FPGAs for Programmers

Frans Skarman

```
fn add_three(a, b, c) {
    return a + b + c;
}
```

```
fn add_three(a, b, c) {
    return a + b + c;
}
```

```
load r1, *a
load r2, *b
add r3, r1, r2
load r4, *c
add r4, r3, r4
ret r4
```

```
fn add_three(a, b, c) {
    return a + b + c;
}
```

```
load r1, *a
load r2, *b
add r3, r1, r2
load r4, *c
add r4, r3, r4
ret r4
```

```
fn add_three(a, b, c) {
    return a + b + c;
}
```

```
load r1, *a
load r2, *b
add r3, r1, r2
load r4, *c
add r4, r3, r4
ret r4
```

```
fn add_three(a, b, c) {
    return a + b + c;
}
```

```
load r1, *a
load r2, *b
add r3, r1, r2
load r4, *c
add r4, r3, r4
ret r4
```

```
fn add_three(a, b, c) {
    return a + b + c;
}
```

```
load r1, *a
load r2, *b
add r3, r1, r2
load r4, *c
add r4, r3, r4
ret r4
```

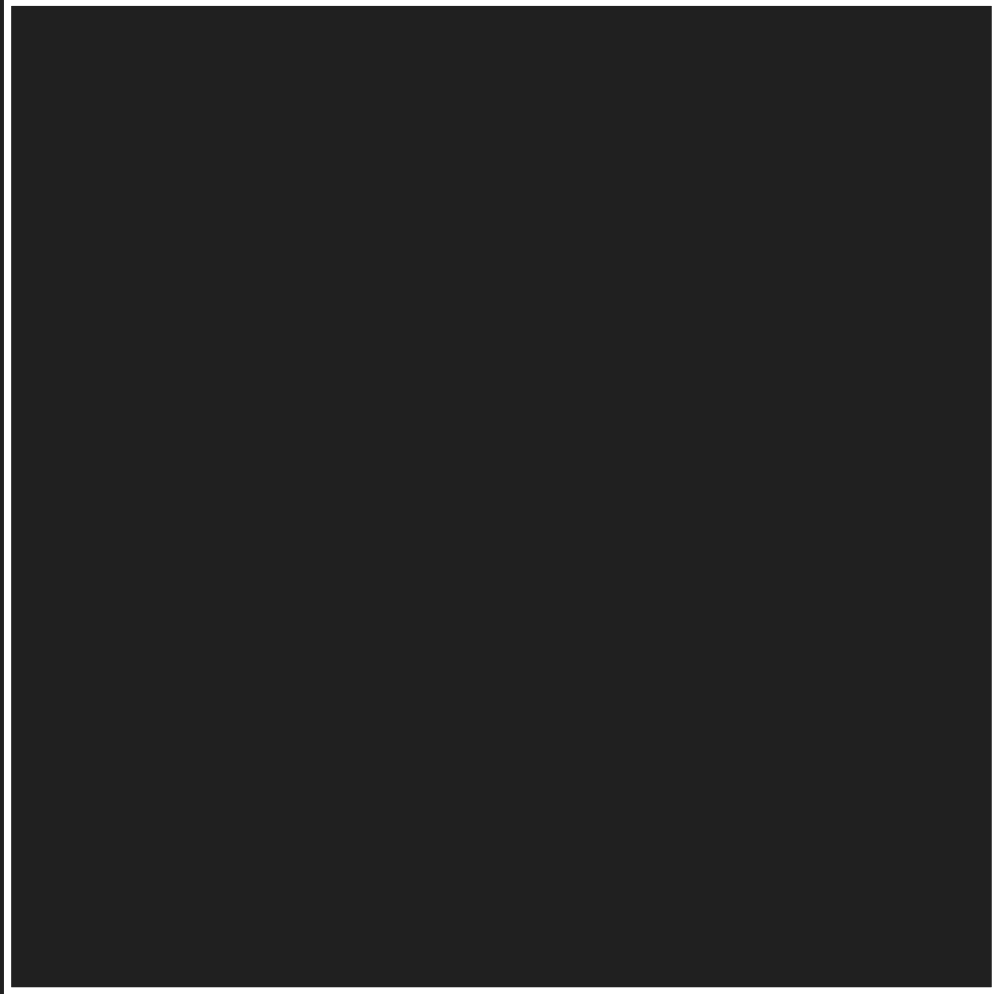
```
fn add_three(a, b, c) {
    return a + b + c;
}
```

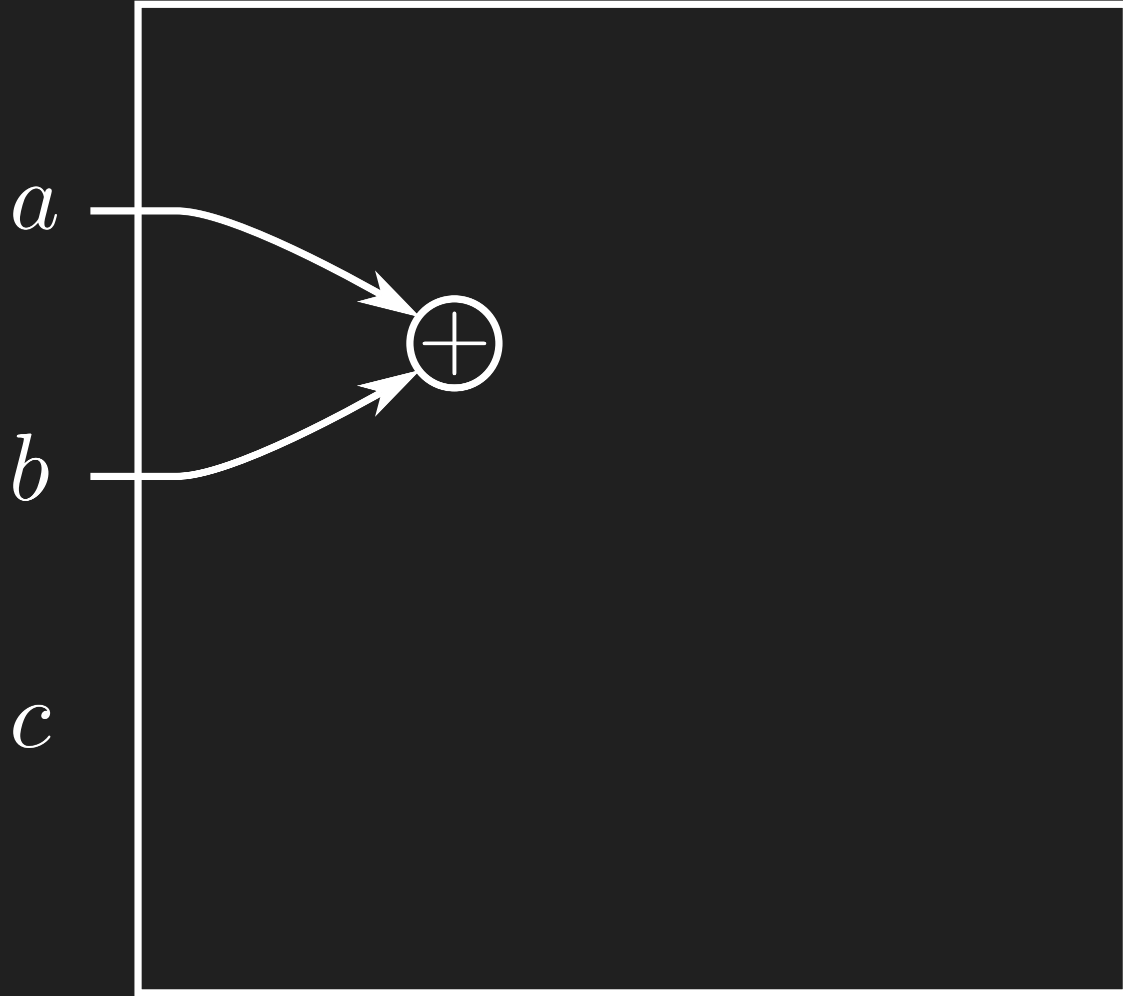
```
load r1, *a
load r2, *b
add r3, r1, r2
load r4, *c
add r4, r3, r4
ret r4
```

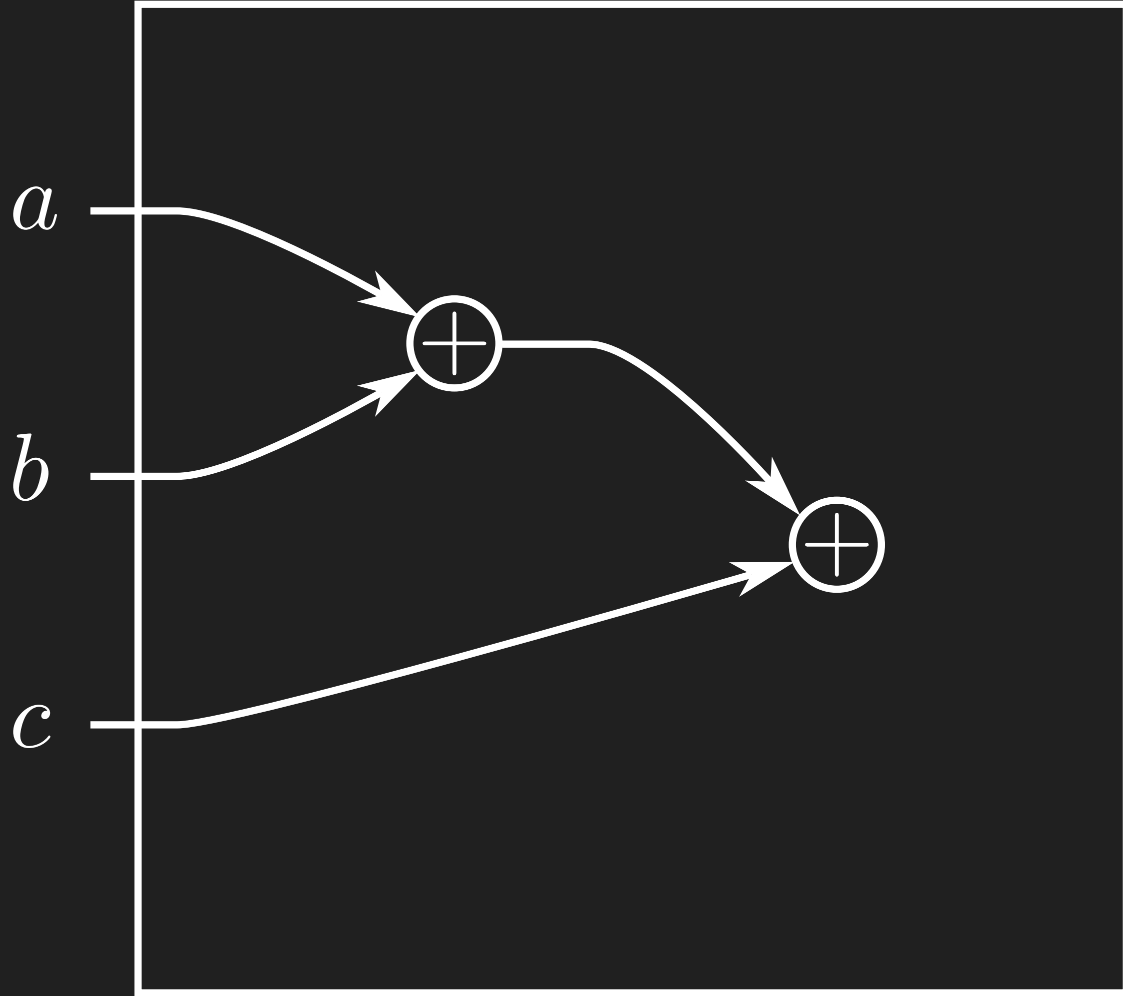
a

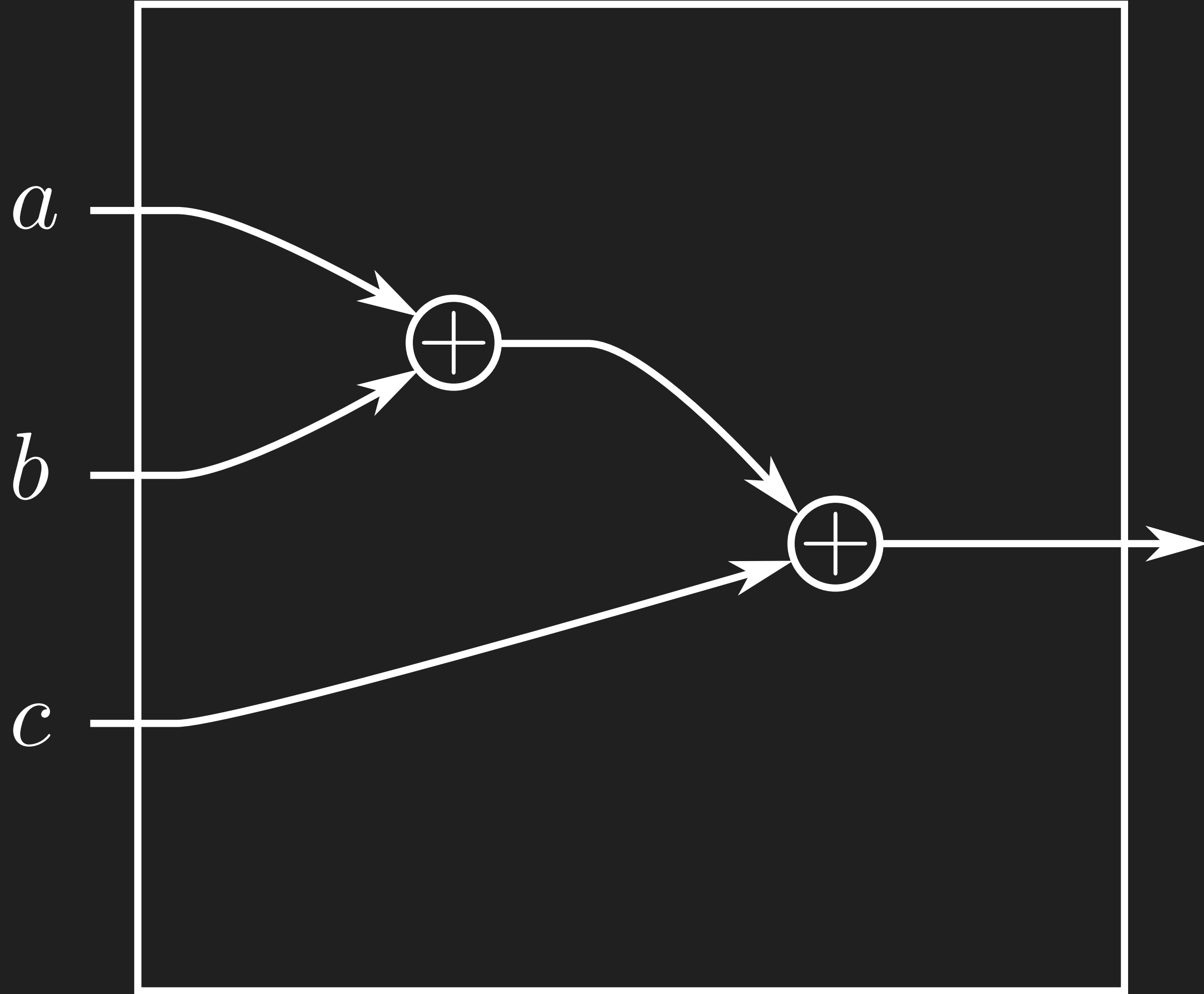
b

c

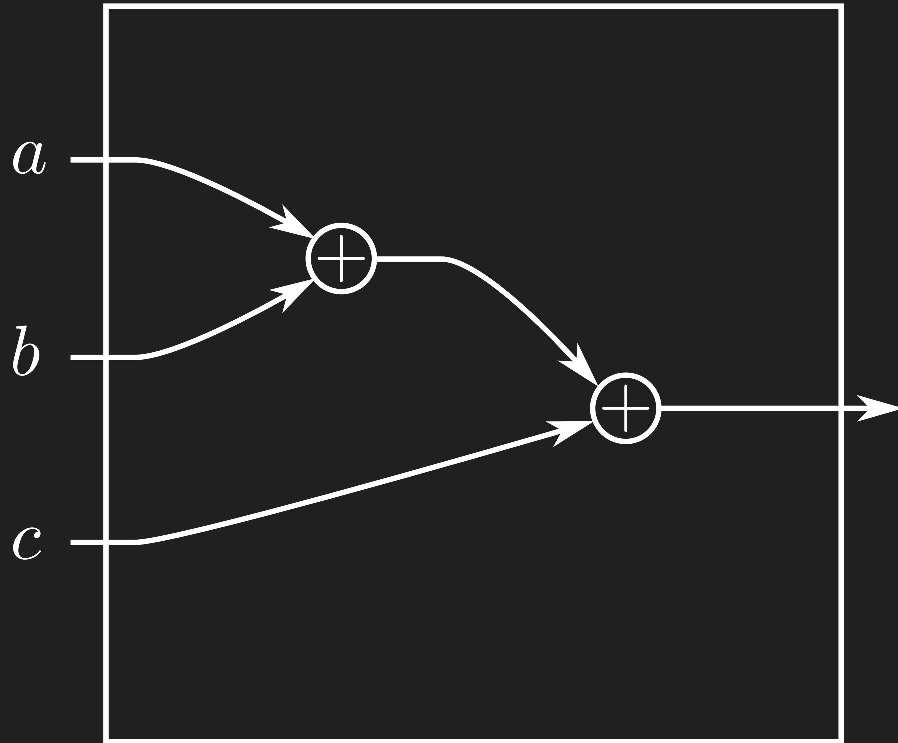



a
b
c
$\oplus$


a
b
c

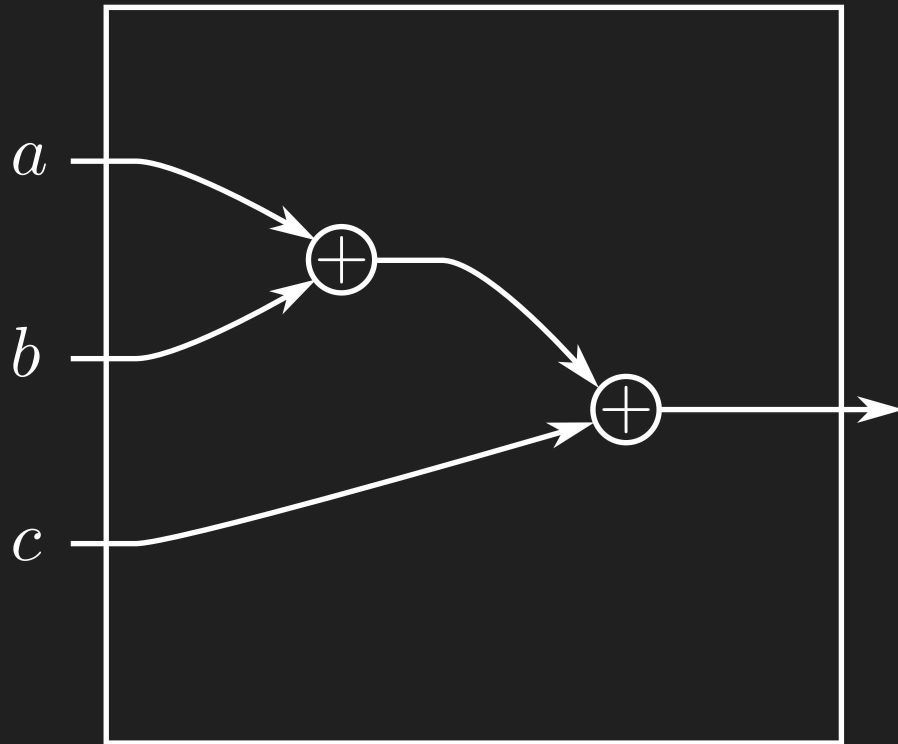

a
b
c

Why FPGA?



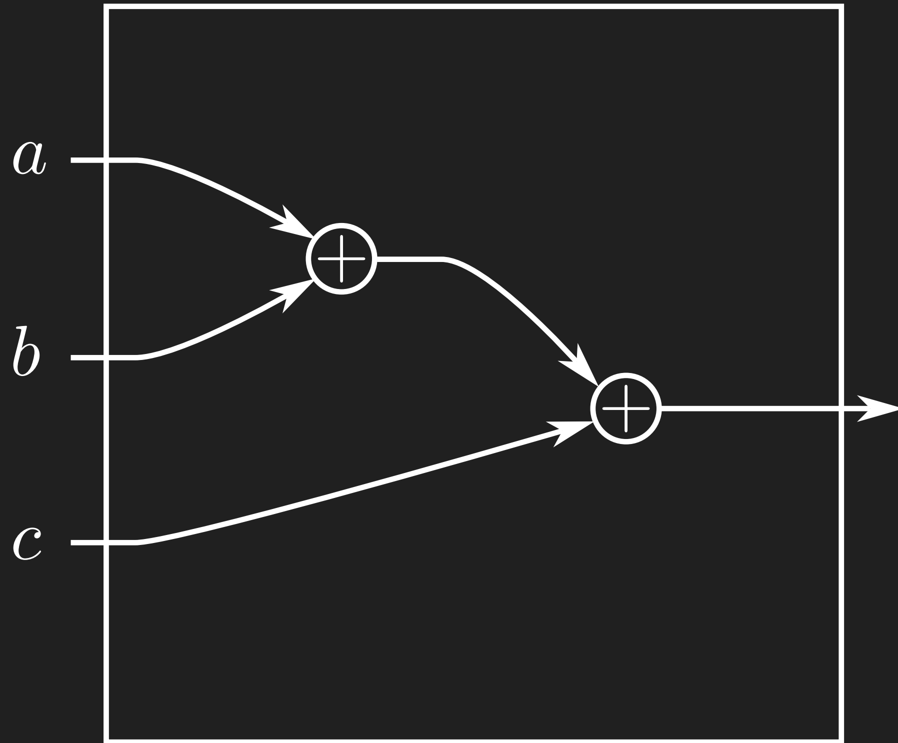
- Everything works in parallel

Why FPGA?



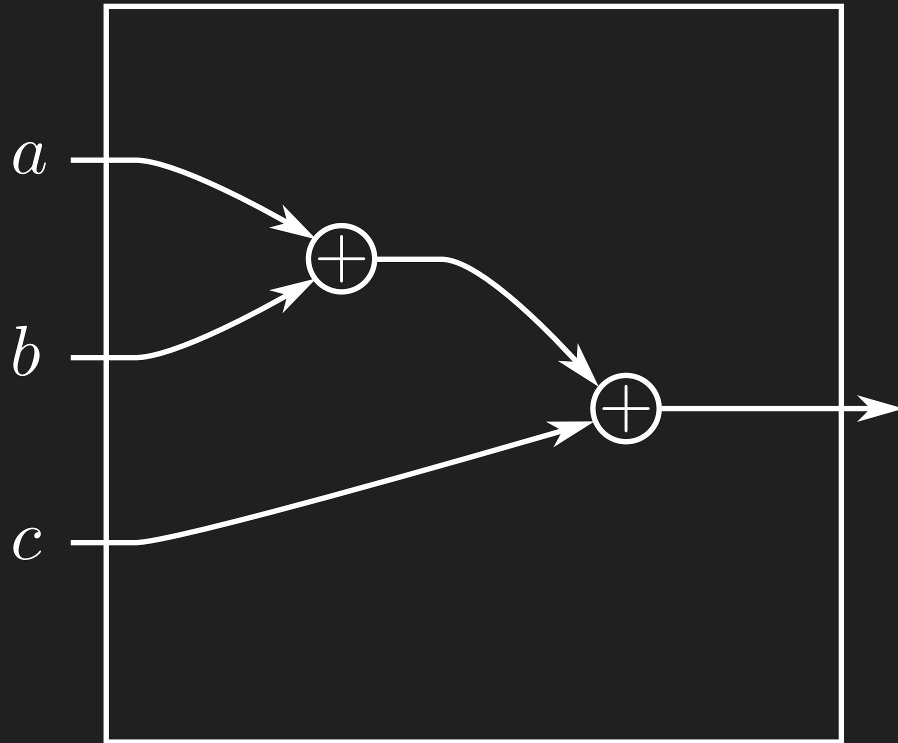
- Everything works in parallel
- The latency is known to the clock cycle

Why FPGA?



- Everything works in parallel
- The latency is known to the clock cycle
- Easy interaction with external hardware

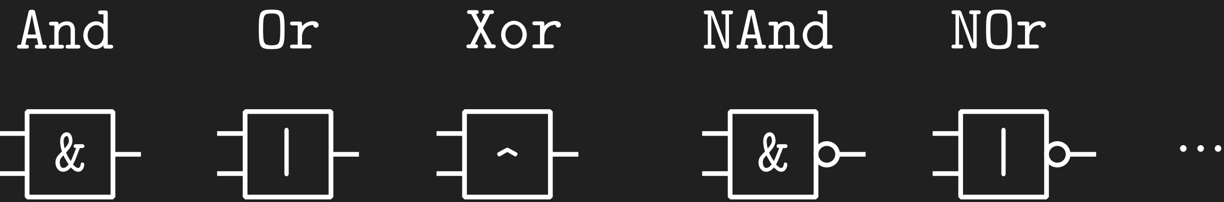
Why FPGA?



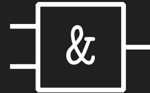
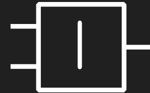
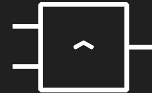
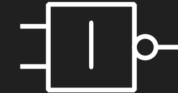
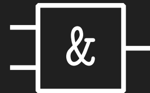
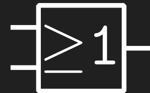
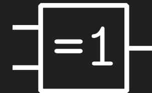
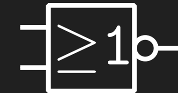
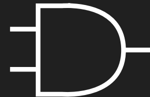
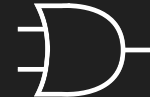
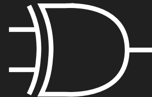
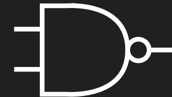
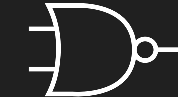
- Everything works in parallel
- The latency is known to the clock cycle
- Easy interaction with external hardware
- **It’s fun!**

Some Hardware Primitives

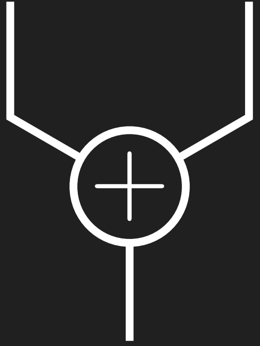
Boolean Logic

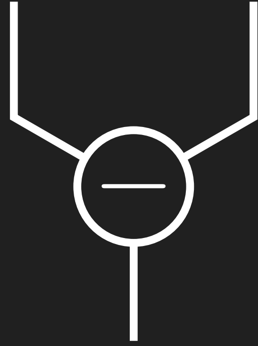


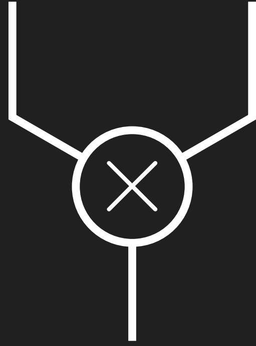
Boolean Logic

	And	Or	Xor	NAnd	NOr	
Programming:						…
EU Hardware:						…
’Murican:						…

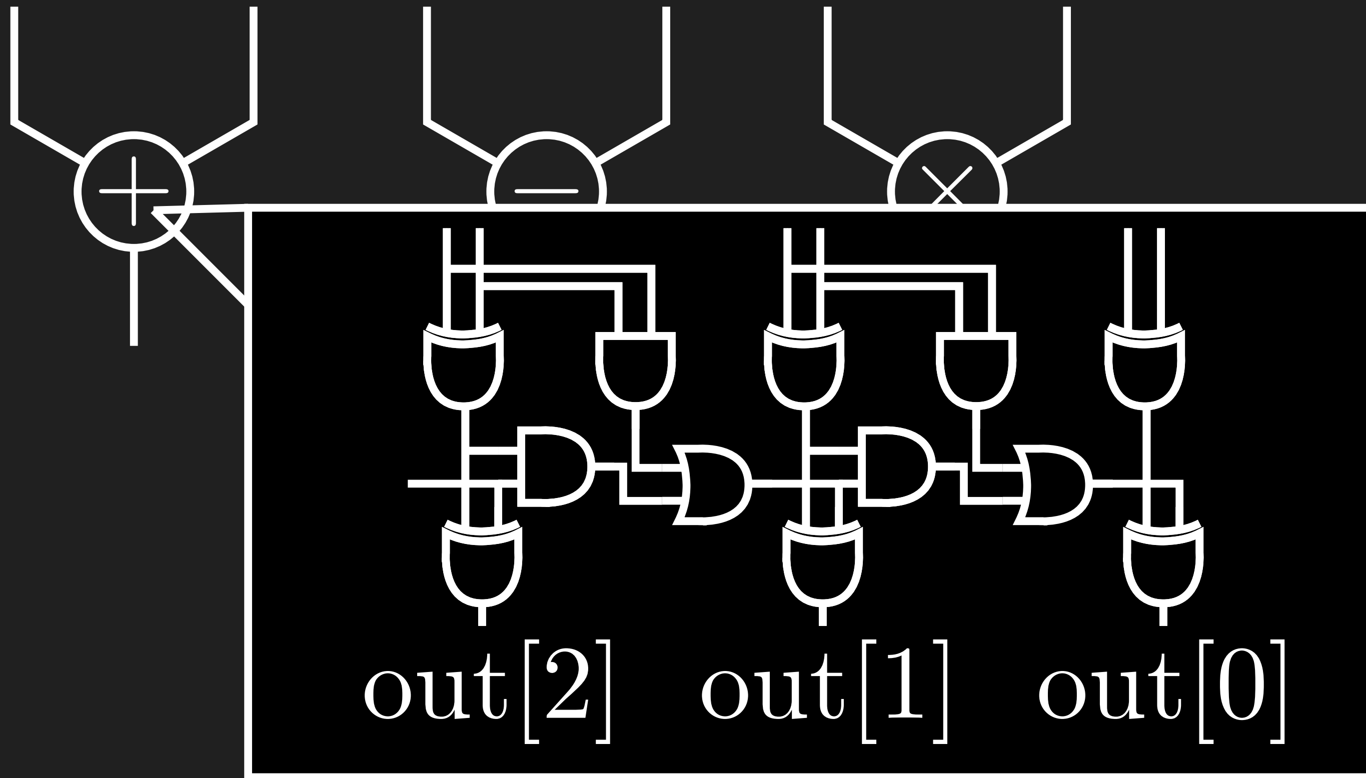
Math



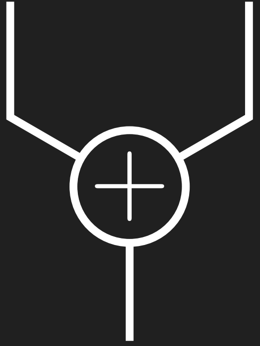


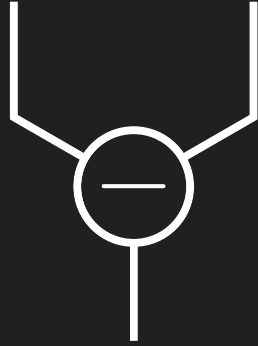


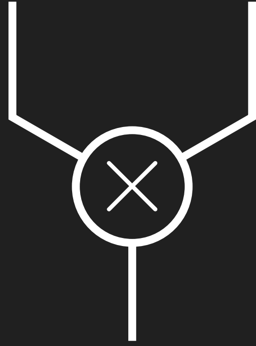
Math



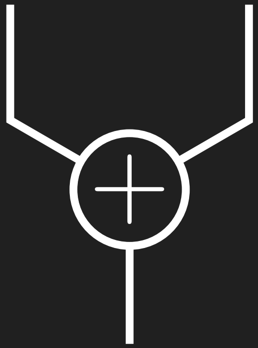
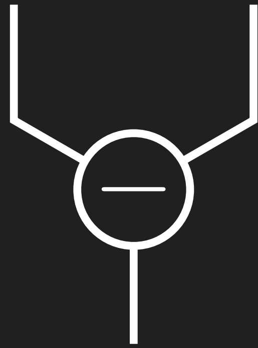
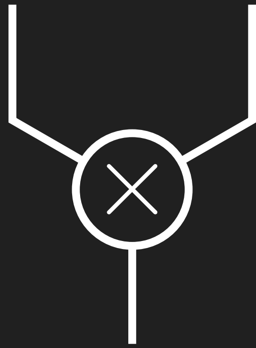
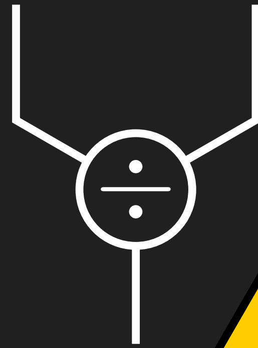
Math



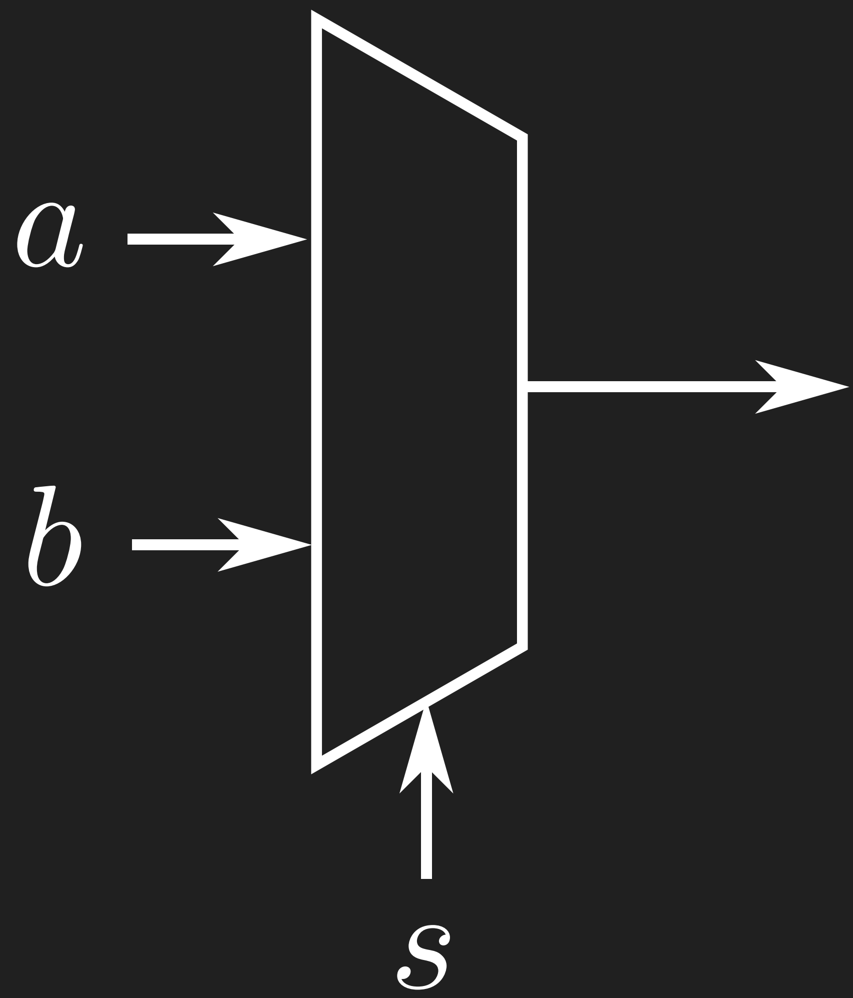


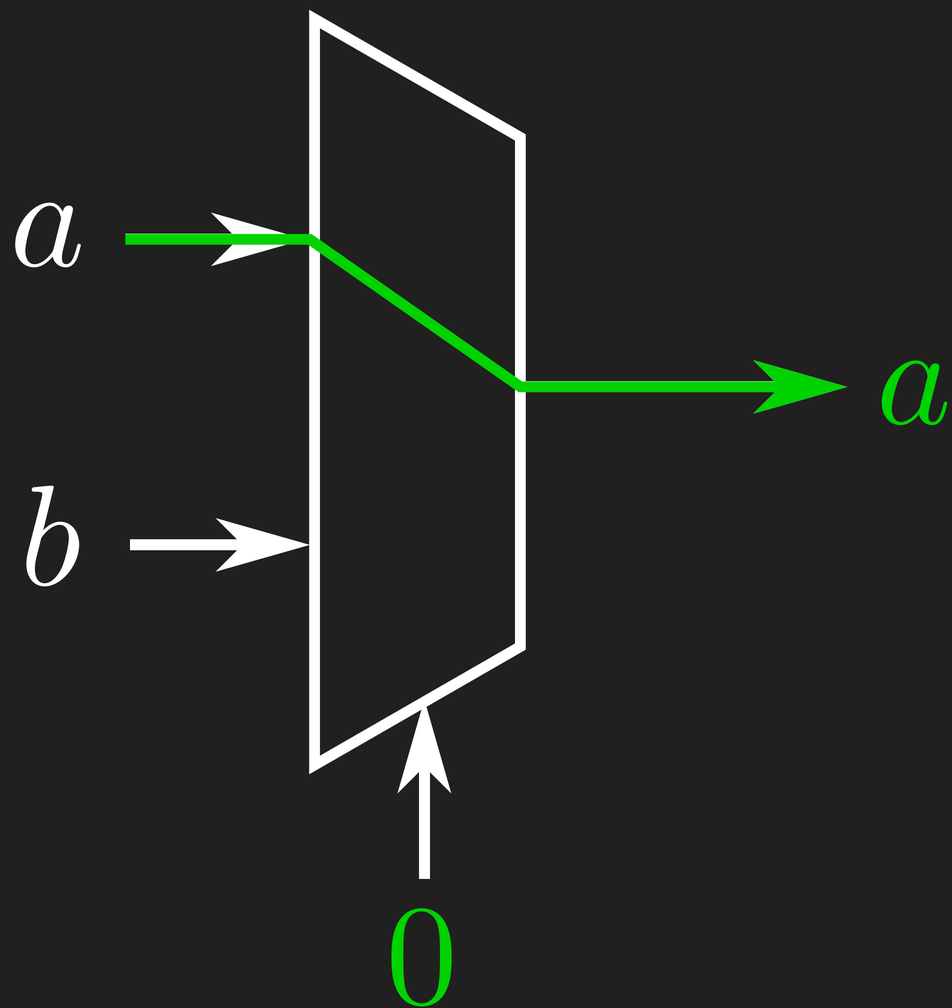


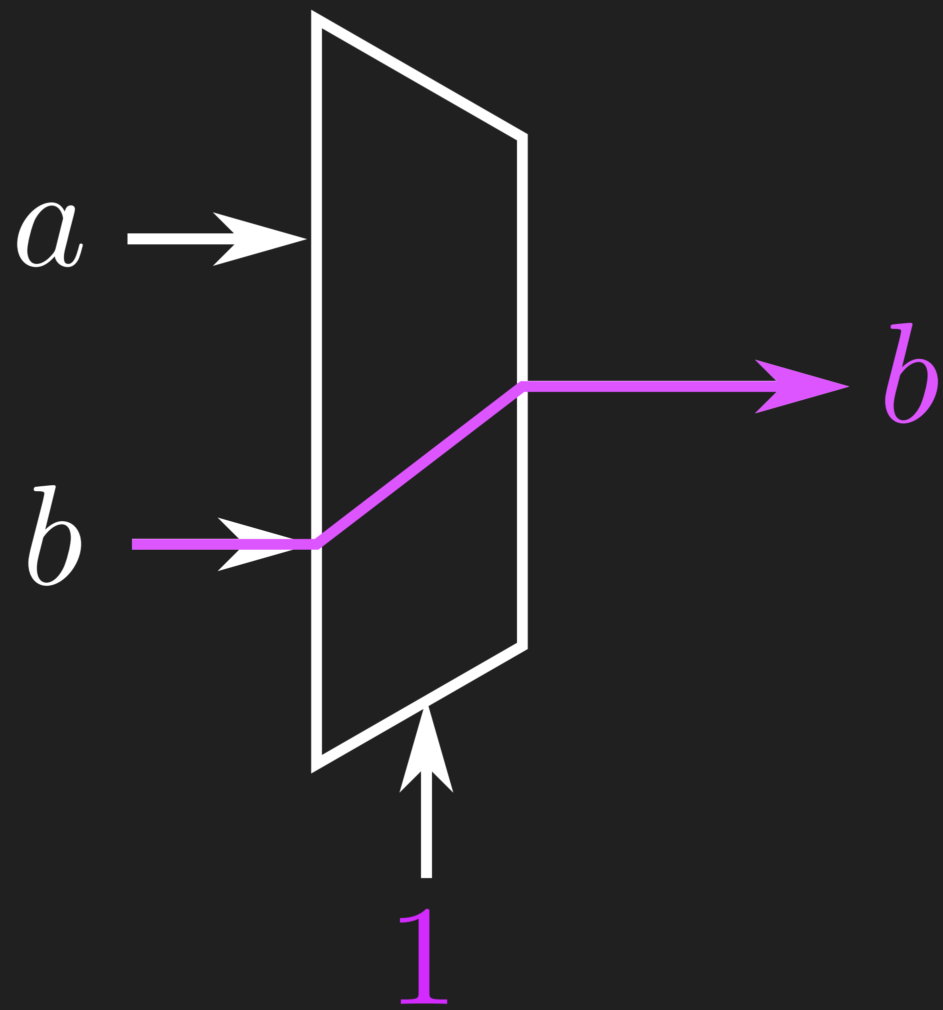
Math

Expensive!


a
b
s


a
b
a
0


a
b
b
1

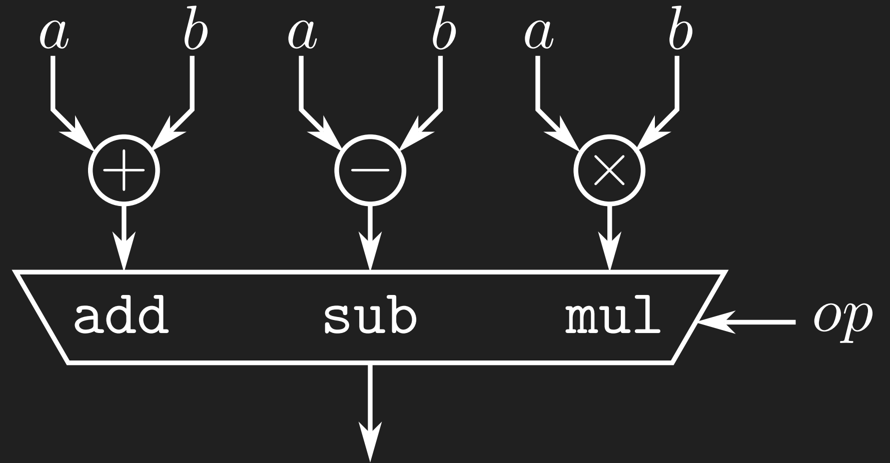
```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```

```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```

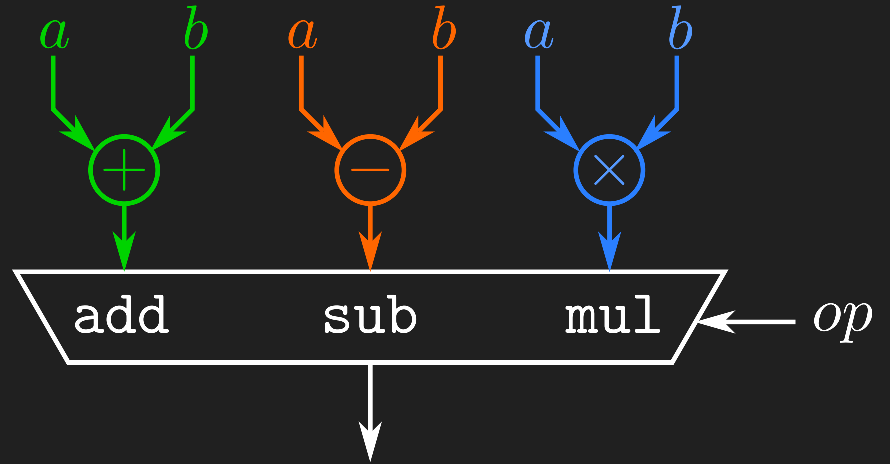
```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```

```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```

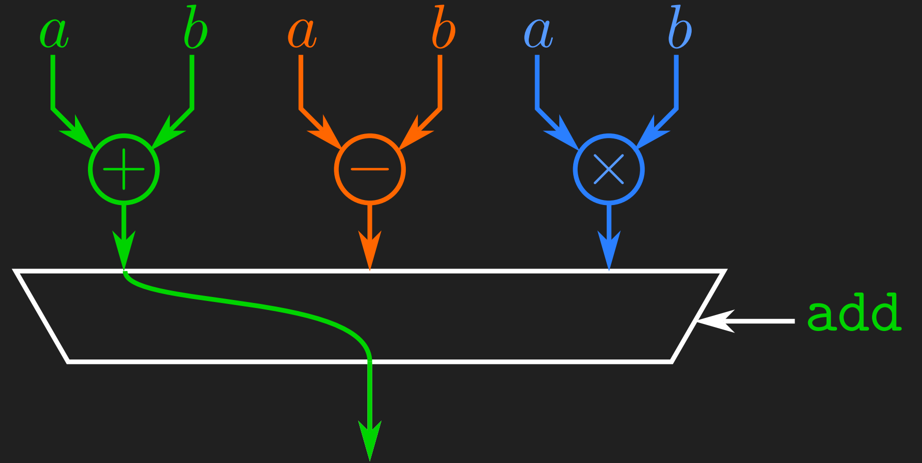
```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```



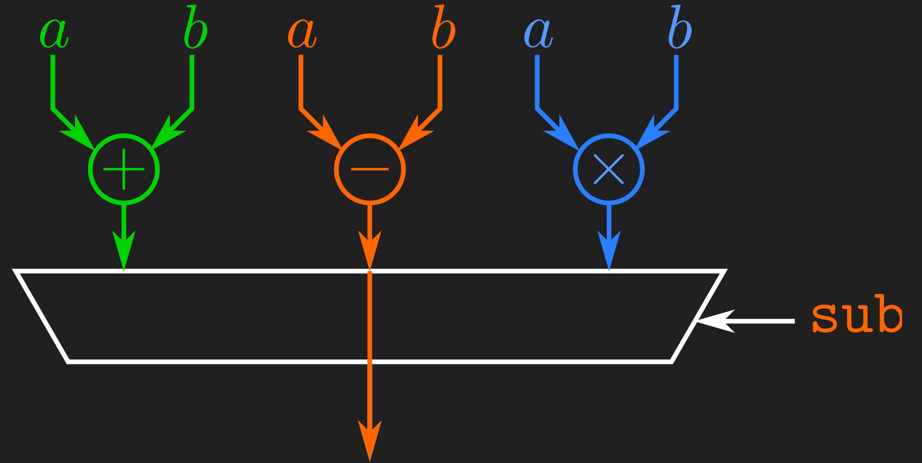
```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```



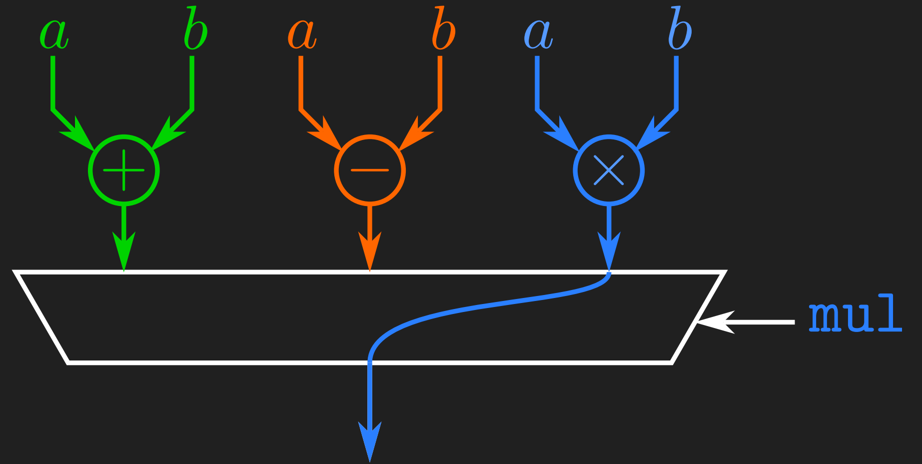
```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```



```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```



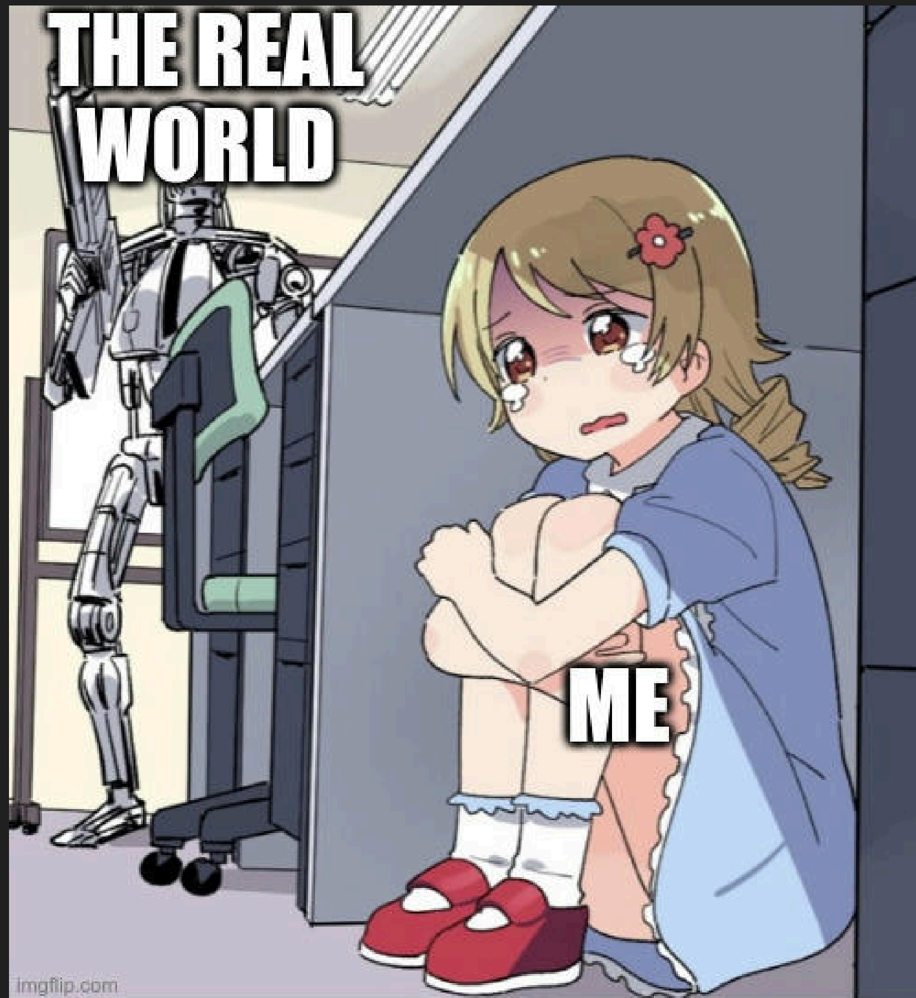
```
enum Op {
  Add, Sub, Mul
}

fn alu(op: Op, a: int<32>, b: int<32>)
  -> int<32>
{
  match op {
    Op::Add => a + b,
    Op::Sub => a - b,
    Op::Mul => a * b
  }
}
```



Recap so Far

- Hardware is physical primitives inside a chip
- Hardware description selects components and how to connect them
- Programming looks similar
- **But:** Everything is done in parallel

Dealing with state


THE REAL WORLD
ME
imgflip.com

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr(
  clk: clock, rst: bool,
  a: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr_max(
  clk: clock, rst: bool,
  a: int<20>
  max: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) = trunc(a + sum);
  sum
}
```

Counter

```
entity cntr_max(
  clk: clock, rst: bool,
  a: int<20>
  max: int<20>
) -> int<20> {
  reg(clk) sum reset(rst: 0) =
    if sum == max {
      0
    } else {
      trunc(sum+1)
    };
  sum
}
```

Cascaded counters

```
entity main(clk: clock, rst: bool) -> int<20> {
    let max = 4;
    let fast_count = inst cntr_max(clk, rst, 1, max);

    let tick = fast_count == max;

    let slow_count = inst cntr(clk, rst, if tick {1} else {0});

    slow_count
}
```

Cascaded counters

```
entity main(clk: clock, rst: bool) -> int<20> {
    let max = 4;
    let fast_count = inst cntr_max(clk, rst, 1, max);

    let tick = fast_count == max;

    let slow_count = inst cntr(clk, rst, if tick {1} else {0});

    slow_count
}
```

Cascaded counters

```
entity main(clk: clock, rst: bool) -> int<20> {
    let max = 4;
    let fast_count = inst cntr_max(clk, rst, 1, max);

    let tick = fast_count == max;

    let slow_count = inst cntr(clk, rst, if tick {1} else {0});

    slow_count
}
```

Cascaded counters

```
entity main(clk: clock, rst: bool) -> int<20> {
    let max = 4;
    let fast_count = inst cntr_max(clk, rst, 1, max);

    let tick = fast_count == max;

    let slow_count = inst cntr(clk, rst, if tick {1} else {0});

    slow_count
}
```

Cascaded counters

```
entity main(clk: clock, rst: bool) -> int<20> {
    let max = 4;
    let fast_count = inst cntr_max(clk, rst, 1, max);

    let tick = fast_count == max;

    let slow_count = inst cntr(clk, rst, if tick {1} else {0});

    slow_count
}
```

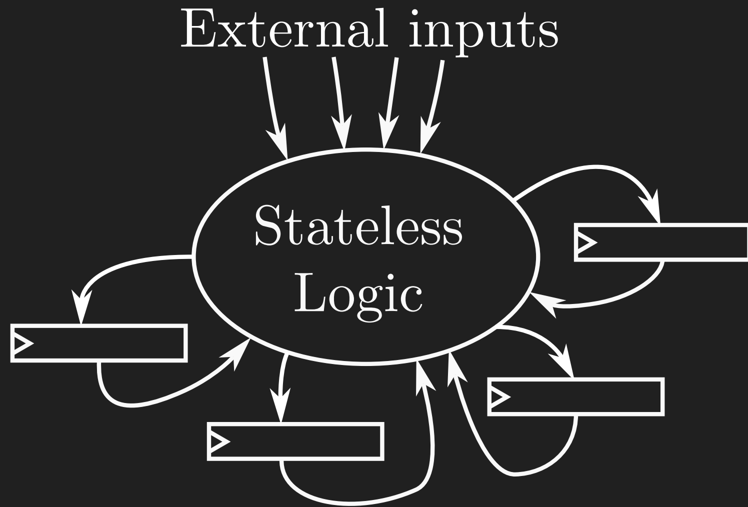
Cascaded counters

```
entity main(clk: clock, rst: bool) -> int<20> {
    let max = 4;
    let fast_count = inst cntr_max(clk, rst, 1, max);

    let tick = fast_count == max;

    let slow_count = inst cntr(clk, rst, if tick {1} else {0});

    slow_count
}
```

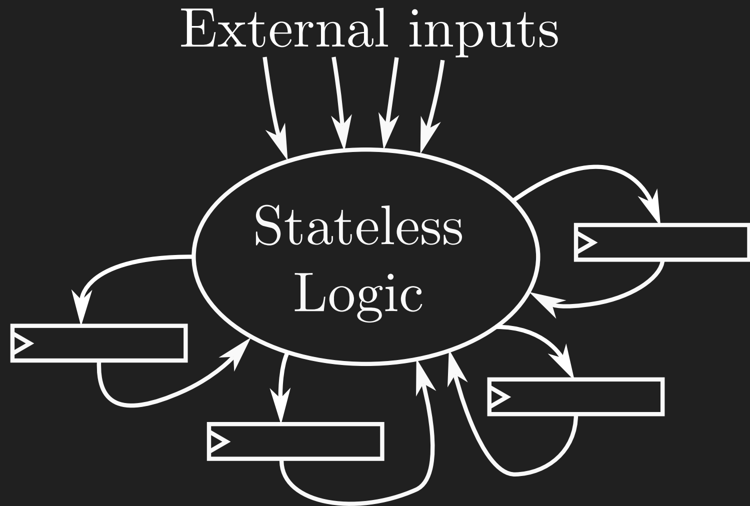
Inlined

```
entity main(clk: clock, rst: bool) -> int<20> {
  let max = 4;
  reg(clk) fast_count reset(rst: 0) =
    if fast_count == max { 0 } else { trunc(fast_count + 1) };

  let tick = fast_count == max;

  reg(clk) seconds reset(rst: 0) =
    if tick {trunc(seconds + 1)} else {seconds};

  seconds
}
```

We have a very limited programming model!

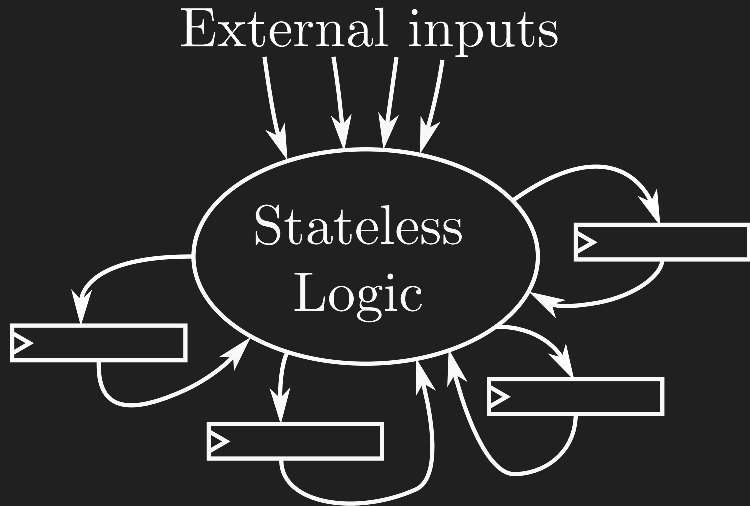


We have a very limited programming model!



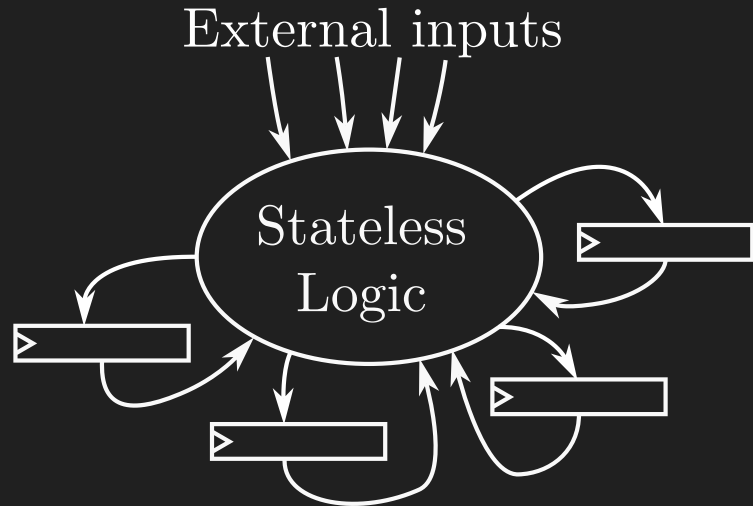
- Compute new value of all registers using current values

We have a very limited programming model!



- Compute new value of all registers using current values
- Update registers simultaneously

We have a very limited programming model!



- Compute new value of all registers using current values
- Update registers simultaneously

No loops, no conditional execution etc.

One more example:

A dot moves along a line. Press a button when it is in the middle

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

Loops kind of encode state

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

Loops kind of encode state

- **Menu**
-

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

Loops kind of encode state

- Menu
- **Game**

In Software

```
fn game(input: bool) {
  let won_last = false;
  loop {
    while !button {}

    let x = 0;
    loop {
      if button && x == 128 {
        won_last = true;
        break;
      }
      else if button {
        won_last = false;
        break;
      }
      x += 1;
    }
  }
}
```

Loops kind of encode state

- Menu
- Game

And associated state

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),        _) => Play(trunc(x + 1))
    };
  state
}
```

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),        _) => Play(trunc(x + 1))
    };
  state
}
```

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),        _) => Play(trunc(x + 1))
    };
  state
}
```

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),        _) => Play(trunc(x + 1))
    };
  state
}
```

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),        _) => Play(trunc(x + 1))
    };
  state
}
```

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),        _) => Play(trunc(x + 1))
    };
  state
}
```

In Hardware

```
enum State {
  Menu{won_last: bool},
  Play{x: int<8>}
}
entity game(
  clk: clock, rst: bool, button: bool
) -> State {
  reg(clk) state reset(rst: Menu(false)) =
    match (state, button) {
      (Menu(won_last), false) => Menu(won_last),
      (Menu(_),        true) => Play(0),
      (Play(128),      true) => Menu(true)
      (Play(_),        true) => Menu(false)
      (Play(x),           _) => Play(trunc(x + 1))
    };
  state
}
```

Compilation and Performance

Basic Building Blocks

Look Up Tables (LUT)

- Programmable to arbitrary $4\ \mathrm{bit} \rightarrow 1\ \mathrm{bit}$ functions
- Thousands per FPGA

Basic Building Blocks

Look Up Tables (LUT)

- Programmable to arbitrary $4 \text{ bit} \to 1 \text{ bit}$ functions
- Thousands per FPGA

Digital Signal Processing (DSP) blocks

- Multiplier and Adder
- Tens to hundreds per FPGA

Basic Building Blocks

Look Up Tables (LUT)

- Programmable to arbitrary $4\ \text{bit} \to 1\ \text{bit}$ functions
- Thousands per FPGA

Digital Signal Processing (DSP) blocks

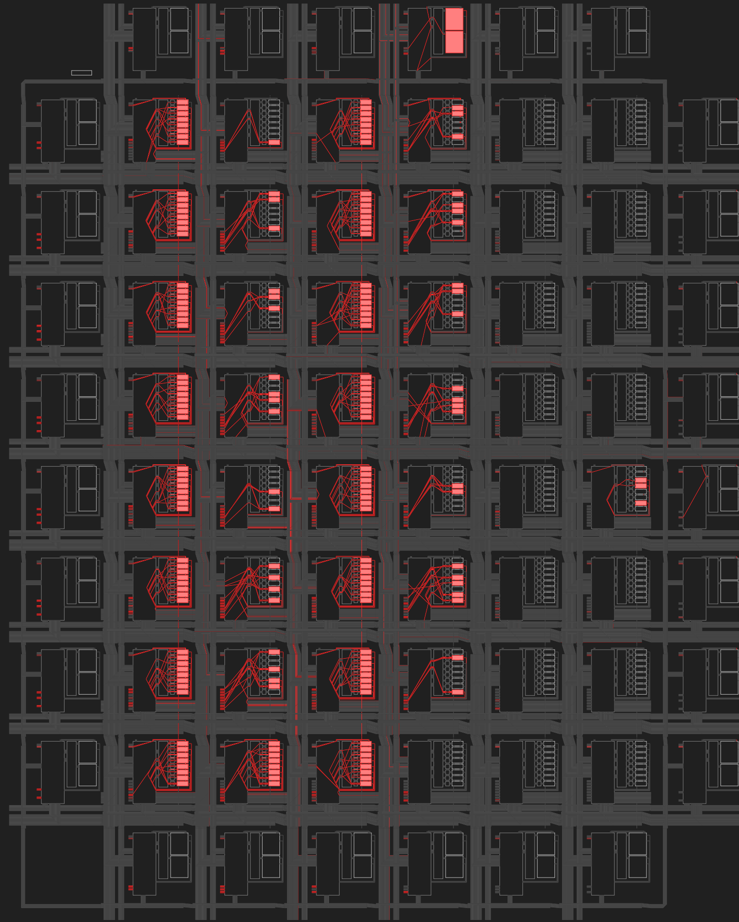
- Multiplier and Adder
- Tens to hundreds per FPGA

Memories

- Blocks of memory in configurable chunks
- Kilobits to megabits per FPGA

Compilation

```
fn add_three(a, b, c,) {
  a + b + c
}
```



- Circuit Board

- Circuit Board
- Blood

- Circuit Board
- Blood
- Sacrifice
  - Preferably a goat
  - Rabbit works in a pinch

- Circuit Board
- Blood
- Sacrifice
  - Preferably a goat
  - Rabbit works in a pinch
- Cat ears

Synthesis

```
fn add_three(a, b, c,) {
  a + b + c
}
```

List of “nets”

2 adders. A should connect to input 1 of adder 1...

Place and Route

List of “nets”

2 adders. A should connect to input 1 of adder 1…

Placement selects a **physical** location for each component

Routing selects how to connect them

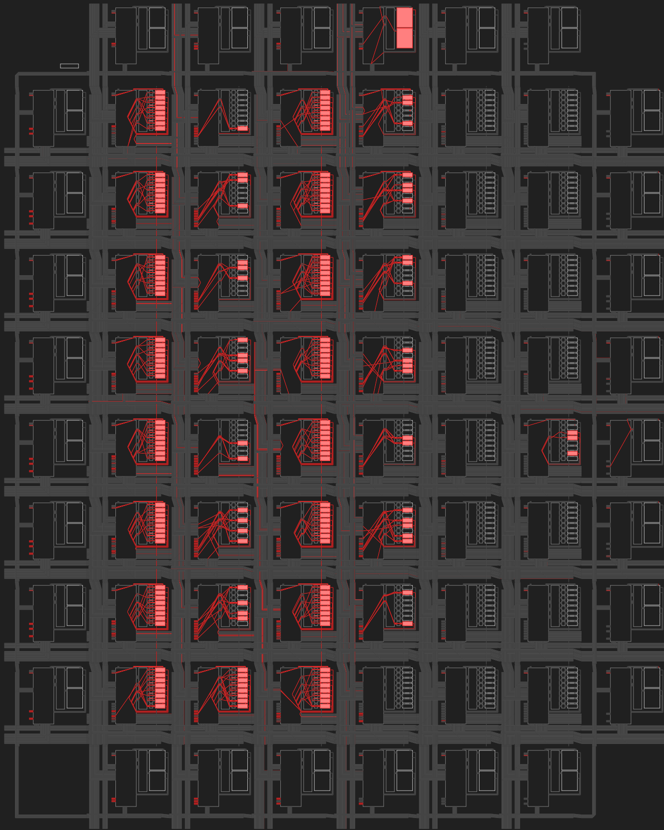
Software performance is simple

- Only 1 metric: runtime

Software performance is simple

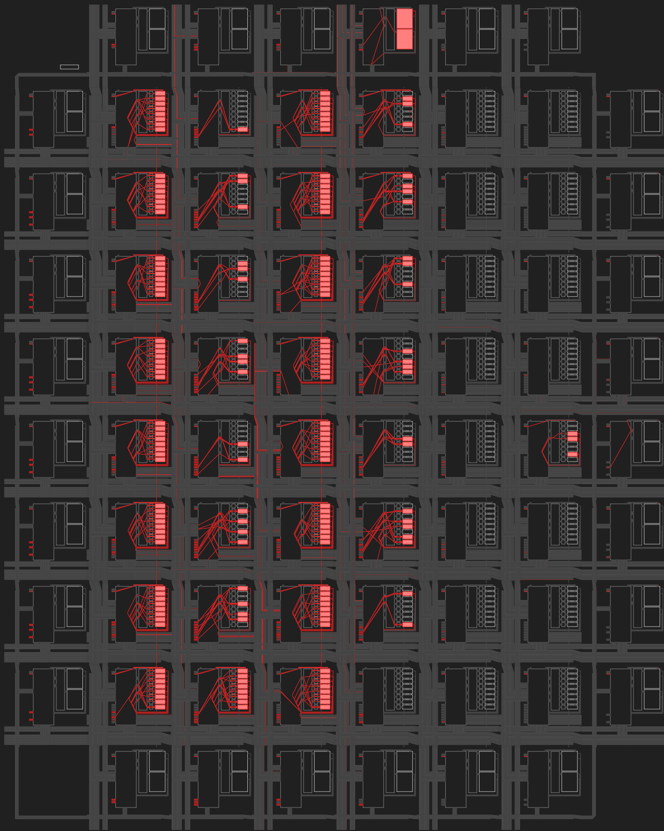
- Only 1 metric: runtime
- Runtime degrades slowly

Resource Usage



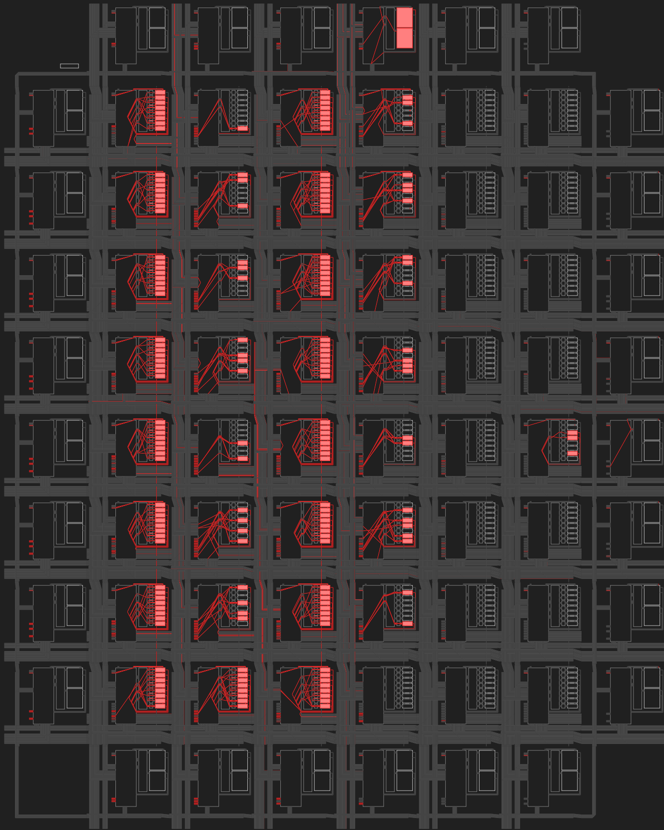
- Each computation you perform takes some basic cells

Resource Usage



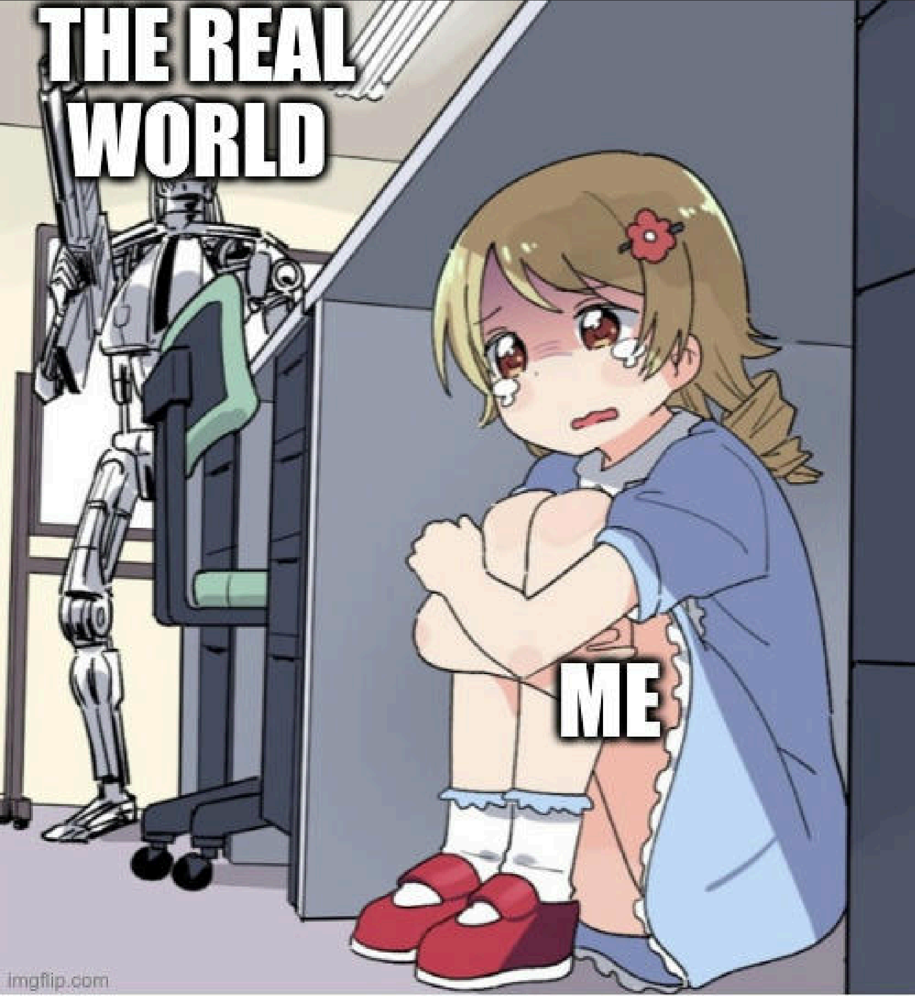
- Each computation you perform takes some basic cells
- Very binary transition from ok to bad

Resource Usage



- Each computation you perform takes some basic cells
- Very binary transition from ok to bad
- You still pay for unused components

Clock Frequency



Clock Frequency

- (Mosty) Fixed frequency clock. $10 - 200\ \mathrm{MHz}$

Clock Frequency

- (Mosty) Fixed frequency clock. $10 - 200\ \mathrm{MHz}$
- **Static** timing analysis

Clock Frequency

- (Mosty) Fixed frequency clock. $10 - 200\ \mathrm{MHz}$
- **Static** timing analysis
- Another pass/fail metric

Performance

```
[INFO] Place and route maximum frequencies:
clk$SB_IO_IN_$glb_clk: 30.5 MHz (target: 12 MHz) <- Is the design fast
enough?
[INFO] Place and route components:
ICESTORM_LC: 181/1280 (14.1%) <- LUTs
ICESTORM_PLL: 0/1      (0.0%)
ICESTORM_RAM: 0/16     (0.0%)
SB_GB: 2/8            (25.0%)
SB_IO: 2/112           (1.8%)
SB_WARMBOOT: 0/1       (0.0%)
```

Performance

```
[INFO] Place and route maximum frequencies:
$glbnet$_e_880[0]: 231.3 MHz (target: 200 MHz) <- Is the design fast enough?
[INFO] Place and route components:
ALU54B: 0/78              (0.0%)
CLKDIVF: 0/4              (0.0%)
DCCA: 6/56               (10.7%)
DCSC: 0/2                 (0.0%)
...
TRELLIS_COMB: 9193/83640 (11.0%) <- LUTs
TRELLIS_ECLKBUF: 0/8      (0.0%)
TRELLIS_FF: 3829/83640    (4.6%) <- Registers
```

WS2812b Addressable RGB LEDs

Gustav Sörnäs

What is WS2812b?

What is WS2812b?

- Addressable RGB LEDs
  - Individually controllable

What is WS2812b?

- Addressable RGB LEDs
  - Individually controllable
- Mounted on a PCB, or on a strip, or in a matrix, or flying free

What is WS2812b?

- Addressable RGB LEDs
  - Individually controllable
- Mounted on a PCB, or on a strip, or in a matrix, or flying free
- Connected in series

What is WS2812b?

- Addressable RGB LEDs
  - Individually controllable
- Mounted on a PCB, or on a strip, or in a matrix, or flying free
- Connected in series
- Many names, same protocol
  - WS2811, 2812, 2812b, 2813
  - APA104, 106
  - SK6812
  - NeoPixel

What is it used for?

Anytime bright lights are fun. (Meaning, wherever you want!)

What is it used for?

Anytime bright lights are fun. (Meaning, wherever you want!)

- Keyboard backlight and underglow

What is it used for?

Anytime bright lights are fun. (Meaning, wherever you want!)

- Keyboard backlight and underglow
- Decoration: Taped to anything, anywhere

What is it used for?

Anytime bright lights are fun. (Meaning, wherever you want!)

- Keyboard backlight and underglow
- Decoration: Taped to anything, anywhere
- Matrix: text, graphics, clocks

What are its issues?

- Connected in series - one break is enough to stop transmission

What are its issues?

- Connected in series - one break is enough to stop transmission
- Power delivery

What are its issues?

- Connected in series - one break is enough to stop transmission
- Power delivery
- Operating frequency

Why do we want it in hardware?

800 KHz operating frequency means a challenge for both:

Why do we want it in hardware?

800 KHz operating frequency means a challenge for both:

- Software implementation in a microcontroller (bit-banging)
  - Slow clock
  - Dependent on clock speed

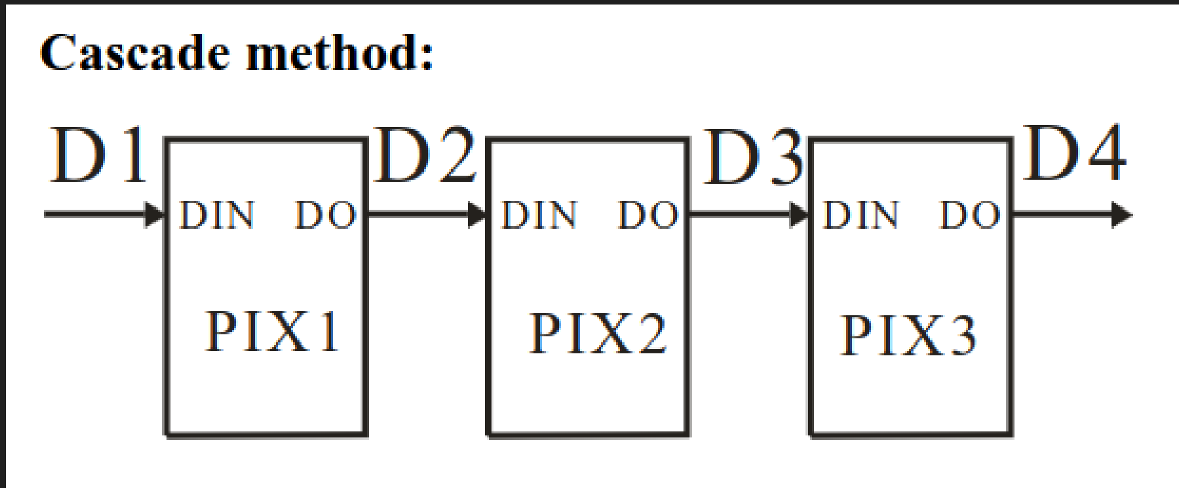
Why do we want it in hardware?

800 KHz operating frequency means a challenge for both:

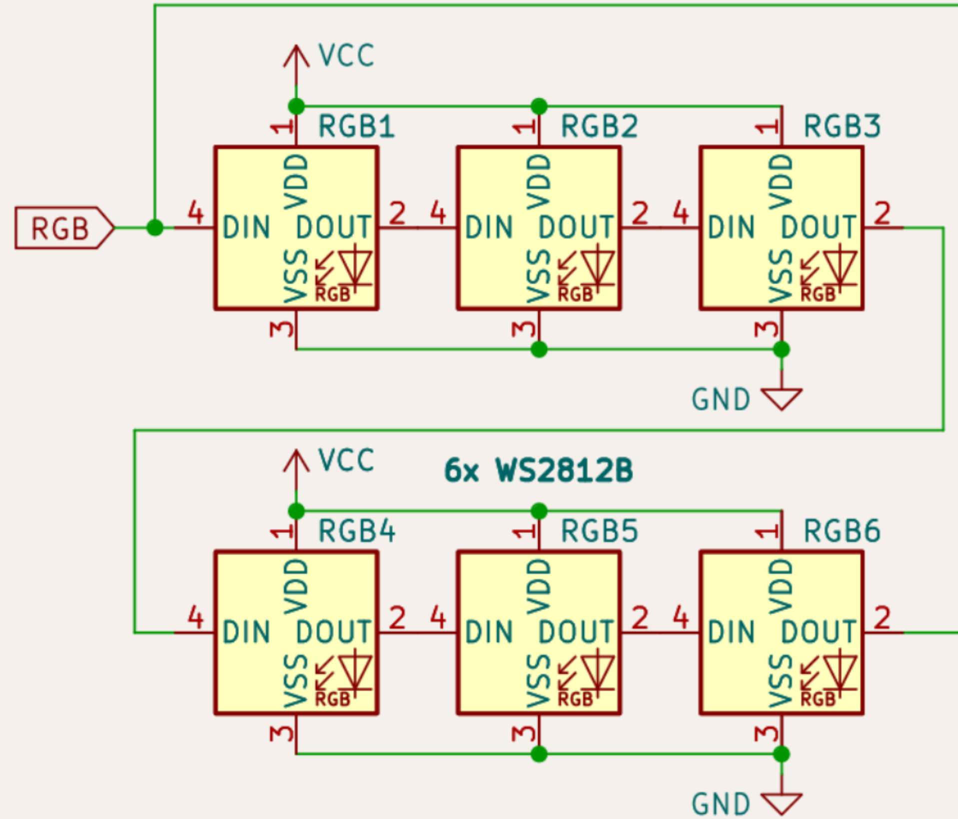
- Software implementation in a microcontroller (bit-banging)
  - Slow clock
  - Dependent on clock speed
- General purpose CPUs (without a real-time operating system)

How does the protocol work?

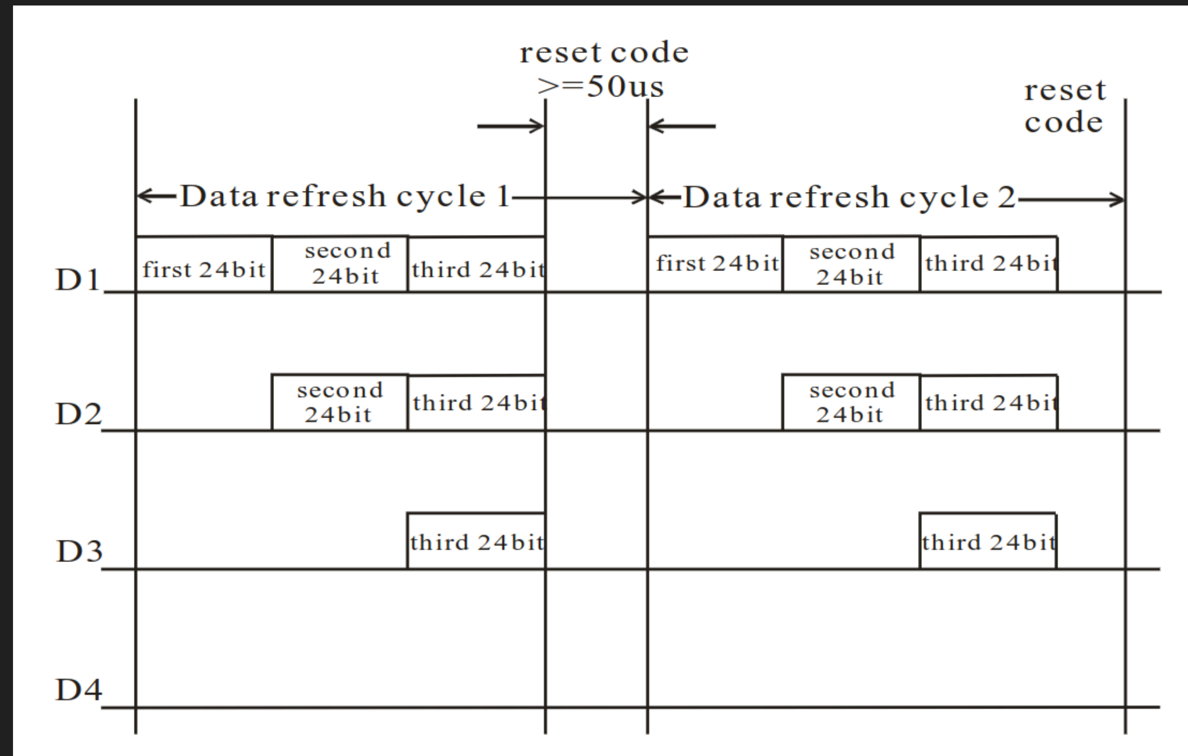
Recall that LEDs are connected serially.



Keyboard example:



Send a stream of colors. Every LED reads the first and sends the rest along.



Pixel data is sent as… *checks notes*

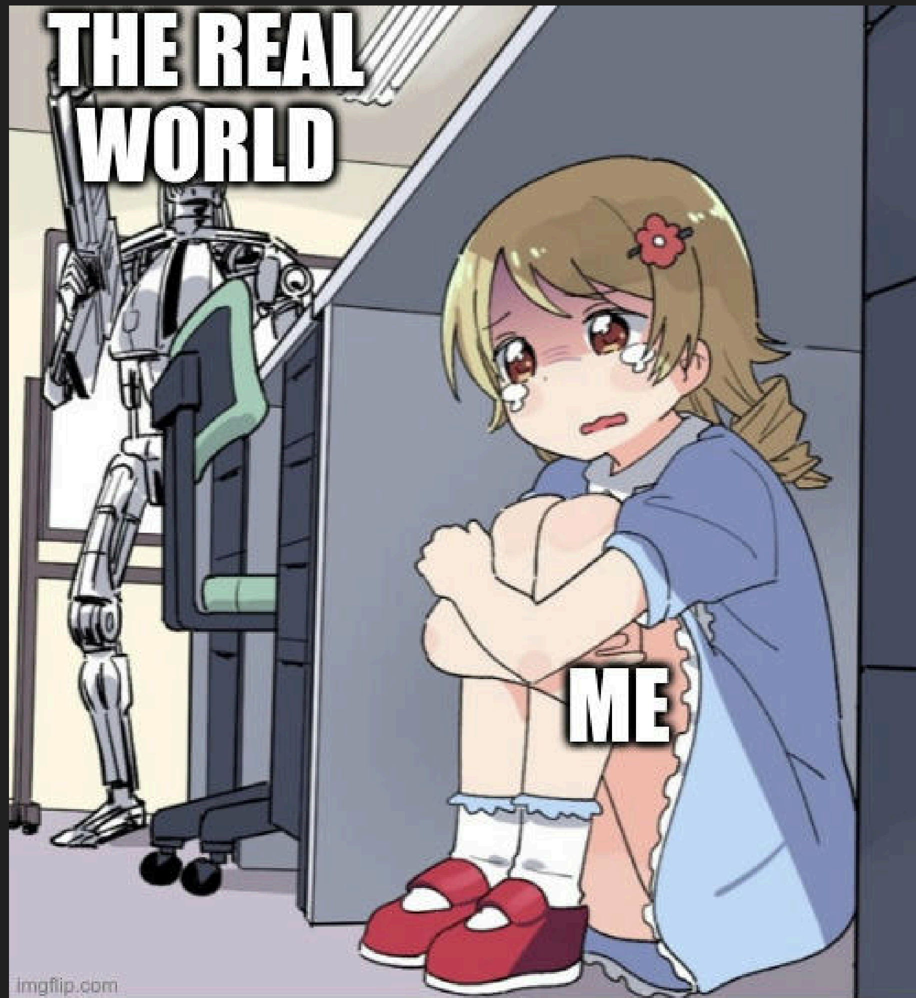
Pixel data is sent as… *checks notes*

Composition of 24bit data:

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Note: Follow the order of GRB to sent data and the high bit sent at first.

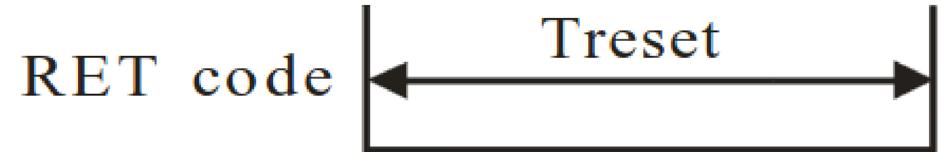
GRB???


THE REAL WORLD
ME
imgflip.com

Bits are different lengths of high+low, *not* simple low/high once per clock cycle.

Reset marks start of next data cycle. (RET is reset, not return.)

Sequence chart:



Protocol recap

Protocol recap

- 24 bits per LED

Protocol recap

- 24 bits per LED
- Bits are have a low pulse and high pulse, with different lengths signifying 0/1

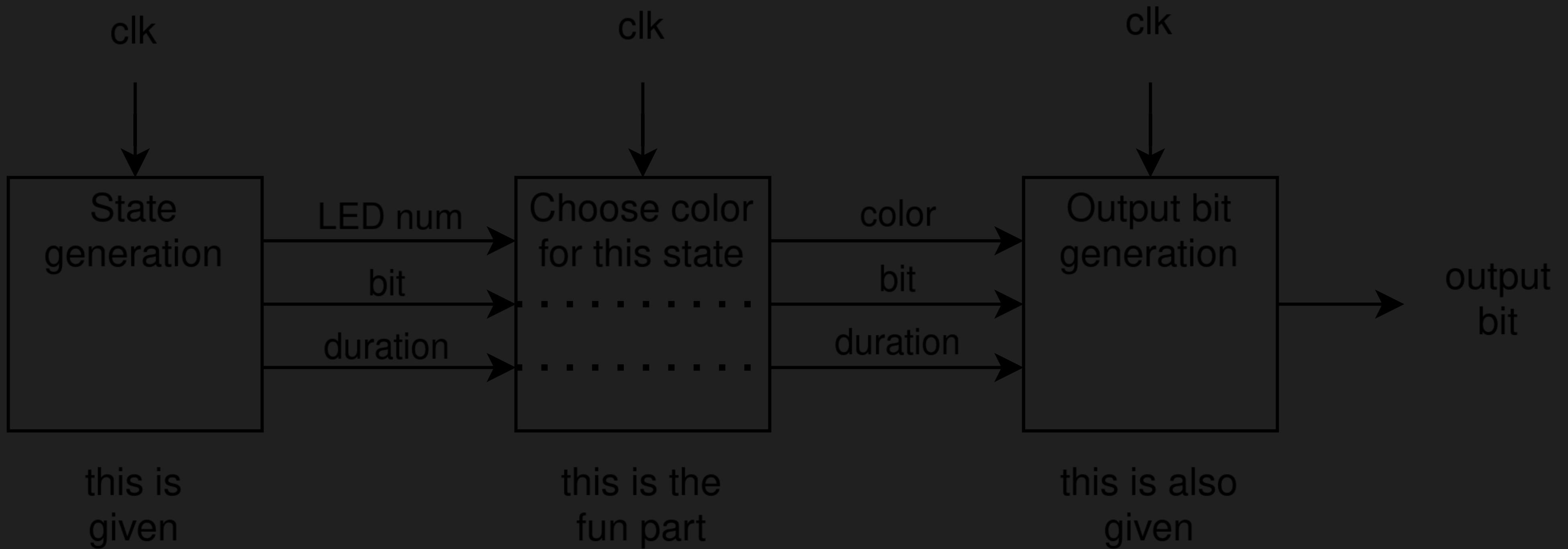
Protocol recap

- 24 bits per LED
- Bits are have a low pulse and high pulse, with different lengths signifying 0/1
- Send LED data in the same order it should be on the strip

Protocol recap

- 24 bits per LED
- Bits are have a low pulse and high pulse, with different lengths signifying 0/1
- Send LED data in the same order it should be on the strip
- Reset between every transmission

How is the project setup?



How is the project setup?

```
entity user_code(clk: clock, rst: bool, timing: Timing) -> bool {
  let state: OutputControl<int<4>> = inst state_gen(clk, rst, 3, timing);
  //                             ~~~~~~                                 |
  //                                                  we want 3 leds --+

  let with_color = match state {
    OutputControl::Ret => OutputControl::Ret,
    OutputControl::Led$(payload: led_num, bit, duration) => {
      let color = Color(0, 20, 0); // everything is green

      OutputControl::Led$(payload: color, bit, duration)
    }
  };

  output_gen(with_color, t)
}
```

Available FPGAs

Available FPGAs

- Go Board
  - iCE40 HX1K
  - 1 280 LUTs
  - 25 MHz clock
  - 1 PMOD (8 pins)
  - 4 buttons
  - UART interface

Available FPGAs

- Go Board
  - iCE40 HX1K
  - 1 280 LUTs
  - 25 MHz clock
  - 1 PMOD (8 pins)
  - 4 buttons
  - UART interface
- ECPIX-5
  - ECP5 45F
  - 45 000 LUTs
  - 100 MHz clock
  - 8 PMOD (64 pins)
  - External buttons
  - UART interface

Available FPGAs

- Go Board
  - iCE40 HX1K
  - 1 280 LUTs
  - 25 MHz clock
  - 1 PMOD (8 pins)
  - 4 buttons
  - UART interface
- ECPIX-5
  - ECP5 45F
  - 45 000 LUTs
  - 100 MHz clock
  - 8 PMOD (64 pins)
  - External buttons
  - UART interface
- ULX3S
  - ECP5 85F
  - 84 000 LUTs
  - 25 MHz clock
  - 28 pins
  - 6 buttons
  - UART interface

Available FPGAs

- Go Board
  - iCE40 HX1K
  - 1 280 LUTs
  - 25 MHz clock
  - 1 PMOD (8 pins)
  - 4 buttons
  - UART interface
- ECPIX-5
  - ECP5 45F
  - 45 000 LUTs
  - 100 MHz clock
  - 8 PMOD (64 pins)
  - External buttons
  - UART interface
- ULX3S
  - ECP5 85F
  - 84 000 LUTs
  - 25 MHz clock
  - 28 pins
  - 6 buttons
  - UART interface

Watch out for different clock speeds when dealing with counters.

For now, choose one clock speed and stick to it.

Installing and running Spade

Spade is the compiler, Swim is the build system.

Swim takes care of the different FPGAs we have.

Installing and running Spade

Spade is the compiler, Swim is the build system.

Swim takes care of the different FPGAs we have.

1. Install Swim:

```
cargo install --git https://gitlab.com/spade-lang/swim
```

Installing and running Spade

Spade is the compiler, Swim is the build system.

Swim takes care of the different FPGAs we have.

1. Install Swim:
```
cargo install --git https://gitlab.com/spade-lang/swim
```
2. Clone the WS2812 repository:
```
git clone https://gitlab.com/lithekod/hardware/fpga-ws2812
```

Installing and running Spade

Spade is the compiler, Swim is the build system.

Swim takes care of the different FPGAs we have.

1. Install Swim:

```
cargo install --git https://gitlab.com/spade-lang/swim
```

2. Clone the WS2812 repository:

```
git clone https://gitlab.com/lithekod/hardware/fpga-ws2812
```

3. Change directory and build:

```
cd fpga-ws2812
swim build
```

Installing and running Spade

Spade is the compiler, Swim is the build system.

Swim takes care of the different FPGAs we have.

1. Install Swim:

   ```
   cargo install --git https://gitlab.com/spade-lang/swim
   ```

2. Clone the WS2812 repository:

   ```
   git clone https://gitlab.com/lithekod/hardware/fpga-ws2812
   ```

3. Change directory and build:

   ```
   cd fpga-ws2812
   swim build
   ```

`https://lithekod.se/hardware/fpga-evening`

How do you get the code to the FPGA?

In a real project, `swim upload` runs synthesis, pnr, and upload.

How do you get the code to the FPGA?

In a real project, `swim upload` runs synthesis, pnr, and upload.

Today, me and Frans will keep the FPGAs by our computers, since we only have a few.

How do you get the code to the FPGA?

In a real project, `swim upload` runs synthesis, pnr, and upload.

Today, me and Frans will keep the FPGAs by our computers, since we only have a few.

Because I didn't think about this earlier, you will probably have to send the code to us using e.g. `https://paste.rs/web`. (Sorry!)

Please ask if something is unclear.

Some things to do

Roughly in increasing order of difficulty.

Some things to do

Roughly in increasing order of difficulty.

1. Change the color of all three LEDs to yellow.

Some things to do

Roughly in increasing order of difficulty.

1. Change the color of all three LEDs to yellow.
2. Control 10 LEDs.

Some things to do

Roughly in increasing order of difficulty.

1. Change the color of all three LEDs to yellow.
2. Control 10 LEDs.
3. Have different colors for all LEDs. Hint: arrays.

Some things to do

Roughly in increasing order of difficulty.

1. Change the color of all three LEDs to yellow.
2. Control 10 LEDs.
3. Have different colors for all LEDs. Hint: arrays.
4. Animate in some way. For example, toggle on/off every second. Hint: counters, array offsets.

Some things to do

Roughly in increasing order of difficulty.

1. Change the color of all three LEDs to yellow.
2. Control 10 LEDs.
3. Have different colors for all LEDs. Hint: arrays.
4. Animate in some way. For example, toggle on/off every second. Hint: counters, array offsets.
5. React to pressing buttons on the FPGA. (You will need to make some more changes in the same file.)

Some more things to do

Please tell us if you want to do any of these!

Some more things to do

Please tell us if you want to do any of these!

6. Control the FPGA from your computer.
    - All FPGAs can be communicated with via UART. We have a UART implementation in Spade somewhere on GitLab.

Some more things to do

Please tell us if you want to do any of these!

6. Control the FPGA from your computer.
   - All FPGAs can be communicated with via UART. We have a UART implementation in Spade somewhere on GitLab.
7. Control the FPGA from a microcontroller.
   - We have Arduino Uno, ESP8266 and Raspberry Pi Pico (and hopefully enough level-shifters).
   - Hint: SPI.

Some more things to do

Please tell us if you want to do any of these!

6. Control the FPGA from your computer.
   - All FPGAs can be communicated with via UART. We have a UART implementation in Spade somewhere on GitLab.
7. Control the FPGA from a microcontroller.
   - We have Arduino Uno, ESP8266 and Raspberry Pi Pico (and hopefully enough level-shifters).
   - Hint: SPI.
8. Use an RFID reader to draw a unique color pattern.
   - We only have one reader. Maybe do this one as a group.

Closing remarks