Proc Design Automation Conf.
pp 196-202, 1984

A Practical Approach to Testing Microprocessors*

M. G. Karpovsky
College of Engineering
Boston University
110 Cummington St.
Boston, MA
USA

R.G. Van Meter
Dept. of Mathematical Sciences
State University College
Oneonta, NY
USA

*This work was supported by the International Business Machines Corporation.

A PRACTICAL APPROACH TO TESTING MICROPROCESSORS *

Abstract. In this paper, we describe functional testing techniques for detecting single stuck-at faults in a microprocessor. These techniques are practical in that a relatively small number of machine language instructions is needed in the programs which implement them, the number of reference outputs which must be stored is small, and hardware redundancy for testing purposes is not needed. The efficacy of use of these functional testing techniques has been demonstrated by applying them to the testing of a simulated 4-bit microprocessor with simulated single stuck-at faults.

Index terms. Single stuck-at faults, functional testing of a microprocessor, linear checks, fault coverage, simulation.

1. INTRODUCTION

In testing a microprocessor for single stuck-at faults, we use functional testing [1], that is, we employ testing techniques which require only a description of the instruction set of the microprocessor (the syntax of each instruction, the result of its execution, the busses and registers exercised, etc.) rather than a gate-level description of the microprocessor.

Because of the ever increasing use of microprocessors, microprocessor testing is of obvious importance. With the increasing density of gates in VLSI devices, gate-level testing is becoming less feasible. In addition, the gate-level description of a microprocessor is generally unavailable to the user. The use of random testing [2]-[4] may require prohibitively much time in order to get reasonable fault coverage. Functional testing seems

* This work was supported by the International Business Machines Corporation

to be a viable alternative to gate-level testing and random testing.

The testing procedure involves the execution of a program in the machine language of the microprocessor with certain data. We assume a tester can enter the instructions of this program and the data at appropriate times via the input pins of the microprocessor, monitor the output pins, and compare observed outputs with expected (or "reference") outputs. Hardware redundancy for testing purposes is not needed.

In subsequent sections, we discuss a model of the microprocessor to be tested, the testing techniques utilized (Linear Checks Method, Fixed Input Data Method, Parity Circuitry Testing, Subroutine Testing, and Interrupt Testing), and a Case Study.

2. MICROPROCESSOR MODEL

We assume the microprocessor to be tested has the following components:

- an n-bit Arithmetic/Logic Unit;
- an n-bit bus for internal data flow;
- n-bit general-purpose registers;
- n-bit data-address registers;
- ALU Status Latches (CARRY, ZERO, NOT ZERO, etc.);
- an Instruction Register;
- a Memory Data Bus In with a Parity Checker;
- a Memory Data Bus Out;
- a Program Counter;
- a Memory Address Register;
- a Memory Address Bus;
- an I/O Data Bus In with a Parity Checker;
- an I/O Data Bus Out with a Parity Generator;
- an I/O Address Bus;
- a Link Register for saving an address for return from a subroutine or interrupt;

- Backup Link Registers for saving addresses for returns from nested subroutines;
- an Interrupt Register for saving the contents of the ALU Status Latches and, possibly, paging bits and other information during an interrupt;
- a control unit;
- an off-chip main memory;
- off-chip general-purpose registers;
- off-chip data-address registers.

If the architecture is different than that assumed, the testing procedures described below can be used with slight modification.

3. LINEAR CHECKS METHOD

For testing the ALU, the busses and lines leading to and from the ALU, and the general-purpose and data-address registers, we use the "Linear Checks Method" [5]-[9] which is a data-compression scheme for reducing the number of reference outputs and, hence, storage requirements.

This method is based on the following property: for almost all Arithmetic/Logic instructions, the sum of the results of executing the instruction with the four pairs of n-bit operands (x,y), $(x,\bar{y})$, $(\bar{x}, y)$, and $(\bar{x}, \bar{y})$ (where "$\bar{x}$" denotes the componentwise complement of x) is a constant with respect to x and y and depends only on n and the instruction in the fault-free case.

In the usual n-bit logic, $f(x,y)$ is an n-bit binary vector; however, for n-bit arithmetic we need at least n+1 bits for $f(x,y)$ (at least a CARRY bit is needed).

If we view the Arithmetic/Logic instructions as functions of two variables over the set of all binary n-vectors with values which are binary m-vectors, where $m \geq n$, then the above property may be stated more formally as follows (let "AL" denote the set of all Arithmetic/Logic instructions of

the microprocessor): for almost all f in AL,

$$(3.1)\quad \begin{cases} \text{there exists an integer constant } K_{n,f} \text{ such that for} \\ \text{all pairs } (x,y) \text{ of binary } n\text{-vectors,} \\ \quad f(x,y) + f(x,\bar{y}) + f(\bar{x},y) + f(\bar{x},\bar{y}) = K_{n,f}. \end{cases}$$

Table 1 gives $K_{n,f}$ for any positive integer n and for n = 4,8, and 16 for some typical Arithmetic/Logic instructions. The n-bit operands are viewed as binary representations of positive integers; however, for convenience, the constants $K_{n,f}$ are given in decimal notation. The proofs of the general results in Table 1 hinge on the fact that

$$x + \bar{x} = \underbrace{11 \cdots 1}_{n} = 2^n - 1.$$

To illustrate, we consider the 4-bit operands (x =) 0101 and (y =) 0011, for which the complements are ($\bar{x}$ =) 1010 and ($\bar{y}$ =) 1100. For the addition instruction (ADD), we have

$$\begin{aligned} K_{4,ADD} &= ADD(x,y) + ADD(x,\bar{y}) + ADD(\bar{x},y) + ADD(\bar{x},\bar{y}) \\ &= (x + y) + (x + \bar{y}) + (\bar{x} + y) + (\bar{x} + \bar{y}) \\ &= (5 + 3) + (5 + 12) + (10 + 3) + (10 + 12) \\ &= 8 + 17 + 13 + 22 \\ &= 60 \; (= 4 \cdot (2^4 - 1)). \end{aligned}$$

For the multiplication instruction (MPY), we have

$$\begin{aligned} K_{4,MPY} &= MPY(x,y) + MPY(x,\bar{y}) + MPY(\bar{x},y) + MPY(\bar{x},\bar{y}) \\ &= x \cdot y + x \cdot \bar{y} + \bar{x} \cdot y + \bar{x} \cdot \bar{y} \\ &= 5 \cdot 3 + 5 \cdot 12 + 10 \cdot 3 + 10 \cdot 12 \\ &= 15 + 60 + 30 + 120 \\ &= 225 \; (= (2^4 - 1)^2). \end{aligned}$$

No.	Instruction (f)	f(x,y)	$K_{n,f}$	$K_{4,f}$	$K_{8,f}$	$K_{16,f}$
1	Clear (CLR)	0	0	0	0	0
2	Transfer (TRN)	x	$2(2^n-1)$	30	510	131070
3	Complement (CMP)	$\bar{x}$	$2(2^n-1)$	30	510	131070
4	Increase (INCR)	$x + 1$	$2(2^n+1)$	34	514	131074
5	Decrease (DECR)	$x - 1$	$2(2^n-3)$	26	506	131066
6	Two's Complement (CMP2)	$\bar{x} + 1$	$2(2^n+1)$	34	514	131074
7	Shift Left (SHL)	$(x_2, x_3, \cdots, x_n, 0)$	$2(2^n-2)$	28	508	131068
8	Shift right (SHR)	$(0, x_1, x_2, \cdots, x_{n-1})$	$2(2^{n-1}-1)$	14	254	65534
9	Rotate left (ROTL)	$(x_2, x_3, \cdots, x_n, x_1)$	$2(2^n-1)$	30	510	131070
10	Rotate right (ROTR)	$(x_n, x_1, x_2, \cdots, x_{n-1})$	$2(2^n-1)$	30	510	131070
11	4-bit shift left (SHL4)	$(x_5, x_6, \cdots, x_n, 0, 0, 0, 0)$	$2(2^n-16)$	0	480	131040
12	4-bit shift right (SHR4)	$(0,0,0,0,x_1,x_2,\cdots,x_{n-4})$	$2(2^{n-4}-1)$	0	30	8190
13	Binary shift left (BSHL)	(SHL(x), SHL(y))	$2(2^{2n}-2^n-1)$	478	130558	8589803518
14	Binary shift right (BSHR)	(SHR(x), SHR(y))	$2(2^{2n-1}-2^{n-1}-1)$	238	65278	4294901758

Table 1 Linear Checks Constants

No.	Instruction (f)	f(x,y)	Kn,f	$K_{4,f}$	$K_{8,f}$	$K_{16,f}$
15	Binary rotate left (BROTL)	(ROTL(x), ROTL(y))	$2(2^{2n}-1)$	510	131070	8589934590
16	Binary rotate right (BROTR)	(ROTR(x), ROTR(y))	$2(2^{2n}-1)$	510	131070	8589934590
17	Binary 4-bit shift left (BSHL4)	(SHL4(x), SHL4(y))	$2(2^{2n}-15.2^{n}-16)$	0	123360	8587968480
18	Binary 4-bit shift right (BSHR4)	(SHR4(x), SHR4(y))	$2(2^{2n}-2^{n}+2^{n-4}-1)$	0	7710	536748030
19	Exchange (EXCH)	(y,x)	$2(2^{2n}-1)$	510	131070	8589934590
20	And (AND)	$x \wedge y$	$2^{n} - 1$	15	255	65535
21	Or (OR)	$x \vee y$	$3(2^{n} - 1)$	45	765	196605
22	Exclusive or (XOR)	$x \oplus y$	$2(2^{n} - 1)$	30	510	131070
23	Compare (CPR)	$\begin{cases} 0 & (\text{if } x \geq y) \\ 1 & (\text{if } x < y) \end{cases}$	$\begin{cases} 2 & (\text{if } x \neq \bar{y}) \\ 3 & (\text{if } x = \bar{y}) \end{cases}$	$\begin{cases} 2 (\text{if } x \neq \bar{y}) \\ 3 (\text{if } x = \bar{y}) \end{cases}$	$\begin{cases} 2 (\text{if } x \neq \bar{y}) \\ 3 (\text{if } x = \bar{y}) \end{cases}$	$\begin{cases} 2 (\text{if } x \neq \bar{y}) \\ 3 (\text{if } x = \bar{y}) \end{cases}$
24	Add (ADD)	x + y	$4(2^{n} - 1)$	60	1020	262140
25	Add with carry (ADDC)	x + y + C	$4(2^{n} - 1) + \sum_{i=1}^{4} C_i$	$60 + \sum_{i=1}^{4} C_i$	$1020 + \sum_{i=1}^{4} C_i$	$262140 + \sum_{i=1}^{4} C_i$

Table 1 (Continued) Linear Checks Constants

No.	Instruction (f)	f(x,y)	$K_{n,f}$	$K_{4,f}$	$K_{8,f}$	$K_{16,f}$
26	Subtract (SUB)	$x - y$	0	0	0	0
27 [3]	Subtract with borrow (SUBB)	$x - y - B$	$-\sum_{i=1}^{4} B_i$	$-\sum_{i=1}^{4} B_i$	$-\sum_{i=1}^{4} B_i$	$-\sum_{i=1}^{4} B_i$
28	Multiply (MPY)	$x \cdot y$	$(2^n - 1)^2$	225	65025	4294836225
29 [4]	BCD add (BCDADD)	$X + Y$	$4(10^s - 1)$	36	396	39996
30	BCD subtract (BCDSUB)	$X - Y$	0	0	0	0
31	BCD multiply (BCDMPY)	$X \cdot Y$	$(10^s - 1)^2$	81	9801	99980001

[1] For instructions 13-19, the value is viewed as a binary 2n-vector.

[2] C_1, C_2, C_3, and C_4 are the carries before the evaluation of $f(x,y)$, $f(x,\bar{y})$, $f(\bar{x},y)$, and $f(\bar{x},\bar{y})$ respectively.

[3] B_1, B_2, B_3, & B_4 are the borrows before the evaluation of $f(x,y)$, $f(x,\bar{y})$, $f(\bar{x},y)$, and $f(\bar{x},\bar{y})$ respectively.

[4] We assume $n = 4s$, X is that decimal s-vector $(X_1, X_2, \cdots, X_s)$, $X_i \in \{0,1,\cdots,9\}$, for which x is the BCD representation (similarly, for Y), $\bar{X}_i = 9 - X_i$ $(i \in \{1,2,\cdots,s\})$, and $\bar{X} = (\bar{X}_1, \bar{X}_2, \cdots, \bar{X}_s)$.

Table 1 (Continued) Linear Checks Constants

For some f in AL (e.g., ADD, ADDC, SUB, SUBB, and those f which are actually functions of one variable (1 - 12 in Table 1)) it can be shown that

(3.2) $\begin{cases} \text{there exists an integer constant } C_{n,f} \text{ such that for} \\ \text{pairs } (x,y) \text{ of binary n-vectors, } f(x,y) + f(\bar{x},\bar{y}) = C_{n,f} \ (= K_{n,f}/2). \end{cases}$

A program for implementing the Linear Checks Method contains instructions enabling the microprocessor to calculate for each instruction f for which (3.1) (or (3.2)) holds the sum in (3.1) (or (3.2)) for some pair (x,y) of n-bit operands. These sums must be compared with the reference sums, $K_{n,f}$ (or $C_{n,f}$) from Table 1. In addition, for some instructions it is desirable to use multiple pairs of operands; for example, in the case n=4, we use 1111 with 0001, 0010, 0100, and 1000 in order to verify that in each bit position a carry can be both generated and propogated. Similarly, for subtraction we use 0000 with 0001, 0010, 0100, and 1000 in order to test borrow generation and propogation. Finally, we use every general-purpose and data-address register as a source register (for the operands x and y) and as a destination register (for f(x,y)) and select pairs (x,y) of operands so as to make all flip-flop outputs in these registers both 0 and 1 and to put 0 and 1 on all lines of the busses exercised.

This program tests the ALU, the general-purpose and data-address registers, the decoding of operation codes, the decoding of register addresses, and the decoding of register addressing modes (we assume, that either operand may come from a general-purpose register or a register which contains a portion of a memory address).

For instructions f for which (3.2) holds, about 15 instructions are required to calculate the linear checks sum. In case (3.1) holds but (3.2) does not hold, about 30 instructions are required to calculate

the linear checks sum. (The number of instructions will depend on the instruction set of the microprocessor.) If there are N instructions in AL for which either (3.2) or (3.1) holds and we do not use multiple pairs of operands, then the number of instructions, I_{LC}, in the above-cited program (approximately) satisfies

$$(3.3) \qquad 15\,N \leq I_{LC} \leq 30\,N\,.$$

If n pairs of operands are used for the instructions ADDC, SUBB, AND, OR and XOR (see Table 1), then I_{LC} is increased by $120(n-1)$ $(= 2 \cdot 15 \cdot (n-1) + 3 \cdot 30 \cdot (n-1))$; however, it is possible to incorporate use of multiple pairs of operands in the program for testing the Parity Checker on the Memory Data Bus In (see Section 5).

The number of reference outputs to be stored, R_{LC}, is given by

$$(3.4) \qquad R_{LC} = 2\,N$$

if multiple pairs of operands are not used; otherwise, $R_{LC} = 2[N + 5 \cdot (n-1)]$. The n-bit output of the ALU - which can be detected at output pins - and the content of the CARRY Latch - which can be detected indirectly (via the Memory Address Bus output pins) by use of a BRANCH ON CARRY instruction - account for the two reference outputs for each f in AL.

The addition of a + 1 unconditional branch instructions to this program or any of the subsequent programs, where a is the number of bits in the Program Counter, allows testing of incrementation by the Program Counter, namely, branches to the following addresses:

$$\underbrace{\begin{matrix} 0000 \cdots 0000 \\ 0000 \cdots 0001 \\ 0000 \cdots 0011 \\ 0000 \cdots 0111 \\ \vdots \quad\quad \vdots \\ 0111 \cdots 1111 \\ 1111 \cdots 1111 \end{matrix}}_{a}$$

In [5]-[8], methods for constructing optimal linear checks and and estimates of their error-detecting capabilities are given.

4. FIXED INPUT DATA METHOD

For testing for faults which have as manifestations the replacement of one Arithmetic/Logic instruction by another instruction, no instruction, multiple instructions, or an invalid instruction, we use the "Fixed Input Data Method." Such faults are one of the important classes of faults in microprocessors [1].

This method is based on the following property: there exists (in the fault-free case) at least one pair (x,y) of binary n-vectors such that different instructions in AL generally yield different results. For most microprocessors, we can choose x and y as follows:

$$x = 1\underbrace{000 \cdots 0}_{n-4}101 \qquad (4.1)$$

$$y = 1\underbrace{000 \cdots 0}_{n-4}011$$

More formally,

(4.2) $\left\{ \begin{array}{l} \text{there exists at least one pair } (x,y) \text{ of binary n-vectors} \\ \text{such that for almost all pairs } (f,g), \text{ where f and g are} \\ \text{in AL and } f \neq g,\ f(x,y) \neq g(x,y). \end{array} \right.$

Prior to add with carry (ADDC), we must initialize CARRY to 1 so that

$$ADDC(x,y) \neq ADD(x,y).$$

Similarly, to distinguish between subtract (SUB) and subtract with borrow (SUBB), we must initialize borrow in to 1.

If testability is considered in designing microprocessors, for those few pairs (f,g) of instructions for which $f(x,y) = g(x,y)$, with (x,y) as in (4.1), the operation codes can be chosen in such a way that a single stuck-at fault can not make these operation codes identical.

In Table 2, we give the value of $f(x,y)$ for some typical Arithmetic/ Logic instructions f and the pair (x,y) defined by (4.1) with $n = 4$, 8 and 16.

A program for implementing the Fixed Input Data Method contains instructions enabling the microprocessor to calculate $f(x,y)$ for each f in the subset of AL consisting of those f satisfying (4.2) and (x, y) as in (4.1).

This program, as well as that for implementing the Linear Checks Method, tests the ALU, the general-purpose and data-address registers, and the decoding of operating codes, register addresses, and register addressing modes.

If there are M instructions in AL, which yield distinct values at the pair (x,y) in (4.1), the number of instructions, I_{FID}, in the above-cited program is given by

$$I_{FID} \cong 4M. \tag{4.3}$$

The number of reference outputs, R_{FID}, is given by

$$R_{FID} = 2M. \tag{4.4}$$

5. PARITY CIRCUITRY TESTING

The circuitry for checking parity and generating parity may constitute a considerable portion of the total hardware of a microprocessor. In the worst case, an n-bit parity checker or generator requires a circuit size proportional to $n \cdot 2^n$ and a test set for detection of all single stuck-at faults which consists of all 2^n distinct binary n-vectors [9]-[10].

Since we have assumed no knowledge of the gate-level implementation of the logic, we must test for this worst case. Fortunately, it is likely that there is a parity bit for each byte of data; thus, 256 $(=2^8)$ appropriately chosen binary n-vectors suffices to detect all single stuck-at faults in a parity checker or generator.

No.	Instruction (f) [1]	(n f(5,[illegible]	[illegible]	[illegible]6) f[illegible]73,32771)
1	CLR	0	0	0
2	TRN	5	133	32773
3	CMP	10	122	32762
4	INCR	6	134	32774
5	DECR	4	132	32772
6	CMP2	11	122	32763
7	SHL	10	10	10
8	SHR	2	66	16386
9	ROTL	10	11	11
10	ROTR	10	194	49154
11	SHL4	0	80	80
12	SHR4	0	8	2048
13	BSHL	166	2566	655366
14	BSHR	33	16961	1073889281
15	BROTL	166	2823	720903
16	BROTR	169	49857	3221405697
17	BSHL4	0	20528	5242928

[1] The instructions corresponding to the above mnemonics are given in Table 1.

Table 2 Reference Outputs for the Fixed Input Data Method

No.	Instruction(f)	(n=4) f(5, 3)	(n=8) f(133,131)	(n=16) f(32773,32771)
18	BSHR4	0	2056	134219776
19	EXCH	53	33669	2147713029
20	AND	1	129	32769
21	OR	7	135	32775
22	XOR	6	6	6
23	CPR	0	0	0
24	ADD	8	264	65544
25	ADDC	9	265	65545
26	SUB	2	2	2
27	SUBB	1	1	1
28	MPY	15	17423	1074003983
29	BCDADD	8	168	16008
30	BCDSUB	2	2	2
31	BCDMPY	15	7055	64064015

Table 2 (continued) Reference Outputs for the Fixed Input Data Method

We assumed in Section 2 that the microprocessor to be tested has a d-bit I/O Data Bus In (=DBI) with a Parity Checker (=PC) and a d-bit I/Ø Data Bus Out (=DBO) with a Parity Generator (=PG) for communication between the CPU chip and external I/O devices. We also assumed it has an m-bit Memory Data Bus In (=MDBI) with a Parity Checker.

A program for testing the PC on the DBI (respectively, PG on the DBO) calls for the inputting (respectively, outputting) of the following 2^8 distinct binary d-vectors (d = 8v, for some integer v)

$$
\underbrace{\begin{array}{ccccc}
0000\ 0000 & 0000\ 0000 & \cdots & 0000\ 0000 \\
0000\ 0001 & 0000\ 0001 & \cdots & 0000\ 0001 \\
0000\ 0010 & 0000\ 0010 & \cdots & 0000\ 0010 \\
\vdots & \vdots & & \vdots \\
1111\ 1111 & 1111\ 1111 & \cdots & 1111\ 1111
\end{array}}_{d} \qquad (5.1)
$$

This program also puts all 2^b distinct binary b-vectors on the b-bit I/O Address Bus for addressing the external I/O devices. We assume that parity on the DBI results in some latch (say, CARRY) being set; thus, before each input instruction, this latch must be reset by using an appropriate instruction. Faulty parity on the DBO is detected at an output pin.

The program for testing the PC on the MDBI calls for putting 2^8 distinct binary m-vectors on the MDBI; these m-vectors are given by (5.1) with "d" replaced by "m", where m = 8w for some integer w. Included among these m-vectors may be instructions of all types (arithmetic / logic, branch, load, store, etc.); thus, this program presents opportunities for detecting single stuck-at faults in the circuitry involved in the execution of such instructions. Some of the 2^8 binary

m-vectors may be invalid instructions. We assume that faulty parity on the MDBI is indicated by a value of 1 at an output pin.

The program for testing the PC on the DBI requires one instruction to reset CARRY, one instruction to input one of the d-vectors in (5.1), and one instruction (a BRANCH ON CARRY) to determine indirectly the value of CARRY; thus, three instructions are needed for each of the 256 d-vectors in (5.1). We assume that data to be output must be put in two on-chip or off-chip general-purpose registers (via, say, transter instructions); thus, the program for testing the PG on the DBO requires three instructions - two to load the registers and one to output - for each of the 256 d-vectors in (5.1). If "I_{IOPAR}" denotes the number of instructions in these two programs, then

(5.2) $$I_{IOPAR} = 1536 \ (=6.2^8).$$

If there is an instruction which both inputs and outputs the same d-vector of data, then $I_{IOPAR} = 768$.

The number of reference outputs, R_{IOPAR}, is given by

(5.3) $$R_{IOPAR} = 512 \ (=2 \cdot 2^8)$$

The number of instructions, $I_{MDBIPAR}$, in the program for testing the PC on the MDBI is given by

(5.4) $$I_{MDBIPAR} = 512 \ (=2 \cdot 2^8)$$

and the number of reference outputs, $R_{MDBIPAR}$, is given by

(5.5) $$R_{MDBIPAR} \leq 512$$

(For all the valid instructions among the 256 m-triples in (5.1) - with "d" replaced by "m" - we are interested in the outputs at certain set of the output pins and for some instructions, we may need to determine indirectly - by branch instructions - the content of one or more of the ALU status latches.)

6. SUBROUTINE TESTING

We have assumed that the microprocessor to be tested has a Link Register (LINK) and p Backup Link Registers (BU_1, BU_2, $\cdots$, BU_p) which are used to save memory addresses and, possibly, other information such as memory paging bits when branches to subroutines are executed.

For testing these registers, we proceed as follows:

· repeat p + 1

(a) Use an unconditional branch to manipulate the content of the Program Counter and other instructions to control any other information which is saved.

(b) Use the instruction for branching to a subroutine, say, BRANCH AND LINK (BAL) to shift "down" the contents of LINK and BU_1, $\cdots$ BU_p and put new information into LINK.

· repeat p + 1 times

Use the instruction for returning from a subroutine, say, RETURN (RTN) to shift "up" the contents of LINK and BU_1, $\cdots$ BU_p, thereby restoring addresses and other information.

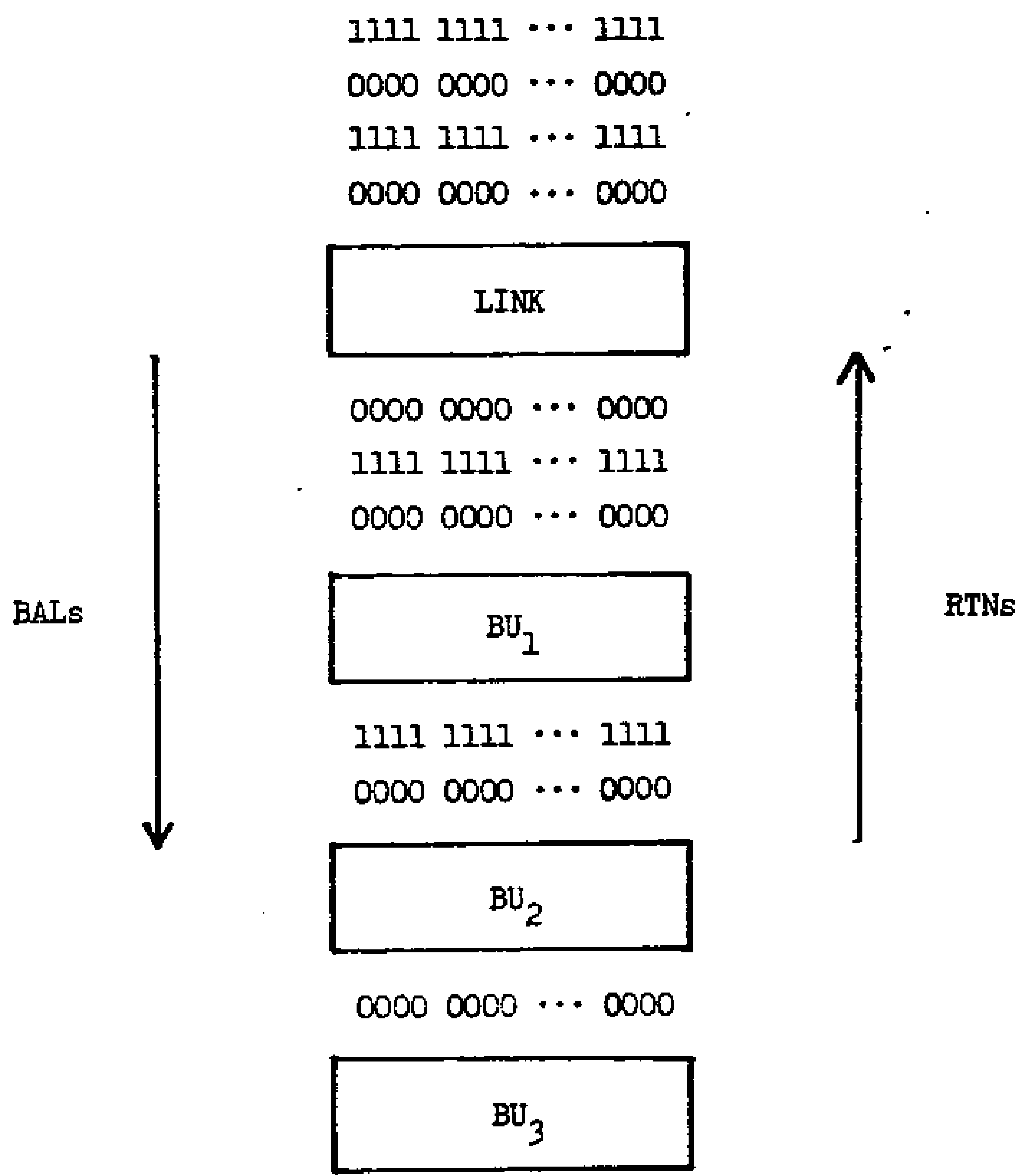
In Figure 1, we illustrate this testing procedure in case there are (p =) 3 Backup Link Registers.

The number of instructions, I_{SUB}, in the program for implementing the above-described testing procedure is given by

$$I_{SUB} = 24 + 8t, \tag{6.1}$$

where t is the number of paging and other flip-flops for which the contents are saved during the execution of subroutines. The number of reference outputs, R_{SUB}, is given by

$$R_{SUB} = 2(p + 1). \tag{6.2}$$



(1) Use four BALs to load the indicated binary k-vectors into the registers; then use four RTNs to unload these registers. Check the restored binary k-vectors.

(2) Repeat (1) with the complements of the indicated k-vectors.

Figure 1 Link and Backup Link Register Testing for $p = 3$

7. INTERRUPT TESTING

We assume that when an interrupt occurs, the contents of certain registers and flip-flops (Program Counter, ALU Status Latches, paging flip-flops, etc.) are saved in the k-bit Link Register (LINK) and the i-bit Interrupt Register (INT) so that at the end of the execution of the interrupt routine restoration of the contents of these registers and flip-flops is possible.

For testing the circuitry involved in the handling of interrupts, we proceed as follows:

- Use an unconditional branch, arithmetic instructions, and instructions for manipulating the paging bits, etc. followed by a call for an interrupt to put some binary $(k + i)$-vector in LINK and INT.
- Call for a return from the interrupt routine.
- Check for proper restoration of the contents of the registers and flip-flops - the content of the Program Counter and the paging flip-flops can be detected by observing output pins (Memory Address Bus for the content of the Program Counter) and the contents of the ALU Status Latches can be detected indirectly by using conditional branch instructions (BRANCH ON CARRY, BRANCH ON ZERO, etc.) and observing output pins (Memory Address Bus).
- Repeat the above with the complementary binary $(k + i)$-vector.

If there are u paging and other flip-flops, in addition to the s ALU Status Latches, for which the contents are saved during interrupts, then the number of instructions, I_{INT}, in the program for implementing the above-described testing procedure is given by

$$I_{INT} = 20 + 4s + 4u \tag{7.1}$$

and the number of reference outputs, R_{INT}, is given by

$$R_{INT} = 2 + 2s. \tag{7.2}$$

8. CASE STUDY

To evaluate the effectiveness of the above-described functional testing techniques, we have applied them to the testing of a 4-bit microprocessor. Programs of the types described in Sections 3 - 7, along with appropriate data, have been run on a simulated microprocessor for the fault-free case and with each of approximately 6800 possible single stuck-at faults simulated.

With the omission of single stuck-at faults associated with the clock system and direct memory access, about which the User's Manual does not give enough information for rational analysis, and the test mode circuitry, the fault coverage for single stuck-at faults was approximately 93.1 percent.

For this microprocessor, the number of instructions, I, in the test program is given by

(8.1) $$I = 1530$$

(of which over 1200 were primarily for testing the parity circuitry) and the number of reference outputs, R, is given by

(8.2) $$R = 892$$

(of which all but 80 were primarily due to the parity circuitry).

This 4-bit microprocessor was designed to be used as a microcontroller and has relatively few Arithmetic/Logic instructions. We have reason to believe that for a microprocessor with a larger set of Arithmetic/Logic instructions and larger operands (say, 16-bit rather than 4-bit) the fault coverage would be higher.

9. SUMMARY

In Sections 3 - 7, we have given functional testing techniques for testing a microprocessor for single stuck-at faults. The testing procedure involves the execution of a machine language program with certain data. We assume a tester can enter the instructions in this program and the data at appropriate times via the input pins of the microprocessor, monitor the output pins, and compare observed outputs with expected (or "reference") outputs.

The number of instructions, I, in the program for implementing these testing techniques may be estimated by by the following approximate inequality:

$$(9.1) \qquad 1324 + 15N + 4M + 8t + 4s + 4u \leq I \leq 2092 + 30N + 4M + 8t + 4s + 4u,$$

where N is the number of Arithmetic/Logic instructions for which (3.1) (or (3.2)) holds, M is the number of instructions in the subset of the Arithmetic/Logic instructions which yield distinct values for the pair of operands given by (4.1), t is the number of paging flip-flops and other flip-flops for which the contents are saved during the execution of subroutines, s is the number of ALU Status Latches, and u is the number of paging flip-flops and other flip-flops for which the contents are saved during interrupts.

The number of reference outputs, R, is given by

$$(9.2) \qquad R \leq 1024 + 2N + 2M + 2p + 2s .$$

The major contributor to both I and R is the parity circuitry testing (see (5.2) - (5.5)). If the nature of the parity circuitry is known, it may be possible to substantially reduce both I and R.

The Case Study of Section 8 demonstrates the efficacy of use of the functional testing techniques given in Sections 3 - 7.

10. ACKNOWLEDGEMENTS

We wish to acknowledge the assistance of Prof. Akfred K. Susskind of Lehigh University and Joel Leininger, Bill Sebesta, and Pat Pignatelli of the International Business Machine Corporation.

REFERENCES

[1] S. M. Thatte and J. A. Abraham, "Test generation for microprocessors," IEEE Trans. on Computers, Vol. C-29, No. 6, June 1980, pp. 429 - 441.

[2] J. C. Rault, "A graph theoretic and probabilistic approach to fault detection of digital circuits," International Symposium on Fault-Tolerant Computing, March 1971, pp. 16 - 29.

[3] R. David and P. Thevenad-Fosse, "Minimal detecting transition sequences: application to random testing." IEEE Trans. on Computers, Vol. C-29, No. 6, June 1980, pp. 514 - 518.

[4] T. Savir, G. Ditlow, and P. H. Bardell, "Random pattern testability," International Symposium on Fault-Tolerant Computing, June 1983.

[5] M. G. Karpovsky, "Error detection in digital devices and computer programs with the aid of linear recurrent equations over finite commutative groups," IEEE Trans. on Computers, Vol. C-26, No. 3, March 1977, pp. 208 - 219.

[6] M. G. Karpovsky and E. A. Trachtenberg, "Linear checking equations and error-correcting capability for computation channels," Proc. IFIP Congress 1977, North-Holland, pp. 619 - 624.

[7] M. G. Karpovsky, "An approach for fault-detection and fault-correction in distributed systems computing numerical functions," IEEE Trans. on Computers, Vol. C-30, No. 12, 1981, pp. 947 - 954.

[8] M. G. Karpovsky and E. A. Trachtenberg, "Fourier transforms over finite groups for error detection and error correction in computation channels," Information and Control, Vol. 40, No. 3, 1979, pp. 335 - 358.

[9] D. C. Bossen, D. L. Ostapko and A. M. Patel, "Optimum test patterns for parity networks," Proc. Fall Joint Computer Conference, 1970.

[10] S. J. Hong and D. L. Ostapko, "A simple procedure to generate optimal test patterns for parity logic networks," IEEE Trans. on Computers, Vol. C-30, No. 5, 1981, pp. 356 - 358.