連想記憶プロセッサのパターン並列故障シミュレーションへの応用

世古 忠† 菊野 亨††

†奈良工業高等専門学校 ††大阪大学基礎工学部

本論文では連想記憶プロセッサを用いて並列故障シミュレーションを高速化する試みについて述べる．先ず，連想記憶プロセッサ上で並列故障シミュレーションを実行した場合の計算複雑度について議論する． 次に， 高位論理合成系 PARTHENON を用いて行ったスタンドアロン型の連想記憶プロセッサのアーキテクチャ設計， 及び，その上で実現したパターン並列故障シミュレーションについて紹介する．最後に， 動作確認の目的で行った適用実験の結果についても報告する．

In this paper, we apply a content addressable memory processor to a parallel fault simulation of combinational circuits. First, we discuss computational complexities of parallel fault simulations on content addressable memory processor. Then, we design architecture of a content addressable memory processor by using high level synthesis system(PARTHENON), and implement pattern parallel fault simulation on it. Finally, some experimental evaluations are executed using several example circuits to show usefulness of the proposed method.

1 まえがき

論理回路の故障シミュレーションは，テストパターンの故障検出率の評価，テストパターンの生成，故障辞書の作成などに用いられており，論理回路の故障検査に不可欠なものである [1][2]. 通常，故障シミュレーションでは，仮定された複数の故障と多くのテストパターンに対して，論理回路のシミュレーションを行う必要がある．このため，回路規模の増大に伴う計算量の増大が問題になっており，この問題を解決することに関心が集まってきている．

こうしたアプローチの１つとして，連想記憶プロセッサのもつビット毎の並列演算機能を利用して計算時間の短縮をはかる試みがある [4] [8][9]. 連想記憶プロセッサは SIMD 型の並列プロセッサと考えられる．連想記憶プロセッサの実現法としては，連想記憶をホスト計算機の主記憶空間の一部に組込む方式と，スタンドアロン型のプロセッサとして実現する方式がある．本論文では，スタンドアロン型の方式に注目し， プロセッサを高位論理合成系 (PARTHENON[5][6]) を用いて設計し，それを用いた故障シミュレーションの高速化の実現について議論する．

一方， 並列アルゴリズムは故障並列法とパターン並列法に分類できる [3][7]. 石浦 [4] は，故障並列アルゴリズムを連想記憶を組み込んだホスト計算機上で実現している．これに対し，本論文ではパターン並列アルゴリズムをスタンドアロン型の連想記憶プロセッサ上で実現している．

本論文での主な成果をまとめると，(1) スタンドアロン型の連想記憶プロセッサ上での並列故障シミュレーションの計算量の解析を行ったこと，(2) スタンドアロン型連想記憶プロセッサを PARTHENON を用いて設計したこと，(3) 設計した連想記憶プロセッサ上でパターン並列アルゴリズムを実現したこと， となる．

2 連想記憶のモデル

ここで考える連想記憶のモデルを図 1 に示す． 基本的に， 連想記憶は記憶アレイ部，マスクレジスタ

Application of Content Addressable Memory Processor to Pattern Parallel Fault Simulation

†Tadashi SEKO, Nara National College of Technology.

††Tohru KIKUNO, Faculty of Engineering Science, Osaka University.

と選択レジスタから構成される．記憶アレイ部の各語はxビットの長さをもち，全体でy語ある．各語は信号線により選択レジスタに接続されている．連想記憶に対して次の7種類の命令が定義されている．

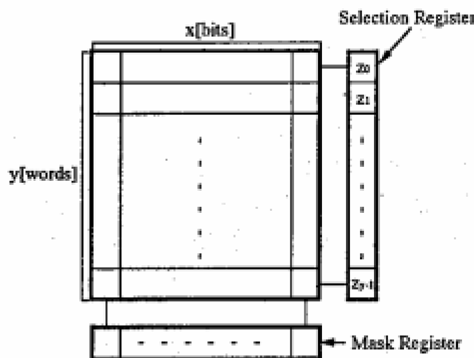


図1 連想記憶の構成

(1) Setmask register(MKSET Data)
長さxビットのDataをマスクレジスタにセットする．

(2) Search data(SEARCH {thru,and,or} Data)
連想記憶の各語に対し，マスクレジスタによってマスクされていない部分の値と，Dataとを比較して，一致するときその語に対応する選択レジスタのビットの値を次の様に定める．thruなら1にする，andなら選択レジスタのビットとの論理積をとり，一方orなら論理和をとり，その結果を選択レジスタの新しい値とする．

(3) Shift selection register(SHIFT {fd,bfd})
選択レジスタの中味をfdなら前方向に，bfdなら後方向(bfd)に1ビットだけシフトする．

(4) Parallel write(PWRITE {all,sel,unsel}, Data)
allなら連想記憶のすべての番地にDataを並列に書き込む．selなら対応する選択レジスタの値が1となっている番地にDataを並列に書き込む．次にunselなら選択レジスタの値が0となっている番地に書き込む．

(5) Parallel read(PREAD {all,sel,unsel)
allなら連想記憶のすべての番地からデータを並列に読み出す．selなら対応する選択レジスタの値が1となっている番地からデータを並列に読み出す．unselなら逆に選択レジスタの値が0となっている番地から並列に読み出す．

(6) Write data(WRITE adr Data)
adrの示す連想記憶の番地にDataを書き込む．

(7) Read data(READ adr)
adrの示す連想記憶の番地からデータを出力用メモリに読み出す．

3 並列故障シミュレーション

故障シミュレーションでは，通常，与えられた回路，故障集合とテストパターン集合に対して，どのテストパターンによってどの故障が検出可能であるかを決定する．ここでは，代表的な2つの並列故障のシミュレーション法であるパターン並列法と故障並列法について説明する．

3.1 パターン並列の故障シミュレーション

連想記憶を用いたパターン並列の故障シミュレーションの実現を例を用いて説明する．パターン並列の故障シミュレーションのアルゴリズム(Algorithm PP)の概要を次に示す．

Algorithm PP

Step1: 故障値格納領域(/0，/1)に故障値0または1をセットする．

Step2: すべてのテストパターンを連想記憶に読み込む．

Step3: 正常回路に対し，全てのテストパターンについて論理シミュレーションを行ない，正常時の出力値を計算する．

Step4: 次の(1)と(2)の一連の処理を各故障について並列に実行する．
(1) 故障回路に対して，全てのテストパターンについてゲートの評価を行って，故障時の出力値を計算する．
(2) 故障時の出力値と正常時の出力値を比較する．もし値が異るテストパターンがあ

れば，そのテストパターンは故障を検出可能であると判断する．

[例 1] ここでは図 2 に示す 4 個の NAND ゲート G_1, G_2, G_3, G_4， 4 本の入力信号線 x_1, x_2, x_3, x_4， 1 本の出力信号線 f， その他の 5 本の信号線 g, h, i, j, k から構成される回路について考える． 図 3(a) は，その回路に対する連想記憶上での表現を示す． 上記の 10 本の信号線に対応する 10 個のビット以外に f_α，/0，/1 の 3 ビットを含んでいる． f_α は与えられた縮退故障が回路上に発生したと仮定した場合の回路の出力を格納する． 一方，/0，/1 は各語について見ると 0，1 の値をもち， それぞれ 0 縮退故障と 1 縮退故障を表す．

図 3(a) は故障シミュレーションの入力を， 図 3(b) は故障シミュレーションの出力を表している． テストパターン $T_1 = (1,1,0,0)$ に対して, 故障のない場合には内部信号線と出力信号線の値は $(g,h,i,j,k,f) = (1,1,1,0,1,1)$ となる． ここで信号線 j の 1 縮退故障について考える． この場合には j の信号線の値としては, /1 フィールドの値 (同様に， 0 縮退故障の場合には/0 フィールドの値) を用いて計算し， 求まった出力信号線での値を f_α とする． この場合 $(g,h,i,j,k,f_\alpha) = (1,1,1,1,1,0)$ となる (図 3(b) 参照)． $f \neq f_\alpha$ が成立するのでテストパターン T_1 により信号線 j の 1 縮退故障が検出できることが分かる．

図 3(b) はすべての可能なテストパターン T_0, T_1, $\ldots, T_{12}$ に対する信号線 j の 1 縮退故障に関する結果を示す． この結果から信号線 j の 1 縮退故障は ($f \neq f_\alpha$ となっている) テストパターン T_1, T_6, T_{12} によって検出可能であることが分かる．

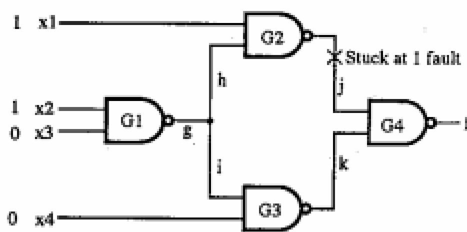


図 2 回路例 1

	x1	x2	x3	x4	g	h	i	j	k	f	fα	/0	/1
T0	1	1	1	0								0	1
T1	1	1	0	0								0	1
T2	0	1	0	0								0	1
T3	0	0	0	0								0	1
T4	0	0	1	0								0	1
T5	0	0	1	1								0	1
T6	1	0	1	1								0	1
T7	1	1	1	1								0	1
T8	1	1	0	1								0	1
T9	1	0	0	1								0	1
T10	0	0	0	1								0	1
T11	0	1	0	1								0	1
T12	1	0	1	0								0	1

(a) 初期状態

	x1	x2	x3	x4	g	h	i	j	k	f	fα	/0	/1
T0	1	1	1	0	0	0	0	1	1	0	0	0	1
T1	1	1	0	0	1	1	1	0	1	1	0	0	1
T2	0	1	0	0	1	1	1	1	1	0	0	0	1
T3	0	0	0	0	1	1	1	1	1	0	0	0	1
T4	0	0	1	0	1	1	1	1	1	0	0	0	1
T5	0	0	1	1	1	1	1	1	0	1	1	0	1
T6	1	0	1	1	1	1	1	0	1	1	0	0	1
T7	1	1	1	1	0	0	0	1	1	0	0	0	1
T8	1	1	0	1	1	1	1	0	0	1	1	0	1
T9	1	0	0	1	1	1	1	0	0	1	1	0	1
T10	0	0	0	1	1	1	1	1	0	1	1	0	1
T11	0	1	0	1	1	1	1	1	0	1	1	0	1
T12	1	0	1	0	1	1	1	0	1	1	0	0	1

(b) 最終状態

図 3 パターン並列の故障シミュレーション

3.2 故障並列の故障シミュレーション

連想記憶を用いた故障並列の故障シミュレーションの実現を例を用いて説明する． 故障並列の故障シミュレーションのアルゴリズム (Algorithm FP) の概要を次に示す．

Algorithm FP

Step1: 各テストパターンについて次の Step2, Step3 を実行する．テストパターン p を 1 つ読み込む．

Step2: 正常回路に対し，読み込んだテストパターンについて論理シミュレーションを行い，正常時の出力値を計算する．

Step3: 各故障について，次の (1)〜(3) を並列に実行する．

(1) 故障に対応して，連想記憶上で対応する信号線の値を書き換える．

(2) 故障回路の出力を計算する．

(3) 正常回路と故障回路の出力値を比較する．もし値が異なる故障回路が存在すれば，その故障は検出可能であると判断する．

[例2] 例1と同様，図2の回路について考える．従って，信号線の集合は $\{x_1, x_2, x_3, x_4, g, h, i, j, k, f\}$ となる．今，次のような信号線の故障集合 $\{x_1/0, x_2/1, x_3/0, x_4/1,\ g/0, h/0, i/1, j/1, k/1, f/0\}$ (ただし，$x_1/0(1)$ は信号線x1の0(1)縮退故障を表す) を考える．便宜上 $x_1/0,\ x_2/1, x_3/0, x_4/1, g/0, h/0, i/1, j/1, k/1, f/0$ をそれぞれ $f_1, f_2, \cdots, f_{10}$と表す． 信号線の信号値の他に故障番号を2進数で格納する．故障番号0の語は正常回路に対応している．回路に故障 f_iを仮定した場合を故障番号 i の行に対応させる．

図4(a) に連想記憶の初期状態を表す． 同図では入力信号線にテストパターン (x_1, x_2, x_3, x_4)=(1,1,0,0) を読み込んだ状態を示す． 図4(b) には故障並列故障シミュレーションの結果を示す． 正常回路の出力は1であり，故障 f_1を仮定したときの出力値0と異なっているのでテストパターンにより，f_1の故障が検出できることが分かる． 図4(b) より， 最終的にはテストパターン (1,1,0,0) によって故障 f_1, f_5, f_6, f_8の故障が検出できることが分かる．

	fault no.				x1	x2	x3	x4	g	h	i	j	k	f
f0	0	0	0	0	1	1	0	0						
f1	0	0	0	1	1	1	0	0						
f2	0	0	1	0	1	1	0	0						
f3	0	0	1	1	1	1	0	0						
f4	0	1	0	0	1	1	0	0						
f5	0	1	0	1	1	1	0	0						
f6	0	1	1	0	1	1	0	0						
f7	0	1	1	1	1	1	0	0						
f8	1	0	0	0	1	1	0	0						
f9	1	0	0	1	1	1	0	0						
f10	1	0	1	0	1	1	0	0						

(a) 初期状態

	fault no.				x1	x2	x3	x4	g	h	i	j	k	f
f0	0	0	0	0	1	1	0	0	1	1	1	0	1	1
f1	0	0	0	1	0	1	0	0	1	1	1	1	1	0
f2	0	0	1	0	1	1	0	0	1	1	1	0	1	1
f3	0	0	1	1	1	1	0	0	1	1	1	0	1	1
f4	0	1	0	0	1	1	0	1	1	1	1	0	0	1
f5	0	1	0	1	1	1	0	0	0	0	0	1	1	0
f6	0	1	1	0	1	1	0	0	1	0	1	1	1	0
f7	0	1	1	1	1	1	0	0	1	1	1	0	1	1
f8	1	0	0	0	1	1	0	0	1	1	1	1	1	0
f9	1	0	0	1	1	1	0	0	1	1	1	0	0	1
f10	1	0	1	0	1	1	0	0	1	1	1	0	1	1

(b) 最終状態

図4 故障並列の故障シミュレーション

4 時間計算量の分析

4.1 準備

連想記憶を用いた並列故障シミュレーションに要する時間計算量を分析するため，幾つかの記号を導入する．シミュレーションの対象になる論理回路のゲート数を n，故障の総数を m，テストパターンの総数を l とする．パターン並列アルゴリズム，及び，故障並列アルゴリズムを適用した場合のゲートの評価回数をそれぞれ，N_p，N_fとし，実行時間を T_p，T_fとする．

更に，図1に示したように連想記憶の記憶アレイ部のサイズを x(ビット)×y(語) とする．以下の分析においては，連想記憶のサイズは充分に大きいものと仮定する．すなわち，記憶アレイ部には回路に含まれるすべての信号線を1語として表現できるものとする．更に，すべてのテストパターン及びすべての故障が記憶アレイ部に格納できるものとする (つまり，$y \geq l$，かつ，$y \geq m+1$)．

4.2 主な結果

ゲートの評価回数について，次の補題1，2が成立する．

補題 1: パターン並列アルゴリズムによる故障シミュレーションの場合，ゲートの評価回数 N_pは次式で与えられる．

$$\begin{aligned} N_p &= \alpha n \qquad (1) \\ &\leq n(m+1) \end{aligned}$$

ここで，$\alpha = 1 + k_1 + k_2 + \cdots + k_m$であり，各$k_i (1 \leq i \leq m)$は$i$番目の故障時に評価したゲートの割合である．

補題 2: 故障並列アルゴリズムによる並列故障シミュレーションの場合，ゲートの評価回数N_fは次式で与えられる．

$$N_f = ln \quad (2)$$

次に，上述の補題より並列故障シミュレーションに要する実行時間に関する定理が証明できる．

定理 1: パターン並列アルゴリズムによる故障シミュレーションの場合，実行時間T_pは，次式で与えられる．ここでc_1はゲートの評価時間の平均とする．

$$\begin{aligned} T_p &= c_1 N_p \quad (3) \\ &= c_1 \alpha n \\ &\leq c_1 n(m+1) \end{aligned}$$

定理 2: 故障並列アルゴリズムによる故障シミュレーションの場合，実行時間T_fは次式で与えられる．ここでc_1はゲートの評価時間の平均とし，c_2は故障の挿入(Algorithm FP の Step3 の (1) の処理) に要する時間の平均とする．

$$\begin{aligned} T_f &= c_1 N_f + c_2 m \quad (4) \\ &= c_1 ln + c_2 m \quad (5) \end{aligned}$$

定理1，2より次の系1，2が導かれる．

系 1: パターン並列アルゴリズムによる故障シミュレーションを行なう場合の計算時間は$T_p = O(n)$である．

系 2: 故障並列アルゴリズムによる故障シミュレーションを行なう場合の計算時間は$T_f = O(n)$である．

いわゆる木状回路の場合，回路内のゲート評価を並列に実行するとき，並列度が急激に減少する．このような性質をもつ回路の場合には故障並列アルゴリズムよりもパターン並列アルゴリズムの方が有利であると思われる．

系 3: $n = 2^s - 1$個の2入力1出力ゲートからなる木状回路について考える．故障の総数を$m = 2(2^{(s+1)} - 1)$，テストパターンの総数を$l = 2^{(2^s)}$とする．このとき，ゲート評価回数N_p, N_fは次式で与えられる．

$$\begin{aligned} N_p &= O(n\ 2^s) \\ N_f &= O(n\ 2^{2^s}) \end{aligned}$$

この系3とは逆に，並列にゲート評価を実行するとき，評価の度に並列度が増大する性質をもつ回路の場合には，パターン並列アルゴリズムに比べ故障並列アルゴリズムが有利になると予想される．

5 連想記憶プロセッサの設計

連想記憶に関する基本機能を実現した連想記憶プロセッサを設計した．本プロセッサはスタンドアロン型の SIMD プロセッサとみなすことができる．

5.1 アーキテクチャ

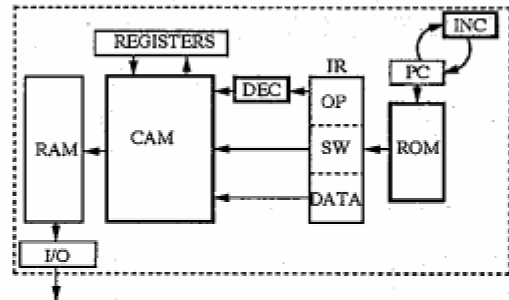


図5 連想記憶プロセッサの構成

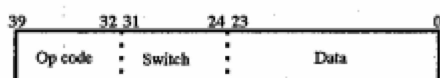
図5に連想記憶プロセッサの構成図を示す．命令を格納する ROM，次に実行すべき ROM のアドレスを指示するプログラムカウンタ PC，インクリメンタ INC，読み出した命令を格納する命令レジスタ IR，デコーダ DEC，連想記憶部，出力用メモリから構成されている．

連想記憶部には，IR を通してアドレス，データ及び命令が与えられる．一致検索機能の高速化をはかるために，連想記憶部からインデックスレジスタを除去

し，命令語内に Data 部を付加して，直接 Data を指定できるようにしている．

5.2　命令セット

図 6 に命令語の形式を示す．命令語は演算コード部 Op，スイッチ部 Switch，データ部 Data から構成される．2．で述べた連想記憶の基本機能を実現するために表 1 の命令セットを実現した．



(注：Switchでは Address,dir,cond,opのいずれかが示される.)

図 6　命令語の形式

表 1　命令セット一覧

Instruction	Mnemonic	Op code	Switch	Data
マスクのセット	MKSET	00000000	——	有り
データ読み出し	READ	10000001	address	—
RP のシフト	RPSFT	10000010	fw:00000000	—
			bfw:00000001	
並列読み出し	PREAD	10000011	all:00000000	—
			sel:00000001	
			unsel:00000010	
データ書き込み	WRITE	01000100	address	有り
一致検索	SEARCH	01000101	thru:00000000	有り
			and:00000001	
			or:00000010	
並列書き込み	PWRITE	01000110	all:00000000	有り
			sel:00000001	
			unsel:00000010	
出力メモリ初期化	CLEAR	11000111	——	—

5.3　SFL による動作記述

連想記憶プロセッサの動作の記述には機能記述言語 SFL を用いた．付録図 (a)〜(c) に SFL での記述例を示す．付録図 (a) では，インクリメンタ inc，命令デコーダ dec，rom，ram の各機能が記述されている．ram では各命令の動作を記述している．付録図 (b) では，一致検索命令の機能の動作定義を書いている．具体的には，命令語の switch 部で与えられる検索条件の下に，マスクされていない部分について入力データとのマッチングをとって一致した語に対応するレスポンスレジスタのビットを 1 にすることが指定されている．この動作はメモリの各語に対して並列に行われるため，1 ユニット時間で一致検索命令が実行される．更に，この命令の実行は付録図 (c) に示す様に，2 段のパイプライン処理方式をとっている．

5.4　並列故障シミュレーションへの適用

設計した連想記憶プロセッサ上で，図 2 の回路と図 7 に示す 1 ビット全加算器の 2 つのサンプル回路について故障シミュレーションのプログラムを作成して評価を行なった．その結果，図 2 と図 7 の回路に対するパターン並列法の命令数はそれぞれ，397 命令，1882 命令であった．一方，故障並列法による命令数は，1982 命令，2988 命令であった。従ってパターン並列法は故障並列法に比べ，それぞれ 1/5，6/10 の命令数に削減が達成されていた．

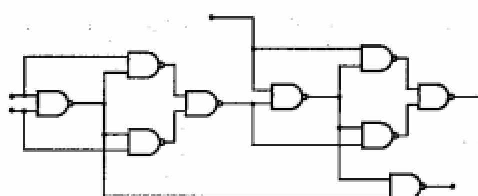


図 7　回路例 2

6　おわりに

本論文では，連想記憶を用いた 2 つの並列故障シミュレーション法の計算複雑性について考察を行なった．その結果ゲート数 n に対してパターン並列法の計算時間が $O(n)$ であることを示した．更に連想記憶プロセッサの設計を行い，機能記述言語 SFL でその機能を記述した．今後の課題としては，CAM チップの実現及び並列処理に有効な命令セットの開発などが挙げられる．

謝辞　本研究を進めるに当たり，貴重なご意見をいただいた大阪大学 石浦 菜岐佐講師に深謝致します．又，本報告の作成を支援していただいた大阪大学楠本 真二博士に深謝致します．最後に，評価実験で御協力いただいた加納健司氏 (現在，福井大学) に感謝致します．

参考文献

[1] M. Abramovici, M. A. Breuer and A. D. Friedman, *Digital systems testing and testable design*, Computer Science Press(1990).

[2] M.A.Breuer and A.D. Friedman,*Diagnosis & reliable design of digital systems*, Computer Science Press(1976).

[3] N. Ishiura, M. Ito and S. Yajima,"Dynamic two-dimensional parallel simulation technique for high-speed fault simulation on a vector processor", *IEEE Trans. on Computer-Aided Design*,vol. 9,no. 8, pp. 868-875(1990).

[4] N. Ishiura and S. Yajima, "Linear time fault simulation algorithm using a content addressable memory", *IEICE Trans. on Fundamentals*,vol. E75-A, no.3, pp. 314-320(1992).

[5] Y. Nakamura, K. Oguri, A. Nagoya and R. Nomura, "A hierarchical behavioral description based CAD system", EURO ASIC-90, pp. 282-287(1990).

[6] NTT データ通信株式会社, " PARTHENON ユーザーズマニュアル",NTT データ通信株式会社(1990).

[7] E.W. Thompson and S.A. Szygenda, "Digital logic simulation in a time-based,table-driven environment:part 2. parallel fault simulation", Comput.,vol. 8,pp.38-49(1975).

[8] 安浦 寛人, "機能メモリによる超並列処理", 情報処理,Vol.32,No.12,pp.1260-1267(1991).

[9] 安浦, 渡部, 左達, 田丸, " 機能メモリ型並列プロセッサ FMPP 上での論理シミュレーション", 電子情報通信学会計算機システム研究会資料 CPSY90-94(1991).

付録 1

```
module cam {
  circuit_type   inc {...}  /* PC=PC+1 */
  circuit_type    dec {     /* 命令デコード */
    term       work<16>;
    input      in<8>;
    output     mkset;
    output     read;
    output     shift;
    output     pread;
    output     write;
    output     search;
    output     pwrite;
    output     clrout;
    instrin    dec;
    instruct_arg    dec(in);
    instruct        dec    par {
      work=/(in<3:0>);
      mkset  = work<0>;
      read   = work<1>;
      shift  = work<2>;
      pread  = work<3>;
      write  = work<4>;
      search = work<5>;
      pwrite = work<6>;
      clrout = work<7>;
    }
  }
  circuit_type    rom {...} /* ROM */
  circuit_type    ram {
    input     data<24>,adrs<8>,sw<8>;
    mem       cell[32]<24>,outcell[32]<24>;
    reg       rp<24>,mk<24>;
    term      rp0,rp1,rp2,...,rp23;
    instrin   mks;
    instrin   rd;
    instrin   sft;
    instrin   prd;
    instrin   wrt;
    instrin   seh;
    instrin   pwt;
    instrin   clo;
    instruct_arg   mks(data);
    instruct_arg   rd(adrs);
    instruct_arg   sft(sw);
    instruct_arg   prd(sw);
    instruct_arg   wrt(adrs,data);
    instruct_arg   seh(sw,data);
    instruct_arg   pwt(sw,data);
    instruct_arg   clo();
```

(a)circuit_type inc,dec,rom,ram

```
  instruct  seh  par {    /* 一致検索 */
    any {
      ((^(/|((^mk&data)@(^mk&cell[0b00000]))
           |rp<0>)&sw<1>)|
       (^(/|((^mk&data)@(^mk&cell[0b00000]))
          &(sw<0>&rp<0>))|
       (^(/|((^mk&data)@(^mk&cell[0b00000]))
          &(^sw<1>&^sw<0>))
            : rp0=0b1;
       else : rp0=0b0;
    }
    ......................................
    any {
      ((^(/|((^mk&data)@(^mk&cell[0b10111]))
           |rp<23>)&sw<1>)|
       (^(/|((^mk&data)@(^mk&cell[0b10111]))
          &(sw<0>&rp<23>))|
       (^(/|((^mk&data)@(^mk&cell[0b10111]))
          &(^sw<1>&^sw<0>))
            : rp23=0b1;
        else : rp23=0b0;
    }
        rp := rp23||rp22||...||rp1||rp0;
}
```

(b) 一致検索命令の動作記述

```
instrin start;
stage_name  if {
  task  ift();
}
stage_name  exec {
  task  ext();
}
instruct  start  generate if.ift();
stage  if {
  state_name  fetch1;
  first_state fetch1;
  state  fetch1 par {
    ir := rom.read(pc).out_data;
    pc := inc.inc(pc).out;
    generate  if.ift(); /* パイプライン処理 */
    relay     exec.ext();
  }
}
stage  exec {
  state_name  exec1;
  first_state exec1;
  state  exec1 par {
    any {
     dec.dec(ir<39:32>).mkset:ram.mks(ir<23:0>);
     dec.read    :ram.rd(ir<28:24>);
     dec.shift   :ram.sft(ir<31:24>);
     dec.pread   :ram.prd(ir<31:24>);
     dec.write   :ram.wrt(ir<28:24>,ir<23:0>);
     dec.search :ram.seh(ir<31:24>,ir<23:0>);
     dec.pwrite :ram.pwt(ir<31:24>,ir<23:0>);
     dec.clrout :ram.clo();
    }
    finish;
   }
  }
}
```

(c) 命令の実行の記述

付録図 SFL による記述