ScalaによるFPGAアプリケーション開発用DSLの設計

三好 健文[1,a)]

概要：FPGA は，I/O インテンシブな処理や，低消費電力かつ高性能での動作が必要なアプリケーションの開発プラットフォームとして注目されている．しかしながら，FPGA 開発において主流の HDL を使った RTL 設計は，あくまで回路設計であり，ユーザが実現したいアプリケーションの開発との乖離が大きく，設計が煩雑である．
そこで，FPGA を使ったアプリケーション開発を手軽かつ迅速に実現するための Scala を用いた DSL を設計した．設計した DSL では，FPGA アプリケーションを実装するために必要な状態遷移や組合せ回路を扱うクラスと演算子を提供する．Scala のクラスやパタンマッチを活用しながら，簡潔，かつ，再利用性の高いハードウェア設計が可能になる．

キーワード：DSL，FPGA

1. はじめに

Field Programmable Gate Array(FPGA)[12] は，プログラム可能なハードウェアデバイスであり，ユーザが自由にハードウェアロジックをその上に構築できる．そのため，ASIC 開発のプロトタイプ環境としての利用に加え，アプリケーションに応じたユーザ独自の専用ハードウェア開発の環境としても利用される．

FPGA を用いたアプリケーション開発では，プロセッサ上でソフトウェアとして実装する場合に比べ，並列性の活用による低消費電力で高い処理能力の実現が期待される．アーキテクチャを工夫し，データ並列性とパイプライン並列性を活用することで，プロセッサと比較して数十倍から数百倍以上の性能向上が得られる [4][10]．また，単に高速に演算処理を実現できるというだけではなく，FPGA はアプリケーションを構成する処理回路が入出力信号を直接操作することができるため，低レイテンシ，高スループットで物理デバイスにアクセスできる点に強みを持つ．文献 [5] では，FPGA を使って 10Gbps のネットワークインターフェースに直結したラインレートで処理可能な Memcached エンジンの実装について述べている．FPGA 上の Memcached は，単位時間あたりのリクエスト処理数の電力効率が x86 サーバの実装に比べて 36 倍であり，ラウンドトリップタイムが最長でも 4.6 μ 秒と，低レイテンシで処理できることが報告されている．

加えて，クロックレベルで決定的な処理を実装できることも，アプリケーションを専用ハードウェアとして実装するときの強みである．現代的なプロセッサで動作するソフトウェアでは，キャッシュなどの実行支援ユニットの動作や，複数のプログラ

[1] わさらぼ合同会社
[a)] miyo@wasa-labo.com

ムのコンテクスト切り替えなどによって，実行タイミングや処理にかかる時間を正確に管理することが難しい．一方で，専用ハードウェアとしてアプリケーションを実装する場合には，クロック単位での信号の変化を自分で制御することができる．

FPGA の性能を効率良く活用するためには，一般に，VHDL や Verilog を用いた RTL 設計が必要であり，プロセッサ上で動作するソフトウェア開発に比べて，人的，時間的な開発コストが大きい．特に，アルゴリズムとして複雑な処理の RTL 記述は繁雑で手間がかかり，時にはバグの温床となる．また，アプリケーションに合ったアーキテクチャで処理を実装することができれば高い演算性能を達成することができる一方で，そうでない場合には処理性能でプロセッサや ASIC を凌駕することは難しい．そのため，FPGA で高性能処理を実現するためにアーキテクチャ上の試行錯誤をする必要があり，このこともまた，開発コストを大きくする要因である．文献 [4] では，同じ処理を汎用プロセッサ，GPU，FPGA で実装する場合の時間を比較した結果，FPGA への実装は汎用プロセッサ向けの実装に比べて，数十〜数百倍の時間が必要だったと報告している．

ソフトウェア開発のように，手軽に FPGA 開発を行えるようにする取り組みとして，プログラムをハードウェアロジックに変換する高位合成がある．C/C++で記述されたプログラムをハードウェアロジックに変換する高位合成処理系は，CyberWorkBench[14] や LegUp[7]，ImpulseC など，多数存在する．FPGA ベンダの Xilinx は，自社の FPGA 開発ツールチェインに C/C++高位合成処理系の VivadoHLS[13] を統合し，この流れを後押ししている．また，同じく大手 FPGA ベンダの Altera は，OpenCL による FPGA 開発用 SDK[1] を提供している．C/C++以外の言語を対象した高位合成処理系もあり，たとえば，Java を入力言語とする Lime[2] や C#を入力言語とする Kiwi[11] などがある．

汎用言語を対象とした高位合成に加えて，特定のアプリケーションに特化した言語に対する処理系もある．たとえば，Matlab からハードウェアを生成する HDL Coder や LINQ を FPGA で高性能化するためのツール LINQits[8] がある．ユーザの導入コストが小さく，また，アプリケーションに特化したカーネル・ハードウェアをテンプレートして用意できるという利点がある．一方で，アプリケーション・ドメインに応じた，それぞれの処理系が必要であり，その開発コストは無視できない．このコスト軽減を目的として LMS を使った処理系開発手法 [9] も提案されている．

高位合成によるアプローチでは，既存の言語で記述したプログラムを FPGA 上の回路として移植することができ，開発コストを大きく下げられる可能性がある．その一方で，生成したハードウェアの質が処理系に強く依存するというリスクがある．処理系を設計する立場に立ってみても，既存のプログラミング言語で記述されたプログラムからデータ並列性やパイプライン並列性を抽出し，効率良く処理できる回路を生成することは簡単ではない．そのため，良い回路の生成には，ディレクティブや専用の構文や演算を導入して処理系に合成のヒントを与える機能が提供される．しかしながら，処理系のユーザからみれば，プログラムのように手軽なハードウェア開発というメリットが損なわれる．

プログラムをハードウェア化する高位合成のアプローチとは別に，FPGA 開発というドメインに対する DSL を定義することによって開発を手軽にする取り組みがある．このアプローチは，大きく二つに分類できる．一つ目は，RTL 設計に対する DSL である．これは，RTL 設計に必要な組合せ回路やレジスタ，およびそれらの結線といったプリミティブを，DSL のホスト言語の持つ型や抽象化システムで組み立てられるようにすることで，見通しの良いハードウェア設計を可能にする．たとえば，Python をホスト言語とする MyHDL や，Scala をホスト言語とする Chisel[3] がある．二つ目は，FPGA 開発に求められるプログラミングモデルをサポートするための DSL である．並行プログラミングモデルに基づいたハードウェア設計言語である Bluespec System Verilog[6](BSV) や，データフローモデルの高性能計算機を記述するた

めの Max Compier などがある．

DSL によるアプローチでは，高位合成処理系を使用する場合とは異なり，DSL を使って処理を新たに記述しなおさなければならない．一方で，処理に内在するデータ並列性やパイプライン並列性を引き出すことで高い処理性能を得る，リソースを節約することでコンパクトに回路を実現する，といった，設計上の工夫の反映が容易であるというメリットがある．特に，FPGA ならではのアプリケーションの実装には，ハードウェアならではの細粒度の並列性を活用して高性能な処理の実現や，クロックレベルでのデータの取り込みや処理が欠かせない．そのため，新規に FPGA アプリケーションを開発する場合には，高位合成を用いるアプローチよりも，DSL を使うアプローチの方が有効であると考えられる．

ここで，プロセッサ開発や評価を行なうような場合には，FPGA 上に実装するどんな回路でも記述できる RTL 設計が必要であり，一つ目のタイプの DSL が有効である．一方で，FPGA の上にアプリケーションを実装しようとする場合には，必ずしも RTL 設計のすべては必要ではないことが多く，二つ目のタイプの DSL のように，アプリケーション開発に適したプログラミングモデルに則した抽象的な表現がある方が開発は手軽になる．

以上を踏まえ，本稿では，FPGA アプリケーション開発を手軽にするための DSL として，Scala をホスト言語とする Synthesijer.Scala を提案する．Synthesijer.Scala では，クロックベースの状態遷移と組合せ回路の表現のみの記述に特化することで，手軽に FPGA アプリケーションを実現できる環境を目指す．また，Scala の内部 DSL とすることで，DSL 自体の仕様をコンパクトに保ちつつ，実装を楽にする仕組みを Scala の言語機能を使って提供する．特に，VHDL や Verilog では難しい，組合せ回路の手軽な再利用や，類似パタンの生成，複雑な計算式に基づいた定数式の事前計算によって開発効率が向上する．

2. FPGA アプリケーションの設計

FPGA は，プログラム可能なハードウェアデバイスであり，ユーザが自由にハードウェアロジックをその上に構築できる．しかしながら，ASIC 開発のプロトタイプ環境として FPGA を利用する場合を除いて，アプリケーションを FPGA を用いて実現するのは，代替の他の手段に比べて，高い処理性能が得られる，多数の I/O を利用するなどの FPGA ならではの特徴が活かされるべきケースである．

FPGA で実現可能なハードウェアの最大動作周波数は，汎用プロセッサの動作周波数に比べて極めて低い．そのため，所望の処理を FPGA で高速化するためには，アプリケーション適したアーキテクチャで実装する必要がある．アプリケーションに適したアーキテクチャとは，内在するデータ並列性が活用できること，及び，組み合せ回路で作成されたアプリケーション専用の演算ユニットを備えることである．また，処理を細かな入出力の単位に分割してパイプライン処理することで，処理スループットを向上させることもアーキテクチャに求められる．

言い換えると，設計言語には，データ並列性を活用するために同じ機能のユニットを処理すべきデータに対応づけして並べる，組み合せ回路による専用演算ユニットを手軽に記述・利用できるようにすること，処理をパイプライニングできるようにタイミングを確定できること，が求められる．

3. Synthesijer.Scala の設計

クロックベースの状態遷移と組合せ回路の表現のみの記述に特化したモデルを仮定し，手軽に FPGA アプリケーションを実現できる環境を目指す Synthesijer.Scala の設計と実装について述べる．まず，定義するモデルとコードの概要を示し，次に，Synthesijer.Scala の構成要素について，それぞれの詳細を説明する．また，Scala を使うことによる簡便な記述方法を示す．

3.1 Synthesijer.Scala で定義するモデル

図 1 に記述対象である FPGA アプリケーションの構成要素を示す．このモデルでは，FPGA 上のアプリケーションを，モジュール (`Module`) とし

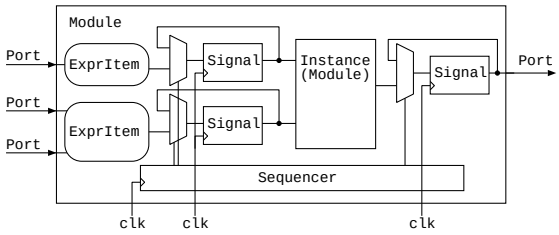

図 1 FPGA アプリケーションのモデル

```
def generate_led(n:Int) : Module = {
  val m = new Module("led")      // (1)
  val q = m.outP("q")            // (2)
  val counter = m.signal(32)     // (3)
  q := counter.ref(n)            // (4)
  val seq = m.sequencer("main") // (5)
  val s0 = seq.add()             // (6)
  seq.idle -> s0                 // (7)
  counter <= (seq.idle, m.VECTOR_ZERO) // (8)
  counter <= (s0, counter + 1) // (9)
  return m // (10)
}
```

図 2 サンプルコード

て定義する．モジュールは外部と信号のやりとりする複数の入出力ポート (Port) を持つ．Module の内部には，クロック毎に遷移可能な状態遷移機械 Sequencer があり，その状態に応じて，レジスタ (Signal) の値が確定する．レジスタとレジスタの間には演算ユニット ExprItem が存在する．モジュールは階層的に構築することができ，他のモジュールのインスタンス (Instance) を並べることができる．

Port，および Signal は，静的に決定された特定の bit 幅をもつ．ExprItem で使用できる演算は，表 A·1，A·2，A·3 に示す通り．ExprItem を組み合わせて作った演算は状態を持たない組み合せ回路に相当するものとする．

このモデルに基づいて，クロック毎にインクリメントするカウンタの特定ビットの値を外部に出力するモジュールを設計する例を図 2 に示す．図 2 のコードは，

(1) led という名前のモジュールを作成 (m と命名)
(2) モジュール m に出力ポート q を追加 (q と命名)
(3) m に 32bit のレジスタを追加 (counter と命名)

```
def main(args:Array[String]) = {
  val led = generate_led(5)
  led.genVHDL() // VHDL を生成
  led.genVerilog() // Verilog を生成
}
```

図 3 合成用の HDL を生成する

(4) 出力ポート q に counter の n-bit 目をバインド
(5) m に main という名前の状態遷移機械を追加 (seq と命名)
(6) seq に状態を追加 (s0 と命名)
(7) seq のアイドル状態 (seq.idle) から s0 への無条件遷移を追加
(8) seq.idle のときは，counter に 0 をセット
(9) s0 のとき，counter に counter+1 をセット (毎クロックのインクリメントに相当)
(10) 作成したモジュール m を返す

というハードウェアの設計情報に相当する．このメソッドで定義したモジュールを実際に FPGA 上で動作させるためには，図 3 に示すコードで，VHDL あるいは Verilog HDL に変換し，FPGA 開発ツールによって合成する．

3.2 Synthesijer.Scala の構成要素

Synthesijer.Scala の主な構成要素について述べる．

3.2.1 Module

ハードウェアモジュールの設計単位．後述する Sequencer，Signal など，すべての要素のインスタンスは，すべて Module に従属する要素として構成する．言語としては，Module に定義されたメソッドを使って生成する．Module 単位で，VHDL の entity あるいは，Verilog の module に対応づける．

3.2.2 Sequencer，State

状態遷移機械に相当する構成要素．状態遷移機械を構成する状態 (State) と状態間の遷移を保持する．State のインスタンスは Sequencer で定義された add メソッドを呼び出して生成する．Sequencer は生成された時点で，初期状態の idle ステートを持つ．状態同士は->演算子によって遷移を設定できる．->に条件に相当する ExprItem と State のタプルを指定すると，条件

が真の場合だけ遷移する遷移を定義できる．たとえば，seq.idle->(req，s0) と記述することで，req が'1' の場合にのみ seq.idle から s0 に遷移するハードウェアを定義できる．

RTL 設計では，状態遷移機械はレジスタを使った同期順序回路の結果として生成されるものであるのに対し，Synthesijer.Scala では状態遷移機械を構成要素として抽象化することで，処理の流れを処理系で簡単に取り扱うことができる．これは，処理の流れの見通しを良くし，また，処理のタイミングの調整を容易にする．

3.2.3 Signal，Port

Port と Signal は，ハードウェア内部での状態を保持する要素である．静的に定義された bit 幅を持つ．Port はモジュール外部との接続用に引き出される Signal である．

Port と Signal の値は，<=演算子あるいは := で値 (ExprItem) を指定する．<=には，値を確定する状態と値のタプルを与える．また，State によらず，常に値を代入する場合には :=を用いる．

図 2 の例では，状態 s0 の場合にのみ counter に counter+1 を代入するために <=を用い，状態によらず counter の n-bit 目を q に代入するため :=を用いている．

3.2.4 ExprItem

ExprItem は，演算構成要素である．Signal，Port および，それらを表 A·1，A·2，A·3 に示す演算子でつないだものが相当する．演算の過程では状態は持たない．すなわちハードウェアとしては組み合せ回路に相当する．

節 2 で述べたように，FPGA を効率良く利用するためには，組み合せ回路による専用演算ユニットを手軽に記述できる必要がある．Synthesijer.Scala では，組み合せ回路 ExprItem として演算を扱うことができるため，構文上の特別な記述を必要とせずに，組み合せ回路の構成を保存，再利用することが簡単になり，複雑な回路の記述が簡潔になる．

しかしながら，やみくもに大きな組み合せ回路を生成すると，回路の動作周波数は極端に低くなる．その解決のためには，組み合せ回路のデータパス中に適切にレジスタを挿入することが必要になる．ExprItem のインスタンスとして，組み合せ回路の部分式をプログラムで取り扱うことができることは，設計時の設計空間探索にも有用である．

```
class MemPort(m:Module, p:String, w:Int, d:Int){
  val we = m.inP(p + "_we")
  val oe = m.inP(p + "_oe")
  val addr = m.inP(p + "_addr", d)
  val din  = m.inP(p + "_din", w)
  val dout = m.outP(p + "_dout", w)
}
```

図 4 メモリアクセスポートの定義

3.2.5 Instance

Instance は，モジュール内で他にモジュールの機能を組込む場合に利用する要素である．Instnace で定義される signalFor メソッドを使って，信号を接続する．

3.3 Scala の構文を利用したコード記述手法

Synthesijer.Scala では Scala の構文を利用して定義した要素を組み立てることで，所望のハードウェアを構成する．要素の組み立てに Scala の構文を利用することで，記述コストの削減と，見通しのよいコードの記述が可能になる．本節では，記述事例を示す．

3.3.1 入出力ポートのユニット化

ハードウェアモジュールを定義する場合，しばしば，複数の入出力信号をまとめて扱いたいことがある．たとえば，メモリにアクセスする場合には，アドレス，データ，読み書きの制御信号が必要になる．これらを一つの意味のあるまとまりとして定義しておけば，再利用が容易である．たとえば，図 5 に示すように Scala のクラスで，複数のポートをまとめることで，これを実現できる．使用する側では，のようにメモリアクセスに必要なポートを生成できる．

また，複数のポート群に対して，初期化シーケンスがあるような場合には，クラス内でそのシーケンスに対応する動作を定義しておくことで，ポートを利用する側に対してブラックボックス化できる．

```
def gen_a_module(n:Int) : Module = {
    val m = new Module("hoge")
    val mem = new MemPort(m, "mem", 32, 10)
  }
```

図 5 メモリアクセスポートの定義

```
val all_din = for(i<-0 until 32)
               yield(inP("din_" + i, 16))

all_din.reduce((x:ExprItem,y:ExprItem) => x & y))
```

図 6 類似のオブジェクトを生成する例

3.3.2 類似オブジェクトの生成

VHDL や Verilog HDL では，言語の提供する generate 文によって，多段の演算ユニットや多数のレジスタを規則的に並べることができる．しかしながら，使える場所が限定的で全般で使用できるわけではない．

特に，類似した名称の大量のオブジェクトを機械的に定義するような場面では generate 文が使用できない．また，generate で生成した集まりそのものを，コード内で使いまわすことができず，煩雑な記述になるケースが多い．

Scala の DSL である Synthesijer.Scala では，たとえば，図 6 のように for を使って多数の入力ポートを生成できる．さらに，生成した集合に対する演算によって手軽に扱うこともできる．図 6 の例では，16bit の入力ポート `din_0`〜`din_31` を for 文でコレクションとして生成している．また，コレクションに対する reduce 関数を使って，全入力ビットの結合を簡潔に定義している．

3.3.3 直列化/復元

ストリーム処理のようなアプリケーションでは，データの直列化と復元はよく記述される．処理レイテンシとスループット向上のためには，直列化した入力データを復元しながら処理する必要があるが，その場合，データの復元ステートマシンと処理の記述は煩雑になりがちである．Synthesijer.Scala では，状態遷移を `Sequecer` を使って扱うことができ，また，その組み立てを Scala のループに任せることで，簡易に記述できる．

```
var s = seq.add()
for(i <- 0 to 1){
  i match {
    case 0 => for_first(s, data)
    case 1 => for_second(s, data)
  }
  s = s -> seq.add()
}
for_rest(s, data)
s -> (done, done_state)
```

図 7 直列化されたデータの復元と処理

4. まとめ

FPGA アプリケーションの開発を容易にするための DSL である Synthesijer.Scala を設計した．Synthesijer.Scala では，状態遷移機械によって処理の流れを取り扱うことができる．また，組み合せ回路に相当する演算要素を ExprItem としてオブジェクト化したことで，大規模な組み合せ回路の設計が容易になる．また，Scala の埋め込み DSL であるため，Scala の構文を使って設計の効率化を図ることができる．

今後の課題として，設計空間の自動探索など，抽象的な設計手法ならではの最適化手法の適用が考えられる．

なお，設計したツールは，`http://synthesijer.sourceforge.net/` で公開している．

参考文献

[1] Altera, Inc.: アルテラ SDK for OpenCL, `http://www.altera.co.jp/products/software/opencl/opencl-index.html`.

[2] Auerbach, J., Bacon, D. F., Cheng, P. and Rabbah, R.: Lime: a Java-compatible and synthesizable language for heterogeneous architectures, *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, New York, NY, USA, ACM, pp. 89–108 (online), DOI: http://doi.acm.org/10.1145/1869459.1869469 (2010).

[3] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J. and

Asanović, K.: Chisel: Constructing Hardware in a Scala Embedded Language, *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, New York, NY, USA, ACM, pp. 1216–1225 (2012).

[4] Benkrid, K.: Reconfigurable Computing in the Multi-Core Era, Internal Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies, Tsukuba, Japan (2010).

[5] Blott, M., Karras, K., Liu, L., Vissers, K., Bär, J. and István, Z.: Achieving 10Gbps Line-rate Key-value Stores with FPGAs, Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing (2013).

[6] Bluespec, Inc.: `http://www.bluespec.com/`.

[7] Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Brown, S. and Czajkowski, T.: "LegUp: high-level synthesis for FPGA-based processor/accelerator systems", *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, New York, NY, USA, ACM, pp. 33–36 (online), DOI: http://doi.acm.org/10.1145/1950413.1950423 (2011).

[8] Chung, E. S., Davis, J. D. and Lee, J.: LINQits: Big Data on Little Clients, *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, New York, NY, USA, ACM, pp. 261–272 (2013).

[9] George, N., Novo, D., Rompf, T., Odersky, M. and Ienne, P.: Making domain-specific hardware synthesis tools cost-efficient, *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 120–127 (online), DOI: 10.1109/FPT.2013.6718341 (2013).

[10] Gomez-Pulido, J. A., Vega-Rodriguez, M. A., Sanchez-Perez, J. M., Priem-Mendes, S. and Carreira, V.: Accelerating floating-point fitness functions in evolutionary algorithms: a FPGA-CPU-GPU performance comparison, *Genetic Programming and Evolvable Machines*, Vol. 12, No. 4, pp. 403–427 (2011).

[11] Greaves, D. and Singh, S.: Designing application specific circuits with concurrent C# programs, *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pp. 21 –30 (online), DOI: 10.1109/MEMCOD.2010.5558627 (2010).

[12] Hauck, S. and DeHon, A.: *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007).

[13] Xilinx, I.: Vivado 高位合成, `http://japan.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[14] 日本電気株式会社：CyberWorkBench, `http://www.nec.co.jp/soft/cwb/`.

表 A·1 ExprItem 同士で定義される二項演算

記号	演算内容
+，-，*	和，差，積
and，or，xor	論理積，論理和，排他的論理和
==，/=	等しい，等しくない
<，>，leq，geq	小さい，大きい，以下，以上
&，concat	結合

表 A·2 ExprItem と Int で定義される二項演算

記号	演算内容
+，-	和，差
==，/=	等しい，等しくない
<，>，leq，geq	小さい，大きい，以下，以上
>>，>>>	算術右シフト，論理右シフト
<<	左シフト

表 A·3 ExprItem に対して定義されるその他の演算

記号 (用例)	演算内容
e!	否定.
?($e0$, $e1$)	選択．自身が'1' のとき $e0$, '0' のとき $e1$ を返す
padding(n)	n-bit の算術 bit 伸長
padding0(n)	n-bit の論理 bit 伸長
drop(n)	先頭から n-bit を落とす
range($n0$, $n1$)	$n0$-bit〜$n1$-bit を切り出す
ref(n)	n-bit 目を取り出す

付　　録

定義される演算子，メソッドを示す．また，アプリケーションの基本的な構成要素となる状態遷移機械およびデコーダの記述例を示す．

A.1 演算子

`ExprItem` 間で使用可能な演算子について示す．二項演算子は `ExprItem` 同士あるいは `ExprItem` と `Int` 型の定数値間の中置演算子として利用する．表 A·3 中の演算の用例は記号覧の通り．

A.2 要素構成用のメソッド

`Module` の持つ構成要素のインスタンスは，定義

表 A·4 Module で定義される構成要素の生成メソッド

メソッド名	説明
inP(s,n) outP(s,n) ioP(s,n)	それぞれ，名前 s の入力，出力， 入出力ポートの生成． 1bit 幅の場合，n は省略可
signal(s,n)	状態を持つ信号の生成．名前 s は省略可． また，1bit の場合 n は省略可
sequencer(s)	状態遷移機械を生成
instance(m, s)	モジュール m のインスタンスを生成
value($n0$, $n1$)	$n1$-bit 幅の値 $n0$ の定数を作る

```
class VendingMachine(n:String,c:String,r:String)
extends Module(n,c,r){
  val nickel = inP("nickel")
  val dime = inP("dime")
  val rdy = outP("rdy")
  val seq = sequencer("main")
  val s5,s10,s15,s_ok = seq.add()
  rdy <= (seq.idle, LOW)
  rdy <= (s_ok, HIGH)
  seq.idle->(nickel, s5)
  seq.idle->(dime, s10)
  s5 -> (nickel,s10)
  s5 -> (dime, s15)
  s10 -> (nickel, s15)
  s10 -> (dime, s_ok)
  s15 -> (nickel, s_ok)
  s15 -> (dime, s_ok)
  s_ok -> seq.idle
}
```

図 A·1 ベンディングマシンの記述例

されたメソッドを使って生成する．メソッドの定義を A·4 に示す．

A.3 ベンディングマシンの例

状態遷移機械の例としてベンディングマシンの記述例を図 A·1 に示す．また，ユーティリティ関数の `visualize_statemachine()` を用いて描画した状態遷移図を図 A·2 に示す．設計するハードウェアモジュールをホスト言語である Scala で解析できることもまた，DSL による記述のメリットである．

A.4 デコーダの記述例

デコーダは入力データ応じて，あらかじめ決め

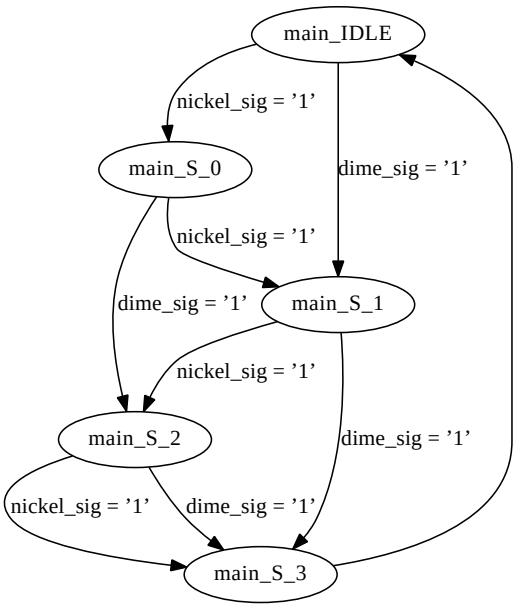

図 A·2 ベンディングマシンの状態遷移図

```
def decoder(s:ExprItem,
            l:List[(Int, Int)],
            w:Int) =
 l.foldRight(value(0,w).asInstanceOf[ExprItem]){
   (a,z) => ?(s == a._1, value(a._2, w), z) }
```

図 A·3 デコーダの記述例

```
// 0 から 9 に対応した LED の点灯マップのリスト
val tbl = List((0,0x7e),(1,0x30), ..., (9,0x79))
// 入力 data に対応したパタンを segment に出力する
segment := decoder(data, tbl, segment.width())
```

図 A·4 7 セグメント・デコーダの記述例

られた出力を生成する．記述例を図 A·3 に示す．これは，セレクタとなる変数 `s` と，セレクタに応じたパタン `l` から，デコーダを生成するメソッドの定義である．Synthesijer.Scala では Scala の `foldRight` によって簡潔に記述できる．

たとえば，7 セグメント LED の点灯パタンを生成する場合，図 A·4 のようにパタンリストを与えればよい．作成すべき回路とデータを分割できることで，コードの再利用によって，記述量が削減できる．