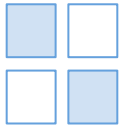
2015年8月28日 14:30-15:50 (s5c)
SWEST17@下呂温泉水明館
(第38回SIGEMB招待講演)

Pythonによる高位設計フレームワークPyCoRAMでFPGAシステムを開発してみよう

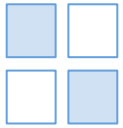
高前田 伸也
奈良先端科学技術大学院大学 情報科学研究科
E-mail: shinya_at_is_naist_jp

今日のテーマ



- 「好きな道具」で「好きなもの」を作る世界
  - 発表者の場合
    - 好きな道具: 好きな言語（Python）・モデル・書き方
    - 好きなもの: FPGAを使ったカスタムコンピュータ
- 「こんな風に設計できたらいいな」を実現しよう
  - 自分の手になじむ道具を実際に作ろう、そして公開しよう
  - 発表者の場合
    - PyCoRAM: PythonによるIPコア高位設計フレームワーク
    - Pyverilog: Verilog HDL解析・生成ツールキット
    - Veriloggen: PythonでVerilog HDLを組み立てる軽量ライブラリ

All software are available!



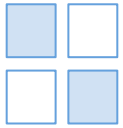
- GitHubで公開中
  - PyCoRAM: https://github.com/PyHDI/PyCoRAM
  - Pyverilog: https://github.com/PyHDI/Pyverilog
  - Veriloggen: https://github.com/PyHDI/veriloggen

```
$ git clone https://github.com/PyHDI/Pyverilog.git
$ git clone https://github.com/PyHDI/PyCoRAM.git
$ git clone https://github.com/PyHDI/veriloggen.git
```

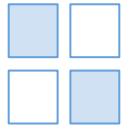
- PIP（Pythonパッケージ管理）でもインストール可
  - ただしGitHubが最新版です

```
$ pip install pyverilog
$ pip install pycoram
$ pip install veriloggen
```

目次

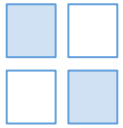


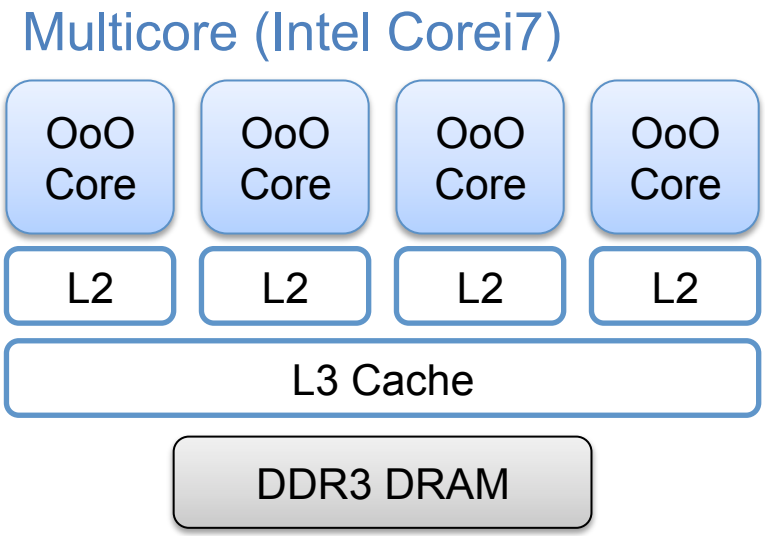
- FPGAとは？
  - FPGAの基礎
  - 近年のFPGAを取り巻く環境
- 高位合成ツールとは？
  - 高位合成の基礎
  - 商用ツールとオープンソースなツール
- Pythonによる高位設計フレームワークPyCoRAM
  - PythonとVerilog HDLを組み合わせたIPコア設計ツール
  - 関連ツールの紹介: Pyverilog, Veriloggen
- 今後の高位合成ツールはどうあるべきか？
  - まじめな話とフランクは話
- まとめ

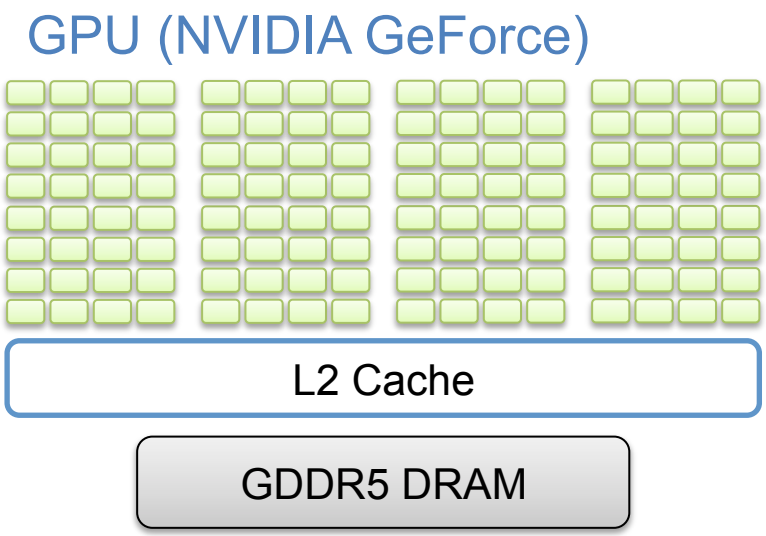


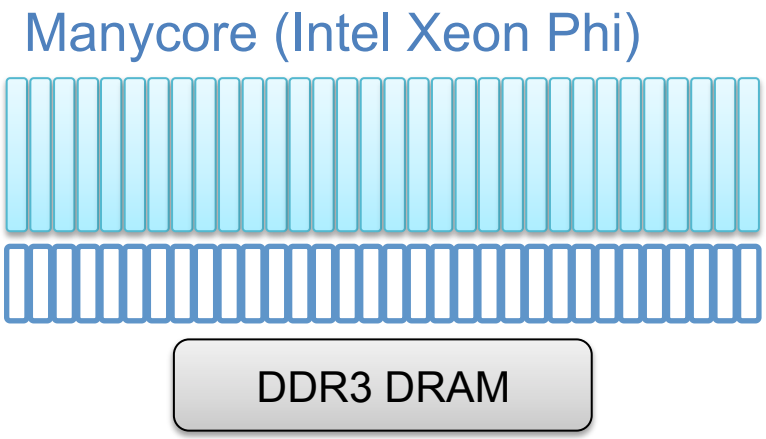
FPGAとは？

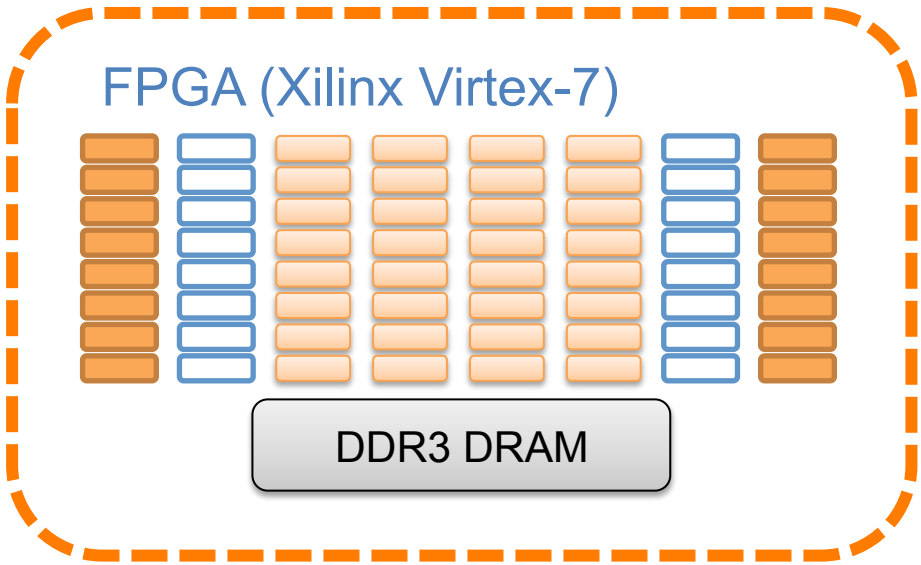
ヘテロジニアスコンピューティング



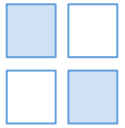




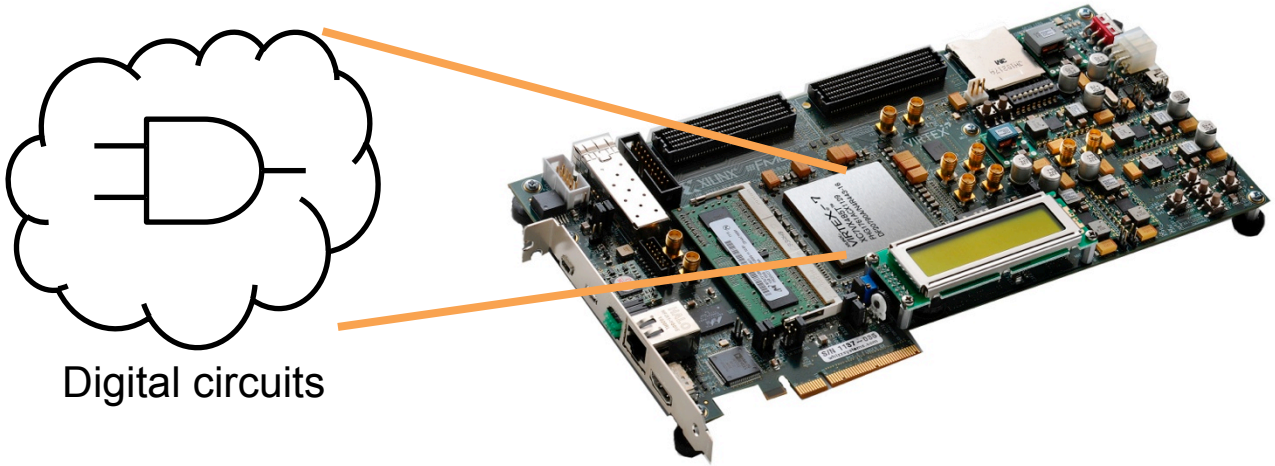




FPGA (Field Programmable Gate Array)

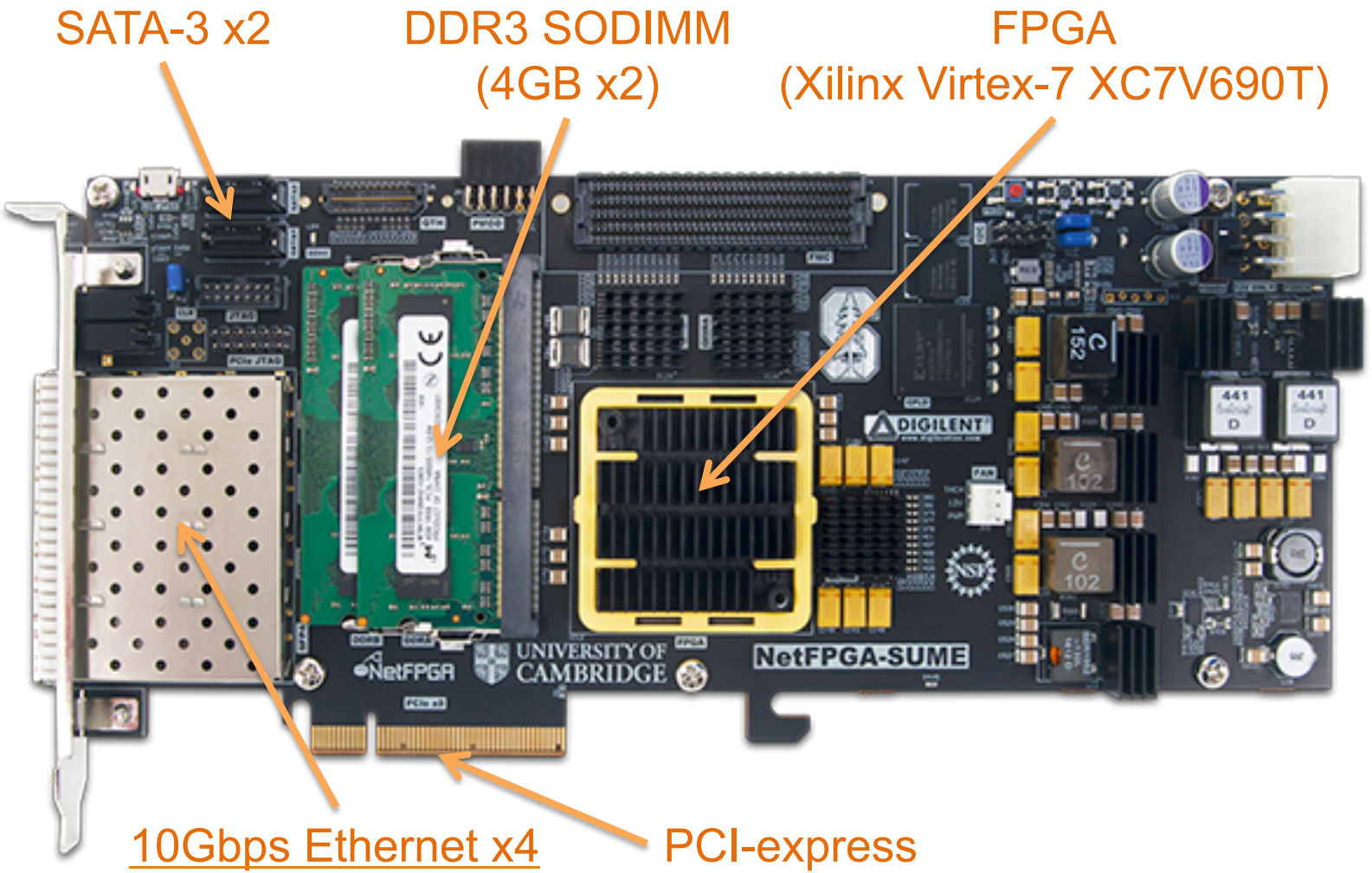


- 中身を改変可能なLSI
  (PLD: Programmable Logic Device)
  - 設計者が独自のデジタル回路を形成することができる
  - 対してCPUやGPUはFLD (Fixed Logic Device)？
- CPUなどとの大きな違いは？
  - CPUの振る舞いはソフトウェア（プログラム）で規定される
    - ユーザはCPUそのものの回路を変更することはできない
  - FPGAの振る舞いはハードウェアそのものの変更で変えられる

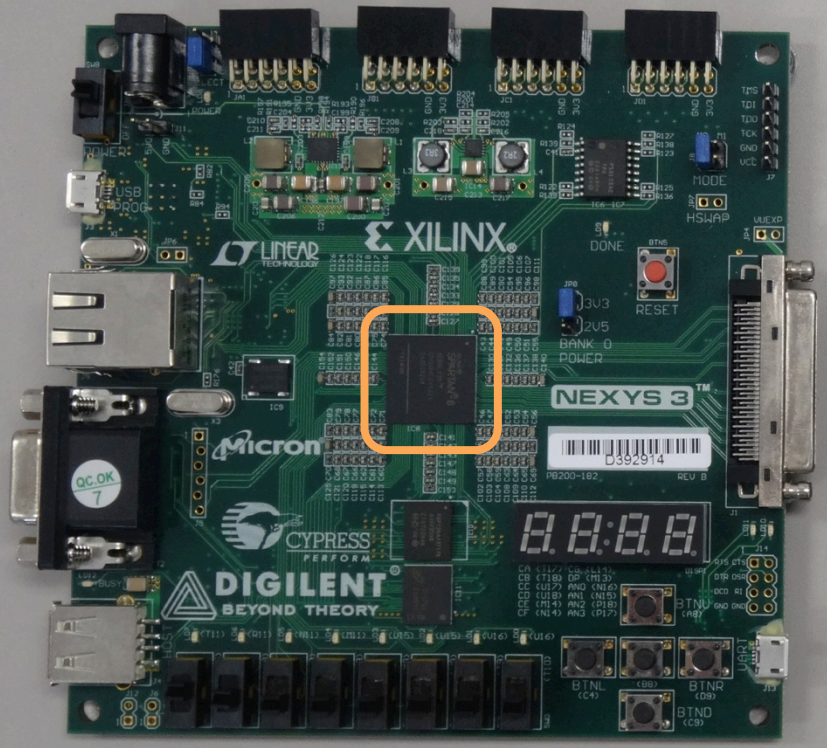


FPGAボードの基礎

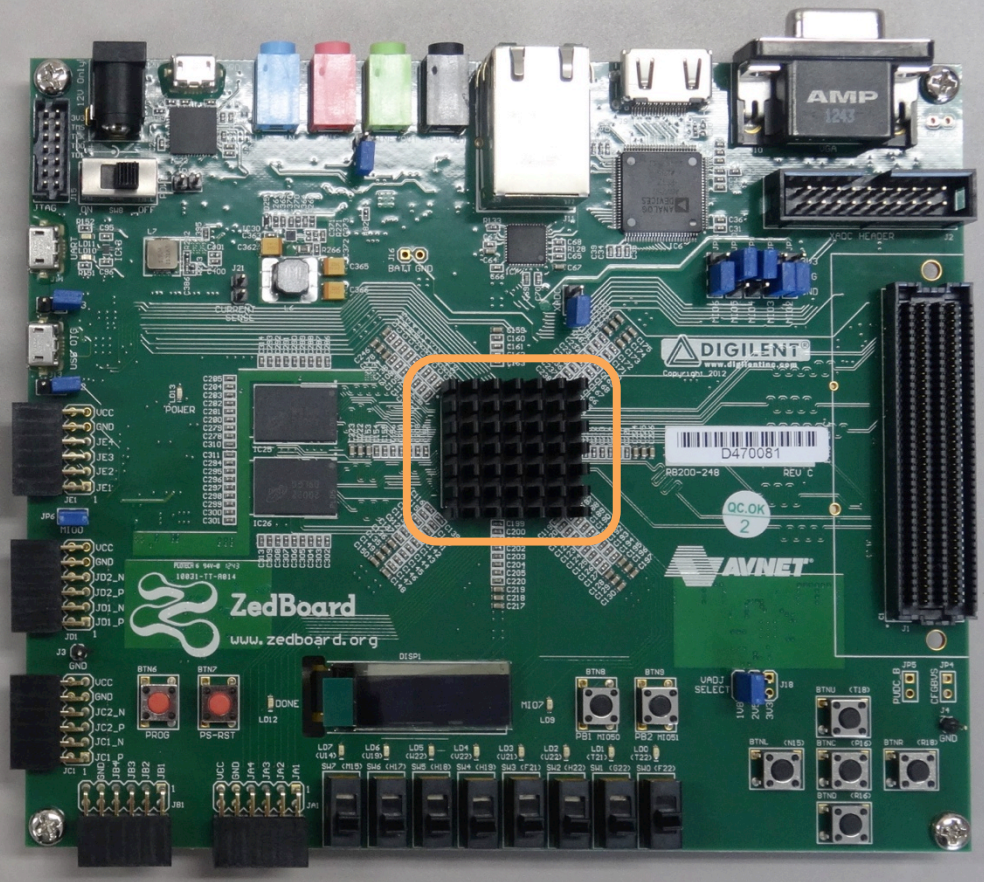
Digilent NetFPGA SUME
Price $24,500

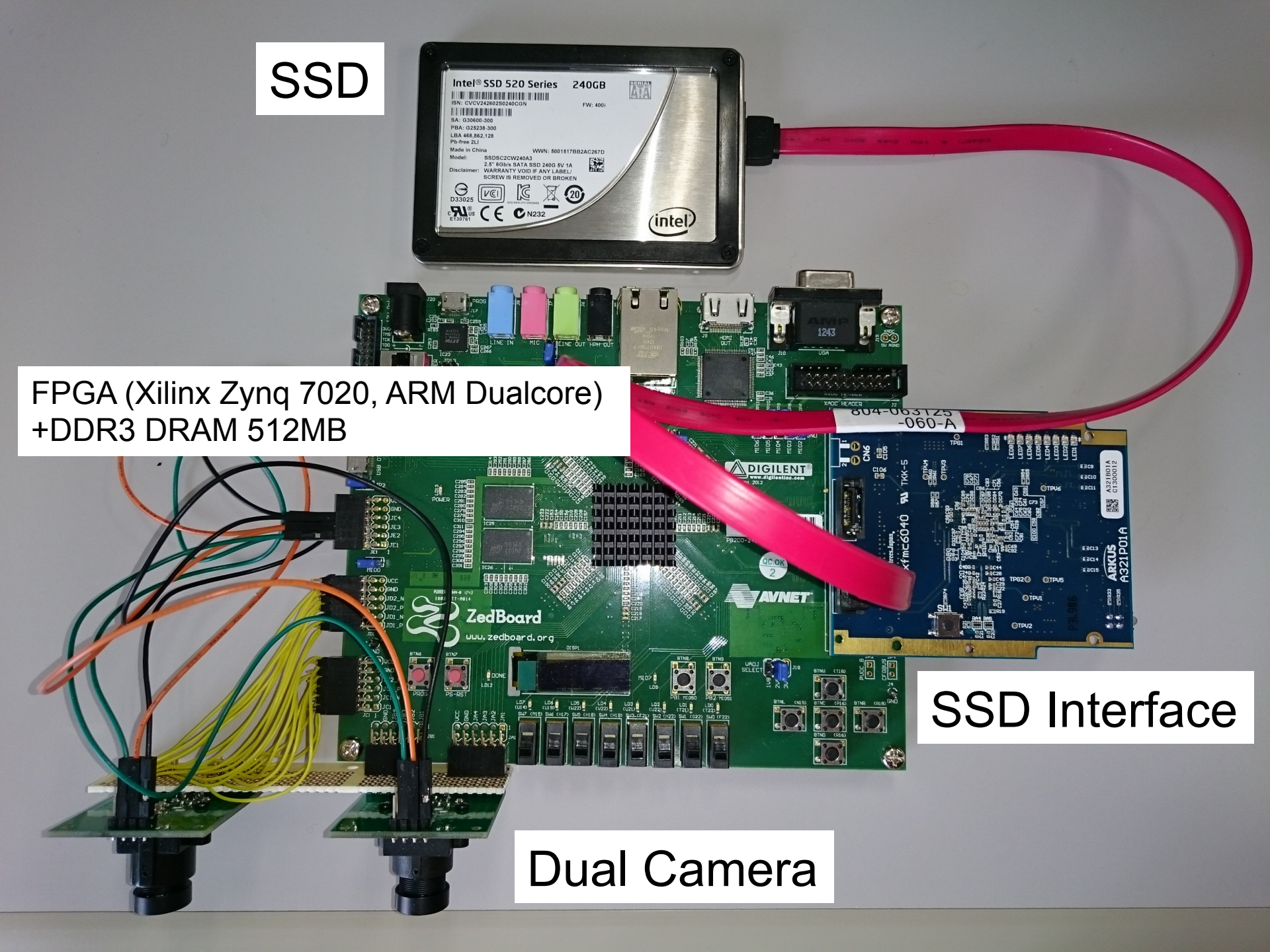


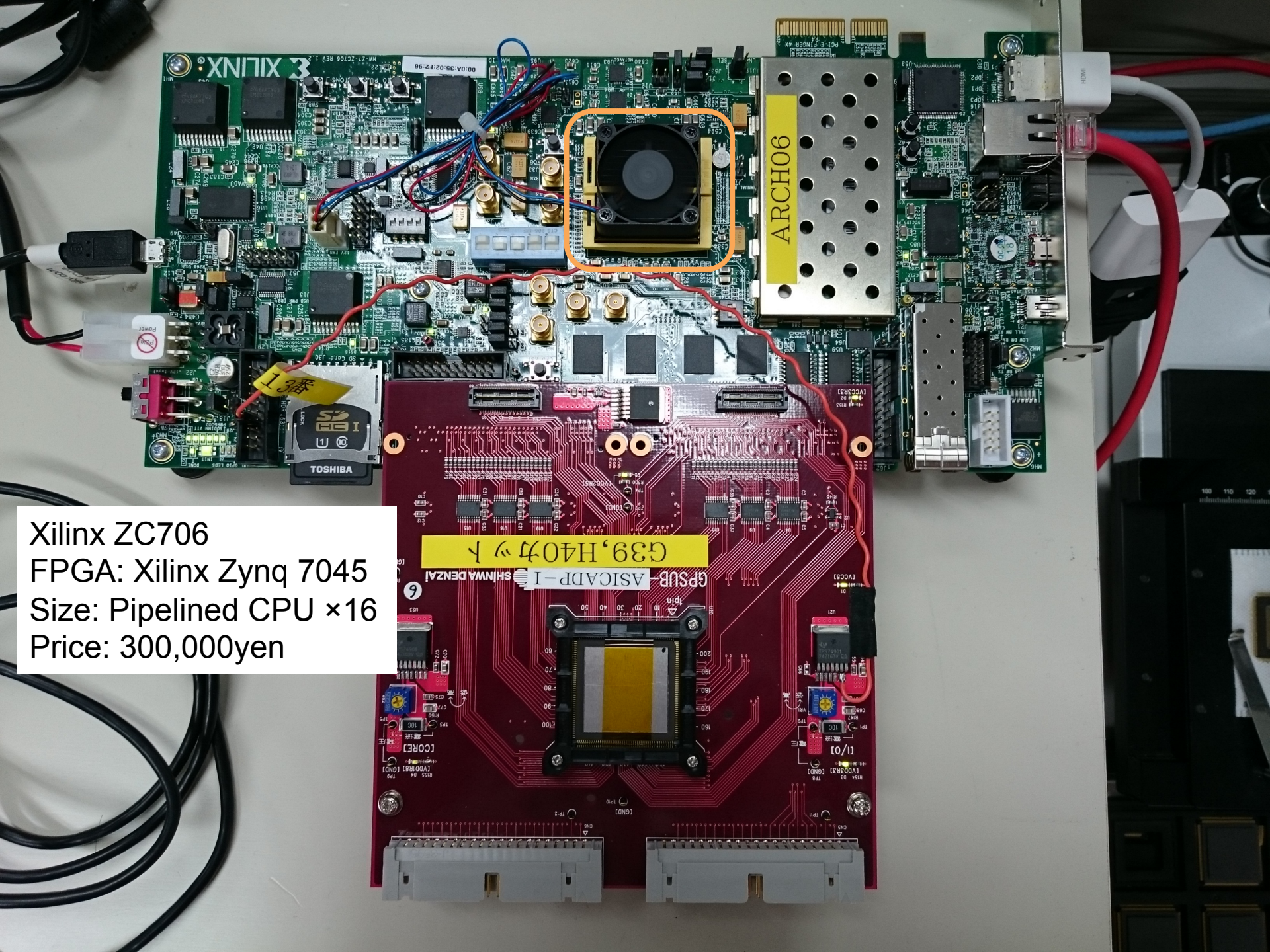
Digilent Nexys3
FPGA: Xilinx Spartan-6 LX16
Size: Pipelined CPU ×2
Price: 15,000yen (Academic)



Digilent ZedBoard
FPGA: Xilinx Zynq 7020
Size: Pipelined CPU ×8 (+ ARM DualCore)
Price: 60,000yen (Academic)

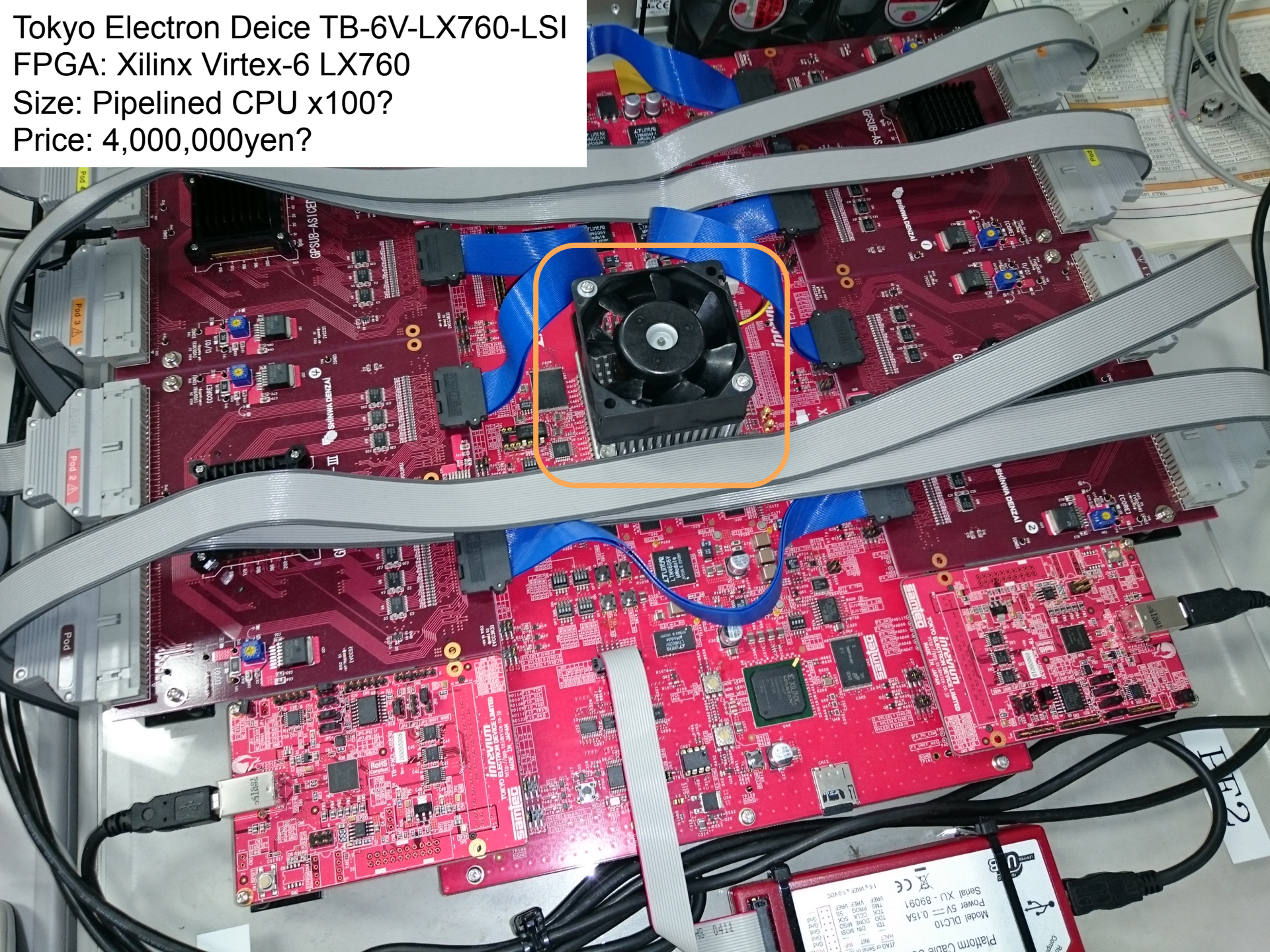


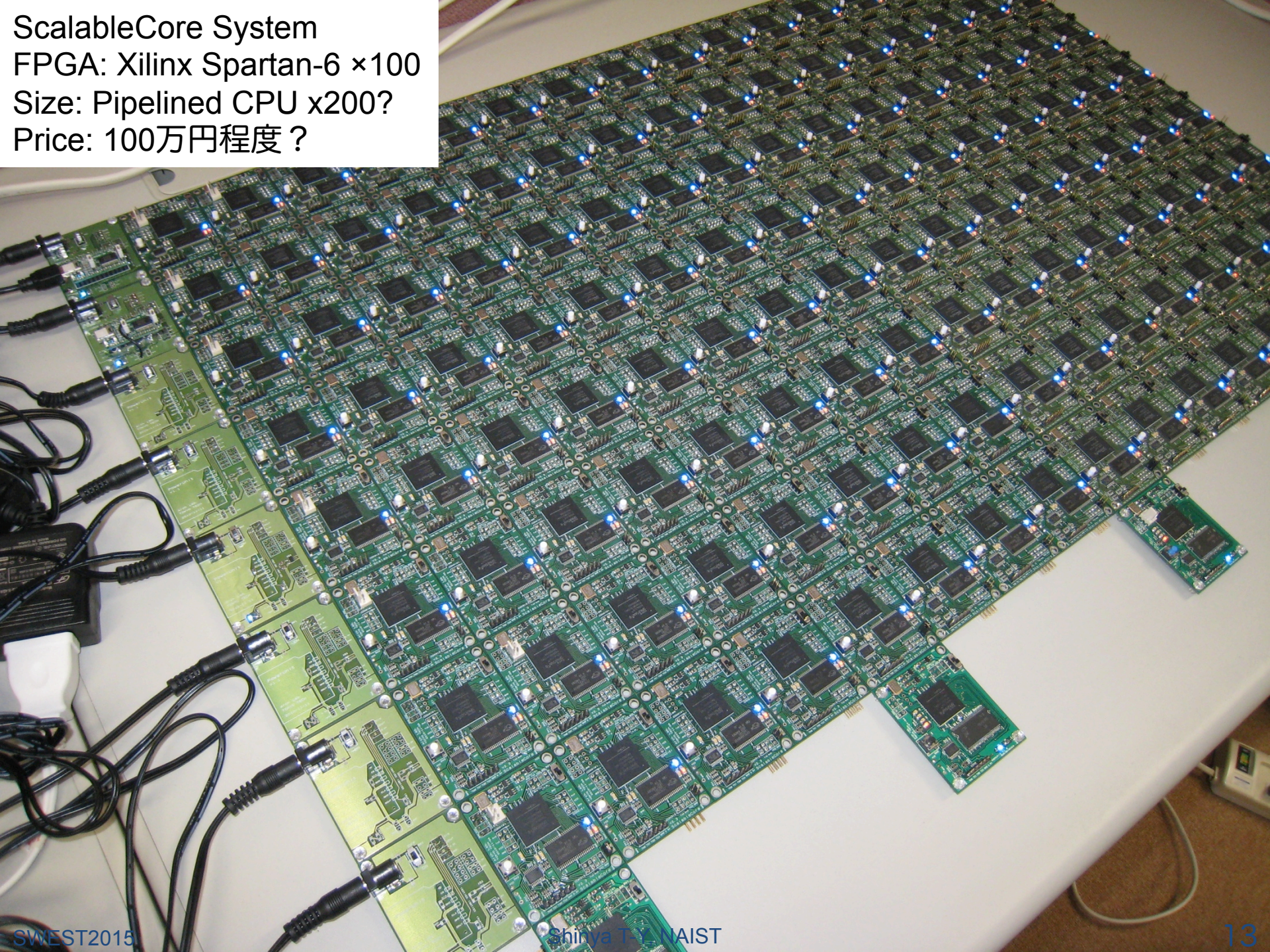

SSD
FPGA (Xilinx Zynq 7020, ARM Dualcore)
+DDR3 DRAM 512MB
SSD Interface
Dual Camera



Xilinx ZC706
FPGA: Xilinx Zynq 7045
Size: Pipelined CPU ×16
Price: 300,000yen

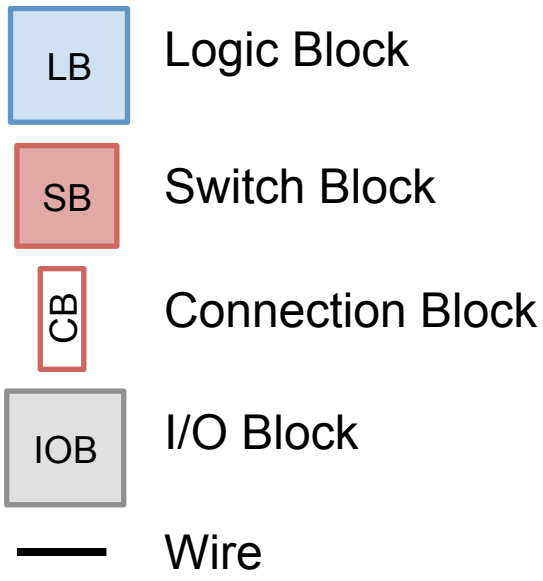
Tokyo Electron Deice TB-6V-LX760-LSI
FPGA: Xilinx Virtex-6 LX760
Size: Pipelined CPU x100?
Price: 4,000,000yen?

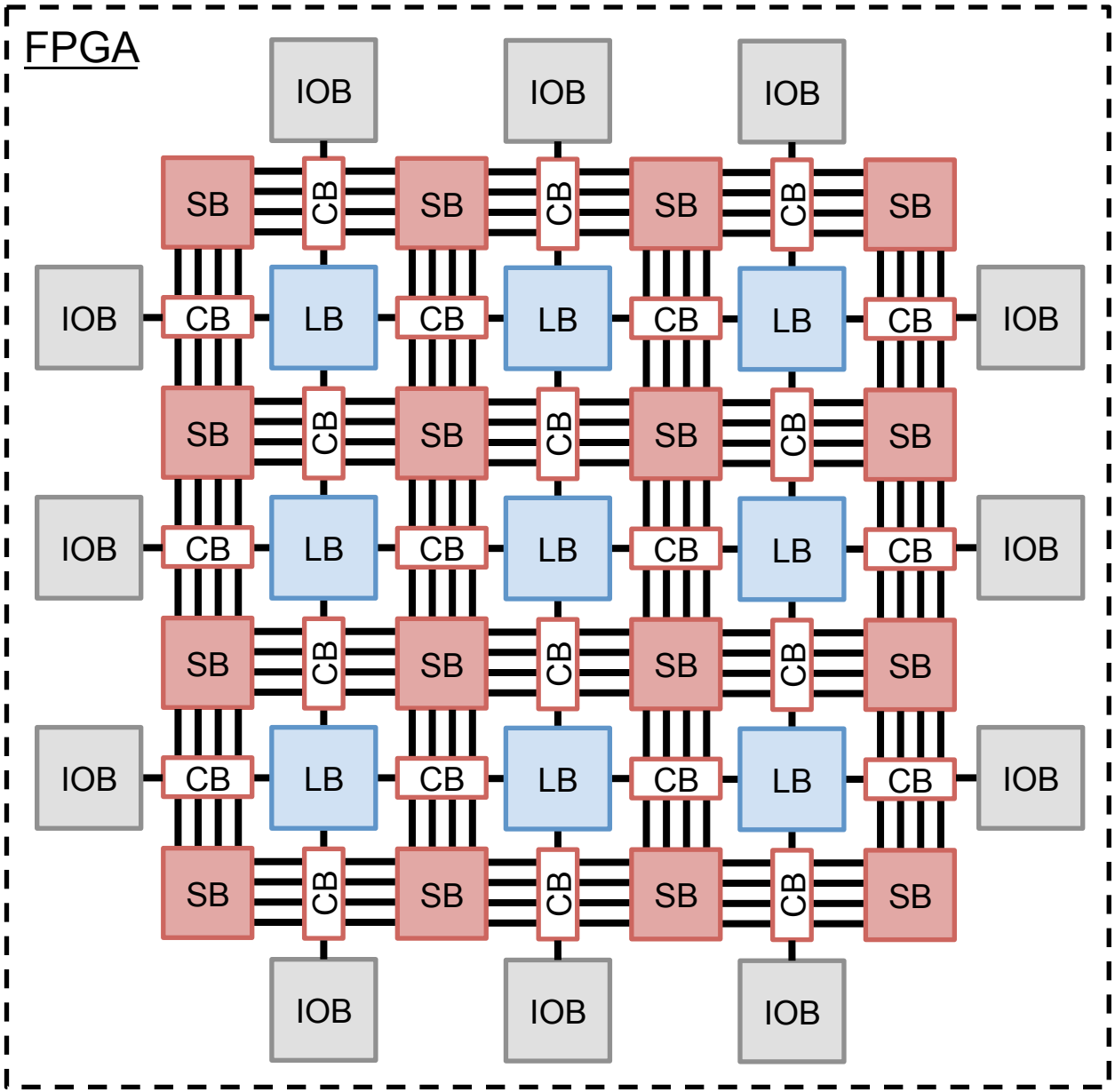



ScalableCore System
FPGA: Xilinx Spartan-6 ×100
Size: Pipelined CPU x200?
Price: 100万円程度？

アイランドスタイルFPGAの構成

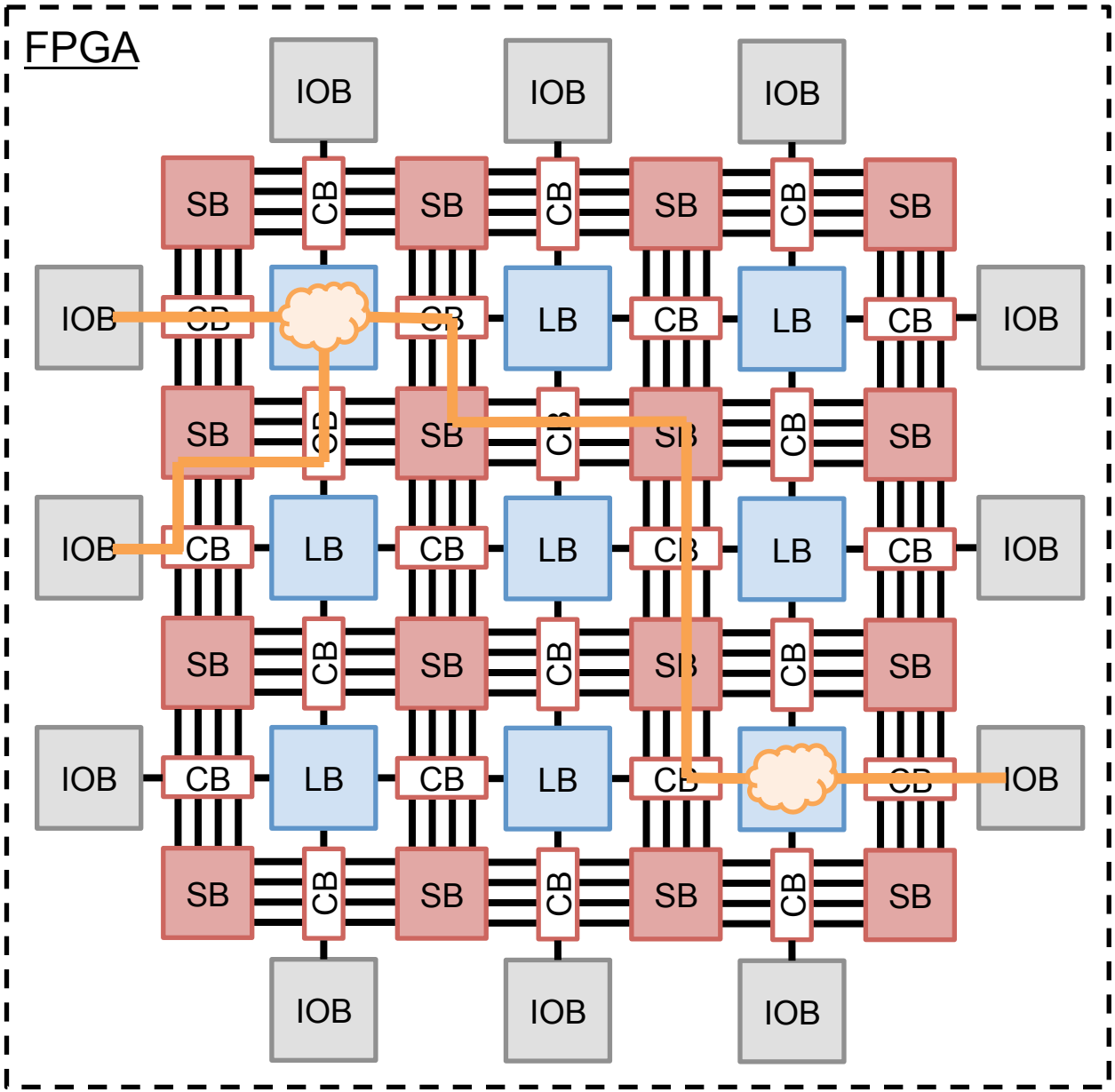
- An LB has logical circuit components for both combinational circuits and sequential circuits
- They are connected via interconnection components (SB, CB and wire)





アイランドスタイルFPGAの構成

- An LB has logical circuit components for both combinational circuits and sequential circuits
- They are connected via interconnection components (SB, CB and wire)



LB Logic Block

SB Switch Block

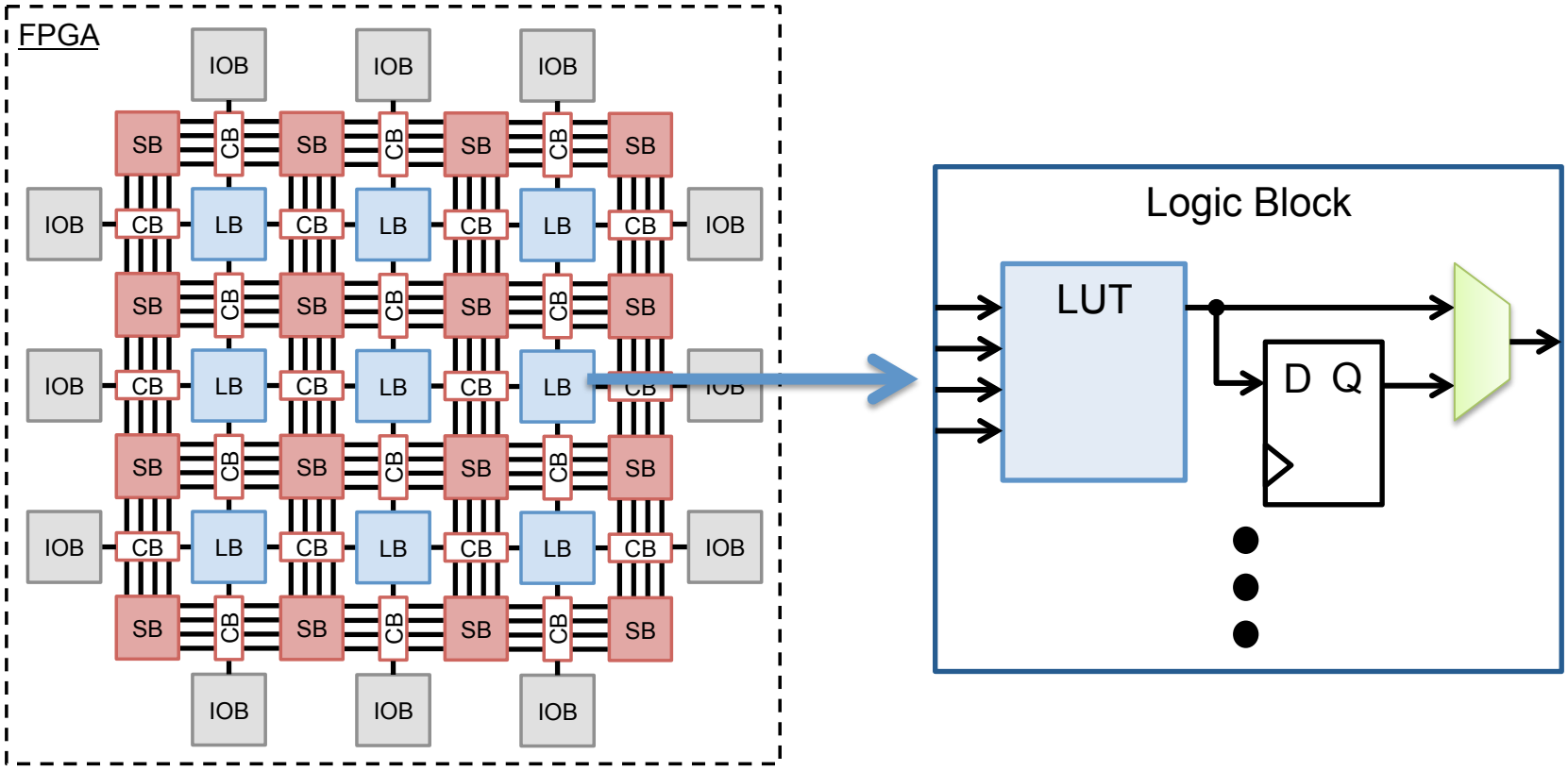
CB Connection Block

IOB I/O Block

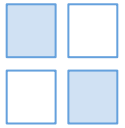
Wire

Logic Block

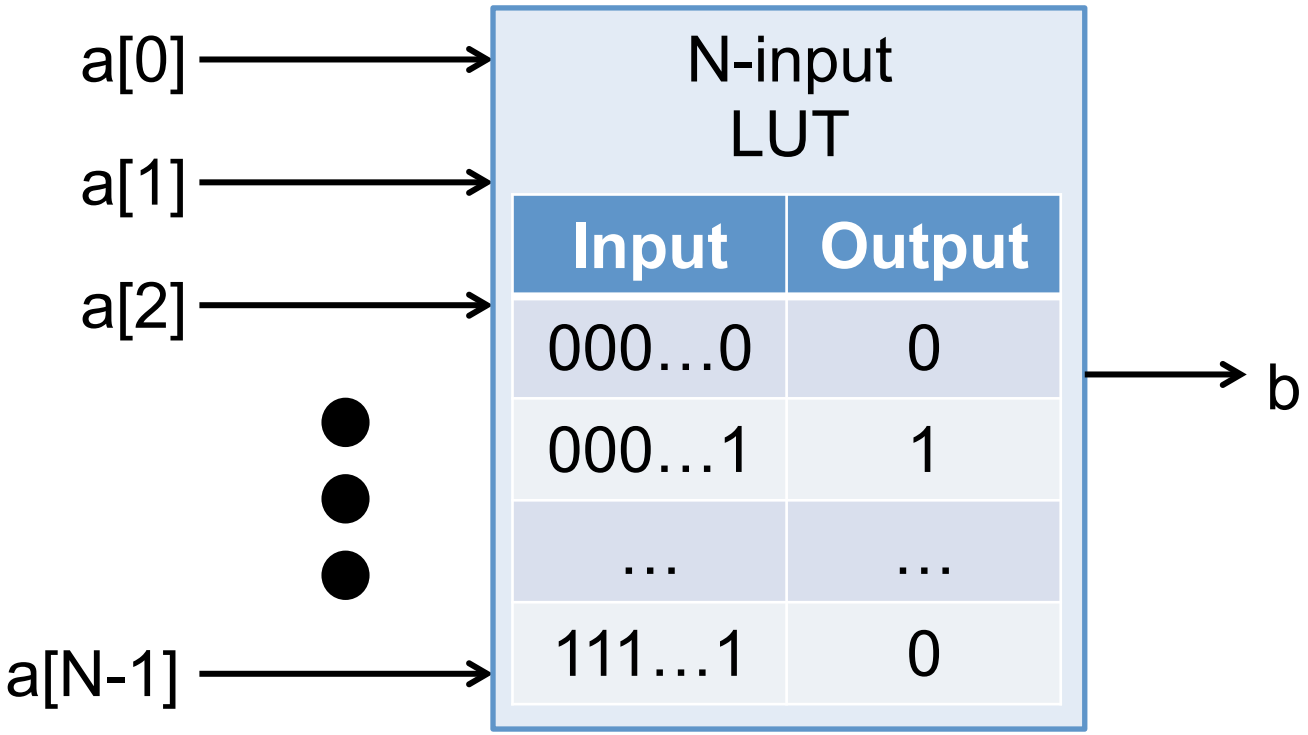
- Two basic elements in a logic block
  - LUT (Look Up Table): for combinational circuits
  - Flip-flop: for memory (sequential circuits)



LUT: Look Up Table

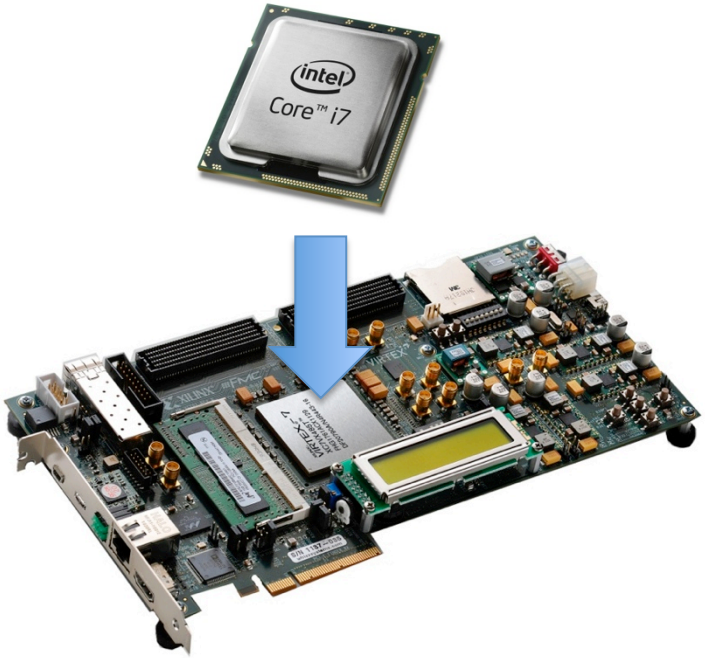


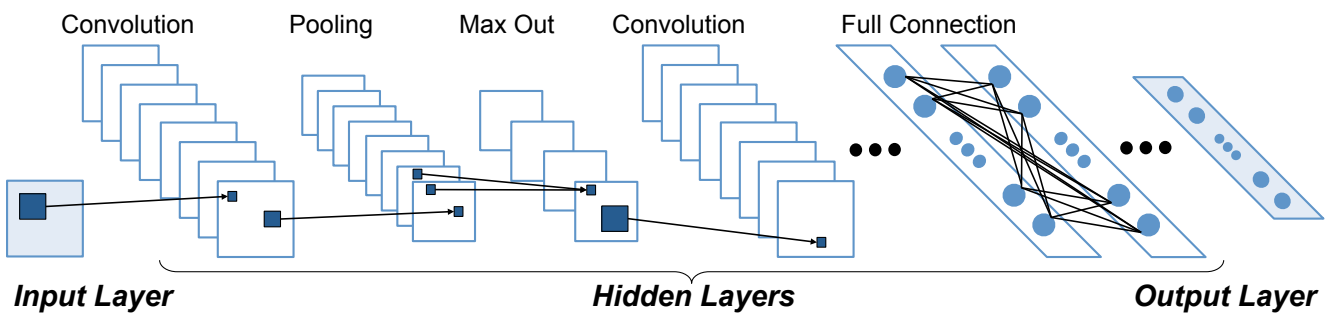
- LUTs realize combinational logics
- An LUT returns a 1-bit value corresponding to the input bit-vector (=Boolean function)
  - N-input LUT has 2^N combinations of results: 4-input LUT has 16



FPGA in Anywhere

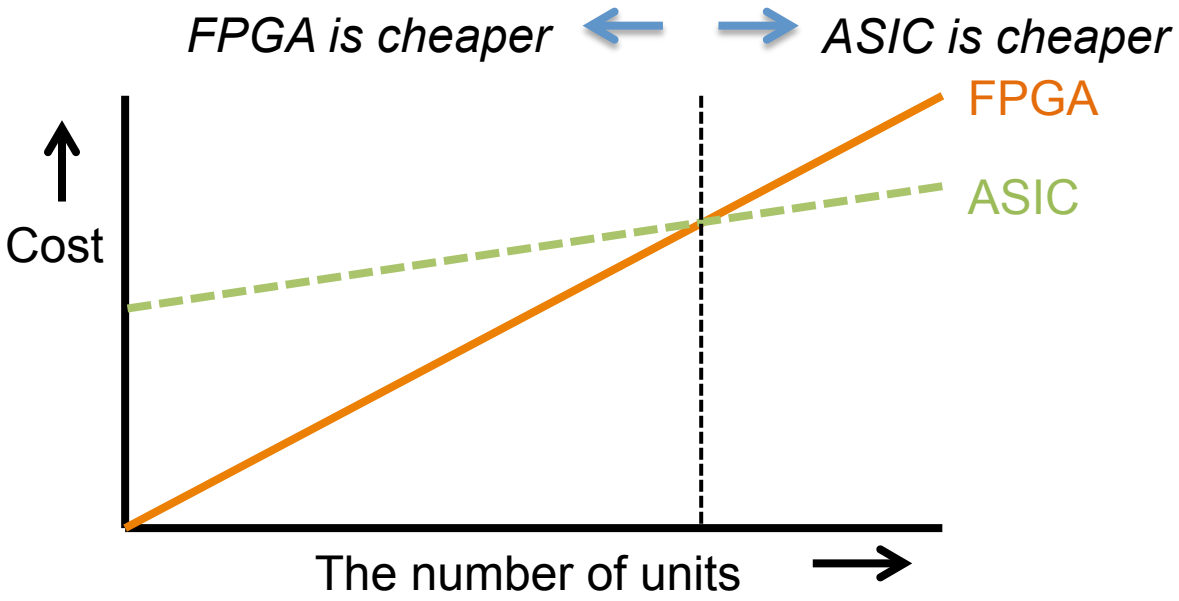
- LSI設計・検証のプラットフォームとして
  - LSIの機能を製造前に検証
  - HWの完成を待たなくても
    SWの開発検証ができる
  - ソフトウェアシミュレータによる
    回路シミュレーションよりも
    高速に検証できる
- 最終製品に組み込むLSIとして
  - 画像処理
  - ネットワーク機器
- アクセラレータとして
  - データベース
  - ウェブ検索
  - 証券取引
  - 機械学習





ASIC vs. FPGA

- ASIC (Application Specific Integrated Circuit)
  - それぞれのアプリケーションに特化した専用回路を設計
    - 専用パイプライン・高い周波数で高い性能
- FPGA (Field Programmable Gate Array)
  - どのアプリケーションも数種類のFPGAで実現：少量生産もOK
  - 製品リリース後の回路構成の改変が可能



増え続けるFPGAの回路規模

- トランジスタのスケーリングによりFPGの回路規模も増大
  - 5年で6倍以上に増加
  - 今後も更なる増加が予想
    - マルチダイ化
    - ３次元積層
- その分設計が大変に
  - 設計検証の時間が増大
  - 効率的な開発方式が必要
    - RTLのみの開発の限界
    - 高位合成処理系の利用

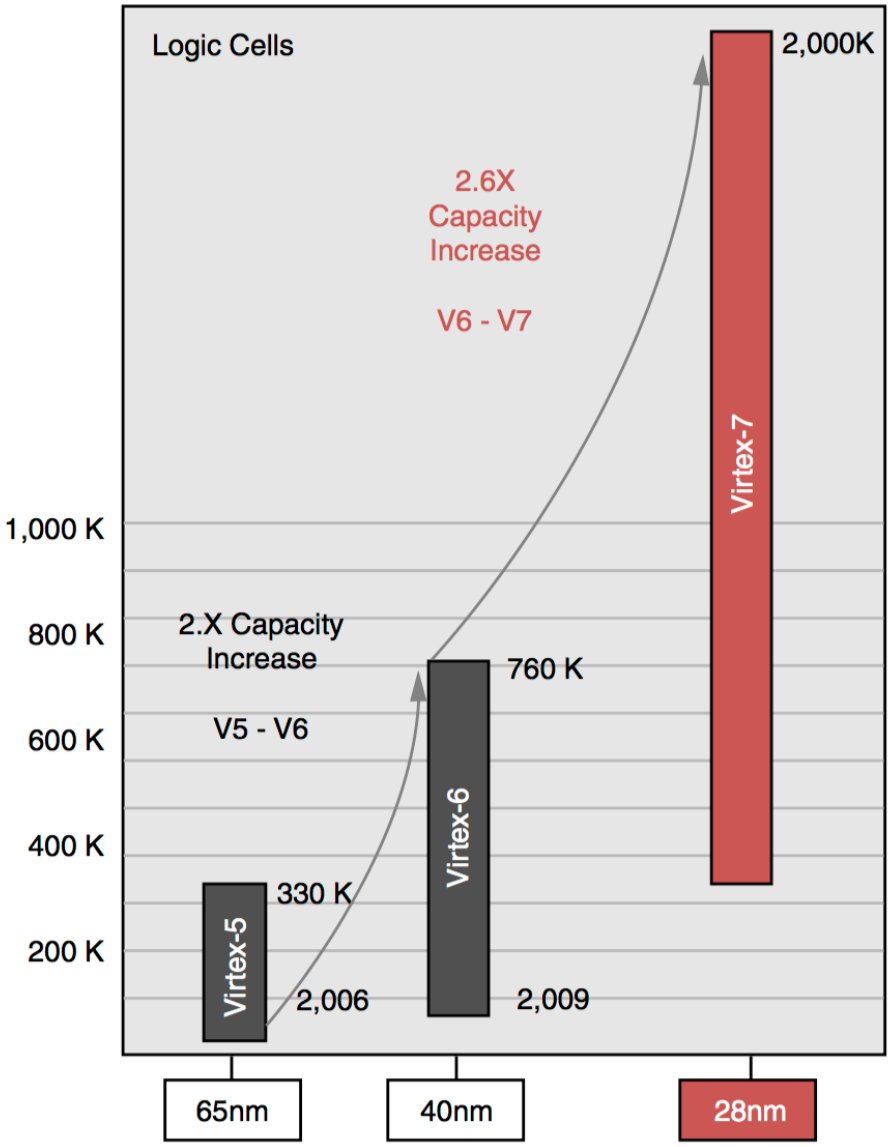
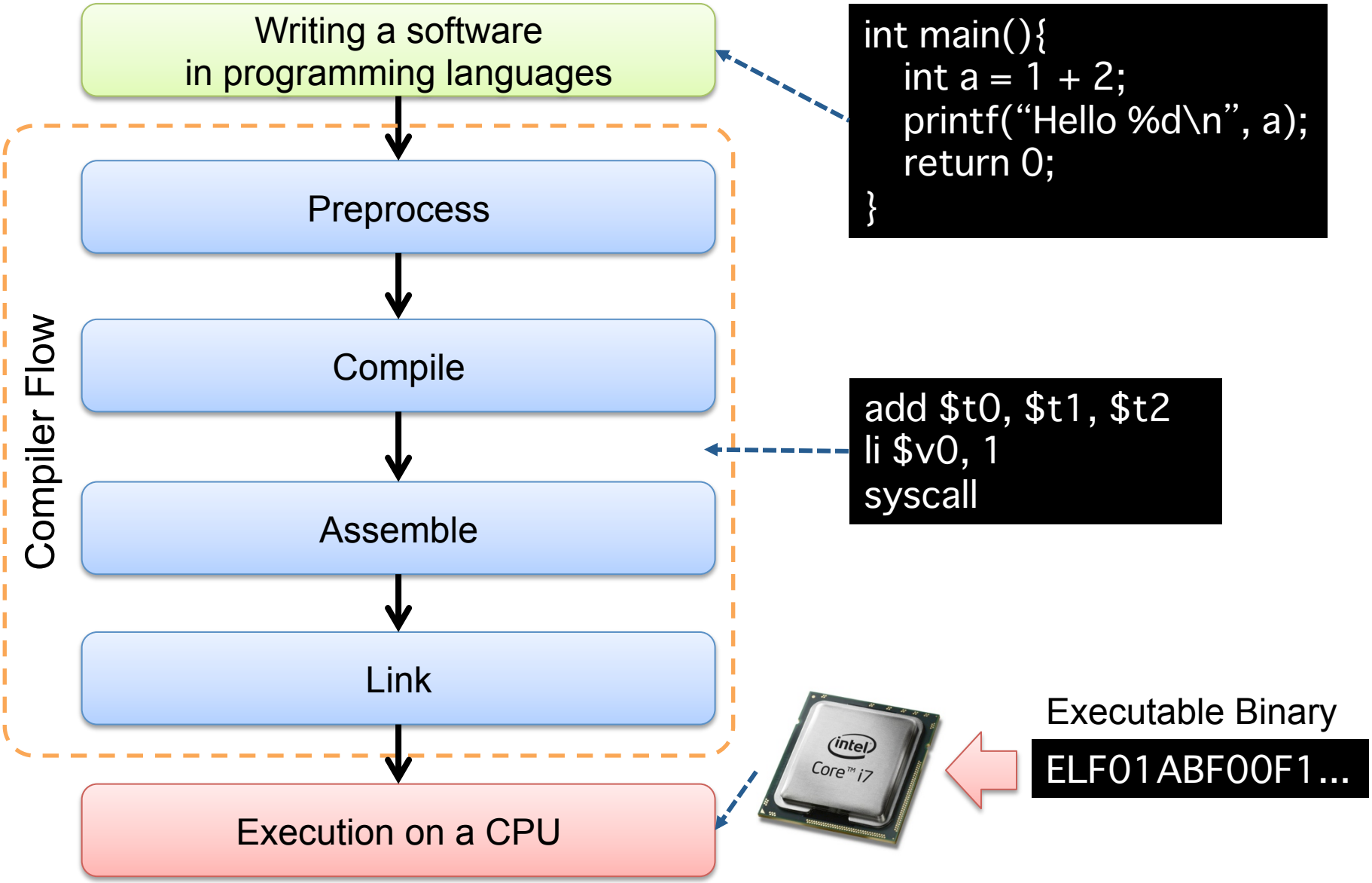


Figure 1-1: **FPGA capacity has increased over 6x in less than 5 years**

From Xilinx UG872

How to Develop a Software?



How to Develop a (FPGA) Hardware?

Writing a hardware design in HDL
(Hardware Description Language)

EDA Flow

Synthesis

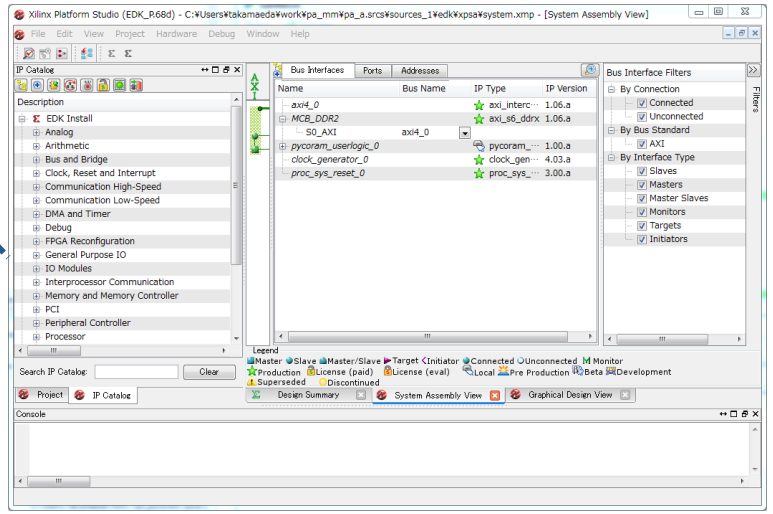
Technology Mapping

Place and Route

Bitstream Generation

Configuration of the bitstream
to an FPGA

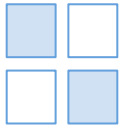
```
module top
  (input CLK, RST,
   output reg [7:0] LED);
  always @(posedge CLK) begin
     LED <= LED + 1;
  end
endmodule
```



Original HW on an FPGA

1A0C021E...

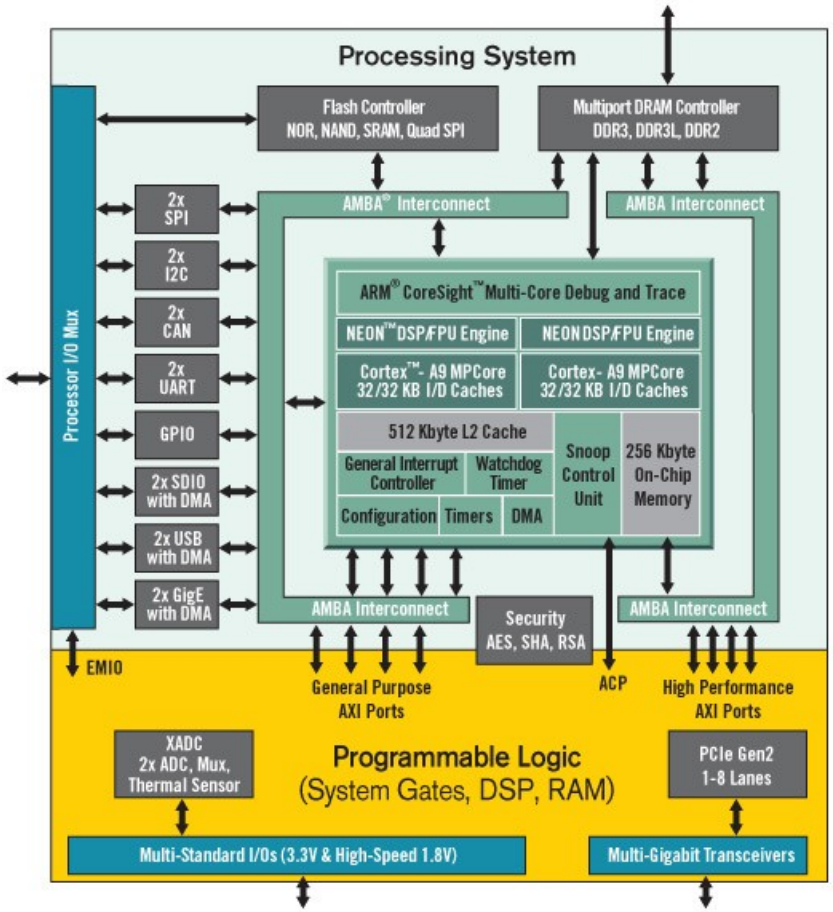
Bitstream

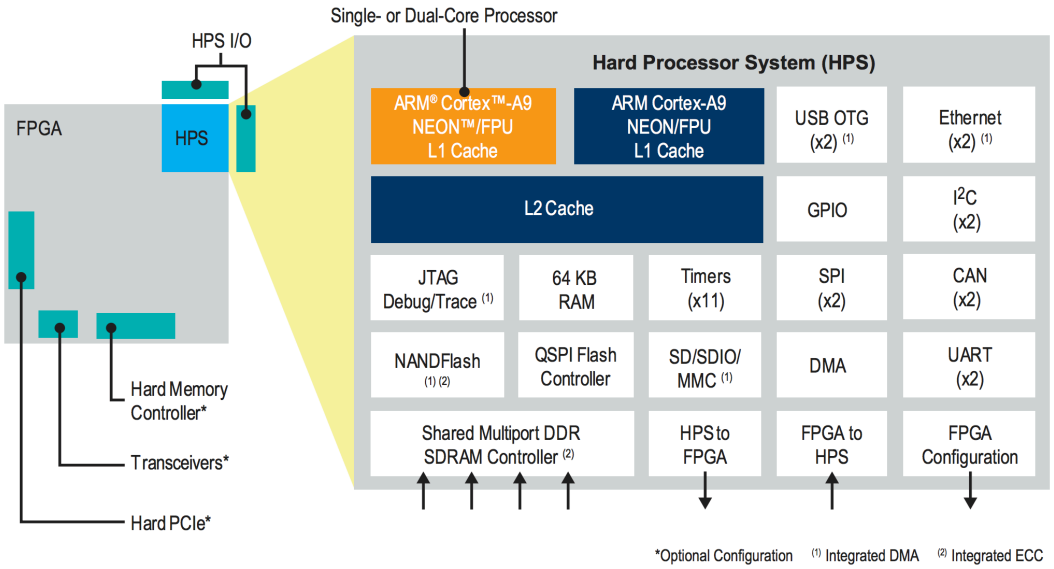


近年のFPGAを取り巻く環境

ARM搭載FPGAの登場 (1)

- ARMプロセッサ+FPGA (Xilinx Zynq, Altera SoC)
  - 専用インターコネクトで密結合，キャッシュ・DRAM共有
  - 普通のLinuxが動作する→大量なソフトウェア資源が利用可能

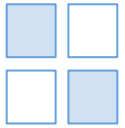




AlteraのARMベースSoC
https://www.altera.com/ja_JP/pdfs/literature/br/br-soc-fpga_j.pdf

Zynq-7000 All Programmable SoC
http://japan.xilinx.com/products/silicon-devices/soc/zynq-7000.html

ARM搭載FPGAの登場 (2)



- 普通のFPGAとは何が違う？
  - そもそも通常のFPGA上もCPU搭載可能（ソフトマクロ）
  - コア性能が数倍～10倍違う
    - MicroBlaze: 100MHz～200MHz, In-order, Single issue
    - ARM: 600MHz～1GHz, OoO, Super scalar
- 独自のHW/SW協調SoCの実現が容易になる
  - 面倒な処理は処理はCPU上のソフトウェアで実現
    - ネットワーク・ファイルシステムなど
  - 並列処理部はFPGA上の専用ハードウェアで実現
    - 並列度とメモリ帯域に応じて演算器を追加し高速化
    - 必要に応じて演算精度を削り更に高速化
  - CPU-FPGAロジック間のデータ共有は？
    - キャッシュ・DRAMをCPUとロジックで共有しているので簡単

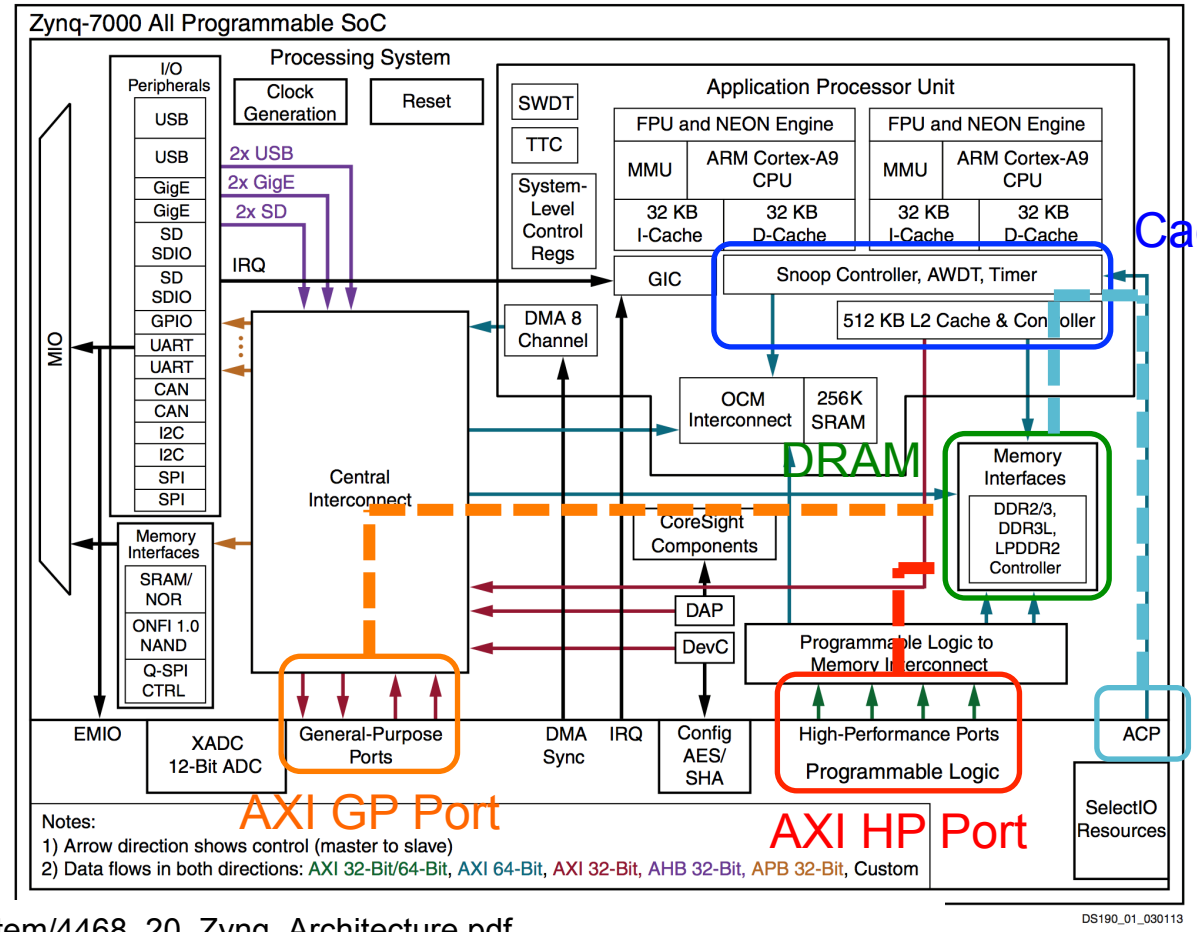
例）Zynq 7000 アーキテクチャ

- ARM Cortex-A9 (Dual-core, OoO, 8-stage)
- ３種類のCPU-PL間接続 (すべてAXIインターフェース)

GP
- 低速
- 制御レジスタ
  アクセス用

HP
- 高バンド幅
- DRAMへの
  バースト転送向け

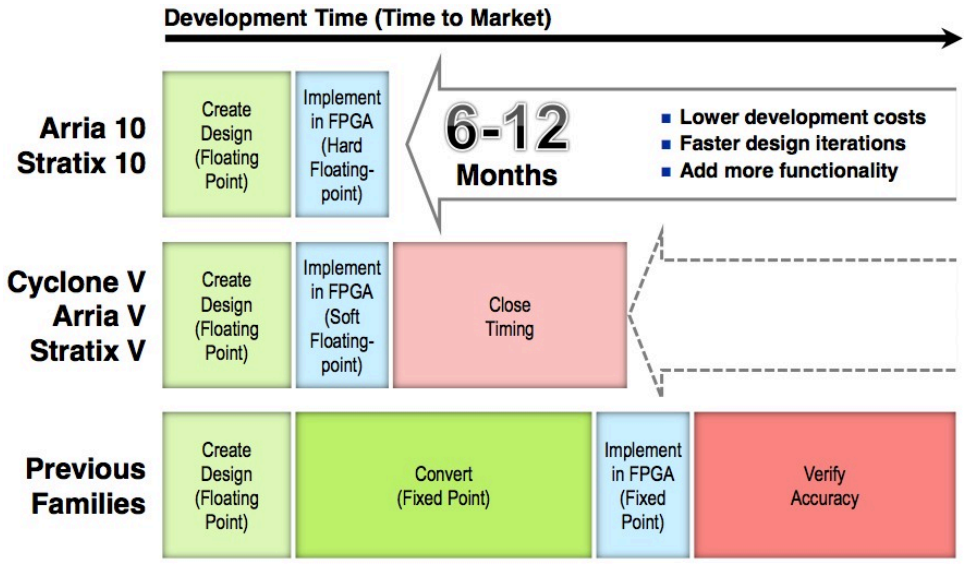
ACP
- 低レイテンシ
- キャッシュ
  コヒーレント
- CPUとの
  データ共有向け

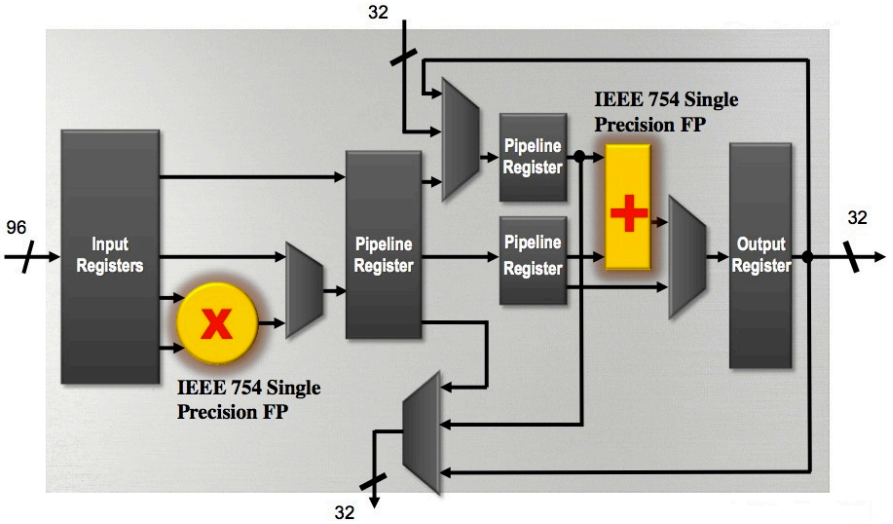


http://www.ioe.nchu.edu.tw/Pic/CourseItem/4468_20_Zynq_Architecture.pdf
http://japan.xilinx.com/support/documentation/data_sheets/j_ds190-Zynq-7000-Overview.pdf

浮動小数点ユニット搭載FPGA

- 従来FPGAのDSP（乗算）ユニットは整数のみ対応
  - 浮動小数点演算は変換ロジックを組み合わせて実現（ソフトマクロ）→大きな回路・電力オーバーヘッド
    - そのため浮動小数点演算ではGPUが有利だった
- Altera次期モデルがハードマクロ浮動小数点DSPを搭載
  - コンピューティングデバイスとしてのFPGAの利用が増加？

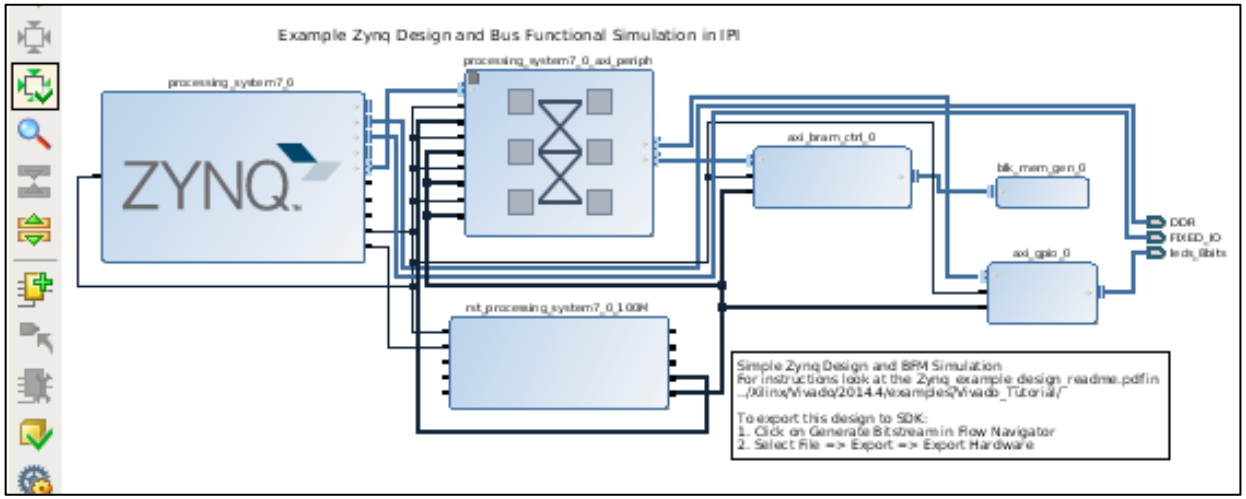




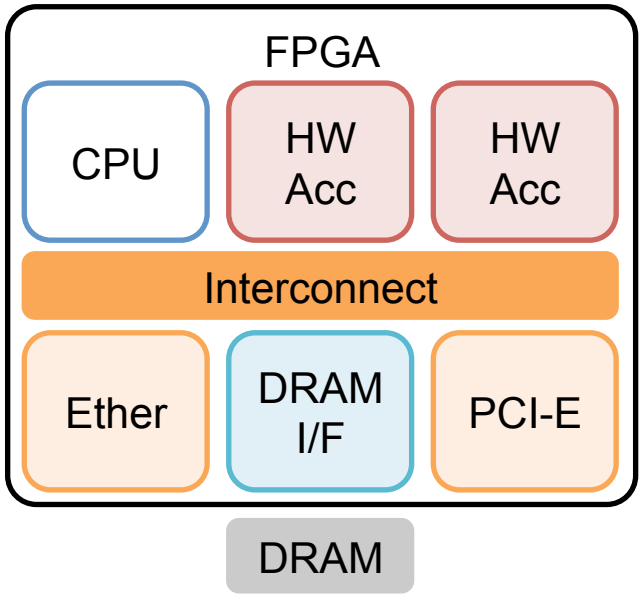
Altera Expands Floating-Point Hardware Support Across Its Product Lines
http://www.bdti.com/InsideDSP/2014/07/22/Altera

IPコアベースのシステム開発環境の普及

- IPコアを開発・追加して繋げばHW完成☺
  - 標準的なインターコネクトでIPコア達を接続
  - EDAツールが自動的にインターコネクトと（いくつかの）デバイス依存のインターフェースを生成してくれるため楽
  - いかにしてIPコアを簡単に開発するかが重要
    - 高位合成ツールの活用



Xilinx VivadoにおけるIPコアとARMの接続



アプリケーションの変化

- 以前は画像処理やネットワーキングなどが主流
- 「ビッグデータ」指向へ: 脱ノイマン型？
  - イーサネットNICでMemcached [Fukuda+, FPL'14]
  - Microsoft Bing search engine (Catapult) [Putnam+, ISCA'14]
    - FPGA間を専用線で接続するクラスタシステム

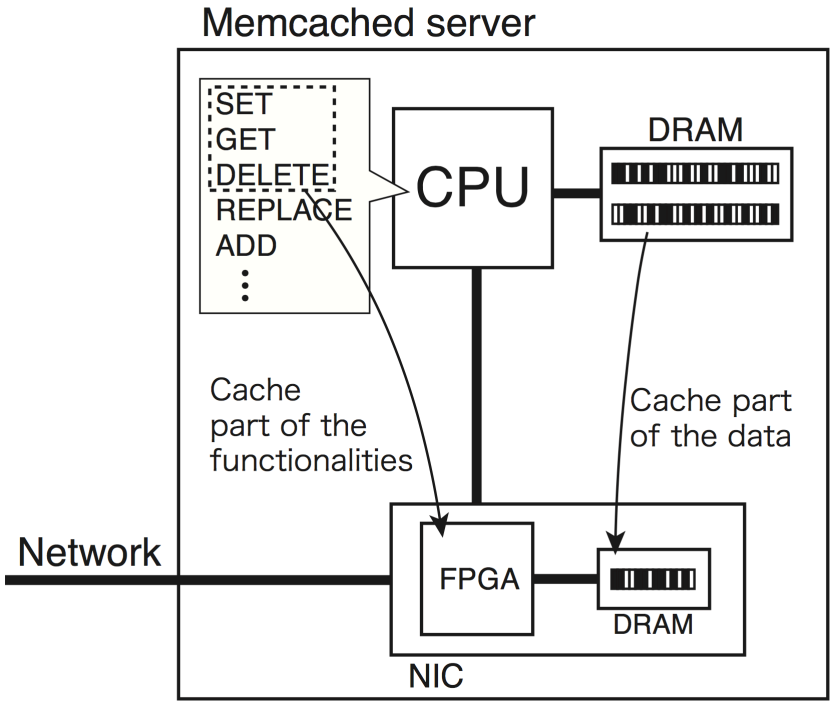


Fig2 from [Fukuda+, FPL'14]

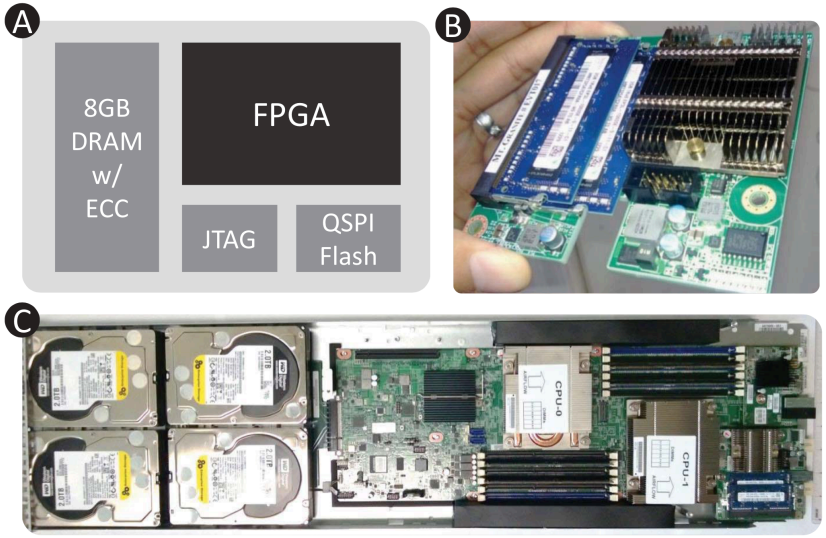
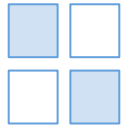


Figure 1: (a) A block diagram of the FPGA board. (b) A picture of the manufactured board. (c) A diagram of the 1 U, half-width server that hosts the FPGA board. The air flows from the left to the right, leaving the FPGA in the exhaust of both CPUs.

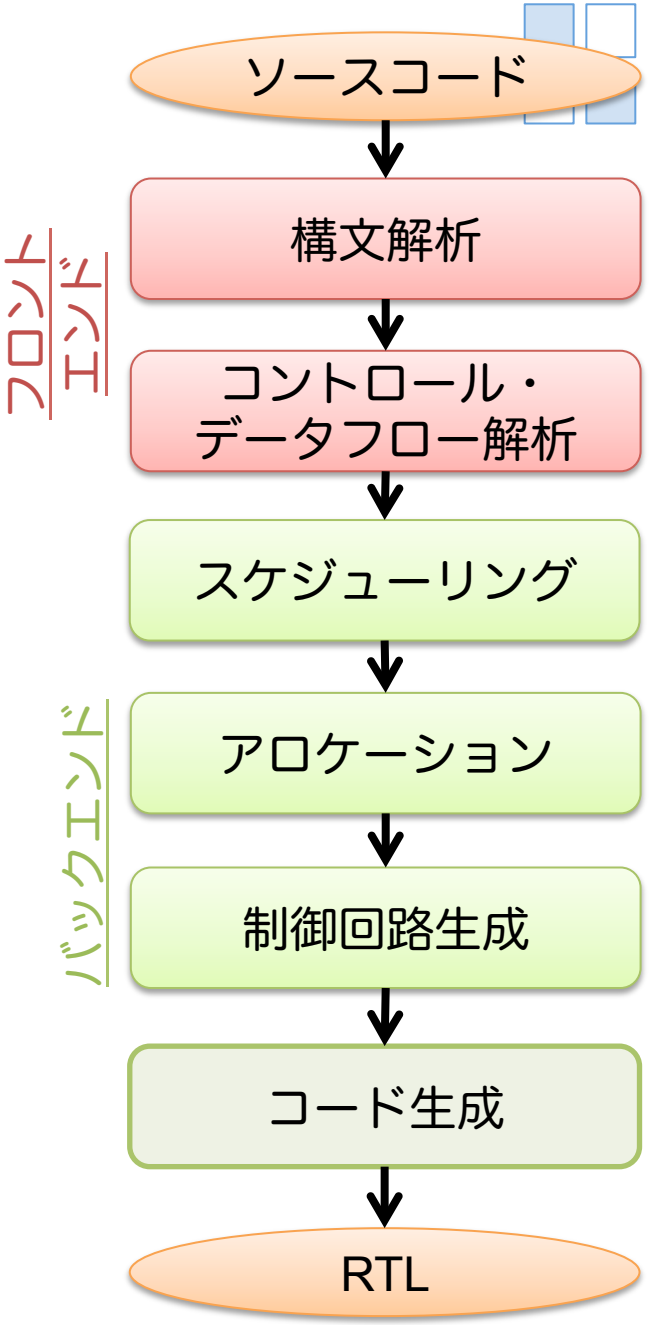
Fig1 from [Putman+, ISCA'14]



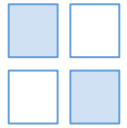
高位合成のあれこれ

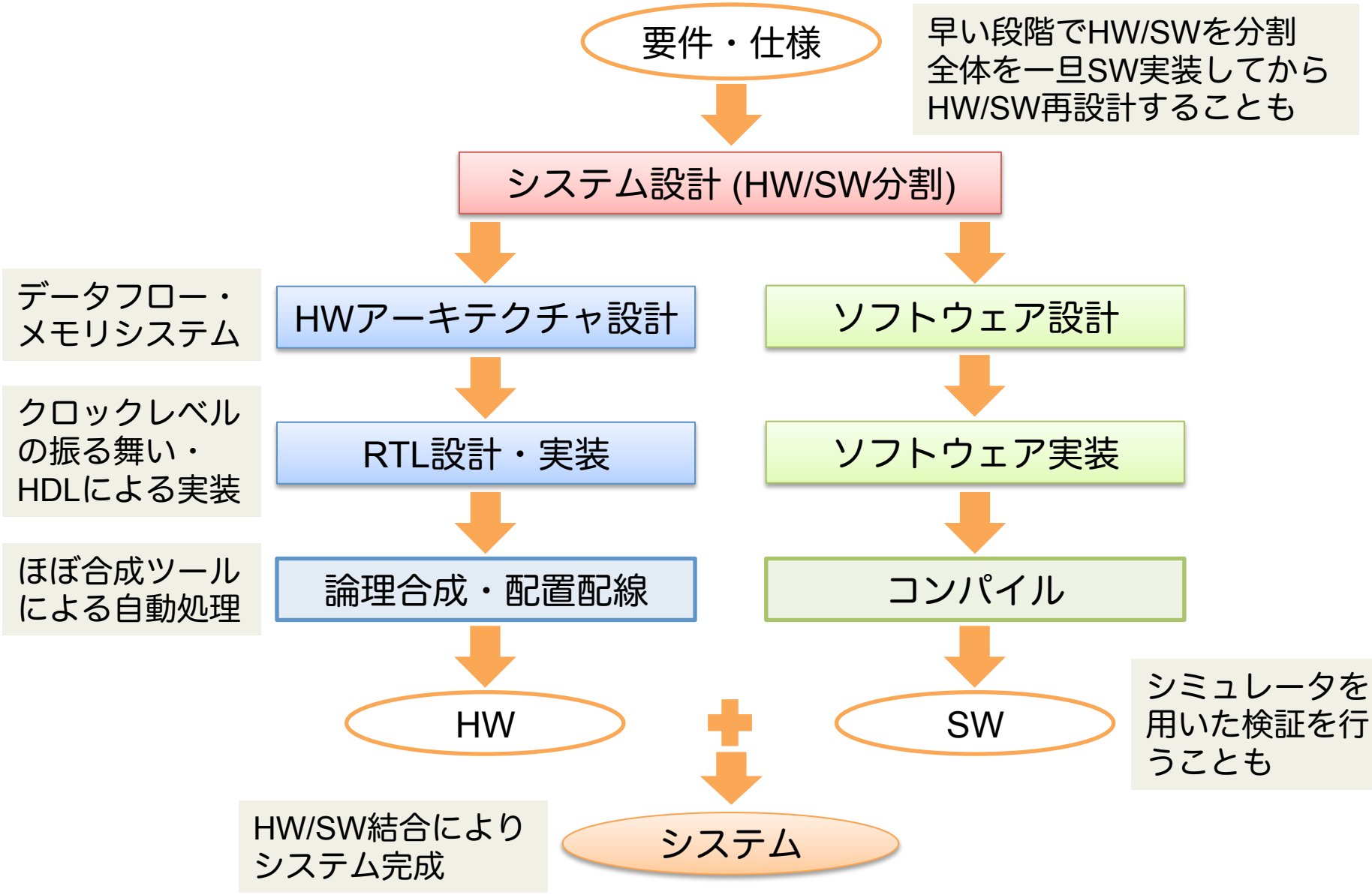
高位合成とは？

- 動作モデルからRTLモデルを生成するツール・コンパイラ
  - 入力: ソフトウェアのソースコード
    - C, C++, OpenCL, Java, Python, ...
  - 出力: RTL（HDLソースコード）
    - Verilog HDL, VHDL
- 構成：SWコンパイラと同等のフロントエンドと高位合成ならではのバックエンド
  - フロントエンド: 抽象構文木(AST)生成・コントロールデータフロー(CDFG)解析
  - バックエンド: 「いつ」「どの演算」をするか決めたり、演算を演算器に、変数をレジスタに割り当てり

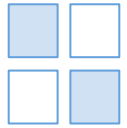


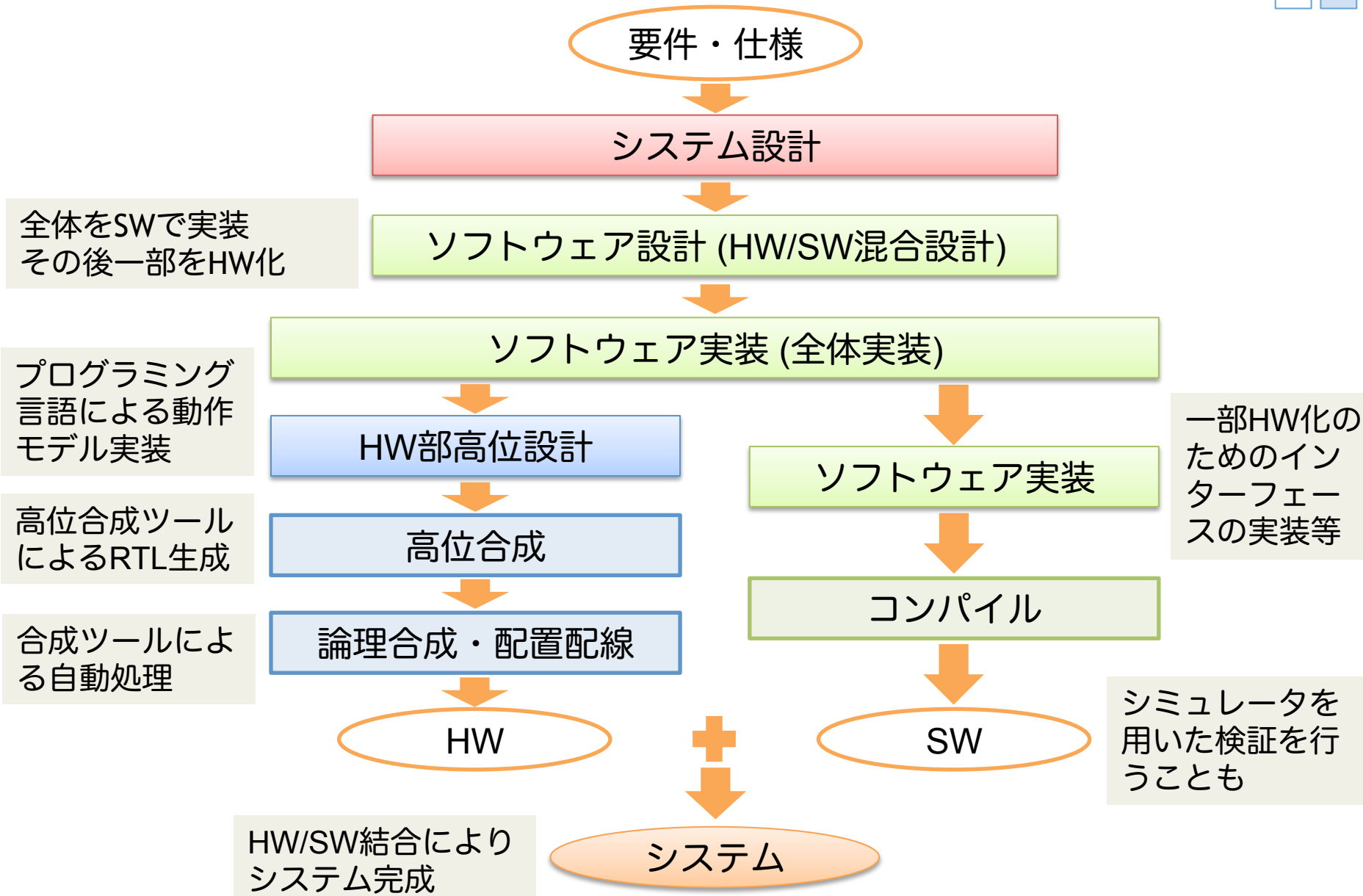
FPGAシステムの設計フロー（高位合成なし）



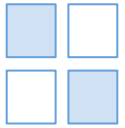


FPGAシステムの設計フロー（高位合成あり）



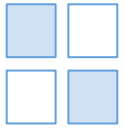


RTL設計と高位設計の違い



- RTL (Register Transfer Level) 設計
  - クロックサイクルレベルでレジスタ間のデータ移動や演算のタイミングを定義する
  - Timed設計：「なに」を「いつ」「どのように」行うかを定義
  - 通常のソフトウェアでいう「アセンブリ」の様なもの
- 高位設計（動作設計・High Level Synthesis）
  - （一般に）汎用プログラミング言語を用いて振る舞いを定義
  - Untimed設計：「なに」を行うかを実装
    - 「いつ」「どのように」かは定義しなくても良い
    - 指示子(Directive)で「いつ」「どのように」の情報を付加することにより、より良いハードウェアが生成される
  - 通常のソフトウェアでいう高級プログラミング言語に設計

例：2配列の積和演算 (c += a * b)



RTL設計 (Verilog HDL): 105行，2098文字，15分

積和演算器（トップレベル）

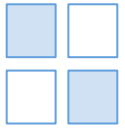
```
module madd #
  (
   parameter DATA_WIDTH = 32,
   parameter DEPTH = 6
   )
  (
   input CLK,
   input RST,

   input start,
   input in_valid,
   output reg out_valid,

   input  [DATA_WIDTH-1:0] a,
   input  [DATA_WIDTH-1:0] b,
   input  [31:0] length,
   output reg [DATA_WIDTH-1:0] sum
   );

  wire [DATA_WIDTH*2-1:0] mult_rslt;

  pipelined_multiplier #
    (
     .DATA_WIDTH(DATA_WIDTH),
     .DEPTH(DEPTH)
     )
  inst_mult
    (
     .CLK(CLK),
     .A(a),
     .B(b),
     .RSLT(mult_rslt)
     );

  reg in_valid_reg;
  reg [DEPTH-1:0] pipe_regs;
  reg mult_rslt_valid;
  integer i;

  always @(posedge CLK) begin
    if(RST) begin
      in_valid_reg <= 0;
      pipe_regs <= 0;
      mult_rslt_valid <= 0;
    end else begin
      in_valid_reg <= in_valid;
      pipe_regs[DEPTH-1] <= in_valid_reg;
      for(i=0; i<DEPTH-1; i=i+1) pipe_regs[i] <= pipe_regs[i+1];
      mult_rslt_valid <= pipe_regs[0];
    end
  end

  reg [31:0] length_reg;
  reg [31:0] count;

  always @(posedge CLK) begin
    if(RST) begin
      sum <= 0;
      count <= 0;
      out_valid <= 0;
      length_reg <= 0;
    end else begin
      if(start) begin
        sum <= 0;
        count <= 0;
        out_valid <= 0;
        length_reg <= length;
      end else if(!out_valid && mult_rslt_valid) begin
        sum <= sum + mult_rslt[DATA_WIDTH-1:0];
        count <= count + 1;
        if(count == length_reg - 1) begin
          out_valid <= 1;
        end
      end
    end
  end

endmodule
```

乗算器

```
module pipelined_multiplier #
  (
   parameter DATA_WIDTH = 32,
   parameter DEPTH = 6
   )
  (
   input CLK,
   input [DATA_WIDTH-1:0] A,
   input [DATA_WIDTH-1:0] B,
   output reg [DATA_WIDTH*2-1:0] RSLT
   );
  reg [DATA_WIDTH-1:0] a_in;
  reg [DATA_WIDTH-1:0] b_in;
  wire [DATA_WIDTH*2-1:0] mult_res;
  reg [DATA_WIDTH*2-1:0] pipe_regs [0:DEPTH-1];
  integer i;
  assign mult_res = a_in * b_in;
  always @(posedge CLK) begin
    a_in <= A;
    b_in <= B;
    pipe_regs[DEPTH-1] <= mult_res;
    for(i=0; i<DEPTH-1; i=i+1) pipe_regs[i] <= pipe_regs[i+1];
    RSLT <= pipe_regs[0];
  end
endmodule
```

「いつ」「なに」を
「どのように」するかを
設計者が決める

例：2配列の積和演算 (c += a * b)



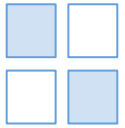
高位設計 (C言語): 11行，163文字，1分
→1/10の記述量と1/15の開発時間
（ただしディレクティブ等はなし）

「なに」をするかだけを
設計者が決める

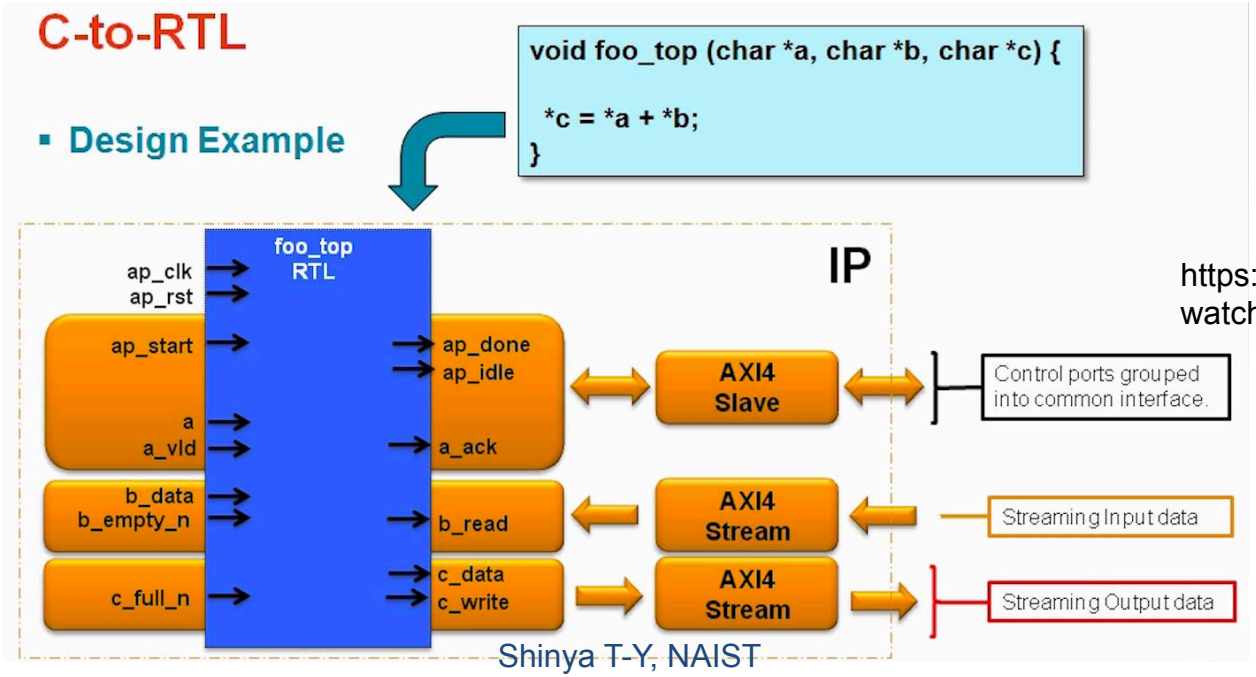
```
typedef unsigned int u32;

u32 madd(u32 a[], u32 b[], u32 length)
{
  u32 i;
  u32 sum = 0;
  for(i=0; i<length; i++){
    sum += a[i] * b[i];
  }
  return sum;
}
```

（性能が出れば）高位設計最高！

FPGA向け商用高位合成ツールが多数登場



- Xilinx Vivado HLS (+ SDSoC)
  - C/C++で振る舞いを定義・ディレクティブで性能チューニング
  - SDSoCならSWコードから部分的にHW化・I/Fも自動生成
  - その他C言語ベースImpulse C・CWB・eXCiteなど
- OpenCL系: Altera OpenCLやXilinx SDAccel
  - ホストPCありき・ホストPC上SWのお作法も定義



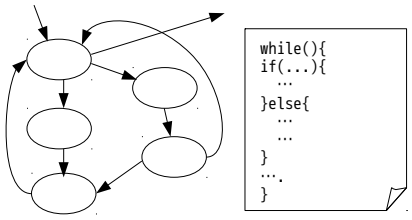
https://www.youtube.com/watch?v=URUVkq6zQhQ

オープンソースな高位合成ツールの登場

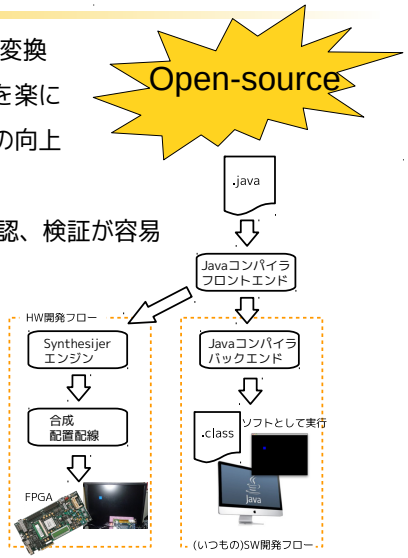
- LegUp: トロント大で開発されているCベース処理系
  - C記述をMIPS CPU用SW部とHW部に自動分割・論文多数
- Synthesijer: Javaによる高位合成処理系
  - Java言語仕様に改変なし，サブセットをそのまま合成可能

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)



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

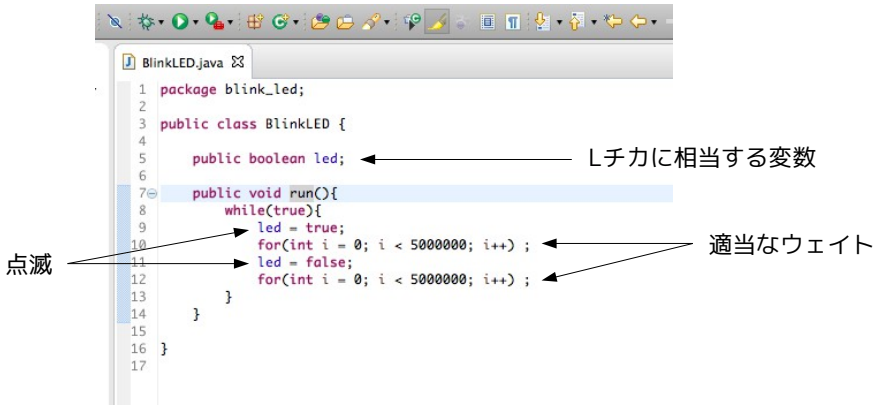


同じJavaプログラムをソフトウェアとしても
FPGA上のハードウェアとしても実行可能

2

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

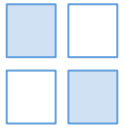


自動コンパイルが裏で動くので，Javaコードとしての正しさは
即座にチェックされる

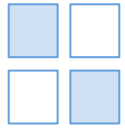
9

http://www.sigemb.jp/ESS/2014/files/IPSJ-ESS2014003-1.pdf

高位合成のメリット・デメリット

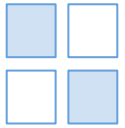


- メリット☺
  - いつものプログラミング言語でハードウェアが開発できる
  - 少ない記述量：RTL設計の1/10以下
  - デバッグのしやすさ：
    通常のソフトウェアとして実行して機能レベルの正しさを検証
- デメリット☹
  - 達成可能な性能・電力効率・面積効率はRTL設計の方が高い
    - RTL設計は開発者の知見（アプリケーションの要件等）が直接反映
  - 癖のあるソースコードを書く必要がある
    - 専用の型の導入や機能制限
    - 性能を追求するにはディレクティブを多数挿入する必要がある
      - 結局書き直すならRTLで書き直しても良いのでは？という人も多い
  - クロックサイクルレベルの振る舞いを書くのが苦手
    - I/Oの制御を細粒度に行う回路等は不向き（なことが多い）



PyCoRAM
Pythonによるハードウェア
IPコア設計フレームワーク

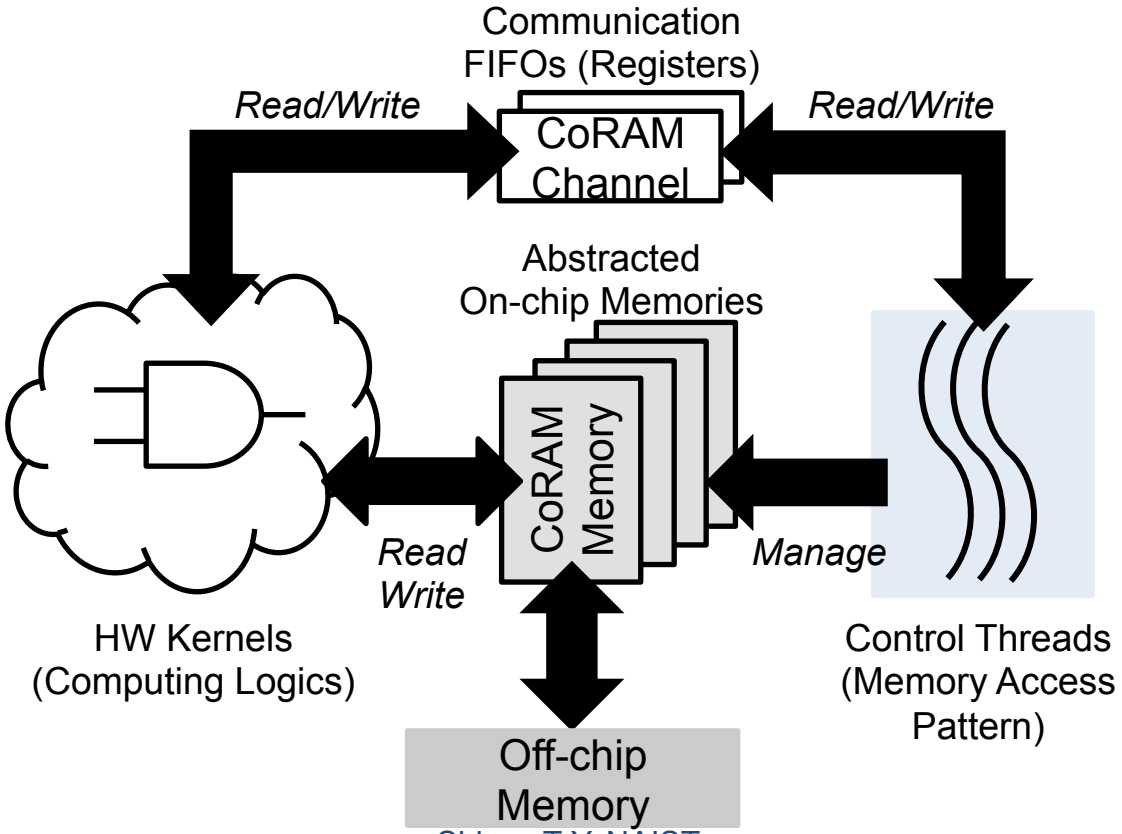
背景



- CPU with IP-cores
  - ハードマクロCPU (ARM) 搭載FPGAが主流に: Xilinx Zynq, etc
  - 専用HWはIPコアとして実装し
    インターコネクト（AXI4やAvalon）を介してCPUと接続
- どのようにしてアクセラレータIPコアを設計するか？
  - HDLですべて設計するのは大変
    - 例）演算とメモリアクセスのスケジューリング
      - ダブルバッファリングとか面倒
      - HDLでステートマシンを書くのは大変だし間違えやすい
  - 性能を出すには高稼働率パイプラインとデータ供給機構が必要
    - ストールのない綺麗なパイプラインを定義したい: HDLが得意
    - （OSSな）高位合成系だとチューニングが難しい
- →データ転送を抽象化すれば幸せそう？

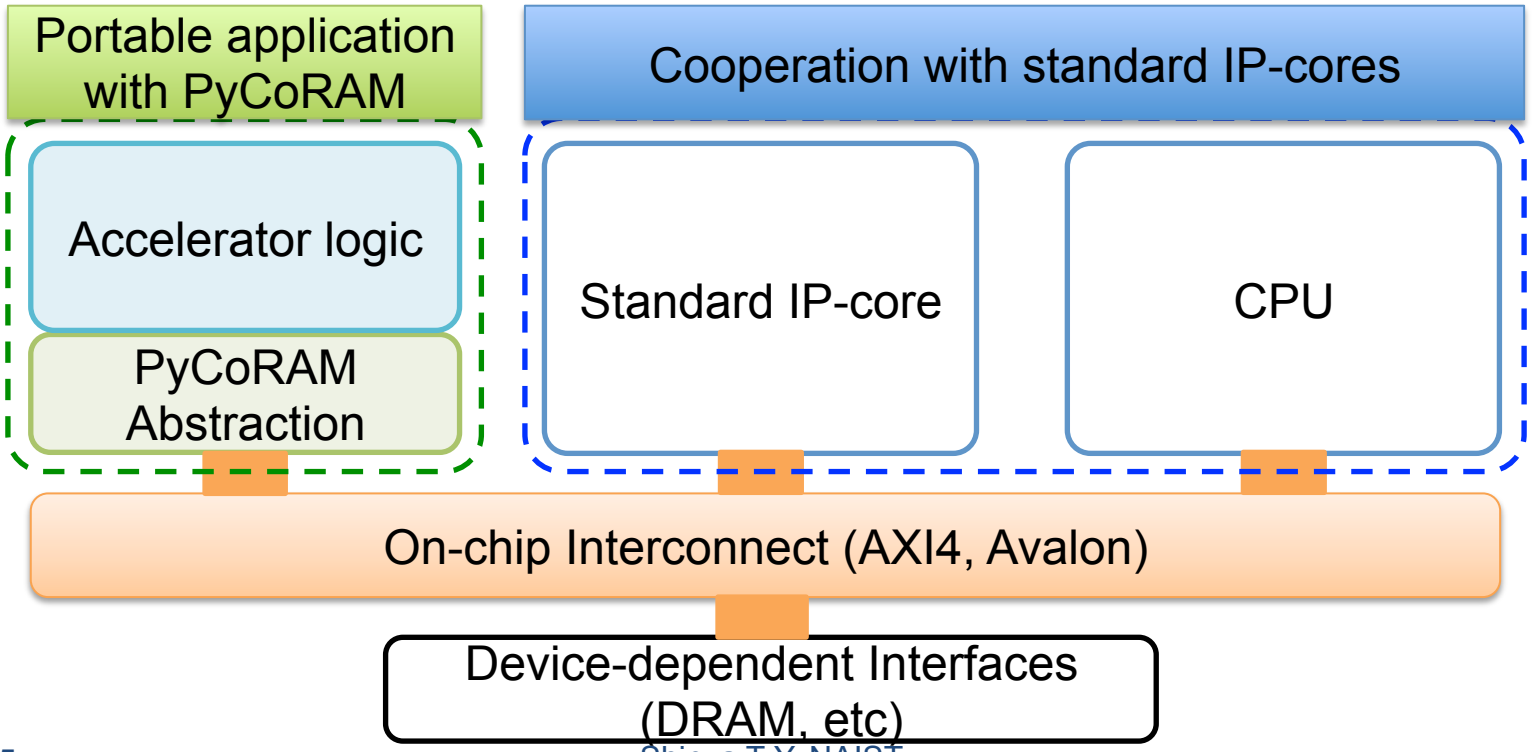
CoRAM [Chung+,FPGA'11]

- FPGAアクセラレータのためのメモリ抽象化
  - 高位モデルによるメモリ管理でアクセラレータをポータブルに
    - 計算カーネルとメモリアクセスの分離
    - ソフトウェアのモデルによるメモリアクセスパターンの記述



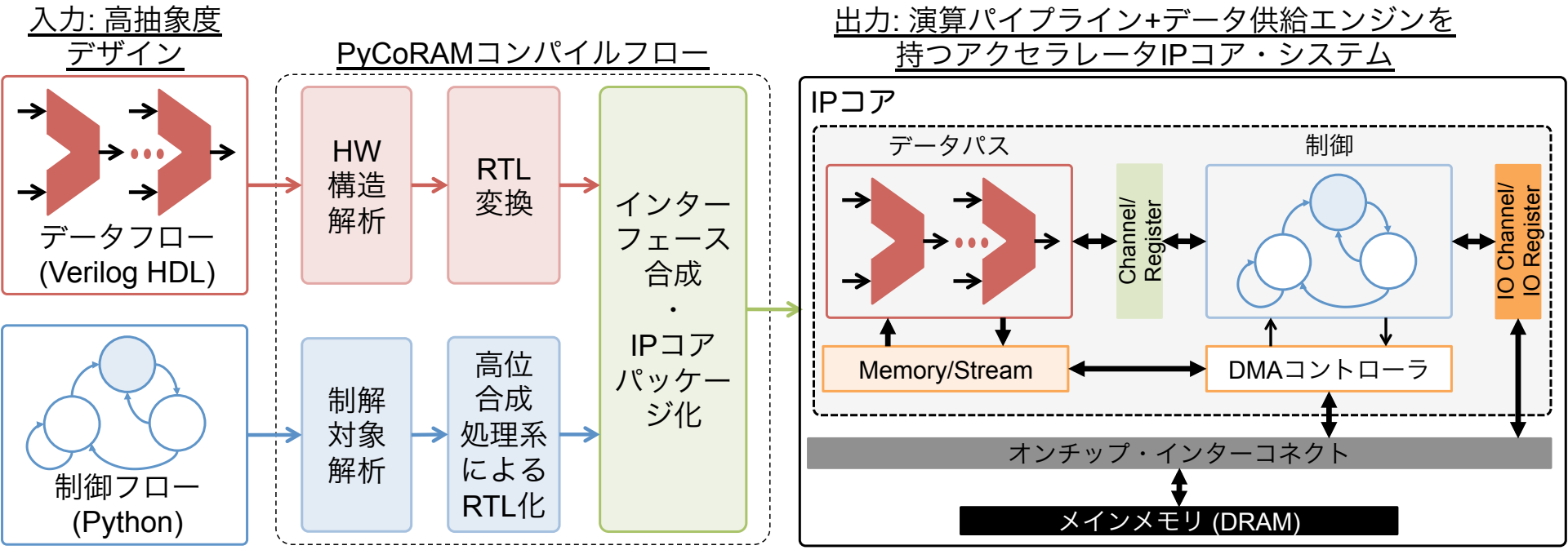
PyCoRAM [Takamaeda+,CARL'13]

- 抽象化されたインターコネクトとメモリシステムの上でIPコアを開発するフレームワーク
  - 標準的なインターコネクトに繋げる：AMBA AXI4, Altera Avalon
  - ポータブル：抽象化がプラットフォームの違いを吸収
  - CPUや他のIPコアと共存が容易：ビルディングブロック開発

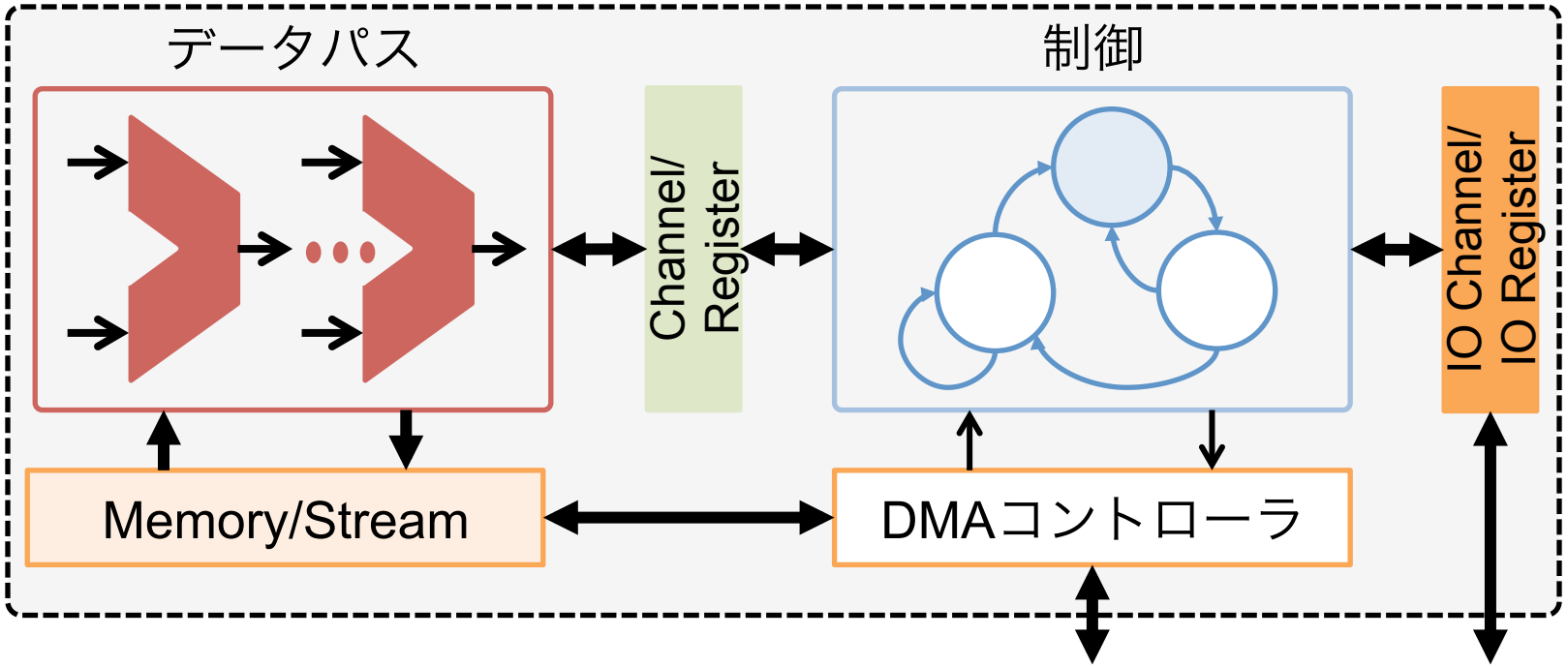


PyCoRAM：フレームワーク構成

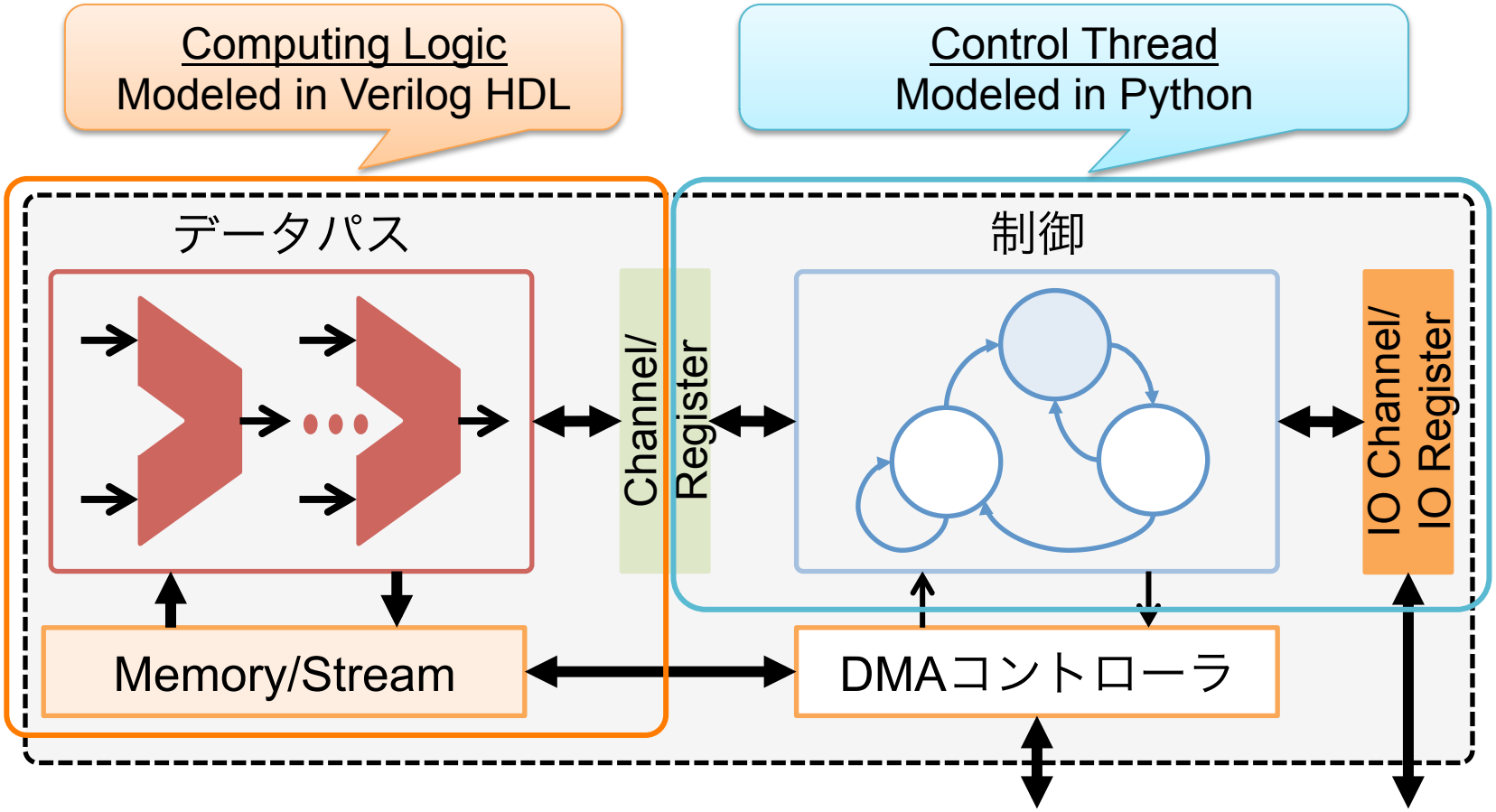
- 入力：2種類のソースコード
  - データフロー（演算パイプライン）：Verilog HDL
  - データ転送制御：Python
- 出力：演算パイプと制御機構を持つIPコア
  - DMAを主体としたデータ転送重点型ハードウェアが合成される



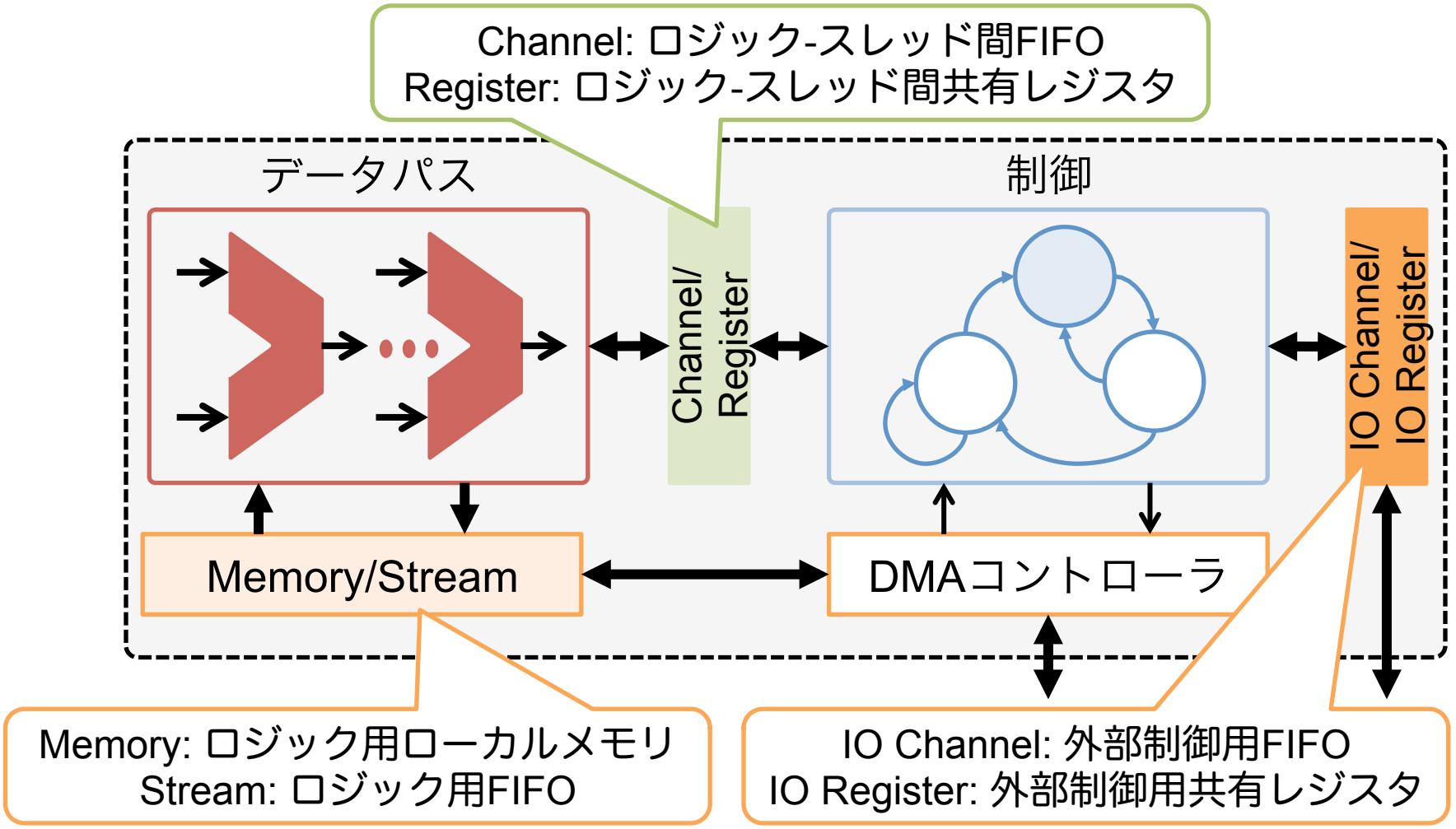
PyCoRAM：アーキテクチャ



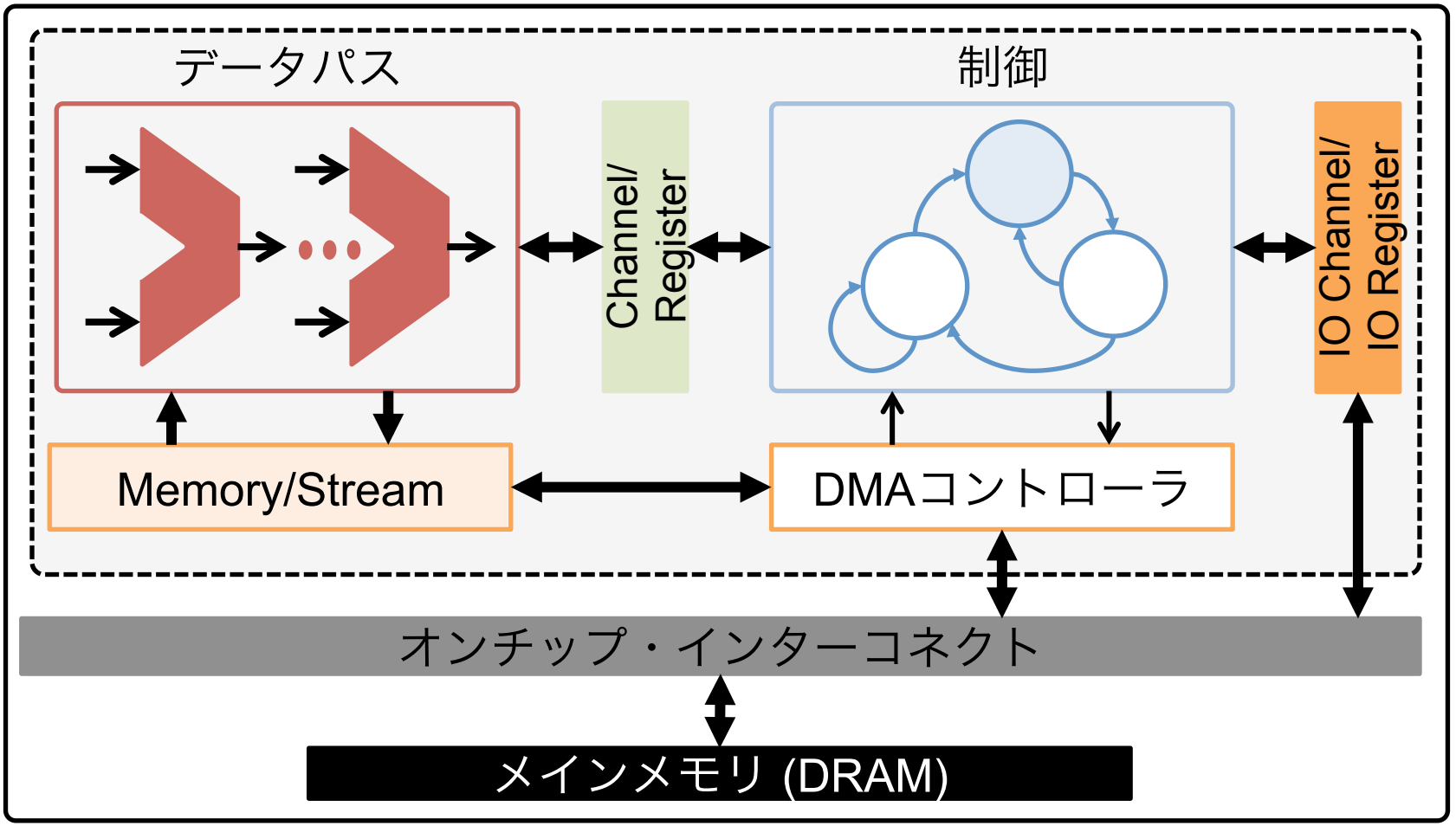
PyCoRAM：アーキテクチャ



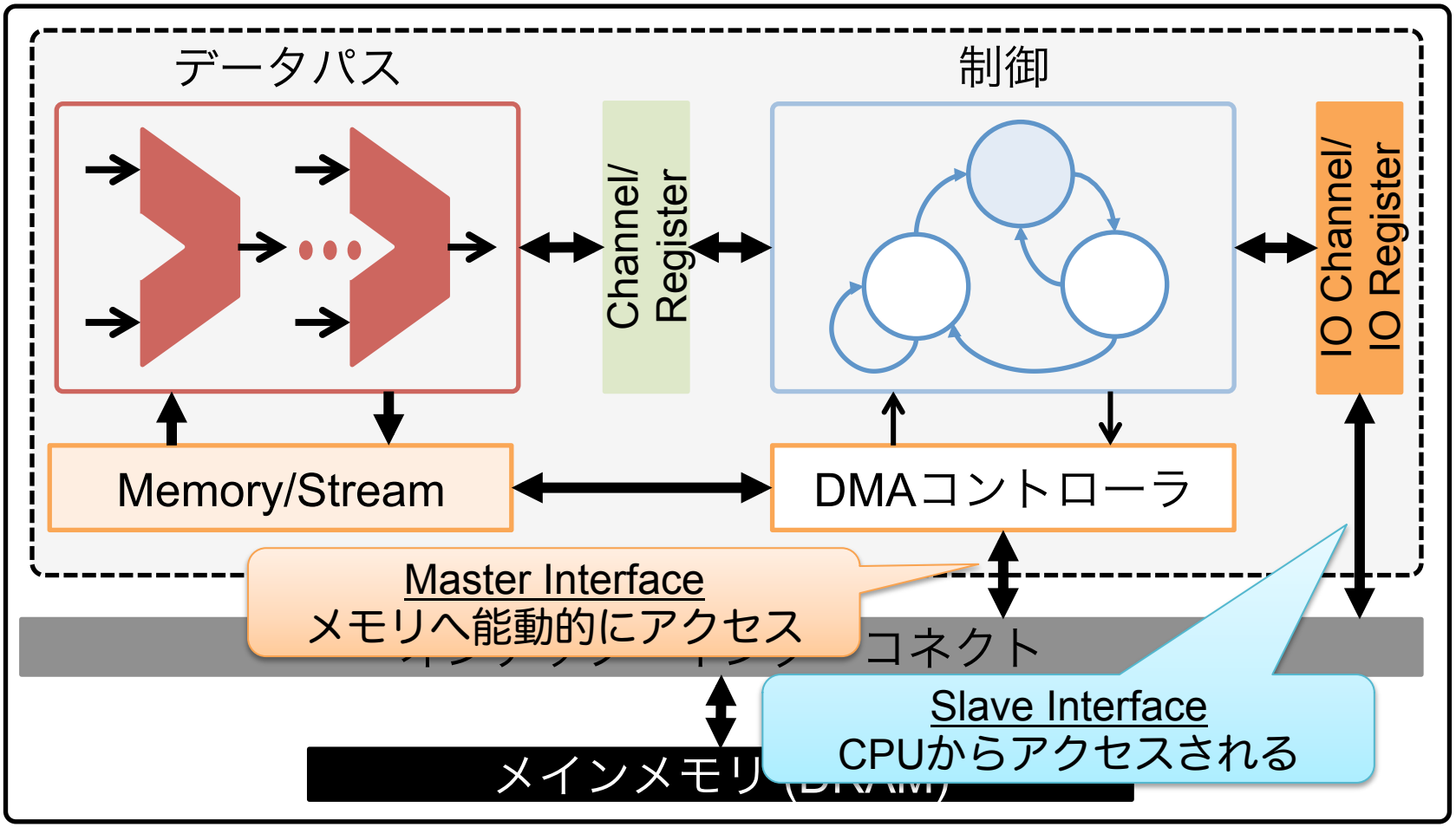
PyCoRAM：アーキテクチャ



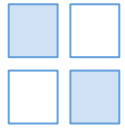
PyCoRAM：アーキテクチャ



PyCoRAM：アーキテクチャ



PyCoRAMにおけるIPコアの作り方・でき方



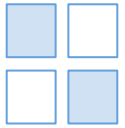
- 2種類のファイルを用意する
  - Verilog HDL: 計算ロジック（データパス）
  - Python: コントロールスレッド（制御）

```
CoramMemory1P
#(
 .CORAM_THREAD_NAME("thread_name"),
 .CORAM_ID(0),
 .CORAM_ADDR_LEN(ADDR_LEN),
 .CORAM_DATA_WIDTH(DATA_WIDTH)
 )
inst_memory
(.CLK(CLK),
 .ADDR(mem_addr),
 .D(mem_d),
 .WE(mem_we),
 .Q(mem_q)
 );
```

```
def calc_sum(times):
    ram = CoramMemory(idx=0, datawidth=32, size=1024)
    channel = CoramChannel(idx=0, datawidth=32)
    addr = 0
    sum = 0
    for i in range(times):
        ram.write(0, addr, 128)
        channel.write(addr)
        sum += channel.read()
        addr += 128 * (32/8)
    print('sum=', sum)
calc_sum(8)
```

- PyCoRAMが自動的にIPコアのパッケージを作成
  - Python-Verilog高位合成とRTL変換を自動で行う

計算ロジックにおけるPyCoRAMオブジェクト



- ブロックRAMやFIFOと同様のインターフェス
  - ロジック側からおなじみのインターフェースでアクセスできる
  - いくつかのパラメータで特性を指定
    - スレッド名，ID，データ幅，アドレス幅，スキャッターギャザー等
  - 外部を接続するインターフェースは自動的に追加される

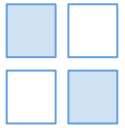
```
CoramMemory1P
#(
  .CORAM_THREAD_NAME("thread_name"),
  .CORAM_ID(0),
  .CORAM_ADDR_LEN(ADDR_LEN),
  .CORAM_DATA_WIDTH(DATA_WIDTH)
  )
inst_memory
(.CLK(CLK),
 .ADDR(mem_addr),
 .D(mem_d),
 .WE(mem_we),
 .Q(mem_q)
 );
```

(a) Memory

```
CoramChannel
#(
  .CORAM_THREAD_NAME("thread_name"),
  .CORAM_ID(0),
  .CORAM_ADDR_LEN(CHANNEL_ADDR_LEN),
  .CORAM_DATA_WIDTH(CHANNEL_DATA_WIDTH)
  )
inst_channel
(.CLK(CLK),
 .RST(RST),
 .D(comm_d),
 .ENQ(comm_enq),
 .FULL(comm_full),
 .Q(comm_q),
 .DEQ(comm_deq),
 .EMPTY(comm_empty)
 );
```

(b) Channel

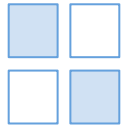
Pythonによるコントロールスレッド



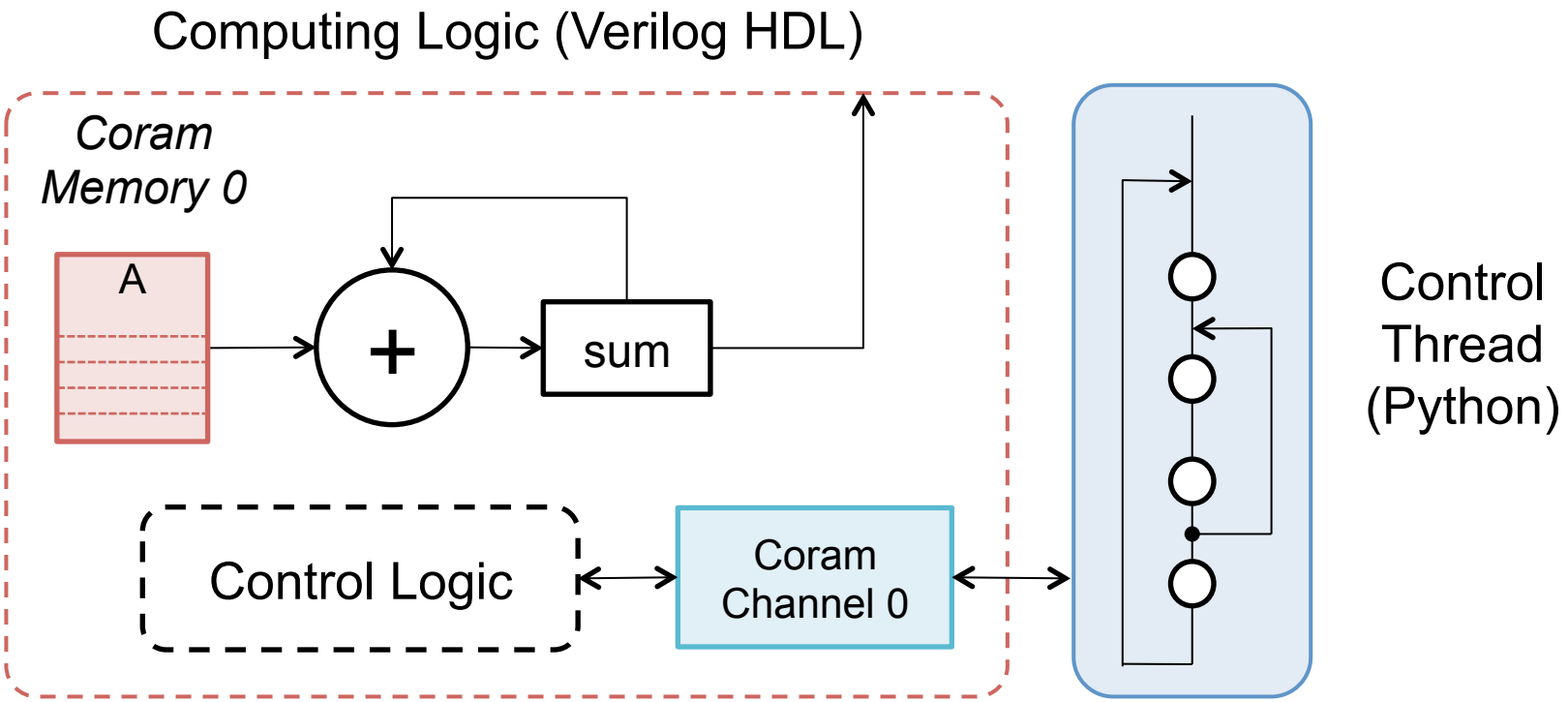
- PyCoRAMオブジェクトに対する処理を記述する
  - CoramMemory: read(), write()
    - MemoryとDRAMとの間のDMA転送によるデータ移動
  - CoramChannel: read(), write()
    - 計算ロジックとコントロールスレッドの間のデータのやりとり

```
0  def calc_sum(times):
1      ram = CoramMemory(idx=0, datawidth=32, size=1024)
2      channel = CoramChannel(idx=0, datawidth=32)
3      addr = 0
4      sum = 0
5      for i in range(times):
6          ram.write(0, addr, 128)  # Transfer (off-chip DRAM to BRAM)
7          channel.write(addr)      # Notification to User-logic
8          sum += channel.read()    # Wait for Notification from User-logic
9          addr += 128 * (32/8)
10     print('sum=', sum)           # $display Verilog system task
11 calc_sum(8)
```

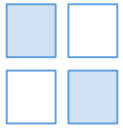
例: 配列の和を求めるアクセラレータ



- 1メモリ+1スレッドの簡単なハードウェア
  - CoramMemoryにDRAMからデータを読み込む
  - スレッド側でCoramMemory-DRAM間のデータ転送系列を表現



計算ロジック (1): I/Oポート



```
`include "pycoram.v"

`define THREAD_NAME "ctrl_thread"

module userlogic #
  (
   parameter W_A = 10,
   parameter W_COMM_A = 4,
   parameter W_D = 32,
   parameter SIZE = 128
   )
  (
   input CLK,
   input RST,
   output reg [31:0] sum
   );

  reg [W_A-1:0] mem_addr;
  reg [W_D-1:0] mem_d;
  reg           mem_we;
  wire [W_D-1:0] mem_q;

  reg [W_D-1:0]  comm_d;
  reg            comm_enq;
  wire           comm_full;
  wire [W_D-1:0] comm_q;
  reg            comm_deq;
  wire           comm_empty;

  reg [3:0] state;
```

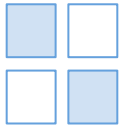
クロック(CLK)とリセット(RST)
以外に専用のI/Oは不要

CoramMemoryのための信号
(BRAMと同じインターフェース)

CoramChannelのための信号
(FIFOと同じインターフェース)

ステートマシン用変数

計算ロジック (2): パイプライン/FSM



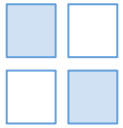
```
always @(posedge CLK) begin
  if(RST) begin
    state <= 0;
    mem_we <= 0;
    comm_deq <= 0;
    comm_enq <= 0;
  end else begin
    // default value
    comm_enq <= 0;
    comm_deq <= 0;

    if(state == 0) begin
      sum <= 0;
      mem_d <= 0;
      mem_we <= 0;
      mem_addr <= 0;
      if(!comm_empty) begin
        comm_deq <= 1;
        state <= 1;
      end
    end else if(state == 1) begin
      state <= 2;
      mem_addr <= 0;
    end else if(state == 2) begin
      state <= 3;
      mem_addr <= mem_addr + 1;
```

CoramChannelから読み出し
（コントロールスレッドから受信）

```
    end else if(state == 3) begin
      mem_addr <= mem_addr + 1;
      sum <= sum + mem_q;
      if(mem_addr == SIZE-2) begin
        state <= 4;
      end
    end else if(state == 4) begin
      state <= 5;
      sum <= sum + mem_q;
    end else if(state == 5) begin
      state <= 6;
      sum <= sum + mem_q;
    end else if(state == 6) begin
      if(!comm_full) begin
        comm_d <= sum;
        comm_enq <= 1;
        state <= 7;
      end
    end else if(state == 7) begin
      comm_enq <= 0;
      if(!comm_empty) begin
        comm_deq <= 1;
        state <= 1;
      end
    end
  end
end
```

CoramChannelに書き込み
（コントロールスレッドに通知）

計算ロジック (3): 子インスタンス



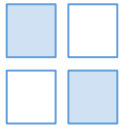
```
CoramMemory1P
#(
   .CORAM_THREAD_NAME(`THREAD_NAME),
   .CORAM_ID(0),
   .CORAM_SUB_ID(0),
   .CORAM_ADDR_LEN(W_A),
   .CORAM_DATA_WIDTH(W_D)
  )
inst_data_memory
(.CLK(CLK),
 .ADDR(mem_addr),
 .D(mem_d),
 .WE(mem_we),
 .Q(mem_q)
 );
```

CoramMemory
(BRAMと同じインターフェース)

```
CoramChannel
#(
   .CORAM_THREAD_NAME(`THREAD_NAME),
   .CORAM_ID(0),
   .CORAM_ADDR_LEN(W_COMM_A),
   .CORAM_DATA_WIDTH(W_D)
  )
inst_comm_channel
(.CLK(CLK),
 .RST(RST),
 .D(comm_d),
 .ENQ(comm_enq),
 .FULL(comm_full),
 .Q(comm_q),
 .DEQ(comm_deq),
 .EMPTY(comm_empty)
 );

endmodule
```

CoramChannel
(FIFOと同じインターフェース)

コントロールスレッド (Python)

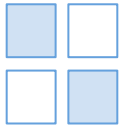


ram (CoramMemory)とchannel (CoramChannel)の宣言

```
def ctrl_thread():
    ram = CoramMemory(idx=0, datawidth=32, size=1024, length=1, scattergather=False)
    channel = CoramChannel(idx=0, datawidth=32, size=16)
    addr = 0
    sum = 0
    for i in range(8):
        ram.write(0, addr, 128) # from DRAM to BlockRAM
        channel.write(addr)
        sum = channel.read()
        addr += 512
    print('sum=', sum)

ctrl_thread()
```

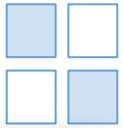
CoramMemoryにDMA転送したり
CoramChannelから読んだり書いたり

コンパイル



```
single_memory — bash
shta@LORRAINE:single_memory$ make build
python3.3  ../../..//pycoram.py -I ../../..//include/ -t userlogic --ipver=v1_00_a --extaddrwidth=32 --extdatawidth=
512 --signalwidth=64 --memimg=../mem-incr.hex  --simaddrwidth=24 --hperiod_ulogic=5 --hperiod_cthread=5 --hperiod_ax
i=5 --singleclock  userlogic.v ctrl_thread.py
Generating LALR tables
WARNING: 146 shift/reduce conflicts
Generating LALR tables
WARNING: 146 shift/reduce conflicts
CoRAM Objects in User-defined RTL
  CoRAM CoramMemory
    CoramMemory(ID:0[0] Thread:ctrl_thread AddrWidth:(9, 0) DataWidth:(31, 0))
  CoRAM CoramChannel
    CoramChannel(ID:0 Thread:ctrl_thread AddrWidth:(3, 0) DataWidth:(31, 0))
CoRAM Objects in Control-Thread 'ctrl_thread', # FSM = 15
  CoRAM CoramMemory:
    CoramMemory(ID:0 Length:1 AddrWidth:10 DataWidth:32) alias: __s0_ram
  CoRAM CoramChannel:
    CoramChannel(ID:0 AddrWidth:4 DataWidth:32) alias: __s0_channel
shta@LORRAINE:single_memory$ ls pycoram_userlogic_v1_00_a/
data doc  hdl  test
shta@LORRAINE:single_memory$
```

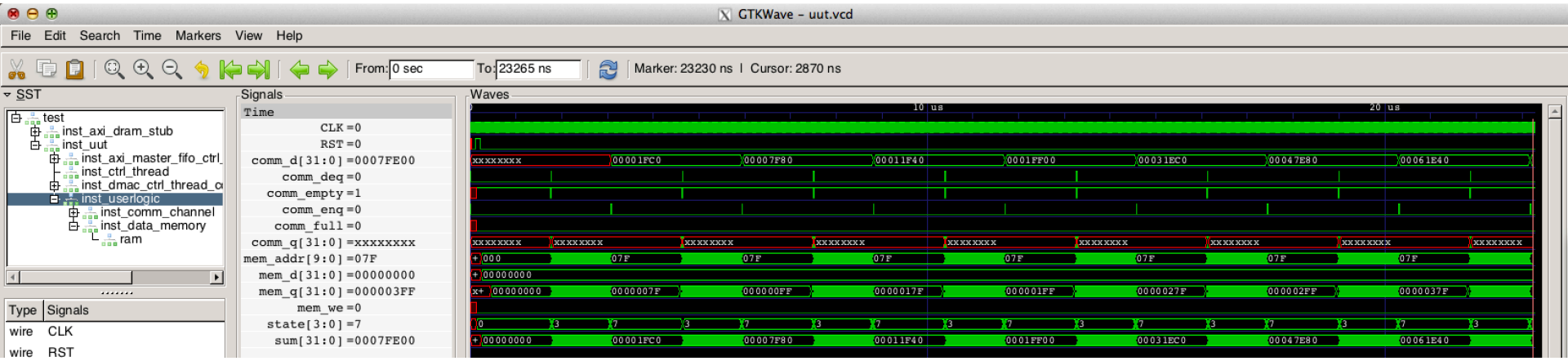
シミュレーション結果



shta — screen

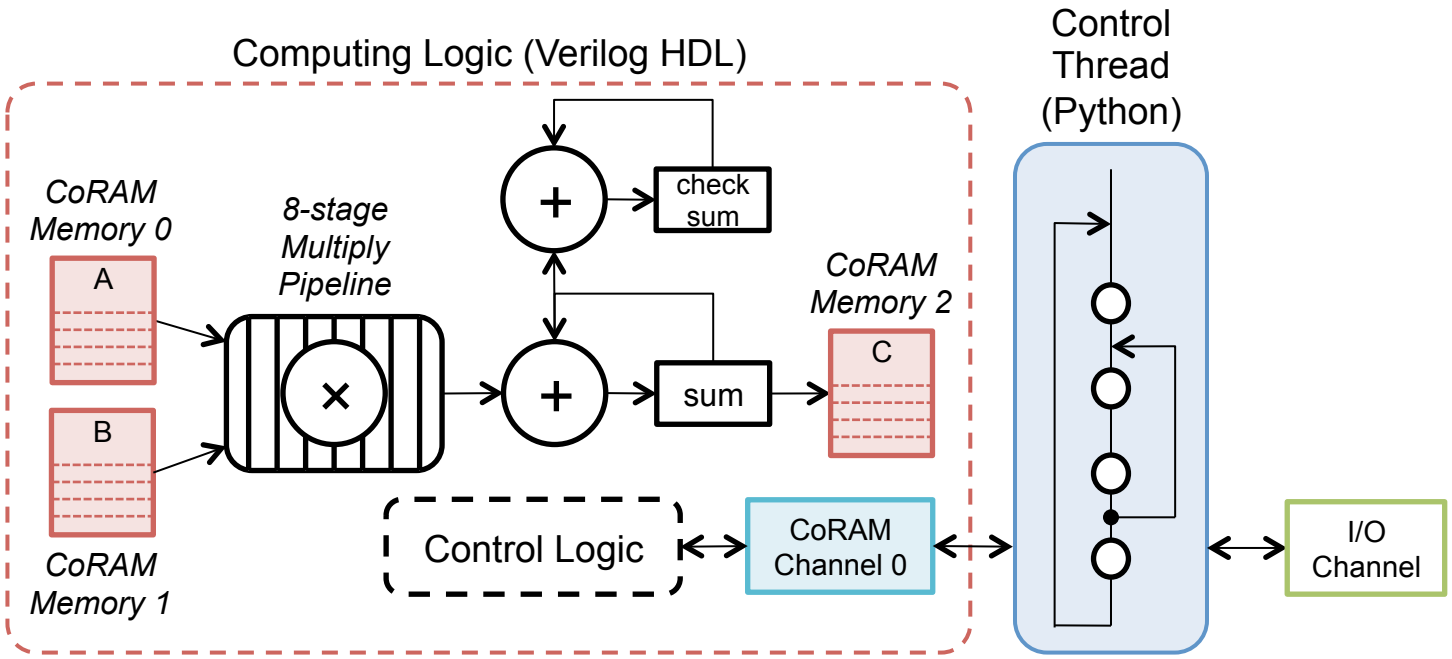
```
shta@LORRAINE:single_memory$ make sim
make compile -C pycoram_userlogic_v1_00_a/test
iverilog -I ../hdl/verilog/ -o a.out test_pycoram_userlogic.v
make run -C pycoram_userlogic_v1_00_a/test
./a.out
WARNING: test_pycoram_userlogic.v:499: $readmemh(mem.img): Not enough words in the file for the requested range [0:16777215].
read memory image file mem.img
VCD info: dumpfile uut.vcd opened for output.
[CoRAM] time:       285 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[                0]
[CoRAM] time:      1735 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[                0]
[CoRAM] time:      1745 thread:ctrl_thread channel:coramchannel_0 write data:                  0
[CoRAM] time:      3105 thread:ctrl_thread channel:coramchannel_0 read data:       8128
[CoRAM] time:      3155 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[              512]
[CoRAM] time:      4605 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[              512]
[CoRAM] time:      4615 thread:ctrl_thread channel:coramchannel_0 write data:                512
[CoRAM] time:      5975 thread:ctrl_thread channel:coramchannel_0 read data:      32640
[CoRAM] time:      6025 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[             1024]
[CoRAM] time:      7475 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[             1024]
[CoRAM] time:      7485 thread:ctrl_thread channel:coramchannel_0 write data:               1024
[CoRAM] time:      8845 thread:ctrl_thread channel:coramchannel_0 read data:      73536
[CoRAM] time:      8895 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[             1536]
[CoRAM] time:     10345 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[             1536]
[CoRAM] time:     10355 thread:ctrl_thread channel:coramchannel_0 write data:               1536
[CoRAM] time:     11715 thread:ctrl_thread channel:coramchannel_0 read data:     130816
[CoRAM] time:     11765 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[             2048]
[CoRAM] time:     13215 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[             2048]
[CoRAM] time:     13225 thread:ctrl_thread channel:coramchannel_0 write data:               2048
[CoRAM] time:     14585 thread:ctrl_thread channel:coramchannel_0 read data:     204480
[CoRAM] time:     14635 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[             2560]
[CoRAM] time:     16085 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[             2560]
[CoRAM] time:     16095 thread:ctrl_thread channel:coramchannel_0 write data:               2560
[CoRAM] time:     17455 thread:ctrl_thread channel:coramchannel_0 read data:     294528
[CoRAM] time:     17505 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[             3072]
[CoRAM] time:     18955 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[             3072]
[CoRAM] time:     18965 thread:ctrl_thread channel:coramchannel_0 write data:               3072
[CoRAM] time:     20325 thread:ctrl_thread channel:coramchannel_0 read data:     400960
[CoRAM] time:     20375 thread:ctrl_thread memory:corammemory_0 write issue size: 128 B[0]<-D[             3584]
[CoRAM] time:     21825 thread:ctrl_thread memory:corammemory_0 write done size: 128 B[0]<-D[             3584]
[CoRAM] time:     21835 thread:ctrl_thread channel:coramchannel_0 write data:               3584
[CoRAM] time:     23195 thread:ctrl_thread channel:coramchannel_0 read data:     523776
sum=              523776
[CoRAM] time:     23265 thread:ctrl_thread finished
[CoRAM] time:     23265 all threads finished
shta@LORRAINE:single_memory$
0 bash  1 bash  2 bash  3 bash                                                    LORRAINE  2014 03/10  6:33:55pm
```

シミュレーション結果（波形）



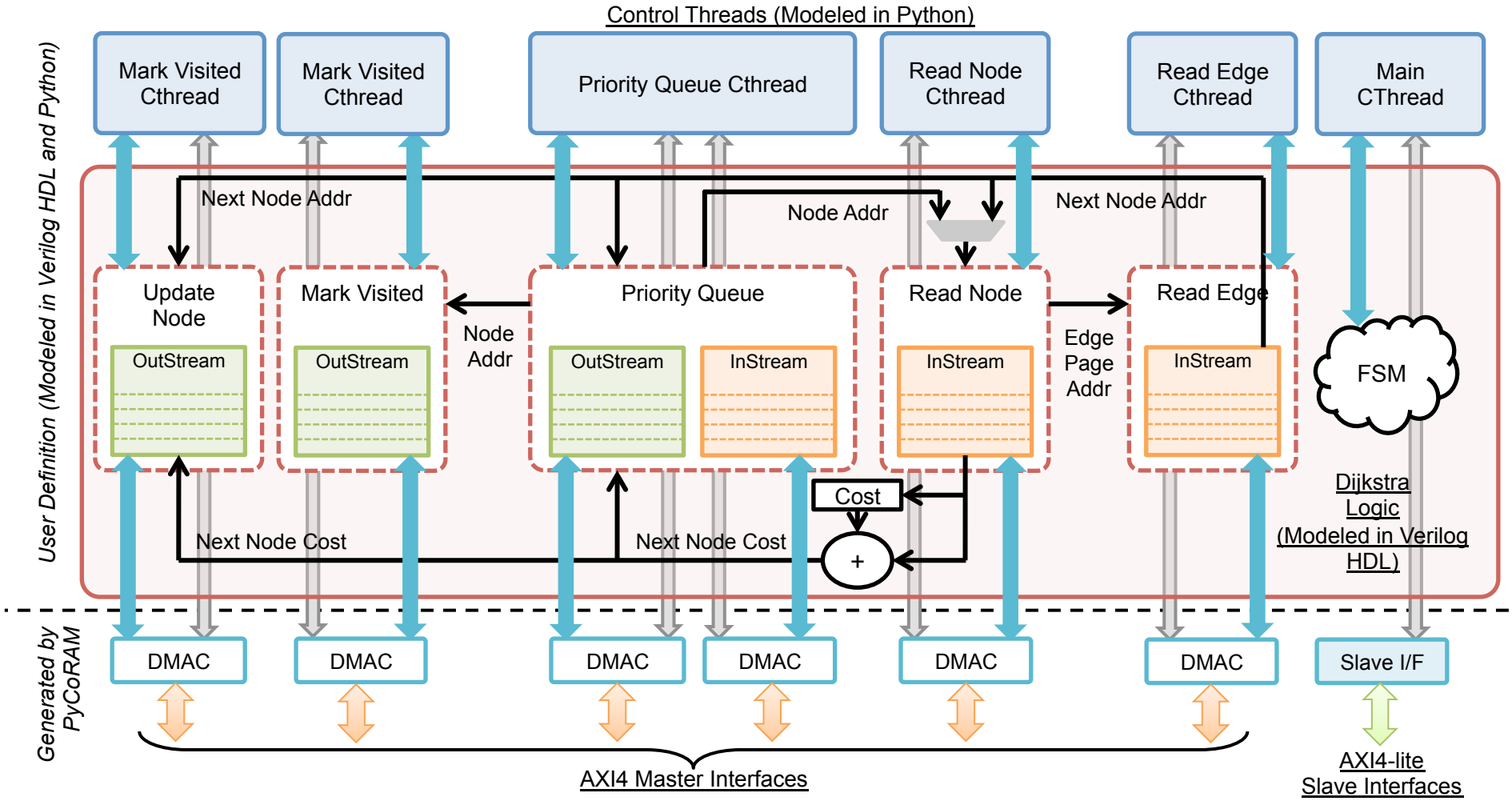
例）行列積IPコア

- 行列A・B・Cの各行をCoRAMメモリに格納
  - DRAMとの間のデータ転送をコントロールスレッドが担当
  - 毎サイクル乗算パイプラインにデータを投入
  - 行列Bの転送と演算をダブルバッファリング
  - SIMD幅をメモリバンド幅を使い切るように合わせる

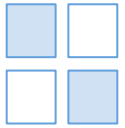


例）ダイクストラIPコア

- PyCoRAMを使って演算モジュールはVerilog HDLで実装
  メモリアクセス制御はPythonで実装



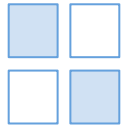
FPGAシステム開発で面倒なところ



- CPUや他のIPとの接続方法・インターフェース実装
  - IPコアのバスインターフェス：AXI4, Avalon
    - PyCoRAMならインターフェースもIPコア設定ファイルも自動生成
- データの置き場・供給方法
  - オンチップメモリ（ブロックRAM）・DRAM
    - PyCoRAMならDRAM-オンチップメモリ間のデータ移動制御はPythonで書けて楽ちん
- CPUとのデータ共有方法
  - OSなし：論理メモリ空間=物理メモリ空間なので簡単
    - volatileな変数を使えばOK
  - OSあり：仮想メモリが存在するため面倒
    - CPU: 仮想メモリあり，論理アドレスでアクセス
    - HW: 仮想メモリなし，物理アドレスでアクセス
      - 数MB以上の物理連続領域を確保するの小技が必要

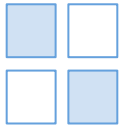
Zynq + PyCoRAM (+Debian) 入門

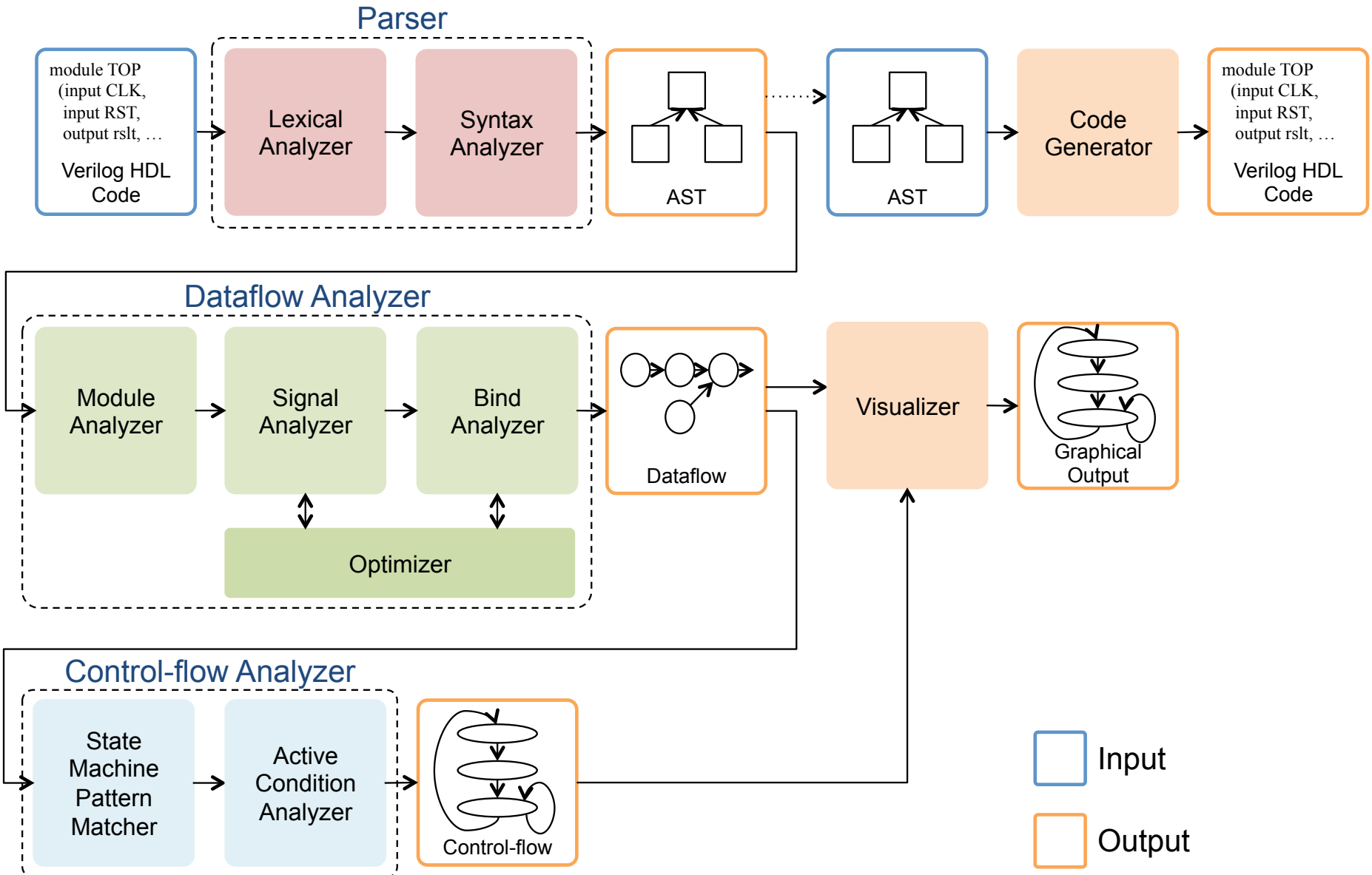
- SlideShareでチュートリアルスライド公開中
  - http://www.slideshare.net/shtaxxx/zynqpycoram
- 今時のFPGAアクセラレータを作る方法の一例をまとめました
  - PyCoRAM IPコアの作り方
  - Zynq (ARM搭載FPGA)の上でDebian Linuxを動作させるには
  - その上でPyCoRAM IPコアを使うにはどんなSWが必要か
- 2015年3月時点の情報なのでそろそろ更新予定
  - Debian 7.0から8.0へ移行
  - HW用メモリ領域確保の方法更新



Pyverilog & Veriloggen

Pyverilog: Verilog HDL解析・生成ツールキット





構文解析・抽象構文木(AST)生成

Verilog HDL
ソースコード

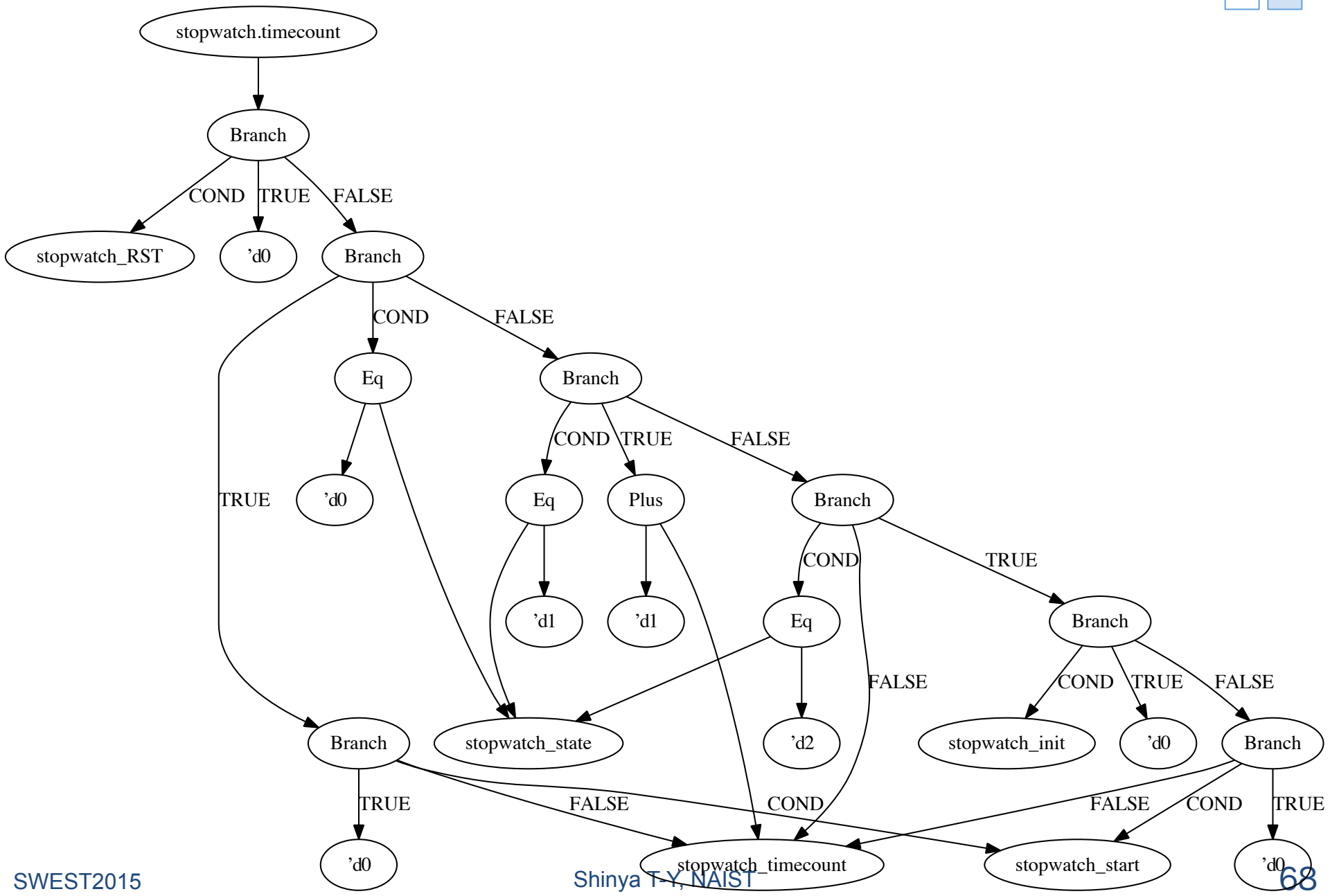
```
1  module stopwatch
2    (
3     input CLK,
4     input RST,
5     input start,
6     input stop,
7     input init,
8     output reg busy,
9     output reg [31:0] timecount
10   );
11   localparam IDLE = 0;
12   localparam COUNTING = 1;
13   localparam WAITINIT = 2;
14   reg [3:0] state;
15   always @(posedge CLK) begin
16     if(RST) begin
17       state <= 0;
18       timecount <= 0;
19     end else begin
20       if(state == IDLE) begin
21         if(start) begin
22           state <= COUNTING;
23           timecount <= 0;
24           busy <= 1;
25         end
26       end else if(state == COUNTING) begin
27         timecount <= timecount + 1;
28         if(stop) begin
29           state <= WAITINIT;
30           busy <= 0;
31         end
32       end else if(state == WAITINIT) begin
33         if(init) begin
34           timecount <= 0;
35           state <= IDLE;
36         end else if(start) begin
37           timecount <= 0;
38           state <= COUNTING;
39         end
40       end
41     end
42   end
43 endmodule
```

変換

AST

```
Source:
  Description:
    ModuleDef: stopwatch
      Paramlist:
      Portlist:
        Ioport:
          Input: CLK, False
            Width:
              IntConst: 0
              IntConst: 0
        Ioport:
          Input: RST, False
            Width:
              IntConst: 0
              IntConst: 0
        Ioport:
          Input: start, False
            Width:
              IntConst: 0
              IntConst: 0
        Ioport:
          Input: stop, False
            Width:
              IntConst: 0
              IntConst: 0
        Ioport:
          Input: init, False
            Width:
              IntConst: 0
              IntConst: 0
        Ioport:
          Output: busy, False
            Width:
              IntConst: 0
              IntConst: 0
          Reg: busy, False
            Width:
              IntConst: 0
              IntConst: 0
        Ioport:
          Output: timecount, False
            Width:
              IntConst: 31
              IntConst: 0
          Reg: timecount, False
            Width:
              IntConst: 31
              IntConst: 0
```

データフロー解析

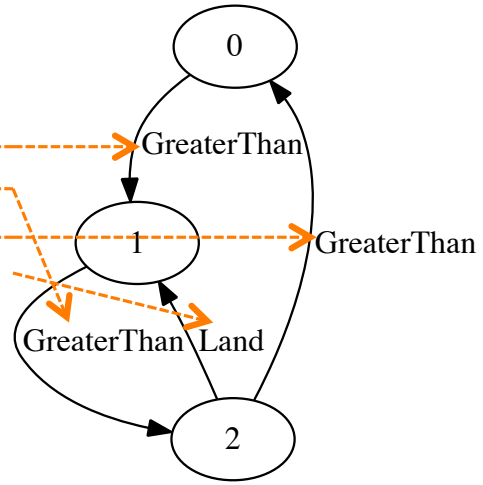

stopwatch.timecount
Branch
COND
TRUE
FALSE
stopwatch_RST
'd0
Branch
COND
FALSE
Eq
Branch
TRUE
'd0
COND
TRUE
FALSE
Eq
Plus
Branch
'd1
'd1
COND
TRUE
Eq
Branch
FALSE
'd2
COND
TRUE
FALSE
Branch
stopwatch_state
stopwatch_init
'd0
Branch
TRUE
FALSE
COND
FALSE
COND
TRUE
'd0
stopwatch_timecount
stopwatch_start
'd0

制御フロー（状態遷移）解析

- 状態遷移（いつ・どこからどこへ）を解析

```
# SIGNAL NAME: stopwatch.state
# DELAY CNT: 0
0 --(stopwatch_start>'d0)--> 1
1 --(stopwatch_stop>'d0)--> 2
2 --(stopwatch_init>'d0)--> 0
2 --((!(stopwatch_init>'d0))&&(stopwatch_start>'d0))--> 1
Loop
(0, 1, 2)
(1, 2)
```

(a) Command Line Output



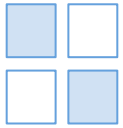
(b) Graphical Output

- ある信号が活性化する条件を解析

```
Active Conditions: stopwatch.busy
[((stopwatch_start 1:None) && (stopwatch_state 0:0))]
Changed Conditions
[((stopwatch_start 1:None) && (stopwatch_state 0:0)), ((stopwatch_state 1:1) && (stopwatch_stop 1:None))]
Changed Condition Dict
[(((stopwatch_start 1:None) && (stopwatch_state 0:0)), 'd1), (((stopwatch_state 1:1) && (stopwatch_stop 1:None)), 'd0)]
```

Condition Signal — *Condition's Min/Max Values* — *AND Condition* — *Target Signal* — *Target's Assigned Value*

PythonでVerilog HDLのコード生成



Python

```
1  import pyverilog.vparser.ast as vast
2  from pyverilog.ast_code_generator.codegen import ASTCodeGenerator
3
4  params = vast.Paramlist(())
5  clk = vast.Ioport( vast.Input('CLK') )
6  rst = vast.Ioport( vast.Input('RST') )
7  width = vast.Width( vast.IntConst('7'), vast.IntConst('0') )
8  led = vast.Ioport( vast.Output('led', width=width) )
9  ports = vast.Portlist( (clk, rst, led) )
10 items = ( vast.Assign( vast.Identifier('led'), vast.IntConst('8') ) ,)
11 ast = vast.ModuleDef("top", params, ports, items)
12
13 codegen = ASTCodeGenerator()
14 rslt = codegen.visit(ast)
15 print(rslt)
```

Execute

Verilog HDL

```
1  module top
2    (
3     input [0:0] CLK,
4     input [0:0] RST,
5     output [7:0] led
6    );
7     assign led = 8;
8  endmodule
```

これをベースに
抽象度の高いHDLを作りたい

Veriloggen

Veriloggen

PythonでVerilog HDLを組み立てるライブラリ

- Pythonで書いた動作をHDLに変換する
  高位合成ではない
- Pythonのオブジェクトとしてverilogの信号や代入を組み上げていく
- 当該オブジェクトの to_verilog() を呼ぶとVerilogのソースコードのテキストに変換

Python

```
import sys
import os
from veriloggen import *

def mkLed():
    m = Module('blinkled')
    width = m.Parameter('WIDTH', 8)
    clk = m.Input('CLK')
    rst = m.Input('RST')
    led = m.OutputReg('LED', width)
    count = m.Reg('count', 32)

    m.Always(Posedge(clk))(
        If(rst)(
            count(0)
        ).Else(
            If(count == 1023)(
                count(0)
            ).Else(
                count(count + 1)
            )
        ))

    m.Always(Posedge(clk))(
        If(rst)(
            led(0)
        ).Else(
            If(count == 1024 - 1)(
                led(led + 1)
            )
        ))

    return m

if __name__ == '__main__':
    led = mkLed()
    # led.to_verilog(filename='tmp.v')
    verilog = led.to_verilog()
    print(verilog)
```

実行

Verilog HDL

```
module blinkled #
  (
   parameter WIDTH = 8
   )
   (
   input [0:0] CLK,
   input [0:0] RST,
   output reg [(WIDTH - 1):0] LED
   );
  reg [(32 - 1):0] count;
  always @(posedge CLK)
    begin
      if(RST) begin
        count <= 0;
      end
      else begin
        if((count == 1023)) begin
          count <= 0;
        end
        else begin
          count <= (count + 1);
        end
      end
    end
  always @(posedge CLK)
    begin
      if(RST) begin
        LED <= 0;
      end
      else begin
        if((count == 1023)) begin
          LED <= (LED + 1);
        end
      end
    end
endmodule
```

例）たくさんLEDを追加してみる

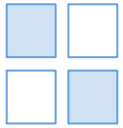
```python
import sys
import os
from veriloggen import *

def mkLed():
    m = Module('blinkled')
    width = m.Parameter('WIDTH', 8)
    clk = m.Input('CLK')
    rst = m.Input('RST')

    # function
    def add_led(m, postfix, limit=1024):
        led = m.OutputReg('LED'+postfix, width)
        count = m.Reg('count'+postfix, 32)

        m.Always(Posedge(clk))(
            If(rst)(
                count(0)
            ).Else(
                If(count == limit - 1)(
                    count(0)
                ).Else(
                    count(count + 1)
                )
            ))

        m.Always(Posedge(clk))(
            If(rst)(
                led(0)
            ).Else(
                If(count == limit - 1)(
                    led(led + 1)
                )
            ))

    for i in range(10):
        add_led(m, '_' + str(i), limit=i*10)

    return m

if __name__ == '__main__':
    led = mkLed()
    # led.to_verilog(filename='tmp.v')
    verilog = led.to_verilog()
    print(verilog)
```

実行

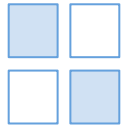
```
module blinkled #
  (
   parameter WIDTH = 8

  )
  (
   input [0:0] CLK,
   input [0:0] RST,
   output reg [(WIDTH - 1):0] LED_0,
   output reg [(WIDTH - 1):0] LED_1,
   output reg [(WIDTH - 1):0] LED_2,
   output reg [(WIDTH - 1):0] LED_3,
   output reg [(WIDTH - 1):0] LED_4,
   output reg [(WIDTH - 1):0] LED_5,
   output reg [(WIDTH - 1):0] LED_6,
   output reg [(WIDTH - 1):0] LED_7,
   output reg [(WIDTH - 1):0] LED_8,
   output reg [(WIDTH - 1):0] LED_9

  );
  reg [(32 - 1):0] count_0;
  reg [(32 - 1):0] count_1;
  reg [(32 - 1):0] count_2;
  reg [(32 - 1):0] count_3;
  reg [(32 - 1):0] count_4;
  reg [(32 - 1):0] count_5;
  reg [(32 - 1):0] count_6;
  reg [(32 - 1):0] count_7;
  reg [(32 - 1):0] count_8;
  reg [(32 - 1):0] count_9;
```

```
  always @(posedge CLK)
    begin
      if(RST) begin
        count_9 <= 0;
      end
      else begin
        if((count_9 == 89)) begin
          count_9 <= 0;
        end
        else begin
          count_9 <= (count_9 + 1);
        end
      end
    end
  always @(posedge CLK)
    begin
      if(RST) begin
        LED_9 <= 0;
      end
      else begin
        if((count_9 == 89)) begin
          LED_9 <= (LED_9 + 1);
        end

      end
    end
endmodule
```

Veriloggenで何ができるか？

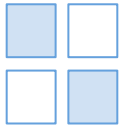


- Verilog HDLのメタプログラミングができる
  - 例）回路生成するレシピをPython+Veriloggenで定義して
    アプリケーション規模に応じて回路を自動生成
    - Verilog HDLにはgenerate構文があるか機能は限定的
- 独自の高位合成コンパイラのバックエンドとして使える
  - （PyCoRAMではない）Pythonベースの高位合成ツールとか？
  - Post PyCoRAMを現在開発中・・・こうご期待
    - 制御とパイプラインを柔軟に結合する仕組み
    - 既存のVerilog HDL資源を再利用する仕組み



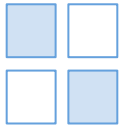
今後の高位合成ツールはどうあるべきか？

私が思う今後の高位合成の要件（まじめな話）



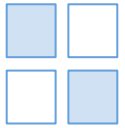
- オープンソースである
  - 普段よく使うコンパイラの多くはオープンソース
    - GCCやLLVM、最近は.NETすらオープンソースに
  - 誰でもすぐ導入できるのは正義
    - 今はFPGAの評価ボードも安価である：2万5千円でZyboが買える
- ソフトウェア開発者が自然に導入できる
  - 記述モデル：結局、手続き型＋オブジェクト指向がよい？
    - より多くの人が使える
    - 慣れ親しんだプログラミングモデル＝手になじんでいる
  - 多少のディレクティブは許容される
    - GPUではCUDAが当たり前
    - FPGAでもOpenACC，OpenMPやMPI程度なら許されるか？
      - Xilinx SDSoCは近いかも？

私が思う今後の高位合成の要件（まじめな話）



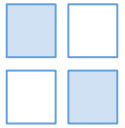
- 高位合成系におけるLLVM的環境
  - データフローを入力したらスケジューリング・アロケーションしてくれるフレームワーク・足回り
    - LLVM: LLVM-IRを入力に最適化したり機械語を吐いたり
    - DFGレベルのスケジューリング・最適化をやってくれる
  - 言語や記述モデルでオリジナリティを出す
    - 粗粒度・スレッドレベルなスケジューリングは自分で

今日のテーマ



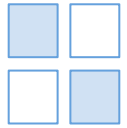
- 「好きな道具」で「好きなもの」を作る世界
  - 発表者の場合
    - 好きな道具: 好きな言語（Python）・モデル・書き方
    - 好きなもの: FPGAを使ったカスタムコンピュータ
- 「こんな風に設計できたらいいな」を実現しよう
  - 自分の手になじむ道具を実際に作ろう、そして公開しよう
  - 発表者の場合
    - PyCoRAM: PythonによるIPコア高位設計フレームワーク
    - Pyverilog: Verilog HDL解析・生成ツールキット
    - Veriloggen: PythonでVerilog HDLを組み立てる軽量ライブラリ

私が思う今後の高位合成の要件（気楽な話）



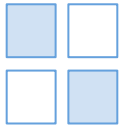
- 好きな言語でハードウェア開発できる
  - ソフトウェア開発は用途と好みに応じて言語を選べる
    - C, C++, C#, Java, Python, Ruby, Perl, JavaScrit, Scala, Go, Haskell
  - ハードウェア開発は？→選択肢が少ない
    - RTL: Verilog HDL, VHDL
    - 高級HDL: Chisel (Scala DSL), PyMTL (Python DSL), Veriloggen
    - 高位合成: C, C++, OpenCL, Java (Synthesijer), Python (PyCoRAM)
- 「好きな道具」で「好きなもの」を作る世界
  - 高位設計 ≠ C設計（Cベースが現状一番良いのは認める）
  - でもRubyで書きたい！Goで書きたい！Pythonで書きたい！
    - 発表者はPythonで書きたいのでこれからもいろいろやってみます
    - でも他の言語も検討中です

「こんな風に設計できたらいいな」を実現しよう



- 自分が使いたい道具を自分で作ろう
  - こんな風に回路が書けたらいいな
  - あの言語でハードウェアを生成したい
  - でもそんな道具ない・・・じゃあ作りましょう！
    - 公開したら誰かが使ってくれるかも？
- （特に学生さん）研究になるか？
  - 研究：基礎研究・応用研究・開発研究
    - 新しいもの・自分がワクワクするものを開発して
      ああでもないこうでもないと検証していくのは開発研究
  - 新規性とか有用性とかは？
    - 新規性：ちゃんと作れば何かしらある
    - 有用性：少なくとも自分には有用

まとめ



- FPGA・高位合成の基礎と
  Pythonによる高位設計フレームワークについて
- 今後の高位合成ツールはどうあるべきか？
  - 「こんな風に設計できたらいいな」を実現しよう
- 各種ツールはGitHubで公開中
  - PyCoRAM: https://github.com/PyHDI/PyCoRAM
  - Pyverilog: https://github.com/PyHDI/Pyverilog
  - Veriloggen: https://github.com/PyHDI/veriloggen