HW/SW Codesign [LU] – WS2023

Get-To-Know Task

Florian Huemer, Dylan Baumann, Andreas Steininger
fhuemer@ecs.tuwien.ac.at
Department of Computer Engineering
TU Wien

Vienna, October 8, 2023

1 Assignment

In this task you will implement a Nios II [1] system and extend it with custom hardware components. This shall help you to get familiar with the tool chain (especially the Platform Designer), the Altera Avalon specification and the Nios II Custom Instruction interface. The system's purpose shall be to normalize 3D vectors, i.e., compute a vector with the same direction as the original one but with length 1 (unit vector). Given a vector $\mathbf{v} = (x, y, z)^T$ the associated unit vector $\hat{\mathbf{v}}$ is given by

$$\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{\sqrt{x^2 + y^2 + z^2}} * \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

All calculations are carried out with fixed point arithmetic in the Q16.16 format (i.e., 16 integer and 16 fractional bits). You don't have to take care of overflows and rounding. The test data, which will be provided, does not trigger overflows.

1.1 Platform Designer System

Your task is to implement a Nios II system using the Platform Designer (formerly Qsys) [6] as shown in Figure 1.1. More details on Avalon interfaces can be found in [4]. The marked blocks are custom hardware modules that you will have to implement, the other blocks can be found in the IP library of the Platform Designer.

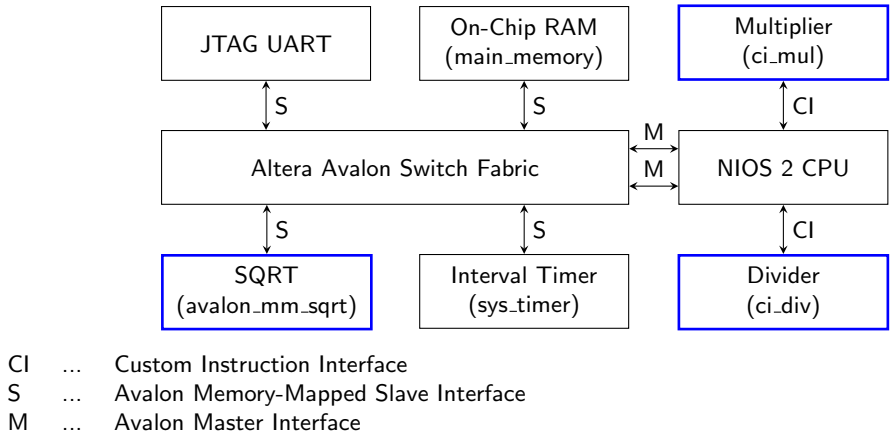

Figure 1.1: System Overview

The system should be clocked with 100 MHz. To generate this clock out of the 50 MHz board clock you will need to instantiate a PLL in the Platform Designer (not shown in the figure). Section 1.3 walks you through the steps necessary to do so.

Important: To avoid inconsistencies with Platform Designer related files, make sure to always save your Platform Designer system before starting the Quartus synthesis.

Component Details

- Nios II Processor
  You can use the default settings for the processor. The data master port must be connected to all slave interfaces, while the instruction master must only be connected to the slave interface of `main_memory`.
- On-Chip RAM (`main_memory`)
  The `main_memory` must have a size of at least 120 kB.
- Interval Timer (`sys_timer`)
  The interval timer is used to measure the performance of your solution. You can also go with the default settings here.

To speed up the fixed point operations involved in normalizing vectors, the following modules must be implemented.

Fixed Point Multiplication
Implement a (fixed length) multi-cycle custom instruction [7] that performs fixed point multiplication.

```
entity ci_mul is
  port (
    clk   : in std_logic;
    clk_en : in std_logic;
    reset : in std_logic;

    dataa : in std_logic_vector(31 downto 0);
    datab : in std_logic_vector(31 downto 0);
    result : out std_logic_vector(31 downto 0)
  );
end entity;
```

For the actual multiplication you can use the `LPM_MULT` IP core [5]. Use an appropriate number of pipeline stages to achieve the specified operation frequency. Remember to specify the number of stages when creating the component in the Platform Designer for your multiplication core, such that the processor is able to stall its pipeline for an appropriate number of cycles to wait for the result of the custom instruction.

Fixed Point Division
Note that when the multiplication instruction is executed, the processor pipeline is stalled until the calculation is completed. To overcome this drawback for the custom instruction for fixed point division, implement two separate instructions (i.e. one extended custom instruction). The first instruction should issue the two operands to the division pipeline (div_write), while the second one should read the result of a previous division (div_read). If no result is ready, div_read should stall the pipeline until one becomes available (delay the *done* signal of the custom instruction). Beware that executing div_read before the associated div_write deadlocks the processor pipeline. Use the `LPM_DIVIDE` IP core [5] to implement the division. To achieve the desired operation frequency implement a 48 stage pipeline. The result values produced by the division core should be collected in a FIFO. For this purpose we provide you with the `alt_fwft_fifo`, which is basically an instance of an `SCFIFO` IP core [3]. This FIFO has first word fall through behavior, which means that the next word that can be read from the FIFO will imminently be presented at the output (see [3], Figure 6: Show-Ahead Mode Waveform). The rd input is then used to acknowledge that the last data has been read (rather than requesting it).

The listing below shows the entity declaration of the extended (variable length) multi-cycle custom instruction for division.

```
entity ci_div is
  port (
    clk   : in std_logic;
    clk_en : in std_logic;
    reset : in std_logic;

    dataa : in std_logic_vector(31 downto 0);
    datab : in std_logic_vector(31 downto 0);
    result : out std_logic_vector(31 downto 0);

    start : in std_logic;
    done : out std_logic;

    n : in std_logic_vector(0 downto 0)
  );
end entity;
```

Hint: When executing div_write or div_read with valid data already present at the FIFO's output, the *done* signal can be generated from the *start* signal in a purely combinational way.

Assume that div_write and div_read have custom instruction opcodes 0 and 1, respectively. Then the following assembly code divides r1 by r2, r3 by r4 and r5 by r6 and stores the results in r1, r2 and r3. Since the calculation takes several clock cycles, the result of the first division will not be available when the first div_read instruction is executed. Hence, the pipeline is stalled until the division is completed.

```
custom 0, r0, r1, r2 // div_write: start calculation of r1/r2
custom 0, r0, r3, r4 // div_write: start calculation of r3/r4
custom 0, r0, r5, r6 // div_write: start calculation of r5/r6
custom 1, r1, r0, r0 // div_read: get result of first div_read
custom 1, r2, r0, r0 // div_read: get result of second div_read
custom 1, r3, r0, r0 // div_read: get result of third div_read
```

Note that the destination register of the div_write and the source registers of div_read are unused. We used register r0 to indicate this fact (this register always reads zero and writing to it has no effect). Custom instructions can of course also be called by C programs, see [7] for details. For that purpose the toolchain automatically generates suitable macros, which are placed in the `system.h` file in the `bsp` directory.

Fixed Point Square Root

The module for the fixed point square root must be implemented as a memory mapped slave component. It has two 32 bit memory locations. Writing to address 0 issues a new value to the module. Reading address 1 should yield the result of previous SQRT operations. Again a FIFO should be used to buffer the results. The square root operation itself should be implemented using the `ALTSQRT` IP core [5] (use 16 pipeline stages).

In contrast to the division instruction, reading a result although none is currently available, should not stall the memory access (with the interface specification shown in the listing below, this would not be possible anyway). The result is simply undefined. To determine whether data is available to be read from the core, reading address 0 should return the empty flag of the FIFO, i.e. if reading address 0 returns 0 then data can be read from address 1.

```
entity avalon_mm_sqrt is
  port (
    clk   : in std_logic;
    res_n : in std_logic;

    -- memory mapped slave
    address   : in  std_logic_vector(0 downto 0);
    write     : in  std_logic;
    read      : in  std_logic;
    writedata : in  std_logic_vector(31 downto 0);
    readdata  : out std_logic_vector(31 downto 0)
  );
end entity;
```

The following C code issues a new SQRT operation and waits until the result is available.

```
IOWR(AVALON_MM_SQRT_BASE, 0, temp);
while (IORD(AVALON_MM_SQRT_BASE, 0)); // Wait for sqrt result
sqrt_result = IORD(AVALON_MM_SQRT_BASE, 1);
```

The same can also be achieved using (inline) assembly. The listing below loads the base address for the SQRT core (stored in the define `AVALON_MM_SQRT_BASE`) into register `r15` and processes the value in `r4`.

```
#define QUAUX(X) #X
#define QU(X) QUAUX(X)
[...]
asm volatile ("\
movhi r15, %hi(" QU(AVALON_MM_SQRT_BASE) ");  /*load base address of SQRT core into r15*/\
ori r15, r15, %lo(" QU(AVALON_MM_SQRT_BASE) ");\
stwio r4, 0(r15);  /*write value stored in r4*/\
1: ldwio r4, 0(r15); bne r0, r4, 1b;  /*wait for SQRT optation to finish (poll busy flag) */\
ldwio r4, 4(r15); /*load SQRT result into r4*/\
");
```

The macros for the base addresses of all slave interfaces are defined in the `system.h` header file. Note that the IOWR/IORD macros as well as the stwio/ldwio instructions bypass the processor cache.

1.2 Software

Basically, you only have to implement the function `v3norm()` in the file `v3norm.c` using the custom instructions and the memory mapped SQRT module from the previous section. Other modifications to the provided source code are not necessary. During the exercise interview we will replace *all* files, except `v3norm.[c|h]` and `cfg.h` with their unchanged versions.

```
void v3norm(fix16_t* a, uint32_t count);
```

- `a`: Pointer to a memory location where the vector data is located (a[3*i], a[3*i+1] and a[3*i+2] contain the x,y and z coordinates of the i-th vector in the Q16.16 fixed point format). Note that this array contains 3*`count` elements. The normalized vectors should be written to the same memory location.
- `count`: The number of 3D vectors stored in `a`. For your implementation you may assume that is value is always greater or equal to 4 (and less than `VEC3_BUFFER_SIZE`).

Your solution shall make use of the fact that some operations can be executed in parallel (e.g., meanwhile a division is being calculated other instructions can be executed). In order to achieve the maximum number of points for this exercise, your solution must

a) produce the correct numerical results and

b) complete the speed test (performed by our check script) in under 7000 cycles

You are not allowed to optimize your code for a specific value of the `count` parameter (i.e., the one used in the speed test). We are going to rank all solutions based on a series of test cases. The best 3 solutions (below 7000 cycles in the speed test) will be awarded with 4, 3 and 2 bonus points. The template contains a Makefile target that calls a script to automatically check your solution (`make check` in the software directory). Furthermore, the template also features a Makefile target (`make remote_check`), that automatically uploads your solution to the TILab (both the Nios II software and the bitstream file for the FPGA), reserves a PC slot using the `rpa` tool, programs the board, performs the test and checks the results. This is useful when you are working locally on your own machine or with the provided VM. Since internally this feature uses the `rpa_shell` command, make sure that this command works before using this make target.

In our reference solution, we were able to complete the speed test in approximately 3000 cycles, using an optimized assembly function [2, ch. 8]. You don't have to achieve this value with your solution, we have just included it as a point of reference for what is possible.

Hint: After you created the basic system in the Platform Designer (at this point you don't need the SQRT module or the custom instructions yet), you can test the system by using the `USE_SOFTWARE_IMPLEMENTATION` define in the `cfg.h` file to compile the Nios II software with a software implementation of the `v3norm` function. The software implementation uses the `fixmath` library. While this yields the correct values it is, however, obviously much too slow! Beware, that depending on your fixed point hardware the results might be off a bit (usually one bit) when compared to the same operation of the library.

1.3 Adding a PLL

The simplest way to add a PLL to your system (and configure it), is to do so directly in the Platform Designer. First, search for *PLL* in the IP catalog and add the *ALTPLL Intel FPGA IP* module to your system. This will open a configuration wizard.

Figure 1.2 to Figure 1.4 show how to generate a PLL configuration sufficient for this assignment's system. All settings not highlighted on a screenshot shall remain at their default state. The central PLL configurations are the frequency of the input clock, i.e., the external oscillator and the frequency of the PLL's output clock.

Once you are done with the configuration, hit "Finish". This will add the module to your system. Proceed by connecting the PLL's in- and outputs to your system. Connect the Avalon memory mapped slave interface to your Nios II's data master interface.

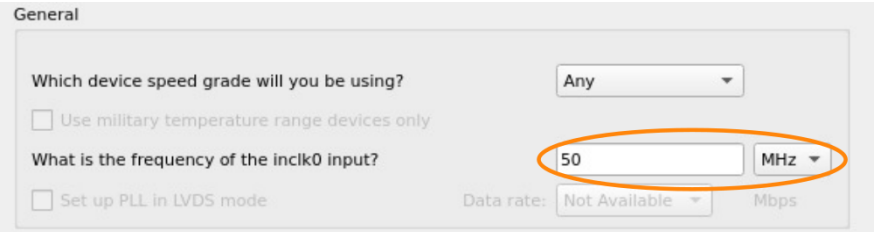

Figure 1.2: Creating a PLL - Step 1.

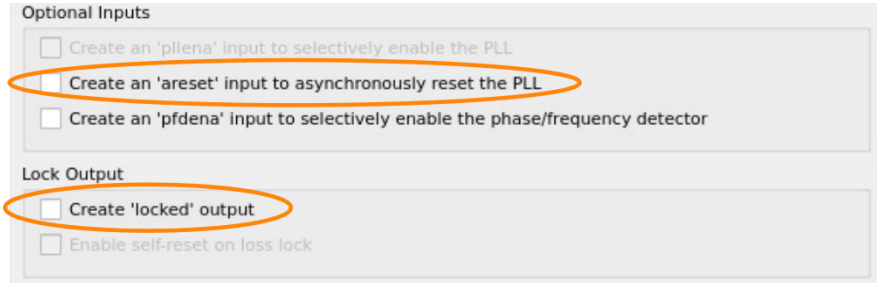

Figure 1.3: Creating a PLL - Step 2.

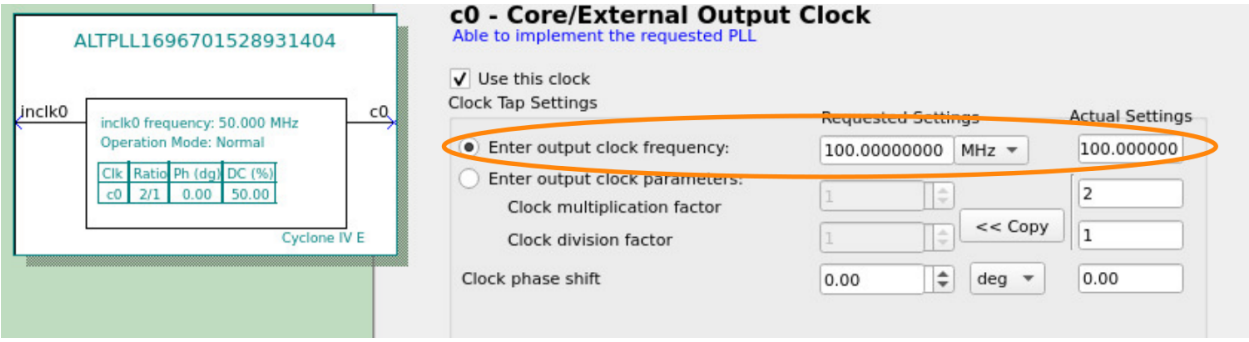

Figure 1.4: Creating a PLL - Step 3.

2 Build Environment

Software Build Process (Make): We provide you with a *make* based build environment to automate the hardware synthesis and software compilation process. For details refer to the README file in the template folder. Before you can build the software for the first time, you have to create the `settings.bsp` file. This file is used by the Nios II build tools to automatically generate the board support package (BSP).

- Use the `edit_bsp` target of the software folder's Makefile
- Configure the paths (see Figure 2.1) and click OK:
  - SOPC Information File Name: Select the *.sopcinfo file created by the Platform Designer, which is located in the quartus directory (`template/quartus/gettoknow.sopcinfo`)
  - BSP target directory: `template/software/bsp`
  - BSP Settings File Name: `template/software/settings.bsp`
- Now you can make customizations to the BSP. For this assignment the default settings are sufficient. However, be sure that the *small C library* is NOT enabled, because the software template will not work otherwise. This is also the reason why you need at least 120 kB of `main_memory`. Additionally, select the *reduced device drivers* option (see Figure 2.2). Furthermore, make sure that both `sys_clk_timer` and `timestamp_timer` are set to "none".
- To complete the process close the BSP editor.
- Use the Makefile in the software folder to start the build process

Testing: To test your solution use the `check` target in the software Makefile. For this target to work the FPGA must already be programmed (`make download` in the quartus folder).

The test script creates a set of random vectors and sends them to the Nios II processor via the `nios2-terminal` (JTAG UART). The data is also processed by the reference C implementation executed on the PC. A final comparison between the outputs of both versions shows if there are errors in

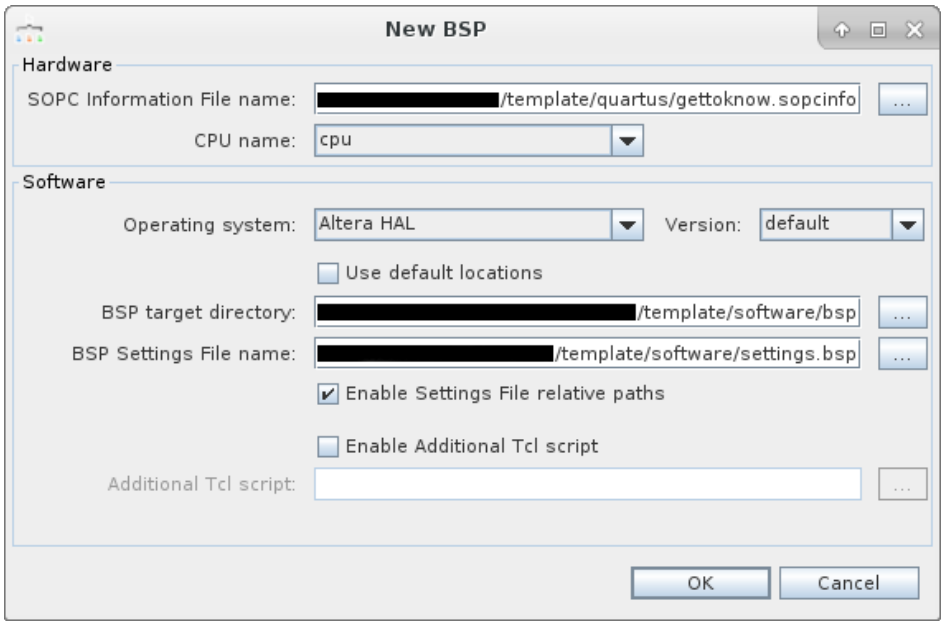

Figure 2.1: Nios II BSP Editor: New BSP

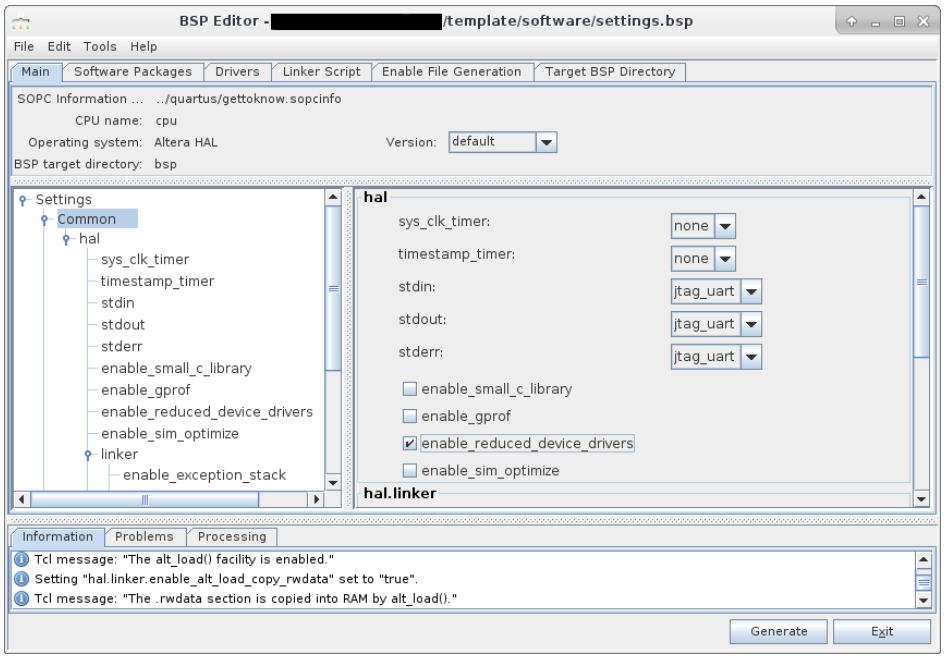

Figure 2.2: Nios II BSP Editor

your solution. Note that the software version of the normalization function (`v3norm_sw`) uses the macro `CALCULATION_METHOD` to switch between two possible calculation methods (`MUL` and `DIV`), which yield different results. Thus, set this macro (`cfg.h`) to match the style of your implementation in `v3norm`.

For the communication over the UART interface a simple text based protocol is used.

```
process n
vec[0].x
vec[0].y
vec[0].z
...
vec[n-1].x
vec[n-1].y
vec[n-1].z
exit
```

A hexadecimal encoding is used for all numbers. The command `process` is followed by a value indicating the number of 3D vectors that should be processed and the list of actual vectors. The program returns a list of processed vectors in the same format. The special command `exit` is used to terminate the communication. It instructs the Nios II processor to send a Ctrl-D command over the UART interface, which will effectively terminate the `nios2-terminal`. Knowledge about the communication protocol format is not required to complete the assignment. It is just included for the sake of completeness.

The Nios II program template also supports the `check_speed` command. This command is used to check the performance of your solution and is also executed by the check script. It basically calls the `process` function for a certain number of times and measures the (minimum) number of clock cycles required for the operation using the interval timer (`sys_timer`).

Hence, the last three lines of the output of the `check` target of the Makefile should be

```
>>> value check PASSED <<<
runtime = [...] cycles
>>> speed check PASSED <<<
```

3 Submission

The deadline for this exercise is October 30, 23:59. Use the `submission` Makefile target to create an archive that you then upload to TUWEL. Using the provided Makefile infrastructure, make sure that, when the archive is extracted, the contained project can be compiled and executed in the lab environment. This is what we will use when evaluating your solution. Finally also register for an exercise interview in TUWEL.

References

[1] Intel NIOS II Processor Support. `https://www.intel.com/content/www/us/en/programmable/products/processors/support.html`, 2020.

[2] Intel. *Nios II Processor Reference Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2cpu-nii5v1gen2.pdf`, August 2019. Version 2019.08.21.

[3] Intel. *SCFIFO and DCFIFO IP Cores User Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_fifo.pdf`, November 2019. Version 2019.11.21.

[4] Intel. *Avalon Interface Specifications*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf`, May 2020. Version 2020.05.26.

[5] Intel. *Integer Arithmetic IP Cores User Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_lpm_alt_mfug.pdf`, April 2020. Version 2020.04.26.

[6] Intel. *Intel Quartus Prime Pro Edition User Guide: Platform Designer*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-qpp-platform-designer.pdf`, July 2020. Version 2020.07.01.

[7] Intel. *Nios II Custom Instruction User Guide*. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nios2_custom_instruction.pdf`, April 2020. Version 2020.04.27.

Revision History

Revision	Date	Author(s)	Description
1.0	09.10.2023	FH, DB	Initial version

Author Abbreviations:

FH Florian Huemer
DB Dylan Baumann