BeiHang Short Course, Part 2: Operation-Centric Hardware Description and Synthesis

James C. Hoe
Department of ECE
Carnegie Mellon University
Collaborator: Arvind (MIT)

Rules

- \( \text{Gcd}(a, b) \) if \( a > b, b! = 0 \) \( \Rightarrow \) \( \text{Gcd}(a\%b, b) \) (mod)
- \( \text{Gcd}(a, b) \) if \( a > b, b!=0 \) \( \Rightarrow \) \( \text{Gcd}(a-b, b) \) (mod-iter)
- \( \text{Gcd}(a, b) \) if \( a<b \) \( \Rightarrow \) \( \text{Gcd}(b, a) \) (flip)
- \( \text{Gcd}(a, 0) \Rightarrow a \) (done)

Execution:

\[
\begin{align*}
\text{Gcd}(2,4) & \xrightarrow{\text{mod-iter}} \text{Gcd}(2,2) \xrightarrow{\text{mod-iter}} \text{Gcd}(0,2) \\
\text{Gcd}(2,0) & \xrightarrow{\text{flip}} \text{Gcd}(2,0)
\end{align*}
\]
FSM#1: what is NS for “b”

\[ b_{\text{next}} = (a < b) \land a : b \]

FSM#2: what is NS for “a”

\[ a_{\text{next}} = (\text{flip or mod}) \land (\text{flip} \land b : a - b) : a \]
Is it clear that the two FSMs together implements GCD?

Cooperating FSM is “State-centric”
Operation-Centric Decomposition

\[
\begin{align*}
&\text{when } a < b \\
&\quad \begin{cases} 
\ a' = b \\
\ b' = a
\end{cases}
\\
&\text{when } a \geq b \land b \neq 0 \\
&\quad \begin{cases} 
\ a' = a - b \\
\ b' = b
\end{cases}
\\
&\text{Otherwise do nothing}
\end{align*}
\]

A Very Complicated Real-Life Example: Out-of-Order Speculative Processor
A Trivial Made-Up Example: Decoupled Fetch/Execute

Fetch

Execute

Just Two Instructions

- Program-visible state
  - program counter: PC
  - register file: RF[

- Add rd, r1, r2
  - RF[rd] ← RF[r1] + RF[r2]
  - PC ← PC + 1

- Bz ra, rc
  - if RF[rc]==0 then PC ← RF[ra]
  - else PC ← PC + 1
Interactions between Fetch and Execute

- Motivations
- Operation-centric hardware abstraction
- Synthesis of an operation-centric description
- Wrap-Up
Operation-Centric Abstraction

\[ \text{STATE} = \text{Proc}(\text{pc}, \text{imem}, \text{bf}, \text{rf}) \]

Processor Model: Fetch Rule

Fetch Rule

\[
\text{Proc}(\text{pc}, \text{imem}, \text{bf}, \text{rf}) \\
\Rightarrow \text{Proc}(\text{pc}+1, \text{imem}, \text{bf}.\text{enq}(\text{inst}), \text{rf}) \\
\text{let inst=imem[pc]}
\]
TRSpec Rewrite Rules

• Takes notation from Term Rewriting Systems (TRS)

<left-hand-side pattern>  
    when <predicate expression>  

==> <right-hand-side rewrite expression>  
    let  
    <variables bindings>

Atomic 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 $s$  
}  

Note: after a rule fires, applicability of rules is re-evaluated from scratch on the new state
**Processor Model: Execute Rules**

Add Rule

Proc( pc, imem, bf, rf )

when bf.first( )=Add(rd, r1, r2)

⇒ Proc( pc, imem, bf.deq( ), rf[ rd:=(rf[r1]+rf[r2]) ] )

**Processor Model: branch-if-zero**

Bz Not-Taken

Proc( pc, imem, bf, rf ) if rf[rc]≠0

when bf.first( )=Bz(ra, rc)

⇒ Proc( pc, imem, bf.deq( ), rf )

Bz Taken

Proc( pc, imem, bf, rf ) if rf[rc]==0

when bf.first( )=Bz(ra, rc)

⇒ Proc( rf[ra], imem, bf.clear( ), rf )

Is this (good) hardware description?
Operation-Centric Abstraction

- Explicit declaration of storage (same as RTL)
- Describes system behavior as a collection of guarded actions (a.k.a. rules); instead a collection of distributed state-machine NS logic
  - a rule is guarded by a predicate condition; if “condition” true then always correct to apply action
  - rule application is atomic, i.e., if multiple rules enabled, pick only one to proceed
  - an execution corresponds to a sequence of rule applications

Excerpt from Superscalar Model: Dataflow-Order Dispatch

Rule “Dispatch Instruction : Non-Branch”

\[
\text{IntU( Queue(.. \{ entry \}[i] ..), \ldots \ldots )} \\
\text{if ( op is a valid type && ALU is available )} \\
\text{where} \\
\text{RsEntry(Valid, id, op, arg1, arg2) = entry} \\
\text{Arg(Valid, value1, --) = arg1} \\
\text{Arg(Valid, value2, --) = arg2} \\
\Rightarrow \\
\text{IntU( Queue(.. \{ RsEntry(Invalid,--,--,--,--,--) }[i] ..), \ldots} \\
\text{\ldots, Result(Valid, id, val), \ldots \ldots )} \\
\text{where} \\
\text{val=Execute(op, value1, value2)}
\]
Outline

- Motivations
- Operation-centric hardware abstraction
- Synthesis of an operation-centric description
- Wrap-Up

Operations to Synchronous C-FSM

Mapping and Scheduling
Rule: A Functional Interpretation

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

$$\text{rule} = \lambda s. \text{if } \pi(s) \text{ then } \delta(s) \text{ else } s$$
Putting Them All Together

enables from different rules that update PC

next state values from different rules that update PC

J. C. Hoe, CMU/ECE/CALCM, © 2014, BHSC-L2-s25
Putting Them All Together

enables from different rules that update PC

next state values from different rules that update PC

Single-Rule-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 rule at a time
   i.e., at most one \( \phi_i \) is true
Correctness

• Implementation is deterministic but the spec is not
  – implementation’s state transitions must correspond to some legal execution of TRSpec
  – implementation must maintain liveness

• Weak-fairness can be achieved
  – if a transition stays applicable, it will be selected within bounded number of steps

Good HW should fire Fetch and Execute rules together

Fetch Rule  Execute Rules (except Bz Taken)
Executing Rules Concurrently

- Applying Fetch and Add together on the same state when both are enabled
  - does not produce conflicting updates
  - gives the same results as if one after the other
  - in particular, applying doesn’t invalidate the other

- Concurrent Execution
  - statically determine which transitions can be safely executed concurrently (formalizing the above)
  - generate a scheduler and update logic that allows as many concurrent transitions as possible

Conflict-Free Rules

\( R_a \) and \( R_b \) are conflict-free if

\[
\forall s. \pi_a(s) \land \pi_b(s) \Rightarrow \\
\begin{align*}
1. & \ \pi_a(\delta_b(s)) \land \pi_b(\delta_a(s)) \\
2. & \ \delta_a(\delta_b(s)) = \delta_b(\delta_a(s)) \\
3. & \ \delta_a(\delta_b(s)) = \delta_a(s) \oplus \delta_b(s)
\end{align*}
\]

You can fire any number of conflict-free rules in a clock cycle as long as they are all pairwise conflict-free!!
Multiple-Rule-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 rules such that $\phi_i \land \phi_j \Rightarrow R_i$ and $R_j$ are conflict-free

Conflict-Free Scheduler

- Partition rules into maximum number of non-overlapping sets such that rules in different sets are conflict-free
  (Best case: All sets are of size 1!!)

- Schedule each set independently
  - e.g., one-rule per cycle per set

- The state update logic is unchanged
CF Scheduling Example

Conflict Graph

CF Graph
Multiple-Rule-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 rules such that \( \phi_i \land \phi_j \Rightarrow R_i \) and \( R_j \) are conflict-free

Performance Gain

- Multiple rules per cycle
  But is this always optimal?

- CF scheduler does not increase critical path
  - partitioned schedulers are smaller and faster than a single monolithic scheduler
  - distributed scheduler lowers wiring delay for \( \pi's \) and \( \phi's \)
**CF Schedule is too strict**

$R_a$ and $R_b$ are sequentially-composable (SC) if

$$\forall s . \pi_a(s) \land \pi_b(s) \Rightarrow$$

1. $\pi_a(\delta_b(s)) \land \pi_b(\delta_a(s))$
2. $\delta_a(\delta_b(s)) = \delta_b(\delta_a(s))$
3. $\delta_a(\delta_b(s)) = \delta_a(s) \oplus \delta_b(s)$

Applying a pair of SC rules concurrently to the same state produce the same outcome as only one ordering, but that is all that is required.

---

**SC Scheduling**

For each CF scheduling group in a given clock cycle,

$$\phi_a \land \phi_b \land \phi_c ....$$

$$\Rightarrow$$ the transitive closure of $R_a, R_b, R_c ....$ on SC is ordered

For the sake of implementation, we further require the orderings to be consistent in all clock cycles.
TRSpec and TRAC
(aka my PhD Thesis)

TRSpec

Design

TRAC

RTL sim

Synopsys

Target Tech.

Std Cell

Gate Array

FPGA

TRSpec vs. Verilog

- 5-stage pipelined, 32-bit MIPS R2000 Integer Core

<table>
<thead>
<tr>
<th></th>
<th>CBA tc6a Area (cells)</th>
<th>LSI 10K Area (gates)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Clock</td>
<td>Clock</td>
</tr>
<tr>
<td>TRSpec</td>
<td>9059 10.3ns 96.6MHz</td>
<td>34674 23.7ns 41.9MHz</td>
</tr>
<tr>
<td>Hand-coded Verilog RTL</td>
<td>7168 10.4ns 96MHz</td>
<td>26543 23.8ns 42.1MHz</td>
</tr>
</tbody>
</table>
Recap the Last Hour

- Operation-centric design abstracts away synchronous clock as the marker of progress
  - designer thinks in terms of a sequence of atomic updates
  - many correct mapping to synchronous FSM-D
  - let compiler pick a “good” one
- What if precise timing is a part of the design specification?
  - need a way to mix abstractions smoothly

Bluespec: doing it for real

- A real commercial implementation
  - operation-centric → “guarded atomic actions”
  - full high-level language with proper modular design support
  - mix seamlessly with RTL-like timing control when necessary
- Free academic license available
- Visit www.bluespec.com