Synthesijer を使った JavaによるFPGA開発のはじめ方

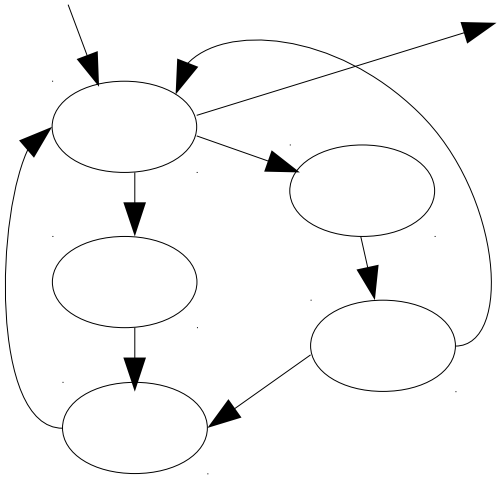
わさらぼ合同会社　三好 健文

2014.10.22

Synthesijer とは

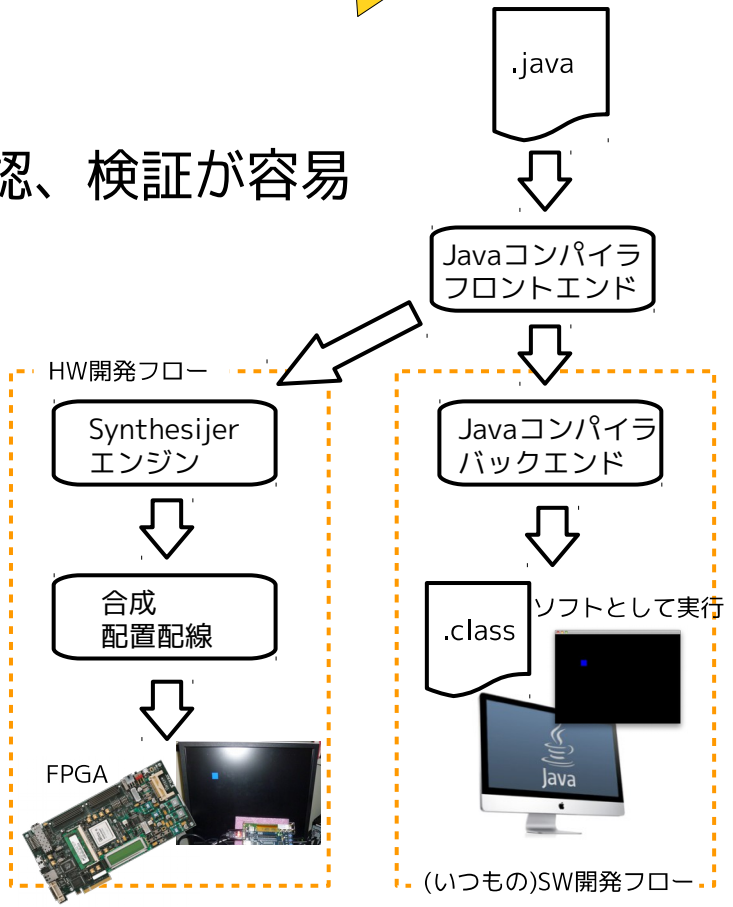
- JavaプログラムをFPGA上のハードウェアに変換
  - 複雑なアルゴリズムのハードウェア実装を楽に
  - オブクジェクト指向設計による再利用性の向上
- 特殊な記法，追加構文はない
  - ソフトウェアとして実行可能．動作の確認、検証が容易
  - 書けるプログラムに制限は加える
    (動的なnew，再帰は不可など)

Open-source



```
while(){
if(...){
  …
}else{
  …
  …
}
….
}
```

複雑な状態遷移も，Javaの制御構文を使って楽に設計できる



同じJavaプログラムをソフトウェアとしても
FPGA上のハードウェアとしても実行可能

Synthesijer 開発の経緯

- 2011年7月

  JavaRockの開発を開始

  JavaをVHDLに変換する．1文1ステート→簡単な並列化

- 2012(?)年〜2013年

  農工大中條研小池さんによるJavaRock-Thrashの開発

  種々の最適化の実装など

- 2014年

  Synthesijerとして名称をあらため開発を再スタート

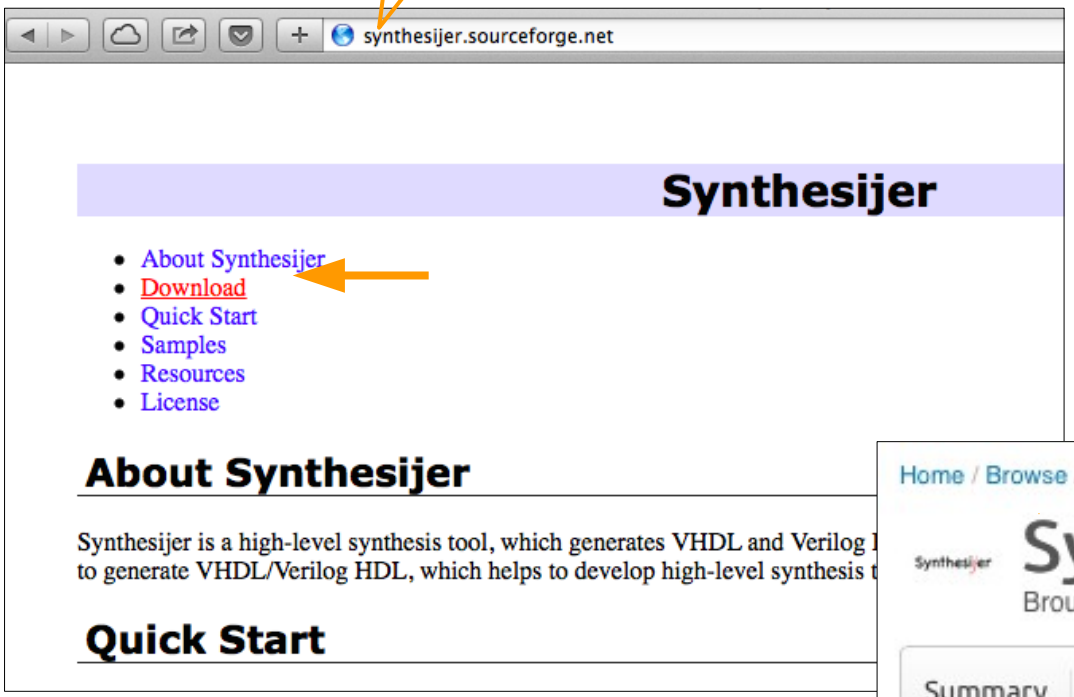
  JavaRock/JavaRock-Thrashの実装を通じた知見の活用

Synthesijer クイックスタート

クイックスタート 1/8

(1) バイナリパッケージをダウンロードします

http://synthesijer.sourceforge.net



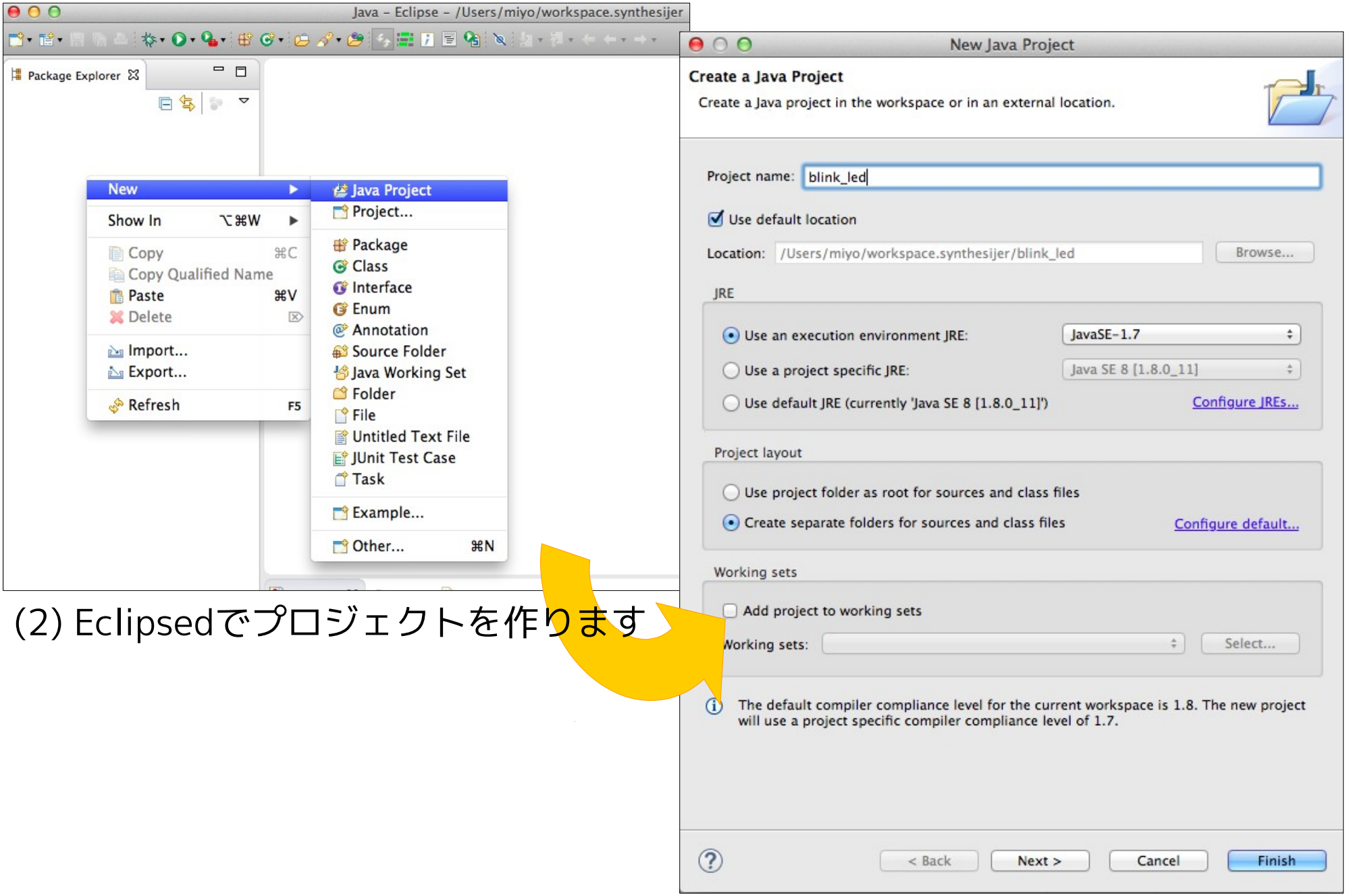
✔ JDK7が必要です.

✔ Eclipseを使って開発してみます.

ダウンロードページに遷移します

なるべく日付の新しいものをDLしてください

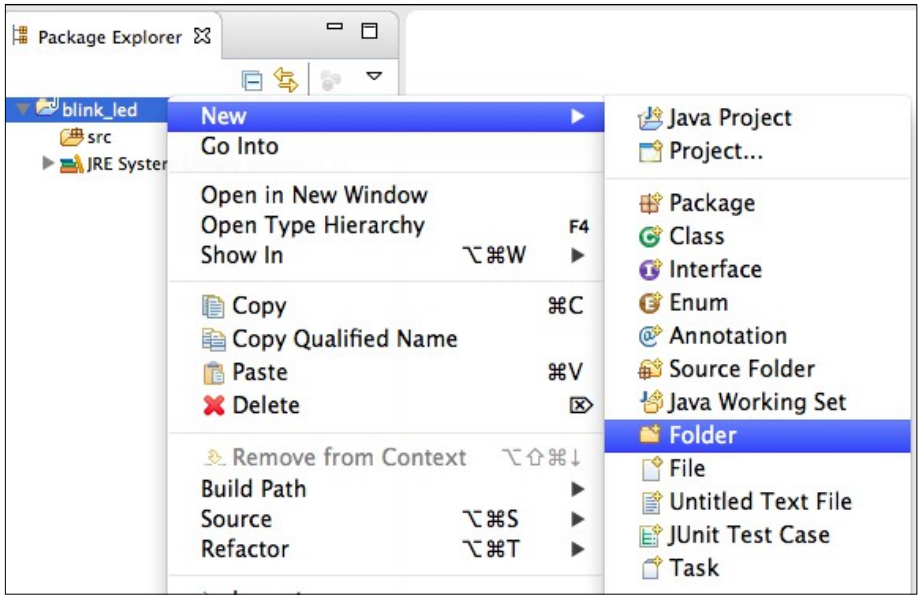
クイックスタート 2/8

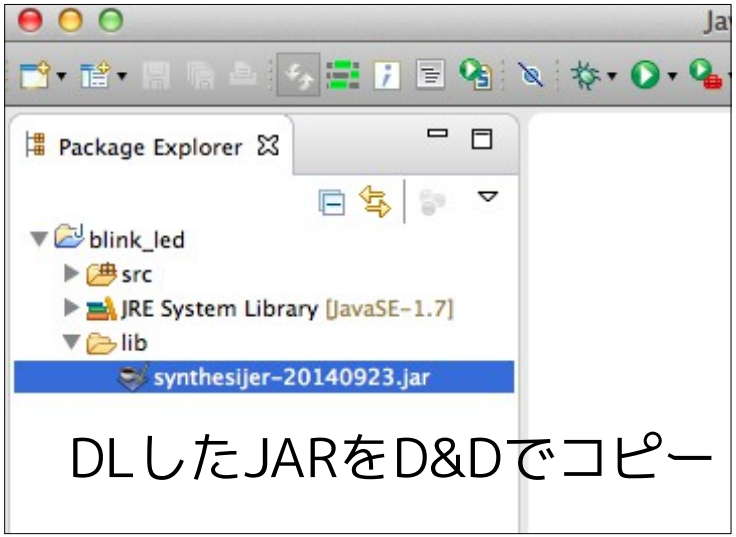


(2) Eclipsedでプロジェクトを作ります

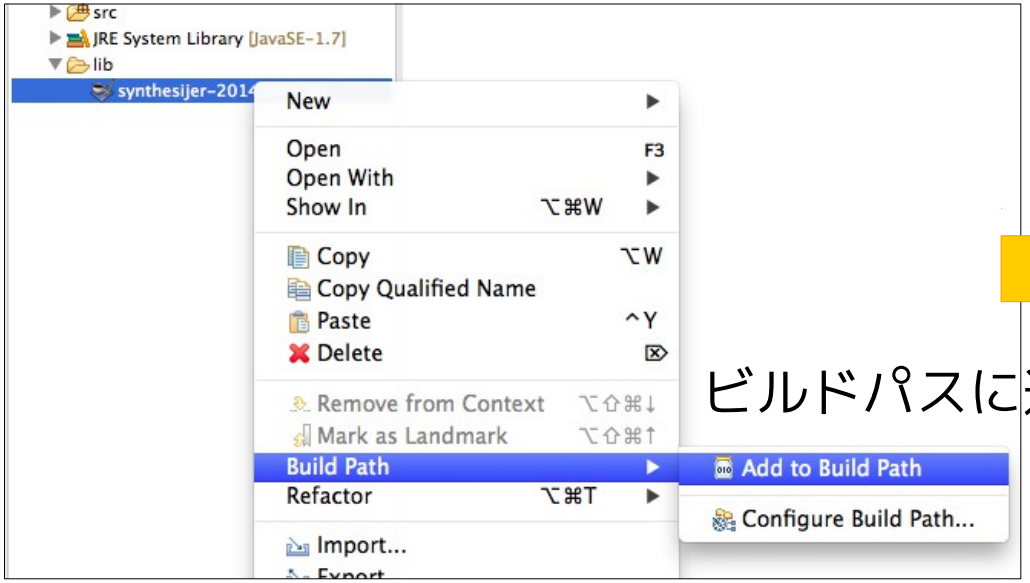
クイックスタート 3/8

(3) libフォルダを作ってDLしたJARをコピー，ビルドパスに追加する

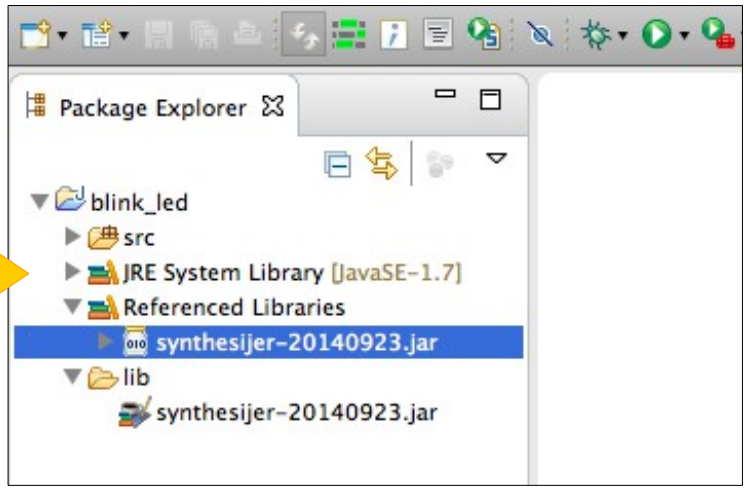




DLしたJARをD&Dでコピー



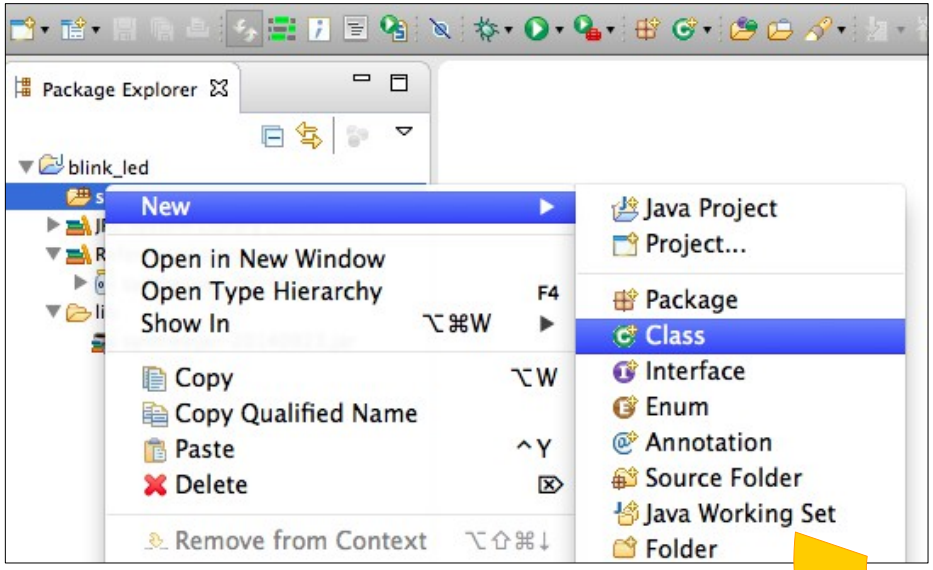
ビルドパスに追加



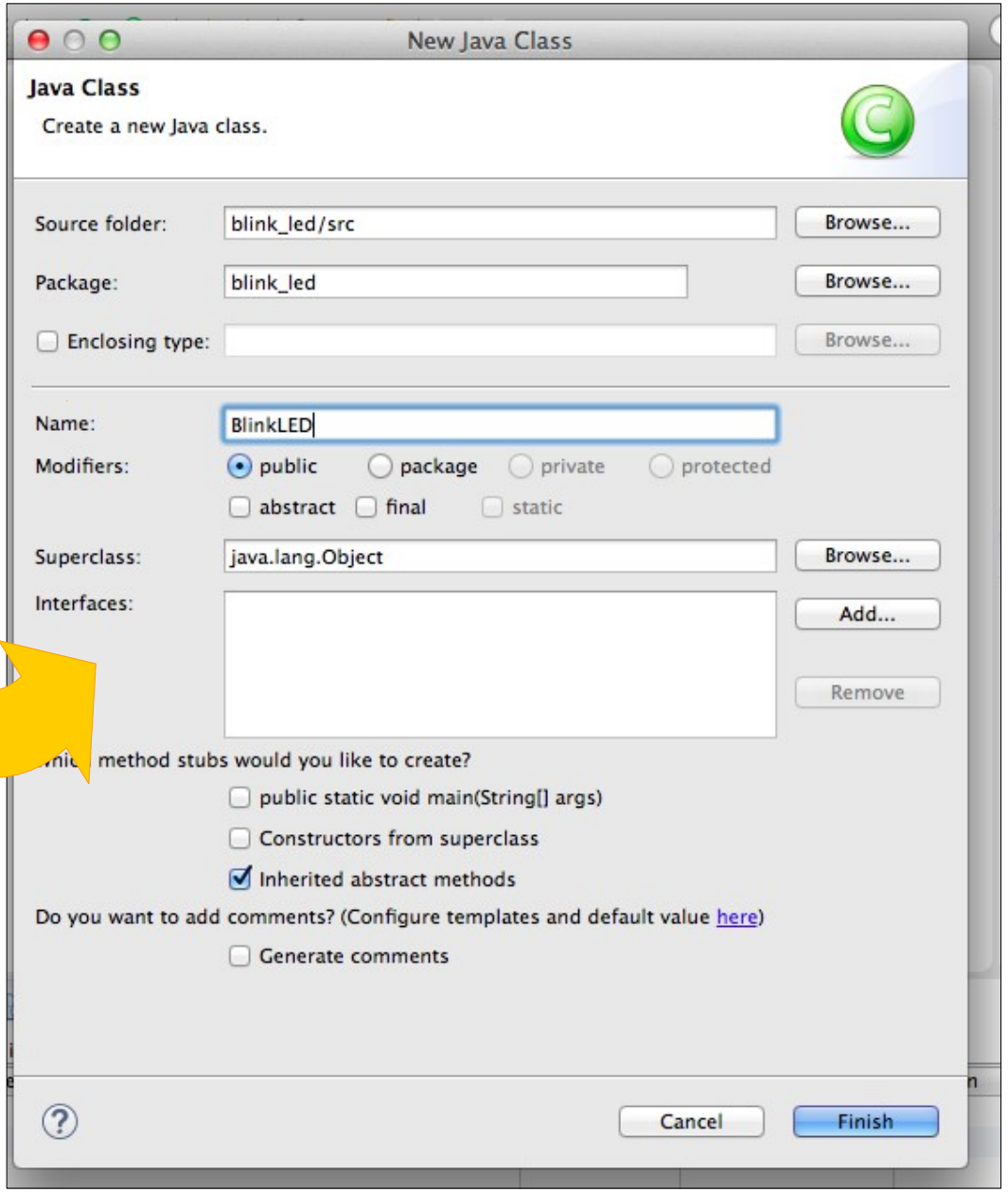
準備完了!!

クイックスタート 4/8

(4) Javaのクラスを作る

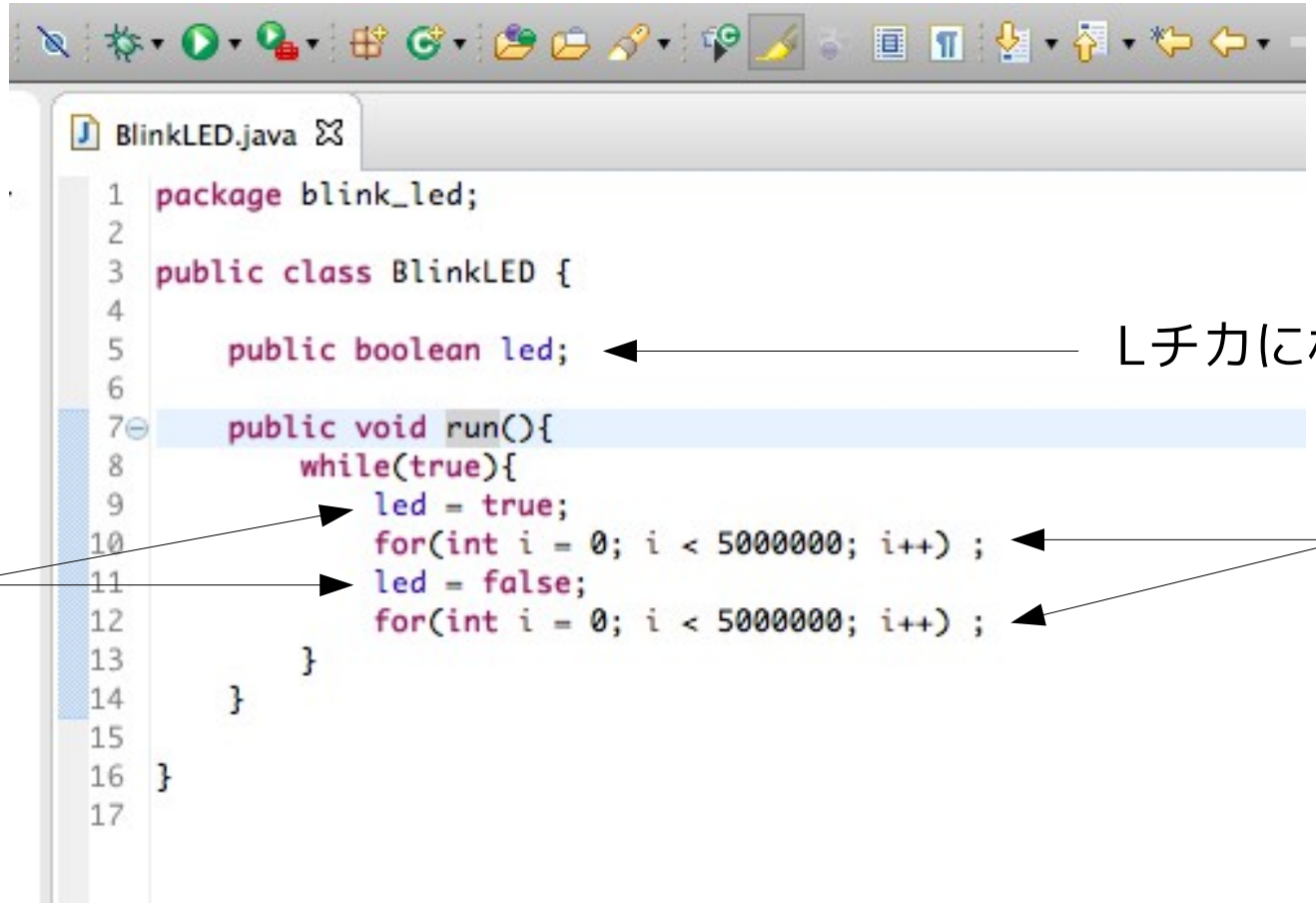


クラス生成ダイアログ



クイックスタート 5/8

(5) 間隔をおいて変数ledをtrue/falseするプログラムを書く



Lチカに相当する変数

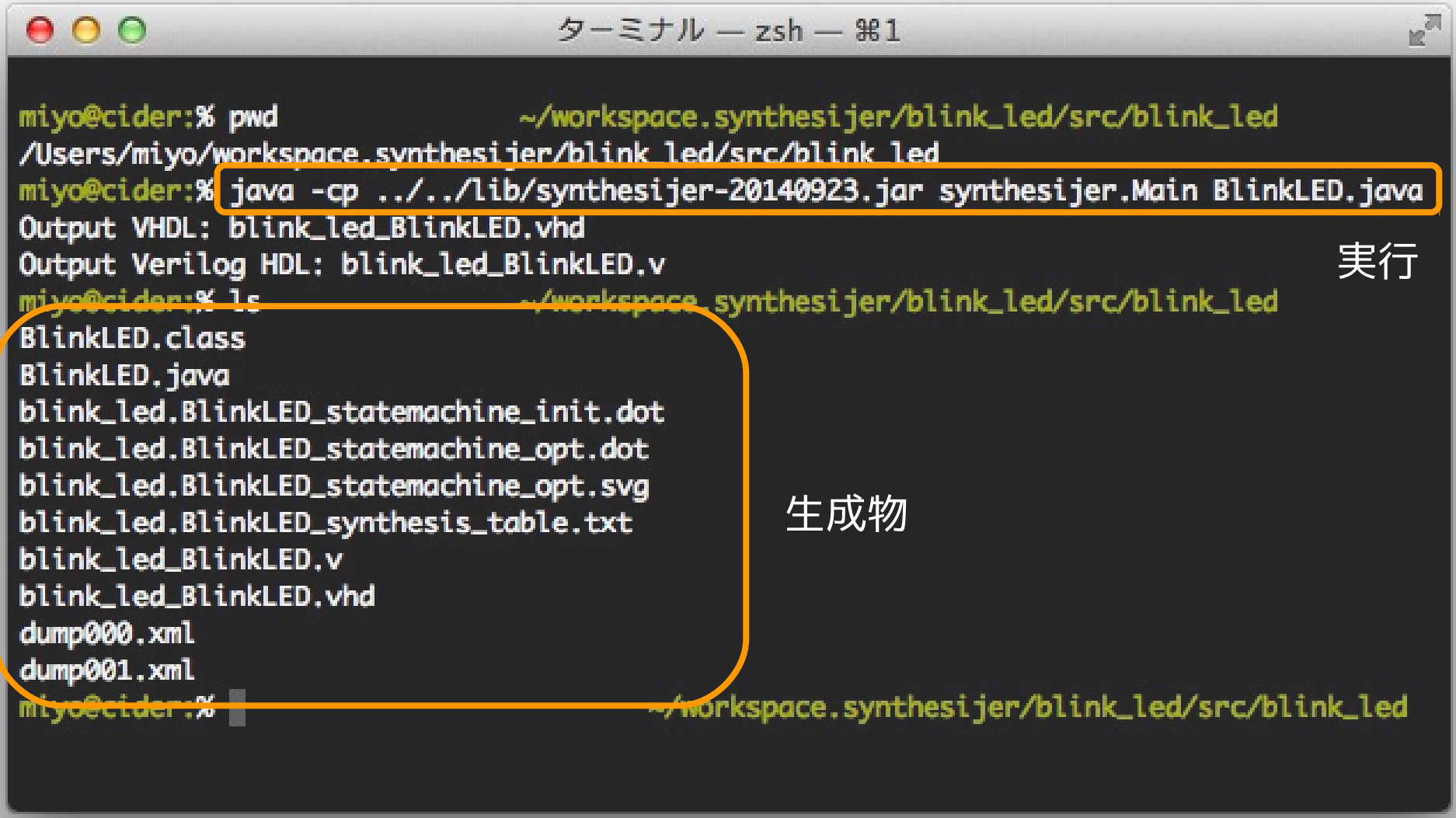
適当なウェイト

点滅

自動コンパイルが裏で動くので，Javaコードとしての正しさは即座にチェックされる

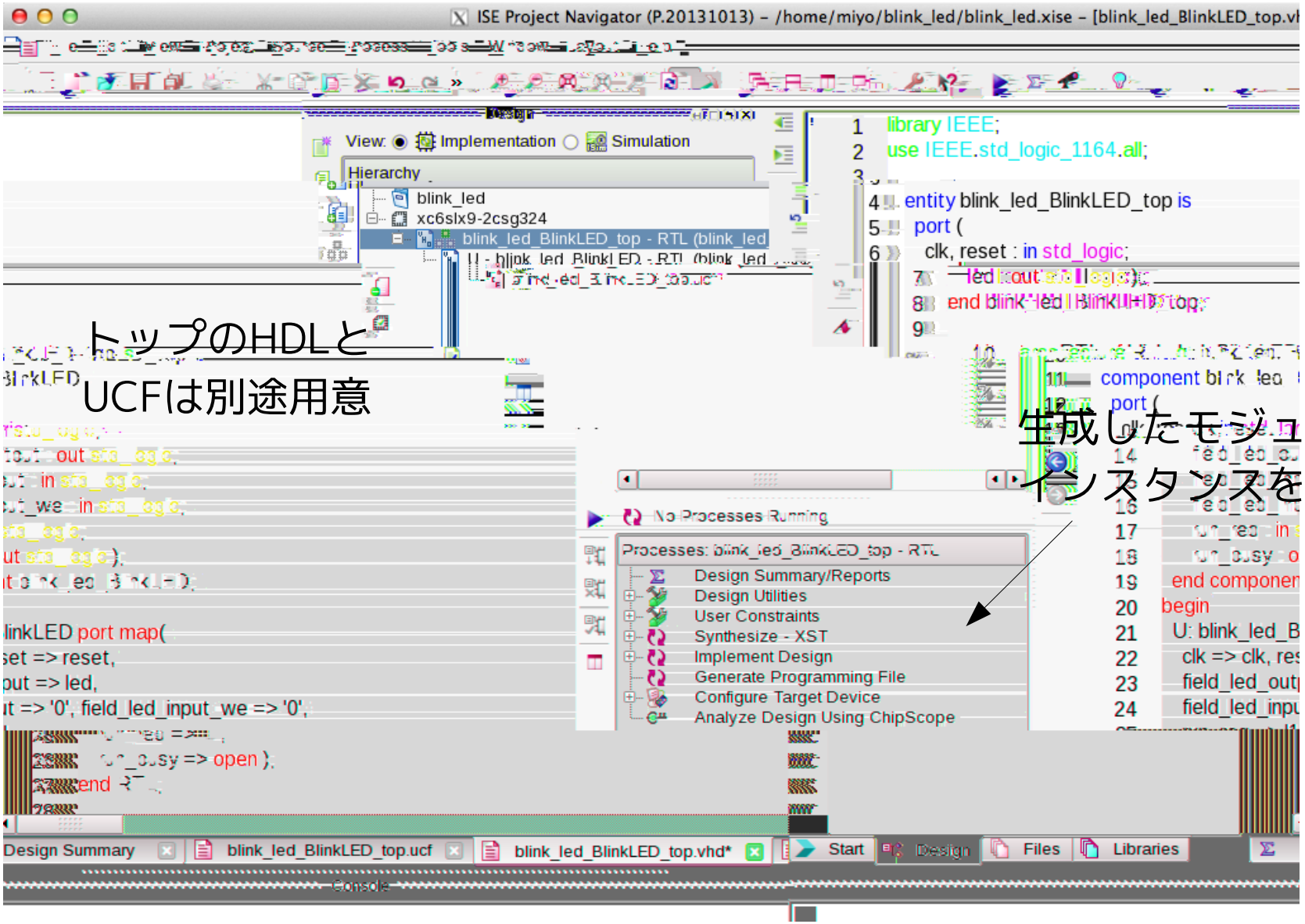
クイックスタート 6/8

(6) Synthesijerを使ってJavaコードをHDLに変換



クイックスタート 7/8

(7) ISEで合成，配置配線

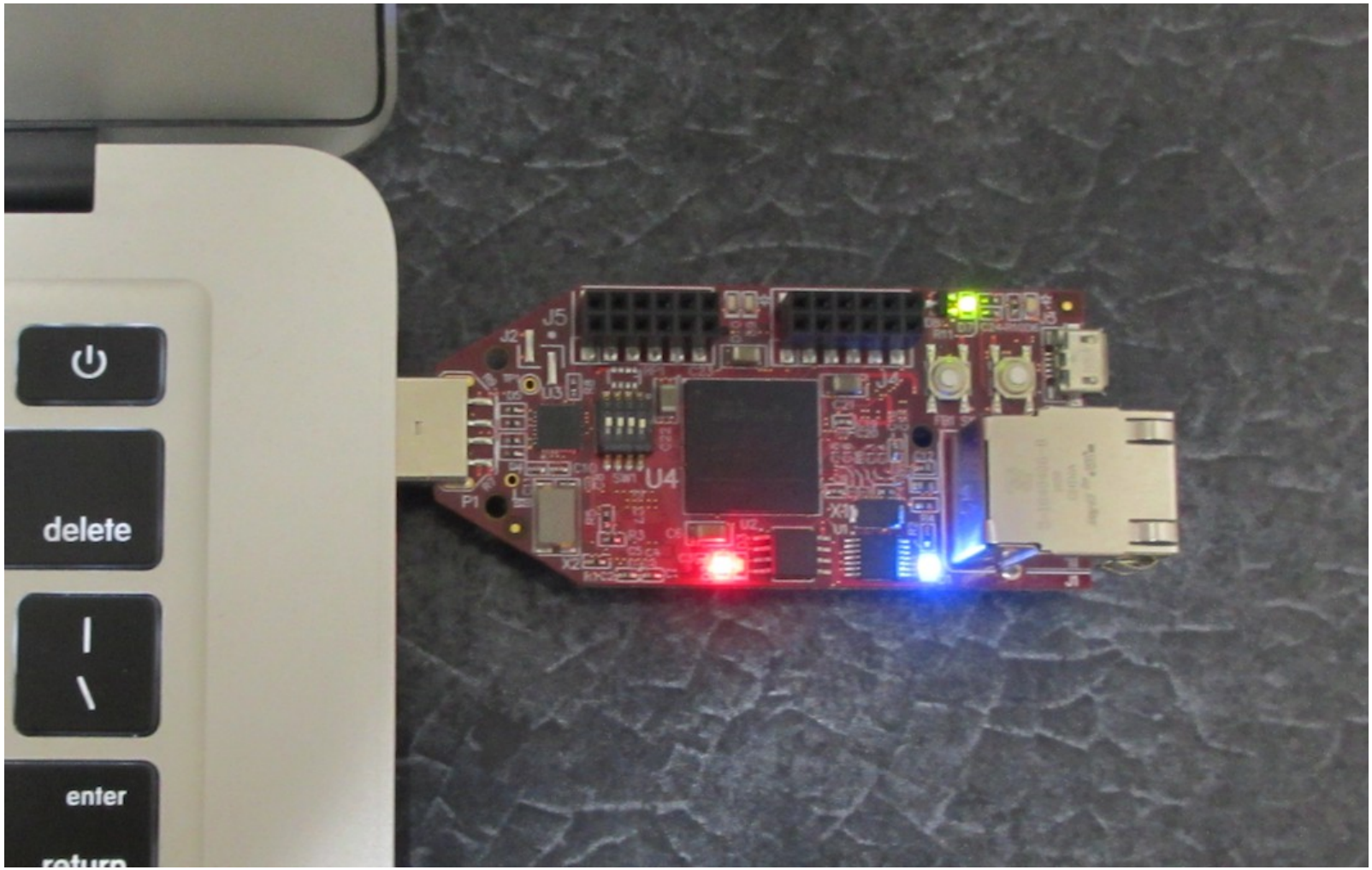


トップのHDLと
UCFは別途用意

生成したモジュールの
インスタンスを生成

クイックスタート 8/8

(8) FPGAをコンフィギュレーションして動作を確認



Synthesijer とは？

高位合成処理系に何を求めるか？

- ✔ 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
  - ✔ 短時間で動作を確認したい
  - ✔ 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
  - ✔ 粗粒度，細粒度の並列性の活用
  - ✔ IPコアや内部機能ユニットを活用したい
  - ✔ 設計空間の探索

高位合成処理系に何を求めるか？

- ✔ 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
  - ✔ 短時間で動作を確認したい
  - ✔ 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
  - ✔ 粗粒度，細粒度の並列性の活用
  - ✔ IPコアや内部機能ユニットを活用したい
  - ✔ 設計空間の探索

既存の高級言語ベース？

“FPGAの”高位合成処理系に何を求めるか？

- ✔ 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
  - ✔ 短時間で動作を確認したい
  - ✔ 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
  - ✔ 粗粒度，細粒度の並列性の活用
  - ✔ IPコアや内部機能ユニットを活用したい
  - ✔ 設計空間の探索

＋導入コストが小さいこと

Synthesijer

Javaベースの高位合成処理系

- ✔ クラスによるオブジェクト指向設計言語

  ← HWのモジュール設計との親和性は高そう

- ✔ 言語仕様で並列処理をサポート

  - ✔ Thread，wait-notify
- ✔ (Cと違い)明示的なポインタの扱いが不要

  - ✔ 言語の想定するメモリ構造から解放され得るかも

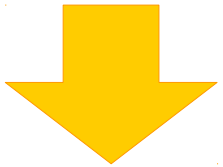
- ✔ 動的な振る舞いは厄介そう

Synthesijer 設計基本方針

- ✔ Javaプログラムをそのまま HDL→HW化
  - ✔ 追加構文，データ型は導入しない
  - ✔ 使える記述には制限を導入

HDLで書けることをすべてJavaで書けるようにするではない

Javaで書けることををすべてHDL(HW)にするではない



Javaとして実行可能なプログラムをHWにする

≒ フェッチのないJavaプロセッサをHDLで直接生成する

Synthesijer 活用方法 - 3つのパタン

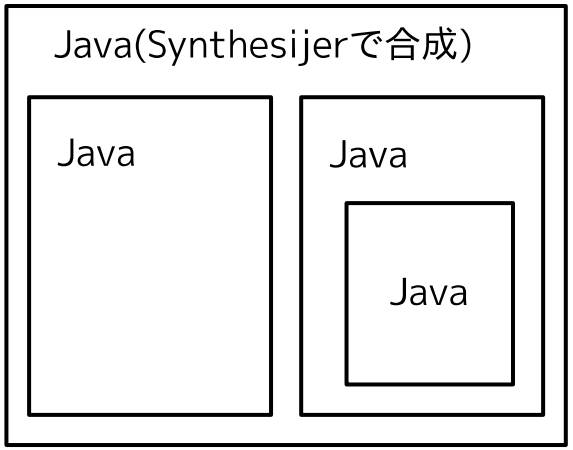
3つの開発パタン

(1) Javaによるモジュールだけでシステムを構成

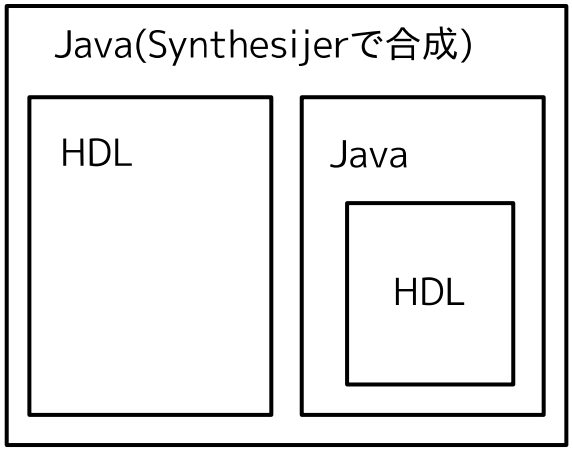
(2) HDLによるモジュールを部品として利用．全体管理はJavaで記述
　＝ 既存IPコア，FPGAの性能を活用

(3) Javaによるモジュールを部品として利用．全体管理はHDLで記述
　＝ 複雑なアルゴリズムをSWプログラマに書いてもらう．SW資産の活用

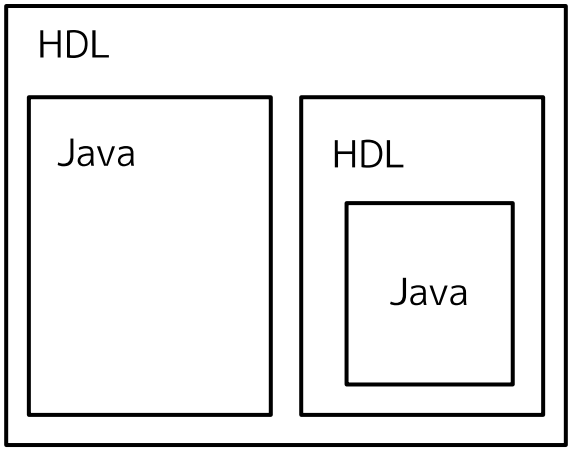
(1)Javaでシステムを構成



(2)Java + HDL記述の部品



(3)HDL + Java記述の部品



(1) Javaでシステムを構成

- すべてをJavaで記述
- トップモジュール，UCF/XDCなんかは必要
- I/Oなどは用意されたライブラリを利用
  - シリアルI/O，SC1602，純粋なInput/Output, …
  - AXI(32bit逐次，シンプルなキャッシュ)，UPL，…

(2) Java+HDL記述の部品

- JavaプログラムにHDL記述の部品を埋め込む
- Javaで記述したオブジェクトのようにHDL記述部品をインスタンス化して使う
- HDLモジュールを簡単に使い回せる
- Synthesijer内部でどうオブジェクトが扱われるかを知る必要がある
  - JavaとのブリッジのためにHDLModuleを継承したクラスを実装
  - 合成時にはHDL記述のコードと組み合わせる
- HWプログラマがサポートしつつSWプログラマにFPGAを活用してもらう!!


```
HogeModule obj = new HogeModule();

obj.x = 0;
obj.data[0] = 100;
```

見かけ上の接続

```
class HogeModule extends HDLModule{

 int x;
 int[] data;

 HogeModule(){ super("hoge_module", ...); }
```

実際の接続

```
entity hoge_module is
 port(
  x : ...
  data_waddr : …
  data_wdata : ...
```

SWプログラマに
HWモジュールを有効活用してもらう

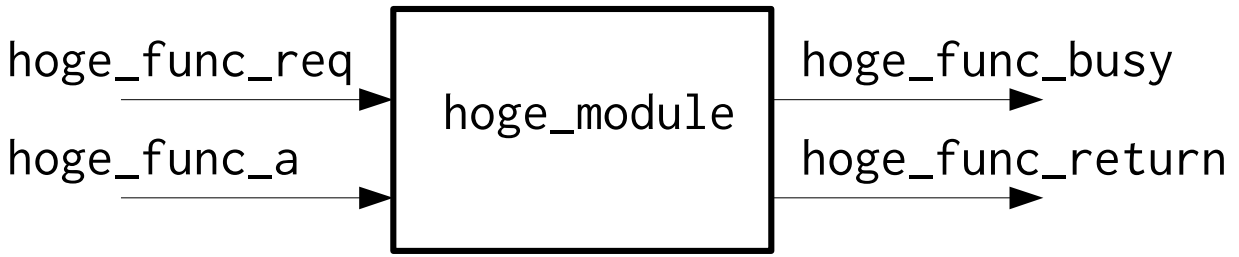
(3) HDL+Java記述の部品

- ✔ HDLで設計したシステムにJava記述の部品を埋め込む
- ✔ 複雑なアルゴリズムの処理でもJavaなら(比較的)手軽に実装
- ✔ HDLで自然に実装できるストリーム処理，並列処理を活用
- ✔ Synthesijerで合成した回路がどういう構成か知る必要がある
- ✔ SWプログラマ/資産をFPGA設計に活用!!

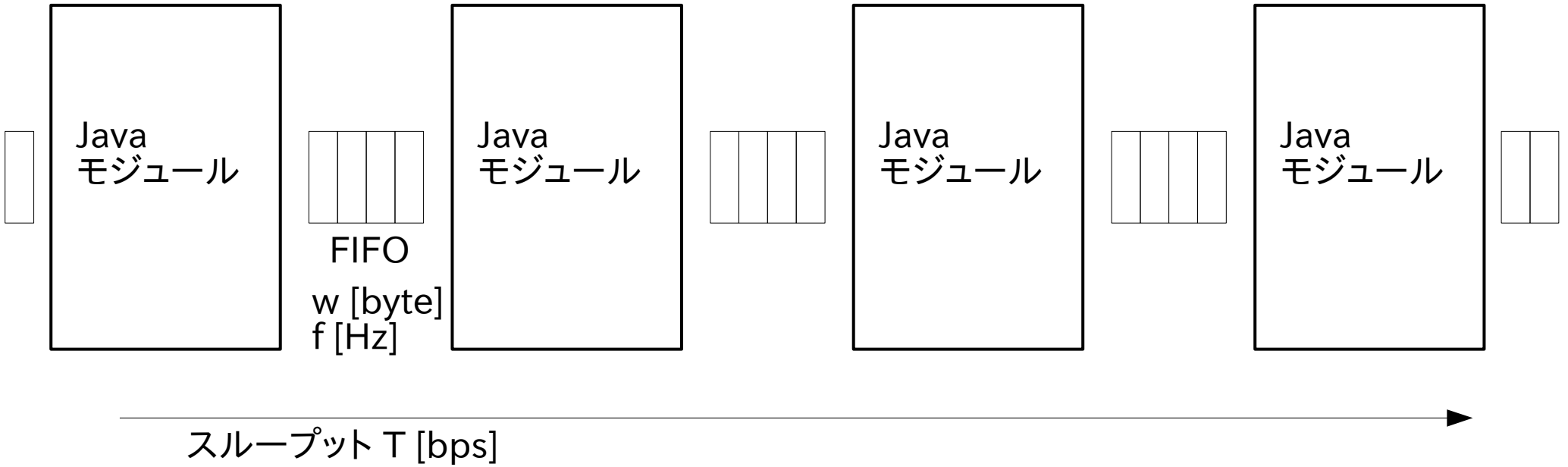

```
class HogeModule{

 int hoge_func(int a){
   …
 }
```

Synthesijer

```
entity hoge_module is
 port(
  hoge_func_a : in ...
  hoge_func_return : out …
  hoge_func_req : in …
  hoge_func_busy : out ...
```



信号処理システムへの応用の展望



パケットデータが d [byte] のとき全データ入力を受け取るのにかかる時間 = d/w * 1/f [sec]
同様に、全データの出力にかかる時間 = d/w * 1/f [sec]

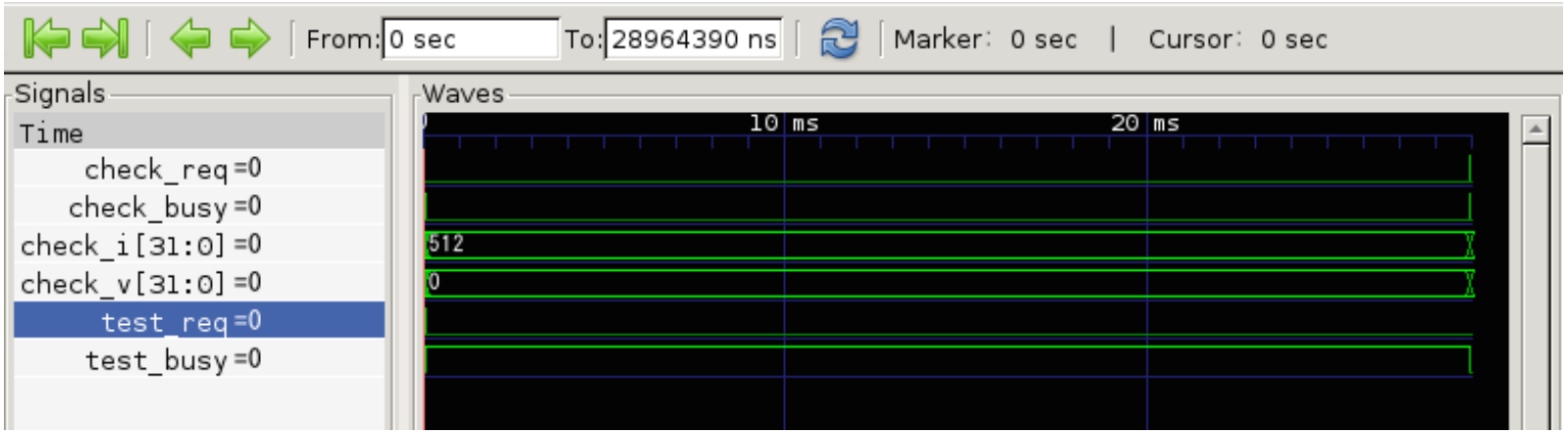
スループットT [bps]を実現するとき、パケットデータを(8*d) * 1/T [sec]内で処理し続ける必要がある

→ 各モジュールで処理に使える時間 t は 8*d/T – 2*d/(w*f) [sec]
= (8*d/T-2*d/(w*f)) / (1/f) [cycle]

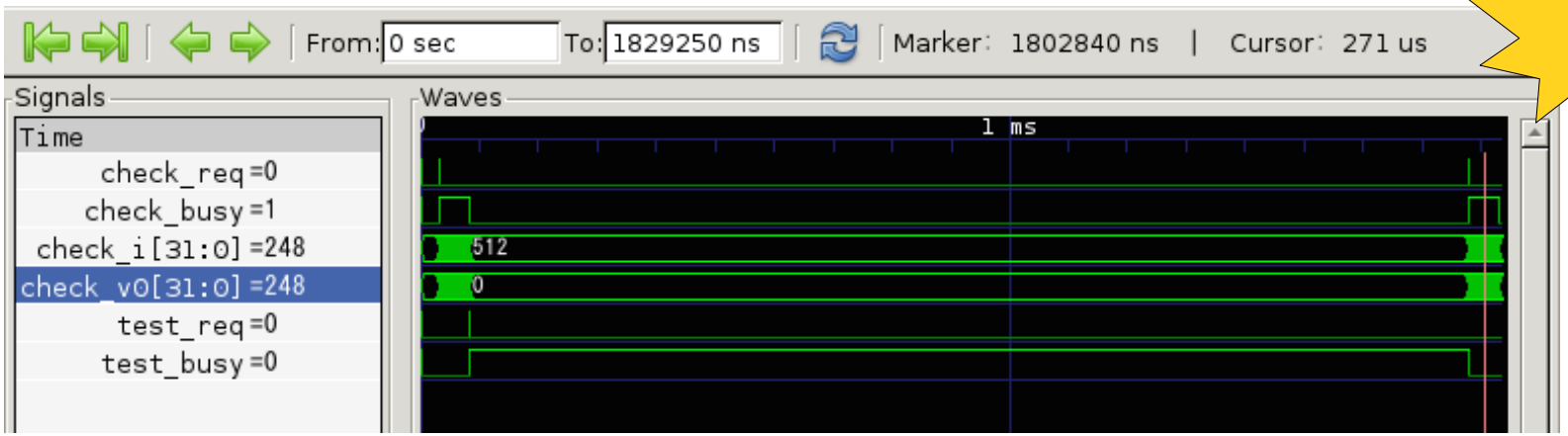
たとえば、d=1500, T=1G, w=4, f=100Mのとき
1パケットあたりの処理にかけられるサイクル数は450サイクル。 f=200Mなら1650サイクル

複雑なアルゴリズム処理はJavaで実装

- 例: バブルソートじゃなくてマージソートの採用，など
  - バブルソート(512個の降順→昇順)



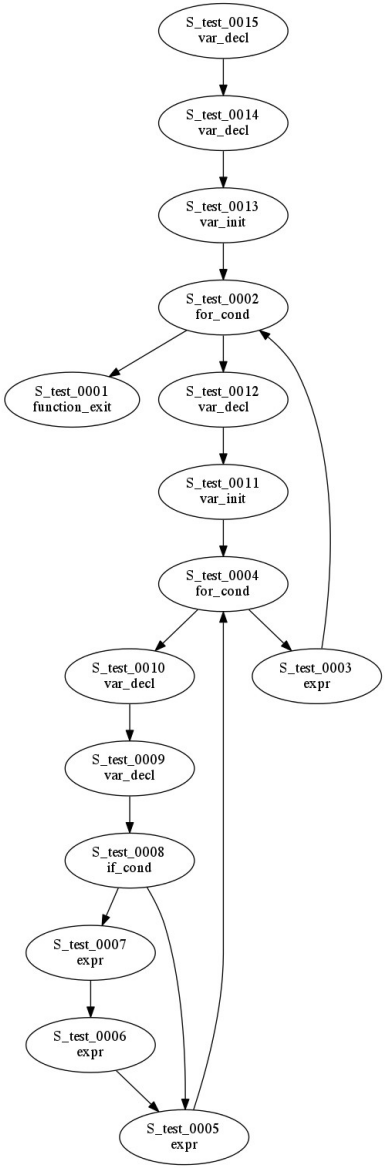
  - マージソート(512個の降順→昇順)



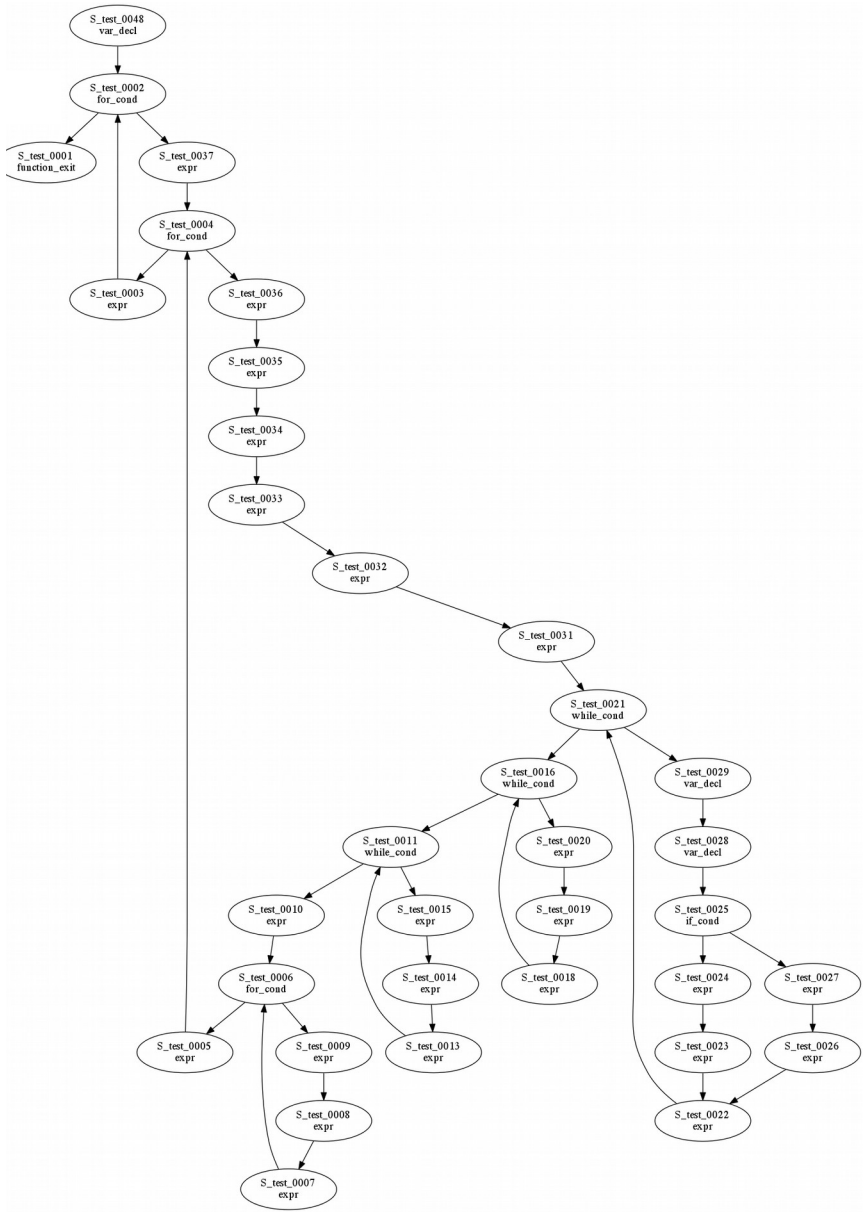
x15

複雑なアルゴリズム処理はJavaで実装

- ✔ バブルソートの状態遷移



- ✔ マージソートの状態遷移



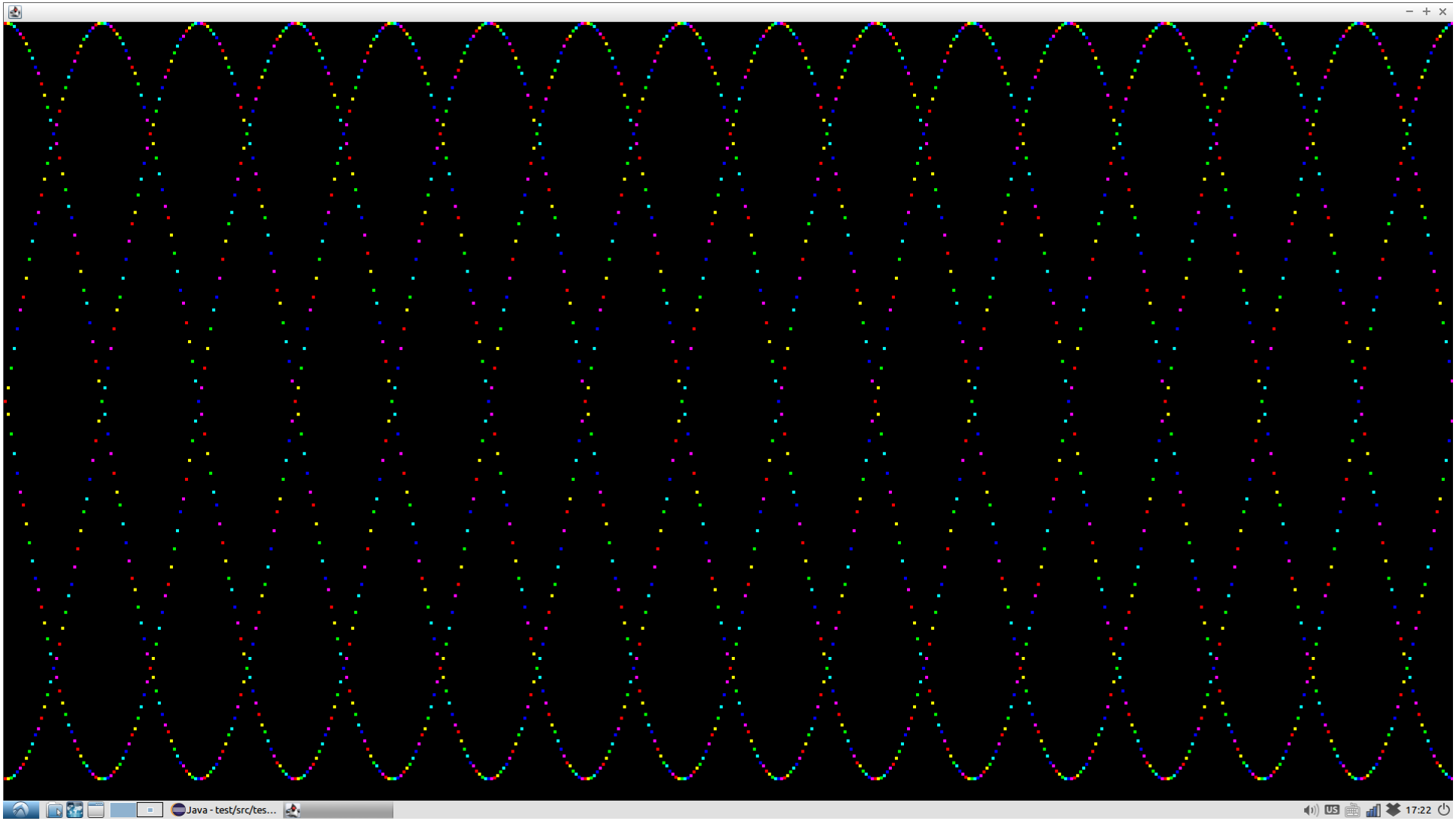
Synthesijer サンプル

サンプル

- ✔ グラフィクス表示
- ✔ BrainF**k インタプリタ

サンプル1: グラフィクス表示

- ✔ 90度位相ずれしたサインカーブをSWで描画

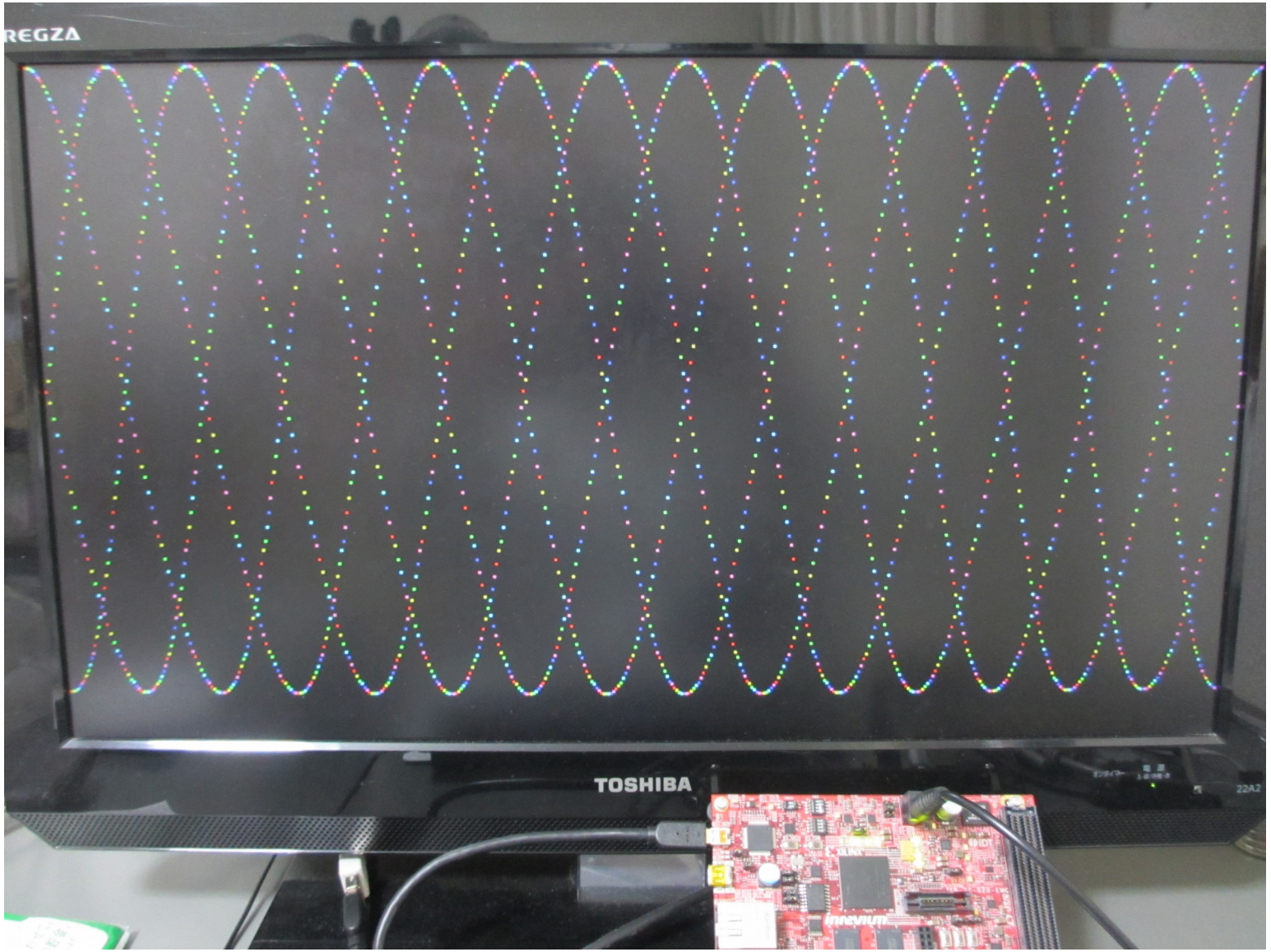


サインカーブ描画のJavaコード(抜粋)

```
public class RGBTest{
    ...
    private final TestFrame obj = new TestFrame();
    private final SinTableRom sin = new SinTableRom();
    private final int colortbl[] = new int[6];

    public void paint_sincurve(int offset){
        int c_id = 0;
        for(int i = 0; i < (1920 >> 2); i++){
            int x = i << 2;
            int y = sin.sintable[(i+offset)&0x7F];
            int c = colortbl[c_id];
            obj.fill_rect(x, y, 4, 4, c);
            c_id = c_id + 1;
            if(c_id == 6) c_id = 0;
        }
    }
    public void run(){
        init_colortbl();
        while(true){
            paint_sincurve(0);
            paint_sincurve(32);
            paint_sincurve(64);
            paint_sincurve(96);
            obj.flush();
        }
    }
```

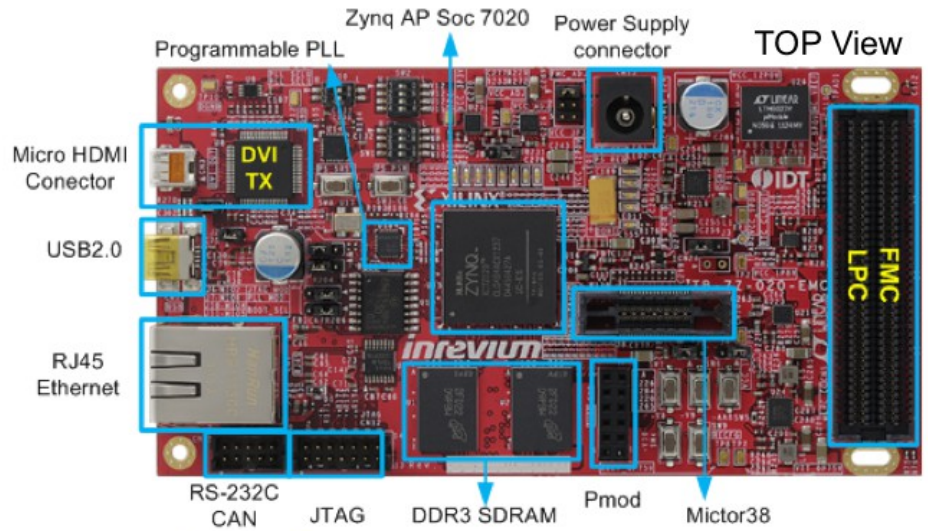
FPGAでの実行例

- ✔ 90度位相ずれしたサインカーブを実機で確認

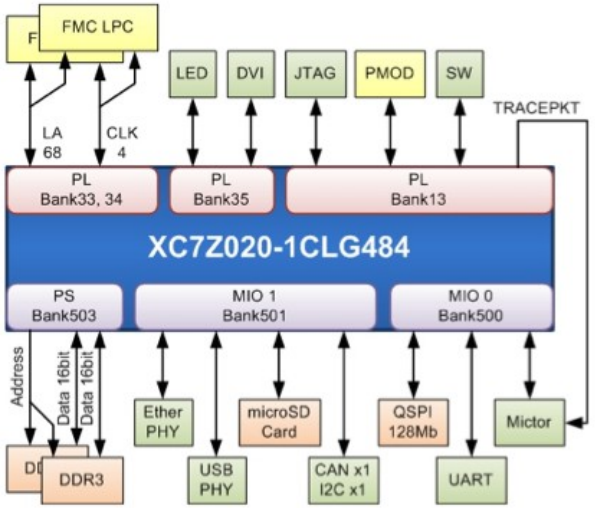


使用したボード

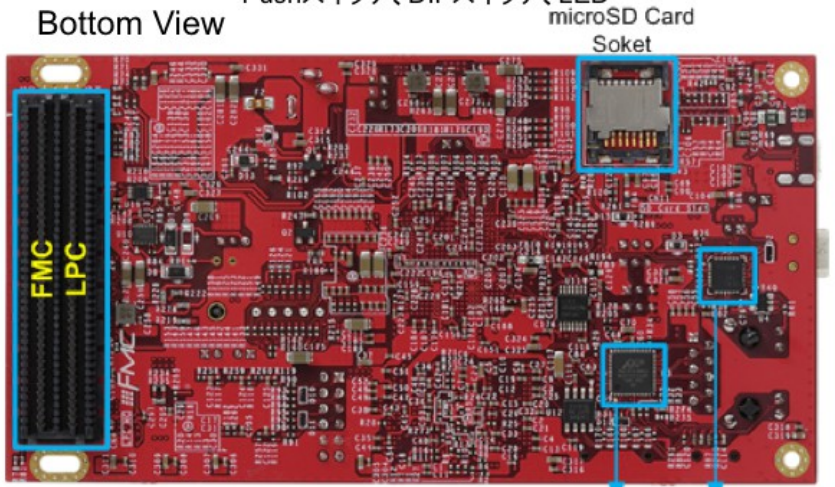
TB-7Z-020-EMC



- 搭載デバイス: XC7Z020-CLG484
- 搭載メモリ:
  - DDR3 SDRAM x 2(16bit data bus x 2)
  - MicroSDカードソケットとメディア x 1
  - 128Mbit QSPI フラッシュメモリ x 1
- 拡張インタフェース
  - FMC(LPC) x 2
  - PMOD(Digilent module interface) x1
- 汎用インタフェース
  - Gigabit Ethernet x 1
  - USB2.0 x 1(Host Device)
  - UART(RS-232C) x 1
  - CAN x 1
  - DVI TX x 1
- デバックインタフェース
  - MICTOR-38 for ARM
  - 10 pin header for Xilinx Cable
  - Pushスイッチ、DIPスイッチ、LED



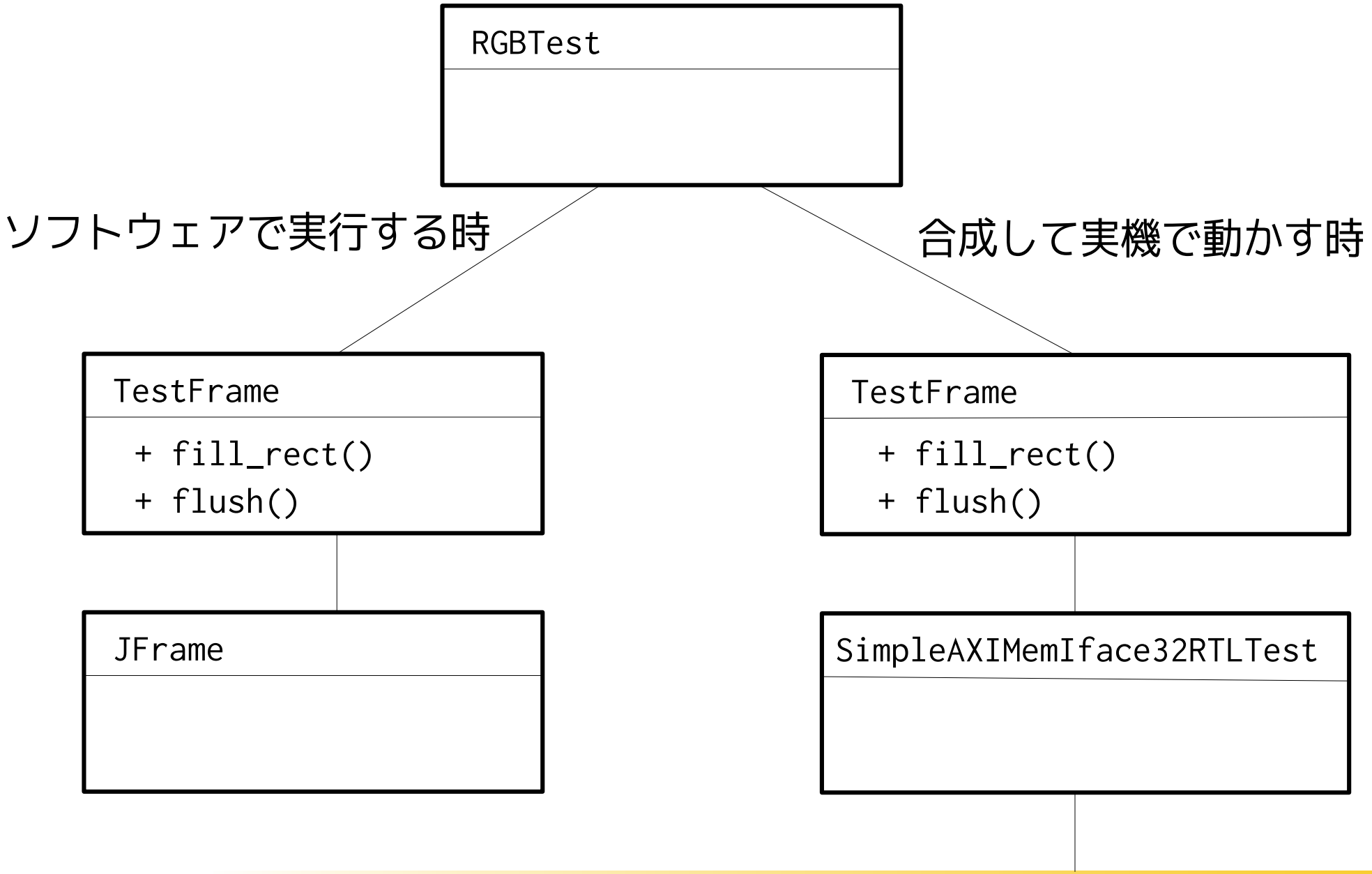
Bottom View



TOKYO ELECTRON DEVICE LIMITED

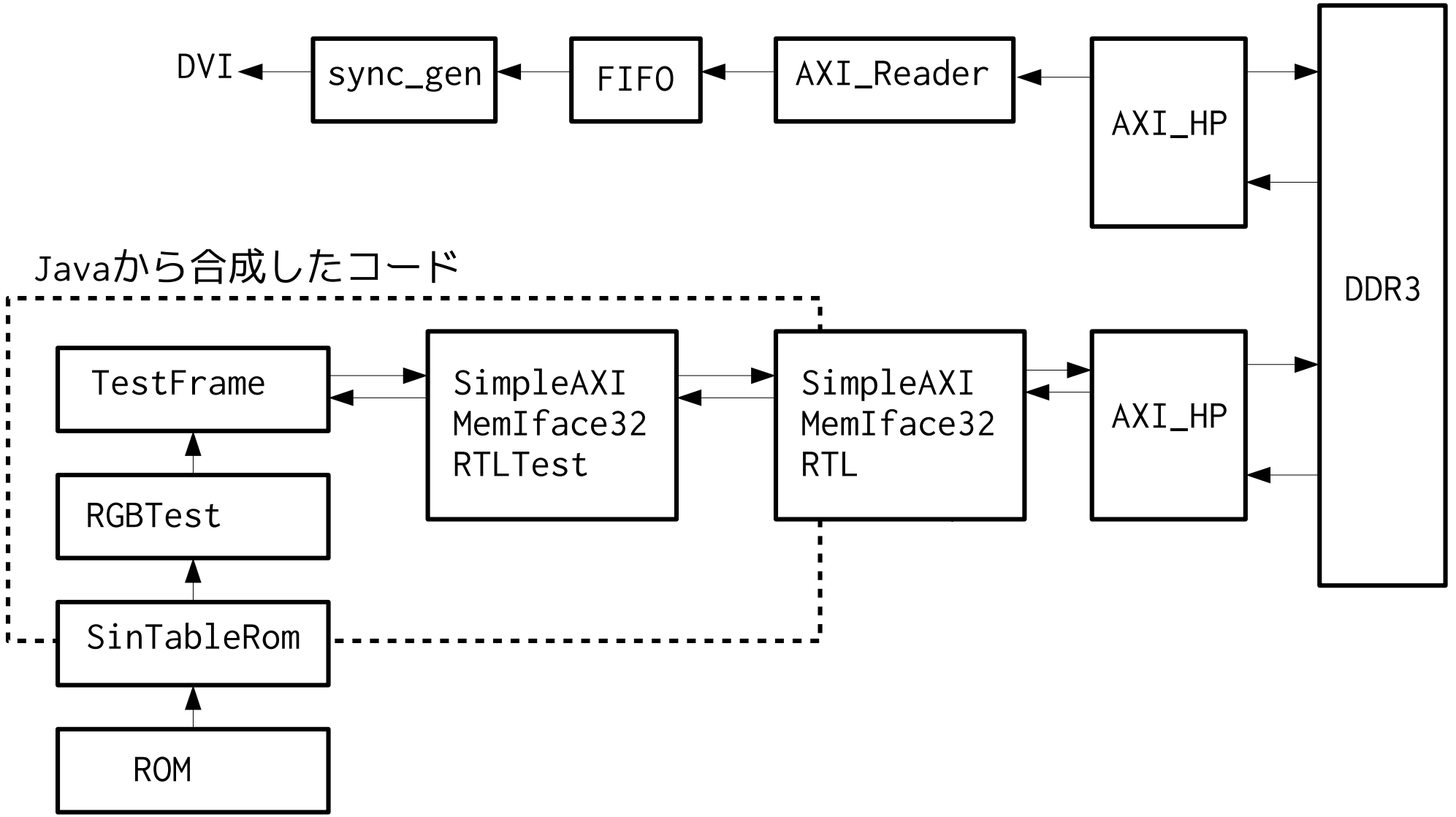
Auther / BU or Group 1

SWとHWで同じコードを利用



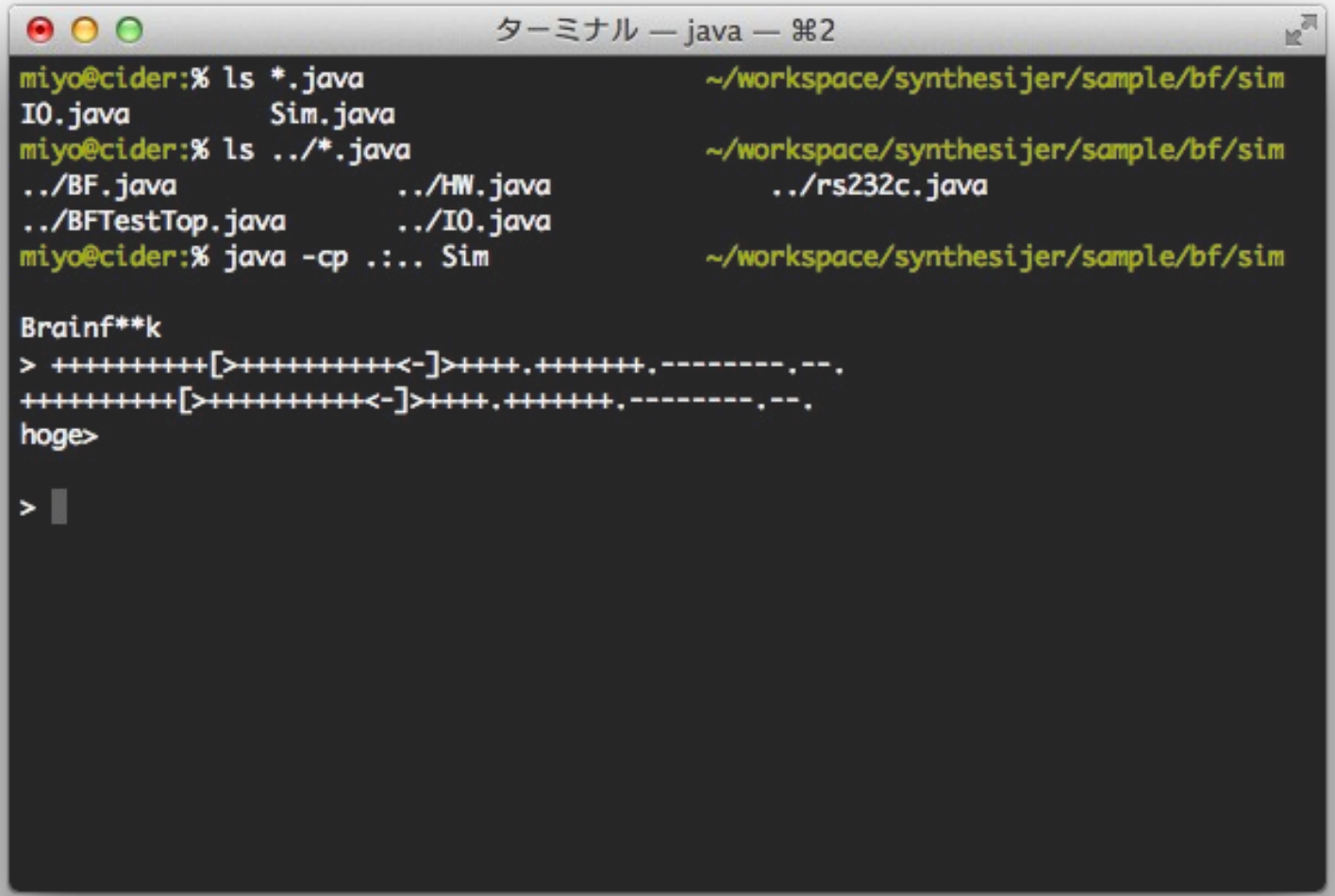
システム全体のブロックダイアグラム

- ✔ 水面下はちょっと複雑



サンプル2: BrainF**k

- 標準入出力越しでBrainF**kインタプリタを実行
  - +, -, >, <, [,], ., ,の記号からなるインタプリタ



サンプル2: BrainF**k

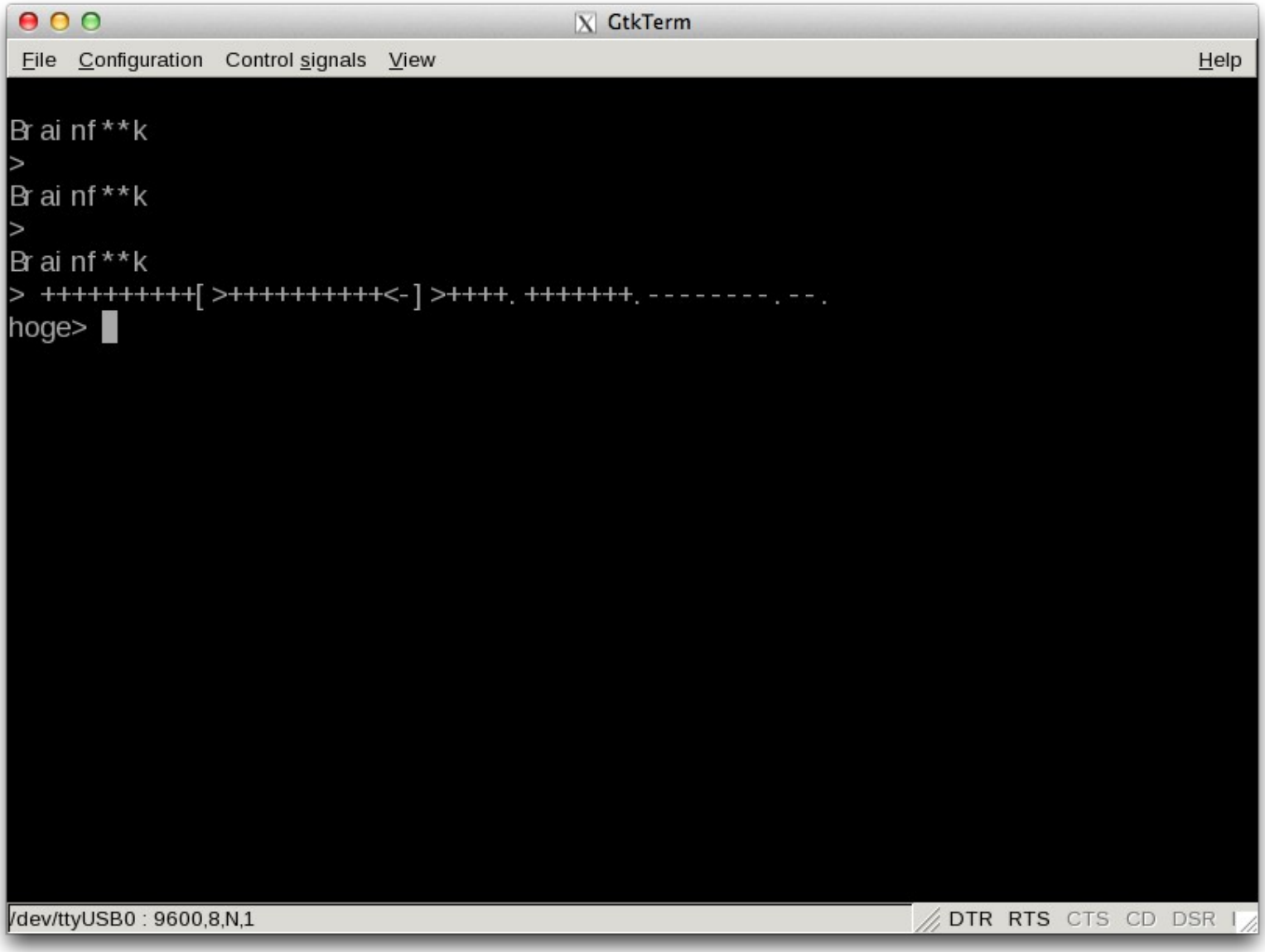
```
private IO io = new IO();
private byte[] prog = new byte[CODESIZE];
private byte[] data = new byte[ARRAYSIZE];
private int ptr, pc;

public boolean step() {
    byte cmd = prog[pc];
    byte tmp;
    int nlvl = 0;
    switch (cmd) {
    case 0: return false;
    case '>': ptr++; break;
    case '<': ptr--; break;
    case '+': data[ptr] = (byte) (data[ptr] + 1); break;
    case '-': data[ptr] = (byte) (data[ptr] - 1); break;
    case '.': io.putchar(data[ptr]); break;
    case ',': data[ptr] = io.getchar(); break;
    case '[':
        if (data[ptr] == (byte) 0) {
            while (true) {
                pc++;
                if(prog[pc] == ']' && nlvl == 0) break;
                if(prog[pc] == '[') nlvl++;
                if(prog[pc] == ']') nlvl--;
            }
        }
        break;
    case ']':
        ...
    }
}
```

サンプル2: BrainF**k

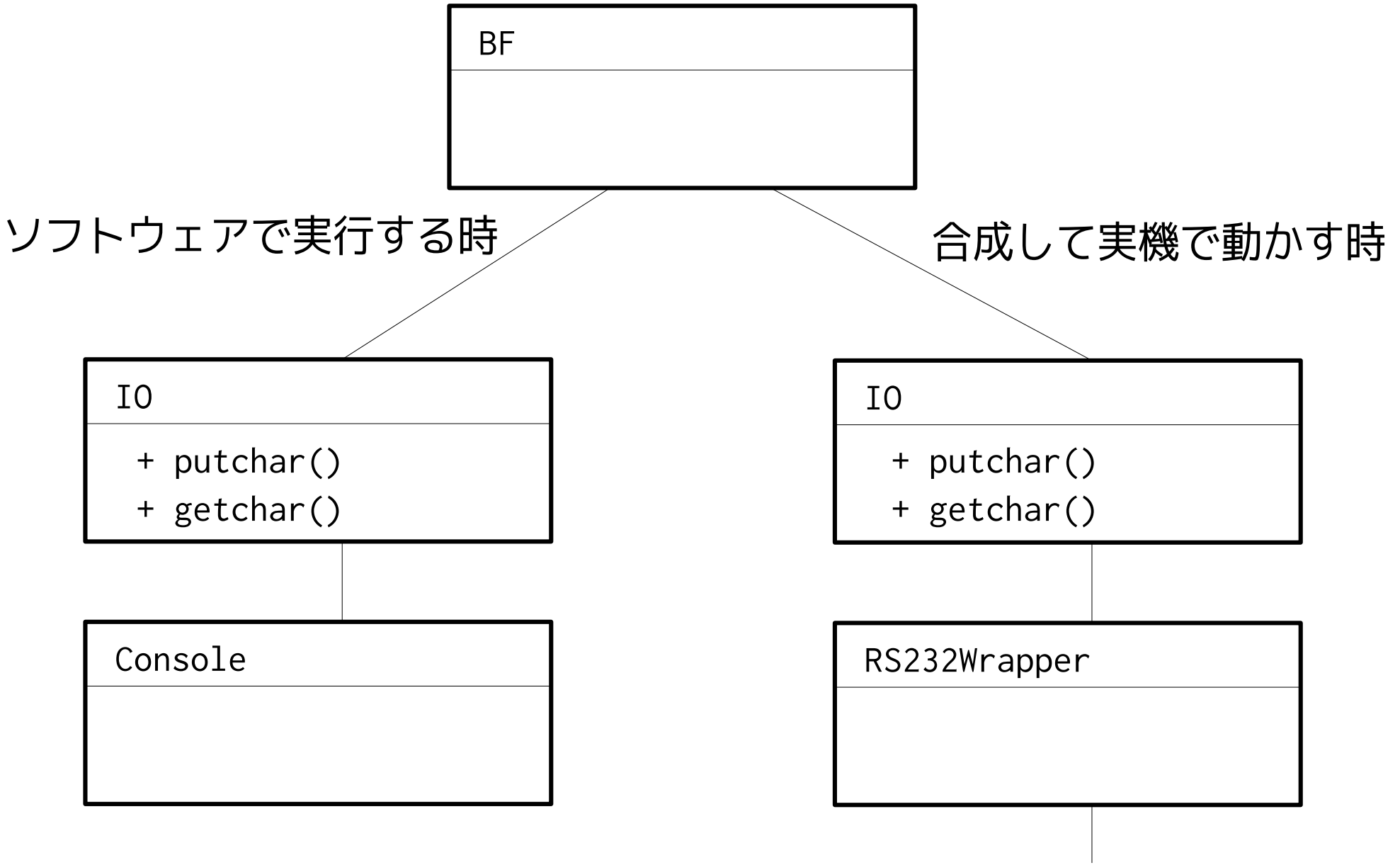
```
public void read() {
    prompt();
    for (int i = 0; i < CODESIZE; i++) {
        byte b;
        b = io.getchar();
        //io.putchar(b);
        if (b == '\n' || b == '\r') {
            prog[i] = (byte) 0;
            break;
        } else {
            prog[i] = b;
        }
    }
}

public void print() {
    boolean flag = true;
    for (int i = 0; i < CODESIZE; i++) {
        byte b = prog[i];
        if (b == 0) {
            break;
        }
        io.putchar(b);
    }
    io.putchar((byte) '\n');
}
```

FPGAで実行

- ✔ 標準入出力の代わりにシリアル通信越しで実行

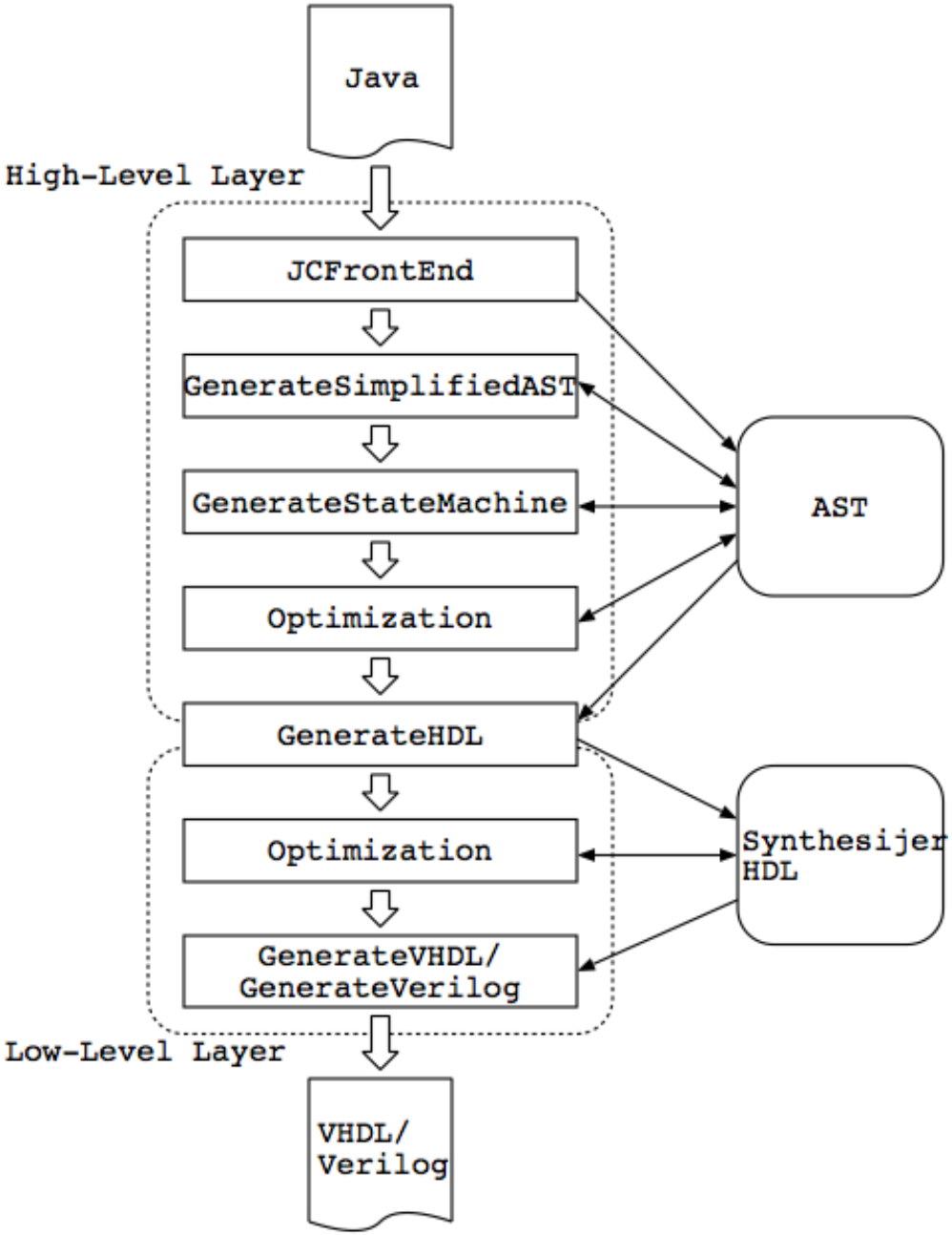


SWとHWで同じコードを利用



Synthesijerの合成方法について

Synthesijer コンパイルフロー

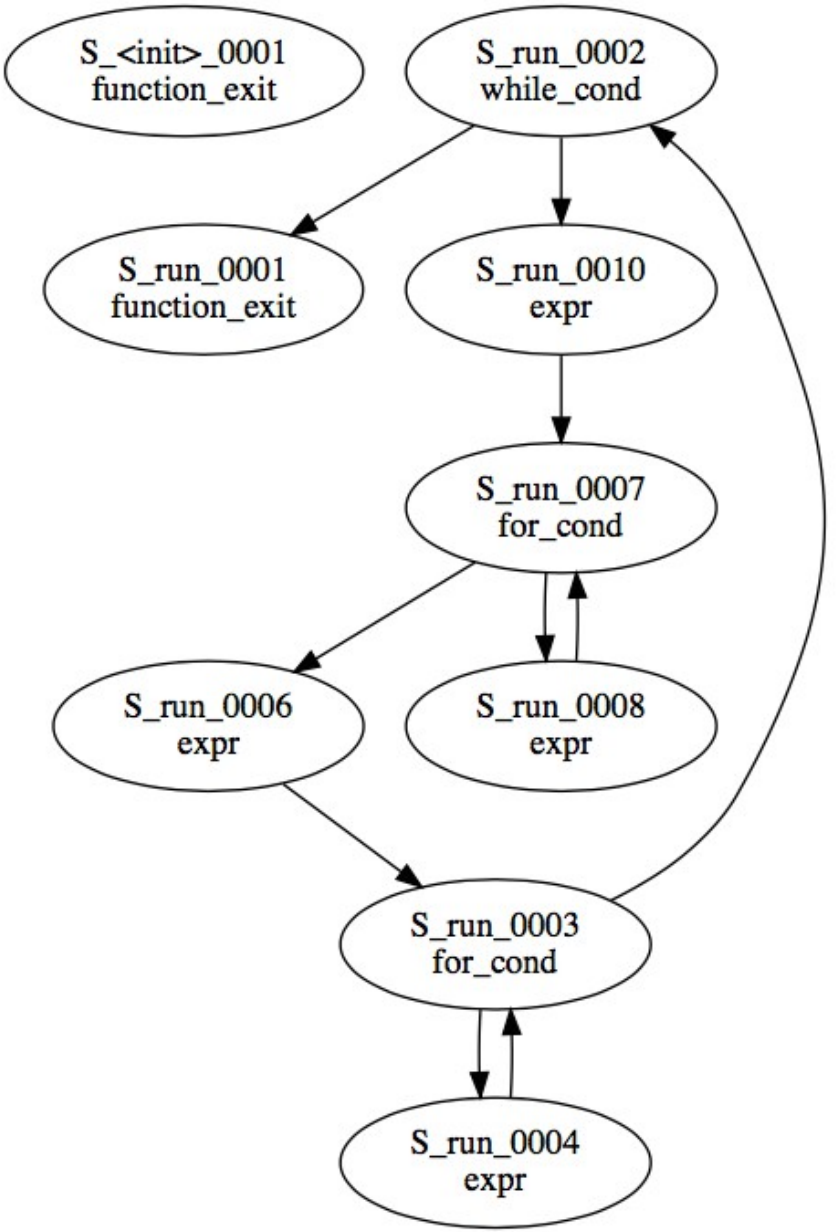


- High-Level Layer
  - 抽象構文木
  - HDLにしやすい形に変換
  - 最適化
- Low-Level Layer
  - モジュール
  - ステートマシン
  - 演算処理
  - 信号/ポート

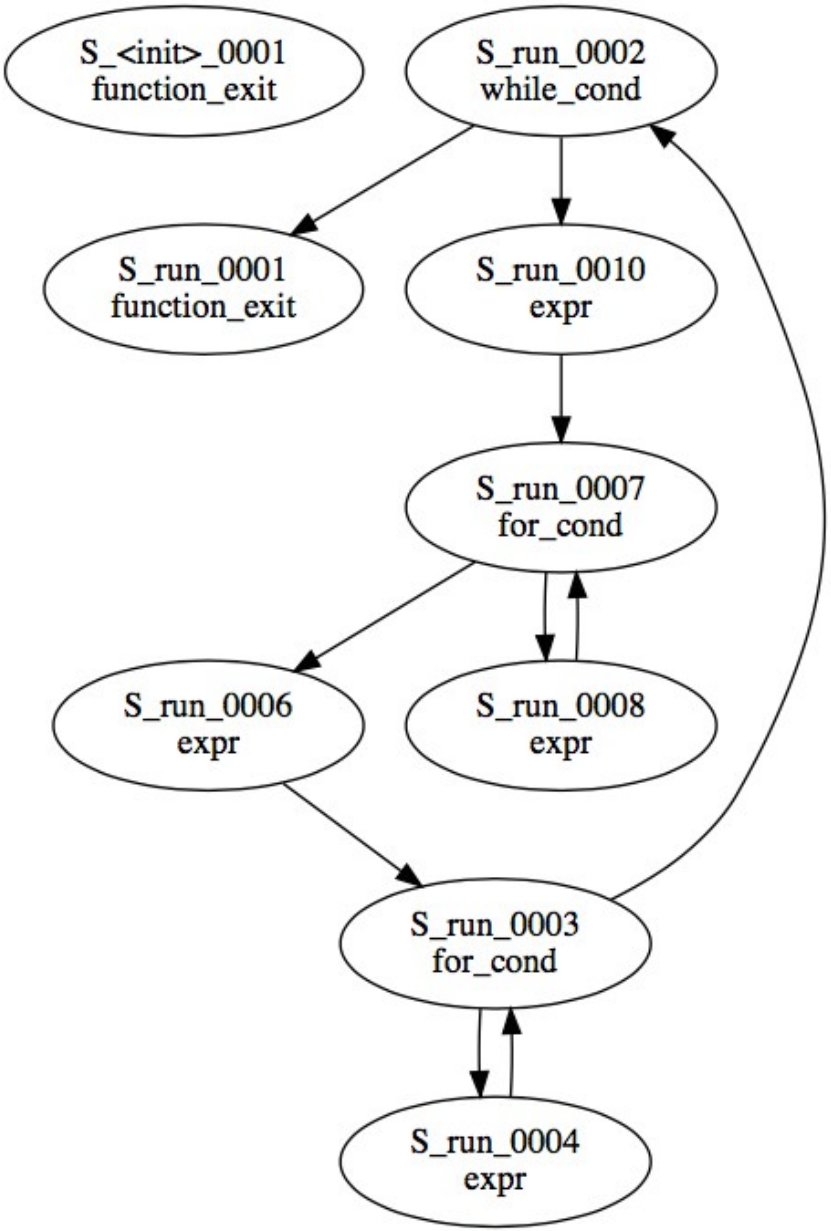
コンパイルの例 - シーケンサの生成

BlinkLED.java

```
1  package blink_led;
2
3  public class BlinkLED {
4
5      public boolean led;
6
7      public void run(){
8          while(true){
9              led = true;
10             for(int i = 0; i < 5000000; i++) ;
11             led = false;
12             for(int i = 0; i < 5000000; i++) ;
13         }
14     }
15
16 }
17
```

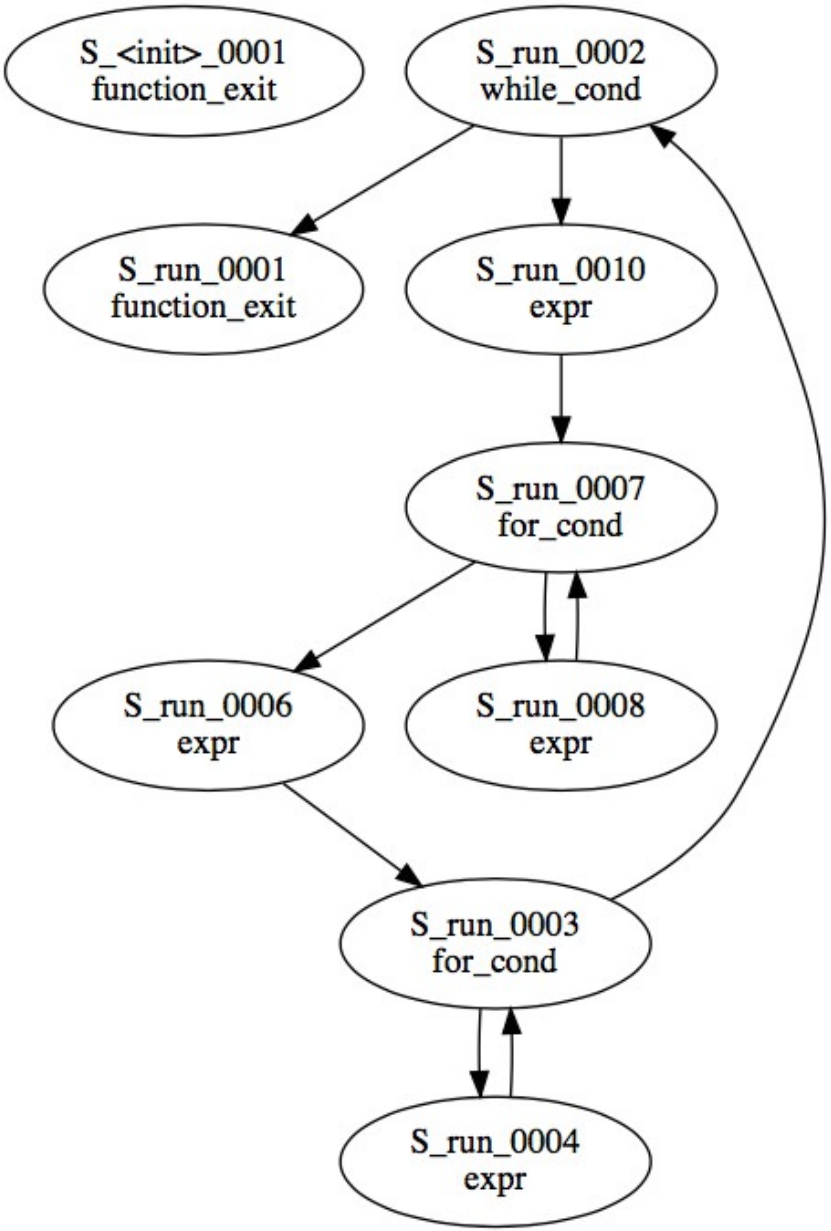


コンパイルの例 - データ処理



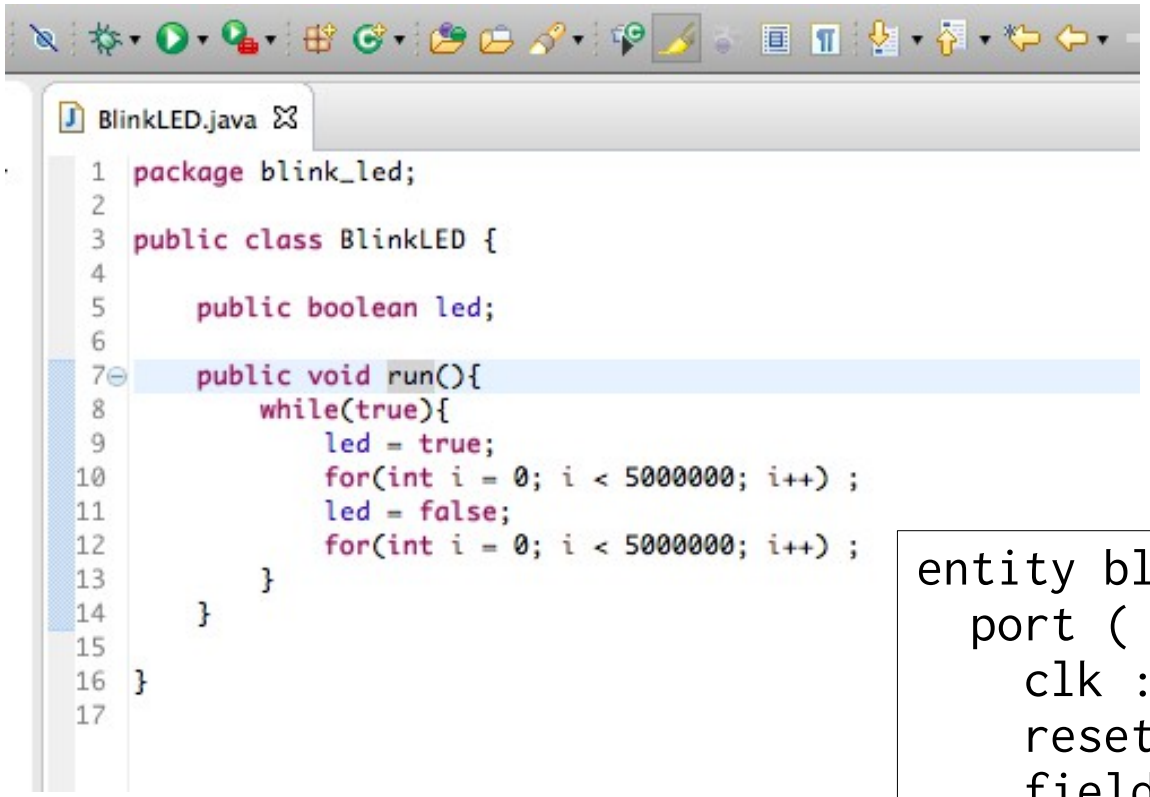

```
...
  tmp_0021 <= '1';
...
  tmp_0024 <= '0';
...
  process(clk)
  begin
    if clk'event and clk = '1' then
      if reset = '1' then
        led <= '0';
      else
        if S_run = S_run_S_run_0010 then
          led <= tmp_0021;
        elsif S_run = S_run_S_run_0006 then
          led <= tmp_0024;
        else
          led <= tmp_0001;
        end if;
      end if;
    end if;
  end process;
...
```

コンパイルの例 - データ処理




```
...
  tmp_0022 <= X"00000000";
  tmp_0023 <= run_i_1 + 1;
...
  process(clk)
  begin
    if clk'event and clk = '1' then
      if reset = '1' then
        run_i_1 <= X"00000000";
      else
        if S_run = S_run_S_run_0010 then
          run_i_1 <= tmp_0022;
        elsif S_run = S_run_S_run_0008 then
          run_i_1 <= tmp_0023;
        end if;
      end if;
    end if;
  end process;
...
```

コンパイルの例 - モジュール



BlinkLED.java

```
1  package blink_led;
2
3  public class BlinkLED {
4
5      public boolean led;
6
7      public void run(){
8          while(true){
9              led = true;
10             for(int i = 0; i < 5000000; i++) ;
11             led = false;
12             for(int i = 0; i < 5000000; i++) ;
13         }
14     }
15
16 }
17
```

```
entity blink_led_BlinkLED is
  port (
    clk : in std_logic;
    reset : in std_logic;
    field_led_output : out std_logic;
    field_led_input : in std_logic;
    field_led_input_we : in std_logic;
    run_req : in std_logic;
    run_busy : out std_logic
  );
end blink_led_BlinkLED;
```

次バージョン:スケジュール表による合成

- ✔ リソース割当，実行サイクルを表で管理

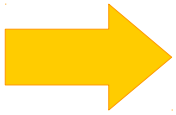
	ADD	SUB	MUL	DIV	FADD	FSUB	FMUL	...
0	c = a + b							
1		c = a - b						
2			c = a * b					
3				c = a / b				
4					fc=fa+fb			
5						fc=fa-fb		
6							fc=fa*fb	
...								

最適化 = 表を小さくする

+α Synthesijer.scala

Synthesijer.scala

- Javaで全部書けない...とはいえ
- もう普通のHDL(= VHDL/Verilog HDL)は書きたくない
  - たくさんの似たような名前の変数の定義
  - リセット，パルス信号をもとに戻すの忘れる
  - などなど
- 制御構造とデータ処理が同じ，な設計モデルはいやだ
- 記述の再利用をしたい
  - データ処理とデータ代入をわけたい



もっと高級なHDLがあればいい

Synthesijer.scala とは

- Synthesijerのバックエンドを使ってScala“で”HDLを書く
  - signal, port: 状態を変更可能なオブジェクト
  - expr: 副作用なしの式
  - sequencer: 状態遷移機械
  - module: モジュール全体
- 上記のオブジェクトをScalaでインスタンス化．つなぎ合わせる．

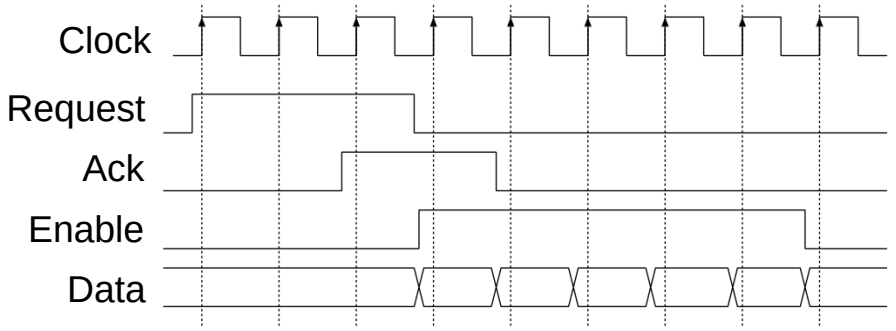
Synthesijer.scala の例(1)

- ✔ Lチカ

```
 def generate_led() : Module = {
   val m = new Module("led")
   val q = m.outP("q")
   val counter = m.signal("counter", 32)
   q <= m.expr(Op.REF, counter, 5)

   val seq = m.sequencer("main")
   counter <= (seq.idle, VECTOR_ZERO)
   val s0 = seq.idle -> seq.add()
   counter <= (s0, m.expr(Op.+, counter, 1))
   return m
 }

def generate_sim(target:Module, name:String) : SimModule = {
  val sim = new SimModule(name)
  val inst = sim.instance(target, "U")
  val (clk, reset, counter) = sim.system(10)
  inst.sysClk <= clk
  inst.sysReset <= reset
  return sim
}
```

Synthesijer.scala の例(2)

```
val m = new Module("UPLTest")
val uplin = new UPLIn(m, "pI0", 32)
val uplout = new UPLOut(m, "pO0", 32)

val ipaddr = m.inP("pMyIpAddr", 32)
val port = m.inP("pMyPort", 16)
val server_addr = m.inP("pServerIpAddr", 32)
val server_port = m.inP("pServerPort", 16)

def wait_trigger():State = ...

val ack_ready = m.expr(Op.==, uplout.ack, Constant.HIGH)
def wait_ack_and_send_data():State = ...

def send_dest_addr():State = ...

def send_port():State = ...

def send_length():State = ...

def send_data():State = ...

(idle -> (m.expr(Op.==, trigger, Constant.HIGH), wait_trigger())
      -> (ack_ready, wait_ack_and_send_data())
      -> send_dest_addr()
      -> send_port()
      -> send_length()
      -> send_data() -> idle)
```



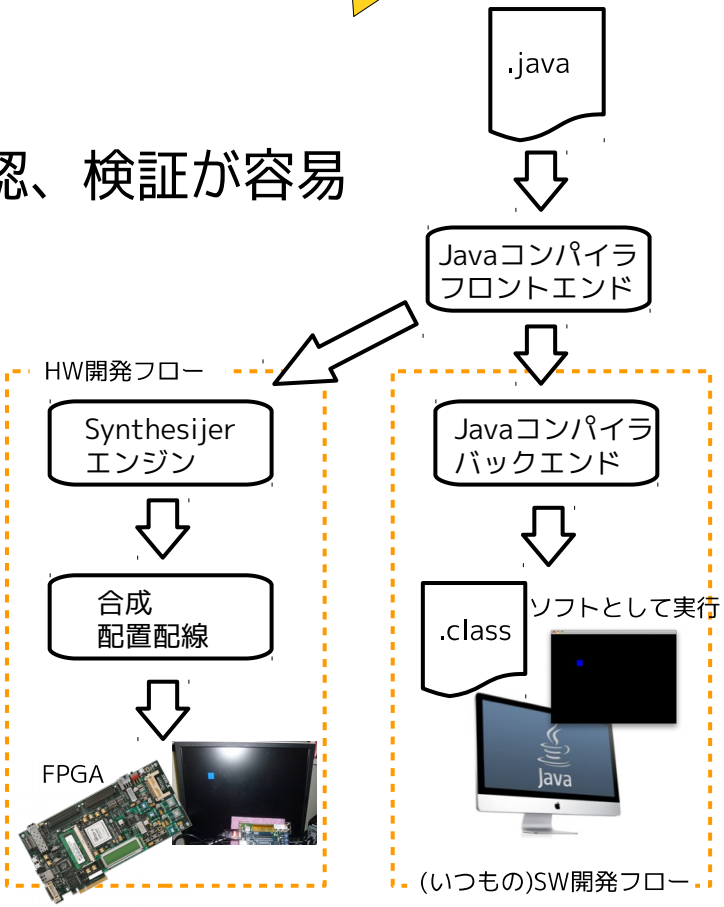
UPL(e-trees.Japan, Incの制御付きFIFO)

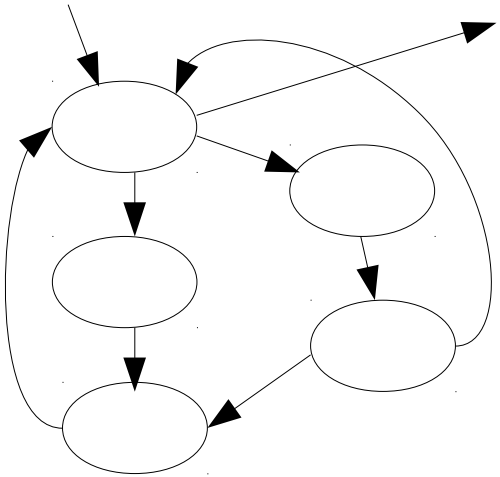
まとめと今後のロードマップ

Synthesijer - まとめ -

- JavaプログラムをFPGA上のハードウェアに変換
  - 複雑なアルゴリズムのハードウェア実装を楽に
  - オブクジェクト指向設計による再利用性の向上
- 特殊な記法，追加構文はない
  - ソフトウェアとして実行可能．動作の確認、検証が容易
  - 書けるプログラムに制限は加える
    (動的なnew，再帰は不可など)

Open-source





```
while(){
if(...){
  …
}else{
  …
  …
}
….
}
```

複雑な状態遷移も，Javaの制御構文を使って楽に設計できる

同じJavaプログラムをソフトウェアとしても
FPGA上のハードウェアとしても実行可能

ロードマップ

手続き型言語的基本演算, 操作	整数プリミティブ型変数の利用	算術, 論理, シフト演算（除算・剰余算以外）	制御構文（分岐, ループ）	メソッド呼び出し	除算・剰余算	浮動小数点数演算		
オブジェクト指向的インスンタンス協調	finalでのインスタンスの生成	インスタンスメソッドの呼び出し	インスタンス変数へのリードアクセス	インスタンス変数へのライトアクセス	インスタンス間での配列読み書きのサポート	インスタンスの配列, 配列の配列のサポート	コンストラクタのサポート	インスタンスのチェインアクセス
並列化パフォーマンス	基本ブロック内自動並列化	Threadによる明示的な処理の並列化をサポート	ループ内パイプライニング					
ライブラリ	整数プリミティブ変数の配列	AXI接続用のコンポーネントライブラリの提供	Stringクラスのサポート					
ユーザビリティ	コマンドラインでのコンパイル	HDLモジュールとのバインディング機構	FPGA合成ツールとの連携, 統一的な開発フローの提供	スケジューリング可視化ツール	HDL生成ライブラリを活用したDSLの提供			

参考

- http://synthesijer.sourceforge.net
  - リソース一式，クイックスタートガイドなど
- http://labs.beatcraft.com/ja/index.php?Synthesijer
  - Altera DE0-nanoでのサンプルの動作手順など
    (ビートクラフト様)
- http://wasa-labo.com/wp/
  - わさらぼ ブログ – 開発状況，Tipsなど
- http://javarock.sourceforge.net
  - Synthesijer の前身であるJavaRockプロジェクト

サポートについて

- バグ，機能追加要望はSourceforgeのBugTrackにお願いします

- Synthesijerについての有償サポートを希望される場合は，

  株式会社アックス様までお問い合わせください．

http://www.axe-inc.co.jp

参考

- http://synthesijer.sourceforge.net
  - リソース一式，クイックスタートガイドなど
- http://labs.beatcraft.com/ja/index.php?Synthesijer
  - Altera DE0-nanoでのサンプルの動作手順など
    (ビートクラフト様)
- http://wasa-labo.com/wp/
  - わさらぼ ブログ – 開発状況，Tipsなど
- http://javarock.sourceforge.net
  - Synthesijer の前身であるJavaRockプロジェクト