FUNCTIONAL TESTING OF A MICROPROCESSOR THROUGH LINEAR CHECKING METHOD

Prof. Dr. Pervez Akhtar
National University of Science & Technology, Karachi Campus, Pakistan
pervez@pnec.edu.pk

Prof. Dr. M. Altaf Mukati
Hamdard Institute of Information Technology, Hamdard University, Karachi, Pakistan
altafmukati@gmail.com

ABSTRACT
The gate-level testing also called low-level testing is generally appropriate at the design time and for small circuits. The chip-level testing and board-level testing also called high-level testing are preferred when the circuit complexities are too high, making it difficult to perform low level testing in a reasonable amount of time. The cost of low-level testing is also generally very high. Such high costs and time are only justified when some design-changes are required. In this paper, a high level quick checking method, known as Linear Checking Method, is presented which can be used to qualify the functionality of a Microprocessor. This can also be used to check hard faults in Memory chips.

Keywords: Microprocessors, ALU, Control Unit, Instructions.

1 INTRODUCTION
Due to the advances in the integrated circuit technology, more and more components are being fabricated into a tiny chip. Since the number of pins on each chip is limited by the physical size of the chip, the problem of testing becomes more difficult than ever. This problem is aggravated by the fact that, in nearly all cases, integrated circuit manufacturers do not release the detailed circuit diagram of the chip to the users [1].

The users are generally more interested to know about the chip, whether it is functionality working and relied upon? if not, the whole chip is replaced with a newer one. This is contrast to the gate-level testing of a digital circuit, which is used to diagnose faulty gates in the given circuit, in case of failing. The idea of using functional testing is also augmented by the fact that in case of any functional failure, caused due to any fault in the chip, the user cannot repair the chip. Hence the users have only two choices: either to continue using the chip with a particular failing function, knowing that the failing function will not be used in the given application or to replace the whole chip.

The functional modeling is done at a higher level of abstraction than a gate-level modeling. This in fact exists between the Gate-level modeling and the Behavioral modeling, which is the highest level of abstraction [2]. The functional fault modeling should imitate the physical defects that cause change in the function or behavior, for example; the function of a synchronous binary up-counter is to advance one stage higher in binary value when clock hits it. A physical defect, which alters this function, can be modeled in terms of its effect on the function. Such defect-findings are extremely important at the design time or if the design changes are required at a later stage.

What, if a microprocessor does not produce the correct results of any single or more functions? From the user’s perspective, it is enough to know which function is failing, but from designer’s perspective, the cause of failing is also important to know, so that the design changes may be carried out, if necessary. This is certainly a time-taking process. For example, the gate level simulation of the Intel 8085 microprocessor took 400 hours of CPU-time and only provided 70% fault coverage [3].

High level functional verification for the complex Systems-On-Chip (SOCs) and microprocessors has become a key challenge. Functional verification and Automatic Test Pattern Generator (ATPG) is one synergetic area that has evolved significantly in recent years due to the blossoming of a wide array of test and verification techniques. This area will continue to be a key focus of future Microprocessor TEST and Verification (MTV) [4].
Functional failures can be caused due to single or multiple stuck-at faults in any of its functional block. The functional-testing, which refers to the selection of tests that verify the functional operation of a device, is one of an efficient method to deal with the faults existing in a processor. Functional testing can also be carried out at a smaller level, for example, a functional test of a flip-flop might be to verify whether it can be set or reset? and further, can it hold the state or not? Similarly, the other MSI chips such as Multiplexers, Encoders, Decoders, Counters, Hardwired-Multipliers, Binary-Adders & Subtractors, Comparators, Parity Checkers, Registers and other similar circuits can also be verified for their required functionalities.

Some designers and manufacturers provide built-in self-test (BIST) these days that generate the test on the chip and responses are checked within the chip itself. However, the widespread use of such testability techniques is hampered by a lack of tools to support the designer and by the additional cost in chip area as well as the degradation in performance [5]. For example, the Intel 80386 microprocessor employs about 1.8% area overhead for BIST to test portions of the circuit [6].

The ever increasing complexity combined with the advanced technology used in the design of the modern microprocessors, has lead to two major problems in producing cost-effective, high quality chips:

1. Verification: This is related to validate the correctness of the complex design. Simulation is the primary means of design validation used today. In the case of processor design validation, the sequences are either written manually or generated automatically by a random sequence generator [7].

2. Testing: This is related to check the manufactured chips for realistic defects. A variety of test generation and design-for-testability (DFT) techniques is used to ensure that the manufactured chips are defect-free.

Both design verification and testing depend, therefore, on test sequences used to expose either the design faults or manufacturing defects. It has also been found that manufacturing test pattern generation can be used for design verification [8] and that design verification techniques can be used to find better manufacturing tests [9]. However, to find the effective test patterns for either of the said purposes is not simple, due to high complexities of microprocessors. Hence the only effective method left is to develop the functional tests. Considerable work has been done in the field of microprocessor functional testing. One of such work, known as ‘Linear Checking Method’ is presented in this paper.

Before performing functional testing, functional description of the chip must be known. In case of microprocessor, this can be obtained through its instruction set. The two most important functional blocks of any microprocessor are the CU (Control Unit) and the ALU (Arithmetic Logic Unit). All the instructions, at low-level, are composed of Op-Codes and operands. An op-code, also called the Macro-instruction, goes to the CU, which decodes each macro-instruction into a unique set of micro-instructions. The operands go to the ALU, which processes it according the tasks defined within the micro-instructions. In between these functional blocks, there exists several registers for the temporary storage of op-codes, decoded-instructions and operands.

The fault may occur at various places in the processors, causing it to function incorrectly. Some of the common faults are: Register Decoding Fault, Micro-Operation Decoding Fault (caused may be due to internal defect to the CU), Data Storage Fault (caused may be due to Stuck-at Fault or Pattern Sensitive Fault in the memory inside the Microprocessor), Data Transfer Fault (caused may be due to Stuck-at Fault or Bridging Fault on the busses connecting the various functional blocks of a Microprocessor) or ALU Fault (caused due to internal defect to the ALU). In each case, the given microprocessor results in producing incorrect function/ functions.

In the subsequent sections, first the functional verification has been described in general and then the Linear Checking Method has been presented through several examples. Based on the results obtained, the conclusion has been drawn and the further work has been proposed.

2 FUNCTIONAL VERIFICATION

The micro-instructions from the CU and the operands of an instruction are sent to the ALU simultaneously. The ALU then carries out the intended task or function. This can be shown with the help of a block diagram, as in Fig. 1.

![Figure 1: Functional testing](image-url)

The typical instructions are ADD, SUB, MUL,
SHL, SHR, ROTL, ROTR, INC, DEC, COMPL, AND, OR, XOR and many others.

3 LINEAR CHECKING METHOD

This method can be used to test and verify, not only the functionality of a microprocessor (more specifically ALU), but the memories as well. Linear checking is based on computing the value of ‘K’ using the equation 3.1:

\[ K = f_i(x, y) + f_i(\overline{x}, y) + f_i(\overline{x}, \overline{y}) + f_i(x, \overline{y}) \]  

Equation 1 is called the ‘basic equation’. The variables \( x \) and \( y \) are the operands, ‘\( i \)’ is the instruction. The value of \( K \) does not depend on the values of \( x \) and \( y \), but only depends on the instruction and on the size of operands (number of bits in the operands). It means the value of \( K \) is unique for every instruction. The chances are very little that the two instructions may have the same constant value of \( K \). An 8 and 16-bit ALUs have different values of \( K \), for the same instruction. Hence, in this method, \( K \) is used as a reference value to verify the functionality of an individual instruction.

3.1 Examples of functional verifications

Consider a 4-bit ALU. The value of ‘\( K \)’ can be computed as follows:

Suppose the instruction is \( \text{ADD}(x, y) = x + y \)
Here, \( n = 4 \). Let \( x = 5 \) (0101) and \( y = 3 \) (0011)
Therefore \( x = 1010 \) and \( y = 1100 \)

The value of \( K \) can be obtained from Equation 1, as follows:

\[ \text{ADD}(5,3)+\text{ADD}(5,12)+\text{ADD}(10,3)+\text{ADD}(10,12) = K \]
\[ 8 + 17 + 13 + 22 = 60 \]

Hence, for a 4-bit ALU, the \( \text{ADD} \) instruction will always be tested with respect to its reference value of 60, regardless what values of \( x \) and \( y \) are taken, i.e. instead of 5 and 3 as in the above example, now these values are taken as 9 and 10 respectively. Still the value of \( K \) remains the same, as proved below:

i.e. for \( x = 9 \) (1001) and \( y = 10 \) (1010)
\[ x = 6 \) (0110) and \( y = 5 \) (0101)

\[ \text{ADD}(9,10)+\text{ADD}(9,5)+\text{ADD}(10,6)+\text{ADD}(6,5)= 60 \]

The generalized formula can also be developed to find the value of \( K \) for the \( \text{ADD} \) instruction, for any size of ALU, as follows:

\[ K_n = 4(2^n - 1) \]

Where, the subscript with \( K \) represents the function. Hence from the generalized form, we obtain the same value of \( K \) i.e. if \( n = 4 \), then \( K_4(4) = 4(15) = 60 \).

Similarly, the value of \( K \) for any instruction can be obtained, provided its functional description is known. The value of \( K \), for the various other frequently used instructions, can be obtained similarly, as follows:

3.1.1 Multiply instruction \( f(x, y) = X \times Y \)
\[ f_i(x,y) = \text{MPY}(x,y) = X \times Y \]

Hence, from equation 3.1, the value of \( K \) can be obtained as follows:

\[ \text{MPY}(12,10)+\text{MPY}(10,3)+\text{MPY}(5,12)+\text{MPY}(5,3) \]
\[ 120 + 30 + 60 + 15 = 225 \]

Generalized form \( \forall K = (2^n - 1)^2 \)

3.1.2 Transfer instruction \( f(x) = x \)
This is a single valued function, hence only one variable is taken in computation of \( K \), thus ‘\( y \)’ is ignored in equation 1.

Thus \( f_i(x) = x \)
\[ = x + x + \overline{x} + \overline{x} = 10 + 10 + 5 + 5 = 30 \]

Generalized form \( \forall K = 2(2^n - 1) \)

3.1.3 Shift-Right instruction \( f(x) = \text{SHR}(x) \)
It is also a single valued function. With \( x = 1010 \):

(a) \( f_i(x, y) \) & \( f_i(\overline{x}, y) \) reduce to \( f_i(x) \) and

(b) \( f_i(\overline{x}, y) \) & \( f_i(\overline{x}, \overline{y}) \) reduce to \( f_i(\overline{x}) \)

Now \( f_i(x) \) represents the value of \( x \), after \( \text{SHR} \) operation i.e. 1010 \( \overline{0} \) 0101 and \( f_i(\overline{x}) \) represents the value of \( \overline{x} \) after \( \text{SHR} \) operation i.e. 0101 \( \overline{0} \) 0010.

Hence, \( K = 0101 + 0101 + 0010 + 0010 \)
\[ = 5 + 5 + 2 + 2 = 14 \]

Generalized form \( \forall K = 2(2^{n-1} - 1) \)

3.1.4 Shift-Left instruction \( f(x) = \text{SHL}(x) \)
With the same explanation as in section 3.1.3, the equation 1 becomes:

\[ K = f_i(x) + f_i(x) + f_i(\overline{x}) + f_i(\overline{x}) \]
Hence, with $x = 1010 \in \mathbb{F}$ and $\bar{x} = 0101 \in \mathbb{F}$, we find $f(\bar{x}) = 1010$

Therefore, $K = 4 + 4 + 10 + 10 = 28$

Generalized form $\forall E K = 2^{(2^n - 2)}$

#### 3.1.5 Logical-OR instruction ($f(x) = x \lor y$)

$K = f_i(x, y) + f_i(x, \bar{y}) + f_i(\bar{x}, y) + f_i(\bar{x}, \bar{y})$

$= x \lor y + (x \lor \bar{y}) + (\bar{x} \lor y) + (\bar{x} \lor \bar{y})$

$= 1110 + 1011 + 1101 + 0111$

$= 14 + 11 + 13 + 7 = 45$

Generalized form $\forall E K = 3(2^n - 1)$

#### 3.1.6 Logical-AND instruction ($f(x) = x \land y$)

$K = f_i(x, y) + f_i(x, \bar{y}) + f_i(\bar{x}, y) + f_i(\bar{x}, \bar{y})$

$= (x \land y) + (x \land \bar{y}) + (\bar{x} \land y) + (\bar{x} \land \bar{y})$

$= 1000 + 0010 + 0100 + 0001$

$= 8 + 2 + 4 + 1 = 15$

Generalized form $\forall E K = 2^n - 1$

#### 3.1.7 Logical-XOR instruction ($f(x) = x \oplus y$)

$K = f_i(x, y) + f_i(x, \bar{y}) + f_i(\bar{x}, y) + f_i(\bar{x}, \bar{y})$

$= (x \oplus y) + (x \oplus \bar{y}) + (\bar{x} \oplus y) + (\bar{x} \oplus \bar{y})$

$= 0110 + 1001 + 1101 + 0110$

$= 6 + 9 + 9 + 6 = 30$

Generalized form $\forall E K = 2^{(2^n - 1)}$

#### 3.1.8 Increment instruction ($f(x) = INC(x)$)

This is also a single valued function:

$K = f_i(x) + f_i(\bar{x}) + f_i(\bar{x}) + f_i(\bar{x})$

$= 1011 + 1011 + 1011 + 1011$

$= 11 + 11 + 6 + 6 = 34$

Generalized form $\forall E K = 2^{(2^n + 1)}$

#### 3.1.9 Decrement instruction ($f(x) = DEC(x)$)

$K = f_i(x) + f_i(x) + f_i(\bar{x}) + f_i(\bar{x})$

$= 1001 + 1001 + 0100 + 0100$

$= 9 + 9 + 4 + 4 = 26$

Generalized form $\forall E K = 2^{(2^n - 3)}$

#### 3.1.10 Complement instruction ($f(x) = \overline{x}$)

$K = f_i(x) + f_i(x) + f_i(x) + f_i(x)$

$= 0101 + 0101 + 1010 + 1010 = 30$

Generalized form $\forall E K = 2^{(2^n - 1)}$

#### 3.1.11 2’s comp. instruction ($f(x) = \bar{x} + 1$)

$K = f_i(x) + f_i(x) + f_i(\bar{x}) + f_i(\bar{x})$

$= 0110 + 0110 + 1101 + 1101 = 34$

Generalized form $\forall E K = 2^{(2^n + 1)}$

#### 3.2 Memory error correction

Linear checks can also be used to verify memories. For example, let the multiplication $x*y$ function is stored in the memory. Let the operands are 4-bit length, with $x = 1010$ and $y = 0010$, it means $x = 0101$ and $y = 1101$. Hence, all the four components of equation 1 are computed and the results are stored in memory, as shown in Table 1.

<table>
<thead>
<tr>
<th>Table 1: linear checks on memories</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x,y)$</td>
</tr>
<tr>
<td>$(x,\bar{y})$</td>
</tr>
<tr>
<td>$(\bar{x},y)$</td>
</tr>
<tr>
<td>$(\bar{x},\bar{y})$</td>
</tr>
<tr>
<td>$K$</td>
</tr>
</tbody>
</table>

If the sum of four components is not equal to the value of $K$, then there must be some fault existing in the memory. Similarly, any of the preceding functions can be used to verify the memories. The testing can be done more accurately if the contents of $f(x,y)$ are stored at address $(x,y)$. In the above example, the contents can be stored on the corresponding addresses as shown in Table 2. If addition of the contents does not come equal to the value of $K$, then it will indicate some fault in the memory. Here, the location of fault is also obtained.

<table>
<thead>
<tr>
<th>Table 2: address versus contents in memory testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>1010</td>
</tr>
<tr>
<td>1010</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0101</td>
</tr>
</tbody>
</table>
4 RESULTS

All the computations done in the previous section are summarized in tables 3 & 4 for n = 4 & 8 respectively:

Table 3: Values tabulated through linear checking method for n = 4

<table>
<thead>
<tr>
<th>Instruction i</th>
<th>( f_i(x, y) )</th>
<th>( K_i(n) )</th>
<th>( K_i(4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transfer</td>
<td>( x )</td>
<td>2((2^n-1))</td>
<td>30</td>
</tr>
<tr>
<td>Add</td>
<td>( x + y )</td>
<td>4((2^n-1))</td>
<td>60</td>
</tr>
<tr>
<td>Multiply</td>
<td>( x * y )</td>
<td>((2^n-1)^2)</td>
<td>225</td>
</tr>
<tr>
<td>Subtract</td>
<td>( x - y )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Logical OR</td>
<td>( x \lor y )</td>
<td>3((2^n-1))</td>
<td>45</td>
</tr>
<tr>
<td>Logical AND</td>
<td>( x \land y )</td>
<td>(2^n-1)</td>
<td>15</td>
</tr>
<tr>
<td>Logical XOR</td>
<td>( x \xor y )</td>
<td>2((2^n-1))</td>
<td>30</td>
</tr>
<tr>
<td>Complement</td>
<td>( \overline{x} )</td>
<td>2((2^n-1))</td>
<td>30</td>
</tr>
<tr>
<td>2’s comp.</td>
<td>( \overline{x} + 1 )</td>
<td>2((2^n+1))</td>
<td>34</td>
</tr>
<tr>
<td>Increment</td>
<td>( x + 1 )</td>
<td>2((2^n+1))</td>
<td>34</td>
</tr>
<tr>
<td>Decrement</td>
<td>( x - 1 )</td>
<td>2((2^n-3))</td>
<td>26</td>
</tr>
<tr>
<td>Shift-Left</td>
<td>( (x_2,x_3,...x_n,0) )</td>
<td>2((2^n-2))</td>
<td>28</td>
</tr>
<tr>
<td>Shift-Right</td>
<td>( (0,x_2,x_3,...x_n,1) )</td>
<td>2((2^n-1))</td>
<td>14</td>
</tr>
<tr>
<td>Rotate-Left</td>
<td>( (x_2,x_3,...x_n,x_1) )</td>
<td>2((2^n-1))</td>
<td>30</td>
</tr>
<tr>
<td>Rotate-Right</td>
<td>( (x_n,x_2,...x_{n-1}) )</td>
<td>2((2^n-1))</td>
<td>30</td>
</tr>
</tbody>
</table>

Table 4: Values tabulated through linear checking method for n = 8

<table>
<thead>
<tr>
<th>Instruction i</th>
<th>( f_i(x, y) )</th>
<th>( K_i(n) )</th>
<th>( K_i(8) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transfer</td>
<td>( x )</td>
<td>2((2^n-1))</td>
<td>510</td>
</tr>
</tbody>
</table>

5 CONCLUSION

It is concluded that the value of K can be obtained for any given instruction. The ‘CLR’ (clear) instruction is a special one, since it does not have any operand; all the four components of the equation 3.1 are taken as 0. Note that almost all the values obtained for K are unique, except for ‘Transfer’, ‘Complement’, ‘Logical XOR’ and ‘Rotate-Left/Right’ instructions, which means if the instructions are transformed due to any fault in the CU (or in any associated circuit) then these particular functional failures cannot be distinguished but the processor as a whole can be declared to have contained a fault. The last column in tables 3 & 4 can be obtained directly from the generalized forms. This column is stored in the memory along with the relevant function.

6 FUTURE WORK

Further research is proposed on the given method, especially in the case when the ‘Reference Value (K)’ of two or more functions is obtained same i.e. to distinguish or identify an individual failing function, in case when their reference values happen to be the same, as mentioned under the conclusion.
7 REFERENCES


