https://doi.org/10.29042/2020-10-5-48-61

Helix (2020) 10 (5): 48-61

Software Models in VHDL Language as a Component of a Dispatching Subsystem of the Embedded Reconfigurable Computing System

Alexey I. Martyshkin
Candidate of Engineering Sciences, Associate Professor, Sub-Department "Computers and systems"
Penza State Technological University, 440039, Russia, Penza, Baydukova Passage / Gagarin Street, 1a/11,
alexey314@yandex.ru

Received: 21st August 2020, Accepted: 14th September 2020, Published: 31st October 2020

Abstract
The paper presents the results of research on the hardware implementation of dispatching functions of an embedded reconfigurable computing system with a single task queue and distributed task queues. The description of the dispatching subsystem component functioning at the hardware level was made in the VHDL language. The purpose and functionality of each unit implemented in the VHDL language is described. Debugging and testing of the developed software models for a dispatching subsystem of an embedded reconfigurable computing system is carried out and the results are analyzed. At the end, conclusions for the work are drawn.

Keywords
Hardware Implementation, Task Manager, Processor, Reconfigurable Computing System, Digital Signal Processing, VHDL Language.

Introduction
Analysis of the computer technology development allows us to identify a tendency towards a gradual transition from software implementation of a number of algorithms performed by computers to hardware. In modern computers, the hardware implementation of multiplication and division algorithms, floating point operations, double precision arithmetic, loop organization, indexing and some other operations has become generally accepted; in the first computers those operations were implemented in software. The transition from software implementation of algorithms to hardware implementation is caused by the desire to increase the performance / cost ratio of computers and is due to the rapid development of electronics. Indeed, the hardware implementation of any algorithm makes it possible to reduce the execution time, and, consequently, to increase the computer performance. On the other hand, the cost of additional equipment for the hardware implementation of algorithms decreases every year thanks to the progress of the element base. Many foreign experts believe that the time for the hardware implementation of operating system algorithms is coming.

Theory
In the paper, the issues of hardware implementation of the process synchronization algorithm for scheduling and task dispatching will be considered on the example of a system consisting of four processors (CPUs) of the same speed united by a common bus (CB). The synchronizing device has its own interface for interaction with each CPU and does not interact with the common bus, so as not to load it and not interfere with the functioning of the CPU. Instead, each CPU has a simplest interface for requesting and receiving task IDs from the task manager (TM). This approach sophisticates the CPU unit by supplementing it with new signals and requiring them to be processed, but at the same time it is obvious that a task manager that does not use a common bus does not affect the operation of other devices connected with it [1-3] and [4-6]. The general view of the investigated reconfigurable computing system (RCS) is shown in Figure 1.

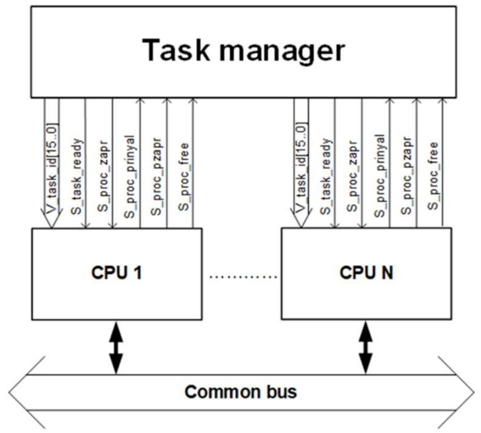

Figure 1: Reconfigurable Computing System with Dedicated Hardware Time-sharing Task Manager

The advantages of such an organization include the fact that the failure of one of the CPUs has absolutely no effect on the performance of the entire reconfigurable computing system. Even if there is only one working CPU in it, it will remain operational, although its performance will be significantly reduced. The interface for interaction between the CPU and the task manager is discussed in detail in [7-10]. The whole procedure for obtaining by a CPU of a task ID via this interface takes no more than 10 clock cycles. In hardware, a time-sharing task manager can be represented in the form of 4 units (Figure 2).

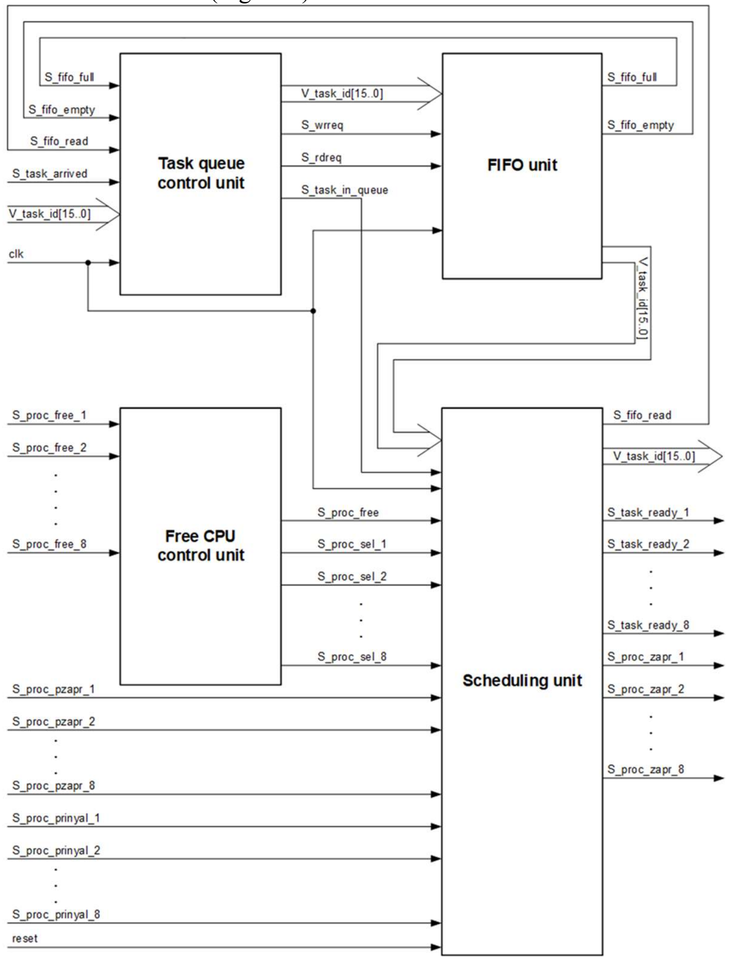

Figure 2: Structural Organization of a Hardware Time-sharing Task Manager

The task queue control unit (TQCU) is responsible for the receipt of tasks in the queue. It monitors the status of the task queue using two signals: *S_fifo_full* and *S_fifo_empty*. A single signal *S_fifo_full* indicates that new tasks cannot be accepted, since there are no free registers for writing and storing new task IDs. A single signal *S_fifo_empty* indicates that the FIFO queue is empty. This signal, on the basis of which the decision is made whether to wake the CPU from sleep mode or not, is translated to the synchronizing unit in the *S_task_in_queue* line. If the task queue is empty, then the sync unit does not access the CPU. The task queue control unit also includes a 16-bit data bus (*V_task_id [15 ... 0]*), through which task IDs are transmitted. The fact that there is a new task ID on the common bus is notified to the scheduler by the rising edge of the *S_task_arrived* signal. If the FIFO is not full, the scheduler reads data *V_task_id [15 ... 0]* from the data bus and writes it to the FIFO at the end of the list. Since the task queue control unit directly controls the FIFO, therefore, when the scheduling unit (SU) needs to read the task ID from the FIFO in order to transfer it to the CPU, the scheduling unit informs the scheduler about it via the *S_fifo_read* signal.
The FIFO unit stores incoming task IDs.
Free Processor Control Unit (FPCU) reads signals *S_proc_free* from all CPUs and to a reconfigurable computing system in the system. It determines whether there are currently free CPUs in the reconfigurable computing system, and, if there is at least one free CPU, sets the single signal *S_proc_free* and signals the scheduling unit about availability of free CPUs. Also, there is a priority scheme inside the free processor control unit that unambiguously selects one of the CPUs to serve the current task, based on the principle of CPU occupancy at the current time. In order to inform the scheduling unit about which of the CPUs is selected for service, there are signals *S_proc_sel_1,*

..., *S_proc_sel_4*. A single signal at each moment of time can be only at one of these conclusions, or be absent altogether. In this case, the dispatcher always knows which free CPU should be assigned to serve the current task. The scheduling unit (SU) directly interacts with the CPU of the reconfigurable computing system. In addition, the scheduling unit interacts with other units of the reconfigurable computing system. The task queue control unit sends information about which free CPU is currently selected to serve the task. The scheduling unit receives from the FIFO unit the task IDs via the input data bus. The task queue control unit informs the scheduling unit about the state of the queue, and fetches task IDs from the FIFO at the request of the scheduling unit.
After the CPU has received the current task ID, it leaves the free CPU pool and the task manager would not consider it at all. The task manager goes to assign a new CPU. It is "not interested" in what happens to a task that ends up in the CPU and is served there. Of course, there must be mechanisms to control how many time slices the CPU is serving a task so that some task does not take up the entire time the CPU is running. Such mechanisms are implemented at the level of the operating system (OS) kernel.

Simulation of Device Operation

The task manager model was implemented in software in the VHDL language in the form of four modules, the first of which implements the logical functions of the task queue control unit; the second implements the FIFO unit; the third implements the free processor control unit, and the fourth one simulated the scheduling unit. Below are the chunks of code to implement some of the above modules.

The Implementation of a Task Queue Control Unit Using Simulation in VHDL Language

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
ENTITY unit_upr_ocher IS
 .. PORT
 .. (
 ..          CLK                  : IN     STD_LOGIC;
 ..          V_task_id            : IN     STD_LOGIC_VECTOR(15 DOWNTO 0);
 ..          S_task_arrived  : IN     STD_LOGIC;
 ..          S_fifo_full          : IN     STD_LOGIC;
 ..          S_fifo_empty    : IN     STD_LOGIC;
 ..          S_fifo_read          : IN     STD_LOGIC;
 ..          V_fifo_data_o   : OUT    STD_LOGIC_VECTOR(15 DOWNTO 0);
 ..          S_wrreq         : OUT    STD_LOGIC;
 ..          S_rdreq         : OUT    STD_LOGIC;
 ..          S_task_in_queue : OUT    STD_LOGIC
 .. );
END unit_upr_ocher;
ARCHITECTURE unit_upr_ocher_arch OF unit_upr_ocher IS
 .. SIGNAL reg_task_id : STD_LOGIC_VECTOR(15 DOWNTO 0);
 .. SIGNAL S_task_arrived1 : STD_LOGIC;
 .. SIGNAL S_task_arrived2 : STD_LOGIC;
BEGIN
PROCESS
BEGIN
 .. WAIT UNTIL CLK='1';
 .. S_task_in_queue<=not S_fifo_empty;
 .. reg_task_id<=V_task_id;
 .. S_task_arrived1<=S_task_arrived;
 .. S_task_arrived2<=S_task_arrived1;
 .. IF S_task_arrived1='0' AND S_task_arrived2='1' THEN
 ..          IF S_fifo_full='0' THEN
 ..                      S_wrreq<='1';
 ..                      V_fifo_data_o<=reg_task_id;
 ..          END IF;
 .. ELSE
 ..          S_wrreq<='0';
 .. END IF;
 .. IF S_fifo_read='1' THEN
 ..          S_rdreq<='1';
 .. ELSE
 ..             S_rdreq<='0';
 .. END IF;
END PROCESS;
END unit_upr_ocher_arch;
```

Implementation of the Free Processor Control Unit in VHDL Language

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
ENTITY unit_upr_sv_proc IS
 .. PORT
 .. (
 ..             S_proc_free_1                 : IN      STD_LOGIC;
 ..             S_proc_free_2                 : IN      STD_LOGIC;
 ..             S_proc_free_3                 : IN      STD_LOGIC;
 ..             S_proc_free_4                 : IN      STD_LOGIC;
 ..             S_proc_free                             : OUT    STD_LOGIC;
 ..             S_proc_sel_1                  : OUT     STD_LOGIC;
 ..             S_proc_sel_2                  : OUT     STD_LOGIC;
 ..             S_proc_sel_3                  : OUT     STD_LOGIC;
 ..             S_proc_sel_4                  : OUT     STD_LOGIC;
 .. );
END unit_upr_sv_proc;
ARCHITECTURE unit_upr_sv_proc_arch OF unit_upr_sv_proc IS
 .. SIGNAL V_proc_free : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
 .. S_proc_free<=S_proc_free_1 or S_proc_free_2 or S_proc_free_3 or S_proc_free_4;
 .. V_proc_free<=S_proc_free_4 & S_proc_free_3 & S_proc_free_2 & S_proc_free_1;
 .. S_proc_sel_4<='1' WHEN V_proc_free(7 downto 3)="00001" ELSE '0';
 .. S_proc_sel_3<='1' WHEN V_proc_free(7 downto 2)="000001" ELSE '0';
 .. S_proc_sel_2<='1' WHEN V_proc_free(7 downto 1)="0000001" ELSE '0';
 .. S_proc_sel_1<='1' WHEN V_proc_free(7 downto 0)="00000001" ELSE '0';
END unit_upr_sv_proc_arch;
```

Implementation of the Scheduling Unit in VHDL Language

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
ENTITY unit_dispetcher IS
 .. PORT
 .. (
 ..             CLK                           : IN STD_LOGIC;
 ..             rst                           : IN STD_LOGIC;
 ..             S_task_in_queue     : IN      STD_LOGIC;
 ..             S_proc_free                   : IN STD_LOGIC;
 ..             V_q_fifo_in                   : IN STD_LOGIC_VECTOR(15 downto 0);
 ..             S_fifo_read_out     : OUT     STD_LOGIC;
 ..             V_task_id_for_proc  : OUT STD_LOGIC_VECTOR(15 downto 0);
 ..             S_task_ready_1      : OUT   STD_LOGIC;
 ..             S_task_ready_2      : OUT   STD_LOGIC;
 ..             S_task_ready_3      : OUT   STD_LOGIC;
 ..             S_task_ready_4      : OUT   STD_LOGIC;
 ..             S_proc_sel_1        : IN      STD_LOGIC;
 ..             S_proc_sel_2        : IN      STD_LOGIC;
 ..             S_proc_sel_3        : IN      STD_LOGIC;
 ..             S_proc_sel_4        : IN      STD_LOGIC;
 ..             S_proc_pzapr_1      : IN      STD_LOGIC;
 ..             S_proc_pzapr_2      : IN      STD_LOGIC;
.. 	S_proc_pzapr_3		: IN	STD_LOGIC;
.. 	S_proc_pzapr_4		: IN	STD_LOGIC;
.. 	S_proc_prinyal_1		: IN	STD_LOGIC;
.. 	S_proc_prinyal_2		: IN	STD_LOGIC;
.. 	S_proc_prinyal_3		: IN	STD_LOGIC;
.. 	S_proc_prinyal_4		: IN	STD_LOGIC;
.. 	S_proc_zapr_1_o		: OUT	STD_LOGIC;
.. 	S_proc_zapr_2_o		: OUT	STD_LOGIC;
.. 	S_proc_zapr_3_o		: OUT	STD_LOGIC;
.. 	S_proc_zapr_4_o		: OUT	STD_LOGIC;
.. );
END unit_dispetcher;
ARCHITECTURE unit_dispetcher_arch OF unit_dispetcher IS
.. SIGNAL reg_task_id : STD_LOGIC_VECTOR(15 DOWNTO 0);
.. TYPE STATE_UPR IS( INIT, ZAPROS_PROC, P_ZAPROS_PROC, READ_TASK,
TASK_IN_PROC, WAIT_PROC_READ_TASK );
.. SIGNAL state			: STATE_UPR;
.. SIGNAL state_next			: STATE_UPR;
.. SIGNAL	S_proc_zapr_1	: STD_LOGIC;
.. SIGNAL	S_proc_zapr_2	: STD_LOGIC;
.. SIGNAL	S_proc_zapr_3	: STD_LOGIC;
.. SIGNAL	S_proc_zapr_4	: STD_LOGIC;
.. SIGNAL	S_proc_g_1		: STD_LOGIC;
.. SIGNAL	S_proc_g_2		: STD_LOGIC;
.. SIGNAL	S_proc_g_3		: STD_LOGIC;
.. SIGNAL	S_proc_g_4		: STD_LOGIC;
.. SIGNAL cnt: STD_LOGIC_VECTOR(3 downto 0):="0000";
BEGIN
.. 	S_proc_zapr_1_o<=S_proc_zapr_1;
.. 	S_proc_zapr_2_o<=S_proc_zapr_2;
.. 	S_proc_zapr_3_o<=S_proc_zapr_3;
.. 	S_proc_zapr_4_o<=S_proc_zapr_4;
PROCESS(rst)
BEGIN
.. IF rst='1' THEN
.. 	state<=INIT;
.. ELSE
.. 	state<=state_next;
.. END IF;
END PROCESS;
PROCESS
BEGIN
.. WAIT UNTIL CLK='1';
.. CASE state IS
.. WHEN INIT =>
.. 		S_proc_zapr_1<='0';
.. 		S_proc_zapr_2<='0';
.. 		S_proc_zapr_3<='0';
.. 		S_proc_zapr_4<='0';
.. 		S_fifo_read_out<='0';
.. 		S_task_ready_1<='0';
.. 		S_task_ready_2<='0';
.. 		S_task_ready_3<='0';
.. 		S_task_ready_4<='0';
.. 		S_proc_g_1<='0';
.. 		S_proc_g_2<='0';
.. 		S_proc_g_3<='0';
.. 		S_proc_g_4<='0';
.. 		V_task_id_for_proc<=x"0000";
.. 		reg_task_id<=x"0000";
..                  state_next<=ZAPROS_PROC;

.. WHEN ZAPROS_PROC =>
..          IF S_task_in_queue='1' and S_proc_free='1' and S_proc_sel_1='1' THEN
..                  S_proc_zapr_1<='1';
..                  state_next<=P_ZAPROS_PROC;
..          ELSIF S_task_in_queue='1' and S_proc_free='1' and S_proc_sel_2='1' THEN
..                  S_proc_zapr_2<='1';
..                  state_next<=P_ZAPROS_PROC;
..          ELSIF S_task_in_queue='1' and S_proc_free='1' and S_proc_sel_3='1' THEN
..                  S_proc_zapr_3<='1';
..                  state_next<=P_ZAPROS_PROC;
..          ELSIF S_task_in_queue='1' and S_proc_free='1' and S_proc_sel_4='1' THEN
..                  S_proc_zapr_4<='1';
..                  state_next<=P_ZAPROS_PROC;
..          END IF;
.. WHEN P_ZAPROS_PROC =>
..          IF (S_proc_zapr_1 and S_proc_pzapr_1)='1' THEN S_proc_g_1<='1';
S_fifo_read_out<='1'; state_next<=READ_TASK;
..          ELSIF (S_proc_zapr_2 and S_proc_pzapr_2)='1' THEN S_proc_g_2<='1';
S_fifo_read_out<='1'; state_next<=READ_TASK;
..          ELSIF (S_proc_zapr_3 and S_proc_pzapr_3)='1' THEN S_proc_g_3<='1';
S_fifo_read_out<='1'; state_next<=READ_TASK;
..          ELSIF (S_proc_zapr_4 and S_proc_pzapr_4)='1' THEN S_proc_g_4<='1';
S_fifo_read_out<='1'; state_next<=READ_TASK;
..          END IF;

.. WHEN READ_TASK =>
..          S_fifo_read_out<='0';
..          IF cnt="0010"THEN
..                  cnt<="0000";
..                  reg_task_id<=V_q_fifo_in;
..                  state_next<=TASK_IN_PROC;
..          ELSE
..                  cnt<=cnt+'1';
..          END IF;

.. WHEN TASK_IN_PROC =>
..          V_task_id_for_proc<=reg_task_id;
..          IF S_proc_g_1='1'      THEN                    S_task_ready_1<='1';
state_next<=WAIT_PROC_READ_TASK;
..          ELSIF          S_proc_g_2='1'          THEN          S_task_ready_2<='1';
state_next<=WAIT_PROC_READ_TASK;
..          ELSIF          S_proc_g_3='1'          THEN          S_task_ready_3<='1';
state_next<=WAIT_PROC_READ_TASK;
..          ELSIF          S_proc_g_4='1'          THEN          S_task_ready_4<='1';
state_next<=WAIT_PROC_READ_TASK;
..          END IF;

.. WHEN WAIT_PROC_READ_TASK =>
..          IF (S_proc_g_1 and S_proc_prinyal_1)='1' THEN V_task_id_for_proc<=x"0000";
S_task_ready_1<='0'; state_next<=INIT;
..          ELSIF (S_proc_g_2 and S_proc_prinyal_2)='1' THEN V_task_id_for_proc<=x"0000";
S_task_ready_2<='0'; state_next<=INIT;
..          ELSIF (S_proc_g_3 and S_proc_prinyal_3)='1' THEN V_task_id_for_proc<=x"0000";
S_task_ready_3<='0'; state_next<=INIT;
..          ELSIF (S_proc_g_4 and S_proc_prinyal_4)='1' THEN V_task_id_for_proc<=x"0000";
S_task_ready_4<='0'; state_next<=INIT;
..          END IF;
.. END CASE;
```

```
END PROCESS;
END unit_dispetcher_arch;
```

The research was carried out in a free version of the Xilinx ISE WebPACK Design Suite 12.1 system; simulation modelling was carried out there.

Before modelling, we need to enter and describe two more additional units that are not part of the task manager, and which are only needed for the modelling stage.

The first unit is a task generation unit (TGU). When modelling, it is necessary to constantly pass to the task manager the IDs of tasks ready to be executed. Tasks enter the system permanently, and their number can be very large. Therefore, in order not to manually set all the incoming information, a task generation unit is installed at the input of the task queue control unit, which sets a new task ID at regular intervals and independently notifies the task queue control unit, imitating the task flow. To start the task generation unit, it is enough to send a single signal to the start unit input. Thus, when modelling, the assignment of incoming task IDs is greatly simplified.

In addition to the task generation unit, a CPU operation simulation unit (POSU) was additionally created, the purpose of which is not to independently expose all signals of the "CPU-task manager" interface during simulation; by simulating CPU responses, a CPU operation simulation unit is used that fully supports interface and independently interacts with the task manager. The specified unit completely repeats the work of a real CPU. When the task ID is received by the CPU operation simulation unit, a counter is started to count the specified number of clock cycles. At this time, the unit is disconnected from the task manager just as if a real CPU would go on to serve it, having received the task. The time while the counter is counting the set clock cycles is actually the time spent by the CPU processing the task. After the counter has counted the specified number of clock cycles, the CPU operation simulation unit will again inform the task manager that it is free and ready for further interaction. Because the task manager is considered using the example of a system with four CPUs, then there should also be four CPU operation simulation units. Obviously, manually maintaining the Task manager interface with four CPUs would complicate the modelling step. Below are the chunks of code for the implementation of the above modules.

Implementation of the Task Generation Unit in VHDL Language

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
ENTITY unit_task_manager IS
 .. PORT
 .. (
 ..         CLK                 : IN    STD_LOGIC;
 ..         start               : IN    STD_LOGIC;
 ..         V_task_id           : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0);
 ..         S_task_arrived  : OUT   STD_LOGIC
 .. );
END unit_task_manager;
ARCHITECTURE unit_task_manager_arch OF unit_task_manager IS
SIGNAL count1 : STD_LOGIC_VECTOR(15 DOWNTO 0):=x"0000";
SIGNAL count2 : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
V_task_id<=count1;
PROCESS
BEGIN
 .. WAIT UNTIL CLK='1';
 .. IF Start='1' THEN
 ..         count1<=count1+'1';
 ..         IF count2="0101" THEN
 ..                 S_task_arrived<='1';
 ..                 count2<="0000";
 ..         ELSE
 ..                 count2<=count2+'1';
 ..                 S_task_arrived<='0';
 ..         END IF;
 .. ELSE
 ..         count1<=x"0000";
 ..         count2<="0000";
 ..             S_task_arrived<='0';
 .. END IF;
END PROCESS;
END unit_task_manager_arch;
```

Implementation of the CPU operation simulation unit in VHDL language

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
USE ieee.std_logic_arith.all;
ENTITY unit_processor IS
 .. PORT
 .. (
 ..             CLK                                         : IN STD_LOGIC;
 ..             rst                                         : IN STD_LOGIC;
 ..             proc_free                                   : OUT  STD_LOGIC;
 ..             task_prinyal                                : OUT  STD_LOGIC;
 ..             zapr                                        : IN STD_LOGIC;
 ..             pzapr                                       : OUT  STD_LOGIC;
 ..             task_ready                                  : IN STD_LOGIC;
 ..             V_task_id                                   : IN STD_LOGIC_VECTOR(15 downto 0);
 ..             Task_ID                     : OUT    STD_LOGIC_VECTOR(15 downto 0);
 ..             PROC_WORK                   : OUT  STD_LOGIC
 .. );
END unit_processor;
ARCHITECTURE unit_processor_arch OF unit_processor IS
 .. SIGNAL reg_task_id                   : STD_LOGIC_VECTOR(15 DOWNTO 0);
 .. TYPE STATE_PROC IS( INIT, WAIT_ZAPR, WAIT_TASK, WORK_PROC );
 .. SIGNAL state                         : STATE_PROC;
 .. SIGNAL state_next                    : STATE_PROC;
 .. SIGNAL cnt                           : STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
 .. Task_ID<=reg_task_id;
PROCESS(rst)
BEGIN
 .. IF rst='1' THEN
 ..             state<=INIT;
 .. ELSE
 ..             state<=state_next;
 .. END IF;
END PROCESS;
PROCESS
BEGIN
 .. WAIT UNTIL CLK='1';
 ..             CASE state IS
 ..                         WHEN INIT =>
 ..                                     cnt<=x"00";
 ..                                     reg_task_id<=x"0000";
 ..                                     proc_free<='1';
 ..                                     pzapr<='0';
 ..                                     task_prinyal<='0';
 ..                                     PROC_WORK<='0';
 ..                                     state_next<=WAIT_ZAPR;
 ..
 ..                         WHEN WAIT_ZAPR =>
 ..                                     IF zapr='1' THEN
 ..                                                 proc_free<='0';
 ..                                                 pzapr<='1';
 ..                                                 state_next<=WAIT_TASK;
 ..                                     END IF;
```

```
..
..                      WHEN WAIT_TASK =>
..                              IF task_ready='1' THEN
..                                      reg_task_id<=V_task_id;
..                                      pzapr<='0';
..                                      task_prinyal<='1';
..                                      state_next<=WORK_PROC;
..                              END IF;
..
..                      WHEN WORK_PROC =>
..                              task_prinyal<='0';
..                              IF cnt="0100000" THEN
..                                      cnt<=x"00";
..                                      PROC_WORK<='0';
..                                      state_next<=INIT;
..                              ELSE
..                                      cnt<=cnt+'1';
..                                      PROC_WORK<='1';
..                              END IF;
..              END CASE;
END PROCESS;
END unit_processor_arch;
```

The general view of the structural diagram, the modelling of which was carried out, is shown below in Figure 3.

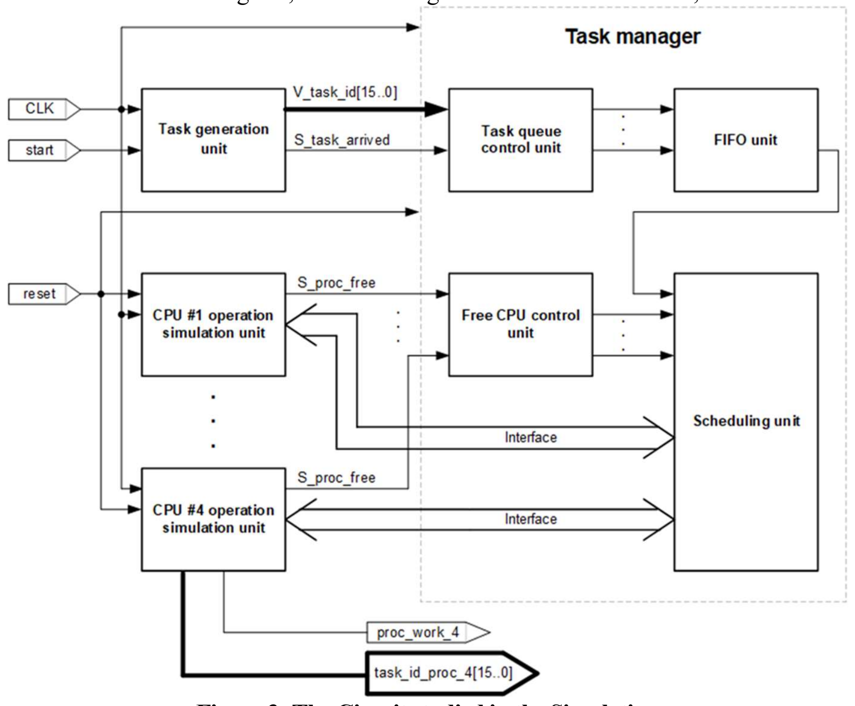

Figure 3: The Circuit studied in the Simulation

As can be seen from Figure 3, almost all of the previously described signals for the reconfigurable computing system are internal and are generated independently. It is only necessary to send to the system input the *CLK* signal, from which all units are clocked. In addition, we need to perform an initial system *reset*, and *start* the task generation unit.
To check the simulation results, the *proc_work_ [j]* outputs were used, which inform that the j-th CPU has received the task ID and started processing it, and also the bus *task_id_proc_ [j] [15 ... 0]* was used: it transmits information about the received j-th CPU for the task ID. During the simulation, the data bus of the FIFO unit *task_id_in_fifo [15 ... 0]* was registered to control the task IDs entering the system. The simulation results are shown in Figure 4.

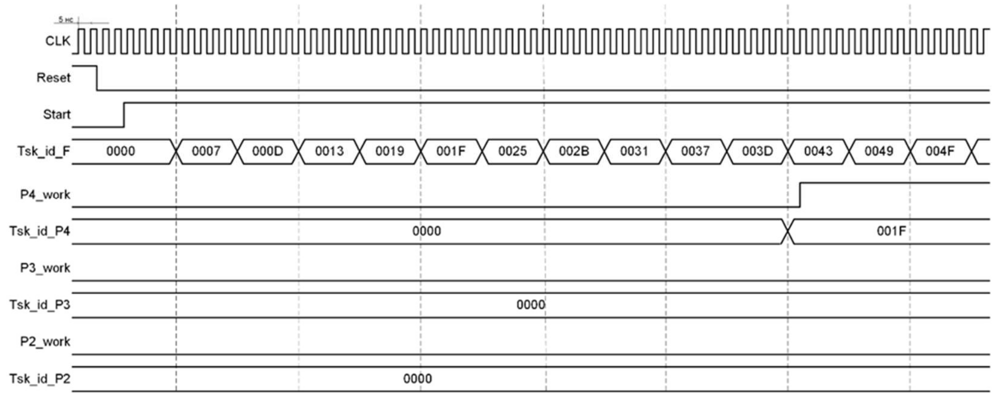

Figure 4: Simulation Results for a Reconfigurable Computing System with a Hardware Time-Sharing Task Manager

From the diagram shown in the figure, we can see how the ID of the first received task (*"001F" hex*) entered the CPU at number 4 after 10 clock cycles, after which the CPU sent a signal that it was busy with servicing. Since the fourth CPU was busy, the second task identifier (*"0025" hex*) arrived the CPU number 3 after 10 clock cycles. This CPU, similar to the previous one, set a signal that it was busy and started servicing the second task. Obviously, those 10 clock cycles that precede the assignment of a CPU task are spent on the work of the task manager itself. As can be seen from the simulation results, the fourth CPU is assigned to serve the current task first, followed by the third CPU, etc.
It is worth noting that tasks enter the reconfigurable computing system faster than the task manager has time to assign CPUs for them; therefore, they are accumulated in the FIFO unit, forming a queue. When the FIFO unit is full, new task IDs will not be registered in the system.
Analysis of the timing diagrams of the reconfigurable computing system that uses a time-sharing task manager revealed a flaw in this organization, which reduces the performance of the entire reconfigurable computing system. Its essence lays in the organization and principles of operation of the task manager itself, namely, in the fact that only this device has the right to interact with the task queue, which takes time. In addition, a CPU needs to interact with the task manager to receive a new task, which takes time again. As a result, a situation arises: for example, 5 CPUs become free at once into a reconfigurable computing system and there are many waiting tasks in a queue. The task manager begins to sequentially interact with each CPU, assigning a task to it, while it interacts with the queue. If we conventionally designate the time that the task manager spends on interacting with one CPU as 10 clock cycles, then the first CPU will start working in 10 clock cycles, the second in 20, and the fifth in 50. By that time, there is a high probability that more CPU would become free, and they will also have to wait for the task manager to start serving them. So, a situation arises in which pending tasks are not processed in the reconfigurable computing system with free CPUs. The way out of this situation can be a task manager [] different from the organization described above, when each CPU has its own queue with which it interacts, retrieving the tasks awaiting service. Also, a task manager interacts with each of the queues, defining the queue with the smallest number of tasks and placing into it a new task having entered the reconfigurable computing system, with ensuring an even distribution of tasks across all queues of this system. In this case, the task manager does not interact with the CPU, but only works with the task queues. Accordingly, the CPUs are no longer tied to the task manager and, as soon as they become free, they immediately start fetching a new task, each from its own queue. In this case, tasks arrive at CPUs in parallel, which saves time and increases system performance. The hardware redundancy associated with increasing the number of FIFO units to the number of CPUs is compensated by the fact that the volume of each of them can be reduced as many times as there are CPUs in the system. Thus, the total amount of memory used for storing task IDs is the same in the first and second cases. It is also worth noting that the algorithm of task manager work with distributed queues is much simpler than the first variant of the organization, since the interface for interacting with CPUs in the task manager is virtually absent [11, 12].

Description of the Hardware Scheduling Unit with Distributed Task Queues

In general terms, the diagram of a task manager with distributed task queues looks like that shown in Figure 5. In order to make the presentation more understandable, the diagram shows only two queues out of four available and executed as FIFO memory and having the same organization and scheme of enabling in the task manager as it made in [13-16] and [17-20].

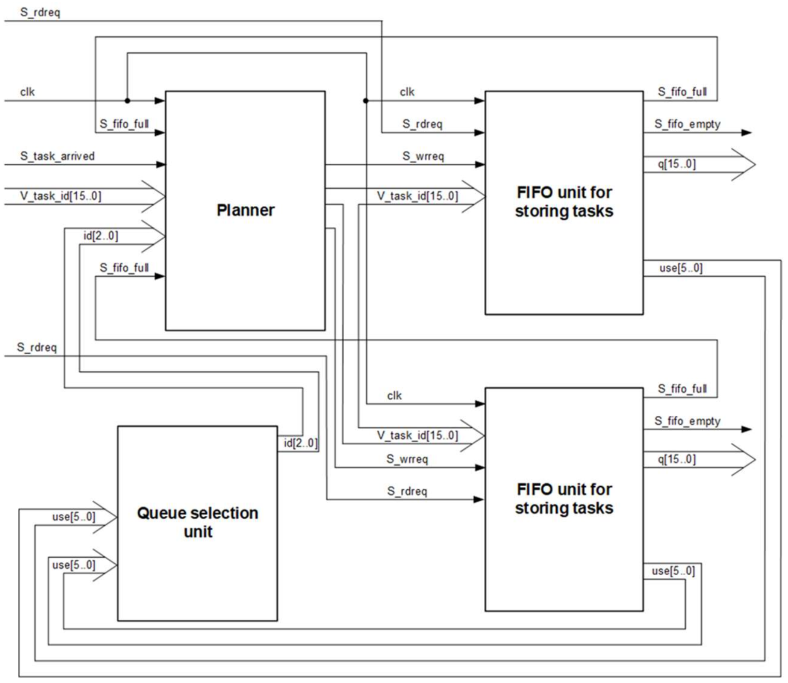

Figure 5: Structural Organization of a Hardware Space-Division Task Manager

FIFO units for storing tasks are no different from those considered for the time-sharing task manager. The main difference is that they have fewer storage cells. Their enabling is also organized differently. Only the task manager can write to the FIFO, and only the CPU can read data, so the write signal *S_wrreq* is sent from the scheduling unit separately for each FIFO unit. The input data bus *V_task_id [15 ... 0]* is common to all FIFO units and also outputs from the scheduling unit. The signal to read from the FIFO unit *S_rdreq* comes from the CPUs, each of which controls only its own queue. Also, each CPU constantly receives a signal from its queue about whether there are tasks in it. If there are tasks in the FIFO unit, the CPU reads them, if not, it goes into sleep mode. The output data bus of each FIFO unit goes to its own CPU. The *S_fifo_full* signal is used by the planner. It indicates that the FIFO unit is full. In this scenario, the task manager stops writing new tasks to this queue.
A queue selection unit (QSU) is designed to determine the FIFO units with the least number of entries. The new ID of the task received by the task manager will be assigned exactly to the FIFO unit indicated by the queue selection unit. The output bus *id [1 ... 0]* of the queue selection unit transfers to the planner the FIFO unit number with the smallest number of entries. The bus is 2-bit, which allows addressing to it four different FIFO units. The selection of the required queue is carried out on the basis of special FIFO outputs *use [5..0]*, which are fed from each memory unit to the selection unit. The bus *use [5..0]* constantly has the value of the number of recorded memory cells; in fact, this value reflects how full the queue is. It is easily formed inside a FIFO unit as the difference between the record counter and the read counter. The queue selection unit passes through its combinational comparison circuit the values from all buses *use [5..0]* to each of the FIFO units and selects the smallest value. Based on this, it is concluded in which queue the new ID should be written. So, the planner always knows exactly where a new task should be written, while it does not itself determine the required queue, which significantly saves the task manager service time and increases the performance of the reconfigurable computing system.
New tasks arrive in the planner, and the incoming IDs are immediately overwritten in the least filled FIFO unit. It controls *S_wrreq* signals of each of the FIFO units, as well as the FIFO input data bus. Also, the following function should be assigned to this unit: monitoring the status of all CPUs. If a CPU fails, then writing new information to its queue should be stopped, even if this queue is considered minimal and the queue selection unit is preferred. If the performance of the CPUs is not controlled, and, for example, the malfunction of one of them was not identified in time, then the queue of this CPU can accumulate a sufficiently large number of tasks, which will not be processed. In any case, it turns out to be necessary to provide for the possibility of switching queues between CPUs in order to ensure that all incoming tasks are served. This is a rather complex mechanism, the need for which did not arise when organizing a task manager with a single task queue. However, we can omit this aspect as a first approximation and not consider the issues of system reliability paying attention only to their functioning and the methods of distributing tasks entering the system.

Device Simulation and Comparison of Results

To simulate the work of the task manager with distributed queues, the following scheme was applied (Figure 6).

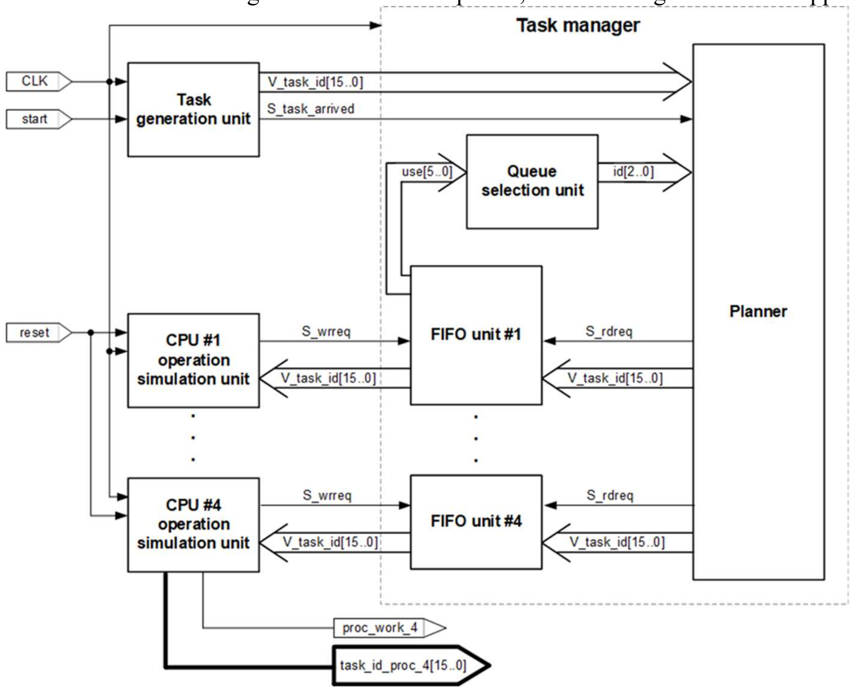

Figure 6: General View of the Structural Diagram Investigated during Modelling.

To compare the operation results for both variants of the task manager implementation in a reconfigurable computing system, the simulation modelling used the same units for simulating the CPU operation, and the same unit for generating tasks as it was in the first case. Thus, the conditions for modelling the system remained the same: tasks enter the reconfigurable computing system with a frequency of 5 cycles; any CPU serves one task for 32 cycles. The timing diagrams obtained as a result of the simulation are shown in Figure 7.

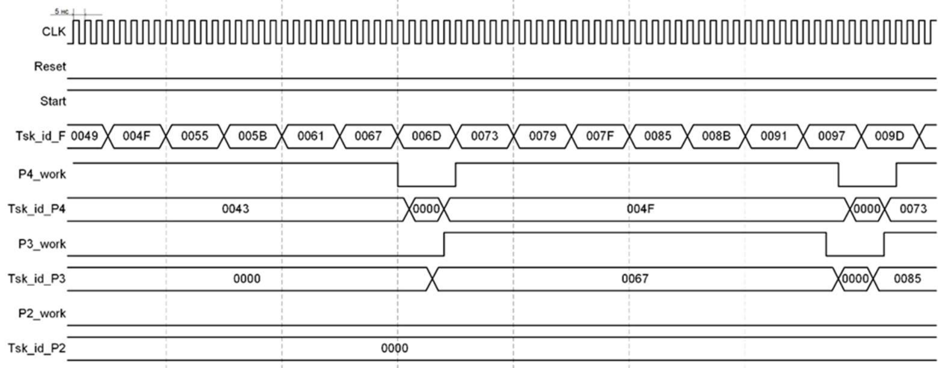

Figure 7: Simulation Results for the Reconfigurable Computing System with a Hardware Space-Division Task Manager

The main difference in the results is that in the first option of organizing the task manager, tasks accumulated in the FIFO and gradually filled it. As a result, the moment came when new IDs were not accepted in the reconfigurable computing system and it became overloaded. In this case, the processing time for a newly arrived task increased greatly. At the same time, free CPUs were constantly present in the system, but the resulting conflict did not allow them to receive tasks. In this case, it is obvious that an increase in the number of CPUs will not affect the performance in any way and the reconfigurable computing system will continue to overload.

The task manager with distributed queues demonstrates the result better than with a single queue during simulation. The reconfigurable computing system has time to receive and assign all incoming tasks. It does not become overloaded at any time and it is also capable of handling a more intensive flow of tasks. The reason for the increase in performance is simple: in the second case, the task manager works many times faster, while performing all the functions assigned to it. It is much simpler to organize and it does not require a CPU to have a dedicated interface to interact with the task manager.

Conclusion

Thus, we can conclude that the principles of device organization directly affect its performance. It is the organization of resources, not their quantity that underlies the increase in the speed of work. For comparison: both task managers are implemented as a project for a FPLD, while the microcircuit resources that would have been spent in either case are almost the same.

Acknowledgements
The paper is published with the support of the scholarship of the President of the Russian Federation for young scientists and graduate students for 2018-2020 (SP-68.2018.5).

References

1. Martyshkin A.I. Mathematical modelling and hardware support methods for control algorithms used for interacting processes in high-performance computing systems // Problems of theoretical cybernetics. Materials. Edited by Yu. I. Zhuravlev. 2017. P. 157-161.
2. Martyshkin A.I. Hardware implementation of algorithms for planning and scheduling tasks of a multiprocessor reconfigurable computing system // Modern information technologies. 2019. No. 29. P. 23-28.
3. Biktashev R.A., Martyshkin A.I. Modelling of task managers for multiprocessor systems // Advances of modern natural science. 2012. No. 6. P. 83-85.
4. Salnikov I.I., Martens-Atyushev D.S. Investigation of the option for implementing a planning subsystem and assigning tasks for a reconfigurable computing system used for digital signal processing // Models, systems, networks in economics, technology, nature and society. 2016. No. 2 (18). P. 261-267.
5. Martens-Atyushev D.S., Martyshkin A.I. Development and research of a reconfigurable system for digital signal processing // International student scientific bulletin. 2016. No. 3-1. P. 86-88.
6. Martens-Atyushev D.S. Development and research of a subsystem for task dispatching in a reconfigurable computing system for digital signal processing // Information technologies in economic and technical problems: collection of scientific papers of the International Scientific and Practical Conference. 2016. P. 247-250.
7. Vashkevich N.P., Biktashev R.A., Merkuriev A.I. Hardware support for a task manager with a global queue in multiprocessor systems. Proceedings of higher educational institutions. Volga region. Technical science. 2011. No. 3 (19). P. 3-14.
8. Biktashev R.A. Task manager for a multiprocessor system with hardware support // XXI century: results of the past and problems of the present plus. 2011. No. 3 (03). P. 111-115.
9. Martyshkin A.I., Martens-Atyushev D.S. Development of a subsystem for planning and assigning tasks of a reconfigurable computing system for digital signal processing // Modern methods and means of processing space-time signals: collection of papers of the XIV All-Russian Scientific and Technical Conference. Edited by I.I. Salnikov. 2016. P. 115-119.
10. Martyshkin A.I. Analysis of the hardware planner application area and assignment of tasks // Modern methods and means of processing space-time signals: Collection of papers of the XVII All-Russian scientific and technical conference. Edited by I.I. Salnikov. 2019. P. 80-83.
11. Martens-Atyushev D.S., Martyshkin A.I. Development and research of a reconfigurable computing cluster for digital signal processing // Modern information technologies. 2015. No. 21. P. 190-195.
12. Martyshkin A.I., Martens-Atyushev D.S. Investigation of nodes belonging to a reconfigurable computing system using hardware means // Modern methods and means of processing space-time signals: Collection of papers of the XVI All-Russian scientific and technical conference. Edited by I.I. Salnikov. 2018. P. 91-95.
13. Volchikhin V.I., Vashkevich N.P., Biktashev R.A. Task scheduler with hardware support for multiprocessor systems // News of higher educational institutions. Volga region. Technical science. 2012. No. 1 (21). P. 12-20.
14. Martyshkin A.I. Modelling of a reconfigurable system with a hardware task manager and a distributed queue among serving nodes // XXI century: results of the past and problems of the present plus. 2019.Vol. 8.No. 2 (46). P. 37-42.
15. Martyshkin A.I. Hardware implementation and verification of the algorithm for the functioning of a task manager in a multiprocessor reconfigurable computing system // Modern information technologies. 2019. No. 29, pp. 29-34.
16. Martyshkin A.I. An option of hardware support for algorithms used in planning and scheduling tasks of a multiprocessor reconfigurable computing system // Modern innovative technologies for training engineering personnel for the mining industry and transport. 2019. Vol. 1.No. 1 (6). P. 211-218.
17. Martyshkin, A.I. Mathematical modelling of Tasks Managers with the strategy of separation in space with a homogeneous and heterogeneous input flow and finite queue // ARPN Journal of Engineering and Applied Sciences, Volume 11, Issue 19, 1 October 2016, Pages 11325-11332.

18. Martyshkin Alexey, I., Martens-Atyushev Dmitry, S. Experimental study of a reconfigurable system with hardware task manager and a distributed queue // Journal of Computational and Theoretical Nanoscience, Volume 16, Issue 7, 2019, Pages 3040-3045.
19. Martyshkin, A.I., Martens-Atyushev, D. Mathematical modelling and evaluation of the characteristics of specialized reconfigurable systems based on a common bus at the stage of synthesis of the system configuration // Journal of Advanced Research in Dynamical and Control Systems, Volume 11, Issue 8 Special Issue, 2019, Pages 2852-2860.
20. Martyshkin, Alexey I. Study of Distributed Task Manager Mathematical Models for Multiprocessor Systems Based on Open Networks of Mass Servicing // AD Alta-Journal of Interdisciplinary Research, Volume 8, Issue 1, Special Issue 3, 2018, Pages 309-314.