INTENSIIVKURSUS
"FPGA BASED CONTROL SOLUTIONS FOR SPECIFIC POWER CONVERTERS"

Prof. ILYA GALKIN, Riga Technical University

Tallinn 2012
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>1</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>2</td>
</tr>
<tr>
<td>Problem of Specific Control Task</td>
<td>2</td>
</tr>
<tr>
<td>Other Power Converters with Complicated Control</td>
<td>3</td>
</tr>
<tr>
<td>Training Task</td>
<td>5</td>
</tr>
<tr>
<td>2. Single-Ended Primary Inductor Converter</td>
<td>7</td>
</tr>
<tr>
<td>Genesis, Schematic and Operation of SEPIC</td>
<td>7</td>
</tr>
<tr>
<td>Choice of elements for SEPIC</td>
<td>8</td>
</tr>
<tr>
<td>SEPIC Converter of Training Kit</td>
<td>10</td>
</tr>
<tr>
<td>3. Interleaved Control of Power Converters</td>
<td>11</td>
</tr>
<tr>
<td>Basics of Interleaved Control</td>
<td>11</td>
</tr>
<tr>
<td>Obtaining the Interleaved Control Signals from Programmable Logic</td>
<td>13</td>
</tr>
<tr>
<td>4. Getting Analog Data through SPI Interface</td>
<td>18</td>
</tr>
<tr>
<td>Main Features of SPI</td>
<td>18</td>
</tr>
<tr>
<td>Communication of Programmable Logic and SPI</td>
<td>20</td>
</tr>
<tr>
<td>5. Features of P and PI regulators made with FPGA</td>
<td>24</td>
</tr>
<tr>
<td>Appendix A: VHDL Quick Reference</td>
<td>27</td>
</tr>
<tr>
<td>Appendix B: Layout and Connections of FPGA Training Kit</td>
<td>34</td>
</tr>
<tr>
<td>Layout of Cyclone III FPGA Training Kit</td>
<td>34</td>
</tr>
<tr>
<td>Connections of Cyclone III FPGA Training Kit</td>
<td>34</td>
</tr>
<tr>
<td>Appendix C: Schematics of FPGA Training Kit</td>
<td>35</td>
</tr>
<tr>
<td>Main Board</td>
<td>35</td>
</tr>
<tr>
<td>FPGA Cyclone III Target Board</td>
<td>36</td>
</tr>
<tr>
<td>SEPIC Module</td>
<td>37</td>
</tr>
<tr>
<td>References</td>
<td>38</td>
</tr>
</tbody>
</table>
1. Introduction

Problem of Specific Control Task

Figure 1.1 presents a regulation loop intended for keeping one parameter, named process variable (in this case it is voltage), at the desirable level. The loop consists of a regulator, actuator, feedback, error detector, as well as reference generator. The regulator produces a control command based on the error signal. The actuator directly influences the process variable depending on the control command. The operation of actuator depends also on various environment variables which are more or less random values (at least within a range). In the given example it is a voltage converter the voltage of which depends not only on the duty cycle, but also on the input voltage, output current, temperature etc. The error is calculated as a difference of reference value provided by the reference generator and the process variable provided by the feedback.

![Figure 1.1 Structure of a Typical Regulator](image)

![Figure 1.2 Two actuators for voltage regulation loop: a) simple boost; b) boost interleaved](image)
Depending on the technical task and operation condition the actuators may be different – more or less complicated. Let’s assume that the voltage regulation loop shown in Figure 1.1 must always have higher output voltage than that of the input. If there are no specific conditions a simple boost converter can be used as an actuator (Figure 1.2-a). Then the control elements of the loop (regulator, reference and, partly, feedback) can be deployed in a simple microcontroller (Figure 1.3), for example, in MSP430G2231, which has an analog-to-digital converter (ADC10) suitable for feedback and general purpose timer (Timer0_A2) with compare function suitable for PWM generation.

However, if special requirements exist, some more sophisticated regulator has to be chosen. For instance, if the above described voltage regulator must have very well smoothed input current a boost interleaved converter could be used (see Chapter 3 for more details). This converter (Figure 1.2-b) has two (or more) parallel branches and two (or more) switches that have to be driven with shifted PWM signals. Such task cannot be anymore solved with MSP430G2231 because it requires 2 timers with compare function, but this MCU has only one. Moreover both PWM signals must be synchronized, that requires additional hardware or software.

**Other Power Converters with Complicated Control**

**Matrix converters**

Matrix converter, most frequently regarded as a direct frequency converter, is just a matrix of bidirectional switches connecting all its inputs to all its outputs (Figure 1.4-a). Being applied in drive applications the converter produces pulse width modulated output voltages. However, unlike common VSI there are 3 input voltages and up to 3 modulated voltages are mixed at the same output (Figure 1.5-a). Each of these input-to-output modulation processes seems simple, but the corresponding control signals to the switches are more complex than traditional PWM.

Moreover, commutation itself in the matrix converters is more complex. Operation of the bidirectional switches is defined by two transistors placed in the switch (Figure 1.4-b). During commutations these transistors (2 in incoming and 2 in outgoing switch – Figure 1.4-c) must behave like those in common VSI – some as main transistor, but some – as freewheeling diode. However, taking into account bipolar input voltage, the same switch acts as the main
transistor during one part of the time period, but the other – as the freewheeling diode. This leads to a quite complicated sequence of control pulses to the commutated transistors known as 4-step commutation strategy (Figure 1.5-b).

Finally, AC drives with matrix converters often utilize some sophisticated hi-level control strategy – field oriented or direct torque control. This makes control solutions for matrix converters more than trivial. It is quite common to combine DSP and FPGA in such solutions.

![Figure 1.4 Schematic features of conventional matrix converter: a) general schematic; b) schematic of bidirectional switch; c) commutation schematic](image)

![Figure 1.5 Operation diagrams of conventional matrix converter: a) multi-phase pulse-width modulation; b) commutation diagram](image)
Multilevel Converters

One mode power converter requiring specific control solution – is multilevel converter. A diode clamped version of three-level converter is given in Figure 1.6-a, but its output phase voltage - in Figure 1.6-b. In such converters additional voltage levels are formed with the help of capacitor divider. Some control strategies assume that during certain intervals the switches are constantly off, during the others - constantly on, but may be also pulse-width modulated, as well as specific dead times of shoot-through states may appear. Again this is not a trivial task that can be solved with a common interval timer and compare module.

Figure 1.6 Schematic (a) and operation diagram (b) of diode clamped multilevel converter

Training Assignment

In the given material a closed loop voltage regulation system for LED lighting application is discussed. It is known that LEDs brightness is almost proportional to their current that, in turn, is defined by the voltage applied to the LEDs as well as on their VA-curve. So in simple
occasion it is possible to adjust the light, produced by LEDs by means of applying to them an adjustable voltage.

The explored regulation system (Figure 1.1) consists of an interleaved 4-channel SEPIC converter with attached LED load, voltage sensor, analog-to-digital converter and a control system. Since the interleaved control principle requires specific control signals it is deployed in a programmable logic IC. The given report describes the part of the proposed system.

Figure 1.7 Voltage regulation system with SEPIC converter, MCP3004 ADC and FPGA based control system
2. Single-Ended Primary Inductor Converter

**Genesis, Schematic and Operation of SEPIC**

The converter utilized in the course is called Single-Ended Primary Inductor Converter (SEPIC). In order to synthesize a SEPIC converter let’s imagine a buck/boost circuit supplied from a large capacitor C. If positive output voltage is required then the polarity of the input capacitor (source) has also to be changed (green area in Figure 2.1-a). The capacitor, however, cannot supply the circuit and its load forever. It has to be recharged from a voltage source, which can be attached as it is shown in the yellow area in Figure 2.1-a. The additional inductor L1 is necessary to avoid a short-circuit of the source. The return of the source could be attached to the point (a), but simpler driver is possible if it is connected to the common ground. The above considerations form a valid SEPIC circuit given in Figure 2.1.

![Figure 2.1 Schematic of single-ended primary inductor converter: 1) genesis; 2) current paths if switch is connected; 3) current path if switch is disconnected](image-url)
The converter given in Figure 2.1 has two stages of operation in the continuous conduction mode. The first one takes place while the controllable switch S of the converter is on. Then the circuit in fact is split into three independent loops: 1) the input source charges inductor L1; 2) the internal capacitor C charges inductor L2 and 3) the output capacitor \(C_O\) supplies the load of the converter (Figure 2.1-b). The second stage starts as soon as the switch S is turned off. Then the input source and input inductor L1 in one branch as well as the inductor L2 in the second branch together feed the output capacitor \(C_O\) and the load. The internal capacitor C is being discharged during this time (Figure 2.1-c).

The static equation of SEPIC converter can be derived from the voltage balance equation of the inductor L2. While the switch S is on this inductor is attached to the internal capacitor C with voltage \(V_C\). During the second stage, when the switch S is off and currents are conducted by the diode D this inductor is tied to the output capacitor \(C_O\) with voltage \(V_O\). Then the average voltage over this inductor that within a commutation period must be 0 looks as:

\[
V_C \cdot D \cdot T - (-V_O)(1-D) \cdot T = 0,
\]

where: \(T\) – is commutation period and \(D\) – duty cycle. (2.1) can be rewritten as:

\[
V_O = \frac{D}{1-D} \cdot V_C.
\]

At the same time it is possible to write Kirchhoff’s voltage law for the average voltages of the loop \(V_{IN}\), L1, C, L2. Taking into account that the average voltages of the inductors must be 0, this gives:

\[
V_{IN} = V_{L1a} + V_C - V_{L2a} = 0 + V_C - 0 = V_C
\]

that, in turns, gives the static equation of buck/boost converter:

\[
V_O = V_C = \frac{D}{1-D} \cdot V_{IN}.
\]

The latter equation proves that SEPIC is essentially buck/boost converter without voltage sign inversion. It also indicates that the switch of SEPIC cannot be left in the on-state (this is a short circuit at \(D=1\)), as well as that the converter cannot operate without any load.

**Choice of elements for SEPIC**

**Maximal Duty Cycle**

A simplified selection procedure for SEPIC elements is given below [1]. First of all the maximal value of the duty cycle \(D_R\) has to be calculated. The corresponding equation following from (2.2) is

\[
D_{max} = \frac{V_O}{V_{INmin} + V_O}
\]

or more accurately
\[ D_{\text{max}} = \frac{V_O + V_D}{V_{\text{INmin}} + V_O + V_D} \]  

(2.6)

where \( V_O \) is voltage drop over diode, but \( V_{\text{INmin}} \) – the minimal input voltage.

**Selection of Inductors**

If the current ripples in the equal inductors \( L_1 \) and \( L_2 \) are 40% of the input current at the minimal voltage then the value of the ripples can be calculated as

\[ \Delta I_O = 0.4 \cdot I_N = 0.4 \cdot \frac{V_O}{V_{\text{INmin}}} \cdot I_O, \]  

(2.7)

but the value of the inductance as

\[ L_1 = \frac{V \cdot \Delta t}{\Delta I} = \frac{V_{\text{INmin}} \cdot D_{\text{max}} \cdot T}{\Delta I_O}. \]  

(2.8)

where \( I_o \) is the output current. As soon as the inductance is known the peak values of the inductor’s currents can be found:

\[ I_{L1\text{max}} = I_O \cdot \frac{V_O + V_D}{V_{\text{INmin}}} \left( 1 + \frac{0.4}{2} \right) \]  

and

\[ I_{L2\text{max}} = I_O \left( 1 + \frac{0.4}{2} \right). \]  

(2.9)

(2.10)

**MOSFET Selection for Switch**

The switch is selected taking into account the maximal voltage which is \( V_{\text{INmax}} + V_{O\text{max}} \), maximal RMS current found as

\[ I_{\text{SMS}} = I_O \cdot \sqrt{\frac{V_O + V_{\text{INmin}} + V_D}{V_{\text{INmin}}}} \left( V_O + V_D \right) \]  

(2.11)

and maximal dissipated power

\[ P_{\text{Smax}} = I_{\text{SMS}}^2 \cdot R_{\text{DSon}} \cdot D_{\text{max}} \cdot \left( V_{\text{INmin}} + V_O \right) \cdot I_{\text{Smax}} \cdot \frac{Q_{GD}}{T \left( V_G/R_G \right)}, \]  

(2.12)

where: \( R_{\text{DSon}} \) is on-state resistance of the MOSFET at the gate voltage \( V_G \), \( R_G \) – total gate resistance, \( Q_{GD} \) – gate-drain charge, but \( I_{\text{Smax}} \) – peak current of the switch equal to the peak current \( I_{L1\text{max}} \).

**Diode Selection**

The diode is selected for the sum of peak input and output voltages \( V_{\text{INmax}} + V_{O\text{max}} \), the maximal output current \( I_{O\text{max}} \) (equal to diode’s average current) and for power losses (that could roughly be estimated as \( P_{\text{Smax}} \)).
Figure 2.2 Current ripples in output capacitor of SEPIC converter: a) schematic of the output node; b) output capacitor's current

**Coupling Capacitor Selection**

This capacitor is selected for the maximal input voltage $V_{INmax}$, maximal RMS current

$$I_{C_{ms}} = I_O \cdot \sqrt{\frac{V_O + V_D}{V_{INmin}}} \quad (2.13)$$

and maximal voltage ripples

$$\Delta V_C = \frac{I_O \cdot D_{max} \cdot T}{C} \leq 0.01 V_{IN} \quad (2.14)$$

**Output Capacitor Selection**

In the similar way, the output capacitor is selected for the maximal output voltage $V_{Omax}$, maximal RMS current

$$I_{C_{Oms}} = I_O \cdot \sqrt{\frac{V_O + V_D}{V_{INmin}}} \quad (2.15)$$

and maximal voltage ripples

$$\Delta V_O = \frac{I_O \cdot D_{max} \cdot T}{0.5 \cdot C_O} \leq 0.01 V_O \quad (2.16)$$

Since the current through this capacitor due to the output diode (Figure 2.2-a) is pulse mode current (Figure 2.2-b) the equivalent series resistance is also important for this capacitor:

$$R_{ESR} \leq \frac{I_O \cdot D_{min} \cdot T}{0.5 \cdot 0.01 V_O} \quad (2.17)$$

**SEPIC Converter of Training Kit**

The provided training kit includes a SEPIC converter containing 4 parallel branches. It is supplied for a laboratory power supply of 15...18V. The load – 15 branches of 6 LEDs of 3.2Vx0.12A and a ballast resistor of the same voltage drop. Therefore $V_{Omax} = 21V$, $I_{Omax} = 1.8A$. The corresponding elements of SEPIC converter are: MOSFET – PHT4NO10T, diode – STTH2R02A, L1 and L2 – 820μHx1.5A, $C_O$ 2x270μFx63V and C – 10μFx25V. See “Appendix C: Schematics of FPGA Training Kit” section “SEPIC Module” on page 37 for more details.
3. Interleaved Control of Power Converters

Basics of Interleaved Control

Voltage fed single switch power converters usually contain an inductor as a main energy transfer element. Inductor’s current in such configuration is triangle shaped – it is rising when the switch is “on” and falling – when it is “off”. The value of these ripples of the inductor current in a traditional single branch configuration depends only on the inductance, input voltage and duty cycle. The idea of the interleaved control assumes that the total inductor’s current is split between several parallel branches. Each of the branches has its own transistor and freewheeling diode. At the same time the beginnings of their periods are shifted so, that rising current edges in one branch compensate current falling edges in the others. Then the overall value of the current ripples may be lower.

![Image of interleaved control principle](image)

**Figure 3.1** Principle of interleaved operation of two parallel branches: a) \( D < \text{SHIFT} \); b) \( D < \text{SHIFT} \); c) \( D = \text{SHIFT} \); d) \( D > \text{SHIFT} \)
If the number of branches is BR then the optimal shift can be found as
\[
\text{SHIFT} = \frac{100\%}{\text{BR}}.
\] (3.1)
The optimal value of the duty cycle is either the same or can be found as
\[
D_{\text{OPT}} = 100\% - \text{SHIFT}.
\] (3.2)

The closer the duty cycle is to its optimal value the lower are the current ripples. At the optimal values of the duty cycle the current ripples are 0%.

The interleaved principle of control is applied to parallel branches (converters) if their inductors are placed at the input and if well smoothed, DC like input current is required. This is quite important, for example, if the converter is supplied from a photovoltaic element or a fuel cell requiring constant load current to keep them at optimal operation point and to achieve the maximal power. The converters with inductor at the input (boost, Čuk, SEPIC etc.) with parallel branches and interleaved control provide such current without significant input filter (or with a small one).

The interleaved principle also allows to reduce the size of the output capacitor if the inductor is placed at the output (Figure 3.2-a). The voltage ripples of the output capacitor are always proportional to the integrated capacitor’s current (charge). For the converters with an inductor at the output (buck, Čuk, ZETA etc.) it can be found as a difference of the output DC current and triangle current of the inductor (Figure 3.2-b). Then the lower inductor current ripples directly lead to lower output voltage ripples (Figure 3.2-c). Therefore the interleaved principle allows using smaller capacitors at the output.

Note that Čuk converter has inductors both at the input and output. That is why the interleaved principle equally forms its input and output currents more smoothed.

---

**Figure 3.2 Impact of current ripples on the voltage ripples:**

- **a)** schematic of the output node
- **b)** output capacitor’s current
- **c)** output capacitor’s voltage
Obtaining the Interleaved Control Signals from Programmable Logic

Interleaved control of parallel power branches is possible with the help of shifted pulse mode signals. In order to design a generator of several pulse mode signals with shift an ordinary generator has to be built at first.

Common (Single Channel) Pulse Width Modulation

Figure 3.3 Ordinary single channel PWM signal generator: a) generalized diagram; b) typical example of digital implementation; c) algorithm of counter operation

Figure 3.4 Operation diagram of single channel PWM signal generator
Pulse Width Modulation (PWM) is a pulse signal of constant frequency the average value of which is controlled via its pulse width. A PWM signal can be obtained at the output of a comparator the inputs of which are tied to a sawtooth (or triangle) signal generator, as well as to a source of reference (Figure 3.3-a). In a digital system (Figure 3.3-b) the comparator is a digital device – subtraction unit which carry bit serves “greater” output. In turn the sawtooth signal generator is an up counter, but the triangle signal generator – up-down counter.

PLDs are usually synchronized with high frequency clock. On the other hand PWM signals which are applied to electronic switches are of values x10kHz…x100kHz. Therefore a prescaler (preliminary divider) has to be utilized. For instance, if the clock is 80MHz, but the required signal is 1Hz PWM (Figure 3.3-c for demonstration purposes) at 0.1% accuracy then the counting frequency must be 1000Hz which requires prescaler 80000.

The structure shown in Figure 3.3 can be used as a basis for a PWM generator designed with mixed (schematic + VHDL) approach. The VHDL description of the counter is given in Example 3-1, but those of the comparator – in Example 3-2. Note, that both blocks are synchronous for more stable operation (although the comparator could be asynchronous). Also note that there is no necessity to reset the prescaler if it reaches its maximum (it happens automatically).

Example 3-1 VHDL script for up-counter with reset and prescaler

```
---------------------- Counter Declaration ----------------------
ENTITY sawtooth IS
  generic (MIN_COUNT: natural :=0; MAX_COUNT: natural :=4095);
  port (clk: in std_logic; reset: in std_logic;
       q: out integer range MIN_COUNT to MAX_COUNT);
END sawtooth;
---------------------- Counter Body ----------------------
ARCHITECTURE sawtooth_architecture OF sawtooth IS
BEGIN
  process (clk)
  variable Prsc: integer range MIN_COUNT to 80000;
  variable Cntr: integer range MIN_COUNT to 1000;
  begin
    if (rising_edge(clk)) then
      Prsc := Prsc + 1;
      if (Prsc=80000) then
        Cntr := Cntr + 1;
      end if;
      if (reset='1') then
        Prsc:=0;
        Cntr:=0;
      end if;
      end if;
      q <= Cntr;
    end process;
```
END sawtooth_architecture;

Example 3-2 VHDL script for synchronous digital comparator

------------------ Digital Comparator Declaration ------------------
ENTITY comparator IS
  port (A: IN STD_LOGIC_VECTOR(11 downto 0);
        B: IN STD_LOGIC_VECTOR(11 downto 0);
        clk: IN STD_LOGIC;
        Switch: OUT STD_LOGIC);
END comparator;
------------------ Digital Comparator Body ---------------------
ARCHITECTURE comparator_architecture OF comparator IS
BEGIN
  process (clk)
  begin
    if (rising_edge(clk)) then
      if (A > B) then Switch<='1'; else Switch<='0'; end if;
    end if;
  end process;
END comparator_architecture;

Multiple Channel Pulse Width Modulation with Shift

The most direct way to generate several shifted PWM signals is to obtain them from several comparators tied to the same reference, but to different shifted sawtooth or triangle signals. A three-channel example is shown in Figure 3.5-a. The corresponding digital system has three comparators and three counters with a common clock. This approach is obviously very extensive if the counters are independent. Then it requires as many times more flip-flops as many channels have to be provided.
Figure 3.5 Multiple channel shifted PWM signals generator: a) generalized diagram; b) typical example of digital implementation; c) operation diagram

However, there is a different design approach that utilizes only one counter and adds the proper shifts just on the corresponding output (Figure 3.5-b). At the same time this approach has very simple VHDL description. All the considerations taking place in the case of single channel PWM are valid also in the case of the described multi-channel generator. Certain prescaler is necessary. Also the counter starts the next cycle of counting automatically and does not need to be watched and reloaded. The same regards also the phase shifts. In VHDL it is possible just to add the corresponding constant to the counted value (Example 3-3).

Example 3-3 VHDL script for 3 shifted up-counters with reset and prescaler

```
ENTITY sawtooth IS
  generic  (MIN_COUNT: natural :=0;
             MAX_COUNT: natural :=4095;
             SHIFT := 333);
  port     (clk: in std_logic; reset: in std_logic;
            q1: out integer range MIN_COUNT to MAX_COUNT;
            q2: out integer range MIN_COUNT to MAX_COUNT;
            q3: out integer range MIN_COUNT to MAX_COUNT);
END sawtooth;

ARCHITECTURE sawtooth_architecture OF sawtooth IS
BEGIN
  process (clk)
  begin
    variable Prsc: integer range MIN_COUNT to 80000;
    variable Cntr1: integer range MIN_COUNT to 1000;
    variable Cntr2: integer range MIN_COUNT to 1000;
    variable Cntr3: integer range MIN_COUNT to 1000;
    if (rising_edge(clk)) then
      Prsc := Prsc + 1;
  end process;
```
if (Prsc=80000) then
    Cntr1 := Cntr1 + 1;
    Cntr2 := Cntr1 + SHIFT;
    Cntr3 := Cntr2 + SHIFT;
end if;
if (reset='1') then
    Prsc:=0;
    Cntr1:=0;
    Cntr2:=Cntr1+SHIFT;
    Cntr3:=Cntr2+SHIFT;
end if;
end if;
q <= Cntr;
end process;
END sawtooth_architecture;
4. Getting Analog Data through SPI Interface

Main Features of SPI

SPI (Serial Peripheral interface) provides a synchronous serial communication between two (usually) or more (rarely) complex digital devices, like microcontroller, ADC, DAC etc. One of the connected devices is “master” – which provides clock and select signals to “slaves”, as well as configures them and receives from them the requested data. In turn, the “slaves” provide the data at the changes of the clock signal, as well as receive and process the configuration codes.

The interface contains 4 signals Figure 4.1-a: 1) SCLK (alternatively DCLOCK, CLK, SCK) – clock signal; 2) MOSI (DO, SDO, DOUT) – master-to-slave data; 3) MISO (DI, SDI, DIN) – slave-to-master data; 4) SS (CS) – slave select. The interface is activated with falling edge of SS. Then data are acquired and changed in both data lines (MOSI and MISO) at the SCLK changes, i.e. SPI utilizes a full duplex data transmission. In fact SPI interface can be regarded as a shift register that operates under the control of SCLK signal (Figure 4.1-b).

<table>
<thead>
<tr>
<th>SPI</th>
<th>SCLK</th>
<th>MOSI</th>
<th>MISO</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPI</td>
<td>SCLK</td>
<td>MOSI</td>
<td>MISO</td>
<td>SS</td>
</tr>
<tr>
<td>SLAVE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.1 Main connections within SPI: a) external connections; b) internal elements**

There are 4 possible types of SPI operation depending on the interface design options CPOL and CPHA.

The CPOL defines which the “main” value of SCLK signal is. If 0 then SCLK=0 during the idle mode (when SPI is inactive) and the first edge is rising edge, but the second one – falling. At 1 the idle value is 1, the first edge is falling, but the second - rising.

In turn, CPHA defines at which edge the data readings occur. If CPHA=0 they occur at the first edge, which is rising if CPOL=0 (Figure 4.2-a) and falling if CPOL=1 (Figure 4.2-b). Then some valid data must appear on MOSI and MISO just after activation of the interface (falling edge of slave select signal SS).

With CPHA=1 readings occur at the second edge, i.e. at the falling edge if CPOL=0 (Figure 4.2-c) and rising edge if CPOL=1 (Figure 4.2-d).
Figure 4.2 Operation modes of SPI: a) mode 0 (CPOL=0, CPHA=0); b) mode 1 (CPOL=1, CPHA=0); mode 2 (CPOL=0, CPHA=1); mode 3 (CPOL=1, CPHA=1);

Multiple slave operation is possible with SPI. There are two options. The first one is an independent slave connection. Then SCLK, MOSI and MISO signals are common for all SPI devices, but there is a dedicated SS signal per each slave (Figure 4.3-a). With this connection the master can communicate only to one slave at once.

Another option is daisy chain. Then SCLK and SS signals are common while master’s MOSI is connected to the 1st slave MOSI and master’s MISO to the last slave MISO. The connections of other slaves form a chain where each previous slave’s MISO connected to the next slave’s MOSI Figure 4.3-b. The daisy chain configurations allow communication to all slaves within the same data exchange cycle, but very rare devices allow this configuration (because do not provide 1 - multiple transfer per activation, 2 - slaves capability of sending previously received data).

Figure 4.3 Multiple slave SPI operation: a) parallel; b) daisy chain
Communication of Programmable Logic and SPI

The simplest occasion of SPI communication is repetitive data exchange with continuous (constant frequency) clock signal. It has the following features:
1) Scheme of communication can be regarded as “clock-slave-slave”, but not “master-slave”;
2) Both devices can be described as finite state machines (a synchronous digital system with limited number of states which are toggled on changes of synchronisation signal – SPI clock in the given case);
3) Programmable logic device can be easy configured from a VHDL script utilizing operator “process”, but the digital entity may have standard logic signal corresponding to SPI interface, an integer or real output V and an integer or real internal variable T to store temporary result; an option is a standard logic output of communication error.

Below two examples of such operation/configuration are described.

Data read from ADCS7476

IC ADCS7476 is a monolithic single channel 12-bit ADC. It does not need any configuration and, besides its clock SCLK and select CS=SS inputs, has only data output SDATA which is equivalent of MISO (Figure 4.4-a). The use of this IC assumes that all modifications occur at the falling edge, but readings – at the rising edge of SCLK which normally (in the idle mode) has to be high (CPOL=1, CPHA=1, SPI mode 3). Taking into account that the number of data bits is 12, number of zero-bits with no meaning is 3, but the minimal number of clock pulses during the idle state is 1, as well as taking into account that there must a delay between the chip select and first reading finally the data transmission cycle can occur in 17 periods of SCLK signal. These intervals can be divided into 4 groups (Figure 4.5):
1) Idle interval in which CS is kept high (1), but the output signal V of the “process” operator is rewritten from its internal temporary variable T;
2) Start interval when CS is set low (0);

![Diagram](image-url)
3) Zero reading intervals at which the temporary variable $T$ is reset;
4) Data reading intervals at which the temporary variable $T$ is updated with the data received from SDATA.

Note that the output signal $V$ of the “process” operator within the most of the cycle remains unchanged and is updated only at the beginning of the idle interval. In contrast, the internal temporary variable $T$ with this approach within the most of the cycle is updated or being reset (Figure 4.6). Also note that with this approach data are acquired from ADCS7476 continuously and the new cycle starts as soon as the previous ends of course as long as SCLK persists.

![State diagram](image)

**Figure 4.5 State diagram of unidirectional data transfer from ADCS7476 to programmable logic device (reading states – green, service states - orange)**

![Timing diagram](image)

**Figure 4.6 Timing diagram of unidirectional data transfer from ADCS7476 to programmable logic device**
Device configuration and data read from MCP3004 (MCP3008)

Another example is IC MCP3004 – monolithic 4-channel 10-bit ADC. Since this IC not only capable of reading analog data from several channels, but also can operate in one of two measuring modes (single channel or differential – in this mode it measures the difference between two analog signals) is has to be configured before the measurement and is equipped with complete 4-wire (CLK=SCLK, DOUT=MISO, DIN=MOSI, SS=CS) SPI interface (Figure 4.4-b).

---

**Figure 4.7 State diagram of bidirectional data transfer between MCP3004 and programmable logic device (reading states – green, writing states – red, service states – orange)**

---
Operation cycle remains similar because it is also a mode 3 SPI device (CPOL=1, CPHA=1) – so that readings occur at the rising edge, but writings – at the falling. Although the number of data bits (10 as against 12) and null bits (1/3) is lower the total number of time intervals in the data exchange cycle is almost the same (19/17). Nevertheless new types of these intervals make the corresponding state diagram more complicated (Figure 4.7). Now it includes the following types of intervals and states:

1) Idle interval (S0) in which CS is kept high (1), the output signal V is rewritten from the temporary variable T; So this interval remains similar like in the previous example;

2) The start interval (S1) now together with CS=0 sets also DIN=1 indicating the start of data writing into MCP3004;

3) In the next interval (S2) the measuring mode is programmed by setting DIN=0 for single channel measurements or DIN=1 for differential measurements;

4) In the following 3 intervals (S3...S5) the channel is chosen by programming DIN with the appropriate value; For example DIN is always 0 if the data have to be acquired from the channel 0; Interval S3 is useful with MCP3008 which has 8 channels and needs 3 bits of address;

5) Note that the intervals S1...S5 starts and ends at the falling edge of the CLK that corresponds to writings; At the same time the next intervals are reading intervals that starts and ends at the rising edge; For this reason in the cycle appears waiting interval S6 that starts at the falling edge of CLK, ends at its rising edge and lasts for a half of a period of CLK;

6) The next waiting interval S7 is complete; It starts and ends at the rising edge of CLK;

7) As soon as the IC is configured and after some delay the data can be read from the IC; However, the first bit read from the IC (state S8) is not a valuable bit and has to be 0; if it is 0 the temporary variable T is reset and is ready to receive data;

8) S9...S17 are the data reading intervals at which the temporary variable is updated with the data received from DOUT;

9) So the S18 is also data reading and variable T updating interval; However, this interval only starts at the rising edge of CLK, but ends – at the falling and lasts one half of the CLK period; This, second (the first is S6), “half” interval is necessary to keep continuity of the clock and to make the complete length of the data exchange cycle equal to integer number of CLK periods (19).

Note that, like in the previous example, the output signal V of the “process” operator the most of the cycle remains unchanged and is updated only in the idle interval (S0). The internal temporary variable T in this example is kept unchanged while the master configures the IC. Then it is reset and updated (Figure 4.8). Also note that, like in the previous example, data are acquired from IC continuously and the new cycle starts as soon as the previous ends.
5. Features of P and PI regulators made with FPGA

The rest of the regulation system includes an error detector and a regulator itself. The error detector just finds a difference of a 12-bit number coming from the generator of reference (that in the explored test bench it could be DIP-switches) and a 10-bit voltage feedback – measurements obtained through SPI interface. The most significant features of the corresponding VHDL script (Example 5-1) are: 1) operations with signed numbers and 2) the equalization of the operands at the length of 12 bit with the help of “resize” function.

Example 5-1 VHDL script for error detector

```vhdl
ENTITY ErrorDetector IS
    PORT (
        dataa : IN std_logic_vector(9 downto 0);
        datab : IN signed(11 downto 0);
        res_error : OUT SIGNED (11 downto 0)
    );
END ErrorDetector;

ARCHITECTURE Architecture_ErrorDetector OF ErrorDetector IS
BEGIN
    res_error <= signed(resize(unsigned(dataa), 12)) - datab;
END Architecture_ErrorDetector;
```
The last, but not the least part of the regulation system is a regulator with the function providing the control command based on the calculated error.

\[ \text{CMD} = \text{CMD}_p + \text{CMD}_i + \text{CMD}_d, \quad (3.3) \]

where

\[ \text{CMD}_p = K_p \cdot \Delta V_o, \quad (3.4) \]
\[ \text{CMD}_i = \frac{1}{T_i} \int_0^1 \Delta V_o \, dt \quad \text{and} \]
\[ \text{CMD}_d = T_d \frac{d\Delta V_o}{dt}. \quad (3.6) \]

In (3.4), (3.5) and (3.6) \( K_p \) – is the proportional gain which expresses the direct amplification rate of the calculated reference and feedback error \( \Delta V_o \), \( T_i \) – is the integration period in which the command coming through the integral stage is becoming equal to those of proportional stage (without taking into account \( K_p \)) and \( T_d \) is differential time constant.

The most popular regulator is proportionally-integral. In practice such regulators are designed based on digital technique. Then the integral becomes a sum of time steps and the values of the error. Then the finite differences equation for PI regulator after \( M \) measurements is the following:

\[ \text{CMD} = K_p \cdot \Delta V_{ON} + \frac{1}{T_i} \sum_{k=1}^{M} \Delta V_{\text{CK}} \Delta t_k . \quad (3.7) \]

If \( \Delta t \) is constant (3.7) can be rewritten as

\[ \text{CMD} = K_p \cdot \Delta V_{ON} + \frac{1}{N} \sum_{k=1}^{M} \Delta V_{\text{CK}} . \quad (3.8) \]

(3.8) is a practical formula usable in real calculations. A version of its realization with VHDL is given below as Example 5-2.

**Example 5-2 VHDL Script for PI regulator**

```vhdl
ENTITY PI IS
  generic
    (
      DATA_WIDTH : natural := 12
    );
  port
    (
      input : in signed ((DATA_WIDTH-1) downto 0);
      clk   : in std_logic;
      reset : in std_logic;
      output : out signed ((2*DATA_WIDTH-1) downto 0)
    );
END PI;
```
------------------------ PI Regulator Body ------------------------
ARCHITECTURE PI_architecture OF PI IS
    signal sum : signed ((2*DATA_WIDTH-1) downto 0);
    signal old_sum : signed ((2*DATA_WIDTH-1) downto 0);
    signal out_reg : signed ((2*DATA_WIDTH-1) downto 0);
    signal in_reg : signed ((DATA_WIDTH-1) downto 0);
    constant TIME_CONSTANT : signed ((DATA_WIDTH-1) downto 0)
        := "000000001000";
    constant P : signed ((DATA_WIDTH-1) downto 0)
        := "000000001000";
BEGIN
    process (sum)
    begin
        old_sum <= sum;
    end process;
    process (clk)
    begin
        if (rising_edge(clk)) then
            if reset = '1' then
                sum <= "00000000000000000000000000000000";
            else
                in_reg <= input;
                sum <= old_sum + "000000000001"*in_reg/TIME_CONSTANT;
                out_reg <= in_reg/P + sum;
            end if;
        end if;
    end process;
    output<=out_reg;
END PI_architecture;
Appendix A: VHDL Quick Reference

Based on [5] and [6].

1. Basic Definitions

VHDL is not case sensitive that can be used for formatting. VHDL constructions are hardly defined by its lexical elements. That is why lines can be of any length, but long constructions can occupy few lines. The most of the constructions end with semicolon “;”. Comments start with “--” and are valid till the end of the line.

VHDL script often starts from a library and package declaration

```vhdl
library <LIBRARY_NAME>;
use     <LIBRARY_NAME>.<PACKAGE_NAME>.<MODULE_NAME>;
```

Frequently used:

```vhdl
library ieee;
and
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
use ieee.numeric_std.all;
```

VHDL script must contain declaration and description of at least one digital entity. The entity is declared as

```vhdl
entity <NAME_OF_ENTITY> is
  [generic (  
    <CONSTANT_NAME1>: <TYPE1> [:=<VALUE1>] ;  
    ...  
    <CONSTANT_NAMEEn>: <TYPEn> [:=<VALUEn>] );]
  port (  
    <PORT_NAMES1>: <MODE1> <TYPE1>;  
    ...  
    <PORT_NAMESn>: <MODEn> <TYPEn>);
end [<NAME_OF_ENTITY>];
```

The description of the entity is declared as

```vhdl
architecture <ARCHITECTURE_NAME> of <NAME_OF_ENTITY> is
  -- declarations
begin
  -- concurrent statements (describes the design)
end [architecture] <ARCHITECTURE_NAME>;
```

where “declarations” are given as in section 4, but concurrent statements as in section 6.

2. Operators

Logical operators:

```vhdl
and, or, xor, nand, nor, xnor
```
Relational operators:
=, /=, <, <=, >, >=

Shift left/right logical operators (vacated bits filled with zeros):
sll, srl

Shift left/right arithmetic operators (vacated bits stay with previous values):
sla, sra

Rotate left/right operators (vacated bits filled with exiting bits):
rol, ror

Arithmetical operators:
+, -, *, /, **

Concatenation operator (‘111’&’000’=’111000’):
&

Reminder operator (10 rem 3 = 1):
rem

Division modulo operator (7 mod 4 = 3; -7 mod -4 = -3; 7 mod -4 = -1)
mod

3. Frequently Used Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>‘0’ and ‘1’</td>
</tr>
<tr>
<td>bit_vector</td>
<td>array of ‘0’ and ‘1’</td>
</tr>
<tr>
<td>std_ulogic</td>
<td>‘U’ (uninitialized), ‘X’ (forcing unknown), ‘0’ (forcing 0), ‘1’ (forcing 1), ‘Z’ (high impedance), ‘W’ (weak unknown), ‘L’ (weak 0), ‘H’ (weak 1), as well as ‘-‘ (don’t care) (in std_logic_1164 package)</td>
</tr>
<tr>
<td>std_logic</td>
<td>array of std_ulogic (in std_logic_1164 package)</td>
</tr>
<tr>
<td>std_ulogic_vector</td>
<td>array of std_ulogic</td>
</tr>
<tr>
<td>std_logic_vector</td>
<td>array of characters</td>
</tr>
<tr>
<td>Boolean</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>character</td>
<td>7-bit ASCII</td>
</tr>
<tr>
<td>string</td>
<td>array of characters</td>
</tr>
<tr>
<td>integer</td>
<td>signed 32 bit at least</td>
</tr>
<tr>
<td>natural</td>
<td>integer ≥ 0</td>
</tr>
<tr>
<td>positive</td>
<td>integer &gt; 0</td>
</tr>
<tr>
<td>real</td>
<td>floating point 32 bit, +1e38...-1e38</td>
</tr>
</tbody>
</table>

4. Declarations

7.1 Declaration of scalar data types

type <TYPE_IDENTIFIER> is range <MIN> to <MAX>;
type <TYPE_IDENTIFIER> is range <MAX> downto <MIN>;
type <TYPE_IDENTIFIER> is (<list_of_values>);

for example

type SMALL_INT is range 0 to 1024;
type MY_WORD_LENGTH is range 31 downto 0;
type MY_VALUES is ('0', '1', 'Z');

7.2 Declaration of array data types
type <ARRAY_ID> is array (<indexing scheme>) of <TYPE_ID>;
for example
type WORD16A is array (15 downto 0) of std_logic;
type WORD16D is array (0 to 15) of std_logic;

7.3 Declaration of constants
constant <LIST_OF_NAMES_OF_CONSTANT>: <type> [:= <INITIAL_VALUE>];
for example
constant PERIOD: time := 10 us;
constant MD_BUS, MA_BUS: integer := 16;

7.4 Declaration of variables
variable <LIST_OF_VARIABLES>: type [:= <INITIAL_VALUE>];
for example
variable VAR_BIT: bit := 0;
variable VAR_BOOLEAN: boolean := FALSE;
variable VAR_INTEGER: integer := 1000;
variable CNTR: integer range 0 to 15;
variable VAR_BIT_V: bit_vector (3 downto 0);

7.5 Declaration of signals
signal <LIST_OF_SIGNALS>: <type> [ := <INITIAL_VALUE> ] ;
for example
signal A_NOT, A_NOT, INT_1, INT_2: std_logic;
signal MY_VALUE: integer := 0;
signal CNTR: integer range 0 to 15;
signal MD_BUS: bit_vector (0 to 15);

7.6 Declaration of components – digital entities described in the same file
component <COMPONENT_NAME> [is]
    [port (<port_name_1>: mode <type>);
     <port_name_2>: mode <type>;
     ... ]
end component [<COMPONENT_NAME>];

for example
component AND3
    port (IN1, IN2, IN3: in std_logic;
          OUTx: out std_logic);
end component;

5. Some Attributes

Length of array (an integer value)
<ARRAY_ID>'length

Signal delayed by T (also a signal)
<SIGNAL_ID>'delayed(T)

Value of driver (also a signal)
<SIGNAL_ID>'driving_value

Event (rising or falling edge) detector (TRUE or FALSE)
<SIGNAL_ID>'event

6. Statements

9.1 Concurrent statements

Statement "process"

[<PROCESS_LABEL>:] process [<sensitivity_list>] [is]
    [<process_declarations>]
begin
    <list of sequential statements such as:
    signal assignments
    variable assignments
    case statement
    exit statement
    if statement
    loop statement
    next statement
    null statement
    procedure call
    wait statement>
end process [<PROCESS_LABEL>];

for example
process(EnableSignal, DataWord)
begin
    if (EnableSignal = '1') then
        LatchVariable <= DataWord;
    end if;
end process;

Instance of component
Example of component with explicit signal assignment

DD1: AND3 port map (OUTx=>Prod, IN1=>InA, IN2=>InB, IN1=>InvInA)

Example of component with implicit signal assignment

DD1: AND3 port map (InA, InB, InvInA, Prod)

Unconditional concurrent signal assignments

<TARGET_SIGNAL> <= <expression> [after <delay>];

For example

MySignal <= '1';
MySignal <= '1' after 10ns;

Conditional concurrent signal assignments

<TARGET_SIGNAL> <=
    <EXPRESSION1> when <CONDITION1> else
    <EXPRESSION2> when <CONDITION2> else
    ...
    <EXPRESSIONn>;

For example

Q <=
    A when SEL="00" else
    B when SEL="01" else
    C when SEL="10" else
    D when SEL="11" else
    'X';

Selective concurrent signal assignments

with <CHOICE_EXPRESSION> select
    <TARGET_SIGNAL> <=
        <EXPRESSION1> when <CHOICE1>,
        <EXPRESSION2> when <CHOICE2>,
        ...
        [<EXPRESSIONn> when others];

For example

with SEL select
    Q <=
        A when "00",
        B when "01",
        C when "10",
        D when "11",
        'X' when others;
9.2 Sequential statements

Waiting statement

wait until <CONDITION>;
wait for <TIME>;
wait on <SIGNAL1> [,<SIGNAL2> ...];
wait;

for example

wait until CNTR=10;
wait until CLK='0';
wait until CLK'EVENT and CLK='0';
wait until CLK'STABLE(10ms) and CLK='1';
wait for 1s

Conditional statement

if <condition 1> then <sequential statements 1...>
   ...>
[elsif <condition k> then <sequential statements k...>
   ...>] 
[else <sequential statements N>]
end if;

for example

if (CLEAR = '1') then
   INT_Q := '0';
elsif (CLK'EVENT and CLK = '1') then
   INT_Q := D;
end if;

Choice statement

case <expression> is
   when <choice 1> => <sequential statements 1>
   ...
   when <choice k> => <sequential statements k>
   ...
   [when others => <sequential statements N>]
end case;

for example

case SEL is
   when "00" => TMP := A;
   when "01" => TMP := B;
   when "10" => TMP := C;
   when "11" => TMP := D;
   when others => TMP := 'X';
end case;
Conditional loop statement

```plaintext
[<LOOP_LABEL>:] while <condition> loop
   -- sequential statements
   [next [<LOOP_LABEL>]] [when <condition_n>];
   [exit [<LOOP_LABEL>]] [when <condition_e>];
end loop [<LOOP_LABEL>];
```

for example

```plaintext
while ( LoopVar < 21 ) loop
   Sum:=Sum+LoopVar;
   LoopVar:=LoopVar+1;
end loop;
```

Counted loop statement

```plaintext
[<LOOP_LABEL>:] for <IDENTIFIER> in <range> loop
   -- sequential statements
   [next [<LOOP_LABEL>]] [when <condition_n>];
   [exit [<LOOP_LABEL>]] [when <condition_e>];
end loop [<LOOP_LABEL>];
```

for example

```plaintext
for LoopVar in (0 to 20) loop
   Sum:=Sum+LoopVar;
end loop;
```
Appendix B: Layout and Connections of FPGA Training Kit

Layout of Cyclone III FPGA Training Kit

Connections of Cyclone III FPGA Training Kit

<table>
<thead>
<tr>
<th>OUTPUTS</th>
<th>INPUTS</th>
<th>CONVERTER &amp; EXTERNAL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LEDs</strong></td>
<td><strong>PUSHBUTTONS</strong></td>
<td><strong>TO MOSFETs</strong></td>
</tr>
<tr>
<td><strong>Function</strong></td>
<td><strong>FPGA</strong></td>
<td><strong>Active</strong></td>
</tr>
<tr>
<td>LED1 (right)</td>
<td>73</td>
<td>Low</td>
</tr>
<tr>
<td>LED2</td>
<td>74</td>
<td>Low</td>
</tr>
<tr>
<td>LED3</td>
<td>67</td>
<td>Low</td>
</tr>
<tr>
<td>LED4 (left)</td>
<td>68</td>
<td>Low</td>
</tr>
</tbody>
</table>

| **DISPLAY** | **DIP SWITCHES** | **ADC via SPI** |
| **Function** | **FPGA** | **Active** | **Function** | **FPGA** | **Active** | **Function** | **FPGA** | **Active** |
| Segment A | 60 | Low | Switch 1 | 115 | High | DOUT (MISO) | 99 | High |
| Segment B | 66 | Low | Switch 2 | 121 | High | DIN (MOSI) | 104 | High |
| Segment C | 58 | Low | Switch 3 | 125 | High | nCS (nSS) | 110 | High |
| Segment D | 54 | Low | Switch 4 | 127 | High | CLK (SCLK) | 112 | High |
| Segment E | 53 | Low | Switch 5 | 129 | High |                  |      |      |
| Segment F | 64 | Low | Switch 6 | 141 | High |                  |      |      |
| Segment DP | 55 | Low | Switch 7 | 135 | High |                  |      |      |
| IND1 (right) | 77 | Low | Switch 8 | 128 | High |                  |      |      |
| IND2 | 76 | Low | Switch 9 | 126 | High |                  |      |      |
| IND3 | 75 | Low | Switch 10 | 124 | High |                  |      |      |
| IND4 (left) | 79 | Low | Switch 11 | 120 | High |                  |      |      |
| IND4 (left) | 79 | Low | Switch 12 | 114 | High |                  |      |      |

| **FPGA outputs** | **FPGA inputs** | **FPGA inputs/outputs** |
| **FUNCTIONS** | **FPGA** | **Active** | **Function** | **FPGA** | **Active** | **Function** | **FPGA** | **Active** |
| BIT0 (X1, LSB) | 113 | High | BIT1 (X2) | 111 | High | BIT2 (X4) | 106 | High |
| BIT3 (X8, MSB) | 103 | High | Clock 80MHz | 23 | High | PIN 1 (VDD) | 3.3V | – |
| PIN 2 (GND) | – | – | PIN 3 (GPIO0) | 34 | High | PIN 4 (GPIO1) | 32 | High |
| PIN 5 (GPIO2) | 28 | High | PIN 6 (GPIO3) | 10 | High | PIN 7 (GPIO4) | 3 | High |
| PIN 8 (GPIO5) | 1 | High | PIN 9 (GPIO6) | 2 | High | PIN 10 (GPIO7) | 4 | High |
| PIN 11 (GPIO8) | 11 | High | PIN 12 (GPIO9) | 30 | High | PIN 13 (GPIO10) | 33 | High |
| PIN 14 (GPIO11) | 38 | High |                  |      |      |                  |      |      |

Page 34 of 38
Appendix C: Schematics of FPGA Training Kit

Main Board
FPGA Cyclone III Target Board

FPGA Based Control Solutions for Specific Power Converters
SEPIC Module

R2 10k
C4 270μF-63V
VD1 STTH2R02A
J3 CON2
1 2
J1 CON2
1 2
C3 270μF-63V
R1 10R
L11 820μH
C1 47μF-25V
L21 820μH
VT1 PHT4NQ10T
J2 CON2
1 2
C2 10μF-25V
R2 10k
C4 270μF-63V
VD1 STTH2R02A
J3 CON2
1 2
J1 CON2
1 2
C3 270μF-63V
R1 10R
L11 820μH
C1 47μF-25V
L21 820μH
VT1 PHT4NQ10T
J2 CON2
1 2
C2 10μF-25V
References


