第 2 章

ハードウェア記述言語 SFL と動作シミュレータ SECONDS

2.1 目的

本実験では，基本的な論理回路をハードウェア記述言語で記述し，動作シミュレーションを行なうことで，ハードウェア記述言語，および，ハードウェア記述言語を用いたハードウェア設計手法について学ぶ．

2.2 ハードウェア記述言語

ディジタル回路設計においては，これまで回路図を描き設計する方法が用いられてきたが，最近では，計算機のプログラミングと同様に，設計する回路を**ハードウェア記述言語** (hardware description language, 以降 HDL と略す) と呼ばれる言語で記述し，その記述からハードウェア (回路) に自動的に変換する方法が一般的になりつつある．ハードウェア記述言語を用いて設計を行なう利点としては，次のことが挙げられる．

- 高いレベルで記述するため，バグを発見しやすく，他人が見ても分かりやすい．
- 論理の簡単化，最適化等の低いレベルの設計は論理合成系にて自動的に行なわれるため，設計に要する時間が短縮でき，また，バグも減る．

回路図レベルで設計するか，あるいは， HDL を用いるかは，計算機のプログラムをアセンブリ言語で書くか，高級言語で書くかという関係に類似している．現在，アセンブリ言語でプログラムを書く機会は極端に減っており，将来は， HDL を用いた記述・設計が主流になることは間違いないと言われている．

現在，一般的に用いられている HDL には次のようなものがある．

- VHDL
- Verilog-HDL
- SFL

これらは，それぞれに特徴があるが，ここでは言及しない．今回，実験で用いるのは， SFL という HDL である．SFL は， NTT で開発された HDL で， SFL で記述された回路は，後述する PARTHENON システムを用いることで 動作シミュレーション，および，論理合成 (回路情報への変換)を行なうことができる．

2.2.1 SFL の概要

SFL(Structured Function description Language) では，他の HDL と同様，回路を構成する基本単位を**モジュール**と呼び，モジュール同士の階層構造で回路を表現する．モジュールには，インタフェースとして入力，出力，制御入力，制御出力の端子があり，内部には仮端子，レジスタ，他のモジュール等を持つことが出来る．各モジュールの機能は， C 言語風に記述する． SFL では，基本的に**同期回路のみの設計が可能**であり，暗黙のうちにシステムクロックが存在し，各制御文は各システムクロック毎に実行される．

SFL に関しては，簡単なマニュアルを付録 Bにつけている．更に詳しい情報は， NTT の PARTHENON ホームページ内の SFL マニュアルのページ [2] を参照して頂きたい．

2.3 PARTHENON システムの概要

PARTHENON(Parallel Architecture Refiner THEorized by Ntt Original coNcept) は，NTT で開発されたハードウェア設計支援システムである．PARTHENON では，ハードウェア記述言語として一般的な Verilog-HDL や VHDL ではなく，独自に開発されたハードウェア記述言語 SFL (Structured Function description Language) を用いる．SFL/PARTHENON の特徴を以下に示す．

- 記述対象を同期システムに限定
- レジスタ・トランスファ・レベル (RTL) の手続き記述のみによってハードウェアの動作を完全に記述

ハードウェア記述言語 SFL が，シミュレータ，合成系の研究と同時に開発されたことにより，ハードウェアの記述からシミュレーション，合成，最適化，テストなどの各機能が効率的にかつ首尾一貫して実現できる枠組を見越した上で SFL の言語仕様があり，これらすべての処理系を統合したシステムが PARTHENON なのである．

PARTHENON は，以下のプログラムにより構成される．

- SECONDS — SFL 動作シミュレータ
- 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
- NLD_PS — 回路図作成プログラム

SECONDS は，SFL で記述されたハードウェアの動作検証を行なうためのシミュレータである．残りのプログラムは，論理合成系にあたる．論理合成系については，第 3 章にて取り扱う．

設計者は，設計対象のハードウェアを SFL を用いて記述し，SFL 動作シミュレータ SECONDS を用いてアーキテクチャ・レベルでより望ましい設計へと改良していくことができる．SFL 記述の段階で，意図する動作の確認さえできれば，設計工程の大部分を終えたことになる．残りの処理 (論理合成・最適化処理) は，ほとんどプログラムまかせで行なわれ，最終ネットリスト，および，回路図を得ることができる．

PARTHENON に関しては，簡単なマニュアルを付録 Aにつけている．更に詳しい情報は，PARTHENON のホームページ [1] を参照して頂きたい．

2.4 SFL による回路記述

2.4.1 基本論理ゲートの SFL 記述

SFL の構文規則は C 言語風である．C 言語を知っていれば容易に理解することができる．SFL は，他の HDL 同様，回路の基本単位は **モジュール** (module) と呼ばれ，モジュール同士を接続することで回路を表現する．

モジュールには，入力 (`input`)，出力 (`output`)，制御入力 (`instrin`)，制御出力 (`instrout`) の 4 種類の入出力端子がある．モジュール内部で用いる端子には，　内部データ端子 (`sel`, `bus`, `sel_v`, `bus_v`, ...)，レジスタ (`reg`)，メモリ (`mem`)，他で定義されたモジュール等がある．

ここでは，まず，簡単な例として，2 入力の基本論理ゲートの SFL 記述を示す．下記の例では，`and`，`or`，`nand`，`xor` の 4 個のモジュールを定義している．各モジュールは，2 つの入力 (`in1` と `in2`) と 1 つの出力 (`out`) をとる．例に示すように，SFL では module 文

```
module { ··· }
```

を用いてモジュールを記述する．

AND ゲート

```
/* (and.sfl) */
module and {
    input in1, in2;
    output out;

    out = in1 & in2;
}
```

OR ゲート

```
/* (or.sfl) */
module or {
    input in1, in2;
    output out;

    out = in1 | in2;
}
```

NAND ゲート

```
/* (nand.sfl) */
module nand {
    input in1, in2;
    output out;

    out = ~(in1 & in2);
}
```

XOR ゲート

```
/* (xor.sfl) */
module xor {
    input in1, in2;
    output out;

    out = in1 @ in2;
}
```

ハードウェア記述言語における最も基本的な記述方法は，論理式を直接記述する方法である．上記の基本ゲートの記述例では，‘`out = ...`’ の部分で論理式を記述していることが容易に分かるであろう．また，演算子 `&` は論理積 (AND) を，`|` は論理和 (OR) を，`~` は論理否定 (NOT) を，`@` は排他的論理和 (XOR) を意味していることも分かるであろう．なお，SFL の演算子は，付録 B.13を参照されたい．

実は，上記に示した SFL 記述は SFL らしい書き方ではない．SFL の特徴の 1 つである **制御入力端子**を用いた 2 入力 AND ゲートの記述を**図 2.1**に示す．

```
1  /* (and2.sfl) */
2
3  module and2 {
4      input in1, in2;
5      output out;
6      instrin enable;
7
8      instruct enable out = in1 & in2;
9  }
10 /* End of file (and2.sfl) */
```

図 2.1: 2 入力 and ゲートの SFL 記述

6 行目の ‘`instrin enable;`’ が制御入力端子を宣言する文である．ここでは，‘`enable`’ という制御入力端子を宣言している．SFL では，モジュールには制御入力を用意する必要がある．制御入力は，モジュール動作の制御を行なうのに用いられる．

8 行目の，`instruct` で制御入力 ‘`enable`’ を利用している．`instruct` の構文は，次のとおりである．

`instruct` 《制御端子》 《動作》

この文は，《制御端子》で与えられた端子の値がアクティブ (= 1) な場合に限り，《動作》に記述された動作を行なうという意味である．**図 2.1**の 8 行目の記述は，‘`enable`’ がアクティブな場合に限り，出力 ‘`out`’ の値を求めることを意味する．

2.4.2 全加算器回路 (full-adder)

次に出力信号が複数ある組合せ回路の記述方法について，全加算器 (full-adder) を例に説明する．

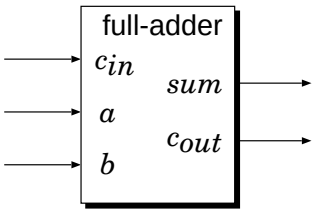

図 2.2: 全加算器の外部仕様

表 2.1: 全加算器の真理値表

入力			出力	
a	b	c_{in}	sum	c_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

全加算器は，2 つの信号 a，b と 下の桁からの桁上げ c_{in} の 3 つの入力から，和の出力 sum と上の桁への桁上げ c_{out} の 2 つの出力を求める回路である．全加算器の入出力信号 (外部仕様) を**図 2.2**に示す．また，全加算器の真理値表を**表 2.1**に示す．

SFL では真理値表をそのまま記述することも可能であるが，ここでは，論理式を記述する方法について説明する．全加算器の和出力 sum を求める論理式は，次のようになる．

$$sum = \overline{a}\overline{b}c_{in} + \overline{a}b\overline{c_{in}} + a\overline{b}\overline{c_{in}} + abc_{in} \quad (2.1)$$

$$= a \oplus b \oplus c_{in} \quad (2.2)$$

ここで，‘$\overline{x}$’ は x の否定，‘xy’ は x と y の論理積，‘$x + y$’ は x と y の論理和，‘$x \oplus y$’ は x と y の排他的論理和を意味する．

一方，出力 c_{out} を求める論理式は，次のようになる．

$$c_{out} = \overline{a}bc_{in} + a\overline{b}c_{in} + ab\overline{c_{in}} + abc_{in} \quad (2.3)$$

$$= ab + ac_{in} + bc_{in} \quad (2.4)$$

全加算器回路の SFL 記述

上で求めた論理式を用いることで，容易に SFL 記述を得られる．全加算器回路の SFL 記述を**図 2.3**に示す． 9 行目で桁上げ出力 (`cout`)， 10 行目で和出力 (`sum`) を計算していることが分かるだろう．

ハードウェアは，逐次実行されるソフトウェアと異なり並列に動作する部分があり，並列動作の記述が要求される． SFL にも，並列動作を記述するための構文が用意されている．**図 2.3**で， 8 行目の `par` は，それに続く動作 (ここでは，ブロック内の複数の動作) を逐次的にではなく，**同時に**処理するという意味である．**図 2.3**に示した例では，信号`cout` と`sum` を同時に計算することを意味する．

2.4.3 4 ビット加算器

4 ビット加算器は， 4 ビット幅の 2 組 (a と b) の入力と 1 ビットの桁上げ入力 (c_{in}) から， 4 ビット幅の加算結果 (sum) と 1 ビットの桁上げ (c_{out}) を出力する回路である． 4 ビット加算器の外部仕様を**図 2.4**に示す．

```
1  /* (fulladder.sfl) */
2
3  module fulladder {
4      input a, b, cin;
5      output sum, cout;
6      instrin enable;
7
8      instruct enable par {
9          cout = (a & b) | (a & cin) | (b & cin);
10         sum =  a @ b @ cin;
11     }
12 }
13 /* End of file (fulladder.sfl) */
```

図 2.3: 全加算器の SFL 記述

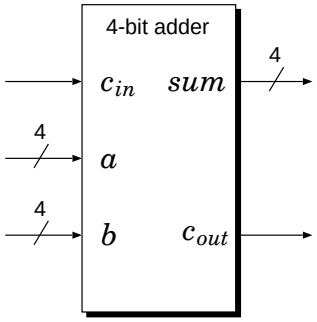

図 2.4: 4 ビット加算器の外部仕様

4 ビット加算器の出力を求める論理式を求める．各ビットにおける桁上げ信号 $c_i(0 \leq i \leq 2)$ と c_{out} の論理式は，次の式で与えられる．

$$c_0 = a_0b_0 + a_0c_{in} + b_0c_{in} \tag{2.5}$$

$$c_i = a_ib_i + a_ic_{i-1} + b_ic_{i-1} \quad (1 \leq i \leq 2) \tag{2.6}$$

$$c_{out} = a_3b_3 + a_3c_2 + b_3c_2 \tag{2.7}$$

ここで x_i は，複数ビットの信号 x の i ビット目を意味する．一方，各ビットにおける和信号 $sum_i(0 \leq i \leq 3)$ の論理式は，次の式で与えられる．

$$sum_0 = a_0 \oplus b_0 \oplus c_{in} \tag{2.8}$$

$$sum_i = a_i \oplus b_i \oplus c_{i-1} \quad (1 \leq i \leq 3) \tag{2.9}$$

以上の論理式より，**図 2.5**に示す 4 ビット加算器モジュール (`add4`) の SFL 記述が得られる．

図 2.5に示す SFL 記述では，複数ビットの信号を扱っている．3 行目，4 行目はモジュールの入力端子の宣言であるが，2 つの入力 `a` と`b` は，共に 4 ビット幅であることを表現している．SFL において，複数ビットの信号を有する端子は次のように記述する．

信号端子名 < ビット幅 >

10 行目，11 行目は，**内部データ端子**の宣言である．内部データ端子は，モジュール内におけるデータの受渡しに利用する．C 言語等のプログラミング言語における関数 (手続き) 内のローカル変数と同様なものだと考えれば良い．　内部データ端子には，`bus`, `bus_v`, `sel`, `sel_v` の 4 種類がある．これらには，論理合成時に 3 ステートを用いたバスに置き換えるか (`bus` と`bus_v`)，あるいは，マルチプレクサで構成するか (`sel` と`sel_v`) の違いがある．末尾に `_v` があるかないかの違いは，論理圧縮時において前後の論理圧縮を別々に行なうか (`_v` なし)，あるいは，全体をまとめて行なうか (`_v` あり) の違いである．

```
1  /* (add4.sfl) */
2  module add4 {
3      input  a<4>;        /* data input (4bit width) */
4      input  b<4>;        /* data input (4bit width) */
5      input  cin;         /* carry input */
6      output sum<4>;      /* sum output (4bit width) */
7      output cout;        /* carry output */
8      instrin enable;     /* control terminal */
9
10     sel_v c0, c1, c2, c3;
11     sel_v sum0, sum1, sum2, sum3;
12
13     instruct enable par {
14         /* 各ビットにおける桁上げ信号 */
15         c0 = (a<0> & b<0>) | (a<0> & cin) | (b<0> & cin);
16         c1 = (a<1> & b<1>) | (a<1> & c0)  | (b<1> & c0);
17         c2 = (a<2> & b<2>) | (a<2> & c1)  | (b<2> & c1);
18         c3 = (a<3> & b<3>) | (a<3> & c2)  | (b<3> & c2);
19
20         /* 各ビットにおける和信号 */
21         sum0 = a<0> @ b<0> @ cin;
22         sum1 = a<1> @ b<1> @ c0;
23         sum2 = a<2> @ b<2> @ c1;
24         sum3 = a<3> @ b<3> @ c2;
25
26         sum = sum3 || sum2 || sum1 || sum0;     /* ビットの連結 */
27         cout = c3;
28     }
29 }
30 /* End of file (add4.sfl) */
```

図 2.5: 4 ビット加算器モジュールの SFL 記述

内部データ端子を用いる場合，内部データ端子の種類と内部データ端子名を与えて**内部データ端子宣言文**を記述する．同じ種類の内部データ端子の場合，データ端子名をカンマ (‘,’) で区切って複数記述することができる．**図 2.5**のモジュール `add4` では，内部データ端子を 8 個使用している (`c0` 〜`c3`, `sum0` 〜`sum3`)．

ビット幅が 2 以上の信号における特定ビットの信号の参照は，次のように記述することで表現される．

信号端子名 < ビット位置 >

ビット位置は，ビット幅が n ビットの信号の場合， 0 から $(n-1)$ の整数値で与える．

26 行目では，ビットの連結を表現している．`sum3` から`sum0` の 4 個の内部端子を連結して， 4 ビットの信号を生成していることを表現している．ビットの連結においては，左側に書かれた信号 (`sum3` 側) が MSB 側 (上位ビット側) となる．

2.4.4 サブモジュールを用いた 4 ビット加算器の階層記述

2.4.3では，出力信号を得る論理式を求めて，その論理式を直接記述する方法を説明した． ここでは， 2.4.2で記述した全加算器をサブモジュール (部品) として用いて記述する方法を述べる．

ここで記述する 4 ビット加算器の外部仕様は， 2.4.3で述べたものに準ずる．全加算器を 4 個用いて構成すると，内部のブロック図は**図 2.6**のようになる．

まず，サブモジュールとして利用する全加算器のヘッダファイル (**図 2.7**) を用意する． ヘッダファイルでは，他のファイルで SFL 記述されたモジュールのインタフェースを`declare` 文を用いて宣言する．**図 2.7**の 4 行目から 6 行目は，**図 2.3**で示した SFL 記述の入力 / 出力端子および制御入力端子の記述と同一である． 8 行目の `instr_arg` で引数の並びを宣言している．例では，`a`，`b`，`cin` の並びでモジュール`fulladder` を使用することを宣言している．ここで説明している全加算器の例では，制御入力端子は`enable` の 1 個しか用いられていないが，複数の制御入力端子を用いた記述も行なうことができる．この場合，各制御入力端子毎に `instr_arg` を用いて端子の並びを宣言することができる．

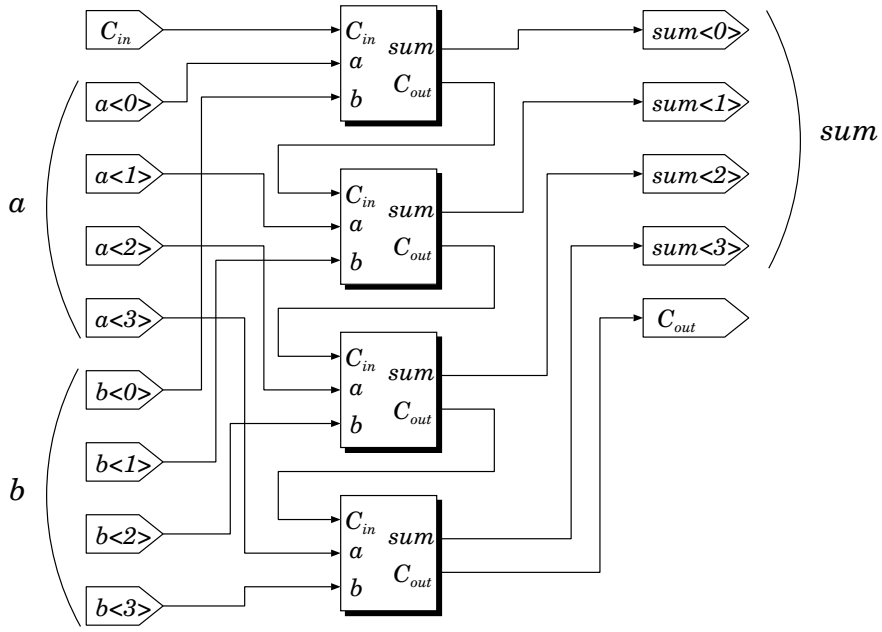

図 2.6: 4 ビット加算器の内部構造

```
1  /* (fulladder.h) */
2
3  declare fulladder {
4      input a, b, cin;
5      output sum, cout;
6      instrin enable;
7
8      instr_arg enable(a, b, cin);
9  }
10 /* End of file (fulladder.h) */
```

図 2.7: 全加算器の SFL 記述 (ヘッダファイル)

サブモジュールを利用して記述した 4 ビット加算器の SFL 記述を**図 2.8**に示す． **図 2.8**の 2 行目にて，ヘッダファイルの読込みを行なっている．これは， C 言語における

`#include "ファイル名"`

と同様，他のファイルを読込むための記述である． SFL 記述における

`%i "ファイル名"`

は C 言語の `#include` と同様，指定したファイルを読み込むものであり，読み込みを行なっているファイル (例えば，**図 2.8**に示すファイル) の中に直接読み込むファイル (**図 2.7**に示すファイル) の内容を書いたのと同じ意味である．つまり，‘`declare` ...’ というヘッダファイルの内容を，モジュールの定義をしているファイルに直接書いてもよいのである．しかし，本テキストでは，モジュール定義した SFL 記述の再利用の観点から`declare` 宣言によるモジュールのインタフェースはそのモジュール定義のファイルとは別ファイルとして用意することにする．

図 2.8の 12 行目では， 1 ビット全加算器サブモジュール `fulladder` の実体 (インスタンス) を 4 個生成し，それぞれに `fa0` 〜`fa3` という名前をつけていることを表現している．

16 行目から 19 行目にて，サブモジュール`fulladder` への信号線の接続を行なっている．信号線の接続は，次の形式で記述する．

サブモジュールインスタンス名 . 制御入力端子名 (入力端子の並び);

```
1  /* (add4_v2.sfl) */
2  %i "fulladder.h"
3
4  module add4_v2 {
5      input  a<4>;        /* data input (4bit width) */
6      input  b<4>;        /* data input (4bit width) */
7      input  cin;         /* carry input */
8      output sum<4>;      /* sum output (4bit width) */
9      output cout;        /* carry output */
10     instrin enable;     /* control terminal */
11
12     fulladder fa0, fa1, fa2, fa3;
13
14     instruct enable par {
15         /* サブモジュール */
16         fa0.enable(a<0>, b<0>, cin);
17         fa1.enable(a<1>, b<1>, fa0.cout);
18         fa2.enable(a<2>, b<2>, fa1.cout);
19         fa3.enable(a<3>, b<3>, fa2.cout);
20
21         sum = fa3.sum || fa2.sum || fa1.sum || fa0.sum;/* ビットの連結 */
22         cout = fa3.cout;
23     }
24 }
25 /* End of file (add4_v2.sfl) */
```

図 2.8: 全加算器を利用した 4 ビット加算器の SFL 記述

「サブモジュールインスタンス名」は，12 行目のサブモジュールの宣言で名前づけした名前を用いる．「制御入力端子名」は，使用するサブモジュールを記述したファイル (ここでは，モジュール `fulladder` の SFL 記述ファイル “`fulladder.sfl`”) で `instrin` を用いて宣言した名前を用いる．「入力端子の並び」は，ファイル “`fulladder.h`” において`declare` でモジュールのインタフェースを宣言しているが，この中の `instr_arg` 文で記述した入力端子の並びと同一となるように使用する．

17 行目から 19 行目において，`fa0` から `fa2` の桁上げ出力 `cout` の信号を参照している．出力信号の参照は，通常，次の形式で記述する．

サブモジュールインスタンス名 . 制御入力端子名 (入力端子の並び). 出力端子名

この形式にしたがうと 17 行目の `fa0.cout` は，次のように記述する必要がある．

```
fa0.enable(a<0>, b<0>, cin).cout
```

ここでは，16 行目にて `fa0` に対して端子の接続が行なわれているため 17 行目では，‘`.enable(a<0>, b<0>, cin)`’ の部分を省略して記述することができる．18 行目，19 行目も同様の理由である．

図 2.9に 4 ビット加算器のモジュール階層を示す．モジュール `add4_v2` は，4 個のサブモジュール `fa0` 〜`fa3` で構成されていることを示している．

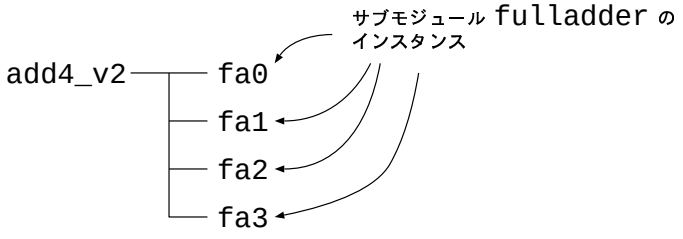

図 2.9: 4 ビット加算器のモジュール階層

2.4.5 フリップフロップ

ここでは，フリップフロップを例にレジスタ (記憶素子) を用いた順序回路の記述方法について述べる．

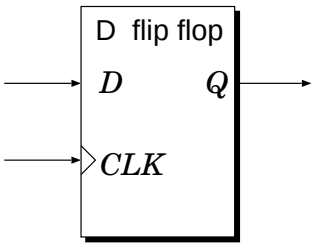

図 2.10: D フリップフロップの外部仕様

一般的な D フリップフロップの外部仕様を**図 2.10**に示す．**図 2.10**に示す D フリップフロップは，エッジトリガ型のフリップフロップで，クロック信号 CLK が立ち上った際の入力信号 D の値を保持し，出力信号 Q に保持している値を出力する．**図 2.11**に D フリップフロップのタイミングチャートを示す．

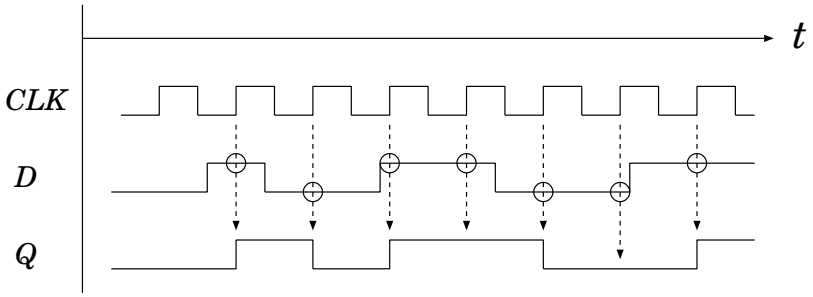

図 2.11: D フリップフロップのタイミングチャート

D フリップフロップの SFL 記述

D フリップフロップの SFL 記述を**図 2.12**に示す． **図 2.12**において， 6 行目の ‘`reg a;`’ という記述は， 1 ビッ

```
1  /* (D_FF.sfl) */
2
3  module D_FF {
4      input D;
5      output Q;
6      reg a;              /* 1 ビットのレジスタ */
7      instrin enable;
8
9      instruct enable par {
10         a := D;         /* レジスタ a への信号の取り込み */
11         Q = a;          /* レジスタ a の出力と Q を接続 */
12     }
13 }
14 /* End of file (D_FF.sfl) */
```

図 2.12: D フリップフロップの SFL 記述

トのレジスタ (記憶素子) を用意しそのレジスタに ‘`a`’ という名前をつけていることを意味する． レジスタの指定には，`reg` 以外に `reg_wr`，`reg_ws` を利用できる．これらの違いは初期値にある．`reg` を用いて宣言されたレジスタは，初期値が不定値である．`reg_wr` を用いて宣言されたレジスタは初期値が 0，`reg_ws` を用いて宣言されたレジス

タは初期値が 1 に設定される．レジスタの初期値が問題となる場合，これらを使い分けて記述することで，レジスタを初期化するための特別な回路を用意する必要がなくなる．

10 行目の‘`a := D;`’という記述は，クロック信号に同期して入力信号 `D` の値をレジスタ `a` に取り込むことを意味する．回路図的な解釈としては，`D` という信号線がレジスタの入力線と接続されているということに相当する．**図 2.10**に示す D フリップフロップでは，クロック信号 CLK が描かれているが，上記の SFL 記述では， CLK に相当する入力は記述されていないことに注意されたい．**SFL では，2.2.1で述べたように同期回路のみを扱うため，クロック信号を明示的に書く必要はない**．

11 行目の‘`Q = a;`’という記述は，レジスタ`a` の出力をこのモジュールの出力信号 `Q` として出力することを意味する．ここでは，‘`:=`’と‘`=`’の違いに注意して欲しい．‘`:=`’はレジスタへの値の取り込みを意味しており，‘`:=`’の左辺は記憶素子 (レジスタ，メモリ) が要求される．一方，‘`=`’は信号線の接続を意味している．

2.4.6 カウンタ

カウンタは，数を数える (カウント) する際に用いられる基本的な順序回路である．本節では，順序回路の記述例として 4 ビットカウンタの設計について述べる．

最も簡単なカウンタは，クロック信号の変化 (信号の立ち上がり，または立ち下がり) の際に数を数える (カウントアップする) カウンタである．ここでは，まず，4 ビットの数 (0 ～ 15) を扱う 4 ビットカウンタを設計してみよう．設計する 4 ビットカウンタのモジュール名は，`count4` とする．

はじめにカウンタの入出力を考えてみよう．入力として必要な信号は，(1) クロック信号と (2)`enable` 信号である．`enable` 信号は，数をカウントする / しないを制御するための信号である．`enable` 信号がアサートされている場合に限って，カウンタの値を更新する． 2.4.5で説明したように， SFL ではクロック信号を明示的に記述する必要はない．一方，`enable` 信号は，カウンタの動作を制御するための信号 (制御信号) なので，制御入力端子として記述すればよい．出力としては，カウンタで保持している 4 ビットの値 (カウント値) を出力する．

次にカウンタの内部について考えてみよう．カウンタはカウント値を保持する必要があるので，レジスタを用いる．ここで設計するカウンタは 4 ビットなので， 4 ビットレジスタを用意すればよい．さて，値の更新はどのようにすれば良いだろうか？ 4 ビットレジスタで保持している値に 1 を加えた値を次のレジスタの値とするようにデータパスを構成すれば値の更新ができる．カウンタの次の値を求めるには， 2.4.3で設計した 4 ビット加算器を利用すれば良いだろう． 4 ビットレジスタと 4 ビット加算器を**図 2.13**に示すように接続する．ここでは， 4 ビット加算器を 1

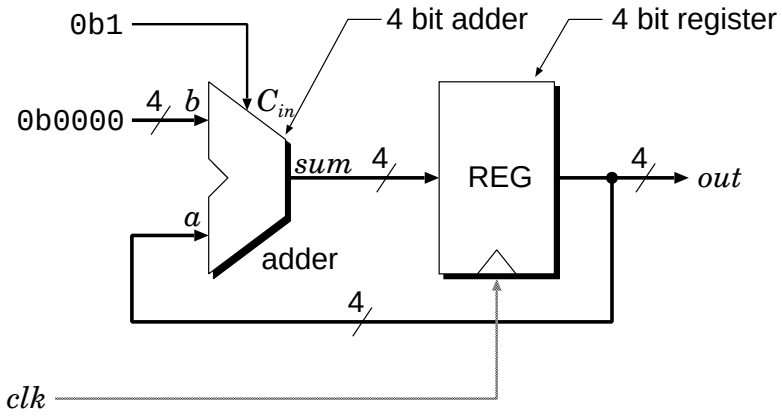

図 2.13: 4 ビットカウンタのデータパス

を加えるためだけに利用するので，入力 b には定数ゼロ (`0b0000` [1])，キャリー入力 C_{in} には定数 1 (`0b1`) を与える．**図 2.14**に 4 ビットカウンタのタイミングチャートを示す．例えば， 4 ビットレジスタに値 2(`0b0010`) が保持されてた場合 (**図 2.14**の **(a)**)，加算器の出力 sum は， 2 に 1 を加えた値 3 (`0b0011`) となる．クロック信号が変化すると，加算器の出力がレジスタに取り込まれて値が更新される (**図 2.14**の **(b)**).

[1] `0b0000` の先頭の `0b` は，表記の値が 2 進数 (binary digit) であることを意味する．

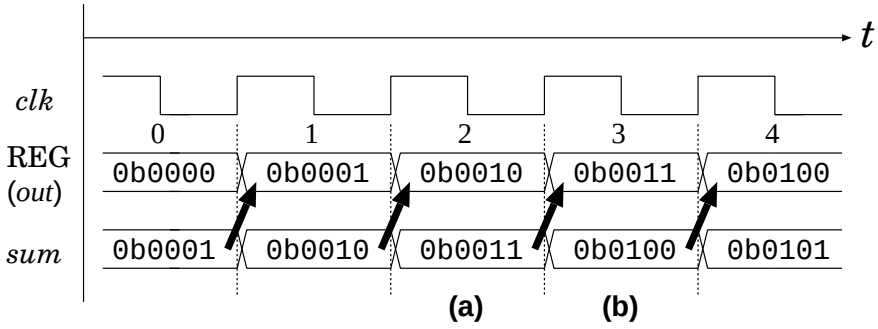

図 2.14: 4 ビットカウンタのタイミングチャート

4 ビットカウンタの SFL 記述

4 ビットカウンタの SFL 記述を**図 2.15**に示す．**図 2.15**に示す記述は，**図 2.12**に示した D フリップフロップの

```
1  /* (count4.sfl) */
2  %i "add4.h"
3
4  module count4 {
5      output out<4>;       /* sum output (4bit width) */
6      instrin enable;      /* control terminal */
7
8      sel_v sum<4>;        /* internal terminal */
9      reg_wr REG<4>;       /* 4 bit register */
10     add4 adder;
11
12     instruct enable par {
13         REG := adder.enable(0b0000, out, 0b1).sum;
14         out = REG;
15     }
16 }
17 /* End of file (count4.sfl) */
```

図 2.15: 4 ビットカウンタの SFL 記述

記述と似ていることがわかるだろう．大きく異なる点は，(1) データ幅が 4 ビットになった点，(2) データパスのループ部分に加算器が入った点である．細かいところでは，9 行目のレジスタの宣言で，`reg_wr` を用いている点が挙げられる．これは，初期状態でレジスタの値を 0 に設定したいためである．`reg` を用いた場合，レジスタの初期値は不定となる．この場合，加算器の出力も不定値となるためにクロックを進めても値が不定値から変化しないことになり，カウンタとして動作しない．このことを後述する動作シミュレーションにて確かめるとよいだろう．

なお，2 行目で読み込んでいる 4 ビット加算器のヘッダファイル “`add4.h`” の記述を**図 2.16**に示す．

2.4.7 10 進カウンタ

2.4.6で述べたカウンタは，4 ビットのカウンタで 0 から 15 の数を数えるものであった．ここでは，信号の値によって異なる動作をおこなうための SFL 記述について説明する．例として，0 〜 9 までの数を数える 10 進カウンタを用いる．設計する 10 進カウンタのモジュール名は，`count10` とする．

基本的な部分は，2.4.6で述べた 4 ビットカウンタと同じである．違いは，レジスタの値が 9 (`0b1001`) の際に次の値を 1 加えた 10 (`0b1010`) にするのではなく，0 (`0b0000`) にする点である．**図 2.17**にデータパスを示す．**図 2.17**では，加算器出力の値によってレジスタに設定する値を変えている．図の controller における論理はつぎのとおりである．加算器出力の値が 10 の場合マルチプレクサの上側，すなわち 0 をレジスタに設定し，加算器出力の値が 10 以外の場合マルチプレクサの下側，すなわち加算器出力をそのままレジスタに設定する．

```
1  /* (add4.h) */
2  declare add4 {
3      input  a<4>;        /* data input (4bit width) */
4      input  b<4>;        /* data input (4bit width) */
5      input  cin;         /* carry input */
6      output sum<4>;      /* sum output (4bit width) */
7      output cout;        /* carry output */
8      instrin enable;     /* control terminal */
9
10     instr_arg enable(a, b, cin);
11 }
12 /* End of file (add4.h) */
```

図 2.16: 4 ビット加算器のヘッダファイル “add4.h” の記述

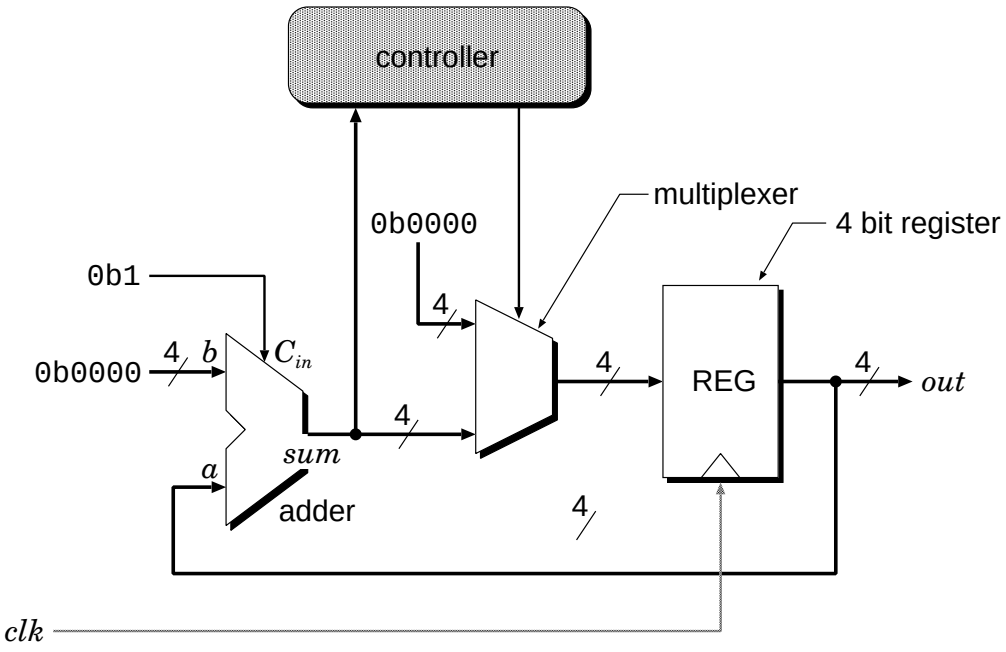

図 2.17: 10 進カウンタのデータパス

10 進カウンタの SFL 記述

2.4.6で示した 4 ビットカウンタの SFL 記述に，マルチプレクサ，コントローラ等を追加すれば 10 進カウンタが記述できる．加算器出力からマルチプレクサの制御を行なう部分は，簡単な論理式でも記述できるが，ここでは，SFL らしい記述の方法を説明する．

SFL には，信号の値によって異なる動作を書くための構文が用意されている．プログラミング言語で例えると，if 文や switch 文に相当する機能である．ここでは， alt 文について説明する． alt 文はつぎのように用いる．

```
alt {
   条件 1:  動作 1;
   条件 2:  動作 2;
   ...
   条件 n:  動作 n;
   else:  動作 n+1;
}
```

上の記述では， “条件 $_i$” が成立する場合， “動作 $_i$” の部分が有効となる．どの条件も成立しなかった場合， “`else:` ” の後に書かれた “動作 $_{n+1}$” が有効となる．条件の一致は，上に書かれているものが優先される． 成立する条件すべての動作を有効とする場合， any 文を用いるとよい．

alt 文を用いて記述した 10 進カウンタを**図 2.18**に示す． 9 行目， 10 行目にて 4 ビット幅の内部端子の宣言を行

```
1  /* (count10.sfl) */
2  %i "add4.h"
3
4  module count10 {
5      output out<4>;       /* sum output (4bit width) */
6      instrin enable;      /* control terminal */
7
8      sel_v sum<4>;        /* internal terminal */
9      sel_v adder_sum<4>;
10     sel_v reg_in<4>;
11     reg_wr REG<4>;       /* 4 bit register */
12     add4 adder;
13
14     instruct enable par {
15         adder_sum = adder.enable(0b0000, out, 0b1).sum;
16         alt {
17             (adder_sum == 0b1010): reg_in = 0b0000;
18             else: reg_in = adder_sum;
19         }
20         REG := reg_in;
21         out = REG;
22     }
23 }
24 /* End of file (count10.sfl) */
```

図 2.18: 10 進カウンタの SFL 記述

なっている． 16 行目から 19 行目の alt 文で異なる動作を記述している．加算器出力 (`adder_sum`) の値が 10 (`0b1010`) の場合レジスタ入力 (`reg_in`) を`0b0000` とし (17 行目)，それ以外の場合は加算器出力 (`adder_sum`) の値をレジスタの入力としている (18 行目).

2.5 SECONDS を用いた動作シミュレーション

回路を HDL で記述した後に行なうことは，その記述が正しいか確かめることである．ここで言う「正しさ」は，その記述が文法的に正しいかということと，設計しようとしている回路の仕様に合った動作をする記述となっているか (意味的な正しさ) である． SFL で記述された回路の「正しさ」を確かめるためには， PARTHENON システムの動作シミュレータ SECONDS を用いる．設計した回路記述を SECONDS に読み込ませ文法的に正しいか？ という

ことと，設計した回路に対してデータ (入力) を与えてシミュレーションを行ない意図した動作をするか？ を確かめる．ここでは，SECONDS の簡単な使用方法を示す．

2.5.1 起動と終了

SECONDS は，会話型のシミュレータである．シェルプロンプトより ‘seconds’ とタイプすればシミュレータが起動する．

```
% seconds                    ··· SECONDS の起動
+-------------------------------------------------------------+
| SECONDS         2.4.0 2000/08/31 (i386-FreeBSD-4.1-RELEASE) |
|              This program is a part of the PARTHENON system. |
|                Licence required; Copyright (C) 1989-2000 NTT |
+-------------------------------------------------------------+

SECONDS>
```

SECONDS が起動されると，上記のようにバージョン情報等[2]を表示し，その下に ‘SECONDS>’ というプロンプトが表示される．ユーザは SECONDS の各種コマンドをタイプし，シミュレーションを行なう．

シミュレータを終了する際には，bye コマンドを用いる．あるいは，ctrl-D (Control キーを押しながら D をタイプ) でもシミュレータを終了することができる．

```
SECONDS> bye
*****************************
*  Good bye, see you later.  *
*****************************

%
```

本テキストでは，ユーザが与える入力部分に下線を引くことによって，OS やプログラム等から端末に出力されるメッセージと区別する．

2.5.2 回路記述ファイル (SFL ファイル) の読込み

シミュレーションする回路記述ファイルを読込む．回路記述のファイルは，拡張子が ‘sfl’(‘.sfl’ で終るファイル名) である必要がある．

回路記述ファイルを読込みには，sflread コマンドを用い，引数として読み込むファイル名を与える．

```
SECONDS> sflread foo.sfl
SECONDS>
```

読み込むファイルに構文エラー等があれば，そのメッセージが出力される．エラーがあった場合，エディタで回路記述ファイルの修正を行ない，エラーがなくなるまで繰り返す．

サブモジュールを使用している場合，使用しているサブモジュールの SFL 記述ファイルをすべて読み込む必要がある．例えば，2.4.4で説明した**add4_v2** のシミュレーションを行なう場合，サブモジュールとして利用している全加算器 fulladder の SFL 記述ファイル fulladder.sfl も読み込んでおく必要がある．

2.5.3 回路のインストール

読み込んだ回路のシミュレーションを行なうには，シミュレータの内部に「**シミュレーション・イメージ**」と呼ばれるシミュレーション実行のためのデータを生成する必要がある．そのために用意されているコマンドが，install である．階層的に記述された回路から自動的にシミュレーション・イメージを生成するためには，autoinstall コマンドを用いると便利である．

```
SECONDS> autoinstall foo
SECONDS>
```

install コマンドを用いる場合は，モジュールを追加する位置を第 2 引数で指定する．

[2]本テキスト執筆時の最新バージョンは，2.4.1 (2001/12/03) である．

```
SECONDS> install foo /
SECONDS>
```

最初のinstall コマンドで指定する位置は，シミュレーションイメージのルート (‘/’) でなければならない．

2.5.4 入力値の設定

シミュレーションを行なうには，シミュレーション対象の回路 (モジュール) に対して入力値を設定する必要がある．入力値の設定は，set コマンドを用いる．set コマンドの書式は，次のとおりである．

set《入力端子名》《値》

ここで，第 2 引数で与える《値》は，デフォルトで 2 進数である．

端子 in1 に 値 0 を設定するには次のようにタイプする．

```
SECONDS> set in1 0
SECONDS>
```

ビット幅の大きい値を設定する際など，2 進数で値を記述するのは面倒な場合がある．このような場合，値を 16 進数で与えることもできる．値の先頭に ‘X’ をつけると，その値は 16 進数とみなされる．

信号の値は，デフォルトでは設定した時点のクロックにおいてのみ有効であり，クロックを更新すると無効となる．クロックを進める毎に毎回，信号の値を設定するのは手間がかかる．クロックが更新されても，信号の値を保持するためには，hold コマンドを用いる．hold コマンドにより指定した信号は，クロック値を更新しても，その信号の値はクロック値を更新する前の値を保持したままとなる．一度 hold 設定された信号は，set コマンドにより値を書換えても hold 設定は保持される．hold 設定を解除するには，release コマンドを用いる．

2.5.5 入力値 / 出力値の表示

print コマンド

回路の入力端子や出力端子の値や内部データ端子の値等を表示するには，print コマンドを用いる．print コマンドの書式は次のとおりである．

print 書式文字列 引数 ...

書式文字列には，特別な書式を除くと C 言語における printf 関数と同一である．

例えば，入力端子 in と出力端子 out の値を 16 進数で表示する場合，次のようにして用いる．

```
print "in=%x out=%x\n" in out
```

値を 2 進数で表示したい場合，書式文字列で ‘%x’ の代わりに ‘%b’ を用いる．一方，値を 10 進数で表示したい場合，‘%d’ を用いる．

シミュレータ内部では，シミュレーション時刻 (クロック) を保持している．この値を表示したい場合，‘%t’ という書式を用いる．この場合，‘%t’ に対応する引数は必要ない．

rpt_add コマンド

上述した print コマンドでは，コマンドを発行した時点における値を表示することができるが，シミュレーションを行なっている過程で何度も同じ信号端子の値を観測したい場合がある．この場合，そのたびに print コマンドをタイプするのは大変である．

SECONDS には，シミュレーションの時刻が進む毎に自動的に指定した値を表示する機能が備わっており，この機能を用いるにはrpt_add コマンドを用いて表示する書式と値を指定する．rpt_add コマンドの書式はつぎのとおりである．

rpt_add キー名 書式文字列 引数 ...

さきほど述べたprint コマンドと異なる部分は，最初に「キー名」があるという点であり，これ以外の部分は，printf コマンドと同じである．「キー名」は適当な文字列を与えれば良い．rpt_add コマンドを発行しておくと，以降，シミュレーション時刻が更新 (進む) 度に自動的に指定された書式でレポートが出力される．「キー名」を変えて登録することで，複数の表示書式を登録することができる．

表示を中止したい場合，

`rpt_rmv` キー名

により，指定した「キー名」のレポート機能をキャンセルできる．

2.5.6 シミュレーションの実行

シミュレーションを実行するには，`forward` コマンドを用いる．この `forward` コマンドは，クロック信号の値を更新するコマンドである．コマンドの引数で整数値の時刻を与えるとその時刻 (クロック値) まで実行を進める．引数で与える整数値の前に '+' を書くと，現在の時刻から指定した整数値の数を加えた時刻まで実行を行なう．引数を与えなかった場合，システムで設定されているデフォルト値 (Halmagedon) まで実行を進める．

2.5.7 スクリプトファイル

SECONDS は会話型のシミュレータであるため，シミュレーションを行なう度に，様々なコマンドを会話的にシミュレータに与える必要がある．しかし，同じシミュレーションを何回も繰り返して行ないたい場合など毎回，コマンドをキーボードより打ち込むのは大変な作業である．SECONDS には，シミュレータに与えるコマンドをあらかじめファイルに書いておき，そのファイルのコマンドを順次，処理してシミュレーションを進める方法が用意されている．このあらかじめコマンドを書いておくファイルを**スクリプトファイル**と呼ぶ．スクリプトファイルに書かれているコマンドを実行するには，`listen` コマンドを用いる．`listen` コマンドの引数で指定されたスクリプトファイルを読み込んで実行する．

論理ゲートをシミュレーションするための SECONDS のスクリプト (ファイル名は，"`gate.scr`" とする) を**図 2.19**に示す．**図 2.19**に示すスクリプトでは，2 つの入力 (`in1`，`in2`) に対してそれぞれ (0, 0)，(0, 1)，(1, 0)，(1, 1) という値を与えた際の出力信号 `out` を表示する．

```
1  # (gate.scr)
2  #
3  sflread $1.sfl
4  autoinstall $1
5  rpt_add A "t=%t in1=%b in2=%b out=%b\n" in1 in2 out
6  set enable 1
7  set in1 0
8  set in2 0
9  hold enable
10 hold in1
11 hold in2
12 forward +1
13 set in1 0
14 set in2 1
15 forward +1
16 set in1 1
17 set in2 0
18 forward +1
19 set in1 1
20 set in2 1
21 forward +1
```

図 2.19: 論理ゲートをシミュレーションするための SECONDS スクリプト

例えば，AND ゲートの SFL 記述ファイルが "`and.sfl`" の場合，下記の操作で AND ゲートの論理シミュレーションを行なうことができる．

```
SECONDS> sflread and.sfl                    SFL ソースの読込み
      SFL ソースを読み込んだことを伝えるメッセージが表示される
SECONDS> gate.scr and              スクリプトファイルの読込み，論理シミュレーションの実行
      シミュレーション結果が表示される
```

スクリプトファイル "`gate.scr`" に与える引数 (上記の例では，`and`) としてモジュール名を指定することで，指定したモジュールのシミュレーションを行なう．ここで与える引数 (文字列) は，スクリプトファイルでは '`$`*i*' で参

照することができる．ここで i は，何番目の引数を参照するかを指定する数字である．**図 2.19**の例では，3 行目と 4 行目に ‘**$1**’ が用いられている．これらは，スクリプトファイルを読み込む際に与えられた引数文字列に置き換えられる変数である．引数が複数与えられる場合，’**$2**’, ‘**$3**’, . . . と書くことにより 第 2 引数文字列， 第 3 引数文字列，. . . と参照することが可能である．

2.5.8 シミュレーション結果の保存

SECONDS にはシミュレーション結果をファイルに保存する機能 (**speak** コマンド，**speaka** コマンド) がある．

speak コマンドは，引数としてファイル名を 1 つ与える．このコマンド以降に行なったシミュレーション結果が引数で指定されたファイルに出力される．例えば，ファイル “**foo.result**” に結果を出力する場合，次のようにタイプする．

```
SECONDS> speak foo.result
SECONDS>
```

すでに存在するファイルに追加して記録したい場合，**speaka** コマンドを用いる．**speaka** コマンドも**speak** コマンド同様，引数としてファイル名を与える．

これらのコマンドを用いた場合，それ以降の結果が端末には全く表示されなくなるので注意されたい．この機能は，スクリプトファイルを用いてシミュレーションする際に利用することで，シミュレーション結果をファイルに保存することができる．

2.6 実験課題

【実験課題 2.1】
図 2.3に示した全加算器の SFL 記述ファイルを作成し， SECONDS で動作シミュレーションを行ない，正しく動作するかを確認せよ．シミュレーション用の SECONDS スクリプトは，**図 2.19**を参考に作成せよ．

【実験課題 2.2】
図 2.5に示した 4 ビット加算器の SFL 記述ファイルを作成し， SECONDS で動作シミュレーションを行ない，正しく動作していることを確認せよ．

【実験課題 2.3】
図 2.7に示した 1 ビット全加算器のヘッダファイル “**fulladder.h**” と**図 2.8**に示した 4 ビット加算器の SFL 記述ファイル “**add4_v2.sfl**” を作成し， SECONDS で動作シミュレーションを行ない，正しく動作していることを確認せよ．与える入力は，**【実験課題 2.2】**で用いたものと同一のものを用いる．ここでは全加算器のサブモジュールを使用しているので， SECONDS でシミュレーションを行なう際にサブモジュールの SFL 記述を読み込む必要があることに注意せよ．

【実験課題 2.4】
8 ビット加算器の SFL 記述を作成し，シミュレータを用いて動作確認を行なえ．

【実験課題 2.5】
図 2.20に示した 4 ビットカウンタの動作シミュレーションを行なう SECONDS スクリプト “**test_count4.scr**” を作成し，**図 2.15**に示した 4 ビットカウンタの SFL 記述 “**count4.sfl**” の動作シミュレーションを行なえ．

【実験課題 2.6】
図 2.18に示した 10 進カウンタの SFL 記述を作成し， SECONDS で動作シミュレーションを行ない，正しく動作していることを確認せよ． SECONDS スクリプトは，**図 2.20**に示したスクリプトを変更したもの (**count4** →**count10**) を用意すればよい．

```
1  # (test_count4.scr)
2  # SECONDS script for count4 simulation
3  #
4  sflread add4.sfl
5  sflread count4.sfl
6  autoinstall count4
7
8  speak count4.result
9
10 rpt_add A " %4t | 0b%b (%2d)   0b%b (%2d)\n" /adder/sum /adder/sum /out /out
11
12 print "[simulation result (count4)]\n"
13 print "------+-------------+-------------\n"
14 print " time | /adder/sum    /out\n"
15 print "------+-------------+-------------\n"
16 set enable 1; hold enable
17 print " %4t | 0b%b (%2d)   0b%b (%2d)\n" /adder/sum /adder/sum /out /out
18 forward +16
19
```

図 2.20: 4 ビットカウンタの SECONDS スクリプト

2.7 設計課題

【設計課題 2.1】
16 ビット加算器モジュール `add16` を設計せよ.

【設計課題 2.2】
アップカウンタとはクロック信号入力によって値が増加するカウンタのことで，2.4.6，2.4.7で述べたカウンタはアップカウンタである．**ダウンカウンタ**とはクロック信号入力によって値が減少するカウンタのことである．アップカウンタとダウンカウンタの双方の機能を有するカウンタを**アップダウンカウンタ**と呼ぶ．アップダウンカウンタでは，外部から与える入力によってアップカウントとして機能させるか，ダウンカウンタとして機能させるかを選択する．4 ビットのアップダウンカウンタ `ud_count4` を設計せよ．アップまたはダウンの機能は，入力端子 `up` の信号で選択すること．`up` が 1 の場合アップカウンタとして動作させ，`up` が 0 の場合ダウンカウンタとして動作させること．

2.8 まとめ

2.8.1 チェック項目

SFL 記述

- SFL 記述の内容を理解できる.
- 組合せ回路を SFL で記述できる.
- 順序回路を SFL で記述できる.
- 複数モジュールで構成させるハードウェアを SFL で記述できる.

動作シミュレータ: SECONDS

- SECONDS の各種コマンドを使える.
- SECONDS のスクリプトファイルを記述できる.
- 動作シミュレーションを行なえる.
- シミュレーション結果をファイルに保存できる.
- シミュレーション結果からハードウェアの動作をイメージできる (タイミングチャートが書ける).

論理回路

- 桁上げ伝搬加算器の構造，原理を理解した．
- カウンタの構造，原理を理解した．

2.8.2 本章で作成 / 設計したモジュール

1. 論理ゲート (`and`, `or`, `nand`, `xor`, `and2`)
2. 全加算器 (`fulladder`)
3. 4 ビット加算器 (`add4`)
4. 4 ビット加算器 (サブモジュール利用) (`add4_v2`)
5. D フリップフロップ (`D_FF`)
6. 4 ビットカウンタ (`count4`)
7. 10 進カウンタ (`count10`)
8. 16 ビット加算器 (`add16`)
9. 4 ビットアップダウンカウンタ (`ud_count4`)