Parallel and Reconfigurable VLSI Computing (7)

Practical RTL Design

Hiroki Nakahara
Tokyo Institute of Technology

Outline

- Practical RTL design methodology
  - From behavior (C/C++ code) to HDL one
- Interface co-design
  - Control a hardware from an ARM processor
- RTL design optimization

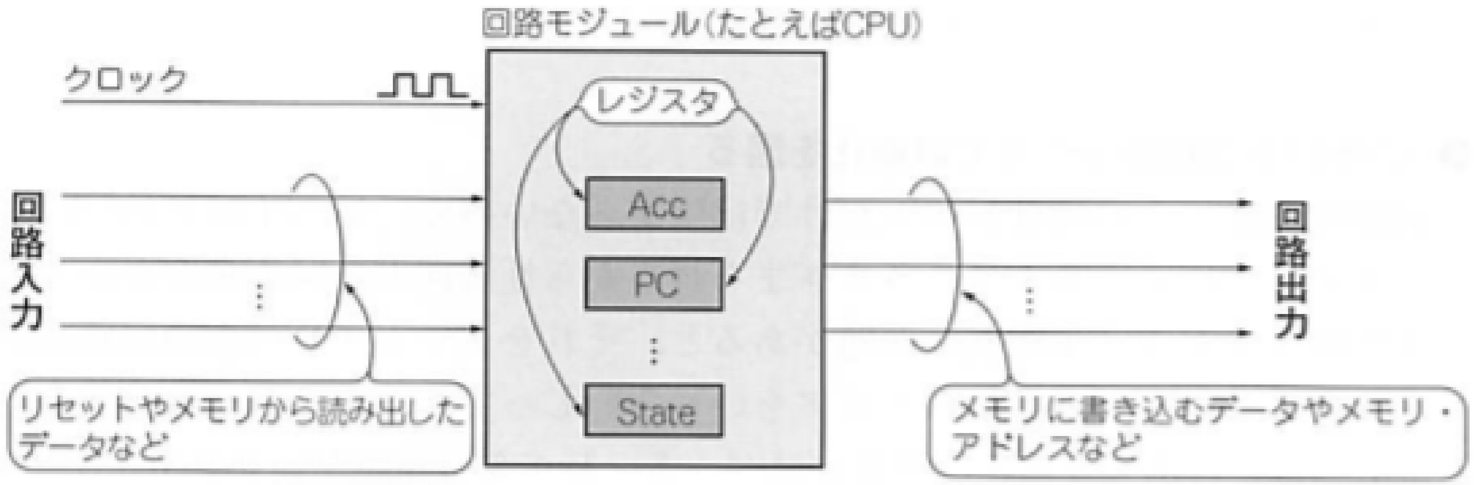
Practical RTL Design Methodology

C/C++ to RTL

- Determine the specifications of the circuit
  - Timing chart, state transition diagram, performance, block diagram
- Consider the configuration of the module
  - Design and combine for each IP core
- Function assigned to each core and its resource
  - Including a consideration of interface
  - Often written in C/C ++
  - It becomes a testbench for a verification
- Convert C/C++ description to RTL
- Optimize behaviors (pipelining and parallelization)
  - Automation by the remaining work with CAD

C/C++ Description for a Concept of Module

- Approximate ~300 lines for single function
- Data input/output (Interface)
- Data processing



Case Study: FIR Filter

- x_n: N sampling signals and y_n: Output signal, then

$$y_n = \sum_{k=0}^{N-1} h_k x_{n-k}$$

, where a filter coefficient h_n is given by

$$h_n = \frac{\rho_n}{2\pi} \int_0^{2\pi} d\tilde{\omega} e^{i\tilde{\omega}(n-\tilde{\tau})} H_0(\tilde{\omega}).$$

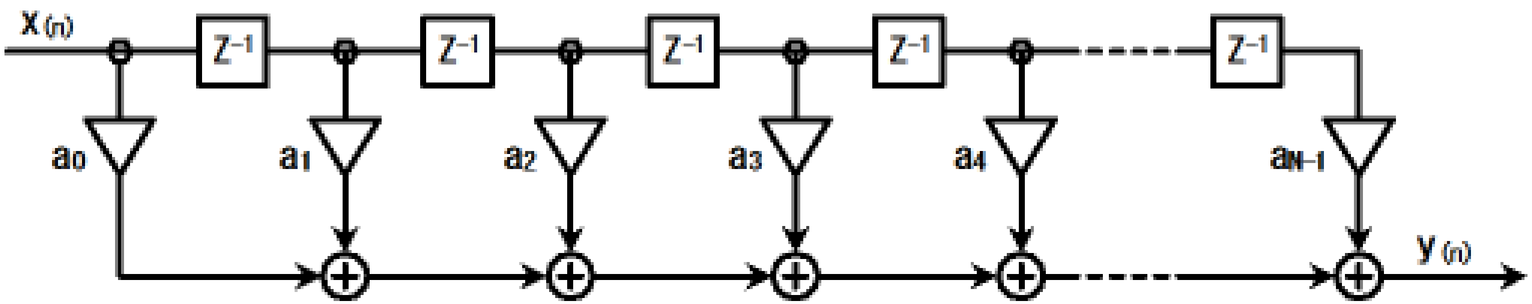
, where $\tilde{\omega}=2\pi\omega/\omega_s$ denotes normalized freq., ω_s denotes sampling freq., $H_0(\tilde{\omega}) \in R$ denotes frequency characteristic, p_n denotes window function, and $\tilde{\Gamma}=(N-1)/2$.

Cont'd

- Deference equation for a FIR filter:

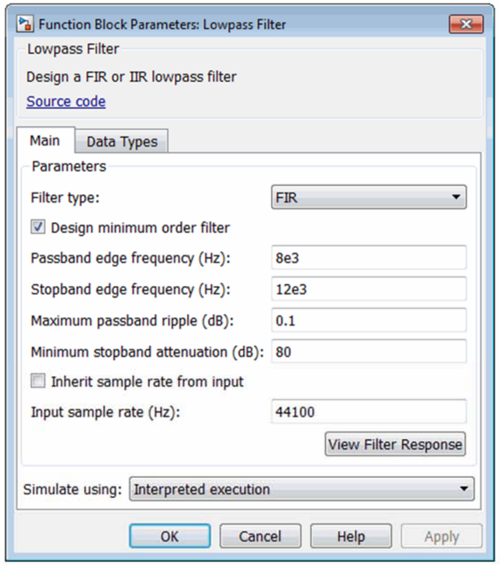
$$y[n] = a_0 x[n] + a_1 x[n-1] + a_2 x[n-2] + a_3 x[n-3] + a_4 x[n-4] + \cdots + a_{N-1} x[n-(N-1)]$$

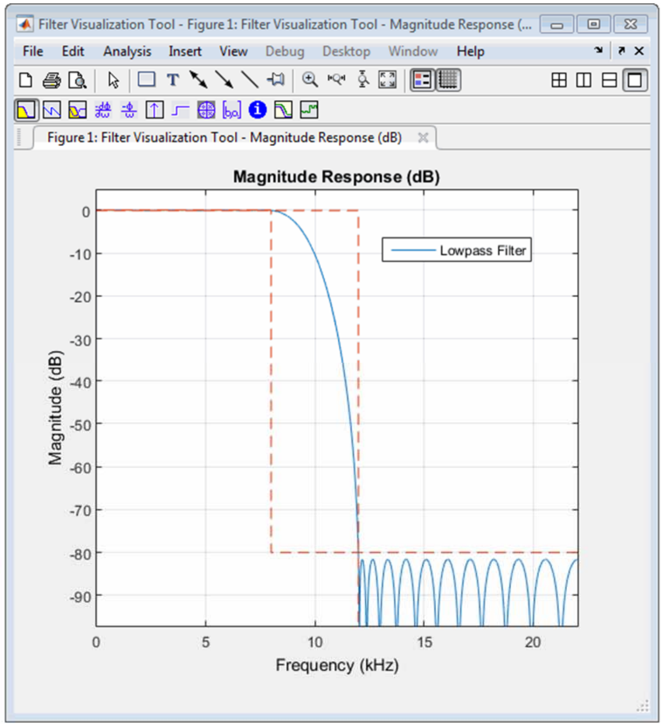
- Diagram for a FIR filter:



FIR Filter Coefficient Design

- MathWorks Matlab with DSP System Toolbox
  - Sampling Freq.: 44.1 kHz
  - LPF for 20 kHz → Normalized cut-off freq.
  - #Taps: 11
  - Window function: Hamming





C Behavior for a FIR Filter

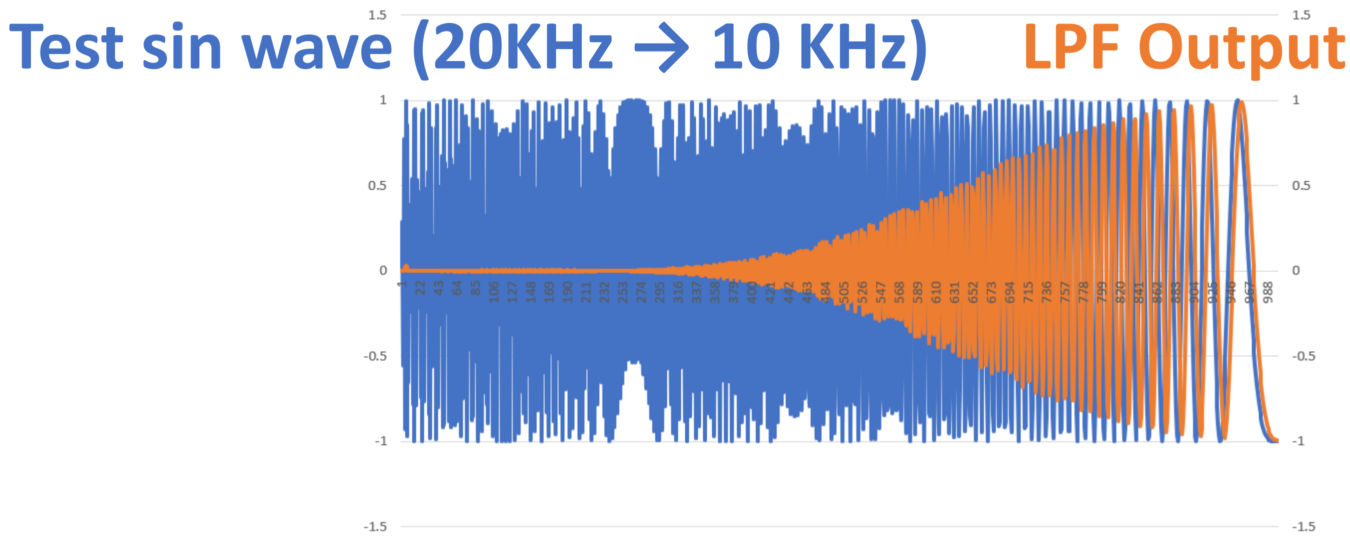
https://github.com/HirokiNakahara/FPGA_lecture/tree/master/Lec7_Practical_RTL_design/fir.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 11

void fir(float *y, float x)
{
    float c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
    -4.120289718403869e-03, -1.208600321298122e-02,
    -2.650603053411641e-03,  9.166631627169690e-02,
     2.544318483405623e-01,  3.400000000000001e-01,
     2.544318483405623e-01,  9.166631627169690e-02,
    -2.650603053411641e-03, -1.208600321298122e-02,
    -4.120289718403869e-03, };

    static float shift_reg[N];
    float acc;
    int i;

    acc = 0.0;
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

```c
void main()
{
    float fs = 44100.0;
    int len = 1000;

    float f0 = 20000.0;
    float sin_wave;
    float fir_out;

    int i;

    for( i = 0; i < len; i++){
        sin_wave = sin( 2.0 * M_PI * f0 * i / fs);

        fir( &fir_out, sin_wave);

        printf("%d %f %f\n", i, sin_wave, fir_out);
        f0 = f0 - 10.0;
    }
}
```

Debug for C Description

- Confirm the operation of FIR
- In/Out are reused as a testbench for HDL simulation
- Note, a parallel operation cannot be verified
- Area and speed of the circuit can not be estimated



Convert to Fixed Point Precision

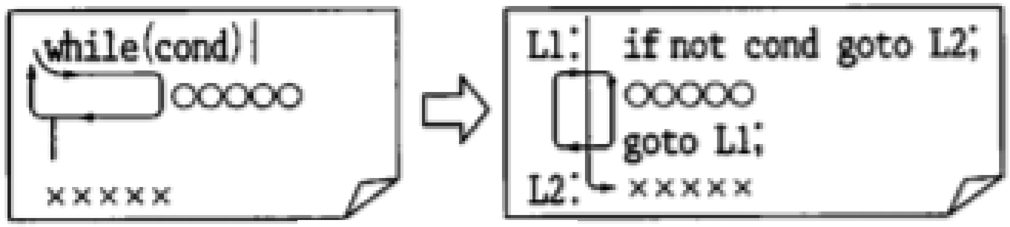
https://github.com/HirokiNakahara/FPGA_lecture/tree/master/Lec7_Practical_RTL_design/fir_int.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define N 11
#define PREC 65536 // 2**16 sign + 15bit precision

void fir(int *y, int x)
{
    int c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamming Window
        -136, -397, -87, 3004, 8338, 11142, 8338,
        3004, -87, -397, -136, };

    static int shift_reg[N];
    int acc;
    int i;

    acc = 0;
    for (i = N - 1; i >= 0; i--) {
        if (i == 0) {
            acc += x * c[0];
            shift_reg[0] = x;
        } else {
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
        }
    }
    *y = acc;
}
```

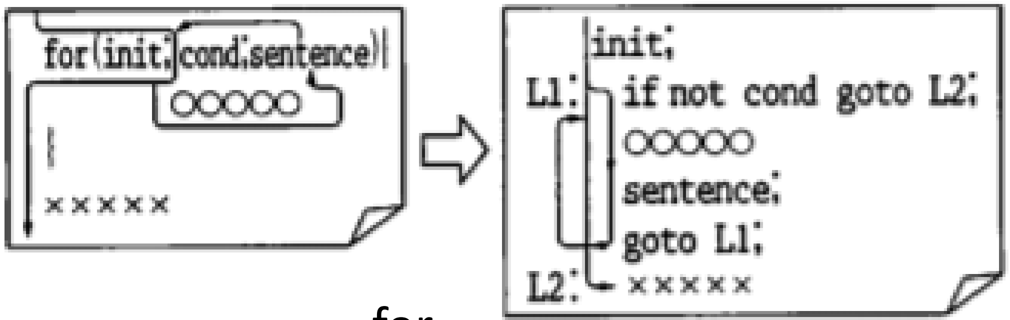
```c
void main()
{
    float fs = 44100.0;
    int len = 1000;

    float f0 = 20000.0;
    float sin_wave;
    int fir_out;

    int i;

    for( i = 0; i < len; i++){
        sin_wave = sin( 2.0 * M_PI * f0 * i / fs);

        fir( &fir_out, (int)(sin_wave * PREC));

        printf("%d %f %f\n", i, sin_wave, (float)fir_out / PREC);
        f0 = f0 - 10.0;
    }
}
```

C Behavior to RTL

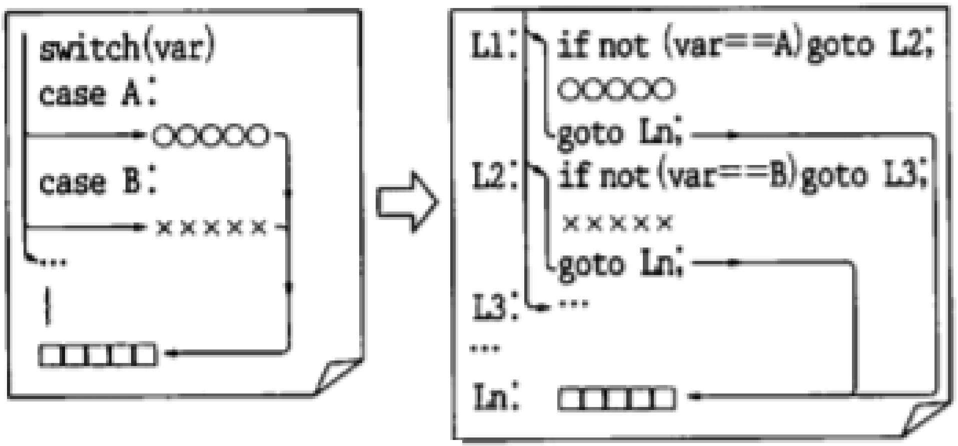
- RTL → Data path + FSM
- Re-write control *while, for, switch* statements to *if-then, goto* statements, then convert FSM
- Assign label to each statement → FSM state number



While



for



switch

Example

```
L1: acc = 0;
L2: for (i = N - 1; i >= 0; i--) {
L3: if (i == 0) {
L3_1:   acc += x * c[0];
L3_2:   shift_reg[0] = x;
    } else {
L3_3:   shift_reg[i] = shift_reg[i - 1];
L3_4:   acc += shift_reg[i] * c[i];
    }
}
L4: *y = acc;
```

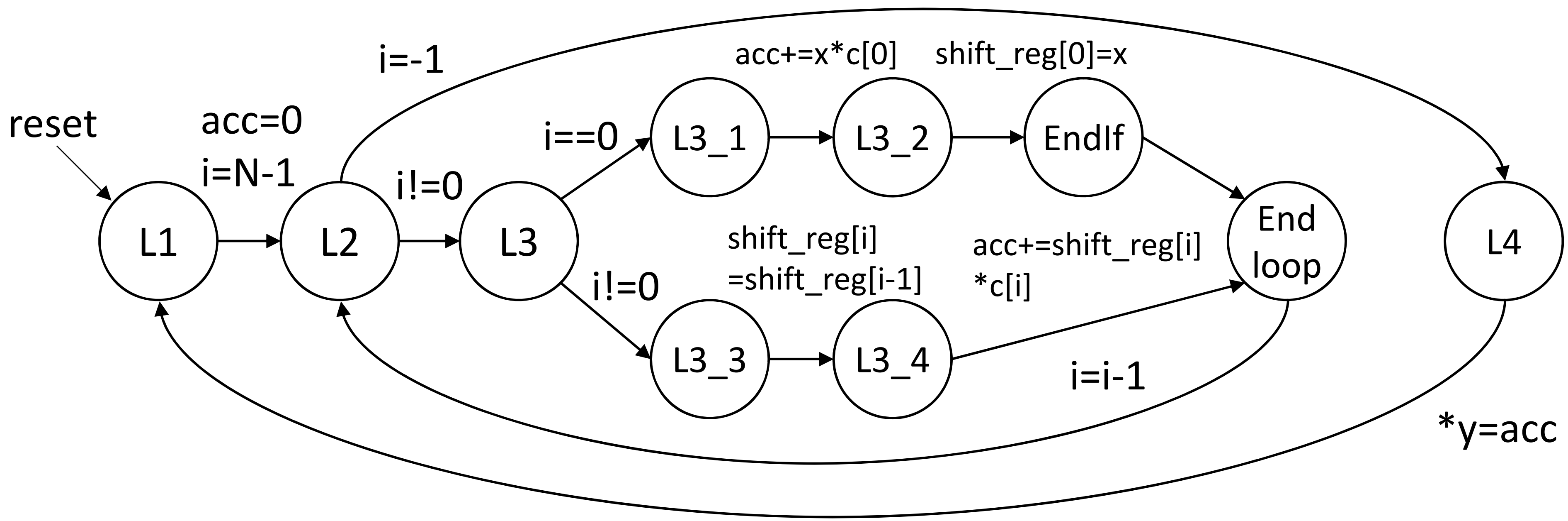
```
L1: acc = 0, i = N - 1;
L2: if i == -1 then goto L4:
L3:     if i != 0 then goto L3_3:
L3_1:       acc += x * c[0];
L3_2:       shift_reg[0] = x;
EndIf:  goto Endloop:
L3_3:       shift_reg[i] = shift_reg[i - 1];
L3_4:       acc += shift_reg[i] * c[i];
Endloop: i = i -1, goto L2:
L4: *y = acc;
```

Write FSM

- Convert *if-then goto* statement to FSM
    - Writing an FSM until you get used to it!
- Add an initialization processing (register value after resetting)
- Make the whole process an infinite loop
    - Generally, return to the initial state after finished all processing

Example

```
L1: acc = 0, i = N - 1;
L2: if i == -1 then goto L4:
L3:     if i != 0 then goto L3_3:
L3_1:       acc += x * c[0];
L3_2:       shift_reg[0] = x;
EndIf:  goto Endloop:
L3_3:       shift_reg[i] = shift_reg[i - 1];
L3_4:       acc += shift_reg[i] * c[i];
Endloop: i = i -1, goto L2:
L4: *y = acc;
```



Parallel Processing

- Concurrent assignment

```
tmp=A;        A<=B;
A = B;        B<=A;
B = tmp;
```

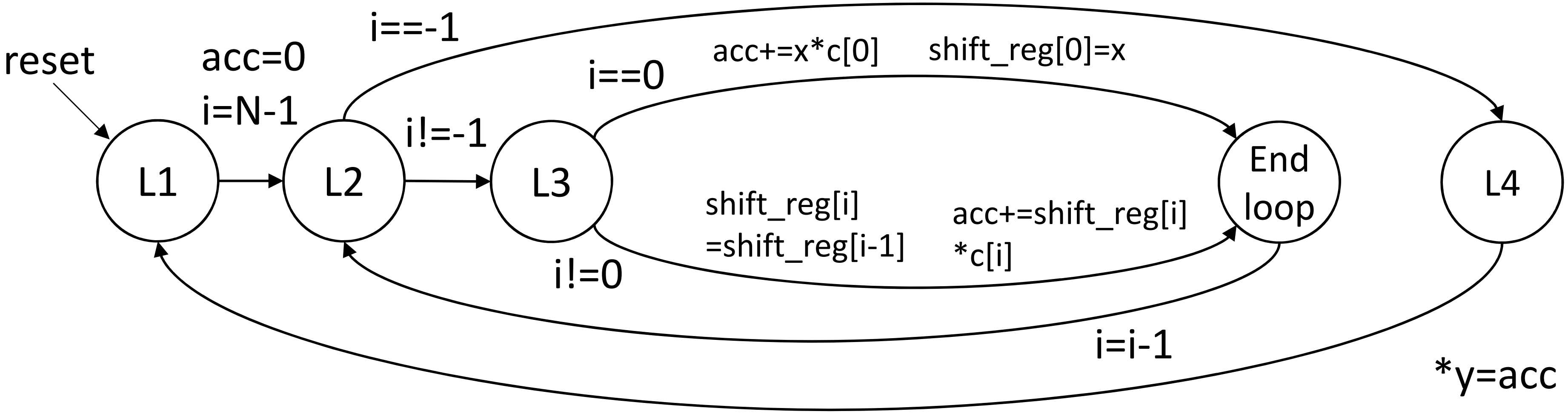
- Continuous assignments

  A=B; B=C; $\rightarrow$ A=C;

- Reduce number of states by parallel processing
- Considering simultaneous assignment from the starting FSM description

More Simplify

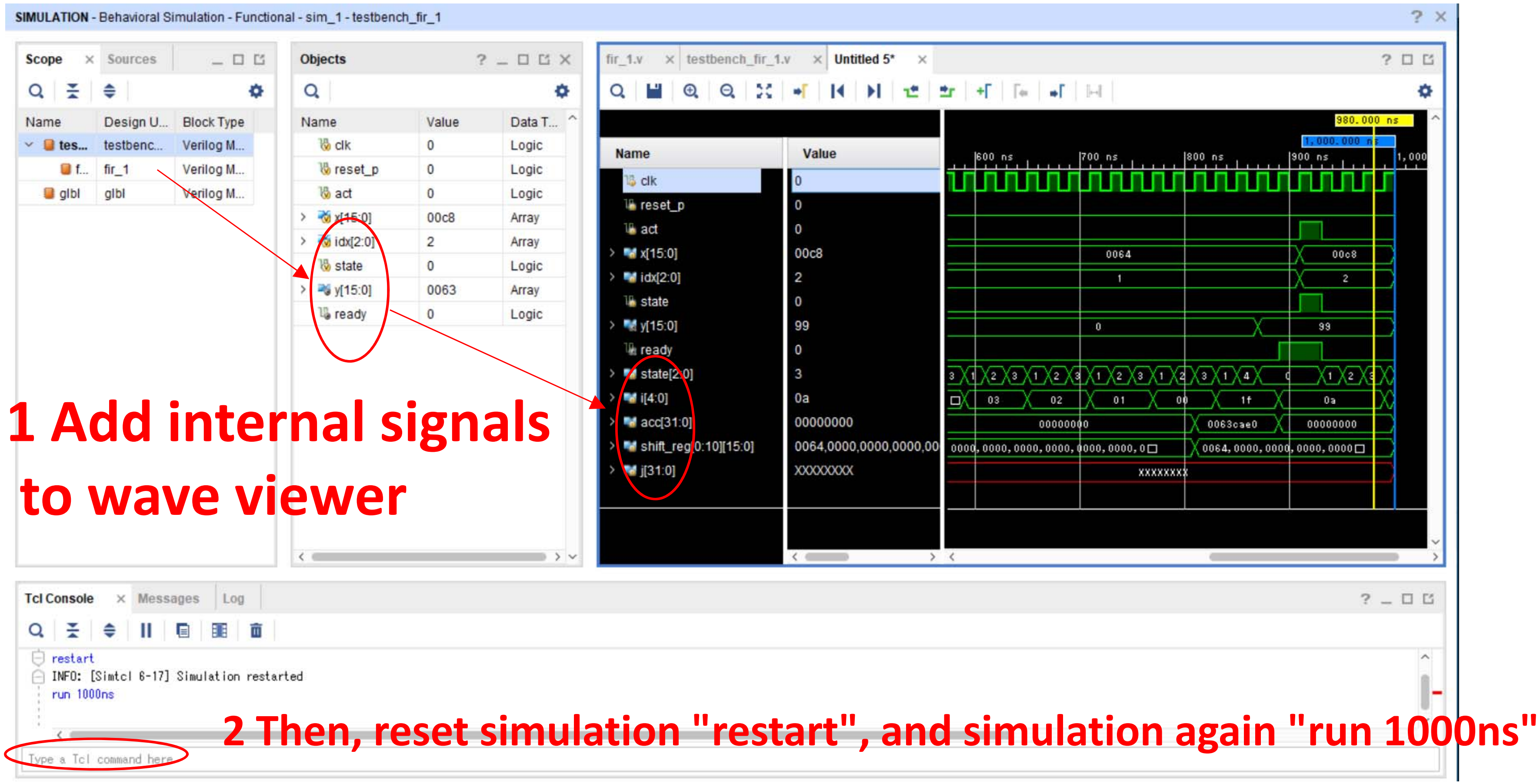
```
L1: acc = 0, i = N - 1;
L2: if i == -1 then goto L4:
L3:     if i == 0 then
            acc += x * c[0];
            shift_reg[0] = x;
            goto Endloop
        else
            shift_reg[i] = shift_reg[i - 1];
            acc += shift_reg[i] * c[i];
Endloop: i = i -1, goto L2:
L4: *y = acc;
```



RTL Simulation for an FIR Filter

See, https://github.com/HirokiNakahara/FPGA_lecture/tree/master/Lec7_Practical_RTL_design/

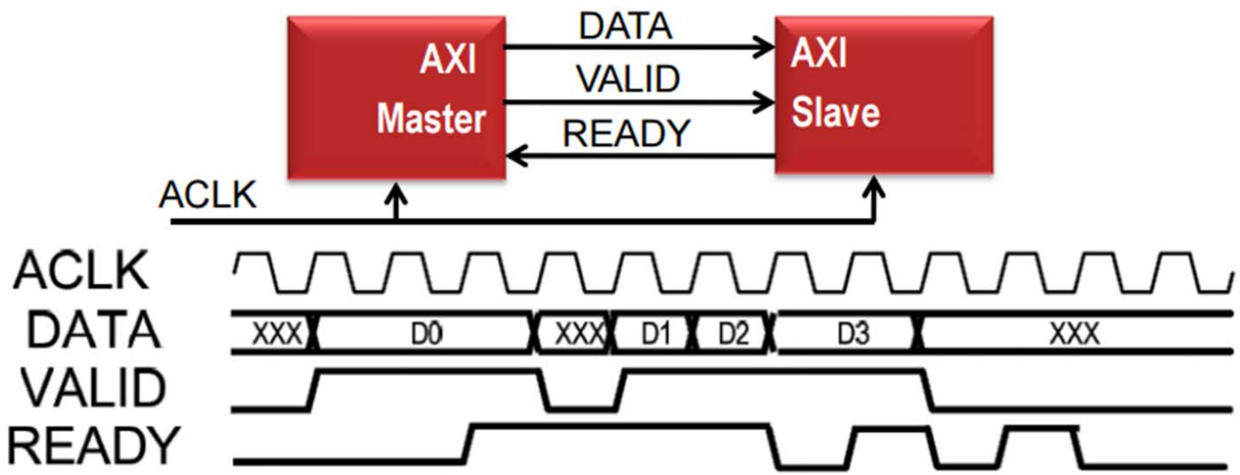
- FIR Filter module: fir_1.v
- Testbench for FIR Filter: testbench_fir_1.v

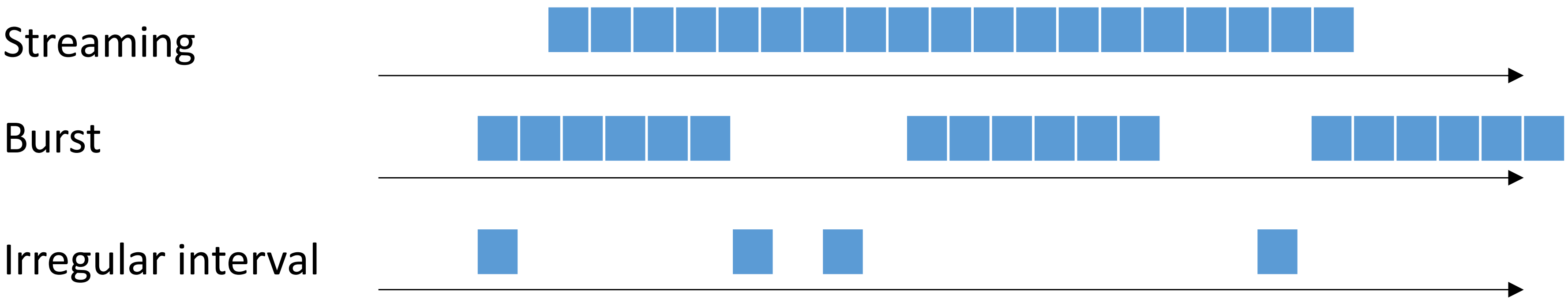


Interface Co-Design

Interface

- Data Transfer/Receive between modules





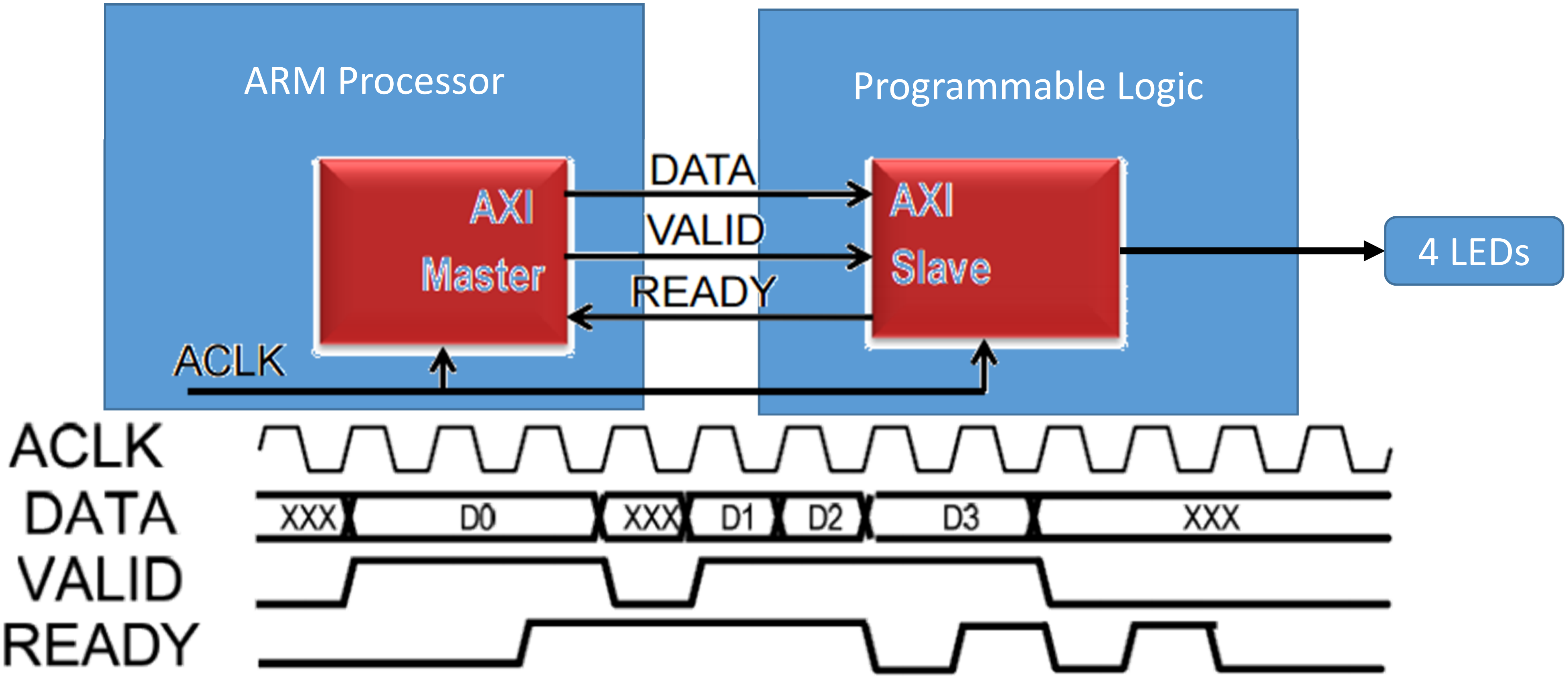
AXI 4 bus: General Interface of ARM Embedded FPGA

- Complex protocols
  - High-level synthesis (HLS) can be easily generated with Directive
  - System design tool (SDSoC) automatically selects the best protocol

	AXI4	AXI4-Lite	AXI4-Stream
Dedicated for	high-performance and memory mapped systems	register-style interfaces (area efficient implementation)	non-address based IP (PCIe, Filters, etc.)
Burst (data beta)	up to 256	1	Unlimited
Data width	32 to 1024 bits	32 or 64 bits	any number of bytes
Applications (examples)	Embedded, memory	Small footprint control logic	DSP, video, communication

Case Study: AXI4 Bus Connection

- Led blinking via AXI-lite bus



Create a New Project

Project location: C:¥FPGA¥lect7_2¥led_axi_lite_1

Target FPGA: Zybo-Z7-10 or (Z7-20)

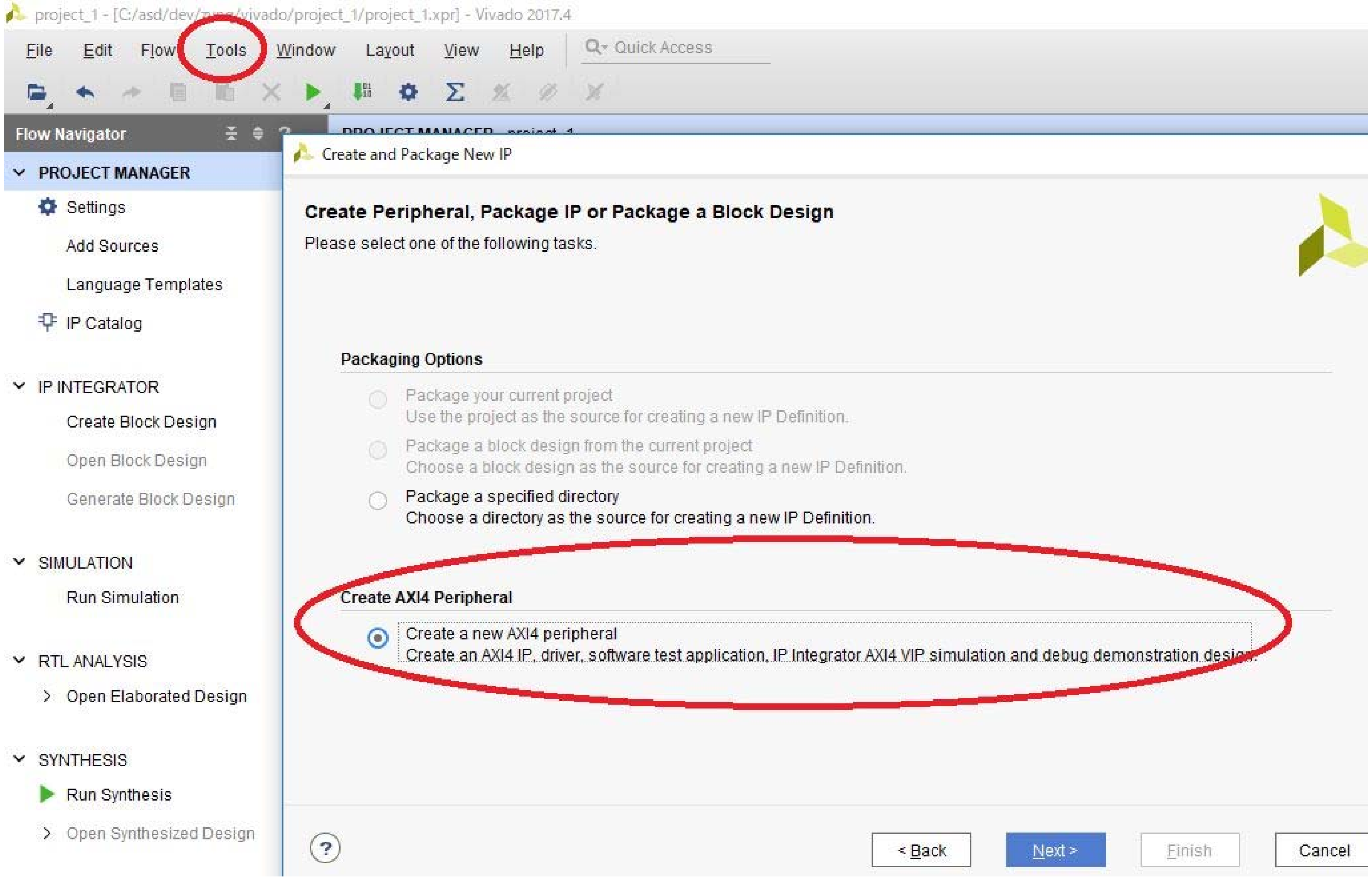
Design Sources: None

Constraints: Zybo-Z7-Master.xdc

Simulation Sources: None

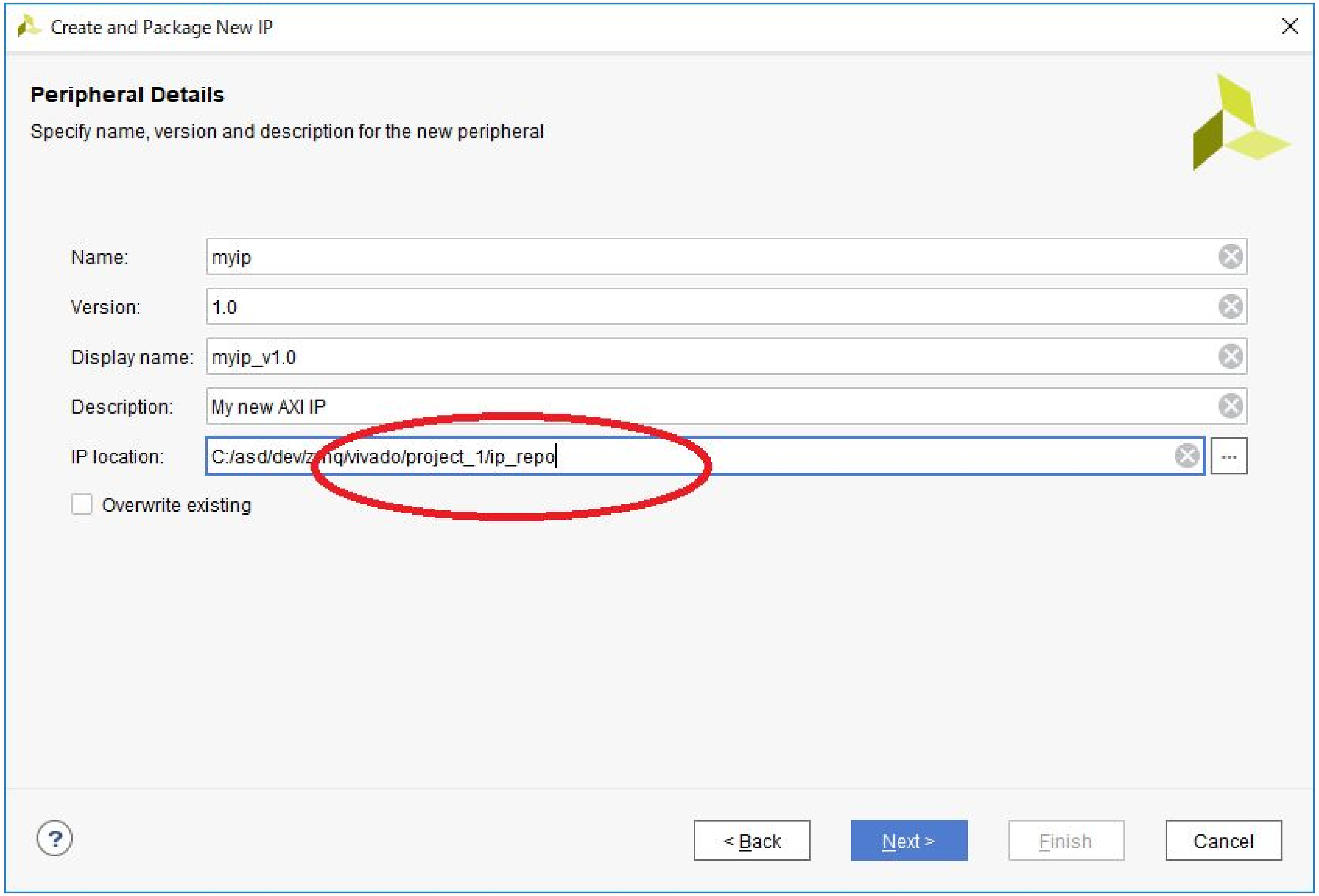
Create AXI4 Peripheral

- Select "Tools->Create and Package New IP", then check "Create AXI4 Peripheral", and "Next"



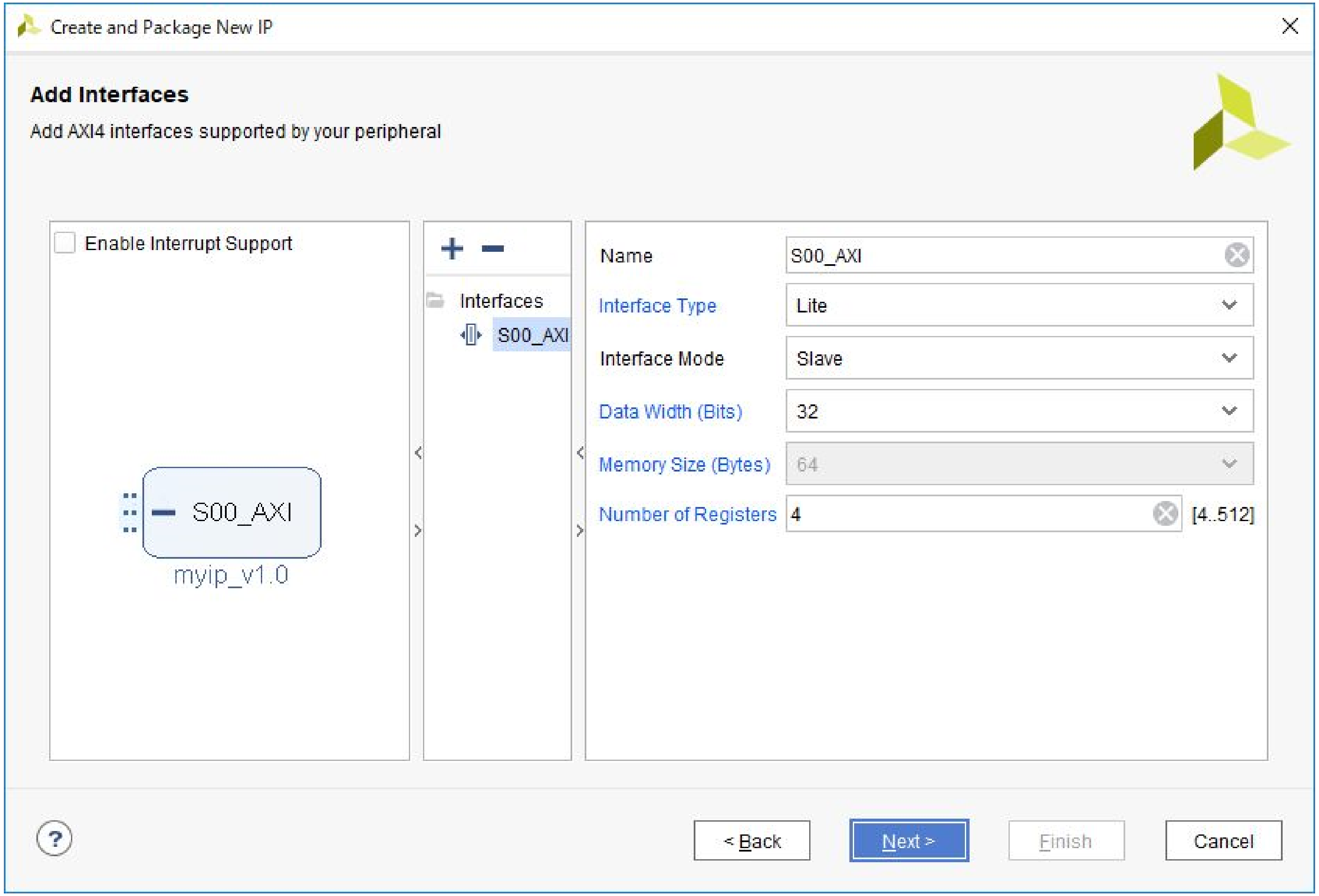
Specify IP Location

Type "ip_repo" on your project directory, then "Next"



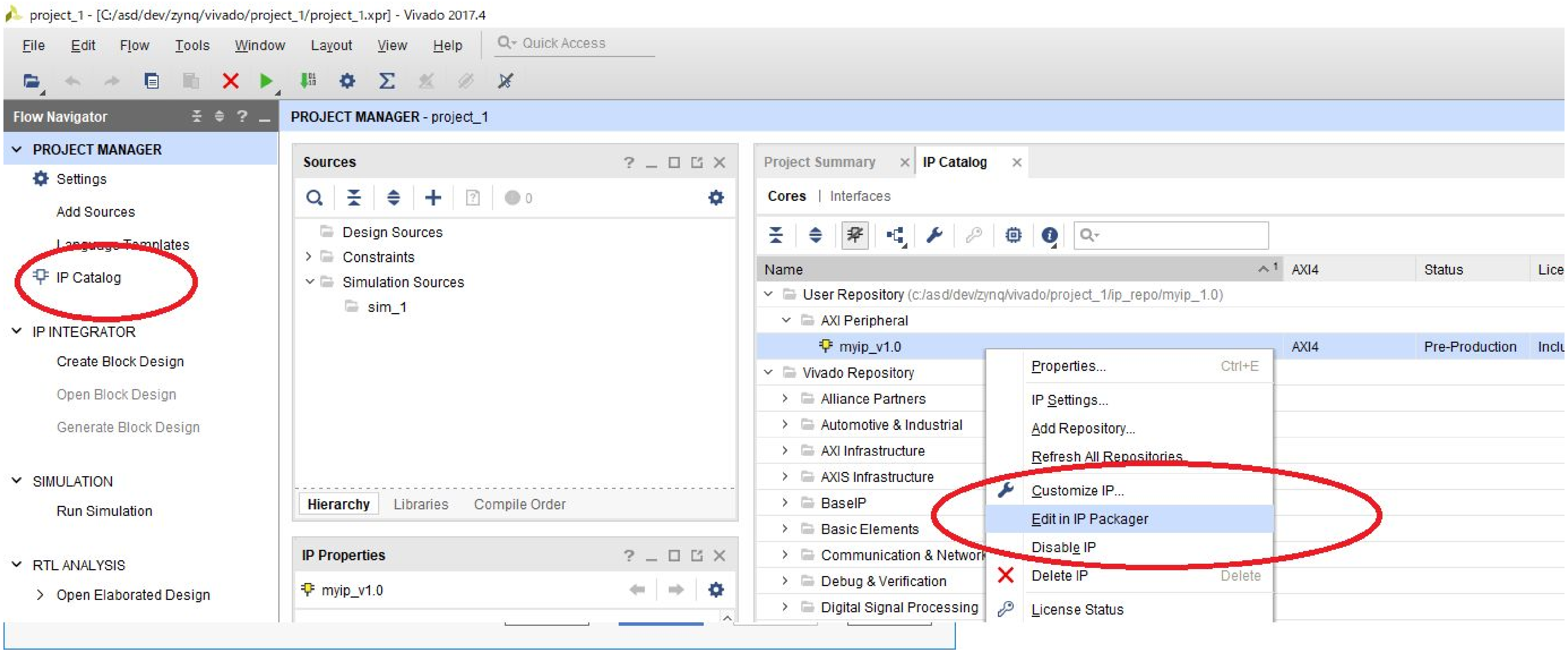
Edit Interface

- Set the default "AXI4-Lite Slave (four 32-bit registers)", and "OK", then "Finish"



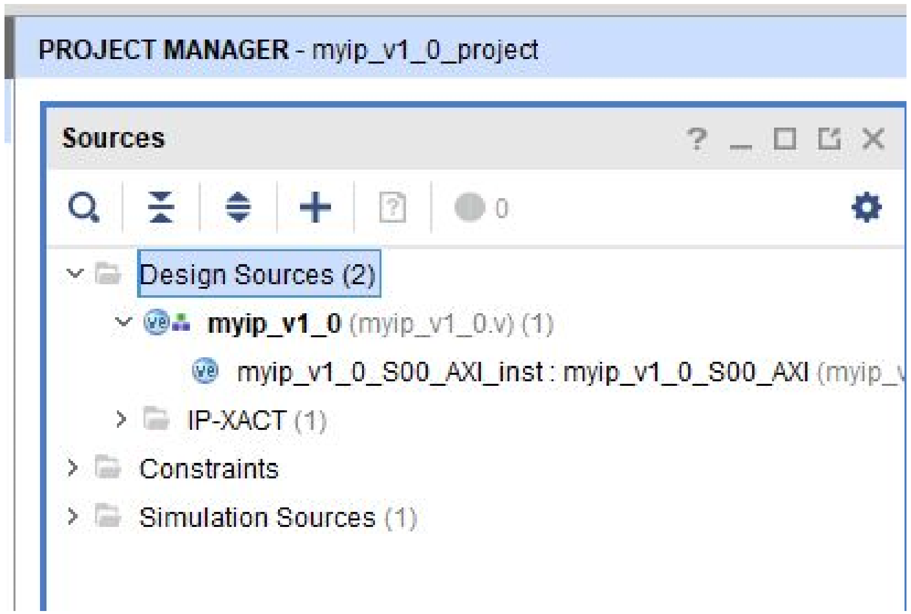
Edit "myip_v1.0"

- Click Flow Navigator->PROJECT MANAGER -> IP Catalog
- Make sure "myip_v1.0" under "User Repository" on "IP Catalog"
- Right click on "myip_v1.0", then select "Edit in IP Packager"
  - Click "OK" to save the project location

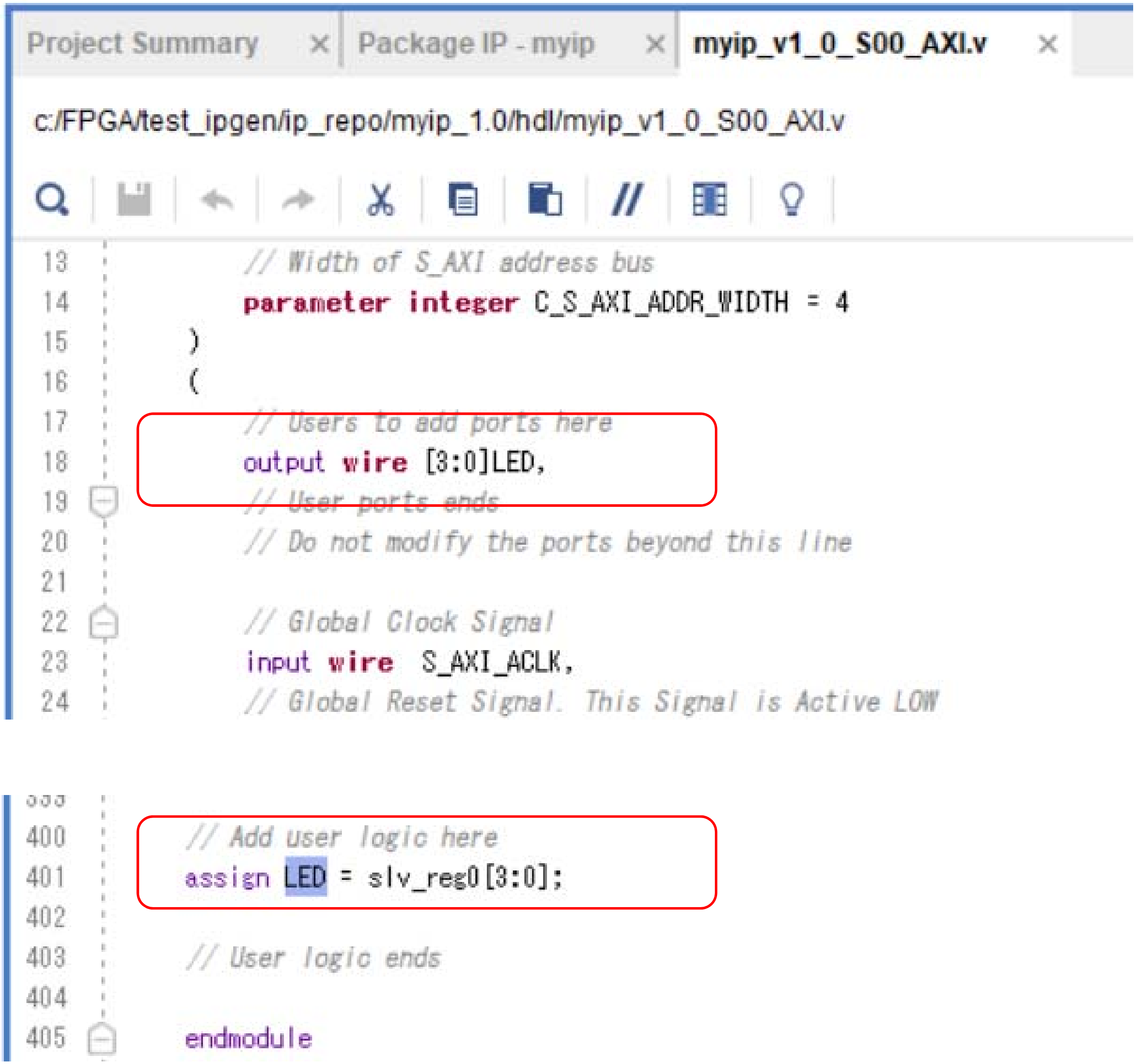


Synthesis "myip" on a New Vivado

- Make sure "myip_v1_0.v" as a wrapper and "myip_v1_0_S00_AXI.v" as a top module



Edit "myip_v1_0_S00_AXI.v"



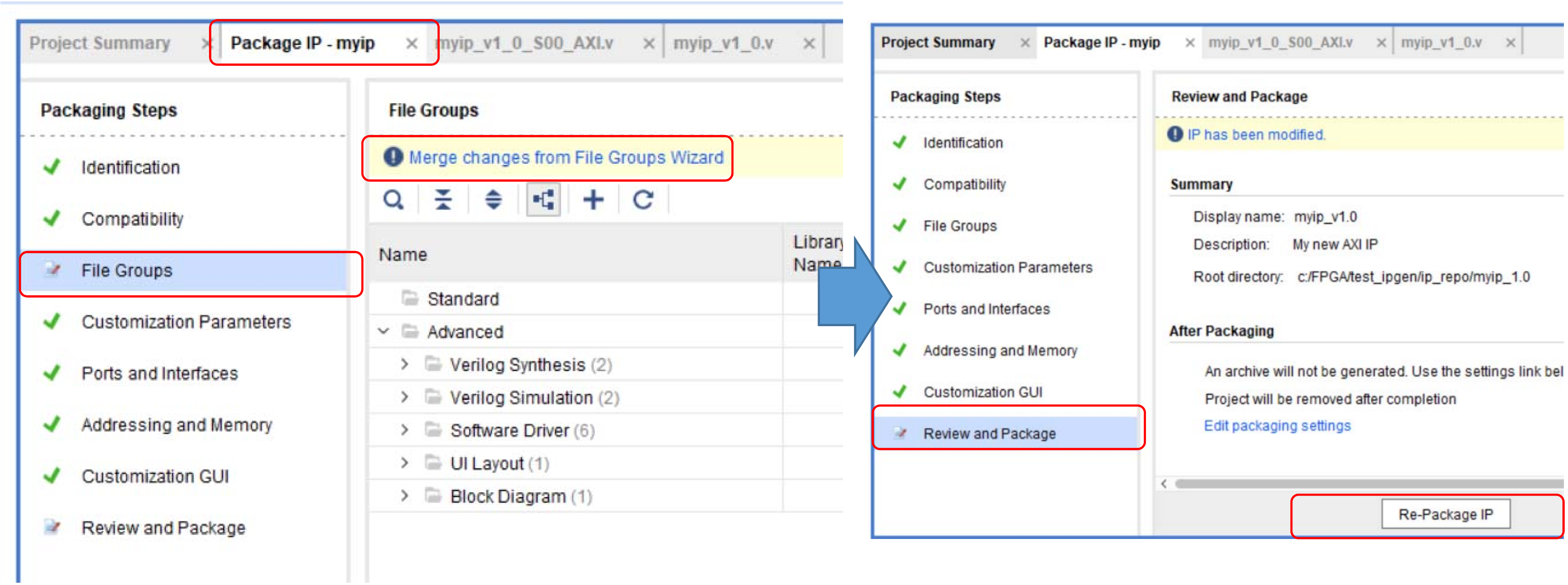
Edit "myip_v1_0.v"

```
        // Parameters of Axi Slave Bus Interface S00_AXI
        parameter integer C_S00_AXI_DATA_WIDTH   = 32,
        parameter integer C_S00_AXI_ADDR_WIDTH   = 4
    )
    (
        // Users to add ports here
        output wire [3:0]LED,
        // User ports ends
        // Do not modify the ports beyond this line

```

```
        input wire  s00_axi_rready
    );
// Instantiation of Axi Bus Interface S00_AXI
    myip_v1_0_S00_AXI # (
        .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
        .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
    ) myip_v1_0_S00_AXI_inst (
        .LED( LED),
        .S_AXI_ACLK(s00_axi_aclk),
        .S_AXI_ARESETN(s00_axi_aresetn),
        .S_AXI_AWADDR(s00_axi_awaddr)
```

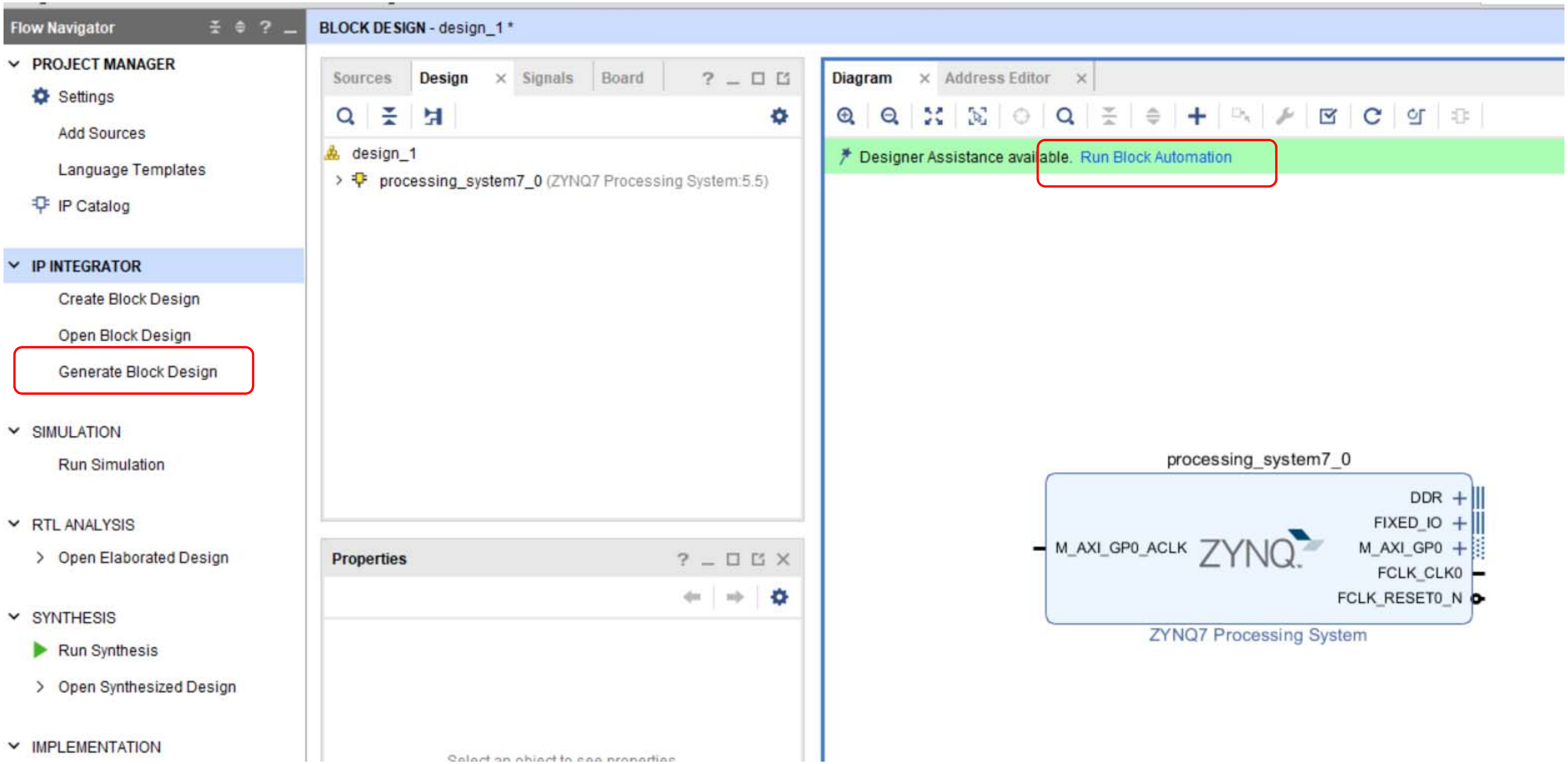
Re-Package IP

- Switch to "Package IP" tab, then "Re-Package IP"



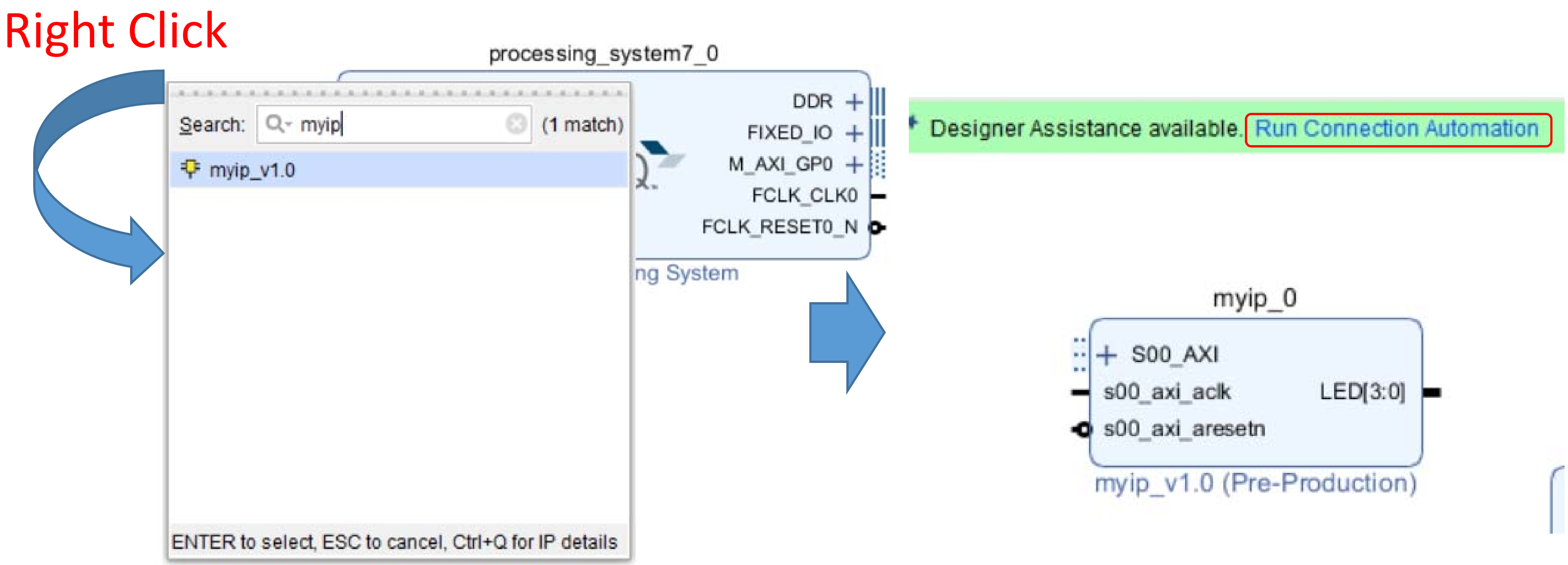
Add a ZYNQ Processor

- In the initial Vivado, Flow Navigator -> IP INTEGRATOR -> Create Block Design, then add a ZYNQ Processor, and "Run Block Automation"

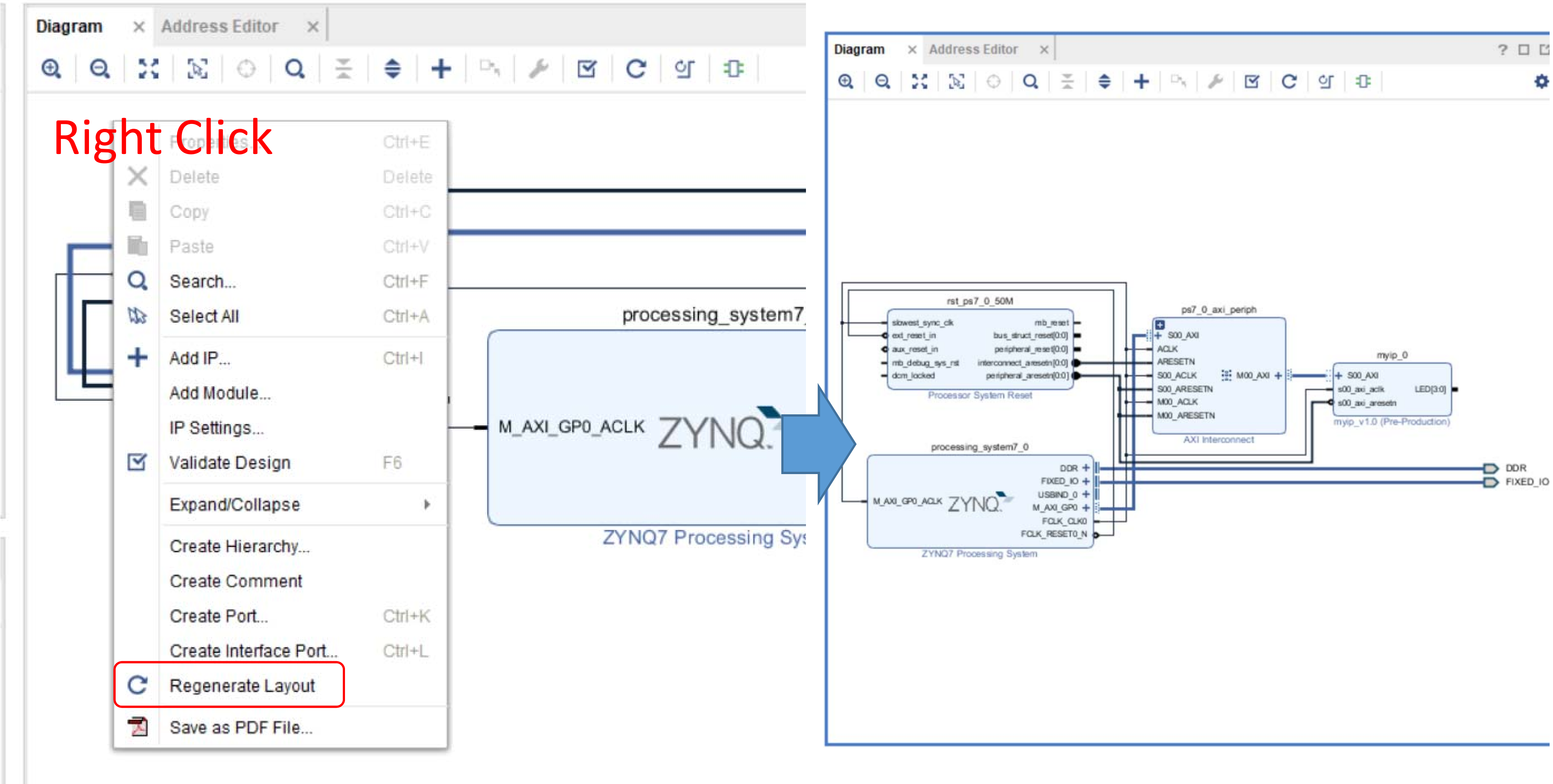


Add a "myip" IP

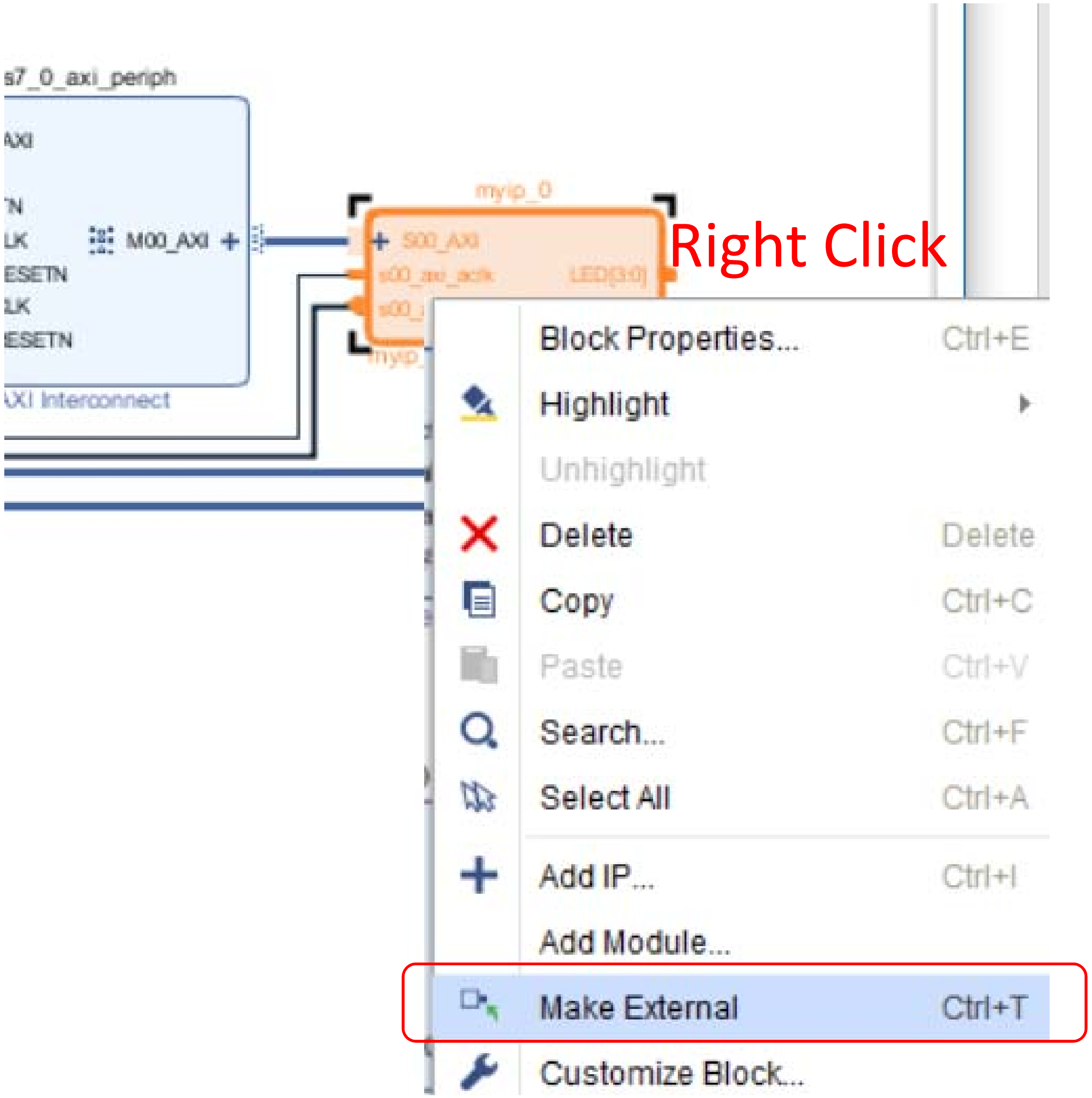
- Place your "myip" on the Block Design View, then click "Run Connection Automation", and "OK"



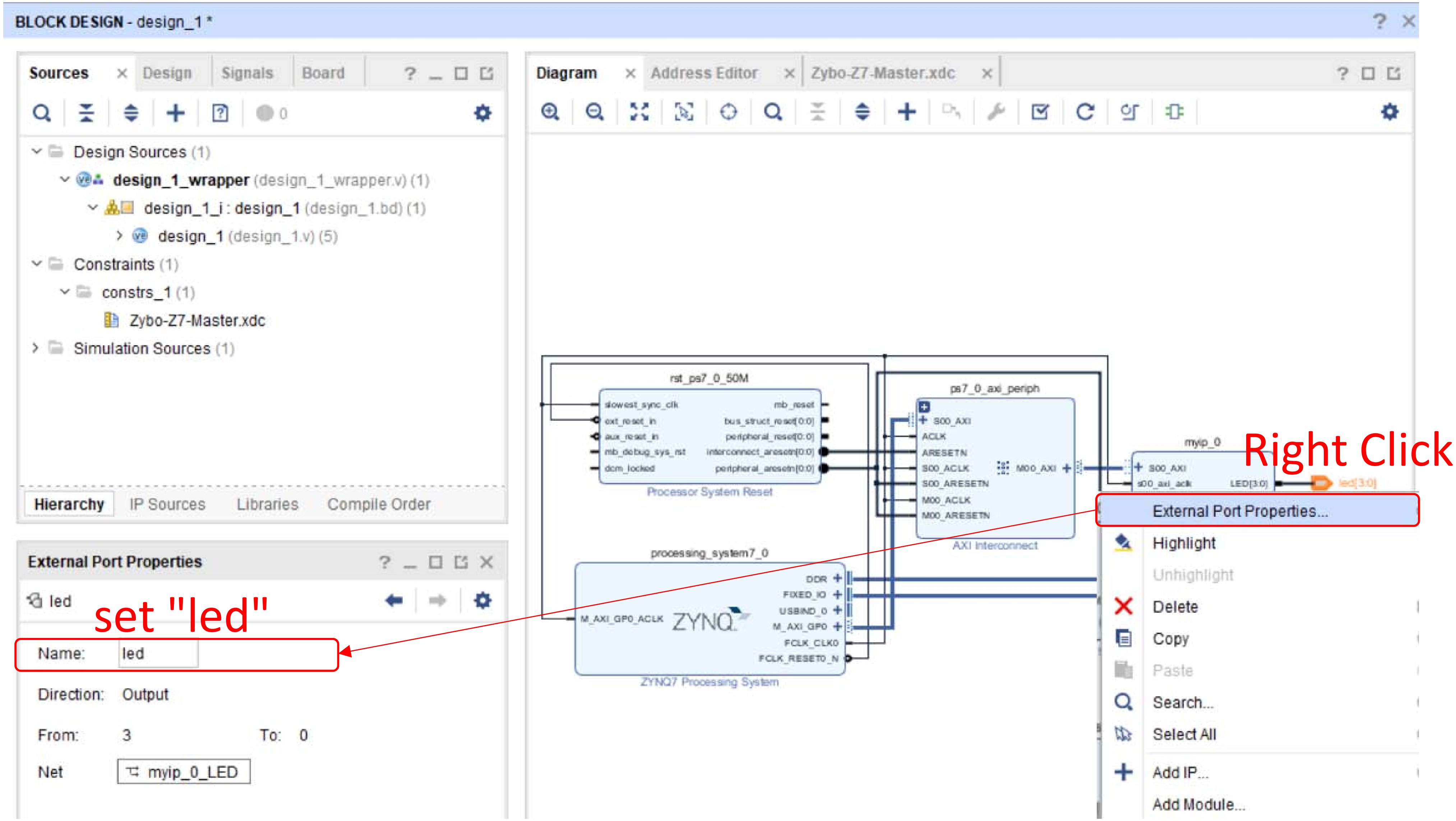
Regenerate Layout



Make External



Specify an External Port Name



Write Software Code to Control "myip" from a ZYNQ Processor

- Click "Generate Bitstream", then "Export Hardware", and next, "Launch SDK"
- Create a new project as "myip_test"

Source Code

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

#include "xparameters.h"

#define LED *((volatile unsigned int *) XPAR_MYIP_0_S00_AXI_BASEADDR)

int main()
{
    init_platform();

    print("Hello World\n\r");

    int i, j;

    while(1){
        for( i = 0; i < 6; i++){
            xil_printf("i=%d\n", i);
            switch(i){
            case 0: LED = 0x1; break;
            case 1: LED = 0x2; break;
            case 2: LED = 0x3; break;
            case 3: LED = 0x4; break;
            case 4: LED = 0x5; break;
            case 5: LED = 0x6; break;
            default: LED = 0x0;
            }
            for( j = 0; j < 10000000; j++);
        }
    }

    cleanup_platform();
    return 0;
}
```

Memory map is automatically generated by Vivado, and it is written in "xparameters.h"

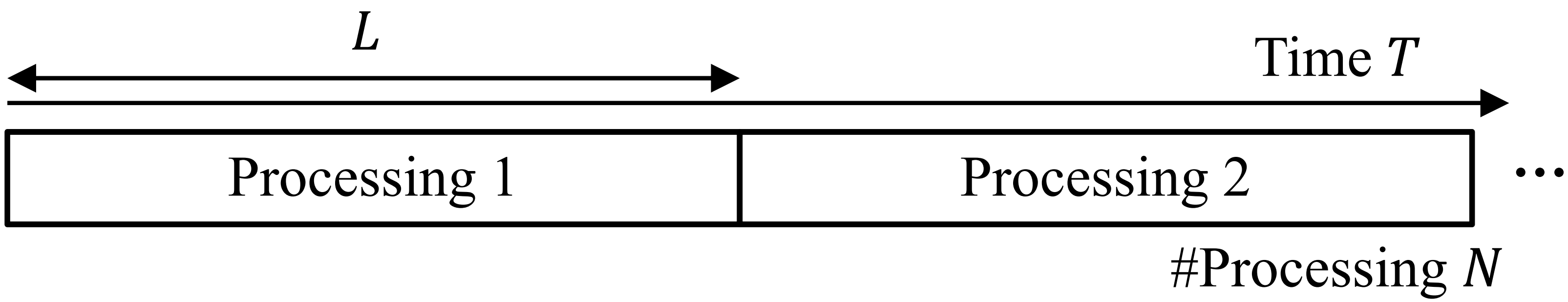
Build the project, then "Xilinx->Program FPGA".
Next, Connect the Zybo to the PC
Run Terminal software (e.g. Tera Term for Windows, gtkterm for Unix)
Connect "USB Serial Port" with 115200 bps
Select the project in the Project Explorer, then, in "Menu", "Run As" -> "Launch on Hardware (System Debugger)"

RTL Design Optimization

Pipelining

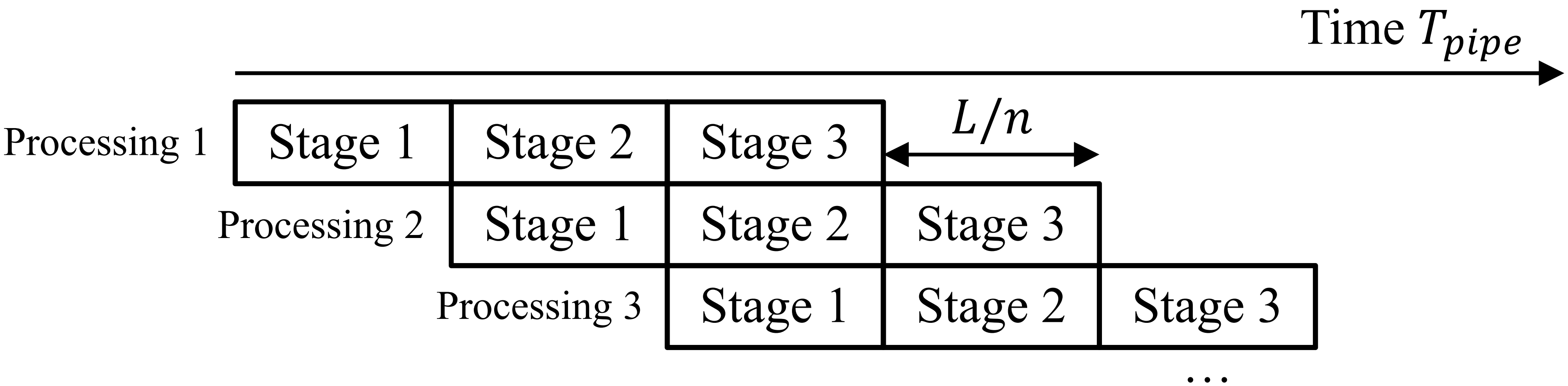
(a) Non-pipelining

Processing iteration 2 is done sequentially **after the completion of iteration 1**



(b) Pipelining (n = 3 stage)

Processing iteration 2 is done **after the completion of stage 1 in iteration 1**



Pipeline Efficiency

Percentage of the actually achieved speedup to the maximum

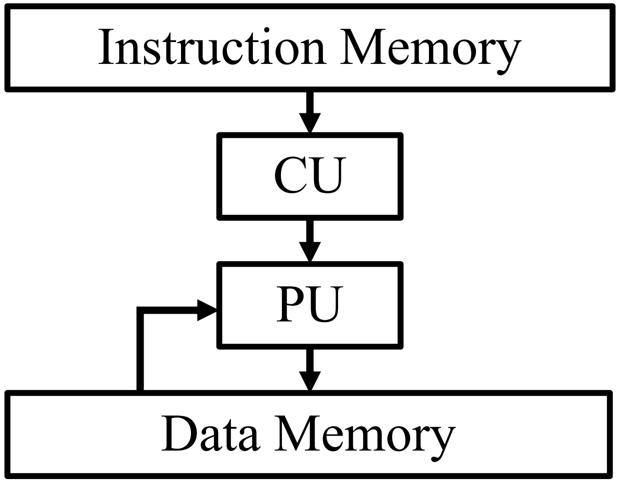
$$S_{pipe}(N) = \frac{T(N)}{T_{pipe}(N)} = \frac{nN}{n+N-1} = \frac{n}{1+\frac{n-1}{N}}$$

If $n << \mathrm{N}$, then $S_{pipe}(N) \cong n$ and the speed-up factor over non-pipelining is n

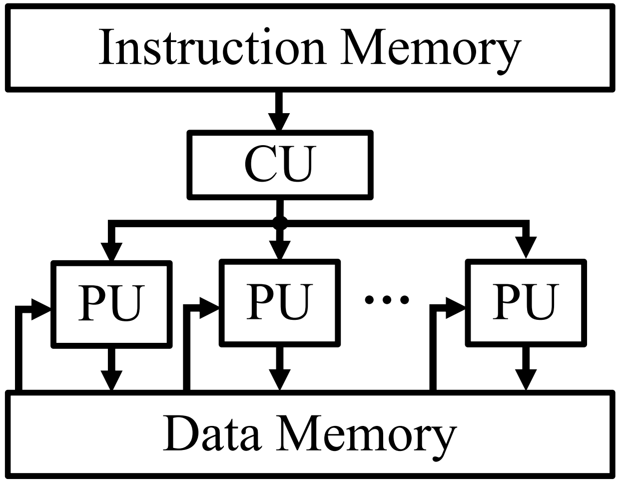
Percentage of the actually achieved speedup to the maximum

$$E_{pipe}(n,N) = \frac{S_{pipe}(N)}{n} = \frac{1}{1+\frac{n-1}{N}} = \frac{N}{N+n-1}$$

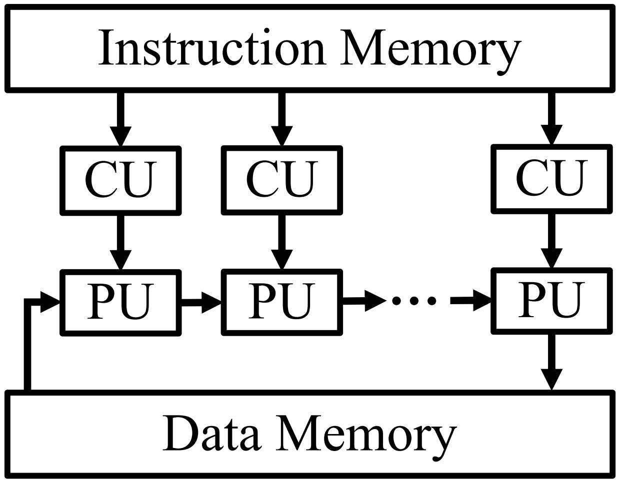
Parallel Processing and Flynn's Taxonomy



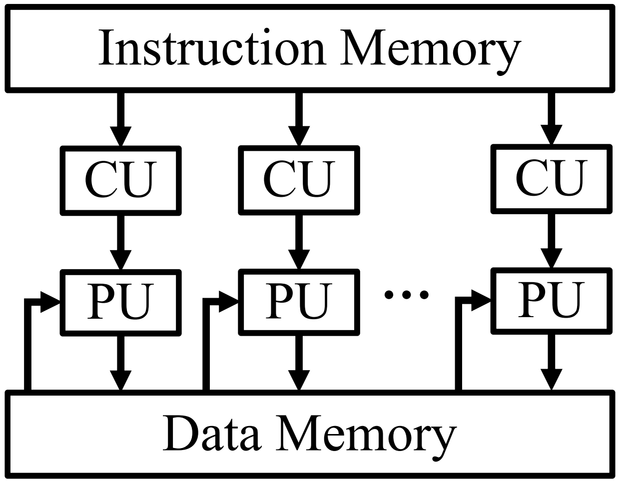
(a) SISD



(b) SIMD



(c) MISD

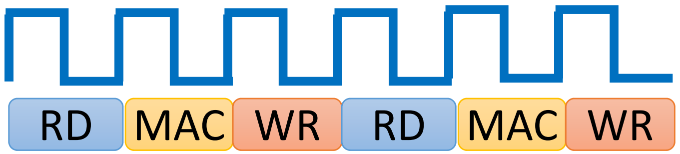


(d) MIMD

Loop Unrolling

- Without unrolling

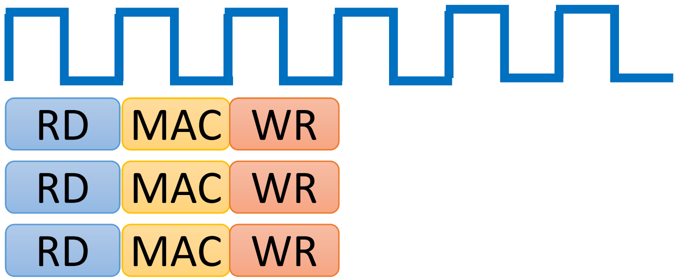
```
for ( int i = 0; i < N; i++){
  op_Read[i];
  op_MAC;
  op_Write[i];
}
```



Throughput: 3 cycles
Latency: 3 cycles
Operation: 1/3 data/cycle

- Loop Unrolling for 3 Operations

```
for ( int i = 0; i < N/3; i+=3){
  op_Read[i*3];
  op_MAC;
  op_Write[i*3];
  op_Read[i*3+1];
  op_MAC;
  op_Write[i*3+1];
  op_Read[i*3+2];
  op_MAC;
  op_Write[i*3+2];
}
```



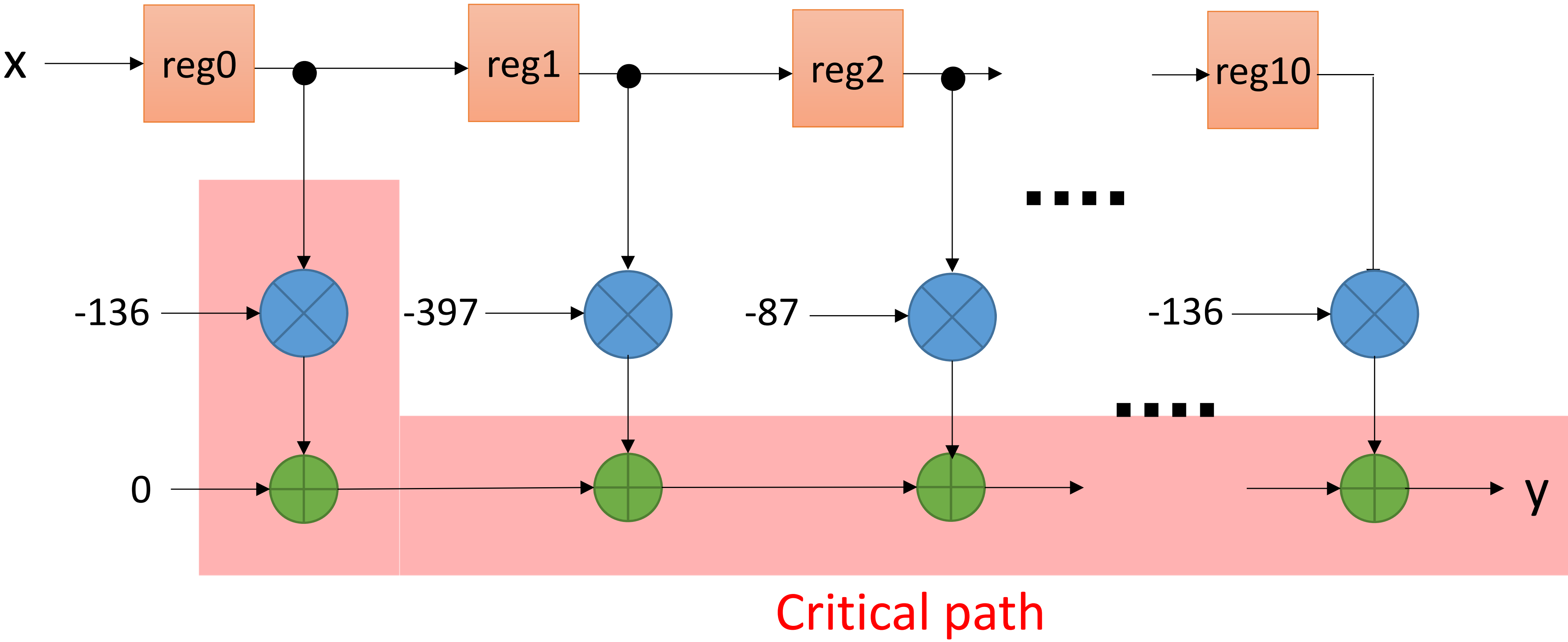
Throughput: 3 cycle
Latency: 3 cycle
Operation: 1 data/cycle

Unrolling for a FIR Filter

```c
int c[N] = { // 0.17 = 20KHz/44.1KHz, LPF, Hamm
    -136, -397, -87, 3004, 8338, 11142, 8338,
    3004, -87, -397, -136, };

static int shift_reg[N];
int acc;
int i;

acc = 0;
for (i = N - 1; i >= 0; i--) {
    if (i == 0) {
        acc += x * c[0];
        shift_reg[0] = x;
    } else {
        shift_reg[i] = shift_reg[i - 1];
        acc += shift_reg[i] * c[i];
    }
}
*y = acc;
```

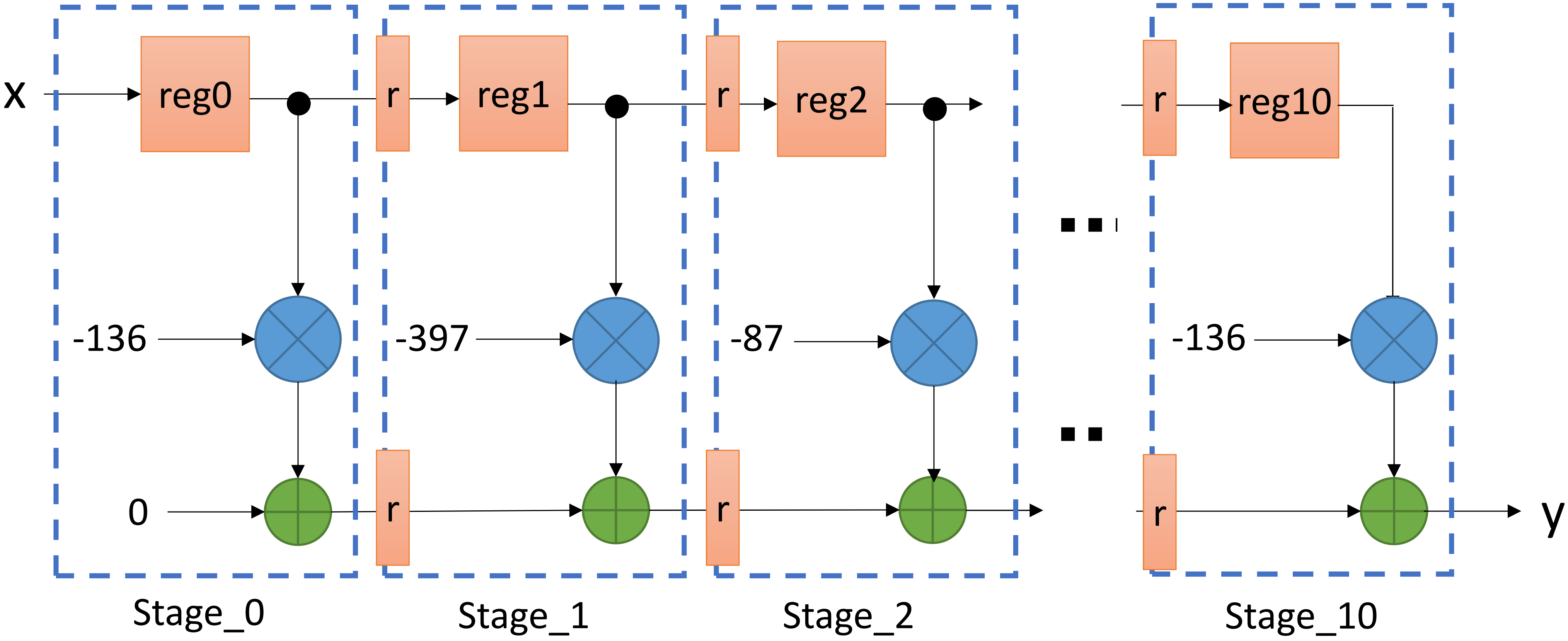
```c
static int shift_reg[N];

shift_reg[10] = shift_reg[9];
shift_reg[ 9] = shift_reg[8];
shift_reg[ 8] = shift_reg[7];
shift_reg[ 7] = shift_reg[6];
shift_reg[ 6] = shift_reg[5];
shift_reg[ 5] = shift_reg[4];
shift_reg[ 4] = shift_reg[3];
shift_reg[ 3] = shift_reg[2];
shift_reg[ 2] = shift_reg[1];
shift_reg[ 1] = shift_reg[0];
shift_reg[ 0] = x;

*y = shift_reg[10] * -136 + shift_reg[9] *  -397
   + shift_reg[ 8] *  -87 + shift_reg[7] *  3004
   + shift_reg[ 6] * 8338 + shift_reg[5] * 11142
   + shift_reg[ 4] * 8338 + shift_reg[3] *  3004
   + shift_reg[ 2] *  -87 + shift_reg[1] *  -397
   + shift_reg[ 0] * -136;
```

Dataflow for Unrolling FIR Filter



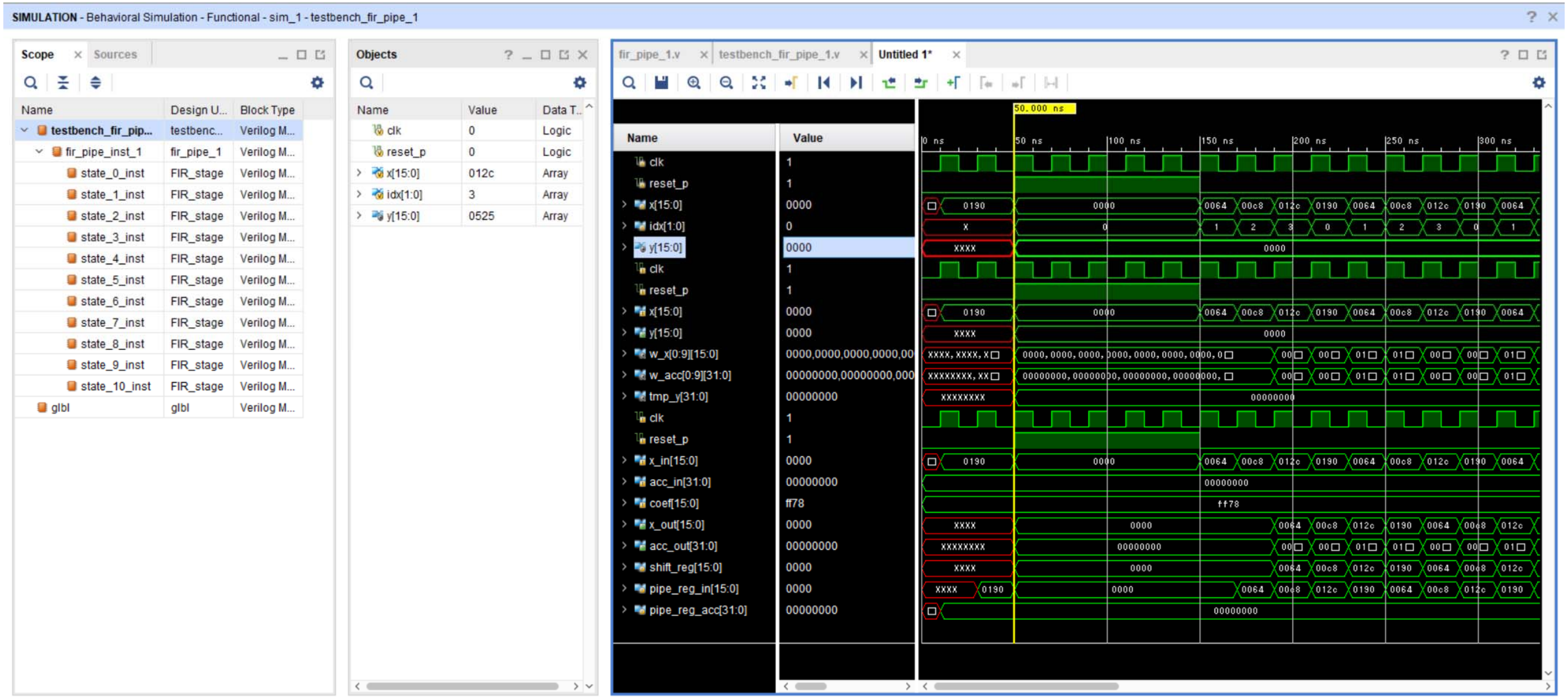
Pipelined Dataflow

- Insert a pipeline register and realized by a DSP block



RTL Simulation

- See, https://github.com/HirokiNakahara/FPGA_lecture/tree/master/Lec7_Practical_RTL_design/
- Source Code: fir_pipe_1.v, Simulation Code: testbench_fir_pipe_1.v



Conclusion

- Conversion from Behavior to RTL by C Description
- Control HW via AXI 4 bus
- Optimization method
  - Concurrent assignment
  - Parallel Processing
  - Unrolling
  - Pipelining

Exercise

- (Mandatory) Control the LED from the ARM processor via the AXI4 bus
- (Mandatory) For the FIR filter, discuss the Pros. and Cons. of pipeline version, unrolling version, sequential version by comparing latency, throughput, and multiplier
- (Optional 1) Reduce the number of multipliers by using a symmetry property for coefficients of an FIR filter
- (Optional 2) Design the RTL for above FIR filter and show the simulation result

Send a report to nakahara@ict.e.titech.ac.jp

Deadline is 17^{th}, July, 2018 (At the beginning of the next lecture)