# *Software code generation for the RVC-CAL dataflow programming language*

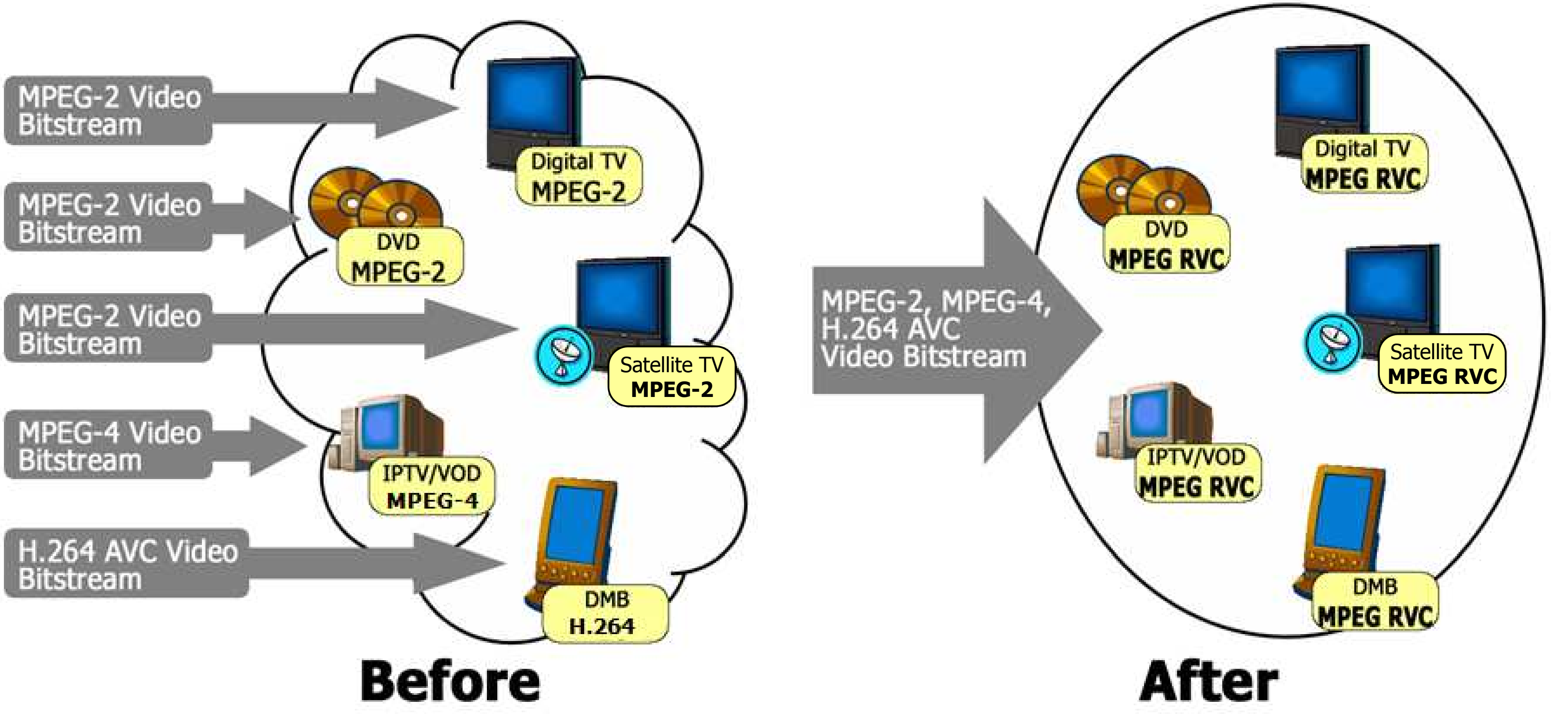
Mickaël Raulet
IETR/INSA of Rennes
France
Friday, April 3rd 2009


UMR 6164

# Outline

- Introduction
- Dataflow programs
  - Description: CAL, NL, and others
  - Generic translation process
- Scheduling inside an actor
  - Simulation
  - Implementation (HW and SW)
- Scheduling actors
- Conclusion/perspectives

- RVC framework

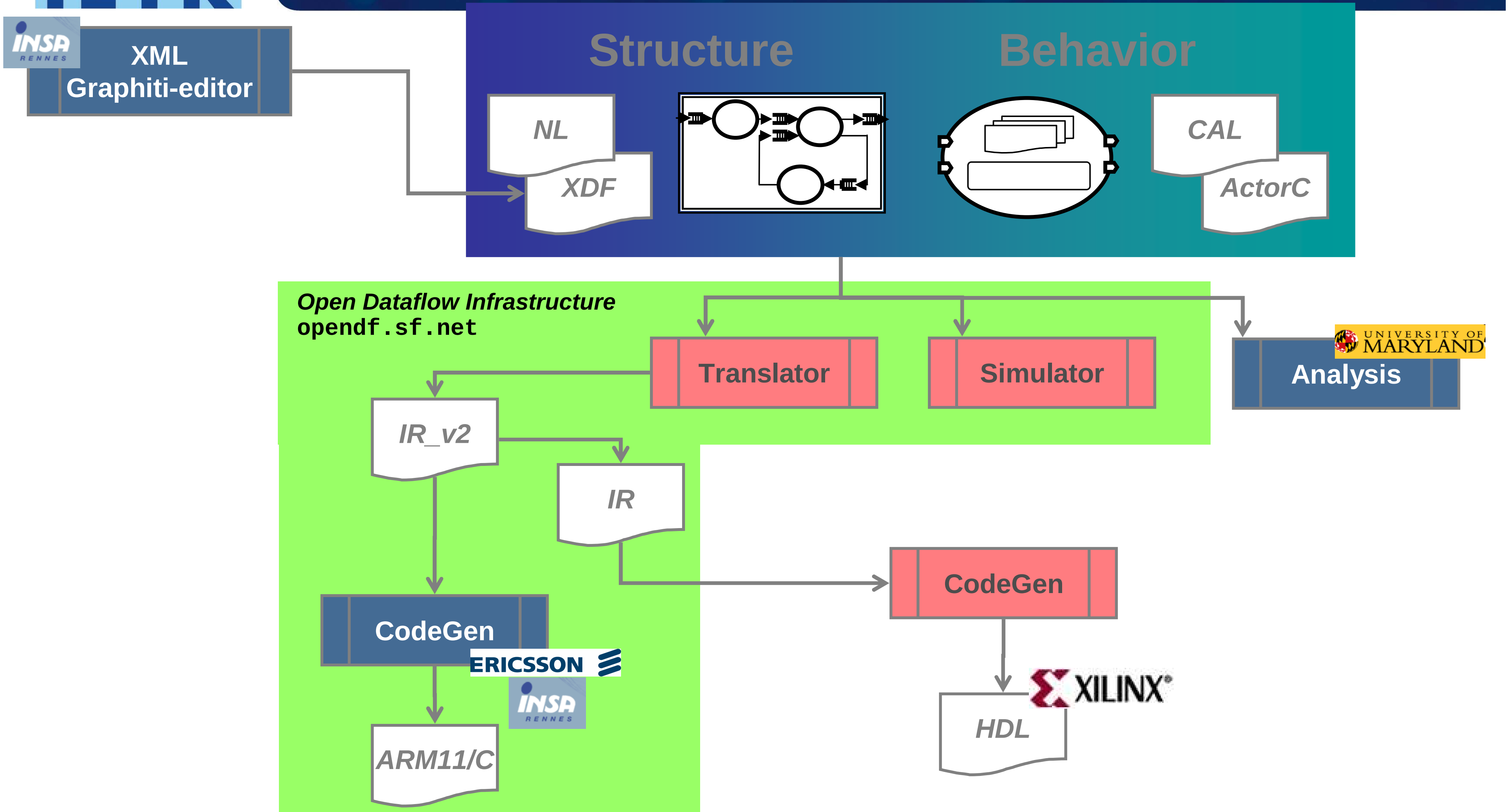



Courtesy of Sunyoung Lee

# Introduction – origins

- Origins of Cal2C
  - Reconfigurable Video Coding framework
  - Software code generator for CAL actors
- Current status
  - Stable version delivered for the 84th MPEG meeting
  - Development version (still experimental) RVC-CAL compliant
- Aims of this presentation
  - Unravel the mysteries of Cal2C
  - Study the integration of SSR to C translation

# Dataflow tools – Current situation






UMR 6164

# Outline

- Introduction
- **Dataflow programs**
  - Description: CAL, NL, and others
  - Generic translation process
- Scheduling inside an actor
  - Simulation
  - Implementation (HW and SW)
- Scheduling actors
- Conclusion/perspectives

# Dataflow program – CAL language

```
actor ID () In ==> Out :

    action In: [a] ==> Out: [a] end
end
```

```
actor ID () In ==> Out :

    action [a] ==> [a] end
end
```

```
actor Add () Input1, Input2 ==> Output:

    action [a], [b] ==> [a + b] end
end
```

```
actor AddSeq () Input ==> Output:

    action [a, b] ==> [a + b] end
end
```

# Dataflow program – RVC-CAL language

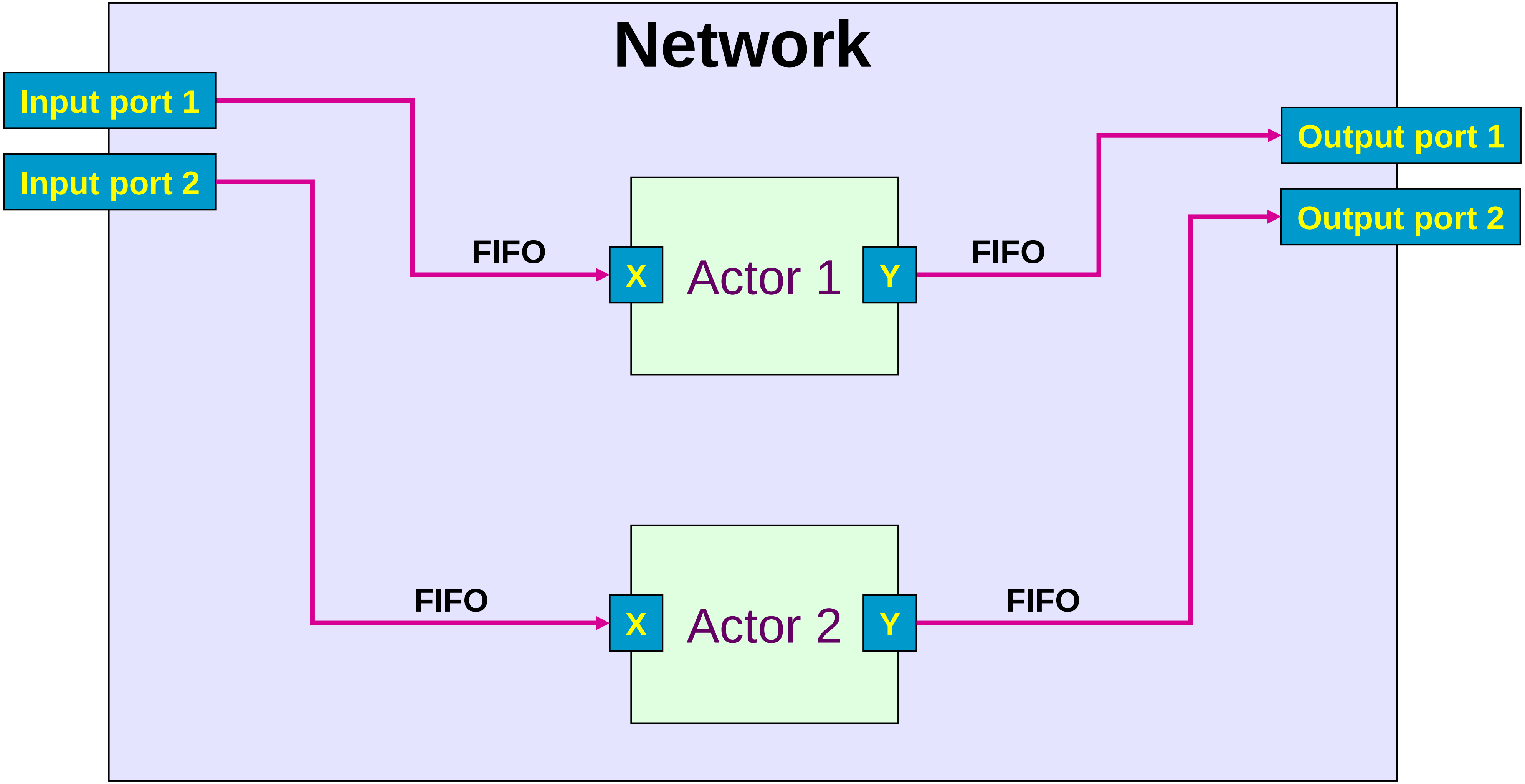
```
actor ID () int In ==> int Out :

    action In: [a] ==> Out: [a] end
end
```

```
actor ID () int In ==> int Out :

    action [a] ==> [a] end
end
```

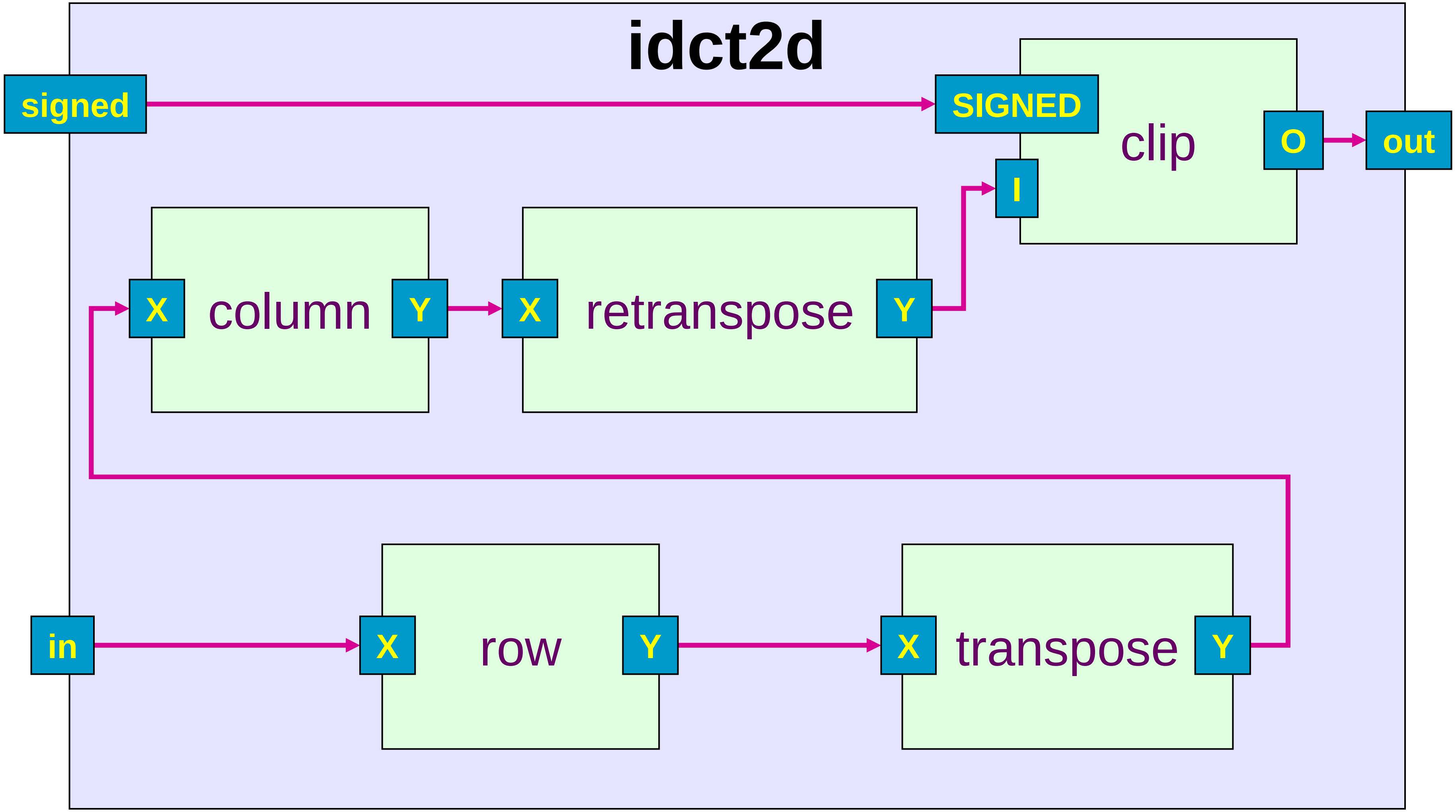
```
actor Add () int Input1, int Input2 ==> int Output:

    action [a], [b] ==> [a + b] end
end
```

```
actor AddSeq () int Input ==> int Output:

    action [a, b] ==> [a + b] end
end
```

# Dataflow program – description



# Dataflow program – Sample



# Dataflow program – NL description

```
network idct2d () in, signed ==> out :

entities

      row = GEN_124_algo_Idct1d(ROW = true);
      transpose = GEN_algo_Transpose( );
      column = GEN_124_algo_Idct1d(ROW = false);
      retranspose = GEN_algo_Transpose( );
      clip = GEN_algo_Clip();

structure
      in --> row.X;
      signed --> clip.SIGNED;

      row.Y --> transpose.X;
      transpose.Y --> column.X;
      column.Y --> retranspose.X;
      retranspose.Y --> clip.I;

      clip.O --> out;
end
```

# Dataflow program – And what about XDF/NL?

- NL is (also) the textual representation
  - With Connection, Instance…
- XDF is equivalent to NL but is XML
  - Describe the structural view of the network
  - Compliant to FNL in ISO/IEC 23001-4
- XDF is an (also) elaboration of an NL network
  - NL computes a graph from a set of parameters
  - XDF contains the whole computed graph – the elaborated one

# Dataflow program – CAL language

```
// Limit pixel value to either [0,255] or [-255,255]

actor GEN_algo_Clip (int isz, int osz) int(size=isz) I, bool SIGNED ==> int(size=osz) O :

  int(size=7) count := -1;
  bool sflag := false;

  read_signed: action SIGNED:[s] ==> guard count < 0 do sflag := s; count := 63; end

  limit.max: action I:[i] ==> O:[ 255 ] guard i > 255 do count := count - 1; end

  limit.zero: action I:[i] ==> O:[ 0 ] guard not sflag, i < 0 do count := count - 1; end

  limit.min: action I:[i] ==> O:[ -255 ] guard i < -255 do count := count - 1; end

  limit.none: action I:[i] ==> O:[ i ] do count := count - 1; end

  priority
    read_signed > limit;
    limit.max > limit.zero > limit.min > limit.none;
  end
end
```

# Dataflow program – Action scheduler

```
// Limit pixel value to either [0,255] or [-255,255]
```

I = -173 SIGNED = true O = -173

```
actor GEN_algo_Clip (int isz, int osz) int(size=isz) I, bool SIGNED ==> int(size=osz) O :

  int(size=7) count := 63;
  bool sflag := true;

  read_signed: action SIGNED:[s] ==> guard count < 0 do sflag := s; count := 63; end

  limit.max: action I:[i] ==> O:[ 255 ] guard i > 255 do count := count - 1; end

  limit.zero: action I:[i] ==> O:[ 0 ] guard not sflag, i < 0 do count := count - 1; end

  limit.min: action I:[i] ==> O:[ -255 ] guard i < -255 do count := count - 1; end

  limit.none: action I:[i] ==> O:[ i ] do count := count - 1; end

  priority
    read_signed > limit;
    limit.max > limit.zero > limit.min > limit.none;
  end
end
```

Legend:

- guard = true
- guard = false
- token
- action enabled

1. Eligible actions
2. Activated actions
3. Firable actions



UMR 6164

# Outline

- Introduction
- Dataflow programs
  - – Description: CAL, NL, and others
  - – Generic translation process
- **Scheduling inside an actor**
  - – Simulation
  - – Implementation (HW and SW)
- Scheduling actors
- Conclusion/perspectives

# Synthesis – Translation of a dataflow program

OpenDF

1. Parse CAL and NL into CALML
2. Elaborate the network
   - Flatten hierarchy, instantiate parameters
3. Transform CAL code
   1. Rename local variables (prevent name clashes)
   2. Transform operators with respect to priority
      - (from “a + b * c” to “$add(a, $mul(b, c))”)
   3. Sort variables by dependency
   4. Order actions by priorities
   5. Propagate constants
   6. …

Cal2C, Cal2HDL

4. Translate CAL code to the target language

# Synthesis – from CAL TO(2) C

## 1. Parser

- Produces an Abstract
- Syntax Tree

## 2. Elaboration

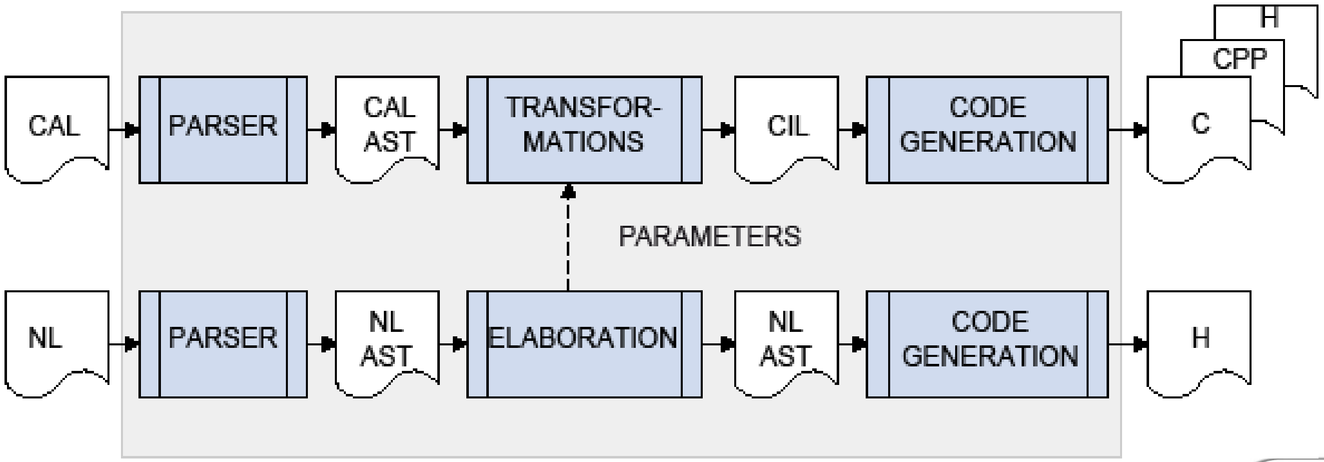
Makes all actors closed

## 3. Transformations on AST

- CAL → CAL
- CAL → CIL

## 4. Code Generation

C Intermediate Language
Pretty-printer Library
CIL → ANSI C



# CAL2?? – SSA form (static single assignment)

```
wait A3_GO do
  v <- IN;
  if count_IN < 63 then
    count_OUT := count_IN + 1;
  else
    count_OUT := 0;
  end
end A3_DONE;
```

➡

```
wait A3_GO do
  v <- IN;
  if count_IN < 63 then
    $1 := count_IN + 1;
  else
    $2 := 0;
  end
  count_OUT := PHI($1, $2);
end A3_DONE;
```

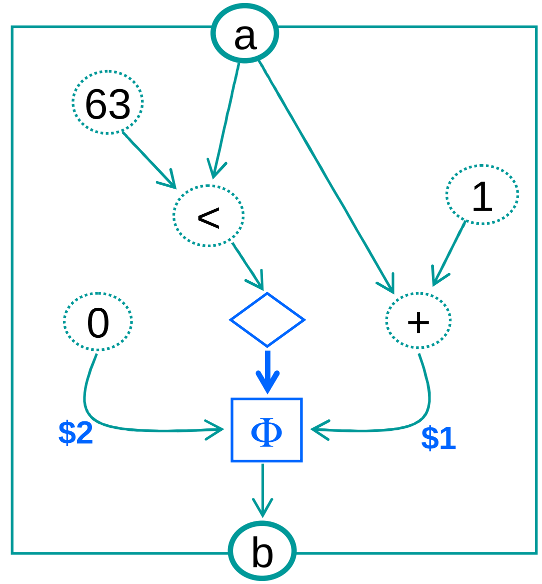
```
n := 0;
while P(n) do
  n := n + 1;
  S1(n);
end
S2(n);
```

➡

```
    $1 := 0;
L1: $2 := PHI($1, $3);
    if not P($2) then goto L2;
    $3 := $2 + 1;
    goto L1;
L2: S2($2);
    n := $2;
```

# CAL2?? – SSA form (static single assignment)

```
if a < 63 then
  $1 := a + 1;
else
  $2 := 0;
end
b := PHI($1, $2);
```




## SSA representation

- straightforward extraction of parallelism
- local scalar variables become arcs
  - = wires in hardware implementation
- good starting point for hardware and software backends

# Scheduling – action scheduler hardware implementation

- An actor is transformed to **several** threads
  - 1 thread called action scheduler
- Each action is transformed to a thread
  - Executing an action ⇔ switch to a particular thread

# Scheduling – generating threads (from Jörn Janneck)

```
actor SendDC (int T_INTER) int TYPE, int IN ==> int DC :

  int count := 0;

  action TYPE: [t], IN: [v] ==> DC: [v]
  guard count = 0, t = 1
  do
    count := count + 1;
  end

  action TYPE: [t], IN: [v] ==>
  guard count = 0, t != 1
  do
    count := count + 1;
  end

  action IN: [v] ==>
  guard count > 0
  do
    if count < 63 then
      count := count + 1;
    else
      count := 0;
    end
  end
end
```

action scheduler

action thread 1

action thread 2

action thread 3

# Scheduling – generating threads (from Jörn Janneck)

```
actor SendDC (int T_INTER) int TYPE, int IN ==>

  int count := 0;

  action TYPE: [t], IN: [v] ==> DC: [v]
  guard count = 0, t = T_INTER
  do
    count := count + 1;
  end

  action TYPE: [t], IN: [v] ==>
  guard count = 0, t != T_INTER
  do
    count := count + 1;
  end

  action IN: [v] ==>
  guard count > 0
  do
    if count < 63 then
      count := count + 1;
    else
      count := 0;
    end
  end
end
```

```
wait A1_GO do
  v <- IN;
  t <- TYPE;
  count_OUT := count_IN + 1;
  v -> DC;
end A1_DONE;
```

```
wait A2_GO do
  v <- IN;
  t <- TYPE;
  count_IN + 1 -> count_OUT;
end A2_DONE;
```

```
wait A3_GO do
  v <- IN;
  if count_IN < 63 then
    count_OUT := count_IN + 1;
  else
    count_OUT := 0;
  end
end A3_DONE;
```

# Scheduling – generating threads (from Jörn Janneck)

```
forever
var
  t = peek(TYPE, 0);
  c1 = TYPE_#1 && IN_#1 && count_IN = 0 && t = 1;
  c2 = TYPE_#1 && IN_#1 && count_IN = 0 && t != 1;
  c3 = IN_#1 && count_IN > 0 &&  count_IN > 0;
do
  parcase
    c1: set A1_GO; wait A1_DONE; unset A1_GO;
    c2: set A2_GO; wait A2_DONE; unset A2_GO;
    c3: set A3_GO; wait A3_DONE; unset A3_GO;
  end
end
```

```
actor SendDC (int T_INTER) int TYPE, int IN ==> int DC :

  int count := 0;

  action TYPE: [t], IN: [v] ==> DC: [v]
                t = T_INTER

                + 1;

                IN: [v] ==>
                t != T_INTER

                + 1;

                =>

  do
    if count < 63 then
      count := count + 1;
    else
      count := 0;
    end
  end
end
```

# Scheduling – action scheduler software implementation

- An actor is transformed to **one** thread
  - The action scheduler
- Each action is transformed to a C function
  - Executing an action ⇔ calling a function

# Scheduling – CAL Sample

```
// Limit pixel value to either [0,255] or [-255,255]

actor GEN_algo_Clip (int isz, int osz) int(size=isz) I, bool SIGNED ==> int(size=osz) O :

  int(size=7) count := -1;
  bool sflag := false;

  read_signed: action SIGNED:[s] ==> guard count < 0 do sflag := s; count := 63; end

  limit.max: action I:[i] ==> O:[ 255 ] guard i > 255 do count := count - 1; end

  limit.zero: action I:[i] ==> O:[ 0 ] guard not sflag, i < 0 do count := count - 1; end

  limit.min: action I:[i] ==> O:[ -255 ] guard i < -255 do count := count - 1; end

  limit.none: action I:[i] ==> O:[ i ] do count := count - 1; end

  priority
    read_signed > limit;
    limit.max > limit.zero > limit.min > limit.none;
  end
end
```

# Scheduling – C Sample translation

SIGNED = true, I = -173

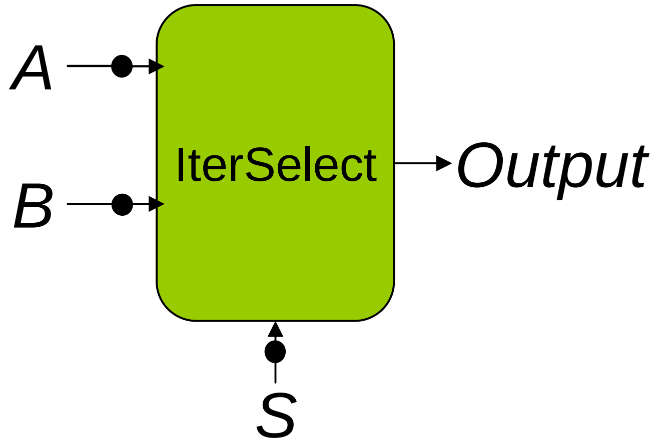
```
_actor_variables->count = 64;
_actor_variables->sflag = 0;
while (1) {
  if (_actor_variables->count < 0) {
    _call_11 = SIGNED->get();
    GEN_algo_Clip_read_signed(_actor_variables, _call_11);
  } else if (I->peek() > 255) {
    _call_12 = I->get();
    GEN_algo_Clip_limit_dot_max(_actor_variables, _call_12, & _out_4);
    O->put(_out_4);
  } else if (! _actor_variables->sflag && I->peek() < 0) {
    _call_13 = I->get();
    GEN_algo_Clip_limit_dot_zero(_actor_variables, _call_13, & _out_3);
    O->put(_out_3);
  } else if (I->peek() < - 255) {
    _call_14 = I->get();
    GEN_algo_Clip_limit_dot_min(_actor_variables, _call_14, & _out_2);
    O->put(_out_2);
  } else {
    _call_15 = I->get();
    GEN_algo_Clip_limit_dot_none(_actor_variables, _call_15, & _out_1);
    O->put(_out_1);
  }
}
```

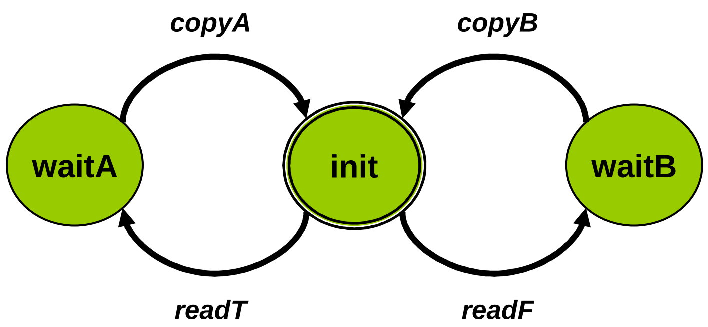


UMR 6164

- Mux with an FSM :

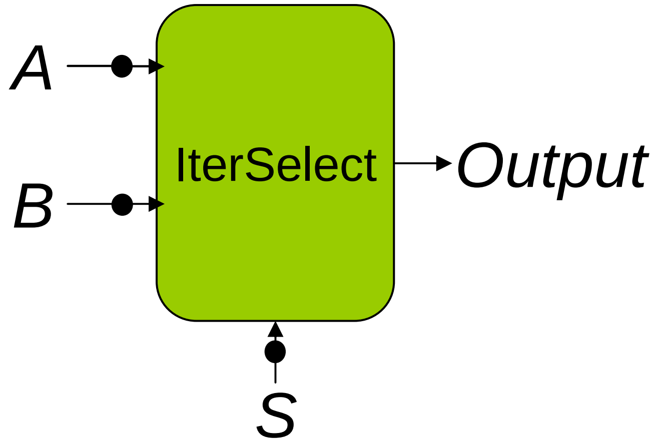
```
actor IterSelect () S, A, B ==> Output:
  readT: action S: [s] ==>
    guard s end

  readF: action S: [s] ==>
    guard not s end

  copyA: action A: [v] ==> [v] end
  copyB: action B: [v] ==> [v] end

  schedule fsm init:
    init (readT) --> waitA;
    init (readF) --> waitB;
    waitA (copyA) --> init;
    waitB (copyB) --> init;
  end
end
```

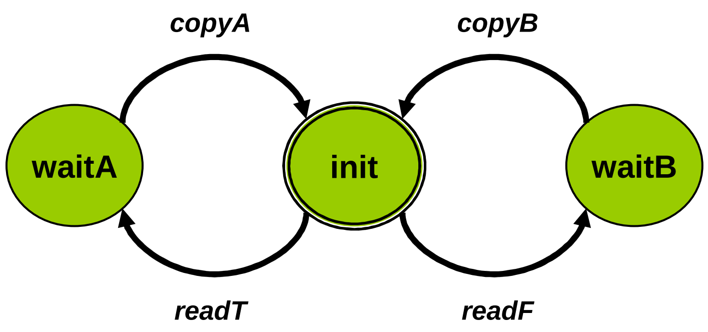


# Scheduling – C action scheduler for FSM

```
// sched_IterSelect.cpp
void sched_IterSelect::process() {
  fsm_state = 1;
  while (1) {
    switch (fsm_state) {
    case 1:
    {           if (S->peek() == 1) {
                        _call_9 = S->get();
                        IterSelect_readT(_actor_variables, _call_9);
                         fsm_state = 2;
                } else {
                        if (S->peek() == 0) {
                        _call_10 = S->get();
                        IterSelect_readF(_actor_variables, _call_10);
                        fsm_state = 3;
                 }}     break; }
    case 2:
    {           _call_11 = A->get();
                IterSelect_copyA(_actor_variables, _call_11, & _out_1);
                Output->put(_out_1);
                fsm_state = 1;
                break; }
  }
   case 3:
    {           _call_12 = B->get();
                IterSelect_copyB(_actor_variables, _call_12, & _out_2);
                Output->put(_out_2);
                fsm_state = 1;
                break; }
  }
  }
}
```



# Cal2C – CAL action samples

```
// Limit pixel value to either [0,255] or [-255,255]

actor GEN_algo_Clip (int isz, int osz) int(size=isz) I, bool SIGNED ==> int(size=osz) O :

  int(size=7) count := -1;
  bool sflag := false;

  read_signed: action SIGNED:[s] ==> guard count < 0 do sflag := s; count := 63; end

  limit.max: action I:[i] ==> O:[ 255 ] guard i > 255 do count := count - 1; end

  limit.zero: action I:[i] ==> O:[ 0 ] guard not sflag, i < 0 do count := count - 1; end

  limit.min: action I:[i] ==> O:[ -255 ] guard i < -255 do count := count - 1; end

  limit.none: action I:[i] ==> O:[ i ] do count := count - 1; end

  priority
    read_signed > limit;
    limit.max > limit.zero > limit.min > limit.none;
  end
end
```

# Cal2C – C action body

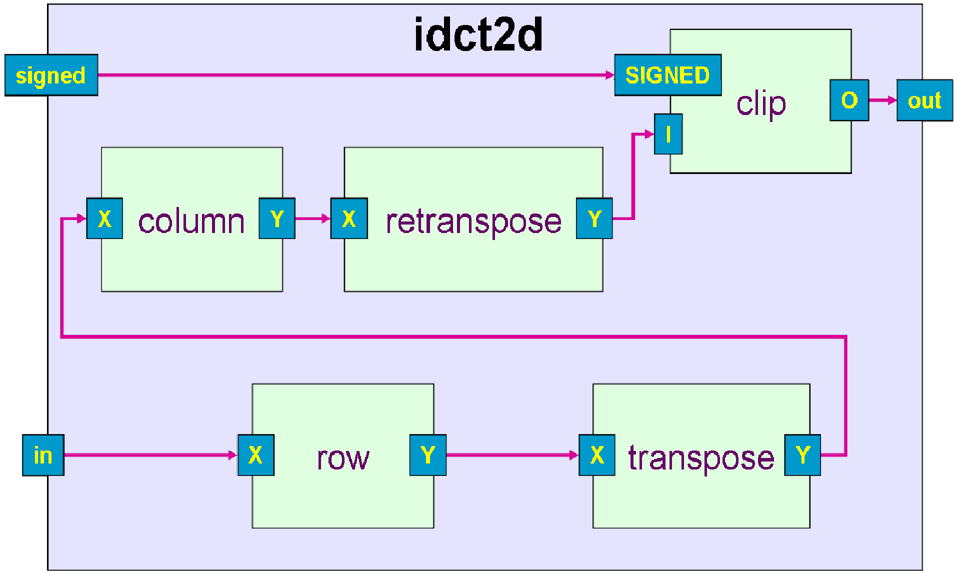
```
struct GEN_algo_Clip_variables {
   int sflag ;
   int count ;
};

// read_signed: action SIGNED:[s] ==> guard count < 0 do sflag := s; count :=
   63; end
void GEN_algo_Clip_read_signed(struct GEN_algo_Clip_variables *_actor_variables
   ,
                               int s )
{
  _actor_variables->sflag = s;
  _actor_variables->count = 63;
}

// limit.none: action I:[i] ==> O:[ i ] do count := count - 1; end
void GEN_algo_Clip_limit_dot_none(struct GEN_algo_Clip_variables
   *_actor_variables ,
                                  int i , int *O )
{
  (_actor_variables->count) --;
  *O = i;
}

…
```

# Scheduling – actors/networks

- Current solution
  - Each actor/network is transformed to a SystemC module
    - The **hierarchy** is preserved
  - The generated code launch a SystemC simulation
    - **MONO**-processor ☹
  - And everything Just Works™ ☺
    - We rely on SystemC data-driven scheduler
    - FIFO sizes are arbitrarily set to 100
      - The MPEG4 SP decoder deadlocks when size is less than 72!

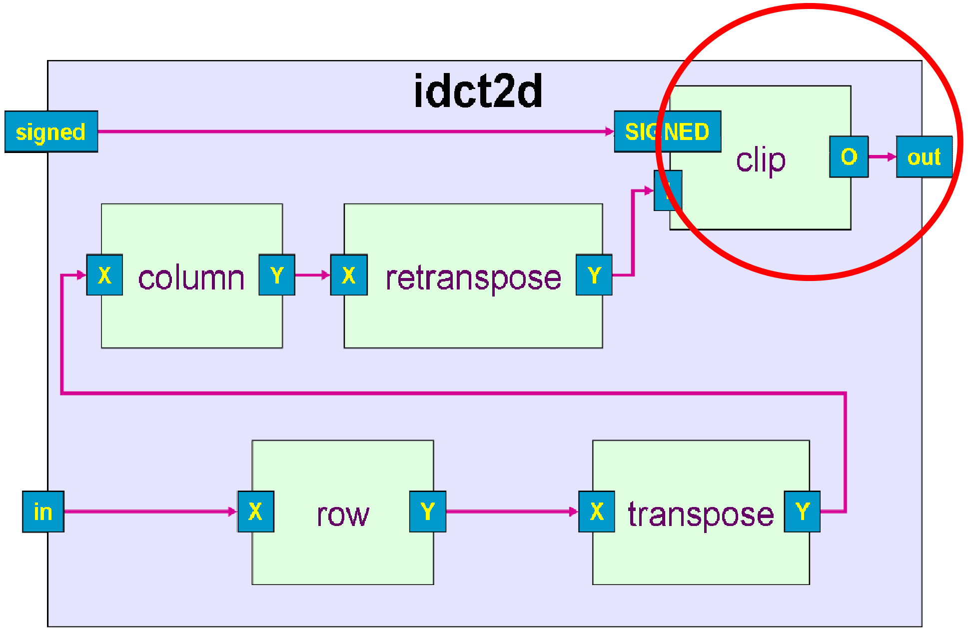
# Scheduling – network translation

```
SC_MODULE(sched_idct2d) {
  // Input and output ports
  sc_port<tlm::tlm_fifo_get_if<int> > _in_;
  sc_port<tlm::tlm_fifo_get_if<int> >
      _cal_signed;
  sc_port<tlm::tlm_fifo_put_if<int> > out;

  // Sub-module instantiation
  sched_GEN_124_algo_Idct1d row;
  sched_GEN_algo_Transpose transpose;
  sched_GEN_124_algo_Idct1d column;
  sched_GEN_algo_Transpose retranspose;
  sched_GEN_algo_Clip clip;

  // Local FIFO channels
  tlm::tlm_fifo<int> row_Y_transpose_X;
  tlm::tlm_fifo<int> transpose_Y_column_X;
  tlm::tlm_fifo<int> column_Y_retranspose_X;
  tlm::tlm_fifo<int> retranspose_Y_clip_I;
…
}
```



# Actor translation

```
SC_MODULE(sched_GEN_algo_Clip) {
  // FIFOs
  sc_port<tlm::tlm_fifo_get_if<int> > I;
  sc_port<tlm::tlm_fifo_get_if<int> > SIGNED;
  sc_port<tlm::tlm_fifo_put_if<int> > O;

  SC_HAS_PROCESS(
    sched_GEN_algo_Clip);

  sched_GEN_algo_Clip(sc_module_name N)
      : sc_module(N) {
    SC_THREAD(process);
  }

  // This function is the action scheduler!
  void process();

};
```

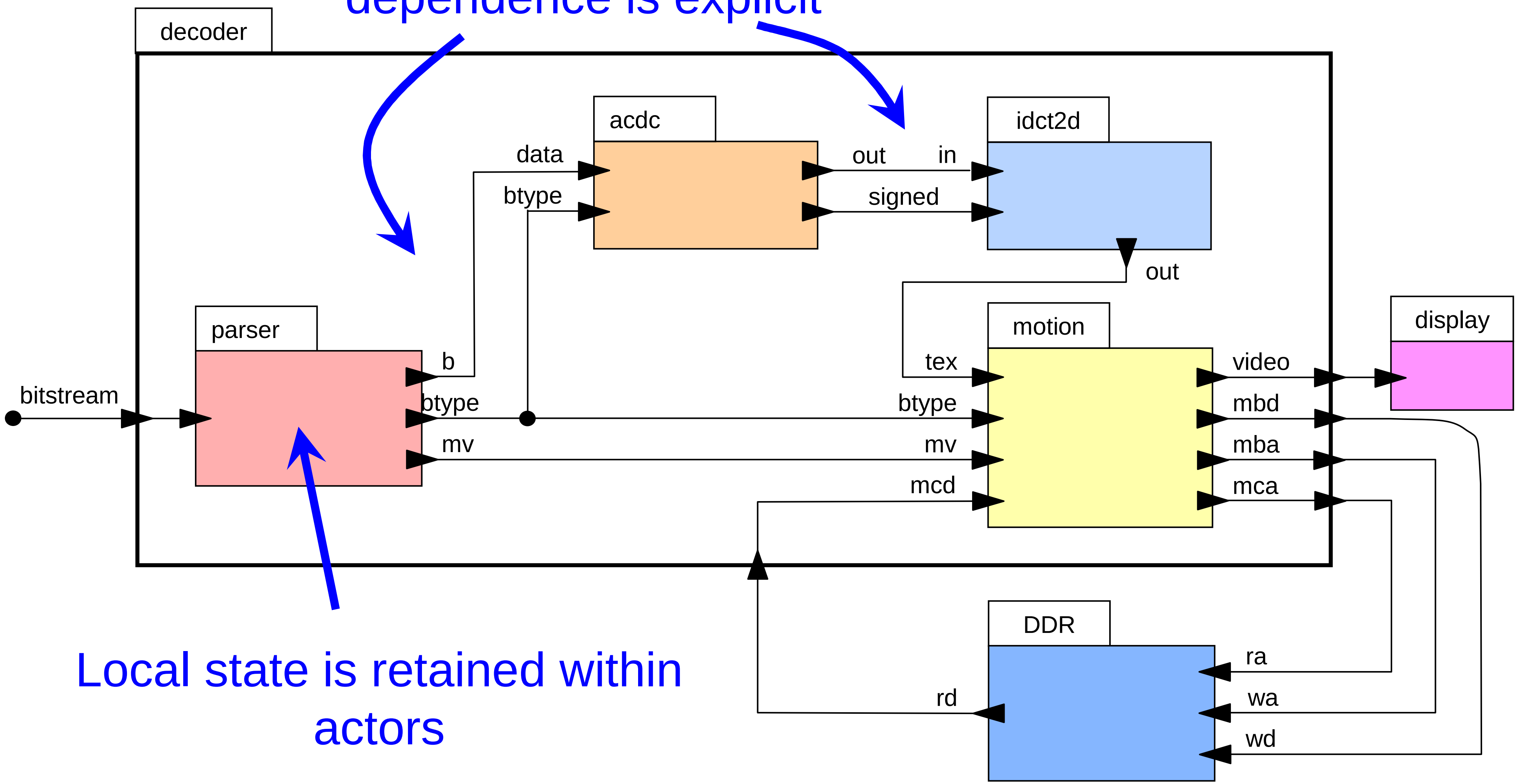


# Scheduling – actors/networks

- Is it possible to replace the current scheduling method?
  - Actor/network scheduler (a.k.a SystemC): yes
  - Action scheduler: it depends…

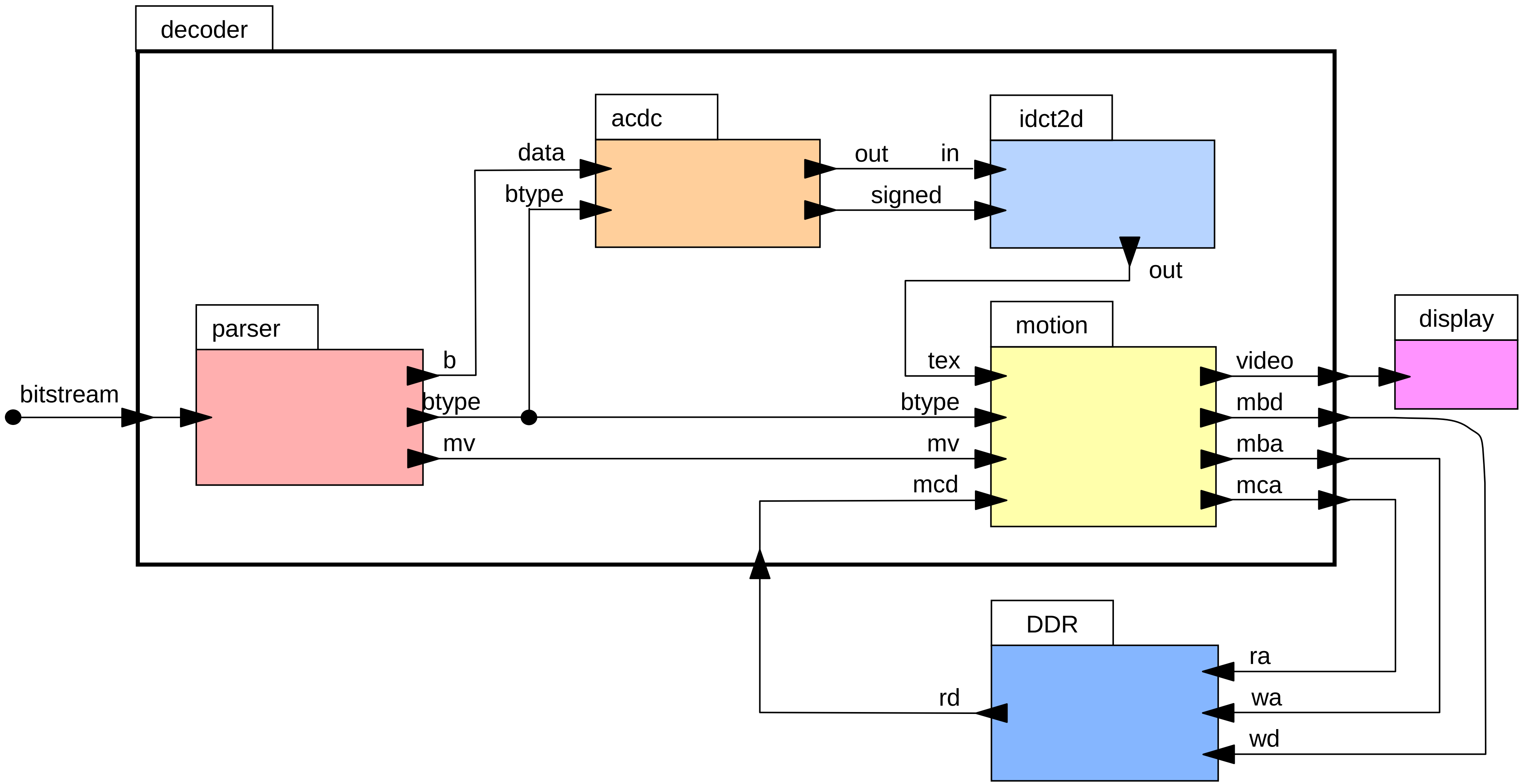
# Scheduling – MPEG4SP decoder

dependence is explicit

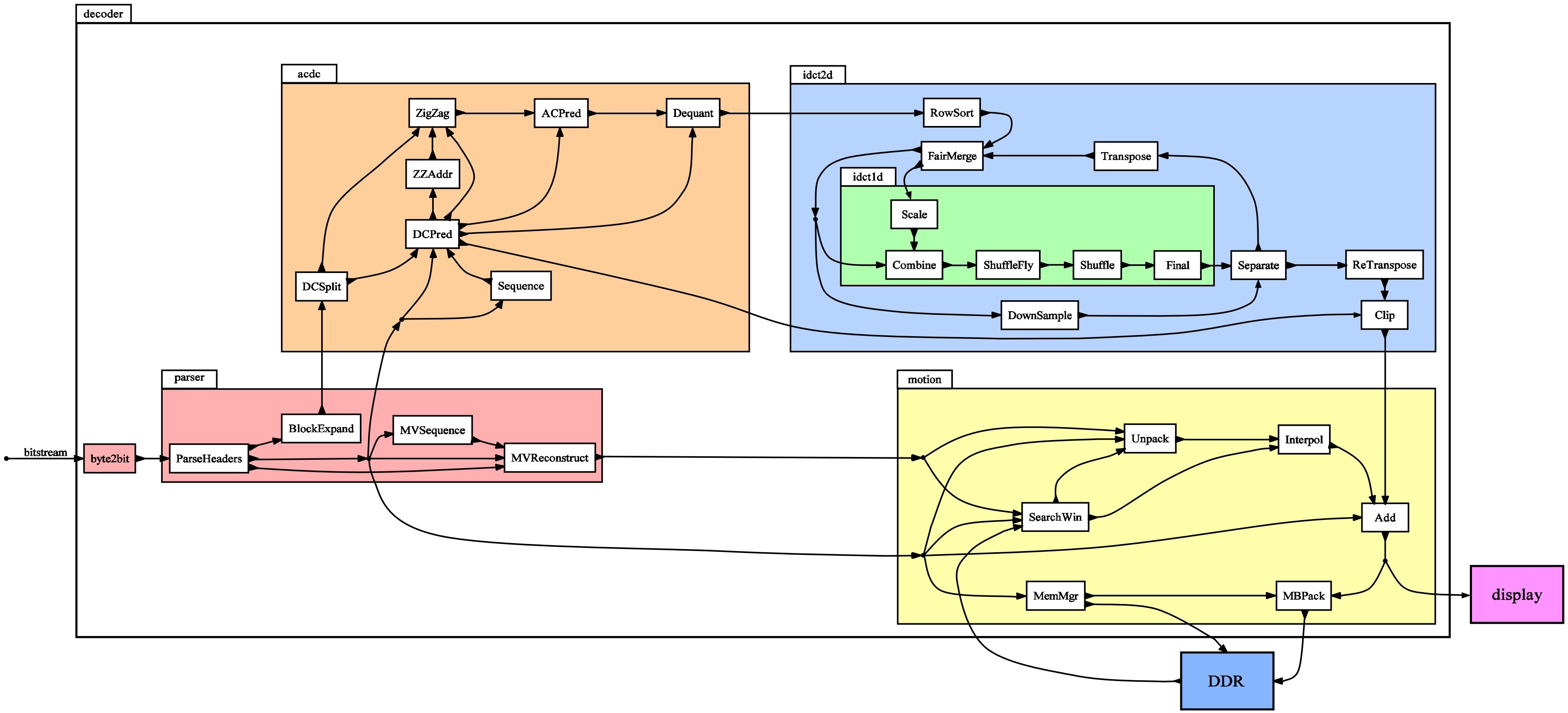



Local state is retained within actors

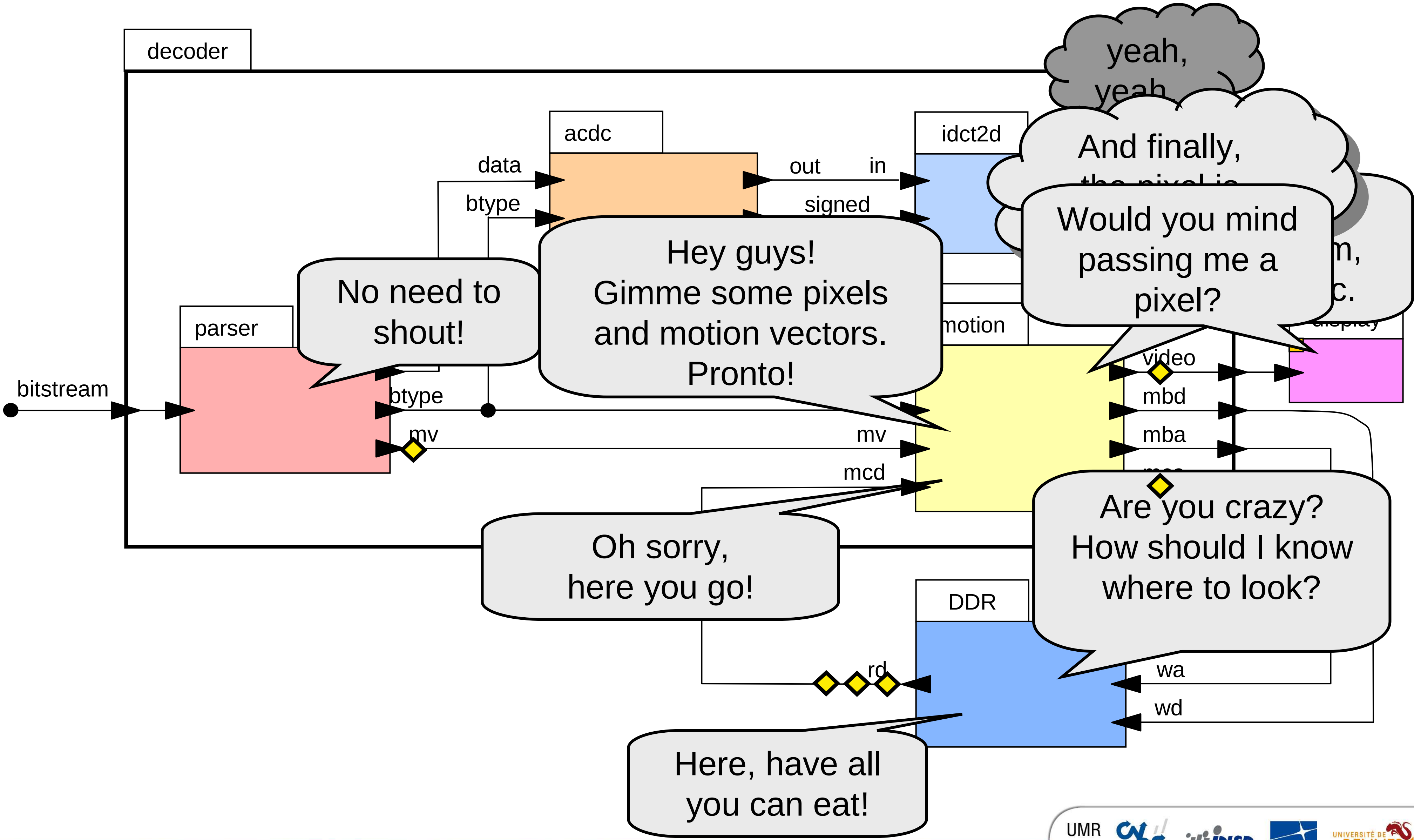
# Scheduling – all actors are concurrent



# Scheduling – …and we've got lots of them



# Actors are too fine-grain…



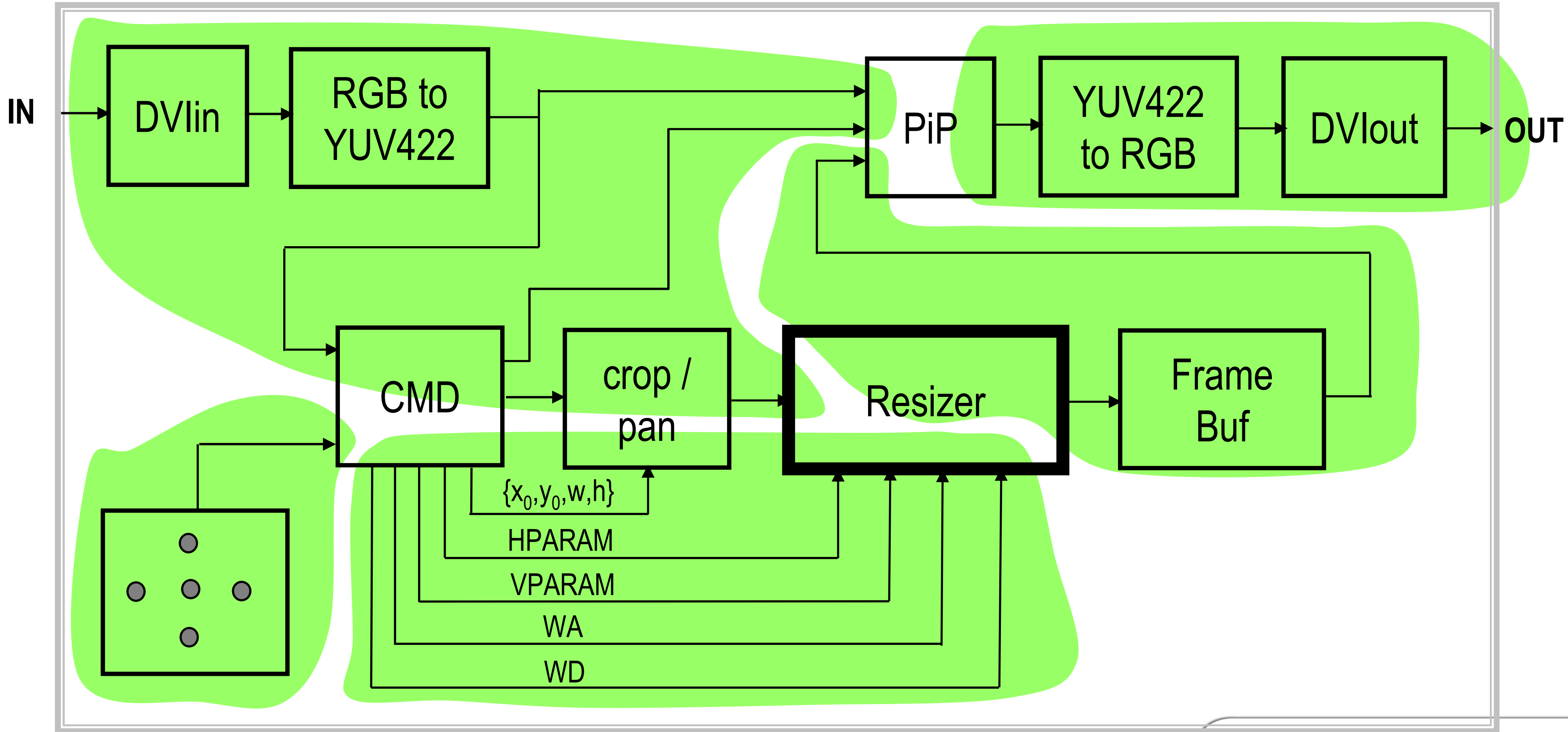
# Scheduling – actors/networks

- Several leads
  - To allow **MULTI**-processor concurrency
    - Move away from SystemC, in favor of YAPI or POSIX threads
    - Implement EPFSS (Jani Boutellier & Shuvra Bhattacharyya)
  - Augment actor granularity (aggregate actors)
    - Thanks to SSR!
    - Particularly useful if we use one “real” thread per actor!
  - Unroll FSMs to statically scheduled loops
    - Work of Jani Boutellier
    - Again, particularly useful to minimize context switching

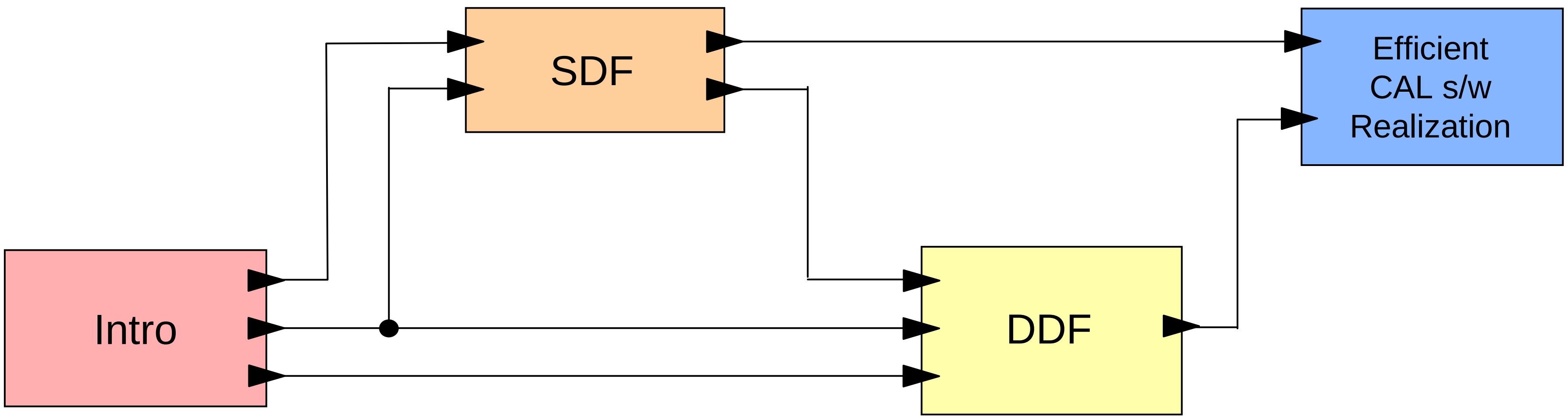
# Scheduling – SSRs (Shuvra's work UMD)

One thread per statically schedulable region.

- multicore software
- starting point for mixed hardware/software implementations



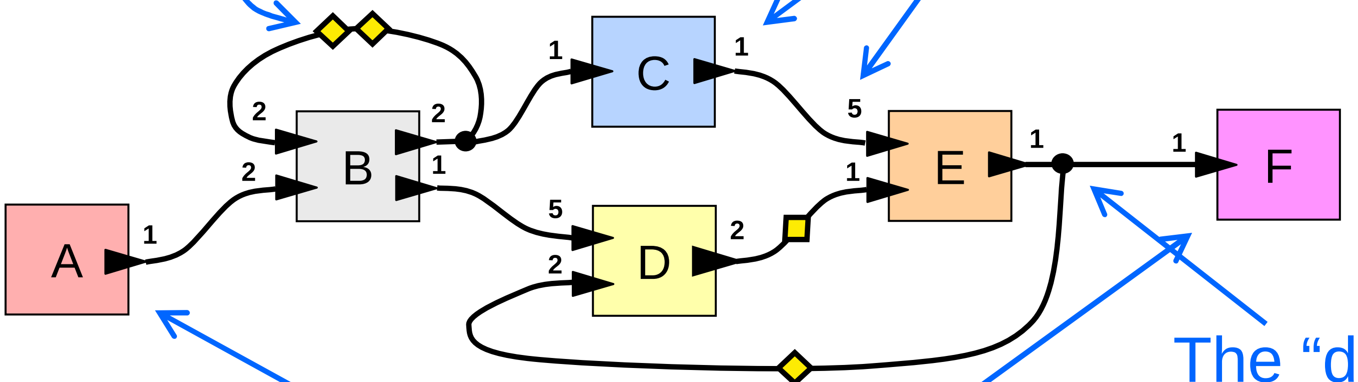
# Scheduling – MoC



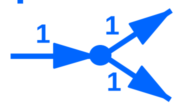
# Scheduling – SDF Example [Lee87]

There may be cycles,
but initial tokens (delays)
are required to avoid deadlock

Token rates are shown
at input/output ports




The "dots" are
duplicators



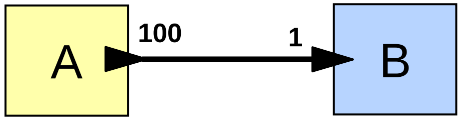

There may be sources
and sinks

# Scheduling – Constructing a schedule

- Any topological ordering of the precedence graph is a valid schedule
  - Fire as soon as enabled
    A A B C C A A B C C A A B C C E F A A B C C A A B C C D E F
  - Minimize buffers
  - Minimize appearances (in looped schedule)
    (A2 B C2)5 E D E F2
  - Other criteria…

# Scheduling – limitations of the SDF model

- Fixed token rates ≈ one CAL action only
- In SDF all tokens must be consumed and produced in a single firing
- SDF can't handle conditional actors
  - Fixed iteration supported by SDF
  - Data-dependent iteration is not

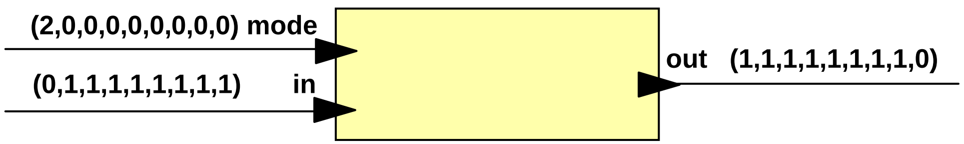



fixed iteration

- Delays required on feedback loops
  - CAL actors can use state variables
  - Avoid reading tokens from loop until tokens produced (e.g. initialization phase)

# Scheduling – Cyclo-Static dataflow [Bilsen96]

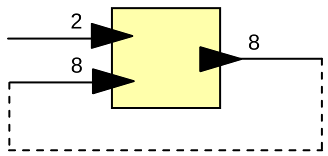
- Actors have periodic token rates




this actor has period 9

each "phase" within the period has fixed rates

- Allows more flexible scheduling
  - Avoids excessive buffer sizes
  - Models dataflow that would deadlock in SDF

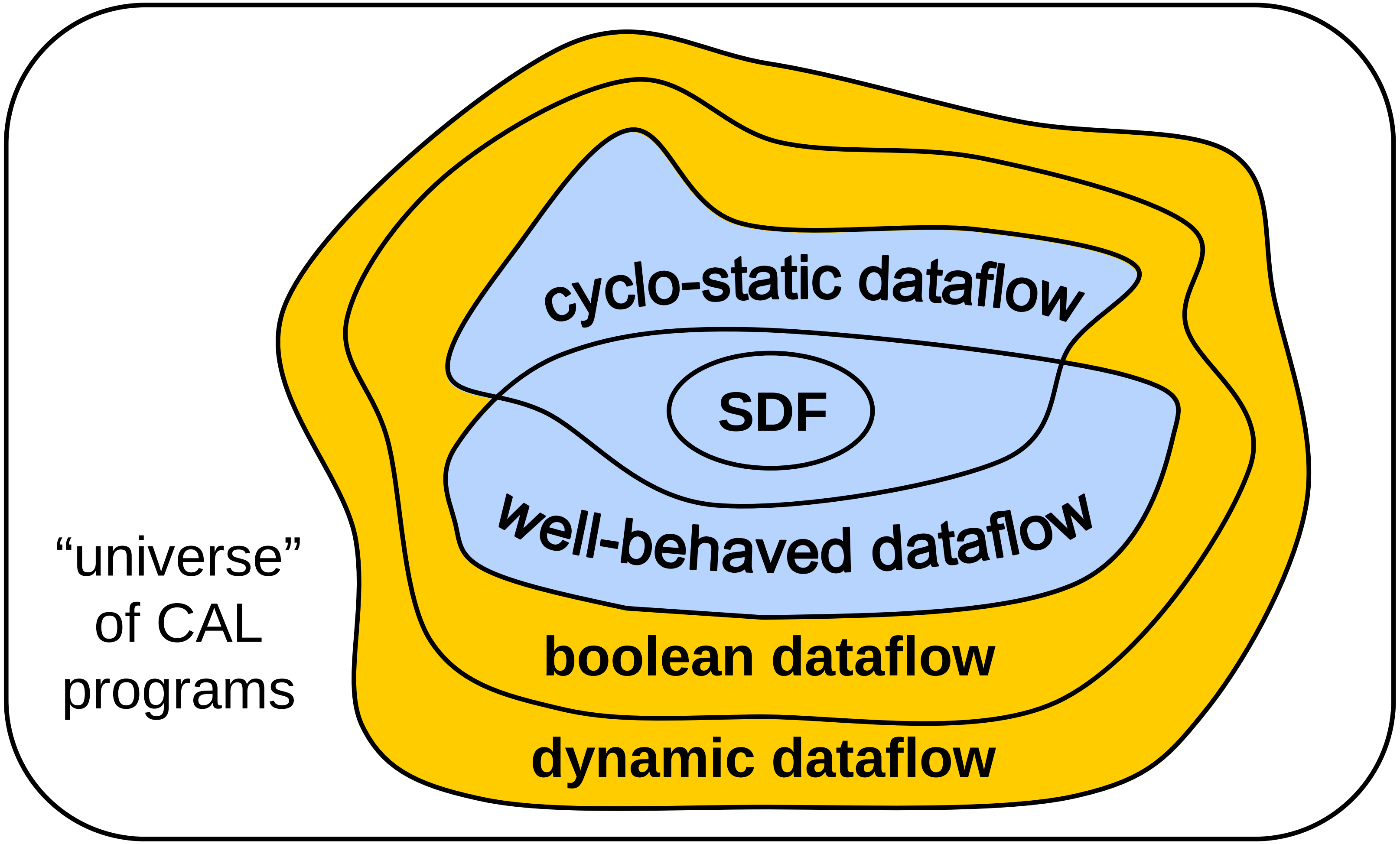



8 delays required on feedback loop



UMR 6164

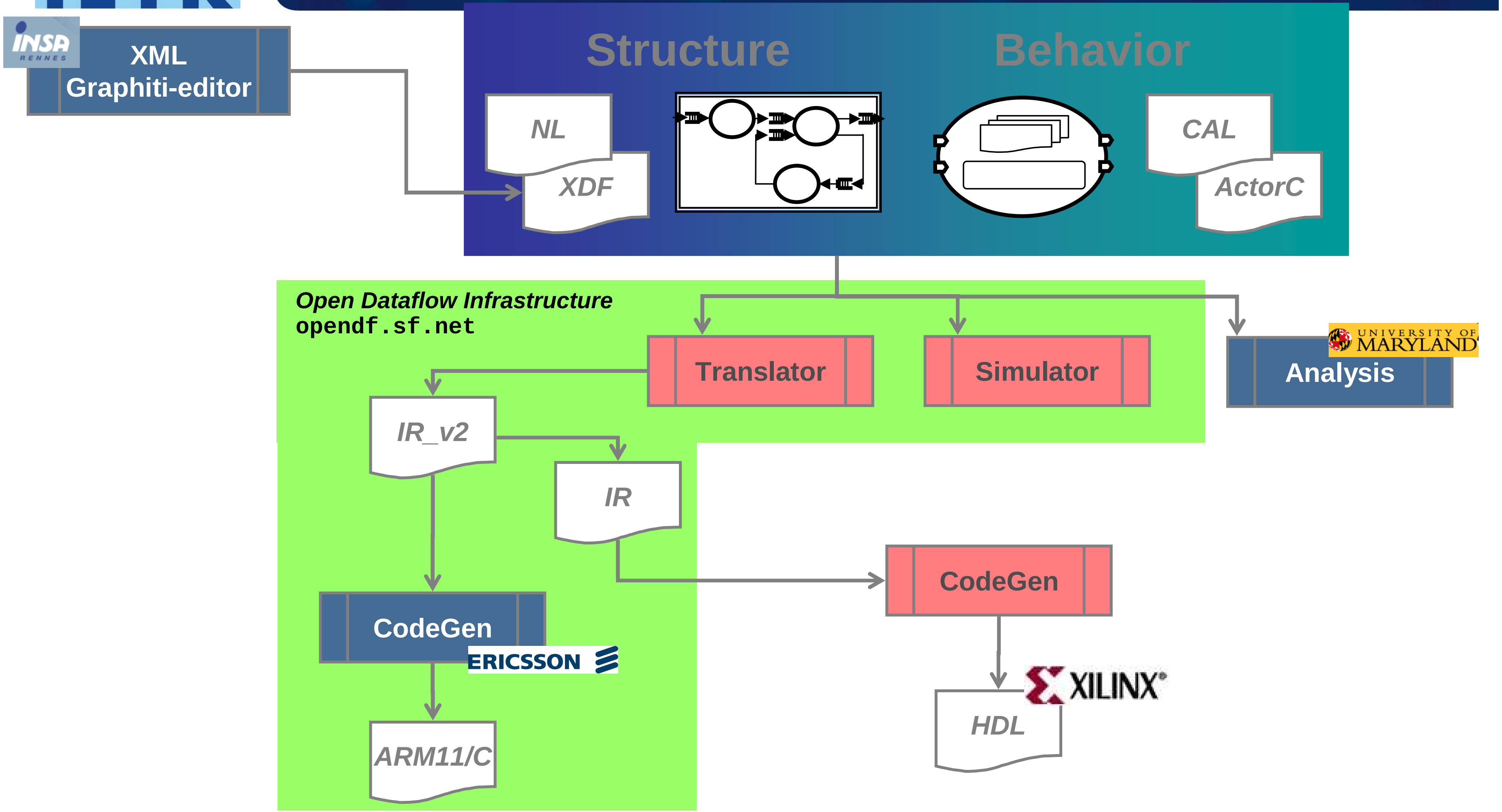
# Scheduling – Dynamic Dataflow (DDF)



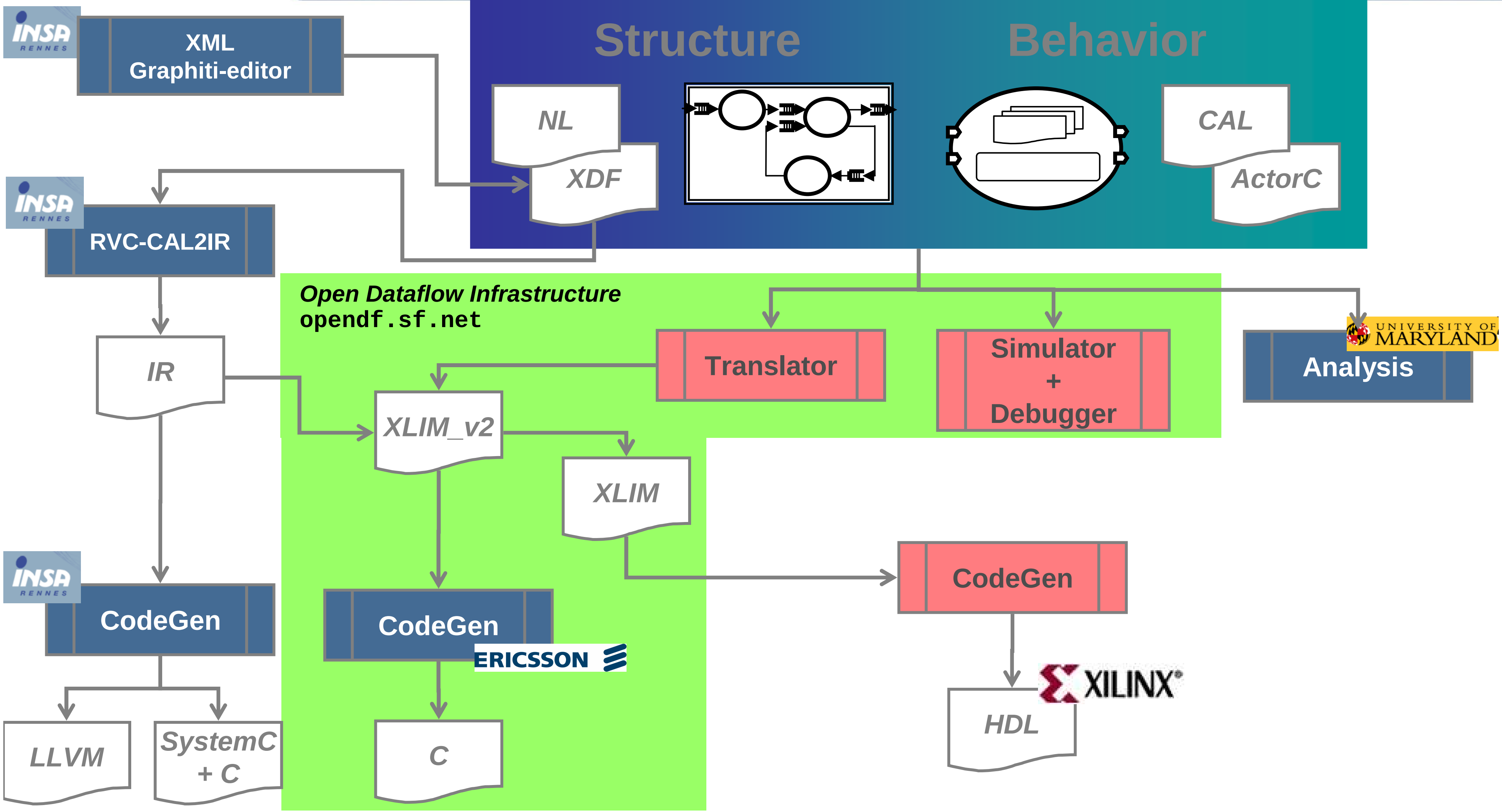
# Scheduling – Dynamic dataflow (DDF) [Lee95]

- A determinate model of computation
  - outputs depend only on past inputs
- Can be implemented using blocking reads from FIFO channels
  - infinite capacity and non-blocking writes assumed
- May have several firing rules (≈CAL actions)
  - conditions on token availability and values (≈guards)
- Mapping from input to output functional
  - but state variables can be thought of as feedback

# Dataflow tools – Current situation

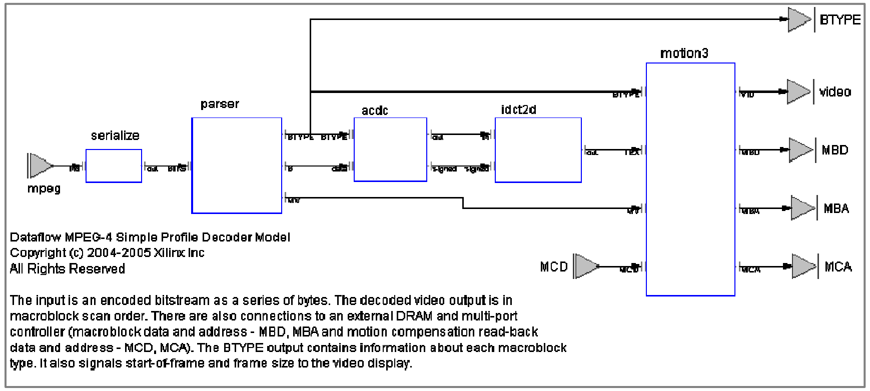


# Dataflow tools - Future

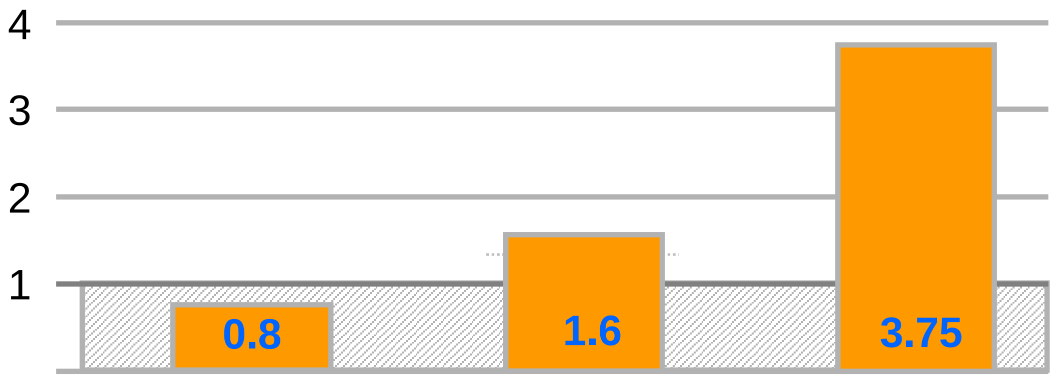


# CAL2HDL – MPEG-4 SP decoder

...0101001111010010010








**Size**
CAL: 3872, 22
VHDL: 4637, 26
(Slices, BRAM)

**Speed**
CAL: 290K
VHDL: 180k
(MacroBlocks/s)
[1080p30: 245k]

**Productivity**
CAL: 4k
VHDL: 15k
(LOC)

**comparison to VHDL reference design**

UMR 6164

# Mpeg-4 SP decoder case study: statistics

**Demo**

# Mpeg-4 SP decoder case study: statistics

| MPEG-4 SP decoder | CAL | NL | C | Cpp | H |
|---|---|---|---|---|---|
| Number of files | 27 | 9 | 27 | 28 | 36 |
| Code size (LOC) | 2 900 | 500 | 5 800 | 3 700 | 900 |

| MPEG-4 SP decoder | Speed (MB/S) | Code size (LOC) |
|---|---|---|
| CAL simulator | 15 | 3 400 |
| Cal2C | 2 000 | 10 400 |
| Cal2HDL | 290 000 | 4 000 |

# SF repositories

- OpenDF
  - http://opendf.wiki.sourceforge.net/
- CAL2ARM11
  - http://opendf.wiki.sourceforge.net/
- CAL2HDL
  - http://sourceforge.net/projects/openforge/
- RVC-CAL 2 IR
  - http://sourceforge.net/projects/orcc/

# Collaborations

- Reconfigurable Video Coding
  - Marco Mattavelli, *EPFL*, Switzerland
  - RVC experts
- Statically schedulable region (SSR)
  - Shuvra Bhattacharyya, Ruirui Gu, *University of Maryland*, USA
- Analysis of configuration and processing actions
  - Jani Boutellier, *University of Oulu,* Finland, Switzerland
- OpenDF, co-design with Cal2C+Cal2HDL
  - Jörn Janneck, Rob Esser, *Xilinx*, USA
  - Ian Miller, Dave Parlour formely Xilinx



UMR 6164

## Conferences

- DASIP ( Chairman Marco Mattavelli)
  - http://www.ecsi-association.org/ecsi/dasip/
  - Deadline April 14
  - Co-located with FDL 2009
- Special session in DASIP (chairman Mickael Raulet)
  - Reconfigurable Video Coding-- Scheduling and Dynamic Reconfiguration of Dataflow Programs