Available online at www.sciencedirect.com
SCIENCE DIRECT®

ELSEVIER

Microprocessors and Microsystems 30 (2006) 72–85

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

HiDRA—A reactive multiprocessor architecture for heterogeneous embedded systems

Zoran Salcic*, Dong Hui, Partha S. Roop, Morteza Biglari-Abhari

Department of Electrical and Computer Engineering, The University of Auckland, Private Bag 92019, Auckland, New Zealand

Received 17 May 2005; accepted 25 May 2005
Available online 20 June 2005

Abstract

Embedded systems are typically heterogeneous requiring interacting hardware and software components, are locally synchronous while being globally asynchronous and combine both control and data dominated blocks. Conventional architectures and hardware–software platforms do not directly support such heterogeneity leading to complex design flow and verification process for such systems. This paper presents a new architecture for heterogeneous embedded systems called HiDRA based on multiple reactive processor cores. The architecture supports globally asynchronous locally synchronous systems with a mix of data-dominated and control-dominated behaviors. The reactive processor cores implement Esterel-like computation with architectural support for signal polling, emission and preemption. HiDRA also provides primitives for communication and synchronization between concurrent processes. A low level (concurrent reactive assembly) language has been specified to model embedded applications, which are executable directly on the HiDRA platform. The first implementations with up to four reactive processors have been done on the standard FPGAs. Performance comparison with a typical application realized from system level language ECL reveals significant speedup and reduction in code size.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Heterogeneous; Embedded; Reactivity; Multiprocessor; FPGA

1. Introduction and background

Embedded systems are ubiquitous computing systems ranging from simple home appliances to complex applications in avionics and defense. A key feature of embedded systems is the complexity arising from heterogeneity of different processing requirements as argued in [1]. Consider, for example, a simple hand-held device such as a recent cell phone. This device combines the functionality of a phone, a PDA, an MP3 player and possibly a digital camera while being quite constrained in terms of its memory requirements and power consumption. This application illustrates the heterogeneous nature of modern embedded systems due to:

1. *Interacting hardware and software components of the system*. A pure hardware solution may be efficient but may not meet the cost requirements set out in the specification document. A pure software implementation, while being cost effective, may not meet the strict timing requirements.
2. *Combination of control dominated and data dominated behaviors*. While the RTOS kernel on the PDA is mostly control-dominated (requiring quick reaction to input events), the MP3 player is mostly data dominated (sampling its input at regular intervals for processing).
3. *Globally asynchronous and locally synchronous design (GALS)*. In order to realize the final product within the specified window, three IP blocks from different vendors was integrated. Each IP block used its own communication primitives and clock frequency and a GALS design using a standard bus and wrappers [2] (protocol converters) was employed.
4. *Heterogeneous nature of embedded systems (the different design styles and tools used and the various models of computations employed are amply illustrated in* [1]). Many languages suitable for control dominated behaviors [3–5] and data dominated behaviors [6,7] have been developed. Synthesis and co-design tools [8] that operate on a given domain are also available. Languages

* Corresponding author. Tel.: +64 9 373 7599x87802; fax: +64 9 373 7461.
E-mail address: z.salcic@auckland.ac.nz (Z. Salcic).

0141-9331/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2005.05.001

targeting GALS systems have been proposed recently [8–10]. Moreover, there has been considerable interest in GALS due to IP based design concerns [11] and some attempts at developing GALS processors [12] have been made. Languages such as SystemC [13] and SpecC [14] have been developed to cater to heterogeneous aspects of embedded systems and simulation and synthesis tools around them are being developed. However, a unifying configurable architecture for rapid prototyping of these systems is lacking and will be of considerable interest.

This paper proposes a flexible and configurable multiprocessor architecture, called *HybriD R*eactive *A*rchitecture (HiDRA) for rapid prototyping and implementation of heterogeneous embedded systems based on a set of *Re*active *MIC*roprocessor (ReMIC) cores. The initial ideas of the HiDRA and ReMIC have been first presented in [15] and [16], respectively. In this paper, the design and implementation of the full HiDRA system, its constituent parts and the tools for development of the heterogeneous applications are presented.

The main contributions presented in this paper are:

1. A new, scalable, multiple processor architecture is proposed for supporting GALS model of computation. The proposed architecture is suitable for implementation of embedded systems that contain reactive or control dominated parts that interact with external environment. Concurrent reactive modules communicate using handshaking.
2. Support for hardware/software co-design through special purpose functional (application-specific) units that implement hardware functions while software functions are implemented on ReMIC reactive processors.
3. Control dominated behaviors are implemented using native ReMIC instructions that support reactivity. The ReMIC instruction set architecture (ISA) has direct support for priority, preemption, signal emission and polling similar to our earlier work [17]. While REFLIX processor [17] was a proof of concept built around already existing traditional processor core by adding reactive features, the ReMIC processor represents the first reactive core developed from the scratch.

Data dominated behaviors may be implemented either on specialized hardware (functional units) or using programs running on the ReMIC processor cores. The low-level instruction set architecture is extended to an intermediate-level called concurrent reactive assembler language (CRAL), which is more suitable to heterogeneous nature of embedded applications. A CRAL program can serve as a starting point in embedded system design, as it contains enough information to synthesize both HiDRA configuration and programs that will be executed in each of the synthesized ReMIC cores. This language is also aimed to serve as an intermediate representation between high-level system languages and system implementation platform. The development of a specialized assembly language like CRAL for asynchronous multiprocessors is novel. CRAL uses CSP [18] and CCS [19] like handshaking synchronization. The set of tools, configuration and program generators, that are used to automate configuration generation, software development and synthesis of system for FPGA platform are also implemented.

The paper is organized as follows. Section 2 further explains the goals and objectives of the HiDRA architecture from the point of view of heterogeneous embedded systems. The main element of the architecture, the ReMIC processor, is presented in Section 3, with emphasis on the part that supports reactivity on both events coming from the external environment, and events internally generated due to process (or) synchronization and communication. Section 4 presents a new assembly language for low-level process synchronization called CRAL. An example of HiDRA architecture implemented on standard FPGA family is shown in Section 5 and an application example running on this implementation is presented in Section 6. Development tools and environment used to prototype test systems based on HiDRA are presented in Section 7. Some results that illustrate the complexity of HiDRA and the associated performance figures are given in Section 8. Finally, conclusions and future work are presented in Section 9.

2. An overview of HiDRA

HiDRA is a heterogeneous system that consists of multiple reactive processor cores for software-implemented behaviors and functional units for hardware-implemented behaviors. It allows interconnecting hardware- and software-implemented behaviors in almost unrestricted way. The architecture is suitable for FPGA prototyping as it uses some of the features of current FPGA devices like distributed SRAM memory blocks, but easily fits to the System-on-Chip approach and ASICs. The architecture has the following major features:

1. It allows execution of concurrent behaviors that can be software-implemented and run on a number of physical ReMIC processor cores or hardware-implemented in the functional units.
2. A master processor (MP) performs system initialization and coordination of concurrent activities, but also can implement application behaviors.
3. Other processor cores, called application processors (AP), are used to implement only concurrent application behaviors.
4. Behaviors implemented on processor cores support FSM-type model of computation, in addition to declarative-type computation used in data-driven applications in traditional processors.

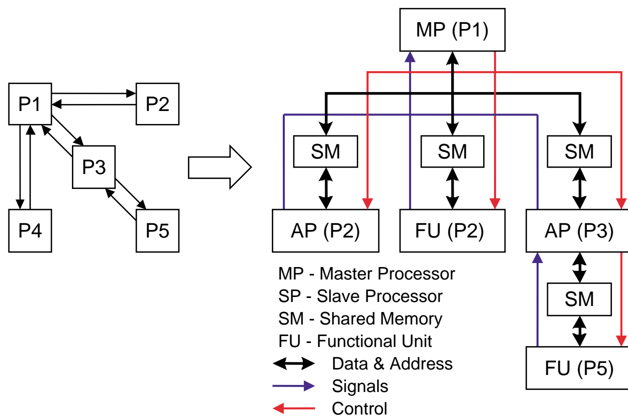

Fig. 1. Mapping of an application to HiDRA architecture.

5. Concurrent processes implemented in hardware functional units, are primarily aimed at data-driven behaviors.
6. Communication between behaviors, whether they are implemented using programs that run on processor cores or hardware-implemented functional units, is achieved by means of a layer of shared memories (sometimes just registers), which connect processor cores and functional units as shown in Fig. 1.

Fig. 1 shows these major concepts and shows an example of possible mappings of an application represented by the dependence graph of processes (P_1,...P_5) on the architecture. HiDRA has a hierarchical topology that contains one master processor (MP) which implements main system behavior in software (MP[P1]) and controls all other units (application processors, APs and functional units, FUs) that implement other behaviors on the lower hierarchical level (e.g., AP(P2), AP(P3) and FU(P4)). Application processor behaviors can activate hardware-implemented behaviors on their local functional units (e.g. FU(P5)). Synchronization between behaviors is accomplished via signal manipulation instructions to be elaborated in Section 4 (that emit a signal or poll a signal for the presence of an event). Communication primitives available in CRAL, as they will be described in Section 4, are implemented using shared memories and signal manipulation statements that perform low-level handshaking. A range of communication primitives such as bounded FIFO and rendezvous [1] may also be supported.

3. Reactive microprocessor—ReMIC

ReMIC processor core has been designed to provide:

1. *Efficient mapping of control-dominated applications on processor ISA*: this approach leads to better performance and code density for control-dominated (part of) applications.
2. *Support for direct execution of reactive language features*: this ensures that control dominated languages can be supported directly on a processor without any intermediate code generation performed by conventional compilers.
3. *Support for concurrency and multiple processor configurations*: the processor architecture supports building multiple processor systems that can simultaneously execute multiple application (concurrent) processes and provide a mechanism for their synchronization and communication.

ReMIC design [20] is inspired by some of the features of the synchronous language Esterel [3]. This is achieved with a set of native instructions for control-dominated applications in addition to instructions found in traditional processors. These new native instructions provide direct support for *delay*, *signal emission*, *priority* and *preemption*. Concurrency is achieved by multiple ReMIC cores, where generic signal manipulation instructions can be used to implement efficiently synchronization of concurrent tasks running on separate processors. The key features of ReMIC that facilitate reactive applications are summarized as follows:

- One 16-bit Signal Input Port (SIP) and one 16-bit Signal Output Port (SOP) are implemented to enable the realization of Esterel-like pure (binary) signals [3]. Simultaneous emission of multiple signals in one clock cycle is also supported.
- Two user programmable internal timers are implemented to generate the timeout signals, which can be fed back to be used internally for synchronization purpose or used to interact with the environment.
- ABORT instruction is introduced to handle Esterel-like preemption. Code can be wrapped up in the abort statement and immediately abandoned when an external event on the specified SIP input occurs. Up to four levels nesting of aborts is supported with maximum one instruction cycle delay to react to the abortion condition. Besides the execution efficiency and determinism, this also leads to very structured programming style when programming the reaction on external events.
- Other instructions including EMIT, SUSTAIN, PRESENT, SAWAIT, TAWAIT, CAWAIT are added to support Esterel-like reactive instructions (refer to Table 1).

Fig. 2 shows ReMIC functionality. It consists of a traditional RISC-type pipelined microprocessor datapath, reactive functional unit (RFU) for handling external and internal signals, and processor control unit. ReMIC has Harvard architecture with 32-bit wide program and 16-bit wide data memory. Although we will not concentrate on its customization features here, ReMIC is a parameterized

Table 1
Instructions supporting reactivity and concurrency

Feature	Instructions	Semantics and descriptions
Signal manipulation		
Signal emission	EMIT *signal(s)*	*Signal(s)* is/are set high for one tick
Signal sustenance	SUSTAIN *signal(s)*	*Signal(s)* is/are set high forever
Delay	TAWAT *delay*	Wait until *delay* (number of instruction cycles) elapses
Signal polling	SAWAIT *signal*	Wait until *signal* occurs in the environment
Conditional signal polling	CAWAIT *signal1*, *signal2*, *address*	Wait until either *signal1* or *signal2* occurs. If *signal1* occurs, execute instruction at the address immediately followed, or else at the specified *address*
Signal presence	PRESENT *signal*, *address*	Instruction at the address immediately followed will be executed if *signal* is present, or else at the specified *address*
Preemption		
Preemption	ABORT *signal*, *address*	Program finishes its current instruction and jumps to *address* in the occurrence of *signal*
Concurrency		
Software process activation	START AP_ID, processID	Send a signal to a process on an application processor to activate a process; implemented using EMIT machine instruction
Wait on software process termination	JOIN AP_ID, processID	Waits on termination signal from the slave process; implemented using SAWAIT; EMIT sequence; emit used to acknowledge slave processes successful joining
Hardware process activation	STARTFU FUID, command	Send a signal to a functional unit to execute specified command using EMIT
Wait on hardware process termination	WAITFU FUID	Waits on termination signal from functional unit using SAWAIT
Communication		
Send message to a software process	SEND (message,, AP_ID, processID)	Copy message into shared memory of the AP; EMIT signal that the message is ready
Receive message from a software process	RECEIVE (message, AP_ID, processID)	Check the presence of signal from process; copy message from the shared memory
Send message to a hardware process	SENDF (message, FUID)	Copy message into shared memory of the functional unit; EMIT signal that the message is ready
Receive message from a hardware process	RECEIVEF (message, FUID)	Check the presence of the signal from the functional unit; copy message from the shared memory
Parallelism control commands		
Main program beginning	&MAIN_BEGIN	Beginning of the master program
Main program end	&MAIN_END	End of the master program
Slave program beginning	&SPR_BEGIN	Transformed into ABORT; SAWAIT sequence which wraps-up code of the slave process; normally the slave process is waiting on activation signal from the master; slave process can be terminated at any time by the master
Slave program end	&SPR_END	Transformed into SUSTAIN; JMP to ABORT sequence; the slave process sustains termination signal until the master process acknowledges termination

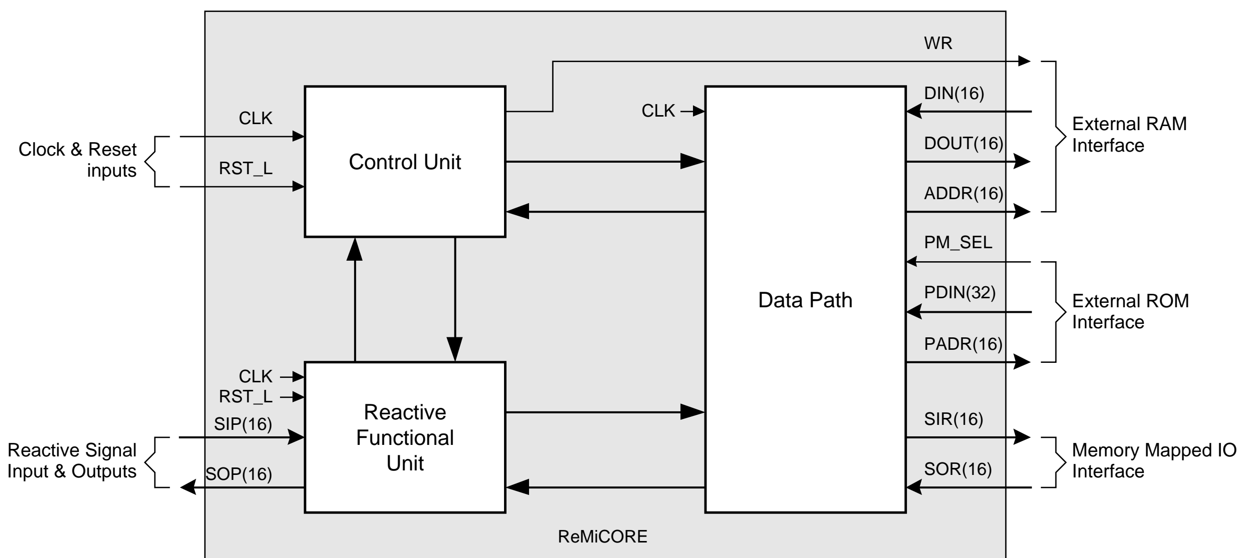

Fig. 2. REMIC block diagram.

soft-core that can be instantiated to a given application-required configuration.

Generally, the hardware implementations of Esterel-like instructions result in better performance and much more efficient compilation of Esterel programs [21]. ReMIC has a group of seven native instructions that support reactive (signal manipulation and preemption) processing. They are shown in Table 1. All ReMIC instructions are 32 bits long. Instruction formats of some reactive instructions are illustrated below.

3.1. EMIT—signal emission

EMIT, with the format as shown below, is used to generate external output signals through the signal output port (SOP). The signals last for one clock cycle. Bits 24–9 of the instruction are mapped to bits 15–0 of the SOP.

31–30	29–23	24–9	8–0
AM(2)	OC(5)	Signals(16)	Unused(4)

3.2. SAWAIT—signal polling

SAWAIT, with the format as shown below, is used to poll for a specified signal from the signal input port (SIP). ReMIC stays in a wait state until the signal occurs in the environment.

31-30	29–23	24–9	8–5	4–0
AM(2)	OC(5)	Unused(16)	SIG(4)	Unused(4)

3.3. ABORT—preemption

ABORT, with the format as shown below, is the most crucial reactive instruction because it is introduced to support preemption with priorities. An ABORT instruction has a signal, which is sensitive to it and a continuation address. ABORT instruction becomes active from the instant it is executed until either (1) it reaches the continuation address, or (2) an event on one of the SIP inputs occurs that preempts all unexecuted instructions within the body. Bits 24–9 of the instruction specify the abort continuation address and bits 8–5 specify the abort signal that is encoded to one of the SIP inputs.

31–30	29–23	24–9	8–5	4–0
AM(2)	OC(5)	Continuation address(16)	SIG(4)	Unused(5)

4. Concurrent reactive assembly language—CRAL

The CRAL has three distinct groups of instructions: (1) the standard assembly language core that can be extended based on the customization of the processor core, (2) novel reactivity support in the form of instructions for signal manipulation and preemption and (3) instructions for concurrency support that enable control of concurrent execution, synchronization with software and hardware implemented behaviors running outside the core. The above instructions also serve as the basis for communication primitives/mechanisms that are implemented by instructions from the previously described groups.

In order to simplify process synchronization and enable process communication, a set of synchronization and communication primitives is included in the CRAL, as shown in Table 1.

A CRAL program consists of two major parts: (1) a main program that will be executed on the master processor and (2) a number of application programs that model concurrent processes running on the application processors. The programmer determines the number of application programs and allocates them to the application processors. The code generator performs automatic allocation and following this the application processor always executes the same program.

An example of the skeleton for a concurrent reactive program that requires a master and two application processors for its execution is shown in Fig. 3. Programs that are executed on application processors are not shown. The START primitive forks a concurrent process to start execution on a AP or FU. The JOIN primitive, similarly, waits to synchronize a forked thread with a main thread. Communication between processes is facilitated using SEND and RECEIVE (SENDF and RECEIVEF for FUs) which are very similar to CSP [18] like handshaking among two concurrent processes. While CSP supports blocking read and blocking write, the SEND in CRAL is non-blocking while RECEIVE is blocking in the current implementation. It is reasonably easy to slightly alter

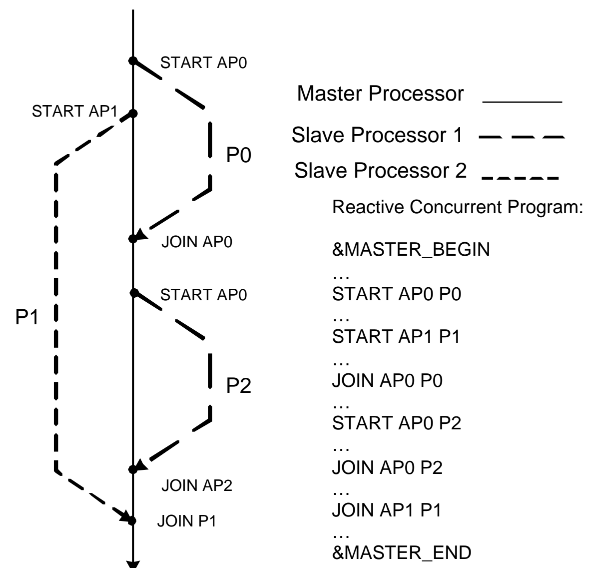

Fig. 3. Example of a CRAL program.

the implementation to support both blocking read and blocking write to ensure lossless data exchange.

Since REMIC is a reactive processor supporting signal emission, signal polling and preemption based on a signal, it is easy to map the communication and synchronization primitives in Table 1 to low-level assembler instructions in REMIC. For example, START maps to an EMIT that emits a signal to a specified processor to start a concurrent process. Similarly, JOIN maps to an AWAIT for a specified signal followed by an EMIT.

Each process may be implemented on an AP or on an FU and the main process is implemented on the MP. Using the following code segment snapshot we will illustrate how the reactive ISA is used for low-level process synchronization in CRAL and HiDRA:

```
;code running on master MP
RST_M: ABORT @RESET AGAIN_M ;body of abort begins
       JSR COMPUTE1
       EMIT #BEGIN_AP0;initiate processing on AP0
       JSR COMPUTE2
       SAWAIT @DONE_AP0 ; synchronize with AP0
       EMIT #ACK_AP0

AGAIN_M: ;body of abort ends - continuation address to
         ;handle the exception
       EMIT #AP_RST
       JMP RST_M

;code running on application processor AP0
RST_0:  ABORT @AP_RST AGAIN_0
        SAWAIT @BEGIN_AP0; wait to be activated by MP
        JSR COMPUTE3
        ABORT @ACK_AP0 AGAIN_0
        SUSTAIN #DONE_AP0
        HALT
AGAIN_0:JMP RST_0
```

Preemption is supported in CRAL through the ABORT construct. An ABORT instruction encloses a body of statements. Execution of an ABORT initializes a signal (an abortion condition, for example RESET) and an address (called the continuation address for example AGAIN_M). Once the ABORT statement is executed, the body starts while the processor control logic monitors in parallel to check the abortion signal. If the designated signal occurs before the body is completed, the body is terminated and program control reaches the continuation address to handle the ABORT. Alternatively, the body may complete prior to the designated signal occurrence and program control will reach the continuation address naturally. This example illustrates how EMIT, SAWAIT and SUSTAIN are used for handshaking between processes and ABORT is used for preemption support.

This code snapshot contains two sequences that implement two processes, one on the master processor (MP) and another on the application processor (AP0). The process execution on the MP starts immediately upon system start and is re-entered (process re-initialized) whenever a global RESET signal is present or the AP0 finishes its computation (terminates its process) and indicates it to the MP (with DONE_AP0). The MP reactivates the process on the AP0 by emitting BEGIN_AP0. In the case of global RESET, process on the MP is terminated by preemption and the process on AP0 is re-initiated (also by preemption). The handshaking protocol between processes running on the MP and AP0 processor is implemented by using EMIT/SAWAIT signal manipulation instructions and ABORT instructions. Application oriented computation is performed within subroutines COMPUTE1, COMPUTE2 and COMPUTE3.

5. An implementation of HiDRA

The details of HiDRA implementation for the Altera FPGA devices [22] are presented in this section. The example contains the following default configurations:

1. All the ReMIC cores are configured with 512×32-bit internal program memory and 256×16-bit internal data memory.
2. Shared memory (SM) is implemented using a triple-port SRAM module (ALT3PRAM) from the Altera megafunction library, which consists of two read ports (Qa, Qb) and one write port (Din). The MP is configured with read-and-write access to one SM and read-only access to the other, as shown in Fig. 4, while the AP is configured the other way around.
3. The data memory maps of the MP and APs are configured as shown in Fig. 5. The current experimental implementations support up to four APs.
4. The control signals between the MP and APs are configured as shown in Fig. 6. The MP uses four SOP signals and four SIP signals for each AP and spares the same number for connection with external environment.

Fig. 7 shows a complete HiDRA implementation with one MP and two APs using the above configurations.

HiDRA gives a number of mechanisms that can be used to implement blocking and non-blocking read and write when necessary. As signal lines are used for synchronization between processors, they can be manipulated directly from the corresponding ReMIC reactive instructions, which all perform in a single instruction cycle. To emit a signal and notify another processor on the event requires single instruction cycle, while the notification is received in either a single cycle (if using ABORT as non-blocking mechanism) or in a number of cycles depending on incoming event (when using AWAIT instructions as a blocking mechanism). Additional time is required for writing and reading actual data if a binary signal is accompanied with the value which is stored in shared memory (writing or reading each word requires one instruction cycle).

A program that helps the user to automatically generate a VHDL description file for the complete HiDRA architecture with a user specified number of APs (currently up to 4) has been developed. The generated VHDL file, called ‘multcore.vhd’, can be synthesized for the Altera FPGAs or used for simulation within the ModelSim [23] environment.

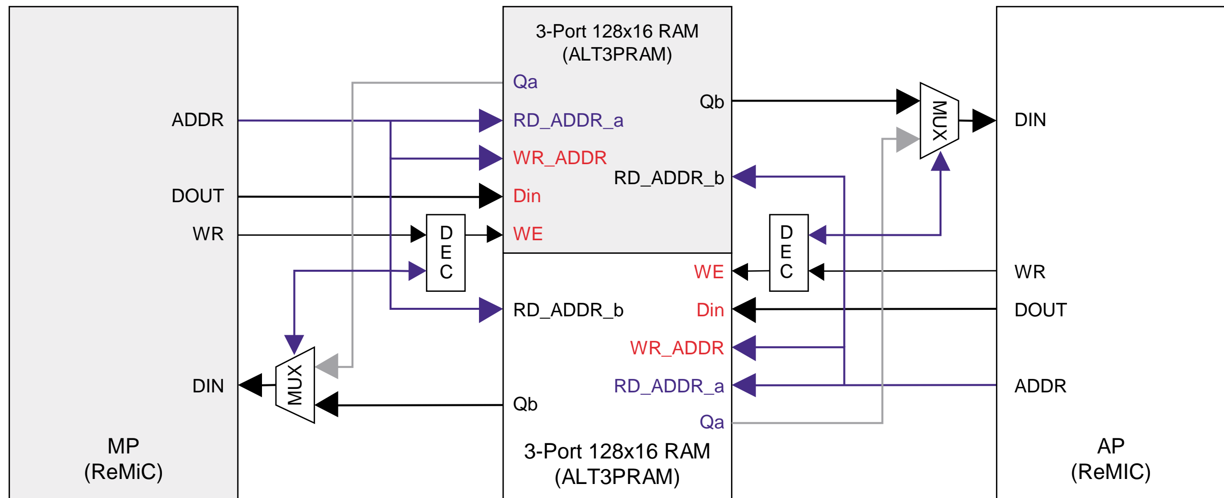

Fig. 4. SM configuration between MP and AP.

6. An application example—protocol stack

In this section, we illustrate how CRAL may be used on the protocol stack example from [5]. The described system receives input bytes on an input signal port called INPUT_BYTE and assembles them into packets of predefined size. Once a packet is assembled a notification is sent to another module that checks for correct packet reception by calculating the CRC code. The received packet is also processed using a lengthy computation in which a matching of the received packet information with a predefined address pattern is performed. If it is detected by CRC check that the packet is incorrectly received, the whole process is abandoned. If the CRC is

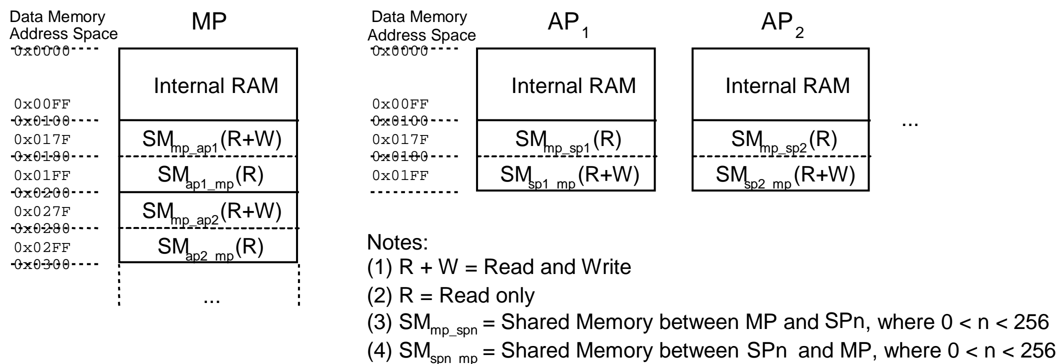

Fig. 5. Data memory address spaces of MP and SPs.

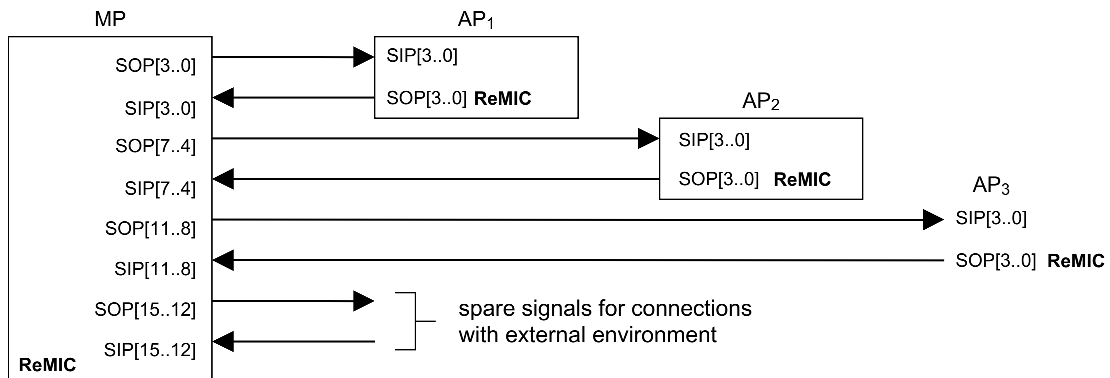

Fig. 6. Signal connections between MP and APs.

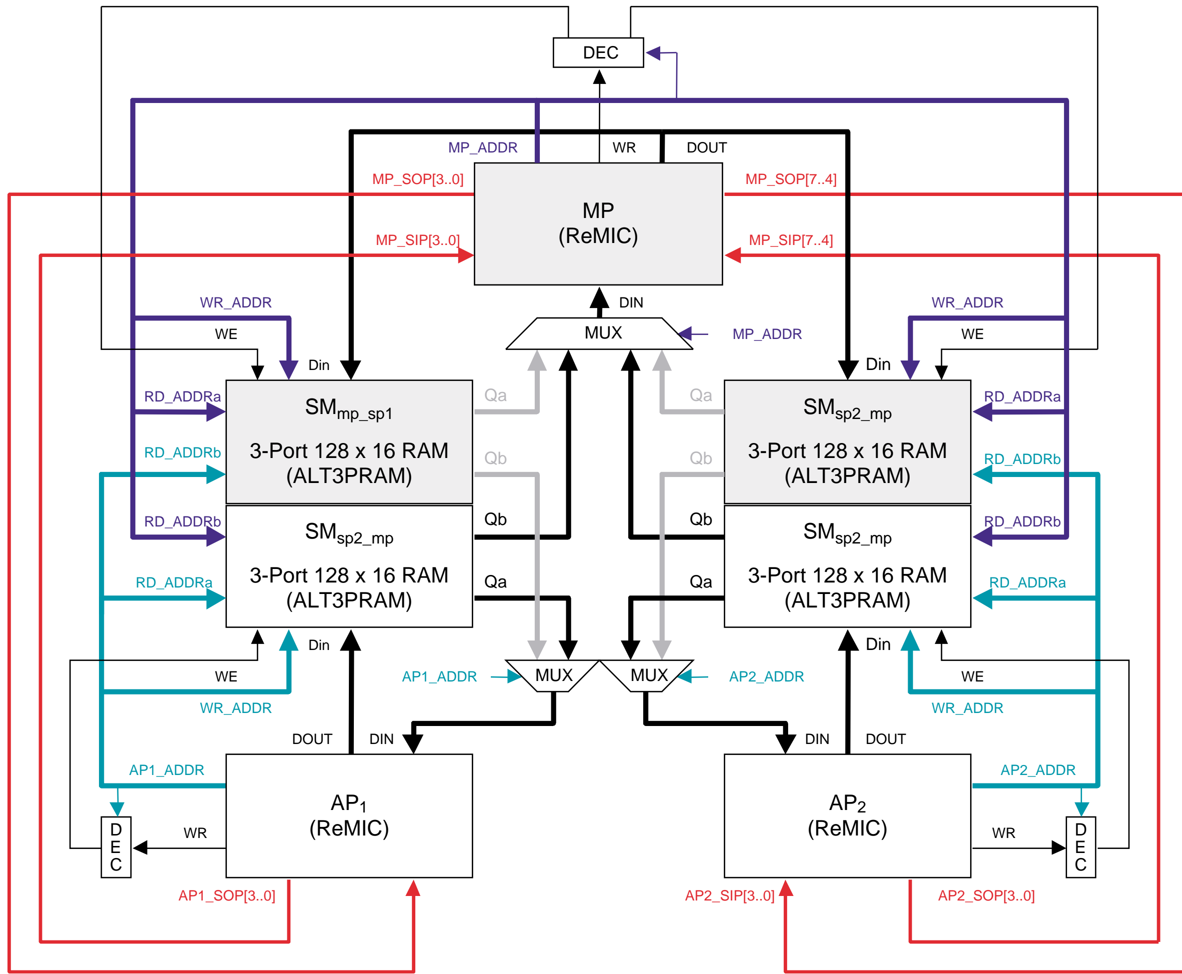

Fig. 7. HiDRA implementation with one MP and two APs.

correct and the searched address pattern is found, the system generates an output signal called ADDR_MATCH. Also, if anytime during the computation process an external RESET signal is activated, system must be initialized and restarted waiting for another packet. In [5], this behavior is described in ECL using three ECL modules called *packet reception*, *CRC check* and *address matching*. Authors have tried two different implementations namely a synchronous implementation (using an FSM for the whole protocol stack) and an asynchronous one (with modules communicating via signals).

The same functionality may be implemented using different number of concurrent processes. An implementation with two processes that run on two processor cores is shown in Fig. 8. The CRAL program that describes the application is shown below (in Fig. 10) and follows the conceptual solution from Fig. 8. The HiDRA hardware configuration, shown in Fig. 9, is automatically extracted from the CRAL program. Those two cores execute two behaviors as conceptually shown in Fig. 8. The first process is executed on the master processor and the second on the application processor. Those behaviors are called master and application process, respectively.

The role of the master process is to receive input bytes by synchronizing with the external environment, assemble them into a packet and calculate the CRC. As soon as the first packet is assembled, the application process is activated to start a lengthy computation for address matching. If the calculated CRC is not correct, CRC_NOK signal is generated and sent to the slave process to abort further computation. Otherwise, it is allowed to finalize address matching and inform the master process if address-matching process was successful. At the same time master process continues to receive the next packet and that behavior continues without termination. Avoidance of the loss of packets is achieved by synchronization of the master and application processes as shown in Fig. 8. The CRAL source code describing the above behavior is shown in Fig. 10. Fig. 11 is the illustration of the final code for the behavior on the master and slave processor core generated by the reactive concurrent code generator (RCCGEN, which is discussed in the next section) from the source code description of Fig. 10.

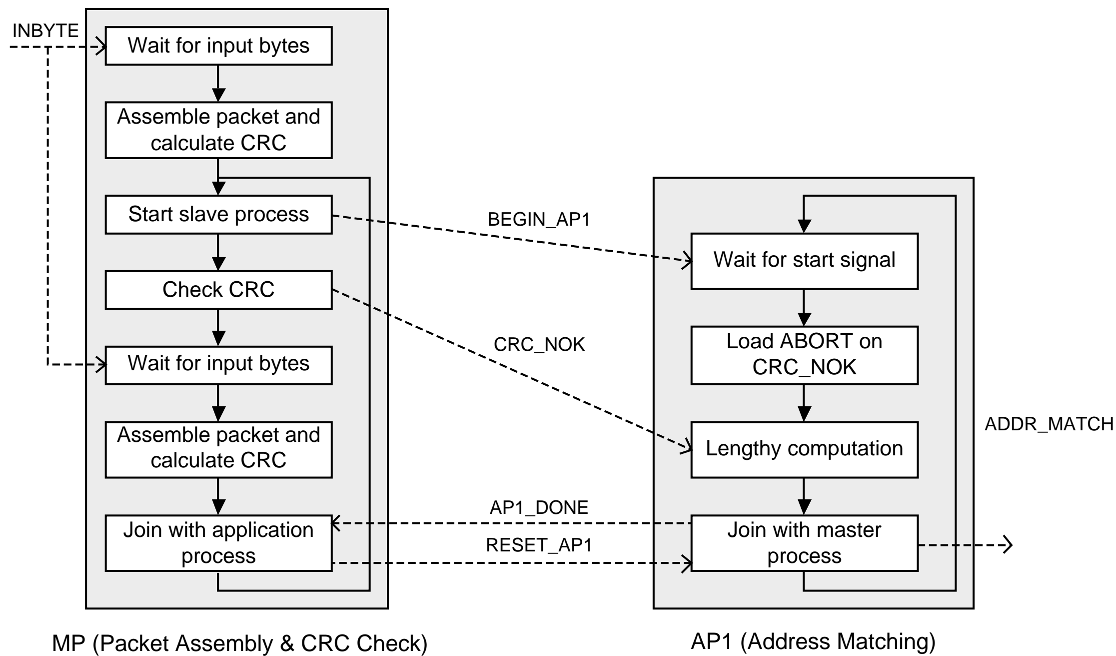

Fig. 8. Protocol stack conceptual solution.

7. Development tools

7.1. Design flow

In this section, a brief description of the design flow and current development support for HiDRA based applications is presented. CRAL language has been introduced to simplify the use and programming of HiDRA by providing native support for concurrent execution on multiple processor cores, as well as synchronization of concurrent processes. The source CRAL description is used for two main purposes: (1) generation of code that will be executed on individual processor cores (master and application processors) and (2) instantiation of cores and the overall architecture. The overall system design flow is shown in Fig. 12.

A program called, *R*eactive *C*oncurrent *C*ode *Gen*erator (RCCGEN), is developed to help the designer identify the number of processors required for a CRAL design and generate the corresponding assembler files for each of the used processors. Once assembler files are generated they are automatically assembled by ReMIC assembler and stored in an appropriate file format that enables loader to download files to the local memories of individual cores. The ReMIC assembler itself is a product of assembler generator that is automatically produced when the ReMIC original instruction set is changed by adding new or removing existing instructions. CRAL program contains other important information for instantiation of ReMIC cores (number of cores and their individual features such as number of input and output signals and memory size). This information is extracted from the CRAL program and forwarded to the HiDRA generator, which generates a VHDL source description of the HiDRA multiple processor system. Typical design flow further includes

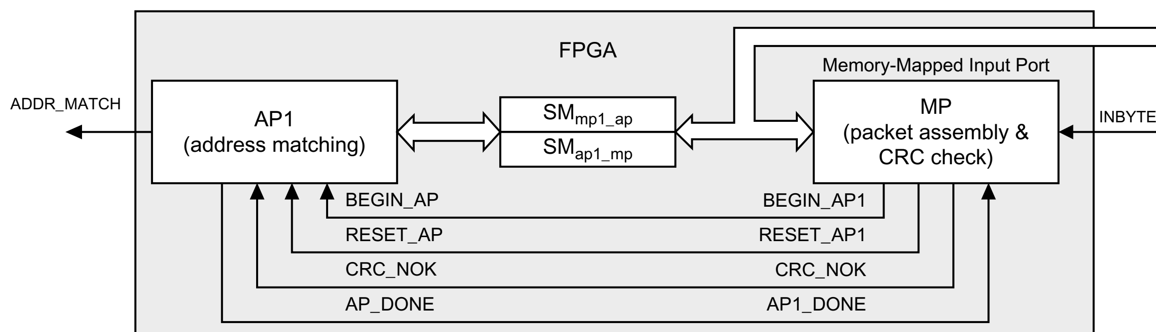

Fig. 9. HiDRA architecture generated for protocol stack example from the CRAL program.

```
&MAIN_BEGIN; Main processor

; Assignments of ports and symbol
; definitions not shown
; RESET signal used for reset of the
; behavior

RESET ABORT @INRST_SIG AGAIN

; Assemble the first packet
      JSR ASSM_PKT

; Start the parallel prochdr process.
; synchronization is required between
; master and the prochdr process to
; prevent packet overrun problem.

PROC_PKT START AP PROCHDR ; initialize AP

; check crc
      JSR CHECK_CRC

; assemble new packets
      JSR ASSM_PKT

; Synchronize with the slave processor
; and continue only if the lengthy
; computation is done.

      JOIN AP PROCHDR

; restart the prochdr process
      JMP PROC_PKT

; initialize slave
AGAIN EMIT #AP_RESET
      JMP RESET

; subroutine: assemble packet
ASSM_PKT        …
; More instructions
; Wait for next byte
B_CNT SAWAIT @INBYTE_SIG
; More instructions
PKT_DONE RET

; subroutine: check crc
CHECK_CRC       …
; More instructions
      EMIT #CRC_NOK
CRC_DONE RET

&MAIN_END
```

```
&SPR_BEGIN PROCHDR; Application processor
; Assignments of ports and symbol
; definitions not shown

RESTART ABORT @AP_RESET RESET
        ABORT @CRC_NOK HDR_DONE

; some lengthy computation determining
; the value of match_ok
      EMIT #ADDR_MATCH
HDR_DONE NOOP
RESET NOOP

&SPR_END
```

Fig. 10. Snapshot of CRAL source code for protocol stack example.

synthesis and downloading to the target FPGA device and start up of HiDRA operation. The other option is simulation of the design in the VHDL simulation environment (ModelSim used in this case), which is not shown in Fig. 12.

Application programs will be loaded from the program file with the assistance of the hardware loader as described in the rest of this Section. In this way, program downloading is separated from design configuration downloading and these two processes can be repeated independently. This further means that there is no need for repeated synthesis if only the CRAL program is changed and no change of system parameters is done and vice versa. This is especially important during system development and prototyping, as it speeds up the overall process.

7.2. FPGA testbed

A testbed for HiDRA implementations on Altera standard prototyping board is shown in Fig. 13. A UART-based loader module is added to the HiDRA implementation so that the test program can be directly downloaded into the FPGA from the MLoader program through a serial link.

Fig. 14 shows the UART-based loader module for HiDRA. Additional Write Enable (WE) signals are added to support programming of local program memories. Also, all the processors are held in their reset state during the programming process and released at the same time once all the program memories are programmed.

```
;MASTER & SLAVE COMMUNICATION SIGNALS;

BEGIN_SPR0      &EQU $1 ;added by RCCGEN
ACK_SPR0        &EQU $2 ;added by RCCGEN
DONE_SPR0       &EQU 0 ;added by RCCGEN

;MASTER CODE BEGIN;

; DECLARATIONS AND DEFINISTIONS NOT SHOWN

RESET   ABORT @INRST_SIG AGAIN

; assemble the first packet
        JSR ASSM_PKT

; start the parallel prochdr process on slave
PROC_PKT EMIT #START_AP0; added by RCCGEN

; check crc
        JSR CHECK_CRC

; assemble new packets
        JSR ASSM_PKT

;Synhronize with PROCHDR on AP JOIN
        SAWAIT @DONE_AP0; RCCGEN
        EMIT #ACK_AP0; RCCGEN

        JMP PROC_PKT

; restart AP if RESET
AGAIN   EMIT #AP0_RESET; added by RCCGEN
        JMP RESET

; subroutine: assemble packet
ASSM_PKT …
;----- as before
ASSM_DONE RET

; subroutine: check crc
CHECK_CRC …
;----- as before
CRC_DONE      RET
&END ;added by RCCGEN
;MASTER CODE END;
```

```
;SLAVE & MASTER COMMUNICATION SIGNALS;

START_AP0       &EQU 0; added by RCCGEN
DONE_AP0        &EQU $1; added by RCCGEN
ACK_AP0         &EQU 1; added by RCCGEN

; AP CODE BEGIN
; ORIGINAL SIGNAL DECLARATIONS AND SYMBOL
; DEFINISTIONS SKIPPED

;SLAVE PROCESS PROCHDR CODE BEGIN
RESTART  ABORT @AP0_RESET RESET
CONT0    ABORT @ACK_AP0 CONT1; by RCCGEN
         SAWAIT @START_AP0; by RCCGEN

         ABORT @CRC_NOK HDR_DONE

; some lengthy computation determining
; the value of match_ok

         EMIT #ADDR_MATCH
HDR_DONE NOOP
         SUSTAIN #DONE_AP0; by RCCGEN
CONT1    JMP CONT0 ; by RCCGEN
RESET    JMP RESTART
&END; added by RCCGEN
;SLAVE PROCESS PROCHDR CODE END
```

Fig. 11. Code generated by RCCGEN.

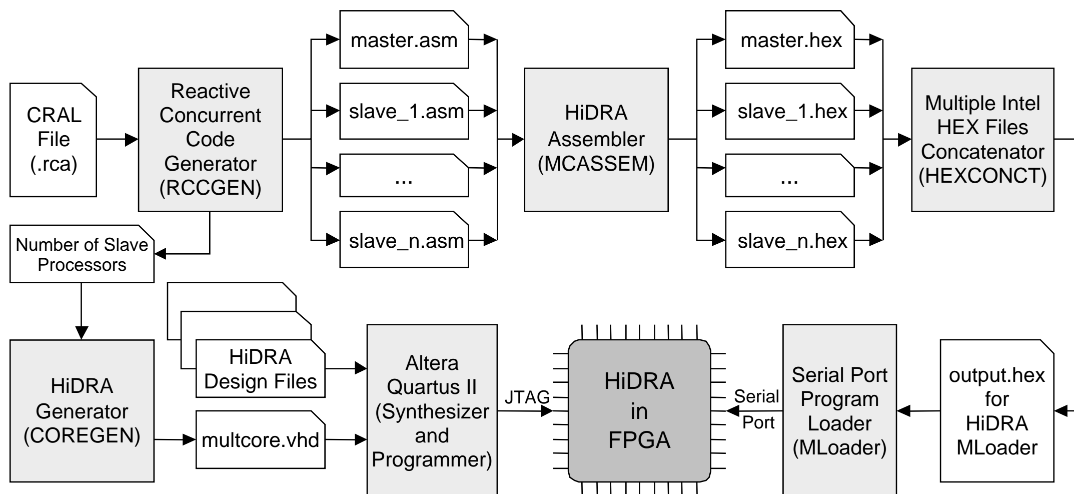

Fig. 12. HiDRA system design process.

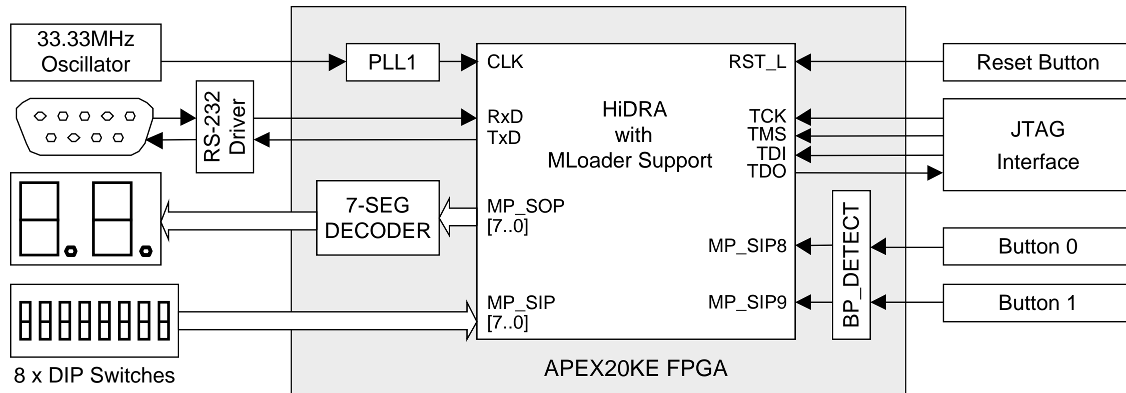

Fig. 13. HiDRA test on excalibur board.

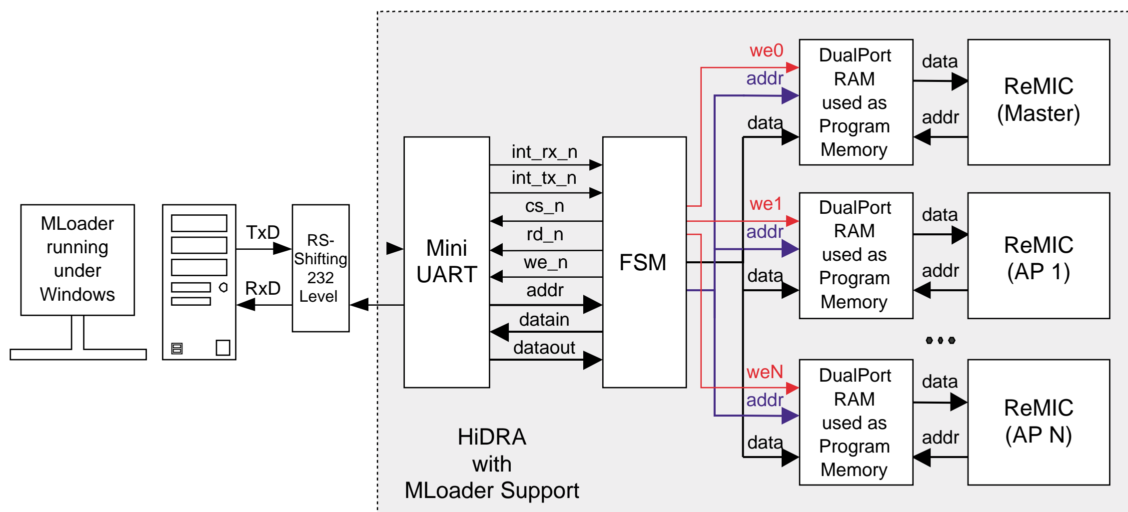

Fig. 14. HiDRA with UART-based programming module.

The Loader facilities on FPGA side are implemented completely in hardware by using a simple UART and an FSM that has dual function: (1) to control the UART and communication between development PC and MLoader program and (2) to implement the loader which interprets the received file from development environment and loads program code segments into SRAM program memories of individual cores. A support for bigger external memories of the ReMIC core has been developed, too, but the full prototyping environment limited with capacities of used Altera prototyping boards allow their very limited use (for one core only).

8. Performance results

ReMIC has been designed using a classical design flow. First, cycle-accurate VHDL model has been developed and thoroughly simulated. ReMIC design has been synthesized for the FPGA implementation. The results of synthesis and some program size comparisons between non-reactive and reactive cores and programs that implement the same functionality using standard microprocessors are shown in

Table 2
Quantitative results and comparisons

EP20K200EF device	Non-reactive core	REMIC
Logic elements	1240	2018
Maximum system clock (MHz)	29.90	21.70

Application	Abort level	Non-reactive core (words)	REMIC (words)
Seat belt controller	1	23	6
Pump controller	2	60	13
Elevator controller	0	86	25
Traffic light controller	2	68	16

Application	ReMIC (16-bit words)	8051 (bytes)	68HC11 (bytes)	NIOS16 (16-bit words)
Pump controller	26	35	56	80
Elevator controller	50	45	79	116
Traffic light controller	32	70	114	147

Table 3
Example of synthesis results for HiDRA implementations

EP20K200EFC484-2	ReMIC	MP+1 AP	MP+2 APs	MP+3 Aps	
Logic elements	Number	2018	4271	6211	8008
	Percentage (%)	25	52	75	96
Memory bits	Number	20480	49152	77824	106496
	Percentage (%)	20	47	74	100
Max system clock		21.7 MHz	20.6 MHz		

Table 2, where data for the non-reactive part of the processor are also provided. More details on the applications used for comparison can be found in [17].

The HiDRA architecture has been first implemented in standard FPGA circuits. The ReMIC processor, in its full configuration, requires around 2000 logic elements as shown in Table 3. The smallest FPGA device from Altera APEX family [22] may accommodate the master processor and several cores, which may operate at frequency around 20 MHz. As shown in the table, the usage of the logic elements per core decreases slightly as the number of AP increases. The system clock for all three implementations is around 20 MHz, which is slightly slower than a single ReMIC implementation. The faster implementations (more than 50 MHz) have been achieved when using STRATIX II FPGA devices.

Simulation and real runs results for the protocol stack example presented in Section 6 are shown in Table 4. This table also compares results with those from [5] when using MIPS R3000 microprocessor. It must be noted that these comparisons are only provisional, as the full conditions of the compiler used and experiments from [5] are not known. We have assumed that input bytes coming to the system are always available (no wait for external events). The proposed approach leads to a speedup of 3.8 times on an average and a huge reduction in code size compared to the implementation in [5].

9. Conclusions and future work

Though heterogeneous embedded systems are all pervading and the design cycle time is shrinking, execution platforms for rapid prototyping of such systems are still missing. In this paper, we proposed multiple processor architecture for heterogeneous embedded systems. The proposed architecture supports globally asynchronous locally synchronous (GALS) models of concurrency, combined control-flow and data-flow behaviors and hardware–software implementations all under one framework. A new instruction set architecture (ISA) called concurrent reactive assembler language (CRAL) is developed for heterogeneous embedded systems. CRAL supports CSP like handshaking communication and synchronization among concurrent processes at the assembly language level. Such communication and synchronization mechanisms are efficiently implemented using the low-level reactive instruction set of the REMIC processors. We demonstrate the efficiency of the approach using a protocol stack example from [5]. Our future work will focus on improving several limitations of the current status of our work. One of the major goals is to provide high-level programming language support in form of extended C/C++ and then to consider other system design language (Esterel, SystemC) compilers that target the HiDRA architecture. Also, we are working on a simulation environment suitable for architecture exploration of different architectural options when implementing concrete heterogeneous embedded applications.

Table 4
Memory and timing performance of the example application

Implementation	Data mem. (bytes)	Code size (bytes)		Exec. time (cycles)	
		MP	SP	MP	SP
HiDRA	64	160	32	810	30
HiDRA (total)	64	192		1113	
MIPS (1 task)	160	1008		4283	
MIPS (3 tasks)	352	1632		4161	

References

[1] S. Edwards, et al., Design of embedded systems: formal models, validation and synthesis, Proc. IEEE 85 (3) (1997).

[2] J.M. Daveau, et al., Protocol selection and interface generation for hardware–software codesign, IEEE Trans. VLSI Syst. 5 (1) (1997) (March).

[3] G. Berry, G. Gonthier, The esterel synchronous programming language, Sci. Comput. Program. 19 (1992) 87–152.

[4] D. Harel, Statecharts: a visual formalism for complex systems, Sci. Comput. Program. 8 (1987) 231–274.

[5] L. Lavagno, E. Sentovich, ECL: a Specification Environment for System-Level Design, Design Automation Conference DAC, 1999.

[6] E.A. Lee, D.G. Messerschmitt, Synchronous dataflow, Proc. IEEE 75 (9) (1987) 1235–1245.

[7] N. Halbwachs, P. Caspi, D. Pilaud, The synchronous dataflow programming language lustre. Another look at real-time programming, Proc. IEEE (1991).

[8] F. Balarin, et al., Hardware Software Codesign of Embedded Systems: the POLIS Approach, Kluwer Academic Press, Dordecht, 1997.

[9] G. Berry, S. Ramesh, R.K. Shyamasundar, Communicating reactive processes, Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (CRP), 1993 pp. 85–98.

[10] S. Ramesh, Communicating Reactive State Machines: Design, Model and Implementation, IFAC Workshop on Distributed Computer Control Systems, Pergamon Press, NY, USA, 1998.
[11] M. Keating, P. Bricaud, Reuse Methodology Manual for System on a Chip Design, Kluwer Academic Publishers, NY, USA, 1999.
[12] A. Iyer, D. Marculescu, Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors, International Symposium on Computer Architecture, IEEE, 2002.
[13] T. Grotker, System design with SystemC, Kluwer Academic Publishers, NY, USA, 2002.
[14] D.D. Gajski, et al., SpecC Specification Language and Methodology, Kluwer Academic, NY, USA, 2000.
[15] Z. Salcic, P. Roop, D. Hui, I. Radojevic, HiDRA: a New Architecture for Heterogeneous Embedded Systems, Proceedings of Embedded Systems and Applications ESA-04, CSREA Press, Las Vegas, 2004. pp. 164–170.
[16] Z. Salcic, D. Hui, P. Roop, M. Biglari-Abhari, REMIC—Design of a Reactive Embedded Microprocessor Core, accepted, to be presented at Asia-South Pacific Design Automation Conference ASPDAC, Shanghai 2005 p. 5.
[17] Z. Salcic, P. Roop, M. Biglari-Abhari, A. Bigdeli, REFLIX: a processor core with native support for control dominated embedded applications, J. Microprocess. Microsyst. 28 (2004) 13–25.
[18] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall International, Englewood Cliffs, NJ, USA, 1985.
[19] R. Milner, Communication and Concurrency, Prentice Hall International, Englewood Cliffs, NJ, USA, 1989.
[20] Z. Salcic, D. Hui, P. Roop, M. Biglari-Abhari, REMIC—design of a Reactive Embedded Microprocessor Core, Proceedings of Asia-South Pacific Design Automation Conference ASP-DAC, Shanghai, January, 2005.
[21] P. Roop, Z. Salcic, S. Dayaratne, Towards Direct Execution of Esterel Programs on Reactive Processors, Embedded Software Conference, EMSOFT'04, Pisa, Italy, September, 2004 pp. 27–29.
[22] The Altera FPGA Family. Details from http://www.altera.com.
[23] Mentor Graphics ModelSim, http://www.mentor.com.