Implementation of a RTOS service for the management of the reconfiguration

Dan Hotoleanu

Implementation of a RTOS service for the management of the reconfiguration

Dan Octavian Hoțoleanu
Supervisor : Jean-Christophe Prevotet
Laboratory : IETR
Team : CPR

Outline

1. Introduction

In the systems that we use today there is a tendency to increase the complexity, the performance and the also the flexibility of digital systems. Also in case of embedded systems that work in industry or on different types of vehicles there is a very strong demand of power reduction, cost and number of resources used. In order to cope with these demands more and more designers adopt the idea of heterogeneous software and hardware solutions. The software part from an embedded processor offers the flexibility needed but at the cost of smaller performance compared to the same task implemented in an ASIC (Application Specific Integrated Circuit). That is why along the embedded processor designers place a couple of accelerators that are circuits specialized in performing a determined task with a very high performance.

The aim of this project was to create a heterogeneous system on an FPGA (Field Programmable Gates Array) device with an embedded processor and dynamic partial reconfigurable tasks. The embedded processor can be a hard PowerPC 405 processor or a soft MicroBlaze processor. The use of dynamic partial reconfiguration creates a more flexible hardware design. This way the same area assigned to a hardware task with a specific hardware architecture can be assigned to another hardware task with a different architecture. The whole reconfiguration functionality will be integrated as a new kernel service for the operating system running on the embedded processor. The operating system running on the embedded processor is a real time operating system.

Throughout the embedded systems on FPGA literature the problem of hardware tasks is discussed and one of the frequent problems encountered is the one of placing the tasks in a manner in which the degree of occupation of the area is maximized. To this purpose, there are many algorithms each of them having a certain degree of FPGA area fragmentation. Another problem is that when using dynamic partial reconfiguration technique the time taken by the FPGA chip to reconfigure is between milliseconds and tens or hundreds of milliseconds. Dynamic partial reconfiguration is possible only in some of the Xilinx FPGAs families.

2. Basic elements

In the following there will be presented the basic elements of the project. These basic elements are the dynamic partial reconfiguration technique, the operating system uCOS-II and the FPGA. This project introduces the ability to dynamically reconfigure a part of a FPGA on which runs the operating system.

2.1. What is a FPGA?

The FPGA circuit is presented in figure 2.1. This figure presents a very general view of an FPGA from the Spartan 3E family of Xilinx company. The chip presents input and output buffers (IOBs) at all four edges of the chip. The FPGA chip is composed of combinational logic blocks (CLBs), which are represented in the picture as dashed lines. These blocks are composed of slices that can

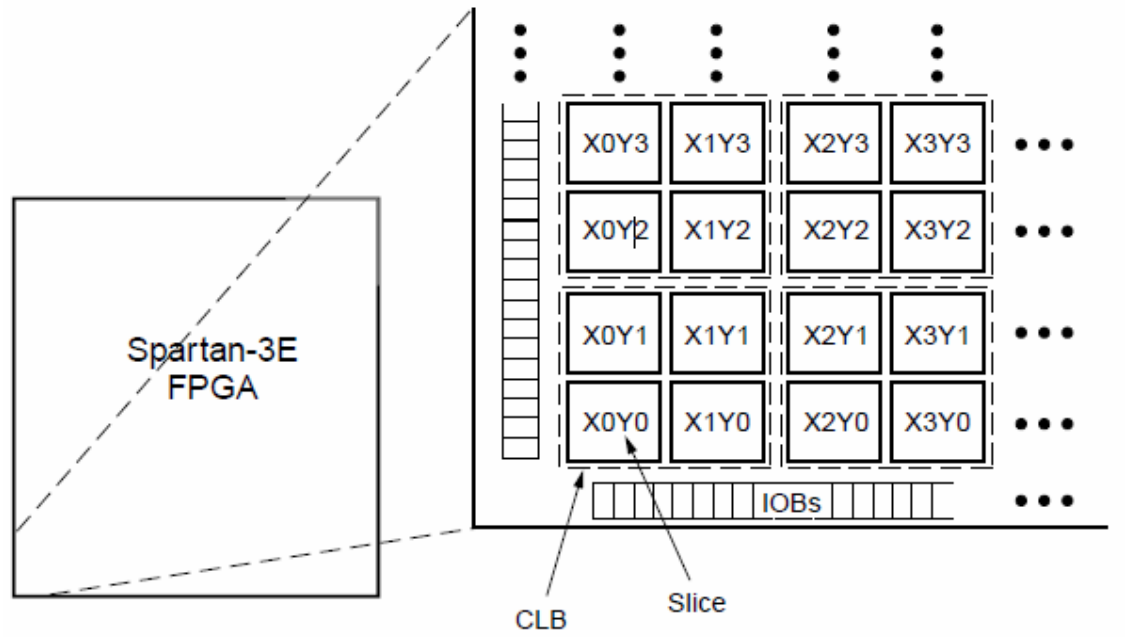

Figure 2.1. The Xilinx Spartan 3E FPGA from [1]

be of two types : SliceL (which means slices for logic) and SliceM (which means slices for memory and logic). Each slice has a position (like in a matrix) and contains one or more look-up tables (LUTs), carry logic and a memory element that can be configured to be triggered on the rising/falling edge of the clock signal (flip-flop) or when the clock signal is stable 0 or 1 (latch). In figure 2.2 is illustrated a part of a slice of type SliceL. In this picture on the left side it can be seen the look-up table with 4 entries (the address lines) and one output. Basically the LUT is a memory that stores one bit for each of its addresses and that can implement any Boolean function with the number of variables equal to the number of address lines. On the right side of the picture the memory element can be seen and in between the LUT and the memory element lies the carry logic and multiplexers. The CLBs are connected through a interconnection logic that contains conductive lines, long lines for connecting distant CLBs and short lines for connecting neighboring CLBs. The connection between two lines is made possible through programmable interconnection points (PIPs). The FPGA can contain additional resources like : BlockRams which are synchronous memories, Digital Signal Processing (DSP) blocks that contain dedicated multipliers and adders that can be used for digital signal processing tasks, embedded PowerPC processors. Each FPGA contains a configuration memory that keeps track of the configuration of the elements that form the FPGA.

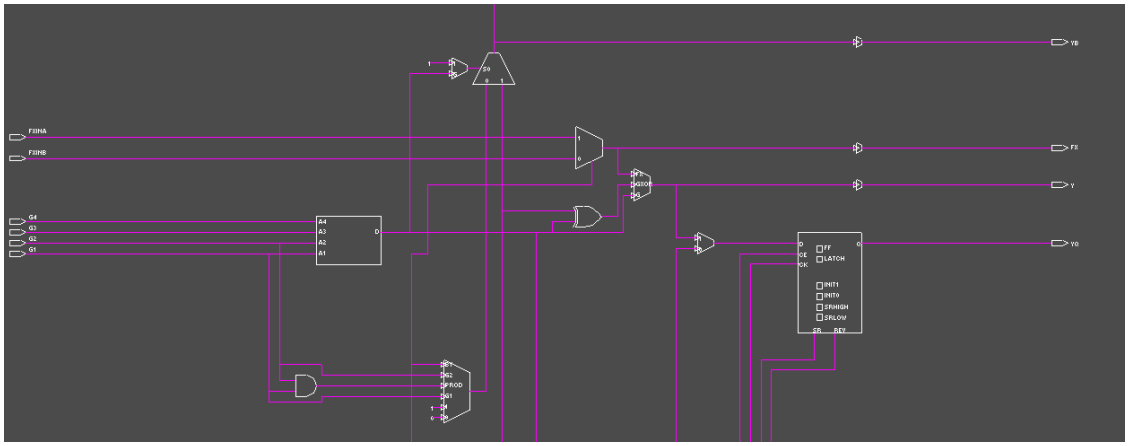

Figure 2.2. A part of a slice of type SLICEL

2.2. The dynamic partial reconfiguration technique

Dynamic partial reconfiguration is the property of a FPGA chip to modify a part of its configuration memory. This way by using the dynamic partial reconfiguration technique only a part of the architecture of a design that is loaded on the FPGA can be changed while the rest of the design still works normally. Dynamic partial reconfiguration can be only performed on Xilinx FPGAs. Dynamic partial reconfiguration technique is a technique that comes as an extension of the modular programming for FPGA chips. By using modular programming different programmers that were working on the same project could synthesize their designs separately from each other. After synthesizing one of the programmers would take the results of the synthesis and merge them together and generate the configuration file for the FPGA [1]. For implementing a system that would contain the technique of dynamic partial reconfiguration Xilinx company offers a guiding tutorial [2]. The steps that the programmer has to follow in the case of dynamic partial reconfiguration are complex and they impose a good understanding of the technique in order to solve the errors that could eventually appear during the design of the system.

Dynamic partial reconfiguration technique can be applied using two main methods : module based dynamic partial reconfiguration and difference based dynamic partial reconfiguration.

Module based dynamic partial reconfiguration means that the FPGA chip is split into two kinds of areas : fixed ones and reconfigurable ones. In the fixed regions the static logic is placed. This logic does not change throughout the reconfiguration process. The reconfigurable types of regions are the ones in which the reconfigurable modules are placed.

In figure 2.3 which is taken from [2] is presented an example of how to use dynamic partial reconfiguration for older versions of Xilinx software and boards. In this picture there are two reconfigurable regions placed in the center of the chip. These reconfigurable regions communicate with the other regions or between themselves by using bus macros. The areas use only the IOBs that are placed in those areas. The IOBs are used for input/output transfers.

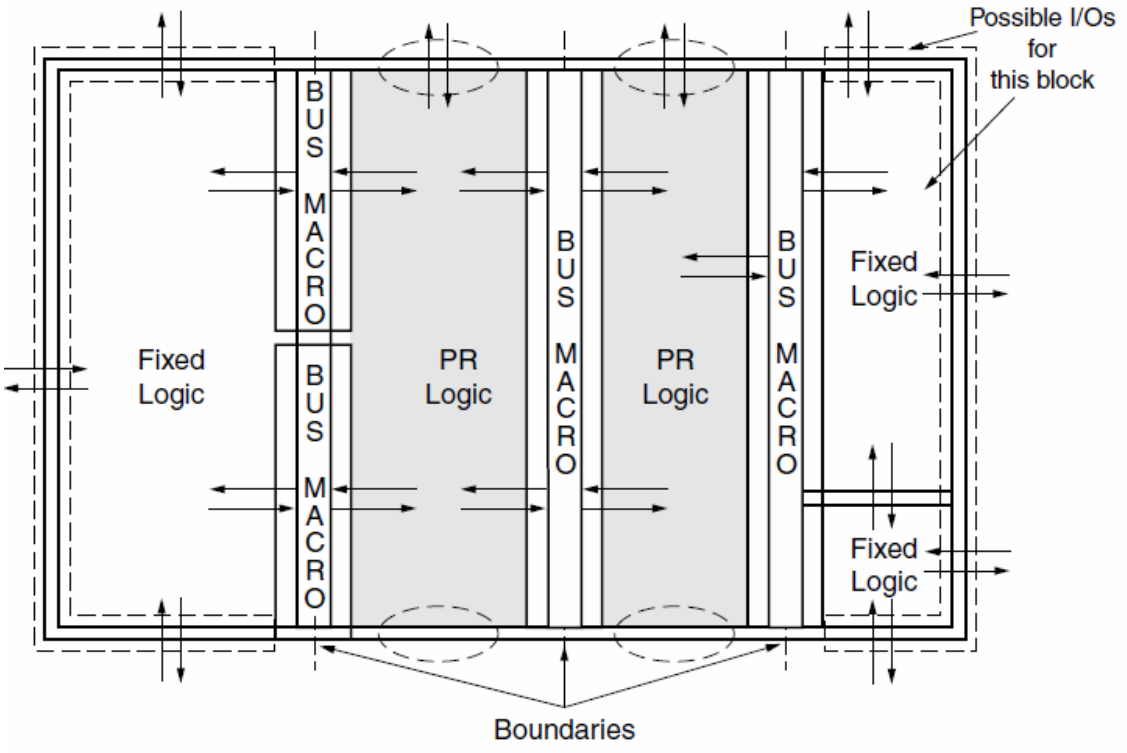

Figure 2.3. An example of a design that uses dynamic partial reconfiguration technique[2]

Xilinx tools recognize two modules that use the same reconfigurable region based on their entity names. Hence, the name of the entities of the reconfigurable modules that use the same reconfigurable region have to be identical. Also the names of the entities of reconfigurable modules have to be the same as the files that store their code description. The interfaces of the entities of the modules that are placed in the same reconfigurable region have to be the same.

The flow of implementing the dynamic partial reconfiguration technique involves respecting some rules. These rules are presented by Xilinx in [2]. The rules are:

1. The height of the reconfigurable region has to be the same as the height of the FPGA chip. This rule applies in the old FPGAs. In the new FPGAs like Virtex 4 and newer this rule does not apply anymore. In these chips the height of the reconfigurable region can be smaller than the height of the FPGA chip.
2. The width of the reconfigurable region has to be minimum 4 slices and maximum the width of the chip. The width of the reconfigurable region in the number of slices has to be divisible with 4.
3. On the horizontal axis the reconfigurable region has to be placed such that the left corner of the reconfigurable region to be placed on a slice that is multiple of 4. This way the left side of the reconfigurable region can be placed on slices 0,4,8 and so on.
4. All the resources that are placed in a reconfigurable region are considered to be configured by the bit file generated for that region. These resources can be

BlockRAM memories, DSP resources, T buffers, IOB components and routing resources.

5. Clock resources are decoupled from the reconfigurable region and special components are used to connect the clock signal to the reconfigurable region. These components are BUFGMUX and CLKIOB components.
6. The input/output buffer components that are placed above and below the reconfigurable region are considered as being a part of this region. This way these components will not be used directly by other modules that are placed in other parts of the chip. The only way to use the input/output buffers from a reconfigurable region from outside of this region is through the reconfigurable module that is placed in that region and through the bus macro components placed at the edge of that region. In the new FPGA chips in which the height of the reconfigurable region is variable these problems can be avoided.
7. If one reconfigurable module occupies the left or the right edge of the FPGA chip then the IOB components from that area cannot be used directly by other components from different areas of the chip. The only way in which these components can be used is through the reconfigurable module that is placed in that region and through the bus macros. This way is good to know that when designing a reconfigurable architecture is good to have in mind the complex aspects that come from placing the reconfigurable region on one of the edges of the chip.
8. In order to minimize the problems that could arise from using dynamic partial reconfiguration technique it is recommended that only one reconfigurable area to be used. The number of reconfigurable regions is constrained only by the fact that on the horizontal axis the reconfigurable region should be of at least 4 slices and on the vertical axis the reconfigurable region should be equal to the height of the chip or smaller if the chip allows that.
9. The area defined as being reconfigurable cannot be changed at runtime. This way when designing the reconfigurable architecture we have to evaluate the dimensions of each module and determine which is the biggest one. Based on determining which is the biggest module we can determine the dimensions of the reconfigurable region.
10. The reconfigurable modules communicate to other static or reconfigurable modules by using the bus macros. Besides the clock signal that is passed through special buffers and that can pass this way the reconfigurable region, no other signal can pass the boundary between a reconfigurable region and a static region or between two reconfigurable regions without passing through a bus macro.
11. The architecture should be designed in such a way that no static module should base its vital functionality on the functionality provided by the reconfigurable modules. There are some cases when there is the need to use handshake signals to determine when a reconfigurable module is ready to be used.
12. An advantage that is offered by dynamic partial reconfiguration is that it keeps the content of memory devices that are placed in the reconfigurable area throughout the reconfiguration. This is an advantage because the modules that

are placed in the reconfigurable area can use the data that was stored in the memory elements by modules that were previously loaded into the reconfiguration area.

To implement a project using the dynamic partial reconfiguration technique it is necessary to follow a flow of design that is taken from [2] and is presented in the following :

1. In the “Design entry” phase the HDL code is written.
2. In the “Initial budgeting” phase location and surface constraints are introduced using the PlanAhead tool from ISE package. Also in this phase the time constraints for the whole design and for the modules are introduced.
3. In the “Active implementation” phase the NGDBUILD, MAP and PAR tools are executed for each reconfigurable module and then for each configuration of the reconfigurable module.
4. In the “Assembly phase implementation” phase the module that will be first downloaded on the chip is assembled. Xilinx company recommends the creation of all the possible configurations with the reconfigurable and the fixed modules in order to test the functionality of the reconfigurable modules.
5. The design is verified by using static time analysis or simulation.
6. The FPGA Editor tool form the ISE package is used to determine if there are signals that pass through the boundary of two reconfigurable regions or a reconfigurable region and a static region and that they pass only through bus macros.
7. The bit file for the complete configuration is created and this bit file is the first file that is downloaded on the chip.
8. For each reconfigurable module there are created individual bit files.
9. The initial complete configuration is downloaded on the FPGA.
10. The FPGA is reconfigured based on the needs with the partial bit files created for each reconfigurable module.

Xilinx company suggests that for using dynamic partial reconfiguration technique a specific directory structure to be used. In figure 2.4 which is taken from [2] presents the directory structure proposed by Xilinx company for dynamic reconfiguration projects. In this structure there is a directory named “Hdl” where the HDL files of the project are stored. The “ISE” directory is used to synthesize each module. The “Modules” directory contains each module routed and placed along with its static part, independent of the other reconfigurable modules. The “Pims” directory contains the modules that are physically implemented. The “Top” directory contains two directories “Assemble” and “Initial”. The “Assemble” directory is used for the creation and assembly of the initial complete design. The “Initial” directory is used in the “Initial budgeting” phase in which the location, the surface and the time constraints are specified. This directory structure is recommended to be used when the dynamic partial reconfiguration design is implemented using the command line. In the command line there are introduced commands for mapping, placing and routing the design. If the visual mode of PlanAhead tool is used these commands are not introduced by the user, but the system knows how to deal with the partial reconfiguration. Also when using the PlanAhead tool we don’t have to worry anymore with the directory structure, because the tool knows how to create it.

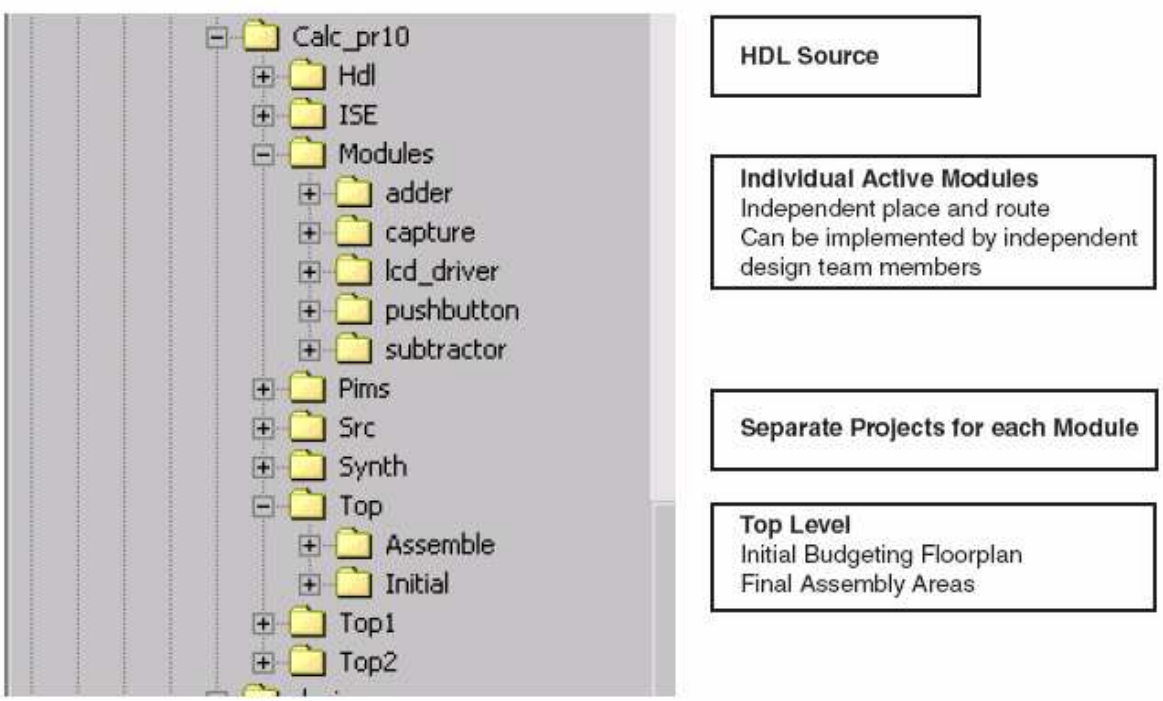

Figure 2.4. The directory structure proposed by Xilinx company for projects that use dynamic partial reconfiguration [2]

In order not to have problems when implementing the partial reconfigurable project the following advices that were taken from [2] should be considered :

1. The design has to be built in such a way that the top module should contain only black boxes of the modules that it contains. At this level the logic should be just for connecting the input/output pins of the chip, clock logic and the instantiation of the bus macros.
2. It must be verified that the signals passing through a reconfigurable region boundary pass only through bus macros. There are different types of bus macros that are used and that depend on the FPGA chip that is used. There are implementations of bus macros on 4 bits or on 8 bits. The macros are defined on the directions the signal propagate through them : this way there exists bus macros that propagate the signal from right to left or from left to right, or from top to bottom or from bottom to top.
3. If we have a design where the height of the reconfigurable module has to be the height of the FPGA chip and we have two static regions that have in between a reconfigurable region and if the two static regions have a signal through which they communicate, then the signal has to pass the boundary of the reconfigurable regions through a bus macro. The signal will not be available when the reconfigurable region is reconfigured.
4. Xilinx company recommends avoiding to use modules that need the use of “Clock template”. Also Xilinx company recommends avoiding the use of DCM modules in the dynamic partial reconfigurable designs.
5. It is very important that the clock resources to use only global resources that are declared in the top file. It is forbidden to use clock resources in the

reconfigurable regions. It is very important to notice that for some reconfigurable modules the synthesis tool inserts clock buffers and because this is a forbidden action it should be specified in the properties of the synthesis tool that it should not introduce clock buffers for those modules.

6. The synthesis of the top module is made with the synthesis option to insert the input/output buffers kept.
7. The synthesis of the modules is made with the synthesis option to insert the input/output buffers not kept.

Bus macros are used to define fixed routing regions through which the signals can pass from and to a reconfigurable region. The placement of bus macros is constrained through location constraints. Current implementations of bus macros use tri state buffers that enable the sending of data bits between two modules on the long lines. There are Xilinx FPGA chips that do not have tri state buffers. On those FPGAs there is no possibility to use module based dynamic partial reconfiguration using bus macros.

In figure 2.5 is presented the way a bus macro is implemented. The figure is copied from [2]. In this picture the tri state buffers are connected to the bus that is implemented using long lines. The LT and RT signals are used to select the direction in which the data is sent : from left to right or from right to left. Data are sent on the LI and RI lines. Generally bus macros are unidirectional, so either we will have selection and data transmission signals for the left side, or for the right side.

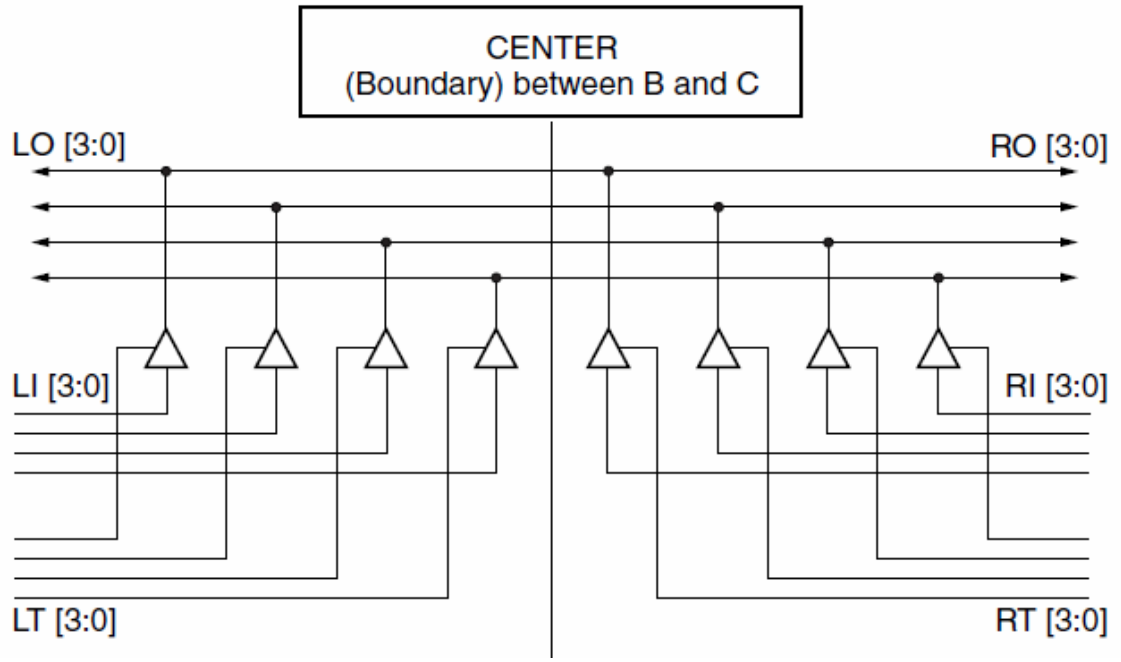

Figure 2.5. The implementation of a bus macro [2]

Dynamic partial reconfiguration using the difference method is described in [3] and it is a practical solution only for designs where the changes made by dynamic partial reconfiguration are small. It becomes a less practical solution when the changes are large (modules). This method is simpler than the one based on modules and implies minor modifications in the generated files after the process of compiling and implementing a design.

With the introduction of version 12 of Xilinx ISE development tool the flow of module based dynamic partial reconfiguration was changed and simplified. The new flow is called partition based partial reconfiguration. The flow is based on partitions. A partition enables the reuse of previous implemented result for a specific module. This way if the implementation of a module respects the timing constraints and the area constraints then we can keep the implementation of that module and make different configurations by modifying the rest of the modules[4]. In this version of dynamic partial reconfiguration technique the bus macros were replaced by partition pins that are flip flops from the boundary of reconfigurable regions. These flip flops ensure the communication between reconfigurable regions and the rest of the design. Another new feature that was introduced in this flow is that partition pins do not need to be constrained as for bus macros. In figure 2.6 it is presented the implementation of a reconfigurable module and in red is highlighted the partition pins that are placed on the left and the right sides of the partial reconfiguration area.

The reconfiguration can be performed from a PC by sending configuration files to the FPGA in order to configure the partial reconfigurable region or it can be done by a processor. If the reconfiguration is done by a processor without human intervention then the method is called dynamic partial self reconfiguration. The processor can be located on the FPGA chip (PowerPC or MicroBlaze) or outside the FPGA chip. This way of reconfiguration needs a memory in which the partial configuration files are stored.

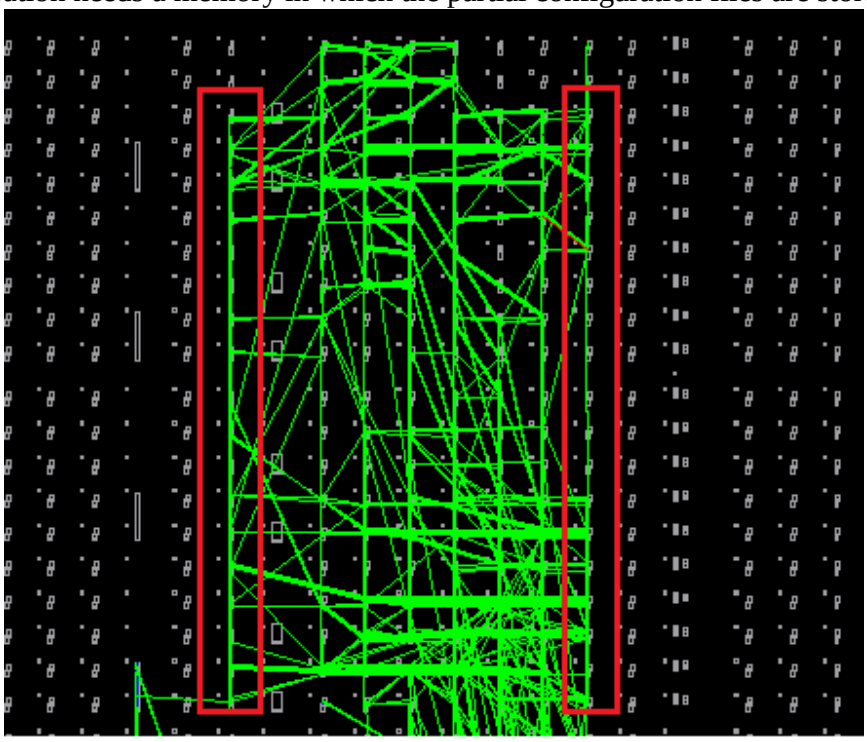

Figure 2.6. The implementation of a reconfigurable module and the partition pins

Hence, by accessing this memory the processor can read the partial configuration file and reconfigure a part of the FPGA.

2.3. Aspects of the real time operating system

The uCOS-II operating system is a real time operating system. A real-time operating system (RTOS) is an operating system that guarantees a certain capability in a specified time. For example, an operating system might be designed to ensure that a certain object was available for a robot on an assembly line. In a hard real-time operating system if the calculation could not be done in a specified amount of time in order too obtain that object, the operating system would terminate with a failure. Hence in this type of RTOS there is a guarantee that a specific task finishes in a determined amount of time. In a soft real-time operating system, the assembly line would continue to function if the specified time is not met but the throughput of the system might decrease, causing the robot to be temporarily unproductive. Some real-time operating systems are created for a special application and others are more general purpose.

In the following I will describe some notions of real time operating systems. Figure 2.1 presents the foreground/background systems. The foreground/background systems are of low complexity and are in general designed as presented in the picture. In these systems there is an infinite loop that calls modules to perform the desired operation. In the foreground there are asynchronous events that are handled by interrupt service routines ISRs. The foreground is also called interrupt level and background is called task level. ISRs have a tendency to take longer than they should. Until the background routine gets to execute, information for a background module that an ISR makes available

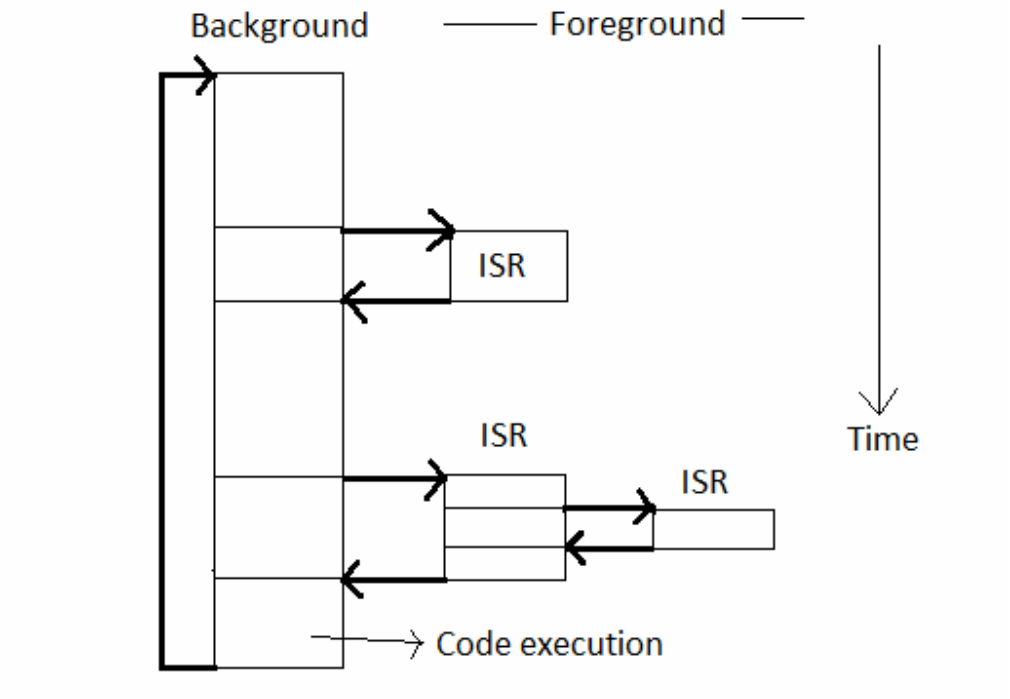

Figure 2.7. Foreground/background systems [18]

is not processed. The worst case task-level response time can be determined depending on how long the background loop takes to execute. The time for successive passes through a portion of the loop is nondeterministic because the execution time of typical

code is not constant. If a code change is made the timing of the loop is affected. Most high-volume microcontroller-based applications are designed as foreground/background systems. From the power consumption point of view in microcontroller based applications it might be better to halt the processor and perform all the processing in ISRs.

The critical sections of code are sections of code that have to be treated indivisibly. So when executing the code in these sections the processor should not be interrupted. That is why when entering a critical section the interrupts are disabled and enabled only when exiting the critical section.

The operating system might contain shared resources. These resources are resources that are used by more than one thread. For gaining control of a shared resource by one task the operating system should check if there is another task that is accessing that resource. If there is then the access should not be granted. If there is no task waiting for the resource than the access should be granted. The process of granting access to only one task is called mutual exclusion.

Multitasking is the ability of the operating system to have more tasks running in almost the same time on the processor. The task are run such that only one task is in the processor at a moment in time. But the task that is in the processor doesn't stay there indefinitely, it stays until some condition is not met anymore (its priority is smaller than of another task). Multitasking is like foreground/background with multiple backgrounds. Multitasking provides for modular construction of applications and maximizes the use of the CPU.

A program is composed of tasks that run on a CPU thinking they have the CPU all to themselves. When designing an application it should be known that the work should be split between the tasks. For a task it must be assigned the stack area, the CPU registers, the priority. Each task is basically an infinite loop that can be in one of the five states : dormant, ready, running, waiting, ISR. The dormant state is considered for a task that resides in memory and that it is not made available for the multitasking kernel. If the priority of the task is smaller than the currently working task then the task it is in ready state. A task is said to be running when it has the control of the CPU. When a task is waiting for the occurrence of an event it is said to be waiting. If an interrupt has occurred and the task is in the state of servicing the interrupt then the task is in state ISR.

The context switch is the mechanism of removing a task from the CPU and placing it in the current task's context in the stack. After performing this operation the new task's context is loaded into the processor and its execution is resumed. One of the drawbacks of context switching is that it adds overhead to the execution of the application. The number of registers determines the overhead of the context switch because they have to be stored on the stack when the context switch is done.

The part that is responsible for the management of tasks and for the communication between tasks is the kernel. Context switching is the fundamental service provided by the kernel. Because the services provided by the kernel require execution time, the kernel adds overhead to the system. Depending on how often these services are invoked the amount of overhead can be determined. The kernel is code that is added to the software that is executed on the processor. This means that the kernel uses ROM memory to store its code and RAM memory to store its data structures. Also each task has a stack which is introduced in the RAM memory. Single-chip microcontrollers are

not able to run a kernel because they have a very small RAM. The kernel provides services that are important to an application. These services are semaphore management, mailboxes, queues, time delays.

The scheduler is the part of the operating system that is responsible of determining which task to run next. Because most real time kernels are priority based, each task receives a priority number based on its importance. In a priority based operating system the task that has the highest priority and that is ready is given the CPU. Depending of the type of kernel it can be determined when the task takes the CPU. There are two types of kernels : preemptive and non-preemptive kernels.

Non-preemptive kernels mean that the task is doing something to explicitly give up the control of the CPU. This type of kernel is called cooperative multitasking because tasks cooperate with each other to share the CPU. The ISR are treated asynchronously. When an ISR appears then the current task is suspended and the ISR is executed. After finishing the ISR returns to the task that was interrupted. If a task with a higher priority appears then it will have to wait for the CPU until the task that is running on the CPU decides to release the CPU. The interrupt latency for a non-preemptive kernel is typically low. Task response time for a non-preemptive kernel may be longer than in the case of foreground/background systems because of the time the longest task needs to execute. An advantage of the non-preemptive kernel is that it does not need to enforce a strong security mechanism for the shared data, because each task holds the CPU for itself and we do not need to fear of it to be preempted. There are still some cases where semaphores should be used. An example of where semaphores should still be used is if the task needs exclusive access to an I/O device like a printer. In figure 2.8 it is presented the way a non-preemptive kernel works. The figure is described in the following : in state

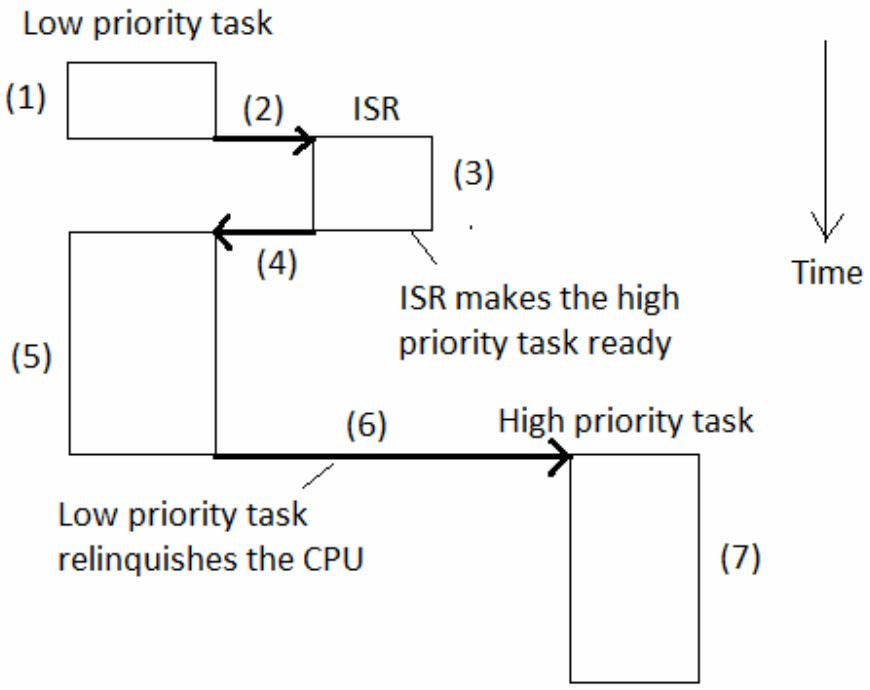

Figure 2.8. The way a non-preemptive kernel works [18]

(1) a task that executes is interrupted, in state (2) the CPU jumps to the ISR if the interrupts are enabled, in state (3) ISR makes a higher priority task ready to run and handles the event, in state (4) the ISR completes the execution of its code and a return from interrupt instruction is executed and then the CPU returns to the interrupted task, state (5) shows that the task code resumes after the interrupted instruction, (6) when the task finishes its execution it calls a service in the kernel to allocate the processor to another task, state (7) the scheduler sees this higher priority task that was made of high priority because of the interrupt execution and runs the code of it to treat the event of the interrupt. Responsiveness is the most important drawback of a non-preemptive kernel. A problem is that a higher priority task that was made ready to run has to wait until the task that runs on the processor finished its execution. So task level response time for a non-preemptive kernel is nondeterministic. So because it is not possible to determine when the highest priority task will obtain the CPU, in the application that is written and that is above this kernel the programmer should know when to give control of the CPU to the high priority task.

The preemptive kernel is a kernel in which the most high priority task is given control of the CPU. This way a preemptive kernel is used when system responsiveness is important. In case there is a task that makes the priority of another task to be the highest priority, immediately the task is preempted and the task with high priority is placed on the CPU. When an interrupt service routine makes a higher priority task ready, when the interrupts service routine finishes, the interrupted task is suspended and the new high priority task is resumed. In figure 2.9 is presented the way a preemptive kernel works. Its

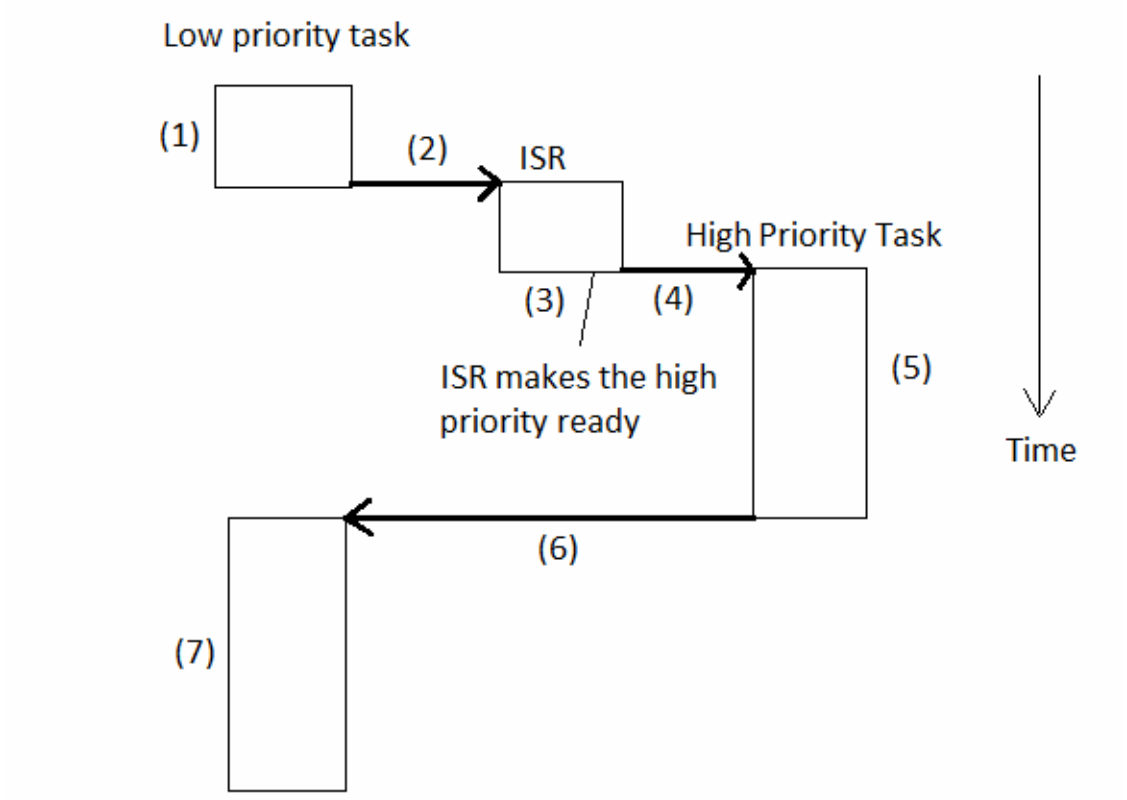

Figure 2.9. The way a preemptive kernel works [18]

functioning is described as follows : in state (1) we have a task that executes on the CPU and that is interrupted at a point in time. In state (2) the CPU jumps to the ISR if the interrupts are enabled. In state (3) the ISR makes a higher priority task ready to run and handles the event, also when the ISR completes a service provided by the kernel is invoked. In state (4) the high priority task is executed. In state (5) because a more important task has been made ready to run instead of returning to the interrupted task, the kernel performs the context switch and gives to the higher priority task control of the CPU. When the task finishes its execution a function from the kernel is called to put the task to sleep. In state (6) the CPU is freed from the higher priority task. In state (7) the kernel observes that there is a task with lower priority that was interrupted and that can be executed now due to the fact that its priority is now the highest one. By using a preemptive kernel we can say when a task is going to be executed and so the task-level response time is minimized by using this kind of kernel.

2.4. The uC/OS-II real time operating system

UC/OS-II is a highly portable, ROMable, very scalable, preemptive real-time, deterministic, multitasking kernel. It can manage up to 64 tasks (56 user tasks available). It has connectivity with μC/GUI and μC/FS (GUI and File Systems for μC/OS-II). It is ported to more than 100 microprocessors. It has ports for most popular processors and boards in the market and is suitable for use in safety critical embedded systems such as aviation, medical systems and nuclear installations. A few ports are for microprocessors and microcontrollers from the following companies Altera, Atmel, Freescale, Motorola, Fujitsu, Intel, IBM, Microchip, Xilinx and other companies[5]. It is simple to use and simple to implement but very effective compared to the price/performance ratio. It supports all type of processors from 8-bit to 64-bit. UC/OS-II can manage up to 64 tasks. The four highest priority tasks and the four lowest priority tasks are used by the uC/OS-II. This leaves 56 tasks available for the application. Each task is assigned a priority. The lower the value of the priority, the higher the priority of the task. The task priority number serves also as task identifier.

The operating system uses the rate monotonic scheduler. In this type of scheduling the tasks with the highest rate of executing are given the highest priority. There are some assumptions that are made for this type of scheduler, this assumptions are : all tasks are periodic, tasks do not synchronize with one another, preemptive scheduling is used (always runs the highest priority task that is ready). Under these assumptions, let n be the number of tasks, E_i be the execution time of task i, and T_i be the period of task i. Then, all deadlines will be met if the following inequality is satisfied : $\Sigma \frac{E_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$.

In figure 2.10 it is presented the process cycle in uC/OS-II. After a new task is created it

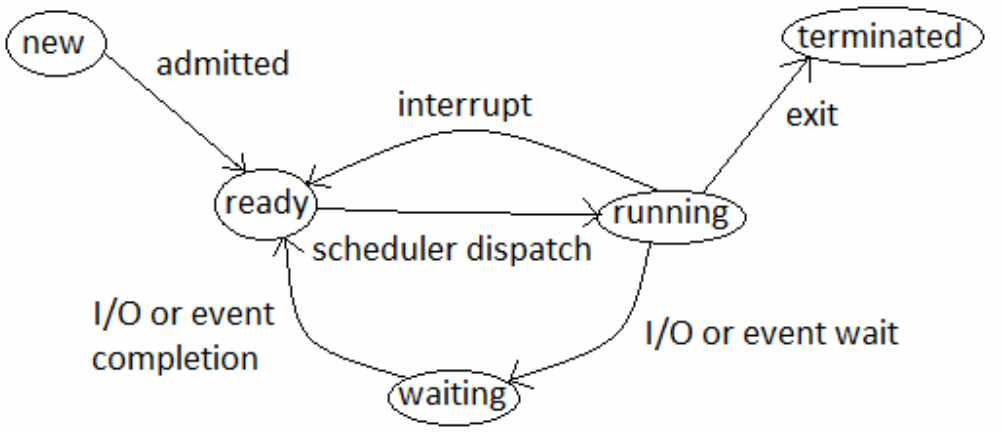

Figure 2.10. Process cycle in uC/OS-II

is admitted in the ready state. In this state the task is ready to be executed. In order to be executed the task passes from the state ready to the state running (scheduler dispatch). If the task is running and there is an interrupt occurring then the task goes from the state running to the state ready. If the task is in the state running and it is waiting for an I/O or an event the process goes from state running to state waiting. In state running if the process exits this state it goes to state terminated. In state waiting the state is waiting for the completion of I/O or the event that it is waiting for. From this state after the completion of the I/O or the event the process goes from state waiting to state ready.

The creation of the tasks is done through the functions OSTaskCreate() and OSTaskCreateExt(). In figure 2.11 it is presented the states of the tasks. If the task is in

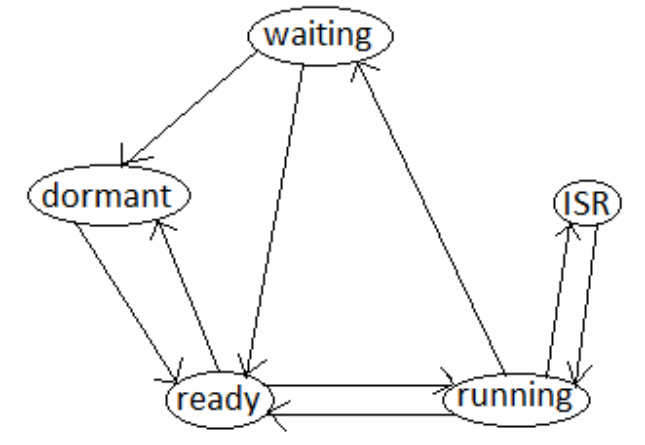

Figure 2.11. The states of the tasks

waiting state and it is deleted it goes in the dormant state. From the waiting state the task can go to ready state if it is ready to be executed. From the ready state the task can move to the dormant state or it can go to running state in which it is run. From the running state the task can go to ISR if it appears an interrupt. In ISR the task can go back to running if the task recovers from the interrupt.

When a task is created the task has to get a stack in which it will store its data. A stack must consist in contiguous memory locations. It is necessary to determine how

much stack space a task actually uses. Deleting a task means the task will be returned to its dormant state and does not mean that the code for the task will be deleted. The calling task can delete itself. If another task tries to delete the current task, the resources are not freed and thus are lost. So the task has to delete itself after it uses its resources. The priority of the calling task or another task can be changed at run time. A task can suspend itself or another task. A suspended task can resume itself. A task can obtain information about itself or other tasks. This information can be used to know what the task is doing at a particular time.

The memory management includes : initializing the memory manager, creating a memory partition, obtaining status of a memory partition, obtaining a memory block, returning a memory block, waiting for memory blocks from a memory partition. Each memory partition consists of several fixed size memory blocks. A task obtains memory blocks from the memory partition. A task must create a memory partition before it can be used. Allocation and de-allocation of these fixed-sized memory blocks is done in constant time and is deterministic. Multiple memory partitions can exist, so a task can obtain memory blocks of different sizes. A specific memory block should be returned to its memory partition from which it came.

The uC/OS-II operating system uses a timer to generate the events of rescheduling and context switching. For this the uC/OS-II defines a clock. The clock has a clock tick. A clock tick is a periodic time source to keep track of time delays and time outs. Tick intervals is between 10 and 100 ms. The faster the tick rate, the higher the overhead imposed on the system. Whenever a clock tick occurs uC/OS-II increments a 32-bit counter. A task can be delayed and a delayed task can also be resumed. There are five services for time management : OSTimeDLY(), OSTimeDLYHMSM(), OSTimeDlyResume(), OSTimeGet(), OSTimeSet().

Inter-task or inter process communication in uC/OS-II takes place using : semaphores, message mailbox, message queues. Tasks and interrupt service routines can interact with each other through an event control block. UC/OS-II semaphores consist of two elements : 16-bit unsigned integer count, list of tasks waiting for semaphore. UC/OS-II provides create, post, pend, accept and query services. Through message mailboxes a task or an interrupt service routine can send a pointer sized variable that points to a message to another task. The available services to message queues are : Create, Post, PostFront, Pend, Accept, Query, Flush. The uC/OS-II message queues are organized as circular buffers. Figure 2.12 presents the way the message queues are organized as circular buffers. The message queue has a starting pointer and an ending pointer that define the the dimension of the message queue. Between OSQOut and OSQIn pointers the message queue stores the pointers to messages.

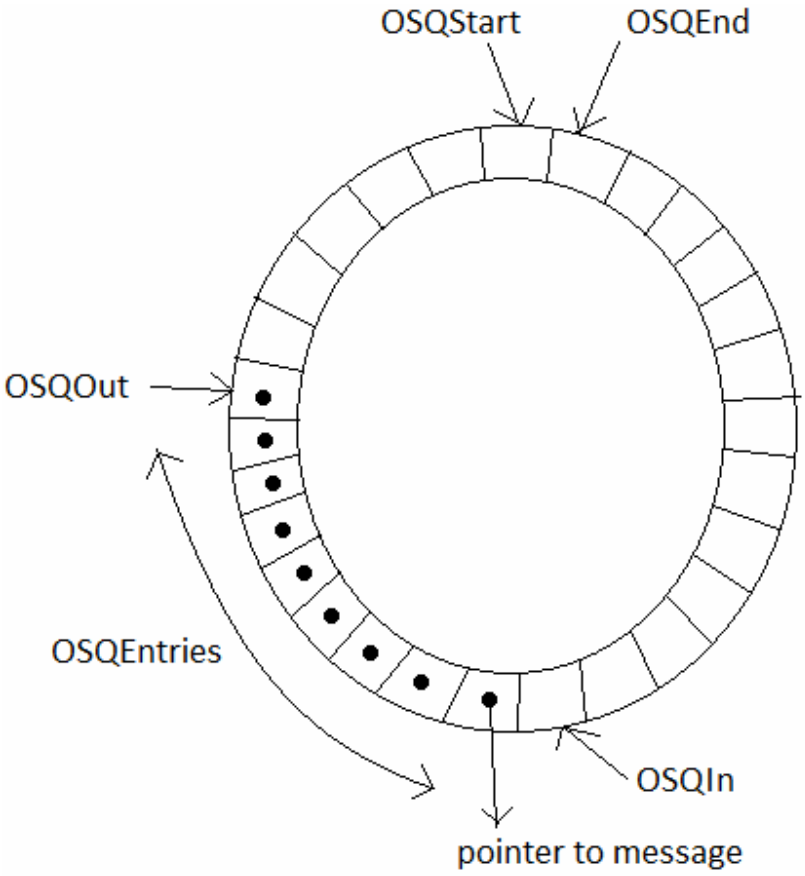

Figure 2.12. Message queue in uC/OS-II

The uC/OS-II file system (uC/FS) is a FAT file system which can be used on any media. To this file system basic hardware access functions has to be provided. This file system offers MS-DOS/MS-Windows compatible FAT12 and FAT16 support. Multiple device driver support allows to access different types of hardware with the file system at the same time. A device driver allows accessing different medias at the same time. UC/OS-II offers OS support for integrating its file system into any OS.

3. Reconfiguration as a service to RTOS

In this I will discuss about systems that integrate real time operating systems, FPGAs and reconfiguration technique.

3.1. Reconfigurable computing system

The reconfigurable computing systems contain a processor embedded in an FPGA [6]. In these systems the hardware tasks are placed in the FPGA. One of the problems of these systems is that the decision where a task is mapped determines the fragmentation of the reconfigurable surface of the FPGA. A high fragmentation can lead to a situation where there is enough space on the chip but it is spread all over the FPGA. In this situation a task that would normally fit in that space could not be mapped.

In figure 3.1. it is presented the system model consisting of a CPU and a FPGA. The CPU runs operating system functions that manage the reconfigurable system resources available in the FPGA. The tasks that arrive are stored in a queue until they are placed

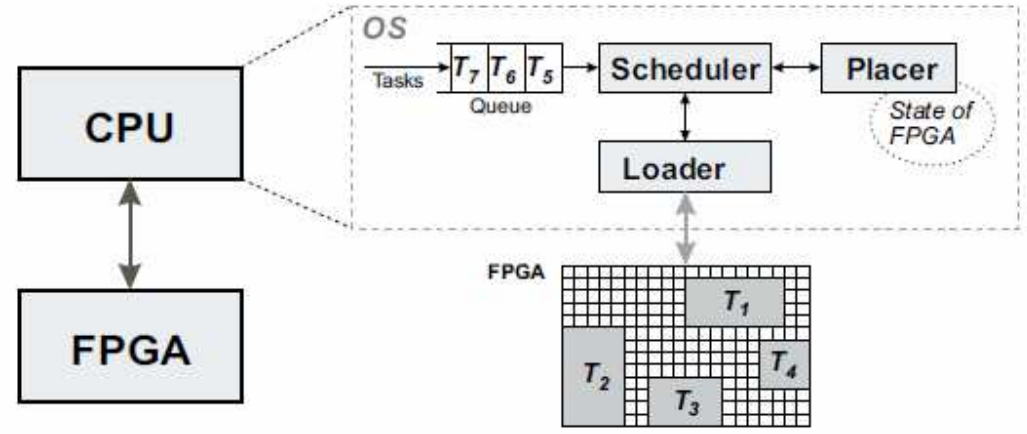

Figure 3.1. The system model and the OS modules [6]

and executed in the FPGA. The scheduler determines which of the tasks should be loaded and executed and calls the placer to determine a good location for the task. Every time the placer finds a good location for the task, the loader conducts all the necessary steps to load the task on the FPGA and start its execution. When the task finishes its execution all the reconfigurable resources occupied by the task are freed.

For this system there can be an arbitrary number of tasks. There are no precedence constraints between the tasks and the execution times are unknown before execution. The system deals with online scenarios, which means that the operating system does not know in advance at what time the tasks arrive and what will be their proprieties. Also the system assumes that the tasks cannot be preempted.

The task can be described by the task width, the task height, the execution time of the task, the arrival time and the maximum allowed time to execute the task. This kind of system [7] has an arrival queue that is sorted according to the timeout of the task. The timeout is the deadline of the task. The scheduler tries to place the task with the closest deadline. The scheduler will reject timed out tasks.

In [8] the authors created a network of FPGAs that were capable of reconfiguring. They connected the system through the RAPTOR2000 system. RAPTOR2000 consists of a motherboard with six application specific extension modules that can connect FPGAs, network interfaces, SDRAM memories. The system also has a PCI interface to connect to the host computer. This system offers high bandwidth and low latency communication between the FPGAs. A software library has been developed that offers access to RAPTOR2000 from C programs on host computer under Windows or Linux. The host computer, a PCI bus device or an ASM can start the reconfiguration of another ASM. Thus it is possible that an FPGA reconfigures itself using the reconfiguration data that is located in the system, hence self-reconfiguration is obtained.

The article [9] proposes an alternative view to dynamic execution of hardware tasks treating them as replacements for the software processes. The solution presented in this paper offers the possibility to integrate in a straightforward manner the software

reconfigurable resources in the software design flow. This paper presents the fact that during creation, communication and destruction the software tasks and the hardware tasks are considered the same. The goal of this paper was to make software and hardware tasks interchangeable during the execution. This means that at run-time there are no strong bindings between the computational resources.

The communication between the hardware and software parts of this kind of systems [9] can be done through FIFOs. Pipe and socket software can be mapped almost directly to the hardware FIFO buffers.

As opposed to [6] where a task runs from the begging to the end without interruption, in [10] the authors propose a system where tasks can be preempted. The preemption is done based on the priority of the task.

In [11] the system consists of a reconfiguration support and a centralized reconfiguration manager that is able to handle dynamic reconfigurability, on top of a Linux based OS. The reconfiguration support has the following benefits in the design of a reconfigurable system : simplification of software calls, increment of code reuse and portability, support of different low-level implementations of the OS reconfiguration support.

In [12] the authors introduce a new method of communicating between the hardware thread and the operating system. In the design, there is a sequential state machine that is used for the interaction between the operating system and the hardware thread. Thus the hardware thread consists of two VHDL processes : the synchronization state machine and the user logic.

As opposed to [6] where the area of the FPGA was only assigned to the tasks in [13] the reconfigurable area is divided into two regions : operating system frames and hardware task area. The operating system frames contains functions that constitute the runtime part of the operating system.

3.2. Placement

Placement in the context of reconfigurable computing systems means that we have a set of tasks that are scheduled by the scheduler and that need to be placed on the FPGA at a moment of time.

For determining where a newly arrived task should be placed, the area must be managed. The area is managed by marking each CLB as free or used and check all possible locations for an arriving task. The areas that are allocated to the tasks have a rectangular shape [6].

The Bazargan's partitioner [6] splits the free space into two smaller rectangles either vertically or horizontally. Because a free rectangle can be split into two new rectangles a binary tree is used to represent the FPGA state. The free rectangles are leaves of the binary tree.

An enhanced Bazargan partitioner [6] can be used, where the splitting decision is delayed and the overlapping rectangles are managed in a restricted form.

Another partitioner is the on-the-fly partitioner [6] that it's implementation differs from the enhanced Bazargan partitioner only in that all rectangles of a subtree might be resized at a later point in time. The on-the-fly partitioning was enhanced with the resizing of rectangles upon task deletion.

There was implemented a faster task placement [6] by starting to load the task immediately after finding a good placement, without waiting to update the internal data structure that contains the binary tree. The placer maintains a hash matrix additionally to the binary tree. A suitable rectangle is found in constant time by using the hash matrix.

The placer should not determine the scattering of small areas in the FPGA, this would lead to a great fragmentation of the FPGA.[7]

In the system presented in [7] another method for placing the tasks in the FPGA is presented. In the reconfigurable matrix of the FPGA if a task uses a cell in the matrix then that cell is represented by a 0, otherwise by a positive number. When a task is inserted the cells above the ones that are occupied by the task are updated. When a task is deleted the cells where the task was are updated to positive number according to the number of the free cell below that cell. This matrix is used because it facilitates the process of checking if there is enough space to allocate a newly arrived task.

In the article [14] a heterogeneous placement algorithm is presented. The algorithm uses a set of cells, each cell having a utilization probability. The position weight is defined as the mean of the utilization probability of the cells corresponding to the feasible position of a requested task m. Deciding which feasible position to select for hardware task placement is done using position weight.

The paper [14] defines a static utilization probability fit algorithm that derives the utilization probability of each cell, the weights of the feasible positions and sorts the feasible positions according to their weight.

The main concepts that static utilization probability fit algorithm uses are used also by the run-time utilization probability fit algorithm [14]. In the case of run-time utilization probability fit algorithm the position weights are updated on each task placement or removal. Based on the available feasible positions of a requested hardware task m the run-time utilization probability fit algorithm generates at run-time the position weights.

In the article [15] the authors propose a three phase algorithm named HeteroFloorplan for determining the resources and the area occupied on the chip by each module of a design. First the method generates a partition tree where it uses min cut partitioning of the module netlist. In the second phase the algorithm performs the topology generation. In the third phase the algorithm performs the realization of the slicing tree on the target FPGA. In this phase the algorithm performs a greedy allocation of rectangular region to a module or a cluster and after that it performs the allocation of RAM and MUL (multipliers) primitives.

3.3 The use of microkernel concept

In the article [16] the authors present an approach towards a reconfigurable RTOS that is able to distribute itself over a hybrid architecture. In order to provide the necessary services for the current application needs the OS is reconfigured online. By analyzing the application requirements the system can decide on which execution domain (CPU or FPGA) the required RTOS components will be placed. A reconfiguration of the OS is necessary whenever the OS components are not placed in the optimal way.

The system is created by using a Virtex 2Pro FPGA as the core of the system. This FPGA chip is used due to its ability to partially reconfigure at run-time and because

it contains two embedded processors. The FPGA has its reconfigurable part divided into n slots. An OS service framework is provided by each slot.

The services that compose the RTOS may run either on the CPU or on the FPGA. There are two versions in which the reconfigurable services are implemented : software and hardware. The application tasks mostly run on the CPU and just the critical ones run on the FPGA. The microkernel concept is used for the target RTOS architecture. In this concept the operating system services and the application are seen as components running on top of a small layer which provides basic functionalities. Communicating in an efficient manner among components running over the hybrid architecture is possible due to the support that the communication infrastructure layer provides.

The problem of assigning RTOS components to the two execution environments is modeled using binary integer programming. The assignment decision needs to be checked continuously due to application dynamism. Hence the use of migration for relocating the components is needed. This implies that a service can migrate from hardware to software or vice-versa.

The RTOS components of the system are located in a limited FPGA area and limited CPU processor workload. Each component has an estimate cost that represents the percentage of the resource from the execution environment used by the component.

In this article the authors define an allocation algorithm. This allocation algorithm is composed of two phases. The first phase starts with an empty CPU and FPGA utilization. It starts to allocate resources on the CPU or FPGA for the components having the smallest cost trying to keep a balance between the CPU and FPGA resource utilization. The second phase means the refining the allocation by changing the previous location of a component pair.

3.4. A simple example of a dynamic partial reconfiguration design

In the article [17] the authors presented a system that used dynamic partial reconfiguration technique. The design was made for a Xilinx Virtex 2Pro FPGA, that contains two embedded PowerPC 405 processors. As seen in figure 3.2, The FPGA was divided into one static region (the pink color inside the FPGA) and two reconfigurable regions (the red color). The reconfiguration was done through the PowerPC 405 processor (the green color), the system having the property of self-reconfiguration. The partial bitstreams were read from a Compact Flash memory and loaded into the partial reconfigurable regions using the internal configuration access port (ICAP). The partial bitstreams contained statistical test from the Nist test suite. The results of these tests were placed on one of the leds. The bus macros (the color yellow) were used to interface the partial reconfigurable regions with the processor system and the static region of the FPGA.

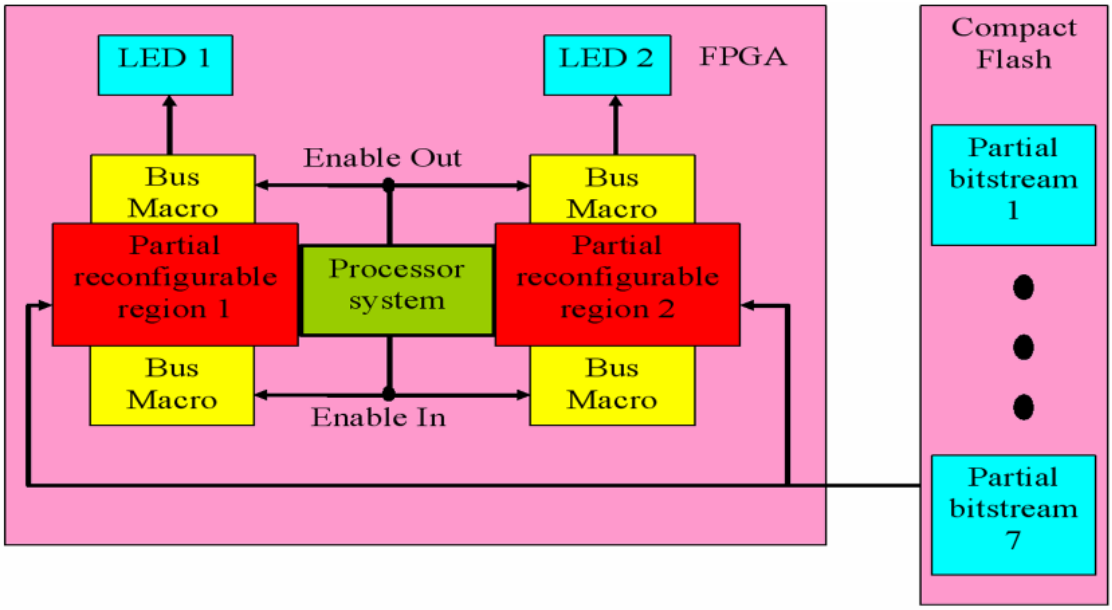

Figure 3.2. The implementation of a simple reconfigurable system [17]

4. Integrating dynamic partial reconfiguration technique into uC/OS-II

In this project the real time operating system uC/OS-II was used. This operating system was enhanced with the ability to work with hardware tasks. This ability is guaranteed by using the dynamic partial reconfiguration technique. In the following the system will be presented and it will also be presented the way to make a preemptive system.

4.1. The system

The system is build using the uC/OS-II operating system and the dynamic partial reconfiguration technique. In figure 4.1 a top view of the system is presented. In this picture it is presented the uC/OS-II component which is a real time operating system running on a processor. In the case of this work, it is the MicroBlaze processor. The Microblaze processor is a soft processor that runs on the FPGA. The operating system contains basic components to ensure the proper working of the applications that run above. The FPGA contains a number of n partial reconfigurable regions in which the hardware tasks run. For each partial reconfigurable region there is data that enters the reconfigurable module and data that exits the reconfigurable module that is placed in the partial reconfigurable region. The hardware tasks that are placed in the partial reconfigurable region communicate with each other through the software task that is created on the uC/OS-II operating system. This task is created when starting the operating system and has a high priority. This task enables the communication of two hardware

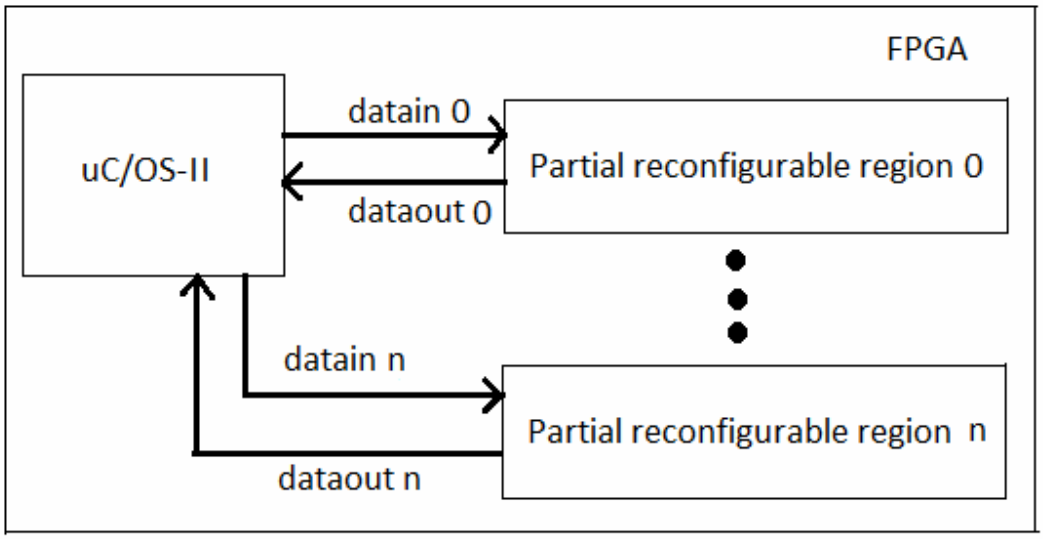

Figure 4.1. The top view of the system

tasks through the use of message mailbox. In figure 4.2 is presented the way a software mail box for the hardware (hardware mail box) is implemented. The hardware mail box is implemented as an array of n

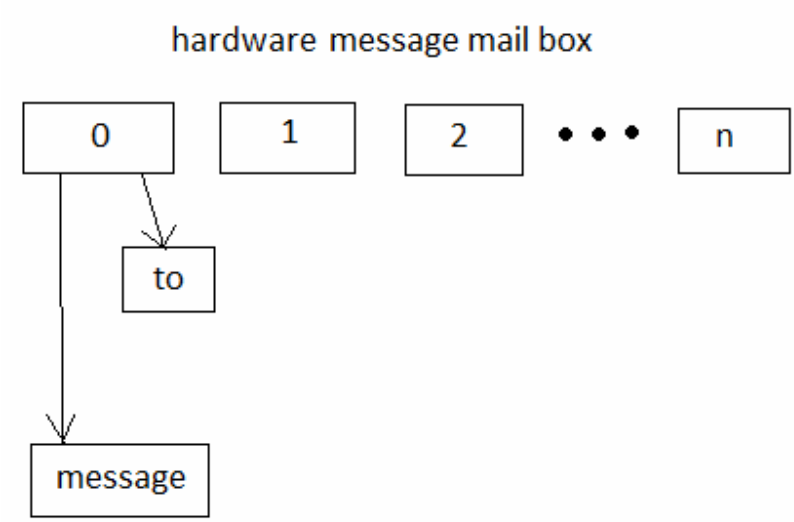

Figure 4.2. Implementation of the hardware mail box

elements, each element contains two references : one reference which is called “to” to the identifier of the destination task and one reference to the message. The indexes in the hardware message mail box array are the identifiers of the tasks. The hardware message mail box works like this : if we have at an index a in the hardware message mail box array and we have stored at the reference to the identifier b then the message is sent from the task with the identifier a to the task with identifier b. The software task when it runs verifies each of the tasks that are working on the FPGA chip and determines which of them send messages. A message is kept in the output buffer of the hardware task until the

message is taken from it and an acknowledgement is sent to the hardware task. Also when a task is waiting for a message the software task sends the message to it and sets a bit in the input buffer of that task in order for that task to know that it has received the message. A better implementation of the hardware message mail box is to have instead of a simple reference to the message a reference to a queue and to introduce the message in that queue. The queue can be of type First In First Out. The reference of to should also be implemented as a queue which should be synchronized with the message one. Figure 4.3 presents this type of implementation. The need of this structure that keeps the messages is

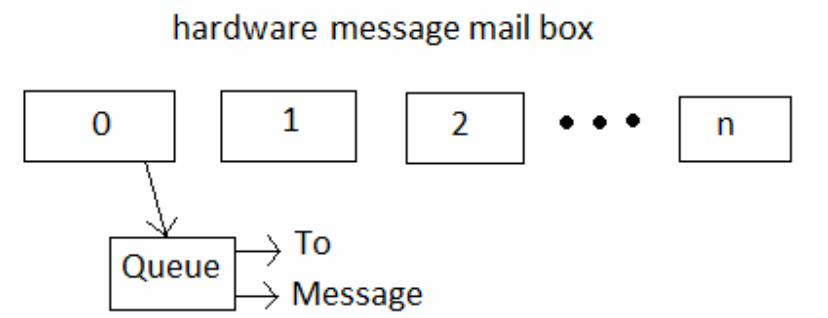

Figure 4.3. The implementation of the hardware message mail box by using queues

that the software task that deals with the hardware task can be preempted before delivering the messages. Also when delivering a message the code that makes the delivery possible should be placed in a critical section.

The necessary information a hardware task should contain in software is presented in figure 4.4. The name field contains the name of the task. The inputs field

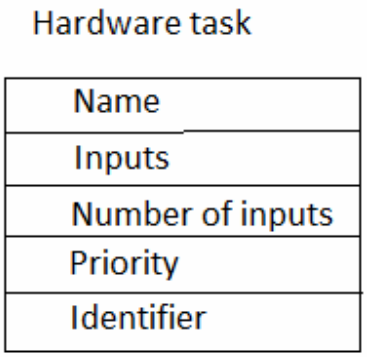

Figure 4.4. The necessary information for a hardware task

contains an array of inputs for the hardware task. The number of inputs field tells how many inputs are in the inputs array. The priority field gives the priority of the task. The identifier field gives the identifier of the hardware task. This structure is kept in the software part of the operating system and it is used when creating a task and before assigning it to the hardware task list.

Figure 4.5. presents the hardware task list. The hardware task list is a list that contains at each node data about the task that its information is stored in that node. The

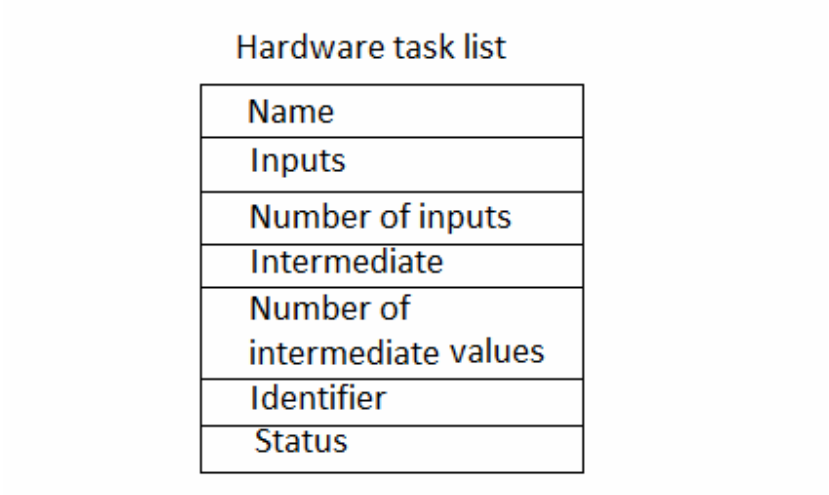

Figure 4.5. Implementation of the hardware task list

List is indexed by the priority of the task, so hardware task list[4] gives the information related to the hardware task stored at index 4 in the list that has the priority 4. Each node in the list contains the following information : the name of the hardware task, the inputs of the hardware task, the number of inputs of the hardware task, the intermediate values of the hardware task, the number of intermediate values of the hardware task, the identifier of the hardware task and the status of the hardware task. The intermediate values of the hardware task refers to the values that are stored in the software part of the operating system when a hardware task is preempted.

In figure 4.6 it is presented the way the hardware scheduler and hardware context switch is called. Periodically the timer component calls the software scheduler to schedule the new task that is going to run on the processor. Then the software context switch is called. If the scheduler schedules a different task than the one that is on the processor then the context switch has to preempt the task that is running on the processor and to allocate the processor to the new task. If the software task that is scheduled is the one that deals with the hardware tasks then it will call the hardware scheduler and the hardware context switch. The hardware scheduler schedules the hardware task for each of the different partial reconfigurable regions. The hardware context switch is responsible for storing the information related to the hardware task that is running on the FPGA and doing the partial reconfiguration. By using the dynamic partial reconfiguration technique the old hardware task is swapped with the new hardware task.

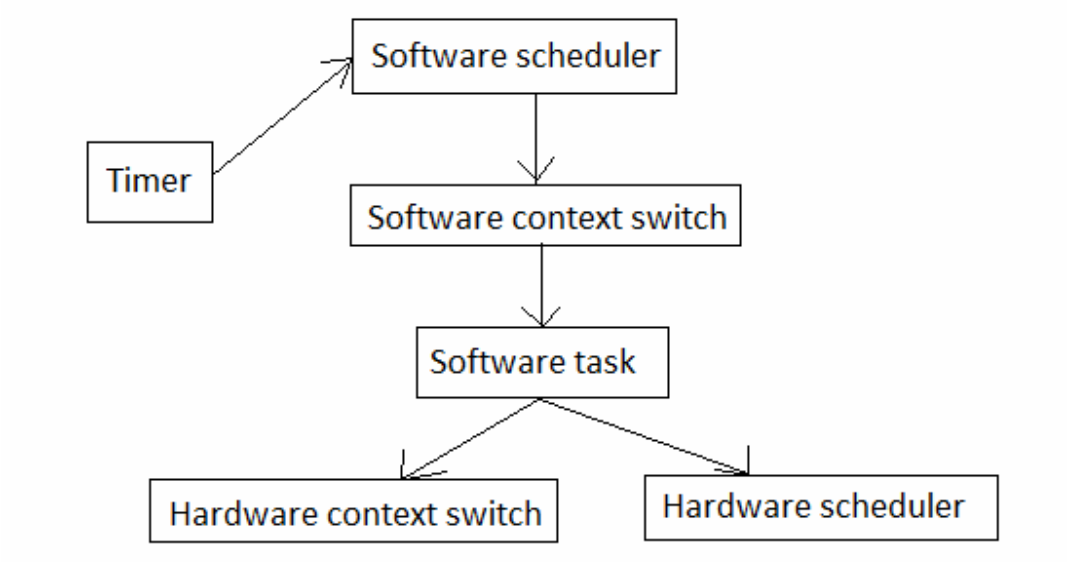

Figure 4.6. The way the hardware scheduler and hardware context switch is called

In the following there will be presented the main part of the algorithm for scheduling the hardware tasks and the algorithm for switching the context. The algorithm for scheduling is presented in the following :

```
int OS_SchedHw (void)
{
        running = MAXHARDWARETASKS;
        foreach identifier of a hardware task
                if the identifier of the hardware task indicates a hardware task
                running then running = the value of the identifier;
                fi;
        end foreach;
        higherPriorityTask = running;
        foreach identifier that is smaller than running
        if there exists a higher priority task then higherPriorityTask = identifier;
                break;
        fi;
        end foreach;
        return higherPriorityTask;

}
```

In this function the first foreach structure tests if there is a tasks that is currently running. The second foreach structure tests if there is a task that has a higher priority than the one that is running. If there is a task that has a higher priority than the task that is currently running than the scheduler schedules that task to be executed. If this is not the case than the scheduler returns the value MAXHARDWARETASKS which means that there isn’t any new task that should be scheduled and run on the next context switch.

In the following the algorithm for the hardware context switch will be presented :

```
void OS_HardwareCtxSw(int prio)
{
        If prio==MAXHARDWARETASKS then return;

        Foreach identifier of a hardware task
        If the hardware task has the status set to running then
                If the hardware task is running then
                        Preempt the task
                        Wait until the tasks stops running
                        Copy the intermediate values that the task outputs
                        Modify the status of the task to preempted
                Else
                        Modify the status of the task to stopped
                Fi;
                Break;
        Fi;
        End foreach;
        If the task that has the priority prio has the status preempted then
                Restore the task to its running state with the intermediate values;
                Change the status of the task to running;
        Else
                Dynamically swap the module that resides in the region where the
                hardware task is placed
                Change the status of the task to running;
        Fi;

}
```

The pseudocode presented above describes the way a context switch is done for the hardware tasks. The software first tests if the priority of the hardware task that was obtained by scheduling is equal to MAXHARDWARETASKS. If it is equal then this means that the scheduler did not find any task to schedule and no context switch is needed. If it is not equal then the context switch procedure tests to see if there is a task that is running and that needs to be preempted. If there is a hardware task that needs to be preempted the task is preempted and its intermediate values are stored. The preemption is done only after waiting for the task to execute until a point where it can be preempted. The status of the task is modified to preempted. If the hardware task stopped running then it does not need to be preempted and its status is modified to stop. If the task that has the priority prio has its status preempted then the intermediate values of the task are restored to the hardware task and its status is modified to running. If the status of the hardware task is not preempted then this means that the task is a task that needs to be started. In this case the hardware task is reset and then started and its status is modified to running.

4.2. Designing a preemptive hardware architecture

In figure 4.7 it is presented a hardware architecture that can be seen as a preemptive hardware task. The architecture contains n functional units and has n inputs. The inputs are buffered in n registers that are placed at the input of the system. The system receives a global clock signal. The preemptive part of the system is made of the middle registers, the multiplexers, the demultiplexer and the counter. Each result of the processing done by the functional unit is stored into the middle registers. The preemption can occur during the time a functional unit is doing its work, but this preemption will not be taken into account in that moment, it will be taken into account only when the functional unit writes the middle register. So if the preemption occurs than the output of the n output multiplexer will output the value that is stored in the middle register. So the software part of the operating system can store the intermediate value. Also the software part has to store the value of the counter and the value of the outaddress signal. When the hardware task is resumed the value of the result that was stored at preemption is put on the instored line, the value of the counter is loaded into the counter and the value of the signal outaddress that was stored at preemption is inserted on the inaddress line. So this architecture permits preemption but only at specific moments in time.

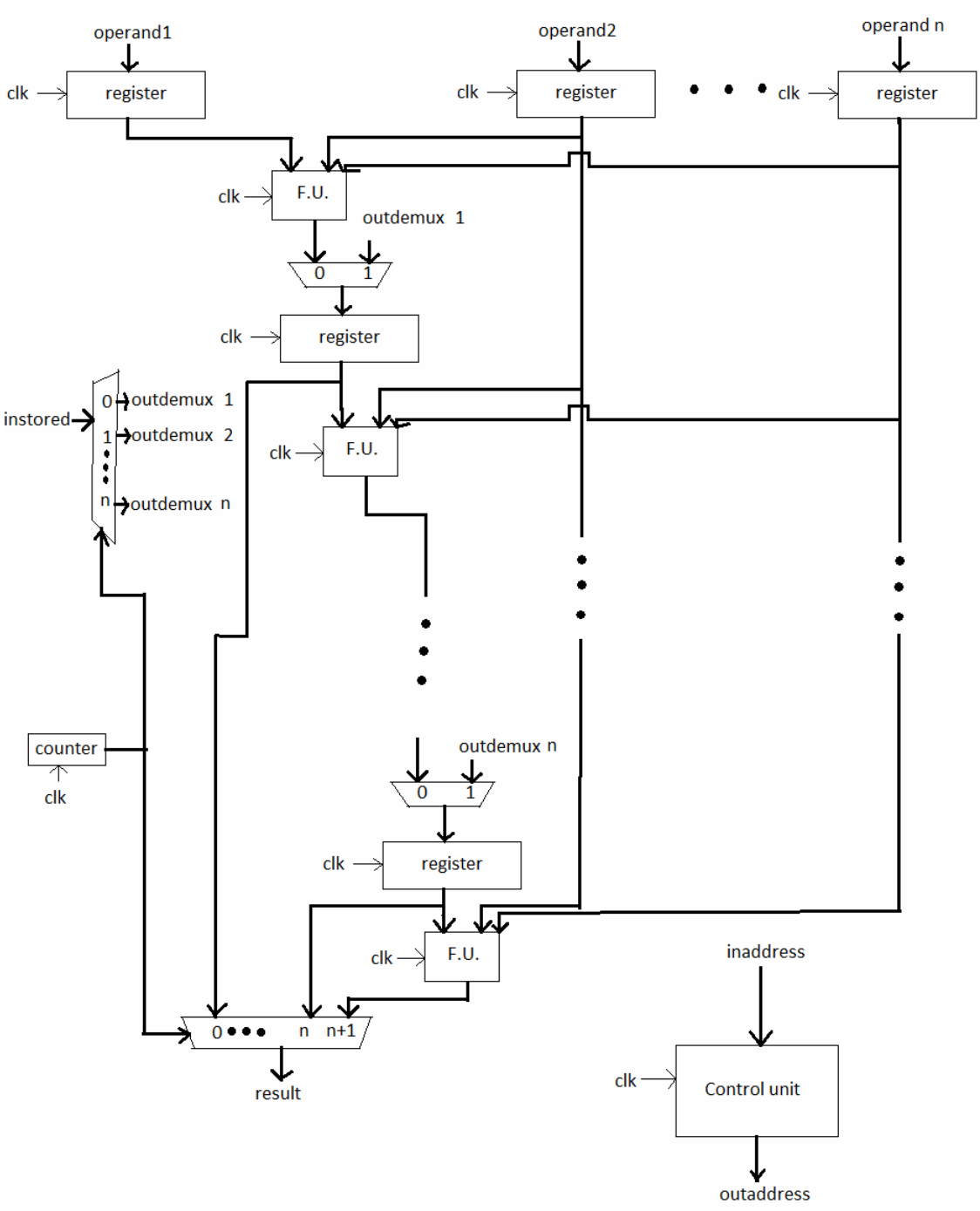

Figure 4.7. A hardware architecture that can be seen as a preemptive hardware task

5. Results

During the development phase there were implemented more than one hardware architectures. These hardware architectures were implemented because the operating system does not fit the local memory available on the FPGA chip. In figure 5.1 is

presented a hardware architecture that involves the usage of DDR2 SDRAM external memory to hold the instruction, data, heap and stack. In this picture it can be seen that the

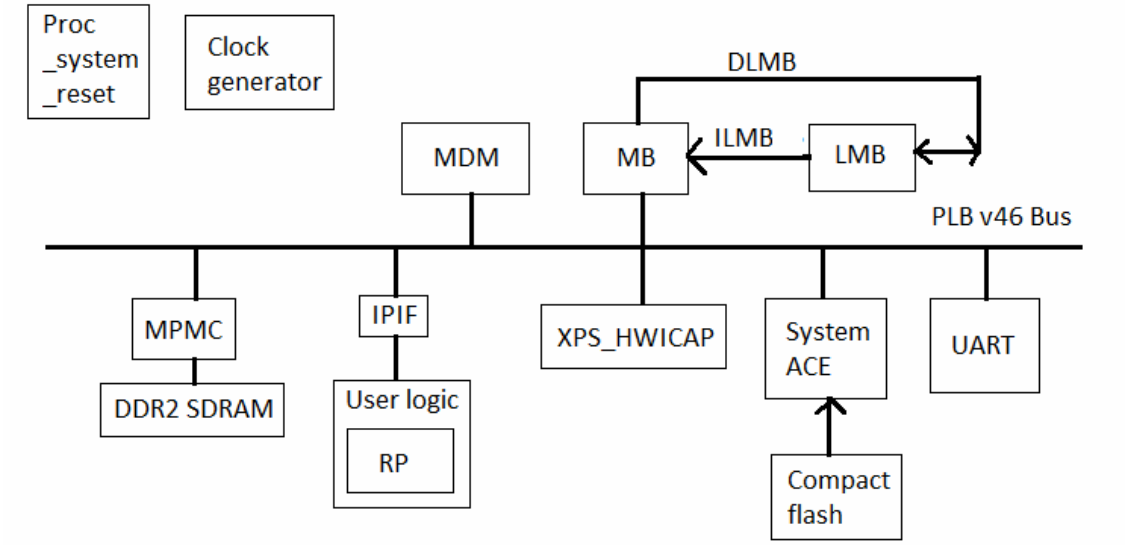

Figure 5.1. The hardware architecture that contains the DDR2 SDRAM external memory

DDR2 SDRAM memory is connected to the MPMC controller that controls its way of working. The MPMC controller is connected to the PLB v46 Bus. The reconfigurable partition RP is stored in the User logic block that connects to the IPIF module. The IPIF module connects to the PLB v46 Bus. The MDM module is used for debugging the system and it is connected to the PLB v46 Bus. The XPS_HWICAP module controls the hardware ICAP (Internal Configuration Access Point). The ICAP is used during the partial reconfiguration design to swap the reconfigurable modules. The Compact flash card is connected to the System ACE controller. This controller is connected to the PLB v46 Bus and its main task is to control the Compact flash, to read or write to the Compact flash. The UART is connected to the PLB v46 Bus and it offers a connection from the system to the user, in which the user can observe the way the system is working. The instruction segment, the data segment, the stack and heap segments for the application that run on this architecture was kept in DDR2 SDRAM. The system worked well in the Xilinx software development kit, but when it was used with dynamic partial reconfiguration it did not work anymore. The problem that happened is that when the system started it did not find the correct address for the program that was stored in the DDR2 SDRAM memory. This way the code that was stored in the DDR2 SDRAM memory, in the instructions segment could not be executed so the system stopped working. This problem was encountered for the system that used the dynamic partial reconfiguration technique. The way to manage this problem is to use a boot loader program that when executed should load into the system the instruction available in the DDR2 SDRAM at a specific address.

In figure 5.2 is presented another version of the hardware architecture in which the DDR2 SDRAM memory was replaced with a BRAM controller and a BRAM memory. The BRAM memory (BlockRAM memory) is connected to the PLB v46 Bus

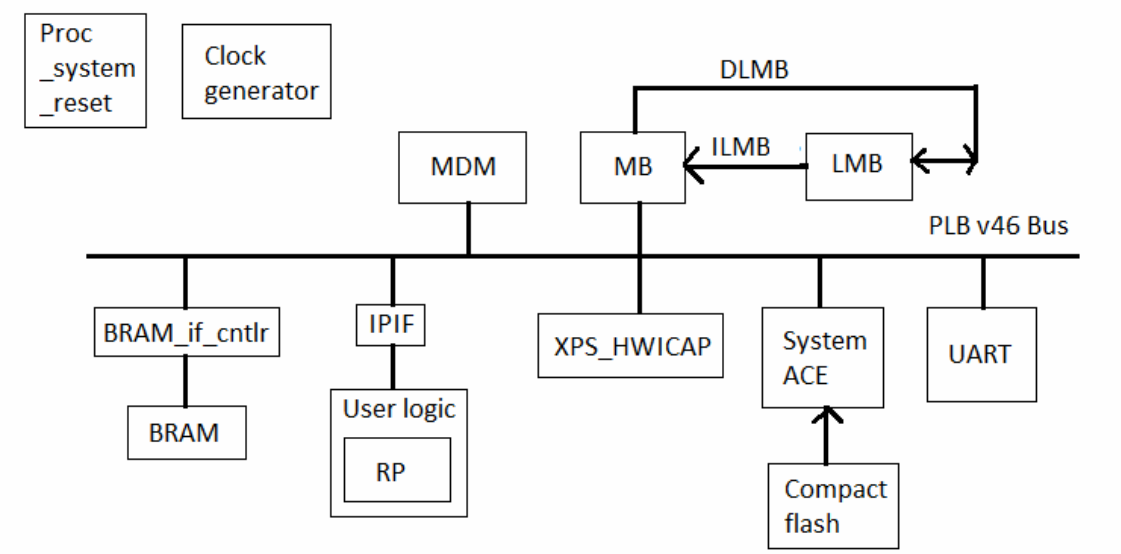

Figure 5.2 A version of the system that contains a BRAM memory

through the BRAM_if_cntlr. This controller controls the reading and writing to the BRAM memory. The problem that I have encountered in this design is the same as the previous one, the system didn't know that the code was kept in the BRAM memory. So at the start of the application the system did not know that it had to jump to an address in the BRAM memory. This problem was encountered for the system that used the dynamic partial reconfiguration technique.

In figure 5.3 it is presented the same system without BRAM memory and with a

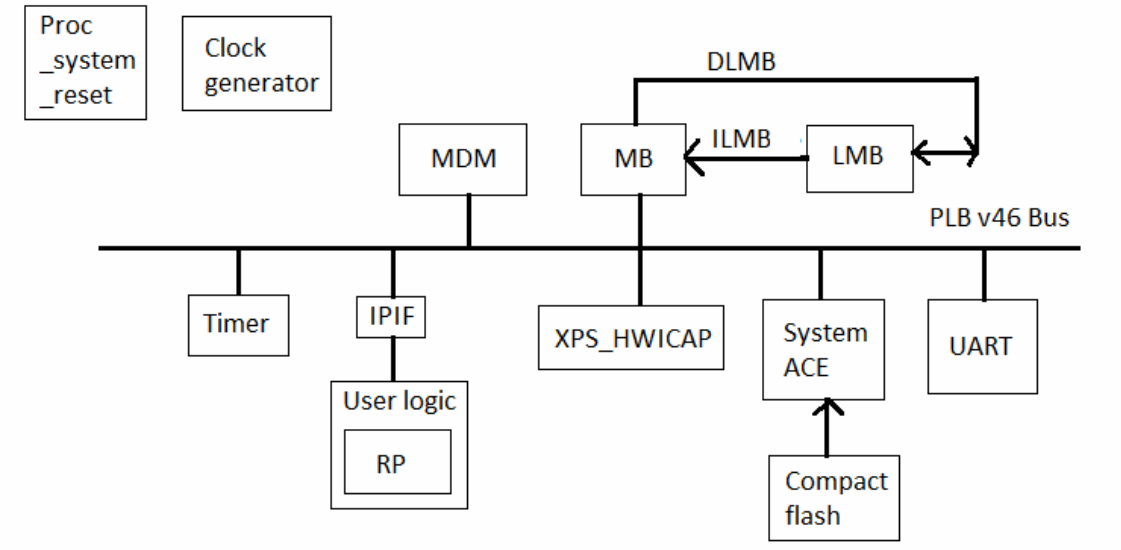

Figure 5.3. The hardware architecture of the system with a timer

timer. The system works correctly in Xilinx software development kit but when the dynamic partial reconfiguration technique is introduced it does not work properly anymore. The problem is the fact that between the timer and the function that initializes the ICAP there is a conflict. The only solution for solving this conflict was to remove the

timer from the system. Instead of using a hardware timer the operating system running on the Microblaze processor is using a software timer.

This way I obtained the final version of the hardware architecture that is presented in figure 5.4. The instruction, the data and the heap and stack are kept in the local

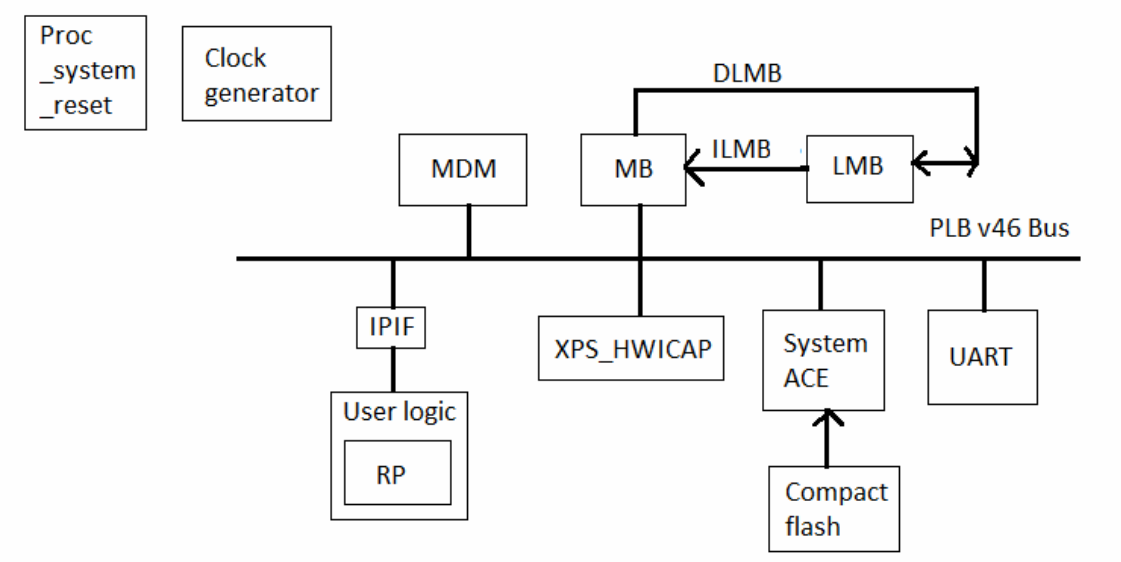

Figure 5.4. The final version of the hardware architecture

memory that is controlled by the LMB controller. On the compact flash it is kept the full bit stream that is used to first configure the system and the partial bit streams. The UART implements the RS232 protocol and is connected to the RS232 connector of a computer through a connection cable. The data is viewed on the computer through the hyperterminal.

In figure 5.5 is presented the hardware architecture of the preemptive hardware task that contains a hierarchy of multipliers. The architecture is a particular version of the one that was presented in the section 4 of this report (figure 4.7). This architecture is the one that it is placed as reconfigurable module in the reconfigurable region. The architecture contains 4 multipliers placed in a hierarchy. The multipliers are separated from the each other by a set of registers. The architecture is preemptible. The mechanism of preemption is done through the multiplexers and the demultiplexer present in the architecture. When a preemption signal is activated the system waits for the data signals to reach one of the internal registers of this architecture. When the data arrives at one of these registers the data is inserted into the register and after that it is sent to the big multiplexer that is placed at the bottom of the architecture. Depending on the register that emitted the data the counter selects the appropriate in[ut into the multiplexer. The output of the big multiplexer that is at the bottom of the hierarchy is stored along with the value of the counter and the value of the address that the control unit outputs. When the task is resumed the value of the output of the multiplexer is placed on the input line of the demultiplexer (instored), the value of the counter is loaded and the address for the control unit.

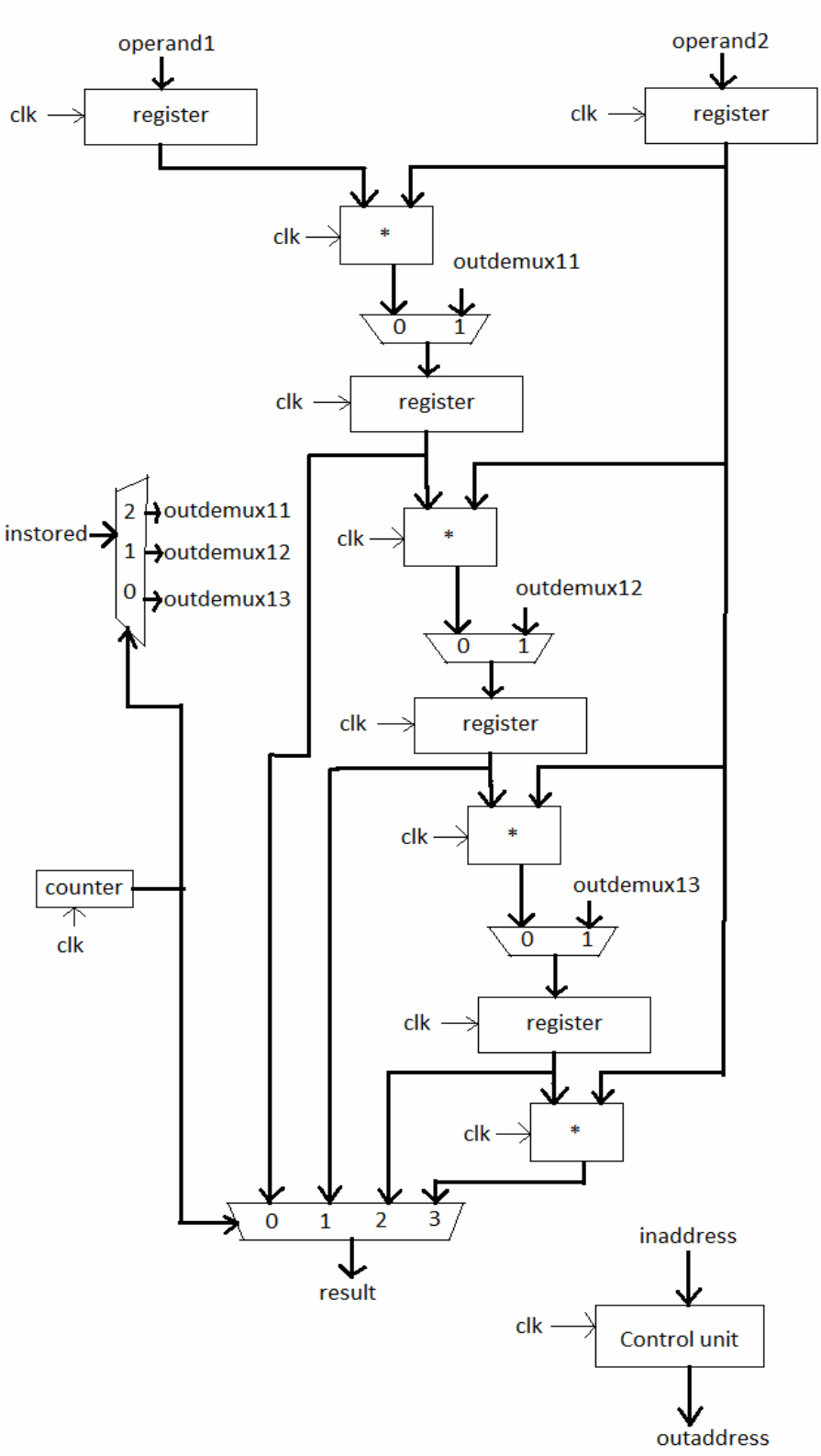

Figure 5.5. Hardware architecture of the hardware task that contains a hierarchy of multipliers

In figure 5.6 it is presented the hardware architecture of the hardware task that is composed of a hierarchy of dividers. The functioning of this architecture is the same as the one presented above for the task that is composed of a hierarchy of multipliers.

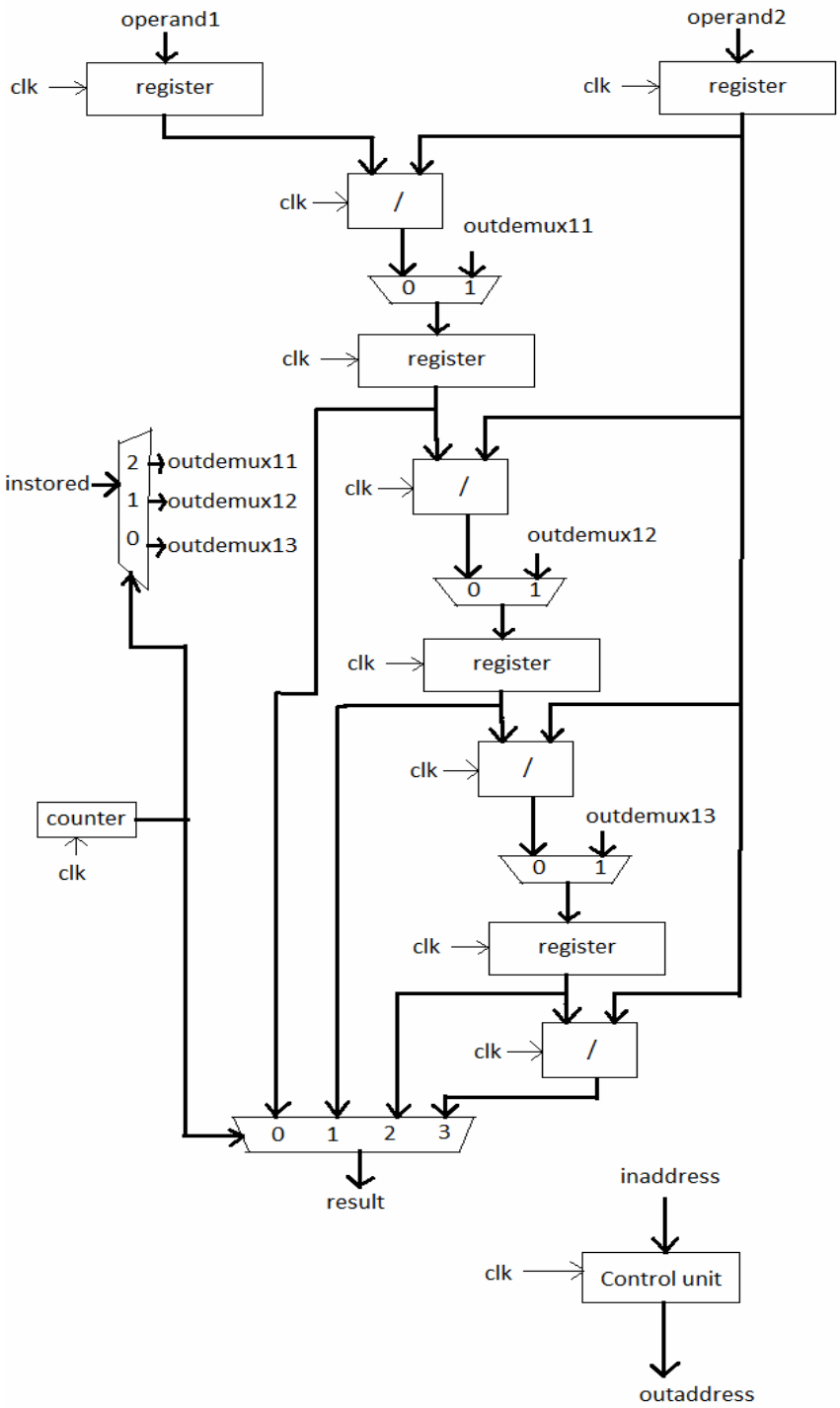

Figure 5.6. The hardware architecture of the hardware task that is composed of a hierarchy of dividers

In figure 5.7 it is presented the difference between the execution time in cycles between the implementation of the hierarchy of multipliers and dividers in software and in hardware. So it can be observed that in the case of hierarchy of multipliers the

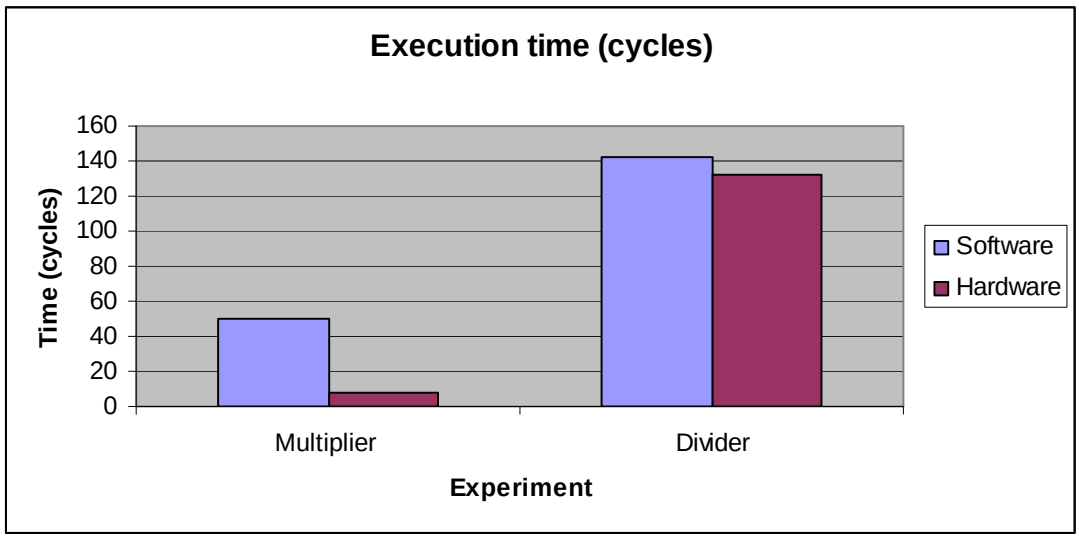

Figure 5.7. The difference in execution time for the software and the hardware implementation of the hierarchy of multipliers and dividers

difference is significant. The number of cycles that the hardware implementation of the hierarchy of multipliers is 8 and in the case of software implementation is 50. The number of cycles that the hardware implementation of the hierarchy of dividers is 132 and in the case of software implementation is 142. The data for the software implementation is taken by studying the assembly code that resulted from writing the code in C language. The clock frequency of the Microblaze processor is the same as the one that drives the hardware implementations so the number of cycles can be compared and accounted as time. The low value for the execution time of the hardware implementation of the multipliers can be explained through the fact that there were used DSP 48 components that contain hardware multipliers that have a latency of one clock cycle. From this data we can conclude that the hardware implementation of the hierarchy of multipliers is executing faster than the software implementation. This result motivates us to implement the tasks in hardware because they will execute faster than in software.

In the table 5.1 it is presented the usage of the resources available on the FPGA by the complete configuration containing alternatively the hierarchy of dividers and the hierarchy of multipliers.

Table 5.1. The usage of resources available on the FPGA

	Design with hierarchy of multipliers	Design with hierarchy of dividers
Number of slice registers	3112 out of 69120	3068 out of 69120
Number of slice LUTs	3106 out of 69120	3114 out of 69120
Number of occupied slices	1548 out of 17280	1519 out of 17280

In the case of the design containing the hierarchy of dividers the maximum allowed clock frequency is 107.643 MHz. The maximum allowed frequency for the design containing the hierarchy of multipliers is the same as for the design containing the hierarchy of dividers.

The reconfiguration time for the reconfigurable module that holds the hierarchy of multipliers as well as the one that holds the hierarchy of dividers is 244 ms. The reconfiguration time is directly proportional to the dimension of the reconfigurable area. In the case of the hierarchy of multipliers and dividers the partial bitstream is 103 KB. The full bistream is 3799 KB.

6. Conclusions

During this project it was modified the uC/OS-II operating system to accommodate the partial reconfiguration technique. The service which was introduced implied the use of hardware tasks for which the context switch is done through the use of dynamic partial reconfiguration. There were created two hardware tasks , one which contained a hierarchy of multipliers and one which contained a hierarchy of dividers. These hardware tasks were managed by a software task running on the uC/OS-II operating system. The uC/OS-II operating system was running on the Microblaze soft processor. In this work it was presented the complete system hardware architecture, that did not contain a hardware timer. The job of the hardware timer was done by a software timer, that simulated the working of the hardware timer.

The results that were obtained indicated that the software version of the hierarchy of multipliers and dividers was executing in more time than the hardware implementation of these hierarchies. Also the results of this project indicated that the reconfiguration time is considerably high (hundreds of ms), so in order to be efficient to use the partial reconfiguration technique we need that the hardware task to execute far more rapidly than the software counter part. This way the time for reconfiguration plus the time for execution of the hardware task to be less than the time needed by the software task to execute.

References

[1] Modular design. Available at:
http://www.xilinx.com/itp/xilinx7/books/data/docs/dev/dev0025_7.html
[2] Two flows for partial reconfiguration : Module based or small bit manipulation. Available at: http://www.kip.uni-heidelberg.de/~abel/projekte/HWS/xapp290.pdf
[3] Difference based partial reconfiguration. Available at:
http://www.xilinx.com/support/documentation/application_notes/xapp290.pdf
[4] Partial reconfiguration of Virtex FPGAs in ISE 12. Available at:
http://www.xilinx.com/support/documentation/white_papers/wp374_Partial_Reconfig_Virtex_FPGAs.pdf
[5] http://micrium.com/page/home

[6] Herbert Walder, Christoph Steiger, Marco Platzner, *Fast Online Task Placement on FPGAs: Free Space Partitioning and 2D-Hashing*, International Parallel and Distributed Processing Symposium, Nice, April 2003.

[7] W.Hu, C. Wang, J.L. Ma, T.Z. Chen, D. Chen, *A novel approach for finding candidate locations for online FPGA placement*, Bradford, June 2010.

[8] Vincenzo Rana, Marco Santambrogio, Donatella Sciuto, Boris Kettelhoit, Markus Koester, Mario Porrmann, Ulrich Rückert, *Partial dynamic reconfiguration in a multi-FPGA clustered architecture based on Linux*, Long Beach, March 2007.

[9] Krzysztof Kościuszkiewicz, Fearghal Morgan, Krzysztof Kępa, *Run-Time Management of Reconfigurable Hardware Tasks Using Embedded Linux*, Kitakyushu, December 2007.

[10] Diana Göhringer, Michael Hübner, Etienne Nguepi Zeutebouo, Jürgen Becker, *CAP-OS : operating system for runtime scheduling, task mapping and resource management on reconfigurable multiprocessor architectures*, Atlanta, April 2010.

[11] Marco D. Santambrogio, Vincenzo Rana, Donatella Sciuto, *Operating system support for online partial dynamic reconfiguration management*, Heidelberg, September 2008.

[12] Enno Lübbers, Marco Platzner, *RECONOS : an RTOS supporting hard and software threads*, Amsterdam, August 2007.

[13] Christoph Steiger, Herbert Walder, Marco Platzner, *Operating systems for reconfigurable embedded platforms : online scheduling of real-time tasks*, Journal IEEE Transactions on Computers, Volume 53 Issue 11, November 2004.

[14] Markus Hoester, Mario Porrmann, Heiko Kalte, *Task placement for heterogeneous reconfigurable architectures*, Singapore, December 2005.

[15] Pritha Banerjee, Susmita Sur-Kolay, Arjit Bishnu, *Fast unified floorplan topology generation and sizing on heterogeneous FPGAs*, May 2009.

[16] Marcelo Götz, Florian Dittmann, *Reconfigurable microkernel-nased RTOS : mechanisms and methods for run-time reconfiguration*, San Luis Potosi, September 2006.

[17] Dan Hoțoleanu, Octavian Creț, Alin Suciu, Tamas Gyorfi, Lucia Văcariu, *Real-time testing of true random number generators through dynamic reconfiguration*, Lille, September 2010.

[18] Jean J. Labrosse, *MicroC/OS-II The Real-Time kernel Second Edition*, CMP Books, 2006.