Weaving Synchronous Reactions Into the Fabric of SSA-Form Compilers

Lustre, Tensor Flow, and MLIR to rule them all

H.Pompougnac (INRIA), D.Potop-Butucaru (INRIA), A.Cohen (Google)

High performance real-time embedded systems

- A real need
  - ML components in real-time systems
    - Autonomous car
    - AI drone
  - Embedded digital twins (avionics…)
    - Model predictive control
    - Predictive maintenance
  - More traditional signal processing
    - FFT...
- Expertise distributed between two different fields
  - Real time embedded (RTE)
  - High performance computing (HPC) and in particular machine learning (ML)

Objective: RTE HPC/ML programming

High performance/ML

- Optimization
  - Average case
  - Incremental lowering
- Mainly data parallelism

Real-time embedded

- Correction guarantees (functional & non-functional)
  - Worst case analysis
  - Global optimization objectives
- Mainly task parallelism

Objective: RTE HPC/ML programming

High performance/ML

- Optimization
  - Average case
  - Incremental lowering
- Mainly data parallelism
- ML frameworks :
  - TensorFlow, PyTorch...

Real-time embedded

- Correction guarantees (functional & non-functional)
  - Worst case analysis
  - Global optimization objectives
- Mainly task parallelism
- Formalisms :
  - Simulink, Scade...

Objective: RTE HPC/ML programming

High performance/ML

- **Optimization**
  - Average case
  - Incremental lowering
- **Mainly data parallelism**
- **ML frameworks :**
  - TensorFlow, PyTorch...
  - **MLIR SSA :** intermediate (internal) representation - LLVM project

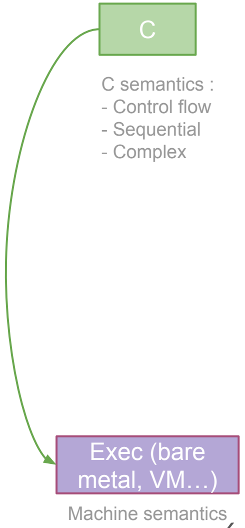
Real-time embedded

- **Correction guarantees (functional & non-functional)**
  - Worst case analysis
  - Global optimization objectives
- **Mainly task parallelism**
- **Formalisms :**
  - Simulink, Scade...
  - **Lustre :** dataflow synchronous specification formalism

Semantic and tooled integration

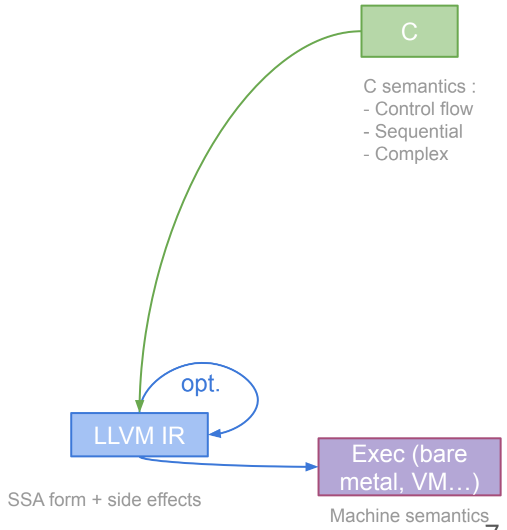
MLIR : Multi Layer Intermediate Representation

- Compiling C
    - From C to machine semantics



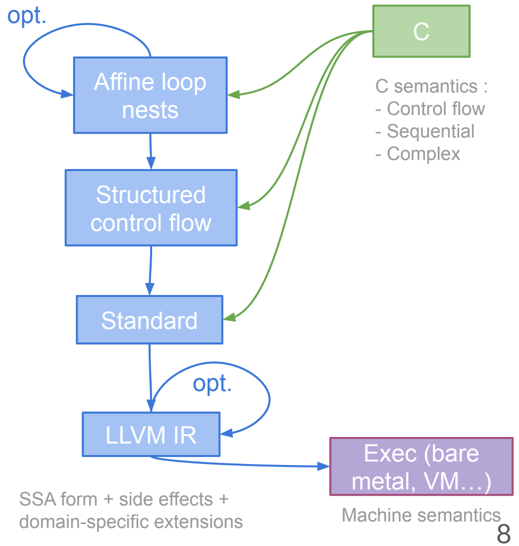
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)



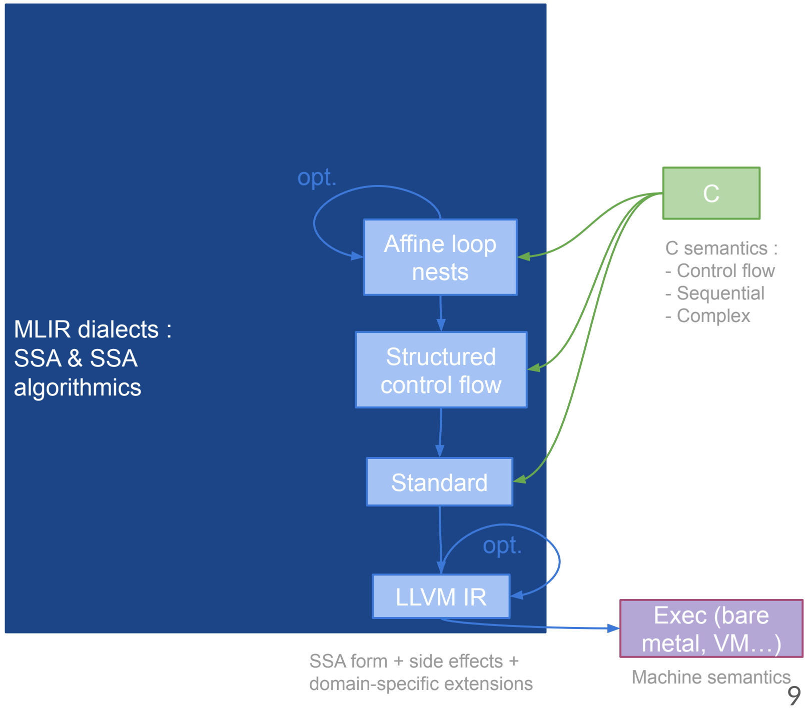
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization



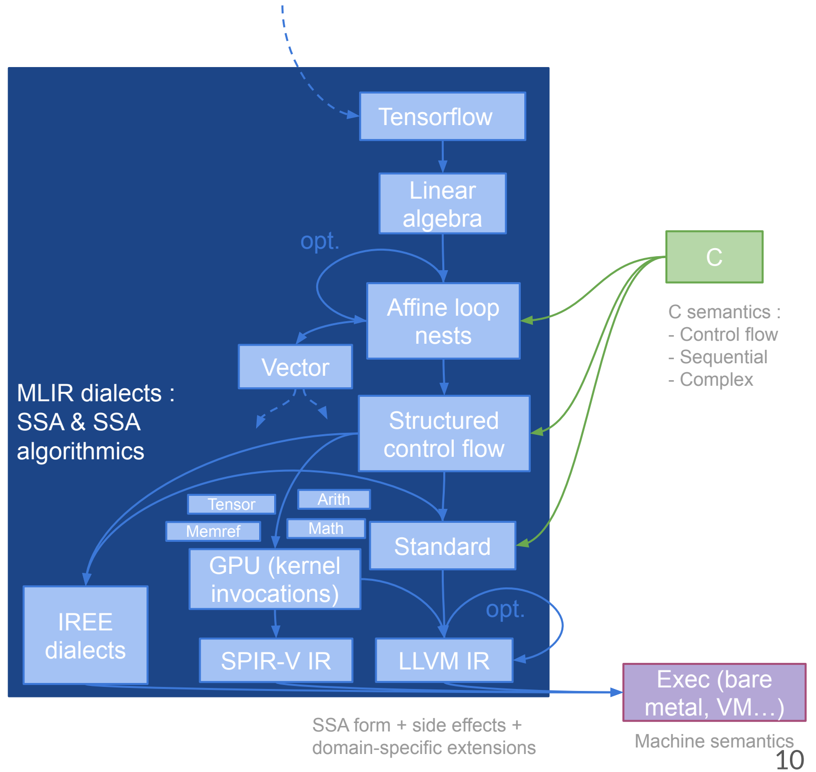
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization
  - MLIR : combine as **dialects**



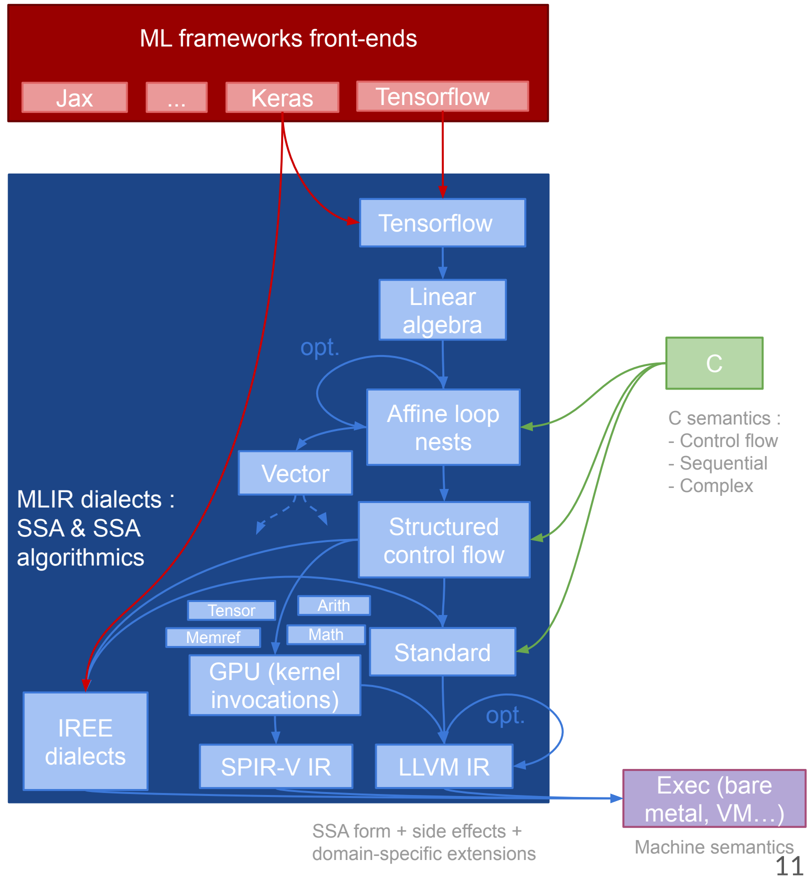
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization
  - MLIR : combine as **dialects**
  - Multiple dialects



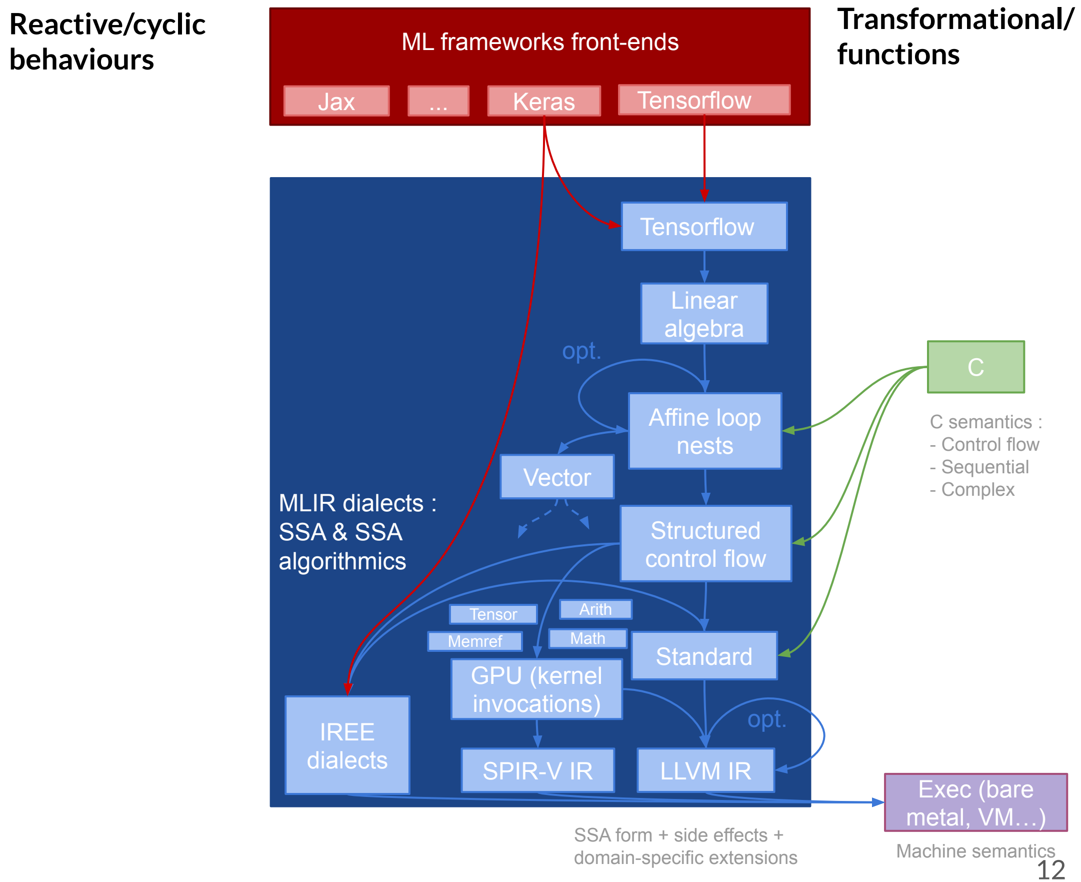
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization
  - MLIR : combine as **dialects**
  - Multiple dialects
  - Hidden behind front-ends



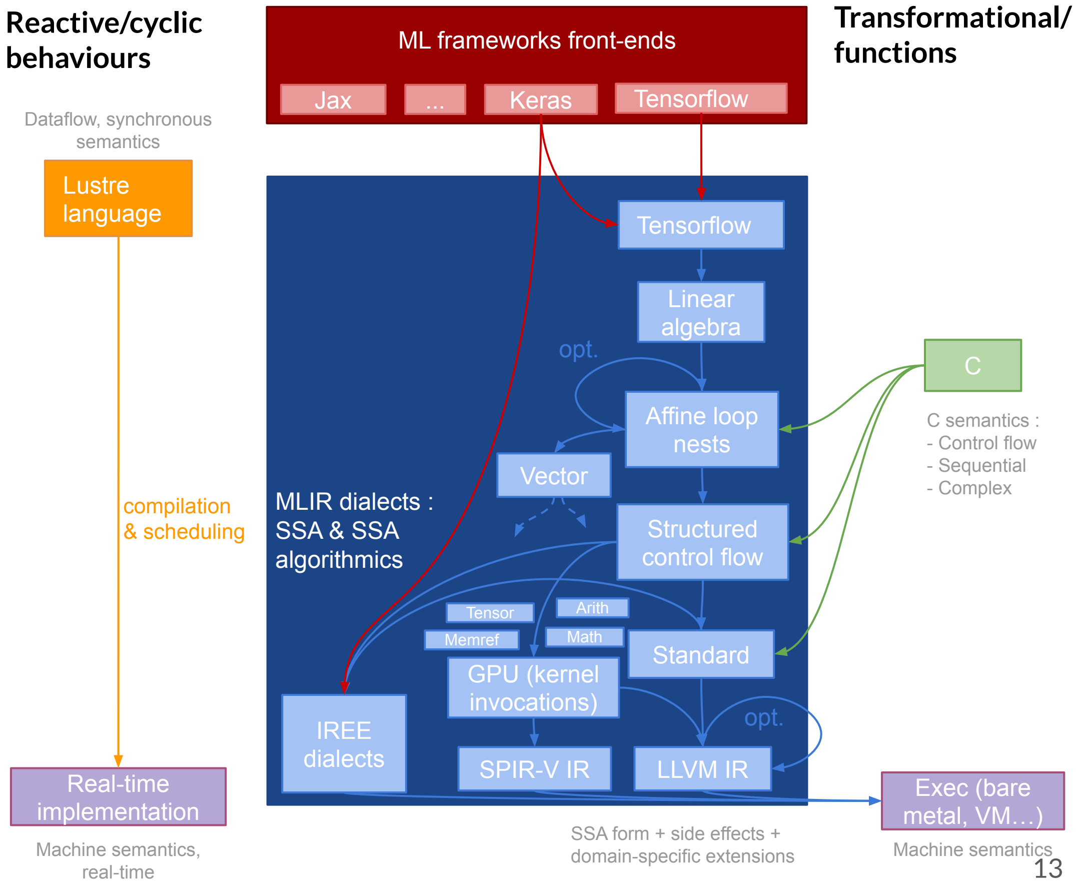
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization
  - MLIR : combine as **dialects**
  - Multiple dialects
  - Hidden behind front-ends
- Paradox : MLIR compiles only functions, not reactive specs



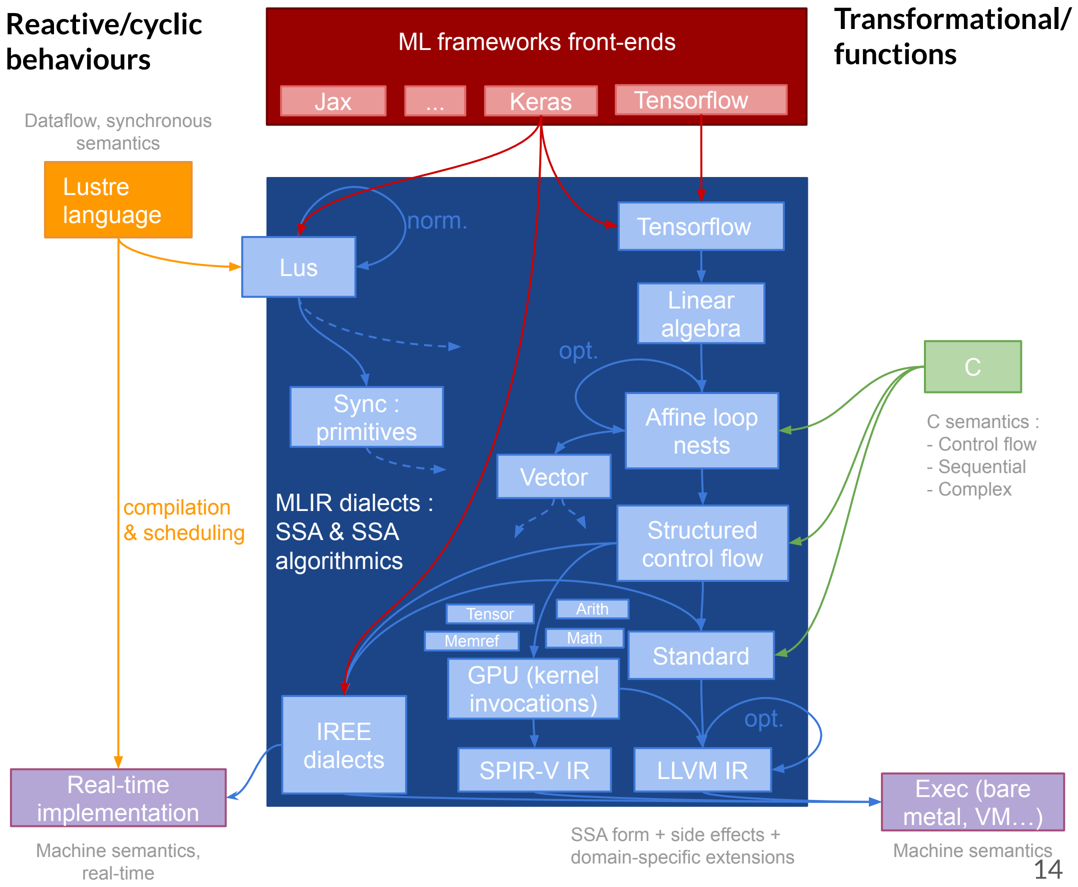
MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization
  - MLIR : combine as **dialects**
  - Multiple dialects
  - Hidden behind front-ends
- Paradox : MLIR compiles only functions, not reactive specs
  - Integration of Lustre



MLIR : Multi Layer Intermediate Representation

- Compiling C
  - From C to machine semantics
  - Under the hood : LLVM (SSA)
- Domain-specific abstractions for optimization
  - MLIR : combine as **dialects**
  - Multiple dialects
  - Hidden behind front-ends
- Paradox : MLIR compiles only functions, not reactive specs
  - Integration of Lustre



MLIR SSA vs Lustre

MLIR (SSA Form)

- **Transformational systems**
  - Focus on ML computational efficiency
- **Core SSA = no absence**
  - LLVM adds undef, poison
  - Focus on semantics preservation

Lustre (synchronous dataflow)

- **Reactive systems**
  - Embedded real-time applications
- **Absence is key part of semantics**
  - Checking correction
  - (semantics preservation too)

MLIR SSA vs Lustre

MLIR (SSA Form)

- Transformational systems
  - Focus on ML computational efficiency
- Core SSA = no absence
  - LLVM adds undef, poison
  - Focus on semantics preservation

- Globally sequential, locally concurrent execution model
- Static Single Assignment principle
  - Single assignment
  - Initialization before use : dominance analysis

Lustre (synchronous dataflow)

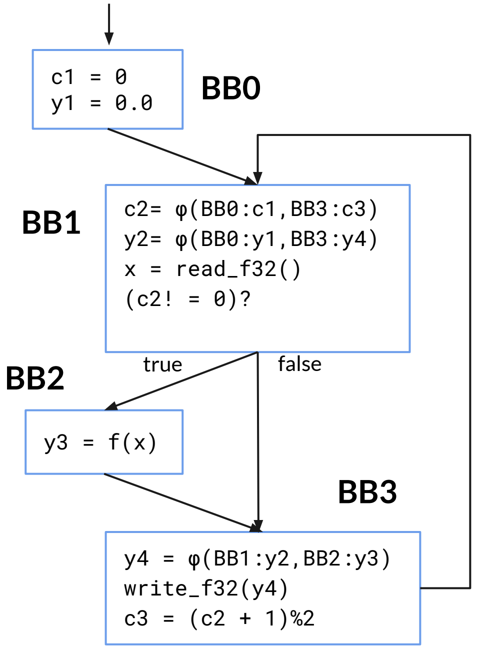
- Reactive systems
  - Embedded real-time applications
- Absence is key part of semantics
  - Checking correction
  - (semantics preservation too)

- Globally sequential, locally concurrent execution model
- Static Single Assignment principle
  - Single assignment
  - Initialization before use : causality & initialization analysis

Static Single Assignment (SSA)

- The SSA principle
  - Each variable is assigned by exactly one (syntactic) operation
  - A variable is assigned before use in each lifetime (dominance)
- The best known instance : the SSA form (1988, Rosen et al.)
  - IR for optimized compilation of sequential languages
  - Two-level representation
    - Top level : **Sequential control flow graph** (SCFG)
      - Vertices are **basic blocks** (BBs)
    - BB level : sequence of **operations** that can read and assign variables
      - Common extension : side effects

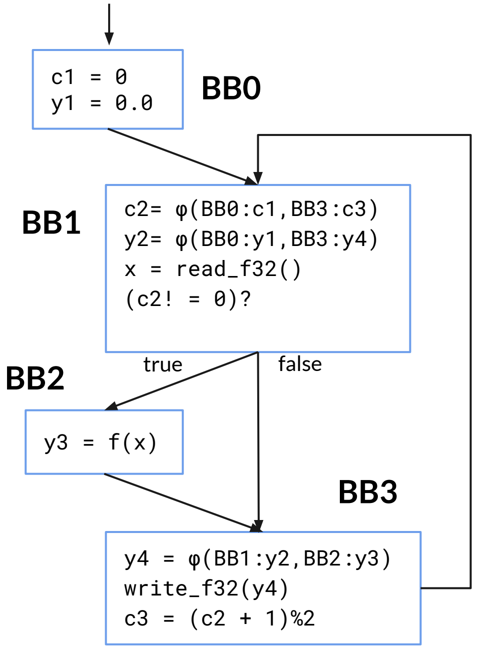
```
c = 0;
y = 0;
while(1) {
  x = read_f32();
  if (c != 0) y = f(x);
  write_f32(y);
  c = (c + 1)%2;
}
```



Static Single Assignment (SSA)

- Basic block level :
    - Topological order
        - If one operation a depends on another operation b then a is sequenced after b
    - Concurrency : 2 successive operations commute if one does not depend on the other (through a variable or a side effect)
    - Branching - at the end of each BB, control is given to at most one BB
- Very SSA-specific : the φ operator
    - Used in BBs that are destination of multiple CF arcs
    - Builds a unique value from multiple sources (depending on the control)
    - SCFG => the output of φ is uniquely defined (deterministic merge)

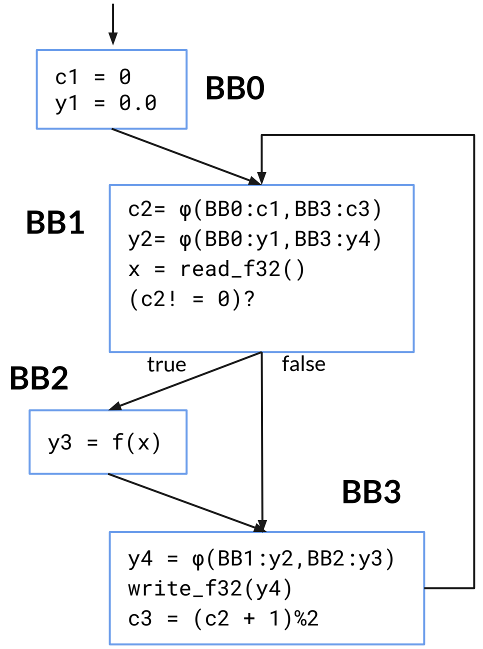
```
c = 0;
y = 0;
while(1) {
  x = read_f32();
  if (c != 0) y = f(x);
  write_f32(y);
  c = (c + 1)%2;
}
```



SSA correctness

- **Sequantiality.** The CFG of BBs is sequential : single entry point, single destination
- **Single assignment.** Each variable is either :
  - An input
  - An output of exactly one operation of one BB
- **Dominance.**
  - Ensures that any variable has been initialized before use(s)
    - BBx dominates BBy if any execution path reaching BBy passes by BBx
    - If operation o of BBx reads variable v, then one (and only one) of the following is true :
      - v is an input and BBx is reachable from the control input(s)
      - v is assigned by an operation of BBy and BBy dominates BBx
      - v is assigned by an operation of BBx preceding o inside BBx
  - No persistency : variables lose their values when control is outside of their dominance zone
- **Data type correctness.**

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

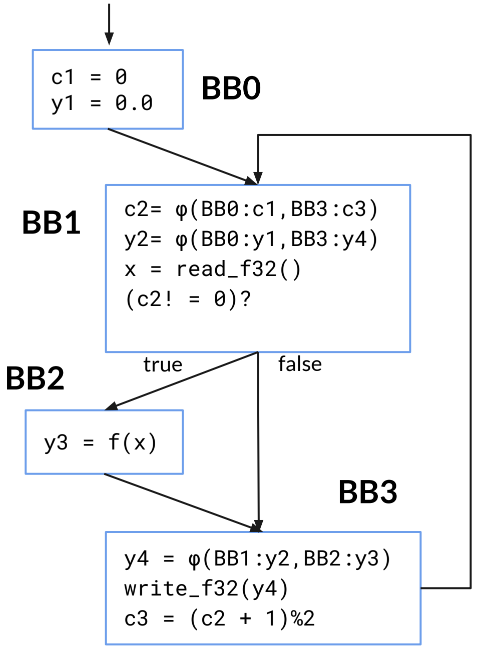


MLIR : an SSA-based IR

- Textual form
- φ operators encoded in BB entry point
    - BB wil specify all its φ ops in its interface
    - Branching op will provide values for all φ in the destination BB
- **Extensible** with “dialects”
    - Dialect = set of domain-specific constructs
    - Main focus on HPC/ML

```
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 “eq”,%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(%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;
}
```



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

Lustre : a high-level specification formalism

- Cyclic execution model
  - Sequence of execution cycles
  - Cycle = read input, compute, write output
    - 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
    - Merge 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;
    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
```

MLIR SSA to Lustre : what do we need ?

- SSA semantics extensions
  - Cyclic execution
    - Cycle barrier : tick operation
  - Modularity & instance of nodes
  - Cyclic IOs
    - IOs signals : sigin & sigout
    - Read/write from/to signals : input and output

Operational mechanisms needed to represent reactive behaviours : sync dialect

- Lustre reactive primitives
  - Dataflow control :
    - When/Merge
    - Clock analysis & signal absence
    - Fby/pre (decentralized representation of state)
  - Cyclic execution (node)

Familiar dataflow representation : lus dialect

Compilation of “lus”

- Normalization
    - Lift all fbys to the base clock
    - Transform all into pre and merge
    - Place pre into the signature
    - Order the equations in dependency order
- SSA-form requirements are satisfied
    - SSA optimizations can be applied (e.g constant propagation, CSE…)
    - Synchronous semantics

```
lus.node @integr(%i: tensor<i32>)
                ->(tensor<i32>) {
  %c0 = constant 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>)->(tensor<i32>)
                 state(%os: tensor<i32>) {
  %c0 = constant dense<0>: tensor<i32>
  %pfsts = lus.when {kp = “1(0)”} %c0: tensor<i32>
  %s = lus.merge {kp = “1(0)”} %pfsts %os:tensor<i32>
  %incr = tf.Add(%s,%i): tensor<i32>
  lus.yield (%incr:tensor<i32>)state(%s:tensor<i32>)
}
```

Compilation of “lus”

- Move into SSA operational semantics world
    - Explicit main loop
    - Read/write IO signals
    - Cycle barrier (tick)
        - Extend SSA semantics
        - Can not be associated to a SSA structural component

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

```
sync.node @integr(%is: !sync.sigin<tensor<i32>>)
               ->(%os: !sync.sigout<tensor<i32>>) {
  %true = constant 1: i1
  %c0 = constant dense<0>: tensor<i32>
  scf.while(%state = %c0):(tensor<i32>){
      scf.condition(%true)
  } do {
      %i = sync.input(%is): tensor<i32> // Input
      %incr = tf.Add(%state, %i):tensor<i32>
      %sy1 = sync.output(%os: %incr): tensor<i32> // Output
      %sy2 = sync.tick(%sy1): i32 // Sync with environment
      %nstate = sync.sync(%sy2,%incr: tensor<i32>)
      scf.yield %nstate: tensor<i32>
  }
  sync.halt
}
```

Compilation of “lus”

- Control : MLIR builtin dialects (std, scf)
- Data handling : tf dialect (input of another standard compile pipeline)
- I/Os by function calls

```
lus.node @integr(%i: tensor<i32>)
                 ->(tensor<i32>) {
  %c0 = constant 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>)->()) {
  %true = constant 1: i1
  %c0 = constant dense<0>: tensor<i32>
  scf.while(%state = %c0):(tensor<i32>) {
    scf.condition(%true)
  } do {
    %mi = memref.alloc() : memref<i32>
    %posi0 = constant 0: i32
    call_indirect %is(%posi0,%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>
    %poso0 = constant 0: i32
    call_indirect %os(%poso0,%mincr):(i32, memref<i32>) -> ()
    call @tick()
    scf.yield %incr: tensor<i32>
  }
  return
}
```

Compilation of modularity

- One solution : traditional step/reset
  - a.k.a control inversion
  - One-size-fits-all
- More freedom : instantiated nodes become concurrent processes with own state
  - At each cycle, the parent node can trigger a reaction
  - Run-time needed (non-preemptive)

```
lus.node @test(%i: tensor<i32>)->() {
  %o = lus.instance @integr(%i)
    :(tensor<i32>) -> (tensor<i32>)
  call @print_i32(%o):(tensor<i32>)->()
  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>
    call @print_i32(%o): tensor<i32> -> ()
    %sy = sync.tick()
    sync.sync(%sy)
    scf.yield
  }
  sync.halt
}
```

Compilation of modularity

- One solution : traditional step/reset
  - a.k.a control inversion
  - One-size-fits-all
- More freedom : instantiated nodes become concurrent processes with own state
  - At each cycle, the parent node can trigger a reaction
  - Run-time needed (non-preemptive)

```
lus.node @test(%i: tensor<i32>)->() {
  %o = lus.instance @integr(%i)
    :(tensor<i32>) -> (tensor<i32>)
  call @print_i32(%o):(tensor<i32>)->()
  lus.yield()
}
```

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

```
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
```

The role of absence/undefinedness

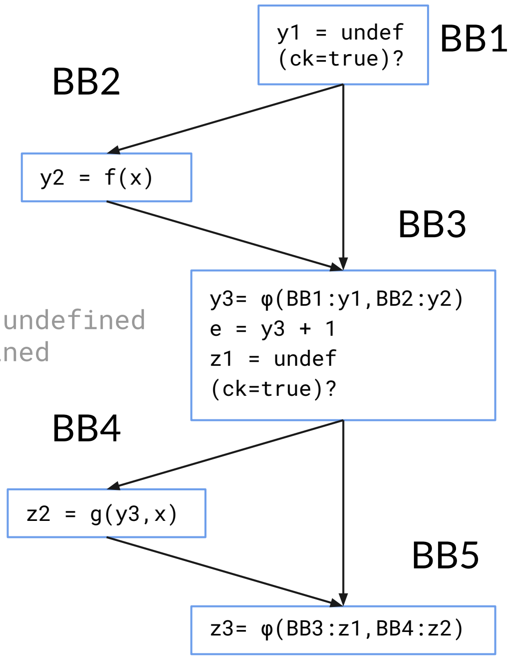
- Central concept in dataflow synchronous programming
- Computation triggered by arriving data
  - Conditional execution = conditional transmission of data = “when” operation
- Synchrony : each variable is either present or absent in each cycle
  - **Correctness : absent values are never used in computations**
    - **No uninitialized data used in computations => no undefined behavior**
  - Checking correctness : clock calculus
    - Clk(x) = predicate that is true in cycles where x is present, false in other cycles
    - Clock calculus : determines the presence/absence condition for each variable
    - 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

The role of absence/undefinedness

- Impossible in core SSA, due to dominance !
- Undefinedness 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
```

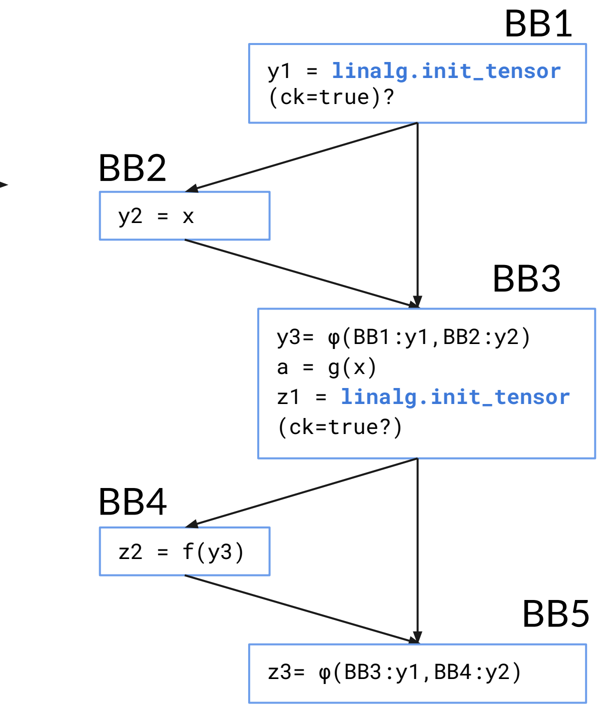
- LLVM IR introduces undef, poison
  - Special values (ok for SSA dominance)
  - Code generation = lack of initialization
    - undef, poison follow different optimization rules
  - Undefined behaviors = part of normal semantics
  - Objectives
    - Representation of C behavior in SSA
    - **Semantics preservation during optimization, code gen**



```
y = x when ck; // if (ck) y = x;
a = g(x);      // u = g(x);
z = f(y);      // if (ck) z = f(y);
```

The role of absence

- Lustre embedding into MLIR
  - Compilation of lus: use any undefined value of MLIR to represent absence (or even defined value)
    - llvm.undef, linalg.init_tensor, or simply allocate (but not initialize) a memory region
  - Clock calculus ensures that undef values are never used in computations (whose value is meant to be observed)



Encoding an LSTM

- Classical ML “layer”
- Complex to represent/manipulate
    - Stateful, multi-rate
    - Especially when generating streaming implementation code for prediction

```
lus.node @lstm(%data: tensor<3x1xf32>)
              -> (tensor<3x100xf32>) {
  // Build a clock that is true every 5 cycles
  %five = constant 5: i32
  %clk = lus.inst @modcount(%five): (i32)->(i1)
  // State and periodic reinit
  ...
  %tmp0 = lus.fby %w1 %stateout: tensor<3x100xf32>
  %tmp1 = lus.fby %w2 %l_out: tensor<3x100xf32>
  %v24 = select %clk, %w1, %tmp0 : tensor<3x100xf32>
  %v25 = select %clk, %w2, %tmp1 : tensor<3x100xf32>
  // LSTM core
  ...
  %v26 = tf.MatMul(%v24, %o76): tensor<3x400xf32>
  %v28 = tf.MatMul(%data, %o22): tensor<3x400xf32>
  %v29 = tf.AddV2(%v28, %v26): tensor<3x400xf32>
  %v30 = tf.BiasAdd(%v29, %o78): tensor<3x400xf32>
  ...
  %l_out = tf.AddV2(%v36, %v34): tensor<3x100xf32>
  %v40 = tf.Relu(%l_out): tensor<3x100xf32>
  %v41 = tf.Sigmoid(%v31_3): tensor<3x100xf32>
  %stateout = tf.Mul(%v41, %v40): tensor<3x100xf32>
  // Output subsampling
  %sample = lus.when %clk %l_out: tensor<3x100xf32>
  lus.yield (%sample: tensor<3x100xf32>)
}
```

Conclusion 1/2

- Formal and tooled integration of Lustre as a dialect of MLIR SSA
    - Full compilation
- From an MLIR perspective :
    - More concise and sound semantics for complex ML specifications
        - Conditional execution, recurrency, multi-rate, periodic execution…
        - Streaming embedded code generation vs (current) transformational code generation
- From a reactive programming perspective :
    - Integrate HPC components into critical embedded applications (ML, digital twin, etc)

Conclusion 2/2

- MLIR = compiler-building framework
  - What we had “for free” (the learning curve is not easy)
    - Type and causality verification
    - General-purpose optimization (CSE, constant propagation…)
    - Code generation and optimization for data processing (tensors, vectorization, linear algebra, ML…)
    - Memory allocation
    - Optimized back-ends (CPU/GPU…)
  - What code we still had to write
    - The definition of the “lus” and “sync” dialect operations
    - The lus normalization
    - The clock analysis
    - The lowering of “lus” operations to “sync” operations
    - The lowering of “sync” operations to MLIR builtin dialects (std, scf…)
    - The OS : implementation of “sync” primitives (outside of MLIR itself)

Future work

- Real-time implementation
    - “lus” : access to real-time specification and implementation methods
    - Control of allocation and scheduling over MLIR back-ends
    - Worst-case analysis
        - Requires timing accounting
- Reactive ML specification
    - Is “lus” a good support for other aspects of ML programming and implementation ?
        - Automatic differentiation

Thank you