

Profiling

SDx 2018.2



Objectives

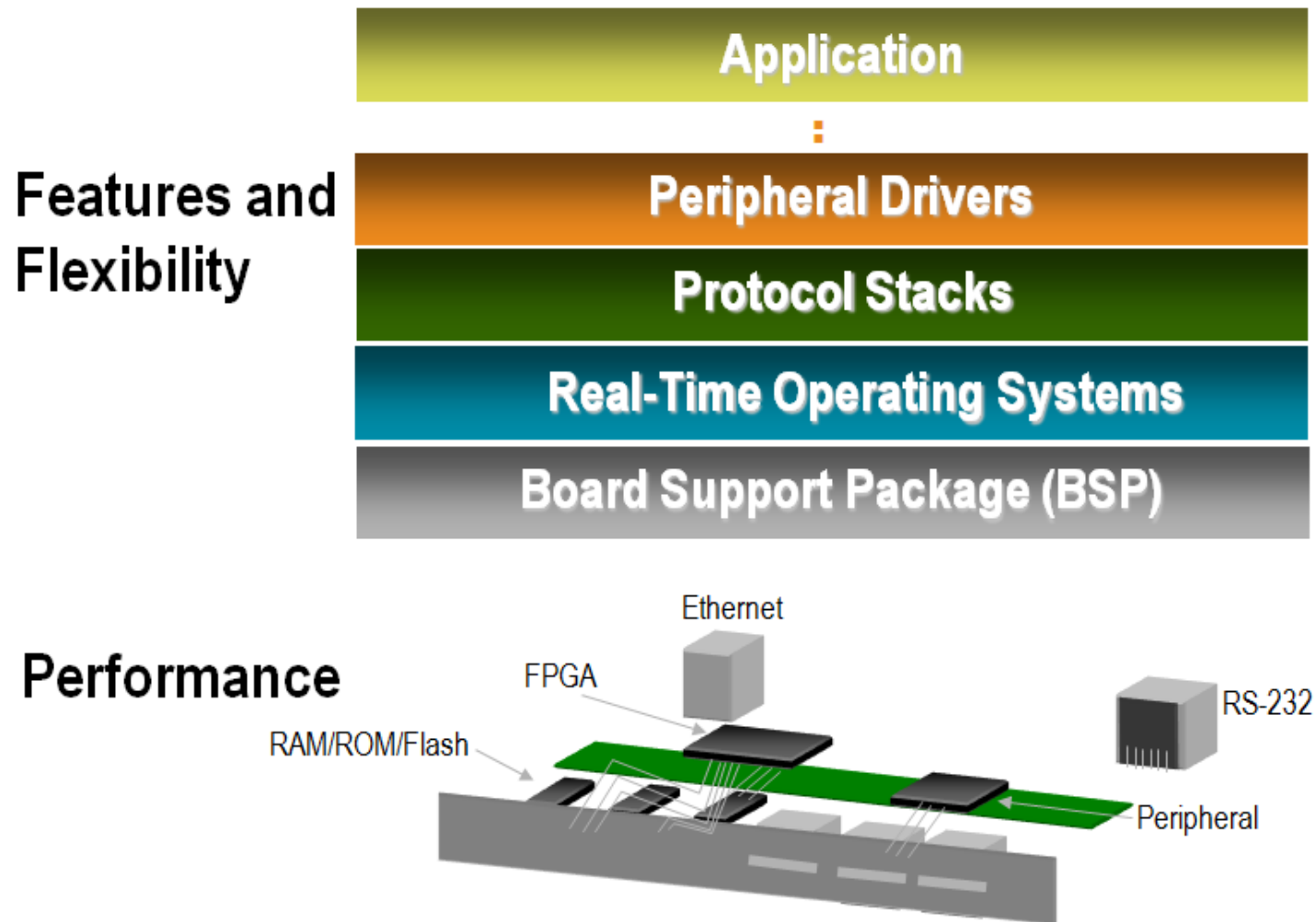
> After completing this module, you will be able to:

- >> 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

Profiling in SDSoC



What is Profiling?

> **Profiling is an analysis of software performance**

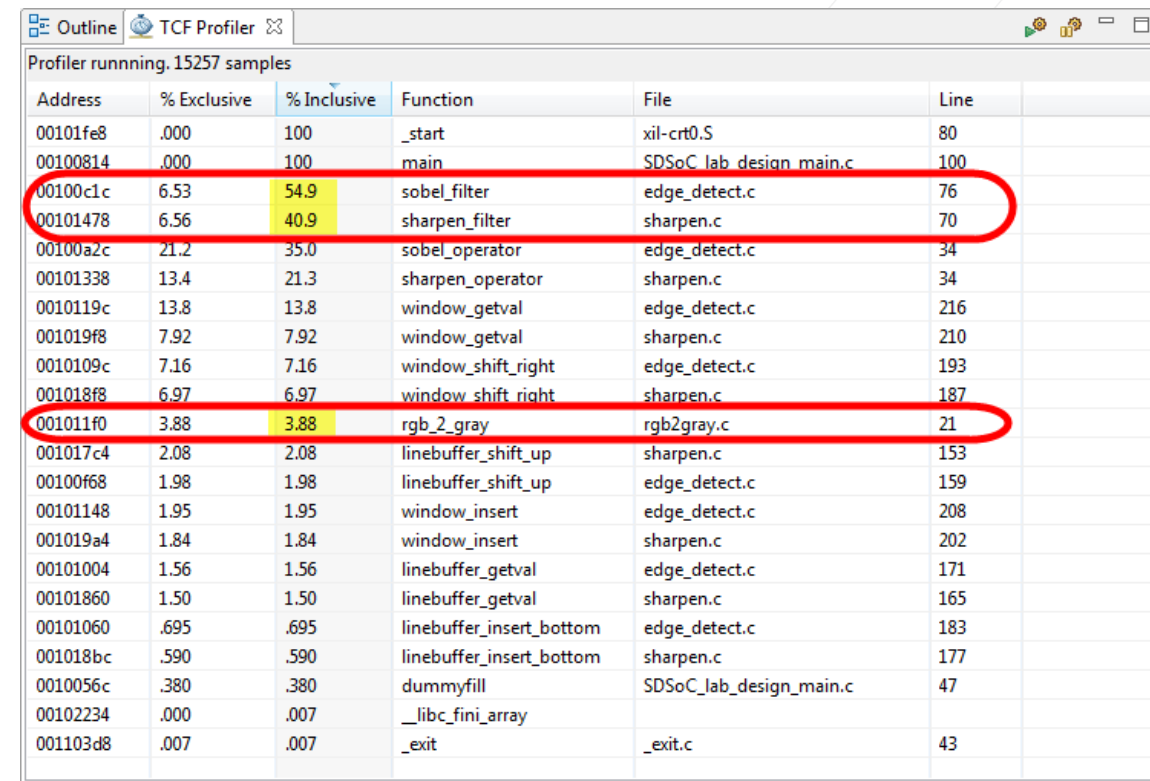
- >> Where routine time is being spent
- >> How many times functions are being called
- >> Which algorithms to consider moving to hardware

> **Several methods**

- >> TCF profiling
 - TCF profiler is the preferred profiling tool and is included with the SDSoC tool
- >> 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 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

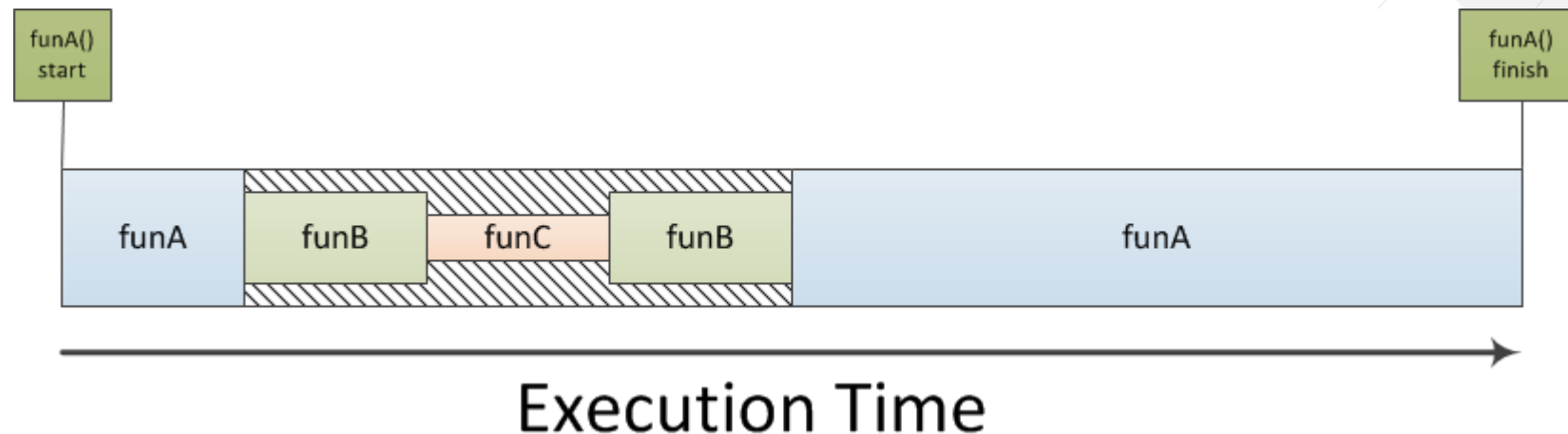


Profiler running. 15257 samples

Address	% Exclusive	% Inclusive	Function	File	Line
00101fe8	.000	100	_start	xil-crt0.S	80
00100814	.000	100	main	SDSoC_lab_design_main.c	100
00100c1c	6.53	54.9	sobel_filter	edge_detect.c	76
00101478	6.56	40.9	sharpen_filter	sharpen.c	70
00100a2c	21.2	35.0	sobel_operator	edge_detect.c	34
00101338	13.4	21.3	sharpen_operator	sharpen.c	34
0010119c	13.8	13.8	window_getval	edge_detect.c	216
001019f8	7.92	7.92	window_getval	sharpen.c	210
0010109c	7.16	7.16	window_shift_right	edge_detect.c	193
001018f8	6.97	6.97	window_shift_right	sharpen.c	187
001011f0	3.88	3.88	rgb_2_gray	rgb2gray.c	21
001017c4	2.08	2.08	linebuffer_shift_up	sharpen.c	153
00100f68	1.98	1.98	linebuffer_shift_up	edge_detect.c	159
00101148	1.95	1.95	window_insert	edge_detect.c	208
001019a4	1.84	1.84	window_insert	sharpen.c	202
00101004	1.56	1.56	linebuffer_getval	edge_detect.c	171
00101860	1.50	1.50	linebuffer_getval	sharpen.c	165
00101060	.695	.695	linebuffer_insert_bottom	edge_detect.c	183
001018bc	.590	.590	linebuffer_insert_bottom	sharpen.c	177
0010056c	.380	.380	dummyfill	SDSoC_lab_design_main.c	47
00102234	.000	.007	_libc_fini_array		
001103d8	.007	.007	_exit	_exit.c	43

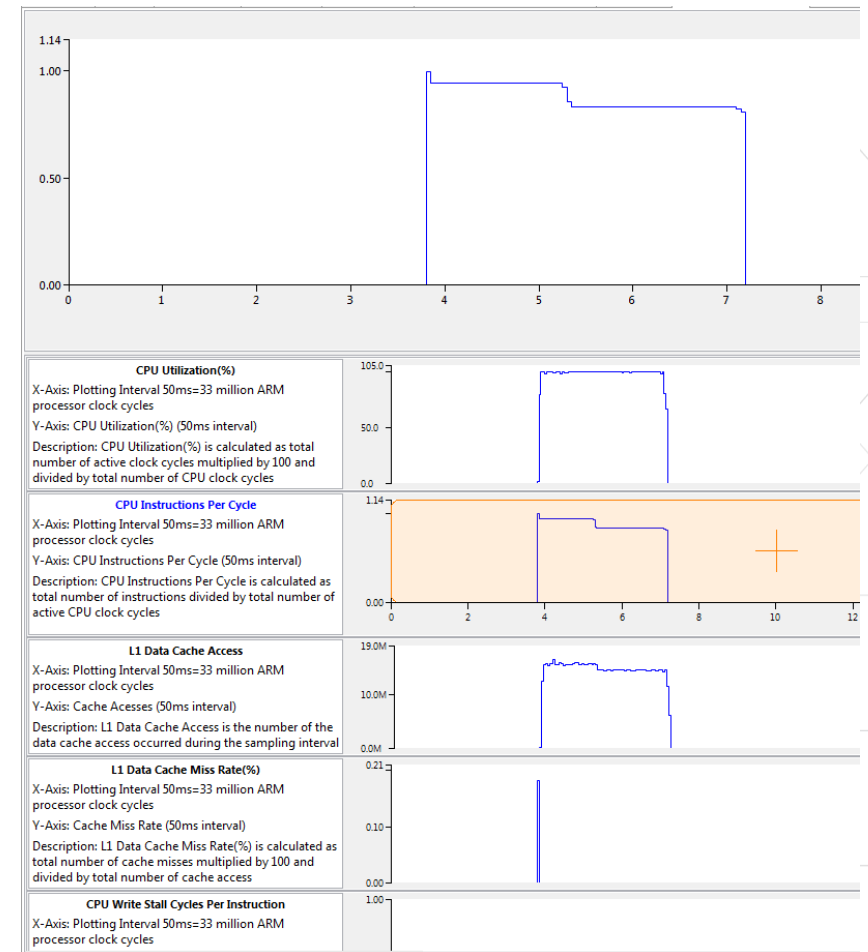
Inclusive vs Exclusive Execution Times

- > **Exclusive:** The amount of execution time spent in *funA* alone. Referencing the diagram below, the exclusive time for *funA* is represented by the combined execution time of the funA blocks only.
- > **Inclusive:** The amount of execution time spent in *funA* and all of its sub-function calls. From the diagram, this is the exclusive time of funA combined with the hatched area during which time *funB* and *funC* are executing.



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
 - >> Compiler 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

Performance Improvement



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

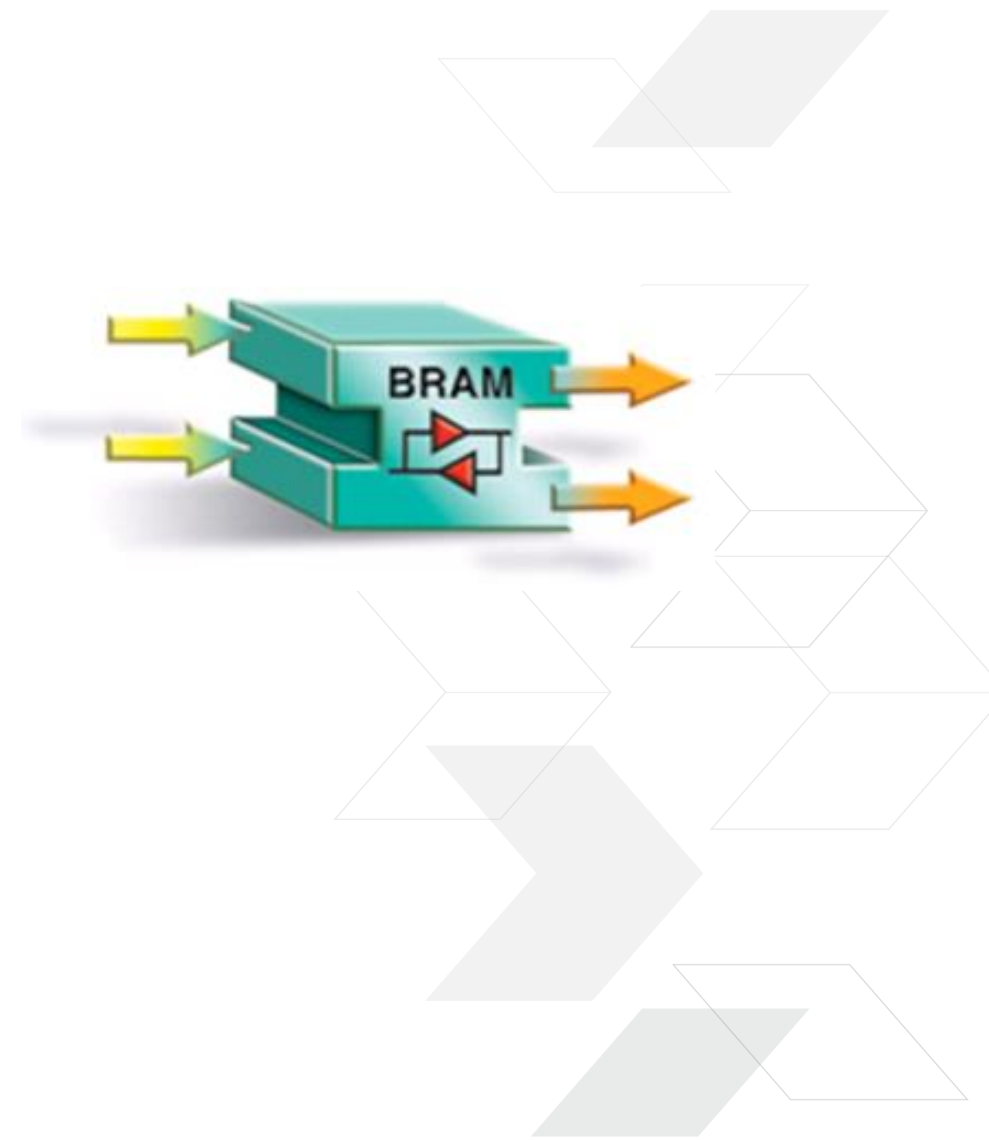
- >> Programmable logic co-processor
 - Customized to user's needs
 - Excellent for iterative and pipelined processing
- >> Add soft core processor in PL
 - Both Cortex-A9 and MicroBlaze processors can co-exist in the AP SoC

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

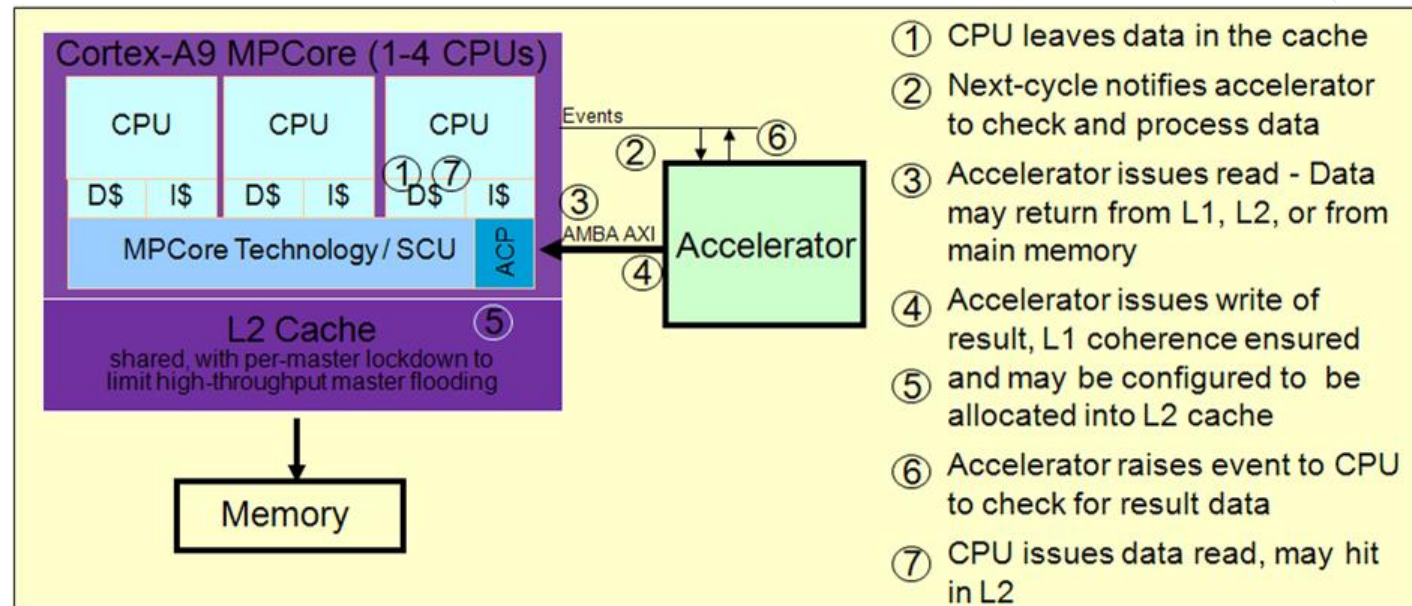
- > **Leverage the dual-port nature of Xilinx block RAM**
- > **Useful for data in block or frame format**
 - >> Video
 - >> 2D matrix maps
- > **Advantages**
 - >> Low silicon overhead
 - >> Fast and deterministic latency



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



Summary



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
- > **Several ways to profile an application**
 - >> TCF profiler is the preferred tool for profiling in the SDSoC (over gprof)
 - Uses ARM's integrated hardware to gather performance statistics
 - >> 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 sds_lib usage

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

Adaptable.
Intelligent.

