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

Lab Assignment 5:

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
- Performance Evaluation

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 lab assignment

- 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 and 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:

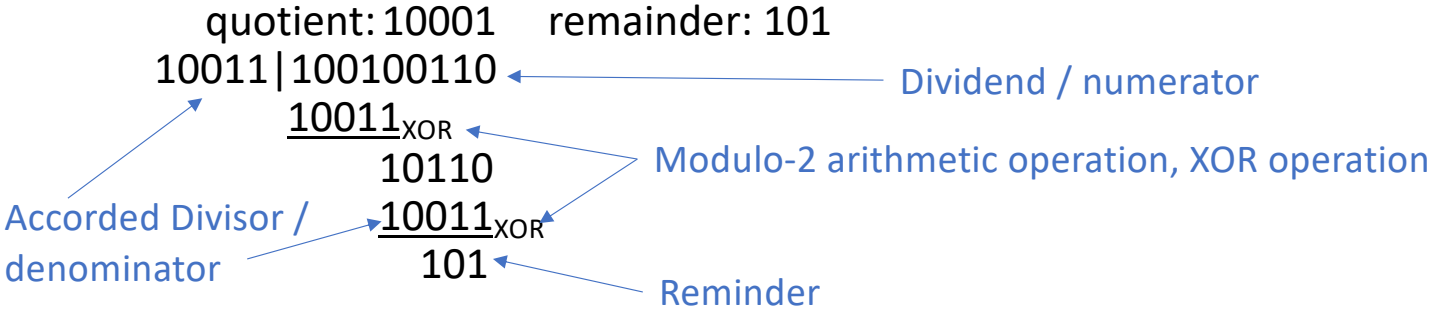
```
   quotient: 10001    remainder: 101
10011|100100110
       10011XOR
           10110
           10011XOR
             101
```

Dividend / numerator

Modulo-2 arithmetic operation, XOR operation

Accorded Divisor / denominator

Reminder



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

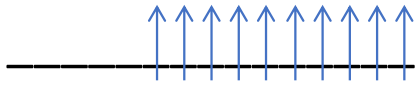
```
            1    remainder: 1010
11001|10011
       11001XOR
        1010
```

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

1100001010 = Quotient (nobody cares about the quotient)



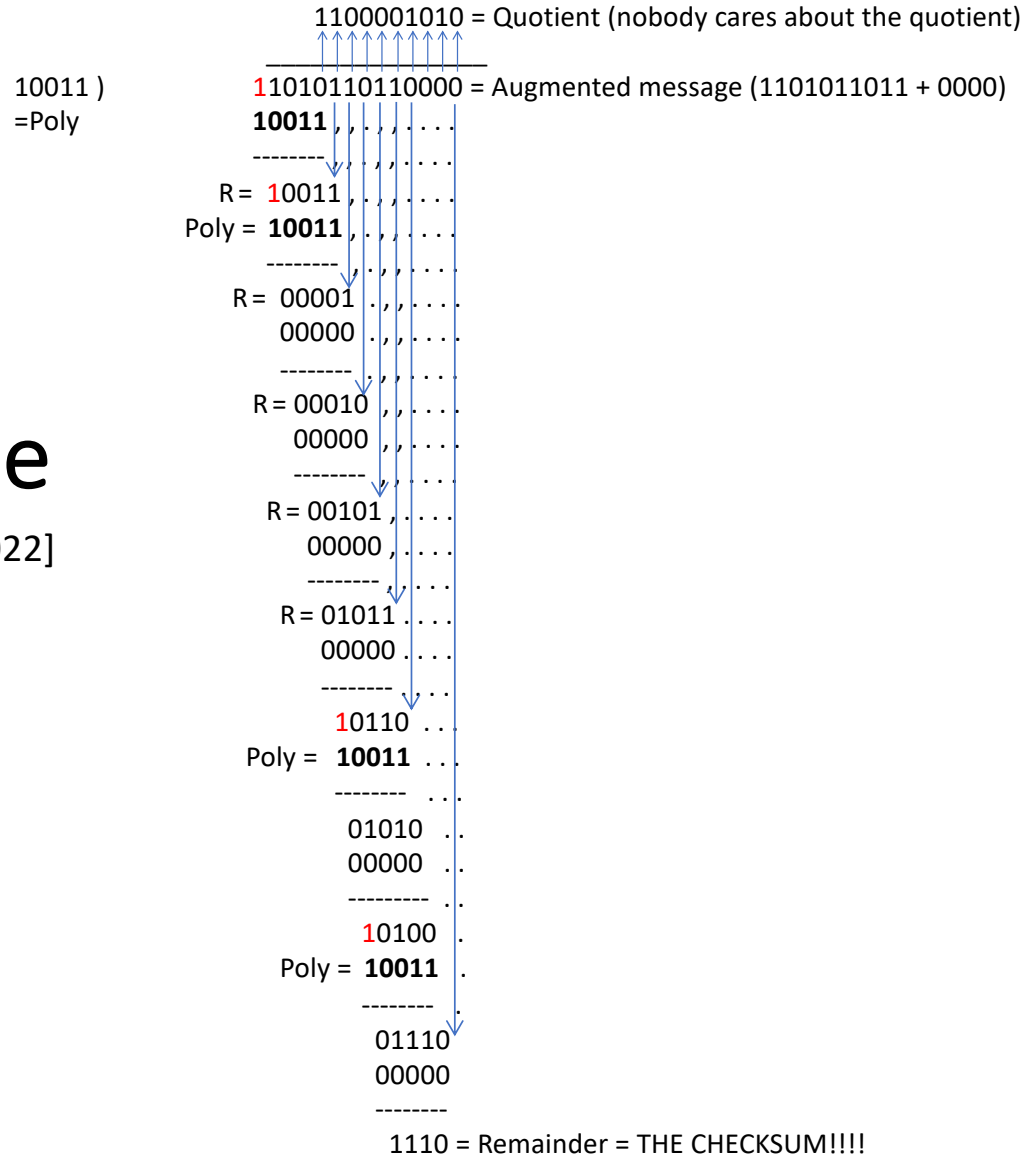
10011) 11010110110000 = Augmented message (1101011011 + 0000)

=Poly

1110 = Remainder = THE CHECKSUM!!!!

CRC example

Reference: [Stackoverflow2022]



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;
  int             byte;
  unsigned char   bit;

  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;
      }
      else  {
        remainder = (remainder << 1);
      }
    }
  }
  return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
}
```

Pre-loop

Loop

Post-loop

0xFFFFFFFF

Shift register

0x04C11DB7

0xFFFFFFFF

- 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.

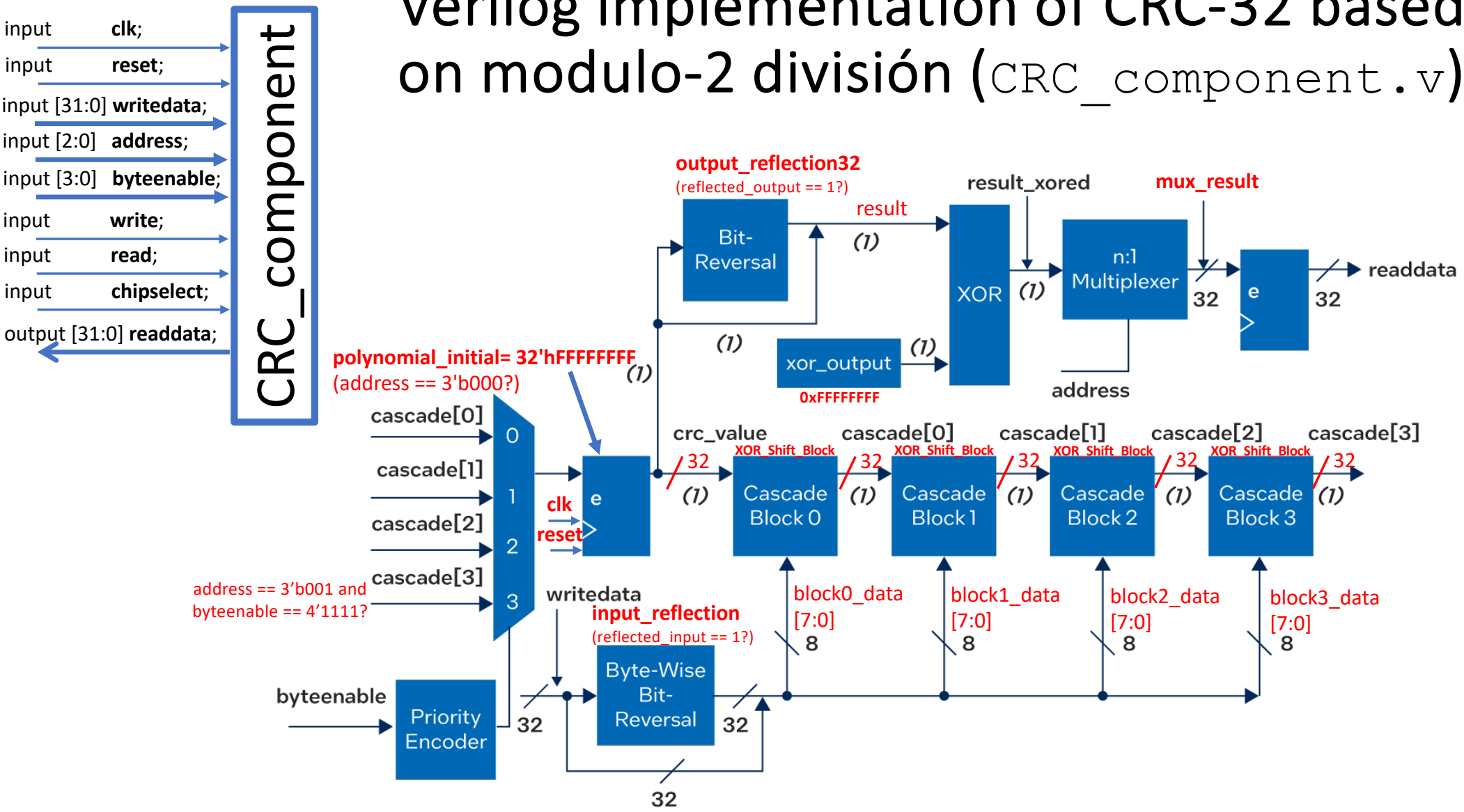
Hardware Engineering of Nios V custom instructions for the CRC-32 algorithm

- 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.

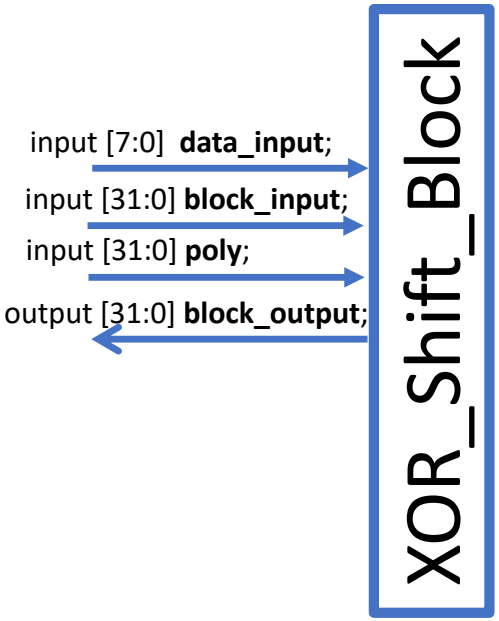
Verilog [Nyasulu1993]

- 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.

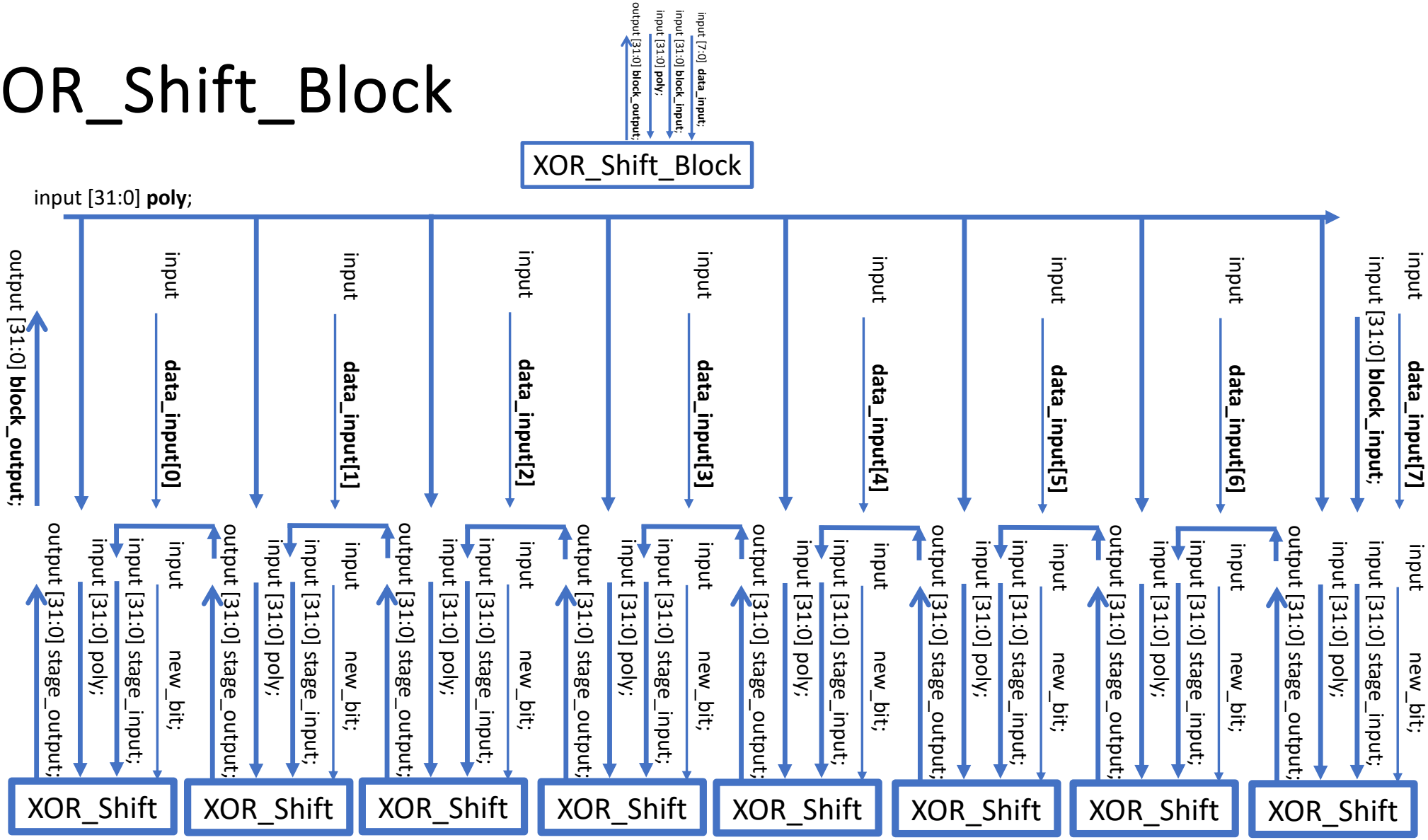
Verilog implementation of CRC-32 based on modulo-2 division (`CRC_component.v`)



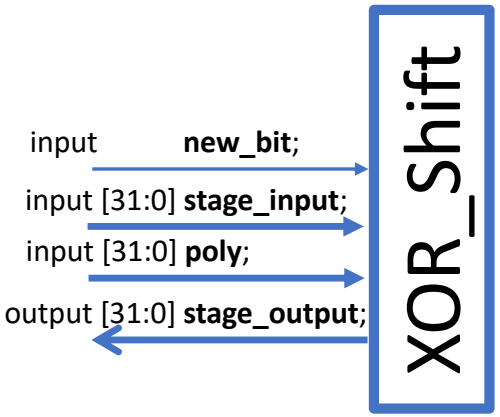
Verilog module: XOR_Shift_Block

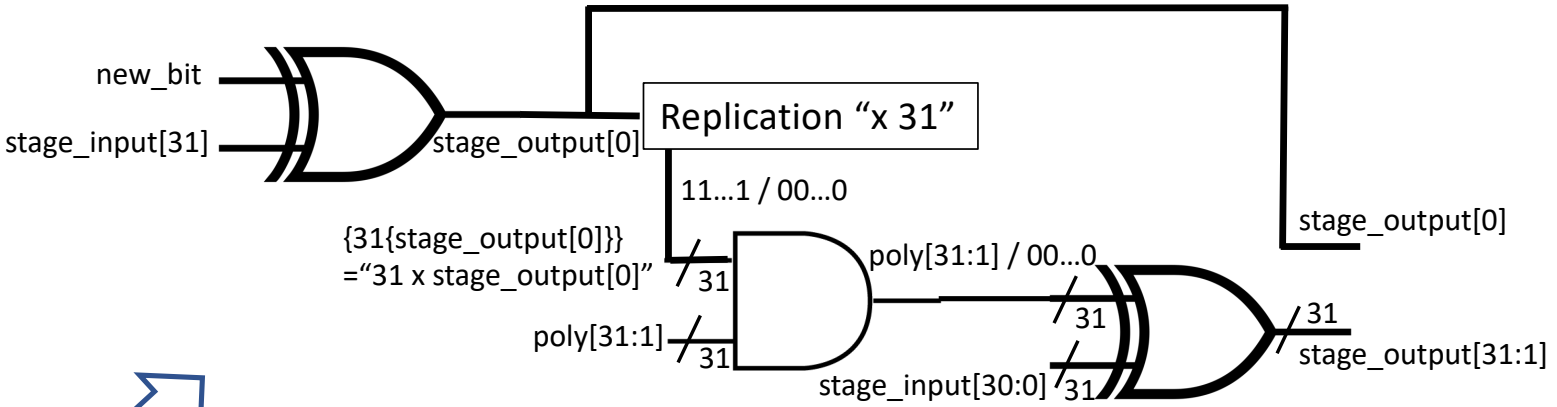


XOR_Shift_Block



Verilog module: XOR_Shift, modulo-2 division





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;
 crc             dumy2, shifted_data;
 int             byte;
 unsigned char   bit, dumy;
 for (byte = 0; byte < nBytes; ++byte)  {
      shifted_data = (REFLECT_DATA(message[byte]) << (WIDTH - 8));
    for (bit = 8; bit > 0; --bit)  {
       dumy2 = remainder ^ shifted_data;
       dumy  = (unsigned char) ((dumy2 & TOPBIT)>>31);
       if (dumy) {
          remainder = (remainder & DOWNBIT) ^ (POLYNOMIAL >> 1);
       }
       remainder = (remainder << 1) | dumy;
       shifted_data = (shifted_data << 1);
    }
  }
 return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);
}
```

0xFF FF FF FF (32 bits)

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

32

Loop

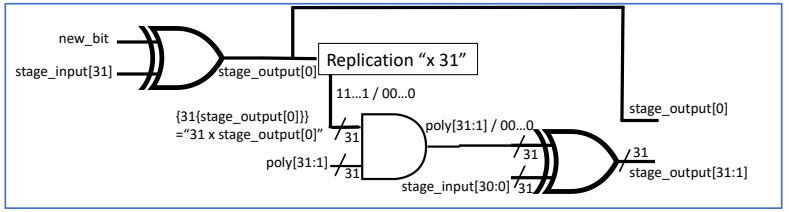
// new_bit= shifted_data[31]
// stage_input[31]= remainder[31]
// dumy2 = new_bit ^ stage_input[31]
// dumy   = stage_output[0]= (new_bit ^ stage_input[31]) >> 31

// stage_output[0] ==? 1

0x7F FF FF FF 0x04 C1 1D B7

// stage_output[0:31] :: remainder

// input message is left shifted in all cases



Post-loop

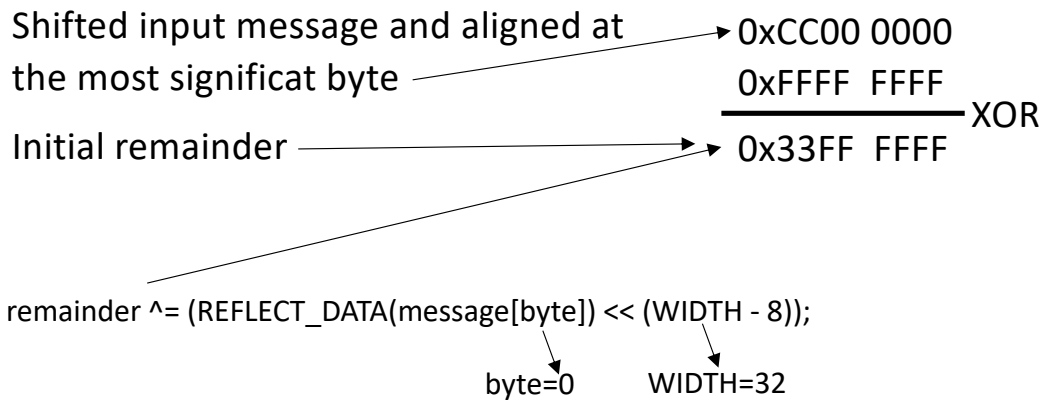
0xFF FF FF FF

Example of the CRC-32 implementation: pre-loop

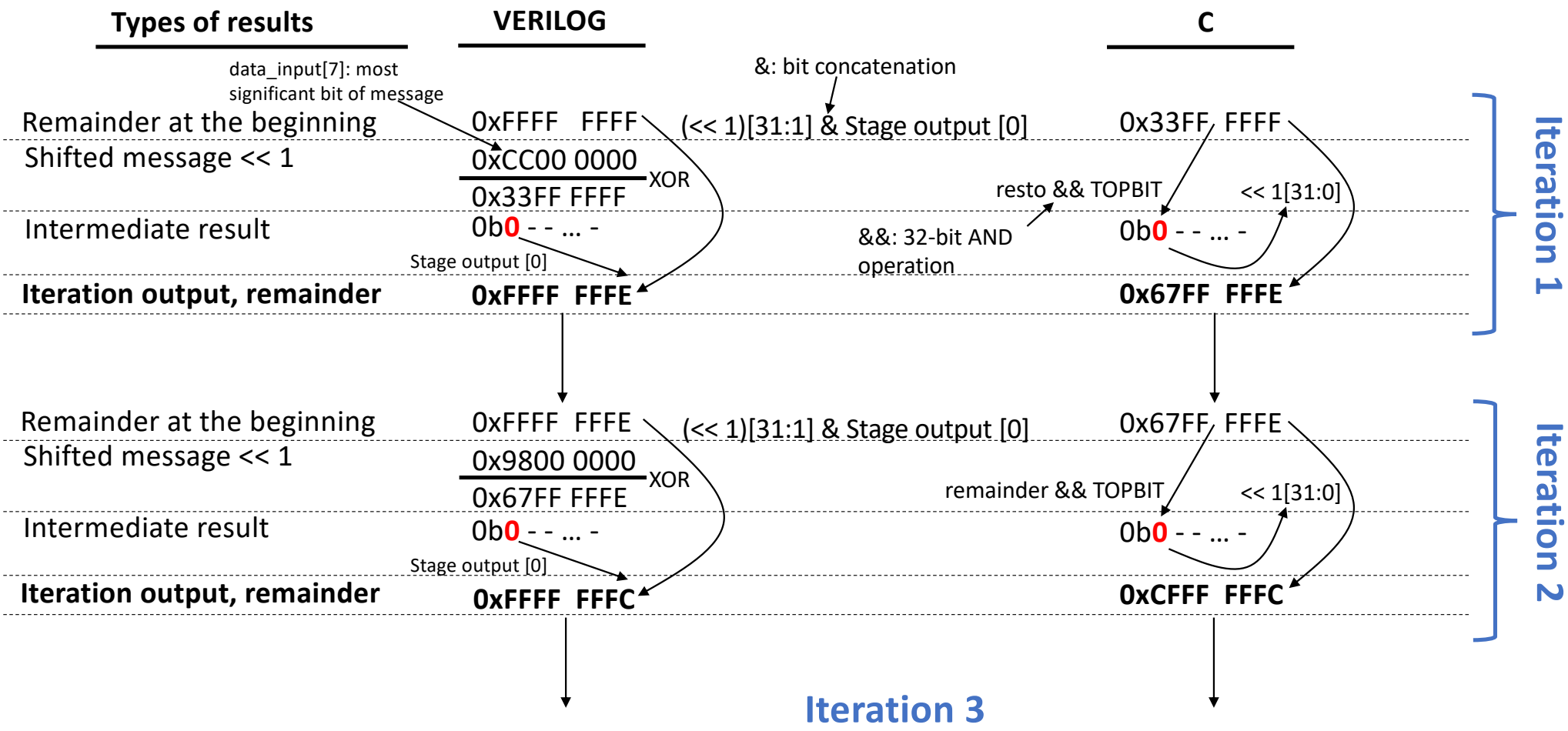
Input message 8-bit	0011 0011	0x33
Reflected input message 8-bit	1100 1100	0xCC
Shifted and aligned message 32-bit	1100 1100 0000 0000 0000 0000 0000 0000	0xCC00 0000
Polynom 32-bit	0000 0100 1100 0001 0001 1101 1011 0111	0x04C1 1DB7
Initial remainder 32-bit	1111 1111 1111 1111 1111 1111 1111 1111	0xFFFF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

	VERILOG
Pre-loop	No operation
Initial remainder	0xFFFF FFFF

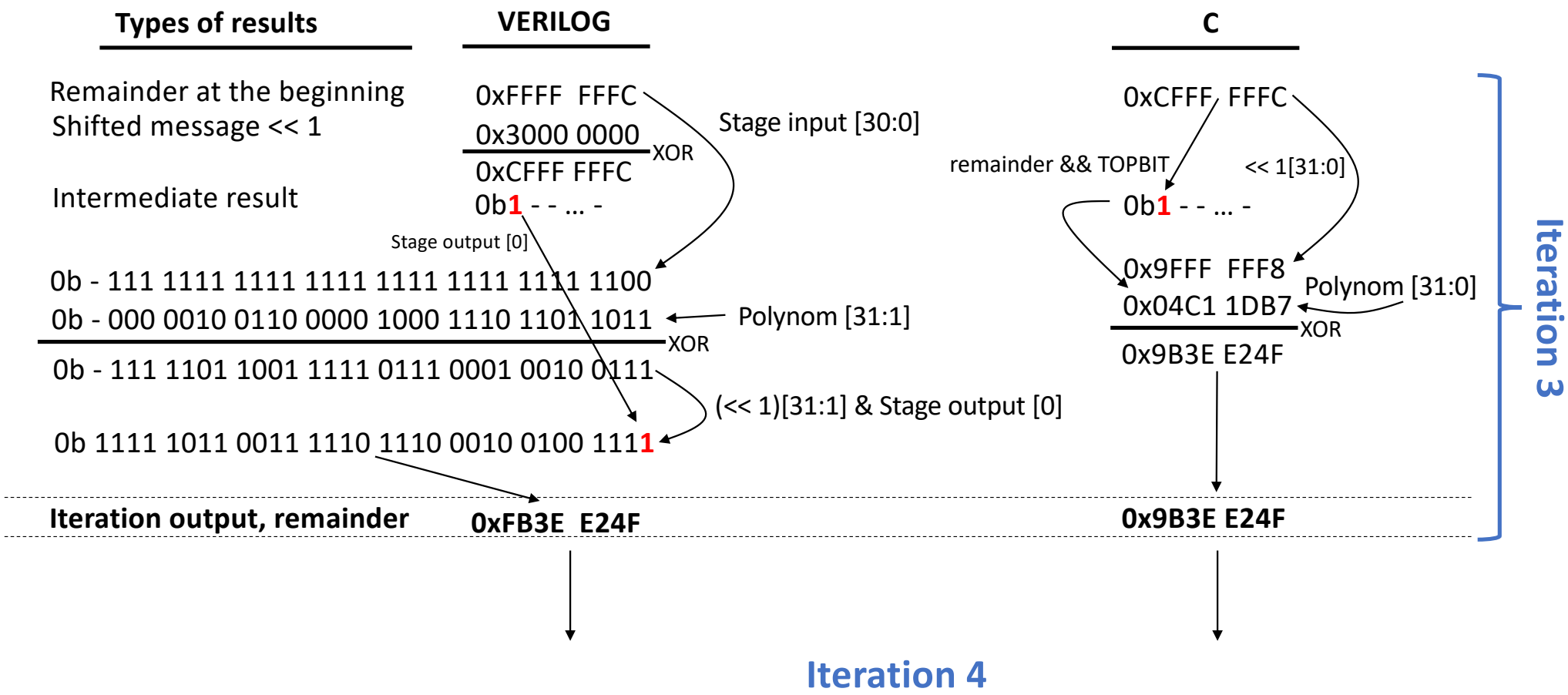
C (crc_slow())



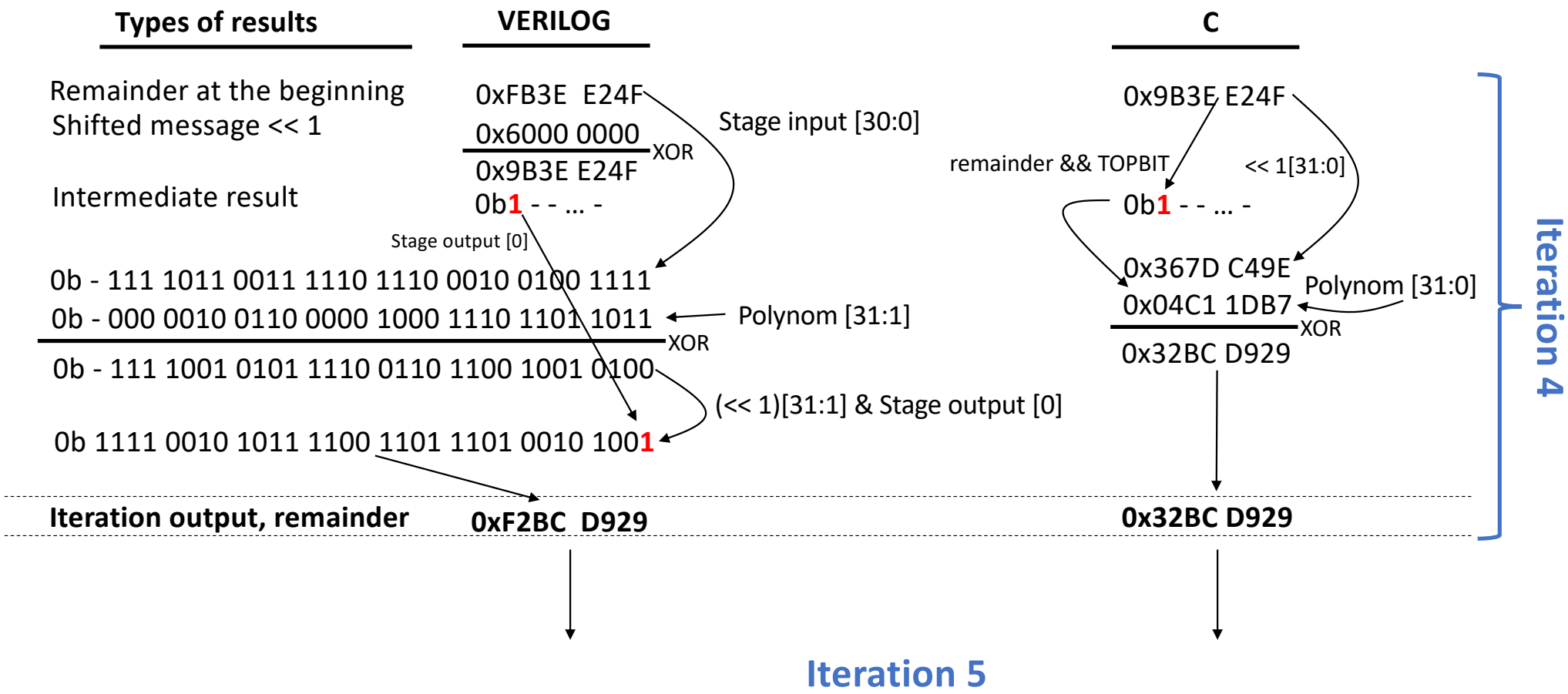
CRC-32 implementation: loop, iteration 1…8



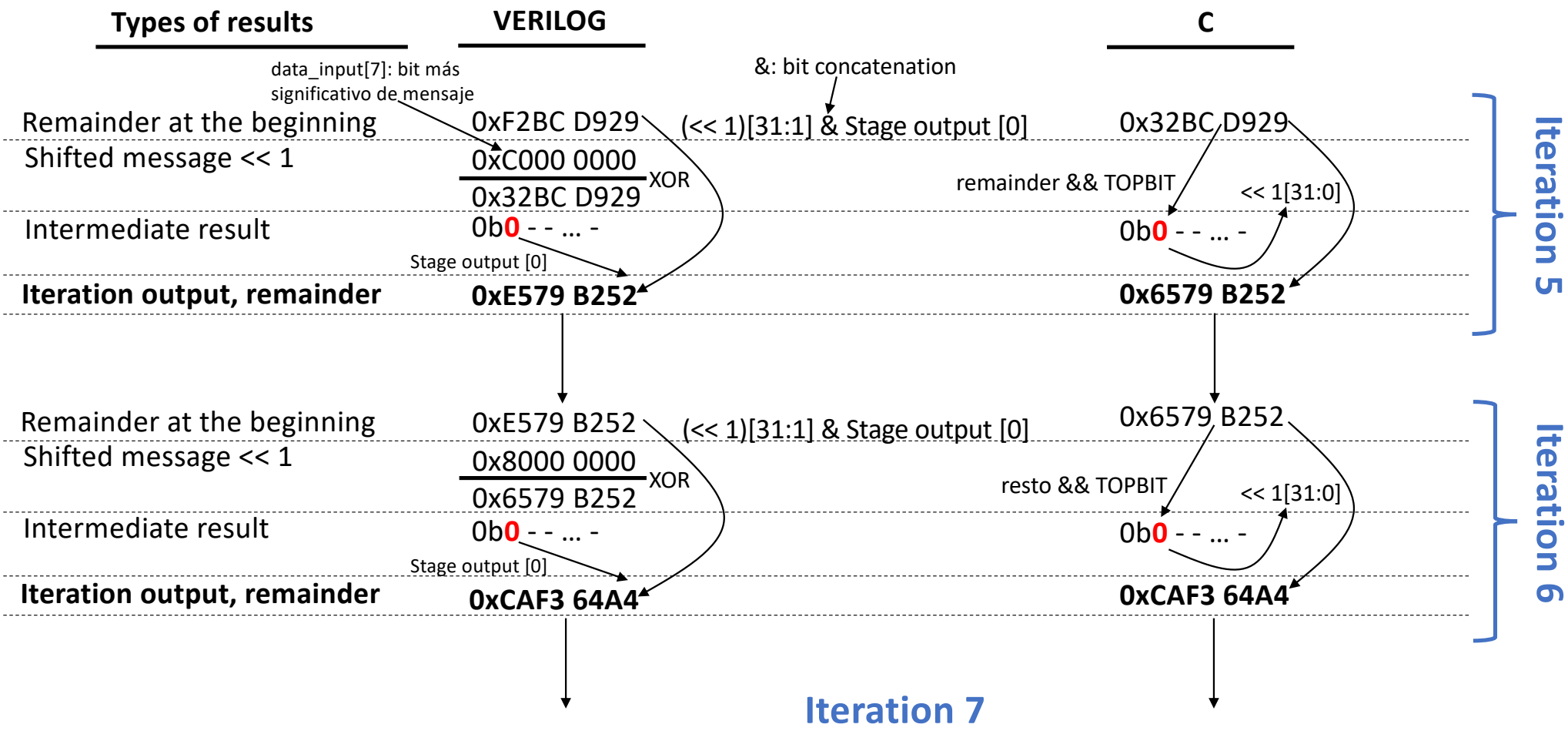
CRC-32 implementation: loop, iteration 1…8



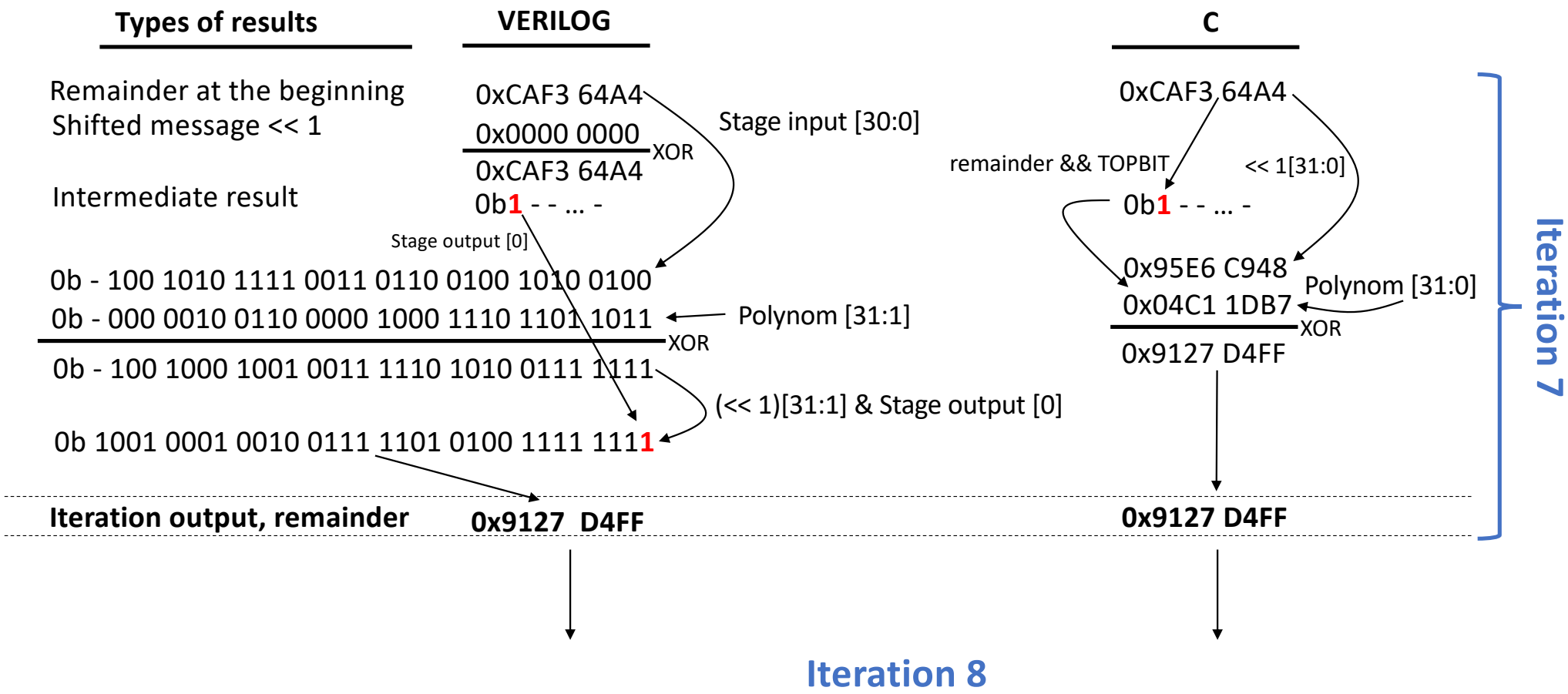
CRC-32 implementation: loop, iteration 1…8



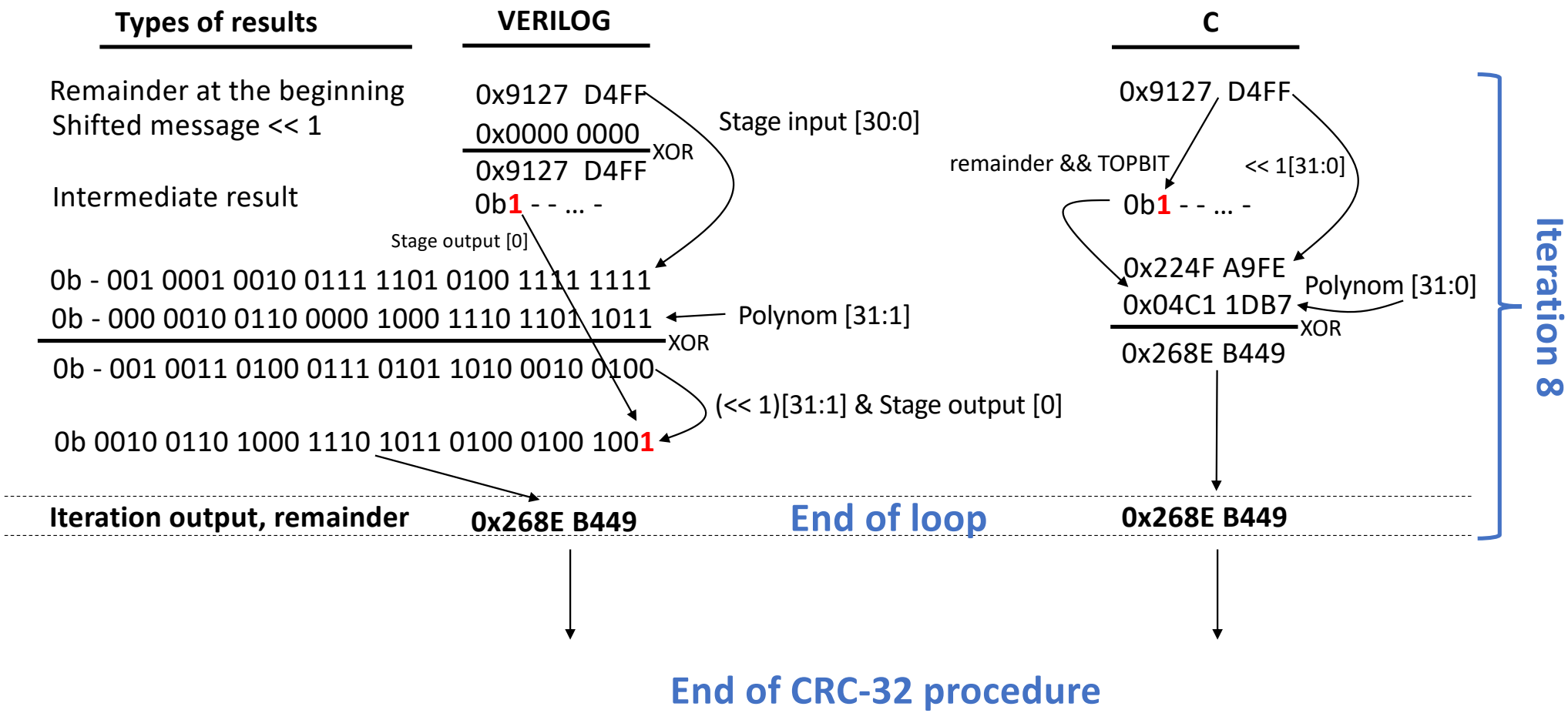
CRC-32 implementation: loop, iteration 1…8



CRC-32 implementation: loop, iteration 1…8



CRC-32 implementation: loop, iteration 1…8



CRC-32 implementation: final output

VERILOG

Remainder, loop output **0x268E B449**

Bit-reversal **0x922D 7164**

XOR_output **0xFFFF FFFF**

XOR

readdata **0x6DD2 8E9B**

C

0x268E B449

REFLECT_REMAINDER(remainder) **0x922D 7164**

FINAL_XOR_VALUE **0xFFFF FFFF**

return (REFLECT_REMAINDER(remainder) ^ FINAL_XOR_VALUE);

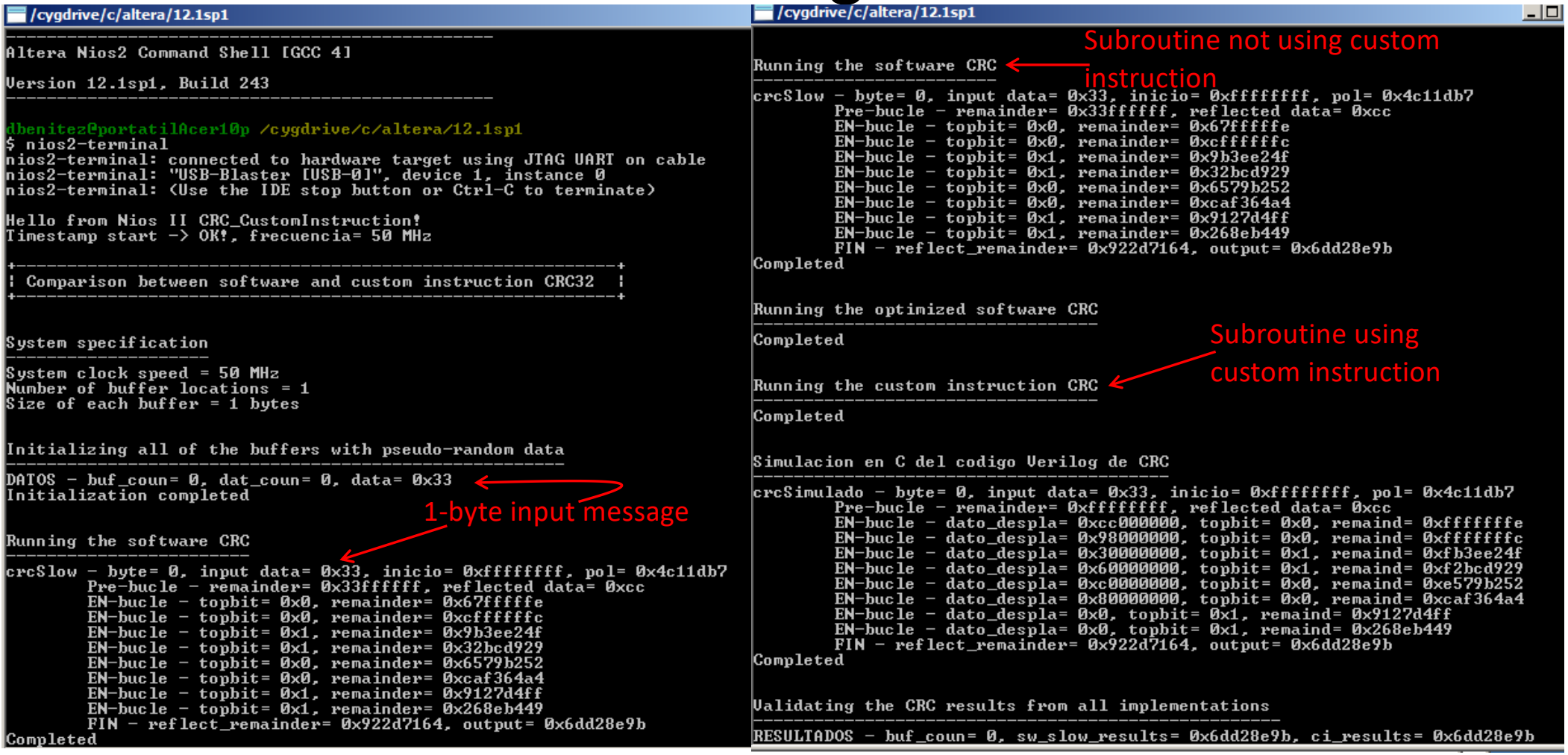
XOR

0x6DD2 8E9B

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



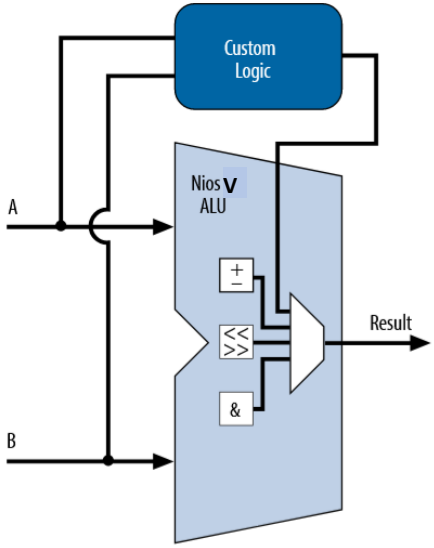
Computer Architecture

- This section of lab assignment 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/g 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: [Intel2023]

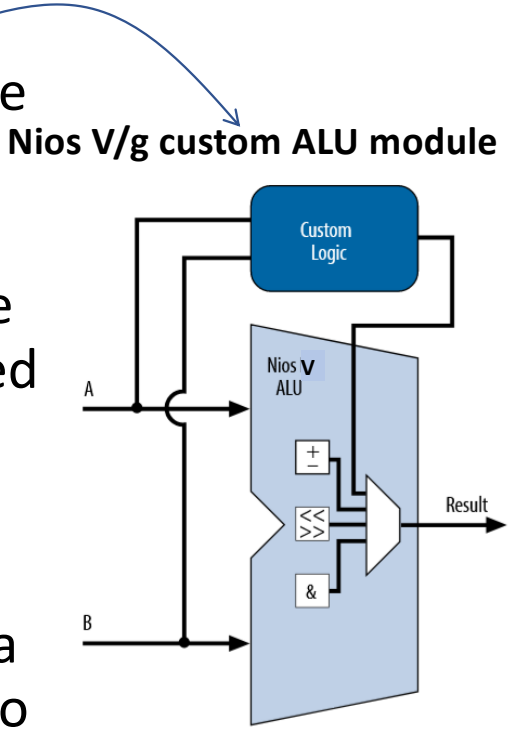
Nios V/g custom ALU module



The custom instruction logic connects directly to the Nios V/g arithmetic logic unit (ALU)

Custom Instruction Implementation

- Nios V/g 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 module of Quartus Prime software. Platform Designer exports the selection index value to the **`system.h`** file for use by the Nios V software build tools.
- For each custom instruction, the Nios V Command Shell generates a macro in the system header file, **`system.h`**. You can use the macro directly in your C or C++ application code
- Reference: [Intel2023]



Custom Instruction Software Interface

- During the build process, the Nios V software build tools generate macros that allow easy access from application code to custom instructions.
- The Nios V/g 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 <parameter types>
  - 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 an `int` variable as input, and returns a `int`.
    - gcc compiler : __builtin_custom_ini(ALT_CI_BITSWAP_N,(A)) → `custom 0,` …

Custom Instruction Software Interface: CRC-32

- Example 2: CRC-32

`system.h` includes 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 an `int` variable.
  - `int __builtin_custom_inii (int n, int dataa, int datab);`
- Reference: [Intel2023]

Custom Instruction Software Interface: CRC-32

- Example 2: CRC-32

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

```
/* * NiosV Custom instruction macros * */
#define ci_crc(VAL_1, VAL_2, F3) ({ \
  int output; \
  asm volatile (".insn r 0x0B, %[FUNCT3], 0x0, %[out], %[input1], %[input2]" \
            : [out] "=r" (output) \
            : [input1] "r" (VAL_1), [input2] "r" (VAL_2), [FUNCT3] "i" (F3)); \
            output;  \
})
```

funct3=F3 funct7=0x0

- Reference: [Intel2023a]

Instruction format of custom instructions: `opcode+rd+funct3+rs1+rs2+funct7`

Table 3. 32-bit Custom Instruction Word (ctrl)

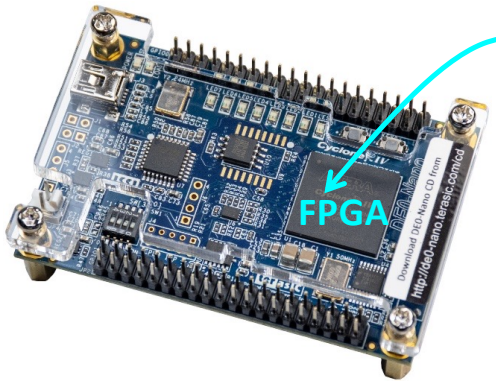
32-bit Custom Instruction Bit Field															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
funct7[6:0]							rs2					rs1			
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
rs1	funct3[2:0]			rd			opcode								

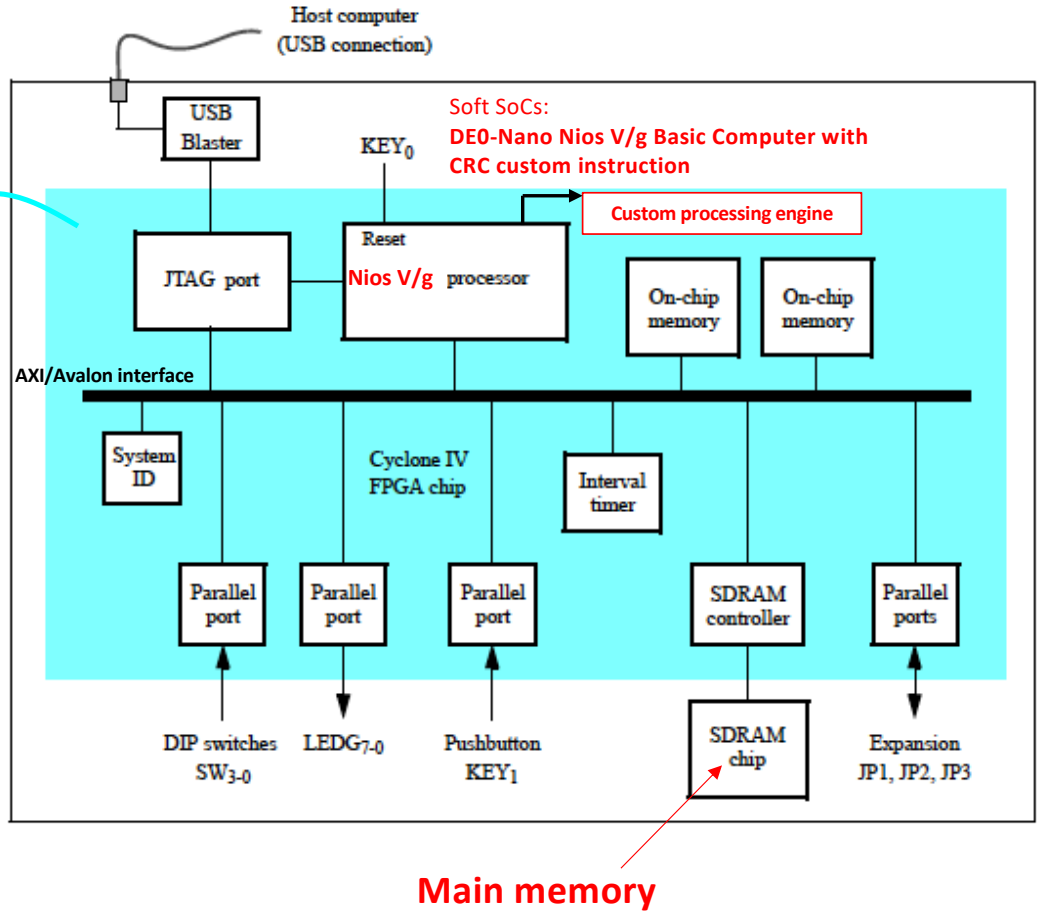
Hardware implementation of the customized Nios V/g soft processor

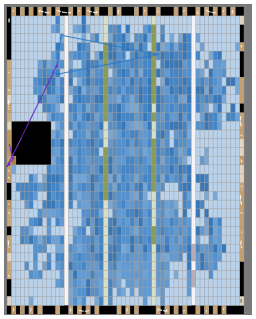
- The hardware of the customized Nios V/g soft processor is built using the FPGA design framework called Quartus Prime from Intel.
- In this work, the customized Nios V/g processor has been built using Quartus and the Verilog files: `CRC_Component.v`, `CRC_Custom_Instruction.v`
- Quartus Prime provides several files that can be downloaded on the github repository: **`DE0_Nano_Basic_Computer_26nov24.sof`**, **`DE0_Nano_Basic_Computer_26nov24.sopcinfo`**. The `sof` file configure the FPGA of the DE0-Nano board. In this way, software programs that use the CRC-32 custom instruction can be executed.

Soft SoC based on a customized Nios V/g soft processor

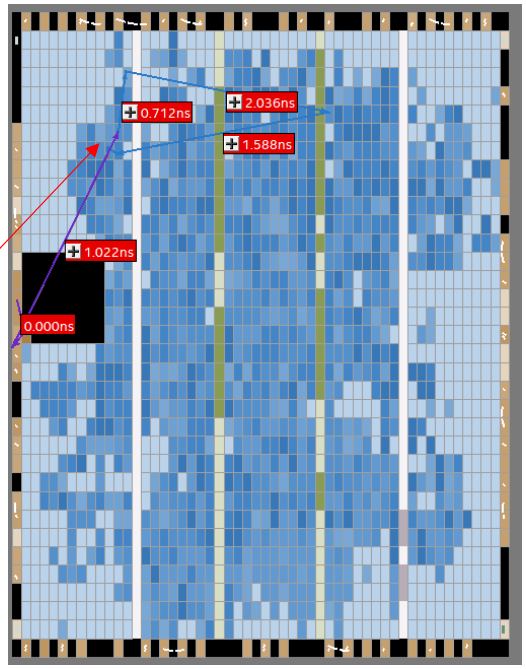
Terasic DE0-Nano board



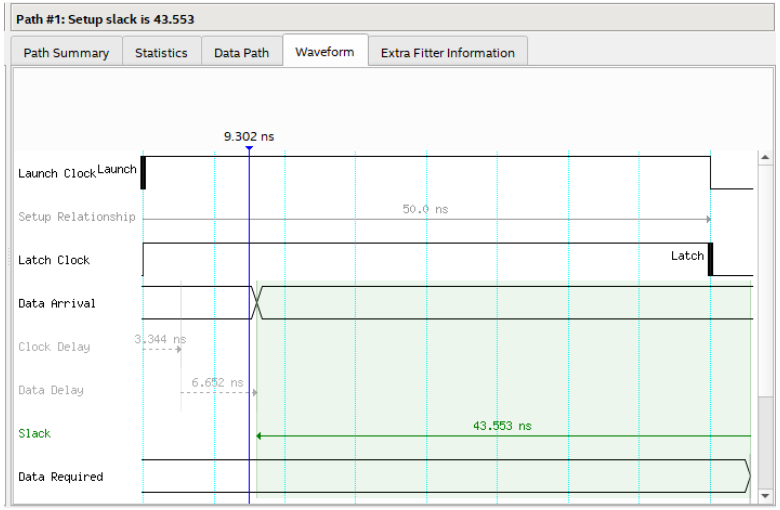




Compiled FPGA design



Quartus Prime Version	23.1std.0 Build 991 11/28/2023 SC Standard Ed.
Top-level Entity Name	DE0_Nano_Basic_Computer
Family	**Cyclone IV E**
Device	**EP4CE22F17C6**
Total logic elements	13,108 / 22,320 **(59 %)**
Total registers	7262
Total pins	133 / 154 (86 %)
Total memory bits	207,504 / 608,256 **(34 %)**
Embedded Multiplier 9-bit elements	8 / 132 (6 %)
Total PLLs	1 / 4 (25 %)
Fmax	**86.63 MHz** (Slow 1200mv 0C Model)



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

Obtaining executable program *.elf

```
$ mkdir lab5 app
$ mkdir lab5 bsp
$ cd lab5 bsp
$ sh
$ niosv-bsp.exe -c -t=hal -s=nios_system_26nov24.sopcinfo
settings.bsp
$ cd ../lab5 app
$ niosv-app.exe -a=. -b=../lab5 bsp -s=.
$ cmake -S . -G "Unix Makefiles" -B build
$ make -C build
```

Executing *.elf

- Open *Nios V Command Shell* using a desktop computer at the Computer Architecture laboratory

`$ cd` to the folder where files `*.sof` y `*.elf` are stored

```
$ jtagconfig.exe
```

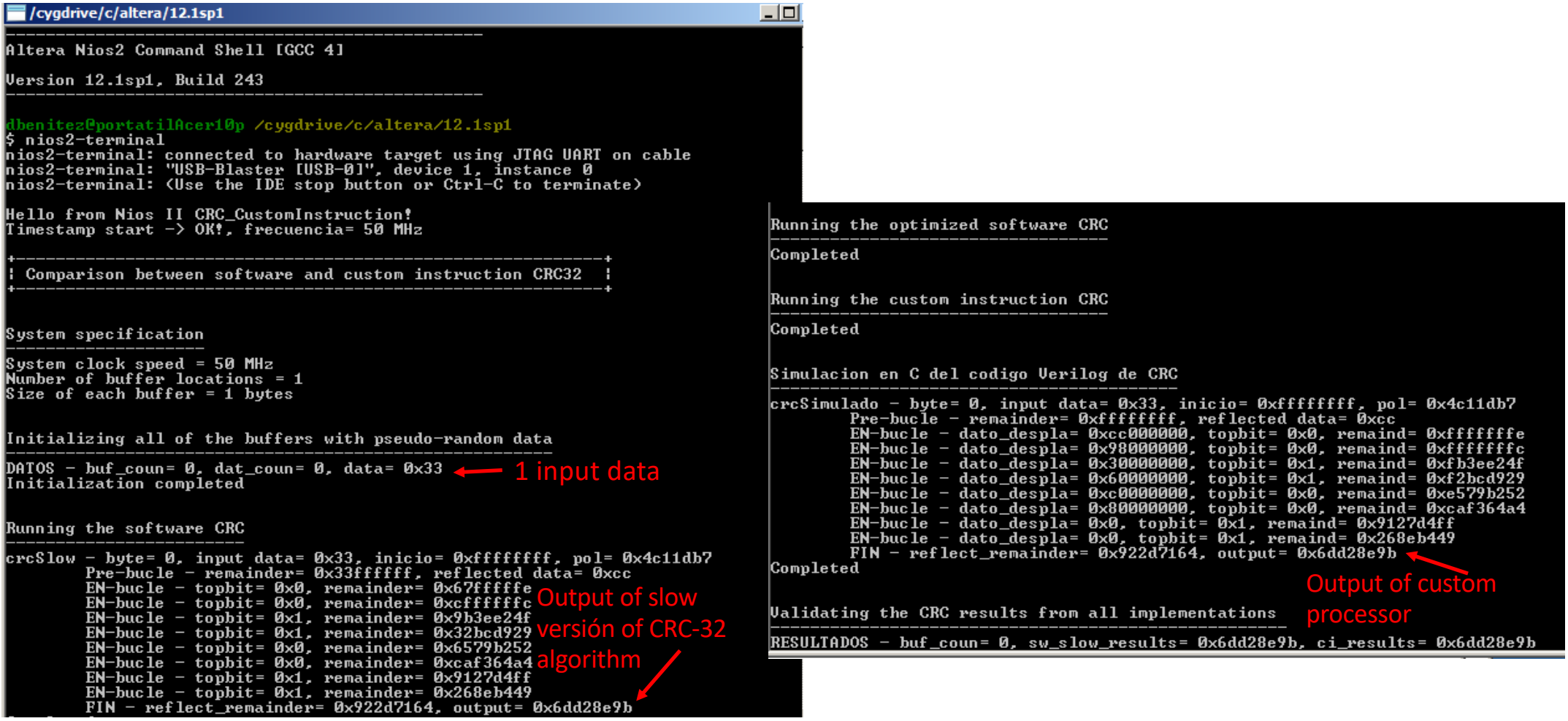
```
$ quartus pgm -c 1 -m JTAG -o
"p;DE0_Nano_Basic_Computer_26nov24.sof.sof@1"
```
→ FPGA of the DE0-Nano board is configured

`$ niosv-download.exe -g build/lab5_app.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.exe` → 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/g instructions	Execution time	Speed-up
Modulo 2 division implementation (slow)	standard		
Lookup table implementation (fast)	standard		
Using custom instruction	customized		

Bybliography

- [Intel2023a] Intel, AN 977: Nios V Processor Custom Instruction
  - URL:https://cdrdv2-public.intel.com/776470/ug-683632-776470.pdf
- [Intel2023b] Intel, Intel Agilex 7 FPGA Custom Instruction Design on Nios V/g processor – CRC
  - URL: https://www.intel.com/content/www/us/en/design-example/789503/agilex-7-crc-custom-instruction-design-on-nios-v-g-processor.html
- [Intel2022a] Intel, Nios II CRC Acceleration Design Example
  - URL: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/exm-crc-acceleration.html
- [Molkenthin2019] Bastian Molkenthin; Understanding and implementing CRC (Cyclic Redundancy Check) calculation
  - URL: http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html . 2019
- [Nyasulu1993] P. M. Nyasulu and J. Knight. Introduction to Verilog. Carleton University.
  - URL: https://www.cs.upc.edu/~jordicf/Teaching/secretsofhardware/ VerilogIntroduction_Nyasulu.pdf, 2003.

Bybliography

- [OSDev] OSDev.org; CRC32
  - URL: https://wiki.osdev.org/CRC32
- [Stackoverflow2022] Stackoverflow; How is a CRC32 checksum calculated?
  - URL: https://stackoverflow.com/questions/2587766/how-is-a-crc32-checksum-calculated
- [Wikipedia] Wikipedia, Cyclic redundancy check
  - URL: https://en.wikipedia.org/wiki/Cyclic_redundancy_check
- [Williams1993] Ross N. Williams; A Painless Guide to CRC Error Detections Algorithms
  - URL: http://chrisballance.com/wp-content/uploads/2015/10/CRC-Primer.html . Section 6. 1993.