Xilinx SDSoC (2016.3版)

解体新書

ソフトウェア編

@Vengineer

SDSoC勉強会 (2017/1/28)

Vengineer DEATH
無限ゲームのなか

いつものように、
よろしくお願いします。

@Vengineer に居ます

本資料では、
SDSoC™ 2016.3が生成する
ソフトウェアについて
調べたのをまとめたものです。

ご利用は、自己責任でお願いします。

内容

- SDSoC™とは
- SDSoCプラットフォーム
- ソフトウェア構成

- 例題で確認しよう
  Zyboでmmult+maddを実装

SDSoC™とは

SDSoC™

Trademark付いていますよ
®ではない

Software-Defined Development Environment for System-on-Chip

SDSoC システムコンパイラをフロントエンドとして、Vivado HLS/Vivado Design Suiteを使って、HDL => FPGA Bitstreamの生成だけでなく、FPGAの部を制御するためのソフトウェアも自動生成するという優れもの！

いやー、びっくりぽんや、ですわ。(古い？)

SDSoC
プラットフォーム

SDSoCプラットフォーム

SDSoCを使うためには、ターゲットボード用のSDSoCプラットフォームを作成しないといけない

SDAccelやAltera SDK for OpenCLでも同じようにターゲットボード用にプラットフォームを用意する必要があるので、特別なことではない

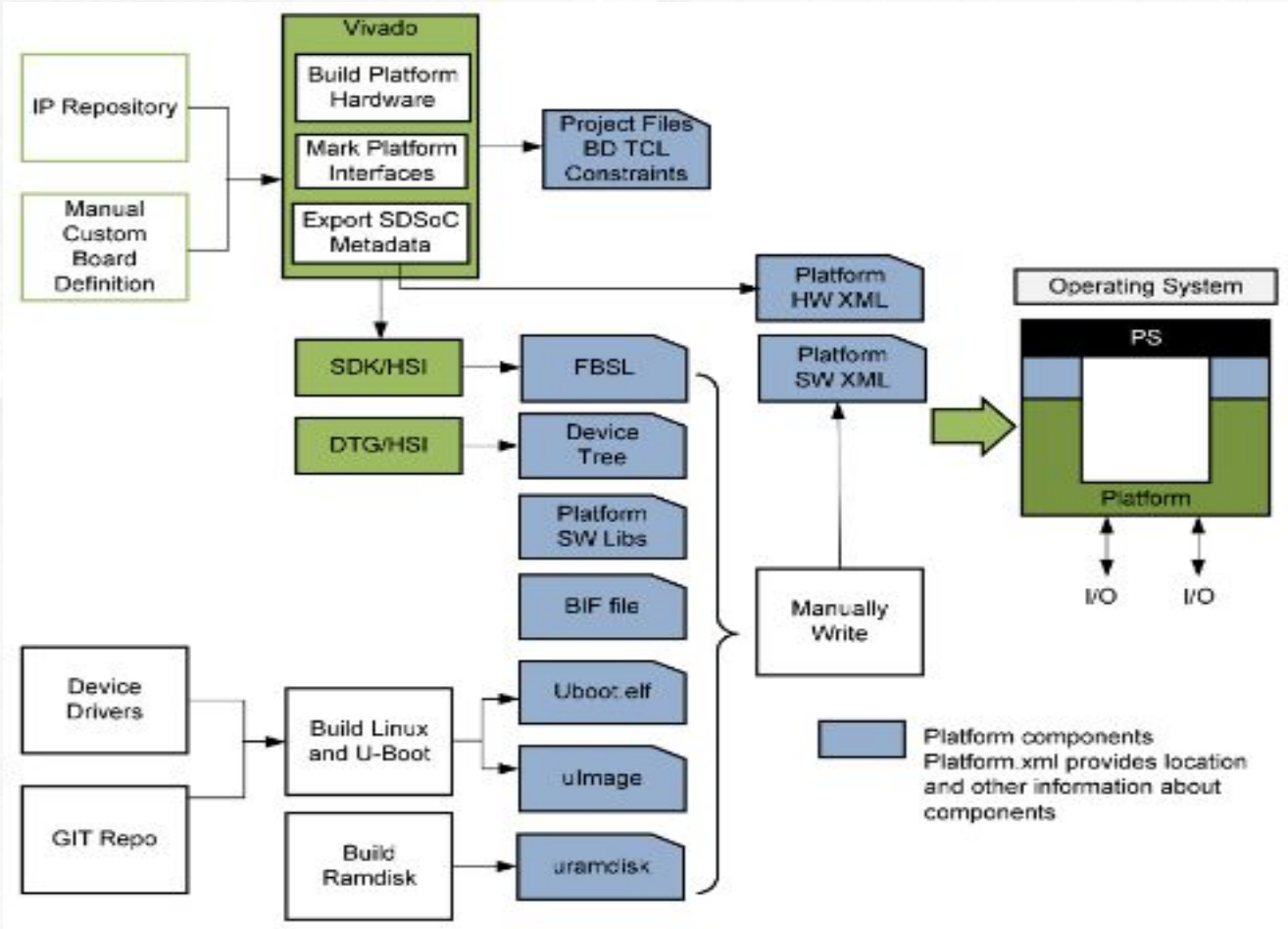
SDSoCプラットフォーム

各ボードに対応したものが必要

2016.3のサンプルプラットフォーム

- microzed
- zc702
- zc706
- zcu102 (Zynq Ultrascale+ MPSoC)
- zed
- zybo

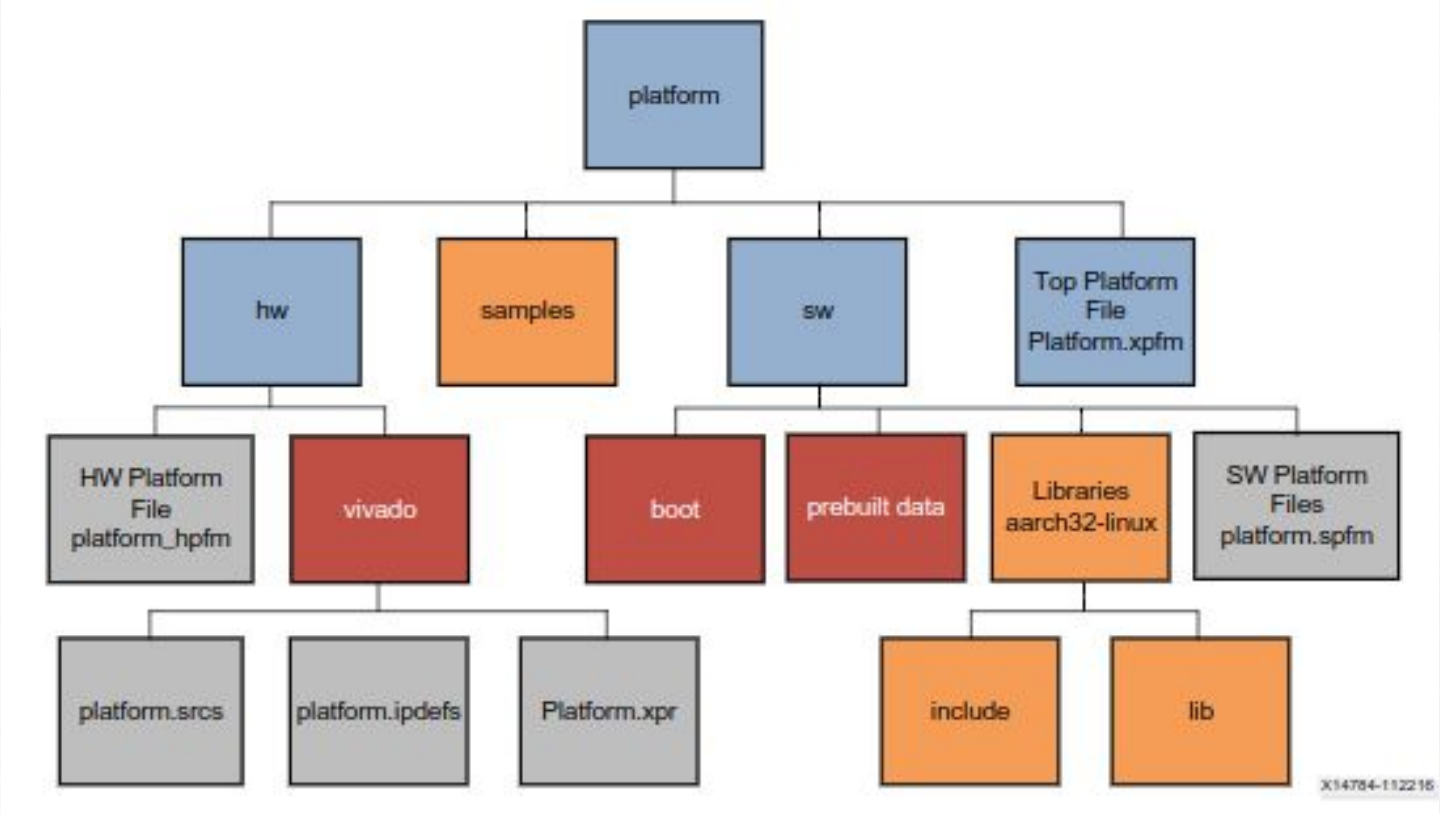
SDSoCプラットフォーム



SDSoCプラットフォーム

- メタデータファイル
  - Vivadoツールを使用して生成されたプラットフォームハードウェア記述
  - 手動で記述したプラットフォームソフトウェア記述ファイル
- Vivado Design Suiteプロジェクト
  - ソース/ 制約/IPブロック
- ソフトウェアファイル
  - ライブラリヘッダーファイル (オプション)
  - スタティックライブラリ (オプション)
  - Linux関連オブジェクト(デバイスツリー、U-Boot、Linuxカーネル、ramdisk)
- ビルド済みハードウェアファイル(オプション)
  - ビットストリーム
  - SDK用にエクスポートされたハードウェアファイル
  - 前もって生成されたデバイス登録およびポート情報ソフトウェアファイル
  - 前もって生成されたハードウェアおよびソフトウェアインターフェイスファイル

SDSoCプラットフォーム



SDSoCプラットフォーム

Zyboのディレクトリ

- hw
  - vivado
  - zybo.hpfm

    Vivadoツールを使用して生成されたプラットフォーム
    ハードウェア記述
- sw
  - aarch32-none/boot/freertos/image
    prebuild_platform/eqmu
  - zybo.spfm

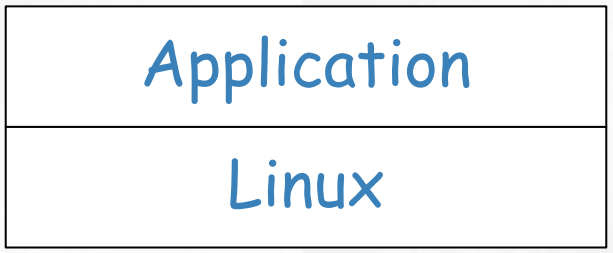
    手動で記述したプラットフォームソフトウェア記述ファイル
- zybo.xpfm

  zybo.hpfmとzybo.spfmのリファレンスへのxmlファイル

SDSoC 2016.2と
構成が変わったよ

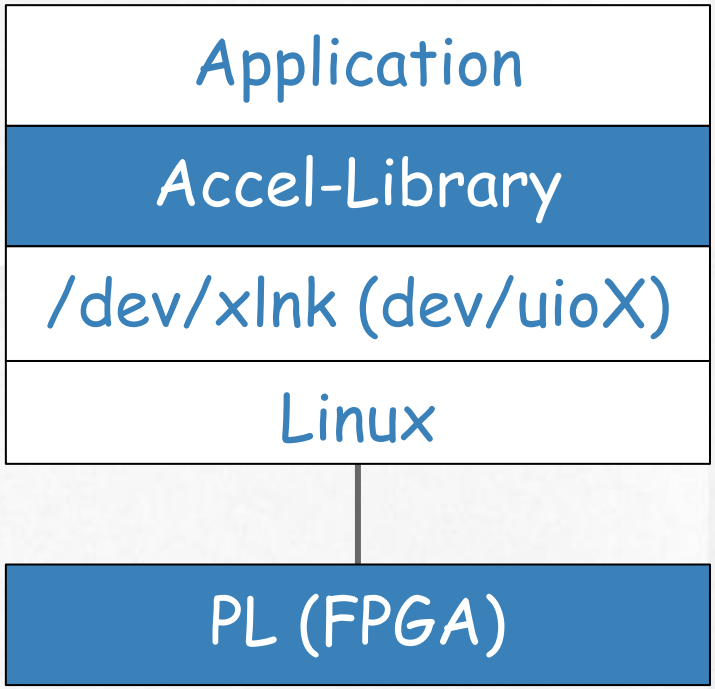
ソフトウェア構成

オリジナルプログラムでの構成



GCC (gcc/g++)にてコンパイル、必要なライブラリをリンクし、すべてをZynq内のCPUでプログラムとして実行する

SDSoCで実装したときの構成



生成されるファイル

- SW
  Accel-Library
- HW
  FPGA bitstream

/dev/uioX デバイスドライバについて

UIOとは

https://www.kernel.org/doc/htmldocs/uio-howto/about.html

UIOはユーザー空間でデバイスドライバを作成する仕組みユーザー空間でUIOを利用する際は、/dev/uio0をopenしてmmapすると、デバイスのレジスタ空間が見える

/dev/uio0にライトすると割り込みを許可する

/dev/uio0をreadすると割り込みが起きるまでブロックする

http://qiita.com/ikwzm/items/b22592c31cdbb9ab6cf7

/dev/xlnk デバイスドライバについて

Xilinx APF Accelerator driver

Linux(githubのXilinx/linux-xlnx)をコンフィギュレーションするときに、下記の設定をする必要がある

```
%git checkout -b sdsoc_release_tag xilinx-v2016.3-sdsoc
```

(SDSoCのパッケージ内のLinuxは、下記の設定済み)

```
CONFIG_STAGING=y
CONFIG_XILINX_APF=y       # drivers/staging/apf
CONFIG_XILINX_DMA_APF=y   # drivers/staging/apf
CONFIG_DMA_CMA=y
CONFIG_CMA_SIZE_MBYTES=256
CONFIG_CROSS_COMPILE="arm-linux-gnueabihf-"
CONFIG_LOCALVERSION="-xilinx-apf"
```

例題で確認しよう

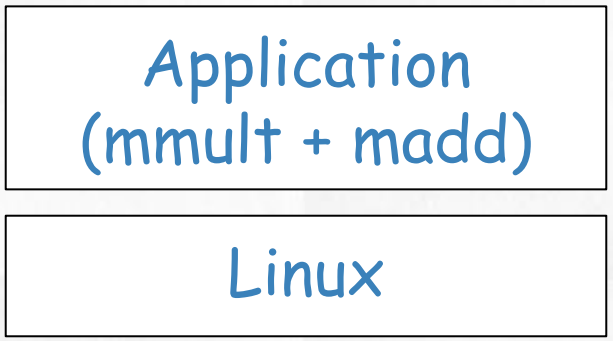
チュートリアルで確認

SDSoC 環境ユーザー ガイド
SDSoC 環境の概要(UG1028)

「第2章 チュートリアル：プロジェクトの作成、ビルド、実行」
で、[Matrix Multiplication and Addition]を
ハードウェア化してみよう！

Releaseではなく、Debugで

チュートリアルのプログラムでの構成



mmult(mmult.cpp) と madd(madd.cpp)を、
SdSoCを使ってハードウェア化しています

FPGAの部屋

SDSoC 2015.2 のチュートリアルをやってみた

http://marsee101.blog19.fc2.com/blog-entry-3212.html
http://marsee101.blog19.fc2.com/blog-entry-3213.html
http://marsee101.blog19.fc2.com/blog-entry-3214.html
http://marsee101.blog19.fc2.com/blog-entry-3215.html

Debugディレクトリ

生成されたファイルを見てみよう

Debugディレクトリ

- _sds : Accel-Library
- makefile
- labn.elf : 実行ファイル
- labn.elf.bit : FPGA部のbitstream
- object.mk
- removedFiles.sh : New
- sd_card : ブートに必要なファイル
- source.mk
- src : mmult/maddソースコード

sd_cardディレクトリ

Zynqでsd_cardに必要なものは？

- BOOT.BIN

以下のファイルは入っている

- fsbl.elf　　Booting
- u-boot.elf

- devicetree.dtb　　Linux
- uImage
- uramdisk.image.gz

参考資料
FPGA+SoC+Linuxのブートシーケンス(ZYNQ+Vivado編)
http://qiita.com/ikwzm/items/1614c35233e1836c7a26

http://www.wiki.xilinx.com/Zynq+2016.2+Release

SDSoCが出力するsd_card中身は？

- BOOT.BIN　　Booting
  (fsbl.elf + u-boot.elf)
- devicetree.dtb　　Linux
- uImage
- uramdisk.image.gz

- README.tx
- _sds/_p0_.bin　　FPGA bitstream
- labn.elf　　Application

ここがポイント！

_sds_ディレクトリ

_sdsディクリトリ

- iprepo　：生成したHWのIP
- p0　：Partition 0に必要なファイル全部
- reports　：レポート/ログファイル
- swstubs　：ソフトウェアのスタブファイル
- vhls　：Vivado HLS実行ディレクトリ

ソフトウェア編なのでp0/reports/swstubsについてのみ解析iprepo とvhlsは、ハードウェア編(作った時)に解析するかも

sds/p0ディレクトリ

sds/p0の中身

- ipi
- sd_card　　：SDカードの内容

SDSoC 2016.2と
構成が変わったよ

sds/p0/sd_cardの中身

- BOOT.BIN　　Booting
- devicetree.dtb　　Linux
- uImage
- uramdisk.image.gz

- README.tx
- boot.bif
- labn.elf
- labn.elf.bit.bin

boot.bifの内容

皆さんおなじみのbootgenスクリプト

```
/* /xxxxxx/Debug/_sds/p0/.boot/boot.bif */
/* Command to create bitstream .bin file:       */
/*   bootgen -image <bif_file> -split bin -w    */
/* Command to create BOOT.BIN file:             */
/*   bootgen  -image <bif_file> -w -o i BOOT.BIN */
/* linux */the_ROM_image:
{
  [bootloader]/xxxx/zybo/boot/fsbl.elf
  /yyyy/Debug/labn.elf.bit
  /xxxx/zybo/boot/u-boot.elf
}
```

sds/p0/sd_cardの中身

- BOOT.BIN　　Booting
  (fsbl.elf + u-boot.elf + mmultadd.elf.bit)
- devicetree.dtb　　Linux
- uImage
- uramdisk.image.gz

- README.tx
- boot.bif
- labn.bit.bin
- labn.elf

SDSoCが生成するポイントは！

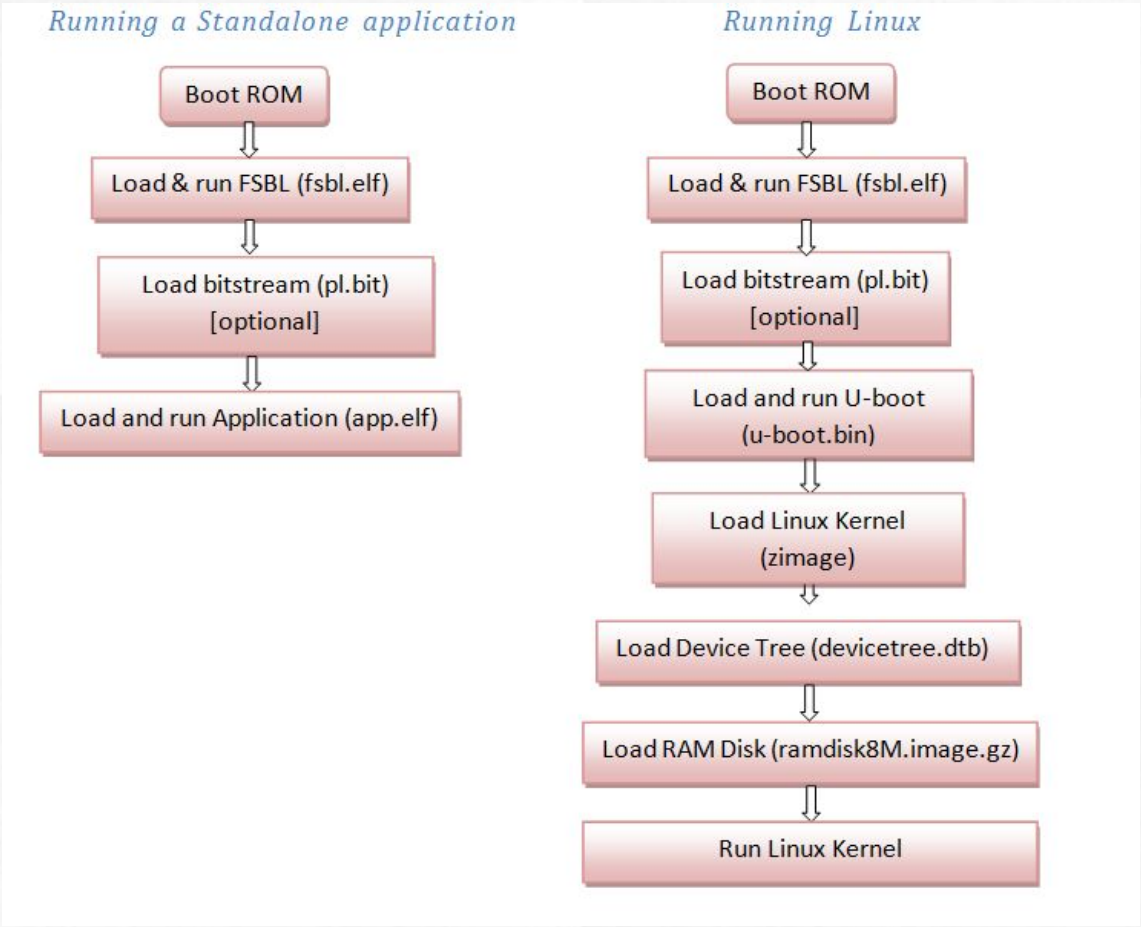
FPGA bitstream (labn.elf.bit.bin)

SDSoCプロントエンドで切り出したハードウェア部分を Vivado HLS/Vivado Design Suite を呼び出し生成する

Application (labn.elf)

SDSoCフロントエンドでライブラリとリンクして生成する

Boot ROM から Linux



http://xilinx.wikidot.com/zc702-boot-from-flash

sds/reportsディレクトリ

reportsディクリトリ

- data_motion.html
  生成されたデータモーション
- sds_madd.{jou,log,rpt} : madd
- sds_mmult.{jou,log,rpt}: mmult
- sds_main.{jou,log} : main
- sds.{jou,log,rpt} : Application
  .log : Log file
  .jou : Journal file
  .rpt : Report file

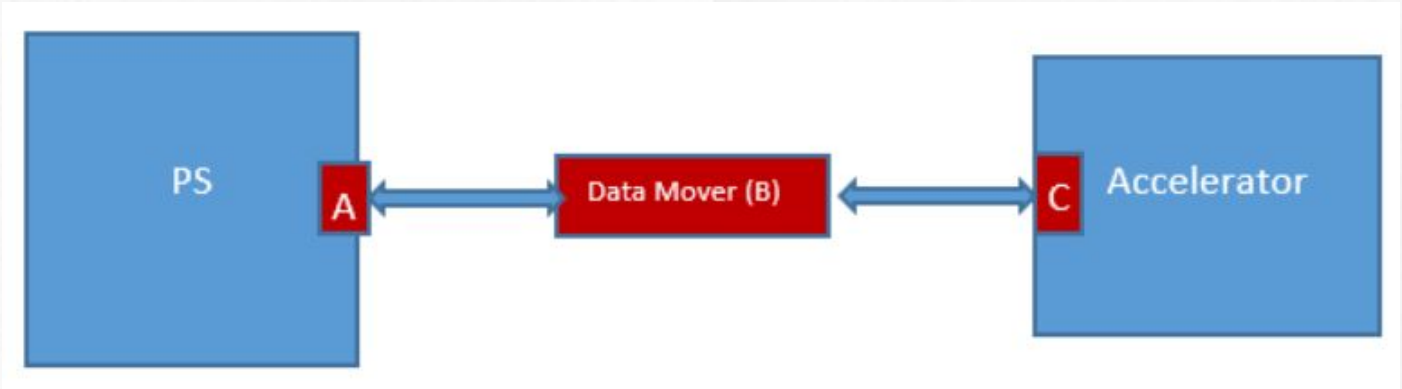
データモーション

SDSoCのデータ モーション ネットワークは、アクセラレータのハードウェア インターフェイス、PS とアクセラレータ間とアクセラレータ同士のデータ ムーバー、PS のメモリ システム ポートの 3 つのコンポーネントから構成される。

・A：システムポート
・B：データムーバー

AXIDMA_SG/AXIDMA_Simple/AXIDMA_2D/AXI_FIFO

・C：アクセラレータのポート



sds_madd.jou/sds_mmult.jou

フロントエンド

sds++

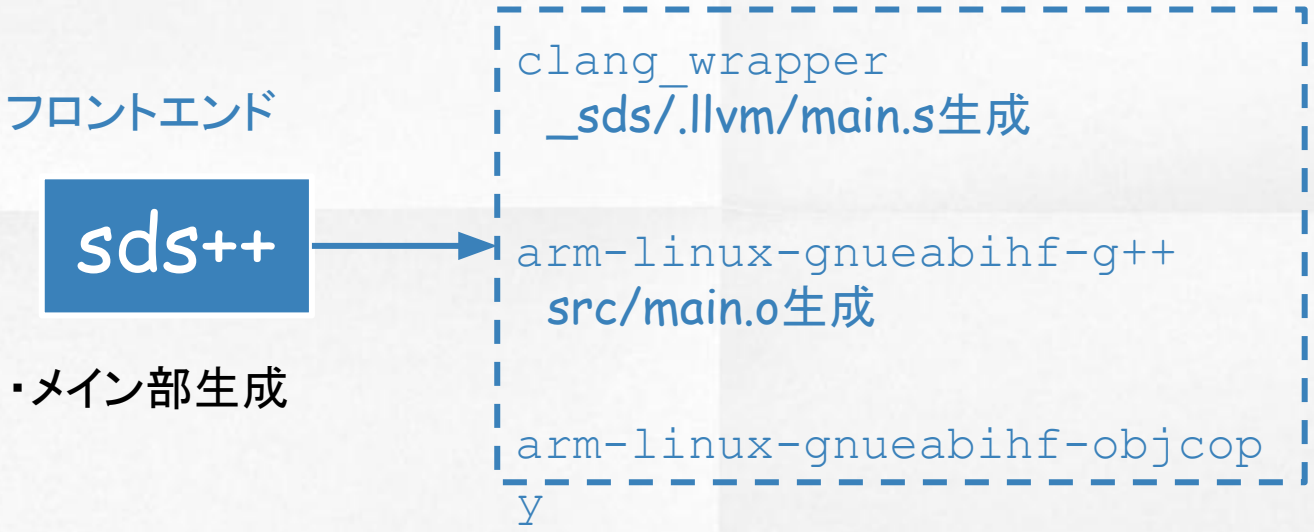
・ハードウェア部生成

```
clang_wrapper
  _sds/.llvm/mXXX.s生成
sdslint
arm-linux-gnueabihf-g++
arm-linux-gnueabihf-objcopy
  src/XXXX.o生成

clang_wrapper
pragma_gen
  tclファイル生成(xxx_run.tcl)
vivado_hls

arm-linux-gnueabihf-g++
arm-linux-gnueabihf-objcopy
```

sds_main.jou


フロントエンド
sds++
・メイン部生成
clang_wrapper
_sds/.llvm/main.s生成
arm-linux-gnueabihf-g++
src/main.o生成
arm-linux-gnueabihf-objcop
y

sds.jou

フロントエンド

sds++

・ラッパー部生成
・リンクし、
　アプリ生成
・FPGAビット
　ストリーム生成

```
stub_gen
  スタブファイル生成 (maddr/mmult)
arm-linux-gnueabihf-gcc
  devreg.c/portinfo.c
  cf_stab.c
  madd.cpp/mmult.cpp
arm-linux-gnueabihf-ar
  liblabn.a
arm-linux-gnueabihf-g++
arm-linux-gnueabihf-objcopy
  labn.elf
system-linker
  labn.elf.bit  (FPGA bitstream)
```

sds/sds++のオプション

```
sdscc | sds++
[hardware_function_options] [system_options]
[performance_estimation_options]
[options_passed_through_to_cross_compiler] [-mno-ir]
[-sds-pf platform_name] [-sds-pf-info platform_name]
[-sds-pf-list] [-target-os os_name]
[-verbose] [ -version] [--help] [files]
```

ハードウェア関数オプション

```
[-sds-hw function_name file [-clkid clock_id_number]
[-files file_list] [-hls-tcl hls_tcl_directives_file]
[-mno-lint] -sds-end]
```

パフォーマンス見積もりオプション

```
[[-perf-funcs function_name_list -perf-root function_name]
|
[-perf-est data_file][-perf-est-hw-only]]
```

SDSoC Environmento User Guide UG1027 (v2016.3) November 30, 2016, Page.93-105

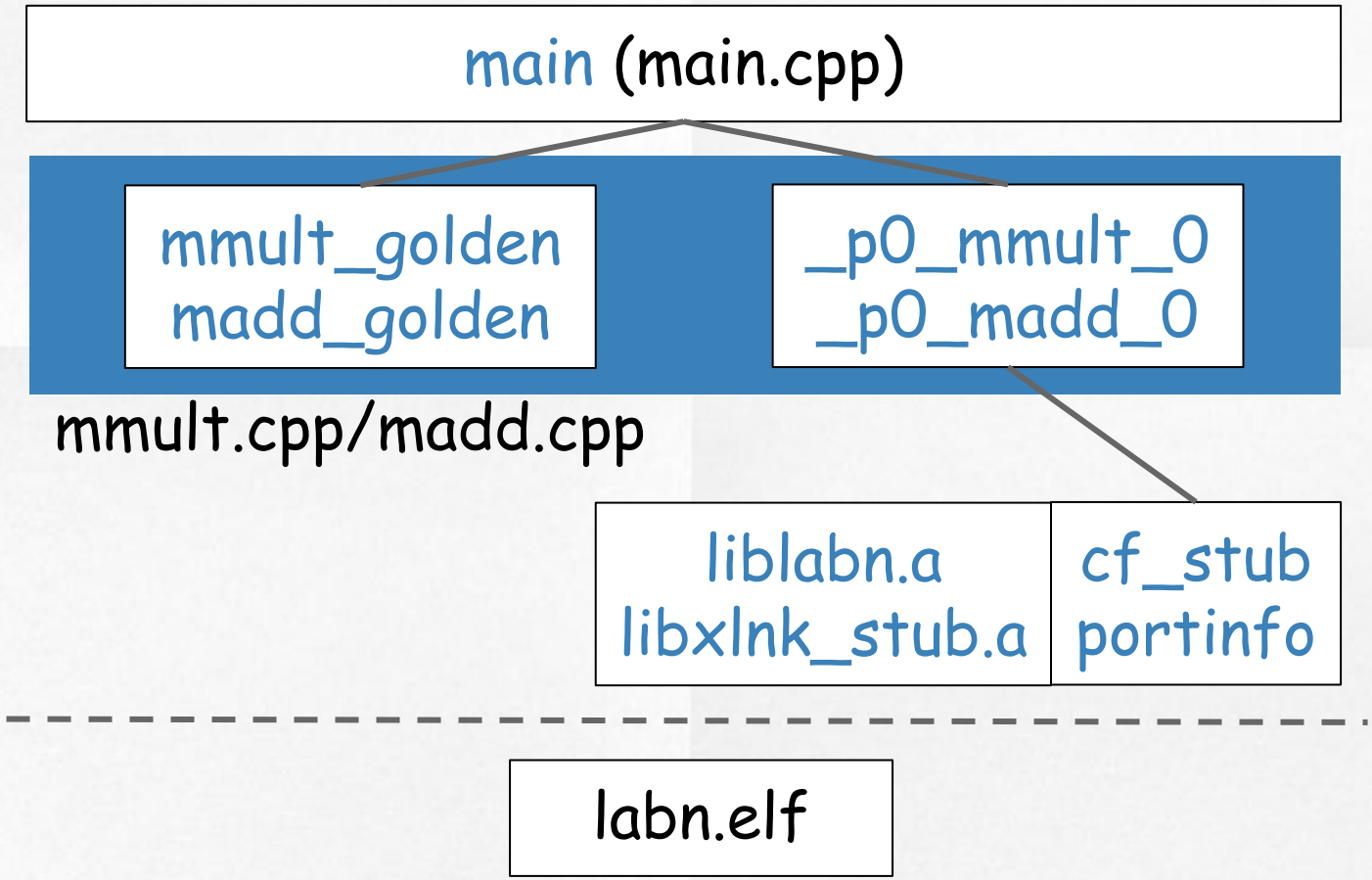
sds/swstubsディレクトリ

swstubsディクリトリ

- main.xxx : mainソースコード
- mmult.xxx : mmultソースコード
- madd.xxx : maddソースコード
- cf_stub.xxx : 生成されたスタブ
- portinfo.xxx : 生成されたスタブ
- liblabn.a :
- libxlnk_stub.a :
- labn.elf : 実行プログラム

SDSoC 2016.2と構成が変わったよ
devreg.xxxが無くなった

各ファイルの関係図


main (main.cpp)
mmult_golden
madd_golden
_p0_mmult_0
_p0_madd_0
mmult.cpp/madd.cpp
liblabn.a
libxlnk_stub.a
cf_stub
portinfo
labn.elf

libxlnk_stub.a

libxlnk_stub.aは、_sds/reports/sds.jou を見ると、portinfo.o/cf_stub.oのアーカイブになっている

```
arm-linux-gnueabihf-ar
   crs libxlnk_stub.a
   portinfo.o cf_stub.o
```

liblabn.a

liblabn.aは、_sds/reports/sds.jou を見ると、portinfo.o/cf_stub.o/mmult.o/madd.oのアーカイブになっている

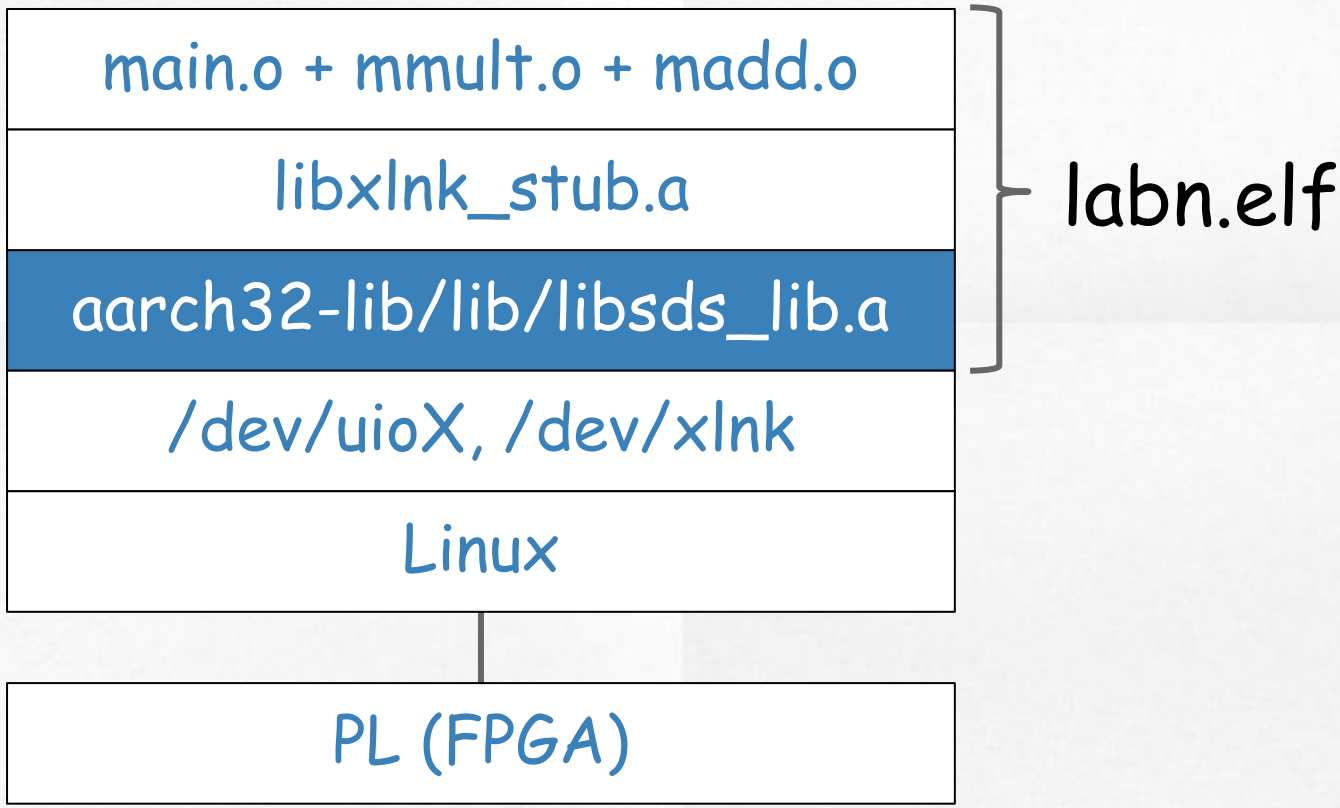
```
arm-linux-gnueabihf-ar
   crs liblabn.a
   portinfo.o cf_stub.o
   mmult.o madd.o
```

labn.elf

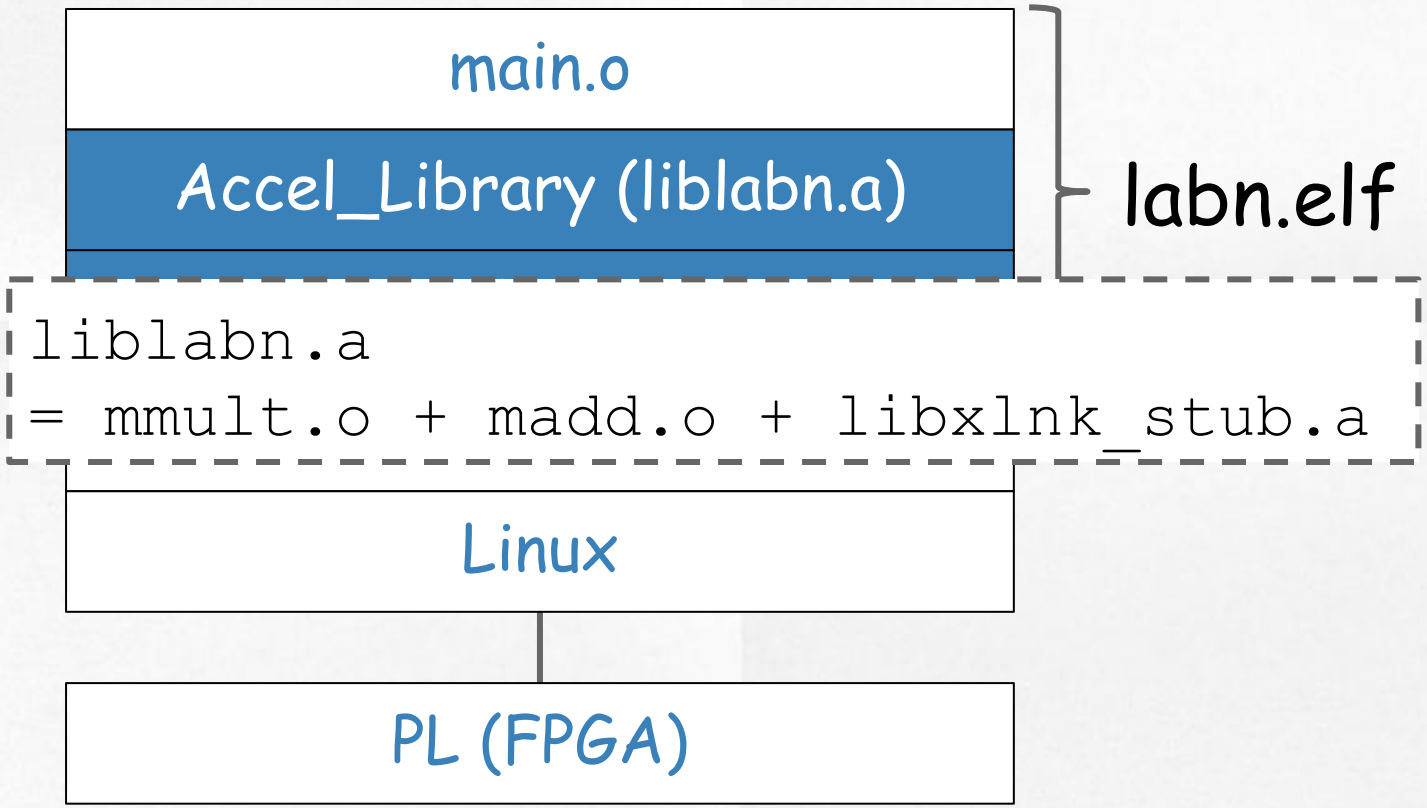
labn.elfは、_sds/reports/sds.jou を見ると、
main.o/mmult.o/madd.o/libxlnk_stub.aに、
libsds_libをリンクしている

```
arm-linux-gnueabihf-g++
   madd.o mmult.o main.o
   -L 2016.3/aarch32-linux/lib -Lswstubs
   -Wl,--start-group  -Wl,--end-group -Wl,--start-group
   -lpthread -lsds_lib -lxlnk_stub
   -Wl,--end-group -o labn.elf
```

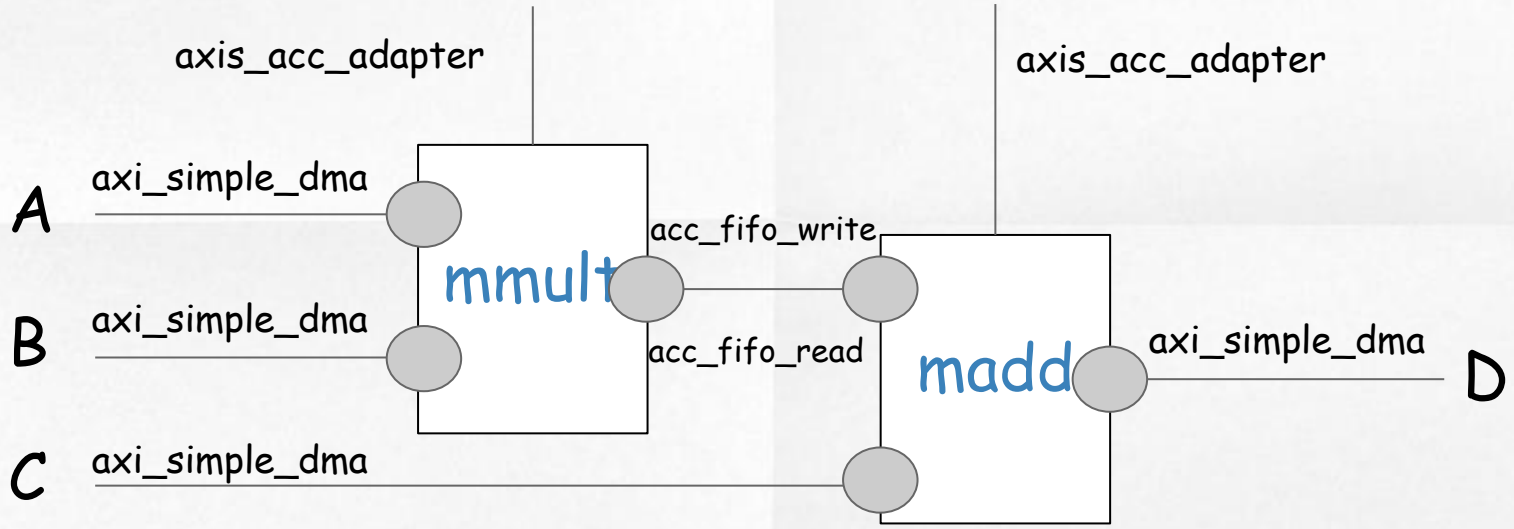
例題をSDSoCで実装したときの構成



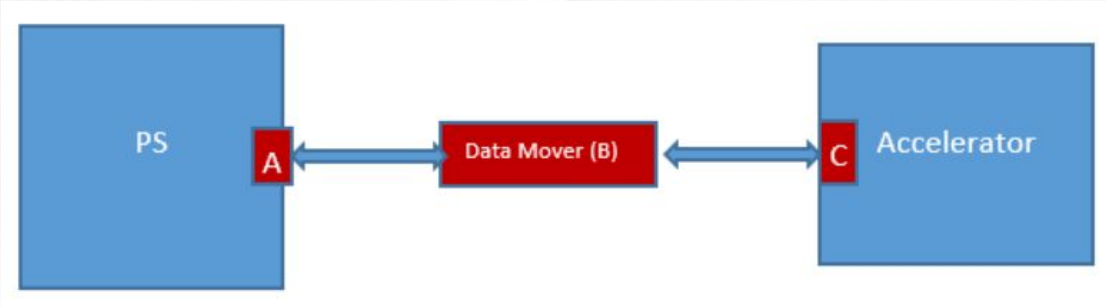
例題をSDSoCで実装したときの構成

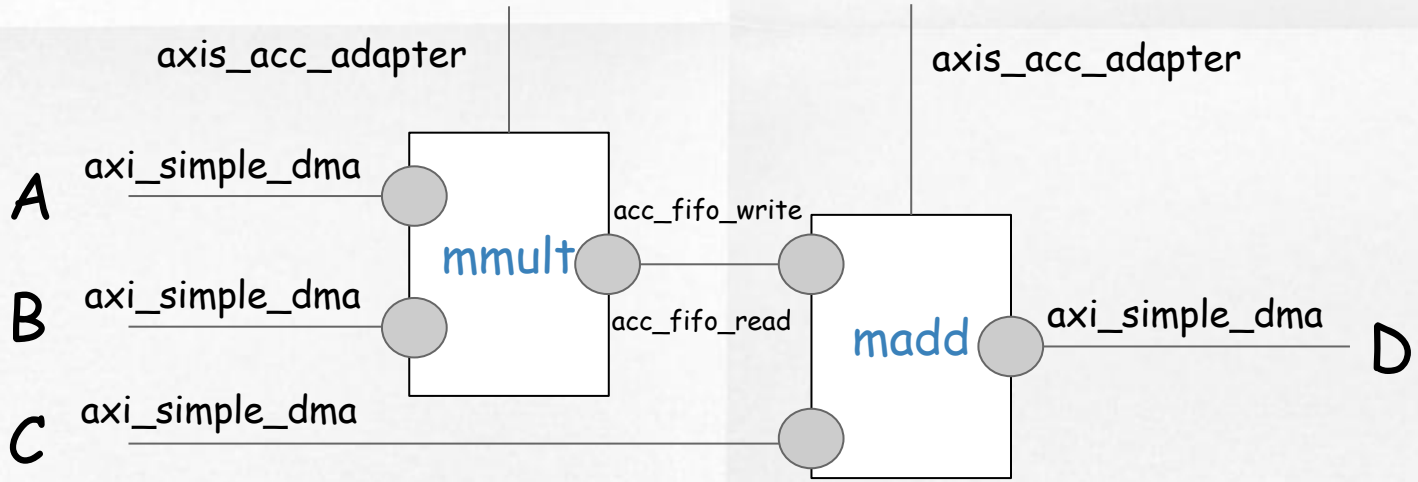


自動生成されたハードウェア構成

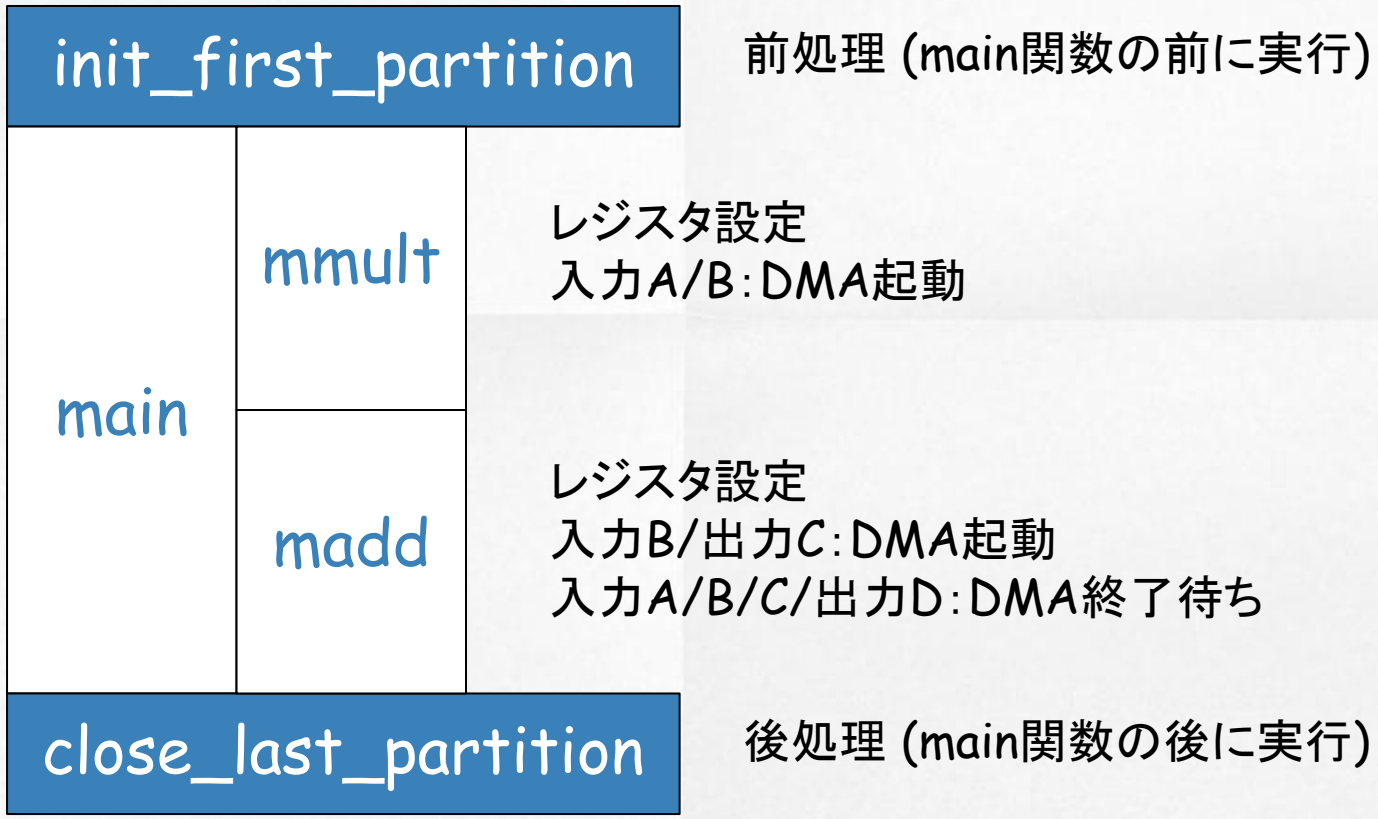


データモーション


PS
A
Data Mover (B)
C
Accelerator


axis_acc_adapter
axis_acc_adapter
A
axi_simple_dma
B
axi_simple_dma
C
axi_simple_dma
mmult
acc_fifo_write
acc_fifo_read
madd
axi_simple_dma
D

プログラム(labn.elf)の流れ



前処理 (main関数の前に実行)

レジスタ設定
入力A/B:DMA起動

レジスタ設定
入力B/出力C:DMA起動
入力A/B/C/出力D:DMA終了待ち

後処理 (main関数の後に実行)

前処理

main関数の前に実行される

init_first_partition (portinfo.c)

```
void init_first_partition() __attribute__ ((constructor));
```

__attribute__ ((constructor));は、
GCCの独自機能でmain関数が呼ばれる前に実行される

```
void init_first_partition()
{
    current_partition_num = 0;
    _ptable[current_partition_num].open(1);

    sds_trace_setup();
}
```

ドライバのオープン (portinfo.c)

```
#define TOTAL_PARTITIONS 1
int current_partition_num = 0;
struct {
  void (*open)(int);
  void (*close)(int);
}

_ptable[TOTAL_PARTITIONS]  = {
    {
        .open = &_p0_cf_framework_open,
        .close= &_p0_cf_framework_close},
    };
```

```
_ptable[partition_num].open(1)
```

_p0_cf_framework_open (portinfo.c)

```
void __attribute__((weak)) pfm_hook_init() {}

// この関数は、2016.2とかなり内容が変わった
void _p0_cf_framework_open(int first)
{
  int xlnk_init_done;
  cf_context_init();
  xlnkCounterMap();                    # xlnk_core_cf.h
```

_p0_cf_framework_open (portinfo.c)

```
  // この部分は、2016.2では、devreg.c内で行われていたが、
  // xxx_register関数への登録が無くなって、portinfo.cに統合された模様

  xlnk_init_done = cf_xlnk_open(first); # xlnk_core_cf.h
  if (!xlnk_init_done) {
    pfm_hook_init();
    cf_xlnk_init(first);                  # xlnk_core_cf.h
  } else if (xlnk_init_done < 0) {
    fprintf(stderr, "ERROR: unable to open xlnk\n");
    exit(-1);
  }
```

_p0_cf_framework_open (portinfo.c)

```
// FPGAのコンテキストを現在のものに設定
cf_get_current_context();

// DMA関連 （4つのDMA）
axi_dma_simple_open(&_p0_dm_0);
axi_dma_simple_open(&_p0_dm_1);
axi_dma_simple_open(&_p0_dm_2);
axi_dma_simple_open(&_p0_dm_3);

// アクセラレータ部のレジスタ関連
axi_lite_open(&_p0_swinst_madd_1_cmd_madd_info);
axi_lite_open(&_p0_swinst_mmult_1_cmd_mmult_info);
```

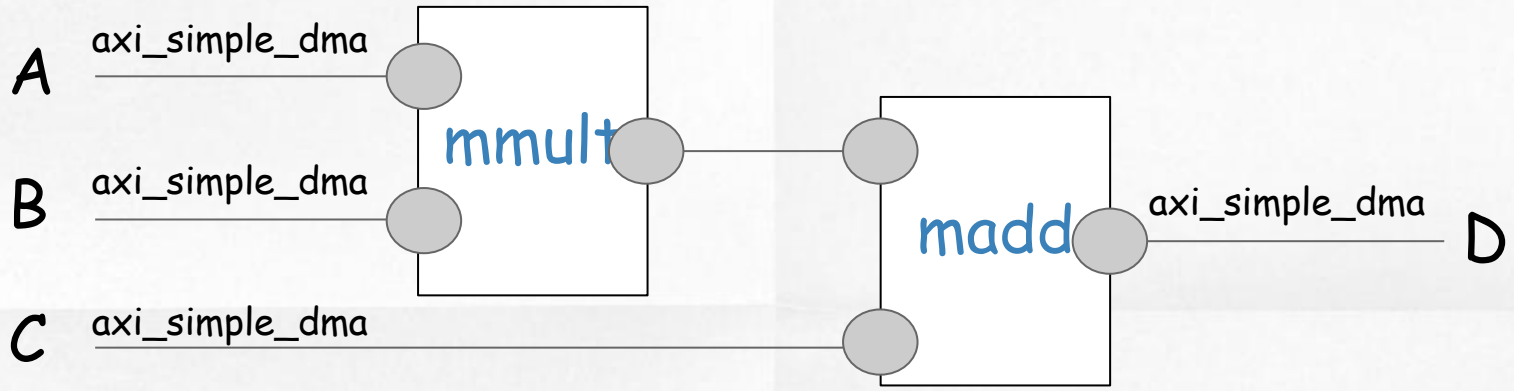
_p0_cf_framework_open (portinfo.c)

```
  // maddアクセラレータへのポート接続
  _sds__p0_madd_1.arg_dm_id[0] =
                        _p0_swinst_madd_1_cmd_madd_info.dm_id;
  _sds__p0_madd_1.arg_dm_id[1] = _p0_dm_0.dm_id;
  _sds__p0_madd_1.arg_dm_id[2] = _p0_dm_3.dm_id;
  _sds__p0_madd_1.arg_dm_id_count = 3;
  accel_open(&_sds__p0_madd_1);

  // mmultアクセラレータへのポート接続
  _sds__p0_mmult_1.arg_dm_id[0] =

_p0_swinst_mmult_1_cmd_mmult_info.dm_id;
  _sds__p0_mmult_1.arg_dm_id[1] = _p0_dm_1.dm_id;
  _sds__p0_mmult_1.arg_dm_id[2] = _p0_dm_2.dm_id;
  _sds__p0_mmult_1.arg_dm_id_count = 3;
  accel_open(&_sds__p0_mmult_1);
}
```

ハードウェア登録



_sds__p0_mmult_0

_p0_dm_0

_p0_dm_1

_p0_dm_2

_sds__p0_madd_0

_p0_dm_3

_p0_dm_Xって？

```
// FPGAのコンテキストを現在のものに設定
 cf_get_current_context();


 // DMA関連 (4つのDMA)
 axi_dma_simple_open(& _p0_dm_0);
 axi_dma_simple_open(& _p0_dm_1);
 axi_dma_simple_open(& _p0_dm_2);
 axi_dma_simple_open(& _p0_dm_3);


 // アクセラレータ部のレジスタ関連
 axi_lite_open(&_p0_swinst_madd_1_cmd_madd_info);
 axi_lite_open(&_p0_swinst_mmult_1_cmd_mmult_info);
```

axi_dma_simple_info_t (portinfo.c)

アトリビュート：device_id, 物理アドレス、サイズ、デバイス ID、
DMA の方向、割り込み、コヒーレントメモリ？、データ幅

```
axi_dma_simple_info_t _p0_dm_0 = {
  .name = "dm_0",
  .phys_base_addr = 0x40400000,
  .addr_range = 0x10000,
  .device_id = 0,
  .dir = XLNK_DMA_TO_DEV,
  .irq = -1,
  .is_coherent = 1,
  .data_width = 64,
};
```

入力DMA

_p0_dm_1と_p0_dm_2も入力DMA

axi_dma_simple_info_t (portinfo.c)

アトリビュート： device_id, 物理アドレス、サイズ、デバイス ID、
DMA の方向、割り込み、コヒーレントメモリ？、データ幅

```
axi_dma_simple_info_t _p0_dm_3 = {
  .name = "dm_3",
  .phys_base_addr = 0x40430000,
  .addr_range = 0x10000,
  .device_id = 3,
  .dir = XLNK_DMA_FROM_DEV,
  .irq = -1,
  .is_coherent = 1,
  .data_width = 64,
};
```

出力DMA

axi_dma_simple_info_t

aarch32-linux/include/axi_dma_simple_dm.h の中で定義している

```
typedef struct axi_dma_simple_info_struct {
        char* name;
        int dm_id;
        int device_id;
        uint64_t phys_base_addr;
        int addr_range;
        void* virt_base_addr;
        int dir;
        int irq;
        int is_coherent;
        int data_width;
        int in_use;
        uint64_t sd_driver_data;
} axi_dma_simple_info_t;
```

_sds__p0_XXX_0って？

```
  // maddアクセラレータへのポート接続
  _sds__p0_madd_1.arg_dm_id[0] =
                          _p0_swinst_madd_1_cmd_madd_info.dm_id;
  _sds__p0_madd_1.arg_dm_id[1] = _p0_dm_0.dm_id;
  _sds__p0_madd_1.arg_dm_id[2] = _p0_dm_3.dm_id;
  _sds__p0_madd_1.arg_dm_id_count = 3;
  accel_open(&_sds__p0_madd_1);

  // mmultアクセラレータへのポート接続
  _sds__p0_mmult_1.arg_dm_id[0] =

_p0_swinst_mmult_1_cmd_mmult_info.dm_id;
  _sds__p0_mmult_1.arg_dm_id[1] = _p0_dm_1.dm_id;
  _sds__p0_mmult_1.arg_dm_id[2] = _p0_dm_2.dm_id;
  _sds__p0_mmult_1.arg_dm_id_count = 3;
  accel_open(&_sds__p0_mmult_1);
}
```

accel_info_t (portinfo.c)

アトリビュート：device_id, 物理アドレス、サイズ、名前、IPタイプ、割り込み

madd

```
accel_info_t _sds__p0_madd_1 = {
  .device_id = 4,
  .phys_base_addr = 0x43c00000,
  .addr_range = 0x10000,
  .func_name = "madd_1",
  .ip_type = "axis_acc_adapter",
  .irq = 0,
};
```

mmult

```
accel_info_t _sds__p0_mmult_1 = {
  .device_id = 5,
  .phys_base_addr = 0x43c10000,
  .addr_range = 0x10000,
  .func_name = "mmult_1",
  .ip_type = "axis_acc_adapter",
  .irq = 0,
};
```

accel_info_t

aarch32-linux/include/accel_info.h の中で定義している

```
struct accel_info_struct {
        int device_id;
        uint64_t phys_base_addr;
        int addr_range;
        char *ip_type;
        void* virt_base_addr;
        int wait_flag;
        int (*is_done)(void* v_addr, unsigned int
done_offset);
        unsigned int done_reg_offset;
        int done_counter;
        int arg_dm_id[256];  int arg_dm_id_count;
        char* func_name;
        int irq;
};
typedef struct accel_info_struct accel_info_t;
```

_p0_swinst_XXXって

```
// FPGAのコンテキストを現在のものに設定
 cf_get_current_context();


 // DMA関連 (4つのDMA)
 axi_dma_simple_open(&_p0_dm_0);
 axi_dma_simple_open(&_p0_dm_1);
 axi_dma_simple_open(&_p0_dm_2);
 axi_dma_simple_open(&_p0_dm_3);


 // アクセラレータ部のレジスタ関連
 axi_lite_open(&_p0_swinst_madd_1_cmd_madd_info) ;
 axi_lite_open(&_p0_swinst_mmult_1_cmd_mmult_info) ;
```

_p0_swblk_mmult (portinfo.c)

```
int _p0_swinst_mmult_1_cmd_mmult_sg_list[] = {0x8};

axi_lite_info_t _p0_swinst_mmult_1_cmd_mmult_info  = {
  .phys_base_addr = 0x43c10000,
  .data_reg_offset = _p0_swinst_mmult_1_cmd_mmult_sg_list,
  .data_reg_sg_size = 1,
  .write_status_reg_offset = 0x0,
  .read_status_reg_offset = 0x0,
  .config = XLNK_AXI_LITE_SG |

XLNK_AXI_LITE_STAT_REG_READ(XLNK_AXI_LITE_STAT_REG_NOCHECK) |

XLNK_AXI_LITE_STAT_REG_WRITE(XLNK_AXI_LITE_STAT_REG_NOCHECK),
  .acc_info = &_sds__p0_mmult_1,
};
```

_p0_swblk_mmult (portinfo.c)

```
struct _p0_swblk_mmult  _p0_swinst_mmult_1  = {
  .cmd_mmult = {
    .base = { .channel_info =
                  &_p0_swinst_mmult_1_cmd_mmult_info
            },
    .send_i = &axi_lite_send
  },
  .A = {
    .base = { .channel_info = &_p0_dm_1},
    .send_i = &axi_dma_simple_send_i
  },
  .B = {
    .base = { .channel_info = &_p0_dm_2},
    .send_i = &axi_dma_simple_send_i
  },
};
```

レジスタ部

入力DMA

入力DMA

_p0_swblk_mmult (portinfo.h)

```
struct _p0_swblk_mmult {

  cf_port_send_t cmd_mmult;

  cf_port_send_t A;

  cf_port_send_t B;

  cf_port_receive_t C;
};
```

レジスタ部

入力DMA

入力DMA

出力FIFO

_p0_swblk_madd (portinfo.c)

```
int _p0_swinst_madd_1_cmd_madd_sg_list[] = {0x8};

axi_lite_info_t _p0_swinst_madd_1_cmd_madd_info = {
  .phys_base_addr = 0x43c00000,
  .data_reg_offset = _p0_swinst_madd_1_cmd_madd_sg_list,
  .data_reg_sg_size = 1,
  .write_status_reg_offset = 0x0,
  .read_status_reg_offset = 0x0,
  .config = XLNK_AXI_LITE_SG |

XLNK_AXI_LITE_STAT_REG_READ(XLNK_AXI_LITE_STAT_REG_NOCHECK) |

XLNK_AXI_LITE_STAT_REG_WRITE(XLNK_AXI_LITE_STAT_REG_NOCHECK),
  .acc_info = &_sds__p0_madd_1,
};
```

_p0_swblk_madd (portinfo.c)

```
struct _p0_swblk_madd  _p0_swinst_madd_1 = {
  .cmd_madd = {
    .base = { .channel_info =
                  &_p0_swinst_madd_1_cmd_madd_info},
    .send_i = &axi_lite_send
  },
  .B = {
    .base = { .channel_info = &_p0_dm_0},
    .send_i = &axi_dma_simple_send_i
  },
  .C = {
    .base = { .channel_info = &_p0_dm_3},
    .receive_ref_i = 0,
    .receive_i = &axi_dma_simple_recv_i
  },
};
```

レジスタ部

入力DMA

出力DMA

_p0_swblk_madd (portinfo.h)

```
struct _p0_swblk_madd {

  cf_port_send_t cmd_madd;

  cf_port_send_t A;

  cf_port_send_t B;

  cf_port_receive_t C;
};
```

レジスタ部

入力FIFO

入力DMA

出力DMA

本処理 (mmult/madd)

main.cpp

```
for (int i = 0; i < NUM_TESTS; i++) {
    init_arrays(A, B, C, D, D_sw);
    float tmp[N*N], tmp1[N*N];
    sw_ctr.start();
    mmult_golden(A, B, tmp);
    madd_golden(tmp, C, D_sw);
    sw_ctr.stop();

    hw_ctr.start();
    _p0_mmult_0(A, B, tmp1);
    _p0_madd_0(tmp1, C, D);
    hw_ctr.stop();
    if (result_check(D, D_sw))
        return 1;
}
```

ソフトウェア処理

ハードウェア処理

mmult.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "cf_stub.h"

void _p0_mmult_1_noasync( float A[1024], float B[1024], float C[1024]);
void _p0_mmult_1_noasync(float A[1024], float B[1024], float C[1024])
{
  switch_to_next_partition(0);
```

必要ならFPGA bitstreamの書き込み処理を行う

後ほど、説明します

mmult.cpp

```
int start_seq[1];
start_seq[0] = 0;

cf_request_handle_t _p0_swinst_mmult_1_cmd;

// 内部レジスタの設定
cf_send_i(&(_p0_swinst_mmult_1.cmd_mmult),
          start_seq, 1 * sizeof(int),
          &_p0_swinst_mmult_1_cmd);

// 内部レジスタの設定終了待ち
cf_wait(_p0_swinst_mmult_1_cmd);
```

mmult.cpp

```
  // 入力AのDMA起動
  cf_send_i(&(_p0_swinst_mmult_1.A), A, 2048,
&_p0_request_2);

  // 入力BのDMA起動
  cf_send_i(&(_p0_swinst_mmult_1.B), B, 2048,
&_p0_request_3);
}
```

madd.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include "cf_stub.h"

void _p0_madd_1_noasync( float A[1024], float B[1024], float C[1024);
void _p0_madd_1_noasync(float A[1024], float B[1024], float C[1024)
{
   switch_to_next_partition(0);
```

必要ならFPGA bitstreamの書き込み処理を行う

後ほど、説明します

madd.cpp

```
int start_seq[1];
start_seq[0] = 0;

cf_request_handle_t _p0_swinst_madd_1_cmd;

// 内部レジスタの設定
cf_send_i(&(_p0_swinst_madd_1.cmd_madd),
          start_seq, 1 * sizeof(int),
          &_p0_swinst_madd_1_cmd);

// 内部レジスタの設定終了待ち
cf_wait(_p0_swinst_madd_1_cmd);
```

madd.cpp

```
  // 入力BのDMA起動
  cf_send_i(&(_p0_swinst_madd_1.B), B, 2048, &_p0_request_0);

  // 出力CのDMA起動
  cf_receive_i(&(_p0_swinst_madd_1.C),
               C, 2048, &_p0_madd_1_noasync_num_C,
               &_p0_request_1);

  // 入力mmult-A/mmult-B/madd-B, 出力madd-CのDMA終了待ち
  cf_wait(_p0_request_0);
  cf_wait(_p0_request_1);
  cf_wait(_p0_request_2);
  cf_wait(_p0_request_3);
}
```

内部レジスタの設定

aarch32-linux/include ディレクトリの
sds_incl.h をインクルードする

```
// 内部レジスタへの書き込み
cf_send_i(&(_p0_swinst_madd_1.cmd_madd),
          start_seq, 1 * sizeof(int),
          &_p0_swinst_madd_1_cmd););
```

cf_send_i は、非同期アクセス （同期は、cf_send関数）

```
// 内部レジスタへの書き込み終了待ち
cf_wait(_p0_swinst_madd_1_cmd));
```

cf_wait は、非同期アクセスの終了待ちを行う

入力DMAの起動

mmulti

```
  // 入力AのDMA起動
  cf_send_i(&(_p0_swinst_mmult_1.A), A, 2048,
&_p0_request_2);

  // 入力BのDMA起動
  cf_send_i(&(_p0_swinst_mmult_1.B), B, 2048,
&_p0_request_3);
```

madd

```
  // 入力BのDMA起動
  cf_send_i(&(_p0_swinst_madd_1.B), B, 2048, &_p0_request_0);
```

cf_send_iは、非同期アクセス （同期は、cf_send関数）

出力DMAの起動

madd

```
// 出力CのDMA起動
cf_receive_i(&(_p0_swinst_madd_1.C),
             C, 2048, &_p0_madd_1_noasync_num_C,
             &_p0_request_1);
```

cf_receive_iは、非同期アクセス （同期は、cf_receive関数）

DMAの終了待ち

madd

```
// 入力mmult-A/mmult-B/madd-B, 出力madd-CのDMA終了待ち
cf_wait(_p0_request_0);
cf_wait(_p0_request_1);
cf_wait(_p0_request_2);
cf_wait(_p0_request_3);
```

cf_waitは、非同期アクセスの終了待ちを行う

DMAのリクエスタの定義

cf_stub.c内で下記のように行っている

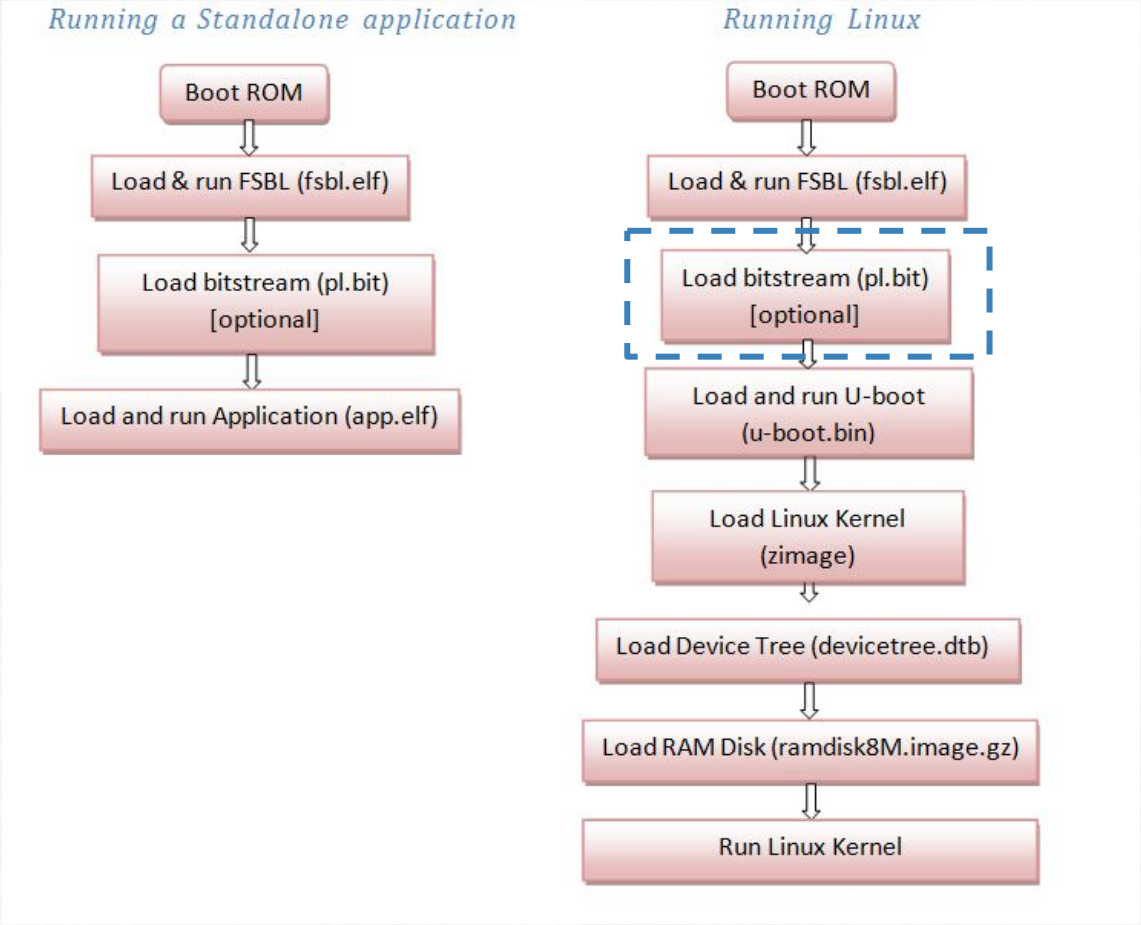
```
#include <sds_incl.h>
#include "cf_stub.h"

cf_request_handle_t _p0_request_0;
cf_request_handle_t _p0_request_1;
cf_request_handle_t _p0_request_2;
cf_request_handle_t _p0_request_3;

Unsigned int _p0_1_noasync_num_C;
```

FPGAの書き替え

ブート時にbitstreamは書き込まれてる



http://xilinx.wikidot.com/zc702-boot-from-flash

FPGA部の書き換え

```
cat /mnt/_sds/_p0_.bin > /dev/xdevcfg
```

XAPP1231 (v1.1) 2015 年 3 月 20 日
Zynq-7000 AP SoC プロセッサにおける Vivado Design Suite を使用したハードウェアアクセラレータのパーシャルリコンフィギュレーション

/dev/xdevcfgドライバに、_sds/_p0_.binを書き込む

実機では、Debug/sd_cardの内容が/mntにマウントされるので、
_sds/_p0_.binは、Debug/sd_card/_sds/_p0_.binになる

FPGA書き換えは、2箇所？

mmult.cpp

```
void _p0_mmult_1_noasync(float A[1024], float B[1024], float C[1024])
{
  switch_to_next_partition(0);
```

madd.cpp

```
void _p0_madd_1_noasync(float A[1024], float B[1024], float C[1024])
{
   switch_to_next_partition(0);
```

switch_to_next_partition (portinfo.c)

```
void switch_to_next_partition(int partition_num)
{
#ifdef __linux__
  if (current_partition_num != partition_num) {
    _ptable[current_partition_num].close(0);
    char buf[128];
    sprintf(buf, "cat /mnt/_sds/_p%d_.bin > /dev/xdevcfg",
                                        partition_num);
    system(buf);                          <= FPGA部の書き換え
    _ptable[partition_num].open(0);       <= ドライバのオープン
    current_partition_num = partition_num;
  }
#endif
}
```

パーティションが違う場合は、
FPGAを書き換えて、ドライバをオープンする

参考) パーティション仕様

SDSoC システム コンパイラ sdscc/sds++ では、ランタイム時にダイナミックに読み込まれた 1 つのアプリケーションに対して複数のビットストリームが自動的に生成される。
各ビットストリームにはそれぞれパーティション識別子を含む

#pragma SDS partition(ID)

```
foo(a, b, c);                    <=  パーティション0
                                     /mnt/_sds/_p0_.bin
#pragma SDS partition (1)
bar(c, d);                       <=  パーティション1
                                     /mnt/_sds/_p1_.bin
#pragma SDS partition (2)
bar(d, e);                       <=  パーティション2
                                     /mnt/_sds/_p2_.bin
```

後処理

main関数の後に実行される

close_last_partition (portinfo.c)

```
void close_last_partition() __attribute__ ((destructor));
```

__attribute__ ((desstructor));は、
GCCの独自機能でmain関数が呼ばれた後に実行される

```
void close_last_partition()
{
#ifdef PERF_EST
    apf_perf_estimation_exit();
#endif
    sds_trace_cleanup();
    _ptable[current_partition_num].close(1);
    current_partition_num = 0;
}
```

ドライバのクローズ (portinfo.c)

```
#define TOTAL_PARTITIONS 1
int current_partition_num = 0;
struct {
  void (*open)(int);
  void (*close)(int);
}

_ptable[TOTAL_PARTITIONS]  = {
    {
        .open = &_p0_cf_framework_open,
        .close= &_p0_cf_framework_close },
    };
```

```
_ptable[partition_num].close(1
);
```

_p0_cf_framework_close (portinfo.c)

```
void __attribute__((weak)) pfm_hook_shutdown() {}

// この関数は、2016.2とかなり内容が変わった
void _p0_cf_framework_close(int last)
{
  accel_close(&_sds__p0_madd_1);
  accel_close(&_sds__p0_mmult_1);
  axi_dma_simple_close(&_p0_dm_0);
  axi_dma_simple_close(&_p0_dm_1);
  axi_dma_simple_close(&_p0_dm_2);
  axi_dma_simple_close(&_p0_dm_3);
  axi_lite_close(&_p0_swinst_madd_1_cmd_madd_info);
  axi_lite_close(&_p0_swinst_mmult_1_cmd_mmult_info);
  pfm_hook_shutdown();
  xlnkClose(last, NULL);          # xlnk_core_cf.h
}
```

DMAを変えてみる

サポートしているDMA

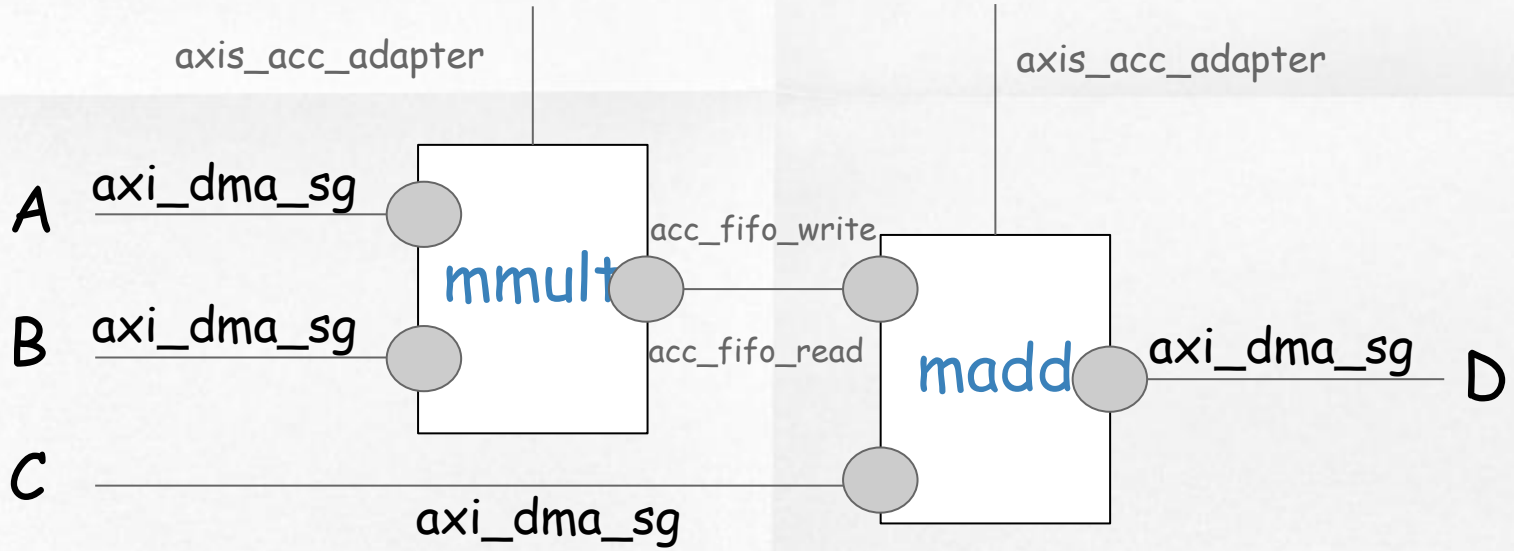
SDSoC(2016.3)でサポートしているDMAは次の2種類

axi_simple_dma　：連続メモリ空間用DMA
sds_alloc関数で取得したメモリに対してDMAを行う

axi_dma_sg　：Scatter/Gather対応DMA
malloc関数で取得したメモリに対してDMAを行う

sds_allocをmallocに変更

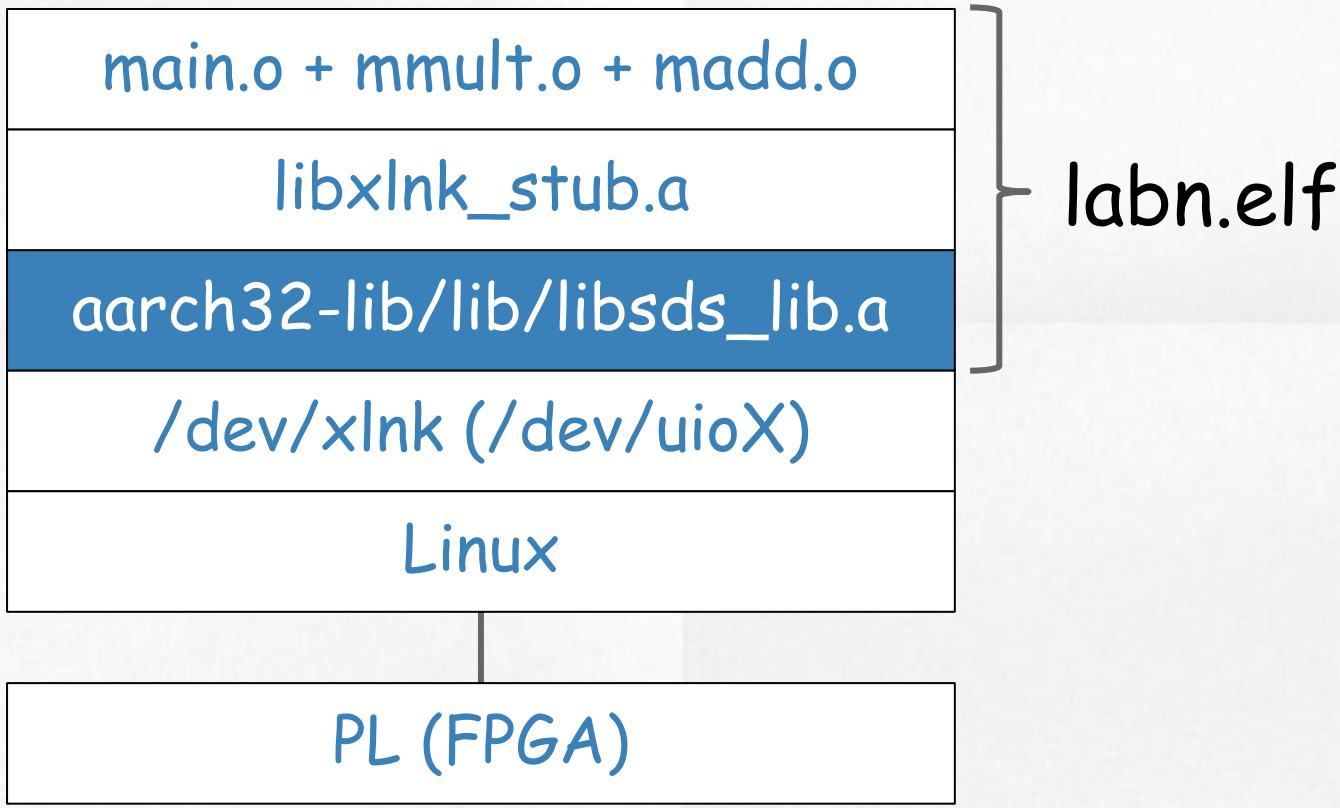
main関数内の sds_alloc/sds_free を malloc/free に変更すると、
DMAが axi_simple_dma から axi_dma_sg に変更される



http://japan.xilinx.com/support/documentation/sw_manuals_j/xilinx2015_2_1/sdsoc_doc/topics/tutorials/task_sdsocgs_lab2_using_malloc.html

libsds_lib.a

例題をSDSoCで実装したときの構成



libsds_lib.aサマリー

aarch32-linux/lib/libsds_lib.a

sds_lib.h
- aarch32-linux/include/sds_lib.h

sds_trace.h
- aarch32-linux/include/sds_trace.h

各種IPインターフェース
- aarch32-linux/include

CF API
- aarch32-linux/include/cf_lib-h,cf-context.h

Accel Library
- aarch32-linux/include/accel_info.h

sds_lib

aarch32-linux/include/sds_lib.h

実体は、aarch32-lib/lib/libsds_lib.a

- sds_wait/sds_try_wait
- sds_alloc/sds_alloc_cacheable
  sds_non_cacheable/sds_free
- sds_map/sds_unmap
- sds_register_dmabuf/sds_unregister_dmabuf
- sds_clock_counter/sds_set_counter
- sds_insert_req
- reset_hw_perf_instr_struct
  get_hw_perf_instr_struct

sds_trace

aarch32-linux/include/sds_trace.h

実体は、aarch32-lib/lib/libsds_lib.a

- trace_list_add
- sds_trace_setup/sds_trace_cleanup
- _sds_print_trace_entry/_sds_print_trace
- _sds_trace_log_HW/_sds_trace_log_SW
- sds_trace/sd_trace_stop

各IPのインターフェース

aarch32-linux/include

実体は、aarch32-lib/lib/libsds_lib.a

- accel_info.h : Accelarator I/F
- axi_dma_2d_dm.h : 2D-Stream DMA
- axi_dma_sg_dm.h : SG DMA
- axi_dma_simple_dm.h : Simple DMA
- axi_fifo_dm.h : FIFO I/F
- axi_lite_dm.h : Register I/F
- zero_copy_dm.h : Zero Copy

cf API

aarch32-linux/include/cf_lib-h,cf-context.h

実体は、aarch32-linux/lib/libsds_lib.a

- cf_get_current_context/cf_context_init
- cf_open_i/cf_open/cf_close_i/cf_close
- cf_send_ref_i/cf_send_ref/cf_send_i/cf_send
- cf_send_2d_i/cf_send_2d
- cf_send_iov_i/cf_send_iov
- cf_receive_ref_i/cf_receive_ref/
  cf_receive_i/cf_receive
- cf_receive_2d_i/cf_receive_2d
- cf_receive_iov_i/cf_receive_iov
- cf_wait/cf_wait_all

Accel Library

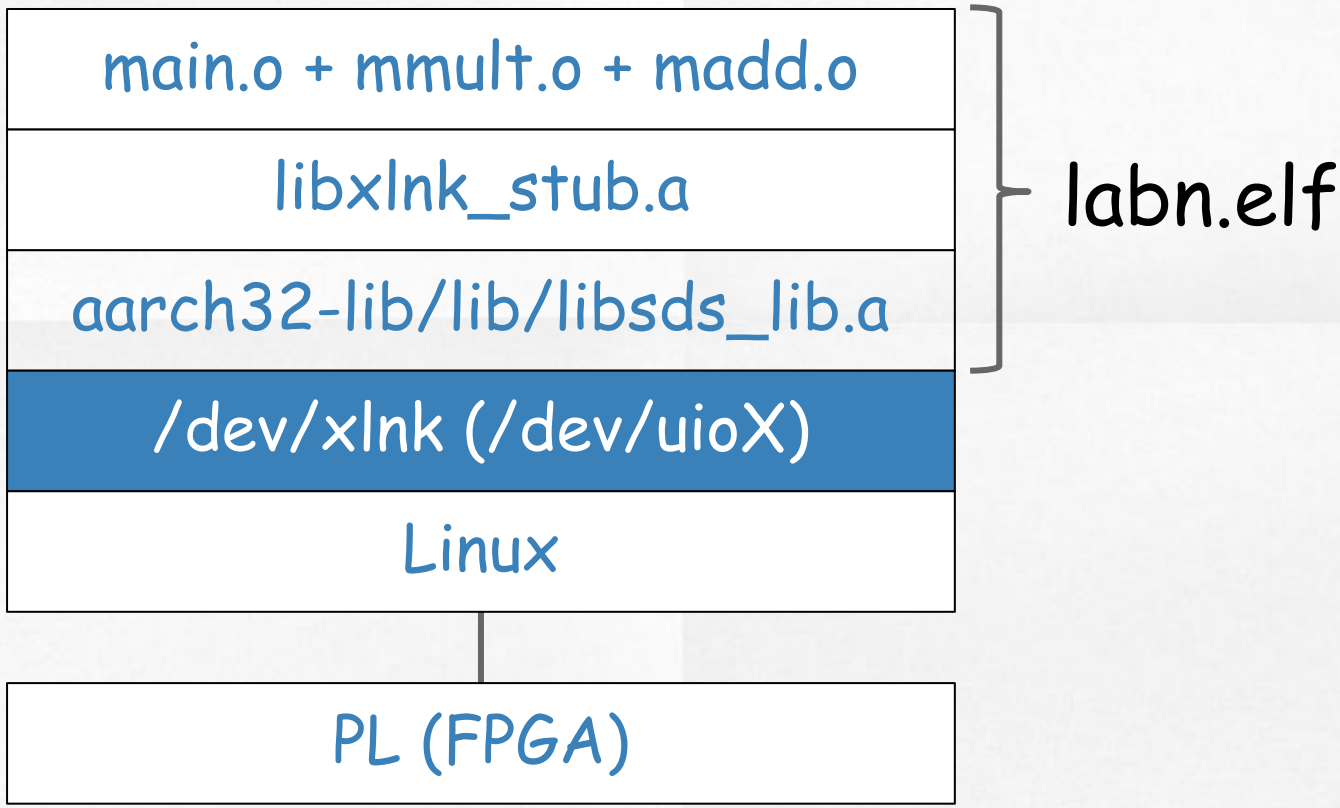
aarch32-linux/include/accel_info.h

実体は、aarch32-linux/lib/libsds_lib.a

- accel_direct_connection_allocate
- accel_open/accel_close
- accel_wait/accel_set_wait_flag
- accel_adapter_has_space
- accel_get_req_info
- accel_get_start_seq
- accel_release_start_req

/dev/xlnk

例題をSDSoCで実装したときの構成



Xilinx APF Accelerator driver

https://github.com/Xilinx/linux-xlnx/drivers/staging/apf

Kconfig
Makefile
xilinx-dma-apf.c/xilinx-dma-apf.h
xlnk-config.c/xlnk-config.h
xlnk-eng.c/xlnk-eng.h
xlnk-ioctl.h
xlnk-sysdef.h
xlnk.c/xlnk.h

devicetree内の
  compatible = "xlnx,xlnk-1.0"

xlnk.c内でuioドライバも使っている

xlnk-ioctl.h

```
#define XLNK_IOCRESET                _IO(XLNK_IOC_MAGIC, 0)
#define XLNK_IOCALLOCBUF             _IOWR(XLNK_IOC_MAGIC, 2, unsigned long)
#define XLNK_IOCFREEBUF              _IOWR(XLNK_IOC_MAGIC, 3, unsigned long)
#define XLNK_IOCADDDMABUF            _IOWR(XLNK_IOC_MAGIC, 4, unsigned long)
#define XLNK_IOCCLEARDMABUF          _IOWR(XLNK_IOC_MAGIC, 5, unsigned long)
#define XLNK_IOCDMAREQUEST           _IOWR(XLNK_IOC_MAGIC, 7, unsigned long)
#define XLNK_IOCDMASUBMIT            _IOWR(XLNK_IOC_MAGIC, 8, unsigned long)
#define XLNK_IOCDMAWAIT              _IOWR(XLNK_IOC_MAGIC, 9, unsigned long)
#define XLNK_IOCDMARELEASE           _IOWR(XLNK_IOC_MAGIC, 10, unsigned long)
#define XLNK_IOCDEVREGISTER          _IOWR(XLNK_IOC_MAGIC, 16, unsigned long)
#define XLNK_IOCDMAREGISTER          _IOWR(XLNK_IOC_MAGIC, 17, unsigned long)
#define XLNK_IOCDEVUNREGISTER        _IOWR(XLNK_IOC_MAGIC, 18, unsigned long)
#define XLNK_IOCCDMAREQUEST          _IOWR(XLNK_IOC_MAGIC, 19, unsigned long)
#define XLNK_IOCCDMASUBMIT           _IOWR(XLNK_IOC_MAGIC, 20, unsigned long)
#define XLNK_IOCMCDMAREGISTER        _IOWR(XLNK_IOC_MAGIC, 23, unsigned long)
#define XLNK_IOCCACHECTRL            _IOWR(XLNK_IOC_MAGIC, 24, unsigned long)
#define XLNK_IOCSHUTDOWN             _IOWR(XLNK_IOC_MAGIC, 100, unsigned long)
#define XLNK_IOCRECRES               _IOWR(XLNK_IOC_MAGIC, 101, unsigned long)
#define XLNK_IOCCONFIG               _IOWR(XLNK_IOC_MAGIC, 30, unsigned long)
```

分類してみると

ALLOCBUF/ FREEBUF

ADDDMABUF/CLEARDMABUF

DMAREQUEST/DMASUBMIT/DMAWAIT/DMARELEASE

DMAREGISTER/MCDMAREGISTER
DEVREGISTER/DEVUNREGISTER

CACHECTRL/SHUTDOWN/RECRES/CONFIG

RESET/CDMAREQUEST/CDMASUBMIT（xlnk.cで未使用）

登録出来るデバイス数

xlnk.cに、

```
#define MAX_XLNK_DMAS 16
```

とあり、デバイスの登録関数 (xlnk_devpacks_add)では、

```
static void xlnk_devpacks_add(struct xlnk_device_pack
*devpack)
{
        unsigned int i;
        for (i = 0; i < MAX_XLNK_DMAS; i++) {
                if (xlnk_devpacks[i] == NULL) {
                        xlnk_devpacks[i] = devpack;
                        break;
                }
        }
}
```

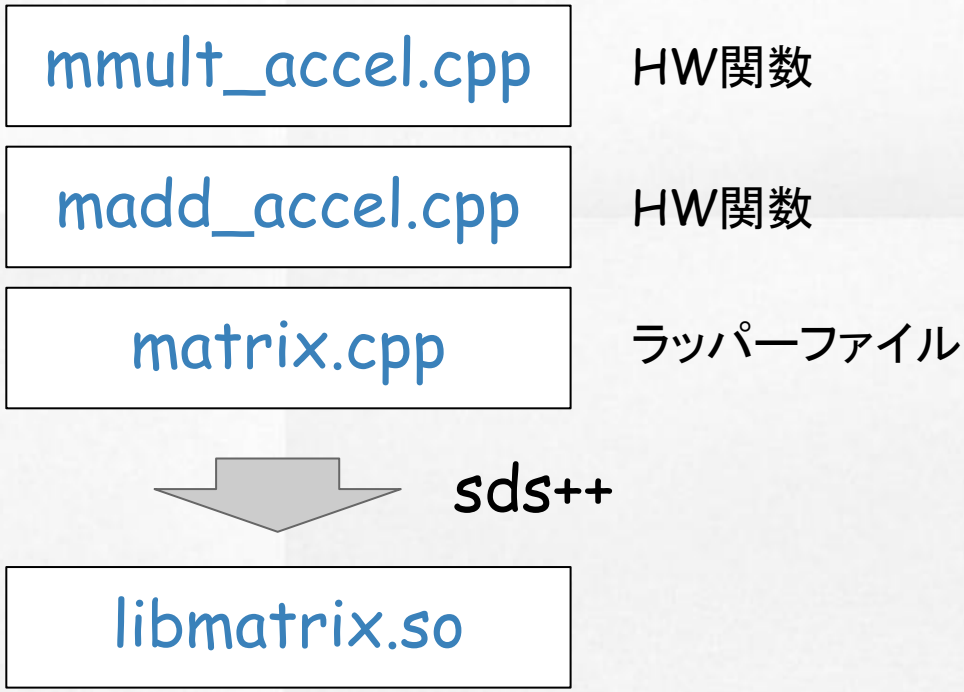
になっているので、

16個

共有ライブラリ化

共有ライブラリによる再利用

ハードウェア関数を共有ライブラリすることで再利用できる



SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.109-113

matrix.cpp

```
#include "madd_accel.h"
#include "mmult_accel.h"
void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE],
          float out_C[MSIZE*MSIZE])
{
    madd_accel(in_A, in_B, out_C);
}
void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE],
           float out_C[MSIZE*MSIZE])
{
    mmult_accel(in_A, in_B, out_C);
}
void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE],
              float in_C[MSIZE*MSIZE], float out_D[MSIZE*MSIZE])
{
    float tmp[MSIZE * MSIZE];
    mmult_accel(in_A, in_B, tmp);
    madd_accel(tmp, in_C, out_D);
}
```

SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.109-113

matrix.h

```
#ifndef MATRIX_H_
#define MATRIX_H_
#define MSIZE 16

void madd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE],
          float out_C[MSIZE*MSIZE]);

void mmult(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE],
           float out_C[MSIZE*MSIZE]);

void mmultadd(float in_A[MSIZE*MSIZE], float in_B[MSIZE*MSIZE],
              float in_C[MSIZE*MSIZE], float out_D[MSIZE*MSIZE]);

#endif /* MATRIX_H_ */
```

SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.109-113

makefile

```
#  sds++への引数がポイント
SDSFLAGS = \
-sds-pf ${PLATFORM} \
-sds-hw mmult_accel mmult_accel.cpp -sds-end \
-sds-hw madd_accel madd_accel.cpp -sds-end

#  基本的には、GCCでコンパイル＆リンクしているのと同じ

#  ここで、各ファイルをコンパイルしてオブジェクト化（-fpicが必要）
sds++ ${SDSFLAGS} -c -fpic -o mmult_accel.o mmult_accel.cpp
sds++ ${SDSFLAGS} -c -fpic -o madd_accel.o madd_accel.cpp
sds++ ${SDSFLAGS} -c -fpic -o matrix.o matrix.cpp

#  ここで、オブジェクトファイルから共有ライブラリ化（-sharedが必要）
sds++ ${SDSFLAGS} -shared -o libmatrix.so \
                  mmult_accel.o madd_accel.o matrix.o
```

SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.109-113

複数の共有ライブラリを使うとき

SDSoCの下記のpragmaにて、生成されるFPGA bitstreamのパーティション番号を変えることで複数の共有ライブラリを使うことができる。
指定しない時は、すべて0パーティションとなるので、複数の共有ライブラリは使えないので注意！

```
# プラグマが無い場合（デフォルト）
foo(a, b, c);                    <=  パーティション0
#pragma SDS partition (1)
bar(c, d);                       <=  パーティション1
#pragma SDS partition (2)
bar(d, e);                       <=  パーティション2
```

SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.91-92

ライブラリの配布

<path_to_library>/include/matrix.h	ヘッダファイル
<path_to_library>/lib/libmatrix.so	共有ライブラリ
<path_to_library>/sd_card	FPGAのbitstream

SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.109-113

ライブラリを使うには

```
/* main.cpp (pseudocode) */
#include "matrix.h"
int main(int argc, char* argv[])
{
    float *A, *B, *C, *D;
    float *J, *K, *L;
    float *X, *Y, *Z;
    ...
    mmultadd(A, B, C, D);
    ...
    mmult(J, K, L);
    ...
    madd(X, Y, Z);
    …
}
```

```
gcc -I <path_to_library>/include -o main.o main.c

gcc -I <path_to_library>/lib -o main.elf main.o
-lmatrix
```

SDSoC Enviironment User Guilde UG1027 (v2016.3) November 30, 2016, Page.109-113

本資料では、
SDSoC™ 2016.3が生成する
ソフトウェアについて
調べたのをまとめました。

ご利用は、自己責任でお願いします。

おしまい