Computer Architecture (40969)
Computer Science School (EII)
University of Las Palmas de Gran Canaria

Lab Assignment 3:

Nios V processor with customized architecture for a software application

Summary

- Objectives of the course project
- Introduction
- CRC-32 Algorithm
- Hardware Engineering
- Computer Architecture
- Software Engineering

Processors can be optimized for any number of reasons, including combinations of throughput, latency, and power.

The goal of **processor customization** is to change an architecture to benefit a small set of applications, while maintaining the flexibility to run many other applications.

Objectives of the course project

- Discover and explain the key mechanisms by which processors with a specialized instruction set reduce the execution time of determined programs.

- Hardware Engineering
  - Student should understand the Verilog hardware description language contained in the *.v files: CRC_Custom_Instruction.v, CRC_Component.v.
  - The circuit schematics that are equivalent to these Verilog files should be generated and explained.

Objectives of the course project

- Computer Architecture
  - How the custom function unit is integrated into the data path of the Nios V/g processor.
  - Explain the performance improvement that is achieved when the custom instructions are used. Hint: $t_{CPU} = N \times CPI / f$ (N: number of executed instructions, CPI: cycles per instruction, f: clock speed).

Objectives of the course project

- Software Engineering
  - CRC-32 algorithm should be analyzed: operations, data structures, data hazards.
  - Software profiling is used to discover the most costly operations (see crc_main.c y crc.c files)
  - The source codes must be compiled and linked. Then, the executable file must be run using the DE0-Nano board. The execution times of three types of programs are measured: slow, fast and customized software versions of the CRC-32 algorithm.
  - The performance of the Nios V/g software processor is evaluated using the measurements of execution times. Performance must be evaluated using CPI, t_{CPU}, and Speed-Up.

Introduction [Williams1993]

- 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. 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

- In the above, the second byte of the message was corrupted from 23 to 27 by the communications channel. However, the receiver can detect this by comparing the transmitted checksum (33) with the computer checksum of 37 (6 + 27 + 4).

- This document addresses only the CRC-32 algorithm, which fall into the class of error detection algorithms that leave the data intact and append a checksum on the end. i.e.:

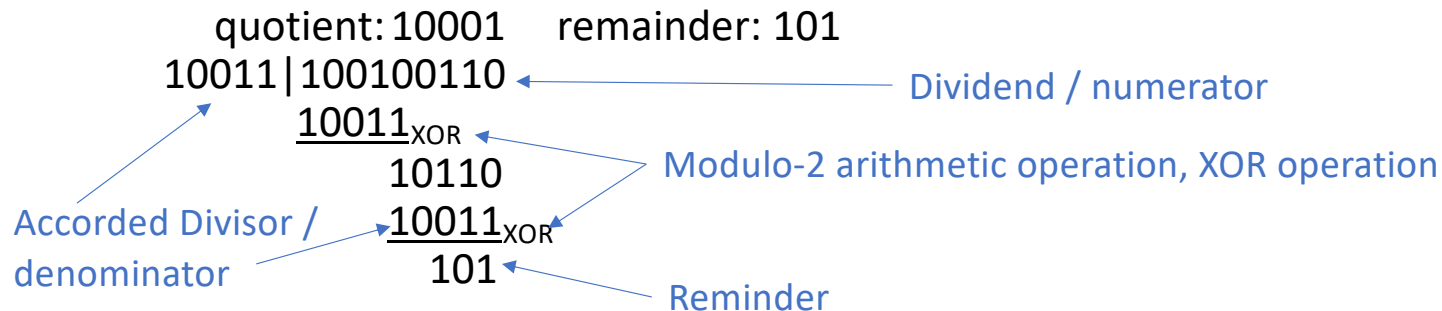 <original intact message> <checksum>

Theoretical CRC algorithm

- Data Message (M, 5 x 2 bits): 0x3, 0x1, 0x1, 0x2, 0x3

- Data Message (M, 10 bits): 11-01-01-10-11

- Accorded divisor (poly of width W=4: G= (x^5)+x^4+x+1, k=5 bits): 10011

- Augmented data before CRC división (M' = M | 0000): 11-01-01-10-11-0000

- A mod$_{CRC}$ B = remainder (A /$_{XOR}$ B); /$_{XOR}$: modulo-2 division (ver sigui. ppt)

- Checksum Message (CRC remainder, W=4 bits):
    - CRC = M' mod$_{XOR}$ G = 11-01-01-10-11-0000 mod$_{XOR}$ 10011 = 1110

- Sending (augmented) message (M'' = M | CRC, 14 bits): data (10 bits) | CRC (4 bits)
    - 11-01-01-10-11-1110

- Test: M'' mod$_{XOR}$ G = 0000 (11-01-01-10-11-1110 mod$_{CRC}$ 10011)

Modulo-2 division (examples)

Modulo 2 division can be performed in a manner similar to arithmetic long division. Subtract the denominator (the bottom number) from the leading parts of the enumerator (the top number). Proceed along the enumerator until its end is reached. Remember that we are using modulo 2 subtraction. For example, we can divide 100100110 by 10011 as follows:

```
    quotient: 10001     remainder: 101
10011|100100110                              ← Dividend / numerator
      10011 XOR                              ← Modulo-2 arithmetic operation, XOR operation
      10110
      10011 XOR
        101                                  ← Reminder
```

Accorded Divisor / denominator

This has the effect that X/Y = Y/X. For example:

```
          1    remainder: 1010                        1    remainder: 1010
11001|10011                                10011|11001
      11001 XOR                                  10011 XOR
       1010                                       1010
```

CRC checksum (example)

- Original message : 11 0101 1011

- Poly : 10011 (G= (x^5)+x^4+x+1, k=5 bits):

- Message after appending W=4 zeros : 11 0101 1011 0000

- Now we simply divide the augmented message by the poly using CRC arithmetic. This is the same division as before:

<div align="center">

1100001010 = Quotient (nobody cares about the quotient)

↑↑↑↑↑↑↑↑↑↑

_____

10011)   11010110110000 = Augmented message (1101011011 + 0000)

=Poly

1110 = Remainder = THE CHECKSUM!!!!

</div>

CRC example

Reference: [Stackoverflow2022]

```
                              1100001010 = Quotient (nobody cares about the quotient)
                          _____
        10011 )           11010110110000 = Augmented message (1101011011 + 0000)
        =Poly             10011
                          --------
              R = 10011
              Poly = 10011
                          --------
              R = 00001
                  00000
                          --------
              R = 00010
                  00000
                          --------
              R = 00101
                  00000
                          --------
              R = 01011
                  00000
                          --------
                  10110
              Poly = 10011
                          --------
                  01010
                  00000
                          ---------
                  10100
              Poly = 10011
                          --------
                  01110
                  00000
                          --------
                  1110 = Remainder = THE CHECKSUM!!!!
```
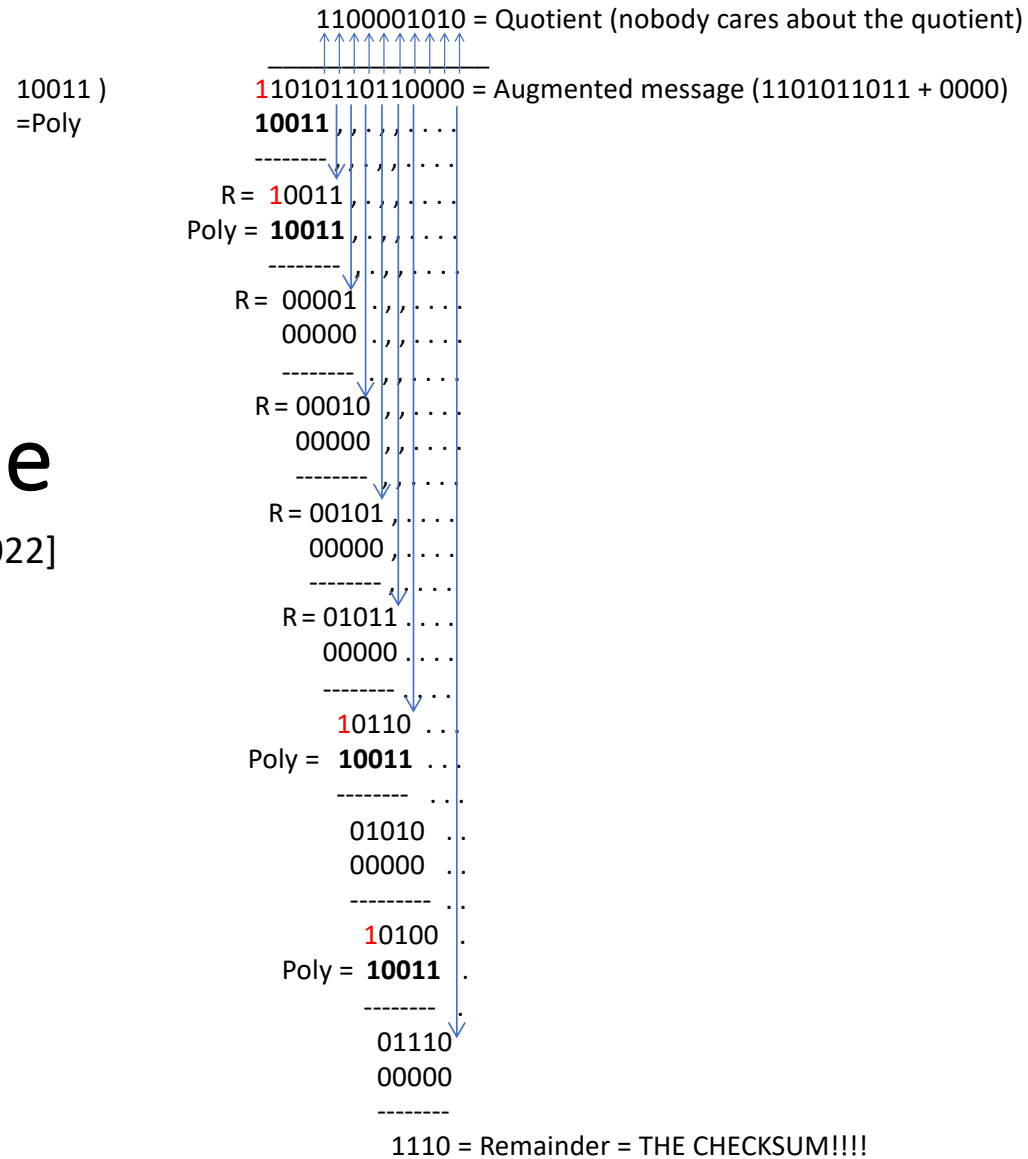
CRC-32 Algorithm [OSDev]

- CRC-32 is a checksum/hashing algorithm that is very commonly used in kernels and for Internet checksums.

- A message composed of multiple bytes is passed to CRC-32 algorithm.

- In the two following slides, the slow version of the software implementation is described.

Slow CRC-32 software implementation, `crcSlow()`

- Start with a 32-bit checksum with all bits set (0xFFFFFFFF). This helps to give an output value other than 0 for an input string of "0" bytes.

- Loop over each byte of message.
  - Take an 8-bit message and bit-reflect all the bits in that byte.
  - Shift it to the upper 8 bits of the current 32-bit checksum.
  - Exclusive-OR: checksum ← checksum ^ shifted byte; ^: XOR operation
  - Loop over those 8 bits.
    - If the top (sign) bit of checksum is set, then:
      - shift the checksum up one bit and
      - exclusive-OR it with the magic value 0x04C11DB7 ((x^{32}) + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1).
    - Otherwise, just shift the checksum up one bit.
  - Then repeat.

- Repeat over each byte of message.

- Bit-reflect the entire checksum. This is the CRC-32 value.

Slow CRC-32 software implementation based on modulo-2 division, `crcSlow(),crc.c`

```
crc crcSlow(unsigned char const message[], int nBytes) {
  crc              remainder = INITIAL_REMAINDER;        0xFFFFFFFF
  int              byte;
  unsigned char    bit;
                                        Shift register

  for (byte = 0; byte < nBytes; ++byte)  {
    remainder ^= (REFLECT_DATA(message[byte]) << (WIDTH - 8));

    for (bit = 8; bit > 0; --bit)  {
        if (remainder & TOPBIT) {
            remainder = (remainder << 1) ^ POLYNOMIAL;
        }                              0x04C11DB7
        else  {
            remainder = (remainder << 1);
        }
      }
    }
  }
  return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
}                                                         0xFFFFFFFF
```

Pre-loop
Loop
Post-loop

- Given a message composed of multiple bytes.
- Start with a 32bit checksum with all bits set (0xffffffff). This helps to give an output value other than 0 for an input string of "0" bytes.
- Loop over each byte of message.
    - Take a 8-bit message and bit-reflect all the bits in that byte.
    - Shift it to the upper 8 bits of the current 32-bit checksum.
    - Loop over those 8 bits.
        - If the top (sign) bit of checksum is set,
            - Then:
            - shift the checksum up one bit and
            - exclusive-OR it with the magic value 0x04C11DB7.
        - Otherwise just shift the checksum up one bit.
    - Then repeat.
- Repeat over each byte of message
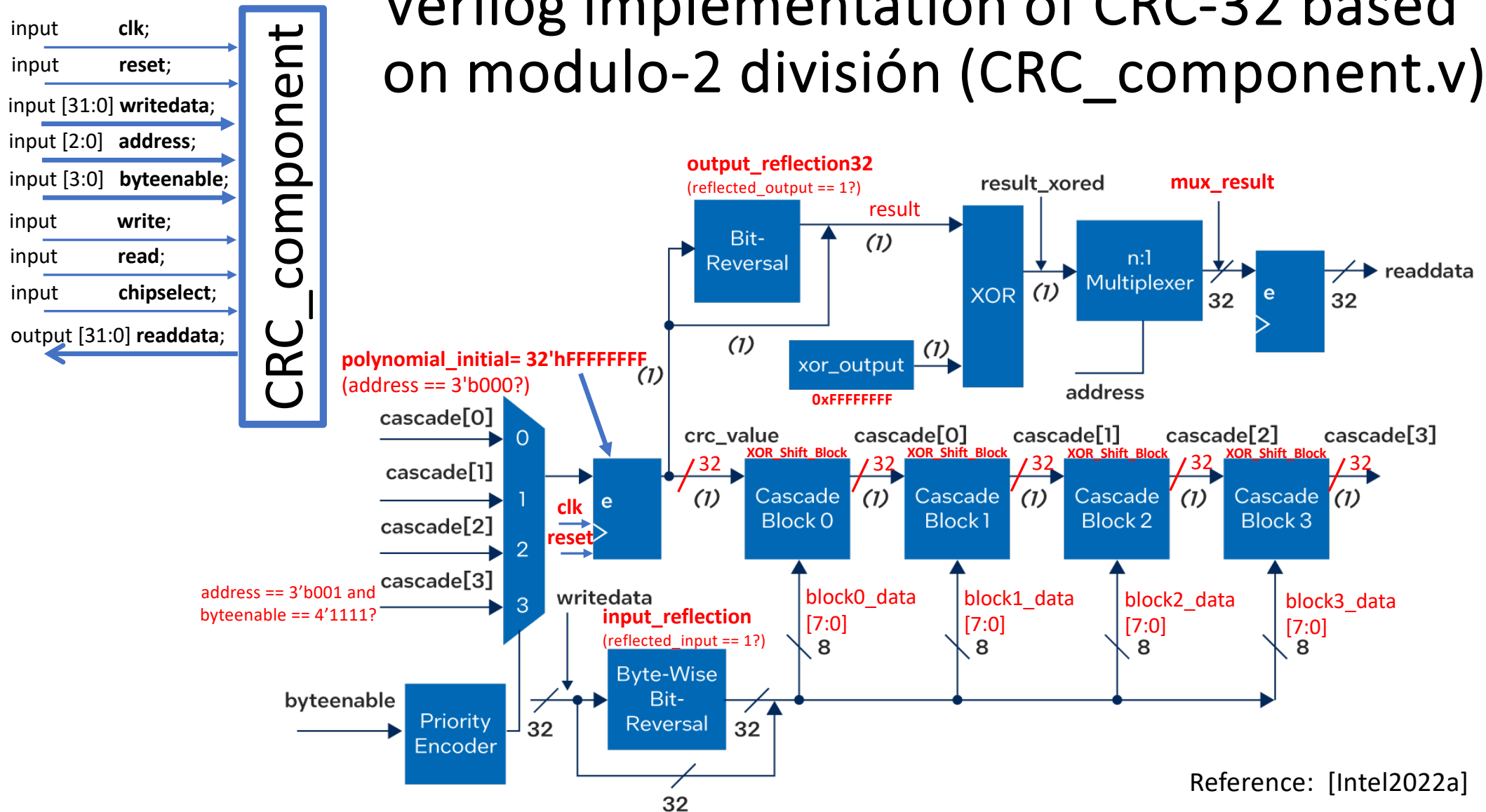- Bit-reflect the entire checksum. This is the CRC-32 value.

Hardware Engineering of Nios V custom instructions for the CRC-32 algorithm

- Scope of hardware engineering in this project: hardware design of computers and their components using a high-level language.

- In this course project, Verilog language is used.

- This section of project describes the internal organization of the CRC-32 hardware module.
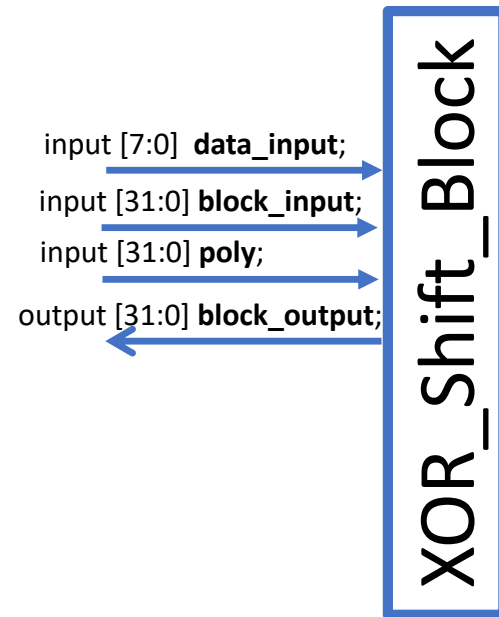
Verilog [Nyasulu1993]

- Verilog is one of several languages used to design hardware. It uses a C-like syntax to define wires, registers, clocks, input-output devices and all of the connections between them. Every useful Verilog design will include some sort of state machine(s) to control sequential behavior.

- Some keywords:
  - wire
    - Wires are used for connecting different elements. They can be treated as physical wires. They can be read or assigned. No values get stored in them. They need to be driven by either continuous assign statement or from a port of a module.
  - reg
    - They represent data storage elements. They retain their value till next value is assigned to them (not through assign statement). They can be synthesized to Flip-Flops.
  - genvar
    - A *genvar* is a variable used in generate-for loop. It stores positive integer values.
  - generate
    - A *generate* loop permits generating multiple instances of modules and primitives, as well as generating multiple occurences of variables, nets, tasks, functions, continuous assignments, initial and always procedural blocks*.
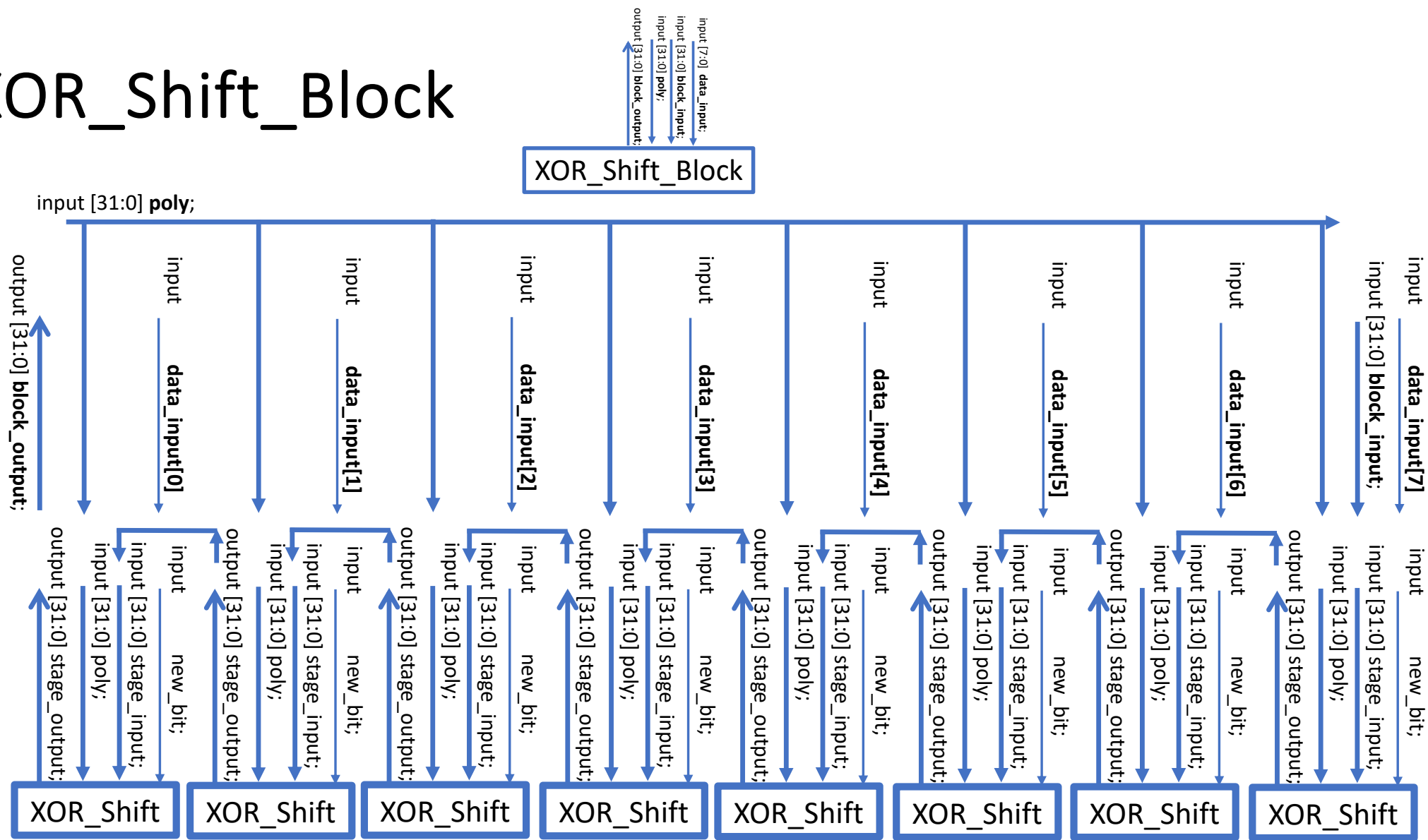
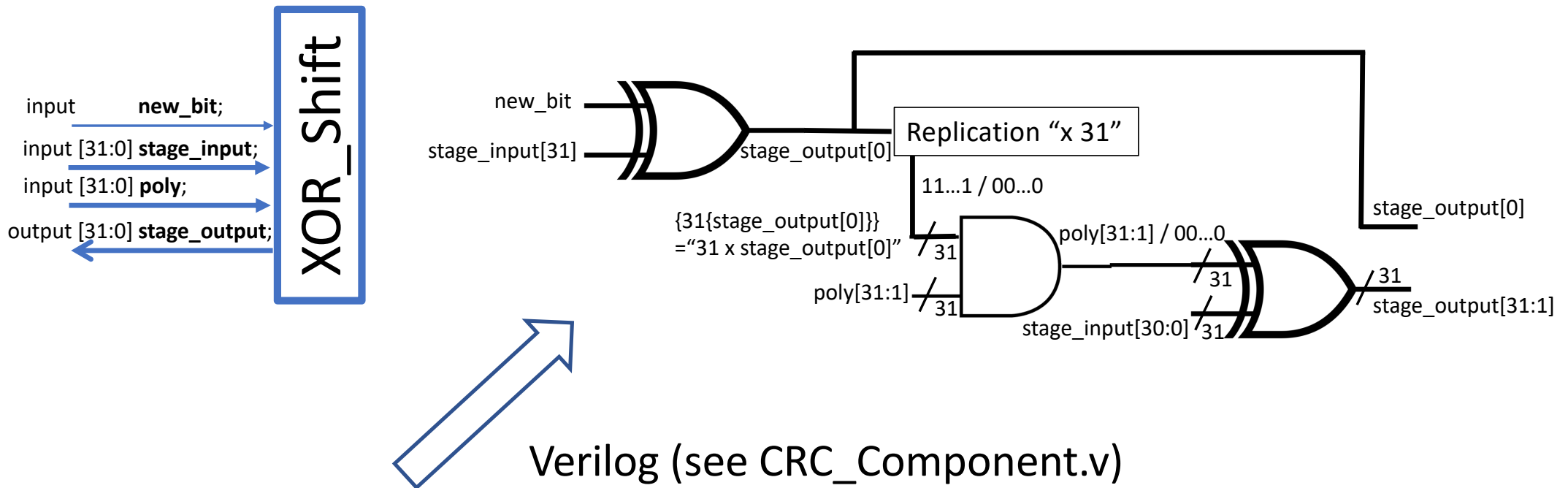# Verilog implementation of CRC-32 based on modulo-2 división (CRC_component.v)

```
input        clk;
input        reset;
input [31:0] writedata;
input [2:0]  address;
input [3:0]  byteenable;
input        write;
input        read;
input        chipselect;
output [31:0] readdata;
```

CRC_component

output_reflection32
(reflected_output == 1?)

result_xored

mux_result

Bit-Reversal → result *(1)*

XOR *(1)* → n:1 Multiplexer → 32 → e > → readdata 32

(1)

(1)

xor_output *(1)*

0xFFFFFFFF

address

polynomial_initial= 32'hFFFFFFFF
(address == 3'b000?)

(1)

cascade[0] 0

cascade[1] 1

cascade[2] 2

cascade[3] 3

clk
reset

e >

crc_value
32
(1)

cascade[0]
32
(1)

XOR_Shift_Block

cascade[1]
32
(1)

XOR_Shift_Block

cascade[2]
32
(1)

XOR_Shift_Block

cascade[3]
32
(1)

Cascade Block 0 → Cascade Block 1 → Cascade Block 2 → Cascade Block 3

XOR_Shift_Block

address == 3'b001 and byteenable == 4'1111?

writedata
input_reflection
(reflected_input == 1?)

block0_data [7:0] 8

block1_data [7:0] 8

block2_data [7:0] 8

block3_data [7:0] 8

byteenable → Priority Encoder → 32 → Byte-Wise Bit-Reversal → 32

32

Reference: [Intel2022a]

Verilog module: XOR_Shift_Block

input [7:0] **data_input**; →

input [31:0] **block_input**; →

input [31:0] **poly**; →

output [31:0] **block_output**; ←

XOR_Shift_Block

XOR_Shift_Block

Verilog module: XOR_Shift, modulo-2 division

input **new_bit**;
input [31:0] **stage_input**;
input [31:0] **poly**;
output [31:0] **stage_output**;

XOR_Shift

new_bit
stage_input[31]
stage_output[0]

Replication "x 31"

11...1 / 00...0

{31{stage_output[0]}}
="31 x stage_output[0]"

poly[31:1] / 00...0

poly[31:1]

stage_input[30:0]

stage_output[31:1]

stage_output[0]

Verilog (see CRC_Component.v)

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]);

C simulation of Verilog implementation: `crcSimulated()`

```
crc crcSimulado(unsigned char const message[], int nBytes) {
 crc                    remainder = INITIAL_REMAINDER;        ──►  0xFF FF FF FF
 crc                    dumy2, shifted_data;                       (32 bits)
 int                    byte;
 unsigned char          bit, dumy;
 for (byte = 0; byte < nBytes; ++byte)  {
        shifted_data = (REFLECT_DATA(message[byte]) << (WIDTH - 8));
```

Input message is aligned at the most significant byte of the 32-bit remainder

32

```
    for (bit = 8; bit > 0; --bit)  {
        dumy2 = remainder ^ shifted_data;
        dumy  = (unsigned char) ((dumy2 & TOPBIT)>>31);
        if (dumy) {                    // stage_output[0] ==? 1
            remainder = (remainder & DOWNBIT) ^ (POLYNOMIAL >> 1);
        }
        remainder = (remainder << 1) | dumy;
        shifted_data = (shifted_data << 1);
    }
 }
 return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
}
```
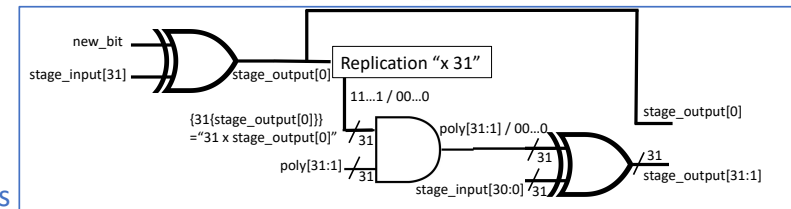
// new_bit= shifted_data[31]
// stage_input[31]= remainder[31]
// dumy2 = new_bit ^ stage_input[31]
// dumy = stage_output[0]= (new_bit ^ stage_input[31]) >> 31

0x7F FF FF FF   0x04 C1 1D B7

// stage_output[0:31] :: remainder

// input message is left shifted in all cases

0xFF FF FF FF

Loop

Post-loop

Example of the CRC-32 implementation: pre-loop

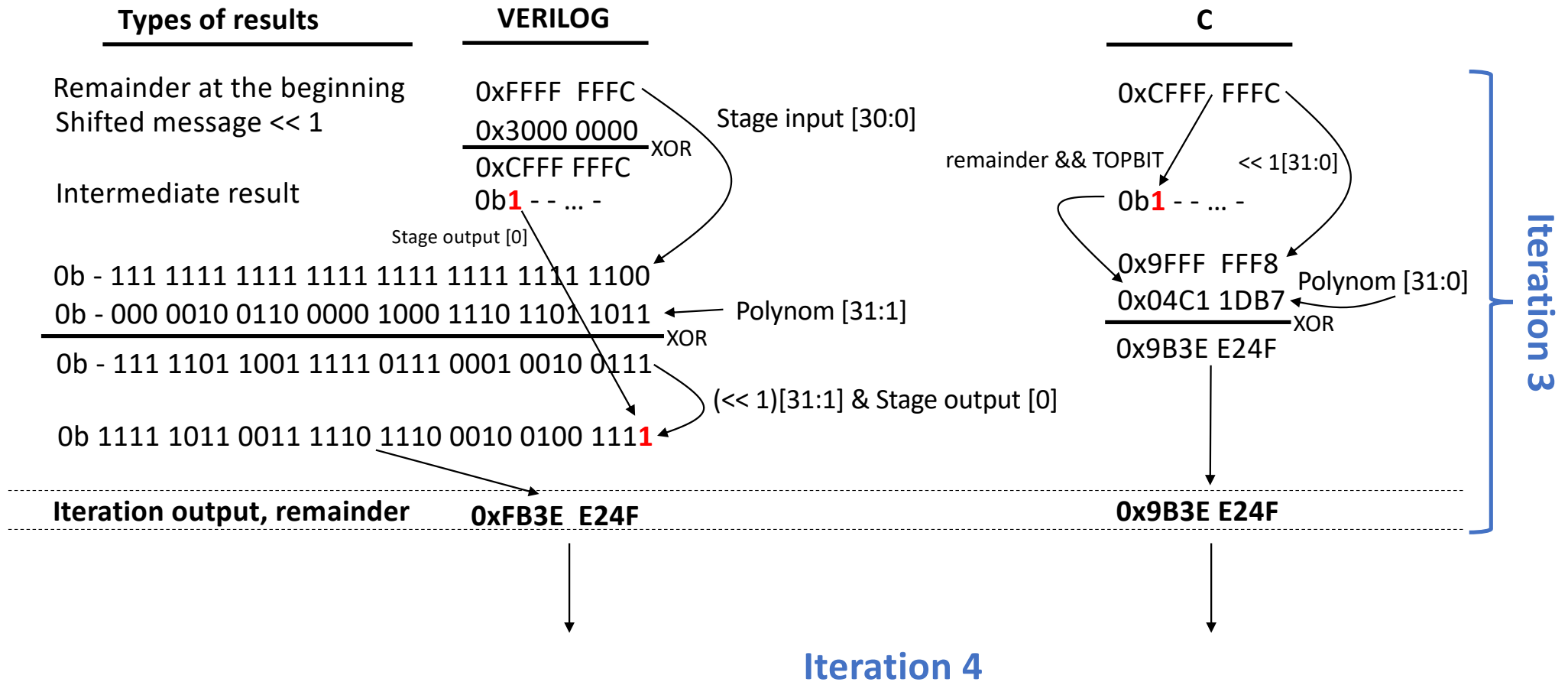|		
Input message 8-bit	0011 0011	0x33
Reflected input message 8-bit	1100 1100	0xCC
Shifted and aligned message 32-bit	1100 1100 0000 0000 0000 0000 0000 0000	0xCC00 0000
Polynom 32-bit	0000 0100 1100 0001 0001 1101 1011 0111	0x04C1 1DB7
Initial remainder 32-bit	1111 1111 1111 1111 1111 1111 1111 1111	0xFFFF FFFF

VERILOG	C (crc_slow())

Pre-loop　　No operation

Initial remainder　　0xFFFF FFFF

Shifted input message and aligned at the most significat byte → 0xCC00 0000

Initial remainder → 0xFFFF FFFF

XOR

0x33FF FFFF

remainder ^= (REFLECT_DATA(message[byte]) << (WIDTH - 8));

byte=0　　WIDTH=32

CRC-32 implementation: loop, iteration 1...8
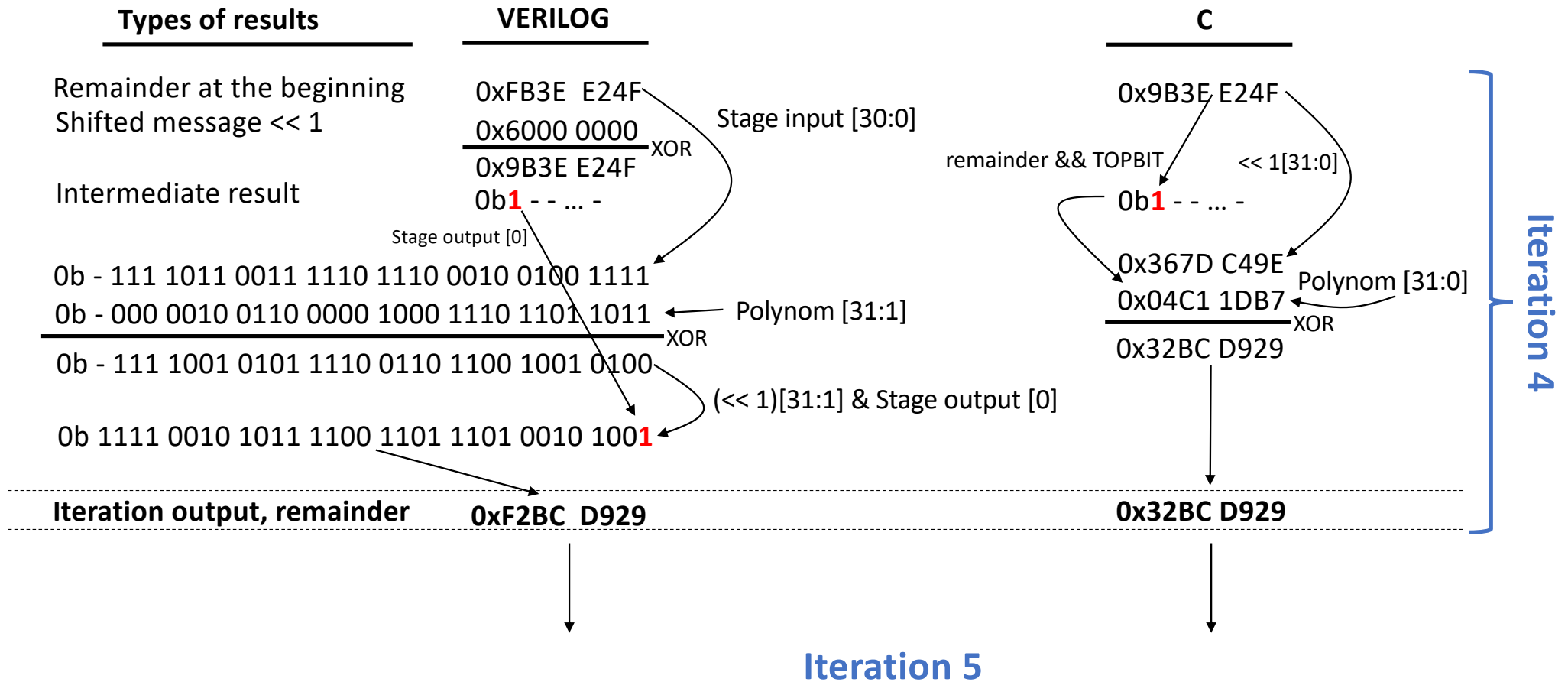
Types of results   **VERILOG**   **C**

data_input[7]: most
significant bit of message

&: bit concatenation

	VERILOG		C
Remainder at the beginning	0xFFFF FFFF	(<< 1)[31:1] & Stage output [0]	0x33FF FFFF
Shifted message << 1	0xCC00 0000		
	0x33FF FFFF	XOR	resto && TOPBIT
Intermediate result	0b**0** - - ... -	&&: 32-bit AND operation	0b**0** - - ... -
	Stage output [0]		<< 1[31:0]
Iteration output, remainder	**0xFFFF FFFE**		**0x67FF FFFE**

Iteration 1

	VERILOG		C
Remainder at the beginning	0xFFFF FFFE	(<< 1)[31:1] & Stage output [0]	0x67FF FFFE
Shifted message << 1	0x9800 0000		
	0x67FF FFFE	XOR	remainder && TOPBIT
Intermediate result	0b**0** - - ... -		0b**0** - - ... -
	Stage output [0]		<< 1[31:0]
Iteration output, remainder	**0xFFFF FFFC**		**0xCFFF FFFC**

Iteration 2

Iteration 3

CRC-32 implementation: loop, iteration 1...8

Types of results

VERILOG

C

Remainder at the beginning
Shifted message << 1

0xFFFF FFFC
0x3000 0000
0xCFFF FFFC   XOR

Stage input [30:0]

0xCFFF FFFC

remainder && TOPBIT   << 1[31:0]

Intermediate result

0b**1** - - ... -

0b**1** - - ... -

Stage output [0]

0x9FFF FFF8

0b - 111 1111 1111 1111 1111 1111 1111 1100
0b - 000 0010 0110 0000 1000 1110 1101 1011   ← Polynom [31:1]
                                              XOR

0x04C1 1DB7   Polynom [31:0]
                        XOR

0b - 111 1101 1001 1111 0111 0001 0010 0111

0x9B3E E24F

(<< 1)[31:1] & Stage output [0]

0b 1111 1011 0011 1110 1110 0010 0100 111**1**

Iteration output, remainder   **0xFB3E E24F**

0x9B3E E24F

Iteration 4

Iteration 3

CRC-32 implementation: loop, iteration 1...8

Types of results   **VERILOG**   **C**

Remainder at the beginning   0xFB3E E24F   0x9B3E E24F
Shifted message << 1   0x6000 0000   remainder && TOPBIT   << 1[31:0]
                                   0x9B3E E24F XOR
                                                        Stage input [30:0]
Intermediate result   0b**1** - - ... -   0b**1** - - ... -

                  Stage output [0]   0x367D C49E
                                                                             0x04C1 1DB7   Polynom [31:0]
0b - 111 1011 0011 1110 1110 0010 0100 1111   XOR
0b - 000 0010 0110 0000 1000 1110 1101 1011 ← Polynom [31:1]   0x32BC D929
                                              XOR
0b - 111 1001 0101 1110 0110 1100 1001 0100
                                                   (<< 1)[31:1] & Stage output [0]

0b 1111 0010 1011 1100 1101 1101 0010 100**1**

Iteration output, remainder   **0xF2BC D929**   **0x32BC D929**

                                        ↓   ↓

                                Iteration 5

CRC-32 implementation: loop, iteration 1…8

Types of results	VERILOG		C

&: bit concatenation

data_input[7]: bit más significativo de mensaje

Iteration 5

Remainder at the beginning	0xF2BC D929	(<< 1)[31:1] & Stage output [0]	0x32BC D929
Shifted message << 1	0xC000 0000		remainder && TOPBIT
	0x32BC D929 XOR		<< 1[31:0]
Intermediate result	0b**0** - - … -		0b**0** - - … -
Iteration output, remainder	**0xE579 B252**	Stage output [0]	**0x6579 B252**

Iteration 6

Remainder at the beginning	0xE579 B252	(<< 1)[31:1] & Stage output [0]	0x6579 B252
Shifted message << 1	0x8000 0000		resto && TOPBIT
	0x6579 B252 XOR		<< 1[31:0]
Intermediate result	0b**0** - - … -		0b**0** - - … -
Iteration output, remainder	**0xCAF3 64A4**	Stage output [0]	**0xCAF3 64A4**

Iteration 7

CRC-32 implementation: loop, iteration 1…8

Types of results	VERILOG	C

Remainder at the beginning
Shifted message << 1

0xCAF3 64A4
0x0000 0000
—————— XOR
0xCAF3 64A4

Stage input [30:0]

0xCAF3 64A4

remainder && TOPBIT   << 1[31:0]

Intermediate result

0b**1** - - … -

0b**1** - - … -

Stage output [0]

0b - 100 1010 1111 0011 0110 0100 1010 0100
0b - 000 0010 0110 0000 1000 1110 1101 1011   Polynom [31:1]
————————————————————————— XOR
0b - 100 1000 1001 0011 1110 1010 0111 1111

0x95E6 C948
0x04C1 1DB7   Polynom [31:0]
—————— XOR
0x9127 D4FF

(<< 1)[31:1] & Stage output [0]

0b 1001 0001 0010 0111 1101 0100 1111 111**1**

Iteration output, remainder   **0x9127 D4FF**   **0x9127 D4FF**

Iteration 8

Iteration 7

CRC-32 implementation: loop, iteration 1…8

Types of results

VERILOG

C

Remainder at the beginning
Shifted message << 1

0x9127 D4FF
0x0000 0000
0x9127 D4FF XOR

Stage input [30:0]

0x9127 D4FF

remainder && TOPBIT   << 1[31:0]

Intermediate result

0b**1** - - … -

Stage output [0]

0b**1** - - … -

0x224F A9FE

0b - 001 0001 0010 0111 1101 0100 1111 1111
0b - 000 0010 0110 0000 1000 1110 1101 1011 ← Polynom [31:1]

Polynom [31:0]

0x04C1 1DB7 XOR

0b - 001 0011 0100 0111 0101 1010 0010 0100 XOR

0x268E B449

(<< 1)[31:1] & Stage output [0]

0b 0010 0110 1000 1110 1011 0100 0100 100**1**

Iteration output, remainder   **0x268E B449**   **End of loop**   **0x268E B449**

Iteration 8

End of CRC-32 procedure

CRC-32 implementation: final output

	VERILOG			**C**
Remainder, loop output	**0x268E B449**			**0x268E B449**
Bit-reversal	**0x922D 7164**	REFLECT_REMAINDER(remainder)		**0x922D 7164**
XOR_output	**0xFFFF FFFF**	FINAL_XOR_VALUE		**0xFFFF FFFF**
readdata	**0x6DD2 8E9B** XOR			

return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);

0x6DD2 8E9B XOR

C subroutine using custom CRC-32 custom instruction (ci_crc.c)

```c
unsigned long crcCI(unsigned char * input_data, unsigned long input_data_length) {

  unsigned long index;

  /* copy of the data buffer pointer so that it can advance by different widths */

  void * input_data_copy = (void *)input_data;

  /* The custom instruction CRC will initialize to the inital remainder value */

  CRC_CI_MACRO(0,0);

  /* 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 */

  }

  /* Write the remainder of the buffer if it does not end on a word boundary */

  if((input_data_length & 0x3) == 0x3)  /* 3 bytes left */  {

    CRC_CI_MACRO(2, *(unsigned short *)input_data_copy);

    input_data_copy += 2;

    CRC_CI_MACRO(1, *(unsigned char *)input_data_copy);

  }

  else if((input_data_length & 0x3) == 0x2) /* 2 bytes left */  {

    CRC_CI_MACRO(2, *(unsigned short *)input_data_copy);

  }

  else if((input_data_length & 0x3) == 0x1) /* 1 byte left */  {

    CRC_CI_MACRO(1, *(unsigned char *)input_data_copy);

  }

  /* There are 4 registers in the CRC custom instruction.  Since this example uses CRC-32 only the first register must be read in order to receive the full result. */

  return CRC_CI_MACRO(4, 0);

}
```

Example of CRC-32 procedure using 1-byte message


```
/cygdrive/c/altera/12.1sp1
------------------------------------------------------------
Altera Nios2 Command Shell [GCC 4]

Version 12.1sp1, Build 243
------------------------------------------------------------

dbenitez@portatilAcer10p /cygdrive/c/altera/12.1sp1
$ nios2-terminal
nios2-terminal: connected to hardware target using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1, instance 0
nios2-terminal: (Use the IDE stop button or Ctrl-C to terminate)

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 = 1
Size of each buffer = 1 bytes


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
```

1-byte input message

```
/cygdrive/c/altera/12.1sp1
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

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


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


Simulacion en C del codigo Verilog de CRC
------------------------------------------------------------
crcSimulado - byte= 0, input data= 0x33, inicio= 0xffffffff, pol= 0x4c11db7
        Pre-bucle - remainder= 0xffffffff, reflected data= 0xcc
        EN-bucle - dato_despla= 0xcc000000, topbit= 0x0, remaind= 0xfffffffe
        EN-bucle - dato_despla= 0x98000000, topbit= 0x0, remaind= 0xfffffffc
        EN-bucle - dato_despla= 0x30000000, topbit= 0x1, remaind= 0xfb3ee24f
        EN-bucle - dato_despla= 0x60000000, topbit= 0x1, remaind= 0xf2bcd929
        EN-bucle - dato_despla= 0xc0000000, topbit= 0x0, remaind= 0xe579b252
        EN-bucle - dato_despla= 0x80000000, topbit= 0x0, remaind= 0xcaf364a4
        EN-bucle - dato_despla= 0x0, topbit= 0x1, remaind= 0x9127d4ff
        EN-bucle - dato_despla= 0x0, topbit= 0x1, remaind= 0x268eb449
        FIN - reflect_remainder= 0x922d7164, output= 0x6dd28e9b
Completed


Validating the CRC results from all implementations
------------------------------------------------------------
RESULTADOS - buf_coun= 0, sw_slow_results= 0x6dd28e9b, ci_results= 0x6dd28e9b
```

Subroutine not using custom instruction

Subroutine using custom instruction

Computer Architecture

- This section of course project describes the hardware-software interface of Nios V/g custom instructions.
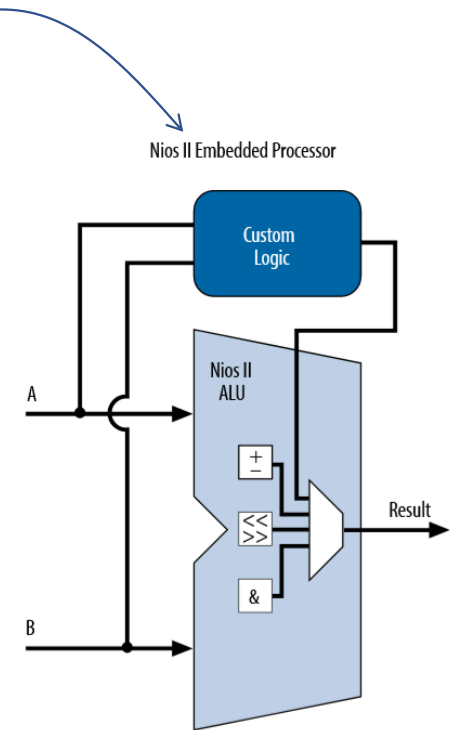
Custom instructions (customized to software)

- Custom instructions give you the ability to tailor the Nios V processor to meet the needs of a particular application.
- You can accelerate time critical software algorithms by converting them to custom hardware logic blocks.
- Reference: [Intel2022b]

Nios II Embedded Processor
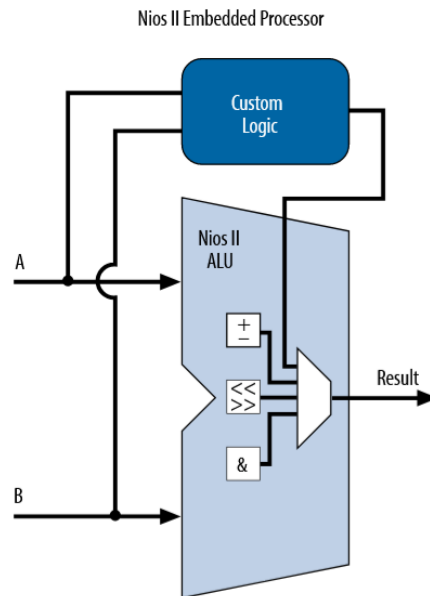
The custom instruction logic connects directly to the Nios V arithmetic logic unit (ALU)

Custom Instruction Implementation

Nios II Embedded Processor

- Nios V custom instructions are custom logic blocks adjacent to the arithmetic logic unit (ALU) in the processor's datapath.

- 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 V software build tools.

- For each custom instruction, the Nios V Embedded Design Suite (EDS) generates a macro in the system header file, **system.h**. You can use the macro directly in your C or C++ application code

- Reference: [Intel2022b]

Custom Instruction Software Interface

- During the build process, the Nios V software build tools generate <u>macros that allow easy access from application code to custom instructions</u>.
- The Nios V processor uses GCC built-in functions to map to custom instructions (`custom 0, r6, r7, r8`). Fifty-two built-in functions are available.
  - __builtin_custom_ <return type> n
  - Example 1:
    - #define ALT_CI_BITSWAP_N 0x00
    - #define ALT_CI_BITSWAP(A) __builtin_custom_ini(ALT_CI_BITSWAP_N,(A))
    - The built-in function __builtin_custom_ini() accepts a int as input, and returns a int.
    - Compiler gcc: __builtin_custom_ini(ALT_CI_BITSWAP_N,(A)) → `custom 0,…`

Custom Instruction Software Interface: CRC-32

- Example 2: CRC-32

`system.h` (GCC built-in functions to map to custom instructions)

/* Custom instruction macros */

```
#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)
```

- The built-in function __builtin_custom_inii() accepts two int values as input, and returns a int.
  - int __builtin_custom_inii (int n, int dataa, int datab);
- Reference: [Intel2022c]

Custom Instruction Assembly Language Syntax

- Nios V custom instructions use a standard assembly language syntax:
  - custom <selection index>, <Destination>, <Source A>, <Source B>
    - <selection index>—The 8-bit number that selects the particular custom instruction
    - <Destination>—Identifies the register where the result from the result port (if any) will be placed
    - <Source A>—Identifies the register that provides the first input argument from the dataa port (if any)
    - <Source B>—Identifies the register that provides the first input argument from the datab port (if any)

- You designate registers in one of two formats, depending on whether you want the custom instruction to use a Nios V register or an internal register:
  - r <i>—Nios V register <i>
  - c <i>—Custom register <i> (internal to the custom instruction component)

- The use of r or c controls the readra, readrb, and writerc fields in the the custom instruction word.
  - Custom registers are only available with internal register file custom instructions.

- Example, Nios V custom instruction: `custom 0, r6, r7, r8`

Custom Instruction Word Format

- R-type instruction: register-register
  - Source operands in registers A y B
  - Result in output register C
  - Operation code for all custom instructions: 0x32 (OP field)
- The 8-bit field N determines the ID of the custom instruction
- The field "register file selector" selects the location of the source and destination registers: Nios V architecture register file or custom register.

Figure 10. Custom Instruction Word Format
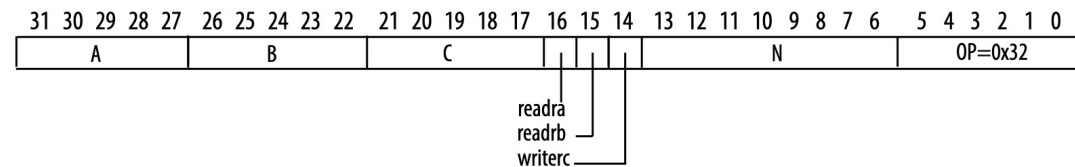
Table 5. Custom Instruction Fields

Field Name	Purpose	Corresponding Signal
A	Register address of input operand A	
B	Register address of input operand B	
C	Register address of output operand C	
readra	Register file selector for input operand A	readra
readrb	Register file selector for input operand B	readrb
writerc	Register file selector for ouput operand C	writerc
N	Custom instruction select index (optionally includes an extension index)	
OP	custom opcode, 0x32	n/a

The register file selectors determine whether the custom instruction component accesses Nios II processor registers or custom registers, as follows:

Table 6. Register File Selection

Register File Selector Value	Register File
0	Custom instruction component internal register file
1	Nios II processor register file

Hardware implementation of the customized Nios V processor

- The hardware of the customized Nios V software processor is built using the FPGA desing framework called Quartus from Intel.

- In this work, the customized Nios V processor has been built using Quartus and the Verilog files: CRC_Component.v, CRC_Custom_Instruction.v

- Quartus provides several files that can be downloaded on the AC Moodle page: `AC_CI_CRC.sof`, `AC_CI_CRC.sopcinfo`. These files configure the FPGA of the DE0-Nano board. In this way, software programs that use the CRC-32 custom instruction can be executed.

Software implementation of the CRC-32 algorithm

- The software of the CRC-32 algorithm is built using compiler tools of Nios V.
- Source code is provided using the following files:

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

- Executable file has *.elf extension.

Software Engineering

- This section of the project describes the design of programs that implement the CRC-32 algorithm.

- Three types of programs are developed:
  - Slow version
  - Fast version
  - Custom instruction version

Using the 8 custom instructions, n=0,…,7 (ci_crc.c)

- #define ALT_CI_NEW_COMPONENTCRC_0_N 0x0

- #define CRC_CI_MACRO(n, A)
  __builtin_custom_ini(ALT_CI_NEW_COMPONENTCRC_0_N + (n & 0x7), (A))
  - /*The n values and their corresponding operation are as follow:
  - * **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 */

Examples of compiler output when the custom instructions are used

C code

Machine code

Assembler

- CRC_CI_MACRO(0,0);
  - 70:   0005883a   mov   r2,zero
  - 74:   1005c032   custom 0,r2,r2,zero

- CRC_CI_MACRO(3, *(unsigned long *)input_data_copy);
  - 84:   18800017   ldw   r2,0(r3)
  - 88:   1005c0f2   custom 3,r2,r2,zero

Second operand is inhabilited

Memory address

ID of the custom instruction

Destination register

Source operand
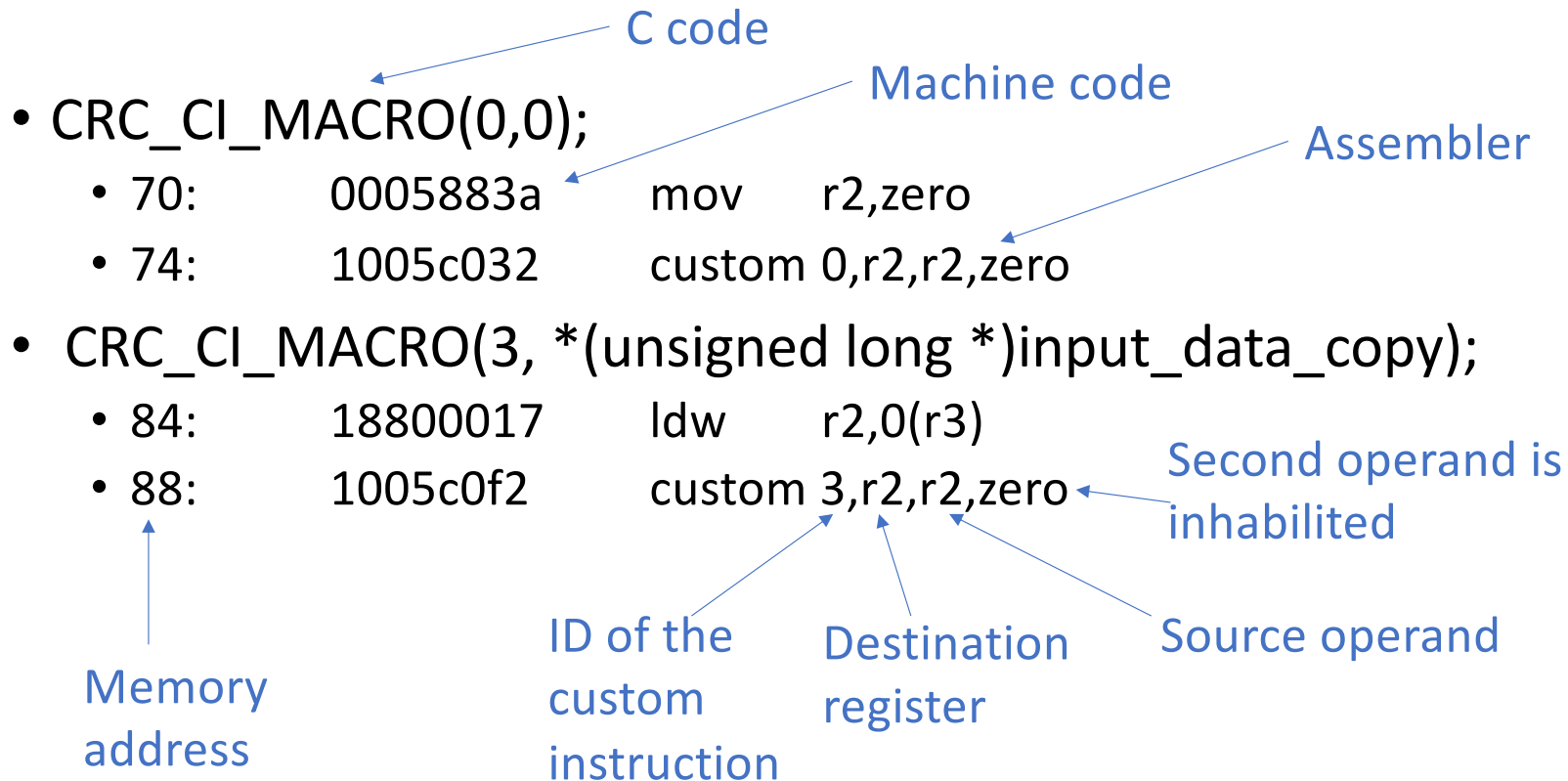
Obtaining executable program *.elf

- Decompress: AC_CI_CRC.rar
- Open *Nios V Command Shell* using a computer at the Laboratorio 2-4
- `cd` to folder CRC_CI
- Modify files *.c y *.h
- `$ make` → compile and link; if all is OK, executable file: CRC_CI.elf

Executing *.elf

- Open *Nios V Command Shell* using a computer at the Laboratorio 2-4
- cd to the folder where files *.sof y *.elf are storaged
- `$ quartus_pgm -c 1 -m JTAG -o "AC_CI_CRC.sof@1"` → FPGA of the DE0-Nano board is configured
- `$ niosv-download -r -g` CRC_CI.elf → executable program is loaded into the SDRAM memory of the DE0-Nano board; then, program is executed using the Nios V/g processor
- Open another *Nios V Command Shell*: `$ juart-terminal` → output results of the program can be seen

Software verification for 1-byte input data

Performance evaluation using the CRC-32 algorithm

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

Software version	Nios V mode	Processing time	Speed-up
Modulo 2 division implementation	standard		
Lookup table implementation	standard		
Use custom instruction	customized		