e-Informatica Software Engineering Journal, Volume 5, Issue 1, 2011, pages: 77–85, DOI 10.2478/v10233-011-0032-2

VERSITA

Multiple tasks in FPGA-based programmable controller

Zbigniew Hajduk*, Jan Sadolewski*, Bartosz Trybus*

*Faculty of Electrical and Computer Engineering, Department of Computer and Control Engineering, Rzeszow University of Technology

zhajduk@kia.prz.edu.pl, js@kia.prz.edu.pl, btrybus@kia.prz.edu.pl

Abstract

An FPGA-based execution platform for PLC controllers with capability to run multiple control tasks is presented. The platform, called multi-CPCore, uses hardware virtual machines to execute control tasks defined in CPDev engineering environment. The tasks consist of one or more programs written in IEC 61131-3 languages, such as ST, IL or FBD. They may run with different cycles and communicate via global variables. Parallel programming mechanisms like process image and semaphores are provided to handle potential conflicts when accessing shared resources.

1. Introduction

CPCore (Control Program Core) is an execution platform for programmable logic controllers (PLCs) designed in FPGA technology. Hardware-implemented IEC 61131-3 virtual machine [1, 2] is its main feature. CPCore is programmed in CPDev engineering environment (*Control Program Developer*) [3], which integrates tools for programming, simulation, hardware configuration, on-line testing and running control applications on different platforms. Programs can be written in ST and IL textual languages (Structured Text, Instruction List) and in FBD graphical language (Function Block Diagram). CPDev compiler produces universal executable code called VMASM (Virtual Machine Assembler), interpreted at the target controller by the CPDev virtual machine (runtime). Software virtual machine written in C is available for multiple target platforms with general-purpose CPUs (ARM, AVR or x86). However, interpretation takes time, so portability of the VMASM code is achieved at the price of slower program execution.

On the contrary, CPCore platform involves FPGA-based hardware virtual machine which directly executes VMASM code. This results in much shorter execution time, from several to a few hundred times, when compared with typical microcontrollers [1]. Similar solutions have been presented in [4, 5]. The CPCore hardware machine is actually a 32-bit microcontroller that executes VMASM code generated in CPDev environment. The microcontroller is built according to Harvard architecture with separate data and program buses. A prototype PLC controller implemented in CPCore technology is shown in Fig. 1. The main board (upper left) includes Xilinx FPGA chip, real-time clock, SRAM, NVRAM and Flash memories. Analog and binary inputs and outputs are handled by slave boards (right side of Fig. 1). The operating panel (lower left) involves LCD display, LEDs and push buttons.

Ability to run multiple IEC control tasks at the same time has been introduced to the CPCore platform recently and is described here.

[1] The research has been supported with MNiSzW grant N N514 412736 (2009-2011).

Each task is executed by a single hardware machine core. The cores are independent and run in parallel, creating a *multiprocessor* architecture (multi-CPCore).

The paper is organized as follows. At first, program execution is characterized. Then the multiple task capabilities are described from the programmer's viewpoint, with the process of creating tasks and setting their parameters. Main aspects of resource sharing are covered, i.e. accessing common hardware blocks and using semaphores for mutual exclusion. Finally, hardware structure of the multi-CPCore platform is presented.

2. Program execution

Source programs for the CPCore controller are processed by the CPDev compiler which generates VMASM universal executable code [3]. The VMASM code can be executed by the controller hardware machine (*executor*). Functional side of the machine corresponds to IEC 61131-3 standard [2] and provides the following capabilities:

- Handling IEC data types: Boolean BOOL, integer BYTE, SINT, INT, WORD, DINT, LINT, DWORD, LWORD, real REAL, LREAL, time and date TIME, DATE, TIME_OF_DAY, DATE_AND_TIME.
- Execution of functions: arithmetic ADD, SUB, MUL, DIV, MOD, numerical SQRT, LOG, SIN, ASIN, EXP, Boolean NOT, AND, OR, XOR, bit shift SHL, ROL, comparison GT, GE, LT, EQ and others.
- Program flow control by means of jumps JMP, JZ, JNZ, calls of function blocks CALB, early exit RETURN, memory handling MCD, MEMCP (Move from Code to Data, Memory Copy).
- Hardware function blocks: invoking blocks implemented in FPGA hardware (called also *native blocks*).
- Parallel programming instructions: global variable handling (Sec. 4.1), semaphore operations LOCK, UNLOCK, TRYLOCK (Sec. 4.3).

The IEC standard also defines multi-element variable types, i.e. arrays and structures. The machine handles these two types by means of a few dedicated commands, e.g. AURD/AUWD read/write data from/to indexed array.

Basic logical registers of the hardware machine are listed in Table 1. Since accumulator does not exist in VMASM specification, results of commands are stored in variables. *Task cycle* can be configured and monitored by the machine during program execution. *Actual task cycle* (last value) is particularly useful for on-line testing (commissioning). *Status1* stores exception flags, including cycle overflow, therefore appropriate reaction can be programmed.

3. CPCore programming with multiple tasks

In the multi-CPCore solution, each task contains its own control algorithm compiled into VMASM. The task is executed by separate instance of the hardware machine. This means that multi-CPCore can be viewed as a group of virtual controllers.

According to IEC 61131-3 standard, engineering project of control system is created in hierarchical manner, i.e. by defining controller configuration, implementing algorithms in programs and function blocks, and assigning them to tasks.

3.1. Creating POUs

In the CPDev environment, the user creates a set of so-called *Program Organizational Units* (POUs). The POUs can be written in ST language or two other languages of IEC 61131-3 standard, i.e. IL and FBD. Main window of CPDev in Fig. 2 shows sample configuration with four POUs, whose names are seen in the project tree on the left. The tree also contains global variables used in the project. START, STOP and ALARM represent digital inputs, while the other, MOTOR, PUMP, OUT0...OUT3 are outputs (all are BOOLs). The global variables are followed by two tasks (described later) and libraries with blocks used by POUs. The first

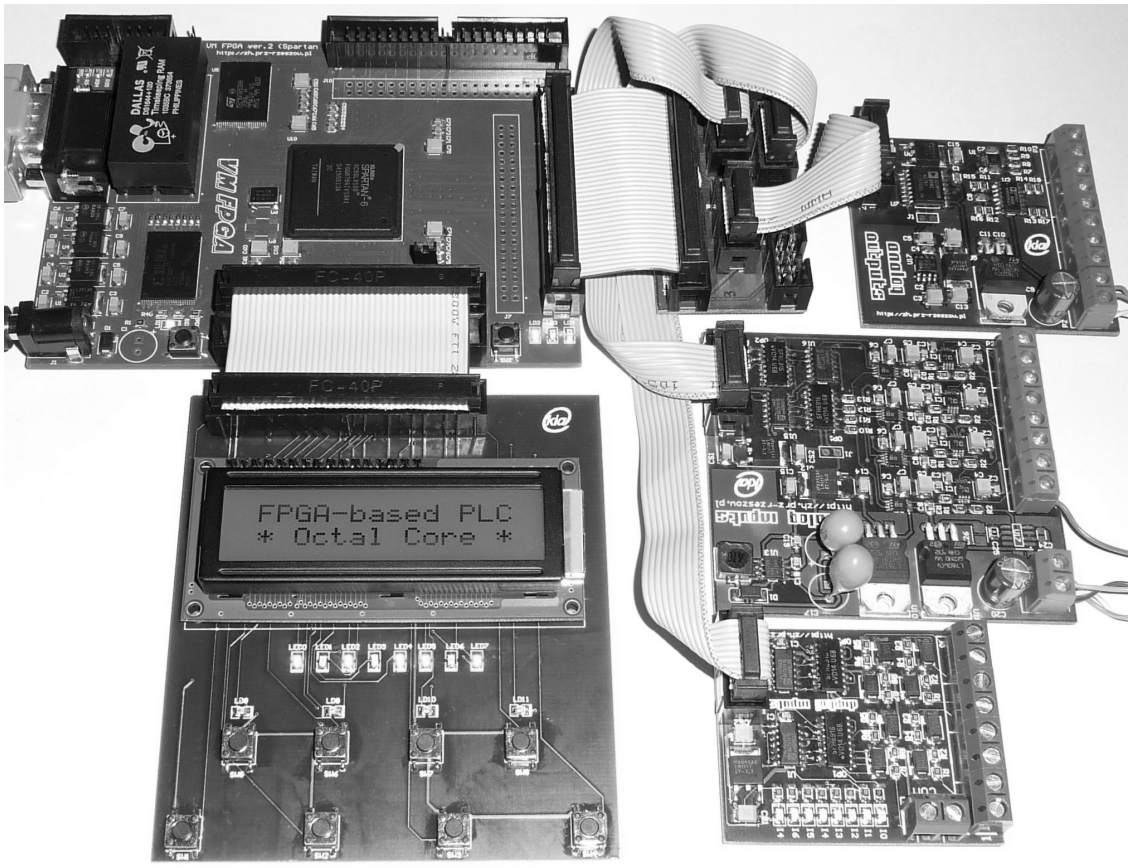

Figure 1. CPCore controller prototype

POU, START_STOP, has been created as an FBD diagram (center part of Fig. 2). It turns MOTOR on if START is pressed, provided that STOP and ALARM are not set. MOTOR continues running after releasing START. PUMP is turned on and off 5 seconds after the MOTOR. Time delay (T#5s) is introduced by two function blocks, TON and TOF. The second POU, MOVE_UNIT, subsequently turns on and off a set of devices in a loop. The algorithm written in ST (right side of Fig. 2) sequentially sets to TRUE one of the global variables OUT0...OUT3 assigned to binary outputs. It is done every 2 seconds (t#2s) by using system clock (CUR_TIME function).

Besides START_STOP and MOVE_UNIT, there are also two other POUs in the project tree, namely LCD_CH and DISPB. The first one is a hardware function block which puts a character onto the CPCore LCD. DISPB is another function block which uses LCD_CH internally to display a string. The block can be invoked by other POUs to print some messages. The two blocks will be described in Sec. 4.2.

3.2. Defining tasks

The user creates a task by selecting appropriate POUs and assigning them to the task. The POUs assigned to the task are executed sequentially in the order defined by the user. The task can group a set of POUs written in different IEC languages. In the sample project of Fig. 2 two task are defined, TASK_SS and TASK_MU. Creation of TASK_SS will be described in more detail.

Task definition is done in CPDev window of Fig. 3. Task name must be entered first, so TASK_SS here. Then task type is selected to indicate execution mode. TASK_SS will be executed cyclically, with cycle time of 1 millisecond. Cyclic task is most common choice, but one can also select “As soon as possible” (endless

Table 1. Logical registers of hardware machine

Register	*Function*
Program counter	Indicates next VMASM command
Data offset	Index to data area being used
Call stack pointer Data stack pointer	Pointers to call stack (POUs) and data stack
Task cycle Actual task cycle	Configured and measured task cycle
Cycle counter	Counts cycles (from reset)
Status1	VM status word (array index faulty, time cycle exceeded, cold start, etc.)
RTC clock	Absolute time

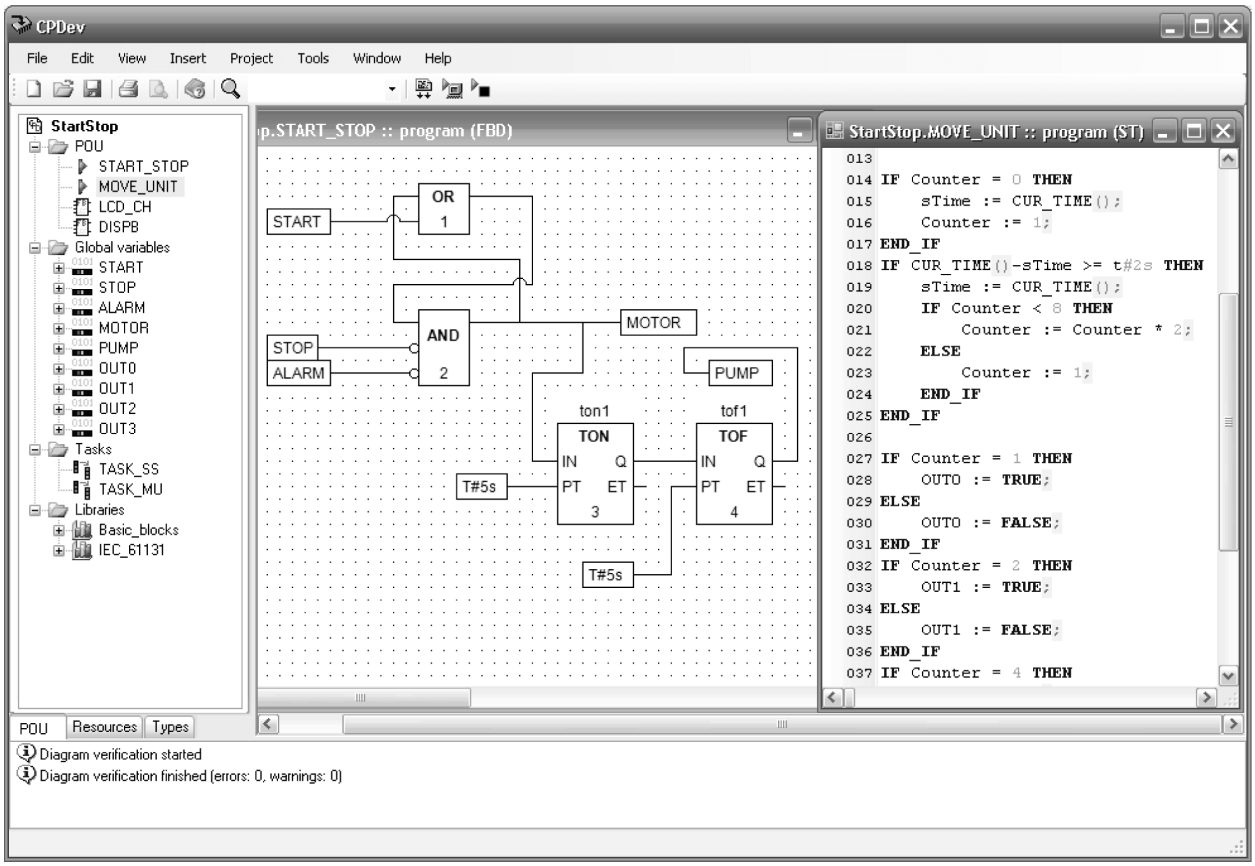

Figure 2. CPDev environment window with two POUs

loop) or single execution (not implemented yet in multi-CPCore prototype). POU assignment is done by moving available programs from the right list to the left (Fig. 3). Two POUs are available here, START_STOP and MOVE_UNIT. The other POUs of the project, LCD_CH and DISPB, do not appear in the window because they are function blocks, not programs. In case of TASK_SS, only START_STOP is selected for execution. The other task, TASK_MU (Fig. 2), involves the MOVE_UNIT program and is also executed cyclically with period of 10 ms.

4. Task communication and resource sharing

As mentioned before, the tasks in multi-CPCore are run independently by their own executors.

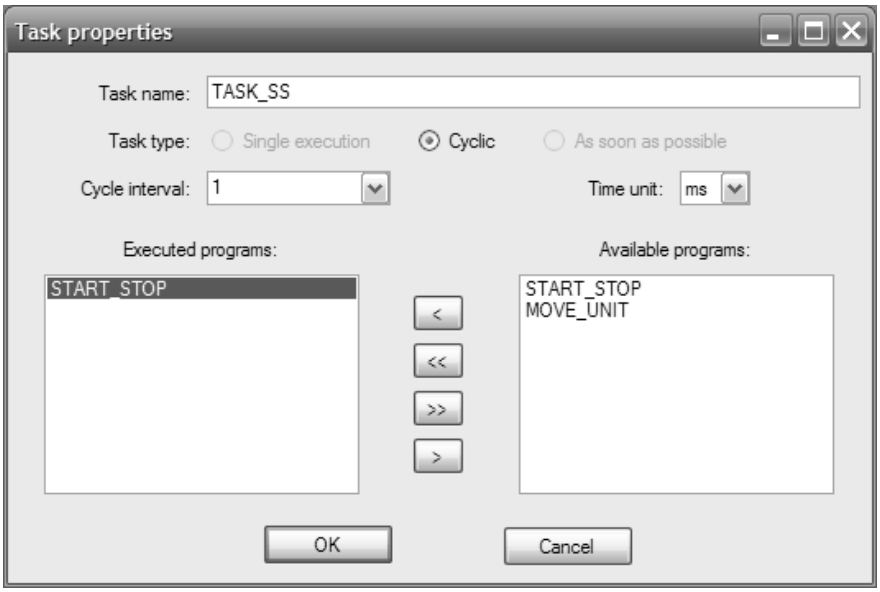

Figure 3. Defining a task for multi-CPCore controller in CPDev

However, as parts of control project, they must communicate and exchange variables. Basic problem in parallel programming is to get access to shared resources. Multi-CPCore can be viewed logically a set of virtual PLCs, which share the same peripherals (inputs and outputs, display, real-time clock, etc.).

4.1. Global variables

Data exchange between CPCore tasks is performed by means of global variables. Such way of task communication is also recommended in IEC standard [2]. Global variables can be accessed by programs and tasks. In the sample project of Fig. 2, START and STOP are used in both tasks (TASK_SS and TASK_MU) to activate corresponding devices (not shown, however, in the part of MOVE_UNIT code).

Upon start of the configuration, multi-CPCore executes special initialization code generated by CPDev compiler, which sets initial values of the global variables (e.g. STOP:=FALSE). This is done before any task is invoked.

To avoid conflicts related to sharing global variables between tasks, CPCore executors operate on so-called *process images*. At the start of the cycle, the task is provided with current copy of the global variables (local shadows). When the task is executed, only the shadows are used, so change of global values caused by other tasks does not affect calculations. When the cycle is about to end, the calculated shadows are stored in the global variables. Synchronization is done only for the variables that have been modified within the cycle.

4.2. Accessing hardware blocks

Two types of function blocks are available in the CPCore platform, i.e. program blocks and hardware blocks. The first ones are created in CPDev environment in one of IEC languages. The platform also supports a set of dedicated hardware blocks used to access low-level functions or to speed up calculations.

As an example, access to the hardware block LCD_CH mentioned above is described now. LCD_CH displays a character on the controller LCD. First, the user creates a new function block (new POU) and chooses ST as implementation language. However, instead of writing an implementation, only the following declaration of the block is entered:

```
FUNCTION_BLOCK LCD_CH
 (*$HARDWARE_BODY_CALL ID:0004; Align:4;
                                Extra:0;*)
```

```
 (*$PLACE_UID_VAR*)
VAR_INPUT
  C:BYTE;
END_VAR
VAR_OUTPUT
END_VAR
END_FUNCTION_BLOCK
```

As seen, single block input is declared (a BYTE value) and no outputs. There is also no body code, however the directive *HARDWARE_BODY_CALL* instructs the compiler to assign the declaration to particular hardware. In CPCore, the LCD_CH block has a unique identifier 4 (ID:0004). After the declaration is entered, an instance of the block can be created and invoked from a program as any other function block.

However, since hardware blocks usually control peripherals, their usage in parallel environment is somewhat limited due to potential conflicts. Typically, such block cannot be concurrently executed by two or more tasks. This is called *mutual exclusion* and can be achieved in two ways.

Some hardware block calls from multiple tasks are queued internally by the CPCore and then executed sequentially. Task execution may be delayed due to queue processing, but the collision does not occur. This mechanism applies mostly to simple blocks executed in one-shot manner. LCD_CH and some flip-flops are examples of queued hardware blocks.

Assignment of hardware block to particular executor (virtual machine) during configuration of CPCore is another way of avoiding conflicts during the calls. In such arrangement, only that executor will be able to call the block. Other tasks cannot call the block directly, however a software solution can be implemented to provide access to the block functionality via a dedicated task. In CPCore this applies to UART and 1-wire interface handling blocks.

Fig. 4 shows CPDev *hardware configurer* window which allows to set up hardware blocks for CPCore controller. The upper area activates blocks related to standard peripheral services (UART, LCD, 1-wire interface) and a type conversion block. The lower part contains a list of IEC standard blocks implemented in hardware. Contrary to software-implemented blocks they are executed extra fast, so the overall performance of the algorithm is increased. In case of CPCore, one can use RS and SR flip-flops, counters, triggers and timers. Here, two instances of TON and TOF blocks are defined, for instance to be used in START_STOP diagram (Fig. 2). According to the settings, the hardware configurer generates appropriate libraries for the CPCore FPGA chip.

4.3. Mutual exclusion with semaphores

Sometimes hardware solutions described above are not sufficient to provide collision-free native block calls. The problem arises especially when a hardware block is called in the code of another function block. For example, DISPB is a conventional function block written in ST, used to print a string on LCD display. DISPB executes actual printing by calling LCD_CH hardware block for every character. Although DISPB can be used by any task, if printing loop is in progress in one task, the other tasks cannot get access to the display. Otherwise consistency of the display would be violated.

Multi-CPCore programmer can use *semaphores* for task synchronization and mutual exclusion. A semaphore is a global integer variable accessed from tasks by LOCK and UNLOCK functions. Unlocking increments semaphore value, while locking decrements it. When the semaphore value is zero, the locking task is suspended until one of other tasks unlocks the semaphore. Semaphores prohibit tasks from running critical part of code, when that part is currently executed.

To provide mutual exclusion upon DISPB call, a semaphore should be created, i.e. VSEM below, with initial value 1. Then the following code protects the printing loop from re-entry.

```
LOCK(VSEM);
FOR i:=1 TO 16 DO       (* 16 chars are kept *)
 LCD_PUTCHAR(c:=str[i]) (* in str array *)
END_FOR
UNLOCK(VSEM);
```

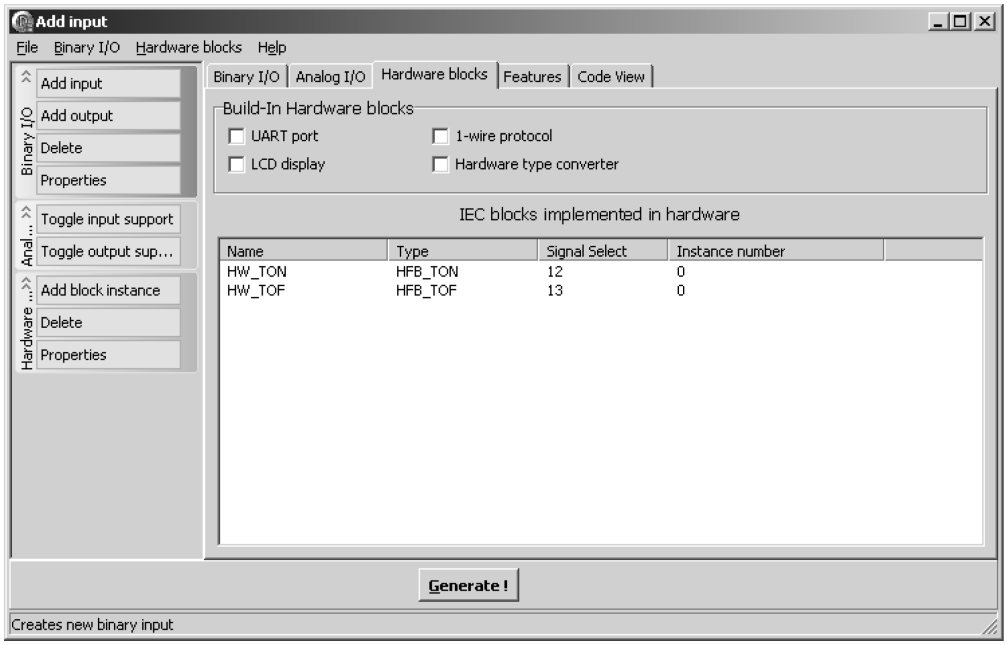

Figure 4. Configuring hardware blocks for the CPCore controller

When a task enters the code, it locks the semaphore by decrementing it. So the printing begins. When another task tries to enter the code while printing, the semaphore value is zero, so that task will be suspended and queued. After the first task unlocks the semaphore, one of the queued tasks is resumed and can execute another printing.

5. Hardware structure

Simplified block diagram of the multi-CPCore controller implemented in FPGA is shown in Fig. 5. The design is based on symmetric multiprocessor architecture [6]. Multiple hardware machine cores in the center of Fig. 5 are hardware machines (executors) which run concurrently. Implementation of the hardware machine core has been described in [1]. In the actual CPCore they are additionally equipped with a floating point unit [7] (useful for continuous control). Each core, being in fact a specialized microprocessor with dedicated program and data memories, is responsible for execution of a single task.

Apart from the machine cores, there is also another unit, called *initiator core*. It is a simple processor responsible for initialization of selected locations in global variable memory. The initiator core is triggered on power up or after a new configuration is loaded into the controller. Only after completing the initialization, other cores begin to work.

As described in Sec. 4.1, communication between machine cores is implemented through common global memory. Collision free access to that memory is provided by the global memory and I/O access arbiter block (Fig. 5). Handshaking protocol is applied for data transfer between the cores and the arbiter. This is a part of the process image mechanism described in Sec. 4.1.

A core, which needs to access the global memory, sets a request signal. The arbiter successively analyses request occurring at its ports and grants access to the global memory. Granting the access to particular core is confirmed by acknowledgment signal. At the end of data transfer, the core releases the request. In response, the arbiter releases the acknowledgment and begins scanning other ports for request signals.

The global memory address space is divided into two ranges. The lower range, starting from address 0 up to a configured value, is reserved for addressing input/output devices (peripherals), such as digital input and output modules. The upper range maps physical synchronous RAM

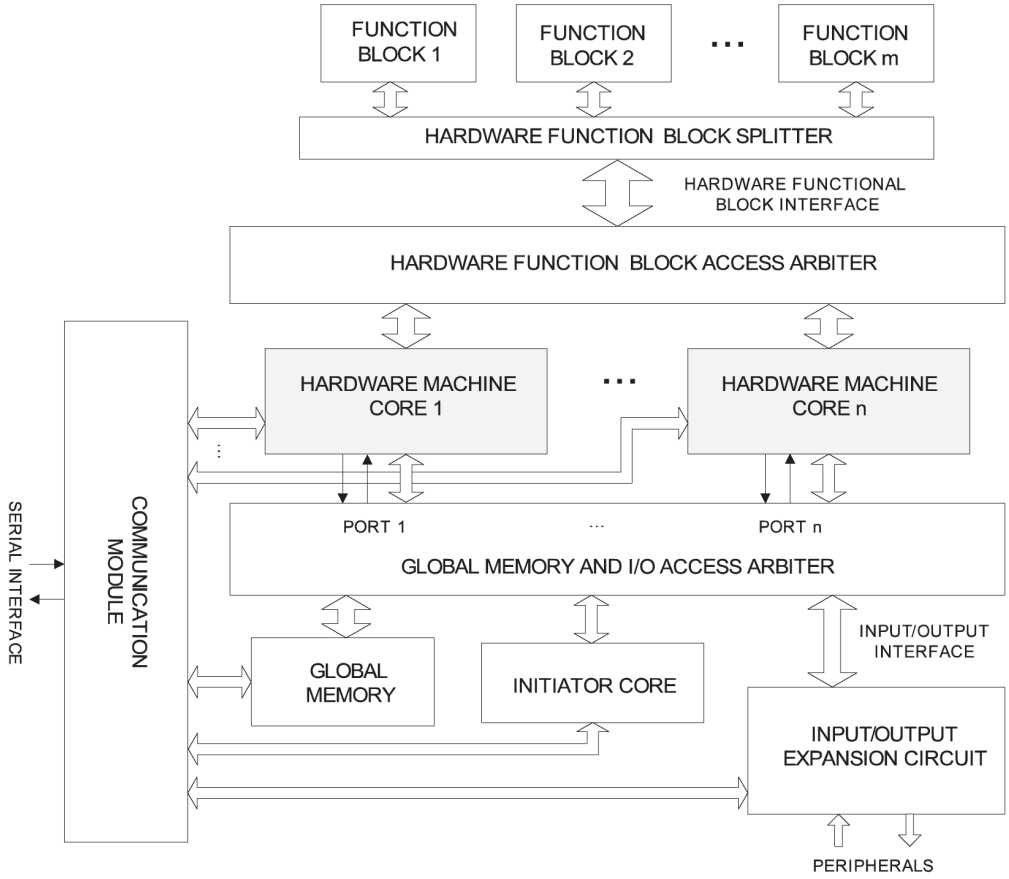

Figure 5. Simplified block diagram of multi-CPCore FPGA implementation

memory, used to hold global variables. Hardware machine cores access the input/output devices in the same way as accessing the global memory (i.e. via the arbiter). Additional expansion circuit is needed to connect peripherals to input/output interface of the arbiter block.

The CPCore FPGA-based controller has been equipped with facility to integrate hardware function blocks. Special mechanism for data transfer between such blocks and executing machines has been designed and implemented. As described in Sec. 4.2, there are two options for connecting hardware blocks. The first one assumes that each core has its own set of hardware blocks. The second one shown in Fig. 5 implements the idea of sharing hardware blocks among executing cores. In this case, each core can invoke any of available blocks. This capability requires the use of arbiter block, which ensures collision-free access to the blocks. The hardware function block access arbiter (Fig. 5) operates similarly to the global memory access arbiter. However, the hardware function block splitter, which consists mainly of a set of multiplexers, is additionally required to connect the arbiter to hardware blocks.

Communication module is an important component of the CPCore platfrom [8]. It provides data transfers with CPDev environment, especially for on-line monitoring and commissioning purposes. The communication module ensures full read and write access to the global variable area, as well as to program and data memories of each hardware machine core. It also performs special functions like in-circuit debugging.

A prototype controller with multi-CPCore technology shown earlier in Fig. 1 consists of eight executing machine cores what allows for execution of up to eight concurrent control tasks. Four hardware function blocks (UART, alphanumeric LCD, 1-wire bus, hardware type conversion) are available. The prototype has been implemented in Xilinx Spartan-6 FPGA

XC6SLX100-3FGG676, with the main circuit board described in [9]. The chip employs 31099 6-input LUTs (49% of all available in FPGA) and 13513 slice registers (10%).

6. Summary

Multi-CPCore FPGA-based PLC execution platform has been described. The platform integrates several hardware machines, each handling one control task. FPGA implementation results in short execution times, if compared to standard microcontroller-based solutions. The CPCore tasks run concurrently and independently, as a set of virtual PLCs. Each task can be set up with its own cycle time. As a result, CPCore controller can handle different applications at the same time. For example, it can execute fast logic control concurrently with continuous control, and also handle HMI operating panel. Such functionalities are available in industrial computers, but CPCore technology can be applied for much smaller devices.

Multi-CPCore is programmed and configured in CPDev engineering environment, compatible with IEC 61131-3 standard. The tasks are composed of programs written in ST, FBD or IL languages. In addition to standard libraries, a library of hardware blocks is available to access peripherals and speed up operations. Task synchronization mechanisms have been developed to eliminate conflicts while accessing hardware resources in parallel environment.

References

[1] Z. Hajduk, B. Trybus, and J. Sadolewski, "Hardware implementation of virtual machine for programmable controllers," in *Metody wytwarzania i zastosowania systemów czasu rzeczywistego*, L. Trybus and S. Samolej, Eds. Warszawa: Wydawnictwa Komunikacji i Łączności, 2010, ch. Chapter 5, pp. 333–342, (in Polish).

[2] *IEC 61131-3 standard: Programmable Controllers – Part 3. Programming Languages*, IEC Std., 2003.

[3] D. Rzońca, J. Sadolewski, A. Stec, Z. Świder, B. Trybus, and L. Trybus, "Open environment for programming small controllers according to IEC 61131-3 standard," *Scalable Computing: Practice and Experience*, vol. Volume 10, no. 3, pp. 325–336, 2009.

[4] D. Gawali and V. Sharma, "FPGA Based Micro-PLC Design Approach," *Advances in Computing, Control, and Telecommunication Technologies, International Conference on*, vol. 0, pp. 660–663, 2009.

[5] M. Adamski and J. L. Monteiro, "From interpreted Petri net specification to reprogrammable logic controller design," in *Proc. IEEE Int. Symp. Industrial Electronics (ISIE 2000)*, vol. 1, 2000, pp. 13–19.

[6] P. Huerta, J. Castillo, C. Pedraza, J. Cano, and J. I. Martinez, "Symmetric multiprocessor systems on FPGA," in *IEEE Int. Conf. on Reconfigurable Computing and FPGAs*. ReConFig '09, 2009, pp. 279–283.

[7] Z. Hajduk, "Floating-point arithmetic unit for a hardware virtual machine," *Pomiary Automatyka Kontrola*, vol. 57, no. 1, pp. 82–85, 2011, (in Polish).

[8] ——, "Communication module for a hardware implemented virtual machine," *Elektronika – konstrukcje, technologie, zastosowania*, no. 5, 2011, (in Polish).

[9] ——, "PLC controller prototype with a hardware virtual machine," *Elektronika – konstrukcje, technologie, zastosowania*, no. 4, pp. 114–118, 2011, (in Polish).