Lab #6: OS + FPGA System

04/12/2018

4190.309A: Hardware System Design
(Spring 2018)

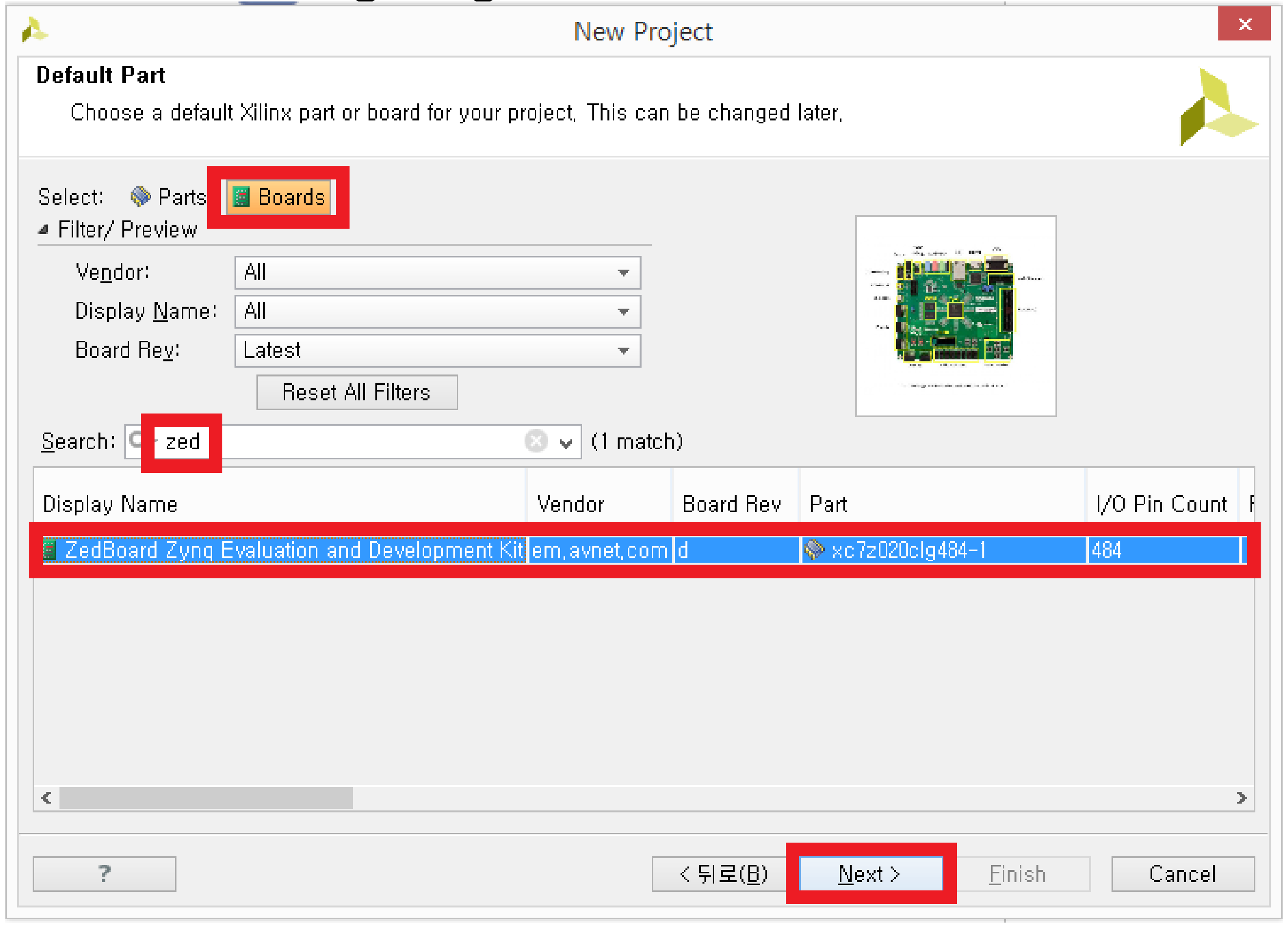
Overview

- Vivado Block Design
  - Processing System + BRAM + Connectivity
- FPGA + Linux (Debian Linux)
  - Access BRAM via C program
- Practice
  - Run a sample project provided by TA and explain what’s happening

Tutorial: Vivado Block Design

Create Vivado project

- Choose part or board
  - We are going to use ZedBoard



New Project

Default Part

Choose a default Xilinx part or board for your project. This can be changed later.

Select: Parts [✓ Boards]

Filter/ Preview

Vendor: All

Display Name: All

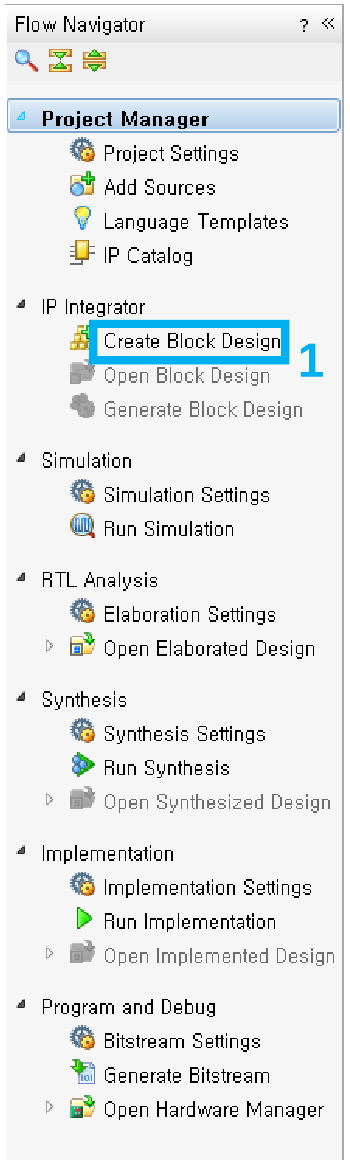
Board Rev: Latest

Reset All Filters

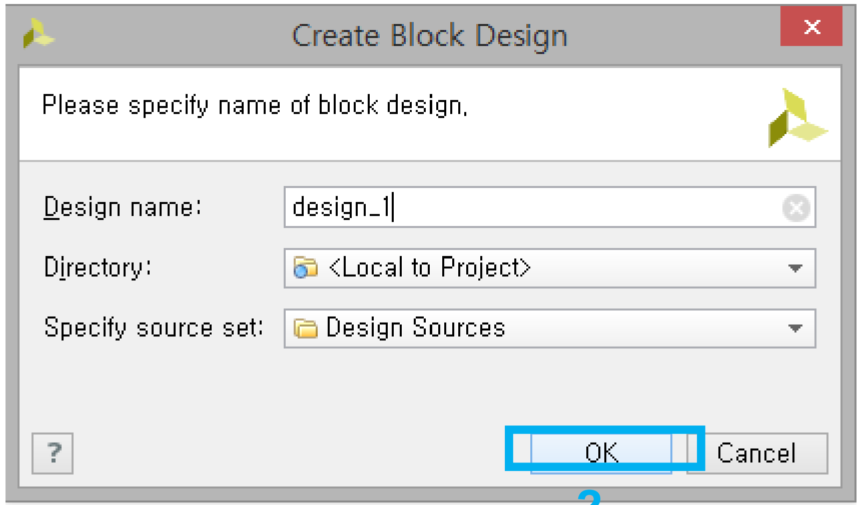
Search: zed (1 match)

Display Name	Vendor	Board Rev	Part	I/O Pin Count
ZedBoard Zynq Evaluation and Development Kit	em.avnet.com	d	xc7z020clg484-1	484

Vivado Block Design - Processing System(1)


Flow Navigator
Project Manager
Project Settings
Add Sources
Language Templates
IP Catalog
IP Integrator
Create Block Design
Open Block Design
Generate Block Design
Simulation
Simulation Settings
Run Simulation
RTL Analysis
Elaboration Settings
Open Elaborated Design
Synthesis
Synthesis Settings
Run Synthesis
Open Synthesized Design
Implementation
Implementation Settings
Run Implementation
Open Implemented Design
Program and Debug
Bitstream Settings
Generate Bitstream
Open Hardware Manager

1

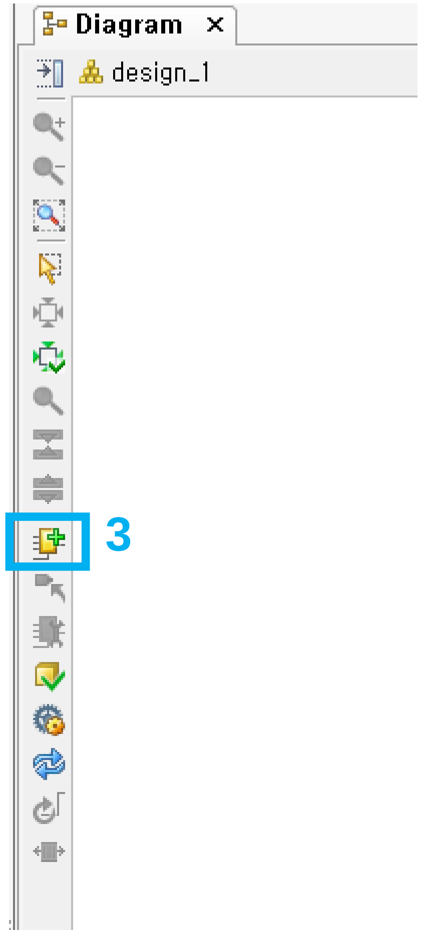

Create Block Design

Please specify name of block design.

Design name: design_1
Directory: <Local to Project>
Specify source set: Design Sources

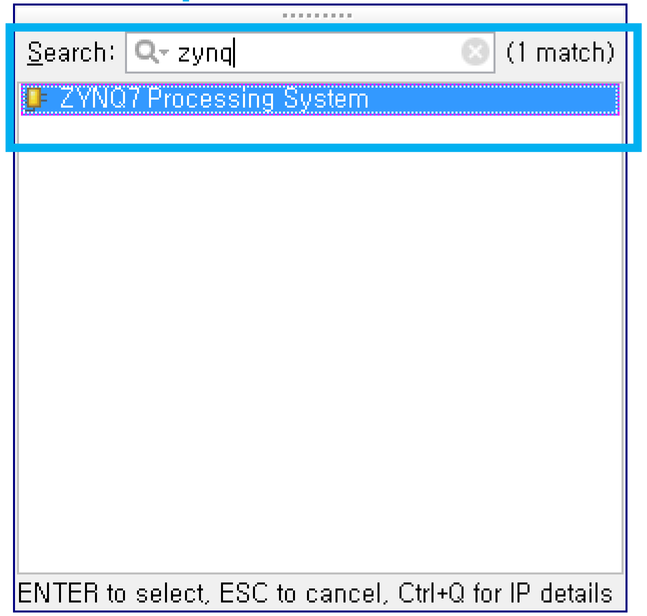
OK Cancel

2

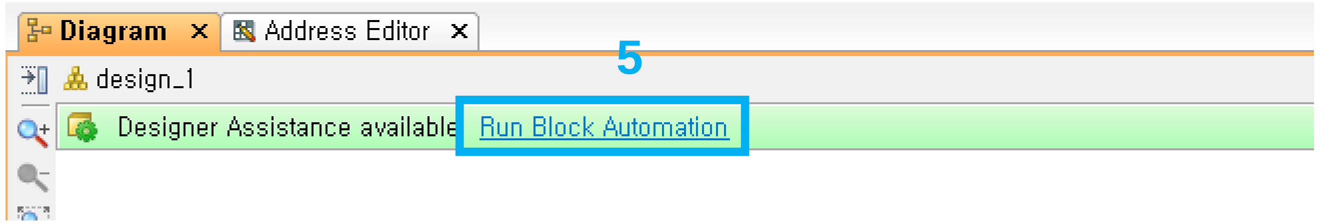


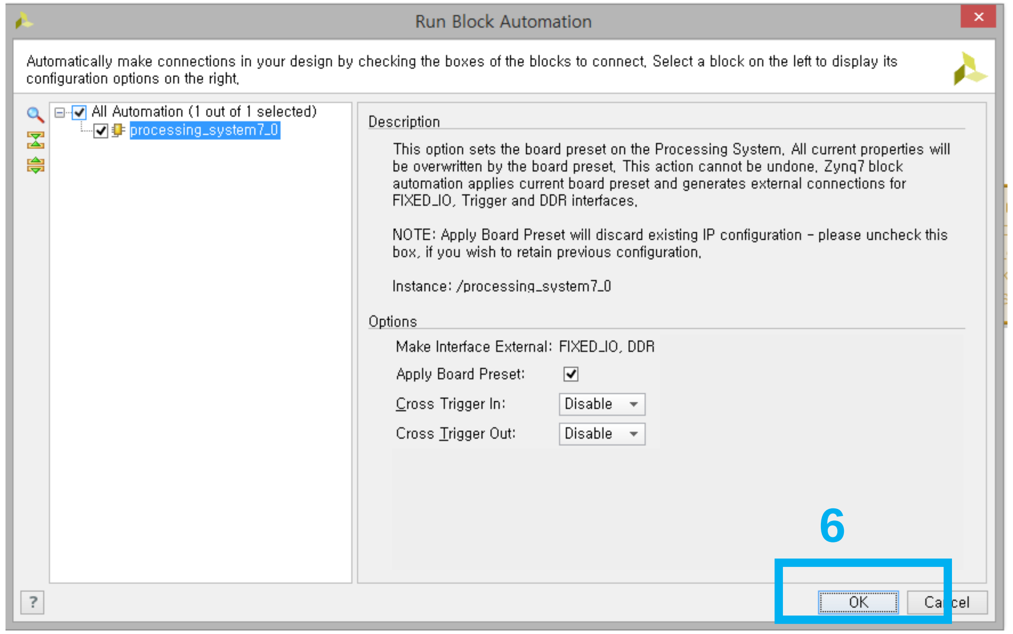
3

4


Search: zynq (1 match)
ZYNQ7 Processing System
ENTER to select, ESC to cancel, Ctrl+Q for IP details

Vivado Block Design - Processing System(2)




Run Block Automation

Automatically make connections in your design by checking the boxes of the blocks to connect. Select a block on the left to display its configuration options on the right.

All Automation (1 out of 1 selected)
processing_system7_0

Description

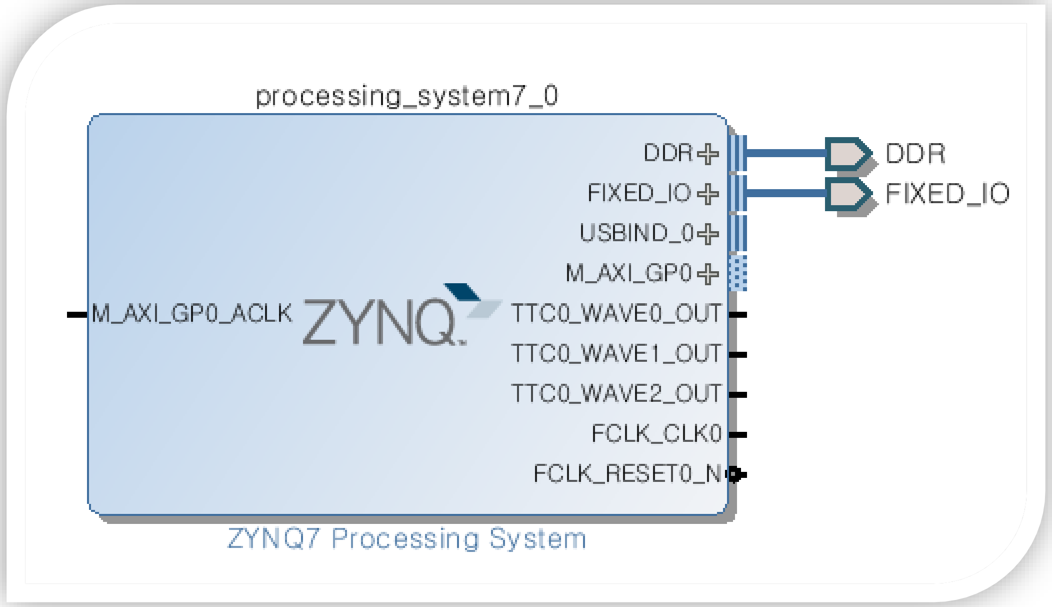
This option sets the board preset on the Processing System. All current properties will be overwritten by the board preset. This action cannot be undone. Zynq7 block automation applies current board preset and generates external connections for FIXED_IO, Trigger and DDR interfaces.

NOTE: Apply Board Preset will discard existing IP configuration - please uncheck this box, if you wish to retain previous configuration.

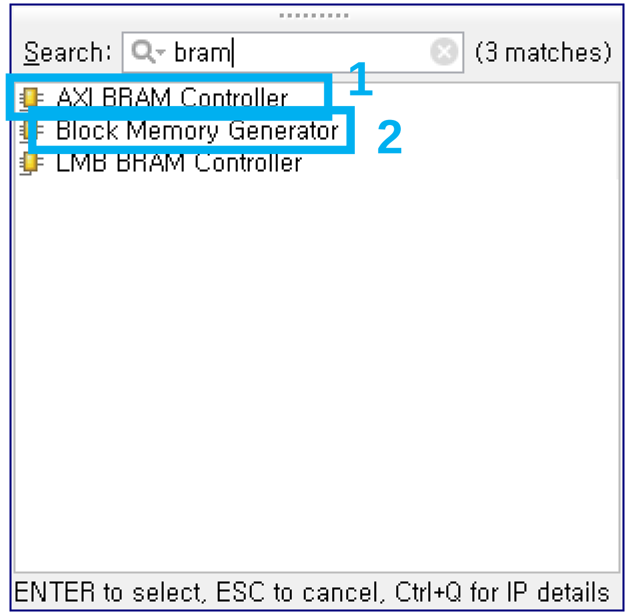
Instance: /processing_system7_0

Options

Make Interface External: FIXED_IO, DDR
Apply Board Preset: ✓
Cross Trigger In: Disable
Cross Trigger Out: Disable



Vivado Block Design - BRAM



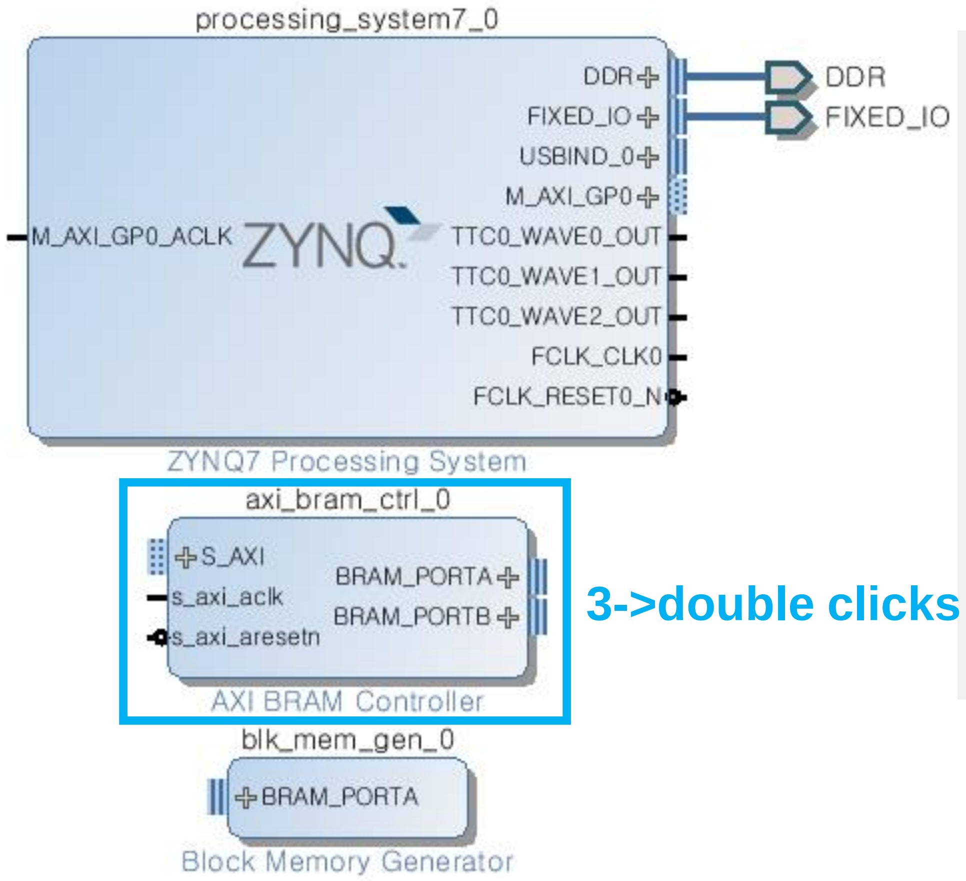
Search: bram (3 matches)

1 AXI BRAM Controller

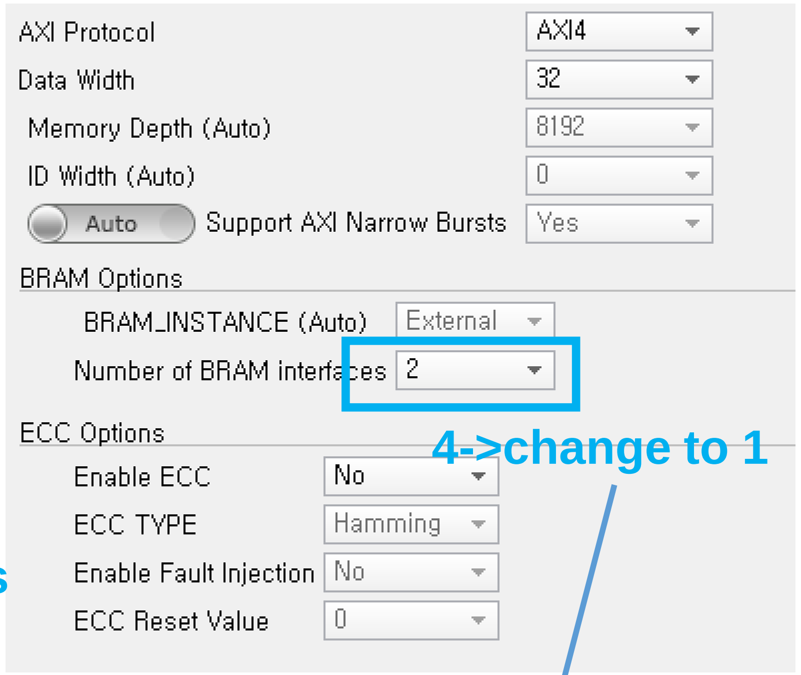
2 Block Memory Generator

LMB BRAM Controller

ENTER to select, ESC to cancel, Ctrl+Q for IP details



3->double clicks



AXI Protocol	AXI4
Data Width	32
Memory Depth (Auto)	8192
ID Width (Auto)	0
Auto Support AXI Narrow Bursts	Yes

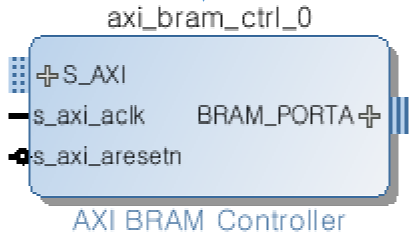
BRAM Options

BRAM_INSTANCE (Auto)	External
Number of BRAM interfaces	2

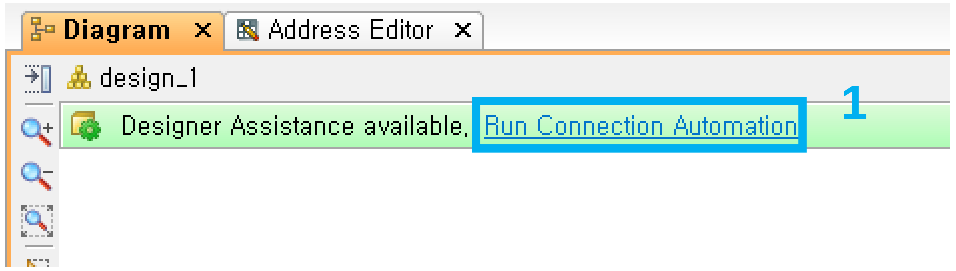
ECC Options

Enable ECC	No
ECC TYPE	Hamming
Enable Fault Injection	No
ECC Reset Value	0

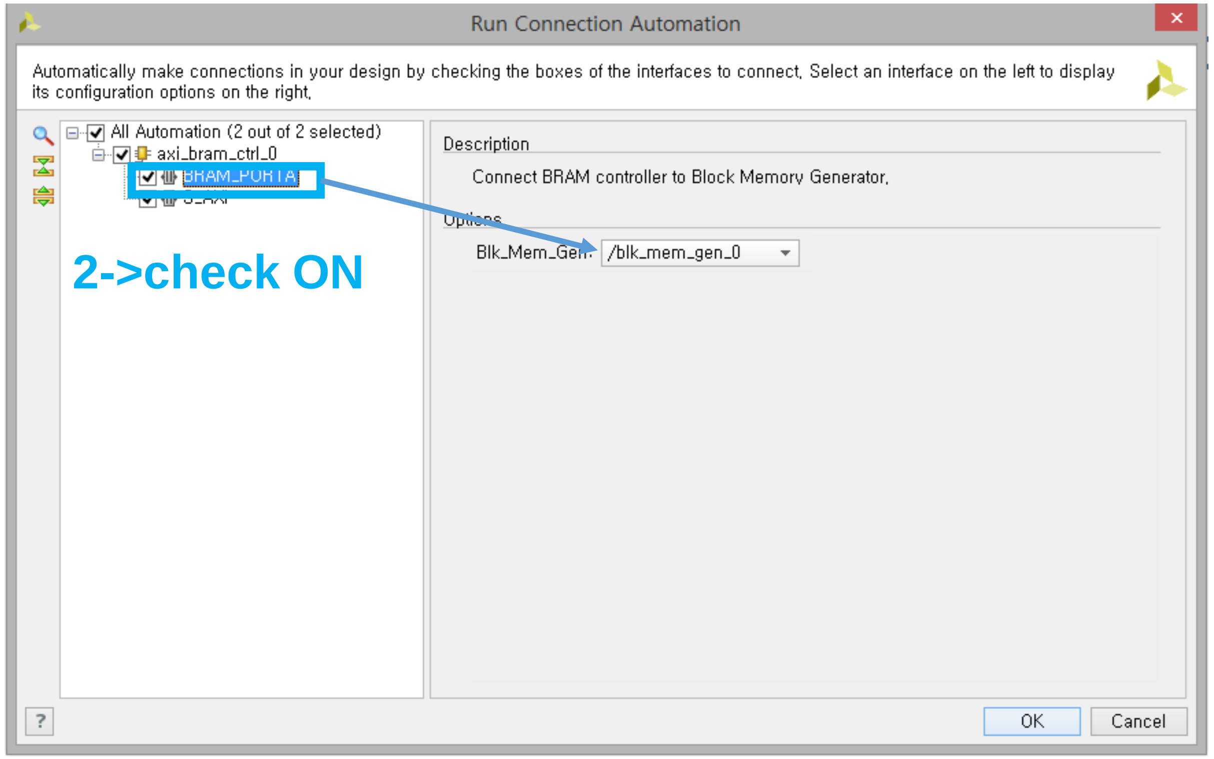
4->change to 1



Vivado Block Design - Connectivity(1)



Designer Assistance available. Run Connection Automation



Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

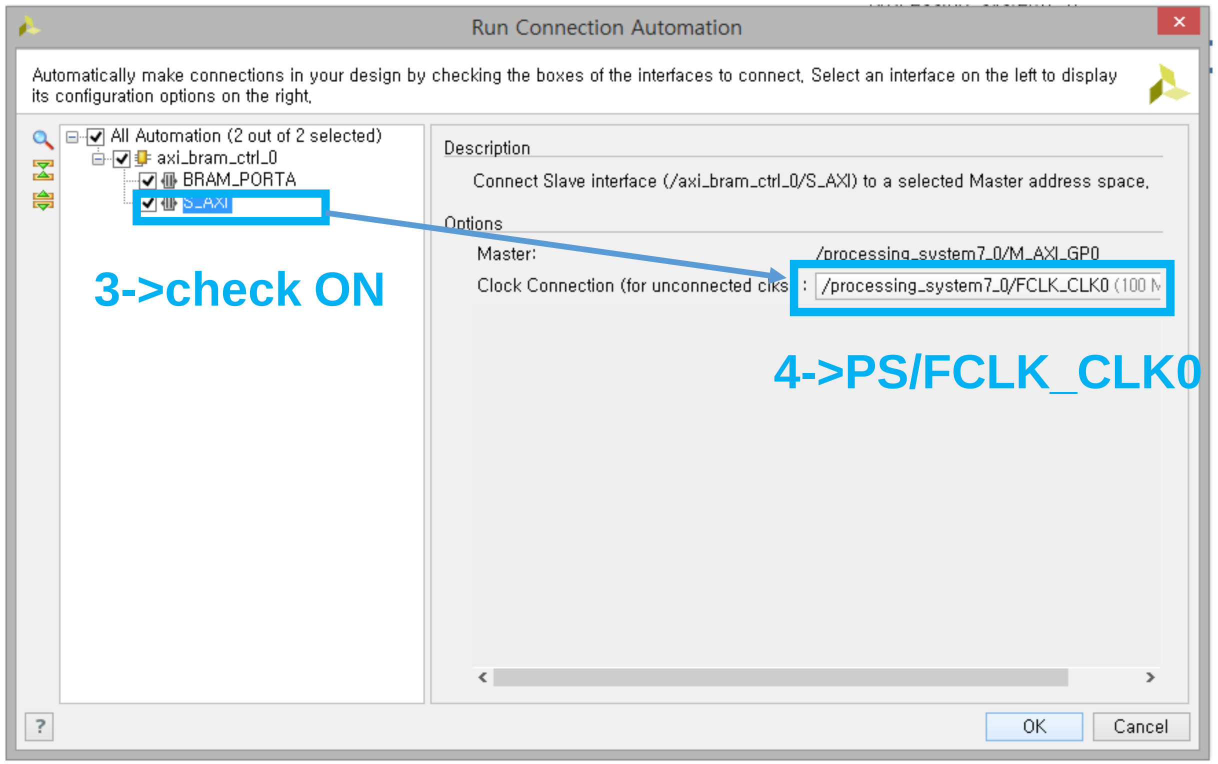
- All Automation (2 out of 2 selected)
  - axi_bram_ctrl_0
    - BRAM_PORTA
    - S_AXI

Description

Connect BRAM controller to Block Memory Generator.

Options

Blk_Mem_Gen: /blk_mem_gen_0



Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- All Automation (2 out of 2 selected)
  - axi_bram_ctrl_0
    - BRAM_PORTA
    - S_AXI

Description

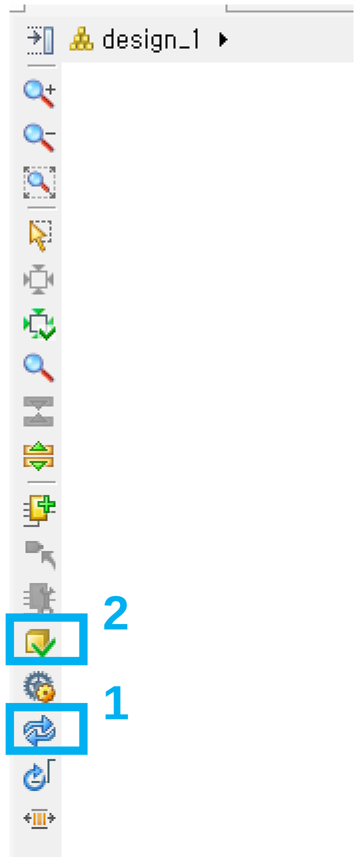
Connect Slave interface (/axi_bram_ctrl_0/S_AXI) to a selected Master address space.

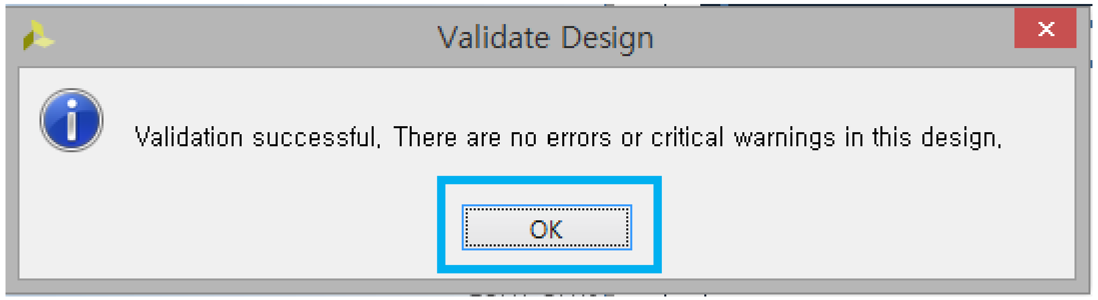
Options

Master: /processing_system7_0/M_AXI_GP0

Clock Connection (for unconnected clks) : /processing_system7_0/FCLK_CLK0 (100 M

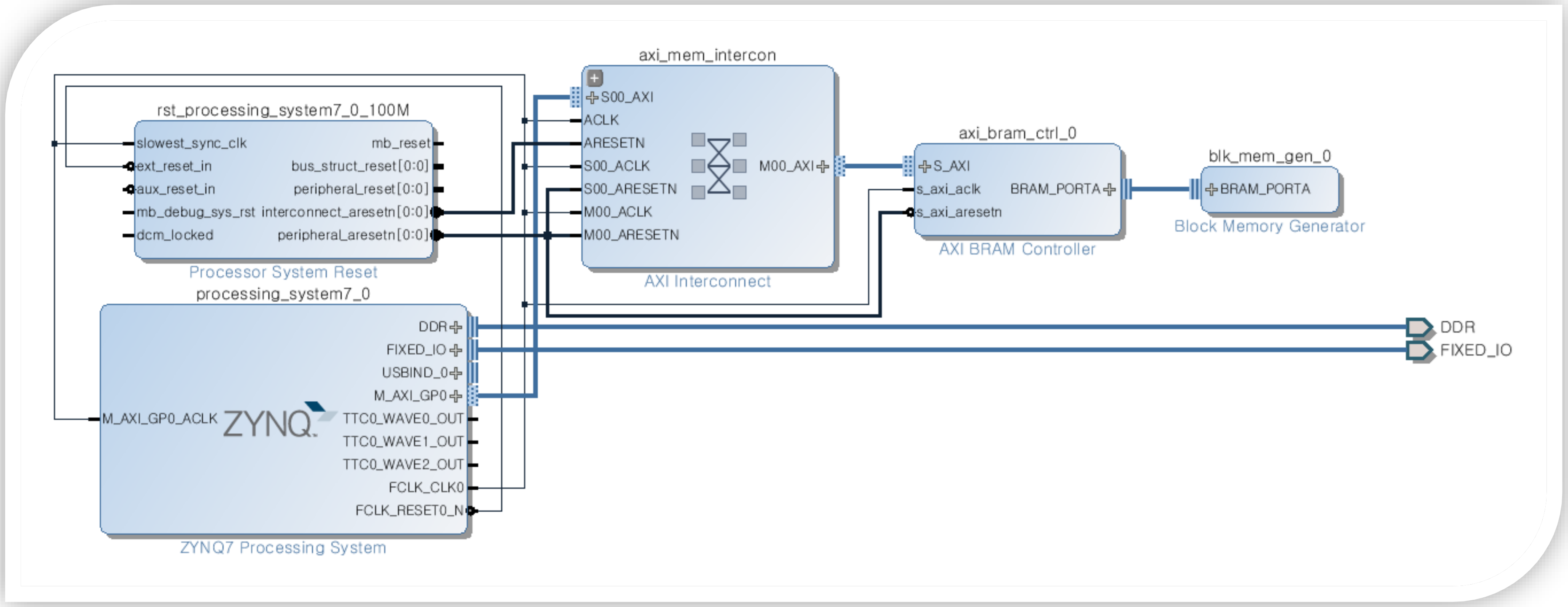
Vivado Block Design - Connectivity(2)



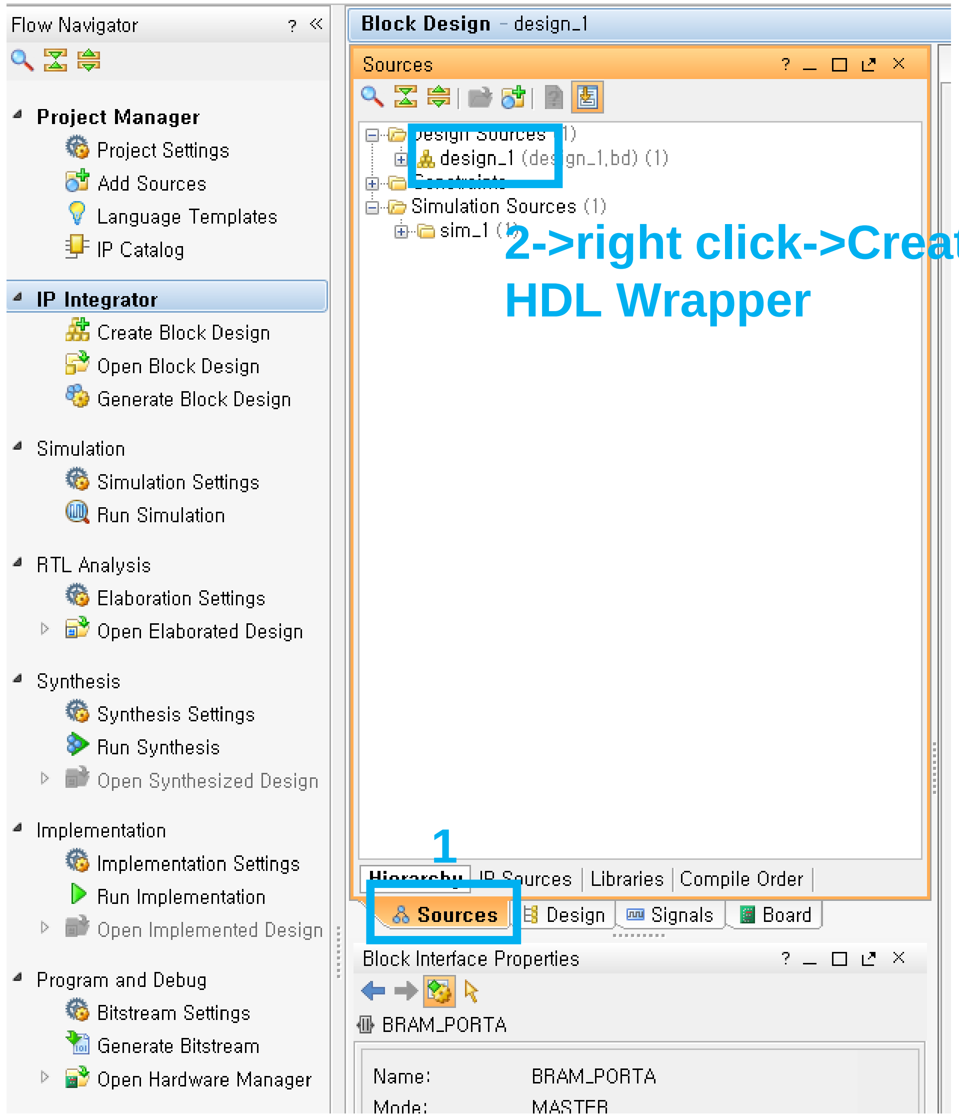


Validation successful. There are no errors or critical warnings in this design.

3->check no error sign



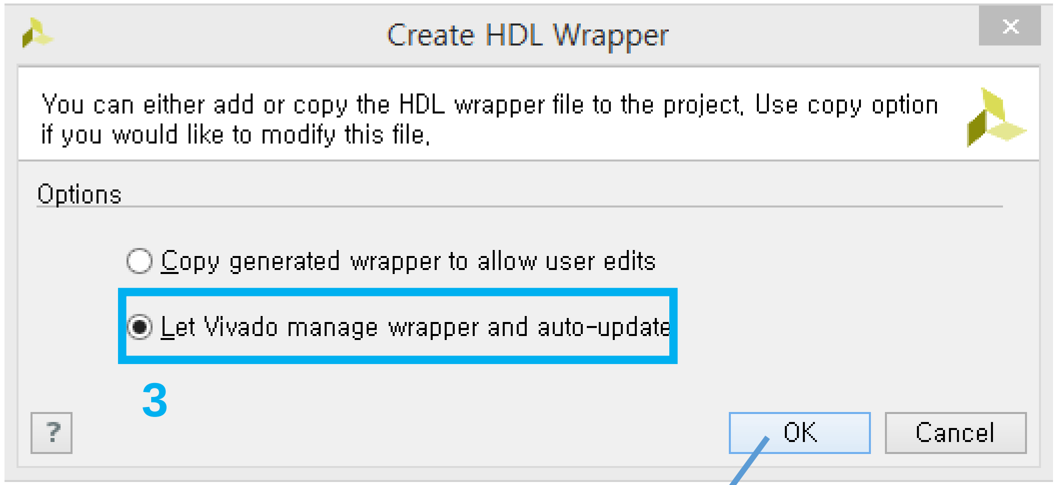
Vivado Block Design - Generate Bitstream



Name: BRAM_PORTA
Mode: MASTER

1

2->right click->Create HDL Wrapper



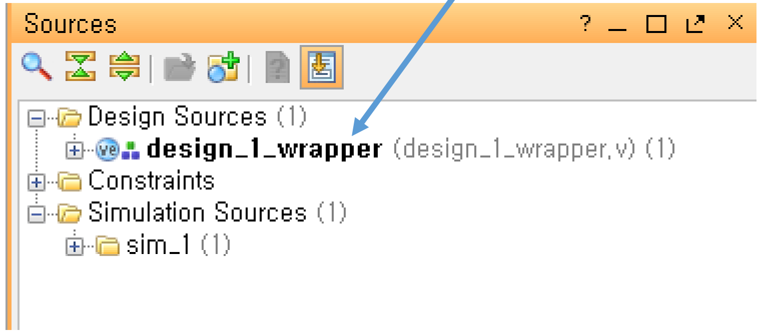
You can either add or copy the HDL wrapper file to the project. Use copy option if you would like to modify this file.

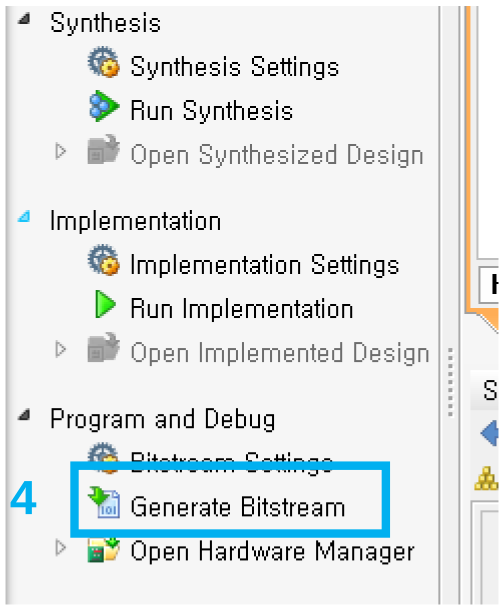
Options

○ Copy generated wrapper to allow user edits

● Let Vivado manage wrapper and auto-update

3

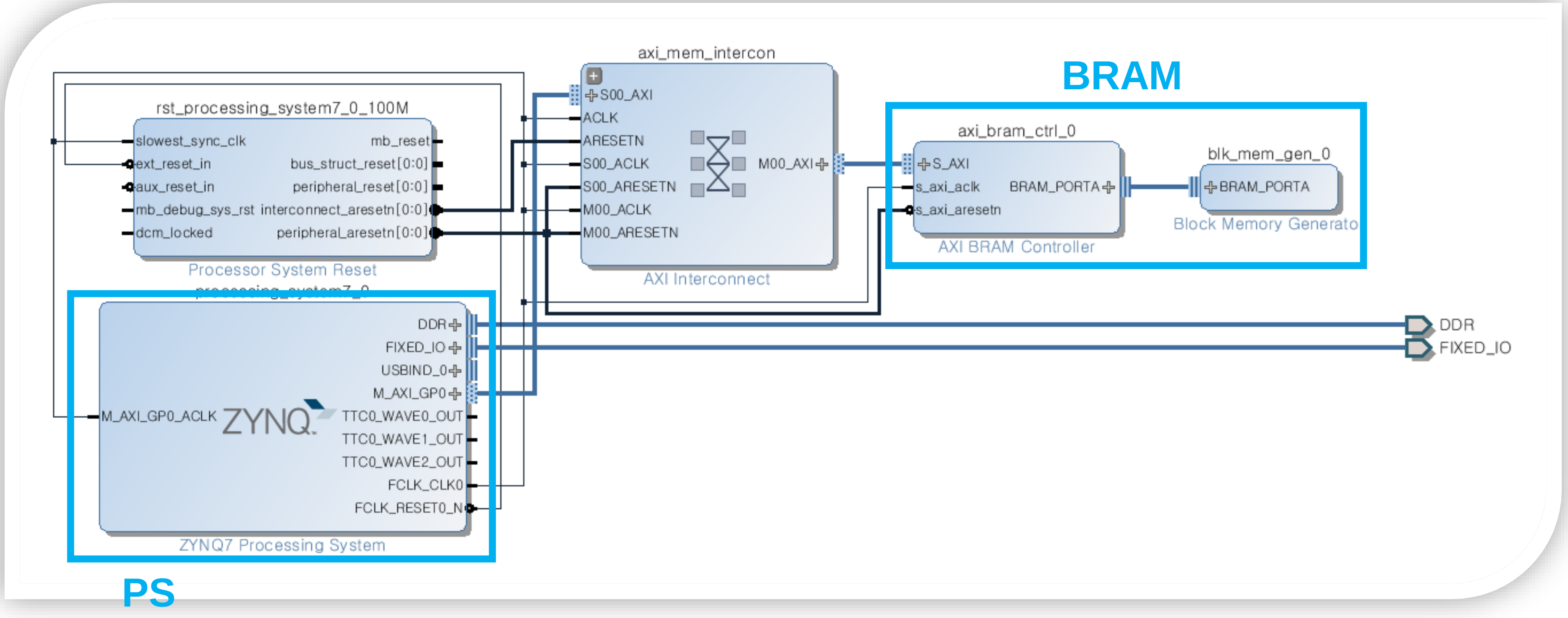




4

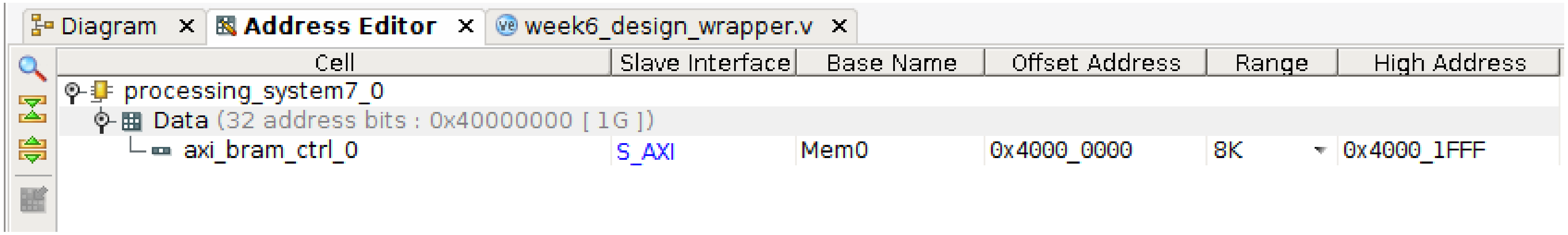
Vivado Block Design - Summary

- Create a block design with
  - PS: Processing System (part of Zynq-PS, ARM Cortex A9)
  - BRAM: Block Random Access Memory (part of Zynq-PL)
  - Connectivity (AXI interconnect, part of Zynq-PL)

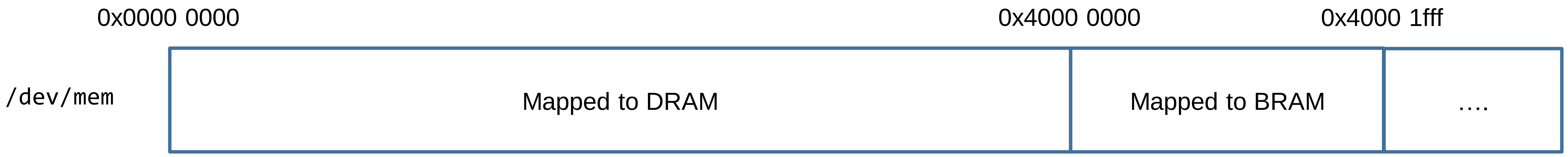


Memory map of ZYNQ SoC(1)

- DRAM is mapped to address 0x0000_0000 ~ 0x3FFF_FFFF.
- BRAM is mapped to address 0x4000_0000 ~ 0x4000_1FFF.



Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF



Memory map of ZYNQ SoC(2)

- System call **mmap()** can be used to access BRAM
  - int fd = open(“/dev/mem”, O_RDWR);
  - float *ptr = mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x40000000);
  - Refer to linux manual page for more details of open() and mmap() function.
  - `Host $> man 2 open      //mmap() system call manual page`
  - `Host $> man mmap        //open() system call manual page`

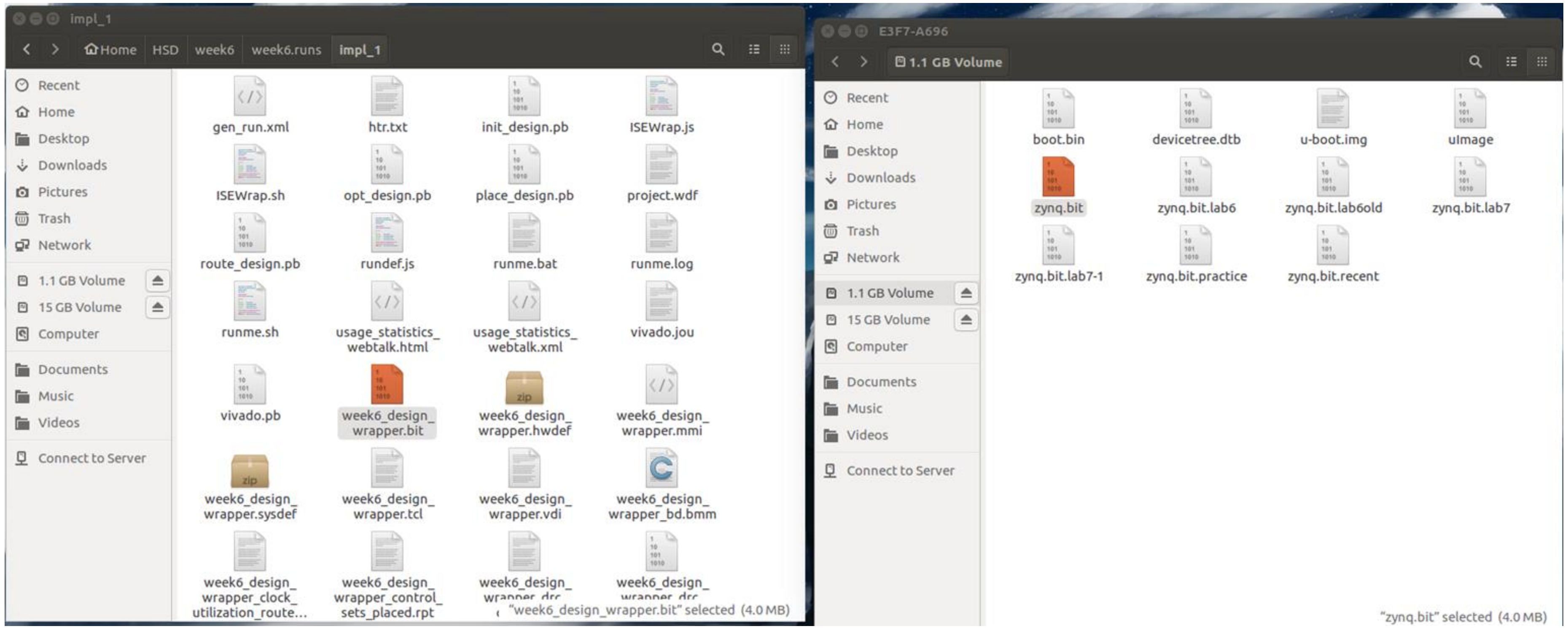
Tutorial: FPGA + Linux

Notations

- `HOST$ XXX`
  - Type XXX at the terminal of your Ubuntu-PC.
- `BOARD$ YYY`
  - Type YYY at the terminal of ZedBoard (on minicom).

Use your bitstream on Zedboard

- Copy the generated bitstream to your SD card.
  - Replace zynq.bit in the 1.1GB partition of SD card with your own bitstream.
  - Location of the generated bitstream:
    - `$(your_project)/$(your_project).runs/impl_1/design_1_wrapper.bit`



"week6_design_wrapper.bit" selected (4.0 MB)

"zynq.bit" selected (4.0 MB)

[ZEDBoard Hardware User's Guide, 2014]

Preparing the Board

1. Set configuration mode pins to **01100** (SD boot mode).
2. Insert USB-JTAG cable at J14.
3. Insert power cable at J20.
   then turn the power on with SW8.
   - LD13 lights up when it is on
4. Insert SD card.

※ CAVEAT: J14 pin is very fragile!

If you cannot pull out a USB-JTAG cable from J14, press hooks with something pointed (sharp pencil, etc.).

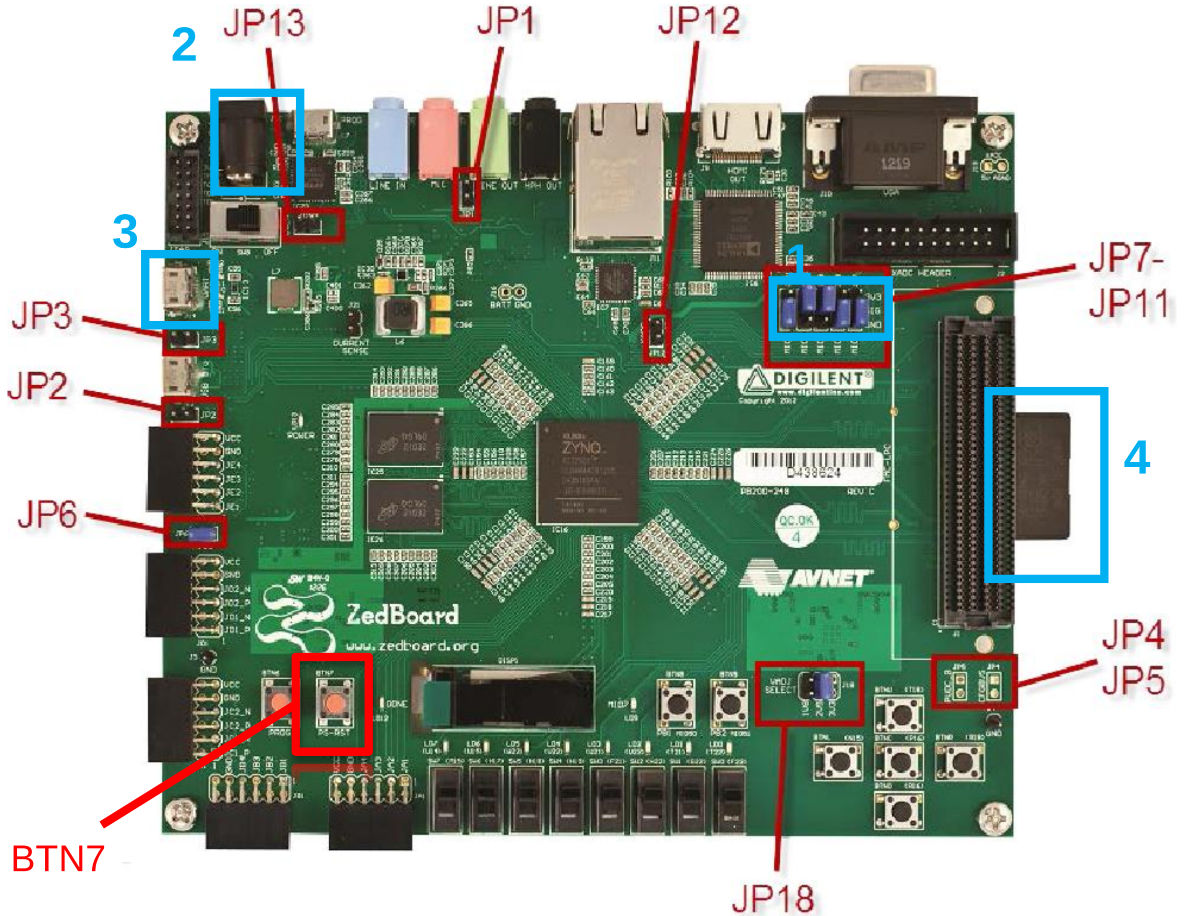
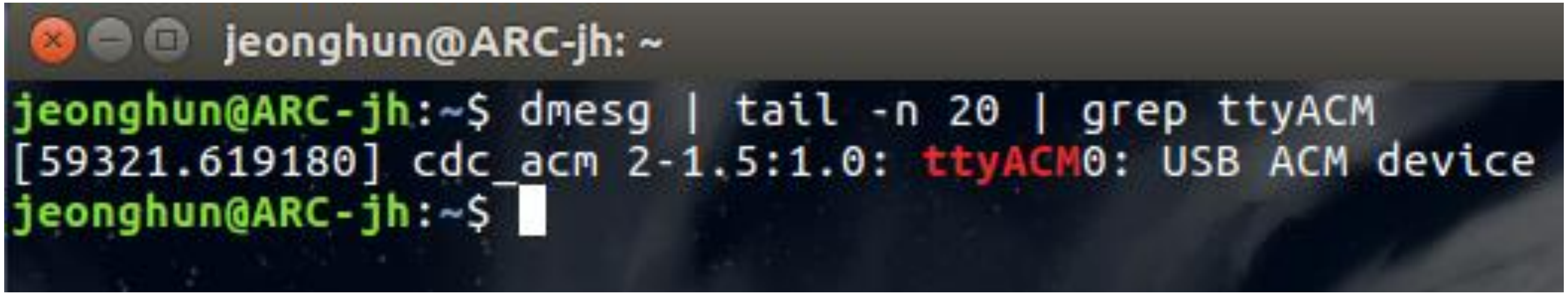


Figure 20 - ZedBoard Jumper Map

Connect your board to host(1)

- Setup from host terminal
  - `Host $> dmesg | tail -n 20 | grep ttyACM`

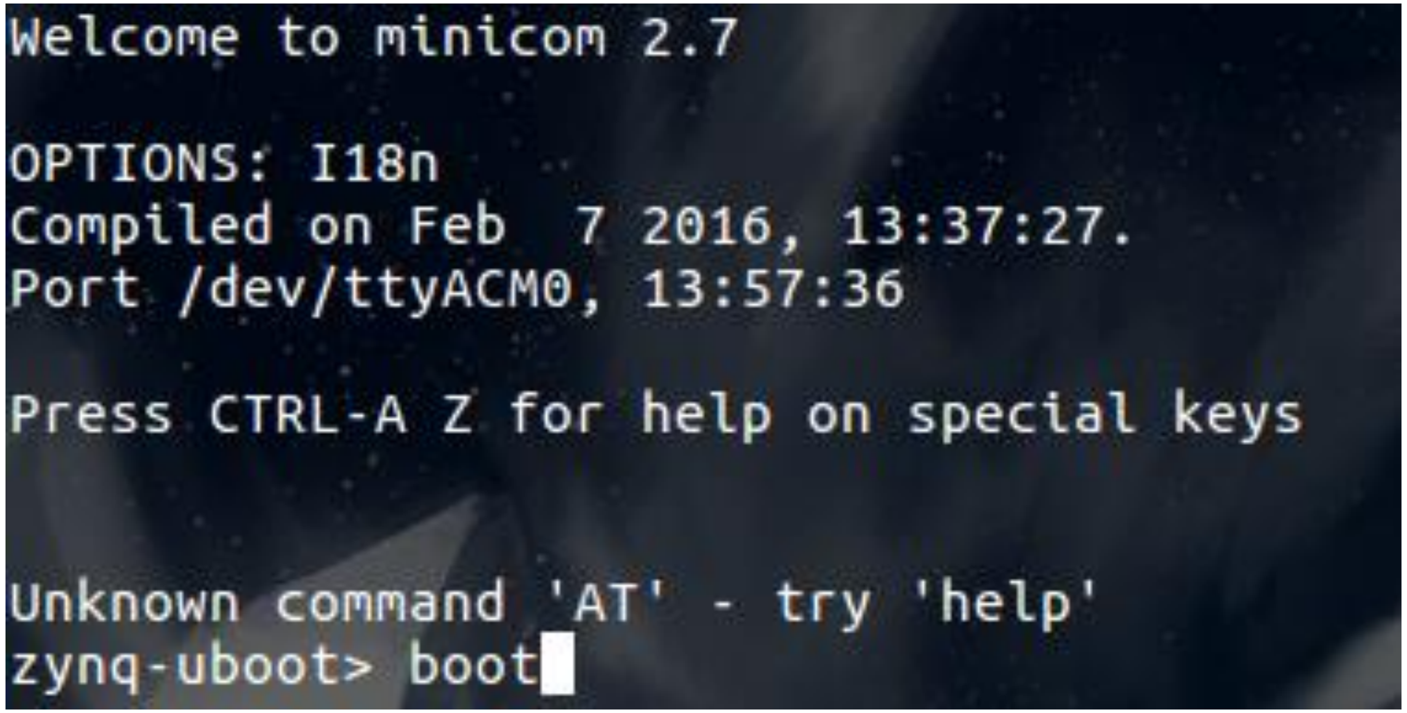


```
jeonghun@ARC-jh: ~
jeonghun@ARC-jh:~$ dmesg | tail -n 20 | grep ttyACM
[59321.619180] cdc_acm 2-1.5:1.0: ttyACM0: USB ACM device
jeonghun@ARC-jh:~$
```

- Connect to terminal of your board
  - `Host $> sudo minicom -D /dev/ttyACMx`
    - Replace x with the number from above

Connect your board to host(2)

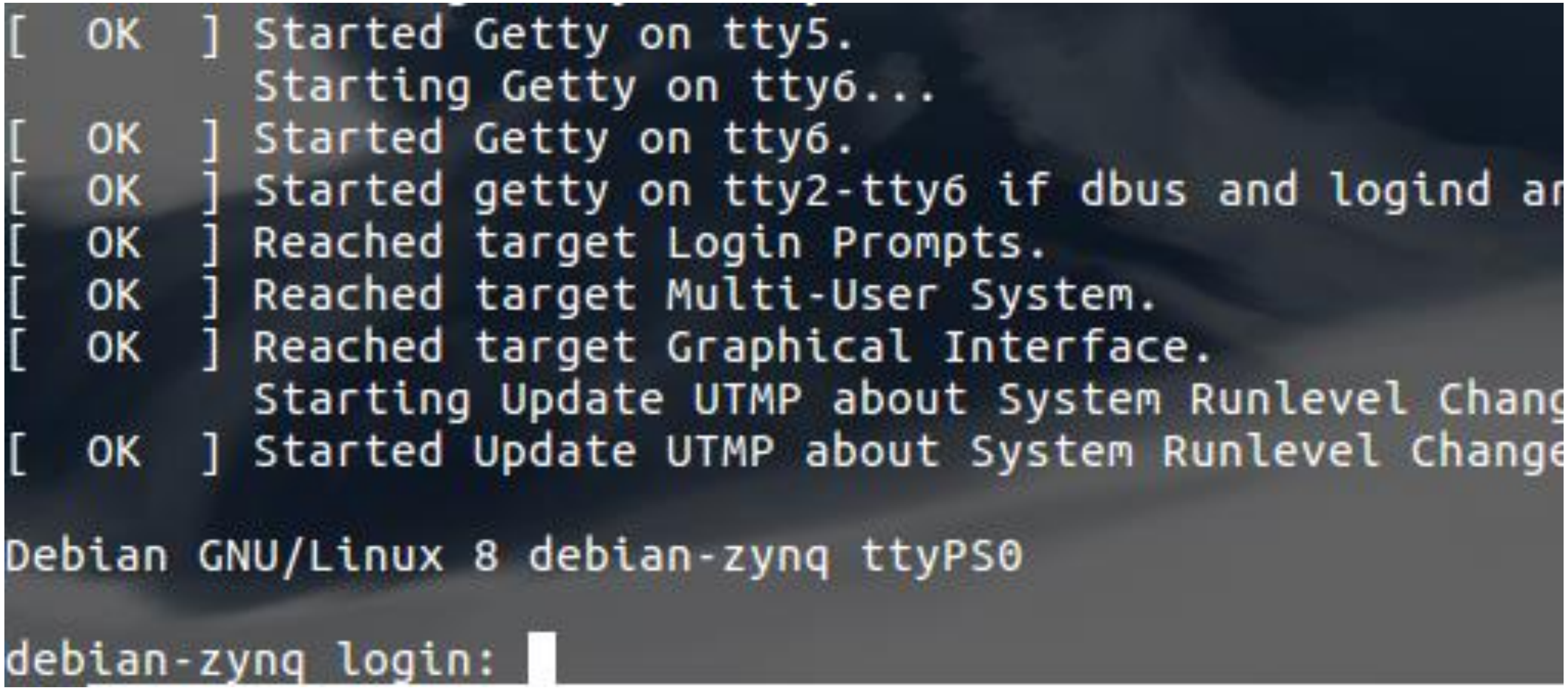
- Boot your board
  - `Board $> boot`
  - Wait 1~2 minutes for board to boot.
- If your Zedboard does not respond
  - Press BTN7(PS-RST).



```
Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Feb  7 2016, 13:37:27.
Port /dev/ttyACM0, 13:57:36

Press CTRL-A Z for help on special keys


Unknown command 'AT' - try 'help'
zynq-uboot> boot
```

Connect your board to host(3)

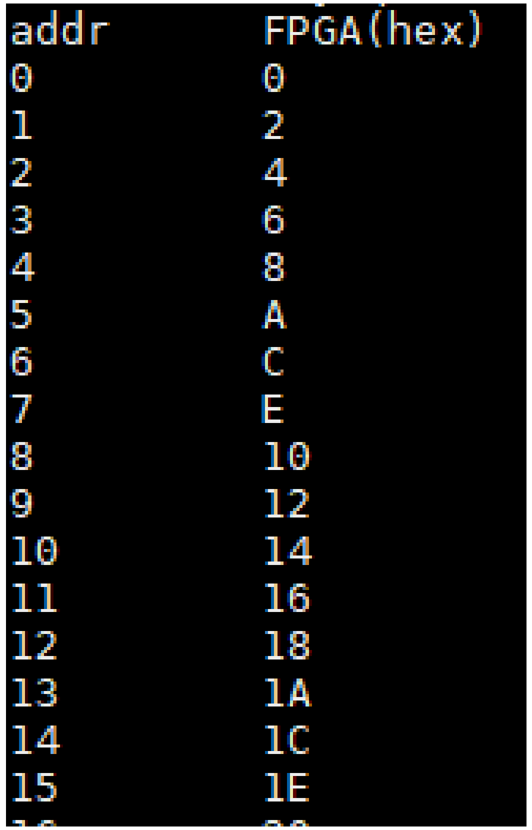
- Login to Debian Linux
  - ID : zed / PW : zedzed



```
[  OK  ] Started Getty on tty5.
         Starting Getty on tty6...
[  OK  ] Started Getty on tty6.
[  OK  ] Started getty on tty2-tty6 if dbus and logind ar
[  OK  ] Reached target Login Prompts.
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Chang
[  OK  ] Started Update UTMP about System Runlevel Change

Debian GNU/Linux 8 debian-zynq ttyPS0

debian-zynq login: █
```

Run example program

- Example program is in github
  - `Host$ git clone git://github.com/K16DIABLO/HSD_LAB6`
  - Copy HSD_LAB6 to Debian root file system in SD card
  - `BOARD$ cd HSD_LAB6/lab6`
  - `BOARD$ sudo make`
    - `When asked for password, input “zedzed”`
- Before turn off the board
  - `BOARD$ sudo poweroff`
- To quit minicom
  - ctrl + a, q



```
addr        FPGA(hex)
0           0
1           2
2           4
3           6
4           8
5           A
6           C
7           E
8           10
9           12
10          14
11          16
12          18
13          1A
14          1C
15          1E
```

Source Code – main.c

```
  int foo = open("/dev/mem", O_RDWR);
  // Given a pathname for a file, open() returns a file descriptor
  // 'dev/mem' refers to the system's physical memory
  // O_RDWR means both readable and writable access mode
  int *fpga_bram = mmap(NULL, SIZE * sizeof(int), PROT_READ|PROT_WRITE, MAP_SHARED, foo, 0x40000000);
  // mmap() creates a new mapping in the virtual address space of the calling process
  // NULL means that the kernel chooses the address for mapping
  // SIZE specifies the length of the mapping
  // PROT_ arguments describe the memory protection (RD/WR)
  // MAP_SHARED makes updates visible to other processes
  // foo indicates the file descriptor to be mapped
  // 0x4000_0000 refers to offset of the file descriptor
```

```
for (i = 0; i < SIZE; i++)
    *(fpga_bram + i) = (i * 2);
 // write arbitrary data on the BRAM area
 printf("%-10s%-10s\n", "addr", "FPGA(hex)");
 for (i = 0; i < SIZE; i++)
    printf("%-10d%-10X\n", i, *(fpga_bram + i));
 // read and show the data to check if BRAM's working correctly
```

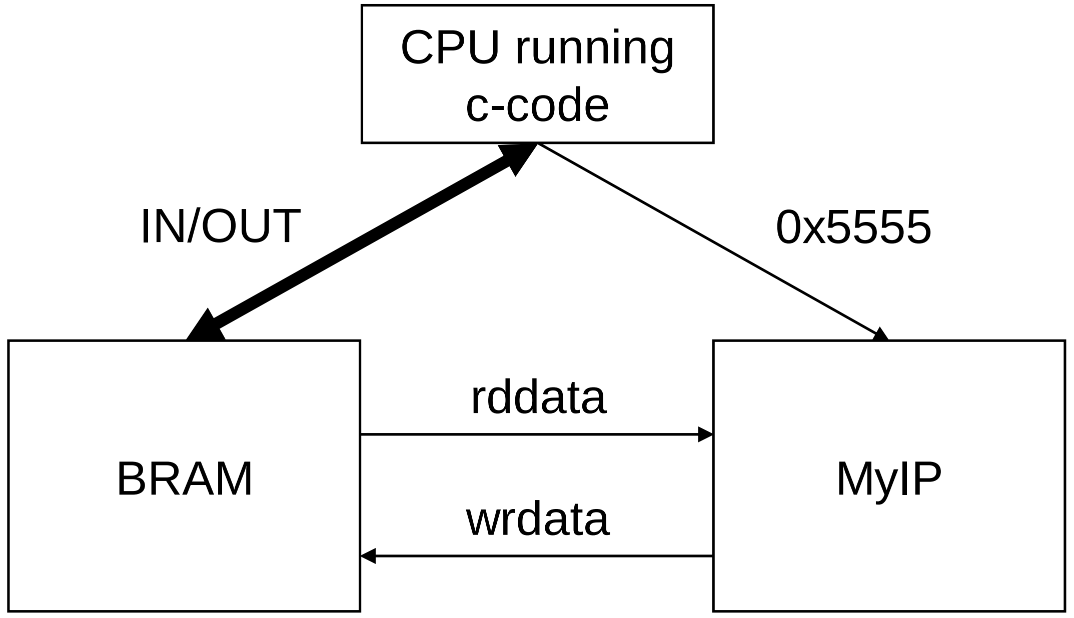
Source Code – Makefile

```
all: main.c
	gcc main.c && sudo ./a.out

// target : prerequisites
// [TAB]recipe
//
// ex)
// make $target : run recipes of $target using prerequisites
// make : (missing $target arguments) run the first target
```

Practice: Explain functionality of given IP

Practice

- Run a sample project provided by TA and explain the functionality of MyIP briefly.
  - Directory: HSD_LAB6/lab6_practice
  - Sample HW: zynq.bit from eTL
  - Sample SW: main.c Makefile



Grading policy

- Check lists
  - Correct result (40 points)
  - Correct explanation (50 points)
  - Finished in class (10 points)

- Show that your board works
  - In class / office hour (Tue) recommended
  - If you cannot come to the office hour, mail us with the result of the practice and brief explanation.

Appendix – Linux Booting on Zynq

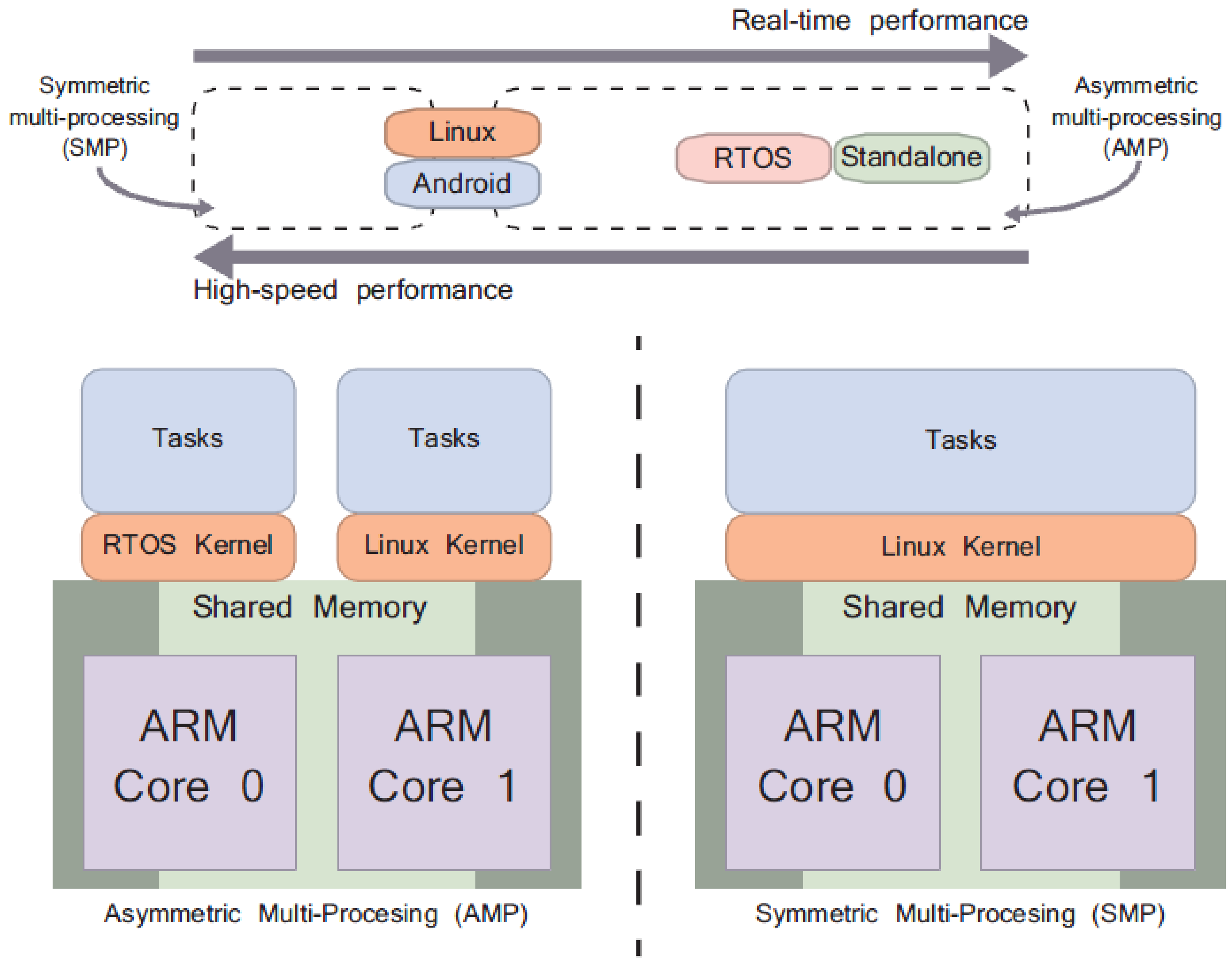
Introduction to OS on Zynq

- Why Use an Embedded Operating System?
  - Reducing Time to Market
  - Make Use of Existing Features
  - Reduce Maintenance and Development Costs
- Choosing the Right Type of Operating System
  - Standalone Operating Systems
  - Real-Time Operating Systems (RTOS)
  - Others: Linux, Android, …

[The Zynq Book, 2014]

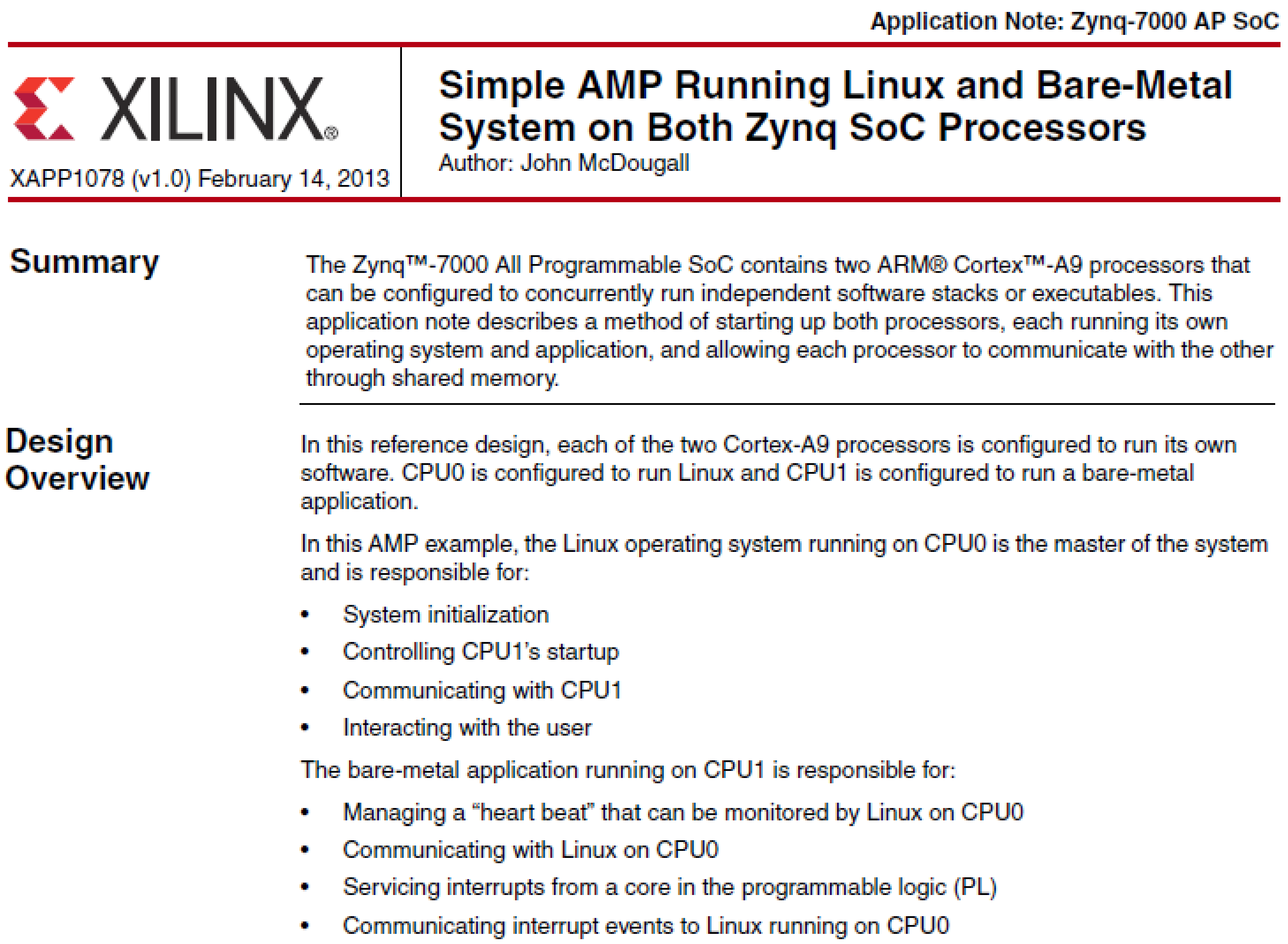
Multi-Processor Systems

- Asymmetric/Symmetric Multi-Processing (AMP/SMP)



[XAPP1078 Simple AMP, 2013]

Note) Zynq AMP Example



Application Note: Zynq-7000 AP SoC

XILINX®

XAPP1078 (v1.0) February 14, 2013

Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors

Author: John McDougall

Summary

The Zynq™-7000 All Programmable SoC contains two ARM® Cortex™-A9 processors that can be configured to concurrently run independent software stacks or executables. This application note describes a method of starting up both processors, each running its own operating system and application, and allowing each processor to communicate with the other through shared memory.

Design Overview

In this reference design, each of the two Cortex-A9 processors is configured to run its own software. CPU0 is configured to run Linux and CPU1 is configured to run a bare-metal application.

In this AMP example, the Linux operating system running on CPU0 is the master of the system and is responsible for:

- System initialization
- Controlling CPU1's startup
- Communicating with CPU1
- Interacting with the user

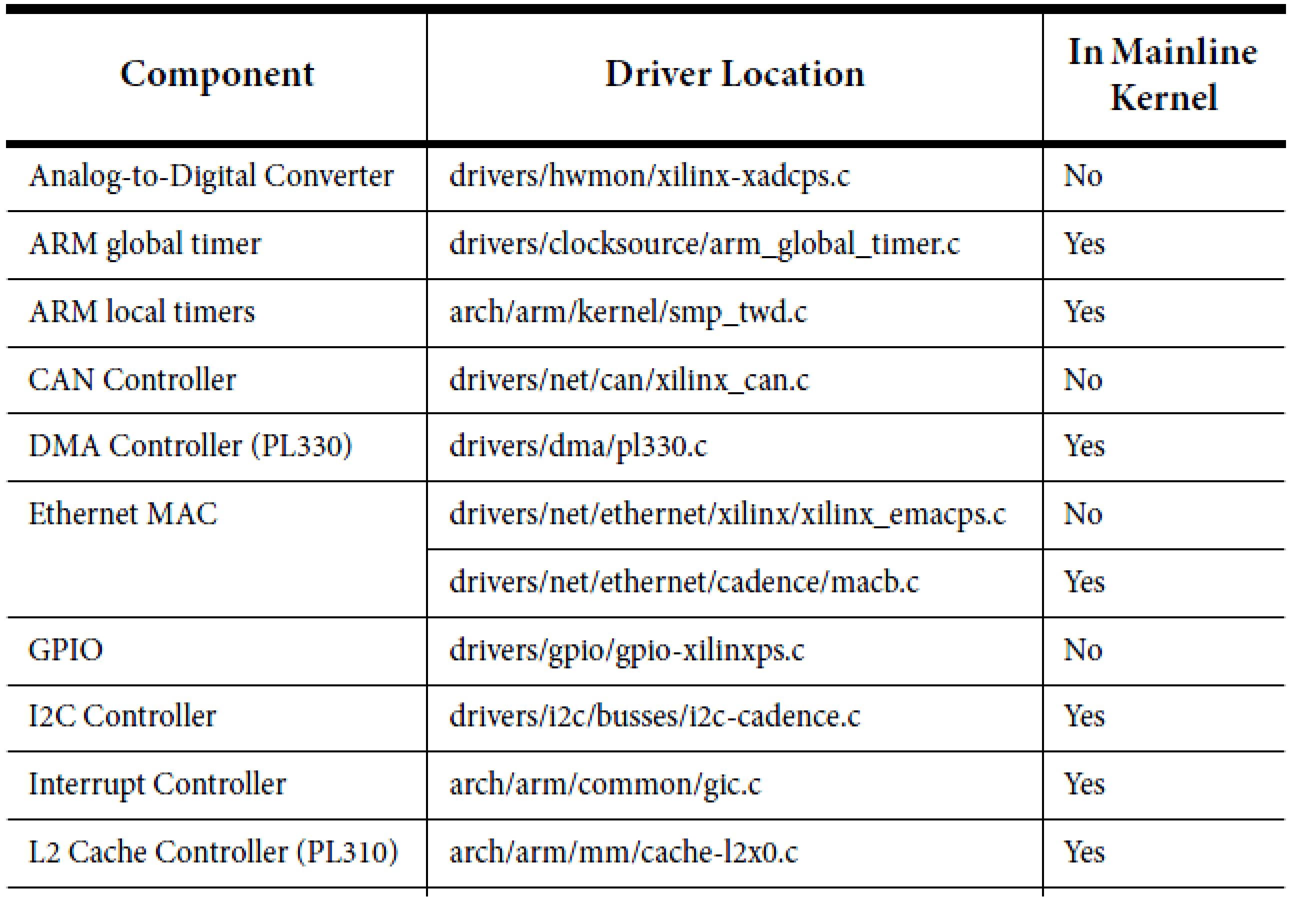
The bare-metal application running on CPU1 is responsible for:

- Managing a "heart beat" that can be monitored by Linux on CPU0
- Communicating with Linux on CPU0
- Servicing interrupts from a core in the programmable logic (PL)
- Communicating interrupt events to Linux running on CPU0

[The Zynq Book, 2014]

Xilinx Zynq-Linux

- Linux kernel + Xilinx BSP & device drivers

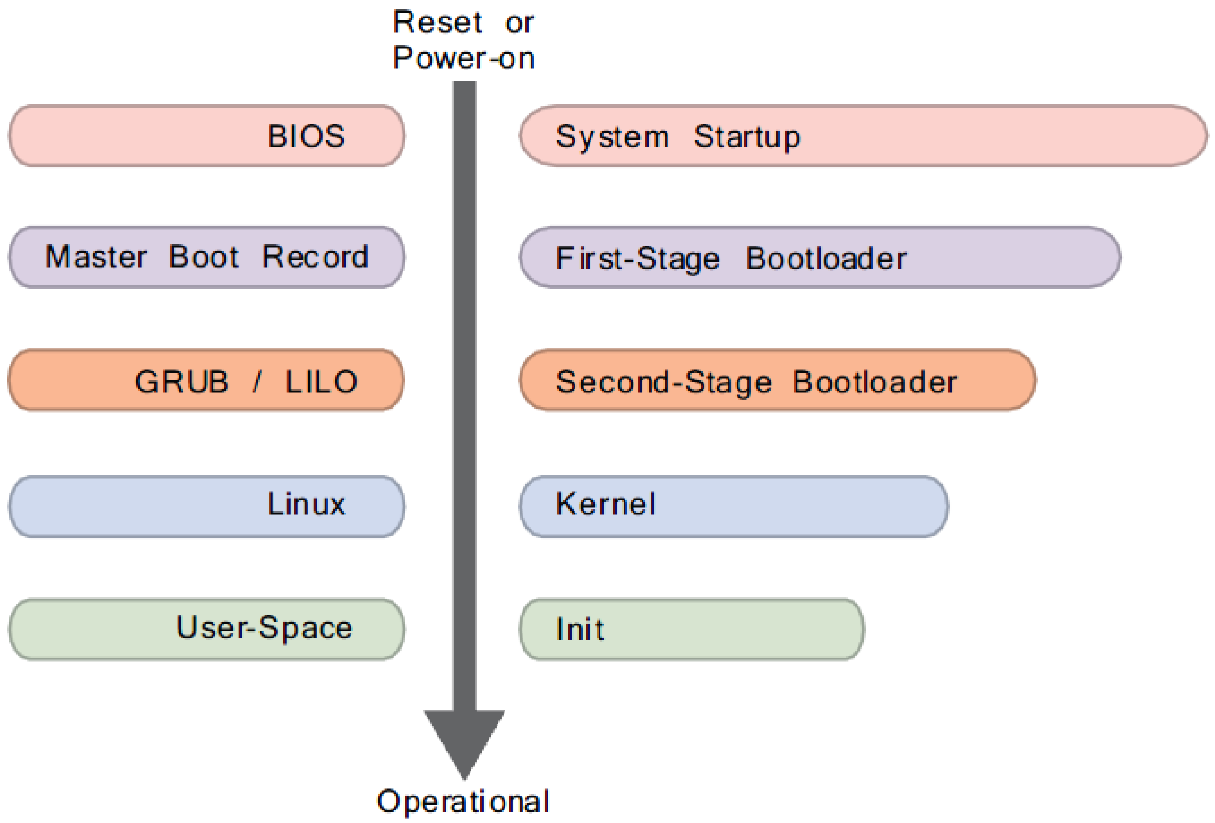


Component	Driver Location	In Mainline Kernel
Analog-to-Digital Converter	drivers/hwmon/xilinx-xadcps.c	No
ARM global timer	drivers/clocksource/arm_global_timer.c	Yes
ARM local timers	arch/arm/kernel/smp_twd.c	Yes
CAN Controller	drivers/net/can/xilinx_can.c	No
DMA Controller (PL330)	drivers/dma/pl330.c	Yes
Ethernet MAC	drivers/net/ethernet/xilinx/xilinx_emacps.c	No
	drivers/net/ethernet/cadence/macb.c	Yes
GPIO	drivers/gpio/gpio-xilinxps.c	No
I2C Controller	drivers/i2c/busses/i2c-cadence.c	Yes
Interrupt Controller	arch/arm/common/gic.c	Yes
L2 Cache Controller (PL310)	arch/arm/mm/cache-l2x0.c	Yes

[The Zynq Book, 2014]

Generic Linux Booting

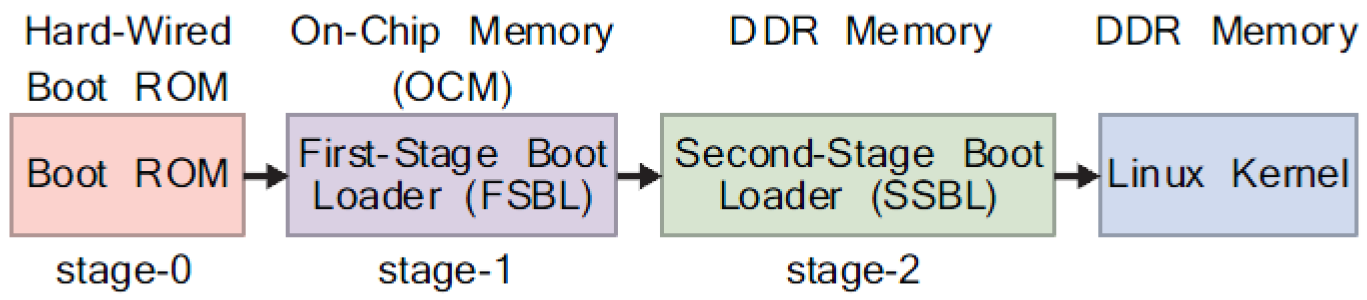
- Overview
  - Basic I/O System (BIOS)
  - First Stage Boot Loader (FSBL)
  - Second SBL (SSBL)
  - Kernel
  - Init

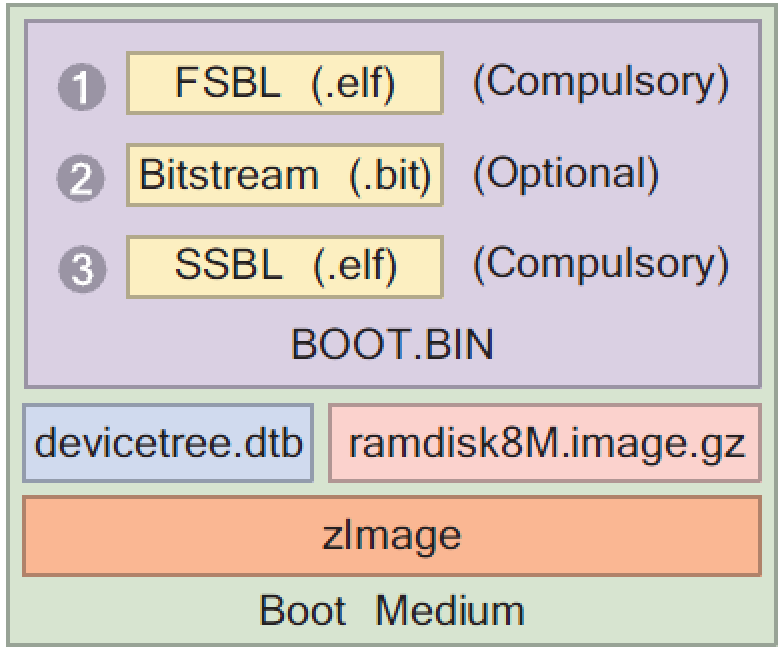


[The Zynq Book, 2014]

Zynq Linux Booting

- Example Boot Files
  - BOOT.BIN
  - zImage
  - devicetree.dtb
  - ramdisk8M.image.gz

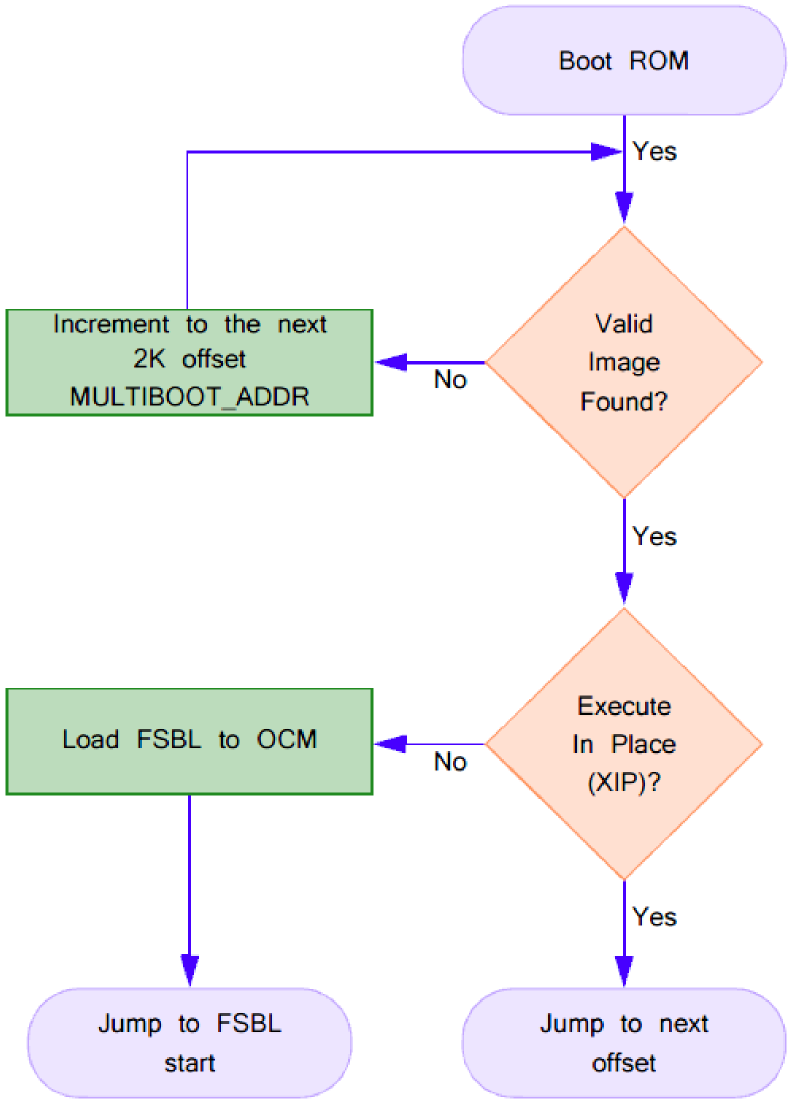


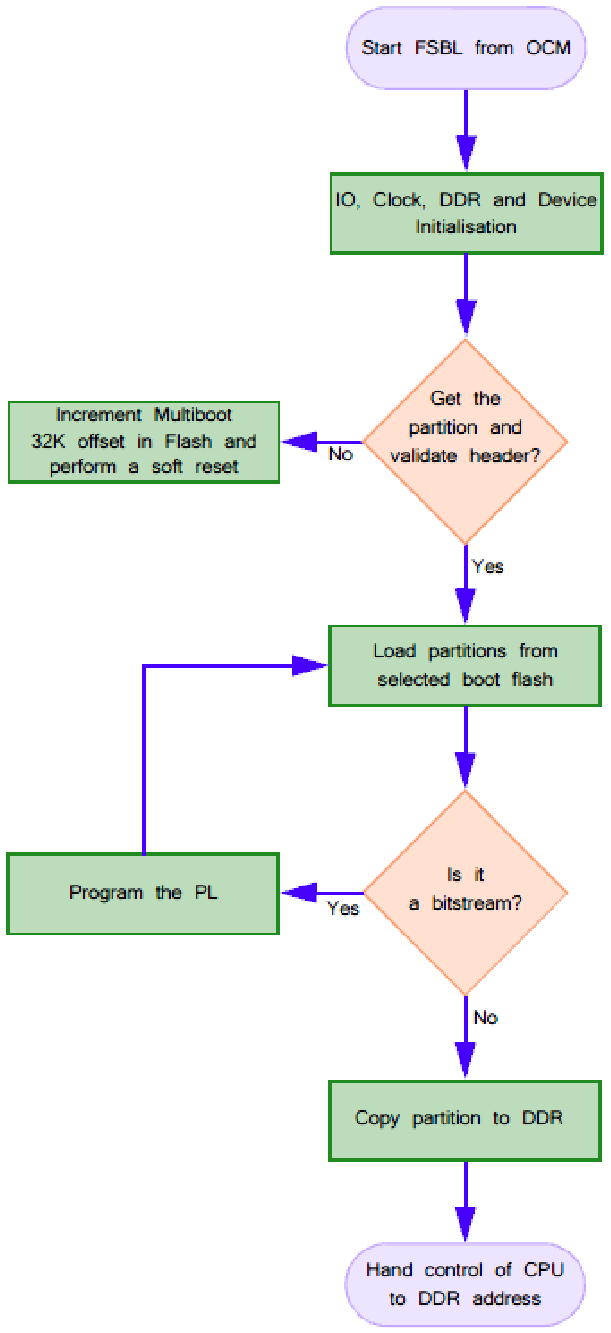

1 FSBL (.elf) (Compulsory)
2 Bitstream (.bit) (Optional)
3 SSBL (.elf) (Compulsory)
BOOT.BIN
devicetree.dtb
ramdisk8M.image.gz
zImage
Boot Medium

[The Zynq Book, 2014]

Zynq Linux Boot Process

- Boot ROM
- FSBL
- SSBL
- Linux




Start FSBL from OCM
IO, Clock, DDR and Device Initialisation
Get the partition and validate header?
No
Increment Multiboot 32K offset in Flash and perform a soft reset
Yes
Load partitions from selected boot flash
Is it a bitstream?
Yes
Program the PL
No
Copy partition to DDR
Hand control of CPU to DDR address

Note) Boot Files from the Lab

- Example Boot Files
  - 1st partition: FAT
    - BOOT.BIN = FSBL
    - u-boot.img = SSBL
    - zynq.bit = bistream
    - devicetree.dtb
    - uImage
  - 2nd partition: ext4
    - Debian root file system