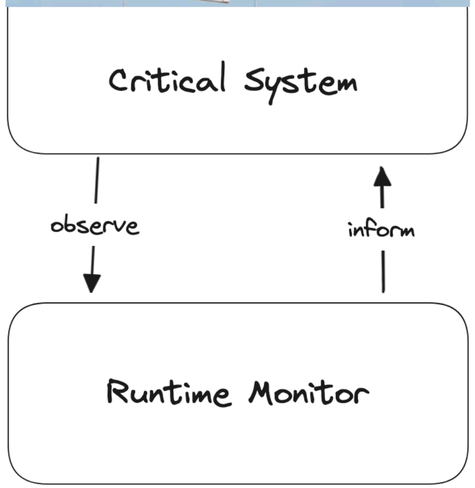
Functional HDLs Meet Stream-based Monitoring

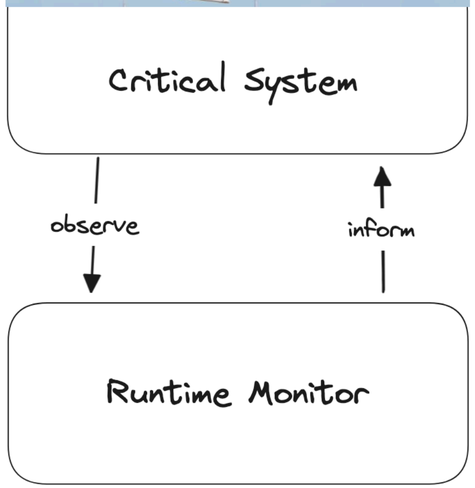
Hardware Monitors from RTLola Using Clash

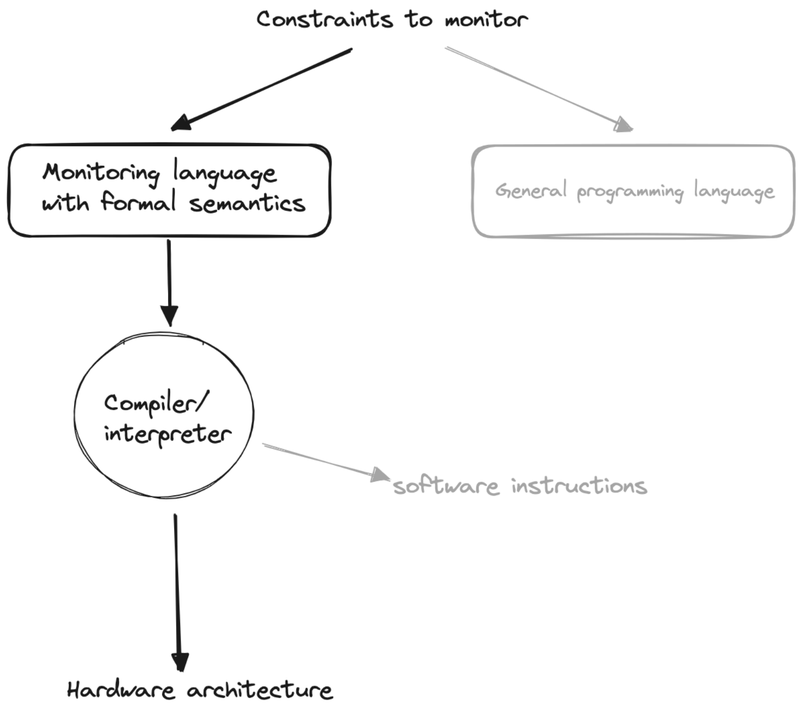
Bipin Oli, 04.09.2025

Hardware-based runtime monitoring



Hardware-based runtime monitoring





Existing hardware compiler for RTLola

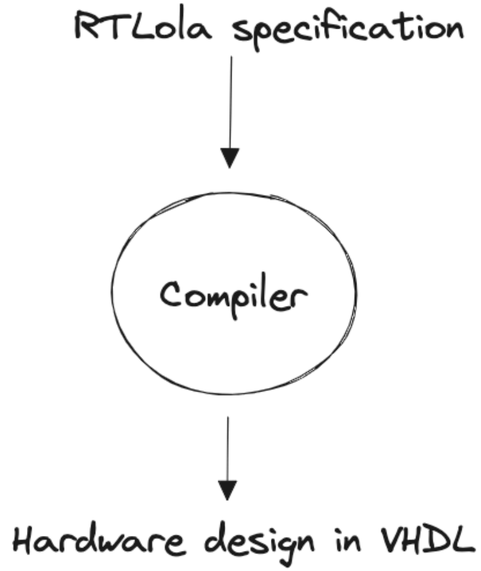
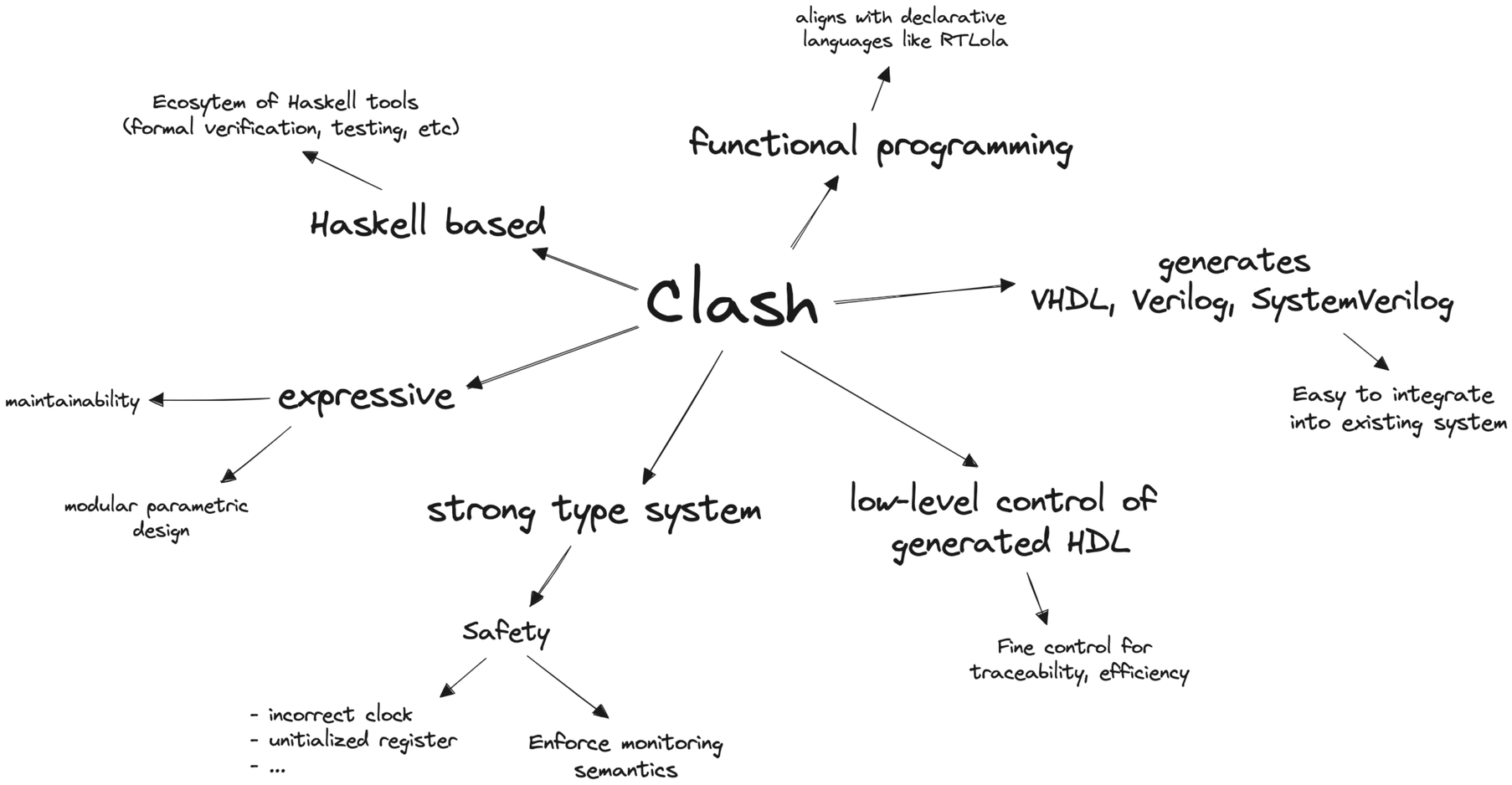
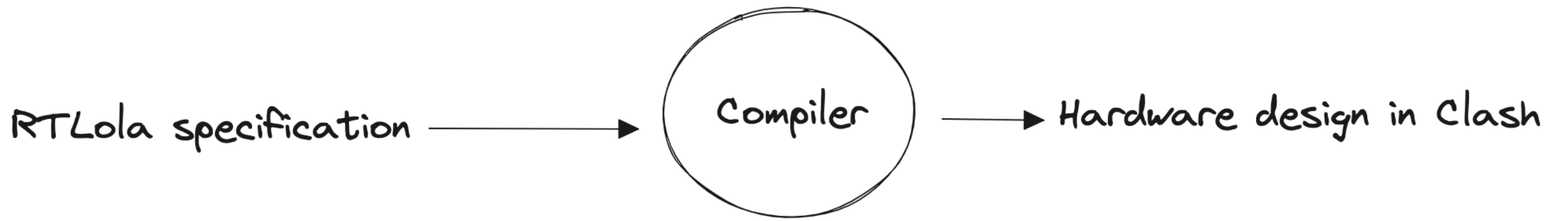


Fig: Existing compiler for RTLola

Functional HDL - Clash



Objective of the thesis



RTLola: input stream

```
input x: Int

output a:Int64 @x  := x.offset(by: -1).defaults(to: 0)
output b:Int64 @1Hz := x.hold(or: 0) + a.aggregate(over: 0.5s, using: sum)
```

event-based

periodic

RTLola: output stream

event-based

periodic

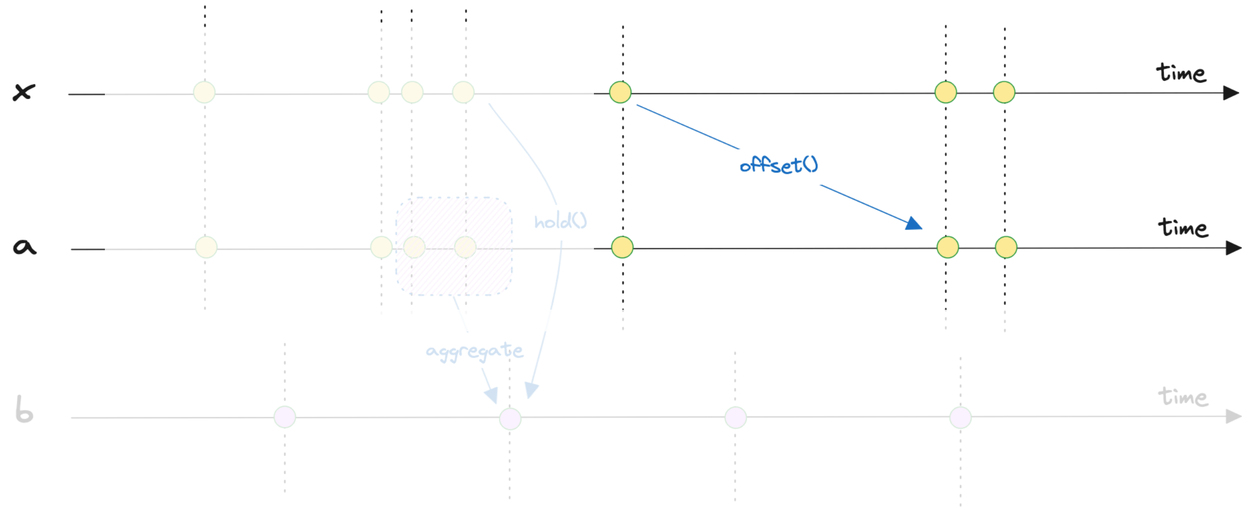
```
input x: Int

output a:Int64 @x  := x.offset(by: -1).defaults(to: 0)
output b:Int64  @1Hz := x.hold(or: 0) + a.aggregate(over: 0.5s, using: sum)
```

RTLola: accessing past value

event-based

periodic

```
input x: Int

output a:Int64 @x  := x.offset(by: -1).defaults(to: 0)
output b:Int64  @1Hz := x.hold(or: 0) + a.aggregate(over: 0.5s, using: sum)
```

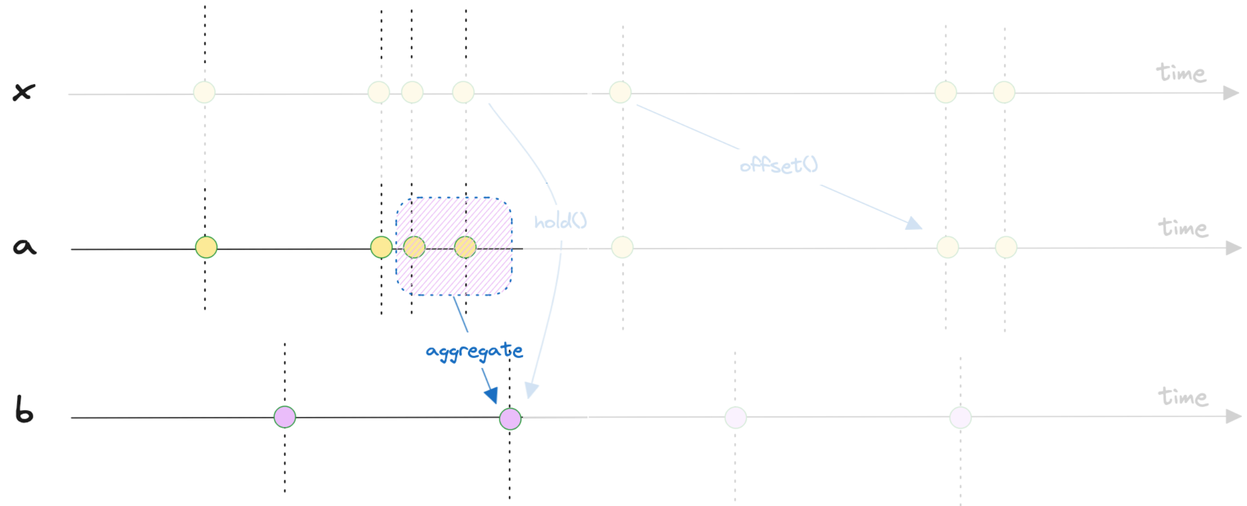


RTLola: aggregating over window of event-based values

event-based

periodic

```
input x: Int

output a:Int64 @x  := x.offset(by: -1).defaults(to: 0)
output b:Int64  @1Hz := x.hold(or: 0) + a.aggregate(over: 0.5s, using: sum)
```

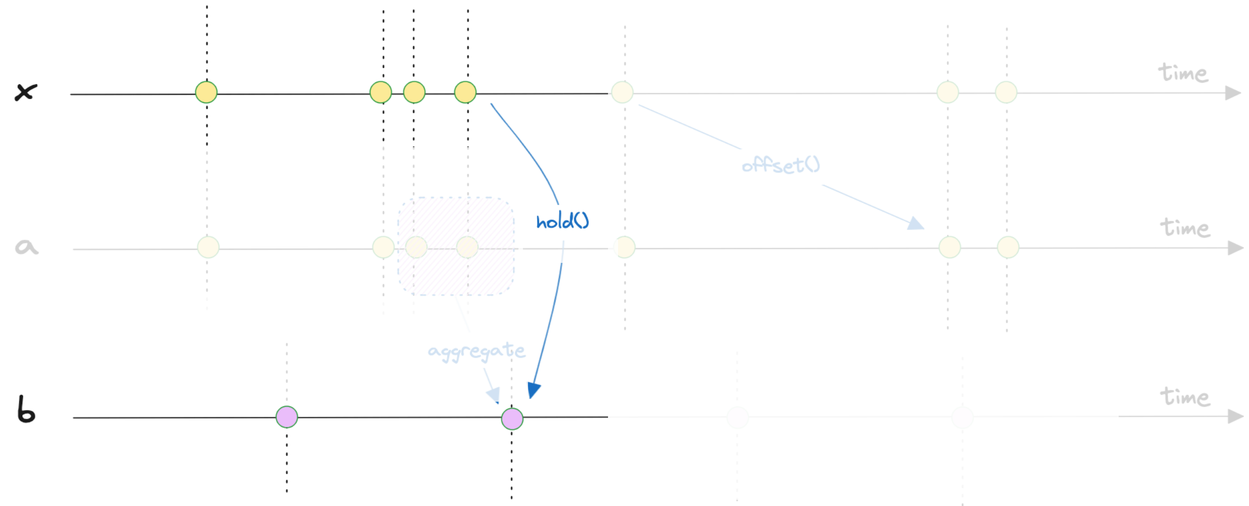


RTLola: accessing latest value

```
input x: Int

output a:Int64 @x  := x.offset(by: -1).defaults(to: 0)
output b:Int64  @1Hz := x.hold(or: 0) + a.aggregate(over: 0.5s, using: sum)
```

event-based

periodic



RTLola - existing hardware architecture

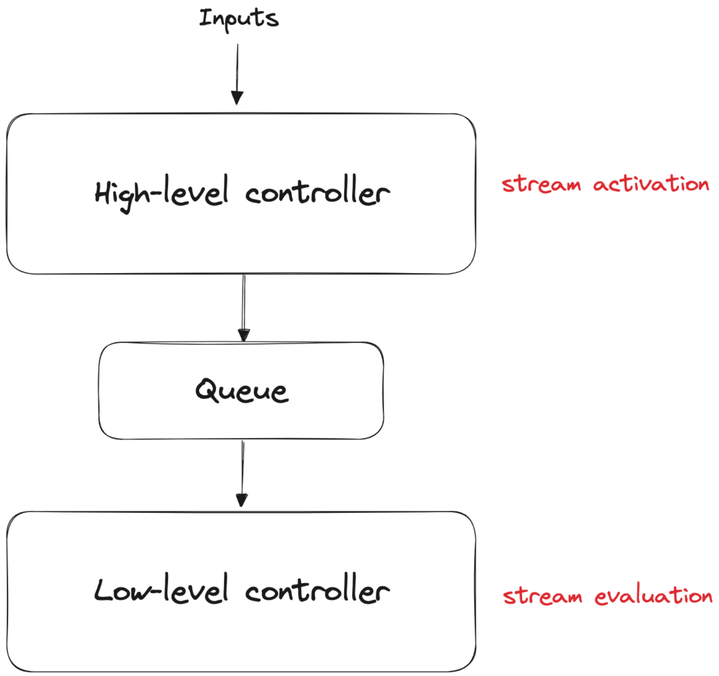


Fig: Hardware architecture (Baumeister et. al.)

Stream activation: High-level controller (HLC)

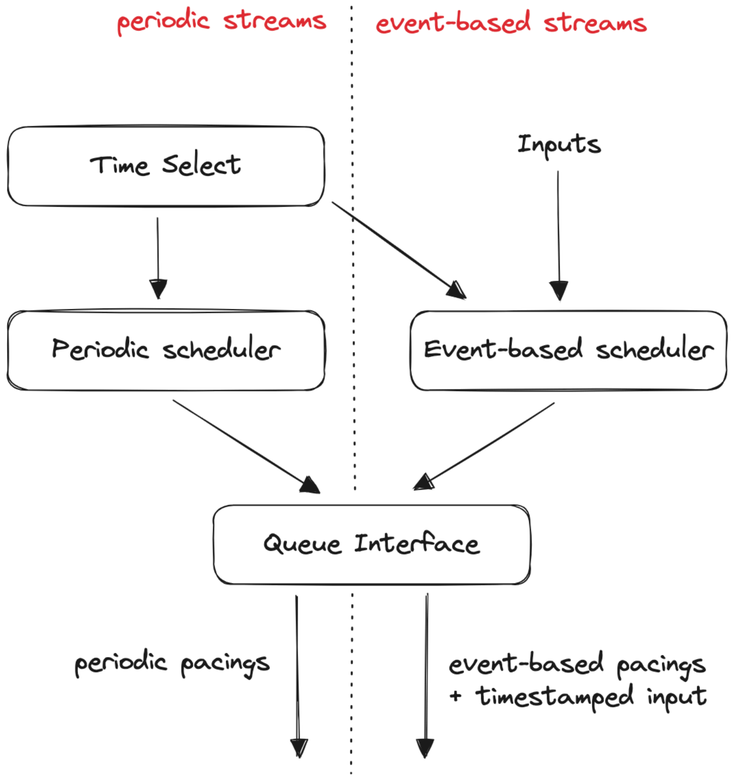
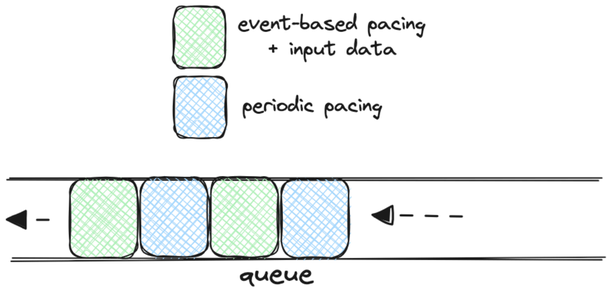


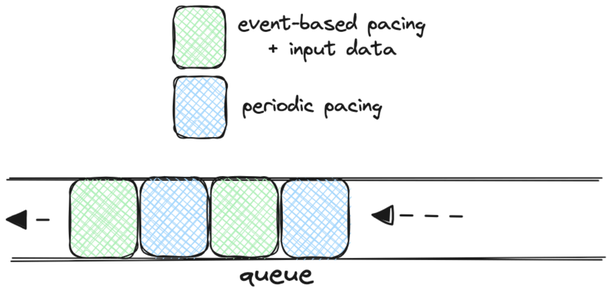
Fig: Structure of HLC

Stream evaluation: Low-level controller (LLC)

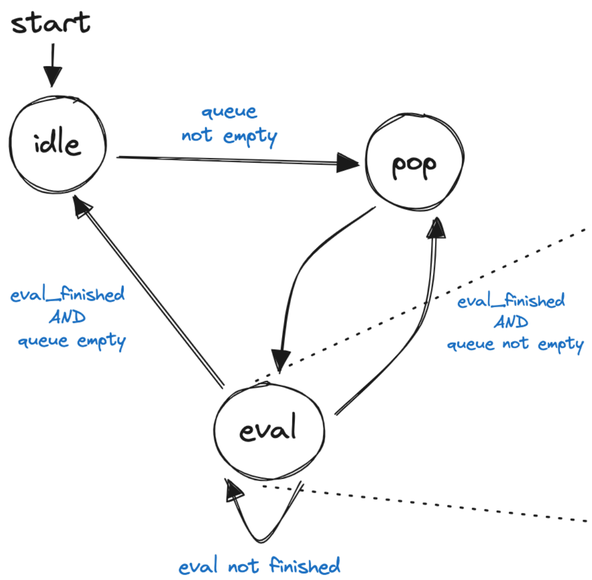



```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64 @1Hz := b.aggregate(over: 1s, using: sum)
```

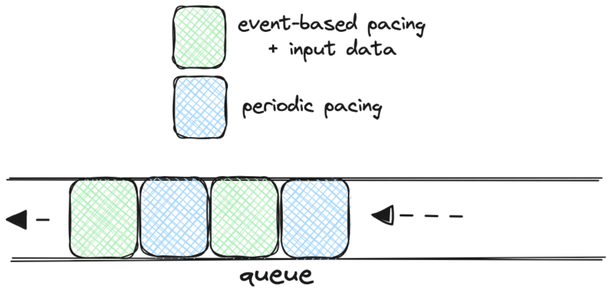
Stream evaluation: Low-level controller (LLC)



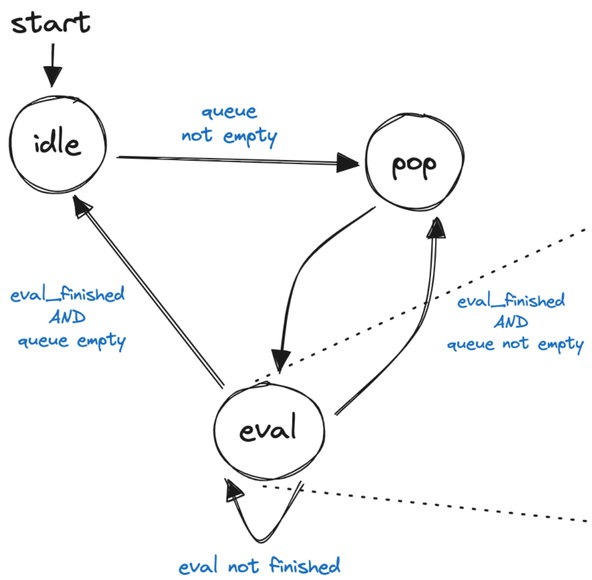

```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64 @1Hz := b.aggregate(over: 1s, using: sum)
```

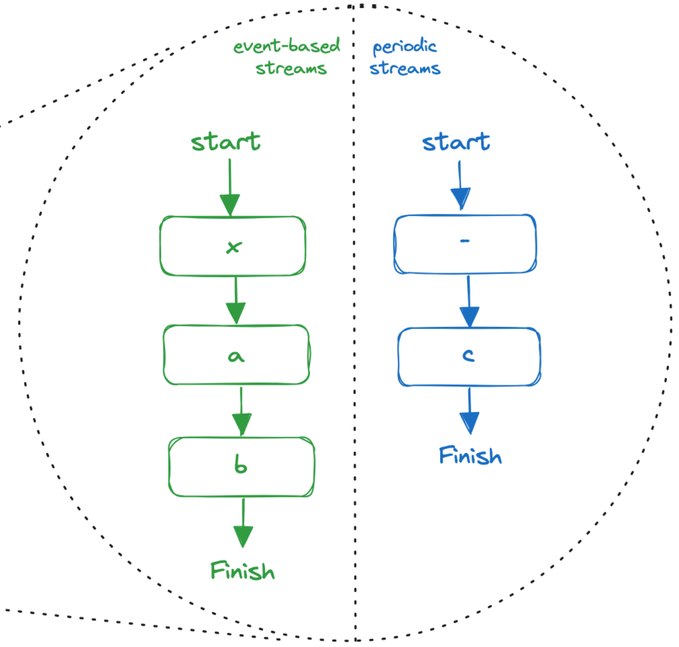


Stream evaluation: Low-level controller (LLC)



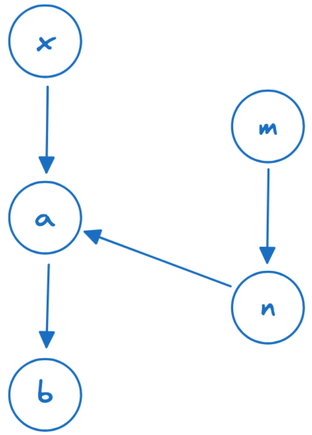

```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64  @1Hz := b.aggregate(over: 1s, using: sum)
```





Implementation: change in architecture - eval order

```
input x : Int

output a:Int64 @x  := x + n.hold(or: 1)
output b:Int64 @x  := a + 1

output m:Int64  @1Hz := 1
output n:Int64  @1Hz := m + 1
```



Existing architecture

Event based:
- x
- a
- b

Periodic:
- m
- n

alternate evaluation

Our architecture

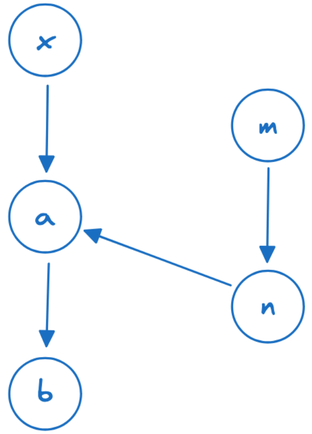
Eval order:
- x
- a, m
- b, n

a before n

single evaluation oder

Implementation: change in architecture - eval order

```
input x : Int

output a:Int64 @x  := x + n.hold(or: 1)
output b:Int64 @x  := a + 1

output m:Int64  @1Hz := 1
output n:Int64  @1Hz := m + 1
```



Existing architecture

Event based:
- x
- a
- b

Periodic:
- m
- n

alternate evaluation

Our architecture

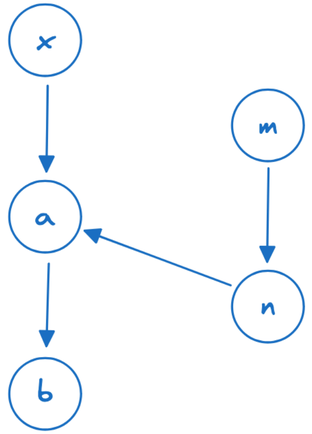
Eval order:
- x
- a, m
- b, n

a before n

single evaluation oder

Implementation: change in architecture - eval order

```
input x : Int

output a:Int64 @x  := x + n.hold(or: 1)
output b:Int64 @x  := a + 1

output m:Int64  @1Hz := 1
output n:Int64  @1Hz := m + 1
```



Existing architecture

Event based:
- x
- a
- b

Periodic:
- m
- n

alternate evaluation

Our architecture

Eval order:
- x
- a, m
- b, n

a before n

single evaluation oder

Implementation: change in architecture - time to slide

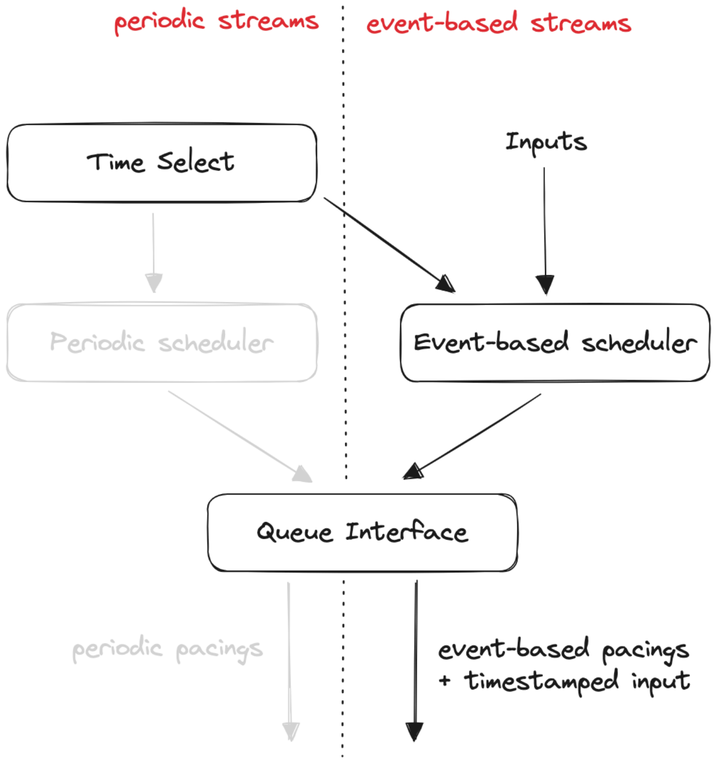


Fig: HLC in existing architecture

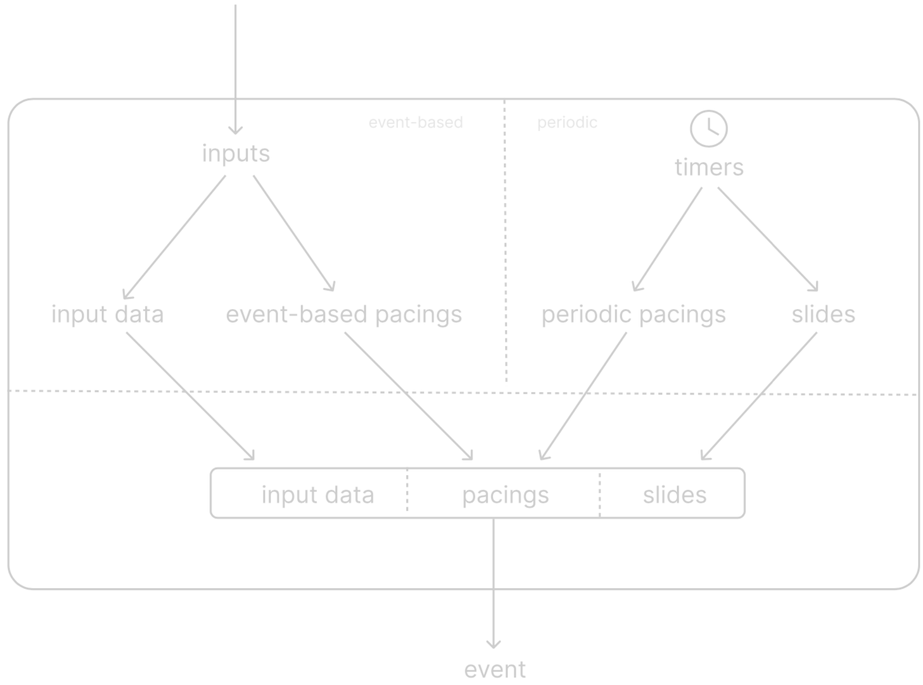


Fig: HLC in our architecture

Implementation: change in architecture - time to slide

Fig: HLC in existing architecture

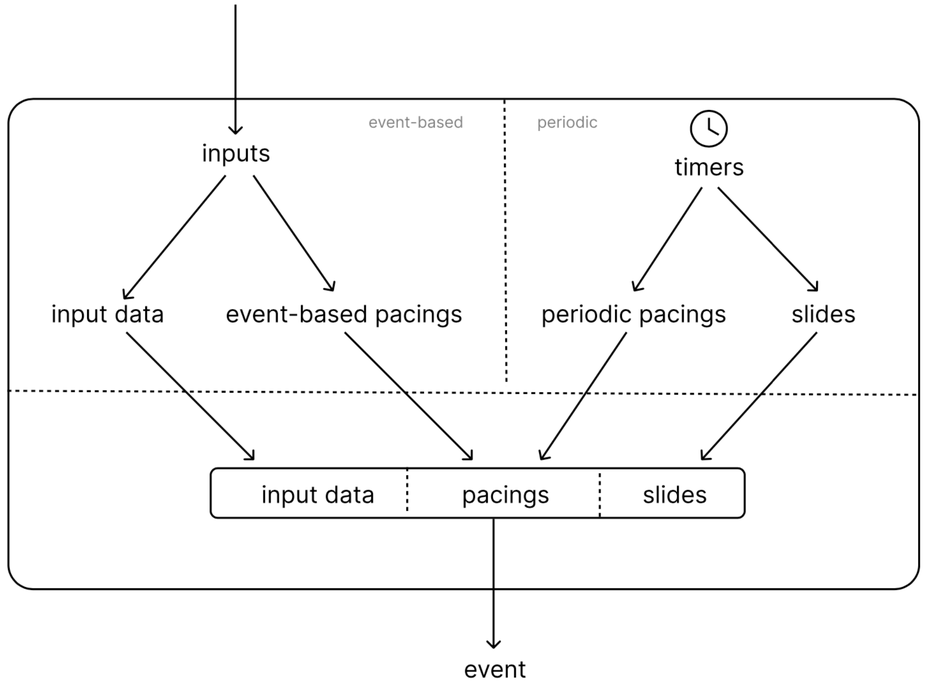


Fig: HLC in our architecture

Implementation: change in architecture - pipeline evaluation

```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64  @1Hz := b.aggregate(over: 1s, using: sum)
```

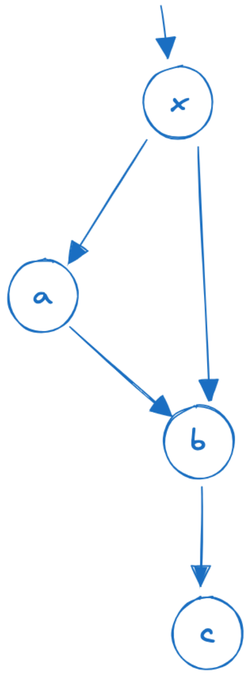


Fig: Data-flow graph

Evaluation steps

1. evaluate x
2. evaluate a
3. evaluate b
4. update sliding window sw(b, c)
5. evaluate c by aggregating sw(b, c)

Fig: without pipeline evaluation (existing architecture)

Fig: with pipeline evaluation (our architecture)

Implementation: change in architecture - pipeline evaluation

```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64  @1Hz := b.aggregate(over: 1s, using: sum)
```

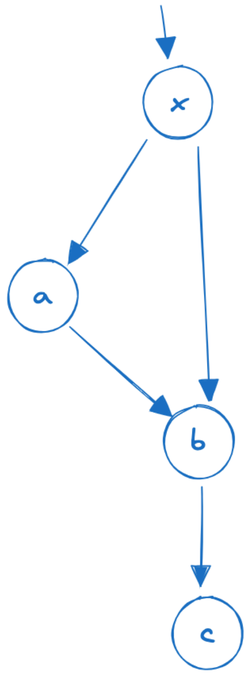


Fig: Data-flow graph

Evaluation steps

1. evaluate x
2. evaluate a
3. evaluate b
4. update sliding window sw(b, c)
5. evaluate c by aggregating sw(b, c)

Fig: without pipeline evaluation (existing architecture)

Fig: with pipeline evaluation (our architecture)

Implementation: change in architecture - pipeline evaluation

```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64  @1Hz := b.aggregate(over: 1s, using: sum)
```

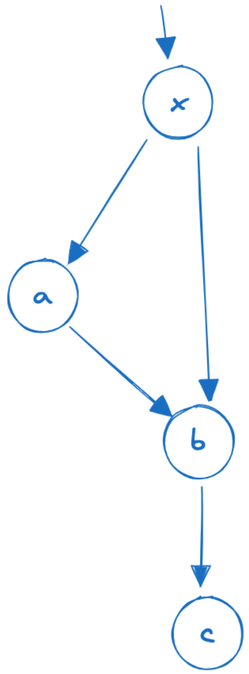


Fig: Data-flow graph

Evaluation steps

1. evaluate x
2. evaluate a
3. evaluate b
4. update sliding window sw(b, c)
5. evaluate c by aggregating sw(b, c)

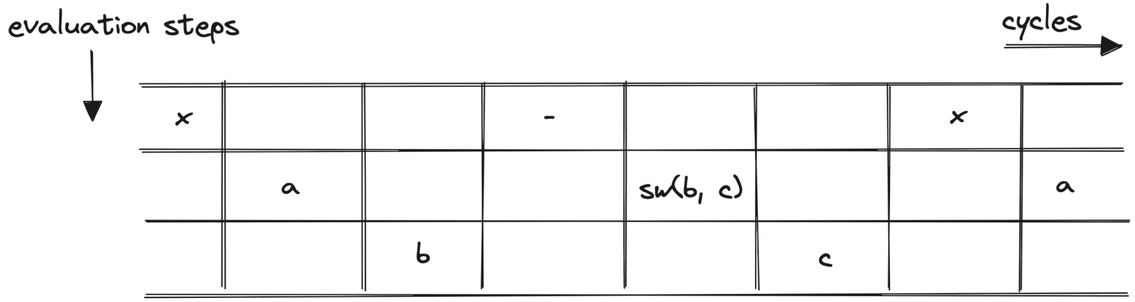


Fig: without pipeline evaluation (existing architecture)

Fig: with pipeline evaluation (our architecture)

Implementation: change in architecture - pipeline evaluation

```
input x : Int

output a:Int64 @x  := x + 1
output b:Int64 @x  := a + x
output c:Int64  @1Hz := b.aggregate(over: 1s, using: sum)
```

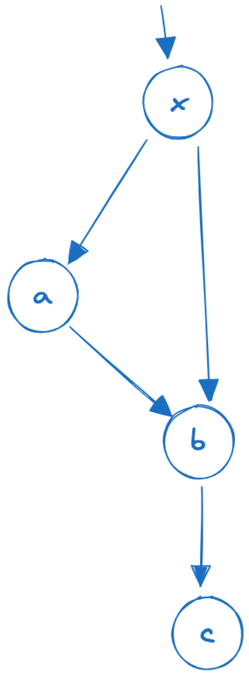


Fig: Data-flow graph

Evaluation steps

1. evaluate x
2. evaluate a
3. evaluate b
4. update sliding window sw(b, c)
5. evaluate c by aggregating sw(b, c)

Fig: without pipeline evaluation (existing architecture)

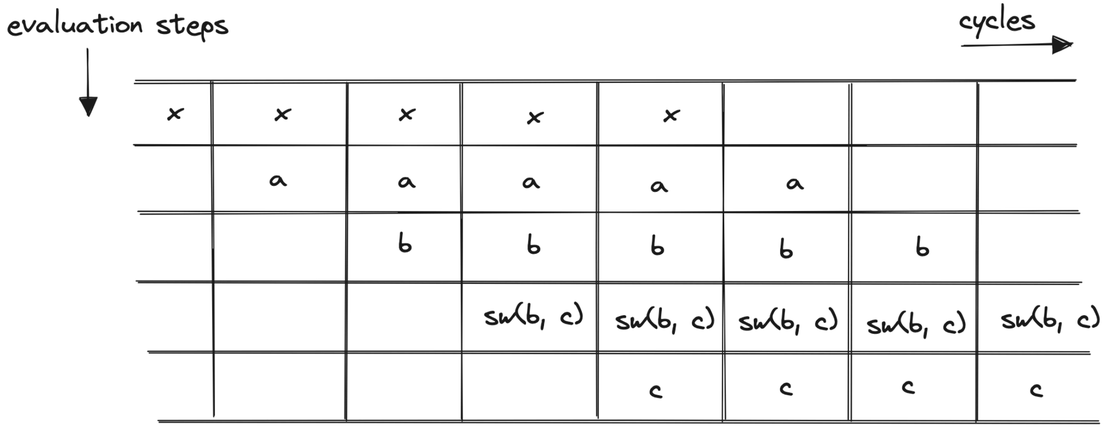


Fig: with pipeline evaluation (our architecture)

Idea behind pipeline evaluation

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -3).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```

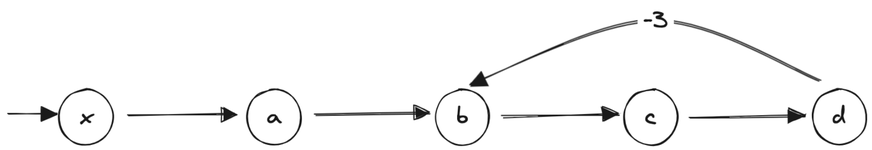


Fig: pipelined evaluation with offset(by: -3)

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -1).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```


Fig: pipelined evaluation with offset(by: -1)

Idea behind pipeline evaluation

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -3).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```

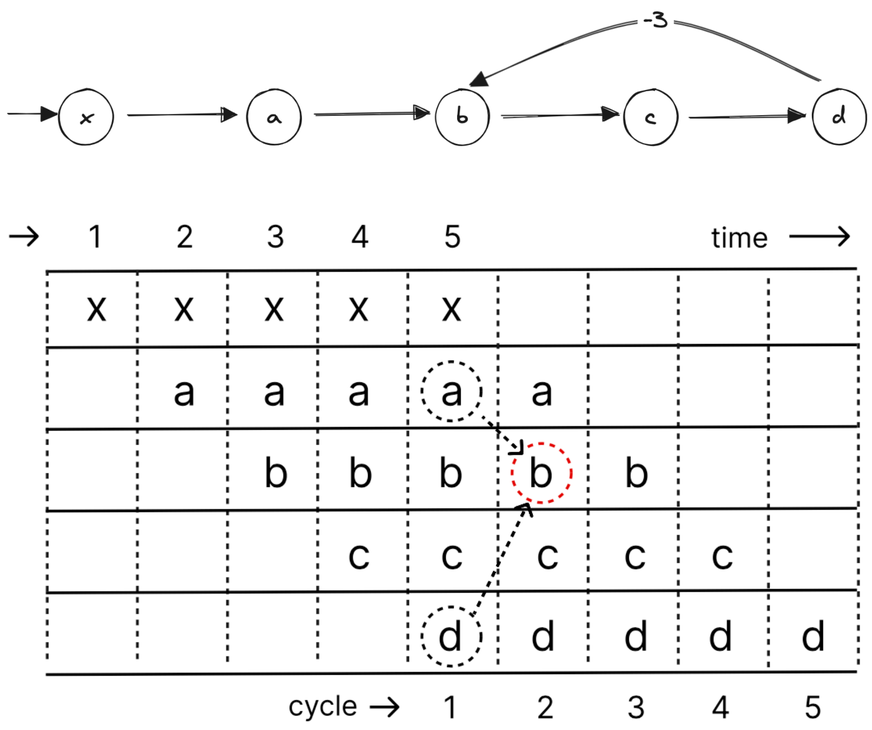


Fig: pipelined evaluation with offset(by: -3)

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -1).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```


Fig: pipelined evaluation with offset(by: -1)

Idea behind pipeline evaluation

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -3).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```


Fig: pipelined evaluation with offset(by: -3)

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -1).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```

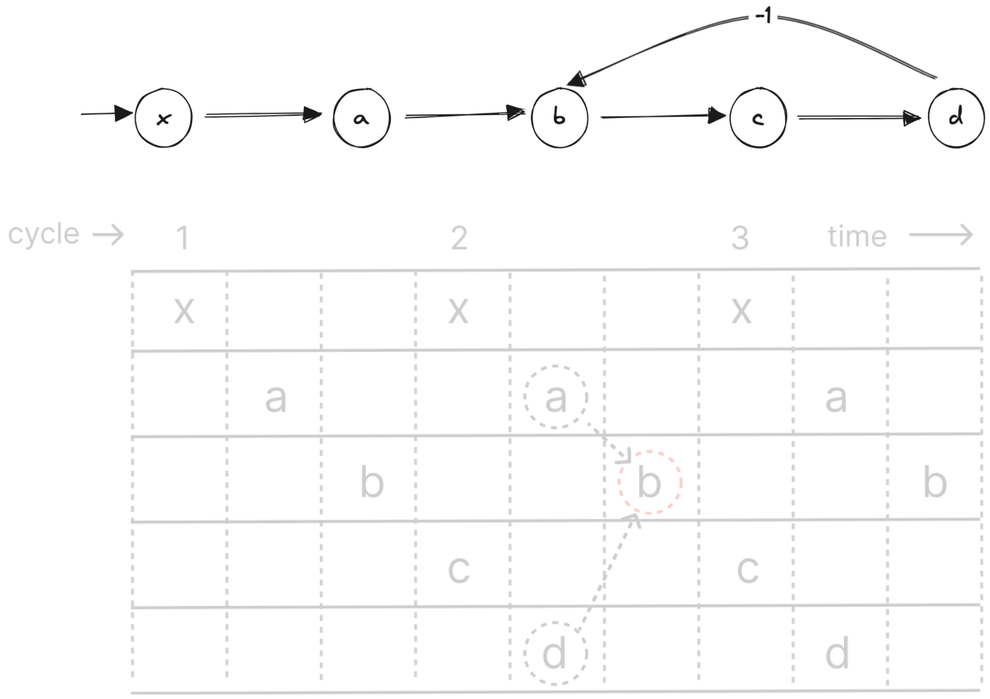


Fig: pipelined evaluation with offset(by: -1)

Idea behind pipeline evaluation

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -3).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```


Fig: pipelined evaluation with offset(by: -3)

```
input x: Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + d.offset(by: -1).defaults(to: 0)
output c:Int64 @x  := b + 1
output d:Int64 @x  := c + 1
```

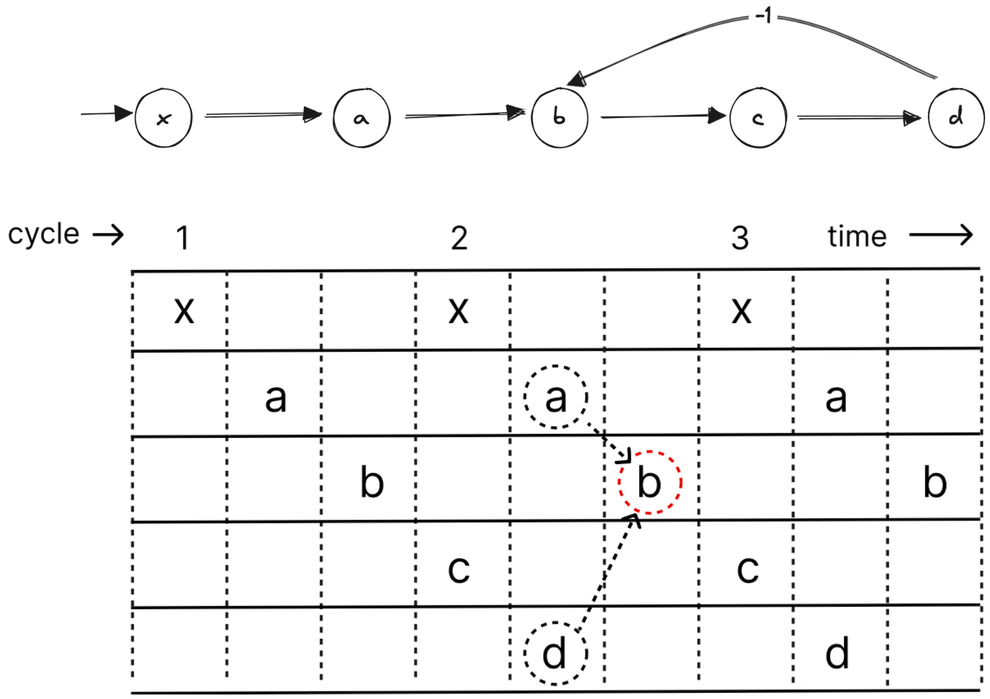


Fig: pipelined evaluation with offset(by: -1)

Memory implication of pipeline evaluation

```
input x : Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + 1
output c:Int64 @x  := x + a + b
```

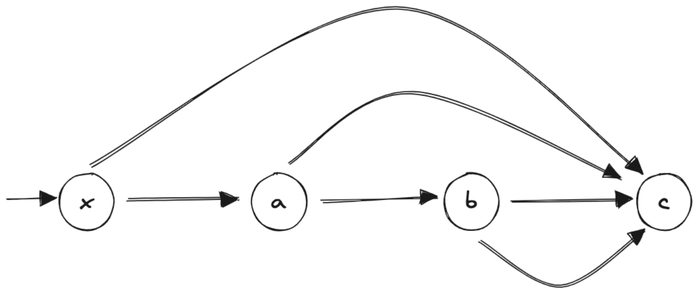
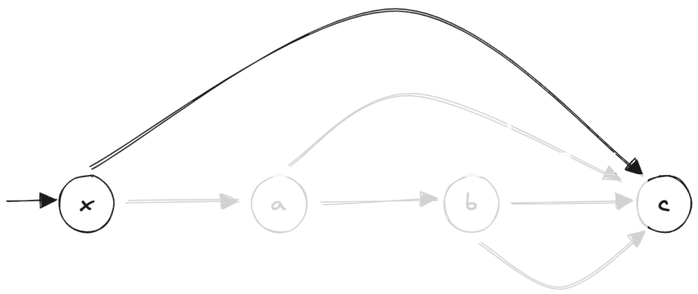


Fig: required memory window due to pipelining

Memory implication of pipeline evaluation

```
input x : Int
output a:Int64 @x := x + 1
output b:Int64 @x := a + 1
output c:Int64 @x := x + a + b
```



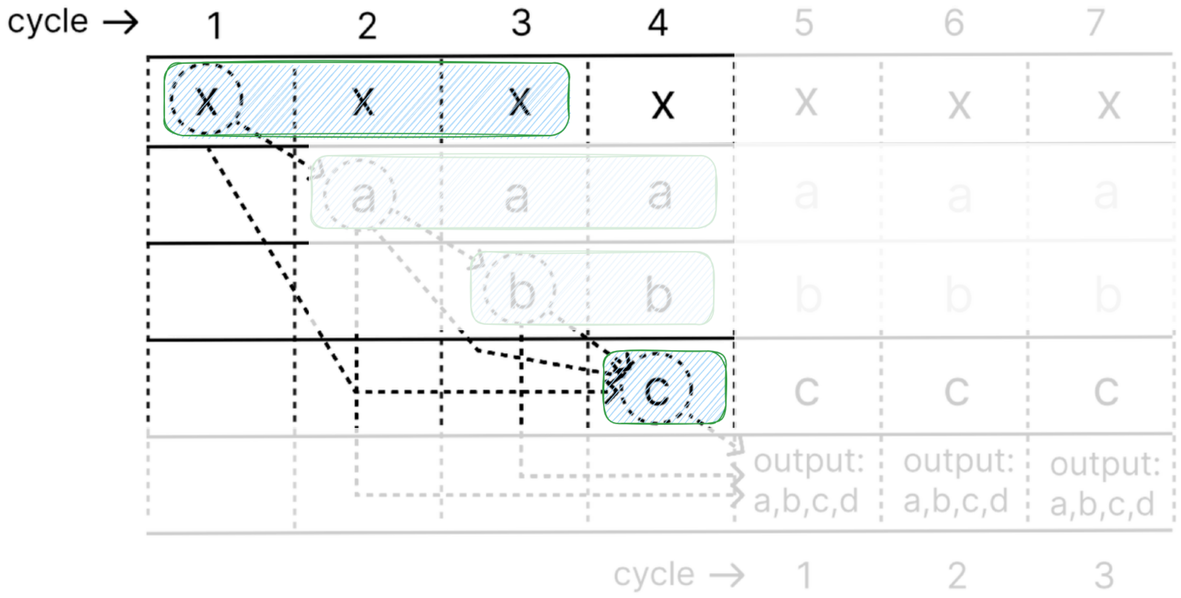
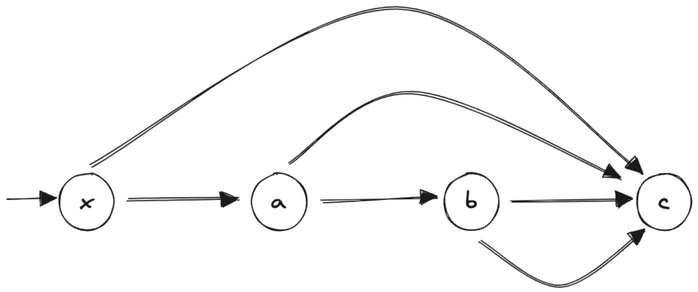


Fig: required memory window due to pipelining

Memory implication of pipeline evaluation

```
input x : Int
output a:Int64 @x := x + 1
output b:Int64 @x := a + 1
output c:Int64 @x := x + a + b
```



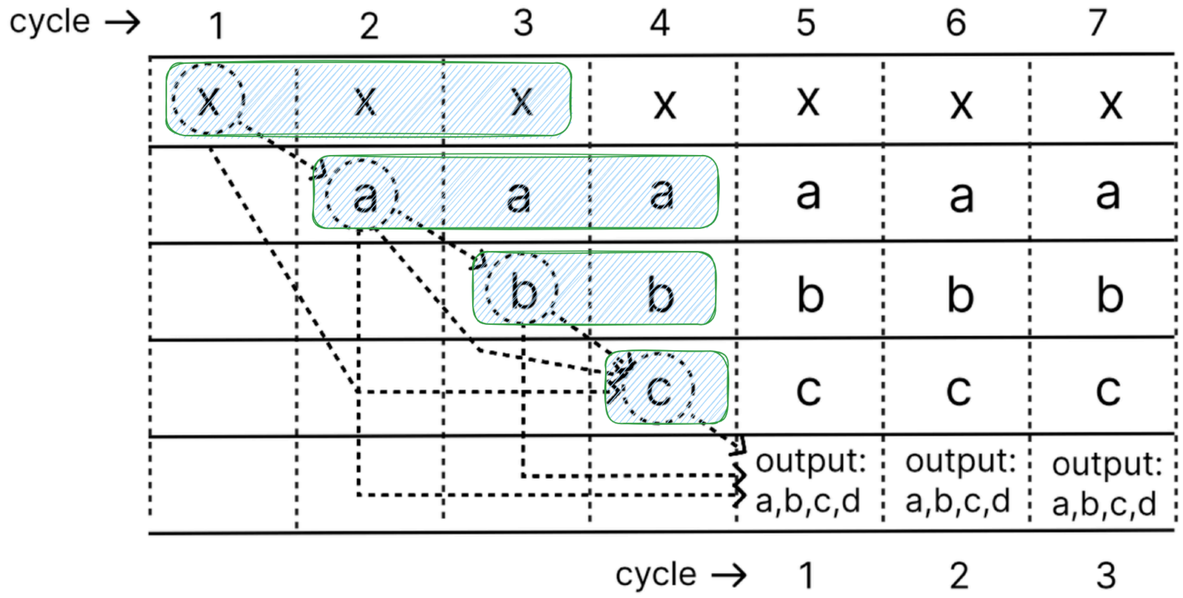
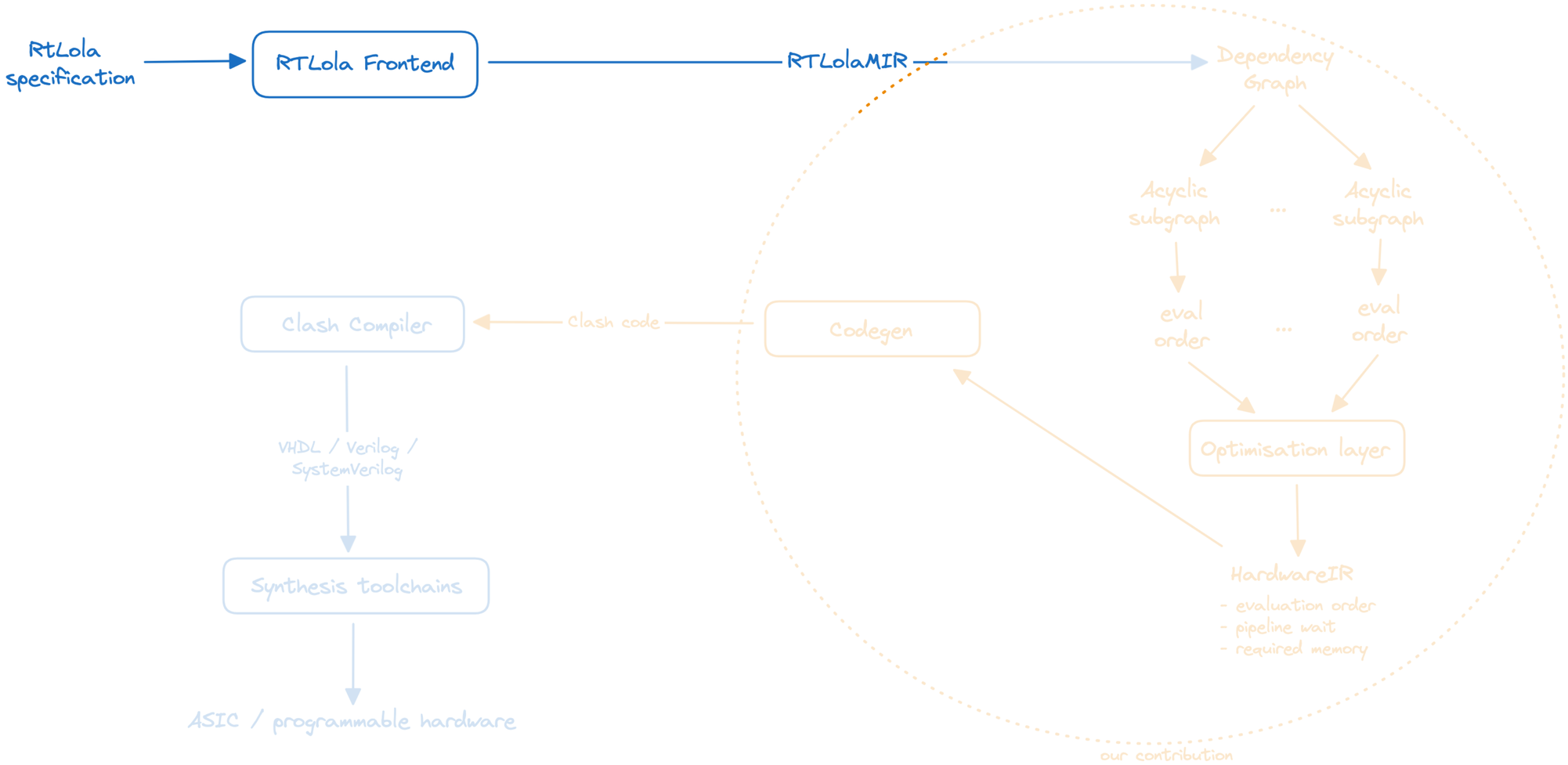
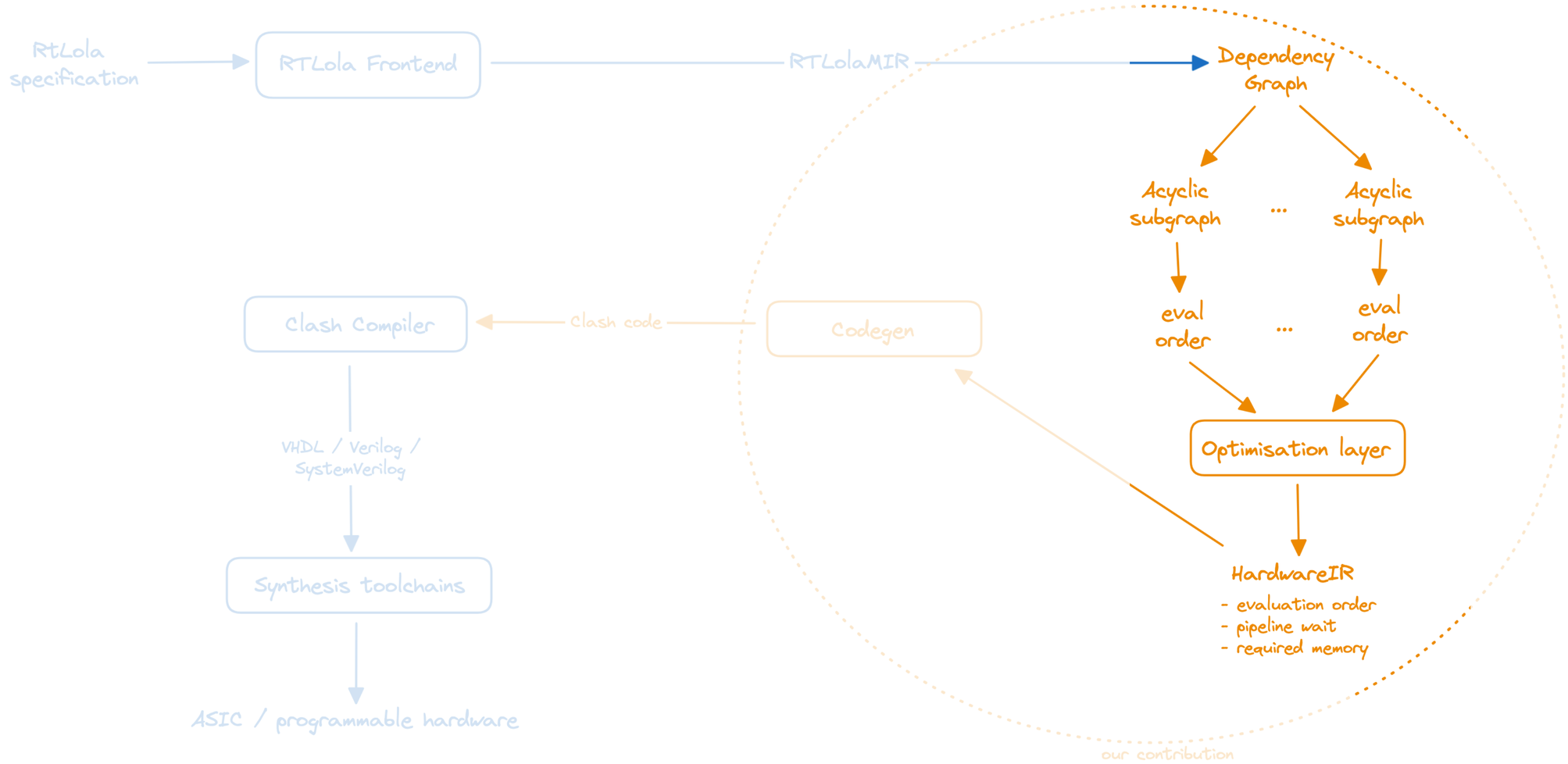


Fig: required memory window due to pipelining

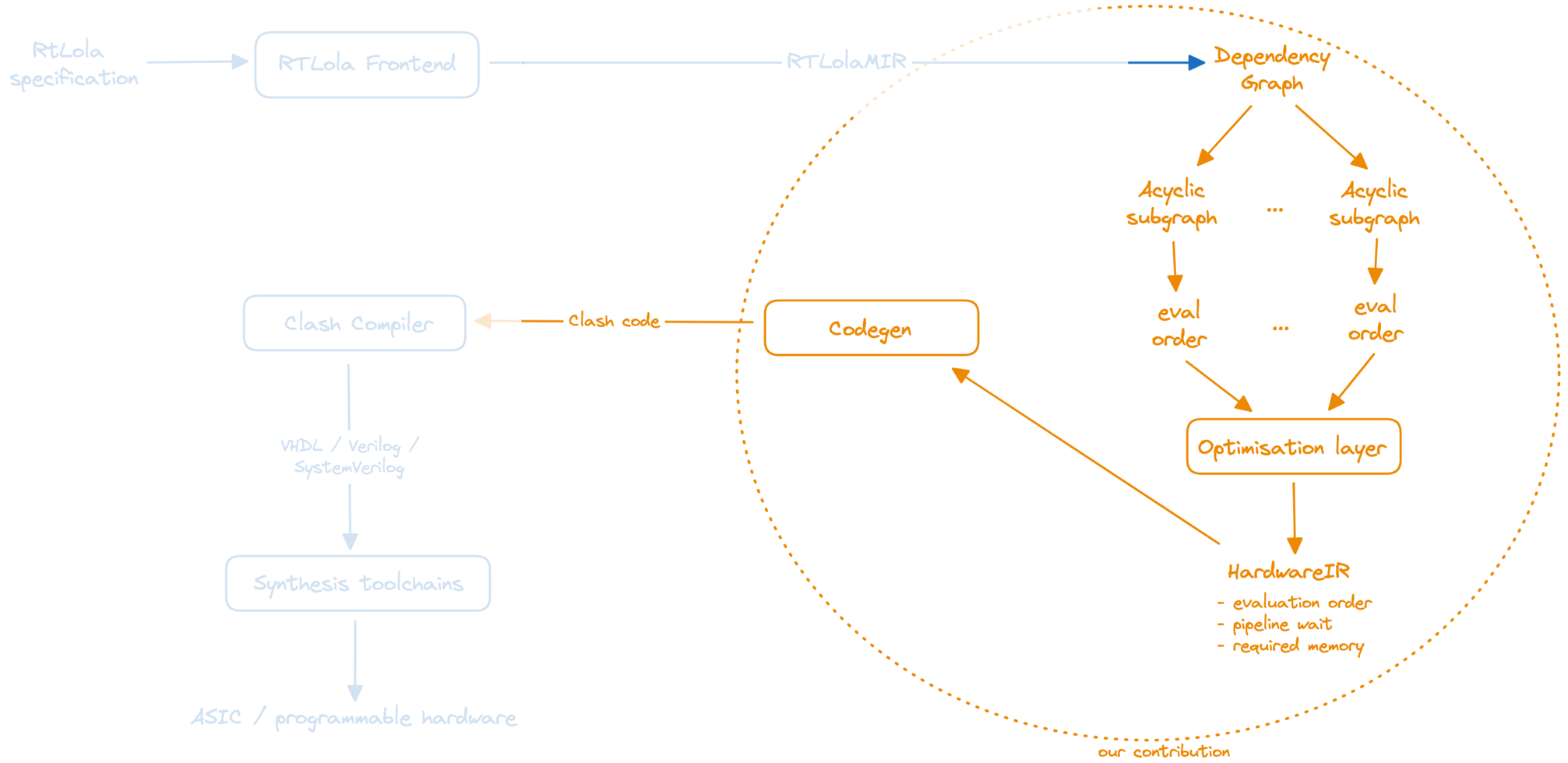
Compilation steps



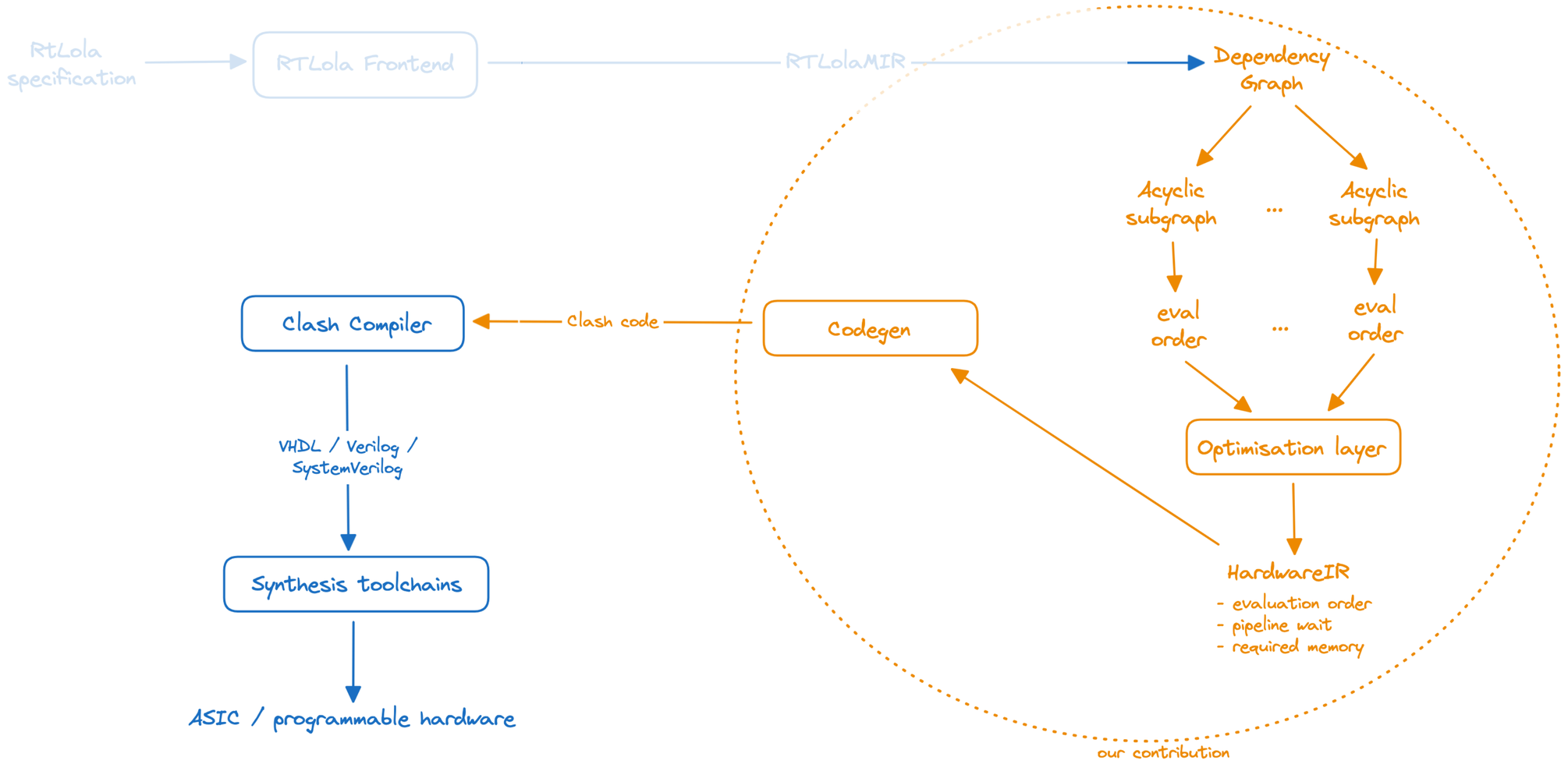
Compilation steps



Compilation steps



Compilation steps



Implementation in Clash: Pacing type system

```
input x : Int
input y : Int

output a @x := x + 1
output b @(x and y) := x + a
```

pacings

```
data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingIn1 = PacingIn1 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 PacingIn1 deriving (Generic, NFDataX)
```

@x @y

```
class Pacing a where getPacing :: a -> Bool

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingIn1 where getPacing (PacingIn1 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x0 x1) = getPacing x0 && getPacing x1
```

@(x and y)

Implementation in Clash: Pacing type system

```
input x : Int
input y : Int

output a @x := x + 1
output b @(x and y) := x + a
```

pacings

```
data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingIn1 = PacingIn1 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 PacingIn1 deriving (Generic, NFDataX)
```

@x @y

```
class Pacing a where getPacing :: a -> Bool

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingIn1 where getPacing (PacingIn1 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x0 x1) = getPacing x0 && getPacing x1
```

@(x and y)

Implementation in Clash: Pacing type system

```
input x : Int
input y : Int

output a @x := x + 1
output b @(x and y) := x + a
```

pacings

```
data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingIn1 = PacingIn1 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 PacingIn1 deriving (Generic, NFDataX)
```

@x @y

```
class Pacing a where getPacing :: a -> Bool

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingIn1 where getPacing (PacingIn1 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x0 x1) = getPacing x0 && getPacing x1
```

@(x and y)

Implementation in Clash: Pacing type system

```
input x : Int
input y : Int

output a @x := x + 1
output b @(x and y) := x + a
```

pacings

```
data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingIn1 = PacingIn1 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 PacingIn1 deriving (Generic, NFDataX)
```

@x @y

```
class Pacing a where getPacing :: a -> Bool

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingIn1 where getPacing (PacingIn1 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x0 x1) = getPacing x0 && getPacing x1
```

@(x and y)

Implementation in Clash: Pacing type system

```
input x : Int
input y : Int

output a @x := x + 1
output b @(x and y) := x + a
```

pacings

```
data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingIn1 = PacingIn1 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 PacingIn1 deriving (Generic, NFDataX)
```

@x @y

```
class Pacing a where getPacing :: a -> Bool

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingIn1 where getPacing (PacingIn1 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x0 x1) = getPacing x0 && getPacing x1
```

@(x and y)

Implementation in Clash: LLC - controlling pipeline

```haskell
llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        ...
```

pop when ready

```haskell
pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 1 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                   (mux (toWait .>. pure 0) (toWait - 1) toWait)
```

ready when no need to wait

Fig: Part of LLC - control of pipeline

Implementation in Clash: LLC - controlling pipeline

```haskell
llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        ...
```

pop when ready

```haskell
pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 1 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                   (mux (toWait .>. pure 0) (toWait - 1) toWait)
```

ready when no need to wait

Fig: Part of LLC - control of pipeline

Implementation in Clash: LLC - controlling pipeline

```haskell
llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        ...
```

pop when ready

```haskell
pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 1 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                   (mux (toWait .>. pure 0) (toWait - 1) toWait)
```

ready when no need to wait

Fig: Part of LLC - control of pipeline

Implementation in Clash: LLC - controlling pipeline

```haskell
llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        ...
```

pop when ready

```haskell
pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 1 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                   (mux (toWait .>. pure 0) (toWait - 1) toWait)
```

ready when no need to wait

Fig: Part of LLC - control of pipeline

Implementation in Clash: LLC - controlling pipeline

```haskell
llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        ...
```

pop when ready

```haskell
pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 1 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                   (mux (toWait .>. pure 0) (toWait - 1) toWait)
```

ready when no need to wait

Fig: Part of LLC - control of pipeline

Implementation in Clash: LLC - controlling evaluation order

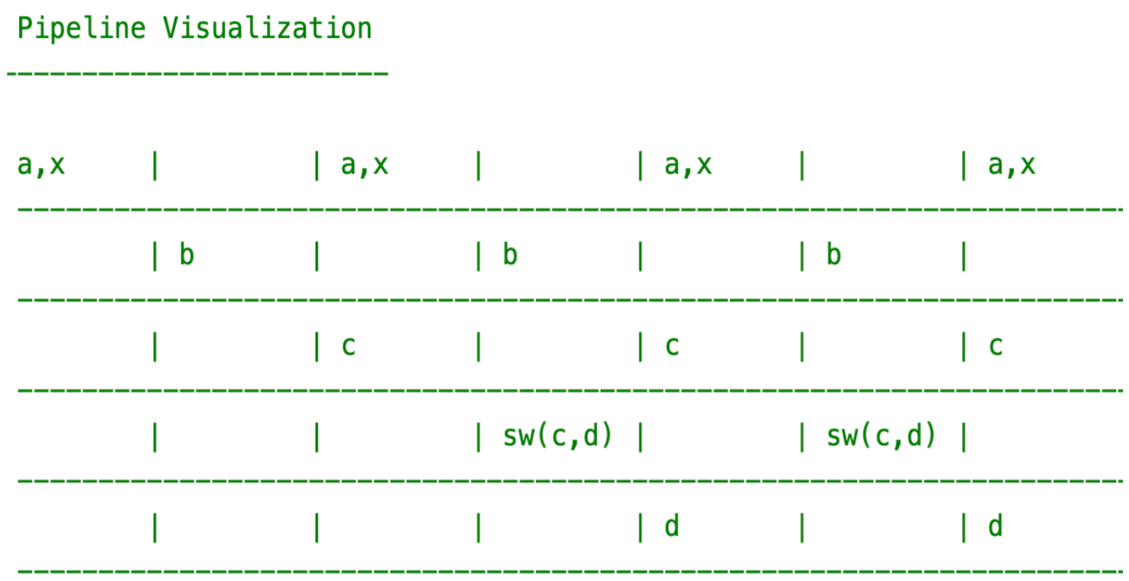
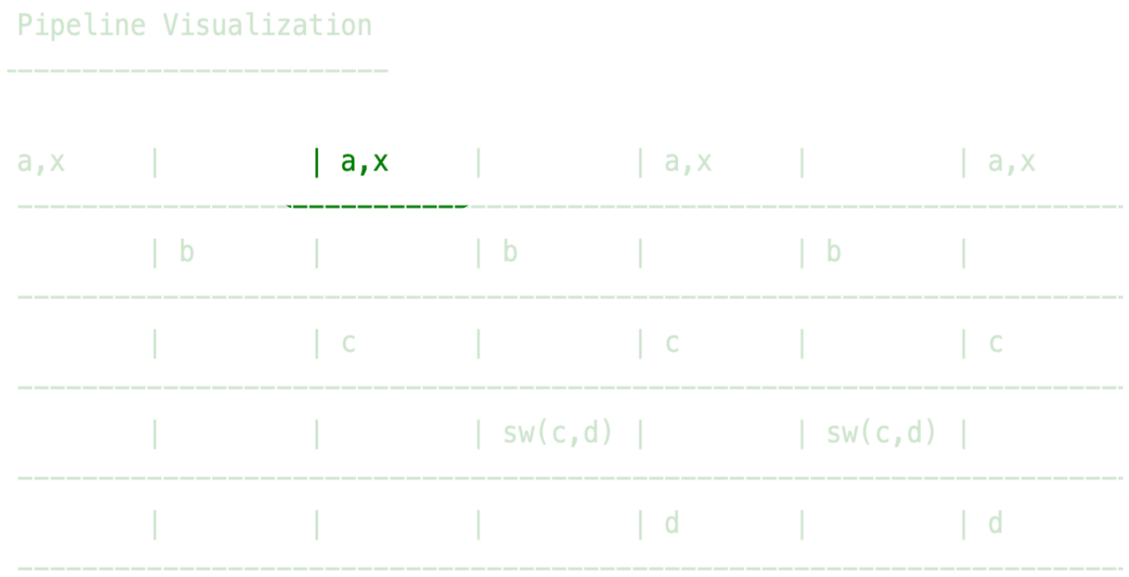


Fig: Controlling evaluation order

Implementation in Clash: LLC - controlling evaluation order



resulting delayed pacing

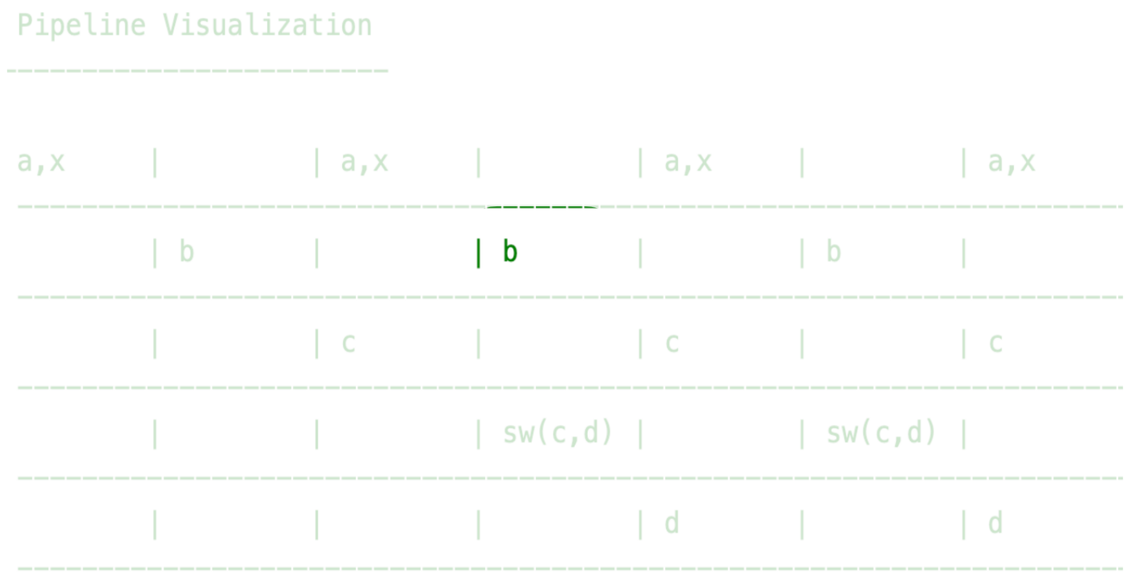
delay the stream by one cycle

pacing stream

```
enIn0 = delayFor d1 nullPacingIn0 pIn0
enOut0 = delayFor d1 nullPacingOut0 pOut0
enOut1 = delayFor d2 nullPacingOut1 pOut1
enOut2 = delayFor d3 nullPacingOut2 pOut2
enSw0 = delayFor d4 nullPacingOut2 pOut2
sld0 = delayFor d4 False slide0
enOut3 = delayFor d5 nullPacingOut3 pOut3
```

Fig: Controlling evaluation order

Implementation in Clash: LLC - controlling evaluation order



resulting delayed pacing

delay the stream by one cycle

pacing stream

```
enIn0 = delayFor d1 nullPacingIn0 pIn0
enOut0 = delayFor d1 nullPacingOut0 pOut0
enOut1 = delayFor d2 nullPacingOut1 pOut1
enOut2 = delayFor d3 nullPacingOut2 pOut2
enSw0 = delayFor d4 nullPacingOut2 pOut2
sld0 = delayFor d4 False slide0
enOut3 = delayFor d5 nullPacingOut3 pOut3
```

Fig: Controlling evaluation order

Implementation in Clash: LLC - controlling evaluation order

```
Pipeline Visualization
----------------------

a,x     |         | a,x     |         | a,x     |         | a,x
--------------------------------------------------------------------
        | b       |         | b       |         | b       |
--------------------------------------------------------------------
        |         | c       |         | c       |         | c
--------------------------------------------------------------------
        |         |         | sw(c,d) |         | sw(c,d) |
--------------------------------------------------------------------
        |         |         |         | d       |         | d
--------------------------------------------------------------------
```

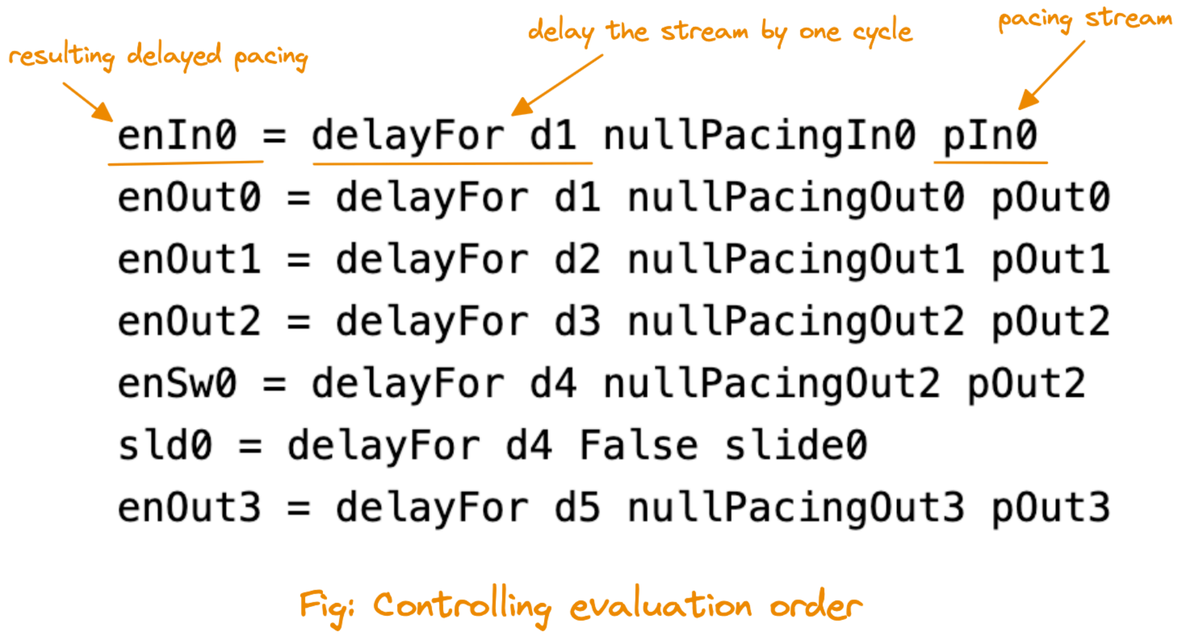
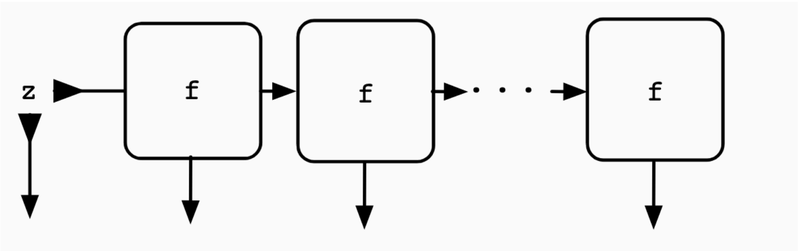


Fig: Controlling evaluation order

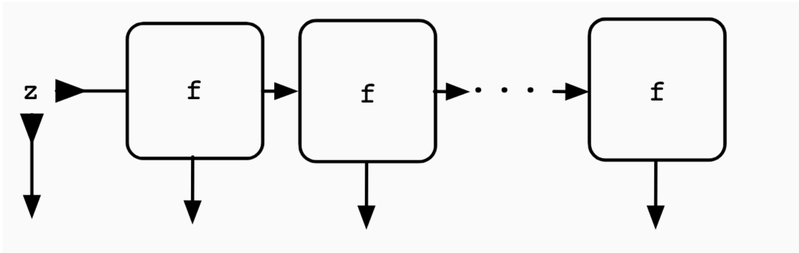
Implementation in Clash: delayFor function

```haskell
delayFor :: forall dom n a .
    (HiddenClockResetEnable dom, KnownNat n, NFDataX a)
    => SNat n
    -> a
    -> Signal dom a
    -> Signal dom a
delayFor n initVal sig = last delayedVec
    where
        delayedVec :: Vec (n + 1) (Signal dom a)
        delayedVec = iterateI (delay initVal) sig
```



Implementation in Clash: delayFor function

```haskell
delayFor :: forall dom n a .
    (HiddenClockResetEnable dom, KnownNat n, NFDataX a)
    => SNat n
    -> a
    -> Signal dom a
    -> Signal dom a
delayFor n initVal sig = last delayedVec
    where
        delayedVec :: Vec (n + 1) (Signal dom a)
        delayedVec = iterateI (delay initVal) sig
```



Implementation in Clash: LLC - stream evaluation

```
output d:Int64  @1kHz := x.hold(or: 0) + c.aggregate(over: 0.01s, using: sum)
```

Fig: Specification snippet

```
outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream3 en tag in0_0 sw0 = result
    where
        result = register (invalidTag, 0)
                 (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + (merge0 <$> sw0))
        merge0 :: Vec 11 Int -> Int
        merge0 win = fold windowFunc0 (tail win)
```

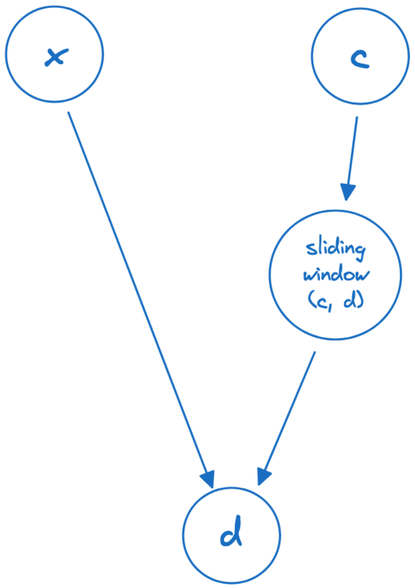
pacing

sliding window

activated by pacing

x + c.aggregate()

Fig: Example stream evaluation function



Implementation in Clash: LLC - stream evaluation

```
output d:Int64 @1kHz := x.hold(or: 0) + c.aggregate(over: 0.01s, using: sum)
```

Fig: Specification snippet

```
outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream3 en tag in0_0 sw0 = result
    where
        result = register (invalidTag, 0)
                 (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + (merge0 <$> sw0))
        merge0 :: Vec 11 Int -> Int
        merge0 win = fold windowFunc0 (tail win)
```

pacing

sliding window

activated by pacing

x + c.aggregate()

Fig: Example stream evaluation function

Implementation in Clash: LLC - stream evaluation

```
output d:Int64 @1kHz := x.hold(or: 0) + c.aggregate(over: 0.01s, using: sum)
```

Fig: Specification snippet

```
outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream3 en tag in0_0 sw0 = result
    where
        result = register (invalidTag, 0)
                 (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + (merge0 <$> sw0))
        merge0 :: Vec 11 Int -> Int
        merge0 win = fold windowFunc0 (tail win)
```

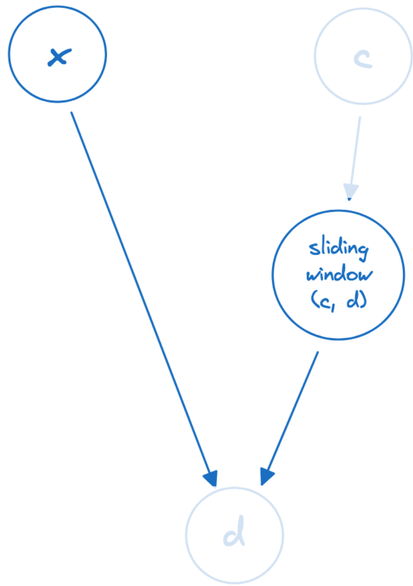
pacing

sliding window

activated by pacing

x + c.aggregate()

Fig: Example stream evaluation function



Implementation in Clash: LLC - stream evaluation

```
output d:Int64 @1kHz := x.hold(or: 0) + c.aggregate(over: 0.01s, using: sum)
```

Fig: Specification snippet

```
outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream3 en tag in0_0 sw0 = result
    where
        result = register (invalidTag, 0)
                 (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + (merge0 <$> sw0))
        merge0 :: Vec 11 Int -> Int
        merge0 win = fold windowFunc0 (tail win)
```

pacing

sliding window

activated by pacing

x + c.aggregate()

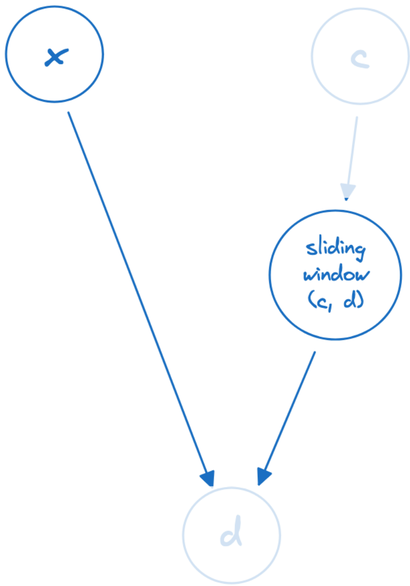


Fig: Example stream evaluation function

Implementation in Clash: LLC - stream evaluation

```
output d:Int64  @1kHz := x.hold(or: 0) + c.aggregate(over: 0.01s, using: sum)
```

Fig: Specification snippet

```
outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream3 en tag in0_0 sw0 = result
    where
        result = register (invalidTag, 0)
                 (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + (merge0 <$> sw0))
        merge0 :: Vec 11 Int -> Int
        merge0 win = fold windowFunc0 (tail win)
```

pacing

sliding window

activated by pacing

x + c.aggregate()

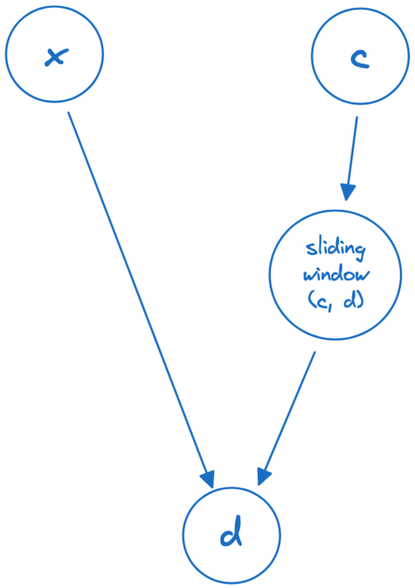


Fig: Example stream evaluation function

Implementation in Clash: LLC - sliding window evaluation

```haskell
slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 11 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                    <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 11 Int
```

new input data OR time to slide

```haskell
nextWindow :: Vec 11 Int
    -> Bool
    -> Bool
    -> Int
    -> Vec 11 Int
nextWindow win toSlide newData dta = out
    where
        out = case (toSlide, newData) of
            (False, False) -> win
            (False, True) -> lastBucketUpdated
            (True, False) -> 0 +>> win
            (True, True) -> 0 +>> lastBucketUpdated
        lastBucketUpdated =
            replace 0 (windowFunc0 (head win) dta) win
```

window bucket update and slide

Implementation in Clash: LLC - sliding window evaluation

```haskell
slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 11 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                    <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 11 Int
```

new input data OR time to slide

```haskell
nextWindow :: Vec 11 Int
    -> Bool
    -> Bool
    -> Int
    -> Vec 11 Int
nextWindow win toSlide newData dta = out
    where
        out = case (toSlide, newData) of
            (False, False) -> win
            (False, True) -> lastBucketUpdated
            (True, False) -> 0 +>> win
            (True, True) -> 0 +>> lastBucketUpdated
        lastBucketUpdated =
            replace 0 (windowFunc0 (head win) dta) win
```

window bucket update and slide

Implementation in Clash: LLC - sliding window evaluation

```haskell
slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 11 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 11 Int
```

new input data OR time to slide

```haskell
nextWindow :: Vec 11 Int
    -> Bool
    -> Bool
    -> Int
    -> Vec 11 Int
nextWindow win toSlide newData dta = out
    where
        out = case (toSlide, newData) of
            (False, False) -> win
            (False, True) -> lastBucketUpdated
            (True, False) -> 0 +>> win
            (True, True) -> 0 +>> lastBucketUpdated
        lastBucketUpdated =
            replace 0 (windowFunc0 (head win) dta) win
```

window bucket update and slide

Implementation in Clash: LLC - sliding window evaluation

```haskell
slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 11 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 11 Int
```

new input data OR time to slide

```haskell
nextWindow :: Vec 11 Int
    -> Bool
    -> Bool
    -> Int
    -> Vec 11 Int
nextWindow win toSlide newData dta = out
    where
        out = case (toSlide, newData) of
            (False, False) -> win
            (False, True) -> lastBucketUpdated
            (True, False) -> 0 +>> win
            (True, True) -> 0 +>> lastBucketUpdated
        lastBucketUpdated =
            replace 0 (windowFunc0 (head win) dta) win
```

window bucket update and slide

Implementation in Clash: LLC - sliding window evaluation

```
slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 11 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 11 Int
```

new input data OR time to slide

```
nextWindow :: Vec 11 Int
    -> Bool
    -> Bool
    -> Int
    -> Vec 11 Int
nextWindow win toSlide newData dta = out
    where
        out = case (toSlide, newData) of
            (False, False) -> win
            (False, True) -> lastBucketUpdated
            (True, False) -> 0 +>> win
            (True, True) -> 0 +>> lastBucketUpdated
        lastBucketUpdated =
            replace 0 (windowFunc0 (head win) dta) win
```

window bucket update and slide

Evaluation: Synthesised hardware primitives

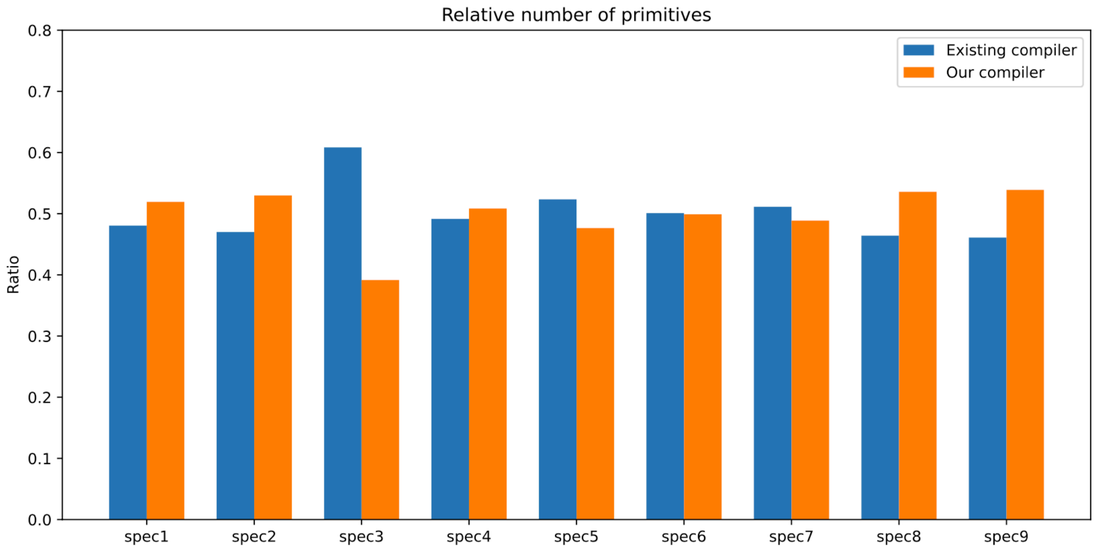


Fig: Relative number of synthesized primitives

Fig: Ratio of sequential primitives over all primitives

Evaluation: Synthesised hardware primitives

Fig: Relative number of synthesized primitives

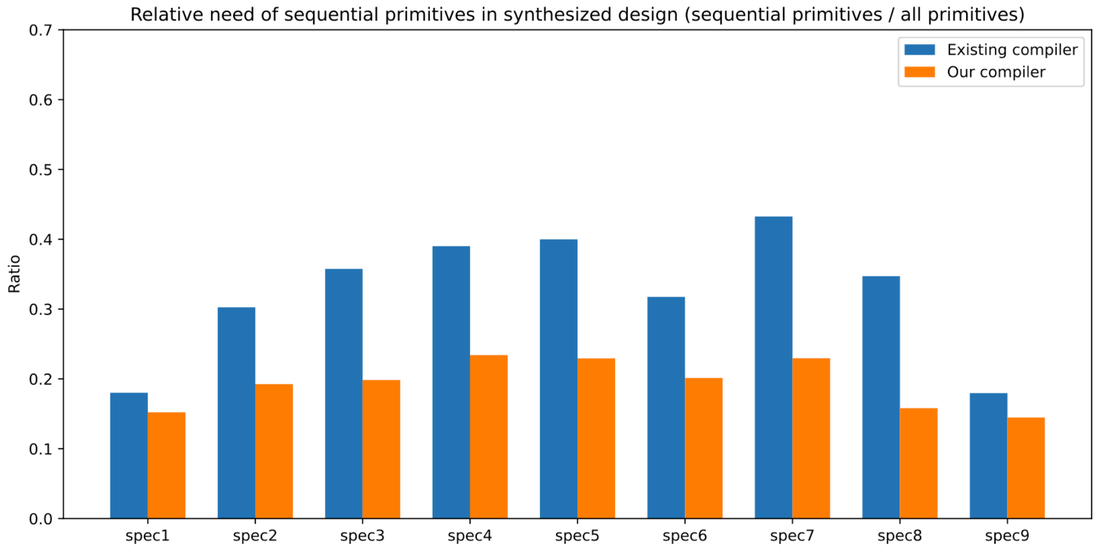


Fig: Ratio of sequential primitives over all primitives

Evaluation: Evaluation throughput

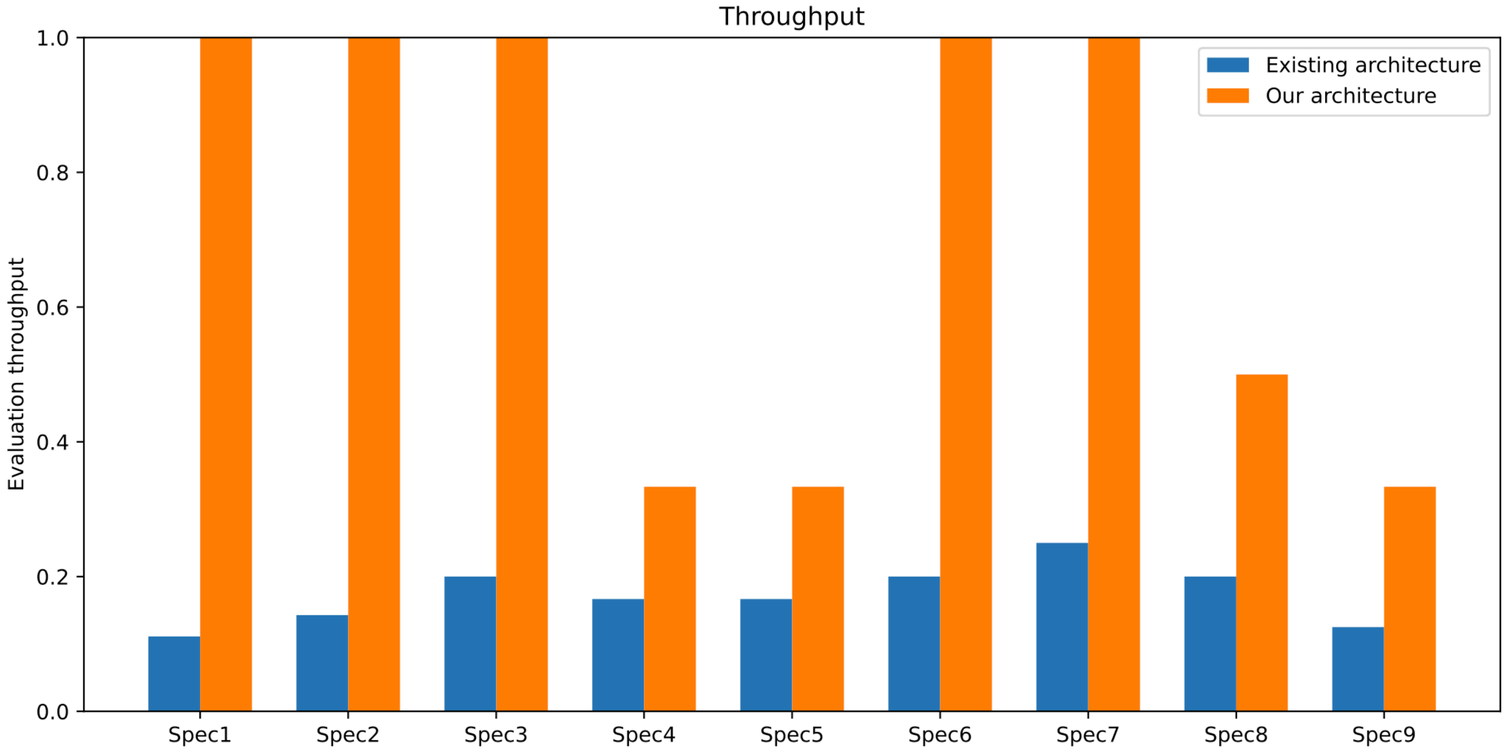
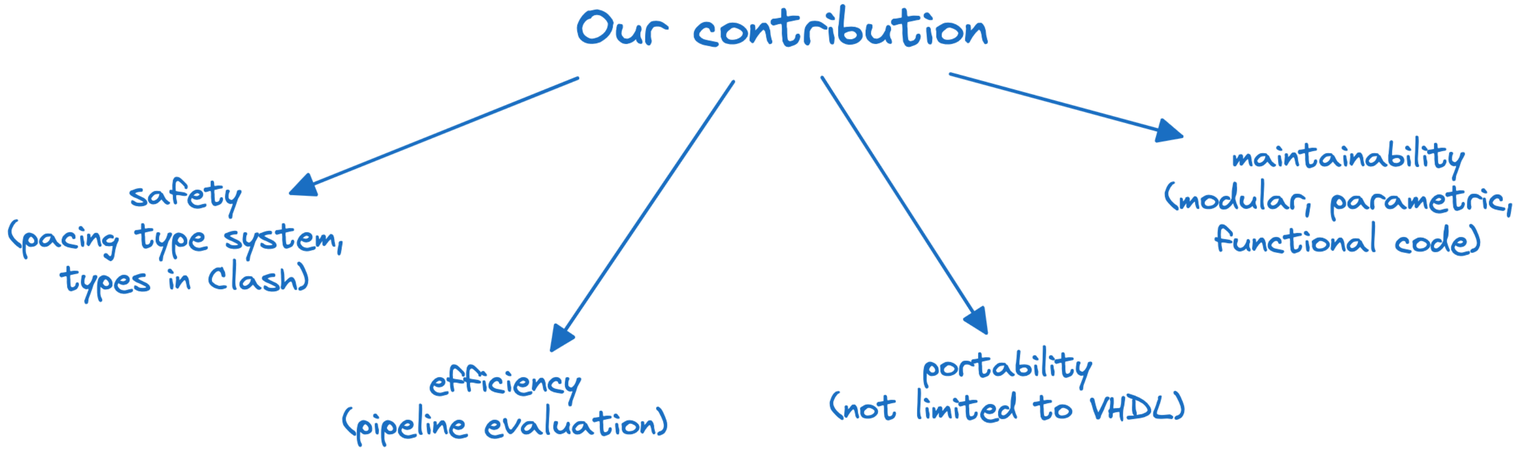


Fig: Evaluation throughput compared with existing architecture

Conclusion


Our contribution
safety
(pacing type system,
types in Clash)
efficiency
(pipeline evaluation)
portability
(not limited to VHDL)
maintainability
(modular, parametric,
functional code)

Thanks

Extra slides

1. Optimisation of evaluation order
2. Evaluation order algorithm
3. Why tag results in a window?
4. Why event-based before periodic semantics in RTLola?
5. Non input root node
6. Queue in Clash
7. HLC in Clash
8. Merging of event-based & periodic
9. Complete generated code example

Opportunity to optimise evaluation order

```
input x : Int
output a:Int64 @x  := x + 1
output b:Int64 @x  := a + 1
output c:Int64 @x  := b.offset(by: -2).defaults(to: 0) + 1
output d:Int64 @x  := c + 1
```

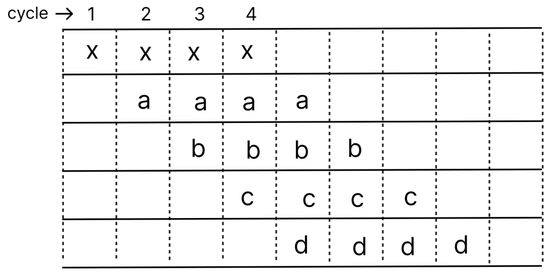


Fig: normal order

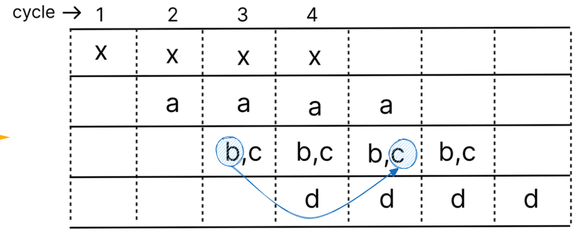


Fig: evaluating c & d one step earlier

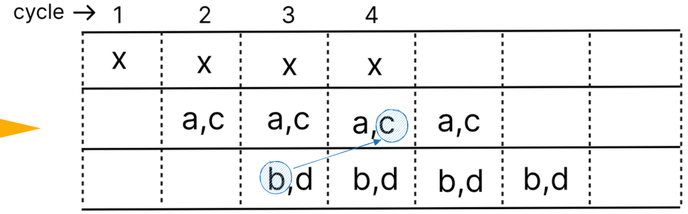


Fig: evaluating c & d two step earlier

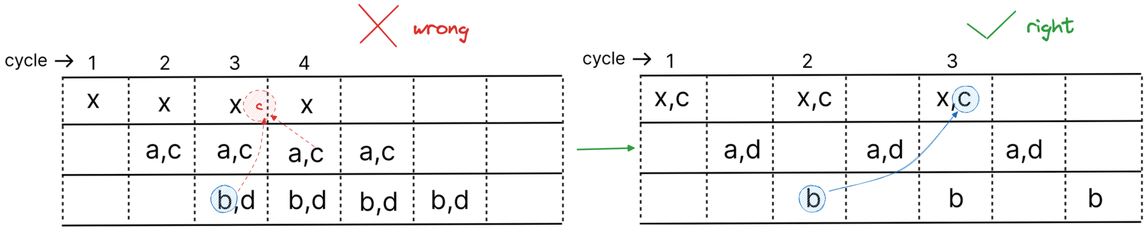
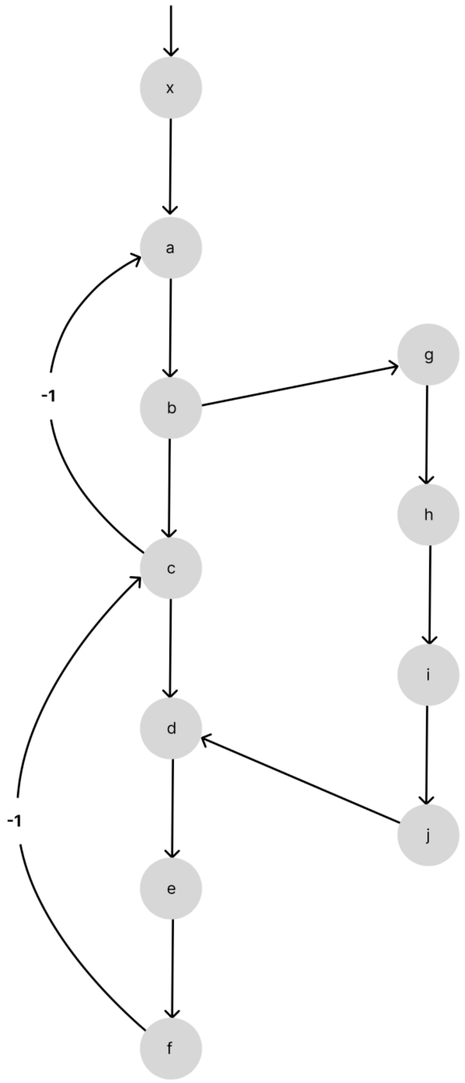


Fig: evaluating c & d even earlier requires pipeline_wait

Optimisation tradeoffs: offset tug of war




```
x
a -----
b      |
   g   | wait: 5
   h   |
   i   |
c, j --
d      | wait: 3
e      |
f -----
```

Evalution order option: 1

```
x
a -----
b      |
   g   | wait: 4
   h   |
c, i --
  j    |
d      | wait: 4
e      |
f -----
```

Evalution order option: 2

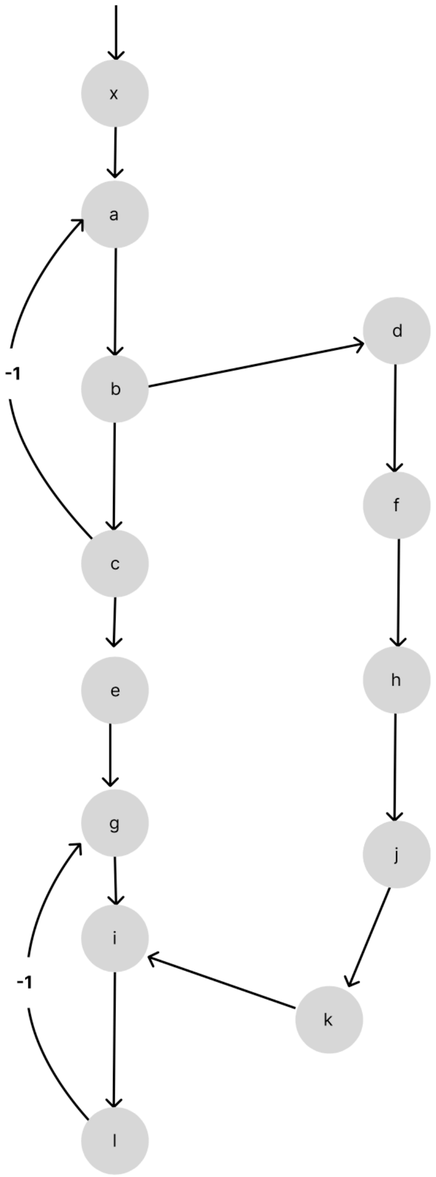
```
x
a -----
b      |
   g   | wait: 3
c, h --
   i   |
   j   |
d      | wait: 5
e      |
f -----
```

Evalution order option: 3

```
x
a -----
b      | wait: 2
c, g --
   h   |
   i   |
   j   |
d      | wait: 6
e      |
f -----
```

Evalution order option: 4

Optimisation tradeoffs: stretching of order



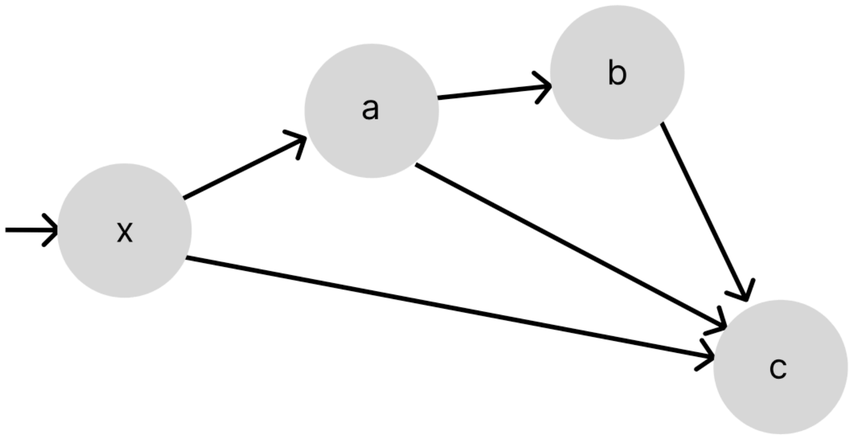
results must be stored until consumed

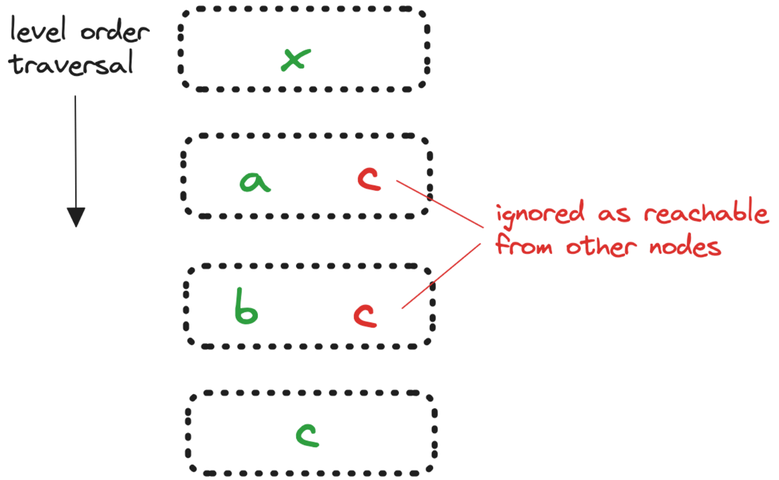
position of e can influence total memory

```
x
a -----
b       | wait: 2
c, d --
   f
e, h
   j
g, k --
i       | wait: 2
l -----
```

Fig: Stretching of order $c \rightarrow e \rightarrow g$

Evaluation order algorithm: Acyclic graph





Cycles in RTLola

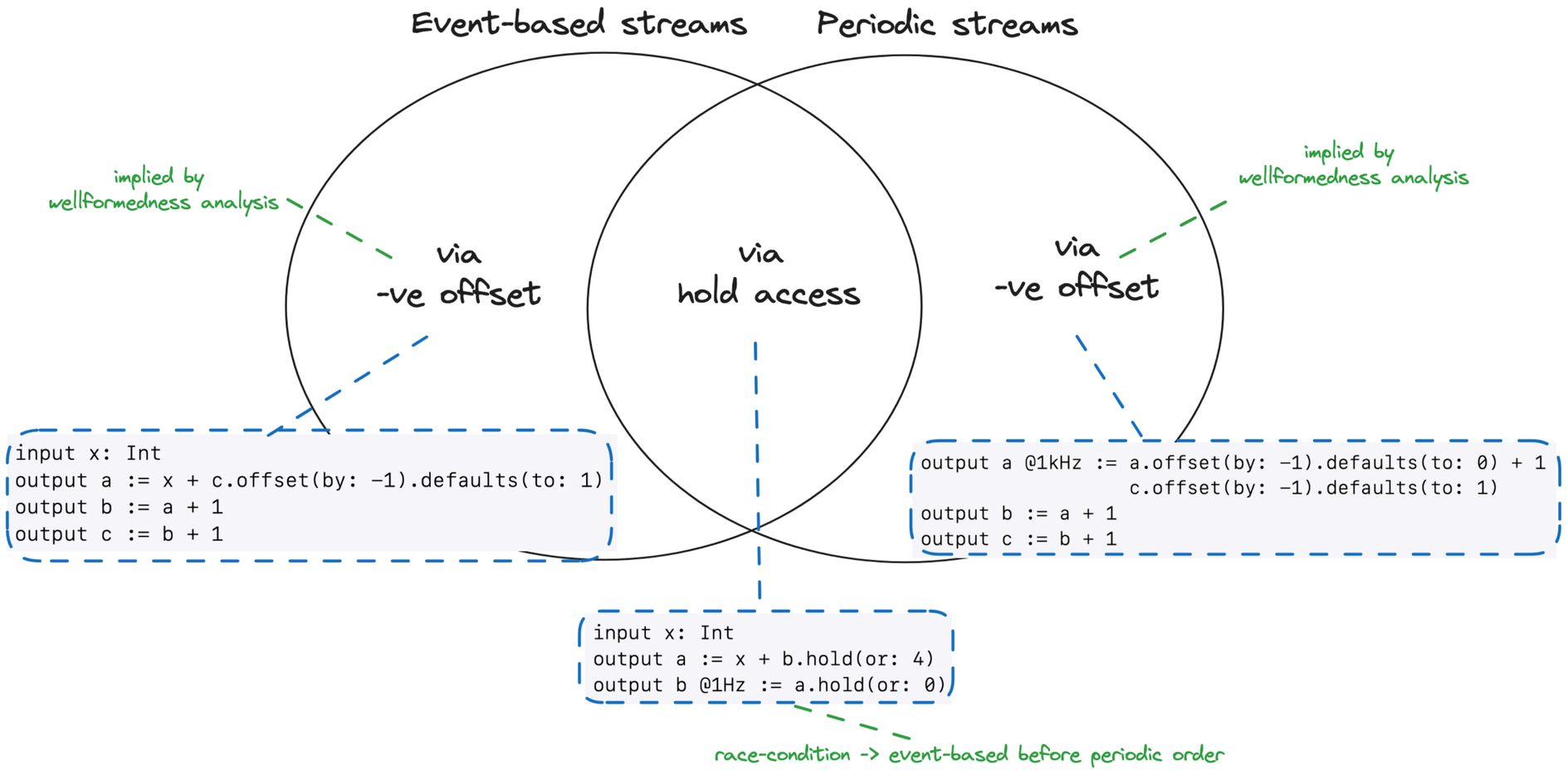
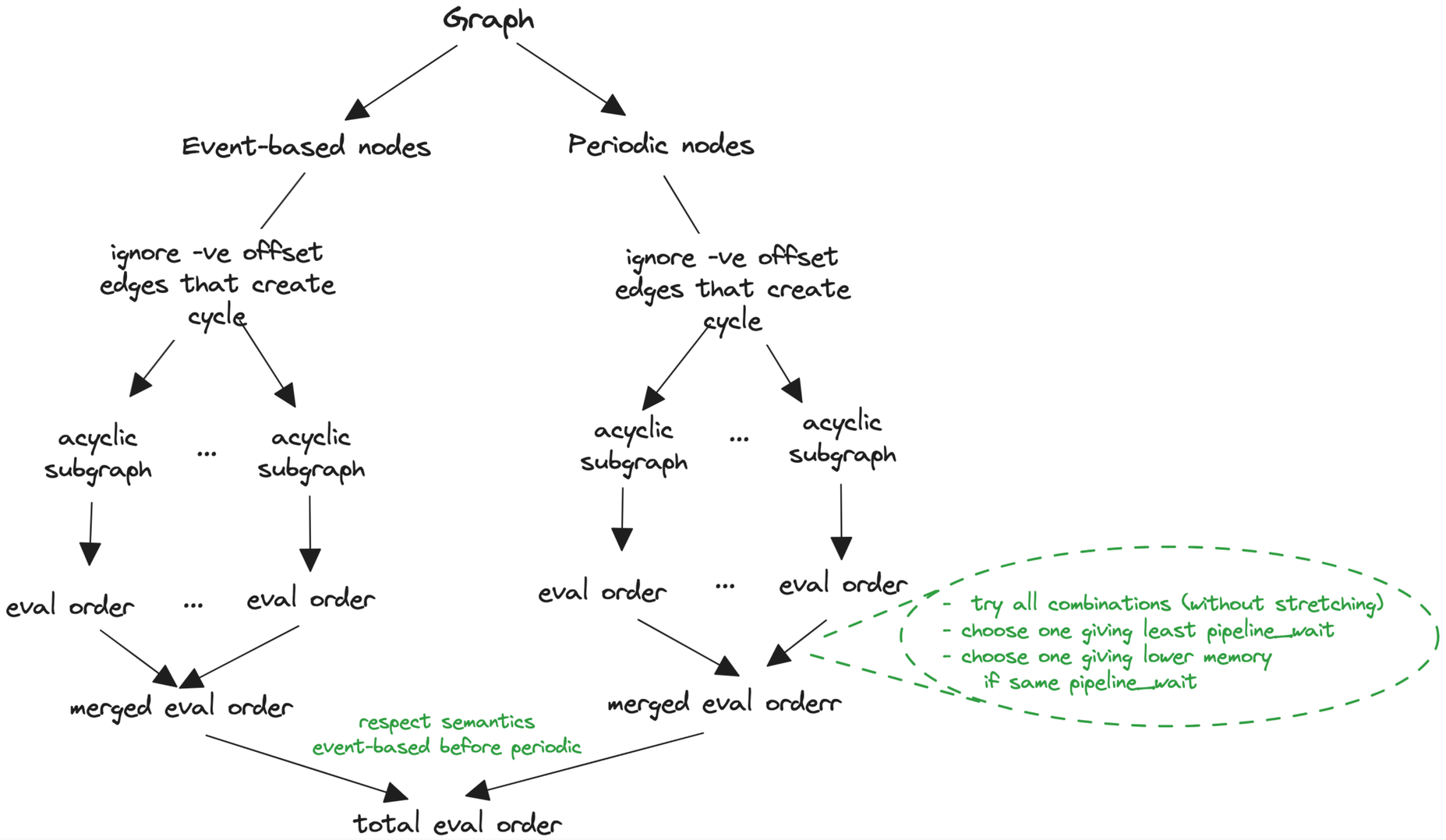


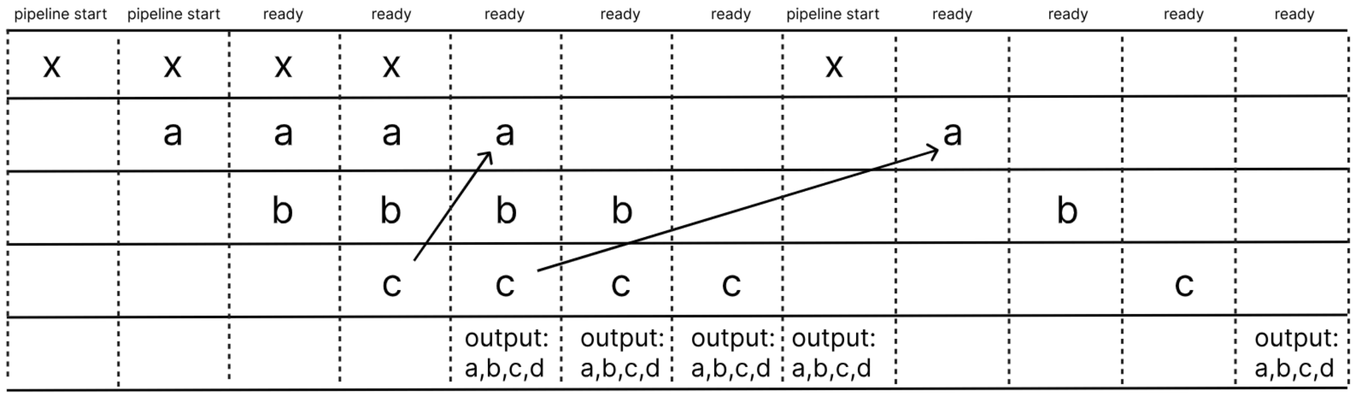
Fig: Cycles in RTLola

Heuristic evaluation order algorithm



Why Tag results in window?

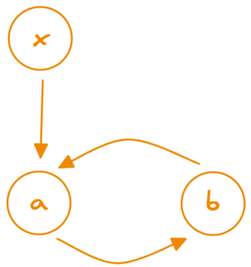
```
output a := c.offset(by: -3).defaults(to: 0)
```




```haskell
getOffset :: KnownNat n => Vec n (Tag, a) -> Tag -> Tag -> a -> a
getOffset win tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = case findIndex (\(t, _) -> t == offsetTag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt
```

Why event-based before periodic?

```
input x: Int
output a := x + b.hold(or: 4)
output b @1Hz := a.hold(or: 0)
```




```
input x: Int
output a := if sum_so_far.hold(or: 0) > 10
            then x
            else x + sum_so_far.hold(or: 0)
output sum_so_far @1kHz := a.aggregate(over: 1s, using: sum)
```

Non input root node

```
output a @1Hz := a.offset(by: -1).defaults(to: 0) + 1
output b := a + 1
output c := b + 1
```

Queue in Clash

```haskell
queue :: HiddenClockResetEnable dom
    => Signal dom QInput
    -> Signal dom QOutput
queue input = output
    where
        output = bundle (pushValid, popValid, outData)
        state = bundle (buffer, cursor)
        buffer = register (repeat nullEvent :: QMem) nextBufferSignal
        cursor = register 0 nextCursorSignal
        pushValid = register False nextPushValidSignal
        popValid = register False nextPopValidSignal
        outData = register nullEvent nextOutDataSignal

        nextBufferSignal = nextBuffer
                           <$> buffer
                           <*> bundle (input, cursor)
        nextCursorSignal = nextCursor
                           <$> cursor
                           <*> bundle (input, buffer)
        nextOutDataSignal = nextOutData
                            <$> bundle (input, cursor, buffer)
        nextPushValidSignal = nextPushValid
                              <$> bundle (input, cursor, buffer)
        nextPopValidSignal = nextPopValid <$> bundle (input, cursor)
```

```haskell
nextBuffer :: QMem -> (QInput, QCursor) -> QMem
nextBuffer buf ((push, pop, qData), cur) = out
    where
        out = case (push, pop) of
            (True, True) -> qData +>> buf
            (True, False) -> if cur == length buf
                             then buf else qData +>> buf
            (False, _) -> buf
```

```haskell
nextCursor :: QCursor -> (QInput, QMem) -> QCursor
nextCursor cur ((push, pop, _), buf) = out
    where
        out = case (push, pop) of
            (True, False) -> if cur == length buf
                             then cur else cur + 1
            (False, True) -> if cur == 0 then 0 else cur - 1
            (_, _) -> cur
```

```haskell
nextOutData :: (QInput, QCursor, QMem) -> QData
nextOutData ((push, pop, qData), cur, buf) = out
    where
        out = case (push, pop) of
            (True, True) -> if cur == 0
                            then qData else buf !! (cur - 1)
            (False, True) -> if cur == 0
                             then nullEvent else buf !! (cur - 1)
            (_, _) -> nullEvent
```

```haskell
nextPushValid :: (QInput, QCursor, QMem) -> QPush
nextPushValid ((push, pop, _), cur, buf) = out
    where
        out = case (push, pop) of
            (True, True) -> True
            (True, False) -> cur /= length buf
            (False, _) -> False
```

```haskell
nextPopValid :: (QInput, QCursor) -> QPop
nextPopValid ((push, pop, _), cur) = out
    where
        out = case (push, pop) of
            (True, True) -> True
            (False, True) -> cur /= 0
            (_, False) -> False
```

Implementation in Clash: HLC

```
input x: Int
input y: Int

output a:Int64 @(x ∧ y)  := x.offset(by: -1).defaults(to: 0) + y
output b:Int64  @1Hz := x.hold(or: 0) + a.aggregate(over: 0.5s, using: sum)
```

Fig: specification

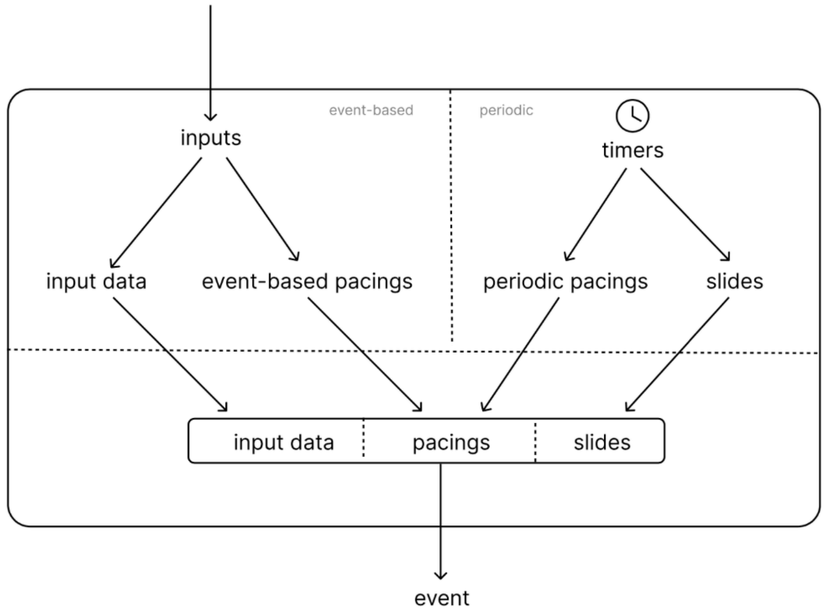
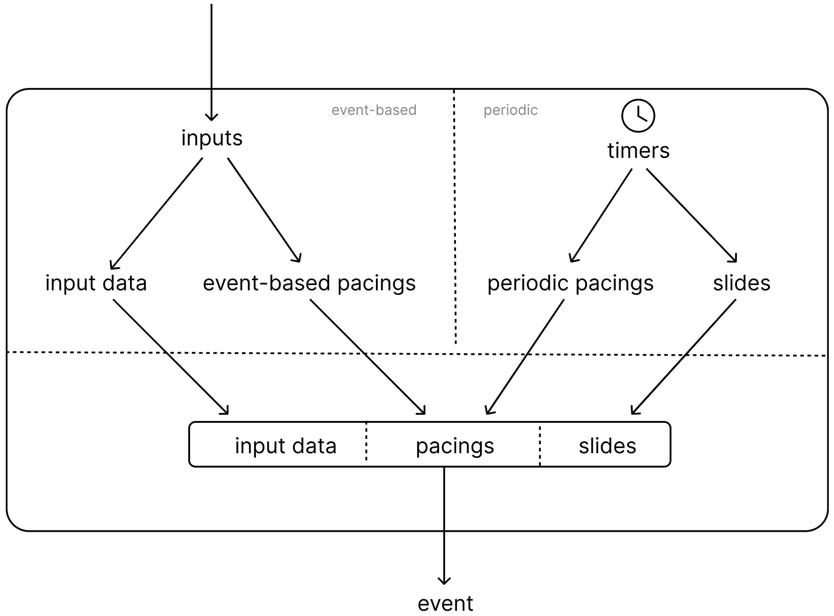


Fig: HLC structure

```
hlc :: HiddenClockResetEnable dom
    => Signal dom Inputs
    -> Signal dom (Bool, Event)
hlc inputs = out
    where
        out = bundle (newEvent, event)
        newEvent = hasInput0 .||. hasInput1 .||. timer0Over .||. timer1Over
        event = bundle (inputs, slides, pacings)

        slides = Slides <$> s0
        pacings = Pacings <$> pIn0
                          <*> pIn1
                          <*> pOut0
                          <*> pOut1
```

Fig: Segment of HLC code in Clash

Implementation in Clash: HLC - stream activation



pacings

```
pIn0 = PacingIn0 <$> hasInput0
pIn1 = PacingIn1 <$> hasInput1
pOut0 = PacingOut0 <$> pIn0 <*> pIn1
pOut1 = PacingOut1 <$> timer1Over
```

```
timer1Over = timer1 .>=. period1InNs
timer1 = timer timer1Over
period1InNs = 1000000000
```

```
timer :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Int
timer reset = register 0 (mux reset (pure deltaTime) nextTime)
    where
        nextTime = timer reset + pure deltaTime
        deltaTime = systemClockPeriodNs
```

slides

```
s0 = timer0Over
```

Eval order merge: event-based & periodic

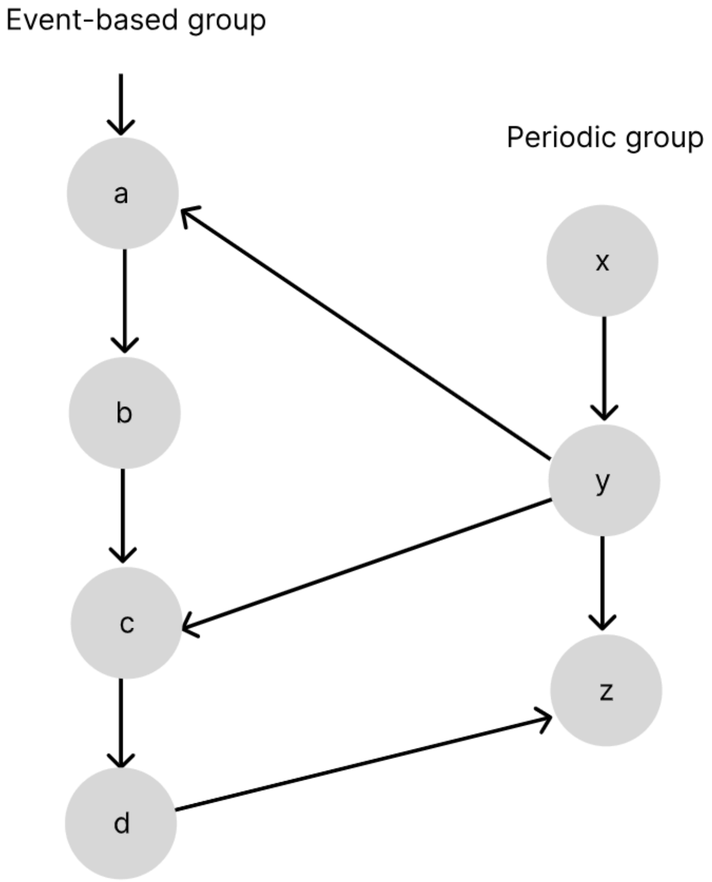


Figure 41: Graph

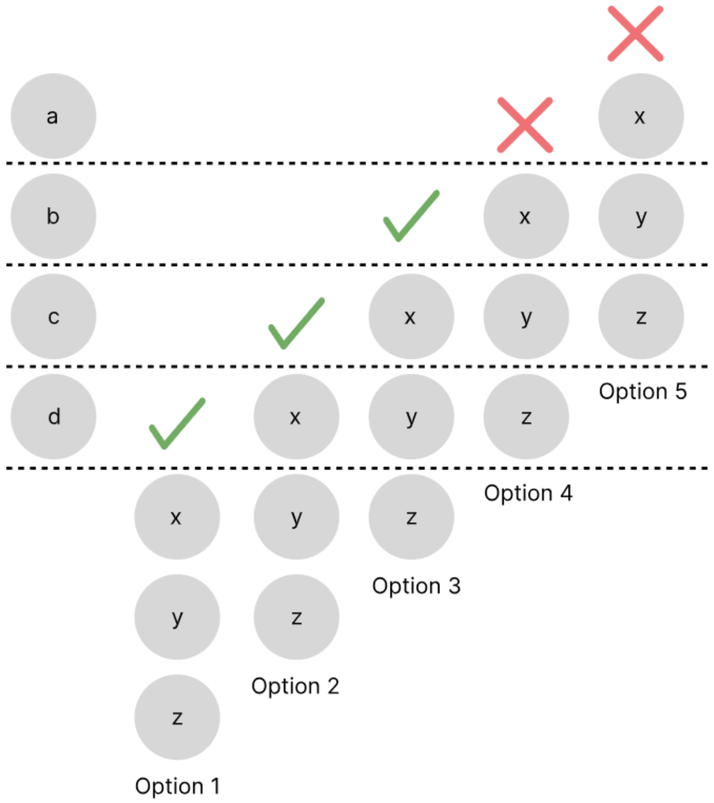


Figure 42: Merge options

Complete code - part 1

```haskell
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE OverloadedRecordDot #-}

module Spec where

import Clash.Prelude

------------------------------------------------------------------

-- input x : Int
-- output a := x.offset(by: -1).defaults(to: 0)
-- output b := a + c.offset(by: -1).defaults(to: x + a)
-- output c := b + 1
-- output d @1kHz := x.hold(or: 0) + c.aggregate(over: 0.01s, using: sum)

------------------------------------------------------------------

-- Evaluation Order
--------------------
-- a, x
-- b
-- c
-- sw(c,d)
-- d

-- Memory Window
-----------------
-- window c = 2
-- window sw(c,d) = 1
-- window a = 3
-- window b = 2
-- window d = 1
-- window x = 2

-- Pipeline Visualization
--------------------------

-- a,x     |         | a,x     |         | a,x     |         | a,x     |         | a,x     |
-- ------------------------------------------------------------------------------------------
--         | b       |         | b       |         | b       |         | b       |         | b
-- ------------------------------------------------------------------------------------------
--         |         | c       |         | c       |         | c       |         | c       |
-- ------------------------------------------------------------------------------------------
--         |         |         | sw(c,d) |         | sw(c,d) |         | sw(c,d) |         | sw(c,d)
-- ------------------------------------------------------------------------------------------
--         |         |         |         | d       |         | d       |         | d       |
-- ------------------------------------------------------------------------------------------

-- Nicknames
-------------
-- input0 = x
-- output0 = a
```

```haskell
-- Nicknames
-------------
-- input0 = x
-- output0 = a
-- output1 = b
-- output2 = c
-- output3 = d
-- sw0 = sw(c,d)

------------------------------------------------------------------

data ValidInt = ValidInt {
    value :: Int,
    valid :: Bool
} deriving (Generic, NFDataX)


-- using newtype to avoid flattening of data
-- https://clash-lang.discourse.group/t/how-to-avoid-flattening-of-fields-in-record/79/5
newtype Inputs = Inputs {
    input0 :: ValidInt
} deriving (Generic, NFDataX)

data Outputs = Outputs {
    output0 :: ValidInt,
    output1 :: ValidInt,
    output2 :: ValidInt,
    output3 :: ValidInt
} deriving (Generic, NFDataX)


class Pacing a where getPacing :: a -> Bool

data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 deriving (Generic, NFDataX)
data PacingOut2 = PacingOut2 PacingIn0 deriving (Generic, NFDataX)
data PacingOut3 = PacingOut3 Bool deriving (Generic, NFDataX)

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x) = getPacing x
instance Pacing PacingOut2 where getPacing (PacingOut2 x) = getPacing x
instance Pacing PacingOut3 where getPacing (PacingOut3 x) = x
```

```haskell
data Pacings = Pacings {
    pacingIn0 :: PacingIn0,
    pacingOut0 :: PacingOut0,
    pacingOut1 :: PacingOut1,
    pacingOut2 :: PacingOut2,
    pacingOut3 :: PacingOut3
} deriving (Generic, NFDataX)

-- using newtype to avoid flattening of data
-- https://clash-lang.discourse.group/t/how-to-avoid-flattening-of-fields-in-record/79/5
newtype Slides = Slides {
    slide0 :: Bool
} deriving (Generic, NFDataX)

type Tag = Unsigned 8

data Tags = Tags {
    input0 :: Tag,
    output0 :: Tag,
    output1 :: Tag,
    output2 :: Tag,
    output3 :: Tag,
    slide0 :: Tag
} deriving (Generic, NFDataX)

type Event = (Inputs, Slides, Pacings)

nullEvent :: Event
nullEvent = (nullInputs, nullSlides, nullPacings)
nullInputs = Inputs (ValidInt 0 False)
nullSlides = Slides False
nullPacings = Pacings
                    nullPacingIn0
                    nullPacingOut0
                    nullPacingOut1
                    nullPacingOut2
                    nullPacingOut3
nullPacingIn0 = PacingIn0 False
nullPacingOut0 = PacingOut0 nullPacingIn0
nullPacingOut1 = PacingOut1 nullPacingIn0
nullPacingOut2 = PacingOut2 nullPacingIn0
nullPacingOut3 = PacingOut3 False


------------------------------------------------------------------

type QMemSize = 3

type QData = Event
```

Complete code - part 2

```
------------------------------------------------------------

type QMemSize = 3

type QData = Event
type QMem = Vec QMemSize QData
type QCursor = Int
type QPush = Bool
type QPop = Bool
type QPushValid = Bool
type QPopValid = Bool

type QState = (QMem, QCursor)
type QInput = (QPush, QPop, QData)
type QOutput = (QPushValid, QPopValid, QData)

queue :: HiddenClockResetEnable dom
    => Signal dom QInput
    -> Signal dom QOutput
queue input = output
    where
        output = bundle (pushValid, popValid, outData)
        state = bundle (buffer, cursor)
        buffer = register (repeat nullEvent :: QMem) nextBufferSignal
        cursor = register 0 nextCursorSignal
        pushValid = register False nextPushValidSignal
        popValid = register False nextPopValidSignal
        outData = register nullEvent nextOutDataSignal

        nextBufferSignal = nextBuffer
                           <$> buffer
                           <*> bundle (input, cursor)
        nextCursorSignal = nextCursor
                           <$> cursor
                           <*> bundle (input, buffer)
        nextOutDataSignal = nextOutData
                           <$> bundle (input, cursor, buffer)
        nextPushValidSignal = nextPushValid
                           <$> bundle (input, cursor, buffer)
        nextPopValidSignal = nextPopValid <$> bundle (input, cursor)

        nextBuffer :: QMem -> (QInput, QCursor) -> QMem
        nextBuffer buf ((push, pop, qData), cur) = out
            where
                out = case (push, pop) of
                    (True, True) -> qData +>> buf
                    (True, False) -> if cur == length buf
                                     then buf else qData +>> buf
                    (False, _) -> buf

        nextCursor cur ((push, pop, _), buf) = out
            where
                out = case (push, pop) of
                    (True, False) -> if cur == length buf
                                     then cur else cur + 1
                    (False, True) -> if cur == 0 then 0 else cur - 1
                    (_, _) -> cur

        nextOutData :: (QInput, QCursor, QMem) -> QData
        nextOutData ((push, pop, qData), cur, buf) = out
            where
                out = case (push, pop) of
                    (True, True) -> if cur == 0
                                    then qData else buf !! (cur - 1)
                    (False, True) -> if cur == 0
                                     then nullEvent else buf !! (cur - 1)
                    (_, _) -> nullEvent

        nextPushValid :: (QInput, QCursor, QMem) -> QPush
        nextPushValid ((push, pop, _), cur, buf) = out
            where
                out = case (push, pop) of
                    (True, True) -> True
                    (True, False) -> cur /= length buf
                    (False, _) -> False

        nextPopValid :: (QInput, QCursor) -> QPop
        nextPopValid ((push, pop, _), cur) = out
            where
                out = case (push, pop) of
                    (True, True) -> True
                    (False, True) -> cur /= 0
                    (_, False) -> False


------------------------------------------------------------------

-- Clock domain with 2 microseconds period (500 kHz)
-- It has been arbitrarily chosen for both monitor
--  and the verilog testbench simulation
createDomain vSystem{vName="TestDomain", vPeriod=2000}
-- period in nanoseconds

systemClockPeriodNs :: Int
systemClockPeriodNs = fromInteger
    (snatToInteger $ clockPeriod @TestDomain)
```

```
hlc :: HiddenClockResetEnable dom
    => Signal dom Inputs
    -> Signal dom (Bool, Event)
hlc inputs = out
    where
        out = bundle (newEvent, event)
        newEvent = hasInput0 .||. timer0Over

        event = bundle (inputs, slides, pacings)

        slides = Slides <$> s0
        pacings = Pacings <$> pIn0
                          <*> pOut0
                          <*> pOut1
                          <*> pOut2
                          <*> pOut3

        hasInput0 = ((.valid). (.input0)) <$> inputs

        pIn0 = PacingIn0 <$> hasInput0
        pOut0 = PacingOut0 <$> pIn0
        pOut1 = PacingOut1 <$> pIn0
        pOut2 = PacingOut2 <$> pIn0
        pOut3 = PacingOut3 <$> timer0Over

        s0 = timer0Over

        timer0Over = timer0 .>=. period0InNs
        timer0 = timer timer0Over
        period0InNs = 1000000

        timer :: HiddenClockResetEnable dom
            => Signal dom Bool
            -> Signal dom Int
        timer reset = register 0 (mux reset (pure deltaTime) nextTime)
            where
                nextTime = timer reset + pure deltaTime
                deltaTime = systemClockPeriodNs


------------------------------------------------------------------

-- To avoid duplidate tags in a window
-- maxTag must be at least the size of the maximum window
-- Also to avoid having to do modulo operations
-- maxTag must be at least as big as the largest offset
maxTag = 12 :: Tag
```

```
-- To avoid duplidate tags in a window
-- maxTag must be at least the size of the maximum window
-- Also to avoid having to do modulo operations
-- maxTag must be at least as big as the largest offset
maxTag = 12 :: Tag
invalidTag = maxTag + 1

getOffset :: KnownNat n => Vec n (Tag, a) -> Tag -> Tag -> a -> a
getOffset win tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = case findIndex (\(t, _) -> t == offsetTag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt

getMatchingTag :: KnownNat n => Vec n (Tag, a) -> Tag -> a -> a
getMatchingTag win tag dflt = out
    where
        out = case findIndex (\(t, _) -> t == tag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt

getOffsetFromNonVec :: (Tag, a) -> Tag -> Tag -> a -> a
getOffsetFromNonVec (winTag, winData) tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = if offsetTag == winTag then winData else dflt

getMatchingTagFromNonVec :: (Tag, a) -> Tag -> a -> a
getMatchingTagFromNonVec (tag, dta) tagToMatch dflt =
    if tag == tagToMatch then dta else dflt

getLatestValue :: KnownNat n => Vec (n + 1) (Tag, a) -> a -> a
getLatestValue win dflt =
    let (tag, dta) = last win
    in if tag == invalidTag then dflt else dta

getLatestValueFromNonVec :: (Tag, a) -> a -> a
getLatestValueFromNonVec (tag, dta) dflt =
    if tag == invalidTag then dflt else dta

earlierTag :: Tag -> Tag -> Tag
earlierTag curTag cyclesBefore =
    if curTag > cyclesBefore
    then curTag - cyclesBefore
    else curTag - cyclesBefore + maxTag
```

Complete code - part 3

```haskell
delayFor :: forall dom n a .
    (HiddenClockResetEnable dom, KnownNat n, NFDataX a)
    => SNat n
    -> a
    -> Signal dom a
    -> Signal dom a
delayFor n initVal sig = last delayedVec
    where
        delayedVec :: Vec (n + 1) (Signal dom a)
        delayedVec = iterateI (delay initVal) sig


llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        input0 = (.input0) <$> inputs

        slide0 = (.slide0) <$> slides

        pIn0 = (.pacingIn0) <$> pacings
        pOut0 = (.pacingOut0) <$> pacings
        pOut1 = (.pacingOut1) <$> pacings
        pOut2 = (.pacingOut2) <$> pacings
        pOut3 = (.pacingOut3) <$> pacings

        tOut0 = genTag (getPacing <$> pOut0)
        tIn0 = genTag (getPacing <$> pIn0)
        tOut1 = genTag (getPacing <$> pOut1)
        tOut2 = genTag (getPacing <$> pOut2)
        tSw0 = genTag (getPacing <$> pOut2)
        tOut3 = genTag (getPacing <$> pOut3)

        -- tag generation takes 1 cycle so we need to delay the input
        input0Data = delay 0 (((.value). (.input0)) <$> inputs)

        -- delayed tags to be used in different levels
        tagsDefault = Tags
                nullT
                nullT
```

```haskell
        -- tag generation takes 1 cycle so we need to delay the input
        input0Data = delay 0 (((.value). (.input0)) <$> inputs)

        -- delayed tags to be used in different levels
        tagsDefault = Tags
                nullT
                nullT
                nullT
                nullT
                nullT
                nullT
        curTags = Tags
                <$> tIn0
                <*> tOut0
                <*> tOut1
                <*> tOut2
                <*> tOut3
                <*> tSw0
        curTagsLevel0 = curTags
        curTagsLevel1 = delayFor d1 tagsDefault curTags
        curTagsLevel2 = delayFor d2 tagsDefault curTags
        curTagsLevel3 = delayFor d3 tagsDefault curTags
        curTagsLevel4 = delayFor d4 tagsDefault curTags
        curTagsLevel5 = delayFor d5 tagsDefault curTags
        nullT = invalidTag

        enOut0 = delayFor d1 nullPacingOut0 pOut0
        enIn0 = delayFor d1 nullPacingIn0 pIn0
        enOut1 = delayFor d2 nullPacingOut1 pOut1
        enOut2 = delayFor d3 nullPacingOut2 pOut2
        enSw0 = delayFor d4 nullPacingOut2 pOut2
        sld0 = delayFor d4 False slide0
        enOut3 = delayFor d5 nullPacingOut3 pOut3

        output0Aktv = delayFor d6 False (getPacing <$> pOut0)
        output1Aktv = delayFor d6 False (getPacing <$> pOut1)
        output2Aktv = delayFor d6 False (getPacing <$> pOut2)
        output3Aktv = delayFor d6 False (getPacing <$> pOut3)

        -- Evaluation of input windows: level 0
        input0Win = input0Window enIn0 tIn0 input0Data

        -- Evaluation of output 0: level 0
        out0 = outputStream0 enOut0
            tOut0
            out0Data0
        out0Data0 = getOffset
            <$> input0Win
            <*> tIn0
```

```haskell
        -- Evaluation of input windows: level 0
        input0Win = input0Window enIn0 tIn0 input0Data

        -- Evaluation of output 0: level 0
        out0 = outputStream0 enOut0
            tOut0
            out0Data0
        out0Data0 = getOffset
            <$> input0Win
            <*> tIn0
            <*> (pure 1)
            <*> out0Data0Dflt
        out0Data0Dflt = pure (0)

        -- Evaluation of output 1: level 1
        out1 = outputStream1 enOut1
            ((.output1) <$> curTagsLevel1)
            out1Data0
            out1Data1
        out1Data0 = getMatchingTag
            <$> out0
            <*> ((.output0) <$> curTagsLevel1)
            <*> (pure (0))
        out1Data1 = getOffset
            <$> out2
            <*> ((.output2) <$> curTagsLevel1)
            <*> (pure 1)
            <*> out1Data1Dflt
        out1Data1Dflt = out1Data1DfltData0 + out1Data1DfltData1
        out1Data1DfltData0 =  getMatchingTag <$> input0Win <*> ((.input0) <$> curTagsLevel1) <*> (pure (0))
        out1Data1DfltData1 =  getMatchingTag <$> out0 <*> ((.output0) <$> curTagsLevel1) <*> (pure (0))

        -- Evaluation of output 2: level 2
        out2 = outputStream2 enOut2
            ((.output2) <$> curTagsLevel2)
            out2Data0
        out2Data0 = getMatchingTag
            <$> out1
            <*> ((.output1) <$> curTagsLevel2)
            <*> (pure (0))

        -- Evaluation of output 3: level 4
        out3 = outputStream3 enOut3
            ((.output3) <$> curTagsLevel4)
            out3Data0
            out3Data1
        out3Data0 = getLatestValue
            <$> input0Win
            <*> out3Data0Dflt
```

Complete code - part 4

```haskell
-- Evaluation of output 3: level 4
out3 = outputStream3 enOut3
    ((.output3) <$> curTagsLevel4)
    out3Data0
    out3Data1
out3Data0 = getLatestValue
    <$> input0Win
    <*> out3Data0Dflt
out3Data0Dflt = pure (0)
(_, out3Data1) = unbundle sw0

-- Evaluation of sliding window 0: level 3
sw0 = slidingWindow0 enSw0 sld0
    ((.slide0) <$> curTagsLevel3) sw0Data
sw0Data = getMatchingTag
    <$> out2
    <*> ((.output2) <$> curTagsLevel3)
    <*> (pure 0)

-- Outputing all results: level 5
output0 = ValidInt <$> output0Data <*> output0Aktv
output0Data = getMatchingTag
    <$> out0
    <*> ((.output0)
    <$> curTagsLevel5)
    <*> (pure 0)
output1 = ValidInt <$> output1Data <*> output1Aktv
output1Data = getMatchingTag
    <$> out1
    <*> ((.output1)
    <$> curTagsLevel5)
    <*> (pure 0)
output2 = ValidInt <$> output2Data <*> output2Aktv
output2Data = getMatchingTag
    <$> out2
    <*> ((.output2)
    <$> curTagsLevel5)
    <*> (pure 0)
output3 = ValidInt <$> output3Data <*> output3Aktv
(_, output3Data) = unbundle out3

outputs = Outputs
    <$> output0
    <*> output1
    <*> output2
    <*> output3
```

```haskell
    genTag :: HiddenClockResetEnable dom
        => Signal dom Bool
        -> Signal dom Tag
    genTag en = t
        where
            t = register 1 (mux en next_t t)
            next_t = mux (t .==. (pure maxTag)) (pure 1) (t + 1)


pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 1 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                (mux (toWait .>. pure 0) (toWait - 1) toWait)


input0Window :: HiddenClockResetEnable dom
    => Signal dom PacingIn0
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 2 (Tag, Int))
input0Window en tag val = result
    where result = register (repeat (invalidTag, 0))
                    (mux (getPacing <$> en)
                        ((<<+) <$> result <*> (bundle (tag, val)))
                        result)


outputStream0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut0
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 3 (Tag, Int))
outputStream0 en tag in0_ = result
    where
        result = register (repeat (invalidTag, 0))
                (mux (getPacing <$> en) next result)
        next = (<<+) <$> result <*> nextValWithTag
        nextValWithTag = bundle (tag, nextVal)
        nextVal = in0_
```

```haskell
outputStream1 :: HiddenClockResetEnable dom
    => Signal dom PacingOut1
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom (Vec 2 (Tag, Int))
outputStream1 en tag out0_0 out2_1 = result
    where
        result = register (repeat (invalidTag, 0))
                (mux (getPacing <$> en) next result)
        next = (<<+) <$> result <*> nextValWithTag
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (out0_0 + out2_1)


outputStream2 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 2 (Tag, Int))
outputStream2 en tag out1_0 = result
    where
        result = register (repeat (invalidTag, 0))
                (mux (getPacing <$> en) next result)
        next = (<<+) <$> result <*> nextValWithTag
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (out1_0 + (1))


outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream3 en tag in0_0 sw0 = result
    where
        result = register (invalidTag, 0)
                (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + (merge0 <$> sw0))
        merge0 :: Vec 11 Int -> Int
        merge0 win = fold windowFunc0 (tail win)


windowFunc0 :: Int -> Int -> Int
windowFunc0 acc item = acc + item
```

Complete code - part 5

```
slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 11 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 11 Int

        nextWindow :: Vec 11 Int
            -> Bool
            -> Bool
            -> Int
            -> Vec 11 Int
        nextWindow win toSlide newData dta = out
            where
                out = case (toSlide, newData) of
                    (False, False) -> win
                    (False, True) -> lastBucketUpdated
                    (True, False) -> 0 +>> win
                    (True, True) -> 0 +>> lastBucketUpdated
                lastBucketUpdated =
                    replace 0 (windowFunc0 (head win) dta) win
```

```
monitor :: HiddenClockResetEnable dom
    => Signal dom Inputs
    -> Signal dom Outputs
monitor inputs = outputs
    where
        (newEvent, event) = unbundle (hlc inputs)

        (qPushValid, qPopValid, qPopData) =
            unbundle (queue (bundle (qPush, qPop, qInptData)))
        qPush = newEvent
        qPop = toPop
        qInptData = event

        (toPop, outputs) = unbundle (llc (bundle (qPopValid, qPopData)))
```

```
topEntity :: Clock TestDomain
    -> Reset TestDomain
    -> Enable TestDomain
    -> Signal TestDomain Inputs
    -> Signal TestDomain Outputs
topEntity clk rst en inputs =
    exposeClockResetEnable (monitor inputs) clk rst en
```

Synthesised wires

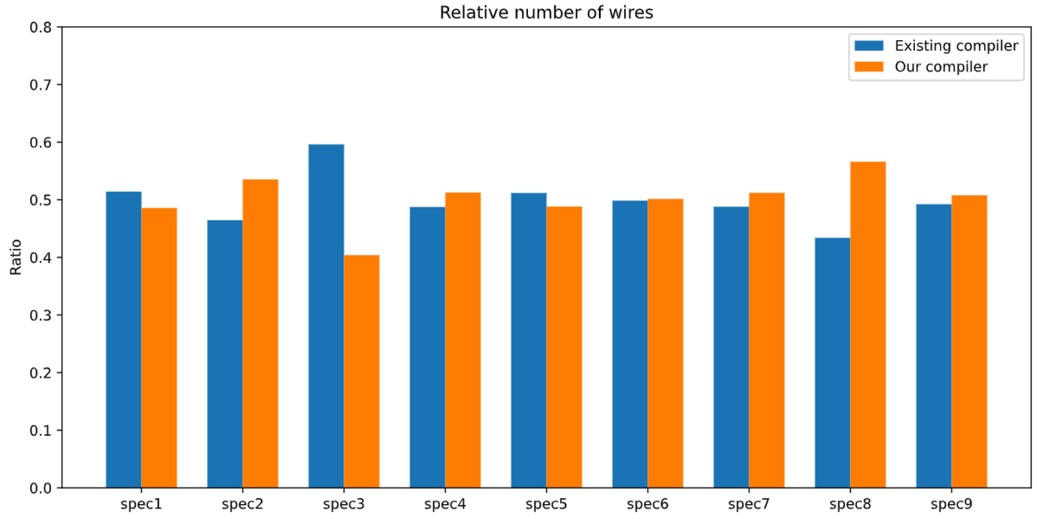


Figure 49: Relative number of synthesized wires

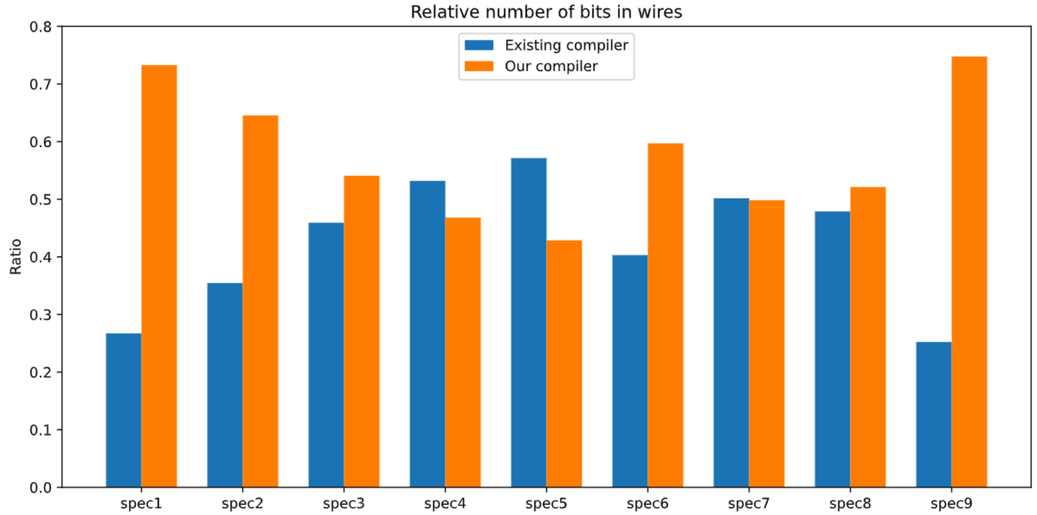


Figure 50: Relative number of total bits in synthesized wires