Lecture 3

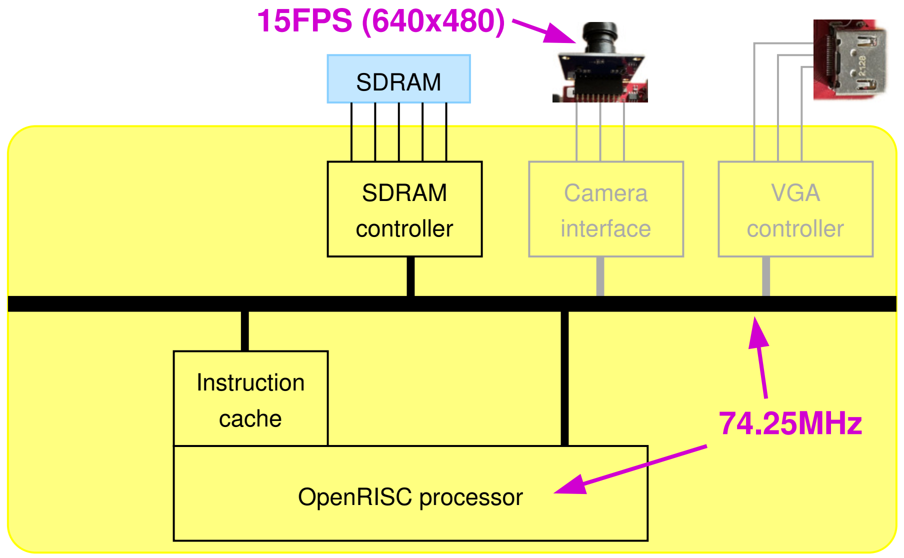
Embedded system design

Custom instructions

CS476 - ESD
March 10, 2024

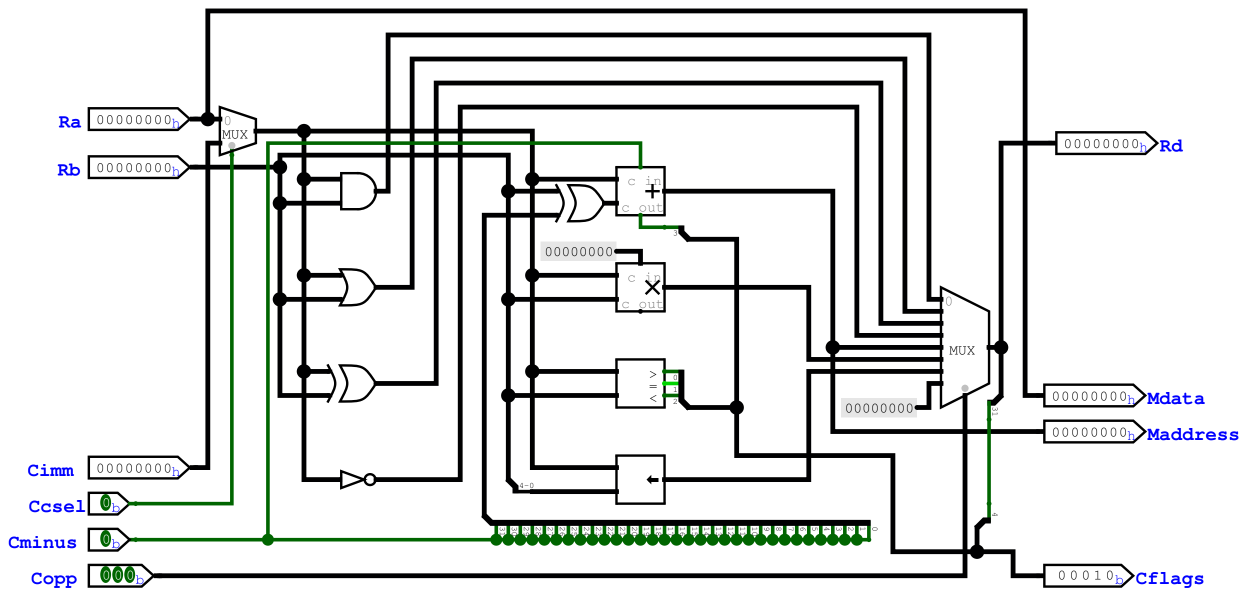
Dr. Theo Kluter
EPFL

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 (CI's)*.
- To understand the concept of custom instructions we have to dive a bit into the architecture of the μC.

Arithmetic Logic Unit



- The Arithmetic Logic Unit (ALU) is the *heart* of the μC.
- It receives two data (`Ra`, `Rb`) from the register-file and produces one result (`Rd`) to the register-file.
- The operation done is selected by the control signals (`Cimm`, `Ccsel`, `Cminus`, `Copp`), that are set depending the instruction.
- Note that only one operation can be selected, although all operations are performed.

Arithmetic Logic Unit

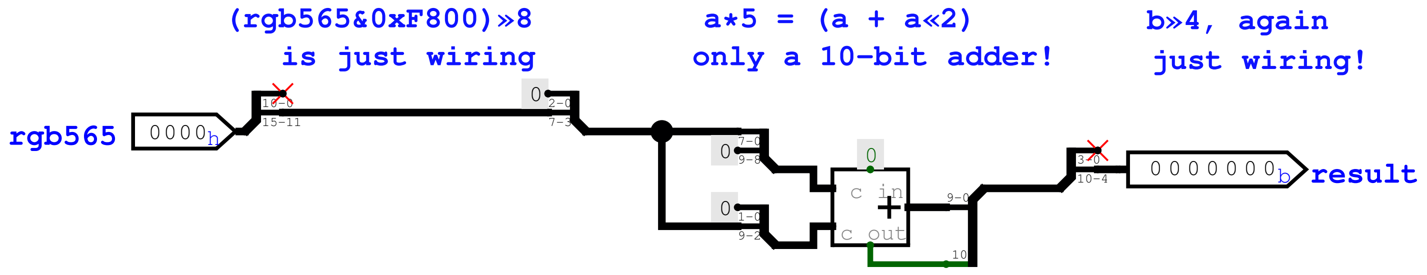
- Assume you have following c-program:

```
result = (((rgb565 & 0xF800) >> 8) * 5) >> 4;
```

- This would result in following four operations on the ALU:

```
l.andi r5,r5,0xF800   # rgb565 & 0xF800
l.sri  r5,r5,8        # (rgb565 & 0xF800) >> 8
l.muli r5,r5,5        # ((rgb565 & 0xF800) >> 8) * 5
l.sri  r5,r5,4        # (((rgb565 & 0xF800) >> 8) * 5) >> 4
```

- However, doing the same thing in hardware is way simpler:



- And this can be executed in a single cycle, a speed-up of 4x!
- This is the basic idea behind the custom instruction (there are of course other applications for it).
- Let's first look into the hardware details.

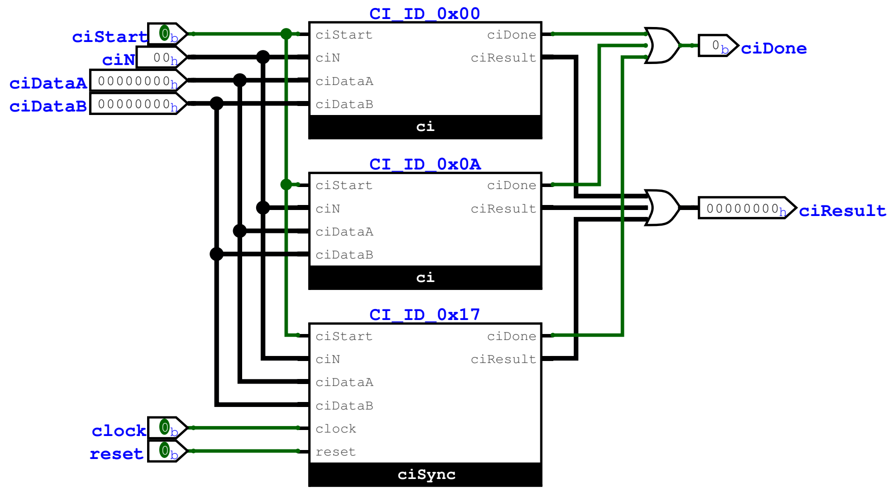
Custom instruction hardware interface

- The minimal set of signals that the μC provides us with to create custom instruction hardware is (note: input/output is from the perspective of the μC):

Name:	**Direction:**	**#bits:**	**Function:**
`ciStart`	output	1	Indicates an active custom instruction.
`ciN`	output	8	The custom instruction identifier code.
`ciDataA`	output	32	The value of register A (`Ra`) going into the ALU/CI.
`ciDataB`	output	32	The value of register B (`Rb`) going into the ALU/CI.
`ciResult`	input	32	The result value to be written to the register file (`Rd`).
`ciDone`	input	1	The signal indicating that the CI performed it's operation.

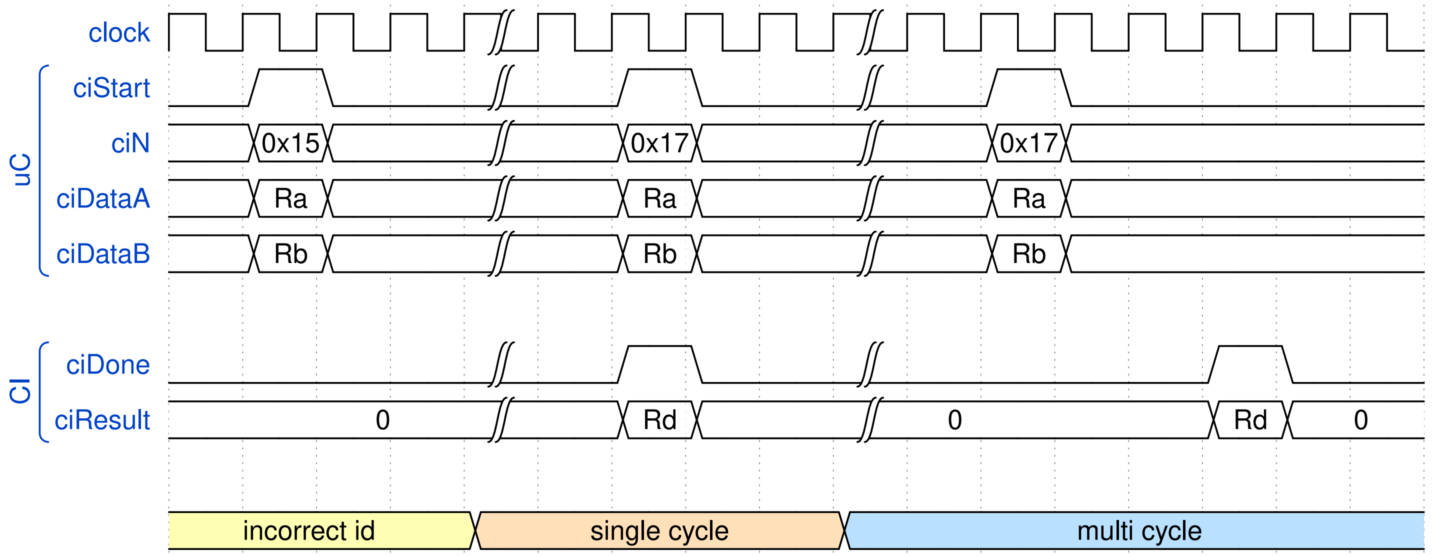
- The `ciDone` signal is a very important signal. If the μC activates a custom instruction by the `ciStart` signal it will wait (stall) till an activation of the `ciDone`. If the `ciDone` is not activated your system will *DEADLOCK*!
- The signal `ciN` indicates which custom instruction is activated. As this signal is 8-bit wide we can implement up to 256 custom instructions.
- So how to combine these different custom instructions in hardware...

Custom instruction hardware architecture



- We can implement multiple custom instructions. Why not "multiplexing" the `ciDone` and the `ciResult` signals by using the `ciN` signal?
- Very simple: multiplexers have more logic as simple or-gates (or and-gates, the alternative)...
- This poses, however, some restrictions that we have to take into account when designing a custom instruction module....
- Let's look into the timing requirements of our custom instruction hardware.

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 μC is stalled!

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`!

Custom instruction usage

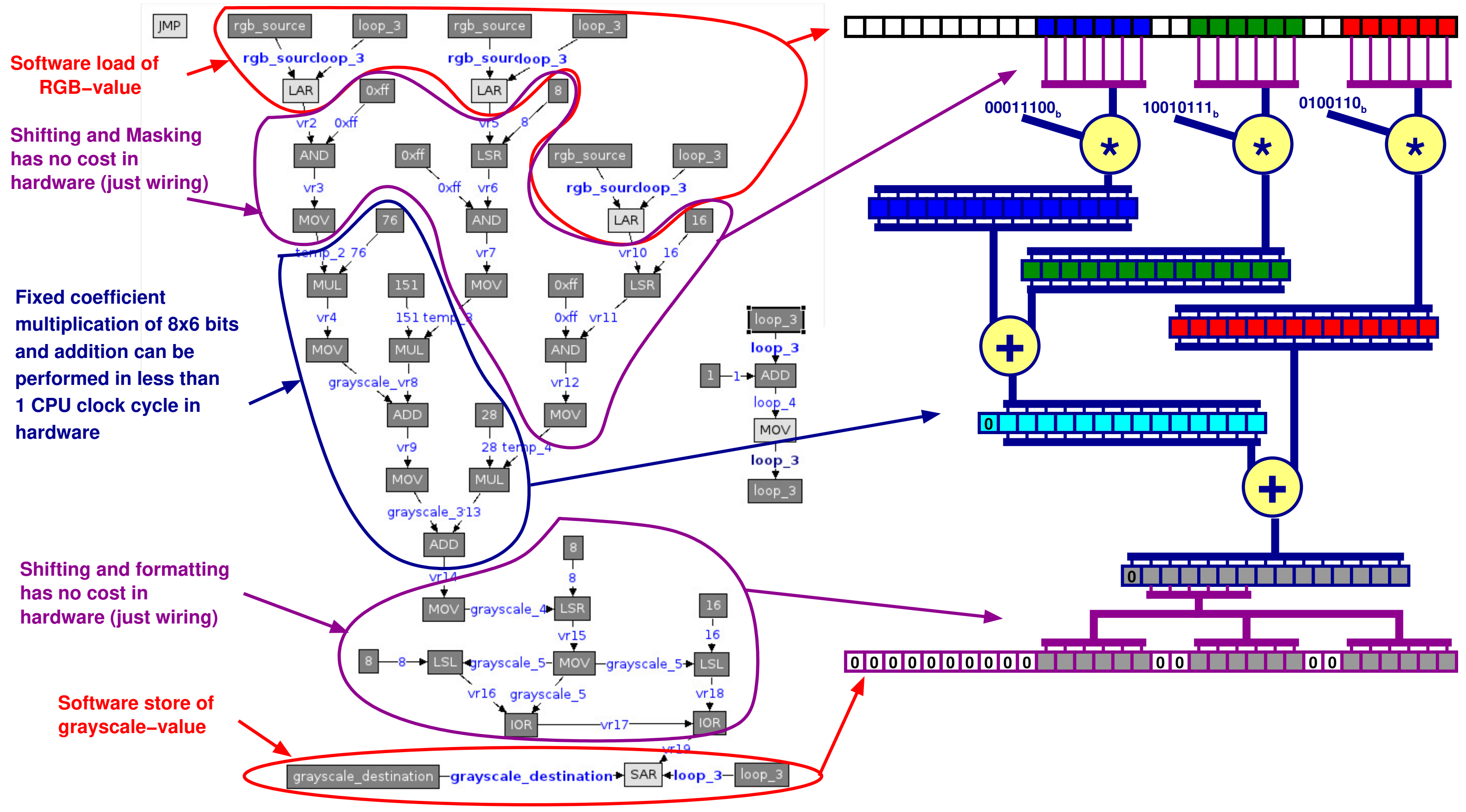
- The question is now: How to use custom instructions?
- Let’s take a design example (important: this is not the grayscale conversion used in our system!):

```
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:

Custom instruction usage (DFG)


Software load of RGB–value
Shifting and Masking has no cost in hardware (just wiring)
Fixed coefficient multiplication of 8x6 bits and addition can be performed in less than 1 CPU clock cycle in hardware
Shifting and formatting has no cost in hardware (just wiring)
Software store of grayscale–value
JMP
rgb_source
loop_3
rgb_sourcloop_3
LAR
0xff
AND
MOV
LSR
MUL
ADD
IOR
SAR
grayscale_destination
00011100_b
10010111_b
0100110_b

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

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 gives 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 “know architectures”, maybe not for the system you are targeting.

Profiling

Profiling has it’s 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 it’s 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 been done correctly, otherwise the real SOC can behave completely different.

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/i/ 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.

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.