High-Level FPGA Design

Lab 9: Data Logger Block Design

In this lab you will:

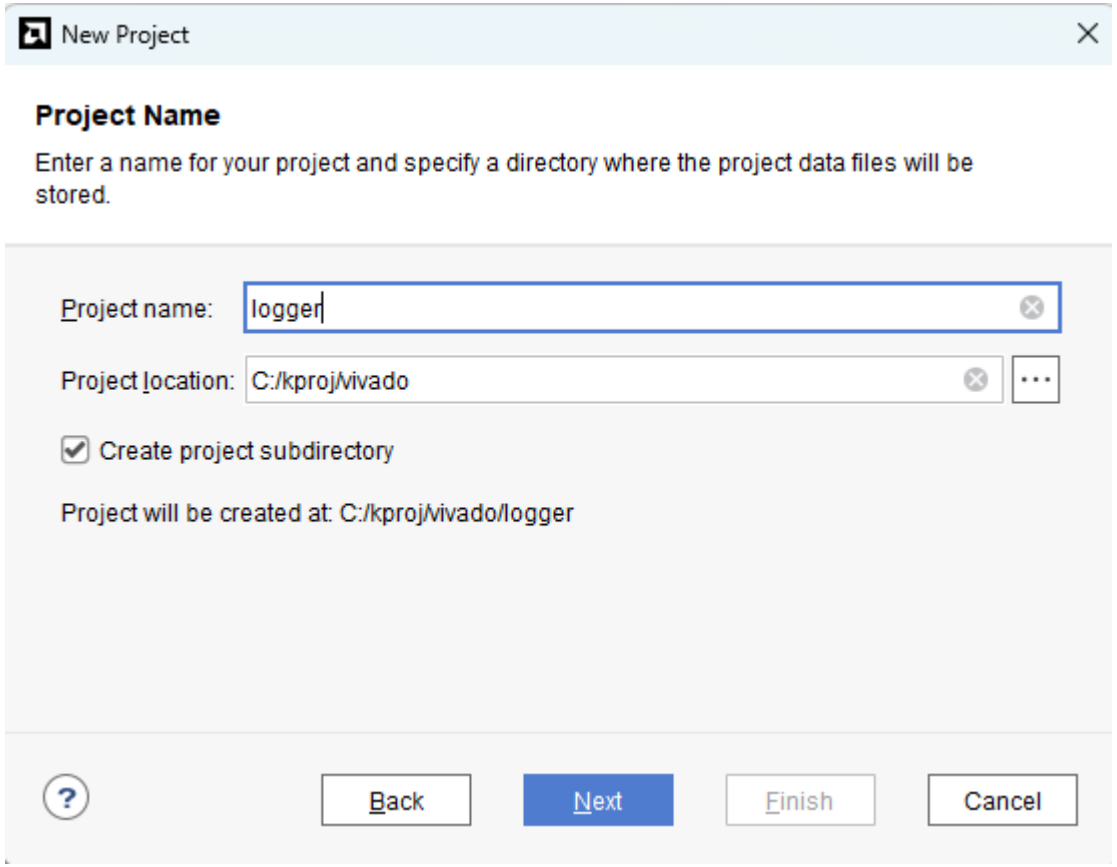
- prepare Vivado Block Design with logger IP for Red Pitaya
- send instructions to IP with monitor
- use custom SW on RedPitaya Linux to read sampled data

Sources for Vivado Project

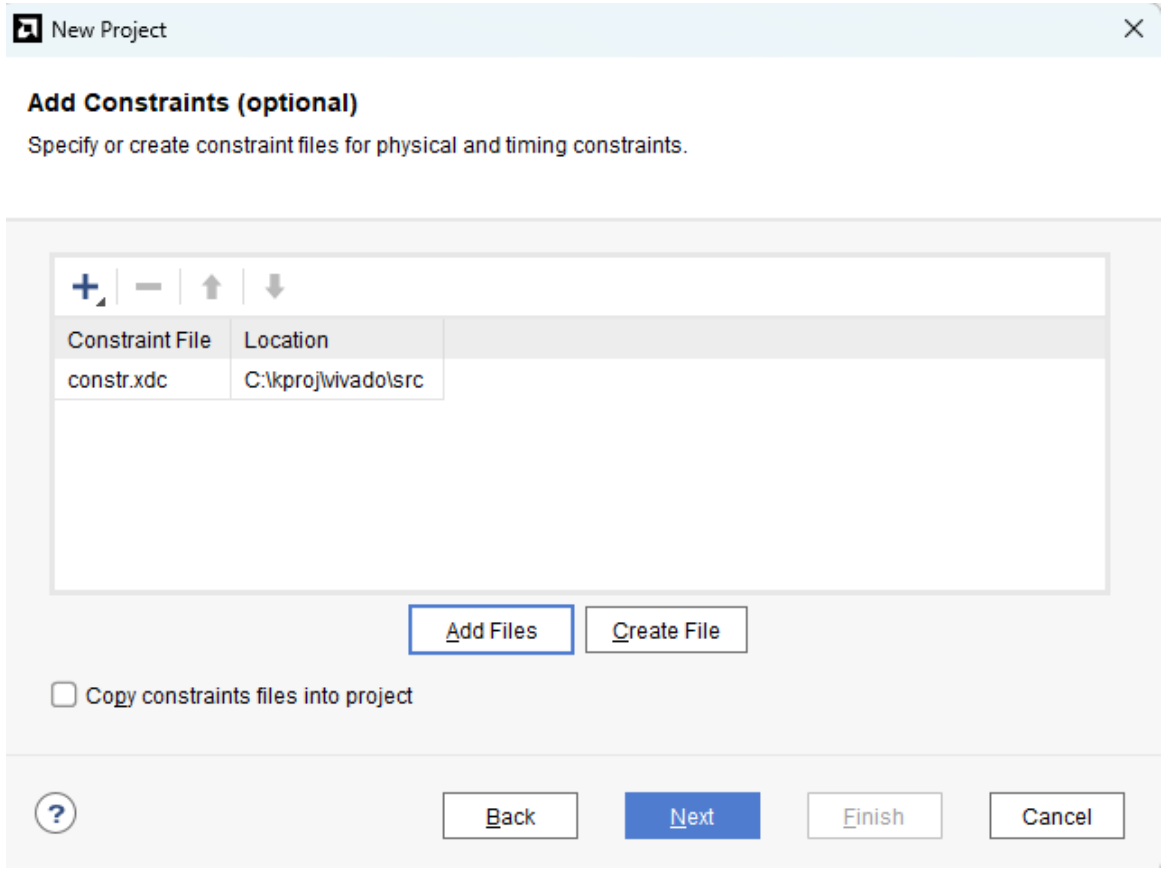
- Start new Vivado project, define project name (**logger**) and location
- Click Next, Select RTL project and Next
  - Vivado will create a new folder, now switch to Windows Explorer and copy sources and constraints
- Project folder structure

```
vivado/
 ├── ip_repo/   ← IP folder
 │    └── logger_1_2
 │          └── component.xml
 │              ...
 ├── format.tcl ← bitstream format script
 ├── src/
 │     └── constr.xdc  ← Vivado constranits
 ├── logger    ← your Vivado project folder
```

9-1 New Vivado Project

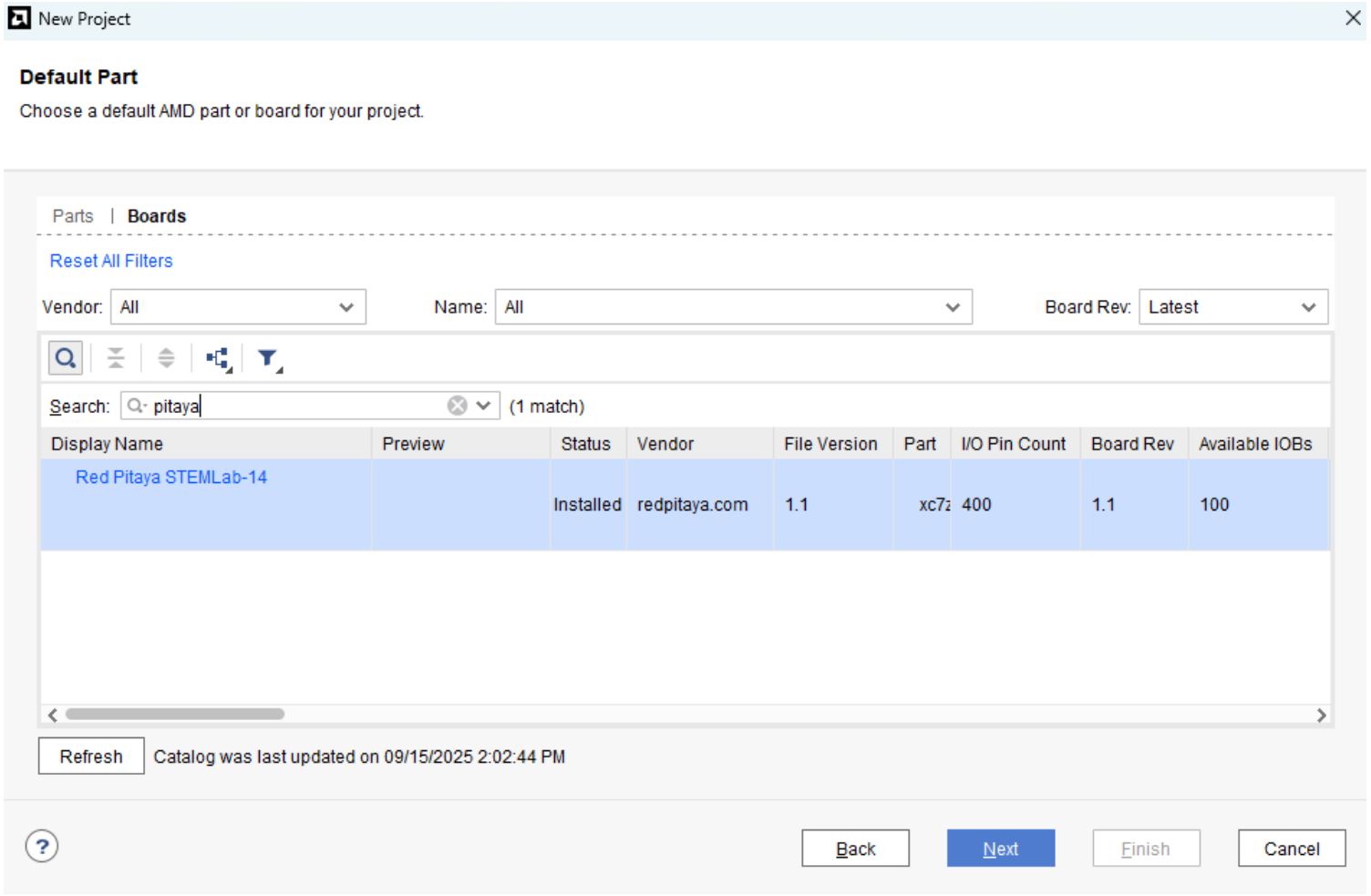


- Add Constraints



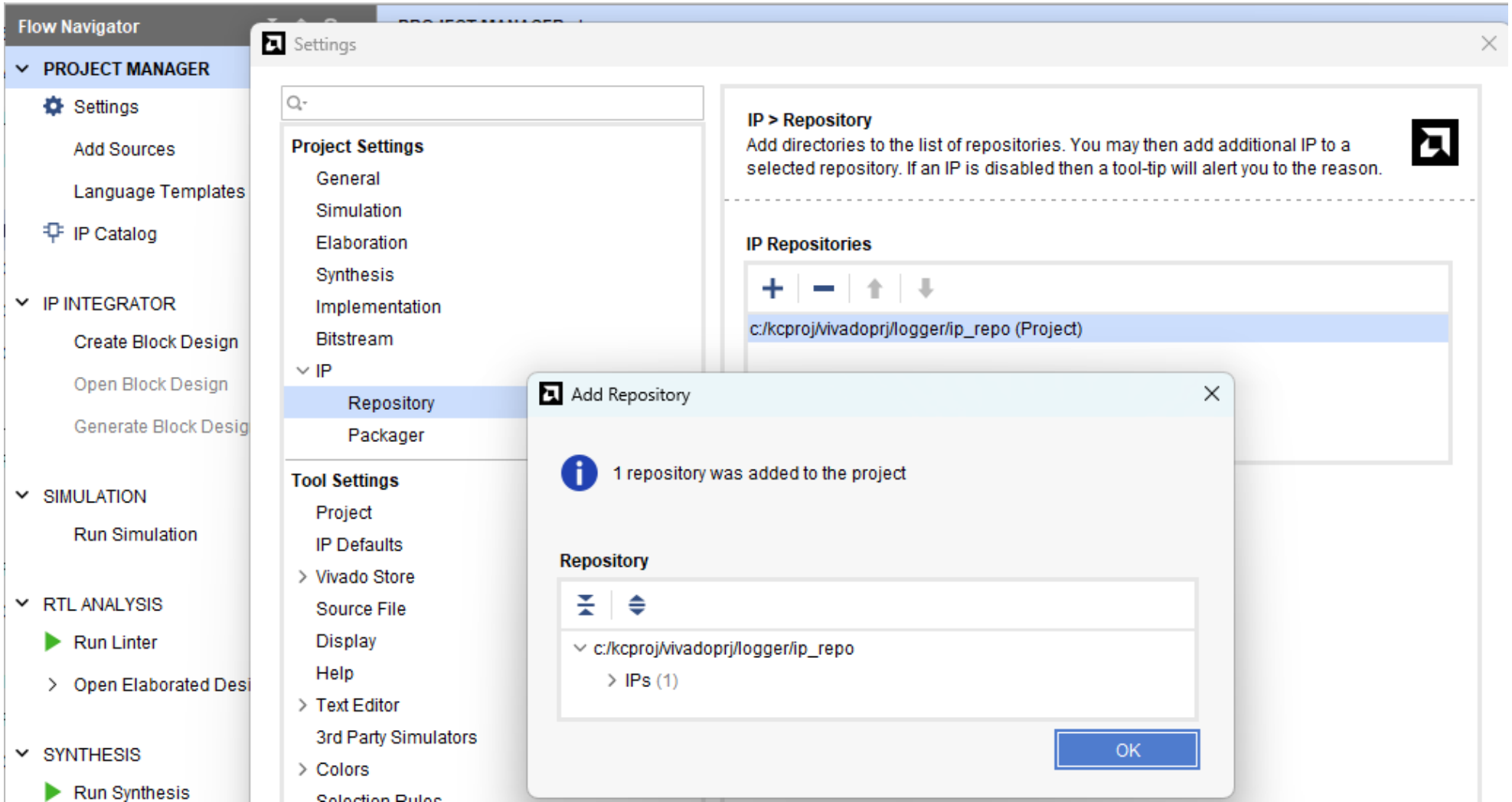
9-1 New Vivado Project

- Select Board: Red Pitaya



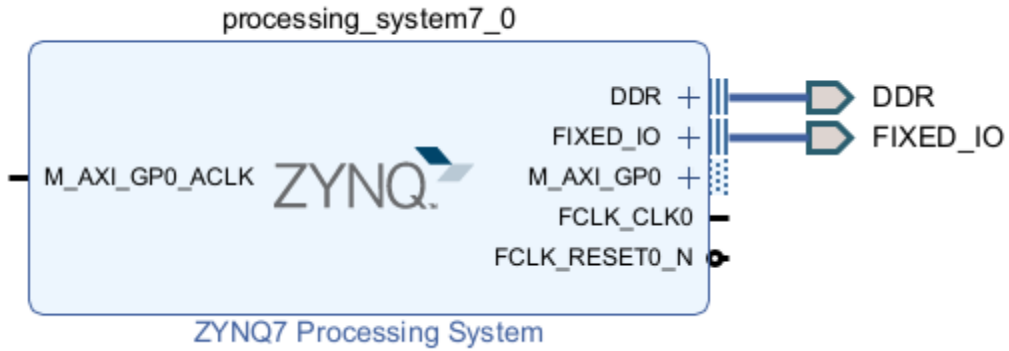
9-1 New Vivado Project

- Project Settings, IP, Repository, + Add Repository, Browse for ip_repo



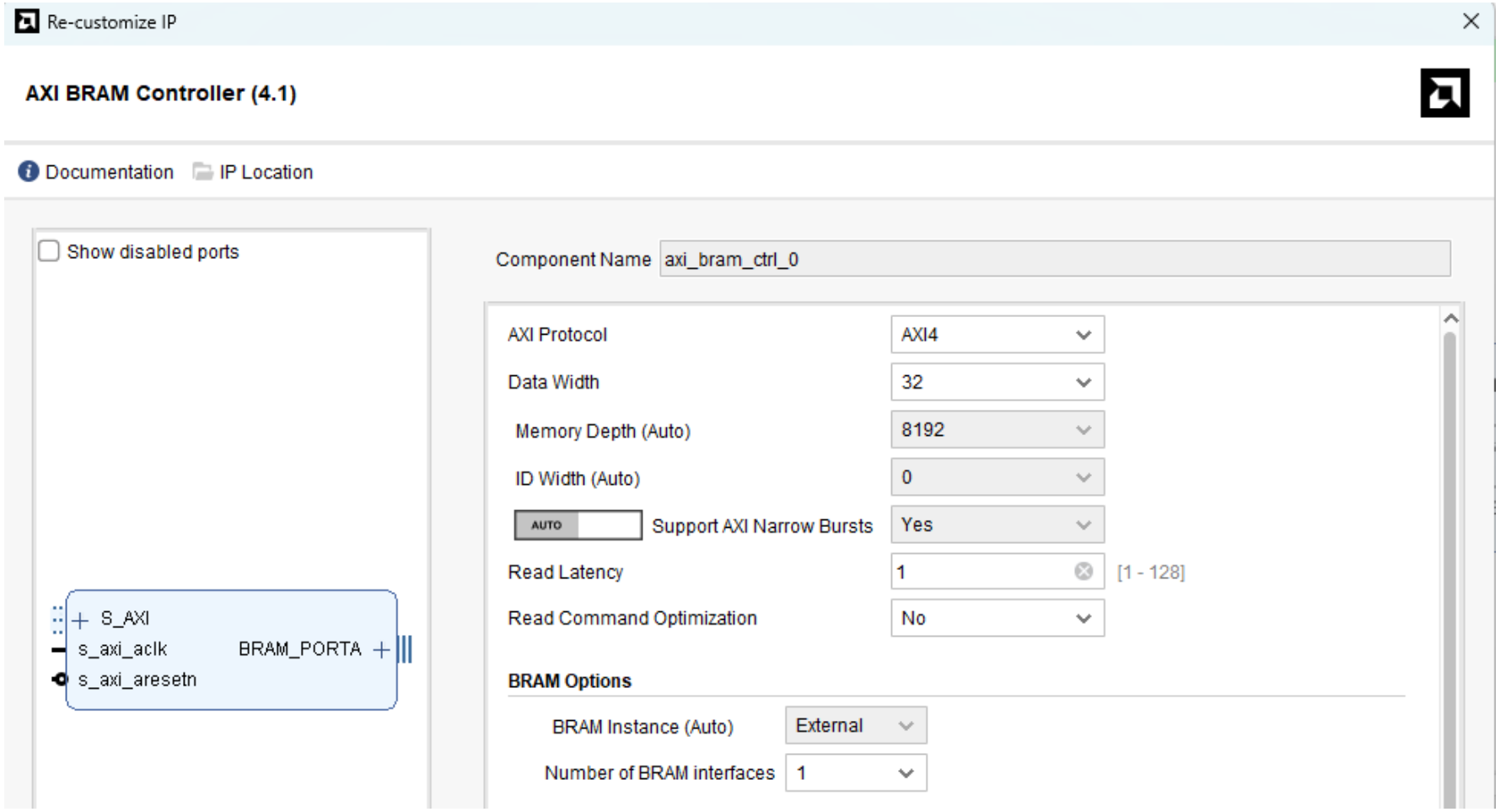
9-2 New Block Design

- Create Block Design
- Add ZYNQ7 Processing System, confirm Run Block Automation



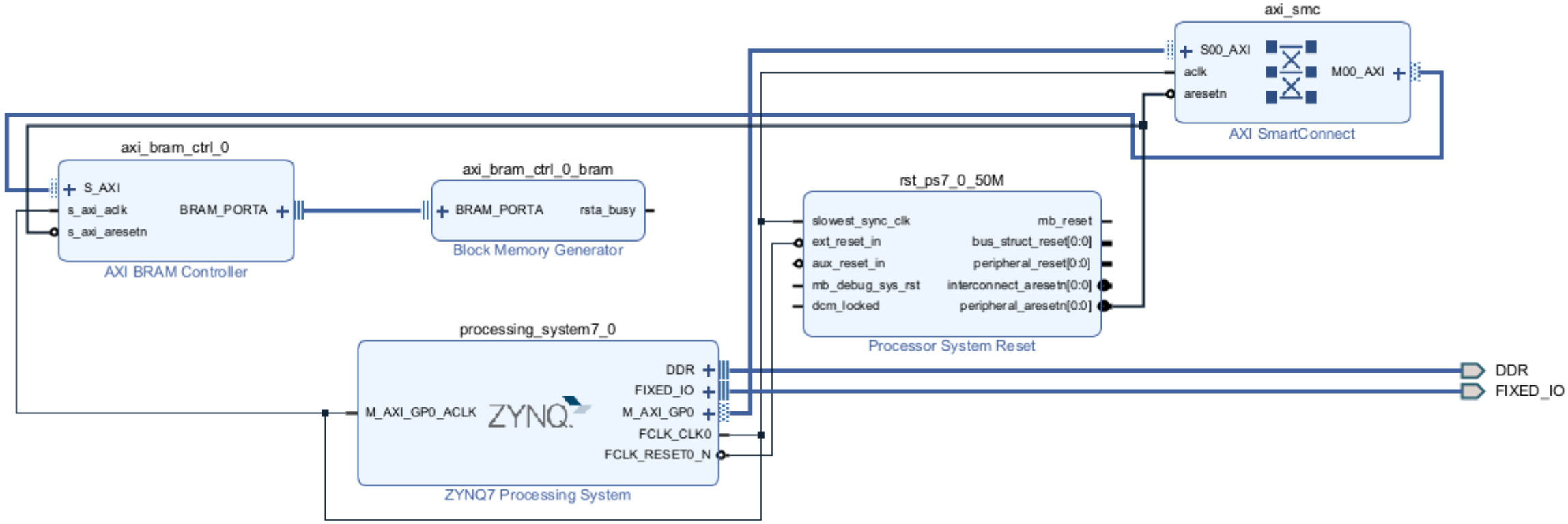
9-2 New Block Design: BRAM Controller

- Add AXI BRAM Controller, double click and and set Number of BRAM interfaces = 1



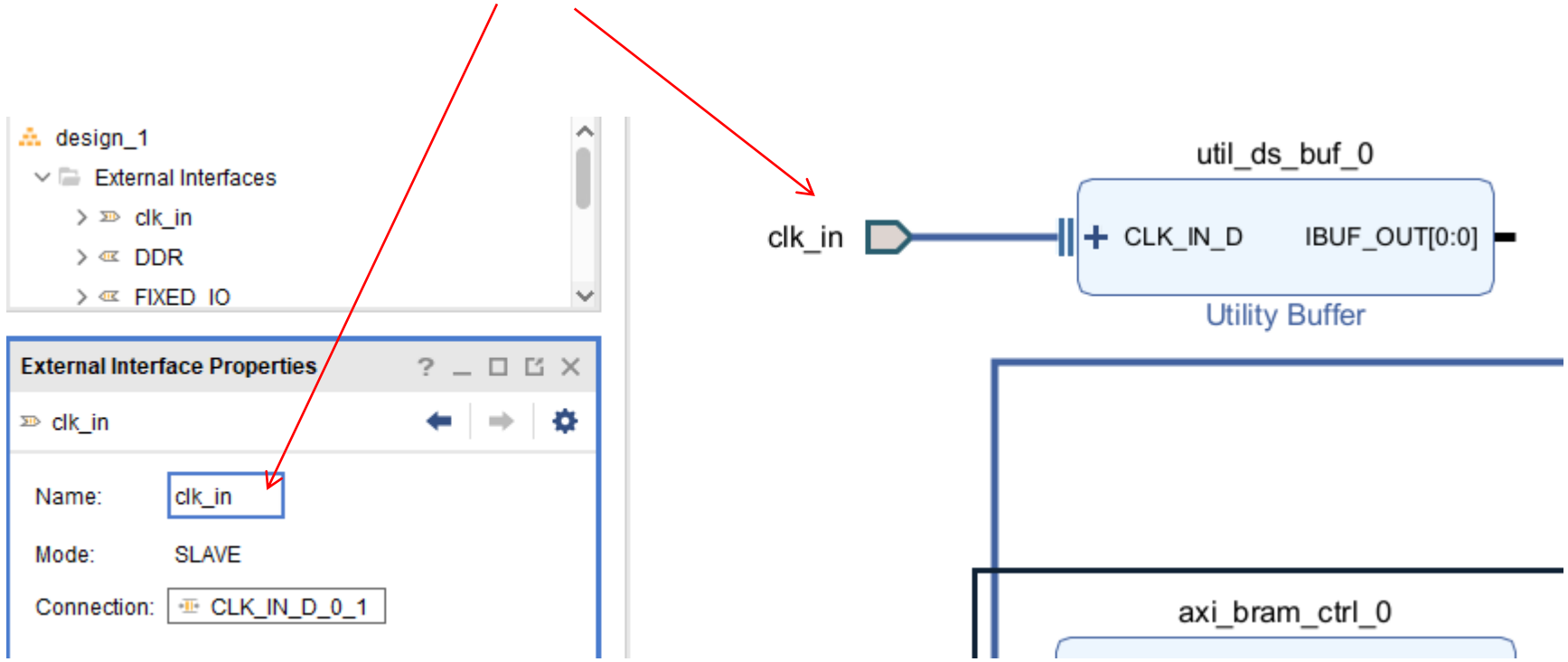
9-2 New Block Design: BRAM Controller

- Run Connection Automation



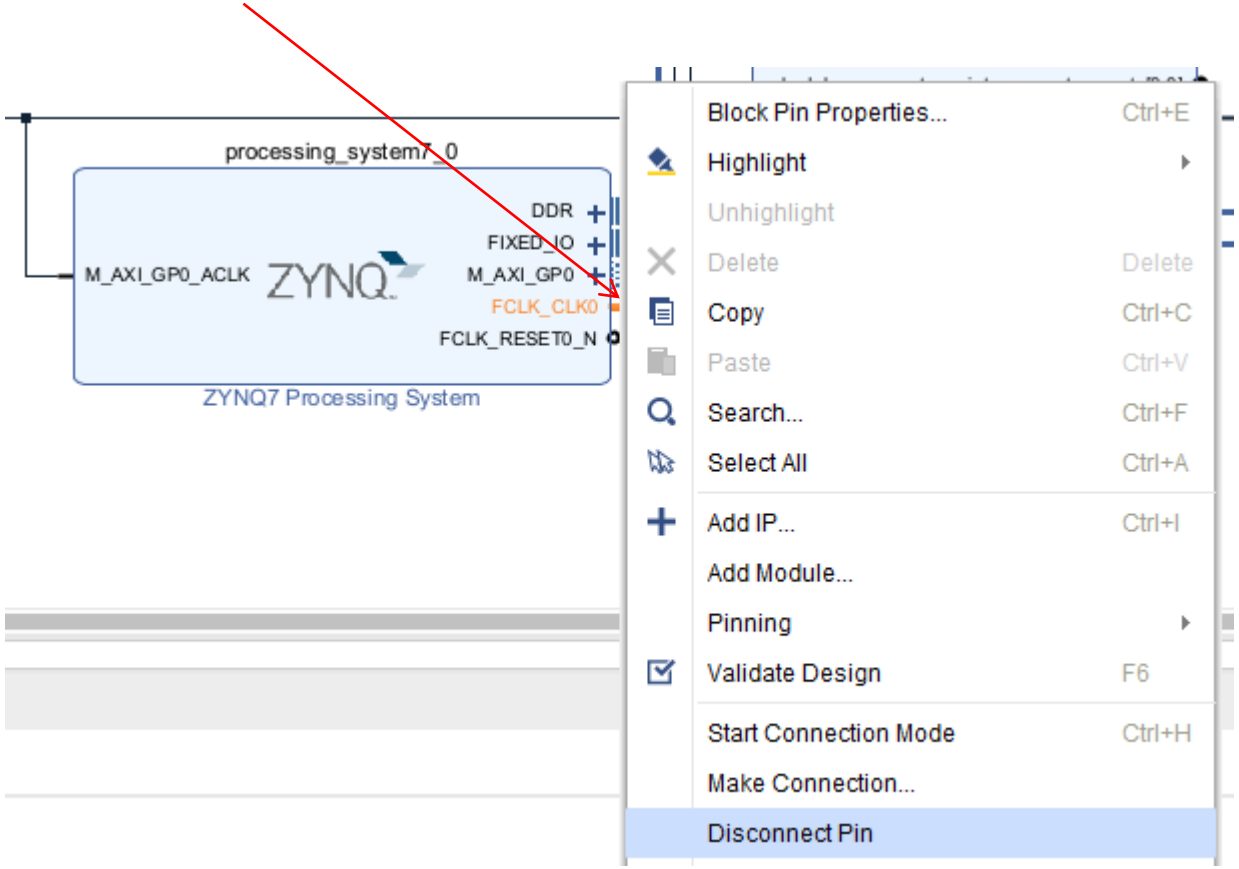
9-2 New Block Design: Clock Buffer

- Add Utility Buffer, right click on input , Make External
- Rename port to clk_in



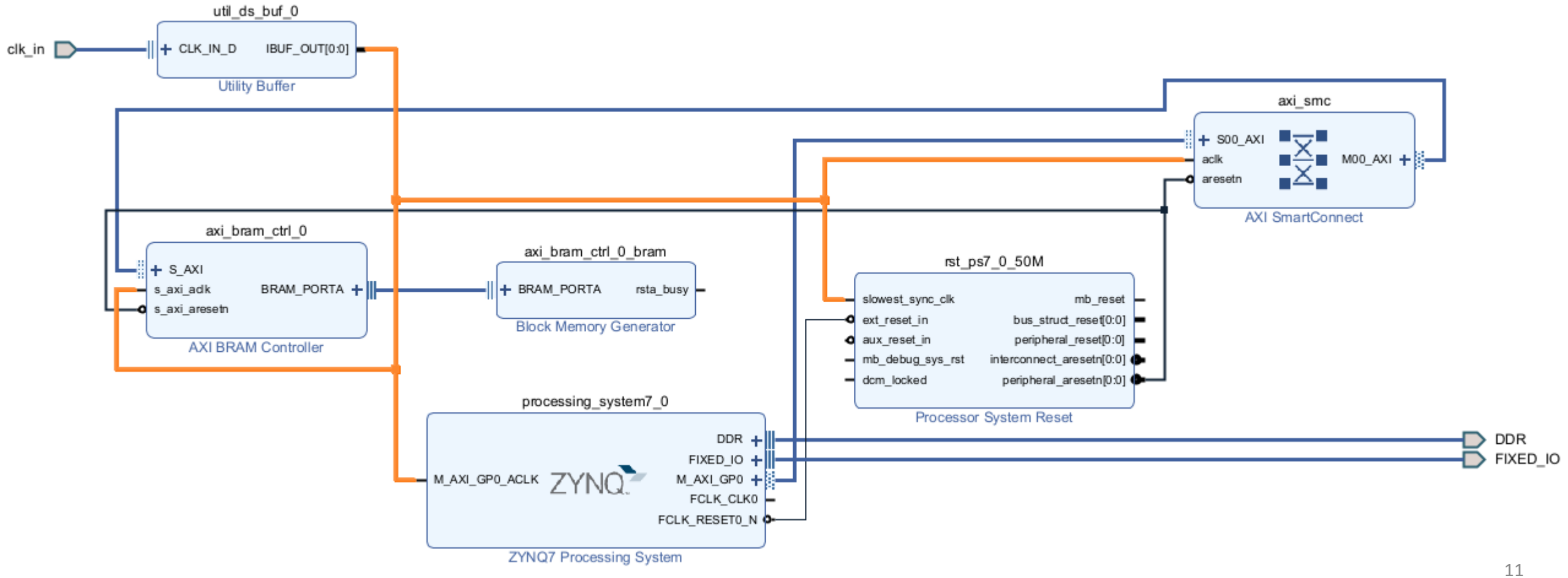
9-2 New Block Design: Connect Clock

- Disconnect FCLK_CLK0 on Zynq Component



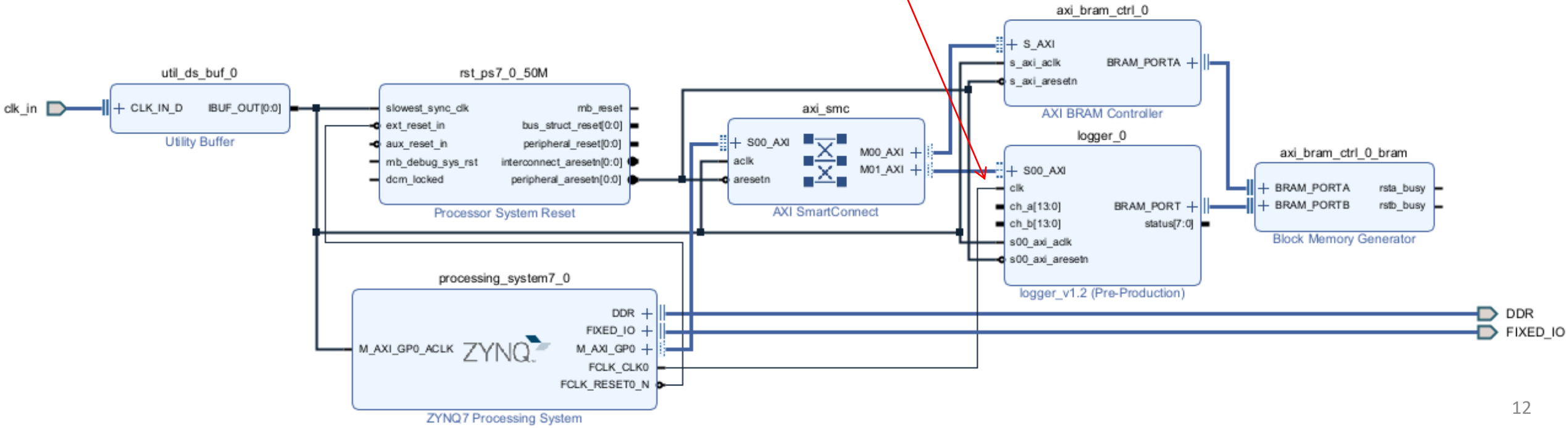
9-2 New Block Design: Connect Clock

- Connect Buffer output clock to M_AXI_GP0_ACLK



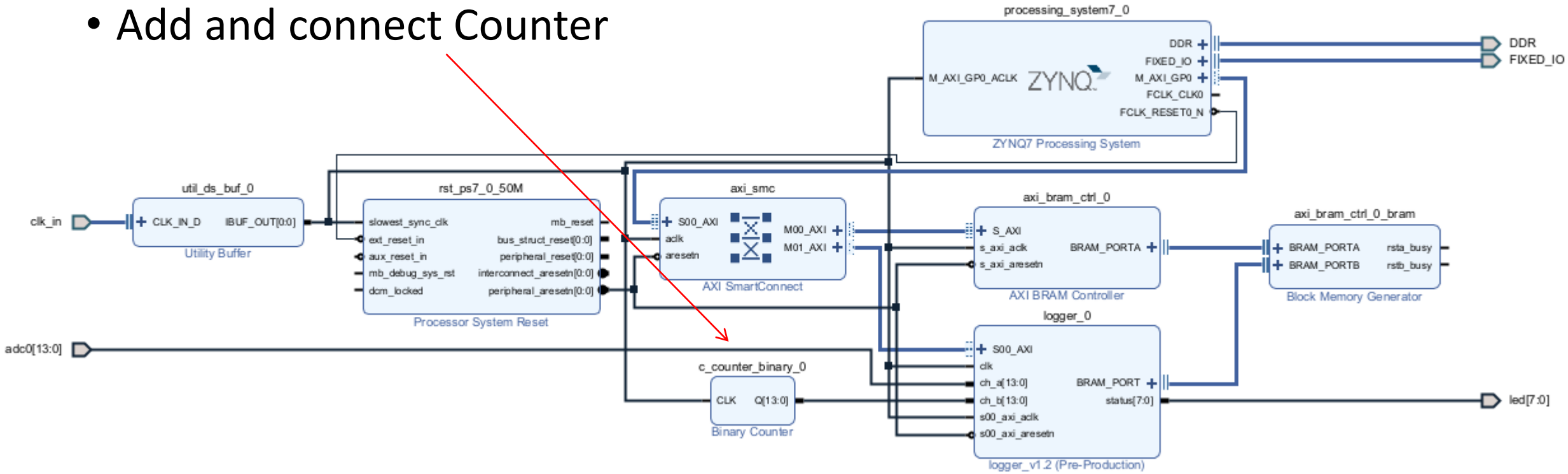
9-2 New Block Design: Logger IP

- Add Logger
- Run Connection Automation
  - clk should not be connected to FCLK_CLK0
  - disconnect and connect to Buffer clock



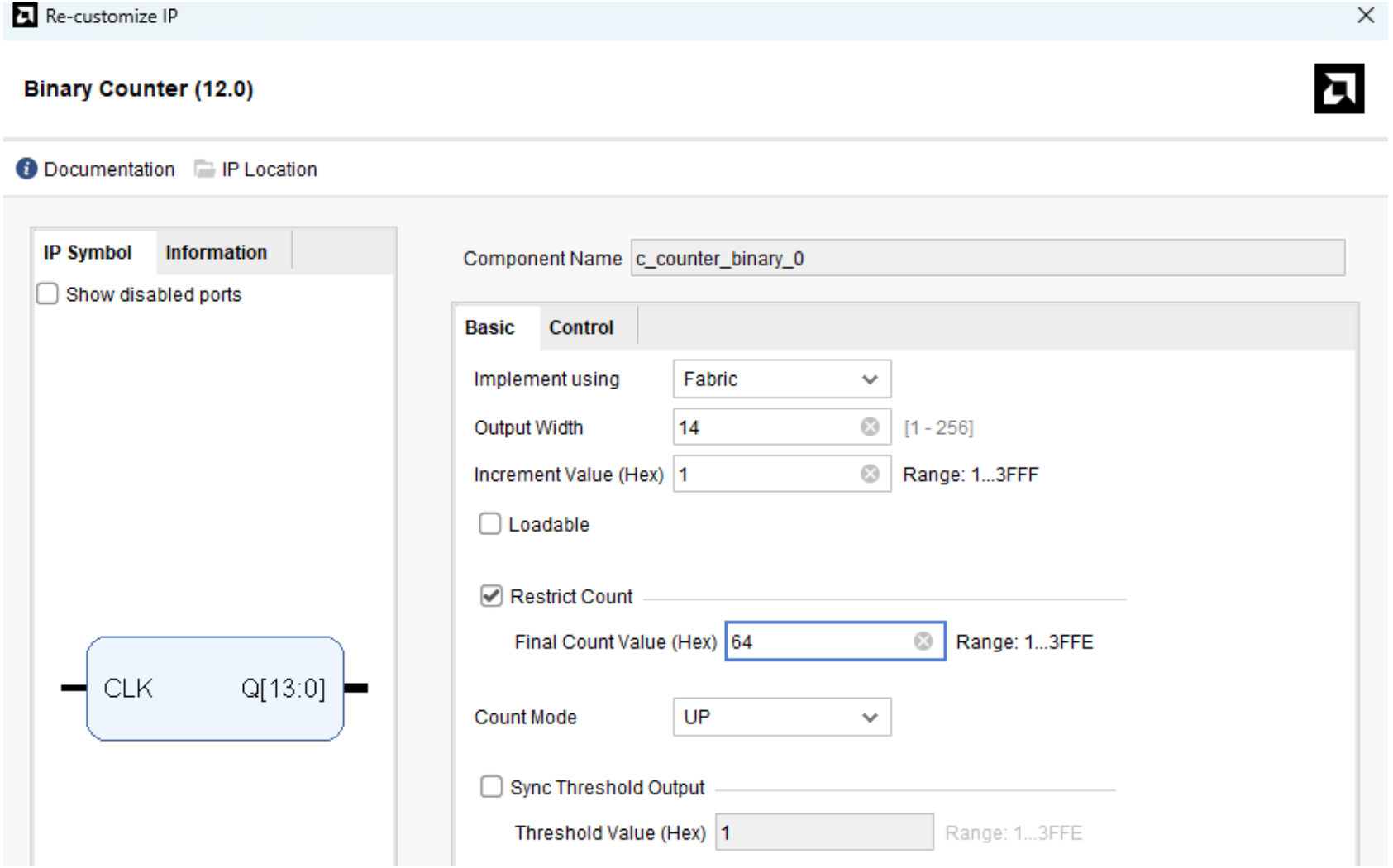
9-2 New Block Design: Logger IP

- Make External ports
  - ch_a input, rename to adc0
  - status output, rename to led
- Add and connect Counter



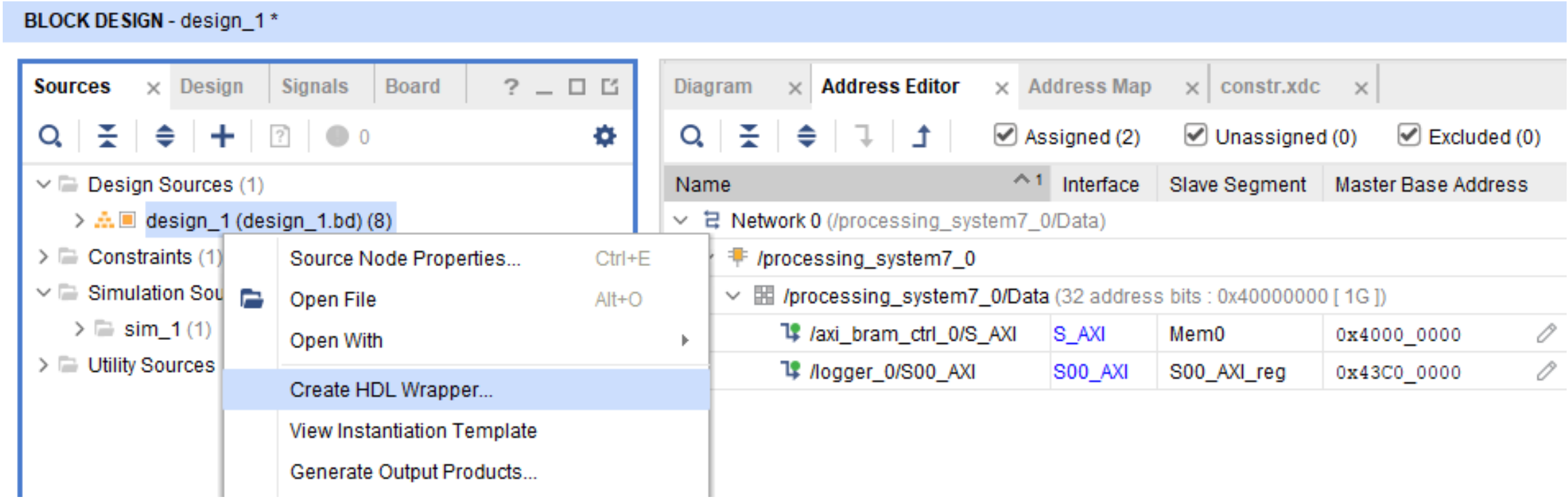
Lab 9-2 New Block Design: Binary Counter

- Binary Counter, set width: 14 and restrict count to 64 (HEX)



9-3 Block Design Wrapper and IP Addresses

- Save Block Design, Validate & Create HDL Wrapper

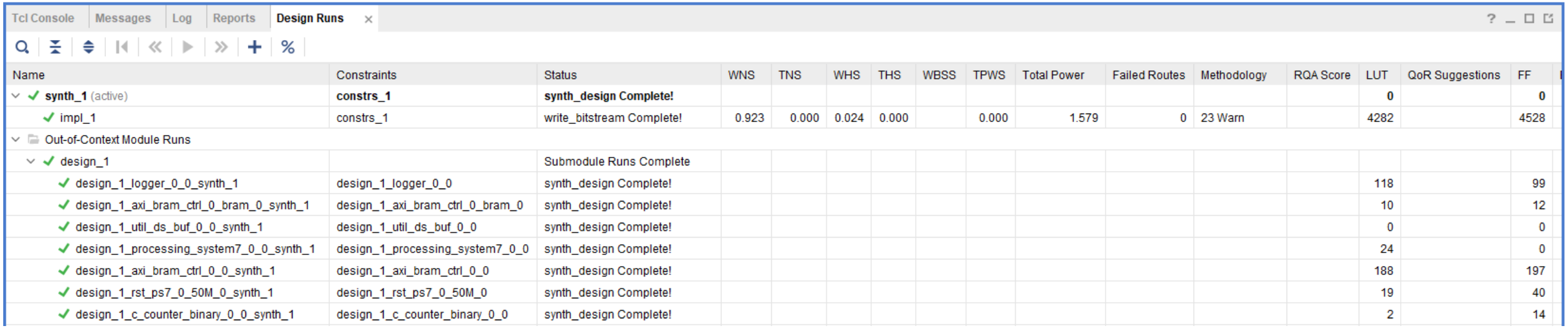


Address Editor

- Check IP Components address range in Address Editor

9-3 Run Synthesis and Implementation

- Click on Generate Bitstream to start running
  - IP and Block Design Synthesis
  - Implementation
  - Bitstream Generation
- Check Design Runs tab

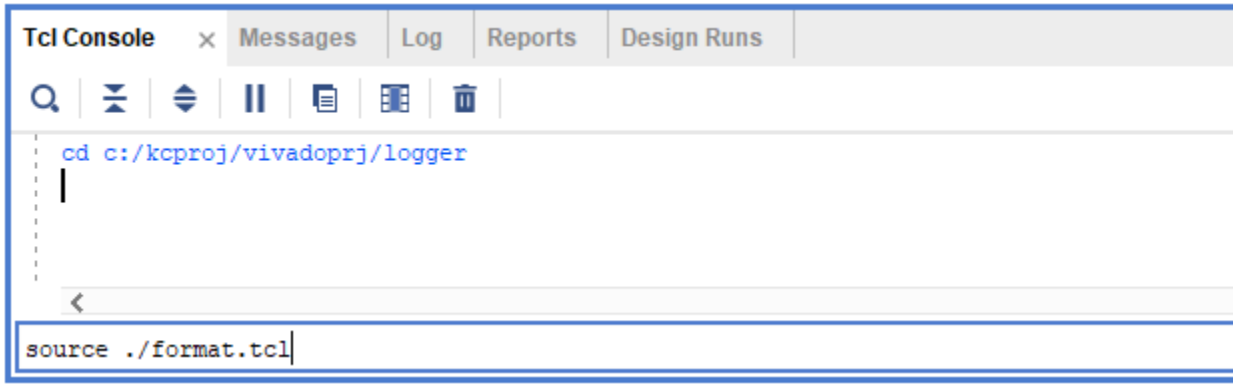


Tcl Console | Messages | Log | Reports | **Design Runs**

Name	Constraints	Status	WNS	TNS	WHS	THS	WBSS	TPWS	Total Power	Failed Routes	Methodology	RQA Score	LUT	QoR Suggestions	FF
synth_1 (active)	**constrs_1**	**synth_design Complete!**											**0**		**0**
impl_1	constrs_1	write_bitstream Complete!	0.923	0.000	0.024	0.000		0.000	1.579	0	23 Warn		4282		4528
Out-of-Context Module Runs															
design_1		Submodule Runs Complete													
design_1_logger_0_0_synth_1	design_1_logger_0_0	synth_design Complete!											118		99
design_1_axi_bram_ctrl_0_bram_0_synth_1	design_1_axi_bram_ctrl_0_bram_0	synth_design Complete!											10		12
design_1_util_ds_buf_0_0_synth_1	design_1_util_ds_buf_0_0	synth_design Complete!											0		0
design_1_processing_system7_0_0_synth_1	design_1_processing_system7_0_0	synth_design Complete!											24		0
design_1_axi_bram_ctrl_0_0_synth_1	design_1_axi_bram_ctrl_0_0	synth_design Complete!											188		197
design_1_rst_ps7_0_50M_0_synth_1	design_1_rst_ps7_0_50M_0	synth_design Complete!											19		40
design_1_c_counter_binary_0_0_synth_1	design_1_c_counter_binary_0_0	synth_design Complete!											2		14

9-3 Format Bitstream

- Tcl Console
  - change to your project folder
  - run format.tcl



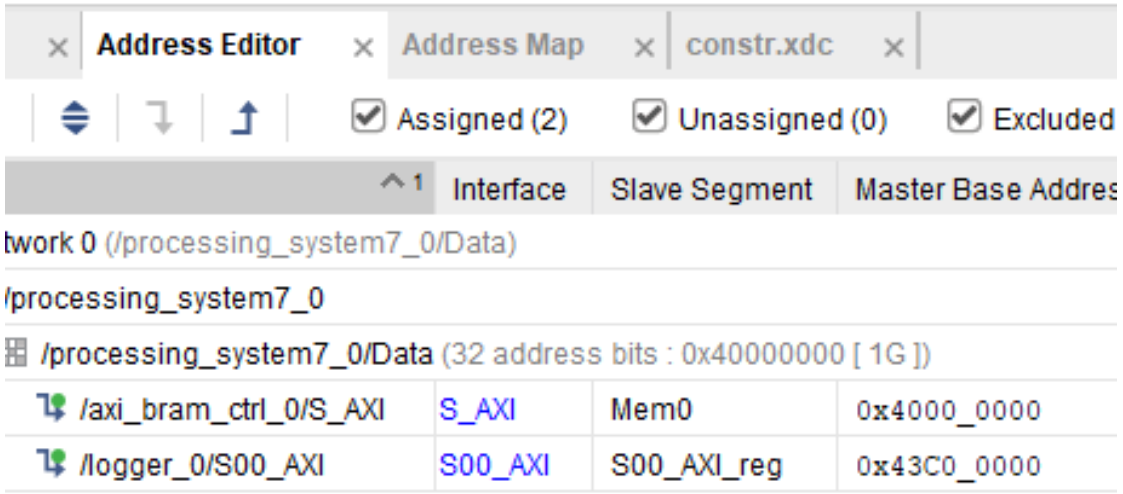
- Check output file in the Vivado Project folder

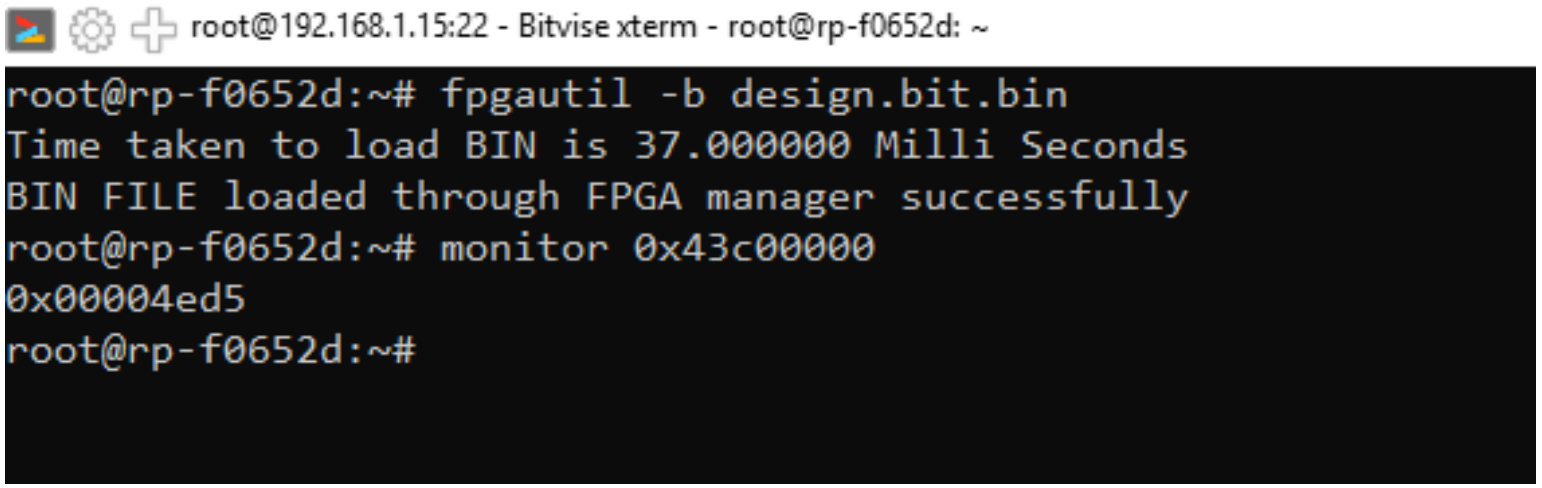
  design.bit.bin

9-4 Upload to RedPitaya

- Open Bitvise SSH Client
- Upload to Red Pitaya
  - design.bit.bin
  - datalog.c (provided source file)
- Open terminal and program FPGA

```
fpgautil -b design.bit.bin
monitor 0x43c00000
```





datalog.c

```
#define CONTROL_BASE_ADDR   0x43C00000
#define DATA_BASE_ADDR      0x40000000
```

AXI Data Logger IP

AXI BRAM Controller

```
// Open physical memory
    fd = open("/dev/mem", O_RDWR | O_SYNC);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    // Map control registers
    ctrl_regs = (volatile uint32_t *)mmap(
        NULL,
        MAP_SIZE,
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        fd,
        CONTROL_BASE_ADDR
    );

    if (ctrl_regs == MAP_FAILED) {
        perror("mmap ctrl_regs");
        close(fd);
        return -1;
    }

    // Map data memory
    data_mem = (volatile uint32_t *)mmap(
        NULL,
        MAP_SIZE,
        PROT_READ,
        MAP_SHARED,
        fd,
        DATA_BASE_ADDR
    );

    if (data_mem == MAP_FAILED) {
        perror("mmap data_mem");
        munmap((void *)ctrl_regs, MAP_SIZE);
        close(fd);
        return -1;
    }
```

```
printf("Starting acquisition...\n");

    // Write 1 to 0x43c00000 to reset logger
    ctrl_regs[0] = 1;

    // Wait 10 ms
    usleep(10000);

    // Write command value to 0x43c00000
    ctrl_regs[0] = command;

    // Write 0 to 0x43c00004 to start logging
    ctrl_regs[1] = 0;

    // Wait 10 ms
    usleep(10000);
```

```
for (uint32_t i = 0; i < num_samples; i++) {
 uint32_t raw = data_mem[i];

 // Split into signed 16-bit values
 int16_t sampleA = (int16_t)(raw & 0xFFFF);
 int16_t sampleB = (int16_t)((raw >> 16) & 0xFFFF);

 // Accumulate sums
 sumA += sampleA;
 sumB += sampleB;

 // Save to CSV
 fprintf(fp, "%u,%d,%d\n", i, sampleA, sampleB);

 // Print first 10 samples
 if (i < 10) {
    printf("%u\t%d\t%d\n", i, sampleA, sampleB);
 }
}
```

9-5 Compile and Run Data Logger

- Compile datalog.c
  gcc -o datalog datalog.c
- Run
  ./datalog
- Check data.csv

```
root@rp-f0652d:~# ./datalog
Command value : 0
Read samples  : 256
Starting acquisition...

First 10 samples:
Idx     SampleA SampleB
0       324     99
1       368     100
2       320     0
3       344     1
4       376     2
5       340     3
6       368     4
7       356     5
8       328     6
9       340     7

Average SampleA: 350.00
Average SampleB: 45.41

Data saved to data.csv
```

- SampleA = analog input, SampleB = test counter

High-Level FPGA Design

Lab 10: Data Logger with Gain IP

In this lab you will:

- Use Vitis HLS to synthesize Gain IP with AXI-Lite interface
- Include Gain IP in Data Logger Vivado project
- use custom SW on RedPitaya for hardware verification

10-1 Clone Vitis Component

- Open Vitis Workspace
- Select gain_ip in VITIS EXPLORER
  - right click and Clone Component
- Edit gain.cpp and add offset input
- Update header file
  - DAT_T and RES_T are fixed-point <14,1>
  - GAIN_T is fixed-point <16, 6>

```
void gain(DAT_T x, DAT_T ofs, GAIN_T k, RES_T& y) {
    y = (x * k) + ofs;
}
```

10-2 Check with C Simulation

- Add ofs to testbench

```
int main() {
    GAIN_T k = 10;
    DAT_T ofs = 0.02;
    std::cout << "k = " << k << ", ofs = " << ofs << std::endl;
```

- Verify with simulation

```
k = 10, ofs = 0.0198975
x = -0.100098, y = -0.981079, y_ref = -0.981079
x = -0.0800781, y = -0.780884, y_ref = -0.780884
x = -0.0600586, y = -0.580688, y_ref = -0.580688
x = -0.0400391, y = -0.380493, y_ref = -0.380493
x = -0.0200195, y = -0.180298, y_ref = -0.180298
x = -0.00012207, y = 0.0186768, y_ref = 0.0186768
x = 0.0198975, y = 0.218872, y_ref = 0.218872
x = 0.039917, y = 0.419067, y_ref = 0.419067
x = 0.0599365, y = 0.619263, y_ref = 0.619263
x = 0.0799561, y = 0.819458, y_ref = 0.819458
x = 0.0999756, y = 0.999878, y_ref = 1.01965
```

10-3 Add Directives

- Add to gain() PIPELINE Directive
- Add to k and ofs INTERFACE directive
  - select s_axilite, set bundle: setting

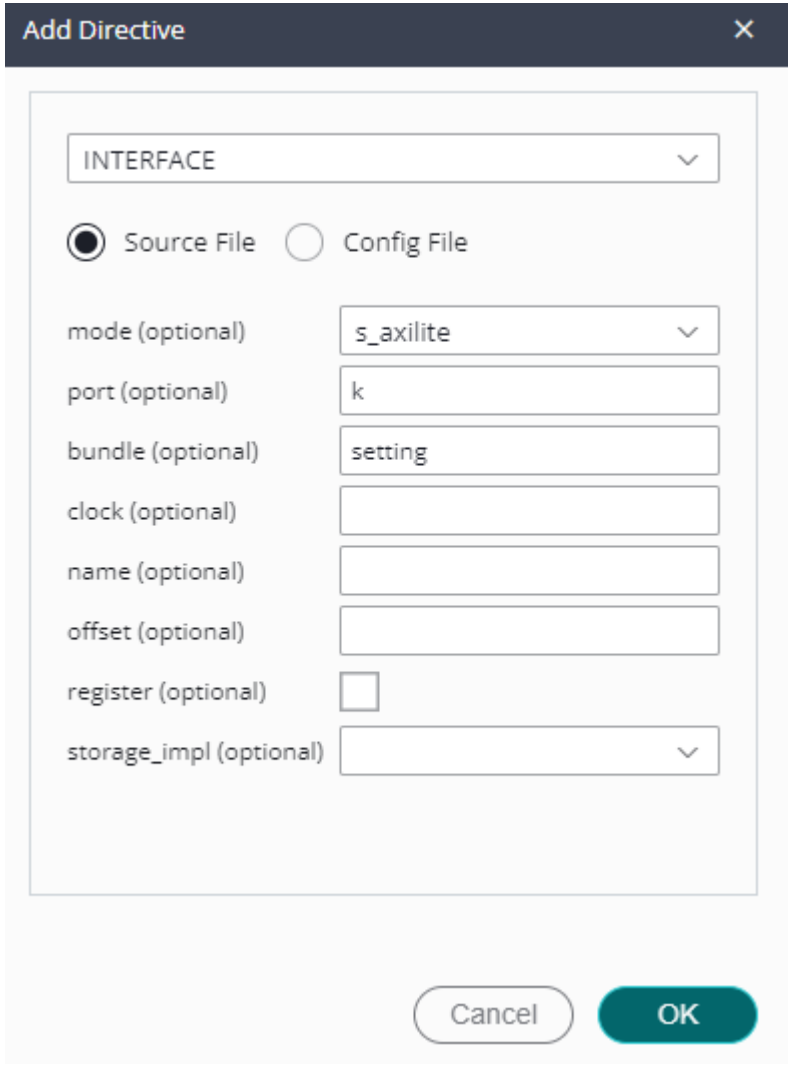
```
void gain(DAT_T x, DAT_T ofs, GAIN_T k, RES_T& y) {
#pragma HLS INTERFACE ap_none port=x
#pragma HLS INTERFACE ap_none port=y

#pragma HLS INTERFACE s_axilite port=ofs bundle=setting
#pragma HLS INTERFACE s_axilite port=k bundle=setting

#pragma HLS INTERFACE ap_ctrl_none port=return

#pragma HLS PIPELINE II=1

    y = (x * k) + ofs;
}
```



10-3 Synthesize GAIN with AXI Interface

- Run C Synthesis
  - set clk: 8ns
- Check Synthesis Report

Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	INTERVAL	PIPELINED	STRUCTURE	BRAM	DSP	FF	LUT	URAM
gain	3	1	yes	function	0	1	211	262	0

HW Interfaces

S_AXILITE Interfaces

INTERFACE	DATA WIDTH	ADDRESS WIDTH	OFFSET	REGISTER
s_axi_setting	32	5	16	0

S_AXILITE Registers

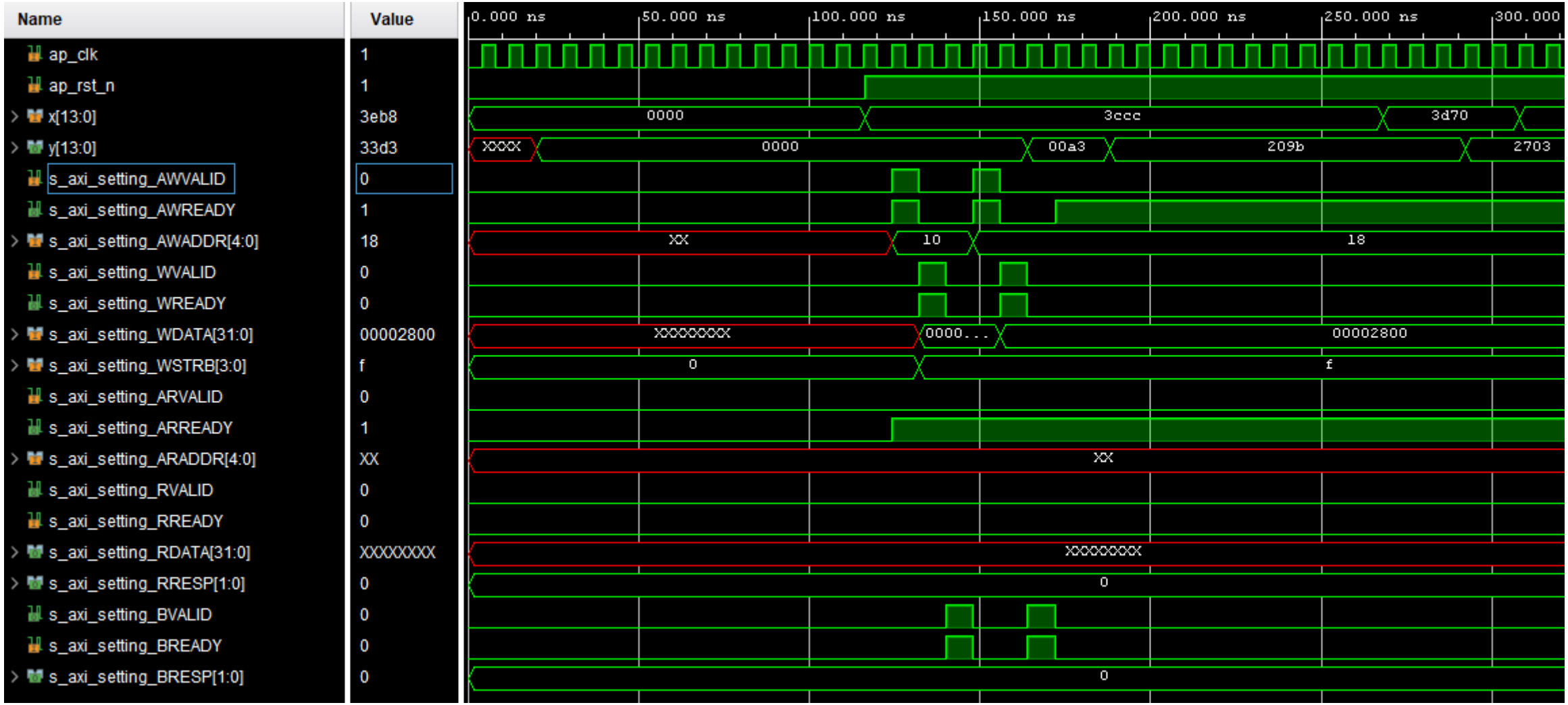
INTERFACE	REGISTER	OFFSET	WIDTH	ACCESS	DESCRIPTION
s_axi_setting	ofs	0x10	32	W	Data signal of ofs
s_axi_setting	k	0x18	32	W	Data signal of k

Other Ports

PORT	MODE	DIRECTION	BITWIDTH
x	ap_none	in	14
y	ap_none	out	14

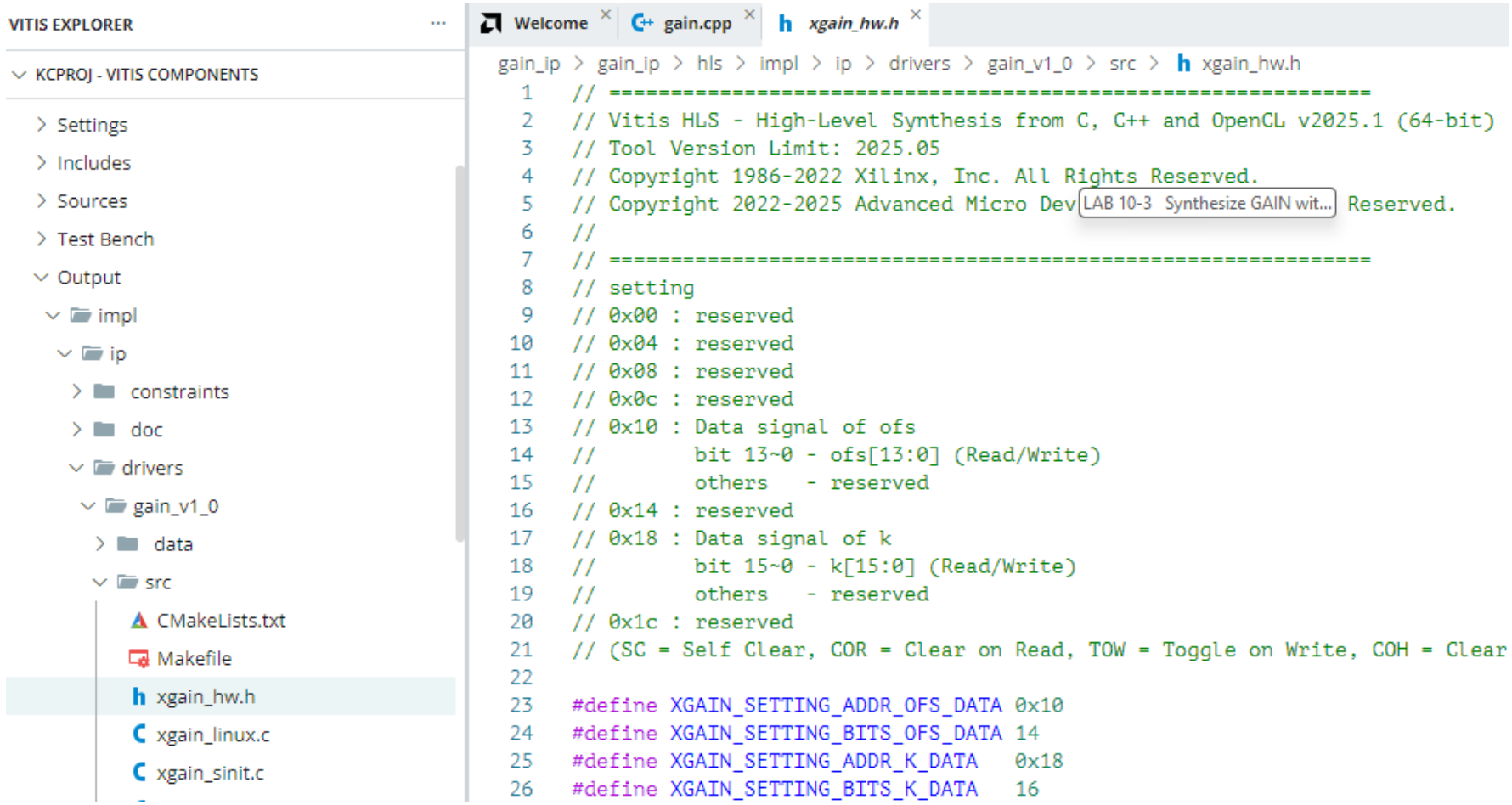
10-4 Run C/RTL Co-Simulation

- Observe AXI-Lite transactions



10-5 Export IP

- Run Package IP
- Check Output, find: impl/ip/drivers/gain_v1_0/src/xgain_hw.h



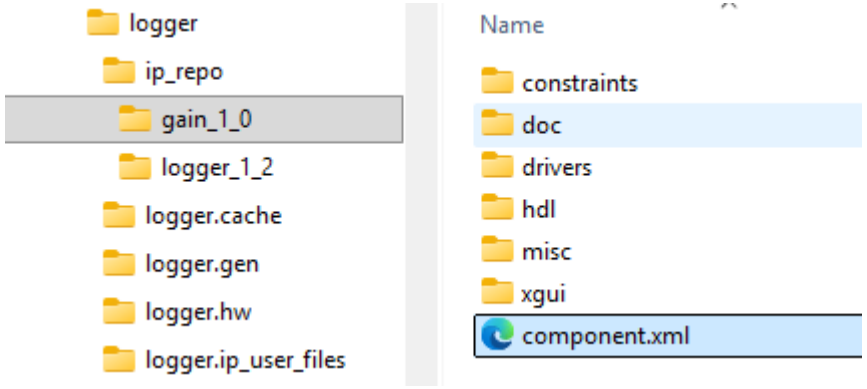
10-5 Copy Exported IP to Vivado ip_repo

Vivado Project (lab9):

- Create New Folder **gain_1_0** in vivado\ip_repo

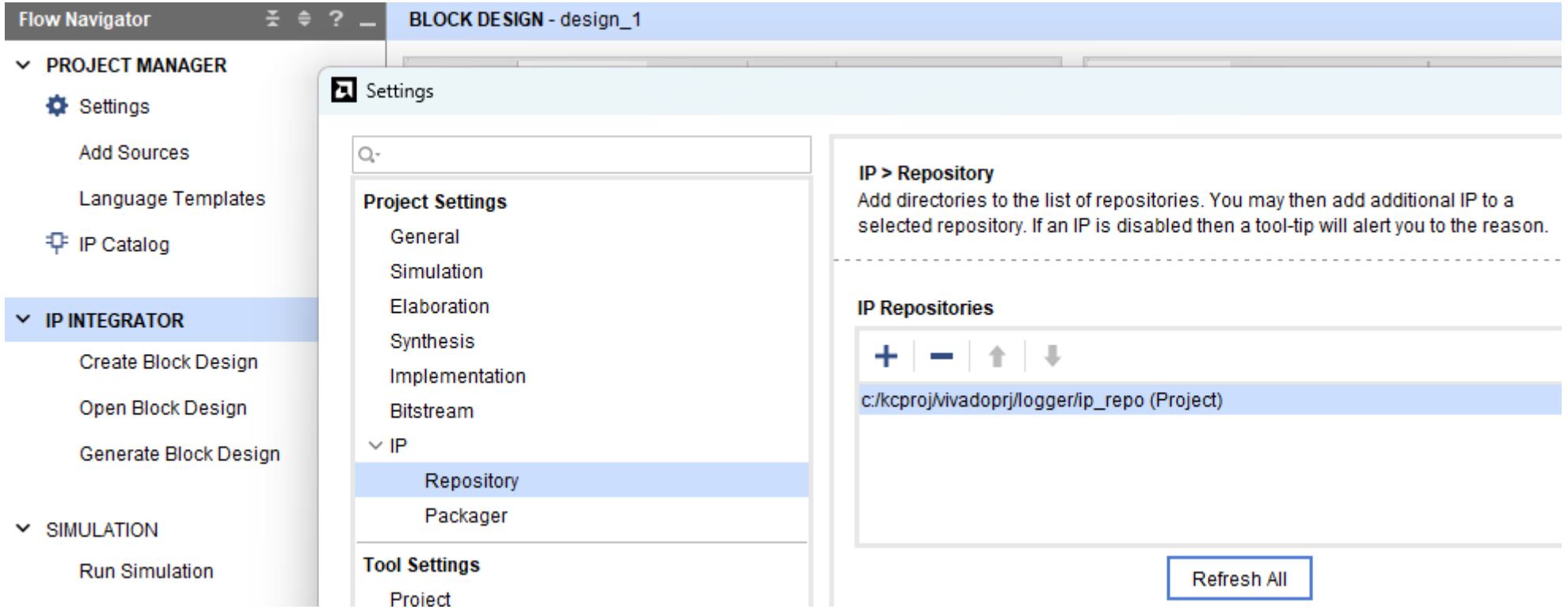
Vitis Project (lab10)

- Open Vitis folder ..\hls\impl\ip
- Extract xilinx_com_hls_gain_1_0.zip to ip_repo\gain_1_0



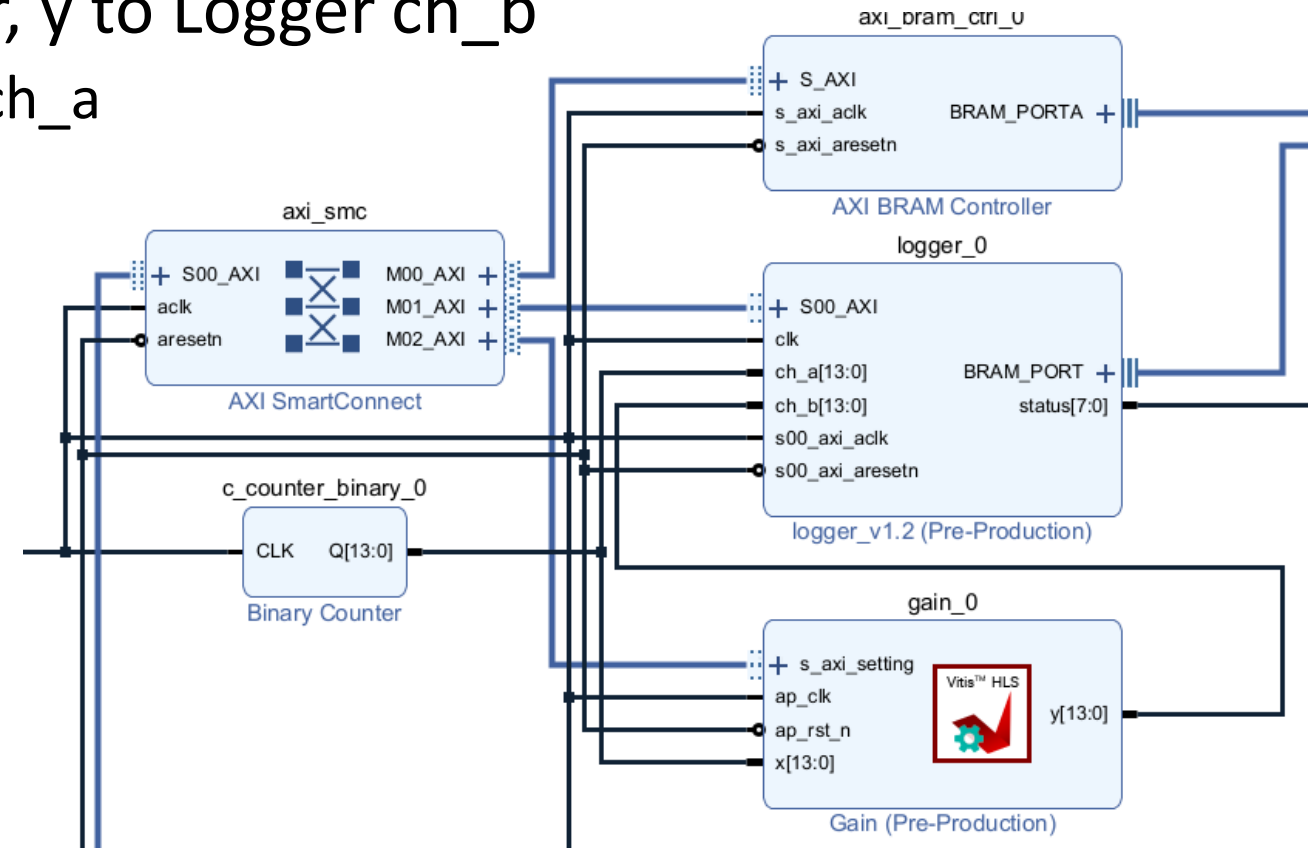
10-6 Vivado Project

- Open Data Logger Project (lab9)
- Settings, Refresh IP Repositories



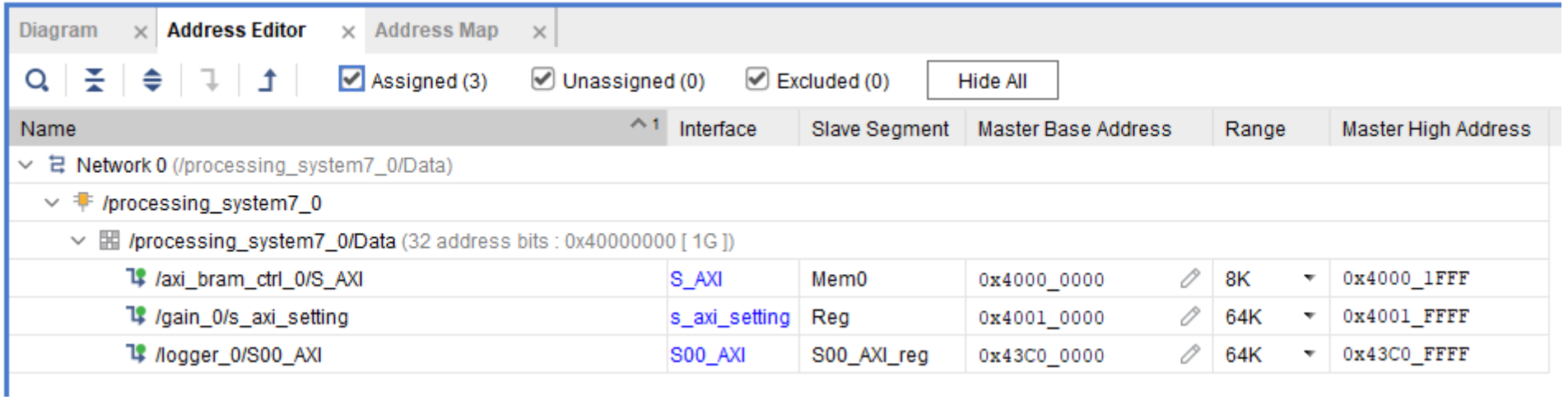
10-7 Logger Project: Gain IP

- Open Block Design
- Add Gain IP, Run Connection Automation
- Connect x to Counter, y to Logger ch_b
  - connect Counter to ch_a for test purpose (disconnect adc0)



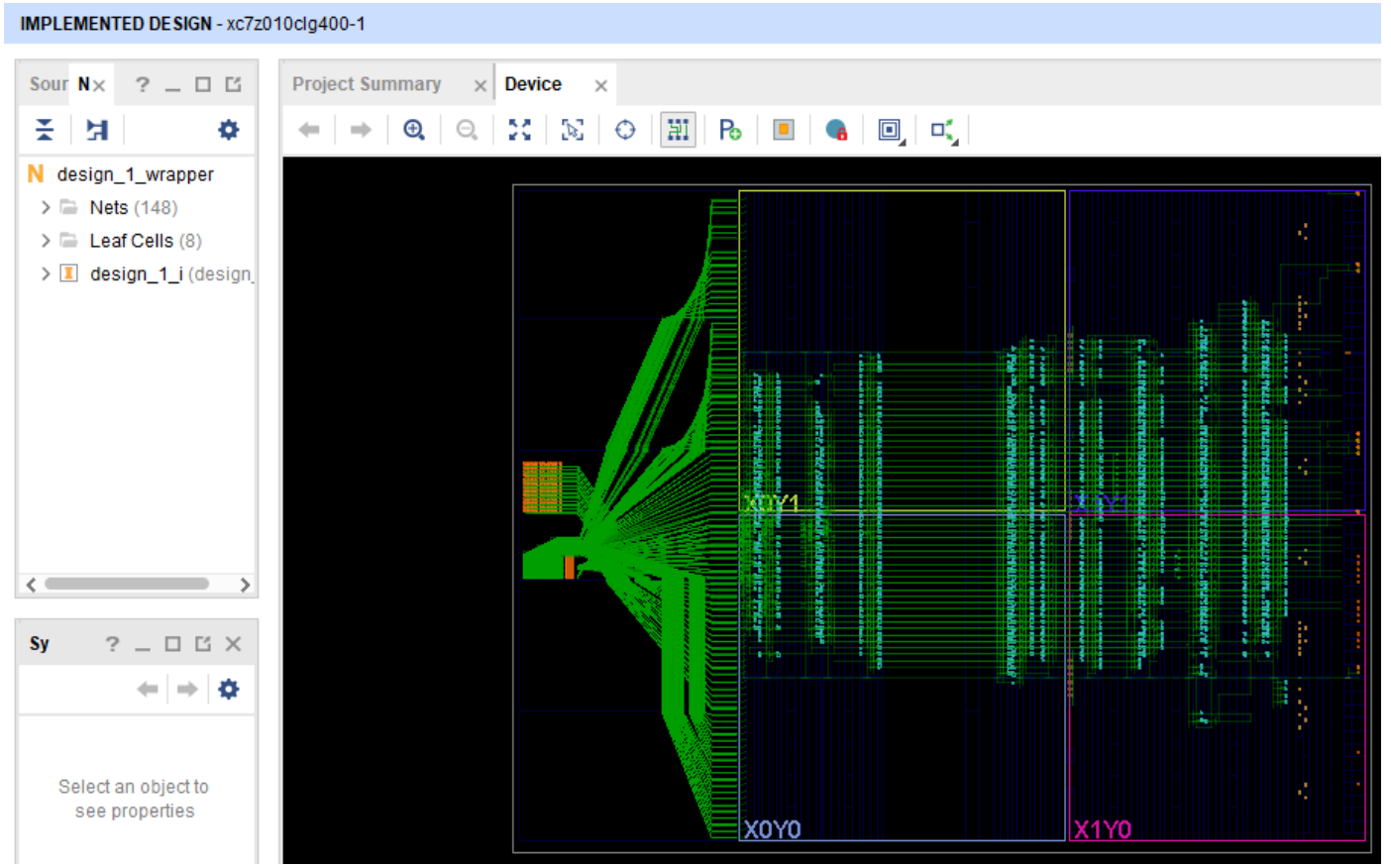
10-7 Check Address Editor

- Save Block Design and Validate
- Open Address Editor



10-8 Synthesize and Generate Bitstream

- Click on Generate Bitstream and wait to finish
- Open Implemented Design
- Tcl Console
  - change to project folder
  - run format.tcl



10-9 Test on RedPitaya

- Open Bitvise SSH Client
- Upload design.bit.bin to Red Pitaya
- Open terminal and program FPGA

```
fpgautil -b design.bit.bin
monitor 0x43c00000
monitor 0x40010018 512
./datalog
```

k = 0.5

```
root@rp-f0652d:~# monitor 0x40010018 512
root@rp-f0652d:~# ./datalog
Command value : 0
Read samples  : 256
Starting acquisition...

First 10 samples:
Idx     SampleA SampleB
0       67      32
1       68      33
2       69      33
3       70      34
4       71      34
5       72      35
6       73      35
7       74      36
8       75      36
9       76      37
```

High-Level FPGA Design

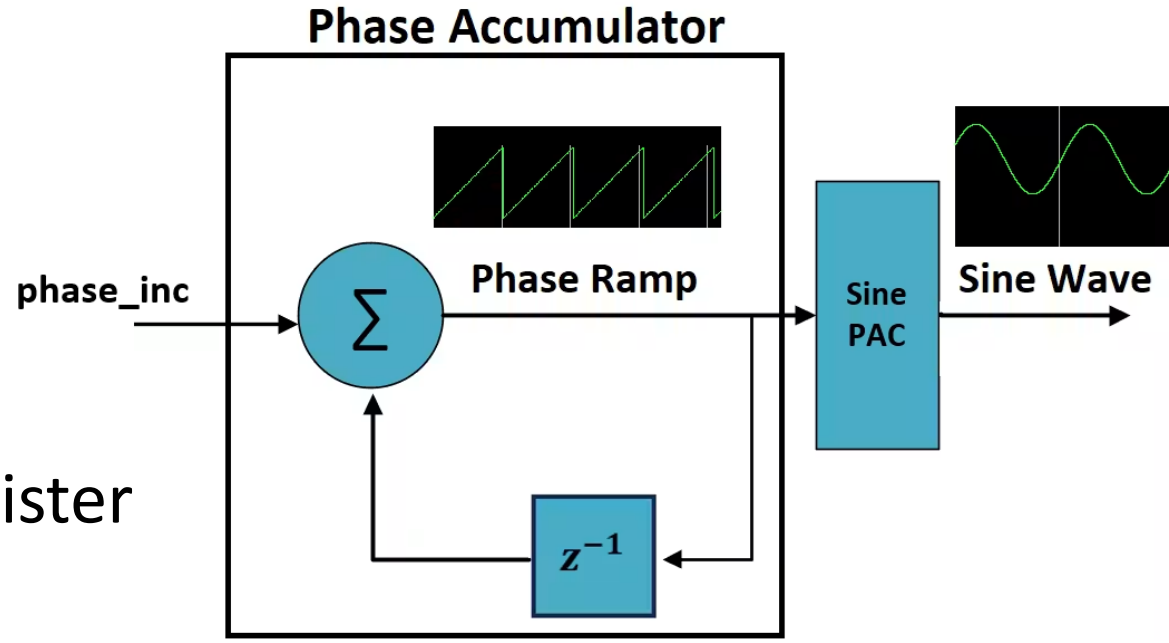
Lab11: Numerically controlled oscillator

In this lab you will:

- Design and synthesize numerically controlled oscillator
- Include NCO IP in Data Logger Vivado project
- Perform hardware verification on Red Pitaya

Numerically Controlled Oscillator (NCO)

- digital signal generator



The NCO consists of:

- Phase accumulator
- A programmable phase increment register
- Waveform generation logic
  - 32-bit phase accumulator
    output frequency

$$f_{out} = \frac{\text{phase\_inc}}{2^{32}} f_{clk}$$

11-1 Create New Component

```
proj/
├── src/
│   └── 11_nco/          ← lab11 C++ sources
│       ├── nco.cpp
│       ├── nco.h
│       └── tb_nco.cpp
│
├── vitis/               ← Vitis workspace (location)
│   ├── nco_hls/         ← lab11 HLS component
```

- File, New Component, Name: **nco_hls**
- Source: nco.cpp, nco.h
- Testbench: tb_nco.cpp
- Part xc7z010clg400-1, clk=8ns

11-1 NCO Model (Phase Accumulator)

- The accumulator acts as a modulo counter
- The upper phase bits form the waveform
- The waveform is scaled by amp_reg, output keeps upper 14 bits

```
void nco(
    ap_uint<32> phase_inc,   // AXI input
    ap_uint<14> amplitude,   // AXI input
    ap_uint<1>  enable,
    ap_uint<14> &saw
) {
    static ap_uint<32> phase = 0;
    static ap_uint<14> amp_reg = 8192;

    amp_reg = amplitude;     // Latch AXI value

    if (enable) {
        phase = phase + phase_inc;
    }

    ap_uint<14> raw_saw = phase.range(31,18);

    ap_uint<28> scaled = raw_saw * amp_reg;
    saw = scaled.range(27,14);
}
```

11-2 C Simulation

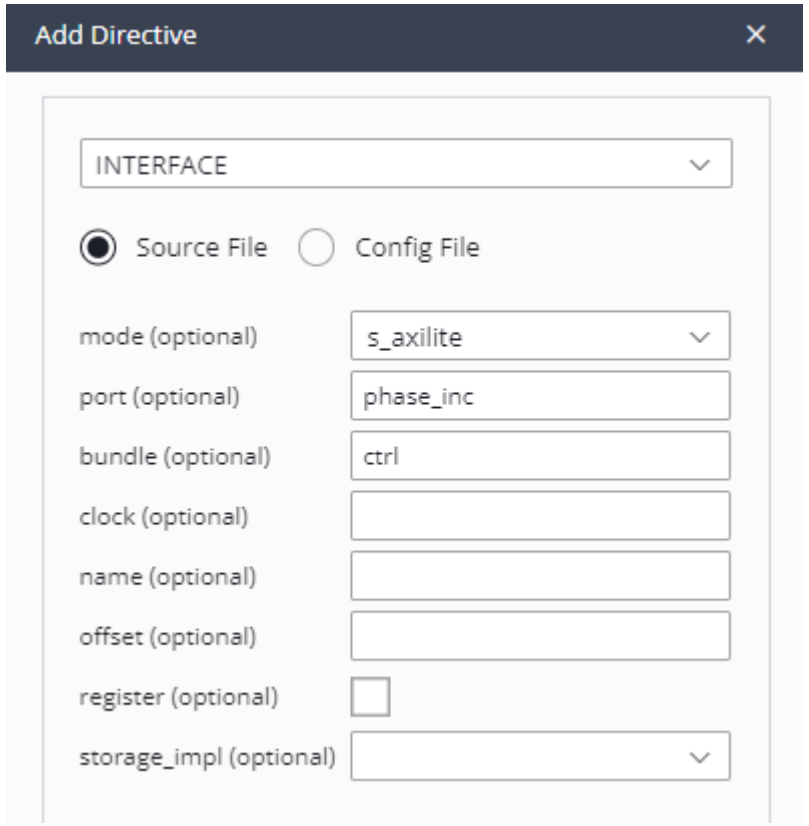
- Larger phase_inc values produce higher output frequency
- Run C Simulation

```
ap_uint<32> phase_inc = 0x10000000; // moderate frequency
ap_uint<14> amplitude = 8192;       // 50%
ap_uint<1>  enable    = 1;
```

```
[TB] Cycle  Saw
[TB] 0  512
[TB] 1  1024
[TB] 2  1536
[TB] 3  2048
[TB] 4  2560
[TB] 5  3072
[TB] 6  3584
[TB] 7  4096
[TB] 8  4608
[TB] 9  5120
[TB] 10 5632
[TB] 11 6144
[TB] 12 6656
[TB] 13 7168
[TB] 14 7680
[TB] 15 0
[TB] 16 512
```

11-3 Directives and Synthesis

- Add Directive INTERFACE s_axilite to phase_inc and amplitude
  - define the same AXI bundle
- Add INTERFACE ap_none to port saw
- Add PIPELINE and INTERFACE ap_ctrl_none to nco()
- Run C Synthesis



Latency	Interval	DSP	FF	LUT
1	1	1	168	195

11-4 NCO with Sine Output

- Add output **sine**
  - data type ap_int<14>
- Add phase to sine LUT transformation
  - nco.h contains 256-entry sine table
- Update Test Bench and simulate

```
void nco(
   ap_uint<32> phase_inc,   // AXI input
   ap_uint<14> amplitude,   // AXI input
   ap_uint<1>  enable,
   ap_uint<14> &saw,
   ap_int<14> &sine
) {
...
   // Sine LUT address
   ap_uint<8> lut_addr = phase.range(31,24);

   sample_t lut_value = sine_lut[lut_addr];

   // Scale sine amplitude
   ap_int<28> sine_scaled = lut_value * amplitude;

   sine = sine_scaled.range(27,14);
```

11-5 NCO IP Component

- Package IP Component
- Include in Vitis Block Design and verify on Red Pitaya