CO503: Practical 2 – Processor Customization

E/17/154

05/03/2022

The practical uses FPGA design tools to create System-on-Chip with a customized NiosII Processor. The customization is done in order to improve the performance of modulo-2 division operation in Cyclic-Redundancy-Check (CRC) algorithm used in network devices.

Tasks:

1. Add a custom instruction to the existing MIPS ISA of the NiosII processor, for performing the modulo-2 division.
2. Implement the required hardware functionality to support the custom instruction using XOR and shift operations.
3. Use the newly added custom instruction in the CRC algorithm and compare its performance against pure software implementations.

Implementation

The hardware implementation was done using IP blocks with minimal component to support the CRC custom instruction.

Used IP blocks,

IP BLOCK	Usage
clk	clk / reset signals for the SoC
timer0	For interval timer operation
high_res_timer	To calculate time intervals between implementations (software/hardware)
sysid	Gives a unique ID to the SoC (prevents accidental downloading of software)
cpu	Processor for the SoC
jtag_uart	UART connection with the computer
onchip_mem	Data/instruction code save

Table01: Used IP Blocks

Hardware

After creating the base system using above components on Qsys we have to implement a new component to handle the custom instruction.

The component is built using the provided Verilog files as **CRC_Custom_Instruction.v** & **CRC_Component.v.** Using the files we can create a single interface with the name

Nios_custom_instruction_slave

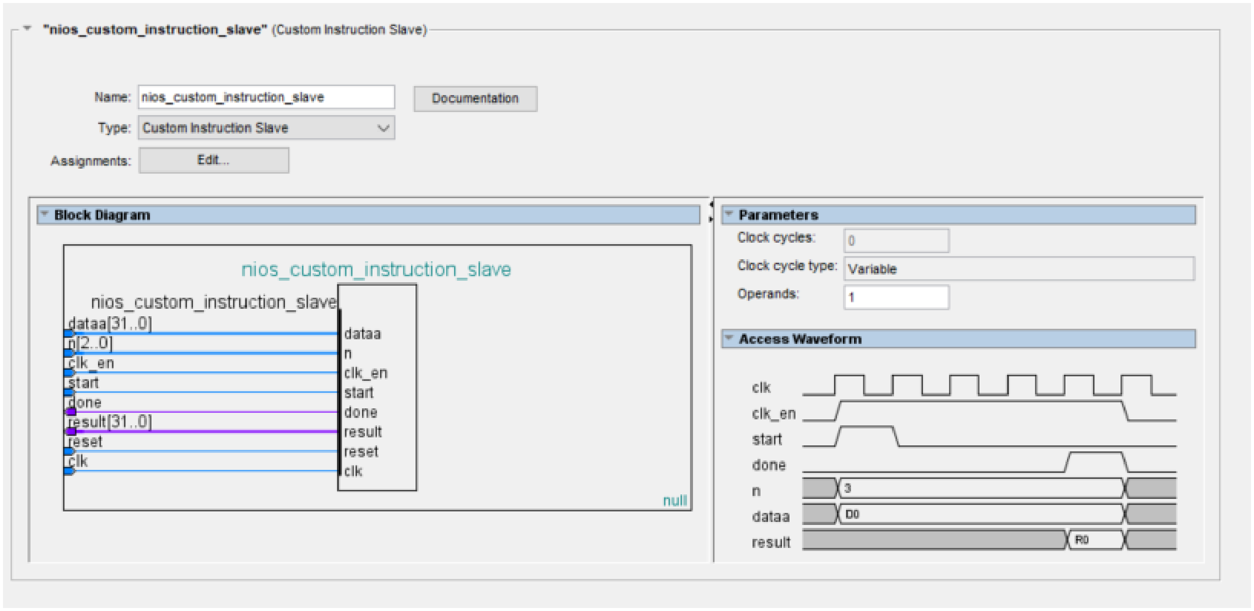
Of the type Custom Instruction Slave

As parameter the clock cycles are set to 0 and the clock cycle type set to Variable with operands as 1.

After defining the interface the signals were set with the interface signal types.

After the creating the component we can make the connection between its Custom instruction slave to CPU's custom instruction master.

Then the BDF file can be completed by importing the newly generated SoC.


"nios_custom_instruction_slave" (Custom Instruction Slave)

Name: nios_custom_instruction_slave

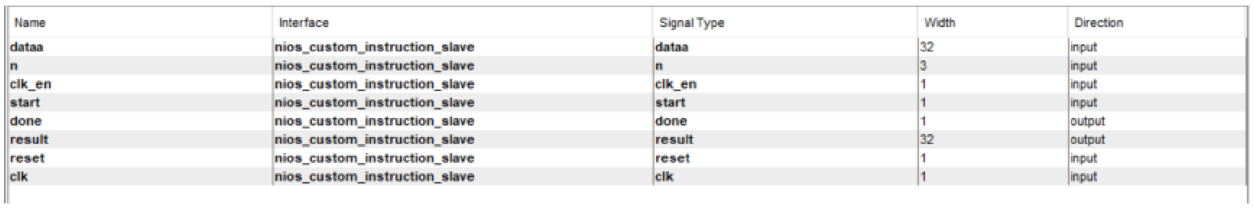
Type: Custom Instruction Slave

Clock cycles: 0

Clock cycle type: Variable

Operands: 1

Fig01: CRC Custom instruction Interface



Name	Interface	Signal Type	Width	Direction
dataa	nios_custom_instruction_slave	dataa	32	input
n	nios_custom_instruction_slave	n	3	input
clk_en	nios_custom_instruction_slave	clk_en	1	input
start	nios_custom_instruction_slave	start	1	input
done	nios_custom_instruction_slave	done	1	output
result	nios_custom_instruction_slave	result	32	output
reset	nios_custom_instruction_slave	reset	1	input
clk	nios_custom_instruction_slave	clk	1	input

Fig02: Signals of CRC Custom Instruction

Software

For the software part after initializing the project provided files have to be imported.

File	Description
ci_crc.h	Header file for ci_crc.c
ci_crc.c	Access the CRC custom instruction
crc.h	Header for crc.c
crc.c	Software CRC algorithm run by the Nios II processor.
crc_main.c	Main program that populates random test data and execute CRC in both software and custom instruction ,validates output and reports processing time.

Table02: Imported Files

- Changes to be done would be setting the **high_res_timer** as the **timestamptimer** and **timer0** as **sys_timer**.
- Once the BSP is generated we can build the project.
- Here we get our first error signifying undefined CRC_MACRO(n,A)
- This is the function called to handle the CRC custom instruction on the hardware level.
- By looking through the error we can see that the definition of this function must be changed to the one provided in the system.h .
- The system.h file is generated along with the SoC when using Qsys.
- Going through the system.h file we get to see the commented CRC custom instruction part and the definition for the ALT_CI_CRC_CUSTOM(n,A)
- Since this is what needs to be accessed using the the CRC_MACRO(n,A) we can copy ALT_CI_CRC_CUSTOM(n,A) and paste it at the definition in the ci_crc.c

Additional errors

1) Errors when building found in BSP ->

These errors occurred because of redefining already defined variables that are already in the bsp. Since the bsp(board support package) always tries to optimize the software bsp gives out errors. Removing the redefinitions and cleaning and building the bsp solves this.

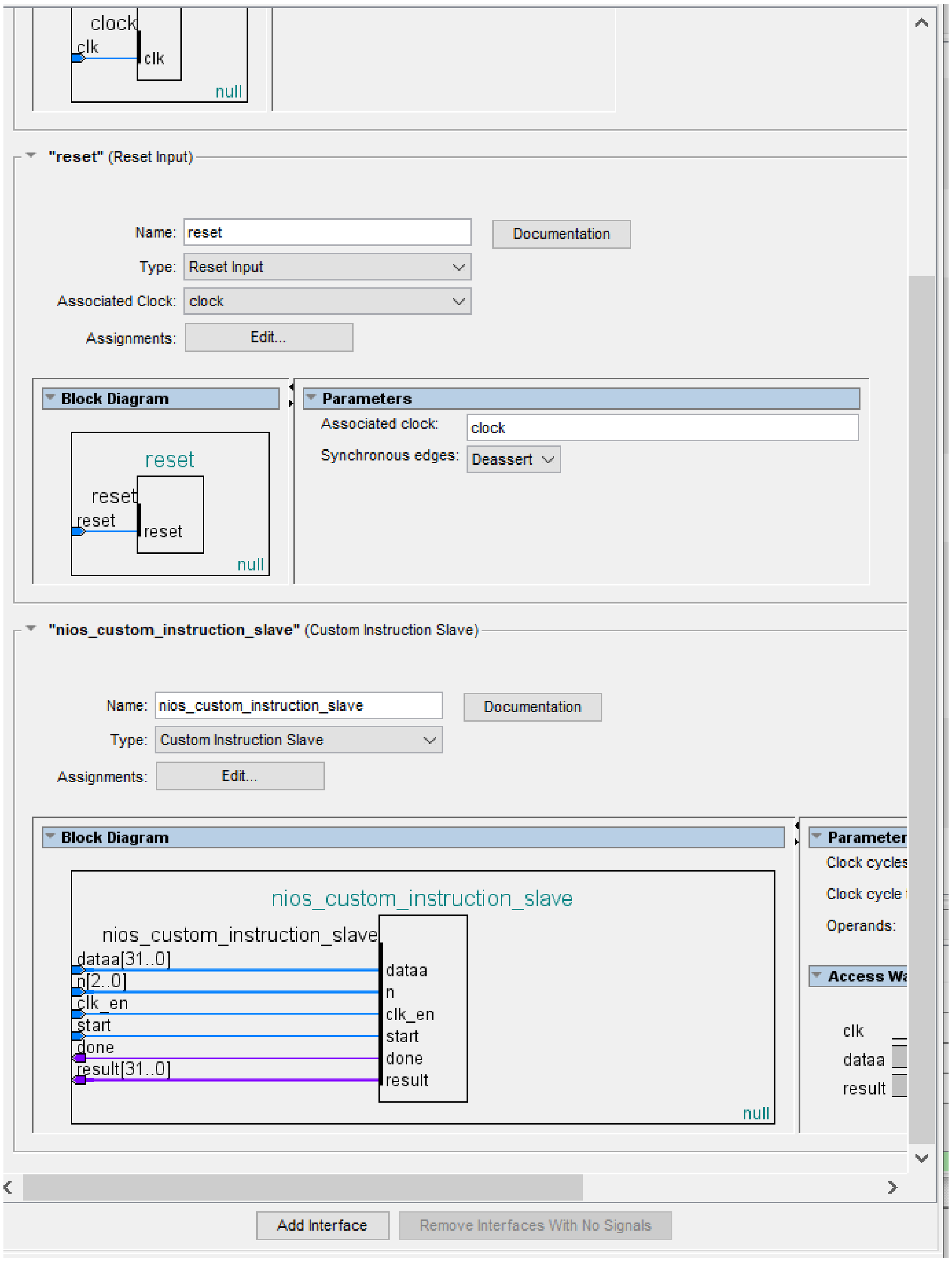
2) Errors on memory allocation ->

My first build I had used the default settings for onchip mem. This memory is not enough. Allocating a memory of 256KB solves this issue.

3) Output Error when run ->

After resolving above errors and running the code on hardware the run report gave out an error “CRC outputs not same”.Going through the message shows that CRC custom instruction outputs -> 0x0

While software implementation gives different output. Going back to QSYS recognized that my implementation of custom instruction has three interfaces, CLK RESET and the custom instruction slave while the manual has these in one. Looking at the signals the CLK and RESET receive the signals separately rather than from the nios_custom_instruction_slave. Removing these extra interfaces by assigning the correct signals corrected the error.



"reset" (Reset Input)

Name: reset

Type: Reset Input

Associated Clock: clock

Assignments: Edit...

Parameters

Associated clock: clock

Synchronous edges: Deassert

"nios_custom_instruction_slave" (Custom Instruction Slave)

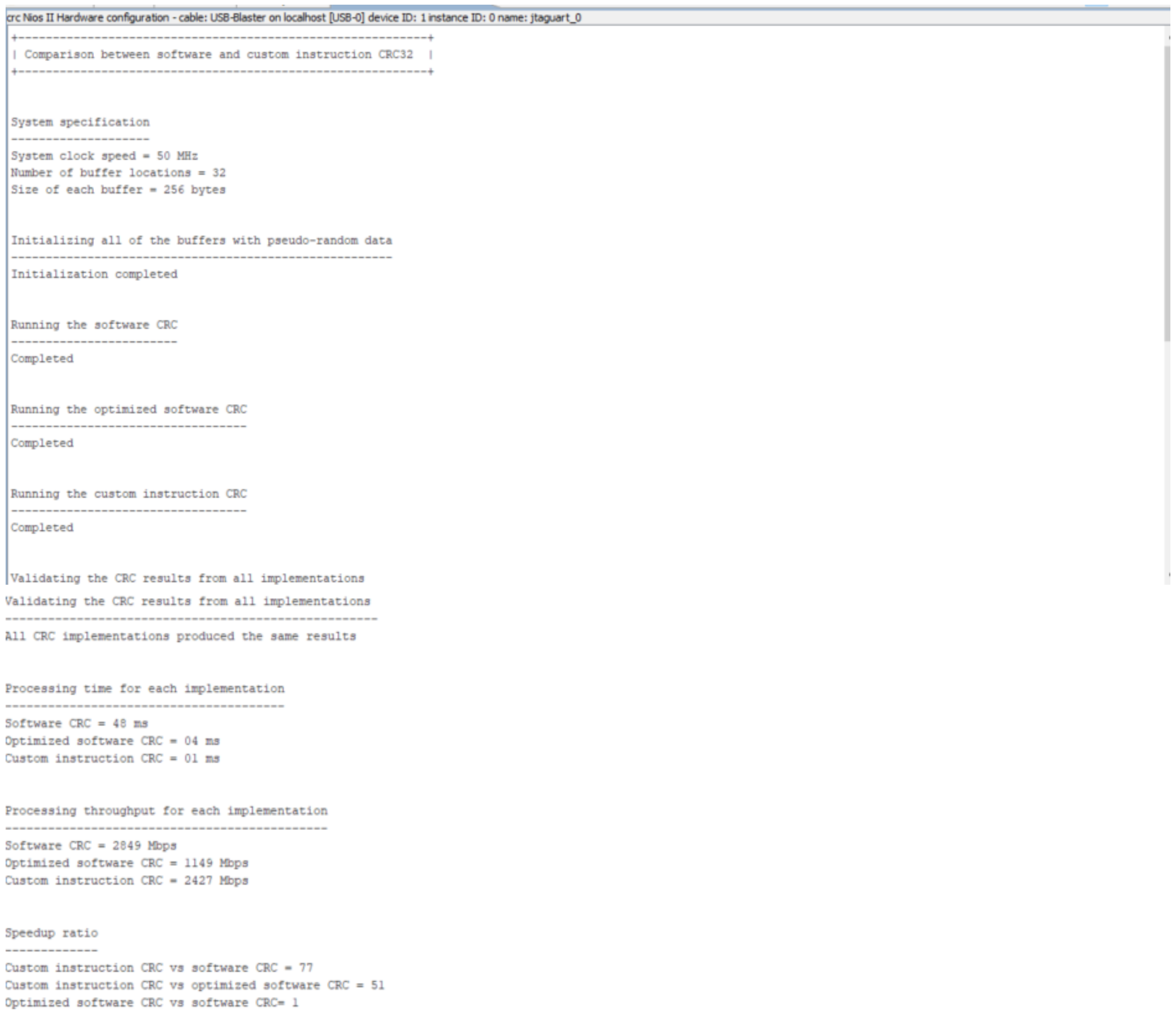
Name: nios_custom_instruction_slave

Type: Custom Instruction Slave

Assignments: Edit...

Fig03: 3 Interface

Observations



crc Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0

```
+----------------------------------------------------------+
| Comparison between software and custom instruction CRC32  |
+----------------------------------------------------------+


System specification
--------------------
System clock speed = 50 MHz
Number of buffer locations = 32
Size of each buffer = 256 bytes


Initializing all of the buffers with pseudo-random data
-------------------------------------------------------
Initialization completed


Running the software CRC
------------------------
Completed


Running the optimized software CRC
----------------------------------
Completed


Running the custom instruction CRC
----------------------------------
Completed


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


Processing time for each implementation
----------------------------------------
Software CRC = 48 ms
Optimized software CRC = 04 ms
Custom instruction CRC = 01 ms


Processing throughput for each implementation
----------------------------------------------
Software CRC = 2849 Mbps
Optimized software CRC = 1149 Mbps
Custom instruction CRC = 2427 Mbps


Speedup ratio
-------------
Custom instruction CRC vs software CRC = 77
Custom instruction CRC vs optimized software CRC = 51
Optimized software CRC vs software CRC= 1
```

<u>Fig04: NiosII Console output</u>

The output reports the system specification

Clock speed = 50Hz

Number of buffer locations = 12

Size of each byte buffer = 256Bytes

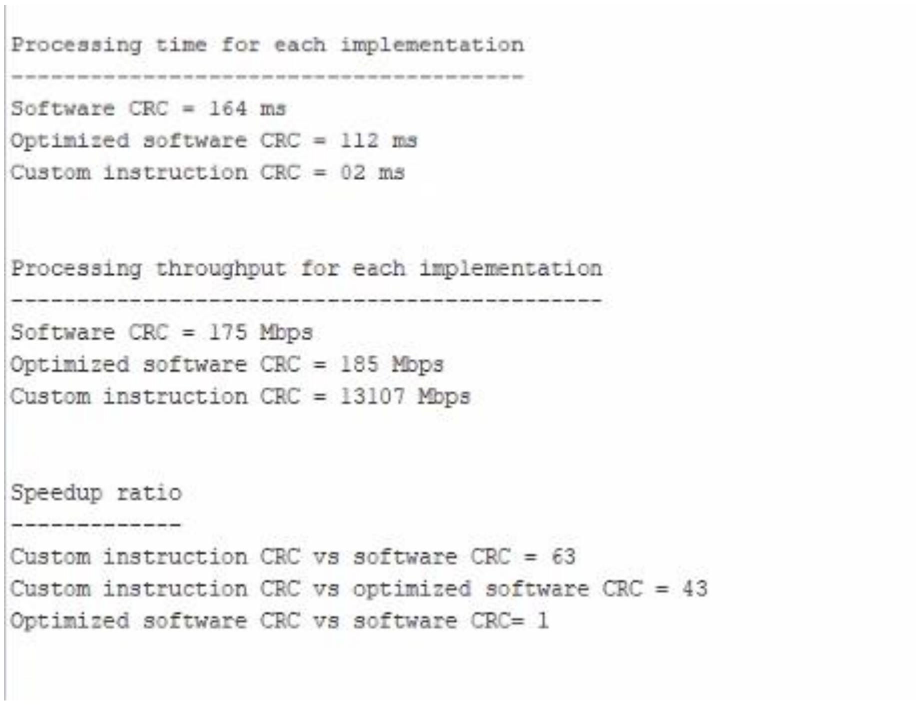
Shows the validation of each output of each method is the same.

Processing time for each implementation

Shows that the unoptimized software CRC is much slower than the optimized software CRC which is expected, as the optimized implementation uses a lookup table to access already calculated middle values. The custom instruction seems to be faster than the optimized implementation with a processing time of around 1 ms , which is a significant improvement.

Processing throughput for each implementation

The software implementation has a highest throughput followed by the custom instruction and with the least throughput optimized software implementation. Usually, this metric shows the throughput during a calculation. My implementation of the CRC algorithm shows that the software throughput has become higher than the custom instruction. Usually this is the other way around. Since the speedup ratios are same as expected results we can assume the above results are a result of some deviation in the acquired data paths.



```
Processing time for each implementation
---------------------------------------
Software CRC = 164 ms
Optimized software CRC = 112 ms
Custom instruction CRC = 02 ms


Processing throughput for each implementation
---------------------------------------------
Software CRC = 175 Mbps
Optimized software CRC = 185 Mbps
Custom instruction CRC = 13107 Mbps


Speedup ratio
-------------
Custom instruction CRC vs software CRC = 63
Custom instruction CRC vs optimized software CRC = 43
Optimized software CRC vs software CRC= 1
```

Fig05: Implementation results using SDRAM memory

By looking at an implementation done using the SDRAM rather than the onchip memory as I have we see how different the results are when it comes to speed of the calculation and Procssing throughput. Here since we are accessing memory outside the SOC there is a significant calculation delay in reaching the software implemenataion.