XILINX

Handshake-based HLS in CIRCT

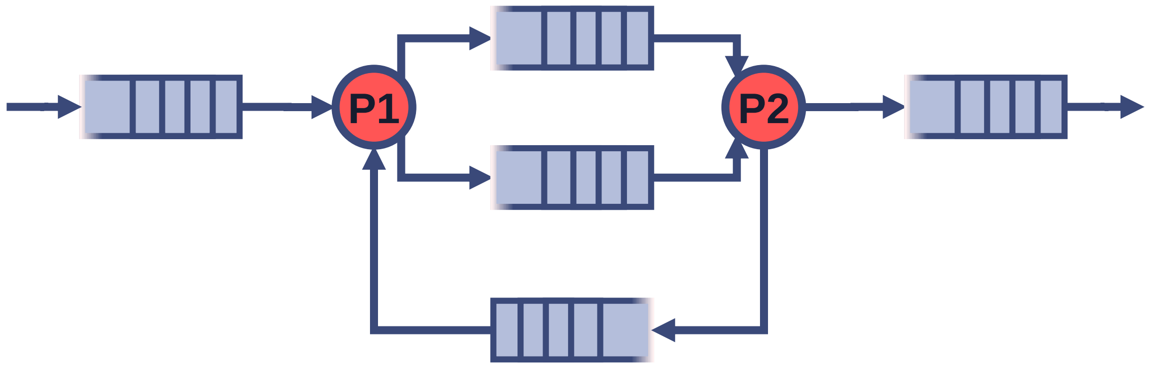
Hanchen Ye (PhD student from UIUC)
Compiler Intern in Xilinx
Advisor: Stephen Neuendorffer
Aug.19, 2020

Outline

- Pipeline Operation
  - Motivation
  - Key Features
- Dialect Conversions and Transforms
  - Overview
  - Standard to StaticLogic Conversion
  - Handshake Dialect Transform: Buffer Insertion
  - Handshake to FIRRTL Conversion
- Next Steps

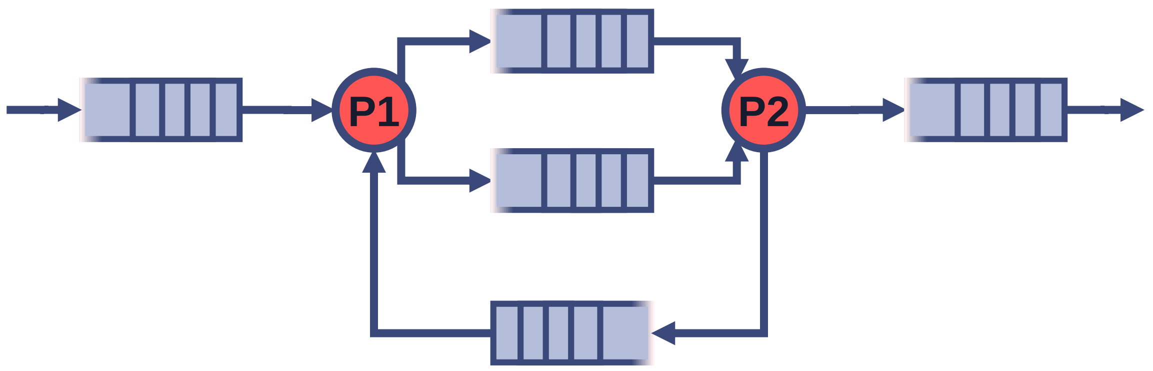
Background

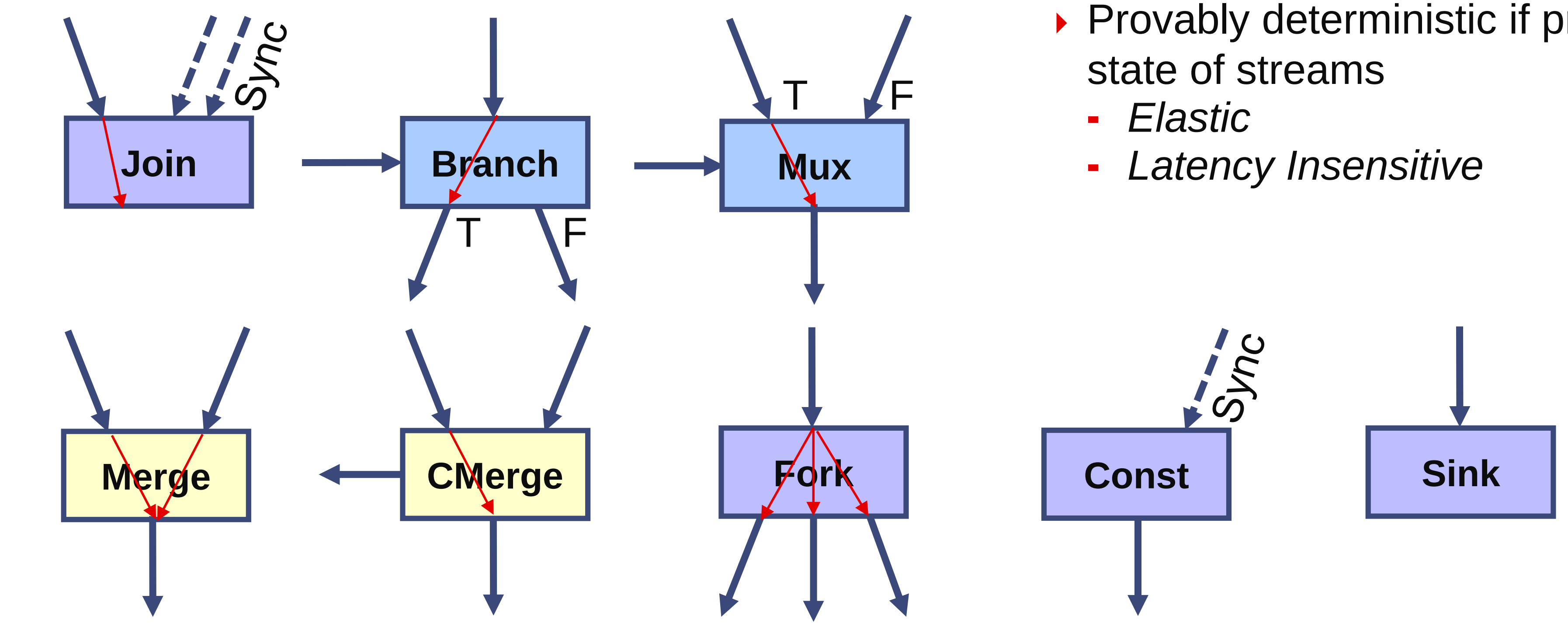
Handshake Dialect (1)



- Processes communicate through stream interfaces (e.g. AXI4-Stream)
- Interfaces connected by single-reader single-writer FIFOs, which are logically unbounded
- Processes can access interfaces in any order
- Provably deterministic if processes cannot test state of streams
  - *Elastic*
  - *Latency Insensitive*

Handshake Dialect (2)



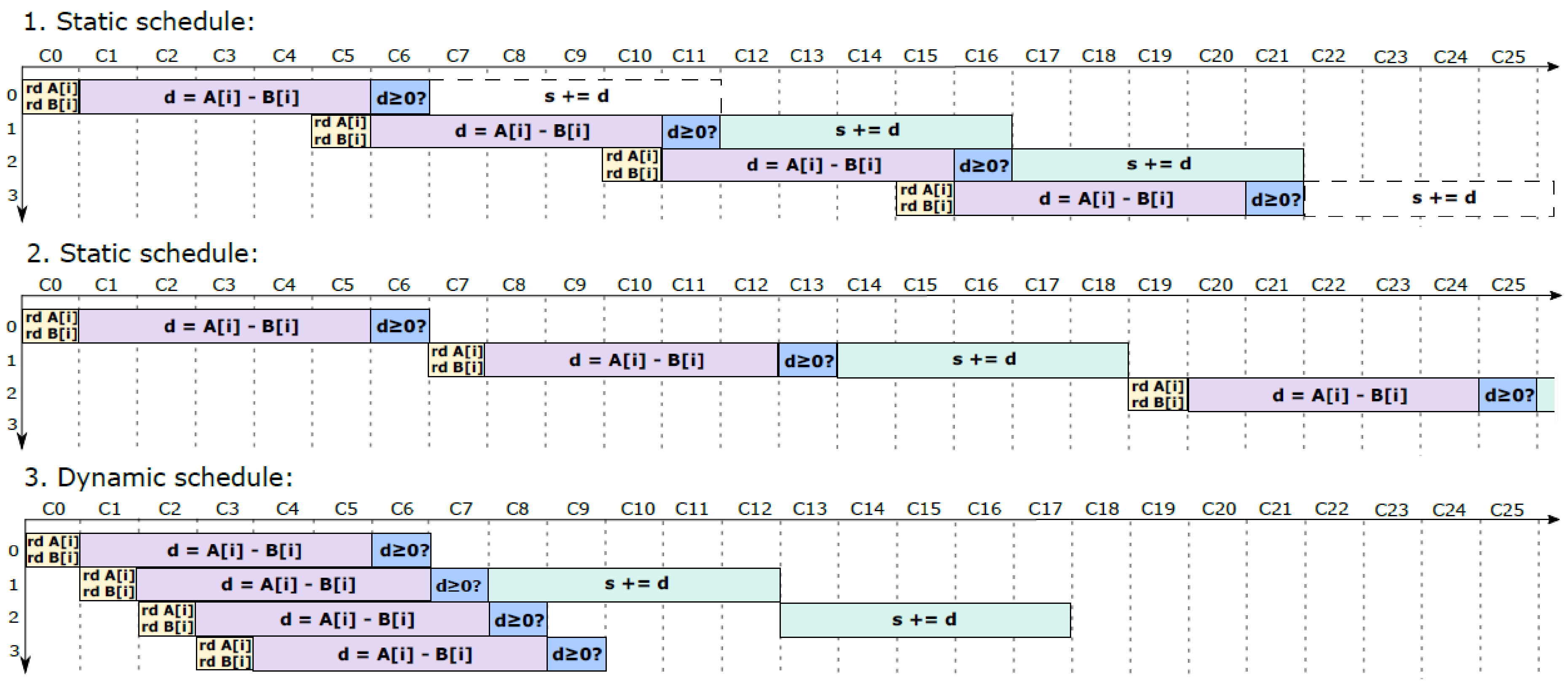


- Processes communicate through stream interfaces (e.g. AXI4-Stream)
- Interfaces connected by single-reader single-writer FIFOs, which are logically unbounded
- Processes can access interfaces in any order
- Provably deterministic if processes cannot test state of streams
  - *Elastic*
  - *Latency Insensitive*

Handshake Dialect (3)

```
float d, s = 0.0;
int i;
for (int i=0; i<100; i++){
      d = A[i] - B[i];
      if (d >= 0)
            s += d;
}
 A[0]=1.0; B[0]=3.0;
 A[1]=4.0; B[1]=3.0;
 A[2]=2.0; B[2]=2.0;
 A[3]=4.0; B[3]=5.0;
          ⋮
```

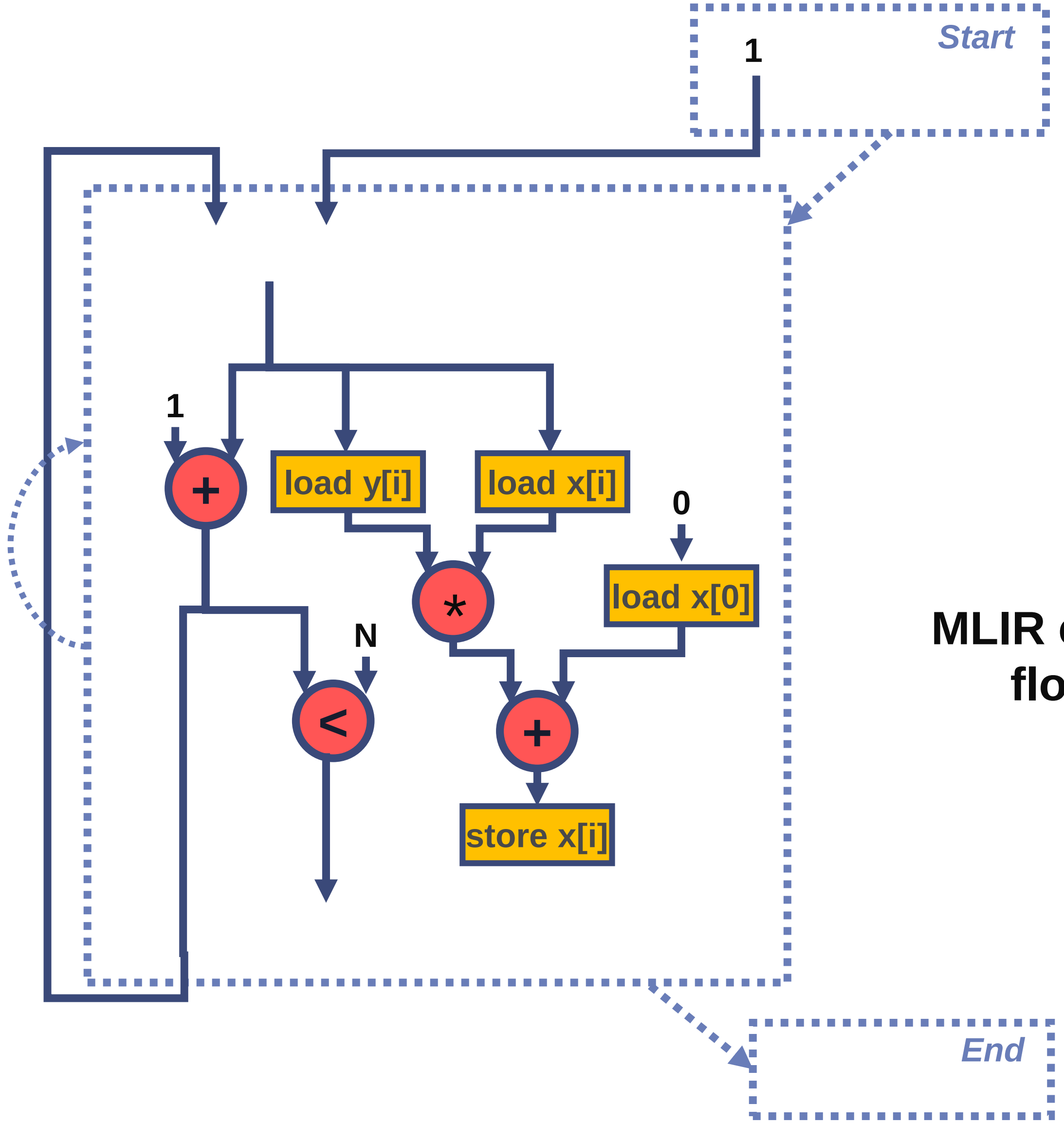
(a)



(b)

* Josipović, Lana, Radhika Ghosal, and Paolo Ienne. "Dynamically scheduled high-level synthesis." *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2018.

Standard to Handshake Conversion (1)

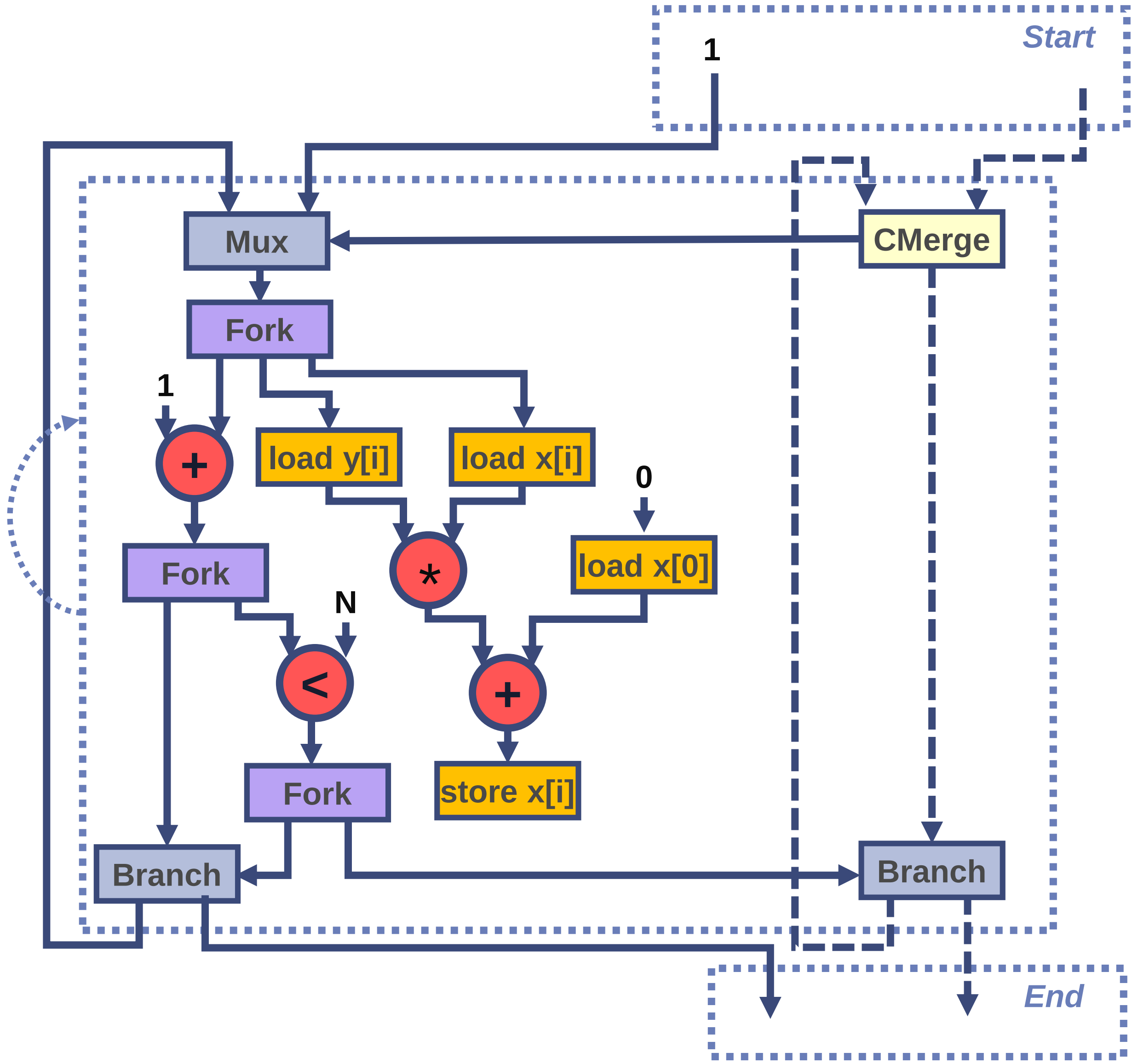


MLIR control/data flow graph

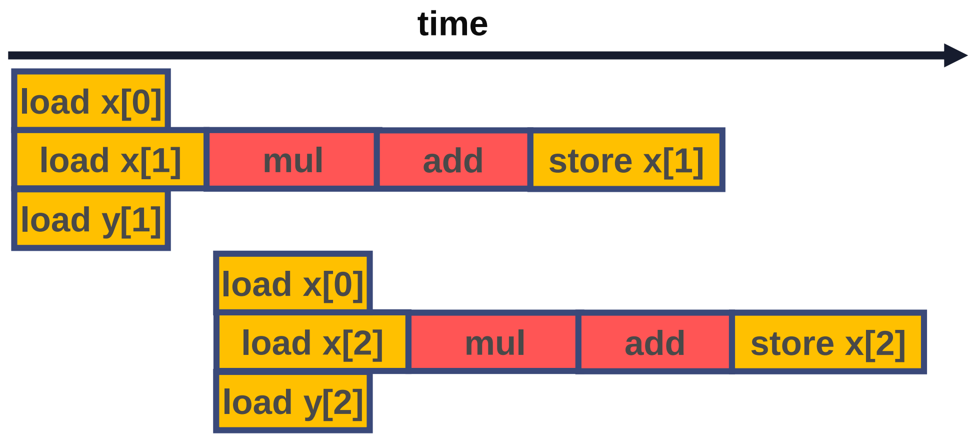
```
for (i = 1; i < N; ++i)
    x[i] = x[0] + x[i] * y[i];
```

```
  ... ...
  br ^bb1(%c1_0 : index)
^bb1(%2: index):
  %c11 = constant 11 : index
  %3 = cmpi "slt", %2, %c11 : index
  cond_br %3, ^bb2, ^bb3
^bb2:
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %4 = load %0[%c0] : memref<16xindex>
  %5 = load %0[%2] : memref<16xindex>
  %6 = load %1[%2] : memref<16xindex>
  %7 = muli %5, %6 : index
  %8 = addi %7, %4 : index
  store %8, %0[%2] : memref<16xindex>
  %9 = addi %2, %c1 : index
  br ^bb1(%9 : index)
^bb3:
  return
```

Standard to Handshake Conversion (2)

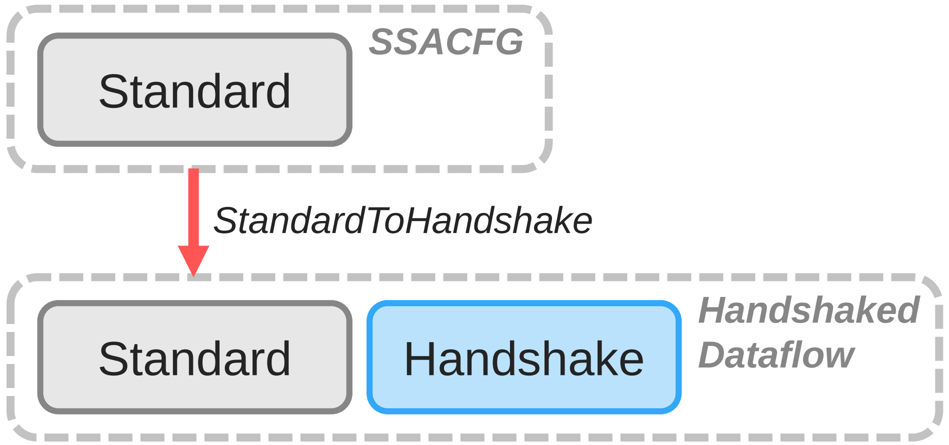


```
for (i = 1; i < N; ++i)
    x[i] = x[0] + x[i] * y[i];
```



Concurrent execution

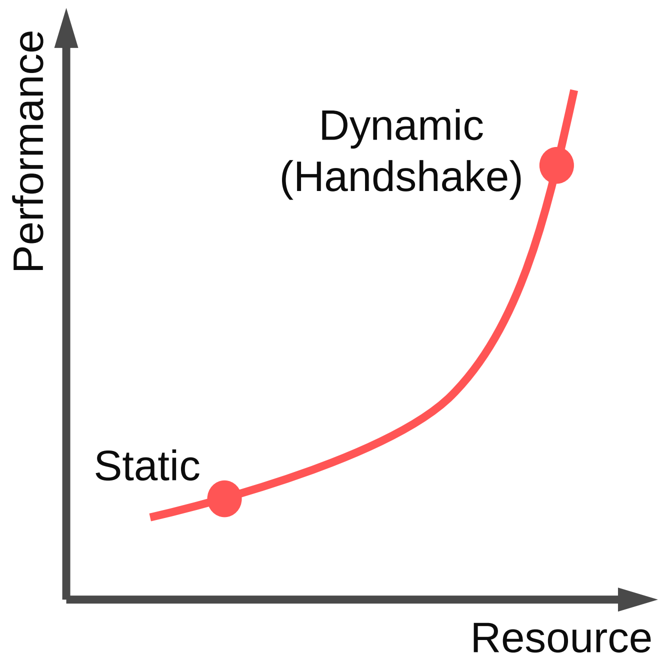
Standard to Handshake Conversion (3)



- Standard to Handshake Conversion

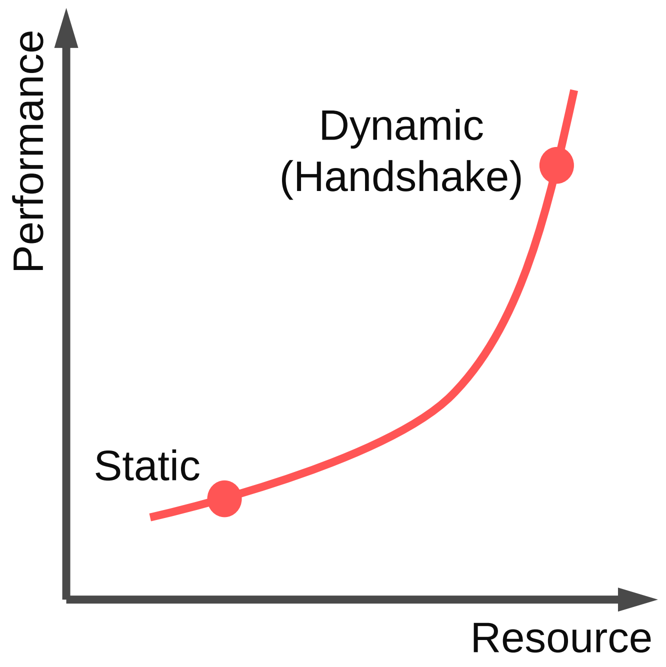
Pipeline Operation

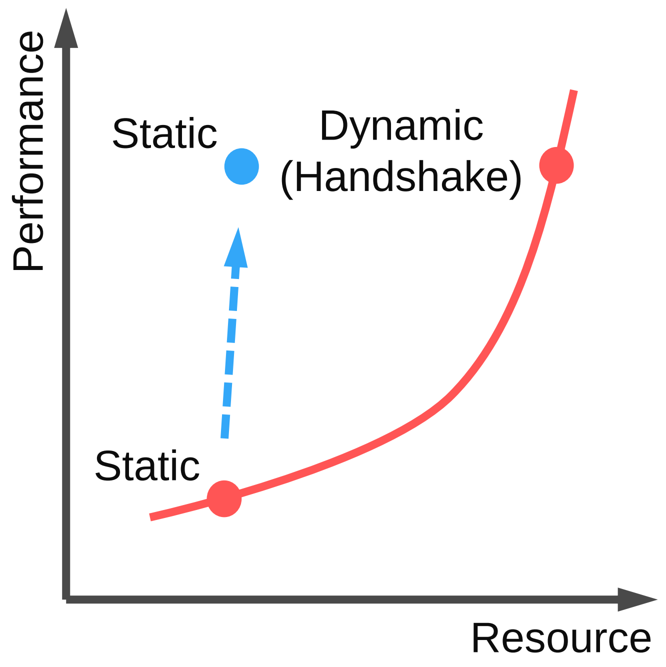
Motivation (1)



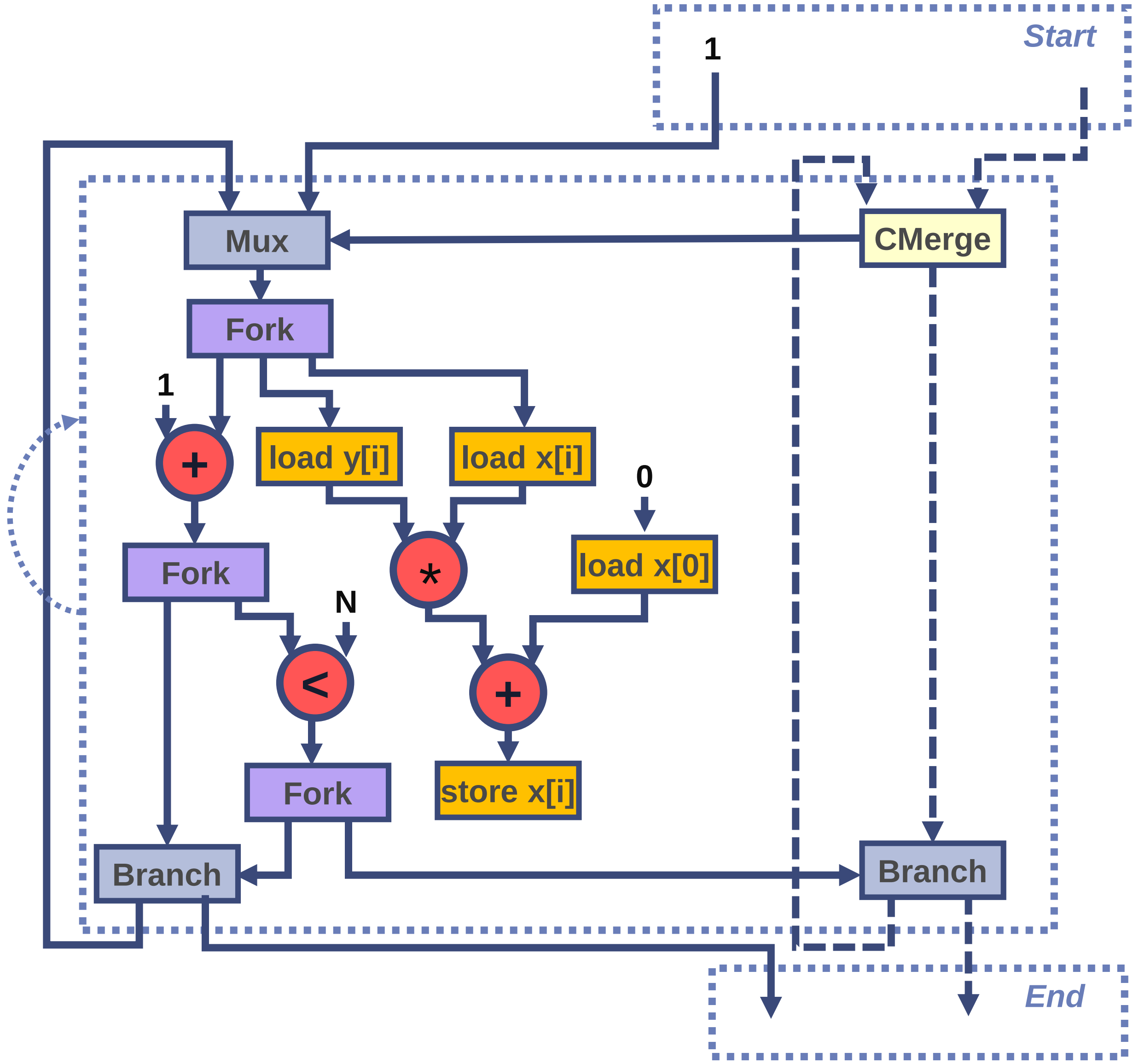
Motivation (2)

- In some cases, dynamic scheduling is Pareto dominated by static scheduling (e.g. perfect loop with the absence of conditional operations).

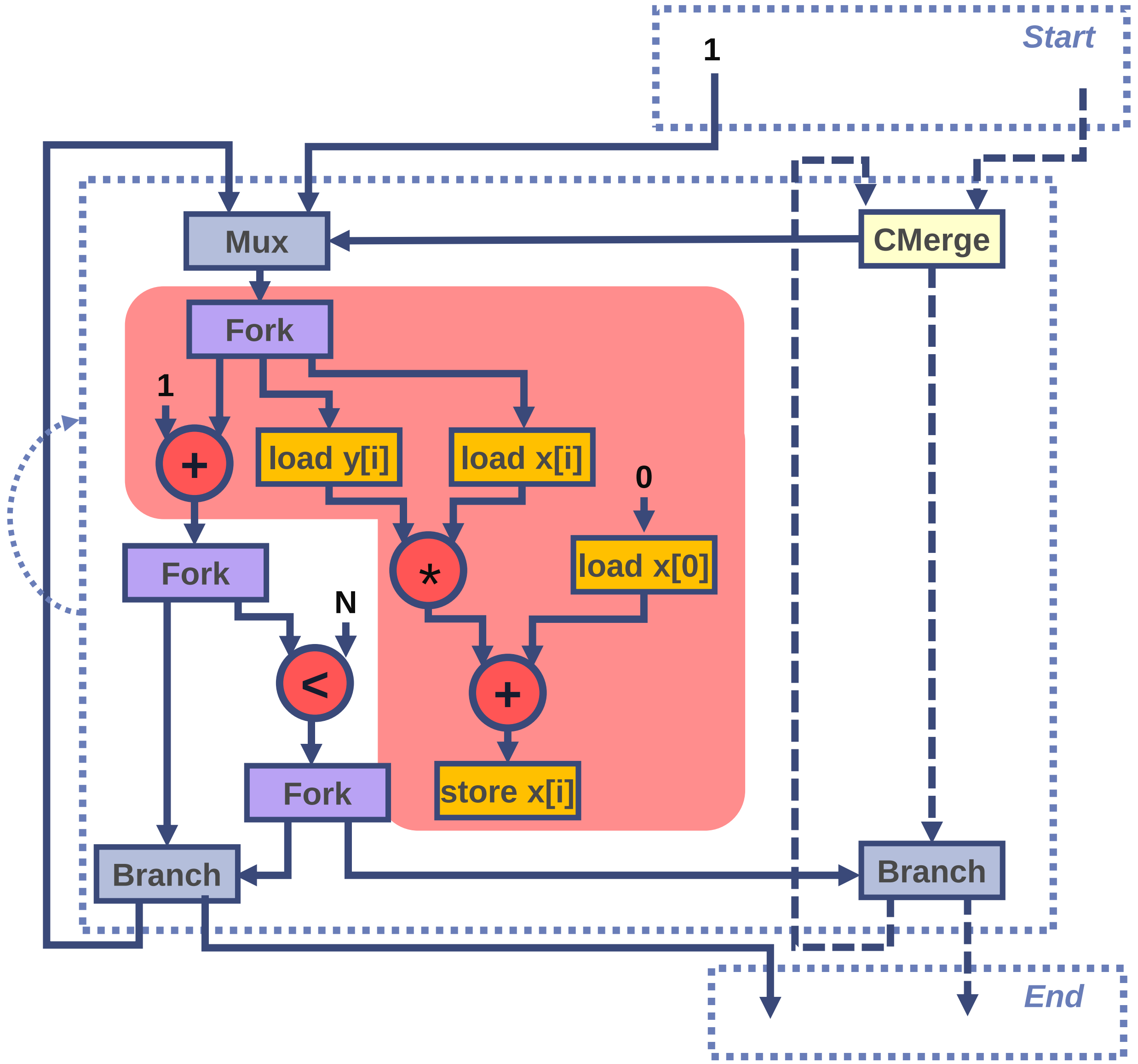




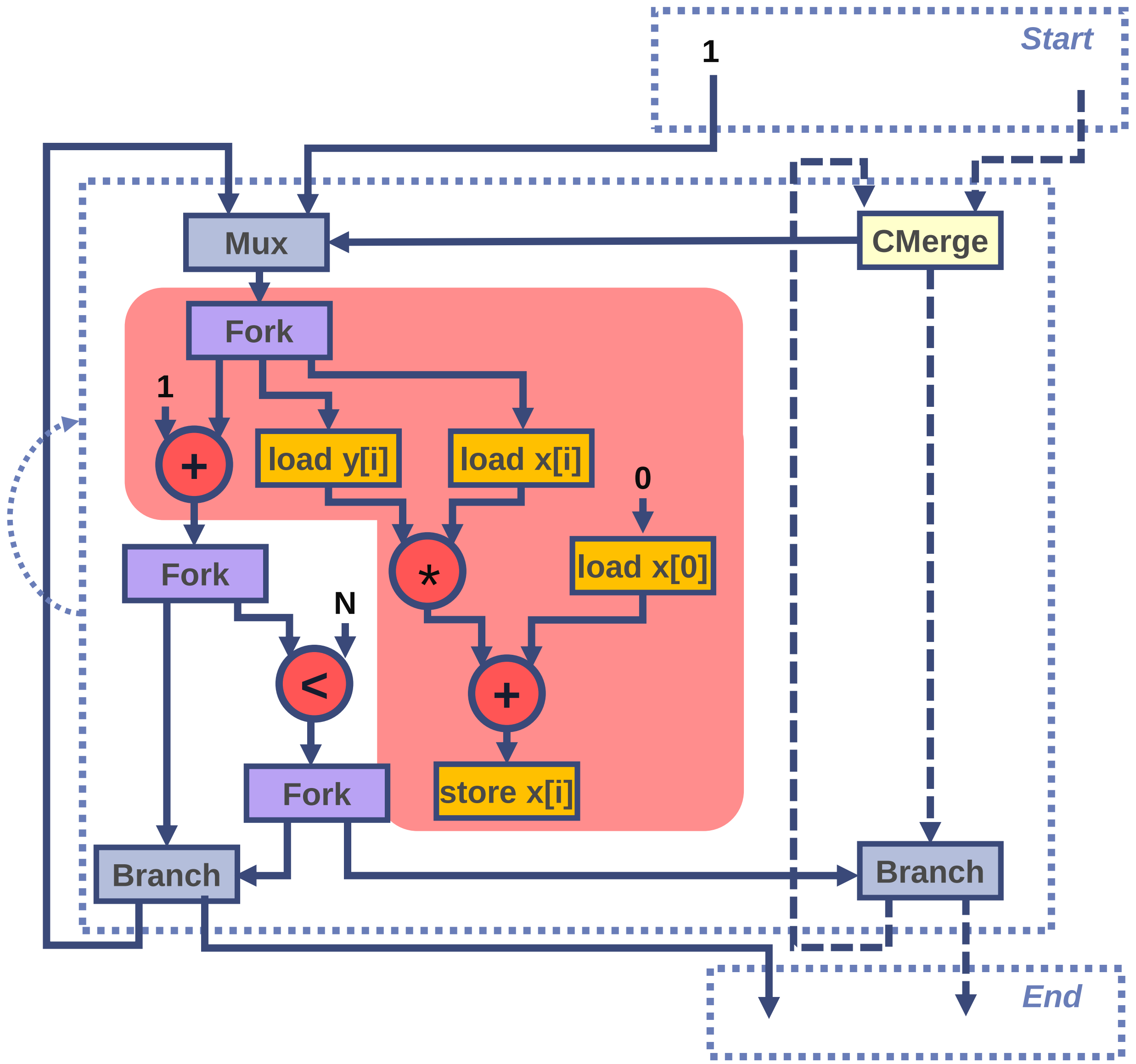
Motivation (3)



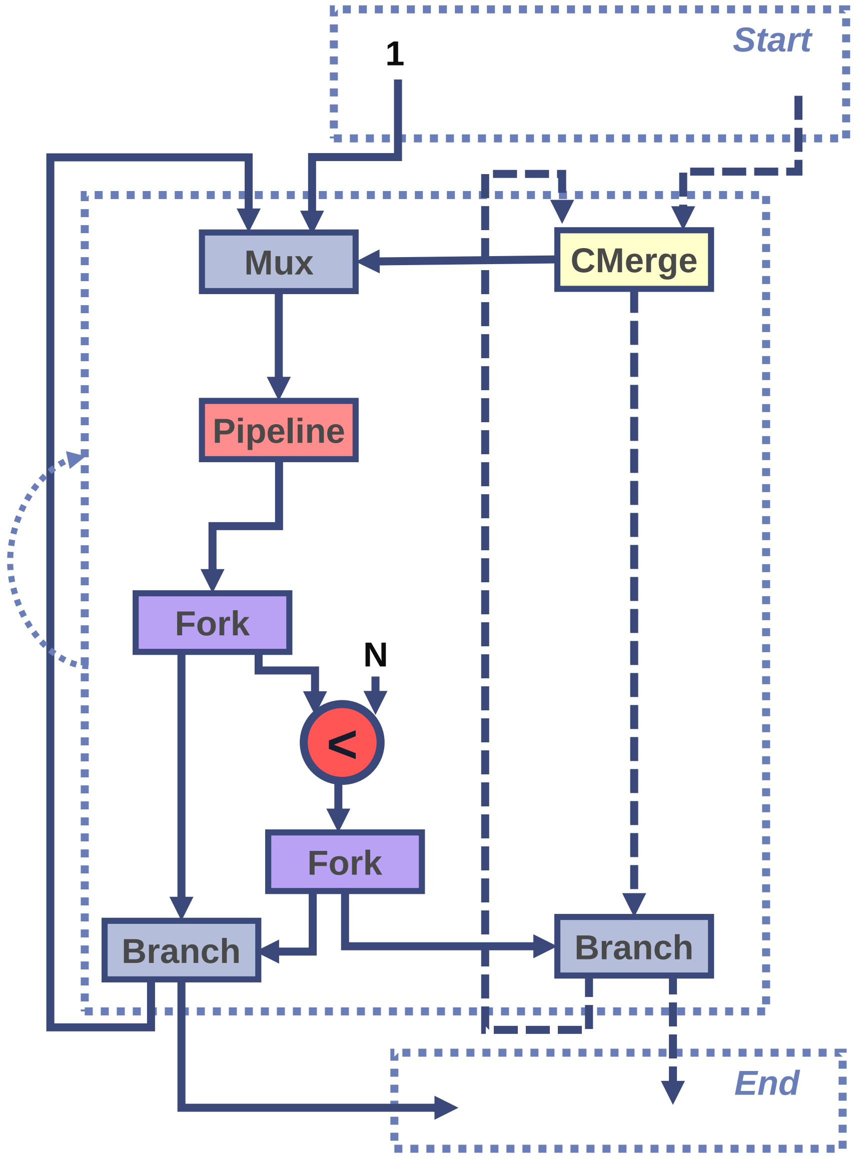
Motivation (4)



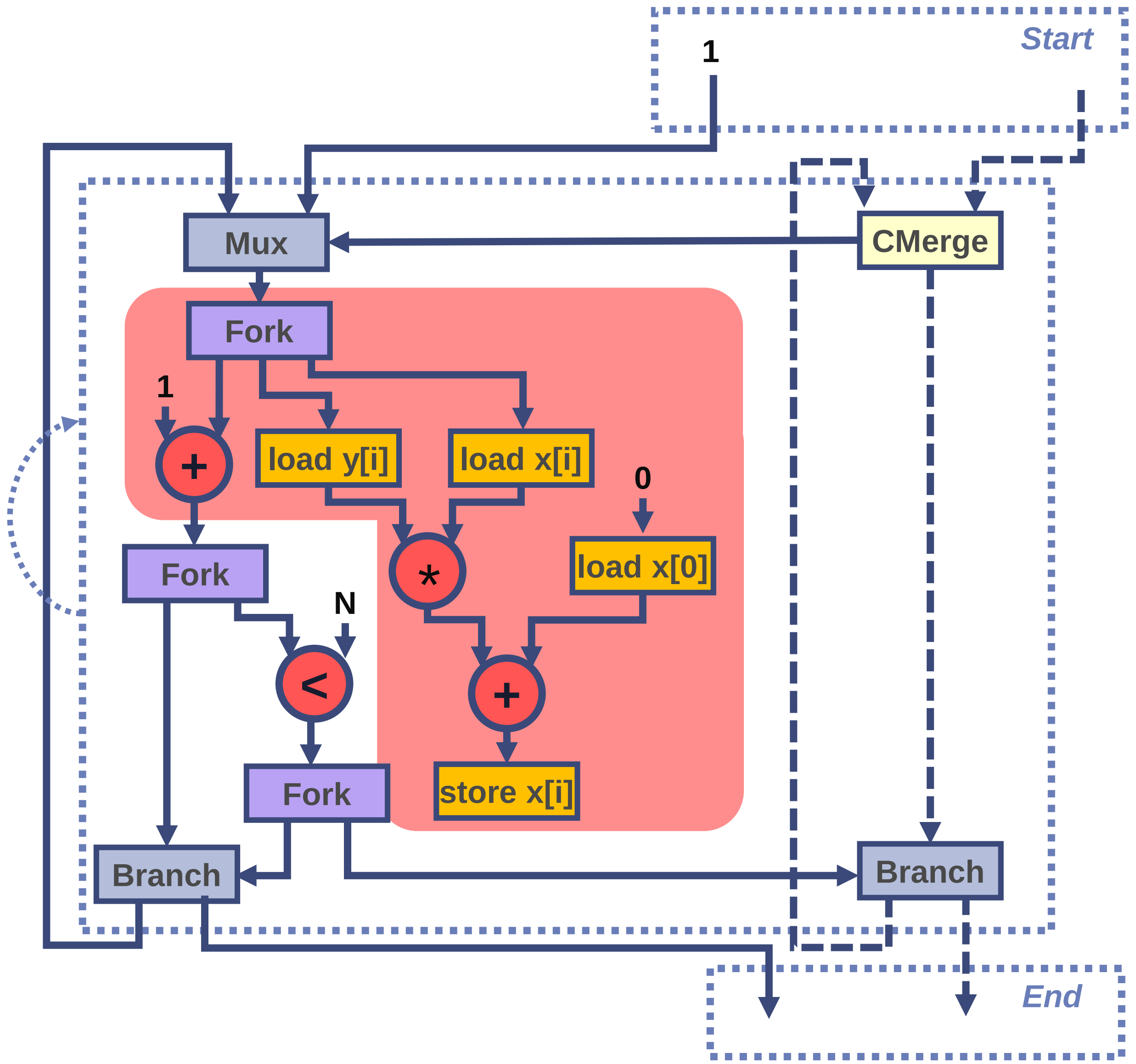
Motivation (5)



V.S.



Motivation (6)



- Reduce the resource overhead of fine-grained handshake interfaces
- Allow to support late input or early output
- Local static scheduling + global dynamic scheduling => coarse-grained handshaked dataflow

Key Features (1)

- Belongs to a new StaticLogic dialect

```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  br ^bb1
^bb1:
  %5 = addi %arg3, %4 : index
  %6 = addi %arg5, %4 : index
  br ^bb2
^bb2:
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Key Features (2)

- Belongs to a new StaticLogic dialect
- Is isolated from above

```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  br ^bb1
^bb1:
  %5 = addi %arg3, %4 : index
  %6 = addi %arg5, %4 : index
  br ^bb2
^bb2:
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Key Features (3)

- Belongs to a new StaticLogic dialect
- Is isolated from above
- Each block represents one pipeline stage: bb0, bb1, bb2

```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  br ^bb1
^bb1:
  %5 = addi %arg3, %4 : index
  %6 = addi %arg5, %4 : index
  br ^bb2
^bb2:
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Key Features (3)

- Belongs to a new StaticLogic dialect
- Is isolated from above
- Each block represents one pipeline stage: bb0, bb1, bb2

```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  %5 = addi %arg3, %4 : index
  br ^bb1
^bb1:
  %6 = addi %arg5, %4 : index
  br ^bb2
^bb2:
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Key Features (3)

- Belongs to a new StaticLogic dialect
- Is isolated from above
- Each block represents one pipeline stage: bb0, bb1, bb2

```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  %5 = addi %arg3, %4 : index
  br ^bb1
^bb1:
  %6 = addi %arg5, %4 : index
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Key Features (4)

- Belongs to a new StaticLogic dialect
- Is isolated from above
- Each block represents one pipeline stage: bb0, bb1, bb2
- We can easily explore the design space of pipeline mapping

```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  %5 = addi %arg3, %4 : index
  br ^bb1
^bb1:
  %6 = addi %arg5, %4 : index
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Key Features (5)

- Belongs to a new StaticLogic dialect
- Is isolated from above
- Each block represents one pipeline stage: bb0, bb1, bb2
- We can easily explore the design space of pipeline mapping
- Is a node of the handshaked dataflow
- Communicates with other nodes through stream interfaces

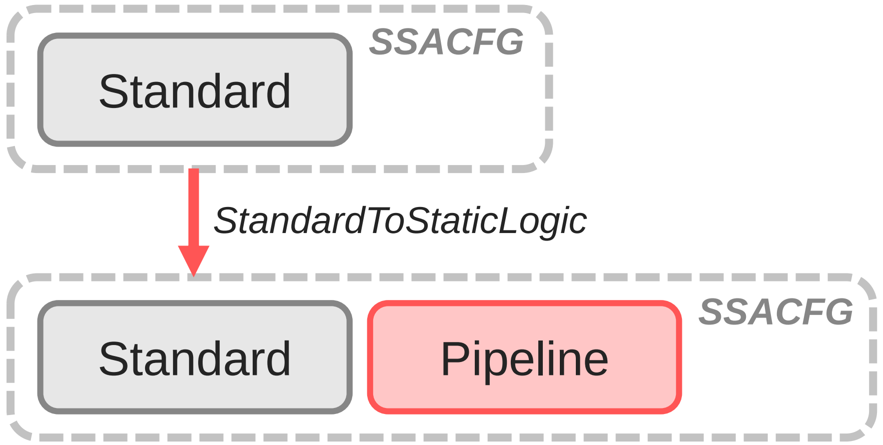
```
... ...
%1:2 = "handshake.fork"(%0) : (index) ->
  (index, index)
%2 = "handshake.merge"(%arg1) : (index) -> index

%3 = "staticlogic.pipeline"(%1#0, %1#1, %2) ( {
^bb0(%arg3: index, %arg4: index, %arg5: index):
  %4 = addi %arg3, %arg4 : index
  %5 = addi %arg3, %4 : index
  br ^bb1
^bb1:
  %6 = addi %arg5, %4 : index
  %7 = addi %5, %6 : index
  "staticlogic.return"(%7) : (index) -> ()
}) : (index, index, index) -> index

handshake.return %3, %arg2 : index, none
... ...
```

Dialect Conversions and Transforms

Overview (1)

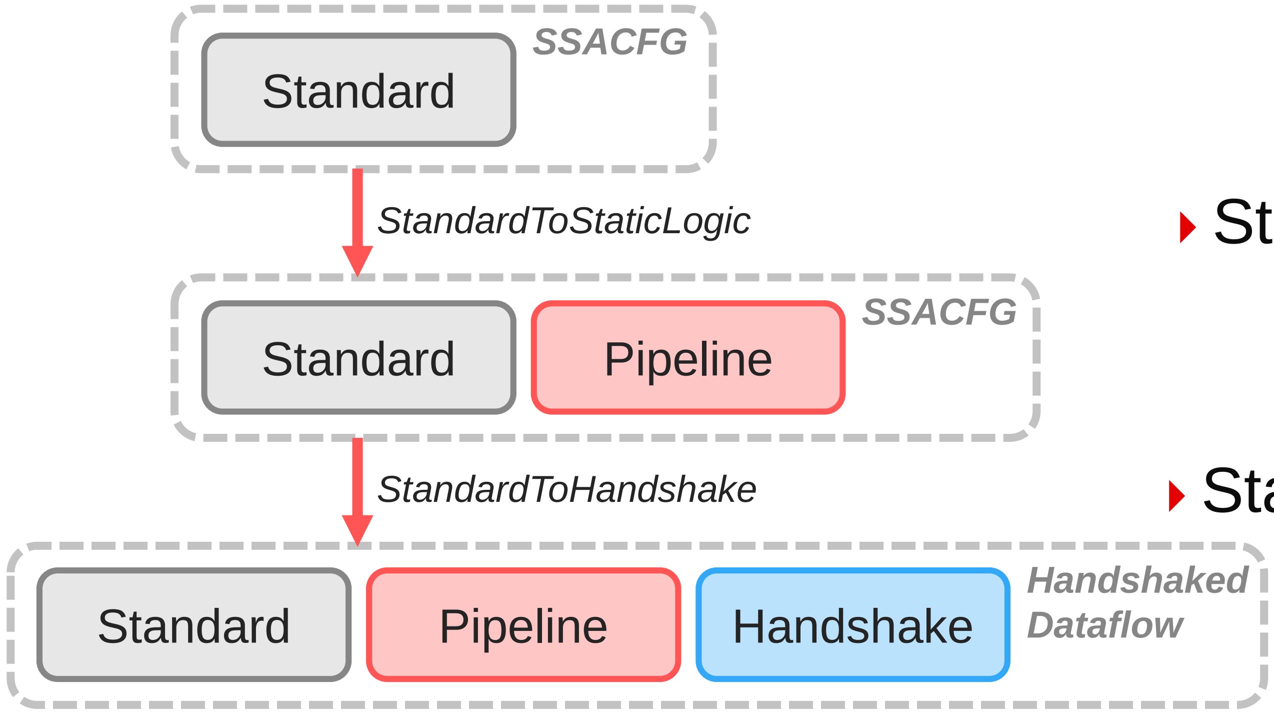
SSACFG
Standard

Overview (2)



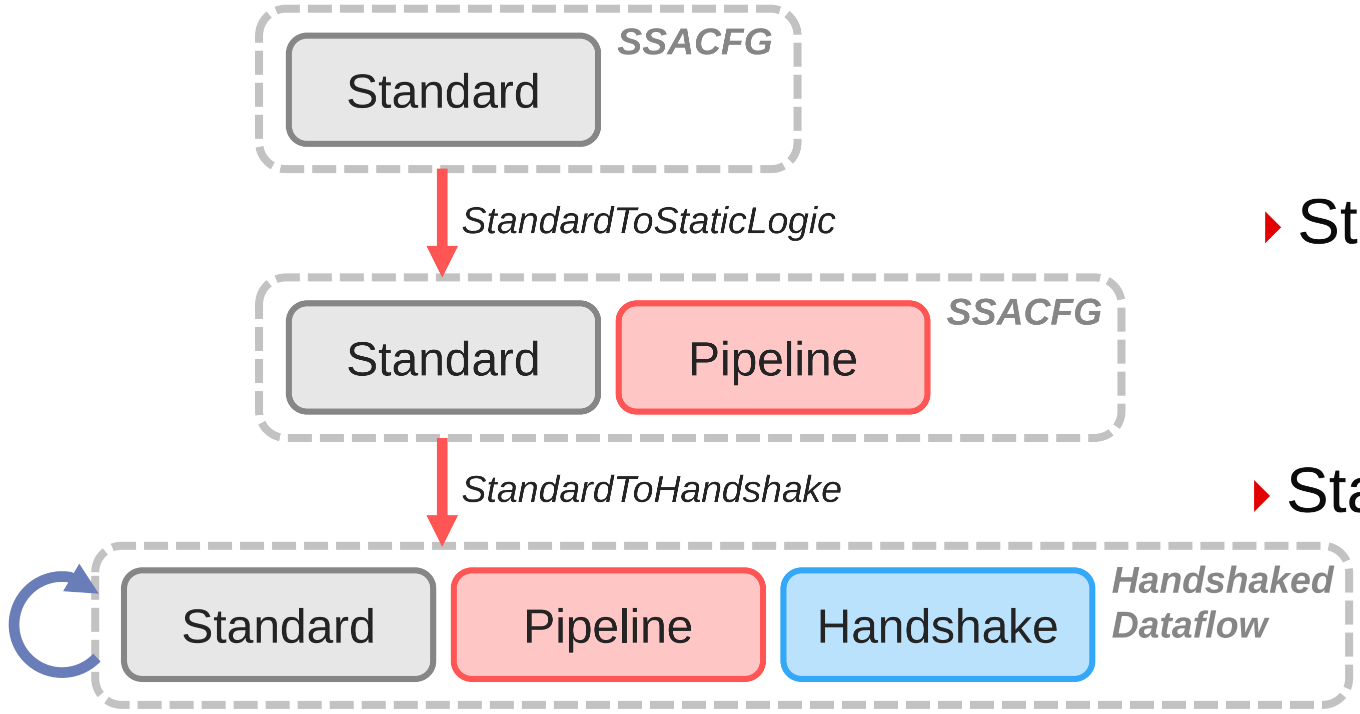
- Standard to StaticLogic Conversion

Overview (3)



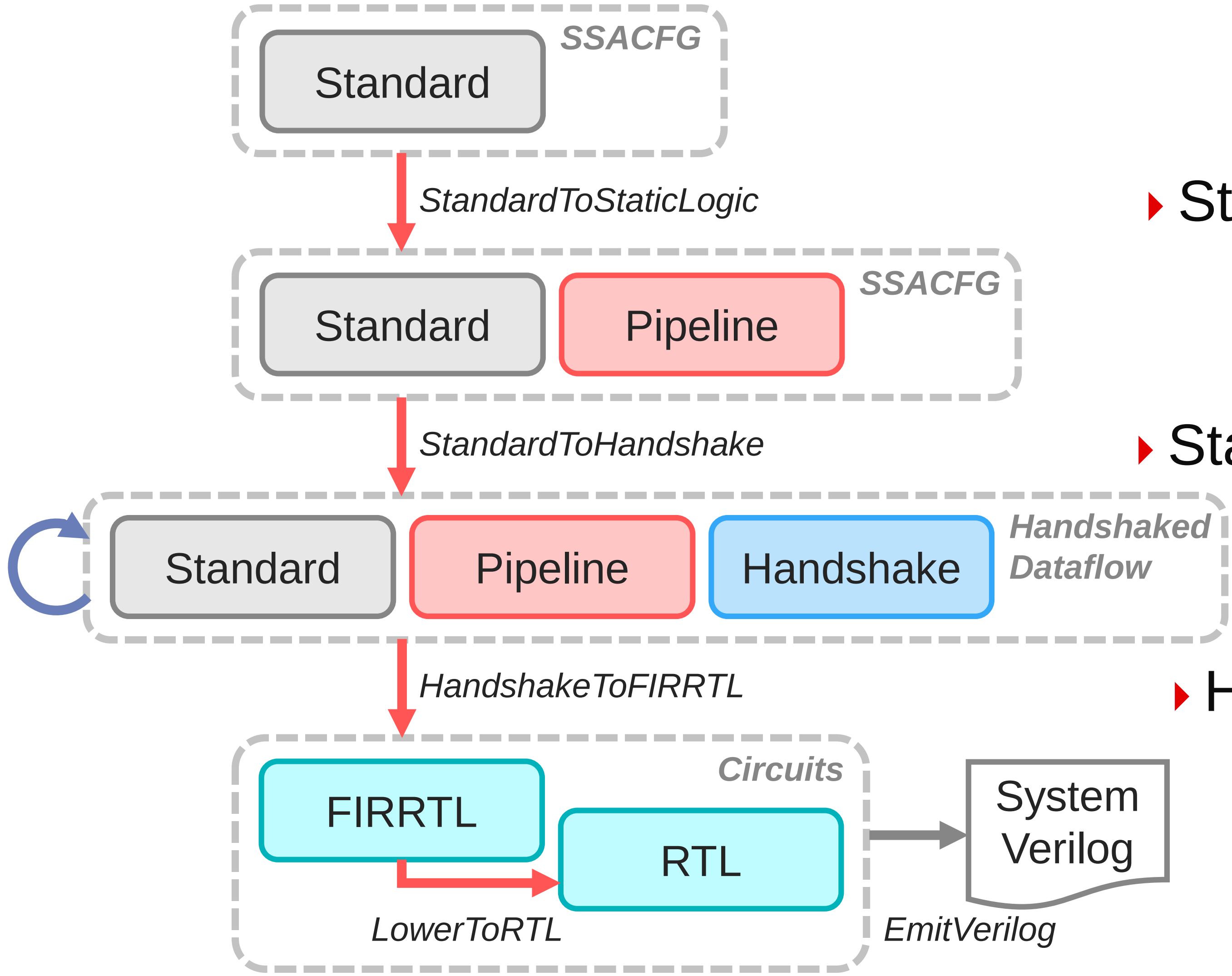
- Standard to StaticLogic Conversion
- Standard to Handshake Conversion

Overview (4)



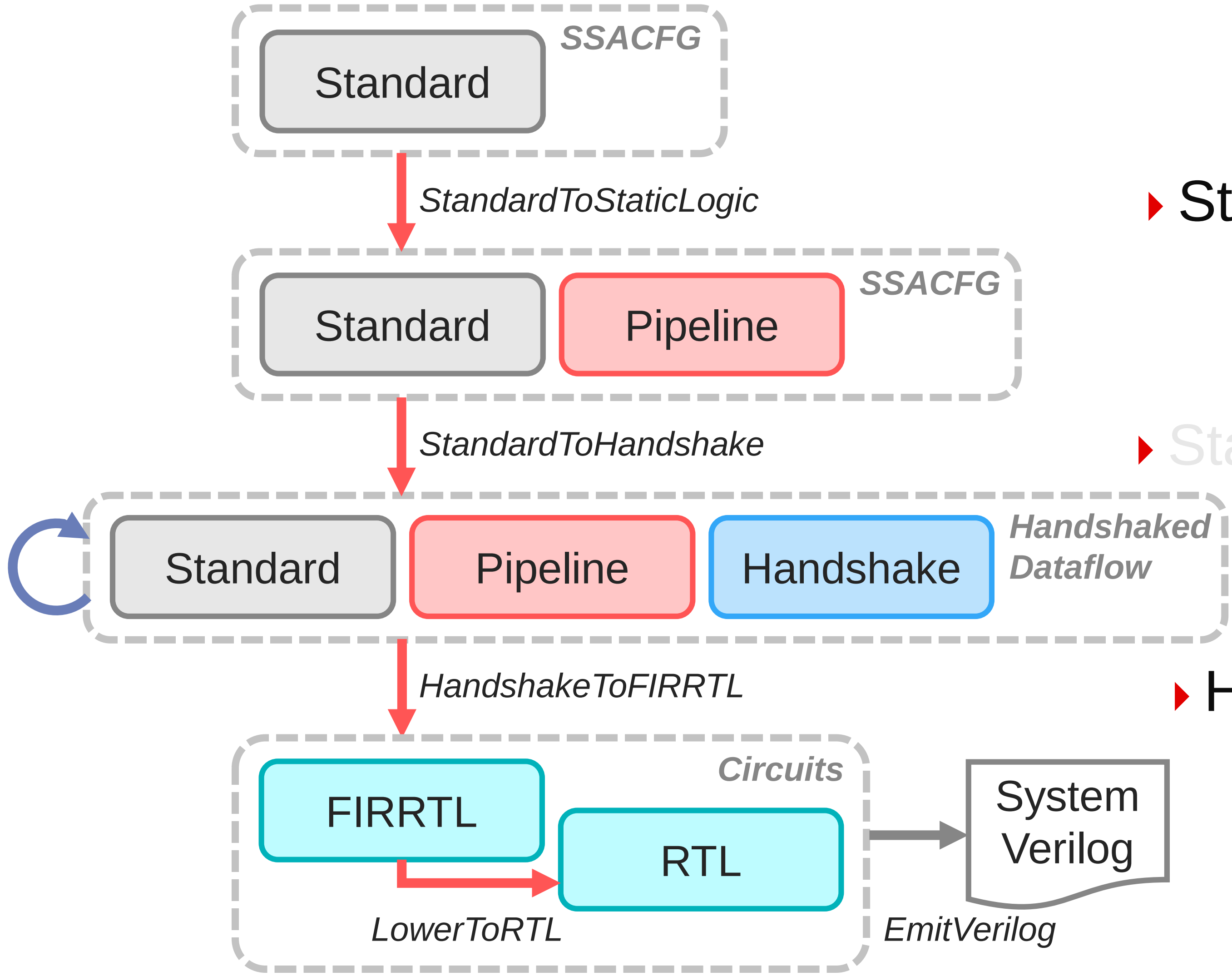
- Standard to StaticLogic Conversion
- Standard to Handshake Conversion
- Handshake Dialect Transform

Overview (5)



- Standard to StaticLogic Conversion
- Standard to Handshake Conversion
- Handshake Dialect Transform
- Handshake to FIRRTL Conversion

Overview (5)



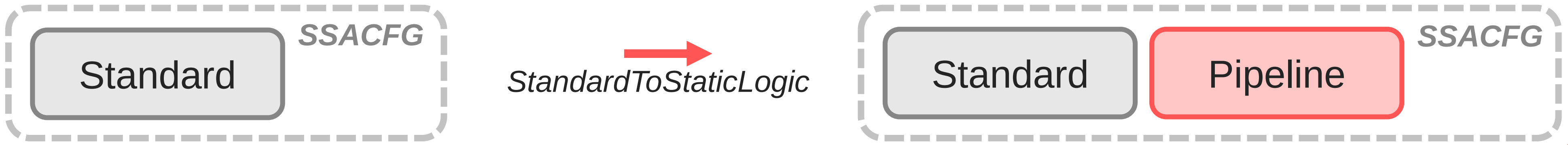
- Standard to StaticLogic Conversion
- Standard to Handshake Conversion
- Handshake Dialect Transform
- Handshake to FIRRTL Conversion

Standard to StaticLogic Conversion

```
for (i = 1; i < N; ++i)
    x[i] = x[0] + x[i] * y[i];
```

```
  ... ...
^bb2:
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %4 = load %0[%c0] : memref<16xindex>
  %5 = load %0[%2] : memref<16xindex>
  %6 = load %1[%2] : memref<16xindex>
  %7 = muli %5, %6 : index
  %8 = addi %7, %4 : index
  store %8, %0[%2] : memref<16xindex>
  %9 = addi %2, %c1 : index
  br ^bb1(%9 : index)
^bb3:
  ... ...
```

```
  ... ...
^bb2:
  %3 = "staticlogic.pipeline"(%0#2, %1, %0#3) ( {
  ^bb0(%arg0: memref<16xindex>, %arg1: index,
%arg2: memref<16xindex>):
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %4 = load %arg0[%c0] : memref<16xindex>
    %5 = load %arg0[%arg1] : memref<16xindex>
    %6 = load %arg2[%arg1] : memref<16xindex>
    %7 = muli %5, %6 : index
    %8 = addi %7, %4 : index
    store %8, %arg0[%arg1] : memref<16xindex>
    %9 = addi %arg1, %c1 : index
    "staticlogic.return"(%9) : (index) -> ()
  }) : (memref<16xindex>, index, memref<16xindex>)
 -> index
  br ^bb1(%3 : index)
^bb3:
  ... ...
```

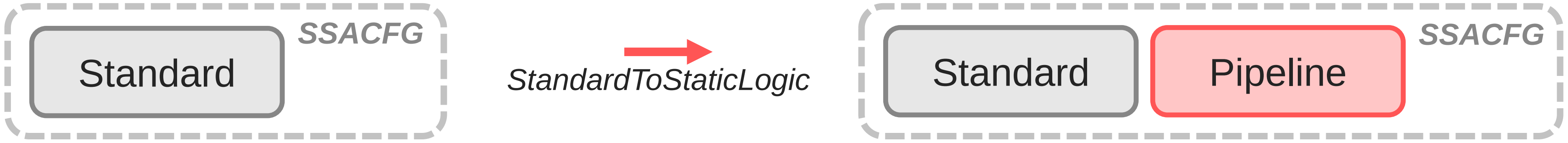


Standard to StaticLogic Conversion

```
for (i = 1; i < N; ++i)
    x[i] = x[0] + x[i] * y[i];
```

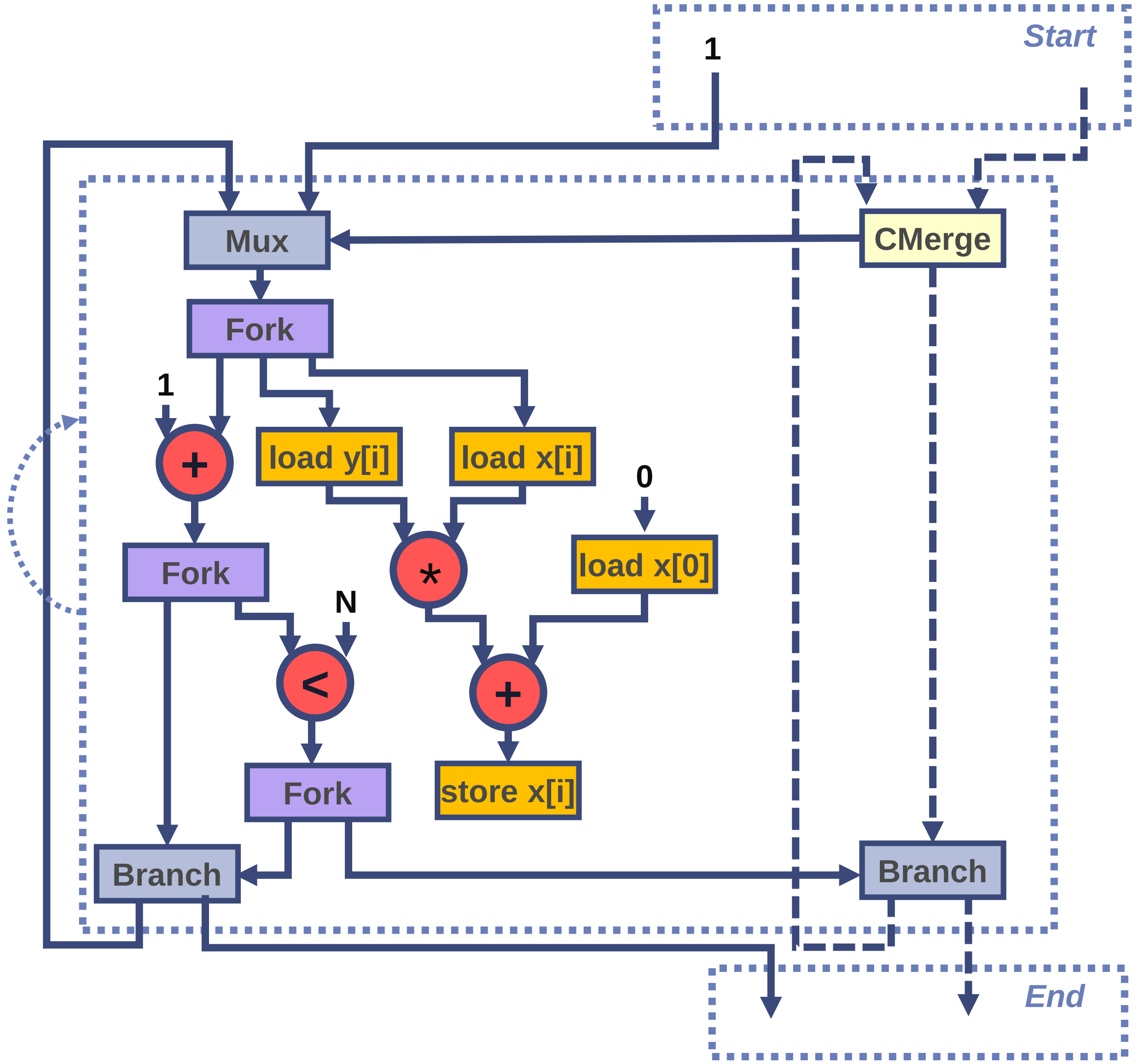
```
  ... ...
^bb2:
  %c0 = constant 0 : index
  %c1 = constant 1 : index
  %4 = load %0[%c0] : memref<16xindex>
  %5 = load %0[%2] : memref<16xindex>
  %6 = load %1[%2] : memref<16xindex>
  %7 = muli %5, %6 : index
  %8 = addi %7, %4 : index
  store %8, %0[%2] : memref<16xindex>
  %9 = addi %2, %c1 : index
  br ^bb1(%9 : index)
^bb3:
  ... ...
```

```
  ... ...
^bb2:
  %3 = "staticlogic.pipeline"(%0#2, %1, %0#3) ( {
    ... ... //pipeline region
} ) : (memref<16xindex>, index, memref<16xindex>) -> index
  br ^bb1(%3 : index)
^bb3:
  ... ...
```



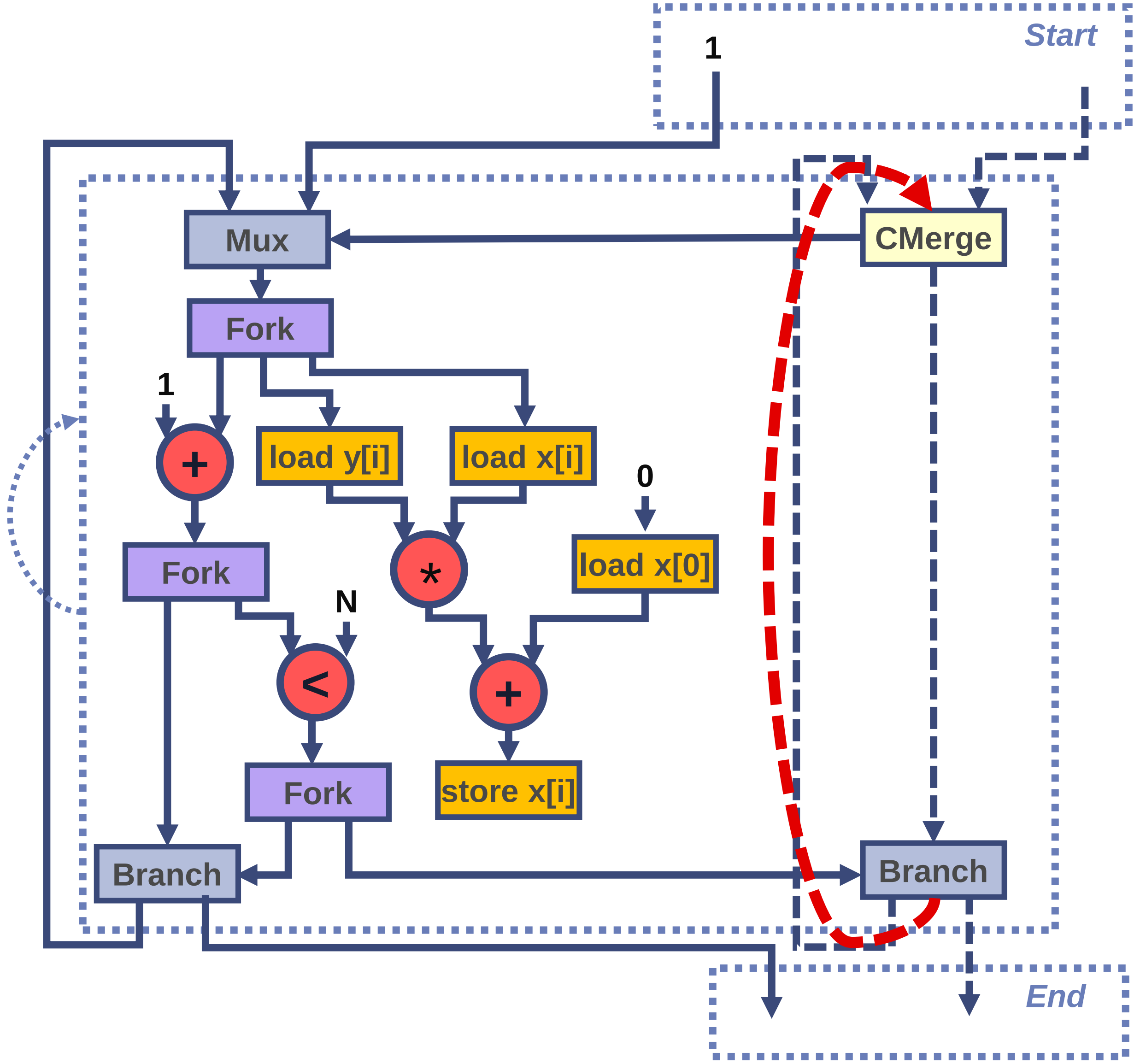
XILINX

Handshake Dialect Transform: Buffer Insertion (1)



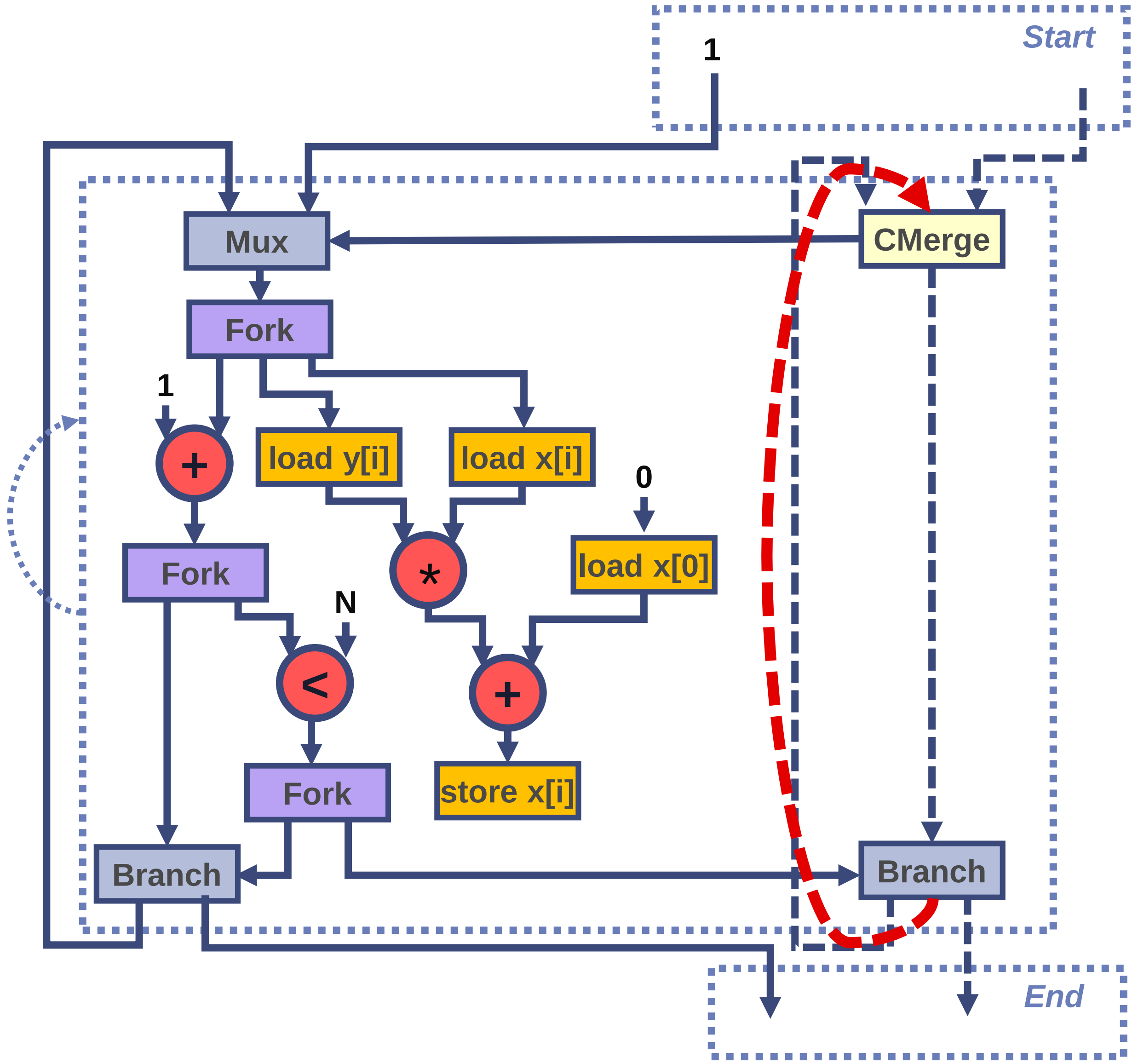
- Most elastic operations (except fork) in handshaked dataflow are implemented as combinational logic

Handshake Dialect Transform: Buffer Insertion (2)



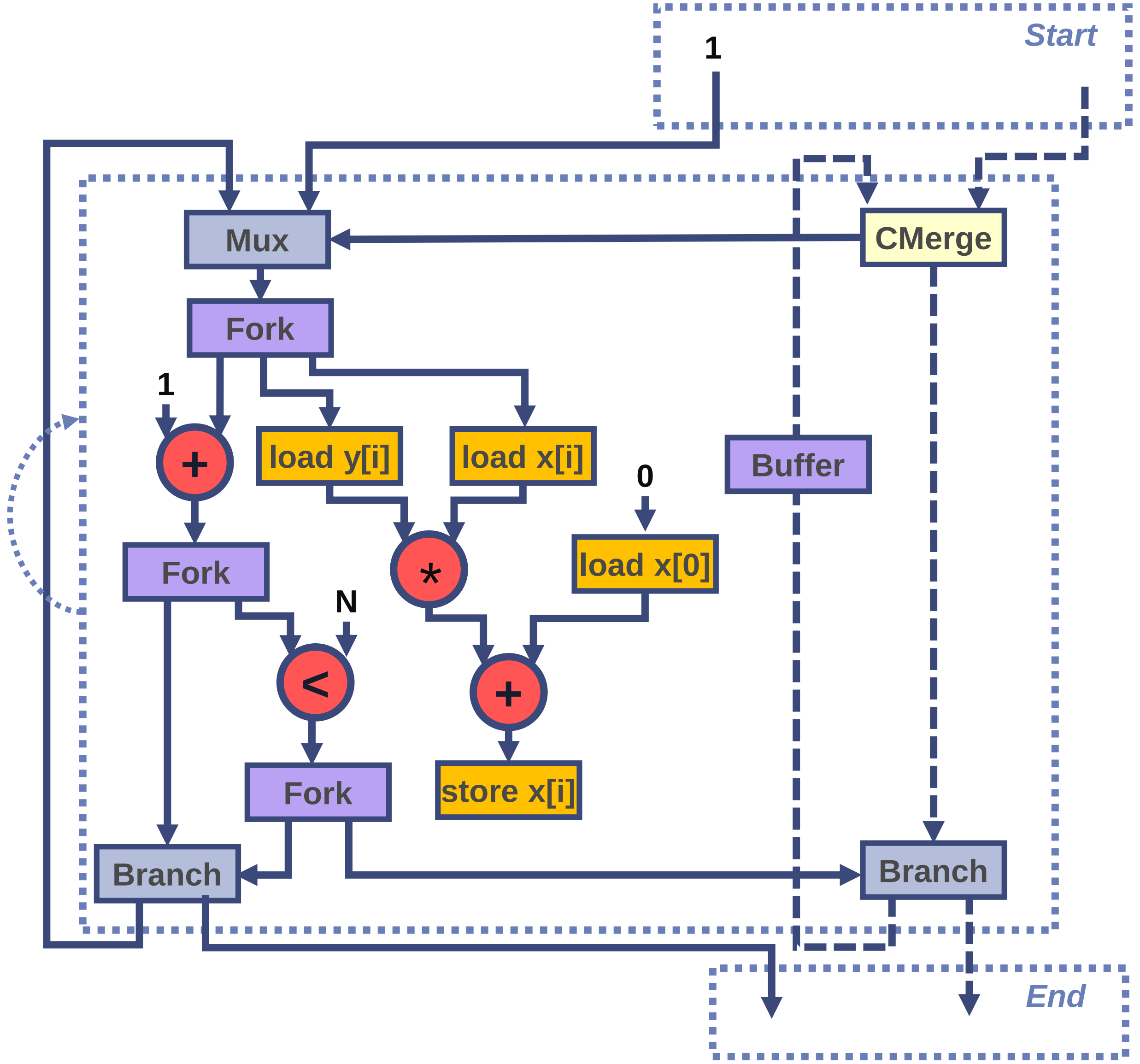
- Most elastic operations (except fork) in handshaked dataflow are implemented as combinational logic
- CMerge waits Branch to ready, while Branch waits CMerge to ready

Handshake Dialect Transform: Buffer Insertion (3)



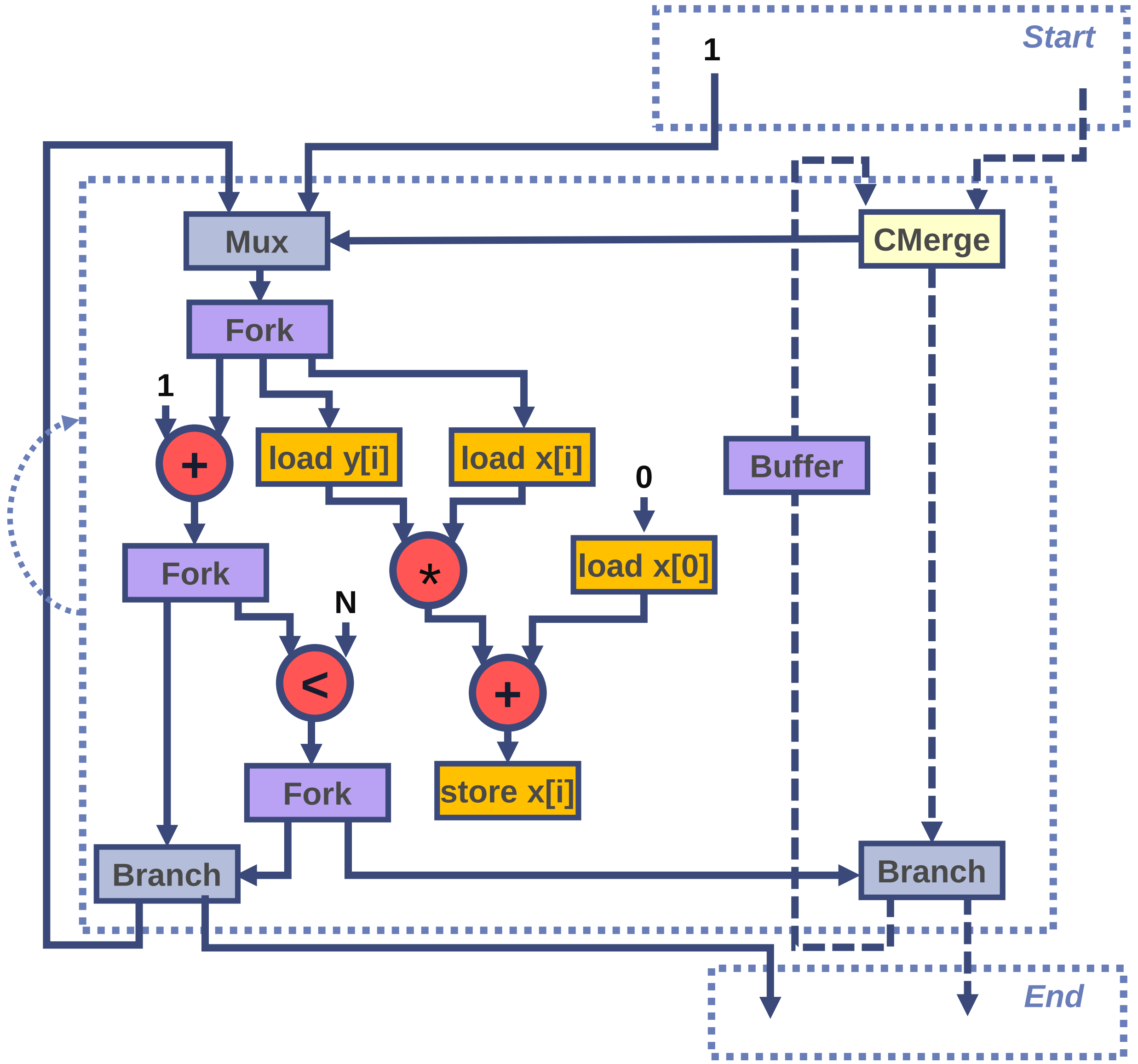
- Most elastic operations (except fork) in handshaked dataflow are implemented as combinational logic
- CMerge waits Branch to ready, while Branch waits CMerge to ready
- => Combinational graph cycle will cause dead lock, at least one sequential component should be inserted to break the cycle

Handshake Dialect Transform: Buffer Insertion (4)



- Most elastic operations (except fork) in handshaked dataflow are implemented as combinational logic
- CMerge waits Branch to ready, while Branch waits CMerge to ready
- => Combinational graph cycle will cause dead lock, at least one sequential component should be inserted to break the cycle
- => Insert a buffer to each detected combinational graph cycle

Handshake Dialect Transform: Buffer Insertion (5)



For correctness

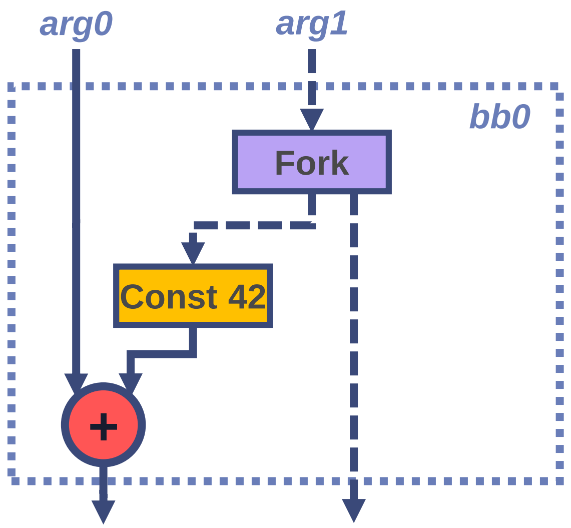
- Most elastic operations (except fork) in handshaked dataflow are implemented as combinational logic
- CMerge waits Branch to ready, while Branch waits CMerge to ready
- => Combinational graph cycle will cause dead lock, at least one sequential component should be inserted to break the cycle
- => Insert a buffer to each detected combinational graph cycle

For performance

- *Buffers can break critical path and eliminate back pressure*

Handshake to FIRRTL Conversion (1)

```
handshake.func @simple_addi(%arg0: index, %arg1: none) -> (index, none) {
  %0:2 = "handshake.fork"(%arg1) {control = true} : (none) -> (none, none)
  %1 = "handshake.constant"(%0#0) {value = 42 : index} : (none) -> index
  %2 = addi %arg0, %1 : index
  handshake.return %2, %0#1 : index, none
}
```



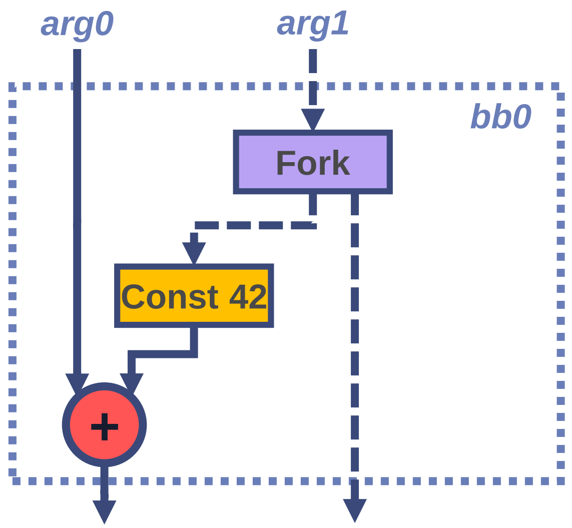
Type Conversion ⇩ **Lowering**

```
firrtl.module @simple_addi(
  %arg0: !firrtl.bundle<valid: uint<1>, ready: flip<uint<1>>, data: uint<64>>,       //arg0
  %arg1: !firrtl.bundle<valid: uint<1>, ready: flip<uint<1>>>,                       //arg1
  %arg2: !firrtl.bundle<valid: flip<uint<1>>, ready: uint<1>, data: flip<uint<64>>>, //result0
  %arg3: !firrtl.bundle<valid: flip<uint<1>>, ready: uint<1>>,                       //result1
  %clock: !firrtl.clock, %reset: !firrtl.uint<1>) {

}
```

XILINX.

Handshake to FIRRTL Conversion (1)

```
handshake.func @simple_addi(%arg0: index, %arg1: none) -> (index, none) {
  %0:2 = "handshake.fork"(%arg1) {control = true} : (none) -> (none, none)
  %1 = "handshake.constant"(%0#0) {value = 42 : index} : (none) -> index
  %2 = addi %arg0, %1 : index
  handshake.return %2, %0#1 : index, none
}
```

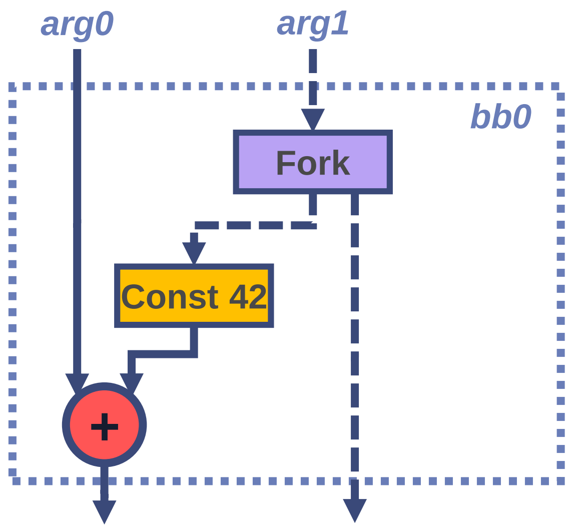


Type Conversion

Lowering

```
firrtl.module @simple_addi(... //ports) {

}
```

Handshake to FIRRTL Conversion (2)



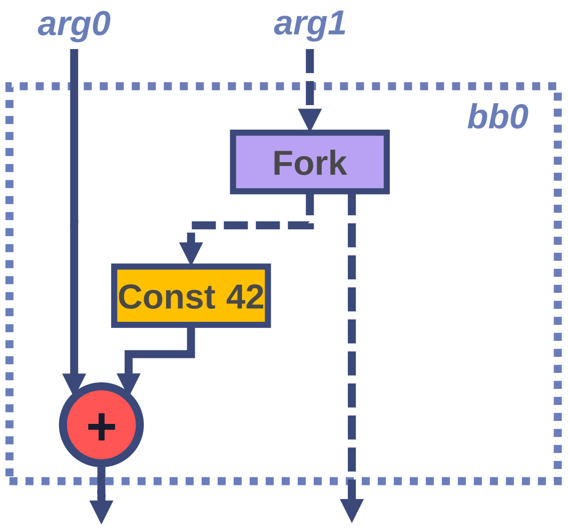
```
handshake.func @simple_addi(%arg0: index, %arg1: none) -> (index, none) {
  %0:2 = "handshake.fork"(%arg1) {control = true} : (none) -> (none, none)
  %1 = "handshake.constant"(%0#0) {value = 42 : index} : (none) -> index
  %2 = addi %arg0, %1 : index
  handshake.return %2, %0#1 : index, none
}
```

Build Submodule

⇩ **Lowering**

```
firrtl.module @handshake.fork_1ins_2outs_ctrl(... //ports) {... //logic implementation}
firrtl.module @handshake.constant_42_1ins_1outs(...) {...}
firrtl.module @std.addi_2ins_1outs(...) {...}

firrtl.module @simple_addi(... //ports) {

}
```

Handshake to FIRRTL Conversion (3)

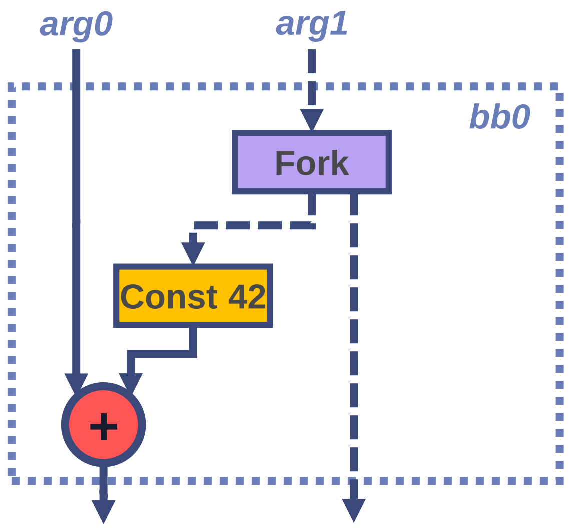


```
handshake.func @simple_addi(%arg0: index, %arg1: none) -> (index, none) {
  %0:2 = "handshake.fork"(%arg1) {control = true} : (none) -> (none, none)
  %1 = "handshake.constant"(%0#0) {value = 42 : index} : (none) -> index
  %2 = addi %arg0, %1 : index
  handshake.return %2, %0#1 : index, none
}
```

Submodule Instantiation ⇩ **Lowering**

```
firrtl.module @handshake.fork_1ins_2outs_ctrl(... //ports) {... //logic implementation}
firrtl.module @handshake.constant_42_1ins_1outs(...) {...}
firrtl.module @std.addi_2ins_1outs(...) {...}

firrtl.module @simple_addi(... //ports) {
  %0 = firrtl.instance @handshake.fork_1ins_2outs_ctrl {name = ""} : !firrtl.bundle<... //ports>
  %4 = firrtl.instance @handshake.constant_1ins_1outs {name = ""} : !firrtl.bundle<...>
  %7 = firrtl.instance @std.addi_2ins_1outs {name = ""} : !firrtl.bundle<...>
}
```

Handshake to FIRRTL Conversion (4)

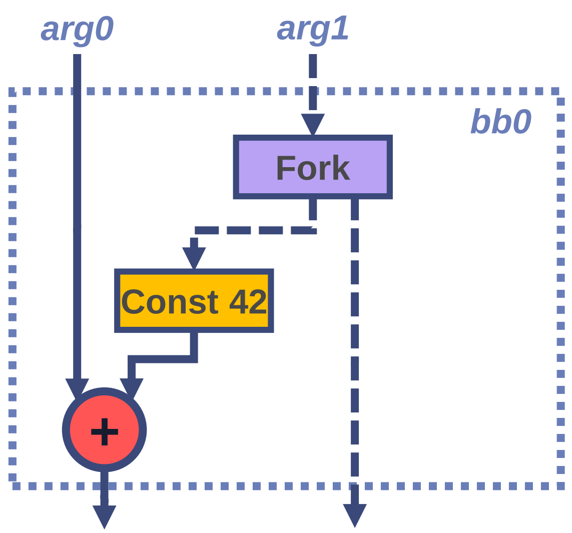


```
handshake.func @simple_addi(%arg0: index, %arg1: none) -> (index, none) {
  %0:2 = "handshake.fork"(%arg1) {control = true} : (none) -> (none, none)
  %1 = "handshake.constant"(%0#0) {value = 42 : index} : (none) -> index
  %2 = addi %arg0, %1 : index
  handshake.return %2, %0#1 : index, none
}
```

Reconnection ⇩ **Lowering**

```
firrtl.module @handshake.fork_1ins_2outs_ctrl(... //ports) {... //logic implementation}
firrtl.module @handshake.constant_42_1ins_1outs(...) {...}
firrtl.module @std.addi_2ins_1outs(...) {...}

firrtl.module @simple_addi(... //ports) {
  ... //firrtl.subfield and firrtl.connect
  %0 = firrtl.instance @handshake.fork_1ins_2outs_ctrl {name = ""} : !firrtl.bundle<... //ports>
  ...
  %4 = firrtl.instance @handshake.constant_1ins_1outs {name = ""} : !firrtl.bundle<...>
  ...
  %7 = firrtl.instance @std.addi_2ins_1outs {name = ""} : !firrtl.bundle<...>
  ...
}
```

Handshake to FIRRTL Conversion (4)



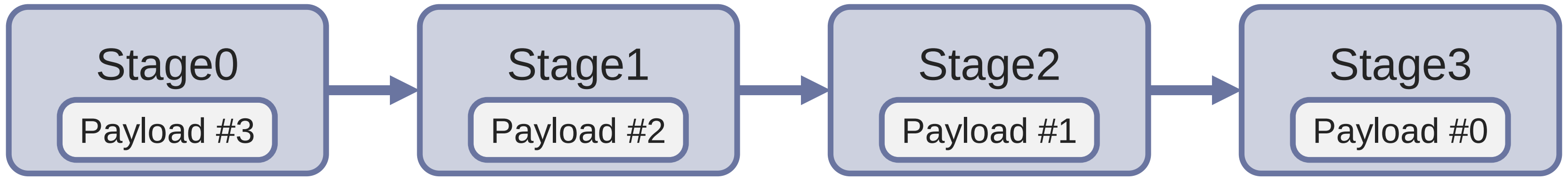
```
handshake.func @simple_addi(%arg0: index, %arg1: none) -> (index, none) {
  %0:2 = "handshake.fork"(%arg1) {control = true} : (none) -> (none, none)
  %1 = "handshake.constant"(%0#0) {value = 42 : index} : (none) -> index
  %2 = addi %arg0, %1 : index
  handshake.return %2, %0#1 : index, none
}
```

Reconnection ⇩ **Lowering**

```
firrtl.module @handshake.fork_1ins_2outs_ctrl(... //ports) {... //logic implementation}
firrtl.module @handshake.constant_42_1ins_1outs(...) {...}
firrtl.module @std.addi_2ins_1outs(...) {...}

firrtl.module @simple_addi(... //ports) {
  ... //firrtl.subfield and firrtl.connect
  %0 = firrtl.instance @handshake.fork_1ins_2outs_ctrl {name = ""} : !firrtl.bundle<... //ports>
  ...
  %4 = firrtl.instance @handshake.constant_1ins_1outs {name = ""} : !firrtl.bundle<...>
  ...
  %7 = firrtl.instance @std.addi_2ins_1outs {name = ""} : !firrtl.bundle<...>
  ...
  firrtl.connect %arg2, %10 : !firrtl.bundle<...>, !firrtl.bundle<...>
  firrtl.connect %arg3, %3 : !firrtl.bundle<...>, !firrtl.bundle<...>
}
```

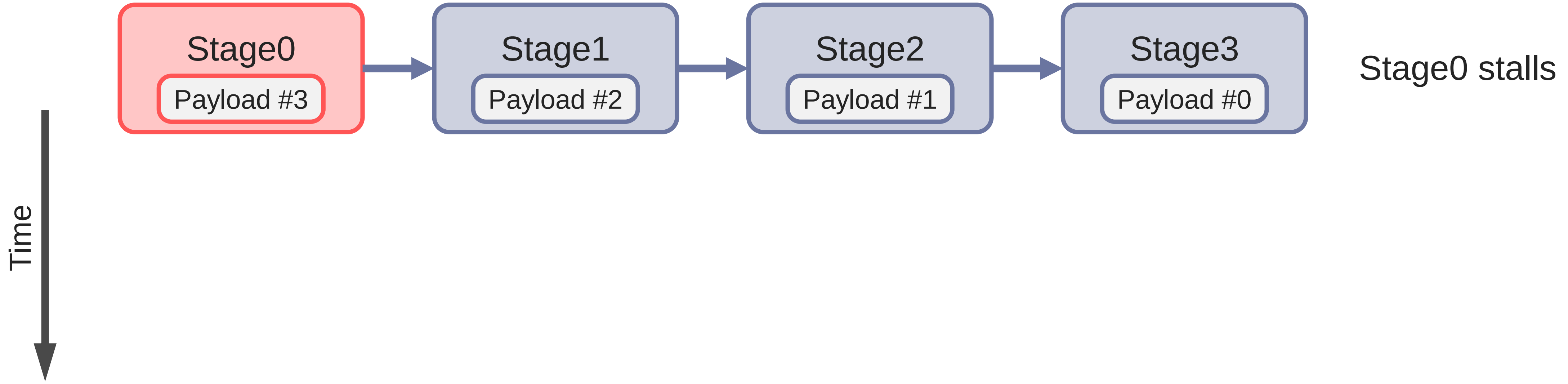
Handshake to FIRRTL Conversion (5)

- Pipeline operations are implemented as flushable pipeline



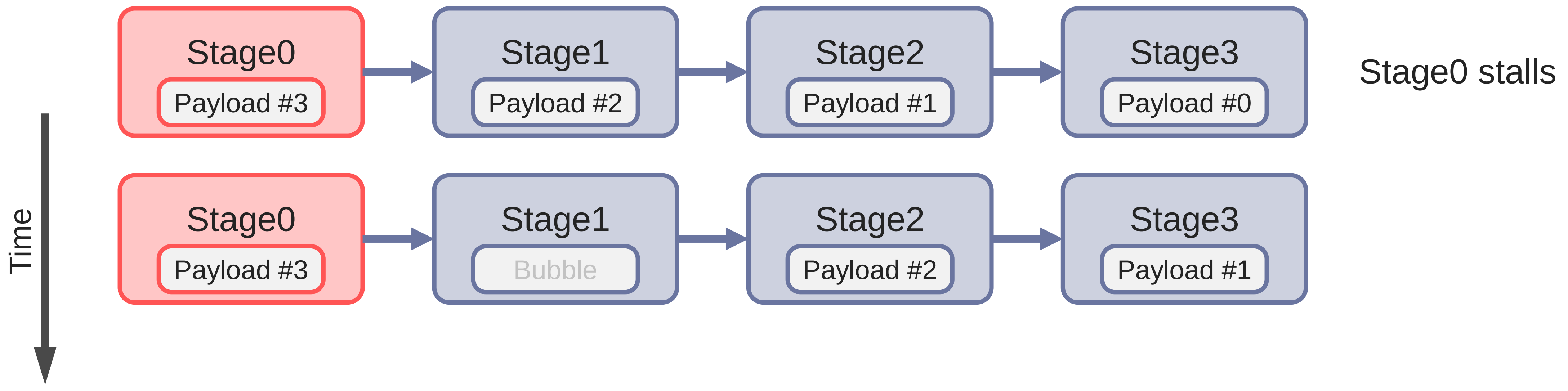
Handshake to FIRRTL Conversion (5)

- Pipeline operations are implemented as flushable pipeline



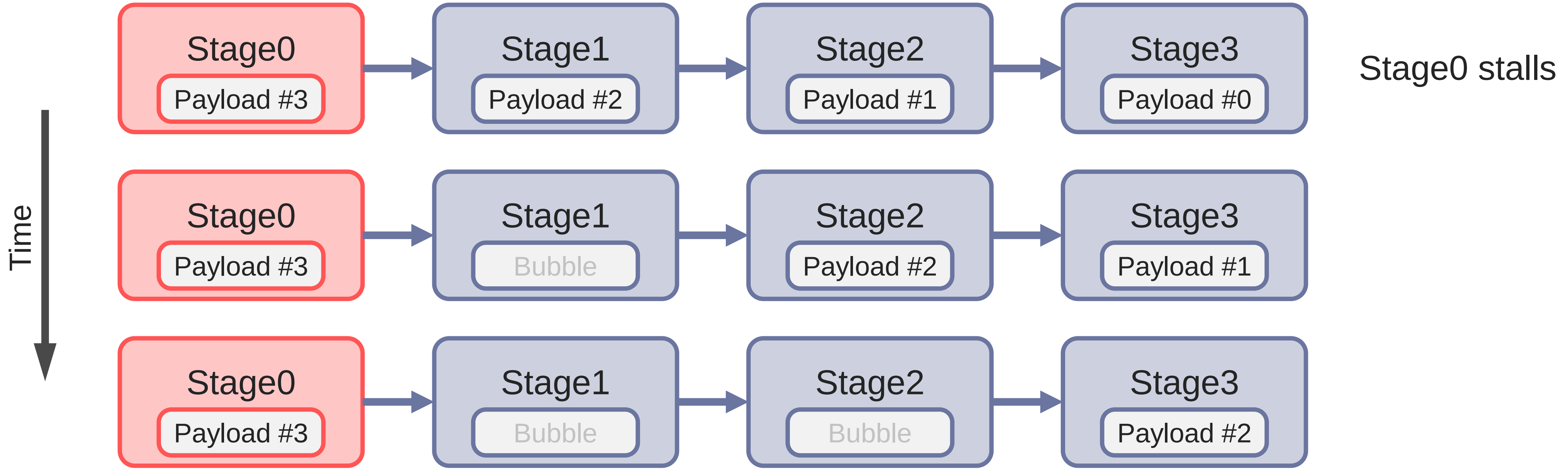
Handshake to FIRRTL Conversion (5)

- Pipeline operations are implemented as flushable pipeline



Handshake to FIRRTL Conversion (5)

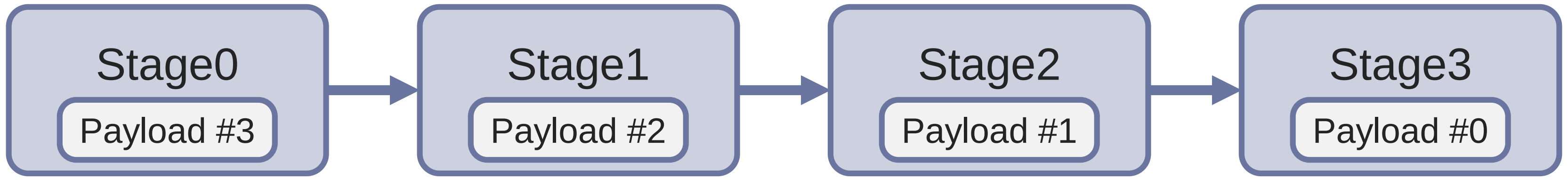
- Pipeline operations are implemented as flushable pipeline



- Valid signals are registered in each pipeline stage from input to output

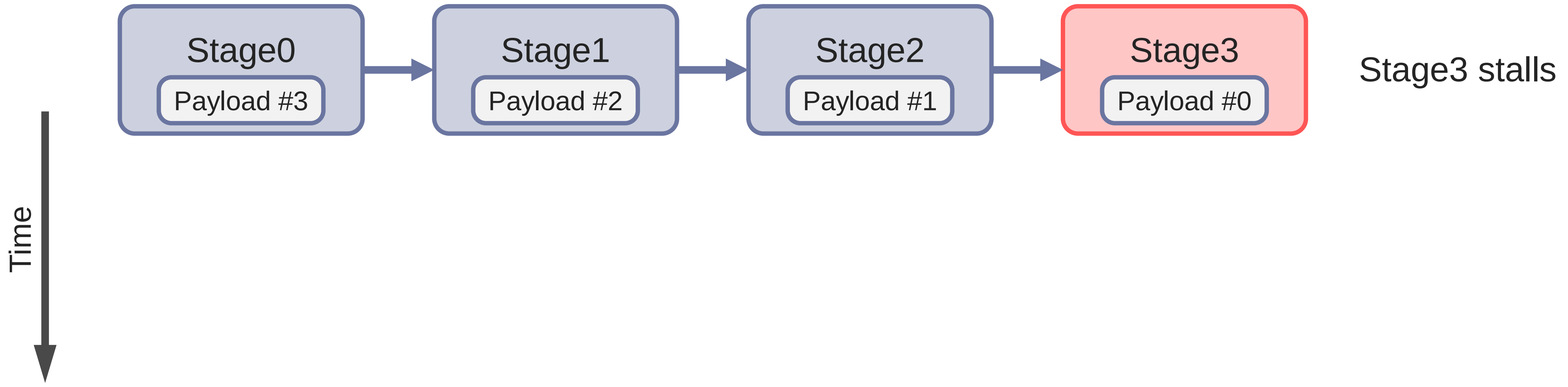
Handshake to FIRRTL Conversion (6)

- Pipeline operations are implemented as flushable pipeline



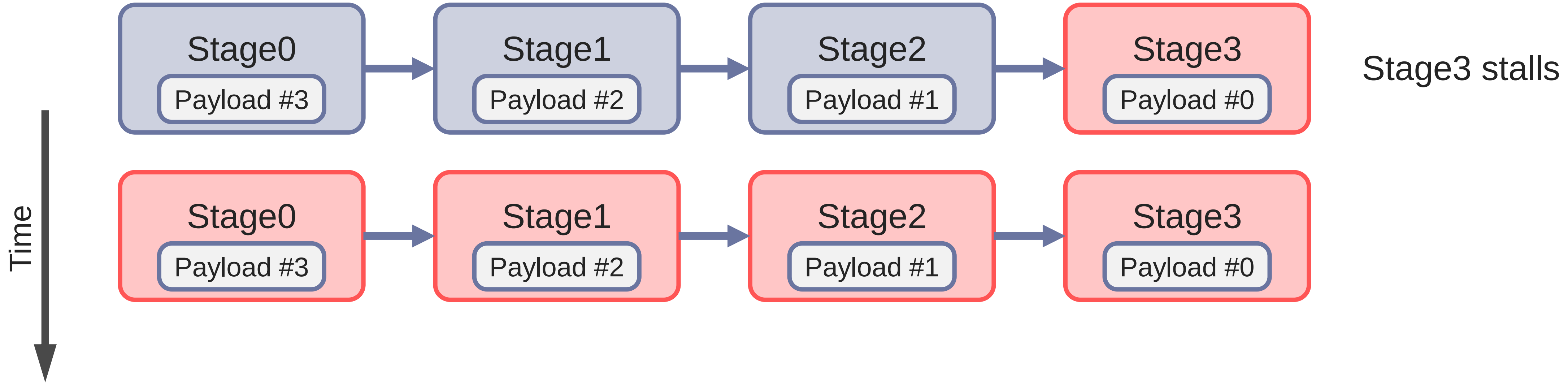
Handshake to FIRRTL Conversion (6)

- Pipeline operations are implemented as flushable pipeline



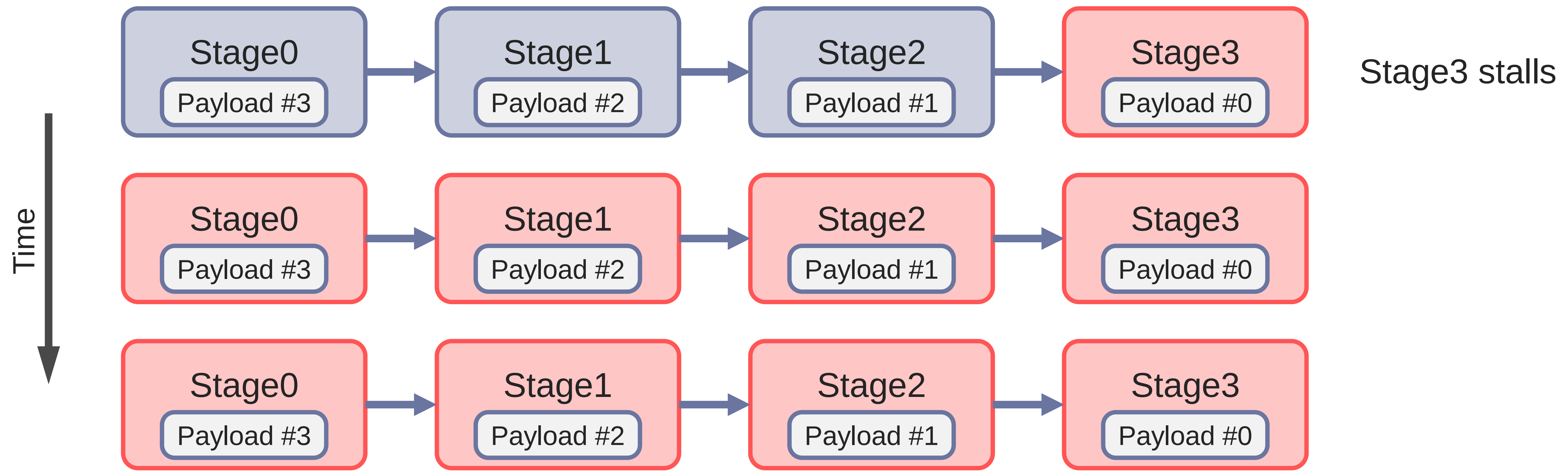
Handshake to FIRRTL Conversion (6)

- Pipeline operations are implemented as flushable pipeline



Handshake to FIRRTL Conversion (6)

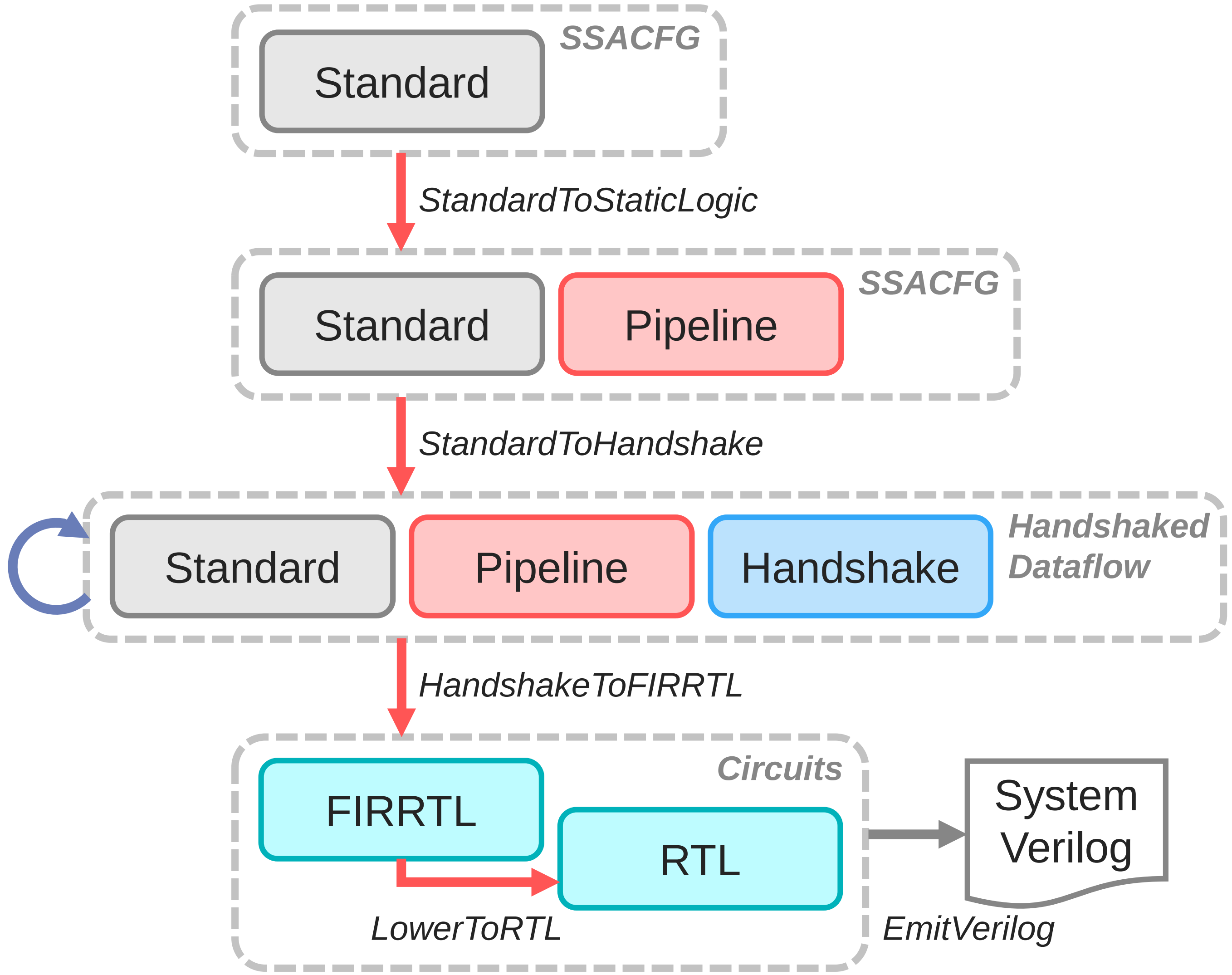
- Pipeline operations are implemented as flushable pipeline



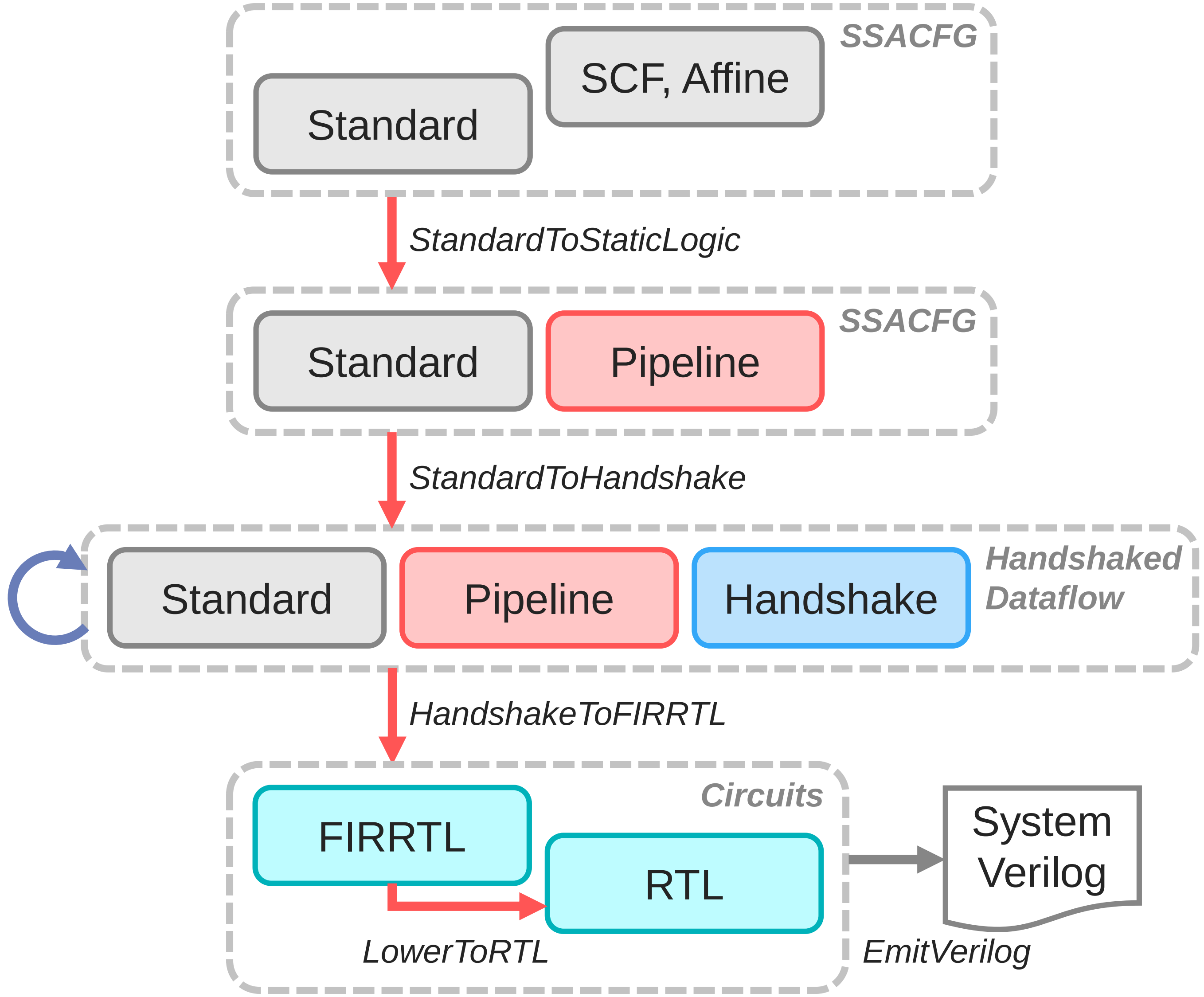
- Ready signals are combinationally propagated from output to input

Next Steps!

On top of pipeline operation, what's next? (1)

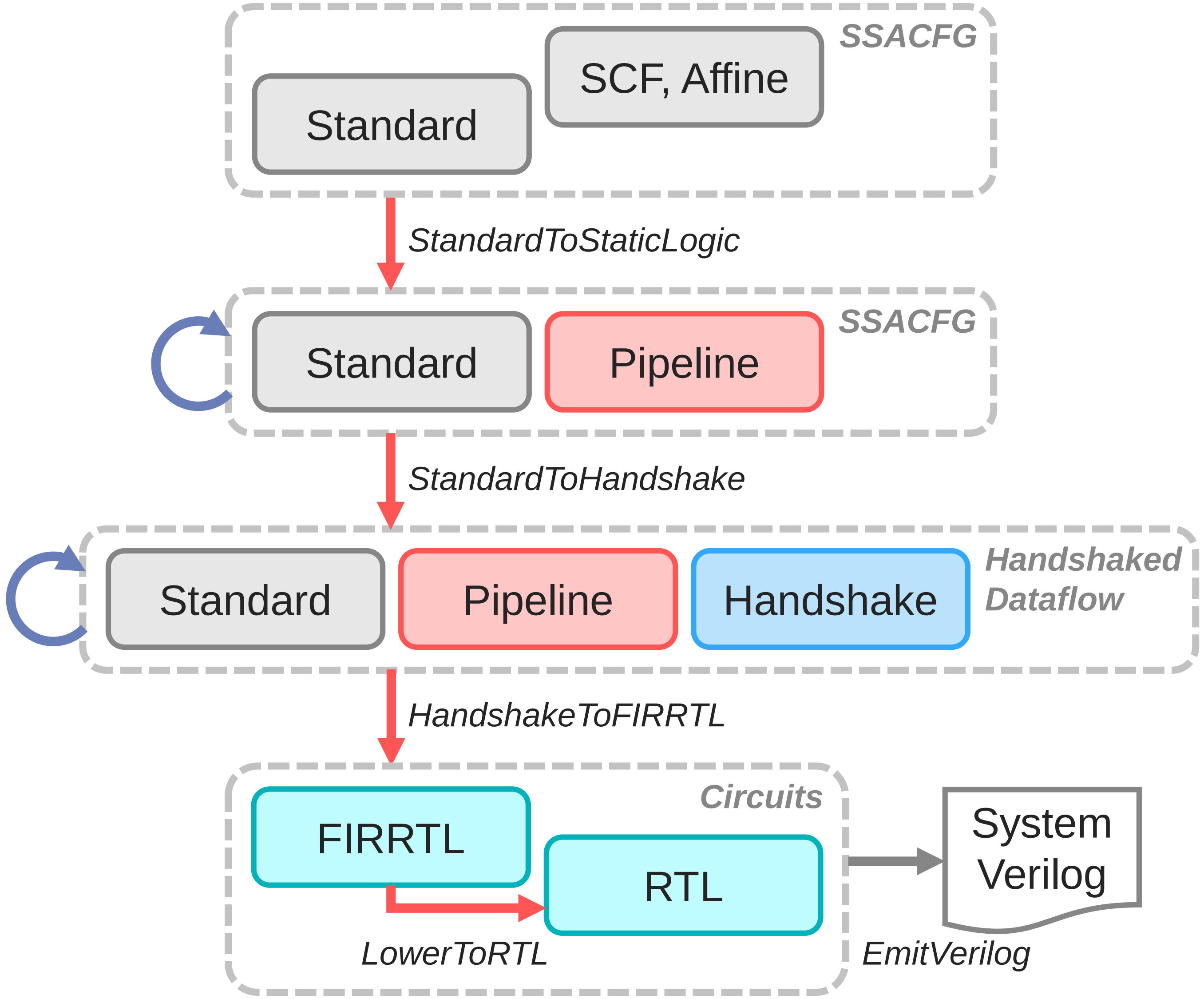


On top of pipeline operation, what's next? (2)



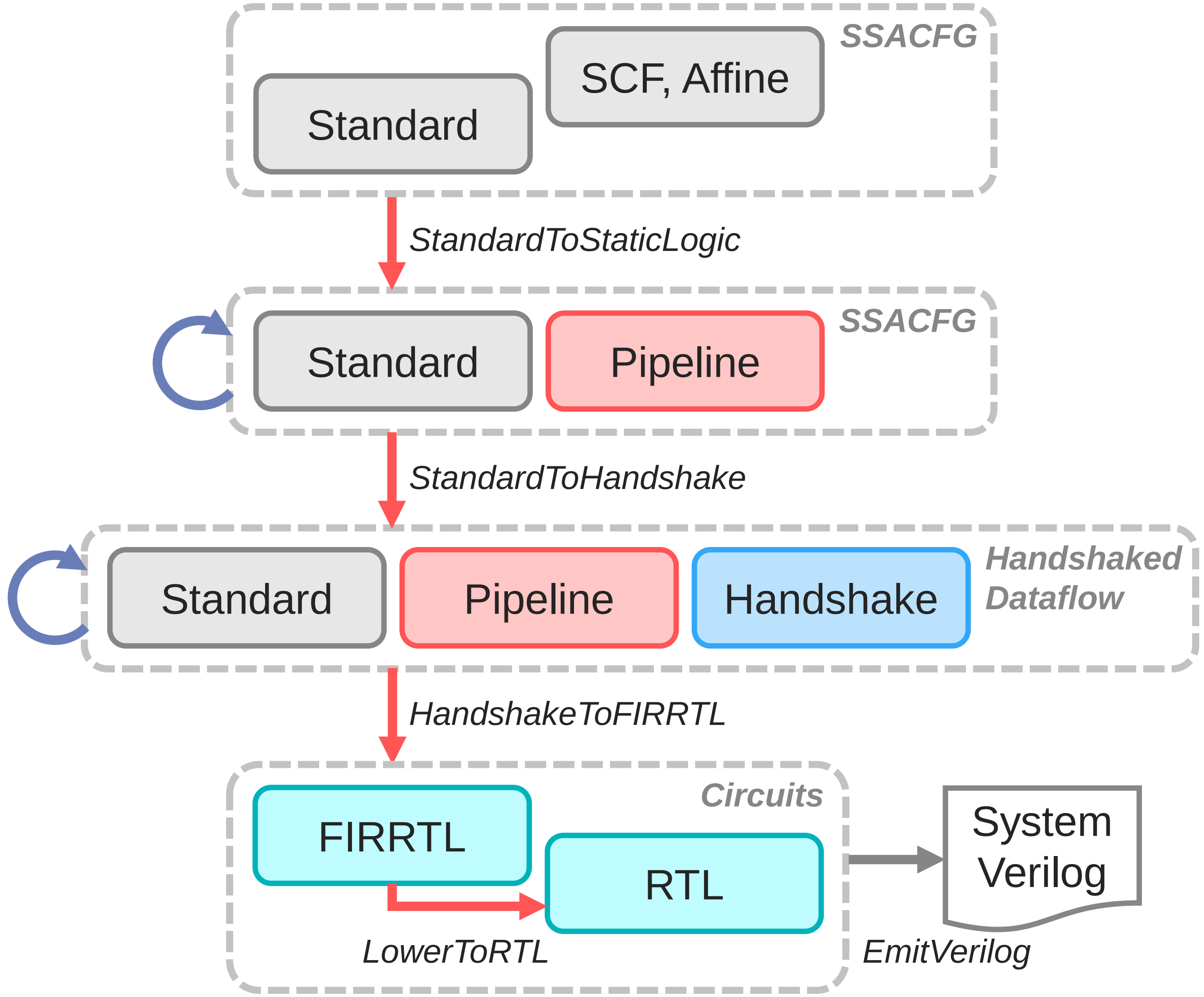
- Standard to StaticLogic Conversion
  - Directly lower from SCF/Affine operations
  - How to distinguish "pipelinable" processes

On top of pipeline operation, what's next? (3)



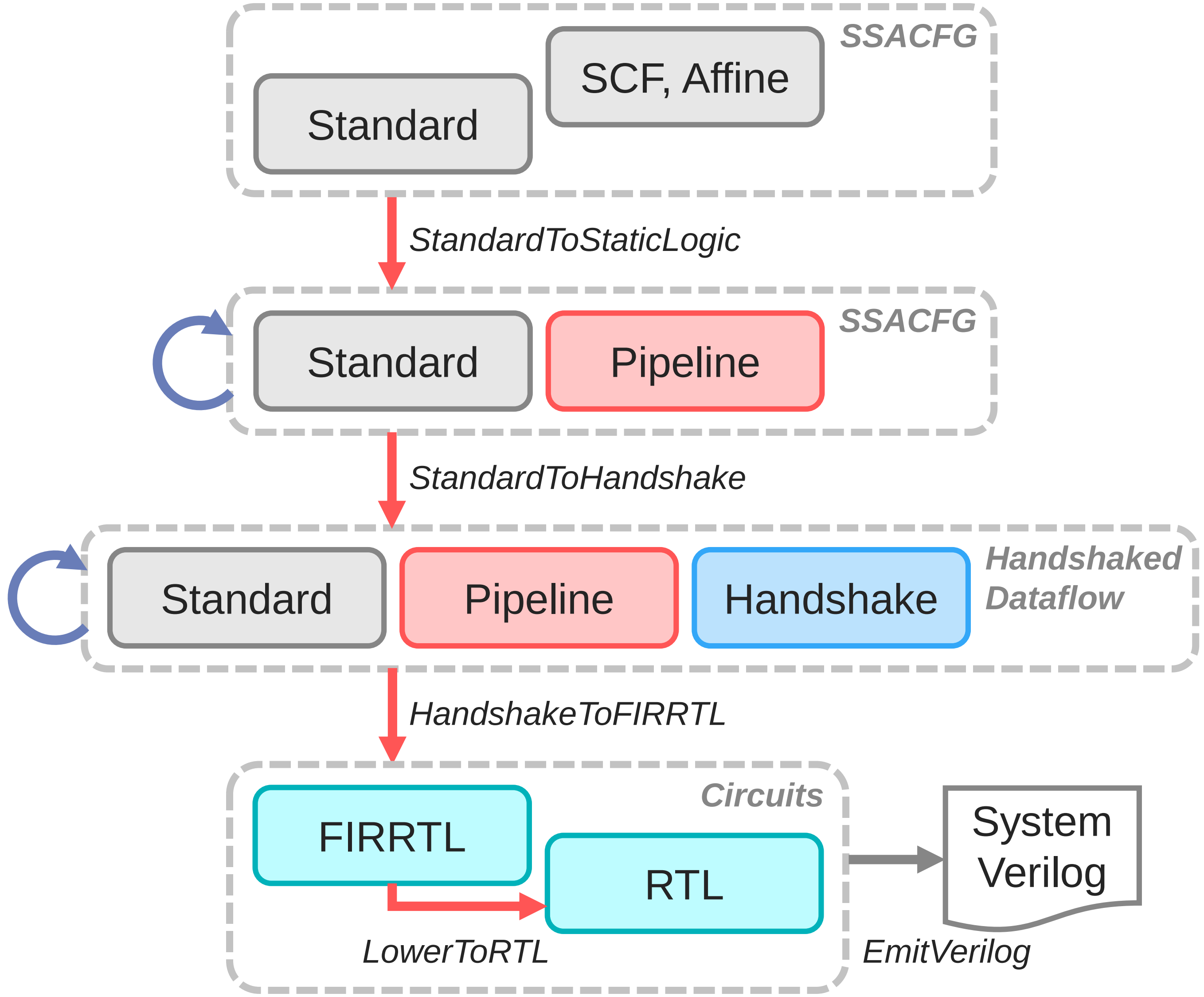
- Standard to StaticLogic Conversion
  - Directly lower from SCF/Affine operations
  - How to distinguish "pipelinable" processes
- Pipeline Dialect Transform
  - Pipeline mapping (e.g. stage splitting)

On top of pipeline operation, what’s next? (4)



- Standard to StaticLogic Conversion
  - Directly lower from SCF/Affine operations
  - How to distinguish “pipelinable” processes
- Pipeline Dialect Transform
  - Pipeline mapping (e.g. stage splitting)
- Handshake Dialect Transform
  - Pipeline-aware buffer insertion

On top of pipeline operation, what's next? (5)



- Standard to StaticLogic Conversion
  - Directly lower from SCF/Affine operations
  - How to distinguish "pipelinable" processes
- Pipeline Dialect Transform
  - Pipeline mapping (e.g. stage splitting)
- Handshake Dialect Transform
  - Pipeline-aware buffer insertion
- Handshake to FIRRTL Conversion
  - Handshake => FIRRTL =🤔=>Verilog
  - Feedback path handling in pipeline
  - Late input and early output in pipeline
  - RTL-level simulation (co-simulation)
  - Incorporate with external FIRRTL or Verilog modules, or IPs

XILINX

XILINX®

Thank You