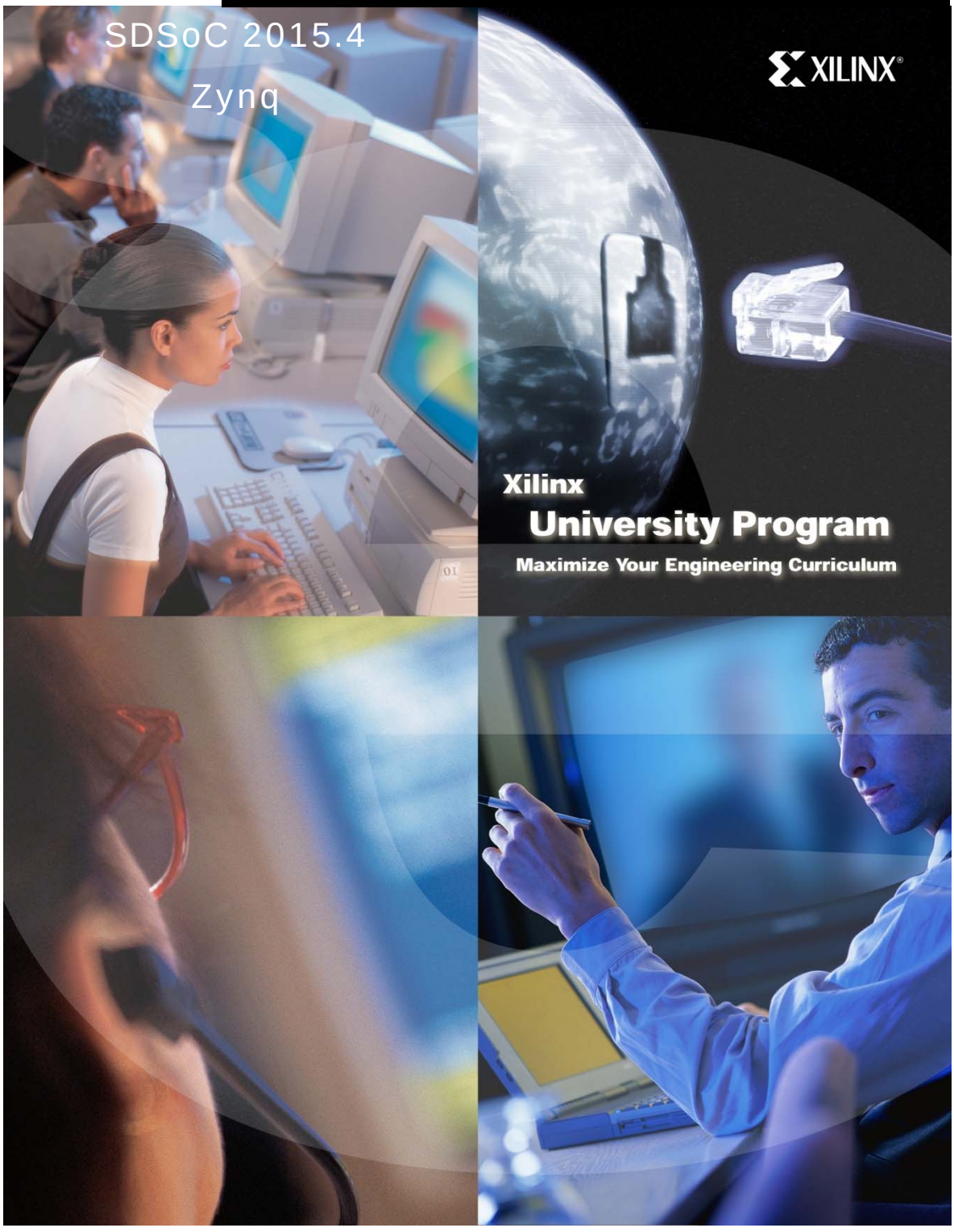
XILINX
UNIVERSITY PROGRAM

Designing Xilinx Zynq-Based Systems using SDSoC

Presentation Manual

SDSoC 2015.4

Zynq



Designing Xilinx Zynq-Based Systems using SDSoC

Zynq
SDSoC 2015.4 Version

Course Objectives

- **After completing this course, you will be able to:**
  - Introduce the concept of “software-defined” systems on chip (SDSoC)
  - Understand the capabilities and limitations of the SDSoC development environment
  - Get hands-on experience of creating application-specific systems on chip from C/C++ programs using the SDSoC
  - Gain practical understanding of the SDSoC design flow
    - How the SDSoC compiler maps programs to HW/SW systems
      - Structure of generated hardware systems
      - Structure of the generated software
    - How to control the compilation and generation process
      - Modifying program source using #pragmas
  - Identify the architectural aspects of an SoC that facilitate hardware acceleration

Course Objectives (2)

- Identify candidate functions for hardware acceleration
- Move designated software functions to hardware and estimate the performance of the accelerator and the effect on the entire system
- Use the System Debugger's capabilities to control the execution flow and examine memory and variables during a debug session
- Create a hardware platform for an custom application

Course Outline
Day 1

The course consists of the following modules:

- **Zynq AP SoC architecture and Vivado IPI**
- **SDSoC tool overview**
- **Lab 1: Getting started with SDSoC design flow**
- **Data motion networks**
- **Lab 2: Pragma and data motion networks**
- **Coding Considerations**
- **Profiling**
- **Lab 3: Profiling application and create accelerators**

Course Outline
Day 2

- Estimation
- Lab 4: Estimating accelerator performance
- Debugging
- Lab 5: Debugging software application
- Using C-callable libraries and creating multiple accelerators
- Improving performance with Vivado HLS
- Lab 6: Fine-tuning with Vivado HLS
- Creating SDSoC platform
- Lab 7: Creating and using platform for an custom application

Prerequisites

- Basic C programming
- Basic understanding of processor-based system

Platform Support

- **SDSoC Suite 2015.4**
- **Xilinx University board**
  - ZedBoard or Zybo
- **Supported Operating Systems**
  - Windows 7 SP1 Professional (64 Bit)
  - Red Hat Enterprise Linux 6.5 – 6.6 (64 Bit)
  - Red Hat Enterprise Linux 7.0 (64 Bit)
  - Ubuntu Linux 14.04 LTS (64 Bit)

Zynq Architecture and Vivado IPI

Zynq
SDSoC 2015.4 Version

Objectives

- **After completing this module, you will be able to:**
  - Identify the basic building blocks of the Zynq architecture processing system (PS)
  - List the available PS to the programmable logic (PL) connections through the AXI ports
  - Identify clocking sources for the PL peripherals
  - List the various AXI-based system architectural models
  - Describe what is Vivado IPI and how it can be used to view the SDSoC generated hardware

Outline

- ***Zynq All Programmable SoC (AP SoC)***
- **Processor Peripherals and Interfaces**
- **Clock, Reset, and Debug Features**
- **AXI Interfaces**
- **Vivado IPI**
- **Summary**
- **Appendix**

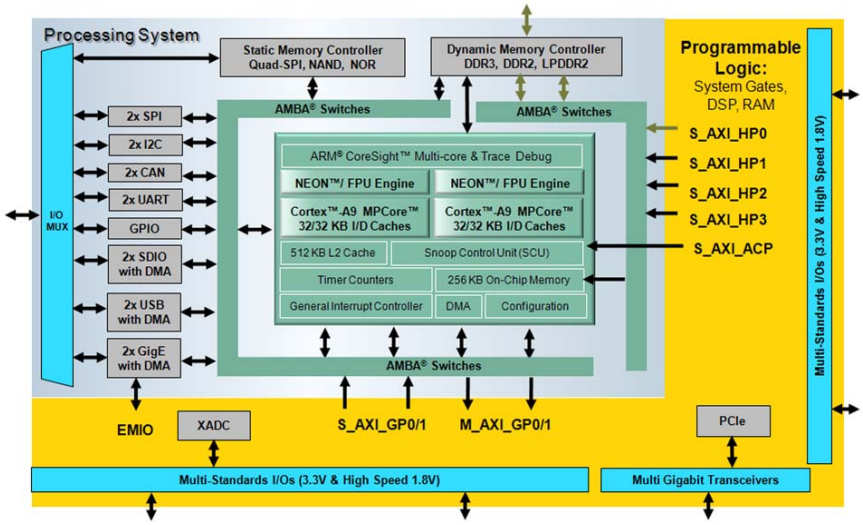
The PS and the PL

- **The Zynq-7000 AP SoC architecture consists of two major sections**
  - PS: Processing system
    - Dual ARM Cortex-A9 processor based
    - Multiple peripherals
    - Hard silicon core
  - PL: Programmable logic
    - Shares the same 7 series programmable logic as
      - Artix™-based devices: Z-7010, Z-7015, and Z-7020 (high-range I/O banks only)
      - Kintex™-based devices: Z-7030, Z-7035, Z-7045, and Z-7100 (mix of high-range and high-performance I/O banks)

Zynq-7000 Family Highlights

- **Complete ARM®-based processing system**
  - Application Processor Unit (APU)
    - Dual ARM Cortex™-A9 processors
    - Caches and support blocks
  - Fully integrated memory controllers
  - I/O peripherals
- **Tightly integrated programmable logic**
  - Used to extend the processing system
  - Scalable density and performance
- **Flexible array of I/O**
  - Wide range of external multi-standard I/O
  - High-performance integrated serial transceivers
  - Analog-to-digital converter inputs

Zynq-7000 AP SoC Block Diagram

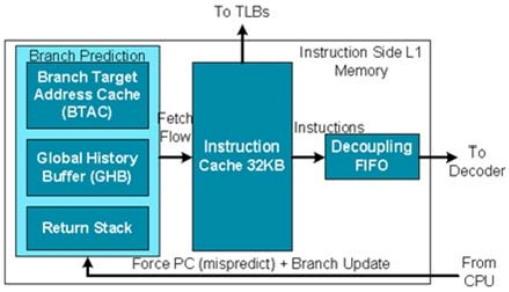


PS Components

- **Application processing unit (APU)**
- **I/O peripherals (IOP)**
  - Multiplexed I/O (MIO), extended multiplexed I/O (EMIO)
- **Memory interfaces**
- **PS interconnect**
- **DMA**
- **Timers**
  - Public and private
- **General interrupt controller (GIC)**
- **On-chip memory (OCM): RAM**
- **Debug controller: CoreSight**

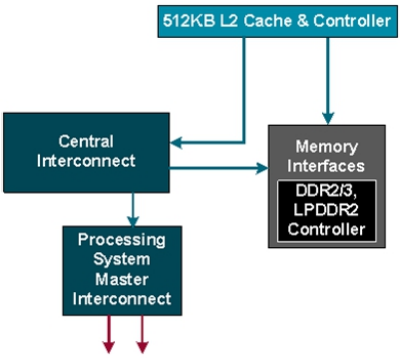
L1 Cache Features

- **Separate instruction and data caches for each processor**
- **Caches are four-way, set associative and are write-back**
- **Non-lockable**
- **Eight words cache length**
- **On a cache miss, critical word first filling of the cache is performed followed by the next word in sequence**



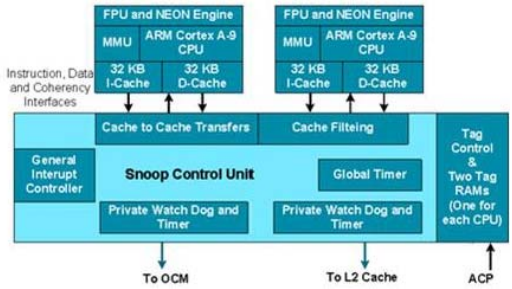
L2 Cache Features

- **512K bytes of RAM built into the SCU**
  - Latency of 25 CPU cycles
  - Unified instruction and data cache
- **Fixed, 256-bit (32 words) cache line size**
- **Support for per-master way lockdown between multiple CPUs**
- **Eight-way, set associative**
- **Two AXI interfaces**
  - One to DDR controller
  - One to programmable logic master (to peripherals)



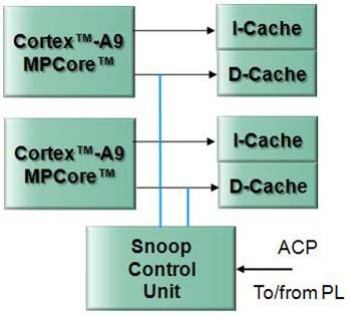
Snoop Control Unit (SCU)

- **Shares and arbitrates functions between the two processor cores**
  - Data cache coherency between the processors
  - Initiates L2 AXI memory access
  - Arbitrates between the processors requesting L2 accesses
  - Manages ACP accesses
  - A second master port with programmable address filtering between OCM and L2 memory support



Cache Coherency using SCU

- **High-performance, cache-to-cache transfers**
- **Snoop each CPU and cache each interface independently**
- **Coherency protocol is MESI**
  - M: Cache line has been modified
  - E: Cache line is held exclusively
  - S: Cache line is shared with another CPU
  - I: Cache line is invalidated
- **Uses Accelerator Coherence Port (ACP) to allow coherency to be extended to PL**

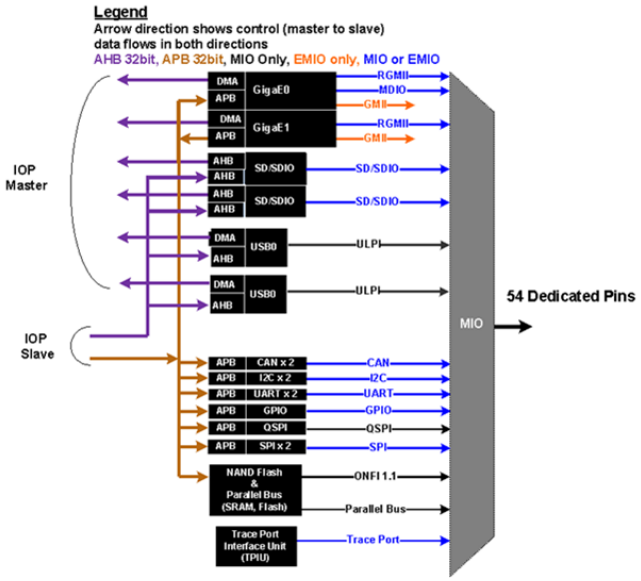


Outline

- **Zynq All Programmable SoC (AP SoC)**
- ***Processor Peripherals and Interfaces***
- **Clock, Reset, and Debug Features**
- **AXI Interfaces**
- **Vivado IPI**
- **Summary**
- **Appendix**

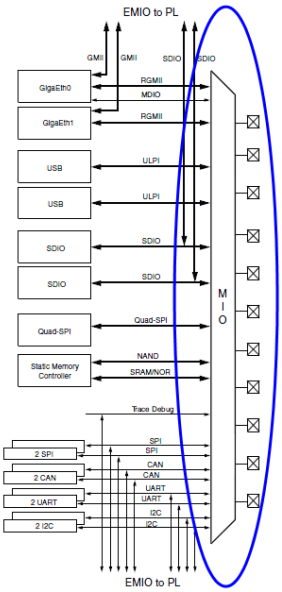
Input/Output Peripherals

- Two GigE
- Two USB
- Two SPI
- Two SD/SDIO
- Two CAN
- Two I2C
- Two UART
- Four 32-bit GPIOs
- Static memories
  - NAND, NOR/SRAM, Quad SPI
- Trace ports



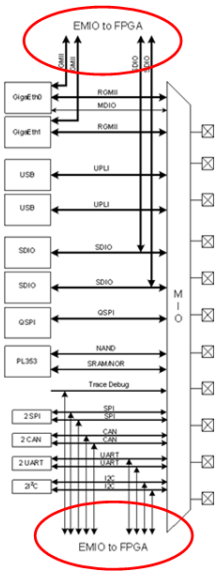
Multiplexed I/O (MIO)

- External interface to PS I/O peripheral ports
  - 54 dedicated package pins available
  - Software configurable
    - Automatically added to bootloader by tools
  - Not available for all peripheral ports
    - Some ports can only use EMIO



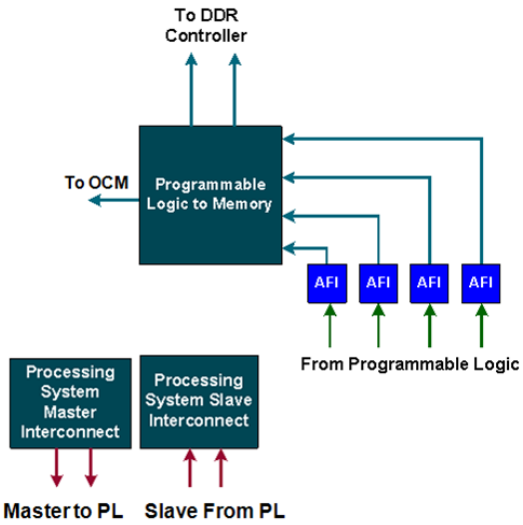
Extended Multiplexed I/O (EMIO)

- **Extended interface to PS I/O peripheral ports**
  - EMIO: Peripheral port to programmable logic
  - Alternative to using MIO
  - Mandatory for some peripheral ports
  - Facilitates
    - Connection to peripheral in programmable logic
    - Use of general I/O pins to supplement MIO pin usage
    - Alleviates competition for MIO pin usage



PS-PL Interfaces

- **AXI high-performance slave ports (HP0-HP3)**
  - Configurable 32-bit or 64-bit data width
  - Access to OCM and DDR only
  - Conversion to processing system clock domain
  - AXI FIFO Interface (AFI) are FIFOs (1KB) to smooth large data transfers
- **AXI general-purpose ports (GP0-GP1)**
  - Two masters from PS to PL
  - Two slaves from PL to PS
  - 32-bit data width
  - Conversation and sync to processing system clock domain



PS-PL Interfaces

- **One 64-bit accelerator coherence port (ACP) AXI slave interface to CPU memory**
- **DMA, interrupts, events signals**
  - Processor event bus for signaling event information to the CPU
  - PL peripheral IP interrupts to the PS general interrupt controller (GIC)
  - Four DMA channel RDY/ACK signals
- **Extended multiplexed I/O (EMIO) allows PS peripheral ports access to PL logic and device I/O pins**
- **Clock and resets**
  - Four PS clock outputs to the PL with enable control
  - Four PS reset outputs to the PL
- **Configuration and miscellaneous**

Outline

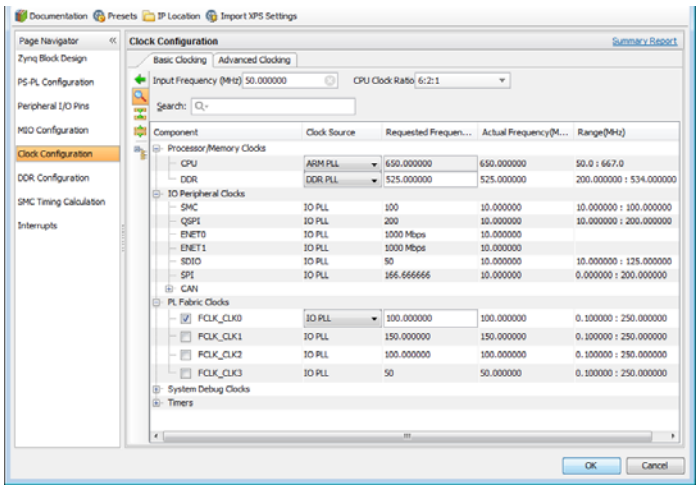
- **Zynq All Programmable SoC (AP SoC)**
- **Processor Peripherals and Interfaces**
- ***Clock, Reset, and Debug Features***
- **AXI Interfaces**
- **Vivado IPI**
- **Summary**
- **Appendix**

PL Clocking Sources

- **PS clocks**
  - PS clock source from external package pin
  - PS has three PLLs for clock generation
  - PS has four clock ports to PL
- **The PL has 7 series clocking resources**
  - PL has a different clock source domain compared to the PS
  - The clock to PL can be sourced from external clock capable pins
  - Can use one of the four PS clocks as source
- **Synchronizing the clock between PL and PS is taken care by the architecture of the PS**
- **PL cannot supply clock source to PS**

Clock Generation (Using Zynq Tab)

- **The Clock Generator allows configuration of PLL components for both the PS and PL**
  - One input reference clock
- **Access GUI by clicking the Clock Generation Block, or select from Navigator**
- **Configure the PS Peripheral Clock in the Zynq tab**
  - PS uses a dedicated PLL clock
  - PS I/O peripherals use the I/O PLL clock and ARM PLL
- **Clock to PL is disabled if PS clocking is present**



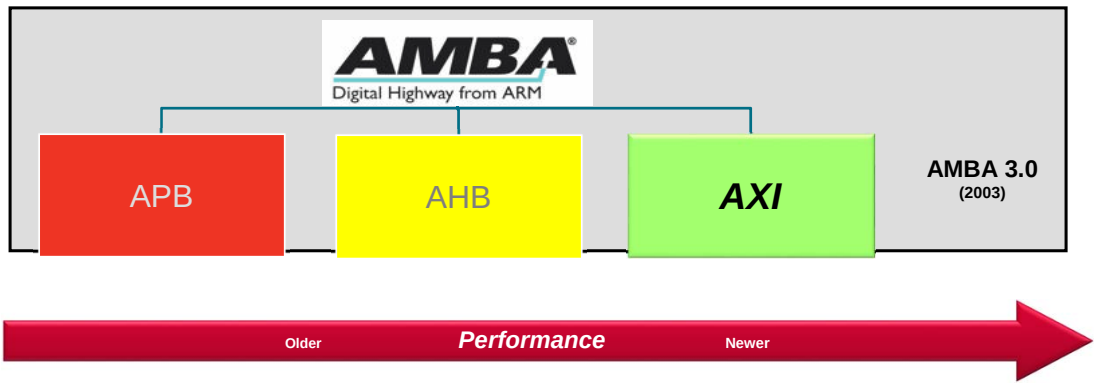
Zynq Resets

- **Internal resets**
  - Power-on reset (POR)
  - Watchdog resets from the three watchdog timers
  - Secure violation reset
- **PS resets**
  - External reset: PS_SRST_B
  - Warm reset: SRSTB
- **PL resets**
  - Four reset outputs from PS to PL
  - FCLK_RESET[3:0]

Outline

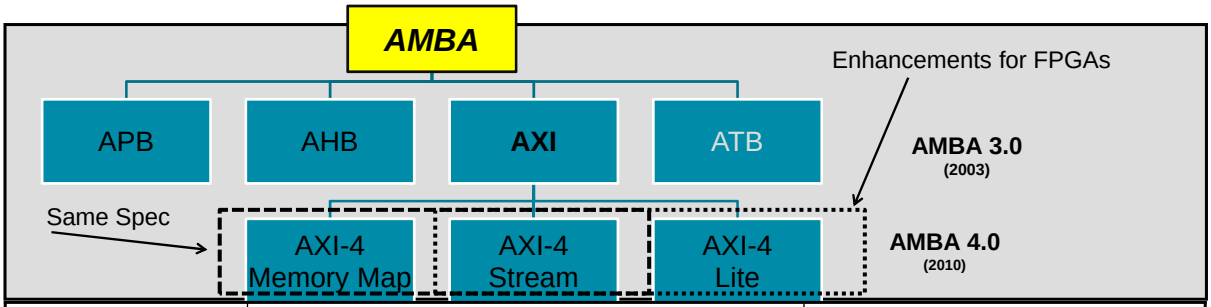
- **Zynq All Programmable SoC (AP SoC)**
- **Processor Peripherals and Interfaces**
- **Clock, Reset, and Debug Features**
- ***AXI Interfaces***
- **Vivado IPI**
- **Summary**
- **Appendix**

AXI is Part of ARM's AMBA



AMBA: Advanced Microcontroller Bus Architecture
AXI: Advanced Extensible Interface

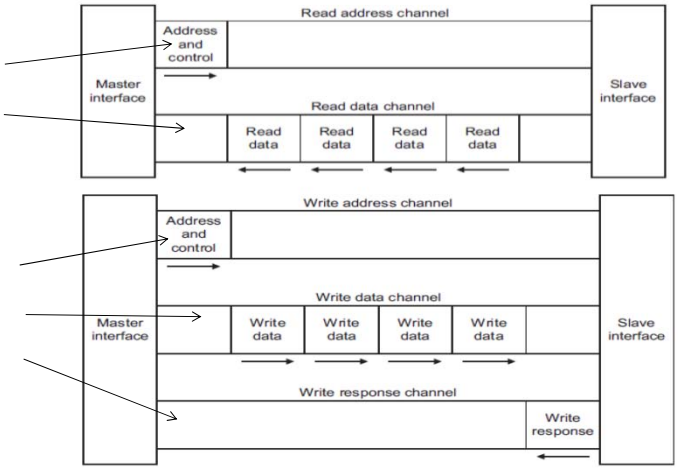
AXI is Part of AMBA



Interface	Features	Similar to
Memory Map / Full (AXI4)	Traditional Address/Data Burst (single address, multiple data)	PLBv46, PCI
Streaming (AXI4-Stream)	Data-Only, Burst	Local Link / DSP Interfaces / FIFO / FSL
Lite (AXI4-Lite)	Traditional Address/Data—No Burst (single address, single data)	PLBv46-single OPB

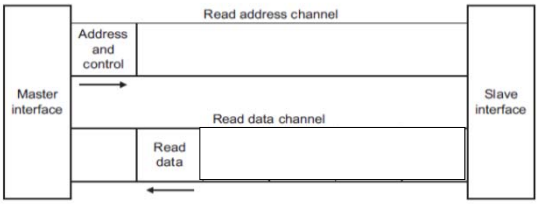
Basic AXI Signaling – 5 Channels

1. **Read Address Channel**
2. **Read Data Channel**
3. **Write Address Channel**
4. **Write Data Channel**
5. **Write Response Channel**

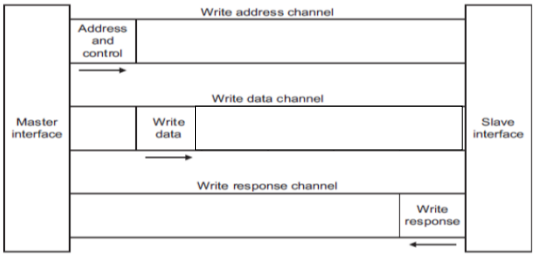


The AXI Interface—AXI4-Lite

- **No burst**
- **Data width 32 or 64 only**
  - Xilinx IP only supports 32-bits
- **Very small footprint**
- **Bridging to AXI4 handled automatically by AXI_Interconnect (if needed)**



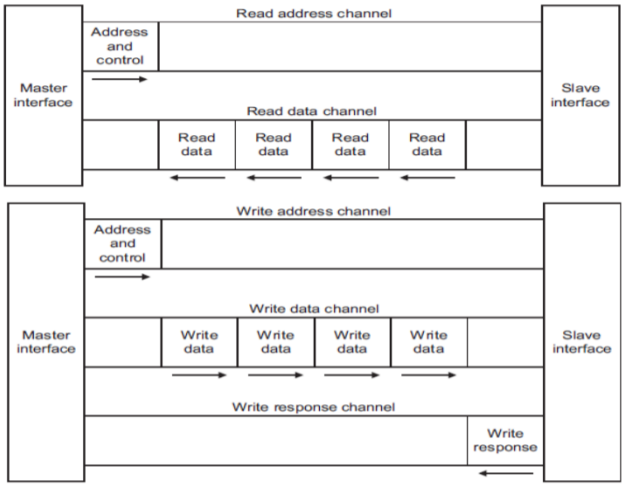
AXI4-Lite Read



AXI4-Lite Write

The AXI Interface—AXI4

- **Sometimes called "Full AXI" or Memory Mapped"**
  - Not ARM-sanctioned names
- **Single address multiple data**
  - Burst up to 256 data beats
- **Data Width parameterizable**
  - 1024 bits



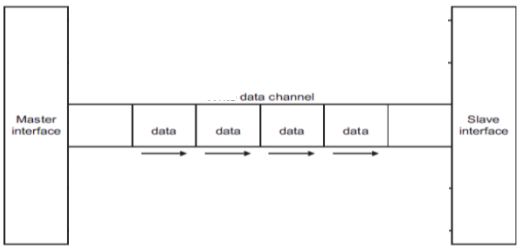
AXI4 Read

AXI4 Write

The AXI Interface—AXI4-Stream

- **No address channel, no read and write, always just master to slave**
  - Effectively an AXI4 "write data" channel
- **Unlimited burst length**
  - AXI4 max 256
  - AXI4-Lite does not burst
- **Virtually same signaling as AXI Data Channels**
  - Protocol allows merging, packing, width conversion
  - Supports sparse, continuous, aligned, unaligned streams

AXI4-Stream Transfer



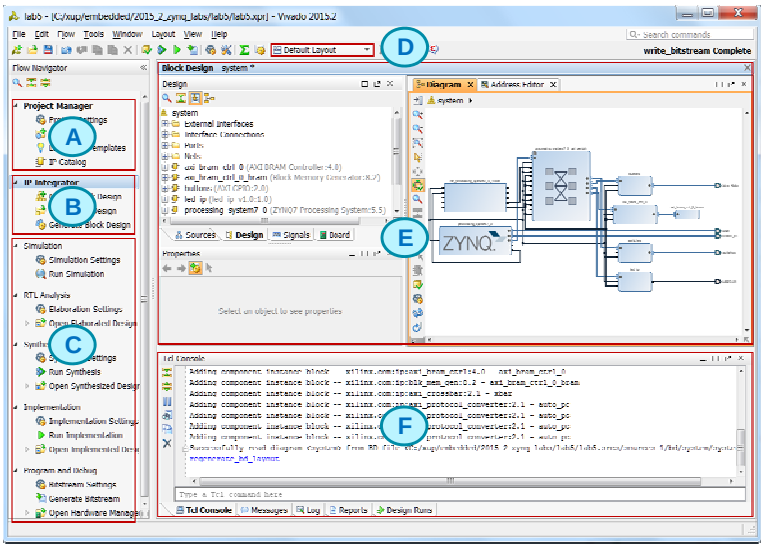
Streaming Applications

- **May not have packets**
  - E.g. Digital up converter
    - No concept of address
    - Free-running data (in this case)
    - In this situation, AXI4-Stream would optimize to a very simple interface
- **May have packets**
  - E.g. PCIe
    - Their packets may contain different information
    - Typically bridge logic of some sort is needed

Outline

Vivado View

- Customizable panels
- A: Project Management
- B: IP Integrator
- C: FPGA Flow
- D: Layout Selection
- E: Project view/Preview Panel
- F: Console, Messages, Logs

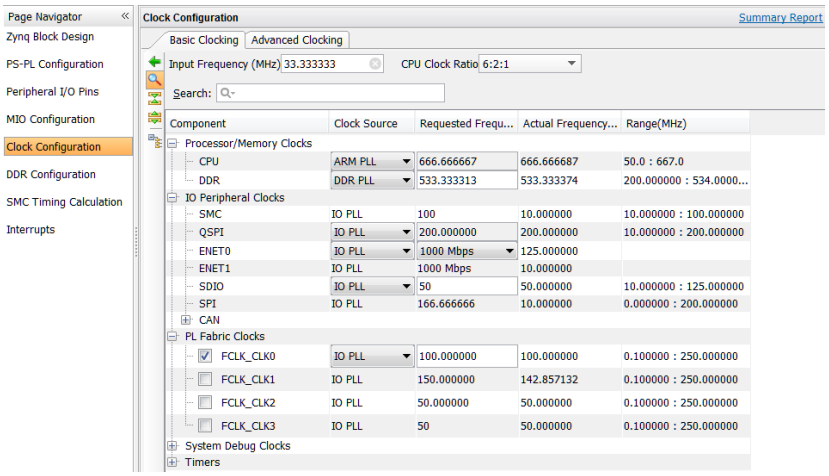


Zynq Configuration GUI

- Provides a graphical view of the PS to configure
  - ARM cores
  - I/O peripherals
  - DDR controller
  - Memory systems
- I/O partitioning between dedicated PS pins and programmable logic I/O
- Zynq-7000 AP SoC PS is configured via a set of memory-mapped configuration registers

Clock Configuration

- **Clock Configuration**
  - Input frequency can be set
    - Processor, DDR
  - All IOP clock frequencies can be set
  - PL fabric clocks can be enabled and configured
  - Set Timers



Outline

- Zynq All Programmable SoC (AP SoC)
- Processor Peripherals and Interfaces
- Clock, Reset, and Debug Features
- AXI Interfaces
- Vivado IPI
- ***Summary***
- Appendix

Summary

- **The Zynq-7000 processing platform is a system on a chip (SoC) processor with embedded programmable logic**
- **The processing system (PS) is the hard silicon dual core consisting of**
  - APU and list components
    - Two Cortex-A9 processors
    - NEON co-processor
    - General interrupt controller (GIC)
    - General and watchdog timers
  - I/O peripherals
  - External memory interfaces
- **The PS provides clocking resources to the PL**
- **Vivado IPI can be used to see the generated hardware**

Summary

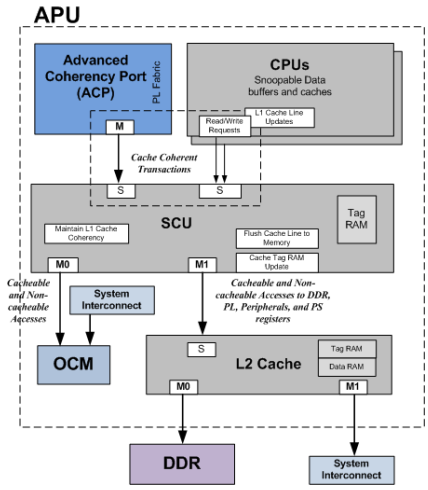
- **The programmable logic (PL) consists of 7 series devices**
- **Communication between the PS and the peripherals in PL may occur using**
  - Four HP slave ports
  - Two GP master ports
  - Two GP slave ports
  - An ACP slave port
- **AXI is an interface providing high performance through point-to-point connection**
- **AXI has separate, independent read and write interfaces implemented with channels**
- **The AXI4 interface offers improvements over AXI3 and defines**
  - Full AXI memory mapped
  - AXI Lite
  - AXI Stream

Outline

- Zynq All Programmable SoC (AP SoC)
- Processor Peripherals and Interfaces
- Clock, Reset, and Debug Features
- AXI Interfaces
- Vivado IPI
- Summary
- ***Appendix***

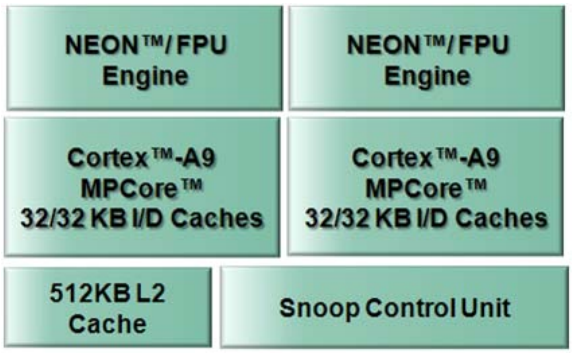
Introduction to the APU

- Heart of the PS
- Tightly coupled processors and sub-components for maximum performance
- Tied to other PS components and PL via the PS interconnect



Inside the APU

- **Dual ARM® Cortex™-A9 MPCore with NEON extensions**
  - Up to 1GHz operation
  - 2.5 DMIPS/MHz per core
  - Separate 32KB instruction and data caches
- **Snoop control unit**
  - L1 cache snoop control
    - Accelerator coherency port
- **Level 2 cache and controller**
  - Shared 512 KB cache with parity



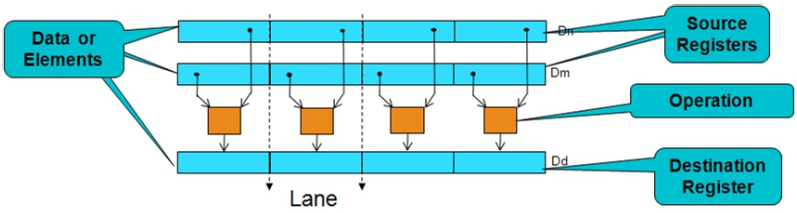
APU Sub-Components

- **General interrupt controller (GIC)**
- **On-chip memory (OCM): RAM and boot ROM**
- **Central DMA (eight channels)**
- **Device configuration (DEVCFG)**
- **Private watchdog timer and timer for each CPU**
- **System watchdog and triple timer counters shared between CPUs**
- **ARM CoreSight debug technology**

Vector Processing using NEON

- **NEON is the ARM codename for the vector processing unit**
  - Provides multimedia and signal processing support
- **FPU is the floating-point unit extension to NEON**
  - Both NEON and FPU share a single set of registers
- **NEON technology is a wide single instruction, multiple data (SIMD) parallel and co-processing architecture**
  - 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide)
  - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, or 32-bit float

NEON™/FPU Engine



SDSoC Development Environment

Zynq
SDSoC 2015.4 Version

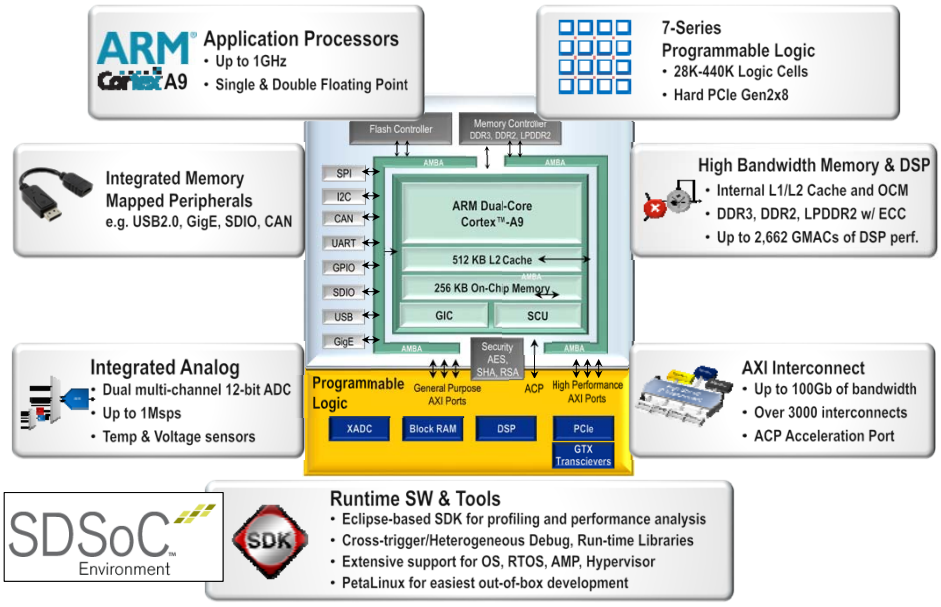
Objectives

- **After completing this module, you will be able to:**
  - Describe the SDSoC development environment
  - List some examples of SDSoC use cases
  - List some of the benefits of using SDSoC
  - Identify some of the underlying tools the SDSoC development environment uses
  - Describe the SDSoC development flow
  - List steps involved in creating an SDSoC project

Outline

- ***SDSoC Development Environment***
- **Use Cases**
- **SDSoC Development Flow**
- **SDSoC Project Creation**
- **Summary**
- **Lab1 Intro**
- **Appendix**

Zynq-7000 All Programmable SoC Highlights



SDSoC Development Environment

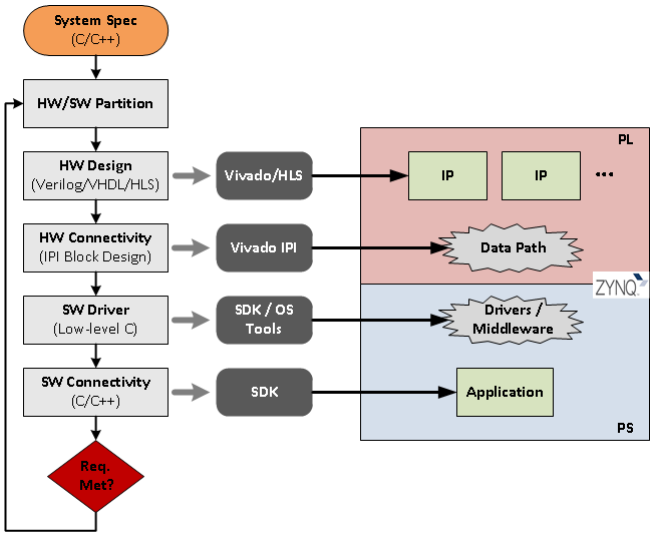
- **What is it?**
  - SDSoC provides a familiar embedded C/C++ application development experience including
    - An easy to use Eclipse IDE
    - A comprehensive design environment for heterogeneous Zynq SoC and MPSoC
    - C/C++ full-system optimizing compiler
  - SDSoC enables function acceleration in PL with a click of button using various technologies
    - Vivado IPI
    - Vivado HLS
    - SDK
    - Profiler
  - SDSoC provides infrastructure to combine processing system, accelerators, data movers, signaling, and drivers

Benefits of SDSoC Development Environment

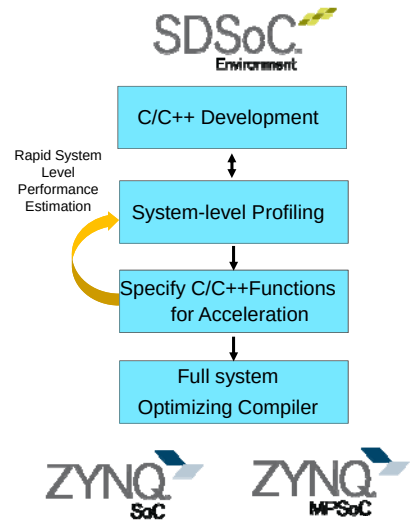
- **Shorter development cycles**
  - Estimation shows improvement over software-only solution for system and accelerators
  - Iterative improvement can be made at early stages of development without the need to build hardware
- **Simplified interface and partitioning between hardware and software**
  - Software-based flow vs RTL
  - User interfaces with software-only tools; hardware management is abstracted
  - Removes much of the hardware/software distinction and throw-over-the-wall effect
- **Automated initial design**
  - Users can tweak code at macro- and micro-architectural levels
  - Designers still have manual control over the constructed Vivado HLS tool and Vivado Design Suite projects

Development Flow Without SDSoC

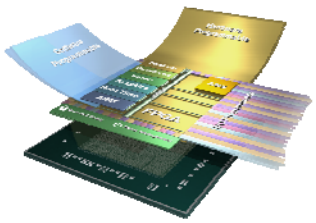
- **Overall process requires expertise at all steps in the development process**
  - Various hardware design entries
  - Hardware connectivity at system level
  - Driver development to drive custom hardware
  - Integrating with application and target OS



SDSoC - Unified Development Environment

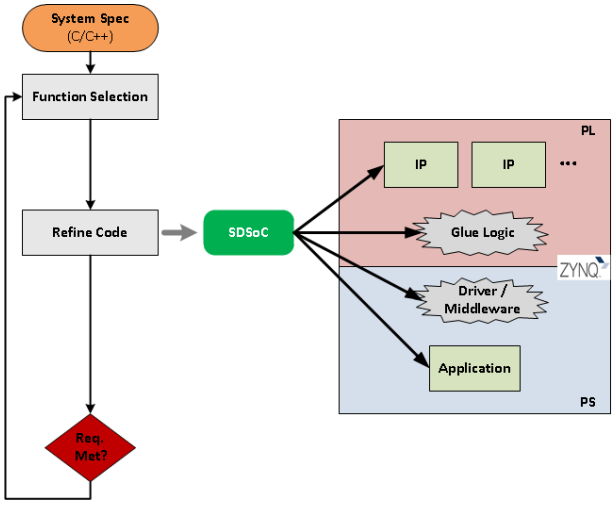


- **ASSP-like programming experience**
- **System-level profiling**
- **Full system optimizing compiler**
- **Expert use model for platform developers and system architects**



SDSoC Development Environment

- **SDSoC development environment consolidates a multi-step/multi-tool process into a single tool and reduces hardware/software partitioning to simple function selection**
  - Code typically needs to be refined to achieve optimal results

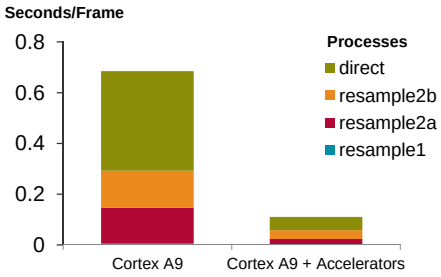


Outline

Example: Back Projection

- HW Solution using 52 floating point operations at 166MHz
- DMAs connected to ACP port for short communication times
- Two-accelerator system an early SDSoC proof of concept

Used in tomography applications, including CAT scanners



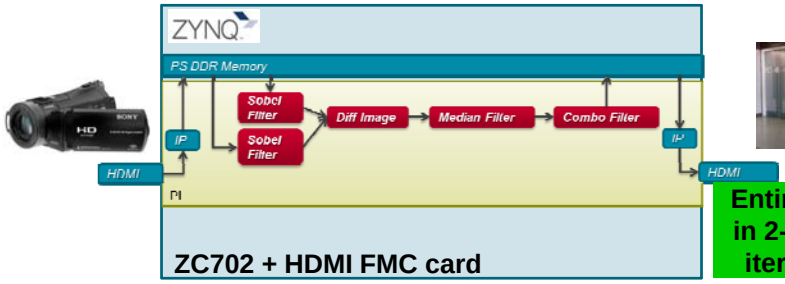
7x Speedup

Example: Motion Detection @ 1080p60

```
main(){
 get_camera(A);
 sobel(A,B);
 diff(B,C);
 …
 display(out);
}
```

ZC702 + HDMI FMC Platform

SDSoC Environment

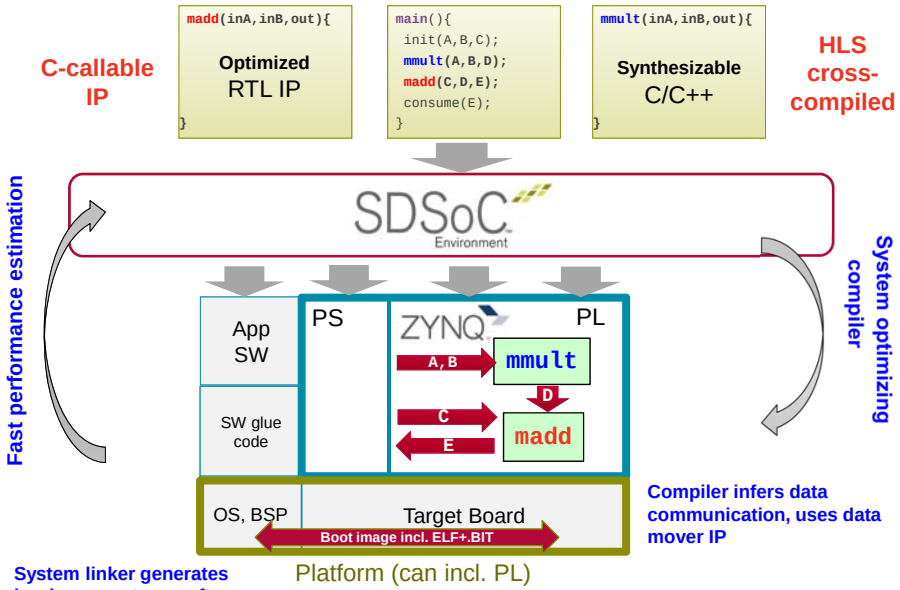


Entire design completed in 2-weeks with multiple iterations of algorithm

Outline

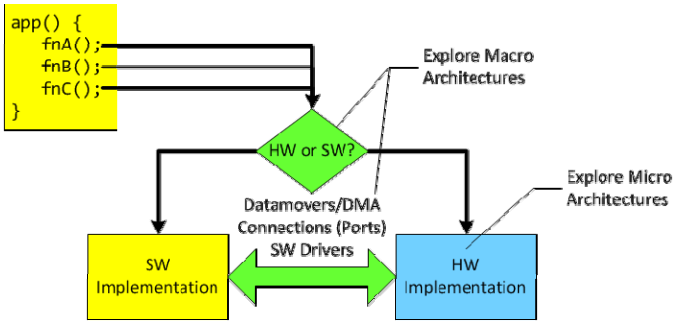
- SDSoC Development Environment
- Use Cases
- ***SDSoC Development Flow***
- SDSoC Project Creation
- Summary
- Lab1 Intro
- Appendix

SDSoC Development Flow



System-level Considerations

- **What gets accelerated?**
- **How is software implemented in hardware?**
  - Is hardware design expertise available?
- **How will software and hardware talk to each other?**
- **Will it meet performance requirements the first try?**
  - What changes are required at the macro/micro-architecture levels (or both)?



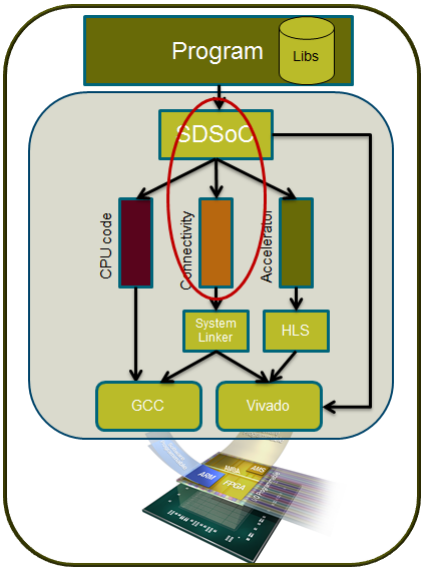
Candidate for an Accelerator

- **Not every function can be a candidate for acceleration**
  - For example, pre-compiled code, some user libraries, OS services, etc.
- **Computationally intensive algorithms are ideal candidates**
  - Profiling results help identify performance bottlenecks
    - Just because a function takes a long time, it is not automatically a candidate
  - Slight code modifications (or pragmas) effect the microarchitecture of the accelerator
- **Trade-off between data movement costs and acceleration benefits should be considered**
- **Amdahl's law can be used to determine potential speedups**

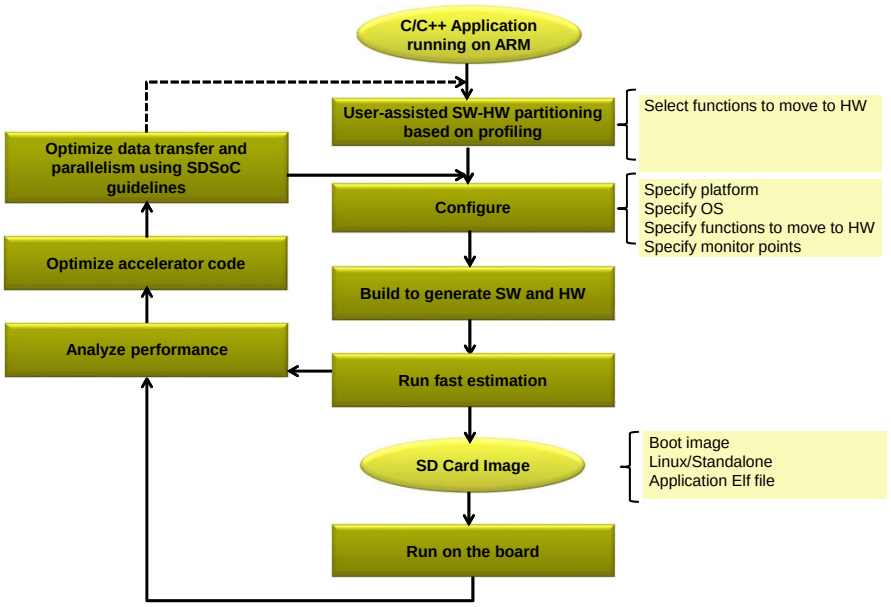
$$S = \frac{1}{(1-\alpha) + \alpha/p}$$

Simplifying Hardware/Software Partitioning

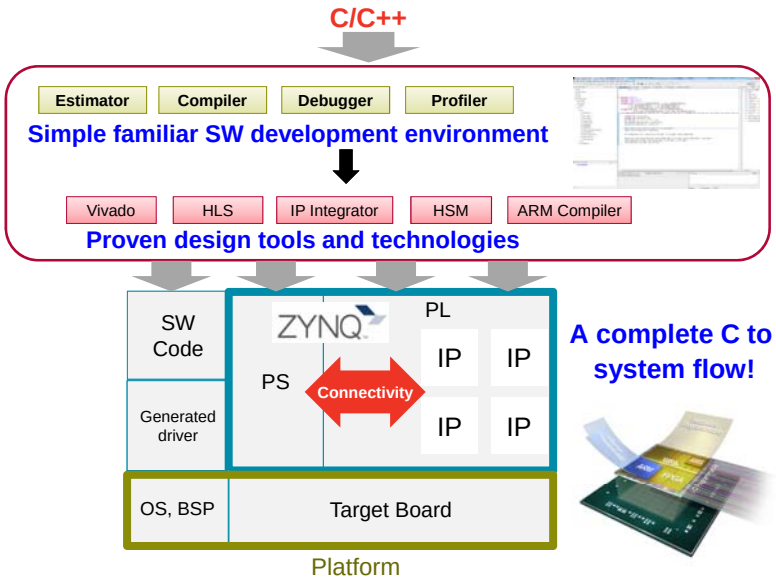
- **SDSoC development environment simplifies the process of identifying and accelerating functions**
  - Integrates profiling tools
  - Provides libraries for accurate timing measurements
  - *User selects functions* to accelerate
  - Estimates hardware speed-up
- ***Automated* acceleration is achieved through**
  - Advanced analysis of code using specialized compilers
  - Assistance of user-specified pragmas
  - Extension of a predefined hardware platform
- **Hardware platforms packaged with the environment and from third parties**
  - Environment extends platform, providing additional capabilities



User Development Flow

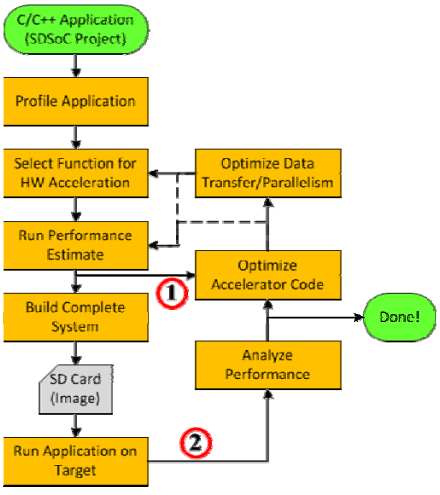


SDSoC: Complete End-to-End Flow



Development Flow

- **Without hardware generation, the SDSoC development environment acts like a normal C/C++ IDE**
- **It starts with a pure software system and ends with an accelerated SW/HW system**
  - Developers select C functions for hardware acceleration
- **SDEstimate provides quick-turns to get architectures close**
  - Used in early development
- **SDRelease generates full hardware/software products**
  - For later and final stages of development

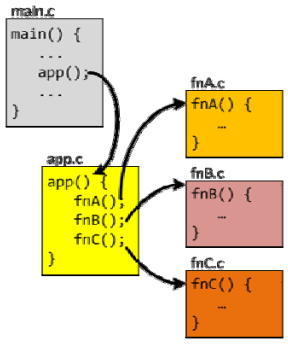


Profiling the Application

- **The goal of offloading software to hardware is to improve overall system performance**
- **Finding software eligible for acceleration follows general software optimization practices**
  - Profiling yields the necessary data by summarizing where execution time was spent
- **The SDSoC development environment includes the TCF profiling tool**
  - No code instrumentation or special recompiling required
  - Profiling results show percentage of time spent in functions relative to a total execution time
- **For absolute timing measurements *sds_lib* offers some functions for collecting system timer values**
  - Timer resource must be available and code must be changed to incorporate timer functions

Accelerating Software Functions

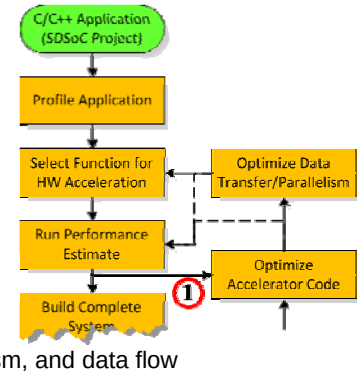
- **Individual functions must be marked for acceleration**
  - Functions selected for acceleration are "hardware functions“
    - Hardware functions can contain sub-functions
  - Refactoring required if critical points are not isolated as functions
  - Each function becomes an individual piece of IP
- **OS-based platforms have only one master thread communicates with all cores**
- **Restrictions**
  - Only one function per file scope
    - Splitting of files may be required to accelerate multiple functions
  - Call to a hardware function must reside in a different file scope than the hardware function
  - Other tool limitations may apply
    - See Vivado HLS tool C/C++ guidelines



Estimating Performance

- **One of three build configurations: SDEstimate, SDDebug, SDRelease**
- **The Estimate configuration builds a project only up to a point that performance estimates can be determined**
  - A complete Release or Debug build can take long time
  - The SDEstimate configuration shows the impact of moving functions to hardware in only a few minutes
- **Comparison of hardware-accelerated system with original software-only system is accomplished by producing two separate sets of data, one for each scenario**
  - The Vivado HLS tool is used to produce data for hardware estimates
  - Generated ELF must be run on the actual target to generate data for software estimates
  - IDE provides a Report viewer that compares and illustrates the data

Initial Development Flow



- **Building the Estimate configuration yields enough information to fine-tune optimization**
- **First attempts often produce poorer performance than software alone**
  - Typical when code has not been refactored and/or annotated for acceleration
- **Recommendation: Initial design iterations should use the "short" path until performance improvements are seen**
  - Begin by optimizing code to improve accelerator performance, parallelism, and data flow
  - Rebuild to observe impact of changes on estimates
  - Repeat cycle until satisfactory results are achieved
- **Detailed system analysis and fine-tuning should be performed next**
  - Building and analyzing the full HW/SW system

Building the Complete System

- **Building a project is where SDSoC does all the heavy lifting**
  - Three basic methods to run the build process
    - Through the IDE, using a makefile, directly from the command line
  - It is during this step that SDSoC calls upon its internal framework and several sub-executables to accomplish
    - (1) system compilation, (2) system analysis, (3) system generation
- **Activate the SDRelease build configuration for a fully optimized build**
- **A successful build will result in several output products including the following key items**
  - Application program in ELF format
  - Bitstream (optional)
  - SD card image (optional)
  - Design projects (e.g. Vivado Design Suite and Vivado HLS tool)

Running the Application

- **SDSoC development environment requires use of an SD card to run applications for Linux**
  - Files required to run the application on hardware are grouped together as an SD card image
    - First stage bootloader (FSBL), bitstream, and second stage bootloader (SSBL) embedded in a single binary file (BOOT.BIN)
    - Linux kernel
    - Application ELF
    - Other miscellaneous files
  - ELF must be explicitly run from the mounted SD card directory
- **Standalone applications do not require but can use an SD card**
  - When SD card is used, the SSBL in the binary file is replaced by the application ELF
    - Boots directly to the application ELF
  - Applications can be run from the IDE using an SDSoC tool Run configuration

Analyzing the Built System

- **The goal of this step is to analyze the performance of the updated hardware/software system**
- **The same tools and techniques used initially apply here: profiling, instrumentation**
  - Profiler tool for relative software execution times
  - Time-stamping functions provided through SDSoC API (sds_lib)
- **System consists of hardware and software with additional tools required for complete analysis**
  - Vivado HLS tool provides reports detailing generated hardware (included with output products)
  - Vivado Design Suite can open the generated system hardware design to review additional reports, modify hardware
  - SDSoC tool compiler can optionally insert an AXI performance monitor (APM) into the generated system
    - APM monitors dataflow between the PS-PL

Optimizing the System

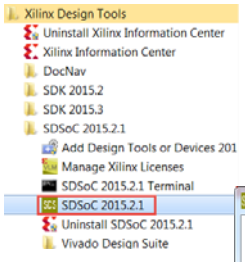
- **Optimizing the system is broken down into two steps: accelerator and then data flow and parallelism**
- **Optimizing the accelerator consists of refining and/or annotating the hardware function**
  - Re-factoring code to be more HLS-friendly
  - Adding code annotations (i.e., pragmas) that HLS can interpret and use to better optimize the accelerator
- **For system-level optimizations (data-flow/parallelism) there are several options including**
  - Using physically contiguous memory
  - Using shared memory
  - Using multiple instances of the same hardware
  - Overriding HLS default behavior

Outline

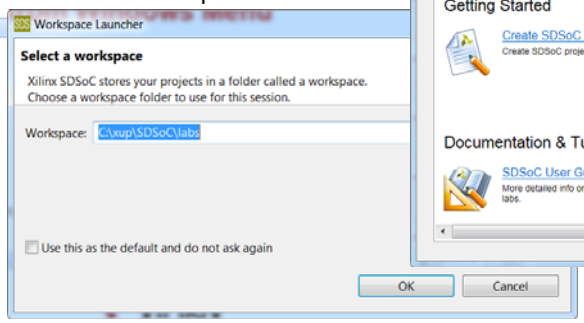
- SDSoC Development Environment
- Use Cases
- SDSoC Development Flow
- ***SDSoC Project Creation***
- Summary
- Lab1 Intro
- Appendix

Invoke SDSoC from Windows Menu

1. Start SDSoC



2. Select a workspace



3. If workspace empty then Welcome screen appears otherwise projects stored in the workspace are displayed

Creating, Importing, and Opening Projects

- **You can create a new project in the selected workspace by clicking on the respective link on the Welcome screen or selecting File > New > SDSoC Project**
  - Identify project name
  - Select target platform
  - Select target OS
  - Select either available template or an empty project option
- **You can import an existing project**
  - Select archive file if the project is in an archived form
  - Identify path if the project is saved in a hierarchical directory structure

Project Definition

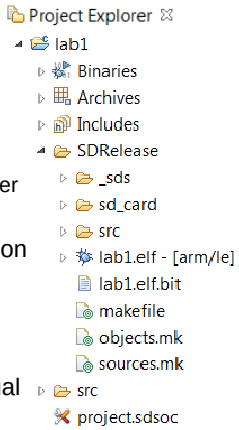
- **An SDSoC tool project is a folder in a workspace that contains all project files and tool metafiles**
- **SDSoC tool projects carry some additional properties beyond these general Eclipse-based concepts**
  - Associated with a specific SDSoC development environment platform
  - Built for a specific type of operating system (OS)
- **Built-in OS support: Linux (default), Standalone (a.k.a bare-metal), and FreeRTOS**
- **An SDSoC development environment platform is a baseline embedded system that the SDSoC tool can build on top of**
  - Consists of a Vivado Design Suite hardware design targeting a specific board
  - Can also include platform-specific software libraries
  - Support for several platforms is included with the SDSoC development environment but platforms can also come from third parties

Operating System Support

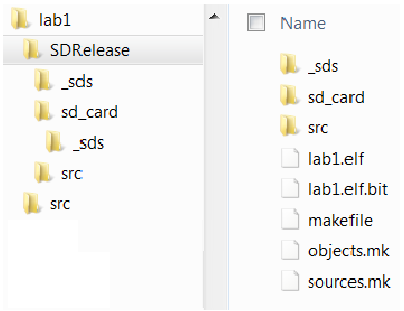
- **The output products of an SDSoC tool project vary depending on the targeted OS**
- **For standalone projects**
  - *swstubs* folder will include the Xilinx standalone library (*libxil*)
- **For Linux projects**
  - SD card image will include a pre-compiled Linux kernel and root file system
  - Pre-compiled kernel contains drivers to communicate with accelerators
- **For FreeRTOS projects**
  - Projects will link to a FreeRTOS library supplied by the SDSoC development environment as an extension to the GNU toolchain

Project Hierarchy

- **Project Explorer View**
  - The root folder of a project hierarchy is named after the project itself (*lab1*)
  - Three virtual branches are provided: Binaries, Archives, Includes
    - These list executables, libraries, and header files found in or used by the project
  - Other branches represent actual folders on the file system: build outputs and source files
  - The *src* folder holds source files
  - The final element at root level is the actual SDSoC project file: *project .sdsoc*

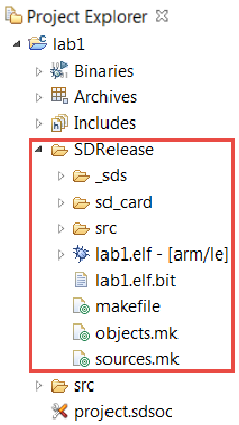


- **Windows Explorer View**
  - Hosts real files and directories



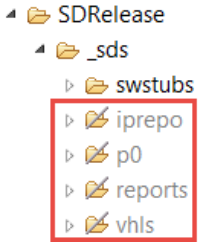
Project Hierarchy (2)

- **Common to every build folder are several important components**
  - The compiled application executable is placed at the root of the build folder (*lab1.elf*)
  - Several auto-generated make files used by the IDE to initiate the build process
  - The *_sds* folder is a catch all for everything else
    - Reports, files generated by back-end tools (e.g. Vivado Design Suite project), software stubs, etc.
  - An *sd_card* folder that contains the SD card image
    - Not strictly an image but rather a collection of files and folders
  - A *src* folder that contains compiled source files (i.e., object files)



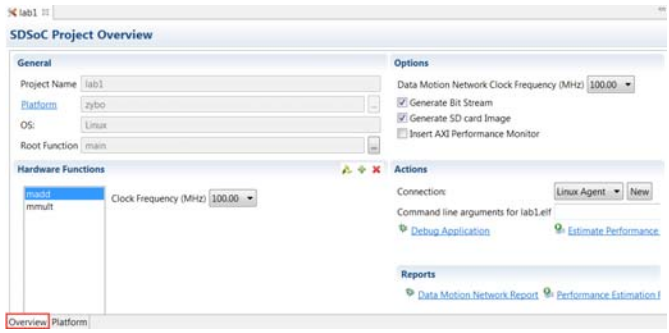
Project Hierarchy (3)

- **The *_sds* folder contains several sub-folders**
  - *swstubs*
    - Software files and stubs generated and supplied by the SDSoC development environment
    - Some original source code modified to interface with accelerators
  - *iprepo*
    - Repository of cores generated by the Vivado HLS tool (i.e., accelerators)
  - *p0*
    - Vivado IPI project and related files, can invoke Vivado by double-clicking on the *.xpr entry
  - *reports*
    - Various reports including data motion network
  - *vhls*
    - Vivado hls project files for each of the targeted accelerators
- **Grayed out folders indicate output products generated by independent back-end tools**

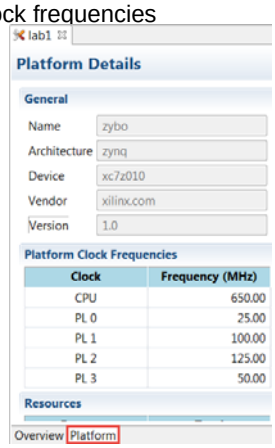


Project Overview

- **Overview tab**
  - Select functions which are hardware target
  - Select Data Motion network clock frequency
  - Other options



- **Platform tab**
  - Shows available clock frequencies



Outline

- SDSoC Development Environment
- Use Cases
- SDSoC Development Flow
- SDSoC Project Creation
- ***Summary***
- Lab1 Intro
- Appendix

Summary

- **The SDSoC development environment provides a familiar embedded C/C++ application development experience**
- **Provides infrastructure to combine processing system, accelerators, data movers, signaling, and drivers**
- **Benefits of using SDSoC include**
  - Shorter development cycles
  - Simplified interface and partitioning between hardware and software
- **SDSoC may be suitable for video, communication, image and digital signal processing applications where either large amount of data and/or higher throughput is required**
- **SDSoC uses Vivado HLS, Vivado IPI, SDK tools**
- **SDSoC project creation involves identifying workspace, project name, selecting target platform, selecting OS, and either selecting pre-defined application templates or creating empty application**

Outline

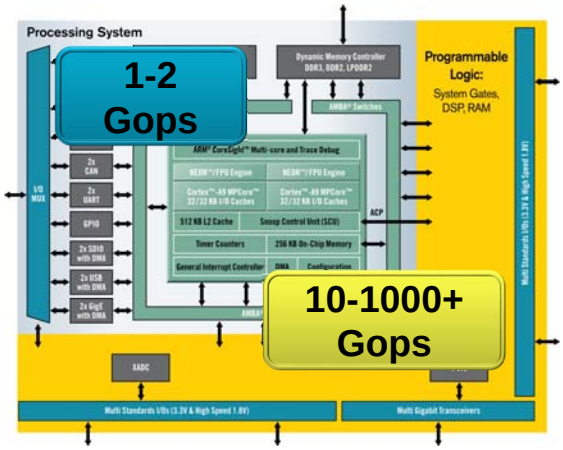
Lab1 Intro

- **Introduction**
  - This lab guides you through the process of using SDSoC to create a new project using available templates, mark a function for hardware implementation, build a hardware implemented design, and run the project on either ZedBoard or ZYBO board
- **Objectives**
  - Create a new SDSoC environment project for your application from a number of available platforms and project templates
  - Mark a function for hardware implementation
  - Build your project to generate a bitstream containing the hardware implemented function and an executable file that invokes this hardware implemented function
  - Test the design in hardware

Outline

Advantage of FPGA Hardware Accelerators

- **Processing System (PS)**
  - Two ARM® Cortex™-A9 with NEON™ extensions
  - Floating Point support
  - Up to 1 GHz operation
  - L2 Cache – 512KB Unified
  - On-Chip Memory of 256KB
  - Integrated Memory Controllers
  - Run full Linux
- **Programmable Logic (PL)**
  - 28K-444K logic cells
  - High bandwidth AMBA interconnect
  - ACP port - cache coherency for additional soft processors



Programmable Logic Provides Superior Performance

Example: Communications Algorithm

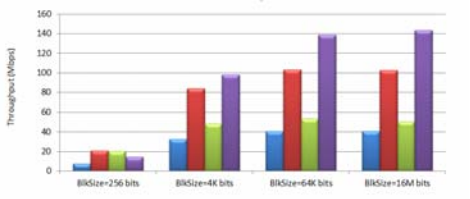
- **Customer Data**
  - Function names redacted
  - CPU cycles to complete the function

Function	Software (cycles)	Hardware (cycles)	Speed Up
Function 1	338~258	61 ~ 30	5-8x
Function 2	Case type: 1: 351 2: 357 3: 448	Case type: 1: 13 2: 13 3: 13	27-37x
Function 3	277~13	13	1-21x
Function 4	1351~492	739~106	2-4x

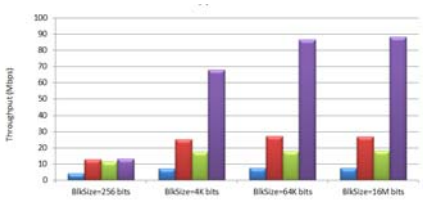
Example: Communications Algorithm

- **ZUC encryption algorithm**
  - Part of the 3GPP LTE EEA/EIA-3
- **EEA performance**
  - 50% with 16M data set
- **EIA performance**
  - 4X with 16M data set
- **Smaller data sets**
  - Typically show less improvement

EEA Block



EIA Block

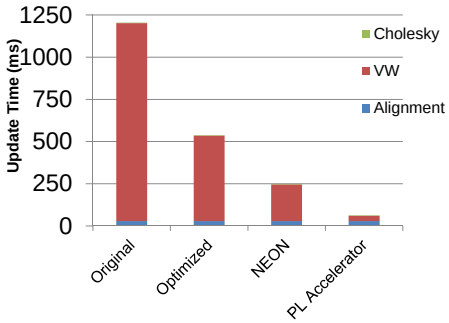


ARM A9 without Compilation Optimation
ARM A9 with Compilation Optimation
Baseline Accelerator
Optimized Accelerator

Example: Digital Pre-distortion

- **ARM NEON function intrinsics**
  - Low-level ARM-A9 programming
  - Sometimes software optimization is good enough (5X in this case)
  - HLS is better but uses HW
- **Data transfer from GP port through AXI interconnect and FIFO**

70x speedup for the main feedback update function



Hardware	FF	LUT
AXI Infrastructure	~300	~300
Accelerator	2552	2605

Data Motion Network

Zynq
SDSoC 2015.4 Version

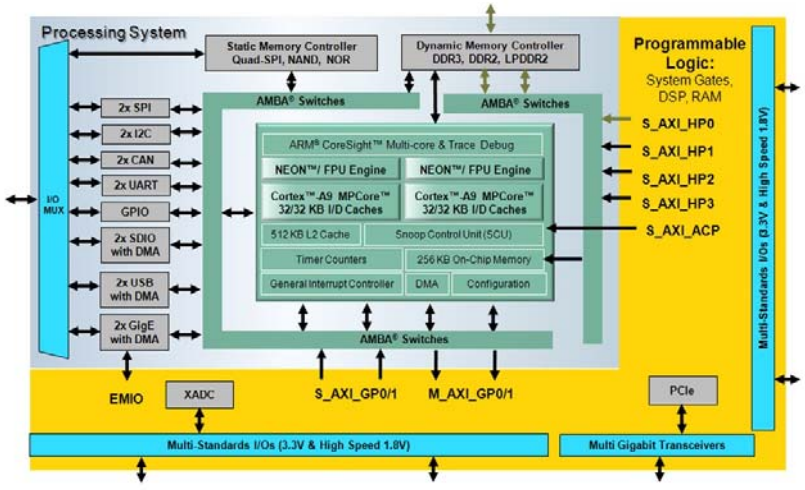
Objectives

- **After completing this module, you will be able to:**
  - List various ports available to move data between the processor and user-generated IP in PL
  - Describe at the hardware level how caches interact with the accelerator coherency port (ACP)
  - List available functions to perform data exchanges and synchronize events
  - Identify some of the techniques used for hardware optimization

Outline

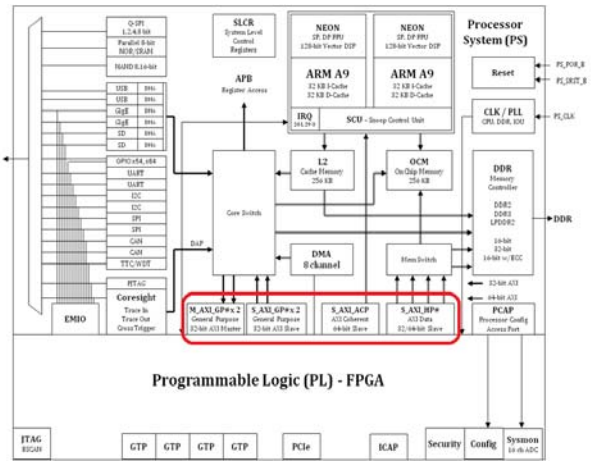
- *Architectural Support for Data Motion*
- Data Motion
- Optimization
- Summary
- Lab2 Intro

Zynq-7000 All Programmable SoC Block Diagram



PS-PL Interface AXI Ports

- **The AMBA AXI ports of the PS-PL interface provide the primary mechanism for the flow of data between the PS and PL**
  - Two general-purpose master ports
  - Two general-purpose slave ports
  - Four high-performance slave ports
  - One accelerator coherency port (ACP) slave port

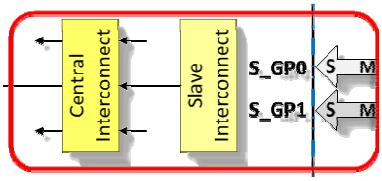


General-Purpose Master Ports

- **Two identical 32-bit AXI master ports**
  - M_AXI_GP0
  - M_AXI_GP1
- **Attaches to slave AXI ports in programmable logic via PL AXI interconnect**
  - PL slave
    - Your peripheral built in programmable logic
    - IP Integrator interface or other third-party-based IP
- **Mostly used for CPU and I/O peripheral block data movement to programmable logic**
- **Why two ports?**
  - Each port, having it's own 1 GB space, is capable of driving a number of peripherals using an AXI switch
  - Multiple ports enable the designer to distribute bandwidth
- **PS masters can be**
  - Cortex-A9 processors via the L2 cache controller
  - USB, Ethernet, SD/SDIO controllers
  - DMAC
  - Debug access port

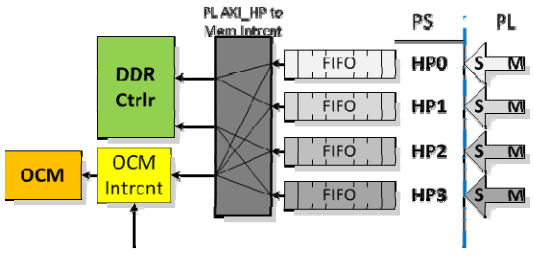
General-Purpose Slave Ports

- **Two identical 32-bit AXI slave ports**
  - S_AXI_GP0
  - S_AXI_GP1
- **Provides initiator access from PL master to PS target**
- **Attaches to master AXI port in PL via AXI interconnect**
  - PL master
    - MicroBlaze processor
    - Your master built in programmable logic
    - Other third-party-based IP
- **PS slaves can be**
  - DDRx controller
  - On-chip memory (OCM)
  - IOP peripheral



High-Performance Slave Ports

- **Asynchronous crossing between the PL clock domain and PS clock domain**
  - S_AXI_HP0, S_AXI_HP1
  - S_AXI_HP2, S_AXI_HP3
- **Provides initiator access from PL master to PS memory**
- **Provides low latency access to DDR and OCM**
- **Attaches to master AXI port in PL**
  - DMA controller
  - MicroBlaze processor
  - Custom master built in programmable logic
  - Other third-party based IP
- **QoS supported from the programmable logic ports**
- **FIFOs smooth out of *long-latency* events**
  - 1 KB (128 by 64 bit) data FIFOs
  - Both read and write paths



Accelerator Coherency Port (ACP) Slave Port

- **One accelerator coherence port (ACP)**
  - S_AXI_ACP
  - High-performance, coprocessor interface to PL
    - Accelerators gain access to the CPU cache hierarchy
      - Enables PL to participate in the coherency among the caches and DDRx memory
    - Supported in hardware; no software needed
  - No drivers required for the ACP; however, the IP connected to the ACP may require drivers
- **Provides initiator access from PL master to PS target**
  - Via L1 and L2 cache
- **Attaches to master AXI port in PL**
  - Typically a PL-based co-processor/accelerator
  - Also could be
    - MicroBlaze processor
    - Custom master built in programmable logic
    - Other third-party-based IP

Why Connection to the SCU is Important for Accelerators

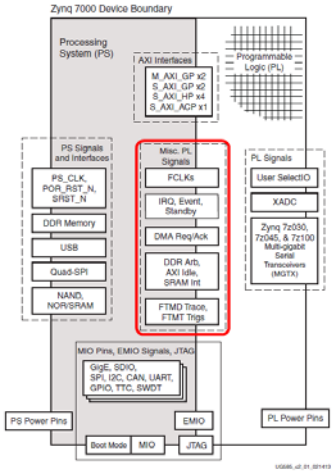
- **ACP bolts directly into the snoop control unit (SCU)**
  - ACP is a PS slave AXI port (that is, accelerator is a master)
  - Enables the accelerator on the ACP to write directly into the L1 and L2 caches
    - And indirectly to the DDR/OCM
  - Data movement is limited by the size of the cache target
    - Typically L2 is the target due to its larger size
    - Cache misses result in moving data to the DDR
    - Frequent misses indicate that the HP port (not the ACP port) should have been used
- **When the ACP is not used, there is no non-standard use of the caches**

Snoop Control Unit (SCU)

- **Shares and arbitrates functions between the two processor cores**
  - Data cache coherency between the Cortex-A9 processors
  - Initiates L2 AXI memory accesses
  - Arbitrates between Cortex-A9 processors requesting L2 accesses
  - Manage ACP accesses
  - A second master port with programmable address filtering between OCM and L2 memory support

Miscellaneous PS-PL Signals

- **The PS-PL interface includes several miscellaneous signals outside of the primary AXI ports**
- **These signals can be grouped into the following categories**
  - Clock & Reset
    - PS can provide up to four clock sources to PL
  - Interrupts
  - Event
  - DMA
  - Trace / Debug
  - EMIO
  - Other



System-Level Considerations that Determine Coding Techniques

- **Which PS interfaces to use for the accelerator?**
  - AXI_HP
  - AXI_ACP
  - AXI_GP (AXI_lite for control)
- **Where does the input to the accelerator connect from?**
  - From PS/PS DDR interface
  - MIG
  - From an external interface (like video)
- **Where does the output from the accelerator connect to?**
  - To PS
  - To an external interface
- **What is the optimal data movement mechanism between the PS and accelerator?**
  - DMA driven
    - Simple
    - Scatter gather
  - Software driven
  - How to manage cache coherency?
    - Software
    - SCU
- **What is the optimal signaling between PS and the accelerator?**
  - Interrupt driven
  - Event interface
  - Polled

Preferred Interfaces for Hardware Accelerators

- **AXI_HP DMA or AXI_ACP DMA give the highest performance**
  - Both DDR and OCM (4x64K) accessible, lower latency through HP ports
- **AXI_ACP offers further performance when cache coherent data transfers are implemented to/from the accelerator**
- **When is ACP more suited?**
  - Not suited for instruction-level acceleration; tightly coupled processor registers perform best
  - Not suited for large data sets; this can cause cache thrashing and degrade performance
  - Suited for small to moderate size data sets that can fit into L1 (<32K) or L2 (<256K) caches

Comparing Data Movement Methods (1)

Method	Benefits	Drawbacks	Suggested Uses	Estimated Throughput
CPU Programmed I/O	• Simple software • Fewest PL resources • Simple PL slaves	• Lowest throughput	• Control functions	<25 MB/s
PS DMAC	• Fewest PL resources • Medium throughput • Multiple channels • Simple PL slaves	• Somewhat complex DMA programming	• Limited PL resource DMAs	600 MB/s
PL AXI_HP DMA	• Highest throughput • Multiple interfaces • Command/data FIFOs	• OCM/DDR access only • More complex PL master design	• High-performance DMA for large datasets	1200 MB/s (per interface)

Comparing Data Movement Methods (2)

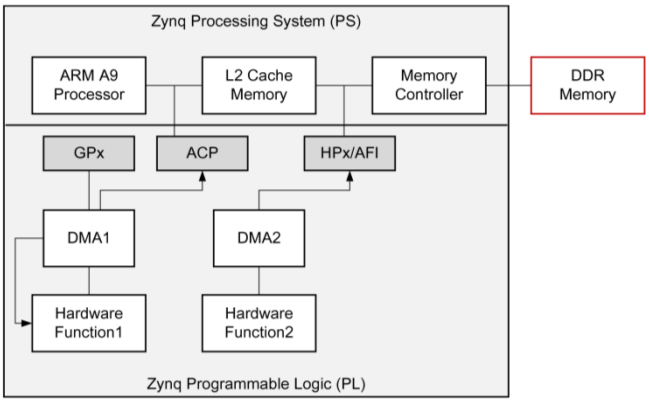
Method	Benefits	Drawbacks	Suggested Uses	Estimated Throughput
PL AXI_ACP DMA	• Highest throughput • Lowest latency • Optional cache coherency	• Large burst might cause cache thrashing • Shared CPU interconnect bandwidth • More complex PL master design	• High-performance DMA for smaller, Coherent datasets • Medium granularity CPU offload	1200 MB/s
PL AXI_GP DMA	• Medium throughput	• More complex PL master design	• PL to PS control functions • PS I/O peripheral access	600 MB/s

Outline

- Architectural Support for Data Motion
- *Data Motion*
- Optimization
- Summary
- Lab2 Intro

Data Movement in Zynq

- Data may move between master/slave IP located in PL and memories located internal to the processor or external to the chip using the nine ports described in the previous section
- The decision is made by the SDSoC compiler after analyzing the software or following user's directives



How SDSoC Compiler Maps Programs to HW/SW?

- **User specifies a program, a platform and a set of functions to map to hardware**
  - This defines the SDSoC-generated hardware / software interface
  - Program may also call other software functions provided by the platform, e.g., “write LED”
- **The SDSoC system compiler analyzes the program and maps it into a hardware / software system**
  - Preserves program semantics, including calls to hardware functions
  - Hardware functions can run concurrently with well-defined synchronization
- **Hardware function calls define “connections” implemented using…**
  - Program properties: data flow between calls to hardware functions, memory allocation,…
  - Function argument properties: payload size, hardware interfaces,…
  - Function properties: memory access patterns, implementation latency,…

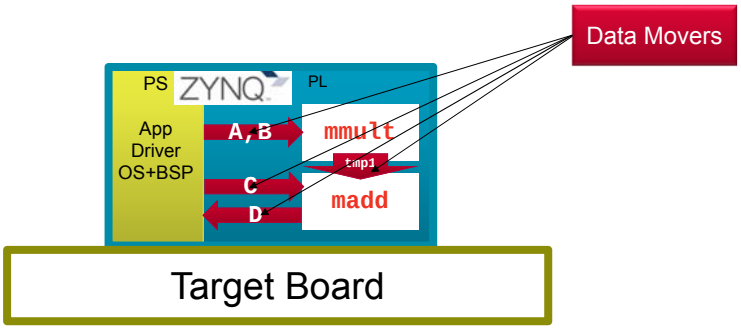
How SDSoC Compiler Maps Programs to HW/SW? (2)

- **Program source code**

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          madd(tmp1, C, D);

          mmult_sw(A, B, tmp2);
          madd_sw(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```

How SDSoC Compiler Maps Programs to HW/SW? (3)

- Structure of generated hardware system



How SDSoC Compiler Maps Programs to HW/SW? (4)

- Structure of generated software

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     std::cout << "Testing mmult .." << std::endl;

     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          _p0_mmult_0(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          _p0_madd_0(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }

     return true;
}
```

```
void _p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000000;
  start_seq[1] = 0x00010000;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_mmult_0
  cf_send_i(&(_p0_swinst_mmult_0.cmd_mmult),
            start_seq, 3*sizeof(int), &_p0_swinst_mmult_0_cmd);
  cf_wait(_p0_swinst_mmult_0_cmd);
```

Control transfer

```
mult_0.in_A), in_A, 1024 * 4, &_p0_request_2);
mult_0.in_B), in_B, 1024 * 4, &_p0_request_3);
```

Data transfers

```
void _p0_madd_0(float A[1024], float B[1024], float C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000003;
  start_seq[1] = 0x00010001;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_madd_0_cmd;
  cf_send_i(&(_p0_swinst_madd_0.cmd_madd), start_seq, 3*sizeof(int),
            &_p0_swinst_madd_0_cmd);
  cf_wait(_p0_swinst_madd_0_cmd);

  cf_send_i(&(_p0_swinst_madd_0.B_PORTA), B, 1024 * 4, &_p0_request_0);
  cf_receive_i(&(_p0_swinst_madd_0.C_PORTA), C, 1024 * 4,
               &_p0_madd_0_num_C_PORTA, &_p0_request_1);
  cf_wait(_p0_request_0);
  cf_wait(_p0_request_1);
  cf_wait(_p0_request_2);
  cf_wait(_p0_request_3);
}
```

Control transfer

Data transfers

Outline

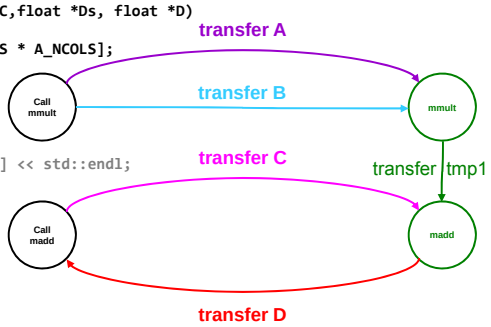
- Architectural Support for Data Motion
- Data Motion
- ***Optimization***
- Summary
- Lab2 Intro

System Level Optimization Using SDSoC

- Program connectivity and implementation
- Datamovers
- Memory models
- Platform Port Selection
- Direct connections
- Hardware function interface

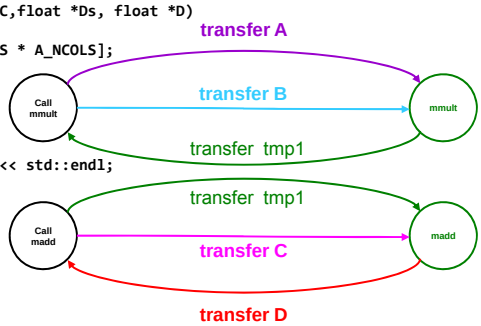
Program Connectivity

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          madd(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```



Program Connectivity (2)

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          madd(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```



- **Simple program changes can cause significant system changes!**

Program Connectivity Implementation

- **Every connection with data movement between the software program and a hardware function requires a *datamover***
- **Program connections can have multiple possible implementations**
  - SDSoC compiler infers datamovers and system ports
  - Datamover and system ports can be overridden with pragma
- **Datamover inference is based on payload size, payload memory attributes, and hardware function argument access pattern**
  - `#pragma SDS data data_mover(arg:<DM_TYPE>) // to override`
- **Note: `#pragma SDS` is always treated as a rule, not a hint**
  - Not all errors will be caught, so care is required

Datamovers

- **Every datamover has two components**
  - Hardware IP, e.g., axi_dma, a hardware function or a CPU
  - OS-specific software library function (userspace for Linux)

Datamover	IP	IP port types	Payload (bytes)	Phys. memory contiguity
axilite	ps7	register, axilite		either
axi_dma_simple	axi_dma	bram,ap_fifo, axis	< 8M	contiguous
axi_dma_sg	axi_dma	bram,ap_fifo, axis		either
axi_dma_2d	axi_dma	bram		contiguous
axi_fifo	axi_fifo_mm_s	bram,ap_fifo, axis	≤ 300	non-contiguous
zero_copy	HW fcn IP	m_aximm		contiguous

Memory Models

- **sds_lib.h functions to "declare" memory properties**
  - `sds_alloc(size_t size);` `// guaranteed contiguity`
  - `sds_mmap(void *paddr, size_t size, void *vaddr);` `// assumed contiguity`
  - `sds_register_dmabuf(void *vaddr, int fd);` `// assumed contiguity`
- **`#pragma SDS data copy (A[offset:array_size])`**
  - Copy in, copy out semantics (e.g., axi_dma, axi_fifo)
  - Also used to specify variable and non-default transfer sizes
- **`#pragma SDS zero_copy(A[offset:array_size])`**
  - Shared memory semantics (hardware function m_aximm only)
- **`#pragma SDS data access_pattern (A:SEQUENTIAL|RANDOM)`**
  - Only valid for "copy" semantics
  - SDSoC generates FIFO or BRAM hardware interface accordingly

Platform Port Selection

- **Compiler infers ACP or HP (AFI) ports based on memory attributes and other criteria**
  - `#pragma SDS data sys_port (A:ACP|AFI|MIG)` to override inference rules
    - A must be one of the formal arguments of the function
- **Memory is by default assumed CACHEABLE, i.e., compiler maintains cache coherency between CPU and hardware function**
  - Cache flush before transferring data to hardware function
  - Cache invalidate before transferring data from hardware function

Direct Hardware Connections

- Compiler maps data movement between hardware functions onto direct connections when possible
- Direct connections implemented entirely in hardware via AXI streams

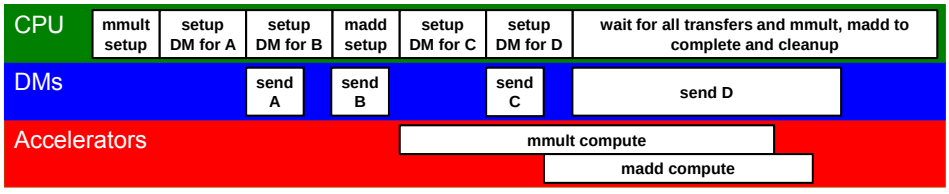
Port IF	BRAM	FIFO	AXIS
BRAM	Y	Y	64-bit TDATA
FIFO	Y	Y	64-bit TDATA
AXIS	64-bit TDATA	64-bit TDATA	Equal TDATA width

- User must guarantee sufficient buffering
  - `#pragma SDS data buffer_depth(A:<buffer_depth>)`

Interface	Type	Depth	Array Size	Data width	Default
BRAM	multi-buffer	1:4	2:16384	8:16:32:64	1
FIFO	buffer	2:(16384/array_size)	2:16384	8:16:32:64	1024

Hardware Function Call Sequencing

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
     float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

     for (int i = 0; i < NUM_TESTS; i++) {
          mmultadd_init(A, B, C, Ds, D);

          mmult(A, B, tmp1);
          //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
          madd(tmp1, C, D);

          mmult_golden(A, B, tmp2);
          madd_golden(tmp2, C, Ds);

          if (!mmult_result_check(D, Ds))
               return false;
     }
     return true;
}
```



Hardware Function Call Interface

- SDSoC rewrites calls to hardware functions and replaces function by a stub to transfer data and control

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
    std::cout << "Testing mmult .." << std::endl;

    float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

    for (int i = 0; i < NUM_TESTS; i++) {
        mmultadd_init(A, B, C, Ds, D);

        _p0_mmult_0(A, B, tmp1);
        //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
        _p0_madd_0(tmp1, C, D);

        mmult_golden(A, B, tmp2);
        madd_golden(tmp2, C, Ds);

        if (!mmult_result_check(D, Ds))
            return false;
    }

    return true;
}
```

In _sds/swstubs directory

```
void _p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000000;
  start_seq[1] = 0x00010000;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_mmult_0_cmd;
  cf_send_i(&(_p0_swinst_mmult_0.cmd_mmult),
            start_seq, 3*sizeof(int), &_p0_swinst_mmult_0_cmd);
  cf_wait(_p0_swinst_mmult_0_cmd);

  cf_send_i(&(_p0_swinst_mmult_0.in_A), in_A, 1024 * 4, &_p0_request_2);
  cf_send_i(&(_p0_swinst_mmult_0.in_B), in_B, 1024 * 4, &_p0_request_3);

}
```

Control transfer

Data transfers

Hardware Function Call Interface (2)

- `cf_wait()` synchronization barriers automatically inserted at the end of hardware function pipeline

```
bool mmultadd_test(float *A,  float *B, float *C,float *Ds, float *D)
{
    std::cout << "Testing mmult .." << std::endl;

    float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];

    for (int i = 0; i < NUM_TESTS; i++) {
        mmultadd_init(A, B, C, Ds, D);

        _p0_mmult_0(A, B, tmp1);
        //std::cout << "tmp1[0] = " << tmp1[0] << std::endl;
        _p0_madd_0(tmp1, C, D);
```

```
void _p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000000;
  start_seq[1] = 0x00010000;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_mmult_0_cmd;
  cf_send_i(&(_p0_swinst_mmult_0.cmd_mmult),
            start_seq, 3*sizeof(int), &_p0_swinst_mmult_0_cmd);
  cf_wait(_p0_swinst_mmult_0_cmd);

  cf_send_i(&(_p0_swinst_mmult_0.in_A), in_A, 1024 * 4, &_p0_request_2);
  cf_send_i(&(_p0_swinst_mmult_0.in_B), in_B, 1024 * 4, &_p0_request_3);

}
```

```
void _p0_madd_0(float A[1024], float B[1024], float C[1024])
{
  switch_to_next_partition(0);
  int start_seq[3];
  start_seq[0] = 0x00000003;
  start_seq[1] = 0x00010001;
  start_seq[2] = 0x00020000;
  cf_request_handle_t _p0_swinst_madd_0_cmd;
  cf_send_i(&(_p0_swinst_madd_0.cmd_madd), start_seq, 3*sizeof(int),
            &_p0_swinst_madd_0_cmd);
  cf_wait(_p0_swinst_madd_0_cmd);

  cf_send_i(&(_p0_swinst_madd_0.B_PORTA), B, 1024 * 4, &_p0_request_0);
  cf_receive_i(&(_p0_swinst_madd_0.C_PORTA), C, 1024 * 4,
               &_p0_madd_0_num_C_PORTA, &_p0_request_1);
  cf_wait(_p0_request_0);
  cf_wait(_p0_request_1);
  cf_wait(_p0_request_2);
  cf_wait(_p0_request_3);
}
```

`cf_wait()` for madd

`cf_wait()` for mmult

Outline

- Architectural Support for Data Motion
- Data Motion
- Optimization
- ***Summary***
- Lab2 Intro

Summary

- **High-performance AXI port connects the accelerator to DDR/OCM or SCU for data moving**
  - Internal routing and port select can provide low-latency paths between the accelerator and memory
- **Caches provide both processors and accelerator with coherent data**
  - Caches can be locked and flushed as needed for the accelerator
- **Processor provides clock sources to accelerators**
- **Compiler rewrites calls to hardware functions and replaces function by a stub to transfer data and control**
- **There are three functions available to exchange data and synchronize events**
  - `cf_send`
  - `cf_receive`
  - `cf_wait`

Outline

- Architectural Support for Data Motion
- Data Motion
- Optimization
- Summary
- ***Lab2 Intro***

Lab2 Intro

- **Introduction**
  - This lab guides you through the process of handling data movements between the software and hardware accelerators using various pragmas and functions
- **Objectives**
  - Use pragmas to select ACP or AFI ports for data transfer
  - Use pragmas to select different data movers for your hardware function arguments
  - Understand the use of sds_alloc() and sds_free() calls
  - Understand the use of malloc() and free() calls
  - Analyze built hardware

Coding Considerations

Zynq
SDSoC 2015.4 Version

Objectives

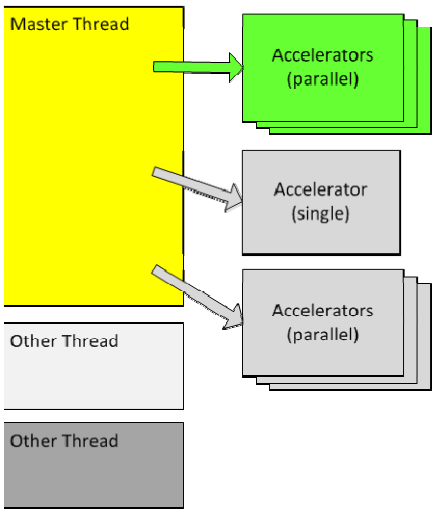
- **After completing this module, you will be able to:**
  - Identify proper coding styles
  - Use high-level techniques to find potential problem areas
  - Avoid common SDSoC tool-related issues

Outline

- ***Coding Style***
- **Development Technique**
- **SDSoC Tool-Related Issues**
- **Summary**

General C/C++ Guidelines

- **Hardware functions can execute concurrently under the control of a single master thread**
  - A program can have multiple threads and processes
  - Concurrently means that there are multiple copies of the accelerator (parallelism)
- **No support for exception handling in hardware functions**
  - Cannot *break* from accelerator
  - Global variables are synthesizable but it is not recommended to use them heavily
    - Potential issues with concurrency



Hardware / Accelerator Guidelines

- **A top-level hardware function must be a global function, not a class method, nor can it be overloaded**
  - Must have at least one argument resolvable to a C99 basic arithmetic type
    - Pointers, arrays, and structures flatten to a C99 basic arithmetic type and are legal
      - Arrays of compound types do not flatten and are not supported as arguments
    - Scalar arguments must fit in a 32-bit container
- **Return type must be a scalar that fits in a 32-bit container**
  - Returns a single value, not an array or structure
    - char, short, int all fit in a 32-bit container

Memory Allocation

- **Both pass-by-value and pass-by-reference are supported as arguments**
- **Array arguments are always pass-by-reference**
  - Only the address to the first element in the array is passed on the stack
- **Locally defined arrays are also stored on the stack**
  - The array can be too large for the stack resulting in a fault
- **Solutions: dynamically allocate the array OR define array in global memory**
  - Dynamic allocation achieved using malloc(), which generates a *virtually* contiguous block
    - But accelerators need a *physically* contiguous block of memory
  - Use `sds_alloc()` instead of `malloc()`
  - Use `sds_free()` to free the allocated memory
  - Use of global memory for a hard function generates an error

Block RAM Utilization

- **If a hardware function is declared with the array size in the declaration**
  - Appears to be "pass-by-value" in traditional coding; however, has special meaning in the SDSoC development environment
  - Indicates that the tools should store the array in block RAM
  - Problem: array can easily overflow block RAM and cause an error during the build process

```
int rgb_2_gray(uint32_t color[2073600], uint8_t gray[2073600]);
...
ERROR: [SDSoC 0-0] Function 'rgb_2_gray' argument 'color' mapped to
a bram interface has invalid array size (2073600), which must be in
the range [1..16384]
ERROR: [SDSoC 0-0] Function 'rgb_2_gray' argument 'gray' mapped to a
bram interface has invalid array size (2073600), which must be in the
range [1..16384]
```

- **Possible solution: consider streaming from dynamically allocated DDR memory**

Tool Implementation Choices Based on Buffer Sizes

- **The SDSoC development environment will attempt to choose optimal parameters for accelerators**
  - May result in unforeseen consequences when developing with a software-oriented mindset
- **Example of developing software**
  - Normal to use smaller buffers for testing and development purposes
    - However, changing the size of buffers modifies the pathways used for communication
    - Impacts choice of transport mechanism; results in different system performance and configuration

Instrumenting the Code

- **Surround the accelerated code with timers**
- **Allows software timing of the accelerator execution**
- **Instrumenting individual functions within a pipe *breaks* the flow**
  - Forces access to main memory
- **Note: The #ifdef code in the image is selectively disabled in the image**

```
    /* process image */
    sw_sds_clk_start(WHOLE_PROCESS);
    for (i = LOOPS; i != 0; i--) {

#ifdef TIME_RGB2GRAY
   sw_sds_clk_start(RGB2GRAY);
#endif
        rgb_2_gray(array_c, array_g_1);
#ifdef TIME_RGB2GRAY
        sw_sds_clk_stop(RGB2GRAY);
#endif
#ifdef TIME_SHARPEN
   sw_sds_clk_start(SHARPEN);
#endif
        sharpen_filter(array_g_1, array_g_2);
#ifdef TIME_SHARPEN
        sw_sds_clk_stop(SHARPEN);
#endif
#ifdef TIME_EDGE_DETECT
   sw_sds_clk_start(EDGE_DETECT);
#endif
        sobel_filter(array_g_2, array_g_3);
#ifdef TIME_EDGE_DETECT
        sw_sds_clk_stop(EDGE_DETECT);
#endif

        printf(".\n");
    }
    sw_sds_clk_stop(WHOLE_PROCESS);
```


Conditional Compilation

- **Predefined compiler marcos allow guard code with *#ifdef* and *#ifndef* preprocessor statements**
  - __SDSCC__ , __SDS++__ defined to compile source files with sdscc/sdscc++ compiler; used to guard code when compiled with other compilers (for example GCC)
  - __SDSVHLS__ macro is defined to be used to guard code depending on whether high-level synthesis is run or not

```
#ifdef __SDSCC__
#include <stdlib.h>
#include "sds_lib.h"
#define malloc(x) (sds_alloc(x))
#define free(x) (sds_free(x))
#endif
```

```
#ifdef __SDSVHLS__
void mmult(ap_axiu<32,1,1,1> A[A_NROWS*A_NCOLS],
ap_axiu<32,1,1,1> B[A_NCOLS*B_NCOLS],
ap_axiu<32,1,1,1> C[A_NROWS*B_NCOLS])
#else
void mmult(float A[A_NROWS*A_NCOLS],
float B[A_NCOLS*B_NCOLS],
float C[A_NROWS*B_NCOLS])
#endif
```

Outline

- Coding Style
- ***Development Technique***
- SDSoC Tool-Related Issues
- Summary

Basic Compile

- **Ensure the project compiles correctly as a *software only* implementation**
  - Deselect all functions from porting to hardware
  - Compile the project under SDDebug configuration
    - Compiler report located in the console and in *.jou* and *.log* files
  - If there are any C/C++ issues, resolve them
  - SDSoC tool/HLS pragma have no effect on the software, only on implementation
- **Proceed to the next step when the code compiles correctly**

Verify Tool Operation for Dummy Hardware Function

- **Add and mark a "known good" function for acceleration**
  - Add and call a "known good" function to the software design
    - Available from the *samples* directory
    - Can use customer-built function
  - Mark the simple function for acceleration
  - Compile for SDEstimate
  - If the build fails
    - Review the console output for information
  - Check for any C syntax issues that are valid for C but invalid for SDSoC

```
int simple_function(uint32_t *
ptr, uint32_t n) {
    int index = 0;
    while (n--) { * (ptr + index++)
= index;    }
    return 0;
}
```

- **If it compiles, then the source code is syntactically acceptable for the tools "under the hood" and you can proceed**

Verify Synthesizability of Individual Functions

- **Mark functions for acceleration one at a time**
  - Mark a single high-level function for acceleration
  - Compile for SDEstimate and review results
    - Review the console output/report
    - Any issues are most likely due to the accelerator code
    - Comment out any pragma for this accelerator and rebuild for SDEstimate
  - Selectively add back in pragmas
    - Improper pragma syntax can be difficult to debug and will cause issues
- **When done with each accelerated function**
  - Remove validated function from hardware
  - Repeat process for all other accelerated functions
  - Alternately, running the integration step may also be a good choice depending on the situation

Integration – Putting all together

- **Mark multiple functions for hardware acceleration**
  - Mark the desired functions for acceleration
    - Use SDEstimate initially
    - Use SDRelease once everything is in place
  - If it does not build
    - Is it exceeding capacity of PL?
    - Are there port/bandwidth conflicts?
    - Review console output/logs
    - Incompatible #pragma being defined

Outline

- **Coding Style**
- **Development Technique**
- ***SDSoC Tool-Related Issues***
- **Summary**

Diagnosing SDEstimate Launch Failure

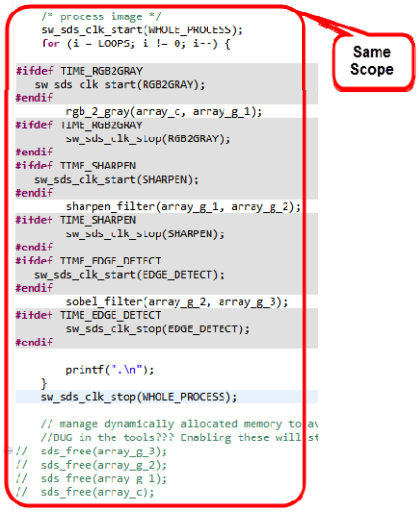
- **If performance estimation in the SDSoC development environment fails to generate a full report**
  - Ensure the development board is powered on and connected correctly
  - Open the Debug perspective
  - Disconnect and remove all debug sessions that are listed
  - Open the SDSoC perspective
  - Open the Project Overview (double-click project.sdsoc)
  - Click Estimate Performance

Error Feedback

- **Coding errors are visible in the editor window and marked with a symbol (?) on the line where the error was discovered**
  - Symbol can easily be mistaken for a warning symbol
- **When you are building for SDEstimate with no accelerated functions, console output shows**
  - Location of the errors
  - Source file
  - Line number
  - Can be used to backtrack the software only issues
- **When you are building for SDEStimate with accelerated functions, console output shows that**
  - An error is present
  - There is a log file/files
  - It is up to developer to track down the errors

sds_free() Breaking Optimizations

- **`sds_free()` cannot be used in the same scope as multiple accelerators**
  - Incorrectly interpreted by the SDSoC tools as access to the buffer by the PS between PL functions
  - Breaks pipelining optimization
  - Bug in SDSoC tools
- **Workaround**
  - Create a function that uses `sds_free()` to free the memory
  - Pass a pointer to the buffer into that function




```
    /* process image */
    sw_sds_clk_start(WHOLE_PROCESS);
    for (i = LOOPS; i != 0; i--) {
#ifdef TIME_RGB2GRAY
    sw_sds_clk_start(RGB2GRAY);
#endif
        rgb_2_gray(array_c, array_g_1);
#ifdef TIME_RGB2GRAY
        sw_sds_clk_stop(RGB2GRAY);
#endif
#ifdef TIME_SHARPEN
    sw_sds_clk_start(SHARPEN);
#endif
        sharpen_filter(array_g_1, array_g_2);
#ifdef TIME_SHARPEN
        sw_sds_clk_stop(SHARPEN);
#endif
#ifdef TIME_EDGE_DETECT
    sw_sds_clk_start(EDGE_DETECT);
#endif
        sobel_filter(array_g_2, array_g_3);
#ifdef TIME_EDGE_DETECT
        sw_sds_clk_stop(EDGE_DETECT);
#endif

        printf(".\n");
    }
    sw_sds_clk_stop(WHOLE_PROCESS);

    // manage dynamically allocated memory to av
    //DUG in the tools??? Enabling these will st
//  sds_free(array_g_3);
//  sds_free(array_g_2);
//  sds_free(array_g_1);
//  sds_free(array_c);
```


Unsupported Memory Access and System Hangs

- **Error: "unsupported memory access on variable 'color' which is (or contains) an array with unknown size at compile time“**
  - The tools do not know how much data is transferred for the variable *color*
- **Solution: Ensure that the quantity of memory to be transported is declared above the function prototype**
  - Incorrect values will cause system hangs
    - The PS or PL are continually waiting for more data that will never arrive
  - Use the pragma parameter buffer_depth above the function prototype
    - Defines the size of the buffers

```
/* Declaring the full 1920 * 1080 buffer size for the rgb_2_gray conversion
*/
#pragma sds data buffer_depth(color:2073600, gray:2073600)
int rgb_2_gray(uint32_t *color, uint8_t *gray);
```

SDSoC Tool Reset Procedure

- **The following assumes that the source code is correct yet the tools fail to build the project or otherwise behaves unexpectedly**
- **To 'reset' the workspace**
  - Clean the project
  - Delete the SDDebug, SDEstimate and SDRelease folders (if they exist)
  - Remove all functions from hardware acceleration
  - Close the SDSoC tool
  - Open the SDSoC tool
- **To 'hard reset' the workspace**
  - Ensure you have your latest source code backed up
  - Delete the project from the SDSoC tool
  - Close the SDSoC tool
  - Find and delete all of the contents of the workspace from the system (Windows or Linux)
  - Start the SDSoC tool
  - Choose a workspace
  - Create a new project
  - Import your backed up source code

Outline

Summary

- **Use the most appropriate coding style**
- **Adopt an incremental approach**
  - Software first
  - Mark functions for hardware one by one
  - Integrate accelerators at the end
- **Pay close attention to errors during the build process**
  - Especially when modifying accelerated functions
- **Avoid triggering common bugs in the SDSoC tool**

Profiling

Zynq
SDSoC 2015.4 Version

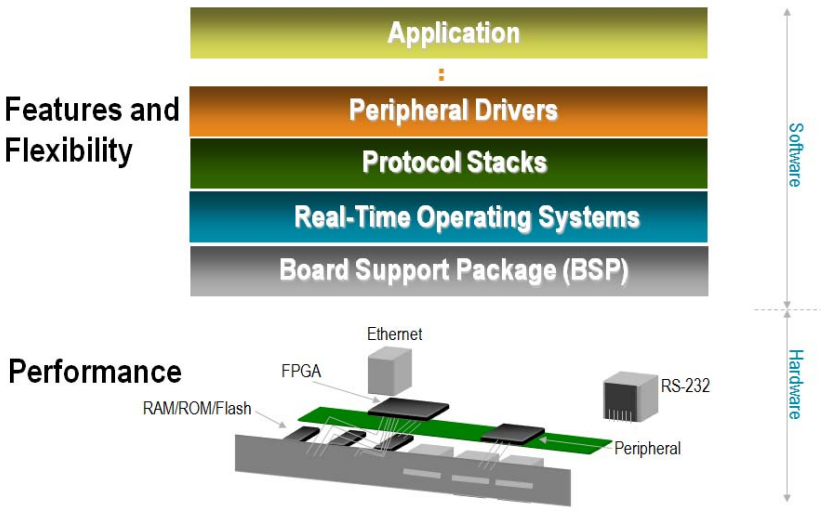
Objectives

- **After completing this module, you will be able to:**
  - Describe what profiling is and how it works
  - Evaluate profiling results for software efficiency
  - Discuss software tradeoffs to hardware
  - List various profiling methods
  - Describe the differences between Standalone versus Linux application profiling

Outline

- *Introduction*
- Profiling in SDSoC
- Performance Improvement
- Summary
- Lab3 Intro

Embedded Systems



Hardware and Software Partitioning

- **Determine the software "critical path" by profiling**
  - Profiling measures where the CPU is spending its cycles on a function-by-function or task-by-task basis
  - Similar to timing analysis in hardware
  - Informs the system designer which software routine may be a candidate to hardware-accelerate
- **Functions can be rewritten to improve efficiency in a number of ways**
  - Implementation in assembly code rather than C
  - Writing faster C code, for example limit pointer use

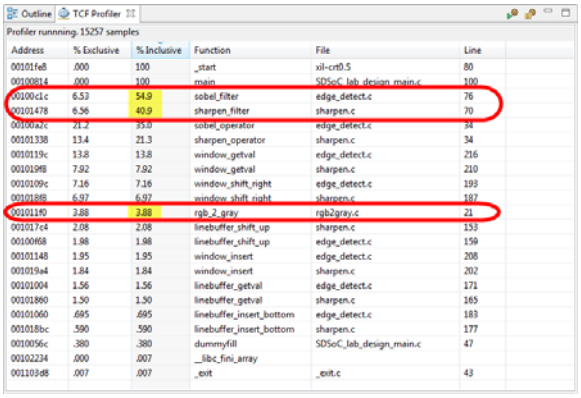
Outline

What is Profiling?

- **Profiling is an analysis of software performance**
  - Where routine time is being spent
  - How many times functions are being called
  - Which algorithms to consider moving to hardware
- **Four methods**
  - TCF profiling
    - TCF profiler is the preferred profiling tool and is included with the SDSoC tool
  - gprof profiling
  - PS performance monitoring
  - Manual profiling
    - It can be performed to get absolute time spent in a function by modifying the code and collecting information from a free-running timer

TCF Profiling

- **Non-invasive, uses ARM's PMU (CoreSight)**
- **TCF profiling provides relative time spent in a function as a percentage**
  - Inclusive and exclusive percentages included
- **No compiler flags need to be set**
- **Profiling output can be sorted by**
  - Code address
  - % exclusive, exclusive time
    - Only time spent in the the function is included (excluding all children function calls)
  - % inclusive, inclusive time
    - Include time spent in the function and spent in any functions called within
  - Function name and file name

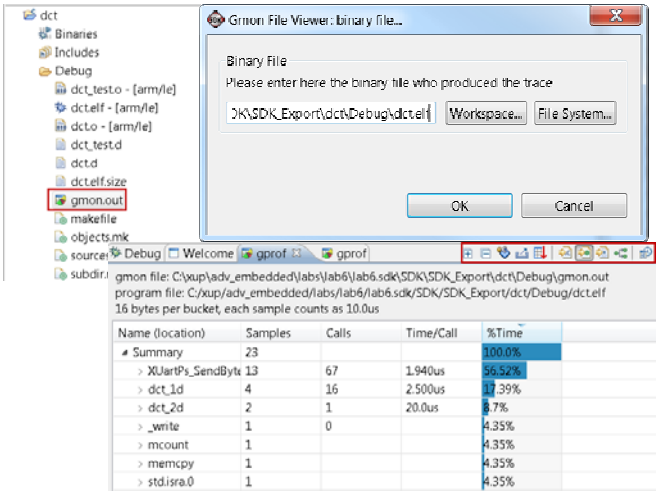


gprof Profiling

- **Profiling using gprof utility is intrusive**
  - Requires hardware timer and dedicated profile RAM area
  - Executable is modified with profiler routines
  - Processor is interrupted at fixed intervals to determine program counter
- **Requires building with -pg profiling configuration and enabling profiling in run configuration**
- **Once program has finished running, *gmon.out* file is generated; double-clicking its entry will launch gprof utility to visualize data**
  - Sort samples per file, function, or line
  - Display function call graph
  - Switch sample/time

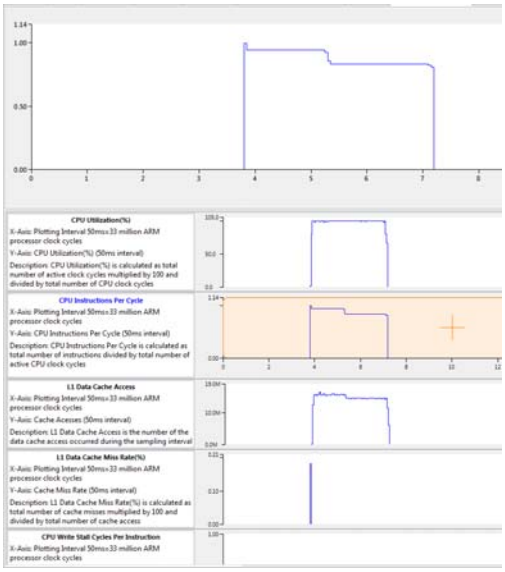
Viewing Profiling Reports: Launching *gprof*

- **Double-click *gmon.out* to launch *gprof***
- **Point to executable ELF; usually selected by default**
- ***gprof* report launches**
- **Report toolbar control report options and view capabilities**
  - Sort samples per file
  - Sort samples per function
  - Sort samples per line
  - Display function call graph
  - Switch sample/time



PS Performance Monitoring

- **The ARM PMU can provide information about APU performance**
  - Utilization percentage and instructions per cycle of either CPU
  - L1 data cache miss rate
  - CPU read/write stall cycles per instruction
- **Cumulative data can be viewed in the APU Performance view; visualization with respect to time is done using the PS Performance view in the SDSoc**



PL Performance Monitoring

- **AXI performance monitor can be added**
  - Analyze AXI interface traffic
    - Overburdened interface
    - Starved interface
- **Collected data includes**
  - Write byte count
  - Read byte count
  - Write transaction count
  - Total write latency
  - Read transaction count
  - Total read latency
- **Data can be viewed in the APU Performance view (cumulative reading) or in the PL monitoring view (data with respect to time)**

Manual Profiling

- The `sds_lib` library included in SDSoC provides a simple, source code annotation based time-stamping API that can be used to measure application performance

```
unsigned long long sds_clock_counter(void);
```

- Using this API to collect timestamps and differences between them, you can determine duration of key parts of your program

```
#include "sds_lib.h"
unsigned long long total_run_time = 0;
unsigned int num_calls = 0;
unsigned long long count_val = 0;
#define sds_clk_start(){ \
count_val = sds_clock_counter(); \
num_calls++; \
}
#define sds_clk_stop() { \
long long tmp = sds_clock_counter(); \
total_run_time += (tmp - count_val); \
}
#define avg_cpu_cycles()(total_run_time / num_calls)
sds_clock_start();
f();
sds_clock_stop();
```


Linux Application Profiling

- **Profiling Linux user applications requires a TCF agent to be running on the Linux system**
- **User application profiling is based on TCF profiler**
  - TCF profiler provides the same information as when profiling standalone applications
- **Profiling all running code: user applications, kernel, interrupt handlers, and other modules**
  - SDSoC tool includes an OProfile plug-in
    - Supports visualization of its call profiling capabilities
    - OProfile is an open-source, system-wide profiler for Linux
    - Requires a kernel driver and daemon to collect sample data

Coding Style Can Impact Profiling

- **Effective profiling is based on how much time is spent in functions, and how often they are called**
  - If your code is just a fall-through main, profiling is not useful because 100 percent of execution time will be in *main( )* with no calls to other functions
  - Carefully architect the application with a structured architecture by using functions
  - Complier does not consider *macros* as functions – the macro will be expanded and treated as in-line code
  - Separate algorithms logically into functions that will help you analyze the flat profile view
  - Think ahead when architecting code—Is this algorithm a candidate for implementing in programmable logic?
- **Coding in hardware**
  - Top-level function must include all hardware function calls

Outline

Task Implementation Decision

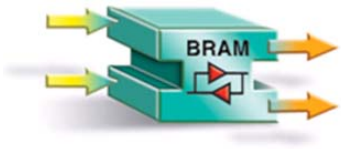
- **Keep it in software**
  - Not in critical path
  - Enough "free" cycles
  - Easier to code in software than in hardware
    - Uses math library functions
  - NEON co-processor
    - Supports integer vector operations
    - Single floating-point operations
- **Move to hardware**
  - Programmable logic co-processor
    - Customized to user's needs
    - Excellent for iterative and pipelined processing
  - Add soft core processor in PL
    - Both Cortex-A9 and MicroBlaze processors can co-exist in the AP SoC

Software to Programmable Logic

- **Slow software tasks can be accelerated by taking them to hardware**
  - Start with functions where the software spends most of its time
  - Consider the hardware implementation and if there is potential benefit implementing in hardware
- **Many mechanisms**
  - Dual-port block RAM
  - Custom AXI peripheral
  - Code optimization: use of macros, increasing compiler optimization
  - Enabling caching if (by default) it is turned OFF

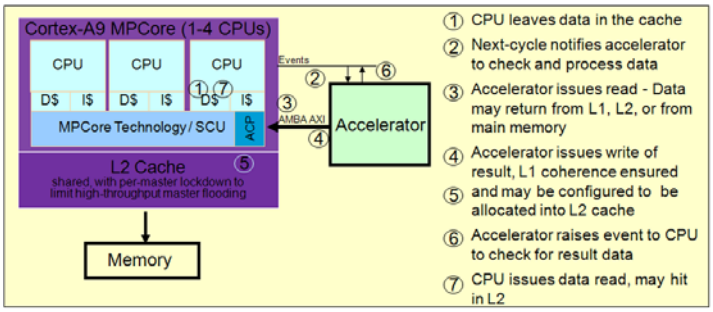
Using Block RAM

- **Leverage the dual-port nature of Xilinx block RAM**
- **Useful for data in block or frame format**
  - Video
  - 2D matrix maps
- **Advantages**
  - Low silicon overhead
  - Fast and deterministic latency



Enhanced Accelerator SoC Integration

- **ARM MPCore: accelerator coherence port (ACP)**
  - Sharing benefits of the ARM MPCore optimized coherency design
  - Accelerators gain access to CPU cache hierarchy
  - Compatible with standard un-cached peripherals and accelerators



Outline

- Introduction
- Profiling in SDSoC
- Performance Improvement
- *Summary*
- Lab3 Intro

Summary

- **Profiling analyzes how CPU cycles are utilized and where software bottlenecks exist**
- **To remove bottleneck, functions can be**
  - Rewritten or moved to another software layer
  - Migrated to hardware
- **Four ways to profile an application**
  - TCF, gprof, performance monitoring, and manual
- **TCF profiler is the preferred tool for profiling in the SDSoC (over gprof)**
  - Uses ARM's integrated hardware to gather performance statistics
- **gprof profiling is an invasive operation—additional hardware and software are required**
- **Xilinx AXI performance monitor in PL can monitor traffic through the AXI interface**
  - Can identify overburdened or starved interface
- **Manual profiling requires code change and sd_lib usage**

Outline

- Introduction
- Profiling in SDSoC
- Performance Improvement
- Summary
- ***Lab3 Intro***

Lab3 Intro

- **Introduction**
  - Program hot-spots that are compute-intensive are good candidates for hardware acceleration, especially when it is possible to stream data between hardware and the CPU and memory to overlap the computation with the communication. This lab guides you through the process of profiling an application, analyzing the results, identifying function(s) for hardware implementation, and then profiling again after targeting function(s) for acceleration
- **Objectives**
  - Use TCF profiler to profile a pure software application
  - Use TCF profiler to profile a software application that calls functions ported to hardware
  - Use manual profiling method by using sds_lib API and counters

Estimation

Zynq
SDSoC 2015.4 Version

Objectives

- **After completing this module, you will be able to:**
  - Describe what estimation is in SDSoC and how it works
  - Distinguish between profiling and estimation
  - Evaluate estimation reports for hardware acceleration effectiveness

Outline

- *Introduction*
- Estimation in SDSoC
- Summary
- Lab4 Intro

What is Performance Estimation?

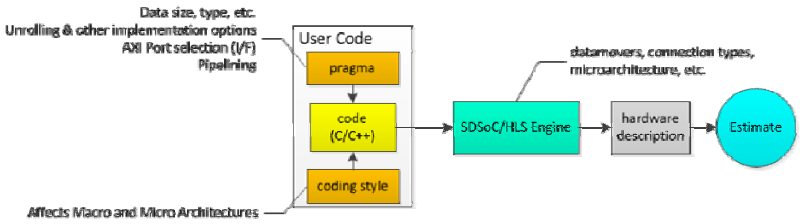
- **Performance estimation is a faster way to determine system speed up by targeting functions in hardware**
  - Quick turn around as it does not require full bitstream generation
  - Provides more design iterations to explore various accelerator optimizations
  - Uses SDEstimate configuration instead of SDRelease or SDDebug
    - SDRelease and SDDebug configurations require full bitstream generation which would be time consuming
- **Restrictions**
  - Performance estimation flow is not supported in the presence of asynchronous calls

Outline

- Introduction
- ***Estimation in SDSoC***
- Summary
- Lab4 Intro

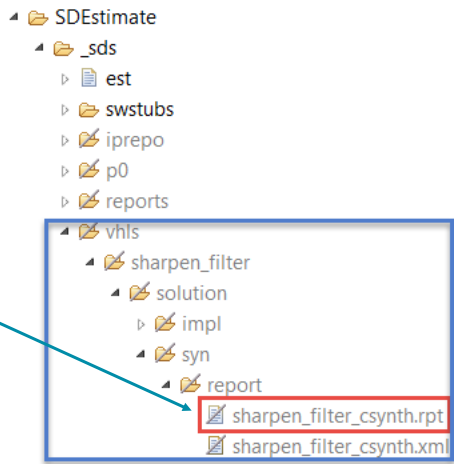
How Does the SDEstimation Flow Work?

- **SDSoC development environment uses HLS engine**
- **HLS engine converts selected C/C++ functions into hardware *fragments***
  - Each fragment has approximated performance and resource usage
- **Software performance determined by running on hardware**
  - Hardware performance estimated from HLS reports
- **Since a final bitstream (with accelerators) is not required, this is a fairly quick process**
  - Allows for many design iterations enabling designers to explore optimization choices



Under the Hood Estimation

- **Estimation performs a number of tasks**
  - Pulls the function marked for hardware from the code and replaces it with HW configuration functions
  - Passes software function to the Vivado HLS tool and compiles it
    - Each accelerator has its own project (defined by its name)
    - Vivado HLS tool results found under *SDEstimate > _sds > vhls*
    - Synthesis and implementation files are under each accelerator
    - Report showing resources and timing estimates found
- **Software performance and comparative performances measured on processor**
- **Resource and timing estimates reports provided in an easy-to-read report**



Scope of Overall Speedup

- **In the Performance Estimation Report's first line shows the estimated speedup for the top-level function (referred to as perf root)**
  - This function is set to "main" by default
  - However, there might be code that you would like to exclude from this comparison, for example allocating buffers, initialization and setup
  - If you wish to see the overall speedup when considering some other function, you can do this by specifying a different function as the root for performance estimation flow
  - The flow works with the assumption that all functions selected for hardware acceleration are children of the root

Performance Estimation Flow with Linux

- **Similar to Standalone OS based flow except**
  - Select Linux as the target OS while creating the project
- **Select the functions you want to target**
- **Set SDEstimate build configuration**
- **Build project**
- **Copy the contents of the `sd_card` folder under SDEstimate to an sd card and boot up the board**
  - Ensure that the board is connected to an Ethernet router using an Ethernet cable
  - Ensure that a serial terminal is connected
- **Power on the board and notice the leased IP address in the Linux boot log**

Performance Estimation Flow with Linux (2)

- **If you don't see the bootup assigning any ip address then do the following**
  - Execute `ifconfig` command in the Linux console and see if any ip address is assigned to eth0
  - If not assigned then execute the following command

    ```
    ifconfig eth0 192.168.0.10
    ```

  - Make sure that the PC has the ip address assigned as 192.168.0.1
- **In the SDSoC environment click on the Estimate Performance Speedup link**
- **Click on the *Click Here* link of the SDSoC Report Viewer**
- **Click on the New button and set up the IP address (192.168.0.10) and the port (1534)**
- **Click OK which will execute the application on the board and the speedup results will be displayed**

After Estimation

- **If you have achieved the desired performance**
  - Build the application for the SDRelease configuration
  - Copy the generated SD card image to an SD card if required
    - Required for Linux, optional for Standalone
  - Run the application on the board to obtain the actual performance of the generated system
- **If you have not achieved the desired performance**
  - Review the reports to identify what is slowing down
  - Use SDSoC tool and/or HLS pragmas to improve performance
  - Review your coding style
  - Leverage more efficient libraries if available
  - Repeat the above steps until you get what you need

Differences Between Profiling and SDEstimation

- **Profiling is used to determine where CPU spends most of its cycles and the order of functions being called**
  - Results identify software "bottlenecks“
  - It is not helpful for identifying accelerator performance improvement
    - Profiling generally does not show actual performance
- **Determining performance by instrumenting in the SDRelease/SDDebug configurations is time consuming**
  - Involves bitstream generation for complete hardware, including hardware accelerators
    - Time consuming task
    - Restricts design exploration
- **SDEstimation executes software on hardware**
  - However, for hardware accelerator performance estimate it uses HLS reports, eliminating actual need of generating full bitstream

Outline

- Introduction
- Estimation in SDSoC
- *Summary*
- Lab4 Intro

Summary

- **The performance estimation flow reports estimated performance improvement**
  - For *root* system
  - For individual hardware functions
- **SDSoC tool and Vivado HLS tool pragmas need to be used efficiently to improve accelerator performance**
- **SDSoC tool and HLS coding guidelines must be followed**
- **Performance estimation flow is not supported in the presence of asynchronous calls (using asynchronous pragmas, sds_wait, etc.)**
- **Profiling of an application which has accelerated functions requires full bitstream generation whereas the SDEstimation configuration uses HLS generated reports, reducing design exploration cycle time**

Outline

- Introduction
- Estimation in SDSoC
- Summary
- ***Lab4 Intro***

Lab4 Intro

- **Introduction**
  - This lab guides you through the steps involved in estimating the expected performance of an application when functions are targeted in hardware, without going through the entire build cycle
- **Objectives**
  - Use the SDSoC environment to obtain an estimate of the speedup that you can expect from your selection of functions to accelerate
  - Differentiate between the flows targeting Standalone OS and Linux OS

Debugging

Zynq
SDSoC 2015.4 Version

Objectives

- **After completing this module, you will be able to:**
  - Identify typical software debugging capabilities
  - List tools available for system debugging
  - Describe how the debugger communicates with hardware
  - Distinguish between hardware and software debugging
    - Identify IP which can be of assistance when debugging

Outline

- ***Introduction***
- **Hardware Debugging**
- **Software Debugging**
- **Summary**
- **Lab5 Intro**

Introduction

- **Debugging is an integral part of embedded systems development**
- **The debugging process is defined as testing, stabilizing, localizing, and correcting errors**
- **Two methods of debugging:**
  - Hardware debugging via a logic probe, logic analyzer, in-circuit emulator, or background debugger
  - Software debugging via a debugging instrument
    - A software debugging instrument is dedicated hardware and part of the silicon that is accessible via JTAG or dedicated part pins
    - Controls the processor as an intrusive debug unit that is disabled during normal operation
    - Some "hard" processors have this feature permanently available while "soft" processors may have this physically removed from the delivered product
- **Debugging types:**
  - Functional debugging
  - Performance debugging

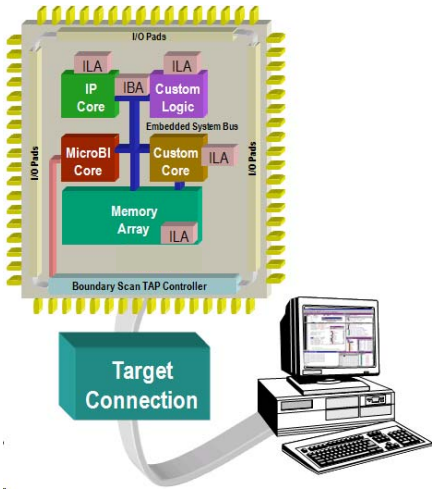
Xilinx Solution for Debugging Embedded Designs

- **Hardware debugging tool**
  - Vivado logic analyzer for hardware; functionally replaces expensive external logic analyzer
- **Software debugging tools**
  - System Debugger
    - SDK Debugging perspective and inexpensive JTAG cable replace ICE
- **Built-in, cross-probing trigger capability for hardware and software debug coherency**
  - Vivado logic analyzer invoked through Vivado
  - SDK Debugging perspective tool accessed from within SDK
- **A single JTAG connection can be used for**
  - Programming the programmable logic
  - Downloading application
  - Hardware and software debugging

Outline

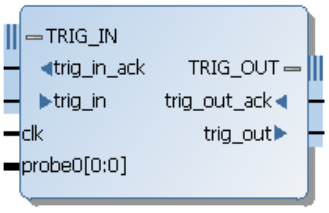
Vivado Logic Analyzer System

- **Vivado logic analyzer tool cores provide full internal visibility to all soft IP**
  - Access to hard IP ports
  - Accesses all the internal signals and nodes within the programmable logic (ILA)
  - Stimulus can be applied using the Virtual I/O core (VIO)
- **Debugging occurs at, or near, system speeds**
  - Debug on-chip using the system clock
- **Minimize pins needed for debugging**
  - Access via the JTAG interface



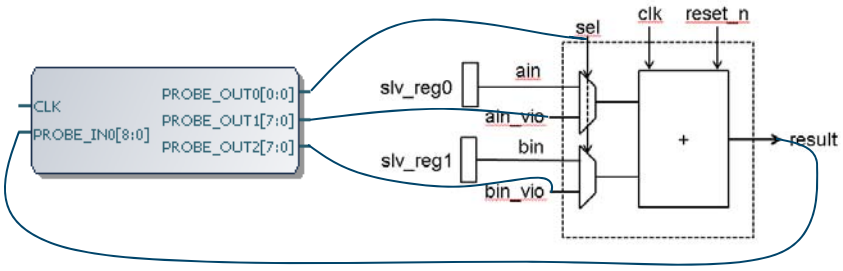
ILA Core

- **Used for monitoring internal programmable logic signals for post-analysis**
- **Multiple configurable ILA trigger units**
  - Configurable trigger input widths and match types for use with different input signals types
- **Up to 64 probes through GUI**
  - Up to 1024 probes through tcl command
- **Sequential triggering**
- **Storage qualification**
- **Configurable cross triggering**
  - Trigger in and Trigger out interfaces
- **Pre- and post-trigger buffering (capture data before, during, and after trigger condition is met)**



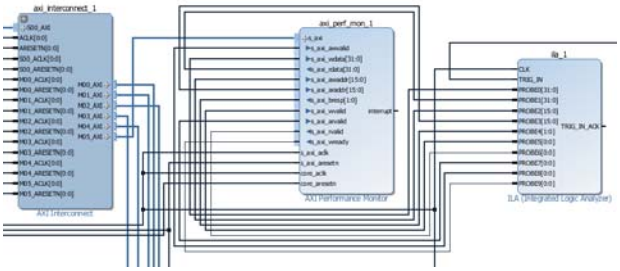
VIO Core

- **Support for monitoring and driving internal programmable logic signals in real time**
- **Probe input unit**
- **Probe output unit**



AXI Performance Monitor Core

- **Enables AXI system performance measurement for multiple slots**
  - AXI4 and AXI4-Stream
- **AXI Performance Monitor supports analyzing system behavior on AXI interfaces**
  - Event logging
    - Captures AXI events and external events
    - Time stamp between two successive events into streaming FIFO
  - Event counting
    - Measure events on AXI4/AXI4-Stream monitor slots or external event ports



Outline

- Introduction
- Hardware Debugging
- ***Software Debugging***
- Summary
- Lab5 Intro

Software Debugging

- **Debuggers must be able to perform a basic set of functions**
  - Regardless of processor or platform
    - Control of the flow of execution
    - Execute only the next instruction (single step)
    - Step into a function/step out of a function
    - Step over a function (treat the function as a single quantity)
    - Stop on a specific line (breakpoint/conditional breakpoint)
    - Resume execution until done or breakpoint reached
    - Stop when a global variable is read or written (watchpoint)
  - Ability to examine and modify memory/variable and registers
  - Directly modify the program counter (PC) and control processor execution location
  - Load the target platform with the code

Software Debugging Support in SDK

- **SDK supports software debugging in SDSoC via:**
  - System Debugger (GUI) or XSDB (command-line)
    - Software debugger runs on PC
  - TCF(Target Communication Framework) debugger over digilent cable for ARM
    - Open source
    - Supports system level debugging
    - Improved performance

Xilinx System Debugger

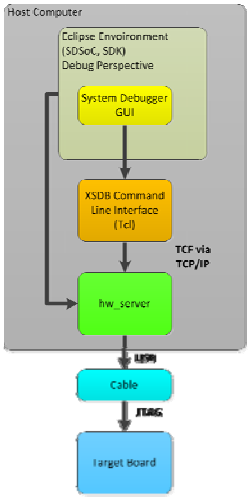
- **Xilinx System Debugger console (XSDB) replaces XMD, providing a variety of user debug services**
  - Connection between your workstation (host) and designs being debugged (targets)
    - `connect arm hw`
  - Processor identification
    - `ta`
    - Lists available processors in the hardware and their status (running, suspended)
  - Program processors
  - Bitstream downloading
    - `fpga` <filename>
  - Low-level debug commands
    - `mrd, mwr, rrd` …
  - General Tcl interface and command interpreter

Configure the Programmable Logic

- **Before debugging session can be launched, the target programmable logic must be configured**
  - PL would have logic analyzer core(s) for hardware debugging
  - SDSoC Debug configuration will automatically do this

Connections During Debugging Session

- **Each of the components connects seamlessly *under the hood***
- **Appears as if the user interface is directly connected to the hardware**
- **User can interact with the debugger graphically or textually**

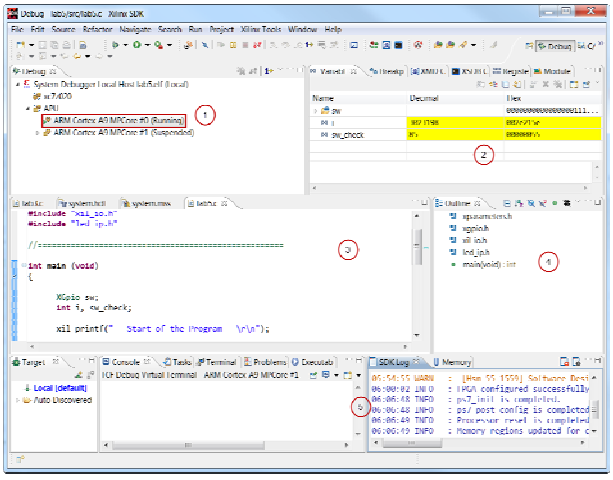


The TCF Agent and hw_server

- **The TCF agent is the program that performs host-to-target communication**
  - Act as server for XSDB and System Debugger
- **Linux TCF agent**
  - Program runs on target embedded Linux; requires IP address and Ethernet connection
- **hw_server**
  - Program runs on computer to which the target is attached; auto-detects targets
  - Launched automatically for local connections; manually for remote or custom connection
- **Environment detects local TCF connections and defined remote connections/services**

SDK Debug Perspective

- **Stack frame for target threads**
- **Variables, breakpoints, and registers views**
- **C/C++ editor**
- **Code outline**
  - Disassembly view can be added using Window > Show View > Disassembly
- **Console, SDK Log, and Memory views**



System Debugger

- **Source-level debugger GUI**
  - Runs through hw_server
  - Capable of remote debugging, as with XSDB, by connecting to remote hw_server instance
- **General process**
  - Start application
  - Set breakpoints/watchpoints
  - Examine state of registers and memory when breakpoint hit/application paused
  - Can change values in memory or registers
    - Allows user to correct for an error or create a difficult to achieve condition to test code
- **Only a single instance of System Debugger to be launched to debug multiple cores**
  - Independent control of each core
- **Supports C and C++**

Interacting With System Debugger

- **Code can be manually paused**
  - No guarantee where the PC is
  - Used when program is not behaving as expected and user wants to see where the code execution is occurring
- **Breakpoints/conditional breakpoints**
  - Set by user to cause the execution to stop at a specific line of C/C++ code
  - Conditional breakpoints also cause execution to stop at a location, but only when the conditions associated with that breakpoint are met
  - Useful when stopping when values are going out of bounds, or in the middle of a large loop
- **Once execution is suspended**
  - Memory, variables, and/or registers can be examined and modified
  - Control flow of execution
  - View assembly instructions related to the C/C++ source

Linux Debugging

- **Kernel debugging**
  - Linux kernel debugging through system debugger
- **Linux application debugging**
  - Linux application debugging using gdb
  - Linux application debugging using system debugger
    - PetaLinux comes with TCF agent
    - For other Linux distributions, need to include the TCF agent

Outline

Summary

- **Debugging is an integral part of embedded systems development**
- **Vivado and SDSoC SDK provides tools to facilitate hardware and software debugging**
  - Hardware debugging is done through using Vivado logic analyzer cores
  - Software debugging is performed using system debugger
- **SDSoC SDK provides environment, perspective, and underlying tools to enable seamless software debugging**
- **With software debugging and the Vivado logic analyzer supporting cross-probing enables simultaneous hardware and software debugging**
  - Use it to find and fix embedded system bugs faster
- **Full range of debugging capabilities is provided**
  - Control over execution using breakpoints, single-step, step-into, etc.
  - Visibility into various regions of memory and registers, including the ability to change values

Outline

- **Introduction**
- **Hardware Debugging**
- **Software Debugging**
- **Summary**
- ***Lab5 Intro***

Lab5 Intro

- **Introduction**
  - This lab guides you through the steps involved in debugging software application in SDSoC. SDSoC supports debugging both Standalone as well as Linux application. SDSoC also provides the Dump/Restore Data File feature which can be used to dump the memory snapshot on a disk and restore the memory content from a pre-defined file
- **Objectives**
  - Use the SDSoC environment to debug an Standalone application
  - Use the SDSoC environment to debug an Linux application

Using C-Callable libraries and creating multiple accelerators

Zynq
SDSoC 2015.4 Version

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
  - Describe how to implement pipelining through the use of pragmas

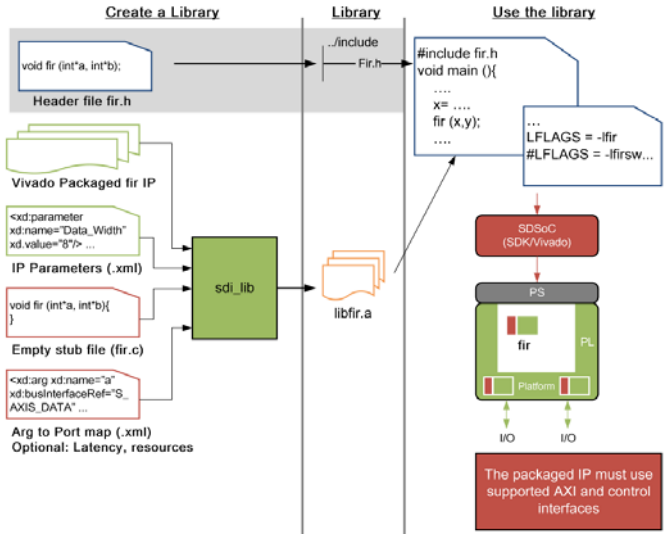
Outline

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

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

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: fir.h
#define N 256
void fir(signed char X[N], short Y[N]);
```

C-Callable Function Components (2)

- **Function definition**
  - The function interface defines the entry points into the library
  - The function definitions can contain empty function bodies since the SDSoC compilers will replace them with API calls to execute data transfers to/from the IP block
    - The implementation of these calls depend upon the data motion network created by the SDSoC system compilers
  - Example:

```
// FILE: fir.c
#include "fir.h"
#include <stdlib.h>
#include <stdio.h>
void fir(signed char X[N], short Y[N]) {
// SDSoC replaces function body with API calls for data transfer }
```


C-Callable Function Components (3)

- **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 (4)

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:

```
<!—- FILE: fir.params.xml --
><?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="fir_compiler">
<xd:parameter xd:name="DATA_Has_TLAST" xd:value="Packet_Framing"/>
<xd:parameter xd:name="M_DATA_Has_TREADY" xd:value="true"/>
<xd:parameter xd:name="Coefficient_Width" xd:value="8"/>
...
```


C-Callable Function Components (5)

Function Argument Map

- Information is captured in a "function map" XML file
  - Function name – the name of the function mapped onto a component
  - Component reference – the IP type name from the IP-XACT Vendor-Name-Library-Version identifier
  - C argument name and direction – an address expression for a function argument, for example x (pass scalar by value) or *p (pass by pointer)
    - Currently the SDSoC environment does not support inout function arguments
  - Bus interface – the name of the IP port corresponding to a function argument
  - Port interface type – the corresponding IP port interface type, which currently must be either aximm (slave only), axis
  - Address offset – hex address, for example, 0x40, required for arguments mapping onto aximm slave ports

C-Callable Function Components (6)

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

  `sdslib [arguments] [options]`

  – Example:

```
sdslib -lib libfir.a fir fir.c fir_reload fir_reload.c \
fir_config fir_config.c \
-vlnv xilinx.com:ip:fir_compiler:7.1 \
-ip-map fir_compiler.fcnmap.xml \
-ip-params fir_compiler.params.xml
```

  - `sdslib` uses the functions fir (in fir.c), fir_reload (in fir_reload.c) and fir_config (in fir_config.c) and archives them into the `libfir.a` static library. The fir_compiler IP core is specified using -vlnv and the function map and IP parameters are specified with `–ip-map` and `–ip-params` respectively

Argument	Description
`-lib <libname>`	Library name to create or append to
`<function_name file_name>+`	One or more `<function, file>` pairs. For example: `fir fir.c`
`-vlnv <v>:<l>:<n>:<v>`	Use IP core specified by this vlnv. For example, `-vlnv xilinx.com:ip:fir_compiler:7.1`
`-ip-map <file>`	Use specified <file> as IP function map
`-ip-params <file>`	Use specified <file> as IP parameters

Option	Description
`-ip-repo <path>`	Add HDL IP repository search path
`-os <name>`	Specify target Operating System • linux (default) • standalone (bare-metal)
`--help`	Display this information

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

Outline

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

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);
```

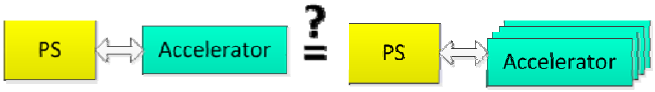
Forcing Multiple Instances of an Accelerator

- **`#pragma SDS async (id)` before function call**
  - 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 async(1)
mmult_accel(...); // instance 1
#pragma SDS async(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

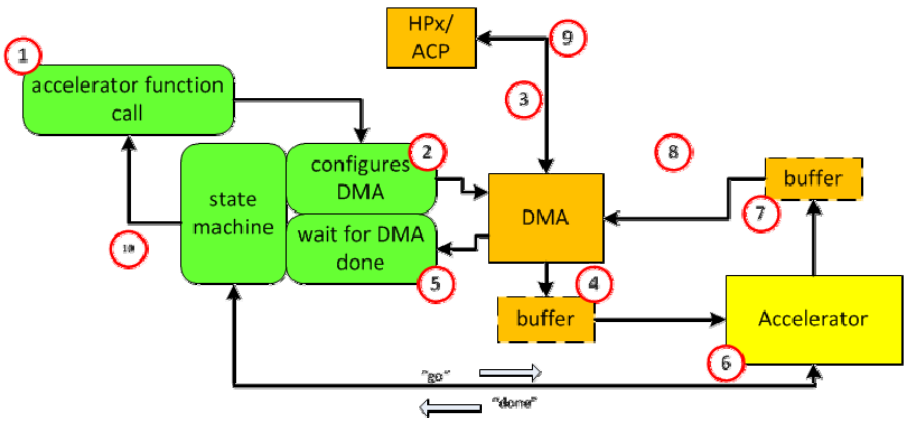


Outline

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

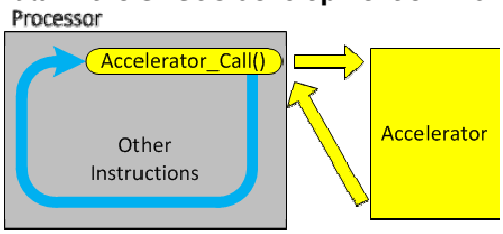
An Accelerator Task

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

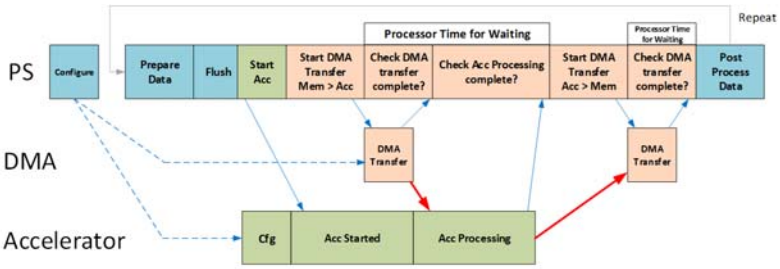


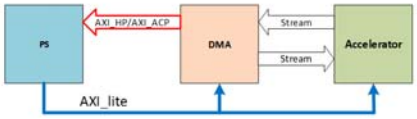
Blocking Behavior

- **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**



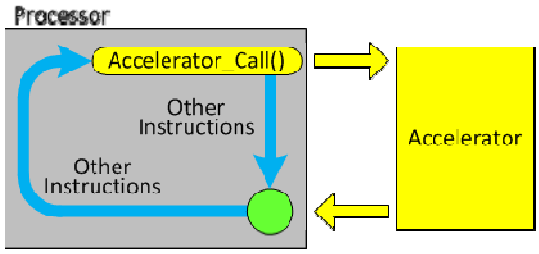
Accelerator I/O Connected to PS: Blocking



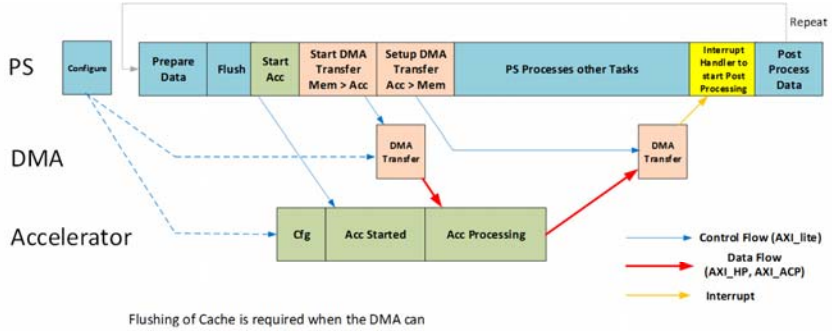


Non-Blocking Behavior

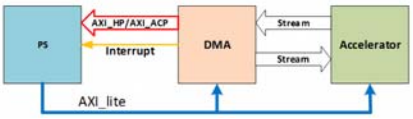
- **True non-blocking (or just non-blocking) behavior allows the processor to continue operation after the accelerator call is made**
- **Benefit is that processor can continue operation simultaneously with accelerator**
- **Drawback is additional complexity (synchronization issues)**
- **SDSoC refers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



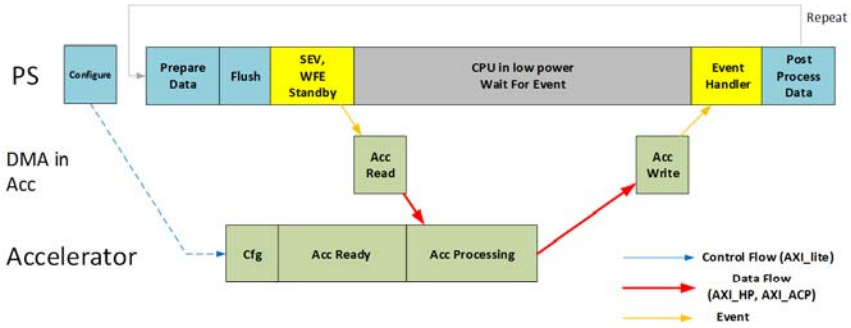
Accelerator I/O Connected to PS: Non-Blocking



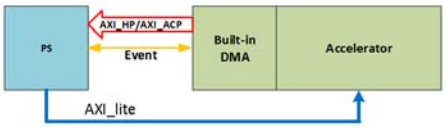
Flushing of Cache is required when the DMA can only access the DDR and not the Cache directly



Accelerator I/O Connected to PS: Blocking for Power

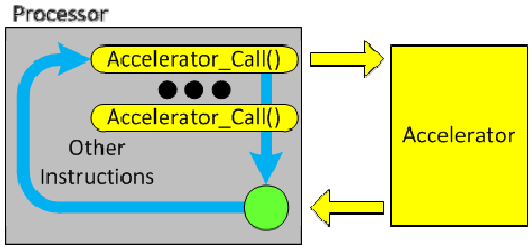


Flushing of Cache is required when the DMA can only access the DDR and not the Cache directly



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*



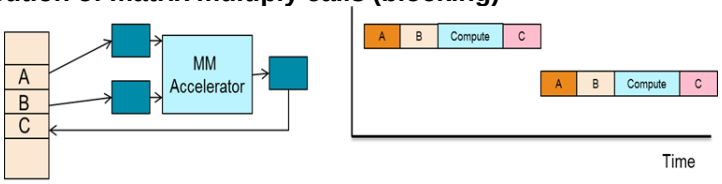
- **The following *stages* of an accelerator task can be independent of each other**
  - Sending input data from shared memory to local accelerator buffers
  - Processing data
  - Returning results from local accelerator buffers to shared memory

Limitations of Task-Level Pipelining

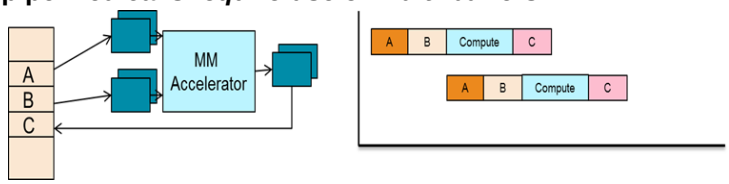
- **Task-level pipelining is a *macro*-level optimization**
  - Only three independent sub-tasks exist for an accelerator call
  - Pipeline depth limited to a maximum of three
- **Asynchronous behavior means inputs/outputs are subject to run-time errors**
  - Program should not access arguments involved in an asynchronous call until the call is complete
  - Premature access can lead to unpredictable results; becomes a procedural/process issue
- **A well-behaved pipeline cannot be guaranteed under all circumstances**
  - Code could still complete sequentially if task latency is too small*
  - Increasingly non-uniform latencies of individual stages will lead to decreasing benefits**

Pipelining Multiple Calls to the Accelerator

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



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



Performance: 23,974 CPU cycles

~7.5X speedup over software version

Outline

- C-Callable Accelerators
- Multiple Accelerators
- Blocking vs Non-Blocking calls
- ***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
- Pipelined tasks include loading data, processing data, and storing results
- Task-level pipelining only applies for consecutive calls to the same accelerator
- A non-blocking accelerator call is just one of several mechanisms to control synchronization and/or pipelining in the SDSoC development environment

Vivado HLS

Zynq
SDSoC 2015.4 Version

Objectives

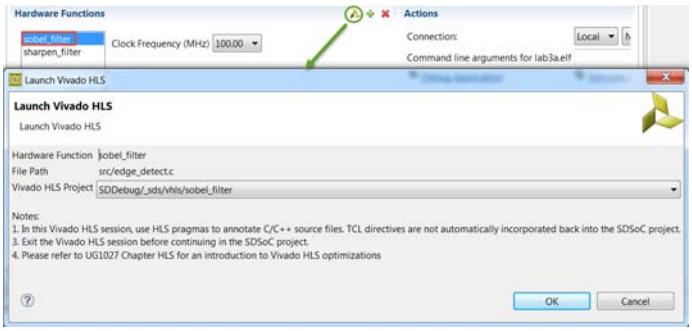
- **After completing this module, you will be able to:**
  - Describe HLS use models in SDSoC
  - List function arguments limitations
  - Explain various optimization techniques
  - Describe what dataflow is

Outline

- ***HLS Use Models***
- **Optimizing Techniques**
- **HLS Libraries**
- **Summary**
- **Lab6 Intro**
- **Case Study**

Relevant HLS Use Models

- **Vivado HLS as IP development environment**
  - Export IP from HLS and import as C-callable IP
- **Vivado HLS as Zynq PL cross-compiler**
  - Hardware function source code shared between SDSoC and VHLS and ***the only*** handoff files
  - VHLS projects are automatically created by SDSoC and are temporary work products (no persistent state)
  - Optionally launch HLS from SDSoC
    - Optimize accelerator code
    - Simulate hardware function



Hardware-aware Software Programming

The most common and important HLS compiler directives are natural to performance-oriented software programmers

- **Use hardware buffers to improve communication between accelerator and external memory**
  - Copy loops at the function boundary when multiple accesses required and to burst data into local buffers
- **Inline functions to improve boundary optimization**
- **Pipeline and unroll inner loops to increase concurrency**
- **Partition arrays to increase memory bandwidth**
- **Use "dataflow" for concurrent execution of independent blocks of code within an HLS function**
  - SDSoC automatically implements dataflow pipelining between cascaded hardware functions

Hardware Function Argument Interfaces

- **Use standard C99 type arguments at the top-level interface**
  - Avoid **`long`** and **`bool B[16]`**
  - Avoid arrays of struct (in future, SDSoC will support packed, gcc-compatible arrays)
  - Avoid **`ap_int<>`**, **`ap_fixed<>`**, **`hls::stream`** except widths **`8:16:32:64`**
    - Must use **`#ifndef __SDS_VHLS__`** to coerce to like-sized C99 type
      - **`sdscc/sds++`** always passes **`__SDS_VHLS__`** macro to HLS
    - **`hls::stream`** arguments must be presented to **`sdscc/sds++`** as arrays
- **Let SDSoC generate HLS interface directives**
  - Use **`#pragma SDS data access_pattern(a:RANDOM, b:SEQUENTIAL)`**
    - Employ **`#pragma HLS interface`** only to override
  - Pitfall: incomplete HLS interface pragma specification can result in cryptic errors
- **Pitfall: "simulates in HLS" does not imply "correct in SDSoC"**
  - Examples: missing **`#pragma HLS interface s_axilite port=return`**
    - **`#pragma HLS interface s_axilite port=foo`**
    - **`#pragma HLS interface m_axi port=foo offset=slave`**

Outline

- HLS Use Models
- ***Optimizing Techniques***
- HLS Libraries
- Summary
- Lab6 Intro
- Case Study

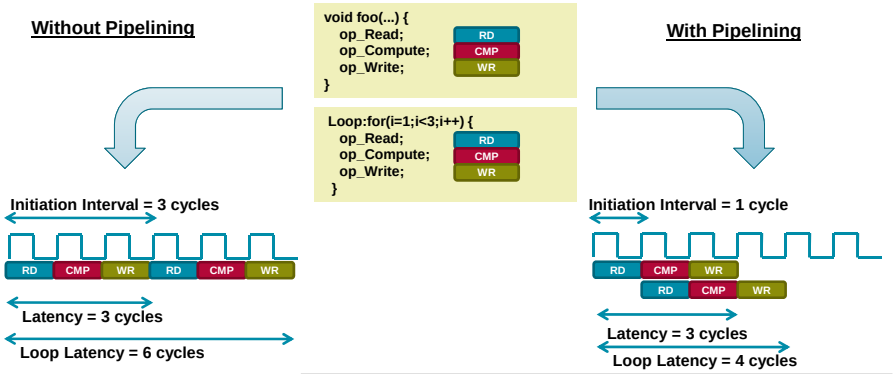
Primary Optimization Techniques

- **Memory access can be the biggest performance bottleneck**
  - May have to restructure algorithm to optimize memory accesses for hardware (similar issues for GPU, DSPs)
  - Pointers are an efficient way to pass ownership to software functions, but not necessarily to hardware functions
  - Burst data into local arrays using memcpy or copy loops (e.g., line buffer, small matrices)
- **Increase local concurrency**
  - Pipeline and dataflow loops, function calls, and operations
  - Enhance pipeline performance

HLS Optimization Pragmas

Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks

Loop and function pipelining

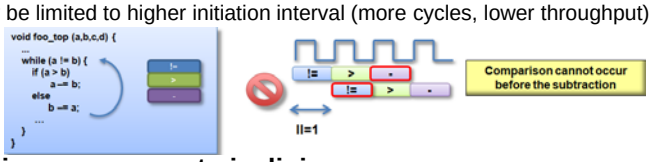



```
for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
     float result = 0;
     for (index_d = 0; index_d < A_NCOLS; index_d++) {
         float product_term = in_A[index_a][index_d] * in_B[index_d][index_b];
         result += product_term;
     }
     out_C[index_a * B_NCOLS + index_b] = result;
}
```

- **Pipelined loops**
  - Combined with array partitioning to achieve II=1

Issues Which Prevent Pipelining

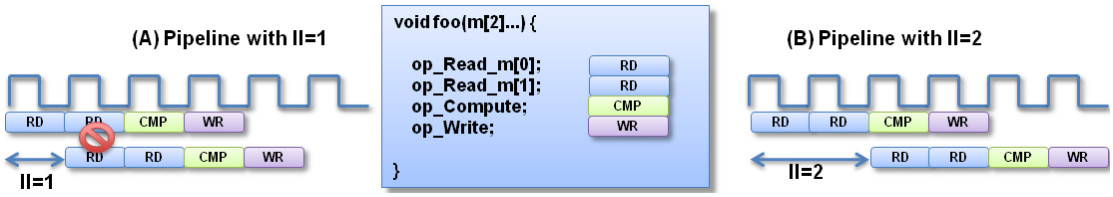
- **Pipelining functions unrolls all loops**
  - Loops with variable bounds cannot be unrolled
  - This will prevent pipelining
    - Re-code to remove the variables bounds: max bounds with an exit
- **Feedback prevent/limits pipelines**
  - Feedback within the code will prevent or limit pipelining
    - The pipeline may be limited to higher initiation interval (more cycles, lower throughput)



- **Resource Contention may prevent pipelining**
  - Can occur within input and output ports/arguments
  - This is a classis way in which arrays limit performance

Resource Contention: Unfeasible Initiation Intervals

- **Sometimes the II specification cannot be met**
  - In this example there are 2 read operations on the same port



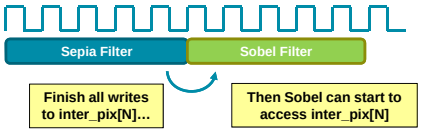
  - An II=1 cannot be implemented
    - The same port cannot be read at the same time
    - Similar effect with other resource limitations
    - For example if functions or multipliers etc. are limited
- **Vivado HLS will automatically increase the II**
  - Vivado HLS will always try to create a design, even if constraints must be violated

Dataflow

- **Default behavior**
  - Complete a function or loop iteration before starting next function or loop iteration

```
//This memory is turned into a FIFO during optimization
 rgb_pixel inter_pix[MAX_HEIGHT][MAX_WIDTH];

// Primary processing functions
 sepia_filter(in_pix,inter_pix);
 sobel_filter(inter_pix,out_pix2);
```

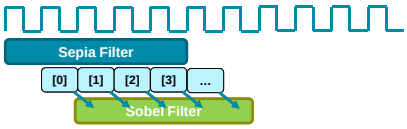
Sepia Filter
Sobel Filter



Finish all writes to inter_pix[N]…

Then Sobel can start to access inter_pix[N]

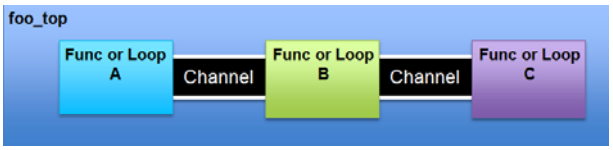
- **Dataflow**
  - Start next function or loop iteration as soon as “ready” and data is available
  - Initiation interval (II) represents number of clocks between ‘starts’
  - Increased concurrency
  - Buffers data between processes
    - Worst case 2-BRAM (ping-pong)
    - Optimized case, 1 reg (1 element FIFO)



- **Apply dataflow within a function but not at the top level**

Dataflow Optimization

- **Dataflow Optimization**
  - Allows blocks of code to operate concurrently
    - The blocks can be functions or loops
    - Dataflow allows loops to operate concurrently
  - It places channels between the blocks to maintain the data rate

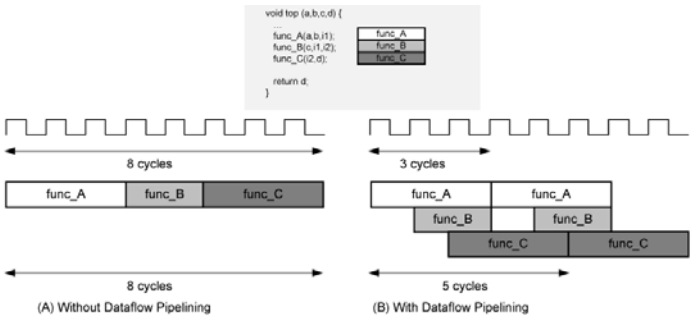


    - For arrays the channels will include memory elements to buffer the samples
    - For scalars the channel is a register with hand-shakes
- **Dataflow optimization therefore has an area overhead**
  - Additional memory blocks are added to the design
  - The timing diagram on the previous page should have a memory access delay between the blocks
    - Not shown to keep explanation of the principle clear

Dataflow Pipelining

- **Function dataflow pipelining**
  - Use `#pragma HLS dataflow` where the data flow optimization is desired
  - Example:

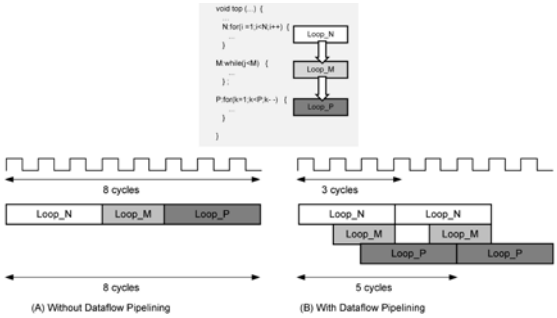
```
void top(a, b, c, d) {
#pragma HLS dataflow
func_A(a, b, i1);
func_B(c, i1, i2);
func_C(i2, d);
}
```



Dataflow Pipelining (2)

- **Loop dataflow pipelining**
  - Enables a sequence of loops, normally executed sequentially, to execute concurrently
- Use `#pragma HLS dataflow` where the data flow optimization is desired
  - Example:

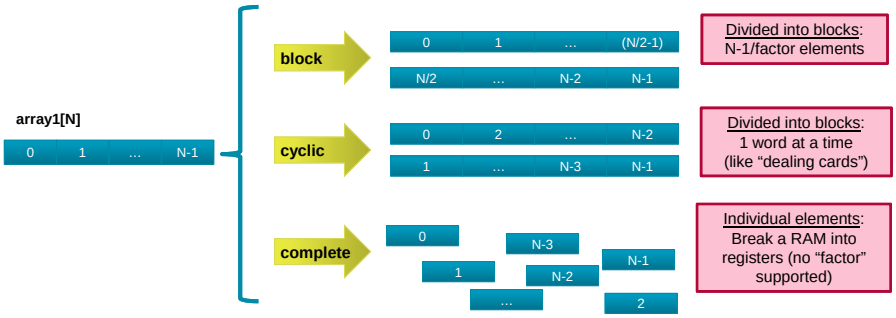
```
void top(a, b, c, d) {
#pragma HLS dataflow
N: for(;;) { }
M: for(;;) { }
P: for(;;) { }
}
```



- Do not apply on a scope which contains a mixture of loops and functions

Array Partitioning

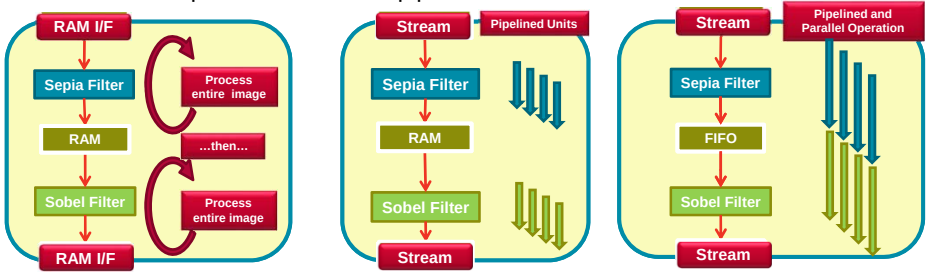
- **Partition into multiple memories to increase concurrent access**




```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS], float in_B[A_NCOLS][B_NCOLS], float out_C[A_NROWS*B_NCOLS])
{
#pragma HLS INLINE self
#pragma HLS array_partition variable=in_A block factor=16 dim=2
#pragma HLS array_partition variable=in_B block factor=16 dim=1
// snip
}
```


Pipelining and Dataflow

- **Optimizations for cascaded loops, function calls, and operators**
  - Eliminate or reduce dependencies across pipelines



	Baseline	*Pipelined*	*Dataflow*
BRAM	2792	2790	**24**
FF	**891**	**1136**	**883**
LUT	**2315**	**2114**	**1606**
Interval	128,744,588	4,150,224	**2,076,613**

Outline

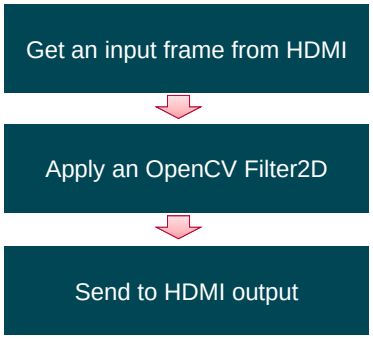
- HLS Use Models
- Optimizing Techniques
- ***HLS Libraries***
- Summary
- Lab6 Intro
- Case Study

HLS Libraries

- Vivado HLS libraries are provided in the SDSoC environment
- Ensure that the source code conforms to the following rules
  - Hardware function arguments resolve to a C99 basic arithmetic type, a pointer to, array of, or a struct whose members flatten to a C99 basic arithmetic type
  - Scalar arguments must fit in a 32-bit container
  - Provide a C/C++ wrapper function to ensure the functions export a software interface

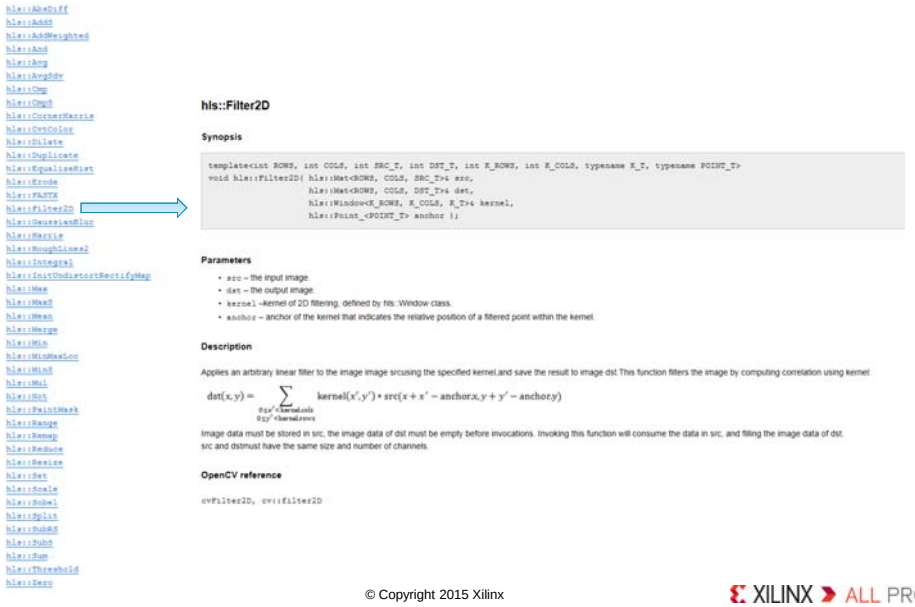
```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_
#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif
void my_sqrt(float x, float *ret);
#endif // _SQRT_H_
```

Basic Filter2D OpenCV Application




```
#include "image_filters.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
using namespace cv;

void image_filters( int array_in[1080*2048], int array_out[1080*2048])
{
      const char coef[3][3] =  {{-1,-2,-1},
                                { 0, 0, 0},
                                { 1, 2, 1}};

      Mat src(1080, 2048, CV_8UC4, (int *)array_in);
      Mat dst(1080, 2048, CV_8UC4, (int *)array_out);
      Mat kernel(3, 3, CV_8SC1, (char *)coef);
      Point anchor;
      double delta = 0;
      int ddepth = -1;
      anchor = Point( -1, -1 );
      filter2D(src, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT );
}
```


Hardware-Optimized VHLS Video Libraries



Using VHLS Video Libraries

```
#include "image_filters.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
using namespace cv;

void image_filters( int hdmi_in[1080*2048], int hdmi_out[1080*2048])
{

      const char coef[3][3] =  {{-1,-2,-1},
                                { 0, 0, 0},
                                { 1, 2, 1}};

      Mat src(1080, 2048, CV_8UC4, (int *)array_in);
      Mat dst(1080, 2048, CV_8UC4, (int *)array_out);
      Mat kernel(3, 3, CV_8SC1, (char *)coef);
      Point anchor;
      double delta = 0;
      int ddepth = -1;
      anchor = Point( -1, -1 );
      filter2D(src, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT );
}
```

```
#include <hls_video.h>
#include "define.h"
#include "hw_dtype.h"
#include "image_filters.h"
using namespace hls;

void image_filters( int hdmi_in[1080*2048],
          int hdmi_out[1080*2048])
{
    const char coef[3][3] = {{-1,-2,-1},
                             { 0, 0, 0},
                             { 1, 2, 1}};

    Mat<1080, 1920, HLS_8UC3>   src(1080, 1920);
    Mat<1080, 1920, HLS_8UC3>   dst(1080, 1920);
    Window<3,3,char> kernel;
    Point_<int> anchor;
    anchor = Point_<int>( -1, -1 );

    for(int j=0; j<3; j++)
        for(int i=0; i<3; i++)
            kernel.val[j][i] = coef[j][i];

    AXIM2Mat<2048>(hdmi_in, src);
    Filter2D(src, dst, kernel, anchor);
    Mat2AXIM<2048>(dst, hdmi_out);
}
```


Outline

- HLS Use Models
- Optimizing Techniques
- HLS Libraries
- ***Summary***
- Lab6 Intro
- Case Study

Summary

- **Vivado HLS can be used as an IP development environment to create C-callable IP**
- **Vivado HLS is used as a Zynq PL cross-compiler in SDSoC**
- **Function arguments of the top-level hardware functions must resolve to a C99 basic arithmetic type, a pointer to, array of, or a struct whose members flatten to a C99 basic arithmetic type**
- **When needed the hardware function performance optimization can be achieved by using pipelining, dataflow, unrolling, and array partitioning**
- **Dataflow optimization places channels between the blocks to maintain the data rate**
  - The channels could be a ping-pong buffer or a FIFO
    - The ping-pong buffer is used when the data are accessed in a random order
    - The FIFO is used when the data are accessed sequentially

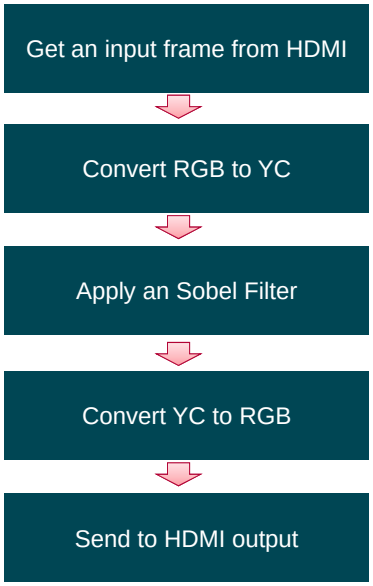
Outline

Lab6 Intro

- **Introduction**
  - This lab introduces various techniques and directives of Vivado HLS which can be used in SDSoC to improve design performance. The design under consideration performs discrete cosine transformation (DCT) on an 8x8 block of data
- **Objectives**
  - Improve performance using PIPELINE directive
  - Understand DATAFLOW directive functionality
  - Apply memory partitions technique to improve data access

Outline

Reference Application




```
#include <stdio.h>
#include "frame_size.h"
#include "img_cores.h"
void img_process(unsigned int *rgb_data_in, unsigned int *rgb_data_out,
{

    rgb_pad2ycbcr(rgb_data_in, yc_data_in);
    sobel_filter(yc_data_in, yc_sobel_out);
    ycbcr2rgb_pad(yc_sobel_out, rgb_data_out);
}
```


Naïve Sobel Filter Implementation

```
int sobel_filter(unsigned short yc_in[NUMROWS*NUMCOLS],unsigned short yc_out[NUMROWS*NUMCOLS])
{
    int row, col;
    for(row = 0; row < NUMROWS+1; row++){
        for(col = 0; col < NUMCOLS+1; col++){
            unsigned short input_data;
            unsigned char edge;

            if((col < NUMCOLS) & (row < NUMROWS))
                input_data = yc_in[row*NUMCOLS+col];
            if( row <= 1 || col <= 1 || row > (NUMROWS-1) || col > (NUMCOLS-1))
                edge=0;
            else{
                short x_weight = 0, y_weight = 0, edge_weight;
                char i, j;
                const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
                const short y_op[3][3] = { {1,2,1}, {0,0,0}, {-1,-2,-1}};

                for(i=0; i < 3; i++){
                    for(j = 0; j < 3; j++){
                        unsigned short temp = (yc_in[(row+i-1)*NUMCOLS+(col+j-1)]) >> 8;
                        x_weight = x_weight + (temp * x_op[i][j]);
                        y_weight = y_weight + (temp * y_op[i][j]);
                    }
                }
                edge_weight = ABS(x_weight) + ABS(y_weight);
                if(edge_weight > 200)   edge_weight = 255;
                else if(edge_weight < 100)      edge_weight = 0;
                edge = (unsigned char)edge_weight;
            }

            if(row > 0 && col > 0){
                if (col < (NUMCOLS >> 1)) // for demo only, copy the input directly to output
                    yc_out[(row-1)*NUMCOLS+(col-1)] = input_data;
                else // for demo only
                    yc_out[(row-1)*NUMCOLS+(col-1)] = (edge << 8) | (unsigned short)128;
            }
        }
    }
    return 0;
}
```

Iterate over an input video image

Sobel filter 2D matrices

Applying 3x3 sobel filter

Edge threshold

Writing to output image

Video 1: Run the Naïve Sobel on a Zynq board

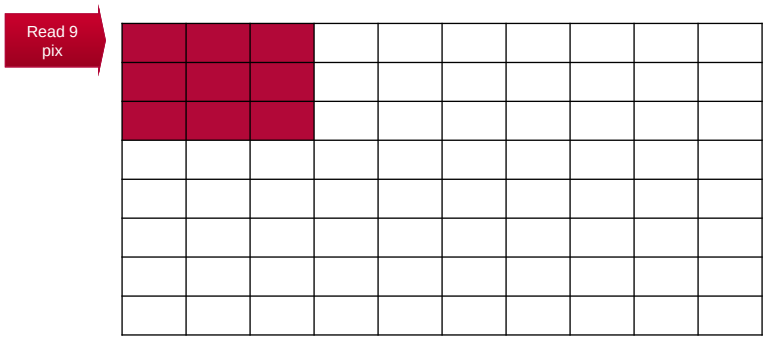
- **0.1 FPS @ 1080p**

Problem 1: Non-Sequential Overlapped Memory Access

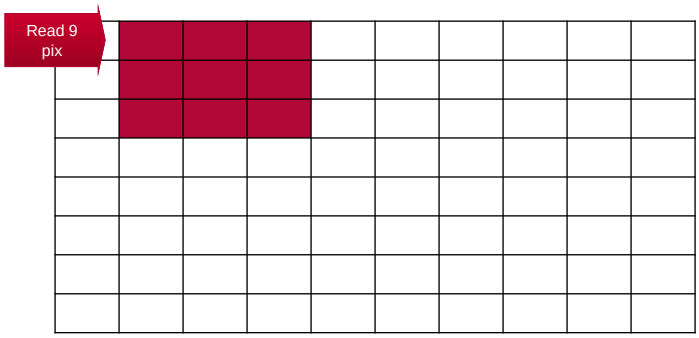
```
int sobel_filter(unsigned short yc_in[NUMROWS*NUMCOLS],unsigned short yc_out[NUMROWS*NUMCOLS])
{
    int row, col;
    for(row = 0; row < NUMROWS+1; row++){
        for(col = 0; col < NUMCOLS+1; col++){
            unsigned short input_data;
            unsigned char edge;

            if((col < NUMCOLS) & (row < NUMROWS))
                input_data = yc_in[row*NUMCOLS+col];
            if( row <= 1 || col <= 1 || row > (NUMROWS-1) || col > (NUMCOLS-1))
                edge=0;
            else{
                short x_weight = 0, y_weight = 0, edge_weight;
                char i, j;
                const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
                const short y_op[3][3] = { {1,2,1},

                for(i=0; i < 3; i++){
                    for(j = 0; j < 3; j++){
                        unsigned short temp = (yc_in[(row+i-1)*NUMCOLS+(col+j-1)]) >> 8;
                        x_weight = x_weight + (temp * x_op[i][j]);
                        y_weight = y_weight + (temp * y_op[i][j]);
                    }
                }
                edge_weight = ABS(x_weight) + ABS(y_weight);
                if(edge_weight > 200)   edge_weight = 255;
                else if(edge_weight < 100)      edge_weight = 0;
                edge = (unsigned char)edge_weight;
            }

            if(row > 0 && col > 0){
                if (col < (NUMCOLS >> 1)) // for demo only, copy the input directly to output
                    yc_out[(row-1)*NUMCOLS+(col-1)] = input_data;
                else // for demo only
                    yc_out[(row-1)*NUMCOLS+(col-1)] = (edge << 8) | (unsigned short)128;
            }
        }
    }
    return 0;
}
```

9 memory access per iteration

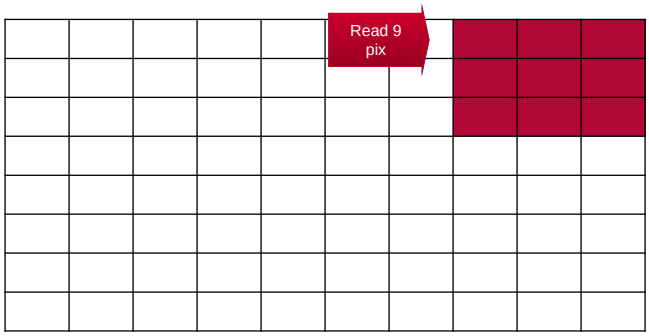
Problem 1: Non-Sequential Overlapped Memory



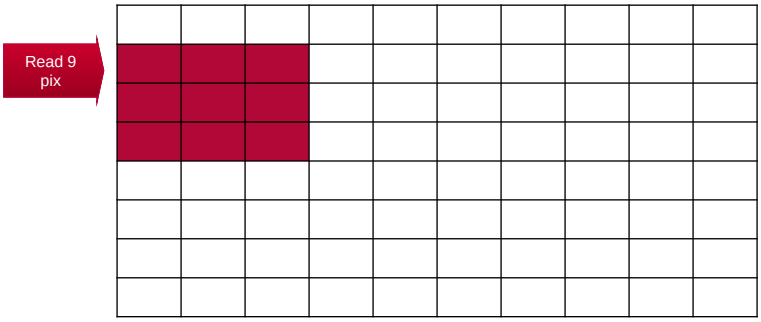
Problem 1:



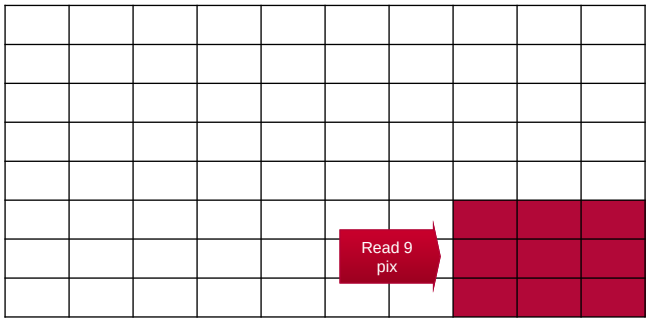
Problem 1:



Problem 1:



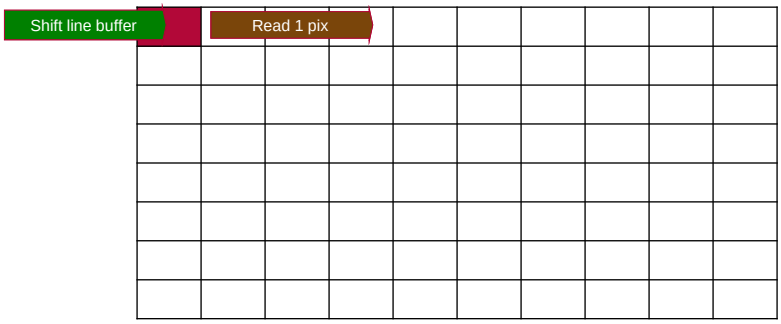
Problem 1:



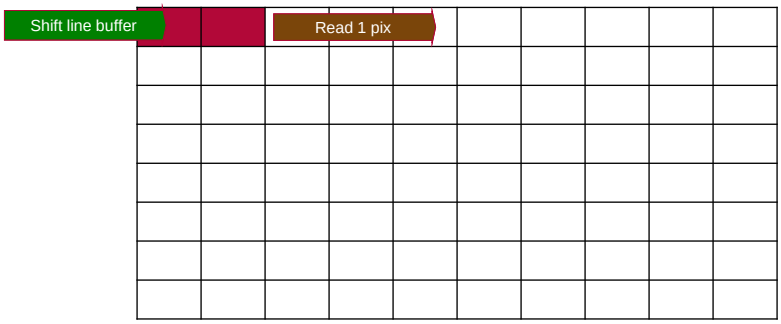
Memory Access: 9 * 48 = 432 pixels

For 1080p: 18,608,436 pixels

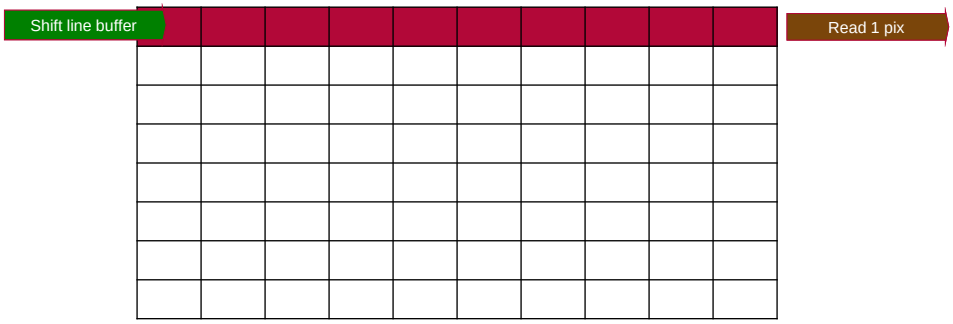
Solution 1: Direct Stream using Line Buffer and Window



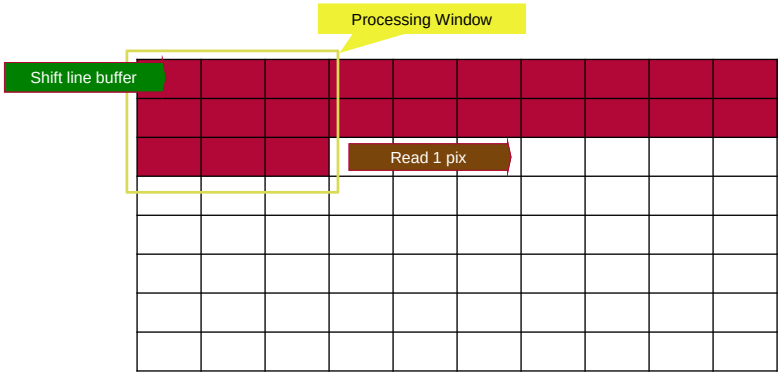
Solution



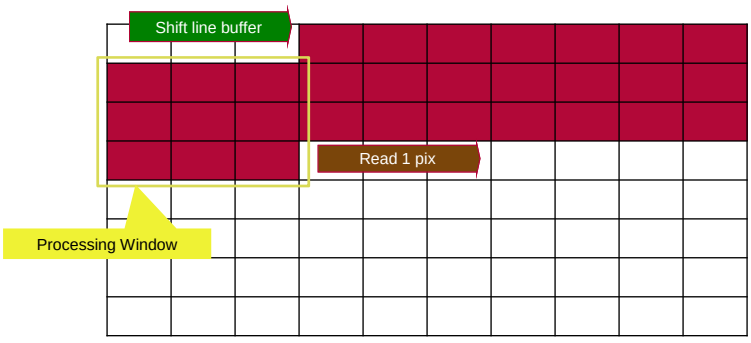
Solution



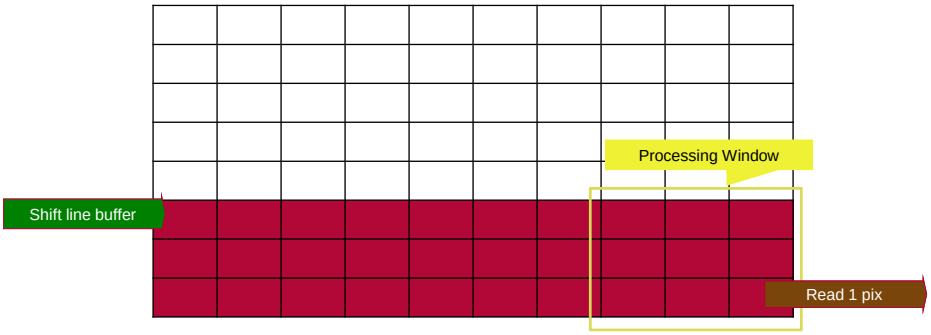
Solution



Solution



Solution



Memory Access: 1 * 80 = 80 pixels

With 1080p, 2,073,600 pixels (9x less!)

Solution 1: Use Line Buffer and Window Classes

```
int sobel_filter(unsigned short yc_in[NUMROWS*NUMCOLS],unsigned short yc_out[NUMROWS*NUMCOLS])
{
    int row;
    int col;

    ap_linebuffer<unsigned char, 3, NUMCOLS> buff_A;
    ap_window<unsigned char,3,3> buff_C;

    for(row = 0; row < NUMROWS+1; row++){
        for(col = 0; col < NUMCOLS+1; col++){
            unsigned short input_data;
            unsigned char temp, tempx;
            if(col < NUMCOLS){
                buff_A.shift_up(col);
                temp = buff_A.getval(0,col);
            }
            if((col < NUMCOLS) & (row < NUMROWS)){
                input_data = yc_in[row*NUMCOLS+col];
                tempx = input_data >> 8 ;
                buff_A.insert_bottom(tempx,col);
            }
            buff_C.shift_right();
            if(col < NUMCOLS){
                buff_C.insert(buff_A.getval(2,col),0,2);
                buff_C.insert(temp,1,2);
                buff_C.insert(tempx,2,2);
            }

            unsigned char edge;
            if( row <= 1 || col <= 1 || row > (NUMROWS-1) || col > (NUMCOLS-1))
                edge=0;
            else{
                short x_weight = 0, y_weight = 0, edge_weight;
                char i, j;
                const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
                const short y_op[3][3] = { {1,2,1}, {0,0,0}, {-1,-2,-1}};

                for(i=0; i < 3; i++){
                    for(j = 0; j < 3; j++){
                        x_weight = x_weight + (buff_C.getval(i,j) * x_op[i][j]);
                        y_weight = y_weight + (buff_C.getval(i,j) * y_op[i][j]);
                    }
                }
                edge_weight = ABS(x_weight) + ABS(y_weight);
                if(edge_weight > 200)   edge_weight = 255;
                else if(edge_weight < 100)      edge_weight = 0;
                edge = (unsigned char)edge_weight;
            }
```

Line buffer and Window classes

Shift line buffer

Read 1 pix and insert to the line buffer

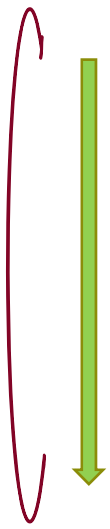
Update the window location

Directly apply sobel filter on the local window

Video 2: Run the Solution 1 on a Zynq board

- 1 FPS @ 1080p
- 10x speedup

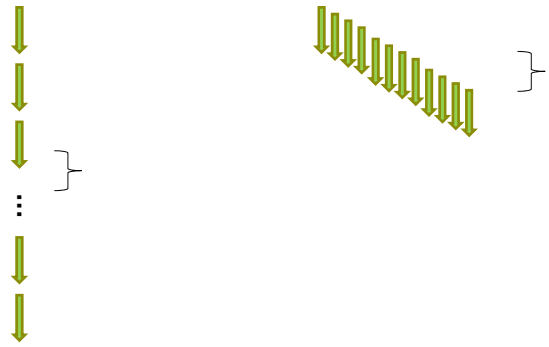
Problem 2: Loop Iterations Are Sequential



```
int sobel_filter(unsigned short yc_in[NUMROWS*NUMCOLS],unsigned short yc_out[NUMROWS*NUMCOLS])
{
    int row, col;
    for(row = 0; row < NUMROWS+1; row++){
        for(col = 0; col < NUMCOLS+1; col++){
            unsigned short input_data;
            unsigned char edge;

            if((col < NUMCOLS) & (row < NUMROWS))
                input_data = yc_in[row*NUMCOLS+col];
            if( row <= 1 || col <= 1 || row > (NUMROWS-1) || col > (NUMCOLS-1))
                edge=0;
            else{
                short x_weight = 0, y_weight = 0, edge_weight;
                char i, j;
                const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
                const short y_op[3][3] = { {1,2,1}, {0,0,0}, {-1,-2,-1}};

                for(i=0; i < 3; i++){
                    for(j = 0; j < 3; j++){
                        unsigned short temp = (yc_in[(row+i-1)*NUMCOLS+(col+j-1)]) >> 8;
                        x_weight = x_weight + (temp * x_op[i][j]);
                        y_weight = y_weight + (temp * y_op[i][j]);
                    }
                }
                edge_weight = ABS(x_weight) + ABS(y_weight);
                if(edge_weight > 200)   edge_weight = 255;
                else if(edge_weight < 100)      edge_weight = 0;
                edge = (unsigned char)edge_weight;
            }

            if(row > 0 && col > 0){
                if (col < (NUMCOLS >> 1)) // for demo only, copy the input directly to output
                    yc_out[(row-1)*NUMCOLS+(col-1)] = input_data;
                else // for demo only
                    yc_out[(row-1)*NUMCOLS+(col-1)] = (edge << 8) | (unsigned short)128;
            }
        }
    }
    return 0;
}
```

Executing sequentially

Solution 2: Pipeline Loop Iterations



60*1920*1080 =
124,416,000 cycles

60+(1920*1080) =
2,073,660 cycles (60x speedup)

Assuming 60 cycles / loop iteration

Solution 2: Add PIPELINE Pragma

```
int sobel_filter(unsigned short yc_in[NUMROWS*NUMCOLS],unsigned short yc_out[NUMROWS*NUMCOLS])
{
    int row;
    int col;

    ap_linebuffer<unsigned char, 3, NUMCOLS> buff_A;
    ap_window<unsigned char,3,3> buff_C;

    for(row = 0; row < NUMROWS+1; row++){
        for(col = 0; col < NUMCOLS+1; col++){
#pragma AP PIPELINE II = 1
            unsigned short input_data;
            unsigned char temp, tempx;
            if(col < NUMCOLS){
                buff_A.shift_up(col);
                temp = buff_A.getval(0,col);
            }
            if((col < NUMCOLS) & (row < NUMROWS)){
                input_data = yc_in[row*NUMCOLS+col];
                tempx = input_data >> 8 ;
                buff_A.insert_bottom(tempx,col);
            }
            buff_C.shift_right();
            if(col < NUMCOLS){
                buff_C.insert(buff_A.getval(2,col),0,2);
                buff_C.insert(temp,1,2);
                buff_C.insert(tempx,2,2);
            }

            unsigned char edge;
            if( row <= 1 || col <= 1 || row > (NUMROWS-1) || col > (NUMCOLS-1))
                edge=0;
            else{
                short x_weight = 0, y_weight = 0, edge_weight;
                char i, j;
                const short x_op[3][3] = { {-1,0,1},{-2,0,2},{-1,0,1}};
                const short y_op[3][3] = { {1,2,1}, {0,0,0}, {-1,-2,-1}};

                for(i=0; i < 3; i++){
                    for(j = 0; j < 3; j++){
                        x_weight = x_weight + (buff_C.getval(i,j) * x_op[i][j]);
                        y_weight = y_weight + (buff_C.getval(i,j) * y_op[i][j]);
                    }
                }
```

Specify pipelining of loop iterations with Initiation Interval = 1

Video 3: Run the Solution 2 on a Zynq board

- **60 FPS @ 1080p**
- **60x speedup**

Summary

	FPS
A naïve sobel filter implementation	0.1
Problem 1: Non-sequential overlaped memory accesses **Solution** 1: Reducing memory access by 9x using line buffer and window	1
Problem 2: Sequential loop iteration **Solution** 2: Pipeline loop iteration using the pipeline pragma	60

600x Speedup with only ~15 lines of code change

SDSoC Platform

Zynq
SDSoC 2015.4 Version

Objectives

- **After completing this module, you will be able to:**
  - Describe what platform is and its components
  - List the supported operating systems

Outline

- *Introduction*
- Creating an SDSoC Platform
- Summary
- Lab7 Intro

What Is An SDSoC Platform?

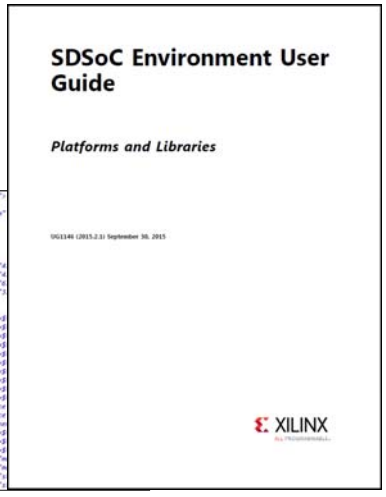
- **An SDSoC platform defines**
  - A base hardware and software architecture and application context
  - The hardware includes processing system, external memory interfaces, and custom input/output
  - The software run time includes operating system (possibly "bare metal"), boot loaders, drivers for platform peripherals and root file system
- **Every project in the SDSoC environment targets a specific platform**
- **Use SDSoC IDE to customize the platform with application-specific hardware accelerators and data motion networks that connect accelerators to the platform**

SDSoC: Software-defined Systems-on-Chip

- **A *software-defined SoC* extends a platform with application-specific hardware *and* software to realize a software application**
  - Multiple applications can target a given platform
  - An application can target multiple platforms
- **Application software defines the SoC**
  - User specifies hardware functions to implement in programmable logic
  - System-optimizing compiler analyzes program dataflow and compiles program into an application-specific SoC
    - Analysis engine employs mappings from function prototypes onto IP blocks
    - Function argument properties are constraints to the system optimizing compiler
  - **`#pragma`** assist the compiler and override inference engine

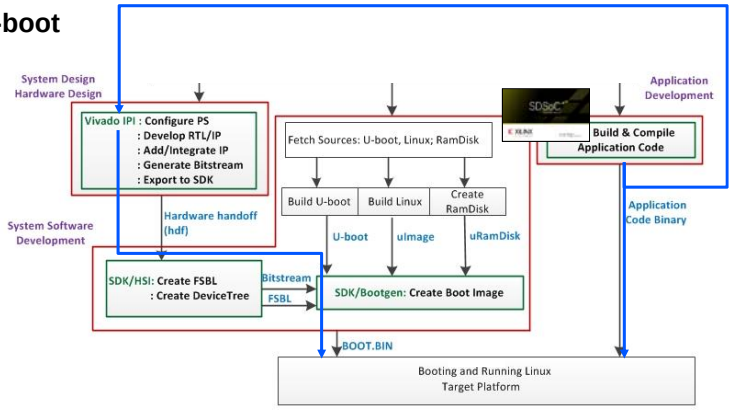
SDSoC: Platform Hardware is a Vivado Design

- **…with a well-defined "platform interface"**
  - AXI, AXI-S, clocks, resets, interrupts
- **Zynq processing system**
- **Memory interfaces and custom I/O**
  - Often domain-specific

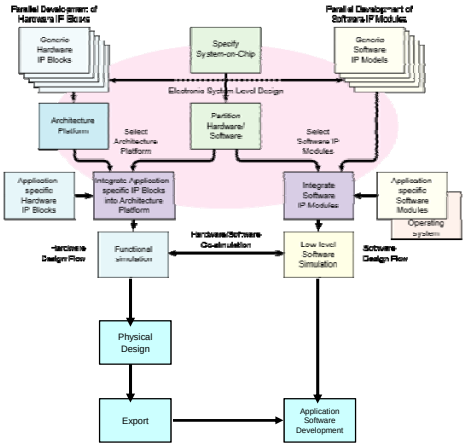


SDSoC: Software is also Part of the Platform

- Operating systems: Linux, FreeRTOS, or bare metal
- Device drivers for platform peripherals
- Boot loaders, e.g., FSBL, u-boot
- Root file system
- Libraries

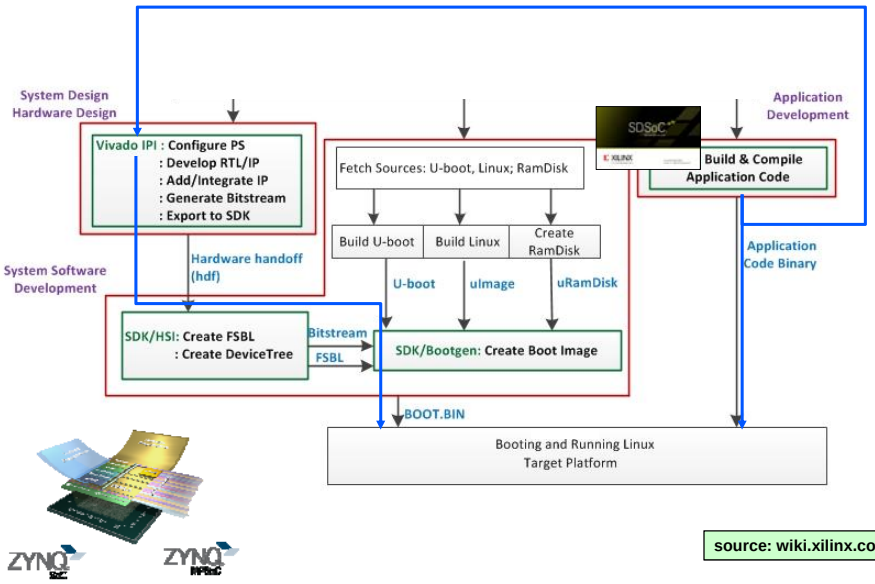


System-on-Zynq SoC / MPSoC Platform Hardware



- Programmable logic “meta-peripheral” for application-specific functions

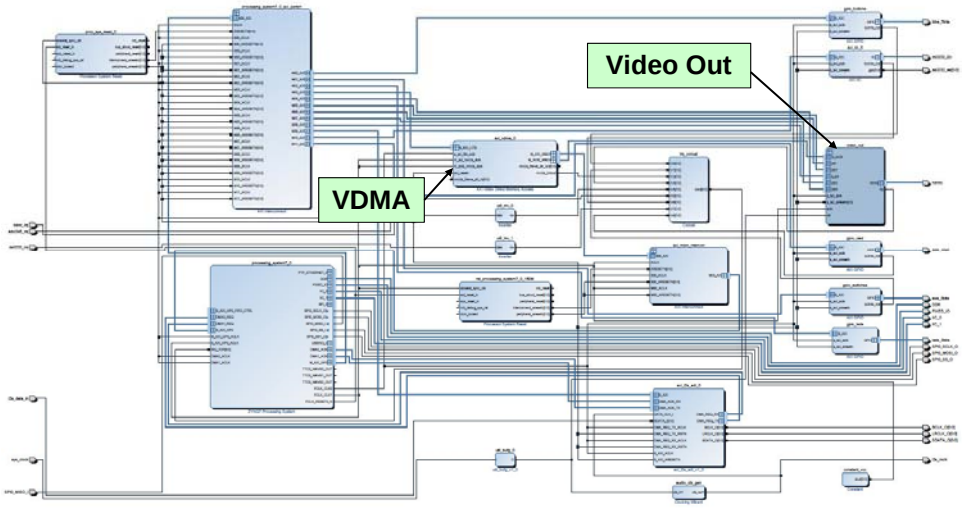
System-on-Zynq SoC / MPSoC Platform Software



SDSoC Platforms

- **The SDSoC development environment includes standard “memory-based I/O” platforms**
  - zc702, zc706, zed, zybo, microzed
- **Several video & image processing oriented platforms**
  - zc702_hdmi, zc702_osd, zed_osd
- **Additional downloads from Xilinx and partners**
  - Zynq Base Targeted Reference Design
  - http://www.xilinx.com/products/design-tools/software-zone/sdsoc.html#boardskits
- **And several “teaching” platform examples**
  - How to support direct I/O
  - How to use platform-specific libraries
  - How to share PS7 AXI interfaces between platform and SDSoC

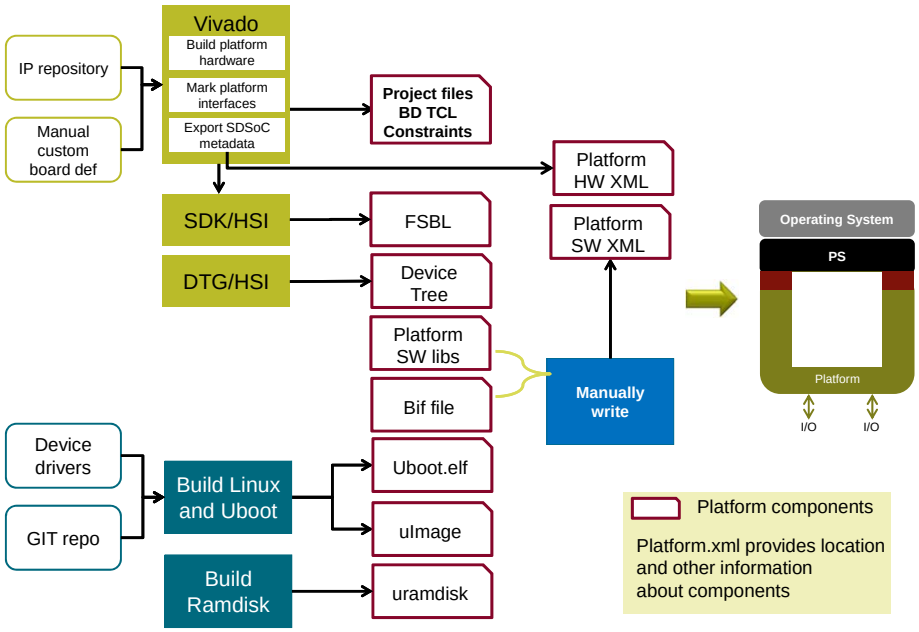
zedboard_osd Platform Hardware



Outline

- Introduction
- ***Creating an SDSoC Platform***
- Summary
- Lab7 Intro

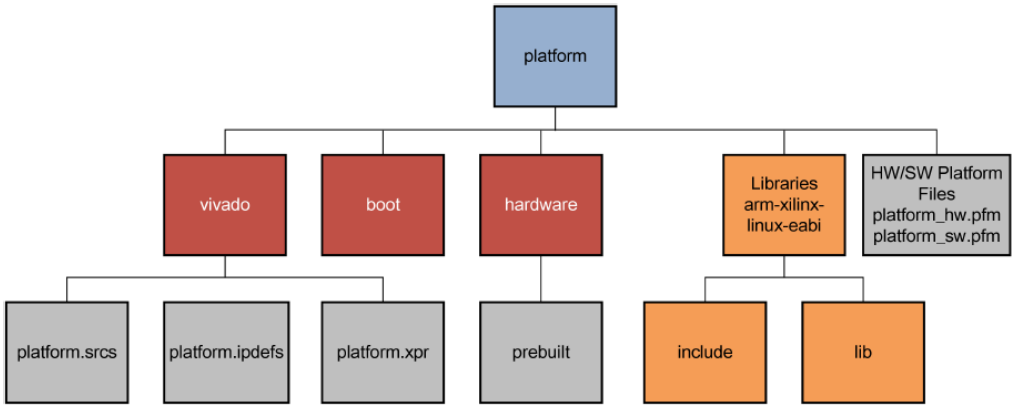
Creating an SDSoC Platform



SDSoC Platform Components

- **An SDSoC platform consists of the following elements:**
  - Metadata files
    - Platform hardware description file (<platform>_hw.pfm)
    - Platform software description file (<platform>_sw.pfm)
  - Vivado Design Suite project
    - Sources
    - Constraints
    - IP blocks
  - Software files
    - Library header files (optional)
    - Static libraries (optional)
    - Linux related objects (device-tree, u-boot, Linux-kernel, ramdisk)
  - Pre-built hardware files (optional)
    - Bitstream
    - Exported hardware files for SDK
    - Pre-generated device registration and port information software files
    - Pre-generated hardware and software interface files

Directory Structure



Hardware Requirements

- **Build and verify the hardware system using the Vivado Design Suite**
- **Execute Tcl APIs in Vivado to accomplish the following steps**
  - Declare the hardware platform name
  - Declare a brief platform description
  - Declare the platform clock ports
  - Declare the platform AXI bus interfaces
  - Declare the platform AXI4-Stream bus interfaces
  - Declare the available platform interrupts
  - Generate the platform hardware description metadata file

Hardware Design Rules

- **The Vivado project name and the IP Integrator block diagram names should be same as the platform name**
  - The block diagram hardware wrapper should be named <platform_name>_wrapper.v
- **Every platform IP that is not part of the standard Vivado IP catalog must be local to the platform Vivado Design Suite project**
  - Create a project, then archive it, and then use the archived project as the hardware platform
    - This ensures that the external referenced Ips are within the Vivado project
- **An SDSoC platform hardware port interface must be an AXI, AXI4-Stream, clock, reset, or interrupt interface only**
  - Custom bus types or hardware interfaces must remain internal to the platform
- **Platform must declare one or more general purpose AXI master ports from a Processing System IP or an interconnect IP connected to such an AXI master port, that will be used by the SDSoC compilers for software control of datamover and accelerator IPs**

Hardware Design Rules (2)

- **To share an AXI port between the SDSoC environment and platform logic, export an unused AXI master or slave of an AXI Interconnect IP block connected to the corresponding AXI port, and the platform must use the ports with least significant indices**
- **Platform AXI4-Stream interface requires TLAST, TKEEP sideband signals to comply with the Vivado tools data mover IP employed by SDSoC environment**
- **Every exported platform clock must have an accompanying Processor System Reset IP block from the Vivado IP catalog**
- **Platform interrupt inputs must be exported by a Concat (xlconcat) block connected to the PS7 IP's IRQ_F2P port**
  - IP blocks within a platform can use some of the sixteen available fabric interrupts, but must use the least significant bits of the IRQ_F2P port without gaps

Software Requirements

- **Software components**
  - Operating system
  - Boot loaders, and
  - Libraries
- **Supported OS**
  - Standalone
  - Linux
  - FreeRTOS
- **Platform may have one or all OS support**
- **By default, the SDSoC environment creates an SD card image to boot a board into a Linux prompt or execute a standalone program**

Software Requirements (2)

- **Boot files**
  - Linux

```
<xd:bootFiles
  xd:os="linux"
  xd:bif="boot/linux.bif"
  xd:readme="boot/generic.readme"
  xd:devicetree="boot/devicetree.dtb"
  xd:linuxImage="boot/uImage"
  xd:ramdisk="boot/uramdisk.image.gz"
/>
```

```
/* linux */
the_ROM_image:
{
    [bootloader]<boot/fsbl.elf>
    <bitstream>
    <boot/u-boot.elf>
}
```

```
<xd:bootFiles
  xd:os="petalinux"
  xd:bif="boot/petalinux.bif"
  xd:readme="boot/generic.readme"
  xd:linuxImage="boot/image.ub" />
```

  - Standalone

```
<xd:bootFiles
  xd:os="standalone"
  xd:bif="boot/standalone.bif"
  xd:readme="boot/generic.readme"
/>
```

```
/* standalone */
the_ROM_image:
{
    [bootloader]<boot/fsbl.elf>
    <bitstream>
    <elf>
}
```

Software Requirements (3)

- **Library files**
  - Linux
    - The platform directory must have arm-xilinx-linux-gnueabi directory
    - Inside the arm-xilinx-linux-gnueabi directory
      - The include directory
      - The lib directory
  - Standalone
    - The platform directory must have arm-xilinx-eabi directory
    - Inside the arm-xilinx-eabi directory
      - The include directory
      - The linker script file
      - The system.mss file

```
<xd:libraryFiles
  xd:os="linux"
  xd:includeDir="arm-xilinx-linux-gnueabi/include"
  xd:libDir="arm-xilinx-linux-gnueabi/lib"
  xd:libName="zc702_axis_io" />
```

```
<xd:libraryFiles
  xd:os="standalone"
  xd:libName="xil"
  xd:bspconfig="arm-xilinx-eabi/system.mss"
  xd:includeDir="arm-xilinx-eabi/include"
  xd:ldscript="arm-xilinx-eabi/lscript.ld"
/>
```


Outline

- Introduction
- Creating an SDSoC Platform
- ***Summary***
- Lab7 Intro

Summary

- **An SDSoC platform defines**
  - A base hardware and software architecture and application context
  - The hardware includes processing system, external memory interfaces, and custom input/output
  - The software run time includes operating system (possibly "bare metal"), boot loaders, drivers for platform peripherals and root file system
- **Creating an SDSoC platform involves generating hardware and software meta data files**
- **The platform name, the Vivado IPI block design name, and the base directory name should be the same**
- **Use hardware meta data file creation API**
- **Software meta data file is created manually**

Outline

Lab7 Intro

- **Introduction**
  - This lab guides you through the steps involved in creating a custom platform for an audio application
- **Objectives**
  - Create an SDSoC platform for a custom application
  - Use the SDSoC environment to test the platform for an audio filtering Standalone application
  - Use the SDSoC environment to test the platform for an audio filtering Linux application