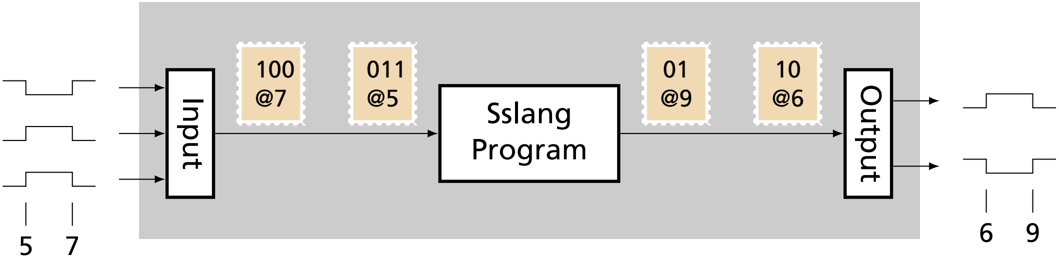
The Sparse Synchronous Model

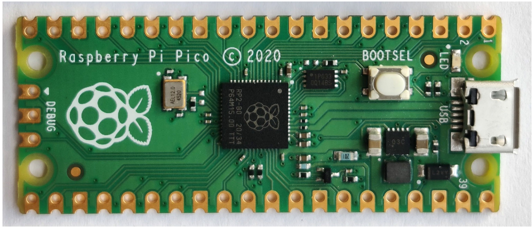
Stephen A. Edwards

CS@CU
COMPUTER SCIENCE AT
COLUMBIA UNIVERSITY

Real-Time Software: Time as Important as Value

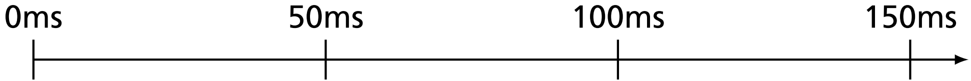


Implemented on Resource-Constrained Microcontrollers



Time modeled arithmetically

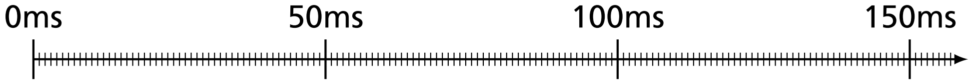
Time in seconds
Can add, subtract, multiply, and divide time intervals



Time modeled arithmetically

Time is quantized;
quantum not user-visible

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

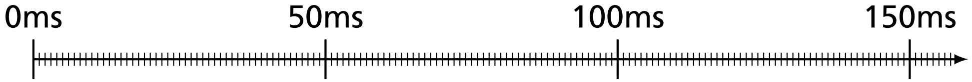


Time modeled arithmetically

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

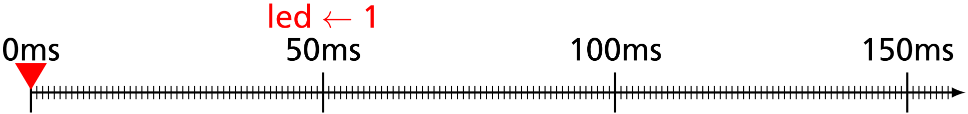
led = 0

```
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

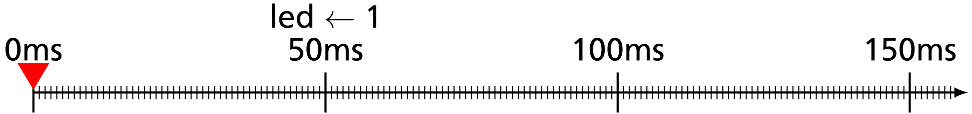
```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



led = 0

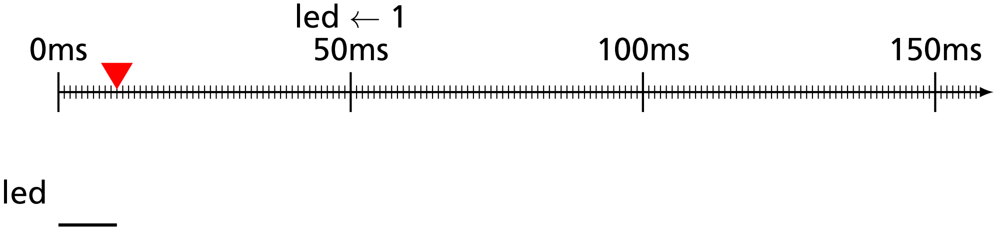
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



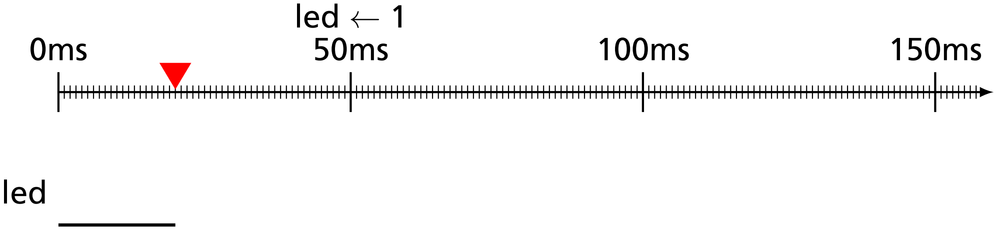
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



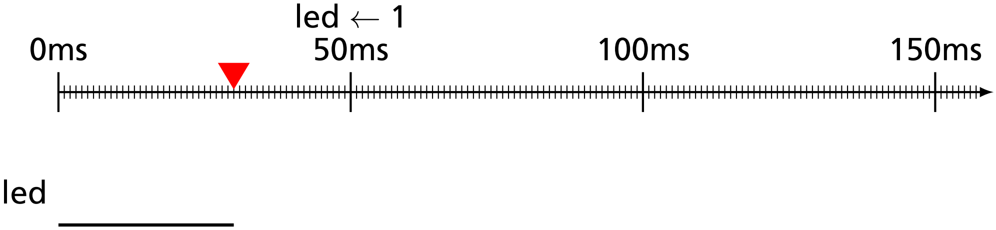
```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

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

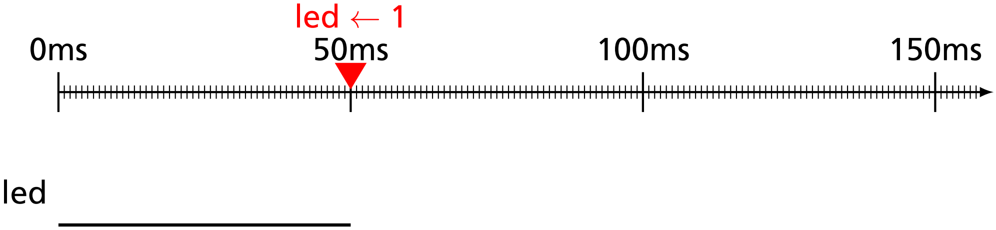
```
blink led =
  loop
    after ms 50,
      led <- not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



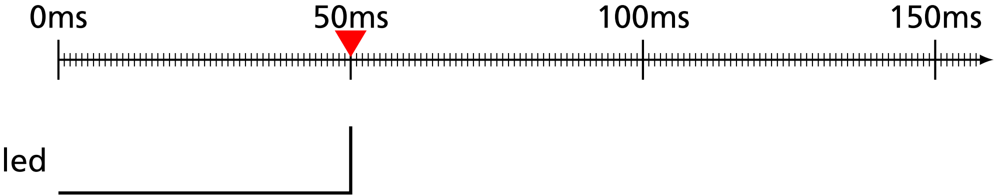
```
blink led =
  loop
    after ms 50,
      led <- not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



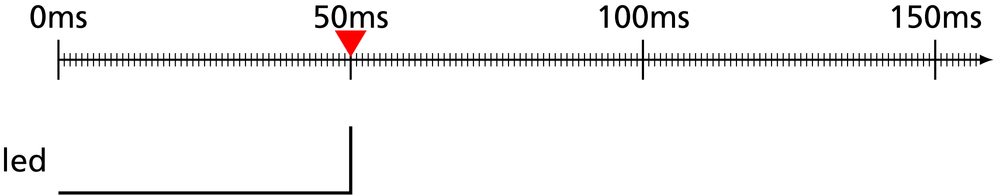
```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



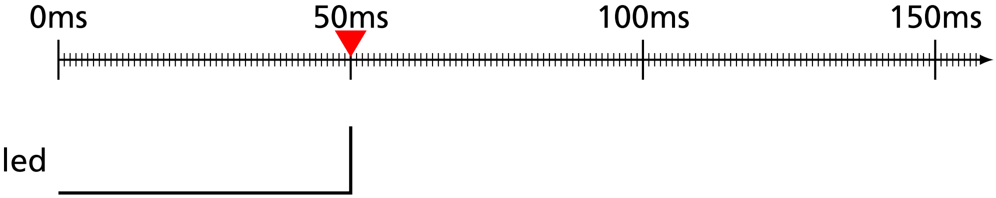
```
blink led =
  loop
    after ms 50,
      led <- not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



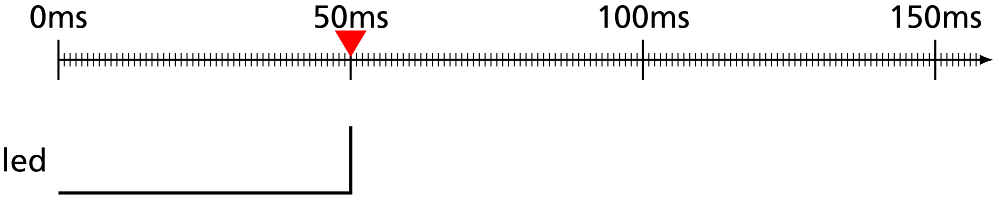
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



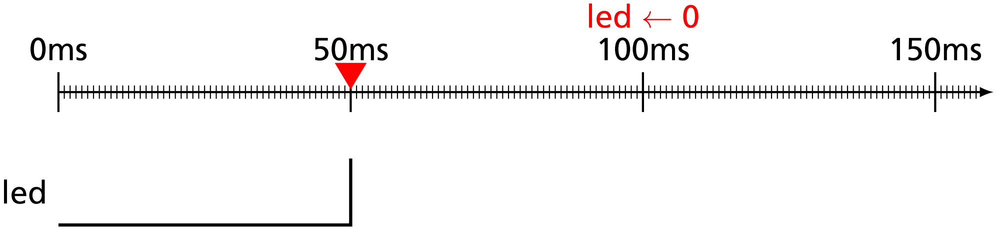
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



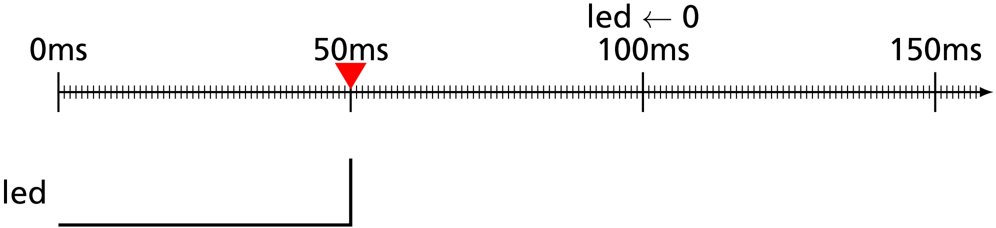
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



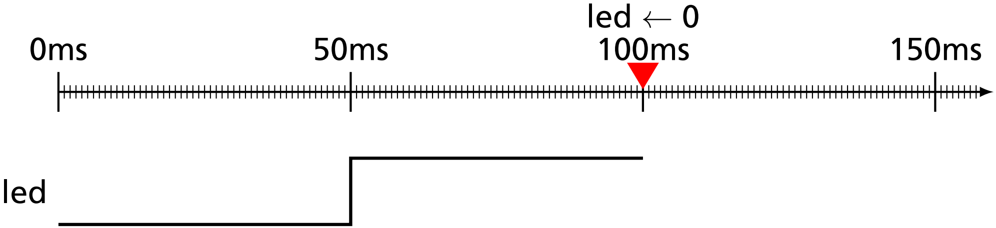
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

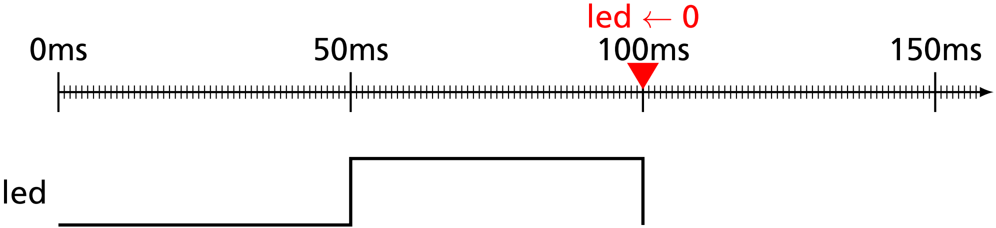
Infinite loop

Schedule a future update

Wait for a write on a variable



blink led =	*led* is mutable; can be scheduled
loop	Infinite loop
after ms 50, led <— not (deref led)	Schedule a future update
wait led	Wait for a write on a variable



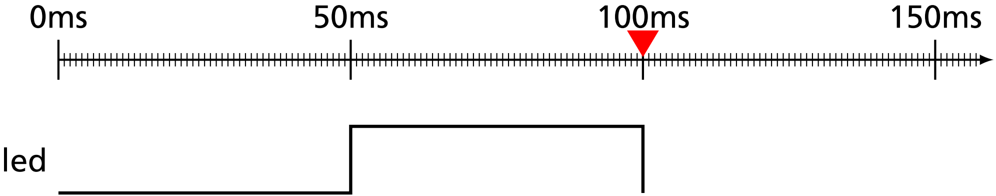
```
blink led =
  loop
    after ms 50,
      led <– not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

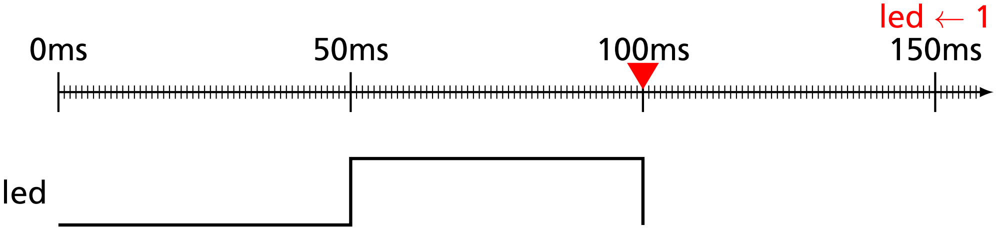
Schedule a future update

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 =

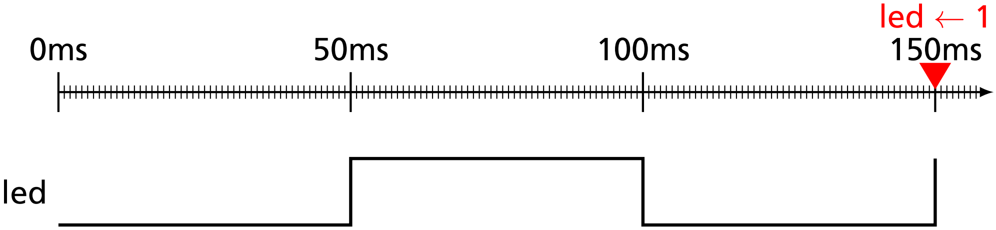
```
blink led =
  loop
    after ms 50,
      led <— not (deref led)
    wait led
```

led is mutable; can be scheduled

Infinite loop

Schedule a future update

Wait for a write on a variable



Concurrent Code Executes in Syntactic Order for Determinism

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

mult4 x = x <- deref a * 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 a * 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 a * 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 a * 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

Second queue for code to be executed in the current instant

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

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

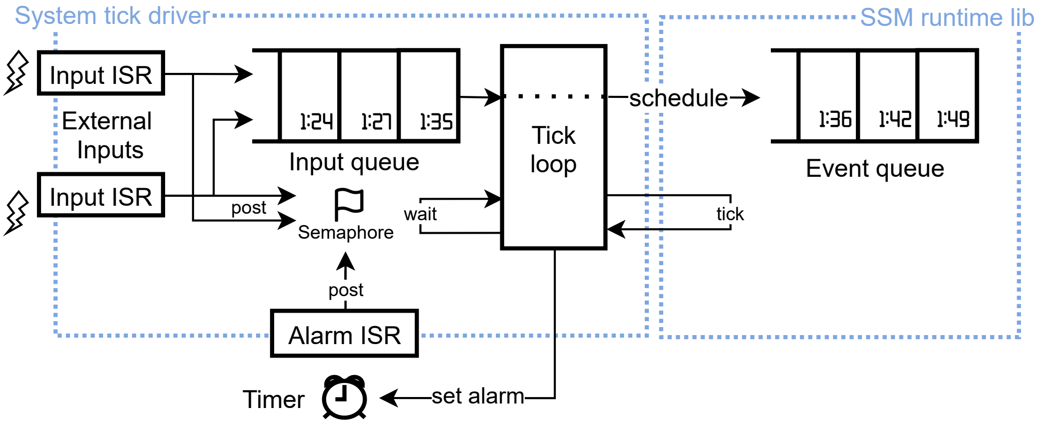
```
rar_examp_t *enter_examp(rar_t *caller, uint32_t priority, unit8_t depth, cv_int_t *a) {
  rar_examp_t *rar = (rar_examp_t *)
     enter(sizeof(rar_examp_t), step_examp, caller, priority, depth);
  rar->a = a;                                    // Store pass-by-reference argument
  rar->trig1.rar = (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
```

MEMOCODE 2022: Scoria: SSM Embedded in Haskell

```
sigGen :: (?out0 :: Ref GPIO) => Ref Word64 -> SSM ()
sigGen hperiod = routine $ while true (do
  after (ns (deref hperiod)) ?out0 (not' (deref ?out0))
  wait ?out0)

remoteControl :: (?ble :: BLE) => Ref Word64 -> SSM ()
remoteControl hperiod = routine $ do
  enableScan ?ble
  while true (do
    wait (scanref ?ble)
    if deref (scanref ?ble) ==. 0
      then hperiod <~ deref hperiod * 2
      else hperiod <~ max' (deref hperiod /. 2) 1)

entry :: (?ble :: BLE, ?out0 :: Ref GPIO) => SSM ()
entry = routine $ do
  hperiod <- var (time2ns (secs 1))
```

MEMOCODE 2022: Timer and Interrupts Drive the Runtime



TCRS 2023: SSM as a Lua Library

```
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

local n = 10
```

MEMOCODE 2023: The RP2040

2 ARM Cortex M0+ processor cores, 133 MHz

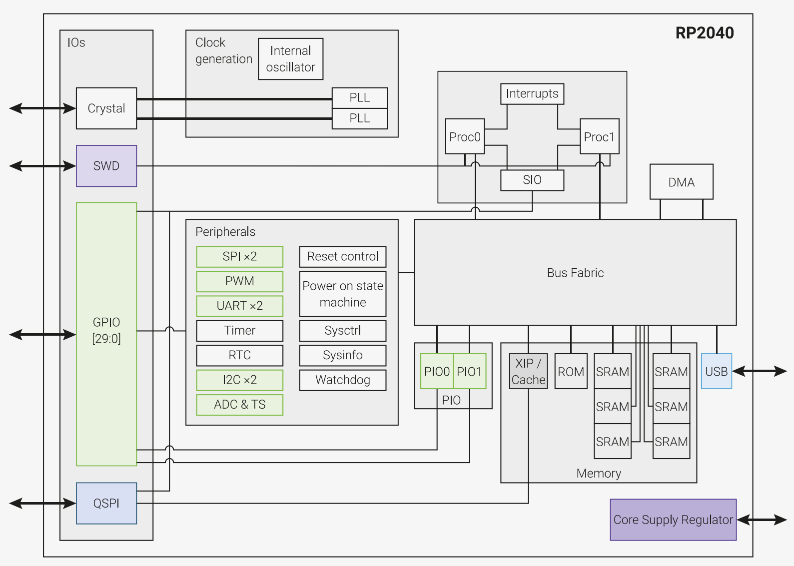
264K SRAM

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

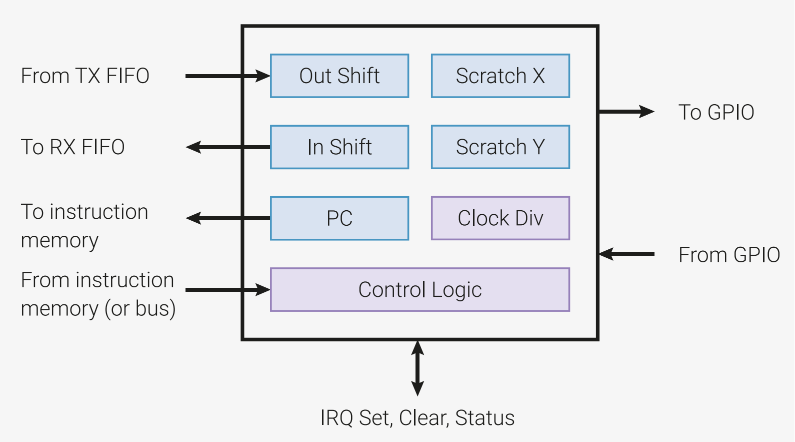
4 “State Machines”

32-instruction memory (shared)

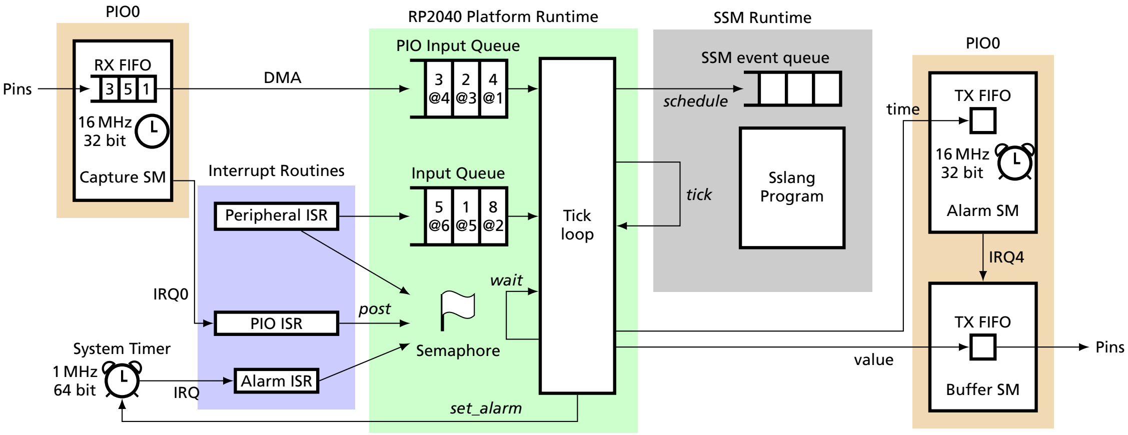
9 instructions (jump, wait, in, out, etc.)

4 32-bit registers

Single-cycle execution

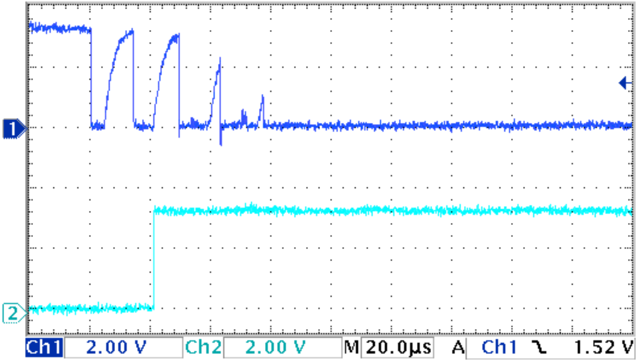


MEMOCODE 2023: Sslang on an RP2040



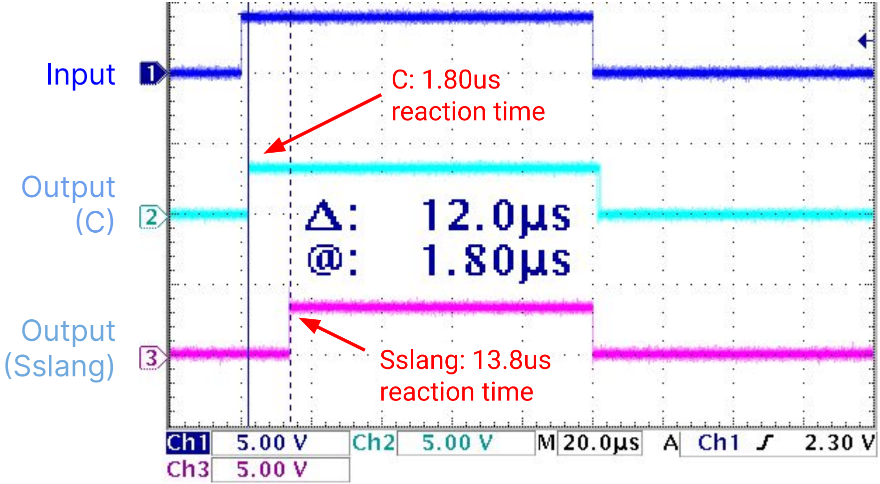
Latency: 10-20 μs Accuracy: 62.5 ns / 16 MHz

```
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
```

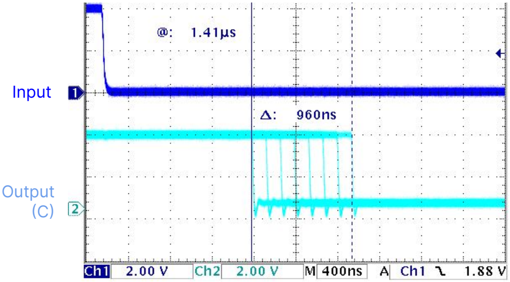


21 μs Button-to-LED latency

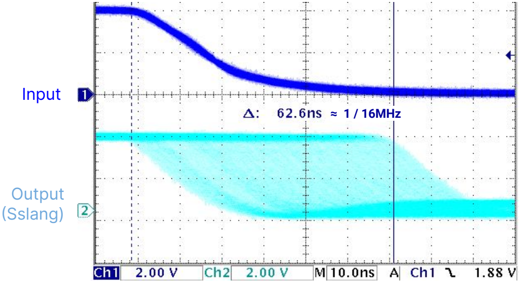
MEMOCODE 2023: 100 µs pulse: C vs Sslang Latency



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)