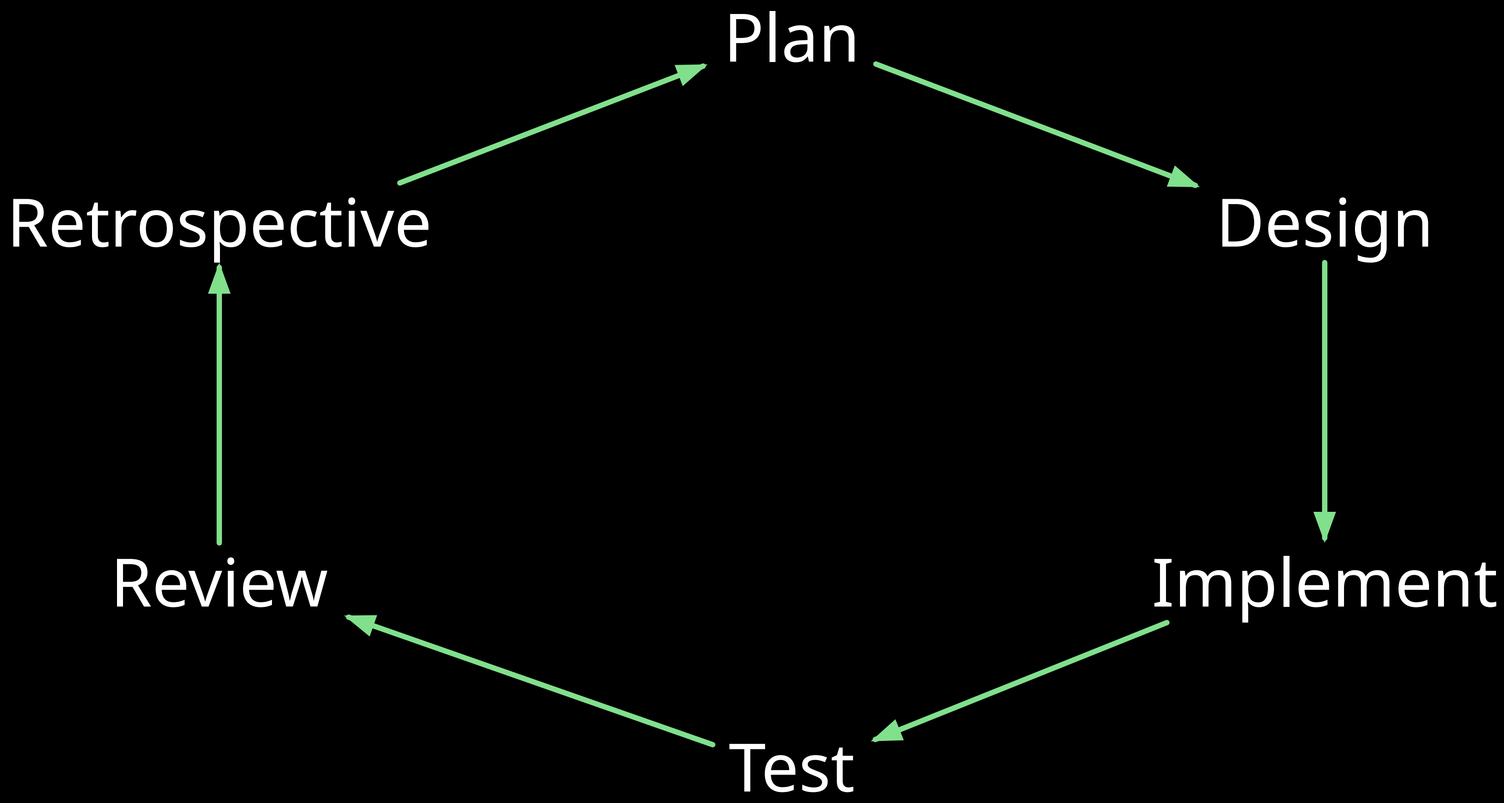
Spade, and Using Your Language in Agile Design

Frans Skarman

Hochschule München



Plan

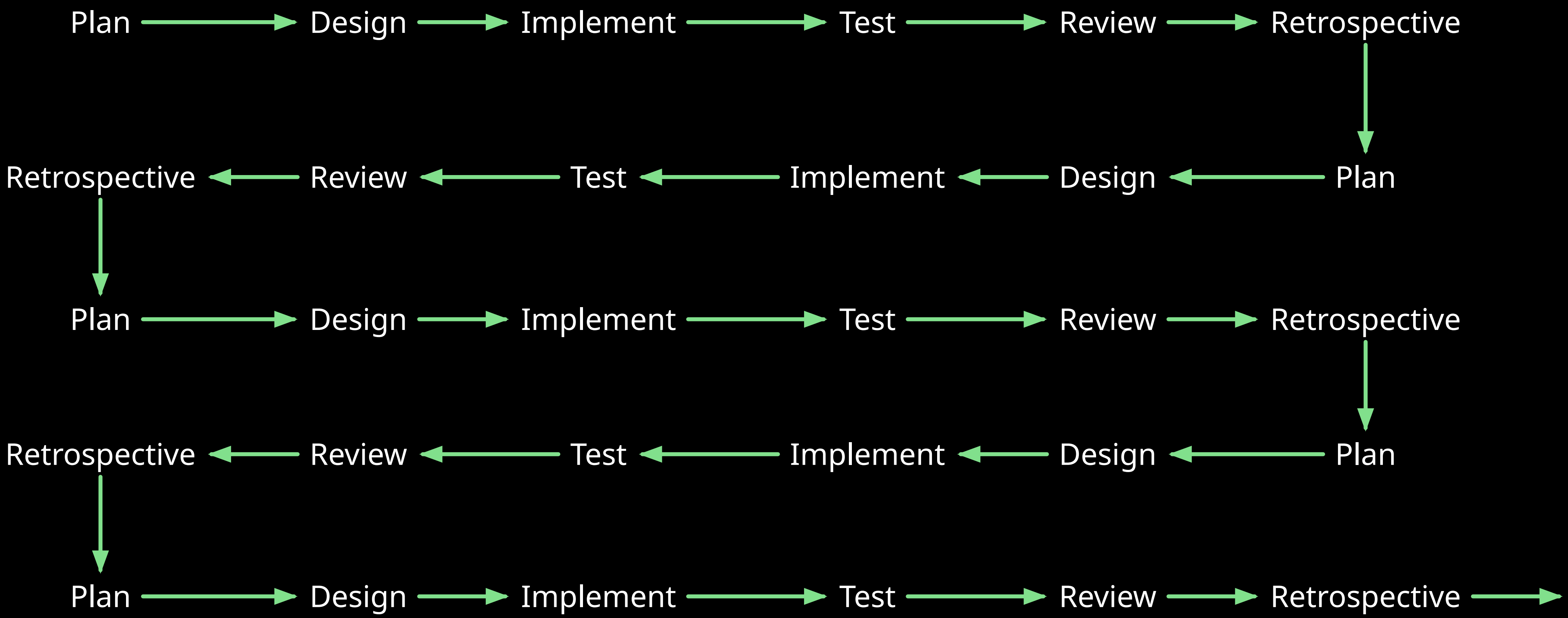
Design

Implement

Test

Review

Retrospective



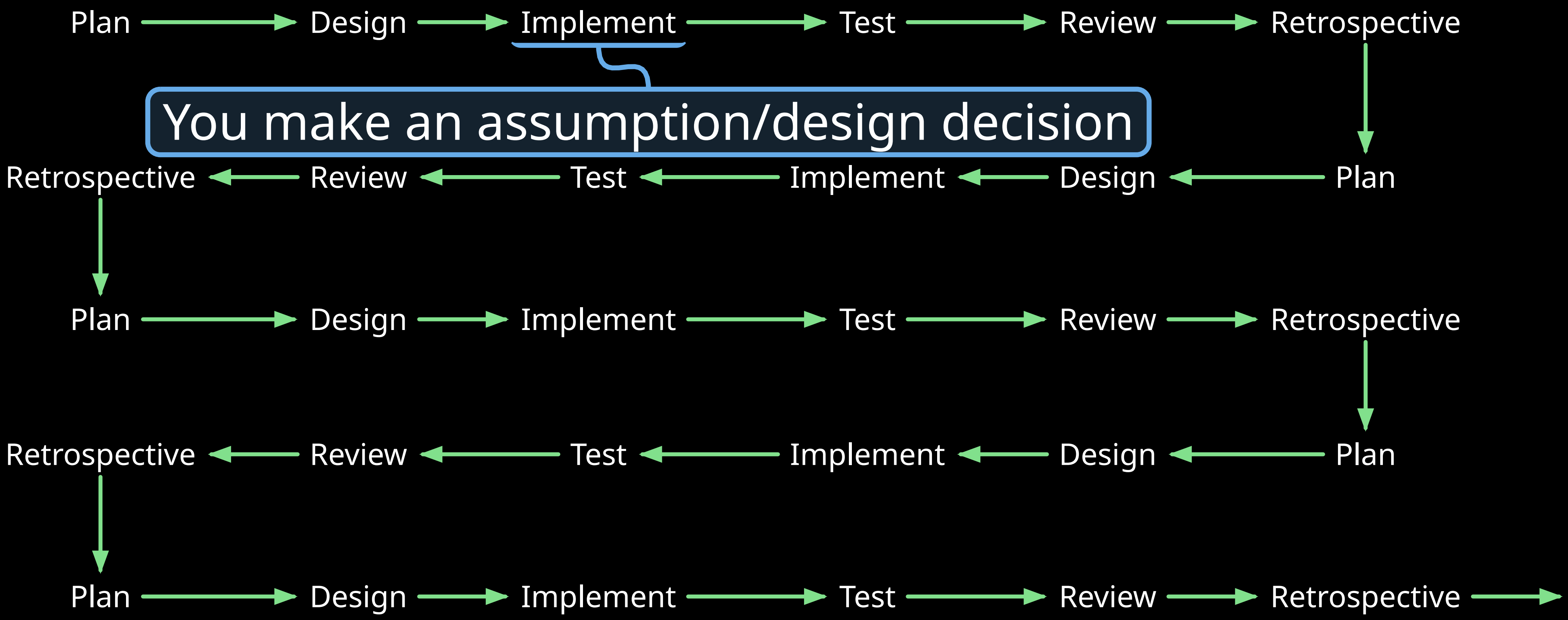
Plan → Design → Implement → Test → Review → Retrospective

Retrospective ← Review ← Test ← Implement ← Design ← Plan

Plan → Design → Implement → Test → Review → Retrospective

Retrospective ← Review ← Test ← Implement ← Design ← Plan

Plan → Design → Implement → Test → Review → Retrospective →



Plan → Design → Implement → Test → Review → Retrospective

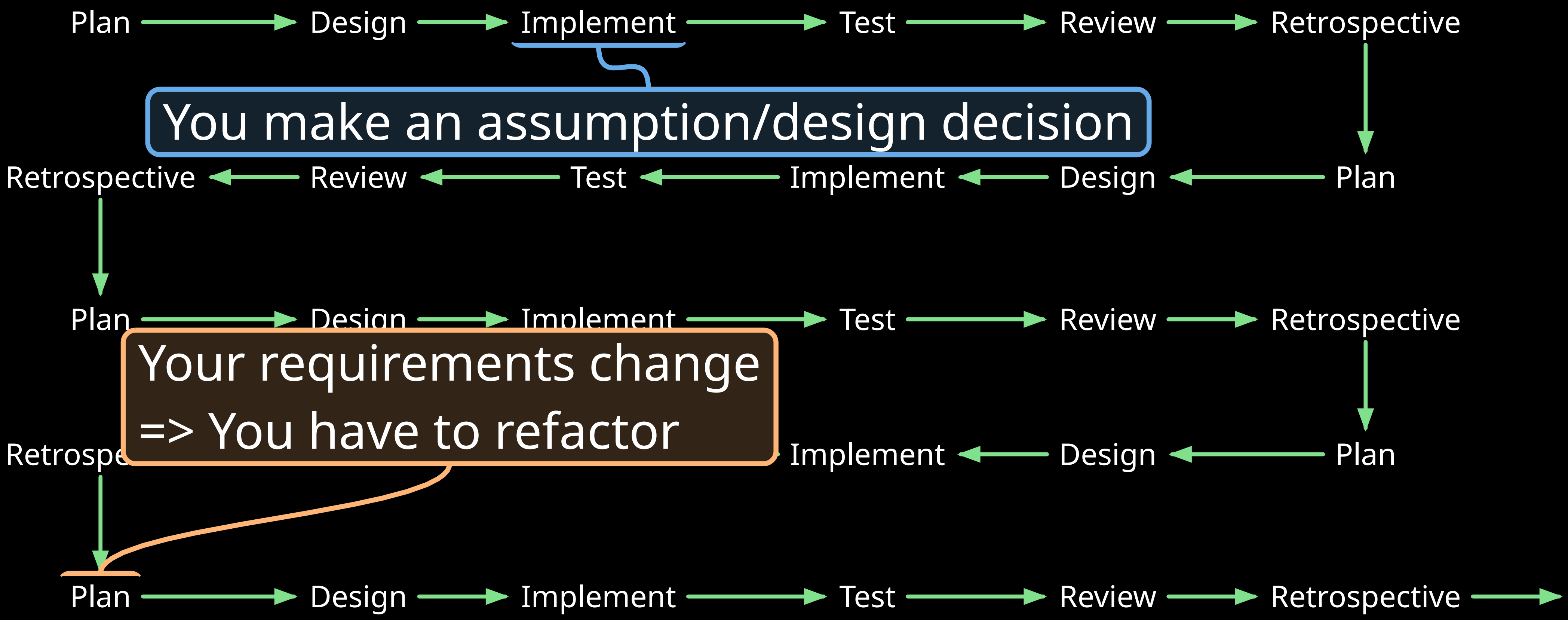
You make an assumption/design decision

Retrospective ← Review ← Test ← Implement ← Design ← Plan

Plan → Design → Implement → Test → Review → Retrospective

Retrospective ← Review ← Test ← Implement ← Design ← Plan

Plan → Design → Implement → Test → Review → Retrospective →



Plan → Design → Implement → Test → Review → Retrospective

You make an assumption/design decision

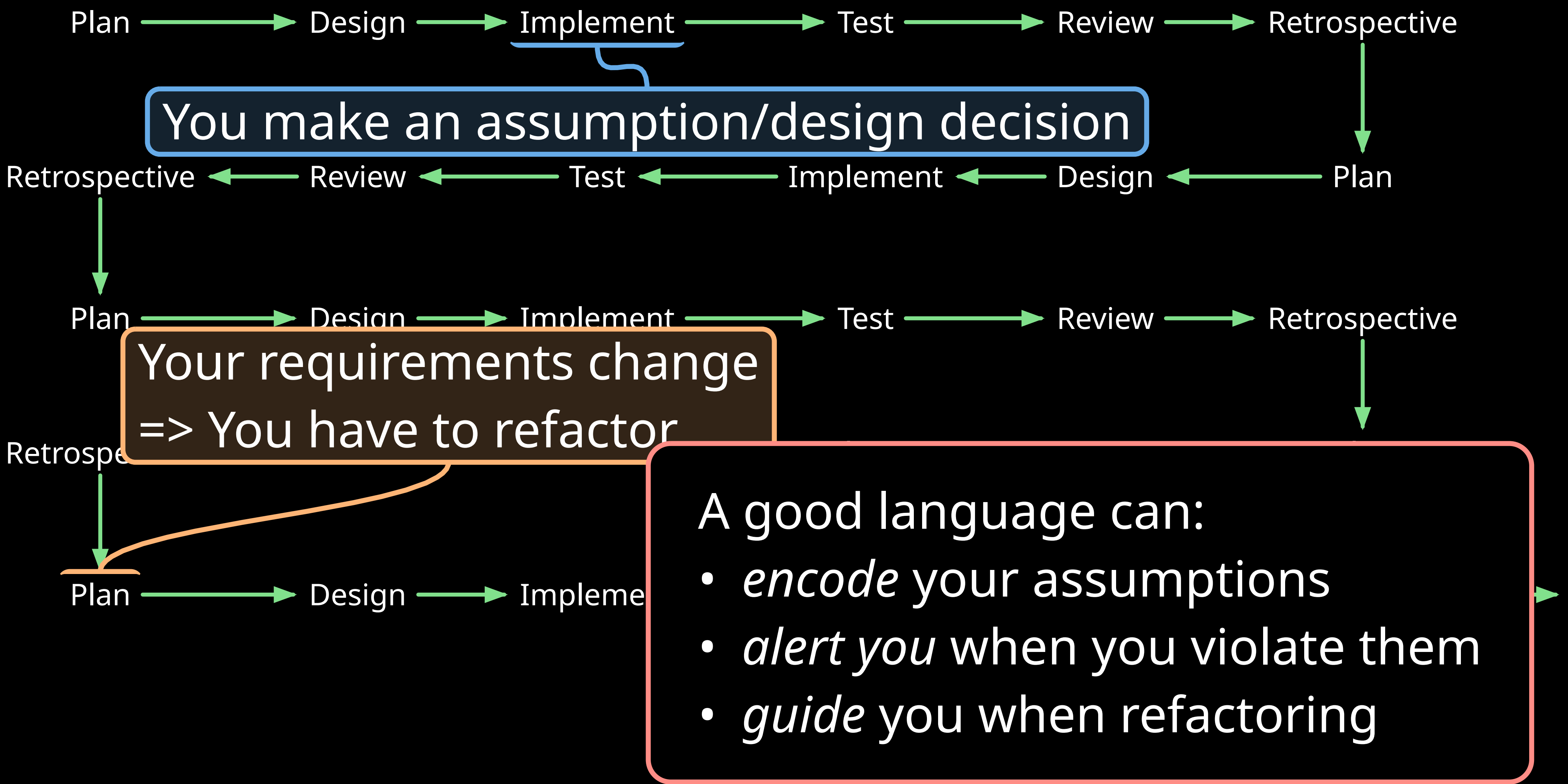
Retrospective ← Review ← Test ← Implement ← Design ← Plan

Plan → Design → Implement → Test → Review → Retrospective

Your requirements change
=> You have to refactor

Retrospe Implement ← Design ← Plan

Plan → Design → Implement → Test → Review → Retrospective →

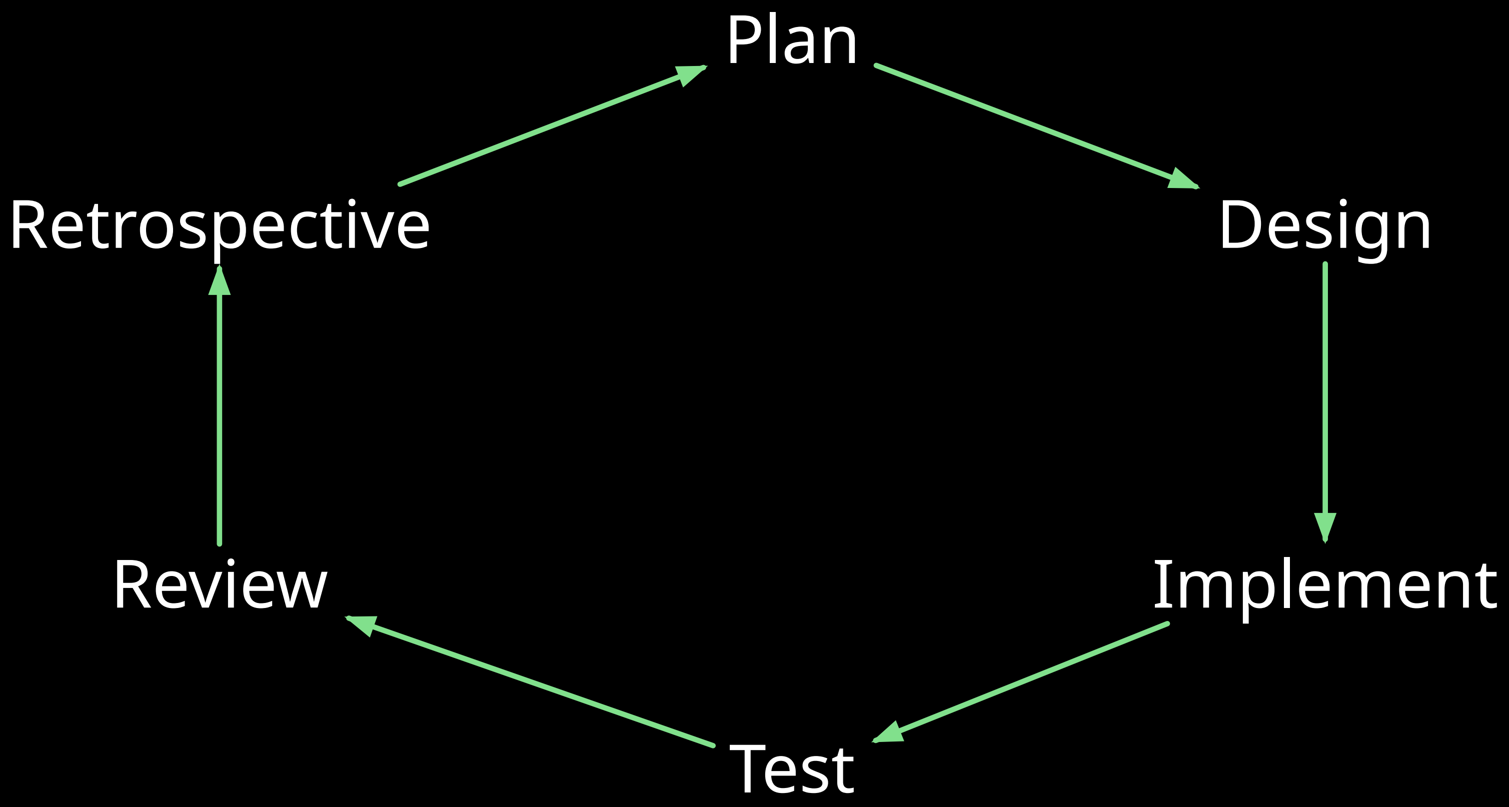


You make an assumption/design decision

Your requirements change
=> You have to refactor

A good language can:

- *encode* your assumptions
- *alert you* when you violate them
- *guide* you when refactoring



Plan

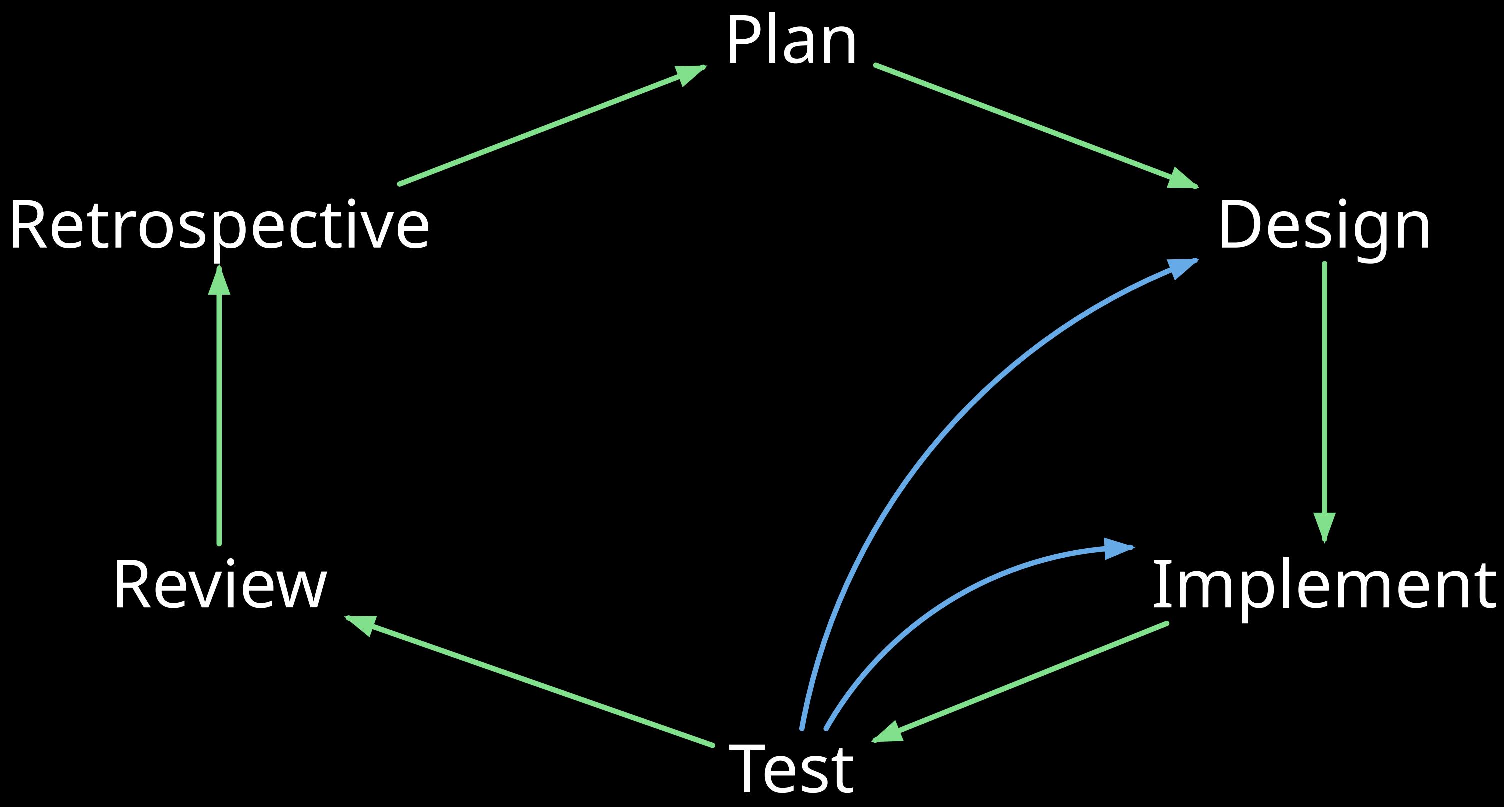
Design

Implement

Test

Review

Retrospective



Plan

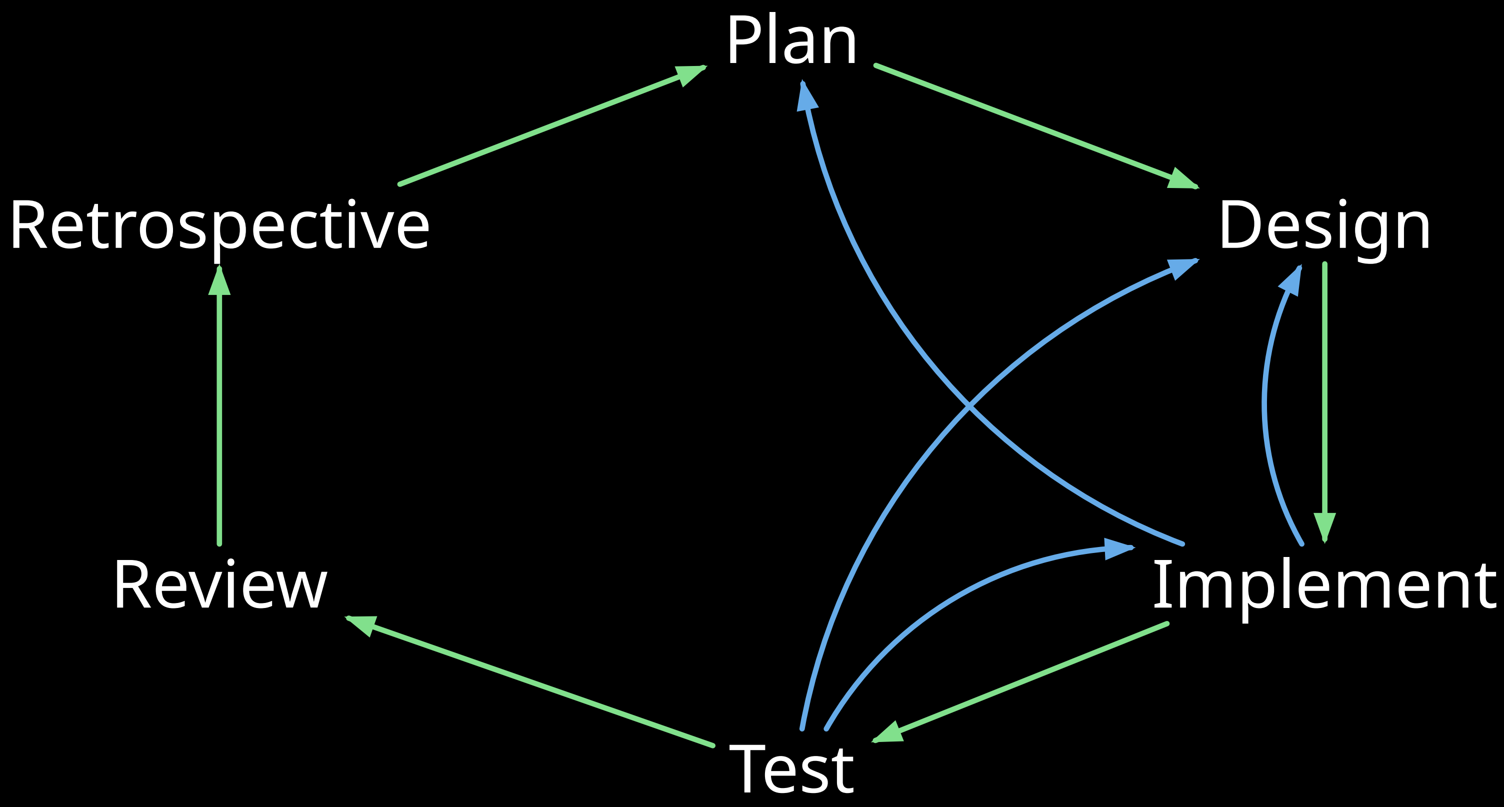
Retrospective

Design

Review

Implement

Test



Plan

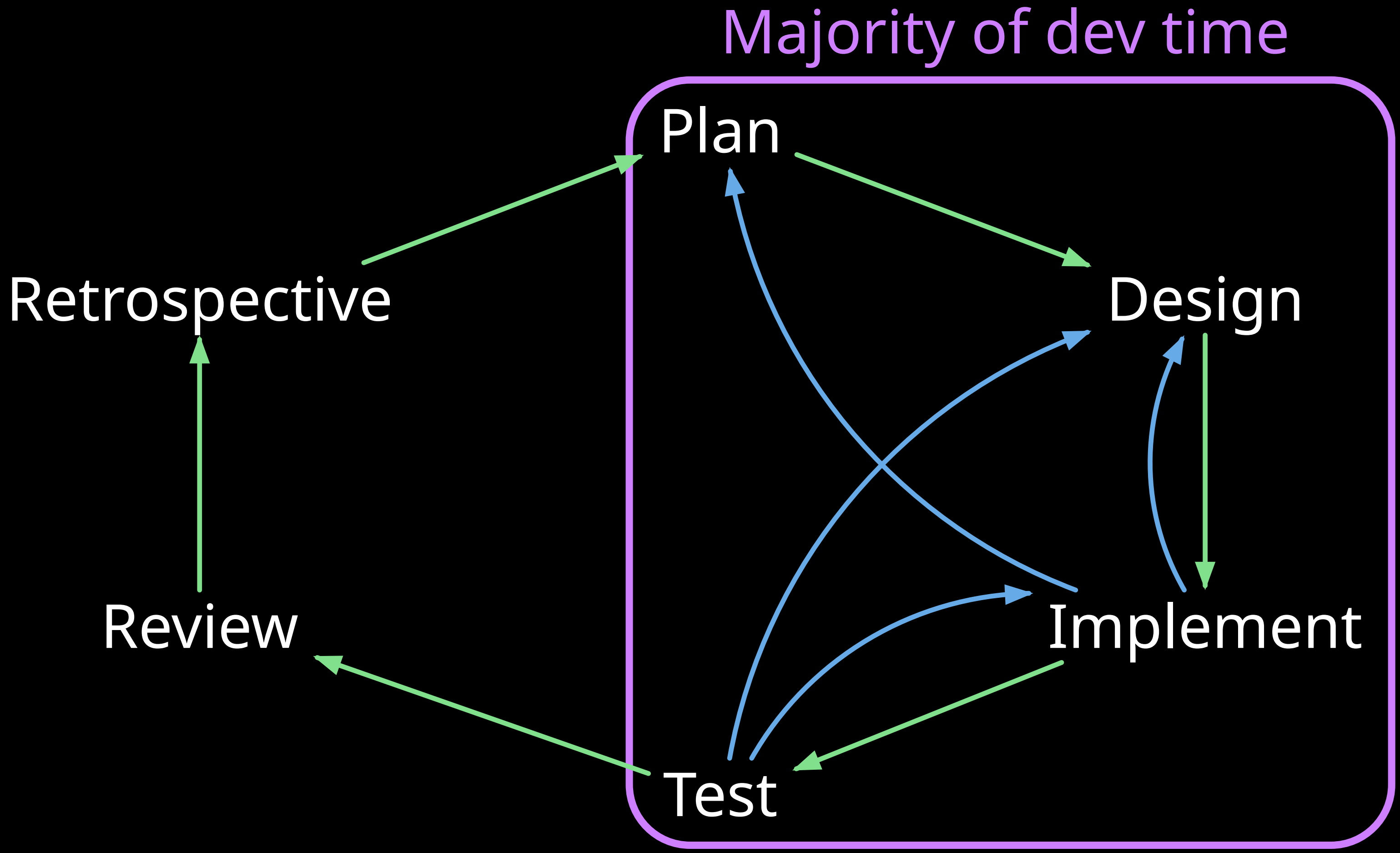
Retrospective

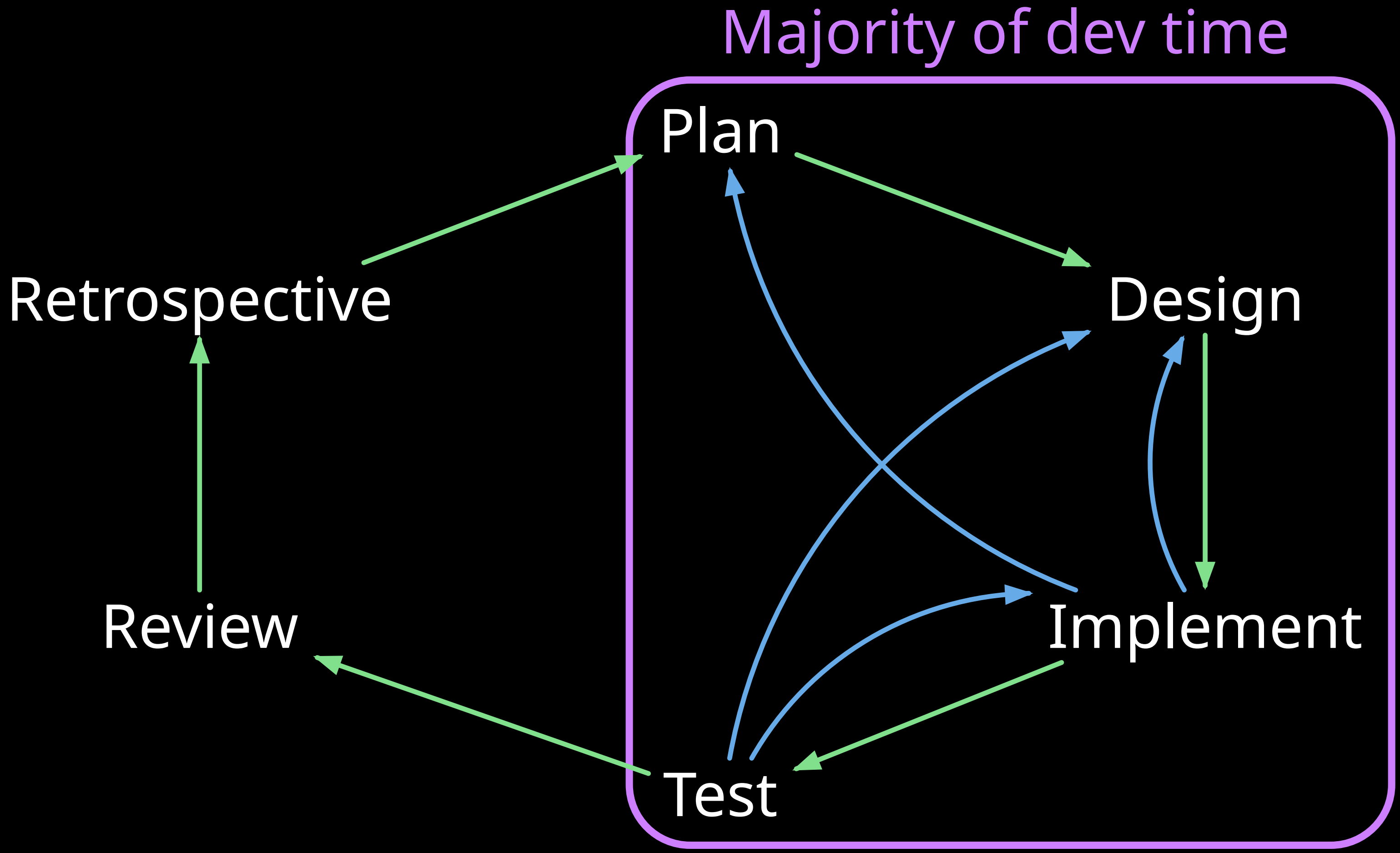
Design

Review

Implement

Test


Majority of dev time
Plan
Design
Implement
Test
Review
Retrospective



Majority of dev time

Plan

Design

Implement

Test

Review

Retrospective

- 0.1 second user feels that the system is **reacting instantaneously**

- Usability Engineering — Jakob Nielsen

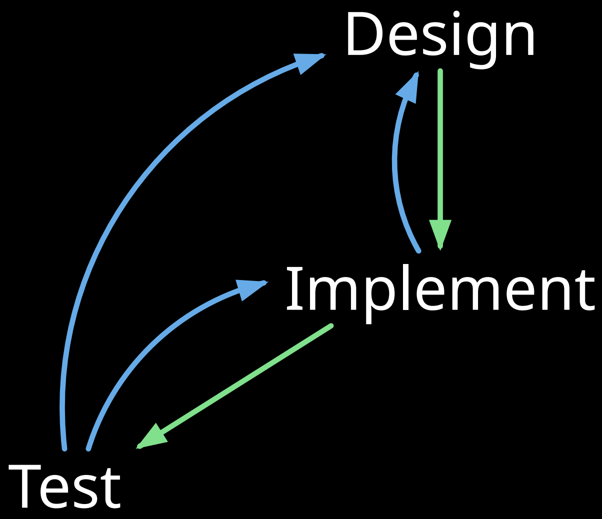
- 0.1 second user feels that the system is **reacting instantaneously**
- 1.0 second flow of thought **stays uninterrupted**

- Usability Engineering — Jakob Nielsen

- 0.1 second user feels that the system is **reacting instantaneously**
- 1.0 second flow of thought **stays uninterrupted**
- 10 seconds is about the limit for **keeping the user’s attention**

- Usability Engineering — Jakob Nielsen

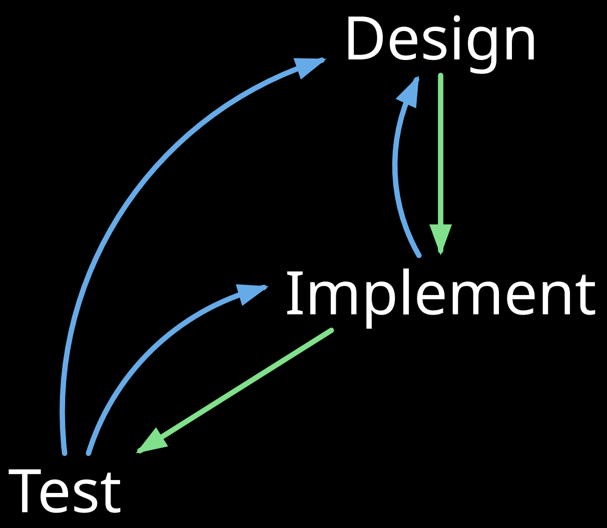
- 0.1 second user feels that the system is **reacting instantaneously**
- 1.0 second flow of thought **stays uninterrupted**
- 10 seconds is about the limit for **keeping the user's attention**



- Usability Engineering — Jakob Nielsen

- 0.1 second user feels that the system is **reacting instantaneously**
- 1.0 second flow of thought **stays uninterrupted**
- 10 seconds is about the limit for **keeping the user's attention**

How long is your simulation runtime?

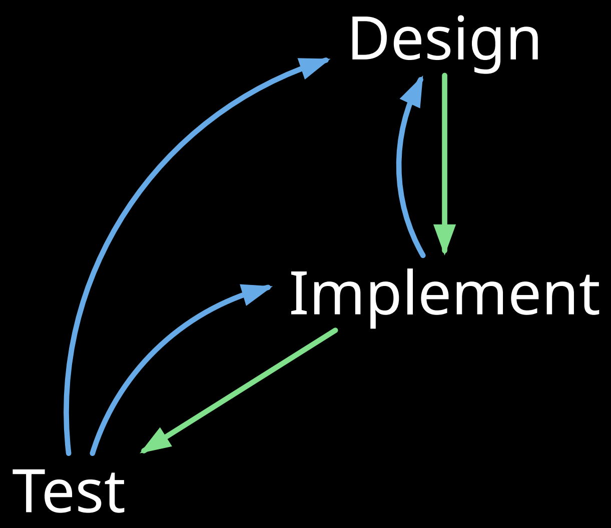


- Usability Engineering — Jakob Nielsen

- 0.1 second user feels that the system is **reacting instantaneously**
- 1.0 second flow of thought **stays uninterrupted**
- 10 seconds is about the limit for **keeping the user's attention**

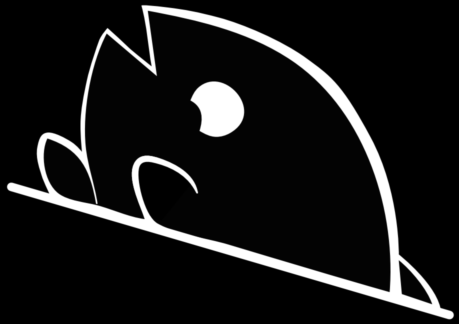
How long is your simulation runtime?

What about synthesis??

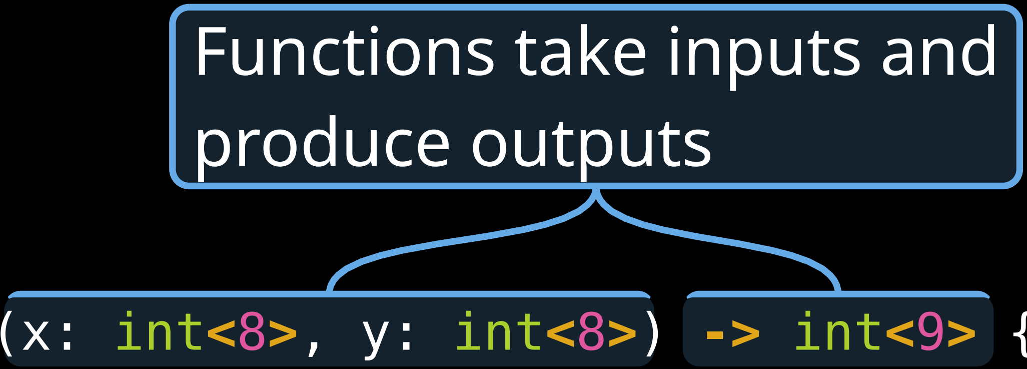


- Usability Engineering — Jakob Nielsen

Spade



```
fn adder(x: int<8>, y: int<8>) -> int<9> {
  x + y
}
```



Functions take inputs and produce outputs

```
fn adder(x: int<8>, y: int<8>) -> int<9> {
  x + y
}
```

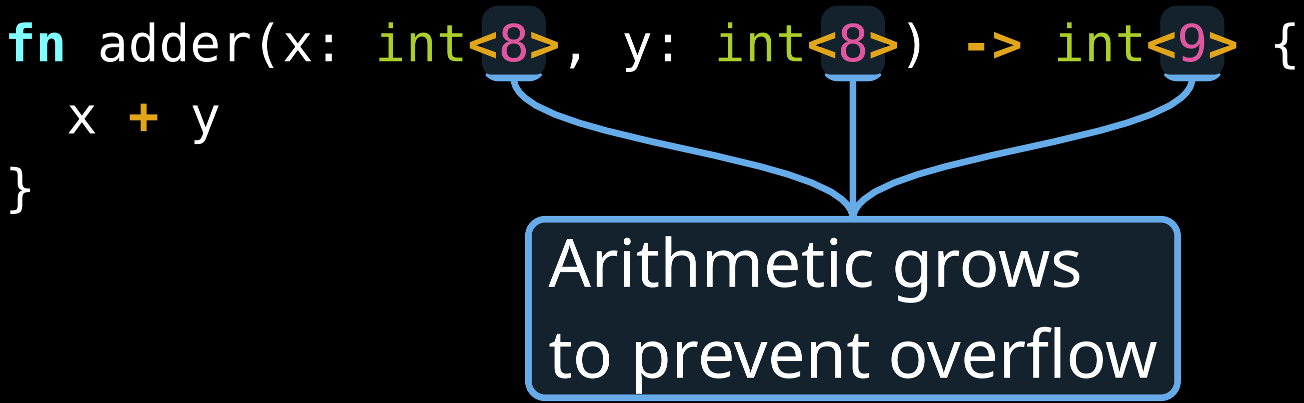
```
fn adder(x: int<8>, y: int<8>) -> int<9> {
  x + y
}
```

Standard math operators

+, -, *, &&, etc.

```
fn adder(x: int<8>, y: int<8>) -> int<9> {
  x + y
}
```

Last value in a block is returned



```
fn adder(x: int<8>, y: int<8>) -> int<9> {
  x + y
}
```

Arithmetic grows
to prevent overflow

```
fn adder(x: int<8>, y: int<8>) -> int<8>) {
  trunc(x + y)
}
```

```
fn adder(x: int<8>, y: int<8>) -> int<8>) {
  trunc(x + y)
}
```

`trunc` truncates back down

```
fn multiply_add(x: int<8>, y: int<8>, z: int<16>) -> int<17> {
  let sum = x * y
  sum + z
}
```

```
fn multiply_add(x: int<8>, y: int<8>, z: int<16>) -> int<17> {
  let sum = x * y
  sum + z
}
```

`let` defines new variables
(like `val` in Scala)

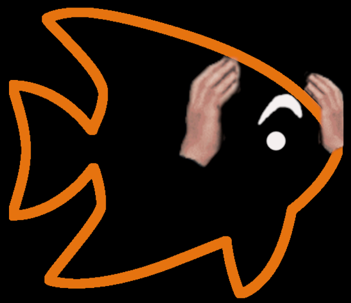
```
fn multiply_add(x: int<8>, y: int<8>, z: int<16>) -> int<17> {
  let sum = x * y
  sum + z
}
```

Typeinference infers types from context

`int<16>` in this case

```
fn multiply_add(x: int<8>, y: int<8>, z: int<16>) -> int<17> {
  let sum = x * y
  sum = sum + 1
  sum + z
}
```

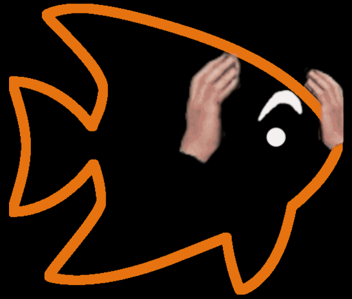
❗ Variables are immutable:
They can only be assigned once



```
fn select(select_a: bool, a: int<8>, b: int<8>) -> int<8> {
    let result = b;
    if select_a {
        result = a;
    }
    result
}
```

```
fn select(select_a: bool, a: int<8>, b: int<8>) -> int<8> {
    let result = b;
    if select_a {
        result = a;
    }
    result
}
```

❗ Mutation

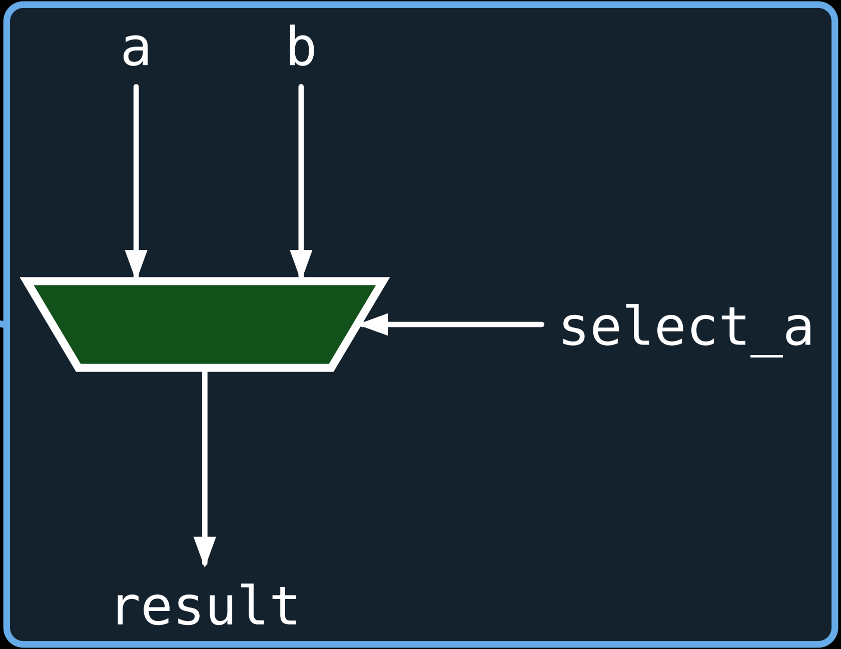


```
fn select(select_a: bool, a: int<8>, b: int<8>) -> int<8> {

    let result = if select_a {
        a
    } else {
        b
    };

    result
}
```

```
fn select(select_a: bool, a: int<8>, b: int<8>) -> int<8> {

    let result = if select_a {
        a
    } else {
        b
    };

    result
}
```

If **expressions** *return* values

```
fn select(select_a: bool, a: int<8>, b: int<8>) -> int<8> {
    let result = if select_a {
        a
    } else {
        b
    };

    result
}
```



Sequential Logic

```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on = !on;
  on
}
```

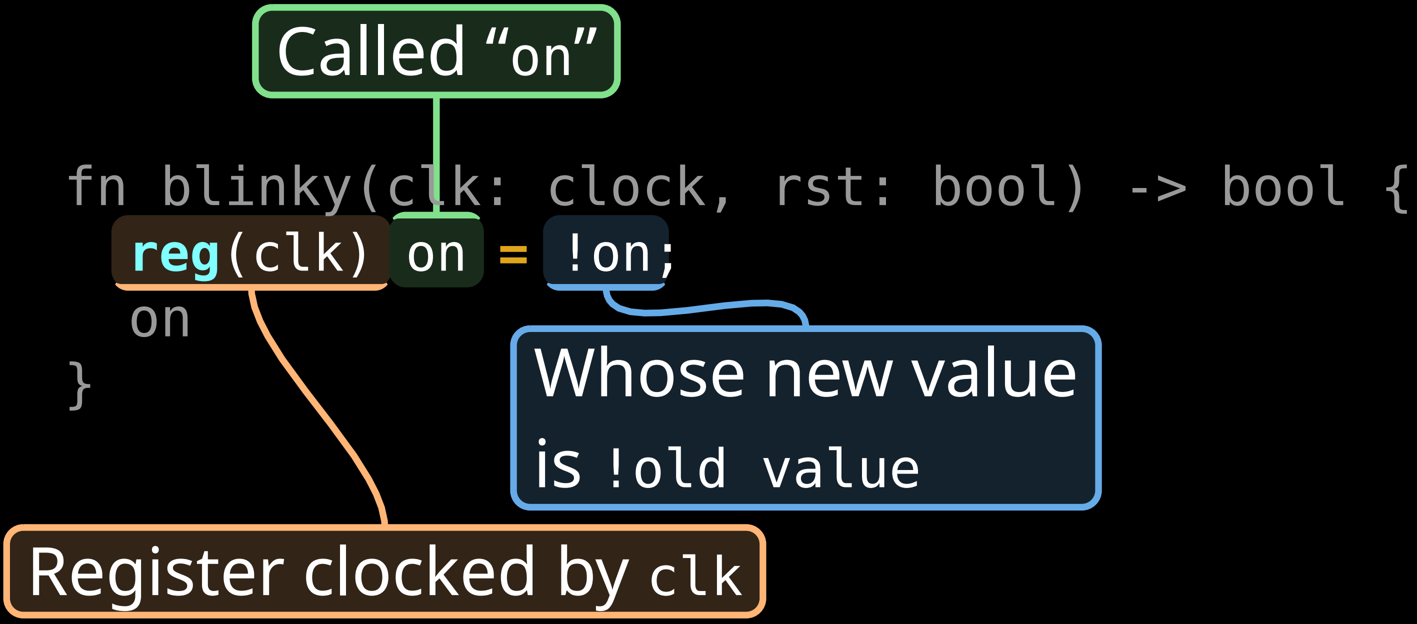
```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on = !on;
  on
}
```

Register clocked by `clk`

Called “on”

```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on = !on;
  on
}
```

Register clocked by `clk`



Called “on”

```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on = !on;
  on
}
```

Whose new value is `!old value`

Register clocked by `clk`

```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on reset(rst: false) = !on;
  on
}
```

Reset to false when rst is true

```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on reset(rst: false) = !on;
  on
}
```

```
error: register declared in function
   ┌─ src/main.spade:2:3
1  │ fn blinky(clk: clock, rst: bool) {
   │ -- this is a function
2  │   reg(clk) on = !on;
   │   ^^^ register not allowed here
   │
   = note: functions can only contain combinatorial logic
   = consider making the function an entity
1  │ entity blinky(clk: clock, rst: bool) {
   │ ~~~~~~
```

```
fn blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on reset(rst: false) = !on;
  on
}
```

```
error: register declared in function
   ┌─ src/main.spade:2:3
1  │ fn blinky(clk: clock, rst: bool) {
   │ -- this is a function
2  │   reg(clk) on = !on;
   │   ^^^ register not allowed here
   │
   = note: functions can only contain combinatorial logic
   = consider making the function an entity
1  │ entity blinky(clk: clock, rst: bool) {
   │ ~~~~~~
```

```
entity blinky(clk: clock, rst: bool) -> bool {
  reg(clk) on reset(rst: false) = !on;
  on
}
```

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            counter + 1
        };

    counter > 10_000 / 2
}
```

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0

            counter + 1
        };

    counter > 10_000 / 2
}
```

Register called counter

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0

            counter + 1
        };

    counter > 10_000 / 2
}
```

Register called counter

Reset to 0

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            counter + 1
        };

    counter > 10_000 / 2
}
```

Next value = 0 if at max, otherwise reset to 0

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            counter + 1
        };

    counter > 10_000 / 2
}
```

Output is `true` if the counter is in its upper half

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            counter + 1
        };

    counter > 10_000 / 2
}
```

```
error: Type of expression is not fully known
  ┌─ src/main.spade:7:8
  │
7 │     if count == duration {
  │        ^^^^^ The type of this expression is not fully known
  │
  = note: Found incomplete type: Number<_>
```

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter: uint<15> reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            counter + 1
        };

    counter > 10_000 / 2
}
```

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            counter + 1
        };

    counter > 10_0
}
```

```
error: Expected type uint<15>, got Number<16>
  6 |     reg(clk) count: uint<15> reset(rst: 0) =
    |                           -- Type 15 inferred here
  7 | ╭       if count == duration {
  8 | │         0
  9 | │       } else {
 10 | │         count + 1
    | │         --------- Type 16 inferred here
 11 | │       };
    | ╰───────^ Expected uint<15>
    = note: Expected: 15 in: uint<15>
                 Got: 16 in: Number<16>
```

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            trunc(counter + 1)
        };

    counter > 10_000 / 2
}
```

```
entity blinky(clk: clock, rst: bool) -> bool {
    reg(clk) counter reset(rst: 0) =
        if counter == 10_000 {
            0
        } else {
            trunc(counter + 1)
        };

    counter > 10_000 / 2
}
```

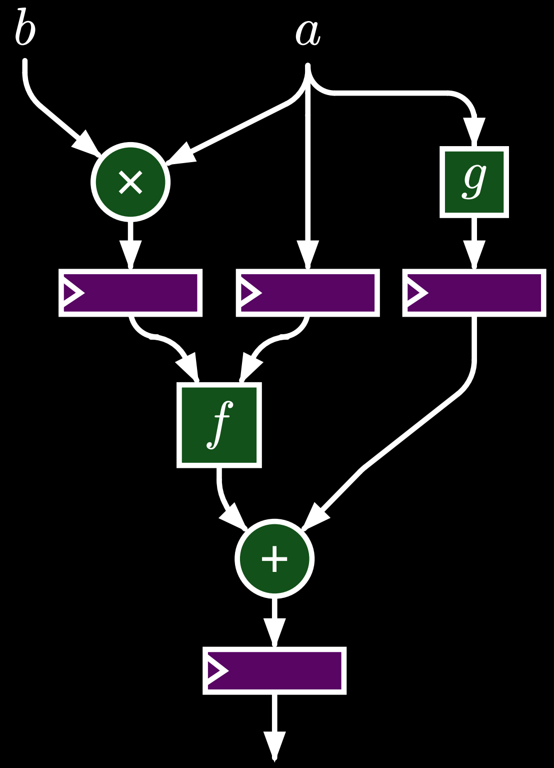
Let’s try it on

https://play.spade-lang.org

Pipelines and Timing

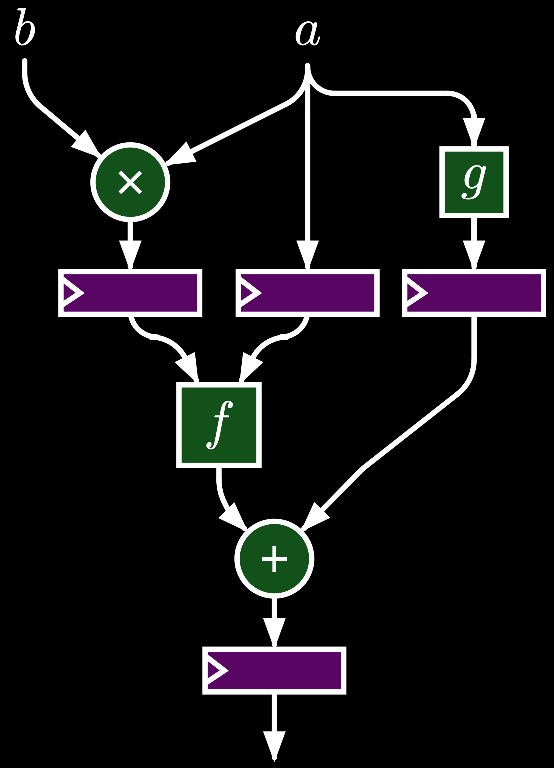
Pipelines

```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = g(a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



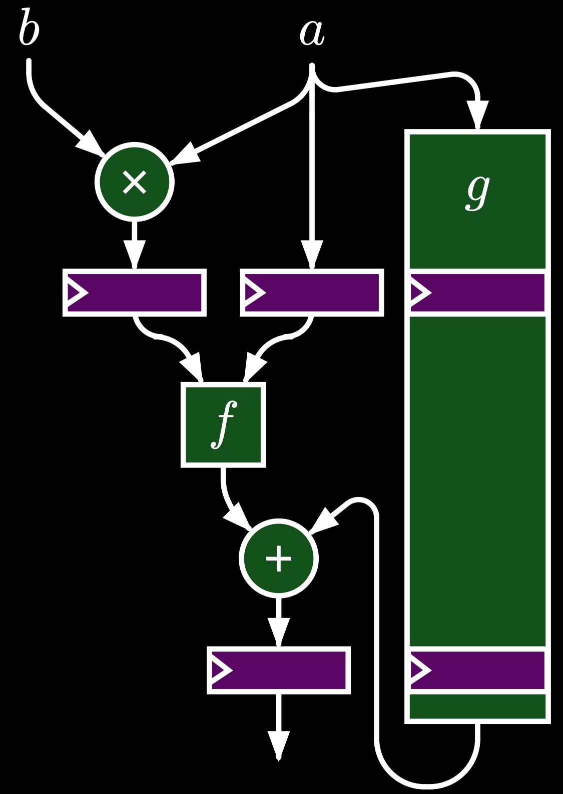
Pipelines

```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = g(a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



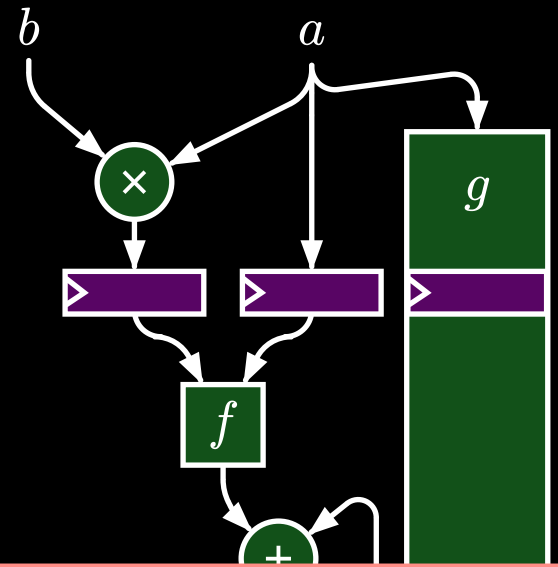
Pipelines

```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = g(a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



Pipelines

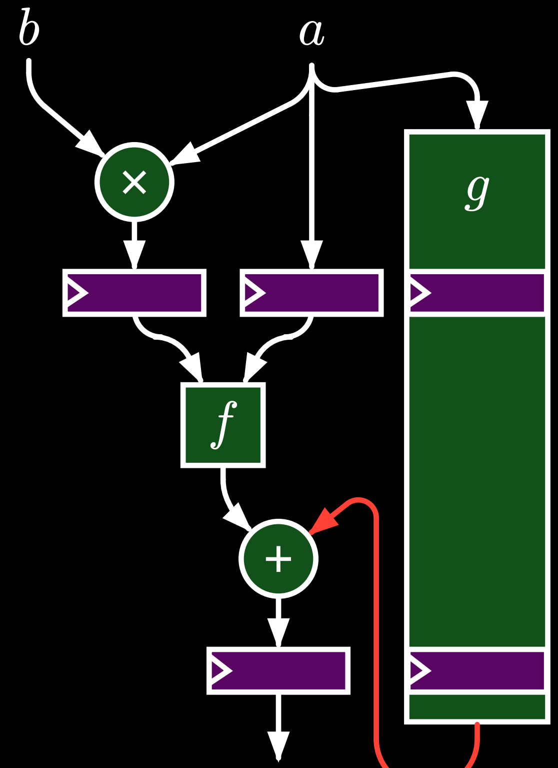
```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = g(a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



```
error: Expected `inst` and pipeline depth for pipeline instantiation
   ┌─ src/main.spade:3:13
   │
 3 │     let x = g(a);
   │             ^
   │             Expected pipeline instantiation
   │             Because g is a pipeline
   ·
   = Consider instantiating the pipeline with a depth
 3 │     let x = inst(/*depth*/) g(a);
                     +++++++++++
```

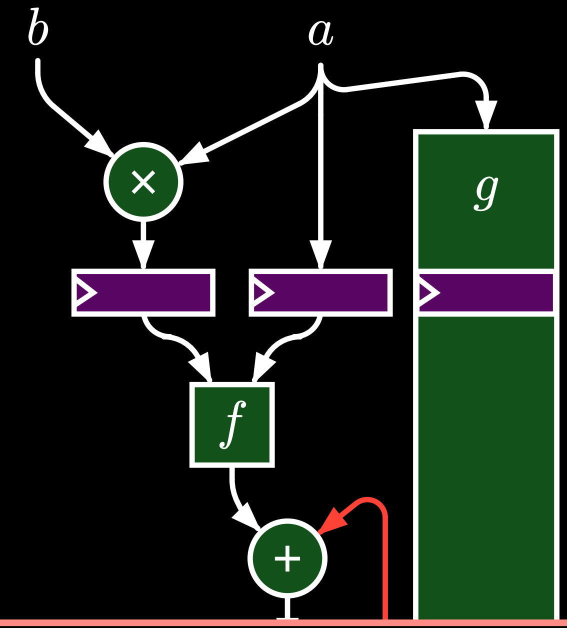
Pipelines

```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = inst(2) g(clk, a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



Pipelines

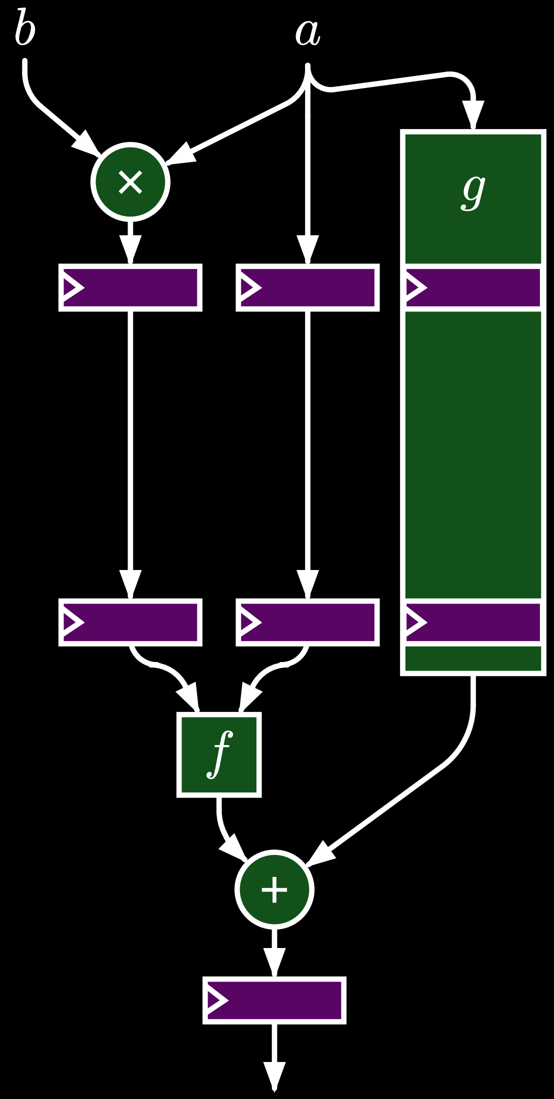
```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = inst(2) g(clk, a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



```
error: Use of x before it is ready
  ┌─ src/main.spade:7:15
3 │     let x = inst(2) g(a);
  │         - x is defined here at stage 0 with a latency of 2
  ·
7 │     let sum = x + f(a, product);
  │               ^
  │               x is unavailable for another 1 stage
  = help: Consider adding more reg; statements between
          the definition and use of x
```

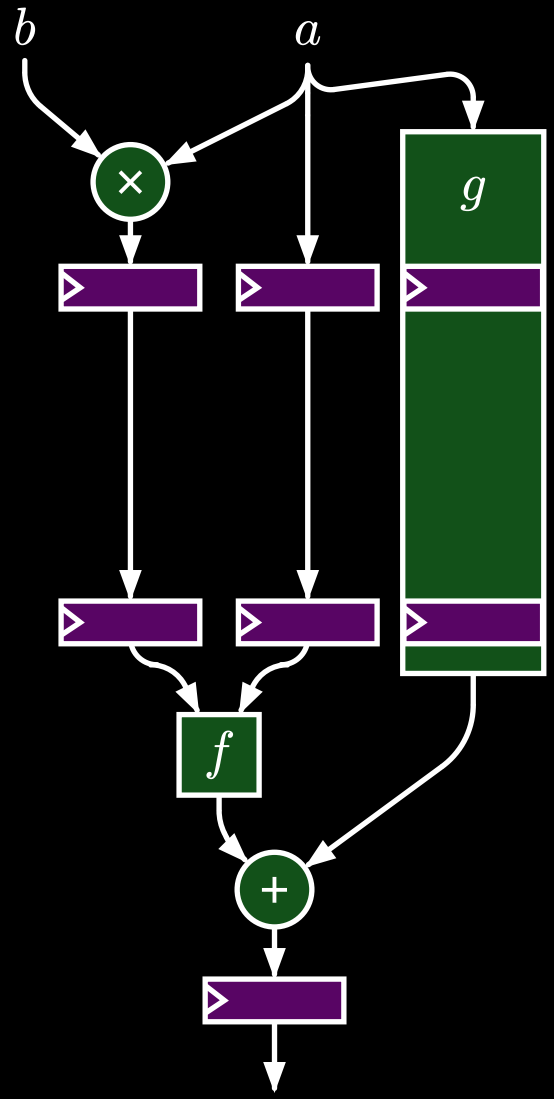
Pipelines

```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = inst(2) g(clk, a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



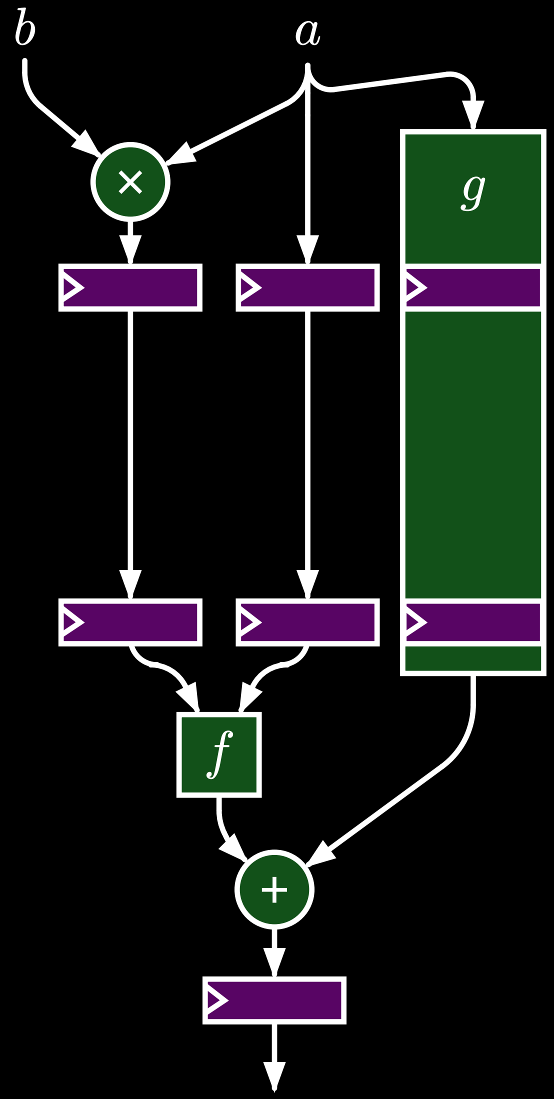
Pipelines

```
pipeline(2) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = inst(2) g(clk, a);
    let product = a*b;
reg;

    let sum = x + f(a, product);
reg;
    sum
}
```



Pipelines

```
pipeline(3) X(clk: clock, a: int<32>, b: int<32>)
    -> int<33> {
    let x = inst(2) g(clk, a);
    let product = a*b;
reg;
reg;
    let sum = x + f(a, product);
reg;
    sum
}
```



Instantiating Things

Definition:	**Instantiation**:
`fn trunc(...)`	`trunc(x + y)`
`entity blinky(...)`	`inst blinky(clk, rst)`
`pipeline(2) pipe(...)`	`inst(2) pipe(...)`

Definition:	**Instantiation:**
`fn trunc(...)`	`trunc(x + y)`
`entity blinky(...)`	`inst blinky(clk, rst)`
`pipeline(2) pipe(...)`	`inst(2) pipe(...)`

`fn` without keyword

Definition:

```
fn trunc(...)

entity blinky(...)

pipeline(2) pipe(...)
```

Instantiation:

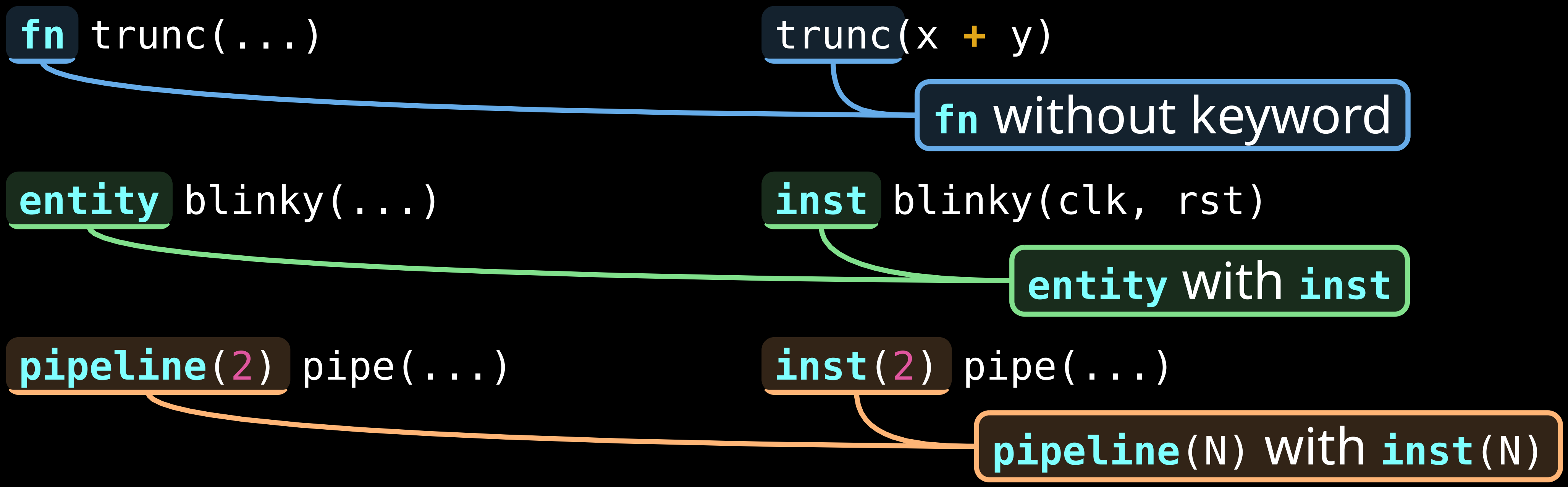
```
trunc(x + y)

inst blinky(clk, rst)

inst(2) pipe(...)
```

fn without keyword

entity with inst

Definition:

Instantiation:



```
fn trunc(...)
```

```
trunc(x + y)
```

fn without keyword

```
entity blinky(...)
```

```
inst blinky(clk, rst)
```

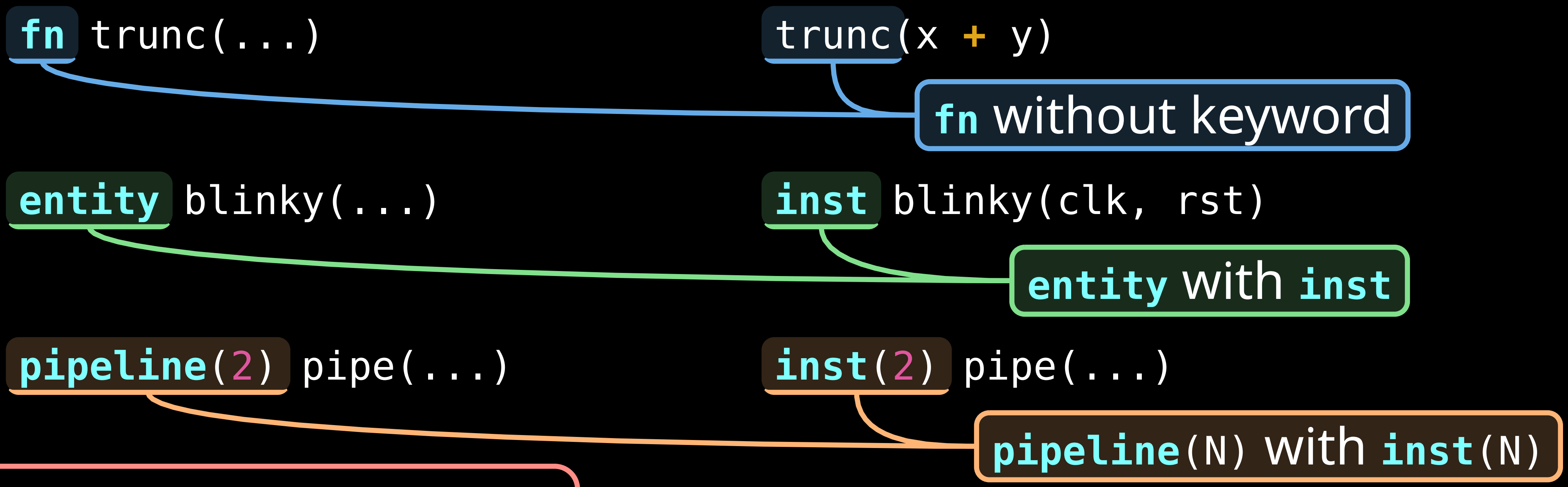
entity with inst

```
pipeline(2) pipe(...)
```

```
inst(2) pipe(...)
```

pipeline(N) with inst(N)

Definition: **Instantiation:**



`fn trunc(...)`

`trunc(x + y)`

`fn` without keyword

`entity blinky(...)`

`inst blinky(clk, rst)`

`entity` with `inst`

`pipeline(2) pipe(...)`

`inst(2) pipe(...)`

`pipeline(N)` with `inst(N)`

Encode assumptions!

Passing arguments

```
fn create_point(x: int<8>, y: int<8>)
```

Passing arguments

```
fn create_point(x: int<8>, y: int<8>)

// Positional arguments
create_point(5, 7)
```

Passing arguments

```
fn create_point(x: int<8>, y: int<8>)

// Positional arguments
create_point(5, 7)

// Named arguments
create_point$(y: 7, x: 5)
```

Passing arguments

```
fn create_point(x: int<8>, y: int<8>)

// Positional arguments
create_point(5, 7)

// Named arguments
create_point$(y: 7, x: 5)

// Shorthand named arguments
let x = ...
create_point$(x, y: 7)
```

Passing arguments

```
fn create_point(x: int<8>, y: int<8>)

// Positional arguments
create_point(5, 7)

// Named arguments
create_point$(y: 7, x: 5)

// Shorthand named arguments
let x = ...
create_point$(x, y: 7)
```

Same name in function as local

Types

```
// Primitive types
int<N>
uint<N>
bool
clock
```

Definition

```
struct Struct {
  x: int<8>,
  y: bool,
}
```

Definition

```
struct Struct {
  x: int<8>,
  y: bool,
}
```

Access

```
value.x
```

Definition

```
struct Struct {
  x: int<8>,
  y: bool,
}
```

Access

```
value.x
```

Instantiation

```
// Positional
Struct(x, y)
// Named
Struct$(x, y)
```

Tuples

Definition

```
(int<8>, bool)
```

Tuples

Definition

```
(int<8>, bool)
```

Access

```
value#0
```

Tuples

Definition

```
(int<8>, bool)
```

Access

```
value#0
```

Instantiation

```
(5, true)
```

Arrays

Definition

```
[int<8>; 3]
```

Arrays

Definition

```
[int<8>; 3]
```

Access

```
value[i]

value[0..1]
```

Arrays

Definition

```
[int<8>; 3]
```

Access

```
value[i]

value[0..1]
```

Instantiation

```
[1, 2, 3]
```

Destructuring

```
let (x, y) = value;
```

Destructuring

```
let (x, y) = value;

let [x, y] = array;
```

Destructuring

```
let (x, y) = value;

let [x, y] = array;

let Struct(x, y) = value;
```

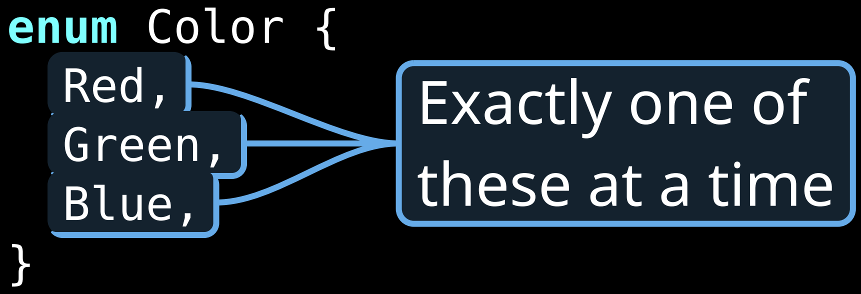
Destructuring

```
let (x, y) = value;

let [x, y] = array;

let Struct$(x, y) = value;
```

Can also be named arguments

More Interesting Types

```
enum Color {
  Red,
  Green,
  Blue,
}
```



```
enum Color {
  Red,
  Green,
  Blue,
}
```

Exactly one of these at a time

```
enum Color {
  Red,
  Green,
  Blue,
}
```

```
fn to_rgb(color: Color) -> [uint<8>; 3] {
  match color {
    Color::Red => [255, 0, 0],
    Color::Green => [0, 255, 0],
    Color::Blue => [0, 0, 255],
  }
}
```

```
enum Color {
  Red,
  Green,
  Blue,
  Grayscale {
    brightness: uint<8>
  },
  Custom {
    r: uint<8>,
    g: uint<8>,
    b: uint<8>
  },
}
```

```
fn to_rgb(color: Color) -> [uint<8>; 3] {
  match color {
    Color::Red => [255, 0, 0],
    Color::Green => [0, 255, 0],
    Color::Blue => [0, 0, 255],
  }
}
```

```
enum Color {
  Red,
  Green,
  Blue,
  Grayscale {
    brightness: uint<8>
  },
  Custom {
    r: uint<8>,
    g: uint<8>,
    b: uint<8>
  },
}
```

```
fn to_rgb(color: Color) -> [uint<8>; 3] {
  match color {
    Color::Red => [255, 0, 0],
    Color::Green => [0, 255, 0],
    Color::Blue => [0, 0, 255],
    Color::Grayscale(b) => [b, b, b],
  }
}
```

```
enum Color {
  Red,
  Green,
  Blue,
  Grayscale {
    brightness: uint<8>
  },
  Custom {
    r: uint<8>,
    g: uint<8>,
    b: uint<8>
  },
}
```

```
fn to_rgb(color: Color) -> [uint<8>; 3] {
  match color {
    Color::Red => [255, 0, 0],
    Color::Green => [0, 255, 0],
    Color::Blue => [0, 0, 255],
    Color::Grayscale(b) => [b, b, b],
  }
}
```

Match accesses the payload

So it can be used in the result

```
enum Color {
  Red,
  Green,
  Blue,
  Grayscale {
    brightness: uint<8>
  },
  Custom {
    r: uint<8>,
    g: uint<8>,
    b: uint<8>
  },
}
```

```
fn to_rgb(color: Color) -> [uint<8>; 3] {
  match color {
    Color::Red => [255, 0, 0],
    Color::Green => [0, 255, 0],
    Color::Blue => [0, 0, 255],
    Color::Grayscale(b) => [b, b, b],
    Color::Custom$(r, g, b) => [r, g, b],
  }
}
```

Enum Example: `Option`

```
enum Option<T> {
  None,
  Some{val: T}
}
```

Enum Example: `Option`

Either `None`

```
enum Option<T> {
  None,
  Some{val: T}
}
```

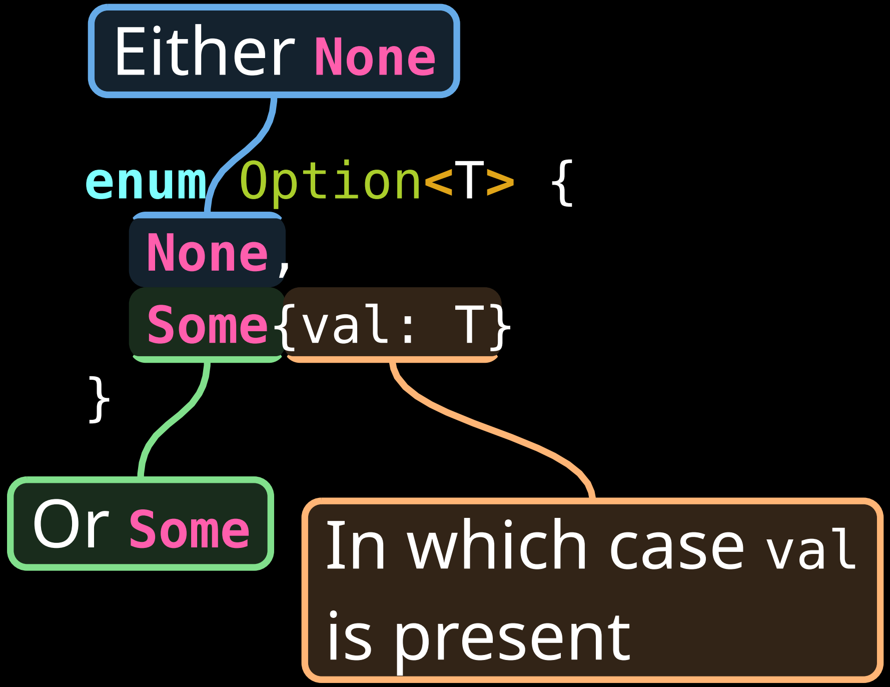
Enum Example: `Option`

Either `None`

```
enum Option<T> {
  None,
  Some{val: T}
}
```

Or `Some`

Enum Example: `Option`



Either `None`

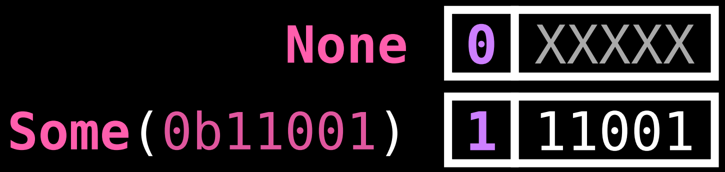
```
enum Option<T> {
  None,
  Some{val: T}
}
```

Or `Some`

In which case `val` is present

Enum Example: `Option`

```
enum Option<T> {
  None,
  Some{val: T}
}
```



```
         None  | 0 | XXXXX |
Some(0b11001)  | 1 | 11001 |
```

Enum Example: `Option`

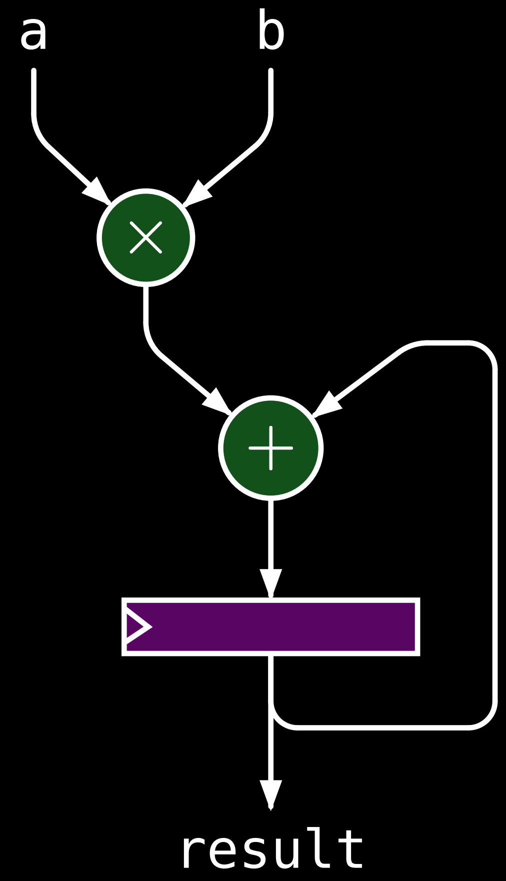
```
enum Option<T> {
  None,
  Some{val: T}
}
```

None	0	XXXXX
Some(0b11001)	1	11001

Intuitively: valid signal + validated data

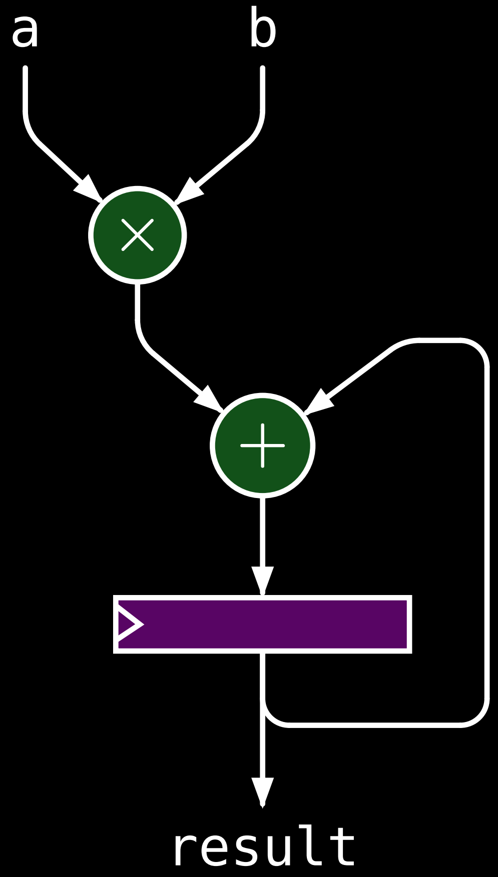
Option Example: MAC

```
entity mac(
  clk: clock, rst: bool,
  input: (int<16>, int<16>)
) -> int<40> {






}
```



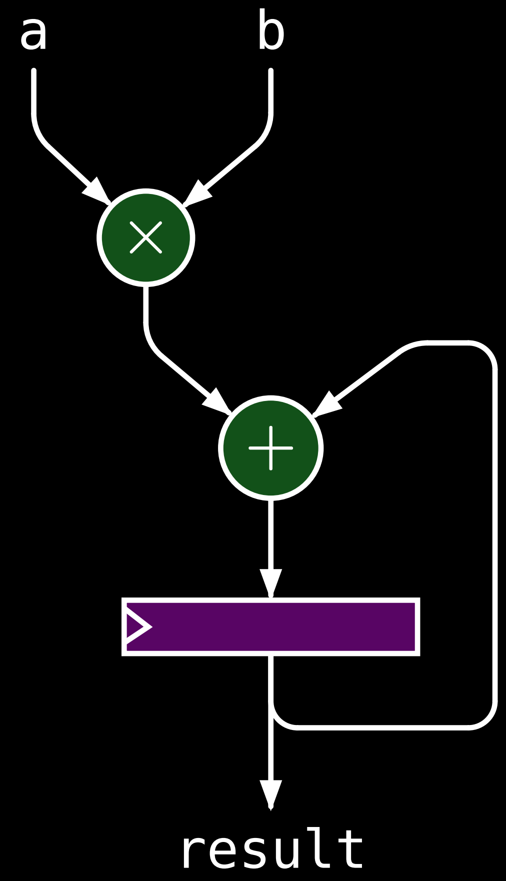
Option Example: MAC

```
entity mac(
  clk: clock, rst: bool,
  input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;




}
```



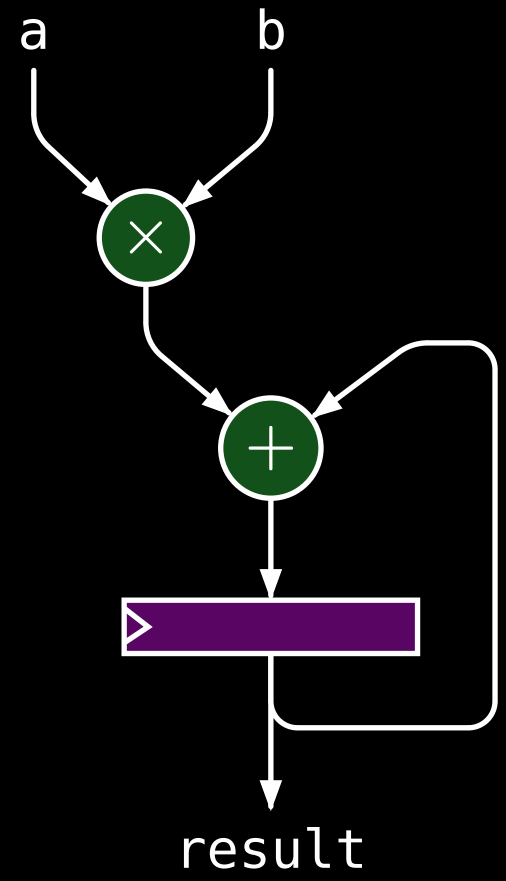
Option Example: MAC

```
entity mac(
  clk: clock, rst: bool,
  input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
          product + sum;


}
```



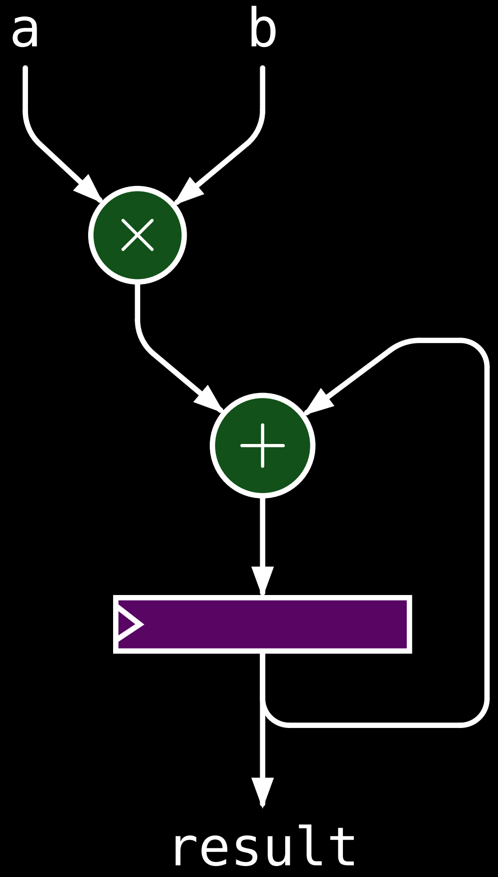
Option Example: MAC

```
entity mac(
  clk: clock, rst: bool,
  input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
          product + sum;


  sum
}
```



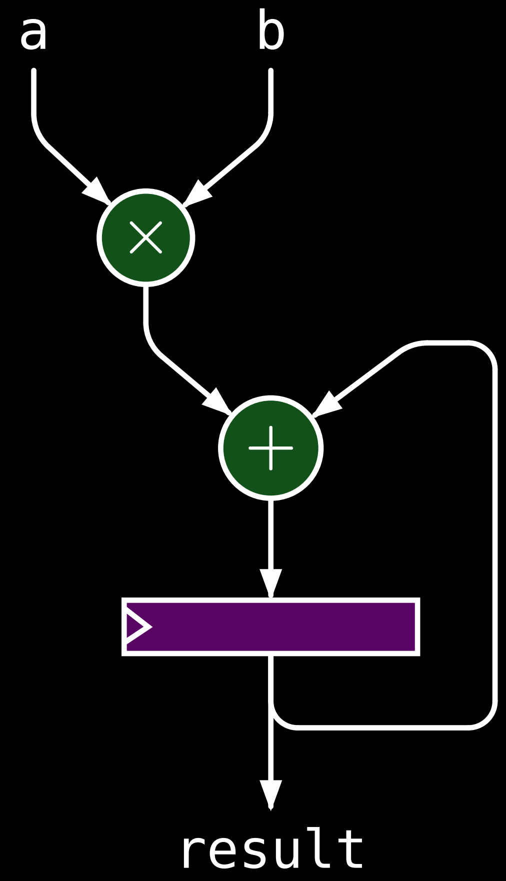
Option Example: MAC

```
entity mac(
  clk: clock, rst: bool,
  input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
          sext(product) + trunc(sum);

  sum
}
```



Option Example: MAC

```
entity mac(
  clk: clock, rst: bool,
  input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
          product + sum;


  sum
}
```



Option Example: MAC

```
entity data_producer(...)
    -> (int<16>, int<16>)
{...}


let mac_in = inst data_producer(...)

let mac_out = inst mac(clk, rst, mac_in);
```

Option Example: MAC

```
entity data_producer(...)
    -> (int<16>, int<16>)
{...}


let mac_in = inst data_producer(...)

let mac_out = inst mac(clk, rst, mac_in);
```

❗ **Requirement change**

Data is not produced every clock cycle

Option Example: MAC

```
entity data_producer(...)
    -> (int<16>, int<16>)
{...}


let mac_in = inst data_producer(...)

let mac_out = inst mac(clk, rst, mac_in);
```

Option Example: MAC

```
entity data_producer(...)
    -> Option<(int<16>, int<16>)>
{...}


let mac_in = inst data_producer(...)

let mac_out = inst mac(clk, rst, mac_in);
```

Option Example: MAC

```
entity data_producer(...)
    -> Option<(int<16>, int<16>)>
{...}


let mac_in = inst data_producer(...)

let mac_out = inst mac(clk, rst, mac_in);
```

Now `Option<...>`

❗ Type error

```
entity mac(
  clk: clock, rst: bool, input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: (int<16>, int<16>)
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let (a, b) = input;
  let product = a * b;

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

Not available every clock cycle anymore

❗ Type error

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => ...,
      None => ...,
  }]

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => ...,
  }]

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

Next compile error

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum reset(rst: 0) =
      sext(product) + trunc(sum);

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum reset(rst: 0) =
      match product {
        Some(product) => sum + product,
        None => sum
      }

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum reset(rst: 0) =
      match product {
        Some(product) => sum + product,
        None => sum
      }

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum reset(rst: 0) =
      match product {
        Some(product) => sum + product,
        None => sum
      }

  sum
}
```

```
entity mac(
  clk: clock, rst: bool, input: Option<(int<16>, int<16>)>
) -> int<40> {
  let product = match input {
      Some((a, b)) => Some(a * b),
      None => None,
  }]

  reg(clk) sum
      match product {
        Some(product) => sum + product,
        None => sum
      }

  sum
}
```

Once per sample,
or continuous output?

Enum example: FSMs

Make an LED blink thrice whenever a button is pressed

```
while True:
  if btn:
    for i in range(0, 3):
      led_on()
      wait(delay)
      led_off()
      wait(delay)
```

```
enum State {


}
```

```
while True:
  if btn:
    for i in range(0, 3):
      led_on()
      wait(delay)
      led_off()
      wait(delay)
```

```
enum State {
  Idle,

}
```

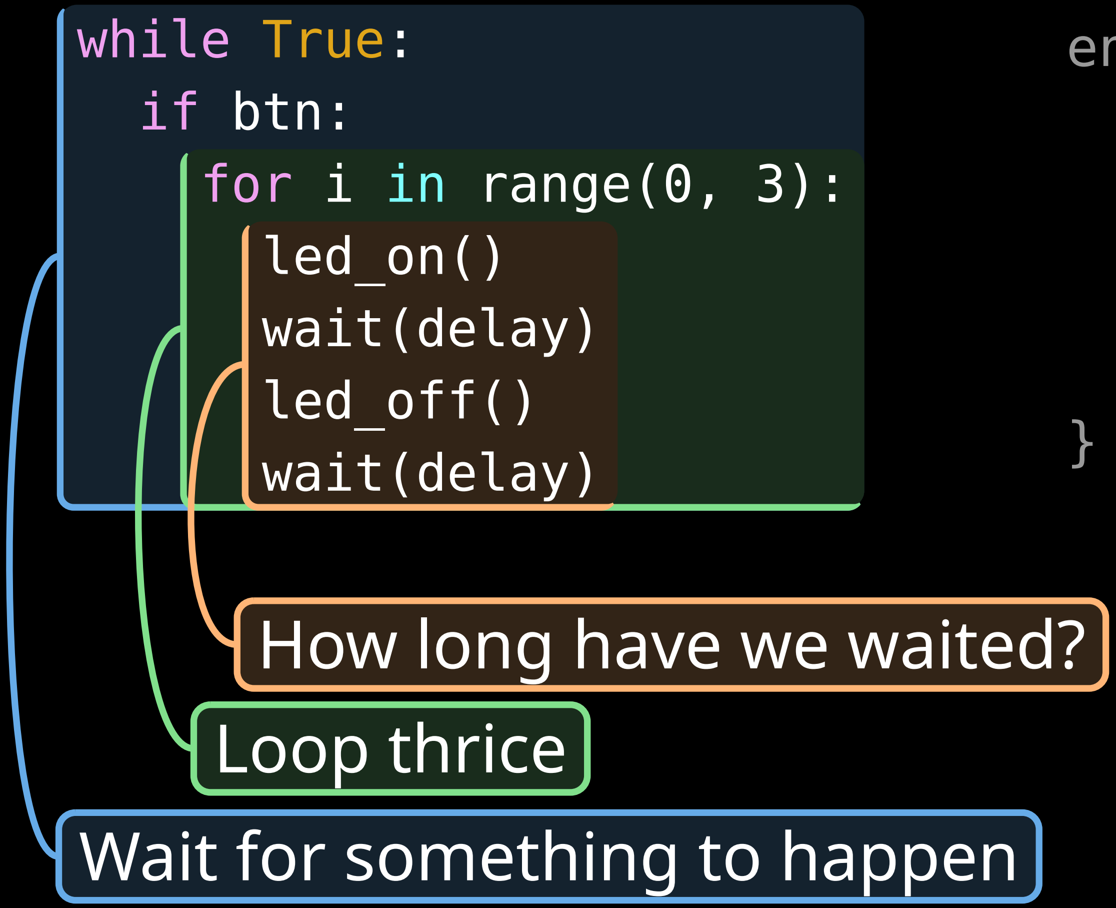
Wait for something to happen

```
while True:
  if btn:
    for i in range(0, 3):
      led_on()
      wait(delay)
      led_off()
      wait(delay)
```

Loop thrice

Wait for something to happen

```
enum State {
  Idle,
  Blink {
    blinks_left: uint<3>,

  }
}
```



```
while True:
  if btn:
    for i in range(0, 3):
      led_on()
      wait(delay)
      led_off()
      wait(delay)
```

How long have we waited?

Loop thrice

Wait for something to happen

```
enum State {
  Idle,
  Blink {
    blinks_left: uint<3>,
    on_duration: uint<15>,
  }
}
```

```
while True:
  if btn:
    for i in range(0, 3):
      led_on()
      wait(delay)
      led_off()
      wait(delay)
```

```
enum State {
  Idle,
  Blink {
    blinks_left: uint<3>,
    on_duration: uint<15>,
  }
}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{




}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =



}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {




    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) =>


    }


}
```

Match works on tuples

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) =>



    }


}
```

And supports patterns

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,



    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),


    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),


    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left, duration), _) => ???,

    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => Stat
      (State::Idle, true) =>
        State::Blink$(blinks_left:

      (State::Blink$(blinks_left, duration), _) => ???,

    }


}
```

We don’t care about the input while blinking

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left, duration), _) => ???,

    }

}
```

Back to idle
or more blinks?

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left, duration), _) => ???,

    }


}
```

Are we done waiting?

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left: 0, duration: 0), _) =>
          State::Idle,

    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left: 0, duration: 0), _) =>
          State::Idle,
      (State::Blink$(blinks_left, duration: 0), _) =>
          State::Blink$(blinks_left: trunc(blinks_left-1), duration: 10_000),

    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left: 0, duration: 0), _) =>
          State::Idle,
      (State::Blink$(blinks_left, duration: 0), _) =>
          State::Blink$(blinks_left: trunc(blinks_left-1), duration: 10_000),

    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left: 0, duration: 0), _) =>
          State::Idle,
      (State::Blink$(blinks_left, duration: 0), _) =>
          State::Blink$(blinks_left: trunc(blinks_left-1), duration: 10_000),
      (State::Blink$(blinks_left, duration), _) =>
          State::Blink(blinks_left, trunc(duration_left - 1))
    }


}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left: 0, duration: 0), _) =>
          State::Idle,
      (State::Blink$(blinks_left, duration: 0), _) =>
          State::Blink$(blinks_left: trunc(blinks_left-1), duration: 10_000),
      (State::Blink$(blinks_left, duration), _) =>
          State::Blink(blinks_left, trunc(duration_left - 1))
    }


}
```

Prioritized in order

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
    match (state, input) {
      (State::Idle, false) => State::Idle,
      (State::Idle, true) =>
        State::Blink$(blinks_left: 2, duration: 10_000),

      (State::Blink$(blinks_left: 0, duration: 0), _) =>
          State::Idle,
      (State::Blink$(blinks_left, duration: 0), _) =>
          State::Blink$(blinks_left: trunc(blinks_left-1), duration: 10_000),
      (State::Blink$(blinks_left, duration), _) =>
          State::Blink(blinks_left, trunc(duration_left - 1))
    }


}
```

The compiler will check your work. Missing cases are errors

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
     ...

  match state {


  }
}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
     ...

  match state {
    State::Idle => false,

  }
}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
     ...

  match state {
    State::Idle => false,
    State::Blink$(blinks_left: _, duration_left) => ...,
  }
}
```

```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
     ...

  match state {
    State::Idle => false,
    State::Blink$(blinks_left: _, duration_left) =>
        duration_left < 10_000 / 2,
  }
}
```

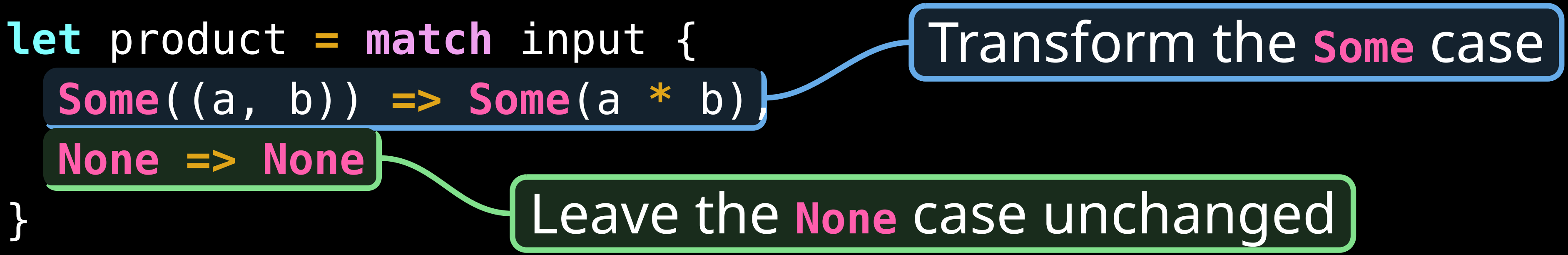
```
entity blink_thrice(clk: clock, rst: bool, btn: bool)
    -> bool
{
  reg(clk) state reset(rst: State::Idle) =
     ...

  match state {
    State::Idle => false,
    State::Blink$(blinks_left: _, duration_left) =>
        duration_left < 10_000 / 2,
  }
}
```

Methods

```rust
let product = match input {
  Some((a, b)) => Some(a * b),
  None => None
}
```

```
let product = match input {
  Some((a, b)) => Some(a * b),
  None => None
}
```

Transform the `Some` case



```
let product = match input {
    Some((a, b)) => Some(a * b),
    None => None
}
```

Transform the Some case

Leave the None case unchanged

```
product.map(...)
product.map(...)
```

Transform the contained value

```
product.map(fn ((a, b)) { a * b })
```

Transform the contained value

Using a lambda function

```
fn to_rgb(color: Color) -> [uint<8>; 3] {
  match color {
    Color::Red => [255, 0, 0],
    Color::Green => [0, 255, 0],
    Color::Blue => [0, 0, 255],
    Color::Grayscale(b) => [b, b, b],
    Color::Custom$(r, g, b) => [r, g, b],
  }
}
```

```
impl Color {
  fn to_rgb(self) -> [uint<8>; 3] {
    match self {
      Color::Red => [255, 0, 0],
      Color::Green => [0, 255, 0],
      Color::Blue => [0, 0, 255],
      Color::Grayscale(b) => [b, b, b],
      Color::Custom$(r, g, b) => [r, g, b],
    }
  }
}
```

```
struct port Rv<T> {
  data: &Option<T>,
  ready: inv &bool
}
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_ethernet_bytes(clk, rst);
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_ethernet_bytes(clk, rst);
```

```
error Option<_> has no method into_ethernet_bytes
 ┌─ src/main.spade:3
 │
2│ let pins = camera_feed
 │            ~~~~~~~~~~~ This has type Option<_>
3│   .inst into_ethernet_bytes()
 │         ^^^^^^^^^^^^^^^^^^^ No such method
 help: The method exists for Rv<_>
```

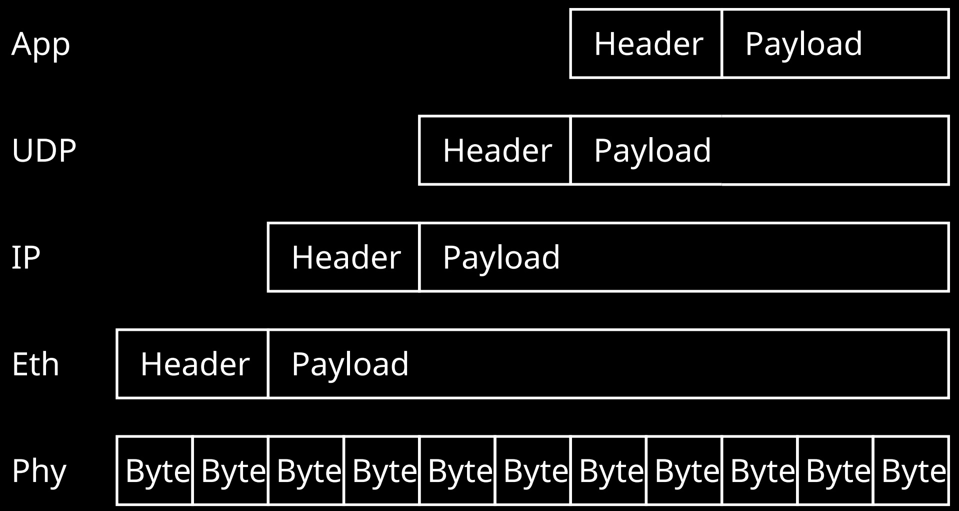
```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst into_ethernet_bytes(clk, rst);
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_ethernet_bytes(clk, rst);
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_ethernet_bytes(clk, rst);
```

```
error Rv<PixelsPackets> has no method
      into_ethernet_bytes
   ┌─ src/main.spade:5
   │
2  │   let eth_pins = udp_stream
   │  ┌─────────────────────────────────────────'
3  │  │     .inst into_rv_fifo::<1024>()
4  │  │     .inst packetize$(len: 1480)
   │  └─────' This has type Rv<PixelPackets>
5  │        .inst into_ethernet_bytes(clk, rst)
   │              ^^^^^^^^^^^^^^^^^^^ No such method
 help: The method exists for Rv<IpPackets>
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)

.inst into_ethernet_bytes(clk, rst);
```



App: Header | Payload

UDP: Header | Payload

IP: Header | Payload

Eth: Header | Payload

Phy: Byte | Byte | Byte | Byte | Byte | Byte | Byte | Byte | Byte | Byte | Byte

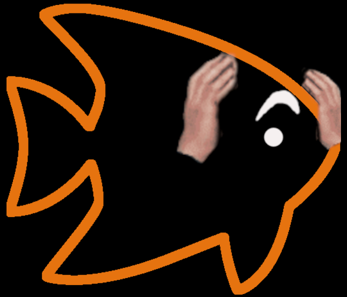
```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ethernet_bytes(clk, rst);
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet_bytes(clk, rst);
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
```

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
```

Oh no, we forgor💀 about timing.



```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst buffer_headers()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
  .inst buffer()
```

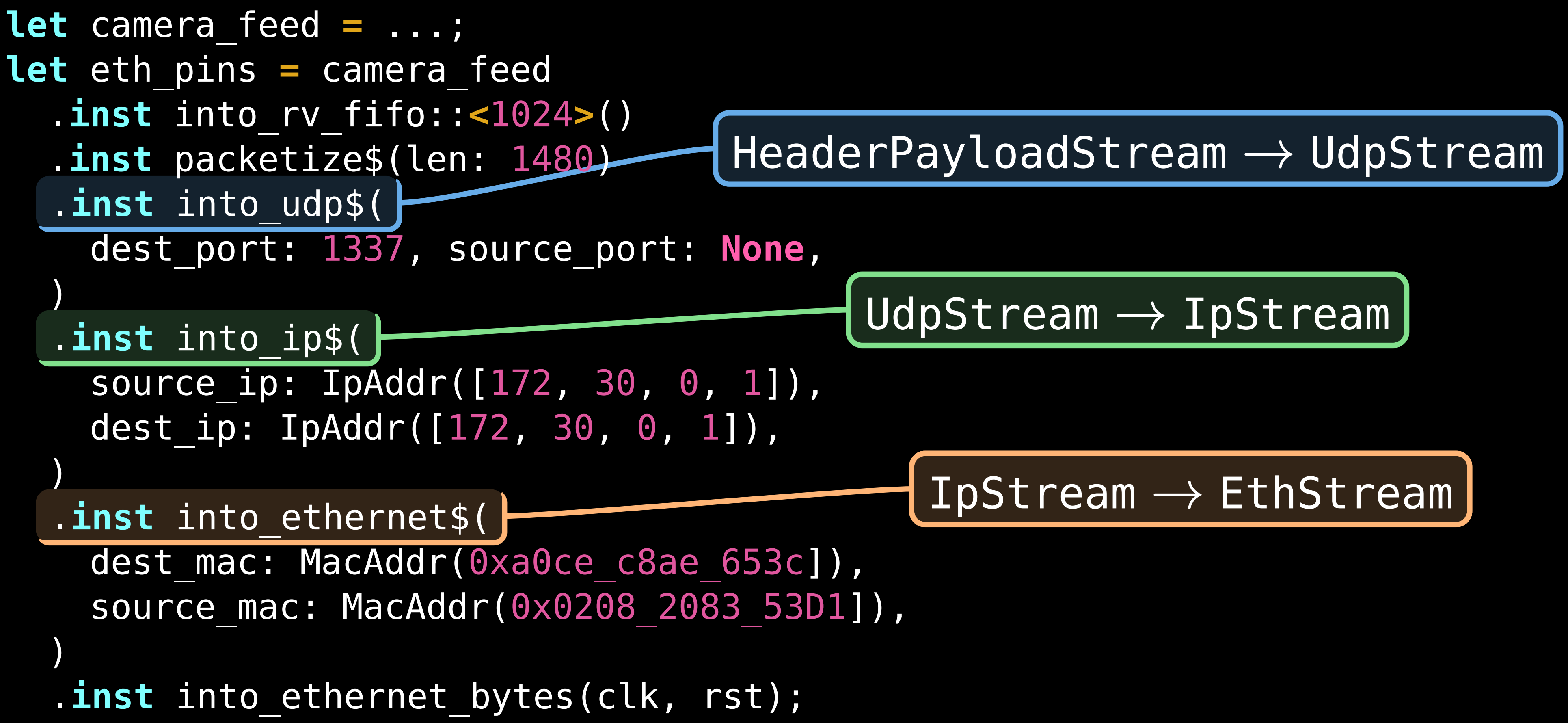
```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst buffer_headers()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
  .inst buffer()
```

HeaderPayloadStream → UdpStream

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst buffer_headers()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
```

HeaderPayloadStream ⟶ UdpStream

UdpStream ⟶ IpStream



```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
```

HeaderPayloadStream → UdpStream

UdpStream → IpStream

IpStream → EthStream

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst into_ethernet_bytes(clk, rst);
```

HeaderPayloadStream → UdpStream

UdpStream → IpStream

IpStream → EthStream

Type safe transformations

```
let camera_feed = ...;
let eth_pins = camera_feed
  .inst into_rv_fifo::<1024>()
  .inst packetize$(len: 1480)
  .inst into_udp$(
    dest_port: 1337, source_port: None,
  )
  .inst into_ip$(
    source_ip: IpAddr([172, 30, 0, 1]),
    dest_ip: IpAddr([172, 30, 0, 1]),
  )
  .inst into_ethernet$(
    dest_mac: MacAddr(0xa0ce_c8ae_653c]),
    source_mac: MacAddr(0x0208_2083_53D1]),
  )
  .inst append_lower_priority(inst handle_arp_icmp(...))
  .inst into_ethernet_bytes(clk, rst);
```

Respond to pings and
Address Resolution Requests

Tooling

Tooling

Will make or break a language

Compiler

- Takes your Spade, emits Verilog
- More importantly, provides helpful and guiding **error messages**

The Swim build tool

- Manages dependencies
- Runs synthesis tools
- Installs CAD tools

Editor Integration with LSP

Provide that 0.1 second interactivity where possible

Tests

Tests

- Spade is designed for *synthesizeable* hardware

Tests

- Spade is designed for *synthesizeable* hardware
- Testbenches are largely a software problem

Tests

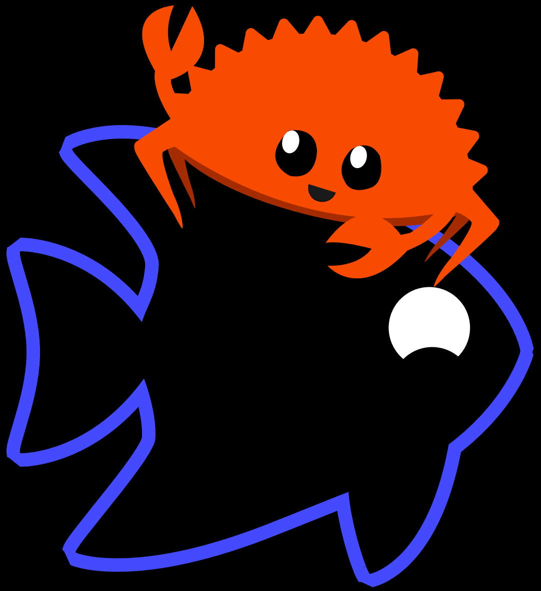
- Spade is designed for *synthesizeable* hardware
- Testbenches are largely a software problem
- Cocotb with Spade intgreation

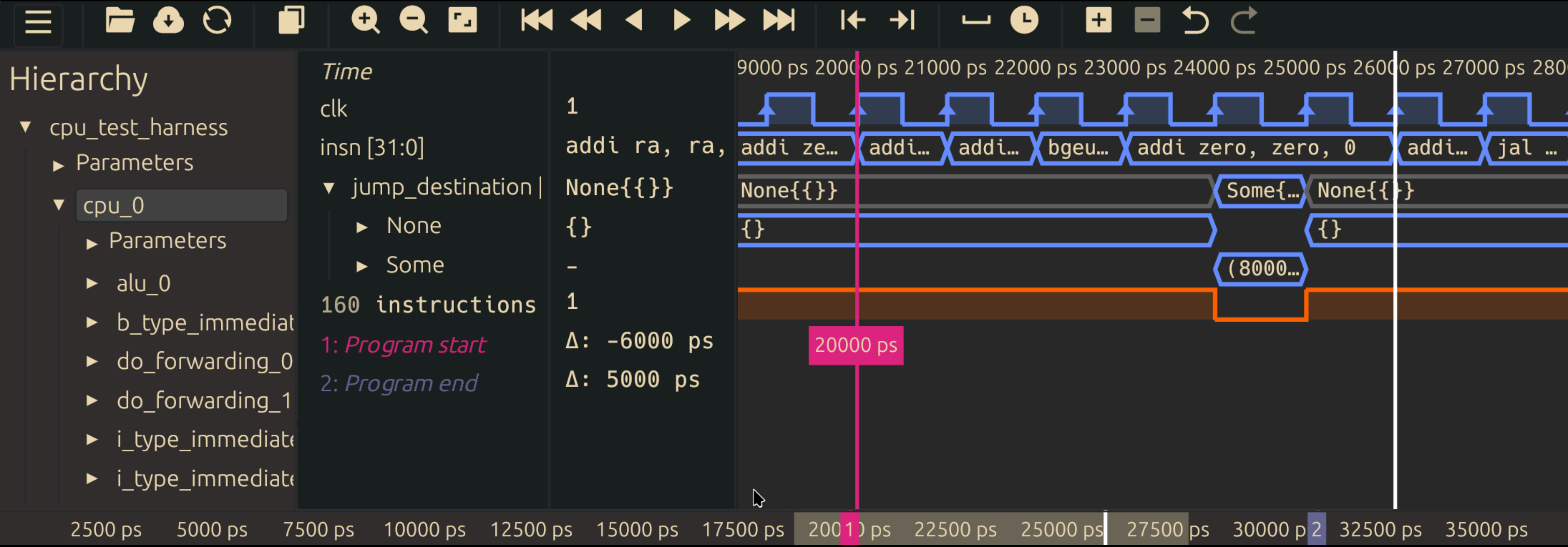
Tests

- Spade is designed for *synthesizeable* hardware
- Testbenches are largely a software problem
- Cocotb with Spade intgreation
- Rust based testbenches are on the way

Tests

- Spade is designed for *synthesizeable* hardware
- Testbenches are largely a software problem
- Cocotb with Spade intgreation
- Rust based testbenches are on the way





Hierarchy

- ▼ cpu_test_harness
  - ▸ Parameters
  - ▼ cpu_0
    - ▸ Parameters
    - ▸ alu_0
    - ▸ b_type_immediat
    - ▸ do_forwarding_0
    - ▸ do_forwarding_1
    - ▸ i_type_immediate
    - ▸ i_type_immediate

Time	
clk	1
insn [31:0]	addi ra, ra,
▼ jump_destination		None{{}}
▸ None	{}
▸ Some	-
160 instructions	1
1: *Program start*	Δ: -6000 ps
2: *Program end*	Δ: 5000 ps

Surrounding tooling

- Documentation generation
- Auto-formatting
- More powerful LSP

Conclusions

Use your language to

- *encode* your assumptions
- *alert you* when you violate them
- *guide* you when refactoring

Lab time!

Play with Spade’s unique features

- Pipelining
- Type system

Play with Spade’s unique features

- Pipelining
- Type system

https://docs.spade-lang.org, go to the **Agile Hardware Design Tutorial**

Play with Spade’s unique features

- Pipelining
- Type system

https://docs.spade-lang.org, go to the **Agile Hardware Design Tutorial**

Estimating difficulty is hard, do task 1 and 2 in both parts first