第 3 章

PARTHENON を用いた論理合成

3.1 目的

本実験では，ハードウェア記述言語で記述された回路を論理合成系を用いて論理合成する方法について学ぶ．また，合成結果から設計した回路の性能やコストを見積もる方法について学ぶ．

3.2 PARTHENON システムの論理合成系

動作シミュレータ SECONDS によって SFL 記述された回路 の動作検証が完了した後に行なう作業は，論理合成である．論理合成は，構造あるいは動作記述された回路を論理に変換する処理である． PARTHENON システムの論理合成系は，次の 4 つのプログラムで構成される．

- SFLEXP — 論理合成プログラム
- OPT_MAP — マッピング＆論理回路最適化プログラム
- ONSET — 組合せ論理回路簡単化プログラム
- RINV — 極性最適化プログラム

また，回路図を作成するための NLD_PS というプログラムがある．ここでは，個々のプログラムの詳細には触れない．

SFL 記述ファイルは， SFLEXP で論理に変換した後， OPT_MAP， ONSET， RINV で論理圧縮 / 論理最適化処理が行なわれる．これらのプログラムを個々に呼び出して，論理合成 / 最適化処理を行なうことで，最終的にネットリストと呼ばれる論理回路を表現したファイルを生成する．

ユーザが個々のプログラムを起動して処理を進めることも可能であるが，ここでは，論理合成 / 最適化を自動的に行なうために用意されたコマンド (プログラム) を用いる．

`auto` コマンドは，上記のプログラムを順次呼び出して最適化されたネットリストを生成する．以下では，この`auto` コマンドを用いた論理合成の方法について述べる．

3.3 4 ビット加算器の論理合成

ここでは，第2章の 2.4.4で説明したサブモジュールを用いた 4 ビット加算器 `add4_v2` を例にその論理合成方法について述べる．

3.3.1 論理合成の手順

論理合成・論理最適化を行なうおおまかな手順は，次のとおりである．

1. サブモジュールの論理合成
2. トップモジュールの論理合成
3. トップモジュールの最適化処理

3.3.2 auto コマンド

まず，論理合成 / 最適化処理を容易に行なうために用意されている auto コマンドについて説明する．auto コマンドは，UNIX シェルのコマンドプロンプトにて用いる．auto コマンドは 4 つの引数をとり，その書式は次のとおりである．

書式 (1): auto 《モジュール名》《リザルトタグ》《ファンドリ》《セルライブラリ》
書式 (2): auto 《モジュール名》 clean

《モジュール名》では，論理合成あるいは最適化を行なうモジュール名を指定する． 《リザルトタグ》は，合成・最適化処理をどこまで進めるかを指定するタグであり，この部分では，hsl，nld1，nld2，nld3，nld4，ps などを指定する． 《ファンドリ》と《セルライブラリ》の部分では，合成に用いるセルライブラリのファンドリとライブラリを指定する．ここでは，PARTHENON に標準で入っているセルライブラリを用いるためそれぞれ，DEMO，demo と指定する．なお，このデフォルトで用意されているライブラリは，0.8μm CMOS プロセスの簡単なゲートのセットである．

書式 (2) は，指定したモジュールに関する作業用ディレクトリ，中間ファイルを削除する場合に用いる．

3.3.3 サブモジュールの論理合成

まず，論理合成の対象となっているモジュールで使用しているサブモジュールの論理合成を行なう．

```
% auto 《モジュール名》 nld1 DEMO demo
```

4 ビット加算器 add4_v2 では，全加算器 fulladder をサブモジュールとして使用しているので，《モジュール名》の箇所を fulladder として auto コマンドを実行する．

```
% auto fulladder nld1 DEMO demo
```

SFLEXP プログラムが起動され，端末にはプログラムからのメッセージが表示される．合成が成功すると，出力されるメッセージの最後のあたりに

```
** out one HSL module(fulladder) **

        There are 0 errors.
```

というメッセージが表示される．また，カレントディレクトリに “fulladder.hsl” というファイルと，“fulladder.1st” というディレクトリが作成される．ディレクトリ “fulladder.1st” には，“fulladder.nld” というファイルが生成されている．この生成された “fulladder.nld” というファイルが，全加算器の SFL 記述 “fulladder.sfl” の論理合成結果を納めているネットリストである．このファイルは，後に説明するトップモジュール (最上位のモジュール) の論理合成の際に使用する．

複数のサブモジュールがある場合，すべてのモジュールに対して nld1 までの処理を行ない，ネットリストファイル (.nld) を生成する．

3.3.4 トップモジュールの論理合成

トップモジュール add4_v2 に対しても，サブモジュールの場合と同様に，nld1 までの処理を行なう．

```
% auto add4_v2 nld1 DEMO demo
```

“add4_v2.1st” というディレクトリが作られ，そこに “add4_v2.nld” というファイルが生成される．

3.3.5 トップモジュールの最適化処理

トップモジュールの論理合成・最適化処理は，サブモジュールの論理合成結果を併せて行なう．そこで，先に生成した各サブモジュールの論理合成結果 (NLD ファイル) をトップモジュールの合成・最適化処理作業用ディレクトリにコピーする．

先に説明したように，モジュール M の論理合成結果ファイル (NLD ファイル) は，ディレクトリ M.1st に格納されている．例えば，全加算器モジュール add4_v2 の場合，ディレクトリは add4_v2.1st となる．サブモジュールすべての NLD ファイルを作業用ディレクトリにコピーする．ここでは，サブモジュール fulladder の NLD ファイルをトップモジュール add4_v2 の作業用ディレクトリにコピーする．

```
% cp fulladder.1st/*.nld add4_v2.1st
```

複数のサブモジュールがある場合，各サブモジュールの NLD ファイルすべてをトップモジュールの作業ディレクトリにコピーする．

コピーを終了したら，次のコマンドにて，トップモジュールの最適化処理を進める．

```
% auto add4_v2 nld4 DEMO demo
```

端末の画面には，プログラムから出力される最適化処理の際のメッセージが表示される．このメッセージに関しては，後ほど説明する．論理合成に成功すると，出力されるメッセージの最後のあたりに

```
** (there were 0 errors, 0 warning) **
```

というメッセージを発見するだろう．処理の過程で，カレントディレクトリに幾つかの中間ファイルとディレクトリが生成される．生成されるディレクトリは，“add4_v2.2nd”，“add4_v2.3rd”，“add4_v2.4th” である．各ディレクトリには，何段階か行なわれる最適化処理の中間結果が置かれる．最終的に得られたネットリストは，“add4_v2.4th” ディレクトリに置かれている “add4_v2.nld” というファイルである．

3.3.6 回路図の生成

回路図のファイル (Postscript 形式) を生成するには “auto コマンド” において《リザルトタグ》に ps を指定する．加算器モジュール add4_v2 の回路図を得るには次のようにタイプする．

```
% auto add4_v2 ps DEMO demo
```

auto コマンドより，NLD_PS というプログラムが起動され回路図生成処理が行なわれる．処理がエラーなく終了すると，カレントディレクトリに “add4_v2.ps” という名前のファイルが生成される．**図 3.1**に 4 ビット加算器の論理合成結果の回路図を示す．

論理合成・最適化処理の過程の中間ファイルから回路図を生成する際には，NLD_PS プログラム[1]を用いる．例えば，ディレクトリ “add4_v2.3rd” に置かれている NLD ファイルから回路図を生成する場合，次のようにタイプする．

```
% nld_ps -c "part of add4_v2" -r "v2.4.1" -a Y -o add4_v2_3rd.ps add4 add4_v2.3rd \
  $PARTHENON/cell_lib.dir/DEMO/demo/cell.dir \
  $PARTHENON/cell_lib.dir/DEMO/demo/start.dir
```

するとファイル “add4_v2_3rd.ps” に回路図が出力される．ここで行末の\ は 1 つのコマンドが複数行に渡る際に用いる文字である．\ の後にリターンキーを押しても次の行にてコマンドの続きを入力できる．なお，\ を省いて 1 行に続けてタイプしても構わない (この際画面上では 2 行以上になることもあるが，1 つのコマンドとして受け付けられる)．

出力された Postscript ファイルをプレビューするには，gv コマンドを用いるとよいだろう．回路図を印刷するには，lpr コマンドを用いる．

3.3.7 作業用ファイル，ディレクトリの削除

作業用ファイル，ディレクトリを削除する際には，次のコマンドを用いる．

```
% auto 《モジュール名》 clean
```

指定されたモジュールの中間ファイルが削除される．

3.3.8 論理合成処理・最適化処理におけるメッセージの保存

論理合成 / 最適化処理を行なう際に，プログラムから出力されるメッセージをファイルに保存するには，tee というフィルタコマンドを用いる．tee コマンドは，標準入力から入力される情報を指定したファイルに出力すると同時に，同じ情報を標準出力にも出力するというフィルタコマンドである．

[1] http://www.kecl.ntt.co.jp/car/parthe/hajimete/10shou.htm 参照

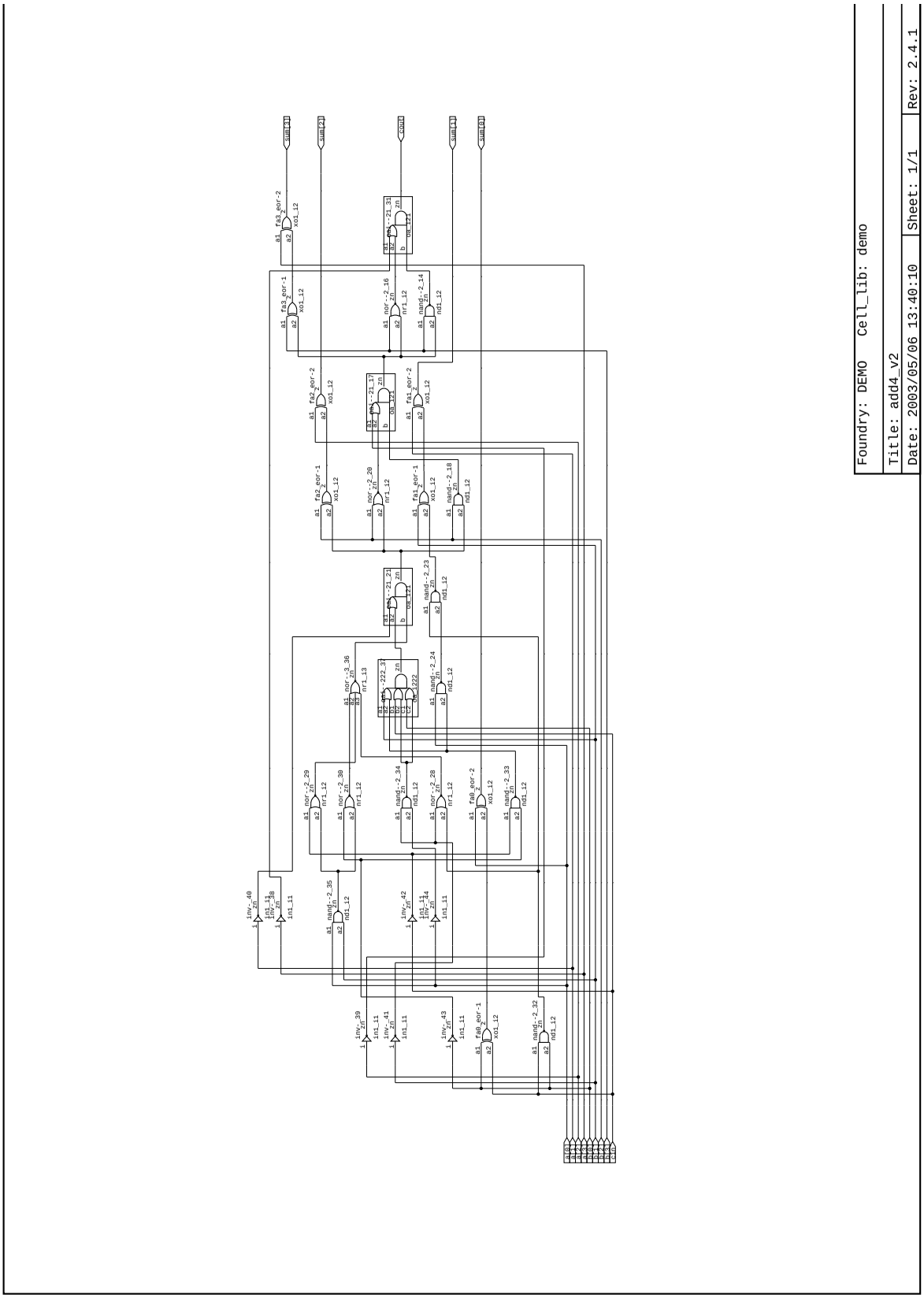

図 3.1: 4 ビット加算器の論理合成結果

`tee` [《ファイル$_1$》《ファイル$_2$》 ...]

`tee` コマンドで指定したファイル (複数でも可) に，標準出力に出力される内容と同じ内容が書き込まれる．

例えば，`add4_v2` モジュールの論理合成 / 最適化のメッセージをファイル “`add4_v2.rpt`” に保存する場合，次のようにタイプする．

```
% auto add4_v2 nld4 DEMO demo | tee add4_v2.rpt
```

すると，画面に出力されるメッセージと同一の内容がファイル “`add4_v2.rpt`” にも書き込まれる．論理合成・論理最適化処理プログラムの実行が終了した後に，ファイル “`add4_v2.rpt`” をブラウズすることで，プログラムからのメッセージをじっくりと眺めることができる．

3.4 論理合成結果の評価

3.3で，4 ビット加算器を例にとって論理合成・論理最適化の手順を示したが，ここでは，その際にプログラムから出力されるレポート (メッセージ) を眺めてみる．

3.4.1 起動されるプログラム

3.3.8で保存したファイル “`add4.rpt`” を眺めてみる．このレポートには様々なメッセージが出力されているが，ここでは，最後の辺りにある次のメッセージに注目する．

```
####### summary ############################

    --- maximum rise delay path histogram
```

(中略)

```
    delay path maximum  3.09866e+01
    delay path minimum  2.25364e+01
    delay path average  2.75128e+01
```

(中略)

```
    all statistics calculated
--- statistics summary --------------------
        position   = /
        type       = NLD
        class_name = add4_v2
        power      = 187.9
        area       = 21.38
        gates      = 84
```

これらの情報[2]から，合成された 4 ビット加算器 ‘`add4_v2`’ の 動作周波数，ゲート数 (実装面積)，消費電力を見積もることができる．なお，ここで使用しているテクノロジーは，$0.8\mu m$ CMOS プロセスである．したがって，現在の最新のテクノロジー[3]ではないことを念頭に置いて頂きたい．

動作周波数

遅延パス (delay path) は，通常，立上り遅延パス (rise delay path) を見る．最大遅延 (delay path maximum) は，`3.09866e+01` となっている．これは，3.09866×10^1 (nsec)，すなわち 30.9866 (nsec) を意味している．この値 ($T_{\max}$ とする) から動作周波数 $f_{\max}$ を計算すると，

$$f_{\max} = 1/T_{\max} = 1/(30.9866 \times 10^{-9}) = 32.272\text{MHz}$$

であり，約 32 MHz で動作可能であることが分かる．

[2]ここに示した結果は，PARTHENON version 2.4.1 で実行した際の出力である．バージョン等の違いにより，ここに示した結果と同一になるとは限らないので注意されたい．

[3]2002 年現在，商品化されているプロセッサ (例えば，Intel 社の Pentium4 プロセッサ) では，$0.13\mu m$ プロセスが利用されている．

ゲート数

ゲート数は gates の箇所から得られ，84 ゲートであることが分かる．また，実装面積は，21.38 ($1000\mu m^2$) である．これは 0.02138 mm^2 であり，この値の平方根より 0.147 mm 角のチップ面積程度であることが分かる．

消費電力

消費電力は power の箇所から得られ，単位は，μW/MHz である．すなわち，187.9μW/MHz である． CMOS の消費電力は周波数に比例するので，先程求めた最大動作周波数の 32MHz で動作させた場合の消費電力は，約 6 mW となることが分かる．

以上のように，論理合成時に出力されるレポートより，設計した回路の性能とコストを評価することができる． 4 ビット加算器論理合成結果をまとめたものを**表 3.1**に示す．

表 3.1: 4 ビット加算器論理合成結果のまとめ

最大遅延 (ns)	最大動作周波数 (MHz)	ゲート数	実装面積 ($1000\mu m^2$)	消費電力 (μW/MHz)	最大動作周波数で動作時の消費電力
30.9866	32.2	84	21.38	187.9	6 (mW)

3.5 桁上げ先見加算器

ここでは，桁上げ伝搬加算器 (ripple carry adder) との比較のために，桁上げ先見加算器 (carry-lookahead adder) を設計する．

3.5.1 4 ビット桁上げ先見加算器

桁上げ伝搬加算器では，桁上げ信号が下位ビットから順次上位ビットへと伝搬する．ビット数が多い場合，上位ビットの結果は多くの段数の論理ゲートを通過するため，遅延が大きくなる．桁上げ先見加算器は，桁上げ信号を少ない段数の論理によって求めることで，伝搬遅延の縮小を狙う回路である．最下位ビットからの桁上げ信号 c_1 は，次の論理式によって求まる．

$$c_1 = a_0b_0 + a_0c_{in} + b_0c_{in} \tag{3.1}$$

同様に，下位 2 ビット目からの桁上げ信号 c_2 は，次の論理式によって求まる．

$$\begin{aligned} c_2 &= a_1b_1 + a_1c_1 + b_1c_1 && (3.2)\\ &= a_1b_1 + a_1(a_0b_0 + a_0c_{in} + b_0c_{in}) + b_1(a_0b_0 + a_0c_{in} + b_0c_{in}) && (3.3)\\ &= a_1b_1 + a_1a_0b_0 + a_1a_0c_{in} + a_1b_0c_{in} + a_0b_1b_0 + a_0b_1c_{in} + b_1b_0c_{in} && (3.4) \end{aligned}$$

下から $i+1$ ビット目からの桁上げ信号 c_{i+1} は，次の論理式によって求まる．ただし，ここで 最下位ビットへの桁上げ入力 $c_0 = c_{in}$ である．

$$\begin{aligned} c_{i+1} &= a_ib_i + a_ic_i + b_ic_i && (3.5)\\ &= a_ib_i + (a_i + b_i)c_i && (3.6) \end{aligned}$$

ここで， $g_i = a_ib_i$， $p_i = a_i + b_i$ とすると， c_{i+1} は，次のようになる．

$$c_{i+1} = g_i + p_ic_i \tag{3.7}$$

この論理式より，下から 4 ビットまでの桁上げ信号は次のようになる．

$$\begin{aligned} c_1 &= g_0 + p_0c_0 && (3.8)\\ c_2 &= g_1 + p_1g_0 + p_1p_0c_0 && (3.9)\\ c_3 &= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0 && (3.10)\\ c_4 &= g_3 + p_3g_2 + p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0 && (3.11) \end{aligned}$$

4 ビット桁上げ先見加算器の SFL 記述を**図 3.2**に示す．

```
1  /* (cla4.sfl) */
2  %i "fulladder.h"
3
4  module cla4 {
5      input   a<4>, b<4>, cin;
6      output  sum<4>, cout;
7      instrin enable;
8
9      fulladder fa0, fa1, fa2, fa3;
10
11     sel_v s0, s1, s2, s3;
12     sel_v g0, g1, g2, g3;
13     sel_v p0, p1, p2, p3;
14     sel_v c1, c2, c3, c4;
15
16     instruct enable par {
17         g0 = a<0> & b<0>;
18         p0 = a<0> | b<0>;
19         c1 = g0 | (p0 & cin);
20         g1 = a<1> & b<1>;
21         p1 = a<1> | b<1>;
22         c2 = g1 | (p1 & g0) | (p1 & p0 & cin);
23         g2 = a<2> & b<2>;
24         p2 = a<2> | b<2>;
25         c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin);
26         g3 = a<3> & b<3>;
27         p3 = a<3> | b<3>;
28         c4 = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
29             (p3 & p2 & p1 & p0 & cin);
30         s0 = fa0.enable(a<0>, b<0>, cin).sum;
31         s1 = fa1.enable(a<1>, b<1>, c1).sum;
32         s2 = fa2.enable(a<2>, b<2>, c2).sum;
33         s3 = fa3.enable(a<3>, b<3>, c3).sum;
34
35         sum = s3 || s2 || s1 || s0;     /* ビットの連結 */
36         cout = c4;
37     }
38 }
39 /* End of file (cla4.sfl) */
```

図 3.2: 4 ビット桁上げ先見加算器の SFL 記述

3.5.2 8 ビット桁上げ先見加算器

4 ビット加算器の場合，桁上げ伝搬による遅延はそれほど大きな問題とならないが，ビット幅がより大きな加算器になると，桁上げ伝搬による遅延による影響が大きくなる．8 ビット，16 ビットの桁上げ先見加算器を作成しやすくするために 3.5.1で述べた 4 ビット桁上げ先見加算器に僅かな修正を行なう．4 ビット桁上げ先見加算器に，次の出力を設ける．

- 伝搬出力 p_{out}
- 生成出力 g_{out}

また，桁上げ信号 (c_{out}) は加算器外部で生成することを想定し，出力から除く．4 ビット桁上げ先見加算器の外部仕様を**図 3.3**に示す．

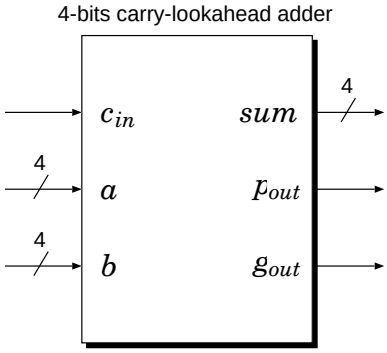

図 3.3: 4 ビット桁上げ先見加算器の外部仕様

p_{out}， g_{out} は，次の論理式で与えられる．

$$p_{out} = p_3p_2p_1p_0 \tag{3.12}$$

$$g_{out} = p_3p_2p_1g_0 + p_3p_2g_1 + p_3g_2 + g_3 \tag{3.13}$$

もし，4 ビット加算器の出力として桁上げ信号 (c_{out}) が必要な場合，次の論理式によって求めることができる．

$$c_{out} = g_{out} + p_{out}c_{in} \tag{3.14}$$

生成出力 g_{out}，伝搬出力 p_{out} をもつ 4 ビット桁上げ先見加算器の SFL 記述を**図 3.4**に示す．

上で作成した 4 ビット桁上げ先見加算器を 2 個用いて，8 ビット桁上げ先見加算器を設計する．ここでは，下位 4 ビットの計算を担当する 4 ビット桁上げ先見加算器を CLA4_0，上位 4 ビットの計算を担当する 4 ビット桁上げ先見加算器を CLA4_1 と呼ぶことにする．

加算結果は，CLA4_0，CLA4_1 から出力されるそれぞれ 4 ビットの加算出力を連結して 8 ビットとすればよい．CLA4_0 は，桁上げ出力が設けられていないが，式 3.14により求まる．これを CLA4_1 への桁上げ入力とすればよい．最上位ビットからの桁上げ出力は，次の式で求まる．

$$c_{out} = g^1_{out} + p^1_{out}g^0_{out} + p^1_{out}p^0_{out}c_{in} \tag{3.15}$$

ここで，x^i は，CLA4_i の出力信号 x を意味する．

8 ビット桁上げ先見加算器の SFL 記述を**図 3.5**に示す．

3.6 算術論理演算ユニット (ALU) の設計

これまで，加算器を中心に SFL 記述，PARTHENON システムの使用方法について述べてきたが，ここでは，機能を増やした論理回路の例として算術論理演算ユニット (ALU) の設計を行なう．

```
1  /* (cla4.sfl) */
2
3  module cla4 {
4      input   a<4>, b<4>, cin;
5      output  sum<4>, pout, gout;
6      instrin enable;
7
8      sel_v c<4>, p<4>, g<4>;
9
10     instruct enable par {
11         p =  (a<3> | b<3>) || (a<2> | b<2>) || (a<1> | b<1>) || (a<0> | b<0>);
12         g =  (a<3> & b<3>) || (a<2> & b<2>) || (a<1> & b<1>) || (a<0> & b<0>);
13         c =  (g<2> | (p<2> & g<1>) | (p<2> & p<1> & g<0>)
14                    | (p<2> & p<1> & p<0> & cin))
15           || (g<1> | (p<1> & g<0>) | (p<1> & p<0> & cin))
16           || (g<0> | (p<0> & cin))
17           || cin;
18         sum = a @ b @ c;
19         pout = p<3> & p<2> & p<1> & p<0>;
20         gout =  (p<3> & p<2> & p<1> & g<0>)
21               | (p<3> & p<2> & g<1>)
22               | (p<3> & g<2>)
23               | g<3>;
24     }
25 }
26 /* End of file (cla4.sfl) */
```

図 3.4: 生成・伝搬出力をもつ 4 ビット桁上げ先見加算器の SFL 記述

```
1  /* (cla8.sfl) */
2
3  %i "cla4.h"
4
5  module cla8 {
6      input  a<8>, b<8>, cin;
7      output sum<8>, cout;
8      instrin enable;     /* 制御端子 */
9
10     sel_v c<2>;
11     cla4 CLA4_0, CLA4_1;
12
13     instruct enable par {
14         c =  (CLA4_0.gout | (CLA4_0.pout & cin)) || cin;
15         sum =  CLA4_1.enable(a<7:4>, b<7:4>, c<1>).sum
16             || CLA4_0.enable(a<3:0>, b<3:0>, c<0>).sum  ;
17         cout =  CLA4_1.gout
18              | (CLA4_1.pout & CLA4_0.gout)
19              | (CLA4_1.pout & CLA4_0.pout & cin);
20     }
21 }
22 /* End of file (cla8.sfl) */
```

図 3.5: 8 ビット桁上げ先見加算器の SFL 記述

3.6.1 ALU

ALU は，算術演算と論理演算の機能を有する論理回路である．ここで設計する ALU の算術演算としては加算，減算を設け，論理演算としては，論理否定 (NOT)，論理和 (OR)，論理積 (AND)，排他的論理和 (XOR) の演算を設ける．データ入力としては，a，b の 2 個の入力をとり，演算を選択する制御信号 ($func$) を用いて，先に上げた演算機能を選択する．

まず，データのビット幅は，4 ビットにて設計を行なう．

先に上げた 6 種類の演算機能と，入力 a，b のデータのいずれかを演算なしにそのまま出力する機能を追加した合計 8 種類の演算機能を実装する．したがって，機能選択信号 ($func$) は 3 ビット幅となる．4 ビット ALU の外部仕様を**図 3.6**に示す．

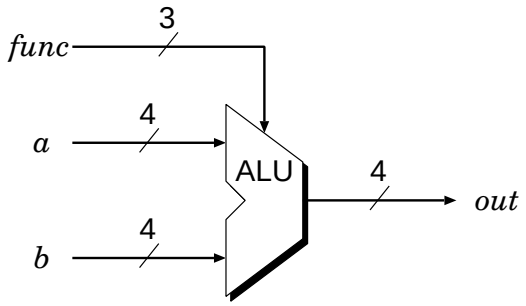

図 3.6: 4 ビット ALU の外部仕様

機能選択信号のコードを決定する．ここでは，8 種類の機能を**表 3.2**に示すように割り当てる． **表 3.2**の「コード」部分は，2 進数で表記している．また，「機能ラベル」は，SFL のソースを記述する際に具体的なコードの値 (例えば，`0b000` といた数値) を直接記述せずに，シンボル (ラベル) として記述するのに用いる．これは，C 言語といったプログラムにおいて，ソースプログラム中に定数値を直接記述せず，マクロ機能を用いてシンボルとして記述することと同じ目的である．

表 3.2: ALU の機能コード

機能ラベル	コード	機能	
THAFUNC	000	$out \leftarrow a$	a をそのまま出力
THBFUNC	001	$out \leftarrow b$	b をそのまま出力
ANDFUNC	010	$out \leftarrow a \ \& \ b$	a と b の論理積を出力
ORFUNC	011	$out \leftarrow a \mid b$	a と b の論理和を出力
XORFUNC	100	$out \leftarrow a$ xor b	a と b の排他的論理和を出力
NOTFUNC	101	$out \leftarrow \overline{a}$	a の論理否定を出力
ADDFUNC	110	$out \leftarrow a + b$	a と b の加算結果を出力
SUBFUNC	111	$out \leftarrow a - b$	a から b を減算した結果を出力

3.6.2 4 ビット ALU の SFL 記述

3.6.1で述べた 4 ビット ALU の SFL 記述を**図 3.7**に示す． **図 3.7**の 3 行目にて読み込んでいる ALU の機能定義ファイル (“`alu4_func.def`”) を**図 3.8**に示す．

一般的なプログラミングと同様に，記述の読み易さや保守の点で記述の中に数値を直接書かない方が良い．SFL では，C 言語における `#define` と同様な機能を有するものとして`%d` という記述が用意されている．例えば，

`%d FOO` *value*

と書かれている場合，この記述以下の記述ファイル中のシンボル`FOO` は，すべて *value* で置き換えられることになる．**図 3.7**では，記述の先頭でファイル “`alu4_func.def`” を読み込んでおり，ファイル “`alu4_func.def`” では，

```
1  /* (alu4.sfl) */
2  %i "add4.h"
3  %i "alu4_func.def"
4
5  module alu4 {
6      input  a<4>, b<4>;  /* input date */
7      input  func<3>;     /* function */
8      output out<4>;      /* output data */
9      instrin enable;
10
11     add4 adder;
12
13     instruct enable alt {
14         func == THAFUNC: out = a;
15         func == THBFUNC: out = b;
16         func == ANDFUNC: out = a & b;
17         func == ORFUNC:  out = a | b;
18         func == XORFUNC: out = a @ b;
19         func == NOTFUNC: out = ^a;
20         func == ADDFUNC: out = adder.enable(a, b, 0b0).sum;
21         func == SUBFUNC: out = adder.enable(a, ^b, 0b1).sum;
22     }
23 }
24 /* End of file (alu4.sfl) */
```

図 3.7: 4 ビット ALU の SFL 記述

```
1  /* alu function */
2  %d THAFUNC       0b000
3  %d THBFUNC       0b001
4  %d ANDFUNC       0b010
5  %d ORFUNC        0b011
6  %d XORFUNC       0b100
7  %d NOTFUNC       0b101
8  %d ADDFUNC       0b110
9  %d SUBFUNC       0b111
```

図 3.8: ALU の機能コードの定義ファイル

`THAFUNC`, `THBFUNC`, ···, `SUBFUNC`

といった 8 個のシンボルを定義している．したがって，**図 3.7**における該当するシンボルは**図 3.8**の中で各シンボルに対して定義されている値にそれぞれ置き換えられることになる．なお，SFL 記述においては，数値の前に `0b` を付けるとその数値は 2 進数であることを意味する．

図 3.7の 6 行目から 9 行目は，モジュール `alu4` の入出力および制御端子の記述である． 11 行目は，内部で用いている 4 ビット加算器サブモジュールの宣言である．

13 行目から 22 行目が ALU 機能の記述である． 13 行目で， alt 文が用いられている． alt 文は，条件が成立する動作のみを実行させる構文である．ここでは， ALU の機能選択をする端子 `func` の値によって異なる動作をするように記述している．例えば， 16 行目は，`func` の値が `ANDFUNC`(すなわち，`0b010`) の場合は，`a` と`b` の論理積 (AND) をとった値を`out` の値とすることを記述している．

3.7 シフタの設計

ここでは， 8 ビットシフタの設計を行なう．

3.7.1 シフト / ローテートの機能

ビットのシフトおよびローテート (rotate，回転) としては，次の 5 種類がある．

- 左シフト (shift left)
- 論理右シフト (shift right logical)
- 算術右シフト (shift right arithmetic)
- 左ローテート (rotate left)
- 右ローテート (rotate right)

以下，それぞれの機能について説明する．

左シフト (shift left)

左シフトは，各ビットの値を指定されたシフト量だけ左にシフトする．下位ビットには， 0 を埋める．上位 (左側) で溢れたビットは，捨てる．

下記は 8 ビットの値を 3 ビット左シフトする場合を示している．空いた下位 3 ビットには 0 が埋められ，溢れた上位 3 ビット (b_7〜b_5) は捨てられる．

$$b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow b_4b_3b_2b_1b_0000$$

論理右シフト (shift right logical)

論理右シフトは，各ビットの値を指定されたシフト量だけ右にシフトする．空いた上位ビットには， 0 を埋める．下位 (右側) で溢れたビットは，捨てる．

下記は 8 ビットの値を 3 ビット論理右シフトする場合を示している．空いた上位 3 ビットには 0 が埋められ，下位 3 ビット (b_2〜b_0) は捨てられる．

$$b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow 000b_7b_6b_5b_4b_3$$

算術右シフト (shift right arithmetic)

算術右シフトは，各ビットの値を指定されたシフト量だけ右にシフトする．空いた上位ビットには，シフトする前の最上位ビットの値を埋める．下位 (右側) で溢れたビットは，捨てる．

下記は 8 ビットの値を 3 ビット算術右シフトする場合を示している．上位のビットは，シフト以前の最上位ビット b_7 の値で埋められ，下位 3 ビット (b_2〜b_0) は捨てられる．

$$b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow b_7b_7b_7b_7b_6b_5b_4b_3$$

左ローテート (rotate left)

左ローテートは，各ビットの値を指定されたシフト量だけ左にシフトする．空いた下位ビットには，上位で溢れた値を埋める．

下記は 8 ビットの値を 3 ビット左回転する場合を示している．溢れた上位 3 ビットの値は，空いた下位 3 ビットに埋められる．

$$b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow b_4b_3b_2b_1b_0b_7b_6b_5$$

右ローテート (rotate right)

右ローテートは，各ビットの値を指定されたシフト量だけ右にシフトする．空いた上位ビットには，下位で溢れた値を埋める．

下記は 8 ビットの値を 3 ビット右回転する場合を示している．溢れた下位 3 ビットの値は，空いた上位 3 ビットに埋められる．

$$b_7b_6b_5b_4b_3b_2b_1b_0 \rightarrow b_2b_1b_0b_7b_6b_5b_4b_3$$

図 3.9にシフトとローテートの演算を示す．ビット移動に伴い溢れたビット，空いたビットの扱いの違いがわかるであろう．

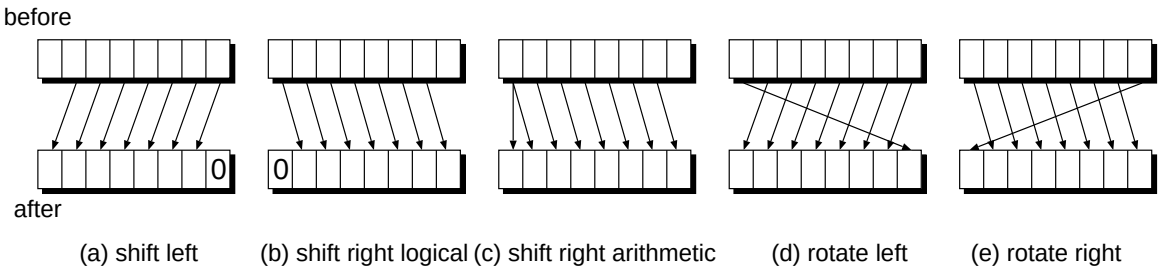

図 3.9: シフト，ローテート

3.7.2 8 ビットシフタの仕様

ここでは，8 ビットシフタの仕様について考える．データとしての入力および出力はそれぞれ 8 ビットの値とする．入力としてシフト量を与える必要がある．シフト量は最大で 0 ～ 7 の値で与えれば良いので，3 ビットの信号を必要とする．シフトの機能を選択する入力信号としては，

- 左シフト (ローテート) か右シフト (ローテート) を選択する信号，
- シフトかローテートを選択する信号，
- シフトの場合，算術シフトか論理シフトかを選択する信号

の 3 ビットがあれば，5 種類のシフト / ローテートの機能を選択することができる．**図 3.10**に 8 ビットシフタの外部仕様を示す．機能選択信号 $func$ の符号は**表 3.3**のように割り当てる．

機能コードの最下位ビット ($func_0$) は，操作が左 / 右のどちらであるのかを表すビットである．左なら $func_0 = 0$，右なら $func_0 = 1$ である．

$func_1$ は，操作がシフト / ローテートのどちらであるのかを表すビットである．シフトなら $func_1 = 0$，ローテートなら $func_1 = 1$ である．

機能コードの最上位ビット ($func_2$) は，操作が論理シフト / 算術シフトのどちらであるのかを表すビットである．論理シフトなら $func_0 = 0$，算術シフトなら $func_0 = 1$ である．

シフタはレジスタを接続して構成することも可能であるが，ここでは，組合せ回路として構成する．8 ビットシフタの場合，**図 3.11**に示すように 3 個のシフタを多段接続することで構成できる．1 段目のシフタはシフト量 1 ビットのシフタ，2 段目のシフタはシフト量 2 ビットのシフタ，3 段目のシフタはシフト量 4 ビットのシフタである．シフト量の異なるシフタを多段に接続することで任意のビット数のシフトを実現することができる．このようなシフタのことを**バレルシフタ (barrel shifter)** と呼ぶ．

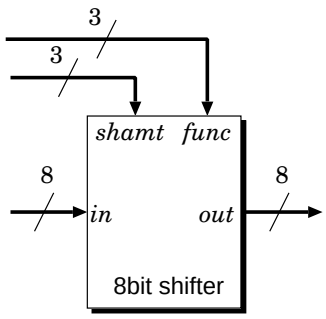

図 3.10: 8 ビットシフタ

表 3.3: 8 ビットシフタの機能コード

機能ラベル	コード	機能
SLFUNC	000	左シフト
SRLFUNC	001	論理右シフト
SRAFUNC	101	算術右シフト
ROLFUNC	010	左ローテート
RORFUNC	011	右ローテート

例えば，シフト量 5 のシフトを行なう場合， 1 段目と 3 段目のシフタでシフト動作を行ない， 2 段目のシフタはシフトせずに入力された信号をそのまま次の段に伝えれば良い．どの段のシフタを動作させるかは，シフト量の信号 *shamt* の各ビットにより決定できる．シフト量 5 (0b101) の場合，ビットの立っている 1 段目と 3 段目をシフト動作させる．シフト量 3 (0b011) の場合， 1 段目と 2 段目をシフト動作させる．

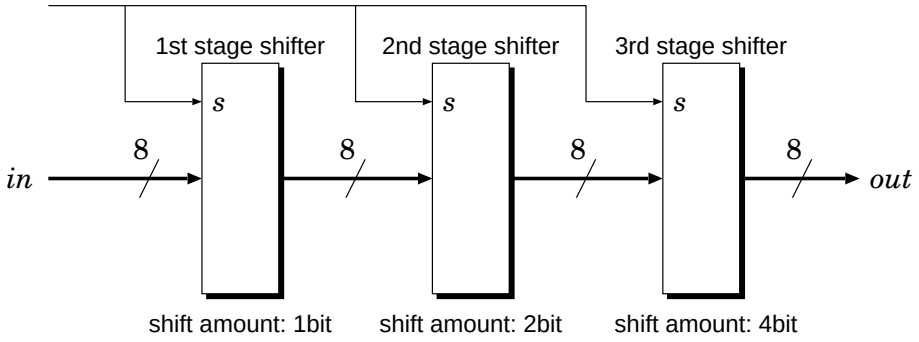

図 3.11: 8 ビットシフタの内部構造

図 3.11において，信号 s はシフトを行なった際に空いたビット部に埋める値を与える信号である．論理シフトの場合 $s = 0$ とし，算術シフトの場合 s に入力 *in* の最上位ビット (MSB) を与える．

3.7.3 8 ビットシフタの SFL 記述

図 3.12〜**図 3.14**に 8 ビットシフタの SFL 記述 (“`shift8.sfl`”, “`shift8.h`”, “`shift8_func.def`”) を示す．ただし，**図 3.12**に示した SFL 記述には，ローテートの機能は実現されていない． **図 3.12**の 10 行目に`instrself` というキーワードが使われている．これはモジュール内部の制御を行なう**制御内部端子**と呼ばれるものである．`instrin` がモジュールの外部から制御されるのに対して，`instrself` はそのモジュール自身が制御を行なう．制御内部端子 *a* をアクティベートするには，次のように記述する．

```
a();
```

```
/* (shift8.sfl)
   NOTE: rotate functions are not implemented */
%i "shift8_func.def"

module shift8 {
    input in<8>, shamt<3>, func<3>;
    output out<8>;
    instrin enable;

    instrself   n1, n2, n4;
    sel_v       o1<8>, o2<8>, right, arith, shift, s;

    instruct enable par {
        right = func<0>;
        shift = ^func<1>;
        arith = func<2>;
        s = in<7> & arith;

        any {
            shift: alt { /* shift */
                right: par {    /* shift right */
                    alt { shamt<0>: o1  = s || in<7:1>;      else: n1(); }
                    alt { shamt<1>: o2  = s || s || o1<7:2>; else: n2(); }
                    alt { shamt<2>: out = s || s || s || s || o2<7:4>;
                          else: n4(); }
                }
                else: par {     /* shift left */
                    alt { shamt<0>: o1  = in<6:0> || 0b0;    else: n1(); }
                    alt { shamt<1>: o2  = o1<5:0> || 0b00;   else: n2(); }
                    alt { shamt<2>: out = o2<3:0> || 0b0000; else: n4(); }
                }
            }
            /* rotate functions are not implemented */
        }
    }

    /* no shift */
    instruct n1 o1 = in;
    instruct n2 o2 = o1;
    instruct n4 out = o2;
}
/* End of file (shift8.sfl) */
```

図 3.12: 8 ビットシフタの SFL 記述 (ローテート機能は未実装)

```
/* (shift8_func.def) */
%d SLFUNC  0b000 /* shift left */
%d SRLFUNC 0b001 /* shift right logical */
%d RLFUNC  0b010 /* rotate left */
%d RRFUNC  0b011 /* rotate right */
%d SRAFUNC 0b101 /* shift right arithmetic */
/* End of file (shift8_func.def) */
```

図 3.13: 8 ビットシフタの機能コードの定義ファイル

```
1  /* (shift8.h) */
2
3  declare shift8 {
4      input in<8>, shamt<3>, func<3>;
5      output out<8>;
6      instrin enable;
7
8      instr_arg enable(in, shamt, func);
9  }
10 /* End of file (shift8.h) */
```

図 3.14: 8 ビットシフタのヘッダファイル

図 3.12の 22 行目から 25 行目，および 28 行目から 30 行目において，条件に応じて制御内部端子**n1**, **n2**, **n4** をアクティベートするように記述されている．

図 3.12において制御内部端子 **n1**, **n2**, **n4** は，シフタ内部に 3 個置かれた特定ビットだけシフトする内部シフタにおいて，シフトを行なわずに入力された信号をそのまま次段の内部シフタに伝えるために用いらている．これらの各制御内部端子がアクティベートされた際の動作は，38 行目から 40 行目に記述されている．

制御内部端子を用いることなく，動作を記述することも可能であるが，同じ動作を複数の箇所に記述していると，可読性や保守性の低下を招くことがある．同じ動作を複数の箇所に記述する必要がある場合，制御内部端子を用いることを勧める．

3.7.4 動作シミュレーション

8 ビットシフタの機能を確認するための表を**表 3.4**に示す．(2) ～ (6) の出力信号の値を考えてみよ．

表 3.4: シフタの機能確認

	入力信号			出力信号
	in	*func*	*shamt*	*out*
(1)	00110101	000	011	10101000
(2)	00110101	001	011	
(3)	00110101	010	011	
(4)	00110101	011	011	
(5)	00110101	101	011	
(6)	10110101	101	011	

図 3.15に 8 ビットシフタの動作を確認する SECONDS スクリプトの例を示す．このスクリプトでは，**表 3.4**に示した値で動作シミュレーションを行なう．8 ビットシフタの動作シミュレーションを行ない**表 3.4**の出力信号と一致するか確認せよ．

3.8 実験課題

【実験課題 3.1】
本章で説明した各回路の SFL 記述を作成して動作シミュレーションを行ない動作を確認せよ．また論理合成を行ない，論理合成により得られる各諸量 (最大遅延，最大動作周波数，ゲート数，最大動作周波数における消費電力) を求め**表 3.5**に示す表の形式にてまとめよ．また，2章で設計した各回路も論理合成して結果をまとめてみよ．

```
1  # (shift8_test01.scr)
2  sflread shift8.sfl
3  install shift8 /
4  rpt_add A "t=%t: in=%b func=%b shamt=%b | out=%b\n" in func shamt out
5  set /enable 1
6  hold /enable
7  set /in 00110101
8  hold /in
9  set /shamt 011
10 hold /shamt
11 set /func 000
12 hold /func
13 forward +1
14 set /func 001
15 forward +1
16 set /func 010
17 forward +1
18 set /func 011
19 forward +1
20 set /func 101
21 forward +1
22 set /in 10110101
23 forward +1
```

図 3.15: 8 ビットシフタの動作を確認する SECONDS スクリプト

表 3.5: 論理合成で得られた諸量のまとめ

モジュール	最大遅延 (ns)	最大動作周波数 (MHz)	ゲート数	実装面積 ($1000\mu m^2$)	消費電力 ($\mu W/MHz$)	最大動作周波数で動作時の消費電力
4 ビット加算器						
4 ビット桁上げ先見加算器						
8 ビット桁上げ先見加算器						
16 ビット桁上げ先見加算器						
4 ビット ALU						
8 ビットシフタ						
4 ビットカウンタ						
10 進カウンタ						
4 ビットアップダウンカウンタ						

3.9 設計課題

【設計課題 3.1】
16 ビットシフタ (`shift16`) を設計し，動作シミュレーションにより動作を確認せよ．動作を確認できたら，論理合成を行なって論理合成によって得られる諸量をまとめよ．

【設計課題 3.2】
16 ビット ALU(`alu16`) を設計し，動作シミュレーションにより動作を確認せよ．動作を確認できたら，論理合成を行なって論理合成によって得られる諸量をまとめよ．

3.10 検討課題

【検討課題 3.1】
PARTHENON の論理合成系を構成する各プログラムでは，それぞれどういった論理最適化処理を行なっているかを調査して簡潔にまとめよ．

3.11 まとめ

3.11.1 チェック項目

論理合成

- PARTHENON の論理合成系の概要を理解した.
- SFL 記述の論理合成を行なえる.
- 論理合成結果から動作周波数，回路規模，消費電力等を求めることができる.

論理回路

- 桁上げ先見加算器の構造，原理を理解した.
- バレルシフタの構造，原理を理解した.
- ALU の構造，原理を理解した.

3.11.2 本章で作成 / 設計したモジュール

1. 4 ビット桁上げ先見加算器 (`cla4`)
2. 8 ビット桁上げ先見加算器 (`cla8`)
3. 4 ビット ALU (`alu4`)
4. 8 ビットシフタ (`shift8`)
5. 16 ビット ALU (`alu16`)