Using C-Callable libraries and creating multiple accelerators

SDx 2018.2

XILINX

Objectives

- **After completing this module, you will be able to:**
  - List components of C-Callable functions
  - Describe the default behavior of the SDSoC development environment for handling multiple accelerators
  - Generate multiple accelerators for the same function
  - Use a pragma to generate multiple instances of an accelerator to override tool defaults
  - Explain the difference between blocking and non-blocking architectures as it relates to the SDSoC development environment

Outline

> C-Callable Accelerators
> Multiple Accelerators
> Blocking vs Non-Blocking calls
> Summary

SDSoC System Compilation and Linking

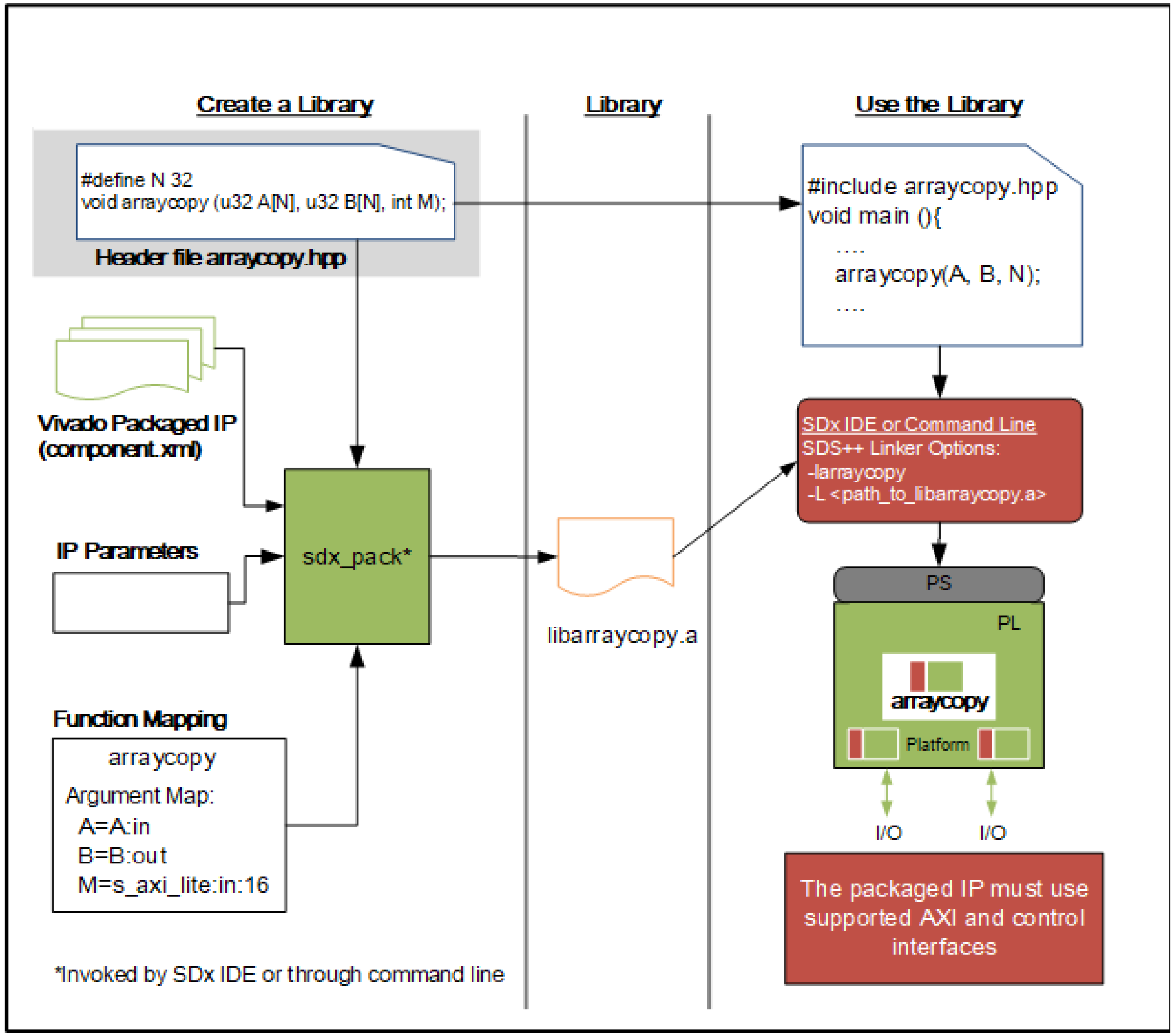
- SDSoC tool compiler
  - sdscc/sds++ compiler built upon Clang/LLVM frameworks, generates LLVRM (.s) files for each file, creating HLS interface pragmas for accelerators
  - Creates a local HLS project for each accelerator and calls HLS
    - HLS returns an IP packaged by the Vivado Design Suite for each accelerator
    - HLS returns an interface XML file (database)
  - Encapsulates everything needed for system linking into a standard .o file
  - C-callable IP must have all of the above
    - User responsibility to provide various components
  - SDSoC system linker: Takes .o files, analyzes entire program to generate accelerator network and software stubs

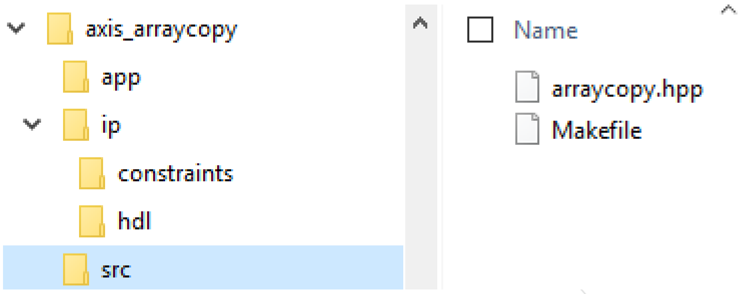
What is C-Callable Library?

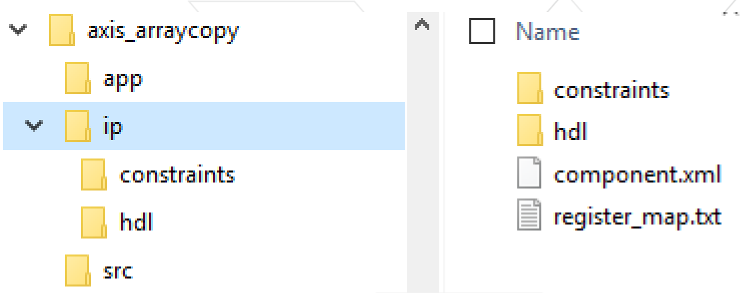
- C-Callable library consists of IP blocks written in VHDL or Verilog which can be called from user applications developed in SDSoC by statically linking it
  - The IP blocks will be instantiated into the generated hardware system
- C-Callable library function consists of two components
  - Header File
    - Function prototype
  - Static Library
    - Function definition
    - IP core
    - IP configuration parameters
    - Function argument mapping

XILINX

C-Callable Library Usage Flow







C-Callable Function Components (1)

> **Header file**
  - A library must declare function prototypes that map onto the IP block in a header file that can be #included in user application source files
  - Example:

```
// FILE: arraycopy.hpp

#ifndef _ARRAYCOPY_H_
#define _ARRAYCOPY_H_
#define N 32
typedef unsigned int u32;
#pragma SDS data copy(A[0:M])
#pragma SDS data copy(B[0:M])
void arraycopy(u32 A[N], u32 B[N], int M);
#endif
```

C-Callable Function Components (2)

> **IP Core**
  - An HDL IP core for a C-callable library must be packaged using the Vivado tools
    - The packager creates a directory structure for the HDL and other source files, and an IP Definition file
    - The IP control register must exist at address offset 0x0
    - The `ap_start` signal initiates the IP execution, `ap_done` indicates IP task completion, and `ap_ready` indicates that the IP can be started
  - Place the IP core either in the Vivado tools IP repository or in any other location
  - Example

```
// 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)
// (COR = Clear on Read, COH = Clear on Handshake)
```

C-Callable Function Components (3)

- **IP Configuration Parameters**
  - Most HDL IP cores are customizable at synthesis time
  - Customization done through IP parameters
  - SDSoC uses this information during the core instantiation in a generated system
  - The information is captured in an XML file
    - Example:


```xml
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component xmlns:xilinx="http://www.xilinx.com" xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1685-2009" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <spirit:vendor>xilinx.com</spirit:vendor>
  <spirit:library>rtl</spirit:library>
  <spirit:name>arraycopy</spirit:name>
  <spirit:version>1.0</spirit:version>
  <spirit:busInterfaces>
    <spirit:busInterface>
      <spirit:name>s_axi_lite</spirit:name>
      <spirit:busType spirit:vendor="xilinx.com" spirit:library="interface" spirit:name="aximm" spirit:version="1.0"/>
      <spirit:abstractionType spirit:vendor="xilinx.com" spirit:library="interface" spirit:name="aximm_rtl" spirit:version="1.0"/>
      <spirit:slave>
        <spirit:memoryMapRef spirit:memoryMapRef="s_axi_lite"/>
      </spirit:slave>
      <spirit:portMaps>
```

C-Callable Function Components (6)

- A utility called `sdx_pack` allows the creation of SDSoC libraries

  `sdx_pack [arguments] [options]`

  - Example:

```
sdx_pack -header arraycopy.hpp -lib libarraycopy.a \
-func arraycopy -map A=A:in -map B=B:out -map M=s_axi_lite:in:16 -func-end \
-ip ../ip/component.xml -control AXI=s_axi_lite:0 \
-target-family zynq -target-cpu cortex-a9 -target-os standalone \
-verbose
```

How to Use C-Callable Library

- Using a C-callable library is similar to using any software library
  - Use `#include` header files for the library in appropriate source files
  - Use the `sdscc -I<path>` option to compile your source from command line
  - Use `C/C++ Build Settings->SDSCC Compiler->Directories` (or `SDS++ Compiler->Directories for C++ compilation`) in GUI
  - Link the library using the -L<path> and -l<lib> options from the command line
  - Use `C/C++ Build Settings > SDS++ Linker > Libraries` in GUI
    - Use `lib<library_name>.a` as the library name and `<library_name>` with –l switch
    - For example, `libMylib.a` as the library name and `–lMylib` as the switch

Multiple Accelerators

XILINX

Default Behaviors for Accelerator Replication

- **Multiple calls to a hardware function are usually mapped to a single accelerator**
- **SDSoC tool will automatically generate multiple instances of an accelerator**
- **Multiple instances will create parallelism and can improve system performance**
- **If the output of an accelerator is an input of an another accelerator, SDSoC tool will *pipeline* the accelerators**
  - Avoiding unnecessary access to main memory
  - The compiler creates two instances of the hardware function as shown below

```
mmult_accel(tin1Buf, tin2Buf, toutBufHwInter);
mmult_accel(toutBufHwInter, tin2Buf, toutBufHw);
```

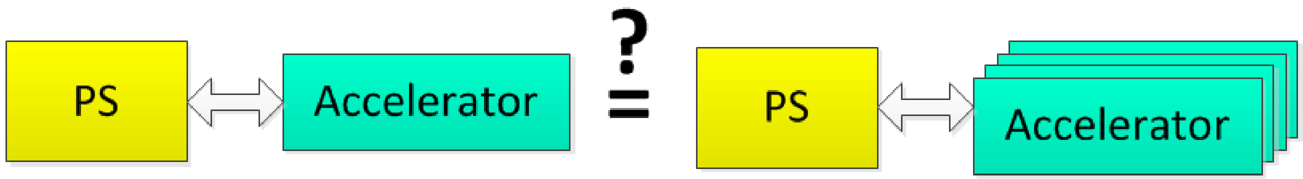
Forcing Multiple Instances of an Accelerator

- **`#pragma SDS resource(id)` before function call**
  - Direct the compiler to create multiple instances of a hardware by inserting this pragma immediately preceding a call to the function
  - Directs SDSoC tool to return control after setting up all data transfers
  - Different ID for the same accelerator function results in different accelerator hardware instance
- **`#pragma SDS wait(id)` must be used at suitable synchronization points**
- **SDSoC tool may create multiple accelerators to improve performance**
  - Sometimes helpful for PL-to-PL connections

```
#pragma SDS resource(1)
mmult_accel(...); // instance 1
#pragma SDS resource(2)
mmult_accel(...); // instance 2
...
#pragma SDS wait(1)
#pragma SDS wait(2)
```

No Estimate Support for Asynchronous Calls

- **Generating multiple instances of an accelerator affects the macro-architecture**
  - Is very difficult to provide accurate estimations with the current SDEstimate flow
    - Therefore, for the current version, there is no support for the performance estimation flow



XILINX

Task-Level Pragmas in the SDSoC Development Environment

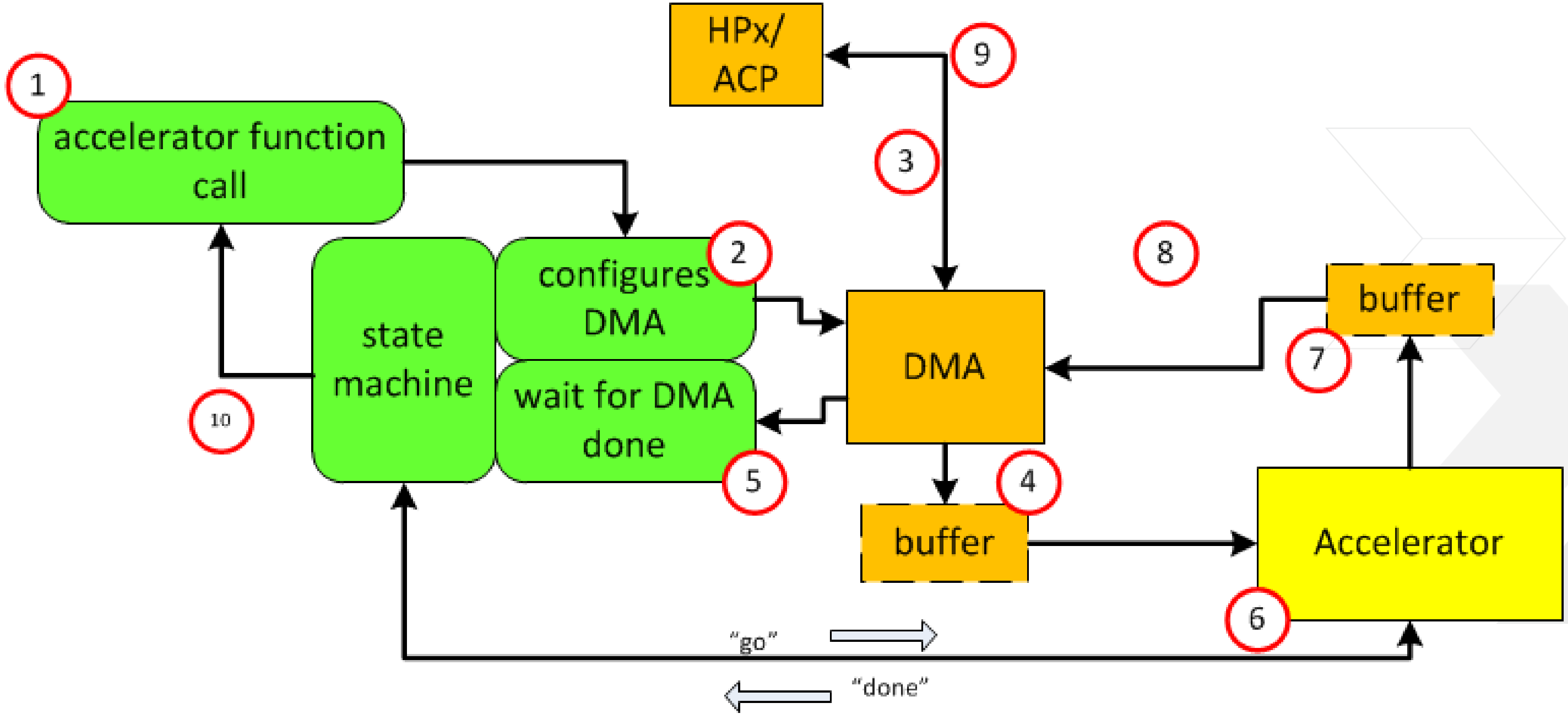
- **Some additional SDS pragmas are necessary to implement task-level pipelining**
  - `#pragma SDS async(ID)`
    - Tells the compiler that the following accelerator call should be executed asynchronously
    - ID specifies a unique ID for each use of async() with a specific accelerator
    - Separate accelerator instance is generated for each unique ID
  - `#pragma SDS wait(ID)` or `function sds_wait(ID)`
    - Allows a program to wait for completion of an asynchronously called accelerator
    - Task completion is always triggered in the same order in which they were called
    - Run-time queue maintains wait-handles associated with the arguments of each task
  - `#pragma SDS buffer_depth(Arg1:BufferDepth, Arg2:BufferDepth, ...)`
    - Creates the multi-buffers required to pipeline
    - Depth must be compatible with pipeline length

Blocking vs Non-Blocking Calls

XILINX

An Accelerator Task

> The *task* of calling an accelerator involves many steps in the background



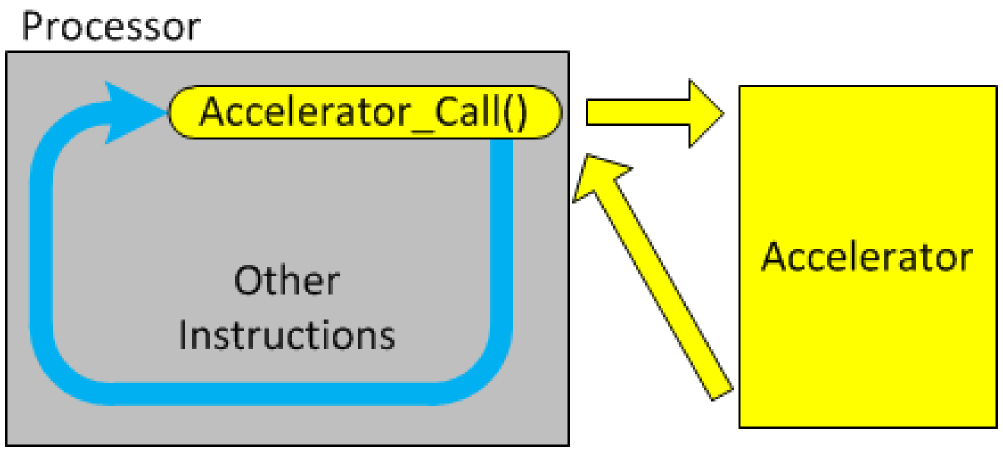
Blocking Behavior

> **Blocking behavior causes the processor to stall during accelerator task**
  - Sending input arguments
  - Processing of arguments in the accelerator
  - Returning results

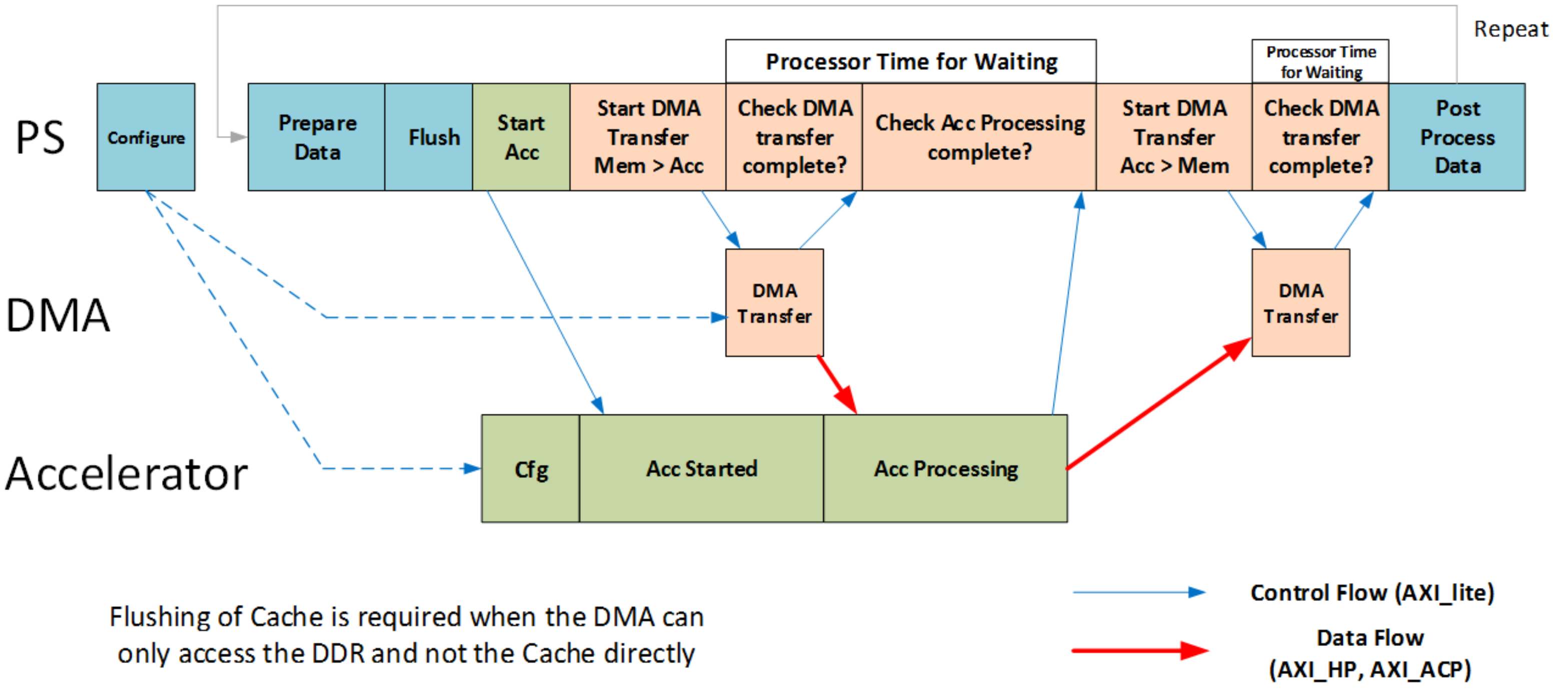
> **Benefit is reduced coding complexity (no synchronization issues)**

> **Drawback is that processor sits idle (waste processing power)**

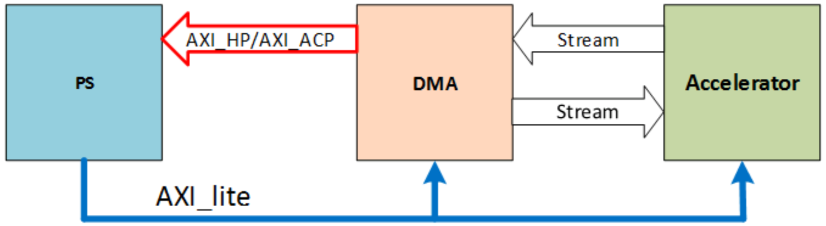
> **Referred to as *sequential* in the SDSoC development environment**



Accelerator I/O Connected to PS: Blocking

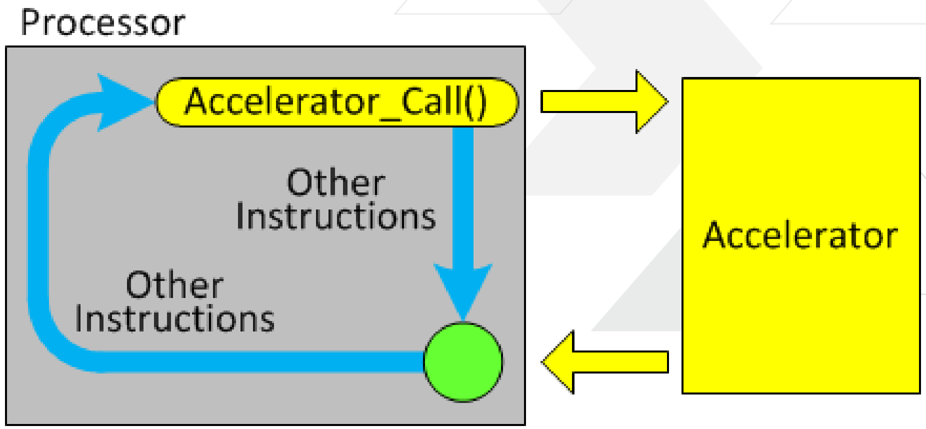


Flushing of Cache is required when the DMA can only access the DDR and not the Cache directly



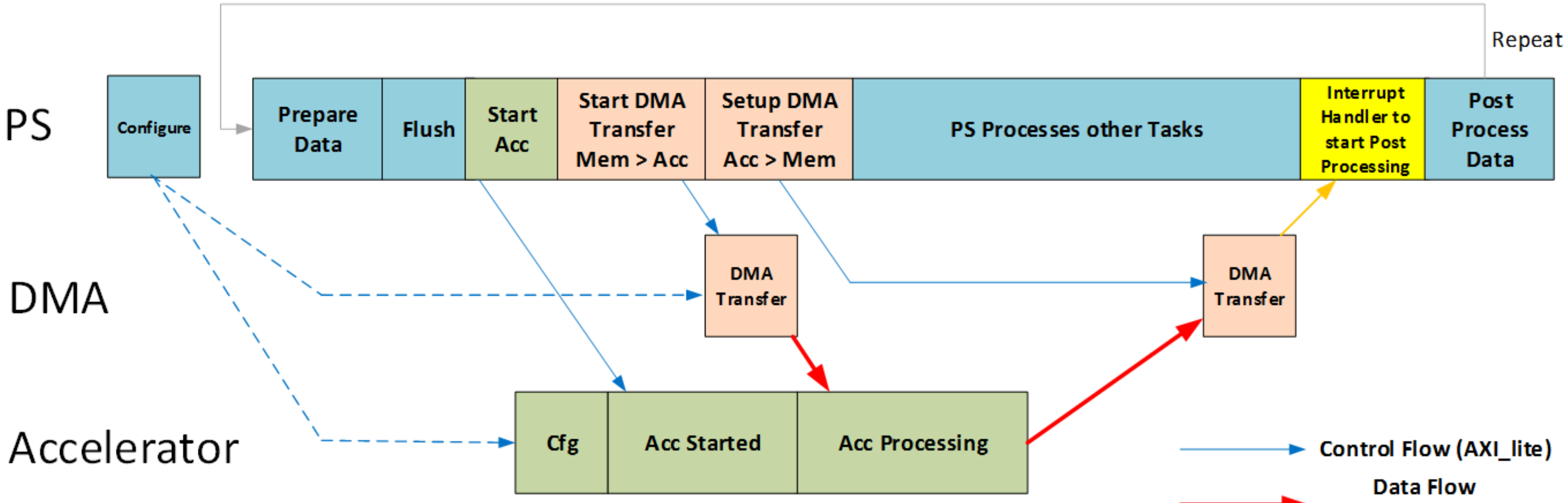
Non-Blocking Behavior

- True non-blocking (or just non-blocking) behavior allows the processor to continue operation after the accelerator call is made
- Benefit is that processor can continue operation simultaneously with accelerator
- Drawback is additional complexity (synchronization issues)
- SDSoC refers to this as *asynchronous* calling
  - Only applies to accelerator calls

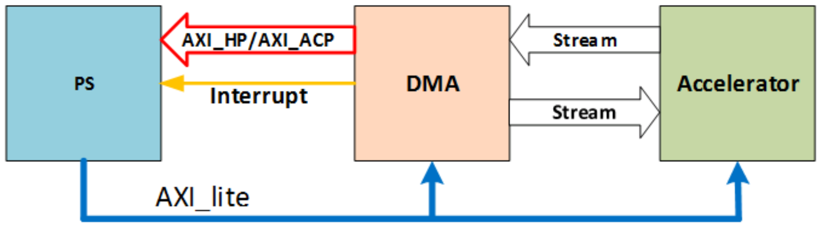


XILINX

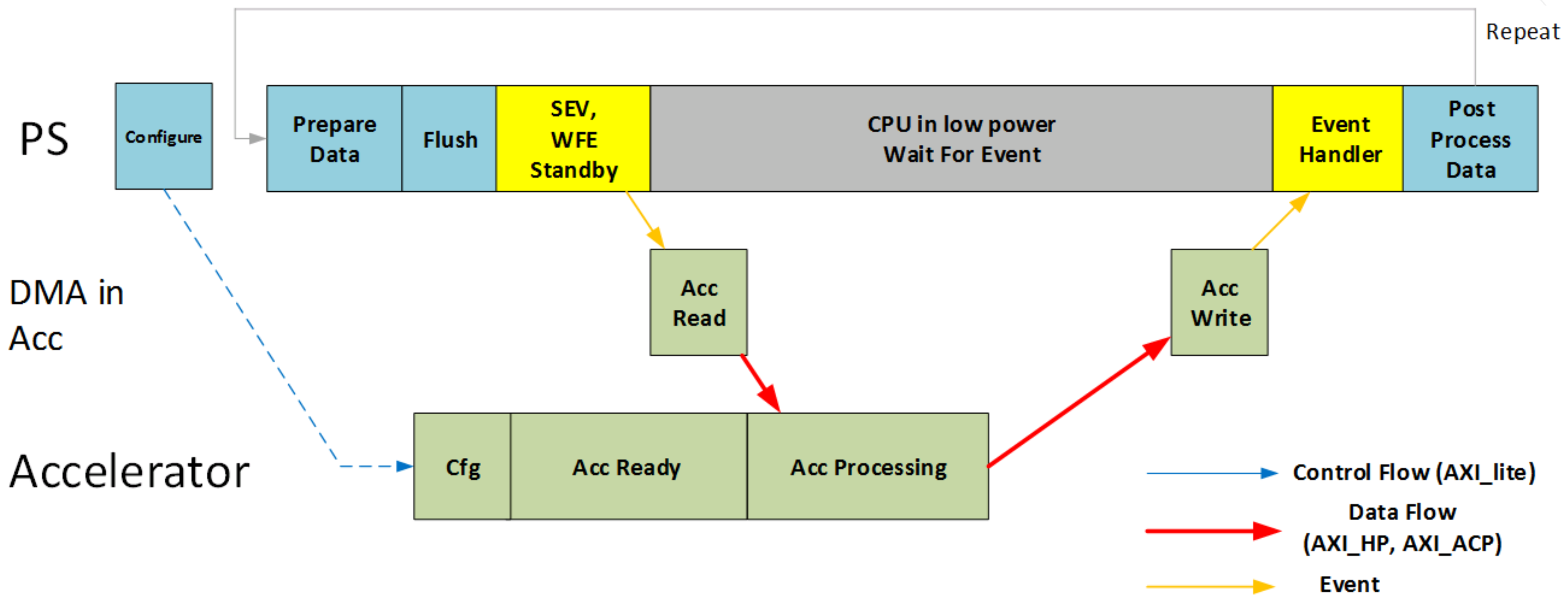
Accelerator I/O Connected to PS: Non-Blocking



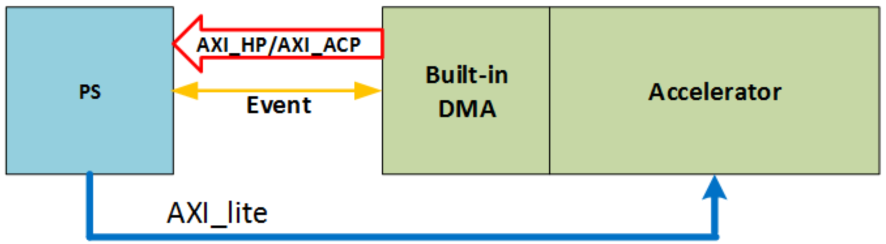
Flushing of Cache is required when the DMA can only access the DDR and not the Cache directly



Accelerator I/O Connected to PS: Blocking for Power



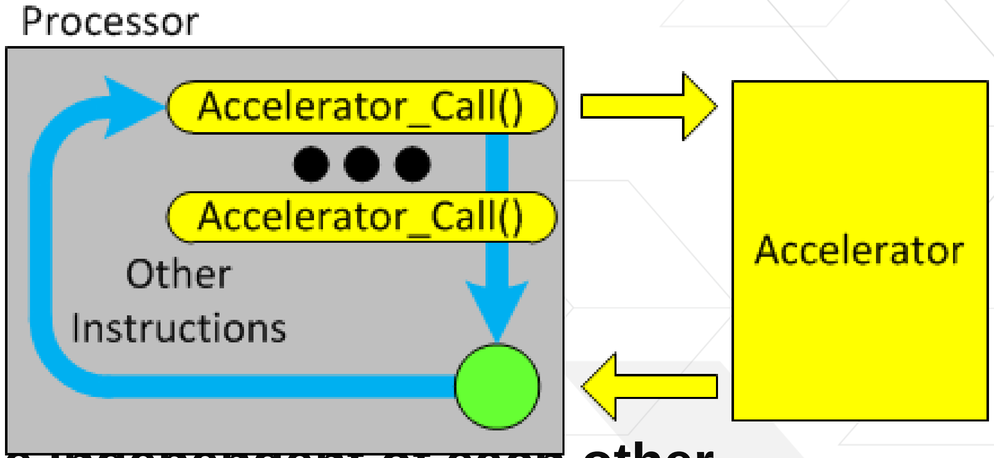
Flushing of Cache is required when the DMA can only access the DDR and not the Cache directly



XILINX.

Synchronization and Pipelining Features

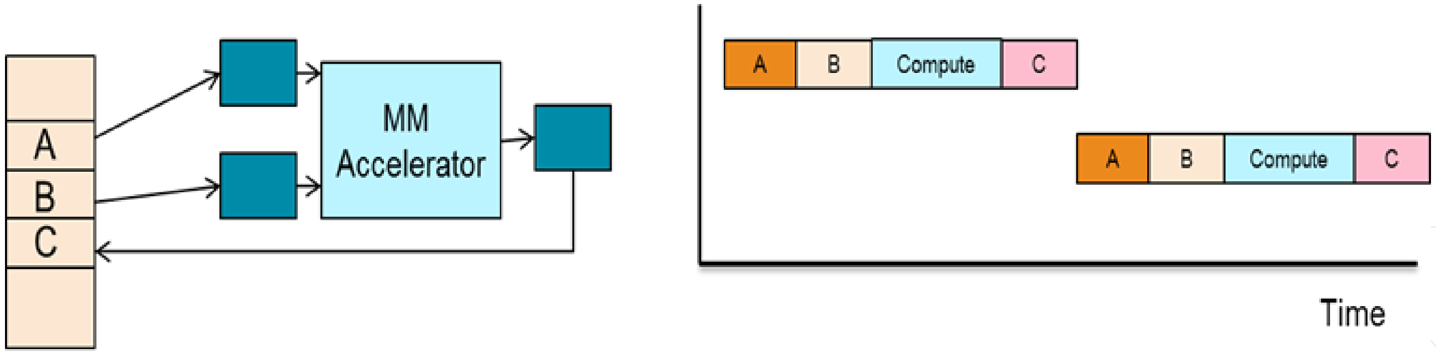
- Consider code that makes several consecutive calls to the same accelerator function
  - If each call is done asynchronously then other code can be run while the '*task*' of calling an accelerator is run
  - If the *task* can be broken into independent *stages* there is opportunity for parallelism
  - Capitalize on parallelism by restructuring code to create a *task pipeline*



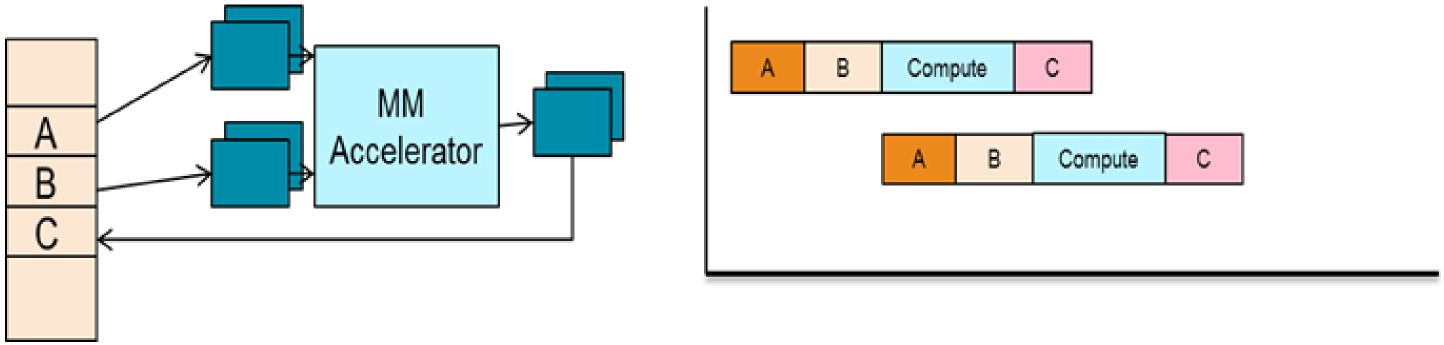
- The following ***stages*** of an accelerator task can be independent of each other
  - Sending input data from shared memory to local accelerator buffers
  - Processing data
  - Returning results from local accelerator buffers to shared memory

Pipelining Multiple Calls to the Accelerator

> **Sequential execution of matrix multiply calls (blocking)**



> **Asynchronous pipelined calls *require* use of multi-buffers**



~7.5X speedup over software version

Synchronization of Software/Hardware Communication

- **Synchronization is important so that the hardware and software communicate effectively**
  - Processor has to know when the accelerator is done
- **Software determines if a hardware accelerator has completed processing data using one of two methods**
  - Interrupt driven (default) or polling
  - The desired method can be selected using an SDSoC tool-specific compiler flag
    - --poll-mode <*0*|*1*> where 1 = poll mode enabled
  - Contrary to common belief, polling can often yield better results for Linux-based systems
    - Due to overhead associated with handling interrupts in Linux

Summary

Summary

> C-Callable library consists of IP blocks written in VHDL or Verilog which can be called from user applications developed in SDSoC by statically linking it

> C-Callable library function consists of two components

>> Header file

>> Static library

> The SDSoC development environment offers a mechanism for generating multiple instances of an accelerator

> `#pragma SDS async(id)` before the function call directs the SDSoC tool to return control after setting up all data transfers

> `#pragma SDS wait(id)` must be used at suitable synchronization points

Summary

> The SDSoC development environment determines the accelerator-to-accelerator connections by analyzing the code that calls accelerators

> Asynchronous/non-blocking accelerator calls allow accelerator tasks to be pipelined

> A non-blocking accelerator call is just one of several mechanisms to control synchronization and/or pipelining in the SDSoC development environment

Adaptable.
Intelligent.

XILINX®