Lab #1
Introduction to Zynq & Xilinx Vitis

ECE4810J System-on-Chip Design

Yihua Liu
UM-SJTU Joint Institute
ayka_tsuzuki@sjtu.edu.cn
Sept 19, 2022

Overview

Overview

① Lab 1: Xilinx Vitis on ZYNQ FPGA

② Lab 2: Xilinx Vitis HLS & PYNQ on ZYNQ FPGA

③ Lab 3: Xilinx Vitis HLS Optimization on ZYNQ FPGA

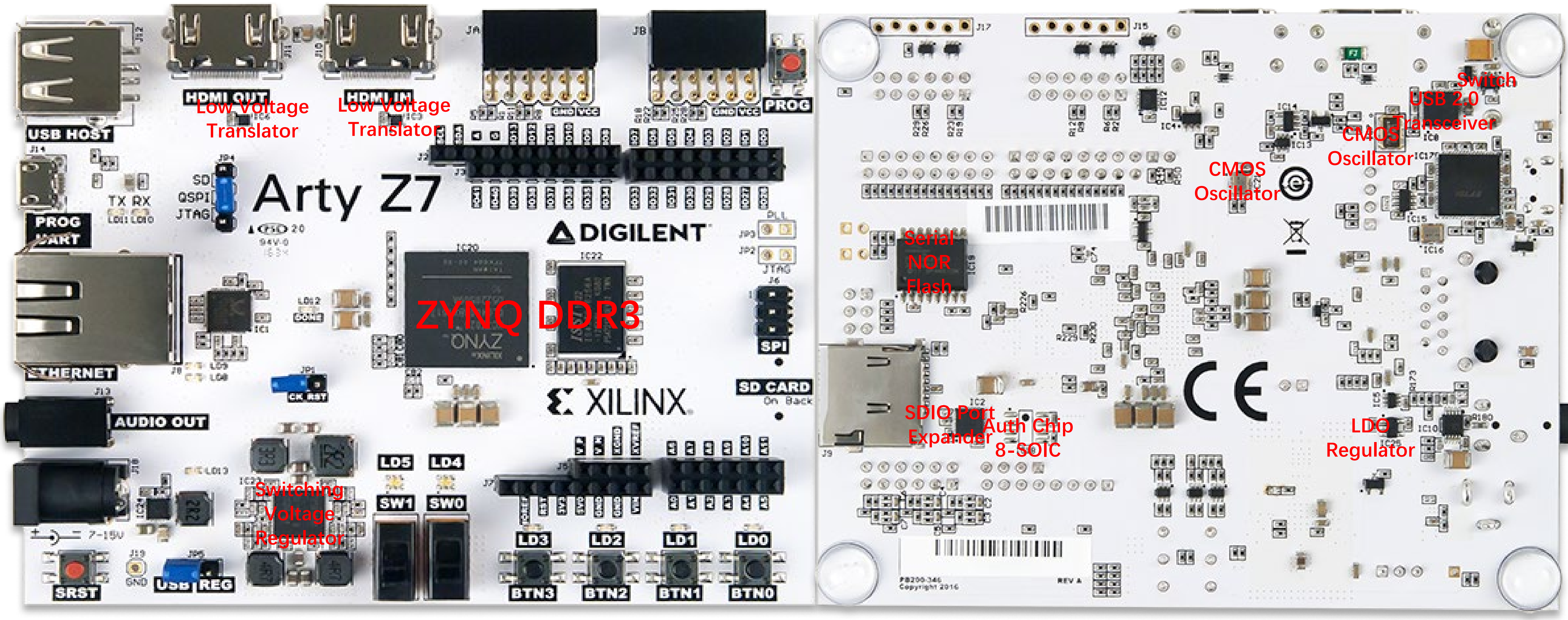
④ Lab 4: ASIC Design Flow I

⑤ Lab 5: ASIC Design Flow II

⑥ Lab 6: ASIC Design Flow III

Reminder: Active involvement in projects & labs, including **Piazza** and **lab attendance**, will be counted into **Participation** grades

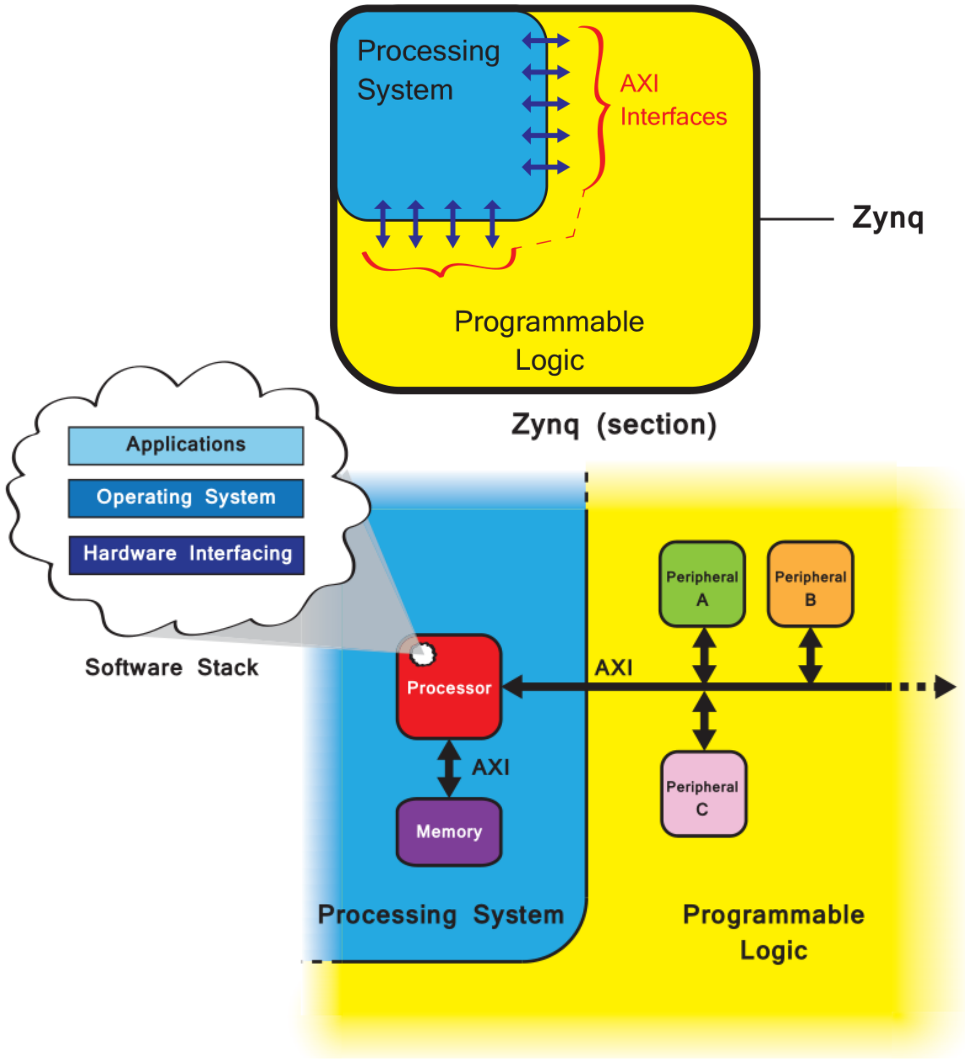
Arty-Z7


HDMI OUT
Low Voltage Translator
HDMI IN
Low Voltage Translator
JA
JB
PROG
USB HOST
SD
QSPI
JTAG
Arty Z7
DIGILENT
TX RX
PROG UART
ETHERNET
ZYNQ
DDR3
SPI
SD CARD On Back
XILINX
CK RST
AUDIO OUT
Switching Voltage Regulator
LD5
LD4
SW1
SW0
LD3
LD2
LD1
LD0
BTN3
BTN2
BTN1
BTN0
SRST
USB REG
Serial NOR Flash
SDIO Port Expander
Auth Chip 8-SOIC
CMOS Oscillator
CMOS Oscillator
Switch
USB 2.0 Transceiver
LDO Regulator

Zynq

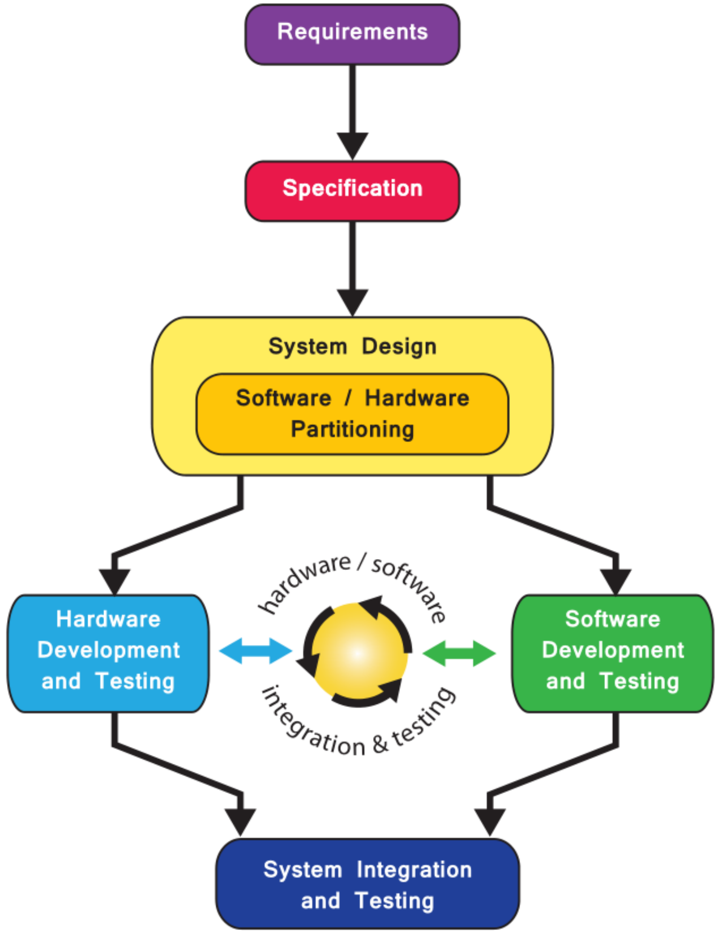
Zynq comprises two main parts: a Processing System (PS) formed around a dual-core ARM Cortex-A9 processor, and Programmable Logic (PL), which is equivalent to that of an FPGA. It also features integrated memory, a variety of peripherals, and high-speed communications interfaces. The PL section is ideal for implementing high-speed logic, arithmetic and data flow subsystems, while the PS supports software routines and/or operating systems. Links between the PL and PS are made using industry standard Advanced eXtensible Interface (AXI) connections.

Fig 1. A simplified model of the Zynq architecture



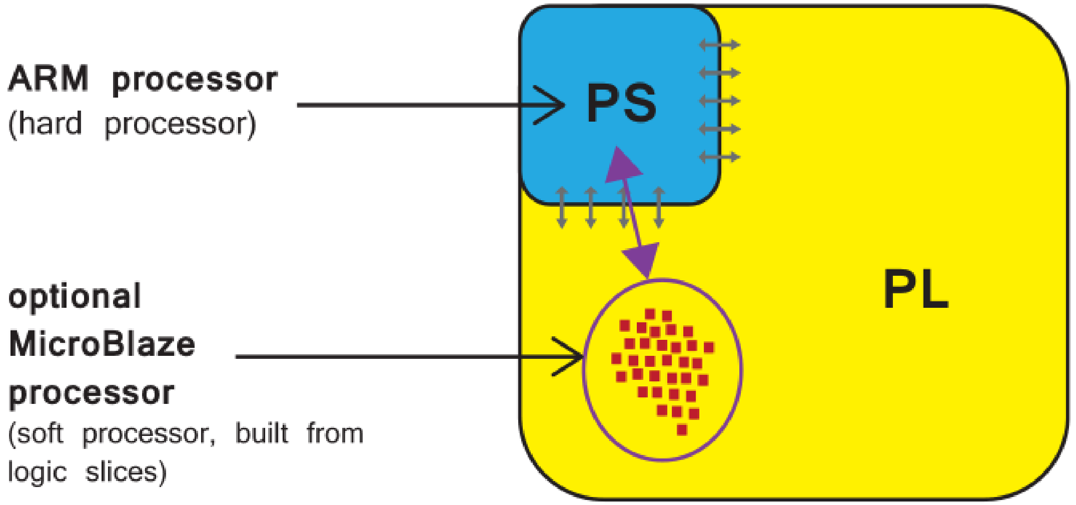
Zynq

Fig 2. A basic model of the design flow for Zynq SoC


Requirements
Specification
System Design
Software / Hardware Partitioning
Hardware Development and Testing
hardware / software
integration & testing
Software Development and Testing
System Integration and Testing

Design Reuse:
Intellectual Property (IP) functional blocks — corresponding to the peripheral components seen in Fig. 1, can be sources in different ways.

Fig 3. Locations of hard (ARM Cortex-A9) and soft (MicroBlaze) processors on a Zynq device


ARM processor
(hard processor)
PS
PL
optional MicroBlaze processor
(soft processor, built from logic slices)

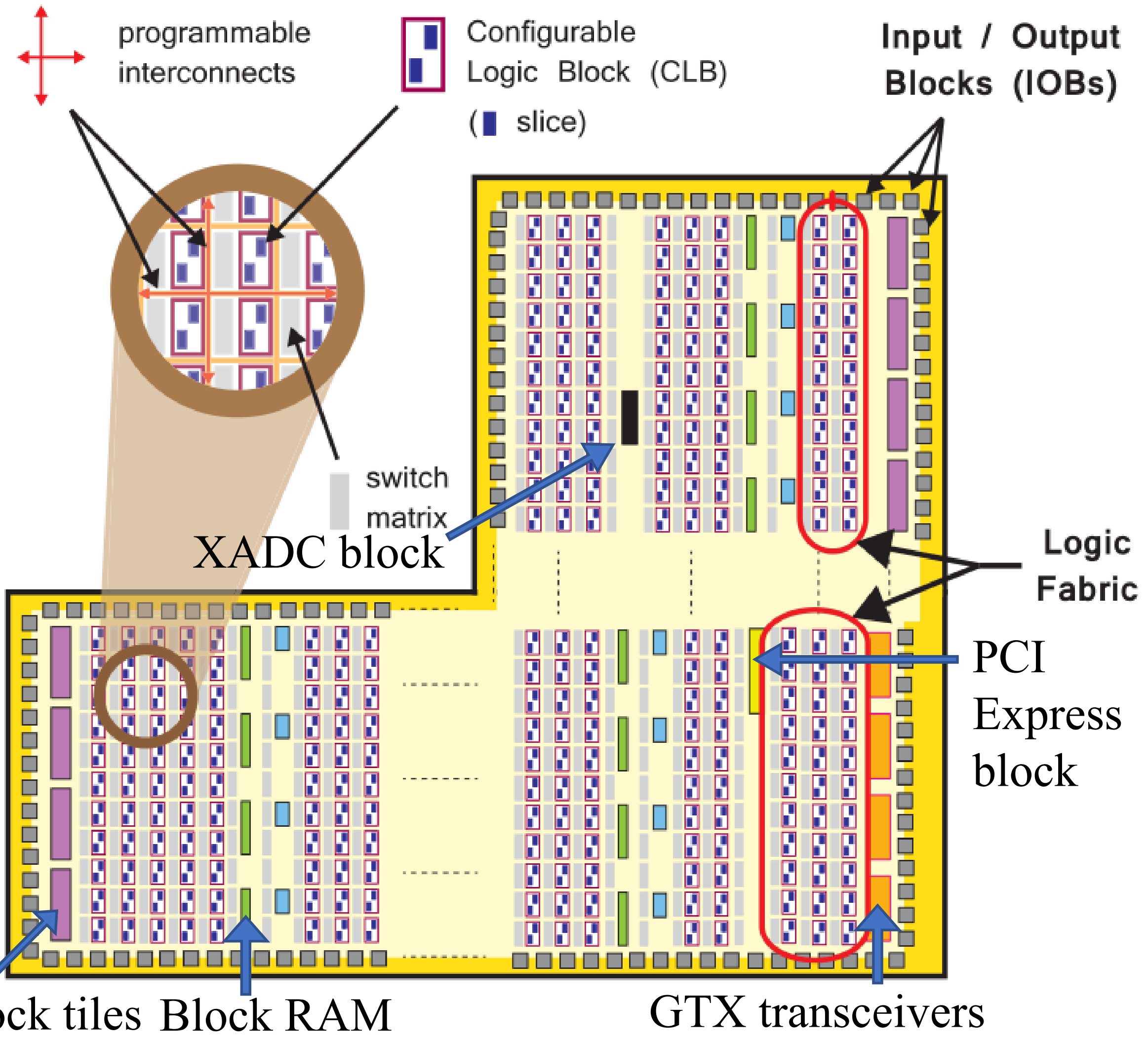
Zynq

Features of PL:

- Configurable Logic Block (CLB)
- Slice
- (6-input) Lookup Table (LUT) 53200
- Flip-flop (FF) 106400
- Switch Matrix
- Carry logic
- Input / Output Blocks (IOBs)

DSP48E1: high-speed arithmetic 220
Block RAMs (BRAMs): can implement RAM, ROM, and FIFO buffers 280
GPIO / SelectIO: HP 200 / HS 0
GTX Transceivers: comm interface blocks
XADC, Clocks, and JTAG for debug

Fig 4. The logic fabric and its constituent elements



Zynq

IP-XACT:
The IP-XACT standard is an eXtensible Markup Language (XML) schema for documenting IP using metadata, developed by the SPIRIT Consortium, now managed by Accellera Systems Initiative.
First-Stage Bootloader (FSBL):
The FSBL is a section of code contained in the MBR (Master Boot Record)

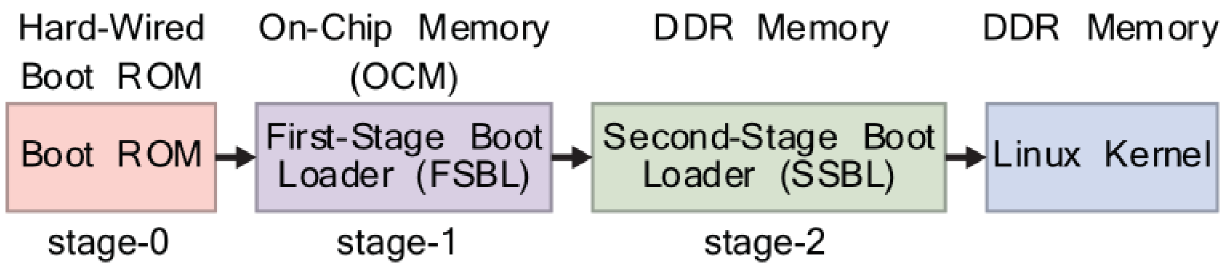


Fig 5. Zynq Linux boot process

Possible boot sources:
NAND Flash, NOR Flash, QSPI Flash, SD Card, JTAG
The FSBL is loaded into the OCM (On-Chip Memory) by the boot ROM after the initial boot period.
BootGen: a standalone app to generate bootable images

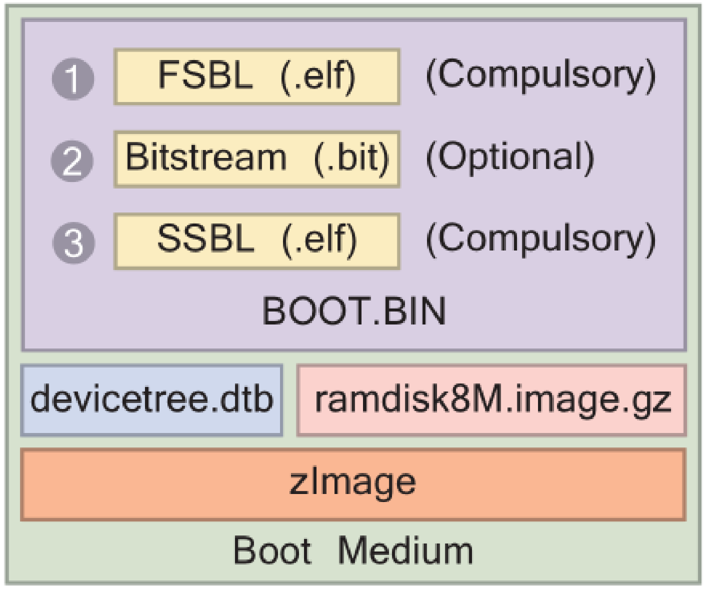

1 FSBL (.elf) (Compulsory)
2 Bitstream (.bit) (Optional)
3 SSBL (.elf) (Compulsory)
BOOT.BIN
devicetree.dtb
ramdisk8M.image.gz
zImage
Boot Medium

Fig 6. Required files for Zynq Linux boot medium

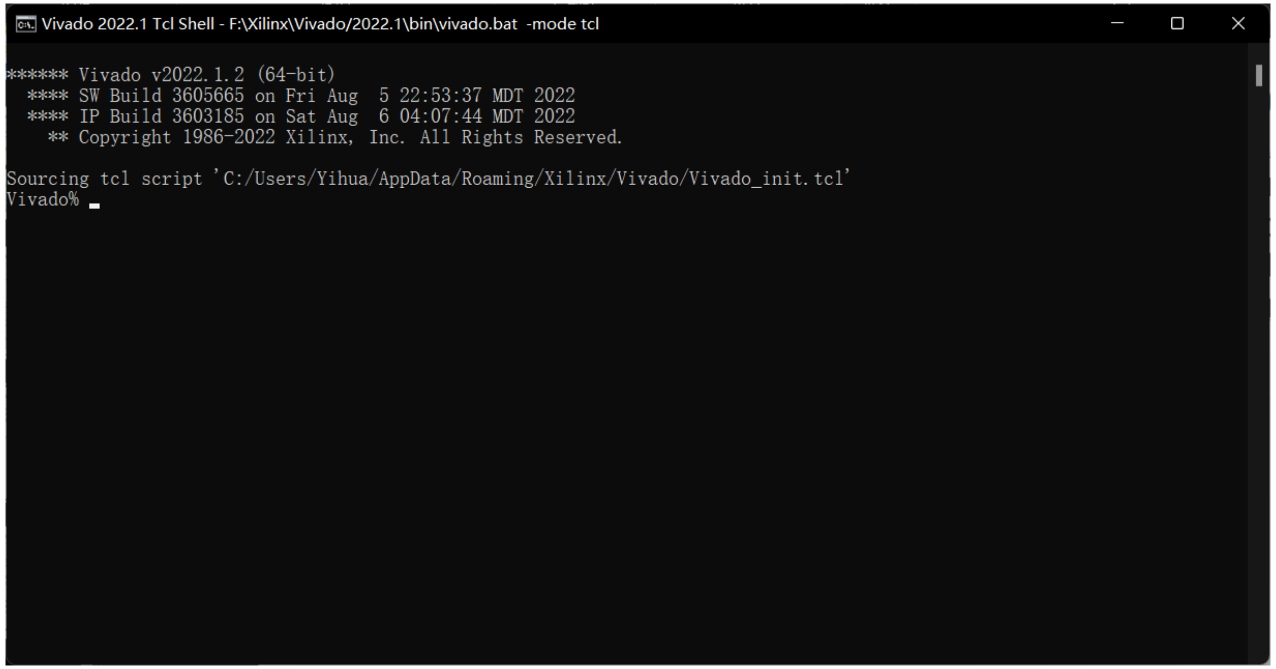
Tcl

Tcl (pronounced “tickle”) is a high-level, general-purpose, interpreted, dynamic programming language created in 1988. The popular combination of Tcl with the Tk extension is referred to as Tcl/Tk and enables building a GUI natively in Tcl. Tcl/Tk is included in the standard Python installation in the form of Tkinter.

Tclsh is bundled in some distributions of Windows/MacOS/Linux; if not, install ActiveTcl.

You will use Tcl in *Vivado 2022.1 Tcl Shell* most of the time.

Please read and play *Reduced Tcl Tutorial All-in-One.pdf* on Canvas Files->Reading Materials ->Literatures up to Chapter *String*


Vivado 2022.1 Tcl Shell - F:\Xilinx\Vivado/2022.1\bin\vivado.bat -mode tcl

```
****** Vivado v2022.1.2 (64-bit)
  **** SW Build 3605665 on Fri Aug  5 22:53:37 MDT 2022
  **** IP Build 3603185 on Sat Aug  6 04:07:44 MDT 2022
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.

Sourcing tcl script 'C:/Users/Yihua/AppData/Roaming/Xilinx/Vivado/Vivado_init.tcl'
Vivado%
```

XSCT

`connect -host localhost -port 3121`	
`connect -url tcp:localhost:3121`	
	Connect to hw_server/TCF agent on host localhost and port 3121
`disconnect <channel-id>`	Disconnect from specified channel
`targets`	List all targets
`targets -set -filter {name =~ "MicroBlaze*"} -index 0`	
	Set current target to target with name starting with "MicroBlaze" and which is on the first JTAG device
`rrd usr r8`	Read register r8 in group usr
`rwr usr r8 0x0`	Write 0x0 to register r8 in group usr

`state`: return state `stop`: suspend `stp`: step into `nxt`: step over `con`: resume

`openhw/closehw`	Open/Close a hardware design from Vivado
`getws/setws`	Get/Set Vitis workspace
`importprojects <path>`	Import projects to workspace

`app: build, clean, config, create, list, remove, report, switch`

XGpio

You may use the following functions of `xgpio.h` in Lab #1:

```
int   XGpio_Initialize (XGpio *InstancePtr, u16 DeviceId)
```
Initialize the XGpio instance provided by the caller based on the given DeviceID.

```
XGpio_Config *    XGpio_LookupConfig (u16 DeviceId)
```
Lookup the device configuration based on the unique device ID.

```
int   XGpio_CfgInitialize (XGpio *InstancePtr, XGpio_Config *Config,
                  UINTPTR EffectiveAddr)
```
Initialize the XGpio instance provided by the caller based on the given configuration data.

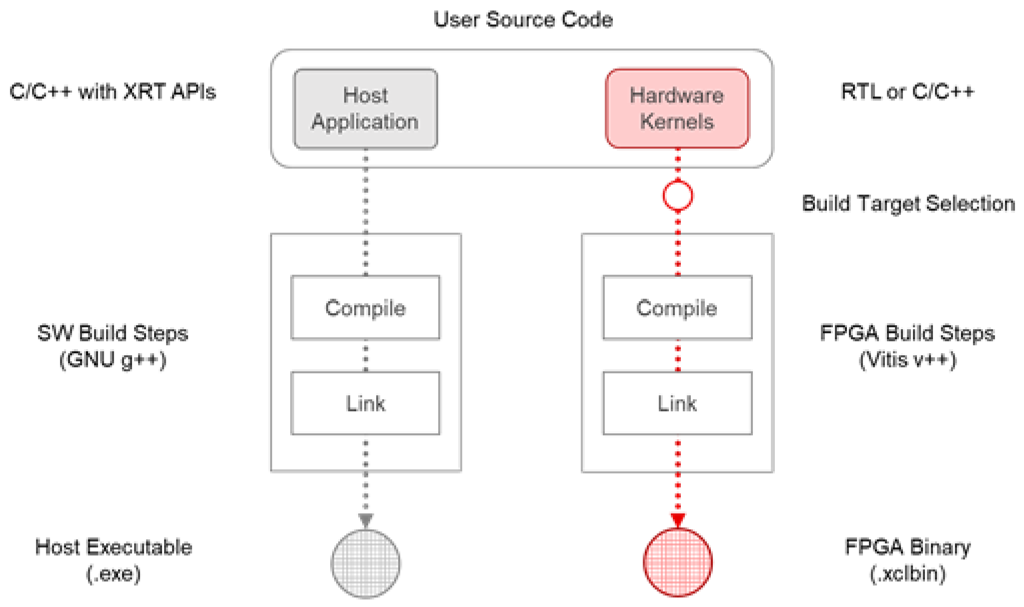
```
void  XGpio_SetDataDirection (XGpio *InstancePtr, unsigned Channel, u32
                  DirectionMask)
```
Set the input/output direction of all discrete signals for the specified GPIO channel.

```
u32   XGpio_DiscreteRead (XGpio *InstancePtr, unsigned Channel)
```
Read state of discretes for the specified GPIO channel.

```
void  XGpio_DiscreteWrite (XGpio *InstancePtr, unsigned Channel, u32 Mask)
```
Write to discretes register for the specified GPIO channel.

Xilinx Vitis

Fig 7. Vitis Build Process



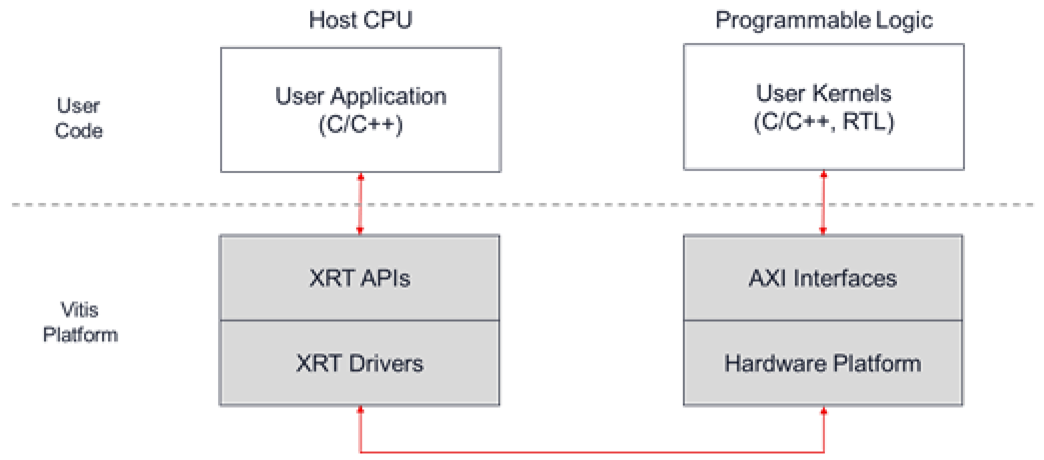
XRT: Xilinx Runtime library

Vitis Build Targets:

- Software Emulation (Linux only): `v++ -t sw_emu`
  Kernel code compiled to run on the host processor
- Hardware Emulation (Linux only): `v++ -t hw_emu`
  Kernel code compiled into RTL run in simulator

ERROR: [v++ 82-987] Sorry, emulation flows are not yet supported on this operating system.

- Hardware: `v++ -t hw`
  Kernel code compiled into RTL implemented on FPGA



Reference

① Arty Z7 - Digilent Reference

② Arty Z7 Reference Manual - Digilent Reference

③ The Zynq Book: Embedded Processing with the ARM® Cortex®-A9 on the Xilinx® Zynq®-7000 All Programmable SoC

④ UG1400 Vitis Unified Software Platform Documentation: Embedded Software Development (v2022.1)

⑤ gpio: xgpio.h File Reference (xilinx.github.io)

⑥ System Design with HDL Code Generation from MATLAB and Simulink - MATLAB & Simulink (mathworks.com)

⑦ Basic HDL Code Generation and FPGA Synthesis from MATLAB - MATLAB & Simulink (mathworks.com)

⑧ Generate HDL Code from Simulink Model - MATLAB & Simulink (mathworks.com)

⑨ Using the Zynq SoC Processing System — Embedded Design Tutorials 2022.1 documentation (xilinx.github.io)