A PROPOSED CACHE LINE IMPLEMENTATION SOLUTION WITH ERROR/CORRECTING CAPABILITIES FOR MANAGING CACHE COHERENCY IN MULTIPROCESSOR SYSTEMS

Daniela Elena POPESCU
Department of Computers, Faculty of Electrical Engineering and Informatics Technology,
University of Oradea, no.1 University Str. 410087, Oradea, tel. 0040 723 268 428, e-mail: depopescu@uoradea.ro

ABSTRACT

The paper presents a solution for designing a cache coherency state machine for a multiprocessor system with error/correcting capabilities for the unidirectional errors and the used resources implied by a FPGA implementation of our solution. The cache coherency method used is based on the MESI protocol. The implementation is realized with FPGA by using the ALTERA Program.

Keywords: *cache memory, memory coherency, error detector*

1. CACHE COHERENCY

In contemporary multiprocessor systems, it is customary to have one or two levels of cache associated with each processor [1]. This organization is essential to achieve reasonable performance. It does, however, create a problem known as the *cache coherence* problem.

The essence of the problem is this: Multiple copies of the same data can exist in different caches simultaneously, and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.

There are two common write policies:

Write back: Write operations are usually made only to the cache. Main memory is only updated when the corresponding cache line is flushed from the cache.

Write through: All write operations are made to main memory as well as to the cache, ensuring that main memory is always valid.

Hardware schemes differ in a number of particulars, including where the state information about data lines is held, how that information is organized, where coherence is enforced, and the enforcement mechanisms [2].

2. ERROR CORRECTION

A semiconductor memory is subject to errors. These can be categorized as hard failures and soft failures.

Both hard and soft errors are clearly undesirable, and most modern main memory systems include logic for both detecting and correcting errors [2] [3]. Fig. 1 illustrates in general terms how the process is carried out.

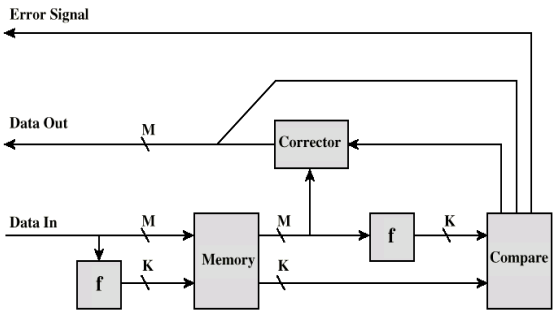

Fig. 1 Error-Correcting Code Function

When data are to be read from memory, a calculation is performed on the data to produce a code. Both the code and the data are stored. Thus, if an M-bit word of data is to be stored, and the code is of length K bits, then the actual size of the stored word is M+K bits. When the previously stored word is read out, the code is used to detect and possibly correct errors. A new set of K code bits is generated from the M data bits and compared with the fetched code bits. The comparison yields one of three results:

- No errors are detected. The sent data bits are sent out
- An error is detected, and it is possible to correct the error. The data bits plus error correction bits are fed into a corrector, which produces a corrected set of M bits to be sent out
- An error is detected, but it is not possible to correct it. This condition is reported

Codes that operate in this fashion are referred to as error-correcting codes [1] [4]. A code is characterized by the number of bit errors in a word that it can correct and detect.

To provide cache consistency on an SMP (Symmetric Multiprocessors), the data cache often supports a protocol known as MESI. For MESI, the data cache includes two status bits per tag, so that each line can be in one of four states:

- **Modified:** The line in the cache has been modified (different from main memory) and is available only in this cache.
- **Exclusive:** The line in the cache is the same as that in main memory and is not present in any other cache.
- **Shared:** The line in the cache is the same as that in main memory and may be present in another cache.
- **Invalid:** The line in the cache does not contain valid data.

Table 1 summarizes the meaning of the four states. Fig. 2 and Fig. 3 displays the state diagram for the MESI protocol. Each line of the cache has its own state bits and therefore its own realization of the state diagram. At any time a cache line is in a single state. If the next event is from the attached processor, then the transition is dictated

by Fig. 2 and if the next event is from the bus, the transition is dictated by Fig. 3 In [1] these transitions are presented in detail.

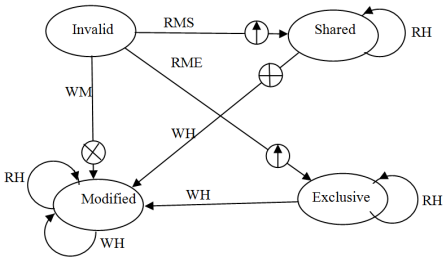

Fig. 2 Master Processor

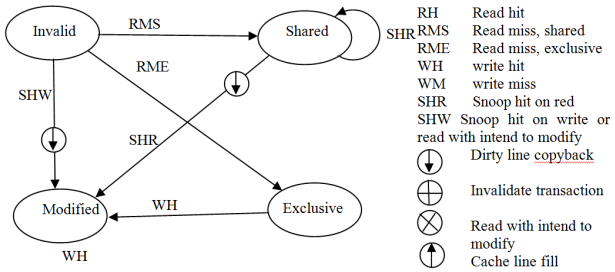

Fig. 3 Snooping Processor

Table 1 MESI Cache Line States

	M Modified	**E Exclusive**	**S Shared**	**I Invalid**
This cache line valid?	Yes	Yes	Yes	No
The memory copy is…	out of date	Valid	Valid	-
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line…	Does not go to bus	Does not go to bus	Goes to bus and updates cache	Goes directly to bus

3. THE BLUE-PRINT OF UNIDIRECTIONAL ERROR-DETECTING/CORRECTING CODE

The unidirectional error detecting/correcting codes are presented in [7].

Unidirectional errors arise as a result of common VLSI failure nodes and certain connection faults, particularly with serial data transmission. Thus many researchers have been motivated to study the properties of unidirectional errors and methods for their detection and correction in digital systems.

The theory of error-detecting and error-correcting codes plays an important role in the design of dependable and fault-tolerant digital systems. An error affecting multiple bits in a codeword is unidirectional if the bit inversions characterizing the error are all of the same type ($0 \to 1$ or $1 \to 0$).

In order to present our code format, we will consider the following:

b the number of bits in the word
m the number of words in the block
r the number of checking bits
k the total number of information bits, so that $k = m \cdot b$

In Fig. 4 we show the code format, where WC_i (i=1,m), represents the i^{-th} word of information. The last two words are the check words:

- parity check (PC), having the same length as the information word
- Arithmetical residual check word (ARC), having the length s less than b.

Let us consider x_{ij} the j bit of i information word.

The j bit of parity word is calculated as *xor* among the corresponding information bits: $y_j = x_{1j} \oplus x_{2j} \oplus ... \oplus x_{mj}$ where $j = 1, 2, \ldots, b$.

Arithmetical residual, ARC, is calculated as follows:

Considering p the smallest prime number, so that $(p-1)/2 \geq m$ and $p > b$, or $p > \max(2m, b)$, the ARC is calculated like this:

$$ARC = -\left(\sum_{i=1}^{m}\sum_{j=1}^{b} x_{ij} \cdot i\right) (\mathrm{mod}\, p) \equiv \equiv -\left(\sum_{i=1}^{m} N_i \cdot i\right) (\mathrm{mod}\, p) \quad (1)$$

where N_i is the number of 1 in the information word i.

The binary representation of ARC is $(b_{s-1} b_{s-2} \ldots b_0)$.

Let's suppose that for a data transmission, we receive:

$$X' = x'_{mb} \ldots x'_{m1} \ldots x'_{2b} \ldots x'_{21} x'_{1b} \ldots x'_{11} y'_b \ldots \ldots y'_1 b'_{s-1} \ldots b'_0 \quad (2)$$

At the receiver, we calculate two syndromes, S_1 and S_2, as follows:

$$S_1 = (e_b e_{b-1} \ldots e_1) \quad (3)$$

where $e_i = y'_i \oplus x'_{1i} \oplus x'_{2i} \oplus x'_{3i} \oplus \ldots \oplus x'_{mi}$, for $i = 1, 2 \ldots, b$, and

$$S_2 = \left(ARC' + \sum_{i=1}^{m} N'_i \cdot i\right) (\mathrm{mod}\, p) \quad (4)$$

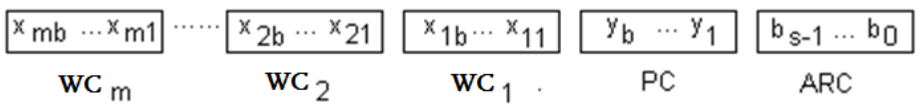

Fig. 4 The code format

Here, ARC' is the arithmetical residual check word received and $N_i^{'}$ is the number of 1 in the i word received. For the analysis of error-correction's capacity for this code, we will discuss the following situations:

Case 1 (no errors): When there are no errors on data, it's easy to verify that the syndromes $S_1 = 0$ and $S_2 = 0$.

Case 2 (the parity check word is wrong): In this case, the syndromes will be: $S_1 \neq 0$ and $S_2 = 0$. Error correction consist in recalculating PC from the data words, that are correct.

Case 3 (ARC is erroneous): For this case. We will have $S_1 = 0$ and $S_2 \neq 0$. The correct value of ARC can be obtained from the data words received. Notice that ARC value is between 0 and $p-1$; if the value received for ARC is greater or equal to p, then the word received for ARC is wrong.

Case 4 (one of the information words is wrong): Let's suppose that in the word j are t errors, where $1 \le t \le b$ and $1 \le j \le m$. For the $1 \rightarrow 0$ errors, the syndrome S_2 will be $S_2 = -t \cdot j (\text{mod } p)$ and for $0 \rightarrow 1$ errors, it will be $S_2 = +t \cdot j$.

In order to correct these errors [6], it will be necessary that:

1. Both syndromes $S_1 \neq 0$ and $S_2 \neq 0$ (to identify this situation among the others). This condition is satisfied.
2. In order to realize the error correction it is necessary that the values of syndromes for two different words are different – this condition is ensured by the imposed relations for m, n and p.

So, the second condition is requested for localizing the erroneous bits.

In conclusion, in S_1 we can find the position and the number of the affected bits by errors, and in order to localize the word with errors, we will use the ARC information.

For most systems the words' length is power of 2. In **Fig. 5**, **Fig. 6**, **Fig. 7** are represented the coding parameters for words with length 4, 8 and 16 bits. We choose as base for verification the largest prime number less then 2^s (so we maximized m).

3.1. Codifying data (the transmitter part)

Codifying consists in adding PC and ARC at the information words.

The i[th] bit for verifying the parity can be obtained by adding modulo 2 the b adjacent bits of useful information. $i = 1,...b$. Let *Nj* be the number of units in word *j*, for $j = 1, ..., m$. Then:

$$ARC = -\left(\sum_{j=1}^{m} N_j \cdot j \right) (\text{mod } p) \qquad (5)$$

That is the units number of each information word is numbered and multiply with the suitable j (the multiplication is made modulo p).

Using the equations presented above, the transmitter calculates PC and ARC, sending them to the receiver unit at the end of the information words block.

3.2. Decodifying data (the receiver part)

At the decodifying place, we must calculate the two syndromes we have already talked about. S_1 is obtained by adding – modulo 2- the b adjacent information bits with the parity bit. S_1 is stored in a parity register RP.

Using the information words received, we will calculate a new ARC, which will be added to the ARC received, obtaining S_2. The circuit that generates S_2 is the same with the one who generates ARC at transmission. S_2 can be stored in a mod p register, ARR (arithmetical residual register).

There is no error when both $S_1 = 0$ and $S_2 = 0$.

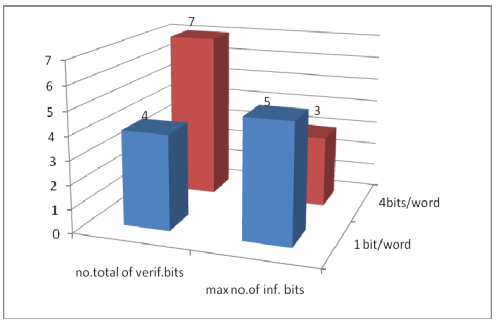

Fig. 5 The total no. of verification bits and the max no. of information bits for for 1 bit/word and 4 bit/word

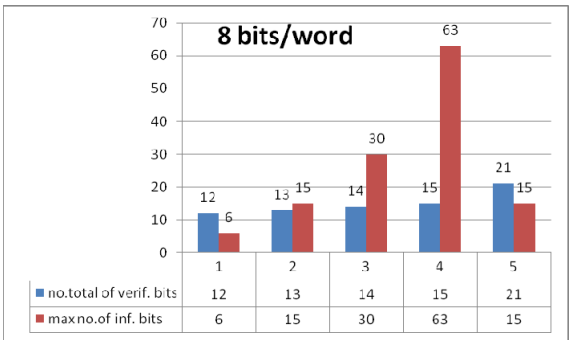

Fig. 6 The total no. of verification bits and the max no. of information bits for 8 bit/word

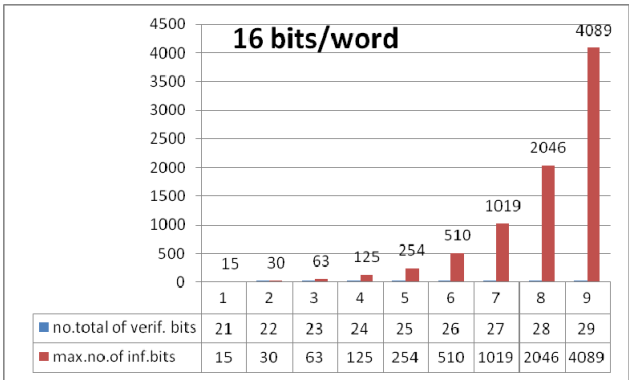

Fig. 7 The total no. of verification bits and the max no. of information bits for 16 bit/word

On the other hand, when $S_1 \neq 0$ and $S_2 \neq 0$, that means that an error occurred on the information data. We count t = the number of 1 bits from S_1 and calculate $S_2 / t = j(\mathrm{mod}\, p)$. If $j \leq m$, then j word is erroneous. Else the $(p-j)$ word is erroneous. When we know which word is affected by error, we will add it with S_1 and so the word will be corrected. Fig. 8 gives the structure for the entire system that ensures the cache coherency and the data transition between source and receptor in the case of using a code like the above depicted code.

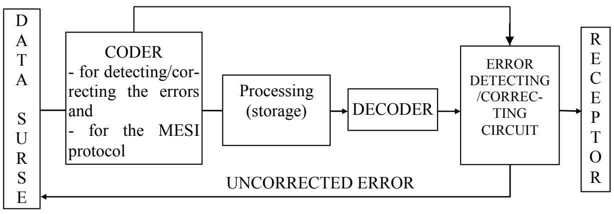

Fig. 8

4. THE FPGA IMPLEMENTATION FOR OUR SOLUTION

4.1. The AHDL coder's implementation

The implementation was realized with ALTERA and its AHDL (Altera Hardware Description. Language) software. In our implementation, we considered $b = 8$ bits of information, $r = 4$ check bits and $m = 6$ information words/block. We used the EPF10K30ETC144-1 chip, provided by ALTERA MAX+plus II [5].

The resources needed for the implementation are listed in Table 2.

4.2. The AHDL decoder implementation

We used the same EPF10K30ETC144-1 chip for the decodr also.

The resources needed for the implementation are listed in Table 2.

The automatons can be incorporated in the control units of the cache memories, offering them facilities of detection/correction unidirectional errors.

4.3. The AHDL implementation for the MESI finite state machine

We present separately the transactions for the master processor (Fig. 2) and for the snooping processor (Fig. 3).

For each transaction we have noted in the above figures the Input/Output that corresponds to that transaction; if there is no output generated, we have noted only the Input. When inputs and outputs have the following significance:

Inputs:

- RH – Read hit
- RMS – Read miss, shared
- RME – Read miss, exclusive
- WH – Write hit
- WM – Write miss
- SHR – Snoop hit on read
- SHW – Snoop hit on write or read with intent to modify

Outputs:

- Dirty line copyback
- Invalidate transaction
- Read with intent to modify
- Cache line fill

We have implemented in AHDL the script for the coherency state machine that implement the MESI protocol (Fig. 9) together with the coder and decoder; so we obtained an automata that control the cache coherency and also detect/correct the one-dimensional errors.

The final used report for our implementation is given in Table 2.

```
SUBDESIGN mesi
(  clk   : INPUT;
   reset : INPUT;
   i[2..0]  : INPUT;
   o[2..0] : OUTPUT;)
VARIABLE
ss: MACHINE WITH STATES (invalid, shared, modified, exclusive);
BEGIN
ss.clk = clk;
ss.reset = reset;

TABLE
   %     input     this_state          next_state        output  %
         i[2..0],     ss          =>   ss,               o[2..0];

         B"010",  invalid      =>   shared,          B"100";
         B"011",  invalid      =>   exclusive,       B"100";
         B"101",  invalid      =>   modified,        B"011";
         B"001",  shared       =>   shared,          B"000";
         B"100",  shared       =>   modified,        B"010";
         B"001",  modified     =>   modified,        B"000";
         B"100",  modified     =>   modified,        B"000";
         B"001",  exclusive    =>   exclusive,       B"000";
         B"100",  exclusive    =>   modified,        B"000";
         B"110",  shared       =>   shared,          B"000";
         B"111",  shared       =>   invalid,         B"000";
         B"111",  modified     =>   invalid,         B"001";
         B"110",  modified     =>   shared,          B"001";
         B"110",  exclusive    =>   shared,          B"000";
         B"111",  exclusive    =>   invalid,         B"000";
END TABLE;
END;
```

Fig. 9 The AHDL script for the state machine that implement the MESI protocol

In case of asymmetric errors in system can arise a single type of errors; let take them as $0 \rightarrow 1$. The type of errors is apriori known. The depicted code can be used also for correcting such errors. In this case, if the base of verification is a prime number, the number of information words can be less of equal with p-1. The coding and decoding algorithms and circuits can be modified for being applied to asymmetric codes.

It is demonstrated that any correcting code for correcting asymmetric errors needs at least $\log_2\left(m\left(2^b-1\right)=1\right)$ verification bits. So, for this code, it will be necessary at least $\log_2\left((p-1)\cdot\left(2^b-1\right)+1\right)=b+\log_2(p)$.

In [7] is given a design procedure for double error correcting codes for bytes, and also the inferior limit for number of verification bits.

Table 2 The final used resource report

	CODIF	DECODIF	MESI machine
Total dedicated input pins used	6/6 (100%)	6/6 (100%)	1/4 (25%)
Total I/O pins used	18/96 (18%)	10/96 (10%)	7/32 (21%)
Total logic cells used	107/1728 (6%)	193/1728 (11%)	5/32 (15%)
Total embedded cells used	0/96 (0%)	8/96 (8%)	0/32 (0%)
Total EABs used	0/6 (0%)	1/6 (16%)	5/32 (15%)
Average fan-in	3.31/4 (82%)	3.33/4 (83%)	2/32 (6%)
Total fan-in	355/6912 (5%)	643/6912 (9%)	5.60
Total logic cells required	107	193	5
Total flipflops required	26	51	2

5. CONCLUSIONS

The implementation (Fig. 8) (Fig. 9) presented in this paper is based on the idea of implementing at the cache memory line level of both the MESI cache coherency state machine with the error correcting facility based on an unidirectional error correcting code that is also presented in the paper. I realized an AHDL implementation of the proposed solution and I summarized the FPGA used resources implied by the AHDL implementation for coder, decoder/corrector and coherency automatons which are used to detect/correct the unidirectional errors in data transmissions for the MESI finite state machine (Table 2).

Although the coder and the decoder/corrector are presented as being separate in this paper, they are both present in every unit that ensure a bidirectional transmission

The efficiency of our implementation is based on the fact that the error-correcting code is suitable to be used for controlling the correctness of data in a line, because the line contains small amount of data.

REFERENCES

[1] Stalings, W., "Computer Organization land Architecture", Prentice Hall Sixth Edition, International Editions, ISBN 0-13-049307, 2003

[2] Hennesy, J., and Patterson, D., "Computer Architecture: A quantitative approach.", San Mateo, CA: Morgan Kaufmann, 1996

[3] Patterson D., and Hannesy, J., "Computer Organization and Design: The hardware/Software Interface". San Mateo, CA: Kaufmann, 1998

[4] Badiu, M., Multiprocesoare simetrice: coerența cache-urilor, PC-Report June 1998, Seria: arhitectura modernă a calculatoarelor, pp.34.-45

[5] ALTERA documentation, http://www.altera.com/support/kdb/kdb-index.jsp

[6] Abramovici, M., Breuer, M.A., Friedman, A.D., Digital Systems and Testable Design, Computer Science Press, 1996

[7] Bose, B., Al-Bassam, S. Byte Unidirectional Error Correcting and Detecting Codes, IEEE Trans.Comp., vol.41, no.12, Dec.1992s

Received May 5, 2009, accepted September 3, 2009

BIOGRAPHY

Daniela E. Popescu was born in Oradea, Romania, in 1961. She received the PhD degree in Computer Science from the University of Timisoara, in 1998.
She is an Professor and head of Department of Computer Science of University of Oradea, where she is member of the Computer Architecture and Computer testing research group. She has written more than 60 papers in international journals. Her current main research field interest is in digital testing, computer architecture and digital circuits design. In addition, he also investigates questions related with the diagnostics of complex systems.