Memory Testing with a RISC Microcontroller

Ad van de Goor[1,2]
[1]ComTex
Gouda, The Netherlands
Ad.vd.Goor@kpnplanet.nl

Georgi Gaydadjiev[2] Said Hamdioui[2]
[2]Computer Engineering
Delft University of Technology, The Netherlands
{G.N.Gaydadjiev, S.Hamdioui}@tudelft.nl

***Abstract*—Many systems are based on embedded microcontrollers. Applications demand for production and Power-On testing, including memory testing. Because low-end microcontrollers may not have memory BIST, the CPU will be the only resource to perform at least the Power-On tests. This paper shows the problems, solutions and limitations of CPU-based at-speed memory testing, illustrated with examples from the ATMEL RISC microcontroller.**

Keywords: Memory testing, CPU-based memory testing, assembler language, ATMEL RISC microcontroller

I. INTRODUCTION

Many systems are microcontroller based. Applications demand for production and Power-On testing, which includes the memories. Because low-end microcontrollers typically do not have BIST [1, 2], the CPU has to apply at least the Power-On tests. Considering the limited amount on-die memory, these tests have to be compact, while the test time should be short and the fault coverage high.

Current technology typically has many long and thin wires with many interconnections (vias), such that speed related faults are a major concern [3,4,5,6]. The way to detect speed related faults is by performing memory tests at-speed, using Back-to-Back (BtB) memory cycles [4,5,6,7,8,9]. Note: BtB means CPU memory access with a minimum number of clock cycles. Hence, the BtB memory cycle requirement will be used as the main criteria to analyze the capability of the CPU architecture to support the algorithms. This demands that the architecture features are exploited, which dictates the use of assembler language. Hence, the solution is not generic; however, it demonstrates the capabilities and limitations of RISC CPU architectures, as will be shown with cases based on the ATMEL RISC microcontroller [1,2]. CISC architectures exhibit a different set of capabilities and limitations [10]. The authors are not aware of any publications in the area of embedded memory testing for RISC microcontrollers.

This paper contributes to the implementation aspects of memory tests, which is of industrial relevance. Its outline is as follows: Section 2 introduces the ATMEL RISC microcontroller which will be used throughout this paper. Section 3 explains the notation for specifying the algorithms and shows the test consequences of folding and scrambling. Section 4 motivates the March element as the implementation target and describes the used programming conventions. Section 5 illustrates the way the BtB cycle requirement can be met; Section 6 shows memory tests using the ATMEL RISC CPU. Section 7 ends with conclusions and recommendations.

II. THE ATMEL RISC MICROCONTROLLER

Because of the BtB cycle requirement, all cases covered in this paper are written in assembler language. Most readers may not be familiar (any more) with this, therefore a minimal subset of the ATMEL assembler language is introduced at this point.

The ATMEL AT90S8515 microcontroller [1,2] is a Reduced Instruction Set Computer (RISC) architecture. Data manipulation instructions typically involve two byte registers. Communication with the data memory is according to the RISC architecture style, via Load and Store instructions. The data memory **S**tarts at address 'S' (the lower locations are used for register overlays and I/O), and **E**nds at 'E'. The number of to-be-tested **M**emory locations is 'M'.

Table I, left part, shows the set of 32 8-bit registers: R0 – R31. Note that R26 – R31 have an additional meaning; e.g., R30 – R31 form the Z-address register, whereby R30 is the low byte of the address. This way a set of three address registers are supported: X, Y and Z, collectively named 'A'; A ∈ {X,Y,Z}. The right part of Table 1 shows the Addressing Modes (AMs) for the set A. E.g., 'A' denotes Indirect A and 'A+' denotes Indirect A with post-increment; i.e., after the use of the address specified by (A), it is incremented: A←(A)+1. Note that (A) denotes the contents of A and M[(A)] denotes a memory location with address (A).

Table II shows the instructions of interest to this paper. The top block lists the two-operand instructions; note that the ADIW instruction operates on a register pair R-high:R-low (i.e., a Word); C is the Carry flag. The middle; block of the table shows the one- and zero-operand instructions; T denotes a Temporary location, used for setting the condition codes; the LPM 'Load Program Memory' instruction fetches a byte from that memory and stores it into R0. The bottom block of the table lists the control-transfer instructions.

TABLE I. REGISTER USE AND ADDRESSING METHODS 'AMS'

Register	***Use***		***Reg.***	***Use***
R0 – R25	Gen. Purp. reg.		A	M[(A)]
R26 – R27	X- address reg.		A+	M[(A)], post incr.
R28 – R29	Y- address reg.		-A	Pre decr., M[(A)]
R30 – R31	Z- address reg.		*Note*: A ∈ {X,Y,Z}	

III. ALGORITHMS, FOLDING AND SCRAMBLING

In this section, first the algorithm specification will be given. Thereafter, the implementation consequences of Folding and Address scrambling are described.

978-3-9810801-6-2/DATE10 © 2010 EDAA

TABLE II. INSTRUCTIONS

Mnem.	*Name*	*Example*
Two-operand instructions		
LD	LoaD	LD R4, Z. R4← (M[(Z)])
ST	Store	ST Y, R15. M[(Y)]← (R15)
AND	AND	AND R3,R7. R3←(R3) AND (R7)
OR	OR	OR R3,R7. R3←(R3) OR (R7)
EOR	Excl. OR	EOR R3,R7. R3←(R3) XOR (R7)
SBCI	SUB Carry	SBCI R7,$4F. R7←(R7)-$4F-C
LDI	LD Im.	LDI R3,$F0. R3←$F0 (*Note*: $F0 = 240)
ADIW	ADD ImW	ADIW R30,$12. R31:R30←(R31:R30) +$12
One- and zero-operand instructions		
Mnem.	*Name*	*Example*
CLR	CLeaR	CLR R1. R1 ←0
SER	SEt Reg.	SER R7. R← $FF (all 1's)
TST	TeST	TST R4. T←(R4)
INC	INCrement	INC R30. R30← (R30)+1
DEC	DECrem.	DEC R0. R0 ←(R0)-1
ROL	ROT. Left	ROL R2. R2(0)←C;R2(n+1)←(R2(n)); C←(R2(7)). Rotate Left through Carry
CLC	CLear C	C←0. Clear Carry bit
LPM	LD PM	LD from Progr. Mem. R0←(PM[(Z)])
Control transfer instructions		
Mnem.	*Name*	*Example*
BRNE	BRanch if Not EQ	IF Z≠0: PC←(PC)+Displ.
CALL	CALL Subr.	PUSH (PC); PC←Subr.Addr.
RET	RETurn subr.	POP PC

A. March algorithms

The most common algorithms used for testing memories are *March algorithms* [8]. An example of a March algorithm is MATS+, defined as: {⇳(w0);⇑(r0,w1);⇓(r1,w0)}. The special symbols '⇳', '⇑' and '⇓' are the Address Orders (AOs); they determine the way one proceeds from one address to the next address. '⇑' denotes an ascending AO (e.g., 0,1,2,3..), '⇓' denotes a descending AO, while '⇳' denotes a freely chosen AO. MATS+ consists of 3 March Elements (MEs), which are separated by the ';' symbol. The ME '⇑(r0,w1)' specifies the '⇑' AO, while to each address a read with expected value '0' will be applied, after which a '1' will be written.

Memory cell arrays have a matrix organization, which means that the AO needs additional specification. *Fast-row* 'Fr' *(also called Fast-X)*, which is indicated with the AO subscript 'r' (e.g., $_{r}⇑$), means that the row address is modified most frequently. Similarly, *Fast-column* 'Fc' *(also called Fast-Y)* means that the column address is modified most frequently.

A last aspect is the *Counting Method* 'CM', which determines the way one counts. The most common way is the Linear CM, denoted by the superscript 'L' of the AO (e.g., $^{L}⇑$, which specifies the address sequence: 0,1,2,3, etc.) Another CM is the Address Complement (AC) [8]. E.g., the $^{AC}⇑$ for 3-bit address specifies the address sequence: 000, **111**, 001, **110**, 010, **101**, 011, and **100**.

B. Folding

Because memories typically have a large number of words, while the number of bits per word is small (typically 8 or 16) [11,12], *folding* is used such that the layout of the memory array approaches that of a square. It maps the logical memory view, consisting of words and bits/word, into the topological view, consisting of rows and columns.

In this paper **4**-way interleaved **F**olding (**F=4**) is assumed. For a memory with a logical width of 8 bits , this implies 32-bit rows. Figure 1 shows the first two 32-bit rows. Because F = 4, the first 4 bits of every row are the bits-0 of the 4 bytes, the next 4 bits are the bits-1, etc.

The folding scheme affects the way a Data Background (DB) has to be written [11,12]. The DBs of interest are, see Figure 2: *sDB* (solid DB: all 0s or all 1s; not shown in Figure 2), *bDB* (checkerboard DB), *rDB* (row stripes DB) and *cDB* (column stripes DB). Each DB requires its own write/read sequence, which depends on Fast-row (Fr) and Fast-column (Fc) addressing. Examples below illustrate the required write data values for a given DB, assuming the Linear CM.

- Perform **w0** with **sDB & Fr** (solid DB): this means $w0_{0,0}$ (i.e., write 0 to location with row 0 and column 0; see Figure 2), $w0_{1,0}$, $w0_{2,0}$, $w0_{3,0}$, $w0_{0,1}$, $w0_{1,1}$, etc.
- Perform write with **bDB & Fr:** this means $w0_{0,0}$, $w1_{1,0}$, $w0_{2,0}$, $w1_{3,0}$, $w1_{1,0}$, $w0_{1,1}$, etc. Note that the bDB pattern of Table 5 shows Row 0 with 0101 and Row 1 with 1010, etc.
- Perform a write with **cDB & Fc:** this means $w0_{0,0}$, $w1_{0,1}$, $w0_{0,2}$, $w1_{0,3}$, $w0_{1,0}$, $w1_{1,1}$, etc.

Bit 0s Bytes 0-3	Bit 1's Bytes 0-3	Bit 2s Bytes 0-3	…..	Bit 6s Bytes 0-3	Bit 7s Bytes 0-3
Bit 0s Bytes 4-7	Bit 1's Bytes 4-7	Bit 2s Bytes 4-7	…..	Bit 6s Bytes 4-7	Bit 7s Bytes 4-7

Figure 1. An 8-bit memory, 4-way interleaved folded

Row 3	1	0	1	0		0	1	0	1		1	1	1	1
Row 2	0	1	0	1		0	1	0	1		0	0	0	0
Row 1	1	0	1	0		0	1	0	1		1	1	1	1
Row 0	0	1	0	1		0	1	0	1		0	0	0	0
Col :	**0**	**1**	**2**	**3**		**0**	**1**	**2**	**3**		**0**	**1**	**2**	**3**
	bDB					cDB					rDB			

Figure 2. Checkerboard, column & row stripes DBs

C. Address scrambling

Address scrambling means that the Logical address sequence, as applied from the outside of the memory, differs from the physical internal address sequence. A very common cause for address scrambling is via sharing in the address decoders [8,10,11]. Typically, the physical address sequence '0,1,2,3' requires the following logical address sequence: 0, 1, 3, 2. The program fragment below shows how to generate this, using the displacement addressing 'D(Z)'; D= a Displacement, and Z = a Base address register, it consists of R30&R31.

```
LD R0, 0(Z).    Load (M[(Z)]) into R0
LD R1, 1(Z).    Load (M[(Z)+1]) into R1
LD R2, 3(Z).    Load (M[(Z)+3]) into R2
LD R3, 2(Z).    Load (M[(Z)+2]) into R3
```

Note: the ATMEL architecture does not support this AM for single-cycle instructions, needed for satisfying the BtB

cycle requirement. Hence, this form of address scrambling cannot be compensated, which will cause a loss of fault coverage of the tests [12]. It should be noted at this point that many memories do not use this from of scrambling.

IV. TEST IMPLEMENTATION ASPECTS

When implementing several tests, it is desirable to share parts of the implementation, in order to reduce program memory requirements. Subsection 4.1 motivates the choice of the ME as the basic implementation building block. Subsection 4.2 lists the used assumptions and programming conventions.

A. The March Element building block

An algorithm is a collection of March Elements (MEs). E.g., the following March C- algorithm [8] consists of 6 MEs: {⇕(w0);⇑(r0,w1);⇑(r1,w0);⇓(r0,w1);⇓(r1,w0);⇕(r0)}. MEs can be shared between algorithms and may be generic in terms of the used DB. In that case a parameter specifies the to-be-used DB. Therefore the ME is chosen as the basic implementation building block [13].

The algorithms are implemented for the largest data type, because this reduces the test time. If the architecture also allows access to smaller data types, then, for speed-related faults in the data multiplexers the following tests are applied for each of the smaller data types: SCAN+ {⇑(w0);⇑(r0); ⇑(w1);⇑(r1);⇓(w0); ⇓(r0);⇓(w1);⇓(r1)}, with bDB, the Linear and the AC CM will be applied for each of the smaller data types. Scan is known to detect speed faults [14].

B. Assumptions and conventions

The algorithms in this paper assume the following:

- 4-way interleaved folding (F=4). Products may even have 16-way folding; this is a simple extension of F=4.
- No address and/or data scrambling is assumed (to keep the algorithms more comprehensible). Section 3 has shown how to deal with this.

Subroutine names explain their functionality, as follows:

- **UwbLc**: **U**p AO '⇑', **w**rite, **b**DB, **L**inear CM, Fast-**c**. This can be written as: {$^{L}{}_{c}$⇑ (w0)} & bDB.
- **Ur0bA**: **U**p AO, **r**ead-**0**, **b**DB, **A**C CM. The March notation for this is: {^{AC}⇑ (r0)} & bDB. *Note*: Read result accumulation is data value sensitive.
- **Dr0wsLr**: **D**own AO, **r0w**1, **s**DB, **L**in. CM, Fast-**r**. The March notation for this is: {$^{L}{}_{r}$⇓(r0,w1)} & sDB.

V. METHODS TO SATISFY THE BTB CYCLE REQUIREMENT

This section addresses the ways the BtB cycle requirement is met for the following key issues:

- Case 1: Loop counting.
- Case 2: Read result evaluation for 'r0' operations.

Case 1: **the BtB problem due to** ***loop counting***

The loop counting problem will be illustrated with the test: {$^{L}{}_{c}$⇑(w0)} & sDB; i.e., the Linear CM and Fast-column. *Note*: Assume that the write data is in R16.

```
        LDI R18, $M.    Initialize Loop count
        CLR R31.        Initialize Z-high
        LDI R30, $S.    Initialize Z-low
L1:     ST Z+, R16.     Perform a write operation
        DEC R18.        Decrement Loop count
        BRNE L1.        Branch if not equal
```

Problem: In the loop, the memory is accessed only every **3** instructions; this violates the BtB requirement!
Solution: *Loop unrolling*. The name of this **subroutine** is '**UwsLc**', see Section 4.2. Assume that the write data is in R16.

```
UwsLc:  LDI R18, $M/8.  Initialize Loop count
        CLR R31.        Initialize Z-high
        LDI R30, $S.    Initialize Z-low
L1:     ST Z+, R16.     Perform a write operation
        -- repeat the instruction 'ST Z+, R16' 7 extra times
        DEC R18.        Decrement Loop count
        BRNE L1.        Branch if not equal
        RET.            Return from subroutine
```

Advantage: the loop is **U**nrolled 8 times (i.e., **U** = 8), resulting in **8** BtB memory accesses every 10 instructions; the execution time is reduced by 58.3%. However, this is only a *partial* solution to the BtB cycle requirement, because U is only 8. An improvement would be to increase U, for example to 16; however, this increases the program size while it remains a partial solution. The real solution is to do the test once more, to cover the address transitions between every (S+U-1)th and every (S+U)th address; wrap around has to be taken into account. This increases the test time only by a factor of 2/U.

Case 2: **the BtB required due to** ***read result evaluation***

The read result evaluation problem is illustrated with the test: {$^{L}{}_{c}$⇑(r0)} & sDB.

```
        LDI R18, $M.    Initialize Loop count
        CLR R31.        Initialize Z-high
        LDI R30, $S.    Initialize Z-low
L1:     LD R0, Z+.      Perform a r0 operation
        BRNE Error.     Error if value ≠0
        DEC R18.        Decrement Loop count
        BRNE L1.        Branch if not equal
```

Problem: Memory is accessed every **4** instructions because the read result has to be evaluated; this is a BtB violation!
Solution: *Loop unrolling* for the test {$^{L}{}_{c}$⇑(r0)} & sDB.
Note: The name of this subroutine is '**Ur0sLc**', see Section 4.2.

```
Ur0sLc: LDI R18, $M/8.  Initialize Loop count
        CLR R31.        Initialize Z-high
        LDI R30, $S.    Initialize Z-low
        CLR R20.        Initialize Read result value
L1:     LD R0, Z+.      Perform 1st r0 operation
        LD R1, Z+.      Perform 2nd r0 operation
        -- repeat 'LD {R2,R3,..,R7}, Z+'; i.e., 6 instructions
        -- this means: LD R2, Z+; LD R3, Z+; ..; LD R7, Z+
        OR R20, R0.     Accumulate r0 result
        -- repeat 'OR R20, {R1,R2,..,R7}';
        -- this means 7 instructions
        DEC R18.        Decrement Loop count
        BRNE L1.        Branch if not equal
```

```
TST R20.        Check for all 0 read results
BRNE Error.     Error is results ≠ 0
RET.            Return from subroutine
```

Conclusion: The 'LD' instruction does not allow for read result accumulation, therefore the read results have to be stored temporarily into U=8 registers. This allows for **8 B**ack-**t**o-**B**ack '**BtB**' memory accesses in 20 instructions, while the execution time is reduced by **37.5%**.

VI. EXPLOITING THE ATMEL RISC CPU

This section shows the capabilities and shortcomings of the ATMEL RISC architecture features to implement memory tests, under constraint of the BtB memory cycle requirement. It consists of two subsections. First, the testing of the data memory is covered; next, the testing of the program memory.

A. Testing the ATMEL data memory

The capabilities and shortcomings of the ATMEL RISC architecture for testing the data memory are shown in the following cases (Fc/Fr=Fast-column/Fast-row):

- *Case 3:* MATS+, with Linear CM, Fc & sDB
- *Case 4:* MATS+, with Linear CM, Fc & bDB
- *Case 5:* Implementing the Address Complement CM
- *Case 6:* Implementing Fr

Case 3: **MATS+, with Linear CM, Fast-column & sDB**
This test can be written as: $\{^{L}_{c}\Uparrow(w0);^{L}_{c}\Uparrow(r0,w1);^{L}_{c}\Downarrow(r1,w0)\}$ & sDB. The test is implemented, using subroutines, as follows:

```
                CLR R16.        Initialize write data
L_c⇑(w0):       CALL UwsLc.     See Case 1 on page 3.
L_c⇑(r0,w1):    CALL Ur0wsLc.   See Case 3.1
L_c⇓(r1,w0):    CALL Dr1wsLc.   See Case 3.2
```

Case 3.1: **Subroutine for $^{L}_{c}\Uparrow$(r0,w1) & sDB**

```
Ur0wsLc:    LDI R18, $M/8.      Initialize Loop count
            CLR R31.            Initialize Z-high
            LDI R30, $S.        Initialize Z-low
            CLR R20.            Initialize Read result
            SER R17.            Initialize w1 value
L1:         LD R0, Z.           Perform 1st r0 operation
            ST Z+, R17.         Perform 1st w1 operation
            -- repeat above two instructions with {R1,R2,..,R7}
            -- for R0; this means 7 times, with 'LD R1, Z', etc.
            OR R20, R0.         Accumulate r0 result
            -- repeat above instruction with {R1,R2,.,R7} for R0
            DEC R18.            Decrement Loop count
            BRNE L1.
            TST R20.            Check r0 values
            BRNE Error.
            RET.                Return from subroutine
```

Case 3.2: **Subroutine for $^{L}_{c}\Downarrow$(r1,w0) & sDB**

```
Dr1wsLc:    LDI R18, $M/8.      Initialize Loop count
            CLR R31.            Initialize Z-high
            LDI R30, $E+1.      Initialize Z-low
            CLR R16.            Initialize write value (for w0)
            SER R21.            Initialize Read result
L1:         LD R0, -Z.          Perform 1st r1 operation
            ST Z, R16.          Perform 1st w0 operation
            -- repeat above two instructions with {R1,R2,..,R7} for R0
            AND R21, R0.        Accumulate read result
            -- repeat above instruction with {R1,R2,..,R7} for R0
            DEC R18.            Decrement Loop count
            BRNE L1.
            INC R21.            Becomes all 0, due to r1
            BRNE Error.
            RET.                Return from subroutine
```

Case 4: **MATS+, with Linear CM, Fast-column & bDB**
This test can be written as: $\{^{L}_{c}\Uparrow(w0);^{L}_{c}\Uparrow(r0,w1);^{L}_{c}\Downarrow(r1,w0)\}$ & bDB. The test is implemented as follows:

```
                CLR R16.        Initialize w0 data
                SER R17.        Initialize w1 data
L_c⇑(w0):       CALL UwbLc.     See Case 4.1
L_c⇑(r0,w1):    CALL Ur0wbLc.   See Case 4.2
L_c⇓(r1,w0):    CALL Dr1wbLc.   See Case 4.3
```

Case 4.1: **Subroutine for $\{^{L}_{c}\Uparrow$ (w0)} & bDB; i.e., Fc**

```
UwbLc:      LDI R18, $M/8.      Initialize Loop count
            CLR R31.            Initialize Z-high
            LDI R30, $S.        Initialize Z-low
L1:         ST Z+, R16.         Write '0'
            ST Z+, R17.         Write '1'
            --repeat above two instructions once
            ST Z+, R17.         Write '1'
            ST Z+, R16.         Write '0'
            -- repeat above two instructions once
            DEC R18.            Decrement  Loop cont
            BRNE L1.
            RET.                Return from subroutine
```

Case 4.2: **Subroutine for $\{^{L}_{c}\Uparrow$ (r0,w1)} & bDB**

```
Ur0wbLc:    CLR R16.            Initialize w0 data
            SER R17.            Initialize w1 data
            LDI R18, $M/8.      Initialize Loop count
            CLR R30.            Initialize Z-high
            LDI R31, $S.        Initialize Z-low
L1:         LD R0, Z.           Read 1st byte (r0)
            ST Z+, R17.         Write '1'
            LD R1, Z.           Read 2nd byte (r1)
            ST Z+, R16.         Write '0'
            -- repeat above 4 instructions with LD {R2,R3},Z
            LD R4, Z.           Read 5th byte (r1)
            ST Z+, R16.         Write '1'
            LD R5, Z.           Read 6th byte (r01)
            ST Z+, R17.         Write '0'
            -- repeat above 4 instructions with LD {R6,R7},Z
            --Value read in R0, R1,.., R6,R7 is now 01011010
            OR R0, R2.          Accumulate r0 values
            OR R0, R5.          Accumulate r0 values
            OR R0, R7.          Accumulate r0 values
            AND R1,R3.          Accumulate r0 values
            AND R1,R4.          Accumulate r0 values
            AND R1,R6.          Accumulate r0 values
            INC R1.             Should be '0' now
            OR R0, R1.
```

```
        BRNE Error.
        DEC R18.          Decrement Loop count
        BRNE L1.
        RET.              Return from subroutine
```

Case 4.3: **Subroutine for {$^{L}_{c}\Downarrow$(r1,w0)} & bDB.**

```
Dr1wbBc: CLR R16.         Initialize w0 data
         SER R17.         Initialize w1 data
         LDI R18, $M/8.   Initialize Loop count
         CLR R31.         Initialize Z-high
         LDI R30, $E+1.   Initialize Z-low
L1:      LD R0, -Z.       Read '1'
         ST Z, R16.       Write '0'
         LD R1, -Z.       Read '0'
         ST Z, R17.       Write '1'
         -- repeat above 4 instructions with LD {R2,R3}, -Z
         LD R4, -Z.       Read '0'
         ST Z, R17.       Write '1'
         LD R5, -Z.       Read '1'
         ST Z, R16.       Write '0'
        -- repeat above 4 instructions with LD {R6,R7}, -Z
        --value read in R0, R1, R2, .., R7 is now: 10100101
         AND R0, R2.      Accumulate r1 results
         AND R0, R5.      Accumulate r1 results
         AND R0, R7.      Accumulate r1 results
         INC R0.          Should be '0'now
         OR R1, R3.       Accumulate r0 results
         OR R1, R4.       Accumulate r0 results
         OR R1, R6.       Accumulate r0 results
         OR R1, R0.       Accumulate final result
         BRNE Error.
         DEC R18.         Decrement Loop count
         BRNE L1.
         RET.             Return from subroutine
```

Case 5: **Implementing the Address Complement CM**

The implementation of the Address Complement (AC) Counting Method will be shown for the test: {$^{AC}\Uparrow$ (w0)} & bDB. Table 3 shows a memory with 4 rows and 4 columns. The cells have a color to specify the bDBs (non-colored =0; colored=1) and a pair of numbers 'a:b'; 'a' denotes the sequential word number and 'b' denotes the step in the AC CM address sequence. Hence, the data pattern for bDB will be: 0011.0011.0011.0011.**1100.1100.1100.1100**; i.e., *the pattern mirrors itself in the middle*.

TABLE III. ADDRESS COMPLEMENT ADDRESSING METHOD

Row 3	0: 0	1: 2	2: 4	3: 6
Row 2	4: 8	5:10	6:12	7:14
Row 1	8:15	9:13	10:11	11: 9
Row 0	12: 7	13: 5	14: 3	15: 1
Col :	**0**	**1**	**2**	**3**

The implementation uses two address registers: Z for counting Up, and Y for counting Down. The implementation of $^{AC}\Uparrow$ (w0)} & bDB will be done as follows: (*Note*: The write data is in R16 and R17).

```
UwbA:    CLR R16.         Initialize w0 data
         SER R17.         Initialize w1 data
         LDI R18, $M/16.  Initialize Loop count
         CLR R29.         Initialize Y-high
         LDI R28, $E+1.   Initialize Y-low
         CLR R31.         Initialize Z-high
         LDI R30, $S.     Initialize Z-low
L1:      ST Z+, R16.      Perform a w0
         ST –Y, R16.      Perform a w0
         ST Z+, R17.      Perform a w1
         ST –Y, R17.      Perform a w1
         -- repeat above 4 instructions one more
         ST Z+, R17.      Perform a w1
         ST –Y, R17.      Perform a w1
         ST Z+, R16.      Perform a w0
         ST –Y, R16.      Perform a w0
         -- repeat above 4 instructions one more
         DEC R18.         Decrement Loop count
         BRNE L1.
         RET.             Return from Subroutine
```

Conclusion: The AC CM can be supported, satisfying the BtB memory cycle requirement, with the Z+ and -Y AMs.

Case 6: **Implementing Fast-row**

The cells in the memory cell array are numbered in a Fast-column way; see Figure 2. Fast-row addressing requires the following address sequence: 0, 4, 8, 12, etc., until the end of the first column, 1, 5, 7, 9, etc. The generation of this address sequence requires the Base-Displacement AM, as also required for coping with address scrambling (see Section 3.2). As already stated in Section 3.2, this is not supported by the ATMEL CPU; hence, considering the BtB memory cycle requirement, *Fast-row addressing cannot be supported*

VII. TESTING THE ATMEL PROGRAM MEMORY

The ATMEL microcontroller is based on a Harvard architecture, which means separate program and the data memories. The Program Memory (PM) can be accessed via the instruction '**LPM**' (Load PM); it performs the operation: R0 ← (PM[(Z)]); a single byte from the PM, addressed by '(Z)', is transferred to the implicit register R0. LPM takes 3 machine cycles [1,2]; hence, BtB operations are not possible! The proposed algorithm is based on Cyclic Redundancy Checking 'CRC' codes. It reads the contents of the memory locations and generates a checksum, using a characteristic polynomial [8]. The following algorithm and stresses are proposed:

- Algorithm: {$\Uparrow$(r_{CRC}); Check CRC; $\Downarrow$(r_{CRC}); Check CRC} $\Uparrow$(r_{CRC}) means: read & compute CRC
- Stresses: Use Linear CM and Fc

Note: Normally, one also has to apply tests using Fast-row; however, this not supported by the architecture. Similarly, the AC CM cannot be applied, because one needs two address registers to access the PM, while the LPM only uses the Z address register.

A precise CRC implementation is not possible, because of the absence of HW support for CRC polynomial evaluation. Therefore an approximation is made, based on a 24-bit CRC stored in R3—R1, as follows: the read result from R0 is

XORed into R1, then R3-R2-R1 are left shifted, while the bit shifted out-of R3 into the Carry bit, is added to R1.

Case 7. **Implementation of $\{^{L}_{c}\Uparrow(r_{CRC})\}$**

```
        CLR R3.                 Initialize CRC result
        CLR R2.                 Initialize CRC result
        CLR R1.                 Initialize CRC result
        CLR R30.                Initialize Z-low
        CLR R31.                Initialize Z-high
        -- Because PM may be large, the register pair (R28:R29)
        -- is used to contain the length of the PM in bytes.
        -- The SBCI instruction operates on register pairs, but can
        -- only subtract; hence the value –(length of PM) is stored.
        LDI R28, -$PM-low.      Note: $PM is size of PM
        LDI R29, -$PM-high.     R28 and R29: Loop count
L1:     LPM. R0←(PM[(Z)]).      Get byte from PM
        ADIW R30,$1.            Increment Z
        EOR R1,R0.              XOR R0 into R1
        CLC.                    Clear Carry bit
        ROL R1.                 Rotate Left R1
        ROL R2.
        ROL R3.
        SBCI R1, $0.            Add Carry to R1
        ADIW R28, $1.           Increment Loop count
        BRNE L1.
```

Conclusions: The BtB memory cycle requirement is not met; however, this is the best one can do.

VIII. Conclusions and Recommendations

This paper illustrates the capabilities and shortcomings of CPU-based memory testing, using a set of cases for the ATMEL RISC microcontroller. The tests are implemented in assembler language in order to be able to satisfy the BtB memory cycle requirement; hence, they cannot be generic.

The capability of the ATMEL RISC CPU to perform data memory testing can be summarized as follows:

1. The support of the Linear Counting Method (CM):
   - The $^{L}_{c}\Uparrow$ AO (Linear, Fast-column) via the Z+ AM.
   - The $^{L}_{c}\Downarrow$ AO via the -Z Addressing Method (AM)
   - The $^{L}_{r}\Updownarrow$ AOs (Linear, Fast-row) cannot be supported in a BtB way.
2. The Address Complement CM '$^{AC}\Updownarrow$' is supported with a combination Z+ and -Y AMs.
3. March Element (ME) operations
   - No restrictions on the # write operations in a ME
   - The # of read operations per ME is restricted, because the evaluation of the read result requires temporary storage of U registers per read operation in a ME. For U=8, a maximum of 3 reads per ME are possible.
4. MEs can be shared between DBs, as follows:
   - A DB and its inverse
   - Between any DB and sDB
5. Considering the above list, many tests can be applied; for example: Scan, March LR, March U, PMOVI, etc.
6. For low-end systems, the authors recommend the following tests: Scan with Linear and AC CM & bDB, MATS+ with Linear CM with sDB & bDB, March C- with Linear CM & sDB, March LR with sDB.

The capability to perform Program memory (PM) tests is very limited, because:

1. Only the $^{L}_{c}\Uparrow$ AM is supported via the LPM instruction, which takes 3 cycles per instruction and uses the implicit R0 destination register, such that read result accumulation is cycle demanding; hence, BtB cycles are not supported.
2. Tests which like to use: $^{L}_{c}\Downarrow$, $^{L}_{r}\Updownarrow$, $^{AC}\Updownarrow$ are not supported, such that test escapes can be expected.

Algorithms which cannot be implemented at speed are not recommended for implementation, because of the BtB memory cycle constraint. However, static faults, such as the state Coupling Faults (CFsts) and the static Address decoder Faults (AFs) [8] can be detected with tests applied at *any speed*. Therefore, the set of tests should at least include March C-, which is able to detect static faults, such as CFsts and AFs [8]. The simplest implementation suffices; hence, Fast-column & sDB are proposed.

Considering the test generation effort and the shortcomings of the architecture, it is recommended to provide more adequate architecture support for test implementation in next generation products. Several companies have applied the methods described above; both to microcontrollers, as well as to PC systems. The results are a significant increase in fault coverage and a reduction in test time by about 60%.

References

[1] http://www.atmel.com/dyn/products/product_card.asp?family_id=607&family_name=AVR+8%2DBit+RISC+&part_id=2002#DataSheets

[2] http://www.atmel.com/dyn/resources/prod_documents/0841s.pdf

[3] X. Du, N. Mukherjee and W-T. Cheng, 'Full-Speed Field-Programmable Memory BIST Architecture", *IEEE Int. Test Conf.* paper 45.3, 2005.

[4] T. Powell, et al., "Chasing Subtle Embedded RAM Defects for Nanometer Technologies", *IEEE Int. Test Conf.* paper 33.4, 2005.

[5] T. Powell, et al., "BIST for Deep Submicron ASIC Memories with High Performance Application", *IEEE Int. Test Conf*., pp. 386-392, 2003.

[6] Z. Conroy, et al. "A Practical Perspective on Reducing ASIC NTFs", *IEEE Int. Test Conf*., paper 14.2, 2005.

[7] J.B. Khare, et al.., "Embedded Memory Field returns-Trials and Tribulations", *IEEE Int. Test Conf*., Paper 26.3, 2006.

[8] A.J. van de Goor, *Testing Semiconductor Memories, Theory and Practice.* ComTex Publishing, Gouda, The Netherlands., 1998. Ad.vd.Goor@kpnplanet.nl.

[9] L. Dilillo, et.al, "ADOFs and Resistive-ADOFs in SRAM Address Decoders: Test Conditions and March Solutions", *Journal of Electronic Testing-Theory & Applications*, Vol. 22, N° 3, pp. 287-296, June 2006.

[10] A.J. van de Goor, S. Hamdioui, and G.N. Gajdadjiev, "Memory Testing with a CISC Microcontroller", *Submitted to DDECS conf.,* Vienna 2010.

[11] A.J. van de Goor and I. Schanstra, "Address and Data Scrambling: Causes and Impact on Memory Tests", *Proc. of IEEE Int. Workshop on Electronic Design, Test and Applications*, pp. 128-136, 2002.

[12] I. Schanstra, I. and A.J. van de Goor, "Logical and Topological Testing of Scrambled SRAMs", *IEEE Latin-American Test Workshop ,* pp. 66-71, 2003

[13] R.C. Aitken, *."A Modular Wrapper Enabling High Speed BIST and Repair for Small Wide Memories*", Int. Test Conf., pp. 997- 1005, 2004.

[14] A.J. van de Goor and A. Paalvast, "Industrial Evaluation of DRAM SIMM Tests", *IEEE Int. Test Conf.* , pp. 426-435, 2000.