第4章　機能・論理設計1

若林　一敏 （NEC）

目次

4.1　動作記述とRTL記述
4.2　動作合成(1)　原理編
4.2.1　動作合成の意義
4.2.2　動作合成の基礎
4.2.3　動作合成のアルゴリズム

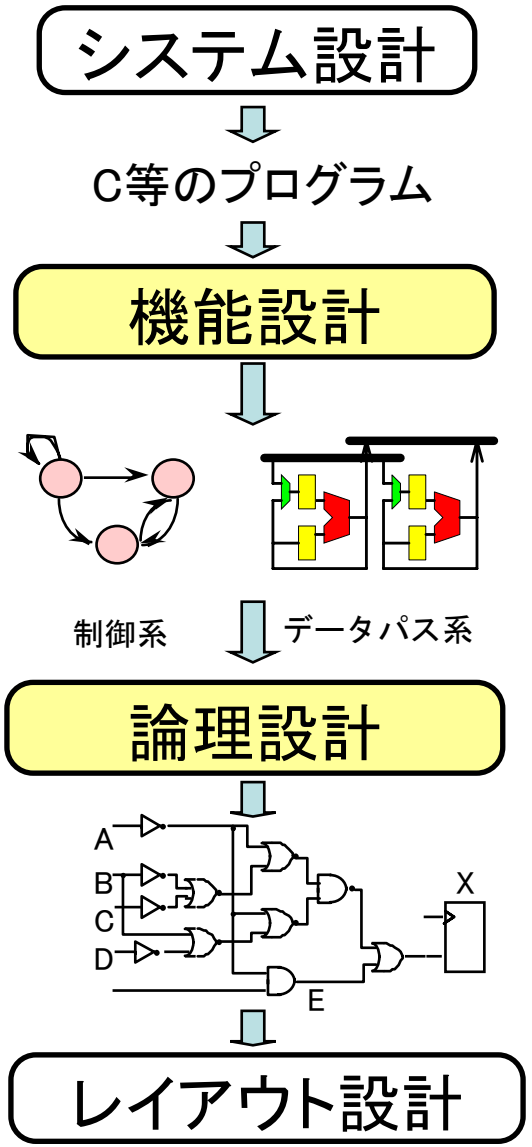
LSIの設計フローと記述の抽象度

仕様記述

動作記述

機能記述
（RTL記述）

論理記述
（ゲート記述）



処理アルゴリズム決定
HW･SWの分担

演算器数やメモリ数
処理ステップ数を決定

回路のゲート構成、
詳細タイミングを決定

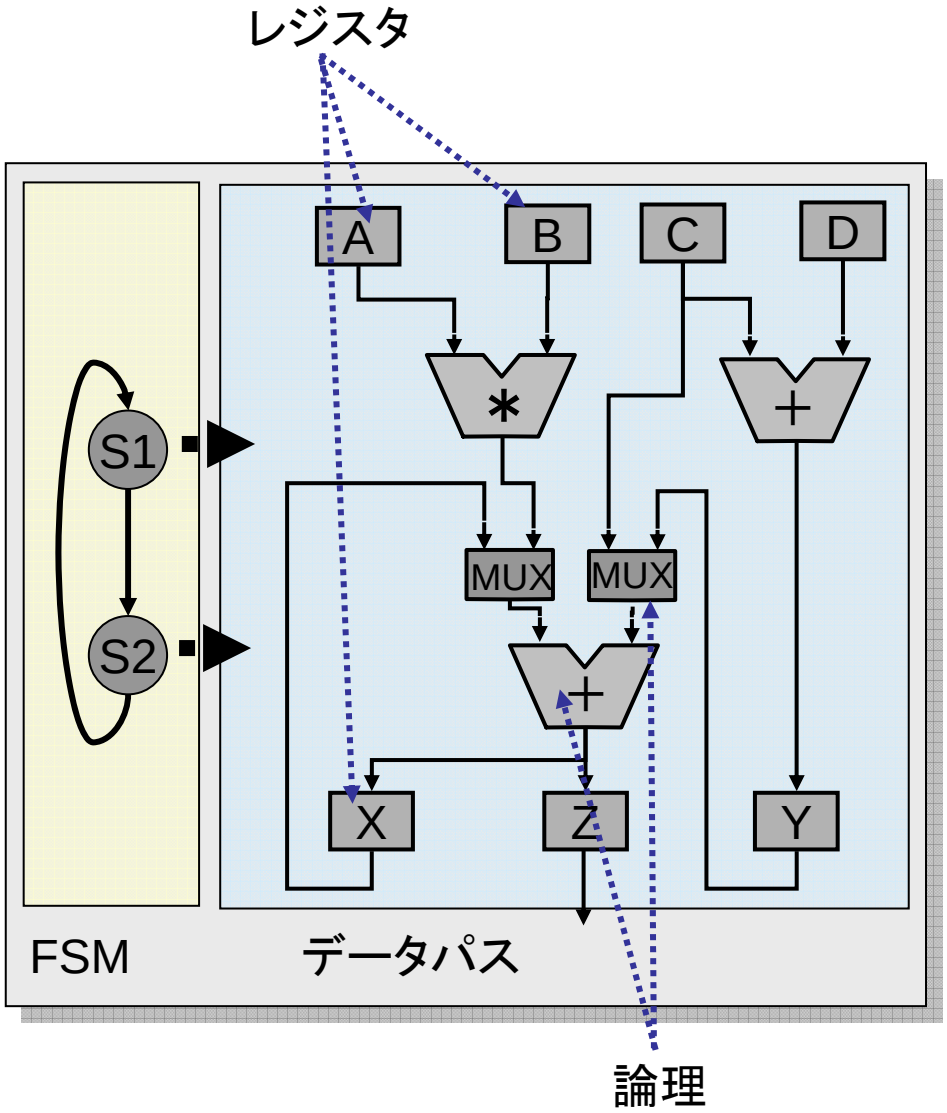
部品の配置位置、
部品間の配線の決定

動作記述

```
x = idata_a_t + idata_b_t;
ictl_c_t = ictl_c ;
if ( ictl_c_t ) {
   idata_d_t = idata_d ;
   odata_y_t = x + idata_d_t;
} else {
   odata_y_t = 2 * x;
}
```

- 回路の動作(アルゴリズム)を抽象的に記述するレベル
- 演算を「どの順番」で実行するかを指定し、「いつ実行するか」「どの部品をつかうか」などは省略可能
- ハードウェアの部品やそのつながり方(構造)は書かない(抽象度が高い)
- 但し、ビット幅や入出力端子は宣言
- 代表的言語:SystemC, SpecC

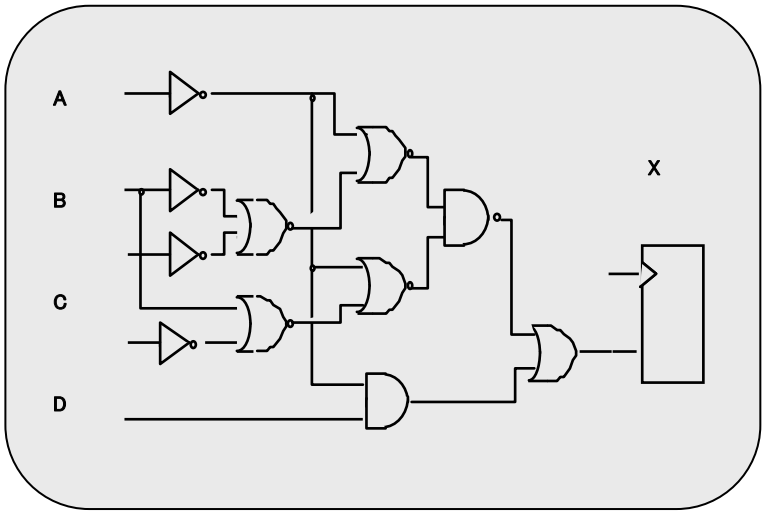
RTL(Register Transfer Level)記述



- レジスタが陽に見えるレベル
- 全体が並列に動作する
  (1ステップ進む度に、全ての行が評価される。)
- レジスタに代入される値の転送関係を示す
- レジスタや演算器等の結線関係を示すブロック図に相当するレベル

FSM:Finite State Machine

ゲートレベル



- 論理ゲート(and/or/not etc)
  まで具体化されたレベル
- 論理段数（遅延）や部品個数（面積）など
  に注意して設計する
- 人手で記述する場合は、回路図で設計
  されることが多い

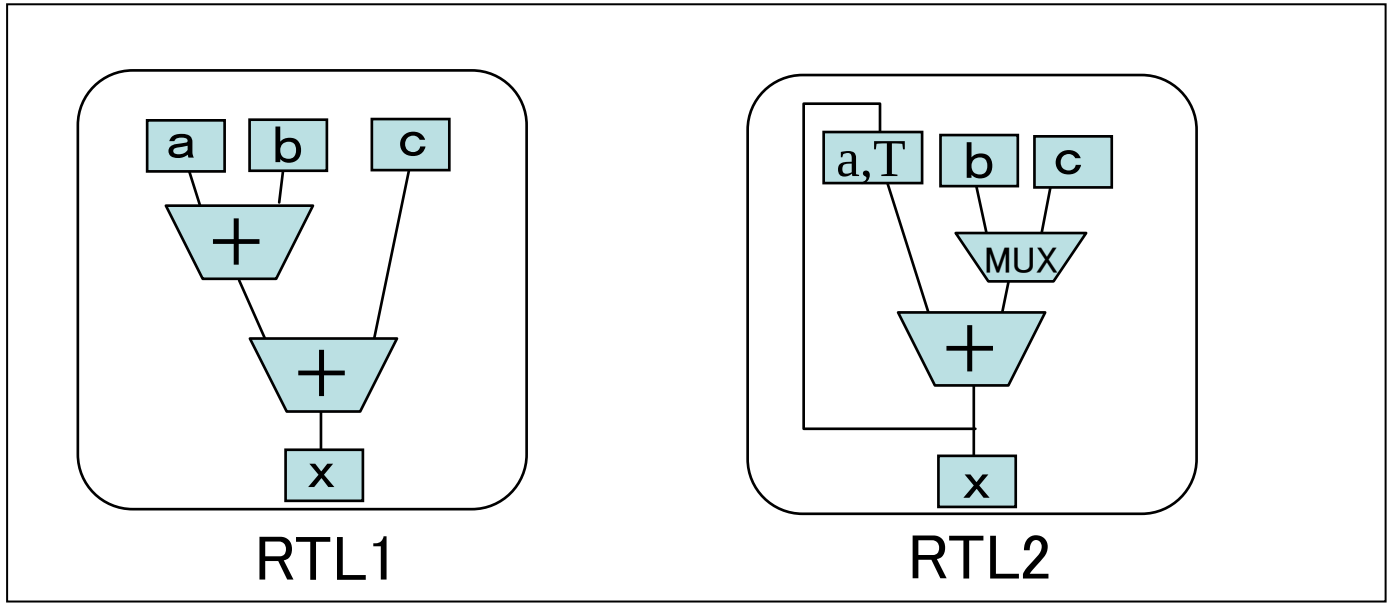
LSIの規模は、このゲート回路の部品（ゲート個数）で表現さ
れることが多い。
ただしプロセッサはトランジスタレベルで設計されることも多く、
トランジスタ数もよく使われる。

動作レベルとRTL

動作レベル

```
T = a + b;
X = T + c;
```

RTL



色々な動作レベル記述

（省）

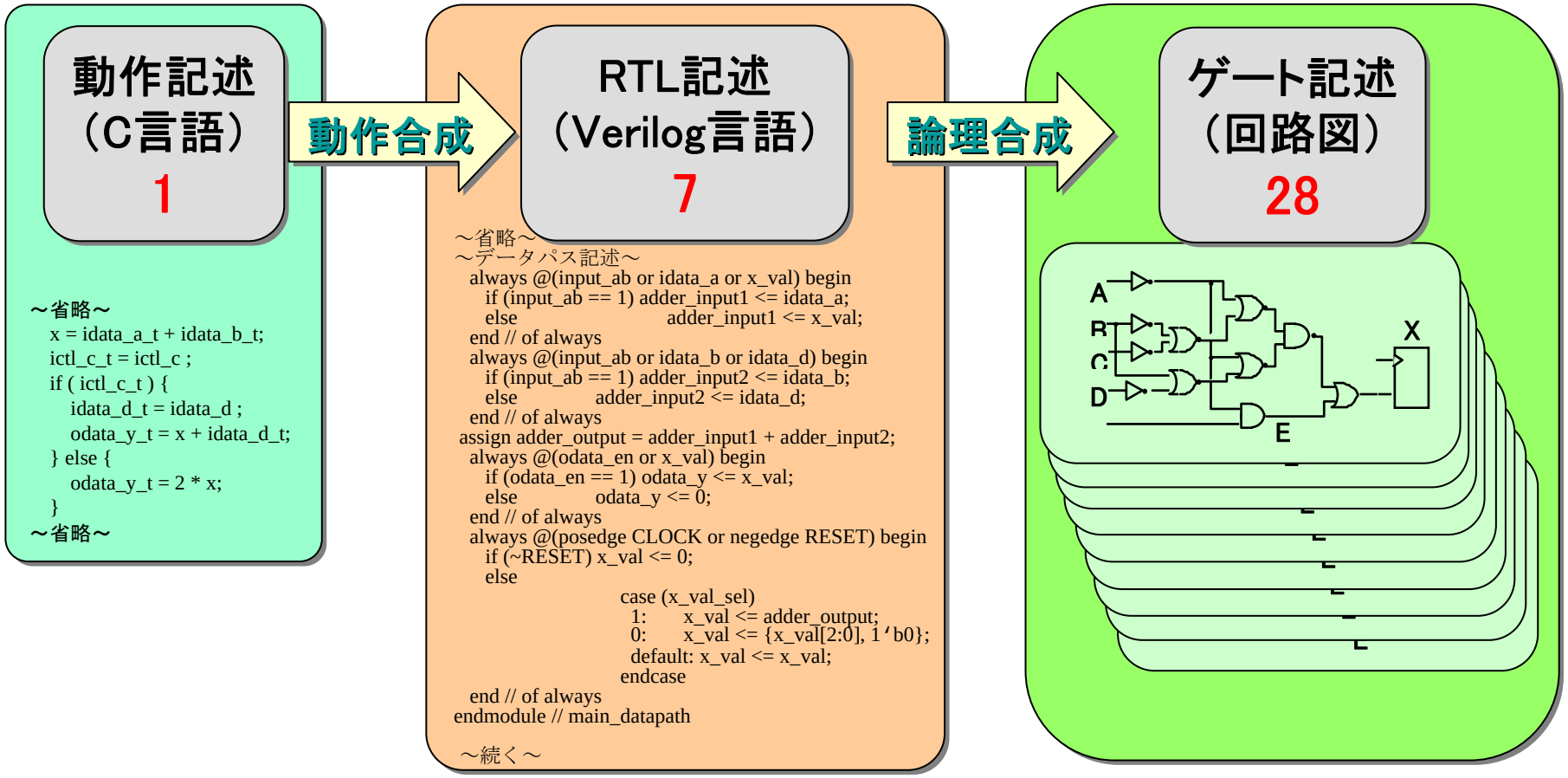
- 純粋なアルゴリズム（実装を意識したもの。意識しないものはアルゴリズムレベル）
- 時間概念：Untimedモデル（実行時刻指定無し）とTimedモデル（入出力のタイミング指定あり）、Cycle精度モデル（演算の実行タイミングの指定あり）
- 通信タイミングの精度（入出力信号について）
  1）Transaction Level（トランザクション精度）
  2）Bus Cycle Accurate（バスステップ精度）
  3）Cycle Accurate（ステップ精度）
- 構造的な精度
  1）ビット精度（Bit Accurate)：変数にビット幅（精度）を与えたレベル
  2）ピン精度（Pin Accurate）：入出力ピンを正確に表現
  3）通信をRead/Write等で表現するもの（入出力ピンが明示されない）

（省）

色々なRTL

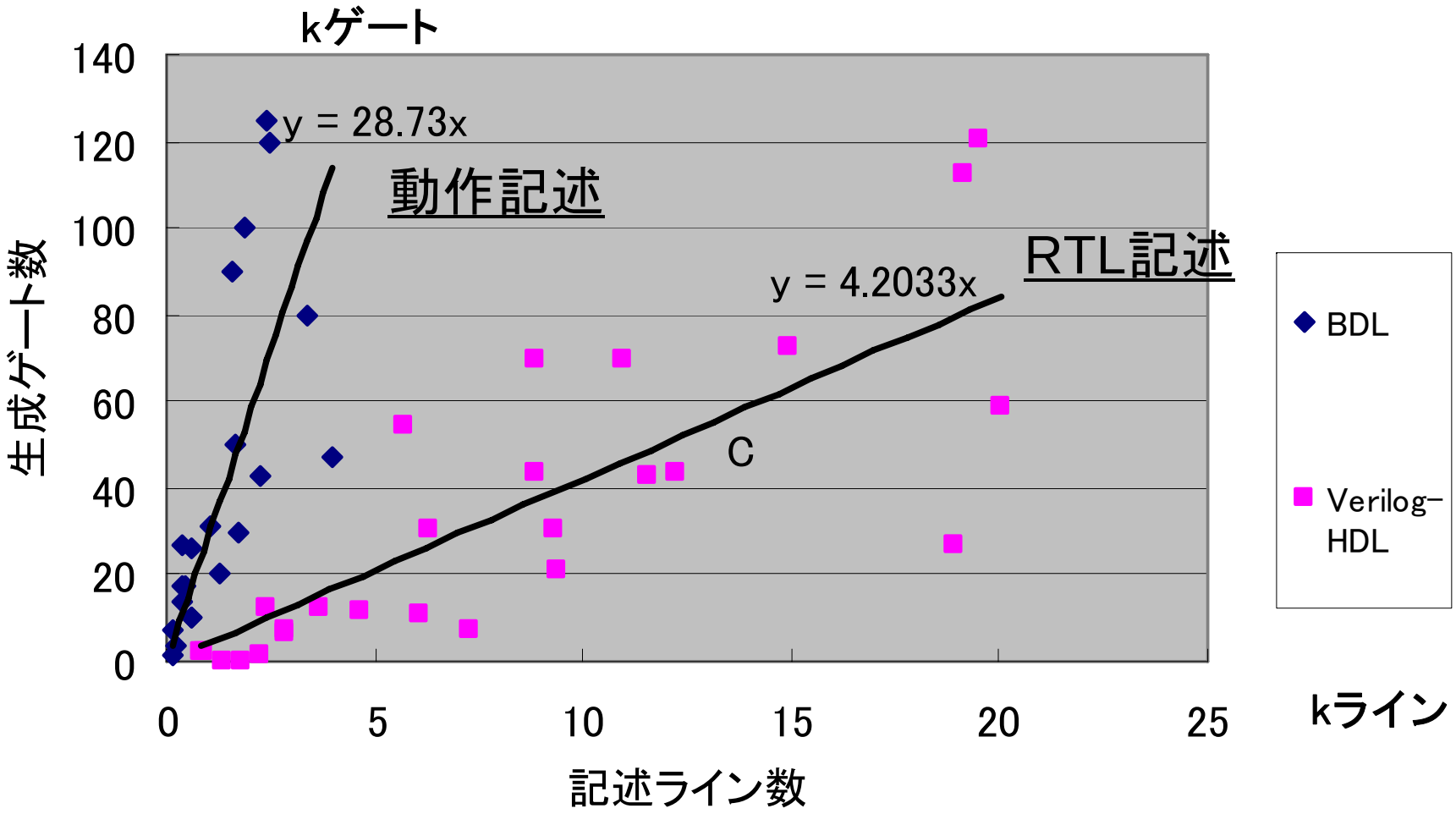
- 機能的RTL（Functional　RTL）
  - 1ステップで実行する機能が記述されている
  - 演算が演算器に割り当てられていない
  - 構造的RTLより抽象的で、機能がつかみやすく、シミュレーションが構造的RTLより通常高速
- 構造的RTL（Structural　RTL）
  - ブロック図を言語で表現
  - 演算器は割り当て済み
  - 論理合成可能（Synthesizable）

記述抽象度と記述量



適用事例に見る
動作記述とRTL記述の記述量の違い

（詳）



（省）

動作記述とRTL記述比較例

7−8倍の記述性というのは、これぐらいの差

```
in ter(3..0) aaa;
in ter(3..0) bbb;
in ter(3..0) ccc;
out reg(1..0) xxx;
process main () {
  reg(1..0) tmp_r;
  tmp_r = func(aaa,bbb,ccc);
$
  xxx = tmp_r;
}
ter(1..0) func(var(3..0) aaa, var(3..0) bbb, var(3..0) ccc) {
  ter(0..0) bit0_t;
  ter(0..0) bit1_t;
  ter(0..0) bit2_t;
  ter(1..0) result_t;
  bit0_t = aaa(3) & aaa(2) & aaa(1) | aaa(0);
  bit1_t = bbb(3) & bbb(2) | bbb(1) | bbb(0);
  bit2_t = ccc(3) & ccc(2) | ccc(1) | ccc(0); /* ･･･ /
  if(bit0_t) {
                                result_t = 0;
  }
  else if(bit1_t) {
                                result_t = 1;
  }
  else if(bit2_t) {
                                result_t = 2;
  }

  return(result_t);
}
```

動作（C）

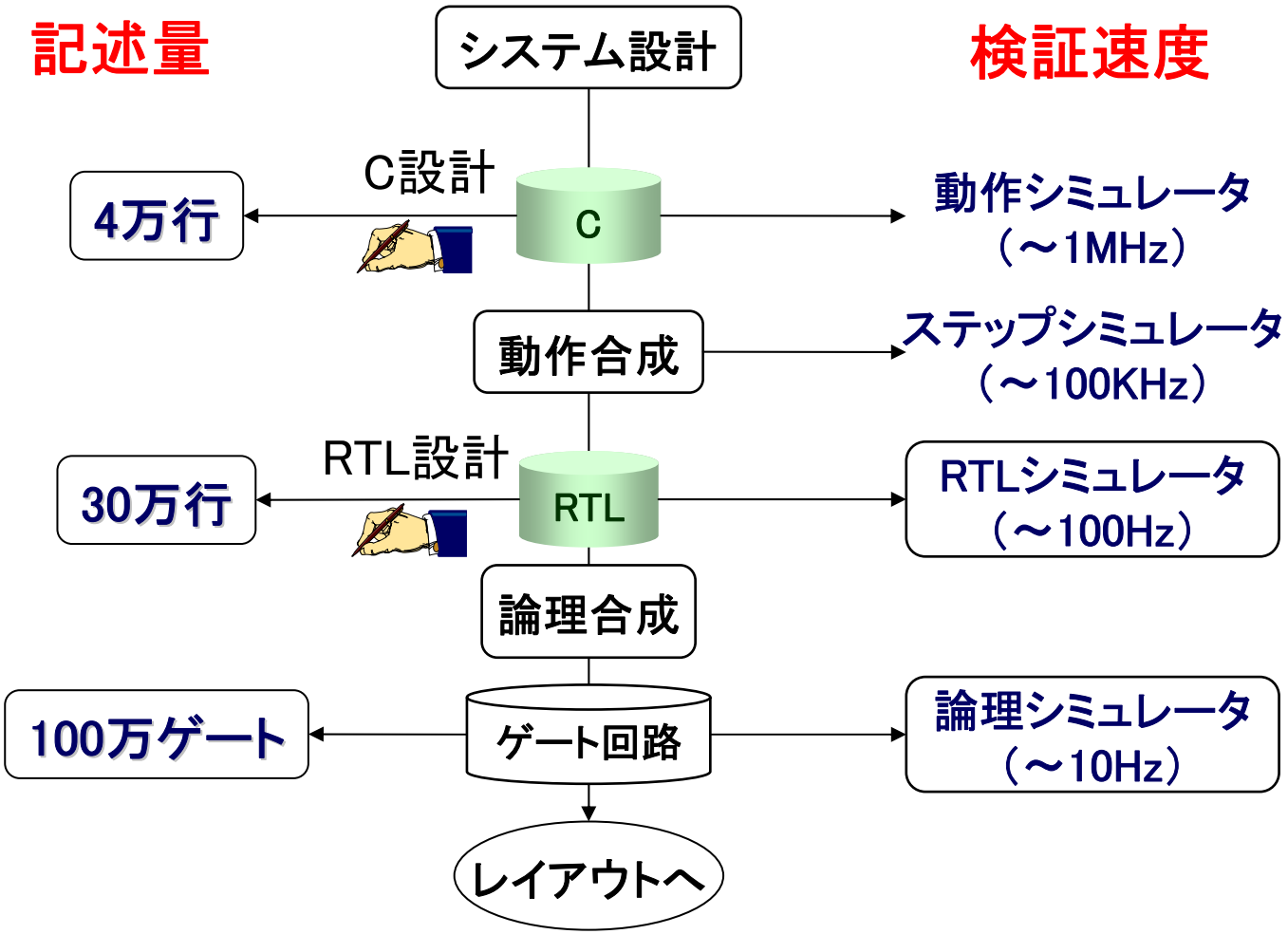
RTL

（VerilogHDL）

```
// verilogHDL_out version 2.01

module main ( aaa ,bbb ,ccc ,xxx ,CLOCK ,RESET );
input                   [3:0]                   ;
input                   [3:0]                   ;
input                   [3:0]                   ;
output                  [1:0]                   ;
input                                           ;
input                                           ;
wire                                            ;
wire                                            ;
wire                                            ;
wire                                            ;

main_fsm INST_1 ( CLOCK ,RESET ,M_01 ,ST1_02d ,ST1_01d ,ST1);
main_dat INST_2 ( aaa ,bbb ,ccc ,xxx ,CLOCK ,RESET ,M_01 ,ST1_0 ,
                        ST1_00d );

endmodule

module main_fsm ( CLOCK ,RESET ,M_01 ,ST1_02d ,ST1_01d
,ST1_0);
input                                           ;
input                                           ;
input                                           ;
output                                          ;
output                                          ;
output                                          ;
wire                    [1:0]                   g ;

assign                  ST1_00d = ~|B01_streg ;
assign                  ST1_01d = ~( B01_streg ^ 2'h1 ) ;
assign                  ST1_02d = ~( B01_streg ^ 2'h2 ) ;

main_FF_NMUX_2 INST_6 ( ST1_01d ,M_01 ,CLOCK ,RESET ,B01);

endmodule

module main_dat ( aaa ,bbb ,ccc ,xxx ,CLOCK ,RESET ,M_01 ,ST1_02,
                        ST1_00d );
input                   [3:0]                   aaa
input                   [3:0]                   ;
input                   [3:0]                   ;
output                  [1:0]                   ;
input                                           ;
input                                           ;
output                                          ;
input                                           ;
input                                           ;
input                                           ;
wire                                            ;
wire                                            ;
wire                                            ;
wire                                            ;
wire                                            ;
wire                    [1:0]                   t ;
wire                                            ;
wire                                            t ;
wire                    [3:0]                   1 ;
wire                    [3:0]                   1 ;
wire                    [3:0]                   ;
wire                    [1:0]                   1 ;
wire                    [1:0]                   r ;
reg                     [1:0]                   4 ;
assign                  xxx = xxx_1 ;           :5
assign                  aaa_1 = aaa ;           1
assign                  bbb_1 = bbb ;           1
assign                  ccc_1 = ccc ;           1
assign                  bit1_t = ( ( ( bbb_1 [3] & bbb_1 [2] )
assign                  bit2_t = ( ( ( ccc_1 [3] & ccc_1 [2] ) ;
assign                  C_01 = ( ( ( aaa_1 [3] & aaa_1 [2] )7
assign                  U_02 = ( ST1_01d & ( ~C_01 ) ) ;
assign                  U_03 = ( U_02 & bit1_t ) ;      0
assign                  U_04 = ( U_02 & ( ~bit1_t ) ) ;     0
assign                  U_05 = ( U_04 & bit2_t ) ;
assign                  result_t = verilogHDLOut_tmp4 ;
assign                  M_01 = ( ST1_00d | ST1_02d ) ;

always @ ( U_03 or U_05 )
                        casex ( { U_05 , U_03 } )
                        // synopsys parallel_case full_case
                        2'b1x :
                        // line#=and.bdl:34             ;
                        2'bx1 :
                        // line#=and.bdl:31             ;
                        default :
                                                        ;
                        endcase

main_FF_NMUX_0 INST_1 ( ST1_02d ,tmp_r ,CLOCK ,RESET
,xxx_1 );
main_FF_NMUX_1 INST_3 ( ST1_01d ,result_t ,CLOCK ,RESET
,tmp_r );
endmodule

module main_FF_NMUX_0 ( ST1_02d ,tmp_r ,CLOCK ,RESET
,xxx_1);
input                                           ;
input                   [1:0]                   r ;
input                                           ;
input                                           ;
output                  [1:0]                   1 ;
reg                     [1:0]                   1 ;
reg                     [1:0]                   ;

always @ ( xxx_1 or ST1_02d or tmp_r )
                        casex ( { ST1_02d } )
                        // synopsys parallel_case full_case
                        1'b1 :
                        // line#=and.bdl:13             r ;
                        default :
                                                        1 ;
                        endcase

always @ ( posedge CLOCK or posedge RESET )
                        if ( RESET )
                                                        ;
                        else
                                                        0 ;
endmodule

module main_FF_NMUX_1 ( ST1_01d ,result_t ,CLOCK ,RESET
,tmp_r );
input                                           d ;
input                   [1:0]                   t ;
input                                           ;
input                                           ;
output                  [1:0]                   r ;
reg                     [1:0]                   r ;
reg                     [1:0]                   ;

always @ ( tmp_r or ST1_01d or result_t )
                        casex ( { ST1_01d } )
                        // synopsys parallel_case full_case
                        1'b1 :
                        // line#=and.bdl:11             t ;
                        default :
                                                        r ;
                        endcase

always @ ( posedge CLOCK or posedge RESET )
                        if ( RESET )
                                                        ;
                        else
                                                        2 ;
endmodule

module main_FF_NMUX_2 ( ST1_01d ,M_01 ,CLOCK ,RESET
,B01);
input                                           d ;
input                                           1 ;
input                                           K ;
input                                           T ;
output                  [1:0]                   g ;
reg                     [1:0]                   eg ;
reg                     [1:0]                   5 ;

always @ ( ST1_01d or M_01 )
                        casex ( { M_01 , ST1_01d } )
                        // synopsys parallel_case full_case
                        2'b1x :
                                                        ;
                        2'bx1 :
                                                        2 ;
                        default :
                                                        ;
                        endcase

always @ ( posedge CLOCK or posedge RESET )
                        if ( RESET )
                                                        ;
                        else
                                                        5 ;

endmodule
module main_FF_NMUX_1 ( ST1_01d ,result_t ,CLOCK ,RESET , _r );
input                                           d ;
input                   [1:0]                   _t ;
input                                           ;
input                                           ;
output                  [1:0]                   r ;
reg                     [1:0]                   r ;
reg                     [1:0]                   2 ;

always @ ( tmp_r or ST1_01d or result_t )
                        casex ( { ST1_01d } )
                        // synopsys parallel_case full_case
                        1'b1 :
                        // line#=and.bdl:11             t ;
                        default :
                                                        ;
                        endcase

always @ ( posedge CLOCK or posedge RESET )
                        if ( RESET )
                                                        ;
                        else
                                                        ;
endmodule

module main_FF_NMUX_2 ( ST1_01d ,M_01 ,CLOCK ,RESET ,B0);
input                                           ;
input                                           1 ;
input                                           ;
input                                           ;
output                                          ;
reg                     [1:0]                   g ;
reg                     [1:0]                   5 ;

always @ ( ST1_01d or M_01 )
                        casex ( { M_01 , ST1_01d } )
                        // synopsys parallel_case full_case
                        2'b1x :
                                                        ;
                        2'bx1 :
                                                        2 ;
                        default :
                                                        0 ;
                        endcase

always @ ( posedge CLOCK or posedge RESET )
                        if ( RESET )
                                                        0 ;
                        else
                                                        5 ;
```

各レベルでの記述量とシミュレーション速度

百万ゲート回路に必要な記述行数の例



4.1　動作記述とRTL記述

4.2　動作合成(1)　原理編

⇨ 4.2.1　動作合成の意義

4.2.2　動作合成の基礎

4.2.3　動作合成のアルゴリズム

VLSIの設計工程と合成ツール

仕様

動作記述

4.2章、4.3章

高位合成
機能合成ツール
動作合成

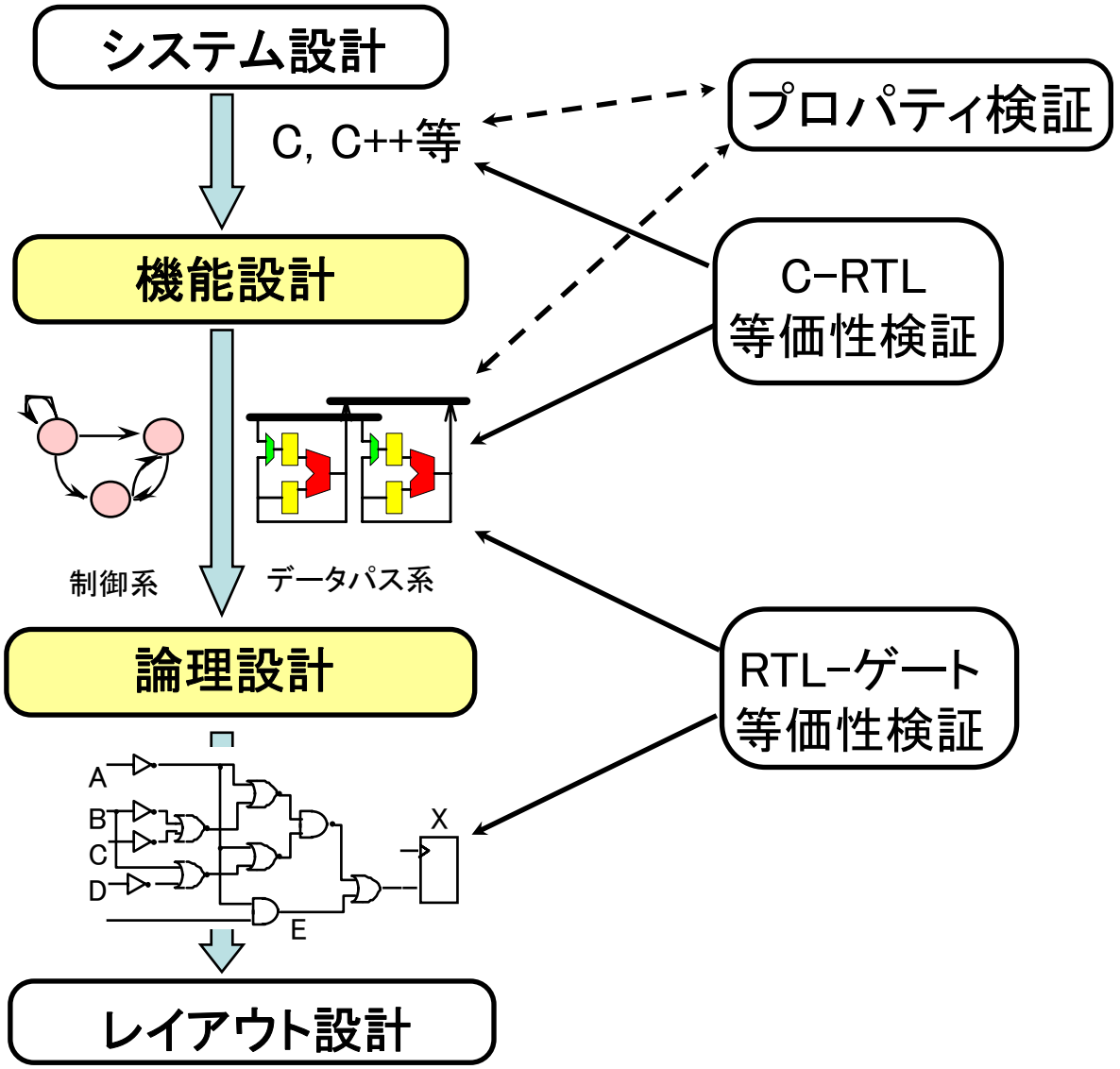
機能記述
（RTL記述）

4.4章

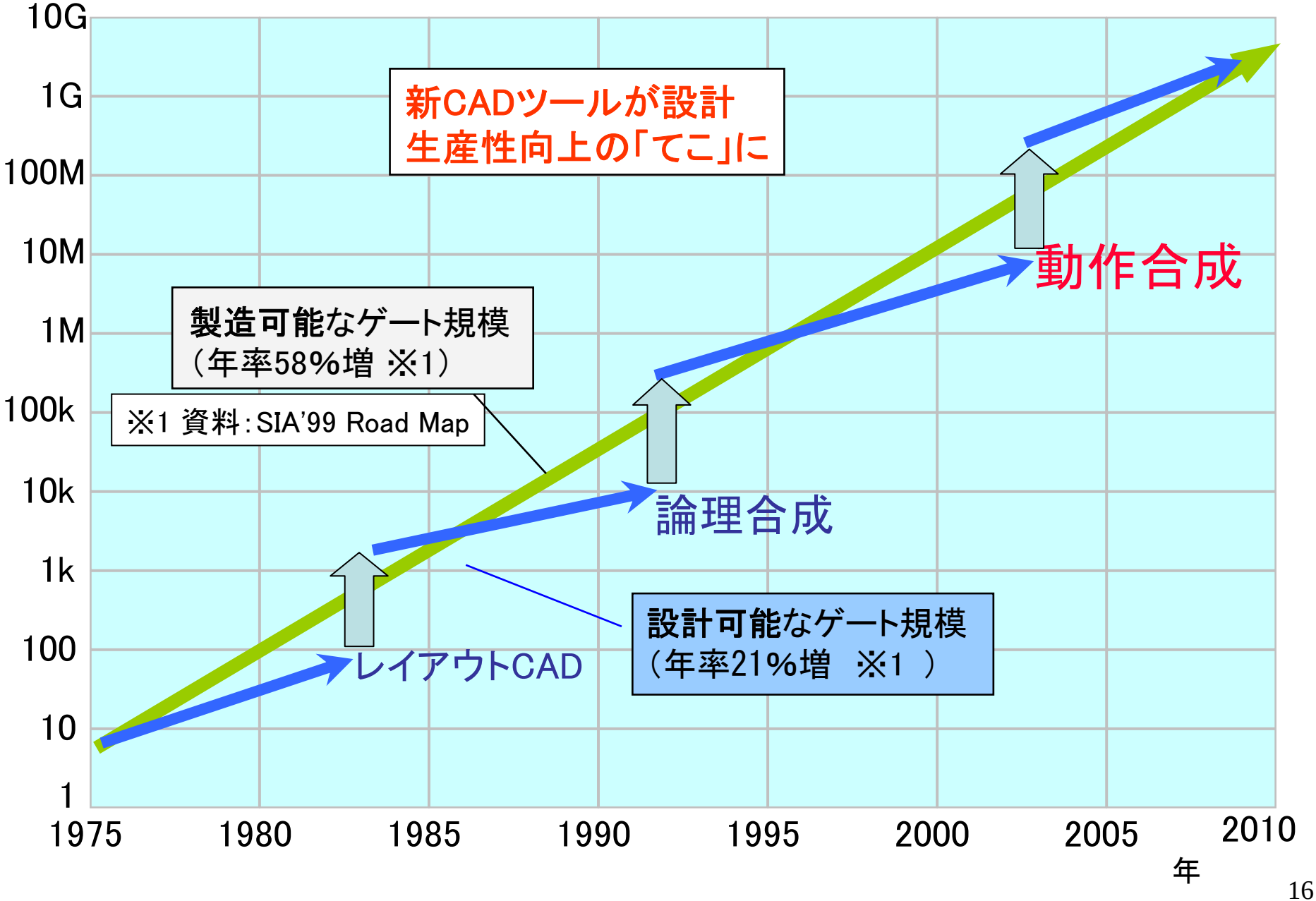
論理合成ツール

論理記述
（ゲート記述）

自動配置配線ツール

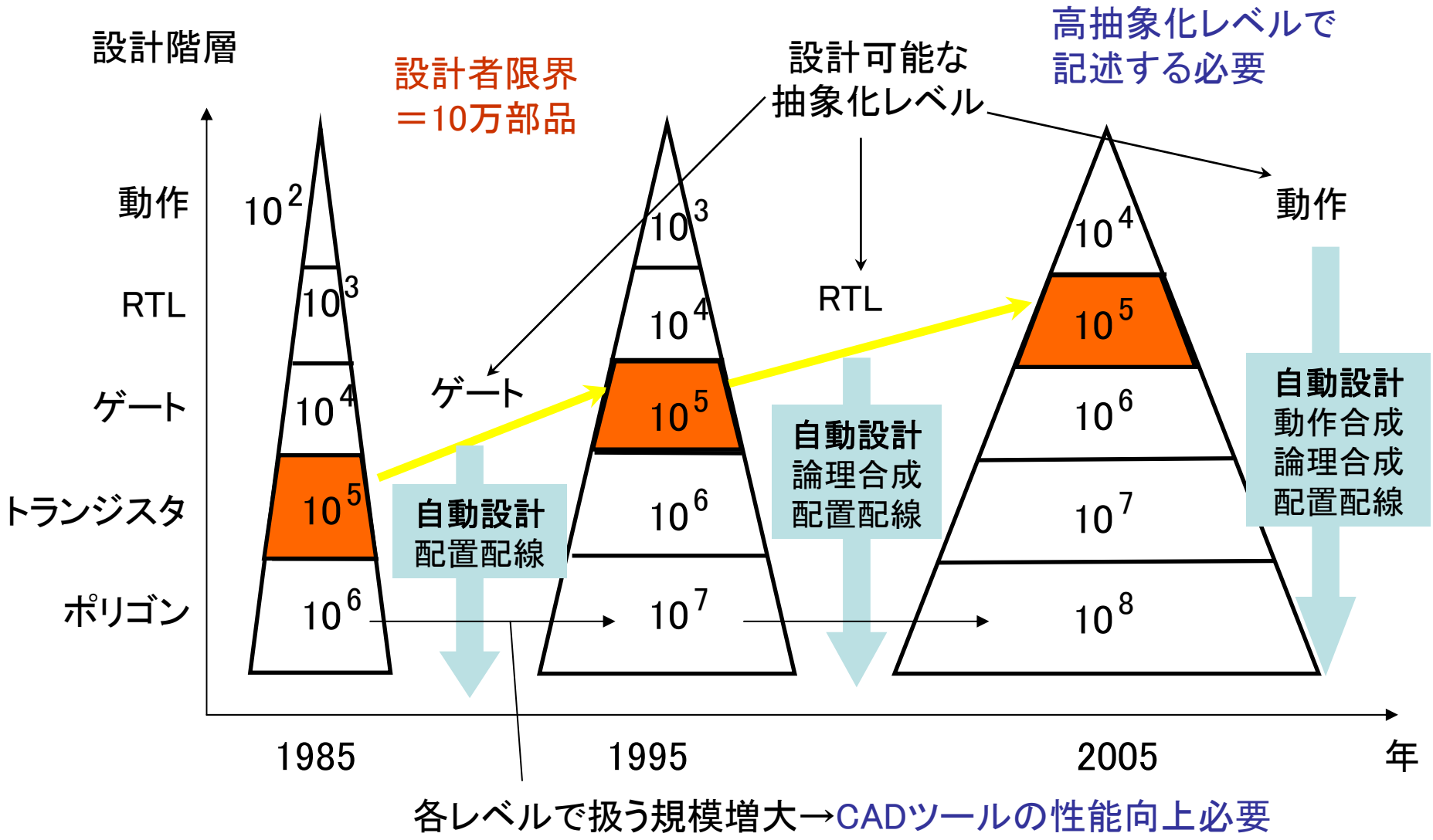


LSI規模の増加と設計生産性



(詳)

設計規模増大と設計可能部品数



4.1　動作記述とRTL記述

4.2　動作合成(1)　原理編

4.2.1　動作合成の意義

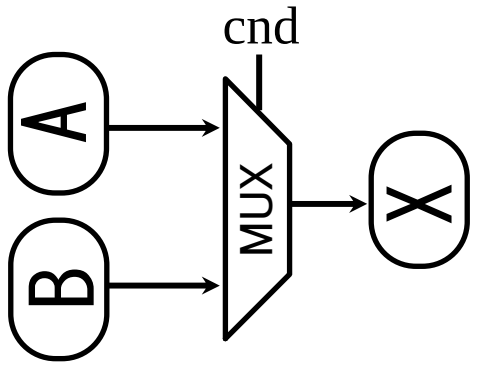
⇨ 4.2.2　動作合成の基礎

4.2.3　動作合成のアルゴリズム

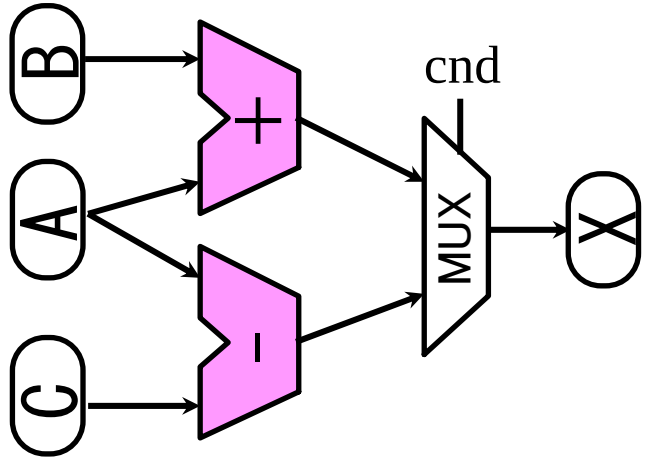
構造レベルC言語とハードウェア合成

C言語記述（構造レベル）

```
if ( cnd )  X = A;
else        X = B;
```




```
if (cnd)   X = A + B;
else       X = A − C;
```



CプログラムからHWを合成

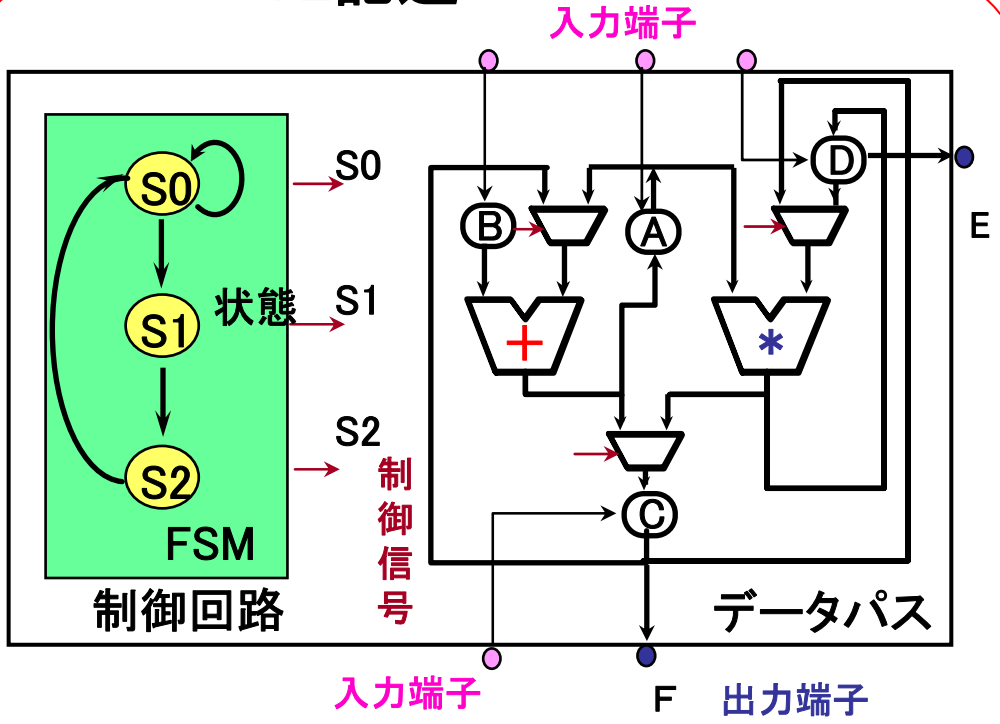
動作記述

```
char A,B,C,D;
char E,F;
main(){
char X;
X = A + B;
E = X * D;
F = (B + C) * X;
}
```

LSIの動作記述
Cプログラム：8行

様々な回路を合成

RTL記述



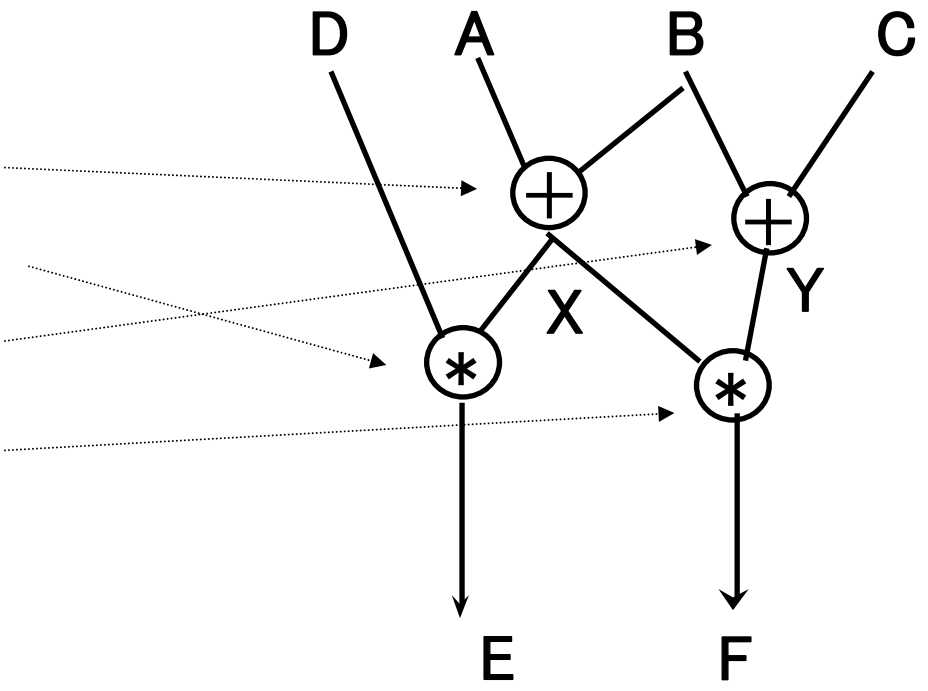
3ステップ、 加算器1個、乗算器1個

合成された回路
（VHDL, VerilogHDLでは100行以上）

C記述からRTL（1）

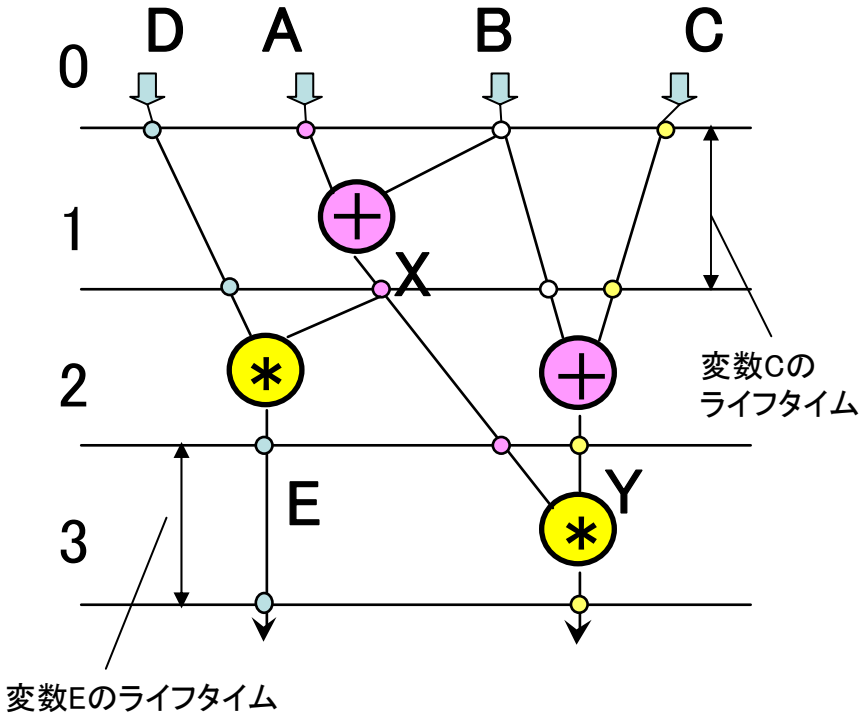
```
X = A + B;
E = X * D;
Y = B + C;
F = Y * X;
```

動作記述



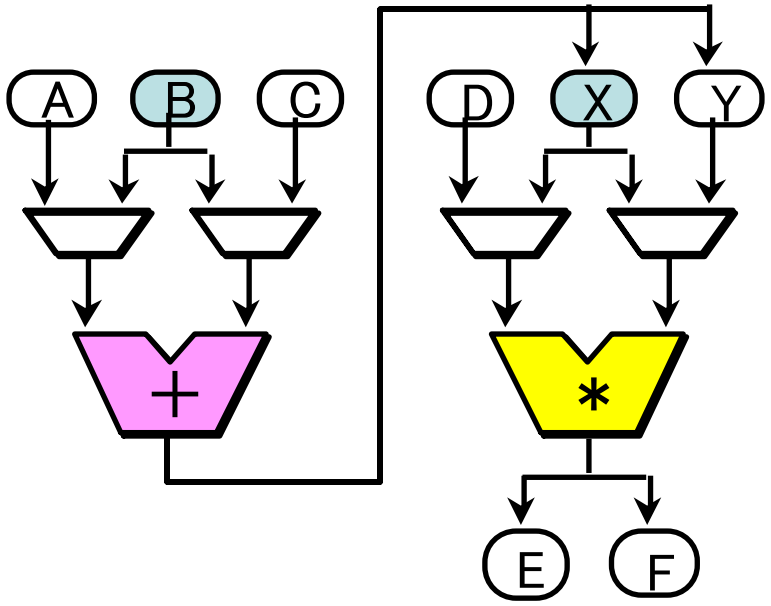
（a）データフローグラフ（DFG）

C記述からRTL （2）



（制約：加算器1、乗算器1）

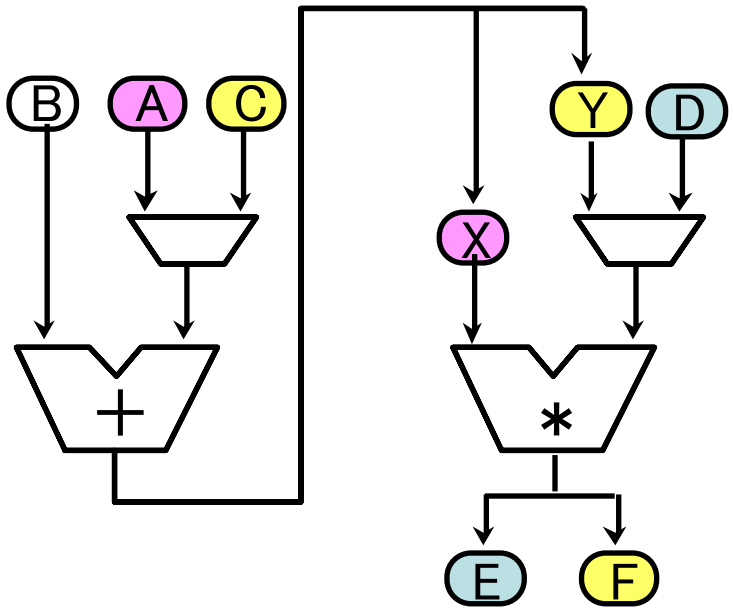
(b)スケジューリング結果



レジスタ8　マルチプレクサ4　通信路12(16)

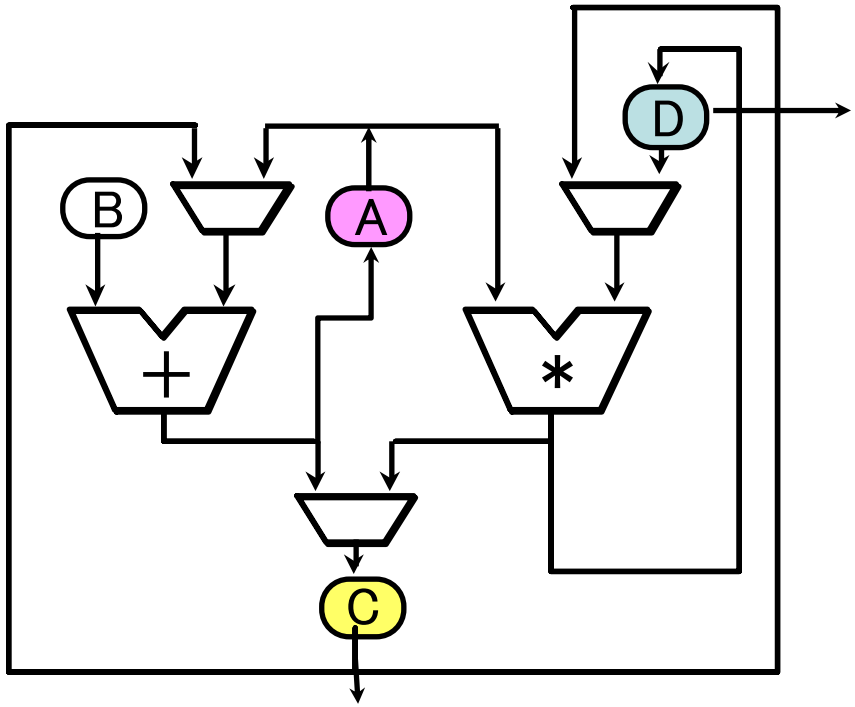
(c)初期データパス

C記述からRTL （3）



レジスタ8　マルチプレクサ2　通信路 10(12)

（d）オペランドの交換

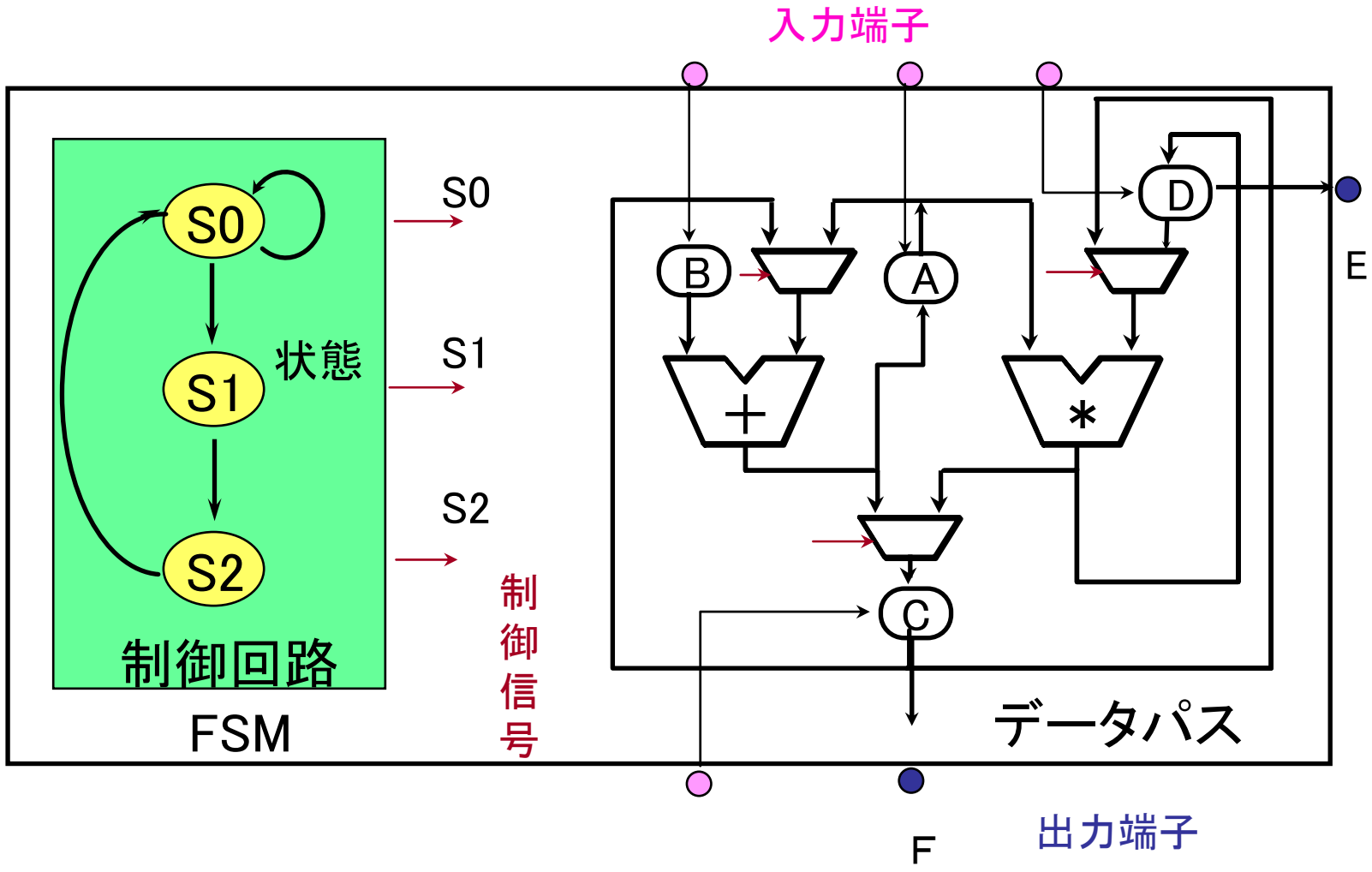


レジスタ 4　マルチプレクサ 3　通信路 9(13)

（e）レジスタの共用

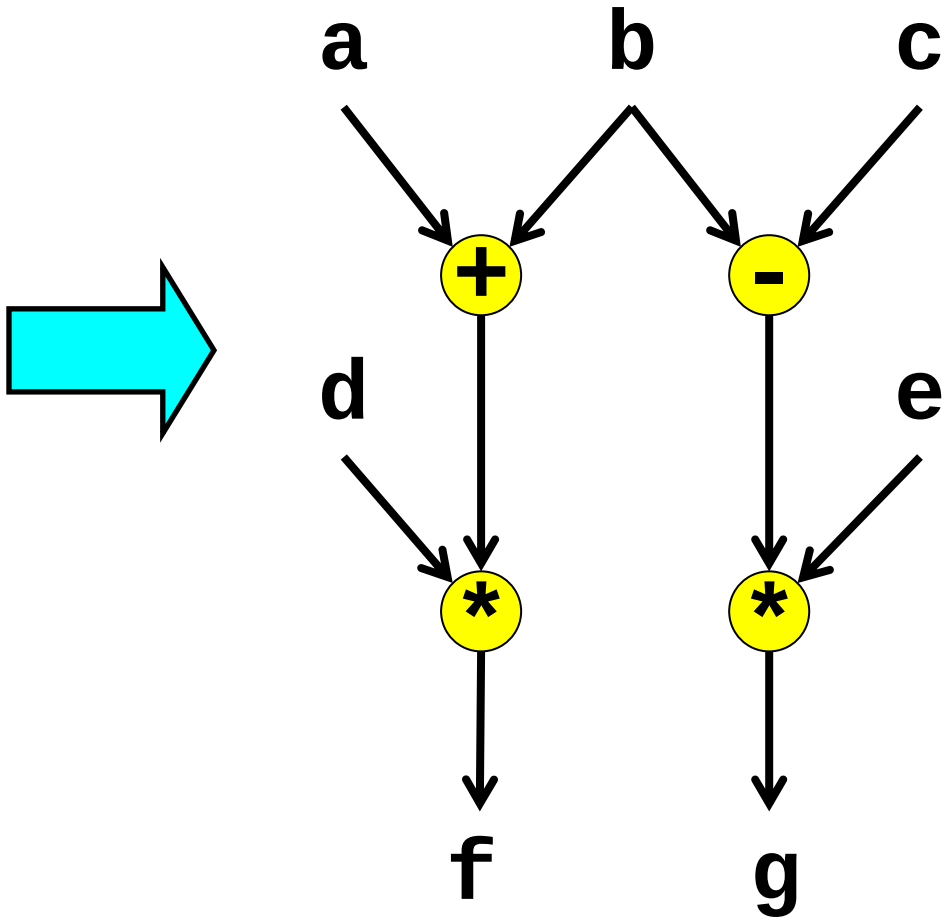
C記述からRTL （4）

＝合成回路　RT図＝



1）DFGへの変換

```
f = (a+b)*d;
g = (b-c)*e;
```



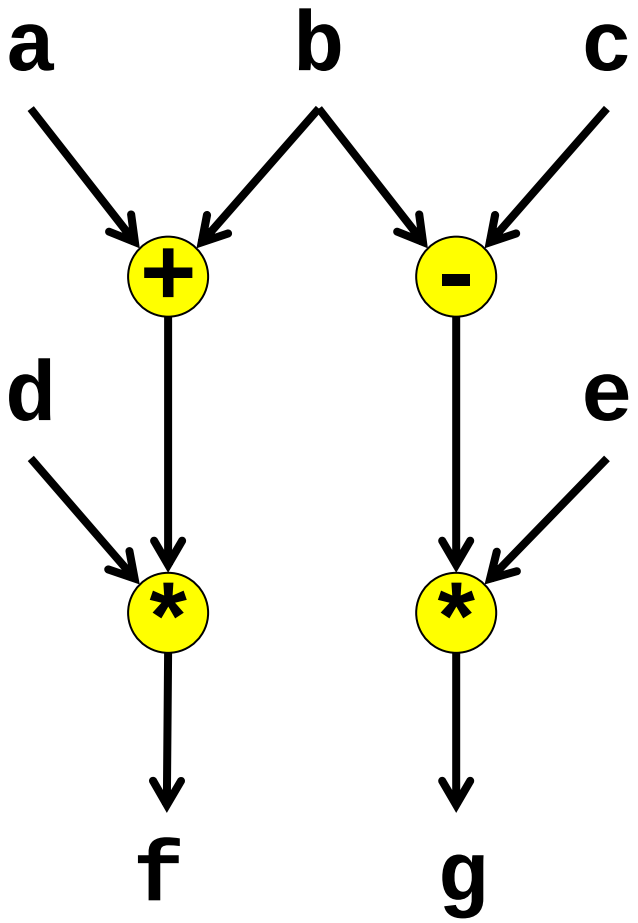
2)アロケーション

- 利用する演算器の選択

アロケーション

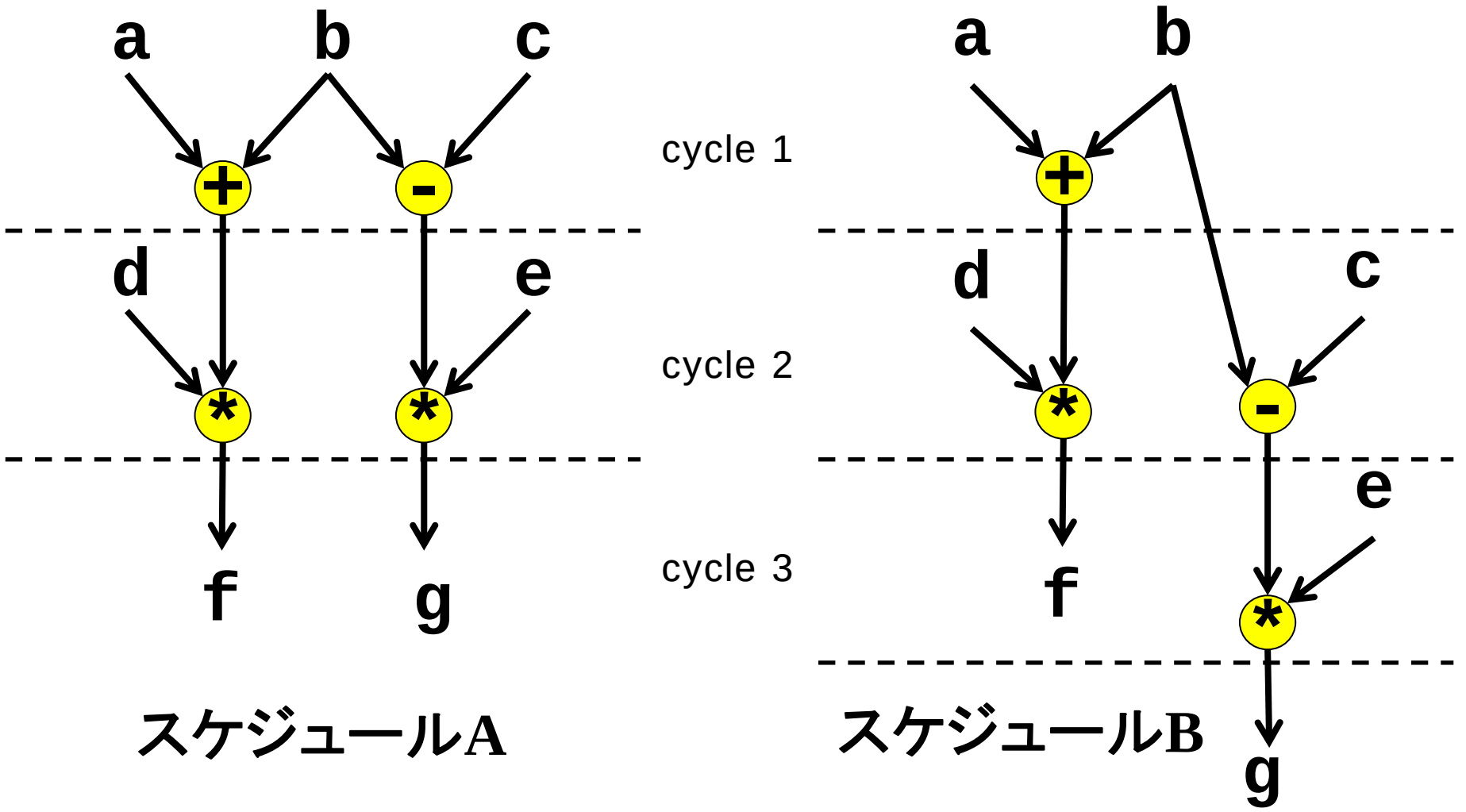
	A	B
Add*cla*	1	
Add*rpl*		
Sub	1	
AddSub		1
Mul	2	1

利用する演算器数



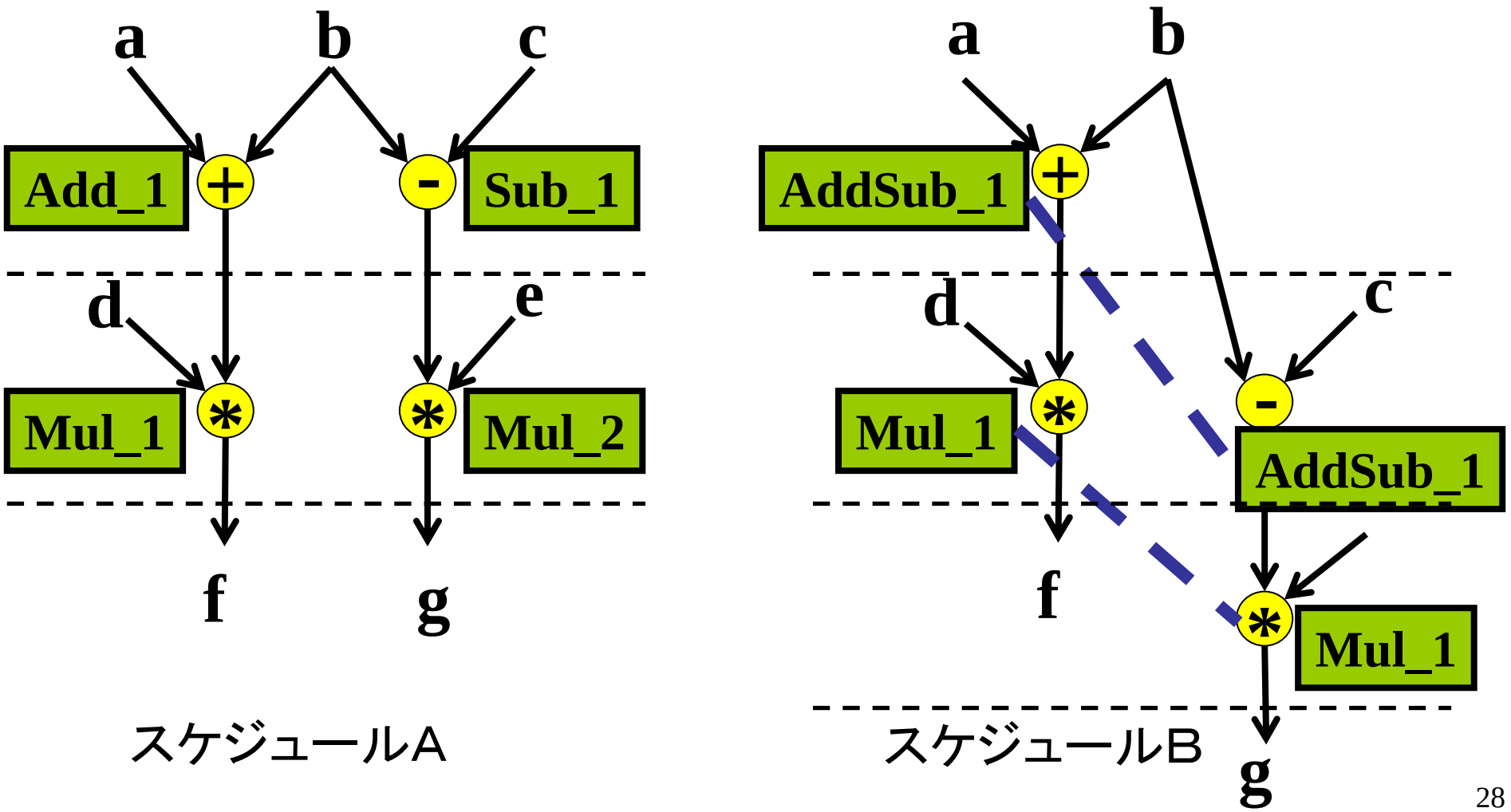
3)スケジューリング

- 演算をステップへ割付(並列化)



4)バインディング（データパス割り当て）

- 演算を演算器に割り当てる



5）転送表作成から、データパス作成

スケジューリングと
バインディング済みCDFG

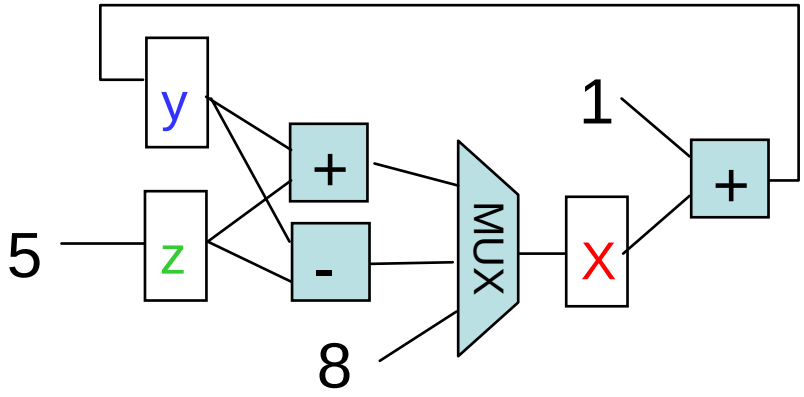
```
S0: if (c1) x=y+z;
      else {  x=y-z;
              z=5;}
       goto S1;

S1: if (c2)  x=8;
      else     y=x+1;
```

x、y、zはレジスタ

レジスタ転送表

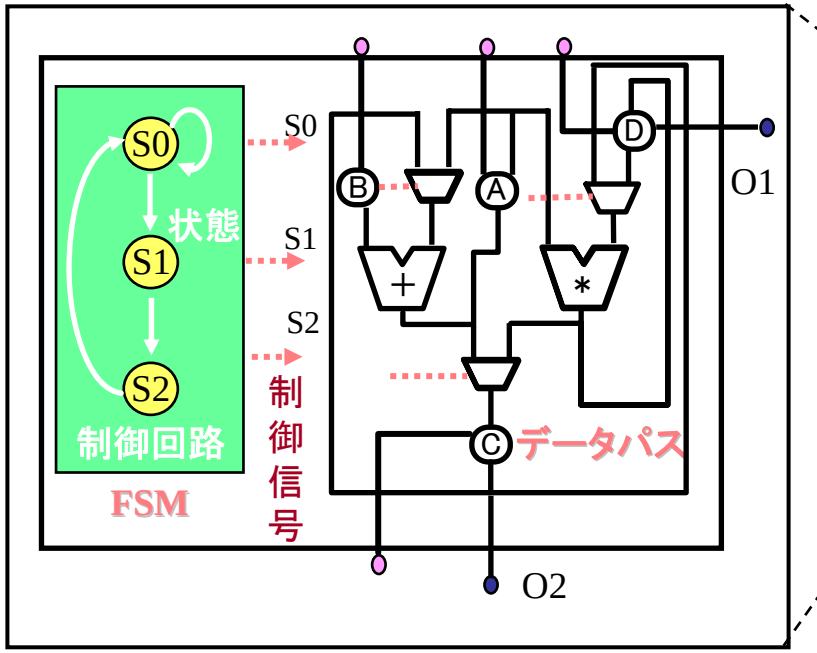
転送先	転送元	状態	条件
x	y + z	S0	c1
x	y - z	S0	!c1
x	8	S1	c2
y	x +1	S1	!c2
z	5	S0	!c1

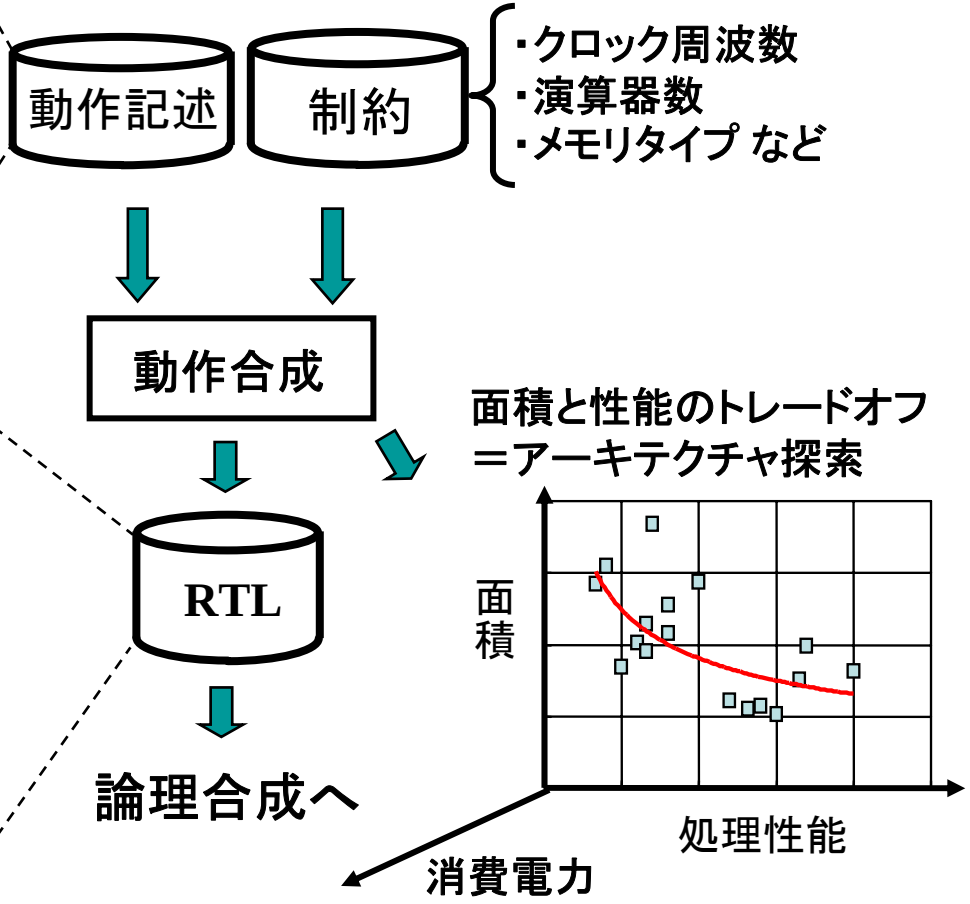
対応するデータパス例



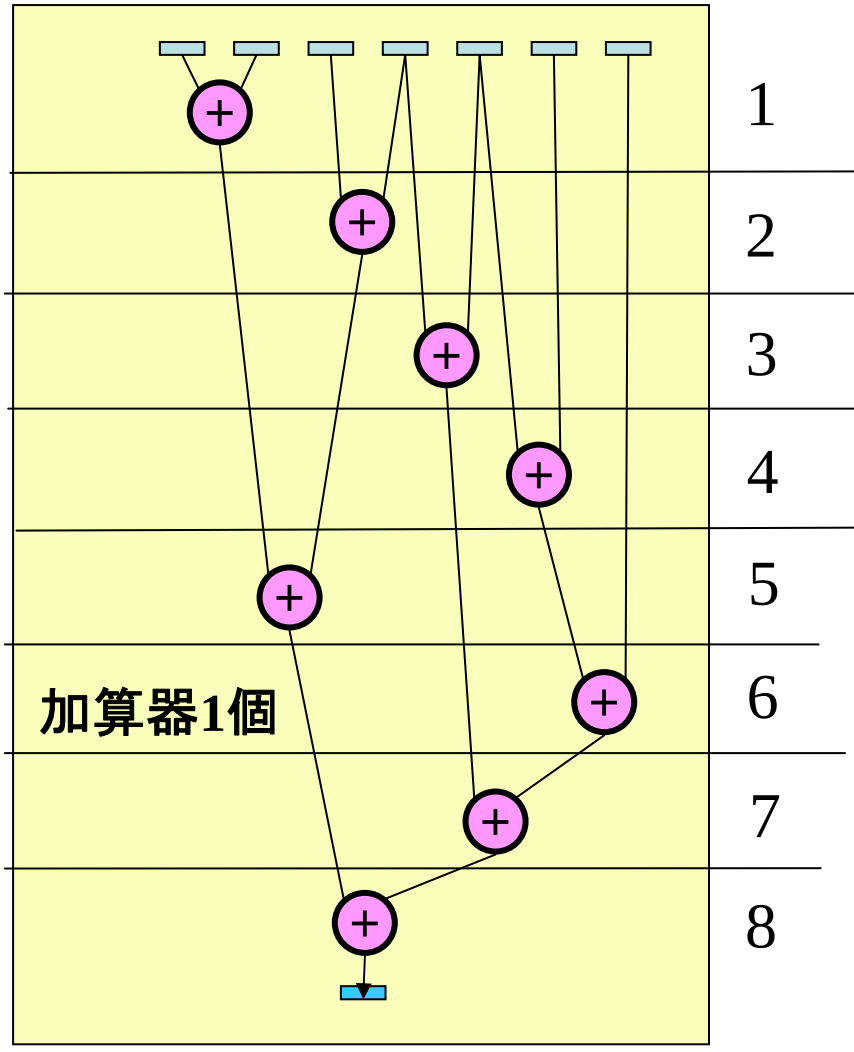
動作合成ツールは合成制約も入力

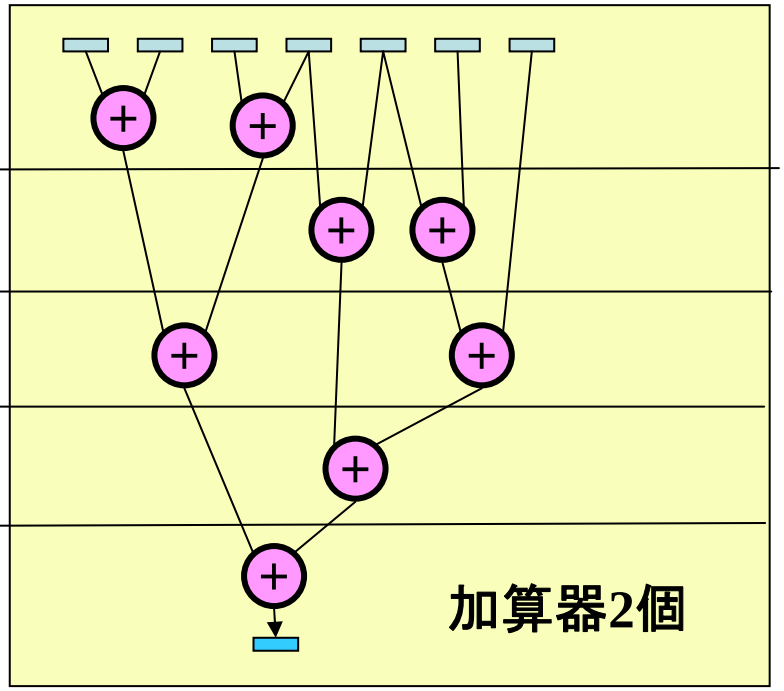
```
in reg(0:8) A,B,C,D;
out reg(0:8) O1,O2;
process ALU ( )
{
    ter(0:8) TMP;

    TMP = A + B;
    O1 = TMP * D;
    O2 = (B + C) * X;
}
```





演算器数の制約を変化させることによって さまざまな回路を自動で合成





性能と面積のトレードオフ

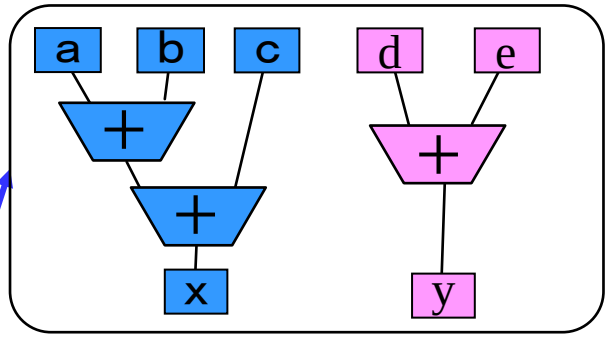
加算器	処理性能	面積
2個	5ステップ	1416ゲート
1個	8ステップ	1227ゲート

さまざまなアーキテクチャの探索

X =(a+b)+c;
Y = d+e;

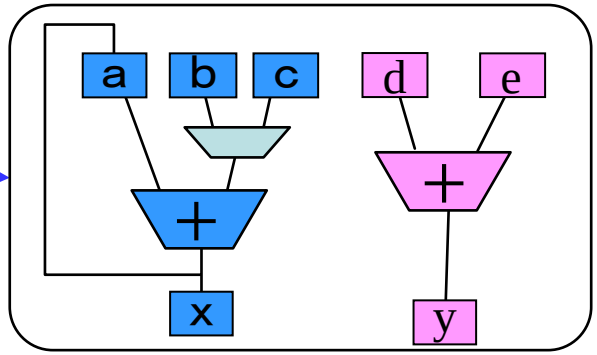
動作記述

動作合成

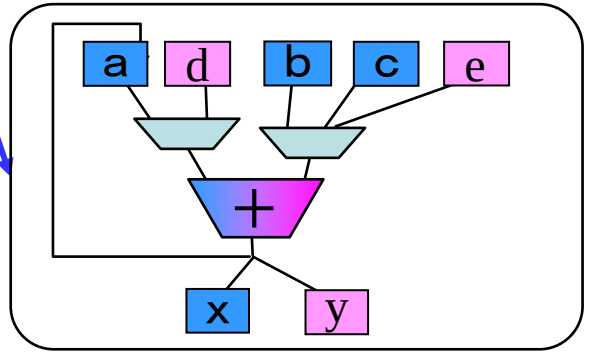


ステップ数
クロック周期

1ステップ
周期2T
加算器3



2ステップ
周期1T
加算器2



3ステップ
周期1T
加算器1

動作記述と構造記述

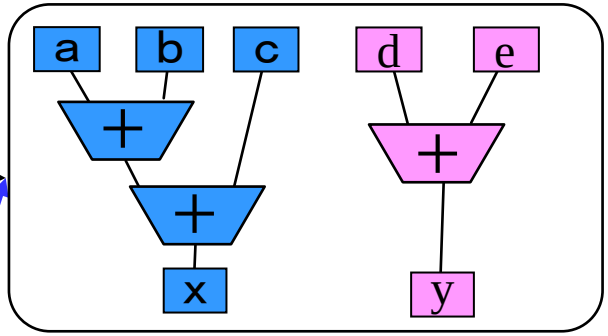
構造記述

動作記述

```
X =(a+b)+c;
Y = d+e;
```

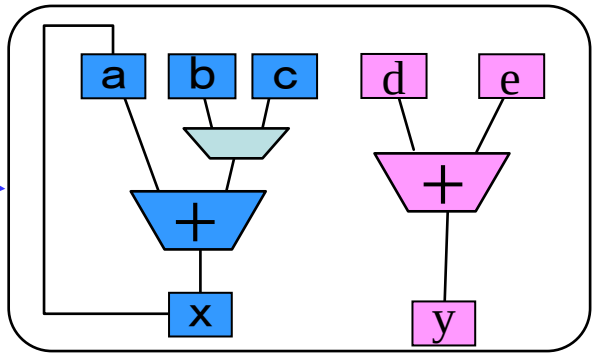
論理合成

動作合成

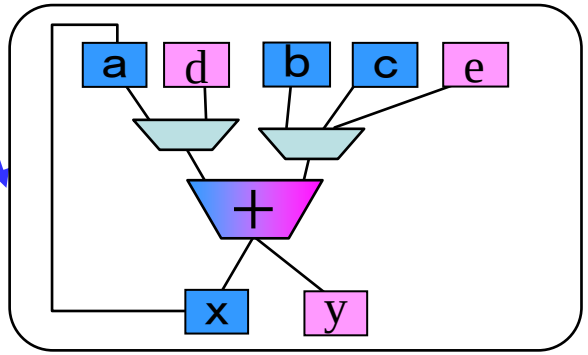


ステップ数
クロック周期

1ステップ
周期2T
加算器3



2ステップ
周期1T
加算器2



3ステップ
周期1T
加算器1

4.1　動作記述とRTL記述

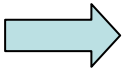
4.2　動作合成(1)　原理編

4.2.1　動作合成の意義

4.2.2　動作合成の基礎

4.2.3　動作合成のアルゴリズム

動作記述から回路を合成する動作合成のステップ



1.言語レベルの最適化

2.コントロールデータフローグラフ作成

3.使用演算器の指定(アロケーション)

4.スケジューリング

5.バインディング

6.制御回路合成とモジュール生成

0)ハード向けC言語拡張例 （BDL言語）

必ず指定

＝入出力とビット幅＝

☆ in var(0..7) I_ch; ー I/O端子の宣言

☆ out var(0..7) o_ch; ー ビット幅の宣言（開始ビット0、ビット幅8）

必要に応じて指定

＝制御、クロック＝

☆ wait（c); ー 条件成立まで停止 外部とのハンドシェーク等

☆ $ ー クロックの明示 外部との通信プロトコル等

＝ハードウェア合成指定＝

☆ reg, ter, var a: ー 特に、入力変数のHW実現方法

☆ 合成指示子〔コメントで〕

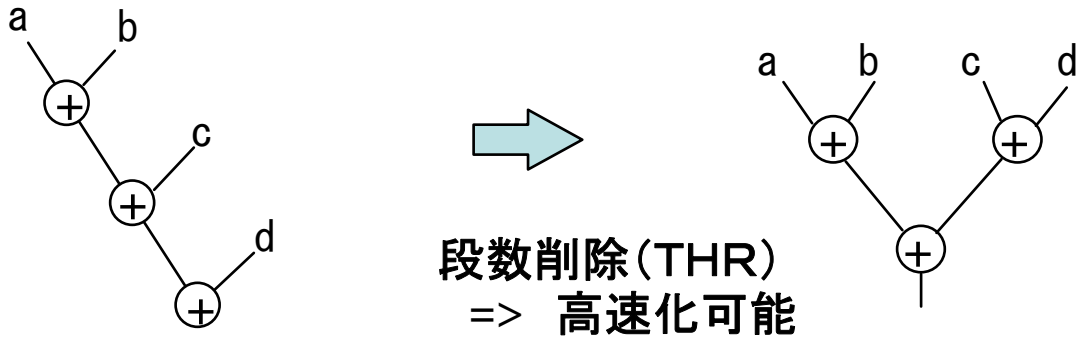
//cyber function=inline #関数をインライン展開

1) 言語レベルの最適化

- 定数伝播　a=1;b=a+2; ⇨　b=3
- デッドコード削除（使用されないコード）
- 共通項の抽出　（演算回数の削減）

  a=(x+y)*z; b=x+y-z; ⇨ t=x+y; a=t*z;b=t-z;
- 関数のインライン展開
- ループの展開　FOR文の展開（後述）
- CDFGの変形　交換則、結合則、分配則等を利用

  バランス化 ～　演算順序の変更　（演算精度が変化）



動作記述から回路を合成する動作合成のステップ

1. 言語レベルの最適化
2. コントロールデータフローグラフ作成
3. 使用演算器の指定(アロケーション)
4. スケジューリング
5. バインディング
6. 制御回路合成とモジュール生成

2)動作記述から内部データ構造へ

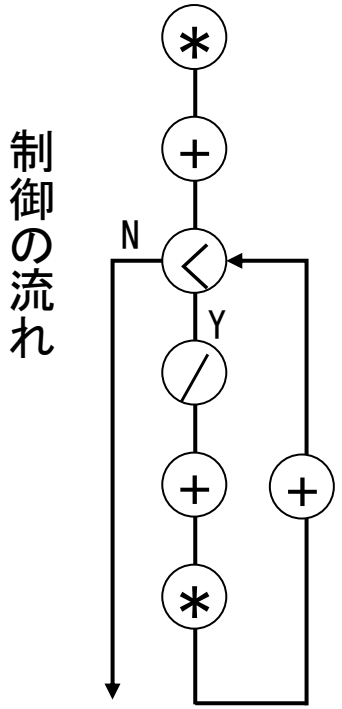
動作記述

```
Y=1+2*X;
I=0;
while (I<3){
   Y=0.5*(Y+X/Y);
   I=I+1;
}
```

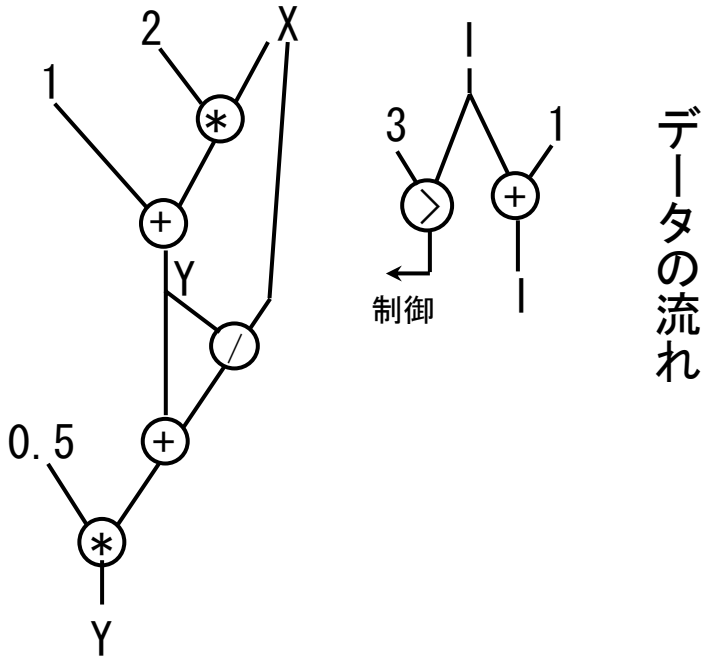
制御とデータの流れを
どのようにに保持するか？

→それぞれをグラフで

コントロールフローグラフ



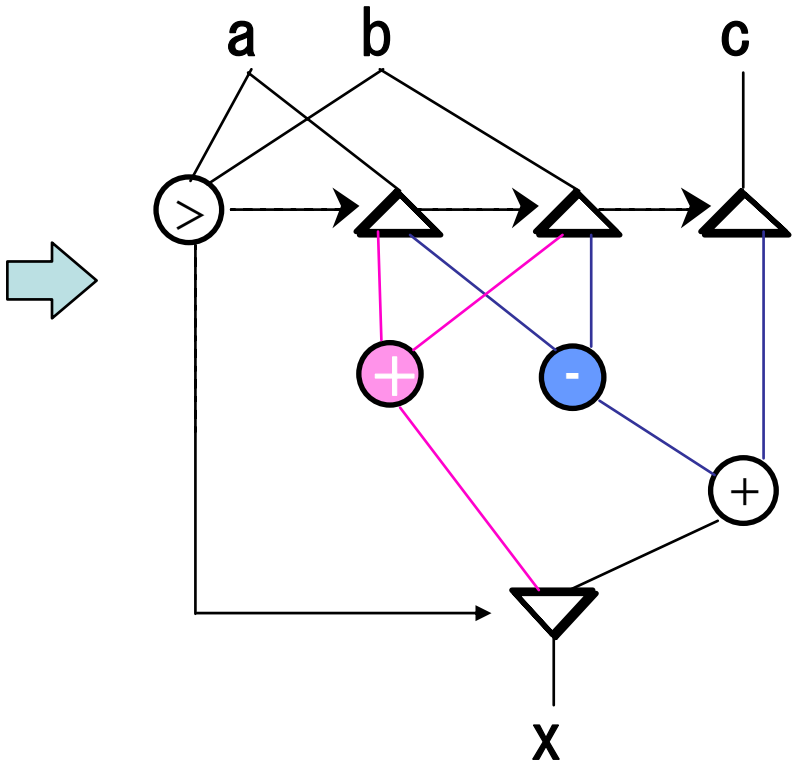
データフローグラフ



コントロールデータフローグラフ(CDFG)

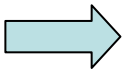
制御とデータの依存関係を同時に表現

```
if (a>b)    x = a+b;
else        x = a-b+c;
```



CDFGには様々な形式がある。制御構造の持ち方によって違ってくる

動作記述から回路を合成する動作合成のステップ



1.言語レベルの最適化

2.コントロールデータフローグラフ作成

3.使用演算器の指定(アロケーション)

4.スケジューリング

5.バインディング

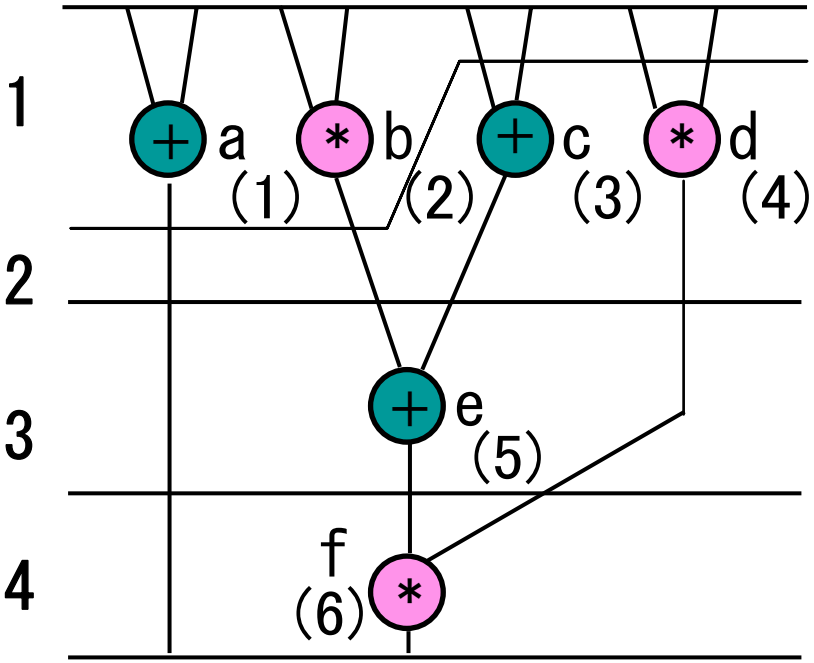
6.制御回路合成とモジュール生成

3)スケジューリング
制約と目標

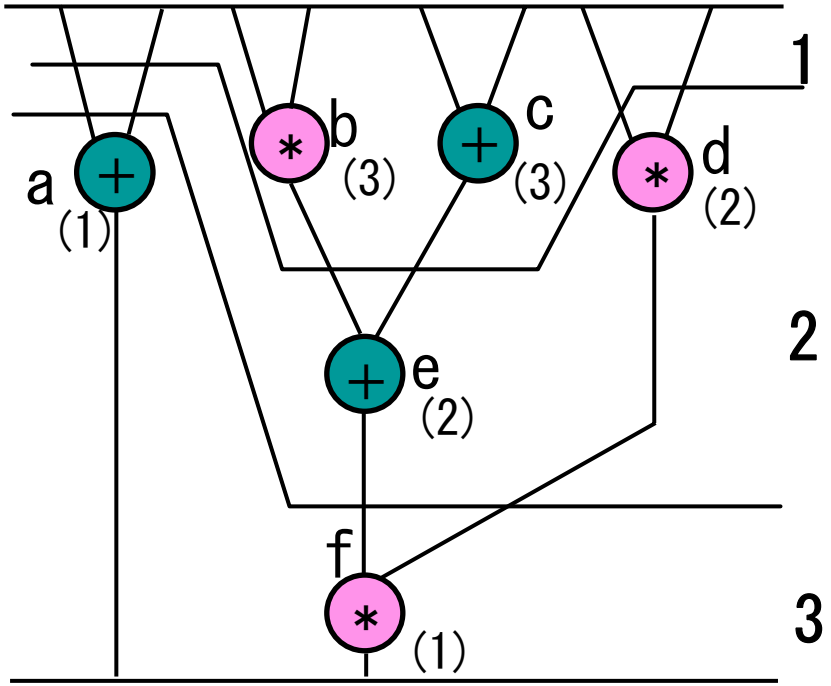
制 約		目 標
A)面積制約スケジューリング		
・ハードウェア量（面積）	→	ステップ数最小（速度）
		→並列性の抽出
B)時間制約スケジューリング		
・ステップ数（速度）	→	ハードウェア量最小（面積）
		→相互排他性の抽出(演算資源を共用)

面積制約スケジューリング（リストスケジューリング）

演算器制約：＋ 1個　＊ 1個



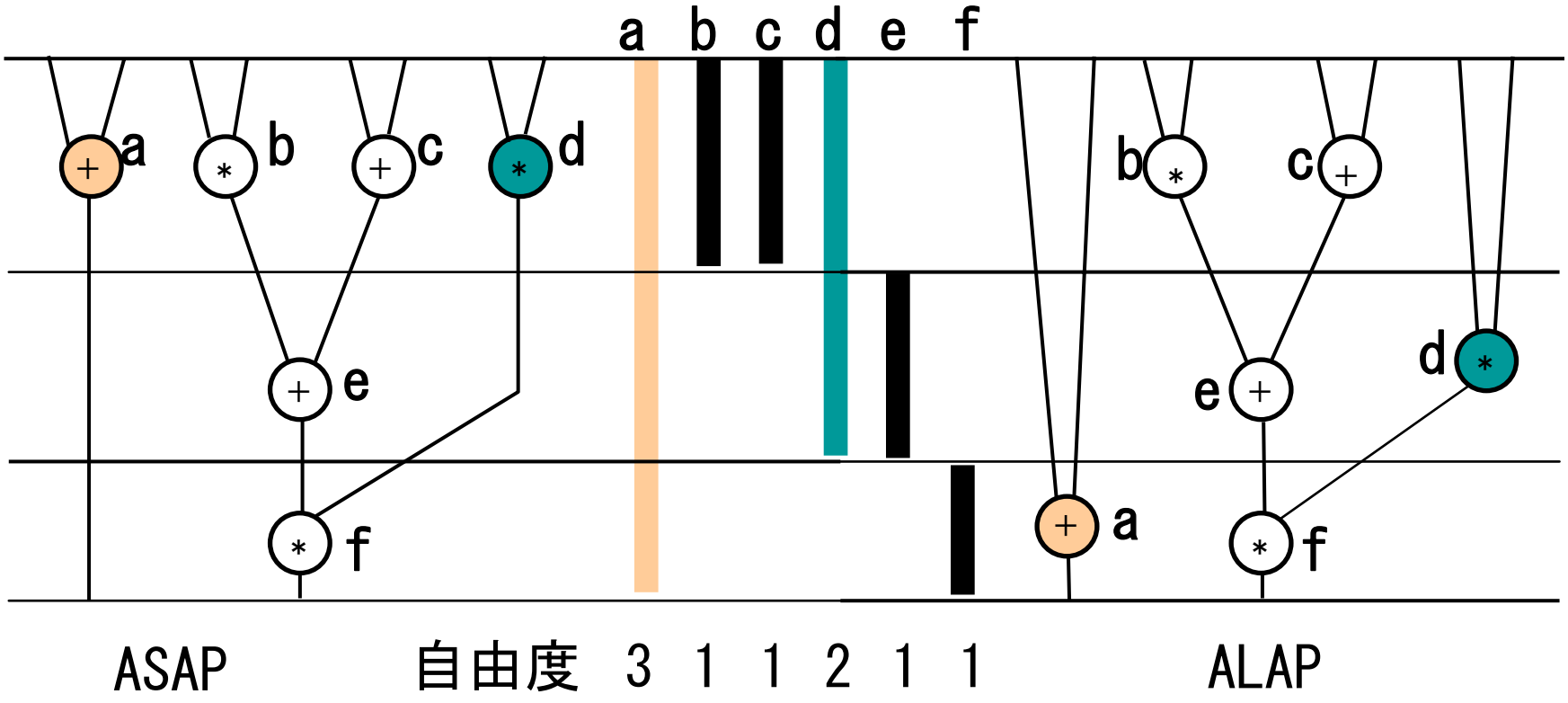
選択基準：
動作記述順



選択基準：
最終ノードからの距離順

時間制約スケジューリング

時間制約：3ステップ



演算選択記述: 演算の自由度

例題1　CDFGとスケジューリング

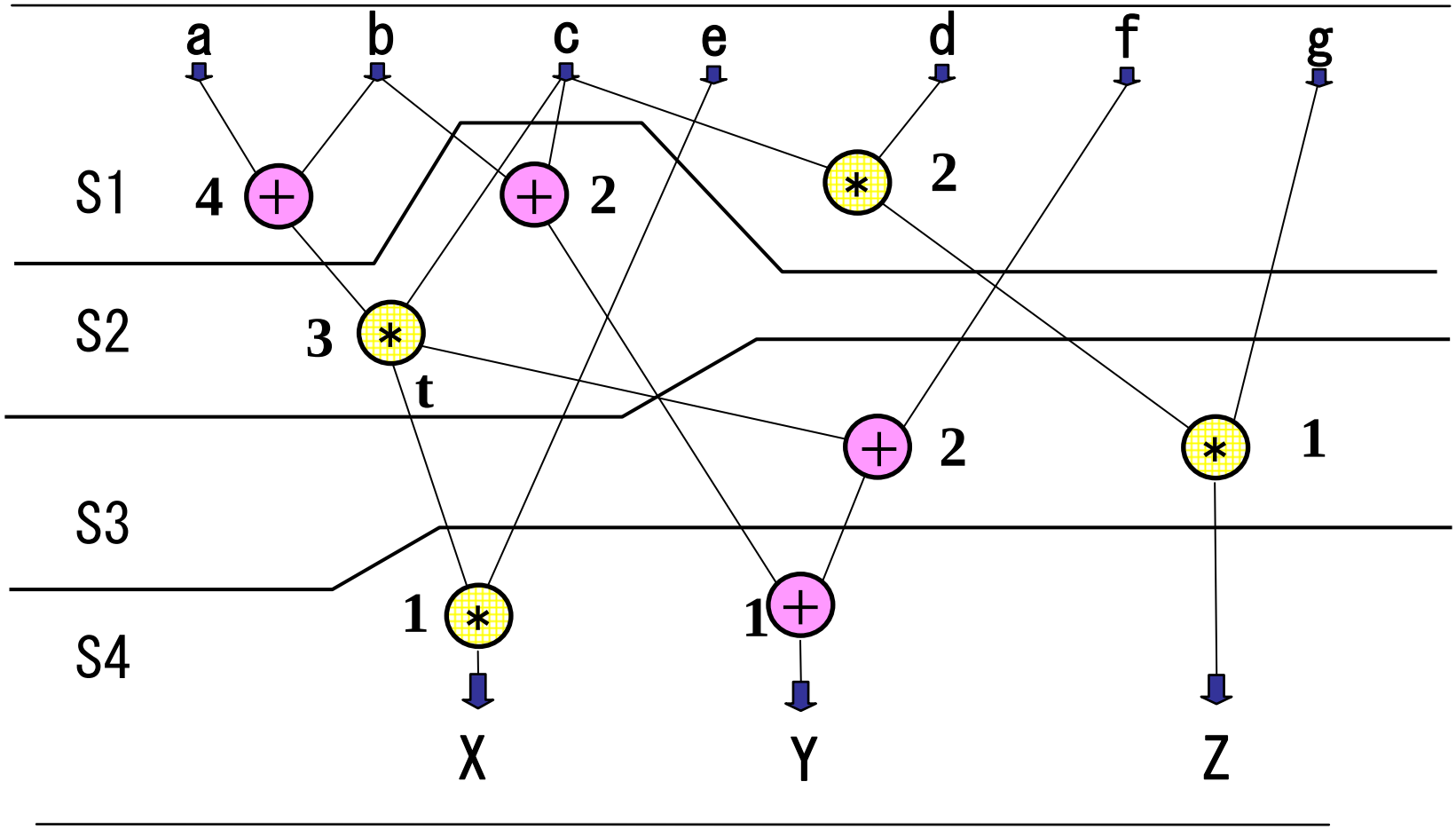
=動作記述1=	t=(a+b) * c;
	x=t*e;
	y=(b+c) + (t+f);
	z=(c*d)*g;
		a, b, c, d, e, f, gは外部入力
		x, y, zは出力

（問1）　動作記述1からCDFGを作成し、スケジューリングせよ。
但し、演算器は加算器、乗算器を1つずつ利用可能、
両演算とも1ステップで実行可能。
aーgの変数が第1ステップで全て値を入力されているとする。
→　リストスケジューリングで、出力から遠いノードから選択という戦略で
最速の回路を求めよ。

（問2 -1）　乗算器の実行に2ステップかかる場合のスケジューリングをせよ。
（問2 -2）　乗算器が2ステージパイプラインの時はどうか（1ステージは1ステップ）？

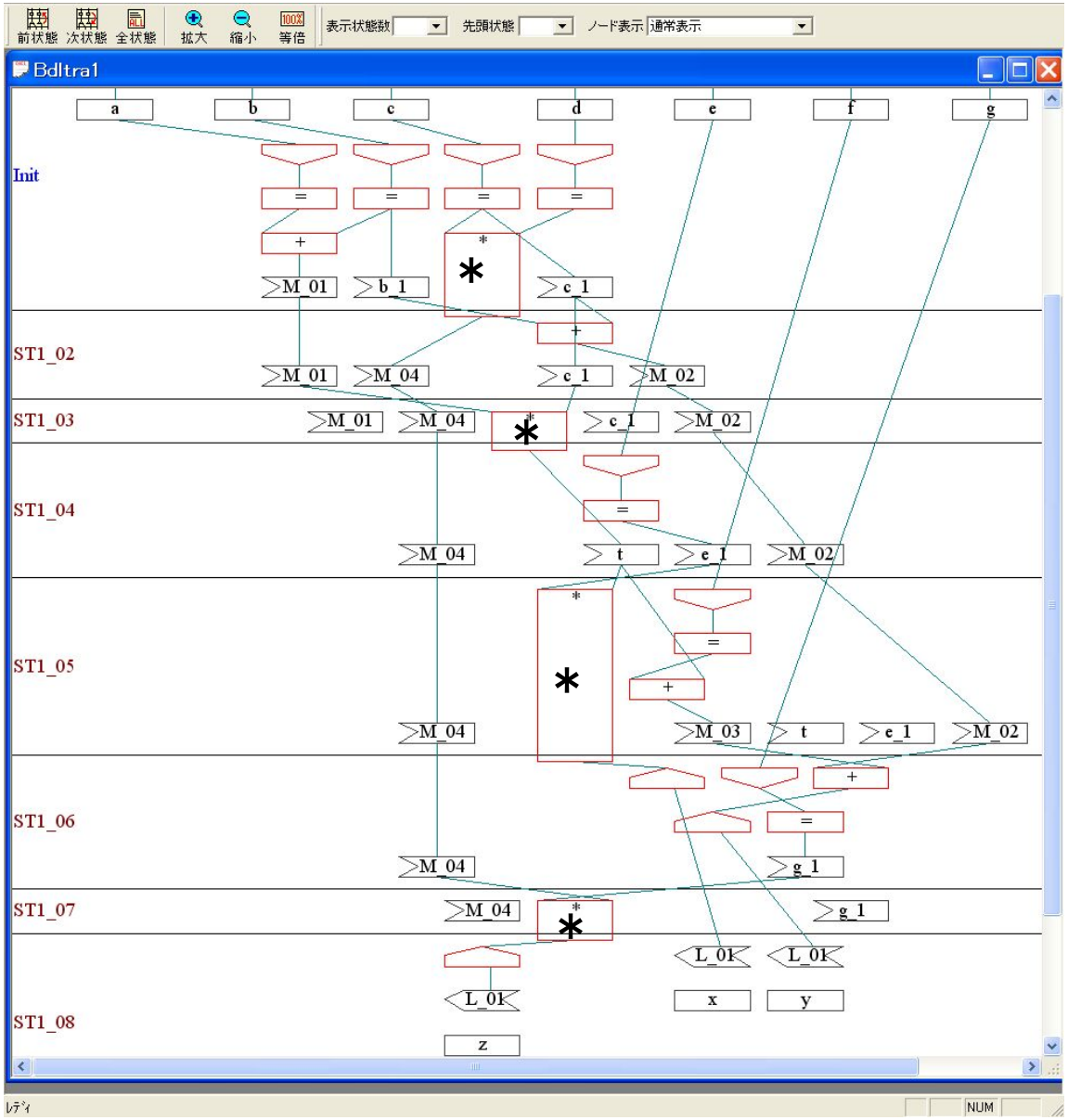
（問3）（問1）の条件で、入力端子が2つ、出力端子が1つの場合のスケジューリング
をせよ。

例題1 （問1）解答



例題1 （問2-1）解答：2ステップ乗算

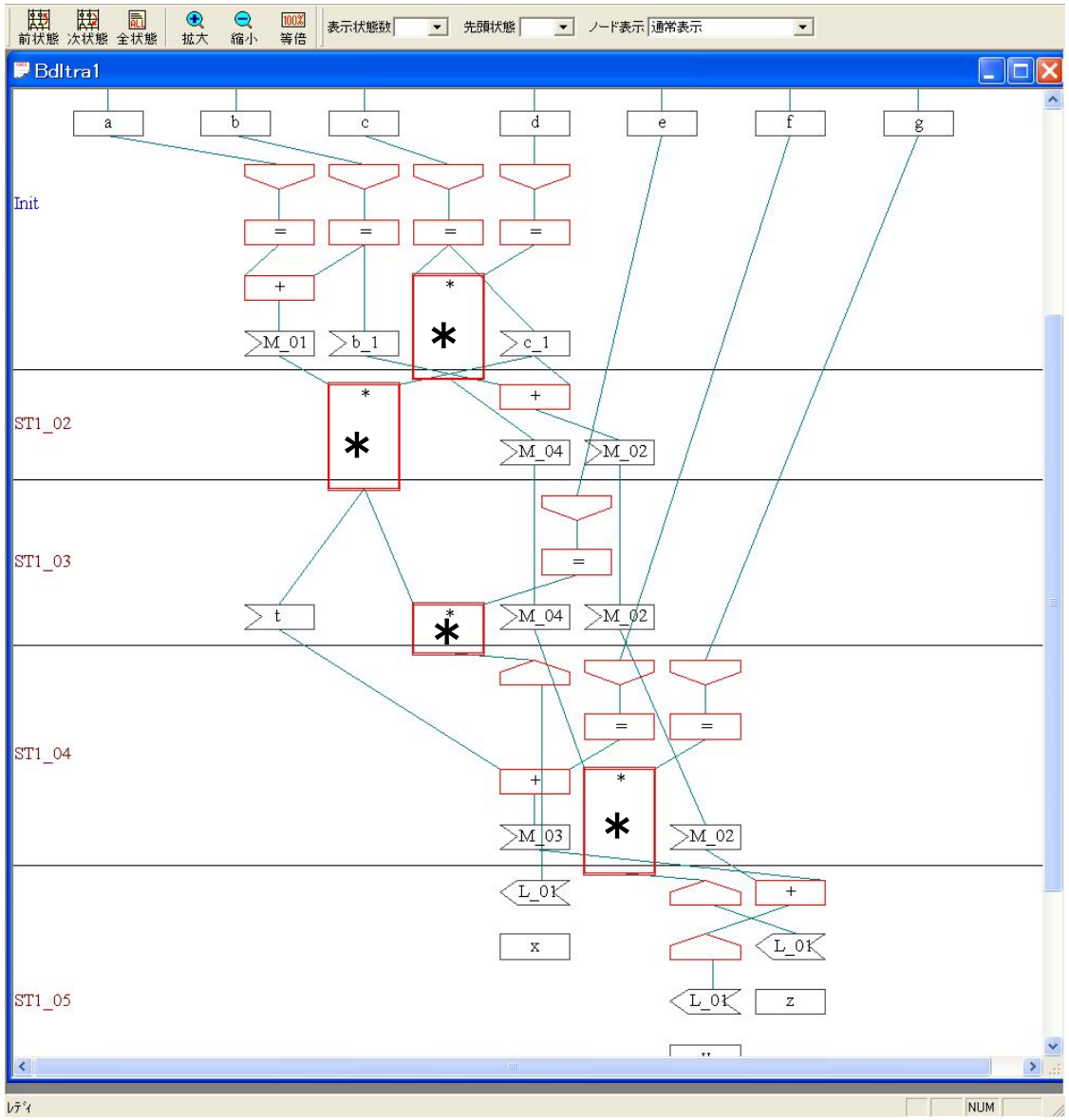
（詳）



8ステップ

例題1 （問2-2）解答：パイプライン乗算

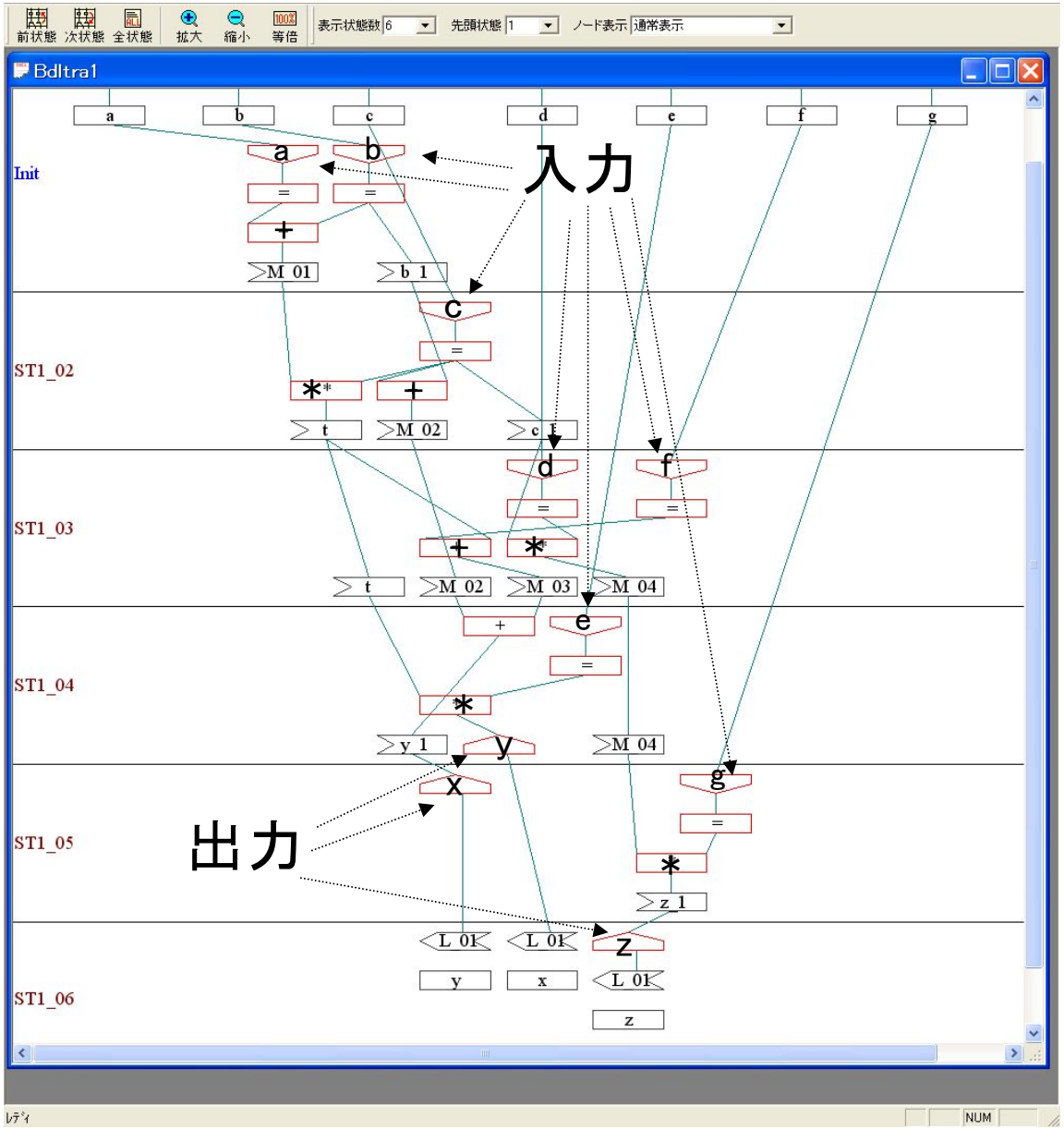
（詳）



5ステップ

例題1 (問3)解答例: 入出力の制限

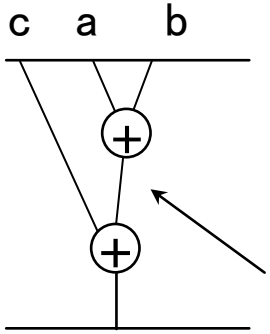
(詳)



6ステップ

現実的なスケジューリング問題:種々の演算1

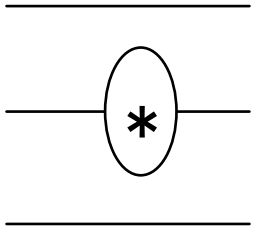
1. 演算のチェーン



1つの制御ステップに
複数の演算をスケジュール

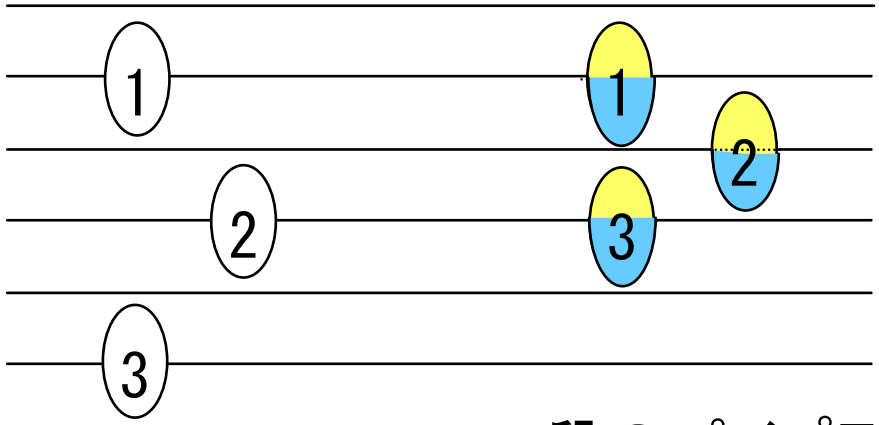
レジスタをはさまない

2. 多クロック演算



入力を2クロック保持

3. 演算パイプライン



2クロック演算1つ　　:6ステップ
2段パイプライン1つ　:4ステップ

2クロック演算　　2段のパイプライン演算

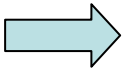
動作記述から回路を合成する動作合成のステップ

1.言語レベルの最適化

2.コントロールデータフローグラフ作成

3.使用演算器の指定(アロケーション)

4.スケジューリング



5.バインディング

6.制御回路合成とモジュール生成

データパス割り当て(バインディング)

スケジュール結果を実行するデータパスを合成

使用可能な演算器はスケジューリングで指定してある

演算をデータパス要素に割当

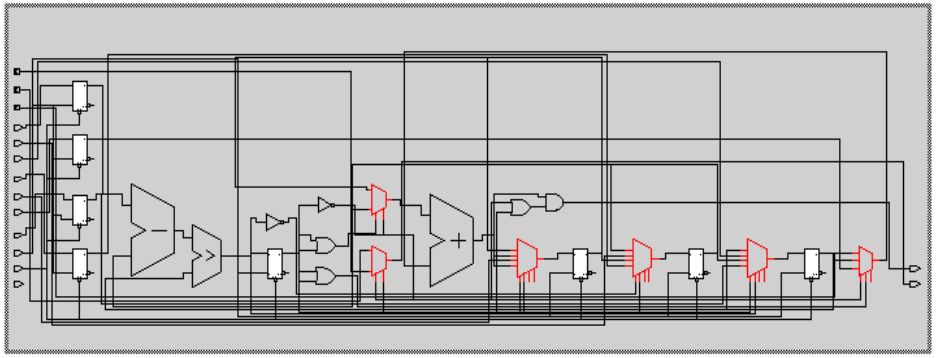
演算	➔	演算器(加算器、ALU、シフタ等)
変数、配列	➔	レジスタ・メモリ
データ転送	➔	結線・マルチプレクサ・バス

Q スケジュール済みCDFGに加算演算が10個
→2つの加算器にどう割り振るか

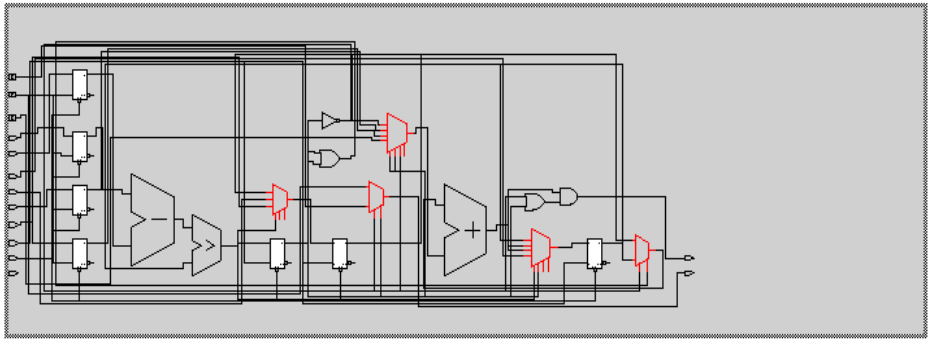
データパス最適化の例

（省）

割り当て最適化によりマルチプレクサ数等を最小化



レジスタ数	8
MUX数（入力数計）	6（19）
セル数［論理合成後］	2049

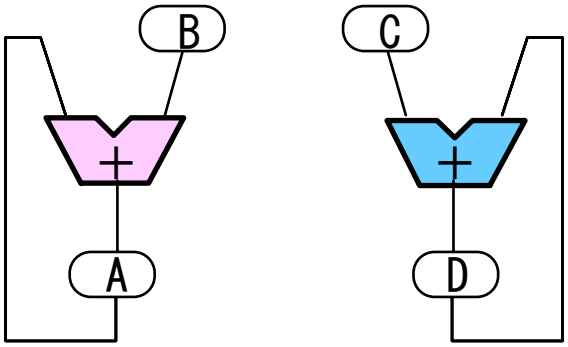


レジスタ数	7
MUX数（入力数計）	5（16）
セル数［論理合成後］	1834

逐次割り当てアルゴリズム

演算器割り当ての例

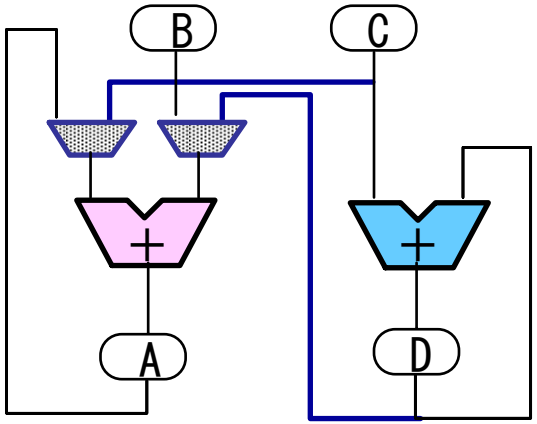
構築中の
データパス →



A=C+Dを割り当て

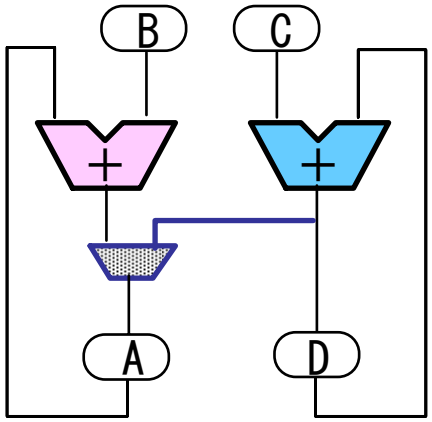
コスト増
MUX2
結線2

左の＋へ



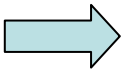
右の＋へ

コスト増
MUX1
結線1

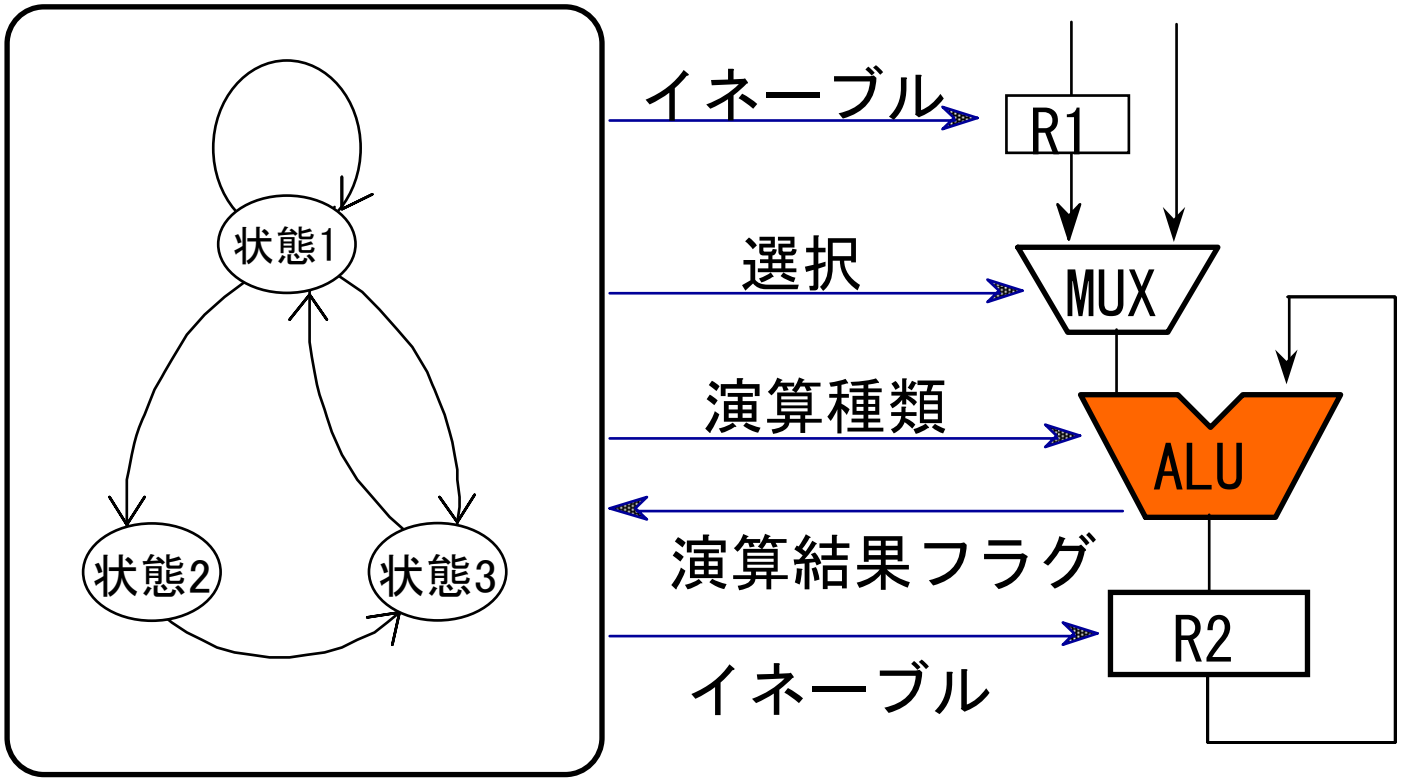


動作記述から回路を合成する動作合成のステップ

1. 言語レベルの最適化
2. コントロールデータフローグラフ作成
3. 使用演算器の指定（アロケーション）
4. スケジューリング
5. バインディング
6. 制御回路合成とモジュール生成

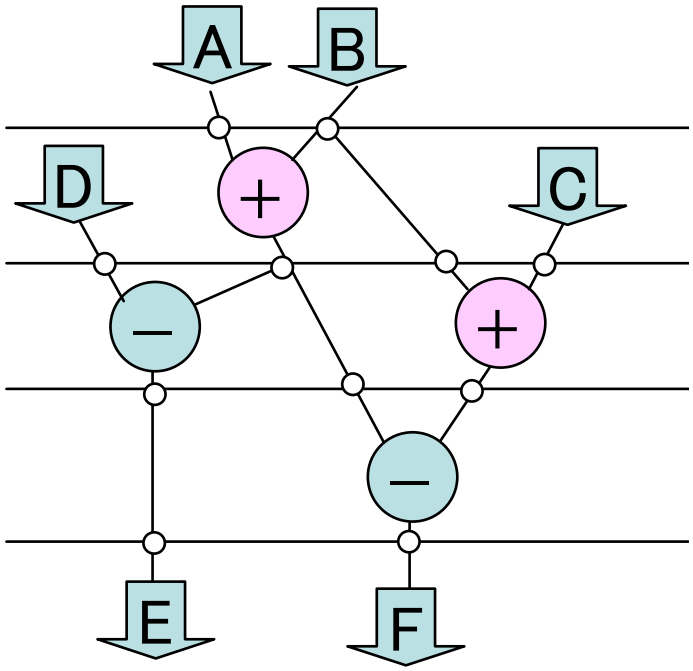


制御回路とデータパスの結合



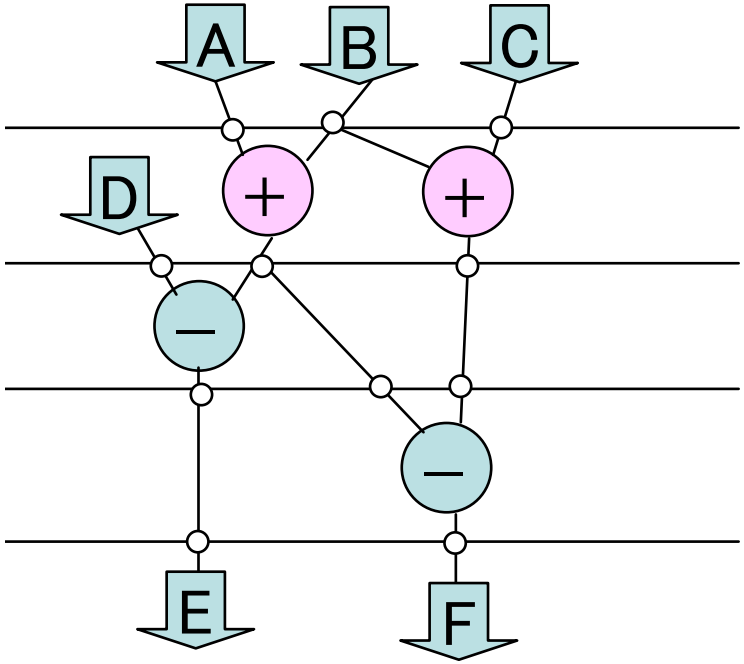
制御回路 ——— 制御信号 ——— データパス

最適な回路は、利用される状況で決まる



演算器 ＋ 1 − 1

3ステップ
レジスタ *4*
ポート *2*
配線 *10*
MUX *2*



演算器 ＋ *2* − 1

3ステップ
レジスタ **3**
ポート **3**
配線 **8**
MUX **1**

例題2

動作合成手法を用いて回路を合成せよ。

入力　a,b,c
出力　x,y

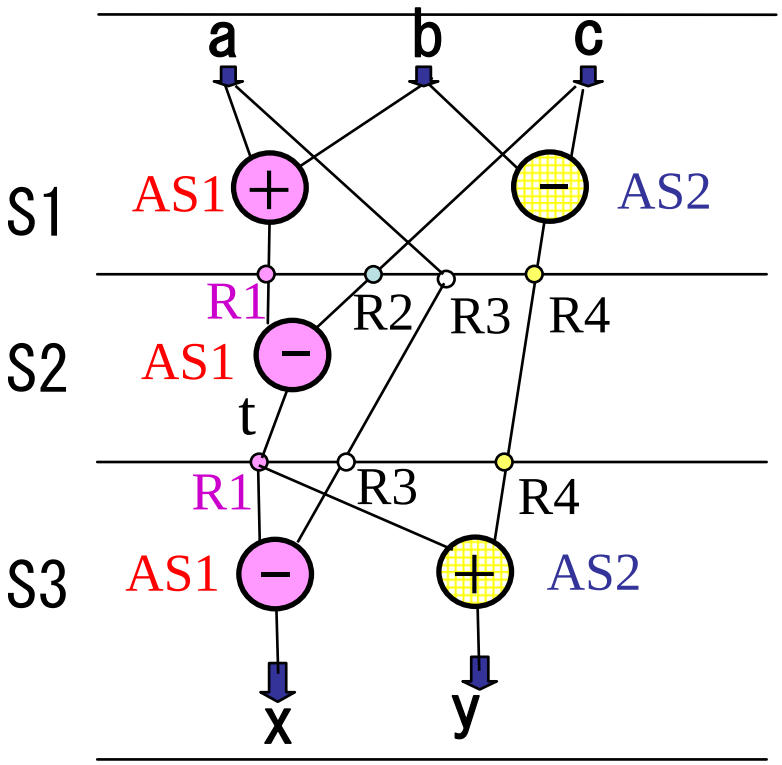
```
t = (a + b) - c;
x = t - a;
y = (b - c) + t;
```

合成制約：演算器は**加減算器を2つ使用可能**

演算器の遅延：　ステップ周期よりやや少ない程度
入力したサイクルですぐに加減算可能とする。
演算結果はそのサイクルで出力可能とする。

（問1）スケジューリングを行い、CDFGを描け。
（問2）CDFG上でデータパス割り当てを行い、
　　転送表を作成し、データパスを描け。
☆仮定：a + b − a = b 等の言語レベル最適化はしない。

例題2 （問1） 解答例



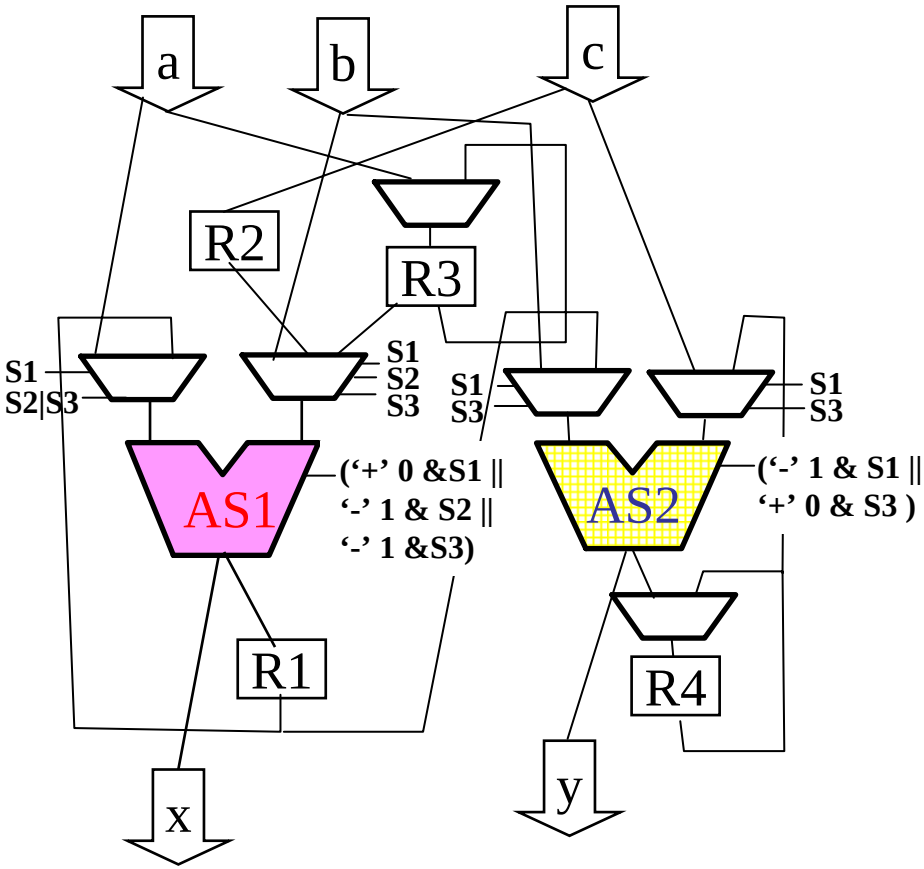
転送先	転送元	条件	状態
R1	AS1.out	1	S1
	AS1.out	1	S2
R2	c	1	S1
R3	a	1	S1
	R3	1	S2
R4	AS2.out	1	S1
	R4	1	S2

X	AS1.out	1	S3
Y	AS2.out	1	S3

例題2 （問2） 解答例

転送先	転送元	条件	状態
AS1.左	a	1	S1
	R1	1	S2
		1	S3
AS1.右	b	1	S1
	R2	1	S2
	R3	1	S3
AS1.種類	+ (0)	1	S1
	- (1)	1	S2
	- (1)	1	S3

AS2.左	b	1	S1
	R1	1	S3
AS2.右	c	1	S1
	R4	1	S3
AS2.種類	- (1)	1	S1
	+ (0)	1	S3



データパス例
REG:4, MUX入力数:13, FU:2

動作合成(1) 原理編 まとめ

- 動作合成は、動作記述(C言語等)から、RTL記述(ブロック図)を合成するものである。
- 動作合成は、人手の機能設計より、設計工数、設計期間を大幅に削減できるだけでなく、面積や、性能、消費電力についても優れた回路を合成可能である。これは、沢山のアーキテクチャから回路を選択可能になるからである。
- 動作合成の本質は、設計者がより多くのアーキテクチャを探索可能とすることにある。
- 動作合成は一部の企業では実用設計に大いに利用されている。しかし、一般にはまだ普及していない。日本は、動作合成の実用化では世界の最先端を走っている。

用語・略語(テクニカルターム)

用語・略語	フルスペル	解　　説
マイクロアーキテクチャ		演算器やレジスタ、バス本数等が陽にあらわれるレベルの回路構成。
CDFG		[重要]コントロールデータフローグラフ。制御の流れ(コントロールフロー)とデータの流れ(データフロー)の両方をひとつのグラフで表現したもの。
スケジューリング		[重要]動作記述上の演算ノードを実行ステップへ割りつけること(演算実行のステップを決定する)。
アロケーション		[重要]利用可能な演算器の種類、ビット幅、数などを決定すること。
バインディング		[重要]動作記述に存在する演算を実行する実際の演算器を決定すること。
構造記述		RTL記述のことをさすことが多いが、レジスタや演算器等のつながりを示すこともある。
動作記述		ハードウェアの構造を陽には示さず、動きやアルゴリズムを示す。CやC++等のソフトウェア向け言語で書かれることが多い。
FSM	Finite State Machine (有限状態機械)	状態数が決まった数から変化しない状態遷移機械。動作合成では、制御回路をFSMで表現される順序機械で実現することが多い。
制御信号		制御回路からデータパスを制御する信号線。セレクタの入力信号や、ALUの演算種類を指示したりする。
レジスタ転送表		レジスタの転送関係を示す表。転送先と転送元、その転送が起こる条件などからなる。
演算チェーン(チェイン)		二つ以上の演算器をレジスタを介さずに直結すること。乗算と加算を直結する積和演算器が有名。演算チェーンによって、それぞれの演算を実行するのに必要な時間の和より、短い時間で演算全体を実行できることが多い。

用語・略語	フルスペル	解説
FU	Funcitonal Unit	演算器
REG		レジスタの略
MUX	マルチプレクサ	複数のデータ信号線からひとつを選択するセレクタのことをLSI設計では、MUXと呼ぶことが多い。MUXは本来は多重化装置で、複数の信号をひとつの信号に多重化して送信する装置。
C言語設計		LSIの設計では、CやC++という言語をLSIの動作記述として利用した設計手法のことを示す。Cで記述された動作記述は、動作合成ツールによって、RTLに自動変換され、さらに論理合成、レイアウトによってLSI設計データが生成される。したがって、LSIをソフトウェアと同じC言語で設計できることになる。
C-RTL等価性検証		動作記述であるC記述と、そこから合成されたRTL記述の機能的な等価性を証明すること。言い換えると、入出力の関数関係が同一であることを証明する。ステップ情報等は考慮に入れない。
プロパティ検証		回路の特性（ある二つの入力は同時に1にならない等）が、どのような入力系列に対しても成立することを証明すること。

参考文献

ー動作合成の教科書
【1】D.Gajski,A.Wu, N.Dutt,S.Lin,
"HIGH-LEVEL SYNTHESIS, Introduction to Chip and System Design," Kluwer Academic Publisher, 1992.
【2】G. De Micheli,
"Synthesis and Optimization of Digital Circuits," McGrawhill, 1994
【3】若林,
「自動設計の手法 機能合成,」最新VLSIの開発設計とCAD(監修 大附,後藤)ミマツデータシステム,pp.105-128,1994

ーC言語によるハードウェア設計手法とその実際
【4】K.Wakabayashi, T.Okamoto,
"C-based SoC Design Flow and EDA Tools: An ASIC and System Ventor Perspective," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, Vol.19. No.12, Dec 2000
ー動作合成Cyber関連
【5】K.Wakabayashi,
"Cyber: High Level Synthesis System from Software into ASIC," High-Level VLSI Synthesis, Kluwer Academic Publisher, 1991
【6】K.Wakabayashi,H.Tanaka,
"A Global Scheduling Method beyond Control Dependencies based on Condition Vectors," Design Automation Conference, 1992.
【7】若林他、
「伝送用LSIを動作合成で開発」、日経エレクトロニクス 1996.2.12(no.655)
【8】黒川他、
「動作合成を利用したC言語ベース設計の設計・検証生産性」、回路とシステム(軽井沢)ワークショップ、2002

B4章
機能・論理設計1
理解度テスト

B4章　理解度テスト （B.4.1）

(1)システムLSI設計の記述の抽象度について誤っている文の番号を答えよ。

① 設計の抽象度を上げると一般に記述量を減少することができる。

② 設計の抽象度を上げると一般にシミュレーション速度を上げることができる。

③ ゲートより演算器を使った記述の方が抽象度が高く、少ない行数で記述できる。

④ 抽象度が高い記述は曖昧なため多少の機能バグがあってもよい。

B4章　理解度テスト　(B.4.2)

(2)動作合成についての説明で誤っている文の番号を答えよ。

① 動作合成は機能設計を自動化するツールである。

② 動作合成は動作記述からRTLを合成する。

③ 動作合成は同一の動作記述から様々な性能や面積の回路(RTL)を合成することが狙いである。

④ 動作合成は機能合成と同じ意味であり、機能記述であるRTLを入力し、ゲート回路を合成する。

B4章　理解度テスト　(B.4.2)

(3)動作記述と構造記述(RTL)に関して正しい文の番号を答えよ。

① 動作記述は回路の入力と出力の関数関係を規定しているが、一般に構造は規定していない。

② 構造記述(RTL)は構造を記述してあるため、動作記述よりも簡潔で行数も少ない。

③ X=a+b+c　という記述は動作記述でありえるが、RTLではありえない。

④ 動作合成の入力は構造が明示されたRTLであるのが普通である。

B4章　理解度テスト （B.4.2）

(4)動作合成の内部処理で正しい文の番号を答えよ。

① 動作合成の主要な処理としてスケジューリング処理がある。これは演算子を実行する演算器を決定する処理である。

② バインディングとアロケーションはともに演算器の数を決定する同一の処理である。

③ スケジューリングとは演算子を実行する順番を決定する処理であり、自動並列化の処理ともいえる。

④ 言語レベルの最適化は処理時間がかかる割には得られる効果が少なく、省いてもさして問題にならない。

B4章　理解度テスト （B.4.2）

(5) スケジューリングについて正しい文の番号を答えよ。

①スケジューリングには大きく分けて二つの方法がある。演算器の数を制約としてステップ数の最小化を目的とする方法と、逆にステップ数を制約として演算器数の最小化を目的とする方法である。

②スケジューリング対象になるのは演算子だけである。

③スケジューリングはクロック周波数とは独立に行える。

④スケジューリングは現実的な計算時間で必ず最適解が求まることが証明されている。

B4章　理解度テスト　(B.4.2)

(6) 制御やデータの依存関係について誤っている文の番号を答えよ。

①データフローグラフ(DFG)は、変数や演算の間のデータの流れを表している。

②コントロールフローグラフ(CFG)は、動作記述の制御の流れを表している。

③制御の流れとデータの流れは通常同一であり、区別しなければならないのは特別な場合だけである。

④コントロールデータフローグラフ(CDFG)は制御とデータの依存関係の両方をひとつのグラフで表している。

B4章　理解度テスト　(B.4.2)

(7) バインディングに関して誤っている文の番号を答えよ。

① 動作記述の演算子は、演算器に割り付けられる。

② 動作記述の変数は、レジスタ等に割り付けられる。

③ データの転送(例a=b;)は、回路上は配線とセレクタなどで実現される。

④ バインディングは、スケジューリングで決定された回路をそのまま構築するため、バインディングアルゴリズムの違いは合成速度に影響があるだけで、合成回路の品質には影響がない。

B4章　理解度テスト　(B.4.2)

(8) 動作合成で合成されるアーキテクチャに関して誤っている文の番号を答えよ。

① 動作合成で生成されるアーキテクチャは、制御回路とデータパスからなっていることがある。

② 制御回路とデータパスは並列に動作するが、お互いに関連はなく独立に動作する。

③ 制御回路とデータパスは制御信号等で結合されている。

④ 制御回路は回路の実行順序(クロックステップ毎に次にどのような演算を実行するか)を決定する。

B4章　理解度テスト（B.4.2）

(9)演算器制約スケジューリングに関して正しい文の番号を答えよ。

① 利用可能な演算器を増加すると、実行に必要なステップ数を必ず削減することができる。

② リストスケジューリングはASAPスケジューリングより必ず良い解を得ることができることが証明されている。

③ 二つの演算子があり、最初の演算子の結果を次の演算子へ入力する場合、その二つの演算子の間にデータの依存関係があるという。

④ スケジューリングでは演算子の遅延はかならず1ステップ以内で実行可能でなければならない。（演算の遅延はクロック周期より必ず小さくなければならない。）

B4章　理解度テスト（B.4.5）

(10)LSIの設計生産性に関して誤っている文の番号を答えよ。

① LSIの設計生産性の向上のペースは、製造生産性の向上のペース（ムーアの法則）よりも低いといわれている。

② LSIの設計生産性を高めるためには、より抽象度の高い合成ツールを利用することが効果がある。

③ LSIは時代とともに大規模化してきているが、設計生産性も同率で向上しており、ゲート回路の設計だけで大きな問題は生じていない。

④ LSIの大規模化に対処するためには、より上位レベルで設計することで取り扱う記述量（部品点数）を削減することが効果的と考えられる。