

**Department of Electrical and Computer Engineering
University of Auckland, New Zealand**

**RePIC – A New Processor Architecture
Supporting Direct ESTEREL Execution**

**C.M. Edmund Chow
Joyce S.Y. Tong
M.W. Sajeewa Dayaratne
Partha S Roop
Zoran Salcic**

Abstract:

Reactive embedded applications require fast reaction to external events (defined as reactivity in this paper). Esterel is a system-level language for the modelling, verification and synthesis of reactive systems through high-level support for efficient interaction with environment, synchronous communication and concurrency. Existing compilers for Esterel compile it to intermediate C code, which preserves Esterel semantics, before generating machine code that runs on a specified processor. Hence, the resultant code is often huge and inefficient, even for very small programs, as the underlying processors have no direct support for reactivity needed to execute Esterel. This paper proposes, for the first time, a reactive microcontroller called RePIC (which is an extension of the commercial PIC microprocessor). RePIC supports direct Esterel execution through a reactive instruction set architecture (ISA), while preserving Esterel semantics. A new benchmark suite for comparing the reactive performance of processors called the Auckland Real-Time Benchmark (ART-Bench) is also presented and used in comparing RePIC with the original PIC to demonstrate the significant improvement in the performance of reactive applications.

1	INTRODUCTION	5
2	RELATED WORK.....	6
3	BACKGROUND	9
3.1	Target Processor for Extensions	9
3.2	Esterel	11
3.3	Implementation in original PIC	12
4	ARCHITECTURAL SUPPORT FOR REACTIVITY IN REPIC	14
4.1	RePIC Core Features.....	14
4.2	Instruction Set Architecture for RePIC	17
4.2.1	EMIT – Signal Emission.....	17
4.2.2	SUSTAIN – Signal Sustenance.....	18
4.2.3	TAWAIT – Delay.....	18
4.2.4	SAWAIT – Signal Polling	18
4.2.5	CAWAIT – Conditional Signal Polling.....	19
4.2.6	PRESENT – Signal Presence	20
4.2.7	ABORT – Preemption	21
4.2.8	SETINTMR – Internal Timers	22
4.3	Summary – New Reactive Instructions.....	23
4.4	ATM Example in RePIC.....	23
5	PROTOTYPE DESIGN	24
5.1	Additional I/O Ports.....	24
5.2	Register mapping	24
5.2.1	SIGINx and SIGOUTx Registers	24
5.2.2	INTMRx Registers	25
5.2.3	INTMRCOND Register	25
5.3	Implementation of New Reactive Instructions	26
5.3.1	EMIT & SUSTAIN.....	26
5.3.2	TAWAIT.....	26
5.3.3	SAWAIT & CAWAIT.....	26
5.3.4	PRESENT	27
5.3.5	ABORT	28
5.3.5.1	ABORT Activation.....	28
5.3.5.2	ABORT Handling.....	29
5.3.5.3	Termination of ABORT	31
6	PRESERVING THE SYNCHRONY HYPOTHESIS OF ESTEREL	33

7	VARIOUS TOOLS AND LANGUAGES USED.....	34
7.1	VHSIC Hardware Description Language	34
7.2	Esterel	34
7.3	Lex and Yacc.....	35
8	SYNTHESIS RESULTS.....	35
9	BENCHMARKING.....	36
9.1	ART-Bench – A New Benchmark Suite	37
9.2	The Benchmarking Process.....	37
9.2.1	Comparison using manual translation of Esterel	38
9.2.2	Comparison of RePIC versus traditional Esterel compilers	39
10	CONCLUSIONS AND FUTURE WORK.....	41
11	REFERENCES	41

1 Introduction

Embedded systems are application specific digital systems that continuously interact with their immediate environment and react to external events (also called reactive systems). Esterel is a synchronous language, which is used for the specification [1], verification [2] and synthesis [2] of large reactive embedded applications such as aircraft flight controllers. Recently, Esterel is attracting a lot of attention from design automation community [3] due to its clean semantics that not only enables automatic design but also formal verification [2] of safety-critical embedded applications. Esterel supports the modelling of delay, preemption, signal emission, synchronous communication, software interrupts and synchronous and asynchronous data handling.

Esterel Compilation	Advantages	Disadvantages
Hardware implementation	<ol style="list-style-type: none"> 1. Reactive behaviours mapped onto FSMs 2. Small footprint and cheap implementation 3. Supports real concurrency 	<ol style="list-style-type: none"> 1. Higher cost. 2. Loss of flexibility (each Esterel program and any modification requires resynthesis). 3. More difficult to map the C – programs used in data handling..
Software Implementation	<ol style="list-style-type: none"> 1.Lower cost. 3. More flexible (same microcontroller can run any Esterel program) 4. Easy mapping of data-handling code. 	<ol style="list-style-type: none"> 1. Large footprint (memory requirement) 2. Scheduling overhead for simulation of concurrency. 3.Overhead for mapping of signals and sensors. 4. Complex compilation process 5. Overhead for abort translation. 6. Overhead for priority resolution. 7. Emulates parallelisms by serialization of concurrent activities
RePIC and Direct Code Generation	<ol style="list-style-type: none"> 1.Same cost as Software. 2. No overhead for signal mapping. 3. No overhead for abort. 4. No overhead for priority resolution. 5. Fast reaction and response times compared to software. 6. Very compact code compared to software. 7.Higher flexibility compared to hardware. 8.Limited concurrency support through dual-processor architecture. 	<ol style="list-style-type: none"> 1.Lower efficiency compared to hardware. 2. Complex approach to concurrency support compared to direct hardware. 3. Scheduling overhead for simulation of concurrency for programs with more than 2 threads.

Conventionally, Esterel models of embedded systems are either synthesized as software running on a microcontroller [4-6], hardware [7, 8] or a combination of the two using codesign as in POLIS [9]. Pure software implementation involves the generation of intermediate code (in C, for example) that is subsequently assembled to

generate the desired machine code. Such mapping of Esterel to intermediate code is quite inefficient due to the requirement of indirectly simulating Esterel concurrency (through some form of scheduling), inefficient mapping of signals and sensors to ports on the microcontroller (through additional reaction code), indirect mapping of aborts and traps through interrupts and polling. Thus, the generated code has huge memory footprint even for simple applications rendering Esterel ineffective for small hand-held embedded devices.

Direct compilation of Esterel to hardware obviously results in quite efficient realization supporting direct concurrency (unlike simulated concurrency in software). Pure hardware implementation also has some problems like loss of flexibility (new hardware needs to be synthesized for every Esterel program) and higher cost. Hardware-software solutions using codesign have been also proposed through the POLIS tools. POLIS, however, uses conventional microcontroller (such as PIC and HC11) for code generation using automata compilers. This has the same limitations of software compilers mentioned earlier.

This paper proposes an intermediate approach by extending a commercial microcontroller (PIC [1]) to support direct Esterel execution so that the intermediate code generation step can be bypassed. This is achieved by incorporating architectural extensions to PIC so that Esterel can be executed directly on the new processor (RePIC, denoting Reactive PIC) preserving the semantics of the language. A dual processor RePIC based architecture is also developed to demonstrate a possible way to directly support Esterel concurrency. Table 1 compares the proposed approach with existing approaches of Esterel compilation.

Recently, a processor called REFLIX [10], based on an open experimental core, has been proposed to provide direct support for mapping reactive embedded applications. REFLIX provides direct support for pure signals, preemption, signal emission and delay. It is demonstrated through a set of simple reactive programs that REFLIX performs much faster and generates more compact code compared to the original core. REFLIX, however, has no support for direct execution of Esterel and does not preserve Esterel semantics. Hence, Esterel programs could not be mapped on to REFLIX directly. Moreover, REFLIX had no support for handling real concurrency. As REFLIX did not support Esterel semantics, only a limited benchmarking comparison was done.

2 Related Work

Esterel models of embedded systems can be implemented in either hardware, software or a mixture of both. A software implementation of an Esterel model usually results in assembly code being generated for a given General Purpose Processors (GPP) while a purely hardware implementation results in an Application Specific Integrated Circuit (ASIC). However, an intermediate option in terms of performance and flexibility is to

map such a model into an Application Specific Instruction Set Processor (ASIP). ASIPs, by definition have an architecture and an instruction set which is optimised for a given application (in this case a given model of the system) hence ASIPs provide a far more optimized solution than GPPs [10]. The main strength of ASIPs is the fact that they have been customised for a given application, although this results in their improved performance and lower silicon use, it also becomes their greatest weakness. Due to the close relationship between the architecture of the processor and the target application, changes to the software cannot always be made, even on occasions where such changes are possible they result in severe performance penalties [11, 12]. Mass production of ASIPs in most cases is not commercially viable due to the limited nature of its application resulting in ASIP having larger production costs compared to general purpose processors. These together with the need to custom build or modify development tools such as compilers results in high total cost of implementing ASIPs.

Software implementations on general purpose processors also have its disadvantages when it comes to reactive or control dominated systems. Reactive systems need to respond to events from its environment, and in GPPs these external signals are checked by either using polling or by using interrupts (as seen in Figure 1).

Polling provides a busy-waiting mechanism which checks for the presence of input signals or events on a regular basis. Interrupts however, follow a non-busy waiting concept which only reacts when specific events occur in the environment[13]. Both polling and interrupts do not handle *reactivity*¹ very well.

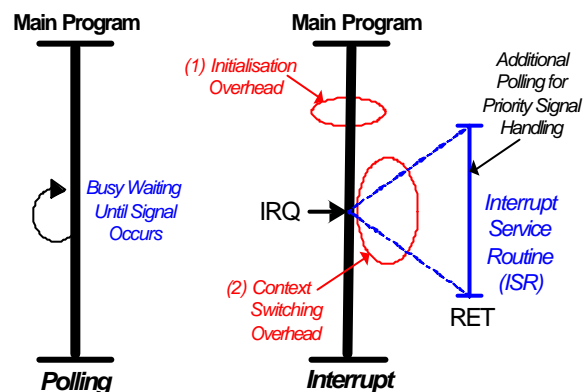


Figure 1 Polling vs Interrupts

Polling is a synchronous mechanism which doesn't efficiently handle asynchronous input events. Although interrupts are asynchronous they suffer from overheads resulting from saving and restoring the context of the parent thread, which hinders the reactivity of interrupts. Furthermore standards interrupts do not handle signal priorities efficiently which results in future performance penalties.

Although software implementations suffer from the problems described above, system level languages such as Esterel inherently support reactivity extremely well. The problems emerge when such a highly reactive language is compiled to an underlying architecture that doesn't support these reactive constructs. Currently the focus has been to implement Esterel compilers which deliver efficient and compact code.

¹ Reactivity is defined as the speed of reaction to an external or internal event

Numerous types of compilers have been presented ranging from automata-based, gate-based to fork-and-join-based compilers (Figure 2).

Automata-based compilers such as Berry et al.'s v3 [2] exhaustively simulate the program and produce very efficient code, but the resulting program size tends to grow exponentially as such compilers produce a separate branching program for each state[14]. Polis [15] is another automata-based compiler which alternatively tries to minimize the code size by sharing code between states using binary decision diagrams [16]. Although this method reduces the code size, it is only practical for small applications with low number of states [14].

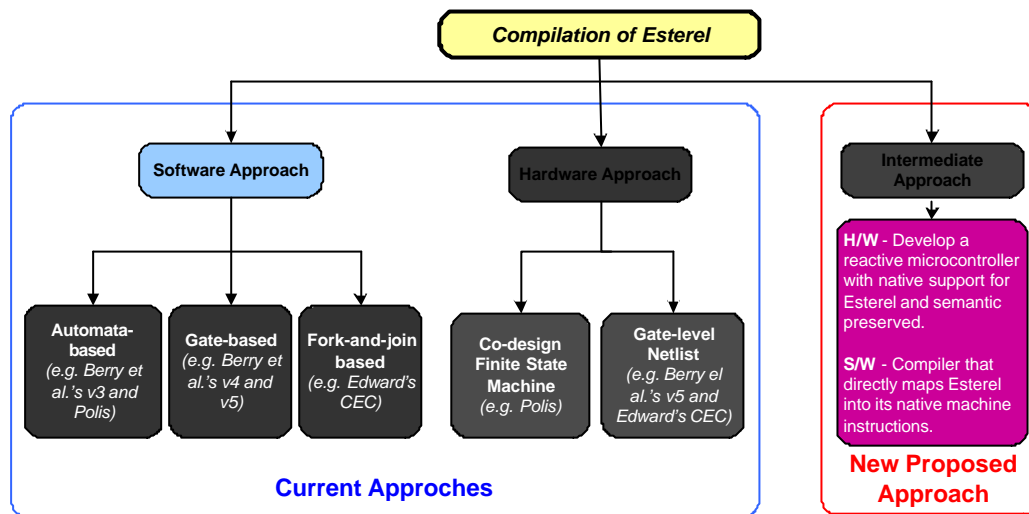


Figure 2 Compilation of Esterel Models

Gate-based compilers such as Berry et al.'s v4 and v5 compilers translate Esterel into a net list of Boolean logic gates and further generate a leveled compiled-code simulator for it [14]. Although this method gives a very compact code size it results in very inefficient code which is in the order of hundred times slower than that generated by automata compilers. This is because the generated code contains many idle sections which consume valuable computation time [14].

Stephen Edward's fork-and-join-based compiler [3] claims to produce the most efficient and shortest code out of all the current compilers. It employs an algorithm that synthesizes a sequential program from the concurrent control-flow graph by using C and fork-and-join statements as an intermediate representation. This in turn avoids the capacity problems of automata compilers and most of the overhead of the gate-based compilers.

Esterel to hardware compilers are broadly categorized into two subsets, the first being the Codesign Finite State Machine approach, used by compilers such as Polis. Such a model is globally asynchronous and greatly simplifies partitioning of the model. They

use S-Graphs as an intermediate step to optimise performance. The second category is gate level net list based compilers such as Esterel v5 and Edward's CEC [7]. Although compiling Esterel applications to hardware results in a drastic increase in performance over its software counterparts, a price is paid when it comes to reusability of the resulting system.

Clearly, there is a tradeoff between reusability and reactivity of the software and hardware approaches. Although the software approach is more commonly exploited, none of the conventional processors truly addresses aspects of reactive applications due to the current structure of standard interrupt mechanism. Large memory footprint could also become a problem to small embedded applications due to the overheads incurred during the intermediate step of generating the C code.

3 Background

This section presents all the background information required for the remainder of this paper.

3.1 Target Processor for Extensions

PIC16F84 is a commercial microprocessor developed by Microchip [1]. PIC is a proven robust design that is specifically targeted at control-dominated embedded applications ranging from high-speed automotive to low-power remote sensors, security devices and smart cards [15]. PIC16F84 is an 18-pin Flash 8-bit processor that has a simple but versatile design. It employs a Reduced Instruction Set Computer (RISC) architecture which utilizes a small, highly-optimized set of instructions (a total of 35 single-word instructions). The separated instruction and data buses of the Harvard architecture in PIC allow two simultaneous memory fetches of a 14-bit instruction word and an 8-bit data word. The employment of two-stage pipelining (fetch and execute) maximizes the utilization of internal resources, increasing the overall throughput of the system. Hence, each PIC instruction effectively executes within a single cycle except for branches. Figure 3 shows the data path of PIC16F84.

Some of its basic features can be summarised as follows:

- RISC architecture
- Harvard architecture with pipelining
- Only 35 single-word instructions
- 14-bit wide instructions
- 8-bit wide data path
- 15 predefined function user registers

- Direct, indirect and relative addressing modes
- All instructions are executed in a single cycle except for program branches which take two cycles

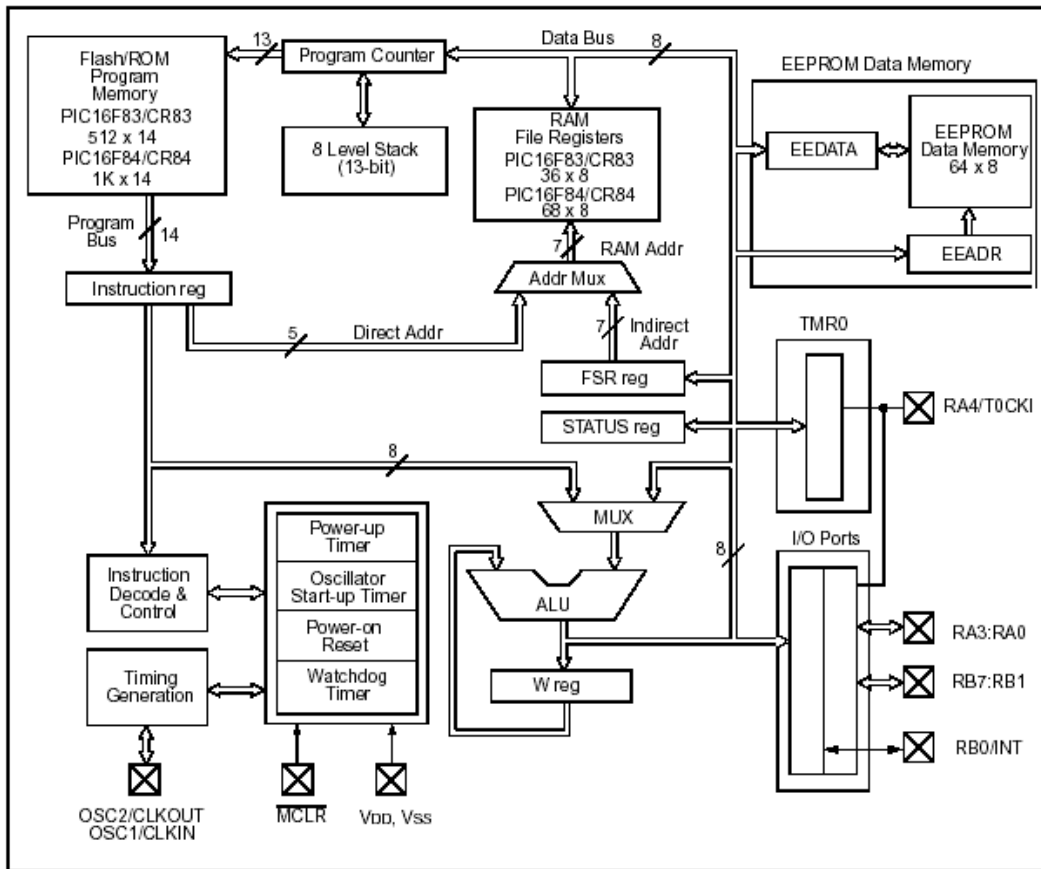


Figure 3 Data path of Microchip's PIC16F84

3.2 Esterel

Esterel is synchronous language which is widely used for modelling embedded systems.. It supports modelling of *delay*, *preemption*, *signal emission*, *synchronous communication between threads and data handling*. Some of these features will be illustrated using the following simplified example of an Automatic Teller Machine (ATM).

```
module atm_controller:

%input signals
  1.input cardInserted, pinEntered, withdraw, sumEntered,
  transactionOK, checkBalance, invalidCard, incorrectPin

%output signals
  2. output insertCard, enterPin, selectOption, processTransaction,
  releaseSum, printReceipt, printReceipt, ejectCard

  3. loop
    4. emit insertCard;
    5. await cardInserted;
    6. weak abort
    7. weak abort
      8. emit enterPin;
      9. await pinEntered;
    10. emit selectOption;
    11. await
      12. case withdraw do
        14. await sumEntered;
        15. emit processTransaction;
        16. await transactionOK;
          17. emit releaseSum;
        18. emit printReceipt;
      13. case checkBalance do
        19. emit processTransaction;
        20. await transactionOK;
        21. emit printReceipt;
          end await
        when incorrectPin;
        when invalidCard;
    22. emit ejectCard;
  end loop
end module
```

Any Esterel program is a collection of modules, each of which is a basic programming unit. Each module has an interface declaration part which declares the signals and sensors in the environment of the program (lines 1 and 2). Signals carry pure control information and are either present or absent in a given tick.

Following the interface declaration part is the body of the program. Being a reactive program, the body is an infinite loop that encloses the main behaviour of the ATM (line 3). Lines 4 and 5 model the emission of signal `insertCard` (for one tick) and then

delay until a card is inserted (using the await statement). While emit is instantaneous like a combinational logic block in hardware, await has at least one tick delay and is like sequential logic blocks. Lines 6 to 22 model the actual behaviour of a typical ATM. This behaviour has two parts: normal behaviour (lines 8 to 21) when a valid card is inserted and the associated PIN is correct or abnormal exception handling (line 22) when either an invalid card or incorrect PIN is detected.

One approach to handle exceptions in Esterel is through the weak abort construct. A weak abort construct kills its body whenever a combination of signals (specified as abortion condition) is present in the environment. In the ATM example, the outer abort statement (line 6) kills its body (lines 7 to 21) when an invalidCard is entered. In this case, control jumps to line 22 directly. Abort incorporates preemption with static priority through nesting of aborts where outer aborts take precedence. In the ATM example, if both incorrectPIN and invlaidCard are simultaneously present then the outer abort takes precedence.

In addition to these features, Esterel supports the notion of synchronous broadcast communication among concurrent threads. Synchrony implies that an input and the corresponding output occur at the same time instant. Moreover, whenever any output is generated, it is synchronously broadcasted to all concurrent threads that may lead to a sequence of outputs all of which have the same time stamp.

3.3 Implementation in original PIC

The same behaviour as discussed in section 3.2 can be implemented using the original PIC instruction set as shown below.

```

BSF INTCON RBIE
loop: BSF insertCard      # emit insertCard
      BCF insertCard
L0:   BTFSS cardInserted # await cardInserted
      GOTO L0
      BSF INTCON GIE     # Enable global interrupts
      BSF enterPin      # emit enterPin
      BCF enterPin
L1:   BTFSS pinEntered   # await pinEntered
      GOTO L1
      BSF selectOption  # emit selectOption
      BCF selectOption
L2:   BTFCS withdraw    # await case withdraw
      GOTO L3
      BTFCS checkBalance # await case checkBalance
      GOTO L5
      GOTO L2
L3:   BTFSS sumEntered   # await sumEntered
      GOTO L3
      BSF processTransaction # emit processTransaction
      BCF processTransaction
L4:   BTFSS transactionOK # await transactionOK
      GOTO L4

```

```

        BSF releaseSum          # emit releaseSum
        BCF releaseSum
        BSF printReceipt       # emit printReceipt
        BCF printReceipt
        GOTO L7
L5:    BSF processTransaction  # emit processTransaction
        BCF processTransaction
L6:    BTFSS transactionOK     # await transactionOK
        GOTO L6
        BSF printReceipt       # emit printReceipt
        BCF printReceipt
L7:    BSF ejectCard          # emit ejectCard
        BCF ejectCard
        GOTO loop

# Interrupt Service Routine (ISR)
# Move PORTB value to W register
INT:   MOVF PORTB, 0
        # Save current PORTB reading
        MOVWF TEMP
        # XOR last PORTB value with TEMP
        XORWF LASTPB, 1

# Check if pin incorrectPin changed
PIN:   BTFSC LASTPB, incorrectPin
        CALL incorrectPin_CHG

# Check if pin invalidCard changed
CARD:  BTFSC LASTPB, invalidCard
        CALL invalidCard_CHG

# Do task for interrupt change on incorrectPin
incorrectPin_CHG:
        MOVF TEMP, 0          # Move TEMP
        MOVWF LASTPB         # to register LASTPB
        BCF INTCON, RBIF# Clear change-in-pin
        GOTO L7              # interrupt flag

invalidCard_CHG:
        MOVF TEMP, 0          # Move TEMP
        MOVWF LASTPB         # to register LASTPB
        BCF INTCON, RBIF# Clear change-in-pin
        GOTO L7              # interrupt flag

```

In this implementation, the main routine performs the normal operation of the ATM controller while the conventional interrupt mechanism is needed to handle the reactive aspects of the application. When an *incorrectPin* or *invalidCard* signal occurs, an interrupt will be triggered by the controller. The “interrupt on change” feature on PORTB allows PIC to support multiple external interrupts, in addition to the only built-in external interrupt. However, additional software overheads are needed to determine which of PORTB pins causes the interrupt. Also, PIC provides no such instructions as PUSH and POP for accessing the system stack. Therefore, a GOTO instruction instead of RETFIE (Return from interrupt) is used, so that control will

return to the correct place in the main program, upon the completion of the interrupt service routine, in an effort to achieve the same behaviour as Esterel's abort.

4 Architectural Support for Reactivity in RePIC

This section gives the details of the architectural extensions made to PIC for better reactivity and direct compilation of Esterel to RePIC machine code.

4.1 *RePIC Core Features*

The RePIC design is obtained by extending PIC in an upward compatible fashion. The instruction length has been extended from fourteen bits to fifteen bits in order to accommodate the new reactive instructions. Other key extensions required to facilitate reactive applications are summarized as follows:

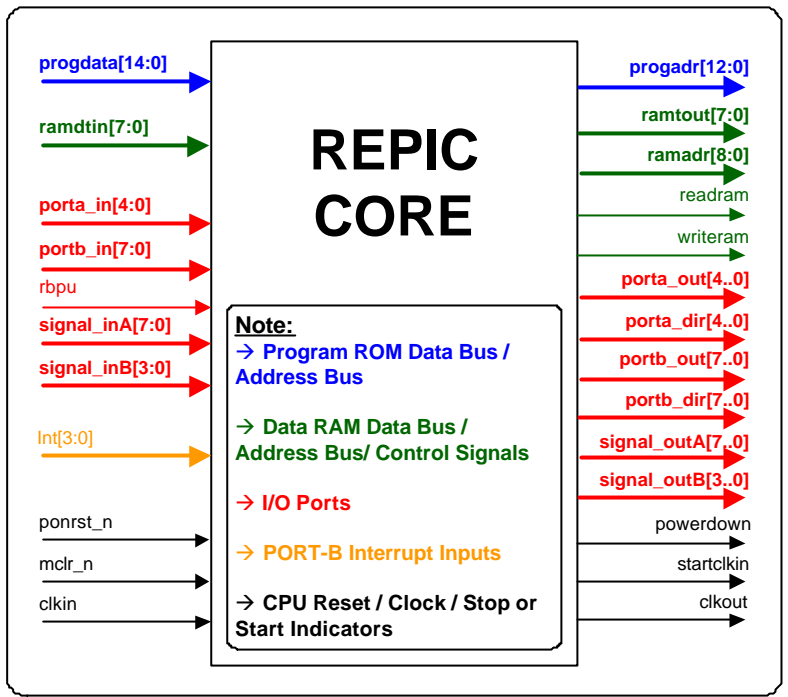


Figure 4 External View of RePIC (Simplified Version)

1. A total of four I/O ports (SIGINA, SIGINB, SIGOUTA and SIGOUTB) have been added, providing a total of 16 single-bit input signal lines (in which 12 are external input signals and the other 4 are internal – for implementing Esterel signals) and 12 single-bit external output signal lines. This enables direct mapping of Esterel pure signals.
2. Introduction of 4 user-programmable internal timers to generate the 4 corresponding single-bit internal input signals upon timer overflows.
3. Abort mechanism is introduced to handle aborts and traps (through weak aborts).
4. Other new instructions to support other Esterel-like reactive instructions are added.
5. Emission of multiple signals within a single instruction cycle is supported in order to preserve the instantaneity principle of Esterel.
6. A variable tick is implemented to preserve the concept of a logical tick in Esterel.
7. A dual-processor architecture is developed to provide support for real concurrency.

A simplified version of RePIC core’s external (interface) view is presented in Figure 4 and Table 1 summarises the major differences between PIC and RePIC.

RePIC Reactive Processor	PIC Conventional Processor
Preserve Esterel semantics through variable tick.	No concept of Esterel tick.
Support both ABORT and interrupt mechanisms. ABORT supports pre-emption with priority.	Only support standard interrupt mechanism.
Support up to 12 external and 4 internal input signals and 12 external output signals in addition to the 13 bi-directional pins available in PIC. The original pins are reserved for receiving/emitting valued signals.	Has only 13 bi-directional pins. These can support up to a total of 13 input/output external signals.
Support emission of multiple signals in a single instruction cycle.	Only one signal can be emitted at a time.
4 internal timers generate 4 internal input signals upon overflow.	No facility for generation of internal signals.
Efficient hardware signal polling mechanisms – CAWAIT and SAWAIT.	Inefficient software implementation for signal polling – loops.

Table 1 The major differences between PIC and RePIC

4.2 Instruction Set Architecture for RePIC

RePIC contains 7 new instructions which complement the ones already available on PIC. These instructions are namely EMIT, SUSTAIN, TAWAIT, SAWAIT, CAWAIT, PRESENT and ABORT.

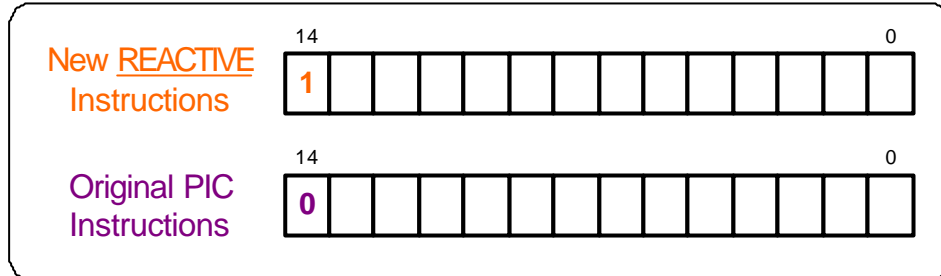


Figure 5 Reactive vs. Original PIC Instructions

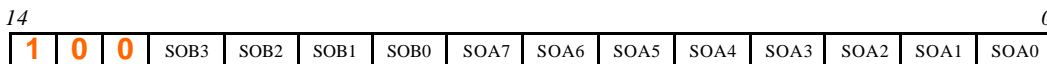
In order to accommodate these new instructions in RePIC, the instruction length had to be modified (because all possible combinations of bits for instruction opcodes have already been used up in PIC). The instruction length is changed to fifteen bits, one bit longer than the original length of fourteen. New reactive instructions are all encoded with this extra bit as *one*, and the old instructions with this bit as *zero*, as shown in Figure 5

4.2.1 EMIT – Signal Emission

EMIT instruction sets a specified signal high for a single tick similar to the Esterel emit statement, except that this instruction can only handle pure signals without any associated values (i.e. no support for valued signals). A pure signal is a binary signal which can either be '1' for ON or '0' for OFF. The first 3 bits of the instruction are used as its opcode, leaving the other twelve bits for specifying the emitted signals. That means at any one time, RePIC can emit up to twelve output signals simultaneously using one hot encoding.

Assembly syntax: EMIT *signal(s)*

e.g. EMIT SOA0, SOB3



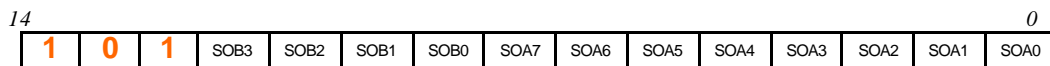
[14:12] Opcode = 'EMIT'

[11:0] specifies the signals to be emitted during the current tick. Each bit corresponds to a pure signal which will be output to one of the output pins on either *signal_outa* port or *signal_outb* port (see Section 5.1).

4.2.2 SUSTAIN – Signal Sustenance

SUSTAIN instruction causes the specified signals to remain high for the entire life of the program. Each signal will be checked in every tick event to ensure that only unsustained signals are cleared, but not those that have been sustained (see Section 5.3.1.1). Similarly, the most significant three bits of the instruction are used for opcode and the other twelve bits are used for specifying the sustained signals.

Assembly syntax: SUSTAIN *signal(s)*
 e.g. SUSTAIN SOA1, SOB0

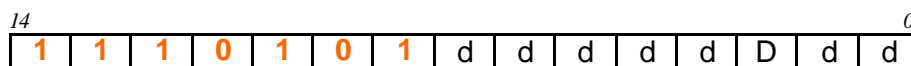


- [14:12] Opcode = ‘SUSTAIN’
- [11:0] specifies the signals to be sustained forever. The twelve signals used here are the same as those introduced in the EMIT instruction. Again multiple signals can be sustained in parallel.

4.2.3 TAWAIT – Delay

TAWAIT instruction causes the system to busy wait until the specified number of clock cycles have elapsed. During delay, program counter will not be incremented and the execution of the next instruction will be stalled until TAWAIT terminates. Also, the busy waiting is “abortable” or interruptible, so that fast response to input events can be made. The opcode is represented by the most significant seven bits and the rest of the bits are used for specifying *delay* where *delay* can be of value between 0 and 255.

Assembly syntax: TAWAIT *delay*
 e.g. TAWAIT 2

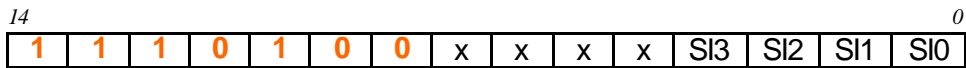


- [14: 8] Opcode = ‘TAWAIT’
- [7:0] specifies the delay or number of instruction cycles

4.2.4 SAWAIT – Signal Polling

SAWAIT instruction polls for a specified signal where the system will busy wait until that signal occurs in the environment. Again, the busy waiting is “abortable”, allowing immediate response to external events. Like TAWAIT, SAWAIT has a 7-bit opcode, but bits seven down to four are Don’t Care unused bits and the least significant four bits are used for specifying the signal to be polled among the sixteen input signals available.

Assembly syntax: SAWAIT *signal*
 e.g. SAWAIT SIA2



[14: 8] Opcode = ‘SAWAIT’
 [3:0] specifies the signal to be polled. The sixteen input signals have been encoded into four bits as listed in the following table:

Input Signal	SI3	SI2	SI1	SI0
SIA0	0	0	0	0
SIA1	0	0	0	1
...	...			
SIA7	0	1	1	1
SIB0	1	0	0	0
SIB1	1	0	0	1
...	...			
SIB7	1	1	1	1

A total of 12 external input signals and 4 internal timer generated signals

SIA0:SIA7 are **external** input signals

SIB0:SIB3 are **external** input signals. SIB4:SIB7 are **internal** signals

Figure 6 Encoding input signals for await instructions

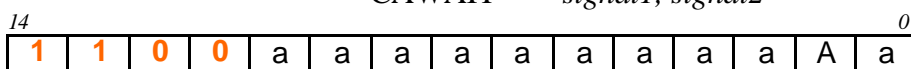
4.2.5 CAWAIT – Conditional Signal Polling

CAWAIT instruction is alike a CASE function with two conditions. This instruction allows polling of two input signals at the same time and system will respond differently depending on which one of the two specified signals has occurred in the environment.

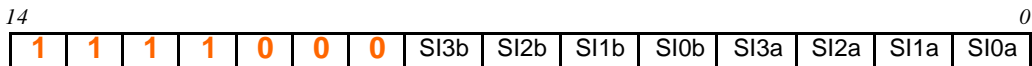
Assembly syntax: CAWAIT *signal1, signal2, address*
 e.g CAWAIT SIB0, SIA4, 64

CAWAIT requires three arguments; they are the two signals being polled, (1) *signal1*{4 bits}, (2) *signal2*{4 bits} and (3) the *address* {11 bits} where the program will jump to if *signal2* is present. Since RePIC’s instructions are only 15-bit wide, the three arguments together with a 4-bit opcode are impossible to fit into one single instruction. Thus, CAWAIT (above) is primarily a pseudo instruction which will further be split into two lines of actual RePIC assembly code during compilation which are shown as follows:

LDCADDR *address*
 CAWAIT *signal1, signal2*



[14:11] Opcode = ‘LDCADDR’
 [10:0] specifies the 11-bit *address*



- [14:8] Opcode = 'CAWAIT'
- [7:4] specifies the second signal being checked – *signal2*
- [3:0] specifies the first signal being checked – *signal 1*

LDCADDR loads the branch address into an internal register for system processing. **CAWAIT** causes the system to busy wait until the presence of *signal1* or *signal2*. If *signal1* occurs, the next instruction will be executed. If *signal2* is present, the instruction at the specified *address* will be executed instead. Otherwise, the system will retain its program counter and the next instruction will only be executed when one of the signals occurs in the environment. In a situation where both *signal1* and *signal2* occur, *signal1* will always take precedence and no branching will be taken.

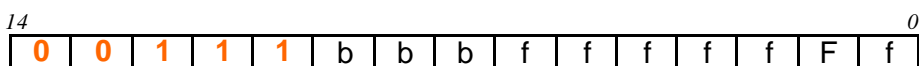
4.2.6 PRESENT – Signal Presence

PRESENT instruction checks if the specified signal is present or not in the current instruction cycle. If the signal is present, the program continues its execution normally. Otherwise, it will branch to the address indicated in the instruction.

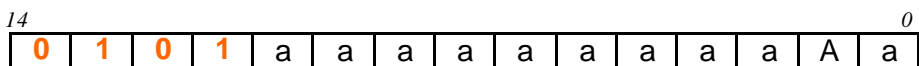
Assembly syntax: PRESENT *signal, address*
 e.g. PRESENT SIA2, 72

Same as CAWAIT, PRESENT is essentially a pseudo instruction which will be implemented by two real RePIC instructions due to the constraint of the instruction length. The two corresponding real instructions are:

BTFSS *SIGINA* (or *SIGINB*), *signal*
 GOTO *address*



- [14:10] Opcode = 'BTFSS'
- [9:7] specifies the bit position (i.e. the *signal*)
- [6:0] specifies the source register – *SIGINA* or *SIGINB*



- [14:11] Opcode = 'GOTO'
- [10:0] specifies the goto *address*

Indeed BTFSS and GOTO are the original instructions provided in PIC processor. BTFSS performs a bit test which skips the instruction immediately followed if that bit is set by inserting an NOP (no operation) instruction. In this case, testing a particular bit in the signal input register (SIGINA or SIGINB) means checking the presence of a specific signal. In order to truly model Esterel's PRESENT, a

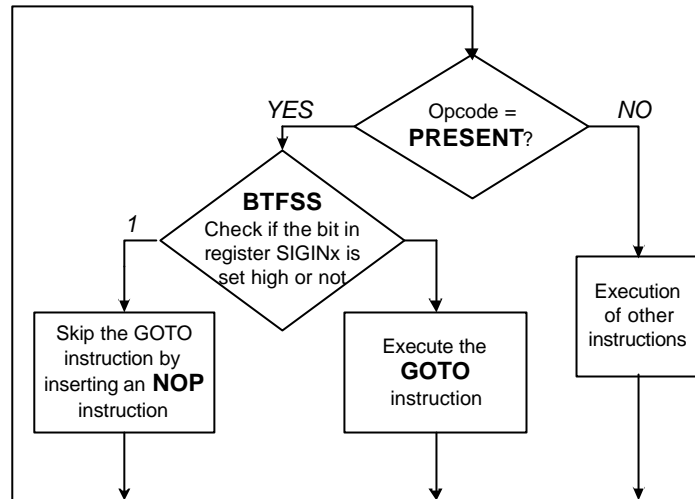


Figure 7 PRESENT Flow Diagram

GOTO instruction is placed straightly after BTFSS so that a jump will be carried out when signal is absent. GOTO will be discarded in the occurrence of signal and an NOP instruction will be executed in that case

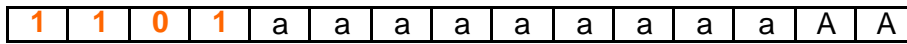
4.2.7 ABORT – Preemption

ABORT is the most important and complicated one among all the new instructions. This instruction is introduced to replace the traditional interrupt mechanism. When an active abort signal occurs, the program will immediately exit from the location its currently executing and jump to the *address* given in the instruction. This is usually called a preemptive termination of abort. The lifetime of an abortion is determined by the abort body and is terminated automatically when the program comes to executing the instruction at the specified *address* regardless of the occurrence of the abort signal. The latter is known as a non-preemptive termination of the abort.

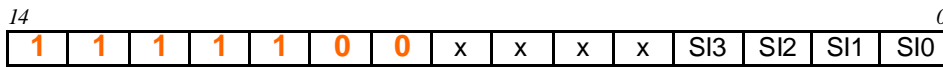
In Esterel, abortions can be strong or weak. When the abort signal occurs, a strong abort will instantly stop what it is currently doing without finishing off the current instruction. On the contrary, a weak abort will first complete the current instruction before performing any actions required in the presence of the abort signal. The ABORT instruction being implemented on RePIC is of weak type. In other words, all instructions in RePIC are atomic and cannot be interrupted half way through their decoding and execution stages.

Assembly syntax: ABORT *signal, address*
e.g. ABORT SB2, 43

Similarly, ABORT itself is a pseudo instruction which requires two actual RePIC instructions to accommodate its arguments, which are (1) the abort signal {4 bits} and (2) the target abort address {11 bits}.



[14:11] Opcode = 'LDAADDR'
 [10:0] specifies the abort continuation *address*



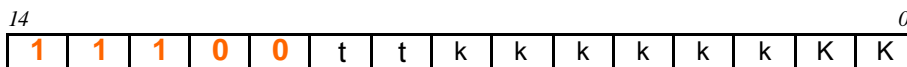
[14:8] Opcode = 'ABORT'
 [3:0] specifies the abort *signal* (Refer to Table 4.2 on page 13)

The above two instructions are used to activate an ABORT (Section 5.3.5.1). The system will run normally while checking for the abort signal aside. Unlike standard interrupts where users have to disable them explicitly using software, RePIC always holds control over the operation and termination of ABORTs once they have been activated.

4.2.8 SETINTMR – Internal Timers

SETINTMR instruction sets up an internal timer for generating input signals internally as mentioned in Section 4.2.4. A total of four 8-bit internal timers are implemented, each corresponds to one of the four input signals (SIB4: SIB7). These timers will be decremented by one every instruction cycle. As zero is reached (or timer overflows), the corresponding input signal will be set high for one *tick* long (Section 5.3.1.1). A 5-bit opcode has been assigned to this instruction. Two bits out of the remaining ten bits are used for selecting the timer (*tmr_id*), leaving the last eight bits for specifying the timeout length. The maximum length of a timeout in terms of instruction cycles is 255.

Assembly syntax: SETINTMR *tmr_id*, *tmr_val*
 e.g. SETINTMR INTMR3, 160



[14:10] Opcode = 'SETINTMR'
 [9:8] *tmr_id* ("00"= INTMR0; "01"=INTMR1; "10"= INTMR2; "11"=INTMR3)
 [7:0] specifies the length of timeout in unit of instruction cycles

4.3 Summary – New Reactive Instructions

Esterel Feature	Assembly Syntax	RePIC Instruction	Length 15-bit word	Description
Signal Emission	EMIT <i>signal(s)</i>	EMIT <i>signal(s)</i>	1	<i>Signal(s)</i> is/are set high for one tick.
Signal Sustenance	SUSTAIN <i>signal(s)</i>	SUSTAIN <i>signal(s)</i>	1	<i>Signal(s)</i> is/are set high forever.
Delay	TAWAT <i>delay</i>	TAWAT <i>delay</i>	1	Wait until <i>delay</i> (number of instruction cycles) elapses.
Signal Polling	SAWAIT <i>signal</i>	SAWAIT <i>signal</i>	1	Wait until <i>signal</i> occurs in the environment
Conditional Signal Polling	CAWAIT <i>signal1, signal2, address</i>	LDCADDR <i>address</i> CAWAIT <i>signal1, signal2</i>	2	Wait until either <i>signal1</i> or <i>signal2</i> occurs. If <i>signal1</i> occurs, execute instruction at the address immediately followed, or else at the specified <i>address</i> .
Signal Presence	PRESENT <i>signal, address</i>	BTFSS <i>register, bit</i> GOTO <i>address</i>	2	Instruction at the address immediately followed will be executed if <i>signal</i> is present, or else at the specified <i>address</i>
Preemption	ABORT <i>signal, address</i>	LDAADDR <i>address</i> ABORT <i>signal</i>	2	Program finishes its current instruction and jumps to <i>address</i> in the occurrence of <i>signal</i>

4.4 ATM Example in RePIC

An implementation of the ATM example presented previously is shown in RePIC assembly below.

```

1  loop:
2      EMIT insertCard
3      SAWAIT cardInserted
4          %Load Abort Address and initiate abort
5          LDAADDR L0
6          ABORT invalidCard
7          %Load Abort Address and initiate abort
8          LDAADDR L0
9          ABORT incorrectPin
10         EMIT enterPin
11         SAWAIT pinEntered
12         EMIT selectOption
13         LDCADDR L1
14         CAWAIT withdraw, checkBalance
15         SAWAIT sumEntered
16         EMIT processTransaction
17         SAWAIT transactionOK
18         EMIT releaseSum, printReceipt
19         GOTO L0
20  L1:
21         EMIT processTransaction
22         SAWAIT transactionOK
23  L0:
24         EMIT printReceipt
25         EMIT ejectCard
26         GOTO loop

```

It can be seen that the assembly code for RePIC resembles the original Esterel code. We can see that two instructions are needed to implement the abort mechanism (line 5,6 & 8,9) *LDAADDR [addr]* and *ABORT [signal]*. The former is used to specify the continuation address of the abort (The address of the next instruction to execute if the body is aborted) while the latter specifies the signal which the abort is sensitive to. As can be seen in the example, no context switching occurs when an abort is taken and there is no need to write a separate interrupt service routine. Such a mechanism allows Esterel abort statements to be easily converted to the equivalent RePIC machine code. Also worth noting is the ability to emit multiple signals in the same instruction as can be seen in line 18. Looking at this example we can clearly see that the resulting assembly code not only resembles the Esterel source code but also has a significantly smaller number of instructions compared with its PIC counterpart discussed in section 3.3. It is important to note that both these applications were hand coded to map the given Esterel model to PIC and RePIC. Each application was carefully written so as to optimize it based on the features of the target processor.

5 Prototype Design

The datapath of PIC is modified to accommodate the new set of reactive instructions natively as discussed in the previous section. This section gives a detailed description of the major modifications that have been done to the original datapath and also the way each new instruction is handled in RePIC.

5.1 Additional I/O Ports

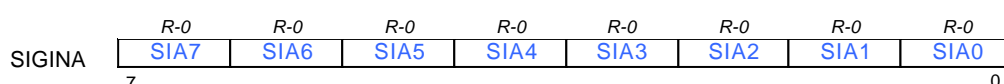
As a control-driven system, REPIC must be able to handle external signals promptly and efficiently. In the original PIC system, two 8-bit bi-directional I/O ports are provided. They are called *porta* and *portb*. However, an analysis of forty-four Esterel benchmarks shows that most of them contain more than sixteen input and output signals. Therefore, four new mono-directional I/O ports are introduced. They are called *signal_ina*, *signal_inb*, *signal_outa* and *signal_outb* respectively. So altogether, REPIC can handle up to twelve external single-bit input and twelve external single-bit output signals. The remaining ports, *porta* and *portb* can be used to output an 8-bit valued signal.

5.2 Register mapping

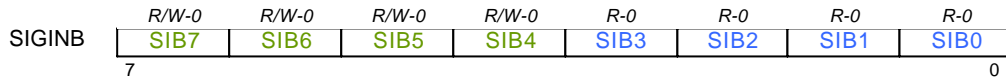
New registers are introduced to ease the execution of the new reactive instructions. The descriptions of each of these new registers are presented in the following sub-sections.

5.2.1 SIGINx and SIGOUTx Registers

SIGINx and SIGOUTx registers are used as a flip-flop to store the input or output signals which are coming from or going out to the external environment and to synchronise all signals with the system clock.

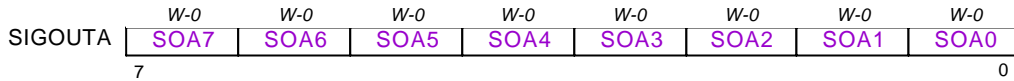


[7:0] Each bit corresponds to an input pin on *signal_ina* port

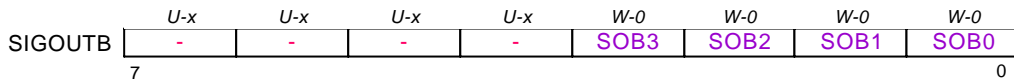


[7:4] Each bit corresponds to an internal timer-generated input signal

[3:0] Each bit corresponds to an input pin on *signal_inb* port



[7:0] Each bit corresponds to an output pin on *signal_outa* port



[3:0] Each bit corresponds to an output pin on *signal_outb* port

The ABORT, PRESENT, CAWAIT and SAWAIT instructions all require access to the SIGINx registers. or EMIT and SUSTAIN instructions, they both employ the SIGOUTx registers for outputting signals to the external world via the output ports.

Key:
R = Readable bit
W = Writable bit
U = Unused bit, read as '0'
-n = Power-on reset value

5.2.2 INTMRx Registers

As mentioned previously, there are a total of four internal timers utilised for the generation of the internal signals SIB4:SIB7. The INTMRx registers store the current timer value which will be decremented automatically in each instruction cycle if its corresponding timer enable bit in the TMRCOND register is set. When the value of the INTMRx registers reaches zero, the corresponding internal signal will be set. These registers are only used by the SETINTMR instruction.

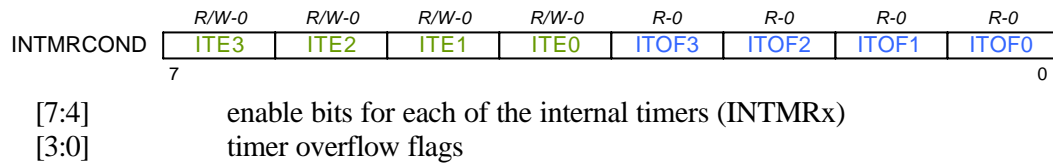


[7:0] specifies the timeout value in unit of instruction cycles

5.2.3 INTMRCOND Register

The INTMRCOND register stores the status of the internal timers (INTMR0, INTMR1, INTMR2 and INTMR3). The most significant four bits indicate whether the corresponding timer has been enabled or not. And the least significant four bits are the flags used to signify that the corresponding timer has overflowed or reached zero. Timers will be disabled upon the time they reach zero. When the system sees that an overflow flag (ITOFx) has been set high, it will generate the appropriate internal

signal and clear the overflow flag accordingly. This register is only accessed by the SETINTMR instruction.



5.3 Implementation of New Reactive Instructions

5.3.1 EMIT & SUSTAIN

With the new output ports (signal_outa and signal_outb) and the signal output registers (SIGOUTx), signal emission and sustenance can be implemented simply by connecting each bit of the registers to its associated output pin. Each connection corresponds to a single-bit signal. Whenever a new value is loaded from the instruction register (INST) to these registers, a new set of signals will be emitted to the environment (synchronous to system clock). However, since signal sustenance causes a signal to be maintained high forever. Two new internal registers (SUSTAINx) are introduced to hold all the sustained signals. In this case, SIGOUTx have to be OR-ed with SUSTAINx each time to ensure all sustained signals will never be cleared (Figure 8).

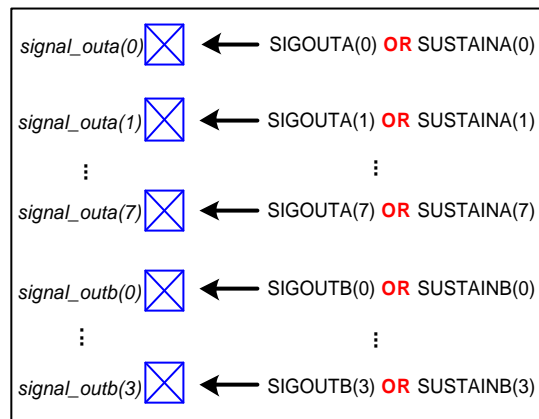


Figure 8 EMIT and SUSTAIN

5.3.2 TAWAIT

Native TAWAIT instruction is implemented in a way very alike to the operation of a traditional counter. The execution of next instruction is stalled until the value of the counter reaches zero. This instruction requires an extra 8-bit register (DELAY_CNT) to store the current value of the counter and a 1-bit delay operation flag (DELAY_OP) to indicate whether the TAWAIT instruction has been completed yet or not. As long as the system checks that DELAY_OP is '1', the counter will be decremented by one in every instruction cycle (one instruction cycle corresponds to the shortest possible tick in REPIC), the program counter will retain its value and the next instruction will not be executed until DELAY_OP is cleared.

5.3.3 SAWAIT & CAWAIT

SAWAIT and CAWAIT instructions are introduced for signal polling and conditional signal polling respectively. Both these instructions stall the execution of next instruction until the specified signal occurs, except that the latter one allows polling of two different signals contemporarily. As mentioned previously, four bits are used to represent each of the sixteen input signals available on REPIC. Therefore, when a SAWAIT instruction is fetched from the program memory and decoded, the lowest four bits of the instruction register (INSTR[3..0]) which specify the signal to be polled, will be passed into a 4-bit decoder, producing a 9-bit mask which is stored in an internal register called SIGPOLLA for checking the presence of the particular signal (Figure 9).

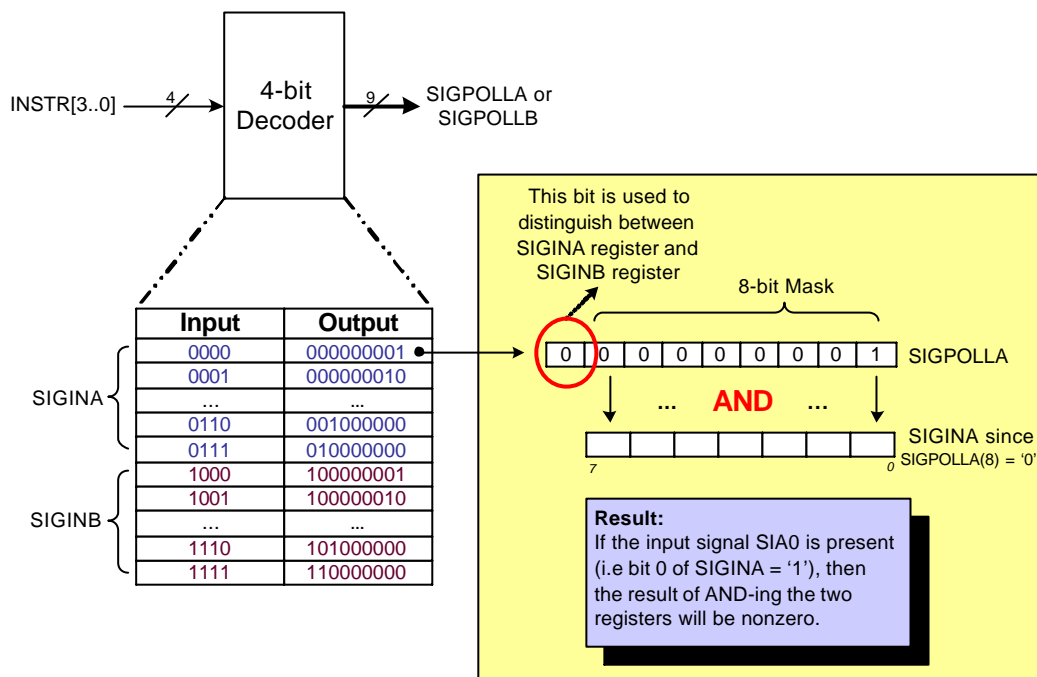


Figure 9 The Operation of SAWAIT

The operation of CAWAIT is basically the same as that of SAWAIT. The only difference is that CAWAIT requires another 4-bit decoder for decoding the second signal. So another hidden register, SIGPOLLB is added for storing the bit mask for checking the second signal. Both SAWAIT and CAWAIT have a 1-bit operation flag, called SIGPOLL_OP and CSIGPOLL_OP respectively. They are used to stall the pipeline while the system is waiting for the specified signal to occur.

5.3.4 PRESENT

The implementation of PRESENT does not require any modifications made to the original PIC data path because two PIC instructions are taken to perform this function. Extending PIC's register file to include the two new input registers, SIGINA and SIGINB, simply allows native support of PRESENT in REPIC.

5.3.5 ABORT

A native ABORT instruction is introduced in REPIC for supporting preemption. In fact, the basic design of REPIC's abortion is originally taken from REFLIX [13]. Figure 10 depicts the extensions of PIC's data path required for handling ABORT. Up to four priorities of nesting of ABORTs are allowed in REPIC. The outermost ABORT will always have the highest priority and the innermost ABORT will always have the lowest priority. Weak abortion is implemented

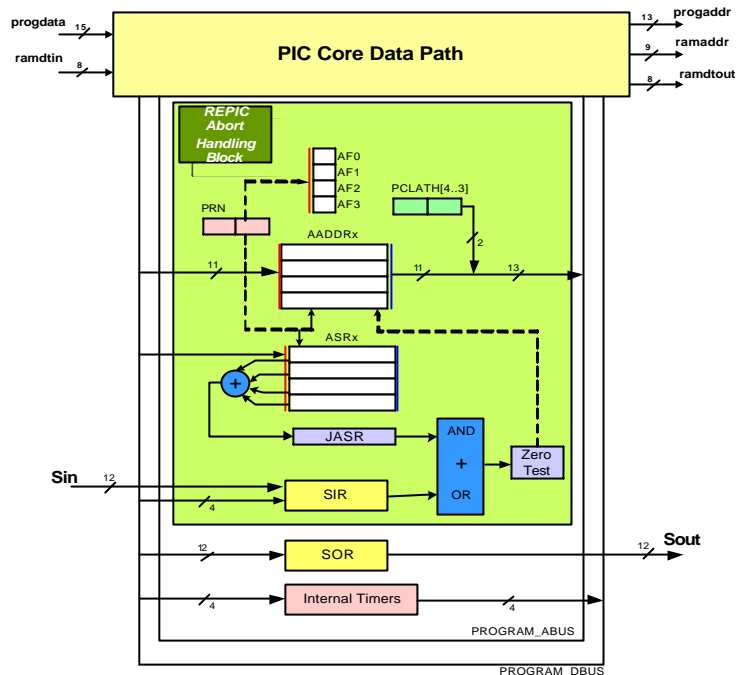


Figure 10 RePIC's Data Path of ABORT Handling

to avoid data inconsistency which may be resulted from interrupting instructions half-way through their execution stage. Currently, ABORT in REPIC can work with up to twelve external and four internal timer generated signals.

5.3.5.1 ABORT Activation

For handling four levels of nesting of ABORTs, a 2-bit priority node (PRN) which controls both the activation and termination of ABORTs is added. Section 4.27 has clearly shown the two REPIC instructions required for activating an ABORT. They are: -

1. LDAADDR address – which loads the abort continuation address into one of the internal abort address registers (AADDRx, where x = 0, 1, 2 or 3) depending on the priority node (PRN).

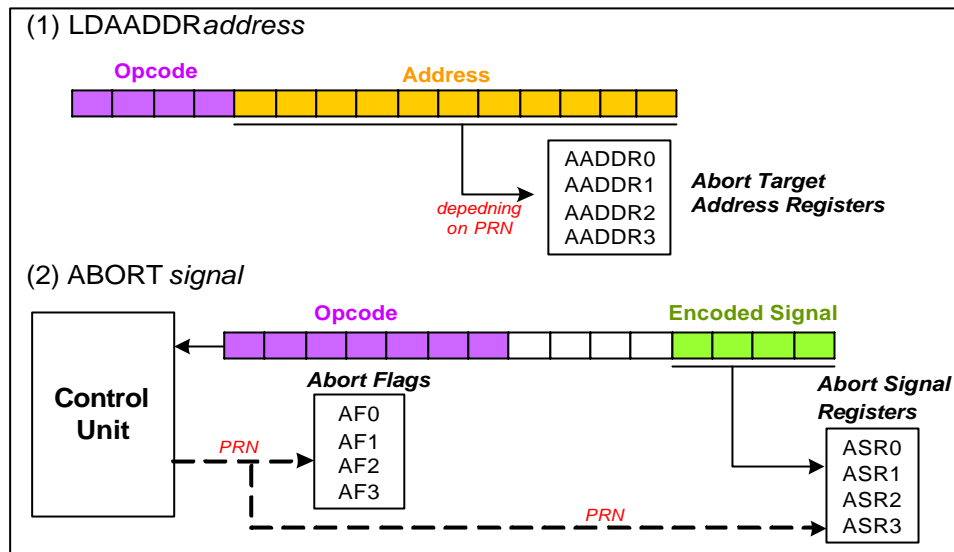


Figure 11 Activation of ABORT

2. A

ABORT signal – which activates an ABORT by setting the appropriate abort flag (AF_x) high (where x = 0, 1, 2 or 3) and also decodes signal into a bit mask which is held in an internal abort signal register (ASR_x, where x = 0, 1, 2 or 3) for checking the presence of the abort signal (Figure 11). The priority node is essential for storing all these data into registers with the correct priority. The priority of each register is denoted by the number attached at the end of its name (i.e. x = 0, 1, 2 or 3). A ‘0’ always has the highest priority, followed by the ‘1’ and then ‘2’ and ‘3’ always denotes the lowest priority.

An ABORT will become active immediately after these two instructions are being executed. REPIC system will then keep monitoring the changes of the designated abort signal until ABORT terminates.

5.3.5.2 ABORT Handling

The ABORT handling block presented in Figure 12 is designed to handle the operation of ABORTs after they have been activated.

The handling block consists of a number of registers for internal operation. They are listed as follows:

- AF_x – Abort Activation Flags. ‘1’ = active and ‘0’ = non-active
- PRN – Priority Node
- AADDR_x – Abort Continuation Addresses, from which the program will continue if preemption happens
- ASR_x – Abort Signal Registers, which store the designated abort signal
- JASR – Joint Abort Signal Register which is the result of bitwise OR-ing the four ASRs

- SIR – Signal Input Register is a combined name given for SIGINA and SIGINB registers

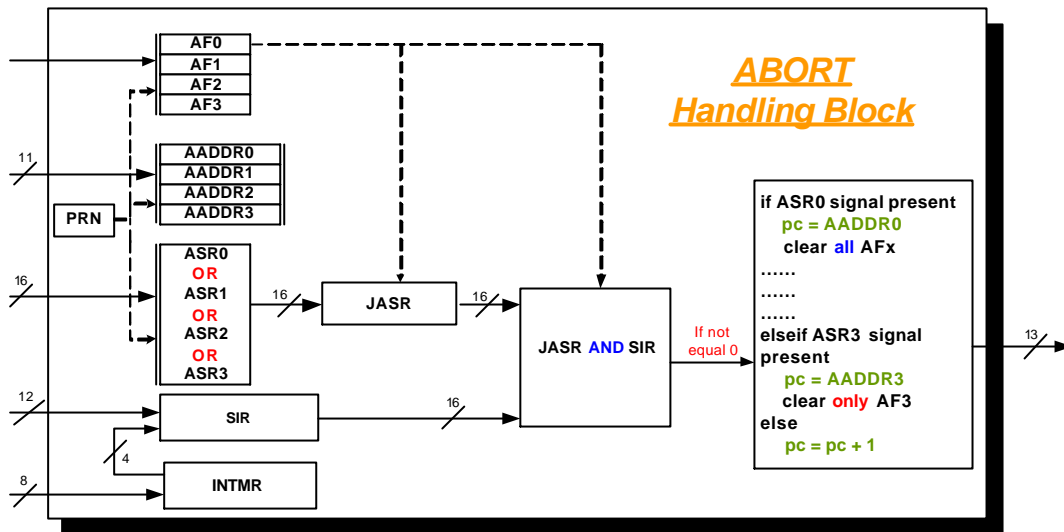


Figure 12 RePIC's ABORT Handling Block

REPIC can support up to four levels of nesting of ABORTs. The priority of them is controlled in hardware by use of a 2-bit priority node, PRN. Initially, PRN has a value of 00b which indicates that there are no active ABORTs. When an ABORT instruction is fetched and decoded, PRN causes the abort continuation address and the abort signal to be stored into the registers with the highest priority (i.e. AADDR0 and ASR0 respectively). Also, AF0 will be set high indicating that an ABORT with highest priority has been activated. Then, PRN will be automatically incremented by one (i.e. 01b). So, if another ABORT instruction comes in, AF1, AADDR1 and ASR1 registers will be updated accordingly. And PRN will be incremented again. It must be mentioned here that the compiler is responsible to ensure that the nesting of ABORTs does not exceed the maximum of four levels.

REPIC only needs to check for changes of AF0 to determine whether there are any active ABORTs pending within the system. This is because the activation of ABORTs always starts from the highest priority. If AF0 is equal to one, this indicates that there is at least one active pending ABORT. Then the system will perform the following two operations alongside with the normal execution of other instructions until the active ABORTs terminate:

- bitwise OR-ing the four ASRs and put the result into JASR
- $JASR_i = ASR0_i + ASR1_i + ASR2_i + ASR3_i$ (for $i = 0, 1, 2, \dots, 15$)
- bitwise AND-ing JASR with SIR (i.e. SIGINA + SIGINB)
- $result_i = JASR_i \times SIR_i$ (for $i=0, 1, 2, \dots, 15$)
-

These operations are done to determine the activation of any designated abort signals. A non-zero result from the second operation would imply that preemption has

happened, REPIC will further find out what priority the occurring signal corresponds to and update the program counter with the appropriate address retrieved from one of the AADDRx registers.

For prioritising nesting of ABORTs, IF-ELSIF statements in VHDL are used to ensure that the earlier activated (i.e. the outermost) ABORT always takes precedence of execution over the later ones in the case when more than one designated abort signals occur in the environment at the same time.

5.3.5.3 Termination of ABORT

Termination of ABORT can be preemptive or non-preemptive. The latter occurs if the entire abort body has been executed prior to the abort signal occurring in the environment. In this case, the corresponding active ABORT will be deactivated or terminated automatically. The former results when the designated abort signal occurs during the abort body execution. This causes the program to abort execution of the body and continue at the continuation address specified in the AADDRx register.

6 Preserving the Synchrony Hypothesis of Esterel

Esterel's synchronous model implies that reactions to input signals are instantaneous and occur at discrete logical instants called *ticks*. This means that a sequence of actions may need to be performed within the duration of a single *tick*.

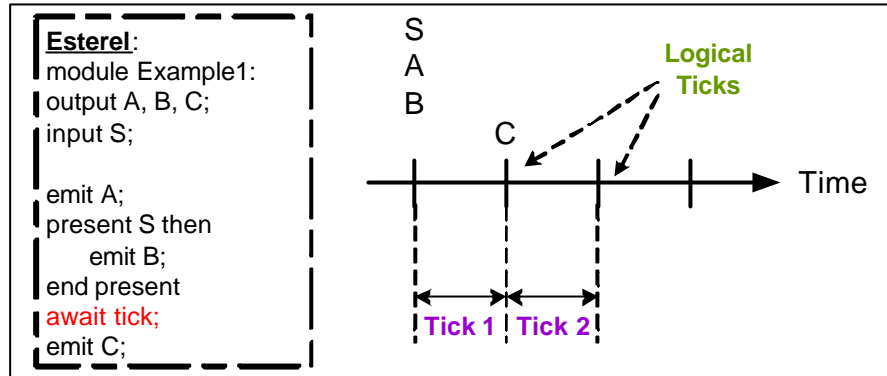


Figure 14 The Esterel Tick

In order to model the logical tick of Esterel in RePIC, a tick with variable length in terms of absolute time is implemented. The duration of one tick is determined by the number of cycles required to execute all the instructions that are placed between any two *await* instructions (TAWAIT, SAWAIT and CAWAIT). Whenever one such instruction appears, the current tick is said to have elapsed and a new tick starts. In

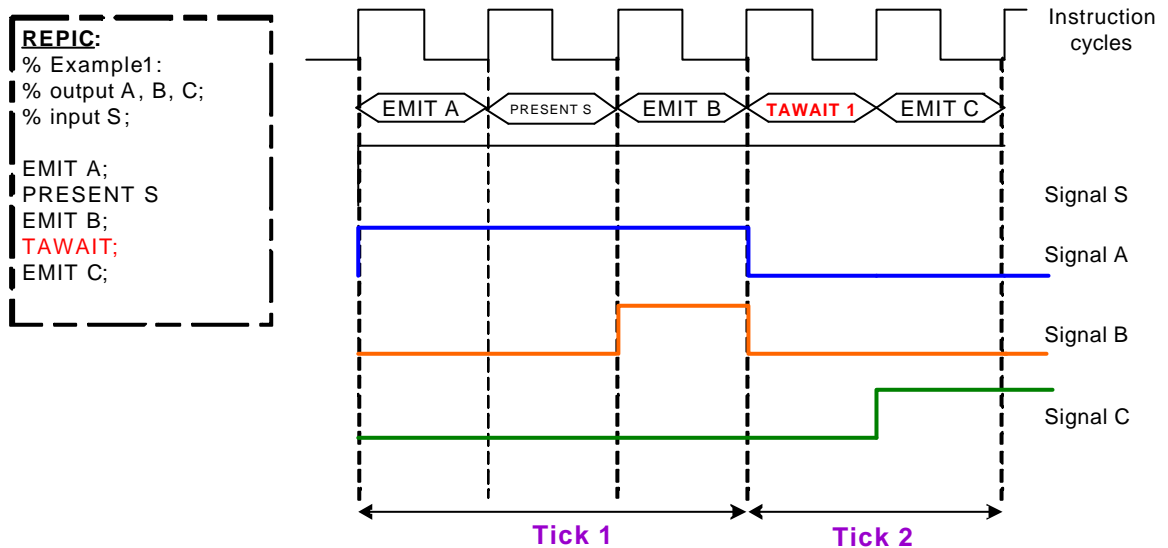


Figure 15 Implementation of Esterel Tick in RePIC

Esterel, all output signals are being emitted instantaneously and last for one logical instant as shown in Figure 14. Similarly, signals in RePIC will be set high after an EMIT or SUSTAIN instruction is executed and will be cleared when the next TAWAIT, SAWAIT or CAWAIT instruction is decoded. In this sense, the instantaneity of signal broadcasting within a logical tick is preserved in RePIC through the introduction of a variable tick depending on how frequent the await instructions occur in the program. An example is given in Figure 15 (also refer to Figure 14).

Looking at the about figure we can see how signal A and signal B are both emitted at the end of the first tick although they are decoded and executed at different instances in real time. The second tick is consumed by the TAWAIT instruction and we can see that signal C is emitted in the third tick.

7 Various Tools and Languages Used

A number of software and hardware tools have been adopted during the development and implementation of RePIC. These will be presented here. Figure 16 shows an overview of the tools used.

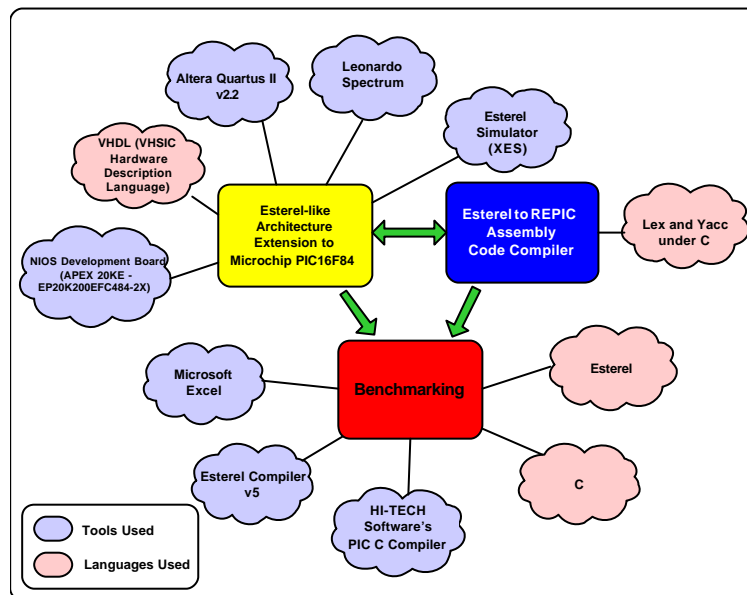


Figure 16 Various Tools and Languages Used

7.1 VHSIC Hardware Description Language

VHSIC Hardware Description Language (VHDL) is a language used to express simple to complex digital systems aiming for documentation, simulation, verification and synthesis. Unlike most programming languages, VHDL supports descriptions of concurrent events where concurrency is a key behaviour of hardware circuits. This language is used as the main design tool for modifying the existing target Microchip processor.

7.2 Esterel

Esterel is a hybrid synchronous programming language used for modelling and simulating reactive systems. Many features [11] in Esterel (Figure 16) behave very similarly as those in hardware circuits (e.g. use of clock ticks to model its time for synchronous execution and use of signals to communicate between concurrent modules) which can easily be translated onto hardware using VHDL. The current processor design is largely based on the semantics and attributes of this high-level programming language.

Basic Features of Esterel

ESTEREL =	
Syntactic construction to deal with signals	+
Synchronous execution	+
Instantaneous broadcast	+
Concurrency	+
Determinism	

Figure 17 Basic Features of Esterel Language

7.3 Lex and Yacc

Lex, also known as Lexical Analyzer Generator [17] is designed for lexical processing of input streams. The program is essentially a table of regular expressions and the corresponding program fragments. It reads an input stream, partitions the input characters into strings (often called tokens) which match the given expressions and executes the corresponding program fragment as each such string is recognized. While Yacc [17] is a parser generator which accepts a large class of context free grammars, each of which is specified by a syntactic grouping with constructing rules attached to it. However, Yacc requires a lower-level analyser to recognise the input tokens first. Therefore, the combination of Lex and Yacc is often appropriate. In this case, they are used to develop a translation tool where its job is to convert Esterel programs straight into RePIC assembly and machine codes.

8 Synthesis Results

The prototype design of RePIC was implemented using FPLD (Field Programmable Logic Device) technology. The open-source VHDL code available for PIC [18] was modified based on the architectural extensions discussed in the previous sections. Simulation of the functionality of RePIC was conducted using Altera Quartus II (version 2.2) software. The entire design was then synthesized and mapped onto the Excalibur NIOS development board.

Microprocessor System (i.e. Core + Data RAM + Program ROM)		PIC (1-processor Architecture)	RePIC (1-processor Architecture)	RePIC (2-processor Architecture)
Logic Elements	Number Used	1082	2068	4761
	Total Available	8320	8320	8320
	Percentage Used	13%	24%	57%
Pins	Number Used	48	72	75
	Total Available	376	376	376

	Percentage Used	12%	19%	19%
Memory Bits	Number Used	58368	62464	63488
	Total Available	106496	106496	106496
	Percentage Used	54%	58%	59%
System Clock Frequency		45.83MHz	40.27MHz	35.38MHz

Table 2 Resource use of the PIC and RePIC systems

The target FPGA used for this synthesis is the EP20K200EFC484-2 which is a member of the high-density APEX20KE family sited on the Excalibur Nios Development Board. The resource utilisations of each of the systems are presented in Table 2.

The RePIC system (with no concurrency support) consumes logic resources nearly double of its unmodified PIC counterpart. The implementation of real concurrency (i.e. the dual processor architecture) further doubles the utilization of logic elements as an additional core together with some synchronization and signal broadcasting mechanisms have been added for correct behaviour of the entire system.

Both RePIC systems utilize more pins than the original PIC system due to the four extra I/O ports on RePIC. Moreover, due to the extra bit on RePIC's instruction word, the program ROM requires more memory bits in order to store the same number of program instructions as the original PIC system.

In terms of instruction execution rate, the single-processor RePIC architecture has a lower frequency of 10MHz compared to 11.46MHz of PIC. For the dual processor architecture, it runs at approximately 9MHz, which is roughly 1.3 times slower than the original processor. In order to ensure a fair comparison among these systems, the different maximum operating frequencies have also been taken into account during the benchmarking process.

9 Benchmarking

This section demonstrates the improvements RePIC is providing over the original PIC, specifically:

- A significant reduction in code size was achieved on RePIC compared to PIC. Code size comparisons were done for both manually translated code as well as those compiled using various Esterel compilers.
- Execution time is compared for both manually translated code as well as those done using Esterel compilers. A significant speedup was achieved in both cases.
- Comparison of RePIC with other microprocessors (which are widely applied in embedded systems) show considerable improvement in code size.

9.1 ART-Bench – A New Benchmark Suite

RePIC was designed for optimised execution of reactive applications. Hence, the reactive performance of the new processors needed to be compared with existing processors. Although [13] provides a set of applications for analysing the performance of reactive applications, the set is insufficient and needs to be extended to enable a thorough analysis of the performance of a processor running reactive tasks. Another benchmark, called EstBench, [14, 19], has been developed to measure efficiency of various Esterel compilers and not target microprocessors on which these reactive programs execute. Therefore it lacked the features needed by a proper reactive benchmark, such as varying levels of pre-emption, multiple priorities and purely reactive tasks. To overcome this limitation a new benchmark suite called the Auckland Real-Time Benchmark (ART-Bench) was created. ART-Bench was formed by integrating all the applications provided in [13] with the Esterel benchmarks from EstBench. ART-Bench also include some programs from other sources such as a traffic light controller from the Polis [9] distribution. On the whole, ART-Bench includes 44 applications including 8 purely reactive applications. The benchmark suite consists of applications with

- Multiple concurrent aborts.
- Multiple nesting of aborts.
- Parallel execution with signal emissions.
- Both reactive and data dominated applications as well as purely reactive applications.
- Varying size of code ranging from toy applications to very complex applications.

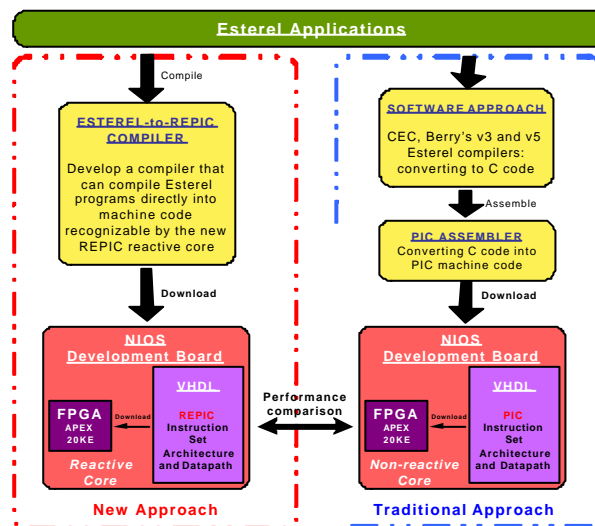


Figure 4-1 Methods of Esterel Software Compilation

9.2 The Benchmarking Process

The benchmarking process was divided into two separate parts. These were as follows

- 1) A subset of applications from ART-Bench was manually translated into the machine code of a number of microprocessors. The selected benchmarks were small purely reactive applications, since manual translation of larger benchmarks becomes humanly impossible for standard microprocessors. Both the execution time and the code size were compared.
- 2) The compactness of the code generated by direct mapping of Esterel was then compared to the code generated by traditional Esterel compilers. This was done by converting Esterel to RePIC assembly using the Esterel to RePIC compiler, and by converting Esterel to C and then to assembly by using a commercial PIC compiler and assembler. Speed up of this new method was also analysed.

9.2.1 Comparison using manual translation of Esterel

The first comparison was done by translating some of the purely reactive benchmarks from ART-Bench to the native code of RePIC, PIC [1], Motorola 68HC11 [20], Intel 8051 [21] and 16-bit NIOS [22]. Table 4-1 summarizes these results and shows a clear advantage of RePIC in terms of code size and memory footprint. Though generating code by hand significantly reduces any overhead incurred by compilers, we can still see a significant reduction in code size for RePIC.

Application	RePIC	PIC	8051	68HC11	NIOS-16
<i>ATM</i>	22	74	102	163	219
<i>Elevator</i>	23	70	45	79	116
<i>Pump Controller</i>	15	50	35	56	80
<i>Startup Benchmark</i>	24	64	48	76	94
<i>TCP Receive</i>	7	27	28	18	41
<i>TCP Transmit</i>	10	29	20	31	46
<i>Traffic Light</i>	18	71	70	114	147
<i>Average</i>	17	55	50	77	106

Table 4-1 Code size comparison in words

This can be mainly attributed to the native support for Esterel in RePIC. By supporting instructions such as abort natively we are able to remove the overhead generated by traditional microprocessors, which require the use of interrupt service routines to handle the same functionality. This is evident by the facts that benchmarks which have a higher number of aborts (ATM, Traffic Light) results in the

Application	Execution Time (us)		Total Speed Up
	RePIC	PIC	
<i>ATM</i>	2.68	8.73	3.26
<i>Elevator</i>	2.38	5.85	2.4
<i>Pump Controller</i>	1.99	5.76	2.89
<i>Startup Benchmark</i>	3.97	7.33	1.85
<i>TCP Receiver</i>	0.70	2.36	3.37
<i>TCP Transmitter</i>	1.00	2.53	2.53
<i>Traffic Light</i>	2.28	7.51	3.29
Average	2.78	6.74	2.70

Table 4-2 Execution time comparisons of PIC and RePIC

greatest reduction in code size. On an average the original PIC processor requires three times larger memory to store the same program functionality than RePIC requires.

Next was a comparison of the execution time of original PIC and RePIC, which is shown in Table 5-2. On average a speedup of 2.7 was achieved. However, applications that contained many levels of nesting of aborts (ATM, Traffic Light) show a speed up of approximately 3.2.

9.2.2 Comparison of RePIC versus traditional Esterel compilers

Next, a comparison was done between the code generated by traditional Esterel compilers and code produced for RePIC by using the RePIC compiler and assembler. As shown in Figure 4-1, traditional Esterel compilers firstly convert a given Esterel application into an intermediate C program. This C code is then assembled for the target architecture. For RePIC, the compiler and assembler are able to directly map the Esterel application onto native code.

The former method of converting to C has certain overheads. Since traditional architectures have no notion of an Esterel tick, extra software is needed to ensure that signals are emitted and polled for correct time durations. Furthermore, instruction-scheduling overhead is also introduced when converting multiple concurrent Esterel code segments into C code. The conversion of Esterel abort statement also leads to a significant increase in code size. An interrupt service routine needs to be implemented for each signal that causes preemption. Moreover, each interrupt has an overhead involved in initialization that also adds to the increase in code size. Although the handling of abort priorities is done using hardware in RePIC, PIC code needs to be generated to handle this in software that also leads to inefficient and large code.

The performance of three Esterel compilers was next compared with the direct code generation of the RePIC compiler. The selected compilers were, Esterel Studio v4.0 by Esterel technologies [8], Barry et al’s Esterel v5 compiler [2] and the automata based Esterel v3 compiler. A summary of the results are shown in Table 4-3 which shows that that the percentage reduction in code size is now in the high nineties compared to the 76% reduction achieved when manual translation of Esterel to machine code was done.

Benchmark	Esterel Studio 4.0	Esterel V5	Esterel V3	RePIC	Average Reduction in Code Size
<i>Abcd</i>	2208	2383	2267	28	98.78%
<i>Driver</i>	1540	1588	1575	68	95.66%
<i>Elevator</i>	1147	1192	474	23	97.55%

<i>PumpController</i>	811	945	346	15	97.86%
<i>Startup</i>	869	1143	305	24	96.89%
<i>TCPReceive</i>	498	499	293	7	98.37%
<i>TCPTransmit</i>	588	593	260	10	97.92%

Benchmark	Esterel Studio 4.0	Esterel V5	Esterel V3	RePIC	
<i>Abcd</i>	291.93	312.53	377.83	4.37	
<i>Driver</i>	191.36	195.99	220.95	15.29	
<i>Elevator</i>	147.01	152.69	59.8	2.38	
<i>PumpController</i>	105.19	122.22	45.83	1.99	
<i>Startup</i>	114.45	149.81	38.15	3.97	
<i>TCPReceive</i>	63.55	63.99	37.62	0.7	
<i>TCPTransmit</i>	74.99	75.78	30.81	1	
<i>TrafficLight</i>	132.34	154.87	51.33	2.28	
<i>ATDS-100</i>	2361.63	2576.57	875.09	68.12	
<i>TrafficLight</i>	1008	1188	373	18	97.90%
<i>ATDS-100</i>	17827	19735	5816	282	98.05%

Table 4-3 Code size comparison between various compilers

Next, the execution time for the above benchmarks were compared. Due to the reactive nature of these applications execution time cannot be calculated as they could for a sequential program, since execution can take any one of many different execution paths (i.e. paths in the underlying

Table 4-4 Execution time in microseconds for various compilers

finite state machine). Thus calculations were based on the assumption that each benchmark executes each state of its inherent finite state machine once and only once. This is a fair assumption since we are only interested in comparison of the speed up of the benchmarks rather than their absolute execution time. Table 4-4 gives a summary of the timing data that was obtained.

It is evident that regardless of the Esterel compiler used the execution time on RePIC by directly converting the Esterel model to machine code is significantly smaller than that for PIC. The average speed up was approximately 30, although for some benchmarks RePIC was almost 100 times as fast as PIC.

These results show that by providing the appropriate framework for directly compiling Esterel models into machine code we are able to not only reduce the code size considerably but also obtain a significant speed up compared to traditional methods.

PIC	RePIC
Average code size for manually translated benchmarks was 55 words	Average code size is only 17 words. A reduction of 69% compared to PIC and an average reduction of 76% when compared with other processors was achieved.
Execution time of the first set of benchmarks was on average 6.74us.	Average execution time for the same set of benchmarks was 2.78us, for a speed up of 2.7
Average code size when using standard compilers was 2498 words.	Average code size for the same benchmarks when directly compiled onto RePIC was only 53 for a reduction of 97.8%

	for a reduction of 97.8%
Average execution time of code compiled using standard Esterel compilers was 334us.	RePIC executes the same applications in 11us on average. This is a speed up of 30.36.

Table 4-5 Summary of results

In summary, in comparison to PIC and other microcontrollers we have been able to achieve an average speed up of 2.7 and an average a reduction in code size of 76% for applications that are manually translated to both PIC and RePIC. Simple reactive benchmarks were used in this analysis. Much higher improvements were achieved when comparisons were done using traditional compilation methods. When compared to conventional Esterel compilers, the RePIC compiler produced code that was on an average 97% smaller. It is able to achieve this by directly converting the Esterel model to machine code without first translating it to C.

10 Conclusions and future work

This paper presents a new approach for direct Esterel execution on a microprocessor while preserving the semantics of the language. Existing compilers for a concurrent and reactive language like Esterel, first compile it to an intermediate high-level language. This code is generated so as to execute on a conventional sequential processor. Such an indirect approach to the execution of a reactive program on a sequential processor introduces several redundancies and the resulting code is both bulky and inefficient. As a result, Esterel is rarely used in the design of small hand-held embedded systems (though it is widely applied to large safety-critical systems). This paper remedies these shortcomings by proposing a new reactive microcontroller (RePIC) that has direct support for features of Esterel and enables the direct conversion of Esterel into the machine code of RePIC. In order to study the efficacy of the approach, a new benchmark suite called ART-Bench has also been proposed for comparing the performance of reactive applications running on different microprocessors. Benchmarking results using ART-Bench reveal many orders of magnitude improvement in performance and a radical reduction in code size using RePIC.

RePIC has some limitations and research is ongoing to develop a multiprocessor architecture around RePIC for real concurrency support (currently we have a dual-processor proof of concept), the development of small hand held embedded systems such as mobile phones and games using RePIC like processors and the comparison of the Esterel compiler developed by Edwards [11] (which could not be done because of lack of availability of this compiler).

11 References

- [1] Microchip.Ltd, (1998 - Last Updated) "PIC16F8X Datasheet" [Online]. Available http://www.microchip.com/download/lit/pline/pic_micro/families/16f8x/30430c.pdf [Last Accessed 10/04/2003]
- [2] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, vol. 19, pp. 87-152, 1992.
- [3] S. A. Edwards, "Compiling Esterel into sequential code," presented at Hardware/Software Codesign, 1999. (CODES '99) Proceedings of the Seventh International Workshop on, 1999.
- [4] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, "Synthesis of software programs for embedded control applications," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 18, pp. 834-849, 1999.
- [5] F. Balarin and M. Chiodo, "Software synthesis for complex reactive embedded systems," presented at Computer Design, 1999. (ICCD '99) International Conference on, 1999.
- [6] R. Leupers, "Code generation for embedded processors," presented at System Synthesis, 2000. Proceedings. The 13th International Symposium on, 2000.
- [7] S. A. Edwards, (1/9/2003 - Last Updated) "CEC: The Columbia Esterel Compiler" [Online]. Available <http://www1.cs.columbia.edu/~sedwards/cec/> [Last Accessed 10/10/2003]
- [8] Esterel-Technologies, (2003 - Last Updated) "Esterel Technologies Website" [Online]. Available <http://www.esterel-technologies.com> [Last Accessed 11/11/2003]
- [9] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. New York: Kluwer Academic Publishers, 1997.
- [10] M. F. Jacome and G. De Veciana, "Design challenges for new application specific processors," *Design & Test of Computers, IEEE*, vol. 17, pp. 40-50, 2000.
- [11] M. Breternitz, Jr. and J. P. Shen, "Architecture synthesis of high-performance application-specific processors," presented at Design Automation Conference, 1990. Proceedings. 27th ACM/IEEE, 1990.
- [12] A. Wolfe and J. P. Shen, "Flexible Processors: A Promising Application-specific Processor Design Approach," presented at Microprogramming and Microarchitecture, 1988. Proceeding of the 21st Annual Workshop on, 1988.
- [13] Z. Salcic, P. S. Roop, Biglari-Abhari, and A. M. and Bigdeli, "REFLIX: A Processor Core for Reactive Embedded Applications," presented at 12th International Conference on Filed Programmable Logic and Applications, Montpellier, 2002.
- [14] S. A. Edwards, "An Esterel compiler for large control-dominated systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, pp. 169-183, 2002.
- [15] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich, "Synthesis of Software Programs for Embedded Control Applications," presented at 32nd Design Automation Conference, San Francisco, CA, 1995.

- [16] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and improvement of Boolean comparison method based on binary decision diagrams," presented at Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on, 1988.
- [17] J. R. Levine, T. Mason, and D. Brown, *Lex & Yacc*. Sebastopol, California: O'Reilly & Associates, 1995.
- [18] S. Morioka, (2003 - Last Updated) "CQPIC: PIC Micro Computer Free Soft IP" [Online]. Available <http://www02.so-net.ne.jp/~morioka/cqplic.htm> [Last Accessed 15/04/2003]
- [19] S. A. Edwards, (2003 - Last Updated) "The Estbench Esterel Benchmark Suite" [Online]. Available <http://www1.cs.columbia.edu/~sedwards/software.html> [Last Accessed 17/11/2003]
- [20] P. Spasov, *Microcontroller Technology: The 68HC11*: Prentice Hall, 1999.
- [21] M. Schutti, M. Pfaff, and R. Hagelauer, "VHDL design of embedded processor cores: the industry-standard microcontroller 8051 and 68HC11," presented at ASIC Conference 1998. Proceedings. Eleventh Annual IEEE International, 1998.
- [22] (2003 - Last Updated) "NIOS Embedded Processor: 16-bit Programmer's Reference Manual" [Online]. Available <http://www.altera.com> [Last Accessed 17/11/2003]