Can we pipeline?

Pipelining: 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

ALU-RF Path

- Only a problem when next instruction depends on value written by immediately previous instruction
- Solve with Bypass
Branch Path

• Only a problem when the 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

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

Unimportant Behavior?

• What behaviors might be unimportant?

“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: \( AF(Fi(q,i)) = Fs(AF(q),i) \)
  • Similar to saying the composite state machines always agree on output (state)
  • ...but have more general notion of outputs and timing

Recall FSM

• Equivalent FSMs with different number of states

Recall FSM

• Maybe right is specification
  • \( AF(s1) = q1, AF(s3) = q1 \)
  • \( AF(s2) = q2, AF(s4) = q2 \)
  • \( AF(s0) = q0 \)
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

Memory: L1, writeback

- Memory with L1 cache
  - L1 cache is extra state
  - Another L1.capacity words of data
  - Check L1 cache first for data on read
  - Miss $\Rightarrow$ load into cache
  - Writes update mapping for address in L1
  - When address evicted form L1
    - write-back to main memory

Memory: L1, writeback

- Specification State:
  - one memory with addr:data mappings
  - $M(a) = MM[a]$
  - L1 writeback cache implementation
  - $AF(L1+M)$: forall $a$
    - If $a$ in L1
      - $M(a)=L1[a]$
    - else
      - $M(a)=MM[a]$

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)
  - Example: stall while fetch data into L1 cache
Filter to Abstract Timing

- Filter input/output sequence
- View computation as: $O_s(in) \rightarrow out$
- $FilterStall(Imp_{in}) = in$
- $FilterStall(Imp_{out}) = out$
- For all sequences $Imp_{in}$
  - $FilterStall(Oi(Imp_{in})) = O_s(FilterStall(Imp_{in}))$

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:
  - PC, RF, Data Memory
- Implementation State:
  + Instruction in pipeline
  + Lots of bits
    - Many more states
    - State-space explosion to track

Revised Pipeline

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

Compare
Return to L1, writeback

• How does main memory state relate to specification state after an L1 cache flush?
  – L1 cache flush = force writeback on all entries of L1

Observation

• After flushing pipeline,
  – Reduce implementation state to specification state (RF, PC, Data Mem)
• Can flush pipeline with series of NOOPs or stall cycles

Pipelined Processor Correctness

• $w =$ input sequence
• $w_f =$ flush sequence
  – Enough NOOPs to flush pipeline state
• For all states $q$ and prefix $w$
  – $F_i(q,w) \Rightarrow F_s(q,w_f)$
  – $F_i(q,w) \Rightarrow F_s(q,w)$
• FSM observation
  – Finite state in pipeline
  – only need to consider finite $w$

Pipeline Correspondence

<table>
<thead>
<tr>
<th>Old Impl State</th>
<th>$F_{impl}(\cdot, f_{impl})$</th>
<th>New Impl State</th>
<th>$F_{impl}(\cdot, f_{impl})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_{impl}(\cdot, f_{impl})$</td>
<td>$F_{impl}(\cdot, f_{impl})$</td>
<td>$F_{impl}(\cdot, f_{impl})$</td>
<td>$F_{impl}(\cdot, f_{impl})$</td>
</tr>
</tbody>
</table>

Equivalence

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

[Burch+Dill, CAV'94]
Ideas

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

Extract Transition Function

- From HDL
- Similar to what we saw for FSMs

Segregate Datapath

- 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

Burch&Dill Logic

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

B&D Logic

- Formula = \texttt{ite}(formula, formula, formula)
  | (term=term)
  | psym(term,…term)
  | pvar | true | false
- Term = \texttt{ite}(formula,term,term)
  | fsym(term,…term)
  | tvar

Sample

- Regfile:
  - (ite stall
    regfile
    (write regfile
    dest
    (alu op
    (read regfile src1)
    (read regfile src2))))
Sample Pipeline

Example Logic

- arg1:
  - (ite (or bubble-ex
    (not (= src1 dest-ex)))
    (read
     (ite bubble-wb
      regfile
      (write regfile dest-wb result))
    src1)
  (alu op-ex arg1 arg2))

Symbolic Simulation

- Create logical expressions for outputs/state
  - Taking initial state/inputs as variables
- E.g. (ALU op2
  (ALU op1 rf-init1 rf-init2)
  rf-init3)

Case Splitting Search

- Satisfiability Problem
- Pick an unresolved variable
  - (= src1 dest-ex)
  - (= 0
    (ALU op2
     (ALU op1 rf-init1 rf-init2)
     rf-init3))

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
- Abstracted the simulation
  - Focus on control
  - Divide and Conquer: control vs. arithmetic
- Used Satisfiability 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+ pipeline stages
  - ...and many other implementation embellishments

Self-Consistency

- Compare same implementation in two different modes of operation
  - (which should not affect result)
  - Examples of different modes of operation that should behave the same?

Self-Consistency

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

Sample Result

- A – stream processor
- B – multithread pipeline

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Gates</th>
<th>Latches</th>
<th>Simulation</th>
<th>Execution</th>
<th>Variables</th>
<th>Time (h)</th>
<th>Equivalent</th>
<th>Simulation Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8402</td>
<td>2568</td>
<td>49</td>
<td>3</td>
<td>11709</td>
<td>10</td>
<td>6 x 10^4</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>2069</td>
<td>11709</td>
<td>49</td>
<td>10</td>
<td>2 x 10^9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Self-consistency checking results.

[Jones, Seger, Dill/FMCAD 1996]

n.b. Jones & Seger at Intel
Sample Result: OoO processor

<table>
<thead>
<tr>
<th>IMPL-ABS Verification</th>
<th>IMPL Reach, Inc Verification</th>
<th>IMPL-ABS Verification</th>
<th>AVS-ISA Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (sec)</td>
<td>Core Splits</td>
<td>CPU (sec)</td>
<td>Core Splits</td>
</tr>
<tr>
<td>Base Case</td>
<td>1.1</td>
<td>10</td>
<td>0.1</td>
</tr>
<tr>
<td>Fron</td>
<td>44.1</td>
<td>2.5</td>
<td>22.4</td>
</tr>
<tr>
<td>Desktop</td>
<td>49.3</td>
<td>12.0</td>
<td>343.1</td>
</tr>
<tr>
<td>Workload</td>
<td>35.0</td>
<td>8.4</td>
<td>32.4</td>
</tr>
<tr>
<td>Setup</td>
<td>38.3</td>
<td>8.9</td>
<td>37.3</td>
</tr>
</tbody>
</table>

Verification running on P2-200MHz


Key Idea Summary

- Implementation state reduces to specification state after finite series of operations
- Abstract datapath to avoid dependence on bitwidth
- Abstract simulation (reachability)
  - Show same outputs for any input sequence
- State→state transform
  - Can reason about finite sequence of steps

Admin

- Last Class
- Assignment 8 out
  - due May 9th (noon)
  - Late assignments will not receive partial credit
  - André traveling May 1—6
    - Ask clarifying questions before May 1
- Normal office hours Tuesday (tomorrow)
  - None on May 3rd
- Course evaluations online

Big Ideas

- Proving Invariants
- Divide and Conquer
- Exploit Structure