Choosing Determinacy: Combining Concurrency and Timing in the Sparse Synchronous Model

Stephen A. Edwards

Boolean Functions as a Table

W	*X*	*Y*	*Z*	*a*	*b*	*c*	*d*	*e*	*f*	*g*
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	0	0	1	1
1	0	1	0	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

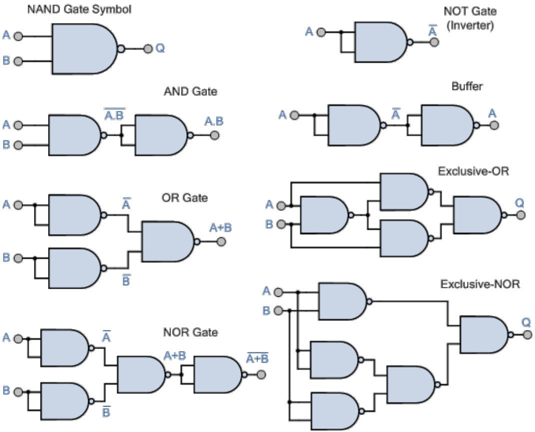
For n Boolean inputs and m Boolean outputs,

Each of 2^n rows lists the m Boolean outputs for that row's input combination

Each possible input combination appears in exactly one row

It is a total function: $2^n \to 2^m$

Acyclic Networks of NAND2 Gates



https://www.electronics-tutorials.ws/logic/universal-gates.html

Directed Acyclic Graph of Two-input NAND gates

Primary inputs: no incoming edges

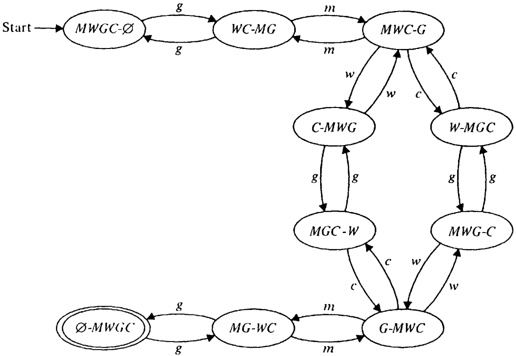
All others: two incoming edges

Semantics: set value of each primary input; in topological order, set each node's value to the NAND of the values of its two incoming edges

Can compute any Boolean function

Deterministic: Assignment of each node's value depends only on the primary inputs, not the particular topological order chosen

Deterministic Finite Automaton as a Table



After Hopcroft and Ullman, Introduction to Automata Theory, Languages, and Computation, 1979

- List of states, some are accepting
- A start state
- List of inputs
- Complete table of transitions (state, input) → state

Deterministic if, for each state and input, there's exactly one next state

Synchronous Digital Logic

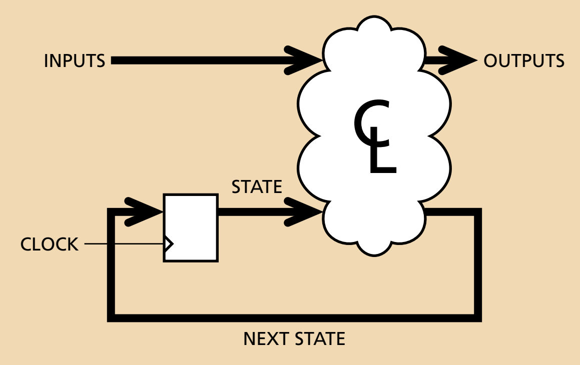
DAG with three types of nodes:

- NAND2: two incoming edges
- flip-flop: one incoming edge
- primary input: no incoming edges

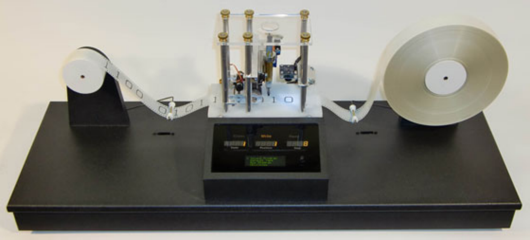
Every cycle in the graph must pass through a flip-flop

In each cycle, primary input nodes set to new value, flip-flop nodes set to input in last cycle (false in first)

NAND2 nodes evaluated in topological order, ignoring flop-flop input edges



Turing Machine



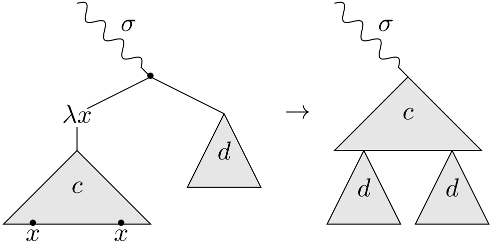
https://aturingmachine.com/

- A tape of symbols
- A head that can read and write symbols and move left or right
- A state register
- A table of instructions: (state, symbol) $\rightarrow$ (state, symbol, left/right)

Deterministic because there's exactly one thing to do at each step

The Lambda Calculus

expr ::= *expr expr*
| λ *variable* . *expr*
| *constant*
| *variable*
| (*expr*)



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

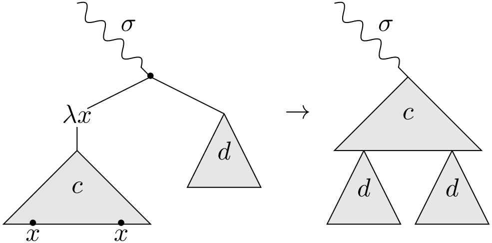
two = $\lambda f \,.\, \lambda x \,.\, f\ (f\ x)$
three = $\lambda f \,.\, \lambda x \,.\, f\ \big(f\ (f\ x)\big)$
five = $\lambda f \,.\, \lambda x \,.\, f\ (f\ (f\ (f\ (f\ x))))$
plus = $\lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x)$

plus three two

Expand plus

The Lambda Calculus

expr ::= *expr expr*
| λ *variable* . *expr*
| *constant*
| *variable*
| (*expr*)



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

$$
\begin{aligned}
\text{two} &= \lambda f\ .\ \lambda x\ .\ f\ (f\ x) \\
\text{three} &= \lambda f\ .\ \lambda x\ .\ f\ \big(f\ (f\ x)\big) \\
\text{five} &= \lambda f\ .\ \lambda x\ .\ f\ (f\ (f\ (f\ (f\ x)))) \\
\text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x)
\end{aligned}
$$

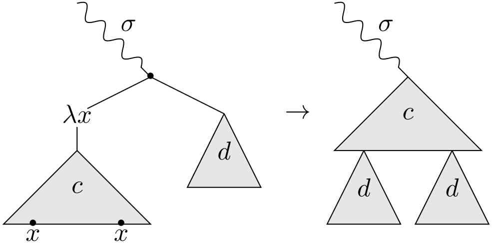
$\underline{\text{plus}}$ three two

$\underline{\big(\lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x)\big)\ \text{three}}$ two

β-reduce (λ m ...) three

The Lambda Calculus

expr ::= *expr expr*
| λ *variable* . *expr*
| *constant*
| *variable*
| (*expr*)



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

$$\begin{aligned}
\text{two} &= \lambda f\,.\,\lambda x\,.\,f\,(f\,x)\\
\text{three} &= \lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,x))\\
\text{five} &= \lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,(f\,(f\,x))))\\
\text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x)
\end{aligned}$$

$\underline{\text{plus}}$ three two

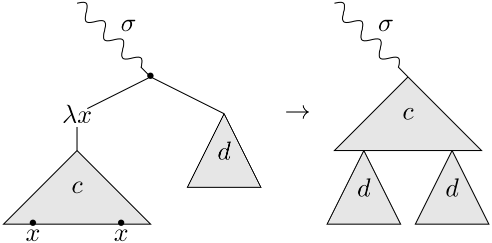
$\underline{(\lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x))\ \text{three}}$ two

$\underline{(\lambda n.\lambda f.\lambda x.\ \text{three}\ f\ (n\ f\ x))\ \text{two}}$

β-reduce (λ n ...) two
(could have expanded three)

The Lambda Calculus

expr ::= *expr expr*
| λ *variable* . *expr*
| *constant*
| *variable*
| (*expr*)



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

$$\begin{aligned} \text{two} &= \lambda f \,.\, \lambda x \,.\, f\,(f\,x) \\ \text{three} &= \lambda f \,.\, \lambda x \,.\, f\,(f\,(f\,x)) \\ \text{five} &= \lambda f \,.\, \lambda x \,.\, f\,(f\,(f\,(f\,(f\,x)))) \\ \text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.\; m\,f\,(n\,f\,x) \end{aligned}$$

$$\underline{\text{plus}}\ \text{three two}$$

$$\underline{(\lambda m.\lambda n.\lambda f.\lambda x.\; m\,f\,(n\,f\,x))\ \text{three}}\ \text{two}$$

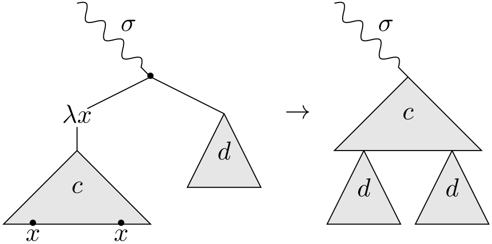
$$\underline{(\lambda n.\lambda f.\lambda x.\; \text{three}\,f\,(n\,f\,x))\ \text{two}}$$

$$\lambda f.\lambda x.\ \underline{\text{three}\,f\,(\text{two}\,f\,x)}$$

Expand three and beta reduce twice
(could have expanded two)

The Lambda Calculus

expr ::= *expr expr*
| λ *variable* . *expr*
| *constant*
| *variable*
| (*expr*)



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

$$\begin{aligned}
\text{two} &= \lambda f\,.\,\lambda x\,.\,f\,(f\,x)\\
\text{three} &= \lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,x))\\
\text{five} &= \lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,(f\,(f\,x))))\\
\text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.\ m\,f\,(n\,f\,x)
\end{aligned}$$

$$\underline{\text{plus}}\ \text{three two}$$

$$\underline{(\lambda m.\lambda n.\lambda f.\lambda x.\ m\,f\,(n\,f\,x))\ \text{three}}\ \text{two}$$

$$\underline{(\lambda n.\lambda f.\lambda x.\ \text{three}\ f\,(n\,f\,x))\ \text{two}}$$

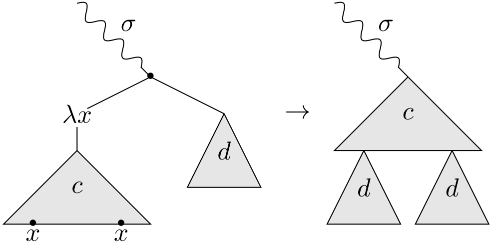
$$\lambda f.\lambda x.\ \underline{\text{three}\ f\,(\text{two}\ f\,x)}$$

$$\lambda f.\lambda x.\ f\,(f\,(f\,(\underline{\text{two}\ f\,x})))$$

Expand two and beta reduce twice

The Lambda Calculus

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{expr}\ \mathit{expr} \\
 & | & \lambda\ \mathit{variable}\ .\ \mathit{expr} \\
 & | & \mathit{constant} \\
 & | & \mathit{variable} \\
 & | & (\mathit{expr})
\end{array}
$$



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

$$
\begin{array}{lcl}
\text{two} & = & \lambda f\ .\ \lambda x\ .\ f\ (f\ x) \\
\text{three} & = & \lambda f\ .\ \lambda x\ .\ f\ (f\ (f\ x)) \\
\text{five} & = & \lambda f\ .\ \lambda x\ .\ f\ (f\ (f\ (f\ (f\ x)))) \\
\text{plus} & = & \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x)
\end{array}
$$

$$\underline{\text{plus}}\ \text{three two}$$

$$\underline{(\lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x))\ \text{three}}\ \text{two}$$

$$\underline{(\lambda n.\lambda f.\lambda x.\ \text{three}\ f\ (n\ f\ x))\ \text{two}}$$

$$\lambda f.\lambda x.\ \underline{\text{three}\ f\ (\text{two}\ f\ x)}$$

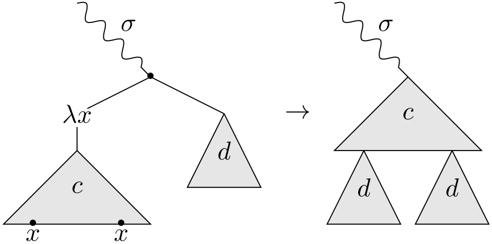
$$\lambda f.\lambda x.\ f\ (f\ (f\ (\underline{\text{two}\ f\ x})))$$

$$\lambda f.\lambda x.\ f\ (f\ (f\ (f\ (f\ x))))$$

Normal form (nothing more to do)

The Lambda Calculus

expr ::= *expr expr*
| λ *variable* . *expr*
| *constant*
| *variable*
| (*expr*)



Kozen, Church-Rosser Made Easy, Fundamenta Informaticae, 103(1–4), 2010

two $= \lambda f\,.\,\lambda x\,.\,f\,(f\,x)$
three $= \lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,x))$
five $= \lambda f\,.\,\lambda x\,.\,f\,(f\,(f\,(f\,(f\,x))))$
plus $= \lambda m.\lambda n.\lambda f.\lambda x.\ m\,f\,(n\,f\,x)$

$\underline{\text{plus}}$ three two

$\underline{(\lambda m.\lambda n.\lambda f.\lambda x.\ m\,f\,(n\,f\,x))\ \text{three}}$ two

$\underline{(\lambda n.\lambda f.\lambda x.\ \text{three}\,f\,(n\,f\,x))\ \text{two}}$

$\lambda f.\lambda x.\ \underline{\text{three}\,f\,(\text{two}\,f\,x)}$

$\lambda f.\lambda x.\ f\,(f\,(f\,(\underline{\text{two}\,f\,x})))$

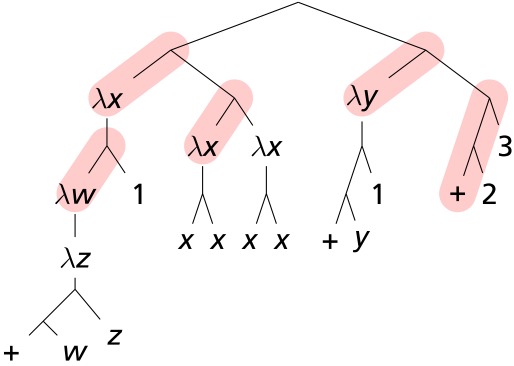
$\lambda f.\lambda x.\ f\,(f\,(f\,(f\,(f\,x))))$

five

This is “five”

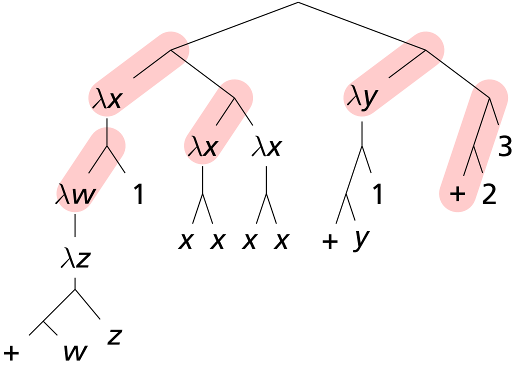
Many reducible sub-expressions: Church-Rosser: all choices OK

$$\Big(\big(\lambda x\;.\;((\lambda w\;.\;\lambda z\;.\;+\;w\;z)\;1)\big)\;((\lambda x\;.\;x\;x)\;(\lambda x\;.\;x\;x))\Big)\;((\lambda y\;.\;+\;y\;1)\;(+\;2\;3))$$



Many reducible sub-expressions: Church-Rosser: all choices OK

$$\Big(\big(\lambda x\,.\,((\lambda w\,.\,\lambda z\,.\,+\ w\ z)\ 1)\big)\ ((\lambda x\,.\,x\ x)\ (\lambda x\,.\,x\ x))\Big)\ ((\lambda y\,.\,+\ y\ 1)\ (+\ 2\ 3))$$

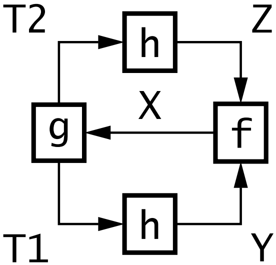


β-reduction is confluent

$$\begin{tikzcd}
L \arrow[r, "\beta^*"] \arrow[d, "\beta^*"] & M_1 \arrow[d, "\beta^*"] \\
M_2 \arrow[r, "\beta^*"] & N
\end{tikzcd}$$

⇒ An expression's normal form, if it exists, is unique

Kahn Process Networks



Network of concurrent processes communicate through FIFOs

Blocking reads; non-blocking writes

Sequence of data values passed through each FIFO is deterministic

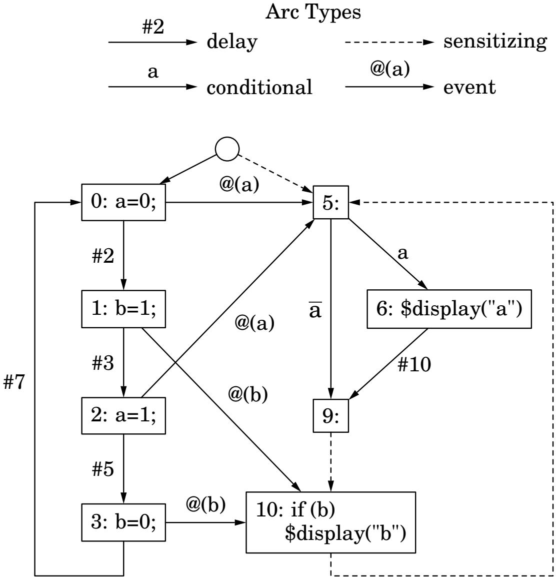
```
process f(in int u, in int v,
          out int w) {
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
process g(in int u, out int v,
          out int w) {
  int i; bool b = true;
  for (;;) {
    i = wait(u);
    if (b) send(i, v);
    else   send(i, w);
    b = !b;
  }
}
```

```
process h(in int u, out int v,
          int init) {
  int i;
  send(v, init);
  for (;;) {
    i = wait(u);
    send(i, v);
  }
}
```

```
channel int X, Y, Z, T1, T2;

f(Y, Z, X);
g(X, T1, T2);
h(T1, Y, 0);
h(T2, Z, 1);
```

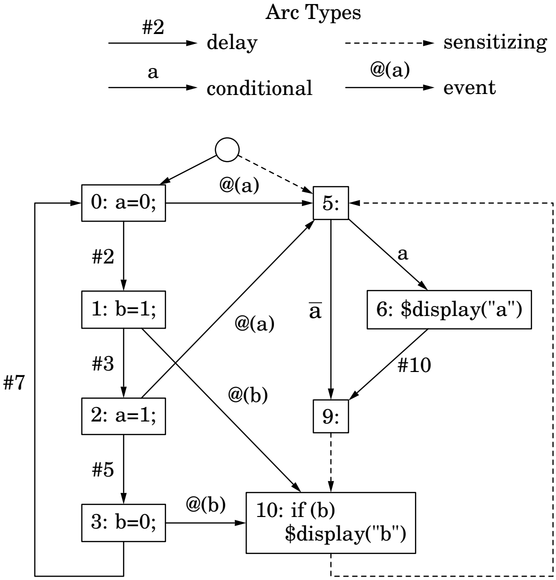
Discrete-Event Simulation: Verilog

```
module ex;
   reg a, b;

   always begin a = 1; #2;
                b = 1; #3;
                a = 0; #5;
                b = 0; #7; end


   always begin
      @(a);
      if (a) begin
        $display("a"); #10; end
      @(b);
      if (b) $display("b");
   end
endmodule
```



Discrete-Event Simulation: Verilog

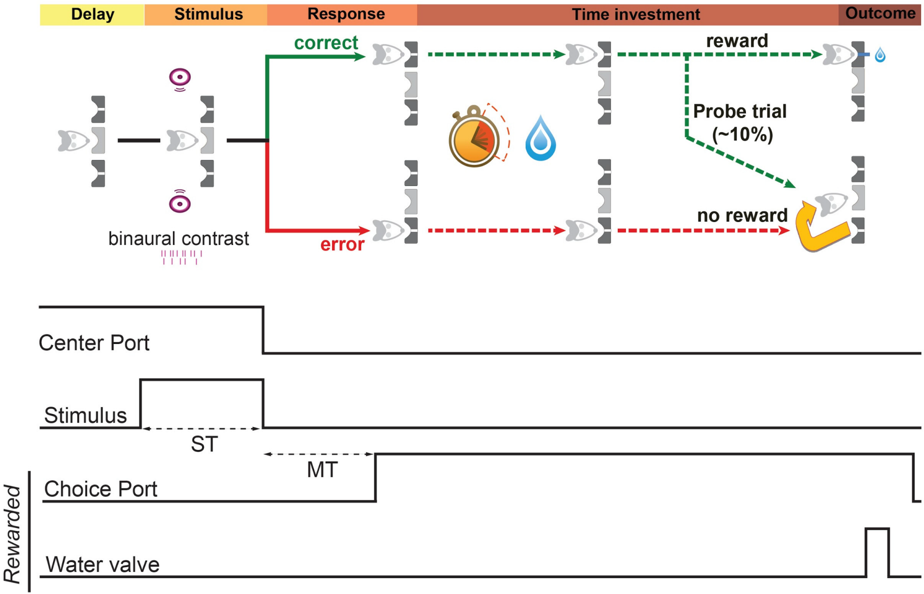
1. Select, remove, and execute earliest pending event e from queue
2. At an event `@()`, mark successor as sensitive
3. On assignment `v =`, schedule all events sensitive to the variable
4. On delay `#`, schedule successor in the future



Nondeterminism in Verilog

```
module race;
  reg a;

  initial begin #10; a = 1;
                #10; a = 0;
                #10; a = 1; end

  always @(a) $display("%0t first", $time);

  always @(a) $display("%0t second", $time);

endmodule
```

```
10 first
10 second
20 second
20 first
30 first
30 second
```

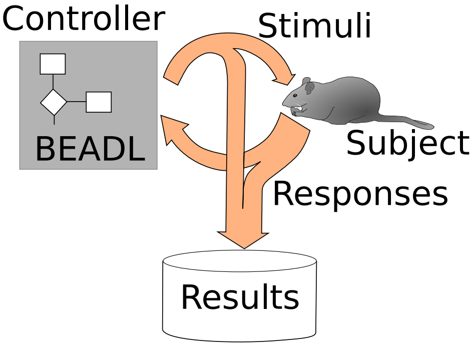


Adam Kepecs, Cold Spring Harbor Laboratory [Lak et al., Neuron 84(1), 2014]

Bpod: An Open Hardware Platform for Behavioral Monitoring and Control

Sanworks.io, spun out of Kepecs’ lab.
Teensy 3.6: ARM Cortex M4, 180 MHz

SSM: The Idea

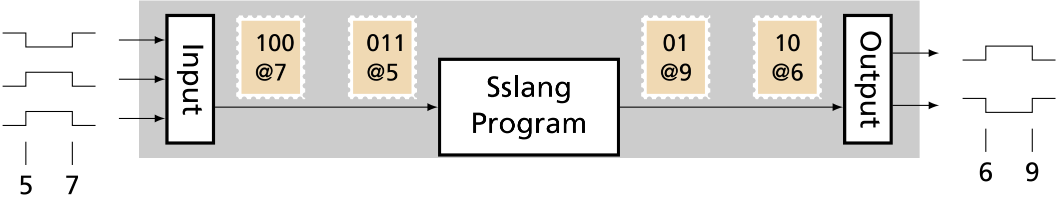


```
training gate valve led =
  let timeout = new 0
  valve <- 1
  delay (ms 100)
  valve <- 0
  after (s 10), timeout <- 1
  wait gate || timeout
  if updated timeout
    failed <- failed + 1
  else
    led <- 1
    after (ms 100), led <- 0
    wait led
```

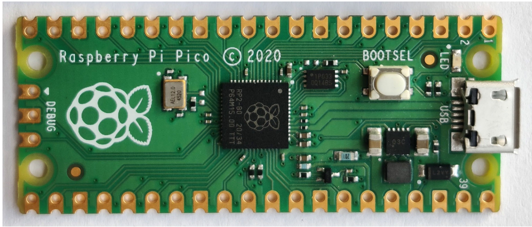
SSM: Wishlist

Deterministic formal semantics

Explicit model-time delays only; platform-independent timing above some minimum delay (synchronous logic)



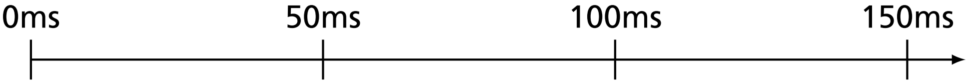
"Bare metal" microcontroller implementations: hardware counter/timer drives timing, timer interrupts for scheduling



Concurrency

Time modeled arithmetically

Time in seconds
Can add, subtract, multiply, and divide time intervals



Time modeled arithmetically

Time is quantized;
quantum not user-visible

Quantum might be
1 MHz, 16 MHz, etc.
Integer timestamps thwart Zeno

Time modeled arithmetically

Time is quantized;
quantum not user-visible

Program thinks processor is infinitely fast: execution a sequence of zero-time instants (hence “synchronous”)

Every instruction that runs in an instant sees the same timestamp

Time modeled arithmetically

Time is quantized;
quantum not user-visible

Program thinks processor is infinitely fast: execution a sequence of zero-time instants (hence “synchronous”)

Nothing happens in
most instants (hence “sparse”)

```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

led = 0

```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

led = 0

```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

led = 0

```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

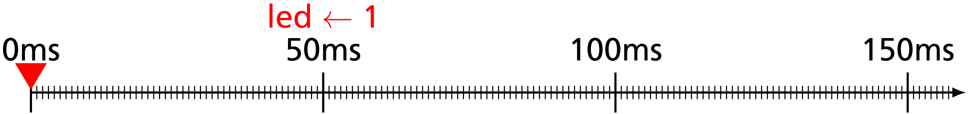
led = 0

```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update



led = 0

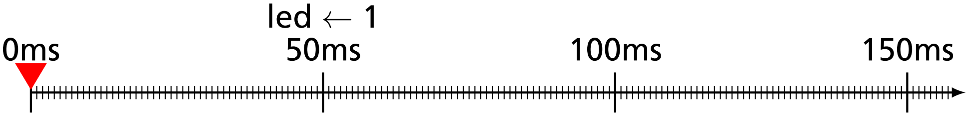
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



led = 0

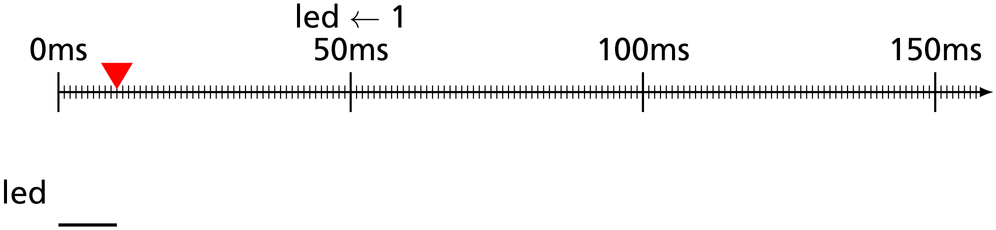
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



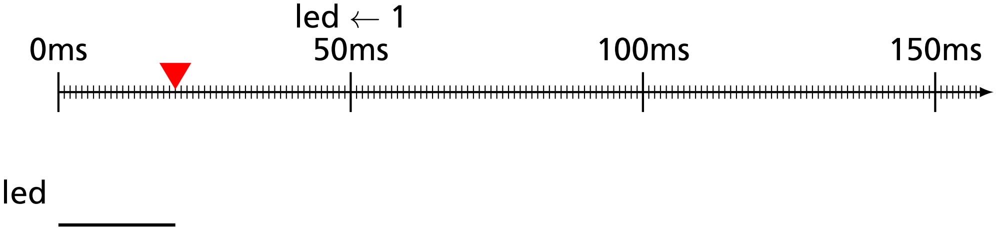
```
blink led =
  loop
    after ms 50,
      led <- not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable

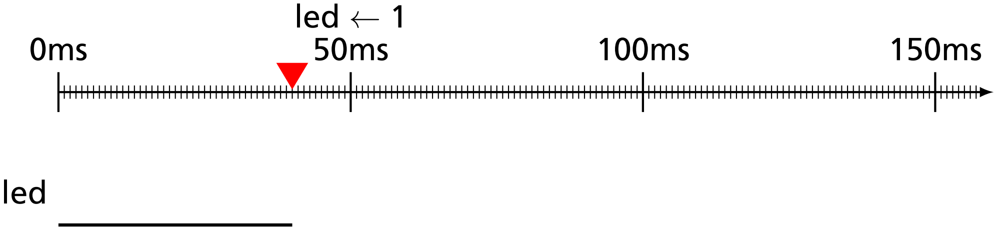
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



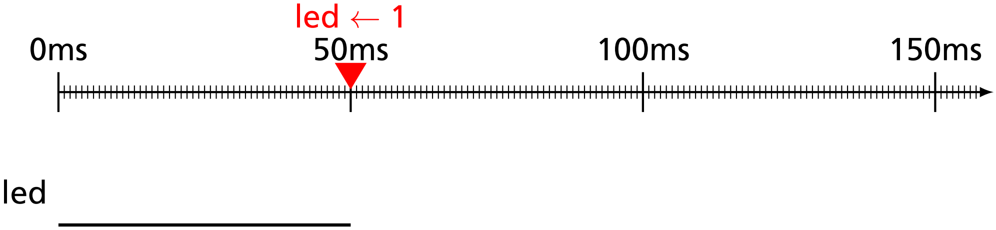
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



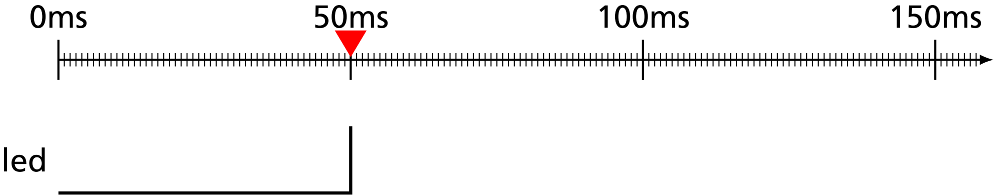
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



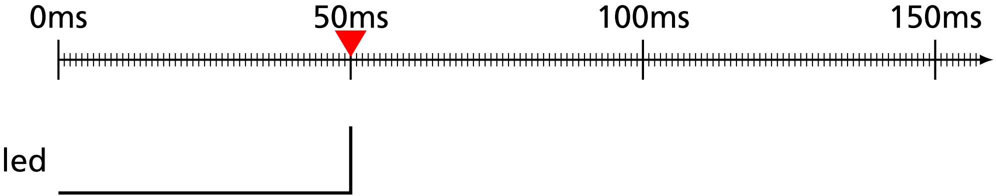
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



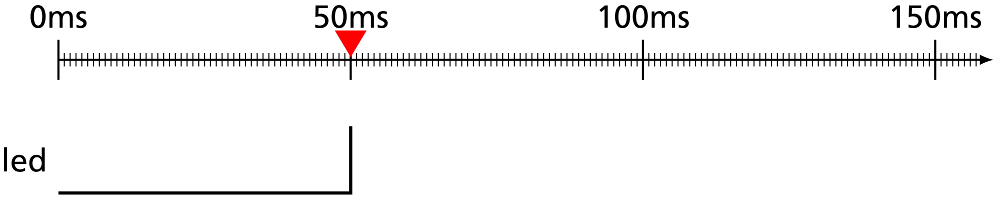
```
blink led =
  loop
    after ms 50,
      led <- not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



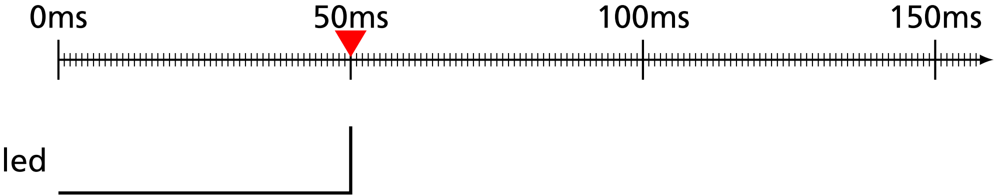
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



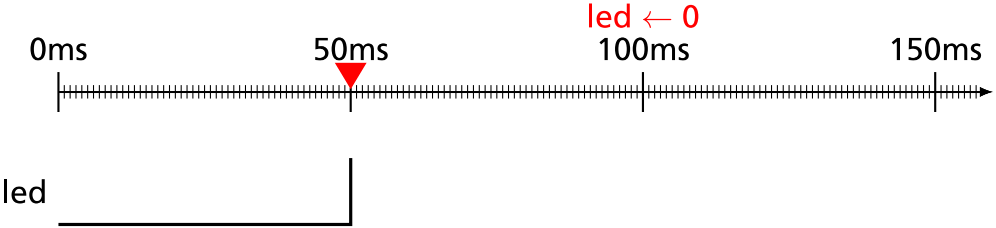
```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



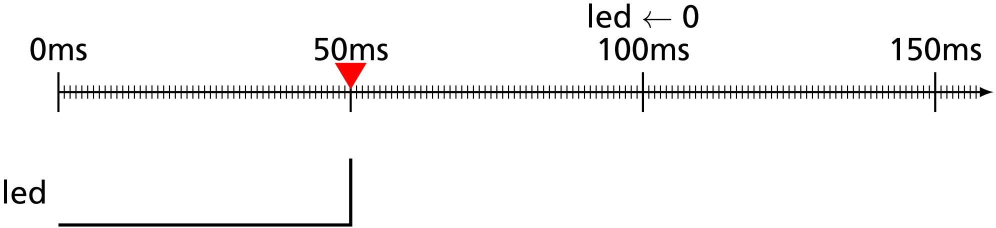
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



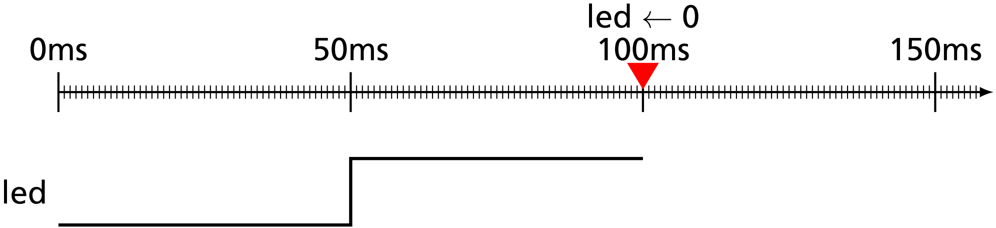
```
blink led =
  loop
    after ms 50,
      led <- not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



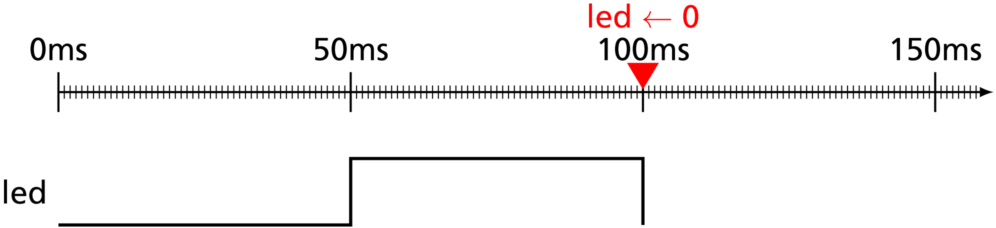
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



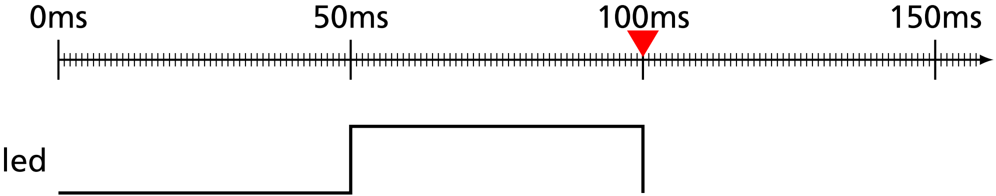
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



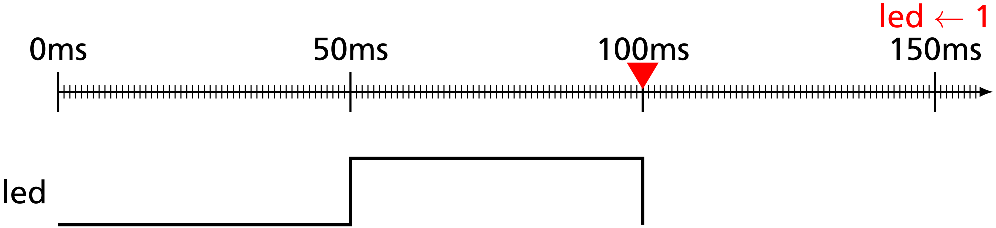
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



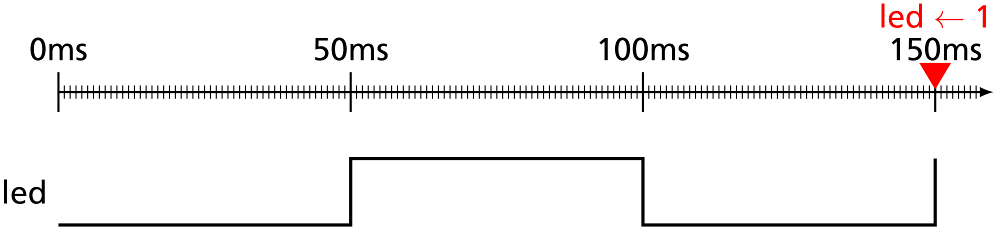
```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



SSM: Parallel Composition

A desired SSM library: input debounce

Nervous rats often jitter before making a decision; want a library that discards "on" events shorter than *x* ms

⇒ Parallel composition?

Feedback loops?

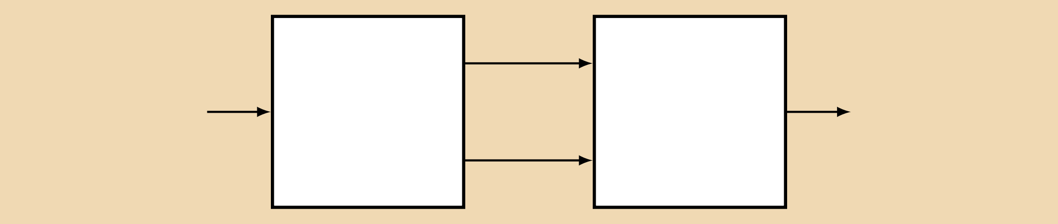
Simultaneous events?

Contradictions?

Simultaneous Events

What should we do with simultaneous events?

We could simply legistate them away at the input, but they are easy to generate internally.

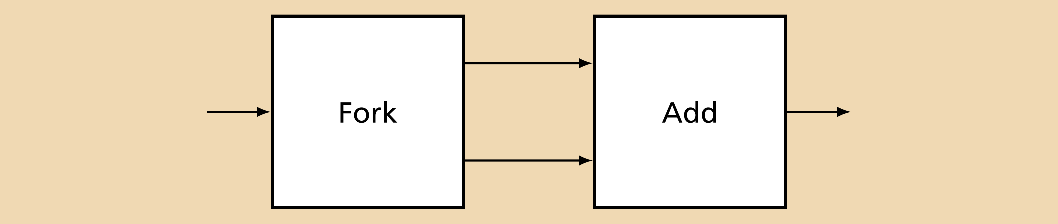


What should this do?

Simultaneous Events

What should we do with simultaneous events?

We could simply legistate them away at the input, but they are easy to generate internally.

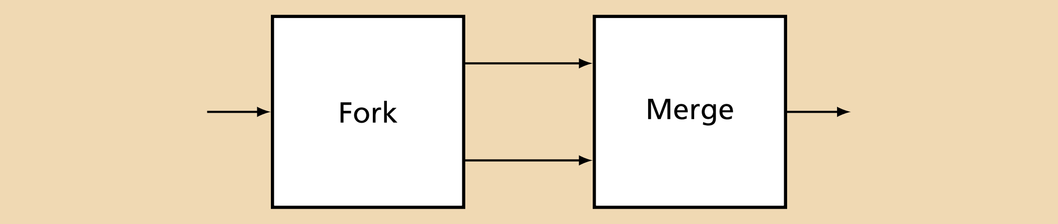


Seems reasonable: output is double the input

Simultaneous Events

What should we do with simultaneous events?

We could simply legistate them away at the input, but they are easy to generate internally.



Should this be allowed? What should its output be?

Concurrent Code Executes in Syntactic Order for Determinism

```
add2 x = x <- deref x + 2      // Add 2 as a side-effect

mult4 x = x <- deref x * 4     // Multiply by 4 as a side-effect
```

Concurrent Code Executes in Syntactic Order for Determinism

```
add2 x = x <- deref x + 2       // Add 2 as a side-effect

mult4 x = x <- deref x * 4      // Multiply by 4 as a side-effect

main =
  let a = new 1       // Allocate a new mutable variable
```

Concurrent Code Executes in Syntactic Order for Determinism

```
add2 x = x <- deref x + 2    // Add 2 as a side-effect

mult4 x = x <- deref x * 4   // Multiply by 4 as a side-effect

main =
  let a = new 1    // Allocate a new mutable variable

  par  add2 a      // Runs first: a ← 1 + 2 = 3
       mult4 a     // Runs second: a ← 3 × 4 = 12
```

Concurrent Code Executes in Syntactic Order for Determinism

```
add2 x = x <- deref x + 2      // Add 2 as a side-effect

mult4 x = x <- deref x * 4     // Multiply by 4 as a side-effect

main =
  let a = new 1      // Allocate a new mutable variable

  par  add2 a        // Runs first: a ← 1 + 2 = 3
       mult4 a       // Runs second: a ← 3 × 4 = 12

  par  mult4 a       // Runs third: a ← 12 × 4 = 48
       add2 a        // Runs fourth: a ← 48 + 2 = 50
```

Concurrent Code May Block on *wait*

```
blink led period =
  let timer = new ()           // void/unit scheduled variable
  loop
    led <- not (deref led)     // Toggle led now
    after period, timer <- ()  // Wait for the period
    wait timer

main led =
  par blink led (ms 50)
      blink led (ms 30)
      blink led (ms 20)        // led toggles three times at time 600
```

FDL 2020: C API for SSM Runtime

Basic trick: Two priority queues

First queue for scheduled variable update events, prioritized by time

Second queue for code to be executed in the current instant; prioritized by structure

A *wait* statement reminds the variable that something is waiting on it

When a variable is written, it schedules the waiting code in the second queue

An *after* statement deletes any existing outstanding event for the variable before scheduling a new one

FDL 2020: C API for SSM Runtime

```
// Routine activation record management
rar_t *enter(size_t size, void (*step)(rar_t *), rar_t *caller,
             uint32_t priority, uint8_t depth)
void call(rar_t *rar)
void fork(rar_t *rar)
void leave(rar_t *rar, size_t size)

// Variable management
void initialize_type(cv_type_t *var, type val)                 // new
void assign_type(cv_type_t *var, uint32_t priority, type val)  // <-
void later_type(cv_type_t *var, uint64_t time, type val)       // after
bool event_on(cv_t *var)

// Trigger management (for wait statements)
void sensitize(cv_t *var, trigger_t *trigger)
void desensitize(trigger_t *trigger)
```

FDL 2020: C API Example

```
rar_examp_t *enter_examp(rar_t *caller, uint32_t priority, unit8_t depth, cv_int_t *a) {
  rar_examp_t *rar = (rar_examp_t *)
     enter(sizeof(rar_examp_t), step_examp, caller, priority, depth);
  rar->a = a;                                   // Store pass-by-reference argument
  rar->trig1.rar = (rar_t *) rar;               // Initialize our trigger
}
void step_examp(rar_t *gen_rar) {
  rar_examp_t *rar = (rar_examp_t *) gen_rar;
  switch (rar->pc) {
  case 0:
    initialize_int(&rar->loc, 0);               // let loc = new 0
    sensitize((cv_t *) rar->a, &rar->trig1);    // wait a
    rar->pc = 1; return;
  case 1:
    if (event_on((cv_t *) rar->a)) {            // if @a then
      desensitize(&rar->trig1);                 // De-register our trigger
    } else return;
    assign_int(&rar->loc, rar->priority, 42);   // loc <- 42
    later_int(rar->a, now+10000, 43);           // after 10ms, a <- 43
    rar->pc = 2;                                // Single routine call: foo 42 loc
    call((rar_t *) enter_foo((rar_t *) rar, rar->priority, rar->depth, 42, &rar->loc));
    return;
  case 2:                                       // Concurrent call: par foo 40 loc; bar 42
    { uint8_t new_depth = rar->depth - 1;       // 2 children
      uint32_t pinc = 1 << new_depth;
      uint32_t new_priority = rar->priority;
      fork((rar_t *) enter_foo((rar_t *) rar, new_priority, new_depth, 40, &rar->loc));
      new_priority += pinc;
      fork((rar_t *) enter_bar((rar_t *) rar, new_priority, new_depth, 42)); }
    rar->pc = 3; return;
  case 3:  ; }
  leave((rar_t *) rar, sizeof(rar_examp_t));    // Terminate
}
```

```
examp a =
  let loc = new 0
  wait a
  loc <- 42
  after ms 10, a <- 43
  par foo 42 loc
  par foo 40 loc
      bar 42
```

TCRS 2023: SSM as a Lua Library

```lua
local ssm = require("ssm")

function ssm.pause(d)
  local t = ssm.Channel {}
  t:after(ssm.msec(d), { go = true })
  ssm.wait(t)
end

function ssm.fib(n)
  if n < 2 then
    ssm.pause(1)
    return n
  end
  local r1 = ssm.fib:spawn(n - 1)
  local r2 = ssm.fib:spawn(n - 2)
  local rp = ssm.pause:spawn(n)
  ssm.wait { r1, r2, rp }
  return r1[1] + r2[1]
  end
```

```lua
local n = 10

ssm.start(function()
  local v = ssm.fib(n)

  print(("fib(%d) => %d"):format(n, v))
  —prints “fib(10) => 55”

  local t = ssm.as_msec(ssm.now())
  print(("Completed in %.2fms"):format(t))
  —prints “Completed in 10.00ms”
end)
```

MEMOCODE 2023: The RP2040

2 ARM Cortex M0+ processor cores, 133 MHz

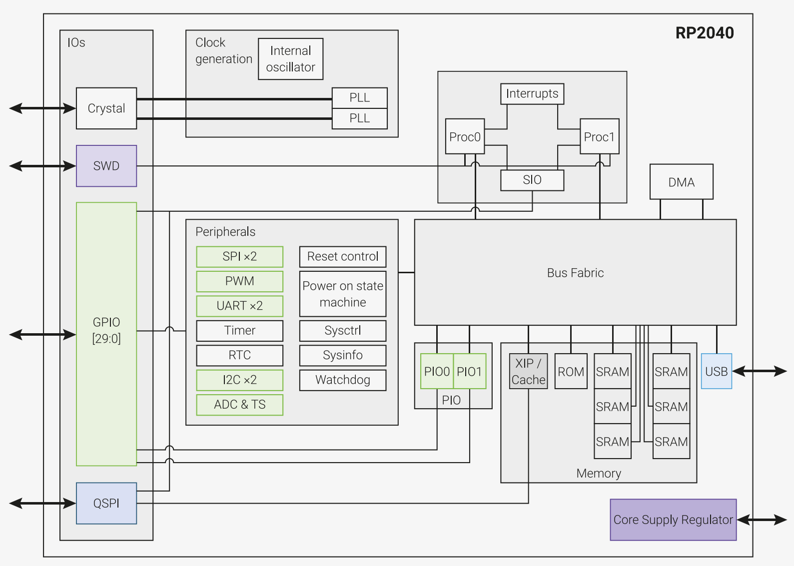
264K SRAM

Off-chip QSPI flash (e.g., 2 MB)

30 GPIO pins

2 Programmable I/O Blocks (PIO)

US$1 quantity 1



MEMOCODE 2023: A PIO Block

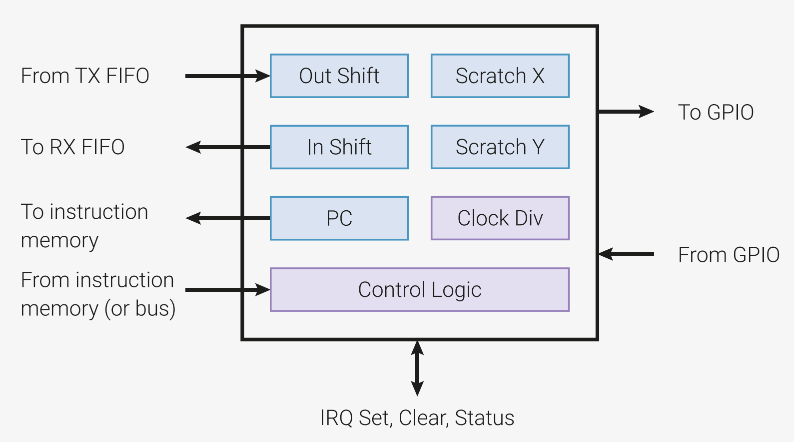
4 "State Machines"

32-instruction memory (shared)

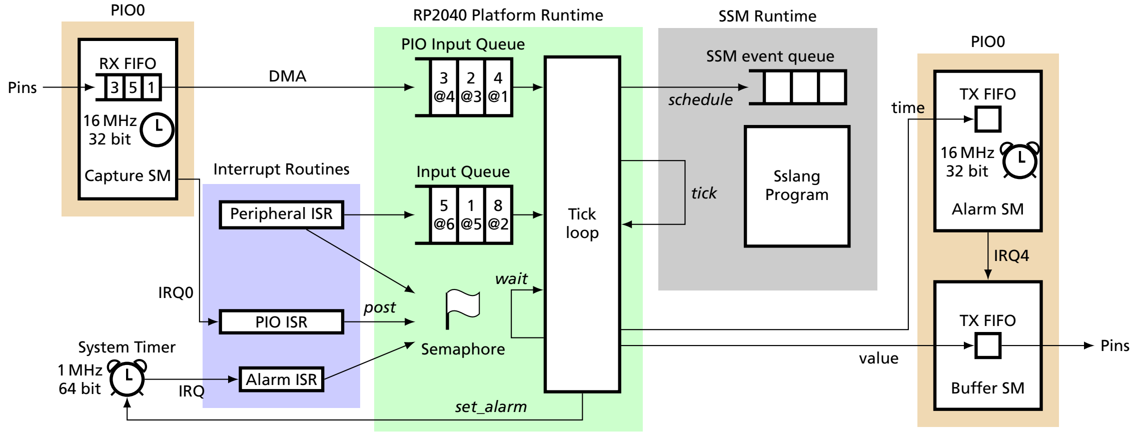
9 instructions (jump, wait, in, out, etc.)

4 32-bit registers

Single-cycle execution



MEMOCODE 2023: Sslang on an RP2040

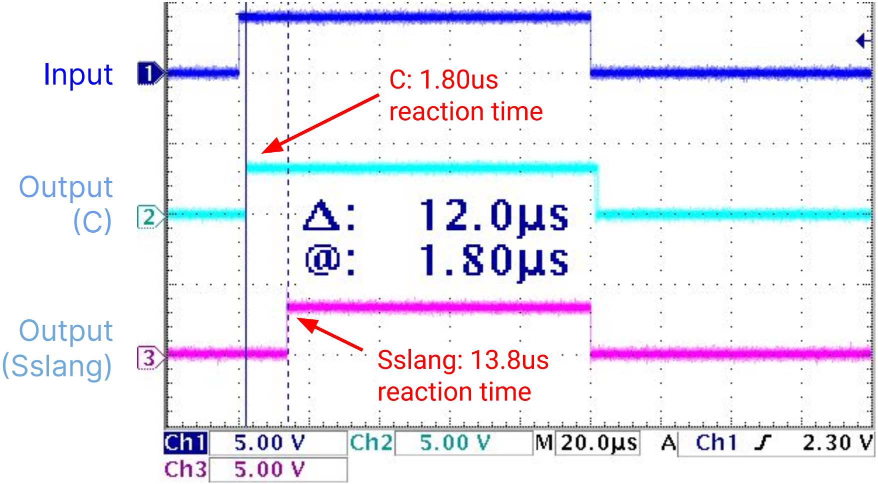


Latency: 10-20 μs Accuracy: 62.5 ns / 16 MHz

```
sleep delay =
  let timer = new ()
  after delay, timer <- ()
  wait timer

waitfor var value =
  while deref var != value
    wait var

debounce delay input press =
  loop
    waitfor input 0
    press <- ()
    sleep delay
    waitfor input 1
    sleep delay

pulse period press output =
  loop
    wait press
    output <- 1
    after period, output <- 0
    wait output

buttonpulse button led =
  let press = new ()
  par debounce (ms 10)  button press
      pulse    (ms 200) press  led
```

21 μs Button-to-LED latency

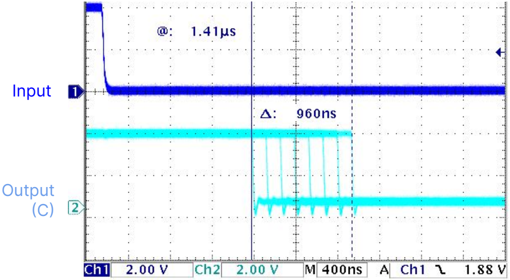
MEMOCODE 2023: 100 μs pulse: C vs Sslang Latency



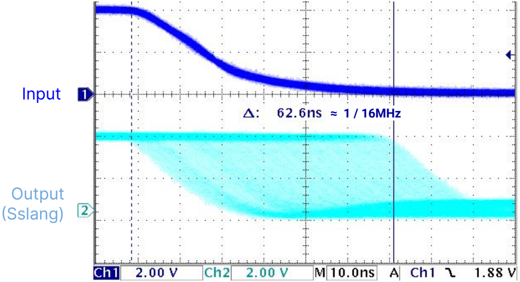
C: 1.80us reaction time

Sslang: 13.8us reaction time

MEMOCODE 2023: 100 µs pulse: C vs Sslang Falling edge



C falling edge:
1.41 µs late, 960 ns jitter



Sslang falling edge:
0 µs late, 62.6 ns jitter (16 MHz clock)