From Lustre V6 to FPGA via Verilog (WIP)

Erwan Jahier, Bruno Ferres

Émile Guillaume (2 months internship)

November 2025

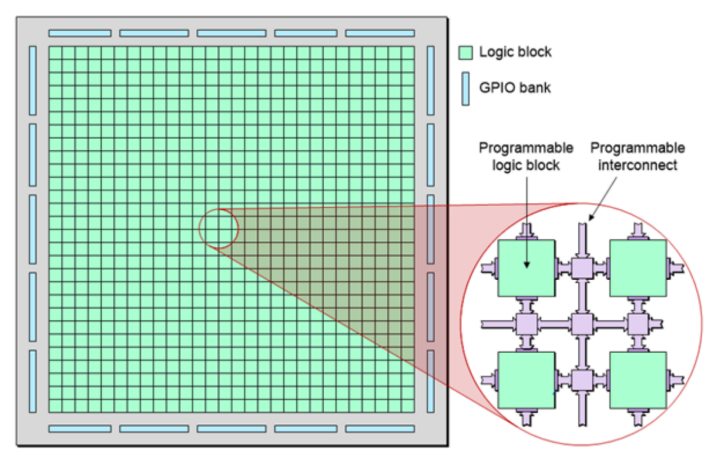
Motivations

- Synchronous Dataflow Languages looks like circuits
- Use FPGA to speed-up (parallel) computations
- Verilog / VHDL
  - De facto standard
  - Dataflow languages
  - A target for other tools
  - A portable assembler for FPGA (and ASICS)

FPGA

Field Programmable Gate arrays

- Programmable circuits
- [CPU , dedicated ASIC]
- Efficiency/flexibility tradeoff
- Inherently parallel processing
- Vendors: Altera, Xilinx (AMD), . . .



Verilog: a lot of flaws as a programming language

- a lot of constructs can not be **synthesized** (simulation)
- black-box
- slow
- more focused on efficiency than on correctness
- semantics issues [OOPSLA23]
- not reproducible

Verilog, a portable assembler

- Tools targeting Verilog/VHDL
  - Verilog extensions: SystemVerilog, BlueSpec, Genesis, TL-Verilog, etc.
  - Imperative Langages : based on python, java, C++, etc.
  - functionnal langages : **Eclat**, HML, Chisel, DFiant, VeriScala, SpinalHDL, lava, etc.
    $\Rightarrow$ heuristics to parallelize sequential code

What about using intrinsically parallel/synchronous data-flow descriptions directly?

Not a new idea: Lustre V4 to FPGA via PAM

Implementing reactive programs on circuits
A hardware implementation of LUSTRE

Frédéric Rocheteau — DEC / Paris Research Laboratory
Nicolas Halbwachs — IMAG / LGI - Grenoble

Abstract

Synchronous languages constitute effective tools for programming real-time systems as far as they can be efficiently implemented. Implementing them by hardware is of course a good way for increasing their performances. Moreover, configurable hardware is now available which makes practical such an implementation. This paper describes an implementation of the synchronous declarative language LUSTRE on a "programmable active memory".

Keywords: Reactive systems, synchronous languages, silicon compilation

Contents

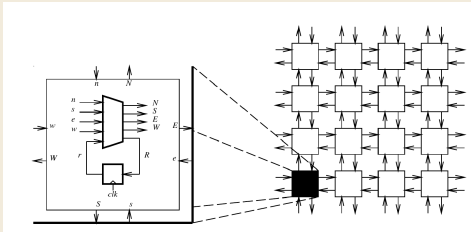
1 Introduction

Synchronous programming [BCG87] has been proposed as a paradigm for designing reactive systems. It is an abstract point of view about real-time, which consists of assuming that a program *instantly* reacts to external events (synchrony hypothesis). It allows providing programs with precise, deterministic and machine-independent semantics. Several programming languages have been designed according to this point of view, e.g., STATECHARTS [Har84], ESTEREL [BG85], SML [BC85], SIGNAL [BL80] and LUSTRE [CPHP87].

In practice, an implementation on a given machine satisfies the synchrony hypothesis if the reaction time is always shorter than the minimum delay separating two successive external events. So, the only real-time problem with a synchronous program is to minimize and measure its reaction time. A specific compiling technique has been proposed [BG85]

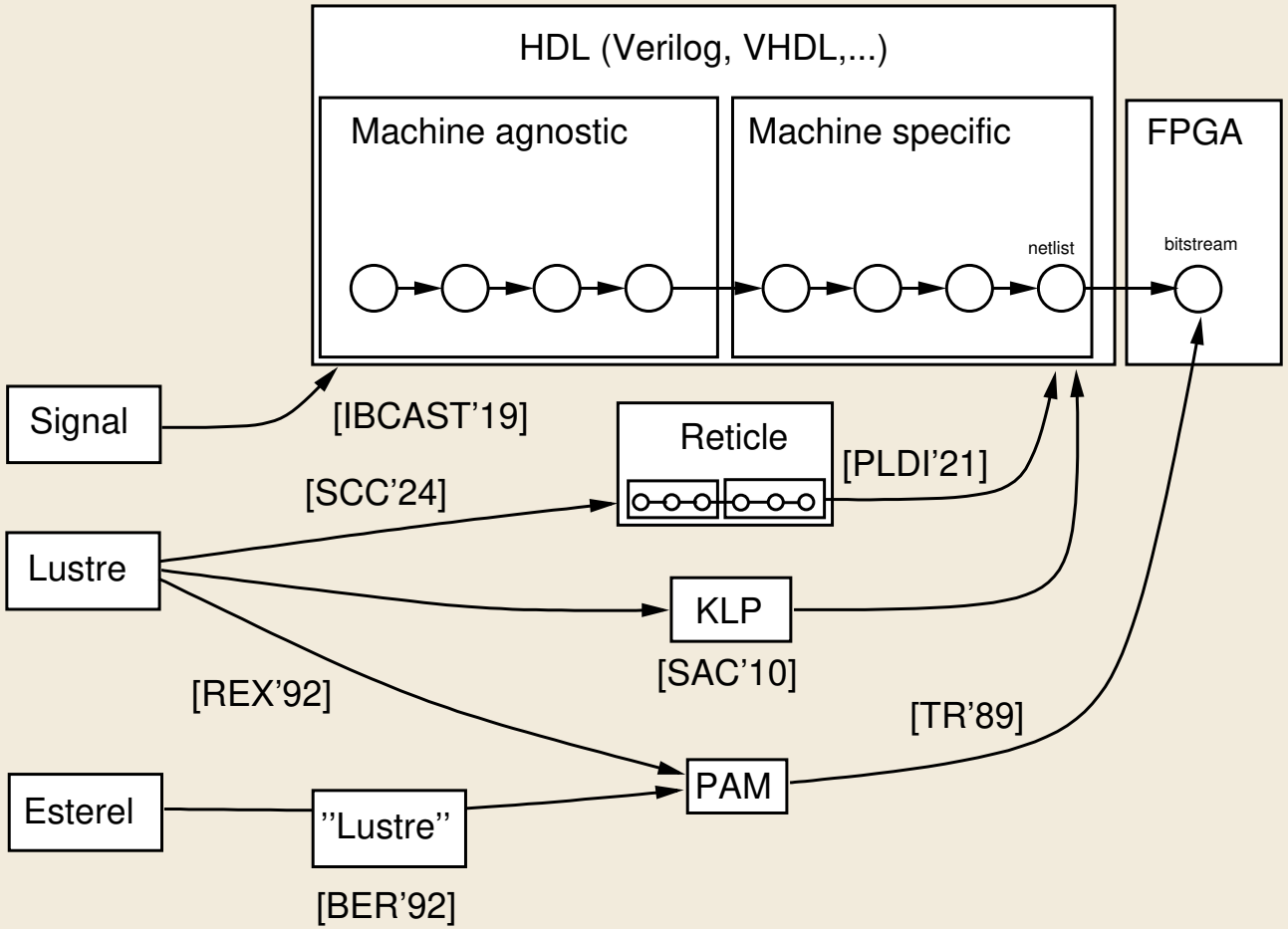
Rocheteau and Halbwachs 1992 - Implementing reactive programs on circuits a hardware implementation of Lustre

- targets PAM (Programmable Active Memory)
  - FPGA Abstract machine
  - Array of PABs (Programmable Active Bits)

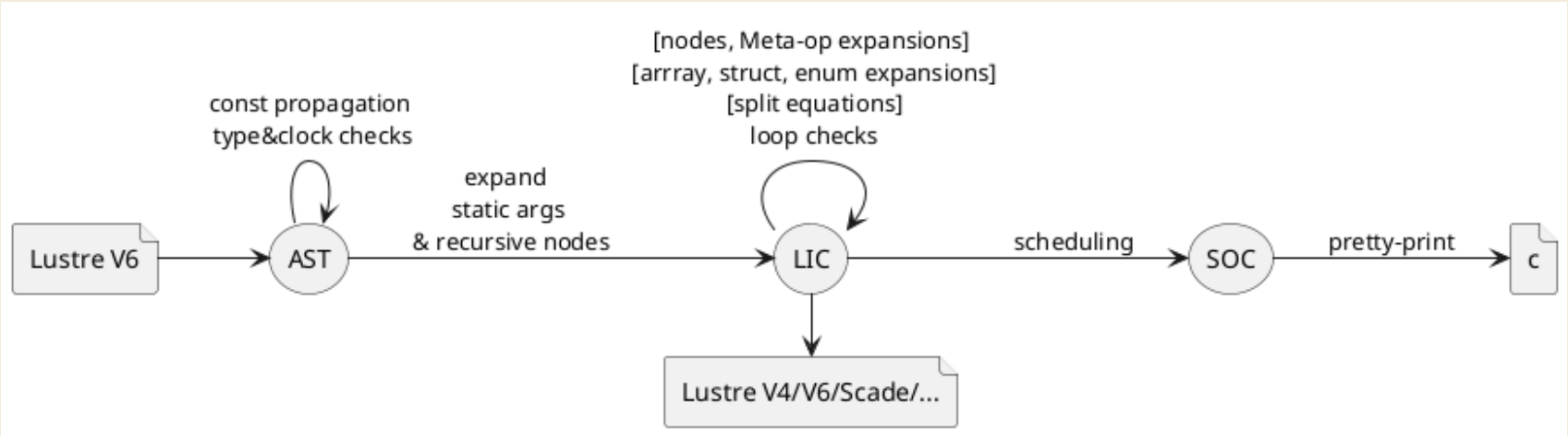


- **heuristics** to pack Lustre equations into PABs

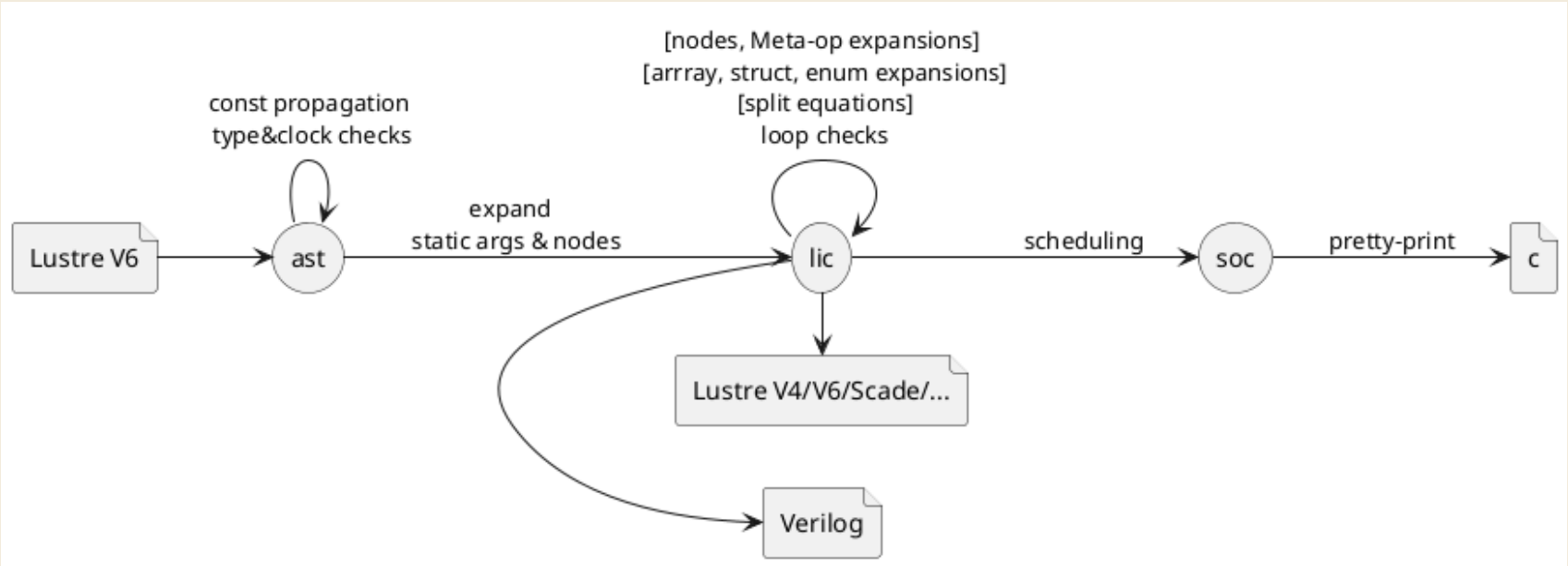
From Synchronous languages to FPGAs



Lustre V6 compiler architecture



Lustre V6 compiler architecture



Example: rising edge

Lustre

```
node rising_edge(x : bool)
returns (res: bool);
let
 res = false -> (x and not pre(x));
tel
```

Verilog

```
module rising_edge (
  input wire veri_rst, // to deal with ->
  input wire veri_clk, // to deal with reg
  input wire x,
  output wire res
 );
  reg pre_x;
  assign res = veri_rst ? (1'b0) : (x && !(pre_x));
  always @(posedge veri_clk)
  // the verilog way to deal with registers
  begin
    pre_x <= x;
  end
endmodule
```

Example: watchdog

Lustre

```
node watchdog(on,off,ms: bool;
              delay: int)
returns (alarm: bool);
var
 active: bool;
 remaining: int;
let
 alarm = active and (remaining = 0);
 active = if on then true
          else if off then false
          else(false -> pre(active));
 remaining = if on then delay
             else if active and ms
             then pre(remaining)-1
             else pre(remaining);
tel
```

Verilog

```
module watchdogV (
 input wire  veri_rst,
 input wire  veri_clk,
 input wire  _Von,
 input wire  _Voff,
 input wire  _Vms,
 input wire signed [63:0] _Vdelay,
 output wire  _Valarm
 );
 wire   _Vactive;
 wire signed [63:0] _Vremaining;
 reg   pre_Vactive;
 reg signed [63:0] pre_Vremaining;
 assign _Valarm = (_Vactive && (_Vremaining == 0));
 assign _Vactive = _Von ? 1'b1
                 : _Voff ? 1'b0
                 : veri_rst ? (1'b0)
                 : pre_Vactive;
 assign _Vremaining = veri_rst ? 0
         : _Von ? _Vdelay
         : (_Vactive && _Vms) ? (pre_Vremaining-1)
         : pre_Vremaining;
 always @(posedge veri_clk) begin
    pre_Vactive <= _Vactive;
    pre_Vremaining <= _Vremaining;
 end
endmodule
```

Example: `current` and `when`

Lustre

```
node test(a:int; clk:bool) returns(b:int)
 var tmp : int when clk;
 let
  tmp = a when clk;
  b = current(tmp);
 tel;
```

Verilog

```
module testV (
 input wire  veri_rst,
 input wire  veri_clk,
 input wire signed [63:0] _Va,
 input wire  _Vclk,
 output wire signed [63:0] _Vb
);
   wire signed [63:0] _Vtmp;
   reg signed [63:0] pre_Vtmp;
   assign _Vb = _Vtmp;
   assign _Vtmp = _Vclk ? (_Va) : pre_Vtmp;
   always @(posedge veri_clk)
   begin
      if (_Vclk) begin
         pre_Vtmp <= _Vtmp;
      end
   end
endmodule
```

Example: call a node

Lustre

```
node A(i: int) returns (res: int);
let
  res = i+1;
tel

node simple_call (i: int) returns (res: int);
let
  res = A(i);
tel
```

Verilog

```
module simple_callV (
 input wire veri_rst,
 input wire veri_clk,
 input wire signed [63:0] _Vi,
 output wire signed [63:0] _Vres
);
AV _AV0 (
 ._Vi(_Vi),
 ._Vres(_Vres),
 .veri_clk(veri_clk),
 .veri_rst(veri_rst)
);
endmodule

module AV (
 input wire veri_rst,
 input wire veri_clk,
 input wire signed [63:0] _Vi,
 output wire signed [63:0] _Vres
);
 assign _Vres = (_Vi + 1);
endmodule
```

Lustre V6 to verilog: mostly straightforward

Lustre	Verilog	

`bool` I/O/local var
`int` I/O/local var
`node`
predef op
`current`
	`wire`
	`wire signed [63:0]`
	`module`
	predef op
		direct

`pre`	`always @(posedge veri_clk)`	
`when` `merge`		less direct
enumerated types	encoded into bool arrays	not hard
struct and arrays parametric nodes etc.		expanded using specific `lv6` options
`real`		not supported

Reproductibility

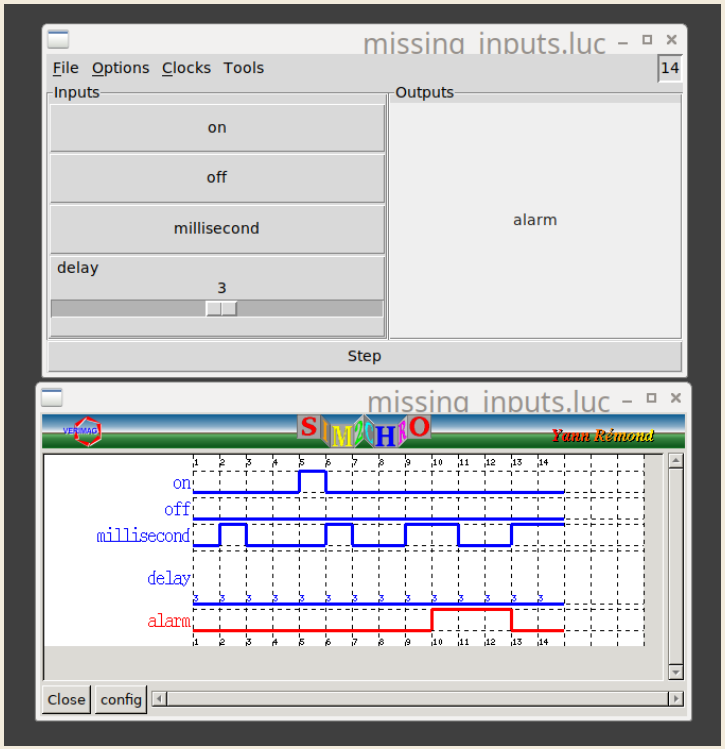
```
guix time-machine -C channels.scm -- shell -CP -m guixVerilo/manifest.scm
```

- `manifest.scm`: the necessary tools (`lustre-v6`, `verilator`, etc.)
- `channels.scm`: a pin of tools versions and their dependencies

https://verimag.gricad-pages.univ-grenoble-alpes.fr/vtt/articles/lustre-verilog/

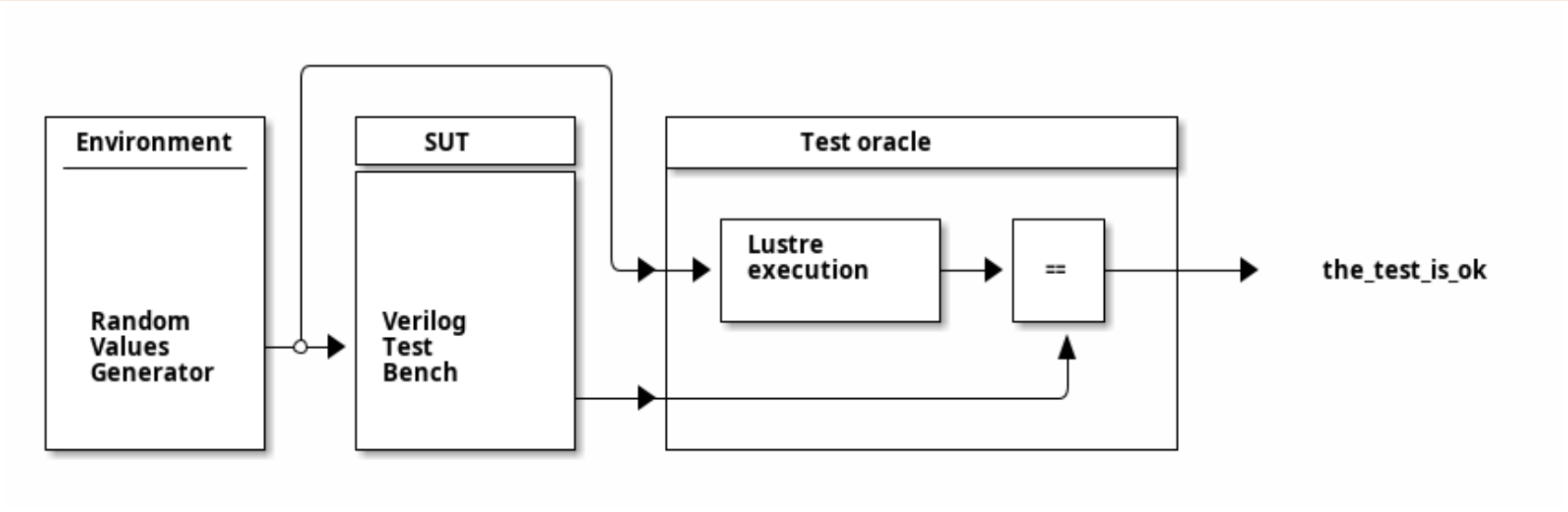
Simulation of the generated Verilog

```
cp ../test/should_work/watchdog.lus .
lv6 -verilog watchdog.lus -n watchdog
generate_tb watchdog.v watchdogV watchdog-tb.cpp
verilator --cc watchdog.v --exe watchdog-tb.cpp --build > /dev/null
./obj_dir/Vwatchdog
luciole -pipe ./obj_dir/Vwatchdog
```



Automated Tests

- Use the Lustre execution as a test oracle



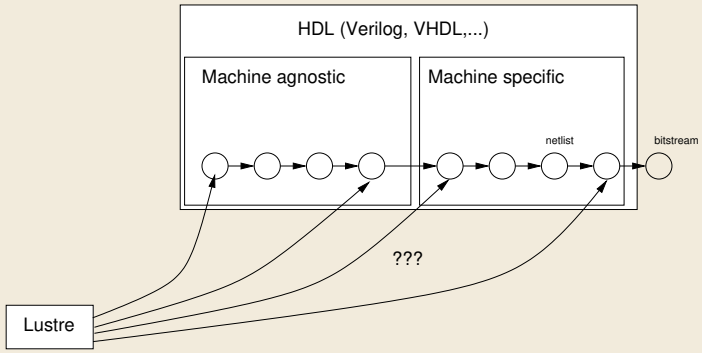
- On 180 programs of the `lv6` non-regression tests suite
  - 10 steps with random values for inputs

Future work

- Explore the influence of options on perf/surface
  - node expansion
  - array expansion
- Implement specific optimizations
  - arrays
  - bit width
- Experiment on a real platform
  - yosys+nextPNR (verilog $\rightarrow$ bitstream)
  - xilinx zybo z7020

Future work

- Generate lower-level verilog
  - dsp



- Automated pipelining
- Target heterogeneous architecture CPU/FPGA(/GPU) (via annotations)
- WCET (WCRT)

The end

more questions?