■ PYNQ/KV260 のビルド

GitHub で公開されてている Kria-PYNQ の base をビルドしてみる.
ライセンスは https://github.com/Xilinx/Kria-PYNQ/blob/main/LICENSE によると
BSD 3-Clause License なので，オリジナル作るときはこれをベースにすすめればよさそう.

```
git clone https://github.com/Xilinx/Kria-PYNQ.git
cd Kria-PYNQ
git checkout v3.0
git submodule update --init --recursive
source /tools/Xilinx/Vitis/2020.2/settings64.sh
kv260/base
make
```

Vivado/Vitis HLS の日付のバグを踏むので
https://support.xilinx.com/s/article/76960
をちゃんと当てとかないとダメ

また，2020.2.2 のアップデートをインストールしとくのと
KV260 のボードファイルを用意しておく必要があるのは注意.
ボードファイルは,
https://github.com/Xilinx/XilinxBoardStore/tree/2020.2.2/boards/Xilinx/kv260
を /tools/Xilinx/Vivado/2020.2/data/boards/board_files にコピーしとく.

ビルドがおわると，PYNQ で PL をオーバーレイするのに必要な

- base.bit
- base.hwh
- base.dtbo

が生成される.

base/base.xpr を開いてブロックデザインをみてみると↓のような感じ.
ロゴが入っているのが ZYNQ UltraSCALE+.
左側の濃いブロックが iop_pmod0 で PMOD につながっていて,
右側の濃いブロックは mipi.

iop_pmod0 の中身はこんな感じ．中に MicroBlaze がいる.

mipi の中身はこんな感じ．Vitis HLS で作ったモジュール達が入っている.
入力は mipi_phy_if で外から入っていて,
計算した結果は AXI Video Direct Memory Access を介して
Zynq UltraSCALE+ の S_AXI_HPI_FPD に入力されている.

■オリジナル IP の組み込み

Kria-PYNQ/base にオリジナルの IP を組み込んでみる.

```
mkdir myipcore
cd myipcore
source /tools/Xilinx/Vitis/2020.2/settings64.sh
vitis_hls
```

とかして，Vitis HLS 開いて，プロジェクトを作る．

プロジェクトを作ったら，たとえば，こんな感じの関数を vector_add.c に実装して，

```
#include "vector_add.h"

void vector_add(int a[128], int b[128], int c[128]){
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=b
#pragma HLS INTERFACE s_axilite port=c
  for(int i = 0; i < 128; i++){
    c[i] = a[i] + b[i];
  }
}
```

こんな感じのテストベンチを vector_add_tb.c に実装．

```
#include <stdio.h>
#include "vector_add.h"

int main(int argc, char **argv){
  int a[128];
  int b[128];
  int c[128];
  int c_sw[128];

  int i;

  for(i = 0; i < 128; i++){
    a[i] = i;
    b[i] = i;
    c[i] = 0;
    c_sw[i] = a[i] + b[i];
  }

  vector_add(a, b, c);

  for(i = 0; i < 128; i++){
    if(c_sw[i] != c[i]){
      printf("%d: expected %d, but the actual %d\n", i, c_sw[i], c[i]);
      return 1;
    }
  }
  return 0;
}
```

C Simulation -> C Synthesis -> Co-Simulation -> Export RTL する．
仮引数の a, b, c は，それぞれ s_axi_control のオフセット 512, 1024, 1536 にマッピングされた．

関数の制御 / ステータス信号および仮引数はすべて一つの AXI4-Lite な Slave にまとめられる．
メモリマップは

```
//------------------------Address Info-------------------
// 0x000 : Control signals
//         bit 0  - ap_start (Read/Write/COH)
//         bit 1  - ap_done (Read/COR)
//         bit 2  - ap_idle (Read)
//         bit 3  - ap_ready (Read)
//         bit 7  - auto_restart (Read/Write)
//         others - reserved
// 0x004 : Global Interrupt Enable Register
```

```
//         bit 0  - Global Interrupt Enable (Read/Write)
//         others - reserved
// 0x008 : IP Interrupt Enable Register (Read/Write)
//         bit 0  - enable ap_done interrupt (Read/Write)
//         bit 1  - enable ap_ready interrupt (Read/Write)
//         others - reserved
// 0x00c : IP Interrupt Status Register (Read/TOW)
//         bit 0  - ap_done (COR/TOW)
//         bit 1  - ap_ready (COR/TOW)
//         others - reserved
// 0x200 ~
// 0x3ff : Memory 'a' (128 * 32b)
//         Word n : bit [31:0] - a[n]
// 0x400 ~
// 0x5ff : Memory 'b' (128 * 32b)
//         Word n : bit [31:0] - b[n]
// 0x600 ~
// 0x7ff : Memory 'c' (128 * 32b)
//         Word n : bit [31:0] - c[n]
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

という感じ.

生成した IP コアを IPI に組み込むために,
IP カタログのリポジトリに myipcore ディレクトリを追加してコアのインスタンスを生成して,
M_AXI_HPM0_LPD から制御される ps8_0_axi_periph なる AXI Interconnect の先にぶらさげる.

で，Generate Bitstream で，しばらく待つと,

- kv260/base/base/base.runs/impl_1/base_wrapper.bit
- kv260/base/base/base.gen/sources_1/bd/base/hw_handoff/base.hwh

ができる.

両方とも Kria に転送して,
たとえば，/home/root/jupyter_notebooks の下に myipcore とか作って置いておく.

あとは，Juypter 環境で，同じフォルダに Python3 ノートを作って,

```
from pynq import Overlay
base = Overlay("./myipcore.bit")
from pynq import MMIO
mmio = MMIO(base_addr=base.ip_dict['vector_add_0']['phys_addr'], length=0x1000, debug=True)
```

として追加した vector_add_0 へのハンドラ (メモリ領域へのアクセサ) を取得する.

```
mmio.read(0)
```

とかすると，ap_idle に相当する 4 が返ってくる.

```
for i in range(128):
    mmio.write(512+4*i, i)
    mmio.write(1024+4*i, i)
    mmio.write(1536+4*i, 0)
```

として，仮引数の a, b, c に相当する領域を初期化した後,

```
mmio.write(0, 1)
```

で，処理の実行開始．処理はすぐに終わるはずなので，

```
mmio.read(0)
```

とすると，今度は ap_ready と ap_idle が立った状態の 6 が返ってくる．

```
for i in range(128):
    print(mmio.read(1536+4*i))
```

で，c の先のメモリを読むと，

```
0
2
4
6
8
10
12
14
16
...
```

と，a と b を足した値が格納されていて，正しく処理が実行できたことがわかる．