

COMPUTATIONAL MODELING AND SIMULATION OF RECONFIGURABLE RESPONSIVE EMBEDDED COMPUTING SYSTEMS

Dietmar P. F. Moeller
University of Hamburg, Department Computer Science
Chair Computer Engineering, AB TIS
Vogt Kölln Str. 30, D-22527 Hamburg, Germany
dietmar.moeller@informatik.uni-hamburg.de

INTRODUCTION

Many computer applications required as solution provider inside larger, in general, complex systems. Hence these computer systems are embedded within the real complex systems, which themselves may consist of mechanical, electrical and/or electronical components etc. This led to the well-known terms of embedded systems, and often emdedded processors or embedded control. Embedded systems, as well as embedded control, with high reliability and hard realtime constraints, are the top end of embedded systems design, called embedded responsive systems.

With the growing demands in this field, solutions are requested integrating operating systems, realtime capabilities as well as fault tolerance within a overall system approach. Due to those facts the term responisble system was introduced for such type of complex and configurable architectures. The most important features of responsive systems deal with reliability and realtime capability.

Reliability as well as hard realtime capability demand both for intelligent scheduling algorithms. Scheduling means, that even in the sequentialized world of the processor concurrent requests may be served. Normally this is performed by static or dynamic scheduling algorithms with well-known advantages and disadvantages and checked by simulation methods.

But intelligent scheduling is nothing else as scheduling per se because looking for the best priority schedule of the tasks in progress. Hence the hardcore boundary for responsive systems cannot be solved by the scheduling schemes known up to now. The other aspect is that the embedded controller as well as the embedded processor architecture inside shows no difference at all in compa-rison to 'normal' architectures.

This paper introduces a block oriented structure for em-bedded controller as well as embedded processors in the area of *responsive systems*. This architecture will include computing capabilities for high performance tasks as well as hardware support for scheduling, and this yields to the need of simulation for system design and evaluation. Even fault tolerance – partial as well as global – is possible done based on simulation within the embedded controller architecture. This support should be introduced as hard-ware/software partitioning based on an extended block-oriented

microarchitecture model for the embedded con-troller design in *responsive systems*.

The demands for the simulation method change from separated hardware, operating system and application software simulation to integrated system co-simulation methods due to the mutual dependance of all system parts.

Using system modelling languages like VHDL for the block oriented FPLD architecture it could be shown that some of the simulation results are obtainable by abstraction but some not. Field Programmable Logic De-vices (FPLDs) like Field Programmable Gate Arrays (FPGAs) have emerged as an ultimate solution to the time-to-market and risk problems in embedded controller design because they provide instant manufacturing and low-cost prototyping e.g. of embedded processor kernels, using simulation as a powerful design tool. An FPLD is a device in which the final logic structure can be directly configured by the end user, without the use of an inte-grated circuit fabrication facility, because the user only needs some simple electrical equipment for programming purposes.

First type of programmable logic device introduced into the market was the Programmable Read Only Memory (PROM), a one-time programmable device that consists of an array of read-only cells. Later these cells were as-sembled in EPROM or EEPROM technology enabling the re-programmability. A specific logic circuit can be imple-mented by using the PROM's address lines as the circuit's inputs, and the circuit's outputs are then defined by the stored binary information. Based on this strategy, any binary function

$$F := B^n \rightarrow B^k$$

can be realized

RESPONSIVENESS OF EMBEDDED INFORMATION SYSTEMS

In embedded information systems it is important to perform with the correctness of computations in a timely manner, meaning responsiveness of the embedded computing system, that requires correctness of the computation stringent timing constraints due to

- Reliability
- Predictability

- Adaptability
- Timely
- Criticalness

Responsiveness can be achieved with real-time embedded information systems. Real-time embedded information systems are fundamentally composed of two or more concurrent processes that execute with stringent timing requirements and cooperate with each other in order to accomplish a common goal. Typically, a real-time embedded information systems have timeliness requirements, typically in the form of deadlines that can't be missed. They consists of a control system and a controlled system. For example, in an automated production line, the controlled system is the production floor with its robots, assembling stations, and the assembled parts, while the control system is the real-time embedded information systems and the human computer interface that manage and coordinate the activities on the factory floor. Henceforth, the controlled system can be viewed as the environment with that the embedded information systems interacts, based on the information available, that requires periodic monitoring of the environment as well as timely processing of the sensed information.

Responsive or real-time embedded information systems differ from conventional information systems while having deadlines or explicit timing constraints, that are attached to tasks, to that the responsive or real-time embedded information systems has to react on-time, otherwise timing faults may cause catastrophic consequences. In addition to timing constraints, a task may also posses the following types of constraints and requirements:

- Resource constraints, means that a task may require access to certain resources other than the CPU such as I/O devices, data structures, files, etc.
- Precedence relationship, means that a complex task may require access to many resources, that can be sufficiently handled break it into several subtasks related by precedence constraints and each requiring a subset of that resources
- Concurrent constraints, means that tasks should be allowed concurrent access to resources providing the consistency of the resources is not violated
- Communication or networking requirements, that are mostly timing requirements
- Placement constraints, means that instances of a task are executed – for fault-tolerant reasons – on different processors
- Criticalness, means that meeting the deadline of one task may be considered more critical than another.

The characteristics of the various application specific tasks are usually known a priori and can be scheduled statically or dynamically. In static scheduling, boosted by parallel code optimization, the compiler of the embedded information system has to

detect and to resolve data, control and resource dependencies during code generation. Moreover, the compiler also performs parallel code optimization. The static specification of schedules is typically in use for periodic tasks, the opposite is true for use in a-periodic tasks. Let an embedded system being static. The characteristics of the controlled system, are assumed to be known a priori, and, hence, the activities and the sequences in which these activities take place can be determined off-line, means before the system is running. Embedded information systems of such type are quite inflexible even though they may incur lower run-time overheads. Whereas a large proportion of currently implemented traditional real-time systems are static, by necessity, responsive embedded computing systems has to adopt solutions that are more dynamic and flexible. Hence, responsiveness in embedded systems requires for system characteristics, such as:

- Fast
- Predictive
- Reliable
- Adaptive

Therefore, a responsive embedded information system has to be fast and predictable. Predictability means that a task that is activated should be possible to determine its completion time with certainty. This can be done taking into account the state of the system and the task resource needs.

Reliability is another prerequisite on a responsive embedded information system, meaning that real-time constraints cannot be achieved if embedded information system components are not reliable. The degree of reliability has to be specifiable and predictability, and will involve determining system performance under different levels of reliability.

Adaptability is of particularly importance for a responsive embedded information system, because if a task deadline can be met only under a restricted system state or configuration, reliability and performance may be compromised.

There are three popular strategies on static scheduling:

- Register distribution, that is mainly used to avoid structural hazards
- Loop reorganization, that prevents both structural and data hazards
- Code promotion, that is used to compensate for losses caused by data hazards

Often code is written such that its behavior is clear to other programmers. Since code is optimized for the programmer its typically not optimized for the architecture. Programmers will often dedicate general-purpose registers to the same task throughout their code. This will make code easier to understand but it can place unnecessary over use some registers and under use others. This creates competition for the use of the over used registers and this causes structural hazards. This is unfortunate since there are usually other registers in the architecture that could be used to avoid the structural hazard. Register redistribution techniques can be applied to a code after the software

designer has written it to evenly distribute the register usage. In this way the software designer can create the code from the designers perspective and the static scheduler can optimize it for the architecture.

Just as the programmer specifies register utilization for readability loops are written to be readable. For this reason the loop index calculations are often calculated at inefficient times. A good example is a loop written to traverse the pixels in an image. The following program code excerpts are written in C and demonstrate loop index inefficiency.

Example 1:

```
char* pImage1;
char* pImage2;

int nRows = 480;
int nCols = 640;
int row, col;

for( row = 0; row < nRows; ++row )
{
    for( col = 0; col < nCols; ++col
    )
    {
        pImage1[(row*nCols) + col] =
        pImage2[(row*nCols) + col];
    }
}
```

Example 2:

```
char* pImage1;
char* pImage2;

int nTotalPixels = 640 * 480;
int pixelIndex;

for( pixelIndex = 0; pixelIndex <
nTotalPixels; ++pixelIndex )
{
    pImage1[ pixelIndex ] =
    pImage2[ pixelIndex ];
}
```

In both examples the data in *Image1* is copied to *Image2*. The difference is the logical technique used for traversing the images and the system for computing the image indexes. In Example 1, two separate loop indices are used, requiring two registers. In Example 2, only a single index is used. Not only are more registers used for the same task but also they both must be compared to detect loop completion. In Example 1 the index of an individual pixel is computed using an integer multiply and an integer add. Additionally this computation is executed twice even though the two computations will have the same result. The total number of computations executed in Example 1 is 614400 multiplies and 922080 adds. In Example 2 the pixel index for both images is the same integer and its computed using a single integer add. Additionally the

integer used as the pixel index is the loop index. In Example 2 the total number of computations is 0 Multiplies and 307200 adds. Example 2 has less than one third the number of adds and expert two and has no multiplies. The relative cost reduction of Example 2 is clear from these figures. The importance of loop reorganization is shown by the fact that Example 2 is easier to understand from a programming perspective. Its very structure suggests that it is traversing through the rows and columns of an image and copying the data across. For this reason the example given above will be written more commonly by software designers. A loop reorganization strategy would identify the dependencies in Example 1 and factor out the redundant actions. After this process Example 2, which was clear to the programmer would be synthesized as Example 1.

Avoiding data hazards is often achieved by simply stalling the issue of instructions that may be affected by a data hazard. When this is done the entire processing stream is held up. A technique for avoiding this stall is to look further back in the processing stream for independent instructions. If instructions that may be executed out of order without creating additional hazards can be found then their execution may be promoted to fill the stalled execution cycles. In this way the data hazard causing instructions are delayed until the hazard is cleared but the progress of the execution stream is not stalled.

Dynamic scheduling involves the addition of specialized hardware to an embedded processor that detects and mitigates hazards. A dynamic scheduler can mitigate hazards using the same techniques applied to static scheduling. These techniques include code promotion, loop reorganization, and resource redistribution. Dynamic scheduling requires a hardware-based system to detect active functional units and data. This management unit then only issues instructions that do not depend on any active data or functional units, and is often called a dispatch unit. Additionally a functional unit must be included which tracks when instructions finish and relinquish resources. This unit is often called a retirement unit. All of the specialized hardware involved in dynamic scheduling takes up precious silicon resources that could be applied to further parallelism. Additionally it requires that the complex techniques involved in safely rescheduling instruction flow must be implemented in hardware and cannot be updated in the field. Since the technical challenges of dynamic scheduling are great it became common practice to boost the performance of dynamic scheduling by using static parallel code optimization, that is performed either by a separate post-pass code optimization that follows a traditional compiler, or by enhancing the traditional compiler with parallel code optimization.

Traditional compilers speed up sequential execution and reduce the required memory space mainly by eliminating redundant operations. In case that the code scheduler usually follows the traditional sequential optimizer in the back-end-part before register allocation and subsequent code generation this type of

code scheduling is called pre-pass-scheduling. The other approach is to use a traditional optimizing compiler and carry out code afterwards, that is called post-pass scheduling.

Code scheduling can be performed on three different levels:

- Basic block scheduling
- Loop scheduling
- Global scheduling

where basic block scheduling is the simplest but least effective code scheduling technique. Here, only instructions within a basic block are eligible for recording. As a consequence, the achievable speed-up is limited by both data and control dependencies.

The next level of code scheduling is loop-level scheduling, where instructions belonging to consecutive iterations of a loop can usually be overlapped, resulting in considerable speed-up. However, recurrences may impede speed-up.

The most effective way to schedule is at the highest possible level, called global code scheduling, where parallelism is sought and extracted beyond basic blocks and simple loops, in constructs called compound program, involving loops as well as conditional control constructs. In case of a chunk of code larger than a basic block, there are many possibilities for code scheduling.

In general, responsiveness of embedded computing systems can be introduced as some kind of process scheduling, that is based on three basic concepts:

- Process states
- State transition diagram
- Scheduling policy

As far as process states are concerned, there are three basic states connected with scheduling:

- Ready-to-run state, meaning processes are able to run when a processor is allocated for them
- Running state, meaning execution on the allocated processor
- Wait or blocked state, meaning processes are suspended or blocked waiting for the occurrence of some event before getting ready to run again

Possible state transitions and their conditions are stated in the state transition diagram, shown in Fig. 5.19. When the scheduler selects a process for execution, its state changed from ready-to-run to running. The process remains in this state until one of the following three events occur:

- Depending on the scheduling policy the scheduler decides to cease execution of the process and puts it into the ready-to-run queue again, changing its state accordingly
- The process in execution may issue an instruction that causes this process to wait until an event takes place, meaning the process state is changed to the wait state
- If the process reaches the end of execution, it terminates

Finally, the scheduling policy component specifies rules for managing multiple competing processes in that way selecting the next thread to run, based on scheduling algorithms, such as run-to-completion, shortest-job-first, earliest-deadline-first, etc. The term thread is another name for a task. This term is more common in operating systems that support processes, while a task is simply a thread in a single-process system.

RECONFIGURABLE EMBEDDED COMPUTING SYSTEMS PLATFORMS

Dynamically reprogrammable field programmable gate arrays or other techniques to implement hardware that varies in form and function over time representing the platform on which reconfigurable embedded computing systems can be build up. In a sense, FPGAs can be thought of as a hardware computing system on which one set of computing primitives might execute for a while, then another set, etc. Henceforth, an array of programmable logic devices, such as FPGAs, can be used as an execution platform for one or more hardware objects, e.g. a reconfigurable processing unit. The novelty component of the FPGA is its large internal configuration memory, and the two possible modes of operation, the download mode and the configuration mode.

For the implementation of reconfigurable embedded computing systems, based on FPGAs, one simply connects together in a regular mesh, $n \times m$ identical programmable logic blocks, that are cell based devices with many simple logic elements, as fundamental basis for the user dependent configuration, to build up complex logic functionality based on the concept of modularity. The interconnections possible with a FPGA are huge but the realized circuits speed depends on the used place and route algorithms. Therefore, the time dependent behavior is not predictable. A FPGA is a device with which the final logical structure can directly be configured by the user, without the use of an integrated circuit fabrication facility, because the user only needs some simple electrical equipment. The fast innovation cycles in the development of FPGAs has gained opportunities which are not possible with other technologies. Because the programming of the FPGA can be changed very fast, without rewiring or refabrication, its only reconfiguration.

The conceptual structure of a typical FPGA consists of a two-dimensional array of configurable logic blocks (CLB) that can be connected by general interconnection resources. The interconnection comprises segments of wire, where the segments may be of various lengths. At present interconnections are programmable switches that serve to connect the CLB's to the wire segments, or one wire segment to another. Logic circuits are implemented in the FPGA by partitioning the logic into individual logic blocks and then interconnecting the blocks as required via the switches.

The structure and the content of a logic block represents the logic block architecture. Logic block architecture can be designed in several ways like static RAM cells (SRAM), anti-fuse technology, EPROM and EEPROM technology. Some FPGA logic blocks are as simple as NAND Gates, other blocks have a more complex structure, such as multiplexers or lookup tables (LUT). Consider the basic elements of the CLB are small SRAM cells (LUT) that can be used implementing any logical function, one can distinguish between three different configuration types:

- Configuration type 1, that consists of an array of discrete LUTs, each of which depending on a set of input variables x

$$f = f_i(x_i); \text{ where } 1 < i < k$$

- Configuration type 2, that can be introduced being a two level LUT structure. The first level is identical to the configuration type 1, but, the outputs of the first level are combined with the second level. The second level logic can be arbitrary or restricted to some specific Boolean functions. Assuming the first level outputs as $g_i = g_i(x_i)$, the second level output can be defined as follows:

$$f(x_1, x_2, \dots, x_k) = g_1(x_1) \circ g_2(x_2) \circ \dots \circ g_3(x_3);$$

$$\text{where } \circ \in \{\Delta, \nabla, \oplus, \equiv, \dots\}$$

- Configuration type 3, that can be introduced as a structure with two LUT's sharing input variables. Additionally to an arbitrary number of common variables x_2 each LUT might depend on a set of further disjunctive inputs x_1 and x_3 . Therefore, the output can be written as follows:

$$f_{pair}(x) = [f_a(x_1 * x_2); f_b(x_2 * x_3)];$$

$$\text{where } x = x_1 \cup x_2 \cup x_3 \text{ and } x_2 \cap x_3$$

Over the last few years, a number of different types of FPGAs have been launched into the market, each of which has unique features, that can be classified into the following four categories:

- Symmetrical array FPGA
- Row based FPGA
- Hierarchical FPGA
- Sea-of-gates FPGA

EMBEDDED PROCESSOR KERNEL DESIGN FLOW

The goal of this part is to present a method for implementing a processor kernel on an FPGA. We restrict ourselves on the design of the ALU as one kernel part of the CPU to show how well-known or specialized operations may be performed on an FPGA.

The description of a processor kernel circuit can be entered using a schematic capture program with a simulation tool box like OrCAD, View-Logic, Synario etc. This involves using a graphical interface to interconnect circuit blocks. The available building blocks are taken from a component library.

An alternative way specifying an embedded processor kernel circuits is to use Boolean expression or State Machine language. With this methods, no graphical interface is needed. After the processor kernel circuit has been fully designed and merged into a circuit, it is translated into a special format that is understood by the used CAD tool, e.g. as netlist format. The partitioner partitions the circuit into logic cells of the selected FPGA which means a technology mapping which converts the processor kernel circuit, which is a netlist of basic logic gates, into a netlist of specific FPGA logic cells.

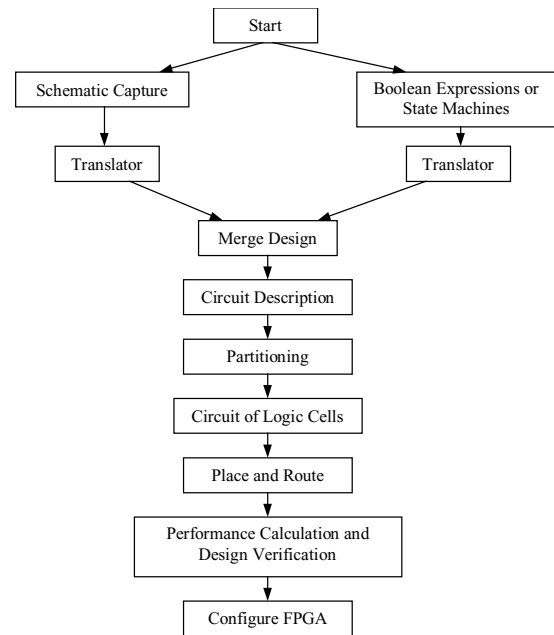


Figure 1: Design Flow for FPGAs

Placement means, each logic cell generated during the partitioning step is assigned to a specific location inside the IC, This is automatically done by CAD tools, or manually by the user in order to optimize the fitting. location in the FPGA. Automated placement is done using simulated annealing algorithms. After placement, the required interconnections among the logic cells must be realized by selecting wire segments and routing switches within the FPGAs interconnection resources.

Once the circuit is routed, the physical paths of all signals within the FPGA are known. Hence it is possible checking the performance of the implementation, which can be done either by downloading the configuration bits into the FPGA and checking the part within its circuit board, which will be a non effective method when designing complex kernels, or, which is the better way, by using a simulation tool box with timing analysis. If the performance or functionality of the circuit is not acceptable, shown by simulation, it will be necessary to modify the design at some point in the design flow in order to optimize timing and functionality by simulation runs with changed parameter sets.

All the descriptions above show how to implement a well-defined processor into any kind of FPL, but using FPLs it would be possible to change the processor kernel itself for obtaining best performance using this system design. This leads to Application-Specific Processor Design (ASP), and it should be clearly understood that inside this reported area of Hardware/Software Co-Design there will be always several level of simulation with mutual dependance. Fig. 2 outlines this situation.

Level 1 describes the simulation on the circuit level using a field-programmable device. This simulation may be performed before or after routing phase and will be responsible for correct functionality during real runtime. Level 1 may be omitted when using CPLDs with guaranteed timing behaviour through the whole device and will then be substituted by using computed running times and clock rates.

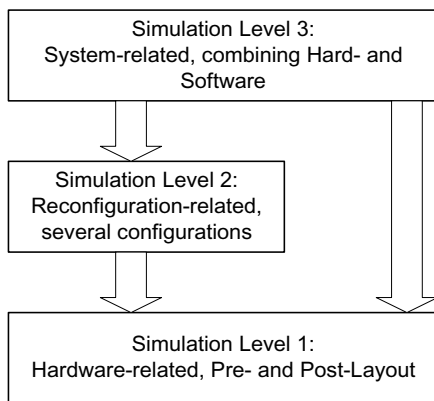


Figure 2: Levels of Simulation for a Mixed Processor/Software Design

On the other side level 3 is a system simulation level. The device, e.g. the processor kernel, will be modelled on a behavioural level for several reasons:

- Simulation on behavioural level saves (simulation) runtime
- Modelling on behavioural level offers a first well-defined device definition and will be often the first existing description.

As a result the runtime behaviour of the whole system will be simulated (and defined) inside this level.

Therefore the system designer has to describe the system knowing many details of level 1.

Situation is getting very complex when level 2 is no longer empty. This level integrates runtime reconfigurability and has to take in account that reconfiguration will cost time. It will be very difficult to obtain any analytical cost function which will compute the complete system runtime including hardware runtime, reconfiguration times (for several possibilities) and number of instructions needed to solve a given problem – and this under all runtime conditions. System simulation with levels 2 and 3 and at least knowledge of the behaviour in level 1 will be the only possible way of obtaining quantitative results for the system behaviour.

The design of a reconfigurable FPGA based CPU kernel as part of an embedded computing system design is assumed being based on the following assumptions.

- CPU kernel, as shown in Fig. 3
- Logical Operations such as
 - AND ACC, D
 - XOR ACC, D
 - NOT ACC
 - NAND ACC, D
 - INC ACC
 - OR
 - DEC ACC
 - NOR ACC, D
- Aruthmetical Operations
- VHDL description, as shown in Figure 4.

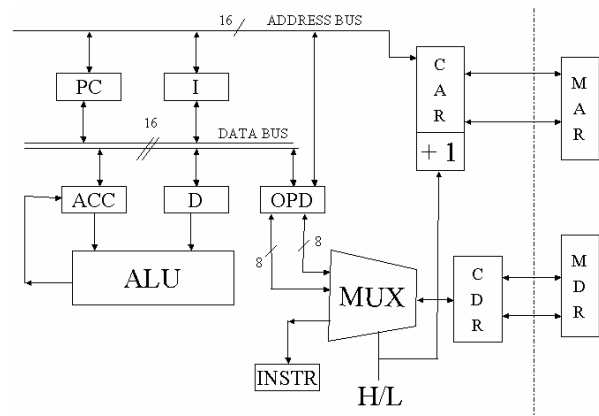


Figure 3: Block diagram of the processor kernel

The implementation of the FPGA based processor kernel may be restricted due to pin limitations by the IOB count or by complexity due to the CLB count of the FPGA chosen. In these cases another FPGA series should be chosen or the respective module has to be partitioned into smaller parts, each of them fitting the given constraints. Partitioning optimize the logical structure of the CLA. It could be seen that the number of internal signals between the different levels is twice the number of outputs. Hence, horizontal cut lines are not suitable for partitioning since in most cases pin limitation is more restrictive than complexity.

```

Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

Entity CSMP02 is
-- THIS SECTION OF THE VHDL CODE DESCRIBES
-- THE EXTERNAL PINS OF THE MICROPROCESSOR
End CSMP02;

Architecture CSMP02_main of CSMP02 is
-- DESCRIBES THE BEHAVIOR OF THE MICROPROCESSOR
-- DECLARATION OF CONTROL UNIT STATES
-- DECLARATION OF MEMORY UNIT STATES
-- DECLARATION OF INTERNAL SIGNALS AND REGISTERS
-- INITIALIZATION OF SIGNALS AND REGISTERS
-- CONTROL UNIT STATE MACHINE
-- RESET, FETCH, DECODE AND EXECUTE.
-- MEMORY UNIT STATE MACHINE.
-- INITIALIZATION OF I/O'S
End CSMP02_main ;

```

Figure 4: VHDL Diagram of the Processor Kernel

One of the most exciting novelties of FPGAs move beyond the implementation of embedded computing systems and instead harness large numbers of FPGAs as general-purpose computation medium.

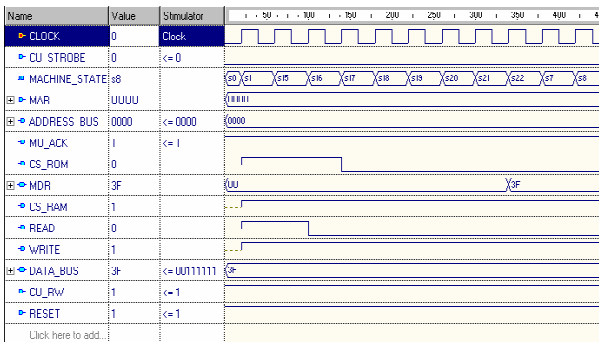


Figure 6: Microprocessor timing diagram

The embedded computing system architecture mapped into the FPGA needs not only be described based on standard hardware equations but can even be operated from general computational algorithms. While these FPGA based customized computing machines may not challenge the performance of embedded computing systems for all application areas, for computations of the right form, FPGA based embedded computing systems can offer extremely high performance, surpassing any other programmable solution. Although a customized hardware approach beats the power of any other generic programmable system, and thus there must always be a faster solution.

An FPGA based embedded computing system, that can be reprogrammed like a standard PC, offers the highest realizable performance for many different applications. In a sense, it is a hardware supercomputer, surpassing traditional machine architectures for certain applications. Because of their reprogrammability and cellular arrangement, FPGA's can process large data streams fast and simultaneous, providing a huge amount of parallel computing capability, a need if not a must in real time embedded computing systems applications.

REFERENCES

Berger, A. S. 2002. "Embedded Systems Design" CMP Books, Berkely, 2002.

Brown, S.D., Francis, R.J., Rose, J., Vranesic, Z.G. "Field-Programmable Gate Arrays". Kluwer Academic Publishers, Boston, 1992

Hartenstein, R.W., Glesner, M. (Eds.) "Field-Programmable Logic and Applications". *Lecture Notes in Computer Science Vol. 1142*, Springer Verlag, Berlin, 1996

Marwedel, P. 2003. "Embedded System Design". Kluwer Academic Publ., Boston, 2003.

Möller, D. P. F. 2003. "Embedded Systems Processor Kernel Design". In: 5th Internat. Conf. Computer Simulation and AI, pp. 11-18, Ed.: S. Raczynski, Universidad Panamericana Publ. Mexiko, 2000

Möller, D.P.F., Siemers C. "Simulation of an Embedded Processor Kernel Design on SRAM based FPGA" In: *Proceed. 31st Summer Computer Simulation Conference*, pp 633-638, Eds.: M. S. Obaidat, A. Nisanci, B. Sadoun, SCS Press San Diego, 1999

Möller, D. P. F., Siemers, C., Roth S. 1998. "A Concept for New Architecture Paradigm for Hardware/Software Co-Design". In: *Computer Architecture 98*, pp. 179-180. Ed.: J. Morris, Springer Publ. Singapore, 1998

Schmitz, M. T., Al-Hashimi, B., M., Eles, P. 2004. "System-Level Design Techniques for Energy-Efficient Embedded Systems". Kluwer Academic Publ. Boston, 2004.

AUTHORS



DIETMAR P. F. MÖLLER was born in Preetz, Germany. He enrolled at the Universities of Lübeck, Bremen Mainz and Bonn, where he studied Electrical Engineering and Human Medicine. He obtained his doctoral degree in 1980. From 1985 to 1991 Dr. Möller lead the [anaesthesia](#) division of Dräger AG in Lübeck, Germany. 1991 he has been elected as Full Professor for Computer Engineering (TI) at the Technical University of Clausthal. In 1998 he has been elected as Full Professor for Computer Engineering Systems (AB TIS) at the University of Hamburg where he holds the chair for Computer Engineering. Since 1998 he also is Adjunct Professor at the California State University Chico. His E-Mail address is dmoeller@informatik.uni-hamburg.de and his Web-page can be found at <http://www.informatik.uni-hamburg.de/TIS/>