ESE535: Electronic Design Automation

Day 24: April 27, 2009
Processor Verification

ALU-RF Path

• Only a problem when next instruction depends on value written by immediately previous instruction
• ADD R3 ← R1 + R2
• ADD R4 ← R2 + R4
• ADD R5 ← R4 + R3

Solve with Bypass

Branch Path

• Only a problem when the branch instruction is a taken branch

Solve by
  – Speculating is not a taken branch
  – Preventing the speculative instruction from affecting state when branch occurs
Example

- Different implementations for same specification

 specification

 specification

 Today

- Specification/Implementation
- Abstraction Functions
- Correctness Condition
- Verification
- Self-Consistency

 Specification

- Abstract from Implementation
- Describes observable/correct behavior

 Implementation

- Some particular embodiment
  - Should have same observable behavior
    - Same with respect to important behavior
  - Includes many more details than spec.
    - How performed
    - Auxiliary/intermediate state

 Important Behavior

- Same output sequence for input sequence
  - Same output after some time?
- Timing?
  - Number of clock cycles to/between results?
  - Timing w/in bounds?
- Ordering?
Abstraction Function

- Map from implementation state to specification state
  - Use to reason about implementation correctness
  - Want to guarantee: $AF(Fi(q,i)) = Fs(AF(q),i)$

Familiar Example

- Memory Systems
  - Specification:
    - $W(A,D)$
    - $R(A) \rightarrow D$ from last D written to this address
  - Specification state: contents of memory
  - Implementation:
    - Multiple caches, VM, pipelined, Write Buffers...
    - Implementation state: much richer...

Memory AF

- Maps from
  - State of caches/WB/etc.
- To
  - Abstract state of memory
- Guarantee $AF(Fi(q,i)) = Fs(AF(q),i)$
  - Guarantee change to state always represents the correct thing

Abstract Timing

- For computer memory system
  - Cycle-by-cycle timing not part of specification
  - Must abstract out
- Solution:
  - Way of saying "no response"
    - Saying "skip this cycle"
    - Marking data presence
      - (tagged data presence pattern)

Filter to Abstract Timing

- Filter input/output sequence
- $Os(in) \rightarrow out$
- $FilterStall(Impl_{in}) = in$
- $FilterStall(Impl_{out}) = out$
- For all sequences $Impl_{in}$
  - $FilterStall(Oi(Impl_{in})) = Os(FilterStall(Impl_{in}))$

DLX Datapath

- DLX unpipelined datapath from H&P (Fig. 3.1 e2, A.17 e3)
Processors

• Pipeline is big difference between specification state and implementation state.
• Specification State:
  – Register contents (incl. PC)
  – Memory contents

Revised Pipeline

DLX repipelined datapath from H&P (Fig. 3.22 e2, A.24 e3)

Processors

• Pipeline is big difference between specification state and implementation state.
• Specification State:
  – Register contents (incl. PC)
  – Memory contents
• Implementation State:
  + Instruction in pipeline
  + Lots of bits
  + Many more states
  + State-space explosion to track

Compare

Observation

• After flushing pipeline,
  – Reduce implementation state to specification state
• Can flush pipeline with series of NOPs or stall cycles

Pipelined Processor Correctness

• w = input sequence
• w_f = flush sequence
  – Enough NOPs to flush pipeline state
• For all states q and prefix w
  – F_i(q,w w_f) Æ Fs(q,w w_f)
  – F_i(q,w w_f) Æ Fs(q,w)
• FSM observation
  – Finite state in pipeline
  – only need to consider finite w
Pipeline Correspondence

Equivalent

Equivalence

• Now have a logical condition for equivalence
• Need to show that it holds
  – Is a Tautology
• Or find a counter example

Ideas

Extract Transition Function

• From HDL
• Similar to what we saw for FSMs

Segregate Datapath

Burch&Dill Logic

• Quantifier-free
• Uninterpreted functions (datapath)
• Predicates with
  – Equality
  – Propositional connectives

• Extract Transition Function
• Segregate datapath
• Symbolic simulation on variables
  – For q, w's
• Case splitting search
  – Generalization of SAT
  – Uses implication pruning

• Big state blowup is in size of datapath
  – Represent data symbolically/abstractly
    • Independent of bitwidth
  – Not verify datapath/ALU functions as part of this
    • Can verify ALU logic separately using combinational verification techniques
    • Abstract/uninterpreted functions for datapath
B&D Logic

- Formula = \text{ite}(\text{formula}, \text{formula}, \text{formula})
  \begin{itemize}
    \item \text{psym}(\text{term}, \ldots \text{term})
    \item \text{pvar} | \text{true} | \text{false}
  \end{itemize}
- Term = \text{ite}(\text{formula}, \text{term}, \text{term})
  \begin{itemize}
    \item \text{fsym}(\text{term}, \ldots \text{term})
    \item \text{tvar}
  \end{itemize}

Sample

- Regfile:
  \begin{itemize}
    \item \text{ite} \text{stall}
    \item \text{regfile}
    \item \text{write} \text{regfile}
    \item \text{dest}
    \item \text{alu} \text{op}
    \item \text{read} \text{regfile} \text{src1}
    \item \text{read} \text{regfile} \text{src2})
  \end{itemize}

Sample Pipeline

Example Logic

- arg1:
  \begin{itemize}
    \item \text{ite} \text{or} \text{bubble-ex}
    \item \text{not} (= \text{src1} \text{dest-ex})
    \item \text{read}
    \item \text{ite} \text{bubble-wb}
    \item \text{regfile}
    \item \text{write} \text{regfile} \text{dest-wb} \text{result})
    \item \text{src1}
    \item \text{alu} \text{op-ex} \text{arg1} \text{arg2})
  \end{itemize}

Symbolic Simulation

- Create logical expressions for outputs/state
  - Taking initial state/inputs as variables

Case Splitting Search

- Satisfiability Problem
- Pick an unresolved variable
- Branch on true and false
- Push implications
- Bottom out at consistent specification
- Exit on contradiction
- Pragmatic: use memoization to reuse work
Review: What have we done?

- Reduced to simpler problem
  - Simple, clean specification
- Abstract Simulation
  - Explore all possible instruction sequences
- Abstraced the simulation
  - Focus on control
  - Divide and Conquer: control vs. arithmetic
- Used Satisifiability for reachability in search in abstract simulation

Achievable

- Burch&Dill: Verify 5-stage pipeline DLX
  - 1 minute in 1994
    - On a 40MHz R3400 processor
- Modern machines 30+ pipestages
  - …and many other implementation embellishments

Self Consistency

- Compare same implementation in two different modes of operation
  - (which should not affect result)
- Compare pipelined processor
  - To self w/ NOPs separating instructions
    - So only one instruction in pipeline at a time

Self-Consistency

- w = instruction sequence
- S(w) = w with no-ops
- Show: Forall q, w
  - F(q,w) = F(q,S(w))

Sample Result

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Gates</th>
<th>Latches</th>
<th>Simulation Time (s)</th>
<th>Execution Time (hr)</th>
<th>Equivalent Simulation Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>142</td>
<td>2296</td>
<td>0.3</td>
<td>3</td>
<td>5 x 10^11</td>
</tr>
<tr>
<td>B</td>
<td>7803</td>
<td>11708</td>
<td>19</td>
<td>2</td>
<td>2 x 10^11</td>
</tr>
</tbody>
</table>

Table 1: Self-consistency checking results.

[Jones, Seger, Dill/FMCAD 1996]

n.b. Jones&Seger at Intel
Sample Result

Verification running on P2-200MHz
[Skakkebæk, Jones, and Dill / CAV 1998]

Key Idea

- Implementation State reduces to Specification state after finite series of operations
- Abstract datapath to avoid dependence on bitwidth

Admin

- Last Class
- Assignment 6 due May 12th (noon)

Big Ideas

- Proving Invariants
- Divide and Conquer
- Exploit Structure