Programming ML algorithms in hardware,

sanely, using Haskell and Rust!

Daniel Hensley and Ryan Orendorff

November 13th, 2020

github.com/ryanorendorff/sbtb-2020-type-safe-fpga

Brief intro: Ryan Orendorff

I am a Research Scientist at Facebook Reality Labs Research (FRLR) working on Brain Computer Interface (BCI) research.

- I have been researching novel image and data reconstruction methods in the biomedical space for the last decade, usually using signal theory and linear algebra.
- I love to poke around with theorem proving, programming language theory and dependently typed languages such as Agda.

This work is done on personal time/equipment and is not sponsored by Facebook, nor is this talk related to work at Facebook.

Repos and other talks can be found here: github.com/ryanorendorff/

Brief intro: Daniel Hensley

I manage the software team at Magnetic Insight, Inc. (MI). We build medical imaging scanners with a focus on magnetic particle imaging (MPI).

- Background in signal processing, image reconstruction, applied math, and electromagnetic physics.
- Experience working on embedded and HW-facing code to operate big, scary machines. Correctness is very important!
- I've worked with FPGAs and Rust to great effect in production applications.

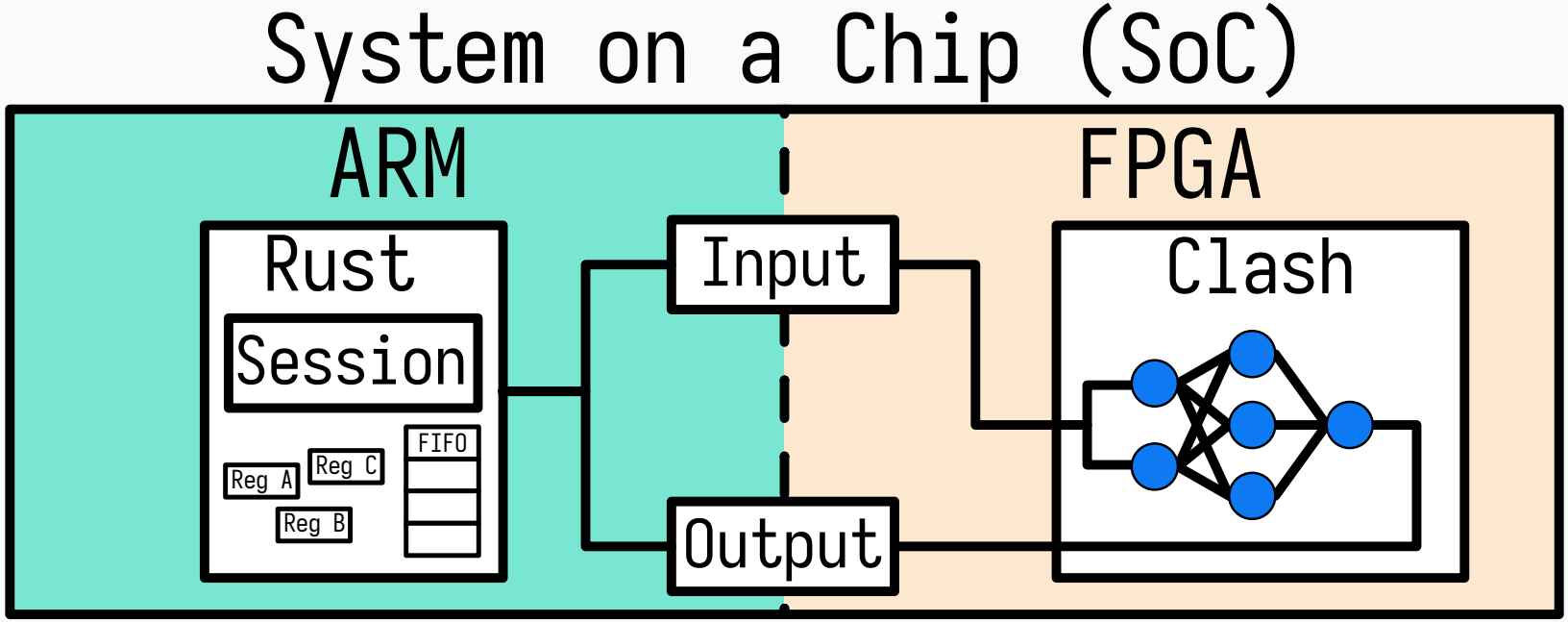
Outline

In this talk, we will go through

- Forward propagation (inference) for neural networks.
- How to implement forward propagation using Clash on an FPGA.
- How to make accessing the FPGA safe using Rust.

Architecture

We will implement a neural network *as hardware* on a Cyclone V chip.

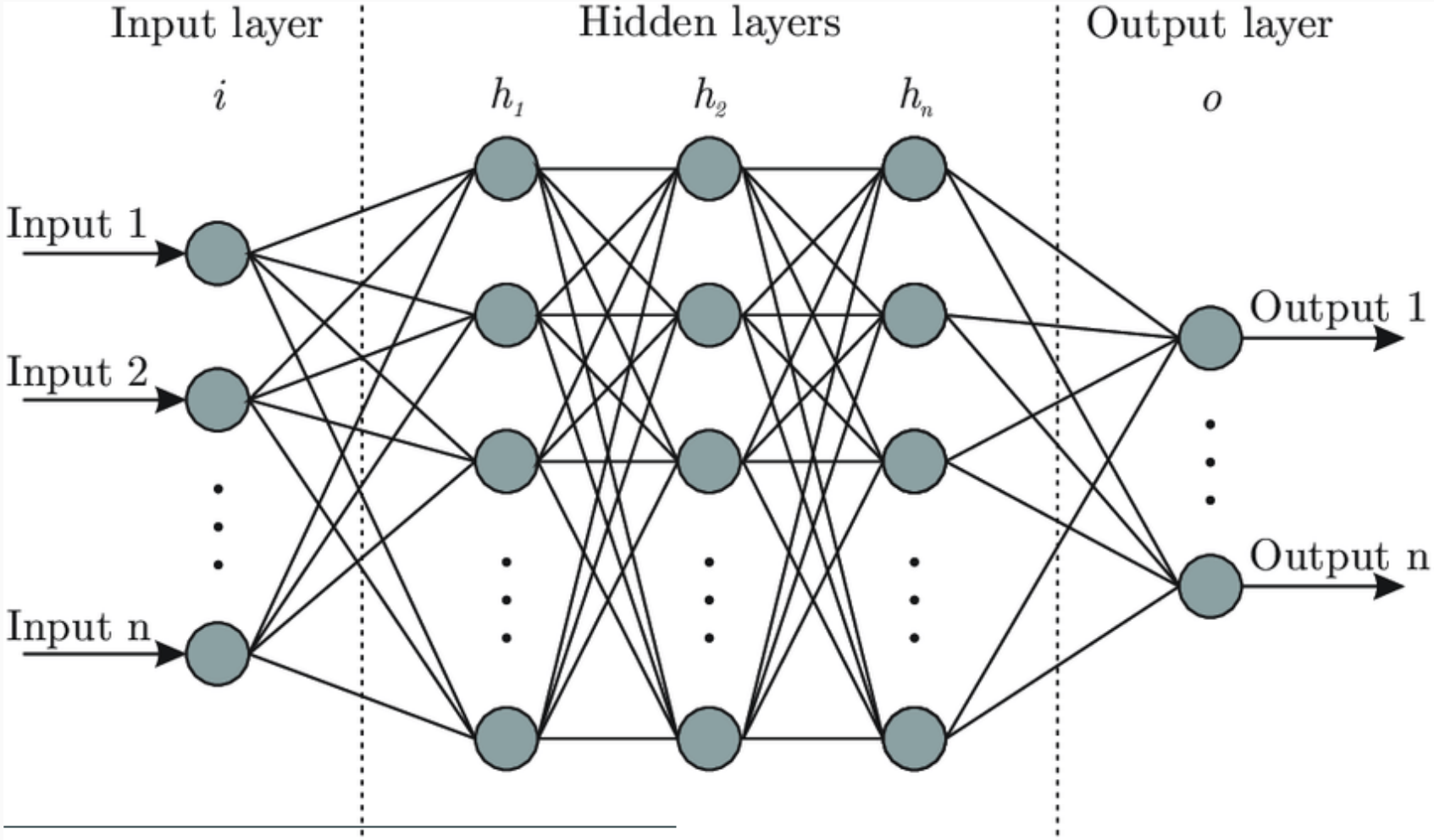


Implementing algorithms in hardware has several advantages:

- Low power
- High throughput (GPU-like or better)
- Deterministic timing

How is a neural network implemented?

A neural network is usually shown as a series of nodes with connections between them [1]. But what does that mean?



[1] https://www.kdnuggets.com/2019/11/designing-neural-networks.html

Neural network basics: how to go from one layer to another

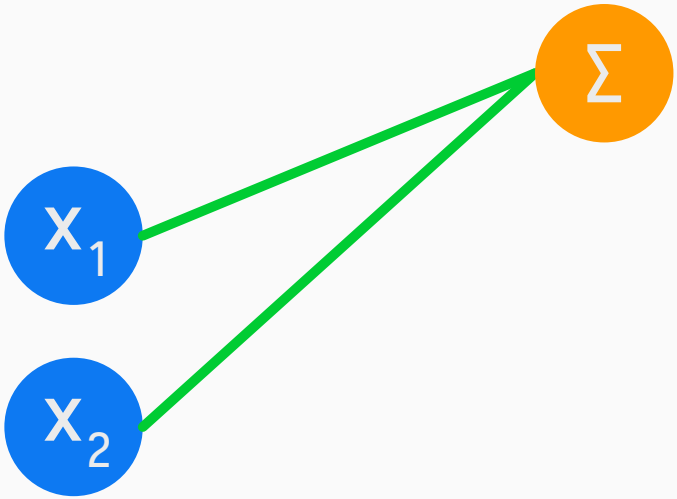
Going from one layer (x) to the next (y) is implemented using the equation $y = g(Mx + b)$.

$$y = g(Mx + b)$$

Neural network basics: how to go from one layer to another

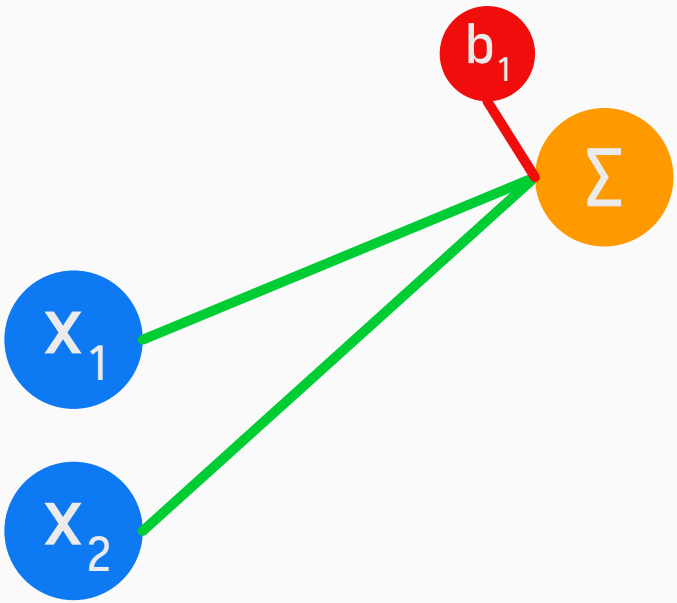
Going from one layer (x) to the next (y) is implemented using the equation $y = g(Mx + b)$.

$$y = g(Mx + b)$$



Neural network basics: how to go from one layer to another

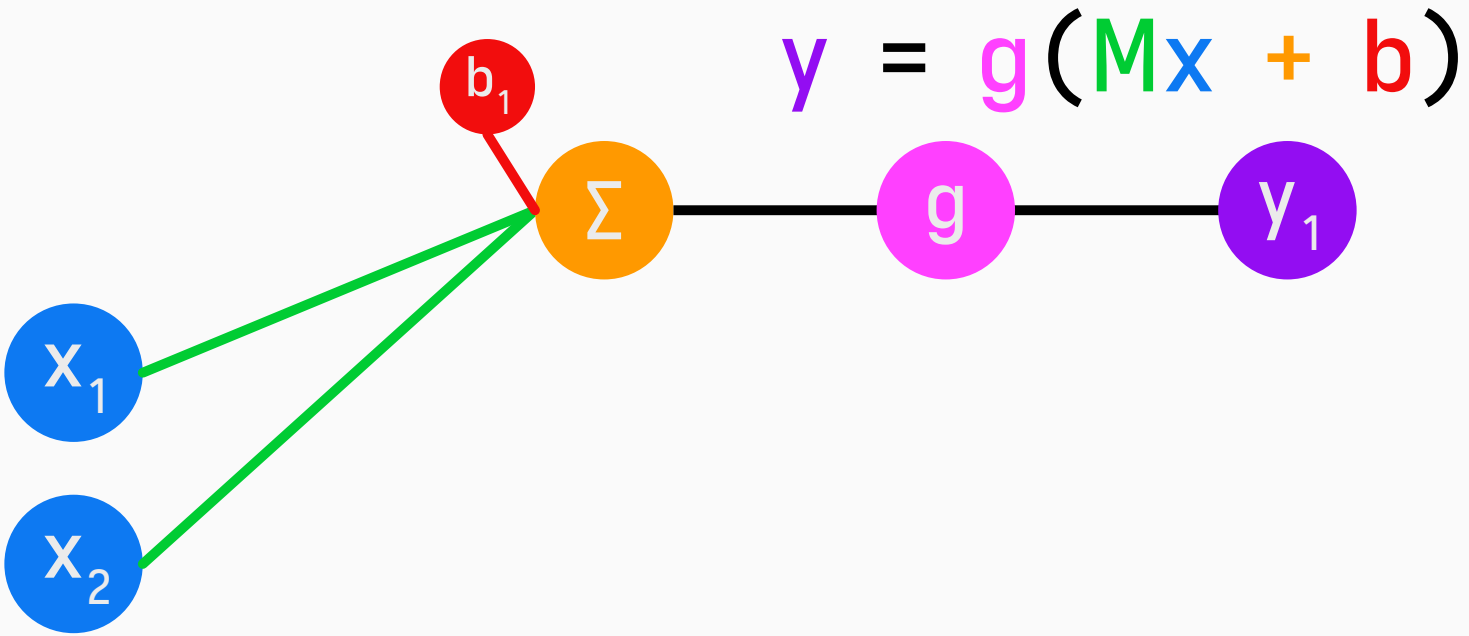
Going from one layer (x) to the next (y) is implemented using the equation $y = g(Mx + b)$.



$$y = g(Mx + b)$$

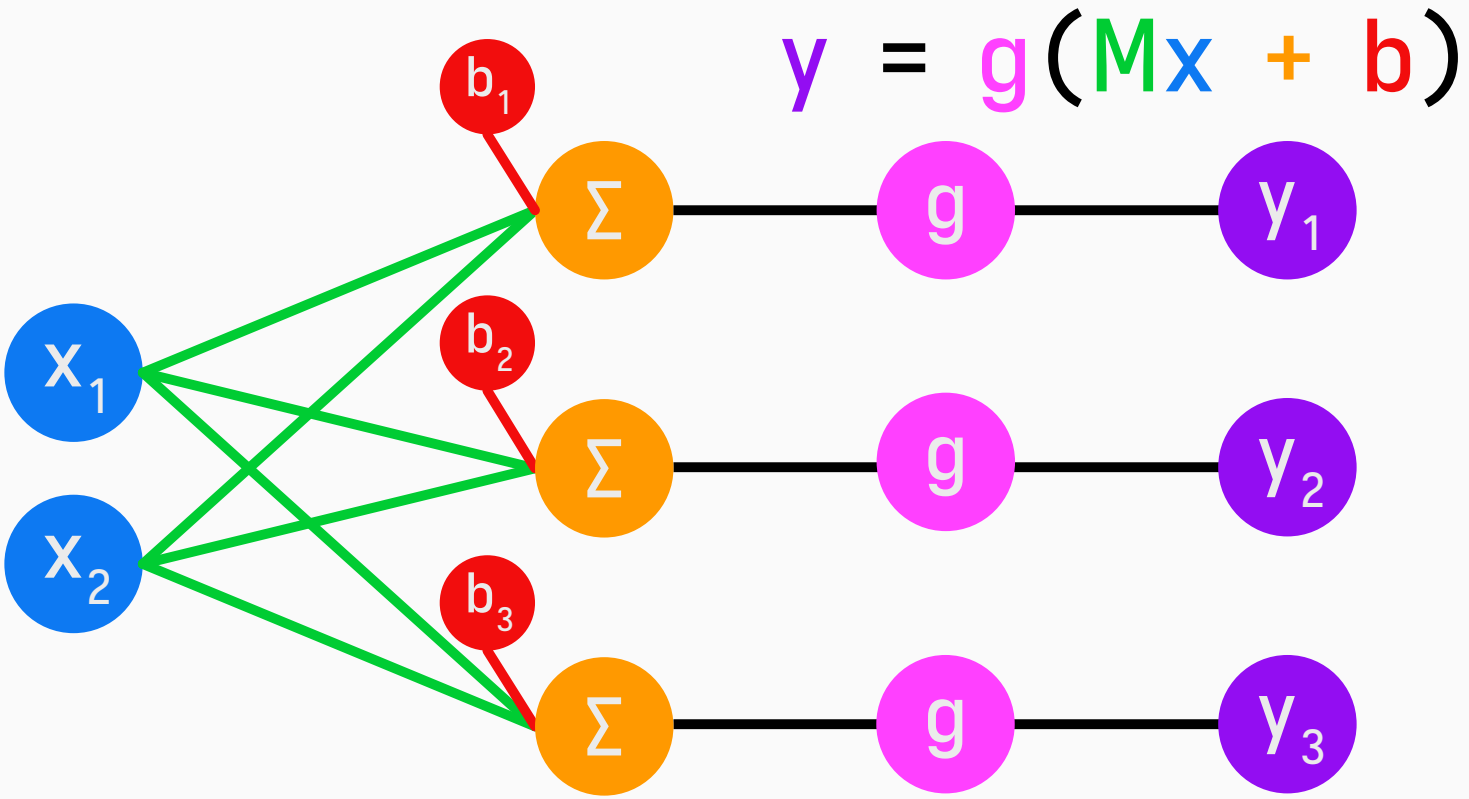
Neural network basics: how to go from one layer to another

Going from one layer (x) to the next (y) is implemented using the equation $y = g(Mx + b)$.



Neural network basics: how to go from one layer to another

Going from one layer (x) to the next (y) is implemented using the equation $y = g(Mx + b)$.



Let's implement the linear algebra basics

For the equation $y = g(Mx + b)$, we need to be able to add two vectors together and perform matrix-vector multiply.

Let's implement the linear algebra basics

For the equation $y = g(Mx + b)$, we need to be able to add two vectors together and perform matrix-vector multiply.

```haskell
-- Add two vectors: x + y
(<+>) :: (KnownNat n, Num a)
      => Vec n a -> Vec n a -> Vec n a
(<+>) = zipWith (+)
```

Let's implement the linear algebra basics: dot product

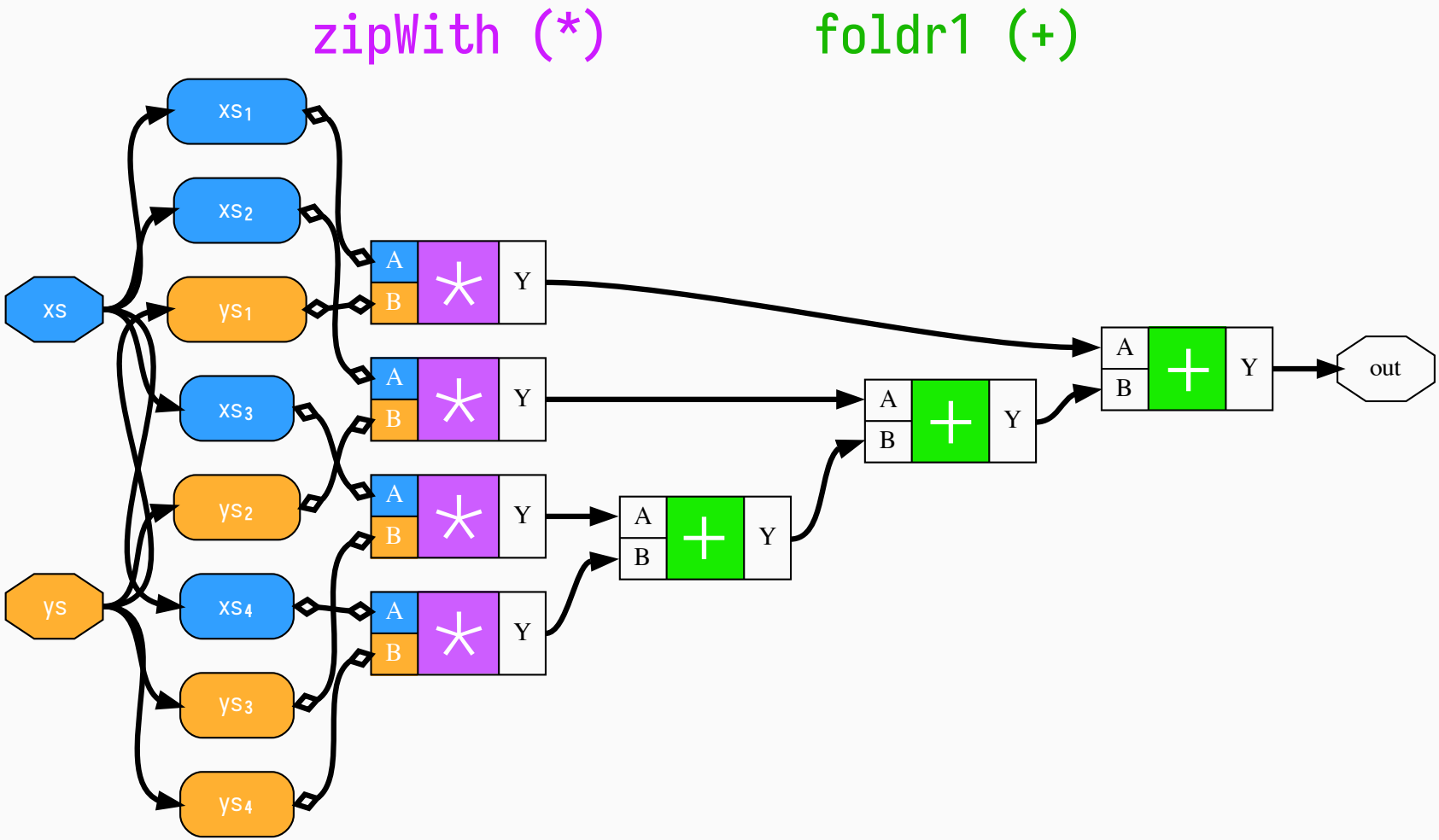
For the equation $y = g(Mx + b)$, we need to be able to add two vectors together and perform matrix-vector multiply.

```haskell
-- Dot product: ⟨x, y⟩
(<.>) :: (KnownNat n, Num a)
      => Vec n a -> Vec n a -> a
(<.>) xs ys = foldr1 (+) (zipWith (*) xs ys)
```

Compiling just the matrix vector multiply

If we compile just the dot product with 4 element vectors filled with `Int` :

```
(<.>) xs ys = foldr1 (+) (zipWith (*) xs ys)
```



Let's implement the linear algebra basics: matrix-vector multiply

To round out our $y = g(Mx + b)$, we need to know how to do the Mx part! This is performing the dot product with each row in the matrix.

```haskell
type Matrix m n a = Vec m (Vec n a)


-- Matrix vector multiply: Mx
(#>) :: (KnownNat m, KnownNat n, Num a)
     => Matrix m n a -> Vec n a -> Vec m a
(#>) m x = map (<.> x) m
```

Encoding the layer as a type called **LayerTransition**

Now that we have the basic linear algebra operators down, lets store everything we need for the $y = g(Mx + b)$.

```haskell
-- Transition of size m to size n
data LayerTransition (i :: Nat) (o :: Nat) a =
  LayerTransition
          { m :: Matrix o i a -- Connections M
          , b :: Vec o a      -- Bias b
          , g :: a -> a       -- Activation function g
          }
```

We can compose layers using a dependently typed list called **Network**

Now we will create a non-empty list of **LayerTransitions** using a GADT[2].

```
data Network (i :: Nat) (hs :: [Nat]) (o :: Nat) a where
```

[2] Inspired by Justin Le's dependent type NN structure.

We can compose layers using a dependently typed list called **Network**

Now we will create a non-empty list of **LayerTransitions** using a GADT[2].

```
data Network (i :: Nat) (hs :: [Nat]) (o :: Nat) a where

    OutputLayer :: (LayerTransition i o a) -> Network i '[] o a
```

[2] Inspired by Justin Le's dependent type NN structure.

We can compose layers using a dependently typed list called **Network**

Now we will create a non-empty list of **LayerTransitions** using a GADT[2].

```
data Network (i :: Nat) (hs :: [Nat]) (o :: Nat) a where

    OutputLayer :: (LayerTransition i o a) → Network i '[] o a

    (:>>) :: (KnownNat i, KnownNat o, KnownNat h)
          ⇒ LayerTransition i h a    -- New input layer
          → Network h hs o a         -- Existing network
          → Network i (h ': hs) o a  -- New network
```

To add a new input layer in front, the *number of output nodes from the new layer must match the number of input nodes to the existing network!*

[2] Inspired by Justin Le's dependent type NN structure.

runLayer consumes a **LayerTransition**, returning the output nodes

We can now run a layer transition by applying our equation $y = g(Mx + b)$ to some input vector x to output vector y.

```
runLayer :: (KnownNat i, KnownNat o, Num a)
         ⇒ LayerTransition i o a -- Layer to evaluate
         → Vec i a               -- Input nodes to the layer
         → Vec o a               -- Output nodes from the layer
runLayer (LayerTransition m b g) x =
  map g $ m #> x <+> b
-- Precisely y = g(Mx + b) from before!
```

runNet folds the **LaryTransitionss** down to output nodes

Now we can run our network by moving data from one layer to the next.

```
runNet :: (KnownNat i, KnownNat o, Num a, Ord a)
       ⇒ Network i hs o a -- Neural network to fold
       → Vec i a          -- Input nodes
       → Vec o a          -- Nodes nodes
```

runNet folds the **LaryTransitionss** down to output nodes

Now we can run our network by moving data from one layer to the next.

```
runNet :: (KnownNat i, KnownNat o, Num a, Ord a)
       ⇒ Network i hs o a -- Neural network to fold
       → Vec i a          -- Input nodes
       → Vec o a          -- Nodes nodes

runNet (OutputLayer l) v = runLayer l v
```

runNet folds the **LaryTransitionss** down to output nodes

Now we can run our network by moving data from one layer to the next.

```
runNet :: (KnownNat i, KnownNat o, Num a, Ord a)
       ⇒ Network i hs o a -- Neural network to fold
       → Vec i a          -- Input nodes
       → Vec o a          -- Nodes nodes

runNet (OutputLayer l) v = runLayer l v

runNet (l :>> n) v = runNet n (runLayer l v)
```

Layers can be composed using `:>>`, only with matching layer sizes!

Let's suppose we have the following four layer neural network.

```
layer1 :: (Fractional a, Ord a) ⇒ Weights 2 3 a
layer2 :: (Fractional a, Ord a) ⇒ Weights 3 3 a
layer3 :: (Fractional a, Ord a) ⇒ Weights 3 2 a
layer4 :: (Fractional a, Ord a) ⇒ Weights 2 1 a
```

Layers can be composed using `:>>`, only with matching layer sizes!

Let's suppose we have the following four layer neural network.

```
layer1 :: (Fractional a, Ord a) ⇒ Weights 2 3 a
layer2 :: (Fractional a, Ord a) ⇒ Weights 3 3 a
layer3 :: (Fractional a, Ord a) ⇒ Weights 3 2 a
layer4 :: (Fractional a, Ord a) ⇒ Weights 2 1 a
```

Now combine them using our `Network` type.

```
network :: (Fractional a, Ord a)
        ⇒ Network 2 '[3, 3, 2] 1 a
network = layer1 :>> layer2 :>> layer3 :>> OutputLayer layer4
```

The type level numbers force us to make a network where the sizes of the output of one layer match the input to the next!

Synthesizing FPGA hardware

We can now synthesize what we have into something the FPGA can understand using `clash File.hs --verilog`.

```
topEntity :: Vec 2 (SFixed 7 25) -> Vec 1 (SFixed 7 25)
topEntity = runNet network
```

Synthesizing FPGA hardware

We can now synthesize what we have into something the FPGA can understand using `clash File.hs --verilog`.

```
topEntity :: Vec 2 (SFixed 7 25) -> Vec 1 (SFixed 7 25)
topEntity = runNet network
```

Note that

- We had to specify the specific number type we were using. Clash must know this to layout the hardware correctly.

Synthesizing FPGA hardware

We can now synthesize what we have into something the FPGA can understand using `clash File.hs --verilog`.

```
topEntity :: Vec 2 (SFixed 7 25) → Vec 1 (SFixed 7 25)
topEntity = runNet network
```

Note that

- We had to specify the specific number type we were using. Clash must know this to layout the hardware correctly.
- We had to specify which type of number we are using in order to synthesize the hardware.

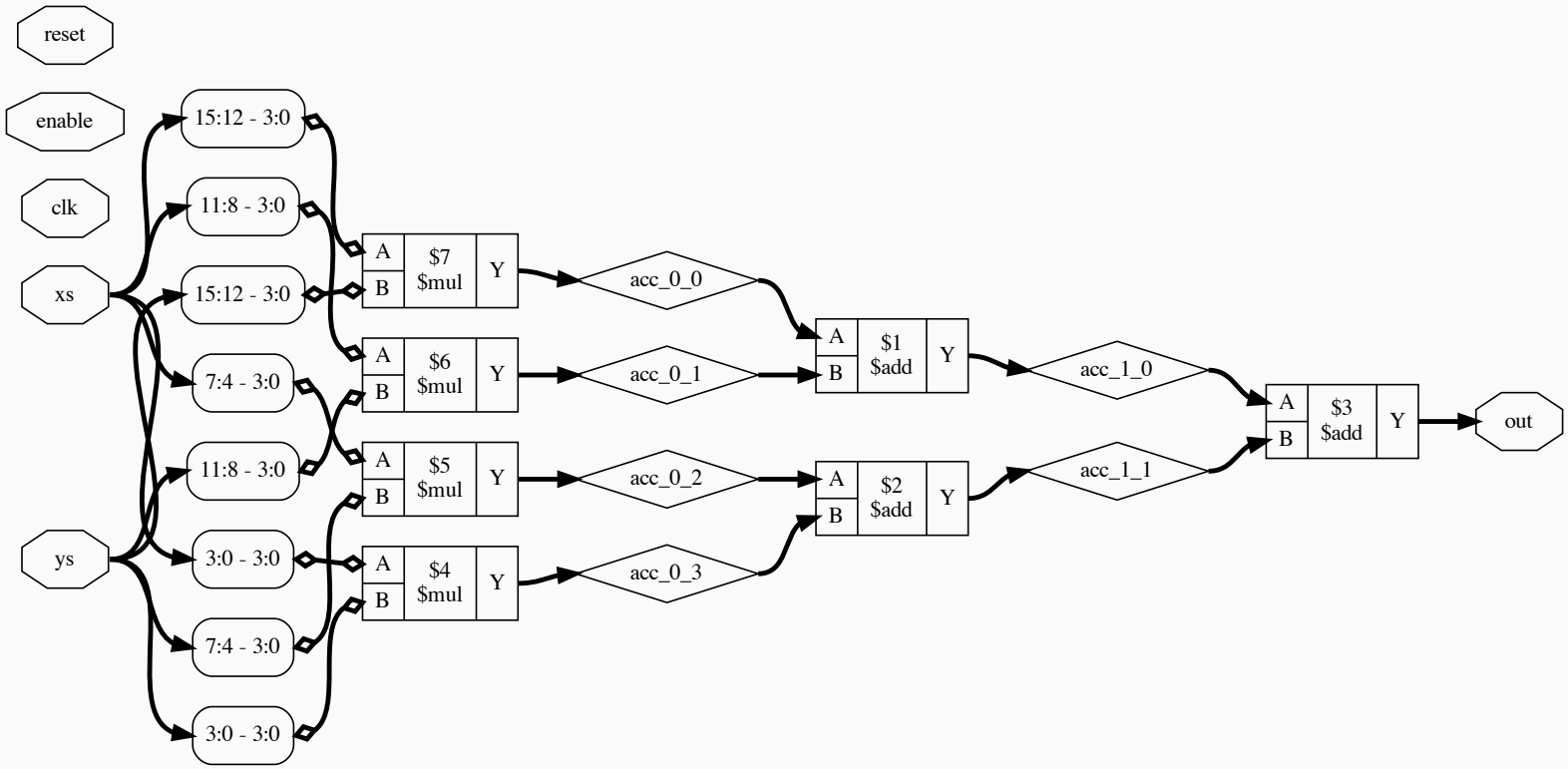
Meeting timing constraints

Experienced FPGA developers will notice that networks above a certain size cannot be synthesized.

Meeting timing constraints

Experienced FPGA developers will notice that networks above a certain size cannot be synthesized.

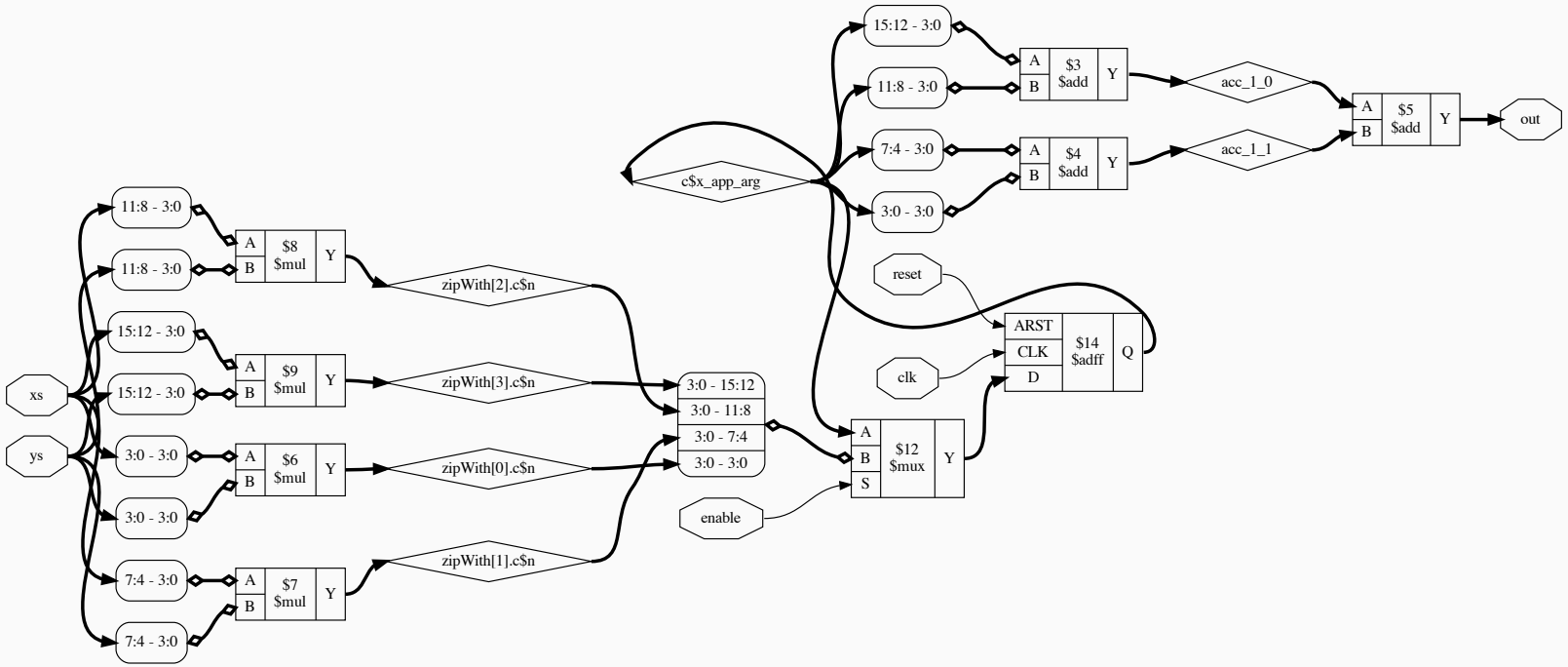
```
fold (+) (zipWith (*) xs ys)
```



Meeting timing constraints

But we can add a register between the zip and the fold to reduce the critical path.

```
fold (+) $ unbundle $ register (repeat 0) $ zipWith (*) <$> xs <*> *ys)
```



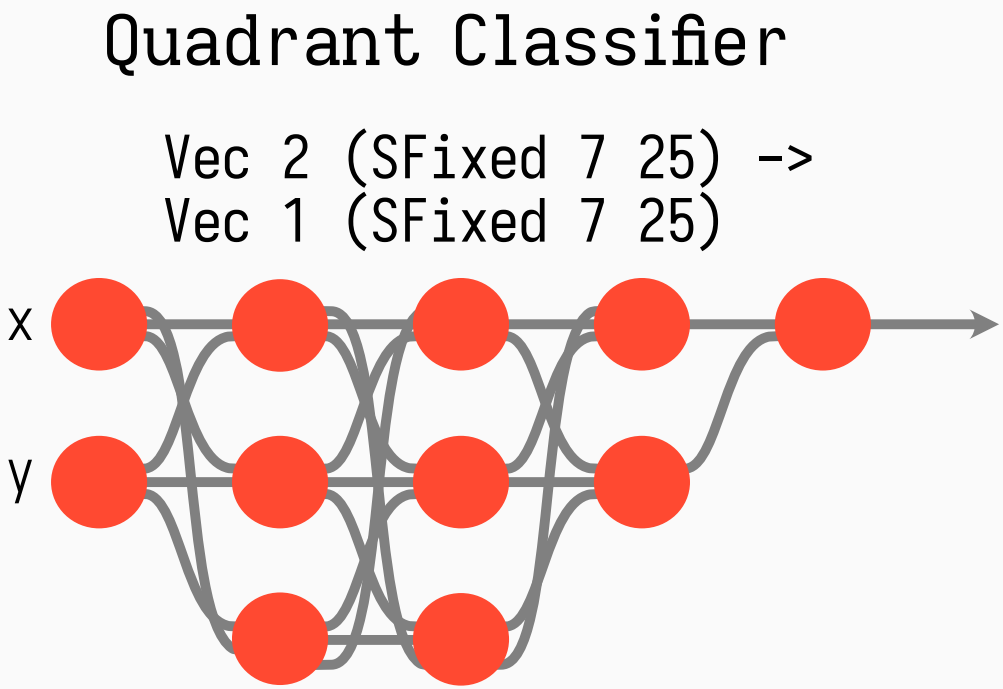
Other benefits to Clash

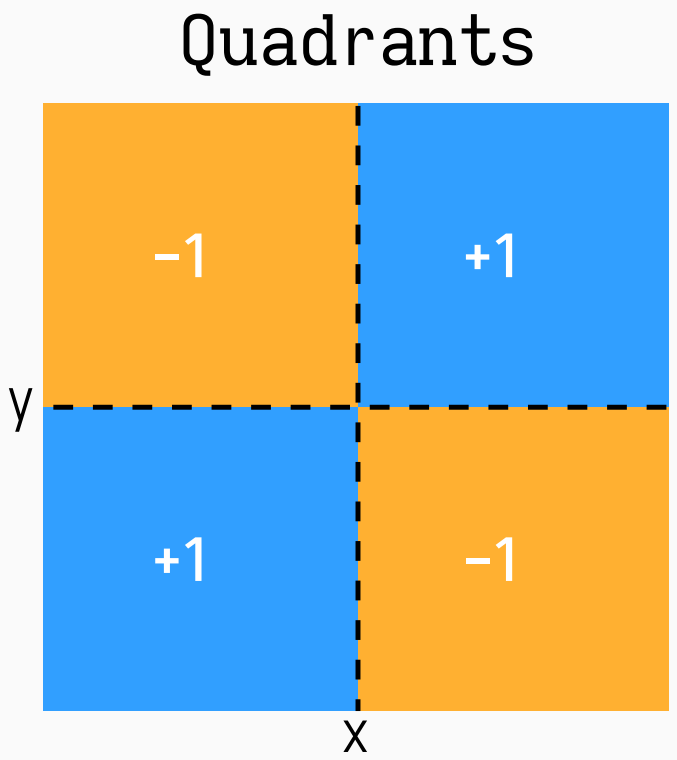
Since Clash is mostly just Haskell, you get a ton of benefits.

- Strong (dependently typed!) type system.
- Complicated FPGA state can be modeled with a Mealy machine, which is basically just `StateT`!
- `Applicative`, `Monad`, etc can be used to create convenient abstractions.
- The base functions can be tested like any other Haskell function (Quickcheck!)
- Resulting hardware can be simulated down to the picosecond using Clash and other tools.

Quadrant detection neural network that Rust will interact with

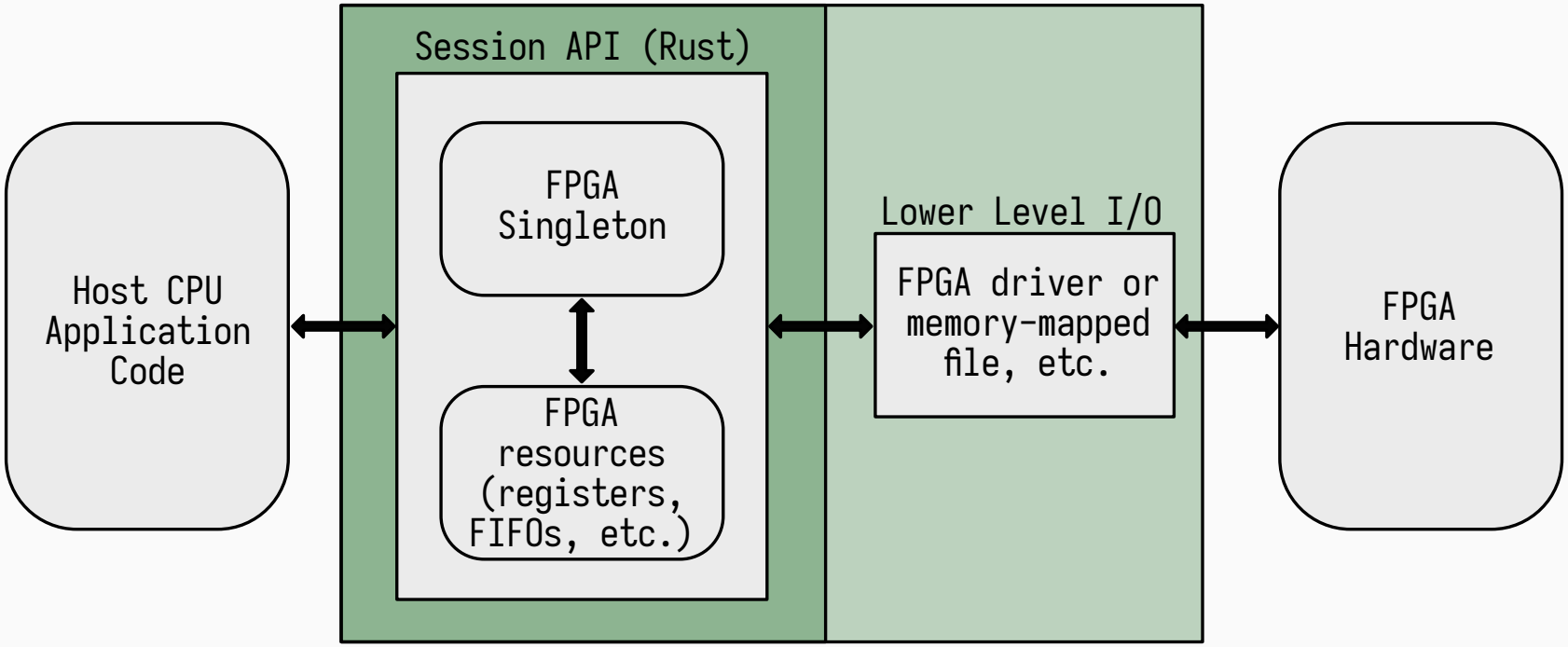
Neural network we actually have on the chip! It runs the whole network *in one clock cycle* at 50 MHz (20 ns!)





Rust program to interact with the FPGA

The big picture – safe control of our FPGA hardware



A common paradigm to interact with an FPGA is via a host CPU. We will build a user-facing **`Session`** API built on top of memory-mapped file I/O with the FPGA.

Key concepts

In our API, we will shoot for the following using Rust's type system:

- Encode and enforce HW invariants.
- Push as much as possible to compile-time checks.
- Maintain ergonomics!

Key concepts

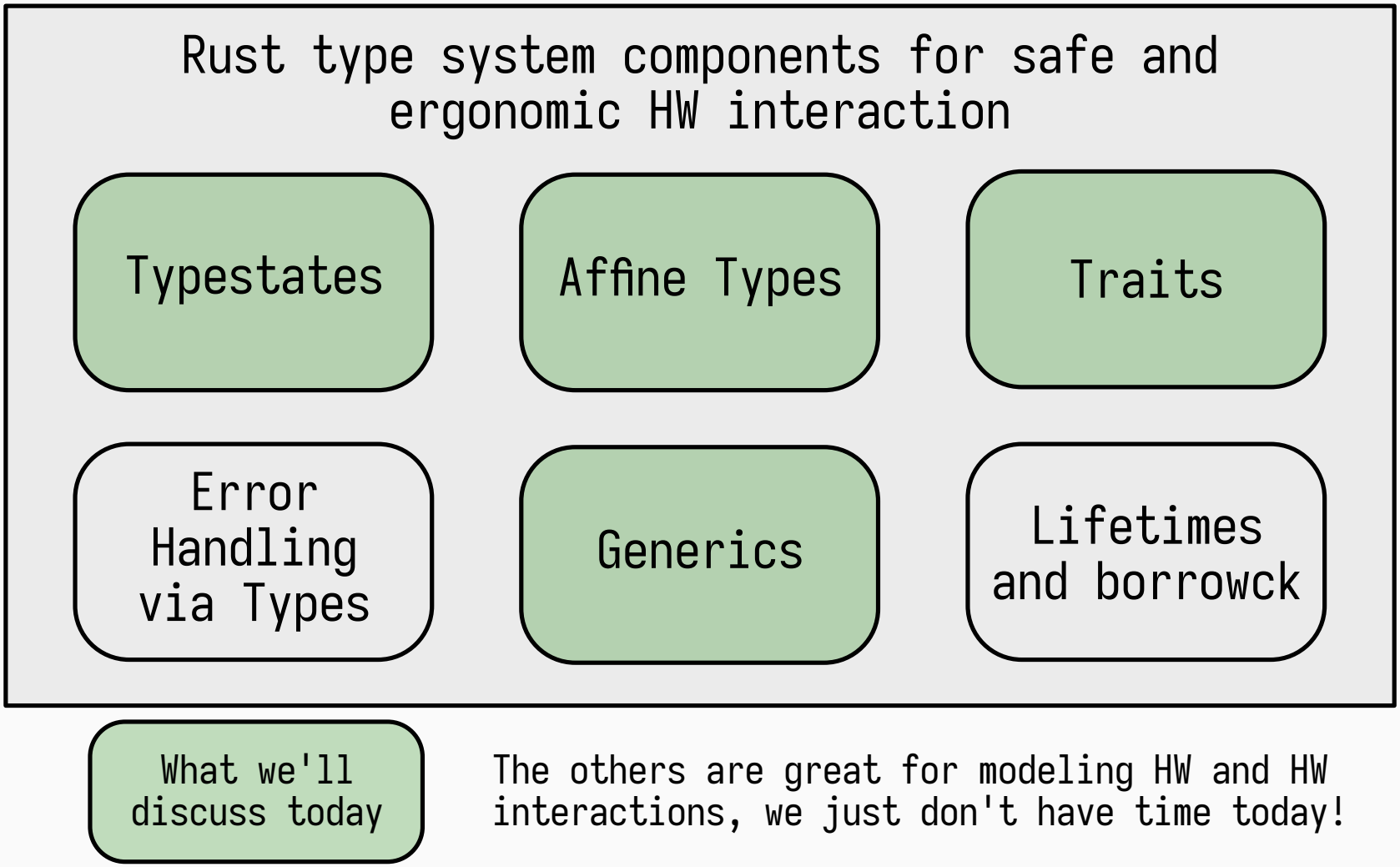
In the end, we want something that looks like this:

```
// Get our FPGA session.
let sesh = take_fpga_session();
// Write our 2D point to the FPGA for the computation.
sesh.write(&input_point, (x, y))?;
// Read back the classification from the net.
let quad_classification = sesh.read(&output_class)?;
```

and has compile-time guarantees so we can sleep well at night!

How do we get there?

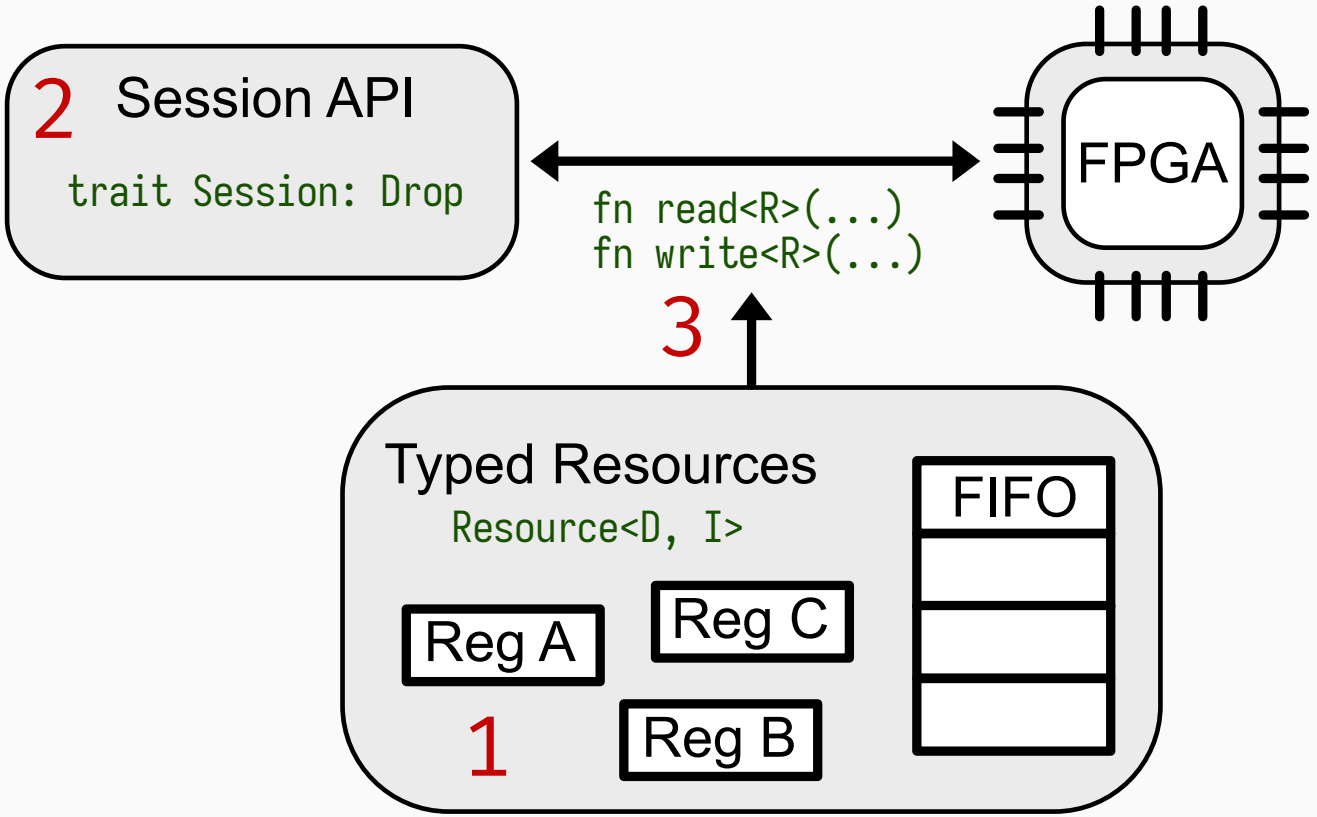
We will build a *Session* API using Rust:



The design

The major components of our Session API:

1. Express application-specific resources (*e.g.*, registers).
2. The session that wraps the FPGA and all interaction with it.
3. Link these together with the type system in a way that is ergonomic.



Traits are one of the anchors of the Rust type system. They allow you to define shared behavior and constraints for sets of types.

They are similar to interfaces and typeclasses in other languages.

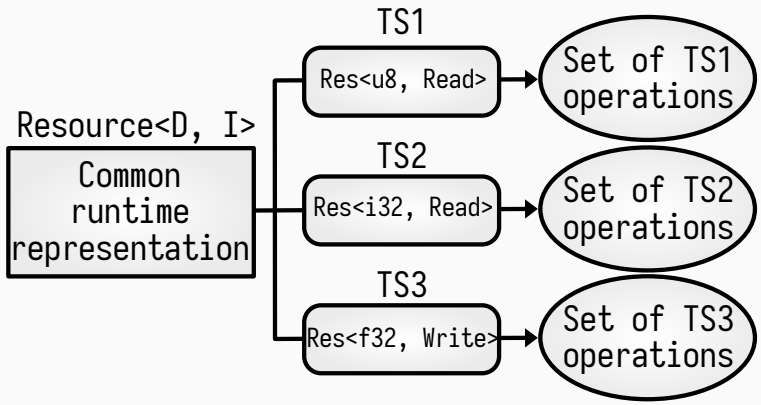
Encoding the data type/primitive with a trait

```rust
/// Trait for FPGA data types.
pub trait Data: Sized {
    fn from_le_bytes(bytes: &[u8]) -> FpgaApiResult<Self>;
    fn from_be_bytes(bytes: &[u8]) -> FpgaApiResult<Self>;
    fn to_le_bytes(self) -> Vec<u8>;
    fn to_be_bytes(self) -> Vec<u8>;
}
```

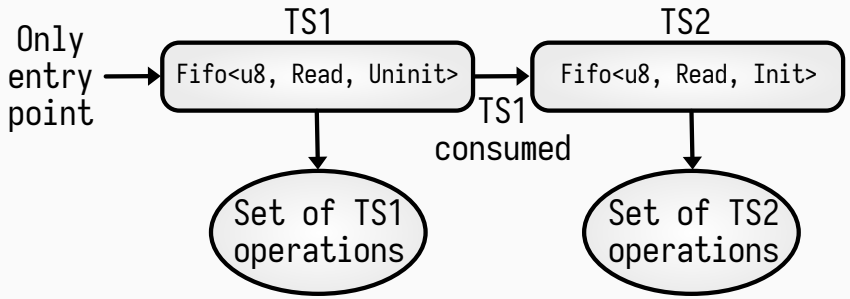
`Data` is implemented for primitives (*e.g.*, `u32`, `f32`, etc.). To plug into the `Session` API with a custom data type, just implement `Data`!

Controlling read/write with a typestate

Typestates are a great way to encode invariants in the type system. There is no runtime cost and if it compiles, the encoded invariants are guaranteed.



Orthogonal typestates



Type-level finite state machine

Note: The common **Builder** pattern in Rust is a form of the latter.

Encoding allowed resource I/O with typestates in Rust

```rust
/// Typestate pattern via empty marker trait.
pub trait IOState {}
/// Uninhabitable `enum` for first typestate.
pub enum ReadOnly {}
impl IOState for ReadOnly {}
/// Uninhabitable `enum` for second typestate.
pub enum ReadWrite {}
impl IOState for ReadWrite {}
```

Putting these together: expressing any FPGA resource

```rust
pub struct Resource<D: Data, I: IOState> {
    name: &'static str,
    offset: usize,
    _ty: PhantomData<D>,
    _st: PhantomData<I>,
}
```

For any resource, only a `name` and a (byte) `offset` are reified at runtime. The `D` and `I` typestates determine the available operations associated with the FPGA (through the `Session`).

What does this give us?

In application code:

1. (In a type sense) we can't send the wrong bytes to the FPGA or interpret incoming bytes incorrectly.
2. We can't mutate a read-only FPGA resource.

```
let sesh = take_fpga_session();
let input_point = Resource::<(I7F25, I7F25), ReadWrite>::new(...);
let output_class = Resource::<I7F25, ReadOnly>::new(...);
// -- snip -- Note: `I7F25` is a "fixed point" numeric type.
let (x, y): (f32, f32) = (1.3, -2.7);
sesh.write(&input_point, (x, y))?;                  // Comp fail (type).
let v: u32 = sesh.read(&output_class)?;             // Comp fail (type).
sesh.write(&output_class, I7F25::from_num(1.0))?; // Comp fail (R-only).
```

Now, to the **`Session`** type

The opaque `Session` type represents the FPGA and our interaction with it.

We will encode the singular nature of the HW and the importance of maintaining appropriate state with the help of the type system.

Encode **Session** HW invariant: singleton

We can't let our devs arbitrarily spawn up or duplicate sessions (there's only 1 piece of HW). We use Rust's version of the singleton pattern for this:

```
struct Fpga(Option<MmapSesh>);
impl Fpga {
    fn take(&mut self) -> MmapSesh {
        let sesh = self.0.take();
        sesh.expect("Forbidden to create more than one FPGA session!")
    }
}
pub fn take_fpga_session() -> MmapSesh {
    POINT_NN_FPGA.lock().unwrap().take()
}
// -- snip -- in application code
let mut sesh = take_fpga_session();
```

Encode **`Session`** HW invariant: initialization and finalization

Rust's RAII and affine type system allows us to ensure FPGA/HW state invariants:

- Can only create a `Session` through constructor that performs proper initialization.
- Must implement 'Drop' to finalize state of the FPGA (and any associated HW) when we're done.
- You cannot then forget to 'Drop' – in happy or sad code paths!

Encode **Session** HW invariant: finalization

```
pub trait Session: Drop { // Note **must** implement `Drop`.
// -- snip --
impl Session for MmapSesh {
// -- snip --
impl Drop for MmapSesh {
    fn drop(&mut self) { // You never have to manually call this.
        // Enforce critical FPGA/HW invariants for "final" state.
        // -- snip --
    }
}
```

What does this give us?

With **Drop** implemented, we cannot "forget" to cleanup the FPGA and associated resources:

```
fn main() {
    let sesh = take_fpga_session();
    // -- snip -- do stuff with the FPGA.
    risky_function().expect("Uh oh, hit a panic!");
    // -- snip -- more stuff
    println!("Done!");
}
```

Whether we **panic** or not, our session will be **Drop**ped and our FPGA/HW will be in the appropriate state.

Bringing it all together in the session: helper traits

```rust
pub trait Readable { // A readable FPGA resource.
    type Value: Data;
    fn byte_offset(&self) -> usize;
    fn size_in_bytes(&self) -> usize;
}
pub trait Writable { // A writable FPGA resource.
    type Value: Data;
    fn byte_offset(&self) -> usize;
    fn size_in_bytes(&self) -> usize;
}
impl<D: Data> Readable for Resource<D, ReadOnly> {  // ...
impl<D: Data> Readable for Resource<D, ReadWrite> { // ...
impl<D: Data> Writable for Resource<D, ReadWrite> { // ...
```

Bringing it all together: the **Session** trait

```
pub trait Session: Drop {

    fn read<R: Readable>(
        &self,
        resource: &R
    ) -> FpgaApiResult<R::Value>;

    fn write<R: Writable>(
        &mut self,
        resource: &R,
        val: R::Value
    ) -> FpgaApiResult<()>

}
```

- We have a simple and ergonomic API with just **read** and **write** but have wired up traits, constraints, and typestates so that a lot is enforced at compile-time.
- The Rust compiler will *monomorphize* concrete **read** and **write** functions based on what resources are defined and used in a given application.

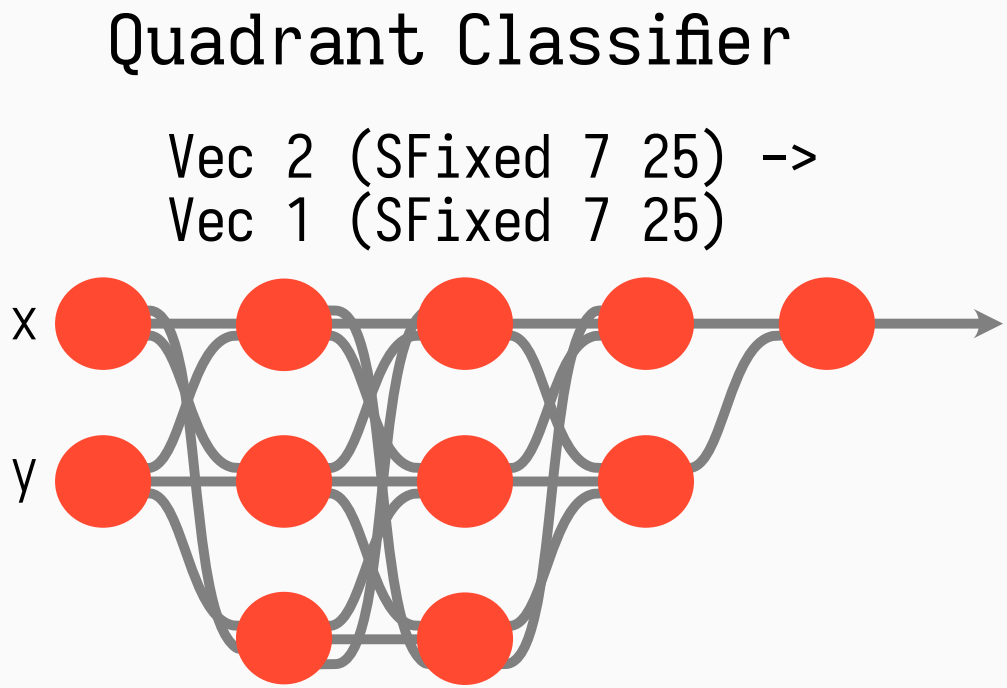
Implementing **Session** for FPGA memory-mapped file I/O

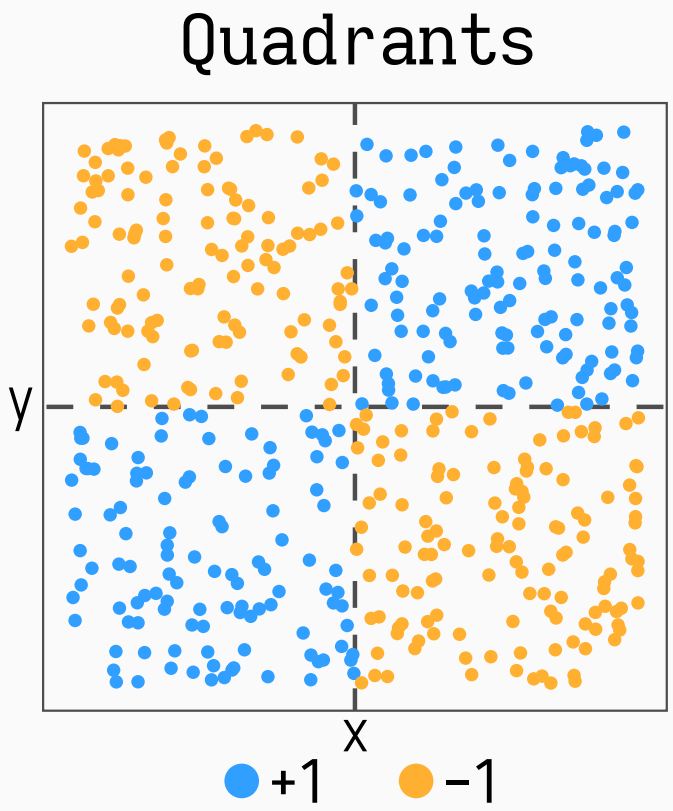
```
impl Session for MmapSesh {
    fn read<R: Readable>(
        &self,
        resource: &R
    ) -> FpgaApiResult<R::Value>
    {
        let start = resource.byte_offset();
        let stop = start + resource.size_in_bytes();
        let slc = &self.mmap[start..stop];
        R::Value::from_le_bytes(slc)
    }
    // -- snip --
```

Implementing **Session** for FPGA memory-mapped file I/O

```
// -- snip --
fn write<R: Writable>(
    &mut self,
    resource: &R,
    val: R::Value
) -> FpgaApiResult<()>
{
    let start = resource.byte_offset();
    let stop = start + resource.size_in_bytes();
    self.mmap[start..stop].copy_from_slice(
        val.to_le_bytes().as_slice()
    );
    Ok(())
}
```

Point quadrant classifier inference: visualization of solution





Point quadrant classifier inference: the code

```
fn run() -> FpgaApiResult<()> {
    // Get the FPGA singleton. Better not try this more than once!
    let mut sesh = take_fpga_session();
    // Define the resources.
    let input_point = Resource::<(I7F25, I7F25), ReadWrite>::new(
        "Input Point Registers(X, Y)",
        POINT_NN_INPUT_VECTOR_OFFSET,
    );
    let output_class = Resource::<I7F25, ReadOnly>::new(
        "Output Classification Register",
        POINT_NN_OUTPUT_CLASS_OFFSET,
    );
    let (pos_x, neg_x) = (I7F25::from_num(1.5), I7F25::from_num(-1.5));
    let (pos_y, neg_y) = (I7F25::from_num(2.5), I7F25::from_num(-2.5));
    // -- snip --
```

Point quadrant classifier inference: the code

```
    // -- snip --
    // Quadrant 1.
    sesh.write(&input_point, (pos_x, pos_y))?;
    let q1_actual = sesh.read(&output_class)?;
    let q1_expected = I7F25::from_num(1.0);
    // -- snip --
    // Quadrant 2.
    sesh.write(&input_point, (neg_x, pos_y))?;
    let q2_actual = sesh.read(&output_class)?;
    let q2_expected = I7F25::from_num(-1.0);
    // -- snip --
}
```

Point quadrant classifier inference: running on the board

```
de10-nano:~/linux_examples/rust_examples$ sudo ./classify_point_nn_example
Password:

Quadrant 1
==========

Writing (1.5, 2.5) to Input Points at byte offset 0
Reading result from Output Classification Register at byte offset 8

Actual output:   1
Expected output: 1

Quadrant 2
==========

Writing (-1.5, 2.5) to Input Points at byte offset 0
Reading result from Output Classification Register at byte offset 8

Actual output:   -1
Expected output: -1
```

Conclusions

Summary:

- FPGAs allow you to map your HW to the algorithm. Neat!
- Clash provides a sane way to write FPGA programs.
- Rust provides a sane way to manage interactions with FPGA/HW.
- We demonstrated this in a simple point classifier ML application.

Takeaway: Type systems and architectures that push guarantees to compile time are really great for modeling and interacting with HW.

Big thanks! to our friend, colleague, and FPGA expert **Blayne Kettlewell** for his help on this project.

Questions?

Our slides and the code to do this yourself can be found at:

github.com/ryanorendorff/sbtb-2020-type-safe-fpga