Computer Architecture - Lab Assignment 5 [*]

Nios II processor with customized architecture for a software application

Contents

[*]AC 40969 - https://www.english.ulpgc.es/

1 Introduction

The aim of an error detection technique is to enable the receiver of a message transmitted through a noisy (error-introducing) channel to determine whether the message has been corrupted [6]. To do this, the transmitter constructs a value (called a checksum) that is a function of the message, and appends it to the message. The receiver can then use the same function to calculate the checksum of the received message and compare it with the appended checksum to see if the message was correctly received. For example, if we chose a checksum function which was simply the sum of the bytes in the message mod 256 (i.e. modulo 256), then it might go something as follows. All numbers are in decimal.

- Message : 6 23 4
- Message with checksum : 6 23 4 33
- Message after transmission : 6 27 4 33

The second value of the message sent has been corrupted, changing from 23 to 27. Therefore, by comparing the sum of the values received (37) to the checksum previously calculated (33), it is clear that the original data has been altered.

As in the previous example, the Cyclic Redundancy Check (CRC) error detection algorithm leaves the data transmitted intact and appends a checksum at the end:

<original message> <checksum>.

In this project, the CRC-32 will be the only algorithm to be implemented and analyzed.

2 Objectives

The main objective of this course project is analyzing and measuring how specialized instruction sets reduce the execution time of certain test programs. To achieve this, the project will be divided in three main tasks:

- Hardware Engineering
  - This task consists in understanding the high-level hardware design language *Verilog*. Specifically, the code contained in the *.v files 'CRC_Custom_Instruction.v' and 'CRC_Component.v'.
  - The circuit schematics equivalent to the Verilog source code in these two files will be generated and explained.

- Computer Architecture
  - This second task addresses the concepts related to the functional unit integrated into the data path of the Nios II/f processor, which will allow the use of custom instructions.
  - Once this functional unit is configured and the use of custom instructions can be made, the next step will be to measure and study the performance improvement (Speed-Up) derived from it. For example, the CPU time will be measured:

    $$t_{CPU} = N * CPI/f$$

    (N = number of instructions executed, CPI = cycles per instruction, f = clock frequency).
- Software Engineering
  - The main objective of this last task is to analyze the CRC-32 algorithm (operations, data structures, data hazards, ...) and to determine which are its most costly operations, which will be achieved thanks to software profiling.
  - The source codes will be compiled and load, generating an executable file that then will be run using the DE0-Nano board. The execution times of three different implementations of the CRC-32 algorithm are measured: slow, fast and a customized software version.
  - As described in the previous task, performance will be evaluated using CPI, t_{CPU} and Speed-Up.

3 CRC-32 Algorithm

CRC-32 is a checksum/hashing algorithm that is very commonly used in kernels and for Internet checksums [5], due to the fact that it detects a higher percentage of errors than a simple checksum. A message composed of multiple bytes is passed to CRC-32 algorithm. It returns the same message that was introduced with a checksum at the end. In order to achieve a better understanding of the CRC-32, the following paragraphs presents a brief explanation of the slow, simple version of the algorithm.

The algorithm uses the modulo-2 division and starts with a 32-bit (4 bytes) checksum with all bits set (0xFFFFFFFF).

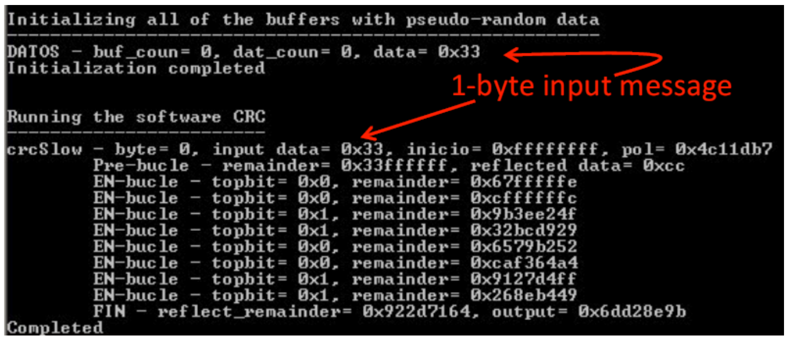
Loop over each byte of the message introduced:

- The byte is taken and all its bits are bit-reflected.

- The byte is shifted to the upper 8 bits of the current 32-bit chekcsum.
- Exclusive-OR operation is executed: checksum = checksum XOR shifted byte.
- Another loop is started, this time over the 8 bits of the byte:
  - If the top (sign) bit of the checksum is set, then:
    - The checksum is shifted up one bit and XOR operation with the magic value (0x04C11DB7) is executed.
  - Otherwise, the checksum is shifted up one bit.
- Repeat the loop until the last byte of the message

When the loop finishes, the entire checksum is bit-reflected. This is the final CRC-32 value.

The following figure shows an execution of the described algorithm:



```
Initializing all of the buffers with pseudo-random data
-------------------------------------------------------
DATOS - buf_coun= 0, dat_coun= 0, data= 0x33
Initialization completed

Running the software CRC
------------------------
crcSlow - byte= 0, input data= 0x33, inicio= 0xffffffff, pol= 0x4c11db7
        Pre-bucle - remainder= 0x33ffffff, reflected data= 0xcc
        EN-bucle - topbit= 0x0, remainder= 0x67fffffe
        EN-bucle - topbit= 0x0, remainder= 0xcffffffc
        EN-bucle - topbit= 0x1, remainder= 0x9b3ee24f
        EN-bucle - topbit= 0x1, remainder= 0x32bcd929
        EN-bucle - topbit= 0x0, remainder= 0x6579b252
        EN-bucle - topbit= 0x0, remainder= 0xcaf364a4
        EN-bucle - topbit= 0x1, remainder= 0x9127d4ff
        EN-bucle - topbit= 0x1, remainder= 0x268eb449
        FIN - reflect_remainder= 0x922d7164, output= 0x6dd28e9b
Completed
```

Figure 1: The Altera Monitor Program window.

In this project, three versions of the CRC-32 algorithm will be studied. One is the slow version, implemented using a modulo-2 division and executed in a standard Nios II processor. The second one is the fast version, implemented using a lookup table and also executed with the standard Nios II. Lastly, the custom instruction version uses a custom instruction created by the programmer and is executed in a modified Nios II processor with custom logic attached to its Arithmetic Logic Unit (ALU).

4 Hardware Engineering

In this course project, the scope of hardware engineering includes the hardware design of computers and their components using a high-level language. We will use the hardware description language called Verilog [4]. This section of project describes the internal organization of the CRC-32 hardware module.

First of all, the Verilog source code of the custom hardware module is contained in the files 'CRC_Component.v' and 'CRC_Custom_Instruction.v'.

The schematic view of the module is shown in the next image:

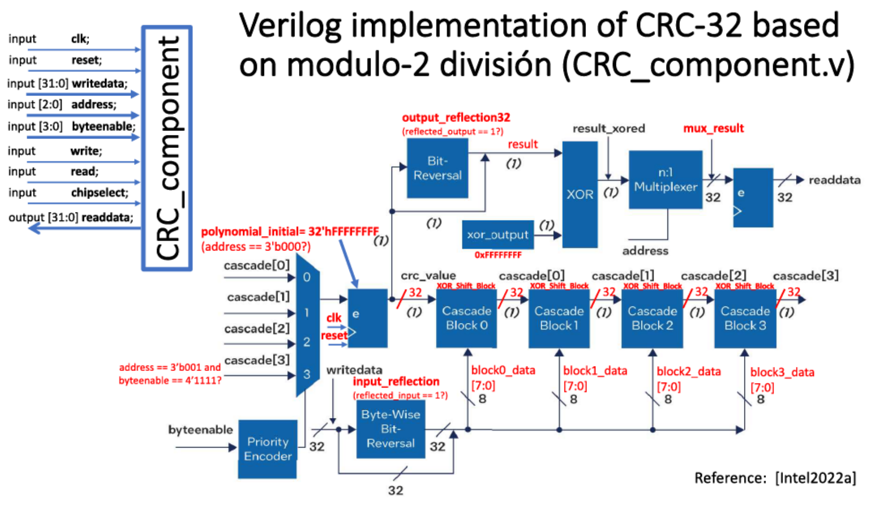


Figure 2: Schematic view of the CRC-32 component.[2]

The Verilog code in CRC_Component.v starts by initializing the module, called CRC_Component, with all its inputs and outputs (clk, reset, address, writedata, byteenable, write, read, chipselect and readdata). Immediately after that, the parameters are also initialized. These parameters allow the programmer to create a module for any CRC standard from 1 to 128 bits. The parameters are:

- crc_width. The width in bits of the CRC result. In CRC-32, for example, this value would be 32. Adjusting it will impact the logic resource usage.

- polynomial_initial. The initial value set for the CRC result register. It must be the the same width as 'crc_width'. By writing any data to address 0, 'polynomial_initial' will be stored in the result register (Figure 2, the element with the letter 'e' in top right corner). That way any previously existing value is cleared.
- polynomial. This is the divisor value used on the input data. Typically shown in polynomial format, the value symbolizes the placement of xor operation on the input data. In synthesis, the polynomial bits that are '1' will create a not gate, whereas the bits that are '0' will simply create a wire. Even with 32 stages of these operations cascaded, the simple logic will not become a significant factor on logic utilization or fmax. This value must be the same width as 'crc_width'.
- reflected_input. Some CRC standards require that all the input bits be reflected around the center point. This option is enabled with '1' and disabled with '0'. Typically this option is enabled or disabled along with 'reflected_output'.
- reflected_output. Some CRC standards require that all the output bits be reflected around the center point. This operation occurs before the final optional xor output step. This option is enabled with '1' and disabled with '0'. Typically this option is enabled or disabled along with 'reflected_input' (to undo the input reversal typically).
- xor_output. This is the value used to perform the xor operation with the result, which is done by the element called 'XOR' at the top of the Figure 2. 'xor_output' value will normally be all zeros or all ones. When it is all zeros, the CRC result remains the same after the xor operation, when it is all ones, the result is inverted.

In the CRC-32 implementation that is being studied, the initial values for all these parameters are:

- crc_width = 32;   - CRC-32 (32 bits)
- polynomial_inital = 32'hFFFFFFFF;
- polynomial = 32'h04C11DB7;
- reflected_input = 1;   - input will be bit-reflected
- reflected_output = 1;   - output will be bit-reflected
- xor_output = 32'hFFFFFFFF;   - CRC result will be inverted

CRC-32 requires bit-reversal for serial devices like ethernet, USB, UART, etc. It is done thanks to a byte-wise bit-reversal (Figure 2, in the lower left corner).

Once the written data has been bit-reflected, each of its bytes is assigned to a block<i>_data, which then enters its respective cascade 'XOR_Shift_Block' (Figure 2, in the middle of the picture). Each of these blocks is, in turn, formed by eight smaller 'XOR_Shift' blocks, that take one bit of the input data each. This is shown in the next image:

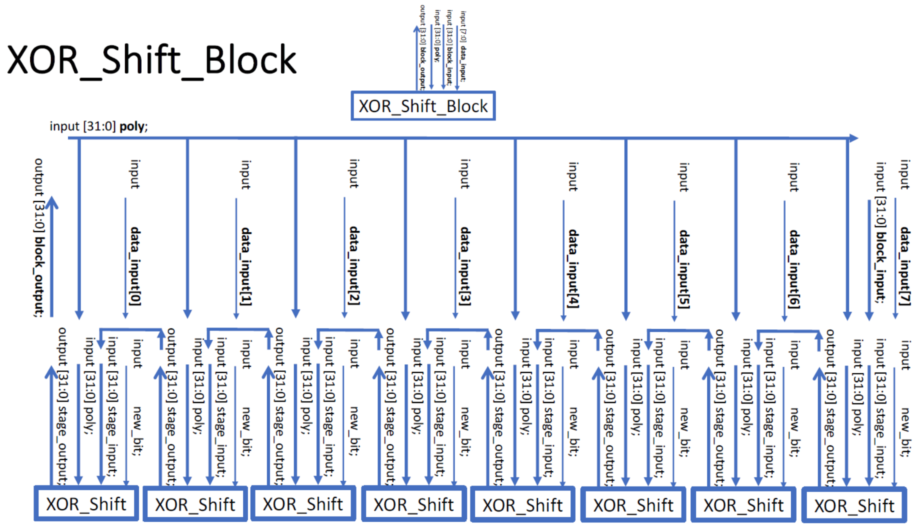
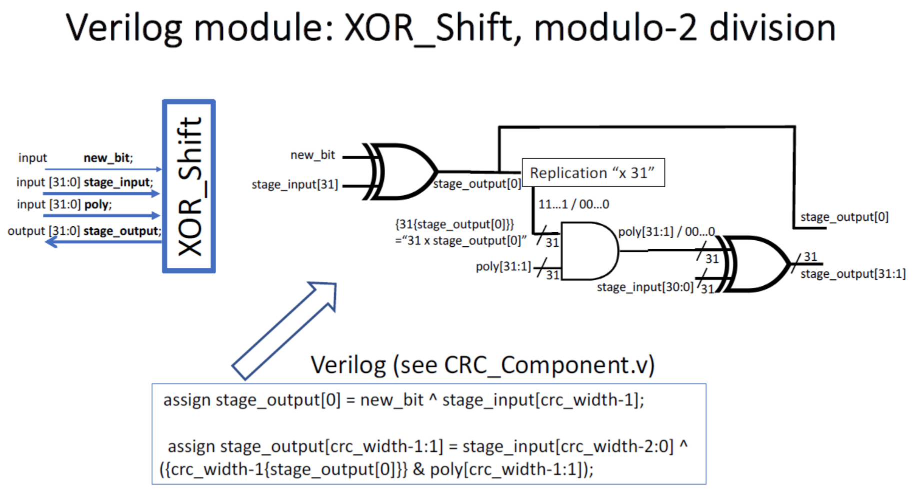


Figure 3: Detailed view of a XOR_Shift_Block.

The 'XOR_Shift' blocks perform the bit stuffing, shifting and XOR operations. The image below gives a more in-depth view of this module:


Verilog module: XOR_Shift, modulo-2 division

```
assign stage_output[0] = new_bit ^ stage_input[crc_width-1];

 assign stage_output[crc_width-1:1] = stage_input[crc_width-2:0] ^
({crc_width-1{stage_output[0]}} & poly[crc_width-1:1]);
```

Figure 4: Detailed view of a XOR_Shift.

The module takes one bit of the input data and is formed by an AND logic gate and two XOR logic gates.

Right after these blocks finish their job, the input data arrives at the output bit-reversal block (Figure 2, in the top left corner). In the case of the CRC-32, the bits are reflected because 'reflected_output' is enabled. Immediately after that, a last XOR operation takes place (Figure 2, between the bit-reversal and the n:1 multiplexer). The operation uses the value in 'xor_output', in this case 0xFFFFFFFF, so the bit-reflected data will be inverted. The result of the XOR operation is stored in the result register, where it can be read when 'read' is set (output[31:0] readdata).

Now addressing the second Verilog file, 'CRC_Custom_Instruction.v', its code is much shorter and simpler in comparison to the first file. In fact, the purpose of this second code is just to instantiate the 'CRC_Component' previously created and wiring it to the custom instruction logic. Additionally, the code associates each of the operations that can be performed by the custom instruction with a 3 bits input called *n*. This will be the first parameter introduced when calling the custom instruction. *n* can take 8 different values:

- 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.

In the code, it can be seen that, depending on the value of *n*, the inputs of the CRC_Component change in order to configure the operation that the custom instruction is being requested to do. For example, when n = 2, 'read' is zero, 'write' is set, 'byteenable' is given the value 4'b0011 and address becomes 3'b001, which means that only two of the cascade blocks are going to be used and the data input is 16 bits long. When n = 7, 'read' is set, 'write' equals zero, 'byteenable' is all zeros and 'address' is given the value 3'b111. This makes the instruction read 128 bits data.

5 Computer Architecture

This section of the course project describes the hardware-software interface of Nios II custom instructions.

Custom instructions allow the programmer to synthesize an algorithm or a code block in a custom hardware logic block. In the Nios II processor, the custom instruction's logic circuits connect directly to the Arithmetic Logic Unit (ALU) in the data path [3], as shown in the following image.

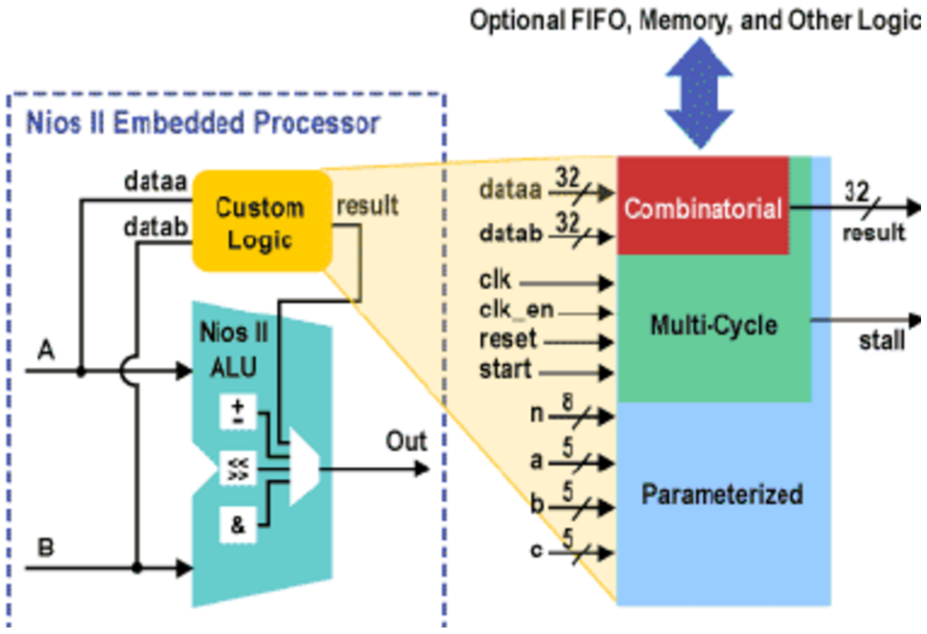
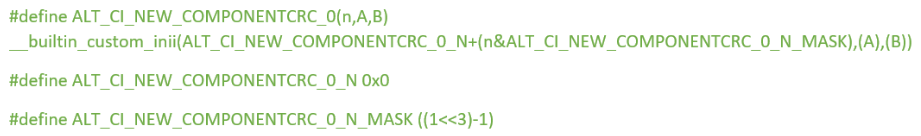


Figure 5: Nios II processor.

Each custom operation is assigned a unique selector index. The selector index allows software to specify the desired operation. The selector index is determined at the time the hardware is instantiated with the Platform Designer or Platform Designer (Standard) software. Platform Designer exports the selection index value to system.h for use by the Nios II software build tools. For each custom instruction, the Nios II Embedded Design Suite (EDS) generates a macro in the system header file, system.h. The macro can be directly used in the C or C++ application code.

The Nios II processor uses GCC built-in functions to map to custom instructions. [1] There are 52 of these functions available. The following example, contained in the file *system.h* illustrates their use:



```
#define ALT_CI_NEW_COMPONENTCRC_0(n,A,B)
__builtin_custom_inii(ALT_CI_NEW_COMPONENTCRC_0_N+(n&ALT_CI_NEW_COMPONENTCRC_0_N_MASK),(A),(B))

#define ALT_CI_NEW_COMPONENTCRC_0_N 0x0

#define ALT_CI_NEW_COMPONENTCRC_0_N_MASK ((1<<3)-1)
```

Figure 6: Custom instruction macros.

In assembly language, the custom instruction can be accessed using normal syntax:

custom <selection index>, <destination>, <source A>, <source B>

<selection index> = The 8-bit number that selects the particular custom instruction.

<destination> = The register where the result from the result port will be placed.

<source A> = The register that provides the first input argument from the dataa port (if any).

<source B> = The register that provides the first input argument from the datab port (if any).

Furthermore, registers within the custom instruction component and Nios II registers can be differentiated using one of two formats:

- r<i> for Nios II register <i>
- c<i> for custom register <i> (internal register)

The software implementation of the CRC-32 algorithm is provided in the following files:

- crc_main.c - Main program that creates random test data and executes the three versions CRC-32 algorithm (slow, fast and customized).
- crc.c - Contains the two software versions of the algorithm, which correspond to the methods crcSlow() and crcFast().
- crc.h - Header file for crc.c.

- ci_crc.c - Program that accesses the custom instruction.
- ci_crc.h - Header file for ci_crc.c.
- crc_main.elf - Executable file that is generated once the program is compiled and loaded.

Now addressing the hardware implementation of custom instructions, the Nios II customized processor has been designed using the FPGA design framework called *Quartus*, from *Intel*. It has been written in the hardware description language called *Verilog*. The source code corresponding to this implementation is located in the files 'CRC_Custom_Instruction.v' and CRC_Component.v'. Additionally, the configuration files for the DE0-Nano board, necessary for the execution of the custom instruction version of the CRC-32 algorithm, are *AC_CI_CRC.sof* and *AC_CI_CRC.sopcinfo*.

6 Software Engineering

This section of the project describes the design of programs that implement the CRC-32 algorithm. Three types of programs were developed:

- Slow version
- Fast version
- Custom instruction version

6.1 Slow CRC-32 software implementation, *crcSlow()*

The source code corresponding to this version of the CRC-32 is contained in the file 'crc.c'.

The algorithm uses the modulo-2 division and starts with a 32-bit (4 bytes) checksum with all bits set (0xFFFFFFFF).

Loop over each byte of the message introduced:

- The byte is taken and all its bits are bit-reflected.
- The byte is shifted to the upper 8 bits of the current 32-bit chekcsum.
- Exclusive-OR operation is executed: checksum = checksum XOR shifted byte.
- Another loop is started, this time over the 8 bits of the byte:
  - If the top (sign) bit of the checksum is set, then:
    - The checksum is shifted up one bit and XOR operation with the magic value (0x04C11DB7) is executed.
  - Otherwise, the checksum is shifted up one bit.
- Repeat the loop until the last byte of the message

When the loop finishes, the entire checksum is bit-reflected. This is the final CRC-32 value.

For further studying the slow implementation, I have written a simplified assembly language code equivalent to the C version. It is shown in the two pictures below:

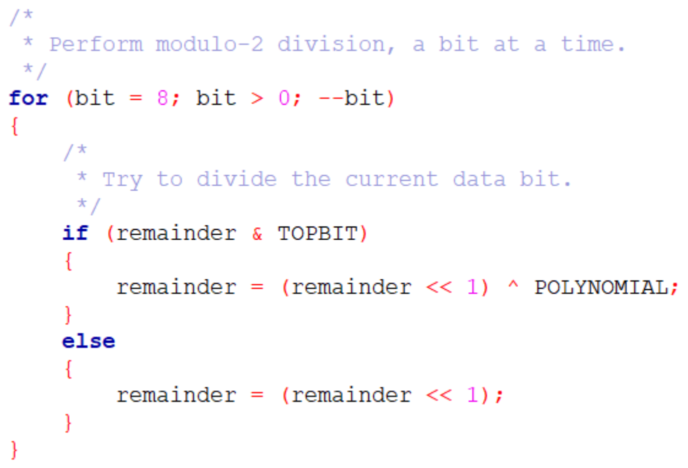
```
.global crcSlow

// The parameters of reflect() are in the registers r1 and r2
// and the result is returned in the register r3

reflect:    subi sp, sp, 20
            stw r4, 0(sp)
            stw r5, 4(sp)
            stw r6, 8(sp)
            stw r7, 12(sp)
            stw ra, 16(sp)

            add r3, r0, r0      // reflection = 0x0;

            add r4, r0, r0      // bit = 0
loop:       andi r5, r1, 0x01   // data(r1) & 0x01
            beq r5, r0, afterif // if(data & 0x01)

if:         subi r6, r2, 1      // nBits - 1
            sub r7, r6, r4      // (nBits - 1) - bit
            slli r7, r7, 1      // 1 << ((nBits - 1) - bit)
            or r3, r3, r7   // reflection |= (1<<((nBits-1)-bit));

afterif:    srli r1, r1, 1      // data = (data >> 1)
            addi r4, r4, 1      // ++bit
            blt r4, r2, loop    // for(bit=0; bit<nBits; ++bit)

            ldw r4, 0(sp)
            ldw r5, 4(sp)
            ldw r6, 8(sp)
            ldw r7, 12(sp)
            ldw ra, 16(sp)
            addi sp, sp, 20

            ret
```

Figure 7: *reflect()* equivalent in assembly language.

```
crcSlow:    movia r4, message
            movia r5, nBytes
            ldw r5, 0(r5)       // r5 = nBytes
            movia r6, INITIAL_REMAINDER
            ldw r7, 0(r6)       // remainder = INITIAL_REMAINDER
            movia r9, WIDTH
            ldw r9, 0(r9)       // r9 = WIDTH
            movia r14, TOPBIT
            ldw r14, 0(r14)     // r14 = TOPBIT
            movia r10, POLYNOMIAL
            ldw r10, 0(r10)     // r10 = POLYNOMIAL
            movia r15, FINAL_XOR_VALUE
            ldw r15, 0(r15)     // r15 = FINAL_XOR_VALUE

            add r8, r0, r0      // byte = 0
            subi r11, r9, 8     // WIDTH - 8
loop1:      ldb r1, 0(r4)       // message[byte]
            addi r2, r0, 8      // second parameter for reflect()
            call reflect        // reflect(message[byte],8)
            sll r12, r3, r11    // reflect(message[byte],8)<<(WIDTH-8);
            xor r7, r7, r12     // remainder ^= ...

            addi r13, r0, 8     // bit = 8
loop2:      and r16, r7, r14    // remainder & TOPBIT
            bne r16, r0, if1    // if(remainder & TOPBIT)

else1:      slli r7, r7, 1      // remainder = remainder << 1
            br end

if1:        slli r17, r7, 1     // remainder << 1
            xor r7, r17, r10    // remainder = (remainder<<1)^POLYNOMIAL;

end:        subi r13, r13, 1    // --bit
            bgt r13, r0, loop2  // for(bit = 8; bit > 0; --bit)
            addi r4, r4, 1
            addi r8, r8, 1      // ++byte
            blt r8, r5, loop    // for(byte = 0; byte < nBytes; ++byte)
            add r1, r7, r0  // first parameter of reflect() = remainder
            call reflect        // reflect(remainder,8)
            xor r18, r3, r15    // (reflect(remainder,8)^FINAL_XOR_VALUE)

STOP:       br STOP
```

Figure 8: *crcSlow()* equivalent in assembly language.

It is clear the the majority of the operations used by this version of the CRC-32 algorithm are arithmetic-logic operations, specifically XOR and left shift. There are several RAW (Read After Write) dependencies, as between the instructions 'sll r12, r3, r11' and 'xor r7, r7, r12', for example. These dependencies are probably being avoided by ordering the instructions, which is done by the compiler. However, the instructions in which most part of the execution time is spent are probably the operations within the nested loop in the *crcSlow()* function (loop2). This loop processes, a bit at a time, the eight bits of the corresponding byte and performs the modulo-2 division. Not only is this nested loop executed eight times in each execution of the outer loop, but it also executes conditional jumps, which usually lead to penalties in the execution time. Additionally, conditional jumps can cause control dependencies if not managed properly. The following image shows the C code section that is being discussed:



```
/*
 * Perform modulo-2 division, a bit at a time.
 */
for (bit = 8; bit > 0; --bit)
{
    /*
     * Try to divide the current data bit.
     */
    if (remainder & TOPBIT)
    {
        remainder = (remainder << 1) ^ POLYNOMIAL;
    }
    else
    {
        remainder = (remainder << 1);
    }
}
```

Figure 9: Most costly operations in the slow CRC-32 implementation.

Lastly, to calculate the number of instructions that is being executed in this implementation, the assembly language code must be carefully studied. The instructions outside the outer loop are not going to be taken into account. This will not seriously impact the calculation because in the main program (crc_main.c), we can see that the second parameter given to the function *crcSlow()* is 65535 (*BUFFER_SIZE*), which means that the instructions within the main loop will be executed much more times than the rest of them, making the instructions outside the loop almost insignificant. The result is 8388480 instructions. However, looking at the main program again, we can see that the whole process is executed as many times as *NUMBER_Of_BUFFERS* dictates. Knowing that *NUMBER_Of_BUFFERS* = 16, the final result is obtained by multiplying the previous result by 16. Therefore, the total amount is 134215680 instructions. The following image shows the part of the main program (crc_main.c) that is being addressed:

```
/* Slow software CRC based on a modulo 2 division implementation */
printf("Running the software CRC\n");
printf("------------------------\n");
sw_slow_timeA = alt_timestamp();
for(buffer_counter = 0; buffer_counter < NUMBER_OF_BUFFERS; buffer_counter++)
{
  sw_slow_results[buffer_counter] = crcSlow(data_buffer_region[buffer_counter], BUFFER_SIZE);
}
sw_slow_timeB = alt_timestamp();
printf("Completed\n\n\n");
```

Figure 10: Execution and processing time measurement of the slow implementation.

6.2 Fast CRC-32 software implementation, *crcFast()*

The source code corresponding to this version of the CRC-32 is also located in the file 'crc.c'.

This implementation of the algorithm starts with the function *crcInit()*, which fills a table called *crcTable*, an array of 256 elements. The function fills the table with 256 different remainders of the modulo-2 division for all the possible dividends. Since the data unit that is being used is the byte, there will be 256 dividends (from 0 to 255). To calculate the remainders, *crcInit()* uses a code very similar to that used in the slow version of the algorithm, *crcSlow()*.

Loop over each value from 0 to 255 (each possible dividend):

- The dividend is shifted to the upper 8 bits and the result is assigned to the remainder variable.
- Another loop is started, this time over the 8 bits of the dividend:
  - If the top (sign) bit of the remainder is set, then:
    - The remainder is shifted up one bit and XOR operation with the magic value (0x04C11DB7) is executed.
  - Otherwise, the remainder is shifted up one bit.
- Repeat the loop until the last possible dividend (255) is processed.

Once the loop finishes, the function ends and the table has been filled.

Once *crcInit()* has been executed, *crcFast()* can be called. Its code is now described:

The checksum is initialized with all bits set (0xFFFFFFFF).

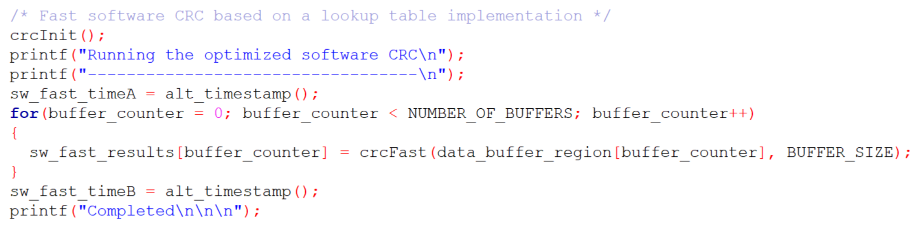
Loop over each byte of the message transmitted:

- The byte is taken and all its bits are bit-reflected.
- The current checksum value is shifted to the right so the eight most significant bits are know the least significant ones.
- XOR operation: data = bit-reflected byte XOR shifted remainder. *data* is the *crcTable* position that contains the corresponding modulo-2 division remainder.
- The position previously calculated is accessed and the corresponding remainder is obtained. With this value and the current value of the checksum shifted 8 bits to the left, a XOR operation is executed and the checksum is updated.

When the loop finishes, the entire checksum is bit-reflected. This is the final CRC-32 value.

The reason why this implementation of the CRC-32 algorithm is faster than the previous one is simple. Instead of calculating the modulo-2 division for each byte transmitted in the message, the fast version inserts all the possible remainders in a lookup table. Once this is done, the amount of code that will be executed each time a message needs to be processed is considerably lighter, since the only thing that will need to be calculated is the position to access in the table. Therefore, a lower number of instructions executed can be expected.

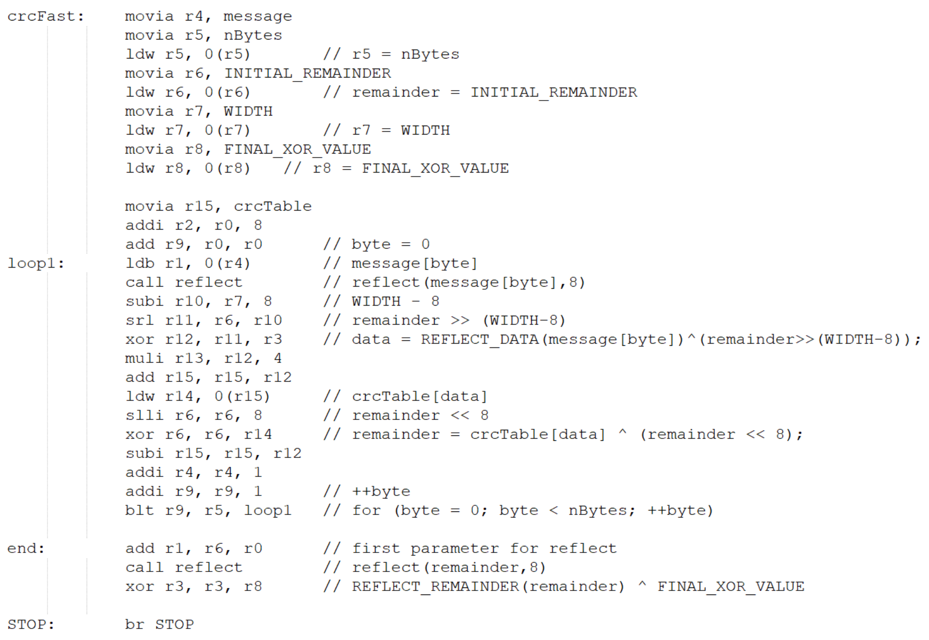
Now with a more thorough analysis, in the case of the lookup table implementation the *crcInit* function only needs to be called once and its execution time is not counted in this implementation processing time, as seen in the next image:



```
/* Fast software CRC based on a lookup table implementation */
crcInit();
printf("Running the optimized software CRC\n");
printf("----------------------------------\n");
sw_fast_timeA = alt_timestamp();
for(buffer_counter = 0; buffer_counter < NUMBER_OF_BUFFERS; buffer_counter++)
{
  sw_fast_results[buffer_counter] = crcFast(data_buffer_region[buffer_counter], BUFFER_SIZE);
}
sw_fast_timeB = alt_timestamp();
printf("Completed\n\n\n");
```

Figure 11: Execution and processing time measurement of the fast implementation.

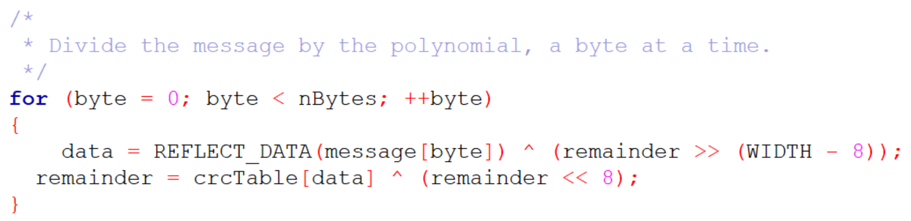
crcInit() is not taken into account for the calculation of the processing time, only the function *crcFast()* is measured. As in the previous implementation studied, I have wrote a simplified assembly code, equivalent to the C code of this function:



```
crcFast:    movia r4, message
            movia r5, nBytes
            ldw r5, 0(r5)       // r5 = nBytes
            movia r6, INITIAL_REMAINDER
            ldw r6, 0(r6)       // remainder = INITIAL_REMAINDER
            movia r7, WIDTH
            ldw r7, 0(r7)       // r7 = WIDTH
            movia r8, FINAL_XOR_VALUE
            ldw r8, 0(r8)   // r8 = FINAL_XOR_VALUE

            movia r15, crcTable
            addi r2, r0, 8
            add r9, r0, r0      // byte = 0
loop1:      ldb r1, 0(r4)       // message[byte]
            call reflect        // reflect(message[byte],8)
            subi r10, r7, 8     // WIDTH - 8
            srl r11, r6, r10    // remainder >> (WIDTH-8)
            xor r12, r11, r3    // data = REFLECT_DATA(message[byte])^(remainder>>(WIDTH-8));
            muli r13, r12, 4
            add r15, r15, r12
            ldw r14, 0(r15)     // crcTable[data]
            slli r6, r6, 8      // remainder << 8
            xor r6, r6, r14     // remainder = crcTable[data] ^ (remainder << 8);
            subi r15, r15, r12
            addi r4, r4, 1
            addi r9, r9, 1      // ++byte
            blt r9, r5, loop1   // for (byte = 0; byte < nBytes; ++byte)

end:        add r1, r6, r0      // first parameter for reflect
            call reflect        // reflect(remainder,8)
            xor r3, r3, r8      // REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE

STOP:       br STOP
```

Figure 12: *crcFast()* equivalent in assembly language.

Only *crcFast()* is shown because the code for the *reflect()* function is the same than in the slow implementation. Now analyzing the code, it is clear that the operations that have the greatest impact on the execution time are the ones in the main loop of the function. These operations, that are mostly arithmetic and logical, calculate the position in the table that needs to be accessed for each byte of the message. The main loop is executed as many times as bytes in the message and there are dependencies and a conditional jump. These two instructions are an example of a true dependency (RAW): 'slli r6, r6, 8' and 'xor r6, r6, r14'. The following image shows the code block section that is being discussed, in C language:



```
/*
 * Divide the message by the polynomial, a byte at a time.
 */
for (byte = 0; byte < nBytes; ++byte)
{
    data = REFLECT_DATA(message[byte]) ^ (remainder >> (WIDTH - 8));
  remainder = crcTable[data] ^ (remainder << 8);
}
```

Figure 13: Most costly operations in the fast CRC-32 implementation.

To conclude the fast version analysis, number of instructions executed must be calculated. As with the slow implementation, instructions outside the loop will not be taken into account. As seen in Figure 9, *crcFast()* is executed 16 times with nBytes being 65535, so the calculation that needs to be done is 16 * 65535 * number of instructions within the loop. 85 instructions are executed in the loop, including the instructions executed by the function *reflect()* when it is called. Therefore, the final result is 89127600.

6.3 Custom instruction CRC-32 implementation, *crcCI()*

The source code corresponding to this version of the CRC-32 is contained in the file 'ci_crc.c'.

Since almost all the operations necessary for calculating the CRC-32 value are done by the custom hardware module, the code of the function *crcCI()* is very simple.

The custom instruction is initialized calling it with the following parameters: CRC_CI_MACRO(0,0)

Loop over each word (4 bytes) of the input data:

- Custom instruction is called with its first parameter n = 3, which means that 32 bits (a word) will be sent to the custom instruction. The second parameter is a pointer to the byte of input data that is being processed.
- The pointer is moved by 4 to reach the next word.

Once the loop has finished, the function continues if the input data does not end on a word boundary. Otherwise, it returns the CRC-32 value.

If there are 3 bytes left to process:

- The custom instruction is called with n = 2 (write 16 bits). The second parameter is the pointer previously mentioned.
- The pointer is moved by 2 to reach the last byte.
- The custom instruction is called with n = 1 (write 8 bits).

Else if there are 2 bytes left:

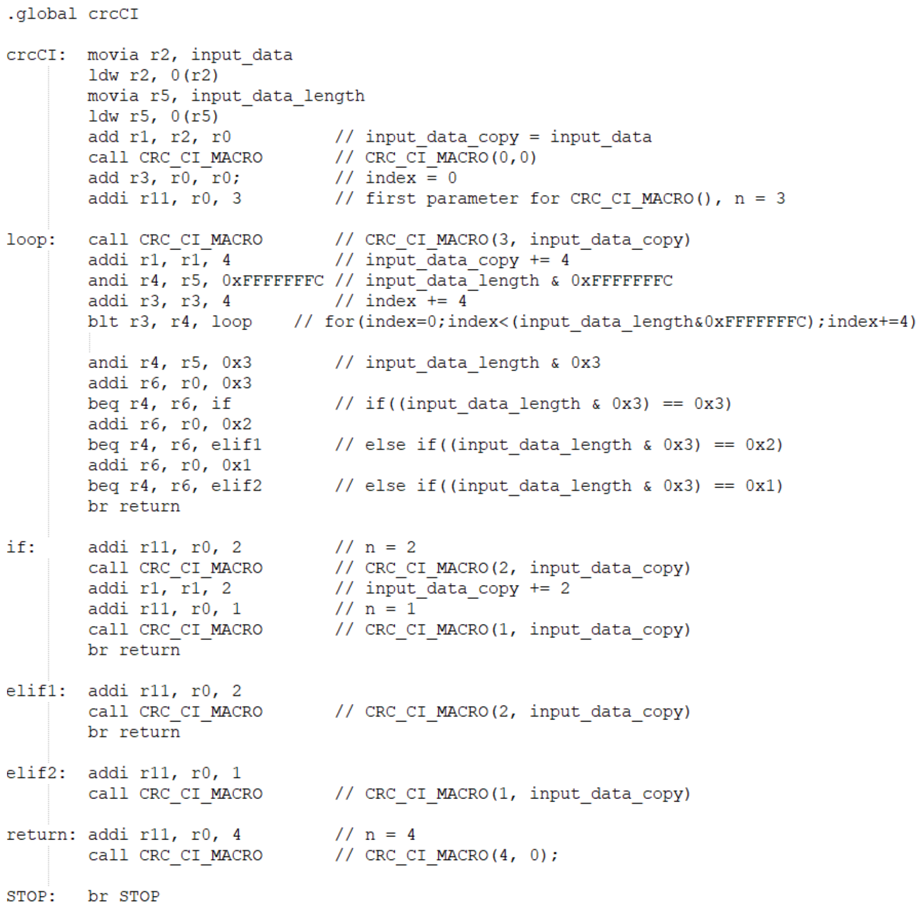
- The custom instruction is called with n = 2 (write 16 bits).

Else if there is only 1 byte left:

- The custom instruction is called with n = 1 (write 8 bits).

The function returns the CRC-32 value.

Seeing how simple the C code of this implementation is, it can be expected to have a much better performance than the previous two, but for a further analysis, the following image shows a simplified equivalent assembly language code.



```
.global crcCI

crcCI:  movia r2, input_data
        ldw r2, 0(r2)
        movia r5, input_data_length
        ldw r5, 0(r5)
        add r1, r2, r0          // input_data_copy = input_data
        call CRC_CI_MACRO       // CRC_CI_MACRO(0,0)
        add r3, r0, r0;         // index = 0
        addi r11, r0, 3         // first parameter for CRC_CI_MACRO(), n = 3

loop:   call CRC_CI_MACRO       // CRC_CI_MACRO(3, input_data_copy)
        addi r1, r1, 4          // input_data_copy += 4
        andi r4, r5, 0xFFFFFFFC // input_data_length & 0xFFFFFFFC
        addi r3, r3, 4          // index += 4
        blt r3, r4, loop    // for(index=0;index<(input_data_length&0xFFFFFFFC);index+=4)

        andi r4, r5, 0x3        // input_data_length & 0x3
        addi r6, r0, 0x3
        beq r4, r6, if          // if((input_data_length & 0x3) == 0x3)
        addi r6, r0, 0x2
        beq r4, r6, elif1       // else if((input_data_length & 0x3) == 0x2)
        addi r6, r0, 0x1
        beq r4, r6, elif2       // else if((input_data_length & 0x3) == 0x1)
        br return

if:     addi r11, r0, 2         // n = 2
        call CRC_CI_MACRO       // CRC_CI_MACRO(2, input_data_copy)
        addi r1, r1, 2          // input_data_copy += 2
        addi r11, r0, 1         // n = 1
        call CRC_CI_MACRO       // CRC_CI_MACRO(1, input_data_copy)
        br return

elif1:  addi r11, r0, 2
        call CRC_CI_MACRO       // CRC_CI_MACRO(2, input_data_copy)
        br return

elif2:  addi r11, r0, 1
        call CRC_CI_MACRO       // CRC_CI_MACRO(1, input_data_copy)

return: addi r11, r0, 4         // n = 4
        call CRC_CI_MACRO       // CRC_CI_MACRO(4, 0);

STOP:   br STOP
```

Figure 14: *crcCI()* equivalent in assembly language.

Studying the code above, it is clear that the section that has the biggest impact in the execution time of this implementation is the main loop. Not only the loop has to be executed once for every byte introduced in the message, but there is also a RAW (Read After Write) dependency between the instructions 'addi r3, r3, 4' and 'blt r3, r4, loop'. Additionally, the presence of a conditional jump, as mentioned in the first implementation analysis, can lead to a time penalty due to the condition evaluation. The part of the code that is being discussed is shown, in C language, in the next image:

```
/* Write 32 bit data to the custom instruction.  If the buffer does not end
 * on a 32 bit boundary then the remaining data will be sent to the custom
 * instruction in the 'if' statement below.
 */
for(index = 0; index < (input_data_length & 0xFFFFFFFC); index+=4)
{
  CRC_CI_MACRO(3, *(unsigned long *)input_data_copy);
  input_data_copy += 4;  /* void pointer, must move by 4 for each word */
}
```

Figure 15: Operations that take most of the execution time in the custom instruction implementation

Lastly, the number of instructions executed will be calculated. Again, the instructions outside the loop will be ignored. The next image shows how this implementation is executed and how its execution time is measured in the main program:

```
/* Custom instruction CRC */
printf("Running the custom instruction CRC\n");
printf("----------------------------------\n");
ci_timeA = alt_timestamp();
for(buffer_counter = 0; buffer_counter < NUMBER_OF_BUFFERS; buffer_counter++)
{
  ci_results[buffer_counter] = crcCI(data_buffer_region[buffer_counter], BUFFER_SIZE);
}
ci_timeB = alt_timestamp();
printf("Completed\n\n\n");
```

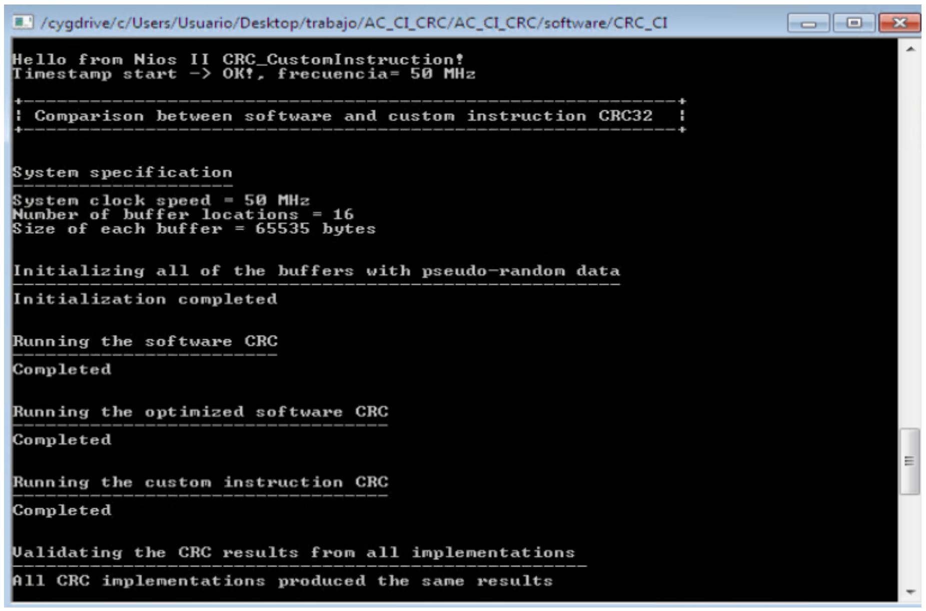
Figure 16: Execution and processing time measurement of the custom instruction implementation

As in the other two implementations, the message transmitted is 65535 bytes long and the whole process is executed 16 times. The calculation that returns the number of instructions executed is 65535 * 16 * 5(number of assembly language instructions within the loop). The final result is 5242800.

7 Performance Evaluation

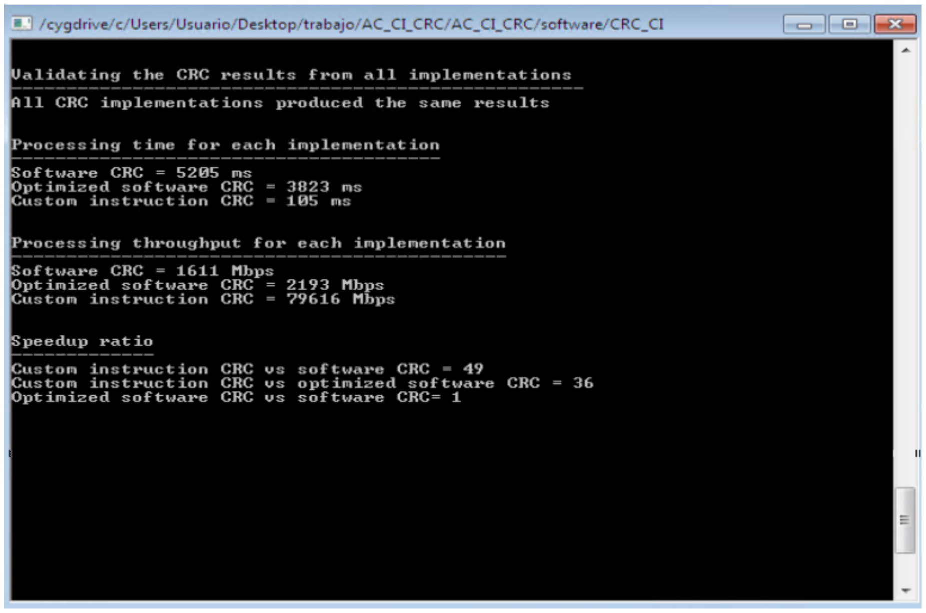
Performance evaluation is defined as the process by which a computer system's resources and outputs are assessed to determine whether the system is performing at an optimal level. This section describes the performance evaluation of a DE0-Nano board whose FPGA is configured to run the CRC-32 algorithm. Three types of software implementations are studied: slow, fast, custom instruction. These programs use two types of Nios II/f processors: standard for the first and second software implementations and customized Nios II processor for the third hardware implementation.

The following two images correspond to the execution of the three versions of the CRC-32 algorithm.



```
/cygdrive/c/Users/Usuario/Desktop/trabajo/AC_CI_CRC/AC_CI_CRC/software/CRC_CI

Hello from Nios II CRC_CustomInstruction!
Timestamp start -> OK!, frecuencia= 50 MHz

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


System specification
--------------------
System clock speed = 50 MHz
Number of buffer locations = 16
Size of each buffer = 65535 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
---------------------------------------------------
All CRC implementations produced the same results
```

Figure 17: Execution of the three versions of the CRC-32 algorithm.



/cygdrive/c/Users/Usuario/Desktop/trabajo/AC_CI_CRC/AC_CI_CRC/software/CRC_CI

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

Processing time for each implementation
---------------------------------------
Software CRC = 5205 ms
Optimized software CRC = 3823 ms
Custom instruction CRC = 105 ms

Processing throughput for each implementation
---------------------------------------------
Software CRC = 1611 Mbps
Optimized software CRC = 2193 Mbps
Custom instruction CRC = 79616 Mbps

Speedup ratio
-------------
Custom instruction CRC vs software CRC = 49
Custom instruction CRC vs optimized software CRC = 36
Optimized software CRC vs software CRC= 1
```

Figure 18: Execution of the three versions of the CRC-32 algorithm.

Table 1 shows the total execution times that were obtained for the three software implementation of the CRC-32 algorithm.

Table 1: Results of the performance evaluation.

Software version	Nios II mode	Processing time	Speed-up
Modulo 2 division implementation (slow)	standard	5205 ms	1,00
Lookup table implementation (fast)	standard	3823 ms	1,36
Use custom instruction	customized	105 ms	49,57

As seen in the table above, the custom instruction version of the CRC-32 algorithm clearly has the best performance, with a processing time of 105 ms and a speed-up of 49,57x in comparison to the modulo-2 division implementation (slow). Furthermore, the lookup table implementation (fast) is also evidently faster than the slow one, with a speed-up of 1,36x. However, the processing time reduction between the two software implementations (5205 to 3823 ms) is not nearly as considerable as the processing time reduction between the slow and the custom instruction implementations (5205 to 105 ms). This is why there is a speed-up of 36x between the custom and fast implementations, as Figure 6 shows. This was to be expected, considering that the logic behind the CRC-32 algorithm is not specially complex and that is has been specifically designed to execute it.

The cycles must be calculated too, which is easily done knowing the processing times and the system clock speed, 50 MHz.

$$cycles = t_{CPU} * f$$

Their values are:

- Slow: 260250000 cycles
- Fast: 191150000 cycles
- Custom instruction: 5250000 cycles

The same applies to the average cycles per instruction values, which can be calculated knowing the number of instructions executed and the amount of cycles that the program took to execute:

$$CPI = \frac{cycles}{instructions}$$

The number of instructions executed for each implementation previously calculated:

- Slow: 134215680 instructions
- Fast: 89127600 instructions
- Custom instruction: 5242800 instructions

Final CPI values:

- Slow: 260250000 / 134215680 = 1,9390 cycles per instruction
- Fast: 191150000 / 89127600 = 2,1447 cycles per instruction
- Custom instruction: 5250000 / 5242800 = 1,0014 cycles per instruction

The CPI result of the custom instruction implementation obtained is to be expected. Considering that the speed-up of the custom instruction implementation in comparison to the slow one is 49,57x, it is not surprising that the difference in the CPI is so drastic. Additionally, the customized hardware module used for the last implementation has been specifically designed to suit the program executed, which justifies the 1,0014 cycles per instruction result. This value is extremely close to the ideal value for the Nios II/f processor, 1 CPI.

On the other hand, the fast implementation CPI being slightly higher than the slower version could seem like a surprising result. However, this is due to the fact that in the lookup table implementation, the software optimizations that have been made had the purpose of decreasing the execution time only by reducing the amount of instructions executed by the algorithm, therefore lowering the denominator in the CPI calculation considerably. Unlike the fast version, for the custom instruction implementation the hardware was modified to improve its performance executing the algorithm, which has a direct and drastic impact on the CPI.

8 Conclusions

Analyzing the results of the processing time of each implementation of the CRC-32 algorithm (slow, fast and custom instruction), it is clear that using a customized processor and custom instructions created by the programmer has achieved an incredibly superior performance in comparison to only using software optimization. Such is this improvement, that in the tests made in this project the custom instruction implementation is 36 times faster than the optimized software implementation. Furthermore, studying the CPI has evidenced that the use of custom instructions can result in CPI values very close to the optimal. In conclusion, the CRC-32 algorithm is executed much faster in hardware than using software.

References

[1] Intel. Built-in functions and user-defined macros. `https://www.intel.com/content/www/us/en/docs/programmable/683242/current/built-in-functions-and-user-defined-macros.html`, 2022. Accessed: 2022-04-19.

[2] Intel. Nios II crc acceleration design example. `https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/exm-crc-acceleration.html`, 2022. Accessed: 2022-04-19.

[3] Intel. Nios II custom instruction overview. `https://www.intel.com/content/www/us/en/docs/programmable/683242/current/nios-ii-custom-instruction-overview.html`, 2022. Accessed: 2022-04-19.

[4] P. M. Nyasulu and J. Knight. *Introduction to Verilog.* Carleton University, `https://www.cs.upc.edu/~jordicf/Teaching/secretsofhardware/VerilogIntroduction_Nyasulu.pdf`, 2003. Accessed: 2022-04-19.

[5] OSDev. CRC32. `https://wiki.osdev.org/CRC32`, 2022. Accessed: 2022-04-19.

[6] Ross N. Williams. A painless guide to CRC error detections algorithms. `http://chrisballance.com/wp-content/uploads/2015/10/CRC-Primer.html`, 2022. Accessed: 2022-04-19.