修士論文

動作レベルハードウェア記述言語 **Evelyn**

指導教官　　富田 眞治 教授

京都大学大学院工学研究科
修士課程情報工学専攻

舟本 一久

平成 10 年 2 月 13 日

動作レベルハードウェア記述言語 Evelyn

舟本 一久

内容梗概

VHDL や Verilog-HDL といった機能レベルハードウェア記述言語 (機能レベル HDL) がいよいよ一般に普及してきた．しかしこれら機能レベル HDL によっても，当初言われた程には設計の抽象度は上がらず，真のトップダウン設計を可能とするには全く不十分であった．

設計の抽象度が上がらなかったのは，HDL の処理系 – 主に論理最適化系の能力が貧弱であったこともさることながら，これらの HDL の記述能力自体の低さにも大きな原因があった．多くの HDL では，論理合成を強く意識していることもあって，実際の論理回路と 1 対 1 で対応するようにしか記述できないのである．

そのため設計者は，設計の最初期から，論理合成結果を予測しながら，デザインが物理的制約を満たすように設計記述を行なわなくてはならない．これではトップダウン設計は二の次にならざるを得ず，設計の初期コスト，変更コストが大きな負担となってしまう．

従来の機能レベル HDL の問題点は，論理合成を強く意識する余り，記述が実際の論理回路に寄りすぎていたところにある:これらの HDL は機能モジュール単位にハードウェアを記述する．この機能モジュールとは実際の論理回路と 1 対 1 に対応する単位であり，物理的制約や回路の共有の可能性まで考慮して分割された単位である．

機能レベル HDL ではハードウェアをまず機能モジュールに分割し，それぞれの機能モジュールの動作の集合としてハードウェア全体の動作を表現する．そのため動作記述が機能モジュールの構成に大きく依存する．

一方で，機能モジュール構成は物理的制約に大きく依存するものである．そのため論理合成/最適化の結果が物理的制約を満たせなかった場合には機能モジュール構成を変更し，動作記述からやり直さなければならないのである．

本論文で提案する動作レベル HDL「**Evelyn**」ならば従来の HDL の問題点を解決し，真のトップダウン設計を実現することができる．

Evelyn では，ハードウェアを高い抽象レベルで記述する．もちろん抽象度を上げることにより記述を簡潔にすることも狙っているが，それ以上に物理的制約

に強く依存するような設計情報をできる限り動作記述から排除することを狙っている．

そのような設計情報は，動作記述に後から付加するという形で与える．詳細な設計情報を少しずつ付加することで設計記述を進めていく．

後から付加する設計情報のうち動作に影響を与えないようなもの – 例えばメモリの種類などの情報を特に**合成ディレクティブ**と呼ぶ．合成ディレクティブは単に機能合成時のオプションとして与えるのではなく，言語仕様の一部として規定する．合成ディレクティブを言語の仕様として組み込むことで，記述の抽象度の高さと合成能力のトレードオフを高いレベルでバランスさせる．

抽象度の高い動作記述をベースに，詳細な設計を付加していくというアプローチにより設計変更のコストを抑える．すなわちこうすることで論理合成/最適化系からのフィードバックにより設計変更を迫られた場合にも，ベースとなる動作記述に手を加えずに，後から付加した情報のマイナーな変更だけで対処できるようにするのである．

このアプローチにより，抽象的な設計記述を徐々に詳細化していくというトップダウン設計を実現することができる．

Evelyn では，ハードウェアを動作単位 – すなわち動作主体とその動作という形で記述する．

1 つ 1 つの動作は，複数サイクルに渡る一連の動作として記述する．記述中の時間関係は，変数の依存関係によって半順序的に定義する.「この処理が始まるまでに…」,「…から…までの間に…」など非常に柔軟に時間関係を表現できる．物理的制約によってのみ決定すればよいような時間関係を無理に決定する必要がない.

時間方向に記述された一連の処理と，そこに暗示的に示された時間関係からパイプラインを自動合成する．ステージ境界は変数の型を変えるだけで簡単に変更することができる．

このように柔軟，かつ強力な記述能力を持つ Evelyn はハードウェア開発を大幅に効率化できるものと考えられる．またこの柔軟さのため，Evelyn を方式設計の後半や，設計前の性能の予備評価にも広く利用できるものと考えられる．

Evelyn : A Behavior-Level Hardware Description Language

Kazuhisa Funamoto

Abstract

Function-level hardware description languages (function-level HDLs) such as VHDL or Verilog-HDL have come into wide use recently. But by using them, designers can't describe designs more abstract than they were expected. In short, these function-level HDLs can't realize the true top-down design.

Certainly, one of the reasons of that problem is the incapacity of their processor – specially their optimizer. But the most important reason is the incapacity of these HDL description. With most of these HDLs, designers must describe designs in one-to-one correspond to physical logical circuits.

So, from the very early stage, designers must consider the physical constraints and describe their designs carefully. In this situation, the top-down design is of secondary importance.And the cost of both the early stage of design and changing the design are too heavy.

The problem of conventional HDLs is that, being too conscious of logic synthesis, the unit of descriptions is too near to physical logical circuits.

When designers describe their design with these function-level HDLs, at first, they must divide their design into function modules.And then they describe the behavior of each module. A set of each behavior constructs the behavior of the whole. Therefore the behavior description of the whole is strongly depend on the division of function modules.

On the other hand, the division of the function modules is strongly depend on physical constraints. Therefore if the result of logic synthesis doesn't satisfy the physical constraint, designers must re-divide their design, and re-describe the behaviors.

New behavior-level HDL **Evelyn** that I propose in this paper can solve such problems and make the top-down designing come true.

With Evelyn, designs are described very abstractly. Of course, one reason of this approach is to make descriptions brief. But more important reason is to remove detail design information which strongly depend on the physical

constraint from the description.

Detail information is given gradually by adding to the abstract description.

Some of detail information – for example, a type of memory – doesn't effect designs' behavior except their performance. Such information is especially called **synthesizer directives**. Evelyn doesn't defines synthesizer directives as synthesizer options but defines it as the part of the language specification. This approach can well balance the trade-off between the abstractness of its description and the capability of the synthesizing.

Evelyn's approach, that designer gradually add detail design information to the base abstract description, can realize the top-down design method.

With Evelyn, the designs are described in blocks of behavior modules – that is a behavior module and its behavior.

Each behavior is described as a series of action over steps. The orders of operations is defined partially by data dependency. So designers can express abstract time conception like 'before that operation' or 'between these operations'. It is not necessary to decide the order between operations which are constrained only by the physical constraint.

Evelyn processor can synthesize pipeline circuits automatically. A series of actions in the time direction and the implicit operation ordering are used in this process. Stage boundaries can be easily changed by trading variable type.

With Evelyn it is possible to design a hardware very efficiently. And because of this flexibility, Evelyn can be use in the latter half stage of the system design and in the evaluation of the performance before the development.

動作レベルハードウェア記述言語 Evelyn

目次

1　はじめに

VHDL や Verilog-HDL といったハードウェア記述言語 (HDL) と，それらからの (半) 自動的な論理合成/論理最適化によるハードウェア設計手法が一般に普及しつつある．これらの HDL による設計手法によって，従来の回路図入力による設計手法に対して，設計の効率は確かに飛躍的に向上した．

しかし，これらの HDL によっても当初言われた程に設計の抽象度が上がるということはなく，本当の意味でのトップダウン設計を可能とするには全く不十分であった．言語入力には回路図入力に比べて入力や修正が容易になるという利点があるが，これらの HDL の導入によって得られた効率化はむしろこのことによる部分が大きい．

設計の抽象度が上がらなかったのは，HDL の処理系 – 主に論理最適化系の能力が貧弱であったこともさることながら，これらの HDL の記述能力自体の低さにも大きな原因があった．論理合成を強く意識していることもあり，多くの HDL では実際の論理回路と 1 対 1 で対応するようにしか記述できないのである．

そのため，設計の最初期から，回路が物理的制約を満たすように，論理合成される回路を予測しながら設計記述を行なわなくてはならない．これではトップダウン設計は二の次にならざるを得ず，設計の初期コスト，修正コストが大きな負担となってしまう．

本論文で提案する HDL「**Evelyn**」ならばこれらの問題点を解決し，トップダウン設計を実現できる．

これまでの HDL では論理合成を強く意識するあまり機能単位に記述し，それをベースに動作を記述するというアプローチを採っている．それに対し Evelyn は，ハードウェアを動作単位に記述し，そこに機能レベルの設計情報を付加するというアプローチを採る．こうすれば，設計の初期には動作設計のみに専心でき，また機能レベル設計工程での変更が動作記述に影響を及ぼすことはない．

Evelyn の動作記述は非常に強力である．ハードウェアの動作を時間方向に素直な形で記述することができ，そこからパイプラインを自動合成できる．しかもパイプライン・ステージ構成をほとんどコストをかけずに変更することができる．そのため Evelyn は，方式設計の終盤から利用できる可能性を持つ．

本稿ではこのような特徴を持つ動作レベル HDL Evelyn を提案する．以下，

まず2章で従来のHDLでは何故トップダウン設計が困難なのか，従来のHDLの問題点を明確にし，3章でEvelynがトップダウン設計を可能とするための方針を述べる．そして5章でEvelynによる動作記述の容易さを述べ，さらに6章でその記述の機能合成手法，7章で動作シミュレーション手法について説明する．

2 背景

Evelynについて述べる前に，なぜ従来のHDLがトップダウン設計に結び付かなかったのかったのか，本章でその問題点を明らかにする．

以下では次のように議論を進める:まず2.1節で既存のHDLを用いたハードウェア設計ではどのような事柄が問題となっているか述べる．そしてそのような問題が何故生じるのか考察した後で，では理想のハードウェア設計環境とはどのようなものか述べる．

2.1 従来の機能レベルHDLの問題点

VHDLやVerilog-HDLなどの従来のHDLを用いても当初言われた程にはトップダウン設計にはつながらなかった．結局従来の回路図入力と同じように，設計の初期から物理的制約を勘案して綿密に設計記述することが要求されるのである．

従来のHDLではトップダウン設計の必要性が考慮されていないかというと，そうではない．実際，VHDLでは次のような機構を用意してトップダウン設計を支援している[1]:VHDLの各モジュールは複数の *architecture body* を持つことができる．設計者は，まず抽象度の高い *architecture body* を記述して設計検証を行ない，それを論理合成可能な抽象度の低いものに順次置き換えていく．

ところが現実にはそのような機構は機能せず，次のような設計コストが問題となっている．

問題-1. 初期工程のコスト

論理最適化系が1度に最適化できる回路の規模には制限がある．そのため1度に最適化できる回路の規模を勘案し，設計上クリティカルな部分回路を1つの機能モジュールにまとめあげることを重視して，設計の最初の段階から綿密に機能モジュールの分割を決定する必要がある．

その結果，ハードウェアの動作を自然に記述することは二の次にならざるを得ず，しばしば

- 意味的に関連性の高い記述を複数の機能モジュールに分割したり，逆に，
- 意味的に関連性の低い記述を 1 つの機能モジュールにまとめたり

する必要が生じる．この作業は人手で行わざるを得ず，アルゴリズム・レベルでバグが混入する可能性が高い．

問題-2．設計変更のコスト

初期工程で綿密に機能モジュール分割を行なったとしても，論理最適化の段階で制約を満たさないことが判明することは完全には避け難い．その場合には，機能モジュール間で記述の移動をやはり人手で行う必要がある．このような作業は，既に作り込まれた記述に対するものであるだけに，やはりバグが混入し易い．

これは，従来の HDL におけるハードウェアの記述方法に根本的な問題がある．これらの HDL は論理合成を強く意識しているということもあり，記述の単位が物理的な回路に寄りすぎているのである．

現在広く使われている VHDL や Verilog-HDL などの HDL は，**機能モジュール**を単位にハードウェアを記述/論理合成するので，特に**機能レベル HDL** と呼ばれている．ここで機能モジュールとは物理的な回路と 1 対 1 で対応するような単位であり，タイミング制約や回路の共有可能性まで考慮して分割されたものである．

機能レベル HDL におけるハードウェアの記述方法 – すなわち機能モジュール毎にその動作を記述して，個々の動作の集合としてハードウェア全体の動作を実現するという方法では，動作記述が機能モジュールに強く依存する．つまり機能モジュールの分割方法を変更すると，動作記述を 1 からやり直さなければならない．

一方で機能モジュールへの分割の仕方は，ゲート数やゲート遅延といった物理的制約に強く依存するものである．従って，論理合成/最適化の結果が物理的制約を満たさないことが分かったら，機能モジュールを変更しなければならない．

確かに経験を積んだ設計者ならば物理的制約を勘案し，合成される回路を予測しながら設計を進めることも全く不可能な訳ではない．しかし論理合成系/最適化系の癖まで考慮に入れて合成結果を完璧に予測しきることは非常に困難である．従って，論理合成/最適化系からのフィードバックによる修正を完全に避けることは難しい．

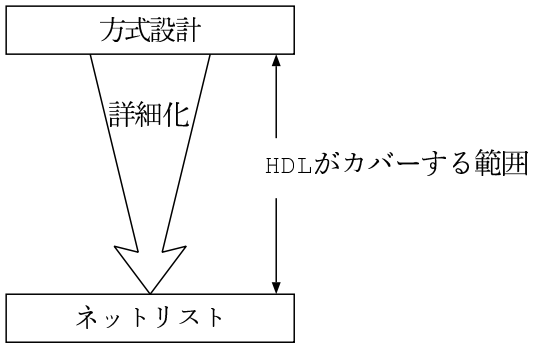

図 1: HDL がカバーする設計工程

つまり先に挙げた 2 つの問題の原因は，機能モジュールという，実際のハードウェアに近いものをベースにしていることにあるのである．

最初に述べた VHDL のトップダウン設計支援が機能しなかったのもまさにそこに原因がある．すなわち，

- 複数の *architecture body* を持つことができるといっても，それらは機能モジュール単位で記述しなくてはならない．先に述べたように，機能モジュールは，物理的制約を満たすために意味的に関連性の低い記述をまとめたような，論理合成本位の分割単位であるから，抽象度の高い *architecture body* を記述するといっても無理がある．
- そのようにして記述した抽象度の高い *architecture body* は機能モジュール境界に強く依存するものである．従って，下流工程からのフィードバックによって機能モジュール境界を変更すると，それらは全く無駄になってしまう．

つまり，機能レベル HDL の採ったアプローチ – すなわち後々変更される可能性が高くしかも変更のコストが非常に大きい機能モジュールを単位にハードウェアを記述するというアプローチは，トップダウン設計の観点からみて，全く間違いであったと言わざるを得ない．

2.2 理想的なハードウェア設計システム

前節で述べた機能レベル HDL の問題点を踏まえて，真にトップダウン設計を実現する理想的なハードウェア設計システムとはどのような条件を満たすべきか考察する．

トップダウンな設計工程とは，より抽象度の高い設計を徐々に詳細化してい

くという過程である．ハードウェア設計における詳細化の工程のうち，HDL がカバーする工程を図 1 に示す．

図 1 の方式設計後からネットリストを得るまでの工程を支援するのが HDL とその処理系の役目である．すなわち自然言語などで記述された仕様に対して，処理間の並列性や資源共有の可能性を考慮して，データ構造やステージ構成を詳細化し HDL のプログラムの形で記述する．完成した HDL 記述を論理合成/最適化系を利用してネットリストに変換する訳である．

トップダウン設計を実現するためには，このような詳細化の工程を言語レベルでサポートすることが望ましい．すなわち方式設計に近い抽象レベルから実際の論理回路と 1 対 1 に対応する程の抽象レベルまで記述できることが望ましい．

しかしそれだけでは不十分である．高い抽象レベルの記述をサポートすることに加えて，設計変更が容易であることが求められる．

何故ならば，ハードウェア設計では下流工程からのフィードバックにより設計変更を迫られることが多いからである．これはハードウェア設計は，ソフトウェア設計に比べて物理的制約が非常に強いことに起因する．いくら優れたアーキテクチャを設計しても，それがチップに収まり目的の周波数で動作しなければ全く意味はないのである．

アルゴリズムレベルの大規模な修正を行なう場合には全て設計し直すことになるのはやむを得ないにしても，例えばパイプライン・ステージの境界を微調整する程度の設計変更で動作レベルから設計をやり直すということはできる限り避けたい．

そこでハードウェアの設計環境に特に求められるのは，下流工程からのフィードバック・ループをできるだけ短かく抑え，修正のコストを最小限にする仕組みである．

- 上流工程で混入したバグをなるべく早い段階で取り除く仕組み
- 下流工程での設計変更が上流工程に影響を与えないような仕組み

設計システムがこれらの仕組みを備えればフィードバック・ループを短かく抑えることができ，設計変更のコストを最小限に留めることができる．

ところで既にいくつかの動作レベル HDL とその処理系が機能レベル HDL の問題点を解決すべく研究されている．しかしそれらはフィードバック・ループの問題を根本的に解決するものでない．

これらの動作レベル HDL の研究は，以下の 2 つのアプローチを採るものが

多い.

1 つは機能レベル HDL の欠点をアドホックに補おうとするアプローチ – すなわち機能レベル HDL でそのまま記述すると記述量が多くなってしまうような特定の回路を簡潔に記述できるようするというものである.

例えば, Bach[2] では, 専用のマクロを提供することでハンドシェイク回路を簡潔に記述できるようにしている. また Cyber[3] では, ステート・マシンを簡潔に記述でき, また合成時のオプションにより時分割多重の処理を自動合成できるようにしている.

もう 1 つのアプローチは, 組み込み用プロセッサなどに記述の対象を限定して, ハードウェアのテンプレートのようなものを与えるものである.

例えば AIDL[4] では, 記述の対象をパイプラインプロセッサに絞っている. パイプライン・プロセッサに限定することで, パイプライン・ステージを簡潔に記述できるようにしている.

確かにこれらのアプローチによって, 機能レベル HDL に対して記述の抽象レベルが上がることで記述量が少なくなり, 結果的に設計コストが削減される.

しかし逆に言えば, これらの HDL が機能レベル HDL に勝るのはこの点に関してのみである. 依然として機能モジュールをベースにハードウェアを記述している. 下流工程からのフィードバックが大規模な設計変更に結び付きやすいという問題点は何ら解決されていないのである.

そこで, 下位工程からのフィードバック・ループを短かく抑え, 真のトップダウン設計を実現することを目指し, 動作レベル HDL「**Evelyn**」を開発する.

3 Evelyn 設計の方針

2章で述べた機能レベル HDL の問題点を踏まえ, 本章では Evelyn 設計にあたっての方針を説明する.

以下本節では, まずトップダウン設計を実現するためのアプローチを述べ, それを実現するためのアイデアを説明する.

3.1 トップダウン設計の実現

Evelyn の最終的な目標は, **真にトップダウンなハードウェア設計環境の実現**である. そのためには, 下流工程からのフィードバック・ループを短かくことが是非とも必要である.

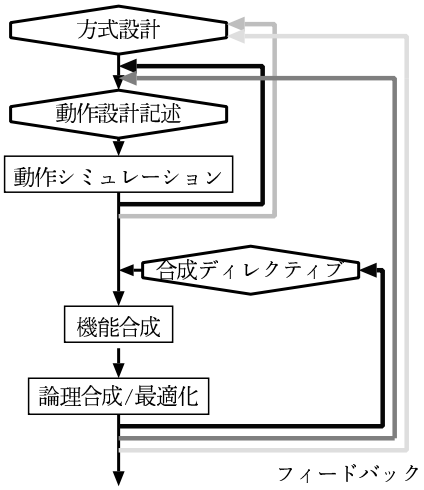

図 2: Evelyn を用いたハードウェア設計フロー

2章でみてきたように，機能レベル HDL で抽象的な記述をサポートしていたにもかかわらずトップダウン設計に至らなかったのは，下流工程からのフィードバック・ループが長く，設計変更のコストが非常に大きかったためである．

Evelyn では，フィードバック・ループを短かくするために次のアプローチを採る:

- モジュール境界を抽象化して記述.
- そこに設計情報を付加していくことで詳細化.

これは以下の理由による: 前節で述べた下流工程からの長いフィードバックの問題を解決するためには，記述におけるモジュール境界と実際に論理合成/最適化系に入力されるモジュール境界とを分離する必要がある．そのためには，記述においてモジュール境界が具体的過ぎてはならない．すなわち，**モジュール境界の抽象化** が必要である．

しかし単に記述の抽象度を上げただけでは，そこからの機能合成が困難となる．従って Evelyn では，抽象的なモジュール境界の記述から論理合成/最適化に適した機能モジュールを合成するに際して，抽象的な設計記述に**合成ディレクティブ**を付加するようにする．詳細な設計情報を合成ディレクティブとして与えることで，記述の抽象度の高さと，合成能力のトレードオフを高いレベルでバランスさせる．

このようなアプローチを採ることで，ハードウェアの設計フローは図 2のようになる．

設計の初期段階では，ハードウェアの動作のみを記述する．このとき機能モジュールのように実際の論理回路に近いレベルのモジュールではなく，動作モジュールというより抽象的な塊を単位に記述する．ハードウェアを動作単位で記述することで，動作設計記述の中に機能レベルの設計情報が入り込む必要をなくす．

動作記述に対して動作シミュレーションをし，動作レベルのバグを取り除く．動作情報のみの簡潔な記述の検証/デバッグなので効率良く行なえる．

そして，抽象的な動作記述に後から付加するという形でより詳細な機能レベルの設計情報を記述する．

詳細化された設計記述を機能合成，そして論理合成/最適化する．合成/最適化結果が物理的制約を満たさない場合には上流工程に戻って設計変更することになるが，多少の変更ならば合成ディレクティブのみを変更すればよく，動作記述にまで手を加える必要はない．

このようにフィードバック・ループを最小限に抑えることでトップダウン設計を実現する．

以下ではこのアプローチの鍵となるモジュール境界の抽象化と合成ディレクティブについてより詳しく説明する．

3.2 モジュール境界の抽象化

Evelyn の大きな特徴の 1 つは，ハードウェアをより高い抽象レベルで記述できるという点である．具体的には Evelyn は，以下の抽象化をサポートする:

データの抽象化 列挙型，多次元の配列型，構造体など，抽象度の高いデータ型をサポートする．それらは (半) 自動的にビット列に合成される．

モジュール境界の抽象化

- **パイプラインの抽象化** 複数のステージに渡る一連の動作を自然に記述し，その記述からパイプラインを (半) 自動的に合成する．
- **その他のモジュールの抽象化** 動作単位にハードウェアを記述する．機能合成時に機能モジュールに改めてマッピングする．

基本的なポリシーは，動作に直接影響を与えない事項を記述する必要をなくす，ということである．

例えば，抽象データのビット列へのマッピング方法，論理の正/負の別などは，回路の面積には影響を与えるが，それによって動作に影響を及ぼすものではな

い．このようなものは動作記述から切り離し，合成ディレクティブとして与えれば設計変更は容易となる．

またパイプラインも抽象化する．パイプラインのステージ構成は確かに動作に大きく影響を与えるものであるが，一方で物理的制約にも強く依存するものである．パイプラインを，一連の動作と，ステージ構成を指示する設計情報で表現するようにすれば，物理的制約により設計変更を迫られた場合にも動作を変更せずに対処できる．

さらにその他のモジュール境界も抽象化する．動作モジュールを単位にハードウェアを記述し，機能合成時に機能モジュールにマッピングする．マッピングの方法は合成ディレクティブとして与える．こうすれば，機能モジュール境界の変更が動作に影響を及ぼさない．

3.3 合成ディレクティブ

例えば，一口に記憶回路といっても RAM，フリップフロップ，ラッチなど様々な実現方法が考えられる．機能合成までには具体的にどのように実現するか，設計情報を与える必要がある．

ところで，このような設計情報はハードウェアに対して基本的には性能以外の点で影響を及ぼさない．そこで Evelyn では，これらの指定を動作記述中に記述するのではなく，機能合成時の合成ディレクティブとして与えるというアプローチを採る．このアプローチにより，物理的制約を満たすために設計変更が必要になった場合でも，動作記述に全く影響を与えることなく設計変更を行なうことができる．

この合成ディレクティブは，単に合成時のオプションとして与えるのではなく，言語仕様の一部として規定する．言語仕様として規定し，そのセマンティクスを明確にすることで，記述の意味が処理系に依存しないようにする．こうすれば記述のポータビリティが高まり，設計者が処理系の癖に悩まされることがなくなる．

3.4 パイプラインの抽象化

以上，Evelyn を設計するにあたっての方針について述べた．ここで読者に Evelyn のイメージを摑んでもらうために，パイプラインを抽象化して記述する方法を紹介する．

$$x_1 <= f_0(x_0); \quad (1)$$
$$x_2 <= f_1(x_0); \quad (2)$$
$$x_3 <= f_2(x_1); \quad (3)$$
$$x_4 <= f_3(x_3, x_2); \quad (4)$$
$$x_5 <= f_4(x_3); \quad (5)$$
$$x_6 <= f_5(x_4); \quad (6)$$

図 3: 記述 1

Evelyn におけるパイプラインの記述方法は，簡単にいうと

1. 一連の処理を時間方向に記述して，
2. ステージ境界の位置を指定し，
3. パイプライン・ラッチの実現方法を指定する，

というものである．

パイプラインの記述法を通して，先に挙げた 2 つの方針 – モジュール境界の抽象化と合成ディレクティブによってハードウェア設計工程がどう変わるのか，ということを述べる．

3.4.1 時間方向の動作記述

Evelyn では，動作を時間方向に記述する．この動作記述中の各処理間の時間関係は，データ依存関係によって半順序的に規定される．

時間関係が半順序的なので「この処理が始まる前に…」や「この処理とこの処理の間に…」など時間関係に関して柔軟な記述が可能となる．また多くの動作レベル HDL では動作記述を行なう時点でステージ構成まで決定しなければならないが，Evelyn では純粋に動作だけを記述できる．

図 3は，Evelyn の動作記述の抜粋である．ここには図 4のデータフロー・グラフに示すような依存関係がある．そのためこの記述からは (1) と (3)，(2) と (4) などの間の順序関係のみが規定され，(2) と (3) などの間には順序関係は規定されない．

このように半順序的な時間関係のみを規定することで，アルゴリズム的な制約がなく，物理的な制約のみで順序を決定すればよいような処理間の順序を動作設計時に無理に決定する必要がなくなる．

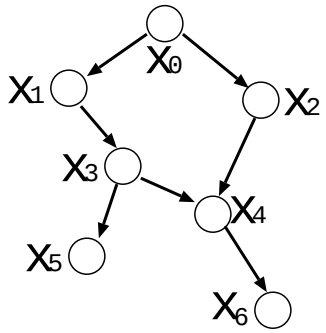

図 4: 記述 1 のデータフロー・グラフ

$$x_1 <= f_0(x_0); \quad (1)$$
$$\mathsf{x}_2 << f_1(x_0); \quad (2)$$
$$\mathsf{x}_3 << f_2(x_1); \quad (3)$$
$$x_4 <= f_3(\mathsf{x}_3, \mathsf{x}_2); \quad (4)$$
$$x_5 <= f_4(\mathsf{x}_3); \quad (5)$$
$$\mathsf{x}_6 << f_5(x_4); \quad (6)$$

図 5: ステージ境界を singal で記述

3.4.2 パイプラインの自動合成

図 3のような形で記述された動作は，実際のハードウェアでは 1 つまたは複数のステージに分割して実行されることになる．Evelyn ではこのステージの境界を変数毎に指定する:一連の動作の中でステージ境界としたい箇所，すなわちパイプライン・ラッチを挿入したい箇所を特殊な変数を用いて記述する．

Evelyn には **signal**(シグナル) と **singal**(シンガル) という 2 種類のデータ・オブジェクトがある．一連の動作の中で，パイプライン・ラッチを挿入してステージ境界としたい箇所は singal で記述し，それ以外の箇所は signal によって記述する．

パイプラインの合成は以下のようにする:動作の先頭から順に現われる singal を結ぶことで，データフロー・グラフの中に “等高線” を規定する．そして，この “等高線” をステージ境界とするパイプラインを構成する．このとき singal の前後の基本的な同期[1)]を実現するためのハンドシェイク回路を自動合成する．

[1)] 有効ビット，ライトイネーブルなどによる簡単な同期.

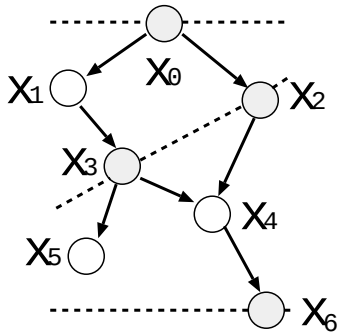

図 6: データフロー・グラフに “等高線” を定義

例えば，図 3の動作において (x_2, x_3) の位置でステージを区切りたい場合，図 5のようにそれらを singal を用いて記述する．ただし図中の<<は singal への代入を表す.

するとデータフロー・グラフに図 6の点線で示したような “等高線” を引くことができる．この “等高線” をステージ境界とするようなパイプラインを合成する．このとき x_2, x_3の前後に同期のための回路を自動合成する.

このような形のパイプライン記述ならば，論理合成系などからフィードバックがあった場合などの設計変更も容易である．例えば，タイミング制約などのため，(x_2, x_3) としていたステージ境界を (x_1, x_3) に変更したいとする．Evelyn ではこのような変更は，x_2を singal から signal に変更して，代わりに x_1を singal にするだけでよい．変更のためのコストはほとんど 0 といってよいだろう.

3.4.3 機能レベル HDL と比較して

さて，図 5のようなパイプラインを機能レベル HDL で記述するとどのようになるのであろうか？図 7に VHDL による記述例を示す．簡単のため図には上流ステージのみを記述した.

図 7は VHDL による典型的なステート・マシンの記述例で，図 8のような回路を実現している．すなわちプロセス control が外部からのリクエスト信号などを受けてステージの制御を行なう．プロセス control に制御されてプロセス main が x_0から x_2, x_3を生成する.

この例から機能レベル HDL の記述の抽象度の低さ，記述量の多さが実感できる．一般に機能レベル HDL ではステート・マシンをこのように状態遷移グラフとして記述しなければならない.

さらに設計変更のコストも非常に大きいことが分かる．先程と同様に図 6に

```
control : process( req_i, ack_i, state )
begin
    if state = FULL then
        ack_o <= req_i and ack_i;
        if ( ack = '1' ) and not( req_i ) then
            state_nxt <= EMPTY;
        else
            state_nxt <= FULL;
        end if;
    else
        ack_o <= req_i;
        if req_i = '1' then
            state_nxt <= FULL;
        else
            state_nxt <= EMPTY;
        end if;
    end if;
end process control;

main : process( rst, clk )
    variable x1 : std_logic;
begin
    if rst = '1' then
        state <= EMPTY;
    elsif clk'event and clk = '1' and clk'last_value = '0' then
        state <= state_nxt;
        if ack_o = '1' then
            x2 <= not( x0 );
            x1 := x0;
            x3 <= x1;
        end if;
    end if;
end process main;
```

図 7: VHDL による記述

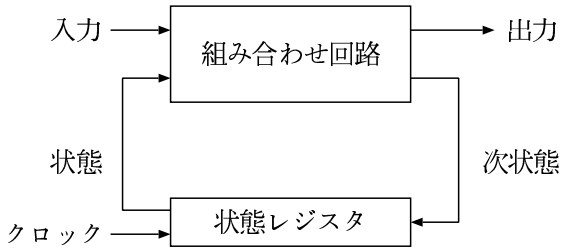

図 8: VHDL 記述が実現するステート・マシン

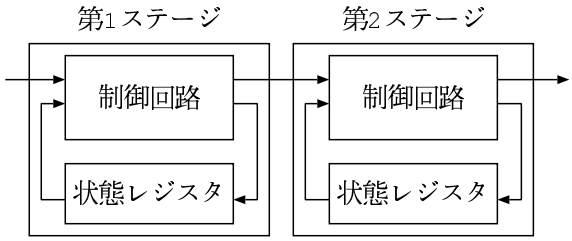

図 9: 機能モジュール構成

おいて (x_2, x_3) としていたステージ境界を (x_1, x_2) に変更する場合を考える.

図 7の記述に対してそのような変更をする場合，プロセス main と，下流ステージの同様のプロセスとの間で処理を cut & paste によって移動しなければならない．これはバグの混入する危険性の高い作業である．実際，より複雑な記述に対してこのような変更を行なう場合，結局 1 から書き直すことになったという場合が多々ある.

設計変更が困難なのは，VHDL では機能モジュール境界を先に決定して，それをベースに動作を記述するためである:この例では動作を書く前にステージ構成，ステージの内部構成を設計して，機能モジュール構成を例えば図 9のようにすると決定しなければならない.

ところがこの機能モジュール構成は物理的制約に強く依存するため，論理合成/最適化系からのフィードバックにより再構成しなければならない確率が極めて高い．機能モジュールの構成が変わると，それをベースに記述していた動作も変更せざるを得ない.

それに対し Evelyn は，動作をベースに記述し，その上に singal と signal で機能モジュール構成に関する設計情報を付加するようになっている．このため，論理合成/最適化系からのフィードバックにより機能モジュール構成を変更する場合には，signal と singal の位置のみ変更すればよく，動作記述まで変更する

必要がない.

なお，VHDL に比べて Evelyn の記述量少ない原因の 1 つとして，singal が高機能であるということが挙げられる．VHDL では各ステージに状態を持たせてステージ間の同期を実現しているが，singal はこの全てを暗示的に実現する．

このような高機能なものをプリミティブとして実装すると限られた種類の回路しか記述/合成できないと考える向きもあるかもしれない．しかし，singal の合成方法は後で合成ディレクティブにより細かく指定できる．合成ディレクティブの詳細については 6章で述べる．

4 Evelyn の概念

本章では，Evelyn における抽象的なハードウェア動作記述について，その基本となる概念について述べる．

4.1 モジュール境界の抽象化

4.1.1 モジュールとファンクション

Evelyn では動作単位にハードウェアを記述する．すなわち，動作の主体を**モジュール**として記述し，モジュールが行なう動作を**ファンクション**として記述する．

ここでいうモジュールとは動作モジュールである．すなわち機能モジュールとは関係ない，あくまで動作の主体である．機能合成時に動作モジュールを自在に分割/統合して機能モジュールへマッピングすることができる．動作モジュールは，動作を記述しやすいように構成すればよい．

ファンクションには 1 つ 1 つの動作を時間方向に記述する．ハードウェアを動作単位に記述できるため，複数サイクルに渡るような動作も非常に素直な形で記述することができる．

Evelyn では各モジュールが互いに相手のファンクションを起動し合うことでハードウェアの動作をモデル化する．つまり，Evelyn では通信をファンクション呼び出しに抽象化する．

4.1.2 ファンクション

前述したように，Evelyn では一連の動作をファンクションとして記述する．モジュールが互いにファンクションを呼び出し合うことでハードウェアの動作をモデル化する．

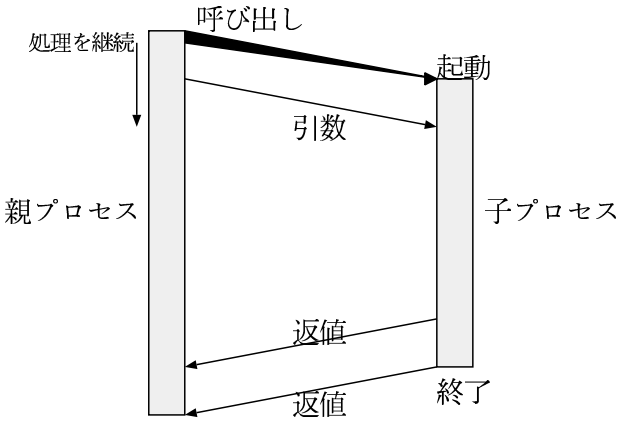

図 10: プロセス

ファンクション呼び出しによって行なわれる 1 回のファンクションの実行を**プロセス**と呼ぶ.

Evelyn のファンクションが VHDL における関数などと異なる点は，Evelyn のファンクションには，多サイクルを要する処理も記述できるという点である. このことはプロセスが複数ステップをかけて実行されるということ，そしてシステム内には複数の実行中のプロセスが存在し得る，ということを意味する.

ファンクションが呼び出されると，呼び出し側プロセスと呼ばれたプロセスは並列に実行される．すなわち，ファンクション呼び出しの後，呼び出し側プロセスはファンクションの完了を待たずに後続命令の実行を続け，返値が必要になった時点で実行をブロックする．また，呼ばれたプロセスは，返値が用意できた時点で呼び出しプロセスに値を返す．なお，返り値が複数ある場合には，それらが別々に返されることがある (図 10参照).

複数サイクルに及ぶプロセスは，singal への書き込み位置でいくつかの**ステージ**に分割される．ステージに分割されたプロセスはパイプラインとして動作する．プロセスの実行モデルの詳細は，4.2節で述べる.

4.1.3 制約ファンクション

ファンクション呼び出しでハードウェアをモデリングする場合，同一モジュールに対してファンクションが同時に呼び出された場合の処理が問題となる．すなわち，一般のソフトウェアのプログラムならばこのような場合はそれらが排他的に実行されればよいことが普通であるが，ハードウェアの場合にはそれらをまとめて統一的に処理する必要がある.

Evelyn ではそのような場合の処理を**制約ファンクション**と呼ぶ特別なファンクションを用いて記述する．制約ファンクションが同時に呼ばれると制約ファンクションが 1 回実行されて，同時呼び出しを統一的に処理する．

4.1.4　ファンクション呼び出しのセマンティクス

あるモジュールに対して同時に複数のファンクションが呼び出された場合のセマンティクスを以下にまとめる．

あるモジュールに対して，

- 同じファンクションが同時に実行されたら，そのうちのどれか 1 つが非決定的に選択されファンクションが実行される．
- 異なるファンクションが同時に呼び出されたら，
  - もしモジュールが制約モジュールならばそれらは統一的に処理される．
  - もしモジュールが制約モジュールでないならばそれらは同時に処理される．

同一のファンクションを同時に処理したい，あるいは統一的に処理したい場合には，**ファンクションの配列**を用いる．ファンクションを配列として定義すると，配列サイズだけ同時，あるいは統一的に処理できるようになる．

4.1.5　イニシャル・ファンクション

最後にイニシャル・ファンクションについて説明する．イニシャル・ファンクションとは，最初に実行されるファンクションで系を始動させる役割を持つ．

モジュールはイニシャル・ファンクションを 1 つ持つことができる．系を動作させると，まず最上位モジュールのイニシャル・ファンクションが実行される．

モジュールがあるモジュールの内部モジュールとしてインスタンス化されている場合，内部モジュールのイニシャル・ファンクションは意味を持たない．

以上，モジュールとファンクションについて述べた．Evelyn はハイレベルな視点ではモジュールが互いにファンクションを呼び出しあうことで動作する．

4.2　ファンクションの動作

本節では，Evelyn のよりローレベルの動作 – ファンクション内部の動作モデルについて説明する．

4.2.1　singal と signal

3章でも述べたように，Evelyn には singal と signal の 2 種類のデータオブジェクトがある．

```
()modA::funcB(){
        ⋮
   data << xxx;
        ↑
   (data=xxx) のライフタイム
        ↓
   data << yyy;
}
```

図 11: データのライフタイム

どちらも記憶を保持するためのものであるが，両者には大きな違いがある．それは，singal は同期機構を備え持つという点である．すなわち signal に対しては常に自由に読み書きでき，上書きされるまでいつまでも値を保持しているのに対し，singal に対しては次のような制約がある．

- 同一サイクルで古い値を参照しながら新しい値を書き込むことができる．
- 古い値がまだ参照される可能性がある場合には，新しい値を書き込むことができない．
- 前のサイクルで有効な値を書き込まれなかった singal の値は参照することができない．

singal への書き込み/参照ができない場合には，実行をブロックし，次のサイクルにもう 1 度実行する．

signal に対しては (1) 値の代入 (代入演算子は<=)(2) 値の参照の 2 つのアクセスが可能である．一方，singal に対しては (1) 代入 (代入演算子は<<)(2) 読み出しに加わえて，(3) 有効な値を持っているかのチェック (演算子は?) の 3 つのアクセスが可能である．

さて，singal について，値が参照される可能性を判定しなければならない．これは処理系が自動的に行なう．

“プログラム中で参照される箇所” は静的に判定できる．すなわち，その singal がファンクションローカルな変数である場合，そのファンクション内でその singal への代入から次のその singal への代入までがそのデータのライフタイムである (図 11参照)．従って，その間にあるその singal へのアクセスが古い値を参照する．

スコープがモジュール内であるような singal の場合は次のようにして決定

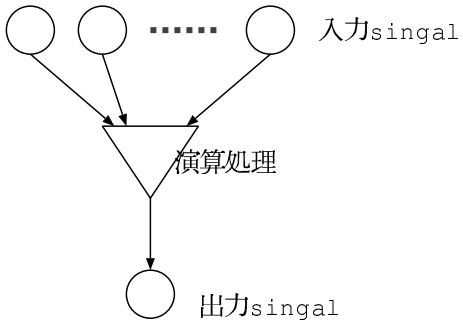

図 12: 実行の単位

する:

1. 最上位モジュールのイニシャル・ファンクションから全てのファンクションを展開して依存グラフを生成する.
2. 依存グラフの頂点から同じレベルにある singal を結ぶ “等高線” を定義する.
3. スコープがモジュール内である singal に対して，最初の書き込みから次の書き込みまでにその値を参照するものが最初の書き込み結果を参照するものである.

なお，最初に書き込まれる前に値を参照した場合には不定値が読まれる．また同一の “等高線” 上で複数の箇所から書き込まれる場合は，それらのうちいずれか 1 つが非決定的に選択され書き込まれる.

4.2.2 ファンクション内部の動作モデル

Evelyn のプログラムは，signal を単位に実行される．すなわちある singal y に対して，その値を計算するための関数を次のように定義し，これを実行の単位とする.

$$y << f(x_0, \cdots, x_n)$$

以降ではこれを実行ユニット と呼ぶ．図 12に実行ユニットを示す.

サイクル毎に，当該サイクルで実行されるべき全ての実行ユニットについて,

1. 入力 singal の中に有効な値を持たないものがあるか，出力 singal がブロックされているならば，サスペンドする.
2. 全ての入力 singal が有効な値を持ち，出力 singal への書き込みがブロックされていないなら，入力 singal から値を読み出し，演算処理を実行し，結果を出力 singal に書き込む.

という処理を行なう.

```
module モジュール識別子 {
    generic:
    private:
    public:
    configuration:
    port:
};
```

図 13: モジュール部の構成

Evelyn 記述は，静的に構築される実行ユニットの有向グラフと考えられる．Evelyn 記述を動作させるとは，この有向グラフに対してデータを流すことに他ならない．

ファンクション内部の動作モデルと，モジュールの動作モデルの関係は以下のようになっている:

Evelyn のファンクションは，一般のプログラミング言語の関数のように呼び出し時に動的にデータ領域が確保されるようなものではない．データ領域は全て静的に確保される．この意味で Evelyn のファンクションはむしろ VHDL のコンポーネントなどに近いものである．

ファンクションは実行時に全て展開され，1 つの実行ユニットの有向グラフが作られる．ファンクション呼び出しとは，それによって新たにデータ領域が確保されるようなものではなく，常に存在している 1 つのファンクション領域に対してデータを渡すことで実現されるものである．

5 動作記述

本章では具体的な動作記述方法を説明する．

5.1 モジュール部

モジュールはモジュール部で定義する．モジュール部の役割は，ファンクション，内部変数などの宣言と内部モジュールのコンフィギュレーションである．

```
x1 << 0;                                (1)
x2 << 1;                                (2)
( y1, y2 )func( x1, x2 );               (3)
z << y1 + 1;                            (4)
```

図 14: ファンクション呼び出し

図 13にモジュール部の構成を示す．図のようにモジュール部は generic 部，private 部，public 部，configuration 部，そして port 部からなる．これらの役割は以下に述べる通りである．

generic 部　コンパイル時に与えられる変数を宣言する．

private 部　モジュールにローカルな変数/ファンクションの宣言，及び内部モジュールのインスタンス生成を行なう．

public 部　モジュール外部からもアクセスできる変数/ファンクションの宣言を行なう．

configuration 部　内部モジュール同士の接続を定義する．

port 部　外部モジュールと通信するためのポート (後述) を定義する．

5.2　ファンクション

5.2.1　ファンクション

ファンクションの外部インタフェースの定義はモジュール部の private 部か public 部で行なう．以下にファンクション定義のフォーマットを示す．

(返値リスト) ファンクション識別子 [配列サイズ](引数リスト);

ファンクションは複数の引数/返値を持つことができる．仮引数リストと仮返値リストに同じ名前のものが存在したら，それは参照呼び (*call by reference*) である．それ以外のものは全て値呼び (*call by value*) である．

ファンクションは配列として定義することもできる．配列として定義したい場合には，宣言時に配列サイズを与える．配列サイズはコンパイル時に決定できるものでなければならない．詳しくは 5.2.2節で説明する．

図 14にファンクション呼び出しの例を示す．引数 x1，x2 の値を (1)(2) で生成し，(3) でファンクション func を呼び出す．

この例では x1，x2 ともに singal であるので，(1)(2) の実行がブロックすると後続命令の実行もブロックする．その場合，ブロックする位置は (3) のファン

クション呼び出し位置ではなく，ファンクション func 内部で x1，x2 の値を参照する位置である．つまりファンクションを呼び出すときに引数が全て揃っている必要はない．

同様に返値も準備のできたものから返される．この例では y2 が返されなくても y1 が返ってきた時点で (4) を実行してしまう．

なおファンクション呼び出し時に引数を揃えておく必要がないからといって，例えば (1) を (3) の後に移動するようなことをしてはならない．これは，Evelyn ではプログラム・オーダによって依存関係を判定するためである．

ところで，実引数から仮引数への値の受け渡しは代入ではなくユニフィケーションである．引数の受け渡しによりステージが区切られることはない．

また返値への代入は，ファンクション内部で行なわれる仮返値への代入が真の代入である．仮返値から実返値への値の受け渡しはユニフィケーションであって代入ではない．従って，ここでステージが区切られることはない．

5.2.2 ファンクションの配列

Evelyn のファンクションは，C などの関数のように呼び出し毎に動的にインスタンスが生成されるようなものではない．ファンクション毎に 1 つのインスタンスが静的に生成される．

同一のファンクションのインスタンスを複数生成したい場合にはファンクションを配列として宣言する．配列インデックスが違うファンクションは別のインスタンスが生成される．

ファンクションは静的に生成されるインスタンスなので，配列サイズは静的に決定するものでなければならない．

5.2.3 制約ファンクション

同時に発行される複数のファンクション呼び出しを統一的に処理したい場合には，処理を制約ファンクションを用いて記述する．

図 15に制約ファンクションの記述例を示す．図はアービトレーション回路で，NUM 個のモジュールからリクエストを受けるとトークン・リング方式によりアービトレーションを行なう．

まず，モジュール部の public 部でアービトレーション処理を行なうファンクション request を宣言する．ファンクション request は，同時に受けたリクエストを統一的に処理しなければならないため，制約ファンクションとして宣言する．ファンクション request は，NUM 個の配列として宣言する．配列インデッ

```
module arbitor{
    generic:
        const int NUM;
    private:
        singal int owner;
    public:
        restricted (singal boolean grant[NUM])request[NUM]();
}
// ------------------------------
restricted(singal boolean grant[NUM])request(){
    signal int i;
    signal int idx;
    signal boolean flag;

    flag <= false;
    for ( i <= 0; i < NUM; i++ ){
        idx <= (OWNER+1+i) % NUM;
        if ( (port[idx] in CALLER) ){
            if ( flag == false ){
                grant[idx] << true;
                owner << idx;
                flag <= true;
            }
            else{
                grant[idx] << false;
            }
        }
    }
}
```

図 15: 制約ファンクションの例

クスにより caller を特定するためである．

コメント行以降が制約ファンクション request の内容である．図中，*CALLER* とは特別な変数で，当該サイクルで呼び出された制約ファンクションの配列インデックスが格納されているものである．ファンクション request には以下のようなルールに従って動作するように記述されている．

- OWNER に記録された現在のオーナの隣のモジュールから順に調べていき，そのサイクルでリクエストを発行している最初のモジュールにグラントを与える (grant = true を返す)．
- それ以外のリクエストを発行しているモジュールに対してはグラントが得られなかったことを伝える (grant = false を返す)．
- リクエストを発行していないモジュールに対しては何もしない．

5.3 外部モジュールとの通信

前にも述べた通り Evelyn では通信をファンクション呼び出しに抽象化する．外部モジュールに対して通信を行なう場合も基本的に同じであるが，1 つ問題が生じる．それは，外部の通信相手を陽に記述してしまうと，設計のポータビリティが大幅に損なわれてしまうことである．

そこで，**ポート** というものを用いて外部のモジュールと通信する．ポートが外部モジュールとの通信を仲介する．

外部モジュールとの通信が必要なモジュールにはポートを用意する．そして上位モジュールでそのモジュールをインスタンス化する際に，ポートを通信相手と接続する．

1 つのモジュールは任意個のポートを持つことができ，また 1 つのポートを任意個のモジュールに接続することができる．従って，複数のモジュールを仮想的に 1 つのモジュールとみなしたり，逆に 1 つのモジュールを複数のモジュールとみなすこともできる．

つまり図16のように，port2 を通して moduleB(の一部) と moduleC (の一部) を 1 つのモジュールとしてみなしたり，moduleB を port1 と port2 を通して別々のモジュールとみなすことができる．

モジュール部でそのモジュールが持つポートの定義を行なう．ここでポートの定義とは，

- ポートの名前

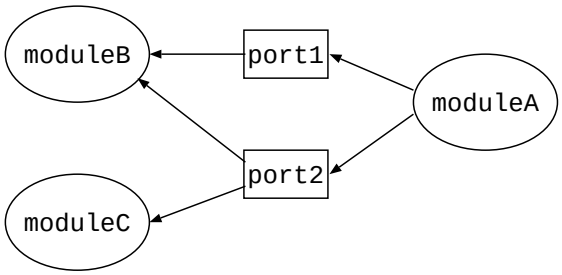

図 16: ポートを介した通信

```
module A{
    private:
        ( singal int i )send();
    port:
        port0( (singal float a)send() );
};
```

図 17: ポートの宣言

- そのポートを介してアクセスするファンクションのリスト

を定義することである．

図 17はポートの定義を含むモジュール部の例である．module A は port0 を介して外部モジュールの send というファンクションにアクセスする．

ポート同士の接続は，上位モジュールでそのモジュールのインスタンスを生成するときに行なう．図 18は，図 17で定義したモジュール A を内部に持つモジュール C のモジュール部である．まず private 部で C が持つ内部モジュール the_A と the_B を宣言する．そして，configuration 部で，図 17で extern と宣言したファンクション send をモジュール the_B の send であると定義する．

5.4 singal

本節では，singal を用いて様々な記憶構造を記述する方法を述べる．

5.4.1 同期方法の指定

Evelyn では，singal を記憶プリミティブとして提供している．singal は読み書きが同時にできる高機能なものである．しかし，高速性よりもハードウェア

```
module C{
    private:
        A the_A;
        B the_B;
    configuration:
        the_A[]{
            port0( send → the_B.port1.send );
        };
    port:
}
```

図 18: ポート同士の接続

量を優先したいときなど，よりシンプルなものが欲しい場合がある．

そのような場合は，singal を組み合わせて目的の回路を表現する．singal を用いて表現するということで，より複雑な回路が合成されるのではと考える読者もいるかもしれないが，機能合成系が正しく合成するので問題ない．

singal とは基本的に読み出しと書き込みが同時にできるものである．もし読み出しと書き込みが同時にできないようにしたい場合には，設計者が singal を用いてそのような記憶を定義する．

図 19は，同時に読み書きのできない記憶 (モジュール sequential_singal) の定義例である．モジュール sequential_singal は write(書き込み)，read(読み出し) の 2 つのファンクションを持つ．これら 2 つのファンクションを用いて，実際に値を保持する変数_core にアクセスする．

ファンクション write では，演算子?を用いている．?は Evelyn に組み込みの演算子で，singal に有効な値が保持されている場合は真を返す．また?の位置によりステージが分割される．

while ループにより，_core が有効な値を持っている間待ち続け，_core が有効な値を失なった時点で新たな値を書き込む．

ファンクション read は，write とは独立に_core の値を読み出す．

このように記述すれば，同時に読み書きのできない sequential singal を定義することができる．singal のみを使う場合に比べて記述量が増えるため，複雑な回路が合成されてしまうようにみえるが，機能/論理合成系が適切に合成する

```
module sequential_singal{
    private:
        singal int _core;
    public:
        (signal int data)read();
        ()write(signal int data);
}
()sequential_singal::write(data){
    while( ?_core ){
        NULL;
    }   _core << data;
}
(singal int data)sequential_singal::read(){
    data << _core;
}
```

図 19: 同時に読み書きできない記憶

ので問題ない.

5.4.2 メモリの記述

Evelyn でメモリを記述する方法を述べる.

Evelyn でメモリを表現するには記憶プリミティブとして signal を用いる. これは, メモリの, あるアドレスの特定の値を参照する回数を静的に特定することが一般に不可能であるからである.

signal の配列で, 後に述べる条件を満たすものはメモリとして機能合成される.

signal は同期機構を持たないため, 同期のための回路は人手で記述しなくてはならない. すなわち, 複数箇所からアクセスされる可能性がある場合には排他制御のための回路を記述する必要がある. singal によりメモリアドレスレジスタやメモリデータレジスタを記述することになるだろう.

さて, signal の配列をメモリ/レジスタファイルとして合成するには, 同一サイクルで 1 つないし 2 つの特定の配列要素にしかアクセスされない必要がある.

このための十分条件は,

- 制約ファンクションで保護されているか, 1 つないし 2 つのポートを介してアクセスされるファンクションからのみアクセスされる.
- それらのファンクション内で同一サイクルで 1 つないし 2 つの配列要素にしかアクセスされない.

ことである.

機能合成系は, メモリとして実現するという合成ディレクティブを与えられた signal の配列で, この条件を満たすものをメモリとして機能合成する.

5.5 記述例

前節までに Evelyn における動作記述方法について述べた. 本節では Evelyn の具体的な記述例を示す.

ここでは動作記述例として IBM 360/91 の演算装置の記述する. IBM 360/91 の演算装置は Tomasulo のアルゴリズムにより演算を Out-of-Order 実行している.

まず図20は, 演算装置全体(alu) に対応するモジュール部の宣言である. generic 部で定数を宣言し, private 部で内部モジュールを宣言している. 内部モジュールの宣言では, generic 部で受けとった定数を使っている. これらの定数は, 内部モジュールの generic 定数となる.

```
module alu{
    generic:
        const int FL_SIZE; // 浮動小数点命令スタック
            ⋮
    private:
        flos the_flos( FL_SIZE ); // 浮動小数点命令スタック
        flr the_flr( FLR_SIZE ); // 浮動小数点レジスタ
            ⋮
    public:
        () set();
        () add( singal tag_t dtag, singal float src1, singal float src2 );
        () mul( singal tag_t dtag, singal float src1, singal float src2 );
};
```

図 20: Tomasulo のアルゴリズム:モジュール部

また public 部で以下の 3 つのファンクションを宣言している.

set デコードした命令をリザベーション・ステーションに書き込む.

add 実行可能な命令をリザベーション・ステーションから取り出して加算を実行する.

mul 実行可能な命令をリザベーション・ステーションから取り出して乗算を実行する.

次に図 21 に演算装置 (alu) のファンクションの 1 つ，set の抜粋を示す．このファンクションには alu の動作の前半部，すなわちデコードした命令をリザベーション・ステーションにセットするまでの動作を記述している.

ファンクションには，(1) 浮動小数点命令スタック (the_flos) からデコード済みの命令を取り出し，(2) 浮動小数点レジスタ (the_flr) からオペランドを読み出し，(3) 加 (乗) 算器用リザベーション・ステーション (the_a(m)rsvst) に書き込み，(4) 最後にリザベーション・ステーションのエントリのタグを the_flr に書き込む，という一連の処理をそのまま記述している.

このような記述は Evelyn によるハードウェア記述の特徴をよく表している. Evelyn ではまず図 21 のように動作を記述し，その後でステージ境界としたい変数を singal に変更するのである.

```
() alu::set(){
                :
    /* 浮動小数点命令スタック (FLOS) から命令を取り出す */
    ( inst, dreg, lreg, rreg )the_flos.get();

    /* 浮動小数点レジスタ (FLR) からオペランドを読み出す */
    ( lbusy, ltag, ldata )the_flr.read( lreg );
    ( rbusy, rtag, rdata )the_flr.read( rreg );

    /* リザベーション・ステーションへの書き込み */
    if ( inst == ADD ){ /* the_arsvst : 加算器用リザベーション・ステーション */
        (dtag)the_arsvst.put(ltag,ldata,rtag,rdata);
    }
    elsif ( inst == MUL ){
                :
    }
    /* FLR のタグ, ビジービットをセットする */
    ()the_flr.setbusy( dreg, dtag );
}
```

図 21: デコードした命令をリザベーション・ステーションにセットするファンクション (抜粋)

```
( singal int tag )rsvst::put
            ( singal int ltag, rtag, singal float ldata, rdata )
{
    signal int i;
    singal boolean success << false;

    while ( !success ){ /* 前のサイクルで書き込めなかったら */

        success << false;
        for ( i <= 0; i < SIZE; i++ ){ /* 全てのエントリを探して */
            if ( rs_entries[i].vld == false ){ /* 空きエントリを見つけたら */
                rs_entries[i].set( ltag, ldata, rtag, rdata, true );
                tag << i;
                success << true;
            }
        }
    }
}
```

図 22: リザベーション・ステーションへの書き込み

図22に示したファンクション put は，リザベーション・ステーション (rsvst) が持つファンクションの 1 つである．ファンクション put の役割は，リザベーション・ステーションの空きエントリにオペランドを書き込み，そのエントリのタグを返すというものである．

このファンクション put は，singal の性質を利用した Evelyn 独特の記述となっている．ポイントはリザベーション・ステイションに空きエントリがないときの処理である:空きエントリがない場合には，そこで実行をブロックしてリザベーション・ステーションが空くのを待たねばならない．ファンクション put ではこれを singal を用いて巧みに実現している．

図中 for ループではリザベーション・ステーション中のエントリを順に調べていき，空きエントリをみつけたらそこにデータを書き込むという処理をしている．この処理を while ループで繰り返し実行することで，エントリが空くのを待っている訳である．

この while ループの条件となっている singal success が記述の肝である．success を singal で実現し，while ループの各イタレーションで必ず success に代入するようにしていることで，while ループのイタレーションを 1 サイクルで実行するように記述しているのである．

すると，

- サイクル毎に for ループが 1 回実行される．
- while ループを抜けるまでは put の引数 (全て singal) にアクセスし続ける．

ために，put の引数の singal のために，後続データの実行がブロックされるのである．

5.6 動作記述のまとめ

以上，Evelyn の動作記述方法について述べた．

Evelyn は従来の機能レベル HDL とは異なり，ハードウェアを動作単位に記述し，そこにより詳細な設計情報を付加していくというアプローチを採る．そのため，動作記述の段階では動作設計のみに専心できる．

動作設計段階で機能レベルの設計情報を必要としないため，動作の記述/変更は容易に行なえる．そのため，7章で述べる動作シミュレータと組み合わせて使えば，開発前の性能評価や動作レベル設計検証を効率よく行なうことができるため，方式設計の後半から利用することができる．しかも Evelyn の動作記述を

その後の設計工程でそのまま利用できることを考えれば，従来，C 言語などで行なわれていたシミュレーションを Evelyn に置き換えられると考えられる．

次章ではこのような記述が問題なくシミュレーション，機能合成ができるということを示す．

6 機能合成

以上，Evelyn によりトップダウンにハードウェアを記述できることを述べてきた．以下本稿ではこの Evelyn 記述のシミュレーション/機能合成手法について述べる．

まず本章で Evelyn の機能合成[1)]手法について述べる．以下機能合成の流れについて述べた後，単一ファンクションに限った機能合成について説明し，ファンクション間での資源共有などモジュール全体の機能合成について説明する．

6.1 機能合成の概要

6.1.1 機能合成系

Evelyn による動作設計記述は，**機能合成系**により機能設計記述に自動合成される．機能合成系は，動作記述と次節で述べる合成ディレクティブを入力として受けて，VHDL による機能設計記述を出力する．

機能レベル記述として VHDL 記述を合成する理由は，その後の論理シミュレーション，論理合成/最適化工程に広く流通している優れたツールを使うことができるという点のみである．そのようなツールを使うことで Evelyn をすぐに実際のハードウェア設計に利用可能とすることを狙っている．

6.1.2 合成ディレクティブ

設計情報のうち，動作に性能以外の点で影響を及ぼさないようなものは，**合成ディレクティブ**として与える．このようにすることで，物理的制約を満たすために設計変更が必要になった場合でも，動作記述に全く影響を与えることなく設計変更を行なうことができる．

この合成ディレクティブは，単に合成時のオプションとして与えるのではなく，言語仕様の一部として規定する．言語仕様として規定し，そのセマンティクスを明確にすることで，記述の意味が処理系依存に依存するようなことが起

[1)] 動作合成と表現されることがあるが，誤用である．機能レベル記述を合成するのだから機能合成と表現すべきである．

```
enum beatle( john, paul, george, ringo )
    map to bit_vector( "00","01","10", "11" );
```

図 23: 列挙型のビット列へのマッピング

```
singal int data range 0 to 255;
```

図 24: データの範囲の指定

らないようにする．このようにすれば記述のポータビリティが高まり，設計者が処理系の癖に悩まされることがなくなる．

合成ディレクティブの例を示す．

Evelyn で使用できる抽象型は，機能合成時に合成ディレクティブを与えることで詳細化することができる．具体的には以下のようなディレクティブを与えることができる．

- boolean 型の'0'，'1' へのマッピング．論理を正論理，負論理のどちらで実現するか選択する．
- 列挙型のビット列などへのマッピング．
- データのビット幅の指定．

ただし VHDL でも列挙型を使うことができるので，全ての変数/型について詳細化を行なう必要はない．設計上クリティカルな場合やチップの外部ピンなどで仕様が予め定められている場合，そしてプログラムの意味を機械的に読みとるのが困難で論理合成系が的確に処理できないと考えられる場合のみ行なえばよい．

列挙型 (boolean を含む) のビット列へのマッピングは図 23のようにする．図中，イタリック体の記述が合成ディレクティブである．動作記述においてその列挙型を定義している箇所に，これを VHDL の bit_vector 型にマッピングするという意味の合成ディレクティブを与える．

一方，データのビット幅の指定は図 24のように行なう．これも図 23と同様，イタリック体の記述は合成ディレクティブである．変数の宣言箇所にビット幅を指定するディレクティブを与える．

では具体的な機能合成方法を説明する．

```
module A{
    private:
        ()funcA(
            addr map to bus : bit_vector( 35 downto 0 ),
            data0 map to bus : bit_vector( 63 downto 0 ),
            data1 map to bus : bit_vector( 63 downto 0 )
        );
}
```

図 25: entity port へのマッピング

6.2 モジュールの機能合成

本節ではモジュールの機能合成手法を説明する．モジュールの機能合成では，複数ファンクション間での資源共有が大きなテーマとなる．

6.2.1 ファンクションのエンコーディング

Evelyn ではモジュール間通信をファンクション呼び出しに抽象化しているが，機能合成する際には，これを論理信号に詳細化する必要がある．

すなわちファンクションを呼び出すということに含まれる情報 (以下ではコマンドと呼ぶ) がある．実際のハードウェアではこれは信号線として実現される．機能合成する際に実現方法を指定しなければならない．

さらに，時間的に幅を持ってやりとりされるファンクションの引数/返値を，適切な信号線にマッピングする必要もある．

マッピングの指定は，6.1.2節で述べたように詳細化したデータを，VHDL の文法で記述した *entity port* にマッピングすることで行なう．

図 25は具体例である．図中イタリック体で記述したものが合成ディレクティブである．この例は，funcA というファンクション を持つモジュール A に合成ディレクティブを付加したものである．

funcA の宣言部で，addr を bit_vector 型の *bus* の 35〜0 ビット目にマッピングし，data0 と data1 を *bus* の 63〜0 ビット目にマッピングするようにディレクティブを与えている．

この例では，モジュール A は 1 つのファンクションしか持たないため，コマ

ンドのエンコーディングは登場していないが，コマンドのエンコーディングも同様に行なう．すなわちモジュールが持つファンクションを一種の列挙型とみなして，6.1.2節で述べたようにビット列にマッピングする．そしてさらにこのビット列を信号線にマッピングする．

6.2.2 資源共有

回路規模の制約のために，回路を共有しなければならないことがしばしばある．ハードウェア記述言語でも，回路の共有を指定できねばならない．

回路の共有は次の 2 つに分類できる．

1. 論理回路レベルの共有．
2. 機能レベルの共有．

前者は，例えば $z_0 = f(x, y)$ と $z_1 = f(x, y)$ の部分回路を共有し最小化する，というようなものである．このような細かい最適化は論理合成系の担当である．従って，Evelyn では 6.2.3 節で述べた機能モジュールへのマッピングのディレクティブにより，最適化しやすい論理を 1 つの機能モジュールにマッピングする，という処理のみを行なう．

一方，後者は演算器，メモリなどのより大きなレベルの共有の話である．このレベルの共有では，資源を排他的に利用しないと正しく動作しない．従って，動作レベルで排他制御のための回路を記述しなければならない．つまり Evelyn では，このような共有を行なうためには制約ファンクションを用いて動作レベルから共有させなくてはならない．

確かに，場合によっては排他制御を行なわなくてもコンフリクトが生じないこともある．しかしそのような状況を処理系が自動的に認識するのは困難である．従って，機能レベルの共有を行なう場合には必ず制約ファンクションを用いて共有される回路を記述しなければならない．

6.2.3 機能モジュールへのマッピング

Evelyn では，動作モジュール単位にハードウェアを記述する．動作モジュールは回路の共有などによる最適化を考慮に入れていない分割単位である．機能合成時には，これを機能モジュールにマッピングする必要がある．

機能モジュールへのマッピング方法は次のように行なう:まず物理的制約上特にクリティカルな部分に対しては，ファンクションのステージをさらに分割した実行ユニットの機能モジュールへのマッピング方法を設計者が記述する．マッピング方法を指定しない部分については，ファンクションの 1 つのステージを

1 つの機能モジュールに機械的にマッピングする.

実行ユニットの機能モジュールへのマッピング方法は次のようにする: 実行ユニットは出力側の singal により一意に指定できる. 1 つの機能モジュールにまとめて合成したい実行ユニットの出力側の singal のリストを合成ディレクティブとして与える.

6.3 ファンクションの機能合成

次に, 単一のファンクションの合成方法を説明する.

ファンクションは, 基本的には全てパイプラインに合成される. しかし例えばゲート数がクリティカルである場合, 処理するデータの性質上パイプライン処理の効果が期待できない場合など, パイプライン動作しない回路を合成したい場合がある.

そのような場合には, singal を組み合わせることでパイプライン動作しないように動作記述する. 5章で, 書き込みと読み出しが同時にできないラッチの記述例を示したが, それと同じ要領でパイプライン動作しないように記述する.

さて, パイプライン動作させるかどうかにかかわらず, ファンクションの機能合成は, (1) 一連の動作をステージに分割し, (2) ステージ毎に機能合成する, という手順で行なう. 動作をステージに分割してしまえば, 後は signal で記述された部分を組み合わせ回路として合成し, singal で記述された部分を 6.4節で述べるように合成すればよい.

そこで本節では, ファンクション機能合成の前半部分, ステージ分割の方法について説明する.

6.3.1 ステージ分割手法

ステージ分割は, 3.4節で述べたように行なう. すなわち,

1. 動作記述からデータフロー・グラフを作成する.
2. データフロー・グラフに対して, グラフの上から同じ高さにある singal を結ぶ “等高線” を引く.
3. “等高線” の位置でステージ分割.

という手順でステージ分割を行なう.

単純な場合ならばこのようにすることでステージ分割を行なうことができるのだが, 場合によっては簡単には “等高線” を定義できないことがある. そのような場合のセマンティクスを説明する.

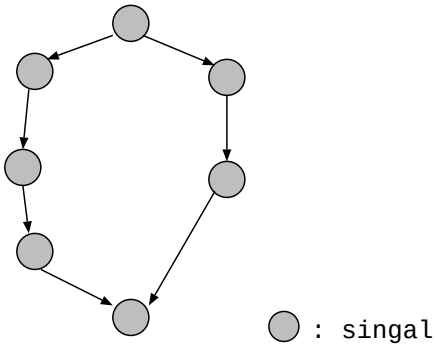

図 26: ステージ数の異なるパスが合流する

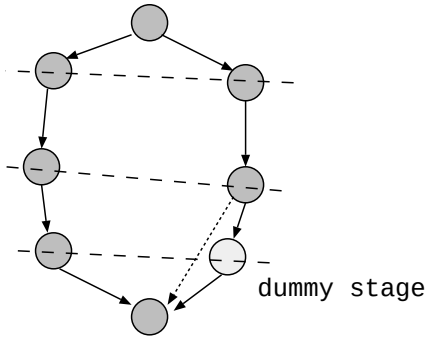

図 27: ダミーの代入を挿入してステージ分割する

ステージ数の異なるパスが存在する場合　図 26のように，合流地点までに至るパスがいくつかあって，そこまでのステージ数がパスによって異なるような場合がある．

この場合，次の 2 つの合成方法のうち，いずれかを選択する．

1. 記述の意味する回路をそのまま合成する．
2. ステージ数が等しくなるように途中にダミーの代入を挿入して合成する．

まず 1 の方法であるが，この場合合成された回路はパイプライン動作しない．流れるデータが疎で，パイプライン処理するメリットがあまりない場合には，この方法で合成すればよい．

2 の場合であるが，機能合成系は合流ポイントの直前にダミーの代入 (ステージ境界) があると解釈する．つまり，図 26のようなデータフロー・グラフに対しては図 27のように解釈し “等高線” を定義する．

ファンクション呼び出し　ファンクションを呼び出す場合，一般にそのファンクションが何サイクルで実行されるのか分からない．このような記述を合成す

```
singal data;
    ⋮
if ( cond == true ){
  data << 0;
}
```

図 28: cond の値によって data への代入回数が異なる

る場合には，設計者は次のいずれかの合成方法を選択する．

- 呼び出されるファンクションを展開してデータフロー・グラフを作る．
  確実であるが，分割合成が行なえない，回路規模が大きくなるので機能合成にかかるコストが増大する，などの欠点がある．
- 呼び出されるファンクションが何サイクルかかるか合成ディレクティブとして指定する．
  分割合成は行なえるが，他の設計変更によって合成をやり直さなくてはならない場合が生ずる．
- 呼び出されるファンクションとハンドシェイク通信を行なう回路を合成する．
  分割合成が安全に行なえるが，性能/ゲート数の点で不利な回路が合成される．

それぞれに一長一短あるので，記述する回路に合わせて適切に選択すればよい．

if 文　if 文では，条件を満たすかどうかで singal への代入回数が異なる場合がある．例えば図 28のような例では，cond の値によって data への代入回数が異なる．これをそのままステージ分割すると，cond の値によってステージ構成が変わってしまう．

この場合，次の 2 種類の機能合成の方法が考えられる:

- ダミーステージを挿入して，各節のステージ数を揃える．
- 記述の意味する回路をそのまま合成する．

ダミーステージを挿入する方法では，最も遅いものに合わせて動作するため性能の点で不利であるが，ハードウェアは単純になる．なお，ダミーステージを

```
if ( cond == true ){
  data << 0;
}
else{
  data << data;
}
```

図 29: 暗黙の else 節が存在するとする

挿入する箇所であるが，特に指定しなければ機能合成系は適当に挿入する．しかし物理的制約などの点でクリティカルな場合には設計者が指定することもできる．

先の図 28の場合では，図 29 のようにダミーの else 節にダミーの代入が記述されているものとして単一のパイプラインを合成する．

一方，記述の意味する回路をそのまま合成する方法では，そのままではパイプライン動作しない回路が合成される．これは，長さの異なるパイプラインを並列に動作させると，複数パイプライン間でデータの追い越しが発生する可能性があるからである．

しかし，データをパイプラインに投入した順序を保存する回路を用意することで，if 節，else 節を別々のパイプラインとして合成することができる．この方法では，性能の点では有利であるが，パイプラインの制御のための回路が複雑になり，回路規模の増大を招くことになる．

ループ　if 文と同様，ループ (for ループ，while ループ) に対しても単純には“等高線” を定義できない．ここでは機能合成のためのループのセマンティクスを述べる．

さて，ループは次のように分類できる．

- ループ回数の上限が静的に決定できるか否か．
- 組み合わせ回路的なループ (signal のみで記述されたループ) か，順序回路的ループ (singal への代入を含むループ) か．

まずループ回数の上限が静的に決定でき (定回ループ) かつ組み合わせ回路的なループであるが，これはループを全て展開した回路を機能合成する．

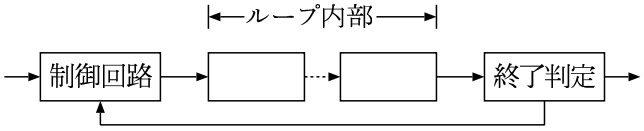

図 30: ループの機能合成

次に，ループ回数の上限が静的には決定できず (不定回ループ) かつ組み合わせ回路的なループは機能合成不可能である．従ってこのような記述は機能合成時エラーとなる[1)]．

そして，順序回路的ループであるが，これは定回，不定回にかかわらず合成可能である．

このようなループは図30のように合成する．すなわちループ内部を通常のパイプラインとして合成し，ループの入口に制御回路，ループの出口に終了判定のための回路を合成する．ループがビジーの場合には，制御回路が続くデータの投入をブロックする．

順序回路的な不定回ループのうち無限ループとなるもの – 特に，状況によって無限ループとなりうるものは，ハードウェアとして重大なバグとなってしまう可能性がある．しかし処理系が自動的にこのようなバグを発見することは極めて困難である．従って，機能合成可能なものをハードウェアとして正しく動作するようにするのは，設計者の責任とする．不定回ループを持つ回路を設計する場合には特に慎重に動作検証を行なわなければならない．

6.4 singal の合成

まず，動作記述の基本となる singal の機能合成方法から説明する．

動作記述におけるプリミティブである singal は，ハードウェアとしては非常に高機能なものである．高機能であるということは，様々な実現方法が考えられるということでもある．機能合成系は，ハードウェア量は増えても速く動作するもの，遅くても構わないからできるだけシンプルなもの等，設計者の様々な要求に柔軟に応えなければならない．

[1)] 7章で述べるように動作シミュレータではこのような記述は受理される．従って，機能合成を目的としない回路 – 例えばテストベンチを記述するためにこのようなループを用いることは一向に構わない．

6.4.1 方針

結局のところ，singal とは同期機構を持つ記憶回路である．従って，singal の合成方法を決める要素として，

- 使用する記憶回路：フリップフロップを使うか，ラッチを使うか．
- 同期の強さ：読み出しと書き込みが同時にできるか．

の 2 つが考えられる．

このうち，同期の強さに関しては，5章でも述べたように，動作記述として記述する．

従って，使用する記憶回路の種類のみ合成ディレクティブとして与える．

6.4.2 クロックの規定

ところで，Evelyn の動作にはクロックは陽に記述されない．同期回路 – つまり singal がクロックに同期して動作する回路を合成する場合には，クロックを定義する必要がある．

Evelyn ではクロックの定義は合成ディレクティブによって与える．合成ディレクティブでは次のようなクロックを定義できる．

- 任意個の external clock(外部から供給されるクロック)
- 任意個の gated clock

しかし，論理合成/最適化系の中には単一のクロックにしか対応していないものもある．下流工程の制約を満たすような合成ディレクティブを与えなくてはならない．

クロックを定義した後で，各 singal に対してどのクロックと同期するかを指示する合成ディレクティブを与えることで，同期回路を機能合成することができる．

また Evelyn は非同期回路を合成することもできる．非同期回路を合成するには singal を非同期式ラッチとして機能合成すればよい．

6.4.3 記憶素子の指定

まず記憶素子の選択について．記憶素子は以下の 3 種類から選択する．

- フリップフロップ (エッジ・トリガ型)
- トランスペアレント・ラッチ (パルス・トリガ型)
- 非同期式ラッチ

エッジ・トリガ型を選択した場合には，クロック信号名も同時に指定する．パルス・トリガ型を選択した場合には，制御信号を明示的に指定しなくても自動

的に合成する．

また，クロックを全く定義せず，全てのsingalに非同期式ラッチを選択すれば，非同期回路を機能合成することができる．

なお，素子へのリセットの必要性は，機能合成系が自動判定する．すなわち，宣言時に値が初期化されているsingalについてはリセット付きの記憶素子を用いて合成する．明示的に初期化されていない場合にはリセットなしの素子を用いる．

7　動作シミュレーション

本節では，Evelynの動作シミュレータについて説明する．まず動作シミュレータがサポートする機能について述べ，動作シミュレーション手法について述べる．

7.1　動作シミュレータが提供する機構

動作レベルHDLを用いた設計の利点の1つは，動作設計の段階でシミュレーションによる動作検証を行なえるという点である．開発の早い段階でアルゴリズムレベルのバグを取り除くことができるため，開発効率が向上する．

Evelynの動作シミュレータも，動作シミュレーション/検証を強力にサポートするために種々の機構を盛り込む．

7.1.1　テストベンチの記述

動作シミュレーション時に，設計対象だけでなく，設計対象のまわりにテストベンチを接続して系を構築し，系全体をシミュレーションするということがよくある．

設計対象単体では抽象的な動作の意味が分かりづらい，あるいは設計対象の外部インタフェースが複雑である，というような場合には，テストベンチを含む系全体が正しく動作するか確認すると設計検証が行ない易いからである．

このような目的のために記述されるテストベンチは，当たり前のことだが機能合成されるものではない．そこで，そのような記述をする場合には，6章で述べたような機能合成のための規定を満たさなくても構わないようにする．例えば組み合わせ回路的な無限ループをテストベンチに用いることができるようにする．

しかし，機能合成するつもりで機能合成不可能なものを記述してしまうのも問題である．そこで，Evelynでは機能合成可能なものと不可能なものを言語仕

様として明確に規定する．そして，コンパイル時にオプションを付けることで合成モードと非合成モードを切り換えるようにする．

7.1.2 時相論理式記述のサポート

合成を必要としない抽象的な記述のサポートをさらに進めて，時相論理式記述もサポートする．

近年の回路の大規模化に形式的設計検証技術がついてこないということもあり，動作シミュレーションによって初めて本格的な設計検証が行なえるということも少なくない [5]．

また，例えば共有メモリ型並列計算機においてソフトウェアとハードウェアのインタフェースを規定するメモリモデルなど，インタフェースの抽象度の高い仕様を時相論理式によって簡潔に規定しているものも多い．

そこで，動作シミュレータで時相論理式記述をサポートする．これによってEvelyn による動作記述がより高い抽象レベルの仕様を満たしているか検証できるようになる．時相論理式による仕様記述がテストベンチよりも簡潔に記述できる場合には，設計検証のコストが大幅に削減できることが期待される．

以上機構を動作シミュレータがサポートする機構について述べた．これらのサポートにより動作レベル設計検/デバッグは非常に効率的なものとなることが予想される．

さて，以下では動作シミュレータの構成方法を述べる．

7.2 Evelyn の動作モデル

動作シミュレータの構成方法について説明する前に，Evelyn の動作モデルについて述べる．動作モデルについては 4章で軽く述べたが，ここではシミュレーションの立場からの Evelyn の動作モデルについて説明する．

Evelyn はローレベルでは実行ユニットが動作の単位となる．実行ユニットは，全ての入力 singal で有効な値が揃い，かつ出力 singal がブロックされていないときに，入力 singal から値を読み出し，演算を実行し，結果を出力 singal に書き込む．

ハイレベルでは，モジュールが互いに相手のファンクションを呼び出しあうことで動作する．しかし Evelyn のファンクションは静的に生成される 1 つのインスタンスである．従って，Evelyn のプログラムは，ファンクションを全て展開した，1 つの実行ユニットのネットワークと捉えてよい．

動作シミュレーションとは，静的に構築した実行ユニットのネットワークにデータを流して動作させることに他ならない．

7.3 シミュレーション手法

それでは，前節で述べた動作モデルを効率的にシミュレーションするための方法について述べる．

シミュレーションする場合，実行ユニットの実行順序のスケジューリングが問題になる．そこで本節では，実行ユニットのスケジューリング方法について述べる．まず，7.3.1節でスケジューリング上問題になる点について述べ，7.3.2節でスケジューリング手法について述べる．

7.3.1 制約ファンクションの実行順序

制約ファンクションは，複数の呼び出しを同時に処理するファンクションである．従って，制約ファンクションを実行する前に当該サイクルでその制約ファンクションを呼び出すファンクションを全て把握していなければならない．

しかしここで問題が生じる．ある組み合わせ回路的な制約ファンクションの実行結果によって別の制約ファンクションを実行するか決まる場合があるのである．この場合先に前者を実行してから後者を実行しないと正しい動作とならない．

従って，単に呼び出される可能性のある制約ファンクションを把握するだけでなく，その実行順序スケジューリングにも注意する必要がある．

制約ファンクションの実行順序は次のように決定すればよい:

1. 予め全てのステージに対して，そのステージが実行する可能性のある制約ファンクション間の依存関係グラフを作成しておく
2. 当該サイクルで実行する可能性のある制約ファンクションに対して，その依存関係グラフに矛盾しないように，制約ファンクションの実行順序を決定する．

もし制約ファンクション依存グラフに矛盾が存在するために制約ファンクションの実行順序が決定できないとしたら，それは組み合わせ回路のループが存在しているということになる．従ってそのような場合は実行時エラーとなる．

従って，制約ファンクションを実行する可能性のあるプロセスの実行順序は次のように決定すればよい: まず制約ファンクションを呼び出す可能性のないファンクションを全て実行する．そして制約ファンクションは制約ファンクショ

ン依存グラフの上から実行していく.

7.3.2 実行スケジューリング

以上述べたことを実現する実行スケジューリング手法を述べる.

アルゴリズムには2つのsingalのリストを用いる:1つはまだ実行されていないsingalのリスト(未実行singalリスト)であり,もう1つは実行は完了したが出力側singalのためブロックされているもののリスト(ブロックド singalリスト)である.

実行アルゴリズムを以下に示す.

1. 当該サイクルで新たに生成された実行ユニットを未実行singalリストに加える.
2. 未実行singalリストの中から入力側singalがブロックされていない実行ユニットを取り出し,ワーキングメモリに移動する.
3. ワーキングメモリにある実行ユニットのうち,制約ファンクションを呼び出す可能性のあるものを前節で述べた実行順序に従って実行する.実行結果は出力側singalには書き込まず別の場所に保持しておく.実行ユニットはブロックド singalリストに移す.
4. それ以外の実行可能なものを実行する.2と同様に実行結果は別の場所に保存し,実行ユニットはブロックド singalリストに加える.
5. ブロックド singalリスト中の実行ユニットのうち,出力側singalがブロックされていないもの,ループがブロックされていないものについて実行結果を出力側singalに書き込む.同時に入力側singalのブロックを解除する.実行ユニットは終了させる.
6. 6によって出力側singalのブロックが解除されたものについて実行結果を出力側singalに書き込む.同時に入力側singalのブロックを解除する.実行ユニットは終了させる.
7. 収束するまで5を繰り返す.

8 おわりに

本稿では動作レベルHDL Evelynを提案した.

Evelynは動作記述に,段階的に詳細な設計情報を付加していくというアプローチを採っている.そのため,論理合成系などからのフィードバックがあった場合にも,そのフィードバックループを最小限に抑えることができ,設計変

更のコストが大幅に削減することができる．

Evelyn では，ハードウェアを動作単位に記述する．動作を時間方向に記述するため，一連の動作を自然な形で記述することができる．

このようなことが可能となったポイントの 1 つとして，singal が挙げられる．同期機構と記憶回路をセットにした singal によって，合成可能性を犠牲にせずに，動作単位時間方向のハードウェア記述を可能にした．

ステージ境界を変数により指定するこのアプローチは，半順序的かつ柔軟な時間関係を記述でき，また設計変更を容易に行なえる．このような時間関係記述は，近年注目されている非同期回路との親和性も高いと考えられる．

本稿では機能合成手法についても述べた．Evelyn は，動作記述の抽象度が高いため合成可能性を疑問視されがちである．しかし，ファンクションをステージに分割さえしてしまえば，あとは各ステージを組み合わせ回路とラッチで実現するだけなので，見た目程機能合成は困難ではない．

加えて，合成ディレクティブにより細かな機能合成方法を指定することができる点も大きなポイントである．

他の動作レベルでは，記述の抽象度と合成能力を両立させるために，記述の対象を限定するというアプローチを採っているものが多い．

それに対し Evelyn では．合成ディレクティブにより合成方法を細かく指示することで記述の抽象度と合成能力のトレードオフを高いレベルでバランスさせている．そのため Evelyn を特定の分野に限らず，一般のハードウェアの設計に活用できる．

さて，本稿では機能レベル HDL について厳しく批判してきた．しかし Evelyn のようなハードウェア記述が可能なのは，実のところ機能レベル HDL とその処理系の論理合成技術によるところが大きい．すなわち，Evelyn を設計するにあたって，機能レベルの設計記述にまで変換できれば満足できるクオリティの回路が論理合成できるということを前提としているのである．

今までは，機能レベルの記述を満足できるクオリティで論理合成することが目標であった．そして今，その論理合成技術は成熟し，その限界が見えつつある．Evelyn は設計自動化の分野の次の一歩である．

謝辞

本研究の機会を与えて下さり，適切な御指導を賜わった富田眞治教授に深甚な謝意を表します。

また，貴重な御助言をいただいた森眞一郎助教授，五島正裕助手に深く感謝致します.

さらに，日頃暖かく御鞭撻下さった京都大学大学院工学研究科情報工学専攻富田研究室の諸兄に感謝致します.

最後に，メンター・グラフィックス・ジャパン株式会社の Higher Education Program の一環として製品とサービスをご提供頂いたことに感謝します.

参考文献

[1] 安浦寛人, 山田輝彦, 他: 特集「ハードウェア記述言語–新しいシステム設計環境の実現に向けて–」, 情報処理, Vol. 33, No. 11 (1992).

[2] 西田浩一, 山田晃久, 神戸尚志, Kay, A., 野村俊夫: ハードウェアコンパイラ Bach, 信学技報 CPSY97–87, pp. 143–148 (1997).

[3] 若林一敏, 古林紀哉, 他: 伝送用 LSI を動作合成で開発，機能設計の期間が 1/10 に短縮, 日経エレクトロニクス 2–12, No. 655, pp. 147–169 (1996).

[4] 森本貴之, 斉藤一志, 中村宏, 朴泰祐, 中澤喜三郎: 方式レベル記述言語 AIDL を用いた高性能プロセッサ設計支援, 情処研報 96–ARC–121–8, 96–DA–82–8, pp. 57–64 (1996).

[5] 舟本一久, 福島直人, 五島正裕, 森眞一郎, 中島浩, 富田眞治: 超並列計算機 JUMP-1 のキャッシュシステムの論理設計検証, 情報処理研究会報告 96–ARC–119–38, pp. 221–226 (1996).