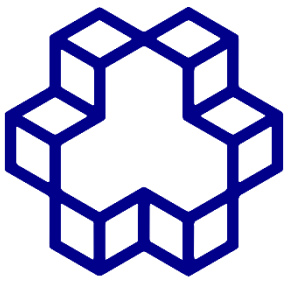
گزارش کار پروژه نهایی پیاده سازی hls روی سخت افزار zynq

علی اکبر محسن نژاد

محمد حسین عالمی رستمی

استاد رامهرمزی

آز طراحی در سطح سیستم



دانشگاه صنعتی خواجه نصیرالدین طوسی

تابستان ۱۴۰۴

فهرست مطالب

مقدمه

این گزارش به بازپیاده‌سازی و ارزیابی شش آزمایش ساده طراحی دیجیتال می‌پردازد که پیش‌تر در محیط **SystemC** *توسعه و شبیه‌سازی شده‌اند و اکنون با هدف انتقال به جریان طراحی سخت‌افزار، در ابزار* **Vivado HLS** *و بر بستر سخت‌افزار* **Zynq** *اجرا می‌شوند.*

برای هر آزمایش، ابتدا مسئله و هدف آن تشریح می‌شود. سپس فرایند نگاشت کدهای **HLS** به‌صورت گام‌به‌گام توضیح داده خواهد شد. در ادامه، نتایج سنتز و پیاده‌سازی شامل مصرف منابع ،تأخیر و... گزارش و با نسخه‌ی مبنا مقایسه می‌گردد. این ساختار امکان می‌دهد اثر انتخاب‌های طراحی در **HLS** به‌روشنی دیده و بهترین شیوه‌ها برای پیاده‌سازی آزمایش‌ها استخراج شود. دقت شود پارت نامبر در برنامه باید برای **zynq7000** انتخاب شود:

Part: xc7z020clg400-1 (Zynq-7000)

آزمایش اول: گیت XOR

هدف از این آزمایش، پیاده‌سازی و شبیه‌سازی یک گیت منطقی **XOR** با استفاده از زبان **C++** در محیط **Vivado HLS** است. در این پروژه، ماژول **XOR** ابتدا با استفاده از کد **HLS** طراحی شده، سپس سنتز شده و در نهایت توسط **testbench** مورد شبیه‌سازی قرار گرفته است، یک فایل هدر، یک سورس فایل و یک فایل تست بنچ برای پیاده سازی ساخته شده اند که به ترتیب توضیح داده می شوند.

xor_gate.h:

```
#ifndef XOR_GATE_H
#define XOR_GATE_H

void xor_gate(bool a, bool b, bool &out);

#endif
```

در این فایل تابع هدر اصلی و سه ورودی و خروجی مورد نیاز تعریف شده اند.

xor_gate.cpp:

```
#include "xor_gate.h"

void xor_gate(bool a, bool b, bool &out) {
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE ap_none port=a
    #pragma HLS INTERFACE ap_none port=b
    #pragma HLS INTERFACE ap_none port=out

    out = a ^ b;
}
```

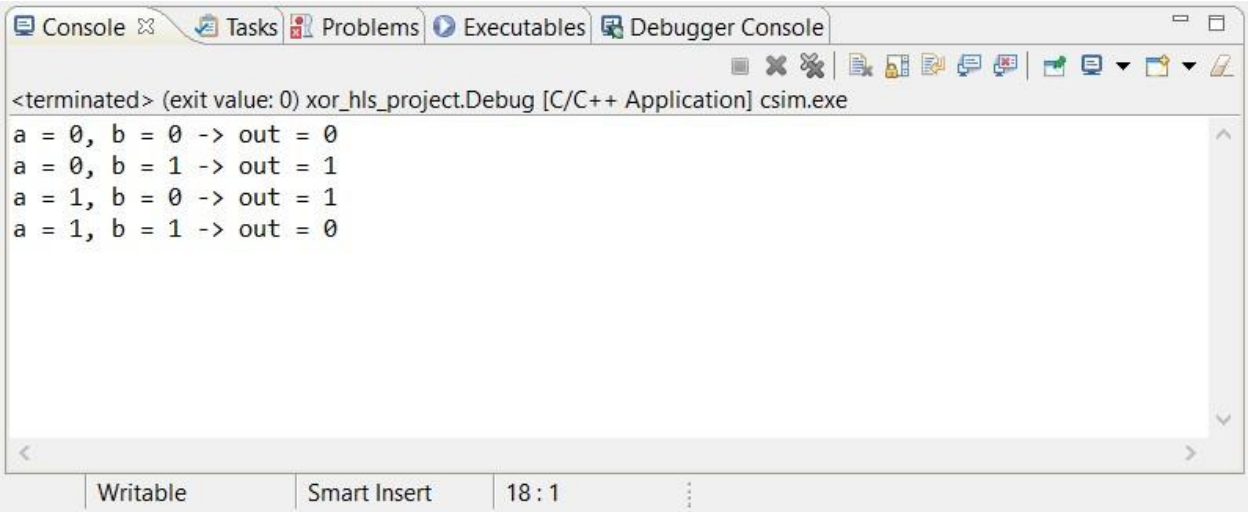
ابتدا هدر را در این فایل **Include** کرده و سپس در در تابع **xor_gate** با استفاده از عملگر **^** عملیات **XOR** روی ورودی‌ها انجام شده و نتیجه در متغیر **out** قرار می‌گیرد.

xor_gate_tb.cpp:

```cpp
#include <iostream>
#include "xor_gate.h"

int main() {
    bool a, b, out;

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            a = i;
            b = j;
            xor_gate(a, b, out);
            std::cout << "a = " << a << ", b = " << b << " -> out = " << out
<< std::endl;
        }
    }

    return 0;
}
```

* یک **Testbench** ساده برای تست چهار حالت مختلف ورودی‌های **a** و **b** نوشته شده است سپس با استفاده از شمارنده **i** از ۰ تا ۳، حالت‌های ممکن ورودی‌ها تولید می‌شوند و در نهایت خروجی برای هر حالت چاپ می‌شود.

تصویر خروجی این تست بنچ را در زیر مشاهده می کنید:



مشخص است که خروجی تست بنچ ها به درستی نشان داده شده اند و پیاده سازی صحیح است.

حال نیم نگاهی به گزارش سنتز این آزمایش می پردازیم: در تصویر پایین تایمینگ و تاخیر گیت طراحی شده گزارش شده است، همچنین اگر مقدار منابع استفاده شده را مشاهده می کنیم می فهمیم که تنها دو **LUT** استفاده

شده است، این باعث می شود که مصرف منابع بسیار پایین باشد و عملکرد دقیق گیت **XOR** با تأخیر بسیار کم حاصل شود.

```
================================================================
== Vivado HLS Report for 'xor_gate'
================================================================
* Date:           Tue Aug  5 18:38:30 2025

* Version:        2019.1 (Build 2552052 on Fri May 24 15:28:33 MDT 2019)
* Project:        xor_hls_project
* Solution:       xor_solution
* Product family: zynq
* Target device:  xc7z020-clg400-1


================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |  10.00|     0.978|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    0|    0|    0|    0|   none  |
    +-----+-----+-----+-----+---------+
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|      2|    -|
|FIFO             |        -|      -|       -|      -|    -|
|Instance         |        -|      -|       -|      -|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|      -|    -|
|Register         |        -|      -|       -|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        0|      0|       0|      2|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |        0|      0|       0|   ~0  |    0|
+-----------------+---------+-------+--------+-------+-----+
```

آزمایش دوم: طراحی واحد محاسبات و منطق (ALU)

هدف این آزمایش طراحی و شبیه‌سازی یک واحد محاسبات و منطق (**ALU**) هشت‌بیتی با **Vivado HLS** به‌گونه‌ای که:

دو ورودی **A,B** هرکدام هشت بیتی به همراه خروجی **R** ۸ بیتی داریم، کد عملگر **OP** ۳ بیتی برای تعیین عملگر مورد نیاز کاربر و همچنین ۴ فلگ که عبارتند از:

- **C (Carry)** : فلگ بیت کری برای جمع و تفریق
- **Z (Zero)** : فلگ صفر شدن پاسخ که در **cpu** واقعی به درد بخور می باشد.
- **N (Negative)**: بیت فلگ کپی از **sign bit** پاسخ برای فهمیدن علامت
- **V (Signed Overflow)**: فلگ برای زمانی که پاسخ سر ریز یا همان **Overflow** می شود.

همچنین ۶ عملگر داریم که توسط **op** انتخاب می شوند:

i. **ADD** : عملگر جمع

ii. **SUB**: عملگر تفریق

iii. **AND**: عملگر **&**

iv. **OR**: عملگر **|**

v. **XOR**: عملگر **^**

vi. **NOT**: عملگر **~**

vii. **INC**: عملگر ۱+ کردن **A**

همچنین فلگ ها بعد از هر عملیات انجام شده محاسبه می شوند.

حال به توضیح کد ها می پردازیم، دوباره سه فایل برای این پروژه داریم که به ترتیب آن هارا شرح می دهیم:

alu_hls.h:

```
#ifndef ALU_HLS_H
#define ALU_HLS_H

#include <ap_int.h>
```

```
void alu_hls_ex(ap_uint<8> A, ap_uint<8> B, ap_uint<3> op,
                ap_uint<8> &R, bool &C, bool &Z, bool &N, bool &V);

#endif
```

در فایل هدر بالا ابتدا کتابخوانه **apt_int** که انواع عدد صحیح با دقت دلخواه به ما می دهد را **include** می کنیم، سپس در تابع **alu_hls_ex** به تعریف ورودی ها، خروجی و کد عملگر و فلگ های توضیح داده شده می پردازیم.

حال به فایل بعدی می رویم:

alu_hls.cpp:

```
#include "alu_hls.h"

    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE ap_none port=A
    #pragma HLS INTERFACE ap_none port=B
    #pragma HLS INTERFACE ap_none port=op
    #pragma HLS INTERFACE ap_none port=R
    #pragma HLS INTERFACE ap_none port=C
    #pragma HLS INTERFACE ap_none port=Z
    #pragma HLS INTERFACE ap_none port=N
    #pragma HLS INTERFACE ap_none port=V


   R = 0; C = 0; Z = 0; N = 0; V = 0;
```

در این قسمت از کد ابتدا پراگما ها را تعریف کردیم، این پراگما ها بیشتر به شبیه سازی سفت افزاری ما کمک می کنند به این صورت که خط اول پراگمای نوشته شده، کاری می کند تا اینترفیس دیفالت برنامه **vivaldo hls** ، برای این پروژه برداشته شود و این **alu** کاملا مانند یک بلوک **combinatinal** کار کند و دستور کنترلی در کار نباشد.

همچنین بقیه پراگما ها کاری می کنند که سیستم در شبیه سازی، ورودی ها، خروجی و فلگ ها را یک سیم ساده در نظر بگیرد تا نیازی به پروتوکل های **handshake** نباشد.

بعد از پراگما های نوشته شده، فلگ ها را ریست می کنیم تا کد **ALU** را بنویسیم.

```
switch (op) {
    case 0: {
        ap_uint<9> sum = (ap_uint<9>)A + (ap_uint<9>)B;
        R = (ap_uint<8>)sum;
        C = sum[8];
        bool SA = A[7], SB = B[7], SR = R[7];
        V = (SA == SB) && (SR != SA);
    } break;

    case 1: {
        ap_uint<9> diff = (ap_uint<9>)A - (ap_uint<9>)B;
        R = (ap_uint<8>)diff;
        bool borrow = (A < B);
        C = !borrow;
        bool SA = A[7], SB = B[7], SR = R[7];
        V = (SA != SB) && (SR != SA);
    } break;
```

در اینجا شروع به نوشتن مود های مختلف **op** می کنیم، در بالا کیس های جمع و تفریق را مشاهده می کنید، که هرکدام علاوه بر انجام عمل خواسته شده و به دست آوردن جواب، فلگ های خاص به خود یعنی کری و بیت اورفلو را محاسبه می کنند.

فلگ های **Z,N** به دلیل اینکه در همه عملگر ها یک جور به دست می آیند در پایان کد به صورت فلگ عمومی آورده شده اند که جلوتر به آن می پردازیم.

```
  case 2: R = A & B; break;
    case 3: R = A | B; break;
    case 4: R = A ^ B; break;
    case 5: R = ~A;    break;

    case 6: {
        ap_uint<9> inc = (ap_uint<9>)A + 1;
        R = (ap_uint<8>)inc;
        C = inc[8];
        bool SA = A[7], SB = 0, SR = R[7];
        V = (SA == SB) && (SR != SA);
    } break;

    default: R = 0; C = 0; V = 0; break;
    }

    Z = (R == 0);
    N = R[7];
}
```

در تیکه نهایی فایل **alu_hls.cpp** ما بقیه کیس های خواسته شده (**AND, OR, XOR, NOT, INC**) را تعریف کردیم، سپس در کیس دیفالت، فلگ های خاص و پاسخ را صفر قرار می دهیم و در نهایت فلگ های عمومی را محاسبه می کنیم، همانطور که گفته بودیم فلگ **Z** زمانی یک می شود که پاسخ ما کاملا صفر باشد و فلگ **N** نیز نشان دهنده مثبت یا منفی بودن جواب است برای همین بیت **MSB** پاسخ را کپی می کند.

حال به فایل تست بنچ کد می رویم:

tb_alu_hls.cpp:

```
#include <iostream>
#include <bitset>
#include "alu_hls.h"
```

ابتدا کتابخوانه های مورد نیاز برای پرینت کردن و فایل هدر پروژه را **include** می کنیم.

```
static void print_u8_bin(ap_uint<8> v, const char* label) {
    std::cout << label << "=" << (unsigned)v
              << " (" << std::bitset<8>(v) << ")";
}
```

با استفاده از این تابع کمکی ما به راحتی می توانیم مقادیر ۸ بیتی خود را هم به صورت باینری و هم دسیمال، پرینت کنیم.

```
static void run_case(ap_uint<8> A, ap_uint<8> B) {
    for (int op = 0; op <= 6; ++op) {
        ap_uint<8> R; bool C, Z, N, V;
        alu_hls_ex(A, B, (ap_uint<3>)op, R, C, Z, N, V);

        print_u8_bin(A, "A"); std::cout << "  ";
        print_u8_bin(B, "B"); std::cout << "  ";
        std::cout << "op=" << op << "  ";
        print_u8_bin(R, "R");
        std::cout << "  C=" << C
                  << "  Z=" << Z
                  << "  N=" << N
                  << "  V=" << V << "\n";
    }
}
```

در تابع **run_case** با استفاده از حلقه **for** ابتدا تمامی ۶ مد **op** را لوپ می کند و می خواند تا همه عملگر ها تست شود سپس تابع **alu_hlx_ex** را از فایل هدر فراخوانی می کند و ورودی ها و مد کنونی را به آن می دهد.

در نهایت موارد زیر را پرینت می کند:

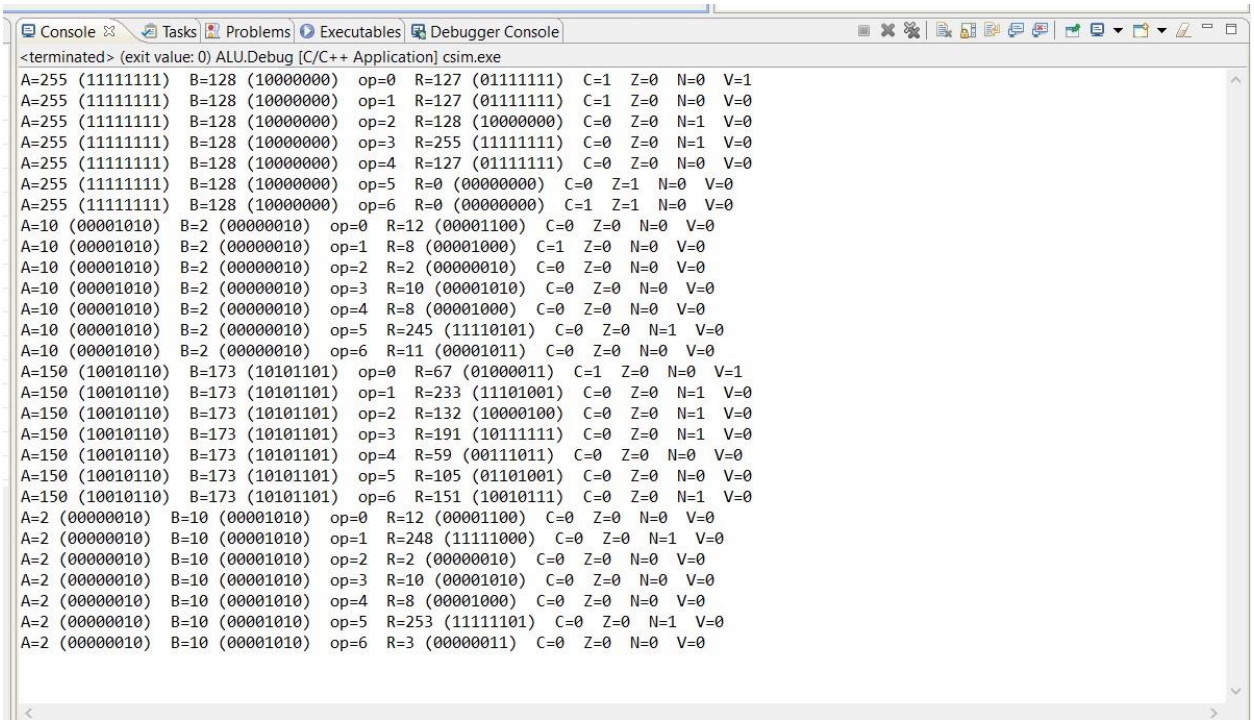
- ورودی های **A,B** ما را به صورت باینری و دسیمال
- مود کنونی **op**
- پاسخ نهایی **R** به صورت باینری و دسیمال
- فلگ های **C, Z, N, V**

در تست بنچ بالا عملا به ازای هر مقدار تست **A,B** ، شما می توانید تمامی حالات ممکن **ALU OP** و فلگ های آن را ببینید.

```
int main() {
    run_case(0xFF, 0x80);
    run_case(0x0A, 0x02);
    run_case(0x96, 0xAD);
    run_case(0x02, 0x0A); // SUB negative example
    return 0;
}
```

در قسمت پایانی تست بنچ به تابع **main** آن می پردازیم، که در آن ۴ حالت مختلف **A,B** را تست می گیریم، به طور مثال در ران کیس اولی درستی کارکرد اورفلو بودن نتیجه را تست می گیریم و در ران کیس آخر، تفریق را در صورتی که **A<B** باشد، تست می کنیم.

بعد از اجرای تست بنچ نتایج زیر را دریافت می کنیم:



همان طور که انتظار داشتیم، **ALU** ما به محاسبه هر ۶ مود، برای هر ران کیس پرداخته است و اگر به نتایج دقت کنیم، متوجه می شویم که بلوک ما به درستی عمل کرده است، برای مثال به همان خط اول پاسخ می پردازیم، اگر **A,B** را علامت دار بگیریم **A=-1** و **B=-128** می شود و اگر آن ها را جمع کنیم سیستم به ما عدد **R=+127** که خب این غلط است چرا که باید جواب **129-** می شد ولی چون این عدد در بازه ۸ بیتی اعداد **signed** قرار ندارد سیستم آن را به **127+** گرد می کند، اینجاست که فلگ **V=1** می شود تا به کاربر یا **CPU** بفهماند که پاسخی که **ALU** به شما داده **Overflow** شده است و جواب اورفلو شده به شما نشان داده شده است.

برای مطمئن شدن از نتیجه خروجی، تفریق تست ران چهارم را نیز چک می کنیم:

عدد اعداد **A=2 , B=10** به صورت **A-B** تفریق می شوند از آنجایی که **A** از **B** کوچکتر است پاسخ ما **8-** می شود و **Two's complement** این عدد، عدد **248** می باشد که در خروجی ما به درستی نشان داده شده است، همچنین از آنجایی که **A<B** بوده است، قرض گرفتن در تفریق اتفاق افتاده و به درستی فلگ **C=0** شده است، همچنین به دلیل منفی بودن پاسخ نیز فلگ **N=1** شده است.

حال که نسبت به درست بودن پاسخ اطمینان پیدا کردیم، نیم نگاهی به گزارش سنتز می اندازیم:

```
================================================================
== Vivado HLS Report for 'alu_hls_ex'
================================================================
* Date:           Sun Aug 10 14:59:01 2025

* Version:        2019.1 (Build 2552052 on Fri May 24 15:28:33 MDT 2019)
* Project:        ALU
* Solution:       ALU_Solution
* Product family: virtex7
* Target device:  xc7vx485t-ffg1157-1


================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |  10.00|     3.981|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    1|    1|    1|    1|   none  |
    +-----+-----+-----+-----+---------+

    + Detail:
        * Instance:
        N/A

        * Loop:
        N/A


================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+--------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |   LUT  | URAM|
+-----------------+---------+-------+--------+--------+-----+
|DSP              |        -|      -|       -|       -|    -|
|Expression       |        -|      -|       0|     153|    -|
|FIFO             |        -|      -|       -|       -|    -|
|Instance         |        -|      -|       -|       -|    -|
|Memory           |        -|      -|       -|       -|    -|
|Multiplexer      |        -|      -|       -|     157|    -|
|Register         |        -|      -|       2|       -|    -|
+-----------------+---------+-------+--------+--------+-----+
|Total            |        0|      0|       2|     310|    0|
+-----------------+---------+-------+--------+--------+-----+
|Available        |     2060|   2800|  607200|  303600|    0|
+-----------------+---------+-------+--------+--------+-----+
|Utilization (%)  |        0|      0|   ~0   |   ~0   |    0|
+-----------------+---------+-------+--------+--------+-----+
```

اگر نگاهی به قسمت تایمینگ بیندازیم، می بینیم که کلاک مد نظر ما ۱۰ نانو ثانیه بوده ولی سفت افزار ما هر کلاک را در ۳.۹۸۱ نانوثانیه اجرا می کند که به مقدار زیادی سریعتر از کلاک مد نظر ما می باشد و این خوب است.

همچنین با توجه به قسمت **latency** متوجه می شویم که **ALU** ما در یک کلاک سایکل برای ما خروجی تولید می کند، همچنین ما می توانیم در شروع هر کلاک ورودی به بلوک بدهیم و نیازی به صبر نداریم.

Pipeline type نیز **none** است و این به این معنا است که بلوک ما کاملا **Combinational** می باشد.

در نهایت با دقت در مقدار منابع استفاده شده متوجه می شویم، مقدار خیلی کمی از منابع مورد نیاز یعنی **Expression, Multiplexer** استفاده شده است.

تمامی موارد بالا این خبر را می دهد که بلوک ما خیلی سریع و تنها پس از یک کلاک، خروجی به ما می دهد که این عالی است و مشکلی برای ما ایجاد نمی کند.

آزمایش سوم: پیاده سازی Register Bank

هدف این آزمایش پیاده‌سازی و ارزیابی یک بانک رجیستر/حافظه ۸بیتی با ۲۵۶ مکان در **Vivado HLS** و سفت افزار **zynq-7000** می باشد.

این کار با استفاده از ورودی آدرس و داده ۸ بیتی و دستور های **Write, Read** ، خروجی هشت بیتی و سیگنال کنترلی **enable** ، صورت می گیرد.

نکته مهم این است که باید یک اولویت رفتاری مشخصی را رعایت کنیم؛ وقتی **enable=1**، ابتدا اگر **write=1** باشد می‌نویسیم و سپس اگر **read=1** باشد می‌خوانیم (از همان آدرس).

پیاده سازی این بانک با استفاده از یک فایل هدر و یک سورس فایل انجام شده و پس از آن با یک فایل تست بنچ، درستی عملکرد آن آزمایش شده است.

فایل ها را به ترتیب توضیح می دهیم و ابتدا به فایل هدر می پردازیم:

regbank_hls.h:

```
#ifndef REGBANK_HLS_H
#define REGBANK_HLS_H

#include <ap_int.h>

void regbank_hls(ap_uint<8> addr,
                 ap_uint<8> in_data,
                 bool write,
                 bool read,
                 bool enable,
                 ap_uint<8> &out_data);

#endif
```

در فایل هدر ورودی ها و خروجی و عملگر های خواندن و نوشتن و سیگنال کنترلی **enable** را تعریف می کنیم همچنین از کتابخوانه **apt_int.h** که انواع عدد صحیح با دقت دلخواه به ما می دهد، استفاده می کنیم.

حال به تشریح فایل سورس می پردازیم:

regbank_hls.cpp:

```
#include "regbank_hls.h"

    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE ap_none port=addr
    #pragma HLS INTERFACE ap_none port=in_data
    #pragma HLS INTERFACE ap_none port=write
```

```
    #pragma HLS INTERFACE ap_none port=read
    #pragma HLS INTERFACE ap_none port=enable
    #pragma HLS INTERFACE ap_none port=out_data

    // 256 x 8-bit register bank (RAM)
    static ap_uint<8> mem[256];
    #pragma HLS RESOURCE variable=mem core=RAM_1P

    // default
    out_data = 0;

    if (enable) {
        if (write) {
            mem[addr] = in_data;
        }
        if (read) {
            out_data = mem[addr];
        }
    }
}
```

ابتدا فایل هدر پروژه را فراخوانی کردیم، سپس با استفاده از پراگما ها، پروژه را به سمت ساده سازی شده ای می بریم، به این معنا که مثلا ورودی ها و خروجی ما تنها یک سیم معمولی هستند و نیازی به **handshake** و سیگنال های کنترلی ندارند، یا اینکه خود پروژه نیازی به فلگ و سیگنال های شروع و پایان ندارد و زمانی که ورودی داده می شود، عمل می کند.

بعد از آن با استفاده از یک آرایه استاتیک، مموری ۸ بیتی، ۲۵۶ خود را می سازیم و با دستورات شرطی، **enable, write, read** را پیاده سازی می کنیم به طوری که اگر **enable** یک بود، حال برو و شرط های نوشتن و خواندن را چک کن و انجامشان بده. همچنین شرط اولویت انجام نوشتن و سپس خواندن نیز در نظر گرفته شده است.

بعد از پیاده سازی کامل بانک رجیستری، تست بنچ را می نویسیم:

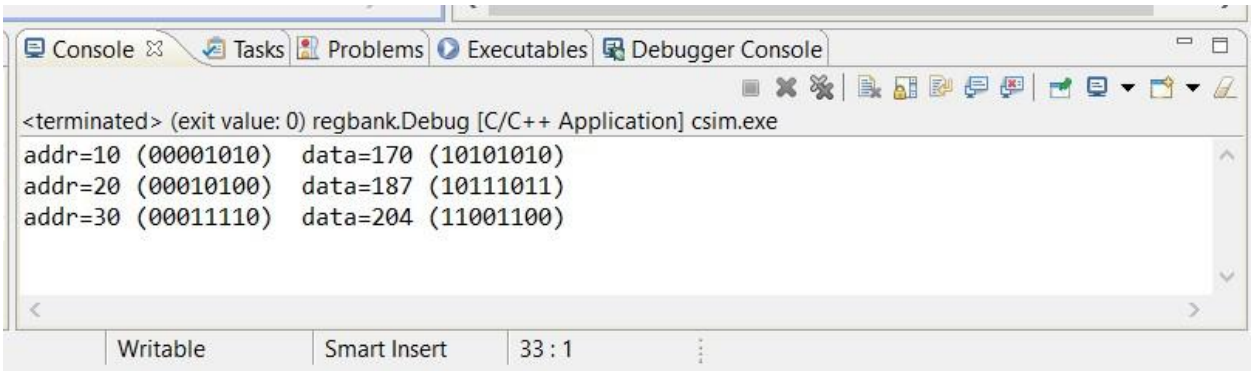
tb_regbank_hls.cpp:

```
#include <iostream>
#include <bitset>
#include "regbank_hls.h"

static void print_u8_bin(ap_uint<8> v, const char* lbl) {
    std::cout << lbl << "=" << (unsigned)v
              << " (" << std::bitset<8>(v) << ")";
}

int main() {
    ap_uint<8> out;

    // --- Writes: addr=10->0xAA, 20->0xBB, 30->0xCC ---
```

```
    regbank_hls(10, 0xAA, /*write=*/true,  /*read=*/false, /*enable=*/true,
out);
    regbank_hls(20, 0xBB, /*write=*/true,  /*read=*/false, /*enable=*/true,
out);
    regbank_hls(30, 0xCC, /*write=*/true,  /*read=*/false, /*enable=*/true,
out);

    // --- Reads (like the SystemC flow) ---
    regbank_hls(10, 0x00, /*write=*/false, /*read=*/true,  /*enable=*/true,
out);
    print_u8_bin(10, "addr"); std::cout << "  "; print_u8_bin(out, "data");
std::cout << "\n";

    regbank_hls(20, 0x00, /*write=*/false, /*read=*/true,  /*enable=*/true,
out);
    print_u8_bin(20, "addr"); std::cout << "  "; print_u8_bin(out, "data");
std::cout << "\n";

    regbank_hls(30, 0x00, /*write=*/false, /*read=*/true,  /*enable=*/true,
out);
    print_u8_bin(30, "addr"); std::cout << "  "; print_u8_bin(out, "data");
std::cout << "\n";

    return 0;
}
```

در تست بنچ بالا پس از فراخوانی کتابخوانه های مورد نیاز، با استفاده از یک تابع کمکی، پرینت اعداد به صورت باینری و دسیمال را ممکن می کنیم.

بعد از آن در تابع **main**، *سه بار تابع* **regbank_hls** *را از فایل هدر برای* **write** *کردن و سه بار برای* **read** *کردن فراخوانی می کنیم، آدرس و داده مدنظر به طور رندوم وارد شده و بعد از خواندن داده ها و پرینت آن ها خروجی زیر قابل مشاهده است:*



همانطور که مشاهده می کنید، توانستیم ابتدا با استفاده از دستور **write** *داده هایی را در آدرس های مشخص قرار دهیم و با دستور* **read** *آن ها را بخوانیم و در خروجی پرینت کنیم.*

حال که از درستی کد مطمئن شدیم به تحلیل گزارش سنتز می پردازیم:

در قسمت تایمینگ مشاهده می کنیم که مقدار زمان هرکلاک تقریبا ۲.۲۶۷ نانو ثانیه می باشد که برای ما که تا ۱۰ نانوثانیه در نظر گرفته بودیم زمان خیلی خوبی است و نشان می دهد، سفت افزار ما خیلی سریعتر از انتظار ما عمل می کند.

```
================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |  10.00|     2.267|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    1|    2|    1|    2|   none  |
    +-----+-----+-----+-----+---------+
```

همچنین در قسمت **latency** متوجه می شویم که زمان انجام دستورات حداقل یک و حداکثر دو سایکل کلاک می باشد و این نتیجه بدی نیست، همچنین **pipeline** داخلی هم نداریم و سیستم کاملا **Combinational** می باشد.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       -|      -|    -|
|FIFO             |        -|      -|       -|      -|    -|
|Instance         |        -|      -|       -|      -|    -|
|Memory           |        1|      -|       0|      0|    0|
|Multiplexer      |        -|      -|       -|     51|    -|
|Register         |        -|      -|       3|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        1|      0|       3|     51|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |    ~0   |      0|   ~0   |   ~0  |    0|
+-----------------+---------+-------+--------+-------+-----+
```

در قسمت مقدار منابع استفاده شده نیز مشاهده می شود که مموری، مولتی پلکسر و رجیستر استفاده شده است. مقدار منابع استفاده شده به شدت کم است که این قضیه هم سرعت را بالاتر می برد و هم در مصرف انرژی به صرفه بودن خود را نشان می دهد.

تمامی موارد بالا این خبر را می دهد که بلوک ما خیلی سریع می باشد و تنها پس از یک یا دو کلاک، خروجی به ما می دهد که این مورد انتظار ما است و مشکلی برایمان ایجاد نمی کند.

آزمایش چهارم: طراحی و پیاده سازی FSM

هدف این آزمایش پیاده‌سازی یک ماشین حالت محدود (**Finite State Machine**) به‌صورت ماژول سخت‌افزاری در **Vivado HLS** و بر روی سخت افزار **Zynq-7000** است که بتواند چهار حالت اصلی زیر را مدیریت کند:

۱. **IDLE** – ریست یا توقف سیستم

۲. **LOAD** – بارگذاری یک مقدار اولیه در شمارنده

۳. **UP_COUNT** – افزایش مقدار شمارنده

۴. **DOWN_COUNT** – کاهش مقدار شمارنده

همچنین یک مالتی‌پلکسر دو به یک نیز پیاده‌سازی شده که بین مقدار شمارنده و یک مقدار ثابت سوئیچ می‌کند.

این ماژول ورودی های زیر را دارد:

- **reset** برای ریست سیستم
- **start** برای فعال‌سازی تغییر حالت
- **mode** (دو بیت) برای تعیین حالت جدید **FSM**
- **sel_mux** برای انتخاب خروجی **MUX** (۰=ثابت، ۱=شمارنده)
- **in_data** داده ورودی برای حالت **LOAD**
- **const_data** داده ثابت برای ورودی **MUX**

خروجی های این ماشین حالت نیز به شرح زیر است:

- **mux_out** خروجی انتخاب‌شده توسط **MUX**
- **status** کد حالت جاری **FSM**

اکنون ابتدا به توضیح فایل های هدر و سورس پیاده سازی ماژول خود می پردازیم و بعد از آن تست بنچ مد نظر خود را تعریف می کنیم:

در فایل هدر نیز موارد زیر را انجام می دهیم:

- تعریف تابع تاپ فانکشن **fsm_hls** با ورودی/خروجی‌ها
- استفاده از **ap_uint** برای اطمینان از اندازه ثابت بیت‌ها

کد مورد نظر به شرح زیر است:

fsm_hls.h:

```
#ifndef FSM_HLS_H
#define FSM_HLS_H

#include <ap_int.h>

// States: 0=IDLE, 1=LOAD, 2=UP_COUNT, 3=DOWN_COUNT
// sel_mux: 0 -> select const_data, 1 -> select counter
void fsm_hls(bool reset,
             bool start,
             ap_uint<2> mode,
             bool sel_mux,
             ap_uint<32> in_data,
             ap_uint<32> const_data,
             ap_uint<32> &mux_out,
             ap_uint<2>  &status);

#endif
```

همانطور که بالاتر گفته شد ابتدا از کتابخوانه **ap_int** برای تعریف ورودی ها و خروجی ها و متغیر ها استفاده می کنیم و سپس طبق کامنتی که در کد بالا نیز وجود دارد ورودی ها و خروجی ها را تعریف می کنیم.

حال به تشریح فایل سورس می پردازیم، در این فایل قصد داریم موارد زیر انجام شود:

- تعریف **enum** برای حالت‌ها (**IDLE, LOAD, UP_COUNT, DOWN_COUNT**)
- استفاده از متغیرهای **static** برای نگهداشتن حالت و شمارنده بین کلاک‌ها
- مدیریت تغییر حالت با توجه به ورودی **start** و **mode**

- اجرای عملیات هر حالت روی شمارنده
- تنظیم خروجی **status** و **mux_out** بر اساس حالت **FSM** و انتخاب **MUX**

کد مدنظر به شرح زیر است:

fsm_hls.cpp:

```
#include "fsm_hls.h"

enum State { IDLE=0, LOAD=1, UP_COUNT=2, DOWN_COUNT=3 };

    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE ap_none port=reset
    #pragma HLS INTERFACE ap_none port=start
    #pragma HLS INTERFACE ap_none port=mode
    #pragma HLS INTERFACE ap_none port=sel_mux
    #pragma HLS INTERFACE ap_none port=in_data
    #pragma HLS INTERFACE ap_none port=const_data
    #pragma HLS INTERFACE ap_none port=mux_out
    #pragma HLS INTERFACE ap_none port=status

    static ap_uint<32> counter = 0;
    static State       state   = IDLE;

    if (reset) {
        state   = IDLE;
        counter = 0;
    } else {
        if (start) {
            // follow mode when start=1 (IDLE/LOAD/UP/DOWN mapping)
            switch ((unsigned)mode) {
                case 0: state = IDLE;       break;
                case 1: state = LOAD;       break;
                case 2: state = UP_COUNT;   break;
                case 3: state = DOWN_COUNT;break;
                default: state = IDLE;      break;
            }
        }

        // state actions
        switch (state) {
            case IDLE:      counter = 0;              break;
            case LOAD:      counter = in_data;        break;
            case UP_COUNT:  counter = counter + 1;    break;
            case DOWN_COUNT:counter = counter - 1;    break;
        }
    }

    status = (ap_uint<2>)state;

    // MUX 2:1 (combinational)
```

```
    mux_out = sel_mux ? counter : const_data;
}
```

ابتدا طبق هرف های مشخص شده، برای حالت های خود **enum state** تعریف کردیم و سپس با استفاده از پراگما ها ، پروژه را به سمت ساده سازی شده ای می بریم، به این معنا که مثلا ورودی ها و خروجی ما تنها یک سیم معمولی هستند و نیازی به **handshake** و سیگنال های کنترلی ندارند، یا اینکه خود پروژه نیازی به فلگ و سیگنال های شروع و پایان ندارد و زمانی که ورودی داده می شود، عمل می کند.

بعد از آن شمارنده و حالت خود را ریست کرده و شروع به نوشتن دستورات شرطی برای پیاده سازی ماشین حالت می کنیم، بدین صورت که ابتدا اگر حالت ریست بودیم، به حالت **IDLE** می رویم و شمارنده را صفر می کنیم در غیر این صورت ماشین حالت شروع به کار می کند و حالت ها با استفاده از دستورات **switch, case** ای که نوشتیم، حالت ها را در هر مرحله تعریف می کنیم، سپس با یک **switch case** دیگری عملکرد های هر حالت را پیاده سازی می کنیم و در پایان حالت یا **status** کنونی ماشین حالت را محاسبه می کنیم تا بتوانیم در خروجی نمایش دهیم.

در نهایت هم خروجی اصلی خود یعنی **mux_out** را از خروجی مالتی پلکسر مد نظر به دست می آوریم.

حال به توضیح تست بنچ می پردازیم:

تست‌بنچ طراحی شده شامل چند مرحله است:

۱. ریست سیستم و بررسی خروجی اولیه

۲. بارگذاری مقدار ۴۲ در شمارنده (**LOAD**)

۳. نمایش مقدار شمارنده با انتخاب **MUX** روی شمارنده

۴. دو بار افزایش شمارنده (**UP_COUNT**)

۵. یک بار کاهش شمارنده (**DOWN_COUNT**)

۶. بازگشت به حالت **IDLE** و ریست شمارنده

برای به دست آوردن مراحل بالا به نکات زیر دقت می کنیم:

- تعریف تابع **step** برای اجرای یک سیکل شبیه‌سازی با پارامترهای مشخص

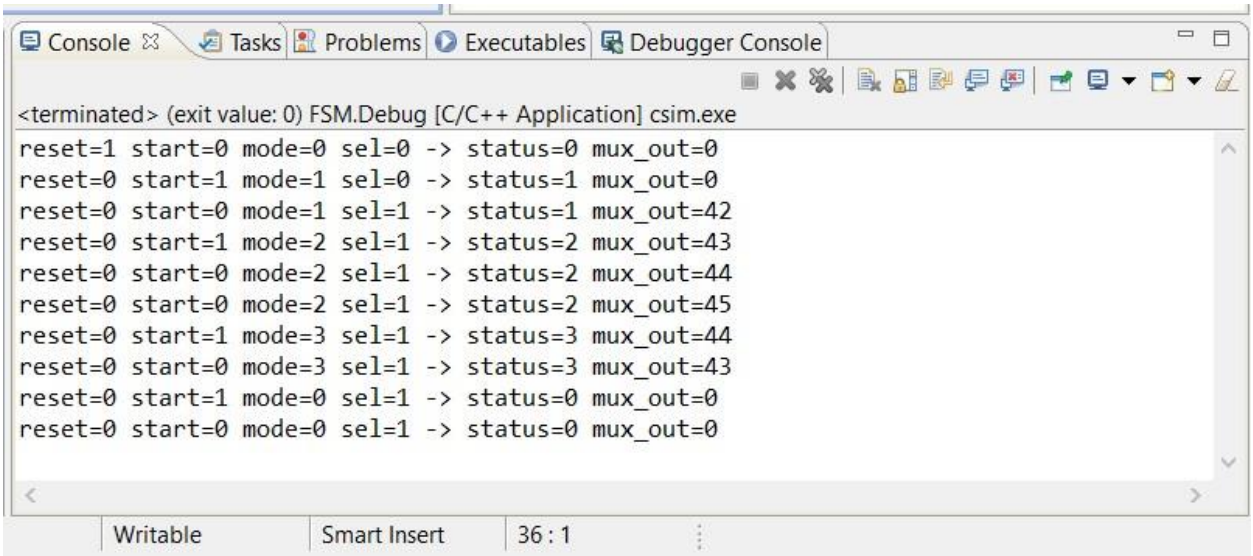
- چاپ مقادیر ورودی‌ها و خروجی‌ها در هر سیکل
- اجرای مراحل تست‌بنچ به ترتیب حالات مختلف **FSM**

کد تست بنچ به شرح زیر می باشد:

tb_fsm_hls.cpp:

```
#include <iostream>
#include "fsm_hls.h"

static void step(bool reset, bool start, ap_uint<2> mode, bool sel_mux,
                 ap_uint<32> in_data, ap_uint<32> const_data)
{
    ap_uint<32> mux_out; ap_uint<2> status;
    fsm_hls(reset, start, mode, sel_mux, in_data, const_data, mux_out,
status);
    std::cout << "reset=" << reset
              << " start=" << start
              << " mode="  << (unsigned)mode
              << " sel="   << sel_mux
              << " -> status=" << (unsigned)status
              << " mux_out=" << (unsigned)mux_out << "\n";
}

int main() {
    // reset
    step(true,  false, 0, 0, 0, 0x00000000);
    // LOAD 42
    step(false, true,  1, 0, 42, 0x00000000); // take mode=LOAD
    step(false, false, 1, 1, 42, 0xAAAA5555); // hold; sel=1 -> counter shown
    // UP_COUNT two times
    step(false, true,  2, 1, 0, 0);           // mode=UP
    step(false, false, 2, 1, 0, 0);           // +1
    step(false, false, 2, 1, 0, 0);           // +1
    // DOWN_COUNT once
    step(false, true,  3, 1, 0, 0);           // mode=DOWN
    step(false, false, 3, 1, 0, 0);           // -1
    // IDLE
    step(false, true,  0, 1, 0, 0);           // mode=IDLE
    step(false, false, 0, 1, 0, 0);           // counter -> 0

    return 0;
}
```

همانطور که بالاتر گفته شد، تابع **step** برای اجرای شبیه سازی یک سیکل مشخص تعریف می کنیم همچنین بعد از آن نحوه پرینت ورودی ها و خروجی ها را مشخص می کنیم و در تابع **main** طبق گفته صفحه پیش، مراحل از پیش تعیین شده را با کمک تابع **step** اجرا می کنیم، خروجی نهایی ما به شرح زیر است:



اگر بخواهیم، خروجی به دست آمده را تفسیر کنیم باید بگوییم، ابتدا ریست اولیه باعث صفر شدن شمارنده و حالت می‌شود (**status=0, mux_out=0**)

سپس در حالت **LOAD** مقدار ۴۲ در شمارنده ذخیره می‌شود و خروجی **MUX** (با **sel=1**) همان مقدار را نشان می‌دهد بعد از آن در حالت **UP_COUNT**، شمارنده به ۴۳ و سپس ۴۴ افزایش یافته است سپس در حالت **DOWN_COUNT**، مقدار شمارنده از ۴۴ به ۴۳ کاهش یافته است و در نهایت بازگشت به حالت **IDLE** شمارنده را صفر کرده است.

حال که از درستی نتیجه مطمئن شده ایم، نیم نگاهی به گزارش سنتز این آزمایش می اندازیم:

```
================================================
== Performance Estimates
================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |  10.00|     5.156|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    1|    1|    1|    1|   none  |
    +-----+-----+-----+-----+---------+
```

در قسمت تایمینگ مشاهده می کنیم که مقدار زمان هرکلاک تقریبا ۵.۱۲۶ نانو ثانیه می باشد که برای ما که تا ۱۰ نانوثانیه زمان کلاک را در نظر گرفته بودیم زمان خیلی خوبی است و نشان می دهد، سفت افزار ما خیلی سریعتر از انتظاری که داشتیم عمل می کند.

همچنین در قسمت **latency** مشاهده می کنیم که این ماشین حالت در یک سایکل کلاک، خروجی را به ما تحویل می دهد و آماده دریافت دستور بعدی خود است.

در مصرف منابع برای پیاده سازی سخت افزاری این ماژول، از **Expression, Multiplexer, register** که از **FF** و **LUT** هستند، استفاده شده است، اما مقدار استفاده بسیار کم است و باعث می شود خیلی به چشم نیاید.

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|    110|    -|
|FIFO             |        -|      -|       -|      -|    -|
|Instance         |        -|      -|       -|      -|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|    177|    -|
|Register         |        -|      -|     102|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        0|      0|     102|    287|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |        0|      0|   ~0   |   ~0  |    0|
+-----------------+---------+-------+--------+-------+-----+
```

تمامی موارد بالا این نوید را می دهد که بلوک ما خیلی سریع می باشد و تنها پس از یک کلاک تاخیر، خروجی را تحویل می دهد که این مورد انتظار ما است و مشکلی برایمان ایجاد نمی کند.

آزمایش پنجم: پیاده سازی Single Interrupt

هدف این آزمایش طراحی و شبیه‌سازی یک ماژول سخت‌افزاری است که بتواند به یک سیگنال وقفه (**Interrupt**) واکنش نشان دهد. در صورت وقوع وقفه، ماژول باید از روند عادی خود خارج شده و به اجرای یک بخش مشخص (سرویس وقفه) بپردازد و پس از اتمام، مجدداً به روند عادی بازگردد. این پیاده‌سازی در محیط **Vivado HLS** انجام شده و قابل استفاده بر روی سخت‌افزارهای مبتنی بر **Zynq-7000** است.

ماژول دارای دو حالت اصلی است:

۱. حالت عادی (**Normal Mode**) – زمانی که سیگنال **interrupt** غیر فعال باشد، خروجی ماژول از ورودی **normal_in** گرفته می‌شود.

۲. حالت وقفه (**Interrupt Mode**) – زمانی که سیگنال **interrupt** فعال شود، خروجی ماژول از ورودی **intr_in** تأمین می‌گردد.

همچنین یک سیگنال وضعیت (**status**) برای مشخص کردن حالت فعلی ماژول استفاده می‌شود:

- مقدار ۰ ⟵ حالت **IDLE** یا ریست
- مقدار ۱ ⟵ حالت عادی
- مقدار ۲ ⟵ حالت سرویس وقفه (**ISR**)

برای پیاده سازی این ماژول، سه فایل زیر را تهیه می کنیم:

`single_intr_hls.h` ⟵ تعریف تاپ فانکشن و پورت‌ها

`single_intr_hls.cpp` ⟵ پیاده‌سازی منطق اصلی تغییر حالت بین حالت عادی و وقفه با استفاده از متغیرهای **static** برای حفظ وضعیت بین سیکل‌ها.

و در نهایت با یک فایل تست بنچ، چندین مرحله تست، با مقادیر مختلف ورودی و چاپ خروجی و وضعیت برای تحلیل عملکرد، انجام می دهیم.

کد فایل هدر به شرح زیر است:

single_intr_hls.h:

```
#ifndef SINGLE_INTR_HLS_H
```

```
#define SINGLE_INTR_HLS_H

#include <ap_int.h>

void single_intr_hls(
    bool reset,
    bool interrupt,
    ap_uint<8> normal_in,
    ap_uint<8> intr_in,
    ap_uint<8> &out_data,
    ap_uint<2> &status
);

#endif
```

در این فایل پس از فراخوانی کتابخوانه **apt_int.h** برای مطمئن بودن از دقیق بودن اعداد بیت در متغیر ها، به ساخت تابع **single_intr_hls** میپردازیم که در آن ورودی نرمال، ورودی وقفه، خروجی ، وضعیت و فلگ های ریست و وقفه قرار دارد.

چگونگی استفاده از این متغیر ها در فایل سورس هم اکنون به شما توضیح می دهیم:

کد فایل سورس به شرح زیر است:

single_intr_hls.cpp:

```
#include "single_intr_hls.h"

#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=reset
#pragma HLS INTERFACE ap_none port=interrupt
#pragma HLS INTERFACE ap_none port=normal_in
#pragma HLS INTERFACE ap_none port=intr_in
#pragma HLS INTERFACE ap_none port=out_data
#pragma HLS INTERFACE ap_none port=status

    static ap_uint<8> data_reg = 0;
    static ap_uint<2> state = 0; // 0=IDLE, 1=NORMAL, 2=ISR

    if (reset) {
        data_reg = 0;
        state = 0;
    } else {
        if (interrupt) {
            data_reg = intr_in;
            state = 2;
        } else {
            data_reg = normal_in;
            state = 1;
        }
    }

    out_data = data_reg;
```

```
    status = state;
```

در این فایل پس از فراخوانی هدر فایلی که در مرحله قبل نوشتیم، با استفاده از پراگما ها ، پروژه را به سمت ساده سازی شده ای می بریم، به این معنا که مثلا ورودی ها و خروجی ما تنها یک سیم معمولی هستند و نیازی به **handshake** و سیگنال های کنترلی ندارند، یا اینکه خود پروژه نیازی به فلگ و سیگنال های شروع و پایان ندارد و زمانی که ورودی داده می شود، عمل می کند.

سپس حالت و دیتا رجیستر خود را ریست و روی حالت دیفالت قرار می دهیم و بعد از آن شروط مورد نیاز را می نویسیم.

اگر ریست اتفاق افتاد، حالت و دیتا رجیستر صفر می شوند، در غیر آن صورت اگر وقفه ای صورت نمی گیرد، **Status=1** می شود و دیتا معمولی در دیتا رجیستر ما ریخته می شود.

ولی اگر حالت وقفه اتفاق افتاد، **Status=2** می شود، دیتای معمولی را رها کرده و دیتای ورودی وقفه را می خوانیم.

حال که ماژول را پیاده سازی کرده ایم، برای تست آن از تست بنچی با مراحل ارزیابی زیر استفاده می کنیم:

۱. اعمال ریست (**reset=1**) و بررسی اینکه خروجی و وضعیت به مقدار اولیه برگردند.

۲. اجرای حالت عادی با **interrupt=0** و بررسی خروجی برابر با **normal_in**.

۳. فعال کردن وقفه (**interrupt=1**) و بررسی تغییر خروجی به **intr_in**.

۴. بازگشت به حالت عادی پس از غیرفعال شدن وقفه.

کد مورد نظر به صورت زیر است:

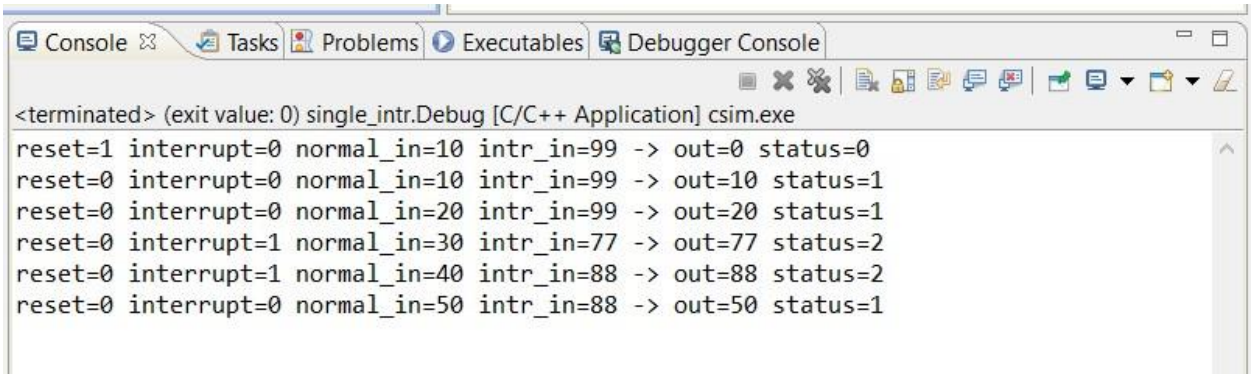
tb_single_intr_hls.cpp:

```
#include <iostream>
#include "single_intr_hls.h"

static void step(bool reset, bool interrupt, ap_uint<8> normal_in, ap_uint<8>
intr_in) {
    ap_uint<8> out;
    ap_uint<2> status;
    single_intr_hls(reset, interrupt, normal_in, intr_in, out, status);
    std::cout << "reset=" << reset
              << " interrupt=" << interrupt
              << " normal_in=" << (unsigned)normal_in
```

```
              << " intr_in=" << (unsigned)intr_in
              << " -> out=" << (unsigned)out
              << " status=" << (unsigned)status << "\n";
}

int main() {
    step(true,  false, 10, 99);  // reset
    step(false, false, 10, 99);  // normal
    step(false, false, 20, 99);  // normal
    step(false, true,  30, 77);  // ISR
    step(false, true,  40, 88);  // ISR
    step(false, false, 50, 88);  // normal
    return 0;
}
```

در تست بنچ بالا پس از فراخوانی کتابخوانه مورد نظر و فایل هدر، یک تابع **step** درست می کنیم که با استفاده از آن ابتدا تابع **single_intr_hls** را فراخوانی و مقدار دهی می کنیم و بعد از آن نیز نحوه پرینت در خروجی را مشخص می کنیم.

در نهایت در تابع **main** با استفاده از تابع **step** نوشته شده، همه سه حالت قابل انجام این ماژول به ترتیب گفته شده در صفحه قبل تست می شود و پس از اجرا کردن آن خروجی زیر نمایش داده می شود:



از خروجی بالا مشخص است که کاملا به هدف و پاسخ خواسته شده رسیده ایم:

- در حالت ریست، مقدار خروجی صفر و وضعیت ۰ (**IDLE**) است.
- در حالت عادی، خروجی برابر با **normal_in** و وضعیت ۱ است.
- در حالت وقفه، خروجی برابر با **intr_in** و وضعیت ۲ است.
- پس از غیرفعال کردن وقفه، سیستم به حالت عادی بازمی‌گردد.

حال که از درستی کارکرد ماژول مطمئن شده ایم، نیم نگاهی به گزارش سنتز نرم افزار می اندازیم:

```
================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |  10.00|     1.248|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    0|    0|    0|    0|   none  |
    +-----+-----+-----+-----+---------+

================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|     23|    -|
|FIFO             |        -|      -|       -|      -|    -|
|Instance         |        -|      -|       -|      -|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|      -|    -|
|Register         |        -|      -|       -|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        0|      0|       0|     23|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |        0|      0|       0|   ~0  |    0|
+-----------------+---------+-------+--------+-------+-----+
```

در قسمت تایمنیگ مشاهده می کنیم ما قصد داشتیم این ماژول را با کلاک ۱۰۰ مگاهرتزی اجرا کنیم ولی انقدر ماژول ساده و بهینه است که بیشترین زمانی که برای اجرای عملکرد آن نیاز داریم تنها ۱.۲۴۸ نانو ثانیه است که خیلی سریعتر از ۱۰ میلی ثانیه ای است که ما در نظر گرفتیم و این عالی است.

همچنین در قسمت **Latency** مشاهده می کنیم که دوباره به دلیل بهینه بودن پیاده سازی، به غیر از کلاک عملکردی که داریم، در گرفتن پاسخ هیچ تاخیری نداریم برای همین **Latency** ما صفر می باشد که این باز هم عالی است و نوید یک طراحی کاملا **Combinational** را می دهد.

در قسمت مصرف منابع هم تنها از ۲۳ **LUT** استفاده شده است که مقدار بسیاری پایینی می باشد و مشکلی برای ما نیست.

تمامی موارد بالا این نوید را می دهد که بلوک ما خیلی سریع می باشد و تنها پس از کلاک عملکردی، خروجی را تحویل می دهد که این مورد انتظار ما است و مشکلی برایمان ایجاد نمی کند.

آزمایش ششم: پیاده سازی Multi-Interrupt

هرف این آزمایش طراحی و شبیه‌سازی یک سیستم چند وقفه‌ای با سه منبع وقفه (**IG1, IG2, IG3**)، یک کنترلر با اولویت ثابت و یک «**CPU**» که هر وقفه را به‌مدت مشخص سرویس می‌دهد؛ سپس بازگشت به «کار عادی» است.

این ماژول با مشخصات زیر طراحی می شود:

- گام شبیه‌سازی: هر فراخوانی تابع = ۱۰ نانوثانیه **(tick)** .
- مولدهای وقفه (**IG1..IG3**): چرخه‌ی ۵۰ نانوثانیه (۵ تیک)؛ اگر در ابتدای چرخه فعال شوند ⇒ ۲۰ نانوثانیه **ON** (۲ تیک) سپس ۳۰ نانوثانیه **OFF** (۳ تیک).
- کنترلر وقفه: در هر تیک چک می‌کند با اولویت: **IG1 > IG2 > IG3**.
- **CPU**: با دریافت **IRQ** وارد **ISR** متناظر می‌شود و ۳۰ نانوثانیه (۳ تیک) در **ISR** می‌ماند؛ سپس به حالت عادی برمی‌گردد.

همچنین ماژول خروجی های زیر را دارد:

- **irq (0=none, 1=IG1, 2=IG2, 3=IG3)**
- **cpu_state (0=Normal, 1/2/3=ISR(IG1/2/3))**
- خطوط مشاهده‌ی وقفه: **ig1/ig2/ig3**

برای انجام این پروژه دو فایل پیاده سازی و یک فایل تست بنچ درست کردیم: **`multi_intr_hls.h`**، **`multi_intr_hls.cpp`** و **tb_multi_intr_hls.cpp** (تست‌بنچ)

ابتدا فایل هدر را شرح می دهیم:

multi_intr_hls.h:

```
#ifndef MULTI_INTR_HLS_H
#define MULTI_INTR_HLS_H
```

```
#include <ap_int.h>
void multi_intr_hls(bool reset,
                    ap_uint<2> &irq,
                    ap_uint<2> &cpu_state,
                    bool &ig1, bool &ig2, bool &ig3);

#endif
```

در فایل هدر بالا پس از فراخوانی تابع **apt_int.h** برای اطمینان از اندازه ثابت بیت‌ها، تابعی برای ساخت خروجی ها و سه وقفه مورد نیاز، تعریف کردیم:

- ورودی: **reset**
- خروجی‌ها: کد وقفه‌ی انتخاب‌شده (**irq**)، وضعیت **CPU (cpu_state)**، و سه خط مشاهده‌ی وقفه (**ig1..ig3**).

حال فایل سورس را قدم به قدم توضیح می دهیم:

```
#include "multi_intr_hls.h"

static ap_uint<16> lfsr_next(ap_uint<16> s) {
    // x^16 + x^14 + x^13 + x^11 + 1 (maximal-length)
    bool b = s[0] ^ s[2] ^ s[3] ^ s[5];
    return (ap_uint<16>)((s >> 1) | ((ap_uint<16>)b << 15));
}
```

ابتدا هدر ماژول خود را فراخوانی می کنیم.

بعد از آن یک تابع کمکی برای تولید بیت شبه‌تصادفی می سازیم به این صورت که:

از چند بیت انتهایی **s** (تپ‌های ۰،۲،۳،۵) **XOR** می‌گیرد تا بیت فیدبک **b** ساخته شود (چندجمله‌ای حداکثرطول).

سپس رجیستر را یک‌بیت راست شیفت می‌کند و **b** را در بیت ۱۵ می‌نویسد.

در نهایت در تست‌بنچ، این «سکه‌ی ۵۰٪» ما می باشد تا در شروع هر چرخه۵۰ نانوثانیه تصمیم بگیرد وقفه روشن باشد یا نه.

```
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=reset
#pragma HLS INTERFACE ap_none port=irq
#pragma HLS INTERFACE ap_none port=cpu_state
#pragma HLS INTERFACE ap_none port=ig1
#pragma HLS INTERFACE ap_none port=ig2
#pragma HLS INTERFACE ap_none port=ig3
```

با استفاده از پراگما ها ، پروژه را به سمت ساده سازی شده ای می بریم، به این معنا که مثلا ورودی ها و خروجی ما تنها یک سیم معمولی هستند و نیازی به **handshake** و سیگنال های کنترلی ندارند، یا اینکه خود پروژه نیازی به فلگ و سیگنال های شروع و پایان ندارد و زمانی که ورودی داده می شود، عمل می کند.

```
static ap_uint<16> l1 = 0xACE1, l2 = 0xBEEF, l3 = 0xC0DE;
```

seed اولیه برای سه **LFSR** (منابع **IG1..IG3**) تا رفتارشان مستقل باشد.

```
static ap_uint<3>  t1 = 0, t2 = 0, t3 = 0;
```

شمارنده تیک برای هر **IG**؛ هر ۵ تیک = ۵۰**ns** ⟵ وقتی به ۴ رسید و تیک بعدی، صفر می‌شود.

```
  static bool        on1 = false, on2 = false, on3 = false;
```

تصمیم «این چرخه ۵۰ نانوثانیه روشن/خاموش است؟» (سکه‌ی ۵۰٪)؛ فقط در ابتدای چرخه تعیین می‌شود.

```
static bool        s1 = false, s2 = false, s3 = false;
```

خروجی فعلی هر **IG** در همین تیک (نتیجه‌ی تصمیم **\on** و پنجره‌ی **ON=20ns**).

```
static ap_uint<2>  cpu = 0;                 // 0 normal, 1..3 ISR
    static ap_int<3>   isr_ticks = 0;      // remaining ticks (3 ticks = 30ns)
    static ap_uint<2>  current_irq = 0;
```

وضعیت **CPU (Normal/ISR)**، شمارنده باقیمانده خدمت **ISR** (۳ تیک = ۳۰ نانوثانیه)، و **IRQ** فعلیِ در حال سرویس.

حال به شرط ریست می پردازیم:

```
if (reset) {
        l1 = 0xACE1; l2 = 0xBEEF; l3 = 0xC0DE;
        t1 = t2 = t3 = 0;
        on1 = on2 = on3 = false;
        s1 = s2 = s3 = false;
        cpu = 0; isr_ticks = 0; current_irq = 0;
        irq = 0; cpu_state = 0; ig1 = ig2 = ig3 = false;
        return;
    }
```

با دستور بالا همه وضعیت‌ها و خروجی‌ها به حالت اولیه برمی‌گردند و همین‌جا تابع برای این تیک تمام می‌شود.

حال باید مولد وقفه ها را با الگوی **50ns**؛ **ON=20ns**، **OFF=30ns** درست کنیم:

```
if (t1 == 0) { on1 = (l1 & 1); l1 = lfsr_next(l1); }
    if (t2 == 0) { on2 = (l2 & 1); l2 = lfsr_next(l2); }
    if (t3 == 0) { on3 = (l3 & 1); l3 = lfsr_next(l3); }
```

ابتدای هر چرخه‌ی ۵۰ نانوثانیه با **LFSR** تصمیم می‌گیریم این چرخه **ON** باشد یا نه (احتمال حدوداً ۵۰٪).

```
  s1 = on1 && (t1 < 2);
    s2 = on2 && (t2 < 2);
```

```
    s3 = on3 && (t3 < 2);
```

اگر چرخه جاری «روشن» باشد، فقط در دو تیک اول (۲۰ نانوثانیه) خروجی روی ۱ می‌رود؛ بعدش ۳ تیک خاموش.

```
t1 = (t1 == 4) ? (ap_uint<3>)0 : (ap_uint<3>)(t1 + 1);
    t2 = (t2 == 4) ? (ap_uint<3>)0 : (ap_uint<3>)(t2 + 1);
    t3 = (t3 == 4) ? (ap_uint<3>)0 : (ap_uint<3>)(t3 + 1);
```

افزایش شمارنده تیک‌ها؛ بعد از ۴ ⟶ صفر (چرخه ۵ تیکی = ۵۰ نانوثانیه).

حال به ساخت اولویت وقفه ها می پردازیم:

```
ap_uint<2> prio_irq = 0;
    if (s1)      prio_irq = 1;
    else if (s2) prio_irq = 2;
    else if (s3) prio_irq = 3;
```

اگر چند **IG** هم‌زمان **ON** باشند، **IG1** مقدم است، سپس **IG2**، بعد **IG3**. اگر هیچ‌کدام **ON** نباشند، صفر.

```
if (isr_ticks > 0) {
        isr_ticks = (ap_int<3>)(isr_ticks - 1);
        if (isr_ticks == 0) {
            cpu = 0;
            current_irq = 0;
        }
    } else {
        if (prio_irq != 0) {
            cpu = prio_irq;
            current_irq = prio_irq;
            isr_ticks = 3; // 30ns service time
        } else {
            cpu = 0;
        }
    }
```

با کد بالا می گوییم اگر **CPU** در حال سرویس است: شمارش معکوس تیک‌ها؛ با رسیدن به صفر، برگشت به **Normal** ولی اگر سرویس نمی‌دهد و **IRQ** تازه وجود دارد؛ ورود به **ISR** متناظر و تنظیم شمارش معکوس ۳ تیک (=۳۰ نانوثانیه) و در نبود **IRQ: Normal**.

```
    // Drive outputs
    irq = prio_irq;
    cpu_state = cpu;
    ig1 = s1; ig2 = s2; ig3 = s3;
}
```

و در نهایت فایل سورس ، مقداردهی پورت‌های خروجی برای این تیک انجام می شود.

حال که کد پیاده سازی کامل این ماژول را توضیح داده ایم، حال به ساخت و توضیح کد تست بنچ می پردازیم:

```
#include <iostream>
#include "multi_intr_hls.h"

int main() {
```

```
    ap_uint<2> irq, cpu;
    bool ig1, ig2, ig3;

    // Reset
    multi_intr_hls(true, irq, cpu, ig1, ig2, ig3);
    std::cout << "[0 ns] RESET\n";
```

ابتدا کتابخوانه و فایل هدر مد نظر را فراخوانی می کنیم و در تابع **main** به تعریف متغیر های مد نظر می پردازیم سپس دستور ریست را اجرا می کنیم.

```
// Run 500 ns -> 50 ticks (10ns per tick)
    ap_uint<2> prev_irq = 0, prev_cpu = 0;
    bool p1=false, p2=false, p3=false;
```

با دستور بالا ذخیره مقادیر قبلی برای چاپ «فقط هنگام تغییر» را انجام می دهیم با این کار خوانایی لاگ بهتر می‌شود.

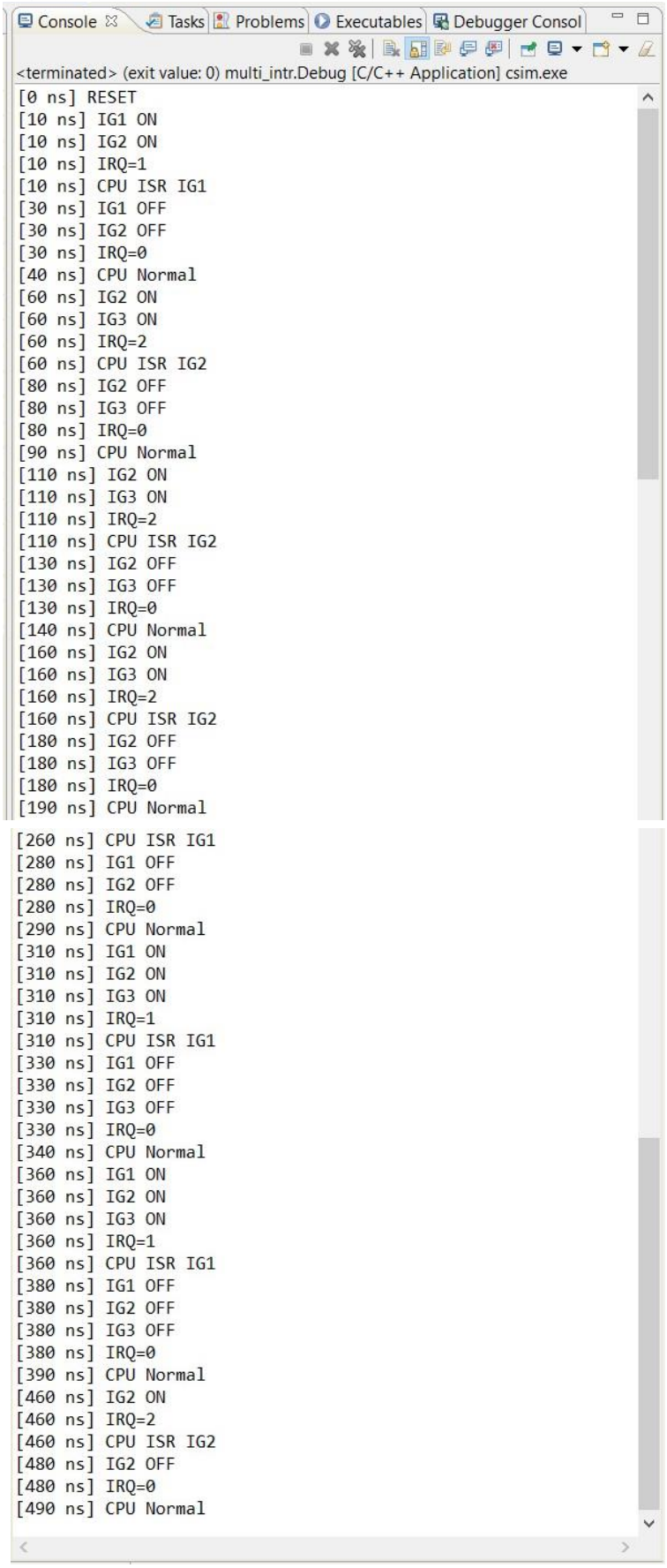
```
for (int t = 1; t <= 50; ++t) {
        multi_intr_hls(false, irq, cpu, ig1, ig2, ig3);
        int time_ns = t * 10;
```

بعد از آن با یک حلقه ۵۰۰ نانوثانیه ای ، ۵۰ تیک را می سازیم.

```
        // Print only on changes to keep log compact
        if (ig1 != p1) { std::cout << "[" << time_ns << " ns] IG1 " << (ig1?
"ON":"OFF") << "\n"; p1 = ig1; }
        if (ig2 != p2) { std::cout << "[" << time_ns << " ns] IG2 " << (ig2?
"ON":"OFF") << "\n"; p2 = ig2; }
        if (ig3 != p3) { std::cout << "[" << time_ns << " ns] IG3 " << (ig3?
"ON":"OFF") << "\n"; p3 = ig3; }
        if (irq != prev_irq) {
            std::cout << "[" << time_ns << " ns] IRQ=" << (unsigned)irq <<
"\n";
            prev_irq = irq;
        }
        if (cpu != prev_cpu) {
            if (cpu == 0) std::cout << "[" << time_ns << " ns] CPU Normal\n";
            else std::cout << "[" << time_ns << " ns] CPU ISR IG" <<
(unsigned)cpu << "\n";
            prev_cpu = cpu;
        }
    }
    return 0;
}
```

در نهایت با اسکریپت بالا ابتدا تغییرات خطوط **IG** را فقط هنگام تغییر چاپ می‌کنیم سپس تغییرات در **IRQ** (نتیجه کنترلر اولویت) ثبت می‌شود و در پایان با دستور شرطی نهایی تغییر وضعیت **CPU** (ورود/خروج **ISR**) لاگ می‌شود.

با ران کردن تست بنچ بالا به نتایج زیر دست پیدا می کنیم:


Console
Tasks
Problems
Executables
Debugger Consol
<terminated> (exit value: 0) multi_intr.Debug [C/C++ Application] csim.exe
[0 ns] RESET
[10 ns] IG1 ON
[10 ns] IG2 ON
[10 ns] IRQ=1
[10 ns] CPU ISR IG1
[30 ns] IG1 OFF
[30 ns] IG2 OFF
[30 ns] IRQ=0
[40 ns] CPU Normal
[60 ns] IG2 ON
[60 ns] IG3 ON
[60 ns] IRQ=2
[60 ns] CPU ISR IG2
[80 ns] IG2 OFF
[80 ns] IG3 OFF
[80 ns] IRQ=0
[90 ns] CPU Normal
[110 ns] IG2 ON
[110 ns] IG3 ON
[110 ns] IRQ=2
[110 ns] CPU ISR IG2
[130 ns] IG2 OFF
[130 ns] IG3 OFF
[130 ns] IRQ=0
[140 ns] CPU Normal
[160 ns] IG2 ON
[160 ns] IG3 ON
[160 ns] IRQ=2
[160 ns] CPU ISR IG2
[180 ns] IG2 OFF
[180 ns] IG3 OFF
[180 ns] IRQ=0
[190 ns] CPU Normal
[260 ns] CPU ISR IG1
[280 ns] IG1 OFF
[280 ns] IG2 OFF
[280 ns] IRQ=0
[290 ns] CPU Normal
[310 ns] IG1 ON
[310 ns] IG2 ON
[310 ns] IG3 ON
[310 ns] IRQ=1
[310 ns] CPU ISR IG1
[330 ns] IG1 OFF
[330 ns] IG2 OFF
[330 ns] IG3 OFF
[330 ns] IRQ=0
[340 ns] CPU Normal
[360 ns] IG1 ON
[360 ns] IG2 ON
[360 ns] IG3 ON
[360 ns] IRQ=1
[360 ns] CPU ISR IG1
[380 ns] IG1 OFF
[380 ns] IG2 OFF
[380 ns] IG3 OFF
[380 ns] IRQ=0
[390 ns] CPU Normal
[460 ns] IG2 ON
[460 ns] IRQ=2
[460 ns] CPU ISR IG2
[480 ns] IG2 OFF
[480 ns] IRQ=0
[490 ns] CPU Normal

با دقت در نتایج اجرا متوجه می شویم که ماژول کاملا صحیح کار می کند زیرا که:

در ۱۰ نانوثانیه و ۳۱۰ نانوثانیه هر سه/دو وقفه هم‌زمان **ON** هستند ولی **IRQ=1** و **CPU** به **ISR IG1** می‌رود پس اولویت **IG1>IG2>IG3** رعایت شده.

شروع **ISR** در ۱۰ نانوثانیه و پایان آن در ۴۰ نانوثانیه (سه تیک) و پیام «**CPU Normal**» دقیقاً در ۴۰ نانوثانیه چاپ شده. همین الگو برای ۶۰⟵۹۰، ۱۱۰⟵۱۴۰، ۱۶۰⟵۱۹۰، ۲۱۰⟵۲۴۰، ۲۶۰⟵۲۹۰، ۳۱۰⟵۳۴۰، ۳۶۰⟵۳۹۰ و ۴۶۰⟵۴۹۰ تکرار شده پس سازگاری زمانی کامل را مشاهده می کنیم.

هر بار که یک **IG** روشن می‌شود، خاموشی آن ۲۰ نانوثانیه بعد گزارش شده (مثلاً **IG1/IG2: 10→30**، **IG2/IG3: 60→80**، **...**) پس مدل **ON=2tick** درست اعمال شده است.

همچنین همیشه بعد از اتمام **ISR**، **IRQ=0** و سپس «**CPU Normal**» گزارش می‌شود پس مسیر بازگشت نیز سالم است.

حال که از درستی نتایج تست بنچ نیز مطمئن شده ایم، نیم نگاهی به گزارش سنتز می اندازیم:

قسمت تایمنیگ مشاهده می کنیم ما قصد داشتیم این ماژول را با کلاک ۱۰۰ مگاهرتزی اجرا کنیم ولی آنقدر ماژول ساده و بهینه طراحی شده است که بیشترین زمانی که برای اجرای عملکرد آن نیاز داریم تنها ۴.۳۹۹ نانو ثانیه است که خیلی سریعتر از ۱۰ میلی ثانیه ای است که ما در نظر گرفتیم و این عالی است.

همچنین در قسمت **Latency**، تاخیر خیلی کم در حد حداقل ۱ و حداکثر ۲ سایکل کلاک داریم که قابل قبول می باشند. مورد دیگر، مصرف منابع خیلی کمی داشتیم و حدود ۴۰۰

```
================================================================
== Performance Estimates
================================================================
+ Timing (ns):
    * Summary:
    +--------+-------+----------+------------+
    |  Clock | Target| Estimated| Uncertainty|
    +--------+-------+----------+------------+
    |ap_clk  |  10.00|     4.399|        1.25|
    +--------+-------+----------+------------+

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |    1|    2|    1|    2|   none  |
    +-----+-----+-----+-----+---------+
```

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-------+--------+-------+-----+
|       Name      | BRAM_18K| DSP48E|   FF   |  LUT  | URAM|
+-----------------+---------+-------+--------+-------+-----+
|DSP              |        -|      -|       -|      -|    -|
|Expression       |        -|      -|       0|    191|    -|
|FIFO             |        -|      -|       -|      -|    -|
|Instance         |        -|      -|       -|      -|    -|
|Memory           |        -|      -|       -|      -|    -|
|Multiplexer      |        -|      -|       -|    201|    -|
|Register         |        -|      -|      70|      -|    -|
+-----------------+---------+-------+--------+-------+-----+
|Total            |        0|      0|      70|    392|    0|
+-----------------+---------+-------+--------+-------+-----+
|Available        |      280|    220|  106400|  53200|    0|
+-----------------+---------+-------+--------+-------+-----+
|Utilization (%)  |        0|      0|   ~0   |   ~0  |    0|
+-----------------+---------+-------+--------+-------+-----+
```

LUT و ۷۰ فلیپ فلاپ استفاده کردیم.

تمامی موارد بالا این نوید را می دهد که بلوک ما خیلی سریع می باشد و تنها پس از کلاک عملکردی، خروجی را تحویل می دهد که این مورد انتظار ما است و مشکلی برایمان ایجاد نمی کند.