EPFL

CS-476:
Embedded System Design

Practical work 2

Profiling and custom instructions

Version:
1.0

Contents

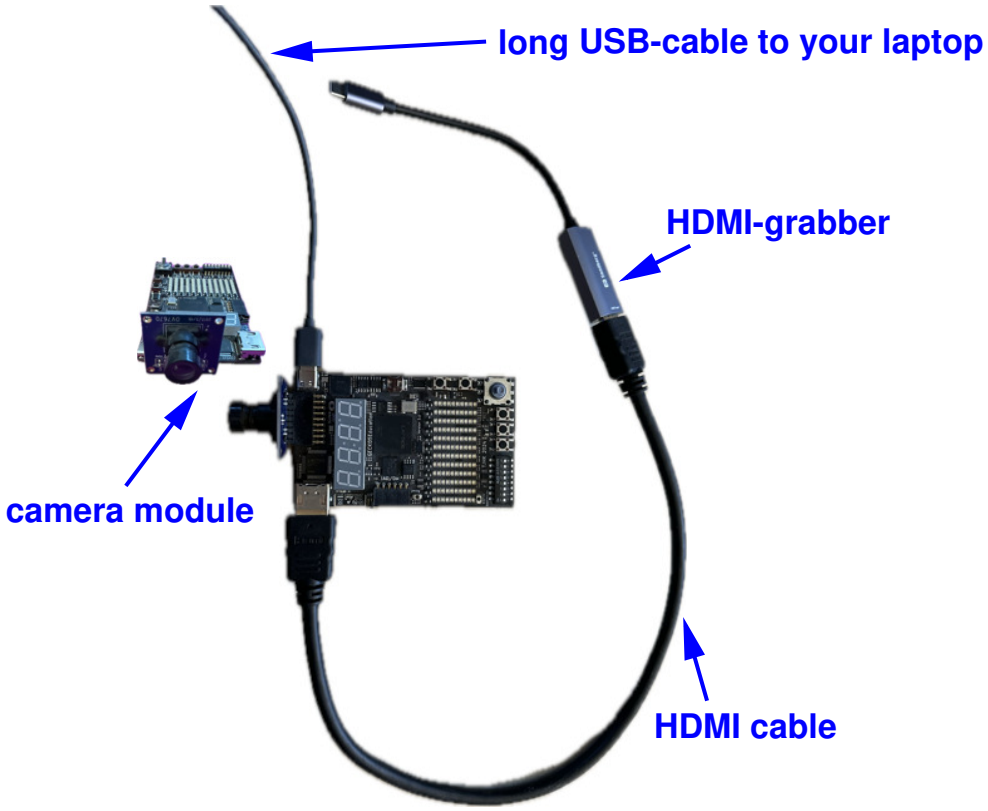
1 Introduction

1.1 Before you start

In this practical work we are going to add modules to our base system. Before continuing with the exercises it is important to read the `README.md` file which you will find in the zip-file of the virtual prototype. Make sure that you can build the virtual prototype and can run the `helloWorld` and `camera` program.
Important: Upload the `.cmem` file to your virtual prototype and not the `.mem` file!

1.2 Setting up your hardware

You have to build-up your hardware as shown below:



The HDMI-grabber will show itself as a webcam on your machine and the image can be shown by any program that can show the contents of a webcam.

2 Exercises

2.1 Prerequisites

Download the `grayscale.zip` file from moodle and unzip it in the `programms/`-directory of your virtual prototype. This zip-file contains all the source file required to grab an image from the camera, convert it to grayscale, and show it on the HDMI-screen/grabber.
Compile the program and run it on your virtual prototype.

2.2 Profiling custom instruction hardware

To be able to know what is going on during execution of the grayscale conversion, we are going to implement profiling counters as a custom instruction. We are going to implement four 32-bit counters, where:

- `Counter0`: Counts the number of CPU-cycles when enabled.
- `Counter1`: Counts the μC stall cycles when enabled.
- `Counter2`: Counts the bus-idle cycles when enabled.
- `Counter3`: Counts the number of CPU-cycles when enabled.

The module definition of the profiling module hardware is given by:

```
module profileCi #( parameter[7:0] customId = 8'h00 )
                  ( input wire         start,
                                       clock,
                                       reset,
                                       stall,
                                       busIdle,
                    input wire [31:0]  valueA,
                                       valueB,
                    input wire [7:0]   ciN,
                    output wire        done,
                    output wire [31:0] result );
```

Where:

- `customId`: The identifier to which this custom instruction should react. Make sure that this identifier does not collide with an already attached custom instruction!
- `start`: The start indicator of the μC.
- `clock`: The system clock (`s_systemClock`).
- `reset`: The system reset signal (`s_cpuReset`).
- `stall`: The stall indicator of the μC. In the toplevel found in:
  `virtual_prototype/systems/singleCore/verilog`
  the μC is the component `cpu1`. The stall indicator is the output `cpuIsStalled`. This output is currently not connected.
- `busIdle`: The bus-idle indicator (`s_busIdle`).
- `valueA`: The value of register A (`s_cpu1CiDataA`).

- `valueB`: The value of register B (`s_cpu1CiDataB`).
- `ciN`: The custom instruction identifier (`s_cpu1CiN`).
- `done`: The done-indicator that needs to be or-ed to the signal `s_cpu1CiDone`.
- `result`: The custom instruction result that needs to be or-ed to the signal `s_cpu1CiResult`.

Important:

- The output `done` should only be asserted when the signal `ciN` equals to `customId` and start is asserted!
- The output `result` should always contain the value `0x00000000`, unless the signal `ciN` equals to `customId` and start is asserted!

Hint: You can use the counter module that we have seen last week. For the functionality, we will:

- Use `valueA[1:0]` to select which counter-value to put on the `result`-output. Where `valueA[1:0] == 2'd0` selects `counter0`, `valueA[1:0] == 2'd1` selects `counter1`, etc.
- Use `valueB[11:0]` to control the counters, where:

bit:	function:
`valueB[0]`	A 1 enables counter0, a 0 does nothing.
`valueB[1]`	A 1 enables counter1, a 0 does nothing.
`valueB[2]`	A 1 enables counter2, a 0 does nothing.
`valueB[3]`	A 1 enables counter3, a 0 does nothing.
`valueB[4]`	A 1 disables counter0, a 0 does nothing.
`valueB[5]`	A 1 disables counter1, a 0 does nothing.
`valueB[6]`	A 1 disables counter2, a 0 does nothing.
`valueB[7]`	A 1 disables counter3, a 0 does nothing.
`valueB[8]`	A 1 resets counter0, a 0 does nothing.
`valueB[9]`	A 1 resets counter1, a 0 does nothing.
`valueB[10]`	A 1 resets counter2, a 0 does nothing.
`valueB[11]`	A 1 resets counter3, a 0 does nothing.

Note: disabling a counter has precedence over enabling the counter.

Implement the required verilog description for this module and integrate the module into the toplevel (`or1420SingleCore.v`) verilog file. Build the new system and check that the grayscale program is still running correctly.
Important: Do not forget to add your verilog files to the `project.files`-file.
Hint: Check the correct functionality of your module by making a simple testbench.

2.3 Profiling of the grayscale conversion

Now that we have the hardware for the profiling, we can use it to profile our program. To control the profiling module, we can use following constructs:

- To control the counters you can use:

```
uint32_t control = 7;
asm volatile ("l.nios_rrr r0,r0,%[in2],0xB"::[in2]"r"(control));
```

- To read a counter you can use:

```
uint32_t result, counterid = 1;
asm volatile ("l.nios_rrr %[out1],%[in1],r0,0xB":[out1]"=r"(result):
                                                 [in1]"r"(counterid));
printf("%d\n", result);
```

- To read a counter and control all counters, you can use:

```
uint32_t result, counterid = 1;
uint32_t control = 7<<4;
asm volatile ("l.nios_rrr %[out1],%[in1],%[in2],0xB":
              [out1]"=r"(result):
              [in1]"r"(counterid),
              [in2]"r"(control));
printf("%d\n", result);
```

In all the above examples `0xB` should correspond to the value that you specified as `customId` in your hardware!

Modify your grayscale program such that it prints for the conversion from rgb565 to grayscale:

- The number of μC execution cycles.
- The number of μC stall cycles.
- The number of bus-idle cycles.

2.4 Handing in

This is part 1 of 2 parts for this graded PW. You have to hand-in the results of this exercise by zipping the verilog and c-files in a single zip archive and upload it to moodle. Next week a solution to this exercise will be available. Not uploading your work before the start of the lecture of next week will cost you 5% of the 15% this PW stands for.

2.5 Next week

Next week we are going to look how we can speed-up the grayscale conversion by using a custom instruction.