

ECS Demo: Embedded System on Robot Control

April 18, 2011

1 Background

Embedded systems are information processing systems that are embedded into a larger product. This demo session gives a brief introduction to building an embedded robot control system with Xilinx Embedded Development Kit (EDK).

1.1 FPGA

FPGA: Field-Programmable Gate Array¹

Field-Programmable: designed to be configured after manufacturing
Gate: Building blocks in unit of million

Similar with LEGO: More building blocks = Bigger construction

Advantages:

Dedicated
Reprogrammable
Low-cost
Shorter time to market comparing with ASICs

FPGA-based embedded system is different from traditional ones on embedded hardware.

Allows users to make customized hardware design
Hardware are modularized in intellectual property (IP) cores

1.2 VHDL

VHDL: Very-high-speed-integrated-circuit Hardware Description Language².

VHDL is widely used:

Flexibility in hardware description
Hardware independent, description != realization
Thinking in HARDWARE because of parallel execution

¹<http://en.wikipedia.org/wiki/FPGA>

²<http://en.wikipedia.org/wiki/VHDL>

2 Embedded robot control system

2.1 Overview

The embedded robot control system in this demo is made up of five parts as shown in figure 1.

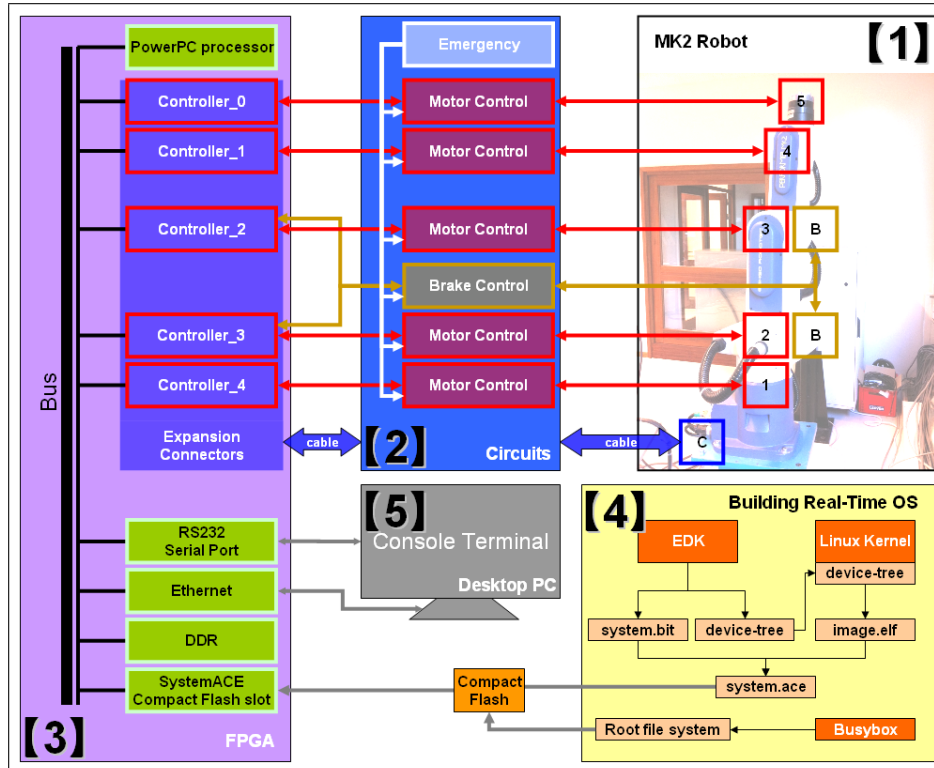


Figure 1: Overview of embedded robot control system

1. Robot

Eshed Performer MK2 robot arm

2. Circuits

Circuits for motor control, brakes control and emergency control

3. FPGA board

Controller hardware, programmed with Xilinx Embedded Design Kit (EDK) toolchain

4. Operating system and software

A CF card is used to store real-time embedded Linux operating system and software of the robot

5. Console Interface

The console interface is via RS-232 serial port

2.2 Hardware

The dedicated computing system is based on Xilinx virtex-II Pro XUP board, including a number of built-in hardware components. There are some well known I/O components on the board, e.g. Ethernet, USB, PS/2, RS-232 serial port, stereo audio, SATA, and there are also a few general purpose I/O connectors on the board, e.g. high-speed and low-speed expansion connectors, as shown in figure 2.

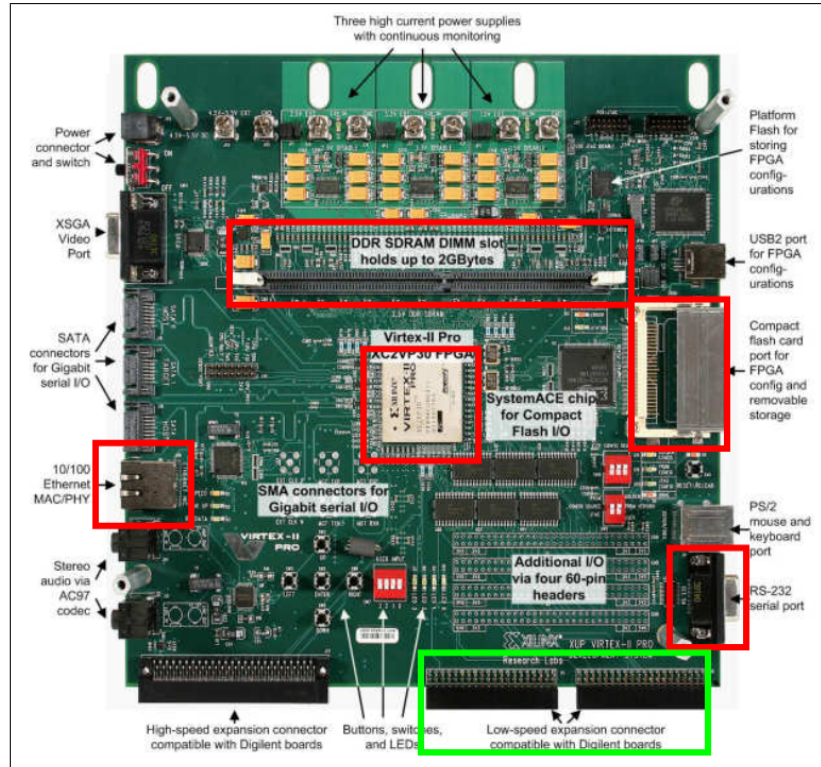


Figure 2: FPGA board (FPGA block in figure 1). Red/green: standard/customizable I/Os

The robot controlling embedded system consists of the following components:

- | | |
|------------------------|--|
| 1. CPU | the processing unit |
| 2. File system storage | for boot loader and file system |
| 3. Console interface | interface of the Linux operating system |
| 4. Memory | memory support of the Linux operating system |
| 5. Ethernet | communication in local network |
| 6. Controller logic | communication with motor control circuits |

Fortunately, components 1 to 5 above are standard components, they are available in EDK library. The PowerPC processor is the central processing unit, and the Compact flash card with SystemACE takes care of file system storage, while the RS-232 port interfaces the operating system console. These components comprise of knowledge from earlier design efforts and constitute intellectual property (IP), which were also known as IP cores. These physical hardware are highlighted in red boxes in figure 2.

The controller logic is a customized hardware component which is created in EDK with VHDL programming language. It uses the expansion connectors on the FPGA board as inputs and outputs. The physical hardware are highlighted in green box in figure 2.

Figure 3 gives a complete hardware component list, with red boxes indicate standard components, and green box indicates customized components.

Name	IP Type	IP Version
External Ports		
ppc405_0	ppc405	2.00.c
ppc405_1	ppc405	2.00.c
plb	plb_v34	1.02.a
opb	opb_v20	1.10.c
jtagppc_0	jtagppc_cntlr	2.00.a
reset_block	proc_sys_reset	1.00.a
plb2opb	plb2opb_bridge	1.01.a
RS232_Uart_1	opb_uartlite	1.00.b
Ethernet_MAC	opb_ethernetlite	1.01.b
SysACE_CompactFlash	opb_sysace	1.00.c
DDR_512MB_64Mx64_ran	plb_ddr	2.00.a
plb_bram_if_cntnr_1	plb_bram_if_cntnr	1.00.b
plb_bram_if_cntnr_1_bram	bram_block	1.00.a
opb_intc_0	opb_intc	1.00.c
sysclk_inv	util_vector_logic	1.00.a
clk90_inv	util_vector_logic	1.00.a
ddr_clk90_inv	util_vector_logic	1.00.a
dcm_0	dcm_module	1.00.c
dcm_1	dcm_module	1.00.c
controller_0	lincontrol	1.20.h
controller_1	lincontrol	1.20.h
controller_2	lincontrol	1.20.h
controller_3	lincontrol	1.20.h
controller_4	lincontrol	1.20.h

Figure 3: Hardware component list in EDK. Red/green: standard/customized components, with respect to FPGA block in figure 1

Each component has a number of ports as interface to communicate with other ports or directly with hardware on the board. For example, there are in total 12 ports in a controller component, as shown in figure 4.

Port Name	IP Type	IP Version	Connection
controller_0	lincontrol	1.20.h	
-CHA			controller_0_CHA I
-CHB			controller_0_CHB I
-CHI			controller_0_CHI I
-emergency			No Connection I
-brake_2			No Connection O
-brake_3			No Connection O
-PWM			controller_0_PWM O
-PWM_inv			controller_0_PWM_inv O
-CHA_test			No Connection O
-CHB_test			No Connection O
-CHI_test			No Connection O
-Home_signal			controller_0_Home_signal I

Figure 4: Port map of controller IP core for axis 5 in EDK (Controller_0 block in figure 1)

There are 5 axes on the Performer MK2 robot; axis 2 and axis 3 are attached with brakes. *brake_2* and *brake_3* ports are for the brake controls on these axes. They are not connected in axis 1, 4 and 5.

The ports are mapped with physical pins on the FPGA Board with respect to a user constraint file (UCF). The following is a part of the UCF file of the robot controller.

```
NET "controller_0_CHA" LOC = "M2";
NET "controller_0_CHB" LOC = "L5";
NET "controller_0_CHI" LOC = "N6";
NET "controller_0_PWM" LOC = "V4";
NET "controller_0_PWM_inv" LOC = "U7";
NET "controller_0_Home_signal" LOC = "T5";
```

For example, *M2* maps pin 8 on the left low speed expansion connector (green box in figure 2). Physically, it is connected to the robot to receive a position signal returned from axis 5. Therefore, *controller_0_CHA* receives the signal applied on pin 8 to the FPGA.

Each controller IP has a number of user accessible registers which allow the user to read from or write to the IP core. There are 13 user accessible registers in the controller IP. In figure 5, the base address of *controller_0* is 0x77200000; this is also the address for the first user accessible register. The last user accessible register address is 0x7720000C although the end address is 0x7720FFFF by default. Without these registers, the user will not be able to get the status of the robot, nor to send command to control it.

Name	Address	Base Address	High Address	Size
controller_0				
SOPB		0x77200000	0x7720FFFF	64K
controller_1				
SOPB		0x77220000	0x7722FFFF	64K
controller_2				
SOPB		0x77240000	0x7724FFFF	64K
controller_3				
SOPB		0x77260000	0x7726FFFF	64K
controller_4				
SOPB		0x77280000	0x7728FFFF	64K

Figure 5: Addresses of user accessible registers in EDK

When all necessary ports are mapped with physical pins, EDK will generate a system bitstream file, *system.bit*, with all the hardware information and configuration included; and a software library called *device-tree*, which provides embedded operating system level software configuration.

2.3 Software

2.3.1 Cross compiling Linux kernel

The operating system kernel running on the FPGA board is Linux 2.6.33. Since the PowerPC processor architecture is different from the one on a desktop or laptop PC, the kernel as well as the software programs must be compiled with PowerPC compiler (cross compiling). The compiler creates executable code for a platform (embedded Linux with PowerPC processor) other than the one on which the compiler is running (Desktop/Laptop PC Linux with Intel processor).

The Linux kernel is available for download at <http://www.kernel.org/>. Modify the Makefile for PowerPC architecture:

```
ARCH          := powerpc
CROSS_COMPILE := powerpc-405-linux-gnu-
```

then it is ready to be compiled for PowerPC architecture, using *powerpc-405-linux-gnu-* cross compiler.

The device-tree files are copied to Linux source folder to pass the software information to the kernel, e.g. addresses of the user accessible registers in figure 5.

2.3.2 Kernel boot-loader

Cross compiling the kernel will generate an executable and linkable format (ELF) file³. Xilinx EDK needs this ELF file and the bitstream file *system.bit* to generate ACE file *system.ace* as boot-loader. A CF card with two partitions is needed to boot the Linux kernel.

Remarks:

1. Format a CF card and create two partitions, FAT16 and EXT2
2. Copy the *system.ace* to the FAT16 partition

2.3.3 Root file system

The root file system is generated by Busybox⁴. Copying the entire generated directory to the EXT2 partition on the CF card and then boot the kernel, the user would be able to see the console via RS-232.

Building the embedded Linux operating system is finished.

2.4 Implementation

2.4.1 Robot control software

The robot is controlled by writing parameters to the user accessible registers. These registers can be accessed at the addresses defined in figure 5 using memory mapping function *mmap* in C/C++ code:

```
int *p=(int*)mmap(0,256,PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0x77200000);
```

```
    p      pointer pointing at address 0x77200000
    *p     value stored in address 0x77200000
```

The contents of the 13 user accessible registers can be accessed by: **p*, **(p+1)*, **(p+2)*, ..., **(p+12)*.

The c code is cross compiled on the desktop/laptop PC with cross compiler:

```
powerpc-405-linux-gnu-gcc robot.c -o robotControl
```

The output file *robotControl* is executable on embedded Linux system with PowerPC processor.

³http://en.wikipedia.org/wiki/Executable_and_Linkable_Format

⁴<http://en.wikipedia.org/wiki/Busybox>

2.4.2 Motor control circuits

The robot needs to be powered by a 24V power supply, and controlled by the PWM signals sent from the FPGA. The motor control circuits are designed to drive the H-bridges. (*motor control* blocks in figure 1)

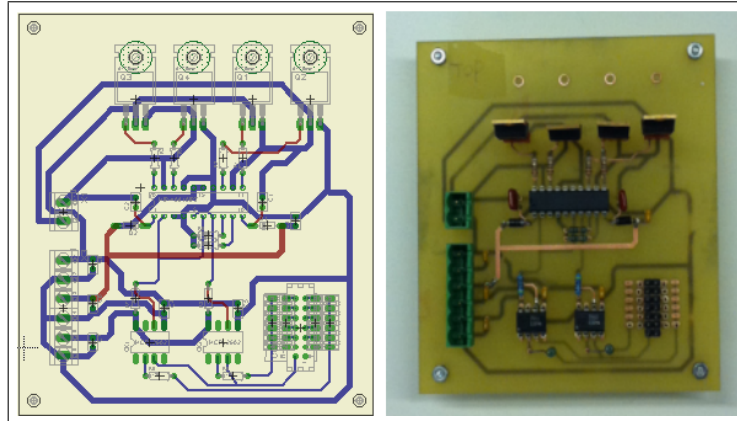


Figure 6: Motor control circuit (Motor Control block in figure 1)

2.4.3 Brake control circuits

The brakes on axis 2 and axis 3 need a 24V voltage to be released. If the power supply is shutdown, the brakes are ON. (*brake control* block in figure 1)

2.4.4 Emergency stop circuit

In emergency case, one may press the emergency button to cut off the power to all axes and brakes of the robot. The emergency signal is received by the FPGA via the expansion connector to stop hardware logic. (*emergency* block in figure 1)

3 Appendix

1. homePosition5_demo.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>

/*****DEFINE CONSTANTS*****/

/*AXIS 5*/ /*- controller_0 in EDK -*/
#define THETA_5 0.003141592654
#define TRANS_5 88
#define T_SPEED_CTE_5 3569.991652
#define P_SPEED_CTE_5 0.006809218696
#define ACCEL_CTE_5 1.2987554
#define KP_5 3000

/* memory operation related constant */
#define MAPPING_FAILED (int*)-1
#define OPENING_MEMORY_FAILED -1

/* number of bits in a register */
#define LENGTH 32
/*****

/* ----- */
/* --- FUNCTIONS --- */
/* ---DECLARATION--- */
/* ----- */

/* getBinaryBit: reads a single bit from a 32-bit register w.r.t. bit index*/
unsigned int getBinaryBit(unsigned int input, unsigned int bitIndex);

/* A finite state machine to position axis @ home */
void state_0_func(int *Axis);
void state_1_func(int *Axis);
void state_2_func(int *Axis);
void state_3_func(int *Axis);
void state_4_func(int *Axis);
void state_5_func(int *Axis);

/* two rotate modes */
/* rotate an axis w.r.t. specified direction slowly */
void rotateA5(int *Axis, int direction, float increment);

/* rotate to a destination (position) */
```

```

void rotateToDest(int *Axis, float increment, float destination);

/* homing Axis 5 function*/
int HomingAs5(int *Axis);

/* ----- */
/* -----MAIN----- */
/* ----- */

int main(void) {
    /*
    des_pos_int_5 = (int)(des_pos_5*65536);
    this integer represents the desired position
    of Axis 5 in integer, which is needed by the
    controller hardware component.
    */

    //desired position of Axis 5
    float des_pos_5 = 0;
    //desired position of Axis 5 in integer
    int des_pos_int_5 = 0;

    //user accessible register
    int *uar5;

    //initialize homePosition variable
    float homePosition=0.0;

    //opening memory for mapping
    int controllerMemory;
    controllerMemory = open("/dev/mem", O_RDWR);

    //if fails opening the momory, quit the program
    if(controllerMemory == OPENING_MEMORY_FAILED) {
        printf("Err: cannot open /dev/mem\n");
        return -1;
    }

    //memory mepping. If fails, quit the program
    uar5 = (int *)mmap(0, 256, PROT_READ|PROT_WRITE, MAP_SHARED, controllerMemory, 0x77200000);
    if (uar5 == MAPPING_FAILED) {
        printf("Err: cannot access controller of axis 5!\n");
        return -1;
    }

    /*
    writing parameters to the user accessible
    registers to configure the motor
    */

```

```

*(uar5+1) = KP_5;
*(uar5+2) = (int)(THETA_5 * 262144);
*(uar5+3) = TRANS_5;
*(uar5+4) = (int)(T_SPEED_CTE_5 * 8192);
*(uar5+5) = (int)(P_SPEED_CTE_5 * 33554432);
*(uar5+6) = (int)(ACCEL_CTE_5 * 33554432);
printf("Done. \n");

/*
In reality, the motor may start at any position,
but the system will not be aware of this. The robot
should be reset before using.

The following code does:
1. moves the axis to a certain position
2. reset it to it's home position.
*/

//1. moves the axis to a certain position
//report the current position
//set desired destination position of axis 5

printf("Input desired position of axis 5: (current = %f)",(float)*(uar5+11)/32768.0);
scanf("%f", &des_pos_5);

//move
rotateToDest(uar5,0.3,des_pos_5);

//2. reset it to it's home position.
//goingHome = 1 : home position is not reached
//goingHome = 0 : home position is reached
int goingHome=1;

while(goingHome){
//goingHome=0 if home position is found
goingHome=HomingAs5(uar5);
}

homePosition=*(uar5+11)/32768.0;
printf("homePosition is: %f\n", homePosition);

//unmap the memory
munmap(uar5, 256);
//close the memory
close(controllerMemory);
return 0;
}

/* ----- */

```

```

/* --- FUNCTIONS --- */
/* ----- */
unsigned int getBinaryBit(unsigned int input, unsigned int bitIndex){
char h[LENGTH];
int i;

for (i=0;i<LENGTH;i++){
h[i]=0;
}

for (i=0;i<LENGTH;i++){
if(input%2==0)
h[i]=0;
else
h[i]=1;
input=input>>1;
}
return h[bitIndex];
}

int HomingAs5(int *Axis){
int current_state=0;
int next_state=0;
int end_state_machine=6;
int time=50;
while(current_state<=5){
current_state=next_state;
switch(current_state){
case 0:
if(getBinaryBit((unsigned int)*(Axis+12),28)==1)
next_state=1;
else
next_state=2;
state_0_func(Axis);
usleep(time);
break;

case 1:
if(getBinaryBit((unsigned int)*(Axis+12),28)==1)
next_state=1;
else
next_state=2;
state_1_func(Axis);
usleep(time);
break;
case 2:
if(getBinaryBit((unsigned int)*(Axis+12),28)==1)
next_state=3;
else
next_state=2;

```

```

state_2_func(Axis);
usleep(time);
break;
case 3:
if(getBinaryBit((unsigned int)*(Axis+12),28)==1)
next_state=3;
else
next_state=4;
state_3_func(Axis);
usleep(time);
break;
case 4:
if(getBinaryBit((unsigned int)*(Axis+12),16)==1){
next_state=5;
}
else{
next_state=4;
}
state_4_func(Axis);
break;
case 5:
state_5_func(Axis);
next_state=end_state_machine;
usleep(time);
break;
}
}
printf("function end ..\n");
return 0;
}

void state_0_func(int *Axis){
printf("starting homing Axis 5 ..\n");
}

void state_1_func(int *Axis){
rotateA5(Axis, 1, 0.3);
}

void state_2_func(int *Axis){
rotateA5(Axis, 1, 0.1);
}

void state_3_func(int *Axis){
rotateA5(Axis, 0, 0.1);
}

void state_4_func(int *Axis){
rotateA5(Axis, 0, 0.1);
}

```

```

void state_5_func(int *Axis){
printf("homing As5 is done.\n");
}

void rotateA5(int *Axis, int direction, float increment){
float des_pos=0.0;
int des_pos_int=0;
des_pos = (float)*(Axis+11)/32768;

if(increment>0.5)
increment=0.4;

if(direction==1)
des_pos += increment;
else
des_pos -= increment;

des_pos_int = (int)(des_pos*65536);
*Axis = des_pos_int;
}

void rotateToDest(int *Axis, float increment, float destination){

printf("in rotateToDest function .. \n");
float current_position, des_pos;
int des_pos_int = 0;
int direction=0;

current_position = (float)*(Axis+11)/32768;
if (current_position-destination>0)
direction = 1;
printf("dir : %d\n",direction);
if(increment>0.5)
increment=0.5;

while(abs(current_position-destination)>0.05){
current_position = (float)*(Axis+11)/32768;
if(direction==1)
des_pos=current_position-increment;
else
des_pos=current_position+increment;
des_pos_int = (int)(des_pos*65536);
*Axis = des_pos_int;
}
printf("rotateToDest function is done.. \n");
}

```

2. user_logic.vhd

```

-----
-- user_logic.vhd - entity/architecture pair
-----

```

```

--
-- *****
-- ** Copyright (c) 1995-2007 Xilinx, Inc. All rights reserved.      **
-- **                                                                 **
-- ** Xilinx, Inc.                                                  **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"  **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND **
-- ** SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE,  **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION   **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT, **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE **
-- ** FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY      **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE       **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS **
-- ** FOR A PARTICULAR PURPOSE.                                     **
-- **                                                                 **
-- *****
--
-----
-- Filename:          user_logic.vhd
-- Version:           1.10.a
-- Description:       User logic.
-- Date:             Fri Apr 1 14:59:54 2011 (by Create and Import Peripheral Wizard)
-- VHDL Standard:    VHDL'93
-----

-- necessary libraries
-- DO NOT EDIT BELOW THIS LINE -----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
-- DO NOT EDIT ABOVE THIS LINE -----

entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE -----
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE -----

    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol parameters, do not add to or delete
    C_DWIDTH          : integer      := 32;
    C_NUM_CE          : integer      := 13
  )

```

```

-- DO NOT EDIT ABOVE THIS LINE -----
);
port
(
-- ADD USER PORTS BELOW THIS LINE -----
CHA      : in std_logic;
CHB      : in std_logic;
CHI      : in std_logic;
emergency : in std_logic;
brake_2  : out std_logic;
brake_3  : out std_logic;
PWM      : out std_logic;
PWM_inv  : out std_logic;
CHA_test : out std_logic;
CHB_test : out std_logic;
CHI_test : out std_logic;
Home_signal : in std_logic;
-- ADD USER PORTS ABOVE THIS LINE -----

-----
-- some template codes are skipped here
--...
--...
-----

);
end entity user_logic;

-----
-- Architecture section
-----

architecture IMP of user_logic is

--USER signal declarations added here, as needed for user logic
-----
-- signals are a kind of intermediate variables
-----

-- current position
signal pos : std_logic_vector(0 to C_DWIDTH-1);
-- current pulse speed
signal sig_p_speed : std_logic_vector(0 to C_DWIDTH-1);
-- current time speed
signal sig_t_speed : std_logic_vector(0 to C_DWIDTH-1);
-- current acceleration
signal sig_accel : std_logic_vector(0 to C_DWIDTH-1);
-- brakes signals
signal brakes_reg, brakes_reg_tmp : std_logic_vector(31 downto 0);
-- desired_position, w.r.t. des_pos_int in c code.
signal des_position_sig : std_logic_vector(0 to C_DWIDTH-1);
-- K value for P-controller
signal K_p_sig : std_logic_vector(0 to C_DWIDTH-1);

```

```

-- radians
signal rad_puls_sig          : std_logic_vector(0 to C_DWIDTH-1);
-- divisor when calculate position
signal trans_sig            : std_logic_vector(0 to C_DWIDTH-1);
-- time speed constant
signal T_speed_cte_sig      : std_logic_vector(0 to C_DWIDTH-1);
-- pulse speed constant
signal P_speed_cte_sig      : std_logic_vector(0 to C_DWIDTH-1);
-- acceleration constant
signal Accel_cte_sig        : std_logic_vector(0 to C_DWIDTH-1);
-- A, B, I signals (to calculate position and rotating direction)
signal CHA_in               : std_logic;
signal CHB_in               : std_logic;
signal CHI_in               : std_logic;
-- PWM control signals.
signal sig_PWM              : std_logic;
signal sig_PWM_inv          : std_logic;

```

```

-----
-- Signals for user logic slave model s/w accessible registers
-- each signal takes care one user accessible register.
-- each signal is 32 bit long as the registers.
-----

```

```

signal slv_reg0              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg1              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg2              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg3              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg4              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg5              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg6              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg7              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg8              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg9              : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg10             : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg11             : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg12             : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg_write_select  : std_logic_vector(0 to 12);
signal slv_reg_read_select   : std_logic_vector(0 to 12);
signal slv_ip2bus_data       : std_logic_vector(0 to C_DWIDTH-1);
signal slv_read_ack          : std_logic;
signal slv_write_ack         : std_logic;

```

```

-----
-- Components
-----

```

```

-- !! Remark:
-- notice that here the component means VHDL component
-- to build a controller, which is different from the
-- component introduced in EDK level

```

```

-- mapping next hierachy level component (system) ports with signals
--or ports in current level (user_logic).
--
--This is similar to calling a function is C/C++.
--
--The function of system is encoding and decoding the signals between
--the FPGA and the robot.

component system is
  Port (
-- system clock and system reset
    clk :      in  STD_LOGIC;
    areset :   in  STD_LOGIC;
-- A, B, I signals (to calculate position and rotating driection)
    CHA_1 :    in  STD_LOGIC;
    CHB_1 :    in  STD_LOGIC;
    CHI_1 :    in  STD_LOGIC;
-- desired_position, w.r.t. des_pos_int in c code.
    des_position : in STD_LOGIC_VECTOR (31 downto 0);
-- K value for P-controller
    Kp :       in  STD_LOGIC_VECTOR (31 downto 0);
-- radians
    rad_puls : in  STD_LOGIC_VECTOR (31 downto 0);
-- divisor when calculate position
    trans :    in  STD_LOGIC_VECTOR (31 downto 0);
-- time speed constant
    T_speed_cte : in STD_LOGIC_VECTOR (31 downto 0);
-- pulse speed constant
    P_speed_cte : in STD_LOGIC_VECTOR (31 downto 0);
-- acceleration constant
    Accel_cte : in  STD_LOGIC_VECTOR (31 downto 0);
-- current time speed
    t_speed :  out STD_LOGIC_VECTOR (31 downto 0);
-- current pulse speed
    p_speed :  out STD_LOGIC_VECTOR (31 downto 0);
-- current acceleration
    accel :    out STD_LOGIC_VECTOR (31 downto 0);
-- current position
    position : out STD_LOGIC_VECTOR (31 downto 0);
-- PWM control signals.
    PWM :      out STD_LOGIC;
    PWM_inv :  out STD_LOGIC);
  end component;

begin
u0: system port map
  (
    clk      => Bus2IP_Clk,
    areset   => reset_tmp,
    CHA_1    => CHA_in,

```

```

    CHB_1      => CHB_in,
    CHI_1      => CHI_in,
    des_position => des_position_sig,
    Kp         => K_p_sig,
    rad_puls   => rad_puls_sig,
    trans      => trans_sig,
    T_speed_cte => T_speed_cte_sig,
    P_speed_cte => P_speed_cte_sig,
    Accel_cte  => Accel_cte_sig,
    t_speed    => sig_t_speed,
    p_speed    => sig_p_speed,
    accel      => sig_accel,
    position   => pos,
    PWM        => sig_PWM,
    PWM_inv    => sig_PWM_inv);

-----
-- some template codes are skipped here
--...
--...
-----

-- implement slave model register read mux
SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0, slv_reg1, slv_reg2, slv_reg3,
slv_reg4, slv_reg5, slv_reg6, slv_reg7, slv_reg8, slv_reg9, slv_reg10, slv_reg11, slv_reg12
) is
begin

    case slv_reg_read_select is
        when "1000000000000" => slv_ip2bus_data <= slv_reg0;
        when "0100000000000" => slv_ip2bus_data <= slv_reg1;
        when "0010000000000" => slv_ip2bus_data <= slv_reg2;
        when "0001000000000" => slv_ip2bus_data <= slv_reg3;
        when "0000100000000" => slv_ip2bus_data <= slv_reg4;
        when "0000010000000" => slv_ip2bus_data <= slv_reg5;
        when "0000001000000" => slv_ip2bus_data <= slv_reg6;
        when "0000000100000" => slv_ip2bus_data <= slv_reg7;
        when "0000000010000" => slv_ip2bus_data <= slv_reg8;
        when "0000000001000" => slv_ip2bus_data <= slv_reg9;
        when "0000000000100" => slv_ip2bus_data <= slv_reg10;
        when "0000000000010" => slv_ip2bus_data <= slv_reg11;
        when "0000000000001" => slv_ip2bus_data <= slv_reg12;
        when others => slv_ip2bus_data <= (others => '0');
    end case;
end process SLAVE_REG_READ_PROC;

-- Data going to the FPGA. (User accessible registers 8 - 12)
To_SW: process ( Bus2IP_Clk ) is begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        -- slv_reg12 acts as "spy" register to for some signals
        -- this is necessary for homing.

```

```

        slv_reg12(0 to 3)          <= CONV_STD_LOGIC_VECTOR(Home_signal, 4);
        slv_reg12(4 to 7)        <= CONV_STD_LOGIC_VECTOR(CHA_in, 4);
        slv_reg12(8 to 11)       <= CONV_STD_LOGIC_VECTOR(CHB_in, 4);
        slv_reg12(12 to 15)      <= CONV_STD_LOGIC_VECTOR(CHI_in, 4);
        slv_reg12(16 to C_DWIDTH-1) <= "0000000000000000";
        slv_reg11                 <= pos;
        slv_reg10                 <= sig_p_speed;
        slv_reg9                  <= sig_t_speed;
        slv_reg8                  <= sig_accel;
    end if;
end process To_SW;

-- Data coming from the FPGA. (User accessible registers 0 - 7)
SW_VALUES: process ( Bus2IP_Clk, emergency ) is begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
if (emergency /= '0') then
        brakes_reg <= (others => '0'); -- reset brake register
else
        des_position_sig    <= slv_reg0;
        K_p_sig             <= slv_reg1;
        rad_puls_sig        <= slv_reg2;
        trans_sig           <= slv_reg3;
        T_speed_cte_sig     <= slv_reg4;
        P_speed_cte_sig     <= slv_reg5;
        Accel_cte_sig       <= slv_reg6;
        brakes_reg          <= slv_reg7;
    end if;
    end if;
end process SW_VALUES;

--process for CHA signal
    CHA_proc: process ( Bus2IP_Clk ) is begin
        if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
            if (CHA = '0') then
CHA_in <= '0';
            else
CHA_in <= '1';
            end if;
        end if;
    end process CHA_proc;
--process for CHB signal
    CHB_proc: process ( Bus2IP_Clk ) is begin
        if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
            if (CHB = '0') then
CHB_in <= '0';
            else
CHB_in <= '1';
            end if;
        end if;
    end process CHB_proc;
--process for CHI signal

```

```

    CHI_proc: process ( Bus2IP_Clk ) is begin
        if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
            if (CHI = '0') then
CHI_in <= '0';
            else
CHI_in <= '1';
            end if;
        end if;
    end process CHI_proc;
--process for Emergency signal
    Emergency_proc: process ( Bus2IP_Clk, emergency ) is begin
        if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
            if (emergency = '0') then
reset_tmp <= '0';
            else
reset_tmp <= '1';
            end if;
        end if;
    end process Emergency_proc;

-- Process to release the brakes.
    Brake_proc: process ( Bus2IP_Clk, brakes_reg ) is
begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        if emergency /= '0' then
brake_2 <= '0';
brake_3 <= '0';
        elsif brakes_reg(3 downto 0) = "1101" then -- =13
-- release brakes of axis 2 and 3
brake_2 <= '1';
brake_3 <= '1';
        elsif brakes_reg(3 downto 0) = "0101" then -- =5
-- release brake of axis 2
brake_2 <= '1';
brake_3 <= '0';
        elsif brakes_reg(3 downto 0) = "1001" then -- =9
-- release brake of axis 3
brake_2 <= '0';
brake_3 <= '1';
        else
            -- brake all
brake_2 <= '0';
brake_3 <= '0';
        end if;
    end if;
end process Brake_proc;

    PWM <= sig_PWM;
    PWM_inv <= sig_PWM_inv;

end IMP;

```