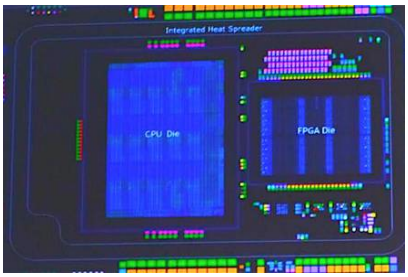
High Level Synthesis
A Dataflow Approach

J. Sérot
Institut Pascal
U. Clermont Auvergne / CNRS
Clermont-Ferrand, France

Context

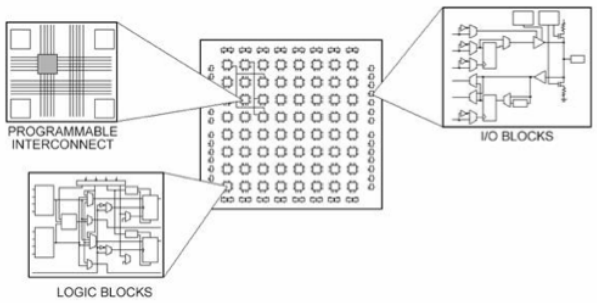
- Most of embedded applications — including CPS ! — are implemented using both software and hardware
  - current trend in CPU technology is many cores + FPGA fabric

- Hardware implementation is beneficial / required for
  - performance (latency, throughput, …)
  - power consumption
  - confidentiality



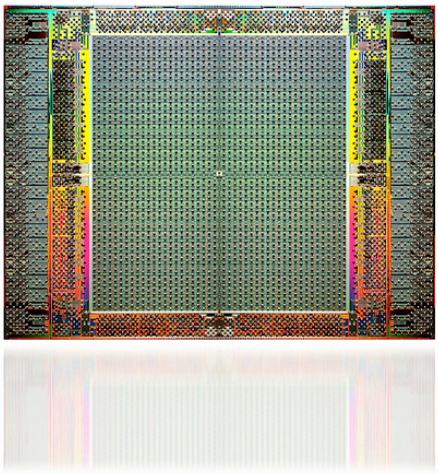
FPGAs

- Field Programmable Gate Arrays
- Most widely used hardware target (cost, reprogrammability, ...)
- Programming means configuring
  - specialize the function of *Logic Elements*
  - configure the communication wires betwen logic elements
- Classically done with hardware description languages (VHDL, Verilog) using RT-level descriptions



FPGAs

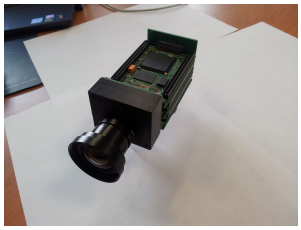
Ex: ALTERA Stratix IV™



- 530K logic elements
- 430K registers
- 20Mb embedded memory
- 1024 18x18 multipliers

➜ huge amount of fine grain parallelism

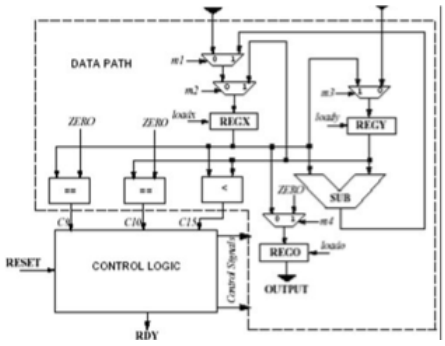
➜ close-to-sensor, on-the-fly, processing



HLS : What is it ?

- High Level Synthesis
- Technology allowing obtention of an hardware implementation of a system directly from a high level specification (typically C-like)
- ... without the need to go through a RTL (VHDL,Verilog) description

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {
  ...

  acc=0;
  loop: for (i=3;i>=0;i--) {
     if (i==0) {
       acc+=x*c[0];
       shift_reg[0]=x;
     } else {
       shift_reg[i]=shift_reg[i-1];
       acc+=shift_reg[i]*c[i];
     }
  }
  *y=acc>>2;
```



Why HLS matters

- Key technology to allow heterogeneous platforms to be programmed by « software » designers
  - the amount of skills required to design hardware using HDLs is the main cause for the productivity gap between software and hardware
- Complex task
  - some reasons are « cultural »
  - ... but some are more fundamental

HLS : Why is it difficult ?

A C function performing image binarisation

```
void binarize(
  pixel *im,
  int nr, nc,
  int th)
{
  for ( int r=0; r<nr; r++)
    for ( int c=0; c<nc; c++)
      im[r*nc+c] =
       im[r*nc+c] > th ? 1 : 0;
    }
  }
}
```

The same task described as a VHDL process

```
entity binarize is
  generic (
    thr : natural  := 0;
    im_width: positive;
    im_height: positive
    );
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
    idata : in   unsigned(7 downto 0);
    ival  : in  std_logic;
    odata : out unsigned(7 downto 0);
    oval  : out  std_logic
    );
end thr_op;

architecture rtl of binariz is
begin
  process ( clk, rst )
  begin
    if ( rst = '1' ) then ...
    elif ( clk'event and clk='1' ) then
      if ( ival = '1' ) then
        if ( idata >= to_unsigned(thr,8) )
        then
          odata <= to_unsigned(1, 8));
        else
          odata <= to_unsigned(0,8));
        end if;
      end if;
    else
        odata <= (others => 'Z');
    end if;
  end process;
  oval  <= ival;
end;
```


Programming h/w is not programming s/w ! (1/2)

- In software, everything is data
- In hardware data is represented by signals
- ... but data signals are not enough : control signals are also needed
- Control signals do not carry data but essentially say *when* a data is valid / to be taken into account
  - ex : `clk, reset, enable, ...`
- Using HDLs, it's the programmer's responsability to assert the control signals « at the right moment »
  - most of bugs in h/w designs do not come from data processing but from the incorrect generation of control signals
  - this is specially true for stream-processing apps

Programming h/w is not programming s/w ! (2/2)

- Software is, essentially, sequential (Von Neumann, imperative)

```
int f(...) {
  q = 0;
  r = q+1;
  }
```

- Hardware in intrinsically parallel (massively)
- As a result, time cannot be implicit in hardware descriptions

```
process(clk)
begin
  q <= '0';
  r <= q+1;
  ...
end;
```


HLS : Current solutions (for FPGAs)

- Several tools both from the commercial and academic domains
  - Xilinx VIVADO™
  - Intel/Altera HLS Compiler™
  - Cadence Stratus™ Compiler
  - Synopsis Synphony™ C Compiler
  - Maxeler MaxCompiler™
- Significant progress in the past few years
- But ...

HLS : still some problems ...

- Vendor tools are device/platform specific
- Performance of the generated code is generally far lower than with handcrafted HDL code
- The generated RTL-code is generally huge, complex and very difficult to understand for human programmers
- Ultimately requires knowledge on hardware programming...

.... which is precisely what we want to avoid !

Whether general-purpose HLS tools may ultimately offer a fully transparent and efficient compilation path is still a open (and debated !) question !

HLS : Mind the MoC !

- A suitable approach to solve this problem is to find a model which can be used both as a model of computation (to describe applications) and a model of execution (to implement them on the target hardware)
- Such a model exists : the dataflow model

HLS : the fondamental problem

- An old idea
  - Dennis, 1974
  - Arvind, 80's, MIT (Monsoon)
  - Sérot, 1993

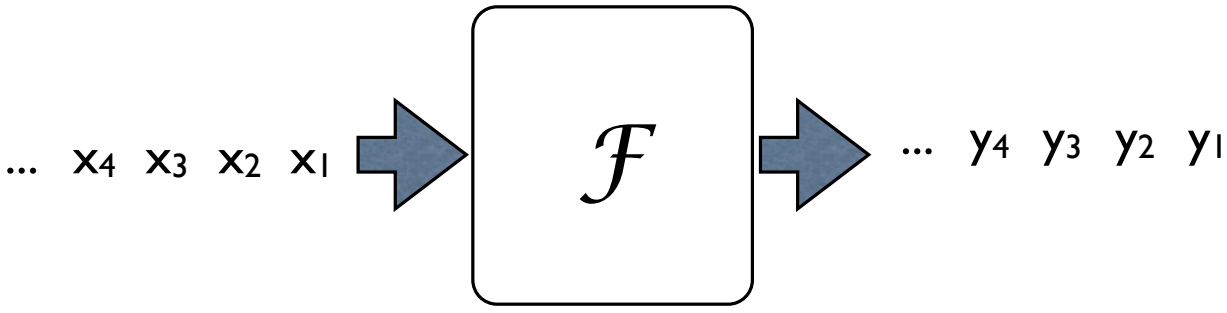
- Najjar-Lee-Gao, 1994 :

  « two characteristics of [the dataflow model] nicely fit the execution model of FPGAs : all data are values and all operations are purely functional [...] »

- Specially suited to stream-processing applications (processing data « on the fly »)
  - ex : signal and image processing

Dataflow model

Example 1



where : $y_i = F(x_i) = (x_i-1)(x_i+1)$

Dataflow model

Example 1

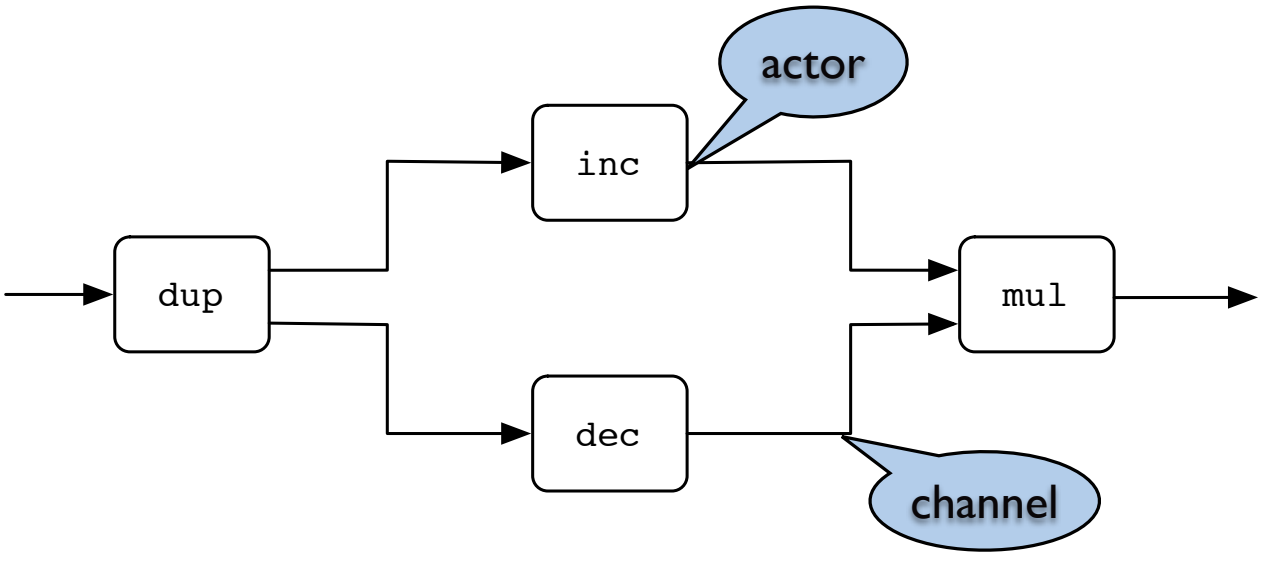
Imperative formulation :

```
while (1) {
   x=read();
   y=(x-1)*(x+1);
   write(y):
}
```


Dataflow model

Example 1

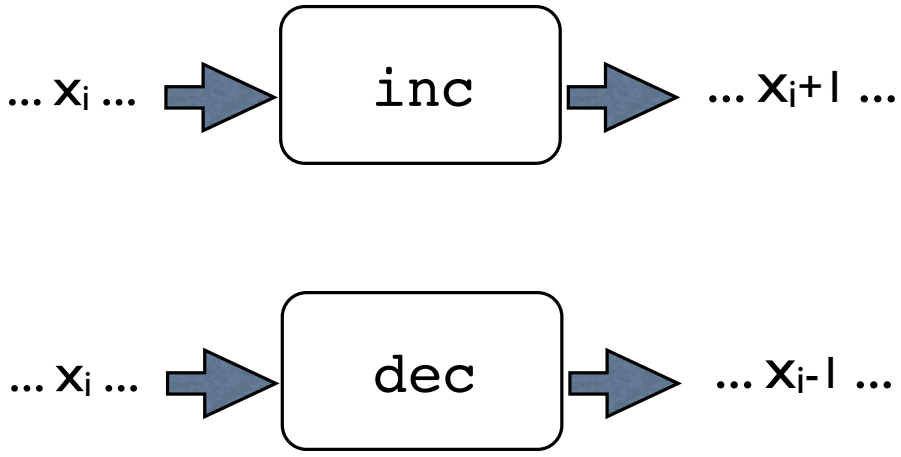
Dataflow formulation



Dataflow model

Example 1

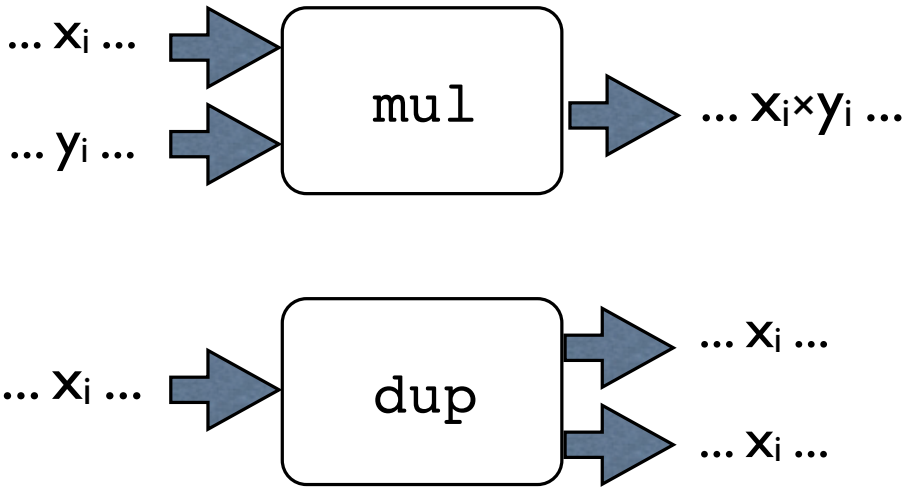
Dataflow formulation



Dataflow model

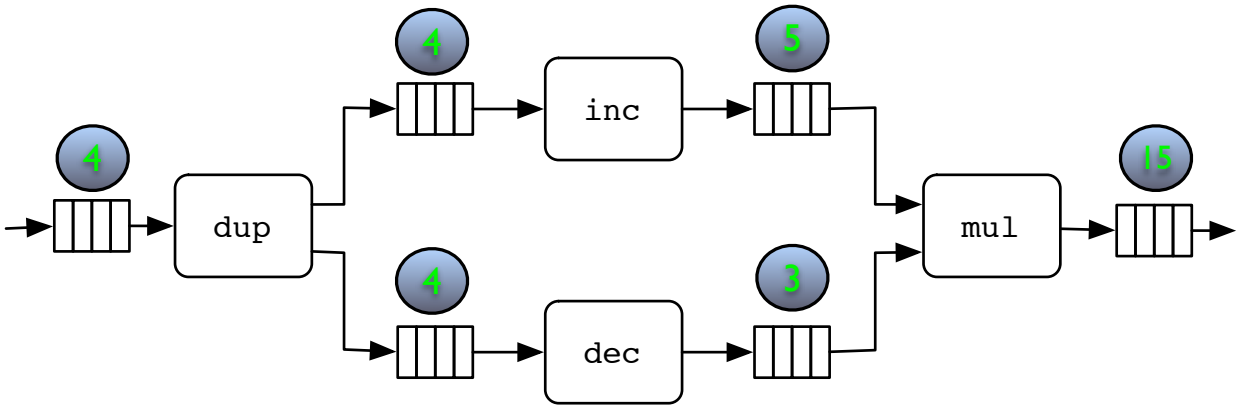
Example 1

Dataflow formulation



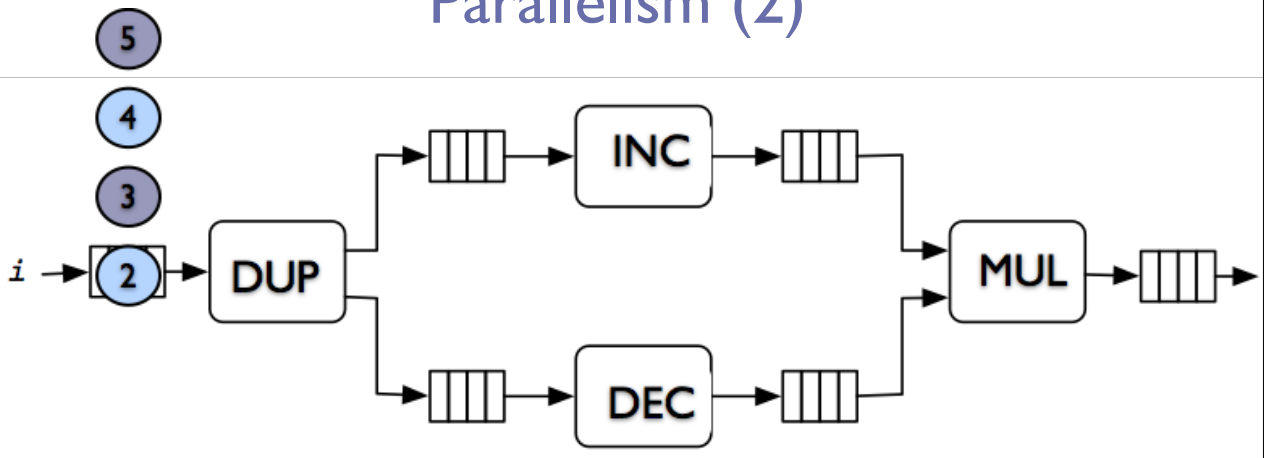
Dataflow model

Parallelism (1)



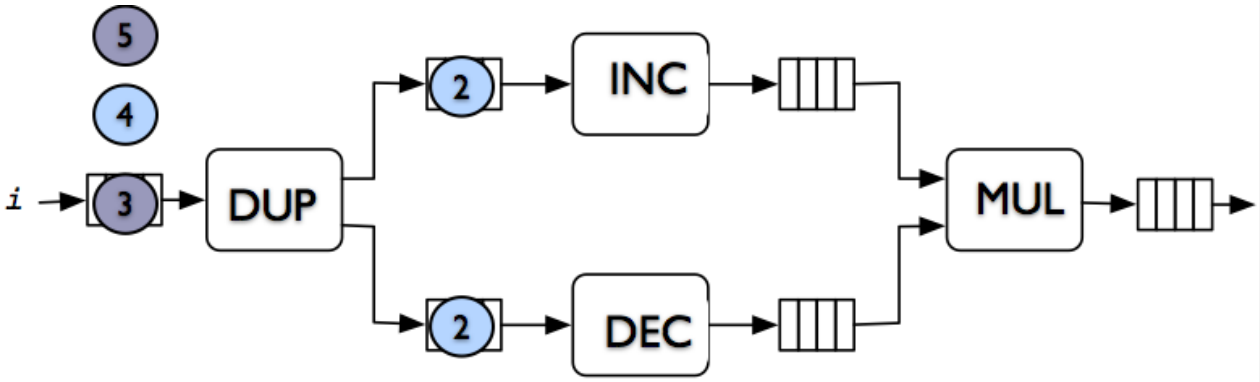
Dataflow model

Parallelism (2)



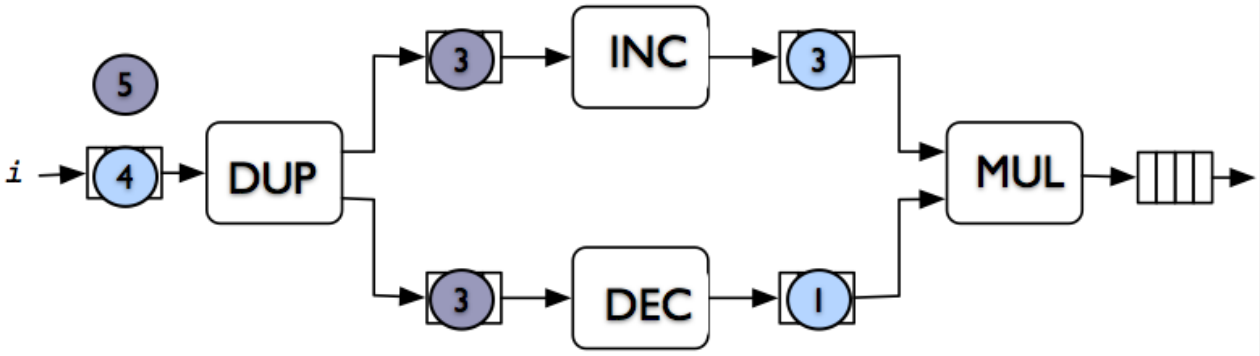
Dataflow model

Parallelism (2)



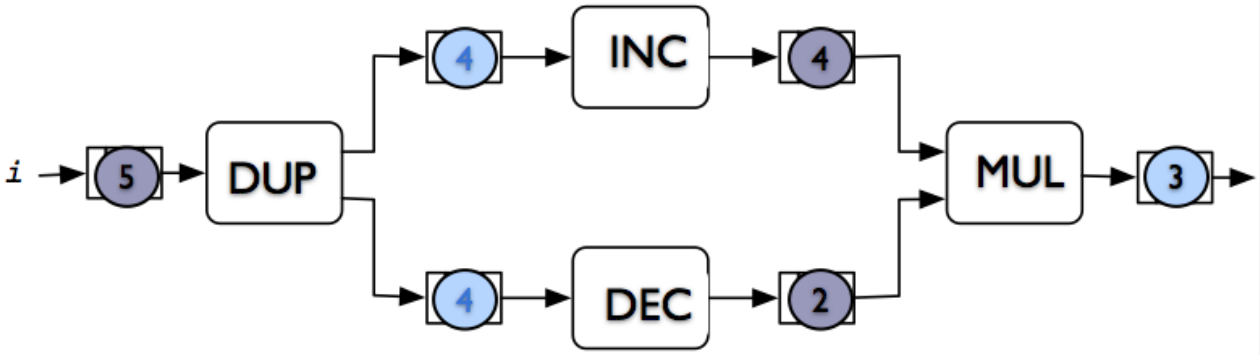
Dataflow model

Parallelism (2)



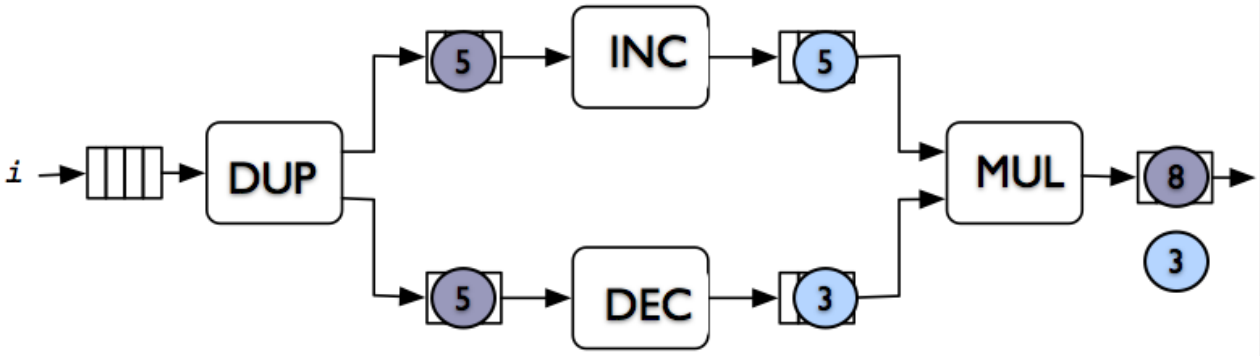
Dataflow model

Parallelism (2)



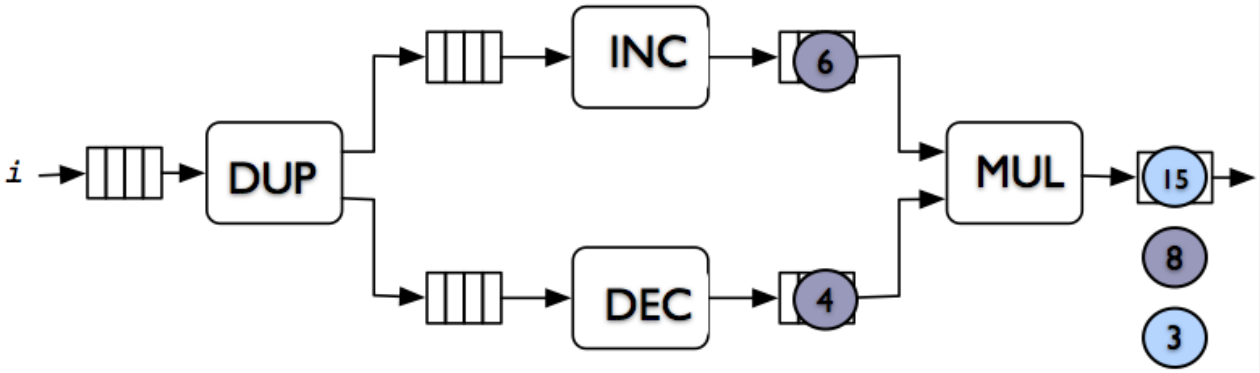
Dataflow model

Parallelism (2)



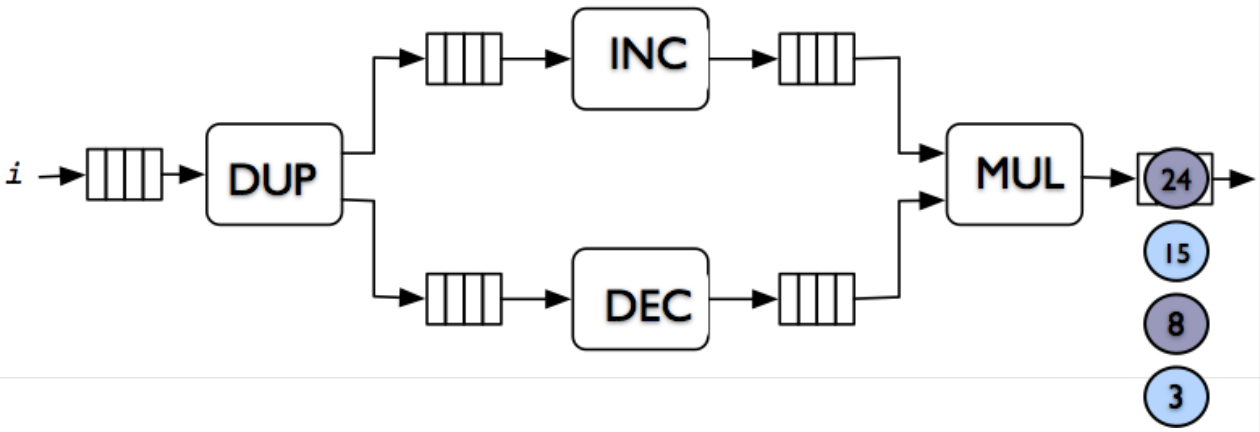
Dataflow model

Parallelism (2)



Dataflow model

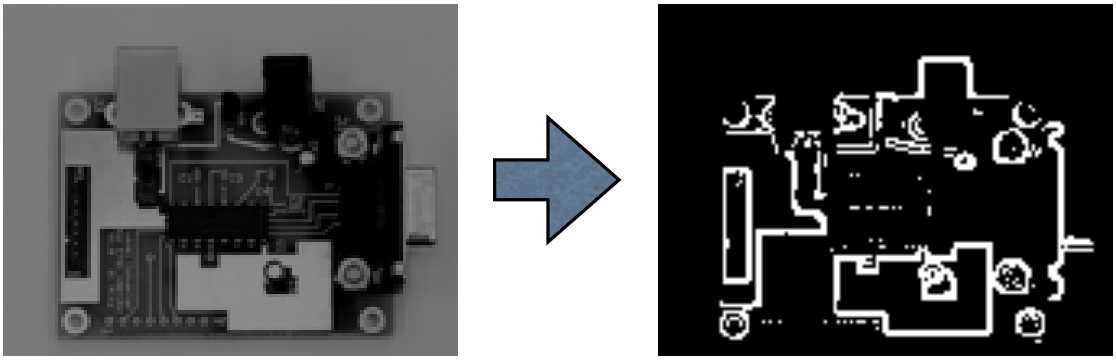
Parallelism (2)



Dataflow model

Example 2 : horizontal gradient computation

$$\forall i,\ j>0 : I'(i,j) = I(i,j)-I(i,j-1)$$
$$\forall i : I'(i,0) = I(i,j)$$



Dataflow model

Example 2 : horizontal gradient computation

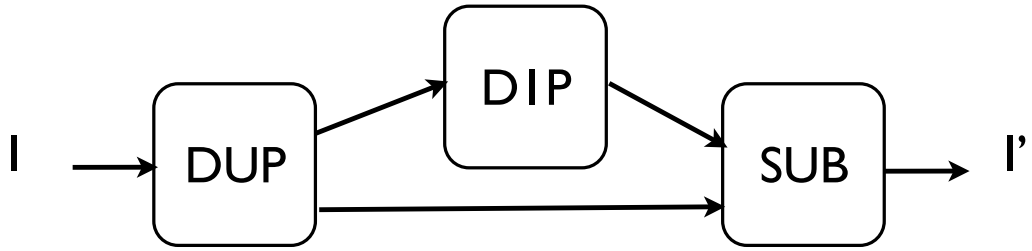
Imperative formulation

```
while (1) {
   for ( r=0; r<nr; r++ ) {
     pp = read_pixel();
     write_pixel(pp);
     for ( c=1; c<nc; c++ ) {
       p = read_pixel();
       p' = p-pp;
       write_pixel(p');
       pp = p';
       }
     }
  }
```

Dataflow model

Example 2 : horizontal gradient computation

Dataflow formulation

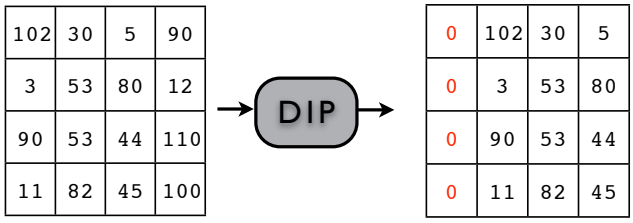



```
I' = I - D1P(I)
```


Dataflow model

Example 2 : horizontal gradient computation

Dataflow formulation

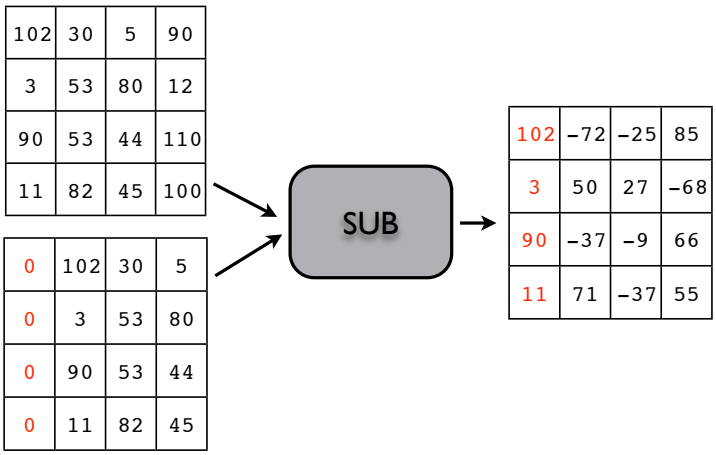



```
D1P: < < 102 30  5 90 > < 3 53 80 12 > < 90 53 44 110 > < 11 82 45 100 > >
 =   < <  0 102 30  5 > < 0  3 53 80 > <  0 90 53  44 > <  0 11 82  45 > >
```

Purely *functional* operator

Dataflow model

Example 2 : horizontal gradient computation
Dataflow formulation




```
  < < 102  30  5  90 > < 3 53 80  12 > < 90  53 44 110 > < 11 82  45 100 > >
- < <  0  102  30  5 > < 0  3 53  80 > <  0  90 53  44 > <  0 11  82  45 > >
= < < 102 -72 -25 85 > < 3 50 27 -68 > < 90 -37 -9  66 > < 11 71 -37  55 > >
```


Dataflow model

- What has been presented is actually of model of computation
- But it can also be viewed as model of execution on a FPGA :
  - channels = FIFOs → direct h/w implementation
  - actors = independent synchronous processes executing asynchronously → direct h/w implementation as FSMs
- No need for global synchronization ☺

  → control signals can be *embedded* as data tokens ☺
- Natural support of pipelining for on-the-fly processing ☺

From model to language

- Ok, but we just can program by drawing bubbles and arrow, can't we ?
- Need a formalism to describe / specify
  - the topology of the actor network
  - the behavior of individual actors
- Formalism means syntax + semantics
  - formal semantics allows sound compilation, proofs, ...
- Several dataflow languages.
  - Let's introduce CAPH

CAPH

- A domain specific programming language *(DSL)* for implementing stream-processing applications on FPGAs
- Developped since 2010 at Institut Pascal / UCA
- Initial goal : provide a tool for s/w programmers to implement applications on FPGA-based *smart cameras*
- Has gradually evolved towards a HLS system

CAPH

- Short introduction to the language concepts here
- More in the « hands on » tutorial on Friday

CAPH

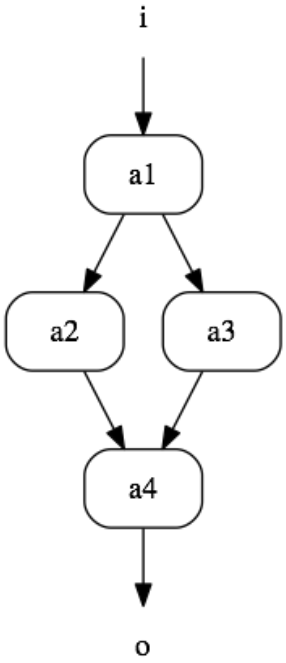
- Two main parts
  - network language
  - actor language

Caph Network Language

- Dataflow graphs may be described as expressions in a purely functional language
  - nodes correspond to function application and links to data dependencies
- Textual descriptions scale (much) better than graphical ones
- Consistency can be checked using type-checking
- Graph patterns can be encapsulated as higher-order functions

The CAPH Network Language

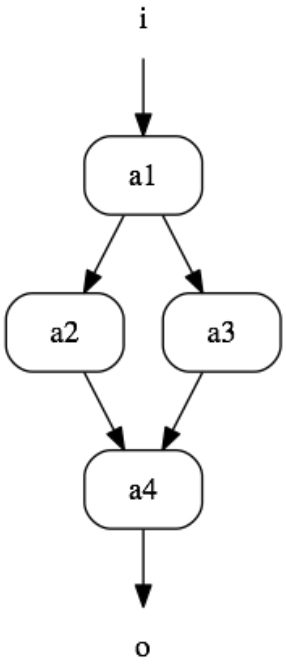
Exemple 1




```
actor a1
  in (i:int) out (o1:int, o2:int) …
actor a2
  in (i:int) out (o:int) …;
actor a3
  in (i:int) out (o:int) …;
actor a4
  in (i1:int, o2:int) out (o:int) …;


net (x,y) = a1 i
net x1 = a2 x
net y1 = a3 y
net o = a4 (x1, y1)
```

The CAPH Network Language

Exemple 2

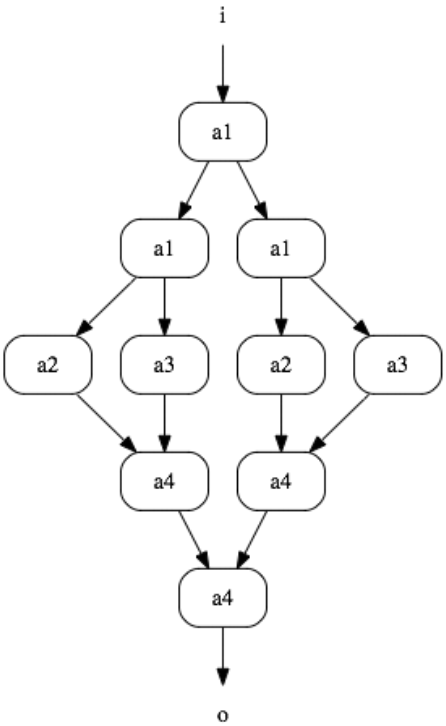



```
actor a1 …
actor a2 …
actor a3 …
actor a4 …


net diamond f1 f2 f3 f4 x =
  let (x2,x3) = f1 x in
  f4 (f2 x2, f3 x3);

net o = diamond a1 a2 a3 a4 i;
```

(higher-order) wiring function

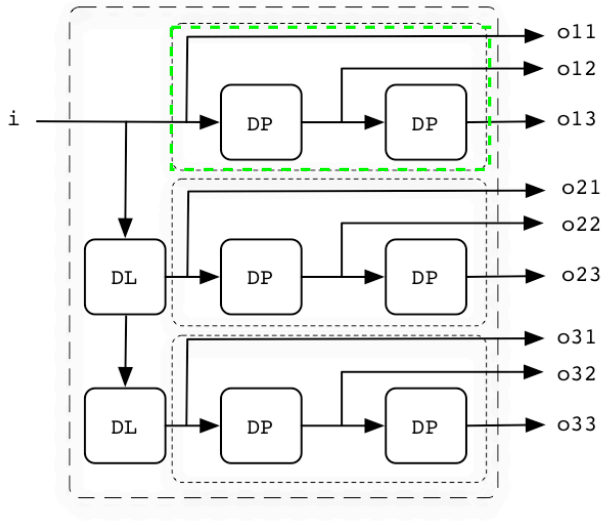
The CAPH Network Language

Exemple 3




```
actor a1 …
actor a2 …
actor a3 …
actor a4 …


net diamond f1 f2 f3 f4 x =
  let (x2,x3) = f1 x in
  f4 (f2 x2, f3 x3);

net f x = diamond a1 a2 a3 a4 x;
net o = diamond a1 f f a4 i;
```

Higher-order wiring functions



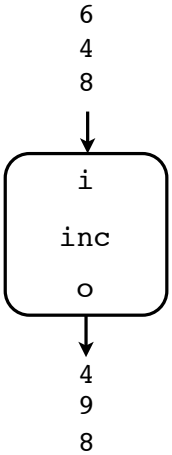

```
net neigh13(x) =
  x,
  dp x,
  dp (dp x);
```

```
net neigh33(x) =
  neigh13 x,
  neigh13 (dl x),
  neigh13 (dl (dl x));

net
 (o11,o12,o13),
 (o21,o22,o23),
 (o31,o32,o33))=
   neigh33(i);
```


The CAPH Actor Language

Exemple 1

- Behavior described a a set of transition rules
- Activation of rules based on pattern-matching



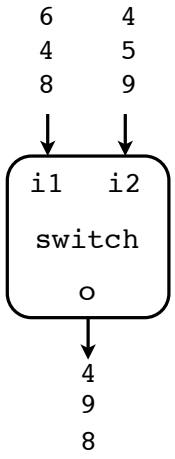

```
actor inc
  in (i : int)
  out (o : int)
  rules
  | i:x → o:x+1
```

I/Os

Transition rules

The CAPH Actor Language

Exemple 2



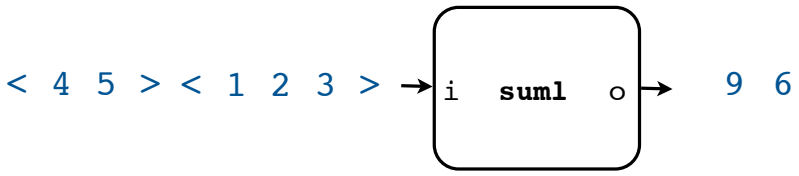

```
actor switch
 in (i1 : int,
     i2 : int)
 out (o : int)
var s : (Left,Right) = Left
rules
| s:Left,  i1:v → o:v, s:Right
| s:Right, i2:v → o:v, s:Left
```

Local var

Transition rules

The CAPH Actor Language

Exemple 3




```
type 'a dc =
  EoS
| SoS
| Data of 'a
```

```
actor suml
  in (a: int dc)
 out (c: int)
 var st: {S0,S1}=S0
 var s : int
 rules
   st:S0, a:SoS → st:S1, s:0
 | st:S1, a:Data v → st:S1, s:s+v
 | st:S1, a:EoS → st:S0, c:s
```

The CAPH Actor Language

Exemple 4

102	30	5	90
3	53	80	12
90	53	44	110
11	82	45	100

→ DIP →

0	102	30	5
0	3	53	80
0	90	53	44
0	11	82	45

```
actor d1p
  in (i:signed<8> dc)
  out (c:signed<8> dc)
var s : {S0,S1,S2} = S0
var z : signed<8>
rules
| s:S0, i:SoS ➔ s:S1, c:SoS          -- Start of Frame
| s:S1, i:EoS ➔ s:S0, c:EoS          -- End of Frame
| s:S1, i:EoS ➔ s:S2, c:EoS, z:0     -- Start of Line
| s:S2, i:Data v ➔ s:S2, c:Data z, z:v   -- Pixel
| s:S2, i:SoS ➔ s:S1, c:EoS          -- End of Line
```


A sample *Caph* program

```
function f_abs x =
  if x < 0 then 0-x else x
  : signed<8> -> signed<8>;

constant threshold = 40;
```

Global values

```
actor d1p () ...

actor d1l () ...

actor add () ...

actor asub () ...

actor thr (t:signed<8>) ...
```

Actors

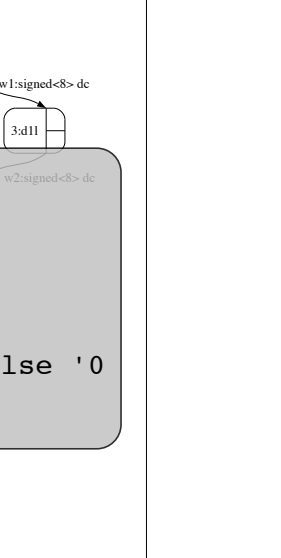
```
actor thr (k:unsigned<8>)
  in (a:unsigned<8> dc)
  out (c:unsigned<1> dc)
rules a -> c
| '< -> '<
| 'p -> if p > k then '1 else '0
| '> -> '>
;
```

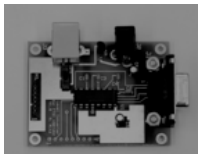
I/Os

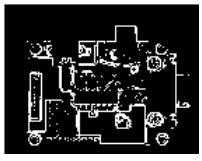
```
stream i:signed<8> dc from "dev:cam0";
stream o:signed<8> dc to "dev:mon1";
```

```
net g = add (asub(i, d1p i), asub(i, d1l i));
net o = thr [threshold] g;
```

Network description







CAPH Language extra features

- Caph is strongly inspired by functional programming languages (ML, Haskell, ...)
- Several powerful concepts offered by these languages supported
  - pattern-matching
  - user-defined algebraic data types
  - higher-order functions
  - parametric polymorphism
  - dependant types
- Caph is the only HLS language offering this level of abstraction
  - this contributes both to expressivity and safety

Parametric polymorphism

```
actor mux
  in (e1:signed<8>,
      e2: signed<8>,
      c:bool)
 out (s: signed<8>)
rules
| e1:x, c:true -> s:x
| e2:x, c:false -> s:x
;
```

```
actor mux
  in (e1:unsigned<4>,
      e2:unsigned<4>,
      c:bool)
 out (s:unsigned<4>)
rules
| e1:x, c:true -> s:x
| e2:x, c:false -> s:x
;
```

```
...
```

Parametric polymorphism

```
actor mux
  in (e1:$t,
      e2: $t,
      c:bool)
 out (s: $t)
rules
| e1:x, c:true -> s:x
| e2:x, c:false -> s:x
;
```

Texte

Implementation :
- SystemC templates
- VHDL replicated code

User-defined algebraic data

```
type $t option =
  Some of $t
| None

actor foo
   in (a: bool option)
  out (c: bool)
rules
| a:Some x -> c:not x
| a:None   -> c:_
```

```
type $t dc =
  EoS          -- aka "'<"
| SoS          -- aka "'>"
| Data of $t   -- aka "'v"
```

Higher-order actors

```
actor inc
  in (e:signed<8>)
 out (s: signed<8>)
rules
| x -> x+1
;

net o = inc i
```

```
actor double
  in (e:signed<8>)
 out (s: signed<8>)
rules
| x -> x*2
;

net o = double i
```

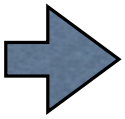
```
...
```

Higher-order actors

```
function inc x = x+1 : signed<8> -> signed<8>;
function double x = x+2 : signed<8> -> signed<8>;

actor map (f: signed<8> -> signed<8>)
  in (e:signed<8>)
 out (s: signed<8>)
rules
| x -> f(x)
;

net o1 = map inc i;
net o2 = map double i;
```

Dependant types

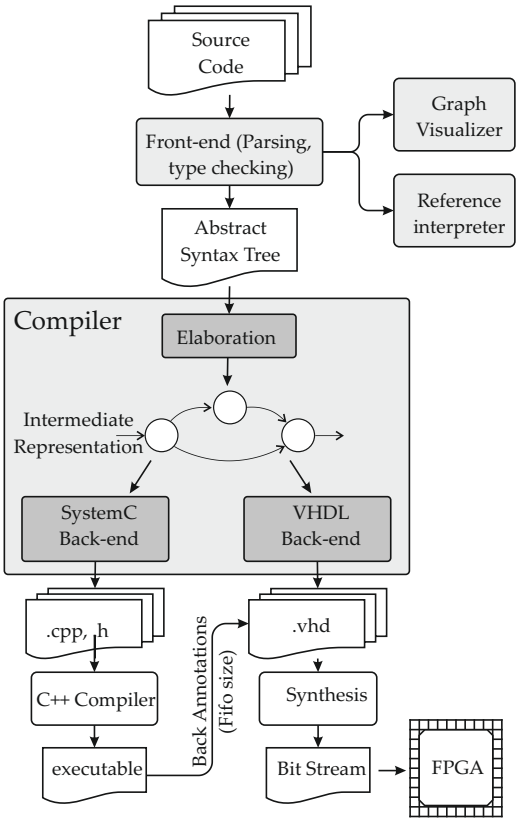
```
actor add
  in (e1:unsigned<8>,
      e2:unsigned<8>)
 out (s: unsigned<9>)
rules
| e1:x, e2:y -> s:x+y
;
```



```
actor add
  in (e1:unsigned<n>,
      e2:unsigned<n>)
 out (s: unsigned<n+1>)
rules
| e1:x, e2:y -> s:x+y
;
```


The *Caph* toolset

- Graph visualizer : `.dot` format
- Reference interpreter :
  - based on the fully formalized semantics
  - tracing, profiling and debugging
- Compiler :
  - elaboration of a target-independant IR
  - specialized backends (SystemC,VHDL)

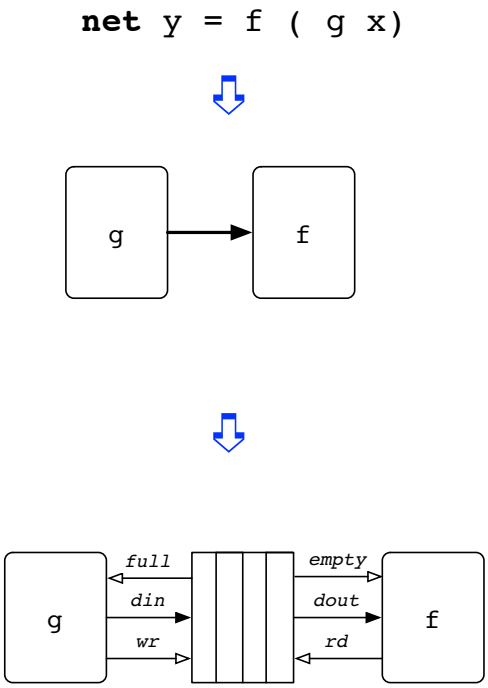


Elaboration

- Three steps
  1. network generation
  2. RT-level description of actors
  3. SystemC/VHDL transcription
- Only a quick overview here (see papers & LRM)

1. Network generation

- Using *abstract interpretation* of the network description, viewed as a *functional program*
- Each application of a function bound to an actor inserts an instance of this actor into the graph
- Each functional dependency inserts a *channel*
  - channels are then instanciated as FIFOs

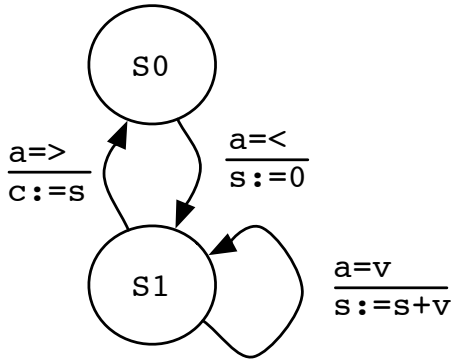


II. Translation of actor to RTL

Set of transition rules → FSM + operations (FSMD)

Example : **suml : < 1 2 3 > = 6**

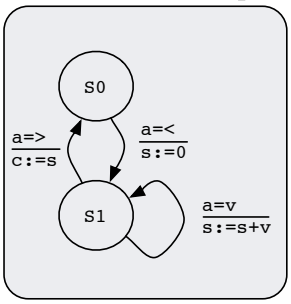
```
actor suml ()
  in (a: int dc)
 out (c: int)
 var st: {S0,S1}=S0
 var s : int
 rules
   st:S0, a:'< ➔ st:S1, s:0
 | st:S1, a:'v ➔ st:S1, s:s+v
 | st:S1, a:'> ➔ st:S0, c:s
```



III. VHDL Transcription

Example

```
actor suml ()
  in (a: int dc)
 out (c: int)
 var st: {S0,S1}=S0
 var s : int
 rules st,a,s-> st,c,s
   S0,'<,_ ➔ S1,_,0
 | S1,'v,s ➔ S1,_,s+v
 | S1,'>,s ➔ S0,s,_
```



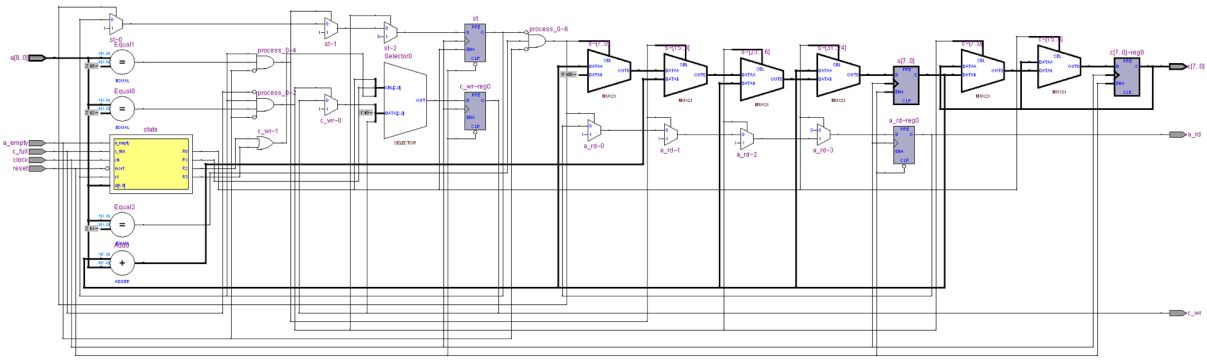

```
entity sum_act is
   port (
    a_empty: in std_logic;
    a: in std_logic_vector(9 downto 0);
    a_rd: out std_logic;
    c_full: in std_logic;
    c: out std_logic_vector(15 downto 0);
    c_wr: out std_logic;
    clock: in std_logic;
    reset: in std_logic
    );
end sum_act;

architecture FSM of sum_act is
    type t_state is (S0,S01,S1,S11,S12);
begin
  process(clock, reset)
    variable s : std_logic_vector(15 downto 0);
    variable st : t_state;
    variable v : std_logic_vector(7 downto 0);
```

```
...
  begin
    if (reset='0') then
      st := S0; a_rd <= '0'; c_wr <= '0';
    elsif rising_edge(clock) then
      case state is
        when S0 =>
          if a_empty='0' and is_sos(a) then
            a_rd <= '1';
            st := S01;
            s := conv_std_logic_vector(0,15);
            end if;
        when S01 =>
            a_rd <= '0'; state <= S1;
        when S1 =>
          if a_empty='0' and is_data(a) then
            a_rd <= '1'; v := data_from(a);
            s := s+v; st := S11;
            end if;
          if a_empty='0' and is_eos(a) then
            a_rd <= '1'; c := s;
            c_wr <= '1'; st := S12;
            end if;
        when S11 =>
            a_rd <= '0'; st := S1;
        when S12 =>
            a_rd <= '0'; c_wr <= '0'; st := S0;
      end case;
    end if;
  end process;
end FSM;
```

III. Synthesis (RT-level → Gate level)

Using vendor-specific tool-chains

Example :
`suml` actor synthetized on a Stratix IV using Altera Quartus 9

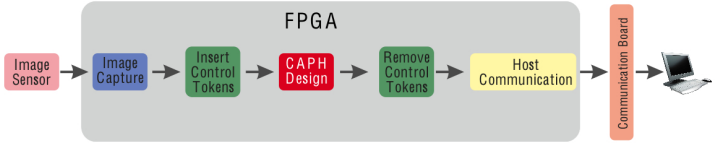


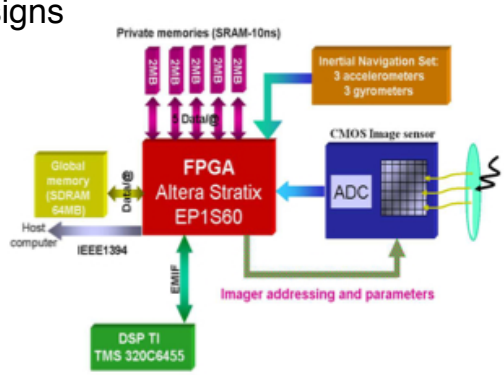
Testbench example

DreamCam Smart camera

•Modular hardware architecture

- CMOS imager
- Inertial device
- Altera Stratix FPGA
- 5 external SRAM memory blocks
- USB interface for host communication
- Dedicated framework for interfacing CAPH designs

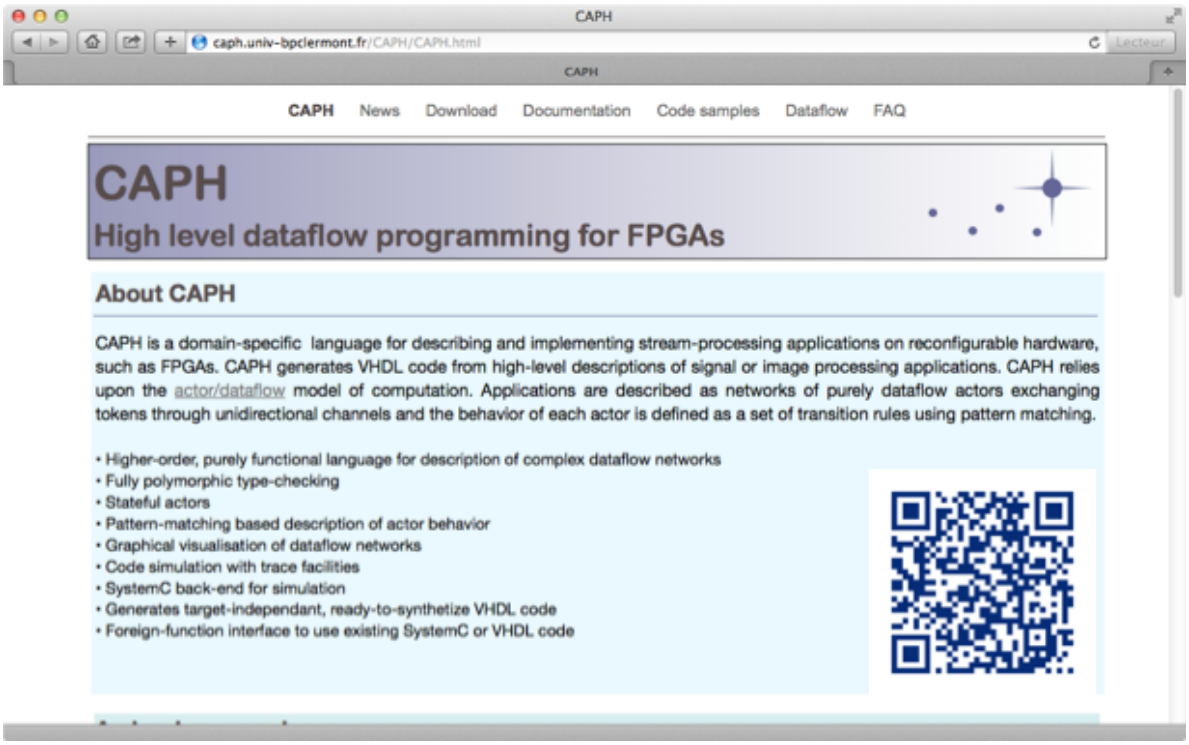




Example applications

- Harris-Stephen detector
- Real-time tracking of moving objects
- Connected component labeling
- HOG-based pedestrian and vehicle detection
- H264 encoder
- ...

HAVE A TRY ?



... or see you Friday ;)