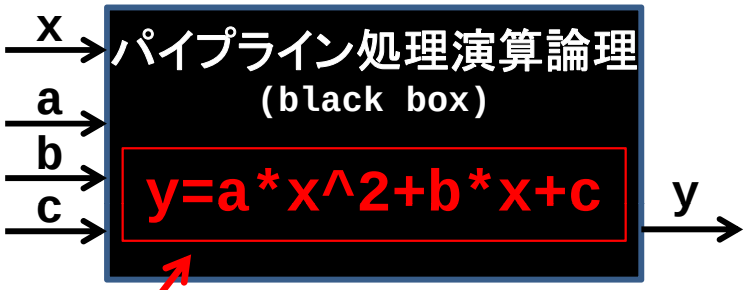
パイプライン処理のための演算仕様記述言語mhdlとその処理系

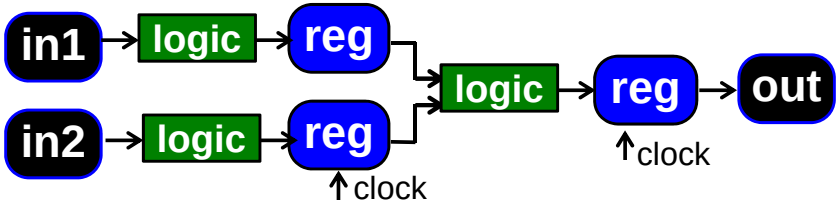
シグナル・プロセス・ロジック株式会社
瀬尾雄三

mhdl 開発の背景

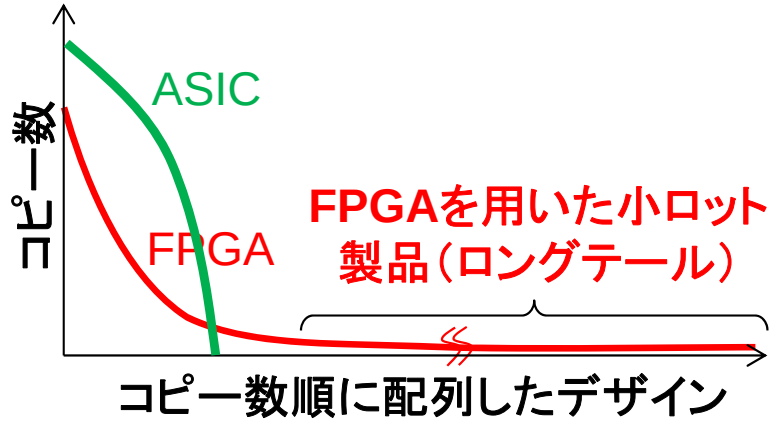
- *CodeSqueezer*
  ボタンを押すだけで数値演算論理を形成



パイプライン処理
演算論理をレジスタを介して接続
各論理回路は演算を同時に処理



- ツールの想定用途と要求
  - FPGAを用いた小ロット製品の設計
  - ツールへの要求
    + 安価で生産性の高いツールであること
    + 生成される論理の性能は優先しない

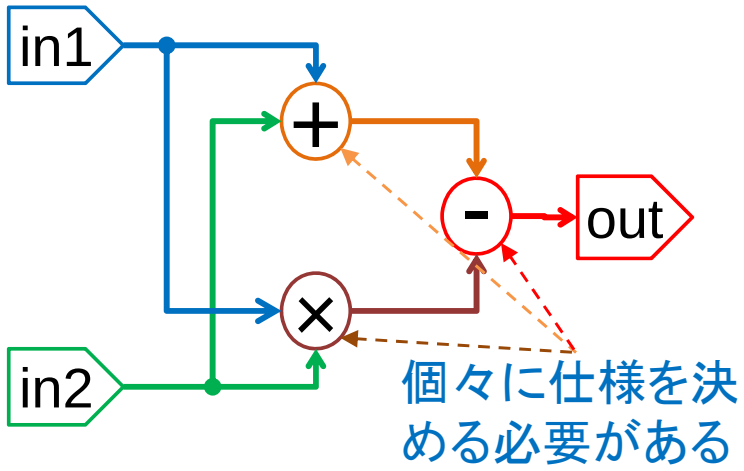


生産性向上のための課題と解決策

- 演算アルゴリズム
  - IPの選択、パラメータ設定は複雑
  ⇒ 演算子を用いた数式記述“mhdl”

```
cramer.(x, y)(a1, a2, b1, b2, f1, f2)
  d = a1 * b2 - b1 * a2
  x = (f1 * b2 - b1 * f2) / d
  y = (a1 * f2 - f1 * a2) / d
```

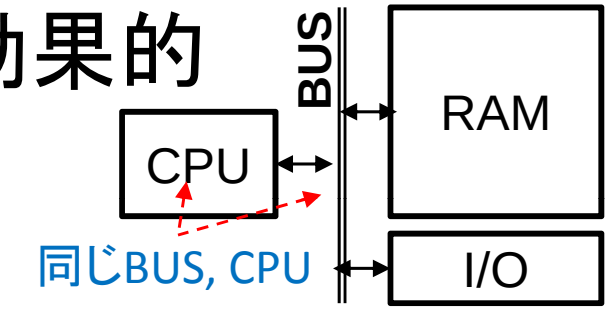
- 数値型の選択
  - ビット幅、符号の有無を個々に設定
  - 固定小数点数の取り扱いは複雑
  ⇒ 有効桁に着目した数値型の自動最適化
- タイミングの解決
  ⇒ レジスタの自動挿入

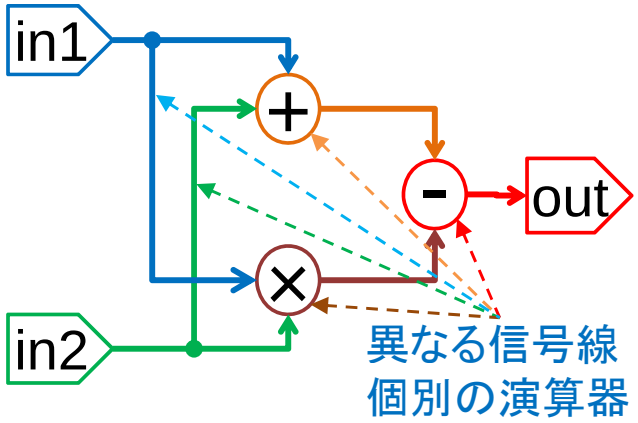


有効桁に着目した数値型の自動最適化

- 有効桁は現在の計算機では重視されていない
- パイプライン処理：個別最適化が効果的
  - 信号ごとに専用の演算器と信号線
  - 論理資源の消費削減にも有用
- 有効桁処理の特徴
  - 判断基準が明確：機械的処理が容易
  - 浮動小数点数を扱う必要があり、
    固定小数点型に比べ論理規模は増大

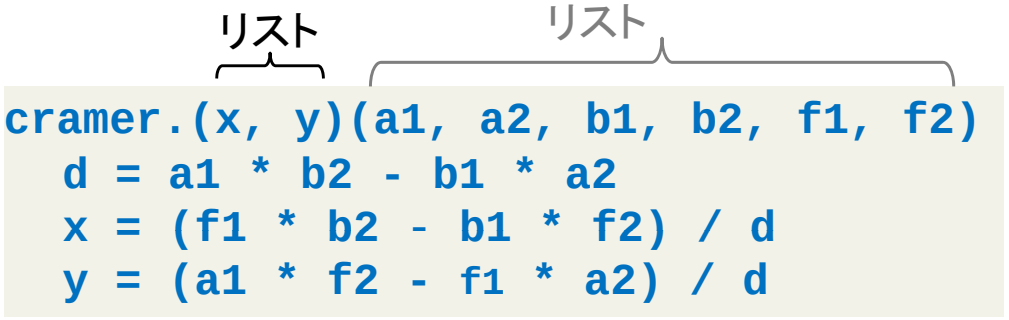
⇒生産性重視のコンセプトに合致





mhdlの言語仕様:基本思想

- 高生産性:容易なプログラミング
  - 高い抽象性、階層化、型なし言語
  - 簡素で読みやすいソースコード
    + 改行と字下げに意味をもたせ ;{}を排除
    + 複数の値を返す手続きを簡単に記述する:リスト



- 処理系とパイプライン処理に由来する制約
  - 入出力が同じ演算子や数値型に制約のある演算子は使用不可
  - フィードバックの禁止:タイミング制約を自動解決するため動作不定になる

⇒ Cとは異なる言語仕様

mhdlの言語仕様：プログラムの要素

- 使用される文字
  - 英字：大小文字を区別し、“_”を含む
  - 数字：0~9
  - 記号：演算子、括弧、空白、“.”、“,”
- 名前：英字に始まる英数字列。個々の変数、手続きを識別
- コメント：“//”から行末まで、および“/*”と“*/”の間
- 文
  - 行末は文を区切る
    ただし、演算子・“(”・“,”で終わる行は次行に継続
  - 文のレベル：行頭の空白の数
    + 文の制御範囲：その文よりもレベルの深い文が続く限り
  - 文の種類：手続き定義文、代入文（現在のところこの2種だけ）

mhdlの言語仕様:手続き

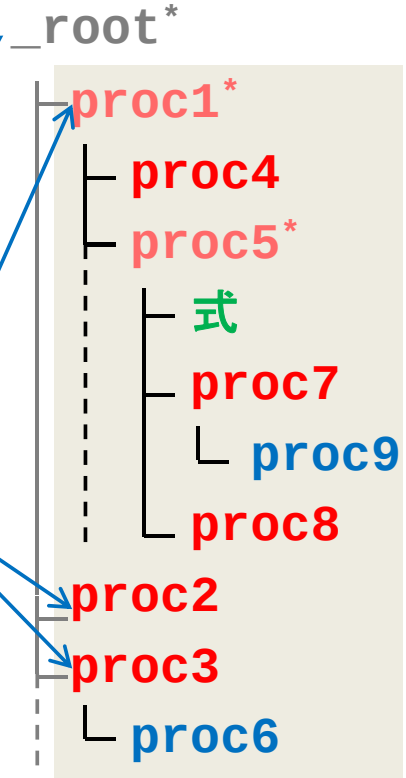
- 手続き宣言文:手続き定義を開始する
  - 形式: ***name.(out1, out2,...)(in1, in2,...)***
    - + ***name***:手続き名
    - + ***.(out1, out2,...)***:出力変数名。ピリオッド“**.**”に続けて記述
      - 単一の場合は括弧は不要
      - 全て省略された場合は手続き名が出力変数名となる
    - + ***(in1, in2,...)***:入力変数名
      - 単一の場合も括弧は省略できない
- 手続き定義本体
  - 手続き宣言文よりもレベルの深い文が続く間に記述
  - 任意の数の代入文と手続き定義が含まれる

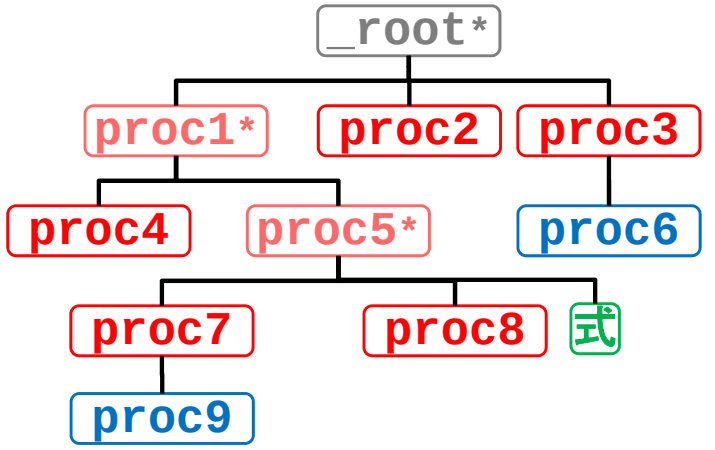
手続きの木構造と手続き名の通用範囲

- 手続き宣言の包含関係により手続きは木構造を形成
  - 手続き名は、その宣言が置かれた手続きの範囲内で通用する

_root：処理系が準備
_root直下に式を記述した場合
_rootが最上位手続きとなる
（手続き宣言文は自動形成）

最上位手続き
コンパイル開始手続きとなりえる





式を含むのはroc5,proc1,_root
この下で宣言された手続きが参照可能
再帰呼び出しはできない
式を含む手続き(*)は参照不可

mhdlの言語仕様：代入文と式

- 演算子：右図
- 代入文：代入演算子をもつ式
  - 代入に限りリストを扱える
    例：`(x, y) = div2(a, b)`
  - 記述順序
    参照される変数はそれ以前に定義される必要がある（フィードバックの禁止）
    定義：代入文左辺または手続き定義文の入力宣言部に変数名を記述すること
- 定数の記述：右図

優先度	演算子	演算内容	項数	演算順序
高 ⇕ 低	^	冪乗	任意	右→左
	*, /	乗算、除算	任意	不定
	+, -	加算、減算		
	<,<=,== !=,>=,>	比較	2	-
	&	論理積	任意	不定
	!	排他論理和		
	|	論理和		
	?, :	条件式	3	-
	=	代入	任意	右→左

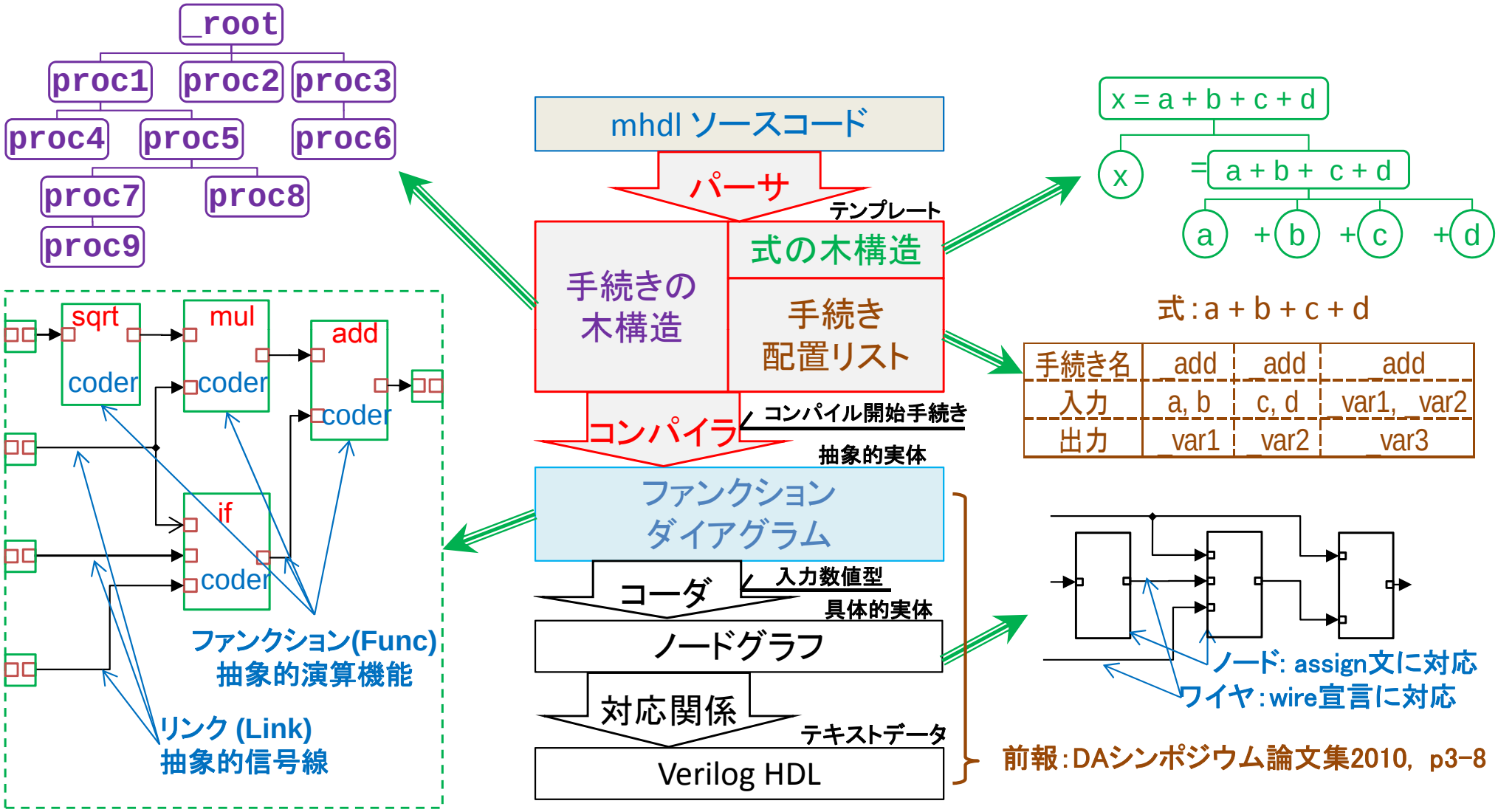
数値：先頭以外に“.”を含むことができる

`9.99E-9`

指数部数値：負号も可
E, D：底10の指数部
B：底2の指数部
e, d, bを用いてもよい

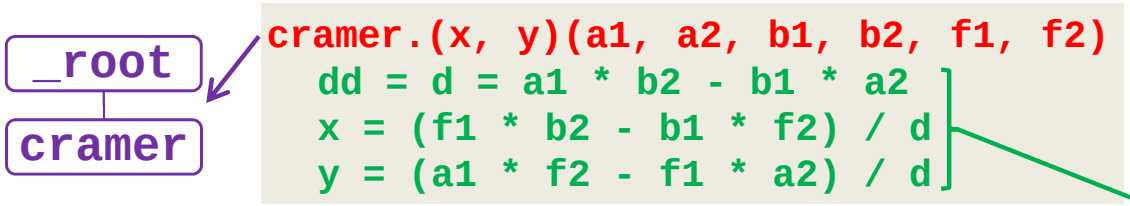
“.”、“E”、“e”を含めば誤差あり

mhdlの処理系：概要


_root
proc1
proc2
proc3
proc4
proc5
proc6
proc7
proc8
proc9
sqrt
coder
mul
coder
add
coder
if
coder
ファンクション(Func)
抽象的演算機能
リンク (Link)
抽象的信号線
mhdl ソースコード
パーサ
テンプレート
手続きの
木構造
式の木構造
手続き
配置リスト
コンパイラ
コンパイル開始手続き
抽象的実体
ファンクション
ダイアグラム
コーダ
入力数値型
具体的実体
ノードグラフ
対応関係
テキストデータ
Verilog HDL
x = a + b + c + d
x
=
a + b + c + d
a
+
b
+
c
+
d
式：a + b + c + d
手続き名
add
add
add
入力
a, b
c, d
var1, var2
出力
var1
var2
var3
ノード: assign文に対応
ワイヤ：wire宣言に対応
前報：DAシンポジウム論文集2010，p3−8

mhdlの処理系：手続きの処理

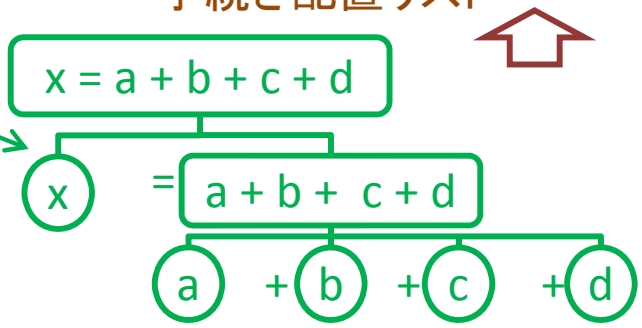
- 手続きクラス（**class Proc**）：手続き定義毎に生成
  - **手続き名、入出力変数名リスト**：手続き宣言文の記述に対応
  - **上位、下位手続きへのポインタ**：手続き包含関係に対応
  - **式のリスト**：括弧と演算子の優先度に基づく木構造に形成
  - **手続き配置リスト**：式のリストを変換して形成



式：a + b + c + d

手続き名	_add	_add	_add
入力	a, b	c, d	var1, _var2
出力	_var1	_var2	_var3

手続き配置リスト



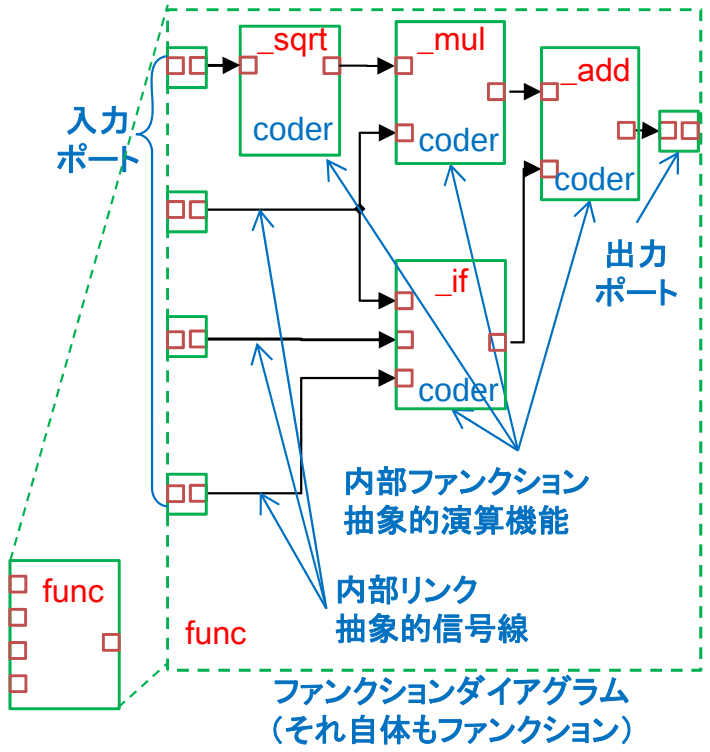
- 式の木構造
  - 式：同じ優先度の演算子で接続された項の並び
  - 項：変数・定数、手続き呼び出し、単項演算子に先導された項、式、のいずれか

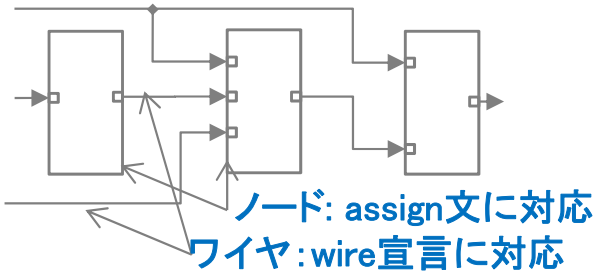
ファンクションダイアグラム（FD）の形成

- 手続きクラスからFDの形成

手続きクラス		FD
手続き名	→	FD名
入出力変数名リスト	→	入出力ポート
手続き配置リスト	→	内部ファンクション
上位下位手続きへのポインタ	→	手続き検索

- 定義済み変数（＝リンク）の管理
  - リストを初期化し入力変数を登録
  - 配置ファンクションの出力をリストに追加
  - 右辺変数は定義済みリストから検索
- FDからVerilog HDLコードの形成
  - 各コード化関数“coder”によりノードグラフに変換
  - 入力型を参照してコードを形成、出力の型を決定
  - ノードグラフに対してタイミング処理を行い、次いでVerilog HDLに変換する





詳細は、前報（DAシンポジウム論文集2010，p3–8）参照

まとめと今後の課題

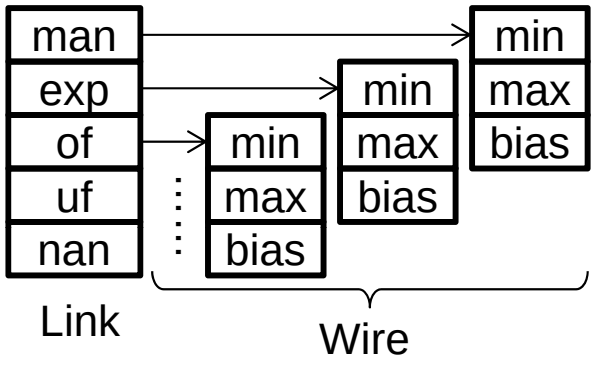
- パイプライン演算処理を記述する言語“mhdl”を提案した
- 特徴は、プログラミングの容易さ
  - 有効桁に着目した数値型の自動最適化
  - 改行と字下げに意味をもたせ、論理構造の把握を容易にした
- 今後の課題
  - 信号処理分野での論理設計支援ツールとしての機能拡充
    + 標準手続きの拡充：関数、時系列処理、...
    + ユーザ側での標準手続き開発環境を提供：Function Developer’s Kit
  - 一般的な数値演算分野を狙った言語仕様の拡張

ご清聴ありがとうございました

Appendix following

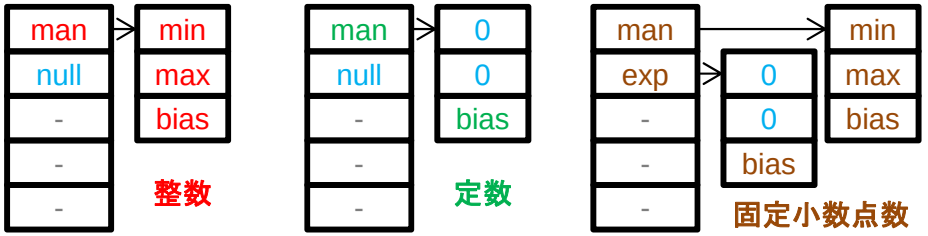
ビット幅を任意に設定できる数値型

- 数値（Link）を整数（Wire）の組で表現
  - Wire：仮数(man)、指数(exp)、異常フラグ（of, uf, nan）
  - Linkの値 = man x 2^{exp}
  - of：正にオーバー、uf：負にオーバー、nan：数値化不能
  - of, uf：数値の値は不定、nan：他のすべてが不定
  - 有効桁を求めるため、Link属性に誤差フラグを含める



- Wireの値 = 信号線の値 + bias
  - 信号線の値の範囲はmin～max（最小値～最大値）
    - ビット幅と符号の有無はmin, maxより形成
  - biasは属性として扱い、必要に応じて信号に加算

- 型の統一的取り扱い
  - 指数部がない：整数
  - min = max = 0：定数
  - 指数部が定数：固定小数点数



パイプライン処理はCPUを代替するか?

- CPU: 単一の万能演算器で複雑な演算を順次実行
  - 簡素な構成:70年前には唯一の実行可能解
    その後、半導体デバイスの高集積化・低価格化により代替技術も可能に
  - 万能演算器は効率的か、という疑問
- パイプライン化はスループット改善の定石
  - 組立産業:流れ作業、化学プラント:連続処理
  - デジタル演算:CPU、記憶装置内の信号処理
- パイプライン処理がCPUを代替する条件
  - コンピュータにはストアードプログラム方式が必須
  - FPGAも論理はPROMに格納⇒動的再構成に期待
  - ツール、OS、資源管理技術も必要