EPFL

CS-476:
Embedded System Design

Practical work 2

Grayscale and custom instructions

Version:
1.0

Contents

1 Introduction **1**

2 Exercises **2**
2.1 Prerequisites 2
2.2 Grayscale custom instruction hardware 2
2.3 Profiling 2
2.4 Optional (not graded) 3
2.5 Handing in 3

1 Introduction

Last week in the first part of this PW you implemented profiling counters to be able to observe run-time parameters of a program during it's execution. This week we are going to dive into the grayscale-conversion algorithm. The conversion from RGB into grayscale is given by:

$$\text{grayscale} = \frac{21}{100} \cdot \text{red} + \frac{71}{100} \cdot \text{green} + \frac{8}{100} \cdot \text{blue} \tag{1.1}$$

As the algorithm requires a division, our software-team optimized this algorithm to:

$$\text{grayscale} = \frac{54}{256} \cdot \text{red} + \frac{183}{256} \cdot \text{green} + \frac{19}{256} \cdot \text{blue} \tag{1.2}$$

As we now divide by the factor 256, we do not require any more a division, but can replace it by a simple shift to the right of 8 positions.
Although the software-team improved on the run-time of this algorithm, we should be able to even go further with a custom instruction.

2 Exercises

2.1 Prerequisites

For this part of the PW we require to also have the profiling counters. If you were not able to perform the task of last week, you can download from moodle a system with integrated profiling counters. The `custom-instruction identifier` for this custom instruction is 12.

2.2 Grayscale custom instruction hardware

We are going to implement a custom instruction that performs the grayscale conversion in a single cycle. The module is given by:

```
module rgb565Grayscalelse #(parameter [7:0] customInstructionId = 8'd0 )
                           ( input wire         start,
                             input wire [31:0]  valueA,
                             input wire [7:0]   iseId,
                             output wire        done,
                             output wire [31:0] result );
```

Where:

- `customInstructionId`: 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.
- `ValueA`: The value of register A (The RGB656 value).
- `iseId`: The custom instruction identifier.
- `done`: The done-indicator to the μC.
- `result`: the custom instruction result (The grayscale value).

Note that only the lower 16-bits of `valueA` contains information, namely the RGB565 value of a pixel. Furthermore, `result` will only contain proper information in the lower 8 bits (the grayscale value), the other bits need to be 0.
Implement this custom instruction in verilog.

2.3 Profiling

To be able to have a good idea how "successful" our custom instruction is, we are going to look into the speed-up. Last week you already modified your grayscale program to record:

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

Now we are going to extend this.
Assignment 1: Record these numbers for the unmodified grayscale code.
Assignment 2: Modify your grayscale code such that it uses the custom instruction.
Assignment 3: Record again the above numbers for the execution of the custom-instruction accelerated grayscale code.
Assignment 4: Compare the above values and take your conclusions.
Note: The "real" cycles the μC is doing actual work is equal to "the number of μC execution cycles"-"the number of μC stall cycles".

2.4 Optional (not graded)

As a custom instruction can take up to 2 32-bit words (`ValueA` and `ValueB`) and provides one 32-bit result (`result`) we should be able to process 4-pixels in a single cycle (4x16bit RGB565 as input and 4x8-bit grayscale as output). This should give us an even higher speed-up. The only aspect to think about here is how to handle the *big- little-endian* problem.

2.5 Handing in

This is part 2 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 (hence the complete virtual prototype) and upload it to moodle also upload a `readme.md`-file where you note the changes you made and the answers to the questions. 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 10% of the 15% this PW stands for.