AVR8 soft core for Papilio board with support for SPI, pin shifting and custom core example

CONTENTS

[TABLE OF FIGURES 2](#_Toc345340180)

[TABLE OF TABLES 2](#_Toc345340181)

[INTRODUCTION 3](#_Toc345340182)

[PIN SHIFTING, SPI SUPPORT AND CUSTOM CORE SUPPORT 3](#_Toc345340183)

[SUMMARY OF CHANGES 3](#_Toc345340184)

[CUSTOM CORE EXAMPLE 4](#_Toc345340185)

[PUTTING THINGS TOGETHER 8](#_Toc345340186)

[RUNNING THE DESIGN 9](#_Toc345340187)

# TABLE OF FIGURES

[Figure 1 Papilio pin assignments 8](#_Toc345335012)

# TABLE OF TABLES

[Table 1 List of Arduino register names, names of corresponding constants in VHDL code and corresponding memory addresses in \_SFR\_IO8 format 7](#_Toc345335100)

[Table 2 Input signal to pin mapping on Papilio board 8](#_Toc345335101)

## INTRODUCTION

The initial AVR8 soft cores for Papilio FPGA boards came in two flavours: “Vanilla” that had limited functionality (but worked out support and example for custom cores!) and “Shifty” that had ability to support SPI, pin shifting and much more. Straight addition of custom core example to “Shifty” core led to some memory mapping conflicts, that are resolved in the current example.

The forked version of AVR8 support SPI, pin shifting and custom cores. The core examples is a specific hardware offload core that adds simple signal processing ability – it measures “up” and “down” time of the high frequency signal, and it also measures how many times signal switches from “down” to “up” and from “up” to “down” (“up” is equivalent to HI voltage, and “down” is equivalent to LO). Also added to the arduino sketch code is the backup of the registers to the SD card, so that the completed design is robust against emergency power-off (EPO). The support for SD card takes advantage of the SPI support, which otherwise would not be possible with Vanilla fork of the AVR8.

## PIN SHIFTING, SPI SUPPORT AND CUSTOM CORE SUPPORT

Support for pin shifting (and SPI) comes from “Shifty” fork of AVR8. The custom core examplehas already been worked out in the “Vanilla” fork. The present examples put it all together, and resolves several issues along the way.

### SUMMARY OF CHANGES

The changes that have happened include following:

1. Transition from 6 bit I/O to 16 bit I/O addressing. While “Vanilla” AVR8 uses 16 bit I/O addressing, “Shifty” core still got a lot of 6 bit I/O addressing used. The new version uses 16 bit I/O for all peripherals, removing any conflicts. This change allowed adding custom core code to “Shifty” branch, and successful synthesis, place and route and bitstream generation, but the core functionality was impeded.
2. By careful simulation of the design, using examples from papilio.cc[[1]](#footnote-1) it was possible to observe that iowe of AVR core was not behaving correctly, and that some changes were necessary to pm\_fetch\_dec.vhd and swap\_pins.vhd. The part of the issue was how pin swapping was implemented - the change to constant const\_ram\_to\_io\_c in pm\_fetch\_dec.vhd to "0010" was necessary as well as the use of memory space from 0x2000 in the custom core – to resolve all the prior conflicts in 0x1000 - 0x1FFFF portion of memory space.

The changes in pm\_fetch\_dec.vhd are outlined in the code sample below:

-- ####################################################

-- INTERNAL SIGNALS

-- ####################################################

-- NEW SIGNALS

signal two\_word\_inst : std\_logic; -- CALL/JMP/STS/LDS INSTRUCTION INDICATOR

signal ram\_adr\_int : std\_logic\_vector (15 downto 0);

constant const\_ram\_to\_reg : std\_logic\_vector := "00000000000"; -- LD/LDS/LDD/ST/STS/STD ADDRESSING GENERAL PURPOSE REGISTER (R0-R31) 0x00..0x19

constant const\_ram\_to\_io\_a : std\_logic\_vector := "00000000001"; -- LD/LDS/LDD/ST/STS/STD ADDRESSING GENERAL I/O PORT 0x20 0x3F

constant const\_ram\_to\_io\_b : std\_logic\_vector := "00000000010"; -- LD/LDS/LDD/ST/STS/STD ADDRESSING GENERAL I/O PORT 0x20 0x3F

--constant const\_ram\_to\_io\_c : std\_logic\_vector := "0001"; -- LD/LDS/LDD/ST/STS/STD ADDRESSING GENERAL I/O PORT 0x1000 0x1FFF

constant const\_ram\_to\_io\_c : std\_logic\_vector := "0010"; -- LD/LDS/LDD/ST/STS/STD ADDRESSING GENERAL I/O PORT 0x2000 0x2FFF -> change by Zvonimir Bandic

constant const\_ram\_to\_io\_d : std\_logic\_vector := "00100000000"; -- LD/LDS/LDD/ST/STS/STD ADDRESSING GENERAL I/O PORT 0x1000 0x3FFF

## CUSTOM CORE EXAMPLE

The custom core implemented in this example is slightly more interesting that the original example shown with “Vanilla” core. It implements the following:

1. It can operate with the input signal with frequencies up to 2MHz[[2]](#footnote-2), and it measures the “up” (HI) time of the signal, the “down” (LO) time of the signal, as well as the number of times signal transitioned from “up” (HI) to “down”(LO), and from “down” (LO) to “up” (HI) and then stores these values encoded as 64 bit numbers, *counter\_up\_addr\_7…0, counter\_down\_addr\_7…0, ticker\_up\_addr\_7…0 and ticker\_down\_addr\_7…0* into the memory locations , shown in the code sample below, and also listed in Table 1.

--Control Register is used to control the output signals. Default Value is all zeroes.

signal control\_reg : std\_logic\_vector(7 downto 0):= "00000000";

constant control\_addr : std\_logic\_vector(15 downto 0):= io\_base\_address\_generic;

--Status Register is used to read the input signals. Default Value is all zeroes.

signal status\_reg : std\_logic\_vector(7 downto 0):= "00000000";

constant status\_addr : std\_logic\_vector(15 downto 0):= io\_base\_address\_generic + 1;

signal control\_Sel, status\_Sel : std\_logic;

--Counter register is used to count the diode signal on time. Default is all zeroes.

signal counter\_up\_reg : std\_logic\_vector(63 downto 0):="0000000000000000000000000000000000000000000000000000000000000000";

constant counter\_up\_addr\_7: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 2;

constant counter\_up\_addr\_6: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 3;

constant counter\_up\_addr\_5: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 4;

constant counter\_up\_addr\_4: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 5;

constant counter\_up\_addr\_3: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 6;

constant counter\_up\_addr\_2: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 7;

constant counter\_up\_addr\_1: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 8;

constant counter\_up\_addr\_0: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 9;

signal counter\_down\_reg : std\_logic\_vector(63 downto 0):="0000000000000000000000000000000000000000000000000000000000000000";

constant counter\_down\_addr\_7: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 10;

constant counter\_down\_addr\_6: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 11;

constant counter\_down\_addr\_5: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 12;

constant counter\_down\_addr\_4: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 13;

constant counter\_down\_addr\_3: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 14;

constant counter\_down\_addr\_2: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 15;

constant counter\_down\_addr\_1: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 16;

constant counter\_down\_addr\_0: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 17;

--Ticker registers are used to count the diode signal uptick or downtick. Default is all zeroes.

signal ticker\_up\_reg : std\_logic\_vector(63 downto 0):="0000000000000000000000000000000000000000000000000000000000000000";

constant ticker\_up\_addr\_7: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 18;

constant ticker\_up\_addr\_6: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 19;

constant ticker\_up\_addr\_5: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 20;

constant ticker\_up\_addr\_4: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 21;

constant ticker\_up\_addr\_3: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 22;

constant ticker\_up\_addr\_2: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 23;

constant ticker\_up\_addr\_1: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 24;

constant ticker\_up\_addr\_0: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 25;

signal ticker\_down\_reg : std\_logic\_vector(63 downto 0):="0000000000000000000000000000000000000000000000000000000000000000";

constant ticker\_down\_addr\_7: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 26;

constant ticker\_down\_addr\_6: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 27;

constant ticker\_down\_addr\_5: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 28;

constant ticker\_down\_addr\_4: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 29;

constant ticker\_down\_addr\_3: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 30;

constant ticker\_down\_addr\_2: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 31;

constant ticker\_down\_addr\_1: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 32;

constant ticker\_down\_addr\_0: std\_logic\_vector(15 downto 0):=io\_base\_address\_generic + 33;

1. The soft core utilizes three input signals, as shown in the code snippet below:

--Two Input Signals and one diode signal which is being processed (three input signals total)

-- input\_sig(0) is used as ENABLE signal, and input\_sig(1) is used to reset counters/tickers

input\_sig : in std\_logic\_vector (1 downto 0);

--diode Signal

diode\_sig : in STD\_LOGIC

1. The following code snippet from top\_module Papilio\_avr8.vhd explains how these signals connect to Papilio board inputs:

-- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* User Cores - Instantiate User Cores Here \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

-- Example Core - core9 - This is an example of implenting a custom User core.

Inst\_diode\_timer:if CImpldiode\_timer generate

diode\_timer\_COMP:component diode\_timer

PORT MAP(

nReset => nrst,

clk => clk16M,

adr => core\_adr,

dbus\_in => core\_dbusout,

dbus\_out => core9\_dbusout,

out\_en => core9\_out\_en,

iore => core\_iore,

iowe => core\_iowe,

output\_sig => porta(1 downto 0), -- this needs to match whatever number of bits we use in the custom core

input\_sig => portb(1 downto 0), -- in diode\_timer; for full width of 8 it becomes just porta,portb -

-- input\_sig(0) is used as ENABLE signal, and input\_sig(1) is used to reset counters/tickers

diode\_sig => portb(3) -- this is the diode signal, i.e. is equal to 1 when diode signal is on

);

-- Example Core - core9 connection to the external multiplexer

io\_port\_out(10) <= core9\_dbusout;

io\_port\_out\_en(10) <= core9\_out\_en;

1. This implies that input signals should be attached to the pins on the Papilio board, as shown in Table 2 below. The pins map from papilio.cc webpage is reproduced in Figure 1 below. (and is also contained in Papilio\_one.ucf constraints file).
2. The functionality of input signals is explained in the Functionality columns of Table 2. This implies that bringing pin P62 to ground level will reset all the counters, and pin P60 should be kept on 3.3V in order for signal processing to proceed, or to ground to pause. The signal for analysis should be connected to pin P67.

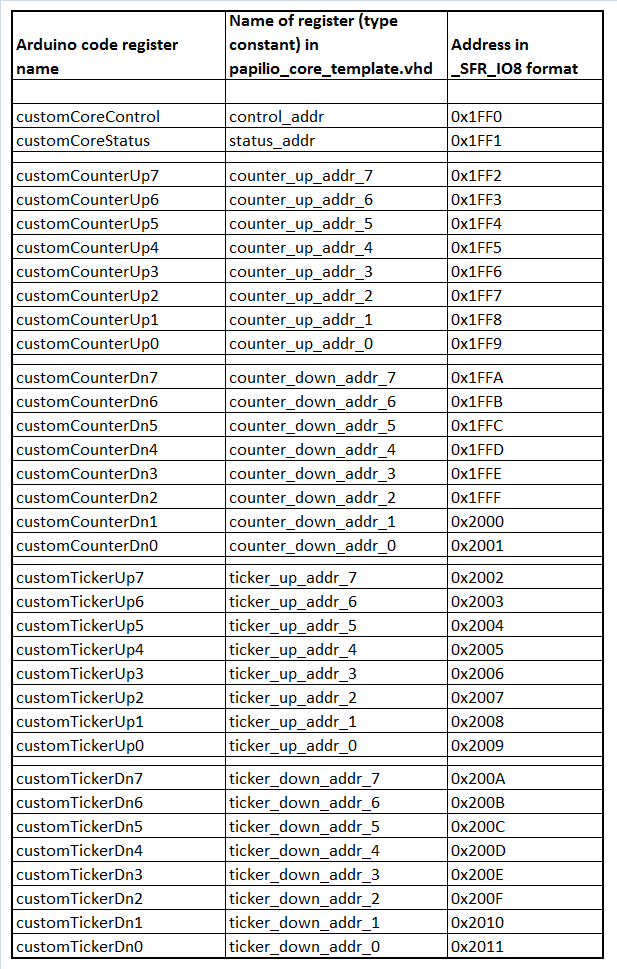


Table List of Arduino register names, names of corresponding constants in VHDL code and corresponding memory addresses in \_SFR\_IO8 format

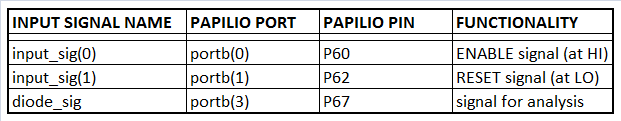


Table 2 Input signal to pin mapping on Papilio board

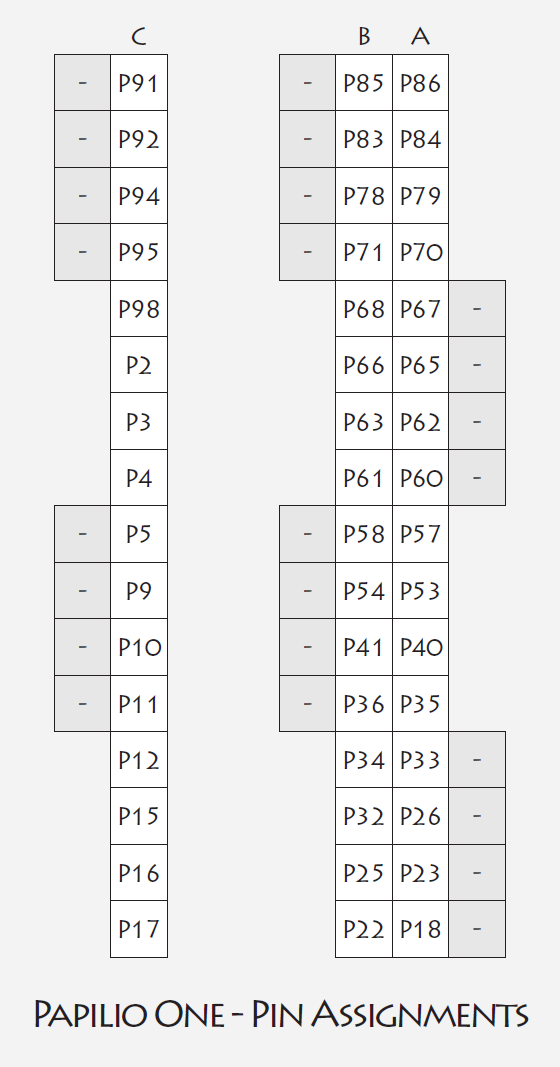


Figure Papilio pin assignments

## PUTTING THINGS TOGETHER

This section is added for the sake of completeness – it is a summary of information already available via papilio.cc and GadgetFactory.net forums. Once bitstream papilio\_avr8.bit and memory map papilio\_avr8\_bd.bmm are generated inside Xilinx ISE, they should be transferred to the appropriate Arduino 18 for Papilio IDE directory, and renamed to custom.bit and custom\_bd.bmm.

For that purpose, one can use script copy-to-custom.cmd located in the scripts folder. This script should be edited to point out final destination for the bitstream, which is currently set to:

C:\Program Files\Papilio-ArduinoIDE0018f\hardware\tools\butterfly\_platform\bitstreams.

Once custom.bit and custom\_bd.bmm are present in the Papilio-ArduinoIDE0018f bitstreams folder, they should be written to the Papilio board flash, using any of the tools available on papilio.cc webpage.[[3]](#footnote-3) After launching arduino.exe from the the Papilio-ArduinoIDE0018f folder, it is important to select Papilio custom board through Tools -> Board -> Gadget Factory Papilio Custom Board. This will ensure that firmware binary code written in Arduino IDE becomes correctly loaded into Papilio memory. Exemplary code for the signal processing core is provided in the AVR8\_Custom\_User\_Core\_Example folder. If one presses SHIFT during upload of the code to the Papilio board, the command shell will reveal that out.bit file is generated, and it will also reveal its temporary folder, that will look similar to:

C:\Users\Username\AppData\Local\Temp\build8538008223705770340.tmp

At that point, if the design is finalized, it is worth saving out.bit file, as it can be used later to burn the bitstream permanently to the flash on Papilio.

## RUNNING THE DESIGN

Once input signal for analysis is connected to pin P67, pin P62 is disconnected and pin P60 is on 3.3V, we can start running the design. Connect the Papilio board to USB connector, and use Putty.exe to read the serial output from the Papilio board. The output should look like:  
  
Write counterUp7-Up0

00

00

00

00

2E

50

9C

7B

Write counterDn7-Dn0

00

00

00

00

2D

D4

D5

0F

Write TickerUp7-Up0

00

00

00

00

0B

3D

B4

E6

Write TickerDn7-Dn0

00

00

00

00

0B

2A

E1

05

The outputs are basically 8 bytes value of each of the registers, shown in small endianess and they are printed out and backed up approximately every 5 seconds (by adjusting threshold for variable backupCounter in the Arduino code, and loop delay (which is currently set to delay(2000)). This is illustrated in the following snippet of the Arduino code:

void loop()

{

//report periodically and save to SD card

if (backupCounter == 6) //6 is about 5 seconds, but adjust value later 600=10 minutes

{

backupCounter = 0;

// open a file

char name[] = "BACKUP.TXT";

file.open(&root, name, O\_CREAT | O\_TRUNC | O\_WRITE);

// if (!file.isOpen()) error ("file.create");

file.write(customCounterUp0);

file.write(customCounterUp1);

file.write(customCounterUp2);

.

.

.

file.write(customTickerDn6);

file.write(customTickerDn7);

//

Serial.println("Write counterUp7-Up0");

PrintHexTwoChar(customCounterUp7);

PrintHexTwoChar(customCounterUp6);

.

.

.

PrintHexTwoChar(customTickerDn1);

PrintHexTwoChar(customTickerDn0);

// close file and force write of all data to the SD card

file.close();

}

backupCounter++;

delay(2000); //10000 seems around 5 seconds with AVR core on running on Papilio

In the event of emergency power off (or power loss), the state of the counters is preserved on the micro SD card, and reloaded into the design. If one once to start counting from zero, then make sure to reset the board at pin P62 – otherwise the state of the counters present in the BACKUP.TXT file on the SD card will be used as the starting state.

1. <http://www.papilio.cc/index.php?n=Papilio.SimulateACustomAVR8UserCore> [↑](#footnote-ref-1)
2. The actual bandwidth is limited by the clock frequency – the same code could be used on a faster board to allow processing of the signal with higher frequency. [↑](#footnote-ref-2)
3. http://www.papilio.cc/index.php?n=Papilio.GettingStarted [↑](#footnote-ref-3)