Embedded reactive programming in MLIR

Hugo Pompougnac, **Dumitru Potop-Butucaru** – Inria

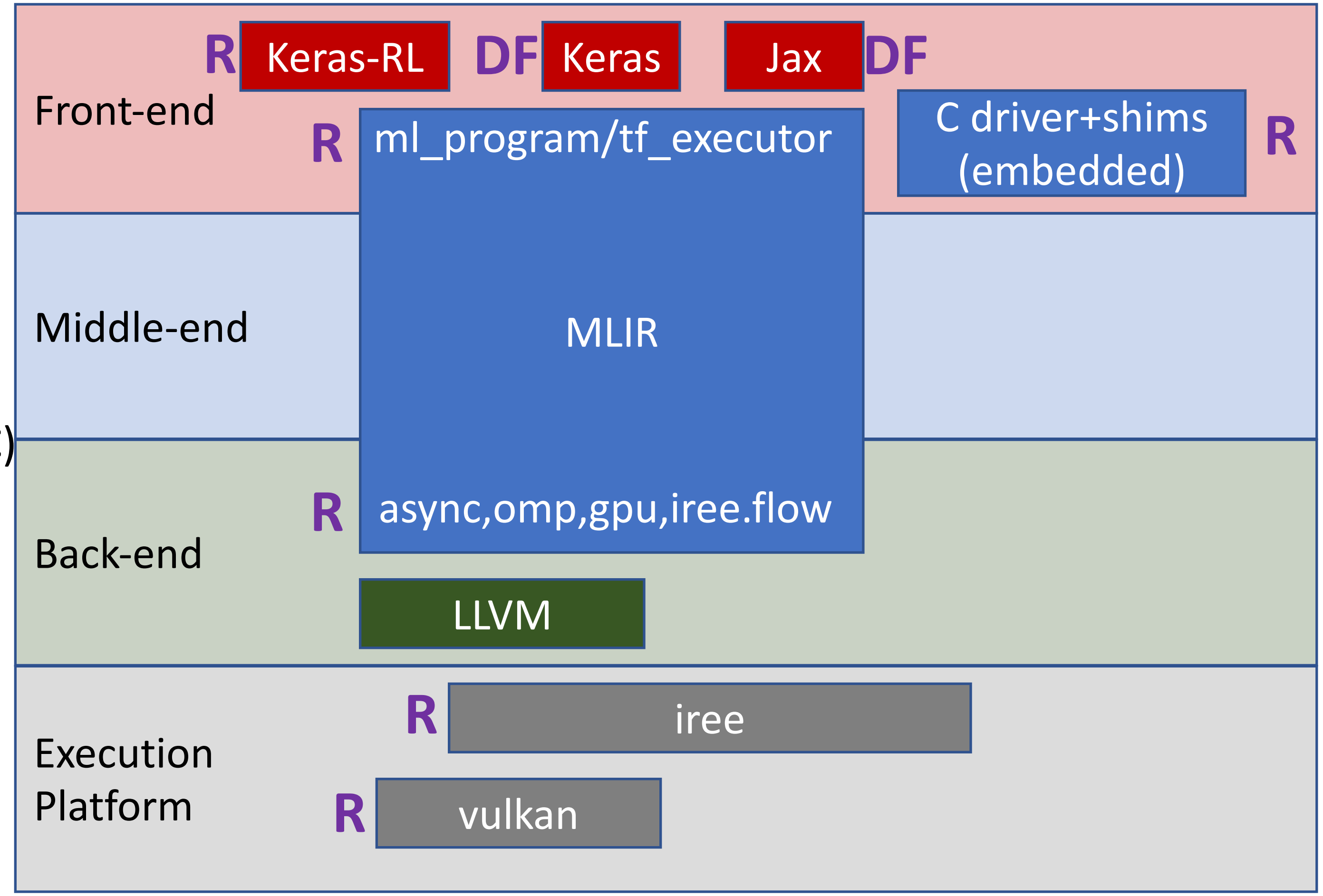
{hugo.pompougnac,dumitru.potop}@inria.fr

Albert Cohen – Google

albertcohen@google.com

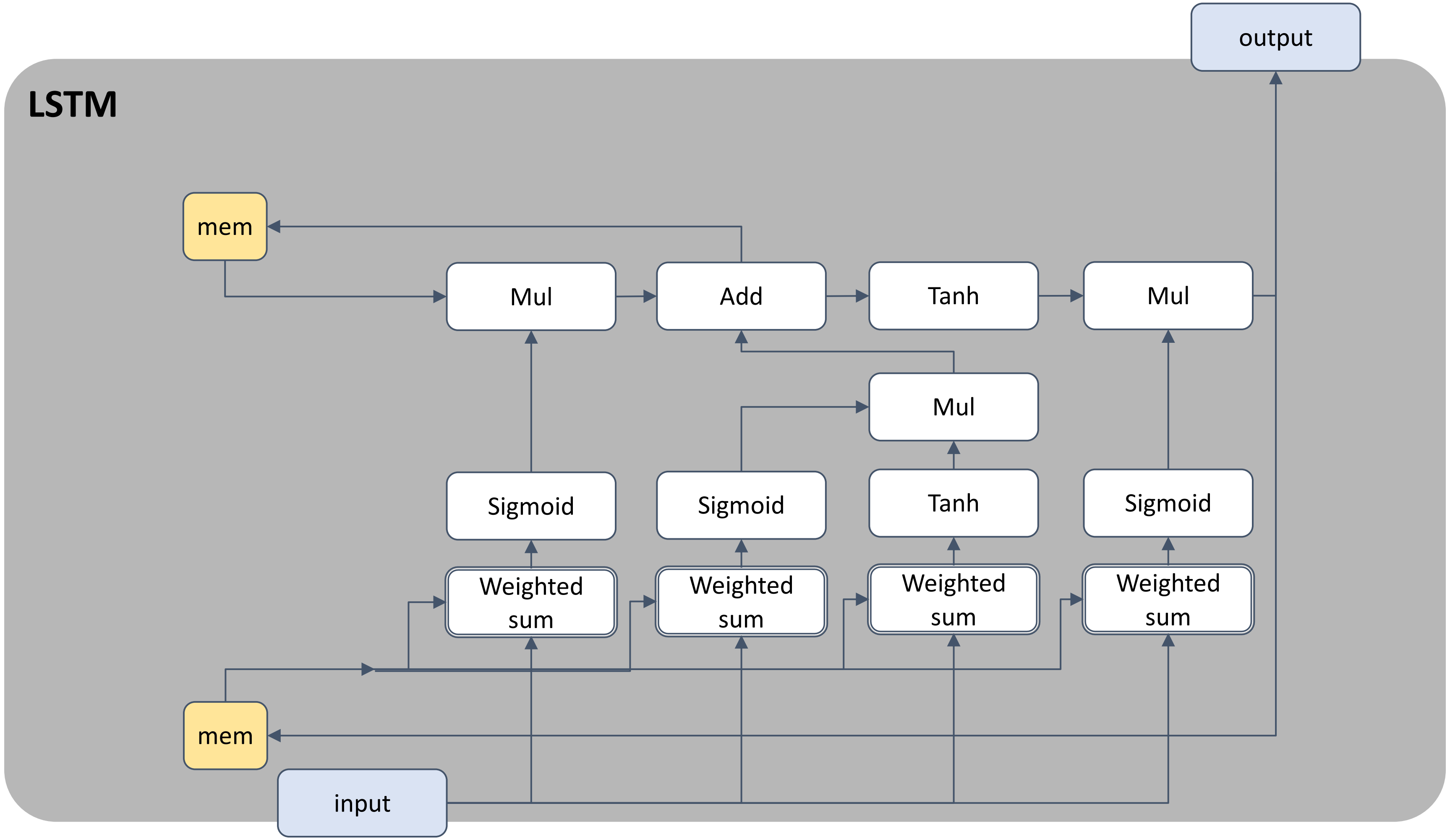
Reactive and dataflow programming: Why?

- Elements in the front-end, the back-end, the run-time
  - Little (no?) middle end
  - Multiple disparate, ad hoc approaches
    - Unclear semantics (Python/MLIR/C)
    - Difficult to specify/compile
  - Back-end choses encodings
    - Loss of optimization opportunities
- Proposal:
  - Unify front-end practice around a general-purpose DF specification
  - Propose a few primitives allowing to connect front-end and back-end



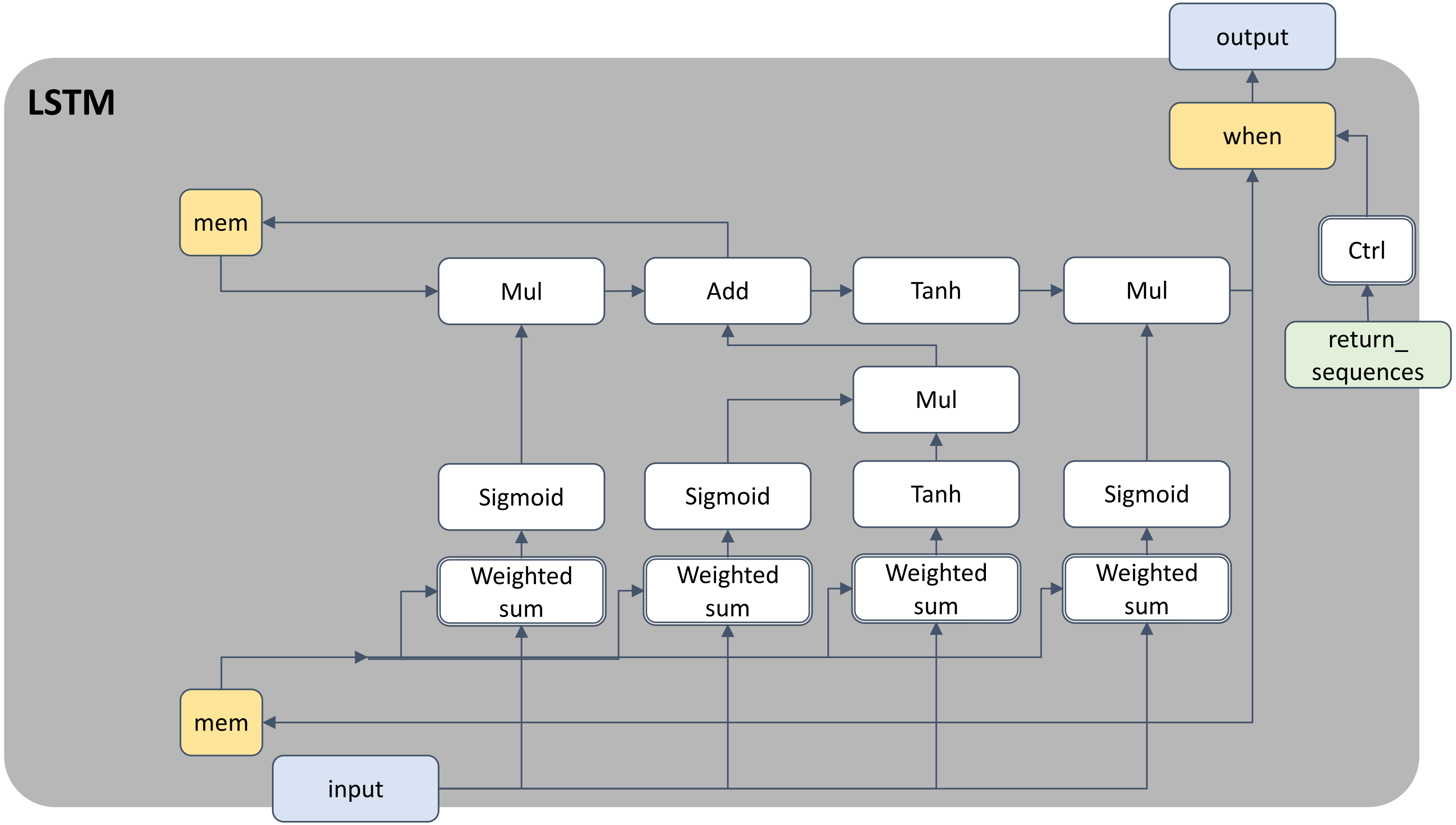
Motivation: **Stateful networks** (e.g. RNNs)

- Specification
  - Intuition: Dataflow



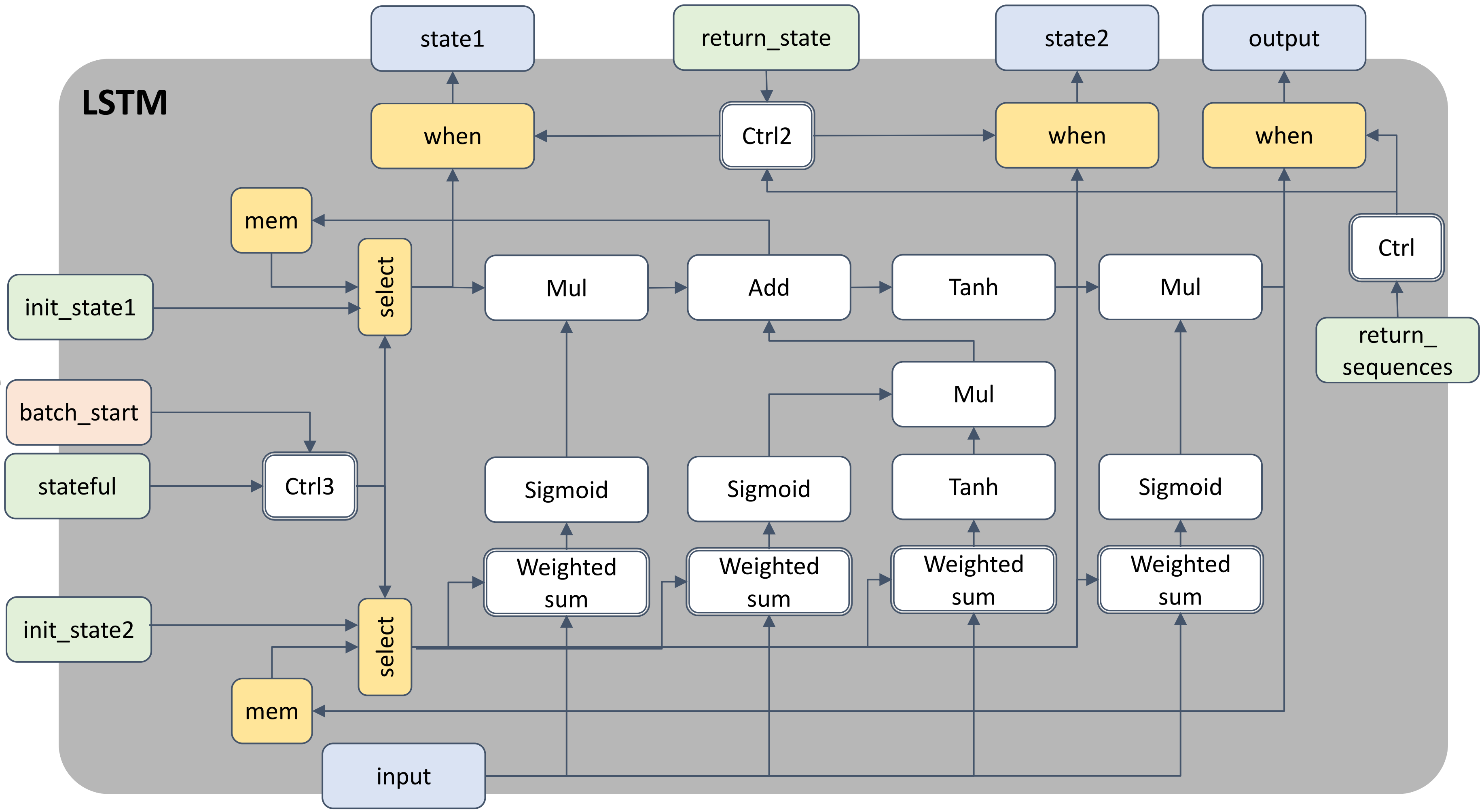
Motivation: **Stateful networks** (e.g. RNNs)

- Specification
  - Intuition: Dataflow
  - Compilation: Time-space conversion
    - Single function call for whole history (no state)



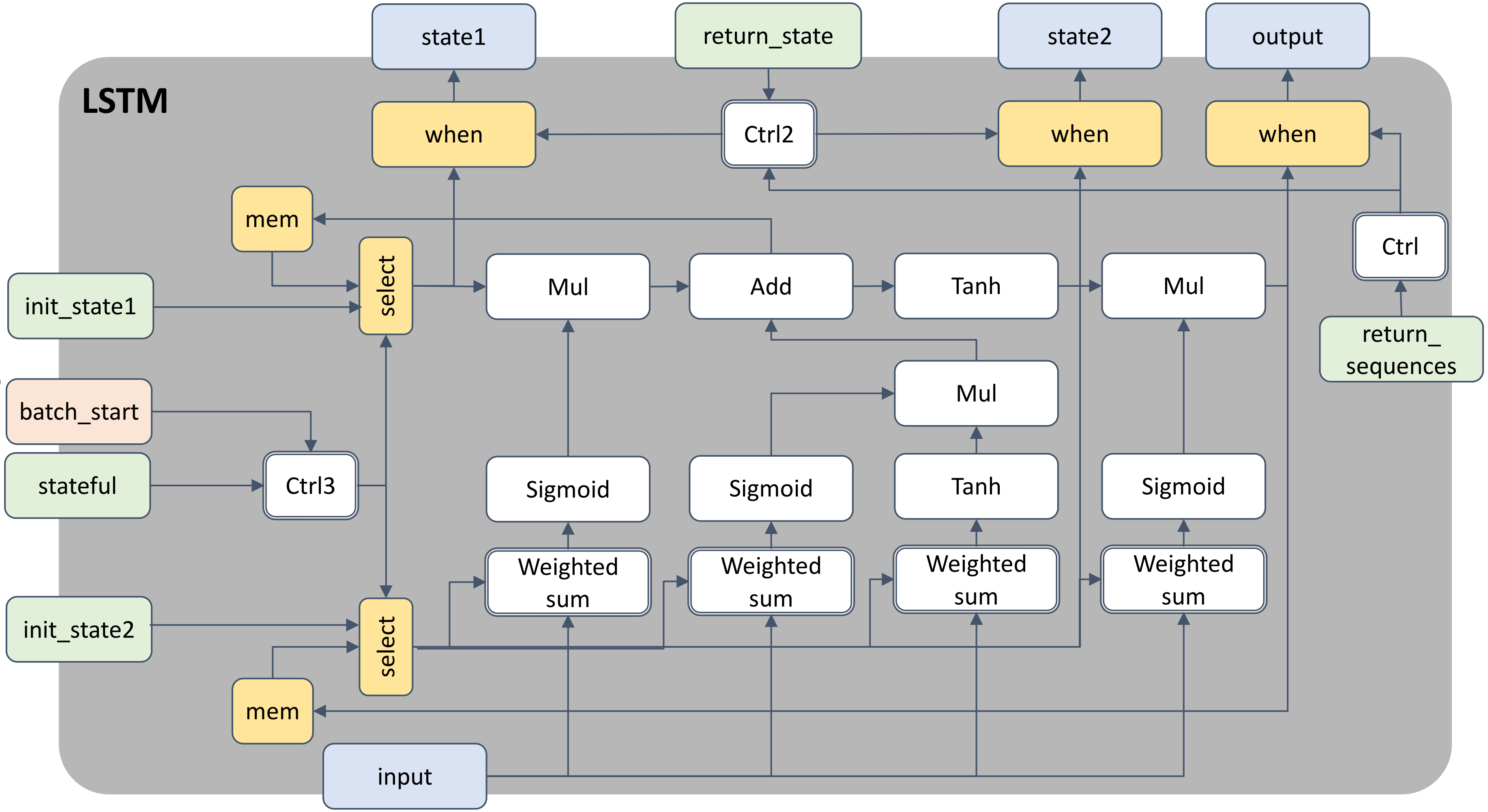
Motivation: **Stateful networks** (e.g. RNNs)

- Specification
  - Intuition: Dataflow
  - Compilation: Time-space conversion
  - Overall: Streaming semantics (sequence of batches)
    - Python-level only



Motivation: **Stateful networks** (e.g. RNNs)

- Specification
  - Intuition: Dataflow
  - Compilation: Time-space conversion
  - Overall: Streaming semantics (sequence of batches)
    - Python-level only
  - Semantic mess
    - And we did not discuss training/RL



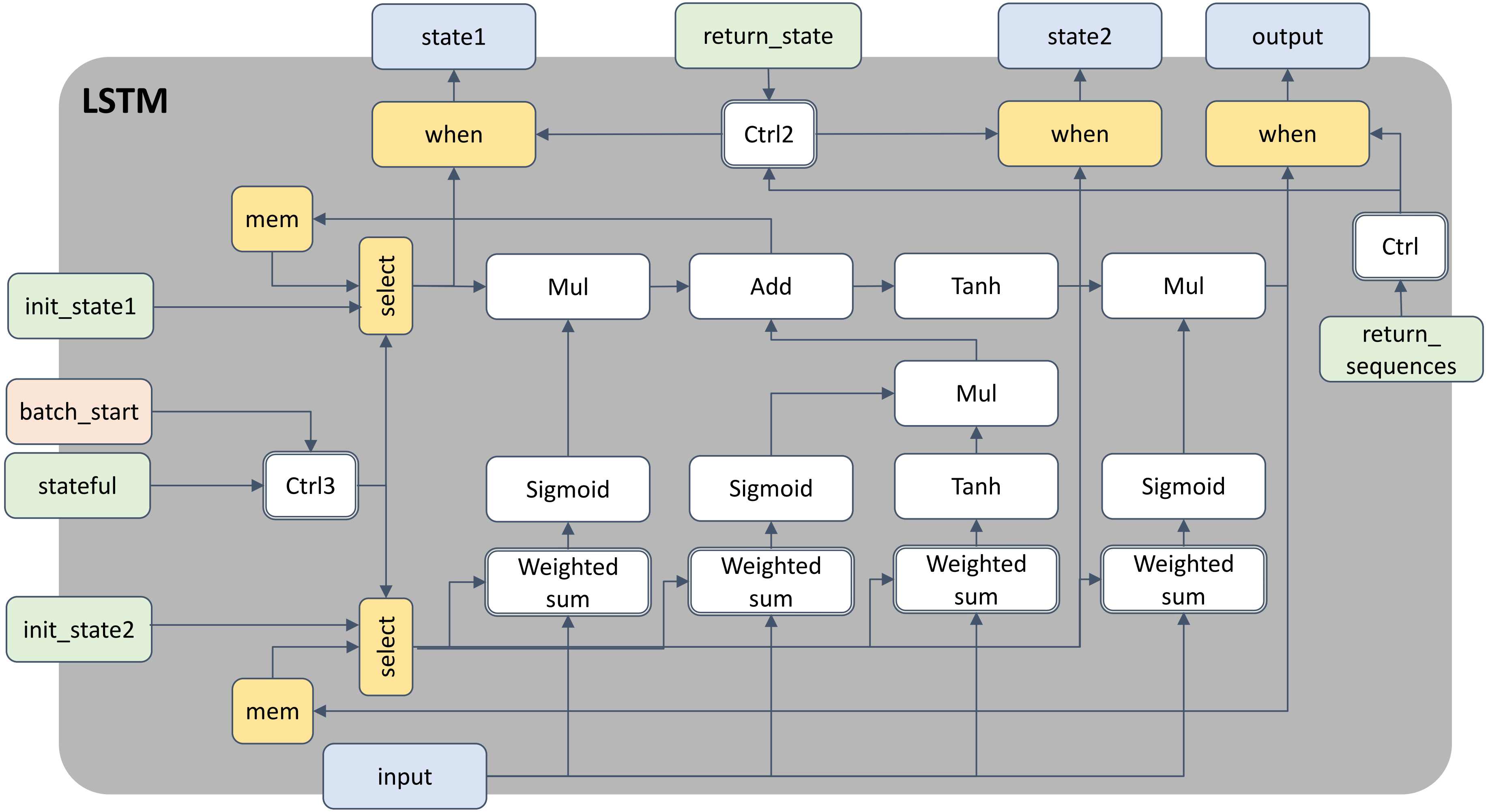
General dataflow specification supports all specializations/compilations

Motivation: **Stateful networks** (e.g. RNNs)

- Implementation
  - Semantic mess

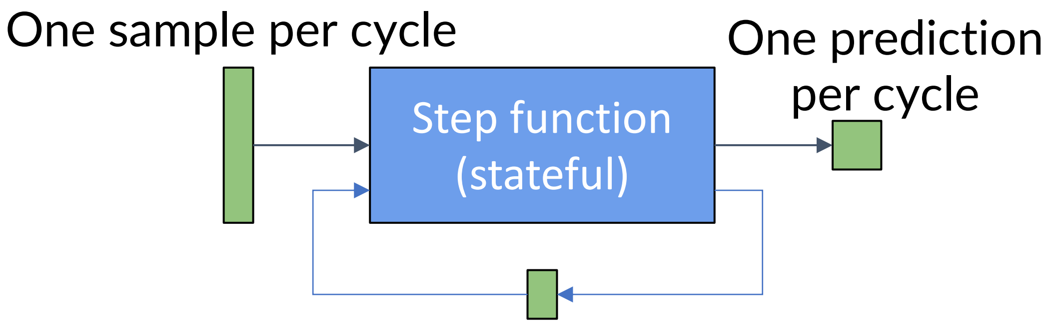
  

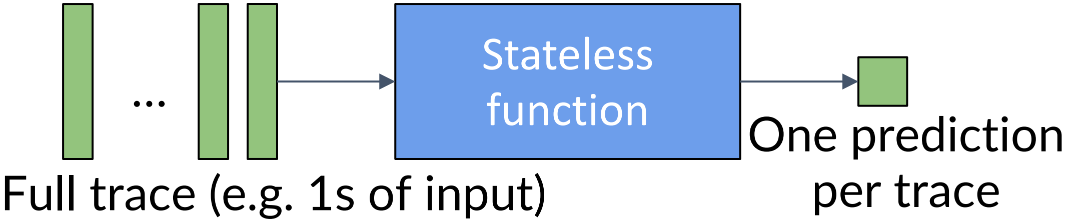
  - High-level escapes (true) compilation
    - Python freedom
    - Ad hoc solutions
    - Difficult to understand/debug
  - Little codegen modularity
  - Efficient algorithms difficult to specify
    - Gating, multi-period



Motivation: **Streaming implementations**

- All in-place implementations (RNNs, convolutional…)
  - Dataflow/streaming intuition
    - Reactive behavior – stateful cyclic execution
    - State initialized once, at execution beginning
  - Keras, PyTorch interpretation = time->space conversion
    - Fixed trace size
      - Training done for fixed trace size
    - Traditional function
      - Fixed-size loop over tabulated input
    - **Back-ends cannot represent stateful behaviors in time**
      - **Unless using ad-hoc extensions (e.g. kws-streaming) or converting to global vars**

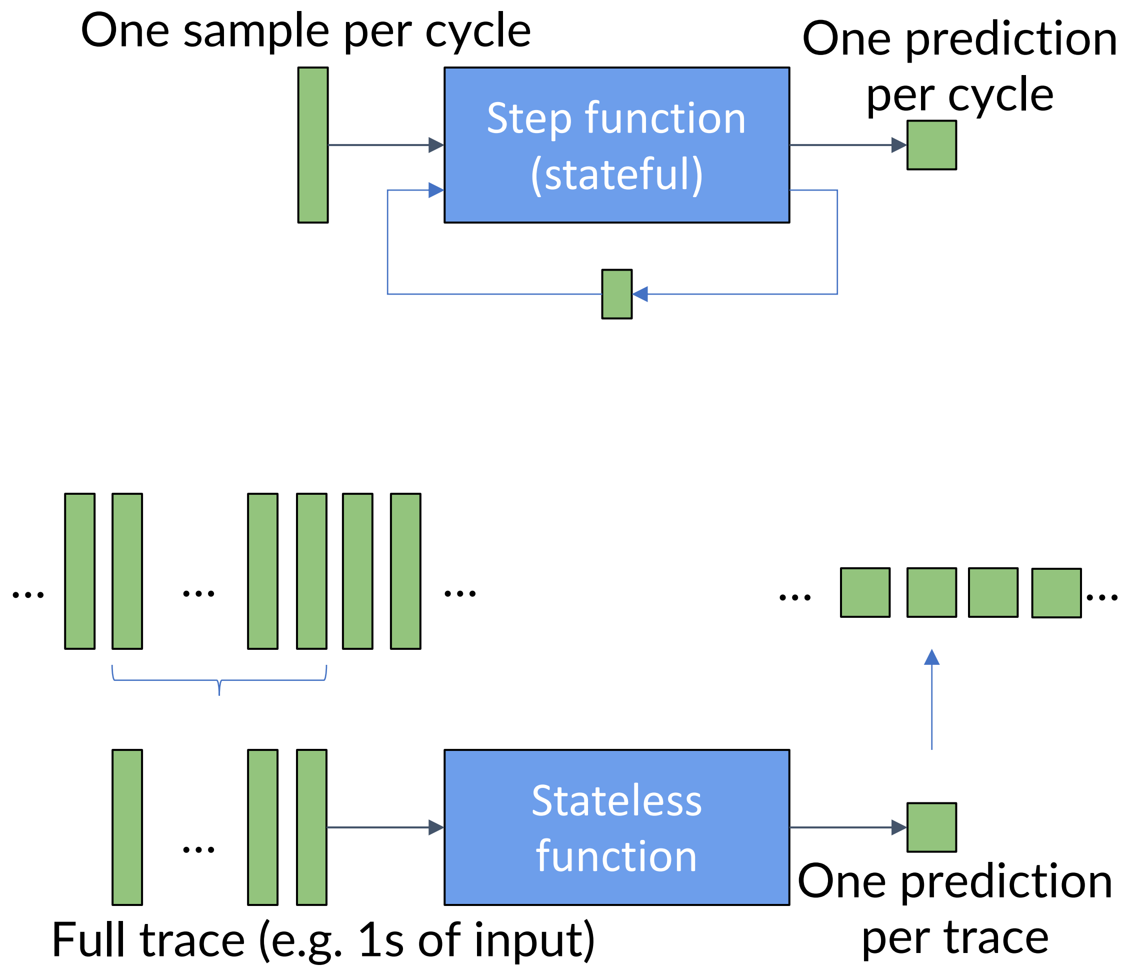




Motivation: **Streaming implementations**

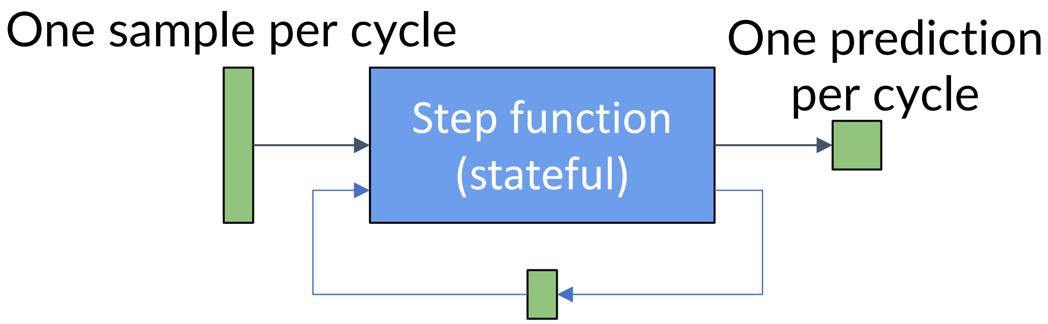
- Streaming RNN implementation
  - V1: Add sliding window over inputs
    - ++ Prediction corresponds to training
    - ++ No changes needed to generated code
    - - - Low efficiency – each sample processed multiple times

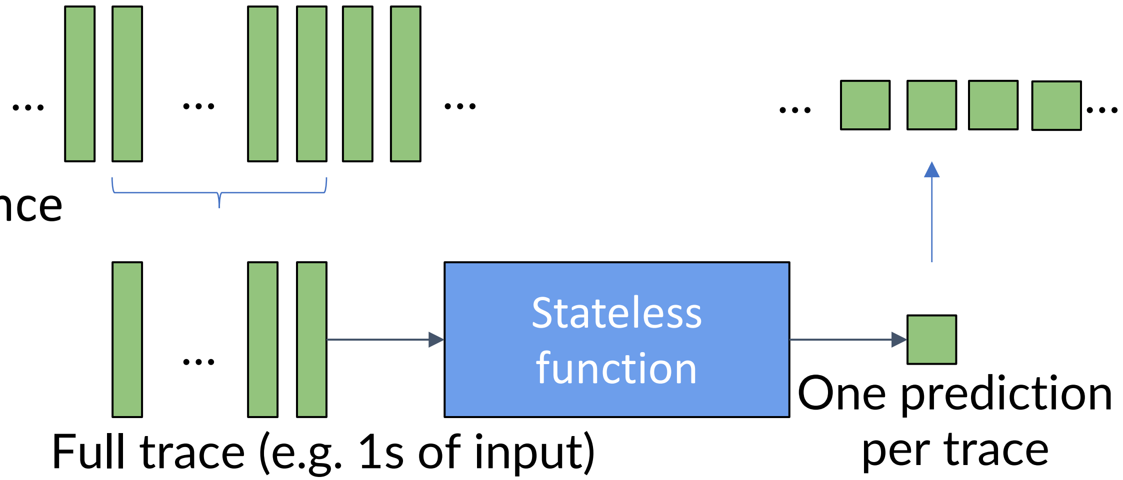
  - **Outside of Keras/PyTorch** (manually)



Motivation: **Streaming implementations**

- Streaming RNN implementation
  - V1: Add sliding window over inputs
    - ++ Prediction corresponds to training
    - ++ No changes needed to generated code
    - - - Low efficiency – each sample processed multiple times
  - V2: Transform the fixed-size loop into an infinite loop
    - ++ High efficiency – each sample processed only once
    - - - Changes needed to generated code
    - - - Prediction does not correspond to training
  - **Outside of Keras/PyTorch** (manually)





Our contribution (in a nutshell)

- lus = dataflow synchronous MLIR dialect
  - **General-purpose reactive specification inside MLIR**
    - Incorporate the primitives of the Lustre language
    - Stateful scheduled components + hierarchy + gated execution (predication)
    - Import from Keras – natural semantics
  - Compilation to **efficient and reactive executable code**
    - No performance loss w.r.t. traditional (non-reactive) implementation
    - Easy to interface with reactive system code (synchronous codegen conventions)

- Ongoing work
  - Synthesis of training code for stateful components
    - Lift reactive specification towards Jax level
  - Resource allocation

HiPEAC2022, https://dl.acm.org/doi/10.1145/3506706

https://github.com/dpotop/mlir-lus-public - the public version of the SW

```
lus.node @lstm(%data:tensor<40xf32>,%rst:i1)
                        -> (tensor<4xf32>) {
  // Feedback and reset control
  %c0   = tf.Const(){dense<…>}
  %tmp0 = lus.fby %c0 %s0o
  %24a  = lus.when     %rst %c0
  %24b  = lus.when not %rst %tmp0
  %24   = lus.merge    %rst %24a %24b
  %c1   = tf.Const(){dense<…>}
  %tmp1 = lus.fby %c1 %s1o
  %25a  = lus.when     %rst %c1
  %25b  = lus.when not %rst %tmp1
  %25   = lus.merge    %rst %25a %25b
  // LSTM computational core
  %v26 = tf.MatMul(%v24, %o76)
  %v28 = tf.MatMul(%data, %o22)
  %v29 = tf.AddV2(%v28, %v26)
  %v30 = tf.BiasAdd(%v29, %o78)
  %dim = tf.Const() {value = dense<1>}
  %v31_0, %v31_1, %v31_2, %v31_3
       = tf.Split(%dim, %v30)
  %v32 = tf.Relu(%v31_2)
  %v33 = tf.Sigmoid(%v31_0)
  %v34 = tf.Mul(%v33, %v32)
  %v35 = tf.Sigmoid(%v31_1)
  %v36 = tf.Mul(%v35, %v25)
  %s1o = tf.AddV2(%v36, %v34)
  %v40 = tf.Relu(%lstm_out)
  %v41 = tf.Sigmoid(%v31_3)
  %s0o = tf.Mul(%v41, %v40)
  // Output subsampling
  %o   = lus.when %rst %s1o
  lus.yield (%o: tensor<3x1xf32>)
}
lus.node @model(%data:tensor<40xf32>)
                    ->(tensor<4x1xf32>) {
 %rst = lus.inst @counter()
 %x = lus.inst @lstm(%data,%rst)
 %o = lus.inst @dense(%x)
 lus.yield (%x3: tensor<1x4xf32>)
```

Why it may interest you even more (1/2)

- Natural expression at IR high level of:
  - Stateful behaviors – hierarchical modular specification
    - RNNs
    - Reinforcement Learning
    - Attention/transformers…
  - Predicated execution
    - Resetting
    - Sparsely-gated mixture of experts
      - Multi-period activation…
  - Preprocessing and post-processing code
    - Sliding windows
    - Mix with ML code for efficient compilation/execution
  - Another approach to undefinedness and correctness

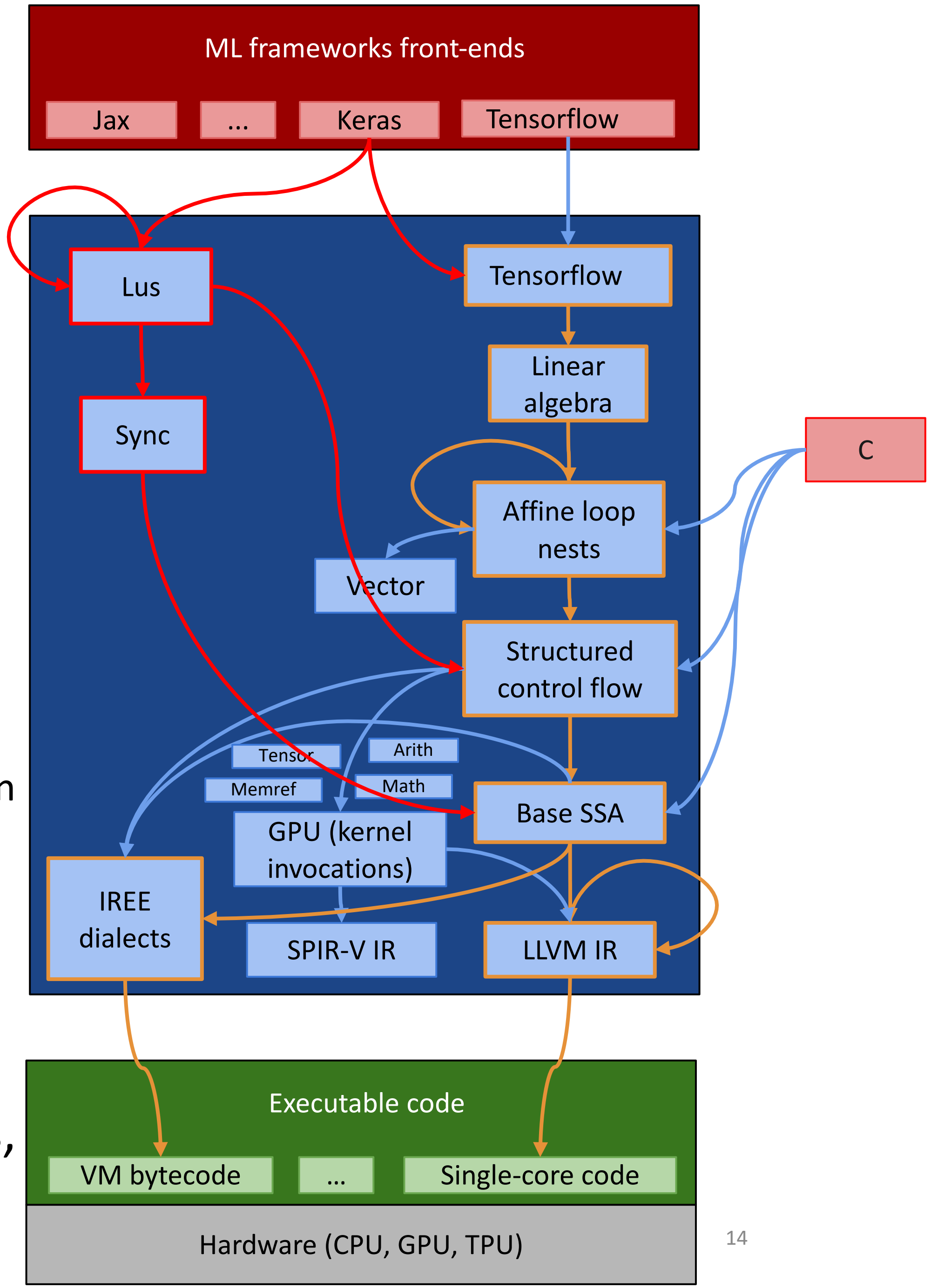
Why it may interest you even more (2/2)

- Maintain statefulness throughout compilation
  - No need to convert (too early) into stateless functions or global variables
    - No (early) loss of
      - High-level information
      - Optimization potential
  - Modular reactive code generation

- Possible reuse of resource allocation approaches of dataflow languages
  - Memory allocation (e.g. static)
  - Resource access ordering, synchronization…

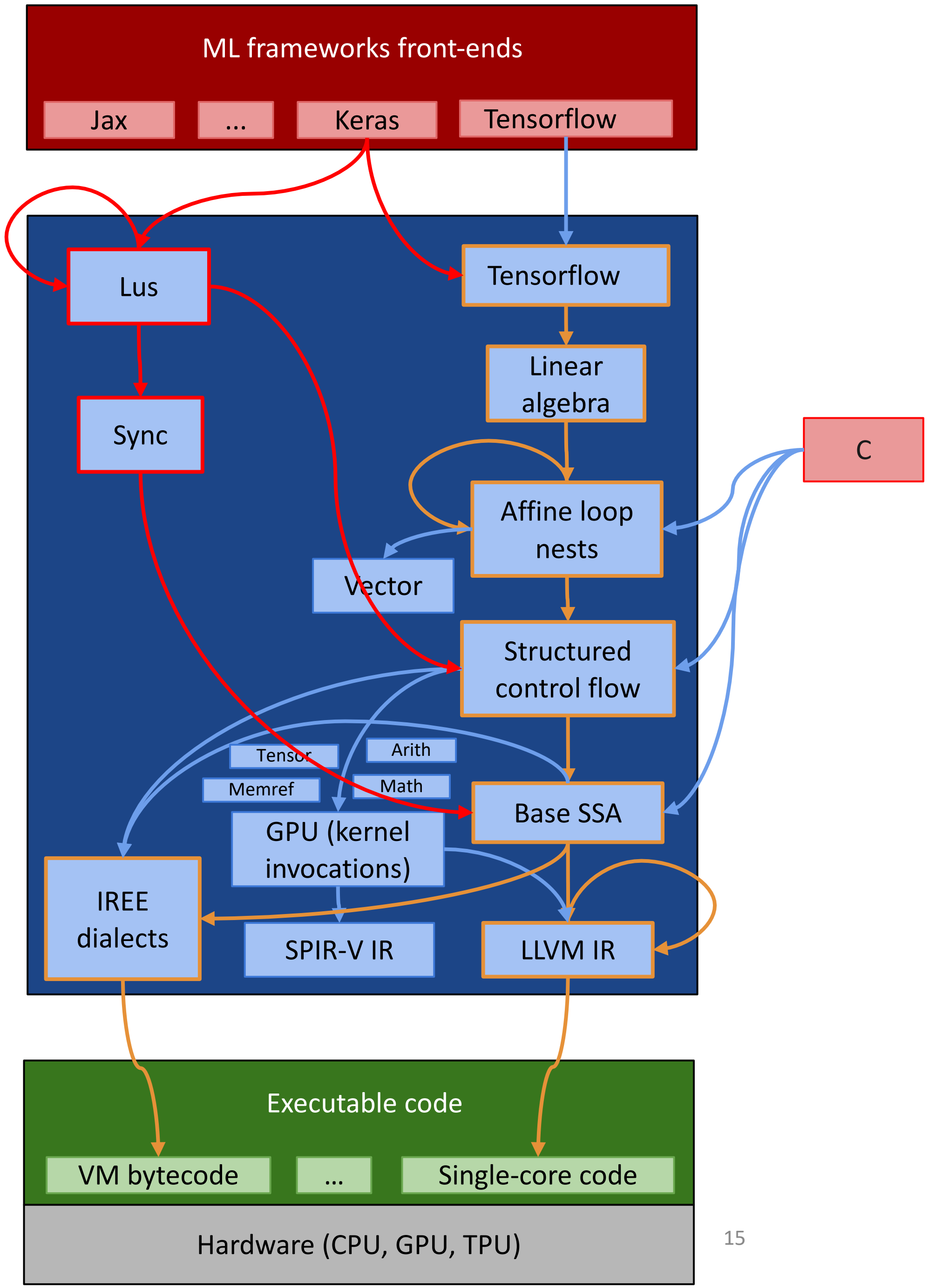
Extensions to MLIR

- Two MLIR dialects
  - lus = dataflow dialect (6 ops, including yield)
    - New programming paradigm
  - sync = low-level reactive dialect (7 ops, 2 types)
    - MLIR/SSA extension – composes with any control flow
  - New passes
    - lus clock analysis
      - Ensures single assignment in the presence of predication
    - lus normalization
    - lus lowering to sync
    - sync lowering to standard dialects
    - keras lowering to lus+tf
    - some sync structural verifications
  - Lots of reuse: causality, bufferization, optimizations, all other lowering…



Extensions to MLIR

- We explored two targets
  - In both cases, little investment in dedicated code generation
    - Modular code generation
    - Good speed
  - V1: Modular execution over single-core
    - One coroutine per reactive component
    - Custom compilation pipeline
  - V2: Classical synchronous compilation (non-modular execution) on iree (GPU or CPU)
    - Early move from sync to standard dialects
    - Standard iree compilation pipeline



Technical focus – building a reactive compiler

- Static Single Assignment (SSA)
  - Introduction and limitations
  - **Contribution 1: Reactive SSA – low-level reactive dialect**
    - Intuition: Vulkan-level, but semantically tied to both SSA and reactive programming
- Incorporating dataflow synchrony into MLIR
  - The Lustre dataflow synchronous language
  - **Contribution 2: Embedding of Lustre in MLIR – the high-level dataflow dialect**
- Experimental results
  - Expressiveness: Joint specification and compilation of high-performance (including ML) embedded applications
  - Performance: No performance loss w.r.t. traditional ML compilation
  - Non-intrusiveness: Potential coexistence with mainstream ML compilation evolution
- Conclusion

SSA - Static Single Assignment

```
c = 0;  y = 0;
while(1) {
  x = read_f32();
  if (c != 0) y = f(x);
  write_f32(y);
  c = (c + 1)%2; }
```

- SSA principle =
  - **[Single Assignment]** A variable is assigned by exactly one operation
  - **[Causality]** A variable is assigned before use
- SSA formalism (*SSA book@Springer, also at* github.com/pfalcon/ssabook)
  - Implementation of the SSA principle
  - IR for compilers: access to a wide variety of optimizations
- MLIR SSA – continuation-passing style (CPS)
  - Textual form

```
func @myfun() {
^bb0:
  %c1 = constant 0: i32
  %y1 = constant 0.0: f32
  br ^bb1(%c1, %y1: i32, f32)

^bb1(%c2: i32, %y2: f32)
  %x = call @read_f32():()->(f32)
  %ck = cmpi "neq",%c1,%c2: i32
  cond_br %ck,^bb2,^bb3(%y2:i32)

^bb2:
  %y3 = call @f(x): f32 -> f32
  br ^bb3(%y3: f32)

^bb3(%y4: f32)
  call @write_f32(%y4): f32 -> ()
  %1 = constant 1: i32
  %2 = constant 2: i32
  %3 = addi %c2, %1: i32
  %c3 = remi_signed %3, %2: i32
  br ^bb1(%c3, %y4: i32, f32)
}
```

SSA - limitations

```
c = 0;  y = 0;
while(1) {
  x = read_f32();
  if (c != 0) y = f(x);
  write_f32(y);
  c = (c + 1)%2; }
```

- Cyclic behaviours possible, but
  - No cyclic I/O at high abstraction level
    - Low-level encodings, no semantics
  - No concurrently running (communicating) functions
    - Nor execution environment
  - No synchronization between functions/environment
    - In particular, scheduling of operations (e.g. I/O functions) into cycles can be changed by SSA code transformations
  - Undefinedness/absence needs better support
    - E.g. output of a non-blocking data reception when no data is available
    - llvm.undef/llvm.poison may not be what you want
      - e.g. immediate undefined behavior upon use

```
func @myfun() {
^bb0:
  %c1 = constant 0: i32
  %y1 = constant 0.0: f32
  br ^bb1(%c1, %y1: i32, f32)

^bb1(%c2: i32, %y2: f32)
  %x = call @read_f32():()->(f32)
  %ck = cmpi “neq”,%c1,%c2: i32
  cond_br %ck,^bb2,^bb3(%y2:i32)

^bb2:
  %y3 = call @f(x): f32 -> f32
  br ^bb3(%y3: f32)

^bb3(%y4: f32)
  call @write_f32(%y4): f32 -> ()
  %1 = constant 1: i32
  %2 = constant 2: i32
  %3 = addi %c2, %1: i32
  %c3 = remi_signed %3, %2: i32
  br ^bb1(%c3, %y4: i32, f32)
}
```

Contribution 1: Reactive SSA (1/2)

- Concurrent design pattern (« collective operations ») ensuring determinism
  - Implements the execution model and causality of synchronous languages
  - Other implementations are possible (BSP, multi-periodic task systems, c11 subsets…)
- Conservative extension of SSA for reactive systems
  - **Concurrent stateful reactive functions** exchanging data and control
    - True concurrency between non-dependent operations of a basic block
  - Execution of each function divided into **non-overlapping cycles**
    - Cycle separator = **tick** operation
    - Once a cycle starts it completes without external interference (**atomicity**)
    - Trigger a cycle in another component: **inst** operation = **synchronous call**
      - Provide inputs -> context to the triggered cycle
      - Get outputs -> produced by the triggered cycle
      - Truly concurrent **inst** operations => true concurrency between function ticks
  - Cyclic I/O: **I/O channel types**, **input** and **output** operations
  - Explicit manipulation of absence: **sync.undef** operation

Contribution 1: Reactive SSA (2/2)

- Conservative extension of SSA for reactive systems
  - Syntactic extension of SSA: sync.func, sync.tick, sync.inst, sync.input, sync.output, sync.undef, sync.sync
  - Formal semantics extending the existing SSA semantics
    - No modifications to old rules
    - Add concurrent execution state
  - Smooth integration with traditional SSA compilation
    - **Reactive semantics is not broken by correct SSA code transformations**

Reactive SSA example

- Cycle barrier : sync.tick
  - Breaks execution into cycles
  - Assignment of each operation to its cycle
  - Synchronization: gives back control until the next cycle
- Cyclic I/O
  - I/O signals + I/O operations
  - Communication with calling function
    - For the root function, communication with the environment
  - Possible implementations: function calls, shared memory…

```
sync.func @myfun(%xs:sync.in<f32>)
        ->(%ys:sync.out<f32>) {
^bb0:
  %c1 = constant 0: i32
  %y1 = constant 0.0: f32
  br ^bb1(%c1, %y1: i32, f32)

^bb1(%c2: i32, %y2: f32)
  %x  = sync.input(%xs):f32
  %ck = cmpi "neq",%c1,%c2: i32
  cond_br %ck,^bb2,^bb3(%y2:i32)

^bb2:
  %y3 = sync.inst 2 @sum(%x):f32->f32
  br ^bb3(%y3: f32)

^bb3(%y4: f32)
  %u0 = sync.output(%ys,%y4):unit
  %1  = constant 1: i32
  %2  = constant 2: i32
  %3  = addi %c2, %1: i32
  %c3 = remi_signed %3, %2: i32
  %u1 = sync.tick(%u0,%c3):unit
  %c4 = sync.sync(%u1,%c3):i32
  br ^bb1(%c3, %y4: i32, f32)
}
```

Reactive SSA example

- Reactive modularity
  - Reactive functions
    - Concurrent automata
    - Internal state – SSA variables
  - inst : trigger one tick of another reactive function

```
sync.func @sum(%i:sync.in<f32>)->(%o:sync.out<f32>) {
^bb0:
  %0 = constant 0: f32
  br ^bb1(%0:f32)
^bb1(%s:f32)
  %x  = sync.input(%i):f32
  %s1 = arith.addf %x,%s: f32
  %u  = sync.output(%s1):unit
  %u1 = sync.tick(%u):unit
  %s2 = sync.sync(%u1,%s1):f32
  br ^bb1(%s2:f32)
}
```

```
sync.func @myfun(%xs:sync.in<f32>)
         ->(%ys:sync.out<f32>) {
^bb0:
  %c1 = constant 0: i32
  %y1 = constant 0.0: f32
  br ^bb1(%c1, %y1: i32, f32)

^bb1(%c2: i32, %y2: f32)
  %x  = sync.input(%xs):f32
  %ck = cmpi “neq”,%c1,%c2: i32
  cond_br %ck,^bb2,^bb3(%y2:i32)

^bb2:
  %y3 = sync.inst 2 @sum(%x):f32->f32
  br ^bb3(%y3: f32)

^bb3(%y4: f32)
  %u0 = sync.output(%ys,%y4):unit
  %1  = constant 1: i32
  %2  = constant 2: i32
  %3  = addi %c2, %1: i32
  %c3 = remi_signed %3, %2: i32
  %u1 = sync.tick(%u0,%c3):unit
  %c4 = sync.sync(%u1,%c3):i32
  br ^bb1(%c3, %y4: i32, f32)
}
```

Synchronous SSA example

- Lowering sync dialect produces functions calling API primitives
  - Example later
  - sync = lowest dialect with concurrent semantics
- **Not a good level for specification**

```
sync.func @sum(%i:sync.in<f32>)->(%o:sync.out<f32>) {
^bb0:
  %0 = constant 0: f32
  br ^bb1(%0:f32)
^bb1(%s:f32)
  %x  = sync.input(%i):f32
  %s1 = arith.addf %x,%s: f32
  %u  = sync.output(%s1):unit
  %u1 = sync.tick(%u):unit
  %s2 = sync.sync(%u1,%s1):f32
  br ^bb1(%s2:f32)
}
```

```
sync.func @myfun(%xs:sync.in<f32>)
         ->(%ys:sync.out<f32>) {
^bb0:
  %c1 = constant 0: i32
  %y1 = constant 0.0: f32
  br ^bb1(%c1, %y1: i32, f32)

^bb1(%c2: i32, %y2: f32)
  %x  = sync.input(%xs):f32
  %ck = cmpi “neq”,%c1,%c2: i32
  cond_br %ck,^bb2,^bb3(%y2:i32)

^bb2:
  %y3 = sync.inst 2 @sum(%x):f32->f32
  br ^bb3(%y3: f32)

^bb3(%y4: f32)
  %u0 = sync.output(%ys,%y4):unit
  %1  = constant 1: i32
  %2  = constant 2: i32
  %3  = addi %c2, %1: i32
  %c3 = remi_signed %3, %2: i32
  %u1 = sync.tick(%u0,%c3):unit
  %c4 = sync.sync(%u1,%c3):i32
  br ^bb1(%c3, %y4: i32, f32)
}
```

Lustre : a dataflow synchronous language

[POPL'87]

- Dataflow yes, but why Lustre?
  - Simple, concurrent&deterministic semantics
  - Proximity points to both Keras-like dataflow (cf. intro) and SSA form
    - Instance of the SSA principle
    - Globally Sequential, Locally Concurrent
  - Natural modeling of all ML applications we worked with
    - RNNs, gated, even RL…
  - Extensive work on code generation for reactive and embedded targets
    - Concurrent implementations of multiple flavors
    - Static memory allocation
    - Resource allocation…

… (and we have extensive experience with it)

Lustre : a dataflow synchronous language

[POPL'87]

```
c = 0;
y = 0;
while(1) {
  x = read_f32();
  if (c != 0) y = f(x);
  write_f32(y);
  c = (c + 1)%2; }
```

- Cyclic execution model
  - **Sequence of execution cycles**
  - Cycle = read input, compute, write output
    - **Cyclic I/O**
- Dataflow language
  - Computation driven by data
  - A var can be absent in a cycle (predicate/gate in dataflow)
    - Absent = not computed and not used
    - Sub-sampling : **when**
    - Combine variables that are never both present : **merge**
- Synchronous language
  - Variables are not persistent - their lifetimes end at the end of the current cycle
    - fby = explicit passing of values from one cycle to the next (where the variable is alive)
    - **Recovering persistency requires copying the old value (like in SSA)**

```
node mynode(x:float)
                returns (y:float)
var c:int; ck:bool;
    xx, fx, y: float;
let
  c = 0 fby ((c+1) % 2);
  ck = (c<>0);
  xx = x when ck;
  fx = f(xx);
  y = 0.0 fby
      (merge ck fx (y whenot ck));
tel
```

Lustre vs SSA formalism – the intuition

```
c = 0;
y = 0;
while(1) {
  x = read_f32();
  if (c != 0) y = f(x);
  write_f32(y);
  c = (c + 1)%2; }
```

- Similarities
  - Both instances of the SSA principle
  - Globally Sequential, Locally Concurrent
  - fby operations ~ loop-carried dependencies
  - Merge ops ~ phi operations of SSA
  - Lustre node ~ SSA spec with single basic block
- Differences
  - Cyclic I/O
  - Each operation is assigned to a cycle
    - Form of high-level scheduling
  - Predicated operations (*à la* predicated SSA)
    - Variables can be undefined in a cycle
      - Clock analysis -> ensure undef vars are not used
    - Including on fby operations
  - Cyclic dependencies -> dominance *a priori* not respected

```
node mynode(x:float)
               returns (y:float)
var c:int; ck:bool;
    xx, fx, y: float;
let
  c = 0 fby ((c+1) % 2);
  ck = (c<>0);
  xx = x when ck;
  fx = f(xx);
  y = 0.0 fby
      (merge ck fx (y whenot ck));
tel
```

Challenge 1: incorporate synchronous absence into SSA

```
y = x when ck;//clk(x)=clk(ck); clk(y)=clk(ck)&ck
z = f(y);     //clk(z)=clk(y)
u = g(x when ck,z);//clk(u)=clk(z)=clk(ck)&ck
```

- Absence : central concept in dataflow synchronous programming
- Computation triggered by arriving data
  - Conditional execution = conditional transmission of data (“when” operation)
- Synchrony : each variable is either present or absent in each cycle
  - **Correctness : absent values are never used in computations (-> SSA principle)**
  - Checking correctness : clock calculus (different from dominance analysis)
    - Determine the presence/absence condition for each variable
      - Clk(x) = predicate that is true in cycles where x is present, false in other cycles
    - System of equations over these predicates
    - Low-complexity calculus, part of the language semantics

Challenge 1: incorporate synchronous absence into SSA

```
y = x when ck;//clk(x)=clk(ck); clk(y)=clk(ck)&ck
z = f(y);     //clk(z)=clk(y)
u = g(x         ,z);//clk(u)=clk(z)=clk(x)
```

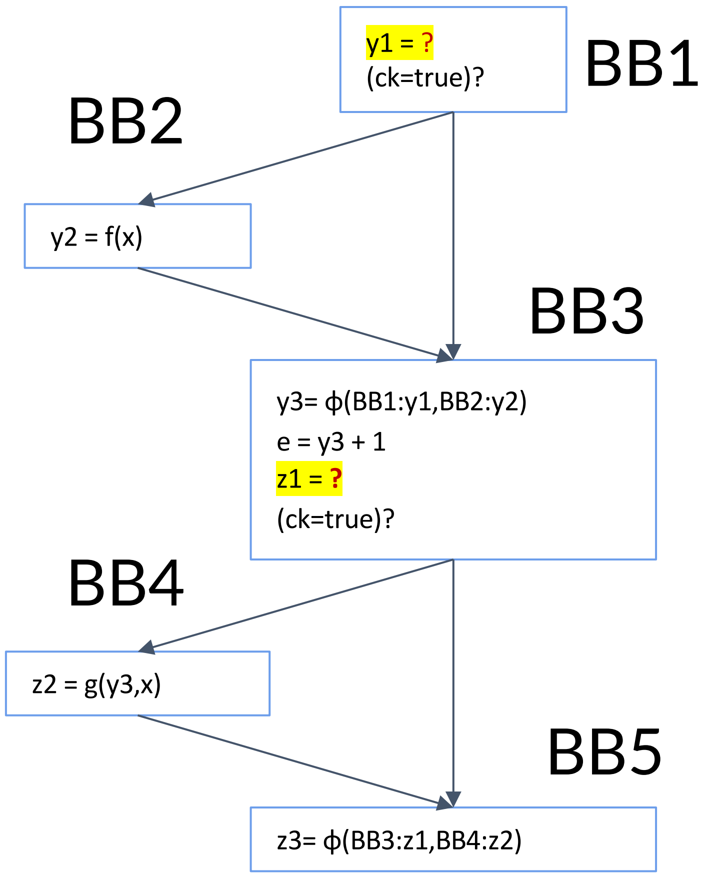
rejected

- Absence : central concept in dataflow synchronous programming
- Computation triggered by arriving data
  - Conditional execution = conditional transmission of data ("when" operation)
- Synchrony : each variable is either present or absent in each cycle
  - **Correctness : absent values are never used in computations (-> SSA principle)**
  - Checking correctness : clock calculus (different from dominance analysis)
    - Determine the presence/absence condition for each variable
      - Clk(x) = predicate that is true in cycles where x is present, false in other cycles
    - System of equations over these predicates
    - Low-complexity calculus, part of the language semantics

```
if(ck) y = f(x);//y undefined in cycles where ck=f
e = y+1;         //e undef/poison when ck=false
if(ck) z = g(y);//y unused when undefined
```

Challenge 1: incorporate synchronous absence into SSA

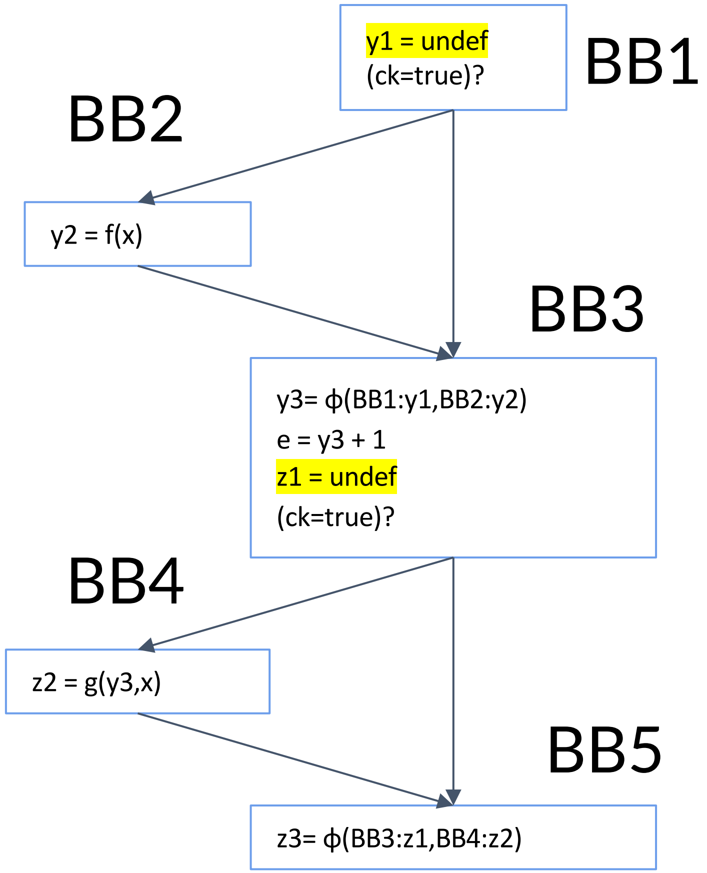
- Same problem exists when converting C to SSA
- Dominance rule => need a value for y even when it is not initialized



Challenge 1: incorporate synchronous absence into SSA

```
if(ck) y = f(x);//y undefined in cycles where ck=f
e = y+1;        //e undef/poison when ck=false
if(ck) z = g(y);//y unused when undefined
```

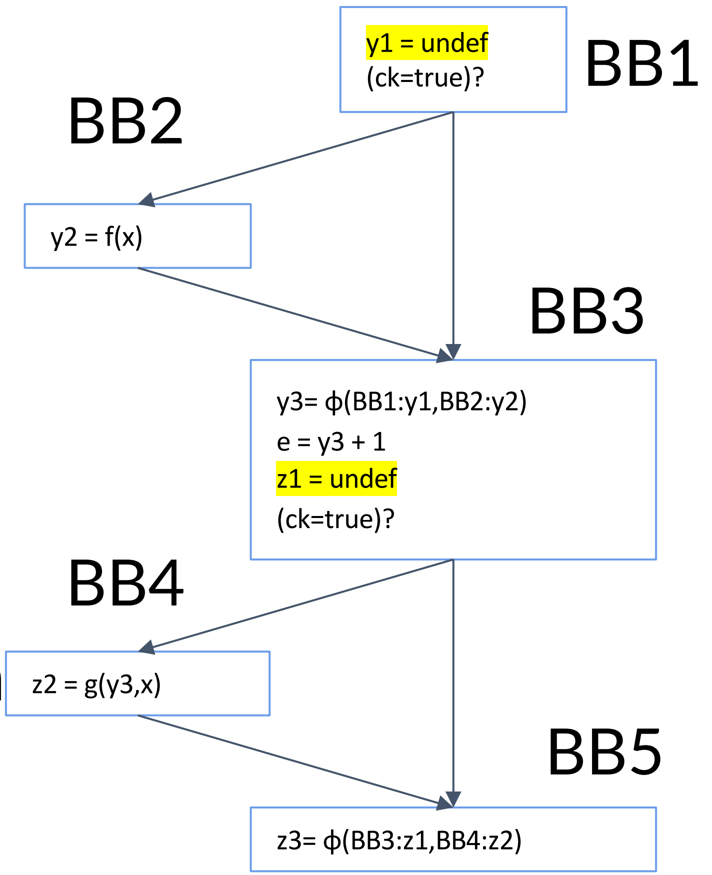
- Same problem exists when converting C to SSA
- Dominance rule => need a value for y even when it is not initialized
  - LLVM -> undefined values (undef, poison)
  - These values can still be used in computations
  - C compilers aim to preserve or refine undefined behaviors



```
if(ck) y = f(x);//y undefined in cycles where ck=f
e = y+1;        //e undef/poison when ck=false
if(ck) z = g(y);//y unused when undefined
```

Challenge 1: incorporate synchronous absence into SSA

- Same problem exists when converting C to SSA
- Dominance rule => need a value for y even when it is not initialized
  - LLVM -> undefined values (undef, poison)
  - These values can still be used in computations
  - C compilers aim to preserve or refine undefined behaviors
- **Lustre/synchronous: more restrictive approach**
  - Undefined values must never be used in computations or tests
    - **No need for complex undefinedness semantics at high level**



Challenge 1: incorporate synchronous absence into SSA

- Theorem [Compilation of sync.undef]

  Given a correct synchronous specification (where sync.undef values are never used), sync.undef values can be lowered to any lower-level SSA value

  - llvm.undef, llvm.poison, constant, malloc without initialization…

Challenge 1: incorporate synchronous absence into SSA

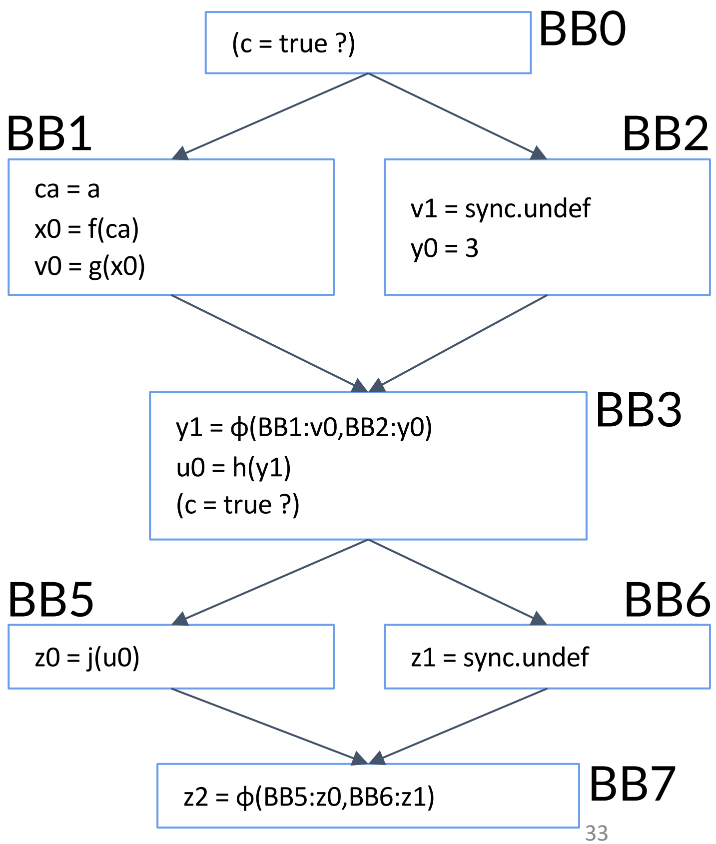
```
x = f(a when c)
v = g(x)
y = merge c v 3;
u = h(y);
z = j(u when c);
```

- Theorem [Compilation of sync.undef]

  Given a correct synchronous specification (where sync.undef values are never used), sync.undef values can be lowered to any lower-level SSA value

  - llvm.undef, llvm.poison, constant, malloc without initialization…

- lus -> sync lowering
  - Clock analysis : ensure that absent values are never used
  - Lustre absence : lowered to sync.undef + SSA branching/merging



```
x = f(a when c)
v = g(x)
y = merge c v 3;
u = h(y);
z = j(u when c);
```

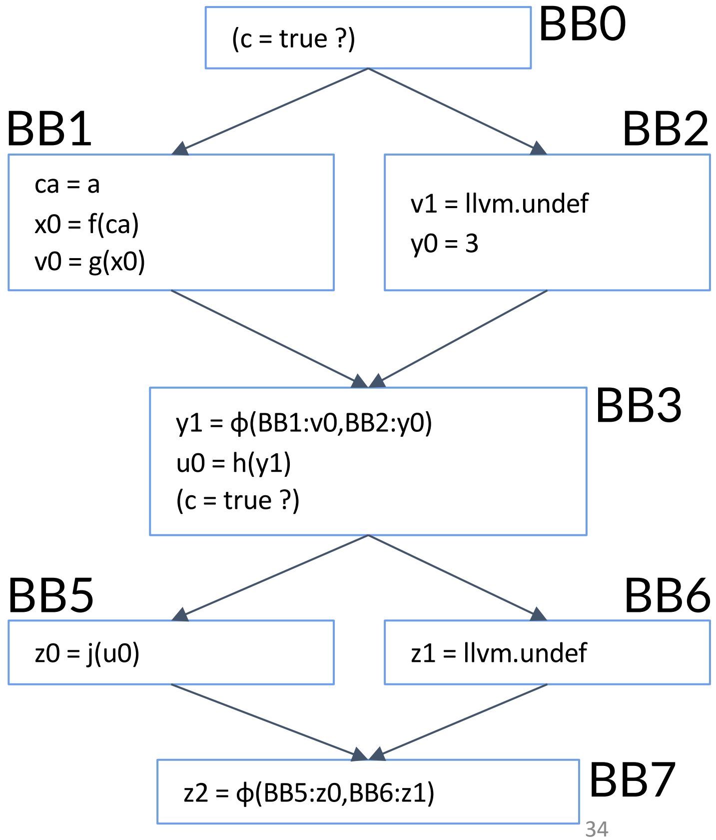
Challenge 1: incorporate synchronous absence into SSA

- Theorem [Compilation of sync.undef]

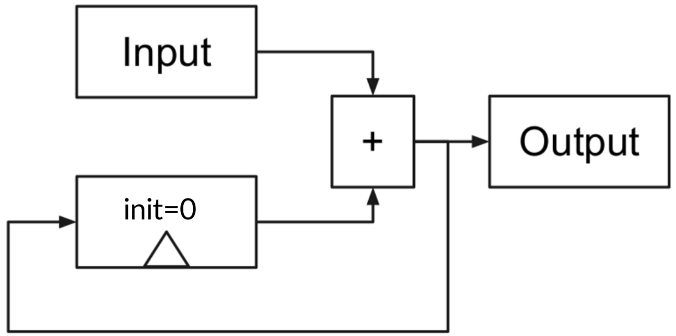
  Given a correct synchronous specification (where sync.undef values are never used), sync.undef values can be lowered to any lower-level SSA value

    - llvm.undef, llvm.poison, constant, malloc without initialization…

- lus -> sync lowering
  - Clock analysis : ensure that absent values are never used
  - Lustre absence : lowered to sync.undef + SSA branching/merging
    - And then to any value (cf. theorem)

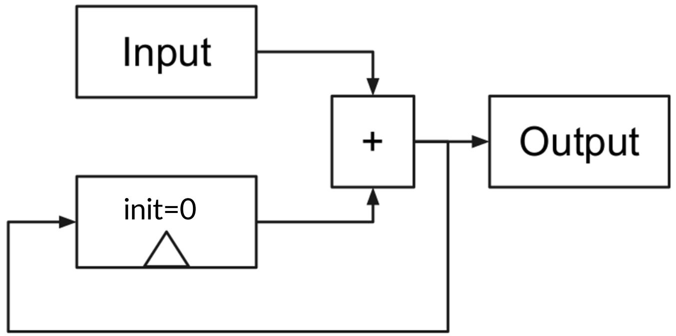


Challenge 2: the internal state



- Exemple: an integrator
  - Sums its input with the output of precedent cycles (init = 0)
  - Outputs the resulting value

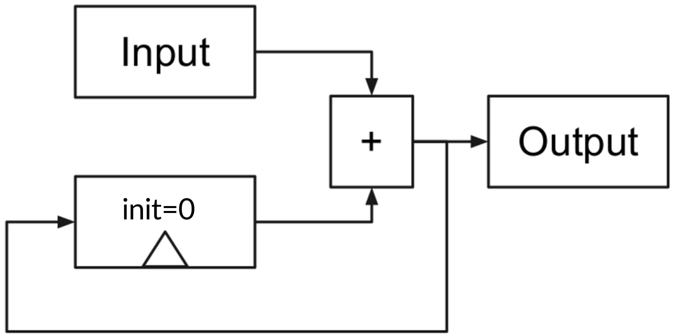
Challenge 2: the internal state



- Exemple: an integrator
  - Sums its input with the output of precedent cycles (init = 0)
  - Outputs the resulting value
- Natural reactive representation
  - Lustre & TensorFlow primitives
  - Dominance is not respected
    - MLIR relaxed dominance

```
lus.node @integr(%i: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```

Challenge 2: the internal state



- Exemple: an integrator
  - Sums its input with the output of precedent cycles (init = 0)
  - Outputs the resulting value
- Natural reactive representation
  - Lustre & TensorFlow primitives
  - Dominance is not respected
    - MLIR relaxed dominance
    - **Normalization**

```
lus.node @integr(%i: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```

```
lus.node @integr(%i: tensor<i32>)
                state(%os: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %f = lus.kperiodic 1(0)
  %s = lus.merge %f %c0 %os: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
       state(%incr:tensor<i32>)
}
```

Normal form:
- All fby operations are executed at each cycle
- Transform all fby operations as loop carried dependencies (in node signature + yield operation)

Challenge 2: the internal state

- lus->sync lowering
  - Traditional (control inversion)
    - Single reactive function (driver) for the whole application (tick, cyclic I/O)
    - Step/reset functions operating on global state representation
  - One reactive function per node
    - Trigger reactions in sub-nodes using **inst**
    - Local node state

```
lus.node @integr(%i: tensor<i32>)
                 ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```

```
sync.func @integr(%is: !sync.in<tensor<i32>>)
               ->(%os: !sync.out<tensor<i32>>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>

  %true = constant 1: i1
  scf.while(%state = %c0):(tensor<i32>) {
    scf.condition(%true)
  } do {



    %i = sync.input(%is): tensor<i32>
    %incr = tf.Add(%state, %i): tensor<i32>
    %sy1 = sync.output(%os: %incr): tensor<i32>
    %sy2 = sync.tick(%sy1)
    %nstate = sync.sync(%sy2,%incr): tensor<i32>
    scf.yield %nstate: tensor<i32>
  }
  sync.halt
}
```

Challenge 2: the internal state

- One reactive function per node
  - Explicit main loop
  - Internal state = loop-carried deps

```
sync.func @integr(%is: !sync.in<tensor<i32>>)
                ->(%os: !sync.out<tensor<i32>>)    {
  %c0 = tf.Const{dense<0>}: tensor<i32>

  %true = constant 1: i1
  scf.while(%state = %c0):(tensor<i32>) {
    scf.condition(%true)
  } do {


    %i = sync.input(%is): tensor<i32>
    %incr = tf.Add(%state, %i): tensor<i32>
    %sy1 = sync.output(%os: %incr): tensor<i32>
    %sy2 = sync.tick(%sy1)
    %nstate = sync.sync(%sy2,%incr): tensor<i32>
    scf.yield %nstate: tensor<i32>
  }
  sync.halt
}
```

```
lus.node @integr(%i: tensor<i32>)
               ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```

Challenge 2: the internal state

- One reactive function per node
  - Explicit main loop
  - Internal state = loop-carried deps
  - sync dialect lowering:
    - buffering
    - I/O, tick = runtime API calls

```
lus.node @integr(%i: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```

```
func @integr(%inst:i32,%is:(i32,memref<i32>)->(),
                       %os:(i32,memref<i32>)->()){
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %p = constant 1 : i32
  %true = constant 1: i1
  scf.while(%state = %c0):(tensor<i32>) {
    scf.condition(%true)
  } do {
    %mi = memref.alloc() : memref<i32>
    call_indirect %is(%inst,%pos,%mi):
      (i32,memref<i32>)->()
    %i = memref.tensor_load %mi : memref<i32>
    %incr = tf.Add(%state, %i): tensor<i32>
    %mincr = memref.buffer_cast %incr : memref<i32
    call_indirect %os(%p,%mincr):(i32, memref<i32>
    call @tick()
    scf.yield %mincr: tensor<i32>
  }
  return
}
```

Challenge 3: Modular execution

- Traditional: modular code generation, non-modular execution
- One reactive function per node
  - Control passing (through context switches) managed by executive

```
lus.node @test(%i: tensor<i32>)->() {
  %o = lus.instance @integr(%i)
    :(tensor<i32>) -> (tensor<i32>)
  call @print_i32(%o):(tensor<i32>)->(none)
  lus.yield()
}
```

```
sync.func @test(%is:!sync.sigin<tensor<i32>>)->(){

  %true = constant 1: i1
  scf.while:  () -> () { scf.condition(%true) } do {

    %i  = sync.input(%is): tensor<i32>



    %o  = sync.inst @integr 2 (%i): tensor<i32>
    call @print_i32(%o):(tensor<i32>)->(none)
    sync.tick()
    scf.yield
  }
  sync.halt
}
```

Challenge 3: Modular execution

- Run-time API calls
  - @sch_set_instance: declare new instance
  - @sch_set_io_X: set I/O buffers
  - @inst: give control to instance for one tick
  - @tick: give control back to caller (another instance or environment)

```
lus.node @test(%i: tensor<i32>)->() {
  %o = lus.instance @integr(%i)
    :(tensor<i32>) -> (tensor<i32>)
  call @print_i32(%o):(tensor<i32>)->(none)
  lus.yield()
}
```

```
func @test(%inst:i32, %is:(i32,memref<i32>)->()) {
  %f = constant @integr:(i32)->()
  call @sch_set_instance(%inst,%f) : (i32,(i32)->())->()
  %true = constant true
  scf.while : () -> () { scf.condition(%true) } do {
    %i = memref.alloc() : memref<i32>
    %pos = constant 0 : i32
    call %is(%pos,%mo):(i32,memref<i32>)->()
    %o = memref.alloc() : memref<i32>
    call @sch_set_io_I(%pos,%i):(i32,memref<i32>)->()
    call @sch_set_io_O(%pos, %o):(i32,memref<i32>)->()
    %inst2 = constant 2:i32
    call @inst(%inst2):(i32)->()
    call @print_i32(%o):(memref<i32>)->()
    call @tick():()->i32
    scf.yield
  }
  return
}
```

A reactive RNN

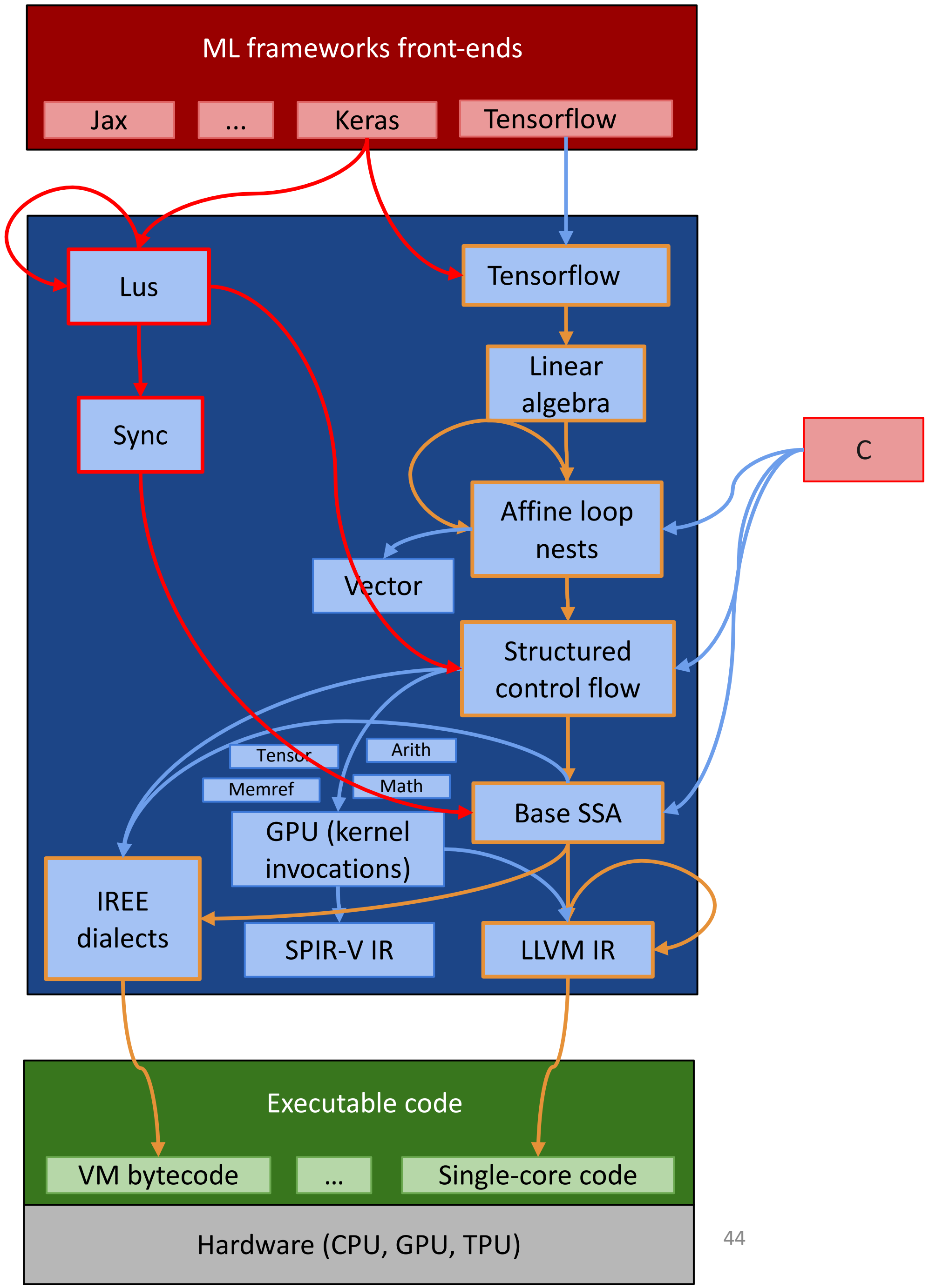
```
input = Keras.Input(shape=49,40)
x = layers.LSTM(units=4)(input)
x = layers.Dense(units=4)(x)
model = keras.Model(input,output)
model.load_weights('lstm_weights.h5')
```

```
lus.node @model(%x0:tensor<40xf32>)->(tensor<4xf32>) {
 %ck = lus.inst @true_every_49():i1
 %x1 = lus.inst @lstm(%res,%x0):tensor<4xf32>
 // output subsampling
 %x2 = lus.when %ck %x1: tensor<4xf32>
 %x3 = lus.inst @dense(%x2): tensor<4xf32>
 lus.yield (%x3: tensor<4xf32>)
}
```

```
lus.node @lstm(%data:tensor<40xf32>,%rst:i1)
                              -> (tensor<4xf32>) {
  // Feedback and reset control
  %c0   = tf.Const(){dense<…>}
  %tmp0 = lus.fby %c0 %s0o
  %24a  = lus.when     %rst %c0
  %24b  = lus.when not %rst %tmp0
  %24   = lus.merge    %rst %24a %24b
  %c1   = tf.Const(){dense<…>}
  %tmp1 = lus.fby %c1 %s1o
  %25a  = lus.when     %rst %c1
  %25b  = lus.when not %rst %tmp1
  %25   = lus.merge    %rst %25a %25b
  // LSTM computational core
  %v26 = tf.MatMul(%v24, %o76)
  %v28 = tf.MatMul(%data, %o22)
  %v29 = tf.AddV2(%v28, %v26)
  %v30 = tf.BiasAdd(%v29, %o78)
  %dim = tf.Const() {value = dense<1>}
  %v31_0, %v31_1, %v31_2, %v31_3
       = tf.Split(%dim, %v30)
  %v32 = tf.Relu(%v31_2)
  %v33 = tf.Sigmoid(%v31_0)
  %v34 = tf.Mul(%v33, %v32)
  %v35 = tf.Sigmoid(%v31_1)
  %v36 = tf.Mul(%v35, %v25)
  %s1o = tf.AddV2(%v36, %v34)
  %v40 = tf.Relu(%lstm_out)
  %v41 = tf.Sigmoid(%v31_3)
  %s0o = tf.Mul(%v41, %v40)
  // Output subsampling
  lus.yield (%s1o: tensor<3x1xf32>)
}
```

Experimental results (1/3)

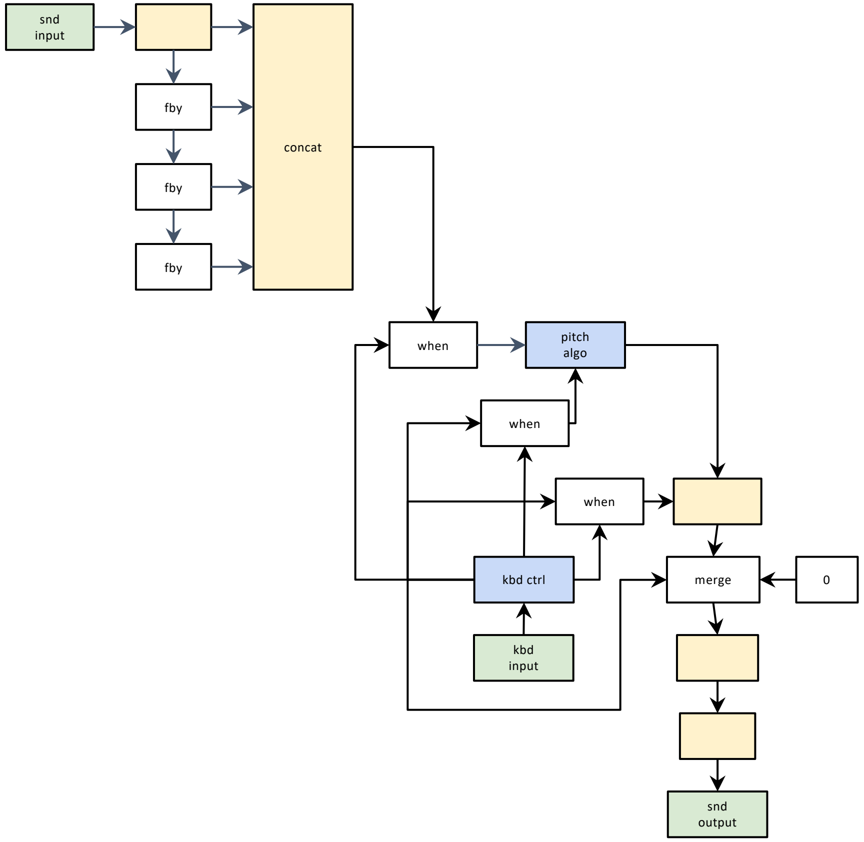
- **Non-intrusiveness** : high degree of MLIR code reuse
  - Need to write:
    - Clock analysis
    - Normalization
    - Synthesis of low-level control
  - Reuse: causality analysis, optimizations, code generation…



Experimental results (2/3)

- **Performance** : no pessimization due to reactive encoding
  - ML usecases (prediction phase) :
    - ResNet50 (K. He et al., CVPR ‘16)
    - LSTM-based RNN
  - Pipeline targetting a CPU towards the LLVM backend:
    - Modular execution
    - **RTE state of the art**: no performance loss w.r.t traditional Lustre compiler + gcc -03
  - Pipeline targetting the IREE VM (CPU, GPU):
    - Traditional code generator
    - **HPC state of the art**: no performance loss w.r.t IREE standard pipeline
    - (Widely more efficient than the previous approach)

Experimental results (3/3)



- **Expressiveness**: complex reactive control+HPC data handling
  - ML applications
    - Recurrence
    - Pre/post treatment of data (sliding windows, sub-sampling)
  - More complex reactive control
    - Pitch tuning vocoder (traditional RT signal processing application)
    - (Soft) real-time execution using MLIR

Current limitations = Ongoing work

- Training
  - Can represent its result, but not the training process itself (yet)
    - Back-propagation in RNNs
    - Our next paper
- Only describe activation, not task length
  - Good for specification and certain types of implementations
  - Can be extended to cover resource allocation durations
    - Long tasks
    - Integration with static resource allocation algorithms
- Time-space conversion – mapfold operation

Conclusion (1/3)

- First presentation of these works to the MLIR community
  - We needed to be confident
  - We need your feedback
  - We hope to contribute to MLIR

Conclusion (2/3)

- I hope we convinced you that MLIR needs a dataflow dialect
  - Natural: concurrent, stateful, predicated, hierarchic
    - RNNs, RL, transformers, sparsely-gated experts…
    - Front-end/back-end data pre-/post-preprocessing code
    - Streaming/embedded, modeling implementations (multiple interacting components, e.g. GPUs)
  - Why in MLIR
    - General-purpose specificaton (including all options, not just the DF core)
    - Refinement into particular implementations (under well-defined semantics)
      - Constant propagation
      - Time-space conversion
      - Synthesis of training code (back-propagation, forward-forward…)
    - Normalization, lowering (avoiding ad-hoc Python semantics/transformations)
  - Existing work on resource allocation specification to appl

- We propose that `lus` is a good minimalist DF dialect
  - Would like to work with you on perfecting it/upstreaming it

Conclusion (3/3)

- sync = low-level dialect for concurrent reactive systems (ABI+API)
  - Needed for reactive/embedded/multi-component implementation
  - Reactive SSA extension
    - Cyclic execution of components - tick (fixed allocation of operations into cycles)
    - Cyclic I/O - input/output
    - Synchronous calls – inst
  - Easy to implement, easy to compile DF Lustre into it, well-defined semantics
  - More mechanisms may be needed in particular cases
    - Concurrency restricted to one BB (and to synchronous calls of one BB)
    - Potential solutions: Asychronous calls, Predicated execution inside BBs
  - Compare Reactive SSA with Vulkan…
    - Clarify semantics of such implementations by using Reactive SSA as reference