Specification and compilation of embedded neural networks

Lustre, TensorFlow, and MLIR to rule them all

Hugo Pompougnac - PhD defense

Paris, 9 dec 2022

High performance & real-time embedded systems

- A real need
  - ML components in real-time systems (autonomous transportation, drones…)
  - Embedded digital twins (e.g. model-predictive control/maintenance)
  - More traditional signal/image processing systems (e.g. FFT…)

- Expertise distributed between two different fields
  - Real time embedded (RTE)
  - High performance computing (HPC) and in particular machine learning (ML) frameworks

- Our objective : natural specification and efficient compilation of combined RTE/HPC systems

RTE vs HPC/ML

High performance/ML

- **Optimization**
  - Average case (functional preservation)
  - Incremental lowering
- Classical/Transformational functions
  - One input, one result
- **Data parallelism** + imperative control
  - **Data structures (tensors), linear algebra…**
- ML frameworks
  - TensorFlow, Keras, JAX, PyTorch…
  - **MLIR/LLVM**: SSA-based representation

Real-time embedded

- **Correctness guarantees (functional/non-functional)**
  - Worst case analysis
  - Global optimization objectives
- **Reactive** (cyclic execution, often periodic)
  - Streams, functions on streams
- Mainly task parallelism/**dataflow**

- Reactive formalisms
  - Expressive DF languages : Simulink, Scade…
  - **Lustre**: dataflow synchronous kernel language

RTE vs HPC/ML

High performance/ML

- Optimization
  - Average case (functional preservation)
  - Incremental lowering
- Classical/Transformational functions
  - One input, one result
- **Data parallelism** + imperative control
  - **Data structures (tensors), linear algebra…**
- ML frameworks
  - TensorFlow, Keras, JAX, PyTorch…
  - **MLIR/LLVM**: SSA-based representation

Real-time embedded

- **Correctness guarantees (functional/non-functional)**
  - Worst case analysis
  - Global optimization objectives
- **Reactive** (cyclic execution, often periodic)
  - Streams, functions on streams
- Mainly task parallelism/**dataflow**
- Reactive formalisms
  - Expressive DF languages : Simulink, Scade…
  - **Lustre**: dataflow synchronous kernel language

Quasi-total disconnect between ML algorithmic research and embedded implementation

RTE vs HPC/ML

High performance/ML

- **Optimization**
  - Average case (functional preservation)
  - Incremental lowering
- Classical/Transformational functions
  - One input, one result
- **Data parallelism** + imperative control
  - **Data structures (tensors), linear algebra…**
- ML frameworks
  - TensorFlow, Keras, JAX, PyTorch…
  - **MLIR/LLVM**: SSA-based representation

Real-time embedded

- **Correctness guarantees (functional/non-functional)**
  - Worst case analysis
  - Global optimization objectives
- **Reactive** (cyclic execution, often periodic)
  - Streams, functions on streams
- Mainly task parallelism/**dataflow**
- Reactive formalisms
  - Expressive DF languages : Simulink, Scade…
  - **Lustre**: dataflow synchronous kernel language

Semantic and tooled embedding of Lustre into MLIR

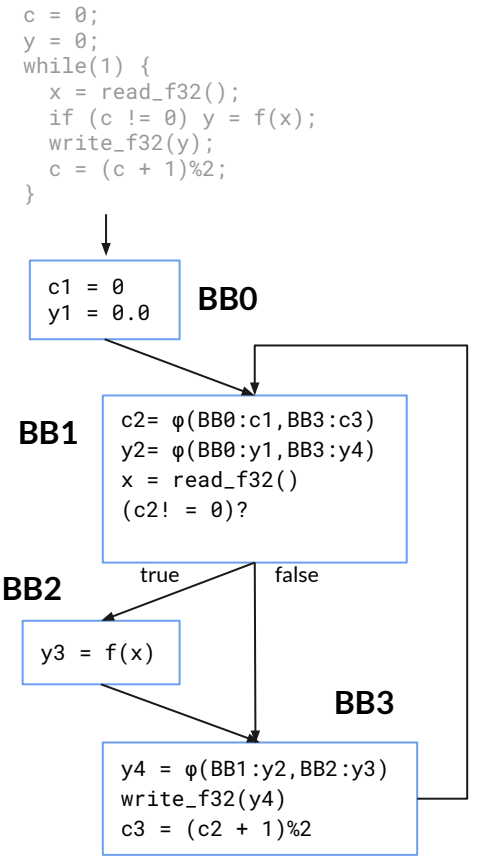
Joint SSA-based specification and compilation of HPC/ML, RTE aspects

Outline

- Static Single Assignment (SSA)
  - Introduction and limitations
  - **Contribution 1: Reactive SSA**
- MLIR - an SSA-based intermediate representation for ML/HPC compilation
- Incorporating dataflow synchrony into MLIR
  - The Lustre dataflow synchronous language
  - **Contribution 2: Embedding of Lustre in MLIR**
- 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

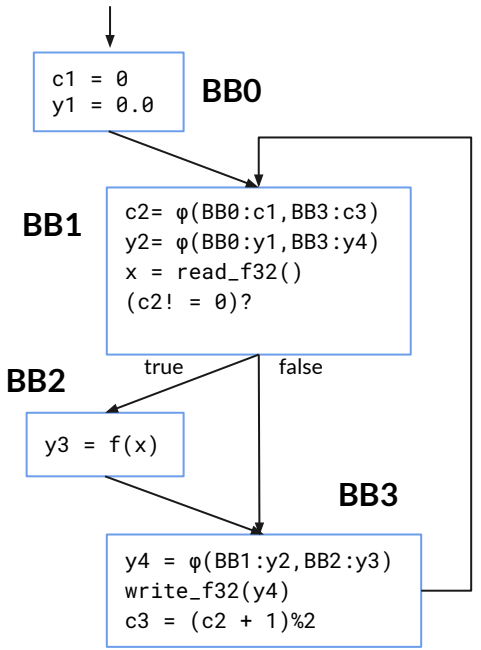
- SSA principle
  - **[Single Assignment]** A variable is assigned by exactly one operation
  - **[Causality]** A variable is assigned before use
- SSA formalism (*SSA book*, github.com/pfalcon/ssabook)
  - Implementation of the SSA principle
  - IR for compilers: access to a wide variety of optimizations
- Two-level representation (Globally Sequential, Locally concurrent)
  - **Sequential control flow graph** (SCFG) of **basic blocks** (BBs)
    - At the end of a BB, control is given to at most one BB
  - BB level : sequence of **operations** (topological order)



SSA - Static Single Assignment

- SSA principle
  - **[Single Assignment]** A variable is assigned by exactly one operation
  - **[Causality]** A variable is assigned before use
- SSA formalism (*SSA book*, github.com/pfalcon/ssabook)
  - Implementation of the SSA principle
  - IR for compilers: access to a wide variety of optimizations
- [Causality] ensured by a stronger property : dominance criterion
  - A variable used in an operation must be defined for every path leading to this operation (its definition dominates its uses)
  - Example: c3 does not dominate operations in BB1
- The φ operator
  - Propagates and multiplexes values from different sources/CF arcs
  - Deterministic : builds a unique value (depending on control)
  - Example: c2 takes the value of c1 or c3

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

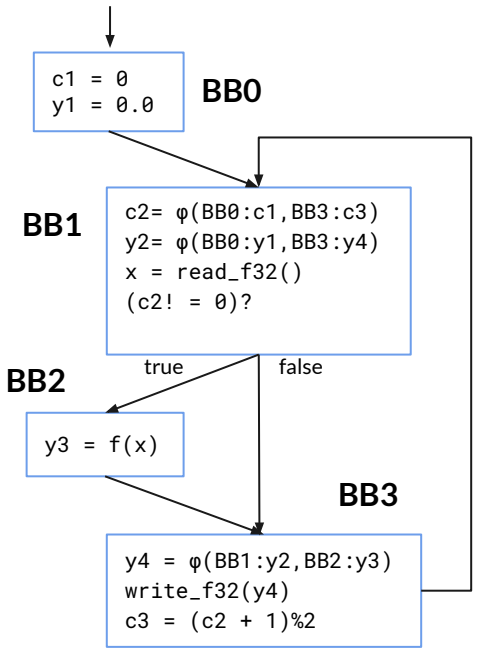


SSA (textual)

- MLIR textual form
- Continuation-passing style
  - φ operators output defined in BB entry point
  - Branching op will provide values for all φ in the destination BB

```
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)
}
```

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

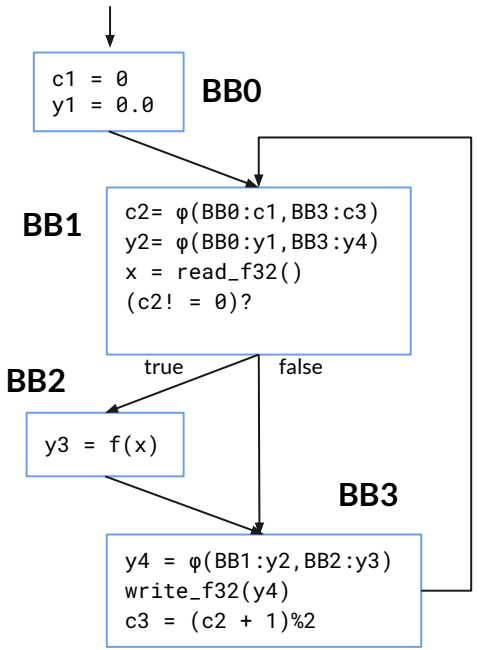


SSA limitations

- Cyclic behaviours possible, but
  - How to specify execution cycles/periodic synchronization with the environment ?
  - How to specify cyclic IOs at a good level of abstraction ?
  - How to specify concurrently running reactive functions (reactive modularity) ?
  - How to specify the undefinedness/absence of a variable ?

```
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)
}
```

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



Contribution 1: Reactive SSA

- Conservative extension of SSA for reactive systems
  - Syntactic extension of SSA (reactive primitives)
  - Formal semantics extending the existing SSA semantics
  - Smooth integration with traditional SSA compilation

Contribution 1: Reactive SSA

- Conservative extension of SSA for reactive systems
  - Syntactic extension of SSA (reactive primitives)
    - Concurrently-running **reactive functions** which can exchange data and control through **instance** operations
    - A **tick** operation separating execution cycles
    - Cyclic I/O: **I/O channel types**, **input** and **output** operations
    - Explicit manipulation of absence: **sync.undef** operation
  - Formal semantics extending the existing SSA semantics
  - Smooth integration with traditional SSA compilation

Contribution 1: Reactive SSA

- Conservative extension of SSA for reactive systems
  - Syntactic extension of SSA (reactive primitives)
    - Concurrently-running **reactive functions** which can exchange data and control through **instance** operations
    - A **tick** operation separating execution cycles
    - Cyclic I/O: **I/O channel types**, **input** and **output** operations
    - Explicit manipulation of absence: **sync.undef** operation
  - Formal semantics extending the existing SSA semantics
    - SOS semantics only adds rules, does not change existing ones
    - Concurrent execution context
      - State of multiple nodes, currently running node
    - Cyclic execution
  - Smooth integration with traditional SSA compilation

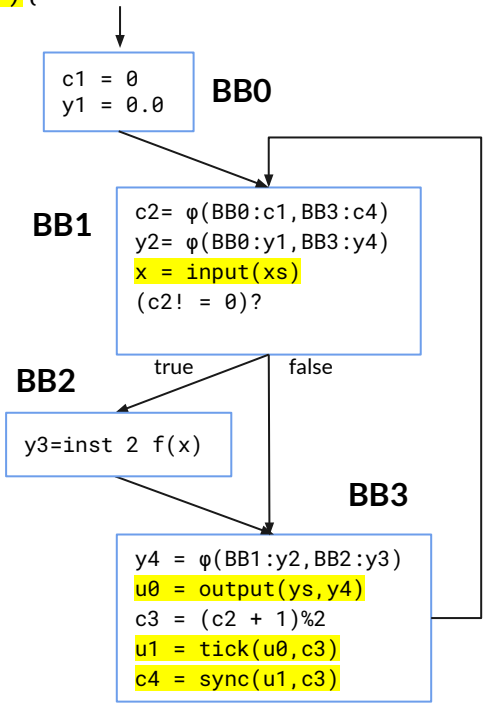
Contribution 1: Reactive SSA

- Conservative extension of SSA for reactive systems
  - Syntactic extension of SSA (reactive primitives)
    - Concurrently-running **reactive functions** which can exchange data and control through **instance** operations
    - A **tick** operation separating execution cycles
    - Cyclic I/O: **I/O channel types**, **input** and **output** operations
    - Explicit manipulation of absence: **sync.undef** operation
  - Formal semantics extending the existing SSA semantics
    - SOS semantics only adds rules, does not change existing ones
    - Concurrent execution context
      - State of multiple nodes, currently running node
    - Cyclic execution
  - Smooth integration with traditional SSA compilation
    - **Reactive semantics is not broken by correct SSA code transformations**

Reactive SSA example (1/2)

- Cycle barrier : tick
  - Breaks execution into cycles
  - Assignment of each operation to its cycle (dominance criterion)
  - Synchronization: gives back control until the next cycle
- Cyclic I/O
  - IO signals + r/w operations
  - Describe the communications (between reactive functions/with their environment)
  - Possible implementations: functions, shared memory

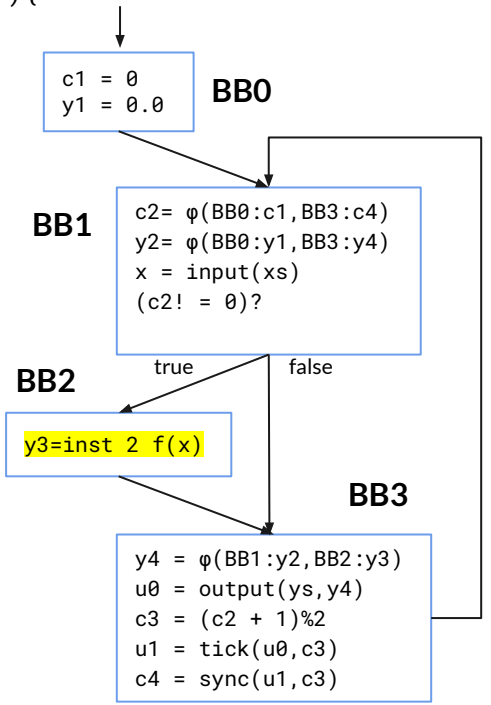
```
sync.func @mynode(%xs: sync.in<f32>)
                 ->(%ys: sync.out<f32>){
^bb0:
  %c1 = constant 0:i32
  %y1 = constant 0.0:f32
  br ^bb1(%c1,%y1:i32,f32)
^bb1(%c2:i32,%y2:f32)
  %x = sync.input(%xs):f32
  %ck = cmpi "neq",%c1,%c2:i32
  cond_br %ck,^bb2,^bb3(%y2:i32)
^bb2:
  %y3 = sync.inst 2 @f(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(%c4,%y4:i32,f32)
}
```



Reactive SSA example (2/2)

- Reactive modularity
  - Operations grouped into reactive functions
    - Concurrent automata
    - Cyclic IO
    - Internal state
  - inst : instantiation of a sub-automaton
    - Each time it is reached: gives control to the instanciated reactive function

```
sync.func @mynode(%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 @f(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(%c4,%y4:i32,f32)
}
```



MLIR : SSA + regions + dialects

- Pure SSA = low-level compiler IR
  - Good for LLVM-like middle-ends and back-ends
  - Not well suited when exploiting **higher-level/domain-specific** semantic information
    - Affine loop nests, linear algebra, machine learning, first-class vectors…
- MLIR (*MLIR website*, mlir.llvm.org)
  - **assembling domain-specific knowledge (dialects) around a common SSA-based core**
  - Two key extensions w.r.t. SSA: regions and dialects

MLIR : SSA + regions + dialects

- **Regions** enable hierarchic SSA specification
  - Region = local graph of basic blocks attached to a higher-level operation
  - Allows structured control. Examples:
    - the “then” and “else” branches of an “if” operation
    - the body of a “for” loop
    - the body of a function…
  - SSA constraints can be relaxed for a given region

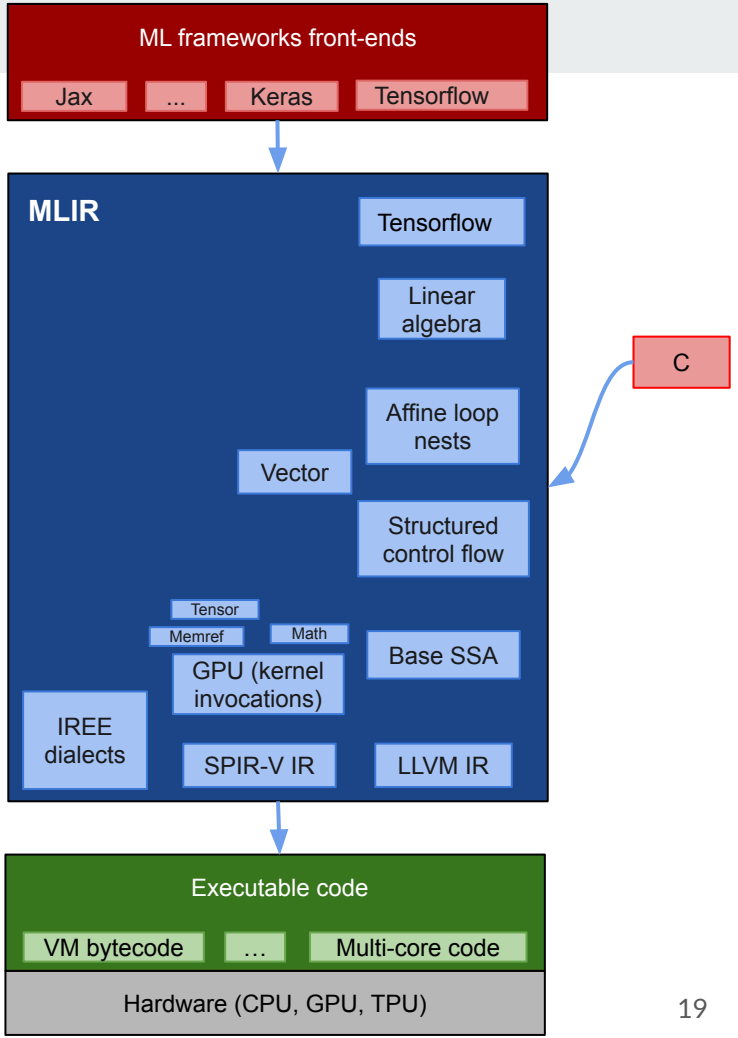
```
func @factorial(%a:i32) -> (i32)
{
  %lb = constant 2: i32
  %step = constant 1: i32
  %ro = constant 1: i32

  %r2 = scf.for %i = %lb
    to %a step %step
    iter_args(%r= %r0)->(i32)
  {
    %r1 = muli %r,%i: i32
    scf.yield %r1: i32
  }
  return %r2: i32
}
```

R1

R2

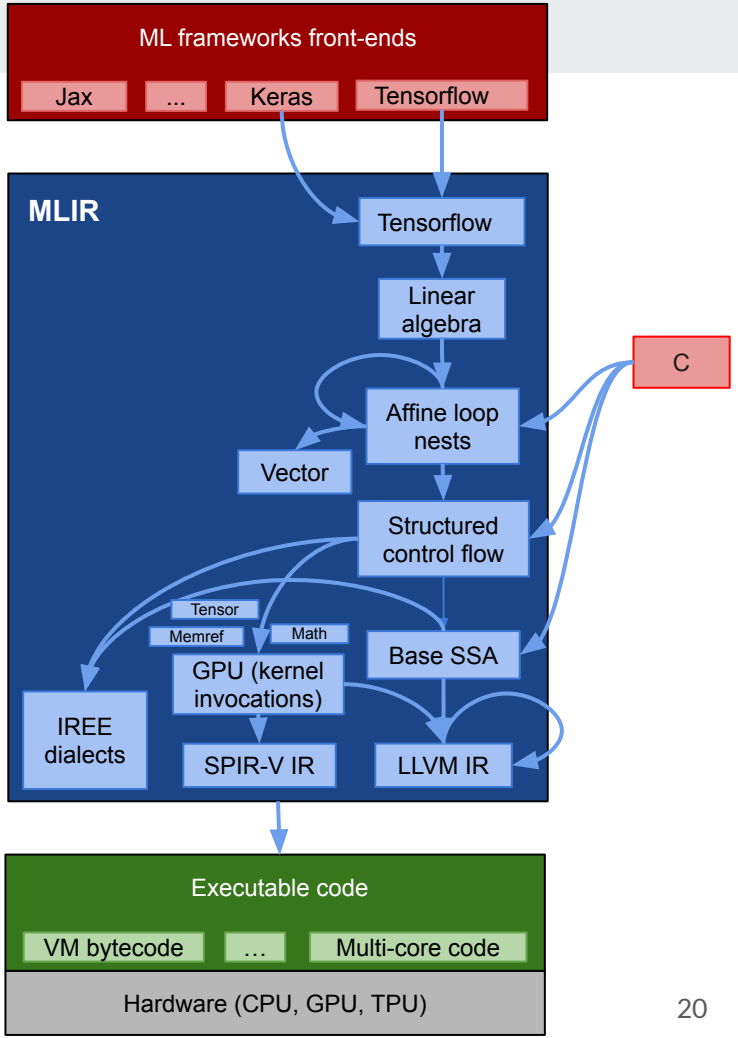
MLIR : SSA + regions + dialects

- **Dialects** group domain-specific knowledge
  - Extensibility point of MLIR
  - Dialect = datatypes, operations, algorithms of a domain
    - tf - TensorFlow ML operations
    - linalg - Linear algebra operations
    - affine - affine loop nests
    - llvmir - low-level IR of LLVM
  - Dialects can be mixed under SSA and domain-specific constraints



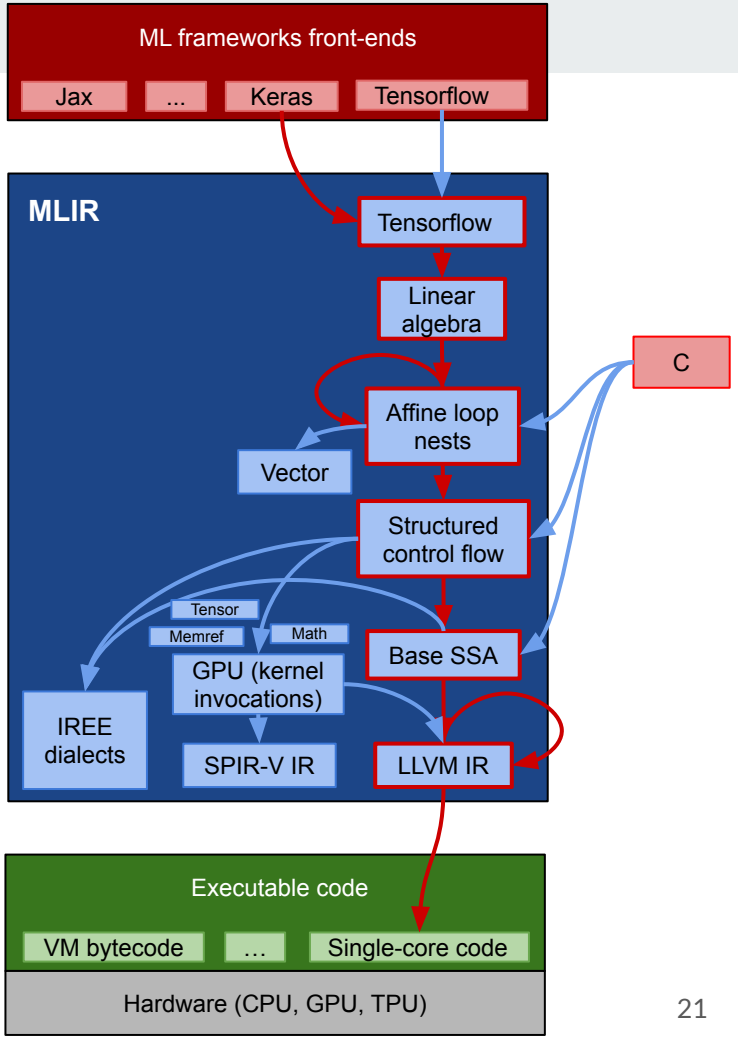
MLIR : SSA + regions + dialects

- **Dialects** group domain-specific knowledge
  - Extensibility point of MLIR
  - Dialect = datatypes, operations, algorithms of a domain
    - tf - TensorFlow ML operations
    - linalg - Linear algebra operations
    - affine - affine loop nests
    - llvmir - low-level IR of LLVM
  - Dialects can be mixed under SSA and domain-specific constraints
- Transformations
  - Lowering/optimization



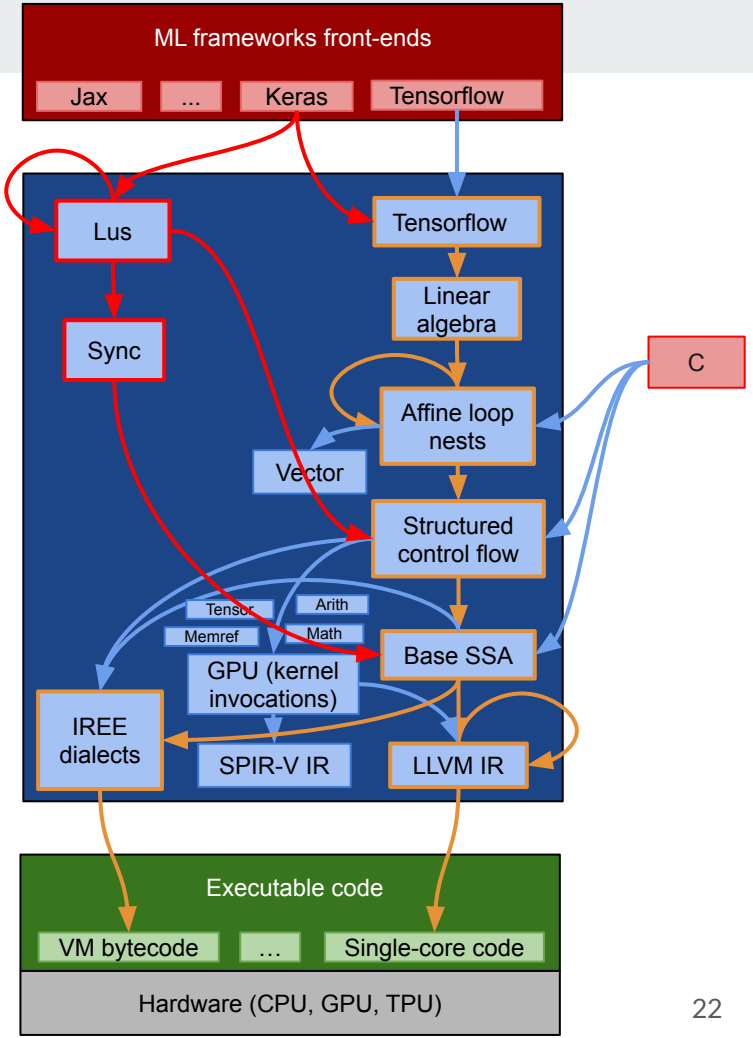
MLIR : SSA + regions + dialects

- **Dialects** group domain-specific knowledge
  - Extensibility point of MLIR
  - Dialect = datatypes, operations, algorithms of a domain
    - tf - TensorFlow ML operations
    - linalg - Linear algebra operations
    - affine - affine loop nests
    - llvmir - low-level IR of LLVM…
  - Dialects can be mixed under SSA and domain-specific constraints
- Transformations
  - Lowering/optimization
  - Compilation pipeline = sequence of transformations
  - Multiple input formalisms, multiple targets



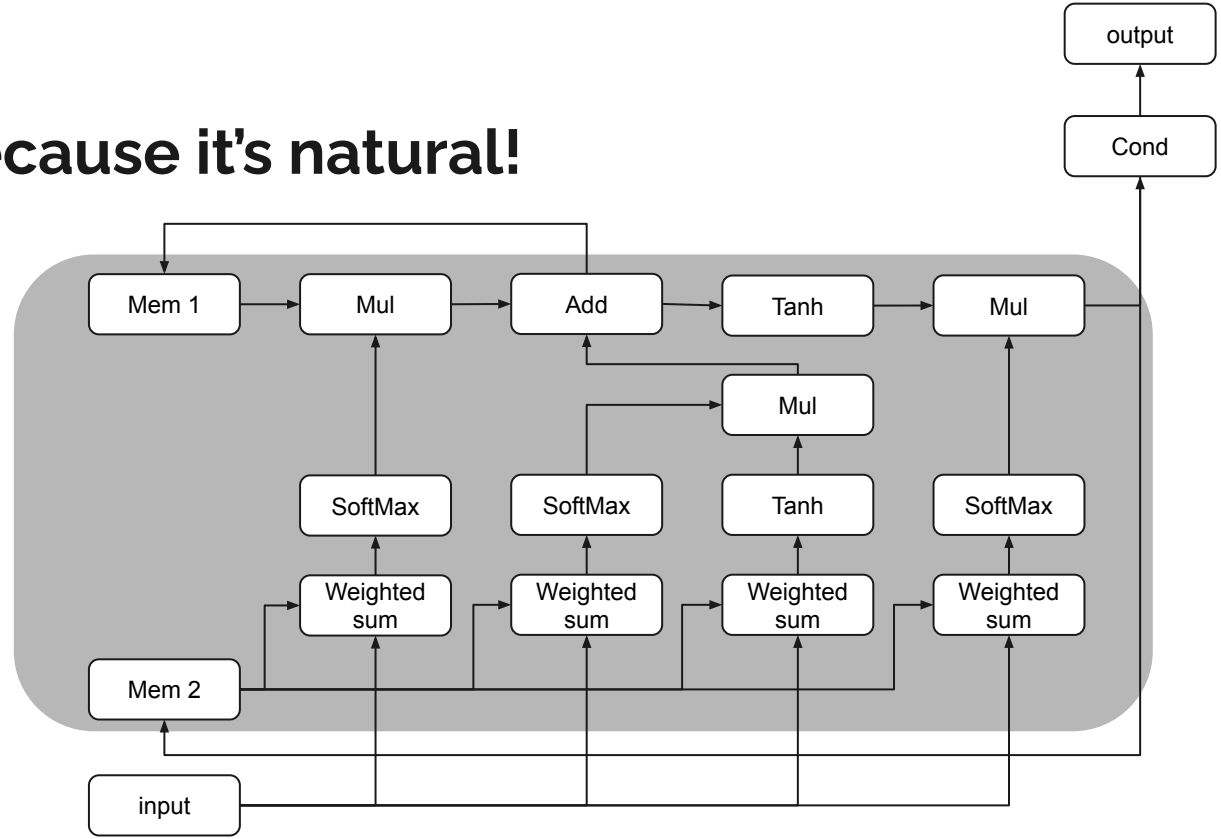
MLIR reactive extension

- Two new MLIR dialects
  - Lowering to MLIR existing dialects
  - sync = SSA extension needed to represent reactive behaviors
    - Operations: tick, input, output, inst, sync, undef
    - Types: in<t>, out<t>
  - lus = Lustre embedding (dataflow control + clock calculus)
- Reactive implementation of a Keras ML specification = **lus** (reactive control) + **tf** (HPC handling of data)
- Compilation to MLIR targets
  - IREE VM (runs on multiple targets: multi-core, GPU…)
  - CPU single-core (“no VM”) with customized compilation pipeline and reactive modularity



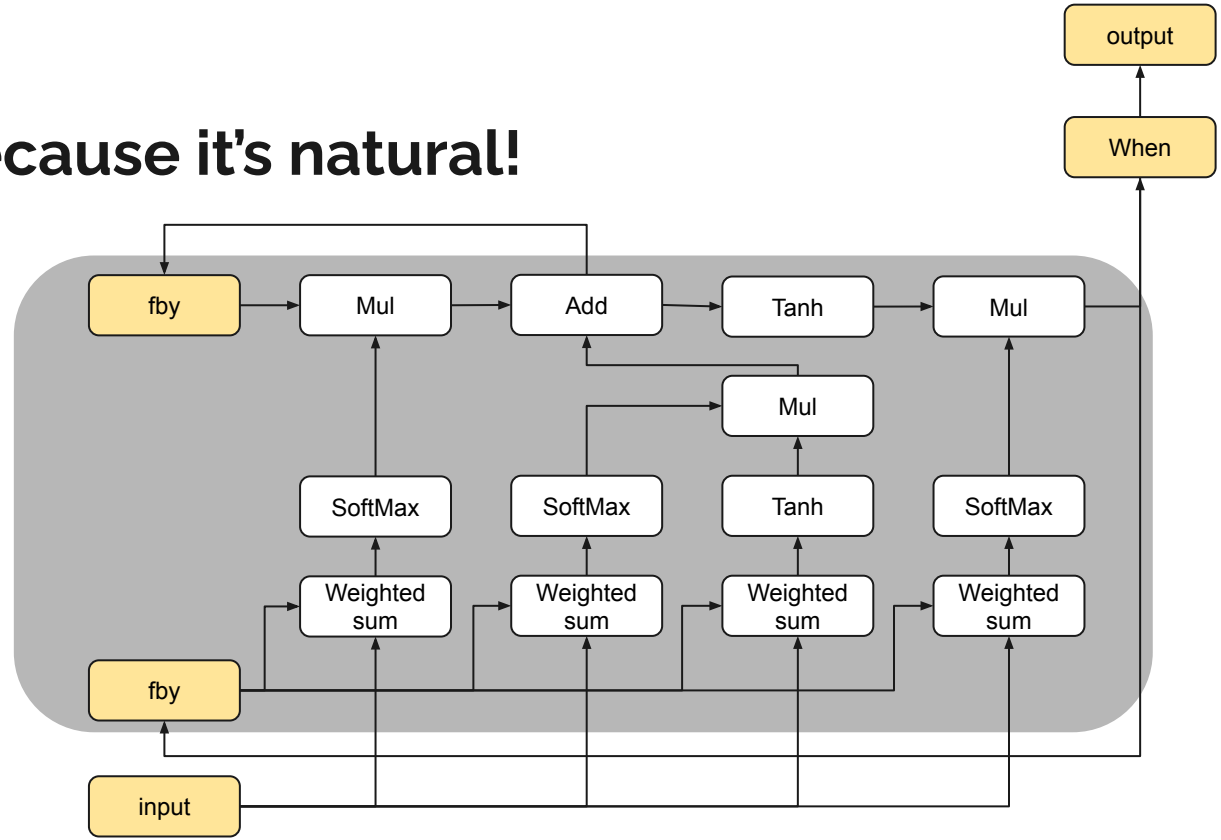
Why dataflow: because it's natural!

- A recurrent neural network: LSTM (Hochreiter & Schmidhuber, Neural computation, '97)
- Dataflow = describes a cyclic behavior
  - Stateful iteration
  - Conditional activation



Why dataflow: because it's natural!

- A recurrent neural network: LSTM (Hochreiter & Schmidhuber, Neural computation, '97)
- Dataflow = describes a cyclic behavior
  - Stateful iteration
  - Conditional activation
- Natural specification
  - Reactive control: Lustre operators (fby, when)
  - Handling of data: existing HPC operators



Lustre : a dataflow synchronous formalism (Caspi et al., 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** (volatile variables, buffers…)
- Dataflow language
  - Computation driven by data
  - A var can be absent in a cycle (predication 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 ends 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
```

```
y = x when ck; // x present, but y absent in cycles where ck is false
z = f(y);      // f is not executed when y is absent
u = g(x,z);    // Error: when ck is false, x is present and z is absent
```

Challenge 1: incorporate Lustre absence in SSA

- 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
    - In the example above :
      - ∀ ck: Clk(x) = Clk(ck), Clk(y) = Clk(x) & ck, Clk(z)=Clk(y), Clk(u)=Clk(x)=Clk(z)
      - Low-complexity calculus, part of the language semantics (example rejected)

Challenge 1: incorporate Lustre absence in SSA

- Not compliant with SSA semantics (due to dominance)!
- BUT: Non-initialization needed to represent C behavior

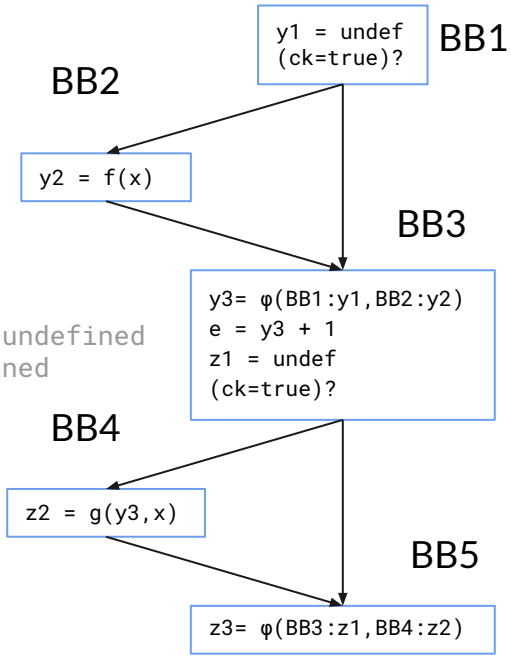
```
int y,z;
if (ck) y = f(x); // undefined in cycles where ck is false
e = y+1;          // Undefined behaviour when ck false, as y undefined
if (ck) z = g(y); // fully defined behaviour if x and ck defined
```

Challenge 1: incorporate Lustre absence in SSA

- Not compliant with SSA semantics (due to dominance)!
- BUT: Non-initialization needed to represent C behavior

```
int y,z;
if (ck) y = f(x); // undefined in cycles where ck is false
e = y+1;          // Undefined behaviour when ck false, as y undefined
if (ck) z = g(y); // fully defined behaviour if x and ck defined
```

- LLVM IR introduces undefinedness (undef, poison)
  - Special values (ok for SSA dominance)
  - **Semantics is not that of synchronous absence**
    - These values can still be used in computations
    - C compilers aim to preserve undefined behaviors
- **Lustre has a more restrictive approach on program correctness**



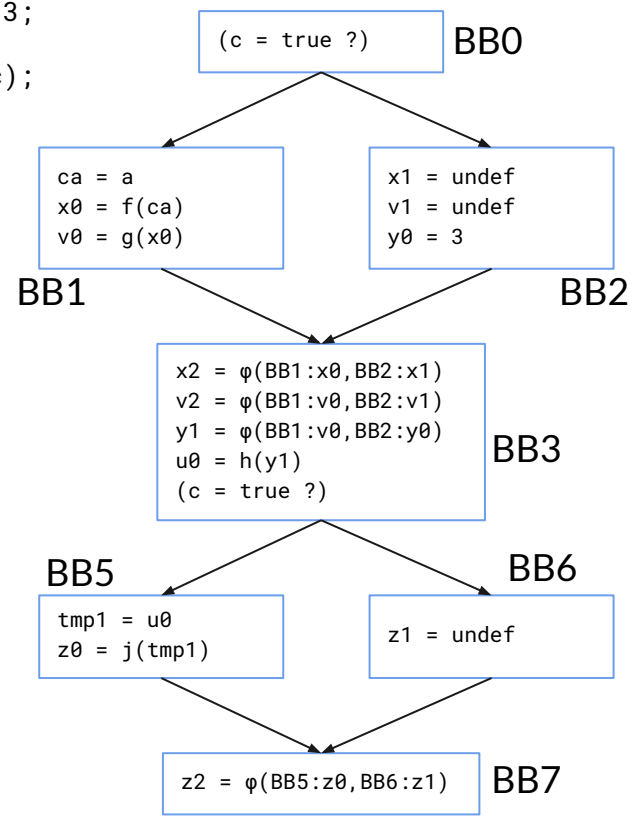
Challenge 1: incorporate Lustre absence in SSA

- Theorem [Compilation of synchronous absence]
  - Given a correct synchronous specification (where sync.undef values are never used)
  - These values can be lowered to any lower-level SSA value
    - llvm.undef, llvm.poison, constant, malloc without initialization…

Challenge 1: incorporate Lustre absence in SSA

- Theorem [Compilation of synchronous absence]
  - Given a correct synchronous specification (where sync.undef values are never used)
  - These values can be lowered to any lower-level SSA value
    - llvm.undef, llvm.poison, constant, malloc without initialization…
- Needed when compiling lus/Lustre
  - Clock analysis : ensure that absent values are never used
  - Lustre absence : lowering to sync.undef + SSA branching/merging

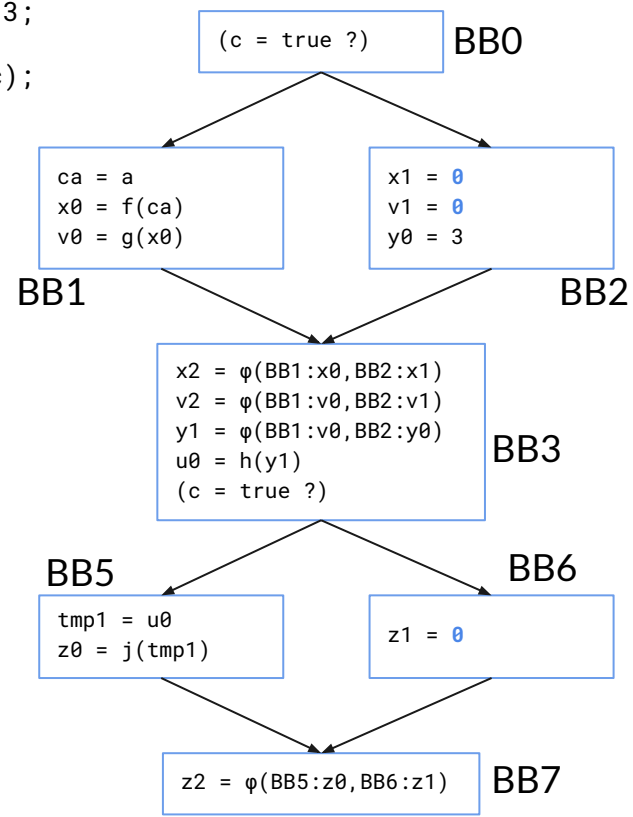
```
x = f(a when c)
v = g(x)
y = merge c v 3;
u = h(y);
z = j(u when c);
```



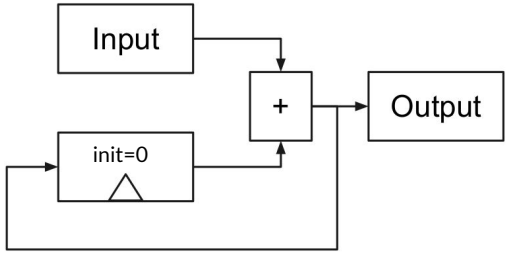
Challenge 1: incorporate Lustre absence in SSA

- Theorem [Compilation of synchronous absence]
  - Given a correct synchronous specification (where sync.undef values are never used)
  - These values can be lowered to any lower-level SSA value
    - llvm.undef, llvm.poison, constant, malloc without initialization…
- Needed when compiling lus/Lustre
  - Clock analysis : ensure that absent values are never used
  - Lustre absence : lowering to sync.undef + SSA branching/merging
  - Sync.undef : lowering to any SSA valid value (here : **constant**)

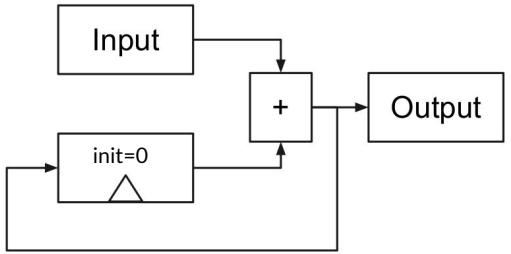
```
x = f(a when c)
v = g(x)
y = merge c v 3;
u = h(y);
z = j(u when c);
```



Challenge 2: the internal state



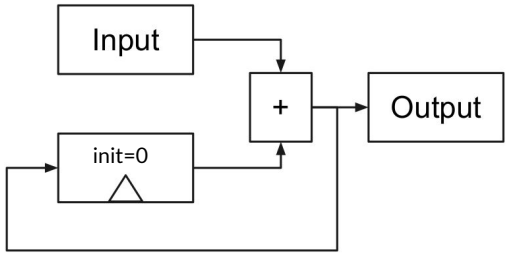
- Exemple: an integrator
  - Sums its input with the output of precedent cycles (init = 0)
  - Outputs the resulting value



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

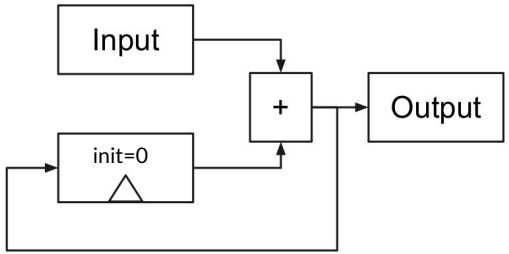
```
lus.node @integr(%i: tensor<i32>)
                 ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```



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
  - But not SSA-compliant
    - Dominance constraints relaxed

```
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
  - But not SSA-compliant
    - Dominance constraints relaxed
    - **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>)
}
```

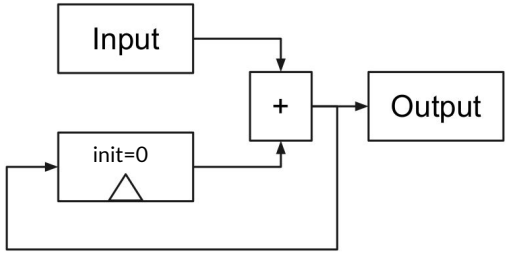
```
lus.node @integr(%i: tensor<i32>)
                 state(%os: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %f = lus.kperiodic 1(0)
  %s = lus.merge %f %c0 %os: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
      state(%incr:tensor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

Normal form:

- Clocking all fbys on the base clock
- Grouping all fbys (in node signature + node terminal operation)

Compilation to *sync*

- Traditional: control inversion
  - Driver: reactive control (tick, cyclic IOs)
  - Step/reset functions
- Communicating automata
  - Explicit main loop + internal state
  - IO signals, cycle barrier (tick)

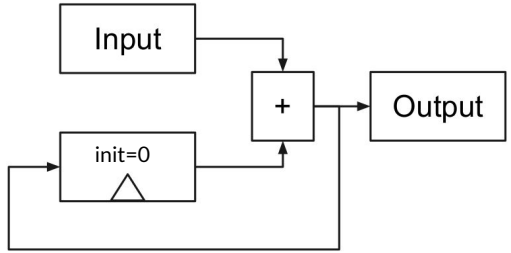



```
lus.node @integr(%i: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = tf.Const{dense<0>}: tensor<i32>
  %s = lus.fby %c0 %incr: tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield(%incr: tensor<i32>)
}
```

```
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
}
```

Compilation to *sync*

- Traditional: control inversion
  - Driver: reactive control (tick, cyclic IOs)
  - Step/reset functions
- Communicating automata
  - Explicit main loop + internal state
  - Function + buffers level
    - Signals -> functions
    - Tick -> function
      - Cycle barrier persistent until executable code




```
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
}
```

Node-based modularity

- Based on node instanciation
  - ≠ transformational function calls
  - describe a (uniquely identified) sub-automaton running under the system scheduler

```
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.node @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>
    %n = call @print_i32(%o): tensor<i32> -> (none)
    %sy = sync.tick(%n)
    sync.sync(%sy)
    scf.yield
  }
  sync.halt
}
```

Node-based modularity

- Implementation: functions interacting with the environment
  - @integr_start: used to launch a new instance of @integr
  - @sch_set_instance: declaration of the new instance
  - @sch_set_io_X: declaration of the instance IO buffers
  - @inst: give back control until the instance achieves a cycle

```
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_start:(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 LSTM

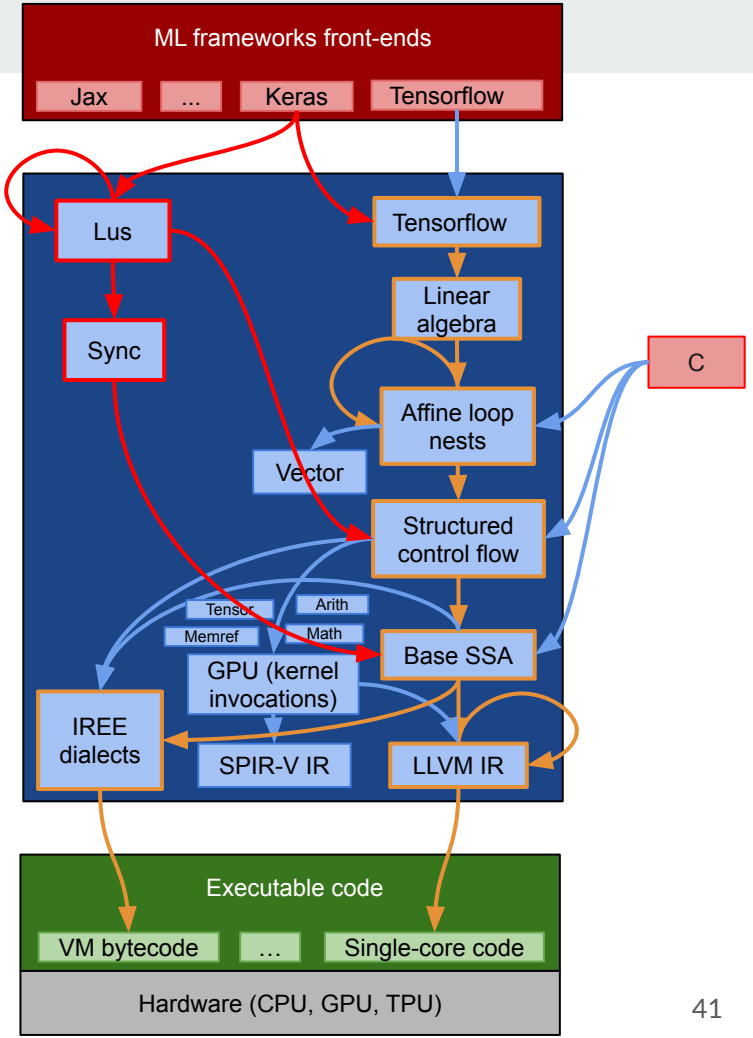
```
input = Keras.Input(shape=49,40)
x = layers.LSTM(units=4)(input)
x = layers.Dense(units=4)(x)
model = keras.Model(input,output)
model.load_weights('lstm_weights.h5')
```

```
lus.node @model(%x0:tensor<1x40xf32>)
                ->(tensor<1x4xf32>) {
 %x1 = lus.inst @lstm(%x0): tensor<1x4xf32>
 %c = lus.inst @true_each_49(): i1
 %x2 = lus.when %c %x1: tensor<1x4xf32>
 %x3 = lus.inst @dense(%x2): tensor<1x4xf32>
 lus.yield (%x3: tensor<1x4xf32>)
}
```

```
lus.node @lstm (%in:tensor<1x40xf32>)->(tensor<1x4xf32>) {
 %res = lus.inst @true_each_49(): i1
 %hmem0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
 %cmem0 = tf.Const{dense<XXX>}: tensor<1x4xf32>
 %hmem1 = lus.fby(%hmem0, %out): tensor<1x4xf32>
 %hmem = arith.select %res,%hmem0,%hmem1: tensor<1x4xf32>
 %cmem1 = lus.fby(%cmem0, %cmem_up): tensor<1x4xf32>
 %cmem = arith.select %res,%cmem0,%cmem1: tensor<1x4xf32>
 %34 = call @WS1(%input, %hmem): tensor<1x4xf32>
 %35 = tf.Softmax(%34): tensor<1x4xf32>
 %47 = call @WS2(%input, %hmem): tensor<1x4xf32>
 %48 = tf.Softmax(%47): tensor<1x4xf32>
 %25 = call @WS3(%input, %hmem): tensor<1x4xf32>
 %26 = tf.Softmax(%25): tensor<1x4xf32>
 %56 = call @WS4(%input, %hmem): tensor<1x4xf32>
 %57 = tf.Tanh(%56): tensor<1x4xf32>
 %39 = tf.Mul(%35, %cmem): tensor<1x4xf32>
 %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
 %cmem_up = tf.AddV2(%39,%58): tensor<1x4xf32>
 %60 = tf.Tanh(%cmem_up) : tensor<1x4xf32>
 %out = tf.Mul(%26,%60): tensor<1x4xf32>
 lus.yield (%res: tensor<1x4xf32>)
}
```

Experimental results (1/3)

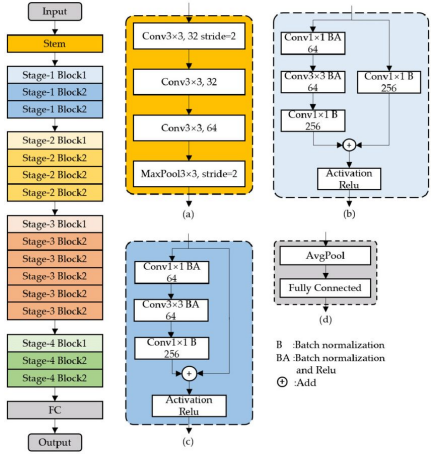
- **Non-intrusiveness** : high degree of MLIR code reuse
  - Need to write:
    - Clock analysis
    - Normalization
    - Synthesis of low-level control
  - Reuse: causality analysis, optimizations, code generation…

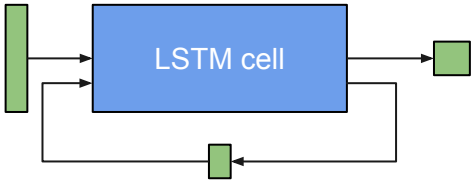


Experimental results (2/3)

- **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:
    - **RTE state of the art**: no performance loss w.r.t traditional Lustre compiler + gcc -03
  - Pipeline targetting the IREE VM (CPU, GPU):
    - **HPC state of the art**: no performance loss w.r.t IREE standard pipeline
    - (Widely more efficient than the previous approach)

A residual network (Resnet50)

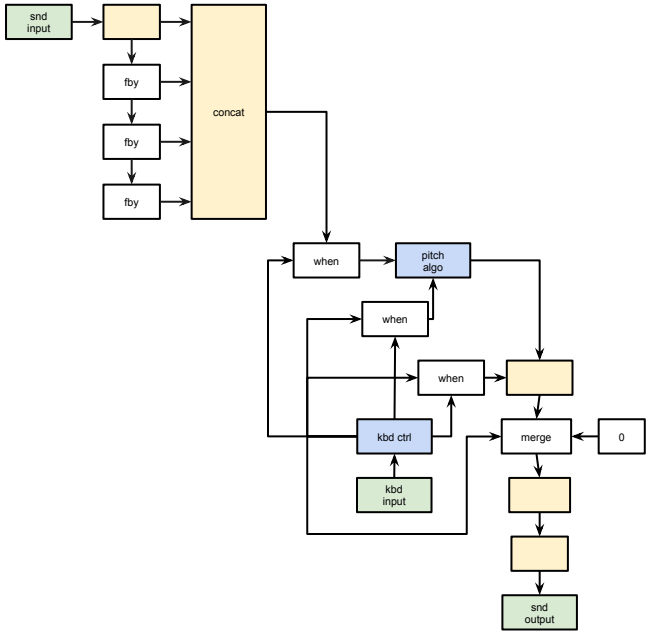




An LSTM layer (reactive representation)

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



Conclusion

- Contribution 1: Conservative reactive extension of the SSA form
- Contribution 2: Formal and tooled integration of Lustre as a dialect of MLIR SSA
  - Natural representation of Keras specifications (dataflow)
    - Conditional execution, recurrence, periodic execution…
  - Beyond Keras specifications: multi-period, input data buffering…
  - Concise and sound semantics
  - Full compilation, no performance loss

Future work: training neural networks at lus level

- Lus-level specification of the training phase of neural networks
  - 1: training and prediction on the same specification
    - State of the art (KWS Streaming): transformational training, cyclic prediction
    - Robustness issue
  - 2: embedded reinforcement learning (learning “in vivo”)
- Effectively predictible embedded implementation
  - Static resources allocation (e.g. memory)

Thank you for the attention

Why would one go beyond step/reset ?

- An example inspired from avionics
  - 2 tasks running concurrently
  - Period 2, but alternately
  - Feedback loop

```
node n()returns ()
var c:bool; x,y: int ;
let
  c = false fby (not c) ;
  x = condact(c    ,f,y      ,init_x) ;
  y = condact(not c,g,0 fby x,init_y) ;
tel
```

Why would one go beyond step/reset ?

- Implementation by a communicating automaton
  - Equivalent to step/reset implementation
  - (but reactive control is incorporated)
  - One-fits-all

```
node n()returns ()
var c:bool; x,y: int ;
let
  c = false fby (not c) ;
  x = condact(c   ,f,y  ,init_x) ;
  y = condact(not c,g,0 fby x,init_y) ;
tel
```

step function equivalent

```
sync.func @n()->() {
  br ^bb1(false,init_x,init_y)
^bb1 (%c,%x,%y):
  %x1 = if %c then {
    %t = call @f(%y)
    yield %t
  }
  else {
    yield %x
  }
  %y1 = if %c then {
    yield %y
  }
  else {
    %t = call @g(%x)
    yield %t
  }
  %c1 = not %c
  tick()
  br ^bb0(%c1,%x1,%y1)
}
```

Why would one go beyond step/reset ?

- Implementation by a communicating automaton
  - Equivalent to step/reset implementation
  - (but reactive control is incorporated)
  - One-fits-all
- Give access to a wider range of SSA transformations
  - loop unrolling (size 2)
  - loop invariant extraction
  - constant propagation, unused variables deletion

```
node n()returns ()
var c:bool; x,y: int ;
let
  c = false fby (not c) ;
  x = condact(c   ,f,y  ,init_x) ;
  y = condact(not c,g,0 fby x,init_y) ;
tel
```

step function equivalent

```
sync.func @n()->() {
  br ^bb1(false,init_x,init_y)
^bb1 (%c,%x,%y):
  %x1 = if %c then {
    %t = call @f(%y)
    yield %t
  }
  else {
    yield %x
  }
  %y1 = if %c then {
    yield %y
  }
  else {
    %t = call @g(%x)
    yield %t
  }
  %c1 = not %c
  tick()
  br ^bb0(%c1,%x1,%y1)
}
```

Why would one go beyond step/reset ?

- Implementation by a communicating automaton
  - Equivalent to step/reset implementation
  - (but reactive control is incorporated)
  - One-fits-all
- Give access to a wider range of SSA transformations
  - loop unrolling (size 2)
  - loop invariant extraction
  - constant propagation, unused variables deletion

hyper-period expansion

```
node n()returns ()
var c:bool; x,y: int ;
let
  c = false fby (not c) ;
  x = condact(c   ,f,y  ,init_x) ;
  y = condact(not c,g,0 fby x,init_y) ;
tel
```

```
sync.func @n()->() {
  br ^bb1(init_x)
^bb1 (%x):
  %y = call @g (%x)
  tick()
  %x = call @f (%y)
  tick()
  br ^bb0(%x1) }
```

```
sync.func @n()->() {
  br ^bb1(false,init_x,init_y)
^bb1 (%c,%x,%y):
  %x1 = if %c then {
    %t = call @f(%y)
    yield %t
  }
  else {
    yield %x
  }
  %y1 = if %c then {
    yield %y
  }
  else {
    %t = call @g(%x)
    yield %t
  }
  %c1 = not %c
  tick()
  br ^bb0(%c1,%x1,%y1)
}
```

Why would one go beyond step/reset ?

- Results
  - Less loop-carried variables (1 instead of 3)
  - Elimination of conditional control

```
node n()returns ()
var c:bool; x,y: int ;
let
  c = false fby (not c) ;
  x = condact(c   ,f,y  ,init_x) ;
  y = condact(not c,g,0 fby x,init_y) ;
tel
```

```
sync.func @n()->() {
  br ^bb1(init_x)
^bb1 (%x):
  %y = call @g (%x)
  tick()
  %x = call @f (%y)
  tick()
  br ^bb0(%x1) }
```

```
sync.func @n()->() {
  br ^bb1(false,init_x,init_y)
^bb1 (%c,%x,%y):
  %x1 = if %c then {
    %t = call @f(%y)
    yield %t
  }
  else {
    yield %x
  }
  %y1 = if %c then {
    yield %y
  }
  else {
    %t = call @g(%x)
    yield %t
  }
  %c1 = not %c
  tick()
  br ^bb0(%c1,%x1,%y1)
}
```

Clocking all fbys on the base clock

```
node n(i:int,c:bool) returns (o:int)
var ic,xc,oc     : int ;
let
  ic = i when c ;
  xc = ic + oc ;

  oc = 0 fby xc ;

  o  = merge c oc 0 ;
tel
```

Clocking all fbys on the base clock

```
node n(i:int,c:bool) returns (o:int)
var ic,xc,oc,u,t : int ;
let
  ic = i when c ;
  xc = ic + oc ;
  u = merge c xc (t whenot c) ;
  t = 0 fby u;
  oc = t when c ;
  o  = merge c oc 0 ;
tel
```

LSTM in Keras

- Keras specification
  - Dataflow graph, but
  - Target = classical function(s)
    - **Collapse the full time horizon (49 time steps, 1s worth of sound data) into a single evaluation**
    - 40 values for each time step

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

LSTM in Keras and MLIR

- Keras specification
  - Dataflow graph, but
  - Target = classical function(s)
    - **Collapse the full time horizon (49 time steps, 1s worth of sound data) into a single evaluation**
    - 40 values for each time step
- Input to the MLIR compiler
  - Generated by Keras (**largely** simplified)
  - Iterates on the full time horizon (for loop)
  - Memory = loop-carried dependencies
  - Cond = end condition of the loop
  - Then returns the prediction of the last step

```
func @lstm (%seq:tensor<49x40xf32>)->(tensor<1x4xf32>) {
 %hmem0 = tf.Const(){dense<XXX>}: tensor<1x4xf32>
 %cmem0 = tf.Const{dense<XXX>}: tensor<1x4xf32>

 %s:2 = scf.for %i = %zero to %size step %one
  iter_args(%hmem=%hmem0,%cmem=%cmem0) {
  %input = tensor.extract_slice %seq,%i: tensor<1x40xf32>
  %34 = call @WS1(%input, %hmem): tensor<1x4xf32>
  %35 = tf.Softmax(%34): tensor<1x4xf32>
  %47 = call @WS2(%input, %hmem): tensor<1x4xf32>
  %48 = tf.Softmax(%47): tensor<1x4xf32>
  %25 = call @WS3(%input, %hmem): tensor<1x4xf32>
  %26 = tf.Softmax(%25): tensor<1x4xf32>
  %56 = call @WS4(%input, %hmem): tensor<1x4xf32>
  %57 = tf.Tanh(%56): tensor<1x4xf32>
  %39 = tf.Mul(%35, %cmem): tensor<1x4xf32>
  %58 = tf.Mul(%48, %57) : tensor<1x4xf32>
  %cmem_up = tf.AddV2(%39,%58): tensor<1x4xf32>
  %60 = tf.Tanh(%cmem_up) : tensor<1x4xf32>
  %output = tf.Mul(%26,%60): tensor<1x4xf32>
  scf.yield (%output: tensor<1x4xf32>,
             %cmem_up: tensor<1x4xf32>)
  }
 return %s#0: tensor<1x4xf32>
}
```