A. Introduction 2

1. SDSoC – Software Defined System on Chip 2

2. SD card image 2

3. Design Flow 3

a. How to invoke SDSoC ? 3

b. Makefile for SDSoC ? 3

B. SDSoC Compilation and System linking 6

1. SDSoC compilation and system linking process 6

2. Connectivity framework (CF) 6

3. Stub function 7

C. SDSoC-Based System Optimization 9

1. Memory allocation for efficient data transfer 9

2. Clock frequency 9

3. Task level pipelining 9

D. Libraries, Application modes 11

1. Using an IP library 11

2. Using HLS library 11

3. Standard Alone 11

E. File I/O 12

1. In Linux 12

2. Standard Alone mode 12

F. HLS-BASED CODING [Hardware Implementation] 13

1. Function Arguments – Accelerators’ interfaces 13

2. Data Mover – Port declaration 14

a. Choosing PS port 14

b. Choosing data mover 14

c. Choosing runtime data copy size between PS and PL 15

H. NOTES WHEN WORKING WITH BAREMETAL AND SDSoC 16

1. Manually programming FPGA and running application on Zedboard 16

2. Board setup 16

I. Basic of Make File 18

1. Comments 18

2. Rules 18

a. Phony target 19

b. Dependency lines (when to build a target) 19

c. Shell lines (or Recipes - how to build, update a target) 19

3. Macro 19

a. Static macro 20

b. Runtime Macro or Dynamic macro 20

# Introduction

## SDSoC – Software Defined System on Chip

* **A “***C/C++ to HDL function***” converter** (invokes ***Vivado HLS*** in background)
* **A C/C++ IDE** to compile source code to object code running on ARM CPU (invokes ***GNU tool chain*** in background)
* **A data mover instantiator** to transfer data between PS and PL

SDSoC linkers invokes tools within **Vivado Design Suite** to compile the system into bitstream

|  |
| --- |
| **Summary**  ***SDSoC development environment*** includes   * Vivado * Vivado HLS * SDSoC tools   + Eclipse/CDT –based GUI   + Command line tools   + ARM GNU toolchain |

## SD card image

* SDSoC generates a complete system running Linux, FreeRTOS or Standard Alone in SD card format.
* A SD card image (for Linux) consists
  + A boot image BOOT.BIN including
    - FPGA bit-stream
    - FSBL – First Stage Boot Loader
    - Boot program (Uboot)
  + Application binary (\*.elf)
  + Linux image
    - uImage
    - devicetree.dtb
    - uramdisk.image.gz

|  |
| --- |
| **Summary**  By adding **-target-os <…>** in Makefile, can direct SDSoC compiler generates target OS   * Linux (by default) * FreeRTOS freertos * [Baremetal](#_Standard_Alone_and_3) standalone |

## Design Flow

|  |
| --- |
| ../Desktop/Screen%20Shot%202016-10-01%20at%205.59.09%20AM.png |

### How to invoke SDSoC ?

SDSoC can be invoked from either:

* Command line (well-suited to scripting flows)
* Make file
* Eclipse-based GUI (interactive features to simplify development)

### Makefile for SDSoC ?

* A Make file drives the compilation and link process from *source code* to *full SD card image*
* Look at “[Basic of Make file](#_Basic_of_Make)” to know basic components of a Makefile

|  |
| --- |
| APPSOURCES = mmult.cpp mmult\_accel.cpp  EXECUTABLE = mmult.elf  SDSFLAGS = -sds-pf zed \  -sds-hw mmult\_accel mmult\_accel.cpp -sds-end \  -poll-mode 1  CC = sds++ ${SDSFLAGS}  CFLAGS = -Wall -O3 -c  LFLAGS = -O3 -sds-pf zed  OBJECTS := $(APPSOURCES:.cpp=.o)  .PHONY: all  all: ${EXECUTABLE}  ${EXECUTABLE}: ${OBJECTS}  ${CC} ${LFLAGS} ${OBJECTS} -o $@  %.o: %.cpp  ${CC} ${CFLAGS} $< -o $@  clean:  ${RM} ${EXECUTABLE} ${OBJECTS}  ultraclean: clean  ${RM} ${EXECUTABLE}.bit  ${RM} -rf \_sds sd\_card |
| F**igure 2: Makefile example** |

**Important parameters**

* **SDSoC flag**

|  |
| --- |
| **SDSFLAGS = -sds-pf** ***name\_of\_platform*** \  **-sds-hw** *funct\_name* *file\_name*  **-sds-end** \  **synchronization\_method \**  **… …** |

|  |  |
| --- | --- |
| -sds-pf <…> | specifies a target platform  **zc702**  **zed**  **zc706**  **microzed** |
| -sds-hw <…> -sds-end | specifies name of the top-level function to transfer into hardware and the file contains that function |
| -target-os <…> | Linux by default (do not need to specify)  **standalone**  **freertos** |
| -poll-mode 1 | Synchronization method |

*Example*

**SDSFLAGS = -sds-pf** zed **\**

**-sds-hw** mmult\_accel mmult\_accel.cpp **-sds-end \**

**-poll-mode 1**

*Remark*

* + Each top-level function, which is compiled for hardware, MUST reside in a separated file
  + That file can contain sub-functions of the top-level function
* **Compiler macro** *to target ARM CPU in Zyn*q

|  |
| --- |
| **CC = *compiler\_name*** **${SDSFLAGS}** |

*Supporting SDSoC compiler*

* + - **sdscc** #for compiling C
    - **sds++** #for compiling C++

*Example*

**CC =** sds++ **${SDSFLAGS}**

Remark

* Underneath the hood, SDSoC automatically invokes the arm-gcc compiler to target the ARM CPU
* Can invoke the arm-gcc compiler directly but not recommend

# SDSoC Compilation and System linking

## SDSoC compilation and system linking process

* *Step 1*

SDSoC compiler **calls Vivado HLS** to do cross-compile a function to programmable logic

* + SDSoC analyzes application code to ***determine data communication patterns and transfer requirement***
  + SDSoC ***builds an AXI-based data motion network*** in hardware based on above requirement. Using standard AXI IP for transport
* *Step 2*

Caller automatically integrates “device drivers” and additional code to transport data. (Note: Callee is the hardware component)

* *Step 3*

SDSoC generates [***connectivity description***](#_Connectivity_framework_(CF)) (channel-based data model) for system

* *Step 4*

SDSoC generates a “[***stub function***](#_Stub_function)”. SDSoC changes the “Caller code” to call the “stub function” instead of calling the function compiled into Hardware

* *Step 5*

SDSoC calls the **system linker** tool to ***generate*** “***Vivado IPI project/TCL***” from the connectivity description got from STEP 3, then to ***generate the bitstream***

* *Step 6*

SDSoC **calls the ARM GNU** compiler on the rest of the code and the “stub function” and links them with predefined SW API libraries ***to generate the ELF file***

* *Step 7*

***Generating a*** [***SD card image***](#_SD_card_image) including

* + the bitstream
  + the ELF file
  + the prebuilt Linux kernel image

## Connectivity framework (CF)

SDSoC design environment builds upon a Connectivity Framework

CF supports multilingual, heterogeneous computing

Connectivity framework consists of:

* An abstract “***channel-based data model***”. It is a high-level description of logical and physical connections between system components (Hardware and Software)
* Software APIs for data transfer and allocation
* A “***system linking***” tool for generating a Vivado-based hardware system

## Stub function

* What is a “Stub function”?

In the original code, data is passed to “hardware function” by arguments like normal function call.

However, in the implementation, obviously, data transfer must obey specific protocol. That why in the final implementation, application code will call a “***special function***” rather than original “hardware function”

That special function named “Stub function”

* “Stub function” helps to:
  + Synchronize control between CPU and Hardware
  + Transfer data between Hardware and Software components
* How could “Stub function” do so?

“Stub function” calls SW APIs defined by CF layer

* Important APIs

cf\_send\_i sends data to accelerators

cf\_receive\_i receives the result from the accelerator.

cf\_wait waits until a transaction completes

**Note**

* + A “stub function” only uses 3 above APIs to communicate btw HW and SW regardless of actual data communication method (Streaming or memory-map …)
    - * The code of “Stub function” is ***completely independent*** of the actual communication protocol
  + Underneath the hood, the implementation of above APIs is automatically generated based on communication protocol which is provided by the system\_linker tool

***Example of Stub Function***

* Firstly, the application sends commands to control the hardware component. Then, wait until this transaction completes
* Secondly, it sends data to ports of the hardware component
* Thirdly, it receives result from hardware component

|  |
| --- |
| ../Desktop/Screen%20Shot%202016-10-03%20at%206.36.06%20PM.png |

# SDSoC-Based System Optimization

## Memory allocation for efficient data transfer

* Standard C **malloc** allocates memory that is contiguous in the virtual memory space, but may not contiguous in the physical memory => High overhead for data transfer
* ***Solution***

SDSoC provides the following 2 APIs that encapsulate the contiguous memory allocation

sds\_alloc (size\_t size) - To allocate memory

sds\_free (void \*memptr) - To free the memory

Libraries

#include “stdlib.h” - Include first to provide the size\_t type

#include “sds\_lib.h” - For above APIs

## Clock frequency

Add following options to **sdscc/sds++** in the **makefile**

-clkid n - specifies the clock ID n should be used for hw accelerator

-dmclkid n - specifies the clk ID to use for data motion network

Note: SDSoC 2014.4 allows to specify only a single clock ID for both the hardware functions and the data motion network

## Task level pipelining

* SDSoC compilers allows to pipeline multiple calls to **an accelerator** to overlap the data transfer

|  |  |  |
| --- | --- | --- |
| ../Desktop/Screen%20Shot%202016-10-04%20at%209.50.12%20AM.png | **=>** | ../Desktop/Screen%20Shot%202016-10-04%20at%209.50.28%20AM.png |

* Underneath the hood, multiple buffers are generated (extra BRAM) to store data of overlapped calls
* Advantage: Reduce latency with small extra resource

***SDSoC pragma (2 important pragmas)***

#pragma SDS async(ID)

* The function call that immediately follows should be executed asynchronously
* The processor initiates the call but continues with its own execution rather than waiting for the call to finish

sds\_wait(ID) or #pragma SDS wait(ID)

* + Used to wait for an asynchronous call to complete
  + Assume: Tasks’ completions are in the same order as their issues
  + Each ID has one queue to keep results
* ***ID*** specifies the ***unique ID of the accelerator*** that is used to execute the call
  + The same ID is used in a subsequent wait statement to synchronize with the accelerator
  + Different ID will create different instances of the accelerator

***Note***

From PL side, a call (hardware instance) can access its inputs and write its outputs anytime between the start of the call and the corresponding wait

* In the application, the arguments of a call should be accessed ***only after*** the sds\_wait

***Example***

|  |
| --- |
| ../Desktop/Screen%20Shot%202016-10-04%20at%2011.14.37%20AM.png |

# Libraries, Application modes

## Using an IP library

* Step 1: MUST include header file of an IP in the source code
* Step 2: In the Make file,
  + ADD the path to the header file using the **–I** switch
  + LINK against the library, use the –L and –I switches

>> sdscc –c **–I<path to header>** -o main.o main.c

>>sdscc -${PLATFORM} ${OBJECT} **–L<path to lib>** **-l<lib>** -o <app elf file>

*Example*

|  |
| --- |
| CC = sdscc -sds-pf zc702  main.o: main.c  ${CC} -c -I./include $< -o $@  fir.elf: main.o  ${CC} main.o -L./lib/hw -lfir -lm -o $@ |

## Using HLS library

* Only need to INCLUDE header file of HLS library
* Do not need to link or add PATH like “Using IP library”

## Standard Alone

Changing in the makefile

CFLAGS = **-target-os** standalone

LFLAGS = **-target-os** standalone

Limitation:

* Does not support multi-threading, virtual memory or address protection
* File I/O: not using usual C APIs, but instead through a special API using libxilffs. MUST disable DCache before doing any file operation

# File I/O

## In Linux

Can use usual C APIs like normal linux applications.

## Standard Alone mode

Not using usual C APIs, but instead through a special API using **libxilffs**.

MUST disable DCache before doing any file operation

|  |
| --- |
| **#include** “ff.h”  **#include** “xil\_cache.h”  **FIL** fil\_in, fil\_out; // File obj  **FRESULT** Res; // File status  **char** \*SD\_in, \*SD\_out;  **char** FileName\_in[32] = "input.yuv";  **char** FileName\_out[32] = "output.yuv";  SD\_in = (char \*)FileName\_in;  SD\_out = (char \*)FileName\_out;  Xil\_DCacheFlush();  ***Xil\_DCacheDisable***();  Res = ***f\_open***(&fil\_in, SD\_in, ***FA\_OPEN\_EXISTING*** | ***FA\_READ***);  Res = ***f\_open***(&fil\_out, SD\_out, ***FA\_CREATE\_ALWAYS*** | ***FA\_WRITE*** | ***FA\_READ***);  ***f\_read***(&fil\_in, y[frame][row], bytes\_per\_cc \* cols, &NumBytesRead);  ***f\_close***(&fil\_in);  ***f\_close***(&fil\_out); |

# HLS-BASED CODING [Hardware Implementation]

## Function Arguments – Accelerators’ interfaces

Each argurment of a hardware function will be transferred to a corresponding AXI interface.

* **AXI stream – Array argument**
  + Array arguments are mapped to either
    - Ap\_fifo
    - Bram interfaces
  + Limitation of SDSoC v2014.4: it be able to automatically transfers up to SIXTEEN array arguments (8 inputs, 8 outputs) to AXIS
  + If #input or #output arguments > 8 => MUST explicitly code axis interfaces in your HLS code

|  |
| --- |
| ../Desktop/Screen%20Shot%202017-01-23%20at%2010.05.45%20AM.png |

* Code in SDSoC includes 2 parts:
  + One is used by Vivado HLS to generate Hardware
  + One is used by GNU toolchain to simulate functional behavior of system
* Union {} is the way to convert from unsigned type to float type with low cost.
* **AXI memory map – Array argument**
  + NOTE: every HLS function containing an arrgument that maps to an AXIMM MASTER requires **a return value** or **other output scalar**

|  |
| --- |
| ../Desktop/Screen%20Shot%202017-01-23%20at%2010.47.04%20AM.png |

* –offset : to apply an address offset
  + off: Does not apply an offset address. This is the default.
  + direct: Adds a 32-bit port to the design for applying an address offset.
  + slave: Adds a 32-bit register inside the AXI4-Lite interface for applying an address offset.
* –depth : size of memory’s chunk which AXIMM will access
* Do not need to declare the base address, it will be handled automatically by the tool
* **AXI Lite – Scalar argument** 
  + DO NOT NEED to use any pragmas for specifying AXI LITE interfaces
  + SDSoC automatically inserts the axis\_accelerator\_adapter for axilite control

## Data Mover – Port declaration

Following pragmas are put in the header file \*.h above hardware function.

### Choosing PS port

|  |
| --- |
| **#pragma** SDS data sys\_port(input: AFI, output: ACP) |

* + AFI : HP0- 3 (Asynchronous FIFO Interface – High Performance port)
  + ACP: ACP port
  + Do not need to specify for GP port

### Choosing data mover

|  |
| --- |
| **#pragma** SDS data data\_mover(input: AXIDMA\_SG, output: AXIDMA\_SIMPLE, out:AXIFIFO) |

* AXIDMA\_SG : Axi DMA with Scatter Gather mode
* AXIDMA\_SIMPLE: Axi DMA with Simple mode
* AXIFIFO: data mover is assigned to the M\_AXI\_GPx port

Note: Data mover will be specified indirectly by specify memory allocation method.

* + Malloc() : AXIDMA\_SG
  + Sds\_alloc() : AXIDMA\_SIMPLE

### Choosing runtime data copy size between PS and PL

|  |
| --- |
| **#pragma** SDS data copy(0:N) |

It is used together with AXIS, it specifies the size of transferred data in runtime.

**Example**: In the header file,

|  |
| --- |
| ../Desktop/Screen%20Shot%202017-01-23%20at%2011.35.26%20AM.png |

**Important note**:

* In version 2014.4, to transfer data which has a size > 8MB, we need to configure DMA under Scatter Gather mode; while allocating data with sds\_alloc API

# NOTES WHEN WORKING WITH BAREMETAL AND SDSoC

## Manually programming FPGA and running application on Zedboard

Sometimes, SDK or SDSoC cannot program FPGA or run the application due to some   
    unknown problems. Here is the way to overcome it

**Three important files** are needed to run an application on Zedboard

* **\*.bit**                   bitstream file including hardware configuration of the Zynq PL
* **\*.elf**                   execution file of the application
* **ps7\_init.tcl**        initialization file for the Zynq PS

      Locating in workspace of the project.

Find their relative paths by these instructions

*find . -name \*.bit*

*find . -name \*.elf*

*find . -name ps7\_init.tcl*

    Manually programming FPGA and running application on Zedboard by following this

    procedure

source  **[vivado\_setting.sh file]**

-- open Xilinx Microprocessor Debugger

xmd

-- program hardware

fpga -f  **[path\_to\_bitstream\_file]**

-- connect debugging system

connect arm hw

-- initialization for zynq ps

source  **[path\_to\_tcl file]**

ps7\_ init

-- load application to memory through data cable ()

dow  **[path\_to\_elf\_file]**

-- run application

run

## 2. Board setup

Jumpers need to be set correctly based on running mode of applications

***Standard Alone***

|  |
| --- |
|  |

***Linux***

|  |
| --- |
|  |

**W**hen working with minicom to communicate with UART port

* Must run as system administrator
* To know which port are connecting with UART using  *dmesg* command
* Should turn off Hardware/Software control flow
* Baud rate for UART1 port of Zedboard, by default, is **115200**

*sudo minicom -D /dev/ttyACM0  -b 115200 -s*

*Choose Serial Port Setup > Turn off HW/SW flow control*

# Basic of Make File

A ***Makefile*** composes from following components:

* Comments
* Rules
* Directives
* Macro
* Response files

## Comments

|  |
| --- |
| **# comments …** |

*Syntax*

*Example*

# Makefile for Opus Make 6.1

#

# Compiler: Microsoft C 6.0

# Linker: Microsoft Link 5.10

## Rules

Tells “MAKE” both ***when*** and ***how*** to make a file

Classification:

* Explicit rules: Supplied explicitly in the Makefile
* Inference rules: Generalize the make process

|  |
| --- |
| **target** : **prerequisites**  **recipes**  ... |

*Syntax*

*Example*

main.o : main.c defs.h

cc -c main.c

*Explain*

|  |  |  |
| --- | --- | --- |
| **Target** *can be either* | **Prerequisite** | **Recipe or Shell lines** |
| * Name of a file that is generated by a program:   + Executable or object files * An action to carry out, such as ‘***clean’***, called PHONY target | Inputs to create target   * Sources | * Action that make carries out. * A recipe may have more than one command, either on the same line or each on its own line. |

### Phony target

Targets are just actions (do not refer to files) are called phony target

### Dependency lines (*when* to build a target)

*Syntax*

|  |
| --- |
| **Target** : ***prerequisites*** |

*Example*

project.exe : main.obj io.obj

At runtime, project.exe is rebuilt WHEN timestamp of main.obj or io.obj is newer than project.exe

*Remark*

* “Make process” is recursive since, it, firstly, check timestamp of sources => Ensure sources always be updated prior target
* Additional dependencies,

|  |  |
| --- | --- |
| Example | Or using additional dependencies |
| main.obj : main.c def.h  io.obj : io.c def.h | main.obj : main.c  io.obj : io.c  main.obj io.obj : def.h |

### Shell lines (or Recipes - how to build, update a target)

* **Need** to put **a tab character at the beginning of every recipe line!**
  + To change prefix with a character other than tab, can set the **.RECIPEPREFIX** variable
* Bear in mind: “MAKE” does not know anything about how the recipes work.

All “MAKE” does is **execute** the recipe when the target file needs to be updated

*Example*

main.o : main.c defs.h

cc -c main.c

When either timestamp of main.c or defs.h is newer than timestamp of main.o, main.o needs to be updated. “Make” will execute the command   
cc -c main.c

## Macro

Assignment symbol: “=”

Calling macro: “$(…)”

or “${…}”

To replace repeated text

### Static macro

|  |
| --- |
| ***Without using Macro*** |
| project.exe : main.obj io.obj  tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib  main.obj : main.c  bcc –ms –c main.c  io.obj : io.c  bcc –ms –c io.c  main.obj io.obj : def.h |
| ***Using Macro*** |
| # Macro declaration  OBJS = main.obj io.obj  MODEL = s  CC = bcc  CFLAGS = –m$(MODEL)  # Makefile content  project.exe : $(OBJS)  tlink c0$(MODEL) $(OBJS), project.exe,, c$(MODEL) /Lf:\bc\lib  main.obj : main.c  $(CC) $(CFLAGS) –c main.c  io.obj : io.c  $(CC) $(CFLAGS) –c io.c  $(OBJS) : incl.h |

**Remark**

* Macro can be declared on the Command Line

Example:

make CFLAGS=–ms

or make "CFLAGS=-ms -z -p" # macro containing spaces must be

# enclosed in a bracket

### Runtime Macro or Dynamic macro

Value of these macros is set dynamically

.TARGET #return current target

.SOURCE #return the first source of the explicit sources (from an inference rule)

.SOURCES #list all sources