Architectural Synthesis and Exploration using Term Rewriting Systems

Arvind
James C. Hoe

Laboratory for Computer Science
Massachusetts Institute of Technology
http://www.csg.lcs.mit.edu
Outline

- Introduction
- Term Rewriting Systems (TRS) as a Hardware Description Language
- Hardware Synthesis from Term Rewriting Systems
- Results
Internet/Communication Space

- Rapidly changing functionality and performance requirements necessitate rapid hardware development
  - ATM, frame-relay, Gigabit Ethernet, packet-over-SONET protocols
  - voice-over-IP, video, streaming data,
    *QoS issues dominant*
  - merger of LAN and WAN infrastructures

- Currently addressed by
  - General-purpose or Embedded processors + ASICs
  - Network processors *(emerging)*

*ASIC development time and cost is the limiting factor in product release*
Current ASIC Design Flow

**Manual Steps**
- Verification nightmare
- Labor Intensive
- Time Consuming
- Error Prone

**Informal Architectural Spec**

**High-level C Simulators**

**RTL Implementation**

**Synthesis/Optimization**

**ASICs**

**Fab**

**Time pressure means:**

little architecture exploration & high technology risk
Our New Design Technology

- Reduces time to market
  - Faster design capture
  - Same specification for simulation, verification and synthesis
  - Rapid feedback ⇒ architectural exploration

- Enables rapid development of a large variety of chips with related designs
  ⇒ complex systems-on-a-chip

- Reduces manpower requirement

  Makes designing hardware as commonplace as writing software
State-Centric Descriptions

Schematics

Hardware description languages

```verilog
always @ (posedge Clk) begin
    if (a >= b) begin
        a <= a - b;
        b <= b;
    end else begin
        a <= b;
        b <= a;
    end
end
```

what does it describe?
Operation-Centric Descriptions

Euclid’s Algorithm

\[
\begin{align*}
\text{Gcd}(a, b) \text{ if } b \neq 0 & \Rightarrow \text{Gcd}(b, \text{Rem}(a, b)) \quad (\text{Rule}_1) \\
\text{Gcd}(a, 0) & \Rightarrow a \quad (\text{Rule}_2) \\
\text{Rem}(a, b) \text{ if } a < b & \Rightarrow a \quad (\text{Rule}_3) \\
\text{Rem}(a, b) \text{ if } a \geq b & \Rightarrow \text{Rem}(a-b, b) \quad (\text{Rule}_4)
\end{align*}
\]

Execution:

\[
\begin{align*}
\text{Gc11d}(2,4) & \Rightarrow \text{Gcd}(4,\text{Rem}(2,4)) \\
\Rightarrow & \text{Gcd}(2,\text{Rem}(4,2)) \\
\Rightarrow & \text{Gcd}(2,\text{Rem}(0,2)) \\
\Rightarrow & 2
\end{align*}
\]

Hardware description?
Operation-Centric Description: MIPS

MIPS Microprocessor Manual

ADD rd, rs, rt

GPR[rd] ← GPR[rs] + GPR[rt]

PC ← PC + 4
TRS as a Hardware Description Language
Term Rewriting System

\[ \text{TRS} \equiv \langle A, R \rangle \]

a set of terms  a set of rewriting rules

hierarchically organized state elements

state transitions

System \equiv \text{Structure} + \text{Behavior}

An operation centric view of the world
**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 $s$

}
Architectural Description
**AX Architectural Description**

- **Type SYS** = Sys( PROC, IPORT, OPORT )
- **Type PROC** = Proc( PC, RF, PROG, BF )
- **Type PC** = Bit[16]
- **Type RF** = Array[RNAME] VAL
- **Type RNAME** = Reg0 || Reg1 || Reg2 || . . .
- **Type VAL** = Bit[16]
- **Type PROG** = Array[PC] INST
- **Type BF** = Fifo INST_D
- **Type IPORT** = Iport VAL
- **Type OPORT** = Oport VAL
**AX Instruction Set**

\[
Type \text{ INST} = \begin{cases} 
\text{Loadi (RD, VAL)} \\
\text{Loadpc (RD)} \\
\text{Add (RD, R1, R2)} \\
\text{Sub (RD, R1, R2)} \\
\ldots \\
\text{Bz (RA, RC)} \\
\text{MovToO (R1)} \\
\text{MovFromI (RD)}
\end{cases}
\]

**Decoded instructions**

\[
Type \text{ INST}_D = \begin{cases} 
\text{Add}_d (RD, V1, V2) \\
\ldots
\end{cases}
\]

RD, RA, etc. are RNAME’s. V1, V2, etc. are values
**AX Processor Model: Fetch Rules**

**Fetch Add Rule**

\[
\text{Proc}( \text{pc, rf, prog, bf} ) \\
\text{if } r_1 \not\in \text{target(bf)} \land r_2 \not\in \text{target(bf)} \\
\text{where } \text{Add}(r, r_1, r_2) = \text{prog}[pc] \\
\Rightarrow \text{Proc}( \text{pc+1, rf, prog, enq(bf, Add_d(r, rf[r_1], rf[r_2]))} )
\]
AX Processor Model: Execute Rules

\[
\text{Proc}( pc, rf, prog, bf ) \text{ if } r_1 \notin \text{target}(bf) \land r_2 \notin \text{target}(bf) \\
\quad \text{where } \text{Add}(r, r_1, r_2) = \text{prog}[pc] \\
\Rightarrow \text{Proc}( pc+1, rf, prog, \text{enq}(bf, \text{Add}_d(r, rf[r_1], rf[r_2]))) \\
\]

\[
\text{Proc}( pc, rf, prog, bf ) \\
\quad \text{where } \text{Add}_d(r, v_1, v_2) = \text{first}(bf) \\
\Rightarrow \text{Proc}( pc, rf[r:=v_1+v_2], prog, \text{deq}(bf) ) \\
\]

“Execute Add”
TRS as an HDL

- Clean, expressive, precise and concise
  - speculative & superscalar microarchitectures
    [IEEE Micro, June ’99]
  - memory models & cache coherence protocols
    [ISCA99, ICS99]
- Supports parallel and non-deterministic specifications
- The correctness of a TRS can be verified against a reference TRS specification
- Some pipelining can be done automatically as a source-to-source transformation on TRS’s
- Superscalar versions of TRS’s can be derived mechanically from pipelined TRS’s.
Synthesis from TRS’s
From TRS to Synchronous FSM

- Extract state elements (registers) from the type declaration
- Extract state transition logic from the rules
Rule: As a State Transformer

Proc( pc, rf, prog, bf ) where $Bz_d(v_a, 0) = \text{first}(bf)$

$\Rightarrow$ Proc( $v_a$, rf, prog, clear(bf) )
Reference Implementation

- Synchronous state elements

- Single transition per clock cycle
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-a-time $\Rightarrow$ at most one $\phi_i$ is true
Combining Logic from Multiple Rules

- **latch enables from different rules**
- **next state values from different rules**

![Diagram showing OR gate connecting latch enables and next state values from different rules]

PC' next state value

δ₀,PC
δ₁,PC
δₙ,PC

sel

φ₀
φ₁
φₙ
Performance Considerations

- Concurrent Execution
  - Statically determine which transitions can be safely executed concurrently
  - Generate a scheduler and update logic that allows as many concurrent transitions as possible

Caution: Concurrent firing of two rules can violate one-transition-at-a-time semantics if, for example, firing of one rule disables the other

Conflict-free rules
Quality of Synthesis
TRAC Synthesis Flow

- Design SPEC
- Transform
- Compile
- RTL Sim
- Synopsys
- C Sim
- Std Cell
- Gate Array
- FPGA

Arvind, MIT Lab for Computer Science

NTT, January 12, 2000, Slide 26
## Performance: TRS vs. Verilog

### 32-bit MIPS Integer Core

<table>
<thead>
<tr>
<th></th>
<th>CBA tc6a</th>
<th>LSI 10K</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Area (cells)</td>
<td>Clock</td>
</tr>
<tr>
<td>TRS</td>
<td>9521</td>
<td>10ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100MHz</td>
</tr>
<tr>
<td>Verilog RTL</td>
<td>8960</td>
<td>11.4ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>88MHz</td>
</tr>
<tr>
<td></td>
<td>30756</td>
<td>19.48ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>51MHz</td>
</tr>
<tr>
<td></td>
<td>29483</td>
<td>23.79ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42MHz</td>
</tr>
</tbody>
</table>

**TRS 1 day**
**Verilog 1 month**

*Dan Rosenband & James Hoe*
Architectural Derivatives

Non-pipelined
2-stage
3-stage

Other Dimensions:
Superscalar, Custom Instructions, Number of Registers, Word Size ...

Arvind, MIT Lab for Computer Science
Derivatives and Feedback

- Derivatives of a 32-bit 4-GPR embedded RISC processor
- Synopsys RTL Analyzer reports GTECH area and gate delays (*no wiring or load model*)

<table>
<thead>
<tr>
<th></th>
<th>simple</th>
<th>2-stage</th>
<th>3-stage</th>
<th>3-stage,2-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>30+X</td>
<td>max(18+X,25)</td>
<td>max(6+X,25)</td>
<td>max(8+X,31)</td>
</tr>
<tr>
<td>Delay(X=20)</td>
<td>50</td>
<td>38</td>
<td>26</td>
<td>31</td>
</tr>
<tr>
<td>Area</td>
<td>4334</td>
<td>5753</td>
<td>6378</td>
<td>9492</td>
</tr>
</tbody>
</table>

unit area=1 NAND  
unit delay=1 NAND
Application: ASPN Chips

Application-Specific Programmable Network (ASPN) Chips are based on a core architecture and a set of domain-specific building blocks.

TRAC allows rapid customization of ASPN designs with ASIC like performance for evolving needs and for different vertical markets within the communication space.