REFLIX: A Processor Core for Reactive Embedded Applications

Zoran Salcic[1,2], Partha Roop[1], Morteza Biglari-Abhari[1], Abbas Bigdeli[1]

[1] Department of Electrical and Electronic Engineering, The University of Auckland, Private Bag 92019, Auckland, New Zealand,
[2] on leave at The University of Technology Vienna, Institute of Communications, Gusshausstr. 25/389, 1040 Vienna, Austria
z.salcic@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. We propose novel microprocessor architecture that has native support for reactivity, with the flexibility to be customized at much higher level than usual microprocessor-based solutions. The proposed microprocessor architecture is an extension of our existing FLIX processor open core. The new processor core, called REFLIX (Reactive FLIX), guarantees at most one instruction cycle delay for priority resolution and preemption and supports design style for reactive applications used in Esterel programming language.

1 Introduction

Embedded systems most often have a dedicated microprocessor or a microcontroller that executes a non-terminating control program, which controls the environment that constitutes of a set of sensors and actuators, which are used for interaction with the environment. 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]. Determination of the environment status can be made 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 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 the interrupt handling is executed concurrently with the main control task, there is a danger of inconsistent

M. Glesner, P. Zipf, and M. Renovell (Eds.): FPL 2002, LNCS 2438, pp. 945-954, 2002.
© Springer-Verlag Berlin Heidelberg 2002

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.

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 such task in addition to a set of standard set of instructions found in common microprocessors. The proposed approach provides mechanism to avoid busy waiting associated with polling, when required, and context switching associated with interrupts completely. The environment interaction model of a reactive programming language Esterel [3] for embedded systems inspires this mechanism. The major contributions of this paper are the following:

a) We propose a microprocessor architecture that supports reactivity through a set of native instructions, which are lacking in previous architectures. All instructions, including those that look like conventional instructions for polling, perform in the same time equal to 4 machine cycles contributing to both efficiency and predictability of program execution.
b) REFLIX supports 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. Our proof of concept prototype can resolve 4 levels of external event priorities during preemption, which is guaranteed to happen in a single instruction cycle. This is an extremely important feature for implementation of real-time tasks.

Section 2 gives background and framework for REFLIX design. In section 3, we introduce the REFLIX architecture and instruction set with emphasis on features that support reactivity. In section 4, we describe in more details the REFLIX data path and control unit together with some implementation aspects. Section 5 presents some concluding remarks related to the current implementation and our future work.

2 Related Work

One of the trends in implementation of embedded systems is to rely on processors for specific applications that better match requirements of those applications than general-purpose instruction processors [4]. There are several approaches suggested or used for customisation 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 are using 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]. Another processor cores [10,11] provide generic mechanisms for new instruction implementation that execute 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 processor RISC-type architecture with more or

less usual instruction sets that belong 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 and mechanisms found in conventional processors.

3 REFLIX Framework

REFLIX 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. 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). In addition to such constructs for control flow Esterel supports data handling through host language such as C or Java.

REFLIX is a processor core designed to capture main ideas of Esterel interaction model with the environment. For that purpose REFLIX provides a set of native instructions suitable for reactive. These include native facilities for *delay*, *signal emission*, *priorities*, *preemption* and *task execution* facility using functional units. REFLIX essentially represents another customization of our existing FLIX processor [10,11] open core. By adopting Esterel model for reactivity and by supporting it 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 complicated software means that can be implemented on standard microprocessors but with many difficulties.

4 REFLIX Core Architecture and Features

The current version of REFLIX has adopted the 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 numbers of input sensor and output sensor lines. Support for up to16 sensor input lines, Sin[15..0], and 16 signal output lines, Sout[15..0].
- Notion of real-time in terms of numbers of instruction cycles. One instruction cycle corresponds to one tick of the Esterel reaction clock and equals to four machine cycles.
- Introduction of internal timers that generate user programmable TimeOut signals. Number of internal timers is customizable and 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 based on external events. Any piece of code can be wrapped up within *abort* statement (abort body) and immediately abandoned in case that an external event on specified sensor input or timing event occurs.

4.1 Reactive Instructions

There are five basic instructions in the reactive category and they are presented in Table 1. 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 1 REFLIX instructions supporting reactive processing

Instruction syntax	**Length and format** (No of bits in parenthesis)	**Function/Description**
SAWAIT signal	W1: Opcode(8) Signal(4)	Wait until signal is present.
TAWAIT delay	W1: Opcode(8) W2: Time(16)	Immediate delay— wait until specified time elapses (wait at least one system tick - time is expressed in the number of instruction cycles)
EMIT signal	W1: Opcode(8) Signal(4)	Specified signal is set high for one instruction cycle
SUSTAIN signal	W1: Opcode(8) Signal(4)	Specified signal is raised forever
ABORT <Timer/Signal> continuation-address	W1:Opcode(10) Timer(2)/Signal(4) W2: Address(16)	Preemption instruction. Abort has a body up to the instruction whose address is indicated by the continuation address. Signal can be either external one or from an internal timer

In order to illustrate the use of new reactive instructions, let's look to a simple example of a program that controls a seat-belt alarm controller. After every time a car's ignition is turned on, check if the driver's seat belt is fastened within some C1 time. If seat belt is not fastened within this time, generate an alarm until alarm-input is turned off. The following is a simple REFLIX program to implement this behavior:

```
start:
  SAWAIT IGNITION-ON;               wait until IGNITION-ON
      START Timer1, C1;             initialize Timer1 with C1
      ABORT Timer1 addr1;           abort activated on Timer1 timeout
        AWAIT BELT-ON;              wait until BELT-ON
addr1: ABORT ALARM-INPUT-OFF addr2; abort activated on
                                    ; ALARM-INPUT-OFF
          SUSTAIN ALARM;            activate ALARM
addr2:    JMP start;                back to beginning
```

4.2 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.

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:

<table>
<tr><td>OPCODE (10)</td><td>Timer(2)/Signal(4)</td></tr>
<tr><td colspan="2">Continuation-address (16)</td></tr>
</table>

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:

- **Abort activation**. It is executed immediately after the fetching and decoding ABORT instruction, when REFLIX starts monitoring change (activation) of the designated signal. The 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, the 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) also supports nesting and prioritizing of abort statements. Current proof of concept implementation supports nesting to up to 4 levels of aborts. The AHB contains active abort signal register (AASR) block with 4 registers with the length that equals 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 an unique code gener-

ated using a one-hot encoding scheme (only one bit can have value 1). Addresses of 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 can't 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 corresponding AASR register (with the same address) is non-empty (with AF(0) being 1 for highest priority monitored signal). Summary joint abort flag (JAF) contains information on presence of monitored signals, or

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

The REFLIX control unit determines an action path during instruction execution based on the value of the JAF bit as it is shown in section 4. Another register block contains four active abort address registers (AAARs), which are used to store continuation addresses of currently active abort instructions. Highest priority ABORT instruction (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 existence of pending (non-processed) abort events. For that purpose, another flag, called 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. It's value is derived as

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

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

Two pointers called abort read pointer (ARP) and abort write pointer (AWP) are used to up-date addresses of registers within the AHB from which information will be read or written to and they are not user visible. They are used only by the control unit and can be considered as its part.

Other parts of programming model include signal output register (SOR) with individually controllable/writtable bits, and pool of timers that appear as memory mapped registers with some programmable features. The level of their programmability is application dependant 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 the REFLIX data path.

5 REFLIX Design and Implementation

REFLIX data path with emphasized differences to original FLIX data path is shown in Figure 1. 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). Abort handling block is shown with the shaded background in Figure 1.

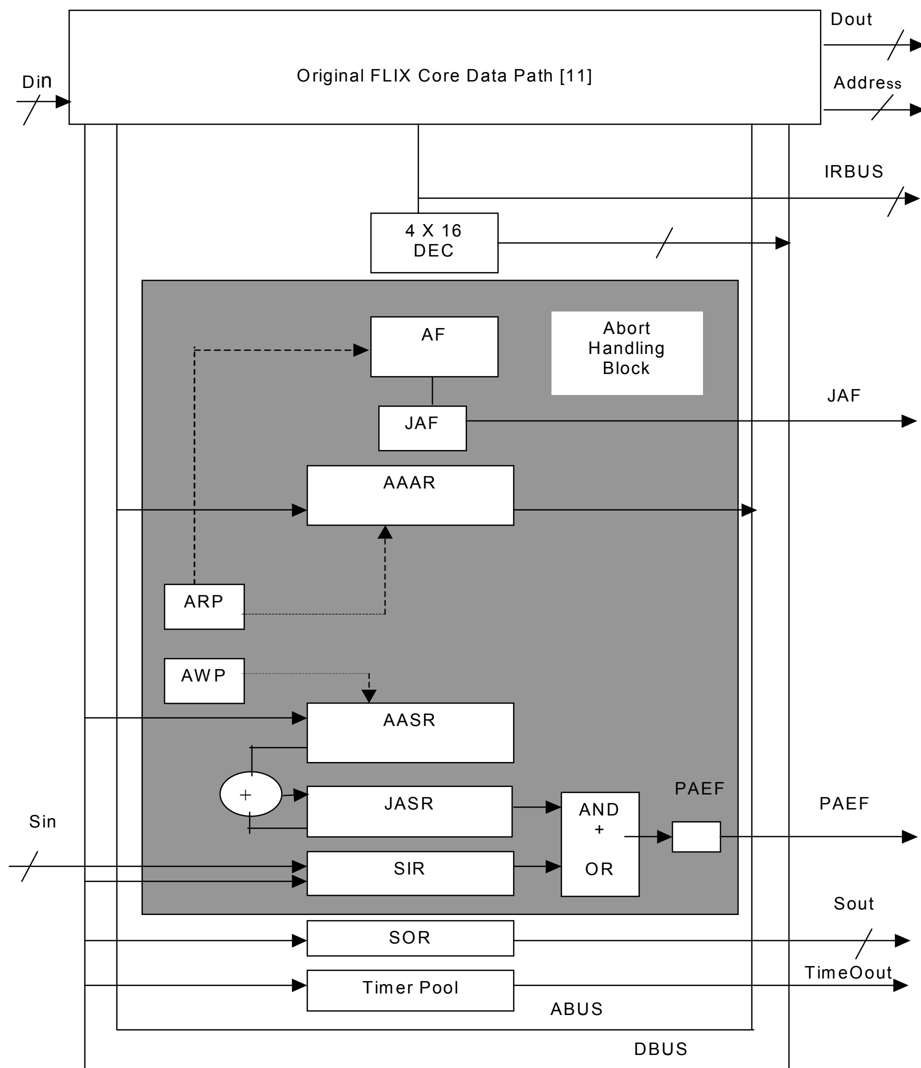

Figure 1 REFLIX data path

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 when using them 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 the easy maintenance of time, both REFLIX global 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 2. All events on monitored signals 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 2, as it carries operations on the registers of the AHB.

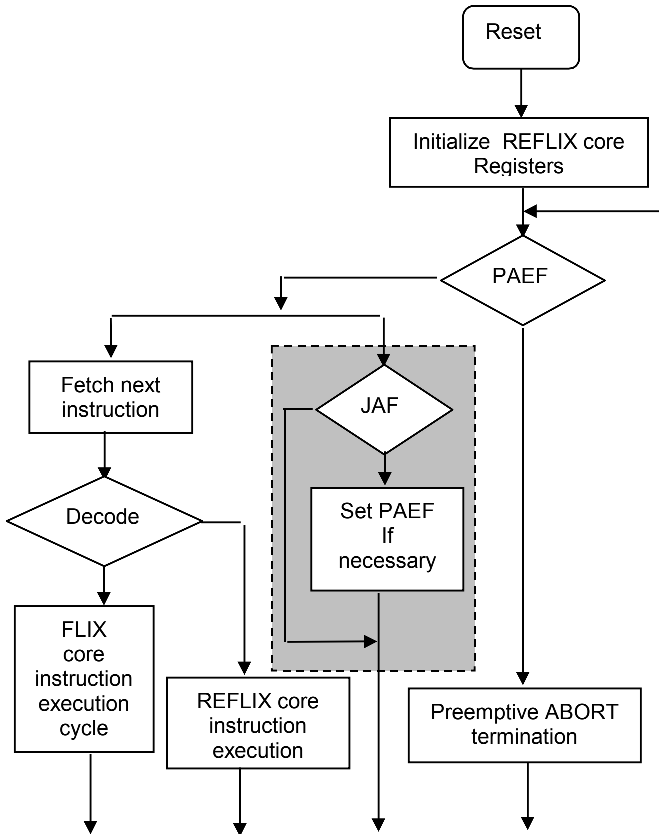

Figure 2 REFLIX control unit global operation

- 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 usable equivalent gates, it becomes feasible to build whole systems on programmable chips (SoPC).

On the other hand FPLDs are accompanied with advanced design tools and HDLs that enable design of parameterized designs that can be relatively easily customized for specific applications. This aspect of REFLIX design has not been emphasized in the paper. The first implementation is used as a proof of concept and incorporates parameterization of only some of 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).

6 Conclusions

The REFLIX approach represents a novel way for supporting reactive systems at the processor hardware level. Inspired by Esterel language, the first REFLIX implementation supports dealing with input and output signals in 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 fact that each instruction performs in time equal to 4 machine cycles. In concrete FPLD implementations minimum clock cycle can go to up to 25ns, which defines temporal features of the solution.

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 REFLIX core is generic in many respects, by providing parameterized configuration that can be easily customized and instantiated for specific application. As such it is suited for SoPC applications. It fits in a fraction of typical high-capacity FPLD, thus leaving plenty of space for modifications, additions and customisation.

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 assembly languages of the conventional processors, which don’t 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, pp. 253–257,1999.
[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] S.P. Peng, W. Luk and P.K.Y Cheung. Flexible instruction set processors. Proceedings *CASES'00,* November 17-19, 2000, San Jose, California