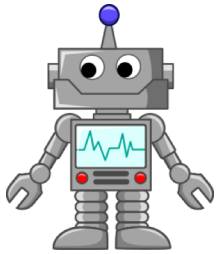
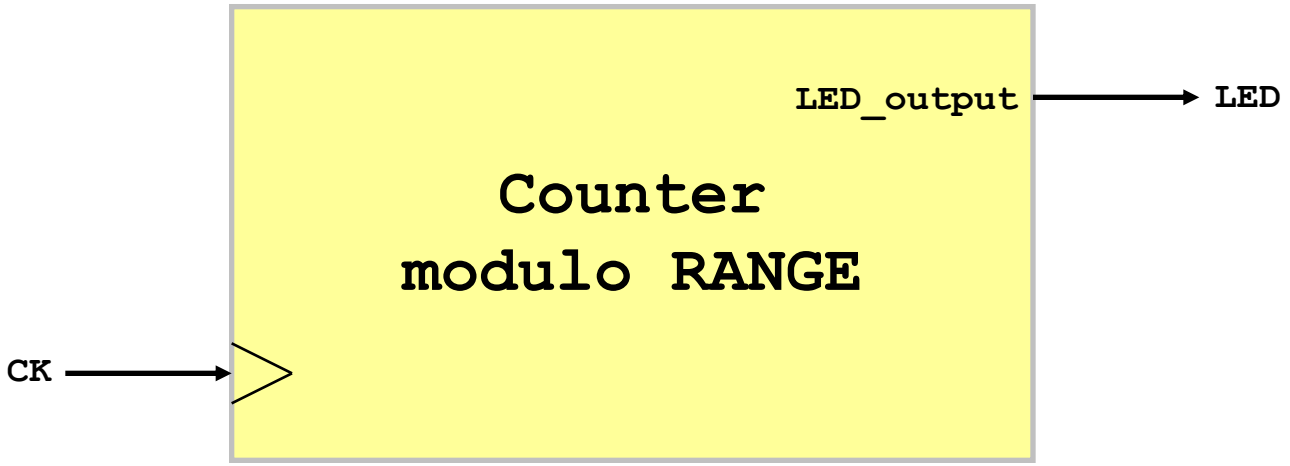
Sistemi Digitali M

06 – High Level Synthesis

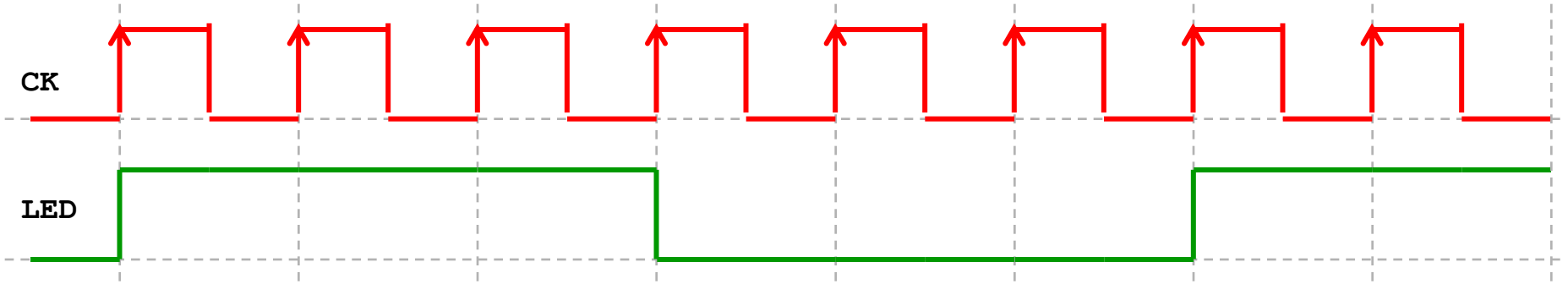


Stefano Mattoccia, DISI - University of Bologna

Progetto HLS di un modulo di blinking



Il modulo deve invertire la propria uscita dopo ogni RANGE periodi di clock. Esempio, se RANGE = 3:



contatore.h

```
#include "ap_int.h"

typedef ap_uint<1> bit;
#define RANGE 1000000
void contatore_no_io(volatile bit *led_output);
```

- **Dichiarare variabili di I/O volatile per evitare ottimizzazioni da parte del compilatore**
- **E' conveniente utilizzare l'estensione .cpp anche per il codice C. In questo modo, mediante "ap_int.h", si possono definire tipi di dati con dimensione variabile**
- **Nell'include è mostrato come definire una variabile di un singolo bit** `ap_uint<1>`
- **E' possibile anche definire tipi signed e unsigned:**
  - `ap_uint<5>` **unsigned a 5 bit**
  - `ap_int<5>` **signed a 5 bit**

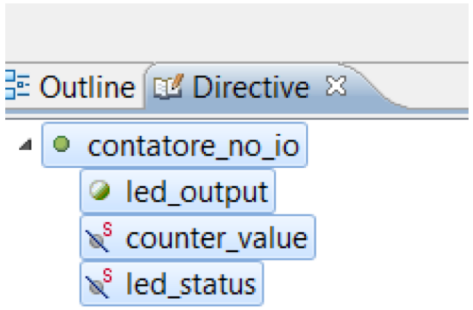
contatore.cpp

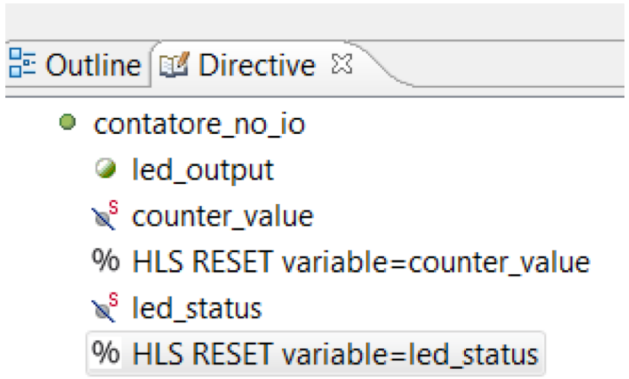
```
#include "contatore.h“

// Contatore modulo RANGE
// ingresso: nessuno, escluso il clock (implicito)
// uscita: un bit, led blinking completato in 2xRANGE clock

void contatore_no_io(volatile bit *led_output)
{
        static unsigned int counter_value = 0;
        static bit led_status = 0;
        counter_value++;

        if (counter_value % RANGE == 0) {
                led_status = not(led_status);
                counter_value=0;
        }

        *led_output = led_status;

        return;
}
```

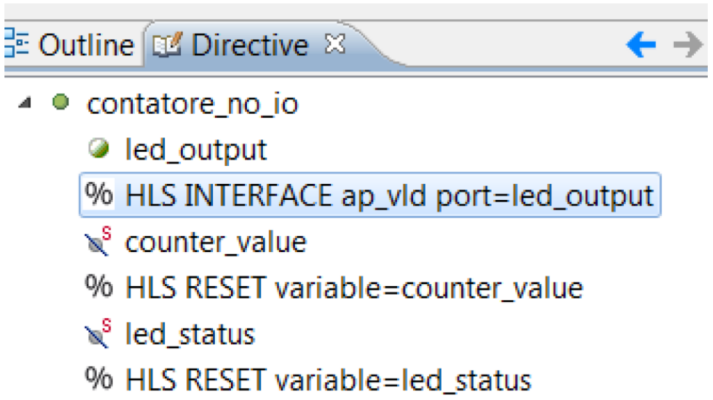
testbench.cpp

```
#include "contatore.h“

int main()
{
	bit led_output_variable=0;

	for (int i=0; i<100; i++)
	{
	contatore_no_io(&led_output_variable);
	printf("Iter %d\t Led = %d\n", i,
		(int)led_output_variable);
	}


	// in un testbench sarebbe mglio verificare se il
	// risultato è corretto (return 0) oppure no (return !=0)

	printf("\n\n>>> End simulation <<<\n\n");

	return 0;
}
```

Senza direttive (default)







Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	contatore_no_io	return value
ap_rst	in	1	ap_ctrl_hs	contatore_no_io	return value
ap_start	in	1	ap_ctrl_hs	contatore_no_io	return value
ap_done	out	1	ap_ctrl_hs	contatore_no_io	return value
ap_idle	out	1	ap_ctrl_hs	contatore_no_io	return value
ap_ready	out	1	ap_ctrl_hs	contatore_no_io	return value
led_output_V	out	1	ap_vld	led_output_V	pointer
led_output_V_ap_vld	out	1	ap_vld	led_output_V	pointer

← ?

? ? ? ? ? ?

← ?

Interfacce per porte 1/3

- ***ap_none*** – This is the simplest protocol type, with no explicit interface protocol, no additional control signals, and no associated hardware overhead. However, there is an implication that timing of input and output operations is independently and correctly handled.

- ***ap_stable*** – This is a similar protocol to *ap_none*, in that it does not involve additional control signals or related hardware. The difference is that *ap_stable* is intended for inputs (only) that change infrequently, i.e. that are generally stable apart from at reset, such as configuration data. The inputs are not constants, but neither do they require to be registered.

- ***ap_ack*** – This protocol behaves differently for input and output ports. For inputs, an output acknowledge port is added, and held high on the same clock cycle as the input is read. For outputs ports, an input acknowledge port is added. After every write to the output port, the design must wait for the input acknowledge to be asserted before it may resume operation.

- ***ap_vld*** – An additional port is provided to validate data. For input ports, a *valid*input control port is added, which qualifies input data as valid. For output ports, a *valid* output port is added, and asserted on clock cycles when output data is valid.

Tratto da: The Zynq Book

Interfacce per porte 2/3

- **ap_ovld** – This protocol is the same as ap_vld, but can only be implemented on output ports, or the output portion of an inout (bidirectional) port.

- ***ap_hs*** – The _hs suffix of this protocol stands for 'handshaking', and it is a superset of *ap_ack*, *ap_vld*, and *ap_ovld*. The *ap_hs* protocol can be used for both input and output ports, and facilitates a two-way handshaking process between the producer and consumer of data, including both validation and acknowledgement transactions. As such, it requires two control ports and associated overhead. It is, however, a robust method of passing data, with no need to ensure timing externally.

- ***ap_memory*** – This memory-based protocol supports random access transactions with a memory, and can be used for both input, output, and bidirectional ports. The only argument type compatible with this protocol is the array type, which corresponds with the structure of a memory. The *ap_memory* protocol requires control signals for clock and write enables, as well as an address port.

- ***bram*** – The same as *ap_memory*, except that when bundled using IP Integrator, the ports are not shown as individual ports, but grouped together into a single port.

Interfacce per porte 3/3

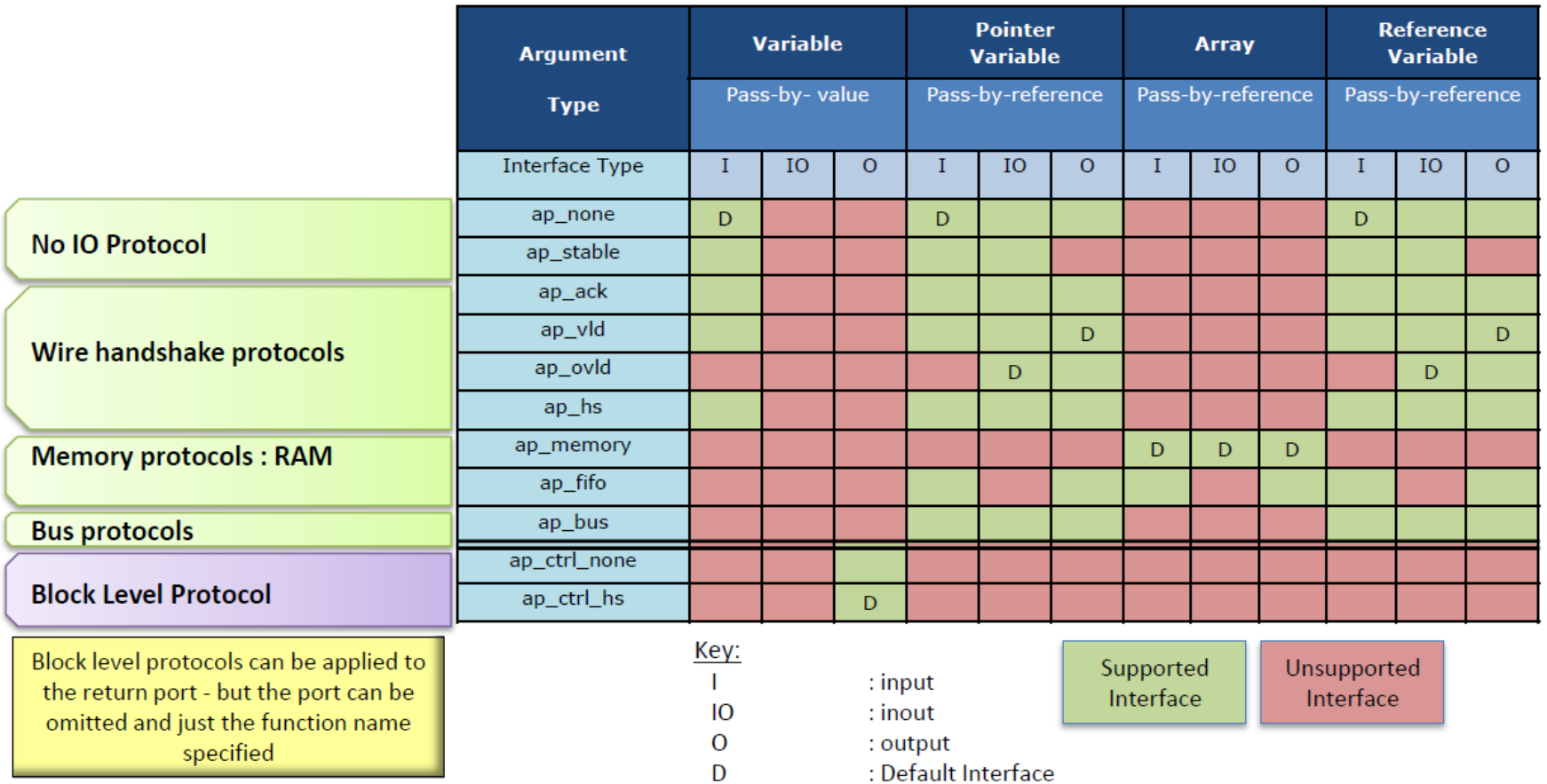
•***ap_fifo*** – The FIFO protocol is also compatible with array arguments, provided that they are accessed sequentially rather than in random order. It does not require any address information to be generated, and therefore is simpler in implementation than the *ap_memory* interface. The *ap_fifo* protocol can be used for input and output ports, but not bidirectional ports. The associated control ports indicate the fullness or emptiness of the FIFO, depending on the port direction, and ensure that processing is stalled to prevent overrun or underrun.

•***ap_bus*** – The *ap_bus* protocol is a generic bus interface that is not tied to a specific bus standard, and may be used to communicate with a bus bridge, which can then arbitrate with a system bus. The *ap_bus* protocol supports single read operations, single write operations, and burst transfers, and these are coordinated using a set of control signals. In addition to this generic bus interface, specific support for AXI bus interfaces can be integrated at a later stage, using an interface synthesis directive.

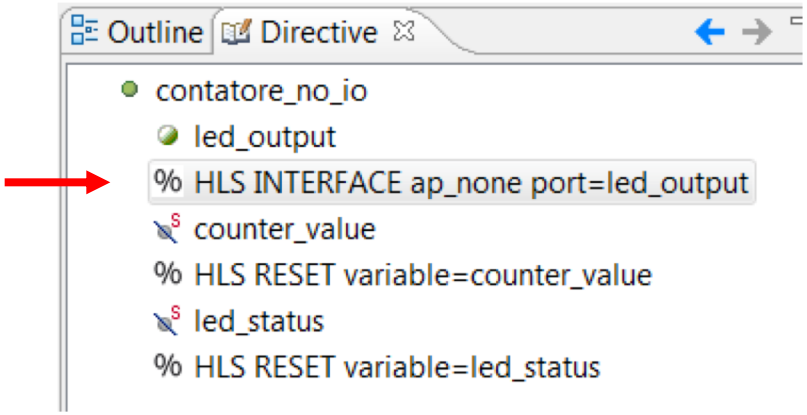
•***axis*** – This specifies the interface as AXI stream.

•***s_axilite*** – This specifies the interface as AXI Slave Lite

•***m_axi*** – This specifies the interface the AXI Master protocol


No IO Protocol
Wire handshake protocols
Memory protocols : RAM
Bus protocols
Block Level Protocol
Block level protocols can be applied to the return port - but the port can be omitted and just the function name specified
Argument Type
Variable
Pass-by- value
Pointer Variable
Pass-by-reference
Array
Pass-by-reference
Reference Variable
Pass-by-reference
Interface Type
I IO O I IO O I IO O I IO O
ap_none D D D
ap_stable
ap_ack
ap_vld D D
ap_ovld D D D
ap_hs
ap_memory D D D
ap_fifo
ap_bus
ap_ctrl_none
ap_ctrl_hs D
Key:
I : input
IO : inout
O : output
D : Default Interface
Supported Interface
Unsupported Interface

Con direttiva ap_none per l'uscita (LED)



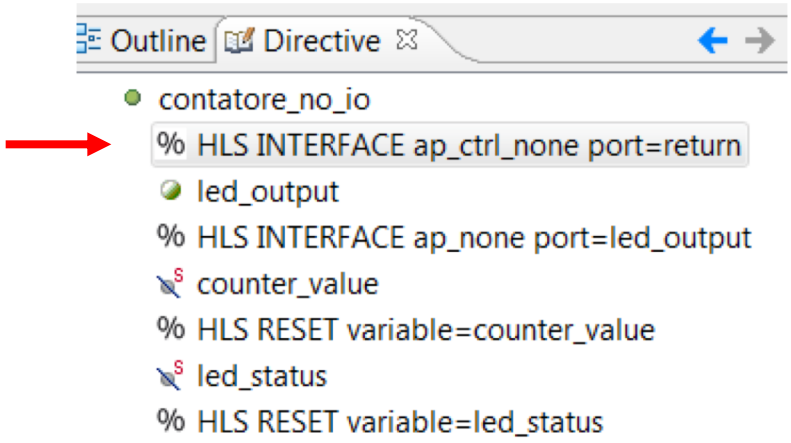
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	contatore_no_io	return value
ap_rst	in	1	ap_ctrl_hs	contatore_no_io	return value
ap_start	in	1	ap_ctrl_hs	contatore_no_io	return value
ap_done	out	1	ap_ctrl_hs	contatore_no_io	return value
ap_idle	out	1	ap_ctrl_hs	contatore_no_io	return value
ap_ready	out	1	ap_ctrl_hs	contatore_no_io	return value
led_output_V	out	1	ap_none	led_output_V	pointer

handshake

Con direttiva ap_ctrl_none per top function



Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_none	contatore_no_io	return value
ap_rst	in	1	ap_ctrl_none	contatore_no_io	return value
led_output_V	out	1	ap_none	led_output_V	pointer

Report: Latency e Interval

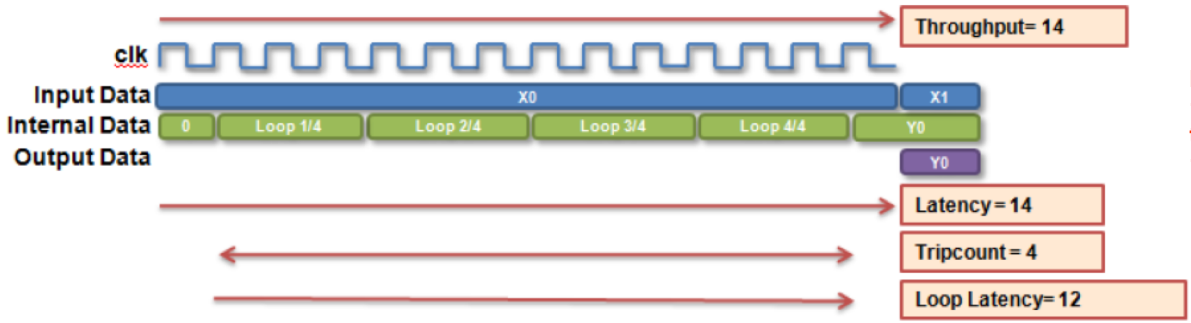
Latency: è il numero di cicli di clock necessari al modulo per generare il risultato a fronte di un nuovo input

Throughput/Interval (II or Initiation Interval): numero di cicli di clock necessari prima di poter elaborare un nuovo dato

Latency e Throughput/Interval non sono la stessa cosa!

Si pensi al DLX: nel caso del DLX sequenziale Latency e Interval sono coincidenti mentre nel caso del DLX pipelined la Latency è 5 clock (senza stalli) mentre l'Interval è pari a 1. Simili metodologie si applicano per rendere i moduli HLS più efficienti mediante delle opportune direttive.

Vivado HLS Terminology for measuring in Clock Cycles



Loop:
1 clock per entrare,
tripcount*N_iterations,
1 clock per uscire

Latency	The number of cycles from input to output (final output of an array write)	14 cycles
Throughput	The number of cycle between new input samples (in this example it must wait for all operations to complete before it can read a new input)	14 cycles
Initiation Interval (II)	The number of cycles between new inputs to a pipeline (the same as throughput, but this term is used with pipelines).	Not shown in this example.
Data Rate	The 1/throughout * clock frequency	10ns clock => 7.14 Mhz, ((1/10e9)*14)
Trip count	The number of iterations in a loop	4
Loop Latency	The latency of the entire loop (divide by tripcount to get the latency for each loop iteration)	12 cycles

You may have your own terminology: this is AutoESL's

```c
#include "contatore.h“

// Contatore modulo RANGE
void contatore_no_io(volatile bit *led_output)
{
          static unsigned int counter_value = 0;
          static bit led_status = 0;
          counter_value++;

          if (counter_value % RANGE == 0) {
                    led_status = not(led_status);
                    counter_value=0;
          }

          *led_output = led_status;
          return;
}
```

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	60
FIFO	-	-	-	-
Instance	-	-	352	416
Memory	-	-	-	-
Multiplexer	-	-	-	66
Register	-	-	41	-
Total	0	0	393	542
Available	280	220	106400	53200
Utilization (%)	0	0	~0	1

```c
#include "contatore.h“

// Contatore modulo RANGE
void contatore_no_io(volatile bit *led_output)
{
          static unsigned int counter_value = 0;
          static bit led_status = 0;
          counter_value++;

          if (counter_value >= RANGE)
          {    led_status = not(led_status);
               counter_value = 0;
          }

          *led_output = led_status;
          return;
}
```

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	74
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	32
Register	-	-	35	-
Total	0	0	35	106
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Contatore con modulo

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	8.56	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1	36	2	37	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	64
FIFO	-	-	-	-
Instance	-	-	256	304
Memory	-	-	-	-
Multiplexer	-	-	-	133
Register	-	-	107	-
Total	0	0	363	501
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Contatore senza modulo

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
default	10.00	6.53	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
0	0	1	1	none

Detail

Instance

Loop

Utilization Estimates

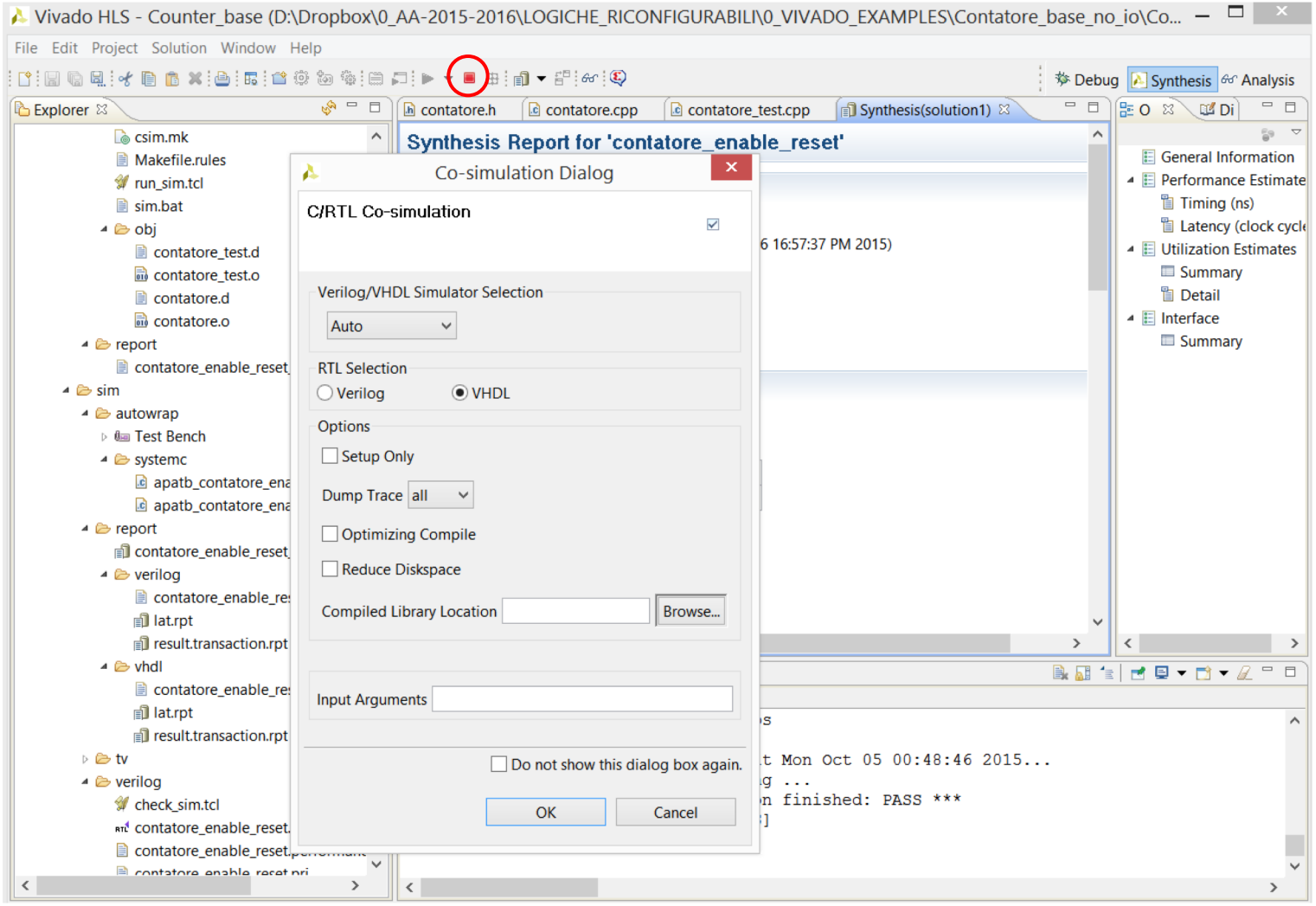
Summary

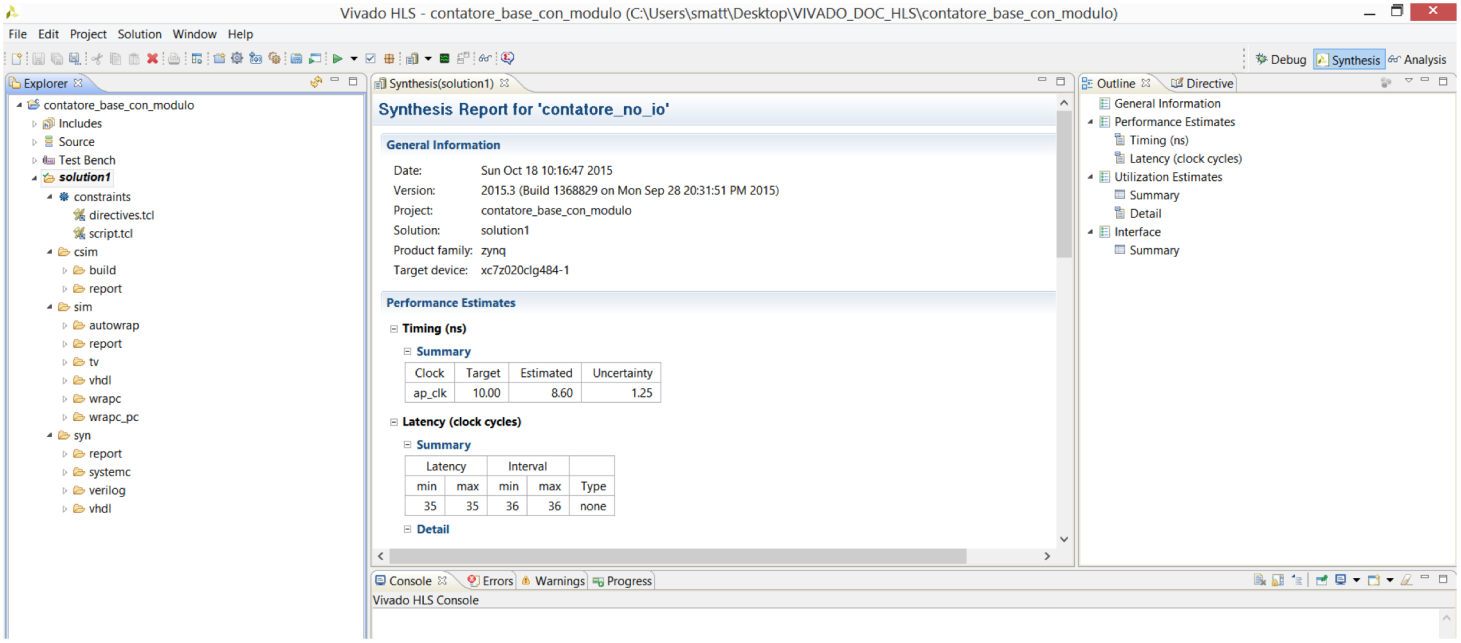
Name	BRAM_18K	DSP48E	FF	LUT
Expression	-	-	0	74
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	34
Register	-	-	34	-
Total	0	0	34	108
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

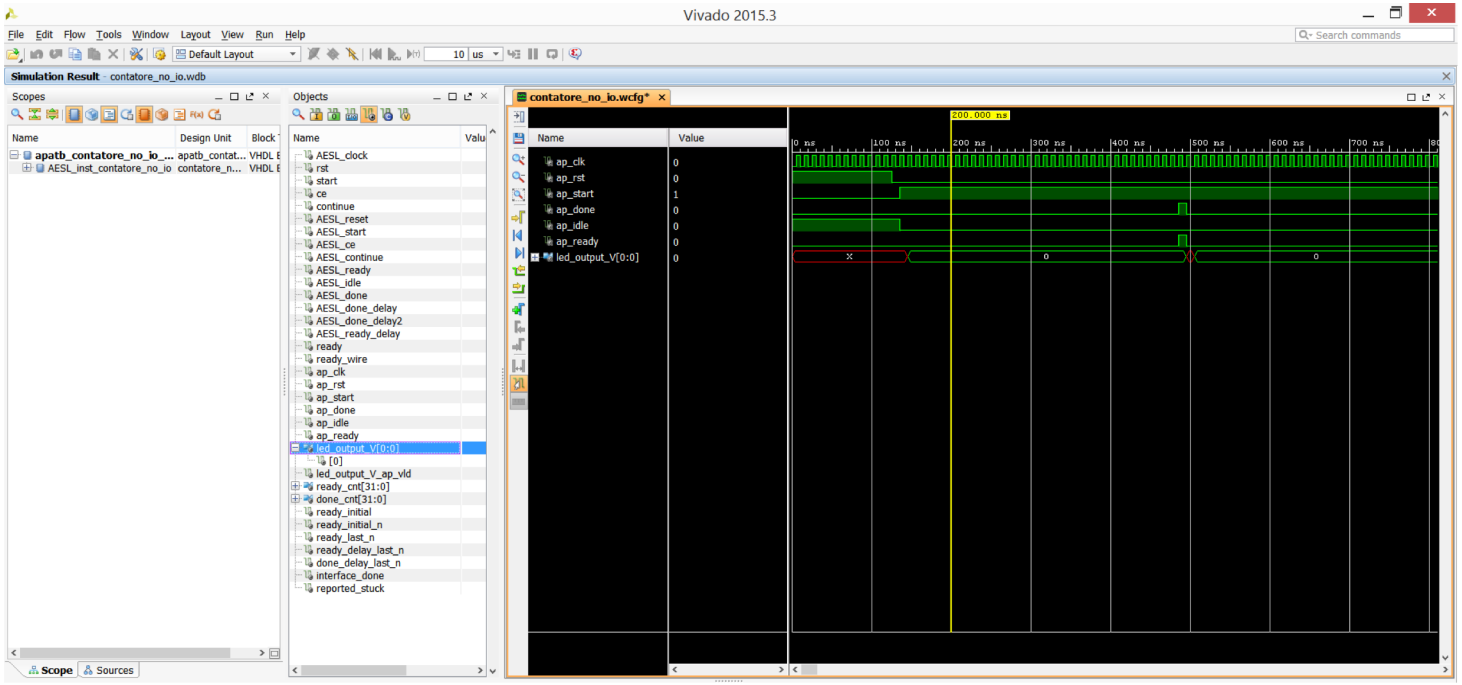
Detail

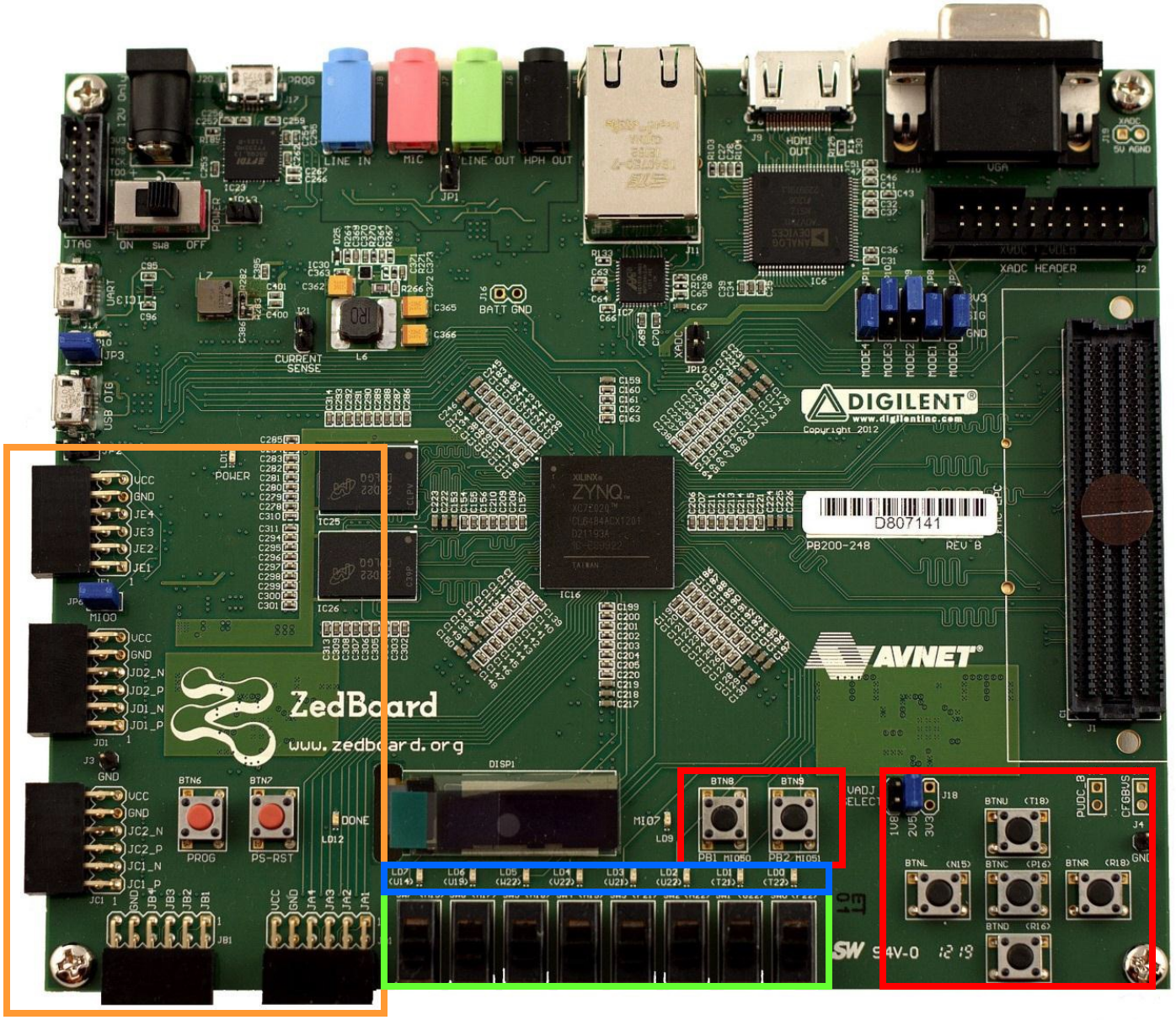
Cosimulazione e forme d'onda 1/2

Dal testbench in C/C++ e l'output della sintesi è possibile esaminare le forme d'onda.




Vivado HLS - contatore_base_con_modulo (C:\Users\smatt\Desktop\VIVADO_DOC_HLS\contatore_base_con_modulo)
File Edit Project Solution Window Help
Debug Synthesis Analysis
Explorer
contatore_base_con_modulo
Includes
Source
Test Bench
solution1
constraints
directives.tcl
script.tcl
csim
build
report
sim
autowrap
report
tv
vhdl
wrapc
wrapc_pc
syn
report
systemc
verilog
vhdl
Synthesis(solution1)
Synthesis Report for 'contatore_no_io'
General Information
Date: Sun Oct 18 10:16:47 2015
Version: 2015.3 (Build 1368829 on Mon Sep 28 20:31:51 PM 2015)
Project: contatore_base_con_modulo
Solution: solution1
Product family: zynq
Target device: xc7z020clg484-1
Performance Estimates
Timing (ns)
Summary
Clock Target Estimated Uncertainty
ap_clk 10.00 8.60 1.25
Latency (clock cycles)
Summary
Latency Interval
min max min max Type
35 35 36 36 none
Detail
Outline
Directive
General Information
Performance Estimates
Timing (ns)
Latency (clock cycles)
Utilization Estimates
Summary
Detail
Interface
Summary
Console
Errors
Warnings
Progress
Vivado HLS Console


Vivado 2015.3
File Edit Flow Tools Window Layout View Run Help
Simulation Result - contatore_no_io.wdb
Scopes
Objects
contatore_no_io.wcfg*
Name
Design Unit
Block
apatb_contatore_no_io_...
apatb_contat...
VHDL E
AESL_inst_contatore_no_io
contatore_n...
VHDL E
AESL_clock
rst
start
ce
continue
AESL_reset
AESL_start
AESL_ce
AESL_continue
AESL_ready
AESL_idle
AESL_done
AESL_done_delay
AESL_done_delay2
AESL_ready_delay
ready
ready_wire
ap_clk
ap_rst
ap_start
ap_done
ap_idle
ap_ready
led_output_V[0:0]
[0]
led_output_V_ap_vld
ready_cnt[31:0]
done_cnt[31:0]
ready_initial
ready_initial_n
ready_last_n
ready_delay_last_n
done_delay_last_n
interface_done
reported_stuck
Value
ap_clk 0
ap_rst 0
ap_start 1
ap_done 0
ap_idle 0
ap_ready 0
led_output_V[0:0] 0
200.000 ns
0 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
700 ns
Scope
Sources


ZedBoard
www.zedboard.org
DIGILENT
www.digilentinc.com
Copyright 2012
AVNET
ZYNQ
XC7Z020
D807141
PB200-248
REV B
POWER
PROG
PS-RST
DONE
DISP1
BTN6
BTN7
BTN8
BTN9
PB1 MIO50
PB2 MIO51
BTNU (T18)
BTNL (N15)
BTNC (P16)
BTNR (R18)
BTND (R16)
LD7 (U14)
LD6 (U19)
LD5 (W22)
LD4 (V22)
LD3 (U21)
LD2 (U22)
LD1 (T21)
LD0 (T22)
LINE IN
MIC
LINE OUT
HPH OUT
HDMI OUT
VGA
XADC HEADER
JTAG
USB OTG
UART
BATT GND
CURRENT SENSE

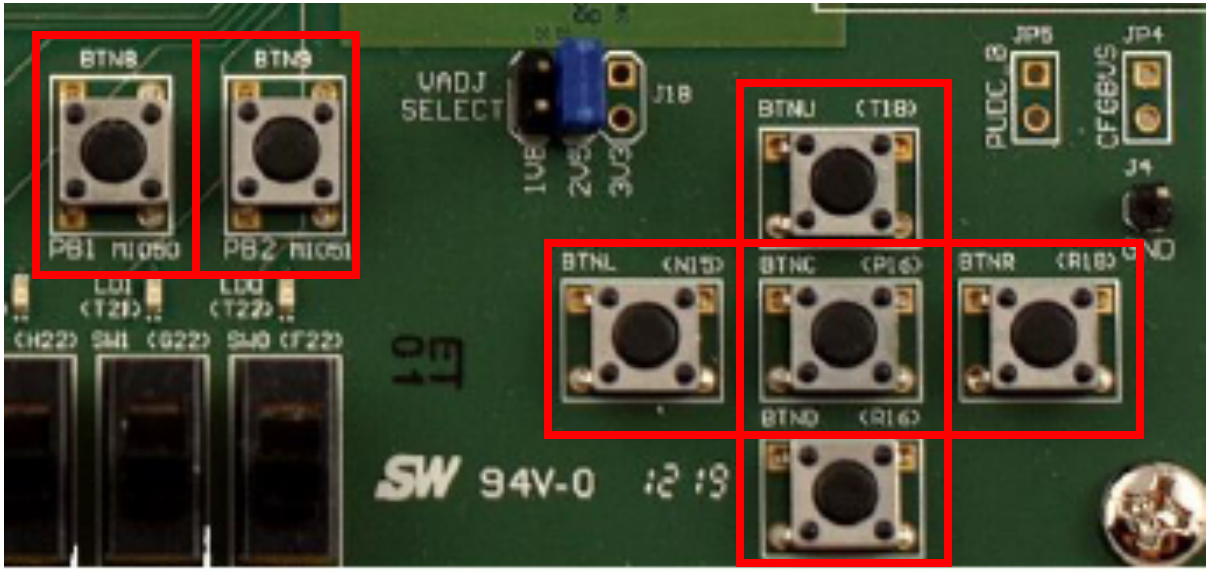


Table 12 - Push Button Connections

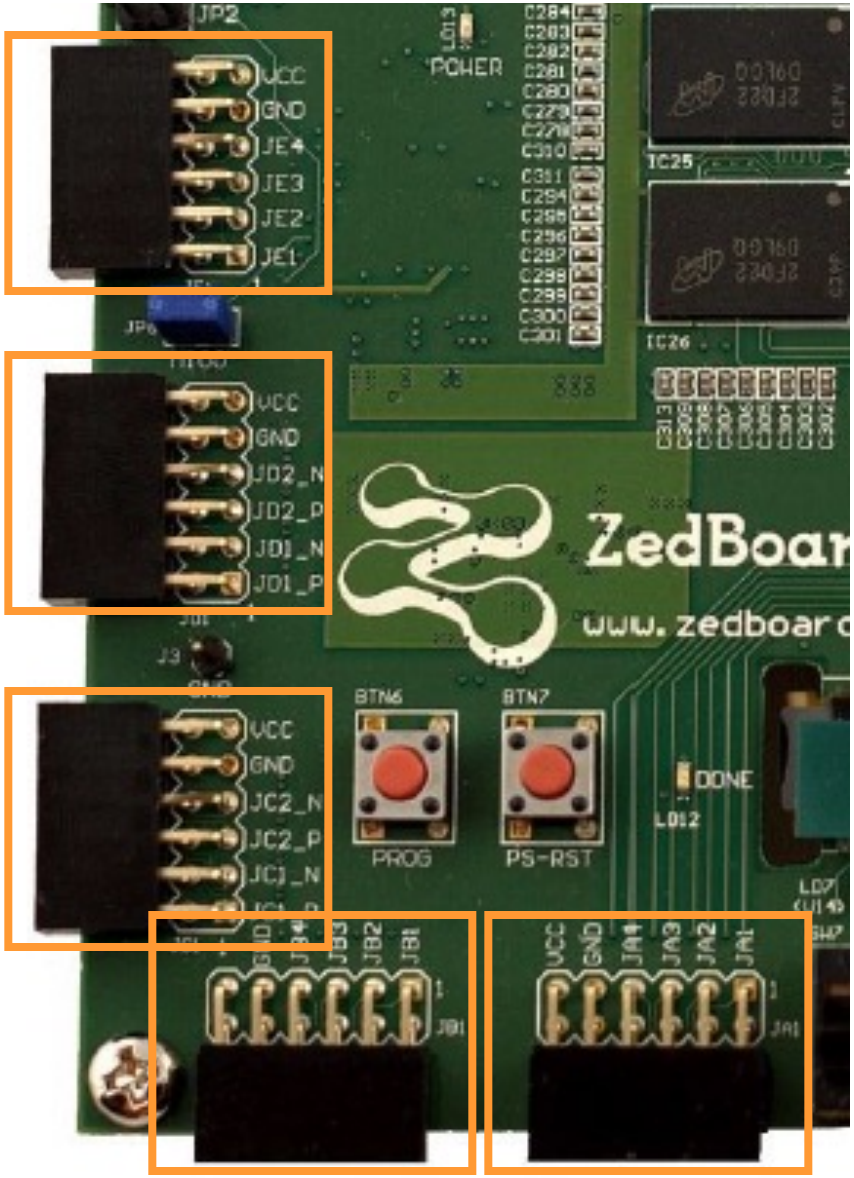
Signal Name	Subsection	Zynq EPP pin
BTNU	PL	T18
BTNR	PL	R18
BTND	PL	R16
BTNC	PL	P16
BTNL	PL	N15
PB1	PS	D13 (MIO 50)
PB2	PS	C10 (MIO 51)

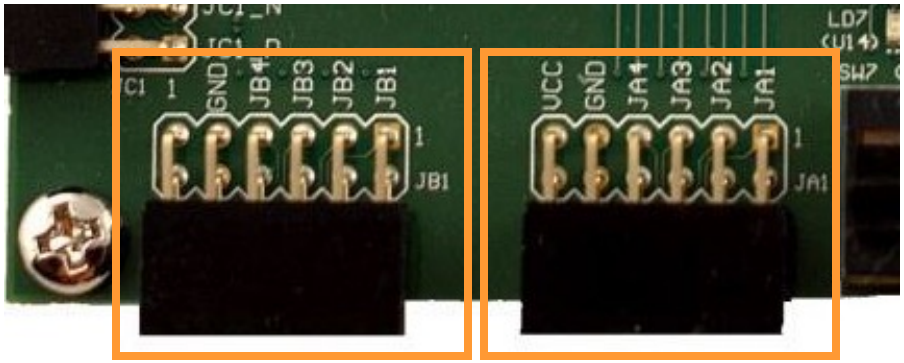
Table 14 - LED Connections

Signal Name	Zynq EPP pin
LD0	T22
LD1	T21
LD2	U22
LD3	U21
LD4	V22
LD5	W22
LD6	U19
LD7	U14

Table 13 - DIP Switch Connections

Signal Name	Zynq EPP pin
SW0	F22
SW1	G22
SW2	H22
SW3	F21
SW4	H19
SW5	H18
SW6	H17
SW7	M15


JP2
VCC
GND
JE4
JE3
JE2
JE1
JP6
VCC
GND
JD2_N
JD2_P
JD1_N
JD1_P
J3
VCC
GND
JC2_N
JC2_P
JC1_N
JB3
JB2
JB1
JB1
VCC
GND
JA4
JA3
JA2
JA1
JA1
POWER
LD13
IC25
IC26
ZedBoar
www.zedboar
BTN6
BTN7
PROG
PS-RST
DONE
LD12



3.3V	GND	JB4	JB3	JB2	JB1
3.3V	GND	JB10	JB9	JB8	JB7

3.3V	GND	JA4	JA3	JA2	JA1
3.3V	GND	JA10	JA9	JA8	JA7

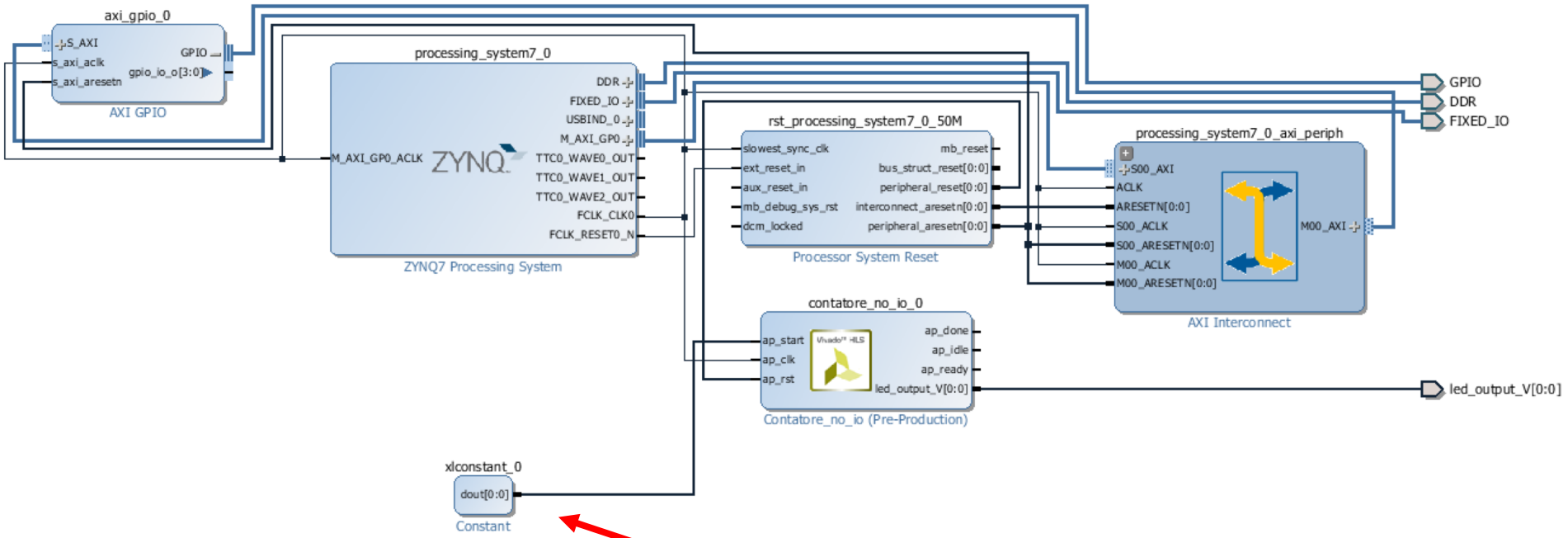
Table 16 - Pmod Connections

Pmod	Signal Name	Zynq EPP pin	Pmod	Signal Name	Zynq EPP pin
JA1	JA1	Y11	JB1	JB1	W12
	JA2	AA11		JB2	W11
	JA3	Y19		JB3	V10
	JA4	AA9		JB4	W8
	JA7	AB11		JB7	V12
	JA8	AB10		JB8	W10
	JA9	AB9		JB9	V9
	JA10	AA8		JB10	V8

Pmod	Signal Name	Zynq EPP pin	Pmod	Signal Name	Zynq EPP pin
JC1 Differential	JC1_N	AB6	JD1 Differential	JD1_N	W7
	JC1_P	AB7		JD1_P	V7
	JC2_N	AA4		JD2_N	V4
	JC2_P	Y4		JD2_P	V5
	JC3_N	T6		JD3_N	W5
	JC3_P	R6		JD3_P	W6
	JC4_N	U4		JD4_N	U5
	JC4_P	T4		JD4_P	U6

Pmod	Signal Name	Zynq EPP pin
JE1 MIO Pmod	JE1	A6
	JE2	G7
	JE3	B4
	JE4	C5
	JE7	G6
	JE8	C4
	JE9	B6
	JE10	E6

Progetto Zynq con IP core del contatore (top function con handshake)



ap_start cablato a 1

- Cablando ap_start a 0, il modulo è bloccato
- L'ARM, in questo caso, è ininfluente

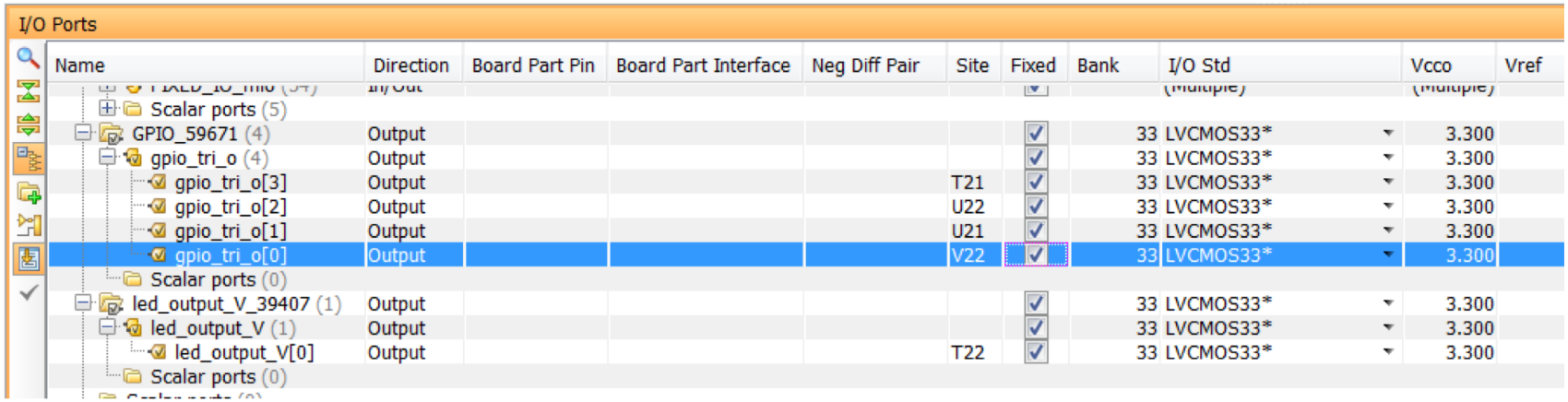
Connessione LED

2.7.3 User LEDs

The ZedBoard has eight user LEDs, LD0 – LD7. A logic high from the Zynq-7000 AP SoC I/O causes the LED to turn on. LED's are sourced from 3.3V banks through 390Ω resistors.

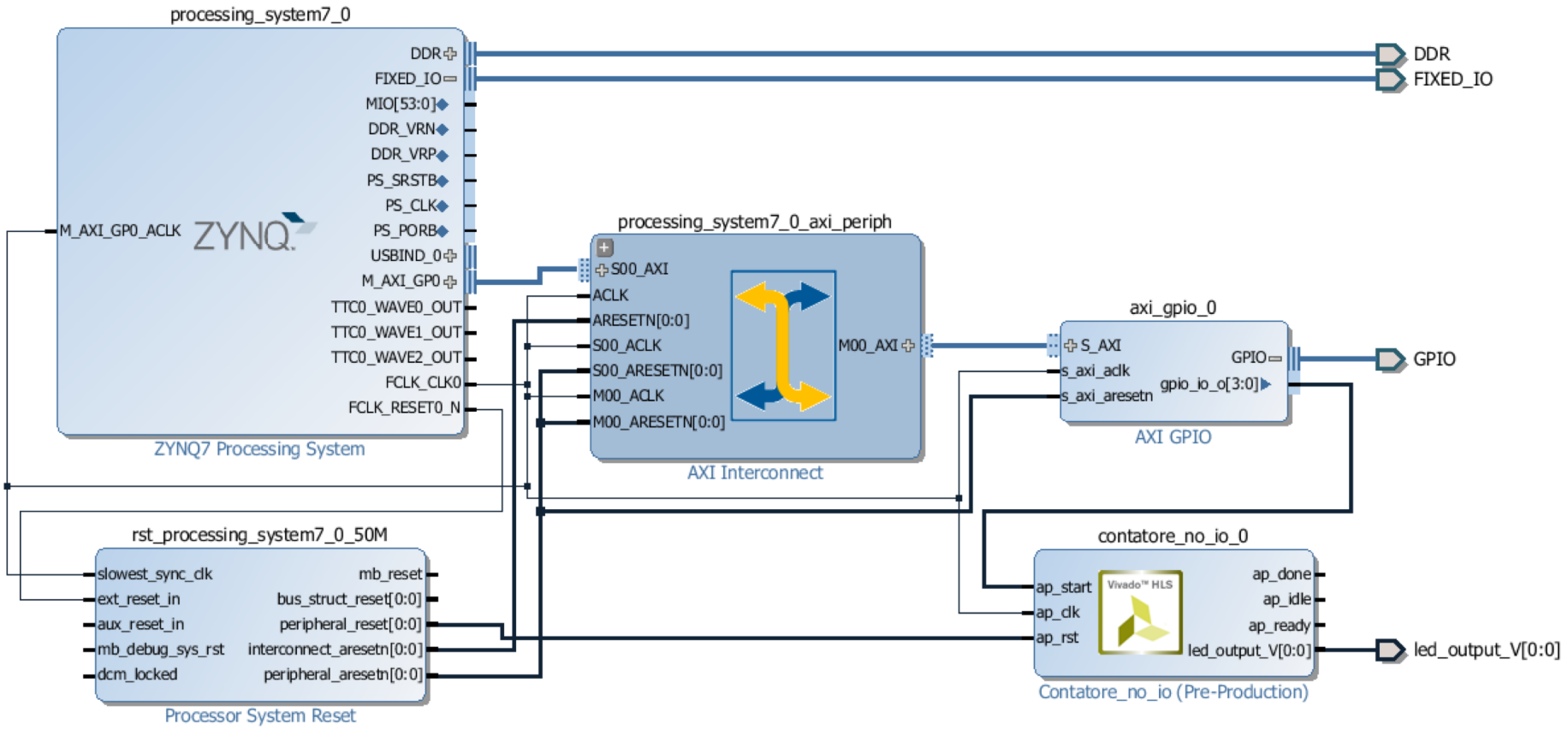
Table 14 - LED Connections

Signal Name	Subsection	Zynq pin
LD0	PL	T22
LD1	PL	T21
LD2	PL	U22
LD3	PL	U21
LD4	PL	V22
LD5	PL	W22
LD6	PL	U19
LD7	PL	U14
LD9	PS	D5 (MIO7)



Controllo di ap_start da ARM via GPIO

L'obiettivo di questo progetto consiste nel controllo del segnale ap_start, del modulo di blinking, mediante comandi software inviati dall'ARM a un modulo GPIO con interfaccia AXI4-light.



Helloworld, controllo ap_start da ARM via GPIO

Lo stralcio di codice seguente, il resto è simile al caso di controllo dei LED via GPIO, accetta un bit da STDIN e lo invia ad ap_start.

```
. . . .

while (1)
{
            printf("Set ap_start to 1 or 0. Please insert value [0,1]: \n");
            char string[6]; // seems REAL_DECIMAL_SIZE + 3 ??
            fgets(string, sizeof(string), stdin);
            input_data_int = atoi(string);

            XGpio_WriteReg(GPIO_REG_BASEADDR, OFFSET_DATA_REGISTER_GPIO_1, input_data_int);
}

 . . . .
```

Mappando il progetto sullo Zynq, si può notare come il conteggio sia bloccato quando ap_start = 0.

Progetto HLS di un modulo configurabile mediante protocollo AXI lite

Mediante direttive (Vivado HLS) è possibile definire:

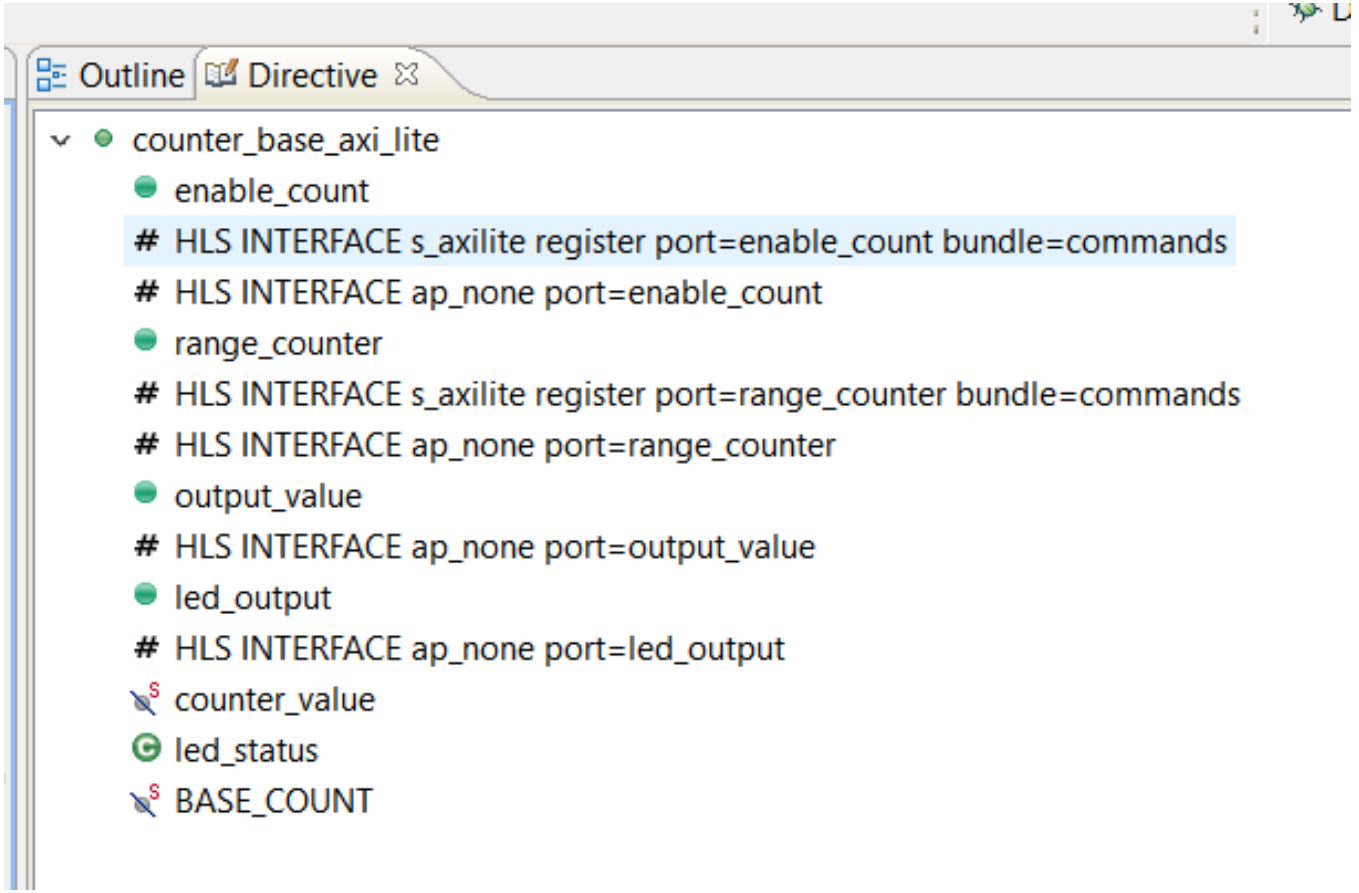
1) Protocollo del modulo/blocco (ap_start, etc)

2) Interfaccia per ciascuna porta di I/O
- **ap_none per il range_counter**
- **ap_none per l'uscita led_output**
- **ap_none per l'uscita output_value**

3) Risorse
- **AXI lite per il range**

counter_base_axi_lite.h

```
#include "ap_int.h"
typedef ap_uint<1> bit;

void counter_base_axi_lite(bit enable_count,
                           volatile unsigned int range_counter,
                           volatile int *output_value,
                           volatile bit *led_output);
```




```
#pragma HLS INTERFACE ap_none port=range_counter
#pragma HLS INTERFACE ap_none port=enable_count
#pragma HLS INTERFACE s_axilite register port=range_counter
bundle=commands
#pragma HLS INTERFACE s_axilite register port=enable_count
bundle=commands
#pragma HLS INTERFACE ap_none port=led_output
#pragma HLS INTERFACE ap_none port=output_value
```

counter_base_axi_lite.cpp

```
#include "counter_base_axi_lite.h"

void counter_base_axi_lite(bit enable_count, volatile unsigned int range_counter,
volatile int *output_value, volatile bit *led_output)
{

static unsigned int counter_value = 0;
static bit led_status = 0;
static unsigned int BASE_COUNT = 5;

if (enable_count == 0)
          BASE_COUNT = range_counter;
else
{         counter_value++;

          if (counter_value >= BASE_COUNT)
          {
                    counter_value = 0;
                    led_status = not(led_status);
          }
}

*led_output = led_status;
*output_value = counter_value;

#ifndef __SYNTHESIS__
printf("Base count = %d\tCounter value = %d\n",BASE_COUNT, counter_value);
#endif

return;

}
```

test_bench.cpp 1/2

```
#include "counter_base_axi_lite.h"

int main()
{
int testbench_error = 0;
int count_value = 0;
bit led_output_variable;
unsigned int RANGE = 10;
bit ENABLE = 0;

for (int i=1; i<20; i++)
{
counter_base_axi_lite(ENABLE, RANGE, &count_value, &led_output_variable);
printf("ENABLE = %d\tRANGE = %d\tConter value = %d\t Led = %d\n",(int)ENABLE, RANGE,
count_value, (int)led_output_variable);
}

ENABLE = 1;

for (int i=1; i<20; i++)
{
counter_base_axi_lite(ENABLE, RANGE, &count_value, &led_output_variable);
printf("ENABLE = %d\tRANGE = %d\tConter value = %d\t Led = %d\n",(int)ENABLE, RANGE,
count_value, (int)led_output_variable);

// cross-check output port count_value
if (count_value != i % RANGE)
testbench_error++;
}
```

test_bench.cpp 2/2

```
if (testbench_error == 0)
{
        printf(">>> C simulation: OK <<<\n");
        return 0;
}
else
{
        printf(">>> C simulation: FAILED <<<\n");
        return -1;
}
}
```

Synthesis(solution1) | *counter_base_axi_lite.h | counter_base_axi_lite.cpp | counter_base_axi_test.cpp

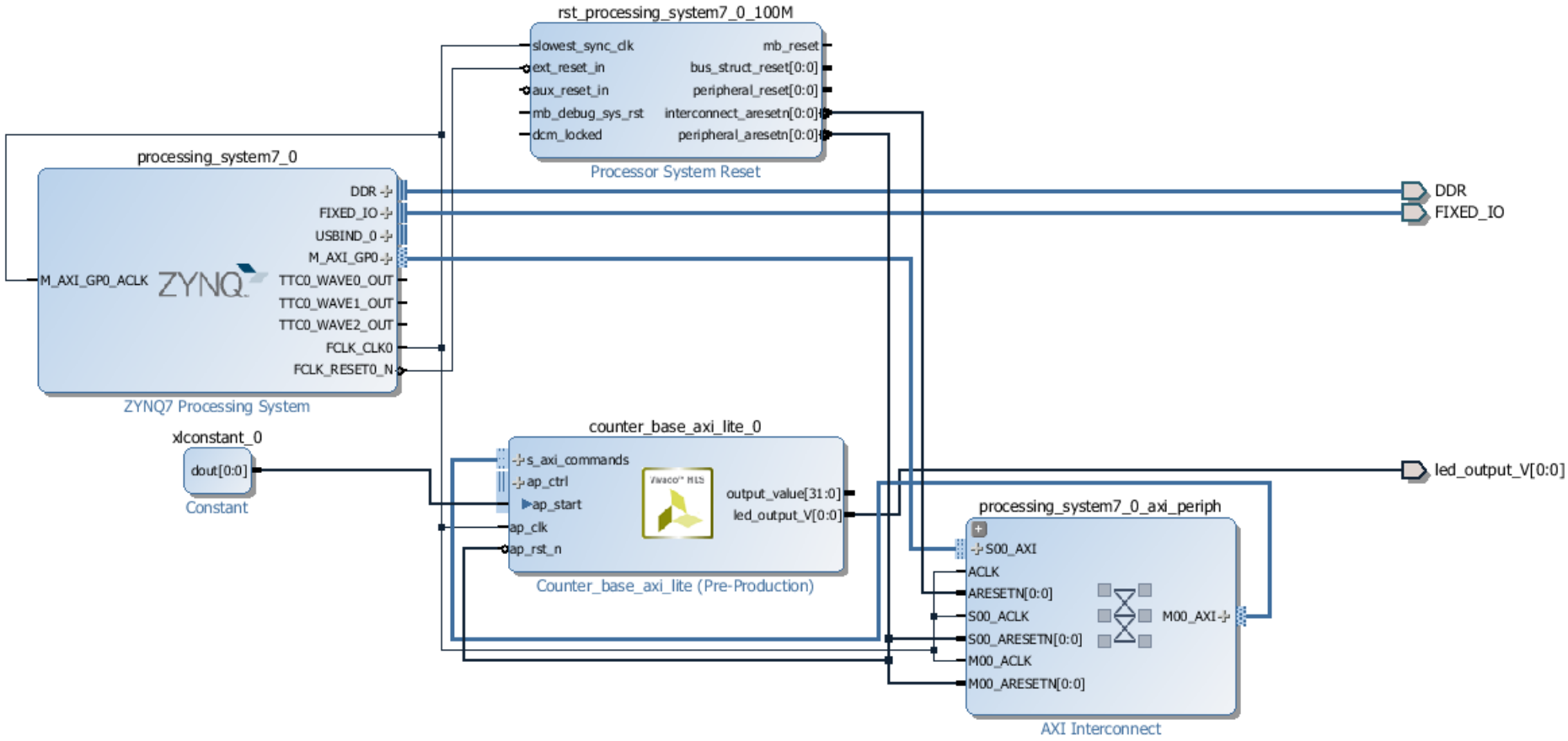
Interface

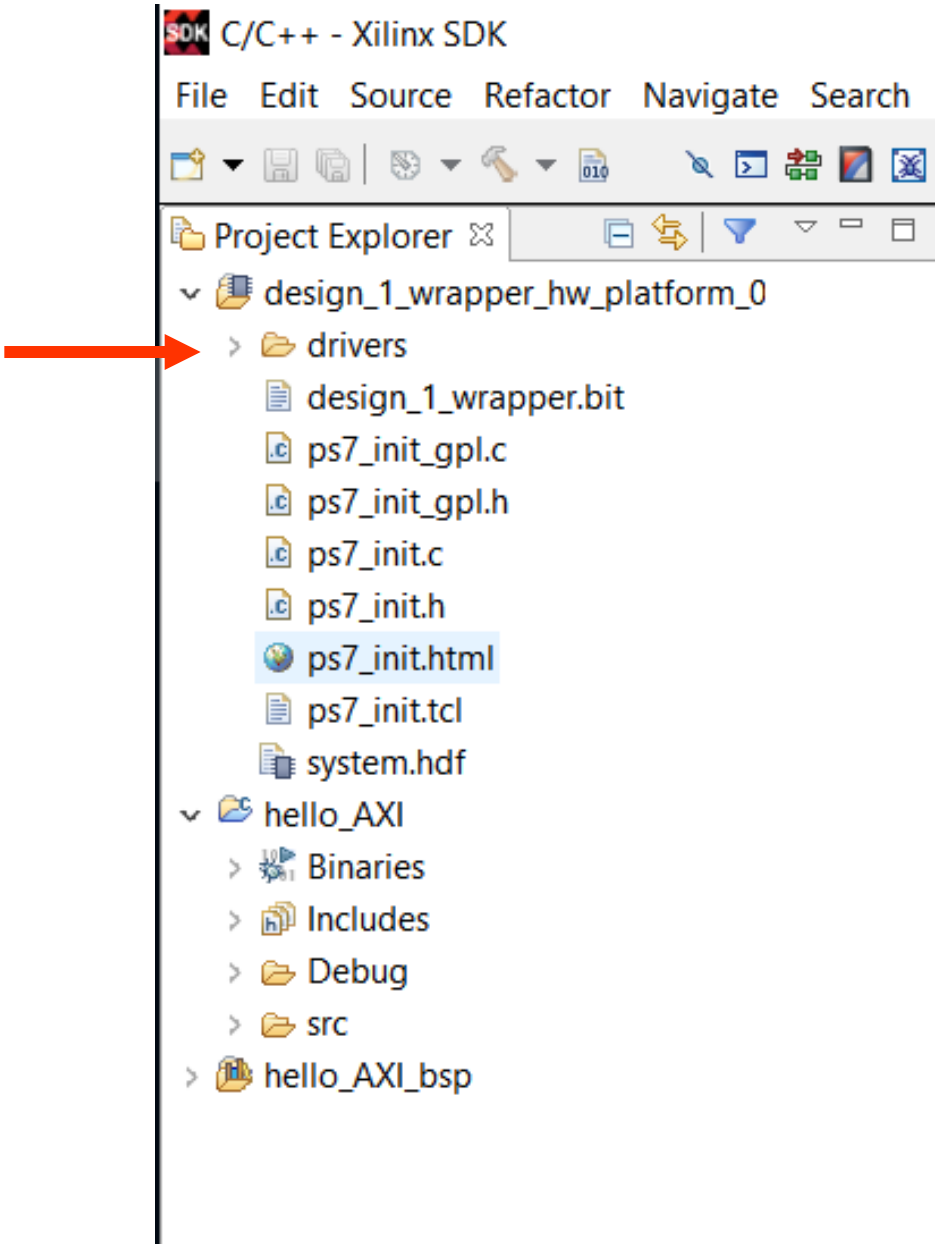
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
s_axi_commands_AWVALID	in	1	s_axi	commands	scalar
s_axi_commands_AWREADY	out	1	s_axi	commands	scalar
s_axi_commands_AWADDR	in	5	s_axi	commands	scalar
s_axi_commands_WVALID	in	1	s_axi	commands	scalar
s_axi_commands_WREADY	out	1	s_axi	commands	scalar
s_axi_commands_WDATA	in	32	s_axi	commands	scalar
s_axi_commands_WSTRB	in	4	s_axi	commands	scalar
s_axi_commands_ARVALID	in	1	s_axi	commands	scalar
s_axi_commands_ARREADY	out	1	s_axi	commands	scalar
s_axi_commands_ARADDR	in	5	s_axi	commands	scalar
s_axi_commands_RVALID	out	1	s_axi	commands	scalar
s_axi_commands_RREADY	in	1	s_axi	commands	scalar
s_axi_commands_RDATA	out	32	s_axi	commands	scalar
s_axi_commands_RRESP	out	2	s_axi	commands	scalar
s_axi_commands_BVALID	out	1	s_axi	commands	scalar
s_axi_commands_BREADY	in	1	s_axi	commands	scalar
s_axi_commands_BRESP	out	2	s_axi	commands	scalar
ap_clk	in	1	ap_ctrl_hs	counter_base_axi_lite	return value
ap_rst_n	in	1	ap_ctrl_hs	counter_base_axi_lite	return value
ap_start	in	1	ap_ctrl_hs	counter_base_axi_lite	return value
ap_done	out	1	ap_ctrl_hs	counter_base_axi_lite	return value
ap_idle	out	1	ap_ctrl_hs	counter_base_axi_lite	return value
ap_ready	out	1	ap_ctrl_hs	counter_base_axi_lite	return value
output_value	out	32	ap_none	output_value	pointer
led_output_V	out	1	ap_none	led_output_V	pointer

Export the report(.html) using the Export Wizard

Open Analysis Perspective Analysis Perspective


rst_processing_system7_0_100M
slowest_sync_clk
ext_reset_in
aux_reset_in
mb_debug_sys_rst
dcm_locked
mb_reset
bus_struct_reset[0:0]
peripheral_reset[0:0]
interconnect_aresetn[0:0]
peripheral_aresetn[0:0]
Processor System Reset
processing_system7_0
DDR
FIXED_IO
USBIND_0
M_AXI_GP0
M_AXI_GP0_ACLK
ZYNQ
TTC0_WAVE0_OUT
TTC0_WAVE1_OUT
TTC0_WAVE2_OUT
FCLK_CLK0
FCLK_RESET0_N
ZYNQ7 Processing System
DDR
FIXED_IO
xlconstant_0
dout[0:0]
Constant
counter_base_axi_lite_0
s_axi_commands
ap_ctrl
ap_start
ap_clk
ap_rst_n
output_value[31:0]
led_output_V[0:0]
Counter_base_axi_lite (Pre-Production)
led_output_V[0:0]
processing_system7_0_axi_periph
S00_AXI
ACLK
ARESETN[0:0]
S00_ACLK
S00_ARESETN[0:0]
M00_ACLK
M00_ARESETN[0:0]
M00_AXI
AXI Interconnect


C/C++ - Xilinx SDK
File Edit Source Refactor Navigate Search
Project Explorer
design_1_wrapper_hw_platform_0
drivers
design_1_wrapper.bit
ps7_init_gpl.c
ps7_init_gpl.h
ps7_init.c
ps7_init.h
ps7_init.html
ps7_init.tcl
system.hdf
hello_AXI
Binaries
Includes
Debug
src
hello_AXI_bsp

helloworld.c 1/2

```c
#include <stdio.h>
#include "platform.h"
#include "xcounter_base_axi_lite.h"
#include <unistd.h>

int main()
{
    init_platform();
    XCounter_base_axi_lite InstancePtr;
    u16 DeviceId = 0;
    XCounter_base_axi_lite_LookupConfig(DeviceId);
    int error = XCounter_base_axi_lite_Initialize(&InstancePtr, DeviceId);

    if (error == 0)
    printf("Custom AXI counter: initialization OK :-)\n");
    else
    printf("Custom AXI counter: initialization failed :-(\n");

    useconds_t sleeping_time_us = 5000000;
    unsigned int RANGE = 0x008FFFFF;

    printf("Disabling counter...\n");
    XCounter_base_axi_lite_Set_enable_count_V(&InstancePtr, 0);
    printf("Setting range...\n");
    XCounter_base_axi_lite_Set_range_counter(&InstancePtr, RANGE);
    printf("Enabling counter with base %d...\n", RANGE);
    XCounter_base_axi_lite_Set_enable_count_V(&InstancePtr, 1);
    printf("Counter up and running\n");
```

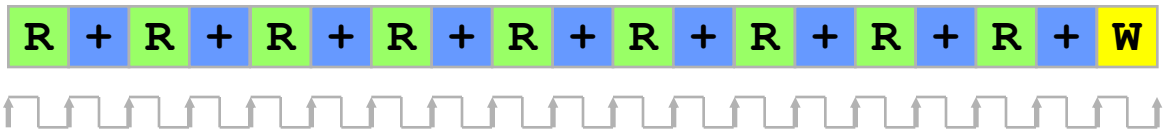
helloworld.c 2/2

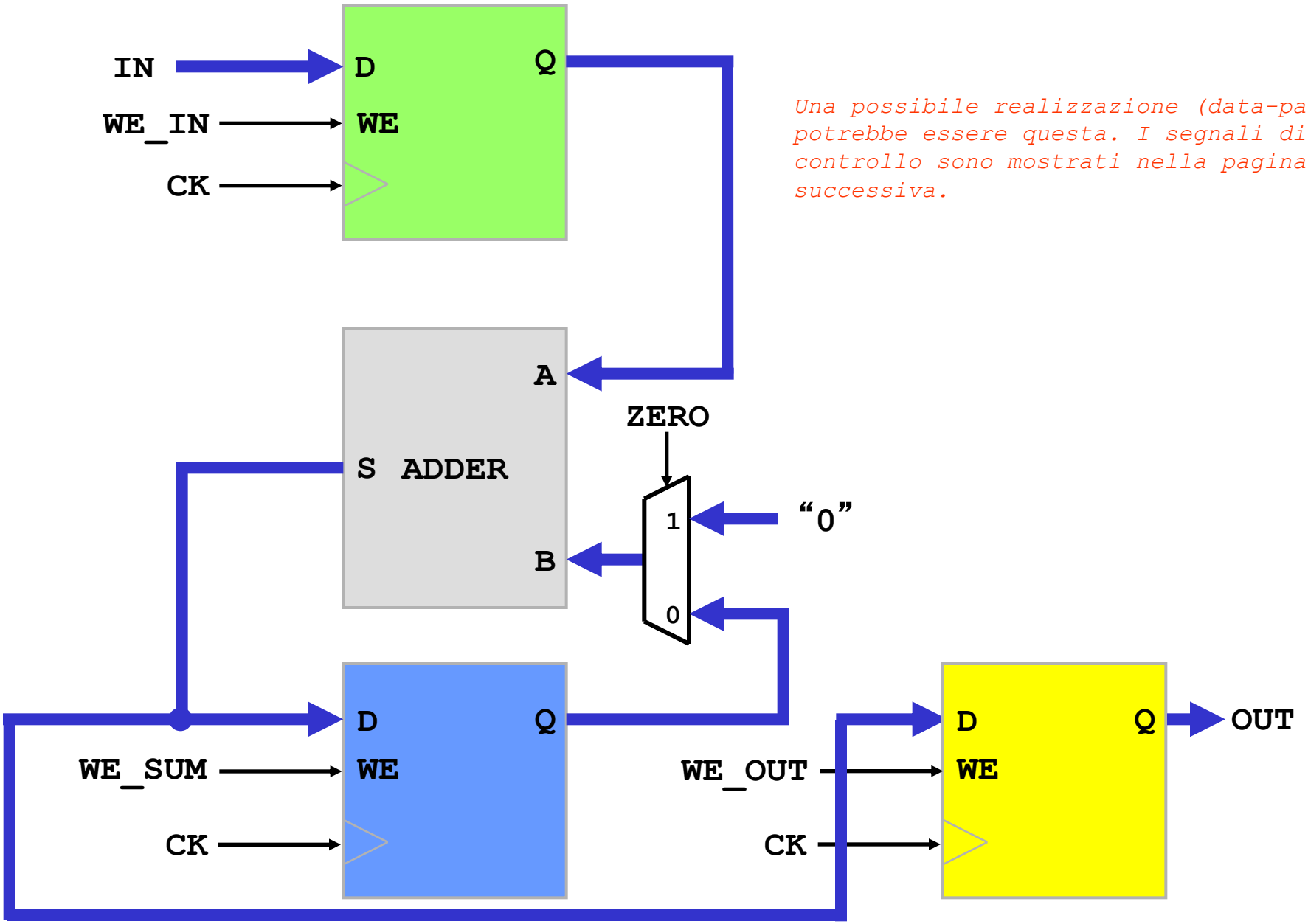
```
printf("Sleeping for %2.1f seconds\n", sleeping_time_us/1000000.0);
    usleep(sleeping_time_us);


    RANGE = 0x001FFFFF;


    printf("Disabling counter...\n");
    XCounter_base_axi_lite_Set_enable_count_V(&InstancePtr, 0);
    printf("Setting range...\n");
    XCounter_base_axi_lite_Set_range_counter(&InstancePtr, RANGE);
    printf("Enabling counter with base %d...\n", RANGE);
    XCounter_base_axi_lite_Set_enable_count_V(&InstancePtr, 1);
    printf("Counter up and running\n");


    cleanup_platform();
    return 0;
}
```

Ottimizzazioni in Vivado HLS

Consideriamo il calcolo della somma degli elementi di un vettore di N elementi.

```
void media(volatile int input_array[LENGTH], volatile int
*average_value)
{
int temp_sum=0;

for (int i=0; i<LENGTH; i++)
{
        temp_sum = temp_sum + input_array[i];
}

*average_value = temp_sum;

return;
}
```

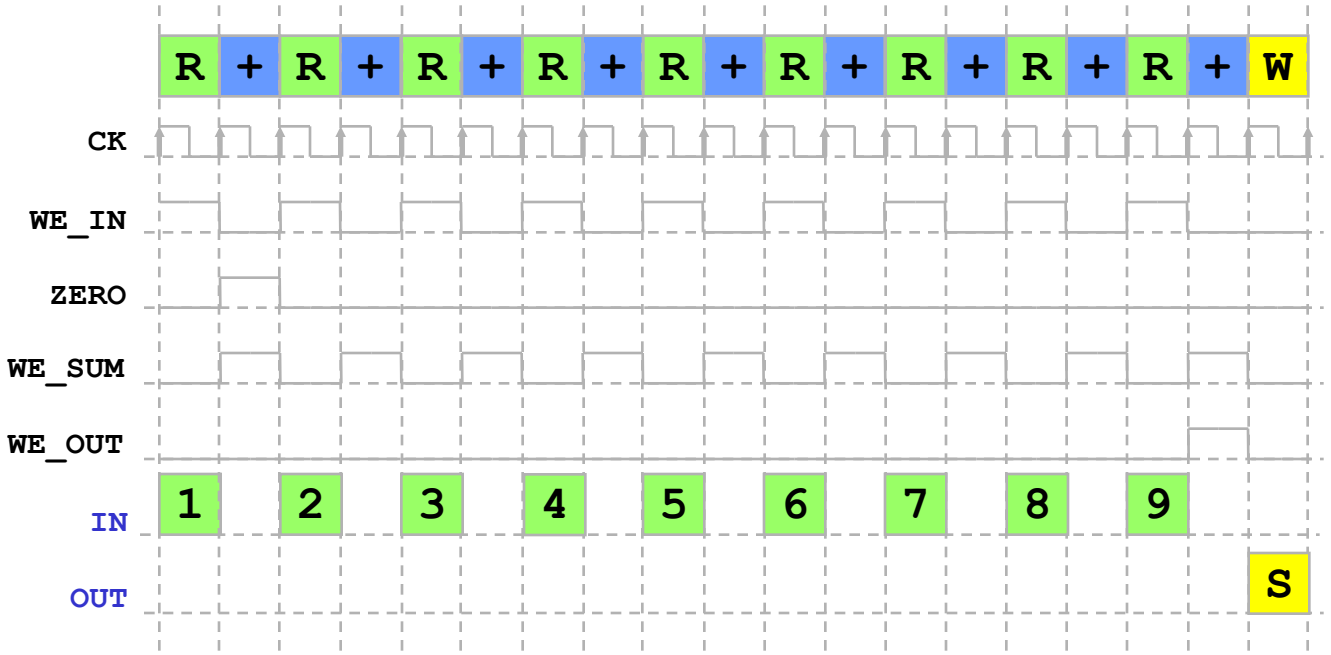




Una possibile realizzazione (data-path) potrebbe essere questa. I segnali di controllo sono mostrati nella pagina successiva.

Il risultato della somma (9 elementi) è disponibile sull'uscita dopo 19 cicli di clock (latency=19).

Le iterazioni del loop sono 9 (tripcount=9) e ciascuna impiega due cicli di clock (uno per leggere e uno per eseguire la somma incrementale).



Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	4.83	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
19	19	20	20	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- looop	18	18	2	-	-	9	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	38
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	37
Register	-	-	43	-
Total	0	0	43	75
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

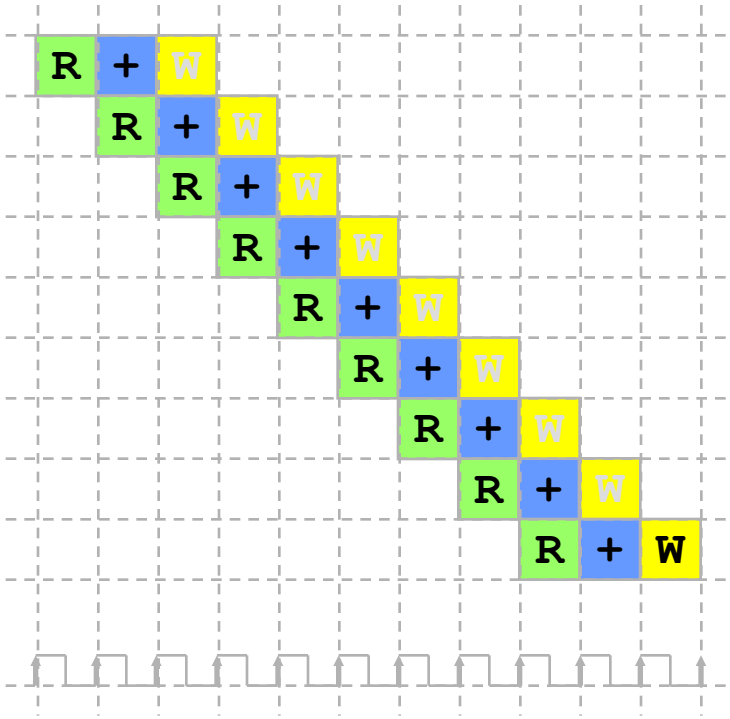
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	media	return value
ap_rst	in	1	ap_ctrl_hs	media	return value
ap_start	in	1	ap_ctrl_hs	media	return value
ap_done	out	1	ap_ctrl_hs	media	return value
ap_idle	out	1	ap_ctrl_hs	media	return value
ap_ready	out	1	ap_ctrl_hs	media	return value
input_array_address0	out	4	ap_memory	input_array	array
input_array_ce0	out	1	ap_memory	input_array	array
input_array_q0	in	32	ap_memory	input_array	array
average_value	out	32	ap_vld	average_value	pointer
average_value_ap_vld	out	1	ap_vld	average_value	pointer

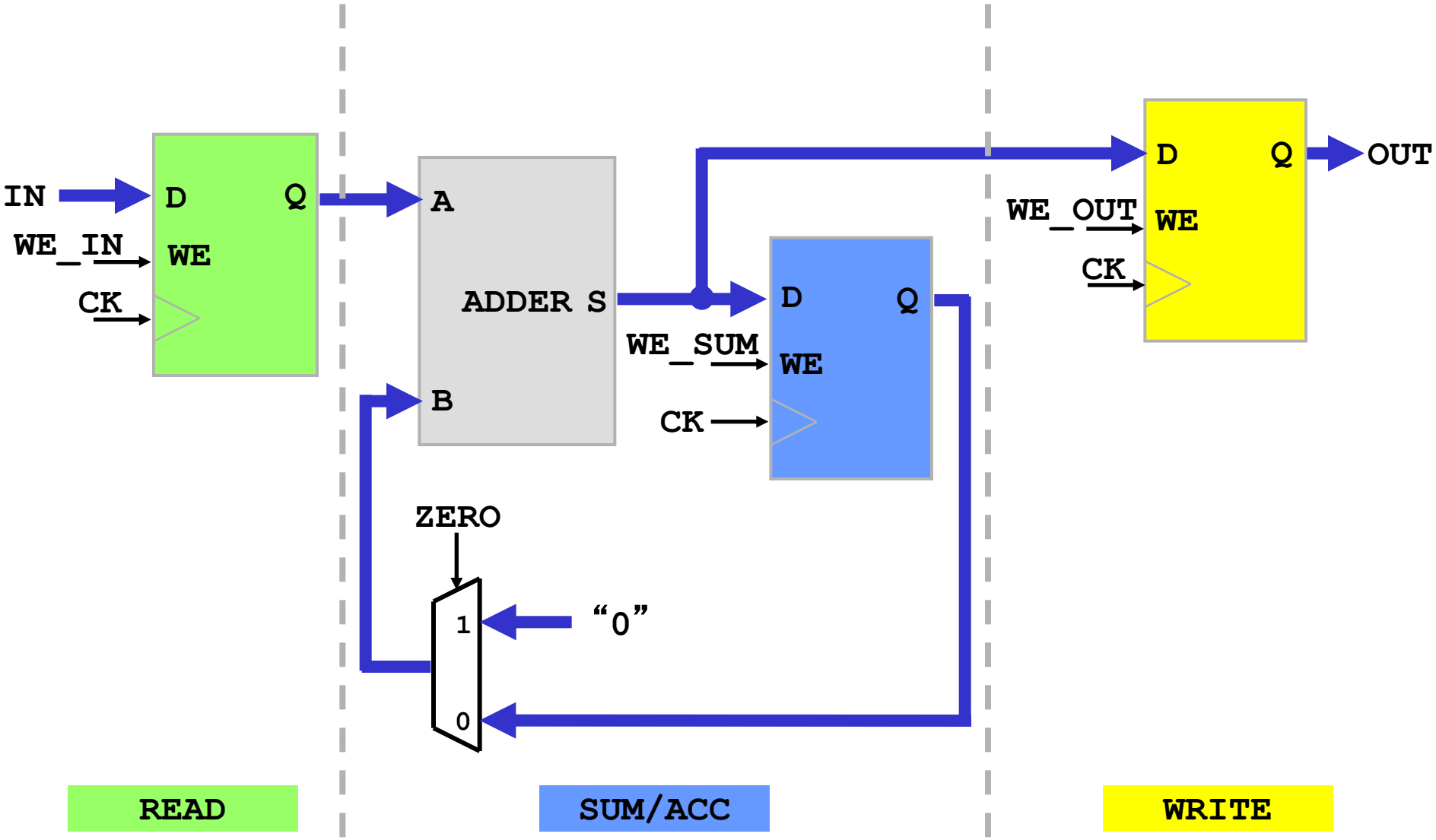
Ottimizzazioni in Vivado HLS: pipelining

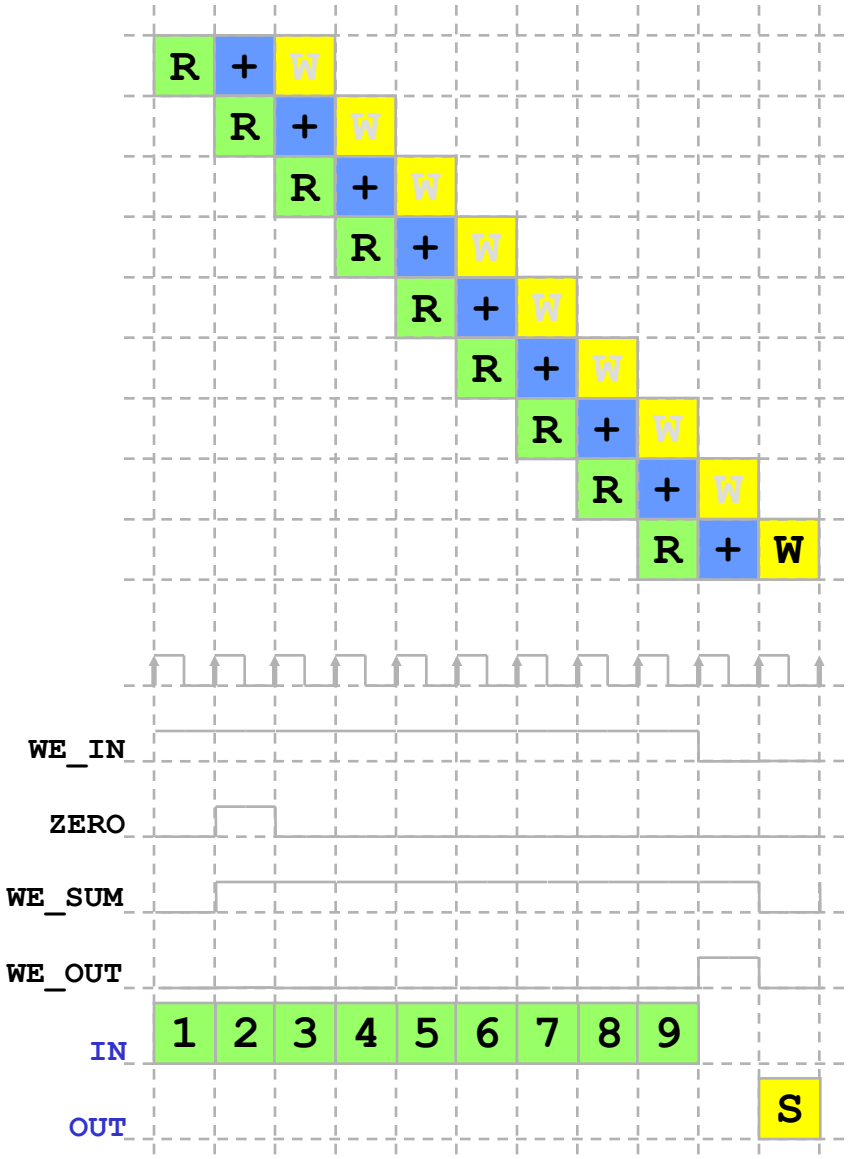
Osservando la rete precedente e le temporizzazioni si può pensare di introdurre il pipeling:



In questo caso la latency si ridurrebbe a 11 clock (vs 19 del caso precedente).

Una possibile rete per eseguire la somma si un array in pipeling (3 stadi) è la seguente.





In questo specifico caso la rete logica è identica; quelli che cambiano sono i segnali di controllo.

La versione pipelined consentirebbe di ottenere una latency di 11 cicli di clock (8 cicli in meno rispetto alla soluzione base).

Quali modifiche è necessario apportare al codice C della versione base/precedente per applicare il pipelining?
Nessuna modifica al codice, è sufficiente una #pragma

```
void media(volatile int input_array[LENGTH], volatile int
*average_value)
{
int temp_sum=0;

loop: for (int i=0; i<LENGTH; i++)
{
#pragma HLS PIPELINE II=1

temp_sum = temp_sum + input_array[i];
}

*average_value = temp_sum;

return;
}
```

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	4.83	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
11	11	12	12	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- looop	9	9	2	1	1	9	yes

La latency è effettivamente di 11 cicli di clock.

Il numero di risorse, rispetto alla versione base on pipelined, è equivalente (in questo caso)

Le modifiche al codice sorgente sono minime

In molti casi il pipeling consente di ridurre la latenza con un overhead non particolarmente elevato in termini di risorse utilizzate

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	38
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	37
Register	-	-	42	-
Total	0	0	42	75
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

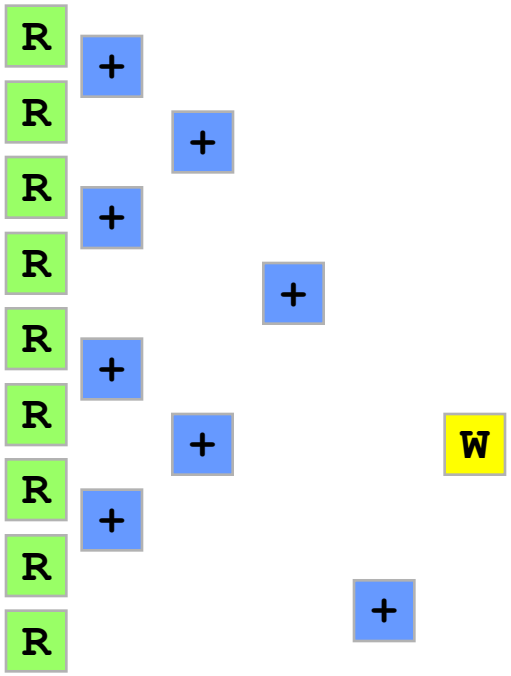
Interface

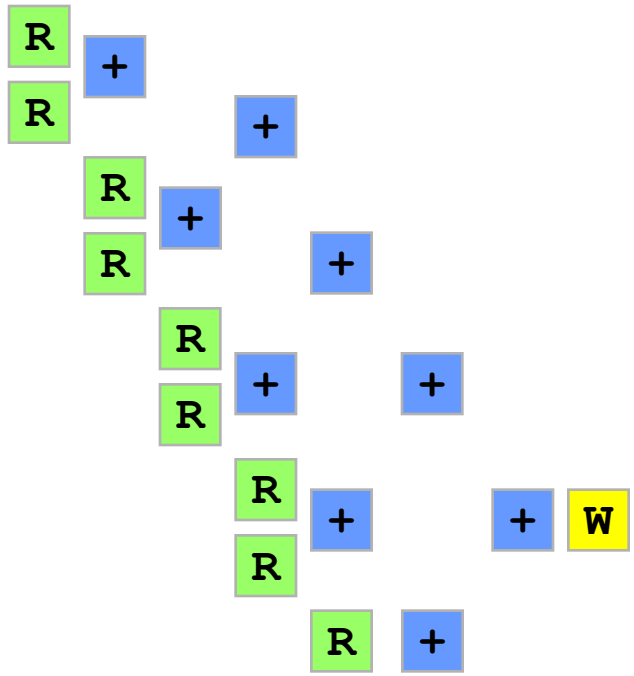
Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	media	return value
ap_rst	in	1	ap_ctrl_hs	media	return value
ap_start	in	1	ap_ctrl_hs	media	return value
ap_done	out	1	ap_ctrl_hs	media	return value
ap_idle	out	1	ap_ctrl_hs	media	return value
ap_ready	out	1	ap_ctrl_hs	media	return value
input_array_address0	out	4	ap_memory	input_array	array
input_array_ce0	out	1	ap_memory	input_array	array
input_array_q0	in	32	ap_memory	input_array	array
average_value	out	32	ap_vld	average_value	pointer
average_value_ap_vld	out	1	ap_vld	average_value	pointer

Ottimizzazioni in Vivado HLS: loop unrolling

Anche per il loop unrolling non è necessaria alcuna modifica al codice, è sufficiente una #pragma





Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.30	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
6	6	7	7	none

Detail

Instance

Loop

N/A

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	160
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	-	-	-	-
Multiplexer	-	-	-	42
Register	-	-	199	-
Total	0	0	199	202
Available	280	220	106400	53200
Utilization (%)	0	0	~0	~0

Progetto HLS di una interfaccia VGA

La VGA è una semplice interfaccia che consente di pilotare un monitor analogico utilizzando dei segnali digitali. Sono supportate varie risoluzioni e frame rate.

I segnali sono le tre componenti di colore R,G,B e due segnali di sincronismo V_SYNC (verticale) e H_SYNC (orizzontale).

I segnali digitali di RED, GREEN e BLUE a n bit sono convertiti in tre segnali analogici tra 0 e 0.7 V mediante un DAC.

Nel caso della zedboard, n=4 e il DAC è costituito da una serie di resistenze.

I pin per gestire l'interfaccia VGA con la Zedboard sono indicati nella pagina successiva.

2.4.2 VGA Connector

The ZedBoard also allows 12-bit color video output through a through-hole VGA connector, TE 4-1734682-2. Each color is created from resistor-ladder from four PL pins.

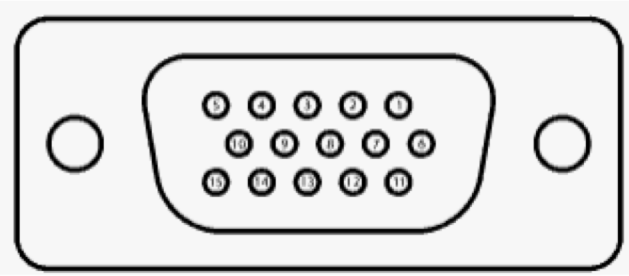
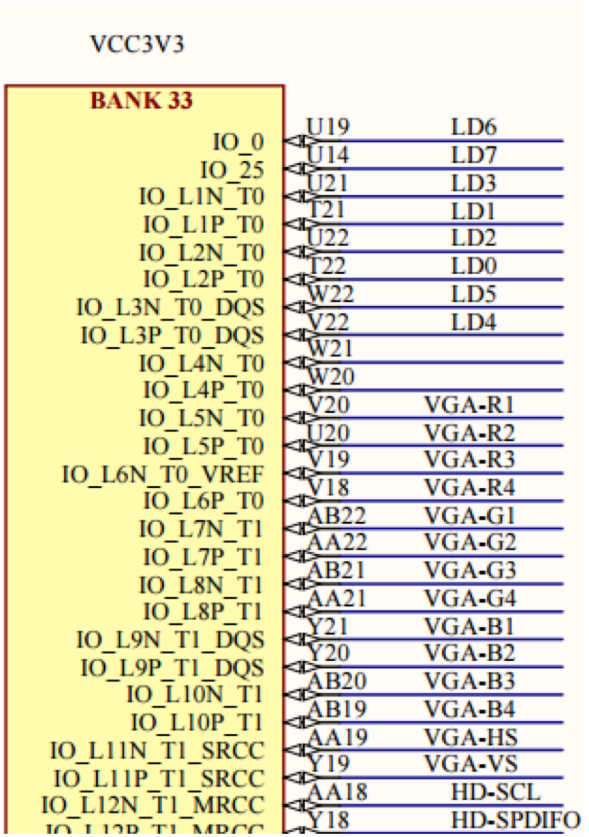


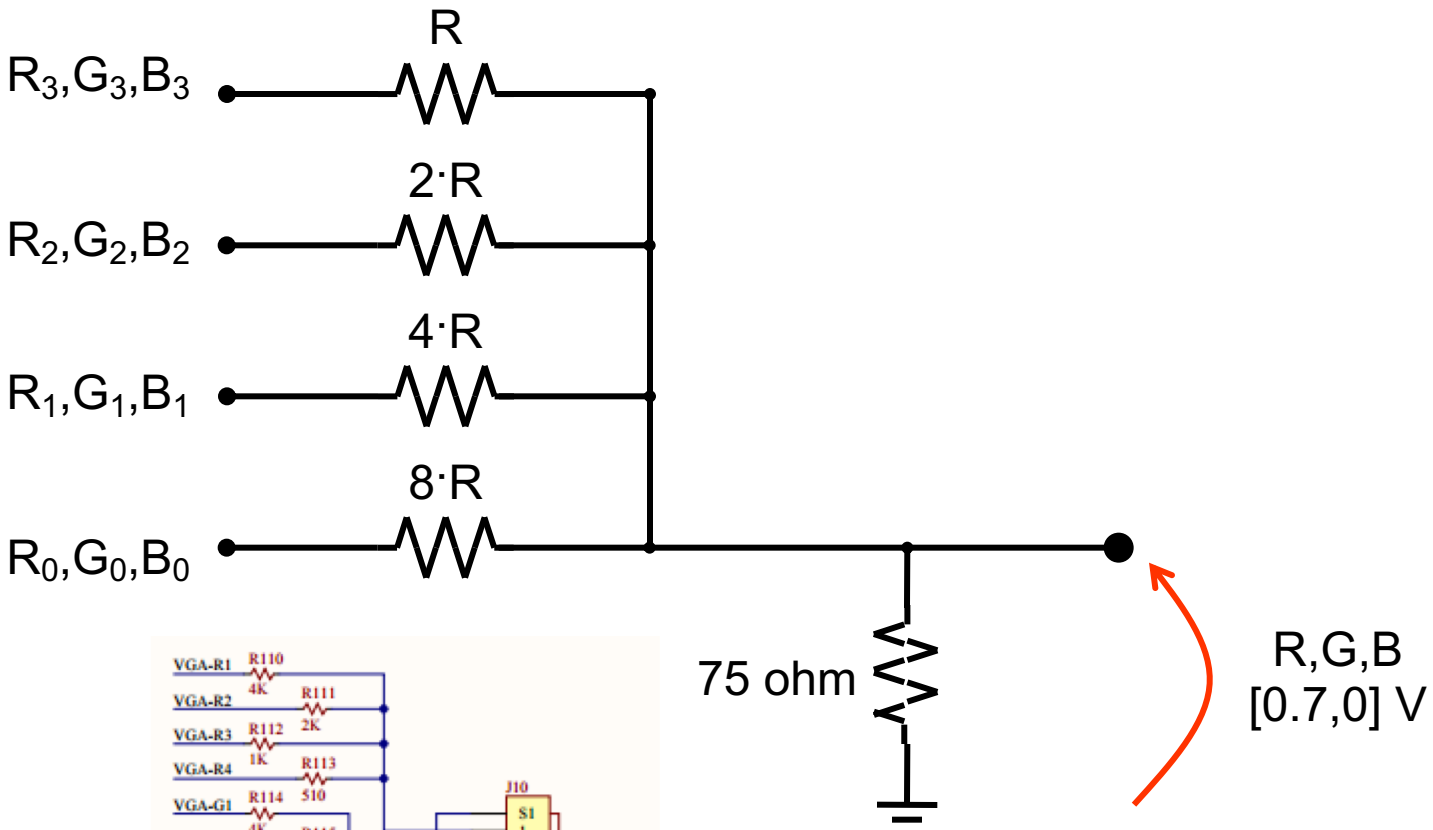
Figure 10 - DB15

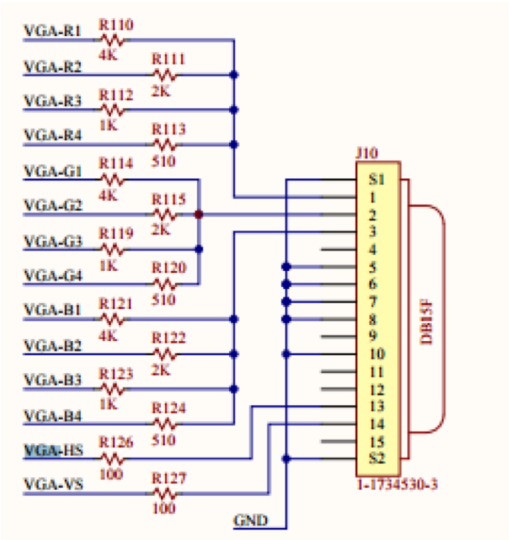
Table 8 - VGA Connections

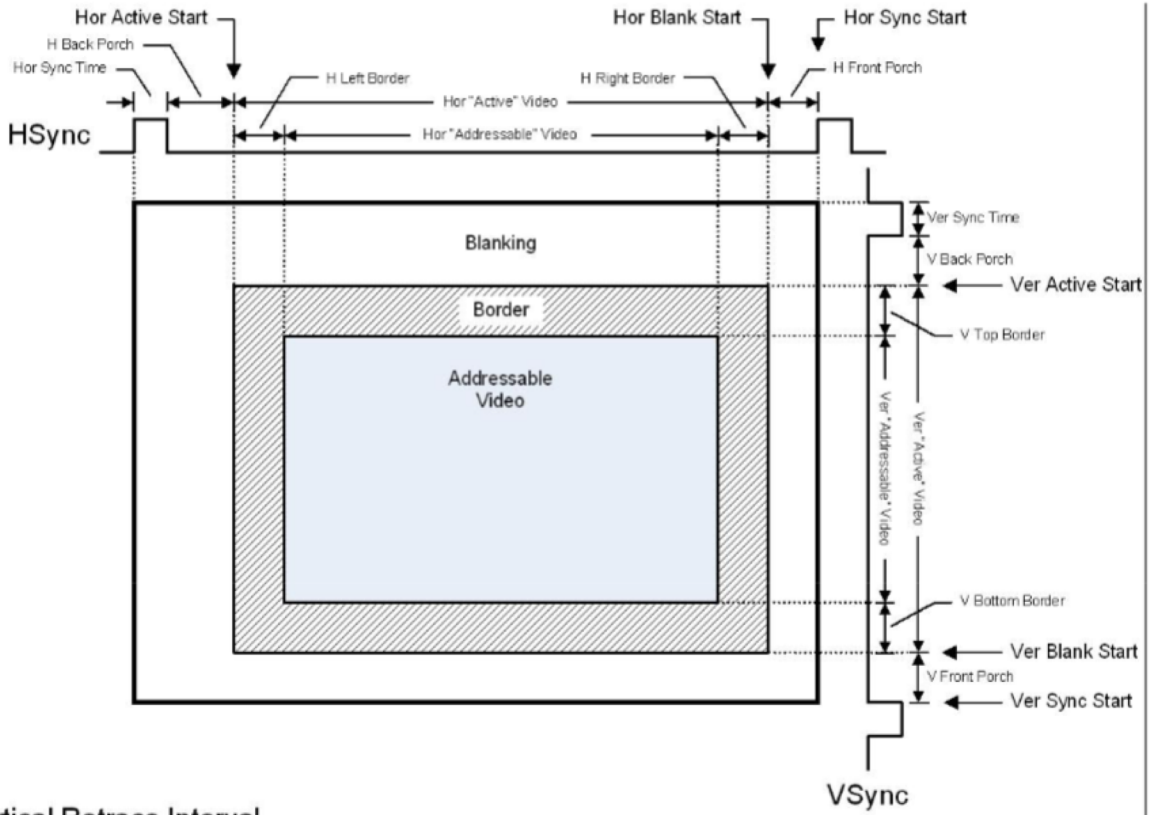
VGA Pin	Signal	Description	EPP Pin
1	RED	Red video	V20, U20, V19, V18
2	GREEN	Green video	AB22, AA22, AB21, AA21
3	BLUE	Blue video	Y21, Y20, AB20, AB19
4	ID2/RES	formerly Monitor ID bit 2	NC
5	GND	Ground (HSync)	NC
6	RED_RTN	Red return	NC
7	GREEN_RTN	Green return	NC
8	BLUE_RTN	Blue return	NC
9	KEY/PWR	formerly key	NC
10	GND	Ground (VSync)	NC
11	ID0/RES	formerly Monitor ID bit 0	NC
12	ID1/SDA	formerly Monitor ID bit 1	NC
13	HSync	Horizontal sync	AA19
14	VSync	Vertical sync	Y19
15	ID3/SCL	formerly Monitor ID bit 3	NC



Con uscite digitali a 3.3 V e 4 bit per ogni canale R,G,B, dalla rete seguente risulta R ≈ 500 ohm.





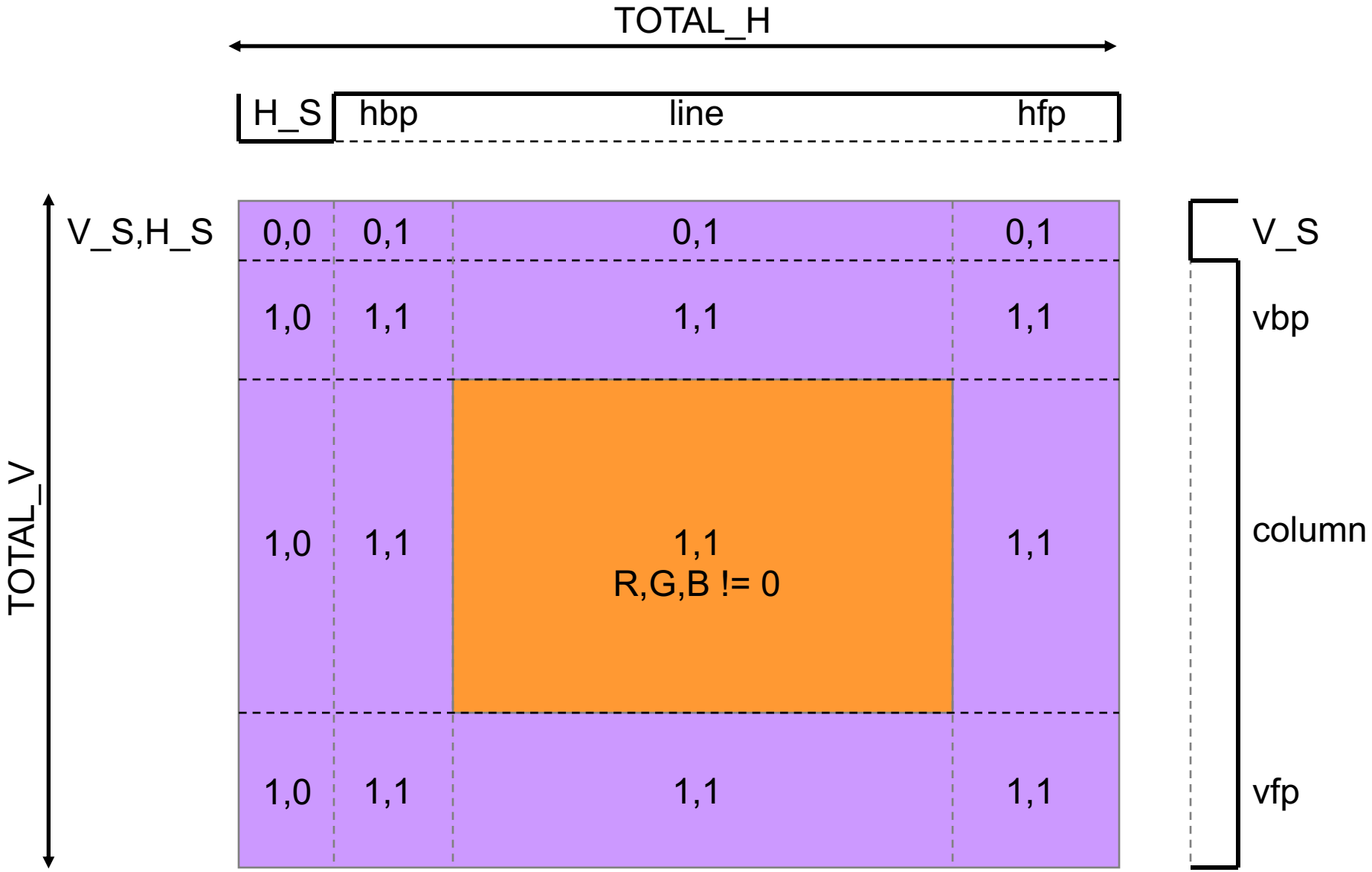


Vertical Retrace Interval

Note:

All syncs shown as active high. For active low invert the waveform as shown below

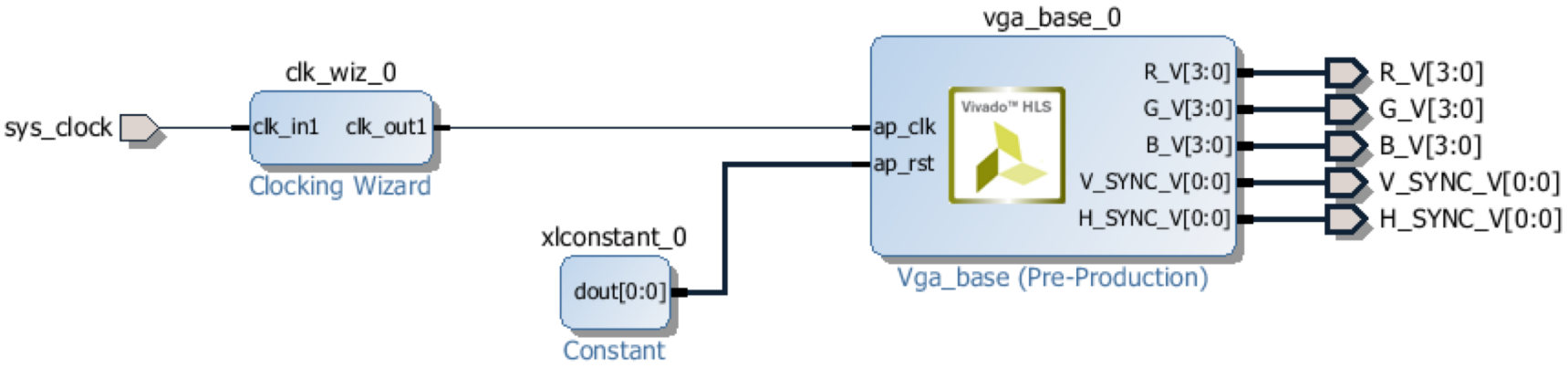
ftp://ftp.cis.nctu.edu.tw/pub/csie/Software/X11/private/VeSaSpEcS/VESA_Document_Center_Monitor_Interface/DMTv1r11.pdf

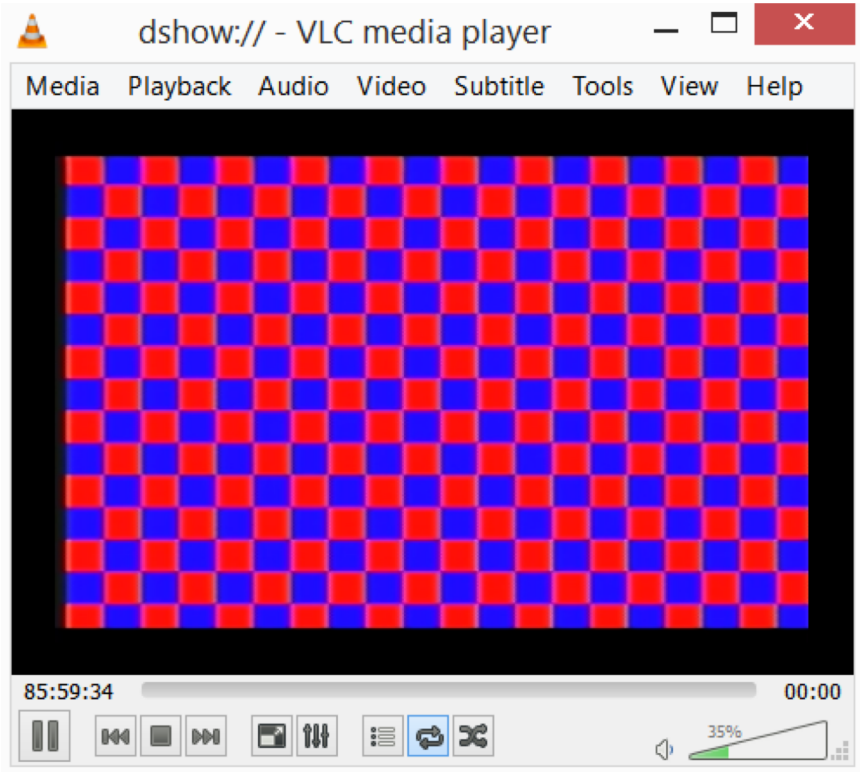

TOTAL_H
H_S
hbp
line
hfp
V_S,H_S
TOTAL_V
0,0
0,1
0,1
0,1
1,0
1,1
1,1
1,1
1,0
1,1
1,1
R,G,B != 0
1,1
1,0
1,1
1,1
1,1
V_S
vbp
column
vfp

```
#include "ap_int.h"

#define BIT_OUT 4

// VGA resolution (640x480) @ 60 Hz
// sync pulse: negative logic

#define WIDTH  640
#define HEIGHT 480

// VERTICAL timing (rows)
#define VERTICAL_FRONT_PORCH 10
#define VERTICAL_SYNC_PULSE   2
#define VERTICAL_BACK_PORCH  33


// HORIZONTAL timing (clocks @ 25.175 MHz)
#define HORIZONTAL_FRONT_PORCH 16
#define HORIZONTAL_SYNC_PULSE  96
#define HORIZONTAL_BACK_PORCH  48

void vga_base(volatile ap_uint<BIT_OUT> *R, volatile ap_uint<BIT_OUT> *G,
              volatile ap_uint<BIT_OUT> *B, volatile ap_uint<1> *V_SYNC,
              volatile ap_uint<1> *H_SYNC);
```

```
#include "vga_controller_bram_interface.h"

void vga_base(volatile ap_uint<BIT_OUT> *R, volatile ap_uint<BIT_OUT> *G,
              volatile ap_uint<BIT_OUT> *B, volatile ap_uint<1> *V_SYNC,
              volatile ap_uint<1> *H_SYNC)
{
#pragma HLS INTERFACE ap_ctrl_none port=return
#pragma HLS INTERFACE ap_none port=R
#pragma HLS INTERFACE ap_none port=B
#pragma HLS INTERFACE ap_none port=G
#pragma HLS INTERFACE ap_none port=V_SYNC
#pragma HLS INTERFACE ap_none port=H_SYNC


const int TOTAL_V = VERTICAL_SYNC_PULSE + VERTICAL_BACK_PORCH + HEIGHT +
                    VERTICAL_FRONT_PORCH;

const int TOTAL_H = HORIZONTAL_SYNC_PULSE + HORIZONTAL_BACK_PORCH + WIDTH +
                    HORIZONTAL_FRONT_PORCH;

int x,y;
ap_uint<1> V_SYNC_temp = 1;
ap_uint<1> H_SYNC_temp = 1 ;


for (y=0; y<TOTAL_V; y++)
for (x=0; x<TOTAL_H; x++)
{
#pragma HLS PIPELINE II=1
```

```
if (y<VERTICAL_SYNC_PULSE) V_SYNC_temp=0;
else V_SYNC_temp=1;


if (x<HORIZONTAL_SYNC_PULSE) H_SYNC_temp=0;
else H_SYNC_temp=1;


    if ((x>= HORIZONTAL_SYNC_PULSE + HORIZONTAL_BACK_PORCH) &&
        (x<  HORIZONTAL_SYNC_PULSE + HORIZONTAL_BACK_PORCH + WIDTH) &&
        (y>= VERTICAL_SYNC_PULSE + VERTICAL_BACK_PORCH) &&
        (y< VERTICAL_SYNC_PULSE + VERTICAL_BACK_PORCH + HEIGHT))
    {
        // DISPLAY IMAGE
        *R=0;
        *G=15;
        *B=0;
        *V_SYNC = V_SYNC_temp;
        *H_SYNC = H_SYNC_temp;
    } else
    {
        *R=0;
        *G=0;
        *B=0;
        *V_SYNC = V_SYNC_temp;
        *H_SYNC = H_SYNC_temp;
    }

}

}
```


vga_base_0
clk_wiz_0
sys_clock
clk_in1
clk_out1
Clocking Wizard
ap_clk
ap_rst
Vivado™ HLS
R_V[3:0]
G_V[3:0]
B_V[3:0]
V_SYNC_V[0:0]
H_SYNC_V[0:0]
Vga_base (Pre-Production)
xlconstant_0
dout[0:0]
Constant


dshow:// - VLC media player
Media
Playback
Audio
Video
Subtitle
Tools
View
Help
85:59:34
00:00
35%