Bluespec-2: Two-level Compilation and Scheduling of Operations

Arvind
Laboratory for Computer Science
M.I.T.

January 13, 2003

http://www.csg.lcs.mit.edu/IAPBlue

Outline

• From Bluespec to TRS to FSM ⇐
• TRS execution semantics
• Compiling a single rule
• Scheduling of multiple rules
  – conflict-free analysis
  – mutual exclusion analysis
  – sequential compositionality
• Putting it all together
Bluespec: A two-level language

Source code

Level 1 compilation

Intermediate form: Rules and Actions (Term Rewriting System)

Level 2 compilation

Object code (Verilog/C)

- Type checking
- Massive partial evaluation
- Rule conflict analysis
- Rule scheduling

From TRS to Synchronous CFSM

Transition Logic

Collection of State Elements

S

I

S_{\text{Next}}

O
Hardware Elements

- **Boolean gates**
  - AND, OR, NOT, ...

- **Simple combinational circuits**
  - mux
  - add, subtract, increment, equal, greater than, ...

- **Synchronous state elements**
  - flipflop with enable
  - register
  - multiported register file, array, ...
  - FIFO

![Diagram of flipflop and register]

FIFO

```plaintext
interface FIFO a =
  enq :: a -> Action -- enqueue an item
  deq :: Action -- remove the oldest entry
  first :: a -- inspect the oldest item
  clear :: Action -- make the FIFO empty

- FIFO can be implemented directly in Verilog or in Bluespec using registers

![Diagram of FIFO module]
```

$n = \# \text{ of bits needed to represent the values of type \text{"a"}}$
TRS Execution Semantics

Given a set of rules and an initial term $s$

While (some rules are applicable to $s$)
- choose an applicable rule (non-deterministic)
- apply the rule atomically to $s$

The trick to generating good hardware is to schedule as many rules in parallel as possible without violating the sequential semantics given above.

Rule: As a State Transformer

- A rule may be decomposed into two parts $\pi(s)$ and $\delta(s)$ such that

$$s_{next} = \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s$$

$\delta(s)$ is expressed as (atomic) actions on the state elements. These actions can be enabled only if $\pi(s)$ is true.
Compiling a Rule

"Bz Taken":
when (Bz rc ra) <- bu.first, rf.sub rc == 0
implies action pc := rf.sub ra
bu.clear

π = enabling condition
δ = action signals & values

π’s from the rules that update R

δ’s from the rules that update R

What if more than one rule is enabled?
Combining State Updates

Scheduler ensures that at most one $\phi_i$ is true

Single-rewrite-per-cycle Scheduler

1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$

3. One rewrite at a time
   i.e. at most one $\phi_i$ is true
Executing Multiple Rules Per Cycle

“Fetch”:
when True
  ==> action pc := pc+1
      bu.enq (imem.get pc)

“Add”:
when (Add rd rs rt) <= bu.first
  ==> action rf.upd rd (rf.sub rs + rf.sub rt)
      bu.deq

Can these rules be executed simultaneously?

These rules are “conflict free” because they manipulate different parts of the state (i.e., pc and rf), and enq and deq on a FIFO can be done simultaneously.

Multiple Rewrites Per Cycle

“Fetch”:
when True
  ==> action pc := pc+1
      bu.enq (imem.get pc)

“Bz Taken”:
when (Bz rc ra) <= bu.first, rf.sub rc == 0
  ==> action pc := rf.sub ra
      bu.clear

Can these rules be executed simultaneously?

Yes, as long as the action of Bz Taken rule appears to take effect after the Fetch rule!

Fetch and Bz taken rules are “sequentially composable”
Conflict-Free Rules

- Rule\textsubscript{a} and Rule\textsubscript{b} are conflict-free if

\[ \forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \]
1. \[ \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \]
2. \[ \delta_a(\delta_b(s)) =\delta_b(\delta_a(s)) \]

**Theorem:** Conflict-free rules can be executed concurrently without violating TRS’s sequential semantics*

*From a practical point of view it does not always make sense to compute \( \delta_b(\delta_a(s)) \) in one cycle

---

Mutually Exclusive Rules

- Rule\textsubscript{a} and Rule\textsubscript{b} are mutually exclusive if they can never be enabled simultaneously

\[ \forall s . \pi_a(s) \Rightarrow \sim \pi_b(s) \]

**Theorem:** Mutually-exclusive rules are Conflict-free

Mutual-exclusive analysis brings down the cost of conflict-free analysis
Sequentially Composable Rules

- Rule\textsubscript{a} and Rule\textsubscript{b} are *sequentially composable* if firing of Rule\textsubscript{a} does not disable Rule\textsubscript{b}

\[ \forall s . \pi_a(s) \land \pi_b(s) \Rightarrow \pi_b(\delta_a(s)) \]

**Theorem**: Sequentially composable rules can be executed concurrently without violating TRS’s sequential semantics provided the state is updated according to $\delta_b(\delta_a(s))$.*

*From a practical point of view it does not always make sense to compute $\delta_b(\delta_a(s))$ in one cycle

Conflict-Free Scheduler

- Partition rules into maximum number of disjoint sets such that
  - a rule in one set may conflict with one or more rules in the same set
  - a rule in one set is conflict free with respect to all the rules in all other sets
    (Best case: All sets are of size 1!!)

- Schedule each set independently
  - Priority Encoder, Round-Robin Priority Encoder
  - Enumerated Encoder

*The state update logic depends upon whether the scheduler chooses “sequential composition” or not*
Multiple-Op-per-Cycle Scheduler

1. $\phi_i \Rightarrow \pi_i$

2. $\pi_1 \lor \pi_2 \lor \ldots \lor \pi_n \Rightarrow \phi_1 \lor \phi_2 \lor \ldots \lor \phi_n$

3. Multiple operations such that $\phi_i \land \phi_j \Rightarrow R_i$ and $R_i$ are conflict-free or sequentially composable

Conflict Analysis

- Register
  - read2 - write2
  - read1 - CF - SC
  - write1 - C - SC

- FIFO
  - enq1 - C - CF - CF - SC
  - first1 - CF - CF - SC - SC
  - deq1 - CF - C - C - SC
  - clear1 - SC - C - C - CF

for SC assume $op_1 < op_2$
**Conflict Analysis**

"1. Fetch":
   when True
   ```
   ==> action
   pc := pc + 1
   bu.enq (imem.get pc)
   ```

"2. Add":
   when (Add rd rs rt) <= bu.first
   ```
   ==> action
   rf.upd rd (rf.sub rs + rf.sub rt)
   bu.deq
   ```

"3. Bz Not Taken":
   when (Bz rc ra) <= bu.first, (rf.sub rc) /= 0
   ```
   ==> action
   bu.deq
   ```

"4. Bz Taken":
   when (Bz rc ra) <= bu.first, rf.sub rc == 0
   ```
   ==> action
   pc := rf.sub ra
   bu.clear
   ```

What are the restrictions on scheduling these rules?

- 2, 3 & 4 are ME
- 1 is CF wrt 2 & 3
- 1 is SC wrt 4

**Generate Enable Signals**

\[\begin{align*}
\pi_1 \cdot \pi_4 & \rightarrow \text{bu.enq} \\
\delta_{1,\text{bu.enq}} & \rightarrow \text{bu enq value} \\
\pi_2 \rightarrow \text{OR} & \rightarrow \text{bu.deq} \\
\pi_3 & \rightarrow \text{bu.clear} \\
\pi_4 & \rightarrow \text{bu.clear} \\
\pi_1 \cdot \pi_4 & \rightarrow \text{OR} & \rightarrow \text{pc._write} \\
\delta_{1,\text{PC}} & \rightarrow \text{AND} & \rightarrow \text{pc value}
\end{align*}\]