CO503 - ADVANCED EMBEDDED SYSTEMS
Practical 2 - Processor Customizations
Group 12 - E/19/057, E/19/063

In this practical, we utilized FPGA design tools to create a System-on-Chip (SoC) featuring a customized Nios II Processor. This customization was undertaken to enhance the performance of the modulo-2 division operation, particularly within the context of the Cyclic Redundancy Check (CRC) algorithm commonly employed in network devices.

Adding a custom instruction to the Nios II Processor

Nios II custom instructions are custom logic blocks placed adjacent to the ALU in the processor's datapath. Each custom operation is assigned a unique selector index, allowing software to specify the desired operation among a maximum of 256 custom operations. This index is determined during hardware instantiation through Qsys, which exports the selection index value to Nios II software build tools. These tools generate a macro that can be directly used in C or C++ application code, abstracting the details of the custom instruction from the software developer.

There are multiple types of Custom Instruction Types available to be implemented using the Nios II custom instruction software interface,

Type	Application
Combinational	Single clock cycle custom logic blocks.
Multicycle	Multi-clock cycle custom logic blocks of fixed or variable durations.
Extended	Custom logic blocks that can perform multiple operations.
Internal register file	Custom logic blocks that access internal register files for input or output or both.

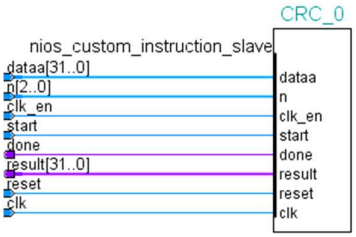
In this lab for our Cyclic Redundancy Check Custom Instruction we used an extended type CI interface because our CI block needs to implement several different operations which will occupy multiple select indices as follows,

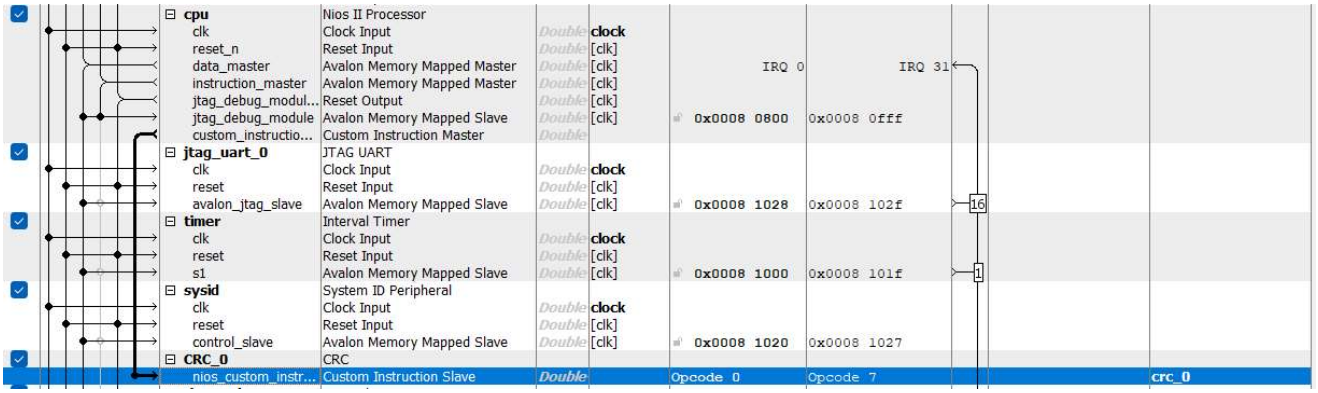
```
n = 0, Initialize the custom instruction to the initial remainder value
n = 1, Write  8 bits data to custom instruction
n = 2, Write 16 bits data to custom instruction
n = 3, Write 32 bits data to custom instruction
n = 4, Read  32 bits data from the custom instruction
n = 5, Read  64 bits data from the custom instruction
n = 6, Read  96 bits data from the custom instruction
n = 7, Read 128 bits data from the custom instruction
```

And the CRC CI is a combinational (meaning that it gives the output within one clock cycle) custom instruction. And only depend on the given selection index,

```
assign done = (n>3)? done_delay : start;
```

Within QSys we can implement our CRC CI as following,





As you can see in the above screenshot, Selection indices (Opcodes) assigned for our CRC CI module is from 0-7. As such Nios II BSP generator will define a macro for our CI in the system header *(system.h)*

```
/*Custom instruction macros*/

#define ALT_CI_CRC_0(n,A) __builtin_custom_ini(ALT_CI_CRC_0_N+(n&ALT_CI_CRC_0_N_MASK),(A))
#define ALT_CI_CRC_0_N 0x0
#define ALT_CI_CRC_0_N_MASK ((1<<3)-1)
```

Here, `ALT_CI_CRC_0_N` marks the beginning of our selection indices and followed by `ALT_CI_CRC_0_N_MASK` to indicate that our CI will occupy the following 7 addresses, and within our macro `__builtin_custom_ini` indicate that our CI block expects a integer as input (dataa) and produces a integer as output (result) and we can use our macro as `ALT_CI_CRC_0(n,A)` by providing the *n* as the selection index (opcode) and *A* as the input operand.

Observations

Within our main program we will carry out the Cyclic Redundancy Check for the same generated random dataset using 3 separate methods,

> ***method 1:*** *Iterative modulo 2 division in S/W*
> ***method 2:*** *Using a look-up table (optimized)*
> ***method 3:*** *Using the custom instruction (XOR and shift in H/W, in parallel)*

After we compare multiple aspects like the time taken and throughput of each method.

```
Validating the CRC results from all implementations
---------------------------------------------------
All CRC implementations produced the same results


Processing time for each implementation
---------------------------------------
Software CRC = 437.538 ms
Optimized software CRC = 294.637 ms
Custom instruction CRC = 5.382 ms


Processing throughput for each implementation
---------------------------------------------
Software CRC = 0.15 Mbps
Optimized software CRC = 0.22 Mbps
Custom instruction CRC = 12.18 Mbps


Speedup ratio
-------------
Custom instruction CRC vs software CRC = 81.29
Custom instruction CRC vs optimized software CRC = 54.74
Optimized software CRC vs software CRC= 1.49
```

Comparison of different CRC methods (Revised to apply floating point arithmetic patch)

As you can see, we have achieved the main objective of implementing our own CI to carry out the cyclic redundancy check operation by getting a considerable speed up in the processing time when compared to other software-based methods.

~~Upon closer inspection, you'll notice some contradictory results in our output. For instance, the Optimized CRC approach demonstrates significantly longer processing times and decreased throughput compared to the standard CRC approach. Throughout our lab sessions, we delved into possible reasons for these observations. The most plausible explanation we arrived at involves potential errors in data representation within the system (specifically, *alt_u32* unsigned 32 bit), possibly leading to truncated or overflowed values. However, without further research, this conclusion remains unproven~~. (This section was revised)

Throughout this lab we also ran into multiple other obstacles we successfully able overcome. Including,

- Although the SoC design was supposed to be done using the extended design which included multiple clock domains and a SDRAM interface which we used the JPEG Encoder, despite numerous attempts, we were unable to achieve successful operation, prompting us to revert to the previous simple SoC design which only used On-chip memory.
- Since we needed a high precision timer for timestamp generation, we tried to adapt the same timer for both sys_clk_timer (Main timer used by the SoC to generate periodic interrupts) and timestamp_timer (for high resolution time measurement). But this design led to non-responsive executions and failed builds. Thus, we had to resort to using a separate alternative timer with microsecond precision for our timestamp generation needs. This may have been due to some other of the SoC components not being able to work with microsecond precision during their normal operation.

Revision

Upon further inspection, we found that the discrepancy in the results occurred due to integer division. The operands were cast as **unsigned long** when evaluating the expression, causing the fractional part of the result to be discarded, thus leading to erroneous values. This finding confirms that our previous suspicions were indeed correct. We successfully addressed this issue by casting the operands as **floats** to enforce floating-point division. Here's the revised part of code,

```
  // Report processing times
  printf("Processing time for each implementation\n");
  printf("---------------------------------------\n");
  printf("Software CRC = %.3f ms\n", 1000*(float)(sw_slow_timeB-sw_slow_timeA)/alt_t_freq);
  printf("Optimized software CRC = %.3f ms\n", 1000*(float)(sw_fast_timeB-sw_fast_timeA)/alt_t_freq);
  printf("Custom instruction CRC = %.3f ms\n\n\n", 1000*(float)(ci_timeB-ci_timeA)/alt_t_freq);

  printf("Processing throughput for each implementation\n"); // throughput = total bits / (time(s) *
1000000)
  printf("-----------------------------------------------\n");
  printf("Software CRC = %.2f Mbps\n", (8 * NUMBER_OF_BUFFERS * BUFFER_SIZE)/(1000000*(float)(sw_slow_timeB-
sw_slow_timeA)/alt_t_freq));
  printf("Optimized software CRC = %.2f Mbps\n", (8 * NUMBER_OF_BUFFERS *
BUFFER_SIZE)/(1000000*(float)(sw_fast_timeB-sw_fast_timeA)/alt_t_freq));
  printf("Custom instruction CRC = %.2f Mbps\n\n\n", (8 * NUMBER_OF_BUFFERS *
BUFFER_SIZE)/(1000000*(float)(ci_timeB-ci_timeA)/alt_t_freq));

  printf("Speedup ratio\n");
  printf("-------------\n");
  printf("Custom instruction CRC vs software CRC = %.2f\n", ((float)(sw_slow_timeB-
sw_slow_timeA))/((float)(ci_timeB-ci_timeA)));
  printf("Custom instruction CRC vs optimized software CRC = %.2f\n", ((float)(sw_fast_timeB-
sw_fast_timeA))/((float)(ci_timeB-ci_timeA)));
  printf("Optimized software CRC vs software CRC= %.2f\n", ((float)(sw_slow_timeB-
sw_slow_timeA))/((float)(sw_fast_timeB-sw_fast_timeA)));
```