

# High-level Modeling and Validation Methodologies for Embedded Systems: Bridging the Productivity Gap

## Part 1: Languages and Models of Computation

**Stephen A. Edwards**

Department of Computer Science,  
Columbia University

[www.cs.columbia.edu/~sedwards](http://www.cs.columbia.edu/~sedwards)

[sedwards@cs.columbia.edu](mailto:sedwards@cs.columbia.edu)

## Domain-specific languages

Language embodies methodology

Verilog: Model system and testbench

Multi-rate signal processing languages: Blocks with fixed I/O rates

Java's concurrency: Threads plus per-object locks to ensure atomic access

## Hardware Languages

Verilog

- Structural and procedural modeling

- Four-valued vectors

- Gate and transistor primitives

- Less flexible

- Succinct

VHDL

- Structural and procedural modeling

- Few built-in types; powerful type system

- Fewer built-in features for hardware modeling

- More flexible

- Verbose

## Premise

Shrinking hardware costs, higher levels of integration allow more complex designs

Designers' coding rate staying constant

Higher-level languages the solution

Succinctly express complex systems

## Types of Languages

Hardware

- Structural and procedural styles

- Unbuffered "wire" communication

- Discrete-event semantics

Software

- Procedural

- Some concurrency

- Memory

Dataflow

- Practical for signal processing

- Concurrency + buffered communication

Hybrid

- Mixture of other ideas

## Hardware methodology

Partition system into functional blocks

FSMs, datapath, combinational logic

Develop, test, and assemble

Simulate to verify correctness

Synthesize to generate netlist

## Diversity

Why not just one "perfect" high-level language?

Flexibility trades off analyzability

General-purpose languages (e.g., assembly) difficult to check or synthesize efficiently.

Solution: Domain-specific languages

## Hardware Languages

Goal: specify connected gates concisely

Originally targeted at simulation

Discrete event semantics skip idle portions

Mixture of structural and procedural modeling

## Verilog

Started in 1984 as input to event-driven simulator designed to beat gate-level simulators

Netlist-like hierarchical structure

Communicating concurrent processes

Wires for structural communication,

Regs for procedural communication

## Verilog: Hardware communication

Four-valued scalar or vector "wires"

```
wire alu_carry_out;
wire [31:0] alu_operand;
```

X: unknown or conflict

Z: undriven

Multiple drivers and receivers

Driven by primitive or continuous assignment

```
nand nand1(y2, a, b);
assign y1 = a & b;
```

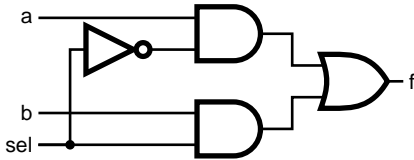
## Mux with User-Defined Primitive

```
primitive mux(f, a, b, sel);
output f;
input a, b, sel;
```

```
table
1?0 : 1;
0?0 : 0;
?11 : 1;
?01 : 0;
11? : 1;
00? : 0;
endtable
endprimitive
```

Behavior defined using a truth table that includes "don't cares"

This is a less pessimistic than others: when a & b match, sel is ignored; others produce X



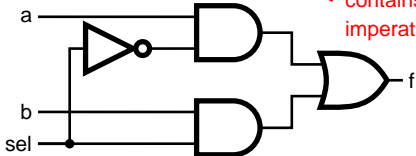
## Multiplexer Built with Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;
```

```
always @(a or b or sel)
if (sel) f = a;
else f = b;
```

A reg behaves like memory: holds its value until imperatively assigned otherwise

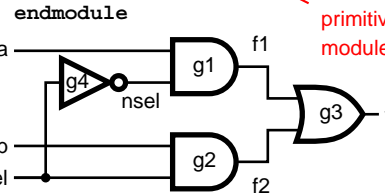
Body of an always block contains traditional imperative code



## Multiplexer Built From Primitives

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
```

```
and g1(f1, a, nsel),
g2(f2, b, sel);
or g3(f, f1, f2);
not g4(nsel, sel);
```



Verilog programs built from modules

Each module has an interface

Module may contain structure: instances of primitives and other modules

## Verilog: Software Communication

Four-valued scalar or vector "register"

```
reg alu_carry_out;
reg [31:0] alu_operand;
```

Does not always correspond to a latch

Actually shared memory

Semantics are convenient for simulation

Value set by procedural assignment:

```
always @(posedge clk)
count = count + 1;
```

## Initial and Always

Run until they encounter a delay

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

or a wait for an event

```
always @(posedge clk) q = d;
```

```
always begin
wait(i);
a = 0;
wait(i);
a = 1;
end
```

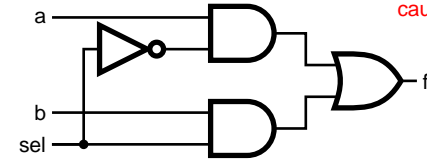
## Mux with Continuous Assignment

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
```

```
assign f = sel ? a : b;
endmodule
```

LHS is always set to the value on the RHS

Any change on the right causes reevaluation



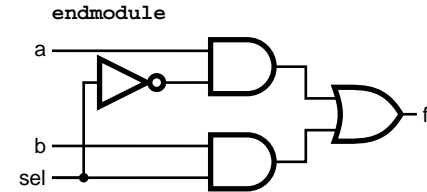
## Multiplexer Built with Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;
```

Modules may contain one or more always blocks

```
always @(a or b or sel)
if (sel) f = a;
else f = b;
```

Sensitivity list contains signals whose change makes the block execute



## Blocking vs. Nonblocking

Verilog has two types of procedural assignment

Fundamental problem:

- In a synchronous system, all flip-flops sample simultaneously
- In Verilog, `always @(posedge clk)` blocks run in some undefined sequence

## A Flawed Shift Register

This does not work as you would expect:

```
reg d1, d2, d3, d4;
```

```
always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

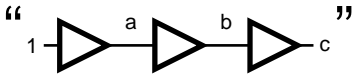
These run in some order, but you don't know which

## Nonblocking Looks Like Latches

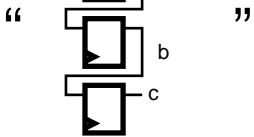
RHS of nonblocking taken from latches

RHS of blocking taken from wires

```
a = 1;
b = a;
c = b;
```



```
a <= 1;
b <= a;
c <= b;
```



## VHDL: Architecture contents

Structural, dataflow, and procedural styles:

```
architecture ex of foo is
begin
  I1: Inverter port map(a, y);

  foo <= bar + baz;

  process begin
    count := count + 1;
    wait for 10ns;
  end
```

## Non-blocking Assignments

This version does work:

```
reg d1, d2, d3, d4;
```

```
always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Nonblocking rule:  
RHS evaluated  
when assignment  
runs

LHS updated only  
after all events for  
the current instant  
have run

## VHDL

Designed for everything from switch to board-level modeling and simulation

Also has event-driven semantics

Fewer digital-logic-specific constructs than Verilog

More flexible language

Powerful type system

More access to event-driven machinery

## VHDL: Communication

Processes communicate through resolved signals:

```
architecture Structure of mux2 is
  signal i1, i2 : Bit;
```

Processes may also use local variables:

```
process
  variable count := Bit_Vector (3 downto 0);
begin
  count := count + 1;
end
```

## Nonblocking Can Behave Oddly

A sequence of nonblocking assignments don't communicate

```
a = 1;
b = a;
c = b;
```

```
a <= 1;
b <= a;
c <= b;
```

Blocking assignment:

```
a = b = c = 1
```

Nonblocking assignment:

```
a = 1
```

```
b = old value of a
```

```
c = old value of b
```

## VHDL: Entities and Architectures

Entity: interface of an object

```
entity mux2 is
  port(a,b,c: in Bit; d: out Bit);
end;
```

Architecture: implementation of an object

```
architecture DF of mux2 is
begin
  d <= c ? a : b;
end DF;
```

## VHDL: The wait statement

Wait for a change

```
wait on A, B;
```

Wait for a condition

```
wait on Clk until Clk = '1';
```

Wait with timeout

```
wait for 10ns;
```

```
wait on Clk until Clk = '1' for 10ns;
```

## VHDL and Verilog Compared

	Verilog	VHDL
Structure	●	●
Hierarchy	●	●
Concurrency	●	●
Switch-level modeling	●	○
Gate-level modeling	●	○
Dataflow modeling	●	●
Procedural modeling	●	●
Type system		●
Event access		●
Interface/implementation		●
Local Variables		●
Shared memory	●	●
Wires	●	●
Resolution functions		●

● Full support    ○ Partial support

## Software methodology

- C
  - Divide into recursive functions
- C++
  - Divide into objects (data and methods)
- Java
  - Divide into objects, threads
- Real-Time Operating Systems
  - Divide into processes, assign priorities

## C Storage Classes

```

/* fi xed address: visible to other fi les */
int global_static;           Three regions:

/* fi xed address: only visible within fi le */
static int file_static;     Static Memory

/* parameters always stacked */
int foo(int auto_param)     The Stack
{
    /* fi xed address: only visible to function */
    static int func_static;  The Heap

    /* stacked: only visible to function */
    int auto_i, auto_a[10];

    /* array explicitly allocated on heap (pointer stacked) */
    double *auto_d =
        malloc(sizeof(double)*5);

    /* return value passed in register or stack */
    return auto_i;
}
    
```

## Software Languages

- Goal: specify machine code concisely
- Sequential semantics: Perform this operation, Change system state
- Raising abstraction: symbols, expressions, control-flow, functions, objects, templates, garbage collection

## The C Language

“Structured Assembly Language”

Expressions with named variables, arrays

```
a = b + c[10];
```

Control-flow (conditionals, loops)

```
for (i=0; i<10; i++) { /* ... */ }
```

Recursive Functions

```
int fib(int x) {
    return x = 0 ? 1 : fib(x-1) + fib(x-2);
}
```

## C++: Classes

C with added structuring features

Classes: Binding functions to data types

```
class Shape {
    int x,y;
    void move(dx, dy) { x += dx; y += dy; }
};

Shape b;
b.move(10,20);
```

## Software Languages

- C
  - Adds types, expressions, control, functions
- C++
  - Adds classes, inheritance, namespaces, templates, exceptions
- Java
  - Adds automatic garbage collection, threads
  - Removes bare pointers, multiple inheritance
- Real-Time Operating Systems
  - Add concurrency, timing control

## Declarators

Declaration: string of specifiers followed by a declarator

```

          basic type
static unsigned int (*f[10])(int, char* [10]);
          specifiers                                declarator
    
```

Base types match the processor's natural ones.

Declarator's notation matches that of an expression: use it to return the basic type.

Largely regarded as the worst syntactic aspect of C: both pre- (pointers) and post-fix operators (arrays, functions).

## C++: Inheritance

Inheritance: New types from existing ones

```
class Rectangle : public Shape {
    int h, w;
    void resize(hh, ww) { h = hh; w = ww; }
};

Rectangle c;
c.resize(5,20);
c.move(10,20);
```

## C++: Namespaces

Grouping names to avoid collisions

```
namespace Shape {
    class Rectangle { /* ... */ };
    class Circle { /* ... */ };

    int draw(Shape* s);
    void print(Shape* s);
}

Shape::Rectangle r;
```

## C++: Operator Overloading

Use expression-like syntax on new types

```
class Complex /* ... */ ;
Complex operator + (Complex &a, int b)
{
    // ...
}

Complex x, y;

x = y + 5;    // uses operator +
```

## Java Threads

Threads have direct language support

`Object::wait()` causes a thread to suspend itself and add itself to the object's wait set

`sleep()` suspends a thread for a specified time period

`Object::notify()`, `notifyAll()` awakens one or all threads waiting on the object

`yield()` forces a context switch

## C++: Templates

Macros parameterized by types

```
template <class T> void sort(T* ar)
{
    // ...
    T tmp;
    tmp = ar[i];
    // ...
}

int a[10];
sort(a);    // Creates sort<int>
```

## C++: Standard Template Library

Library of polymorphic data types with iterators, simple searching algorithms

vector: Variable-sized array

list: Linked list

map: Associative array

queue: Variable-sized queue

string: Variable-sized character strings with memory management

## Java Locks/Semaphores

Every object has a lock; at most one thread can acquire it

Synchronized statements or methods wait to acquire the lock before running

Only locks out other synchronized code: programmer responsible for ensuring safety

```
public static void abs(int[] vals) {
    synchronized (vals) {
        for (int i = 0; i < vals.length; i++)
            if (vals[i] < 0)
                vals[i] = -vals[i];
    }
}
```

## C++: Exceptions

Handle deeply-nested error conditions:

```
class MyException {}; // Define exception

void bar()
{
    throw MyException; // Throw exception
}

void foo() {
    try {
        bar();
    } catch (MyException e) {
        /* ... */ // Handle the exception
    }
}
```

## Java: Simplified C++

Simpler, higher-level C++-like language

Standard type sizes fixed (e.g., int is 32 bits)

No pointers: Object references only

Automatic garbage collection

No multiple inheritance except for interfaces: method declarations without definitions

## Java Thread Example

```
Class OnePlace {
    Element value;

    public synchronized void write(Element e) {
        while (value != null) wait(); // wait
        value = e; // suspends the thread
        notifyAll();
    }

    public synchronized Element read() {
        while (value == null) wait();
        Element e = value; value = null;
        notifyAll(); // notifyAll awakens all waiting threads
        return e;
    }
}
```

## Java: Thread Scheduling

Scheduling algorithm vaguely defined: Made implementers' lives easier, programmers' lives harder

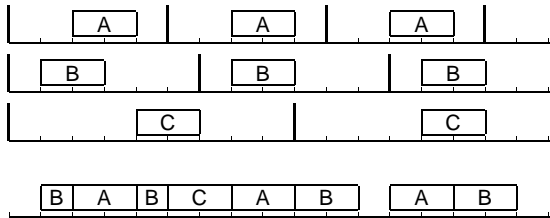
Threads have priorities

Lower-priority threads guaranteed to run when higher-priority threads are blocked

No guarantee of fairness among equal-priority threads

## Priority-based Preemptive Scheduling

Always run the highest-priority runnable process



## Software languages compared

	C	C++	Java	RTOS
Expressions	●	●	●	
Control-flow	●	●	●	
Recursive functions	●	●	●	
Exceptions	○	●	●	
Classes & Inheritance		●	●	
Templates		●		
Namespaces		●	●	
Multiple inheritance		●	○	
Threads & Locks		●	●	●
Garbage collection		○	●	

● Full support    ○ Partial support

## Real-Time Operating Systems

Provides concurrency to sequential languages

Idea: processes handle function, operating system handles timing

Predictability, responsiveness main criteria

## Rate Monotonic Analysis

Common priority assignment scheme

System model:

Tasks invoked periodically

Each runs for some fraction of their period

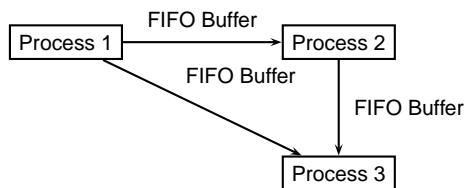
Asynchronous: unrelated periods, phases

Rate Monotonic Analysis assigns highest priorities to tasks with smallest periods

## Dataflow Languages

Best for signal processing

Concurrently-running processes communicating through FIFO buffers



## RTOS scheduling

Fixed-priority preemptive

Sacrifices fairness to reduce context-switching overhead

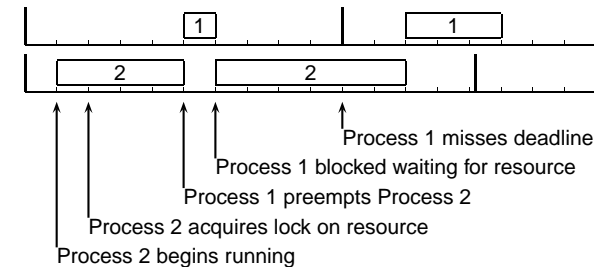
Meeting deadlines more important

Process preempted when higher-priority process is activated

Process otherwise runs until it suspends

## Priority Inversion

Shared resources can enable a lower-priority process to block a higher-priority one.



## Dataflow Languages

Kahn Process Networks

Concurrently-running sequential processes

Blocking read, non-blocking write

Very flexible, hard to schedule

Synchronous Dataflow

Restriction of Kahn Networks

Fixed communication

Easy to schedule

## Dataflow methodology

Kahn:

Write code for each process

Test by running

SDF:

Assemble primitives: adders, downsamplers

Schedule

Generate code

Simulate

## A Process from Kahn's 1974 paper

```

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

Interface includes FIFOs

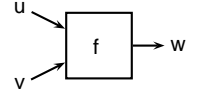
wait() returns the next token in the FIFO, blocking if empty

send() writes a token into a FIFO without blocking

## A Process from Kahn's 1974 paper

```

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



Process alternately reads from u and v, prints the data value, and writes it to w

## Kahn Networks: Determinacy

Sequences of communicated data does not depend on relative process execution speeds

A process cannot check whether data is available before attempting a read

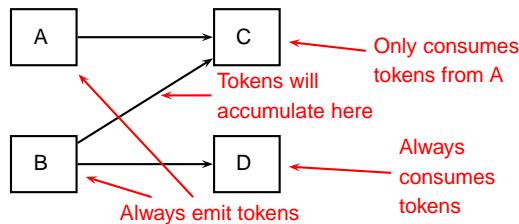
A process cannot wait for data on more than one port at a time

Therefore, order of reads, writes depend only on data, not its arrival time

Single process reads or writes each channel

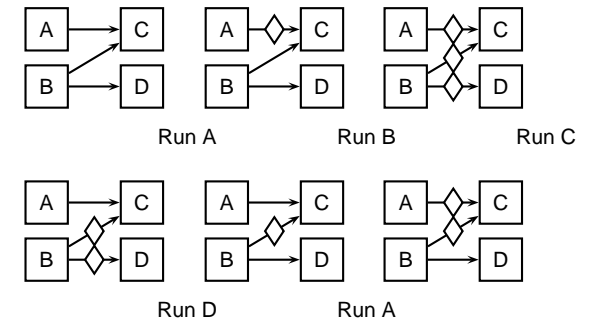
## Scheduling Kahn Networks

Challenge is running without accumulating tokens

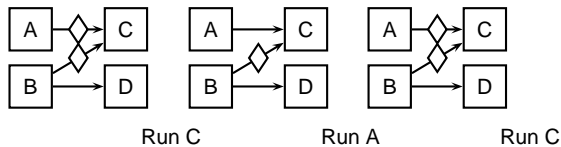


One solution, due to Tom Parks: Start with bounded buffers and increase the size of the smallest buffer when buffer-full deadlock occurs.

## Parks' Algorithm in Action



## Parks' Algorithm in Action



B blocked waiting for space in B→C buffer

Run A, then C, then A, then C, ...

System will run indefinitely

## Synchronous Dataflow

Each process has a firing rule: Consumes and produces a fixed number of tokens every time

Predictable communication: easy scheduling

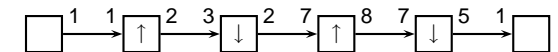
Well-suited for multi-rate signal processing

A subset of Kahn Networks: deterministic

## Multi-rate SDF System

DAT-to-CD rate converter

Converts a 44.1 kHz sampling rate to 48 kHz



↑ Upsampler

↓ Downsampler

## Delays

Kahn processes often have an initialization phase

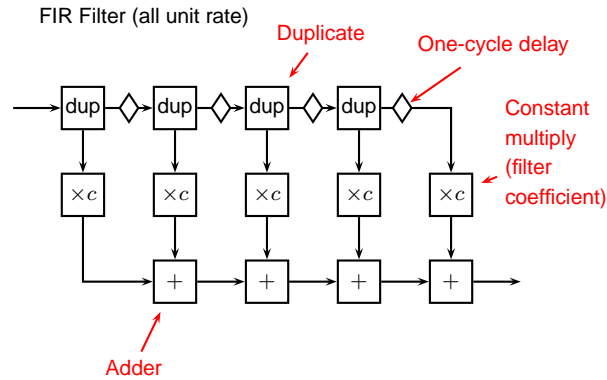
SDF doesn't allow this because rates are not always constant

Alternative: an SDF system may start with tokens in its buffers

These behave like signal-processing-like delays

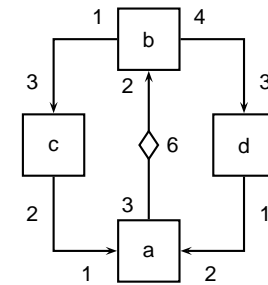
Delays are sometimes necessary to avoid deadlock

## Example SDF System



## SDF Scheduling: Calculating Rates

Each arc imposes a constraint



$$3a - 2b = 0$$

$$4b - 3d = 0$$

$$b - 3c = 0$$

$$2c - a = 0$$

$$d - 2a = 0$$

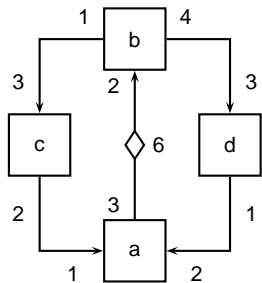
Solution:

$$a = 2c$$

$$b = 3c$$

$$d = 4c$$

## SDF Scheduling: Details



$$a = 2 \quad b = 3$$

$$c = 1 \quad d = 4$$

Possible schedules:

BBBCDDDDAA

BDBBCADDA

BBDDDDCAA

⋮

BC... is not valid

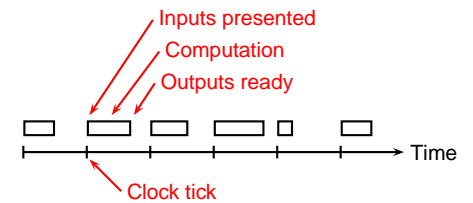
## Kahn and SDF

	Kahn	SDF
Concurrent	●	●
FIFO communication	●	●
Deterministic	●	●
Data-dependent behavior	●	
Fixed rates		●
Statically Schedulable		●

## Esterel's Model of Time

Like synchronous digital logic, it uses a global clock

Provides precise control over which events appear in which clock cycles



## Two Types of Esterel Statements

### Combinational

Execute in one cycle

A bounded number may execute in a single cycle

Examples:

emit

present / if

loop

### Sequential

Take multiple cycles

The only statements that consume any time

Examples:

pause

await

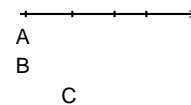
sustain

## Simple Example

```
module Example1:
  output A, B, C;
```

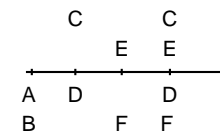
```
  emit A;
  present A then
    emit B
  end;
  pause;
  emit C
```

```
end module
```



## Sequencing and Decisions

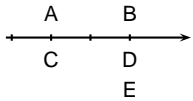
```
emit A;
emit B;
pause;
loop
  present C then emit D end;
  present E then emit F end;
  pause;
end
```



## Concurrency

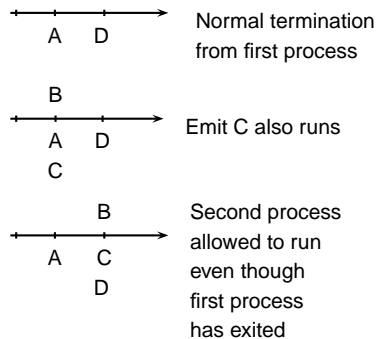
```
[
  await A; emit C
||
  await B; emit D
];
emit E
```

- Parallel statements start in same cycle
- Block terminates once all have terminated

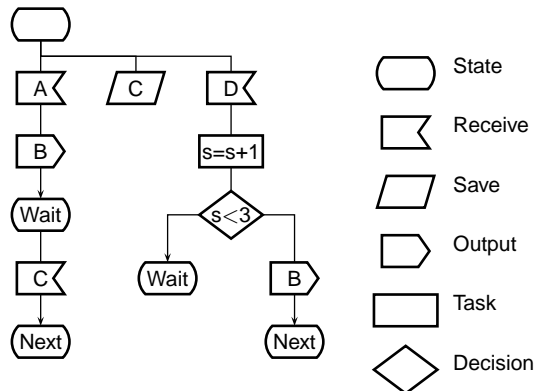


## The Trap Statement

```
trap T in
[
  pause;
  emit A;
  pause;
  exit T
||
  await B;
  emit C
]
end trap;
emit D
```

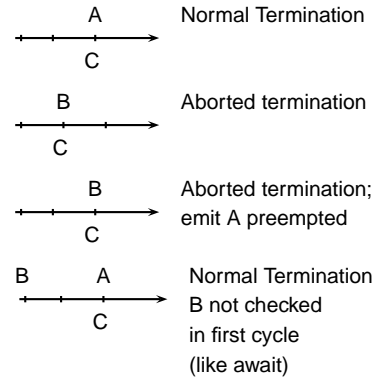


## SDL Symbols



## The Abort Statement

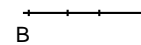
```
abort
  pause;
  pause;
  emit A
when B;
emit C
```



## Nested Traps

```
trap T1 in
  trap T2 in
  [
    exit T1
  ||
    exit T2
  ]
end;
emit A
end;
emit B
```

Outer trap takes precedence; control transferred directly to the outer trap statement.  
**emit A** not allowed to run.

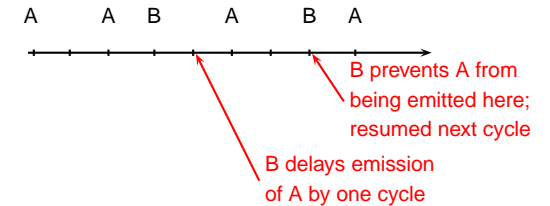


## Conclusions

Many types of languages  
Each with its own strengths and weaknesses  
None clearly "the best"  
Each problem has its own best language  
Hardware languages focus on structure  
Verilog, VHDL  
Software languages focus on sequencing  
Assembly, C, C++, Java, RTOSes  
Dataflow languages focus on moving data  
Kahn, SDF  
Others a mixture  
Esterel, SDL

## The Suspend Statement

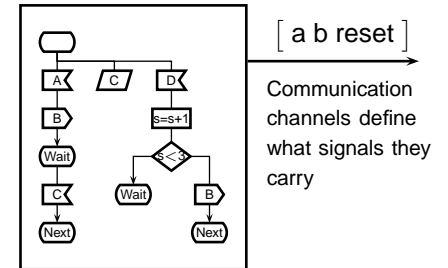
```
suspend
  loop
    emit A; pause; pause
  end
when B
```



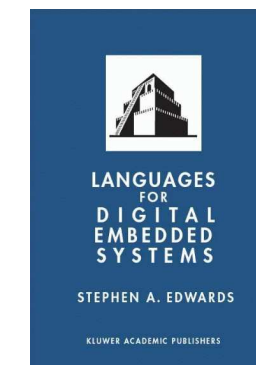
## SDL

Concurrent FSMs, each with a single input buffer

Finite-state machines defined using flowchart notation



## Shameless Plug



All of these languages are discussed in greater detail in

Stephen A. Edwards.  
*Languages for Digital Embedded Systems.*  
Kluwer 2000.