平成１３年度

修士学位論文

スキャンパス構成を利用した
マルチポートメモリアレイの試験

Testing Embedded Multi-Port Memory Array through a Scannable Configuration

1045020　　坂下　雄一

指導教員　　矢野政顕教授

2001 年 1 月 15 日

高知工科大学大学院 工学研究科 基盤工学専攻 情報システムコース

要　　旨

スキャンパス構成を利用したマルチポートメモリアレイの試験

坂下　雄一

本論文は、ＬＳＩ内の埋め込みメモリアレイの試験に有効なスキャン可能メモリ構成をマルチポートメモリアレイに適用して、スキャン可能メモリ構成がマルチポート特有の故障の検出に有効であることを示したものであり、全体は以下の６章から構成されている。

第１章では、研究の主旨と論文全体の構成ついて述べた。

第２章では、ディジタル回路の故障について、想定される故障とそのモデル化、メモリ固有の故障とそのテストについて概観し、大規模集積回路のテストに対処するためにはテスト容易化設計が必須であることを述べた。

第３章では、ディジタル回路に対するテスト容易化設計技術の現状について概観した。それらの中でもランダムロジック部のテスト容易化設計技術であるスキャン設計については、組織的な方法であることから、今後の集積度の向上に伴って有効であることを示した。スキャンパス方式についてはその原理・特長、およびその変形について述べた。

第４章では、第３章で述べたスキャンパス方式をメモリアレイにまで拡張したスキャン可能メモリ構成について、その動作原理、特長について詳述した。具体的には、スキャン可能メモリ構成によってメモリアレイ部とランダムロジック部とを同一のスキャンパスに組み込めること、この結果ランダムロジック部とメモリアレイ部を区別せず試験パターンの自動生成が可能であること、さらにスキャン動作自身によってメモリ固有の故障を検出できることなどについて述べた。

第５章では、第４章で述べたスキャン可能メモリ構成をマルチポートメモリアレイに適用し、スキャン動作によってマルチポート固有の故障が検出できることを示した。具体的には、1Write ポートと 2Read ポートを有するマルチポートメモリアレイを対象とし、マルチポートメモリアレイ固有の故障として２つのワード線短絡故障を想定し、スキャン可能メモリ構成でマルチポートメモリアレイ固有のポート間ワード線短絡故障を有効に検出できることを示した。

第６章では、本研究で得られた成果を要約し、今後に残された課題について述べた。

目次

第 1 章

はじめに

VLSI 技術の進歩によって、複雑なディジタル回路システムを 1 つのチップに実装することが可能になったが、集積度の向上とともにチップが設計通りに機能しているかどうかを検証するテストが複雑になってきた。基本的にテストはチップの入出力端子から行われるが、チップ内部の回路数の増大に比べ、入出力端子の増加が少ないためである。このため設計段階からテストを想定してチップ設計設計するため、テスト容易化設計(Design for Testability)技術の開発に多大な努力が払われてきた。

本論文の前半では、ディジタル回路のテスト技術、特にテスト容易化設計技術について概説する。後半では組織的なテスト容易化設計技術であるスキャン設計技術の適用をメモリアレイに拡張する方法と、さらにその方法をマルチポートメモリアレイに適用して、マルチポート特有の故障を検出する方法について論じている。本論文は以下の 6 章から構成されている。

第 1 章では、研究の主旨と論文全体の構成ついて述べる。

第 2 章では、ディジタル回路の故障について、想定される故障とそのモデル化、メモリ固有の故障とそのテストについて概観する。具体的には VLSI ベースのディジタル回路で起こる故障のタイプと原因、特にメモリ固有の故障について述べる。

第 3 章では、ディジタル回路に対するテスト容易化設計技術の現状について概観した。それらの中でもランダムロジック部のテスト容易化設計技術であるスキャン設計については、組織的な方法であることから、今後の集積度の向上に伴って有効であることを示した。スキャンパス方式についてはその原理・特長、およびその変形について述べた。テスト容易化設計におけるアドホック（Ad-hoc）な方法、埋め込みメモリなどのテストに有効である、BIST(Built-In Self Test)技術の一般的なアーキテクチャについても述べる。

第４章では、第３章で述べたスキャンパス方式をメモリアレイにまで拡張したスキャン可能メモリ構成について、その動作原理、特長について詳述する。具体的には、スキャン可能メモリ構成によってメモリアレイ部とランダムロジック部とを同一のスキャンパスに組み込めること、この結果ランダムロジック部とメモリアレイ部を区別せず試験パターンの自動生成が可能であること、さらにスキャン動作自身によってメモリ固有の故障を検出できることなどについて説明する。

第５章では、第４章で述べたスキャン可能メモリ構成をマルチポートメモリアレイに適用し、スキャン動作によってマルチポート固有の故障が検出できることを示す。具体的には、1Write ポートと 2Read ポートを有するマルチポートメモリアレイを対象とし、マルチポートメモリアレイ固有の故障として２つのワード線短絡故障を想定し、スキャン可能メモリ構成でマルチポートメモリアレイ固有のポート間ワード線短絡故障を有効に検出できることを示す。なお、対象となる回路はすべて VHDL で記述し、シミュレーションを行った。

第６章では、本研究で得られた成果を要約し、今後に残された課題について述べる。

第 2 章

ディジタル回路の故障モード

2.1 ディジタル回路の故障

ディジタル回路またはそのシステムが設計時の意図通りに動作しない場合、その原因は設計バグか、またはそのデバイスの故障である。故障は物理的欠陥により回路が正しく動作しないことであり、主に製造プロセス時に生じるものと考えられる。ディジタル回路の構成要素は論理回路であるが、論理回路の論理機能が故障により異なる論理機能に変化してしまう故障を論理故障(logical fault)という。以下で扱う故障はこの論理故障である。

2.2　故障モデル

VLSI ベースのディジタル回路で起こる故障のタイプと原因について述べるとともに、ゲートレベルでの故障とトランジスタレベルでの故障のモデリング化を行う。

2.2.1 縮退故障(stuck-at fault)

論理故障の中で最もよく扱われるものに、素子の入出力線の値が 1 または 0 に固定される縮退故障がある。ここでは、それぞれ s-a-1 、s-a-0 という省略表記を以後使用する。図 2.1 に示す NAND ゲートの入力 A が s-a-1 であると想定する。図 2.1 の NAND ゲートの出力は s-a-1 が入力 A に存在する場合、入力 A は 1 に縮退しているため、出力 C は常に 0 である。故障の無いゲートは 1 を出力する。結局図 2.1 の故障モデルは入力 A の s-a-1 のテストに使用するのに有効である。なぜなら正しいゲートと故障ゲート間の違いが容易に識別できるからである。縮退故障モデルは故障のタイプを提供する際にもっとも一般的である。ディジタル回路のテストに関する論文の多くが縮退故障モデルをサンプルに使用してきた[1]。

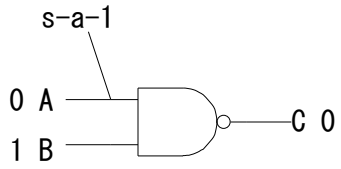


図 2.1 入力 A が s-a-1 の NAND ゲート

縮退故障をトランジスタレベルで考える。図 2.2 は CMOS(Complementary Metal Oxide Semiconductor)で実現させた NAND ゲートである。故障 1、2、3、4 はオープンが生じている。故障 5 と 6 は出力ノードとグランド、出力ノードと Vdd にそれぞれショートが生じていることを示す。CMOS ではフォトリソグラフィによるメタルの除去が不十分だとショートになり、除去し過ぎるとオープンになる。

図 2.2 の故障 1 は入力 A が T1 と T3 から断線している。それは 1 つのトランジスタが導通した場合、もう 1 つは導通しないことを示す。故に故障は s-a-A だと認識される。A が s-a-0 だと T1 は ON で T3 は OFF、s-a-1 だと、T1=OFF、T3=ON となる。故障-3 が存在する場合、T3 は ON のままで、故に故障は縮退故障モデルで表せない。故障 2 と故障 4 はそれぞれ 1 と 3 と同様である。

故障 5 は出力ノードが Vdd とショートしている。それは s-a-1 と想定される。同様に故障 6 は出力ノードを GND とショートさせているので s-a-0 と想定される。

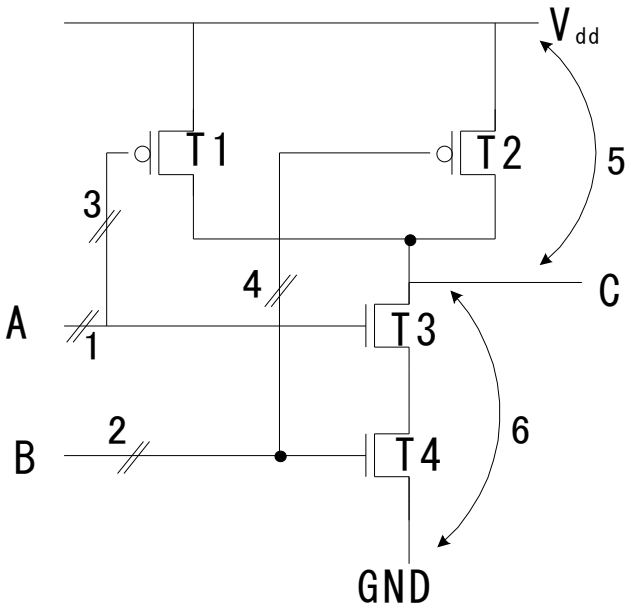
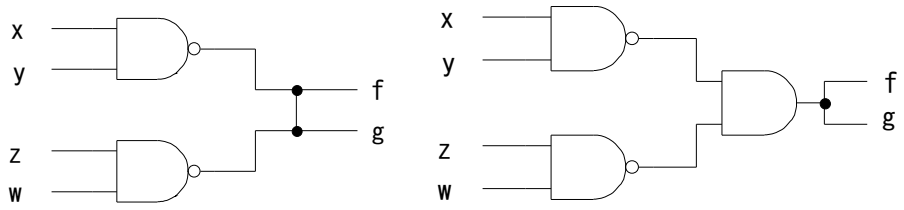


図 2.2　CMOS で実現した NAND ゲート

2.2.2　ブリッジ故障(bridging fault)

ブリッジ故障とは隣接する信号線が短絡する故障である。回路の性質によっては、ショートしている部分が AND ゲートと等価になるか、または OR ゲートと等価になる場合がある。AND ゲートになるブリッジ故障を AND 型ブリッジ故障といい、OR ゲートになるブリッジ故障を OR 型ブリッジ故障という。図 2.3 に示すように AND 型ブリッジの故障の場合は短絡箇所に AND ゲートを挿入したのと等価になる[2] 。



(a)　　(b)

図 2.3　AND 型ブリッジ故障

ブリッジ故障は、「ショートしている信号線がワイアド AND または、ワイアド OR されていることである」というのは、TTL 回路などでは、それで正しいとされてきたが、CMOS 回路におけるブリッジ故障は、それだけでは描写しきれないということが最近、認識されてきている。CMOS でブリッジ故障を分析するには、ゲートレベルよりは、トランジスタレベル、レイアウトレベルで考慮する必要がある。ここでは単ブリッジ故障(Single-Bridging-fault)のみを想定している。CMOS 回路でのブリッジ故障はレイアウトによって決定することを認識することが重要である。

2 入力 CMOS-NOR ゲート回路を考える。単ブリッジ故障の作用を分析するために図 2.4 と図 2.5 を示す。実際起こり得るブリッジ故障は主に 4 つに分類される。表 2.1 は NOR ゲートの真理値表と 4 つの主な故障の作用を示す。分類される 4 つは、

(1) メタルポリシリコンのショート(図 2.4 と図 2.5 の a)
(2) ポリシリコン n 拡散層((図 2.4 と図 2.5 の c、d)
(3) ポリシリコン p 拡散層((図 2.4 と図 2.5 の e、f)
(4) メタルポリシリコンのショート((図 2.4 と図 2.5 の g)

表 2.1 NOR ゲートの真理値表と 4 つの主な故障の作用

Input		Output				
A	B	期待値	ショートa	ショートg	ショートc	ショートe
0	0	1	1	0	0	0
0	1	0	0	0	1	0
1	0	0	1	1	0	0
1	1	0	0	1	1	0

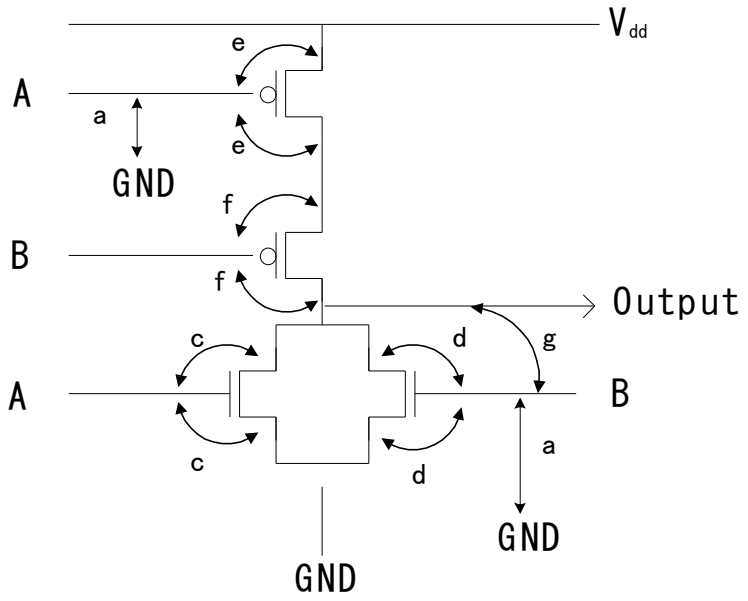


図 2.4　2 入力　CMOS-NOR ゲート

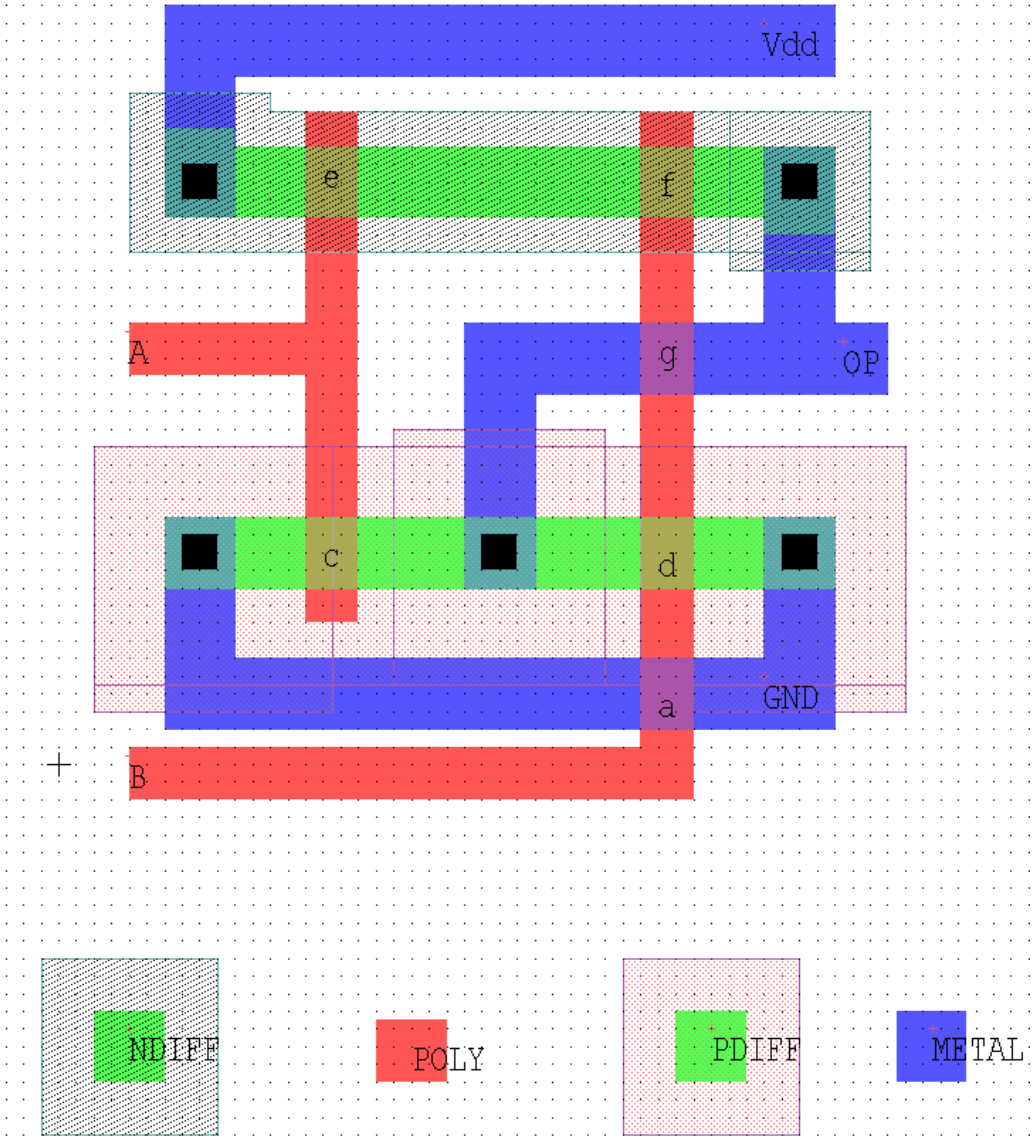


図 2.5　2 入力 CMOS_NOR ゲートのレイアウト図

2.3　メモリ特有の故障

ランダムアクセスメモリ(RAM)はデータを書き込み(write 命令)と読み出し(read 命令)を行うことができる。半導体 RAM は通常、2 つのタイプに分類される。すなわち dynamic(動的)と static(静的)である。

Dynamic RAM(DRAM)はただ 1 つのトランジスタを使用し、データビットを保持する為のキャパシタはきわめて小さい。メモリセルへの、キャパシタの漏れ電流が存在し、そのキャパシタの電流損失は極めて速い。そのためにデータを保持する為にはキャパシタを周期的にリフレッシュチャージしなくてはならない。Static RAM(SRAM)のデータの各ビットは、数個のトランジスタで構成されたメモリセルに保持される。SRAM は一般的に DRAM より高速でアクセスできる。DRAM は、構成要素が小さいため高集積化に適している。両方の RAM は揮発性であるため、電源をオフにすると、データは失われてしまう。

RAM の高集積化が進むほど、チップのテストは難しくなる。また多くの ASIC(Application Specific Integrated Circuit)チップはワード長の異なる RAM チップを混載している。

埋め込みメモリのテストは更に難しく、メモリチップの入出力ピンが通常のメモリと比較すると非常に少ないため、アクセス能力が欠如している。実際的方法としてはメモリへの BIST が行われている。

2.3.1　RAM 故障モデル

様々な故障が RAM で生じる。どのような故障が RAM で起こるかを考慮する前に、一般的な RAM の機能モデルを示して、RAM のアーキテクチャについて述べる。メモリアレイはそれぞれのロウ(横列)とコラム(縦列)の配列により構成される。このアーキテクチャでは、2^n のロウと 2^m のコラムある；各ロウはワード線、各コラムはビット線と呼ばれる。メモリセルはワード線とビット線の交点を 1 つのアドレスと考え、それぞれの物理アドレスと論理アドレスを対応させることにより、メモリの指定されたアドレスにアクセスすることができる。数学でよく使用する x 軸と y 軸で対応させる座標を思い浮かべればよい。

2^n の中から 1 つのロウデコーダがワード線の選択に使用され、2^m 中から 1 つのコラムデコーダがビット線の選択に使用される。読み出し命令の間、アクセスされたメモリセルの内容は、センスアンプ(Sense amplifier)と呼ばれるバッファを通過してデータ出力に転送される。書き込み命令の場合、データ入力からのデータは書

き込みバッファを経由して、メモリセルの指定されたアドレスにアクセスされる。

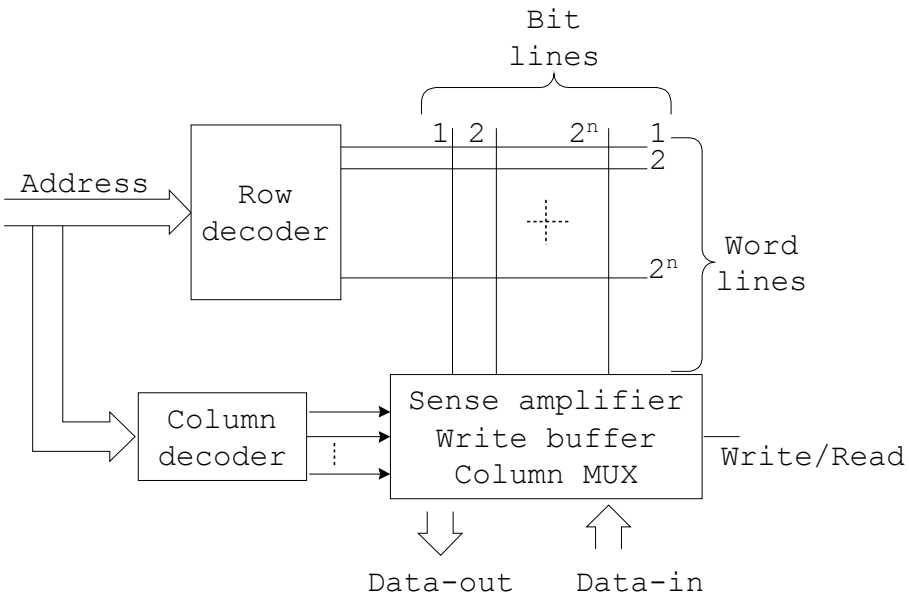


図 2.6 メモリの基本構成図

2.3.2　デコーダの故障

デコーダは、論理アドレスを物理アドレスに写像する装置である。正常なデコーダでは論理アドレスと物理アドレスとが 1 対 1 に対応されている。故障が発生するとこの対応は崩れる。その崩れ方により故障の種類が分類される。アドレスデコーダの故障は以下のものが想定される。

1. メモリロケーションの無い場所へアクセスされてしまう
2. 望まないロケーションへアクセスされてしまう。
3. 1 つ以上のロケーションに同時にアクセスしてしまう。

これらの故障は、論理アドレスと物理アドレスの対応関係を、メモリセルの書き込みデータと読み出しデータとのデータの比較によりテストできる。

2.3.3　メモリセルの故障

メモリセルは 0 または 1 のデータを記憶する回路である。したがって機能をテストするには、各セルに 0 および 1 を書き込み、それを読み出せばよいように思われる。しかし、多数のセルが接近して配置されるためにメモリ特有の故障が発生する。

1. 縮退故障・・セルの内容が 0 または 1 に固定される故障。各セルに 0 と 1 を書き込み、それを読み出すことによりテストできる。

2. 結合故障(coupling fault)・・セル p の内容によって、あるいは内容が変化することによって、別のセル q の内容が変化する故障。前者を静的 CF、後者を動的 CF という。q に 0(1)を書き込んだ後、p の内容を変化させ、その後 q の内容を読み出すことによりテストできる。

3. パターン依存故障(pattern sensitive fault)・・ある注目しているセルの内容が、隣接するセルに記憶されているデータのあるパターンによって、あるいは、あるパターンと近傍セル中の 1 個のセルの内容の変化により、変化する故障。前者を静的 PSF、後者を動的 PSF という。注目するセルに 0(1)を書き込んだ後、近傍セルにデータを書き込み、その後注目するセルの内容を読み出すことによりテストできる。

2.4 RAM へのテストパターン・アルゴリズム

これまでに、様々な RAM へのテストアルゴリズムが提案されてきた。それらのアルゴリズムで取り扱う数の規模は O(n)から O(n*n)に変化した。n は RAM チップ内のビット数である。ここでは簡単に幾つかの RAM へのテストパターンのアルゴリズムについて述べる[3]。

2.4.1 GALPAT (Galloping 0s And 1s)

この技術は、Breuer と Friedman により提案された、メモリアレイの全セルは 0 に初期化される。アドレスはその時、参照アドレスとして選択され、その内容は 0 から 1 に変更される(各ロケーションは 1 ビットのデータを保持すると想定する)。次にそれと別の場所へアクセスされる、そしてその内容が 0 である場合、検証するために読み出される。読み出しと検証は全メモリセルのロケーションに対して行われる。その後、参照アドレスはそのアドレスのデータが 1 かどうか確認するために読み出される。そして同時にその場所に 0 が書き込まれる。その全工程は全メモリセルに対して繰り返される。期待値とテストベクタの比較は 1 つの命令である、そうすると GALPAT の実行時間は N*N である。(N はメモリテスト下内のメモリアレイのセル数である)。

2.4.2 WALKING 0s AND 1s

この技術は GALPAT のようにまず全メモリデータを 0 に初期化する。そして選択された参照アドレスに 1 を書き込む。それ以外のアドレスは，その時各自 0 を読み出す。

これは参照アドレス内での書き込みによりどのロケ-ションも妨害されないよう保証する為に行われる。次に参照アドレスに 0 が書き込まれる、そしてそれ以外は１を参照アドレスとして選択する。

全メモリセルにこの工程が繰り返し行われる。GALPAT との違いは参照するアドレスがこの技術では１度しか読み出されないことである。実行時間は 2(n*n)である。

2.4.3 March-Test

まずメモリを 0 に初期化する。各メモリアドレスは昇順に読み出されてチェックされる．もしその値が 0 であるならば(各アドレスは１ビットデータであると考える)、その時 1 が書き込まれる。1 度読み出し/書き込みの工程が全セルに行われる。次に各ロケーションは降順で 1 を読み出してゆく、そしてその中に 0 を書き込む。図 2.7 は 4*1 のメモリでの各ステップの最後の過程をしめす。何故ならメモリロケは昇順でアクセスされたからである。高順位ロケーションが読み出された時、アクセスされたアドレスのメモリセル間と高順位アドレス間の全ての結合が検知される。同様に降順の方も行われる。またアドレスされる全てのセルは 0、1 両方にセットされるのでセルの縮退故障が検出される。実行時間は N である。N はセルの数である。

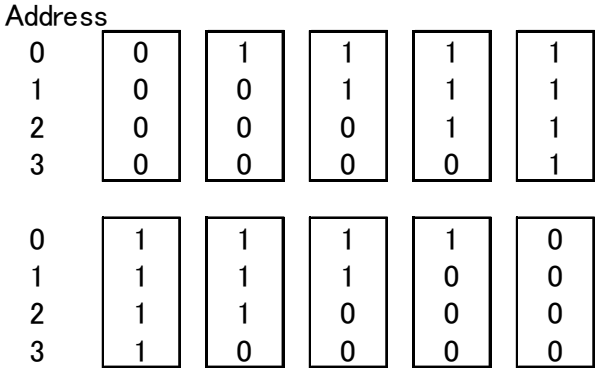


図 2.7　March パターン

第 3 章

テスト容易化設計

3.1　順序回路のテスト容易化設計

近年組み合わせ回路のテストに関する様々な技術が、盛んに議論され、かつ提案されてきた。組み合わせ回路のテストの技術はある程度完成しているといえる。しかしその状況と比較すると順序回路へのテストはまだこれからだという段階である。順序回路のテストは組み合わせ回路のテストよりはるかに難しい。多くの設計指標ではアドホックに観測点や制御点を設け、外部から直接操作できるようにしてきた。しかしこのやり方は経験則的で、設計者の技量に大きく依存してしまう。この方法はあまり工学的ではない。

可テスト性向上のための組織的な手法として回路のメモリ部分に直接アクセスする方法がある。これはスキャンパスとして知られる方法であり、可テスト性改善ために、順序回路に用いられてきた。この章では、アドホックな方法、スキャン設計、および組み込み自己テスト(Built-In Self-Test)ついて説明する。

3.2　可制御性(Controllability)と可観測性(Observability)

テスト容易化設計（DFT : Design for Testability）には 2 つの鍵となる概念がある。それは可制御性と可観測性である。可制御性はテストパターンを回路の 1 次入力を通って内部回路の 2 次入力を制御する能力をいう。例えば、図 3.1 の回路等価性検査回路は、回路がいつも等価の状態である場合、この検査回路が正しく検査しているかどうか把握するのは不可能である。もし制御ゲートが付加されれば、回路の内部状態を制御することができる。つまりは、回路の可制御性を高めることは、1 次入力から制御できない内部入力箇所を減少させることである。

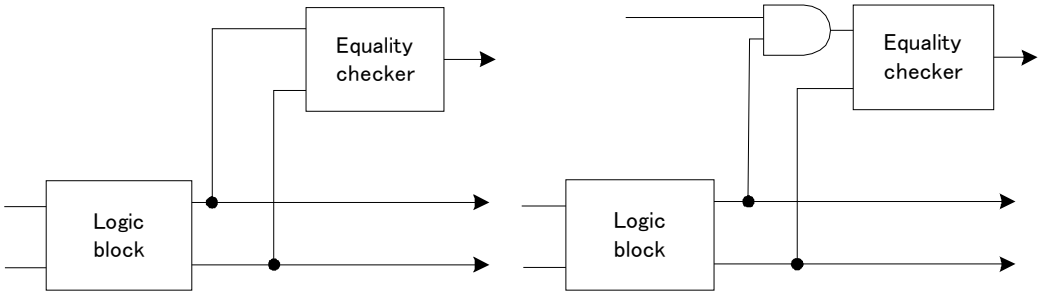


図 3.1 可制御性

可観測性は対象回路の 1 次出力もしくは幾つかの新たにセットした出力点を通って内部回路の応答を観測する為の能力のことである。例えば図 3.2 は 3 つの AND ゲートの出力は OR ゲートの入力に接続されている。s-a-0 が AND ゲート 3 の出力に存在する場合、それは故障の作用が覆われてしまい、1 次入力からは検出することができない。そういう場合、図 3.2 に示すようにゲートの出力を分離しなくてはならない。通常、回路の可制御性／可観測性は幾つかの制御ゲートと入力線を付加することにより実現する(可制御性)。そして幾つかの出力線を付加することにより達成する(可観測性)[5]。

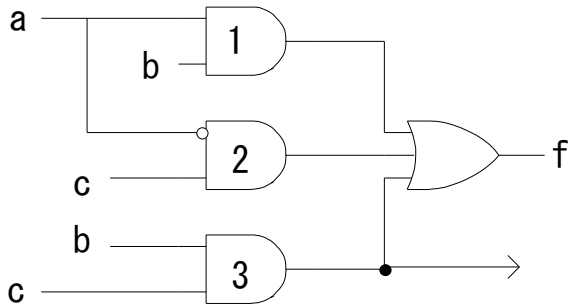
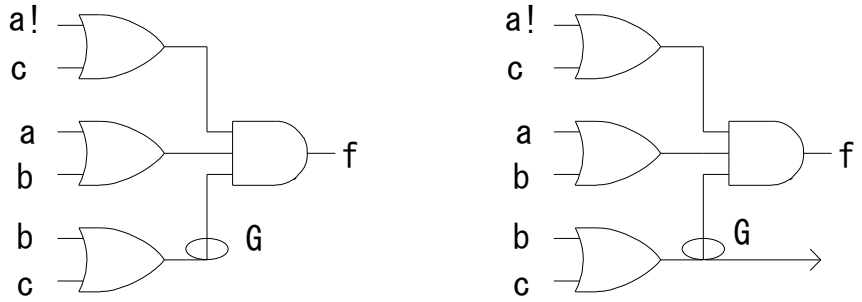


図 3.2　可観測性

3.3　アドホック(Ad-Hoc)手法

アドホックな手法はディジタル回路の可テスト性を改善するために使用される。最も簡単なやり方は、制御点と観測点を回路に付加することである。例えば、図3.3(a)の回路に s-a-1 の故障 G が存在するとする。この回路の出力からは G は検出することはできない。G を検出するには、図 3.3(b)のように、あるポイントにもう1つ出力線を挿入してやればいい。

制御点を付加することによる利点は図 3.4 に示す。NOR ゲートの出力は常に 0 であるが、このままではゲートが正確に動作しているかどうか確認できない。図3.4(b)のように制御点が回路に挿入された場合、NOR ゲートが縮退しているかどうかは容易に検証することができる。

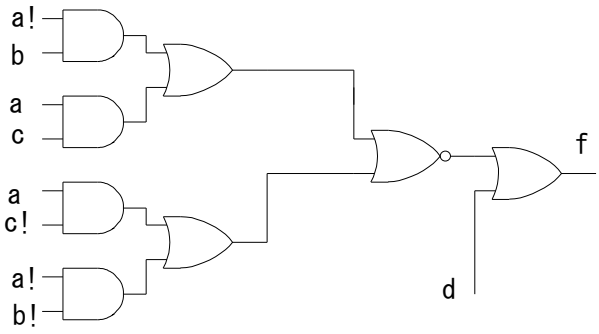


(a)　　(b)

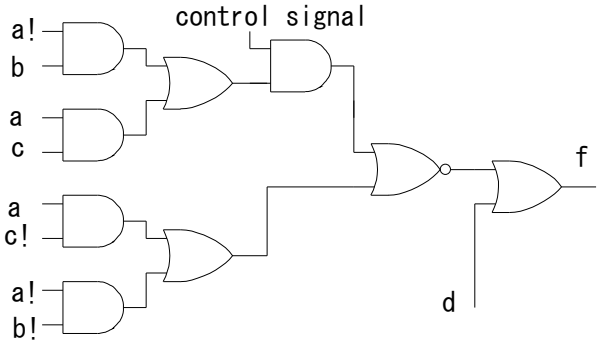
図 3.3　(a)検出できない故障、(b)出力線の挿入

テスト能力を改善する他の方法として、セレクタを挿入して、外部から制御、観測する方法がある。例えば、図 3.5(a)には出力 Z からは検出することができない s-a-0 の故障 G が存在する。図 3.5(b)ではその回路にマルチプレクサを挿入している、入力の組み合わせが 010 の場合にマルチプレクサを通過して故障を検出させることができる。

また内部にアクセスするための別のやり方として、トライステートバッファを使用する方法がある、図 3.6 で示す。テストモード信号はハイインピダンス状態時に使用される。テストモードでは、内部ポイント制御点として使用する。ドライバが活性化された時、内部ポイントはテストポイントとなる。

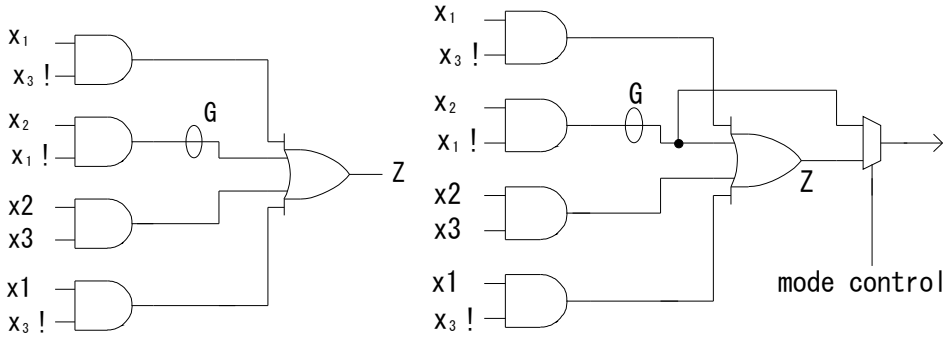


(a)



(b)

図 3.4 制御信号線の挿入



(a) (b)

図 3.5　セレクタを挿入する方法

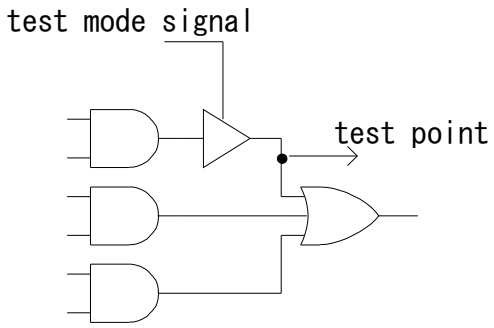


図 3.6　トライステートバッファの挿入

3.4　順序回路のスキャンパス技術を用いたテスト

順序回路のテストが複雑なのは、記憶部分の状態の設定と観測が困難だからである。それらの問題は既存の順序回路に少しの回路を付加することにより克服できる。以下にスキャンパス技術の基本指針を記す[5]。

1. 回路はいかなる所望の内部状態も容易に設定できなくてはならない。
2. 結果の出力パターンから内部状態の入力パターンを容易に抽出できなくてはならない。即ち、もとめる回路の内部状態が観測できなくてはならない。

スキャンパスの基本的な考え方は、回路モードを切り替えるために新たに選択回路を加えることである。この選択回路のスイッチ c=0 の場合、回路は通常の順序回路としての動作を行う。一方 c=1 の場合はレジスタ部分を直列に接続することにより、シフトレジスタとして動作させる。各記憶部分に選択回路を差し込むことにより、シフトレジスタと通常の順序回路の 2 つの動作をスイッチ 1 つで切り替えることができ、2 つの機能は統合できる。それらの選択回路の入力 c は全て同期して一斉に動作する。そして回路は通常モードまたは、シフトレジスタモードに切り替えられる。図 3.8 は D フリップフロップで構成された順序回路で図 3.9 ではその回路をスキャンパス構成にしたものを示す。選択回路を図 3.7 で示す。

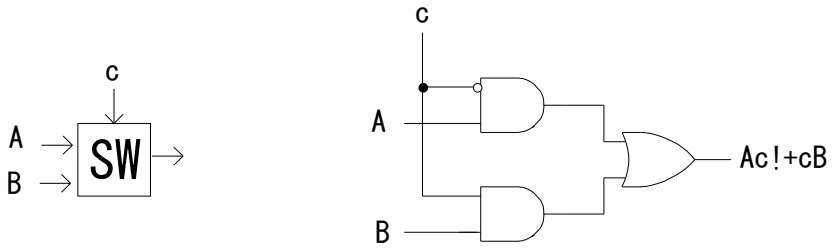


図 3.7　選択回路

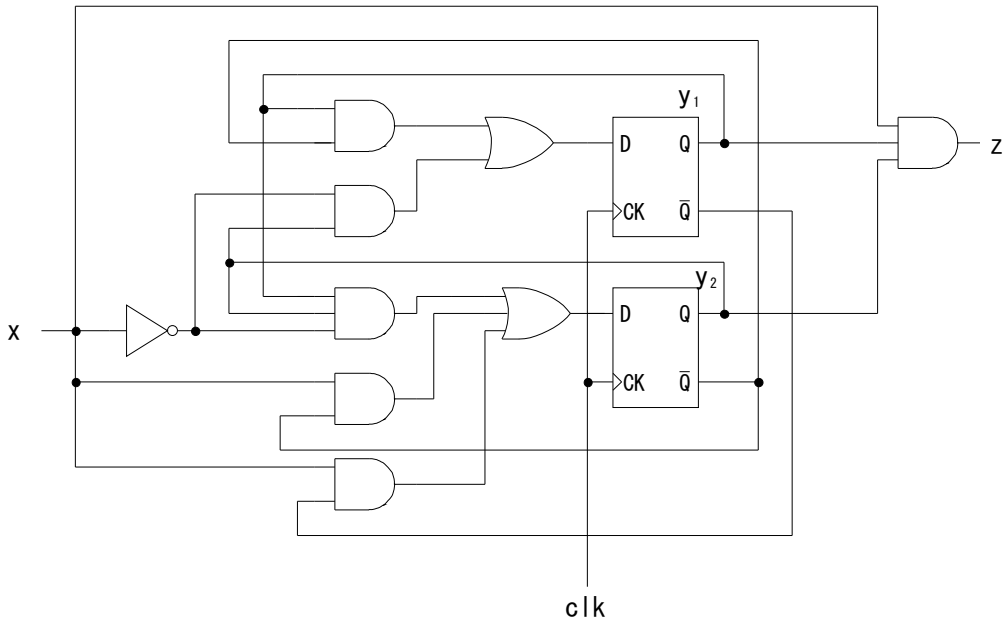


図 3.8　順序回路の例

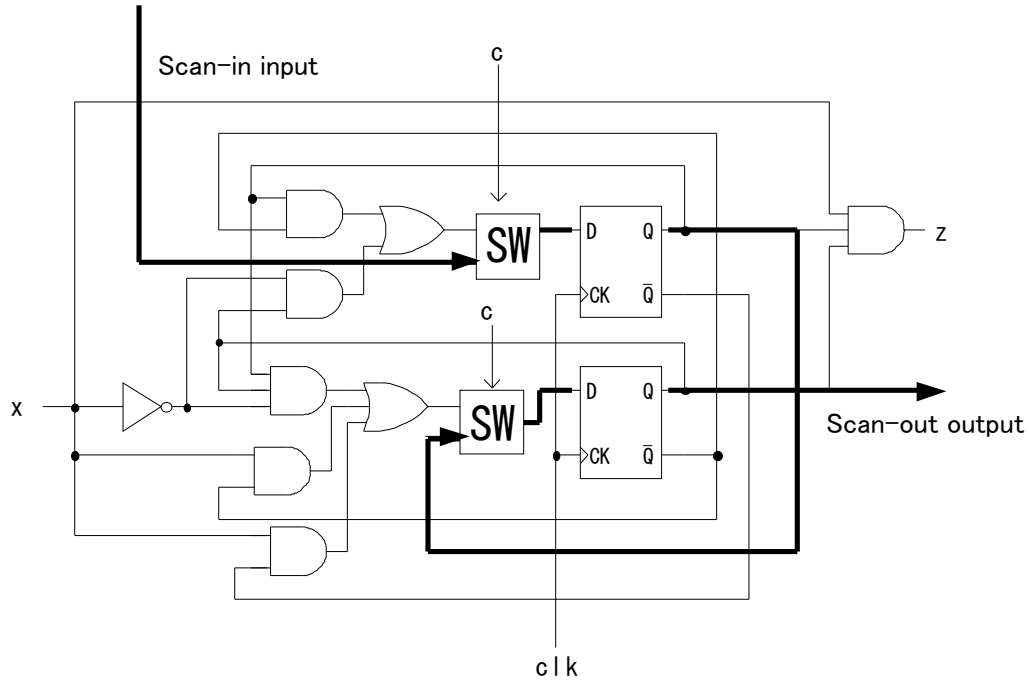


図 3.9 スキャンパスを構成した順序回路の例

シフトレジスタモードでは、最初のフリップフロップは 1 次入力から直接セットすることができる、そして 1 番最後のフリップフロップの出力は 1 次出力から直接観測することができる。これは回路がスキャンイン入力をとおして、内部の下部回路を所望の内部状態に設定することができる。どの外部回路の 2 次出力もスキャンアウトにより完全に観測することができる。テスト方法を以下簡単に述べる。

1. c=1 にセットすると回路はシフトレジスタモードになる。
2. スキャン入力、スキャン出力、そしてクロックによりシフトレジスタは動作する。
3. シフトレジスタの内部状態をセットする。
4. c=0 にして通常モードに戻す。
5. 組み合わせ回路部へテストパターンを入力する。
6. c=1 にセットして、再びシフトレジスタモードにする。
7. 次のテストへの初期状態をセットする間に、現在行っているテストでの内部状態を全てシフトアウトさせる。
8. 手順 4 にもどる。

この手順で、テスト時間の大部分は内部状態をセットするのに割かれる。この方法だとシフトレジスタの長さと同じクロックが必要である。つまりフリップフロップが増えるとテスト時間は比例して増加する。この問題を解決するには、シフトレジスタをいくつかに分割し、並列に動作させることにより解決する。このときのテスト時間は幾つかのシフトレジスタの中で最長のものの、レジスタ長で決まる。

3.5 組み込み自己テスト(Built-In Self-Test)

ここでは、今後の DFT において主流となるであろう BIST(Built-In Self-Test)、とくに論理 BIST について簡素に述べる。論理 BIST とは、いわば LSI テスタをチップ上に実装する技術である。チップ内部にテストパターンを生成する回路と判定データを圧縮する回路を組み込む。BIST はテストコストを低減し、テスト品質を向上させるなどの利点がある。しかし、スキャン設計以上に厳しい設計制約があり、テストポイントの挿入が必要になるという問題もある[6]。

3.5.1 LSI テスタのチップへの実装

図 3.10 では、LSI テストの概念を示す。図 3.10(a)は従来の手法である。テストパターンは LSI テスタからテスト対象となるチップに入力し、その出力値を LSI テストに取り出して、期待値と比較・判定する。これに対して、図 3.10(b)はテストパターンを発生回路とテスト結果評価回路をチップ内に取り込むため、BIST 方式と呼ばれる。こうすると、LSI テスタとチップ間の信号のやり取りを格段に減らせる。

BIST はその用途やテストパターン発生方法により表 3.1 に示すように分類できる。このうち、メモリ BIST はすでに多くの企業で実用化されている。論理 BIST は、ここ数年、多くの LSI メーカが本格的に採用している。

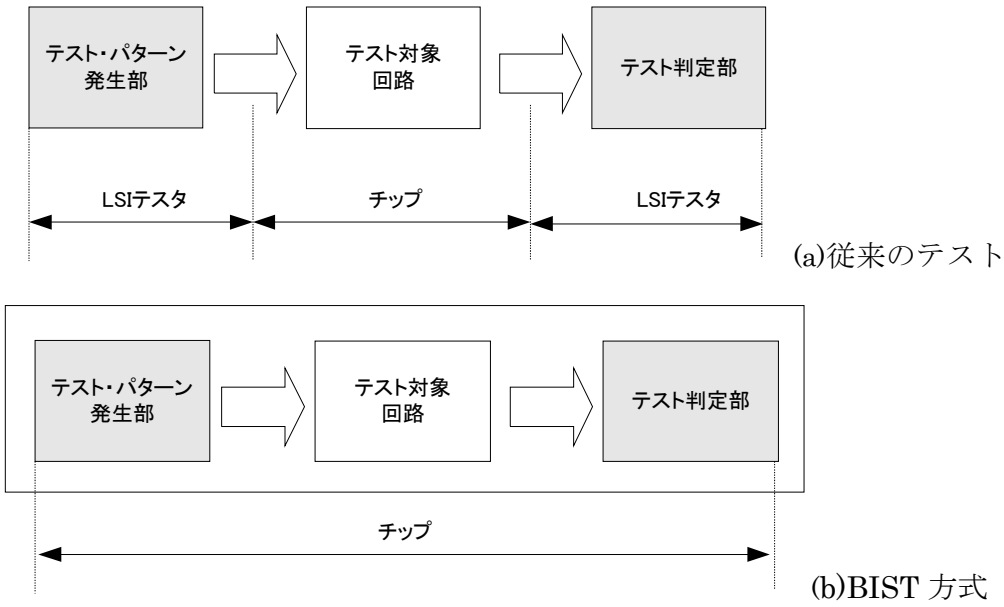


図 3.10 従来のテスト方式と BIST 方式

表 3.1　BIST の分類

	テスト対象となる回路	テストパターン発生部
メモリBIST	SRAM、ROM、DRAMなど	規則的テスト（MARCHなど）
論理BIST	ランダム・ロジック	擬似乱数パターン
機能BIST	プロセッサ・ロジックなど	命令コードなど
アナログBIST	PLL、A-D/D-Aコンバータなど	アナログ波形など

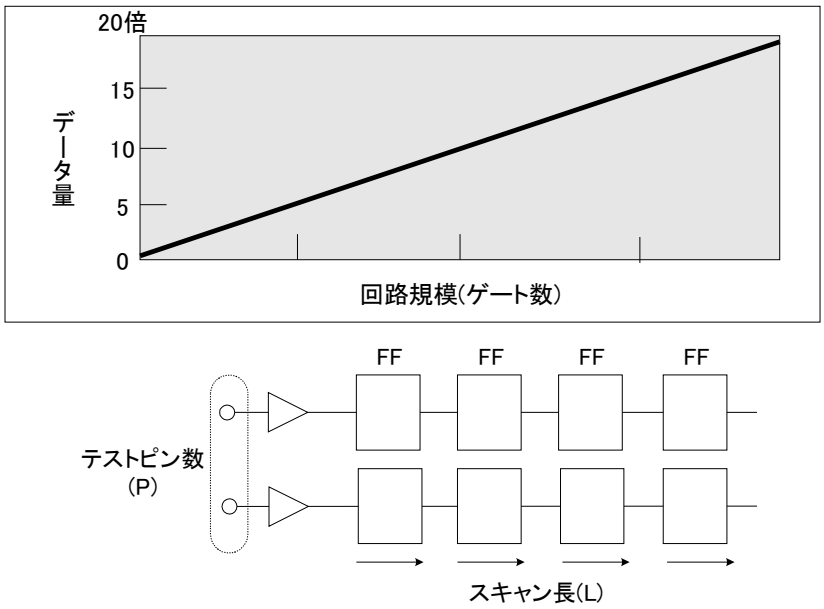
3.5.2　BIST による可テスト性向上

論理 BIST は長い歴史を持っているが、最近特に注目を浴びている理由は大きく 2 つある。即ち

(1) 従来手法ではテストできない。

(2) もっとよいテストを行いたい。

ということである。



データ量/テストピン∝スキャンチェーン長(L)×テスト数(T)
テスト時間∝スキャンチェーン長(L)×テスト数(T)
　　L∝FF 数/P∝回路数(G)/P
　　$T \propto G^{\alpha}$（α=1～2)
→データ量/テストピン、テスト時間$\propto G^{1+\alpha}/P$

図 3.11　スキャンテスト法のテストデータ量とテスト時間

まず、従来手法ではテストできない、という問題について述べる。数百万ゲート規模の LSI を従来のテスタを使用する方式でテストを行う場合、テストパターンのデータ量、あるいはテスト時間がもはや限界にきている。図 3.11 は先ほど述べたスキャンテスト方式を用いた場合のテストデータ量およびテスト時間の推移を示している。スキャンテスト方式では、LSI テスタから LSI ピンを通して、テストパターンが次々と入力される(スキャン入力)。複数本のスキャンチェーンを伝って、全 FF に決められた値をセットし終わると、クロックをたたいて FF にはさまれた組み合わせ回路部を 1 回テストする。このサイクルをテストの回数分だけ繰り返す。従がって LSI ピンあたりのデータ量およびテスト時間は、スキャンチェーン長(L)×テストパターン数(T)に比例する。これは図 3.11 に示すように、回路規模(G)および、テストピン数(P)で表すと、$G^{1+\alpha}/P$ に比例する(α は経験上 1～2)。P を増加させればよさそうに見えるが、LSI ピン数の増加傾向は回路規模と比べて緩やかなことと、LSI テスタの投資コストはテストピンに比例して増加することなどから、これを実現することはほぼ不可能である。結局、テストデータ量とテスト時間は、回路規模の増大とともに指数関数的に増大するため、早晩、現在のテスト方法が破綻することはまちがいない。

論理 BIST を用いると、LSI テスタとチップの間のテストパターンの出し入れが最低限ですむので、この問題を本質的に回避することができる。

次に(2)のもっとよいテストを行いたいという問題について述べる。論理 BIST を積極的に活用すると、従来の方法より質の良いテストを行うことができる。その例を幾つか述べていく。

例1)　テストコストを削減できる

論理 BIST を採用すると、LSI テストピン数を減らすことができる。これにより、複数チップの同時テストが可能になる。例えば、図 3.12 のように従来方法で 256 ピンの LSI テスタが必要であったところを、論理 BIST の採用によりテストピン数を 64 ピンに減らしたとする。これによって、既存の LSI テスタで 4 チップを同時にテストできるようになる。すなわち、テスト用の装置コストをおよそ 1/4 に低減できる。また新規の LSI テスタを導入する場合、64 ピンの低価格のテスタですませるという選択肢も生まれてくる。LSI テスタの価格は、ほぼ対応するテストピン数に比例する。

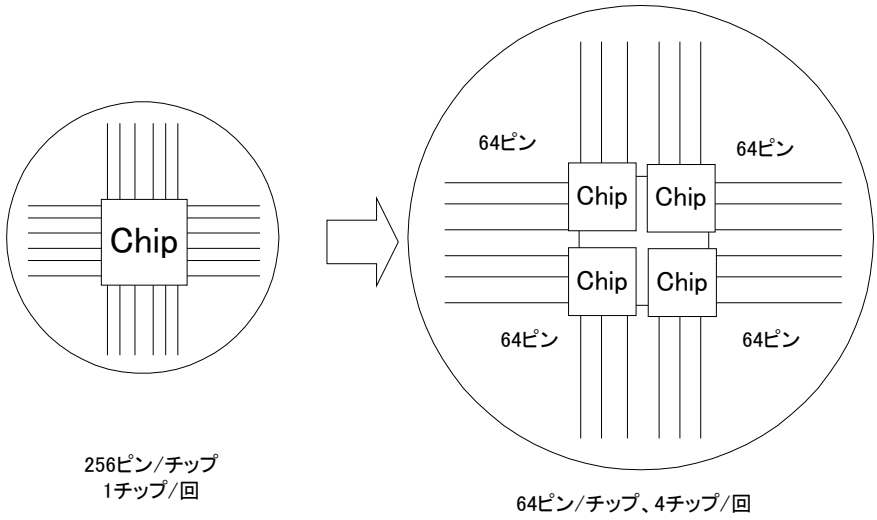


図 3.12　テストコストの積極的な削減

例2)　スクリーニング能力が向上する

論理 BIST は、テストパターンを多数、高速に発生することができるので、LSI の物理的欠陥のスクリーニング能力(良/不良の選別能力)は、従来のスキャン方式より高いといわれている。

図 3.13 に例を示す。LSI チップ上でとなり合う 2 つの信号配線が製造時の異物などでショートしている場合を考える。これを LSI テスタで検出するには、性常時に 2 つの信号配線の電位が異なる必要がある。従来の縮退故障を用いたスキャンテストでは、これを検出できる保証なない。論理 BIST では、擬似ランダムのテストパターンを多数発生させるため、1 回のテストで両方の信号配線の電位が異なる確立は約 1/2 となる。ほぼ独立な多数のテストを行うことで、その確立は 1 に近づく。

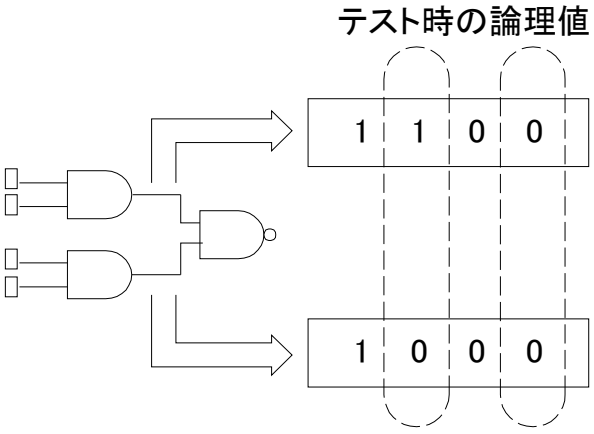


図 3.13　スクリーニング能力の向上

例 3) SOC(System On a Chip)テストに対応できる

SOC では概設計の回路ブロックを IP コア(Intellectual Property)とし扱うが、それぞれの IP コアのテストをどう実施するかは悩ましい問題である。入出力ピンまでなんらかの手段で信号を引き出してテストするのが一般的な考え方であるが、IP コアのピン数が LSI のピン数より多いというケースも珍しくない。図 3.14 の IP コア用 BIST はこの問題の解決策になる。またこの考え方を拡張して、ボード上で LSI を自己テストさせることも可能である。これは装置の保守や故障診断のための強力な武器となる。

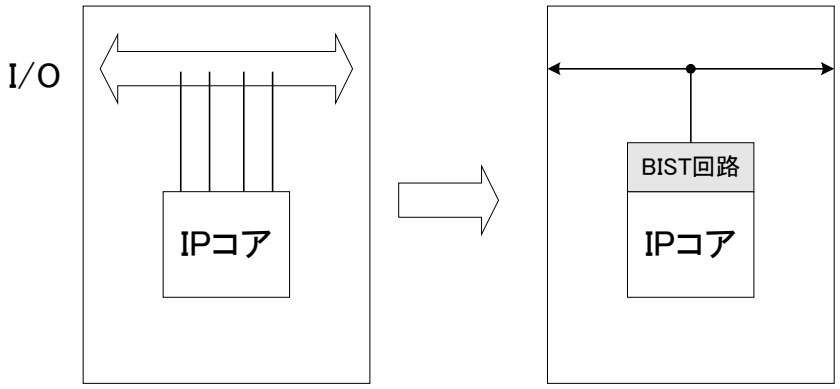


図 3.14　SOC への対応

3.5.3　BIST アーキテクチャ

典型的な BIST ではスキャンテスト方式をそのまま踏襲するが、テストパターンが LSI 内部の擬似乱数パターン発生回路出力される点と、テスト結果が LSI テスタではなく出力パターン圧縮回路へ送られる点が異なる。これはスキャンベース BIST とも呼ばれている。擬似乱数パターン発生回路の構造は図 3.15 に示す LFSR(Linear Feedback Shift Register)が基本となっている。この例では、3 ビットのシフトレジスタなので、全ビット 0 の状態をのぞけば、$2^3-1=7$ の状態が存在する。途中のビットからフィードバックがかかることにより、内部状態の出現順序はランダム性が増す。結論として以下のことが証明される。

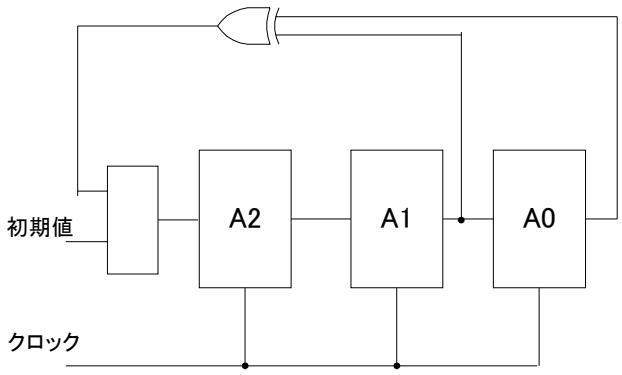
(1) フィードバックのビットを適切に選べば、N ビットのシフトレジスタの場合、(2^N-1)回目に初めて同じ状態が出現する。

(2) 各ビットバターンの圧縮回路は MISR(Multiple Input Signature Register)

と呼ばれる。パターン発生回路とは相対の関係になっている。N 次のビットパター

ンをテスト結果として次々と入力し、シグネチャと呼ぶ N 次ビット列状態に圧縮する。途中で異なるビット列を入力すると、シグネチャは必ず異なる。

即ち、実用範囲内では、パターン発生回路は毎回ランダムなテストパターンを発生し、圧縮された N ビットのシグネチャを見れば良、不良の判定を行える。LSI テスタからはパターン発生回路の初期値の入力と最終シグネチャの読み出し、クロックおよび電源の供給、テストモードの制御しか行わないので、ピン数は少なくてすむ。



	A2	A1	A0
状態0（初期値）	1	0	0
状態1	0	1	0
状態2	1	0	1
状態3	1	1	0
状態4	1	1	1
状態5	0	1	1
状態6	0	0	1
状態7	1	0	0

図 3.15 擬似乱数パターン発生器(LFSR)

第4章

スキャンパス方式によるメモリアレイの試験

4.1　メモリアレイのスキャン方式

スキャンパス方式は順序回路に対する DFT の 1 つの手法としてよく知られており、プロセッサのランダムロジック部の試験にも採用されている。しかし同じチップに内蔵されているメモリアレイに対しては、スキャンパス方式を直接適用できないため、アドホックな方法や BIST 等が適用されてきた。本章では、レジスタファイルやキャッシュメモリのような論理回路で構成されたメモリアレイをスキャン可能にし、そのメモリアレイをワード長やビット長の異なるレジスタ群と同一のスキャンパスへ組み込むことを可能にする方式について述べる[7]。まずメモリアレイをシフトレジスタとして動作させるアルゴリズムと、このスキャン方式を実現するために付加する回路について述べる。さらにワード数の異なるレジスタといかにして同一のスキャン構成に接続できるかを述べる。この方式の特色は、メモリセル、アドレスデコーダなどを通常の論理回路と同等に扱い縮退故障検出の対象とできること、スキャン動作自身がメモリアレイの試験になっていること、故障の解析が容易であることなどである[8]。

4.2　スキャンアルゴリズム

このスキャンパス方式で使用される回路は、モード制御信号によって通常動作とスキャン動作のデータを切りかえる 2 入力選択回路を備え、クロックをもちいて同期動作するものとする。

書き込みと読み出し各 1 ポートを備えたメモリアレイのスキャン方式について述べる。このメモリアレイは、クロックのフロントエッジでデータを書き込むタイ

プを前提としている。また読み出しアドレスで指定されるメモリデータはクロックに非同期で出力されるものとする。更にメモリアレイは、n ビット、m ワードで構成されており、M(i, j)でワード i の j ビット(大→小)のデータを表すものとする。更にメモリアレイは、1 ビットのスキャンデータ入力(SDI)と 1 ビットのスキャンデータ出力(SDO)を備えているものとする。

本章で取り扱う方式では、メモリアレイの総ビット数に相当するクロックを供給してメモリアレイをシフトレジスタとして動作させる。すなわちスキャンモードでメモリアレイに w×n の数のクロックを供給して、全メモリセルの初期データを M(0, 0), M(1, 0), …, M(w-1, 0), M(1, 1), …, M(w-1, 1), …, M(1, n-1), …, M(w-1, n-1)という順序で SDO から読み出し、SDI から入力されるデータを同じ順序で書き込む。簡単な例として、4 ワード×4 ビットのメモリアレイについて上記のアルゴリズムを説明する。図 4.1(a)はスキャン動作の初期状態を示し、16 ビットの各メモリセルの状態を A、B、C、・・、O、P またスキャン入力データを a、b、c、・・・、o、p(各文字は論理値「1」または論理値「0」を表す)と表示している。読み出しデータはワード 0 の「M-I-E-A」であり、最下位ビットの「A」がスキャン出力される。また書き込みデータもワード 0 からであり、読み出しデータを 1 ビットシフトした、「a-M-I-E」であり、このデータは次のクロックでワード 0 に書き込まれる。

同様に 1、2、3、4 番目のクロックが入力されると、メモリアレイの状態は図のように変化する。この様に、既存のセルアレイの内容は右に 1 ビットシフトし、左端にテスト入力を 1 ビット付加したものを書き込みデータとする。この様に 16 クロックが供給されると、メモリアレイの初期データは全てシフト出力され、スキャン入力データと置き換えられる。メモリアレイのスキャンデータを時系列で示したものが図 4.1 である。

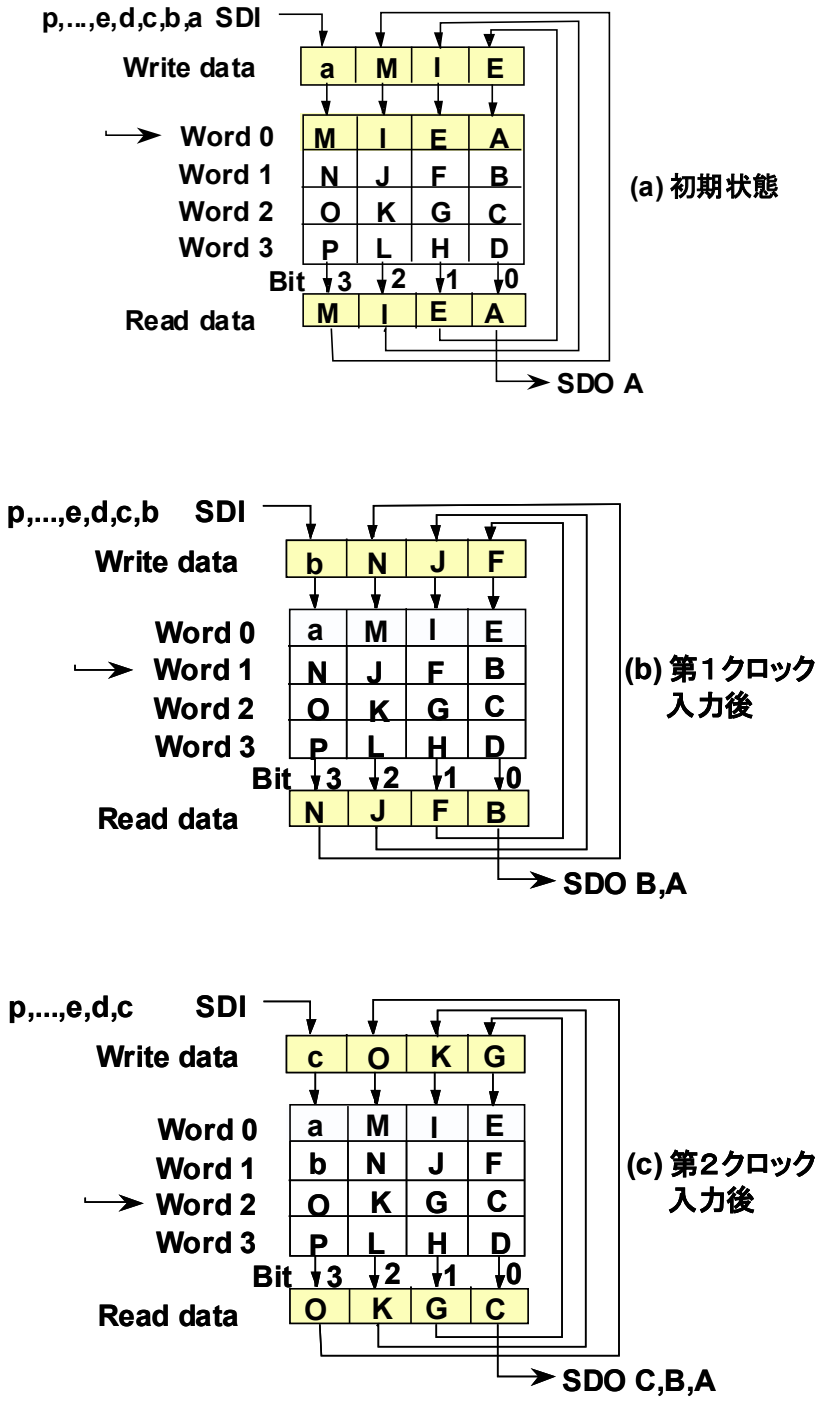


図 4.1　４ワード・４ビットメモリアレイのスキャン動作(1/2)

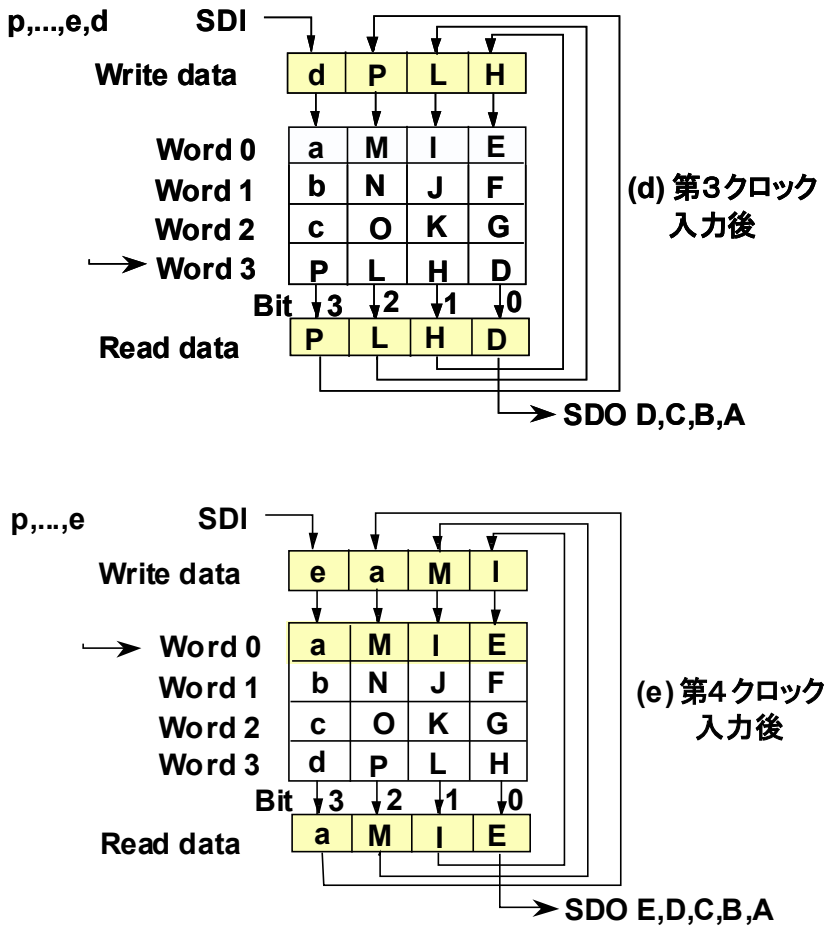
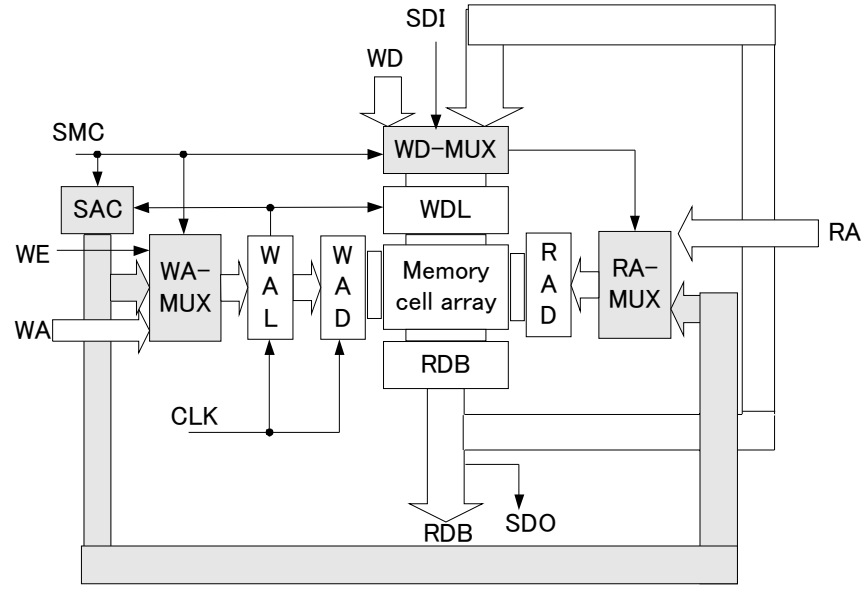


図 4.1　4 ワード・4 ビットメモリアレイのスキャン動作(2/2)

4.3　対象回路の構成と動作

先述したアルゴリズムは、通常のメモリアレイに簡単な回路を付加することによって実現できる。



WD: Write−data
RD: Read−data
RA: Read−address
SDI: Scan−data input
RDB: Read−databuffa
WA−MUX: Write−address multipulexer
WD−MUX: Write−data multipulexer
RA−MUX: Read−address multipulexer
SAC: Scan−address counter
SDO: Scan−data output
SMC: Scan−mode controll

図 4.2　スキャン可能なメモリアレイの構成

付加する回路は図 4.2 で示す SAC(Scan Address Counter)、WA-MUX (Word Address-Multiplexer)および、RA-MUX(Read Address-Multiplexer である。SAC は、スキャン状態でのアドレスを供給するためのカウンタであり、通常動作モードでは、リセットされている。従がって SAC は、スキャン動作の開始時点ではワード 0 を指定しているが、以後はクロック同期して 1 づつ増加していく。また最大ワードの次はワード 0 に戻る。WA-MUX (Word Address-Multiplexer)および、RA-MUX(Read Address-Multiplexer)は、スキャン状態では、SAC の内容を選択し、通常動作状態ではそれぞれ通常の書き込みアドレス(WA)および通常の読み出しアドレスを選択する。SAC がワード i を指定している時、WD-MUX (Word

Data-Multiplexer)は読み出されたワード i の内容を 1 ビット右シフトしその左端にスキャン入力ビットを付加したものをワードiへの書き込みデータとして選択する。このとき右にシフトされた右端の 1 ビットのデータがスキャン出力ビットとなる。この状態でクロックを供給すると、ワードiへの書き込みが実行されると同時に、SAC は変化してワード i+1 を指定する。i＋ 1 が最大ワードである場合、ワード 0 に戻る。この動作をメモリアレイの全セル数分繰り返すと、初期状態における全メモリセルのデータは SDO からシフトされ、SDI からの入力データに全て置き換えられる。このとき SAC はワード 0 を指定した状態で停止する。これにより先述のアルゴリズムは実現できる。

4.4　スキャンパス方式の適用領域拡大

4.4.1　ワード数の異なるメモリアレイの直列接続

スキャンパス方式をランダムロジック部だけではなくメモリアレイを含む順序回路全体に拡大適用するには、メモリアレイと FF 回路を同一のスキャンパスに組み込む必要がある。

その準備として、まずワード数・ビット数の異なる複数個のメモリアレイを、スキャンパスに関して直列接続する場合を考える。接続するメモリアレイのワード数が同じでビット数が異なる場合は、ビット幅が広い 1 つのメモリアレイとみなすことができるので、従来の方法をそのまま適用できる。ワード数が異なるメモリアレイを直列接続した場合には、スキャン動作の最終状態において、スキャン入力データがワードの昇順に配列されない場合が生じ問題となる。

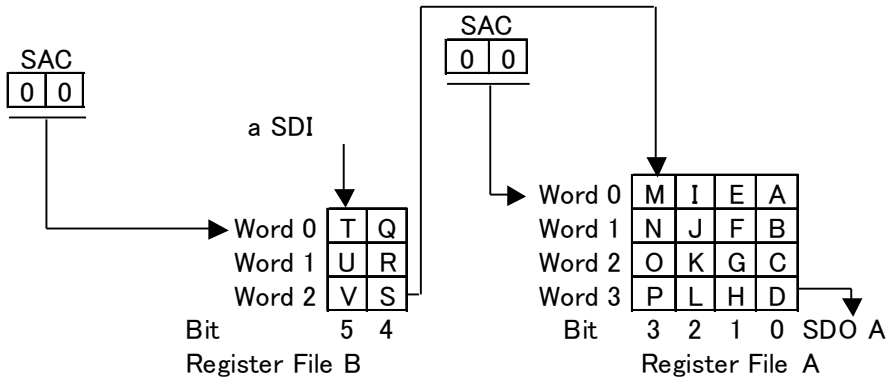


図 4.3　ワード数の異なるメモリアレイの直列接続

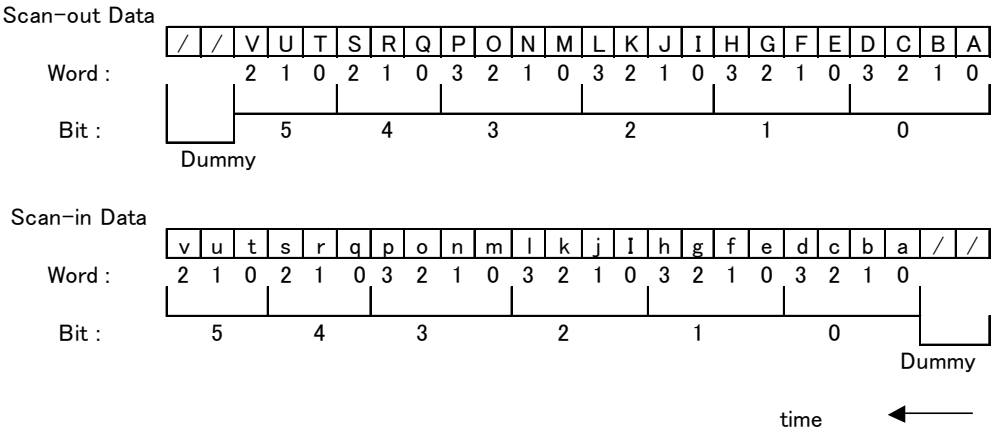


図 4.4　ワード数の異なるメモリアレイの直列接続のデータ

例えば図 4.3 に示すような、4 ワード×4 ビットのレジスタファイル A と、3 ワード×2 ビットのレジスタファイル B が直列接続されている場合を考える。この状態からスキャン動作に入った場合、全ビットをシフト出力するには 22 クロックを供給すればよい。このときシフト出力データの時系列は所望のビット配列になる。しかし、22 番目のクロックが供給された後、レジスタファイル A の SAC は第 2 番地を、またレジスタファイル B の SAC は第 1 番地を指定しているため、シフト入力データの時系列とワードの昇順が一致しない。このため、更に 2 クロック、すなわち合計 24 クロックを供給して、各 SAC が初期状態と同じく第 0 番地を指定するようにする必要がある。この 24 は、各レジスタファイルのワード数 3 と 4 の最小公倍数 12 の整数倍に対応する。24 ビットのスキャン入力データのうちレジスタファイルに残るのは 22 ビットであるから、先頭の 2 ビットはダミービットとなる。スキャン出力は、LSI テスタのスキャン動作時間を短縮するため、次のテストパターンのスキャン入力と同時に実行されるので、スキャン出力ビットも 24 ビットでなければならない。従がって、スキャン出力データについては最後の 2 ビットを無視する。この条件を満たした完全なスキャンデータを図 4.4 に示す。

以上のことから、スキャンパスに関して複数のメモリアレイを直列接続した場合、スキャンパスの総ビット数が、その中に含まれる各メモリアレイのワード数の最小公倍数の整数倍になるように調整すれば、メモリアレイの接続位置に関係なく、ビットおよびワードの昇順に配列されたスキャンデータを生成できる。メモリアレイのワード数は 2 のべき乗になっていることが多いので、通常、この最小公倍数は最も大きなメモリアレイのワード数で済む。

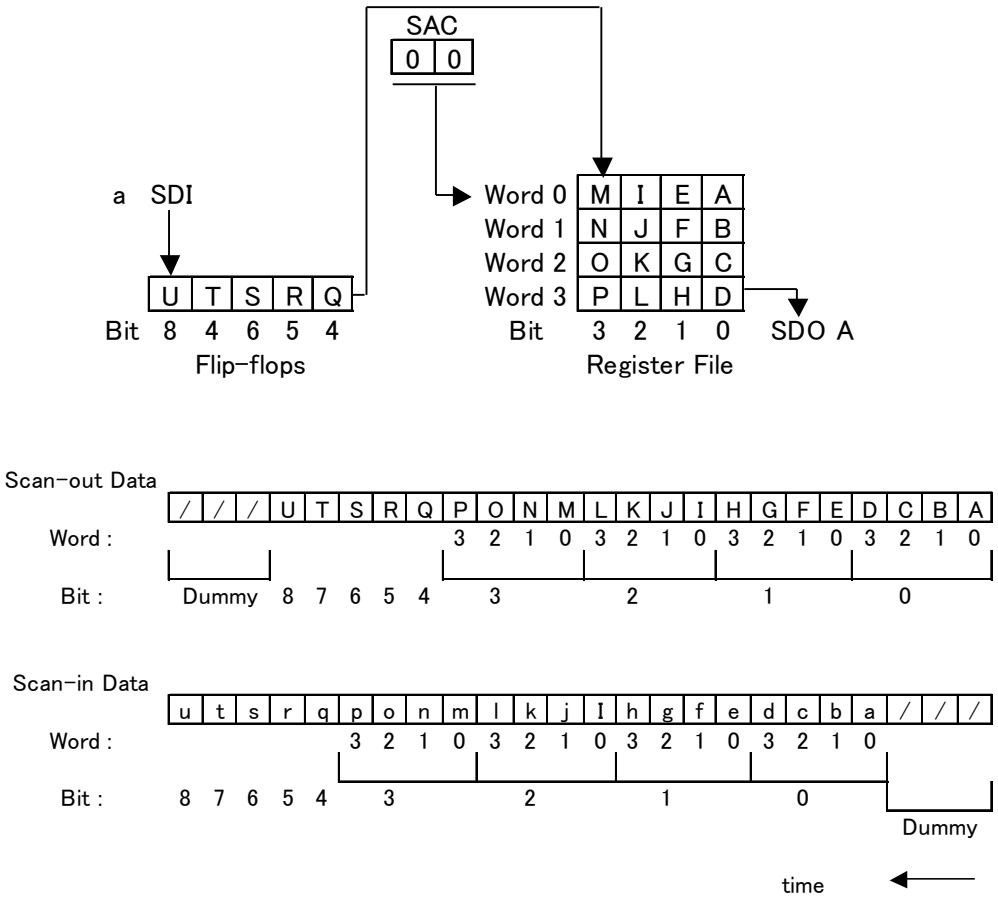


図 4.5　FF 回路とメモリアレイの直列接続とそのスキャンデータ

4.4.2　メモリアレイと FF 回路の直列接続

メモリアレイと FF 回路を直列接続した場合は、前節で説明したワード数の異なる複数のメモリアレイを直列接続した場合の特殊ケースと考えられる。即ち、FF 回路を 1 ワードのメモリアレイとみなしてスキャンデータを生成すればよい。これによって、メモリアレイと FF 回路を 1 つのスキャンパスに組み込むことが可能となり、スキャンパス方式をメモリアレイを含む順序回路全体に適用できるようになる。

簡単な例を図 4.5 に示す。すなわち 5 ビットの FF 回路と、4 ワード×4 ビットのレジスタファイルが 1 つのスキャンパスに組み込まれている場合である。FF 回路を 1 ワード×5 ビットのレジスタファイルとみなす。全ビットをシフト出力するためには 21 クロックを与えればよいが、完全なスキャンデータを得るためには、前節で述べたことから各レジスタファイルのワード数(この場合は FF 回路のワー

ド数 1 とレジスタファイルのワード数 4)の最小公倍数の整数倍である 24 クロックを供給しなければならない。従がって、図 4.5 に示すようなスキャンデータでシフト動作を行えば所望の最終状態が得られる。以上によって、スキャンパス方式の適用領域をメモリアレイを含む順序回路全体に拡張できるこがわかる。

多数のメモリアレイを集積化している LSI では、キャッシュメモリのような大容量のメモリアレイを除いても、全体のビット数が数千ビットないし数万ビットに及ぶことも考えられる。これらの全ビットを 1 つのスキャンパスに組み込むと、スキャン動作に必要な LSI テスタのテスト時間が非常に長くなるので、長いスキャンパスを複数のスキャンパスに分割する。このときには、各スキャンパスの長さをできるだけ等しくするように分割することが望ましい。メモリアレイは分割できないが、FF 回路は 1 ビット単位まで分割できるので複数のスキャンパスへ有効に分散する。

4.5　新スキャンパス方式の特長

スキャンパス方式の適用をメモリアレイの領域にまで拡張したことにより、以下のような特長が生まれる。

まず第 1 の特長は、このスキャンパス方式ではメモリセル、アドレスデコーダなどのメモリ回路も通常の論理回路と同一に扱って縮退故障検出の対象にできることである。論理設計者は、スキャン状態においてメモリセルがシフトレジスタとして動作することだけを認識し、通常のフリップフロップ回路と同様、メモリアレイのスキャンデータ入力、スキャンデータ出力、およびスキャンモード制御信号の 3 本を配線すればよい。スキャンパス方式導入するためにメモリアレイを付加した回路は、試験のためだけの回路であって、論理設計者には見えない。このスキャンパス方式により、メモリアレイ部を含めた順序回路全体の試験パターンを ATPG を用いて短時間に自動生成できるという利点が生じる。例えば[A]*[B]→[C]の演算回路において、[A]、[B]がメモリアレイの直接出力である場合、または[C]が直接入力である場合、メモリアレイのアドレスを指定するフリップフロップ回路、データを保持するメモリアレイ、およびそのデータを使用する演算回路を区別することなく、スキャンパスを前提とした ATPG で対応できる。すなわち本提案のスキャンパスを介してフリップフロップ回路とメモリアレイの内容を設定し、通常動作後、演算結果を格納しているメモリアレイとフリップフロップ回路の内容をスキャン出力して期待値と比較するという一連のテストパターンを自動生成できる。

第 2 の特長は、このスキャンパス方式ではメモリアレイ内の故障解析が容易になることである。メモリアレイ部の故障はスキャン出力での周期的なビット不良を引き起こすことが多く、通常のランダムロジック部の故障と判別しやすいからである。例えば図 4.6 に示すように、メモリセルのワード 1 のビット 1 が縮退故障だった場合(x で示されている)、そのスキャン出力データには周期的に x が出力される。

更に、スキャンパス用に選定したポートのアドレスデコーダ部の故障については、その故障モードに従がって特長のあるスキャン出力データが得られる。例えば、読み出しアドレスデコーダの入力段における縮退故障の場合、スキャン出力データにはエラーを含まないビット列と、同ビット数のエラーを含むビット列が交互に出現する。このビット列の長さが 4 ビットの場合には最下位より 3 ビット目、8 ビットの場合には 4 ビット目の縮退故障は、連続する 8 ビットの 1 と連続する 8 ビットの 0 を交互に含むようなテストパターンを用い、スキャン出力データにおいて連続する 8 ビットの不良が発生することで確認できる。また書き込みアドレスデコーダにおきる同様の縮退故障は、スキャン乳油力パターンに関係なく周期的に同一のスキャン出力パターンが得られることから推定できる(但し故障ゲートの特定まではできない)。

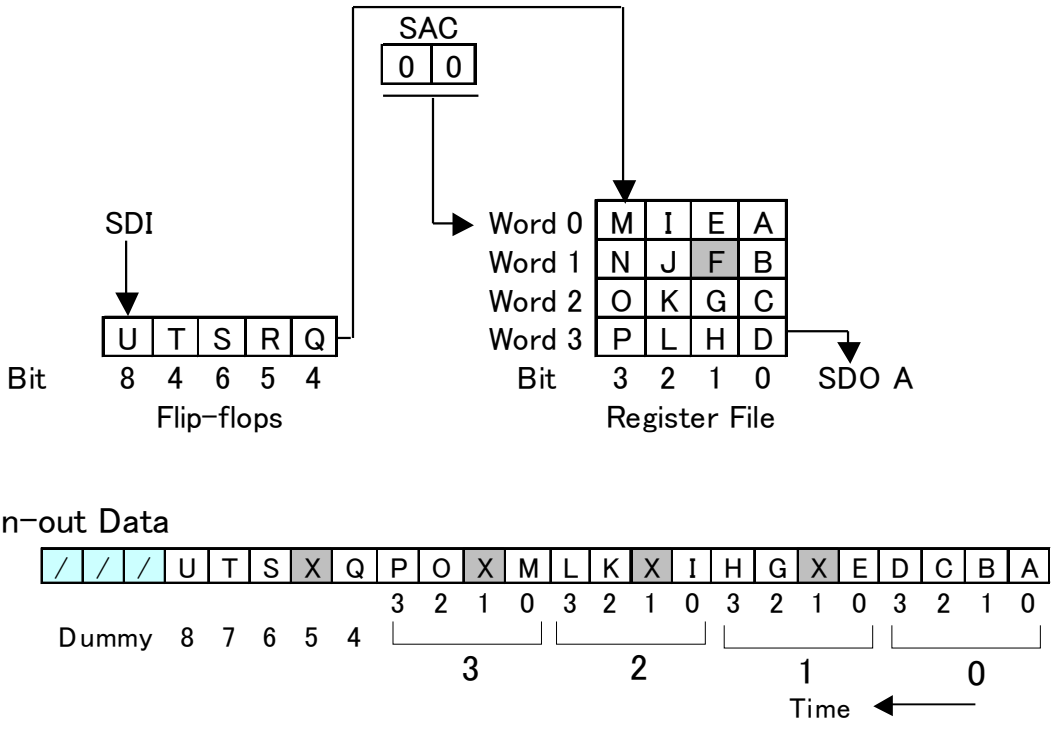


図 4.6　メモリセルの縮退故障とスキャンデータ

第 3 の特長は、提案のスキャンパス方式ではスキャン動作自身がメモリアレイの

テストになっていることである。スキャン動作時には、クロックごとにメモリアレイに対する読み出し及び書き込み動作が行われ、スキャンデータが全てメモリアレイを通過して入出力されるからである。例えばスキャンデータの長さが 1000 ビットとすると、メモリアレイの読み出し・書き込み動作を 1000 回行った後で 1 回の通常動作が行われるので、LSI テスタの占有時間のうち、99.9%以上がメモリアレイに対する読み出し・書き込み動作時間だということになる。これに対してバウンダリスキャンを用いたメモリアレイのテストでは、スキャン動作中はメモリアレイへのアクセスが禁止されており、テスト時間の大部分はスキャン動作のみのために費やされる。従がって、限られた試験時間内で見るとメモリアレイへのアクセス回数が非常に少なく、一方アクセス回数を増加させると試験時間が非常に長くなる。

以上のことから、このスキャンパス方式によるテストは、図 4.6 に示すようにすべての組み合わせ回路を周辺回路とするような、縮退故障検出試験であると見なすことができる．このスキャンパス方式は、メモリアレイ特有の結合故障、パターン依存故障の検出についても有効であることが示されている[8]。

第 5 章

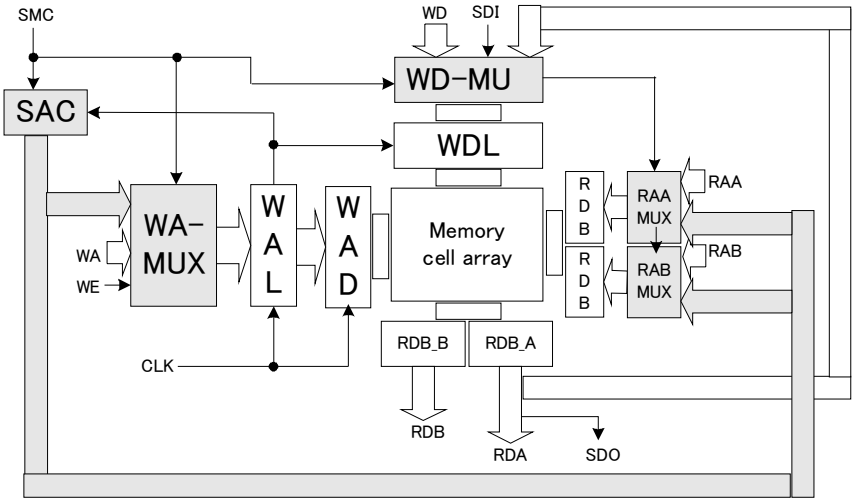
マルチポートメモリアレイへのスキャンパス構成適用

マイクロプロセッサの中枢にあるレジスタファイルは、演算の高速化を図るため通常マルチポートメモリ構成となっている。大部分のマルチポートメモリの試験では、シングルポートメモリ用のテスト方法をポート毎に行ってきた。すなわちマルチポートメモリは一般的に「複合的なシングルポートメモリアレイ」としてテストされてきた。この方法での主な問題点は、マルチポート固有の故障検出が保証されていないことである。このポート間にまたがるブリッジ故障の発生確率は決して低くはない。本章では、第 4 章で述べたメモリアレイのスキャンパス構成が、マルチポートメモリアレイ固有の故障検出に対しても有効であることを示す[9]、[10]。

5.1　対象回路

第 4 章ではレジスタファイルのようなロジック部分のメモリアレイをスキャン可能とし、そのメモリアレイを通常のフリップフロップ回路と同一のスキャンパスへ組み込む方法について説明したが、第 5 章ではその方法をマルチポートポートメモリアレイに適用した場合、従来のテストでは検出できなかったマルチポート特有の故障を検出できることを示す。

対象とする回路は図 5.1 に示すような、書き込み用のポートが 1 つと、読み出し用のポートを 2 つ備えたマルチポートメモリアレイである。



WA: Write-Address
WE: Write-Enable
WD: Write-Data
SDI: Scan-Data Input
RAA: Read-Address-A
RAB: Read-Address-B
RAD: Read-Address decoder
RDA: Read-Data-A
RDB: Read-Data-B
SDO: Scan-Data-Output
SMC: Scan-Mode-Control
CLK: Clock
SAC: Scan-address counter
RDB: Read-Data Buffer
WD-MUX: Write-Data MUX
WA-MUX: Write-Address MUX
RAA-MUX: Read-Address-A MUX
RAB-MUX: Read-Address-B MUX
WDL: Write-Data latch
WAD: Write-Address Decoder

図 5.1　スキャンパス構成メモリアレイのマルチポート化

5.2　マルチポートメモリアレイの故障の定義

ここでは、ポート間ブリッジ故障について述べる。即ちビット線、あるいは、ワード線が短絡してしまうブリッジ故障である。他の故障モデル、すなわち縮退故障、セル結合故障、パターン依存故障は第４章で述べたスキャンテストで検出されるため割愛する。

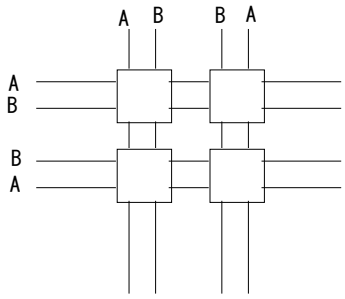


図 5.2　ミラーイメージのレイアウト

5.2.1　ビット線ショート

一般的に、ビット線のポート間故障とはどれもブリッジ故障である。すなわち、あるポートと交錯する他のポートとの短絡である。実際に、物理的に十分に分離されているビット線は短絡しないので、考えられる短絡は同コラムまたは隣接コラムでのビット線間短絡である。

図 5.3 では 2 ポートメモリの異なるレイアウトスタイルのポート間ビット線ショートの例を幾つか示す。(a)は同、近傍の両ビット線の交錯ショートが考えられる。(b)はミラーイメージのレイアウトで、その場合、同コラムのみのショートを考えればよい。ここでは両方について述べる。

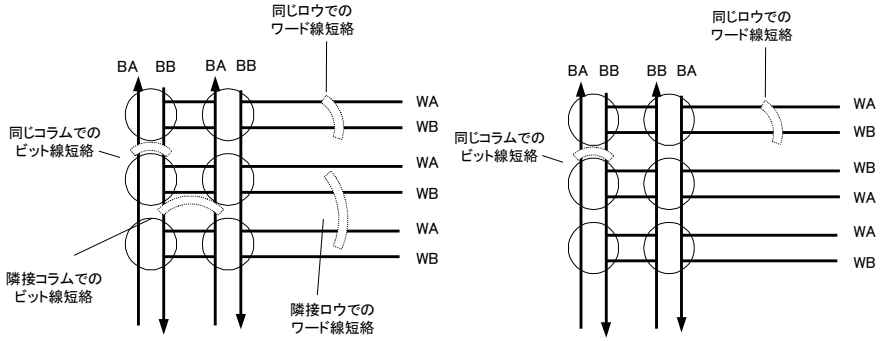


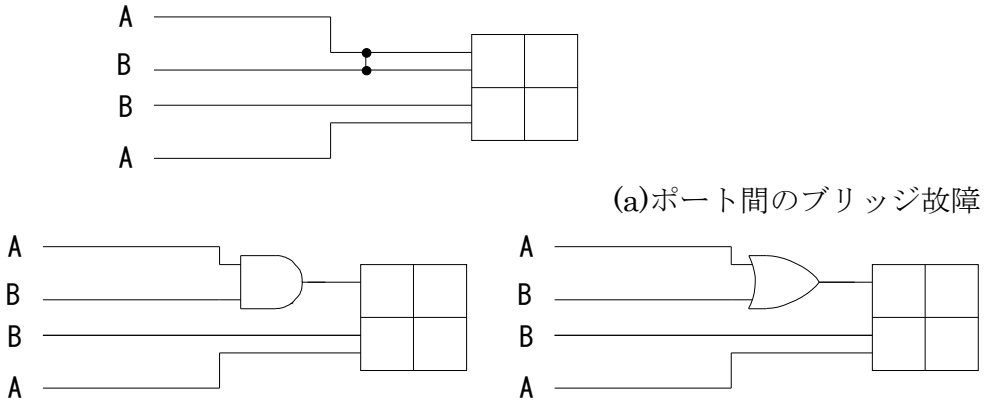
図 5.3　ビット線短絡とワード線短絡の例

5.2.2　ワード線ショート

ビット線と同様、もっとも起こり得るポート間ワード線短絡は同一または隣接ロウのブリッジである。ワード線ショートを、大きく 2 種類に分類し、それぞれ WSA と WSD と定義する。定義を以下に述べる。

WSA(Word-line Short Asserts both word lines) : 2 つの Read ポートのワード線が短絡し一方が選択された場合、両方のワード線が選択されてしまうワード線の短絡故障である。両方選択されたデータはワイアド AND、ワイアド OR される。

WSD(Word-line Short De-asserts both word lines) : 2 つの Read ポートのワード線が短絡し、一方が選択されなかった場合、両方のワード線とも非選択となる短絡故障である。両方とも非選択となったポートからの読み出しデータは 1 または 0 に縮退してしまうと考えられる。



(a)ポート間のブリッジ故障

(b)ワイアド AND

(c)ワイアド OR

図 5.4　WSA 故障の例

5.3 故障の検出方法

対象とする回路は、図 5.1 に示す回路である。この回路は第 4 章の図 4.3 の回路に読み出しポートを新たにもう 1 つ備えた、1 つの書き込みポートと、2 つの読み出しポートを備えたマルチポートメモリである。

テストモードでは読み出しポート A と書き込みポートでスキャンパスを構成する。読み出しポート B は、読み出しポート A を選択したアドレス、即ち SAC でカウントされた数の下位 2 ビット目を反転させたアドレスにアクセスする。例えば読み出しポート A がワード線 xx101 にアクセスする時、読み出しポート B は下位 2 ビットを「0」→「1」に反転させワード線 xx111 にアクセスする[11]。

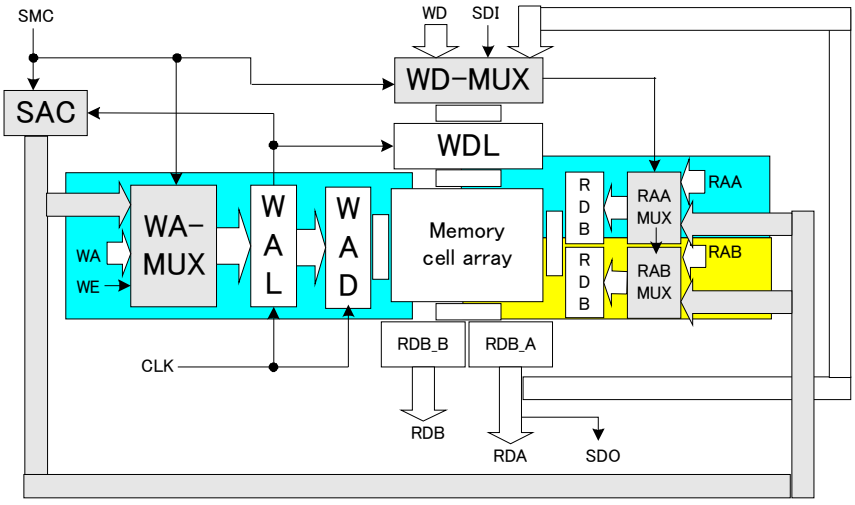


図 5.5　マルチポートメモリのポート間故障検出のしくみ

ポートＢからの擬似読み出しは次のアドレスを与えることによって行う。

ReadPortBのアドレス

$$= \text{ReadPortA}\text{のアドレス} + (-1)^{\left\lfloor \frac{\text{ReadPortA}\text{のアドレス}}{2} \right\rfloor} \times 2 \qquad (5.1)$$

で表される。

5.4　読み出しポート間ワード線短絡故障の検出例

ここでは 4 ワード×2 ビットのメモリアレイを例にして、読み出しポート間のワード線故障(WSA、WSD)について述べていく。説明や図 5.6 の「1」や「0」は、メモリセルのデータの論理値である。「＊」はドントケアを表す。期待値とスキャンアウトされたデータが不一致の場合、読み出しポートのワード線間の故障として検知される。

5.4.1　WSA 検出例(ワイアド OR の場合)

読み出しポート A と B のワード 00 に WSA 故障が存在する場合の検出方法について説明する。読み出しポート A がワード 10 を読み出しているとき、読み出しポート B はワード 00 にアクセスしているため、リードポート A ではワード 00 も選択される。従って、リードポート A ではワード線を複数選択することになる。ワード 00 のデータとワード 10 のデータが OR されて出力されるとすると、この故障はワード 10 に「0」をスキャンインし、ワード 00 に「1」をスキャンインすれば検出できる。

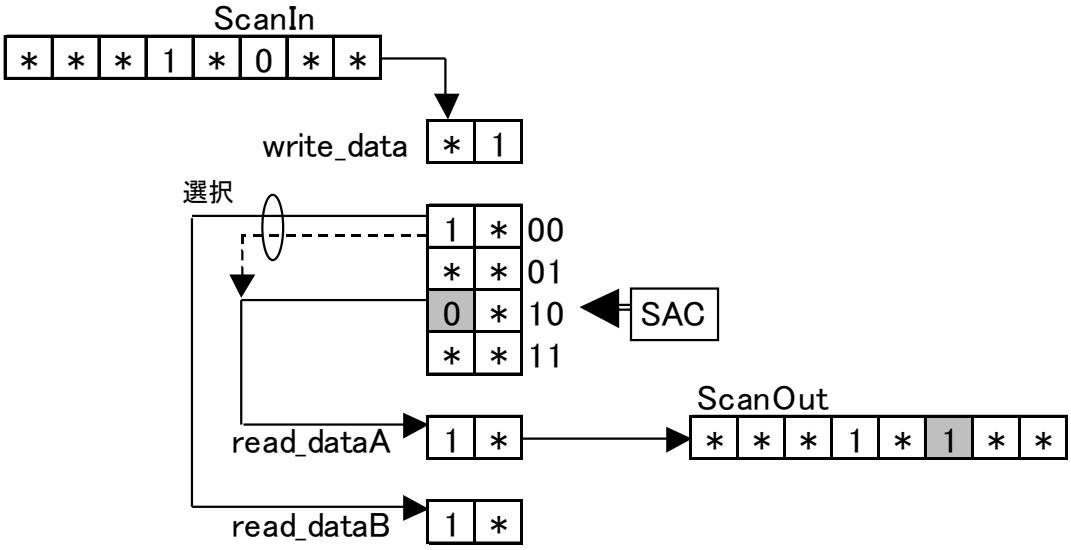


図 5.6　WSA 検出例(a)

5.4.2　WSA 検出例(ワイアド AND の場合)

同様に、読み出しポート A と B のワード 00 に WSA 故障が存在し複数選択時のリードデータが AND で出力される場合について述べる。読み出しポート A はワード 10 を読みだしているとき、読み出しポート B はワード 00 にアクセスしているので、ワード 00 のデータとワード 10 のデータが AND されて出力される。従ってワード 10 に「1」、ワード 00 に「0」をスキャンインするとこの故障を検出できる。

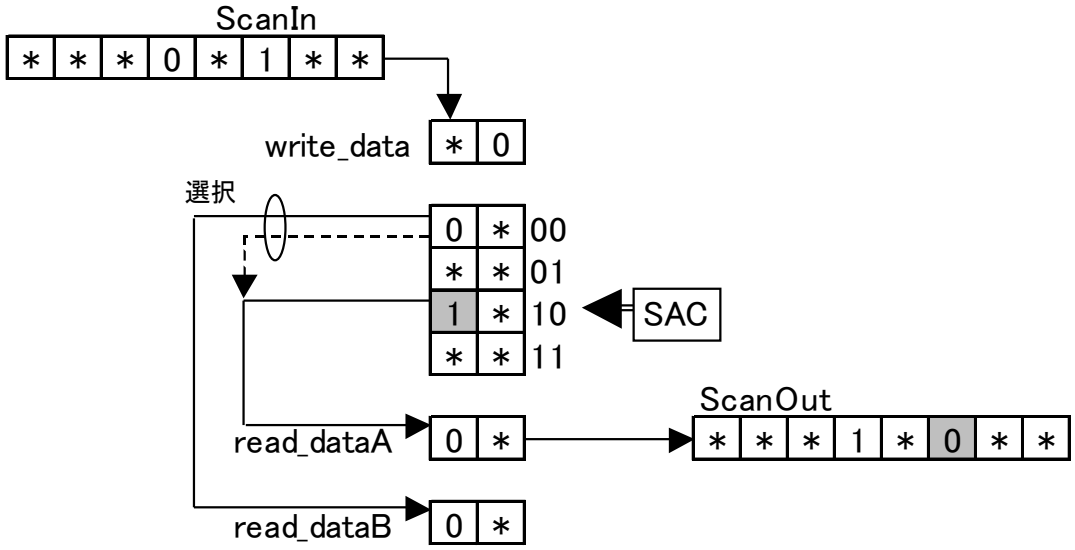


図 5.7　WSA 検出例(b)

5.4.3　WSD 検出例（非選択時の出力データ 0 の場合）

読み出しポート A と B のワード線 00 に WSD 故障が存在する場合について説明する。この場合、読み出しポート A がワード 00 をアクセスしても、読み出しポート B がワード 10 にアクセスしているため読み出しポート A のワード 00 が非選択となる。従って読み出しポート A は全アドレスが非選択となる。この時読み出しデータが「0」になるとするとワード 00 に「1」をスキャンインしても「0」と読み出されるため、この故障を検出できる。

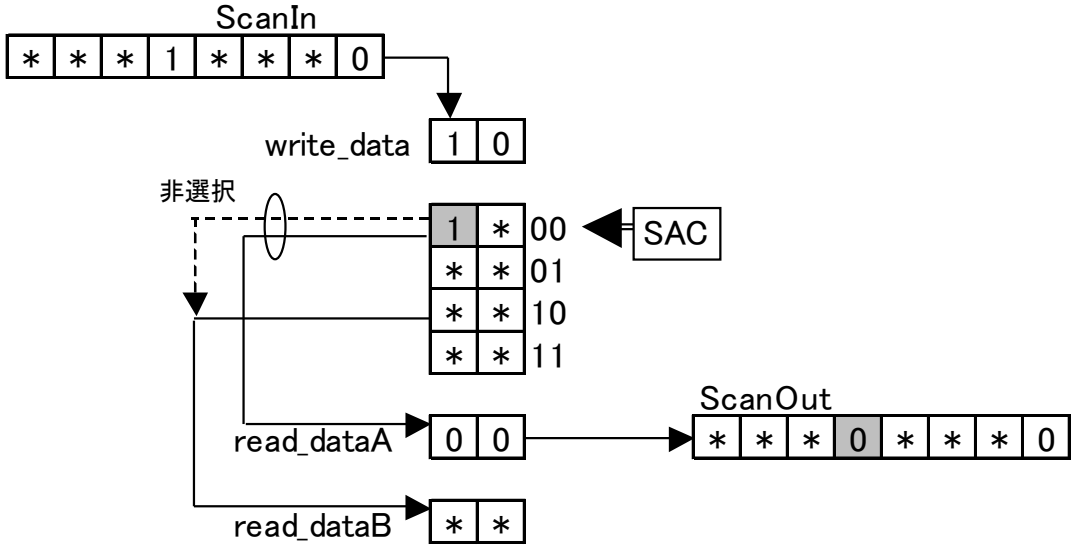


図 5.8　WSD 検出例(a)

5.4.4　WSD 検出の例（非選択時の出力データ 1 の場合）

読みだしポート A と B のワード 00 に WSD 故障が存在する場合で、短絡ワード線が非選択の時の読み出しデータが「1」になる場合、ワード 00 に「0」をスキャンインすれば、この故障は検出できる。

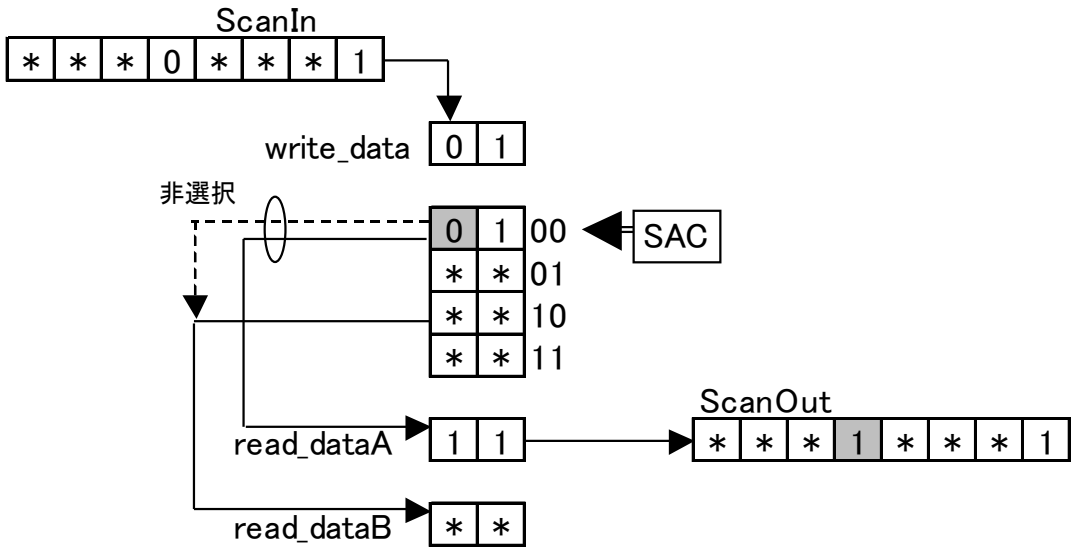


図 5.9　WSD 検出例(b)

5.5　テストパターン

WSA 故障のテストパターンは、擬似読み出しが、ショートした箇所にアクセスした場合に故障が検出される。前節での読み出しポート A、B のワード 00、01、10、11 に WSA 故障が存在する場合の故障検出パターンを求めると図 5.10 になる。①と③が WSA でワイアド OR された場合、一方、②と④がワイアド AND された場合のテストパターンである。ワイアド OR された故障の検出の場合、読み出されるデータが「0」で、擬似読み出しされるデータの値が「1」の場合に故障は検出される。ワイアド AND された故障の検出はその逆で、読み出しデータが 1 で、擬似読み出しされるデータが 0 の場合に故障は検出される。

従ってこれら全ての故障を検出するテストパターンは、図 5.10 の⑤の 12 ビット

のテストパターンとなる。WSD 故障の検出は「1」に縮退しているセルには「0」の入力を、「0」に縮退しているセルには「1」の論理値のパターンを入力すれば検出することができる。つまり、この⑤のテストパターンは WSD の検出条件も満たしている。従ってこの 12 ビットのテストパターンによって、全ワードの WSA、WSD を検出できる。

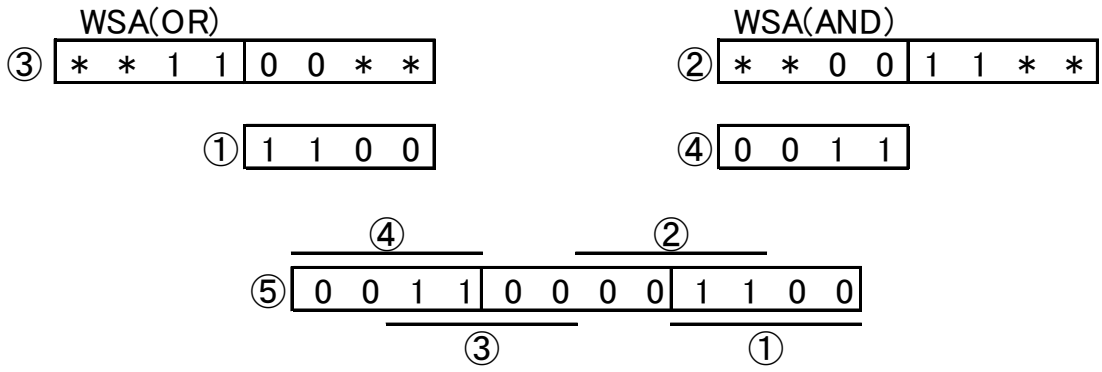


図 5.10　故障検出テストパターン

ビット線のショートの場合は、読み出しポート A と読み出しポート B が異なるワード線を指す限り、ショートしているビット線の出力は競合する。その出力はワード線同様にワイアド OR、またはワイアド AND されて出力される。読み出しポート A と読み出しポート B が指定するワード線のメモリセルのデータに「1」「0」と「0」「1」のパターンがあればビット線のショートは検出することができる。ワイアド AND されたビット線ショートの場合、読み出しポート A が「1」を読み出した場合、読み出しポート B は「0」を選択し、AND され読み出しポート A の値は「0」になり、故障は検出される。ワイアド OR の場合は逆に「1」が出力されてしまい、故障は検出される。

第6章

おわりに

本論文では、ランダムロジックのテストとレジスタファイルのようなメモリアレイを統一的に扱うスキャン方式について説明した後、この方式をマルチポートメモリアレイに適用し、スキャン動作によってマルチポート固有の故障が検出できることを示した。具体的には、1Write ポートと 2Read ポートを有するマルチポートメモリアレイを対象とし、マルチポートメモリアレイ固有の故障として 2 つのワード線短絡故障を想定し、スキャン可能メモリ構成でマルチポートメモリアレイ固有のポート間ワード線短絡故障を有効に検出できることを示した。

今後の課題は、他のマルチポートメモリアレイ特有の故障検出、スキャンパス構成の適用によるハードウェアオーバヘッドと性能への影響の評価、さらにポート数が増加した場合への対応などである。

謝辞

高知工科大学大学院工学研究科基礎工学専攻情報システムコース長の島村和典教授、指導教員の矢野政顕教授をはじめ諸先生方からご指導、ご助言をいただくことによって本研究を進めることができました。ここに深く感謝いたします。

また研究を進める過程で、電子・光エレクトロニクスコース長の原 央教授をはじめ電子･光システム工学科の諸先生､学生の皆さんから大変お世話になりました。心からお礼申し上げます。

参考文献

[1] Parag k .Lala “Digital Circuit Testeng and Testability”, pp.2-19、1997 by Academic Press

[2] 藤原秀雄 “コンピュータの設計とテスト”, pp.137-145、 平成 2 年　工学図書株式会社

[3] Rochit Rajsuman, “System-on-a-Chip Design and Test”, pp.155-179、

2000 by Artech House Publishers

[4] Parag k .Lala “Digital Circuit Testeng and Testability” pp.20-38、 1997 by Academic Press

[5] Revised Printing “Digital Systems Testing And Testable Design”, pp.343-351、IEEE PRES

[6] 佐藤康夫、中尾教伸 “設計者に必要な BIST 技術の基礎知識” pp.55-67、Design Wave Magazine, Marh.2001, CQ 出版

[7] 矢野政顕、石浦菜岐佐 “メモリアレイを含む順序回路へのスキャンパス方式適用” vol. J79-D-I no.12, pp.1055-1062 (1996)

[8] S. Yano and N. Ishiura: “Application of scan path approach to sequential circuits including memory arrays” (in Japanese), IEICE Trans. Inf. & Syst., vol. J79-D-I, no. 12, pp. 1055-1062, Dec. 1996

[9] 坂下雄一、木村知史、矢野政顕 “スキャンパス構成を利用したマルチポートメモリアレイの試験”, p.149, 平成 13 年度電気関係学会四国支部連合大会

[10] 木村知史、坂下雄一、矢野政顕, “スキャンパス構成を利用したマルチポートメ

モリアレイの試験", FTC 研究会、2002 年 1 月

[11] Yuejian Wu, Sanjay Gupta, "Built-In Self-Test for Multi-Port RAMs" pp.398-403, ATS'97. (1997)

付録

A.1 スキャンパス構成マルチポートメモリアレイの VHDL 記述

第 5 章での対象回路であるスキャンパス構成を適用させたマルチポートメモリアレイの VHDL 記述を掲載する。

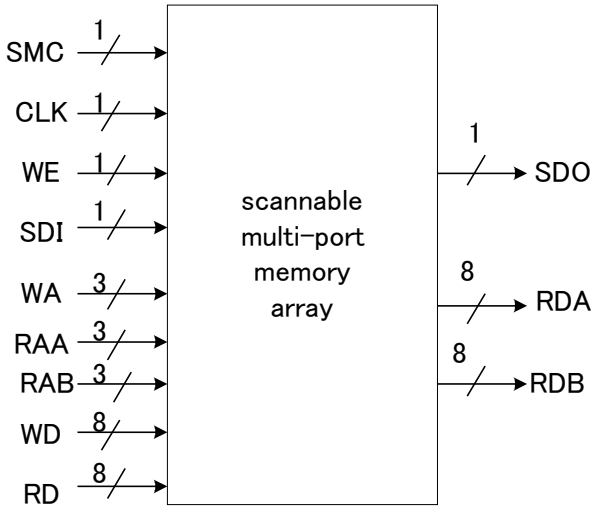


図 A.1　マルチポートアレイの最上位図

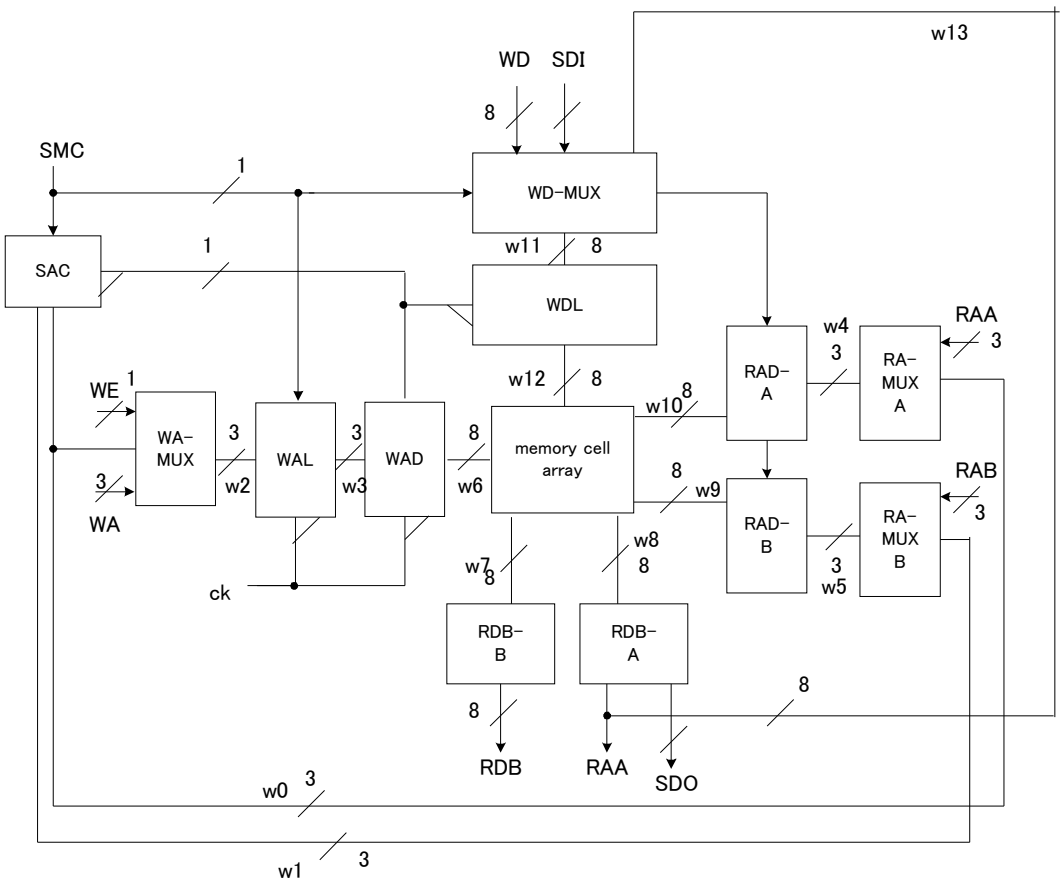


図 A.2　スキャンパス構成マルチポートメモリアレイ

A.1.1 Memory-Array

```
library ieee;
use ieee.std_logic_1164.all;

entity MEMORY is
    port(ra_a, ra_b, wa, wd : in std_logic_vector(7 downto 0);
                rd_a, rd_b: out std_logic_vector(7 downto 0));
end MEMORY;

architecture BEHAVIOR of MEMORY is
    component reg
            port(wa : in std_logic;
                 d: in std_logic_vector(7 downto 0);
                 q   : out std_logic_vector(7 downto 0));
    end component;

    component demux
            port(   wd, wa   : in std_logic_vector(7 downto 0);
            reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7
                 : out std_logic_vector(7 downto 0));
    end component;

    component mux
            port(reg0, reg1, reg2, reg3, reg4, reg5, reg6, reg7
                 : in std_logic_vector(7 downto 0);
                 ra_a, ra_b : in std_logic_vector(7 downto 0);
                 rd_a, rd_b : out std_logic_vector(7 downto 0));
            end component;
signal w0, w1, w2, w3, w4, w5, w6, w7,
        w8, w9, w10, w11, w12, w13, w14, w15 :
        std_logic_vector(7 downto 0);
begin
demux0 : demux port map(
                 wd=>wd, wa=>wa, reg7=>w0, reg6=>w1, reg5=>w2,
                 reg4=>w3, reg3=>w4, reg2=>w5, reg1=>w6,
                 reg0=>w7);
mux0: mux port map(
            reg7=>w8, reg6=>w9, reg5=>w10, reg4=>w11, reg3=>w12,
            reg2=>w13, reg1=>w14, reg0=>w15, ra_a=>ra_a,ra_b=>ra_b, rd_a=>rd_a, rd_b=>rd_b);

reg0 : reg port map(
            wa=>wa(0), d=>w7, q=>w15);
reg1 : reg port map(
            wa=>wa(1), d=>w6, q=>w14);
reg2 : reg port map(
            wa=>wa(2), d=>w5, q=>w13);
reg3 : reg port map(
            wa=>wa(3), d=>w4, q=>w12);
reg4 : reg port map(
            wa=>wa(4), d=>w3, q=>w11);
reg5 : reg port map(
            wa=>wa(5), d=>w2, q=>w10);
reg6 : reg port map(
            wa=>wa(6), d=>w1, q=>w9);
reg7 : reg port map(
            wa=>wa(7), d=>w0, q=>w8);
end BEHAVIOR;
```

A.1.2 RAD(Read-Address-Decoder)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_RAD_A is
    port(
        din : in std_logic_vector(2 downto 0);
        dout: out std_logic_vector(7 downto 0));
end M_RAD_A;

architecture BEHAVIOR of M_RAD_A is
begin
    process(din)begin
        case din is
            when "000" => dout <= "00000001";
            when "001" => dout <= "00000010";
            when "010" => dout <= "00000100";
            when "011" => dout <= "00001000";
            when "100" => dout <= "00010000";
            when "101" => dout <= "00100000";
            when "110" => dout <= "01000000";
            when "111" => dout <= "10000000";
            when others => dout <= "XXXXXXXX";
        end case;
    end process;
end BEHAVIOR;
```

A.1.3 RA-MUX(Read-Address Multiplexer)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_RAD_A is
    port(
        din : in std_logic_vector(2 downto 0);
        dout: out std_logic_vector(7 downto 0));
end M_RAD_A;

architecture BEHAVIOR of M_RAD_A is
begin
    process(din)begin
        case din is
            when "000" => dout <= "00000001";
            when "001" => dout <= "00000010";
            when "010" => dout <= "00000100";
            when "011" => dout <= "00001000";
            when "100" => dout <= "00010000";
            when "101" => dout <= "00100000";
            when "110" => dout <= "01000000";
            when "111" => dout <= "10000000";
            when others => dout <= "XXXXXXXX";
        end case;
    end process;
end BEHAVIOR;
```

A.1.4 RDB_A(Read-Data-Buffer A)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_RDB_A is
    port(
        din : in std_logic_vector(7 downto 0);
        sdo : out std_logic;
        dout1, dout2: out std_logic_vector(7 downto 0));
end M_RDB_A;

architecture BEHAVIOR of M_RDB_A is
begin
    process(din)begin
        dout1 <= din;
        dout2 <= din;
        sdo <= din(0);
    end process;
end BEHAVIOR;
```

A.1.5 RDB_B(Read-Data-Buffer B)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_RDB_B is
    port(
        din : in std_logic_vector(7 downto 0);
        dout: out std_logic_vector(7 downto 0));
end M_RDB_B;

architecture BEHAVIOR of M_RDB_B is
begin
    process(din)begin
        dout <= din;
    end process;
end BEHAVIOR;
```

A.1.6 SAC(Scan-Address-Counter)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity M_SAC is
    port(clk, smc : in std_logic;
         q1, q2 : out std_logic_vector(2 downto 0));
end M_SAC;

architecture BEHAVIOR  of M_SAC is
    signal d :std_logic_vector(2 downto 0);
begin
    q1 <= d;
    q2(0) <= d(0);
    q2(1) <= not d(1);
  q2(2) <= d(2);
    process(clk, smc)
    begin
    if (smc = '1')then
     if(clk'event and clk = '1')then
        d <= d+1;
     end if;
    end if;
    end process;
end BEHAVIOR ;
```

A.1.7 WAD(Word-Adress-Decoder)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_WAD is
    port(
        din : in std_logic_vector(2 downto 0);
        clk : in std_logic;
        dout: out std_logic_vector(7 downto 0));
end M_WAD;

architecture BEHAVIOR of M_WAD is
    begin
        process(din, clk)begin
            if(clk'event and clk = '1')then
             case din is
                when "000" => dout <= "00000001";
                when "001" => dout <= "00000010";
                when "010" => dout <= "00000100";
                when "011" => dout <= "00001000";
                when "100" => dout <= "00010000";
                when "101" => dout <= "00100000";
                when "110" => dout <= "01000000";
                when "111" => dout <= "10000000";
                when others => dout <= "XXXXXXXX";
            end case;
        end if;
    end process;
end BEHAVIOR;
```

A.1.8 WAL(Word-Address-Latch)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_WAL is
    port(
        din : in std_logic_vector(2 downto 0);
        clk : in std_logic;
        dout : out std_logic_vector(2 downto 0));
end M_WAL;

architecture BEHAVIOR of M_WAL is
    signal reg_q : std_logic_vector(2 downto 0);
    begin
        dout <= reg_q;
        process(din, clk)
        begin
            if(clk = '1')then
                reg_q <= din;
            end if;
        end process;
end BEHAVIOR;
```

A.1.9 WA-MUX(Word-Address Multiplexer)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_WAM is
    port(
        wa, sac : in std_logic_vector(2 downto 0);
        smc, we : in std_logic;
        dout: out std_logic_vector(2 downto 0));
end M_WAM;

architecture BEHAVIOR of M_WAM is
begin
    process(wa, sac, smc, we) begin
        if (smc = '1')then
            dout <= sac;
        elsif(we = '1')then
            dout <= wa;
        end if;
    end process;
end BEHAVIOR;
```

A.1.10 WDL(Word-Data-Latch)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_WDL is
    port(
        din : in std_logic_vector(7 downto 0);
        clk : in std_logic;
        dout: out std_logic_vector(7 downto 0));
end M_WDL;

architecture BEHAVIOR of M_WDL is
    signal reg_q: std_logic_vector(7 downto 0);
    begin
        dout <= reg_q;
        process(din, clk)
        begin
            if(clk = '1')then
                reg_q <= din;
            end if;
        end process;
end BEHAVIOR;
```

A.1.11 WD-MUX(Word-Data Multiplexer)

```
library ieee;
use ieee.std_logic_1164.all;

entity M_WDM is
    port(
        wd, rd  : in std_logic_vector(7 downto 0);
        smc, sdi: in std_logic;
        dout: out std_logic_vector(7 downto 0));
end M_WDM;

architecture BEHAVIOR of M_WDM is
begin
    process(wd, rd, smc, sdi, dout)begin
        if (smc = '0')then
            dout <= wd;
        else
            dout(7) <= sdi;
            dout(6) <= rd(7);
            dout(5) <= rd(6);
            dout(4) <= rd(5);
            dout(3) <= rd(4);
            dout(2) <= rd(3);
            dout(1) <= rd(2);
            dout(0) <= rd(1);
        end if;
    end process;
end BEHAVIOR;
```

A.1.12 Scanable Multi-port-Memory-Array

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity M_portM is
    port(
        wa, ra_a, ra_b   : in std_logic_vector(2 downto 0);
      wd, rd : in std_logic_vector(7 downto 0);
        clk, we, sdi, smc    : in std_logic;
        sdo : out std_logic;
        rdout_a, rdout_b :out std_logic_vector(7 downto 0));
end M_portM;

architecture arch of M_portM is
    component M_sac
        port(
            clk, smc: in std_logic;
                q1, q2   : out std_logic_vector(2 downto 0));
    end component;
    component M_wdl
        port(
        din : in std_logic_vector(7 downto 0);
        clk     : in std_logic;
        dout: out std_logic_vector(7 downto 0));
    end component;
    component M_ram_a
        port(
            ra_a, sac    : in std_logic_vector(2 downto 0);
            smc : in std_logic;
            dout_a   : out std_logic_vector(2 downto 0));
    end component;
    component M_ram_b
        port(
            ra_b, sac    : in std_logic_vector(2 downto 0);
            smc : in std_logic;
            dout_b   : out std_logic_vector(2 downto 0));
    end component;
    component M_rad_a
        port(
            din : in std_logic_vector(2 downto 0);
            dout: out std_logic_vector(7 downto 0));
    end component;
    component M_rad_b
        port(
            din : in std_logic_vector(2 downto 0);
            dout: out std_logic_vector(7 downto 0));
    end component;
    component M_wam
        port(
            wa, sac : in std_logic_vector(2 downto 0);
            smc, we : in std_logic;
            dout: out std_logic_vector(2 downto 0));
    end component;
    component M_wad
        port(
            din : in std_logic_vector(2 downto 0);
            clk : in std_logic;
            dout: out std_logic_vector(7 downto 0));
    end component;
    component M_wdm
        port(
            wd, rd   : in std_logic_vector(7 downto 0);
            smc, sdi: in std_logic;
            dout: out std_logic_vector(7 downto 0));
    end component;
    component M_mem
        port(
```

```
            ra_a, ra_b, wa, wd   : in std_logic_vector(7 downto 0);
            rd_a, rd_b   : out std_logic_vector(7 downto 0));
    end component;
    component M_wal
        port(
            din : in std_logic_vector(2 downto 0);
            clk     : in std_logic;
            dout: out std_logic_vector(2 downto 0));
    end component;
    component M_rdb_a
        port(
        din : in std_logic_vector(7 downto 0);
        sdo : out std_logic;
        dout1, dout2: out std_logic_vector(7 downto 0));
    end component;
    component M_rdb_b
        port(
            din : in std_logic_vector(7 downto 0);
            dout : out std_logic_vector(7 downto 0));
    end component;
    signal w0, w1, w2, w3, w4, w5
                                : std_logic_vector(2 downto 0);
    signal  w6, w7, w8, w9, w10, w11, w12, w13
                    : std_logic_vector(7 downto 0);
    begin
        sac1: M_sac port map(
            clk=>clk, smc=>smc, q1=>w0, q2=>w1);

        wa_mux1 : M_wam port map(
            wa=>wa, sac=>w0, smc=>smc, we=>we,
            dout=>w2);

        wal1: M_wal port map(
            clk=>clk, din=>w2, dout=>w3);

        wad1: M_wad port map(
            clk=>clk, din=>w3, dout=>w6);

        rdb_a1   : M_rdb_a port map(
            sdo=>sdo, dout1=>w12, dout2=>rdout_a,
            din=>w8);

        rdb_b1 : M_rdb_b port map(
            din=>w7, dout=>rdout_b);

        ra_mux_a1 : M_ram_a port map(
            ra_a=>ra_a, sac=>w0, smc=>smc,
            dout=>w4);

        ra_mux_b1 : M_ram_b port map(
            ra_b=>ra_b, sac=>w1, smc=>smc,
            dout=>w5);

        rad_a1   : M_rad_a port map(
            din=>w4, dout=>w10);

        rad_b1 : M_rad_b port map(
            din=>w5, dout=>w9);

        wd_mux1 : M_wdm port map(
            wd=>wd, rd=>w13, smc=>smc,
            sdi=>sdi, dout=>w11);

        wdl1: M_wdl port map(
            din=>w11, clk=>clk, dout=>w12);

        memory1 : M_mem port map(
            ra_a=>w10, ra_b=>w9, wa=>w6, wd=>w12,
            rd_a=>w8, rd_b=>w7);
end arch;
```

図 A.3　マルチポートメモリアレイの階層構造