragma HLS INTERFACE ap_vld port=ap_return register

	maxval_data 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;
}
```

Listing 1 Source code of `FindMaxVal` function in synthesizable C

NOTE: We have partitioned and set up input data interface to `ap_none` value to make it implement as 16 simple data ports (similar to previous implementations) instead of default RAM interface.

3. Write high-level testbench for your synthesizable function

Write high-level testbench for your synthesizable function in `main.c` file. The file should contain `main` function that starts simulation and calls synthesizable function. The testbench can contain non-synthesizable constructs. Source code for the example in shown in Listing 2.

```
#define ARR_SIZE 16

typedef struct
{
	int max_elem;
	int max_index;
} maxval_data;
maxval_data FindMaxVal(unsigned int x[ARR_SIZE]);

int main()
{
	unsigned int x[ARR_SIZE] = {0x112233cc, 0x55aa55aa, 0x01010202, 0x44556677, 0x00000003,
0x00000004, 0x00000005, 0x00000006, 0x00000007, 0xdeadbeef, 0xfefe8800, 0x23344556,
0x05050505, 0x07070707, 0x99999999, 0xbadc0ffe};
	maxval_data ret_data = FindMaxVal(x);
	printf("Max value: 0x%x\n", ret_data.max_elem);
	printf("Max index: %d\n", ret_data.max_index);
    return 0;
}
```

Listing 2 Source code of `main` testbench function

4. Verify correctness of implementation using functional simulation

Run high-level simulation and verify correctness of implementation. Console output for the example is shown in Listing 3.

```
Max value: 0xfefe8800
Max index: 10
INFO: [SIM 211-1] CSim done with 0 errors.
INFO: [SIM 211-3] *************** CSIM finish ***************
Finished C simulation.
```

Listing 3 Console output for high-level simulation

5. Verify correctness of implementation using C-RTL co-simulation

Run C-RTL co-simulation (solution → context menu → "Run C/RTL Cosimulation") with the following parameters:

- Verilog/VHDL Simulator Selection: Vivado Simulator
- Dump trace: all
- Wave Debug: Yes.

C-RTL co-simulation results for our example are shown in Figure 1.

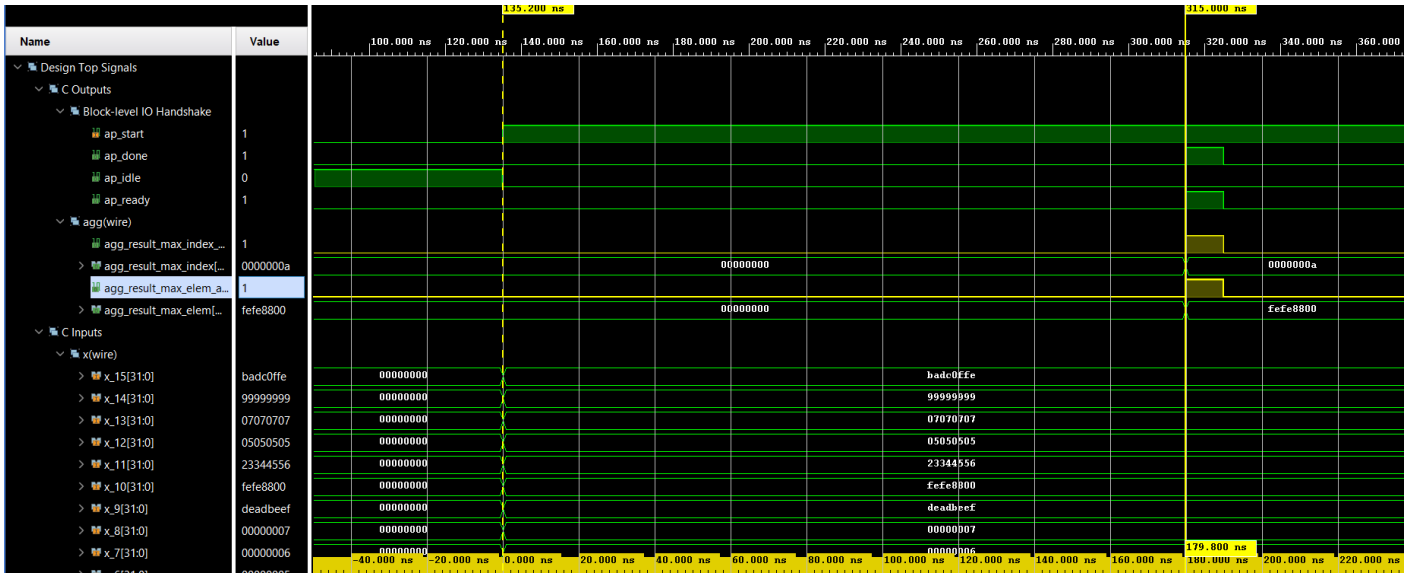

Figure 1 Co-simulation waveform for synthesized HLS-based implementation

The module starts when it receives `ap_start` pulse. Once computation is completed, `ap_done` signal is generated. The module outputs correct results.

6. Integrate the synthesized implementation in UDM baseline project

Add HLS-generated RTL in the following directory to UDM baseline project:

```
<HLS PROJECT NAME>/<HLS SOLUTION NAME>/impl/verilog
```

NOTE: In addition to Verilog, HLS can generate `.dat` files for internal memories initialization. Don't forget to add them to UDM project as well (use "All Files" filter when adding source file). Also, some cores are outputted in the form of `.tcl` scripts that should be called in Vivado to generate the instance of the core (`source <tcl script>` command).

Integration of HLS module is almost identical to multicycle and pipelined implementations (see Lab 2). The difference is in module instantiation code. Instantiation code for our example is shown in Listing 4.

```
// HLS module instantiation
logic hls_module_done;
FindMaxVal FindMaxVal_inst (
    .ap_clk(clk_gen)
    , .ap_rst(srst)
    , .ap_start(1'b1)
    , .ap_done(hls_module_done)
    //, .ap_idle()
    //, .ap_ready()
    , .agg_result_max_elem(csr_max_elem_out)
    //, .agg_result_max_elem_ap_vld()
    , .agg_result_max_index(csr_max_index_out)
    //, .agg_result_max_index_ap_vld()
    , .x_0(csr_elem_in[0])
    , .x_1(csr_elem_in[1])
    , .x_2(csr_elem_in[2])
    , .x_3(csr_elem_in[3])
    , .x_4(csr_elem_in[4])
    , .x_5(csr_elem_in[5])
    , .x_6(csr_elem_in[6])
```

```
    , .x_7(csr_elem_in[7])
    , .x_8(csr_elem_in[8])
    , .x_9(csr_elem_in[9])
    , .x_10(csr_elem_in[10])
    , .x_11(csr_elem_in[11])
    , .x_12(csr_elem_in[12])
    , .x_13(csr_elem_in[13])
    , .x_14(csr_elem_in[14])
    , .x_15(csr_elem_in[15])
);
```

Listing 4 Instantiation of HLS-generated module in `NEXYS4_DDR.sv` module

RTL simulation results for our example are shown in Figure 2.

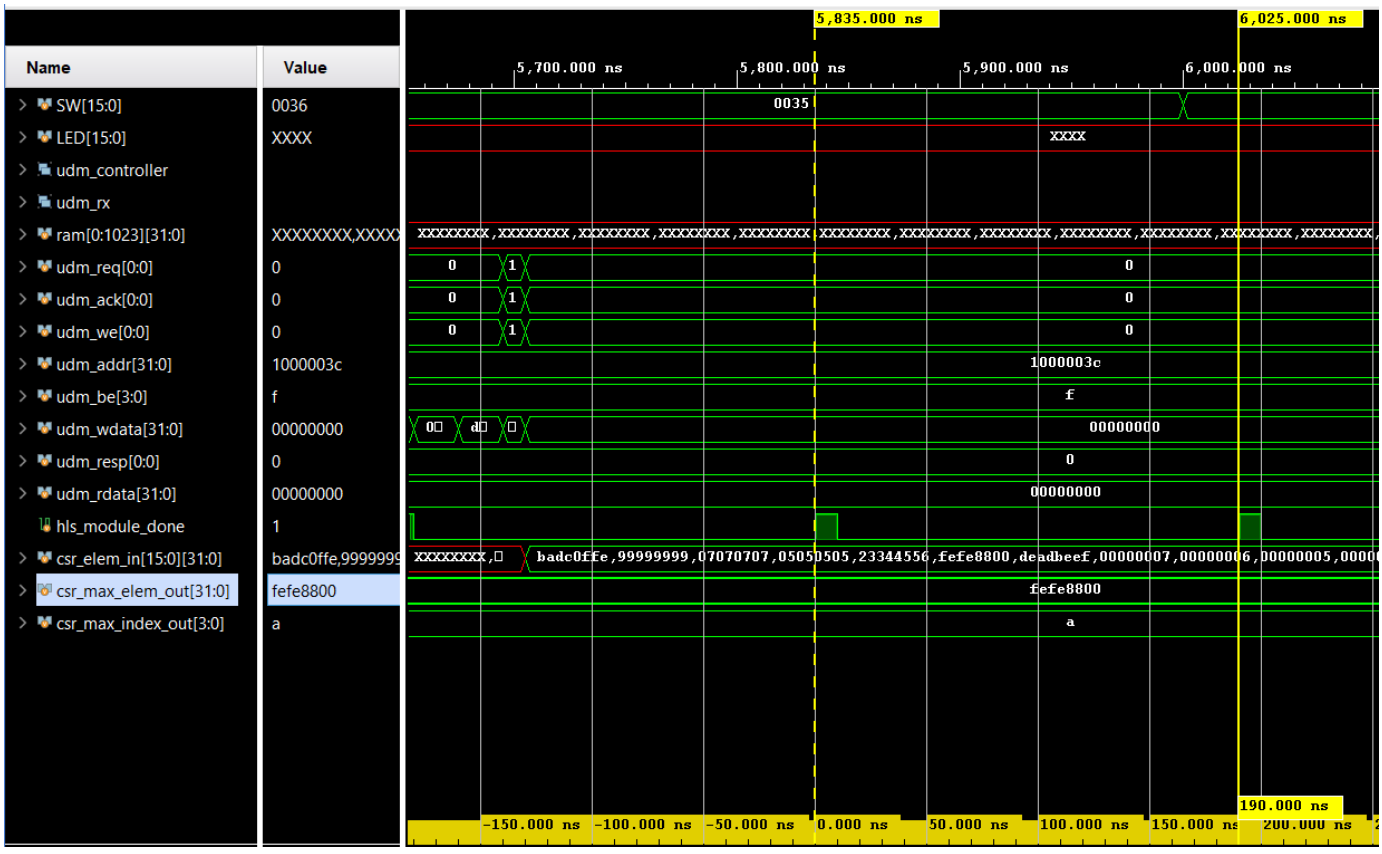

Figure 2 Simulation waveform for synthesized HLS-based implementation in UDM project

7. Implement the design, gather and analyze metrics of the implementation

Generate the bitstream for FPGA device.

Metric values for our example are the following:

- Timing:
  - WNS: 3.787 ns (fine)
  - TNS: 0 ns (fine)

- Performance:
  - Clock frequency: 10 ns (100 MHz)
  - Initiation Interval: 19 clock cycles; 190 ns
  - Bandwidth: 0,0526 op/cycle; 5,26 Mop/second
  - Latency: 19 clock cycles; 190 ns
- HW resources (Implementation → Open Implemented Design → Report Utilization):
  - LUTs: 155
  - FFs (registers): 80

The timing closure is **successful**.

8. (if FPGA board available) Perform HW testing on FPGA board

Python tests and HW testing procedure are the same as in Lab 1.

9. (optional) Obtain alternative implementation and analyze its characteristics

Try to do HLS pipelining. Add the following directive after other pragmas in your synthesizable C function:

```
#pragma HLS pipeline II=<YOUR INITIATION INTERVAL>
```

Now repeat steps 4–8 to get another implementation. Verify its correctness and analyze characteristics.

10. 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 Vivado HLS project. Double-check you included C sources for HLS project.

6. VARIANTS

Same as for Lab 1.