Coding Considerations

SDx 2018.2

XILINX.

Objectives

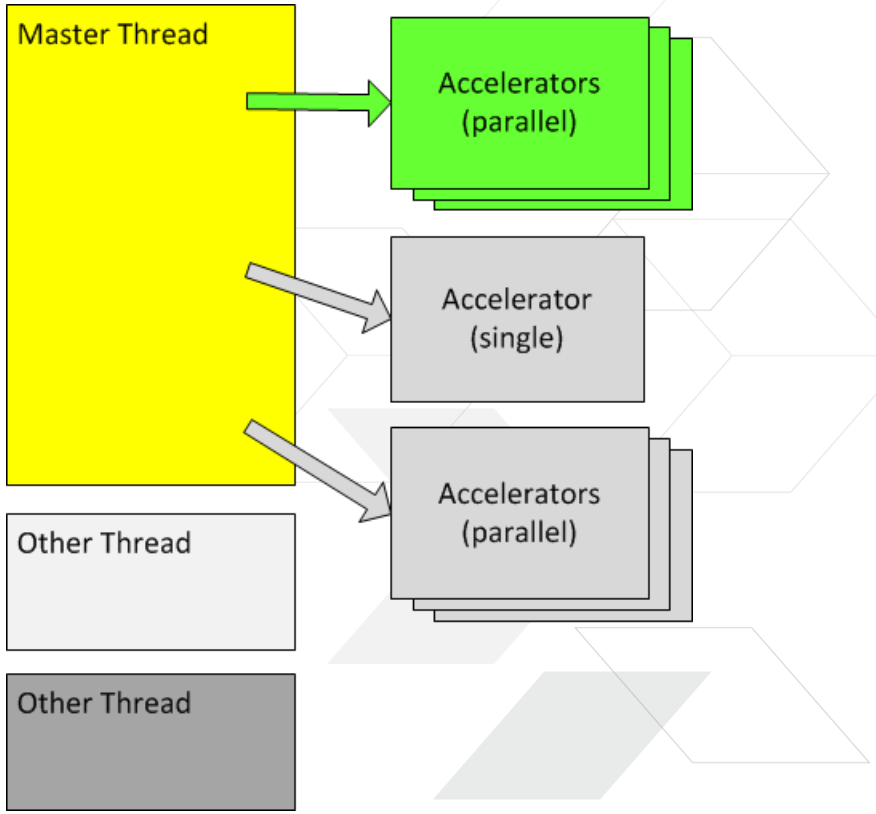
- **After completing this module, you will be able to:**
  - Identify proper coding styles
  - Use high-level techniques to find potential problem areas
  - Avoid common SDSoC tool-related issues

Outline

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

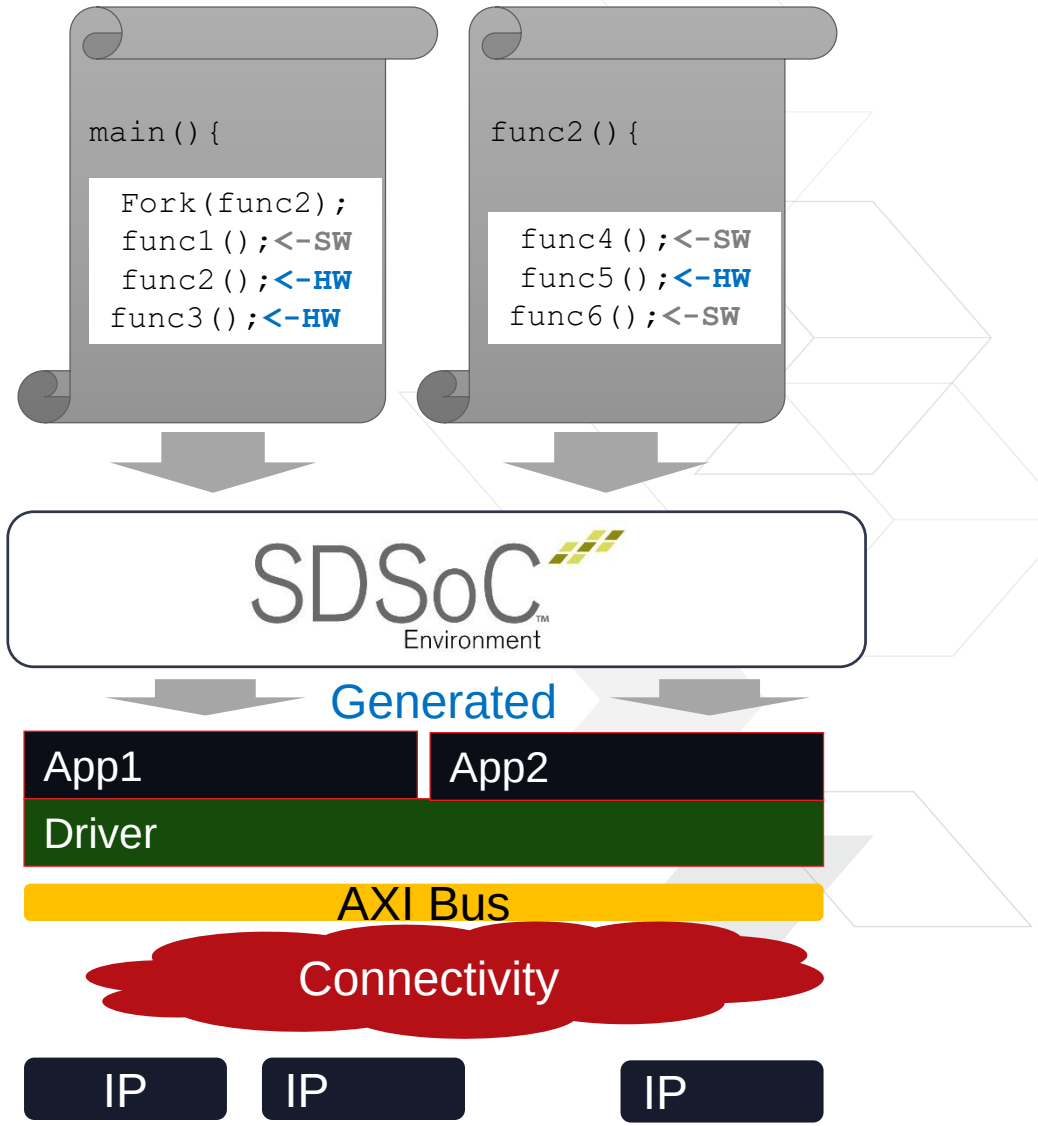
General C/C++ Guidelines

- 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



Multiple Threads / Processes with HW functions

- Multiple threads / processes moving functions in HW are supported in SDSoC
- Accelerators are disjoint between applications
- Generates optimal driver and connectivity architectures



XILINX.

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
- **It is an error to refer to a global variable within a hardware function or any of its sub-functions when this global variable is also referenced by other functions running in software**
- **Hardware functions support scalar types up to 1024 bits, including double, long long, packed structs, etc**

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 macros allow guard code with *#ifdef* and *#ifndef* preprocessor statements**
  - __SDSCC__ 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
```

Development Technique

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 the Estimate Performance
  - 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 the Estimate Performance 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 Estimate
  - 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 Estimate Performance with Debug configuration initially
    - Use Release configuration 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

SDSoC Tool-Related Issues

Diagnosing Estimate Performance Launch Failure

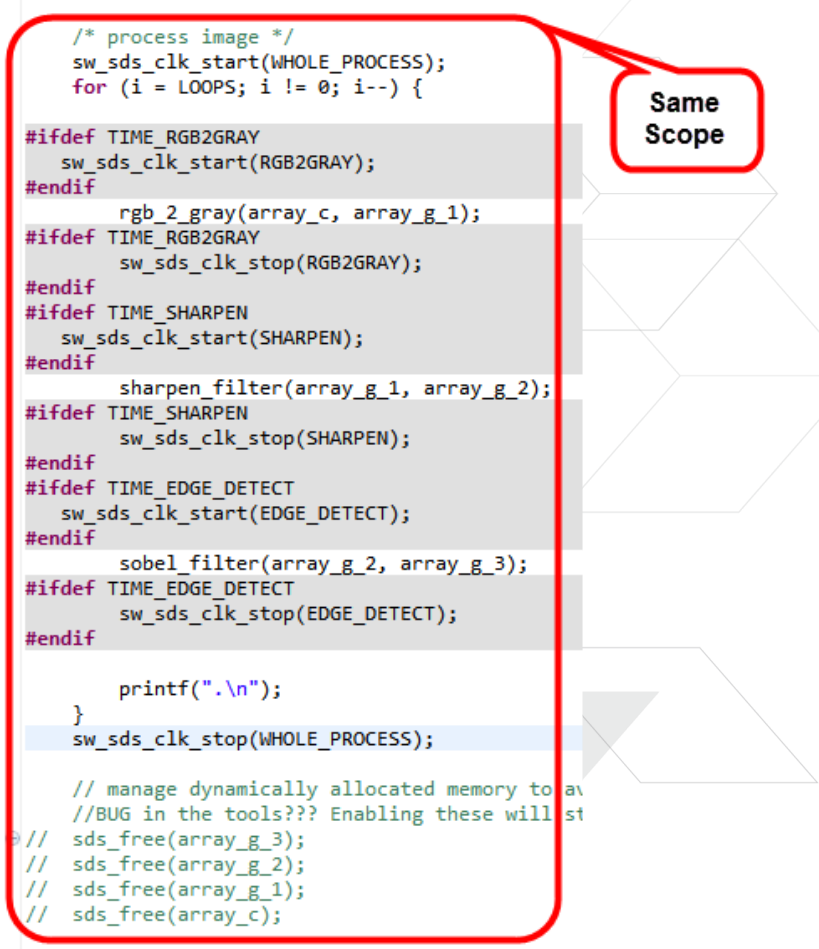
- **If performance estimation in the SDSoC development environment fails to generate a full report**
  - Ensure the development board is powered on and connected correctly
  - Open the Debug perspective
  - Disconnect and remove all debug sessions that are listed
  - Open the SDSoC perspective
  - Open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



XILINX.

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 Debug and Release 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

Summary

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**

Adaptable.
Intelligent.

XILINX