Perancangan *SoC* Sederhana

Dismas Widyanto
Teknik Elektro
Institut Teknologi Bandung
Bandung, Indonesia
13218065@std.stei.itb.ac.id

***Abstract*—Dalam dokumen ini dituliskan proses desain dalam perancangan suatu *system on chip* sederhana. Sistem tersebut kemudian diimplementasikan dalam FPGA ZYNQ-7000. Implementasi sistem dievaluasi dengan melihat hasil serial monitor dan dari jumlah *resource* yang digunakan. Sistem yang dibuat berupa perkalian sederhana dengan banyak data sebesar 100 buah data dan pengali sebesar 10. Jumlah *resource* yang digunakan yaitu *Slice LUT* sebesar 13,61%, *LUT as Logic* sebesar 11,02%, *LUT as Memory* sebesar 7,62%, BRAM sebesar 6,67%, dan DSP sebesar 2,5%.**

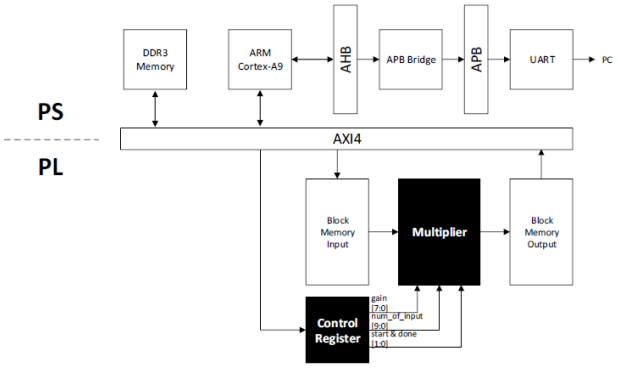
Keywords—FPGA, System on Chip, Verilog.

I. PENDAHULUAN

Melalui dokumen ini dilakukan percobaan mengenai pembuatan *System on Chip* sederhana. Sistem tersebut dibuat sebagai bahan pembelajaran untuk nantinya dilakukan pembuatan *System on Chip* dengan aplikasi yang lebih spesifik.

Sistem yang dibuat terdiri dari blok *multiplier* dan RAM sederhana, serta blok kendali untuk aplikasi yang digunakan. Sistem akan menerima masukan dari aplikasi kemudian disimpan pada RAM *input*. Selanjutnya sistem akan melakukan perhitungan berupa perkalian dari data pada RAM *input* dengan suatu *gain*. Hasil perkalian kemudian disimpan pada RAM *output* dan ditampilkan pada serial monitor.

Sistem ini secara garis besar terdiri dari dua buah bagian yaitu PL *(programmable logic)* dan PS (*processing system)*. Bagian PL merupakan perangkat keras yang diatur untuk melakukan perhitungan sesuai kebutuhan dalam hal ini adalah perkalian. Sedangkan bagian PS merupakan bagian perangkat lunak yang berguna untuk memberikan masukan dan menampilkan keluaran dari sistem. Diagram blok dari sistem dapat dilihat pada gambar di bawah. Sistem ini kemudian diimplementasikan dengan menggunakan FPGA ZYBO.



Gambar 1 Blok Diagram Sistem

II. PROSES DESAIN

Proses desain dari sistem ini dilakukan dalam enam tahap yaitu mempelajari Verilog, pembuatan blok RAM, pembuatan IP *multiplier*, pembuatan IP kontrol, pembuatan perangkat lunak, dan integrasi sistem.

A. *Mempelajari Verilog*

Proses belajar dilakukan dengan membuat tutorial yang diberikan. Tutorial tersebut terdiri dari modul Bahasa Verilog, modul rangkaian kombinasional, dan modul rangkaian sekuensial. Pada modul Bahasa Verilog, dijelaskan mengenai *syntax* yang digunakan dan struktur dari Bahasa Verilog. Struktur Bahasa Verilog terdiri dari nama blok dan pin masukan serta keluaran kemudian dilanjutkan dengan perilaku dari blok tersebut. Sebagai contoh adalah rangkaian *half_adder* di bawah ini, nama bloknya ada *half_adder* dengan pin masukan yaitu a dan b serta pin keluaran yaitu sum dan carry. Kemudian pada bagian bawah adalah perilaku dari blok yaitu sum merupakan operasi XOR antara a dan b serta carry merupakan operasi AND antara a dan b.

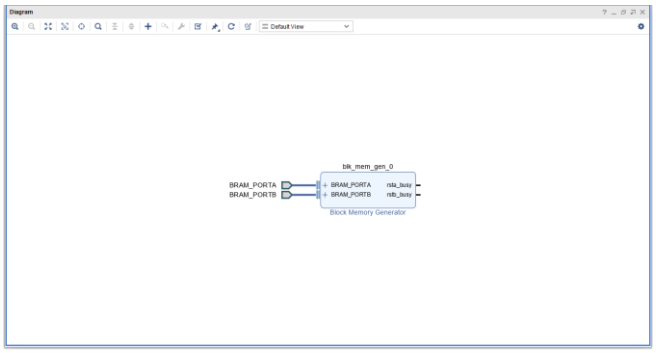
```
`timescale 1ns / 1ps

module half_adder
    (
        input wire  a,
        input wire  b,
        output wire sum,
        output wire carry
    );

    assign sum = a ^ b;
    assign carry = a & b;

endmodule
```

Pada modul rangkaian kombinasional dan rangkaian sekuensial dijelaskan mengenai rangkaian tersebut dan cara membuatnya dalam Bahasa Verilog. Rangkaian kombinasional adalah rangkaian yang keluarannya hanya bergantung dari masukan saat ini saja, sedangkan rangkaian sekuensial adalah rangkaian yang keluarannya juga bergantung dari masukan pada waktu sebelumnya. Umumnya, rangkaian sekuensial akan menggunakan *clock*.

B. *Pembuatan Blok RAM*

Pembuatan blok RAM memanfaatkan blok *Memory Generator* yang sudah tersedia pada VIVADO. Blok tersebut perlu diatur menjadi *True Dual Port RAM*. Kemudian *port* dari BRAM diberi antarmuka untuk jalur masukan dan keluaran. Blok BRAM yang dibuat dapat dilihat pada gambar

di bawah ini. Sesuai dengan pengaturan sebelumnya blok tersebut memiliki dua buah *port* [1].



Gambar 2 Blok BRAM

C. *Pembuatan IP Multiplier*

Blok *multiplier* merupakan blok untuk melakukan perkalian antara data dari BRAM dengan suatu *gain*. Kode Verilog dari blok ini dapat dilihat pada lampiran. Blok *multiplier* terdiri dari bagian *clock* dan *reset*, BRAM *input*, BRAM *output*, dan sinyal kontrol. Bagian BRAM terdiri dari alamat dan data serta *enable* untuk BRAM *output*. Bagian sinyal kontrol terdiri dari banyaknya data, nilai *gain*, dan tanda *start* serta *done* [1].

Blok *multiplier* pertama akan melakukan *reset* terlebih dahulu. Selanjutnya sinyal *start* akan menjadi penanda untuk memulai proses. Blok *multiplier* akan meminta data pada BRAM dengan alamat tertentu, kemudian data tersebut akan dikalikan dengan *gain*. Berikutnya data tersebut akan ditulis pada BRAM *output*. Proses penulisan akan melibatkan alamat dan juga sinyal *enable*. Proses tersebut akan diulang sebanyak data yang ada pada BRAM. Setelah selesai, blok akan menghasilkan sinyal *done*.

D. *Pembuatan IP Kontrol*

Blok kontrol digunakan untuk menghubungkan antara bagian PL dan PS. Blok ini terdiri dari bagian *clock* dan *reset*, *Bus* AXI, dan sinyal kontrol. Kode Verilog dari blok ini dapat dilihat pada lampiran [1].

Sinyal kontrol yang digunakan pada blok ini akan dihubungkan sebagai masukan ke blok *multiplier*. Sinyal kontrol terdiri dari banyaknya data, nilai *gain*, dan penanda *start* serta *done*.

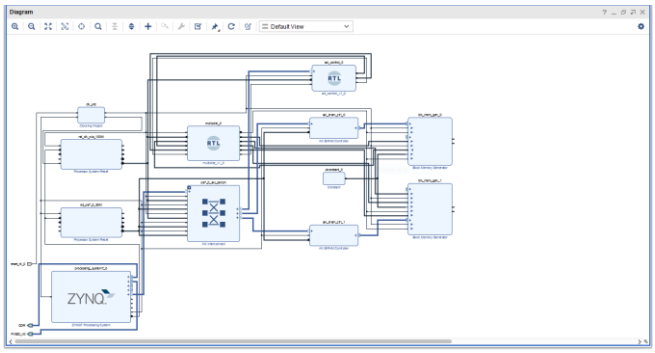
E. *Pembuatan Perangkat Lunak*

Perangkat lunak yang digunakan pada sistem ditulis menggunakan Bahasa C. Kode yang digunakan dapat dilihat pada lampiran. Perangkat lunak ini berfungsi untuk memberikan masukan dan menampilkan keluaran dari sistem [1].

Perangkat lunak ini akan mengakses alamat memori dari blok kontrol dan blok BRAM. Pada blok kontrol dapat diatur nilai *gain*, banyaknya data, dan penanda *start*. Selain itu dapat diakses pula penanda *done* pada perangkat lunak ini. Akses blok BRAM pada perangkat lunak ini digunakan untuk memberikan masukan dan membaca hasil perkalian. Perangkat lunak ini juga bisa digunakan untuk menampilkan nilai pada serial monitor.

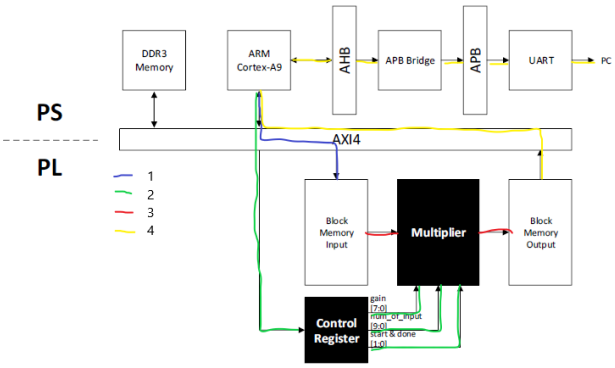
F. *Integrasi Sistem*

Sistem yang digunakan terdiri dari dua buah blok BRAM, blok *multiplier*, dan blok kontrol. Selain itu perlu ditambahkan blok ZYNQ *processing system* untuk mengatur hubungan dengan bagian PS. Blok yang digunakan sistem ini dapat dilihat pada gambar berikut [1]



Gambar 3 Blok BRAM

Sistem ini bekerja dengan mengikuti *data flow* seperti berikut



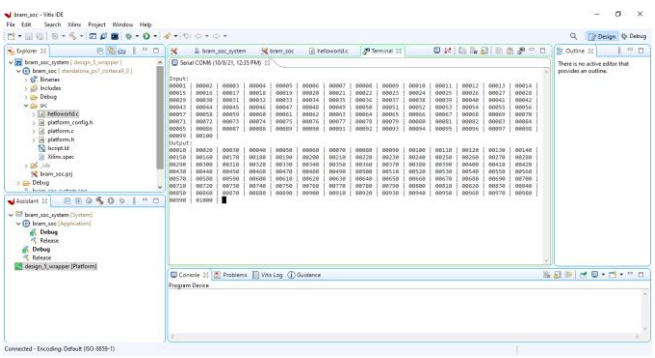
Gambar 4 Data Flow Diagram Sistem

Pertama-tama bagian PS akan menuliskan data ke BRAM *input*, selanjutnya PS akan mengirim nilai sinyal kontrol dan diteruskan ke blok *multiplier*. Kemudian data pada BRAM *input* akan dikalikan dengan *gain* dan hasilnya disimpan pada BRAM *output*. Terakhir, data pada BRAM *output* akan dibaca oleh PS dan ditampilkan pada serial monitor.

III. IMPLEMENTASI

A. *Hasil*

Desain sistem tersebut diimplementasikan ke dalam FPGA XILINX ZYNQ-7000. Hasil implementasi tersebut dapat dilihat pada gambar berikut



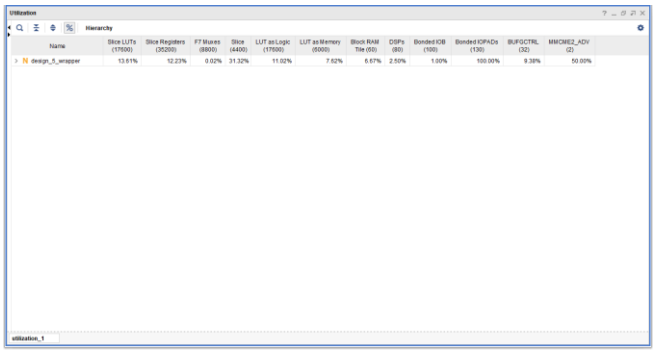
Gambar 5 Hasil Serial Monitor Implementasi Sistem

Dari gambar tersebut dapat dilihat bahwa sistem tersebut dapat menampilkan data pada BRAM *input* dan *output* dengan

baik. Data tersebut sesuai dengan perangkat lunak yang dibuat sebelumnya dan hasil perkalian juga sudah sesuai. Dalam implementasi ini digunakan banyak data yaitu 100 buah data dan nilai *gain* sebesar 10.

B. Resource

Resource yang diperlukan dalam implementasi sistem ini ditampilkan dalam gambar di bawah.



Gambar 6 Resource Implementasi

Sistem memerlukan *Slice LUT* sebesar 13,61%, *LUT as Logic* sebesar 11,02%, *LUT as Memory* sebesar 7,62%, BRAM sebesar 6,67%, dan DSP sebesar 2,5%. Sistem tersebut mengolah data sebanyak 100 buah data dengan nilai *gain* 10.

IV. KESIMPULAN

Dari perancangan sistem tersebut dapat disimpulkan bahwa desain yang dibuat dapat bekerja dengan baik pada FPGA ZYNQ-7000. Proses desain yang dilakukan sudah benar dan dapat ditingkatkan ke skala yang lebih besar

REFERENSI

[1] E. Setiawan, Tutorial Block Memory Generator, Interface Memory, dan Control Register. Bandung. 2018.

1. Kode Multiplier

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05.10.2021 16:22:43
// Design Name:
// Module Name: multiplier
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module multiplier (
    // *** Clock and reset ***
    input wire clk,
    input wire rst_n,
    output wire rst,

    // *** RAM input ***
    output wire [31:0] rd_addr,
    input wire [31:0] rd_data,

    // *** RAM output ***
    output wire [31:0] wr_addr,
    output wire [31:0] wr_data,
    output wire [3:0] wr_en,

    // *** Control signal ***
    input wire [9:0] num_of_inp,
    input wire [7:0] gain,
    input wire start,
    output wire done
    );

    localparam [1:0] S_IDLE = 4'h0,
    S_READ = 4'h1,
    S_WRITE = 4'h2,
    S_DONE = 4'h3;

    reg [1:0] _cs, _ns;
    reg [31:0] rd_addr_cv, rd_addr_nv;
    reg [31:0] wr_addr_cv, wr_addr_nv;
    reg [31:0] wr_data_cv, wr_data_nv;
    reg [3:0] wr_en_cv, wr_en_nv;
    reg done_cv, done_nv;

    // *** Register ***
    always @(posedge clk)
```

```
        begin
            if (!rst_n)
            begin
                _cs <= S_IDLE;
                rd_addr_cv <= 0;
                wr_addr_cv <= 0;
                wr_data_cv <= 0;
                wr_en_cv <= 0;
                done_cv <= 0;
            end
            else begin
                _cs <= _ns;
                rd_addr_cv <= rd_addr_nv;
                wr_addr_cv <= wr_addr_nv;
                wr_data_cv <= wr_data_nv;
                wr_en_cv <= wr_en_nv;
                done_cv <= done_nv;
            end
        end
        // *** Next state and data path ***
        always @(*)
        begin
            _ns = _cs;
            rd_addr_nv = rd_addr_cv;
            wr_addr_nv = wr_addr_cv;
            wr_data_nv = wr_data_cv;
            wr_en_nv = wr_en_cv;
            done_nv = done_cv;
            case (_cs)
                S_IDLE:
                begin
                    if (start)
                    begin
                        if (num_of_inp == 0)
                        begin
                            _ns = S_DONE;
                        end
                        else begin
                            done_nv = 0;
                            rd_addr_nv = 0;
                            _ns = S_READ;
                        end
                    end
                end
                S_READ:
                begin
                    rd_addr_nv = rd_addr_cv + 4;
                    wr_data_nv = rd_data * gain;
                    wr_en_nv = 4'hf;
                    _ns = S_WRITE;
                end
                S_WRITE:
                begin
                    wr_addr_nv = wr_addr_cv + 4;
                    wr_en_nv = 4'h0;
                    if (rd_addr_cv == num_of_inp)
                    begin
                        _ns = S_DONE;
                    end
                    else begin
                        _ns = S_READ;
                    end
                end
```

```
                S_DONE:
                begin
                    rd_addr_nv = 0;
                    wr_addr_nv = 0;
                    done_nv = 1;
                    _ns = S_IDLE;
                end
            endcase
        end

        // *** Output ***
        assign rst = ~rst_n;
        assign rd_addr = rd_addr_cv;
        assign wr_addr = wr_addr_cv;
        assign wr_data = wr_data_cv;
        assign wr_en = wr_en_cv;
        assign done = done_cv;
    endmodule
```

2. Kode Kontrol

```
    `timescale 1ns / 1ps
    //////////////////////////////////////////////////////////////////////////////////
    /////////
    // Company:
    // Engineer:
    //
    // Create Date: 06.10.2021 22:48:29
    // Design Name:
    // Module Name: axi_control
    // Project Name:
    // Target Devices:
    // Tool Versions:
    // Description:
    //
    // Dependencies:
    //
    // Revision:
    // Revision 0.01 - File Created
    // Additional Comments:
    //
    //////////////////////////////////////////////////////////////////////////////////
    /////////


    module axi_control
        #(
            C_ADDR_WIDTH = 32,
            C_DATA_WIDTH = 32
        )
        (
            // *** Clock and reset signals ***
            input wire aclk,
            input wire aresetn,

            // *** AXI4-lite slave signals ***
            output wire s_axi_awready,
            input wire [C_ADDR_WIDTH-1:0] s_axi_awaddr,
            input wire s_axi_awvalid,
            output wire s_axi_wready,
            input wire [C_DATA_WIDTH-1:0] s_axi_wdata,
            input wire [C_DATA_WIDTH/8-1:0] s_axi_wstrb,
            input wire s_axi_wvalid,
```

```
            input wire s_axi_bready,
            output wire [1:0] s_axi_bresp,
            output wire s_axi_bvalid,
            output wire s_axi_arready,
            input wire [C_ADDR_WIDTH-1:0] s_axi_araddr,
            input wire s_axi_arvalid,
            input wire s_axi_rready,
            output wire [C_DATA_WIDTH-1:0] s_axi_rdata,
            output wire [1:0] s_axi_rresp,
            output wire s_axi_rvalid,

            // *** Control signals ***
            output wire [9:0] num_of_inp,
            output wire [7:0] gain,
            output wire start,
            input wire done
        );

        // *** Register map ***
        // 0x00: done[11] (R) | start[10] (R/W) | num_of_inp[9:0] (R/W)
        // 0x04: gain[7:0] (R/W)
        localparam C_ADDR_BITS = 4;
        // *** Address ***
        localparam C_ADDR_CTRL = 4'h00,
        C_ADDR_GAIN = 4'h04;

        // *** AXI write FSM ***
        localparam S_WRIDLE = 2'd0,
        S_WRDATA = 2'd1,
        S_WRRESP = 2'd2;

        // *** AXI read FSM ***
        localparam S_RDIDLE = 2'd0,
        S_RDDATA = 2'd1;

        // *** AXI write ***
        reg [1:0] wstate_cs, wstate_ns;
        reg [C_ADDR_BITS-1:0] waddr;
        wire [31:0] wmask;
        wire aw_hs, w_hs;

        // *** AXI read ***
        reg [1:0] rstate_cs, rstate_ns;
        wire [C_ADDR_BITS-1:0] raddr;
        reg [31:0] rdata; wire ar_hs;

        // *** Internal registers ***
        reg [9:0] num_of_inp_reg;
        reg start_reg;
        reg done_reg;
        reg [7:0] gain_reg;

        // *** User signals ***
        // *** AXI
    write************************************************************

        assign s_axi_awready = (wstate_cs == S_WRIDLE);
        assign s_axi_wready = (wstate_cs == S_WRDATA);
        assign s_axi_bresp = 2'b00;

        // OKAY
        assign s_axi_bvalid = (wstate_cs == S_WRRESP);
```

```
    assign wmask = {{8{s_axi_wstrb[3]}}, {8{s_axi_wstrb[2]}},
{8{s_axi_wstrb[1]}}, {8{s_axi_wstrb[0]}}};
    assign aw_hs = s_axi_awvalid & s_axi_awready;
    assign w_hs = s_axi_wvalid & s_axi_wready;

    // *** Write state register ***

    always @(posedge aclk)
    begin
        if (!aresetn)
            wstate_cs <= S_WRIDLE;
        else
            wstate_cs <= wstate_ns;
    end

    // *** Write state next ***
    always @(*)
    begin
        case (wstate_cs)
            S_WRIDLE:
                if (s_axi_awvalid)
                    wstate_ns = S_WRDATA;
                else
                    wstate_ns = S_WRIDLE;
            S_WRDATA:
                if (s_axi_wvalid)
                    wstate_ns = S_WRRESP;
                else
                    wstate_ns = S_WRDATA;
            S_WRRESP:
                if (s_axi_bready)
                    wstate_ns = S_WRIDLE;
                else
                    wstate_ns = S_WRRESP;
            default:
                wstate_ns = S_WRIDLE;
        endcase
    end

    // *** Write address register ***
    always @(posedge aclk)
    begin
        if (aw_hs)
            waddr <= s_axi_awaddr[C_ADDR_BITS-1:0];
        end

    // *** AXI read
************************************************************
    assign s_axi_arready = (rstate_cs == S_RDIDLE);
    assign s_axi_rdata = rdata;
    assign s_axi_rresp = 2'b00; // OKAY
    assign s_axi_rvalid = (rstate_cs == S_RDDATA);
    assign ar_hs = s_axi_arvalid & s_axi_arready;
    assign raddr = s_axi_araddr[C_ADDR_BITS-1:0];

    // *** Read state register ***
    always @(posedge aclk)
    begin
        if (!aresetn)
            rstate_cs <= S_RDIDLE;
        else
            rstate_cs <= rstate_ns;
    end
```

```
    // *** Read state next ***
    always @(*)
    begin
        case (rstate_cs)
            S_RDIDLE:
                if (s_axi_arvalid)
                    rstate_ns = S_RDDATA;
                else
                    rstate_ns = S_RDIDLE;
            S_RDDATA:
                if (s_axi_rready)
                    rstate_ns = S_RDIDLE;
                else
                    rstate_ns = S_RDDATA;
            default:
                rstate_ns = S_RDIDLE;
        endcase
    end

    // *** Read data register ***
    always @(posedge aclk)
    begin
        if (!aresetn)
            rdata <= 0;
        else if (ar_hs)
            case (raddr)
                C_ADDR_CTRL:
                    rdata <= {done_reg, start_reg, num_of_inp_reg[9:0]};
                C_ADDR_GAIN:
                    rdata <= gain_reg[7:0];
            endcase
    end

    // *** Internal registers
************************************************
    // *** num_of_inp_reg[9:0], start_reg ***
    always @(posedge aclk)
    begin
        if (!aresetn)
            begin
                start_reg <= 0;
                num_of_inp_reg[9:0] <= 0;
            end
        else if (w_hs && waddr == C_ADDR_CTRL)
            begin
                if (s_axi_wdata[10] == 1)
                    start_reg <= 1;
                    num_of_inp_reg[9:0] <= (s_axi_wdata[31:0] & wmask) |
(num_of_inp_reg[9:0] & ~wmask);
            end
        else
            begin
                start_reg <= 0;
            end
    end

    // *** gain_reg[7:0] ***
    always @(posedge aclk)
    begin
        if (!aresetn)
            gain_reg[7:0] <= 0;
        else if (w_hs && waddr == C_ADDR_GAIN)
```

```
                gain_reg[7:0] <= (s_axi_wdata[31:0] & wmask) | (gain_reg[7:0]
& ~wmask);
    end

    // *** done_reg ***
    always @(posedge aclk)
    begin
        if (!aresetn)
            done_reg <= 0;
        else
            done_reg <= done;
    end

    // *** Instantiation
*****************************************************
    assign num_of_inp = num_of_inp_reg;
    assign gain = gain_reg;
    assign start = start_reg;
endmodule
```

3. Kode Perangkat Lunak

```
/*************************************************************************
******
*
* Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.
*
* Permission is hereby granted, free of charge, to any person obtaining a
copy
* of this software and associated documentation files (the "Software"),
to deal
* in the Software without restriction, including without limitation the
rights
* to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell
* copies of the Software, and to permit persons to whom the Software is
* furnished to do so, subject to the following conditions:
*
* The above copyright notice and this permission notice shall be included
in
* all copies or substantial portions of the Software.
*
* Use of the Software is limited solely to applications:
* (a) running on a Xilinx device, or
* (b) that interact with a Xilinx device through a bus or interconnect.
*
* THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
OR
* IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
* FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
* XILINX  BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
* WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT
OF
* OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
* SOFTWARE.
*
* Except as contained in this notice, the name of the Xilinx shall not be
used
* in advertising or otherwise to promote the sale, use or other dealings
in
* this Software without prior written authorization from Xilinx.
*
```

```
*****************************************************************************
*****/

/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include <stdint.h>

#define MEM_INP_BASE 0x40000000
#define MEM_OUT_BASE 0x41000000
#define CTRL_BASE 0x42000000
#define NUM_OF_INPUT 100

uint32_t *meminp_p, *memout_p, *ctrl_p;
int main()
{
   // *** Initialize pointer ***
   meminp_p = (uint32_t *)MEM_INP_BASE;
   memout_p = (uint32_t *)MEM_OUT_BASE;
   ctrl_p = (uint32_t *)CTRL_BASE;

   // Write gain
   *(ctrl_p+1) = 10;

   // *** Write input ***
   printf("\nInput:\n");
   for (int i = 0; i < NUM_OF_INPUT; i++)
   {
         *(meminp_p+i) = i+1;
         printf("%05d | ", (unsigned int)*(meminp_p+i));
   }

   // Write number of input and set start bit
   *(ctrl_p+0) = (1 << 10) | NUM_OF_INPUT*4;

   // Polling until process is done
   while (!(*(ctrl_p+0) & (1 << 11)));

   // *** Read from block memory output ***
   printf("\nOutput:\n");
   for (int i = 0; i < NUM_OF_INPUT; i++)
         printf("%05d | ", (unsigned int)*(memout_p+i));

   return 0;
}
```