

Lecture 3

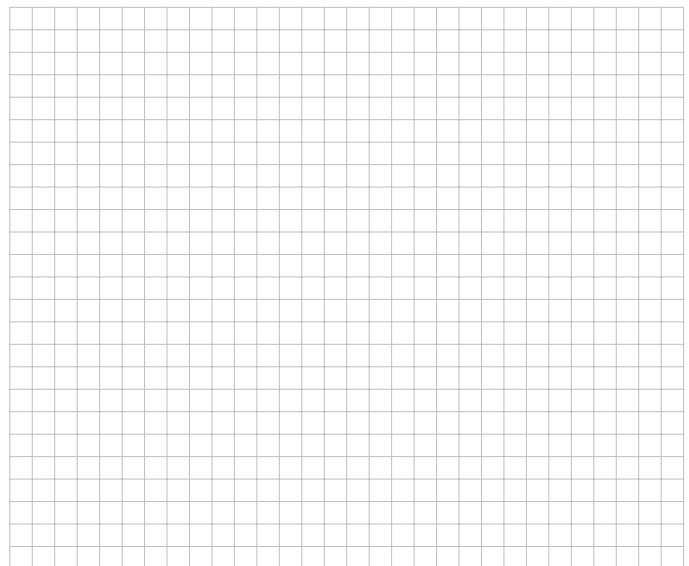
Embedded system design

CS476 - ESD
March 10, 2024

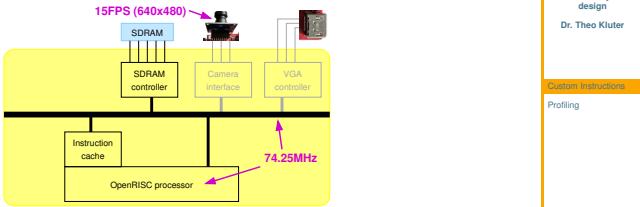
Dr. Theo Kluter
EPFL

Rev. 1.0 - 3.

Notes



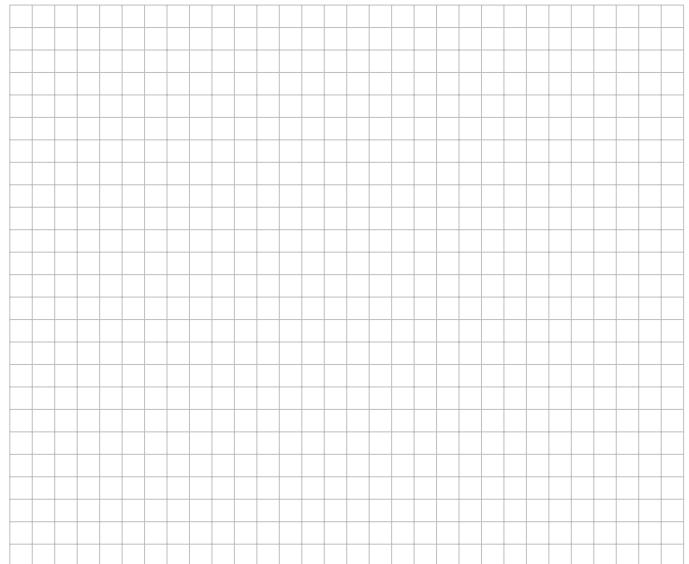
Acceleration



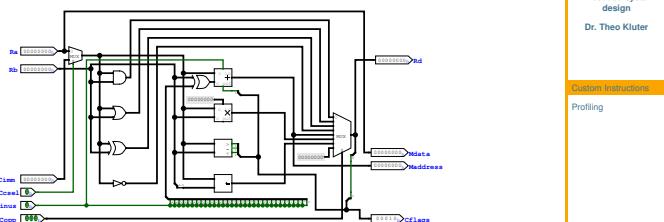
- ▶ In the first week we have seen that our system cannot calculate Sobel in real time.
- ▶ Last week we saw on-chip memories and their usage.
- ▶ We are now going to look into the details how we can add hardware to aid the software.
- ▶ And we will start off with the *custom instructions (CIs)*.
- ▶ To understand the concept of custom instructions we have to dive a bit into the architecture of the μ C.

Rev. 1.0 - 33

Notes



Arithmetic Logic Unit



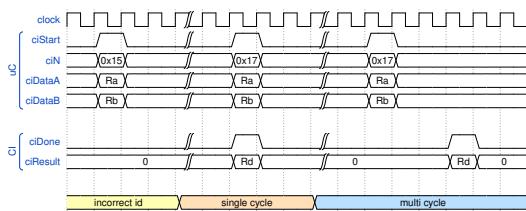
- The Arithmetic Logic Unit (ALU) is the *heart* of the μ C.
- It receives two data (R_a, R_b) from the register-file and produces one result (R_d) to the register-file.
- The operation done is selected by the control signals ($C_{imm}, C_{csel}, C_{minus}, C_{opp}$), that are set depending the instruction.
- Note that only one operation can be selected, although all operations are performed.

Rev. 1.0 - 30

Notes



Custom instruction hardware architecture



- ▶ Assume that the custom instruction hardware has the *custom instruction identifier* 0x17.
- ▶ When the `cIN` does not correspond to the *custom instruction identifier* no `done` is generated.
- ▶ Otherwise we can have a single-cycle, or a multi-cycle response.
- ▶ Note that in case of a multi-cycle response the μ C is stalled!

Rev. 1.0 - 31

Notes



Embedded system
design
Dr. Theo Kluter

Custom Instruction

Custom instruction software interface

- ▶ Now that we have seen how to make the hardware part of a custom instruction, we also want to use it.
- ▶ Of course the compiler has no knowledge nor support for these instructions.
- ▶ We have to activate them with an assembly instruction:

```
uint32_t result, regA, regB;  
  
asm volatile ("l.nios_rrr %[rd],%[ra],%[rb],0x17":[rd] "=r"(result):  
           [ra]"r"(regA), [rb]"r"(regB) );
```

Note: The `0x17` is the *custom instruction identifier* of the custom instruction you want to activate.
- ▶ There are variations, like:
 - ▶ A custom instruction with only inputs:

```
asm volatile ("l.nios_rrr r0,%[ra],%[rb],0x1a":[ra]"r"(regA),[rb]"r"(regB) )
```
 - ▶ A custom instruction with only an output:

```
asm volatile ("l.nios_rrr %[rd],r0,r0,0x72":[rd] "=r"(result) );
```
- ▶ ...
- ▶ Note the usage of the register `r0`!

Rev. 1.0 - 33

Notes



Embedded system
design
Dr. Theo Kluter

Custom Instruction

Custom instruction usage

- ▶ The question is now: How to use custom instructions?
- ▶ Let's take a design example (important: this is not the grayscale conversion used in our system!)

```
void rgbToGrayscale( int width,
                      int height,
                      const uint32_t *rgb_source,
                      uint32_t *grayscale_destination ) {
    int loop;
    uint32_t temp, grayscale;

    for (loop = 0; loop < width*height; loop++) {
        temp = rgb_source[loop] & 0x3F; // red value
        grayscale = temp*77;
        temp = (rgb_source[loop] >> 8) & 0x3F; // green value
        grayscale += temp*151;
        temp = (rgb_source[loop] >> 16) & 0x3F; // blue value
        grayscale += temp*28;
        grayscale |= 0xFF00;
        grayscale_destination[loop] = (grayscale << 8) | grayscale | (grayscale >> 8);
    }
}
```

► We can look at the *Data Flow Graph (DFG)* of this function.

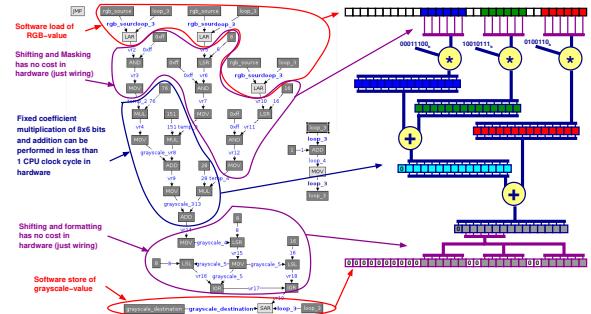
Notes



Embedded system
design

Custom Instructions

Custom instruction usage (DFG)



- And all this can execute in 1 CPU-cycle (of course without the load/store, and loop; they are still required)!

EPFL

EPFL

Embedded system
design
Dr. Theo Kluter

Rev. 1.0 – 3.11

Notes

Profiling

- ▶ But how to know the influence of these hardware enhancements?
- ▶ We can just insert counters in our program that counts the number of function calls, execution time, etc.
- ▶ We call this process *Profiling*
- ▶ And the insertion of these counters can be automated.
- ▶ A classic example of this automatic insertion is gprof of the GNU-tool-chain.
- ▶ Some more advanced profiling tools are valgrind and kcachegrind.
- ▶ However:
 - ▶ These tools give us just information on execution time, not if the limitations are due to software or hardware hot spots.
 - ▶ Many of these tools are only available for "known architectures", maybe not for the system you are targeting.

Notes

EPFL

Embedded system design

Rev. 1.0 - 3.11

Notes

Profiling

Profiling has its limitations, from a software point:

- We require representative data-sets to profile as:
 1. A given data-set might not trigger some parts of the code resulting in improper profiling information.
 2. A given data-set might be a *corner case* only banging on one function, resulting in improper profiling information.
 3. In general: *garbage-in* results in *garbage-out*.
- Profiling should be performed on the target hardware, as compilers optimize differently for different targets. Profiling on a desktop gives other results as profiling on for example an ARM system.
- The program should behave properly, e.g. the extensive use of function pointers might render the profiling tool useless.

Profiling has its limitations, from a hardware point:

- If profiling is done on another architecture the results can be bogus as it does not represent the dynamic behavior of the target system.
- Modeling of all parameters in the virtual prototype has to be done correctly, otherwise the real SOC can behave completely different.

Notes

EPFL

Embedded system design

Rev. 1.0 – 3.12

Notes

Profiling

Which information we require to have?

- ▶ On fixed systems we are only interested in the number of cpu-cycles burned, as we cannot change the underlying architecture.
- ▶ This is very often accomplished by using *performance counters*. Performance counters are hardware counters that count clock-cycles (your i3/i5/i7 for example has such counters build in).
- ▶ In SOC design we have the liberty to modify the architecture and the software.
- ▶ Hence here we are often also interested in more hardware specific parameters as:
 - ▶ Bus occupation
 - ▶ Cpu stall cycles
 - ▶ Cache hit/miss ratio
 - ▶ Cache trashing latency's
 - ▶ ...
- ▶ Also this can be accomplished with performance (hardware) counters.

EPFL

Embedded system design

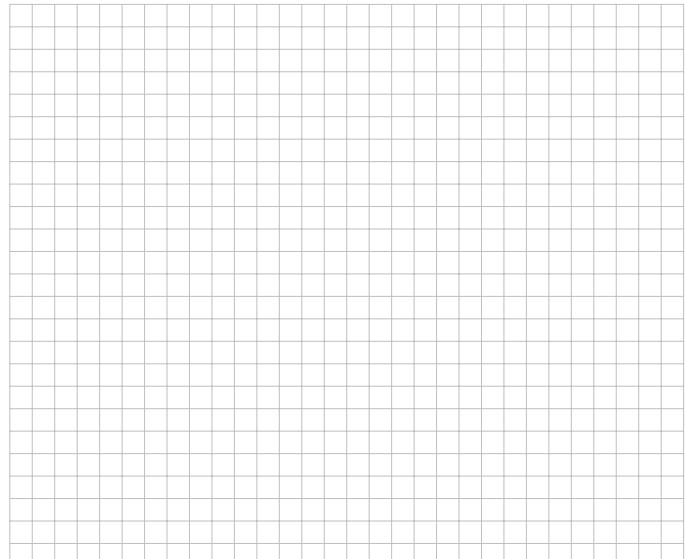
Dr. Theo Kluter

Custom Instructions

Profiling

Rev. 1.0 - 3.12

Notes



Profiling

Limitation of performance counters

- ▶ Of course performance counters are limited by the number of bits they have (hence the "time" they can measure).
- ▶ Furthermore they take silicon area, this is one of the reasons (when time allows):
 - ▶ To tape out a chip with the performance counters.
 - ▶ To suppress the production chip the performance counters (by using `performance_empty.v`).
- ▶ To be able to profile hardware aspects, the hardware needs to be observable (as in our case where everything is available in Verilog).
- ▶ In many cases this is not the case as some parts are provided as IP-cores (for example an ARM-System), in this case the performance counters can use "models".
- ▶ The sets of models known are:
 - ▶ Worst case.
 - ▶ Typical case.
 - ▶ Best case.
- ▶ These models are often derived from previous taped-out chips.

The EPFL logo is a red rectangular box containing the letters "EPFL" in a bold, white, sans-serif font.

Embedded syst
design

Dr. Theo Kluter

Custom Instructions

Profiling

Page 10 - 31

Notes



Notes

