SADIQ M. SAIT, BENTEN MS, KHAN AMT
DESIGNING ASICS WITH UAHPL
IEEE CIRCUITS AND DEVICES MAGAZINE 11 (2): 14-24 MAR 1995

HARDWARE DESIGN AND VLSI IMPLEMENTATION OF A BYTE-WISE CRC GENERATOR CHIP

Sadiq M. Sait and Wasif Hasan
Department of Computer Engineering
King Fahd University of Petroleum and Minerals
Dhahran, Saudi Arabia

Abstract

In this paper the hardware design and VLSI implementation of a byte-wise CRC generator is presented. The algorithm is based on the work presented in [10] in which a software implementation was proposed. The byte-wise CRC algorithm is translated to hardware and expressed in AHPL [6]. The method used here calculates CRC 'on the fly' and is much faster than the look-up table method proposed by Lee [5]. The chip is 8 times faster than the serial implementation of [12] with smaller hardware requirements (occupies lesser area). The number of clock cycles required to generate and transmit any CRC (for an 8 byte message) is just two more than the time required to calculate it (in all 10 clock pulses). The CRC chip can be used in a number of applications. These include areas such as error detection and correction in data communications, signature analysis, and mass storage devices for parallel information transfers.

1 Introduction

Data corruption is the principal problem associated with the storage and transmission of data. Whenever data is transmitted over communication channels, there is always a finite probability of occurrence of some errors. The errors may occur due to a number of factors, some of which are radiation, atmospheric conditions, fading of signals, interference between channels etc. Hence it is of paramount importance that the receiving module must be able to differentiate between an error free message and an erroneous one. There are various methods which are used to detect errors. Most of them do so by introducing some extra (redundant) bits exclusively for the purpose of detecting errors. Some of the commonly used approaches for error detection in data communications include parity check codes (vertical and horizontal), longitudinal redundancy check (LRC) codes, checksums, and cyclic redundancy check (CRC). The most widely used method for error detection is CRC.

In the modules using CRC, the data bits are represented as a binary polynomial, $m(D)$, with coefficients $m_{K-1}, m_{K-2}, \ldots, m_0$ where K is the length of the string of data bits.

$$m(D) = m_{K-1}D^{K-1} + m_{K-2}D^{K-2} + \ldots + m_0 \quad (1)$$

The CRC is represented as another polynomial given below.

$$C(D) = C_{L-1}D^{L-1} + \cdots + C_1 D + C_0 \quad (2)$$

where L is the length of the CRC. The frame to be transmitted (including the message and CRC) can be represented as:

$$F(D) = m(D)D^L + C(D) \quad (3)$$

The CRC polynomial $C(D)$ is defined in terms of a *generator polynomial* $g(D)$ which is a polynomial of degree L with binary coefficients that specify the particular CRC code to be used. If we divide $m(D)D^L$ by $g(D)$, the remainder obtained is the CRC polynomial, i.e.,

$$C(D) = remainder\left[\frac{m(D)D^L}{g(D)}\right] \quad (4)$$

Let $q(D)$ be the quotient resulting from dividing $m(D)D^L$ by $g(D)$. Then we have

$$m(D)D^L = g(D)q(D) + C(D) \quad (5)$$

Subtracting $C(D)$ modulo-2 from both sides of the above equation and recognizing the fact that modulo-2 addition and subtraction are same we get

$$m(D)D^L + C(D) = g(D)q(D) \quad (6)$$

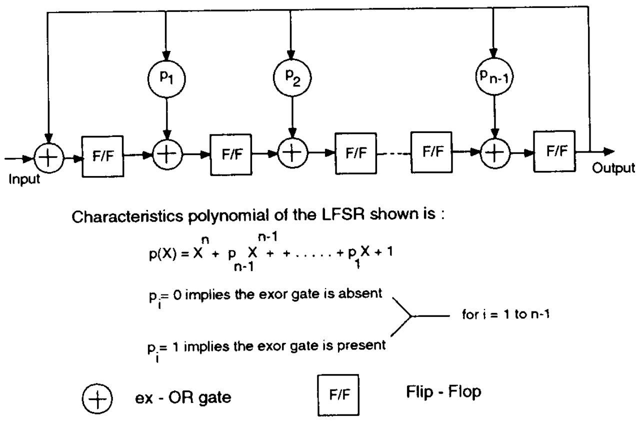
Hence all the code words are divisible by $g(D)$ and all polynomials divisible by $g(D)$ are code words. $C(D)$ is of degree at most $L-1$.

Thus the CRC algorithm calculates $C(D)$ and appends it to the data string being sent. At the receiver, the received message (with CRC appended) is divided by the generator polynomial $g(D)$. If the remainder is zero it is assumed that there are no errors

Contributed Paper
Manuscript received December 29, 1994

CRC-16	$D^{16} + D^{15} + D^2 + 1$
CRC-CCITT	$D^{16} + D^{12} + D^5 + 1$
CRC-12	$D^{12} + D^{11} + D^3 + D^2 + D + 1$
LRCC-16	$D^{16} + 1$
LRCC-8	$D^8 + 1$

Figure 1: Common CRC generator polynomials.



Input
Output
F/F
P_1
P_2
P_{n-1}
Characteristics polynomial of the LFSR shown is :
$p(X) = X^n + p_{n-1} X^{n-1} + \ldots\ldots + p_1 X + 1$
$p_i = 0$ implies the exor gate is absent
$p_i = 1$ implies the exor gate is present
for i = 1 to n-1
ex - OR gate
Flip - Flop

Figure 2: Shift register implementation of CRC generator.

in the received message or the errors have gone undetected due to one valid code word being erroneously converted to another. The probability of such an occurrence is 2^{-L}. If however the remainder is non-zero, the message is declared to be in error. There are various generator polynomials in use. The most common are given in Figure 1.

Implementation of CRC generators can be done both in software as well as hardware. Several algorithms to compute the CRC by considering a bit, a byte or even a word of the message have been suggested. Conventionally, bitwise CRC computations are implemented in hardware by means of a simple shift register, with exclusive-or (EXOR) logic gates. Such a shift register arrangement has been shown in Figure 2. It is popularly known as linear feedback shift register (LFSR), which handles one bit of the data stream at a time. This paper discusses a hardware implementation of a CRC generator chip which calculates CRC byte-wise using CRC-16 as the generator polynomial. Since the method is general, only slight modification in the hardware program will be needed to generate CRC for other generator polynomials. The CRC generator chip can be fitted on the data link control (DLC) module and may prove to be quite convenient and economical.

In the following sections, we discuss the need to calculate CRC byte-wise, the algorithm, the design considerations in brief, the hardware modeling and simulation in *Universal AHPL* [6], the process of VLSI layout generation and finally conclusions.

2 Need to Calculate CRC Bytewise

At the turn of the century an entire new set of applications is emerging which will require extremely high bit rates. Some such applications are HDTV, digital audio, high resolution color graphics, medical imaging, interconnection between mainframes, supercomputer support for research environments etc. Although the reliability of the physical media is likely to increase manifolds with the introduction of fiber-optic communication which has error rate 10 orders of magnitude lower than telephone lines, there is still some probability of error in the transmitted messages. Some of the applications may require ultra high precision and in such cases the use of CRC or other methods to quickly detect any errors occurring during transmission and/or propagation becomes imperative. The CRC checks being so versatile, powerful and easy to implement are obviously preferred over other methods of error detection. For any given message the CRC can detect the following types of errors:

1. All single bit errors.
2. All double bit errors as long as the generator polynomial, $g(D)$ has a factor with at least three terms.
3. All odd number of bit errors as long as $g(D)$ has a factor $(D + 1)$.
4. Any burst error for which for which the length of the burst is less than the length of the CRC.
5. Most larger burst errors.

Before a packet is transferred, the CRC is appended to it at the end. The message bits along with the header and trailer constitute what is known as the *frame* to be transmitted. So in order to attain high speed not only the transmission rate should be high but also the frame should be synthesized at rapid speed. Hence there is also a need to speed up the process of CRC generation which is the motivation behind calculating CRC byte-wise.

3 Algorithm Derivation

In order to calculate CRC byte-wise i.e., simultaneously for 8 bits of the message, we got to have an algorithm that will produce the same value of CRC as produced by the bitwise CRC method for 8 bits of the message. What happens at each of the eight shifts of a bitwise CRC calculation is shown in Figure 3 [10].

As is clear from the Figure, the final contents of the CRC register, after eight shifts is the exclusive-OR of various combinations of the input data byte and the previous contents of the CRC register. Hence the byte-wise CRC calculation algorithm must produce these same values. We can define a function X_i, to further simplify the contents of the CRC register. The function X_i is the exclusive-OR of the i^{th} bit of the input data byte with the i^{th} bit of the CRC register, i.e.,

$$X_i = C_i \oplus M_i \tag{7}$$

Hence we can say that the vector X (composed of X_i's) can be obtained by exclusive-ORing the low order byte of the CRC register and the input message byte. The new simplified version is also shown in the Figure 4. As can be observed from the Figure, that after 8 shifts

- the higher byte of the CRC register is a function of only the initial lower byte while
- the lower byte of the CRC register is a function of the initial higher byte of the CRC register, initial lower byte of the CRC register and the input message byte.

The final contents of the CRC register after eights shifts using CRC-CCITT as the generator polynomial has also been shown in Figure 4. From the above we can conclude that it is possible to calculate CRC byte-wise using the following algorithm:

1. Exclusive-OR the lower byte with the message byte to get the values of X_i.
2. Shift the higher order byte of the CRC register into the lower order byte and then discard the lower order byte.
3. Exclusive-OR a 16-bit word defined by everything below the dotted line (as shown in the Figure) with CRC register to get the new contents of the CRC register.

For VLSI synthesis, the hardware that implements the above given algorithm is first modeled in AHPL [6]. Details of hardware are described in the following sections.

SH	IN	CRC-16 REGISTER															
		R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
0		C16	C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2	C1
1	M_1	C1	C16	C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3	C2
		M1		C1													C1
				M1													M1
2	M2	C2	C1	C16	C15	C14	C13	C12	C11	C10	C9	C8	C7	C6	C5	C4	C3
		C1	M1	C2	C1												C2
		M1		C1	M1												C1
		M2		M1													M1
				M2													M2
								•									
								•									
								•									
								•									
8	M8	R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
		0	0	0	0	0	0	0	0	C16	C15	C14	C13	C12	C11	C10	C9
		C8	C7	C8	C7	C6	C5	C4	C3	C2	C1						C8
		M8	M7	M8	M7	M6	M5	M4	M3	M2	M1						M8
		C7	C6	C7	C6	C5	C4	C3	C2	C1							C7
		M7	M6	M7	M6	M5	M4	M3	M2	M1							M7
		C6	C5														C6
		M6	M5														M6
		C5	C4														C5
		M5	M4														M5
		C4	C3														C4
		M4	M3														M4
		C3	C2														C3
		M3	M2														M3
		C2	C1														C2
		M2	M1														M2
		C1															C1
		M1															M1

Figure 3: CRC register after eight shifts.

SH	IN	CRC-16 REGISTER															
		R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
		0	0	0	0	0	0	0	0	C16	C15	C14	C13	C12	C11	C10	C9
		X8	X7	X8	X7	X6	X5	X4	X3	X2	X1						X8
		X7	X6	X7	X6	X5	X4	X3	X2	X1							X7
		X6	X5														X6
		X5	X4														X5
		X4	X3														X4
		X3	X2														X3
		X2	X1														X2
		X1															X1

SH	IN	CRC-CCITT REGISTER															
		R16	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
		0	0	0	0	0	0	0	0	C16	C15	C14	C13	C12	C11	C10	C9
		X4	X3	X2	X1	X4	X3	X2	X1	X1	X4	X3	X2	X1	X3	X2	X1
		X8	X7	X6	X5		X4	X3	X2	X5				X4	X7	X6	X5
							X8	X7	X6					X8			

Figure 4: Simplified CRC register contents after eight shifts using X_i.

4 Design Considerations

Several software implementations of CRCs have been reported in literature. In [9] an emphasis has been given on parallel implementation of the code. In [10], an attempt has been made to emulate the hardware process of CRC generation in software. A byte-wise algorithm has been proposed based on look-up table techniques in [10]. The hardware implementation is preferred for several reasons namely:

(1) bit-wise software implementations are very slow, their applicability is limited only to low encoding rates and they introduce a considerable amount of delay before delivering data.
(2) byte-wise or word-wise software implementations are normally based on table look-up methods. These algorithms have a large memory and considerable CPU time requirements.
(3) hardware implementation is fast, simple and easy to realize.

It can be concluded from the results of [9], [10] and [11] that a software implementation is not feasible for high speed information transfers. Thus, hardware implementation offers the necessary solution. A few hardware implementations of CRCs have also been reported in literature [1], [7]. In these implementations, an emphasis has been given on VLSI realization to exploit the high transmission bandwidth of the communication media. From the above discussion it is clear that a hardware implementation of byte-wise CRC generator which calculates CRC 'on the fly' is a novel idea and can prove to be extremely useful. In the next section, we present the implementation details of a byte-wise CRC generator realized in VLSI.

5 Implementation

The hardware description languages (Verilog, CDL, DDL, AHPL, VHDL, ISPS etc.) have been used successfully for documentation, communication and verification. They have also been used as input specification languages to design automation (DA) systems which synthesize VLSI layouts [3]. We have used UAHPL [6] (derived from AHPL), a RTL HDL for modeling the 16-bit byte-wise CRC generator. UAHPL facilitates structural specification of a design. The hardware compiler and functional simulator of UAHPL are used to perform logic synthesis. A wire/gate list generated is fed to a translator program to obtain a netlist. This netlist is used by VPNR, a layout sub-system of OASIS to generate the layout. OASIS is a cell-based silicon compiler that enables the design of semi-custom testable integrated circuits [2], [4], [8]. The UAHPL model of the CRC generator, the process of translating wirelist to netlist and the layout generation are discussed in detail in the following sections.

Any sequential circuit can be considered to be consisting of finite state machines (FSM). Our CRC generator is a 4-state FSM as shown in Figure 6. It has following characteristics:

```
MODULE:BCRCGENERATOR.
MEMORY:CREG{16};COUNT{3}.
BUSES: FBUS{8};F07;F17.
BUSES: MBUS{7};TEMP{3};WBUS{8}.
EXINPUTS:CLOCK;RESET;START.
EXINPUTS:MESIN{8}.
OUTPUTS: CRCRDY;MESOUT{8}.
CLUNITS : INC{3}
BODY SEQUENCE: CLOCK.
1     ⇒~(START)/(1)
      "goback to 1 if start is low"
2     MESOUT=MESIN;
      FBUS=MESIN@CREG{8:15};
      TEMP=CREG{0:1},CREG{7};
      F17=FBUS{1}@FBUS{2}@FBUS{3}@FBUS{4}@
      FBUS{5}@FBUS{6}@FBUS{7};
      F07 = FBUS{0}@ F17; WBUS = FBUS{0:7};
      MBUS = WBUS{0:6} @ WBUS{1:7};
      CREG{10:14}⇐CREG{2:6};
      CREG{15}⇐TEMP{2} @ F07;
      CREG{0:7}⇐F07,F17,MBUS{0:5};
      CREG{8}⇐CREG{0} @ MBUS{6};
      CREG{9}⇐CREG{1} @ FBUS{7};
      COUNT⇐INC(COUNT);
      ⇒~(& /COUNT)/(2).
      "goto 3 if all bits of count are high"
3     MESOUT = CREG{8:15};CRCRDY=\1\.
      "Least significant byte of CRC is ready"
4     MESOUT = CREG{0:7};CREG ⇐ 16$0;
      CRCRDY=\1\;⇒ (1).
      "Most significant byte of CRC is ready"
ENDSEQUENCE
      CONTROLRESET(RESET)/(1);
END.
```

Figure 5: UAHPL description of byte-wise CRC generator.

- based on internal Exclusive-Or (IE) type LFSR
- simple to design and easy to implement
- high speed and optimal hardware and memory space requirements
- enables efficient realization in VLSI

With each clock pulse the data comes in continuously in bytes, and a 16-bit CRC is generated *on the fly* which means that all the necessary calculations are done in a single clock pulse. The data moves out continuously through the MESOUT bus. When the whole data has moved out the final 16-bit result is transferred to the output in two clock pulses, 8-bits at a time. This way, any desired generator polynomial of degree-16 can be implemented.

6 Hardware Design and UAHPL Model

The UAHPL description of CRC generator is shown in Figure 5. The declaration of inputs, outputs, internal and external buses, memory elements/registers, combinational logic unit (CLU) and the clock precedes the actual body of the CRC generator. The code corresponding to the four states of the FSM is enclosed within **BODY SEQUENCE** and **END-SEQUENCE** keywords.

The initialization is done in the 1st STATE. In the 2nd STATE, a 64-bit message is supplied byte-wise, in each clock pulse, on the line MESIN. The message byte is processed in the same clock pulse by the data path logic and transmitted 'on the fly' on the output line MESOUT. The processing of each byte of the message is done as follows:

- The message byte (on the MESIN bus) is exclusive-ORed with the lower 8-bits of the CRC register, CREG, to get X_is.
- The 0th bit of the CREG gets the exclusive-OR of all X_is, i.e., X_1 to X_8.
- The 1st bit of the CREG gets the exclusive-OR of X_1 to X_7.
- This process is continued till the final contents of the CREG are as shown in Fig. 4.
- In the same state (i.e., 2nd STATE), the COUNT register, which stores the number of loops performed, is incremented. The contents of COUNT are checked to see if all the bits are high (by taking NAND of all the bits). If it is so, then we go to the next state; otherwise we go back to the 2nd STATE.
- The loop is executed 8 times since we have 8 bytes of the message. At the end of the last iteration, CREG contains the CRC generated for the 64-bit message.
- In the 3rd STATE, the lower byte of the CRC is appended at the end of the message (on MESOUT bus) and the signal CRCRDY becomes high to indicate this.
- In the 4th STATE, the higher byte of the CRC is transmitted on the MESOUT bus. CRCRDY is still high. The control is returned to the 1st STATE to perform the same task as described, for another message, if any.

The 16-bits CRC pattern generated is available in only the 9th clock pulse. The transmission of the first byte of the CRC can take place in 9th clock pulse and the next byte is transmitted in the 10th clock pulse. Thus, the number of clock cycles required to generate and transmit any CRC in our implementation, is just two more than the time required to calculate it.

The hardware of a small segment of the CRC circuit is given in Figure refcreg. Note that CREG{9} corresponds to bit R_7 which must get the exclusive-OR of C_{15} (CREG{1}) and X_1; where X_1 corresponds to the exclusive-OR of C_1 and M_1 (FBUS{7}). C_1 and M_1 being the least significant bits of CREG and MESIN, that is, CREG{15} and

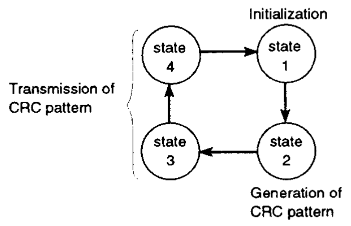


Figure 6: Four state FSM.

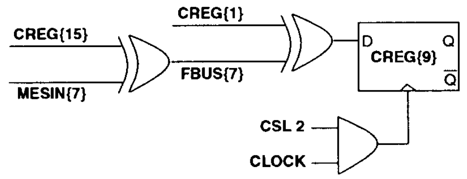


Figure 7: Hardware segment of the CRC generator constructed from its UAHPL model.

MESIN{7} respectively. Also note that the transfer into this register is enabled only during state 2, hence the clock is ANDed with the signal CSL2. [1]

7 Hardware of a small segment of the CRC generator.

Logic synthesis: A hardware compiler and a functional-level simulator were used to compile and simulate the UAHPL model of CRC generator respectively. After the UAHPL code is compiled by Stage-1 compiler, it is simulated at the RTL level. The Stage-2 compiler is used for logic synthesis. It generates a logic netlist of the hardware circuit.

Logic/Gate-level simulation: The **RNL** netlist was simulated at the logic/gate-level by performing **RNL** simulation and the results were observed on **SigView**, a simulation (signal) viewing program.

Translation of netlist to layout: The standard cell library of OASIS [8] consists of scalable **CMOS** cells compatible with 2μ SCMOS technology of MOSIS. The layout sub-system of **OASIS** called **V**anilla **P**lace a**N**d **R**oute (VPNR) generates the standard cell layout from the RNL netlist. VPNR uses a library of pre-designed standard cells to make the layout. It also incorporates testability by including scan path based testing circuitry in the layout.

[1] Note that in UAHPL, by convention the MSBit of a vector has zero index. For example, CREG{0} refers to R16.

8 Conclusion

In this paper we have presented the VLSI implementation of a byte-wise CRC generator written in UAHPL. The design and implementation details were discussed. Besides a small area, the chip generates CRC at an enhanced speed which is much faster than the bitwise implementation for the same purpose. It can be used in various data communication and data compression applications. Simulation has been carried out at various levels to verify the design process and ensure that the chip will work after fabrication. The chip consists of 27 I/O pins and is fully testable.

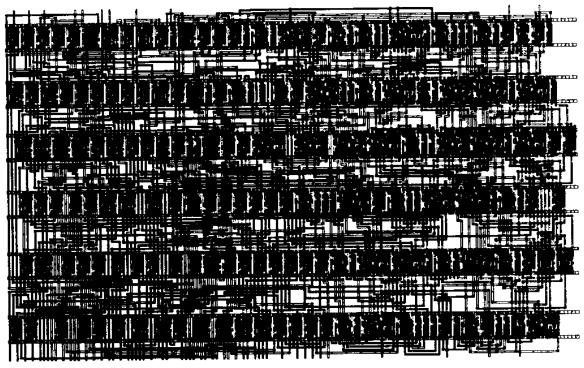


Figure 8: Layout of byte-wise CRC generator chip.

Acknowledgement

This work was done as part of a course project (VLSI Design Tools). Authors acknowledge King Fahd University of Petroleum and Minerals for support of this research.

References

[1] Guido Albertengo and Riccardo Sisto. Parallel CRC Generation. *IEEE Micro*, 10(5):63–71, October 1990.

[2] F. Brglez, D. Bryan, J. Calhoun, and R. Lisanke. Automated Synthesis for Testability. *IEEE Transaction on Industrial Electronics*, 36(2):263-277, May 1989.

[3] Fredrick J. Hill. AHPL: Then and Now. *IEEE Design and Test of Computers*, pages 73–75, June 1992.

[4] Gershon Kedem, Franc Brglez, and Krzysztof Kozminski. ASIC Design with OASIS. *Proceedings IEEE/ISCAS*, 4(4):2580–2583, 1990.

[5] R. Lee. Cyclic Code Redundancy. *Digital Design*, 11(7):75–85, July 1981.

[6] M. Masud and Sadiq M. Sait. Universal AHPL- a language for VLSI Design Automation. *IEEE Circuits and Devices Magazine*, September 1986.

[7] Amar Mukherjee, M. A. Bassiouni, and N. Ranganathan. Improving Bandwidth of Communication Controllers. *IEEE International Conference on Communications*, 3(3):1390–94, 1988.

[8] Open Architecture Silicon Implementation Software, MCNC.

[9] A. K. Pandeya and T. J. Cassa. Parallel CRC lets many lines use one circuit. *Computer Design*, 14(9):87–91, September 1975.

[10] Aram Perez, Wismer, and Becker. Byte-wise CRC Calcutations. *IEEE Micro*, 3(3):40–50, June 1983.

[11] T. V. Ramabadran and S. S. Gaitonde. A Tutorial on CRC Computations. *IEEE Micro*, 8(4):62–75, August 1988.

[12] Sadiq M. Sait and Shahid Tanvir Khan. VLSI Layout Generation of a Programmable CRC Chip. *IEEE Transactions on Consumer Electronics*, 39(4):911–916, November 1993.

Biography

Wasif Hasan obtained a Bachelor's degree in Electronics and Communications Engineering from Aligarh Muslim University (India) in 1993. He has been awarded a research/teaching assistantship by the Department of Computer Engineering and is currently doing his M.S in Computer Engineering from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran.