

Compiling Esterel into Better Circuits and Faster Simulations

Stephen A. Edwards

Department of Computer Science,
Columbia University

www.cs.columbia.edu/~sedwards

sedwards@cs.columbia.edu

Esterel for Hardware Specification

Why consider Esterel?

- Semantics more abstract than RTL
More succinct descriptions faster and easier to write
- High-level semantics enable high-level optimizations
State assignment a hierarchical problem
- High-level semantics enable more efficient simulation
Semantics are more like an imperative program
- Esterel's semantics are deterministic
Simulation-synthesis mismatches eliminated

Applications of Esterel

Systems with complex (non-pipelined) control-behavior:

- DMA controllers
- Cache controllers
- Communication protocols

(Not processors)

Verilog More Verbose Than Esterel

```
case (cur_state) // synopsys parallel_case
  IDLE: begin
    if (pcsu_powerdown & !jmp_e &
        !valid_diag_window) begin
      next_state = STANDBY_PWR_DN;
    end
    else if (valid_diag_window | ibuf_full |
             jmp_e) begin
      next_state = cur_state;
    end
    else if (icu_miss & !cacheable) begin
      next_state = NC_REQ_STATE;
    end
    else if (icu_miss & cacheable) begin
      next_state = REQ_STATE;
    end
    else next_state = cur_state;
  end
  NC_REQ_STATE: begin
    if (normal_ack | error_ack) begin
      next_state = IDLE;
    end
    else next_state = cur_state;
  end
  REQ_STATE: begin
    if (normal_ack) begin
      next_state = FILL_2ND_WD;
    end
    else if (error_ack) begin
      next_state = IDLE;
    end
    else next_state = cur_state;
  end
  FILL_2ND_WD: begin
    if (normal_ack) begin
      next_state = REQ_STATE2;
    end
    else if (error_ack) begin
      next_state = IDLE;
    end
    else next_state = cur_state;
  end
  REQ_STATE2: begin
    if (normal_ack) begin
      next_state = FILL_4TH_WD;
    end
    else if (error_ack) begin
      next_state = IDLE;
    end
    else next_state = cur_state;
  end
  FILL_4TH_WD: begin
    if (normal_ack | error_ack) begin
      next_state = IDLE;
    end
    else next_state = cur_state;
  end
  STANDBY_PWR_DN: begin
    if (!pcsu_powerdown | jmp_e) begin
      next_state = IDLE;
    end
    else next_state = STANDBY_PWR_DN;
  end
  default: next_state = 7'bx;
endcase
```

```
loop
  await
  case [ic_u_miss and
        not cacheable] do
    await [normal_ack or error_ack]
  end
  case [ic_u_miss and
        cacheable] do
    abort
    await 4 normal_ack;
    when error_ack
  end
  case [pcsu_powerdown and
        not jmp_e and
        not valid_diag_window] do
    await [pcsu_powerdown and
           not jmp_e]
  end
end;
pause
end
```

Why is Esterel More Succinct?

Verilog:

```
REQ_STATE2: begin
  if(normal_ack) begin
    next_state = FILL_4TH_WD;
  end
  else if (error_ack) begin
    next_state = IDLE ;
  end
  else next_state = cur_state;
end
```

Esterel:

```
abort
  await normal_ack
when error_ack
```

- Esterel provides cross-clock control-flow
- State machine logic represented implicitly
- Higher-level constructs like *await*



Generating Fast Circuits

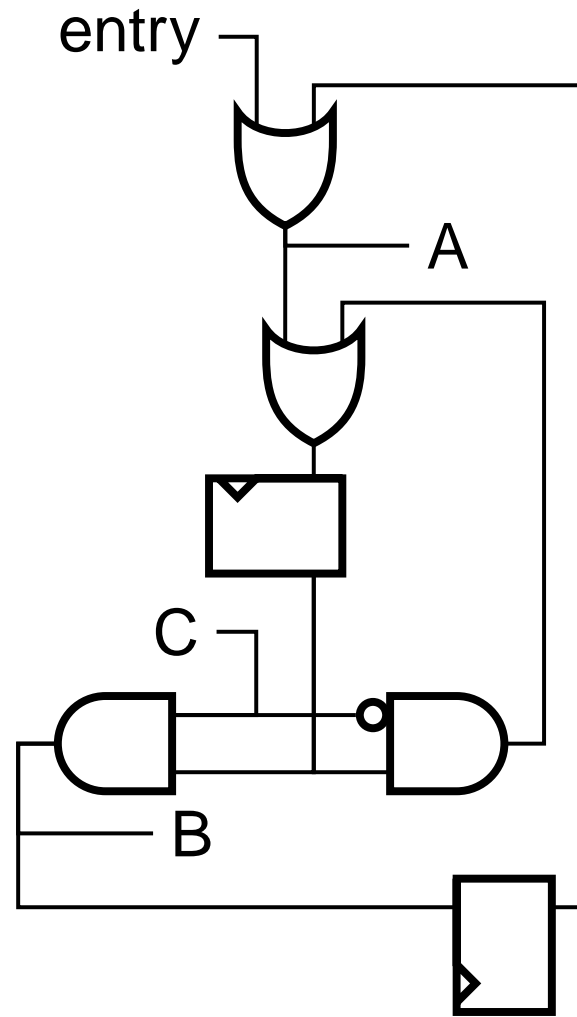
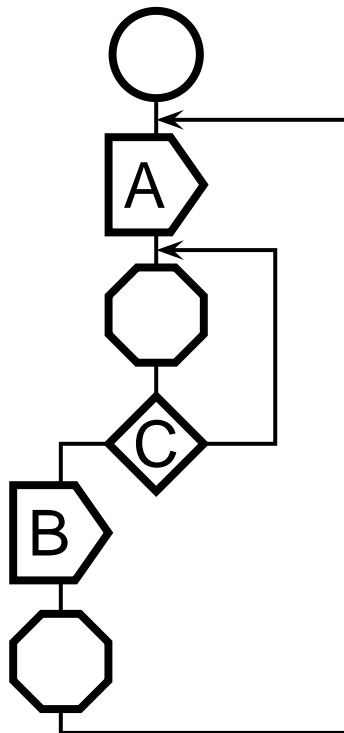
Basic Circuit Generation

loop

emit A; await C;

emit B; pause

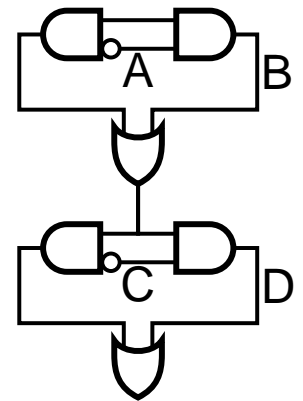
end



Basic Circuit Generation

Berry's technique [1992] works, but is fairly inefficient:

- Many combinational redundancies. E.g.,
`present A then emit B end;`
`present C then emit D end`
produces two redundant OR gates



- Many sequential redundancies

One flop per pause can be very wasteful

Touati, Toma, Sentovich, and Berry [1993–1997] proposed techniques to eliminate many, but requires reachable state space and only works on circuit.

Generating Fast Circuits

Esterel's semantics match hardware. Translation is straightforward.

Nice feature: state space is well-defined and hierarchical (e.g., due to abort and concurrency).

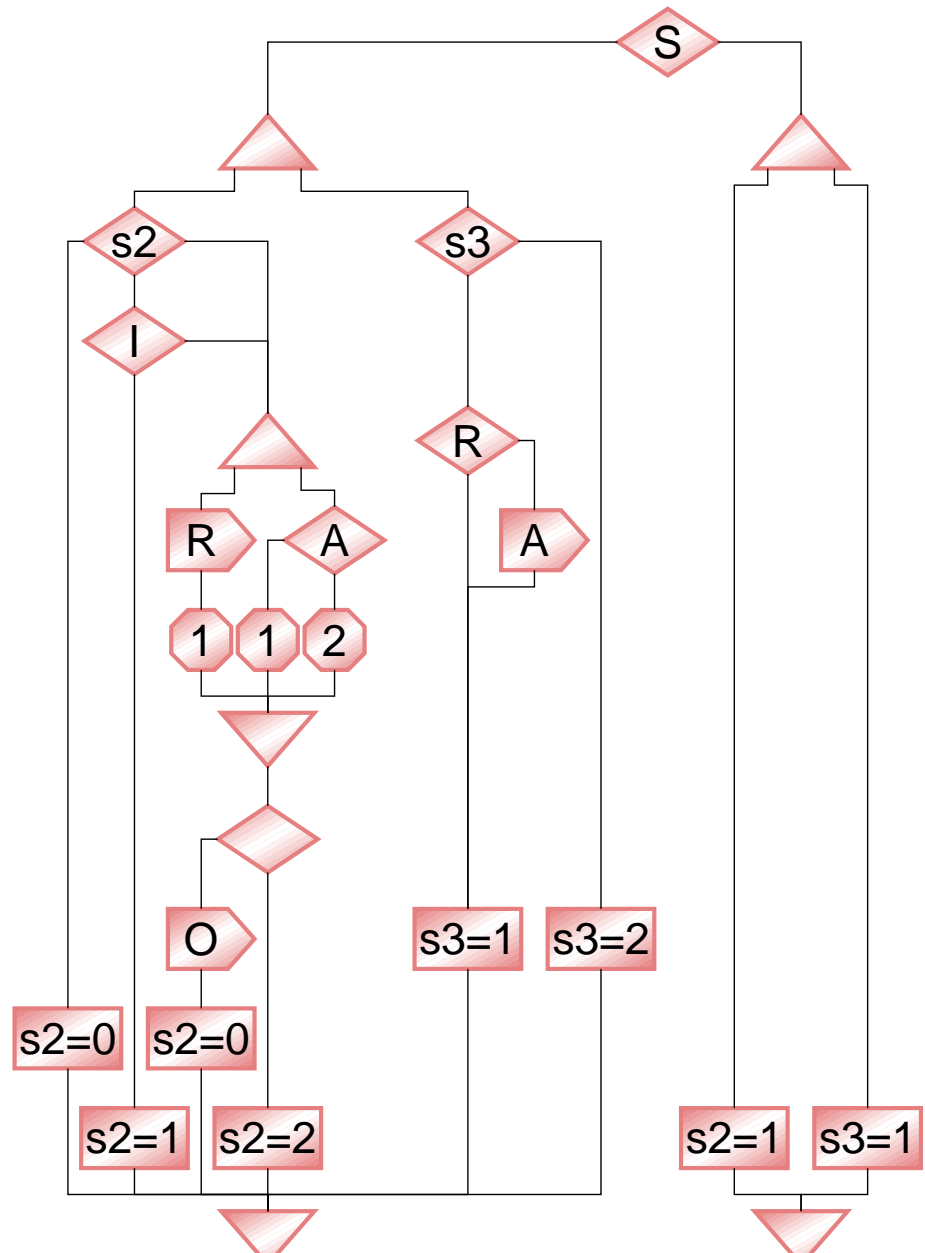
Enables a hierarchical state assignment/synthesis procedure.

Translation to CCFG

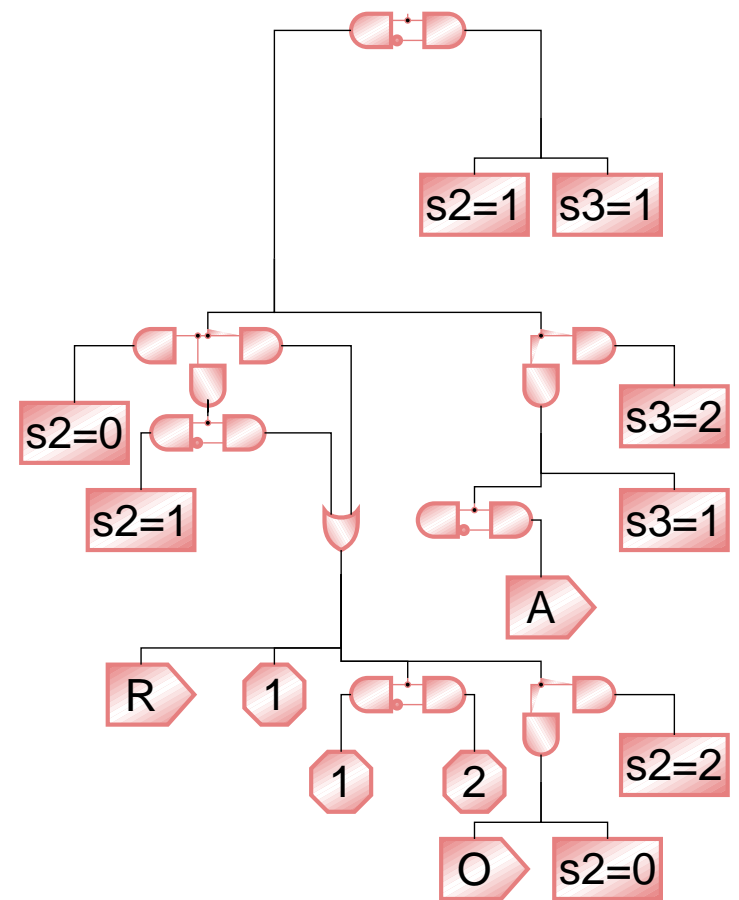
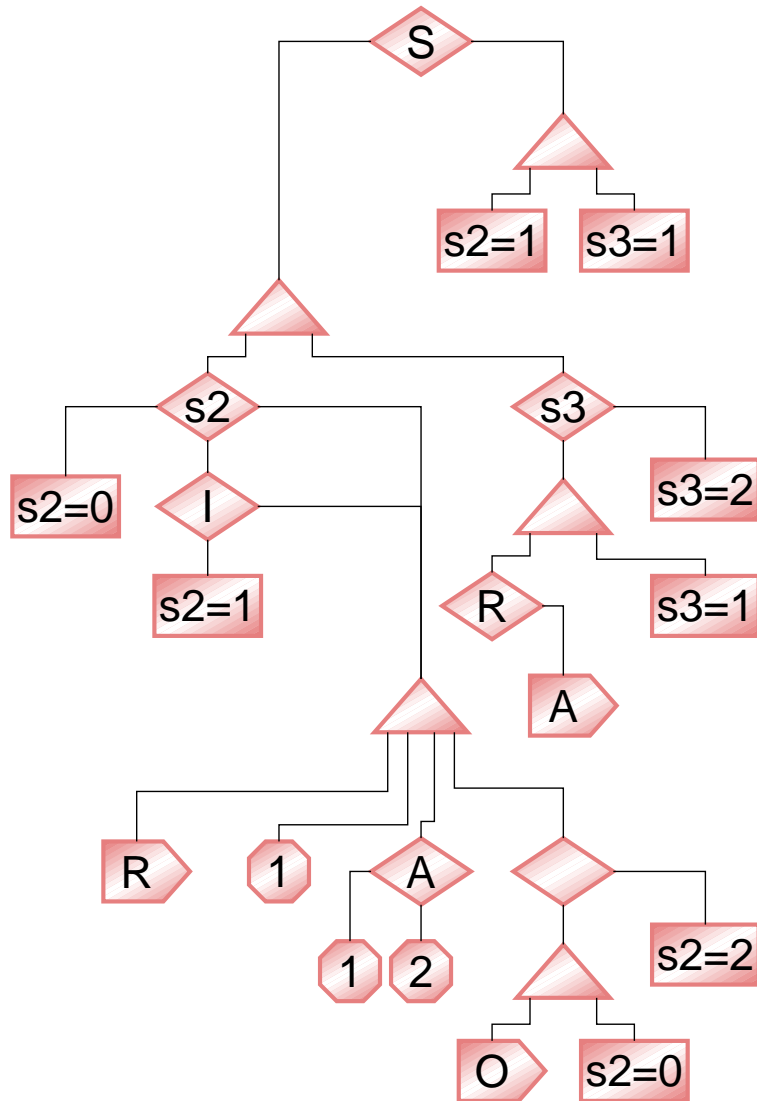
```

every S do
  loop
    await I;
    weak abort
      sustain R
    when immediate A;
    emit O
  end
||
  loop
    pause; pause;
    present R then
      emit A
    end
  end
end
end

```



Translation to Circuitry



A State Assignment Example

```
abort
  [
    await A; await B
    ||
    await C
  ]
when D;
emit E;
pause;
[
  await F
  ||
  await G
]
```

Hierarchical States

```
abort
```

```
[
```

```
  await A;
```

```
  await B
```

```
  ||
```

```
  await C
```

```
]
```

```
when D;
```

```
emit E;
```

```
pause;
```

```
[
```

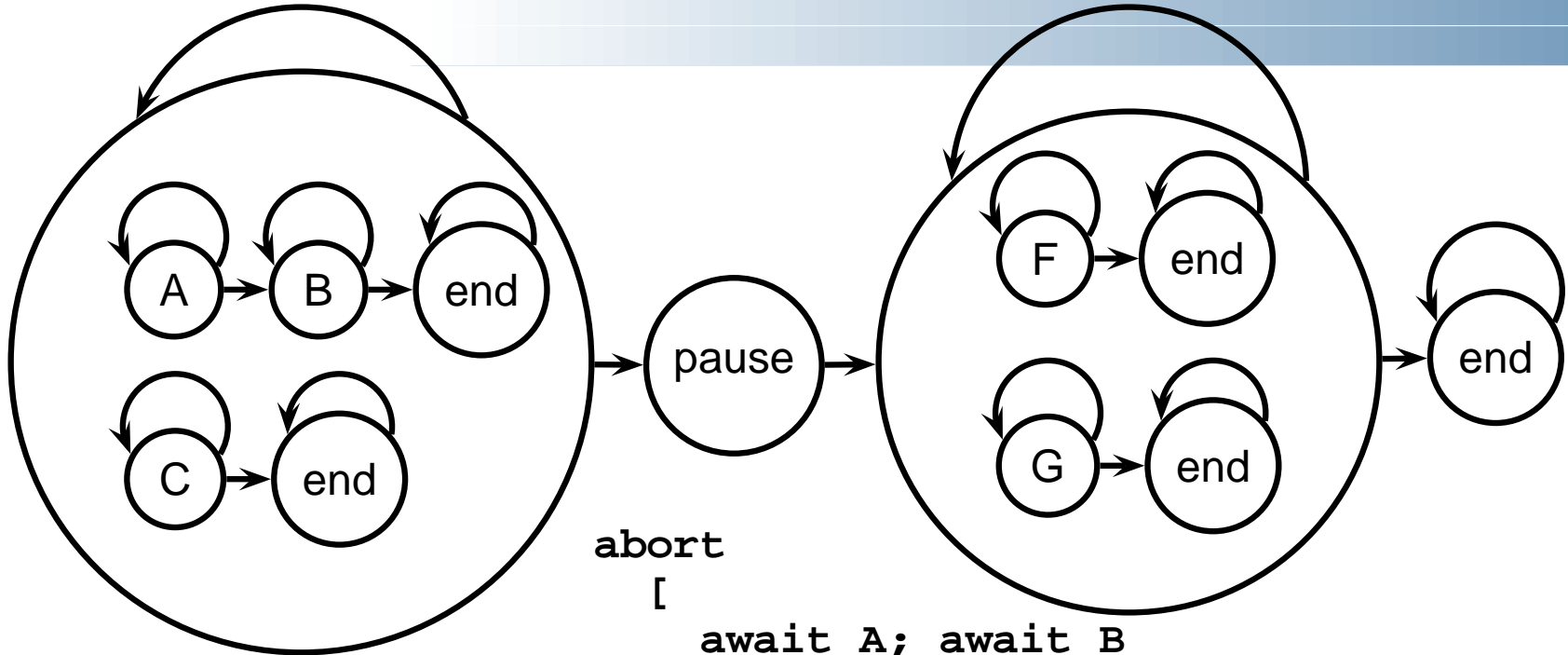
```
  await F
```

```
  ||
```

```
  await G
```

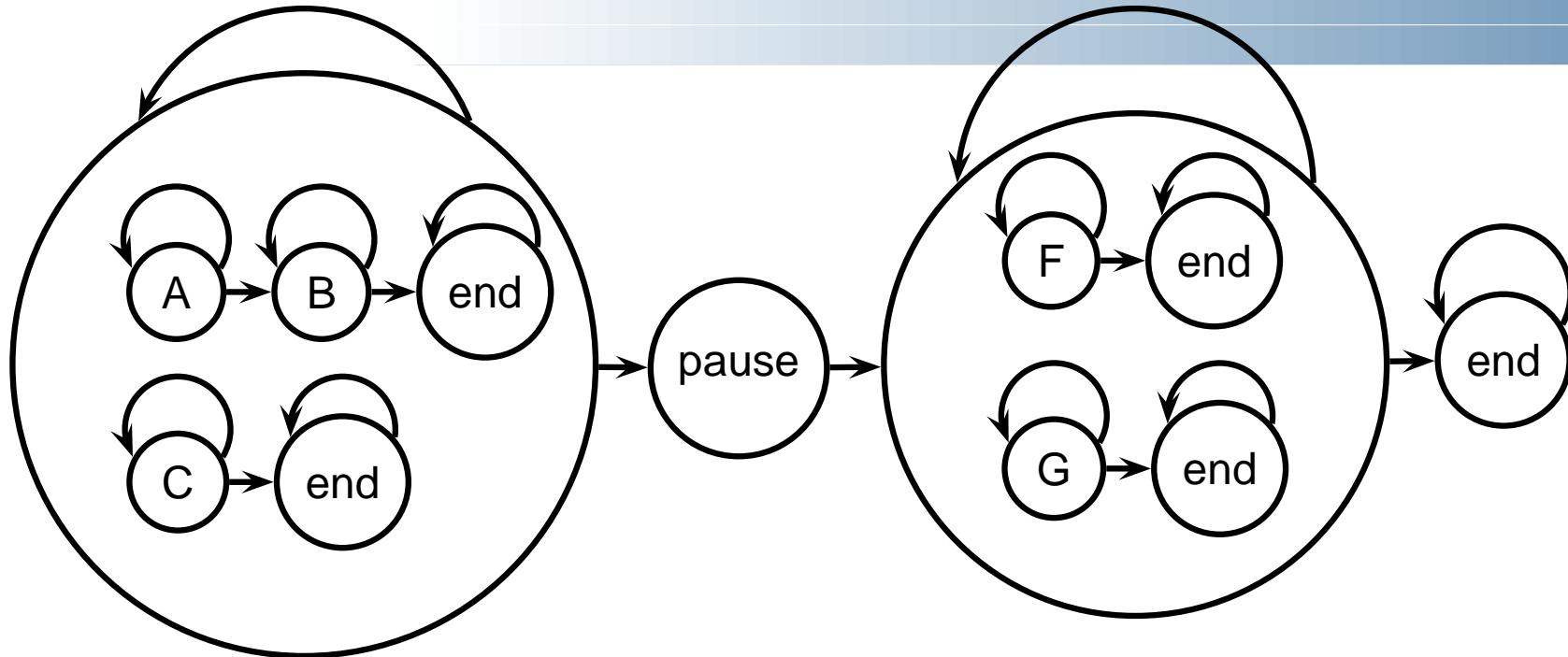
```
]
```

Five Simple FSMs



```
abort
[
  await A; await B
  ||
  await C
]
when D;
emit E;
pause;
[
  await F
  ||
  await G
]
```

Five Simple FSMs

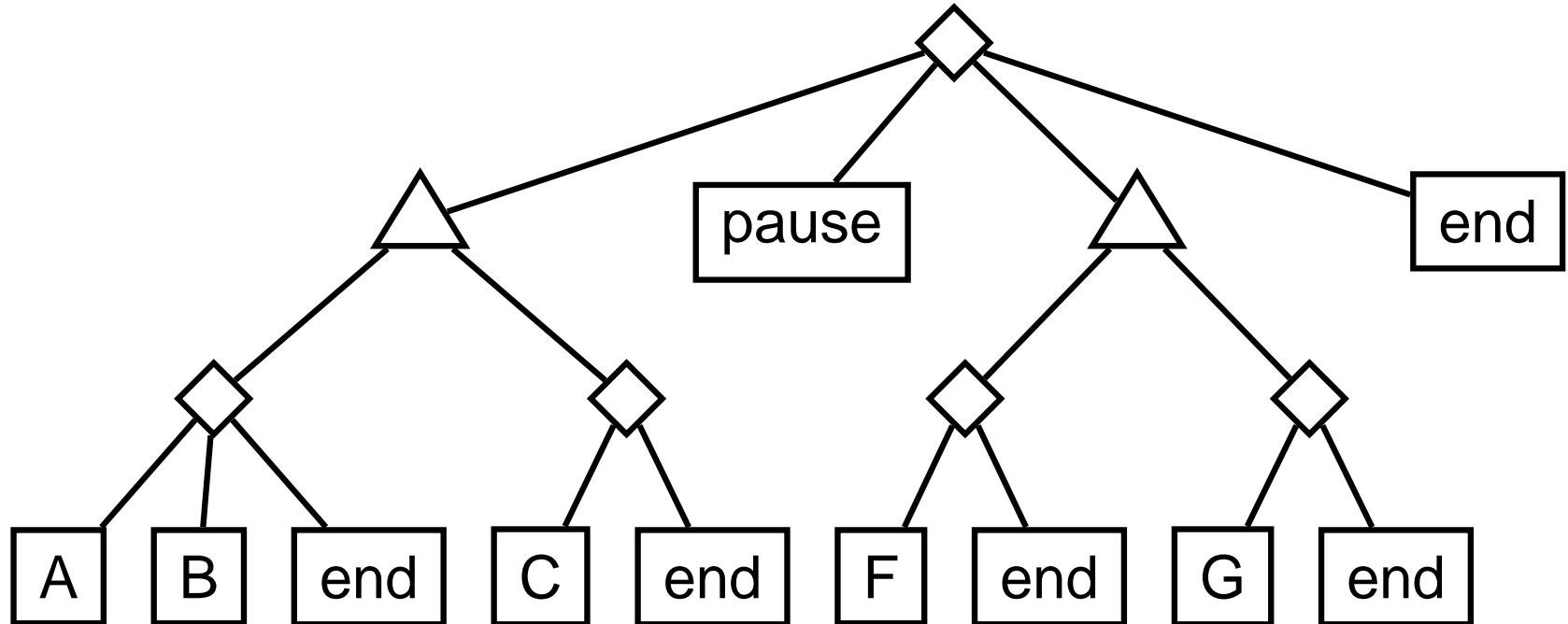


Obvious questions:

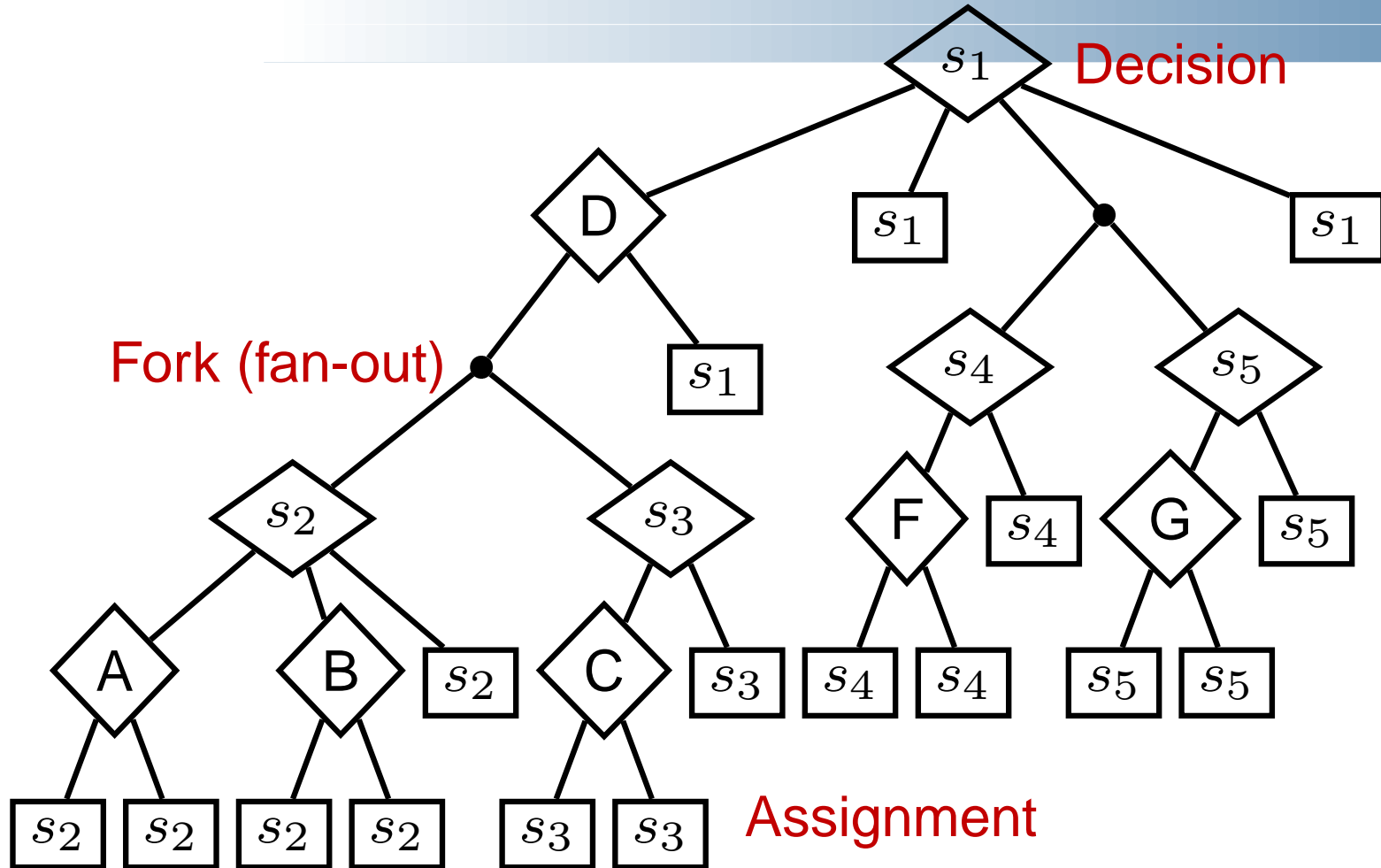
- How should each state machine be encoded?
- Should state be shared between the AB/F and C/G machines?

General Problem Statement

States in an Esterel program are an arbitrary tree of sequential and parallel state machines.



Choosing an Encoding



- How should s_1, \dots, s_4 be encoded?
- Should s_2 or s_3 be shared with s_4 or s_5 ?

Choosing a Good Encoding

Goal: The smallest circuit that meets a timing constraint

1. Start with largest, fastest circuit (one-hot, no sharing)
2. Estimate the slack at each state decision point by estimating how much the delay could be increased at that point while still meeting the timing requirement
3. Attempt to share states at the lowest decision point with the largest slack or reencode the widest-fanout decision point with sufficient slack.
4. Repeat steps 2–3 until no further gain possible



Simulating Esterel

Automata Compilers

Esterel is a finite-state language, so build an automata:

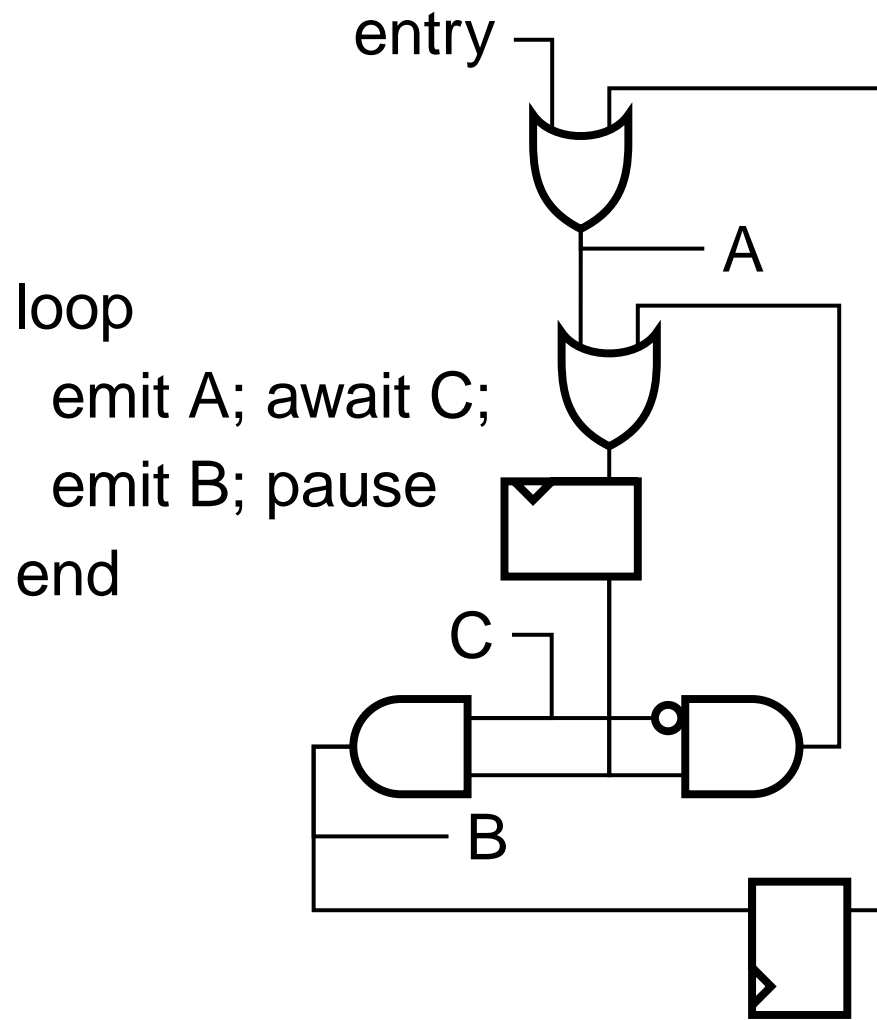
```
loop                               switch (s) {
  emit A; await C;                 case 0: A = 1; s = 1; break;
  emit B; pause                    case 1: if (C) { B = 1; s = 0; } break;
end                                 }
```

V1, V2, V3 (INRIA/CMA) [Berry, Gonthier 1992]

Fastest known code; great for programs with few states.

Does not scale; concurrency causes state explosion.

Netlist-based Compilers



```
A = entry || s2q;  
cf = !C && s1q;  
s1d = cf || A;  
B = s2d = C && s1q;
```

Clean semantics,
scales well, but
inefficient.

Can be 100 times
slower than automata
code.

Discrete-Event Based Compilers

SAXO-RT [Weil et al. 2000] Divides Esterel program into event functions dispatched by a fixed scheduler.

```
loop
  emit A; await C;
  emit B; pause
end

unsigned curr = 0x1;
unsigned next = 0;

static void f1() {
  A = 1;
  curr &= ~0x1; next |= 0x2;
}

static void f2() {
  if (!C) return;
  B = 1;
  curr &= ~0x2; next |= 0x1;
}

void tick() {
  if (curr & 0x1) f1();
  if (curr & 0x2) f2();
  curr |= next;
  next = 0;
}
```



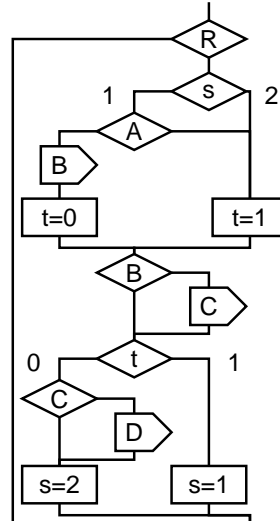
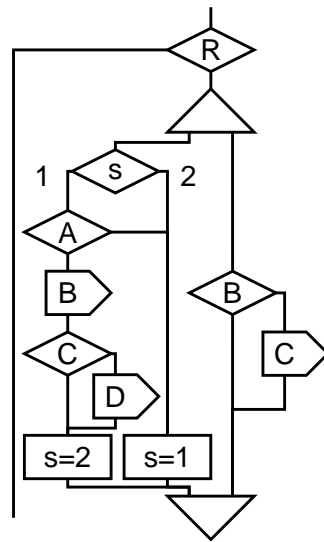
Generating Fast Simulations

My Previous Technique

```

every R do
  loop
    await A;
    emit B;
    present C then
      emit D end;
    pause
  end
||
loop
  present B then
    emit C end;
  pause
end
end

```



```

if ((s0 & 3) == 1) {
  if (s) {
    s3 = 1; s2 = 1; s1 = 1;
  } else
  if (s1 >> 1)
    s1 = 3;
  else {
    if ((s3 & 3) == 1) {
      s3 = 2; t3 = L1;
    } else {
      t3 = L2;
    }
  }
}

```

Esterel

Concurrent Sequential

C code

Source

CFG

CFG

My Previous Technique

1. Translate Esterel into a concurrent control-flow graph
2. Analyze static data dependencies
3. Schedule
4. Generate sequential control-flow graph by inserting context-switching code
5. Translate to C

Comments on Previous Technique

Much more efficient (can be $100\times$) than netlist simulation.

Currently used within Synopsys' CoCentric System Studio for control-code generation.

Context-switching idea powerful, but does not have much insight into program behavior.

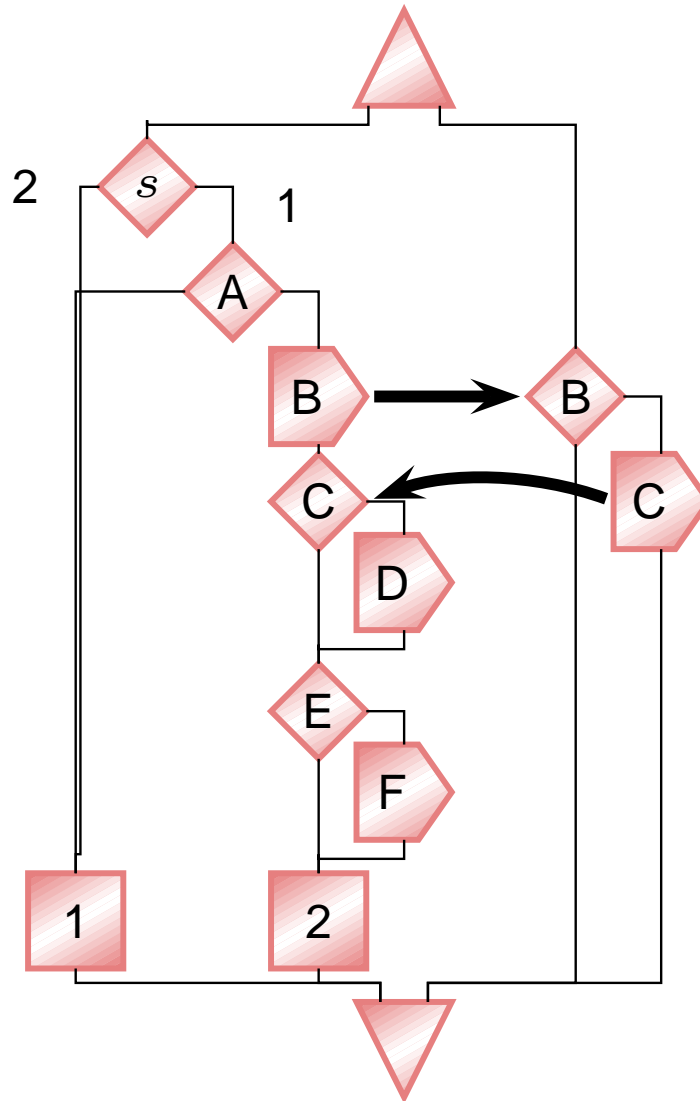
Adheres too closely to control dependencies; many more opportunities to reorder code and simplify scheduling.

New Technique

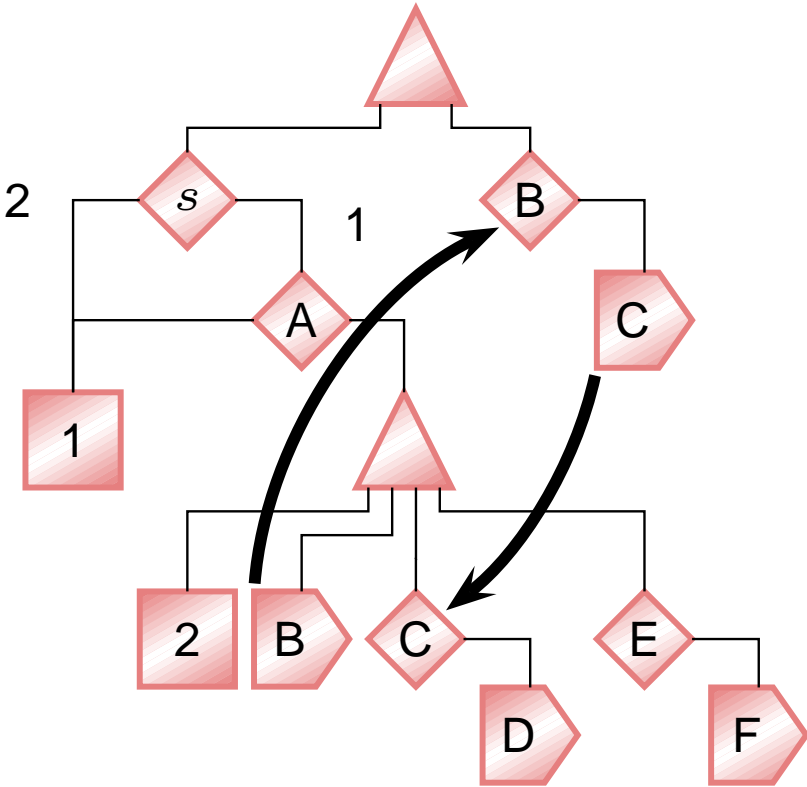
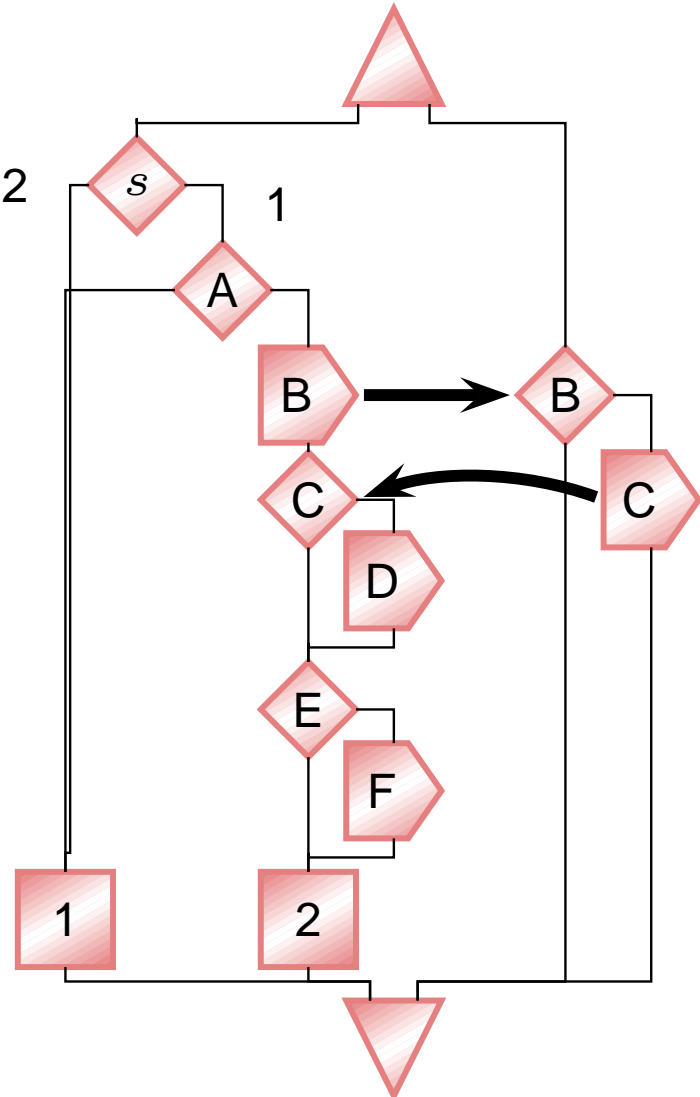
1. Translate Esterel into a concurrent control-flow graph
2. Transform into Program Dependence Graph
3. Analyze static data dependencies
4. Insert control predicates to enable scheduling
5. Schedule
6. Generate sequential control-flow graph
7. Translate to C

A Code-Generation Example

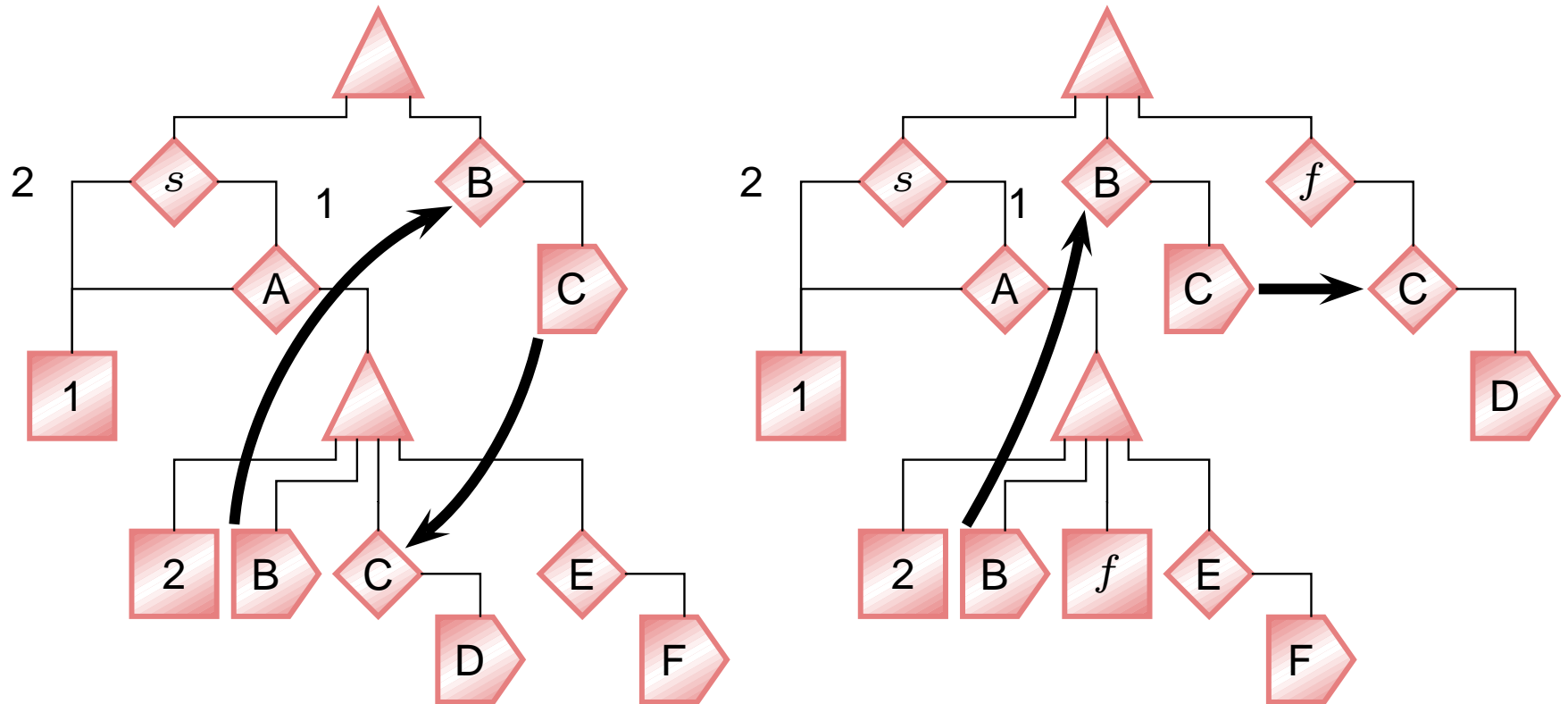
```
loop
  await A;
  emit B;
  present C then
    emit D end;
  present E then
    emit F end;
  pause
end
||
loop
  present B then
    emit C end;
  pause
end
```



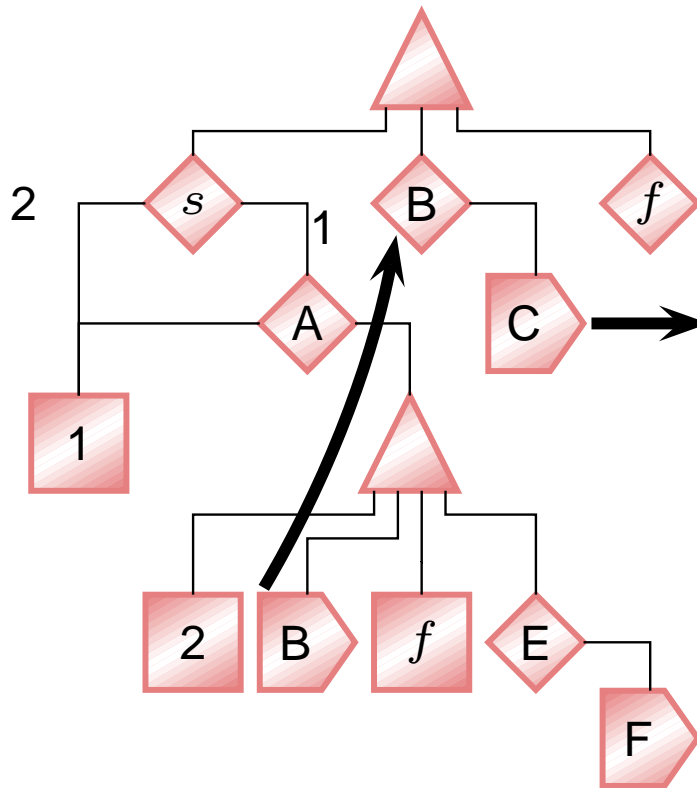
Concurrent Control-Flow and the PDG



Splitting the PDG for Scheduling



Generating Code



```
f = 0;  
if (s1 == 1 && A) {  
    s1 = 2;  
    B = 1;  
    f = 1;  
    if (E) F = 1;  
} else {  
    s1 = 1;  
}  
  
if (B) C = 1;  
if (f && C) D = 1;
```