REFLIX: A Processor Core with Native Support for Control-Dominated Embedded Applications

Zoran Salcic, Partha Roop, Morteza Biglari-Abhari, Abbas Bigdeli
Department of Electrical and Electronic Engineering, University of Auckland, Private Bag 92019, Auckland, New Zealand
e-mail: {z.salcic, p.roop, m.abhari, a.bigdeli}@auckland.ac.nz

Abstract: Efficient and reliable interaction with the environment (reactivity) is a key feature for many embedded system applications. Current implementation technologies that include standard microprocessors and microcontrollers, or fully customized systems, are not ideally suited to such reactive tasks either in terms of their performance constraints or in terms of design implementation and programming. We propose a microproc essor architecture that has native Esterel-like support for reactivity, flexibility of using programs and design styles as used in Esterel programming language for reactive embedded system implementation and provides time-predictable behaviors in reaction to external events. The new processor, called REFLIX, is built around already existing processor core and exploits its flexibility in allowing customization at much higher level than usual microprocessor cores. REFLIX shows manifold improvement in speed and memory footprint in dominantly reactive applications compared to the traditional microprocessors.

Keywords: reactive systems, processor core, embedded systems, FPGA

1 Introduction

Embedded systems most often have a dedicated microprocessor or a microcontroller that executes a non-terminating control program, which controls its environment. The environment constitutes of a set of sensors and actuators, which the microprocessor controls. The control program repeatedly determines the status of the environment (by checking the status of the sensors and actuators) and then reacts based on the current status (hence embedded systems are often called *reactive systems)* [1], [2].

The environment status can be determined either by polling (which checks for the presence of certain signals routinely) or by using interrupt mechanism (which is like an alert mechanism when certain signals occur in the environment). Polling is also known as *busy waiting* since CPU cycles are wasted while checking for the presence of signals in environment. Interrupts avoid busy waiting but have context switching overhead since the occurrence of an interrupt requires the execution of specific code (called an interrupt service routine) leading to a change in the standard control flow of the program. Hence, the context of program execution needs to be saved prior to branching for interrupt handling and has to be restored after interrupt handling is completed. Such context-switching overhead can be considerable in an embedded system where environment interaction is a key. Moreover, as interrupt handling is executed concurrently with the main control task, there is a danger of inconsistent system behavior due to mishandling of common resources (data). Also, different events often have different level of importance and priority-based interrupt schemes have to be used.

While context switching is necessary for data dominated tasks, where return to main program is important, it is not crucial for many control-dominated tasks as it is illustrated further in this paper. Moreover, priority signal handling is done using either software means (additional polling through daisy-chaining) or hardware means (using programmable peripheral devices or daisy chaining using external hardware) both of which are quite inefficient and inelegant ways to handle priorities in control dominated tasks.

This paper describes a novel processor core, called REFLIX, which is aimed at reactive embedded applications. REFLIX provides a primitive set of features and instructions suited to reactive systems in addition to a set of standard set of instructions found in common microprocessors and generic instructions to control external hardware functional units. The proposed approach provides mechanism to avoid busy waiting associated with polling, when required, and context switching associated with interrupts for control dominated tasks. The environment interaction model of a reactive programming language called Esterel [3] inspires this mechanism, but does not follow fully the Esterel semantics. The major contributions presented in the paper are the following:

a) Support for reactivity through a set of native instructions is incorporated in REFLIX, which are lacking in previous architectures. All instructions, including those that look like conventional instructions for polling, perform in the same time tick that is equal to 4 machine cycles contributing to both efficiency and predictability of program execution.
b) Support for preemption and priority resolution based on external events using a new native instruction called ABORT, which can be nested to achieve priorities of external events. This is an extremely important feature for implementation of control-dominated real-time tasks.
c) Reusability of the core for different embedded applications and their variations is achieved primarily by change of an application program; hence efficient implementation of control-driven software with the ability for compiling new Esterel-like specifications to REFLIX instructions directly. In case of need a further hardware customization can be achieved using parameterized nature of the processor core description.

The paper is organized as follows. Sections 2 and 3 give an overview of the related work and explain the background and motivation of REFLIX design and some of the major features that support reactivity. In section 4, we introduce the major REFLIX features with the emphasis on those that support reactivity. Section 5 gives a flavor for REFLIX programming when using the reactive instructions. In section 6 we describe the REFLIX data path and control unit together with some implementation details. Section 7 gives some further performance comparison using a set of application benchmarks. REFLIX is first compared to original FLIX core and demonstrates manifold speed-up in dominantly reactive tasks. The same benchmarks are used to demonstrate REFLIX's low memory footprint compared to footprint of standard microprocessors. Section 8 presents some concluding remarks related to the current implementation and limitations of the current design and the scope for future work.

2 Related Work

One of the trends in implementation of embedded systems, such as cell phones, medical appliances, home appliances, and similar applications, is to rely on application-specific processors that better match requirements of those applications than general-purpose instruction processors [4]. There are several approaches suggested or used for customization of those processors. Some of them rely on using existing architectures, such as those from ARM or MIPS [5, 6, 7]. Standard fixed processor cores are connected to programmable logic to implement additional instructions and functions. Some of the solutions employ parameterized processor cores that are customized at the time of their compilation/synthesis for FPGAs, such as Altera NiOS processor [5], or in run-time during system operation [8, 9]. Some other processor cores [10,11] provide generic mechanisms for new instruction implementation. These instructions are executed in functional units external to the processor core and are readily supported by software. A further step towards generalization has been proposed in [12], where a number of processor "templates" is used to provide a framework for different customization strategies. All above processors have general-purpose RISC-type architecture with more or less typical instruction sets common to RISC-type processors. None of those processors addresses aspects of reactive applications by supporting generic mechanisms for reactivity and preemption beyond usual interrupt structures found in conventional processors.

3 Background and Motivation

REFLIX processor operation is inspired by Esterel, which is a synchronous reactive programming language that provides a neat set of constructs for modeling, verification and synthesis of reactive systems. Esterel language has been used in the past for specification and verification of processors [13], generation of hardware circuits [14] and for rapid system prototyping [15]. The environment of any Esterel program consists of a set of sensors and signals, which can be modeled abstractly using constructs available in the language. The activation clock of the Esterel program is a predefined event called the *tick* event. During every tick the Esterel kernel samples its environment and performs a set of *instantaneous* reactions based on the values present in its environment during the present tick. The main constructs for interacting with the environment are *await* (which is a delay construct), *emit* (which performs signal emissions to the environment), *sustain* (which sustains a signal forever), *abort* (which is preemption construct), and *trap* (which is similar to software interrupts). Esterel is also a concurrent language and its model of concurrency is known as synchronous broadcast which means that input and the corresponding output both occur at the same instant (*tick*) and also an event generated in any concurrent module is instantaneously broadcasted to all other concurrent modules. In addition to such constructs for control flow Esterel supports data handling through a suitable host language such as C or Java. Data handling can be either performed synchronously (consumes no time) by performing procedure calls (which are defined in the host language) or asynchronously (takes time) using tasks (which are also defined in the host language).

REFLIX is a processor core designed to follow main ideas of Esterel is environment interaction model. REFLIX provides a set of native instructions suitable for reactive systems in addition to providing standard instructions for data processing. These include native facilities for *delay*, *signal emission*, *priorities*, *preemption* and *task execution* facility using functional units. The main unsupported Esterel feature is concurrency (parallel execution), which must be implemented in a similar way as it has been done when Esterel is compiled for execution on standard microprocessors. In order to make its implementation easy we used our customizable FLIX core [10,11] as the basis for the REFLIX design.

By adopting and supporting Esterel-like model for reactivity on machine instruction level, we achieve two major goals:

a) the same processor core can be used to implement different reactive algorithms for different applications by changing only programs and not processor hardware and
b) preserve performance predictability by guaranteeing execution times for all primitive instructions.

In this way we provide a generic platform for implementation of a large class of embedded applications, which would otherwise be implemented either by separately synthesized hardware (usually finite state machine - FSM) or by software means that can be implemented on standard microprocessors but with many difficulties.

Original FLIX processor supports customization by allowing the designer to add new instructions or resources to the datapath by implementing new functional units (FUs). This feature is very convenient for designing embedded applications as the core can be customized based on the application requirements. In REFLIX, we have extended FLIX core by providing native instructions for preemption, priority, delay, suspension and signal emission while preserving its flexibility for customization of the data processing part. Some of the key features of the FLIX core have been preserved in order to support the notion of its local time and "time tick" by executing all native instructions in equal time that corresponds to one "system tick".

A qualitative comparison of different implementation strategies of embedded applications and our motivation for REFLIX development are summarized in Table 1. Reactive features in embedded systems may be mapped into finite state machines (FSMs) and subsequently synthesized as custom logic. Alternatively, a general purpose processor may be used to implement such reactive features using interrupts. The proposed approach is a novel intermediate approach that combines the efficiency of the FSMs and the flexibility of conventional processor.

Table 1 Qualitative comparison of embedded application implementations

Implementation approach	Advantages	Disadvantages
HDLs and hardware implementation	• Reactive behaviors mapped onto FSMs • Small footprint and cheap implementation • Supports real parallelism	• Not suitable for non-reactive parts • Each application and modification requires full synthesis
High-Level Programming Languages	• Good handling of non-reactive parts • Easy for modification • High abstraction level	• Large footprint (memory requirement) • Often requires RT OS and appropriate scheduling • Sow in reacting to events • Difficult to integrate non-standard interfaces • Complex compilation process • Complex context switching increases overhead • Emulates parallelisms by serialization of concurrent activities
Native Standard Microprocessor Assembly Languages	• Smaller footprint (memory requirements) • Ease of control of low-level details • Relatively easy for behavior modification • Low abstraction level • Easier to integrate non-standard interfaces • Faster reaction times	• Low abstraction level • Unsuitable reactive behaviors mechanisms • Difficult to integrate non-standard interfaces • Emulates parallelisms by serialization of concurrent activities • Often requires real-time operating system (RTOS) or kernel
REFLIX	• Small footprint • Specialized support for reactive situations • Fast reaction and response times • Customization at both hardware and software level • Ease of behavior modification • Generic support for hardware implemented functional units • Easy to verify behaviors • Can support reactive HLLs	• Low abstraction level for data processing • Less flexible for algorithm modifications than HLLs • No support for true parallelism • Requires RT OS or kernel to support concurrency

4 REFLIX Core Features

In this section we introduce the main architectural features of the REFLIX processor and describe the main ideas that lead to its design. The current version of REFLIX has adopted original FLIX core for its base with removal of the interrupt structure and asynchronous event handling altogether. REFLIX preserves FLIX word length (16 bits) and instruction execution principles (4 machine cycles make one instruction cycle). Main departures and extensions to the original core that directly aim reactive applications are:

- Variable number of single-bit input sensor (Sin) and single-bit output sensor (Sout) lines. The number of these lines is a design parameter and can be instantiated for each specific application.
- Introduction of internal timers that generate user programmable timing (TimeOut) signals. Number of internal timers is customizable and is represented by a design parameter. TimeOut signals can be used for interaction with the environment or can be fed back to the REFLIX core itself and used for synchronization purposes.
- Introduction of ABORT mechanism for preemption, which is activated upon an event occurrence. Any piece of code can be wrapped up in the ABORT statement (abort body) and immediately abandoned in case that an external event on specified sensor input or timeout occurs. Using nested aborts it is possible to guarantee maximum one instruction cycle delay in response to a set of external events (including the time to resolve event priorities).
- Introduction of other instructions that support reactivity in the native instruction set.

These new features are primarily visible through the REFLIX programming model and its instruction set.

REFLIX core's external (interface) view is presented in Figure 1. In addition to the common and above mentioned ports, it also contains three specific ports:

- T, which provides information on the current processor machine cycle.

- IRBUS, which provides the access to currently executed instruction operation code and can be used by external hardware (e.g. to start operation of an external functional unit).
- EndFU, which provides feedback information on the status of external hardware (e.g. functional units) similar to [10].

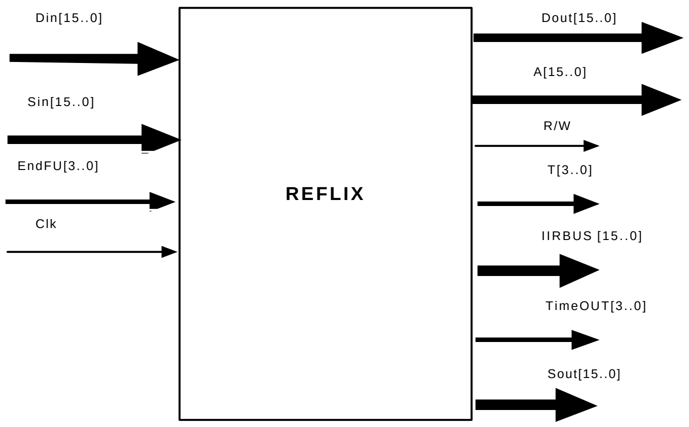

Figure 1 REFLIX external view

4.1 Native Reactive Instructions

The original FLIX instruction set is appended with new group of instructions that support reactive processing. There are seven basic instructions in the reactive category and they are presented in Table 2. Most of REFLIX instructions are only one word long, but some of the reactive instructions require two words for immediate operands or address information.

Table 2 REFLIX instructions supporting reactive processing

Feature	**Instruction syntax**	**Instruction Length (words)**	**Function/Description**
Preemption	**ABORT** *signal, address*	2	Preemption instruction. ABORT has a body up to the instruction whose *address* is indicated in the instruction (called continuation address since after preemption program continues from this address). *signal* can be either an external one or a *TimeOut* received from internal timer.
Signal emission	**EMIT** *signal*	1	The specified *signal* is set high for one instruction cycle
Signal sustenance	**SUSTAIN** *signal*	1	Specified signal is set high forever (indefinitely)
Signal polling	**SAWAIT** *signal*	1	Wait until the specified *signal* occurs
Delay	**TAWAIT** delay	2	Immediate delay— wait until specified time elapses (wait at least one system tick - time is expressed in the number of instruction cycles)
Conditional signal polling	**CAWAIT** *signal1, signal2, address*	2	Wait until either *signal1* or *signal 2* occurs. If signal1 occurs then execute instructions from the address following this instruction, else from specified *address*.
Signal presence	**PRESENT** *signal, address*	2	If *signal* is present the next instruction is fetched from the next consecutive address. Otherwise, it is fetched from the specified *address*.

Two instructions generate external signals (outputs), which can last one system tick (EMIT) or indefinitely (SUSTAIN). The next tree instructions, SAWAIT, TAWAIT and CAWAIT, provide waiting mechanism on events on external signals and timeouts (generated by internal timers) and are used for synchronization purpose. The duration of time to wait is under the explicit program control. PRESENT instruction provides a mechanism for conditional execution depending on the presence of the specified signal. Finally, ABORT is preemption and priority resolution instruction which is explained in detail in the next section.

4.2 Semantics and Implementation of Preemption Support

Native ABORT instruction is introduced to support preemption with priorities. In the current REFLIX prototype ABORT instruction can work with up to 16 different external input signals and up to four internal timers generated signals. ABORT instructions can be nested to support up to four levels of priorities. These numbers are obviously implementation dependent and can be changed as the design parameters for the REFLIX parameterized core.

An ABORT instruction is active from the instant it is executed until its entire body is executed or until an event on the signal occurs that preempts all unexecuted instructions within the body. Format of the instruction is as follows:

OPCODE (10)	Timer(2)/Signal(4)
Continuation-address (16)	

Two different operation codes are used for abort operations, one for an abort on an external signal and the other for an abort on a timer. The ABORT instruction is executed in two stages with the support of a dedicated hardware unit called the abort-handling block (AHB):

- **Abort activation**. It is executed immediately after fetching and decoding the ABORT instruction, when REFLIX starts monitoring change (activation) of the designated signal. Continuation address, from where the program will continue execution if preemption happens, is stored into the REFLIX abort handling block.
- **Abort termination**. Once the designated signal is activated, abort is taken and an unconditional jump to the continuation address is executed, or, if the continuation address is reached and the designated signal has not been activated, the abort is automatically terminated.

The abort handling block (AHB), which is a part of the REFLIX datapath, supports nesting and prioritizing of abort statements. The AHB contains active abort signal register (AASR) block with 4 registers with a length that equals to the number of input sensing signals, which can abort current program execution. Registers are used to store the code of the signal line that starts to be monitored for signal activation. Each signal line has a unique code generated using a one-hot encoding scheme (only one bit can have a value 1). The addresses of the AASR registers, 0 to 3, at the same time represent, in ascending order, priorities of signals that are monitored. The first executed ABORT instruction always stores the monitored signal code into AASR(0), next nested ABORT instruction stores its monitored signal code into AASR(1), and so on. Summary information on all currently monitored signals that can abort program sequence is stored in joint abort signal register (JASR). Its value is obtained by bit-wise OR-ing values of all AASRs:

$$JASR_i = AASR_i(0) + \ldots + AASR_i(3) \text{ for } i=0, 1, \ldots, 15$$

As JASR cannot preserve information on priorities of monitored signals, each AASR is associated with a single bit flag called abort flag (AF), and individual AF bits will be set if the corresponding AASR register (with the same address) is non-empty (with AF(0) being 1 for the highest priority monitored signal). The summary joint abort flag (JAF) contains information on the presence of monitored signals, or

$$JAF = AF(0) + AF(1) + AF(2) + AF(3)$$

REFLIX control unit determines an action path during instruction execution based on the value of the JAF bit as it is shown in section 6.

Another register block contains four active abort address registers (AAARs), which are used to store the continuation addresses of currently active abort instructions. The highest priority ABORT instruction’s (outermost one) continuation address is in AAAR(0), next lower priority continuation address is in AAAR(1) and so on. Signal input register (SIR) is used to capture (latch) activation of signals on individual input sensing lines. This information is used, together with the information on currently monitored signals, to identify the presence of pending (non-processed) abort events. For that purpose, another flag, called the pending abort event flag (PAEF) is introduced and used by REFLIX control unit to provide proper and immediate reaction when events on monitored signal lines occur. Its value is derived as

$$PAEF = (SIR_0 \; JASR_0) + \ldots + (SIR_{15} \; JASR_{15})$$

The abort termination stage is executed when a monitored event occurs, or when abort instruction reaches its continuation address without occurrence of event. Termination of an ABORT instruction causes also the termination of all other ABORT instruction nested within its body that are of the lower priority.

It should be noted that both JAF and PAEF flags are not programmer visible. However, they can be made such and used in new currently non-implemented instructions to enable a certain level of programmer's control over reactive core features.

Two pointers, called the abort read pointer (ARP) and the abort write pointer (AWP), are used to up-date addresses of registers within the AHB from which information will be read or written to. However, these pointers are not a part of the programming model as they are not user visible. They are used only by the control unit and can be considered as its part. They are effectively 2-bit (mod 4) counters, which are initialized to a value 0 on the system power-up or reset.

Other parts of the programming model include signal output register (SOR) with individually controllable/writtable bits, and pool of timers which appear as memory mapped registers with some programmable features. The level of their programmability is application dependent and can be customized by the selection of configuration (VHDL generics) parameters. Their meanings are more or less obvious and they are described further in the following section where we discuss REFLIX data path.

5 Programming Examples

In order to get a flavor of programming at a low level when reactive native instructions are available, in this section we consider a few simple code examples.

5.1 Pump Controller Example

Consider for example the following specification of a simple *pump controller* [8]:

A pump controller is used to control the operation of a pump inside a mine which may have high methane levels. The pump is used to pump out water (whenever the water level exceeds the desired level) provided the methane level is below the desired level (RIGHT-METHANE). Whenever, methane level goes above this desired level (NOT-RIGHT-METHANE), the controller must stop the pump and wait until right methane level is restored.

An implementation of this specification using REFLIX assembly language is as follows:

```
start1:
      ABORT NOT-RIGHT-METHANE ADDR
      #abort body
      loop:
            SAWAIT HIGH-WATER-LEVEL
            EMIT START-PUMP
            SAWAIT LOW-WATER-LEVEL
            EMIT STOP-PUMP
            JMP loop
      #end of abort body
      ADDR:
      #handle exception
      EMIT STOP-PUMP
      SAWAIT RIGHT-METHANE
      #return to resume normal operation
      JMP start1
```

The pump controller is implemented in two parts: normal behavior, which is enclosed by an ABORT statement (preemption mechanism in REFLIX), and exception behavior, which is provided from the continuation address of the ABORT (when the body of the ABORT is preempted, program control resumes from this address). The behavior of the ABORT body has an SAWAIT statement (for usual polling of a signal) that samples the water level. Whenever water level is high (HIGH-WATER-LEVEL triggers) signal START-PUMP is emitted, to start the pump. This water level is checked again till it reaches the low level (using another SAWAIT). Then the pump is immediately stopped using another EMIT statement.

This normal behavior is continued until enclosing ABORT triggers. ABORT triggers either when the signal NOT-RIGHT-METHANE occurs in the environment and the body has not finished execution or when the body terminates before this signal occurs. In this example, since the body is an infinite loop, it never terminates and ABORT triggers only when the signal NOT-RIGHT-METHANE occurs.

When ABORT triggers, current instruction in the body is completed and the next instruction is automatically fetched from the continuation address provided with ABORT (in this case ADDR). So, whenever, NOT-RIGHT-METHANE is detected the pump is immediately stopped (by an EMIT statement) and the controller waits for methane level to restore (RIGHT-METHANE is sensed using another SAWAIT) before resuming normal operation.

5.2 Extended Pump Controller Example - Comparison with a Conventional Microprocessor

In this section we consider how two of most important reactive features, priority and preemption, are supported in conventional microprocessors and how they compare with REFLIX. We have selected Intel 8051 microcontroller [16] to illustrate the ideas.

Preemption. Consider the pump controller example, presented in the previous section. now implemented using native instruction set of Intel 8051:

```
#setup interrupt vector
ORG addr
DD HighMethanLevel

      #wait for high water level
start:      MOV A, HIGH-WATER-LEVEL
loop1 CJNE  A, PX, LOOP1
      #water level high detect.start pump
      MOV Px, START-PUMP
      #wait for low water level
      MOV A, LOW-WATER-LEVEL
loop2 CJNE PX, A, loop2
      #water level low detected; stop pump
```

```
        MOV Px, STOP-PUMP
        LJMP start

        #Interrupt service routine (ISR)
        HighMethaneLevel
        #save registers to be used in ISR
        PUSH A
        #stop the pump
        MOV Px, STOP-PUMP
        #wait for right methane level
        MOV A, RIGHT-METHANE
loop    CJNE Px, A, loop
        #restore registers
        POP A
        #return from interrupt
        RETI
```

To handle preemption, an interrupt vector needs to be setup for handling high methane level within the mine. The main routine samples the water level and starts or stops the pump appropriately. The interrupt routine stops the pump when methane level is not right and waits until right level is detected before returning to the main program. Even ignoring the initialization and context-switching overhead, we have 13 instructions in 8051 compared to 9 instructions in REFLIX.

Priority. To handle priority conventional processors use either software mechanism (daisy chaining) or hardware mechanism (external peripheral). Software mechanism is very inefficient as it uses polling after interrupt to determine the highest priority device. The hardware mechanism use external peripherals such as programmable interrupt controller (Intel 8259 [17]) to resolve priorities and are quite inefficient. If, however, the control dominated task is such that after handling an exception it is undesirable to return to the main program, interrupt mechanism cannot be efficiently employed (as it forces the return to the interrupted program).

Let us now consider a slightly modified pump controller specification as given below [8]:

A pump controller is used to control the operation of a pump inside a mine which may have high methane levels. The pump is used to pump out water (whenever the water level exceeds the desired level) provided the methane level is below the desired level (RIGHT-METHANE). Whenever, methane level goes above this desired level (NOT-RIGHT-

METHANE), the controller must stop the pump and wait until right methane level is restored. If at any time, however, the methane level is too high (NOT-RIGHT-METHANE is only marginally high) then the pump must be stopped immediately and an ALARM must be generated. Pumping is stopped until right methane level is restored.

Note that in this specification there is a higher priority preemption condition triggered by HIGH-METHANE over NOT-RIGHT-METHANE. Priority is implemented in REFLIX using nesting of ABORTs with outer ABORTs having higher priority over inner ABORTs. Let us consider the following implementation of this specification using REFLIX instructions:

```
start:
ABORT HIGH-METHANE ADDR1
start1:
      ABORT NOT-RIGHT-METHANE ADD1
             #abort body
             loop:
             SAWAIT HIGH-WATER-LEVEL
             EMIT START-PUMP
             SAWAIT LOW-WATER-LEVEL
             EMIT STOP-PUMP
             JMP loop
      #end of abort body
      ADDR:
      #handle exception
            EMIT STOP-PUMP
            SAWAIT RIGHT-METHANE
            # resume normal operation
            JMP start1
ADD1:
#handle high methane
EMIT STOP-PUMP
EMIT ALARM
SAWAIT RIGHT-METHANE
JUMP start
```

In this implementation, HIGH-METHANE has higher priority over NOT-RIGHT-METHANE. Hence, whenever HIGH-METHANE is detected, the pumping is stopped and alarm is generated (if HIGH-METHANE and NOT-RIGHT-METHANE occur in the same instant, the inner ABORT will be ignored and the outer ABORT will trigger). This example illustrates the simplicity of priority handling in REFLIX. In a conventional processor implementation of this would require several initialization steps (either to setup a programmable interrupt controller hardware or to setup vector addresses in a processor that supports priority interrupt structure). ABORT provides an elegant mechanism to incorporate priorities in control-dominated tasks using simple nesting of aborts. Also, the overhead associated with context switching is completely eliminated, since this is not a requirement for the task involved.

6 REFLIX Architecture and Implementation

The REFLIX data path with emphasized differences to the original FLIX data path [11] is shown in Figure 2. The data path is organized around two internal buses, called ABUS and DBUS, which are used for transfers of address and data information between internal registers, respectively, and enable to carry two register transfers, between two pairs of registers, at the same time (machine cycle). The abort handling block (AHB) is shown with the shaded background in Figure 2.

One of the major issues in the overall REFLIX design was to fit it within the basic and very simple FLIX framework and to preserve some of the original core features, which have been found useful in a number of customization projects. The instruction cycle is one of those features, which permits each of the instructions to be completely executed in four machine cycles. This leads to an easy maintenance of time, both the REFLIX global (absolute) time and individual timing relationships, especially those which use locally generated relative times and timing based events. Both the global clock and locally generated non-overlapping four clock phases are available to external logic to drive external circuits (including FUs).

A conceptual REFLIX instruction execution cycle, which also depicts control unit operation, is shown in Figure 3. Upon power-up or reset REFLIX goes through the initialization phase to set initial conditions for all registers. Then, it enters instruction execution cycles in which priority is given to the handling of preemptions that have been awaited for by previously executed ABORT instructions. All events on monitored signals (ABORT triggers) recorded during one instruction cycle will be given attention in the next instruction cycle (tick of time). In case of the absence of pending events, the control unit performs two actions:

- Next instruction is fetched from memory and executed. Although there is no special difference between instructions, the ABORT instruction is emphasized in diagram of Figure 3, as it carries operations on the registers of the AHB.
- Non-preemptive termination of ABORT instruction, if any, is performed in parallel with instruction fetching and execution.

For REFLIX implementation and prototyping we have used field-programmable logic devices (FPLDs). There two major reasons for this: (a) FPLDs provide an ideal prototyping environment with very fast turn-around time between two versions of the design, and (b) with the appearance of huge FPLDs with millions of usable equivalent gates, it becomes feasible to build whole systems on programmable chips (SoPC).

In addition, FPLDs are accompanied by advanced design tools and HDLs that enable parameterization of designs, which can be relatively easily customized for specific applications. This aspect of the REFLIX design has not been emphasized in the paper. The first implementation is used as a proof of concept and incorporates only parameterization of some of the resources, such as number of timers, number of sensing input signals, number of output signals and number of priority levels of external events (depth of nesting of ABORTs).

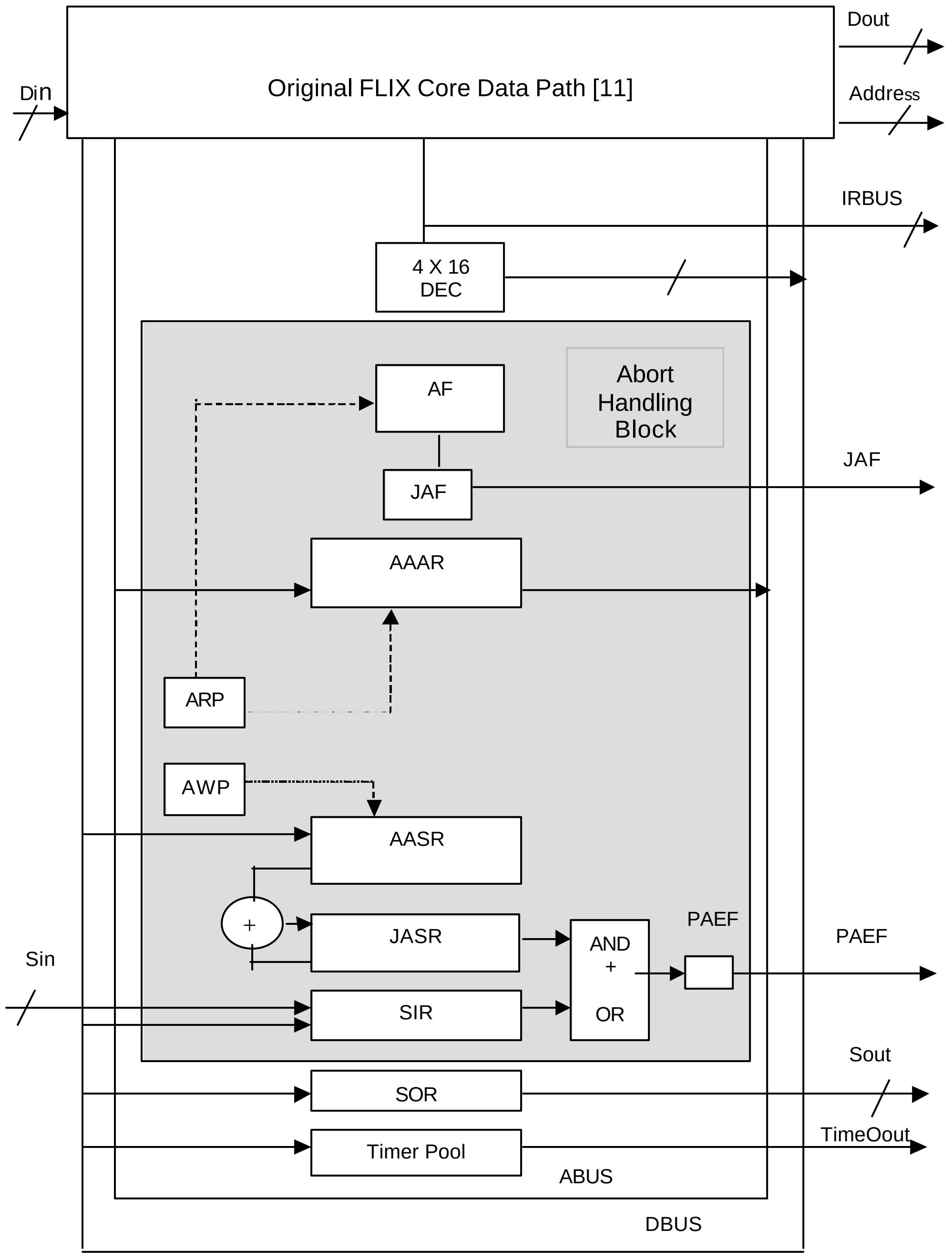

Figure 2 REFLIX data path

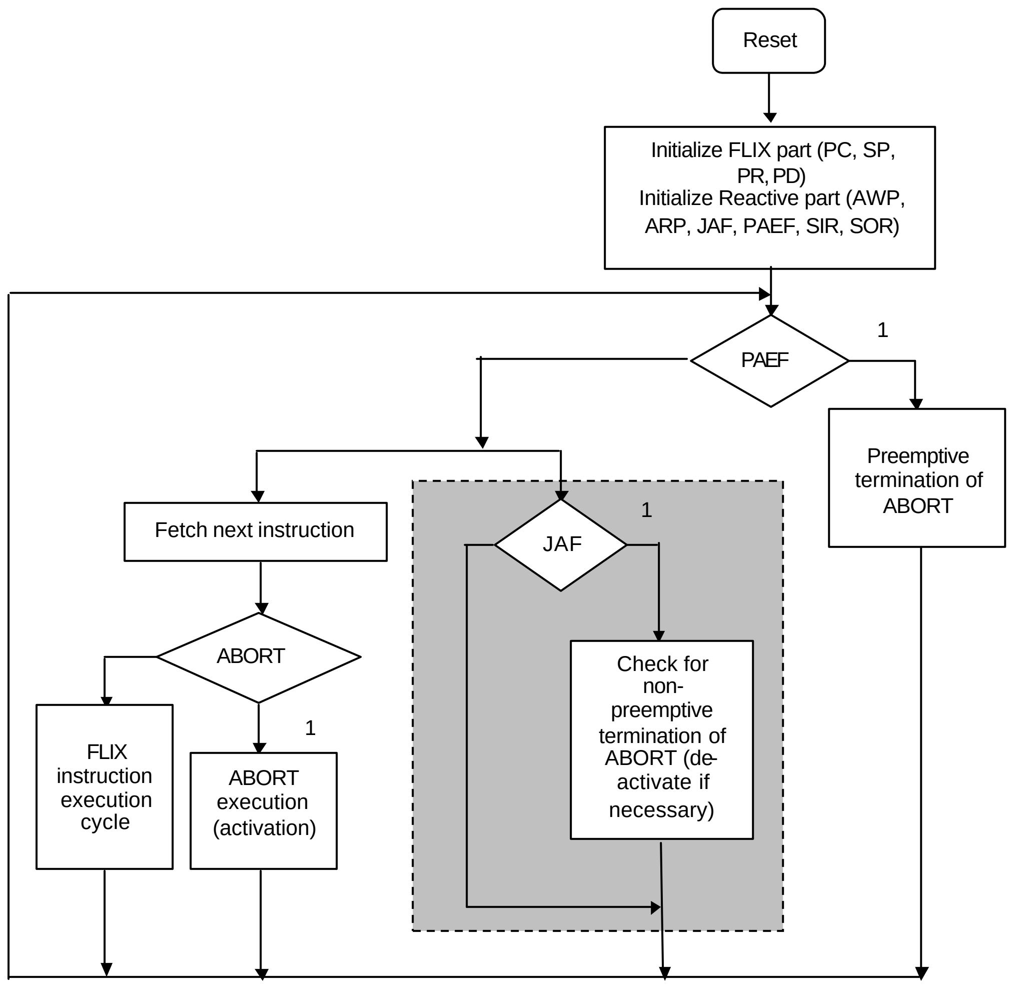
Reset
Initialize FLIX part (PC, SP, PR, PD)
Initialize Reactive part (AWP, ARP, JAF, PAEF, SIR, SOR)
PAEF
1
Preemptive termination of ABORT
Fetch next instruction
JAF
1
ABORT
Check for non-preemptive termination of ABORT (de-activate if necessary)
1
FLIX instruction execution cycle
ABORT execution (activation)

Figure 3 REFLIX control unit conceptual operation

7. Implementation of the ABORT Mechanism

The abort mechanism consists of two-stage process; abort activation performed by ABORT instruction execution, and abort termination, performed when preemption occurs or by natural expiration of the need for an event monitoring.

7.1 ABORT instruction execution

Activation of the abort mechanism and start of monitoring of specific signal is initiated explicitly by programmer using the ABORT instruction. The ABORT instruction execution is similar to all other instructions. Once ABORT instruction is fetched and decoded, it is executed in two machine cycles (T2 and T3) as illustrated in Figure 4.

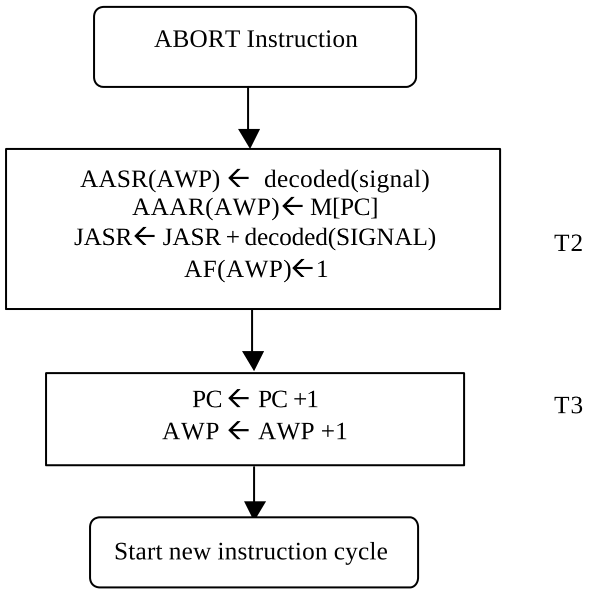
ABORT Instruction
AASR(AWP) ← decoded(signal)
AAAR(AWP)← M[PC]
JASR← JASR + decoded(SIGNAL)
AF(AWP)←1
T2
PC ← PC +1
AWP ← AWP +1
T3
Start new instruction cycle

Figure 4 ABORT instruction execution

First, the one-hot-encoded code of the signal that will be monitored is stored to the next location in AASR and continuation address is transferred from memory (second instruction word) to the next location in AAAR. The JASR is updated to record all monitored signals, and corresponding abort flag is set. In the next cycle (T4), the values of program counter and AWP pointer are updated.

7.2 Preemptive abort termination

The algorithm for preemptive termination of abort instruction is presented in Figure 5. The algorithm first checks the values of abort flags in the order of their priority. The first active one corresponding to a pending ABORT signal that triggers will be taken. Program execution continues from the corresponding continuation address. Abort write and read pointers are updated accordingly and abort flags reinitialized. For example, if an event with the highest priority has occurred, its index i is set equal to 0, and all abort flags are cleared. In case of any other event, all abort flags with equal or the same priority will be cleared, and those with higher priority will be left intact.

```
FOR i = 0 TO 3

IF AF(i) = 1 THEN
        ; Check event occurrence
        IF AASR(i) AND SIR <> 0 THEN
                PC ← AAAR(i); continuation address into PC
                AWP ← i; update AWP and ARP
                ARP ← i;
                FOR k=3TO i
                        AF(k) ← 0; reinitialize abort flags
                NEXT k;
                CLEAR SIR; clear signal input register
                EXIT FOR;
        END IF;
END IF;

NEXT i;
```

Figure 5 Preemptive abort termination

7.3 Non-preemptive abort termination

A non-preemptive abort termination happens whenever program execution reaches any continuation address. The only action, which has to be carried out, in this case is to deactivate corresponding abort and update all internal registers (AF, AWP and ARP). This operation can be performed in parallel with any other instruction execution except with the ABORT instruction itself. For this purpose, a separate finite state machine (FSM) checks the value of the joint abort flag (JAF). In case the JAF is set, the FSM checks whether any of the ABORT instructions has to be deactivated (naturally terminated) and up-dates appropriate registers within AHB. This FSM can be considered as a part of the AHB itself. The algorithm implemented within the FSM is shown in Figure 6.

```
TEMP ← PC;
IF JAF = 1 THEN
        FOR i = 0 to 3

                IF TEMP = AAAR(i) THEN

                        AWP ← i; update AWP and ARP
                        ARP ← i;
                        FOR k=3TO i
                                AF(k) ← 0; reinitialize abort flags
                        NEXT k;
                        EXIT FOR;

                END IF;

        NEXT i;
END IF;
```

Figure 6 Non-preemptive abort termination

The algorithm starts at the beginning of each instruction cycle. The current instruction address (program counter) must be temporarily saved in the TEMP register as it has to be compared with the continuation addresses stored in the active abort address registers. The value of the program counter itself will be changed by execution of the current instruction. If the values of TEMP and any of AAARs match, corresponding abort must be deactivated by up-dating values of appropriate registers.

8 Results of Performance Comparison

In this section, we present comparisons of REFLIX core to some similar standard microprocessors. As the benchmarks we used the following typical control dominated applications, initially written in Esterel and subsequently mapped to a number of standard microprocessors:

- Transmission control protocol (TCP) transmitter [18]: TCP is a connection-oriented protocol used in the TCP/IP which uses three way handshaking for connection establishment and termination. This involves sending and receiving several synchronizing messages and is control dominated. We have modeled the transmitter behavior here.

- TCP receiver [18]: We have modeled the TCP receiver behavior here.
- Startup benchmark which builds a segment of the *call mode modem* startup procedure [19]: This application models the startup procedure of the call-mode modem. It is control dominated and involves sending and receiving specific streams of characters (of specific length) during startup.
- Pump controller application [20]: This is an application which controls the working of a pump inside a mine with high methane levels and is a typical control dominated application with both preemption and priority.
- An automatic teller machine (ATM) controller [21]: This models a subset of the ATM behavior focusing on control dominated actions and preemption and priority.
- A traffic light controller [22]: This code was available as part of POLIS co-design tool distribution [23]. We slightly upgraded it by adding a bus-lane and incorporated priorities.
- A lift controller [21]: We have modeled this application which is control dominated involving sampling of signals, signal emission etc.

In all these applications, we abstracted the data handling code and only focused on the reactive code (as a result the benchmarks are small). As our first comparison, we compared the execution time for FLIX and REFLIX over the same application programs. They are compiled and executed for both REFLIX and FLIX. Table 3 indicates the total number of instruction cycles for each of these benchmarks. Since the instruction cycle duration of FLIX and REFLIX are identical, comparing the number of instruction cycles is equivalent to the execution time. On an average, REFLIX turns out to be 5.92 times faster than FLIX while executing these control-dominated programs. This shows that an existing processor can be modified to enable more efficient implementation of the same control dominated application programs.

Table 3 Comparison of execution time of REFLIX and FLIX in instruction cycles

Application	**Levels of ABORT nesting**	**REFLIX**	**FLIX**	**Speedup**
TCP Transmitter	0	10	46	4.6
TCP Receiver	0	9	41	4.55
Startup benchmark	25	25	102	4.43
Pump Controller	2	14	83	5.92
ATM Machine	3	26	244	11.9
Traffic Light Controller	2	25	152	6.6
Lift Controller	0	29	116	4.14
Average				5.92

We have made comparisons between REFLIX and a number of popular processors in terms of their code size (which plays an important role for embedded applications [24]) and execution times for the same set of control dominated applications which have initially been written in Esterel and then manually translated into the native code of each of those processors (Motorola 68HC11 [25], Intel 8051 [16] and 16-bit NiOS [26]). Table 4 shows a clear advantage of REFLIX in terms of compactness of code and small memory footprint. The execution times, shown in Table 5, are expressed in the number of system clock cycles for each of the processors. Absolute execution times may be calculated by multiplying these figures with system clock period, which depends on the concrete processor implementation. We should note that figures used for FLIX and REFLIX are for the non-pipelined version of the processors (implemented in the current prototype). If pipelined version is used, those figures will be reduced by a factor of 3.

Table 4 Code size comparison for some benchmarking examples in words

Application	REFLIX	FLIX	8051	68HC11	NIOS-16
TCP Transmitter	10	46	20	31	46
TCP Receiver	9	41	28	18	41
Startup benchmark	25	102	48	76	94
Pump Controller	14	83	35	56	80
ATM Machine	26	244	102	163	219
Traffic Light Controller	25	152	70	114	147
Lift Controller	29	116	45	79	116
Average	19.7	112	46.7	77.8	106.1

Table 5 Execution time comparison (in the number of system clock cycles)

Application	REFLIX	FLIX	8051	68HC11	NIOS-16
TCP Transmitter	40	184	240	64	46
TCP Receiver	36	164	216	58	41
Startup benchmark	100	408	972	200	94
Pump Controller	56	332	708	146	80
ATM machine	104	976	2040	446	219
Traffic light controller	100	608	1320	275	147
Lift controller	116	464	732	188	116
Average	78.8	448	889.7	196.7	106.1

Finally, Table 6 presents some example implementation figures for FLIX, REFLIX and Altera NiOS processors in the same FPLD device (APEX EP20K200EFC484-2). As can be seen from these figures, implementation of reactive features in REFLIX processor requires very small increase of logic elements compared to non-reactive version of the processor with a minimal impact on maximal clock frequency.

Table 6 Resource requirement comparison for some of the FPGA implemented processors

Processor	**Maximal clock frequency (MHz)**	**Minimal machine cycle timeT = 1/f (μs)**	**Logic elements LE utilisation**	**Embedded memory blocks (ESB)**
NIOS 1.1 Reference design	37.5MHZ	0.0266 μs	25%	14%
FLIX	31.8MHz	0.0314 μs	11%	0%
REFLIX	30.5MHz	0.0327 us	13%	0%

9 Conclusions

The REFLIX approach proposes a novel way for supporting reactive systems at the processor hardware level. Inspired by the Esterel language, the first REFLIX implementation supports dealing with input and output signals in a Esterel-like model of computation, without true concurrency. However, by supporting constructs for synchronization and preemption on random and timing signals, REFLIX enables writing programs, which can be easily verified. REFLIX programs are predictable in their temporal performance and provide guaranteed reaction times on external events without unnecessary overheads and context-switching found in conventional microprocessors. Processor supports notion of time, which can easily be derived based on the fact that each instruction performs in time equal to 4 machine cycles.

We built a prototype processor, REFLIX, by extending the open source FLIX processor core with native support for reactivity and a new preemption mechanism called ABORT. We then compared execution time of the same processor core without and with reactivity support (FLIX vs REFLIX) on a set of control dominated applications and obtained an average speedup of 5.92 times. We made memory footprint comparisons of REFLIX and a set of conventional processors. This comparison clearly shows the advantage of using native reactive instructions over using common instructions to deal with reactivity. REFLIX produced considerably more compact code compared to all these processors. Execution time comparisons are relative (in terms of system clock cycles) and obviously depend on the maximum system clock frequency. Maximal system clock has been found for FPLD-implemented processors, and it shows no disadvantage of the REFLIX architecture. Finally, as the new processor is implemented in an FPLD, we have shown its feasibility and low resource requirements compared to the requirements of initial processor core and standard microprocessor implementation in the same device.

REFLIX architecture is based on a concept of flexible instruction execution unit, which is well suited to embedded systems by keeping the core simple and small (essential for embedded systems) and providing facilities for interaction with a set of functional units to achieve more complex tasks (which is also very similar in spirit to the Esterel tasking model).

The major limitation of the current implementation is that it does not support fully Esterel model of computation, which is one of our goals, particularly true concurrency and valued signals. Despite of that, we have found a number of applications that can be described much more clearly and concisely than with the native/assembly languages of the conventional processors, which do not have similar support for reactivity, and also verified using formal methods.

References

[1] Harel D. Statecharts: A Visual Formalism for Complex Systems, Sci. Comput. Prog., 8; 1987, pp. 231-274

[2] Pnueli A. Application of temporal logic to the specification and verification of reactive systems: a survey of current trends, Lecture notes in computer science, 224; pp. 510-584. Springer Verlag, 1986

[3] Berry G. and Gonthier G. The ESTEREL synchronous programming language, Sc. Comput. Prog., 19; 1992, pp. 87-152

[4] Fisher J.A. Customized instruction sets for embedded processors. In Proc. 36th Design Automation Conference, 1999, pp. 253–257

[5] Altera Corporation. Excalibur Embedded Processor Solutions, http://www.altera.com

[6] Triscend. The Configurable System on a Chip, http://www.triscend.com

[7]] Xilinx Corporation. IBM and Xilinx team to create new generation of integrated circuits, http://www.xilinx.com/prs rls/ibmpartner.htm

[8] Wirthlin M and Hutchings B. A dynamic instruction set computer. In Proc. IEEE Symp. on Field Programmable Custom Computing Machines, pp. 99–107. IEEE Computer Society Press, 1995.

[9] Donlin A. Self modifying circuitry - a platform for tractable virtual circuitry. In Field Programmable Logic and Applications, LNCS 1482, pp. 199–208. Springer, 1998

[10] Salcic Z. and Maunder B. “CCSimP - an Instruction-level Custom-Configurable Processor for FPLDs”, in *Field-Programmable Logic FPL ‘96*, *Lecture notes in Computer Science 1142* (R.Hartenstein, M.Gloessner and M.Servit editors), Springer, 1996, pp. 280-289

[11] Salcic Z. and Mistry T. FLIX Environment for Generation of Custom-Configurable Machines in FPLDs for Embedded Applications, *Elsevier Journal on Microprocessors and Microsystems*, vol.23(8-9), December 1999, pp. 513-526

[12] Peng S. P., Luk W. and Cheung P.K.Y. Flexible instruction set processors. Proceedings *CASES’00*, November 17-19, 2000

[13] Ramesh S. and Bhaduri P. Validation of pipelined processor designs using Esterel tools, *Proceedings of the 11th International Conference on Computer Aided Verification*, Springer Verlag, 1999, pp. 84 - 95

[14] Girault A. and Berry G. Circuit generation and verification of Esterel, International Symposium on Signals, Circuits and Systems, Tech. Univ. Iasi., Romania, 1999, pp. 85-90

[15] Belachew M. and Shyamasundar RK. MSC/sup +/: From requirement to prototyped systems, *Proceedings of the 13th EUROMICRO conference on real-time systems*, IEEE Comput. Soc., 2001, pp. 117-124

[16] M. E. Schrader, R. Sridhar, T. Buechner, and P. P. K. Lee, “VHDL design of embedded processor cores: the industry standard microcontroller 8051 and 68HC11”, in *IEEE International ASIC Conferenc*e, 1998, pp. 256–259

[17] “Intel Peripheral Datasheets for 82C59 Programmable Peripheral Interface”, http://developer.intel.com/designer/datasheets, 1995.

[18] Kurose J. F. and Ross K. W., *Computer Networking: A Top Down Approach Featuring the Interne*t, Addison Wesley, 1999.

[19] *International Teleommunication Unio*n, ITU-T recommendation v.32 edition, 1993.

[20] Gomaa H., *Software design methods for concurrent and real-time systems*, Addison-Wesley, 1993.

[21] Gomaa H., *Designing concurrent distributed and real-time applications with UML*, Addison-Wesley, 2000.

[22] Mead C. and Conway C., *Introduction to VLSI system*s, Addison-Wesley, 1980.

[23] Balarin F., Chiodo M., Guisto P., Hsieh H., Jurecska A., Lavagno L., Passerone C., Sangiovanno-Vincentelli A., Sentovich E., Suzuki K., and Tabbara B., *Hardware Software Codesign of Embedded Systems - The POLIS Approach*, Kluwer, 1997.

[24] Furber S., *ARM system-on-chip architectur*e, Addison-Wesley, 2000.

[25] Spasov P., *Microcontroller Technology: The 68HC1*1, Prentice Hall, 1999.

[26] *NIOS Embedded Processor: 16-bit Programmer’s Reference Manual*, www.altera.com.