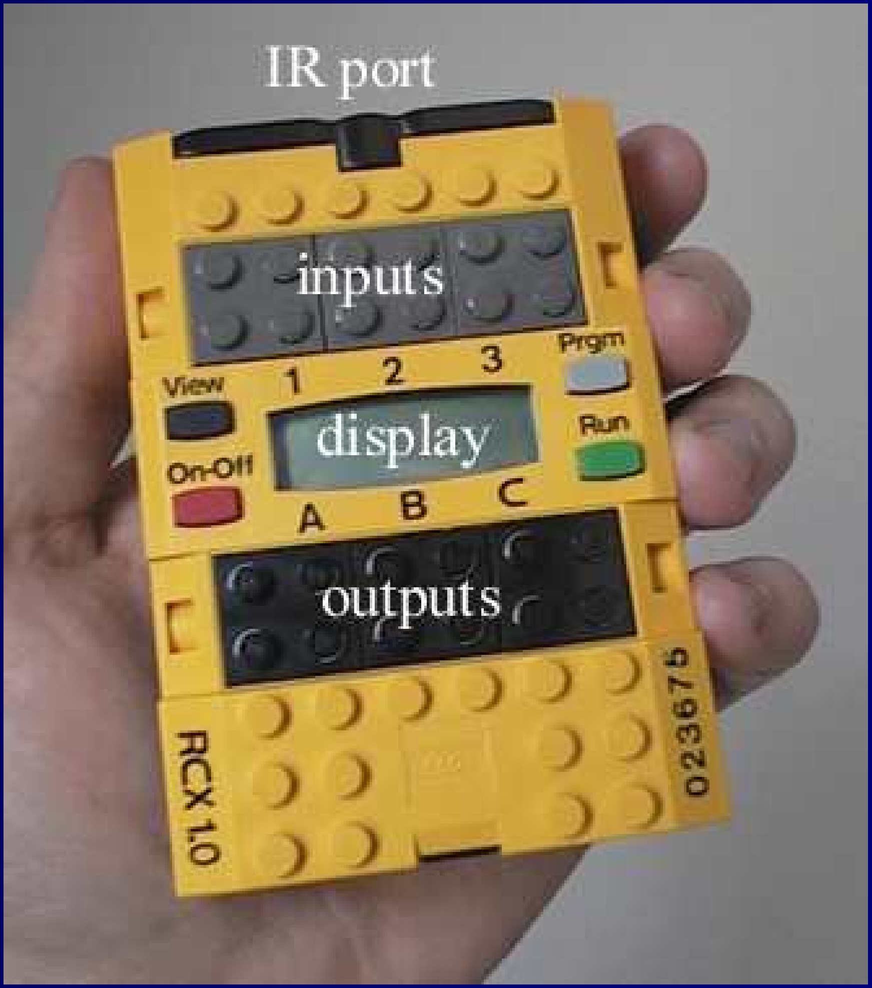
An Esterel Virtual Machine for Embedded Systems

Becky Plummer   Mukul Khajanchi   Stephen A. Edwards

Columbia University

An Esterel Virtual Machine

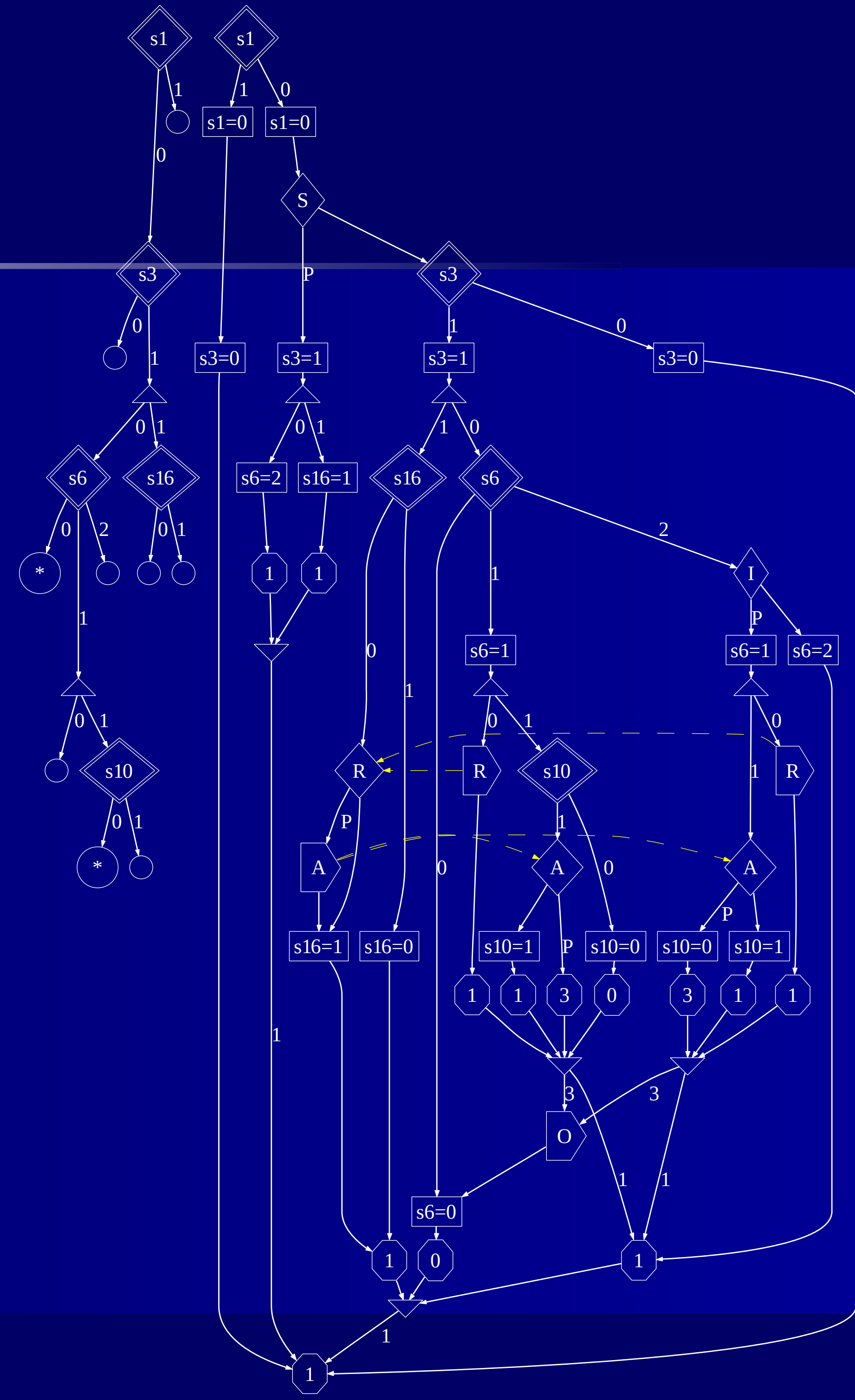


Goal: Run big Esterel programs in memory-constrained settings.

Our target: the Hitachi H8-based RCX Microcontroller for Lego Mindstorms

An Example

```
module Example:
input I, S;
output O;
signal R,A in
  every S do
      await I;
      weak abort
        sustain R
      when immediate A;
      emit O
    ||
      loop
        pause; pause;
        present R then
          emit A
        end present
      end loop
  end every
end signal
end module
```



Challenges

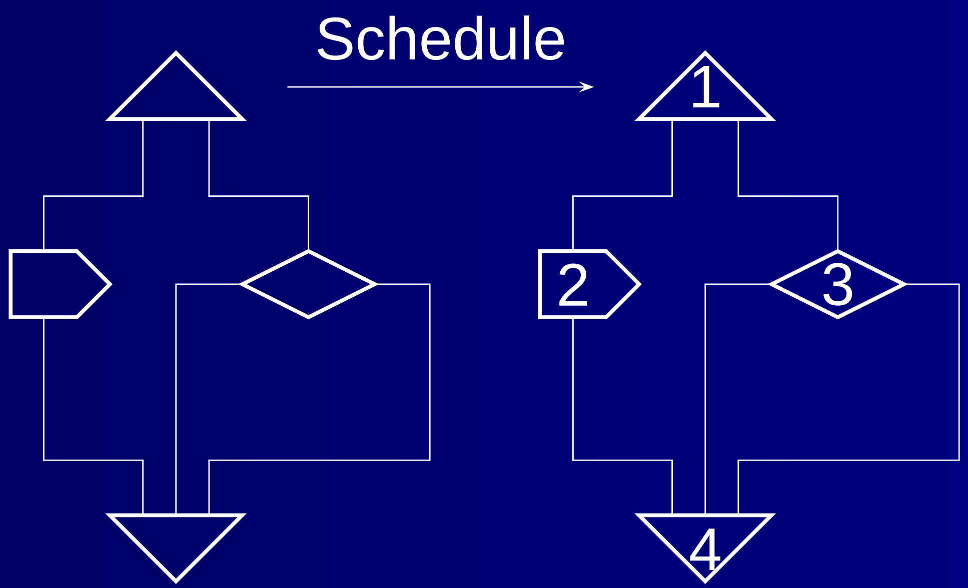
Esterel's semantics require any implementation to deal with three issues:

- Concurrent execution of sequential threads of control within a cycle
- The scheduling constraints among these threads due to communication dependencies
- How control state is updated between cycles

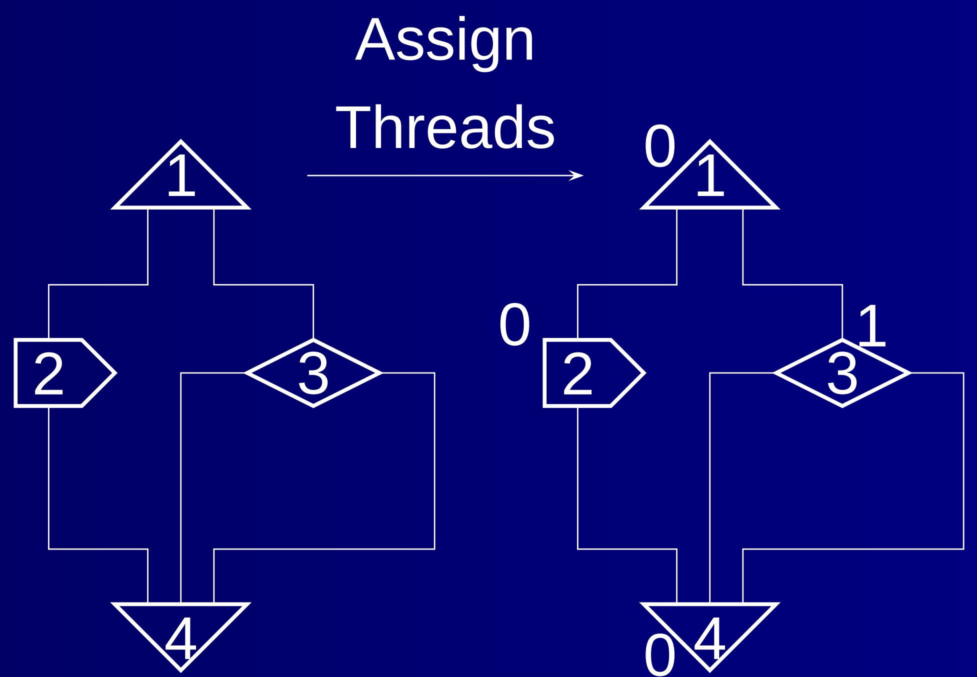
How did we handle them?

- A virtual machine specifically designed to support Esterel features
- A sequentializing algorithm
- Conversion from GRC to BAL and then to a compact byte code

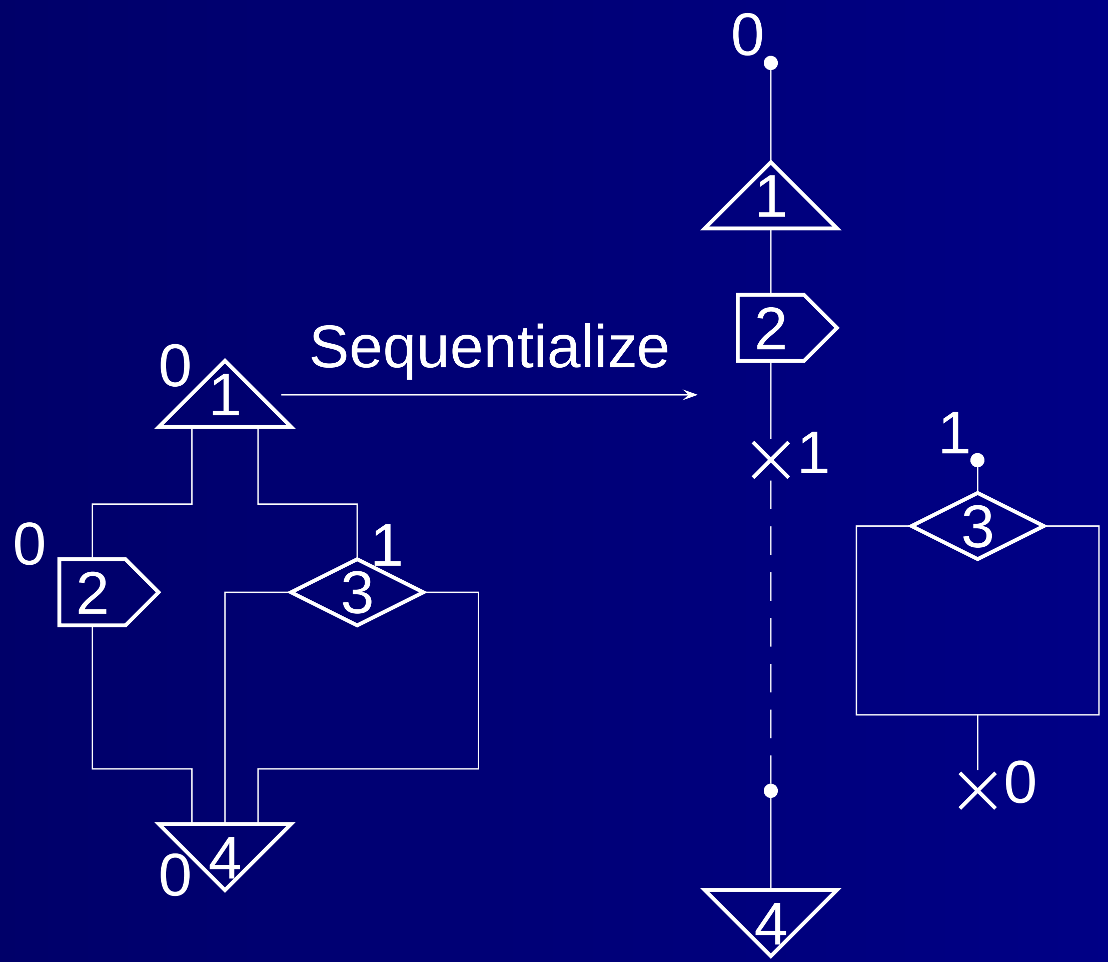
Phase 1: Schedule



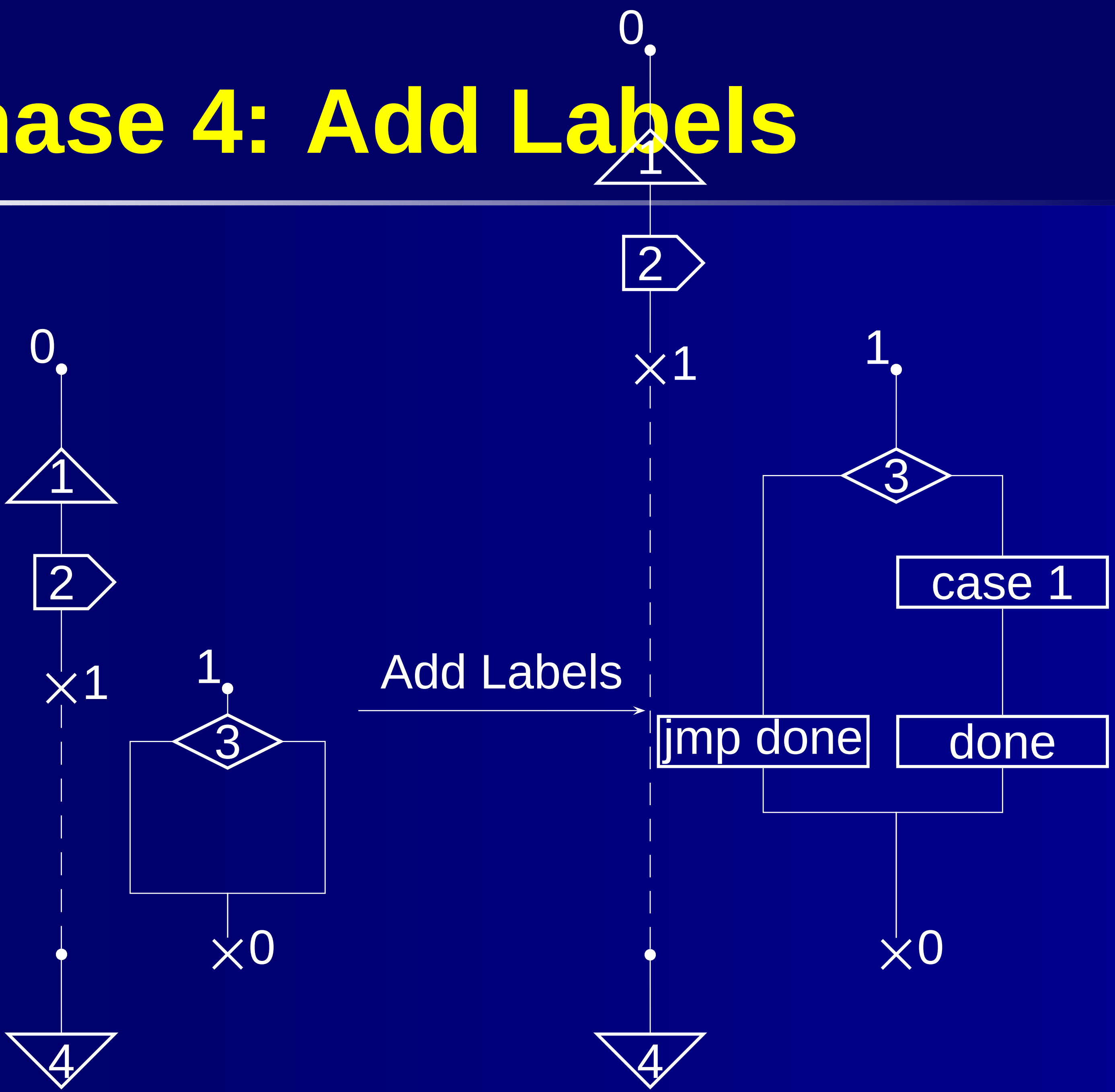
Phase 2: Assign Threads



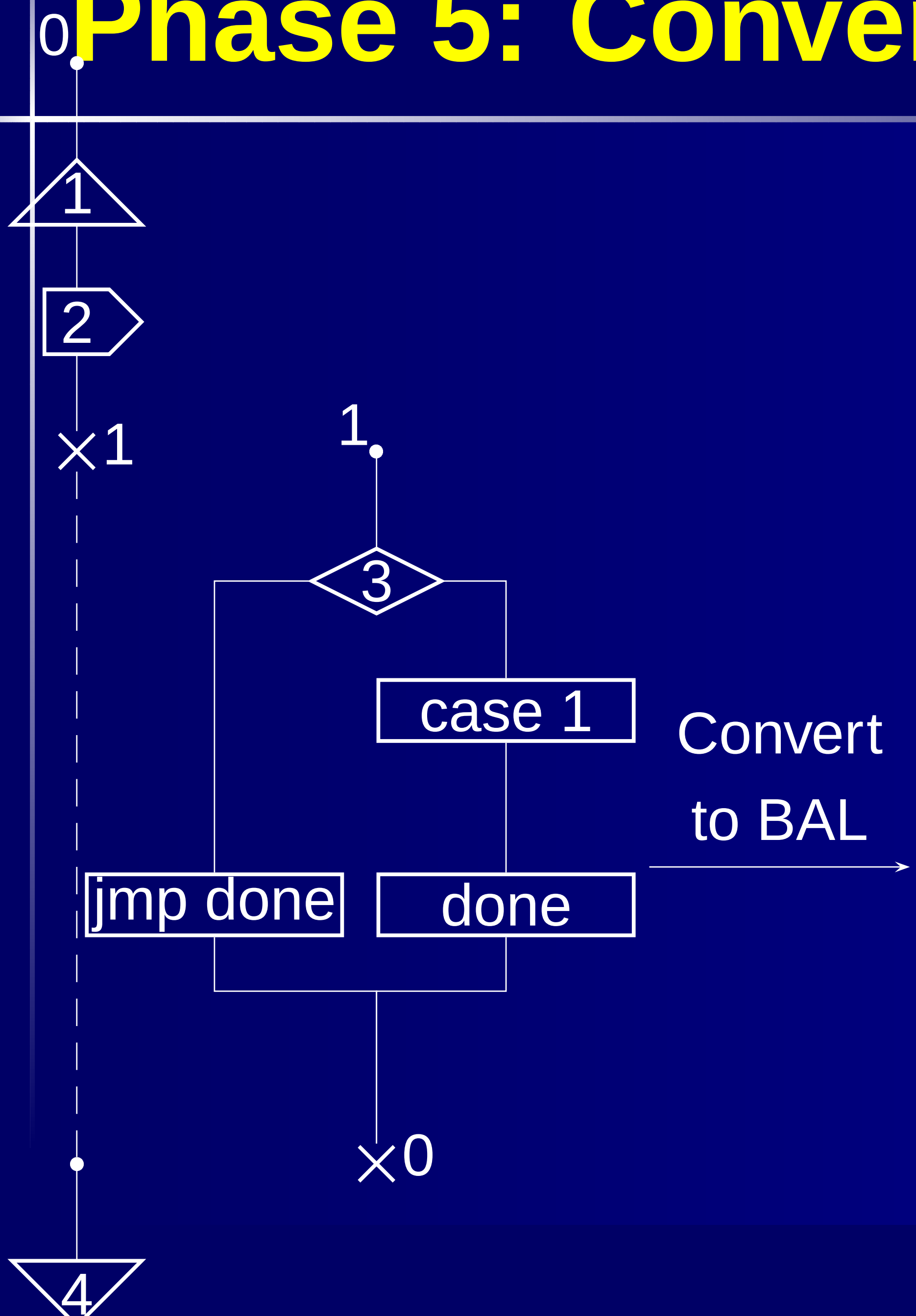
Phase 3: Sequentialize



Phase 4: Add Labels



Phase 5: Convert to BAL




```
t0
    STHR 1 t1
    EMT 1
    SWC 1
    STHR 1 NR1
    END

NR1
    SWCU
t1
    TWB 2 2 case_1
    JMP done
case_1
done
    SWC 0
```

Phase 6: Convert to Byte Code

```
t0
    STHR 1 t1                          07 01 00 0e
    EMT 1                              04 01
    SWC 1                              05 01
    STHR 1 NR1                         07 01 00 0d
    END                                03

NR1
    SWCU                               0c
t1
    TWB 2 2 case_1                     49 02 00 15
    JMP done                           06 00 15
case_1
done
    SWC 0                              05 00
```

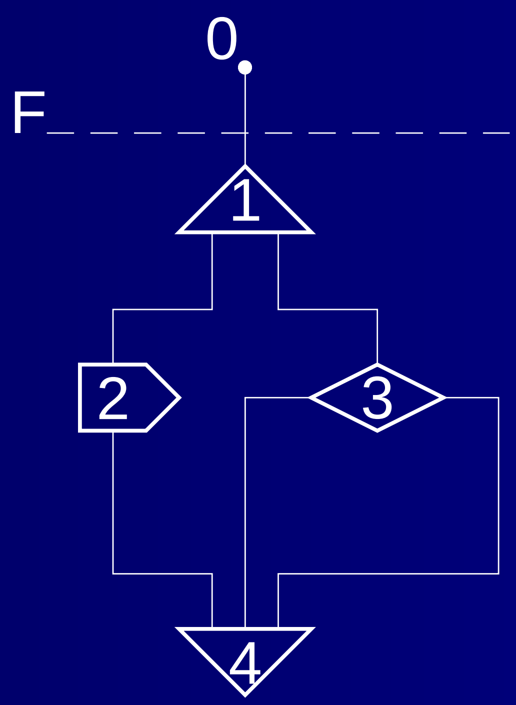
Convert to byte code →

Sequential Code Generation

1. Schedule the nodes in the graph
2. Assign thread numbers
3. Sequentialize the graph
4. Set the execution path by adding labels
5. Convert to BAL
6. Assemble to produce bytecode

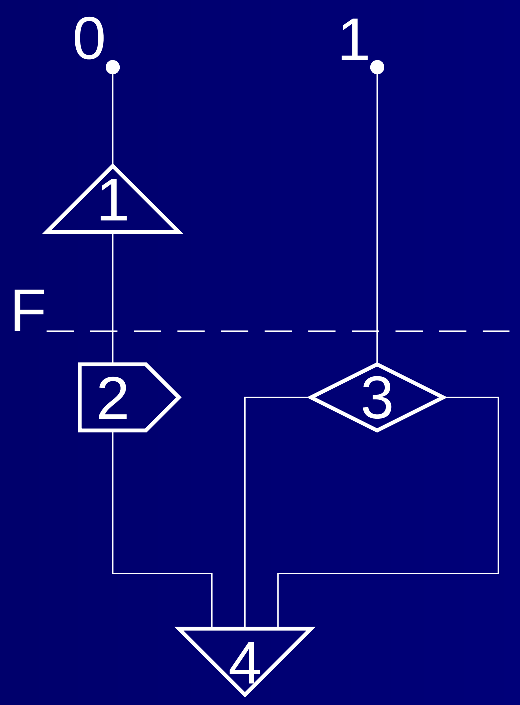
Sequentialization

Sequentialization



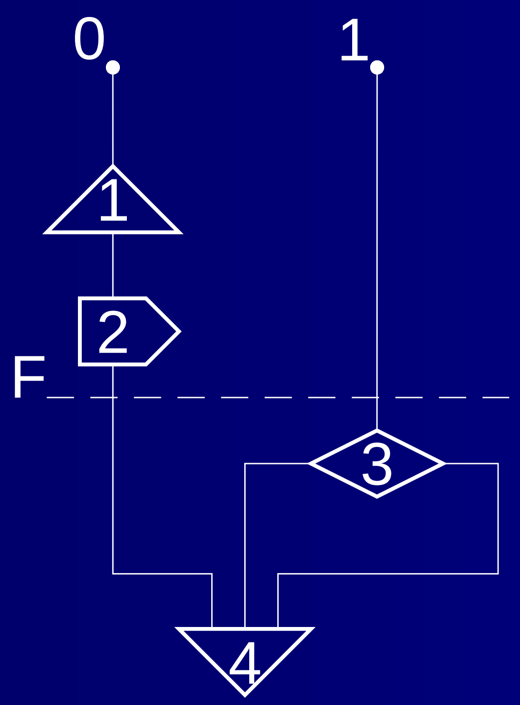
The dotted line labeled F represents the frontier. The frontier starts at the top of the graph.

Sequentialization



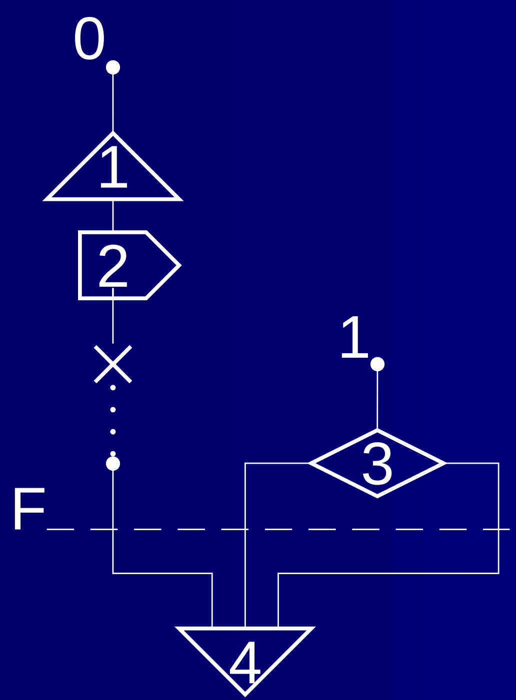
The frontier moves down a node at a time in scheduled order.

Sequentialization



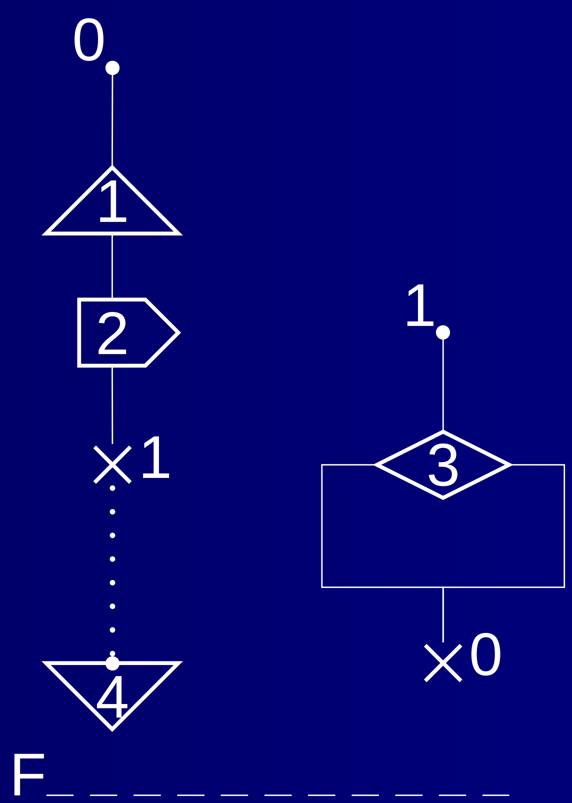
When a node is in the same thread as the most recently moved one, it is simply moved above the frontier.

Sequentialization



However, when the next node is from a different thread, a switch is added to the previous thread and an active point is added to the new thread just above the just-moved node.

Sequentialization



The algorithm is complete when the frontier has swept across all nodes in scheduled order.

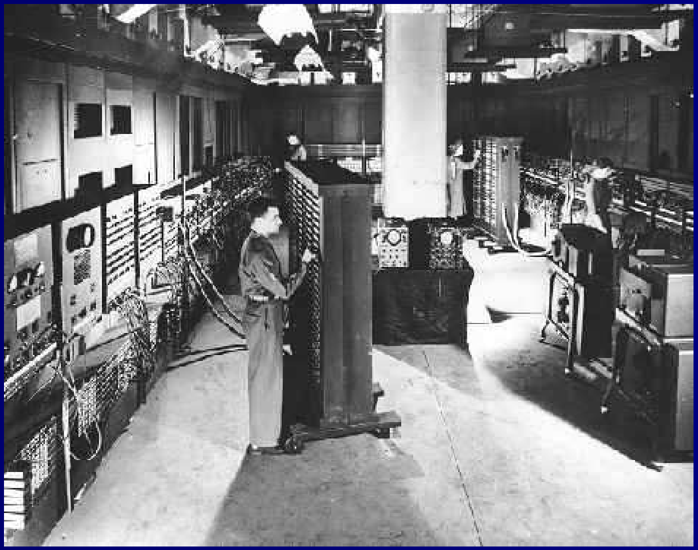
Sequentializing Algorithm

```
for each thread t in G do
    create new active point p
    copy first node n of t in G to n' new node in G'
    connect p and n'
    add p to P[t] and add n' to A[t]
t' = the first thread
for each node n in scheduled order do
    t is thread of n
    if t ≠ t' then
        for each parent p in P[t'] do
            for each successor c of p in A[t'] do
                create switch node s from t' to t and connect s between p and c
        replace P[t'] with the set of new switch nodes
    move n to P[t] and remove it from A[t]
    for each unreached successor c of n do
        copy c to c' new node in G'
        if n is a fork then
            add child to A[thread of c]
        else
            add child to A[t]
    t' = t {remember the last thread}
```

Why VM?

- Goal: constrained-memory environment
- Instruction set has direct support for Esterel constructs like concurrency, preemption, and signals
- E.g., a context switch can be specified in just two bytes

VM Details



VM Details

- Signal status registers
- Completion code registers
- Per-thread program counters
- Inter-instant state-holding registers

VM: Signal, State, and Thread

Opcode	Description	Encoding
EMT	Emit a Signal	04 RR
SSIG	Clear Signal	0A RR
SSTT	Set State	0B RR VV
STHR	Set Thread	07 TT HH LL

VM: Control Flow Instructions

Opcode	Description	Encoding
END	Tick End	03
JMP	Jump	06 HH LL
NOP	No Operation	01

VM: Branch, Switch, Terminate

Opcode	Description	Encoding
MWB	Multiway Branch (State)	2D NL RR HH2 LL2 ...
	Multiway Branch (Comp.)	4D NL RR HH2 LL2 ...
TWB	Two Way Branch (State)	29 RR HH LL
	Two Way Branch (Signal)	49 RR HH LL
	Two Way Branch (Comp.)	69 RR HH LL
SWC	Switch Thread	05 TT
SWCU	Switch Unknown	0C
TRM	Set Completion Code for Join	08 RR VV

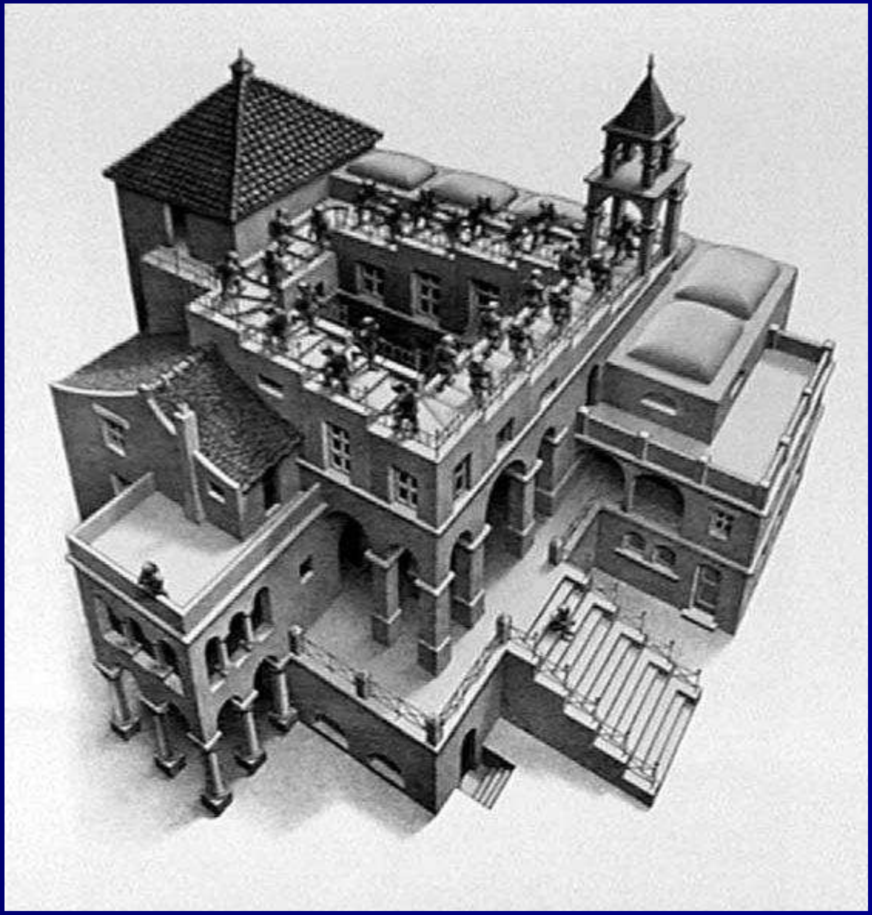
VM: Context Switch

```
 ...
switch(opcode & 0x1F){
    ...
    case SWC:
        // Increment the program counter
        ++pc;
        // Store the current thread as the last thread
        last_thread = current_thread;
        // Get the next thread
        current_thread = *pc;
        // Increment the program counter
        ++pc;
        // Store old pc associated with the old thread
        threads[last_thread] = pc;
        // Load the pc associated with the new thread
        pc = threads[current_thread];
        break;
    ...
```

VM: Switch Unknown

```
...
case SWCU:
    // Make the thread stored in last_thread, the current thread
    temp = current_thread;
    current_thread = last_thread;
    last_thread = temp;
    // Store old pc
    threads[last_thread] = pc;
    // Load new pc
    pc = threads[current_thread];
    break;
...
```

VM in action



VM in action

```
    t0
00:    STHR 1 t1
04:    EMT 1
06:    SWC 1
08:    STHR 1 NR1
12:    SWC 1
14:    END
    NR1
15:    SWCU
    t1
16:    TWB 2 1 case_1
19:    JMP done
    case_1
    done
22:    SWC 0
```

pc = 0

last_thread = 0

Threads	Signals
0	0
0	0

States	Joins
..	..

VM in action

```
     t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
     NR1
15:     SWCU
     t1
16:     TWB 2 1 case_1
19:     JMP done
     case_1
     done
22:     SWC 0
```

pc = 4

last_thread = 0

Threads	Signals
0	0
16	0

States	Joins
..	..

VM in action

```
    t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
    NR1
15:     SWCU
    t1
16:     TWB 2 1 case_1
19:     JMP done
    case_1
    done
22:     SWC 0
```

pc = 6

last_thread = 0

Threads	Signals
0	0
16	1

States	Joins
..	..

VM in action

```
     t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
     NR1
15:     SWCU
     t1
16:     TWB 2 1 case_1
19:     JMP done
     case_1
     done
22:     SWC 0
```

pc = 16

last_thread = 0

Threads	Signals
8	0
16	1

States	Joins
..	..

VM in action

```
      t0
00:      STHR 1 t1
04:      EMT 1
06:      SWC 1
08:      STHR 1 NR1
12:      SWC 1
14:      END
      NR1
15:      SWCU
      t1
16:      TWB 2 1 case_1
19:      JMP done
      case_1
      done
22:      SWC 0
```

pc = 19

last_thread = 0

Threads	Signals
8	0
16	1

States	Joins
..	..

VM in action

```
     t0
00:      STHR 1 t1
04:      EMT 1
06:      SWC 1
08:      STHR 1 NR1
12:      SWC 1
14:      END
     NR1
15:      SWCU
     t1
16:      TWB 2 1 case_1
19:      JMP done
     case_1
     done
22:      SWC 0
```

pc = 22

last_thread = 0

Threads	Signals
8	0
16	1

States	Joins
..	..

VM in action

```
      t0
00:      STHR 1 t1
04:      EMT 1
06:      SWC 1
08:      STHR 1 NR1
12:      SWC 1
14:      END
      NR1
15:      SWCU
      t1
16:      TWB 2 1 case_1
19:      JMP done
      case_1
      done
22:      SWC 0
```

pc = 8

last_thread = 1

Threads	Signals
8	0
24	1

States	Joins
..	..

VM in action

```
    t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
    NR1
15:     SWCU
    t1
16:     TWB 2 1 case_1
19:     JMP done
    case_1
    done
22:     SWC 0
```

pc = 12

last_thread = 1

Threads	Signals
8	0
15	1

States	Joins
..	..

VM in action

```
     t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
     NR1
15:     SWCU
     t1
16:     TWB 2 1 case_1
19:     JMP done
     case_1
     done
22:     SWC 0
```

pc = 15

last_thread = 0

Threads	Signals
14	0
15	1

States	Joins
..	..

VM in action

```
    t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
    NR1
15:     SWCU
    t1
16:     TWB 2 1 case_1
19:     JMP done
    case_1
    done
22:     SWC 0
```

pc = 14

last_thread = 1

Threads	Signals
14	0
15	1

States	Joins
..	..

VM in action

```
     t0
00:     STHR 1 t1
04:     EMT 1
06:     SWC 1
08:     STHR 1 NR1
12:     SWC 1
14:     END
     NR1
15:     SWCU
     t1
16:     TWB 2 1 case_1
19:     JMP done
     case_1
     done
22:     SWC 0
```

pc = 15

last_thread = 0

Threads	Signals
15	0
15	1

States	Joins
..	..

The engineering details

- brickOS 2.6.10 on Redhat Linux
- gcc cross compiler 4.0.2. for H8300
- Download lx files to the lego RCX via USB IR tower

Code Sizes

Example	BAL	x86		H8	
dacexample	369	917	60%	842	57%
abcd	870	2988	71%	2648	68%
greycounter	1289	3571	64%	2836	55%
tcint	5667	11486	51%	10074	51%
atds-100	10481	38165	73%	26334	60%

BAL: the size of our bytecode (in bytes)

x86: the size of optimized C code for an x86

H8: the size of optimized C code for an Hitachi H8

Percentages represent the size savings of using bytecode.

Execution Times

Example	x86	BAL	
dacexample	0.06μs	1.1μs	$18\times$
tcint	0.28μs	1.1μs	$4\times$
atds-100	0.20μs	1.4μs	$7\times$

Future Work

- Arithmetic Support
- Support for externally-called functions

Conclusions

- Simple Virtual Machine
- Compilation scheme statically schedules the concurrent behavior and generates straight-line code for each thread
- VM supports context-switching well
- Bytecode for our virtual machine is roughly half the size of optimized native assembly code generated from C
- Speed tradeoff not that bad! Between 4 and 7 times slower than optimized C code