情報工学実験

— ハードウェア実験 —

PARTHENON と SFL の簡易マニュアル

岡山大学大学院自然科学研究科

渡邊 誠也

2005 年 5 月 16 日

目次

第 1 章

高位論理合成システム PARTHENON 簡易マニュアル

本章は，高位論理合成システム PARTHENON の簡易マニュアルである．本マニュアルは，下記の URL

`http://www.kecl.ntt.co.jp/car/parthe/index_j.htm`

に置かれているマニュアルを元に作成した．本マニュアルに記載されていない部分や不十分な項目に関しては，上記 URL のマニュアルを参照されたい．

1.1 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，すでに，産業界において，実用あるいは研究試作用の ASIC 設計に利用されているほか，大学，高専等の教育機関においては，研究および教育目的で多数利用されている．

1.2 PARTHENON システムの構成

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

1.3 PARTHENON を用いた設計の流れ

PARTHENON を用いた LSI 設計の流れを**図 1.1**に示す．設計者は，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 を利用することで，いわゆるトップダウン設計の恩恵を最大限に受けることができる．つまり，設計者は，SFL によるアーキテクチャ・レベルの設計に専念することができるのである．

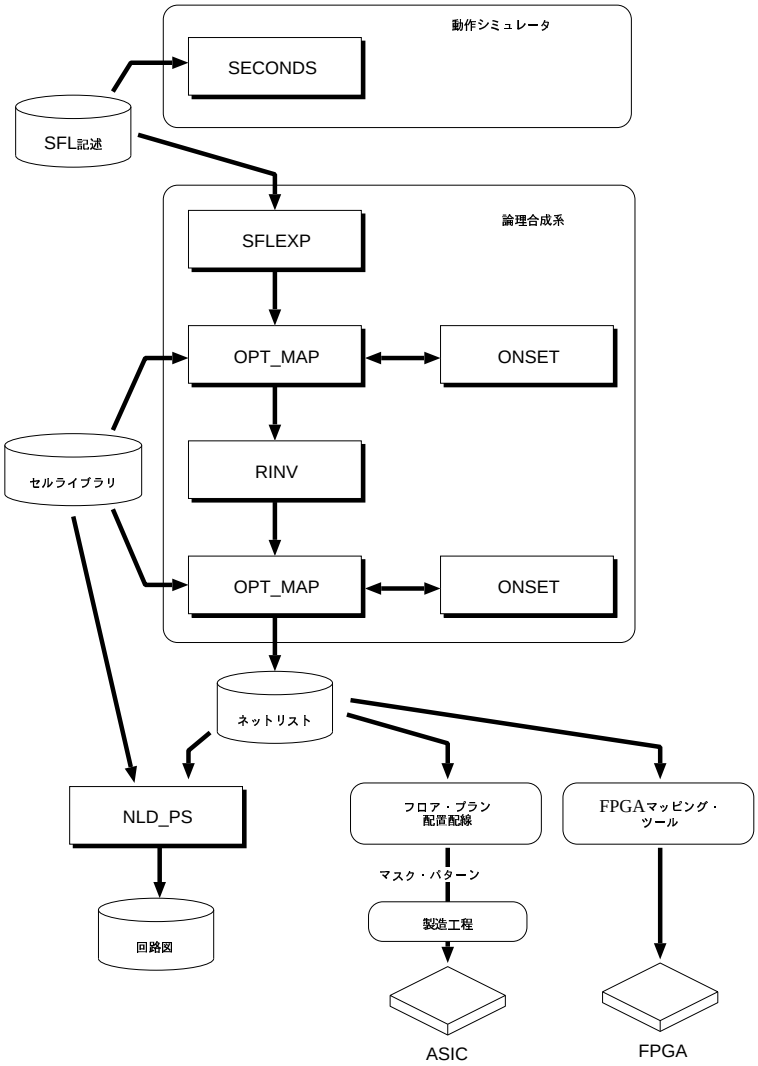

図 1.1: PARTHENON を用いた LSI 設計の流れ

1.4 PARTHENON システムの使用方法

本節では，PARTHENON システムの使用方法 について簡単に説明する．まず，環境変数の設定について述べ，動作シミュレータ SECONDS の使用方法を説明する．その後，論理合成・最適化処理系の使用方法について述べる．

1.4.1 環境変数の設定

PARTHENON システムを利用するには，

- 環境変数 `PARTHENON` の設定と，

- PARTHENON システムの各種コマンドパスのコマンドサーチパスへの追加

が必要である.

環境変数とパスを次のように設定する[1].

(C シェル系の場合)

```
setenv PARTHENON /usr/local/parthenon
set path = ($path ${PARTHENON}/com)
```

(B シェル系の場合)

```
PARTHENON=/usr/local/parthenon
export PARTHENON
PATH=${PATH}:${PARTHENON}/com
export PATH
```

上記の設定を “~/.cshrc ファイル”(C シェル系の場合), あるいは, “~/.profile ファイル”(B シェル系の場合) に追加しておく.

1.4.2 動作シミュレータ SECONDS の使用方法

SECONDS は, SFL で記述した回路の動作を検証するために用いる論理シミュレータである. なお, SECONDS の詳細なマニュアルは, 下記の URL

```
http://www.kecl.ntt.co.jp/car/parthe/hajimete/5shou.htm
```

を参照されたい.

SECONDS の起動と終了

SECONDS の起動は, シェルのコマンドプロンプトにて “seconds” とタイプすることで行なう.

```
% seconds
+-------------------------------------------------------------+
| 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> ” は, シミュレータのプロンプトである. SECONDS は, 会話的にシミュレーションを進めることができる.

SECONDS を終了するには, SECONDS のプロンプトにおいて “bye コマンド” を用いる.

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

%
```

SFL 記述ファイルの読込み

設計した回路の SFL 記述ファイルの読込みを行なうには, “sflread コマンド” を用いる.

```
SECONDS> sflread 《SFL ソース》
```

[1] PARTHENON システムが/usr/local/parthenon 以下のディレクトリにインストールされている場合.

モジュールのインストール

読み込まれた SFL ファイルの各モジュールは，**インストール**することにより， SECONDS 内部にモジュールのインスタンスが生成されシミュレーションを行なうことが可能となる．モジュールのインストールには，

- “`install` コマンド”
- “`autoinstall` コマンド”

のいずれかを用いる．

ここでは，“`autoinstall` コマンド” について説明する．”autoinstall コマンド” では，第 1 パラメータとしてトップモジュールを指定することにより，そのモジュール内部で使用しているサブモジュールも自動的にインストールされる．

```
SECONDS> autoinstall  《トップモジュール名》
```

レポート機能

(ここは未完成)

シミュレーションの開始

(ここは未完成)

スクリプトファイルを用いた一括処理

上では会話的にシミュレーションする方法を示したが，一連のコマンドを記述したスクリプトファイルを用意することで，シミュレーションにおけるコマンド入力を省力化できる．

SECONDS プロンプトにおいて，“`listen` コマンド” を用いてスクリプトファイル名を指定するか，あるいは，スクリプトファイル名をそのままタイプすることで，スクリプトファイルに記述されたコマンドが順番に処理される．

```
SECONDS> listen  《スクリプトファイル名》
```

あるいは，

```
SECONDS> 《スクリプトファイル名》
```

1.4.3 論理合成系の使用方法

PARTHENON システムの合成系は，次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

各プログラムの詳細に関しては， PARTHENON のマニュアルを参照されたい．

SFL 記述を合成・最適化するには，上記の各プログラムを 1 つ 1 つ呼び出す必要があるが，ここでは，論理合成と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” を用いた方法を紹介する．

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

auto コマンド

“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イリを指定する．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．

【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．

サブモジュールの論理合成

```
% auto 《モジュール名》nld1 DEMO demo
```

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 までの処理を行なう．

トップモジュールの論理合成

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 《モジュール名》nld1 DEMO demo
```

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．

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，モジュールadd4 の場合，ディレクトリはadd4.1st 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．

```
% cp 《サブモジュール名》.1st/*.nld 《トップモジュール名》.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，最適化処理を行なう．

```
% auto 《モジュール名》 nld4 DEMO demo
```

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．

```
% auto 《モジュール名》 ps DEMO demo
```

処理がエラーなく終了すると，カレントディレクトリに “《モジュール名》.ps” という名前のファイルが生成される．

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[2]を用いる．

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．

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

第 2 章

SFL 簡易マニュアル

2.1 はじめに

SFL(Structured Function description Language) は，ハードウェアの動作を記述する言語である．
本章では， SFL の概念，形式，意味を簡単に説明する．

2.2 記述の単位

2.2.1 モジュールとサブモジュール

SFL における記述の基本単位は，**モジュール**と呼ばれる．モジュールは，物理的な境界をもつ部品（厳密にはその種類）を表し，階層化の単位となる．モジュールは，その定義の中で他のモジュール (すなわち，**サブモジュール**) を使用することができる．サブモジュールを用いて回路を階層表現する際には，下位モジュール (サブモジュール) のインタフェースをあらかじめ宣言しておく必要がある．下位モジュールの内容は，別途，定義する．
例えば，モジュール A がモジュール B を使用する場合には，次のように記述する．

```
declare B { ... }          ; モジュール B の宣言

module A { ... B ... }     ; モジュール B を用いたモジュール A の定義

module B { ... }           ; モジュール B の定義
```

`declare` はモジュールのインタフェースの宣言を，`module` はモジュールの定義を表すキーワードである．
階層表現を行なう際，下位のモジュールの 1 個 1 個 (インスタンス) をサブモジュールと呼ぶ．例えば，モジュール `X` の中で，モジュール `Y` を部品として 2 つ使用している場合，モジュール `Y` の 2 つの部品 (インスタンス) を区別できる必要がある．それら 2 つの部品に名前をつけることで区別するが，この個々の部品につけれらた名前のことをインスタンス名と呼ぶ．

2.2.2 機能回路

論理合成の対象としない特別なモジュールのことを**機能回路**と呼ぶ．これは，「既存の回路」つまり内部構造がすでに外部で定義されているような部品をモジュールとして利用する場合に用いる．
機能回路は，キーワード `circuit` を用いて次のように記述する．

```
circuit C { ... }          ; 機能回路モジュール B の定義
```

2.3 動作の主体と客体

SFL では，「レジスタ A の内容とレジスタ B の内容を加えて，その結果をレジスタ C へ書き込んで，次に ...」というような制御の流れをそのまま手続きとして記述する．そのためモジュールの要素は，「制御を行なう側: **主体**」と「制御を受ける側: **客体**」の 2 つに分けて考える．

値の参照，演算，レジスタやメモリへの書込み，データ端子への出力など，すべてのデータ転送・加工動作は，「制御端子，または，ステージの制御により実施される」ものと考え，制御端子とステージを**動作の主体**と呼ぶ．一方，レジスタ，メモリ，データ端子は，「制御を受ける側の要素」と考え，**動作の客体**と呼ぶ．

制御端子あるいは，ステージは，他の動作主体から起動されると，その制御端子あるいはステージに対応づけられた動作を行なう．

2.3.1 ステージの動作と状態

制御端子の動作は，起動されたマシンサイクル内で完了する．一方，ステージの動作は，複数のマシンサイクルに渡って実行される．そこで，ステージが動作中であることを記憶するためのレジスタ (これを**タスクレジスタ**と呼ぶ) を用意し，これを**タスク**と呼ぶ．したがって，「ステージは，そのステージのタスクがセットされると動作を開始し，いったん動作を開始すると，タスクがリセットされるまで複数マシンサイクルに渡って動作を続ける」ということになる．

ステージのタスクは，「他の動作主体からジョブの生成，またはジョブの転送を受けるとセット」され，「そのステージ自身のジョブの転送，あるいはジョブの終了によりリセット」される．つまり，ステージの起動は，タスクのセットにより行なうということである．制御端子やステージの起動方法をまとめたものを**表 2.1**に示す．

表 2.1: 動作主体の起動

場合分け				形式
制御端子	起動された マシンサイクルで 完了	制御端子の起動		制御端子名 (引数，...);
ステージ	開始	タスクの セット	ジョブの生成	generate ステージ名. タスク名 (引数，...);
			ジョブの転送	relay ステージ名. タスク名 (引数，...);
	終了	タスクの リセット	ジョブの転送	relay ステージ名. タスク名 (引数，...);
			ジョブの終了	finish;

ステージにおける複数マシンサイクルにわたる動作は，複数の状態と状態遷移により表現する．ステージの状態は状態レジスタにより保存され，状態遷移はこのレジスタを更新することである．ステージが停止している (タスクがリセットされている) ときは，ステージの状態は変化しない．ステージは直前に停止した状態で再起動される．

ステージのいくつかの状態をソフトウェアのサブルーチンのように**セグメント**としてまとめることができる．セグメントは，戻り状態を保持することにより，状態遷移の複数箇所から遷移してまた戻ることができるようにした状態と状態遷移のひとかたまりである．

制御端子とステージは，起動するときに引数により付加情報を与えることができる．

2.4 外部端子と構成要素の定義

2.4.1 モジュールの構造

モジュールの記述は，次の 2 つの部分に分けることができる．

外部端子 そのモジュールの外側とのインタフェースを記述する部分

構成要素 そのモジュール自体の内側を記述する部分

表 2.2に外部端子の分類とタイプ名を，**表 2.3**に構成要素の分類とタイプ名を示す．

外部端子は，さらに動作の主体か客体かによって**制御端子**と**データ端子**とに分けることができる． 制御端子には，外部からの制御を表す**制御入力端子**と，外部への制御を表す**制御出力端子**がある． データ端子には，外部からのデータ入力を表す**データ入力端子**，外部へのデータ出力を表す**データ出力端子**，外部からのデータ入力と外部へのデータ出力の双方向のデータを表す**データ双方向端子**がある．

構成要素には，モジュール内部の要素ということで，**制御内部端子**，**ステージ**，**データ内部端子**，**レジスタ**，**メモリ**，**サブモジュール**がある．制御内部端子とステージが動作の主体であり，データ内部端子，レジスタ，メモリが動作の客体である．サブモジュールは，その外部端子の種類毎に動作の主体か客体となる．

表 2.2: 外部端子

分類		タイプ名
制御端子	制御入力端子	instrin
	制御出力端子	instrout
データ端子	データ入力端子	input
	データ出力端子	output
	データ双方向端子	bidirect

表 2.3: 構成要素

分類		タイプ名
制御系	制御内部端子 ステージ	instrsefl
データ系	データ内部端子 レジスタ メモリ	bus, bus_v, sel, sel_v reg, reg_ws, reg_wr mem
混合	サブモジュール	モジュール名

2.4.2 構成要素定義の基本形式

外部端子とステージを除く構成要素は，次の形式で定義する．

タイプ名　インスタンス名 { ， インスタンス名 }* ；

ここで，タイプ名は，組込みのタイプ名，あるいは，他のモジュール名である．

2.4.3 データ内部端子の定義

データ内部端子定義用のタイプ名には，sel, sel_v, bus, bus_v の 4 種類がある．sel は論理ゲートによるセレクタを表し，bus は 3 ステートによるセレクタを表す．これらのタイプ名で定義したデータ内部端子では，論理合成時にそのデータ内部端子の前後が別々に論理圧縮される．つまり，データ内部端子は，論理合成結果に含まれることになる．

一方，sel_v とbus_v で定義された内部データ端子は，論理合成時に可能であれば，その内部データ端子を含む全体の論理が圧縮される．つまり，データ内部端子自体が論理圧縮の対象となり，論理合成結果に含まれなくなることがある．なお，v は virtual のことであり，もし，論理圧縮が困難であれば，sel_v とbus_v で定義された内部データ端子は，それぞれ sel, bus で定義された端子と同じ扱いとなる．

データ端子のビット幅は， 1 ビットから 256 ビットの範囲でなければならない．ビット幅の指定が省略されると，1 ビットとみなされる．

2.4.4 レジスタ，メモリの定義

レジスタ

レジスタ定義用のタイプ名には，レジスタの初期値の扱い方によって次の 3 種類がある．

- reg: パワーオンによりレジスタの内容がセットもリセットもされないレジスタ
- reg_ws: パワーオンによりレジスタの内容がセットされるレジスタ (with set)
- reg_wr: パワーオンによりレジスタの内容がリセットされるレジスタ (with reset)

レジスタのビット幅は， 1 ビットから 256 ビットの範囲でなければならない．ビット幅の指定が省略されると， 1 ビットとみなされる．

メモリ

メモリは，機能回路の中でのみ使用できる．さらに，次の制限がある．メモリのワード数は，$2^{27} = 134,217,728$ までで，かつ，2 の巾乗でなければならない．また，ワード数を省略することはできない．なお，メモリのアドレスは，0 から始まる．

メモリのビット幅は，1 ビットから 256 ビットの範囲でなければならない．ビット幅の指定が省略されると，1 ビットとみなされる．

2.5 時間

SFL は，「単相クロックの同期回路」を対象としている．そのため，レジスタ，メモリ，記憶要素で構成されるステージの状態，タスク，セグメントの戻り状態は，クロックに同期して変化する．

SFL では，最初のクロック投入の前にパワーオンリセットが行なわれ，ステージの状態は初期状態とされ，タスクを表すレジスタはリセットされる．レジスタはその種類によりセット，リセット，または，不定 (unknown) のままとされる．メモリの内容は，不定のままとされる．

また，SFL では，システムリセットやユニットリセットなどを扱うための特別な形式はない．各種のリセットは，通常の動作として設計・記述しなければならない．これらは，すべて同期リセットとなる．

2.5.1 マシンサイクル

クロックから次のクロックまでを**マシンサイクル**と呼ぶ．厳密に考える必要はないが，クロックはマシンサイクルに含めないとする方が考えやすいだろう．クロックはマシンサイクルとマシンサイクルを区切る契機ということになる．

2.5.2 ステージの動作の記述例

制御端子の動作，ステージの動作は，マシンサイクルを時間の最小単位として記述する．マシンサイクル内の前後関係は指定しない．例えば，次の記述

```
state st1 par {
    goto st2;
    out = counter;
    counter := in;
    do();
}
```

は，「あるステージ `st1` という状態にあるマシンサイクルでは，その本体

```
    goto st2;
    out = counter;
    counter := in;
    do();
```

を実行する」ことを表し，これら 4 つの動作は，記述の順序に関係なく並列に実行される．ここで，`par` は複数の動作をまとめるキーワードで，また，`st2` は別の状態，`out` はデータ出力端子，`counter` はレジスタ名，`in` はデータ入力端子の名称をそれぞれ意味する．

“`counter := in;`” という記述により，レジスタへの書込みを意味するが，このレジスタの値が更新されるのはクロックの時点である．更新されたデータを参照できるのは，次のマシンサイクルとなる．また，レジスタの値は，次に更新されるまで保存される．

ステージの状態は，状態レジスタにより保存されるので，“`goto st2;`” という状態遷移によってステージの状態が変更されるのは，クロックの時点である．ステージの状態が変更された状態となるのは，次のマシンサイクルである．また，ステージの状態は，次に変更されるまで変わらない．

一方，“`out = counter;`” というデータ端子への値の出力では，データ端子の値はすぐに更新される．

2.5.3 複数動作の記述例

レジスタやメモリへの書込みや状態遷移，タスクのセット / リセットは，クロックの時点で行なわれること，また，データ端子への値の出力や制御端子の起動はすぐに行なわれることから，あるマシンサイクル内での複数の動作は，記述された順序によらず，同一の結果となる．

2.6 値

各構成要素がとりうる値は，次のようになっている．

データ端子 “1”， “0”， unknown, 値なし

レジスタとメモリ “1”， “0”， unknown

制御端子とタスク “1”， “0”

データ端子は，マシンサイクル毎に値が設定されるとその値を持ち，設定されなければ値なしとなる．値なしのデータ端子を参照すると unknown とみなされる．

レジスタやメモリは，値が設定されると次に設定されるまでその値を保持し続ける．一度も値が設定されていないメモリやレジスタ (パワーオンリセットにより初期化されるものを除く) の値は unknown である．

データ端子，レジスタ，メモリの値は，ビット毎に定義される．

制御端子は，マシンサイクル毎に起動されるときには，値 “1” を持ち，起動されないときには，値 “0” を持つ．

ステージのタスクは，セットされているときには値 “1” を持ち，リセットされているときには値 “0” を持つ．

2.7 ステージ

ステージは手続きを表現するための構成要素である．ハードウェアの並列性を表現するために，複数の手続きを記述できるステージと呼ばれる構成要素が導入された．

2.7.1 複数のステージの制御

ステージを複数用いることにより，複雑な制御を簡明に記述することができる．

(1) 並列制御 あるステージから他の 2 つ以上のステージへ下請け的処理を生成し，自身のステージは，その終了を待つようにして並列度を向上できる．下請け的処理の生成は，ジョブの生成により記述する．下請けステージは，処理を完了した段階でジョブを終了させる．

(2) おいてきぼり制御 処理結果をすぐには必要としない場合には，他のステージへ下請け的処理を生成して，自身のステージはその終了を待たずに後続の処理を行なうことで並列度を向上できる．

(3) パイプライン制御 処理を複数のステージで流れ作業的 (パイプライン的) に行なうことで並列度を向上できる．複数のステージ間の処理の受渡しは，ジョブの転送により記述する．

ここで，ジョブの生成とは「自身が動作状態であることとは別の新たな動作状態であることを，他のステージに生成すること」を意味する．ジョブの転送とは「他のステージへ自身が動作状態であることを転送する (渡す)」ことを意味する．ジョブの終了とは「動作状態であることを終了 (消滅) させる」ことを意味する．ジョブとは「ステージ間で生成，転送，終了される動作状態であること」を意味する．ジョブの数を増やすことが，並列度を上げることになる．

2.7.2 パイプライン制御におけるステージ，タスク，ジョブの関係

「複数ステージに渡って実施される一連の処理」のことを**ジョブ**と呼ぶ．ジョブは，マシン・サイクル毎に見れば，いずれかのステージに存在する．このとき，ジョブが存在するステージは動作状態にある．

ジョブを，あるステージから他のステージに渡すことを「ジョブの転送」と呼ぶ．ジョブの転送を行なったステージは，「タスクがリセット」されることで動作状態でなくなり，一方，ジョブの転送を受けたステージは，「タスクがセット」されることで動作状態となる．

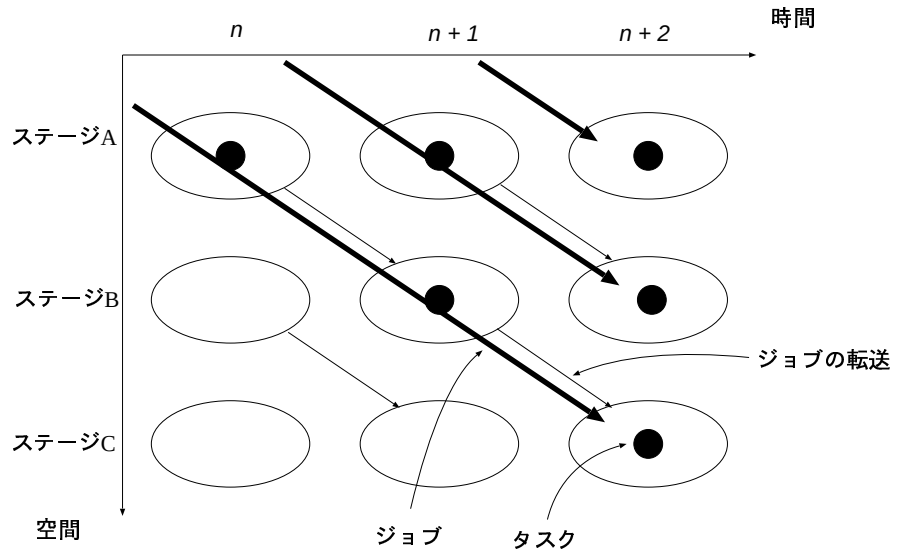

図 2.1: ステージ，タスク，ジョブの関係

2.7.3 複数ステージの記述例

ステージの記述は，大きく 2 つの部分に分けられる．1 つは，ステージの存在を宣言する部分で，もう 1 つは，ステージの内容を定義する部分である．

ステージの存在を宣言する部分では，タスクと引数の定義の双方を共に行なう．下記の例は，ステージ A，B，C の宣言の記述である．

```
stage_name A {
    task a();
}

stage_name B {
    task b();
}
stage_name C {
    task c();
}
```

上記例では，各ステージのタスクを 1 個しか宣言していないが，複数個宣言することができる．

次にステージ本体の定義の記述を示す．何もしないステージ (仕事をしたふりをするステージ) であれば，下記のように記述する．

```
stage A {
    relay B.b();
}

stage B {
    relay C.c();
}

stage C {
    finish;
}
```

以上で，**図 2.1**に示したパイプライン動作を記述できる．ステージ A，B，C は状態遷移を行なわず，単にジョブを次のステージへ転送 (たらい回し) しているだけであるが，状態遷移を行なう場合，次のように記述する．

```
stage X {
    stage_name st1, st2;
    first_stage st1;

    stage st1 par { ... }
    stage st2 par { ... }
}
```

2.8 セグメント

何度も用いる下請け的処理ではあっても，独立な制御による並列度の向上が期待できないものは，「ステージの中の閉じた枠組」である**セグメント**により記述する．セグメントは，ソフトウェアのサブルーチンと同様なものである．

セグメントは，つぎの形式で呼び出す．

```
call セグメント名 (戻り状態名);
```

2.9 制御端子

2.9.1 制御端子の必要性

制御端子は，手続きによって構造上の階層を扱うために導入された SFL のもっとも重要な概念である．これによって，静的な接続表現を使わずに複雑なシステムを記述できるようになった．静的な接続表現，すわなち，回路図を使うべきでないという考えは多くの設計経験から得られたものである．人は回路図から一旦，動作を抽出しなければ回路を理解できない．回路図による設計では，設計の内容を接続に変換して記述し，逆に接続から動作を抽出理解して設計を追加，あるいは，完成させていくという負担の大きい作業が必要である．

ところが，対象を構成要素間の接続で表すと，その構成要素はたまさらに下位の構成要素間の接続で表されるというように，接続記述は再帰的な性質を持つ．このため，自然に階層化や部品化が行なえ，また，工程によらず使用することができる．これが，上に指摘した問題があるにもかかわらず回路図による設計が広く行なわれている理由である．

したがって，記述性を高めるためには，動作 (手続き) と構造上の階層を両立させることが必要となった．このために考えられたものが制御端子である．

2.9.2 制御端子の記述例

手続き (制御) を階層の外から内へ通過させるために制御入力端子を用いる．たとえば，

```
foo.bar();
```

という記述は，「foo という構成要素に bar をしなさい」という指示を行なっていることを意味する．foo からみると，bar から制御が入ってくるので，bar は**制御入力端子**と呼ばれる．制御入力端子には，動作が対応づけられているので，外からの指示によってその動作が起動される．例えば，指示bar を実行するのに複数マシンサイクルが必要であれば，制御端子bar からさらに

```
instruct bar par {
    generate stg.tsk();
    ...
}
```

といった記述を用いステージstg へジョブを生成することになる．ここで，instruct は，制御端子に動作を対応づける形式である．また，generate stg.tsk(); は，ステージstg のタスクtsk というジョブを生成することを意味する．

制御端子やステージの起動は，“()” が名前の後につくことによって，他のデータ系の操作と区別がつけられている．

2.10 引数

制御端子やステージは，それらを起動することにより特定の動作を行なわせることができる．起動時に，動作の内容やその対象を指定できるとさらに記述性が高まる．このための形式が**引数**である．

引数には，

- 仮引数
- 実引数

がある．仮引数は，「あらかじめ制御端子毎，あるいは，タスク毎にどの端子やレジスタを上記指定データの中継点とするか」を定義しておくものである．実引数は，「指定データ」そのもので，制御端子やステージの起動時に記述する．

2.11 動作

2.11.1 動作と単位動作

1 マシン・サイクル分の動作を表す形式を**動作**と呼ぶ． SFL では，`par`，`alt`，`any`，`if`，`else` といった構文要素により，並列性や条件の階層的な関係を表現する．一方，個々の動作のことを**単位動作**と呼ぶ．

1 個の単位動作はそのまま動作となる． 2 個以上の動作は，つぎのように記述することで動作となる．

```
par {
    動作 1
    動作 2
      ⋮
    動作 n
}
```

2.11.2 条件による制御

動作を条件で制御する最も簡単な方法は，`if` を用いた制御である．

```
if ( 条件式 ) 動作
```

ここで，「条件式」のビット幅は 1 でなければならない．「条件式」の値が “1” の場合に，「動作」が実行される．条件によって制御される動作では，条件は “unknown” であってはならない．このような SFL 記述に対して処理系は正しく動作しない．

`any` を用いると，条件つきの動作をまとめて動作とすることができる．例えば，次のように記述した場合，条件 $_n$ (ただし， $1 \leq n \leq 3$) が成立したときに動作 $_n$ が実行される．条件が複数個成立した場合，成立した条件に対応する動作のすべてが実行される．動作 $_4$ は，条件 $_1$ 〜条件 $_3$ のどれも満たさなかった場合にのみ実行される．

```
any {
    条件 1: 動作 1;
    条件 2: 動作 2;
    条件 3: 動作 3;
    else: 動作 4;
}
```

`alt` を用いると，優先つきの条件をもつ動作をまとめて動作とすることができる．例えば，次のように記述した場合，条件 $_1$ が成立したときに動作 $_1$ が実行される．条件 $_1$ が成立せずに，条件 $_2$ が成立したときに動作 $_2$ が実行される．条件 $_1$，条件 $_2$ が成立せずに，条件 $_3$ が成立したときには動作 $_3$ が実行される．動作 $_4$ は，条件 $_1$ 〜条件 $_3$ のどれも満たさなかった場合にのみ実行される．

```
alt {
    条件 1: 動作 1;
    条件 2: 動作 2;
```

```
        条件3: 動作3;
        else: 動作4;
    }
```

par，any，alt は，ネスティングして (入れ子で) 使用することができる．

2.12 単位動作

単位動作は，**表 2.4**に示すように 5 種類に分類される 11 のタイプがある．

表 2.4: 単位動作の種類

分類	単位動作	キーワード
端子への値の出力	端子への値の出力	=
記憶素子への書込み	レジスタへの書込み	:=
	メモリへの書込み	:=
状態変更	状態遷移	goto
	セグメント呼出し	call
	戻り状態を指定しないセグメント呼出し	call
	セグメントからの復帰	return
ステージの起動	ジョブの生成	generate
	ジョブの転送	relay
	ジョブの終了	finish
制御端子の起動	制御端子の起動	()

2.13 式と演算子

SFL の演算子を**表 2.5**に示す．SFL では，同一優先順位の演算子は「式の右から左へ」評価される．

論理演算は，ビット毎に行なわれる．演算子 |，@，& では２つのオペランドのビット幅は等しくなければならない．

連結では，左項を MSB 側，右項を LSB 側として，左項と右項が連結される．

2.14 構成要素の参照

構成要素の参照を表す一般形は，つぎの形式をとる．

構成要素名. 指示名(実引数 , ...). データ名

この形式は，「構成要素の起動」と「結果の参照」を合体させたものであり，「ある構成要素をある指示で起動して結果を得る」という意味である．

実引数は式であるため，この形式はネスティングさせることができる．ここで，構成要素名はサブモジュール名，あるいは，ステージ名であり，指示名は制御端子名，あるいは，タスク名であり，データ名は，データ端子名，あるいは，レジスタ名である．ステージの起動は，起動したマシンサイクルでは結果が得られないため，実際に意味があるのは，次の形式となる．

サブモジュール名. 制御入力端子名(実引数 , ...). データ出力端子名

他の構成要素 (レジスタやデータ端子など) の参照も考え方は同じであるが，記述を簡単にするために，単に

構成要素名

で値を参照することもできる．

また，構成要素を起動した結果をこの形式によらずに直接，

表 2.5: SFL の演算子

優先度	演算子	機能	例		備考
			記述例	記述例の演算結果 ※	
高	<n:m>	ビット切出し	a<2:1>	01	
	<n>	ビット切出し	a<2>	0	
中	^	否定	^a	0100	
	/|	桁方向の OR	/|a	1	
	/@	桁方向の EXOR	/@a	1	
	/&	桁方向の AND	/&a	0	
	/	デコード	/a	0000 1000 0000 0000	機能回路のみで使用可能
	\	エンコード	\a	011	機能回路のみで使用可能
	#	符号拡張	8#a	11111011	
低	|	OR	a | b	1111	
	@	EXOR	a @ b	0100	
	&	AND	a & b	1011	
	||	連結	a || b	10111111	
	+	加算	a + b	11010	機能回路のみで使用可能
	>>	ビット右シフト	a >> 0x2	0010	機能回路のみで使用可能
	<<	ビット左シフト	a << 0x2	1100	機能回路のみで使用可能
	==	一致判定	a == b	0	機能回路以外では，右項は定数

(※ a= 1011，b=1111 の場合)

構成要素名. データ端子名

で参照したり，引数を使わずに直接，

構成要素名. データ端子名 = 式 ;

で構成要素の端子へ値を設定することもできる.

2.15 その他

SFL 言語には，ファイルの取込み (インクルード)，マクロ定義，コメントの記述のための**表 2.6**に示す形式がある.

表 2.6: SFL でのインクルード，マクロ定義，コメント

形式	意味
%i "ファイル名"	カレントディレクトリのファイルの内容の引用
%i "パス名"	カレントディレクトリを起点とするパスの内容の引用
%i <ファイル名>	SFL ライブラリディレクトリのファイルの内容の引用
%i <パス名>	SFL ライブラリディレクトリを起点とするパスの内容の引用
%d 名前 文字列	名前を文字列で置き換える
/* コメント */	コメント

索引