

A Multi-Threaded Reactive Processor

Reinhard v. Hanxleden

Joint work with Xin Li and Marian Boldt*

Real-Time Systems and Embedded Systems Group
Department of Computer Science
Christian-Albrechts-Universität zu Kiel, Germany
www.informatik.uni-kiel.de/rtsys

*Dept. of Electrical and Computer Eng., University of Minnesota

UC Berkeley, CHES Seminar, Fall 2007
18 September 2007



Reactive vs. Non-Reactive Systems

Transformational systems *numerical computation programs,
compilers . . .*

Interactive systems *operating systems, databases . . .*

Reactive systems *process controllers, signal processors . . .*

Why “Reactive Processing” ?

Control flow on traditional (non-embedded) computing systems:

- ▶ Jumps, conditional branches, loops
- ▶ Procedure/method calls

Control flow on embedded, reactive systems: all of the above, plus

- ▶ Concurrency
- ▶ Preemption

The problem: mismatch between traditional processing architectures and reactive control flow patterns

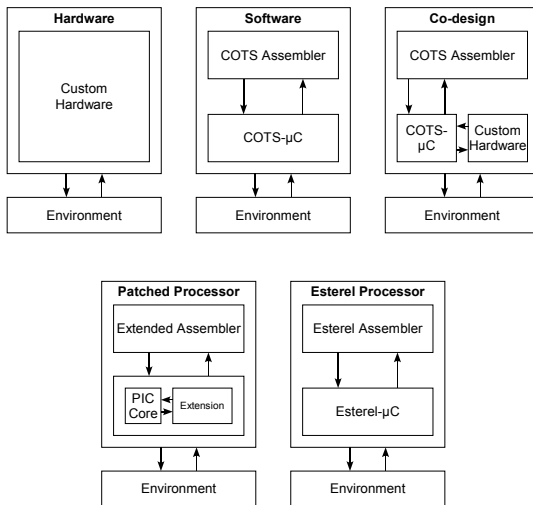
- ▶ Processing overhead, e. g. due to OS involvement or need to save thread states at application level
- ▶ Timing unpredictability

Reactive Processing Part I: The Language

Have chosen **Esterel**:

- ▶ Created in the early 1980's
- ▶ For programming control-dominated reactive systems
- ▶ Used as intermediate language for Statecharts (Safe State Machines)
- ▶ Textual imperative language with reactive control flow constructs
 - ▶ Concurrency
 - ▶ Weak/strong abortion
 - ▶ Exceptions
 - ▶ Suspension
- ▶ A synchronous language
- ▶ Deterministic behavior, clean semantics
- ▶ Currently undergoing IEEE standardization

Reactive Processing Part II: The Execution Platform



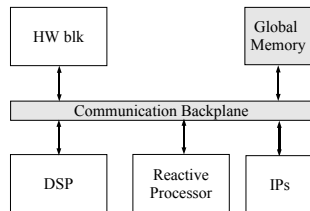
Why bother?

Reactive processing yields

- ▶ Low power requirements
- ▶ Deterministic control flow
- ▶ Predictable timing
- ▶ Short design cycle

Can use reactive processor

- ▶ in stand alone, small reactive applications
- ▶ as building block in SoC designs



Overview

Introduction

The Kiel Esterel Processor

The Esterel Language
Instruction Set Architecture
Processor Architecture
Compiler

Experimental Results

Summary and Outlook

The Esterel Language

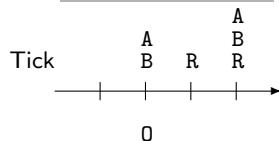
Logical Ticks

- ▶ Execution is divided into *ticks*
- ▶ **Synchrony hypothesis:**
Outputs generated from given inputs occur at the same tick

Signals

- ▶ *Present* or *absent* throughout a tick
- ▶ Used to communicate internally and with the environment

```
module ABRO:  
input A, B, R;  
output O;  
loop  
  abort  
    [ await A  
      ||  
        await B ];  
  emit O  
  halt;  
  when R  
end loop;  
end module
```



Candidates for the Instruction Set

Esterel kernel statements

- ▶ `||`
- ▶ `suspend ... when S`
- ▶ `trap T in ... exit T ... end trap`
- ▶ `pause`
- ▶ `signal S in ... end`
- ▶ `emit S`
- ▶ `present S then ... end`
- ▶ `nothing`
- ▶ `loop ... end loop`
- ▶ `;`

Derived statements

- ▶ `[weak] abort ... when S`
- ▶ `await S`
- ▶ `...`

The KEP Instruction Set

- ▶ Includes all kernel statements
- ▶ In addition, some derived statements
This redundancy improves space/time efficiency

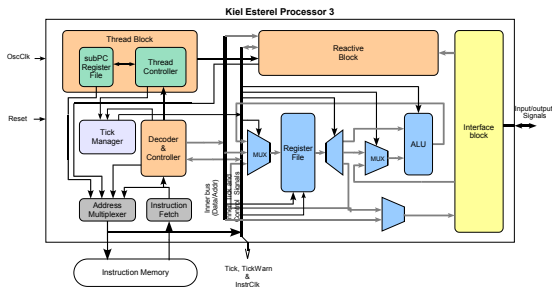
```
TOS:           % trap T in
AO:            % loop
  PAUSE        % pause;
  PRESENT S,A1 % present S then
  EXIT TOE, TOS % exit T
A1:            % end present
  GOTO AO      % end loop
TOE:           % end trap;
```

≡

```
AWAIT S % await S
```

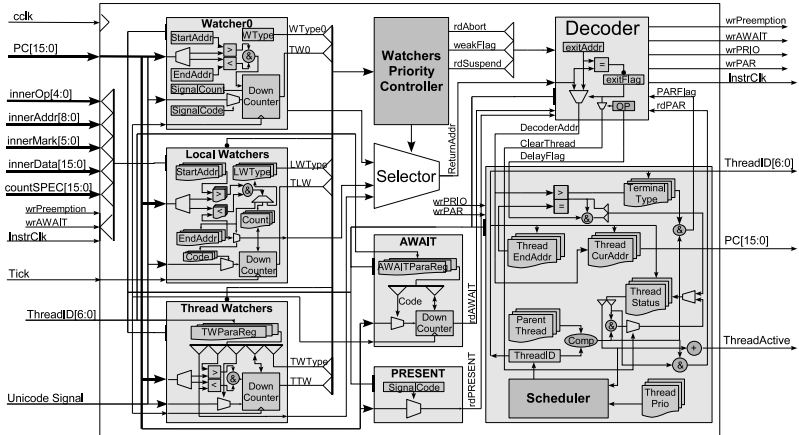
- ▶ **Refined** ISA to reduce HW usage
Example: abort can translate to
 - ABORT** in the most general case
 - LABORT** if no other [L]ABORTS are included in abort scope
 - TABORT** if neither || nor other [L|T]ABORTS are included
- ▶ Furthermore: valued signals, pre, delay expressions, ...

The Kiel Esterel Processor Architecture



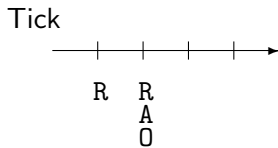
- ▶ Reactive Core
 - ▶ Decoder & Controller, Reactive Block, Thread Block
- ▶ Interface Block
 - ▶ Interface signals, Local signals, ...
- ▶ Data Handling
 - ▶ Register file, ALU, ...

The Architecture of the Reactive Core



The Compilation Challenge: Thread Dependencies

```
module Example:
  output O;
  signal A,R in
  [
    weak abort
      sustain R;
    when immediate A;
    emit O
  ||
    await R;
    emit A
  ];
  end signal
end module
```



The KEP Compiler

Thread scheduling:

1. Construct Concurrent KEP Assembler Graph (CKAG)
2. Compute thread priorities/*ids* that respect dependencies
3. Generate PAR and PRI0 statements accordingly

Other tasks:

- ▶ Analyze Watcher requirements
- ▶ Map Esterel statements to KEP refined ISA
- ▶ Worst Case Reaction Time (WCRT) analysis

Optimizations:

- ▶ Dead code elimination, based on CKAG
- ▶ “Undismantling” of kernel statements

Example Compilation

```

module Example:
  output O;
  signal A,R in
  [
    weak abort
    sustain R;
    when immediate A;
    emit O
  ||
    await R;
    emit A
  ];
end signal
end module
    
```

sustain S

≡

```

loop
  emit S;
  pause;
end loop
    
```

```

loop
  p
end loop
    
```

≡

```

A:
  p;
  goto A
    
```

```

% module Example
OUTPUT O
[L00,T0]  EMIT _TICKLEN,#12
[L01,T0]  SIGNAL A
[L02,T0]  SIGNAL R
[L03,T0]  PAR 2,A0,1
[L04,T0]  PAR 1,A1,2
[L05,T0]  PARE A2,2
[L06,T1]  A0: WABORTI A,A3
[L07,T1]  A4:  EMIT R
[L08,T1]           PRIO 1
[L09,T1]           PRIO 2
[L10,T1]           PAUSE
[L11,T1]           GOTO A4
[L12,T1]  A3:  EMIT O
[L13,T2]  A1:  AWAIT R
[L14,T2]           EMIT A
[L15,T0]  A2:  JOIN O
[L16,T0]           HALT
    
```

Example—Execution Trace

Scheduling criteria: 1. active, 2. highest priority, 3. highest *id*

```
module Example:
  output O;
  signal A,R in
  [
    weak abort
    sustain R;
    when immediate A;
    emit O
  ]
  ||
  await R;
  emit A
];
end signal
end module
```

```
% module Example
OUTPUT O
[L00,T0]   EMIT _TICKLEN,#12
[L01,T0]   SIGNAL A
[L02,T0]   SIGNAL R
[L03,T0]   PAR 2,A0,1
[L04,T0]   PAR 1,A1,2
[L05,T0]   PARE A2,2
[L06,T1]   A0: WABORTI A,A3
[L07,T1]   A4: EMIT R
[L08,T1]           PRIO 1
[L09,T1]           PRIO 2
[L10,T1]           PAUSE
[L11,T1]           GOTO A4
[L12,T1]   A3: EMIT O
[L13,T2]   A1:AWAIT R
[L14,T2]           EMIT A
[L15,T0]   A2:JOIN O
[L16,T0]           HALT
```

```
- Tick 1 -
! reset;
% In:
% Out: R
T0: L01, L02, L03, L04, L05
T1: L06, L07, L08
T2: L13
T1: L09, L10
T0: L15
- Tick 2 -
% In:
% Out: A R O
T1: L10, L11, L07, L08
T2: L13, L14
T1: L09, L10, L12
T0: L15, L16
- Tick 3 -
% In:
% Out:
T0: L16
```


Overview

Introduction

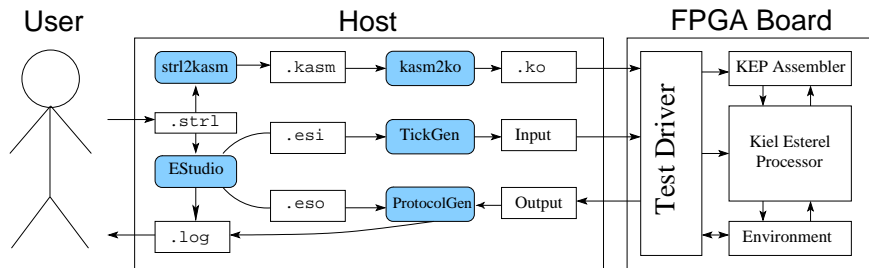
The Kiel Esterel Processor

Experimental Results

KEP Evaluation Platform
Performance
Scalability

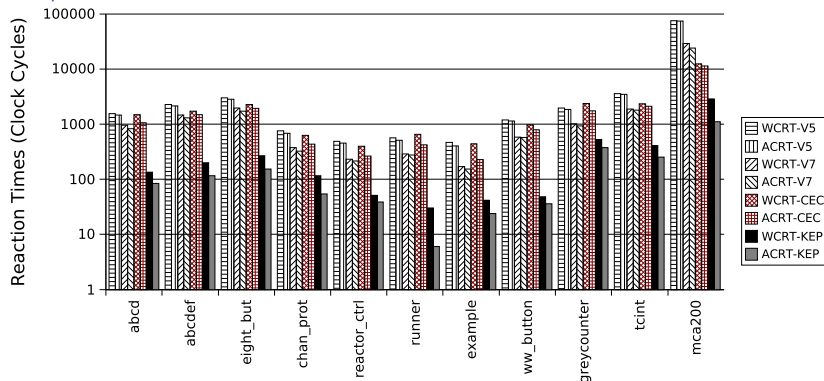
Summary and Outlook

The KEP Evaluation Platform



- ▶ Highly automated process, currently using 470+ benchmarks
- ▶ End to end validation of hardware and compiler against “trusted” reference (Esterel Studio)
- ▶ Detailed performance measurements

Worst/Average Case Execution Times



Comparison of MicroBlaze (using V5/V7/CEC compilers) and KEP

- ▶ WCR speedup: typically $>4x$
- ▶ ACRT speedup: typically $>5x$

Further Comparison with Microblaze

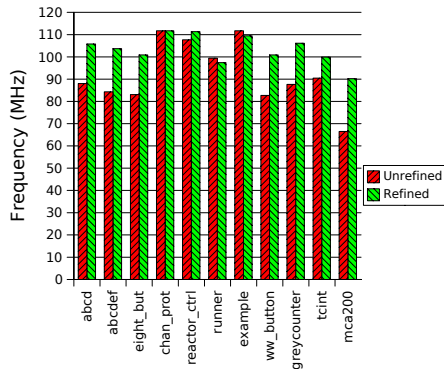
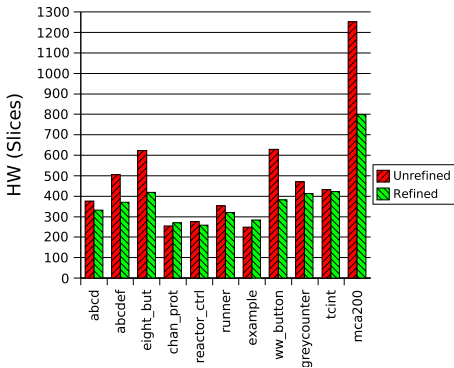
Memory usage

- ▶ **Unoptimized:** 25–94% (83% avg) reduction of memory usage (Code+RAM)
- ▶ **Optimized:** Yield further 5% to 30+% improvements

Power

- ▶ Peak energy usage reduction: 46–84% (75% avg)
- ▶ Idle (= no inputs) energy usage reduction: 58–97% (86% avg)

Efficacy of ISA Refinement



Scalability

Synthesis results for Xilinx 3S1500-4fg-676¹

Thread Count	Slices	Gates (k)
2	1295	295
10	1566	299
20	1871	311
40	2369	328
60	3235	346
80	4035	373
100	4569	389
120	5233	406

- ▶ 48 valued signals
up to 256 possible
- ▶ 2 Watchers, 8 Local Watchers
either up to 64 possible
- ▶ 1k (1024) instruction words
up to 16k possible
- ▶ 128 registers (in word)
up to 512 possible
- ▶ 16-bits (65536) max counter value
- ▶ Frequency is stable (around 60 MHz)

¹For comparison, a MicroBlaze implementation requires around 1k slices and 309k gates; a two threads EMPEROR platform requires around 2k slices

Overview

Introduction

The Kiel Esterel Processor

Experimental Results

Summary and Outlook

KEP Context

Related Work

Summary Reactive Processors

Outlook

Not discussed further today

WCRT analysis

SLA++P'07

- ▶ Compute safe upper bound on instruction cycles per logical tick
- ▶ Judicious traversal of Concurrent KEP Assembler Graph gives linear (in CKAG size) complexity
- ▶ Estimates accurate within 20% on average
- ▶ Compiler annotates code for Tick manager
- ▶ Allows short, constant reaction time—no jitter

HW/SW Co-Design

CODES'07

- ▶ Speedup signal expression computations with external logic block
- ▶ Semantics-preserving source code transformation hoists computations into external modules
- ▶ Use MV-SIS to minimize logic, transform back into Esterel

Related Work

RePIC [Roop et al.'04]/EMPEROR [Yoong et al.'06]

- ▶ Multi-processing patched reactive processor
- ▶ Three-valued signal logic + cyclic executive

Kiel Esterel Processor 1–3

SYNCHRON'04, CASES'05, SAC'06

- ▶ Multi-threading custom reactive processor
- ▶ Provides most Esterel primitives, but still incomplete
- ▶ No compilation scheme to support concurrency

KEP3a

ASPLOS'06

- ▶ Provides all Esterel primitives
- ▶ Refined ISA
- ▶ Compiler exploits multi-threading

Summary Reactive Processors

Processor supports reactive control flow directly, at hardware level

- ▶ “Watchers” monitor preemption signals
No need for polling, interrupts
- ▶ Support for concurrency
Multi-threading or multi-processing
- ▶ Synchronous model of computation
Perfectly deterministic, predictable timing

Outlook

- ▶ Improve priority assignments
- ▶ Extend to Esterel v7
- ▶ KEP in Esterel—*e. g.*, to produce Esterel virtual machine
- ▶ Combination with multi-core (for data handling)
- ▶ Adaptation to non-Esterel languages
- ▶ Further study + formalization of KEP execution model

www.informatik.uni-kiel.de/rtsys/kep

Thanks!

Questions/Comments?

Appendix

KEP3a Instruction Set + Architecture

- Esterel-Type Instructions

- Handling Concurrency

- Handling Preemption

- WCRT Self-Monitoring

The Compiler

- Three Compilation Steps

- The Concurrent KEP Assembler Graph

- Cyclicity

- Constraints

Further Measurements

- Code Characteristics and Compilation Times

- Speed, Size, Power, Scalability

- Analysis of context switches

- Another Example

Summary

- Multi-processing vs. Multi-threading

- Comparison of Synthesis Options

- Application Scenarios

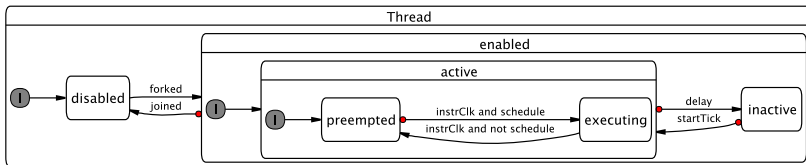
Instruction Set Summary 1/2

Mnemonic, Operands	Esterel Syntax	Notes
PAR <i>Prio</i> , <i>startAddr</i> [, <i>ID</i>] PARE <i>endAddr</i> JOIN	[<i>p</i> <i>q</i>]	Fork and join. An optional <i>ID</i> explicitly specifies the ID of the created thread.
PRIO <i>Prio</i>		Set the priority of the current thread
[W]ABORT [<i>n</i> ,] <i>S</i> , <i>endAddr</i>	[weak] abort ... when [<i>n</i>] <i>S</i>	<i>S</i> can be one of the following: <ol style="list-style-type: none"> 1. <i>S</i>: signal status (present/absent) 2. PRE(<i>S</i>): previous status of signal 3. TICK: always present <i>n</i> can be one of the following: <ol style="list-style-type: none"> 1. #<i>data</i>: immediate data 2. <i>reg</i>: register contents 3. ?<i>S</i>: value of a signal 4. PRE(?<i>S</i>): previous value of a signal
[W]ABORTI <i>S</i> , <i>endAddr</i>	[weak] abort ... when immediate <i>S</i>	
SUSPEND[I] <i>S</i> , <i>endAddr</i>	suspend ... when [immediate] <i>S</i>	
EXIT <i>TrapEnd</i> [, <i>TrapStart</i>]	trap <i>T</i> in exit <i>T</i> end trap	

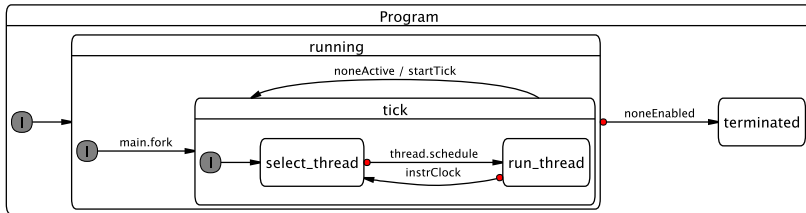
Instruction Set Summary 2/2

Mnemonic, Operands	Esterel Syntax	Notes
PAUSE	pause	Wait for a signal. AWAIT TICK is equivalent to PAUSE
AWAIT [<i>n</i> ,] <i>S</i>	await [<i>n</i>] <i>S</i>	
AWAIT[I] <i>S</i>	await [immediate] <i>S</i>	
CAWAITS CAWAIT[I] <i>S</i> , <i>addr</i> CAWAITE	await case [immediate] <i>S</i> do end	wait for several signals in parallel
SIGNAL <i>S</i>	signal <i>S</i> in ...end	Initialize a local signal <i>S</i>
EMIT <i>S</i> [, {# <i>data</i> <i>reg</i> }]	emit <i>S</i> [(<i>val</i>)]	Emit (valued) signal <i>S</i>
SUSTAIN <i>S</i> [, {# <i>data</i> <i>reg</i> }]	sustain <i>S</i> [(<i>val</i>)]	Sustain (valued) signal <i>S</i>
PRESENT <i>S</i> , <i>elseAddr</i>	present <i>S</i> then ...end	Jump to <i>elseAddr</i> if <i>S</i> is absent
NOTHING	nothing	Do nothing
HALT	halt	Halt the program
GOTO <i>addr</i>	loop ...end loop	Jump to <i>addr</i>
CALL <i>addr</i> RETURN	call <i>P</i>	call a procedure, and return from the procedure

Handling Concurrency



Execution status of a single thread



The status of the whole program, as managed by the Thread Block

Handling Concurrency

A thread has its

- ▶ *thread id*
- ▶ *address range* and independent *program counter*
- ▶ *priority value*
 - ▶ assigned when a thread is created
 - ▶ dynamically changed via `PRIO` instruction
- ▶ *status flags*
 - ▶ `ThreadEnable`
 - ▶ `ThreadActive`

```
% Esterel
[
    p
    ||
    q
];
```

```
% KEP Assembler
    PAR 1,A0,1
    PAR 1,A1,2
    PARE A2
A0: p
A1: q
A2: JOIN
```


Handling Preemption

Watcher contains

Enable Watcher (EW)

- ▶ Watches the PC (Program Counter)
- ▶ Compares PC
- ▶ Preemption *enabled?*

Trigger Watcher (TW)

- ▶ Watches the Signal
- ▶ Counts down the counter (abortion)
- ▶ Preemption *active?*

```
% Esterel
abort
  weak abort
    p;
  when S2;
  q;
when S1;
```

```
% KEP Assembler
ABORT S1,A1
WABORT S2,A0
  p
A0: q
A1:
```

Watcher Refinement

- Thread Watcher
 - ▶ belongs to a thread directly
 - ▶ can neither include concurrent threads nor other preemptions
 - ▶ least powerful, but also cheapest
- Local Watcher
 - ▶ may include concurrent threads and also preemptions handled by a Thread Watcher
 - ▶ cannot include another Local Watcher
- Watcher
 - ▶ may include concurrent threads and any preemptions
 - ▶ most powerful, but also most expensive

Handling Exceptions

Exception

- ▶ has its address range
- ▶ sets an exitFlag
 - ▶ cleared when reaching the end of the trap scope
 - ▶ effects control at the join point
- ▶ can be overridden based on the corresponding trap scopes (address range)

```
% Esterel
trap T1 in
  trap T2 in
    [ p;
      exit T1;
    ||
      q;
      exit T2; ];
  end trap;
  r;
end trap;
```

```
% KEP Assembler
T1S: T2S:
  PAR 1,A1,1
  PAR 1,A2,2
  PARE A3
A1: p
  EXIT T1,T1S
A2: q
  EXIT T2,T2E
A3: JOIN
T2E:r
T1E:
```

WCRT (Tick Length) Self-Monitoring

- ▶ **OscClk**: external clock; **InstrClk**: instructions; **Tick**: logical ticks
- ▶ Emitting special signal **_TICKLEN** configures Tick Manager with WCRT
- ▶ **TickWarn** pin indicates WCRT timing violation

```
% KEP Assembler
% module OVERRUN
```

```
INPUT D
```

```
OUTPUT A,B,C
```

```
EMIT _TICKLEN, #3
```

```
EMIT A
```

```
EMIT B
```

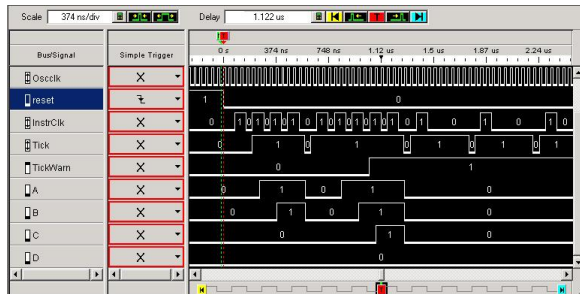
```
PAUSE
```

```
EMIT A
```

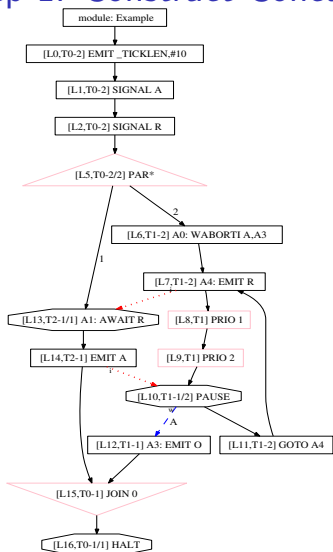
```
EMIT B
```

```
EMIT C
```

```
AWAIT D
```



Step 1: Construct Concurrent KEP Assembler Graph



```

% module Example
OUTPUT 0
[L00 ,T0]   EMIT_TICKLEN ,#12
[L01 ,T0]   SIGNAL A
[L02 ,T0]   SIGNAL R
[L03 ,T0]   PAR 2 ,A0 ,1
[L04 ,T0]   PAR 1 ,A1 ,2
[L05 ,T0]   PARE A2 ,2
[L06 ,T1]   A0 : WABORTI A ,A3
[L07 ,T1]   A4 : EMIT R
[L08 ,T1]   Prio 1
[L09 ,T1]   Prio 2
[L10 ,T1]   PAUSE
[L11 ,T1]   GOTO A4
[L12 ,T1]   A3 : EMIT 0
[L13 ,T2]   A1 :AWAIT R
[L14 ,T2]   EMIT A
[L15 ,T0]   A2 :JOIN 0
[L16 ,T0]   HALT
    
```

Step 2: Compute Thread Priorities/*ids*

- ▶ Compute priority for current tick at each node
- ▶ Compute priority for next tick at tick boundaries
- ▶ Priority within tick must not increase
- ▶ Initialize tick boundaries with lowest priority, compute priorities backwards
- ▶ Judicious traversal of CKAG allows to compute each priority just once
 - ▶ Facilitates correctness argument
 - ▶ Complexity linear in CKAG size

Step 3: Generate PAR/PRIO Statements

- ▶ Enforce that a statement is always executed with same priority, irrespective of control flow
- ▶ Must consider priorities for current and for next tick
- ▶ Again linear complexity

CKAG Node Types

The CKAG distinguishes the following sets of nodes:

D: Delay nodes (octagons)

- ▶ PAUSE, AWAIT, HALT, SUSTAIN

F: Fork nodes (triangles)

- ▶ PAR/PARE

T: Transient nodes (rectangles/inverted triangles)

- ▶ EMIT, PRESENT, etc. (rectangles)
- ▶ JOIN nodes (inverted triangles)

N: Set of all nodes, $N = D \cup F \cup T$

The Concurrent KEP Assembler Graph (CKAG)

Define

- ▶ for each fork node n :
 - n.join**: the JOIN statement corresponding to n ,
 - n.sub**: the transitive closure of nodes in threads generated by n .
- ▶ for abort nodes n (`[L|T] [W] ABORT [I]`, `SUSPEND [I]`):
 - n.end**: the end of the abort scope opened by n ,
 - n.scope**: the nodes within n 's abort scope.
- ▶ for all nodes n :
 - n.prio**: the priority that the thread executing n should be running with
- ▶ for $n \in D \cup F$,
 - n.prioxnext**: the priority that the thread executing n should be resumed with in the subsequent tick.

CKAG Dependency Types

Define dependencies

n.dep_i: the dependency sinks with respect to n at the current tick (the *immediate dependencies*)

n.dep_d: the dependency sinks with respect to n at the next tick (the *delayed dependencies*)

Induced by emissions of strong abort trigger signals and corresponding delay nodes within the abort scope

CKAG Successor Types

Define following types of successors for each n :

$n.suc_c$: the *control successors*.

$n.suc_w$: the *weak abort successors*

$n.suc_s$: the *strong abort successors*

$n.suc_f$: the *flow successors*

the set $n.suc_c \cup n.suc_w \cup n.suc_s$

For $n \in F$ we also define the following *fork abort successors*

$n.suc_{wf}$: the *weak fork abort successors*

$n.suc_{sf}$: the *strong fork abort successors*

Program Cycle

An Esterel program is considered **cyclic** iff the corresponding CKAG contains a path from a node to itself, where for all nodes n and their successors along that path, n' and n'' , the following holds:

$$\begin{aligned}
 & n \in D \wedge n' \in n.suc_w \\
 \vee & n \in F \wedge n' \in n.suc_c \cup n.suc_{wf} \\
 \vee & n \in T \wedge n' \in n.suc_c \cup n.dep_i \\
 \vee & n \in T \wedge n' \in n.dep_d \wedge n'' \in n'.suc_c \cup n'.suc_s \cup n'.suc_{sf}.
 \end{aligned}$$

Constraints

A correct priority assignment must fulfill the following constraints, where m, n are arbitrary nodes in the CKAG

Constraint (Dependencies)

- ▶ For $m \in n.dep_i$: $n.prio > m.prio$
- ▶ For $m \in n.dep_d$: $n.prio > m.prioxnext$

Constraint (Intra-Tick Priority)

- ▶ For $n \in D$ and $m \in n.suc_w$, or $n \in F$ and $m \in n.suc_c \cup n.suc_{wf}$, or $n \in T$ and $m \in n.suc_c$:
 $n.prio \geq m.prio$

Computing Thread Priorities

Constraint (Inter-Tick Priority for Delay Nodes)

- ▶ For all $m \in n.suc_c \cup n.suc_s$: $n.prionext \geq m.prio$

Constraint (Inter-Tick Priority for Fork Nodes)

- ▶ $n.prionext \geq n.join.prio$
- ▶ For all $m \in n.suc_{sf}$: $n.prionext \geq m.prio$

Computing Thread Priorities

```

module Edwards02:
input S, I;
output O;

signal A,R in
every S do
  await I;
  weak abort
  sustain R;
  when immediate A;
  emit O;
||
  loop
    pause;
    pause;
    present R then
      emit A;
    end present
  end loop
end every
end signal
end module
    
```

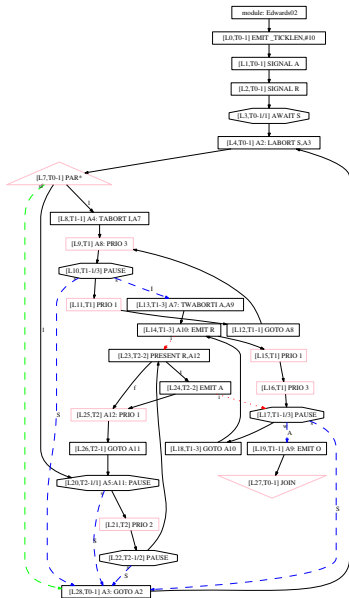


```

module Edwards02-dism:
input S,I;
output O;
signal A, R in
  abort
  loop
    pause
  end loop
  when S;
  loop
    abort
    [ abort
      loop
        pause
      end loop
      when I;
      weak abort
      loop
        emit R;
        pause
      end loop
      when immediate A;
      emit O
    ]
  ||
  % cont...
    
```

```

% cont...
||
  loop
    pause;
    pause;
    present R then
      emit A
    end present
  end loop ];
  loop
    pause
  end loop
  when S
  end loop
end signal
end module
    
```



	INPUT S,I
	OUTPUT 0
[L00,T0]	EMIT_TICKLEN,#20
[L01,T0]	SIGNAL A
[L02,T0]	SIGNAL R
[L03,T0]	AWAIT S
[L04,T0]	A2: LABORT S,A3
[L05,T0]	PAR 1,A4,1
[L06,T0]	PAR 1,A5,2
[L07,T0]	PAUSE A6,1
[L08,T1]	A4: TABORT I,A7
[L09,T1]	A8: PRIO 3
[L10,T1]	PAUSE
[L11,T1]	PRIO 1
[L12,T1]	GOTO A8
[L13,T1]	A7: TWABORTI A,A9
[L14,T1]	A10:EMIT R
[L15,T1]	PRIO 1
[L16,T1]	PRIO 3
[L17,T1]	PAUSE
[L18,T1]	GOTO A10
[L19,T1]	A9: EMIT 0
[L20,T2]	A5:A11: PAUSE
[L21,T2]	PRIO 2
[L22,T2]	PAUSE
[L23,T2]	PRESENT R,A12
[L24,T2]	EMIT A
[L25,T2]	A12:PRIO 1
[L26,T2]	GOTO A11
[L27,T0]	A6: JOIN
[L28,T0]	A3: GOTO A2

Optimized Priority Assignment

```

        INPUT S,I
        OUTPUT 0
[L00,T0]  EMIT _TICKLEN,#20
[L01,T0]  SIGNAL A
[L02,T0]  SIGNAL R
[L03,T0]  AWAIT S
[L04,T0]  A2: LABORT S,A3
[L05,T0]  PAR 1,A4,1
[L06,T0]  PAR 1,A5,2
[L07,T0]  PARE A6,1
[L08,T1]  A4: TABORT I,A7
[L09,T1]  A8: PRIO 3
[L10,T1]  PAUSE
[L11,T1]  PRIO 1
[L12,T1]  GOTO A8
[L13,T1]  A7: TWABORTI A,A9
[L14,T1]  A10:EMIT R
[L15,T1]  PRIO 1
[L16,T1]  PRIO 3
[L17,T1]  PAUSE
[L18,T1]  GOTO A10
[L19,T1]  A9: EMIT 0
[L20,T2]  A5:A11: PAUSE
[L21,T2]  PRIO 2
[L22,T2]  PAUSE
[L23,T2]  PRESENT R,A12
[L24,T2]  EMIT A
[L25,T2]  A12:PRIO 1
[L26,T2]  GOTO A11
[L27,T0]  A6: JOIN
[L28,T0]  A3: GOTO A2
    
```



```

        INPUT S,I
        OUTPUT 0
[L00,T0]  EMIT _TICKLEN,#20
[L01,T0]  SIGNAL A
[L02,T0]  SIGNAL R
[L03,T0]  AWAIT S
[L04,T0]  A2: LABORT S,A3
[L05,T0]  PAR 3,A4,1
[L06,T0]  PAR 2,A5,2
[L07,T0]  PARE A6,1
[L08,T1]  A4: AWAIT I
[L09,T1]  A7: TWABORTI A,A9
[L10,T1]  A10:EMIT R
[L11,T1]  PRIO 1
[L12,T1]  PRIO 3
[L13,T1]  PAUSE
[L14,T1]  GOTO A10
[L15,T1]  A9: EMIT 0
[L16,T2]  A5:A11: PAUSE
[L17,T2]  PAUSE
[L18,T2]  PRESENT R,A12
[L19,T2]  EMIT A
[L20,T2]  A12:GOTO A11
[L21,T0]  A6: JOIN
[L22,T0]  A3: GOTO A2
    
```

Code Characteristics and Compilation Times

Module Name	Esterel				KEP3a (Unoptimized optimized)								MicroBlaze Compiling Time (Sec) (V5/V7/CEC)			
	Threads		Preemptions		CKAG				Preemption handled by			Compiling Time (Sec)				
	Cnt	Max Depth	Max Conc	Cnt	Max Depth	Nodes	Dep. Num	Max Priority	PRIO Instr	Local Watcher	Thread Watcher			Thread Watcher		
abcd	4	2	4	20	2	211	36	3	30	0	4 3	16 11	0.15	0.12	0.09	0.30
abcdef	6	2	6	30	2	313	90	3	48	0	6 5	24 17	0.21	0.71	0.46	0.96
eight_but	8	2	8	40	2	415	168	3	66	0	8 7	32 23	0.26	0.99	0.54	1.25
chan_prot	5	3	4	6	1	80	4	2	10	0	0	6 4	0.07	0.35	0.35	0.43
reactor_ctrl	3	2	3	5	1	51	5	1	0	0	1 0	4	0.06	0.29	0.31	0.36
runner	2	2	2	9	3	61	0	1	0	3 2	1	5 3	0.05	0.30	0.34	0.40
example	2	2	2	4	2	36	2	3	6	0	1	3 2	0.05	0.28	0.31	0.31
ww_button	13	3	4	27	2	194	0	1	0	0	5	22 10	0.10	0.44	0.40	0.64
greycounter	17	3	13	19	2	414	53	6	58	0	4	15	0.34	0.57	0.43	0.75
tcint	39	5	17	18	2	583	65	3	20	0	1	17 10	0.34	0.41	0.52	1.11
mca200	59	5	49	64	4	11219	129	11	190	2	14	48	11.25	69.81	12.99	7.37

Note: In the `mca200`, the watcher refinement reduces the hardware requirements from 4033 slices (if all preemptions were handled by general purpose Watchers) to 3265 slices (19% reduction).

Worst-/Average-Case Reaction Times

Module Name	MicroBlaze						KEP3a-Unoptimized				KEP3a-optimized			
	WCRT			ACRT			WCRT Ratio to		ACRT Ratio to		WCRT Ratio to		ACRT Ratio to	
	V5	V7	CEC	V5	V7	CEC	best MB		best MB		Unopt		to Unopt	
abcd	1559	954	1476	1464	828	1057	135	0.14	87	0.11	135	1	84	0.97
abcdef	2281	1462	1714	2155	1297	1491	201	0.14	120	0.09	201	1	117	0.98
eight_but	3001	1953	2259	2833	1730	1931	267	0.14	159	0.09	267	1	153	0.96
chan_prot	754	375	623	683	324	435	117	0.31	60	0.19	117	1	54	0.90
reactor_ctrl	487	230	397	456	214	266	54	0.23	45	0.21	51	0.94	39	0.87
runner	566	289	657	512	277	419	36	0.12	15	0.05	30	0.83	6	0.40
example	467	169	439	404	153	228	42	0.25	24	0.16	42	1	24	1
ww_button	1185	578	979	1148	570	798	72	0.12	51	0.09	48	0.67	36	0.71
greycounter	1965	1013	2376	1851	928	1736	528	0.52	375	0.40	528	1	375	1
tcint	3580	1878	2350	3488	1797	2121	408	0.22	252	0.14	342	0.84	204	0.81
mca200	75488	29078	12497	73824	24056	11479	2862	0.23	1107	0.10	2862	1	1107	1

The worst-/average-case reaction times, in clock cycles, for the KEP3a and MicroBlaze

- ▶ WCRT speedup: typically >4x
- ▶ ACRT speedup: typically >5x
- ▶ Optimizations yield further improvements

Memory Usage

Module Name	Esterel LOC [1]	MicroBlaze			KEP3a-Unopt.				KEP3a-opt.	
		Code+Data (byte)			Code (word)		Code+Data (byte)		Code (word)	
		V5	V7	CEC	abs.	rel.	abs.	rel.	abs.	rel.
		[2] (<i>best</i>)		[3]	[3]/[1]	[4]	[4]/[2]	[5]	[5]/[3]	
abcd	160	6680	7928	7212	168	1.05	756	0.11	164	0.93
abcdef	236	9352	9624	9220	252	1.07	1134	0.12	244	0.94
eight_but	312	12016	11276	11948	336	1.08	1512	0.13	324	0.94
chan_prot	42	3808	6204	3364	66	1.57	297	0.09	62	0.94
reactor_ctrl	27	2668	5504	2460	38	1.41	171	0.07	34	0.89
runner	31	3140	5940	2824	39	1.22	175	0.06	27	0.69
example	20	2480	5196	2344	31	1.55	139	0.06	28	0.94
ww.button	76	6112	7384	5980	129	1.7	580	0.10	95	0.74
greycounter	143	7612	7936	8688	347	2.43	1567	0.21	343	1
tcint	355	14860	11376	15340	437	1.23	1968	0.17	379	0.87
mca200	3090	104536	77112	52998	8650	2.79	39717	0.75	8650	1

- ▶ **Unoptimized:** 83% avg reduction of memory usage (Code+RAM)
- ▶ **Optimized:** May yield further 5% to 30+% improvements

Power Consumption

Module Name	MicroBlaze (82mW@50MHz)	KEP3a ² (mW)		Ratio (KEP to MB)	
	Idle	Peak	Idle	Peak	Idle
abcd	69	13	8	0.16	0.12
abcdef	74	13	7	0.16	0.09
eight_but	74	13	7	0.16	0.09
chan_prot	70	28	12	0.34	0.17
reactor_ctrl	76	20	13	0.24	0.17
runner	78	14	2	0.17	0.03
example	77	25	9	0.30	0.12
ww_button	81	13	4	0.16	0.05
greycounter	78	44	33	0.54	0.42
tcint	80	18	10	0.22	0.13

- ▶ Peak energy usage reduction: 75% avg
- ▶ Idle (= no inputs) energy usage reduction: 86% avg

²Based on Xilinx 3S200-4ft256, requires an additional 37mW as quiescent power for the chip itself

Scalability

Synthesis results for Xilinx 3S1500-4fg-676³

Thread Count	Slices	Gates (k)
2	1295	295
10	1566	299
20	1871	311
40	2369	328
60	3235	346
80	4035	373
100	4569	389
120	5233	406

- ▶ 48 valued signals
up to 256 possible
- ▶ 2 Watchers, 8 Local Watchers
either up to 64 possible
- ▶ 1k (1024) instruction words
up to 64k possible
- ▶ 128 registers (in word)
up to 512 possible
- ▶ 16-bits (65536) max counter value
- ▶ Frequency is stable (around 60 MHz)

Note: In the `mca200`, the watcher refinement reduces the hardware requirements from 4033 slices (if all preemptions were handled by general purpose Watchers) to 3265 slices (19% reduction).

³For comparison, a MicroBlaze implementation requires around 1k slices and 309k gates; a two threads EMPEROR platform requires around 2k slices

Analysis of Context Switches

Module Name	Instr's total abs. [1]	CSs total		CSs at same priority		PRI0s total		CSs due to PRI0		
		abs. [2]	ratio [1]/[2]	abs. [3]	rel. [3]/[2]	abs. [4]	rel. [4]/[1]	abs. [5]	rel. [5]/[2]	rel. [5]/[4]
abcd	16513	3787	4.36	1521	0.40	3082	0.19	1243	0.33	0.40
abcdef	29531	7246	4.08	3302	0.46	6043	0.20	2519	0.35	0.42
eight_but	39048	10073	3.88	5356	0.53	8292	0.21	3698	0.37	0.45
chan_prot	5119	1740	2.94	707	0.41	990	0.19	438	0.25	0.44
reactor_ctrl	151	48	3.15	29	0.60	0	0	0	0	-
runner	5052	704	7.18	307	0.44	0	0	0	0	-
example	208	60	3.47	2	0.30	26	0.13	9	0.15	0.35
ww_button	292	156	1.87	92	0.59	0	0	0	0	-
greycounter	160052	34560	4.63	14043	0.41	26507	0.17	12725	0.37	0.48
tcint	80689	33610	2.4	16769	0.50	5116	0.06	2129	0.06	0.42
mca200	982417	256988	3.82	125055	0.49	242457	0.25	105258	0.41	0.43

Edwards02: Esterel to KEP

```

module Edwards02:
input S, I;
output O;

signal A,R in
every S do
await I;
weak abort
sustain R;
when immediate A;
emit O;
||
loop
pause;
pause;
present R then
emit A;
end present
end loop
end every
end signal
end module
    
```

```

every S do
P
end
    
```

```

await S;
loop
abort
P;
halt
when S
end loop
    
```

```

sustain S
    
```

```

loop
emit S;
pause;
end loop
    
```

```

loop
P
end loop
    
```

```

A:
P;
goto A
    
```

```

INPUT S,I
OUTPUT O
[L00,T0] EMIT _TICKLEN,#20
[L01,T0] SIGNAL A
[L02,T0] SIGNAL R
[L03,T0] AWAIT S
[L04,T0] A2: LABORT S,A3
[L05,T0] PAR 1,A4,1
[L06,T0] PAR 1,A5,2
[L07,T0] PARE A6,1
[L08,T1] A4: TABORT I,A7
[L09,T1] A8: PRIO 3
[L10,T1] PAUSE
[L11,T1] PRIO 1
[L12,T1] GOTO A8
[L13,T1] A7: TWABORTI A,A9
[L14,T1] A10:EMIT R
[L15,T1] PRIO 1
[L16,T1] PRIO 3
[L17,T1] PAUSE
[L18,T1] GOTO A10
[L19,T1] A9: EMIT O
[L20,T2] A5:A11: PAUSE
[L21,T2] PRIO 2
[L22,T2] PAUSE
[L23,T2] PRESENT R,A12
[L24,T2] EMIT A
[L25,T2] A12:PRIO 1
[L26,T2] GOTO A11
[L27,T0] A6: JOIN
[L28,T0] A3: GOTO A2
    
```


Edwards02: a Possible Execution Trace

```

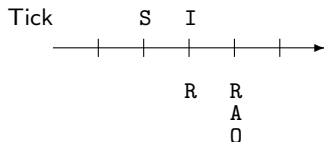
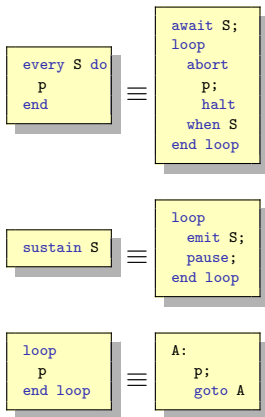
module Edwards02:
input S, I;
output O;

signal A,R in
every S do
    P
end

await I;
loop
    abort
    P;
    halt
    when S
end loop

weak abort
sustain R;
when immediate A;
emit O;

||
loop
    pause;
    pause;
    present R then
        emit A;
    end present
end loop
end every
end signal
end module
    
```



```

module Edwards02:
input S, I;
output O;

signal A,R in
every S do
    await I;
    weak abort
        sustain R;
    when immediate A;
    emit O;
||
loop
    pause;
    pause;
    present R then
        emit A;
    end present
end loop
end every
end signal
end module
    
```

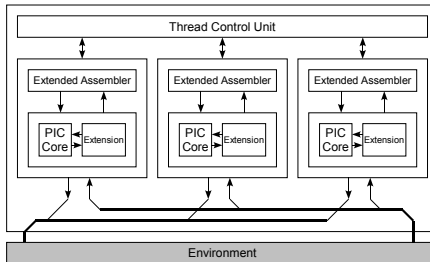
```

                INPUT S,I
                OUTPUT O
[L00,T0]        EMIT _TICKLEN,#20
[L01,T0]        SIGNAL A
[L02,T0]        SIGNAL R
[L03,T0]        AWAIT S
[L04,T0]        A2: LABORT S,A3
[L05,T0]        PAR 1,A4,1
[L06,T0]        PAR 1,A5,2
[L07,T0]        PARE A6,1
[L08,T1]        A4: TABORT I,A7
[L09,T1]        A8: PRIO 3
[L10,T1]        PAUSE
[L11,T1]        PRIO 1
[L12,T1]        GOTO A8
[L13,T1]        A7: TWABORTI A,A9
[L14,T1]        A10:EMIT R
[L15,T1]        PRIO 1
[L16,T1]        PRIO 3
[L17,T1]        PAUSE
[L18,T1]        GOTO A10
[L19,T1]        A9: EMIT O
[L20,T2]        A5:A11: PAUSE
[L21,T2]        PRIO 2
[L22,T2]        PAUSE
[L23,T2]        PRESENT R,A12
[L24,T2]        EMIT A
[L25,T2]        A12:PRIO 1
[L26,T2]        GOTO A11
[L27,T0]        A6: JOIN
[L28,T0]        A3: GOTO A2
    
```

```

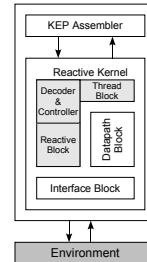
- Tick 1 -
! reset;
% In:
% Out:
[L01,T0] [L02,T0] [L03,T0]
- Tick 2 -
% In: S
% Out:
[L03,T0] [L04,T0] [L05,T0]
[L06,T0] [L07,T0]
[L20,T2] [L08,T1]
[L09,T1] [L10,T1]
[L27,T0]
- Tick 3 -
% In: I
% Out: R
[L10,T1] [L13,T1]
[L14,T1] [L15,T1]
[L20,T2] [L21,T2] [L22,T2]
[L16,T1] [L17,T1] [L27,T0]
- Tick 4 -
% In:
% Out: A R O
[L17,T1] [L18,T1]
[L14,T1] [L15,T1]
[L22,T2] [L23,T2] [L24,T2]
[L25,T2] [L26,T2] [L20,T2]
[L16,T1] [L17,T1] [L19,T1]
[L27,T0]
    
```

Multi-processing vs. Multi-threading



Multi-processing (EMPEROR/RePIC)

- ▶ Esterel thread \approx one independent RePIC processor
- ▶ Thread Control Unit handles the synchronization and communication
- ▶ Three-valued signal representation
- ▶ **sync** command to synchronize threads



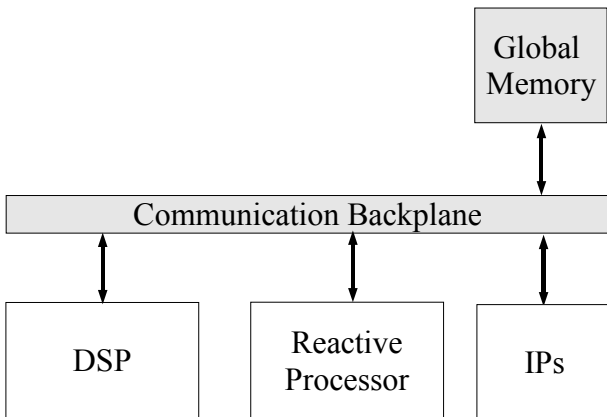
Multi-threading (KEP)

- ▶ Esterel thread \approx several registers
- ▶ priority-based scheduler
- ▶ **PRIO** command to synchronize threads

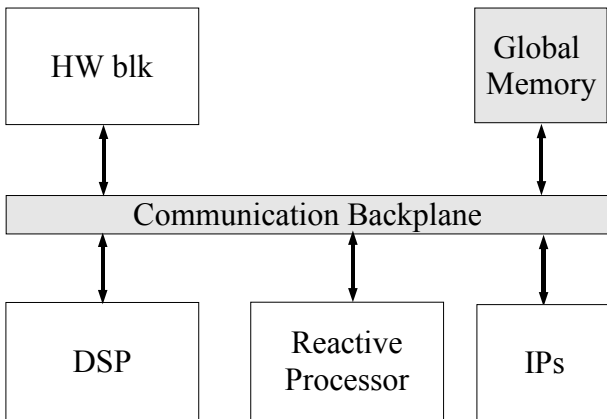
Comparison of Synthesis Options

		HW	SW	Co-design	Reactive Processor	
					Multi-processing	Multi-threading
Speed		++	-	+	+	+
Flexibility		--	++	-	+/-	+
Scalability		+	++	+	--	+
Cost	Logic Area	++/-	+	+	--	+/-
	Memory	++	--	-	+	+
	Power Usage	++	-	-	--	+
Appl.	Design Cycle	--	++	+/-	++	++

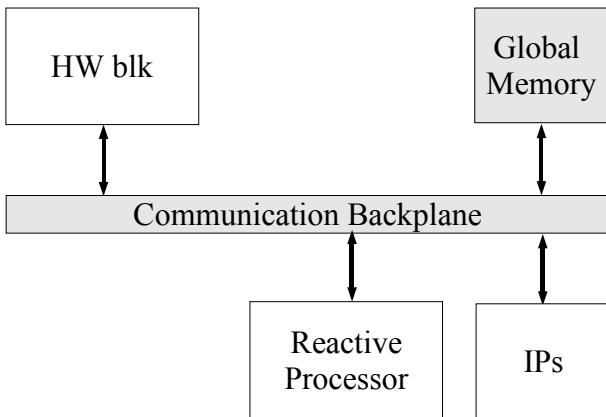
Scenario I: DSP + Reactive Processor



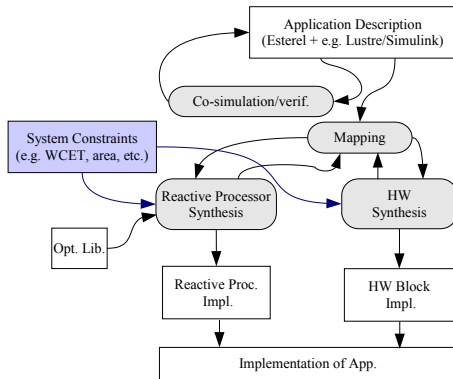
Scenario II: DSP + HW Block + Reactive Processor



Scenario III: HW Block + Reactive Processor



Possible Co-Design Development Flow



Reactive processing ...

- ▶ permits a simple mapping strategy
- ▶ allows optimizations on high-level
- ▶ can meet stricter constraints than classical architectures
- ▶ permits a better tradeoff between all cost factors