



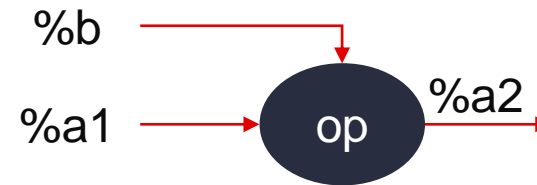
Representing Concurrency with Graph Regions in MLIR

Stephen Neuendorffer
LLVM Developer Meeting 2021



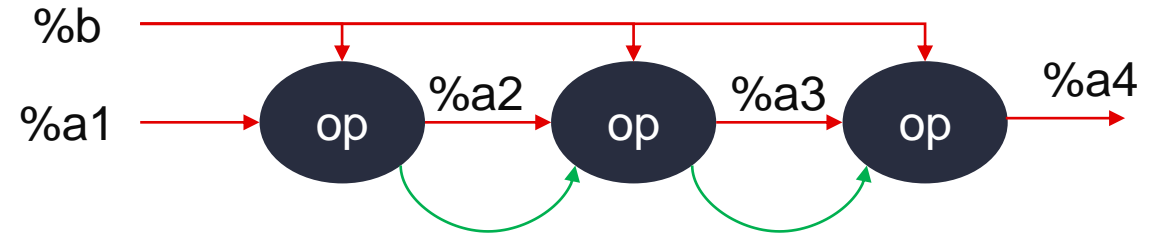
Basic MLIR Concepts

```
%a2 = op(%a1, %b)
```



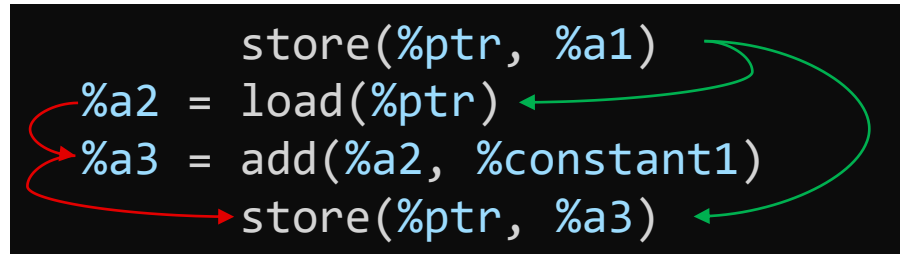
Basic MLIR Concepts

```
ssaRegion(%a1, %b) {  
  %a2 = op(%a1, %b)  
  %a3 = op(%a2, %b)  
  %a4 = op(%a3, %b)  
}
```



- ▶ At a fundamental level MLIR represents:
 - An ordered list of operations (nodes in a graph)
 - The connections between operations (edges in a graph)
- ▶ A list of operations represents a DAG (with multiple destination edges).
- ▶ Every DAG can be represented as a (not unique) list of operations.

This is great for representing sequential programs!



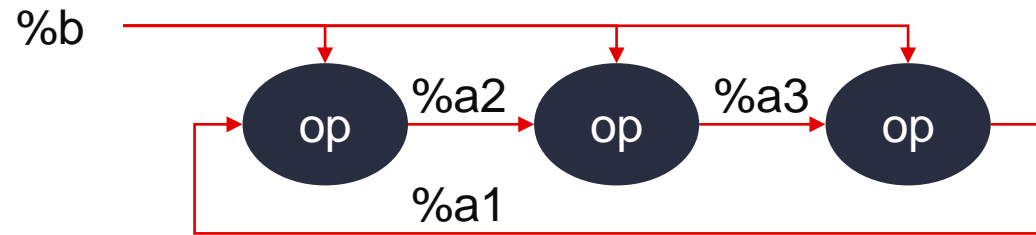
```
store(%ptr, %a1)
%a2 = load(%ptr)
%a3 = add(%a2, %constant1)
store(%ptr, %a3)
```

The diagram illustrates a code block with four lines of code. Red arrows indicate data dependencies: from the first `store` to the `load`, and from the `load` to the `add`. A green arrow indicates a sequential dependency from the first `store` to the final `store`. The code is as follows:

- ▶ Encodes data dependencies (load -> add -> store)
- ▶ Encodes sequential dependencies (store -> load, store -> store)
- ▶ With Basic Blocks can represent Single-static assignment control flow graph
- ▶ Arbitrary operations enables representing structured control flow

Goal of this talk

```
NOTSSA(%b) {  
  %a2 = op(%a1, %b)  
  %a3 = op(%a2, %b)  
  %a1 = op(%a3, %b)  
}
```



- ▶ But can we represent an *arbitrary graph with cycles*?
 - Yes! Graphs are directly supported in MLIR through *Graph Regions*
- ▶ Is this useful? What can it mean?
 - Yes! Graphs are great for representing concurrent programs!

Regions in MLIR

- ▶ Every operation can contain regions of code.
 - addf: no regions
 - func: 1 region
 - scf.if: 2 regions

```
func @std_if_else(%arg0: i1, %arg1: f32) {  
  scf.if %arg0 {  
    %0 = addf %arg1, %arg1 : f32  
  } else {  
    %1 = addf %arg1, %arg1 : f32  
  }  
  return  
}
```

MLIR Region Kinds

- ▶ The containing operation determines the kind of the region
 - see `RegionKindInterface`
 - see `HasOnlyGraphRegion`

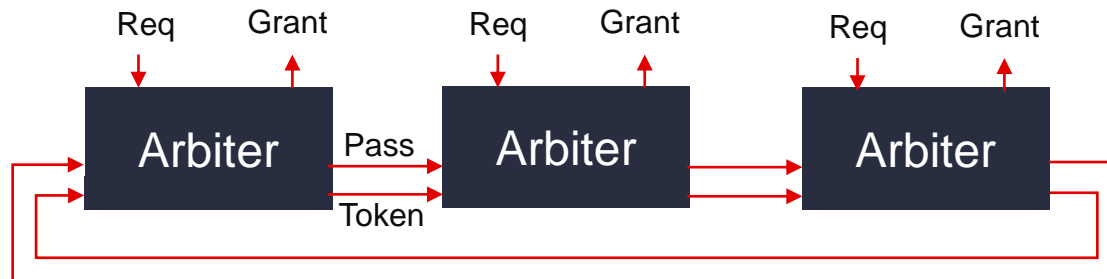
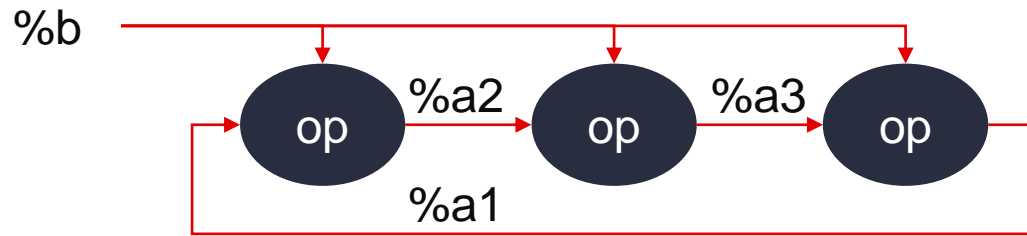
SSACFG: Multiple basic blocks and no cycles within a basic block

```
func @std_if_else(%arg0: i1, %arg1: f32) {  
  scf.if %arg0 {  
    %0 = addf %arg1, %arg1 : f32  
  } else {  
    %1 = addf %arg1, %arg1 : f32  
  }  
  return  
}
```

Graph: A single basic block allowing cycles

```
hw.module @top(%clk: i1, %rst: i1)  
  -> (o: i1) {  
    %t = hw.constant true  
    %f = hw.constant false  
    %r0 = seq.compreg %0, %clk, %rst, %f : i1  
    %0 = comb.xor %r0, %t : i1  
    hw.output %r0 : i1  
  }
```

Circuits with Combinational Cycles



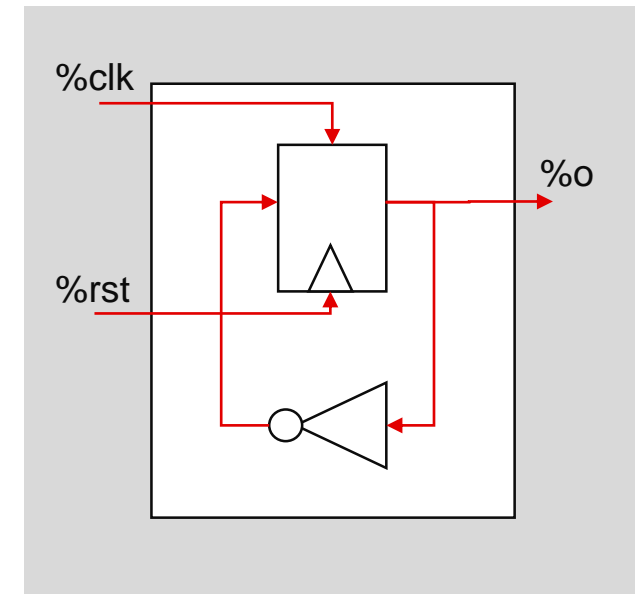
PassOut = (PassIn or TokenIn) and not Req
Grant = (PassIn or TokenIn) and Req
TokenOut = prev TokenIn

- ▶ Each clock
 - One Arbiter has the priority token
 - One request will be granted to the next arbiter in the chain after the priority token
- ▶ The priority token sequences through the arbiters
- ▶ Observation:
 - The 'Pass' cycle has no registers and cannot be statically ordered

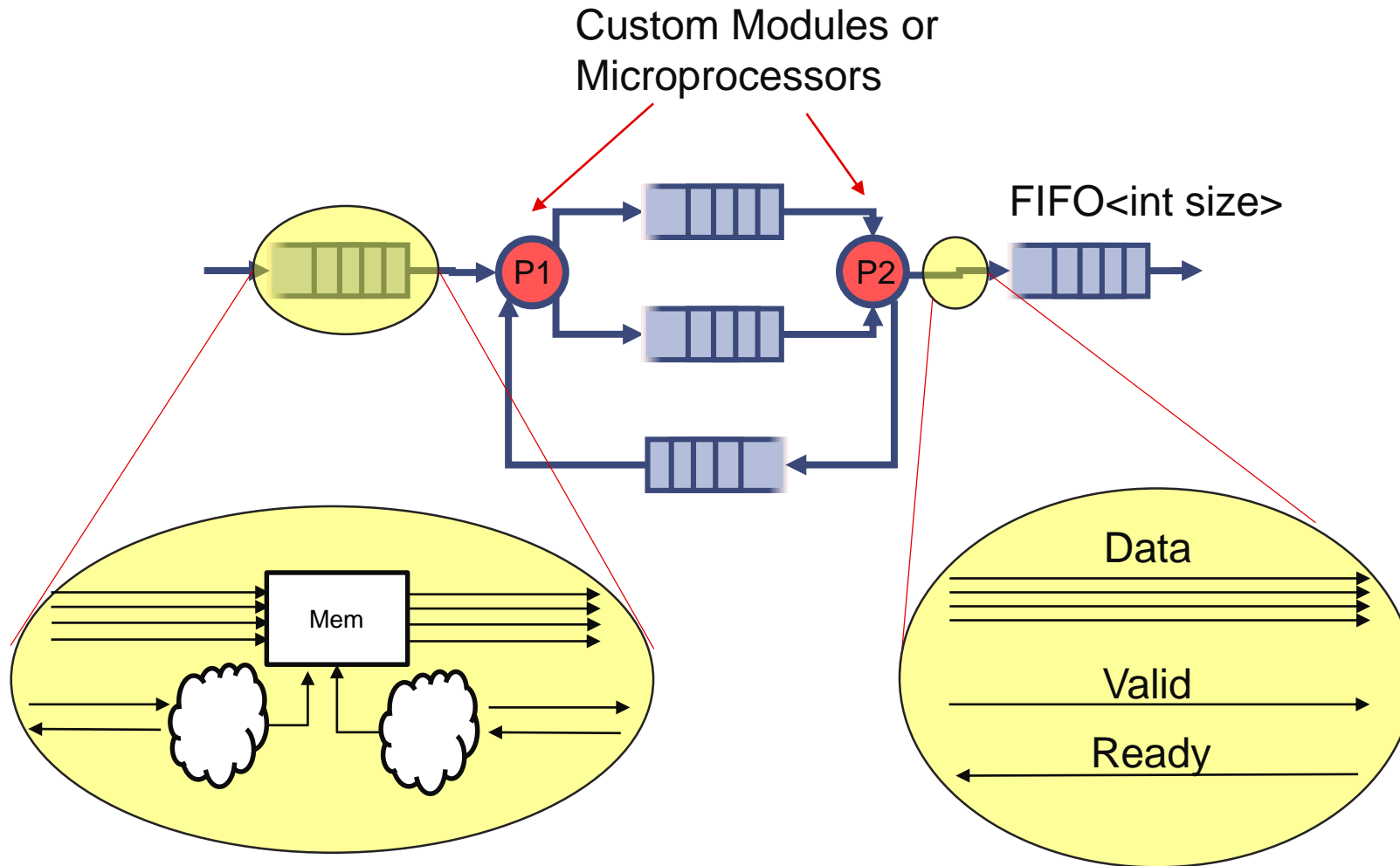
CIRCT RTL Dialects

- ▶ `hw` Dialect: basic hierarchy for circuit descriptions
- ▶ `comb` Dialect: combinational logic gates
- ▶ `seq` Dialect: sequential (registered) components

```
hw.module @top(%clk: i1, %rst: i1) -> (o: i1) {  
  %true = hw.constant true  
  %false = hw.constant false  
  %r0 = seq.compreg %0, %clk, %rst, %false : i1  
  %0 = comb.xor %r0, %true : i1  
  hw.output %r0 : i1  
}
```



Simple Hardware Representation



AXI4-stream protocol

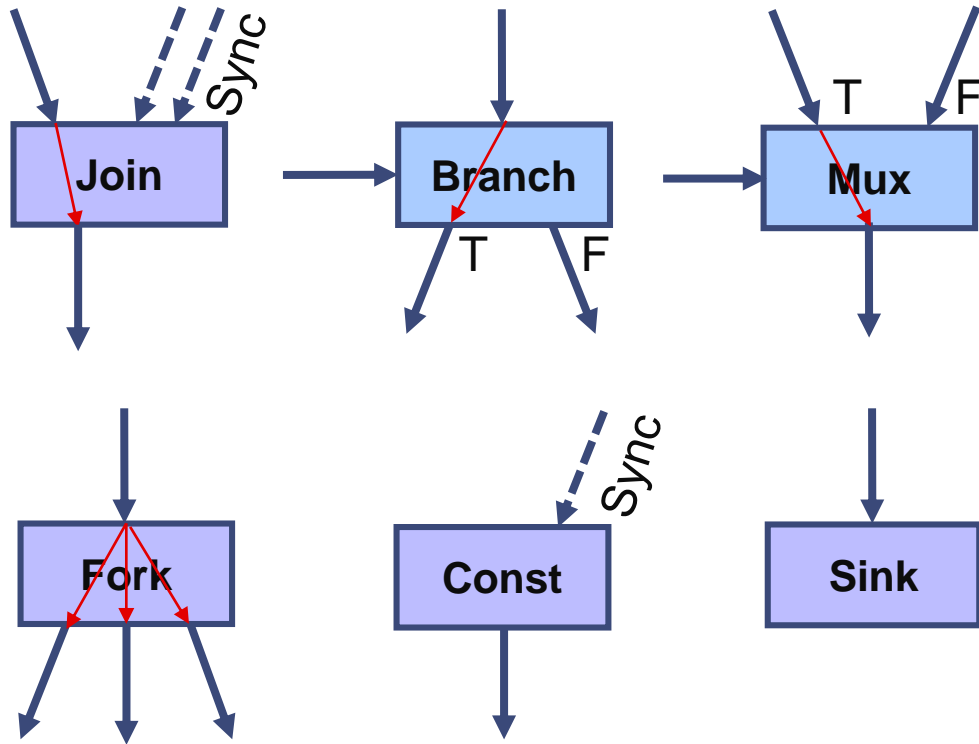
Data is transferred when Valid and Ready are both asserted in the same clock cycle

Valid can only be de-asserted when Ready is asserted and data is transferred

Asserting Valid cannot be dependent on waiting for Ready to be asserted

Latency Insensitive Elastic Circuits -> Simple Timing Closure

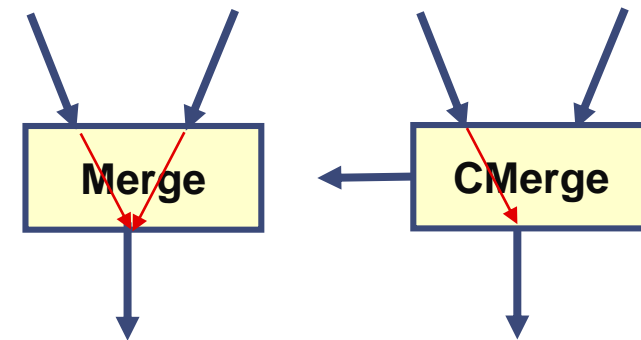
CIRCT Handshake Dialect



Deterministic operators

```
handshake.func @name (...) -> (...) {...}
```

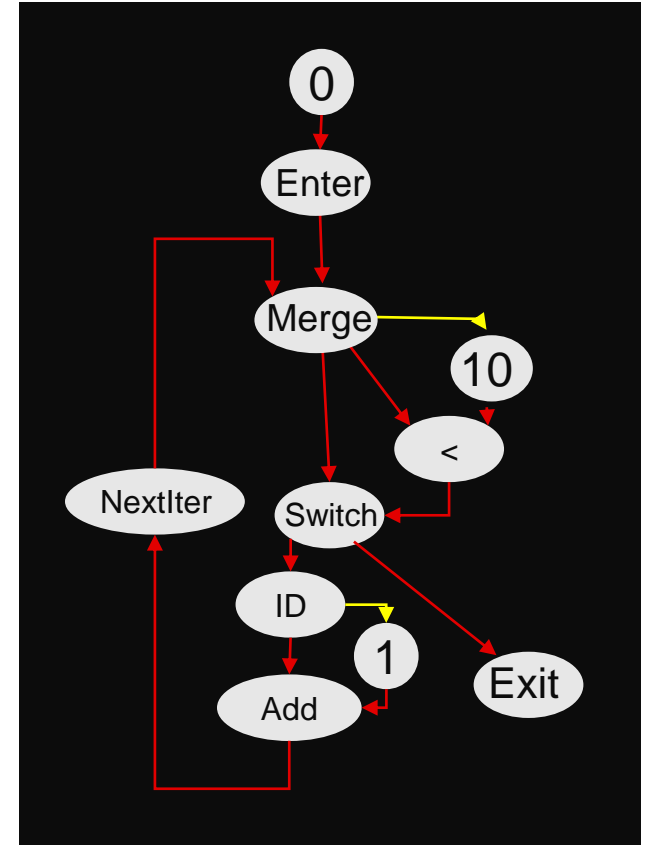
Graph-Region container



Non-Deterministic operators

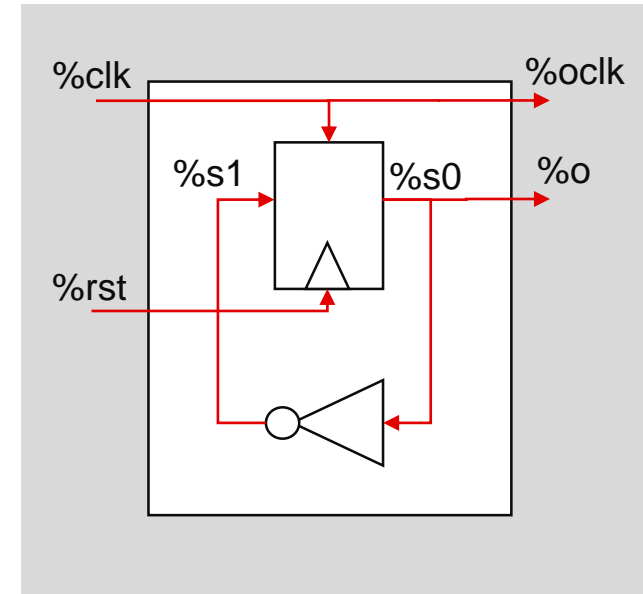
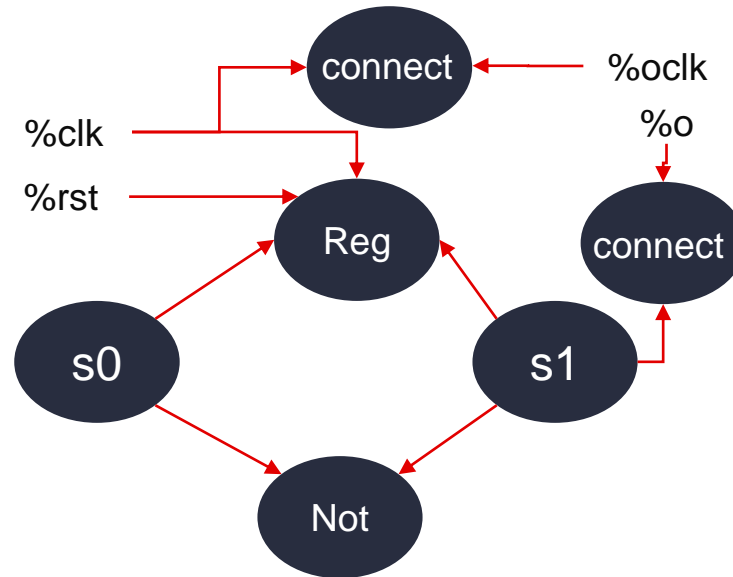
TensorFlow Graphs

```
module {
  tfg.graph {
    %Const, %ctl_1 = Const {value = dense<0> : tensor<i32>}
    %Enter, %ctl_0 = Enter(%Const)
    %NextIteration, %ctl_1 = NextIteration(%Add)
    %Merge:2, %ctl_2 = Merge(%Enter, %NextIteration)
    %Const_3, %ctl_4 = Const [%ctl_2] {value = dense<10> : tensor<i32>}
    %Less, %ctl_5 = Less(%Merge#0, %Const_3)
    %LoopCond, %ctl_6 = LoopCond(%Less
    %Switch:2, %ctl_7 = Switch(%Merge#0, %LoopCond)
    %Identity, %ctl_8 = Identity(%Switch#1)
    %Const_9, %ctl_10 = Const [%ctl_8] {value = dense<1> : tensor<i32>}
    %Add, %ctl_11 = Add(%Identity, %Const_9)
    %Exit, %ctl_12 = Exit(%Switch#0)
  }
}
```



What if we didn't have Graph Regions?

```
Module(%clk, %rst, %oclk, %o) {  
  %s0 = signal()  
  %s1 = signal()  
  Reg(%s0, %s1, %clk, %rst)  
  Not(%s1, %s0)  
  connect(%s1, %o)  
  connect(%clk, %oclk)  
}
```



- Signals are always operands -> *Less convenient to traverse*
- Explicit signal and connect operations -> *Less concise to write*
- 'connect' doesn't process data -> *Less consistent for optimizations*

Note: This style is sometimes necessary, e.g., for bus wires with multiple drivers

Conclusion

- ▶ **Graph Regions** are an important feature of MLIR that makes representing some kinds of designs easier
- ▶ Several out-of-tree dialects are using it to directly represent programs with concurrent behavior
- ▶ I would love to discuss new uses!