SHIM: A Deterministic Model for Heterogeneous Embedded Systems

Stephen A. Edwards and Olivier Tardieu

Department of Computer Science,
Columbia University

www.cs.columbia.edu/~sedwards

{sedwards,tardieu}@cs.columbia.edu

Definition

shim \’shim\ *n*

1 : a thin often tapered piece of material (as wood, metal, or stone) used to fill in space between things (as for support, leveling, or adjustment of fit).

2 : *Software/Hardware Integration Medium*, a model for describing hardware/software systems

Conclusions

SHIM is an effective model of computation for embedded hardware/software systems

Formal semantics guarantee determinism & boundedness

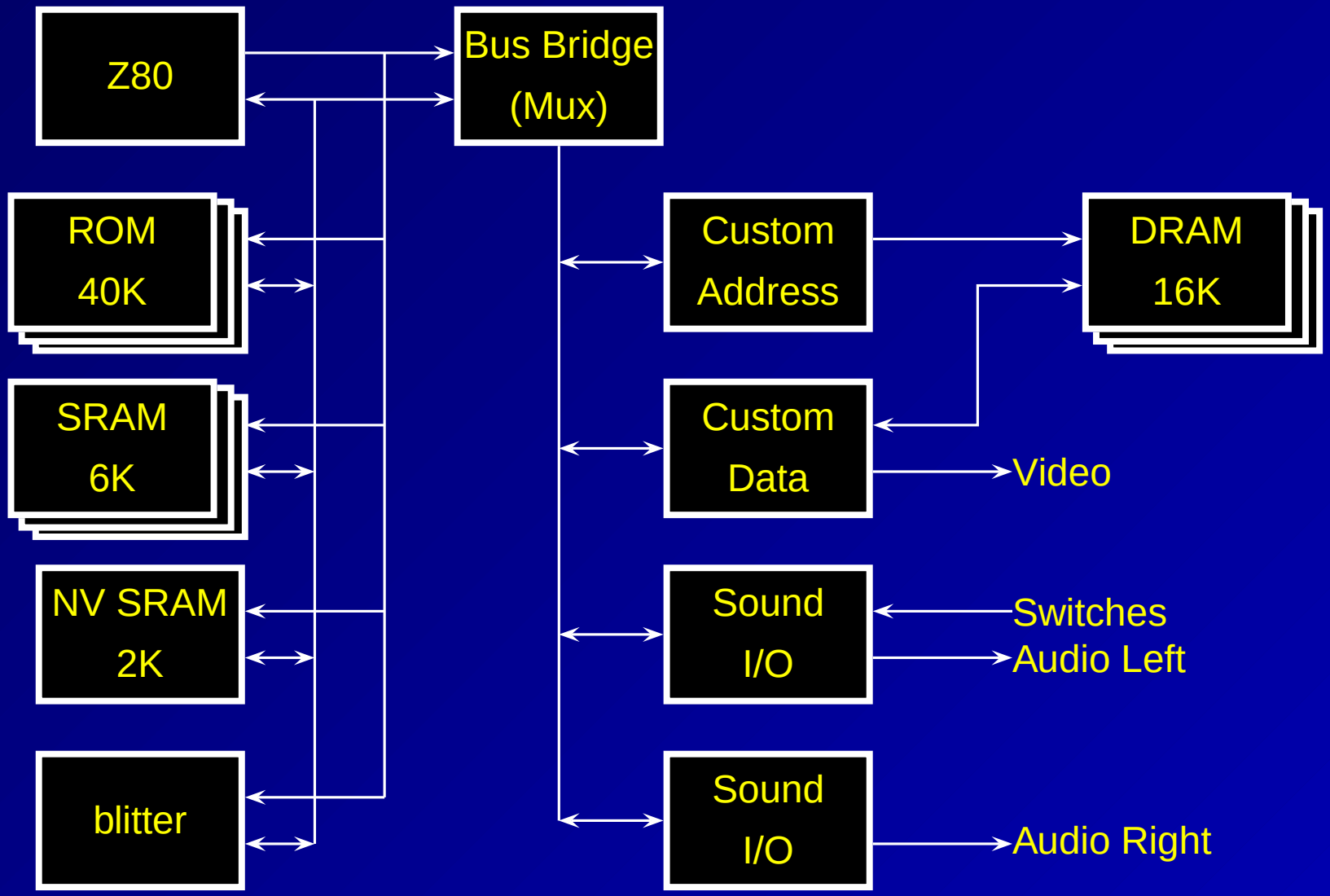
Easy to synthesize into hardware and software

Applicable to large, important class of systems, but not all

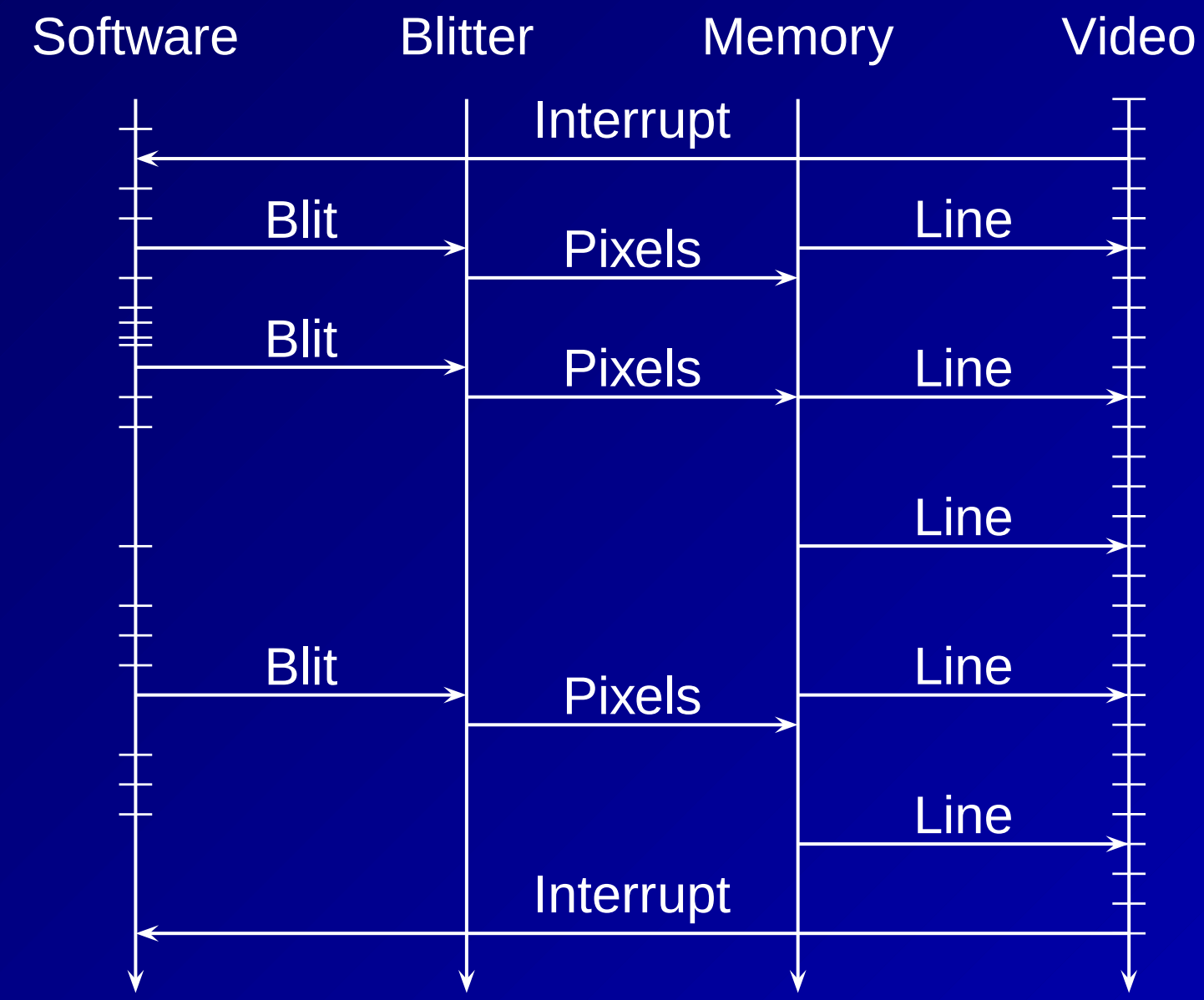
Embedded systems should be designed on the SHIM model of computation

Robby Roto (Bally/Midway, 1981)

Robby Roto Block Diagram



HW/SW Interaction



SHIM Wishlist

- *Mixes synchronous and asynchronous styles*

  Need multi-rate for hardware/software systems

- *Delay-insensitive (Deterministic)*

  Want simulated behavior to reflect reality

  Verify functionality and performance separately

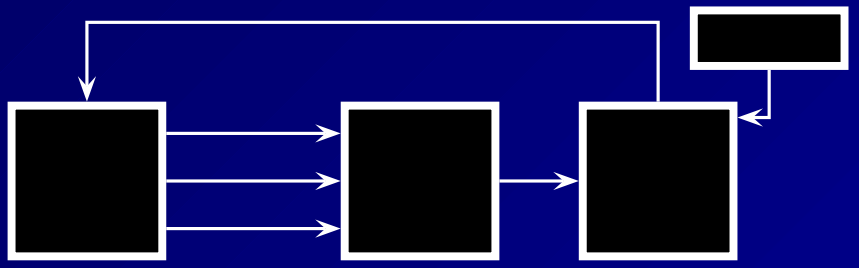
- *Only requires bounded resources*

  Hardware resources fundamentally bounded

- *Formal semantics*

  Do not want arguments about what something means

The SHIM Model



Sequential processes

Unbuffered point-to-point communication channels exchange data tokens

Fixed topology

Asynchronous

Synchronous communication events

Delay-insensitive: sequence of data through any channel independent of scheduling policy (the Kahn principle)

“Kahn networks with rendezvous communication”

SHIM vs. Other Models

	SHIM	CSP	Kahn	SDF	Haste	Sync	Petri
Deterministic	√		√	√		√	
Blocking Communication	√	√			√	√	√
Bounded Buffers	√	√		√	√	√	
Multi-Rate	√	√	√	√	√		√
Data-Dependent Rates	√	√	√		√		√
Easy-To-Schedule	√	√		√	√	√	√
Static Scheduling				√		√	

Modeling in SHIM

To model	introduce
Buffers	Buffer processes
Interrupts	Polling and periodic communication
Synchrony	Clock signals
Synchronous dataflow	Buffers
Sensors	Source processes
Arbiters	A deterministic algorithm

Modeling Time in SHIM

SHIM is timing-independent

Philosophy: separate functional requirements from performance requirements

Like synchronous digital logic: establish correct function independent of timing, then check and correct performance errors

Vision: clock processes impose execution rates, checked through static timing analysis

The Syntax of Tiny-SHIM

$e ::= L$		(literal)
	$\mid V$	(variable)
	$\mid$ *op* e	(unary op)
	$\mid e$ *op* e	(binary op)
	$\mid$ **(** e **)**	(paren)

$s ::= V$ **=** e		(assignment)
	$\mid$ **if (** e **)** s **else** s	(conditional)
	$\mid$ **while (** e **)** s	(loop)
	$\mid s$ **;** s	(sequencing)
	$\mid$ **read(** C, V **)**	(blocking read)
	$\mid$ **write(** C, e **)**	(blocking write)
	$\mid$ { s }	(grouping)

Example Processes

Local variables: d, e

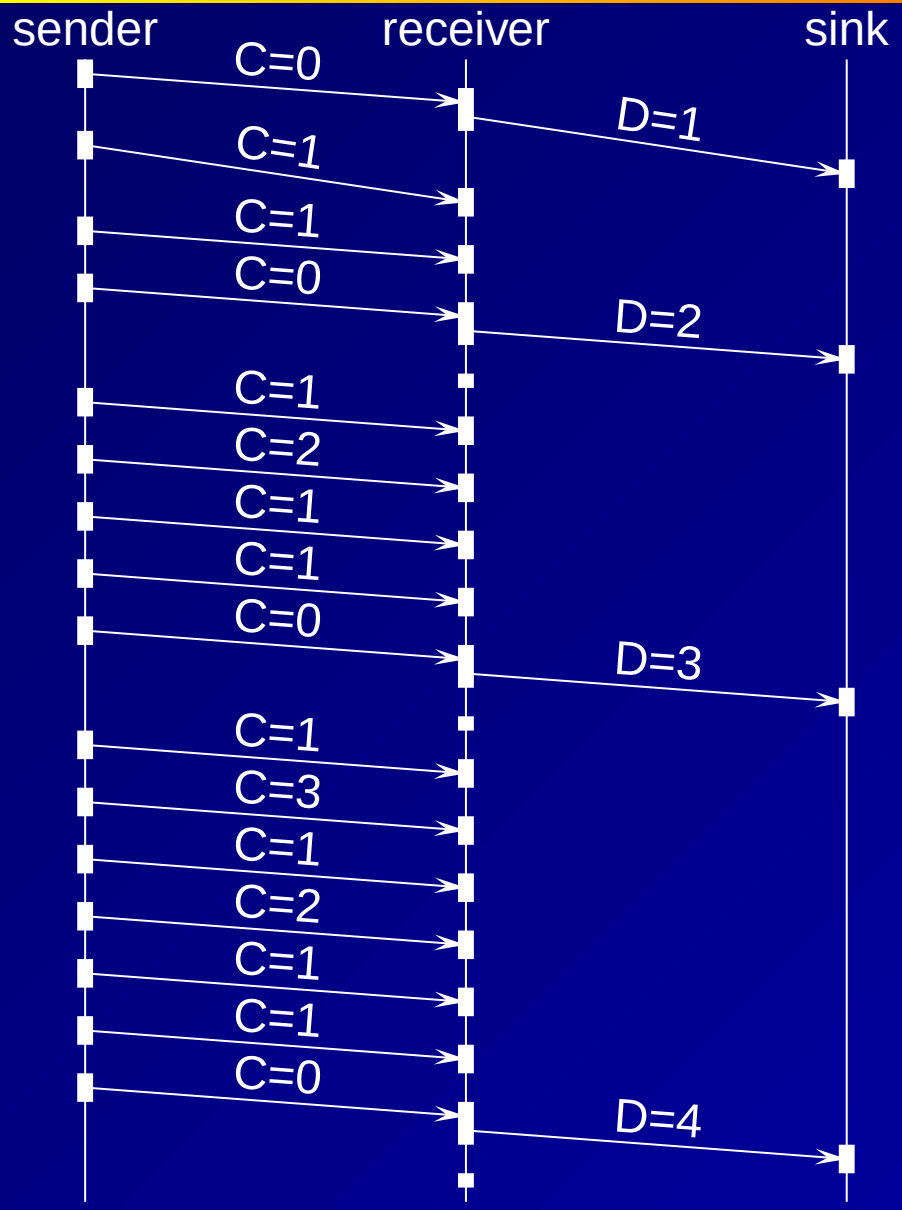
```
d = 0;
while (1) {
  e = d;
  while (e > 0) {
    write(c, 1);
    write(c, e);
    e = e - 1;
  }
  write(c, 0);
  d = d + 1;
}
```

c

Local variables: a, b, r, v

```
a = 0;
b = 0;
while (1) {
  r = 1;
  while (r) {
    read(c, r);
    if (r != 0) {
      read(c, v);
      a = a + v;
    }
  }
  b = b + 1;
}
```

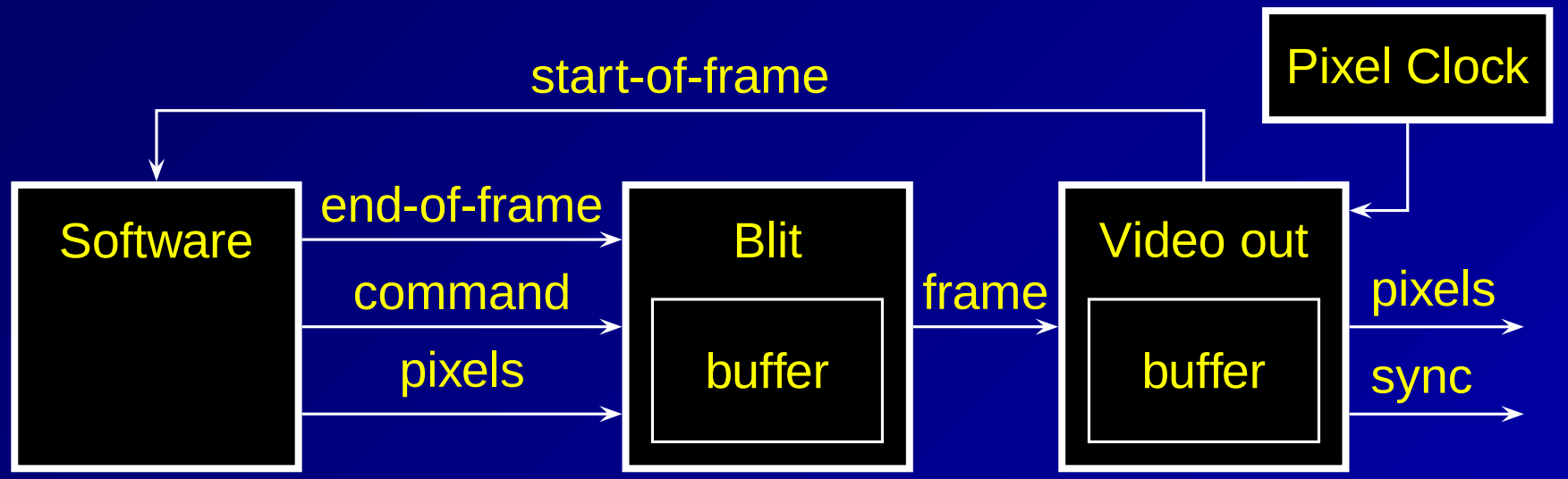
Behavior of the Processes




```
d = 0;
while (1) {
  e = d;
  while (e > 0) {
    write(c, 1);
    write(c, e);
    e = e - 1;
  }
  write(c, 0);
  d = d + 1;
}
```

```
a = 0;
b = 0;
while (1) {
  r = 1;
  while (r) {
    read(c, r);
    if (r != 0) {
      read(c, v);
      a = a + v;
    }
  }
  b = b + 1;
}
```

Robby Roto in SHIM: Block Diagram



while the player is alive **do**
- Wait for start-of-frame
- ...game logic...
- Write “false” to end-of-frame
- Write to the blitter
- ...game logic...
- Write “true” to end-of-frame

while 1 **do**
- **while** not end-of-frame **do**
  - Read blit command
  - Write pixels to memory
- Write frame

while 1 **do**
- Write start-of-frame
- **for each** line **do**
  - Emit line timing signals
  - **for each** pixel **do**
    - Wait for pixel clock
    - Read pixel from memo
    - Send pixel to display
- Read next frame

Translating Tiny-SHIM to Software

Each process becomes a C function with static variable that can resume itself

Main scheduler takes runnable process from head of list and calls its function.

Processes mark themselves blocked, can scheduler their communication partner.

C Translation Example

```
process p1 {
  output C;
  write(C, 42);
}
```

```
process p2 {
  input C;
  int v;
  read(C, v);
}
```

C Declarations

```
typedef struct process_struct {
  void (*process)(void);
  struct process_struct *next;
} process_t;
```

Linked list of runnable processes
Function of process
Next runnable process

```
typedef struct {
  int value;
  process_t *waiting;
} channel_t;
```

Channel datatype
Value being transferred
Blocked process, if any

```
channel_t C = { 0, 0 };
```

Defi nition of channel C

```
int p1_state = 0;
int p2_state = 0;
```

State of each process

```
void p1_function(void);
void p2_function(void);
```

Process functions
(forward declarations)

```
process_t p1 = { p1_function, 0 };
process_t p2 = { p2_function, &p1 };
process_t *head_process = &p2;
```

Linked List
of runnable
processes

The (Trivial) Scheduler

```
int main()                                          The scheduler
{
  process_t *running_process;
  while (head_process) {
    running_process = head_process;                 Remove head
    head_process = running_process->next;
    (*(running_process->process))();                Run it
  }
  return 0;                          Everything terminated or deadlocked
}
```

The Writing Process

```
void p1_function() {
  switch (p1_state) {
  case 1: goto L1;
  case 0: goto L0;
  }

L0:
  C.value = 42;
  if (C.waiting) {
    (C.waiting)->next = head_process;
    head_process = C.waiting;
  }
  C.waiting = &p1;
  p1_state = 1;
  return;
L1:
  ;
}
```

Resume at current state

write(C, 42)

Schedule reading process

Suspend

```
process p1 {
  output C;
  write(C, 42);
}
```

The Reading Process

```
void p2_function() {
  static int v;
  switch (p2_state) {
  case 0: goto L0;
  case 1: goto L1;
  }

 L0:
  if (!C.waiting) {
    C.waiting = &p2;
    p2_state = 1;
    return;
  }
 L1:
  v = C.value;
  (C.waiting)->next = head_process;
  head_process = C.waiting;
  C.waiting = 0;
}
```

resume at current state

```
process p2 {
  input C;
  int v;
  read(C, v);
}
```

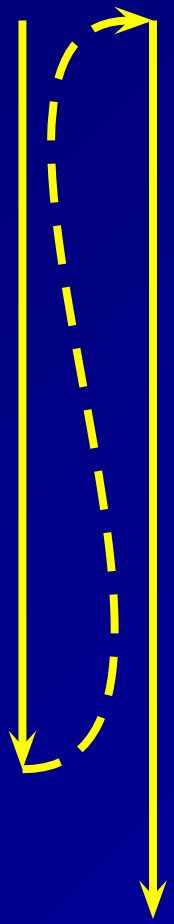
read(C, v)

Suspend

Schedule writing process

Write before Read

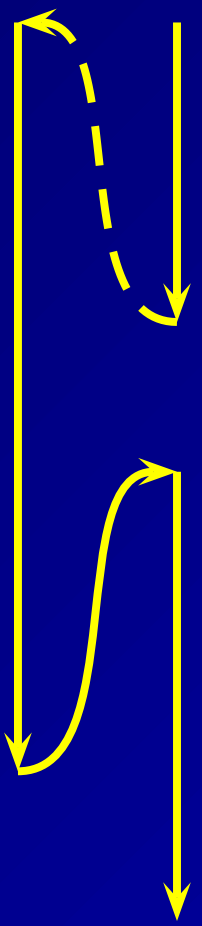
```
/* write(C, 42) */
C.value = 42;
if (C.waiting) {
   (C.waiting)->next =
     head_process;
   head_process =
     C.waiting;
}
C.waiting = &p1;
p1_state = 1;
return;
L1:
```



```
/* read(C, v) */
if (!C.waiting) {
   C.waiting = &p2;
   p2_state = 1;
   return;
}
L1:
v = C.value;
(C.waiting)->next =
   head_process;
head_process =
   C.waiting;
C.waiting = 0;
```

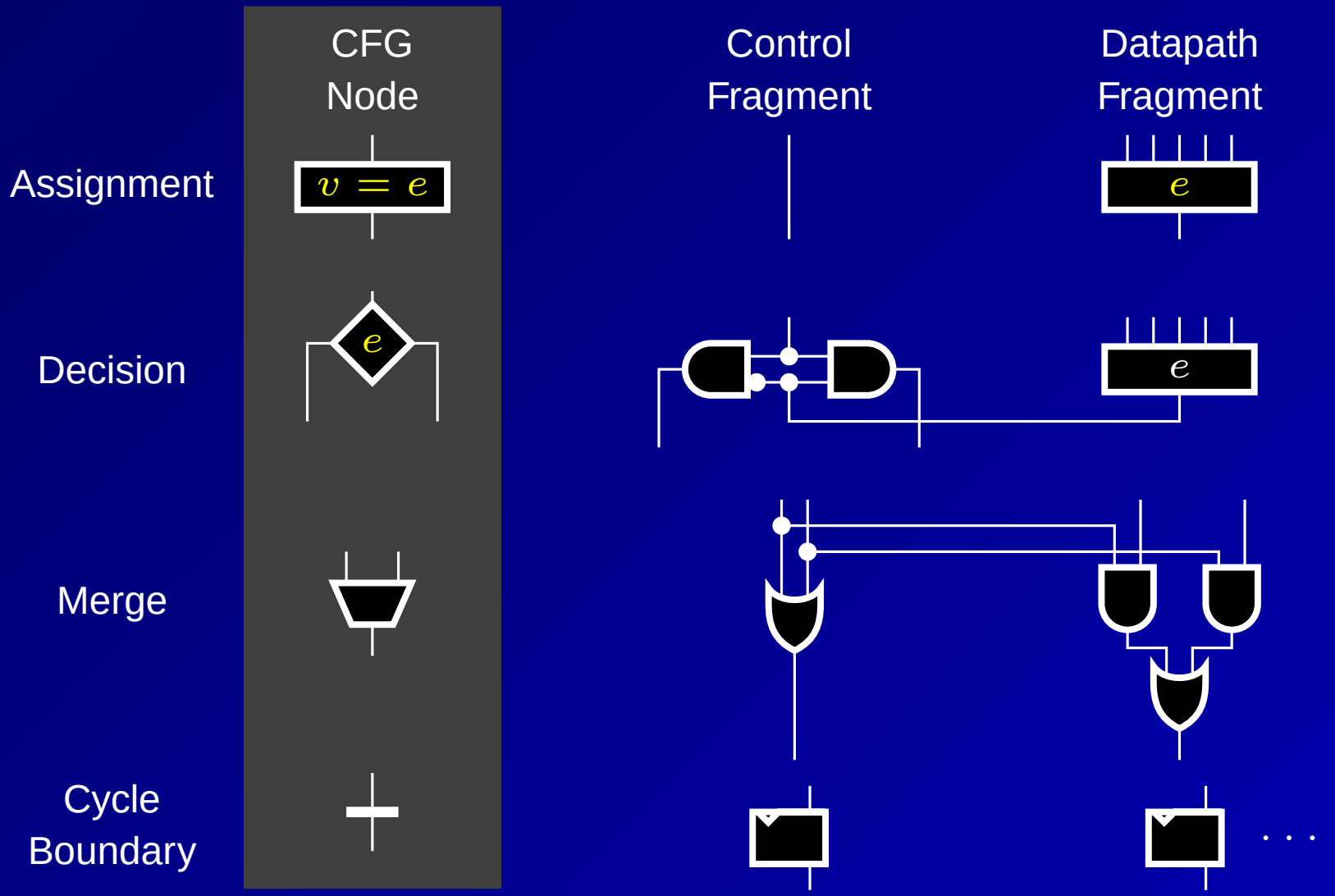
Read before Write

```
/* write(C, 42) */
C.value = 42;
if (C.waiting) {
  (C.waiting)->next =
    head_process;
  head_process =
    C.waiting;
}
C.waiting = &p1;
p1_state = 1;
return;
L1:
```

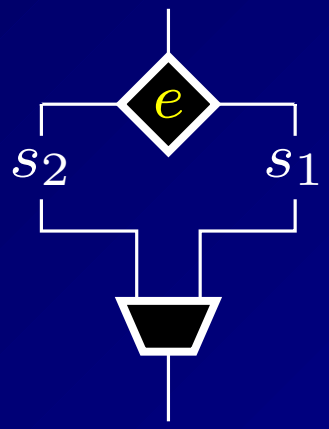


```
/* read(C, v) */
if (!C.waiting) {
  C.waiting = &p2;
  p2_state = 1;
  return;
}
L1:
v = C.value;
(C.waiting)->next =
  head_process;
head_process =
  C.waiting;
C.waiting = 0;
```

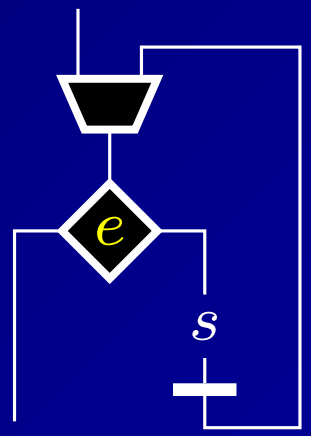
Translating Tiny-SHIM to Hardware



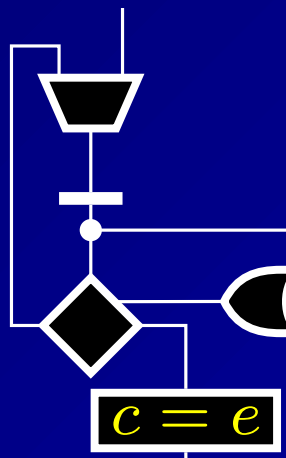
Translation Patterns



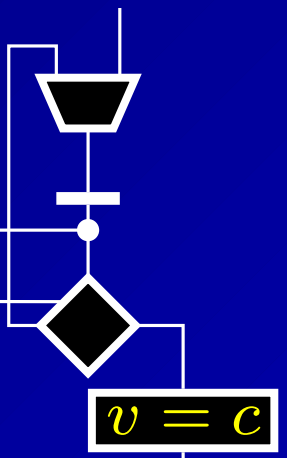
if (e **)** s_1 **else** s_2



while (e **)** s

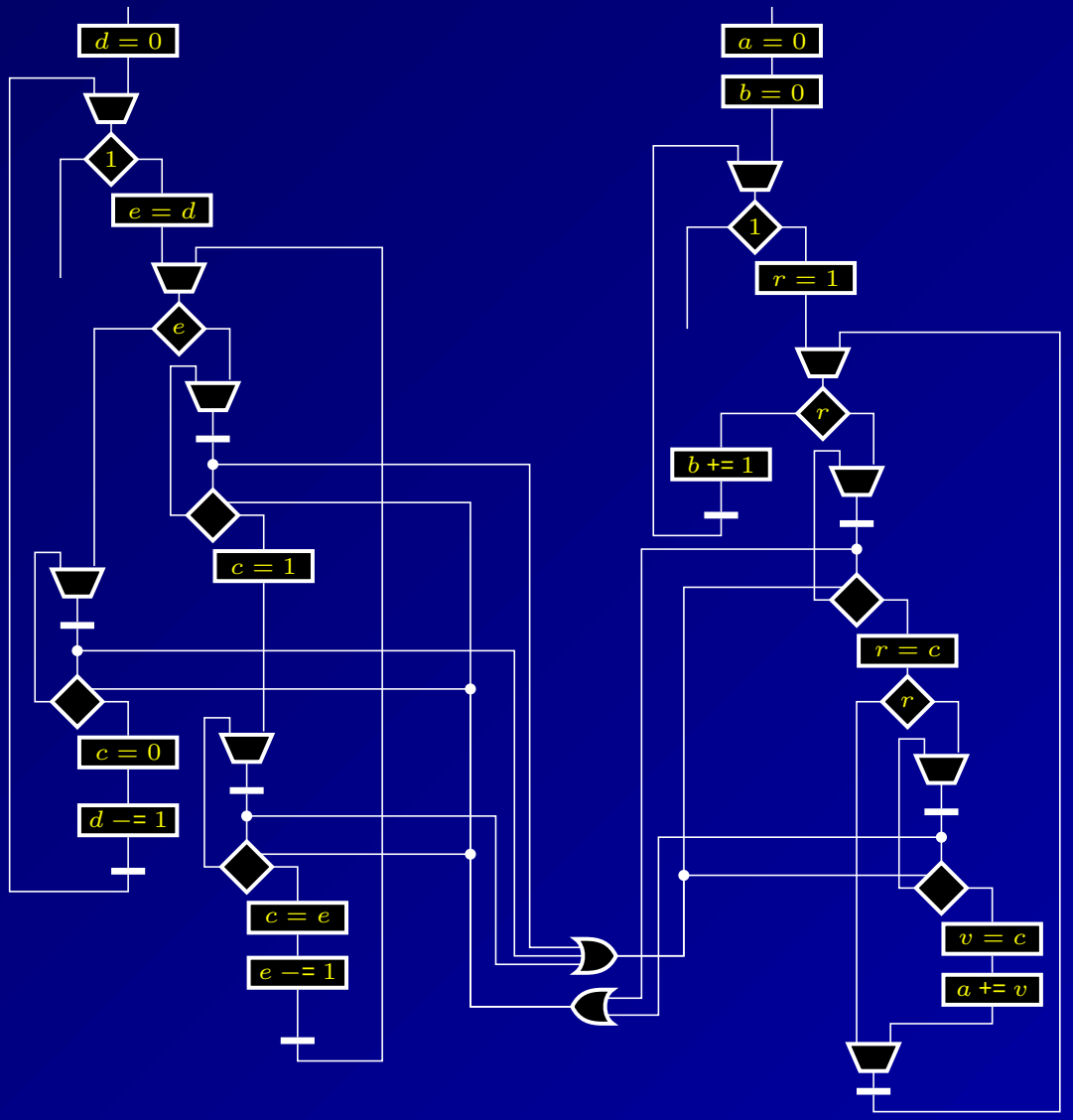


write(c, e **)**



read(c, v **)**

Hardware Translation Example




```
d = 0;
while (1) {
  e = d;
  while (e > 0) {
    write(c, 1);
    write(c, e);
    e = e - 1;
  }
  write(c, 0);
  d = d + 1;
}
```

```
a = 0;
b = 0;
while (1) {
  r = 1;
  while (r) {
    read(c, r);
    if (r != 0) {
      read(c, v);
      a = a + v;
    }
  }
  b = b + 1;
}
```

The SOS Semantics of Tiny-SHIM

σ	Process memory state	p	Process code
$\langle \sigma, p \rangle$	Process p in state σ	$\langle \sigma \rangle$	Terminated in state σ
$\xrightarrow{a}$	Single-process rule	$\Rightarrow$	System rule
$\mathcal{E}(\sigma, e)$	Value of e in σ		

$$\frac{\mathcal{E}(\sigma, e) = n}{\langle \sigma, v \texttt{ = } e \rangle \rightarrow \langle \sigma[v \leftarrow n] \rangle} \quad \text{(assign)}$$

$$\frac{\mathcal{E}(\sigma, e) \neq 0}{\langle \sigma, \texttt{if (}e\texttt{)}\ p\ \texttt{else}\ q \rangle \rightarrow \langle \sigma, p \rangle} \quad \text{(if-true)}$$

$$\frac{\mathcal{E}(\sigma, e) = 0}{\langle \sigma, \texttt{if (}e\texttt{)}\ p\ \texttt{else}\ q \rangle \rightarrow \langle \sigma, q \rangle} \quad \text{(if-false)}$$

Semantics of Looping & Sequencing

$$\frac{\mathcal{E}(\sigma, e) \neq 0}{\langle \sigma, \texttt{while (}e\texttt{)}\ p \rangle \rightarrow \langle \sigma, p\ \texttt{;}\ \texttt{while (}e\texttt{)}\ p \rangle} \quad \text{(while-true)}$$

$$\frac{\mathcal{E}(\sigma, e) = 0}{\langle \sigma, \texttt{while (}e\texttt{)}\ p \rangle \rightarrow \langle \sigma \rangle} \quad \text{(while-false)}$$

$$\frac{\langle \sigma, p \rangle \xrightarrow{a} \langle \sigma', p' \rangle}{\langle \sigma, p\ \texttt{;}\ q \rangle \xrightarrow{a} \langle \sigma', p'\ \texttt{;}\ q \rangle} \quad \text{(seq)}$$

$$\frac{\langle \sigma, p \rangle \xrightarrow{a} \langle \sigma' \rangle}{\langle \sigma, p\ \texttt{;}\ q \rangle \xrightarrow{a} \langle \sigma', q \rangle} \quad \text{(seq-term)}$$

Communication and Concurrency

$$\langle \sigma, \texttt{read(}c, v\texttt{)} \rangle \xrightarrow{c \text{ get } n} \langle \sigma[v \leftarrow n] \rangle \qquad \text{(read)}$$

$$\frac{\mathcal{E}(\sigma, e) = n}{\langle \sigma, \texttt{write(}c, e\texttt{)} \rangle \xrightarrow{c \text{ put } n} \langle \sigma \rangle} \qquad \text{(write)}$$

$$\frac{\langle \sigma, p \rangle \rightarrow s}{\{\langle \sigma, p \rangle\} \uplus S \Rightarrow \{s\} \uplus S} \qquad \text{(step)}$$

$$\frac{\langle \sigma, p \rangle \xrightarrow{c \text{ put } n} s \qquad \langle \sigma', p' \rangle \xrightarrow{c \text{ get } n} s'}{\{\langle \sigma, p \rangle, \langle \sigma', p' \rangle\} \uplus S \Rightarrow \{s, s'\} \uplus S} \qquad \text{(sync)}$$

Summary

- SHIM: A delay-insensitive (deterministic) model of computation that supports synchrony and asynchrony
- Tiny-SHIM: A little language that embodies the model
- A procedure for translating Tiny-SHIM into software
- A procedure for translating Tiny-SHIM into hardware
- Formal operational semantics of Tiny-SHIM

Ongoing Work

- Relaxation of block-on-single-channel rule
- Complete hardware/software design language
- Static analysis of deadlock
- Translation optimization for hardware and software