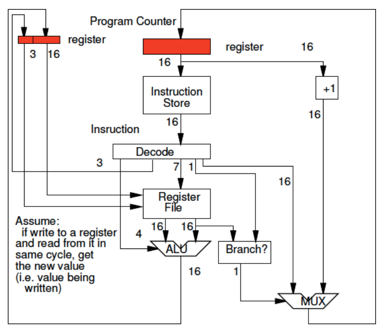
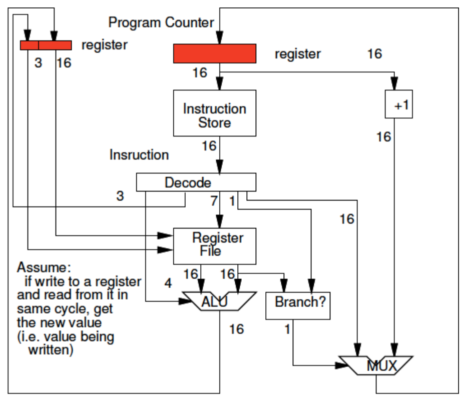
ESE535: Electronic Design Automation

Day 26: April 29, 2015

Processor Verification

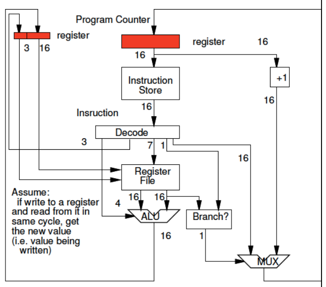
Can we pipeline?





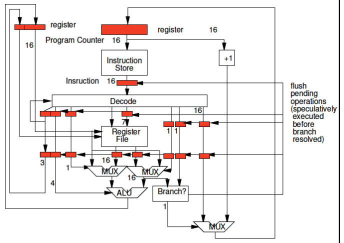
Pipelining: ALU-RF Path

- 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


Program Counter
register
register
Instruction Store
+1
Insruction
Decode
Register File
Assume: if write to a register and read from it in same cycle, get the new value (i.e. value being written)
ALU
Branch?
MUX

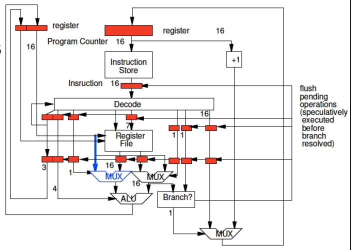
ALU-RF Path

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



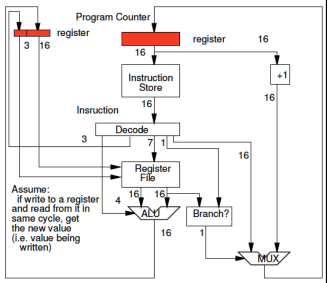
ALU-RF Path

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



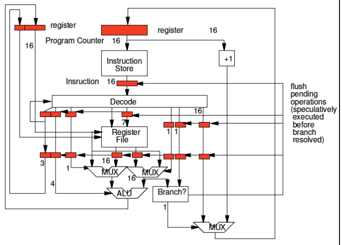
Branch Path

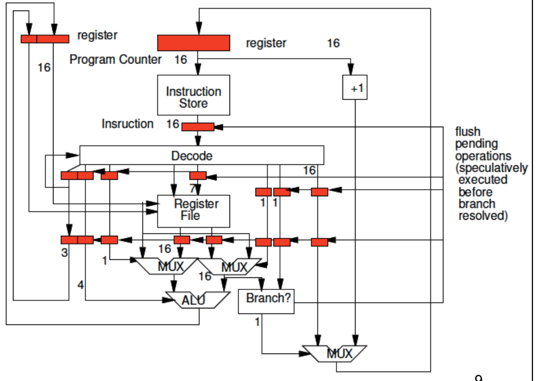
- Only a problem when the instruction is a taken branch



Branch Path

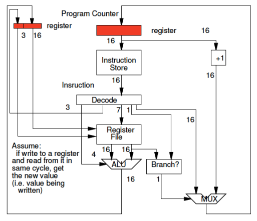
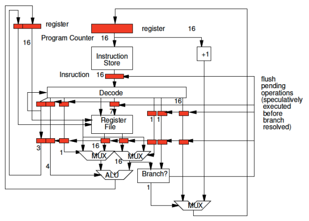
- Only a problem when the instruction is a taken branch
- Solve by
  - Speculating is not a taken branch
  - Preventing the speculative instruction from affecting state when branch occurs





Example

- Different implementations for same specification

Today

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

Behavioral
(C, MATLAB, …)

Arch. Select
Schedule

RTL

FSM assign
Two-level,
Multilevel opt.
Covering
Retiming

Gate Netlist

Placement
Routing

Layout

Masks

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?

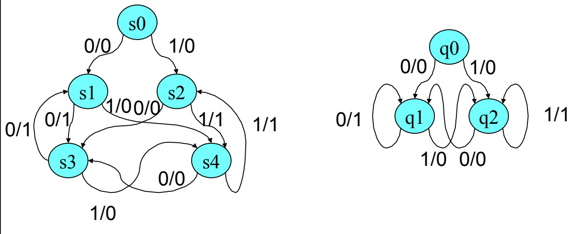
Abstraction Function

- 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

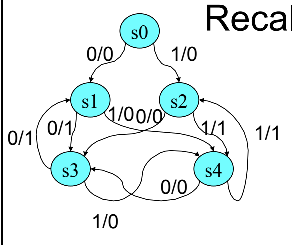
$$\begin{tikzcd} \bullet \arrow[r, "AF"] \arrow[d, "Fi"'] & \bullet \arrow[d, "Fs"] \\ \bullet \arrow[r, "AF"'] & \bullet \end{tikzcd}$$

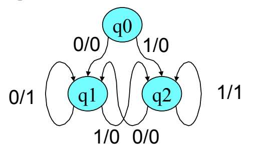
Recall FSM

- 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

$$\begin{tikzcd} \bullet \arrow[r, "AF"] \arrow[d, "Fi"'] & \bullet \arrow[d, "Fs"] \\ \bullet \arrow[r, "AF"'] & \bullet \end{tikzcd}$$

Familiar Example

- Memory Systems
  - Specification:
    - W(A,D)
    - R(A)→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→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]

$$\begin{tikzcd} \bullet \arrow[r, "AF"] \arrow[d, "Fi"'] & \bullet \arrow[d, "Fs"] \\ \bullet \arrow[r, "AF"'] & \bullet \end{tikzcd}$$

Memory: L1, writeback

- Specification State:
  - one memory with addr:data mappings
  - M(a) = MM[a]
- What are several (different) implementation states that map to same specification state?
  - Concrete: M(0x100C)=0xBEC1

$$\begin{tikzcd} \bullet \arrow[r, "AF"] \arrow[d, "Fi"'] & \bullet \arrow[d, "Fs"] \\ \bullet \arrow[r, "AF"'] & \bullet \end{tikzcd}$$

Abstract Timing

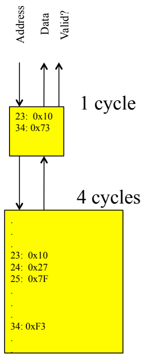
- For computer memory system
  - Cycle-by-cycle timing **not** part of specification
  - Must abstract out
- Solution:
  - Way of saying "no response"
    - Saying "skip this cycle"
    - Marking data presence
      - (tagged data presence pattern)
    - Example: stall while fetch data into L1 cache

Filter to Abstract Timing

- Filter input/output sequence
- View computation as: Os(in)→out
- FilterStall(Impl$_{in}$) = in
- FilterStall(Impl$_{out}$) = out
- Forall sequences Impl$_{in}$
  - FilterStall(Oi(Impl$_{in}$)) = Os(FilterStall(Impl$_{in}$))

Filter Example

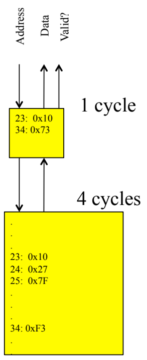
- Only one cache + main memory
- L1 answers in 1 cycle
- 4 cycle delay to fetch from main memory into L1



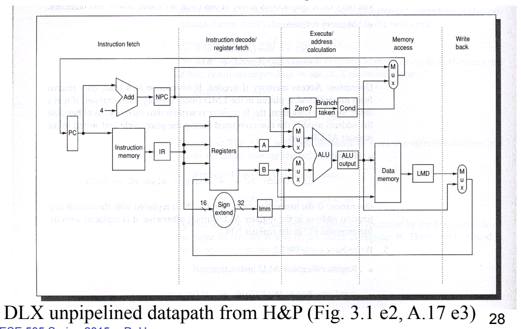
Filter Example

- Read Sequence:
  - 23 (request on cycle 0)
  - 24
  - 34
- Cycle by cycle results?

0:
1:
2:
3:
4:
5:
6:
7:
8:
9:
10:



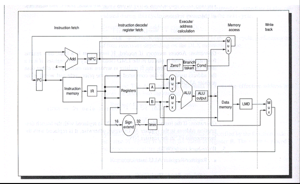
DLX Datapath



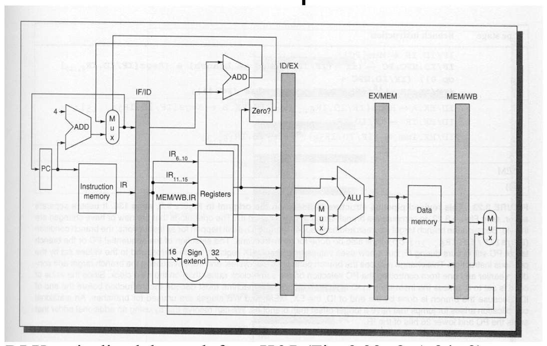
DLX unpipelined datapath from H&P (Fig. 3.1 e2, A.17 e3)

Processors

- Pipeline is big difference between specification state and implementation state.
- What is specification state?

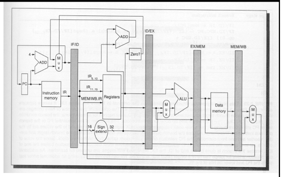


Revised Pipeline

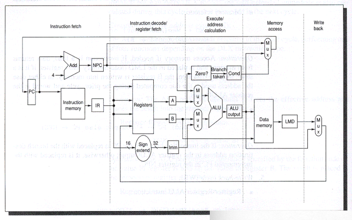


DLX repipelined datapath from H&P (Fig. 3.22 e2, A.24 e3)

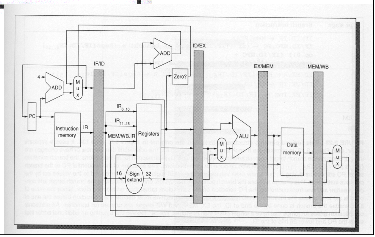
Processors



- Pipeline is big difference between specification state and implementation state.
- Specification State:
  - PC, RF, Data Memory
- Implementation State:
  - + Instruction in pipeline
  - + Lots of bits
    - Many more states
    - State-space explosion to track

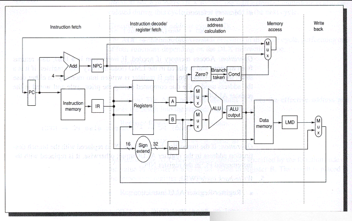


Compare



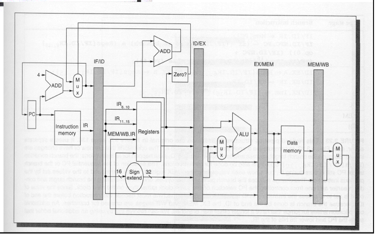
Return to L1, writeback

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



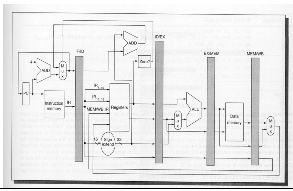
Compare

How make the shared state the same?



Observation

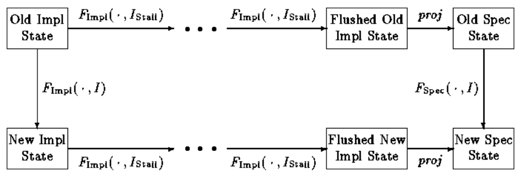
- After flushing pipeline,
  - Reduce implementation state to specification state (RF, PC, Data Mem)
- Can flush pipeline with series of NOOPs or stall cycles
- NOOP "No Operation"
  - An instruction that does not change any state
  - (except the PC)



Pipelined Processor Correctness

- w = input sequence
- w_f = flush sequence
  - Enough NOOPs to flush pipeline state
- Forall states q and prefix w
  - $Fi(q,w\ w_f) \rightarrow Fs(q,w\ w_f)$
  - $Fi(q,w\ w_f) \rightarrow Fs(q,w)$
- FSM observation
  - Finite state in pipeline
  - only need to consider finite w

Pipeline Correspondence



[Burch+Dill, CAV'94]

Equivalence

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

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

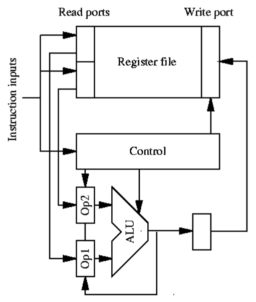
B&D Logic

- Formula = **ite**(formula, formula, formula)
  - | (term=term)
  - | psym(term,…term)
  - | pvar | **true** | **false**
- Term = **ite(**formula,term,term)
  - | fsym(term,…term)
  - | tvar

Sample

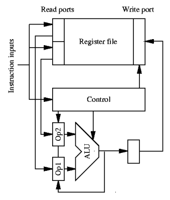
- Regfile:
  - (ite stall
    regfile
    (write regfile
    dest
    (alu op
    (read regfile src1)
    (read regfile src2))))

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)

Example

- ADD R3←R1+R2
- ADD R4←R2+R4
- ADD R5←R4+R3

After

- R1: rf-init1
- R2: rf-init2
- R3: (ALU add rf-init1 rf-init2)
- R4: (ALU add rf-init2 rf-init4)
- R5: (ALU add (ALU add rf-init2 rf-init4) (ALU add rf-init1 rf-init2))

This is what checking equivalence on.

Case Splitting Search

- Satisfiability Problem
- Pick an unresolved variable
  - (= src1 dest-ex)
    - [relevant to bypass]
  - (= 0

```
    (ALU op2
          (ALU op1 rf-init1 rf-init2)
          rf-init3)
 )
```

    - [relevant to branching]

Case Splitting

- Some case-splitting will be
  - Ops – explore all combination of op sequences
  - Registers – all interactions of registers among ops (ops in pipeline)
  - Stalls – all possible timing of stalls
- Like picking all output conditions from a state
  - Case-splitting – picking cube cases

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

Case Split Example: Cache

- Only three operations: op A D
  - R A
  - W A D
  - NOOP
- Two implementations
  - One with single memory
  - One with cache
- Want to be sure all sequences of operations return same results

Specification and Implementation

- Specification

```
(ite (op==R)
   (read M A)
   (ite (op==W)
      (write M A D)
      (noop)
   )
```

- Cached Implementation

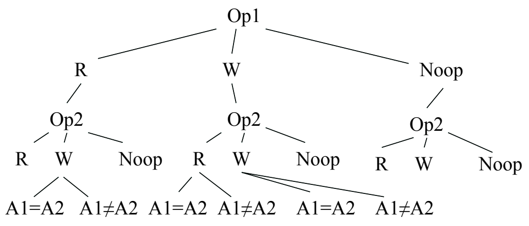
```
(ite (op==R)
   (ite (in L1 A)
      (read L1 A)
      (seq
         (write L1 A (read M A))
         (read M A))
   (ite (op==W)
      (seq
         (write M A D)
         (ite (in L1 A)
            (write L1 A D)))
      (noop))))
```

Sequence

- (Op1 A1 D1)
- (Op2 A2 D2)
- (Op3 A3 D3)
- (Op4 A4 D4)
- (Op5 A5 D5)

- Forall (Op1,Op2,Op3,Op4,Op5, A1,A2,A3,A4,A5,D1,D2,D3,D4,D5)
  (Spec ((Op1 A1 D1) …. (Op5 A5 D5))
  == (Cache ((Op1 A2 D1) … (Op5 A5 D5))

Case Split



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

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

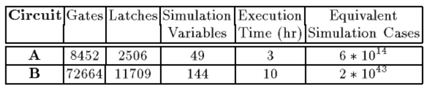
- Compare same implementation in two different modes of operation
  - (which should not affect result)
- Compare pipelined processor
  - To self w/ NOOPs separating instructions
    - So only one instruction in pipeline at a time
  - Why might this be important?

Self-Consistency

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

Sample Result

- A – stream processor
- B – multithread pipeline

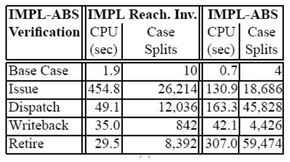


Circuit	Gates	Latches	Simulation Variables	Execution Time (hr)	Equivalent Simulation Cases
A	8452	2506	49	3	$6 * 10^{14}$
B	72664	11709	144	10	$2 * 10^{43}$

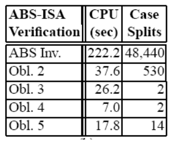
Table 1. Self-consistency checking results.

[Jones, Seger, Dill/FMCAD 1996]
n.b. Jones&Seger at Intel

Sample Result: OoO processor



IMPL-ABS Verification	**IMPL Reach. Inv.**		**IMPL-ABS**	
	CPU (sec)	Case Splits	CPU (sec)	Case Splits
Base Case	1.9	10	0.7	4
Issue	454.8	26,214	130.9	18,686
Dispatch	49.1	12,036	163.3	45,828
Writeback	35.0	842	42.1	4,426
Retire	29.5	8,392	307.0	59,474



ABS-ISA Verification	**CPU (sec)**	**Case Splits**
ABS Inv.	222.2	48,440
Obl. 2	37.6	530
Obl. 3	26.2	2
Obl. 4	7.0	2
Obl. 5	17.8	14

Verification running on P2-200MHz

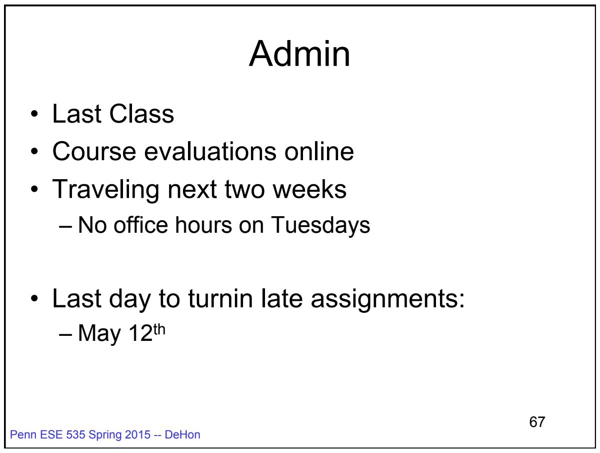
[Skakkebæk, Jones, and Dill / CAV 1998,
Formal Methods in System Design v20, p139, 2002]

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

Big Ideas

- Proving Invariants
- Divide and Conquer
- Exploit Structure



Admin

- Last Class
- Course evaluations online
- Traveling next two weeks
  - No office hours on Tuesdays
- Last day to turnin late assignments:
  - May 12th