University Bremerhaven

Course Documentation

Embedded System Design Example (Class/Lab) [ES–STP]

- Part 1: Modeling
- Part 2: Programmable Logic
- Part 3: Processing System
- Part 4: Simple Motion Control
- Part 5: Continuous Motion
- Part 6: Experimental System

Revision: V0.0a

Release: February 2013

Prof. Dr.-Ing. Kai Mueller
University of Applied Sciences Bremerhaven
Institute for Automation and Electrical Engineering
An der Karlstadt 8

D–27568 Bremerhaven / Germany

Phone: +49 471 48 23 – 415
FAX: +49 471 48 23 – 555

Email: kmueller@hs–bremerhaven.de

I Contents

Embedded System Design Example

1 Preface

This design example demonstrates how to create an embedded system with modern DSP solutions in programmable logic and a processing system.

Electrical drives are often essential parts of embedded ISM devices. In this example we will design a digital subsystem to control a small drive which can be used in a medical infusion pump.

Fig. 1.1-1.4 show the historic development of embedded systems in the industrial/scientific/medical area (ISM market).

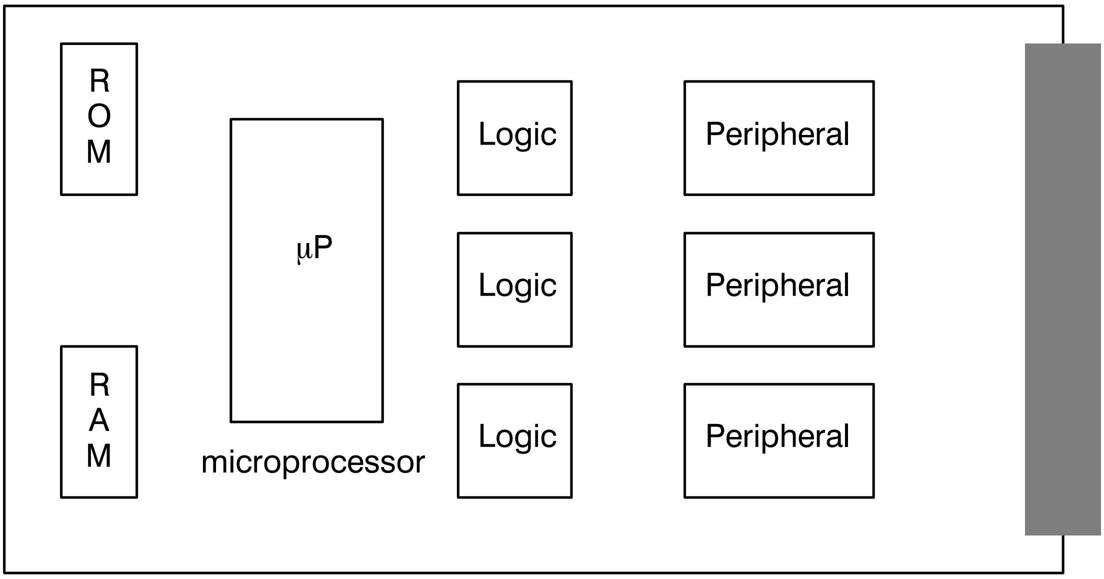

Figure 1.1: Single board computer with microprocessor

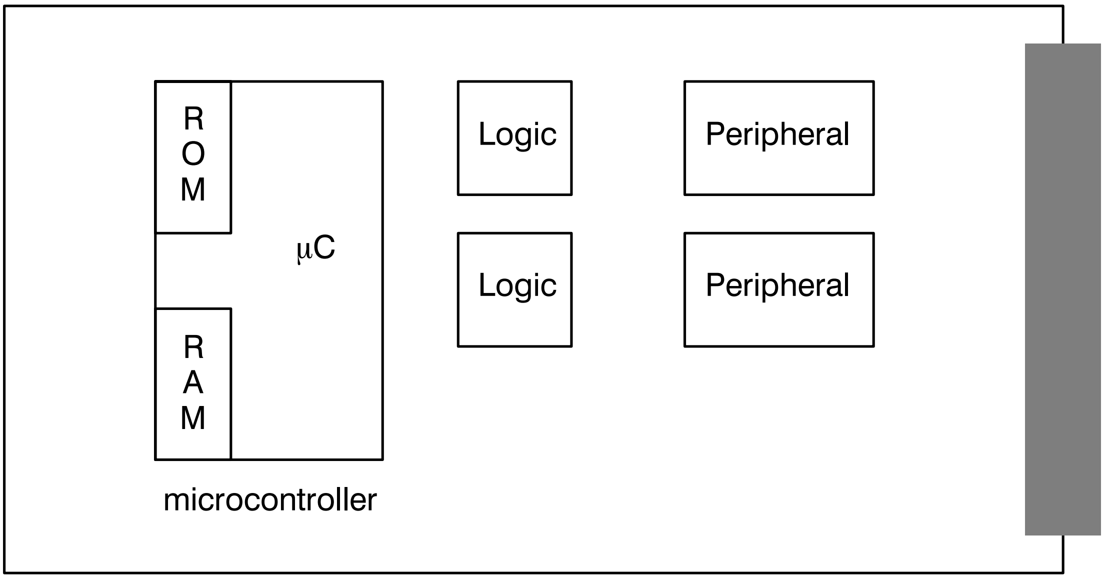

Figure 1.2: Single board computer with microcontroller

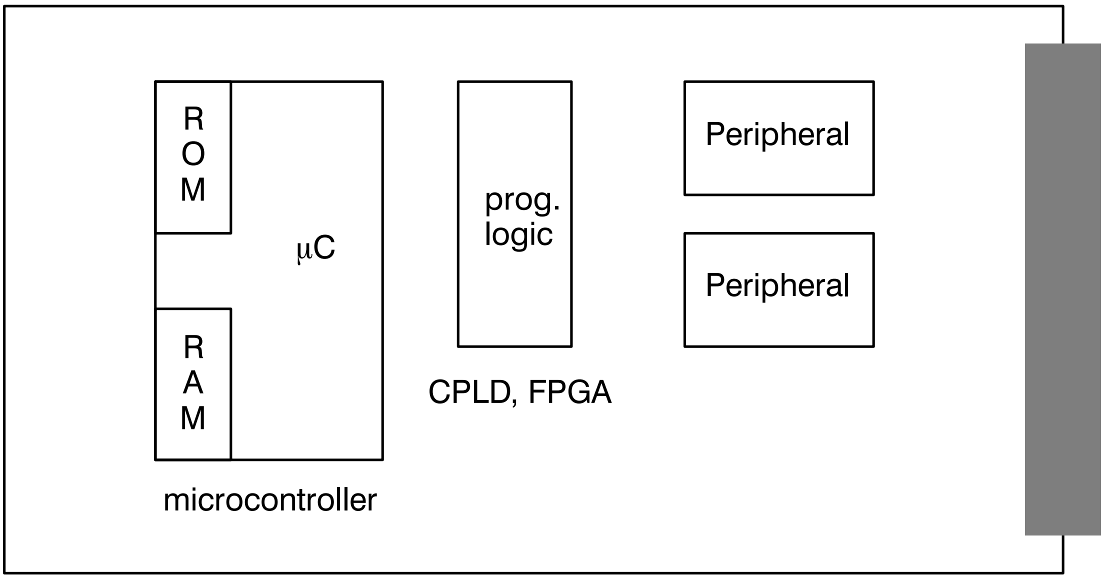

Figure 1.3: Single board computer with microcontroller and programmable logic

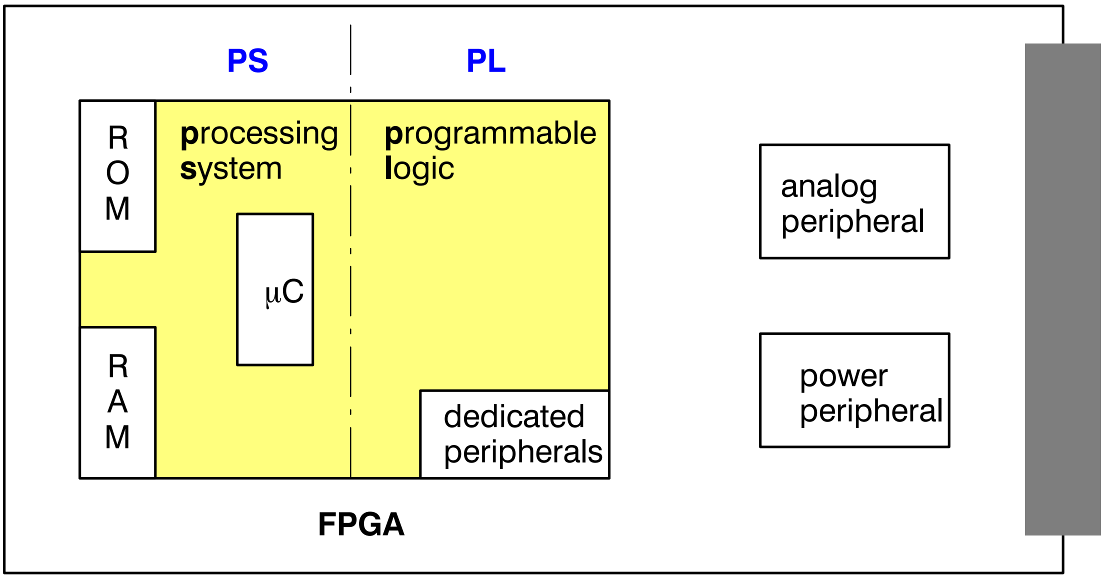

Figure 1.4: SoC structure (System-on-Chip)

2 Modeling an Electrical Drive

Whenever high torque and maintenance-free operation is required a stepper type motor is a possible choice. Fig. 1.5 shows the principle of operation.

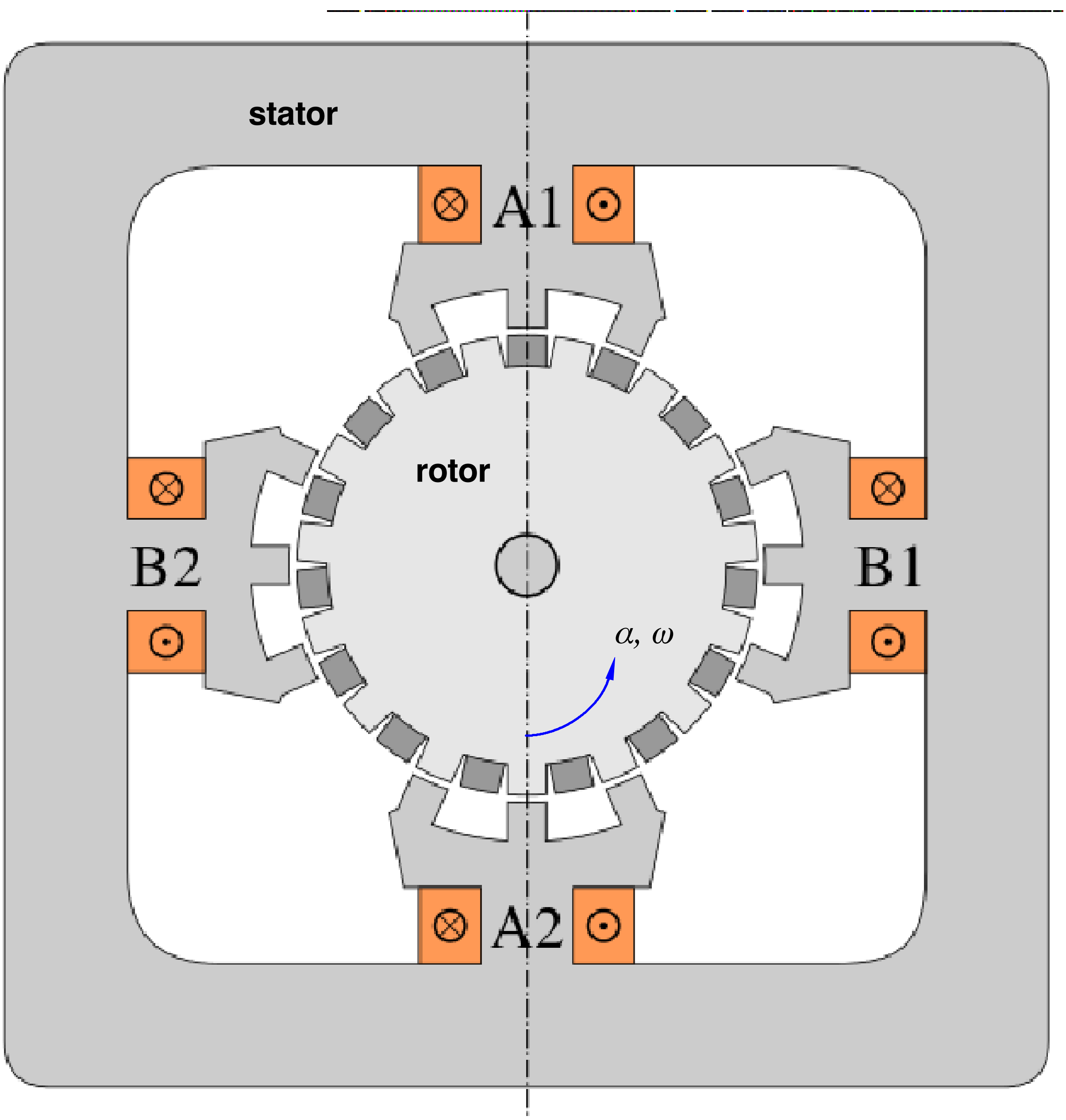

Figure 1.5: Stepper motor schematic (n = 15), 2-phase motor

Based on the physical principle that the integral over kinetic energy and the work of external forces always is an extremum we can derive the equations of motion.

An electrical drive converts magnetic energy to mechanical energy. This happens whenever the magnetic energy depends on the rotation angle α. The magnetic energy (more exact the so-called magnetic coenergy) is given by

$$W_m^i = \left(k_0 + k_1 \cos(n\alpha)\right) \frac{i^2}{2} \, . \tag{1.1}$$

2.1 Mechanical Subsystem

The kinetic energy is

$$V = \frac{1}{2}J\omega^2 \,. \tag{1.2}$$

The sum of these energies is the so-called *extended Lagrange energy* function

$$L^{ex} = W_m^i + V = \left[k_0 + k_1\cos(n\alpha)\right]\frac{i^2}{2} + \frac{1}{2}J\omega^2 \,. \tag{1.3}$$

The equation of motion follows from the well-known Euler-Lagrange differential equation

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L^{ex}}{\partial\omega}\right) - \frac{\partial L^{ex}}{\partial\alpha} = 0 \,. \tag{1.4}$$

This results in the *mechanical equation of motion*

$$J\frac{\mathrm{d}\omega}{\mathrm{d}t} = \underbrace{-\,k_1\,n\frac{i^2}{2}\,\sin(n\alpha)}_{\text{torque}} \,. \tag{1.5}$$

2.2 Electrical Subsystem

According to Telegen's theorem the sum of all powers in an electrical network is always zero. If a phase is powered ba y voltage and taking the electrical resistance into account the electrical power sum is

$$P^i = R\frac{i^2}{2} - ui \,. \tag{1.6}$$

Similar to the mechanical subsystem the *electrical system* is given by

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial W_m^i}{\partial i}\right) + \frac{\partial P_i}{\partial i} = 0 \,. \tag{1.7}$$

This results ion a somewhat complicated ODE for the current i

$$\underbrace{\left[k_0 + k_1\cos(n\alpha)\right]\frac{\mathrm{d}i}{\mathrm{d}t}}_{\substack{\text{induced voltage}\\ \text{by change of}\\ \text{current}}} - \underbrace{nk_1\,\omega\,i\,\sin(n\alpha)}_{\substack{\text{induced voltage}\\ \text{by rotational}\\ \text{speed (back-emf)}}} + Ri = u \,. \tag{1.8}$$

2.3 The KIS Principle (Keep It Simple)

If the maximum performance is required we need to take advantage of the model information given by (1.5) and (1.8). It is common practise for engineers to simplify the complete model if we can restrict the mode of operation. If the drive operates at low speed than ω is very small. At the same time the current will be constant for a period of time, so $\mathrm{d}i/\mathrm{d}t$ becomes zero. In this case (1.8) simplifies to

$$Ri = u \, , \tag{1.9}$$

ant therefore equation (1.5) results in

$$J\frac{\mathrm{d}\omega}{\mathrm{d}t} = -k_1 n \frac{u^2}{2R^2} \sin(n\alpha) \, . \tag{1.10}$$

The maximum torque is available for $n\alpha$ = -90°. The drive will perform a movement to $n\alpha$ = 0°, where the driving torque will be zero. This is called a *step* for the motor. If we switch to the second phase B, the same situation as for phase A occurs.

2.4 Stepper Motor Sequence

The following sequence of currents in the two phases result in a more or less discontinuous motion of the drive in one direction.

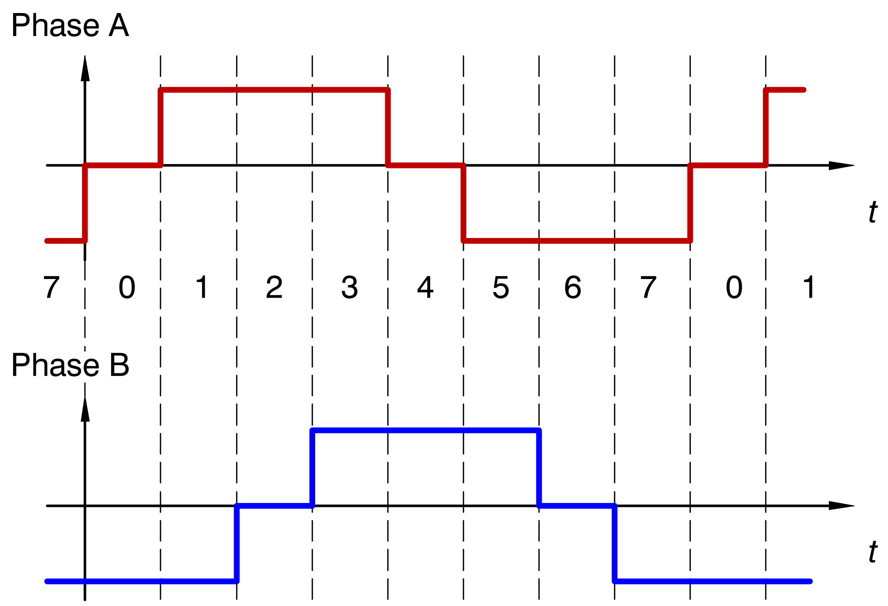

Figure 1.6: Sequence of motor currents for one direction

Please note, that other schemes to control the motor exist. The benefit of this scheme is that the current requirements are low (for a mobile device) and that the so-called full step is divided into 8 sub-steps (0...7) for better position resolution.

2.5 Power Section

The digital current information requires amplification to drive the motor. For each phase the MOSFET bridge in fig. 1.7 drives the necessary currents.

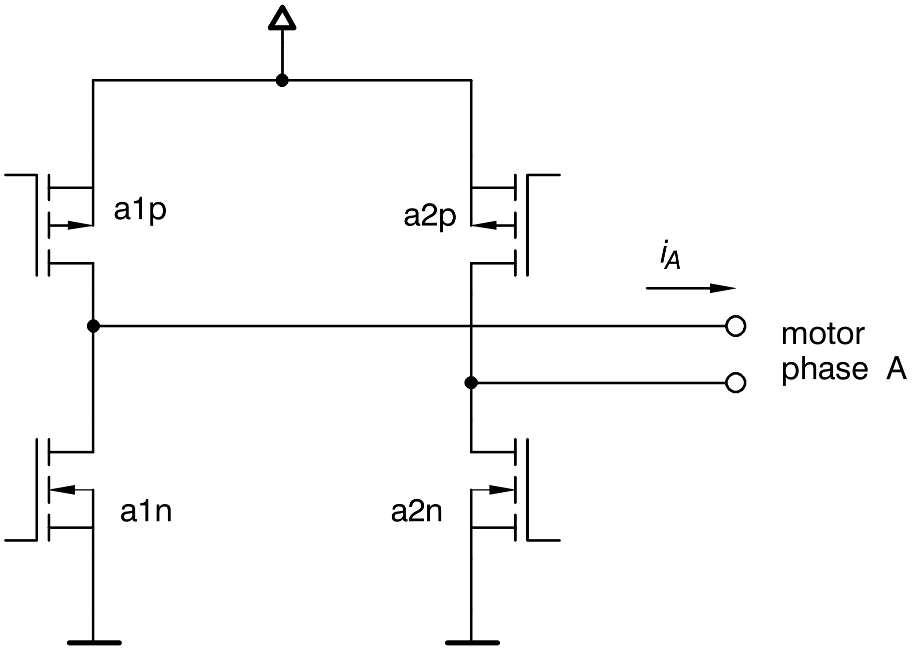

Figure 1.7: MOSFET power section

The same power section is used for phase B of the motor.

The gate signals of the power MOSFETs are the outputs of a digital system suitable for programmable logic. The benefits of a programmable logic solution are:

- fast and nanosecond precision
- reliable, safe with respect to software errors
- will result in much simpler software
- easy to synchronize many drives

For one phase the transistor gate signals need to be selected according to fig. 1.8.

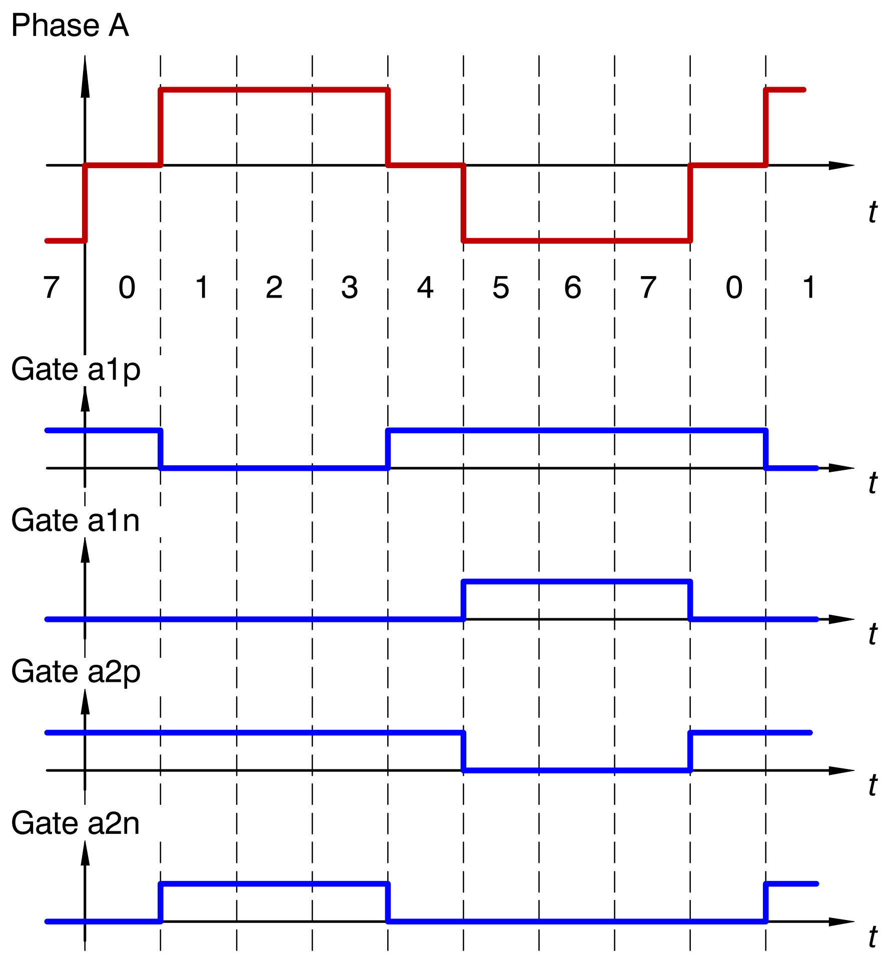

Figure 1.8: Gate signals for one phase

It is obvious that a1p = a2n* and a2p = a1n*, so only two signals need to be generated per phase.

3 Programmable Logic

The programmable logic consists of a (timed) state machine where the time between the steps determine the motor speed. The state diagram is as follows.

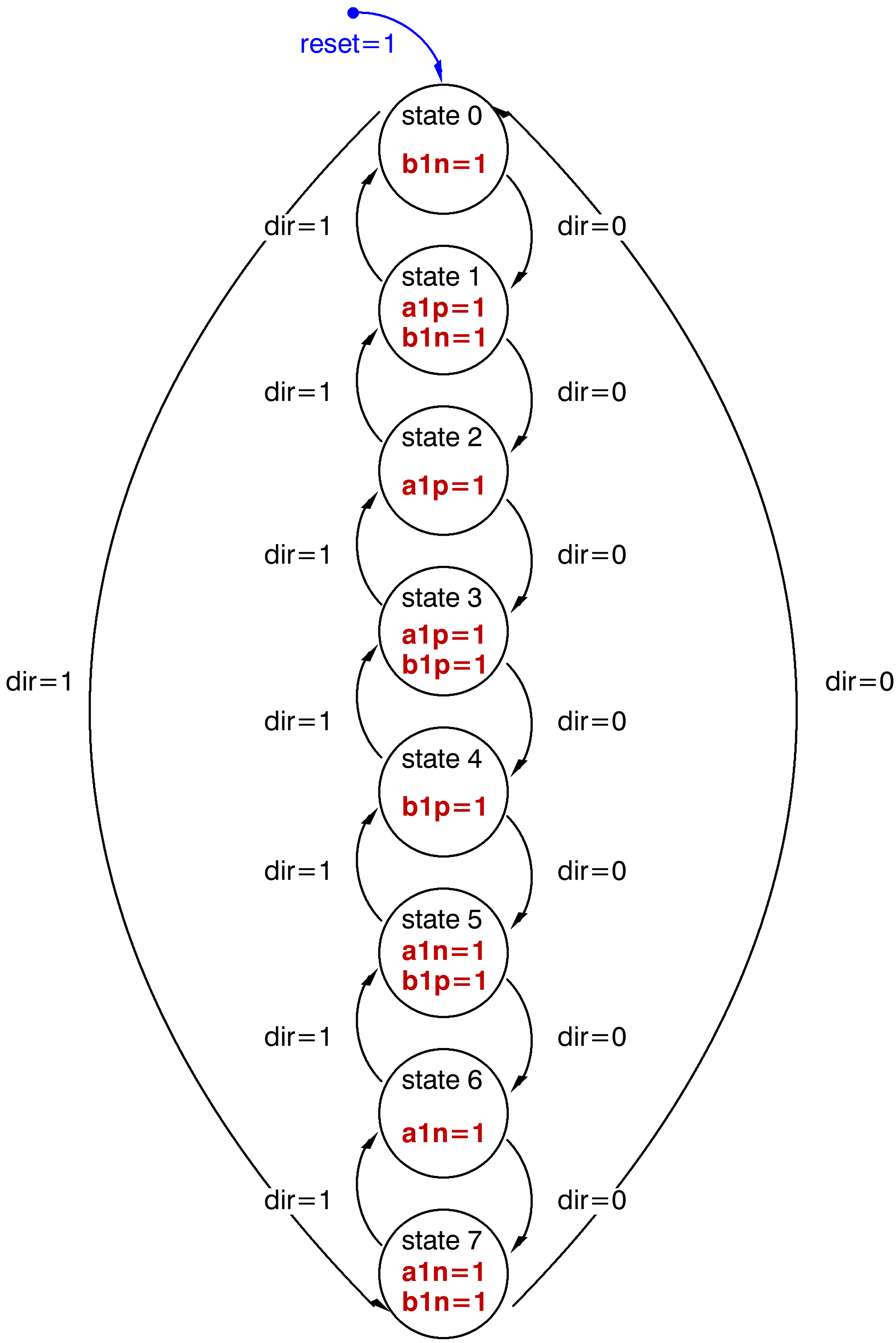

Figure 1.9: Finite state-machine (FSM) for MOSFET signals (two phases)

The short sequence of the state machine simulation is shown in fig. 1.10.

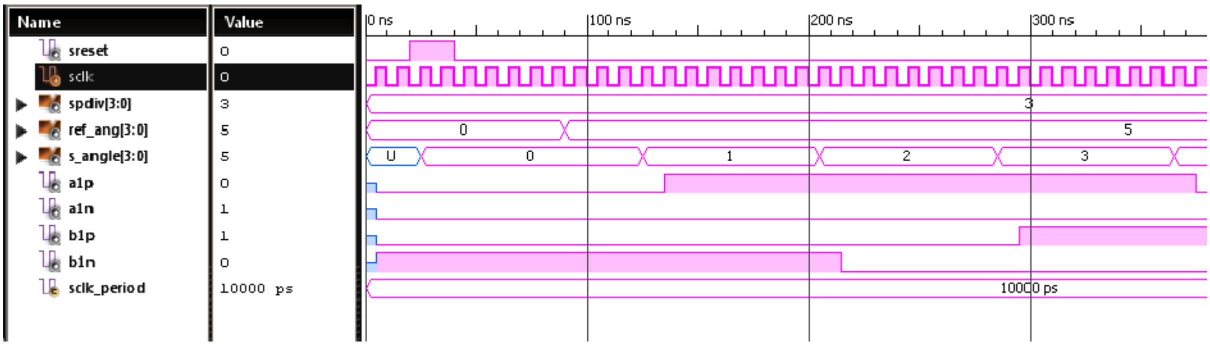

Figure 1.10: State machine (Moore machine) simulation

The VHDL (**V**ery **H**igh **S**peed **H**ardware **D**escription **L**anguage) realization of the state machine is shown below. For the reason of simplicity only the combinational block is shown.

```
comb_fsm: PROCESS(state, ref_ang, sangle_cnt, a_diff)
BEGIN

    next_state <= state;
    a_diff <= ref_ang - sangle_cnt;

    a1pi <= '0';
    a1ni <= '0';
    b1pi <= '0';
    b1ni <= '0';

    CASE state IS
        WHEN st0 =>
            b1ni <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st1;
            ELSE
                next_state <= st7;
            END IF;
        WHEN st1 =>
            a1pi <= '1';
            b1ni <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st2;
            ELSE
                next_state <= st0;
            END IF;
        WHEN st2 =>
            a1pi <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st3;
            ELSE
```

```
                next_state <= st1;
            END IF;
        WHEN st3 =>
            a1pi <= '1';
            b1pi <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st4;
            ELSE
                next_state <= st2;
            END IF;
        WHEN st4 =>
            b1pi <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st5;
            ELSE
                next_state <= st3;
            END IF;
        WHEN st5 =>
            a1ni <= '1';
            b1pi <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st6;
            ELSE
                next_state <= st4;
            END IF;
        WHEN st6 =>
            a1ni <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st7;
            ELSE
                next_state <= st5;
            END IF;
        WHEN st7 =>
            a1ni <= '1';
            b1ni <= '1';
            IF a_diff(STC_BITS-1)='0' THEN
                next_state <= st0;
            ELSE
                next_state <= st6;
            END IF;
    END CASE;

END PROCESS comb_fsm;
```

The hardware is attached to the microprocessor (PS) by a 32 bit AXI-lite bus component.

4 Processing System

The software executes on the 32 bit microcontroller MicroBlaze ™ which is part of the Xilinx EDK (Embedded Design Kit). Since the hardware does all logic the software becomes very simple. The AXI-bus component provides two registers (speed and position).Therefore, the software requires only to instructions to specify **speed**

```
CI_REG(SPDIV_REG_WR) = x_data;
```

and **position**

```
CI_REG(PREF_REG_WR) = x_data;
```

Here, CI_REG is the C macro

```
#define CI_REG(regc) (*(volatile u32 *)
                       (XPAR_SCIF_0_BASEADDR+4*regc))
```

to access memory mapped IO registers.

The rest of the software is required to perform user IO for monitoring the drive.

The communication is done by USB connection.

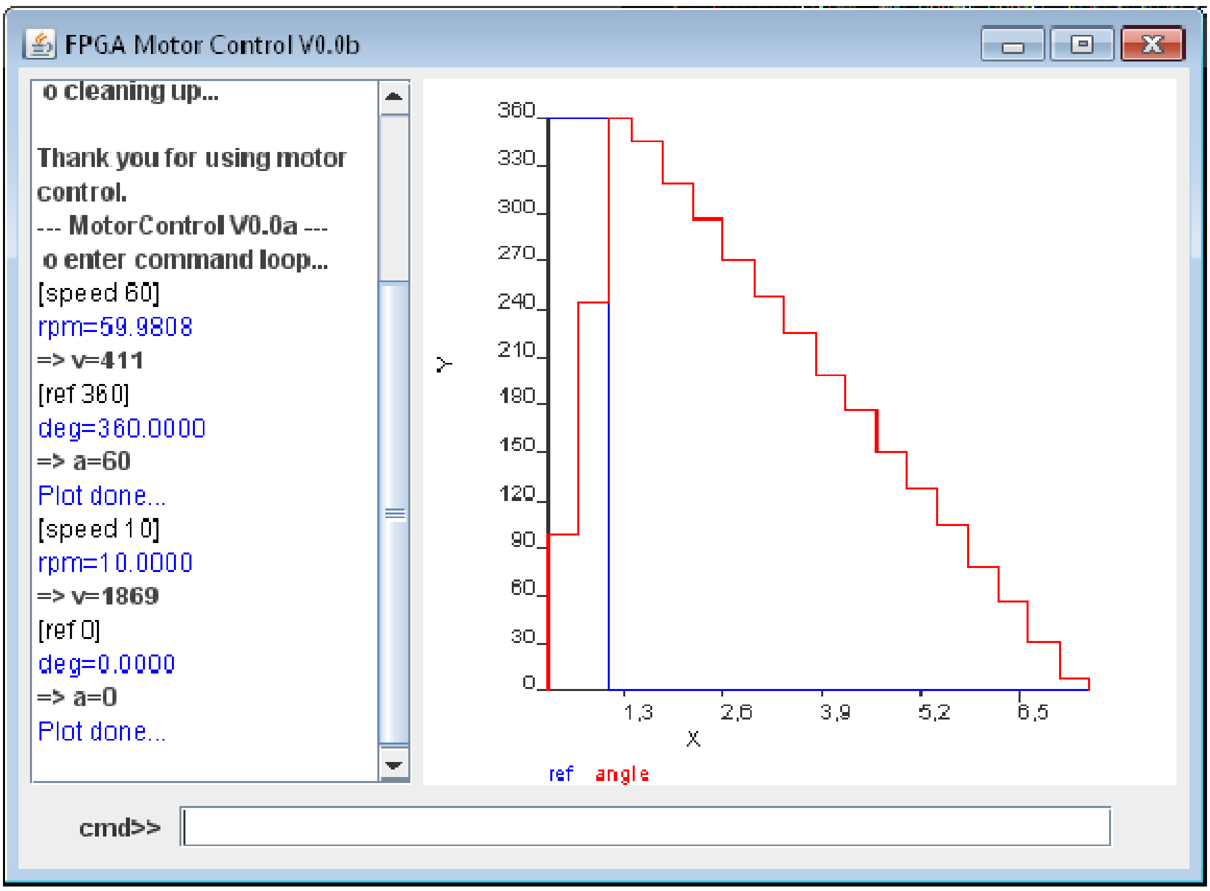

Figure 1.11: Motor control user interface

5 Programmable Logic for Continuous Motion

For some application the step type movement of the motor is not appropriate. Continuous motion requires sine/cosine currents in the two phases of the motor. This mode is closely related to synchronous servo drives – the highest performance drives in industry.

The power transistors are used in switch mode, i.e. the transistors are switched completely on and completely off. Only the mean value of the output voltage can take any value. This is accomplished by PWM (pulse width modulation).

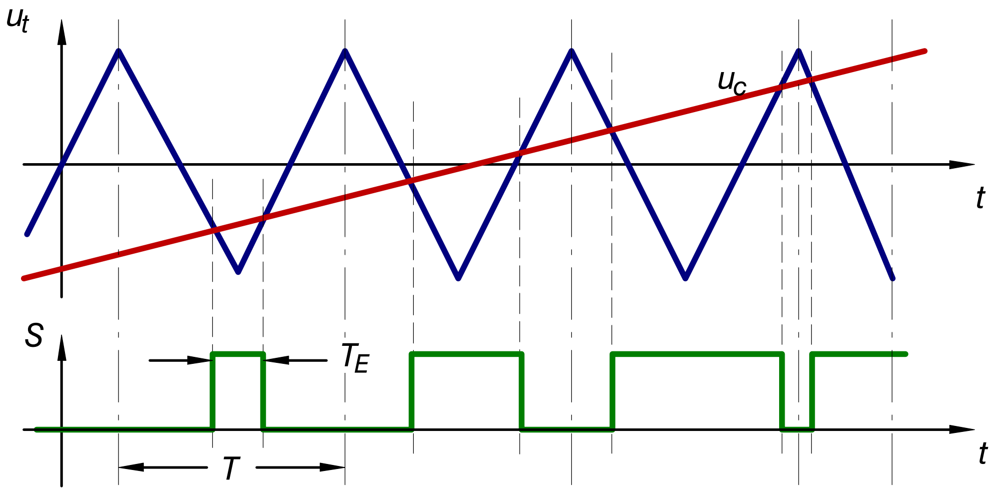

Figure 1.12: Pulse width modulation (PWM)

The PWM compares a continuous function u_c with a triangular signal u_t. Since ut is piecewise linear over time the output signal S (a binary signal!) has a mean value which corresponds to u_c by

$$\bar{S} = \frac{T_E}{T} = \frac{u_c + 1}{2} \, . \tag{1.11}$$

The input signal for phase A is cosine and for phase B the input signal is sine. Taking into account left and right half bridges we need four modulators. All modulator are synchronized to the same triangular signal. This signal is created by an up-down 12 bit counter. The hardware solution for the up-down counter is shown below.

```
udctr: PROCESS(sclk, clkdiv_p, reset, pwmcnt, din_0, din_1)
BEGIN
    if sclk'event AND sclk='1' THEN
        IF reset='1' THEN
            pwmcnt <= (OTHERS => '0');
            up_dwn <= '0';
            hithr_a <= (OTHERS => '0');
            lothr_a <= (OTHERS => '0');
            hithr_b <= (OTHERS => '0');
```

```
            lothr_b <= (OTHERS => '0');
        ELSIF clkdiv_p='1' THEN
            IF up_dwn='0' THEN
                pwmcnt <= pwmcnt + 1;
                IF pwmcnt=UD_COUNT_MAX THEN
                    up_dwn <= '1';
                END IF;
            ELSE
                pwmcnt <= pwmcnt - 1;
                IF pwmcnt=UD_COUNT_MIN THEN
                    up_dwn  <=  '0';
                    hithr_a <=  din_0;
                    lothr_a <= -din_0;
                    hithr_b <=  din_1;
                    lothr_b <= -din_1;
                END IF;
            END IF;
        END IF;
    END IF;
END PROCESS udctr;
```

5.1 Sine/Cosine Computation

If mathematical functions need to be computed at high speed (nanoseconds) CORDIC (**CO**rdinate **R**otation **DI**gital **C**omputer) provides a powerful algorithm for this purpose. Calculating sine and cosine is equivalent to the rotation of a unit vector in the cartesian plane.

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}. \tag{1.12}$$

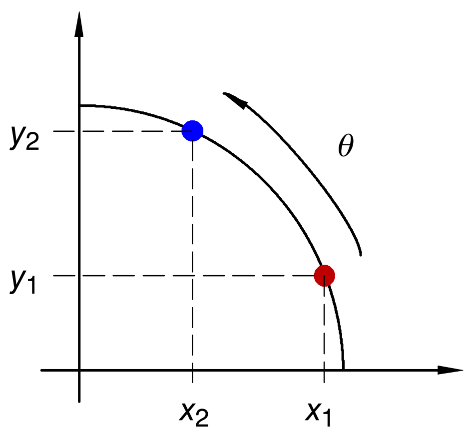

Figure 1.13: Rotation of a vector in the cartesian plane by an angle θ

If we rotate the vector $(x_1, y_1) = (1, 0)$ by θ then $x_2 = \cos(\theta)$ and $y_2 = \sin(\theta)$. Unfortunately this rotation is not suited for fast DSP. In 1959 Jack E. Volder discovered the CORDIC algorithm, the modified vector rotation

$$x_2 = x_1 \cos\theta - y_1 \sin\theta = \cos\theta \left(x_1 - y_1 \tan\theta\right), \quad (1.13)$$

$$y_2 = x_1 \sin\theta + y_1 \cos\theta = \cos\theta \left(y_1 + x_1 \tan\theta\right). \quad (1.14)$$

Dropping the $\cos\theta$ term lead to the *pseudo-rotation*

$$\hat{x}_2 = x_1 - y_1 \tan\theta, \quad (1.15)$$

$$\hat{y}_2 = y_1 + x_1 \tan\theta. \quad (1.16)$$

The error by $\cos\theta$ can be easily corrected later. If we restrict $\tan\theta$ to be powers of 2 than the algorithm requires only add, subtract and binary shift operations.

i	$\tan\theta^i$	θ^i (degrees)
0	1	45.0
1	0.5	26.565
2	0.25	14.036
3	0.125	7.125
4	0.0625	3.576
5	.	.
.	.	.

Any angle rotation between $\pm 90^\circ$ can be achieved by a sequence of so-called microrotations. The number of stages n $(0 \le i \le n\text{-}1)$ determines the precision. The angles θ^i are constants, it is not necessary to compute them on-line.

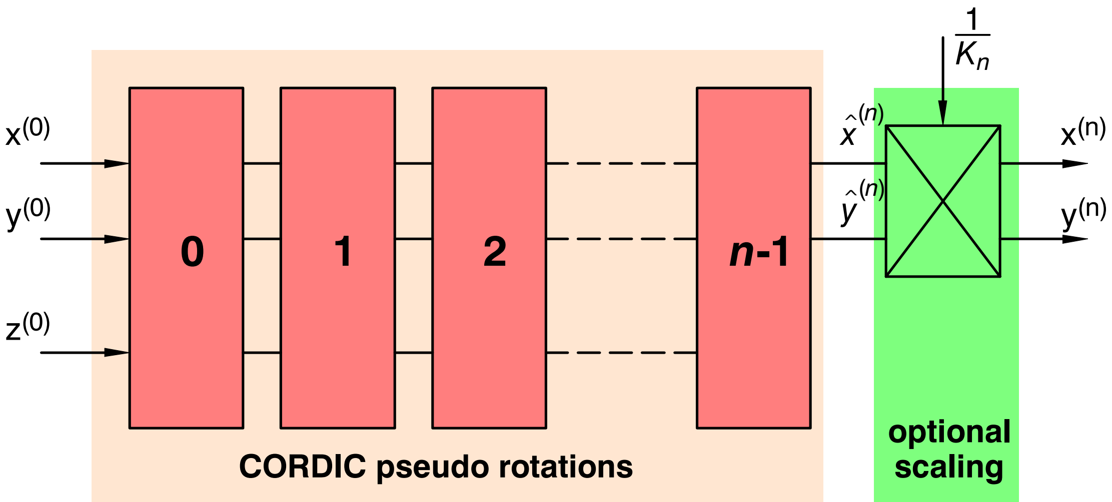

Figure 1.14: CORDIC hardware/software structure

The following function shows the CORDIC algorithm for a 16 bit sine/cosine computation. The function has been optimized for fixed point computation, an angle of 360° corresponds to $2^{15} = 32{,}768$.

```c
static void CordicSinCos(s16 z_ang, s16 *x_cos, s16 *y_sin)
{
    s16  x_in, y_in, z_in, x_out, y_out, z_out, reduct;
    int  k;

    x_in = 9900;
    y_in = 0;
    reduct = (z_ang >> 13) & 0x03;
    if (reduct == 1) {
        z_ang -= 16384;
    } else if (reduct == 2) {
        z_ang += 16384;
    }
    z_in = z_ang;
    for (k = 0; k < CORDIC_STAGES; k++) {
        if (z_in >= 0) {
            x_out = x_in - (y_in >> k);
            y_out = y_in + (x_in >> k);
            z_out = z_in - ZValues[k];
        } else {
            x_out = x_in + (y_in >> k);
            y_out = y_in - (x_in >> k);
            z_out = z_in + ZValues[k];
        }
        x_in = x_out;
        y_in = y_out;
        z_in = z_out;
    }
    if (reduct == 1) {
        *x_cos = -x_out;
        *y_sin = -y_out;
    } else if (reduct == 2) {
        *x_cos = -x_out;
        *y_sin = -y_out;
    } else {
        *x_cos = x_out;
        *y_sin = y_out;
    }
}
```

The algorithm provides 14 bit precision with only add, subtract, and shift operations.

6 Experimental System

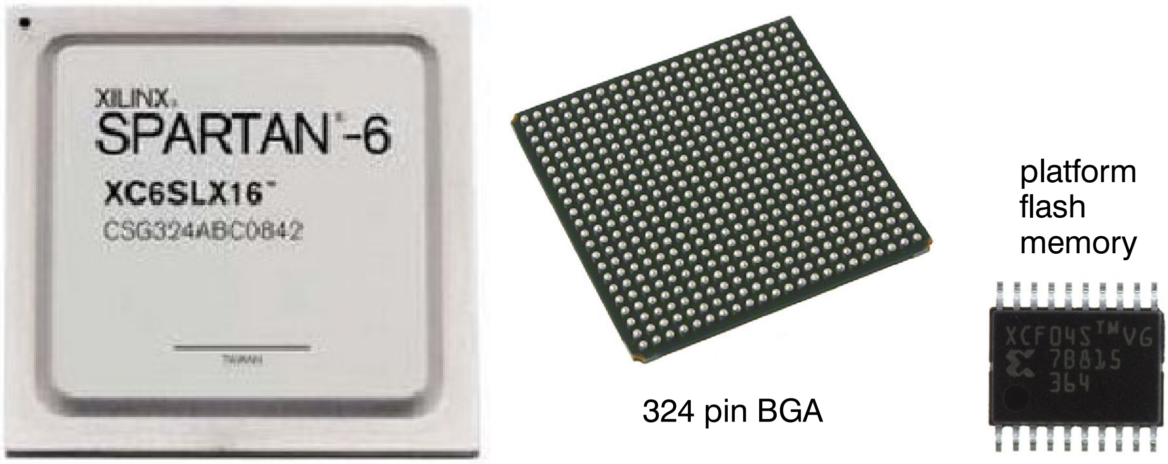

Figure 1.15: Required devices for DSP and control

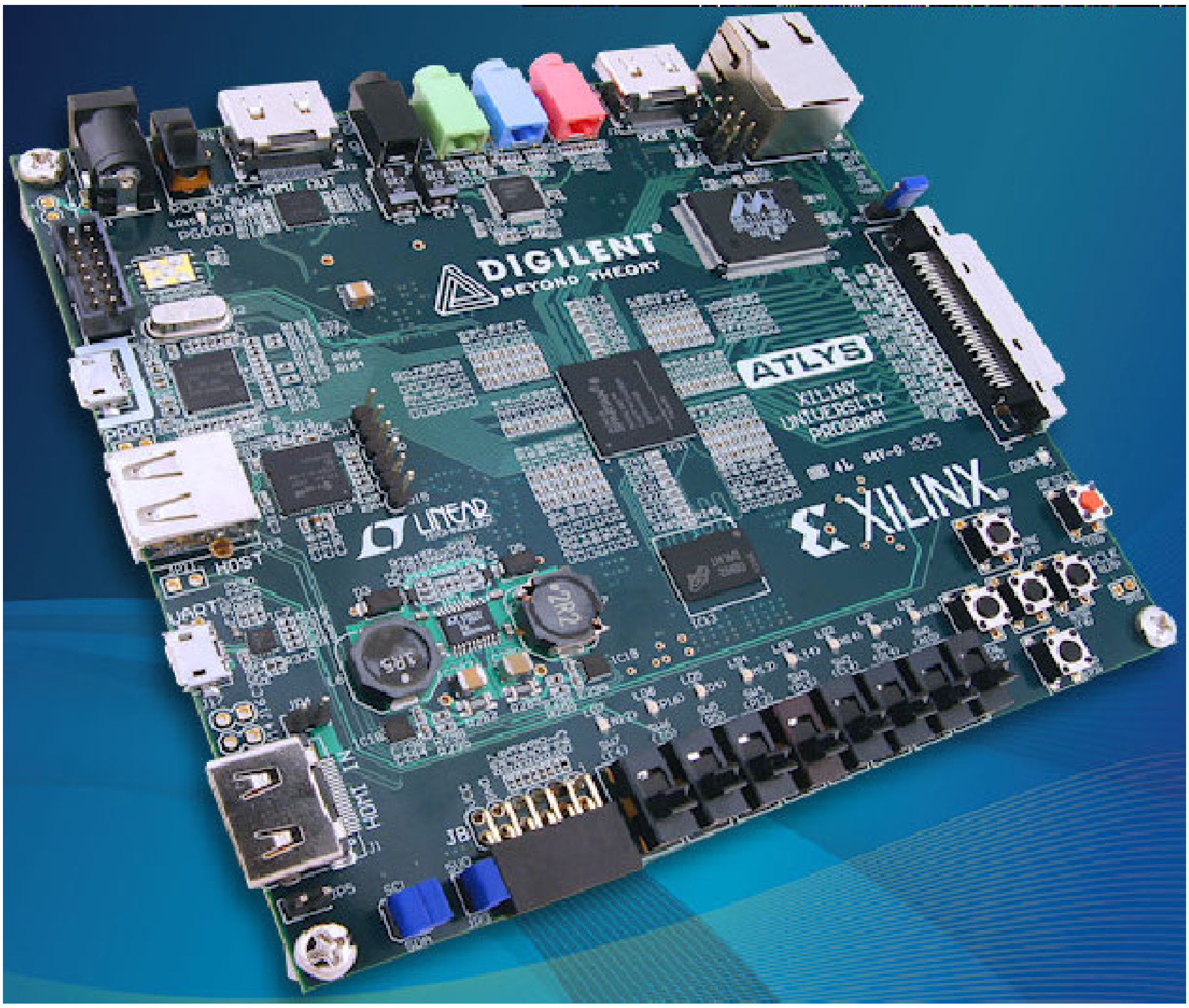

Figure 1.16: Spartan-6 experimental board (6,822 slices, 6-input LUTs, 58 DSP slices)

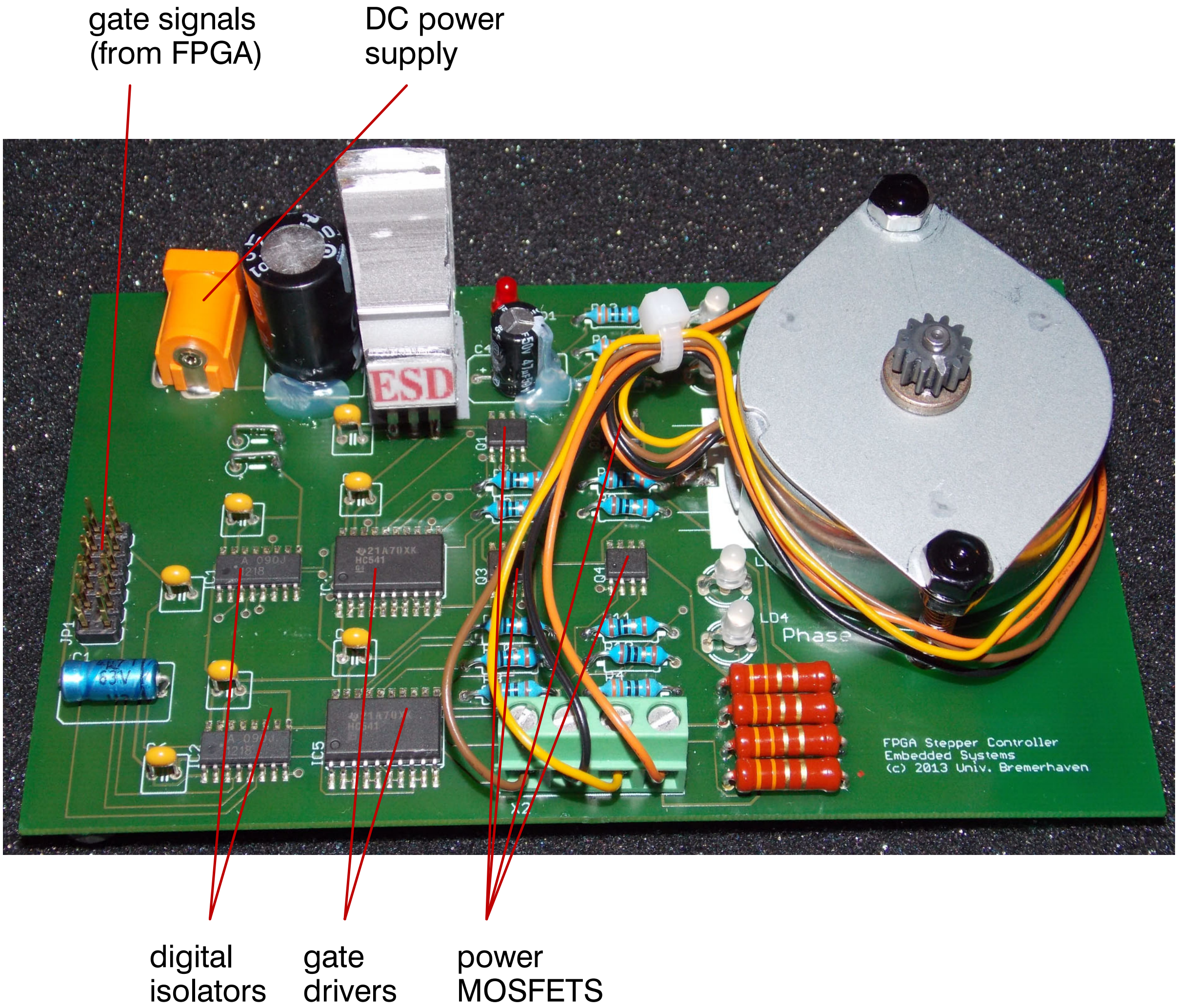

Figure 1.17: MOSFET power module and drive

***

7 Bibliography

[1] Ashenden, Peter J.: The Designer's Guide to VHDL, 3rd. Ed.
Morgan Kaufmann, 2008

[2] Al-Hashimi.Bashir M.: System-on-Chip: Next Generation Electronics.
Institution of Electrical Engineers, 2006

[3] Lipsett, Rger, Schaefer, Carls and Ussery Cary: VHDL Hardware Description and Design.
Kluwer Academic 1990

[4] Lyons, Richard G.: Understanding Digital Signal Processing.
Prentice Hall, 2011

[5] Predroni, Volnei A.: Circuit Design and Simulation with VHDL, 2nd. Ed.
MIT Press, 2010

[6] Prutchi, David and Norris, Michael: Design and Development of Medical Electronic Instrumentation.
John Wiley& Sons, 2005

[7] Reis, Ricardo, Lubaszewski, Marcelo and Jess, Jochen: Design of Systems on a Chip: Design and Test.
Springer, 2010

[8] Reichardt, J. und Schwarz, B.: VHDL-Synthese.
Oldenbourg, 2001

[9] Sass, Ron and Schmidt, Andrew G.: Embedded Systems Design with Platform FPGAs: Principles and Practices.
Elsevier Inc. 2010

[10] Wakerly, John F. : Digital Design, Principles & Practices.
Prentice Hall, 2001

***