FPGA向け高位合成言語としてのJavaの活用手法の検討

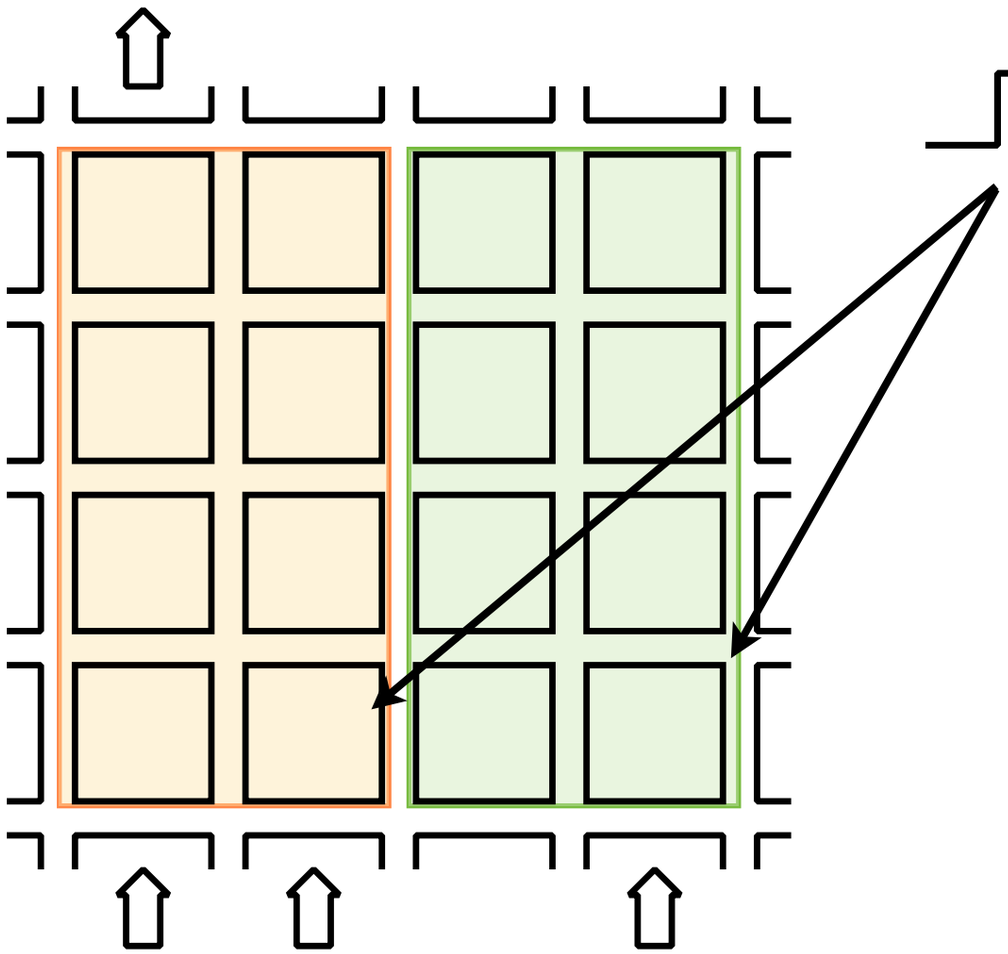
三好 健文[1)]　　船田 悟史[2)]

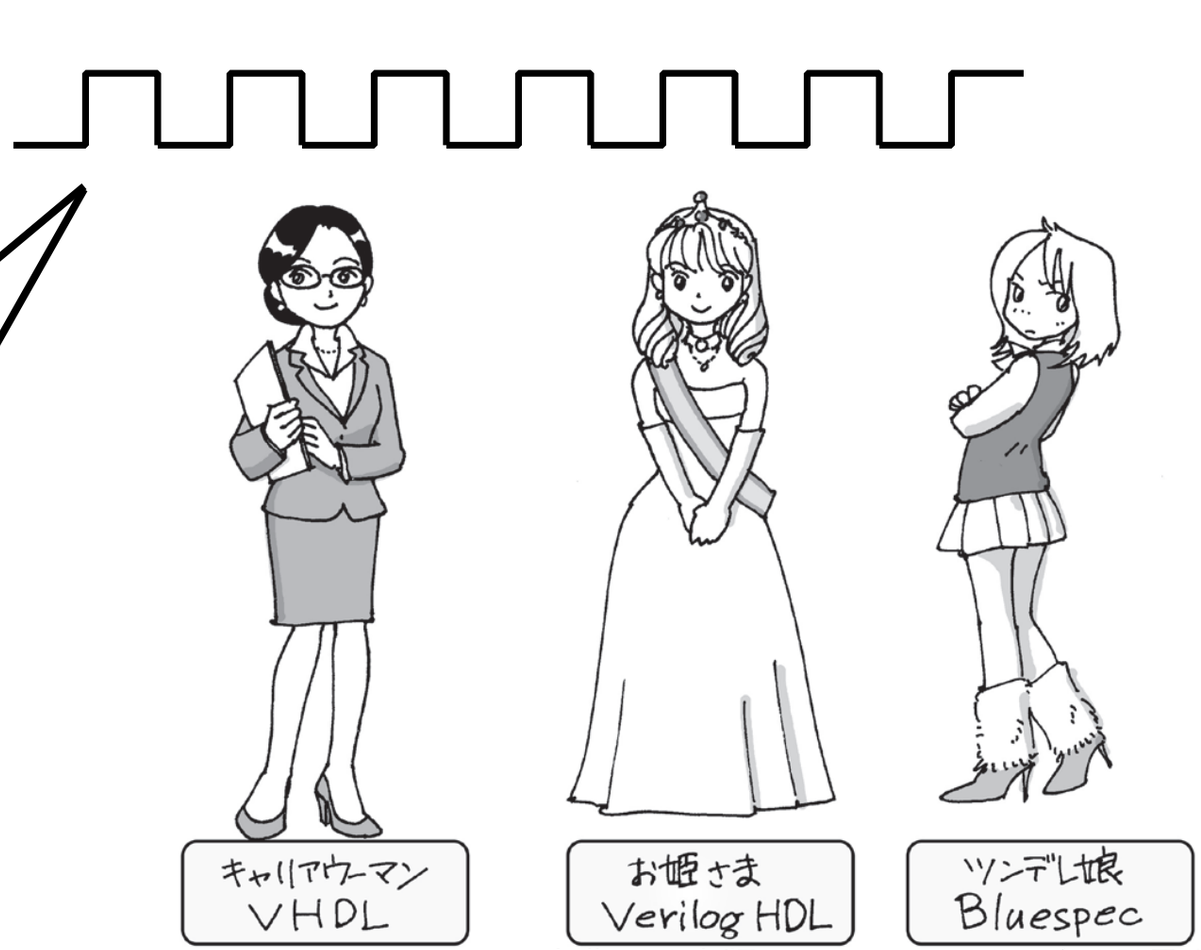
[1)]電気通信大学大学院情報システム学研究科
[2)]株式会社イーツリーズ・ジャパン

FPGAとは？

Field Programmable Gate Array

- 論理回路・データパスを自由に作り込める
- クロックレベルの同期と並列性の活用





出典: CQ出版　Interface 2011年2月号より

HDL(RTL記述)によるFPGA開発

hoge.vhd

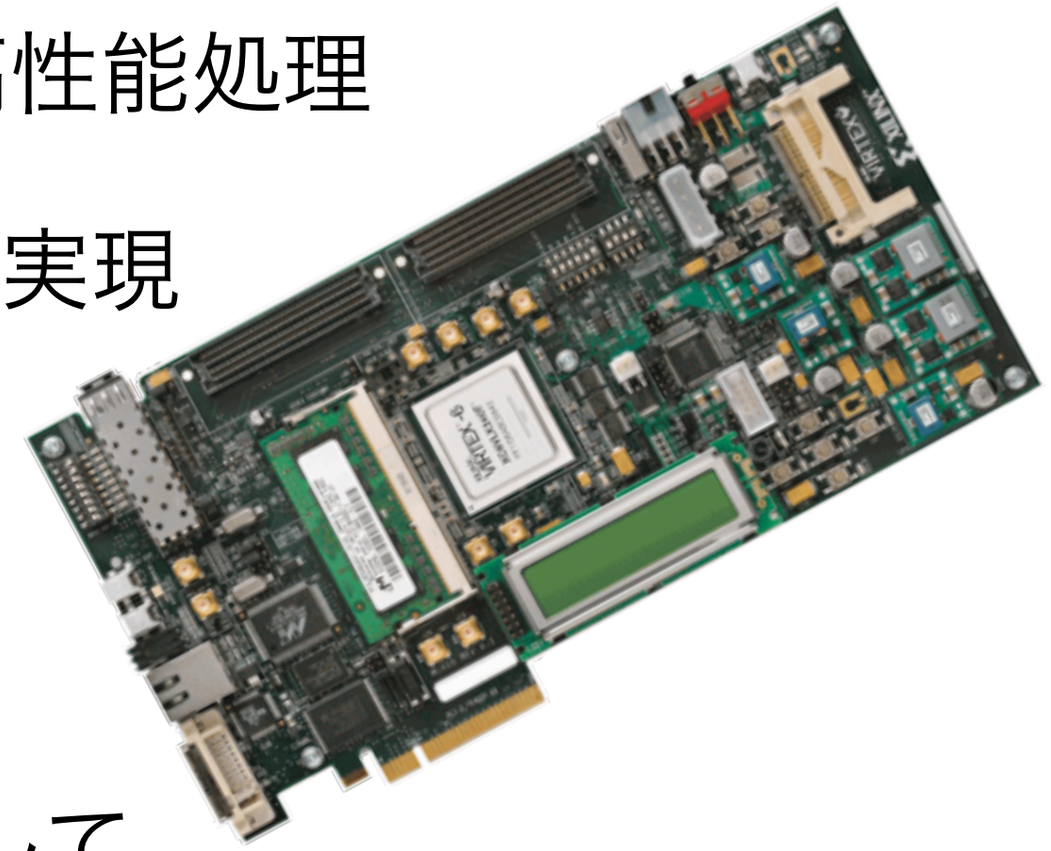
<-> [X] 0+ hoge.vhd

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.ALL;
use ieee.std_logic_unsigned.ALL;

entity hoge is

  port (
    clk   : in  std_logic;
    reset : in  std_logic;
    q     : out std_logic
    );

end hoge;

architecture RTL of hoge is

  signal counter : std_logic_vector(24 downto 0);

begin  -- RTL

  q <= counter(24);

  process (clk)
  begin  -- process
    if clk'event and clk = '1' then  -- rising clock edge
      if reset = '1' then
        counter <= counter + 1;
      end if;
    end if;
  end process;

end RTL;
```

-U:--- hoge.vhd All (1,0) [0] (VHDL/es)--2:18PM 0.35----------

FPGAの利用シーン

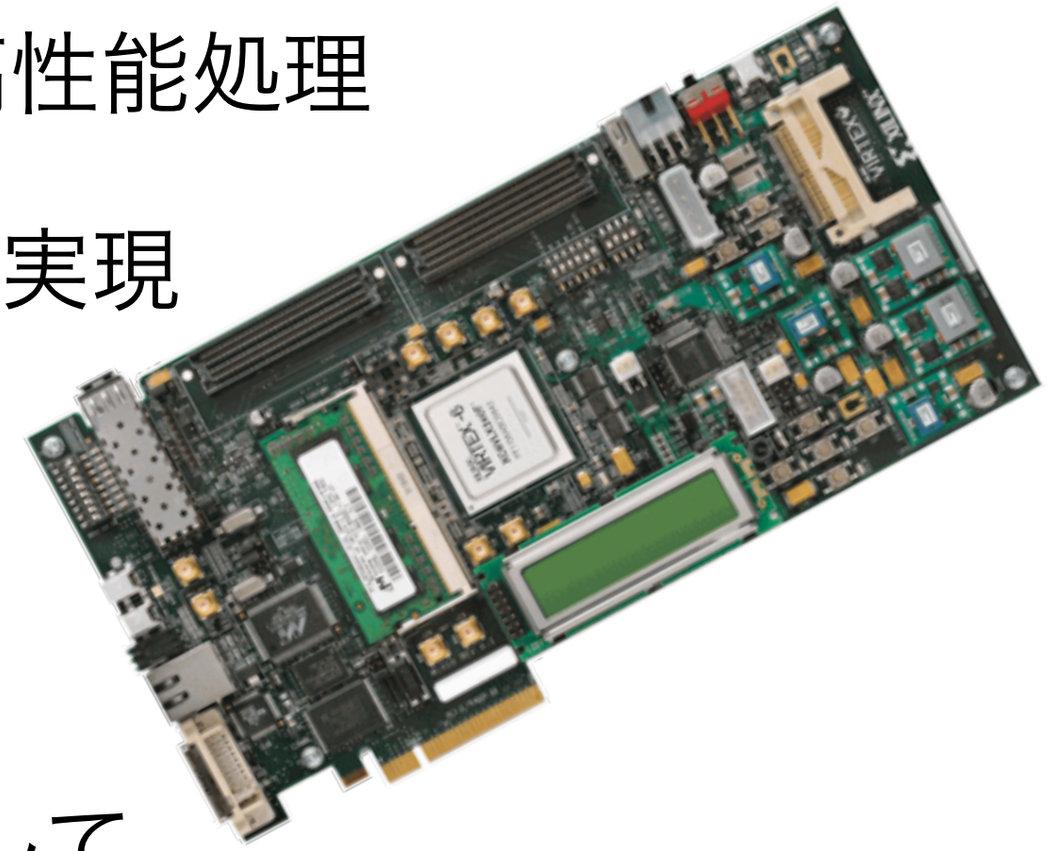
- 独自の回路を実現できるハードウェア
- 特定の処理を低消費電力で高性能処理
- デバイスに近い処理を簡単に実現
  - 自由なI/Oポートの定義



- ASIC開発のプロトタイプとして
- 特定用途向け少数生産の製品として

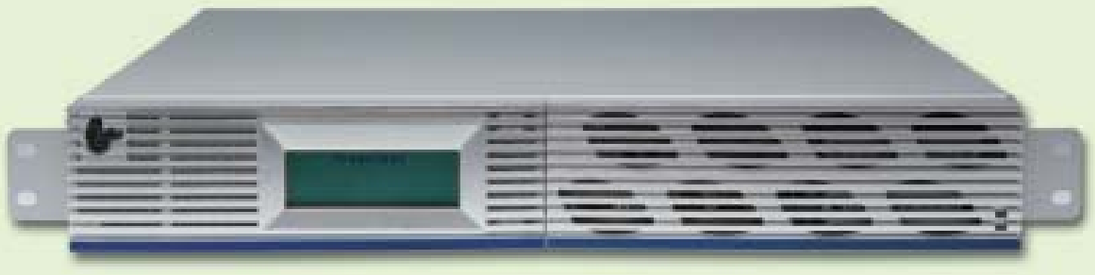
FPGAの利用シーン

- 独自の回路を実現できるハードウェア
- 特定の処理を低消費電力で高性能処理
- デバイスに近い処理を簡単に実現
  - 自由なI/Oポートの定義



- ASIC開発のプロトタイプとして
- 特定用途向け少数生産の製品として

●表面



●裏面

●基板

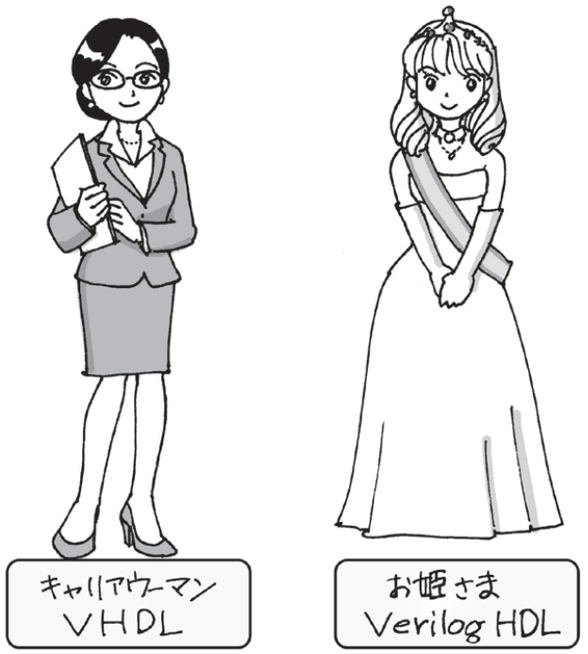
L2からL7までFPGAで処理

trees.Japan®

HDLによる開発のメリット/デメリット

メリット

- ロジックをテキストで設計できる
- クロックレベルのデータ制御
- 細粒度の並列性の活用



出典: CQ出版　Interface 2011年2月号

デメリット

- ソフトウェアとのプログラミングモデルの差
- デバッグ/動作検証が難しい
- アルゴリズムの記述では記述が煩雑

HDL設計の面倒さを解決するアプローチ

- (1) FPGAを使うのを諦める
  - 外付けのCPU/コントローラを使う
  - ソフトコアを利用する
- (2) 高位合成言語による設計

HDL設計の面倒さを解決するアプローチ

- (1) FPGAを使うのを諦める
  - 外付けのCPU/コントローラを使う
  - ソフトコアを利用する

環境整備/保守が面倒

- (2) 高位合成言語による設計

HDL設計の面倒さを解決するアプローチ

- (1) FPGAを使うのを諦める
  - 外付けのCPU/コントローラを使う
  - ソフトコアを利用する

- (2) 高位合成言語による設計

高位合成言語

タイプ1: 既存の言語を活用したHDL

既存の言語に文法，型を導入．親しんだ文法，抽象表現でのHDL設計を実現.

- SystemC，ImpulseC，Handel-C，Kiwi (C/C++/C#)
- JHDL，MaxCompiler，Lime (Java)
- PHDL (Python)，RHDL (Ruby)
- Lava (Haskell), DSL-Based (ML)

タイプ2: C言語をそのままHWに

Cで書かれたソフトウェアから，自動的にハードウェアを合成

- CyberWorkBench，LegUp

タイプ3: 抽象度の高い新しいHDL

- Bluespec System Verilog

高位合成言語に何を求めるか

- 記述コストの軽減
  - 高抽象度の表現方法を利用したい
  - 言語習得にコストをかけたくない
- 動作検証/デバッグ効率の向上
  - 短時間で検証をしたい
  - 見通しよく，手軽なデバッグをしたい
- FPGAのパフォーマンスの活用
  - 疎粒度，細粒度の並列性を活用したい
  - IPコア，内部機能ユニットを活用したい

既存の高位合成言語に残る不満

タイプ1: 既存の言語を利用したHDL

既存の言語に文法，型を導入．親しんだ文法，抽象表現でのHDL設計を実現.

- ハードウェアの知識(妙なテクニック)が必要
- デバッグにはRTL/TLMでのシミュレーションが必要
  ＝時間がかかる

タイプ2: C言語をそのままHWに

Cで書かれたソフトウェアから，自動的にハードウェアを合成

- FPGAを活用するための並列性の記述手段がない

タイプ3: 抽象度の高い新しいHDL

- 新しい言語習得のコストが大きい

JavaRock

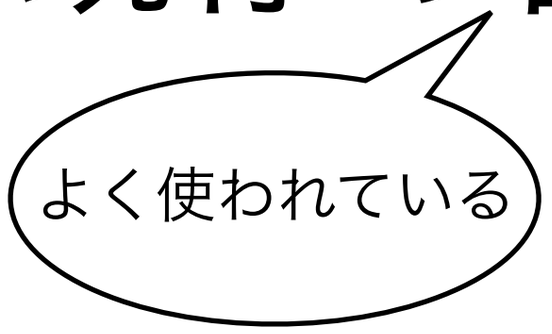
なんでJava？

Javaが好きだから

なんでJava？

=

並列性の記述能力をもつ既存の言語，だから



高位合成言語としてみたJava

- クラスによるオブジェクト指向設計
  ←HWのモジュール設計との親和性が高そう
- Threadやwait-notifyの仕組みがある
  ←言語仕様内で並列性の記述ができそう
- 明示的なポインタを扱う必要がない
  ←言語の想定するメモリ構造から自由になれそう
- 動的な振る舞いがたくさんある
  ←HW化するのは厄介そう

JavaRockの設計方針

- Javaプログラムをそのまま HW化する
  - 追加構文，データ型は導入しない
  - 記述に制限は加える
- プログラムカウンタをステートマシンに置換
  - １文１ステート
  - forやwhileに合わせたステートマシン生成
- メソッド呼び出し相当のHDLコード生成

JavaRockの設計方針

- “HDLで書けることをJavaで書けるようにする”ではない
- “Javaで書けることを全部HDLにする”ではない

Java(のサブセット）を実行可能なHDLコード規約

or

フェッチのないJavaプロセッサの生成

コード例

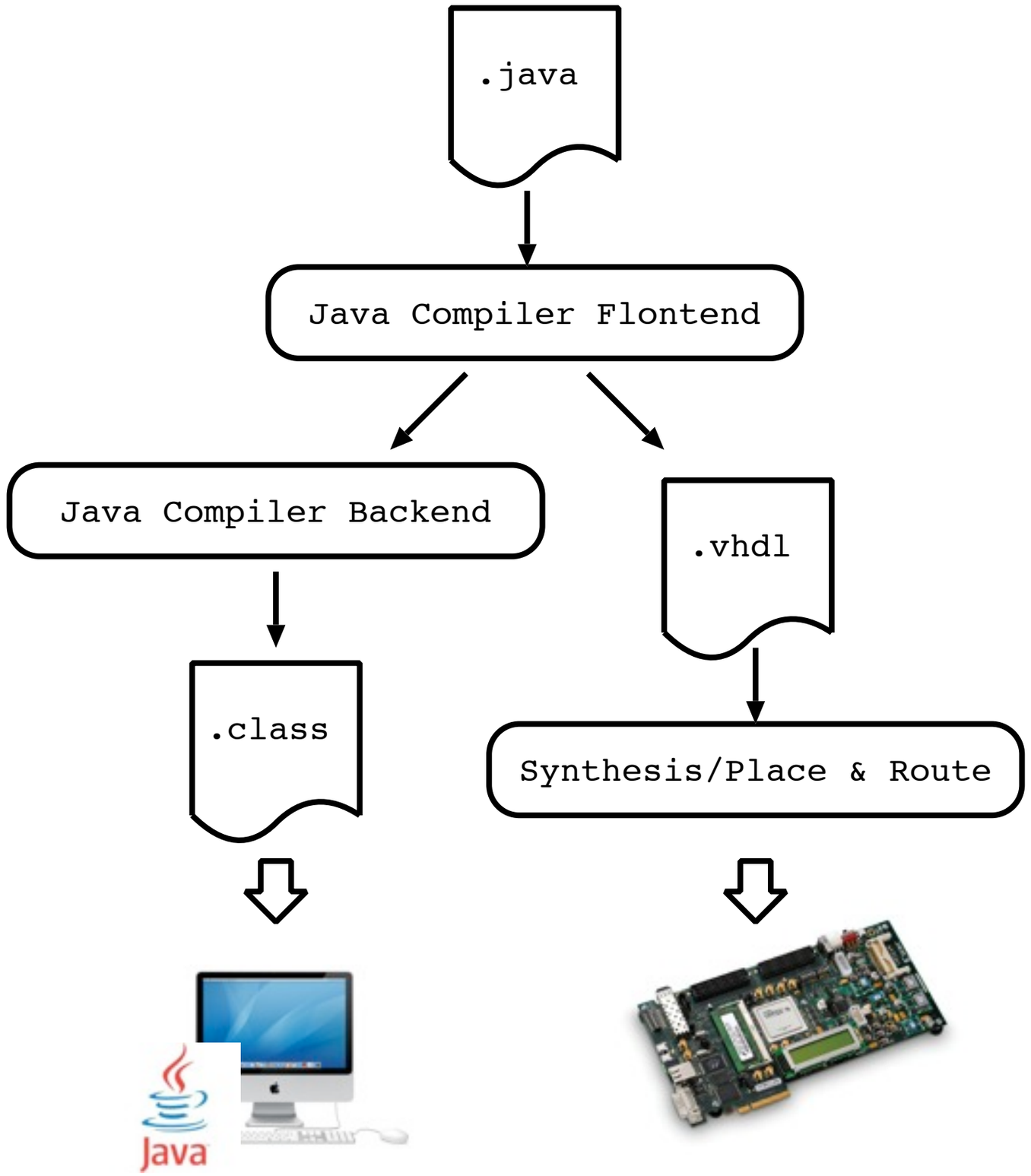
Hoge.java

<-> [X] 0+ Hoge.java

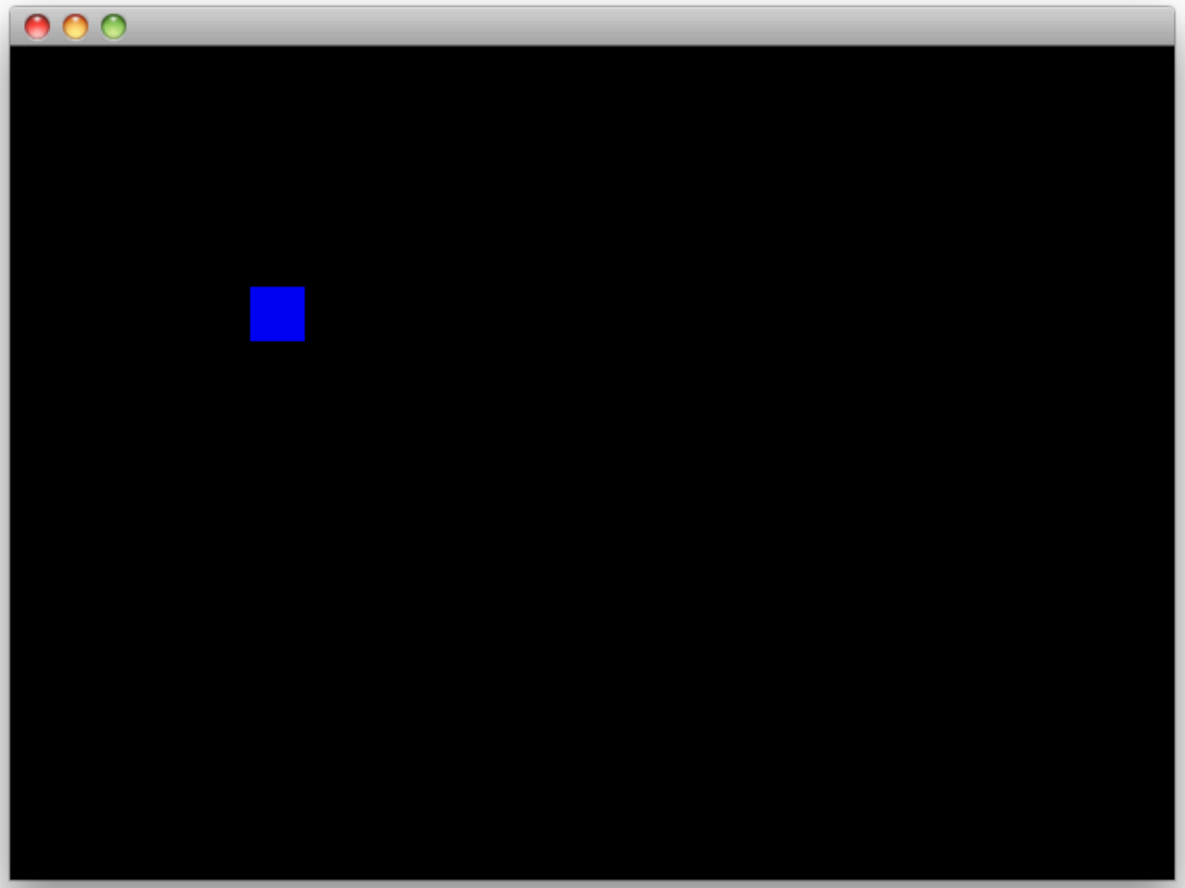
```
public class Hoge extends Thread{

  private int counter;

  public boolean isFlag(){
    if(counter > 1000000){
      return true;
    }else{
      return false;
    }
  }

  public void run(){
    while(true){
      counter++;
    }
  }

}
```

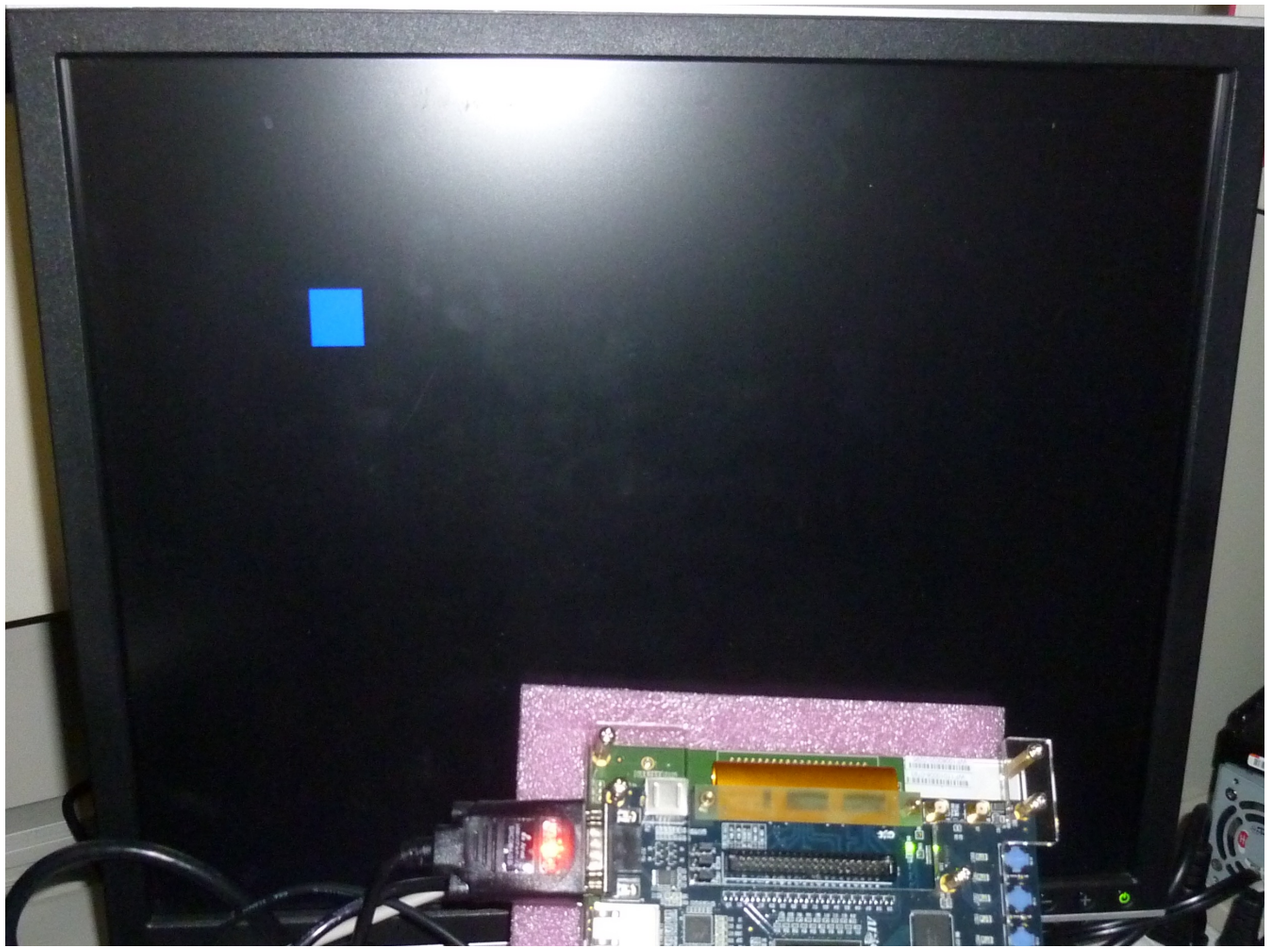
-U:--- Hoge.java All (1,0) [0] (Java/l Abbrev)--2:55PM 0.39----------

コンパイルフロー


.java
Java Compiler Flontend
Java Compiler Backend
.vhdl
.class
Synthesis/Place & Route
Java

ソフトウェアとハードウェアで同じ動作





どうやって

JavaをHDLにするか

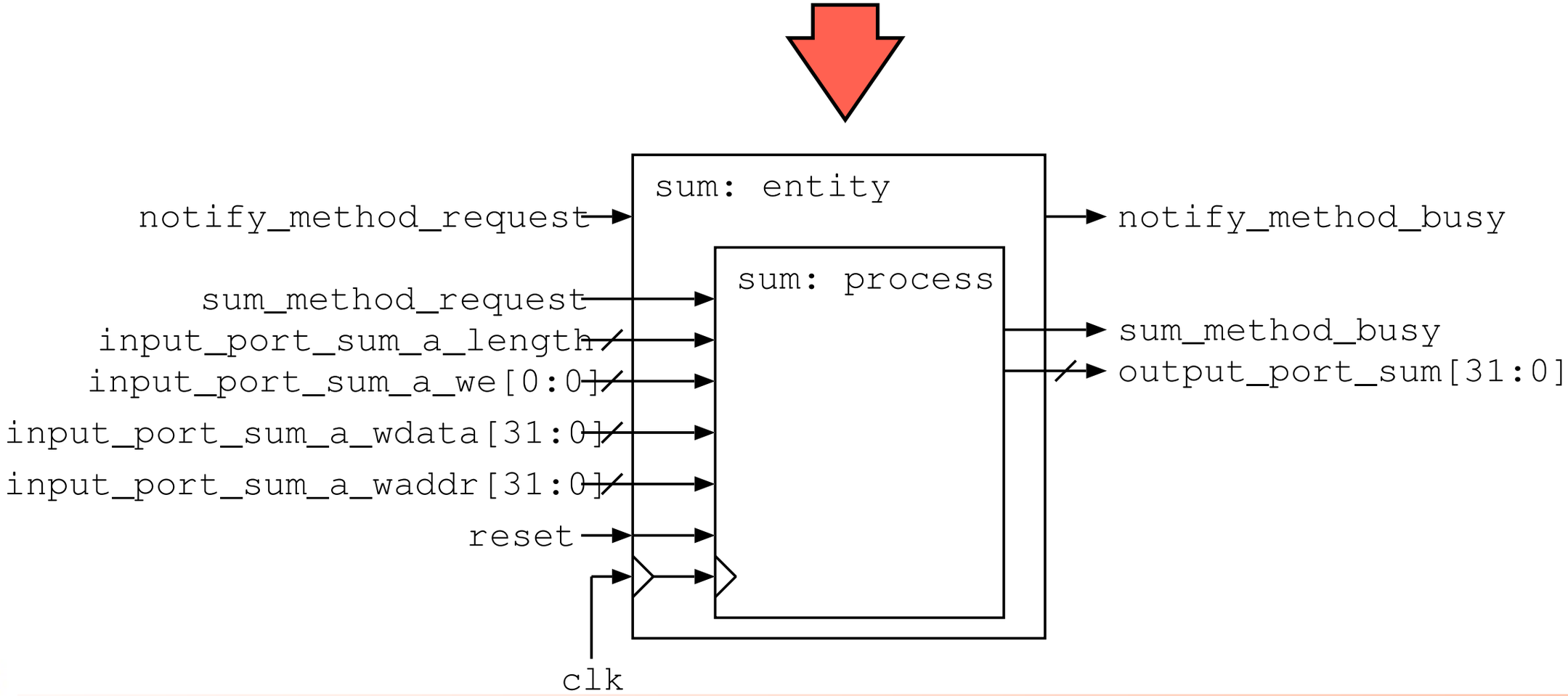
構文木をJavaからVHDLに変換

- class
  - var decl
  - method decl
    - var decl
    - block
      - expr statement
        - expr
          - ident
          - binary expr
          - unary expr
          - method invocation
          - assign
          - parens
            - expr
          - array access
      - var decl
      - block
      - while
      - for
      - if
      - continue
      - block
      - return

- *entity*
  - *architecture*
    - *signal decl*
    - *process*
      - state machine
        - expr statement
          - expr
            - *ident*
            - binary expr
            - unary expr
            - method invocation
            - *parens*
              - expr
            - array access
        - state machine
        - while
        - for
        - if
        - continue
        - return
        - assign
        - block

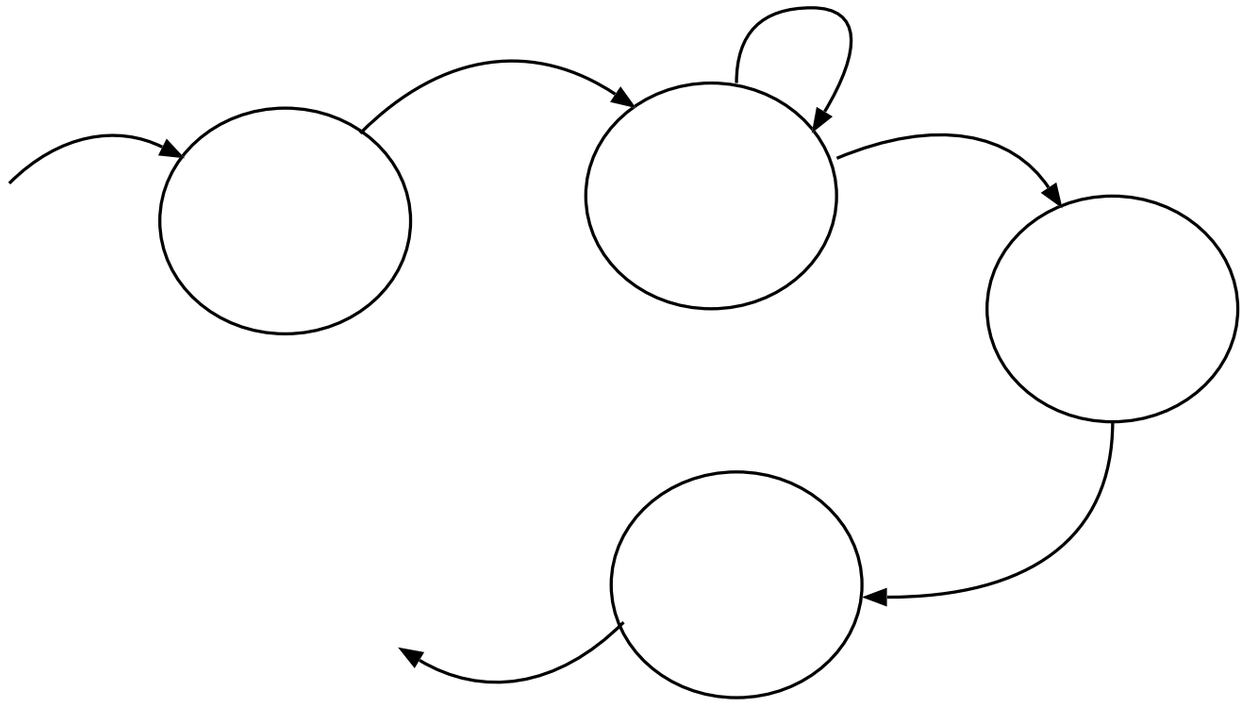
生成する基本モジュール

```
1: public class sum{
2:  public int sum(int[] a){
3:    int sum = 0;
4:    for(int i = 0; i < a.length; i++){
5:      sum += a[i];
6:    }
7:    return sum;
8:  }
9: }
```



コード生成規約 基本的なストラテジ

- Javaの1文を1ステートとするステートマシン
- ブロック文はさらにステートマシンを分割
  - if, while, for, {}
- 配列アクセスはBRAMアクセスに分解



```
int sum = 0;
```

```
1: case conv_integer(sum_method_state) is
2:   ...
3:  when 2 =>
4:    sum_0 <= conv_std_logic_vector(0, 31-0+1);
5:    sum_method_state <= sum_method_state + 1;
6:  when 3 =>
7:   ...
```

```
for(int i = 0; i < a.length; i++){ }
```

```
 1: case conv_integer(sum_method_state) is
 2:  ...
 3:   when 3 =>
 4:     case conv_integer(state_counter_sum_1) is
 5:       when 0 =>
 6:         i_1 <= conv_std_logic_vector(0, 31-0+1);
 7:         state_counter_sum_1 <= state_counter_sum_1 + 1;
 8:       when 1 =>
 9:         if (conv_integer(i_1) < input_port_sum_a_length) then
10:           state_counter_sum_1 <= state_counter_sum_1 + 1;
11:         else
12:           sum_method_state <= sum_method_state + 1;
13:           state_counter_sum_1 <= (others => '0');
14:         end if;
15:       when 2 =>
16:            ...
17:       when 3 =>
18:           i_1 <= conv_std_logic_vector(conv_integer(i_1 + 1), 31-0+1);
19:           state_counter_sum_1 <= conv_std_logic_vector(1, 32);
20:       when others => state_counter_sum_1 <= (others => '0');
21:     end case;
22:   when 4 =>
23:  ...
```

sum += a[i];

```
1: when 2 =>
2:   case conv_integer(state_counter_sum_2) is
3:     when 0 =>
4:       case conv_integer(array_index_operation_state_counter_3) is
5:         when 0 =>
6:           param_input_port_sum_a_raddr <=
                      conv_std_logic_vector(conv_integer(i_1), 11-1-0+1);
7:           array_index_operation_state_counter_3 <=
                                array_index_operation_state_counter_3 + 1;
8:         when 1 =>
9:           array_index_operation_state_counter_3 <= (others => '0');
10:          state_counter_sum_2 <= state_counter_sum_2 + 1;
11:        when others => array_index_operation_state_counter_3 <=
                                                        (others => '0');
12:      end case;
13:    when 1 =>
14:      sum_0 <= conv_std_logic_vector(
            conv_integer(sum_0 + param_input_port_sum_a_rdata), 31-0+1);
15:      state_counter_sum_1 <= state_counter_sum_1 + 1;
16:      state_counter_sum_2 <= (others => '0');
17:    when others => state_counter_sum_2 <= (others => '0');
18:  end case;
19: when 3 =>
20: ...
```

int sum(int[] a){ }

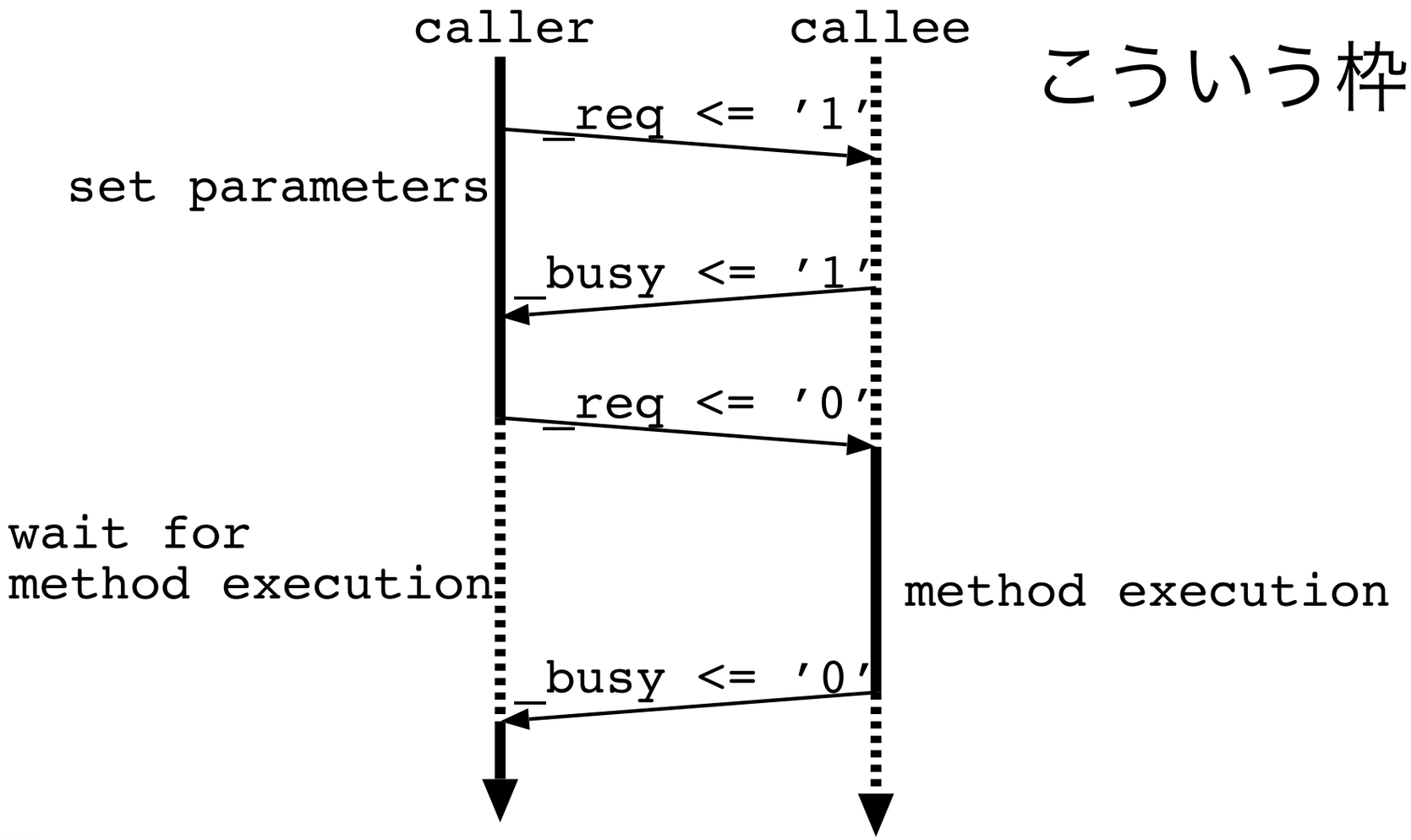
```
 1: case conv_integer(sum_method_state) is
 2:   when 0 =>
 3:     if(sum_method_request = '1') then
 4:       sum_method_busy <= '1';
 5:       sum_method_state <= sum_method_state + 1;
 6:     else
 7:       sum_method_busy <= '0';
 8:     end if;
 9:   when 1 =>
10:      if(sum_method_request = '0') then
11:        sum_method_state <= sum_method_state + 1;
12:      end if;
13:   when 2 =>
14:    ...
15:   when 4 =>
16:    output_port_sum <= conv_std_logic_vector(conv_integer(sum_0), 31-0+1);
17:    sum_method_state <= (others => '0');
15:   when 5 =>
16:     sum_method_busy <= '0';
17:     sum_method_state <= (others => '0');
18:   when others => sum_method_state <= (others => '0');
19: end case;
```

コード生成規約: メソッド呼び出し

Java

```
public class Hoge{
    // caller
    public int zero(){ return 0; }
```

```
    Hoge hoge = new Hoge();
...
        hoge.zero(); // callee
```



こういう枠にはめる.

JavaRock(r120)での制約

- インスタンスはファイナルでの生成のみ
  - スタティックなHWモジュールとして合成
- インスタンスの共有は不可
  - アービトレーションまだ機構を持ってない
- 配列はlong/int/short/char/byteのみ
- /とか%とか未サポート

どの程度の回路が
できるか？

リソース使用量/最高動作周波数

- XC6VLX240T-1を対象に合成
  (#. Reg: 301,440,　#. LUT: 150,720)

	レジスタ数	LUT数	BlockRAM	Max Freq.
sieve	106	246	16	269MHz
sort	268	408	1	230MHz
参考 SMM	440	690	-	-

SMM: Simple Microblaze Microcontroller

処理性能

- XC6VLX240T-1を対象に合成．200MHz
- Core i7 2.93GHz，メモリ16GBの計算機と比較

	FPGA	Core i7
sieve	7.3m秒	10.2m秒
sort	9.8m秒	7.6m秒

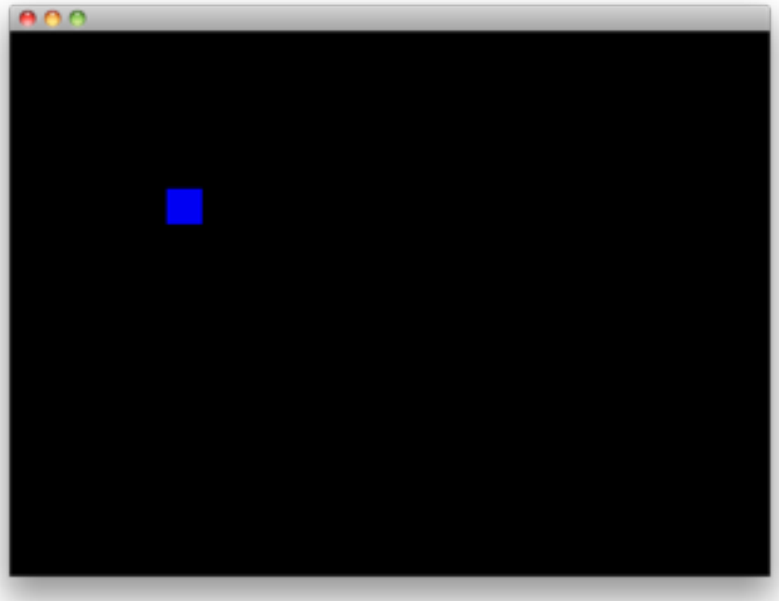
どのように使えるか

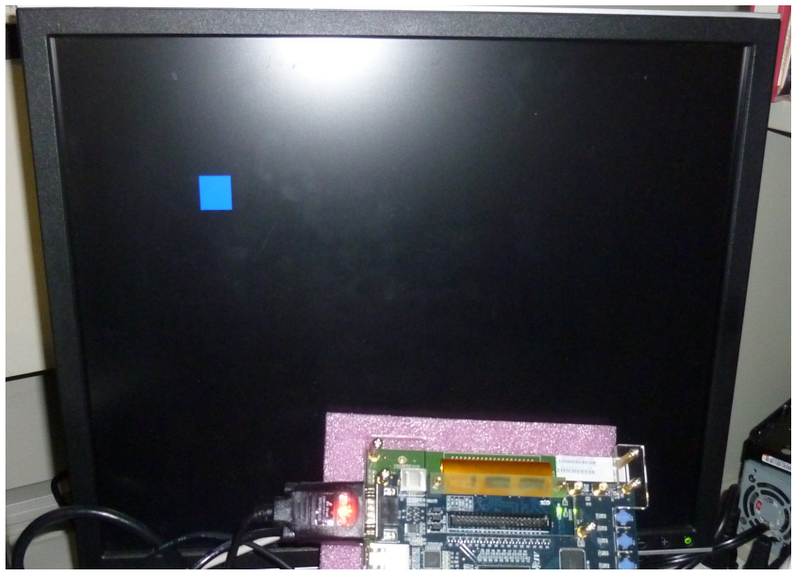
ケーススタディ

- 画像処理プログラム
- connect6

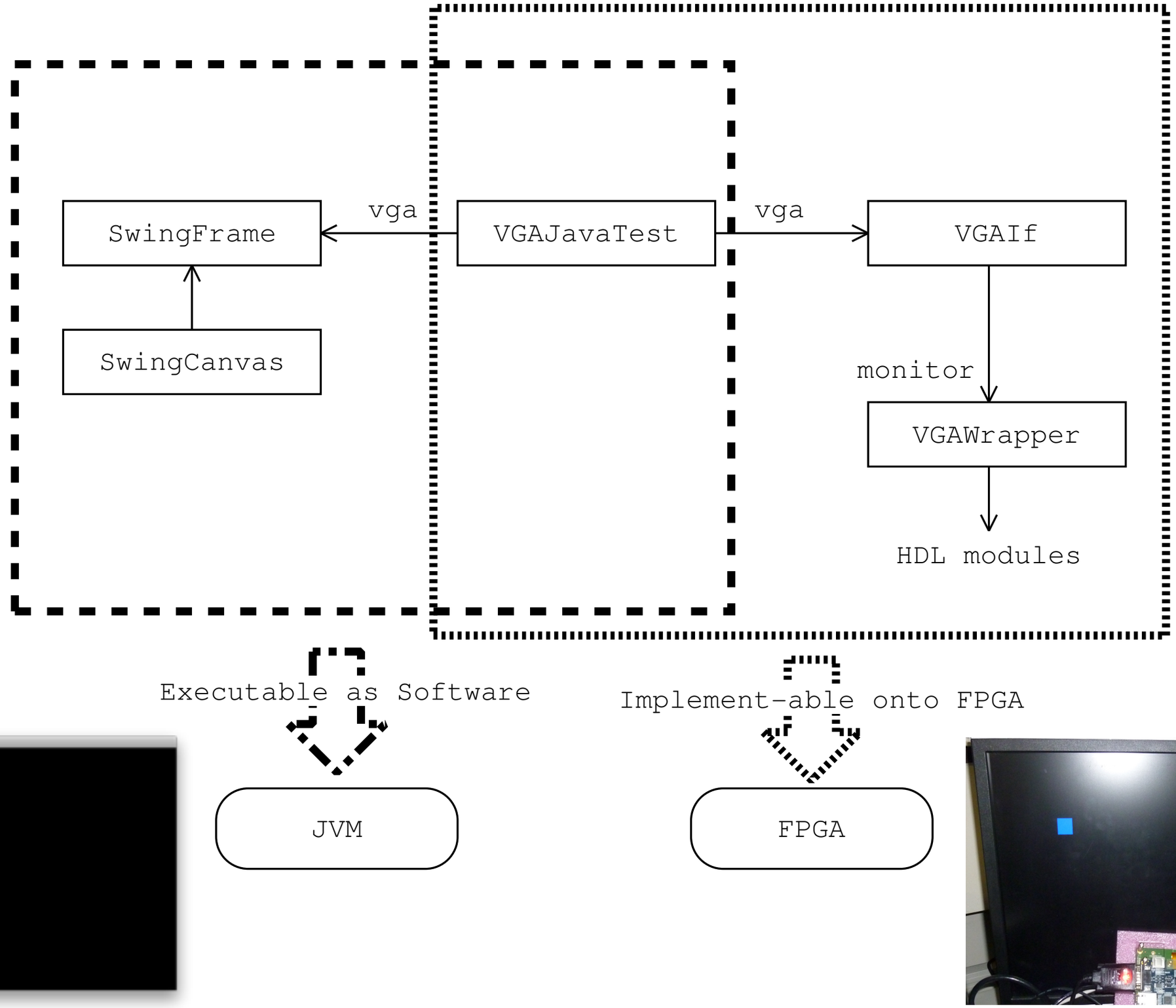
ケーススタディ1: 画像表示プログラム

```
public class VGAJavaTest implements Runnable{

  VGAIf vga = new VGAIf();

    vga.clear();
    for(int k = 0; k < 8; k++){
      c = color[k];
      for(int i = 100; i < 130; i++){
        for(int j = 100; j < 130; j++){
          vga.pset(i+(k<<4), j+(k<<4), c);
        }
      }
      for(int i = 0; i < 10000; i++){
        for(int j = 0; j < 1000; j++){
          ;
        }
      }
      for(int i = 100; i < 130; i++){
        for(int j = 100; j < 130; j++){
          vga.pset(i+(k<<4), j+(k<<4), (char)0x0000);
        }
      }
    }
  }


}
```





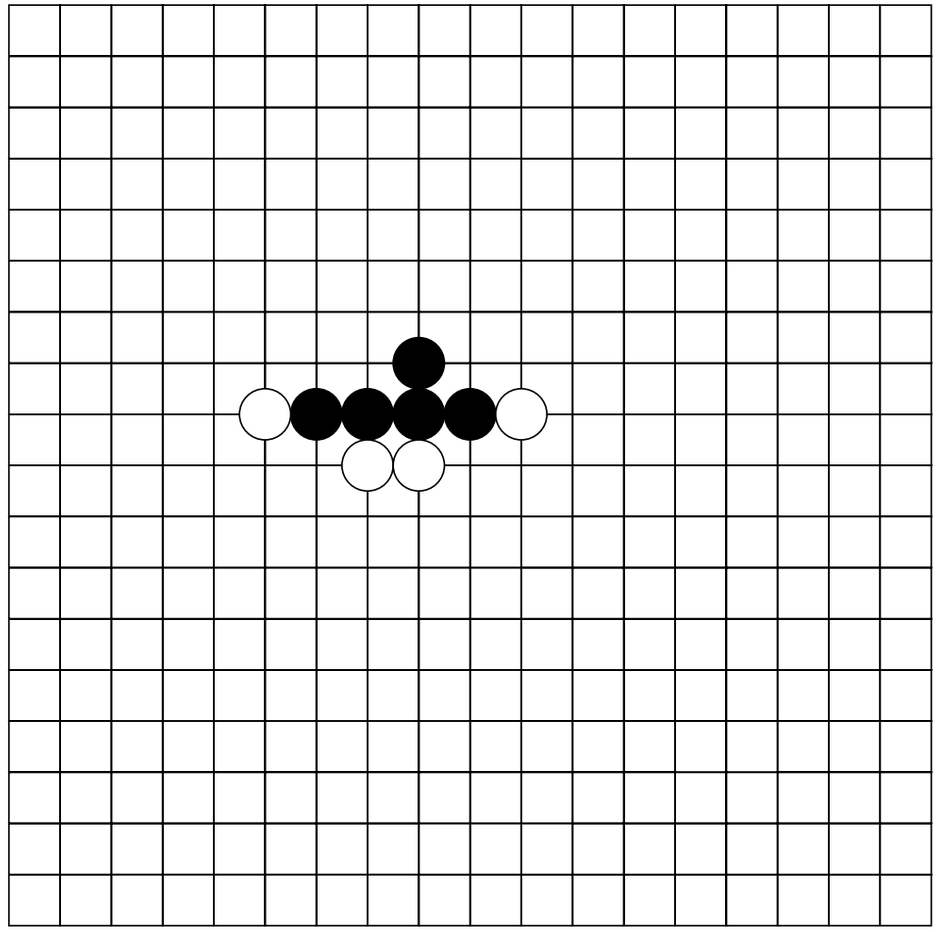
ケーススタディ1: 画像表示プログラム



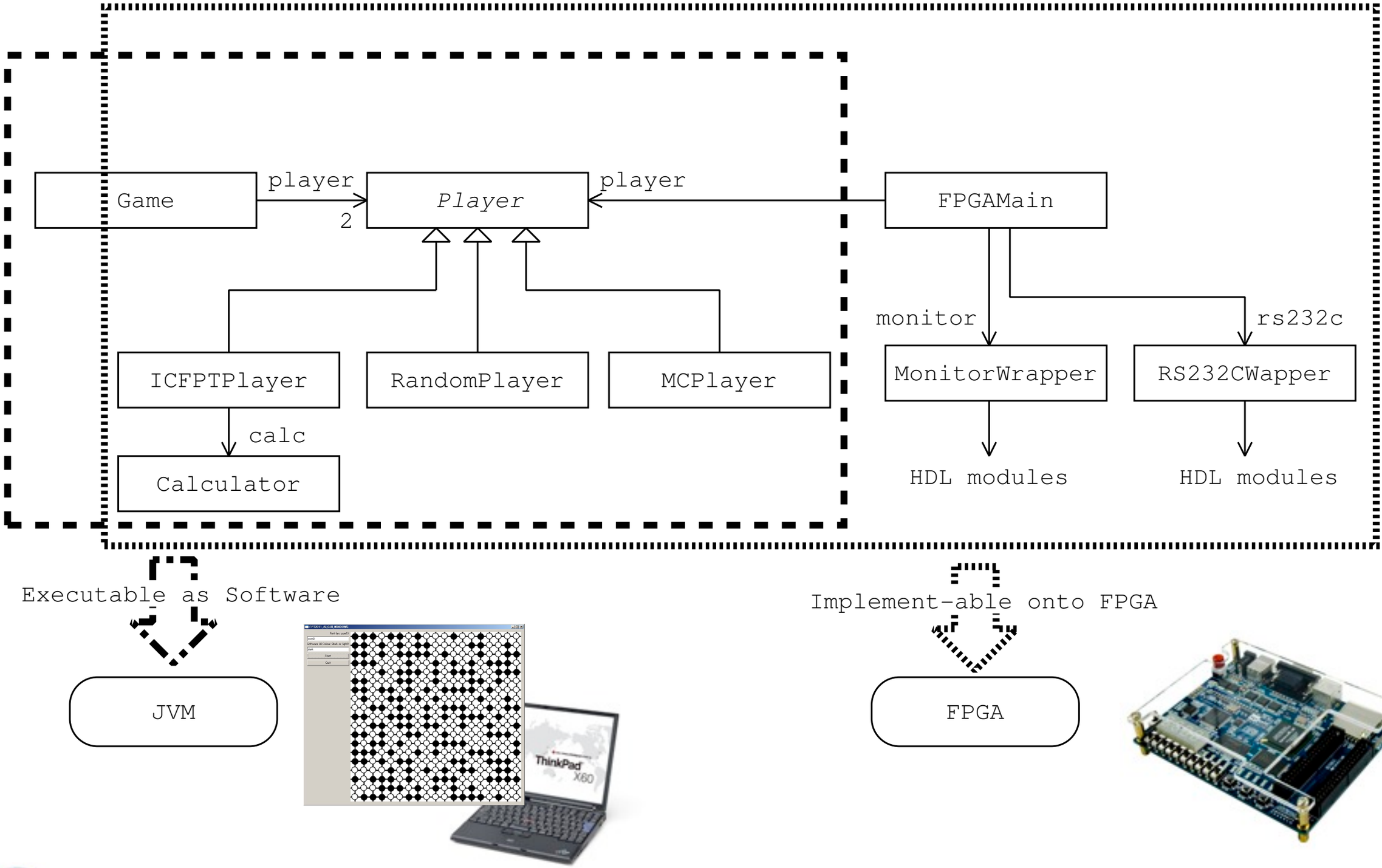
ケーススタディ2: connect6

- 6つ石を並べれば勝ち
- 先手の黒は最初に１個石を置く.
- 以降, 白と黒が交互に２個ずつ石を置く.

- FPTデザインコンテスト
- HEART2012



ケーススタディ2: connect6



FPGAを活用するために

FPGAを活用するために

- 並列化

- HDLモジュールの利用
  - JavaRock Hardware Interface(JRHI)
  - JavaRock HDL

並列化手法について

- 演算レベルの並列性

  →基本ブロック内の自動並列化

- タスク・データレベルの並列性

  →JavaのスレッドをHWにマッピング

- パイプライン並列性

  →wait-notifyで記述

JavaのスレッドのHWへのマッピング

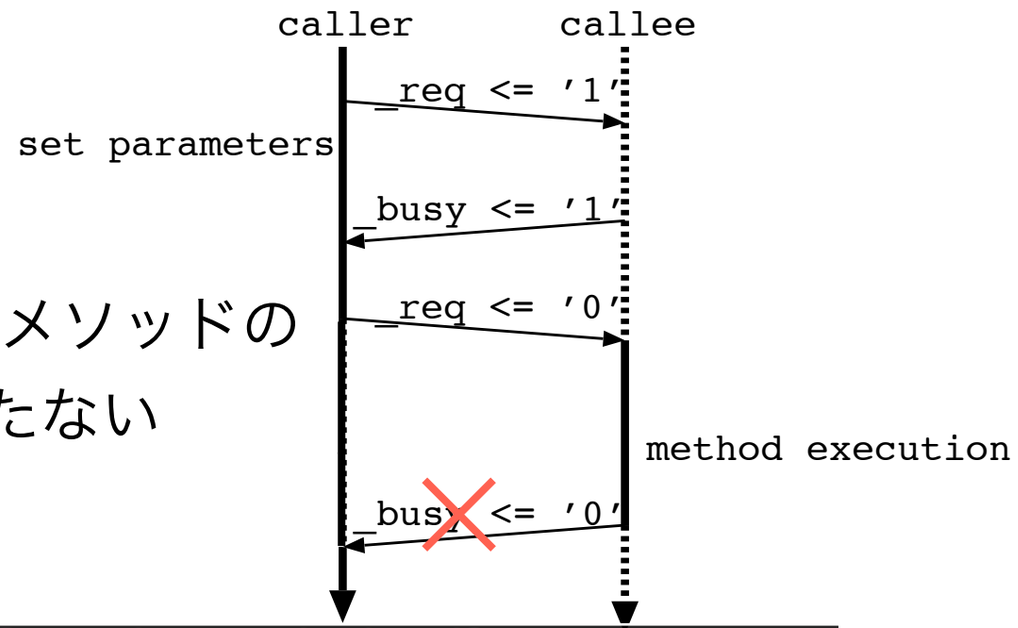
```
led obj0 = new led();
echo obj1 = new echo();
sc1602_test obj2 = new sc1602_test();
```

```
obj0.start();
obj1.start();
obj2.start();
```

スレッドの生成
&
実行

```
public class led extends Thread{
  counter c = new counter();
  boolean flag;

  public void run() {
    while(true){
      c.up();
      int v = c.read();
      if (v == 1000000) {
        c.clear();
        flag = !flag;
      }
    }
  }
```

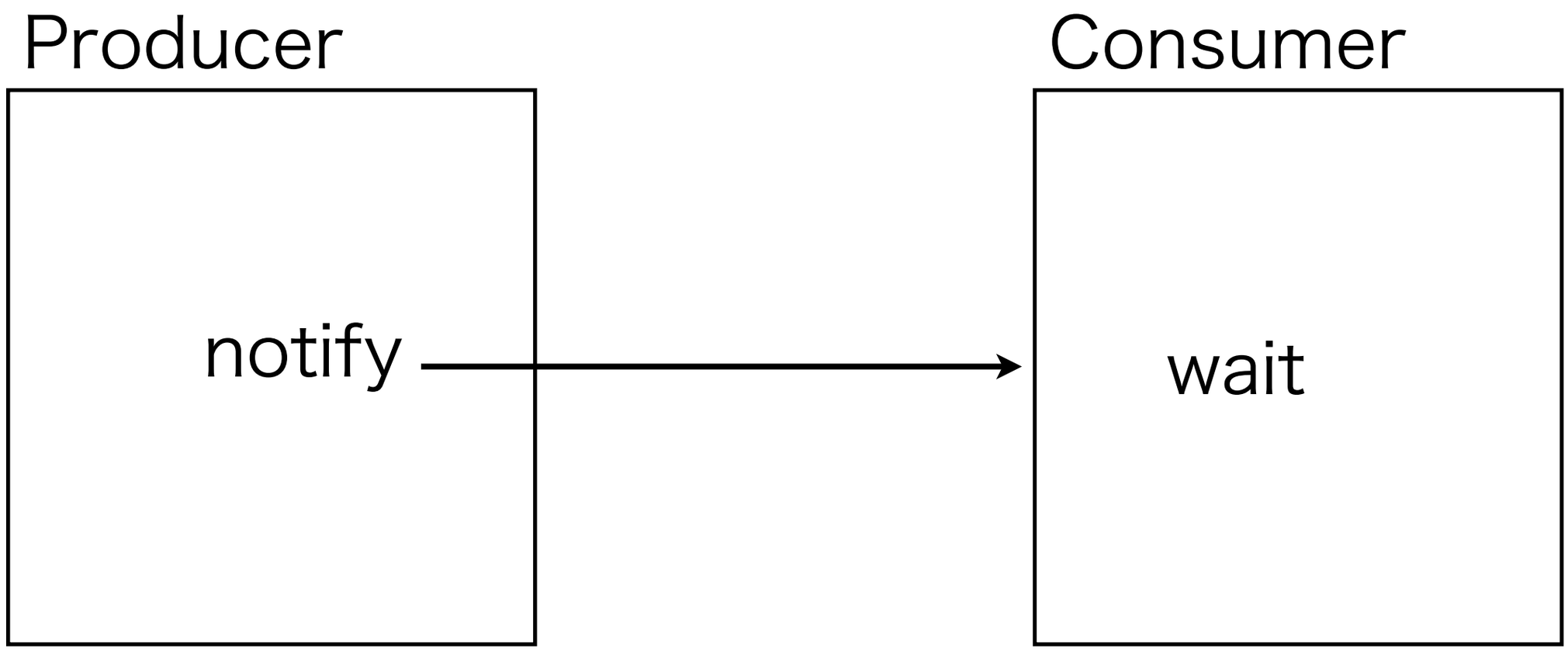
startではメソッドの
終了を待たない




```
public class echo extends Thread{
  rs232c obj = new rs232c();
  byte[] data = new byte[128];

  public void run(){
    while(true){
      obj.write((byte)'>');
      int i = 0;
      byte c = 0;
      boolean flag = true;
      while(true){
        c = obj.read();
        if(c == (byte)'\n' || c == (byte)'\r'){
          break;
        }else{
          data[i] = c;
          i++;
        }
      }
      for(int j = 0; j < i; j++){
        c = data[j];
```

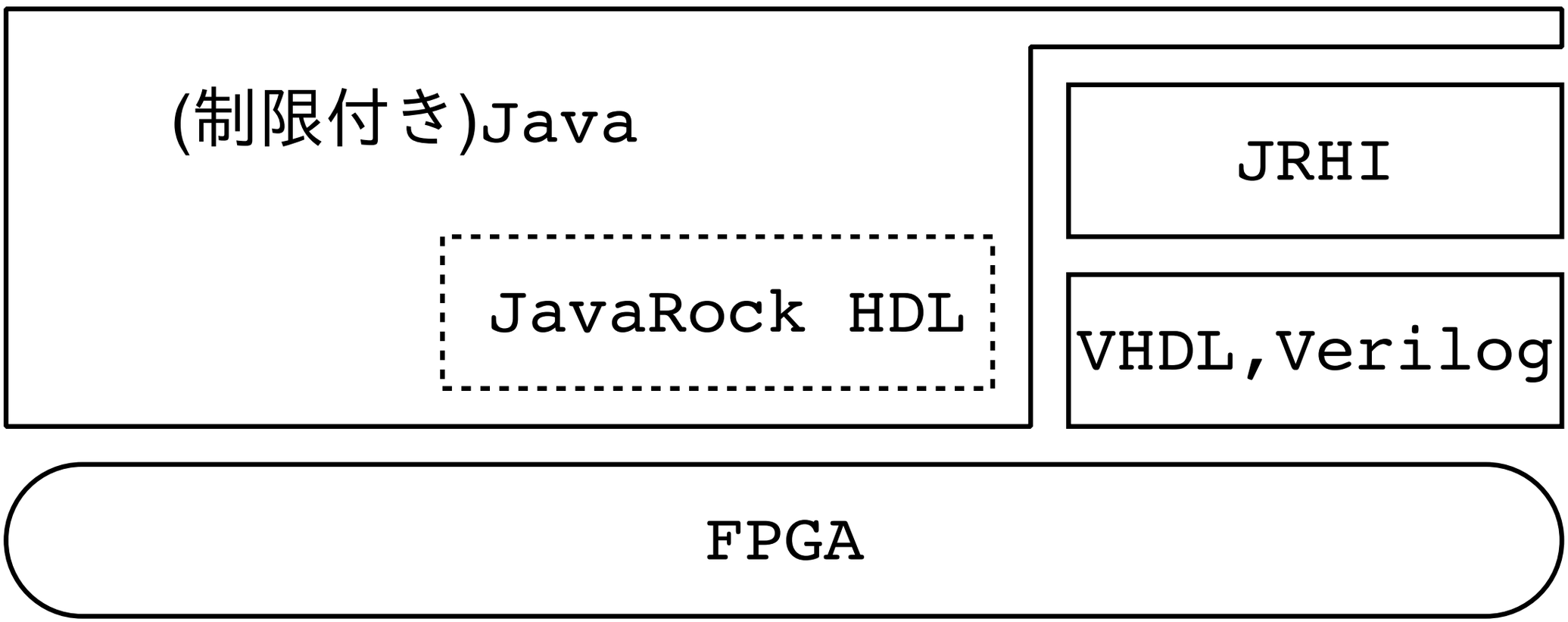
パイプライン並列化

- Producer-Consumerパタンで設計



HDLモジュールの活用

- IPコア，手書きHDLモジュールの活用
- ハードウェアを意識したJavaコード記述支援



JavaRock Hardware Interface

```
entity sc1602_wrapper is
  port (
    clk     : in  std_logic;
    pLCD_RS  : out std_logic;
    pLCD_E   : out std_logic;
    pLCD_DB  : out std_logic_vector(3 downto 0);
    pLCD_RW  : out std_logic;

    pReq    : in  std_logic;
    pBusy   : out std_logic;
    pWrData : in std_logic_vector(7 downto 0);
    pWrAddr : in std_logic_vector(31 downto 0);
    pWrWe   : in std_logic;

    reset  : in  std_logic
  );
end sc1602_wrapper;
```

使いたいモジュール

JRHI

```
public class SC1602Wrapper extends VHDLSimpleLibrary implements VHDLComponentIface{

  public boolean pReq;
  public boolean pBusy;
  public boolean pWrWe;
  public byte pWrData;
  public int pWrAddr;

  public SC1602Wrapper(String... args){
    super("sc1602_wrapper", args);
```

JavaRock HDL

- アノテーションによるJavaのHDL化

test.java

<-> [X] 0+ test.java

```
import net.wasamon.javarock.rt.*;

@javarockhdl
public class test{
  led obj0 = new led();
  echo obj1 = new echo();
  sc1602_test obj2 = new sc1602_test();
  //VGAJavaTest vga = new VGAJavaTest();

  @combination
  public boolean led(){
    return obj0.led();
  }

  @auto
  public void main(){
    obj0.start();
    obj1.start();
    obj2.start();
    //vga.start();
  }
}
```

--:--- test.java All (7,0) [0] SVN-128 [(Java/l Abbrev)]--8:09AM 0.49----------

アノテーション

name	description	target
javarockhdl	enables JavaRock HDL annotations in the class.	class
raw	converts conditional branches to HDL as is.	method
auto	executes the method automatically.	method
width	defines bit-width of the variable.	variable

まとめ

- JavaRockを作っています
  - Java”を”HDLにします
  - Javaのサブセットです
- それなりの回路が生成されます
  - 合成ツールでかなり最適化してもらえる
  - Core i7とコンパラブルな処理性能を達成

今後の課題

- 各種の最適化
- JavaをどこまでHWにできるのか追求
  - 動的なインスタンス生成のサポート
- 現在のHW化手法の妥当性についての考察

http://javarock.sourceforge.net/