CMSC 611: Advanced Computer Architecture

Pipelining (2)

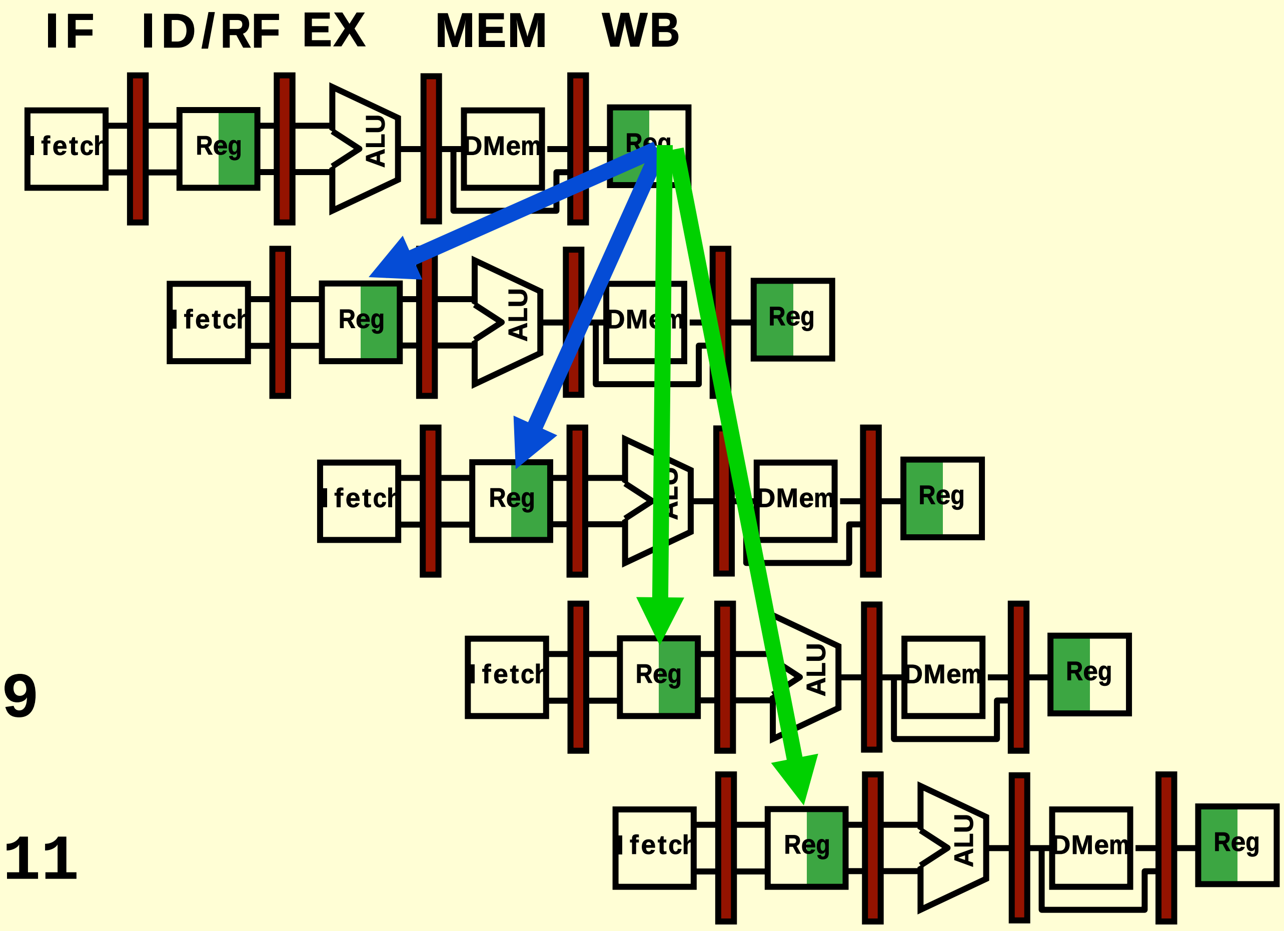
Pipeline Hazards

- Cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types
  - Structural hazard: attempt to use a resource two different ways at same time
    - Single memory for instruction and data
  - Data hazard: attempt to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazard: attempt to make a decision before condition is evaluated
    - branch instructions
- Hazards can always be resolved by waiting

Data Hazards

Time (clock cycles)

Instr. Order



add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or   r8,r1,r9

xor r10,r1,r11

Slide: David Culler

Three Generic Data Hazards

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Data Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- Write After Read (WAR)
  Instr_J writes operand before Instr_I reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" in compilers.
  - This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Slide: David Culler

Three Generic Data Hazards

- Write After Write (WAW)
  Instr_J writes operand before Instr_I writes it.

```
I: mul r1,r4,r3
J: add r1,r2,r3
K: sub r6,r1,r7
```

- Called an "output dependence" in compilers
  - This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Do see WAR and WAW in more complicated pipes

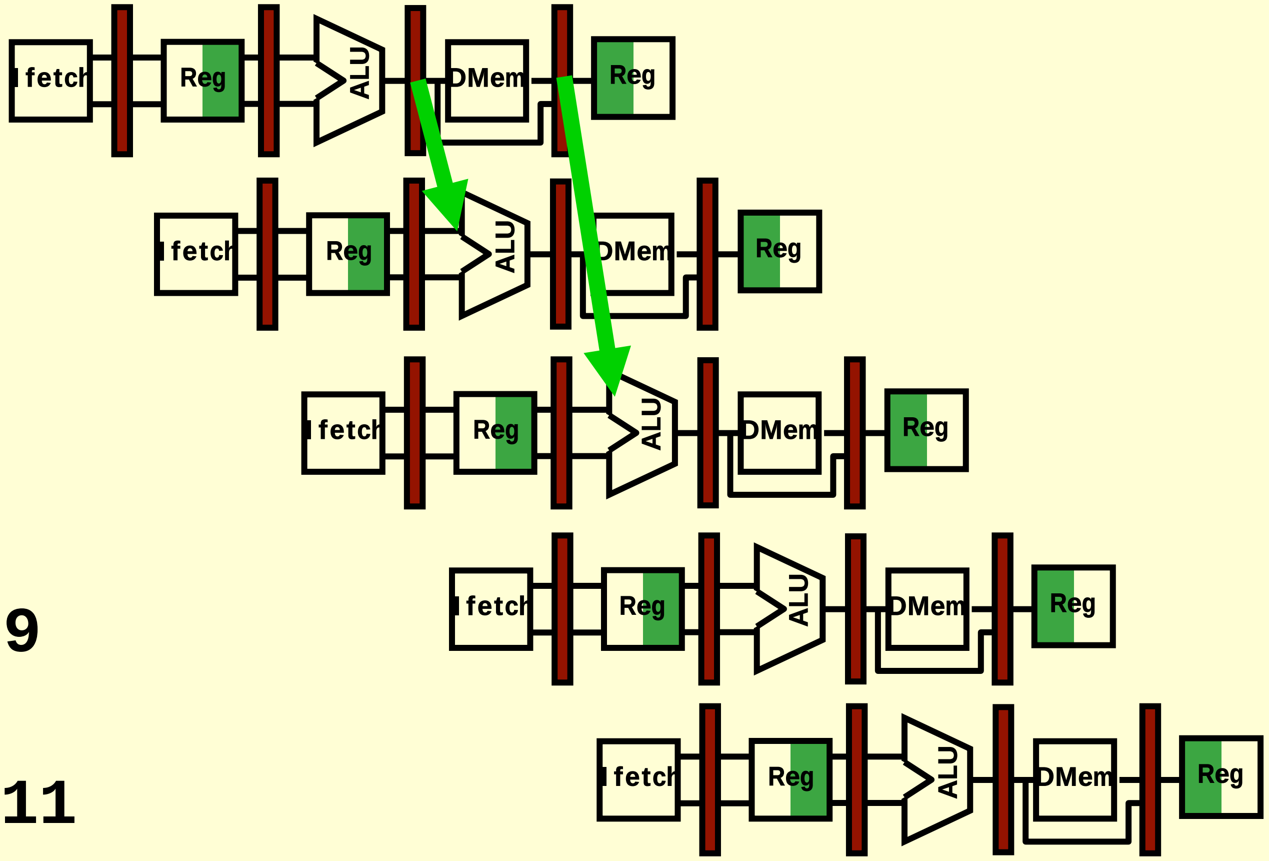
Slide: David Culler

Forwarding to Avoid Data Hazard

Time (clock cycles)

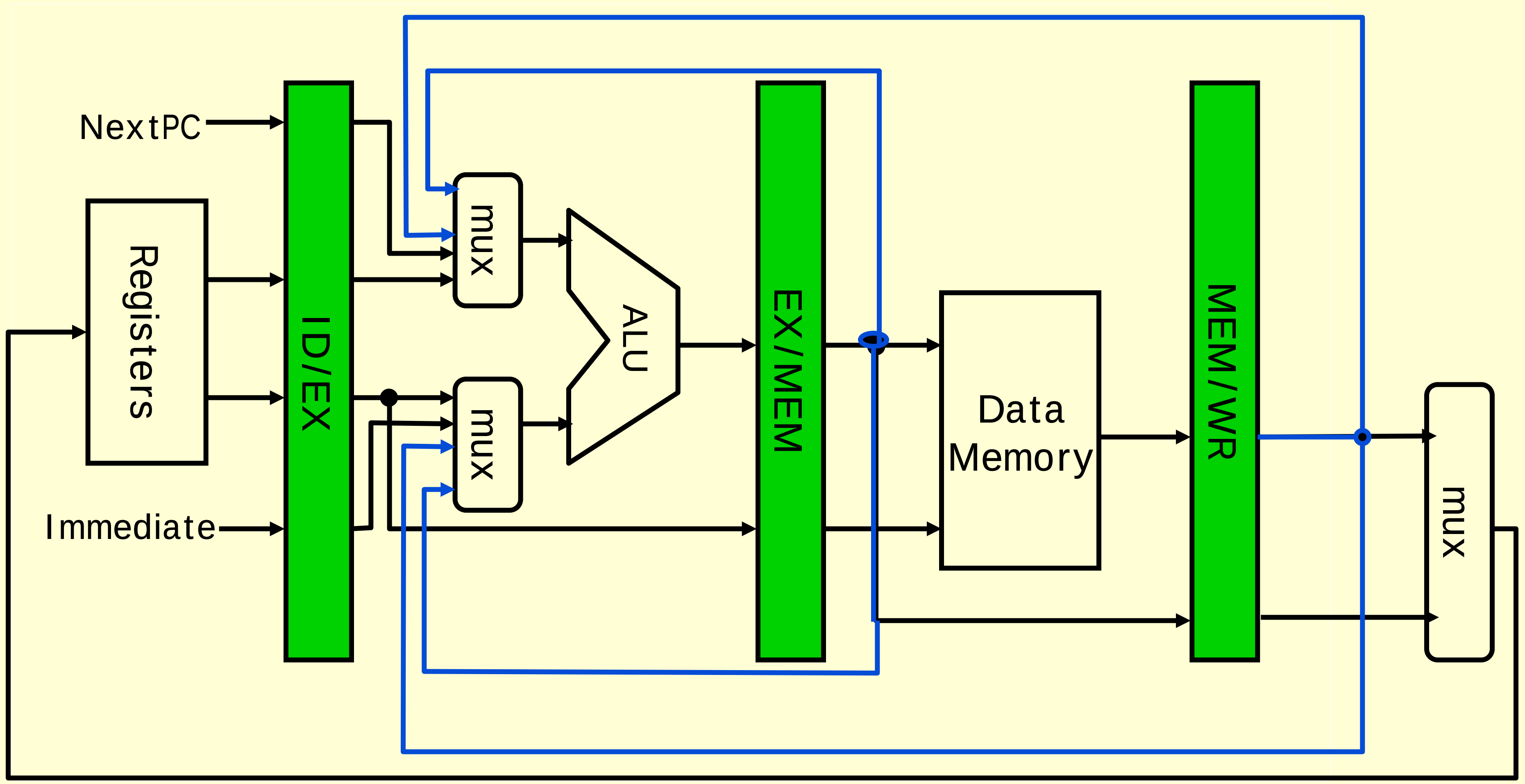
Instr. Order

```
add r1,r2,r3
sub r4,r1,r3
and r6,r1,r7
or   r8,r1,r9
xor r10,r1,r11
```



Slide: David Culler

HW Change for Forwarding



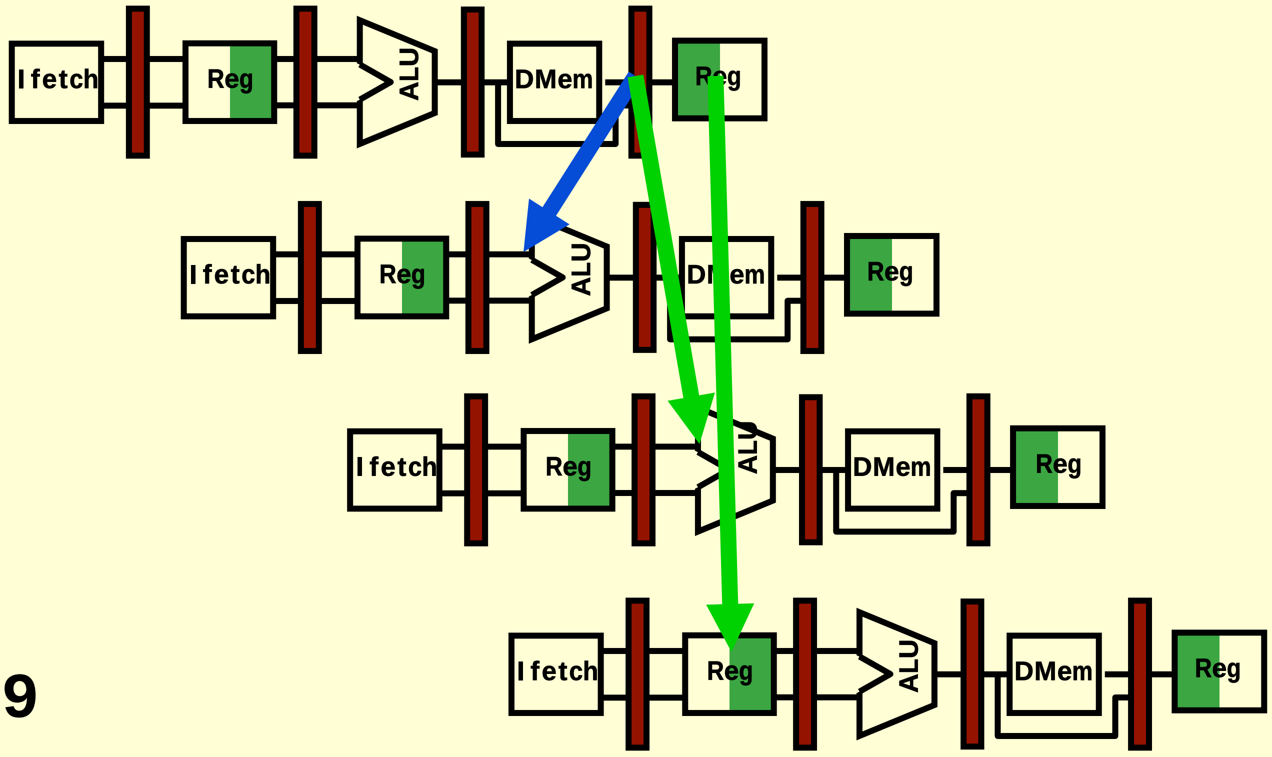
Slide: David Culler

Data Hazard Even with Forwarding

Time (clock cycles)

Instr. Order

```
lw r1, 0(r2)
sub r4,r1,r6
and r6,r1,r7
or   r8,r1,r9
```



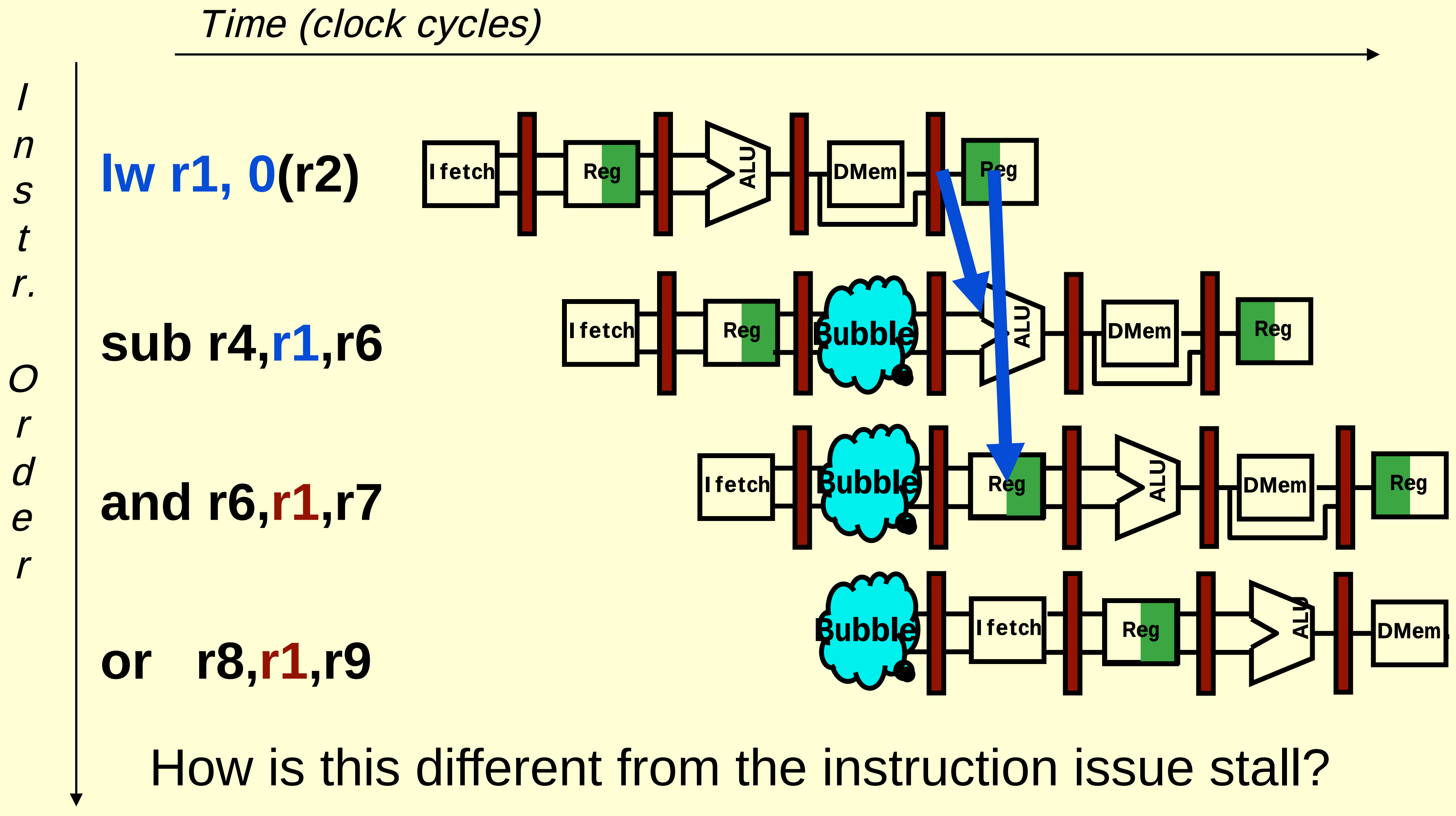
Resolving Load Hazards

- Adding hardware? How? Where?
- Detection?
- Compilation techniques?

- What is the cost of load delays?

Slide: David Culler

Resolving the Load Data Hazard



How is this different from the instruction issue stall?

Slide: David Culler

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

```
LW   Rb,b
LW   Rc,c
ADD  Ra,Rb,Rc
SW   a,Ra
LW   Re,e
LW   Rf,f
SUB  Rd,Re,Rf
SW   d,Rd
```

Fast code:

```
LW   Rb,b
LW   Rc,c
LW   Re,e
ADD  Ra,Rb,Rc
LW   Rf,f
SW   a,Ra
SUB  Rd,Re,Rf
SW   d,Rd
```

Slide: David Culler

Instruction Set Connection

- What is exposed about this organizational hazard in the instruction set?
- k cycle delay?
  - bad, CPI is not part of ISA
- k instruction slot delay
  - load should not be followed by use of the value in the next k instructions
- Nothing, but code can reduce run-time delays
- MIPS did the transformation in the assembler

Slide: David Culler

Pipeline Hazards

- Cases that affect instruction execution semantics and thus need to be detected and corrected
- Hazards types
  - Structural hazard: attempt to use a resource two different ways at same time
    - Single memory for instruction and data
  - Data hazard: attempt to use item before it is ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazard: attempt to make a decision before condition is evaluated
    - branch instructions
- Hazards can always be resolved by waiting

Control Hazard on Branches
Three Stage Stall

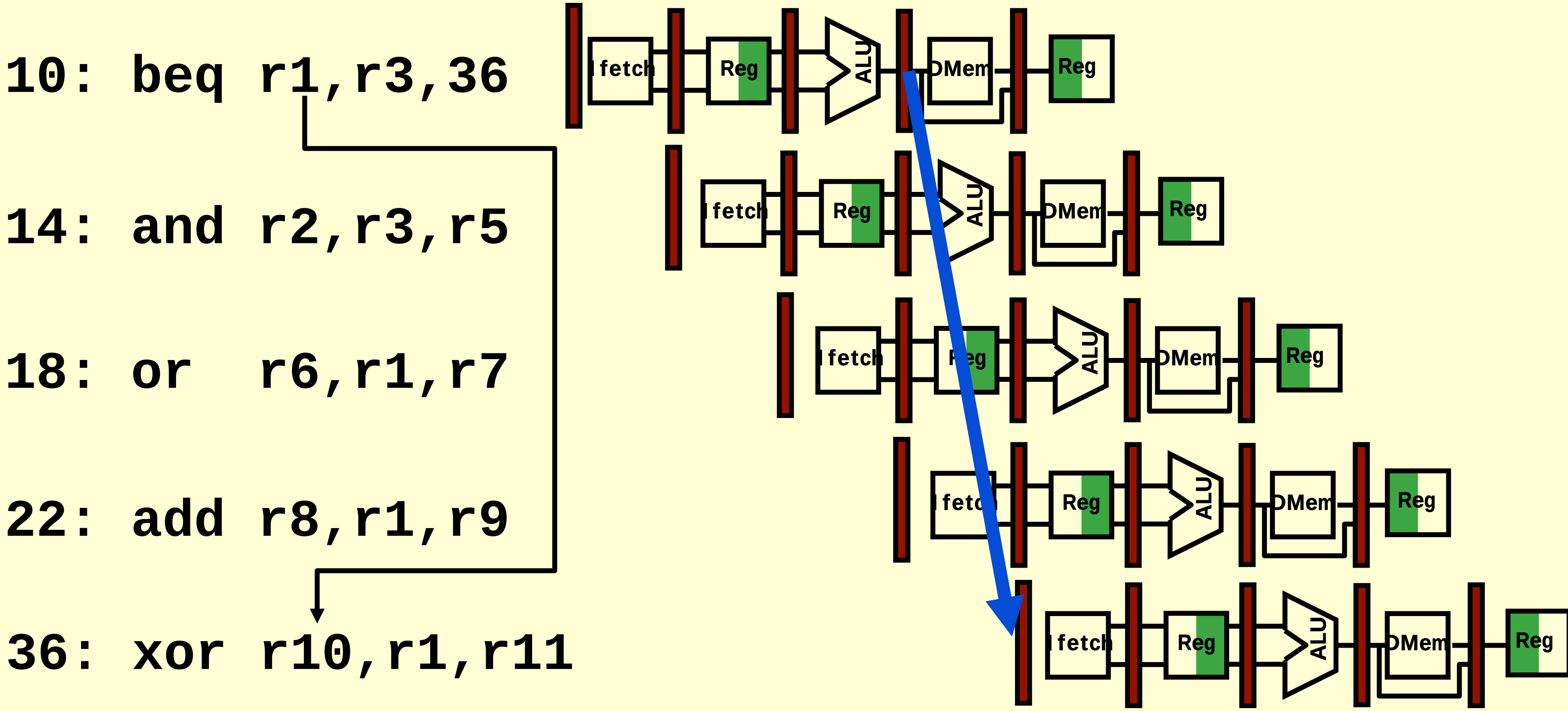
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



Slide: David Culler

Example: Branch Stall Impact

- If 30% branch, 3-cycle stall significant!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Slide: David Culler

Pipelined MIPS Datapath

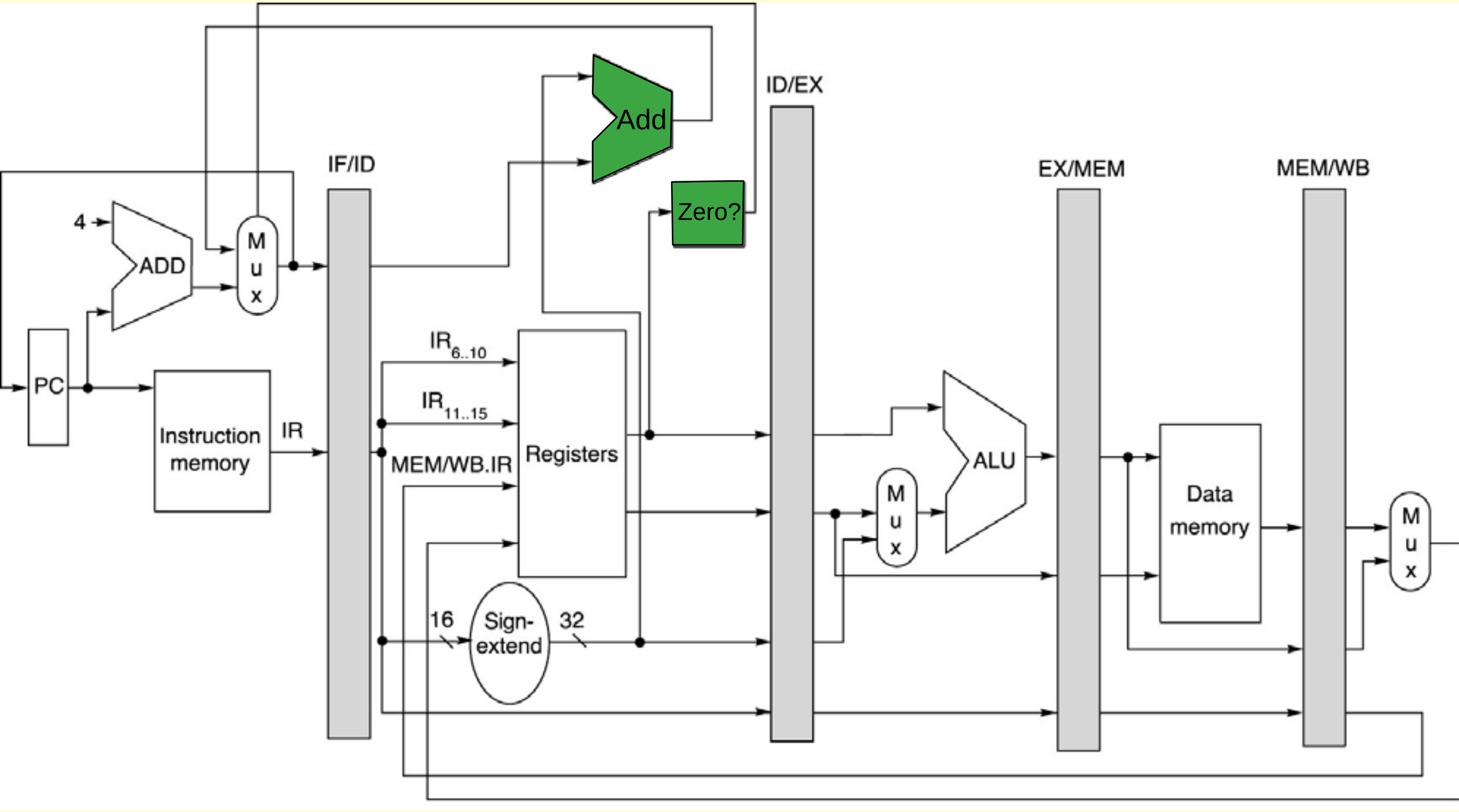


Figure: Dave Patterson

Four Branch Hazard Alternatives

1. Stall until branch direction is clear
2. Predict Branch Not Taken
   - Execute successor instructions in sequence
   - “Squash” instructions in pipeline if branch taken
   - Advantage of late pipeline state update
   - 47% MIPS branches not taken on average
   - PC+4 already calculated, so use it to get next instruction
3. Predict Branch Taken
   - 53% MIPS branches taken on average
   - But haven’t calculated branch target address in MIPS
     - MIPS still incurs 1 cycle branch penalty
     - Other machines: branch target known before outcome

Slide: David Culler

Four Branch Hazard Alternatives

4. Delayed Branch
   - Define branch to take place AFTER a following instruction

     branch instruction

     $$\left.\begin{array}{l}\text{sequential successor}_1\\ \text{sequential successor}_2\\ \text{........}\\ \text{sequential successor}_n\end{array}\right\}\textbf{Branch delay of length } \boldsymbol{n}$$

     ....

     branch target if taken

   - 1 slot delay allows proper decision and branch target address in 5 stage pipeline
   - MIPS uses this

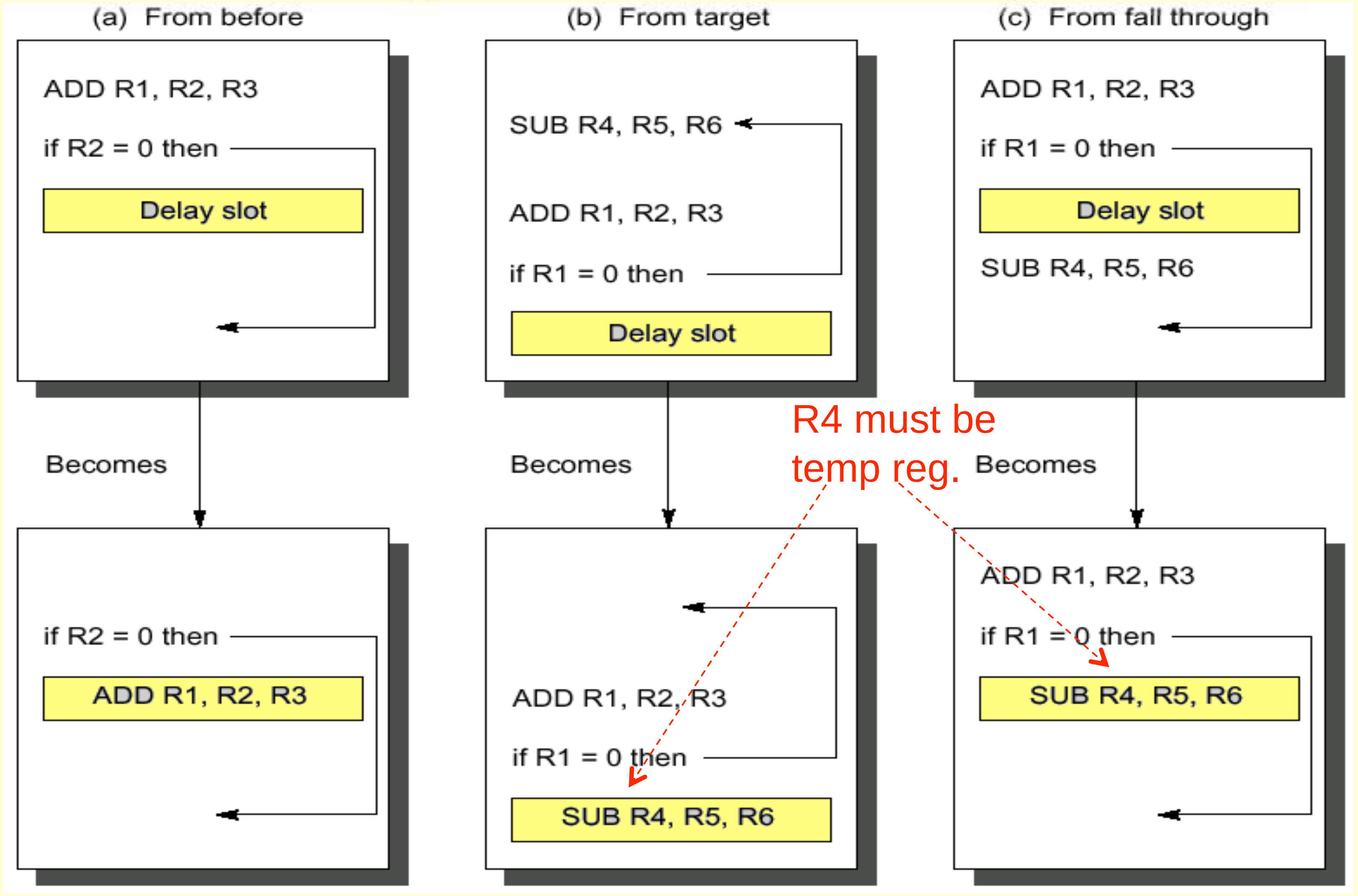
Slide: David Culler

Delayed Branch

- Where to get branch delay slot instructions?
  - Before branch instruction
  - From the target address
    - only valuable when branch taken
  - From fall through
    - only valuable when branch not taken
  - Canceling branches allow more slots to be filled
- Compiler effectiveness for single delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - 48% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Slide: David Culler

Scheduling Branch-Delay Slots



Best scenario **Good for loops** **Good taken strategy**

Branch-Delay Scheduling Requirements

Scheduling Strategy	Requirements	Improves performance when?
(a) From before	Branch must not depend on the rescheduled instructions	Always
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge programs if instructions are duplicated.
(c) From fall through	Must be okay to execute instructions if branch is taken.	When branch is not taken.

- Limitation on delayed-branch scheduling arise from:
  - Restrictions on instructions scheduled into the delay slots
  - Ability to predict at compile-time whether a branch is likely to be taken
- May have to fill with a no-op instruction
  - Average 30% wasted
- Additional PC is needed to allow safe operation in case of interrupts (more on this later)

Example: Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}}$$
$$= \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume:

14% Conditional & Unconditional

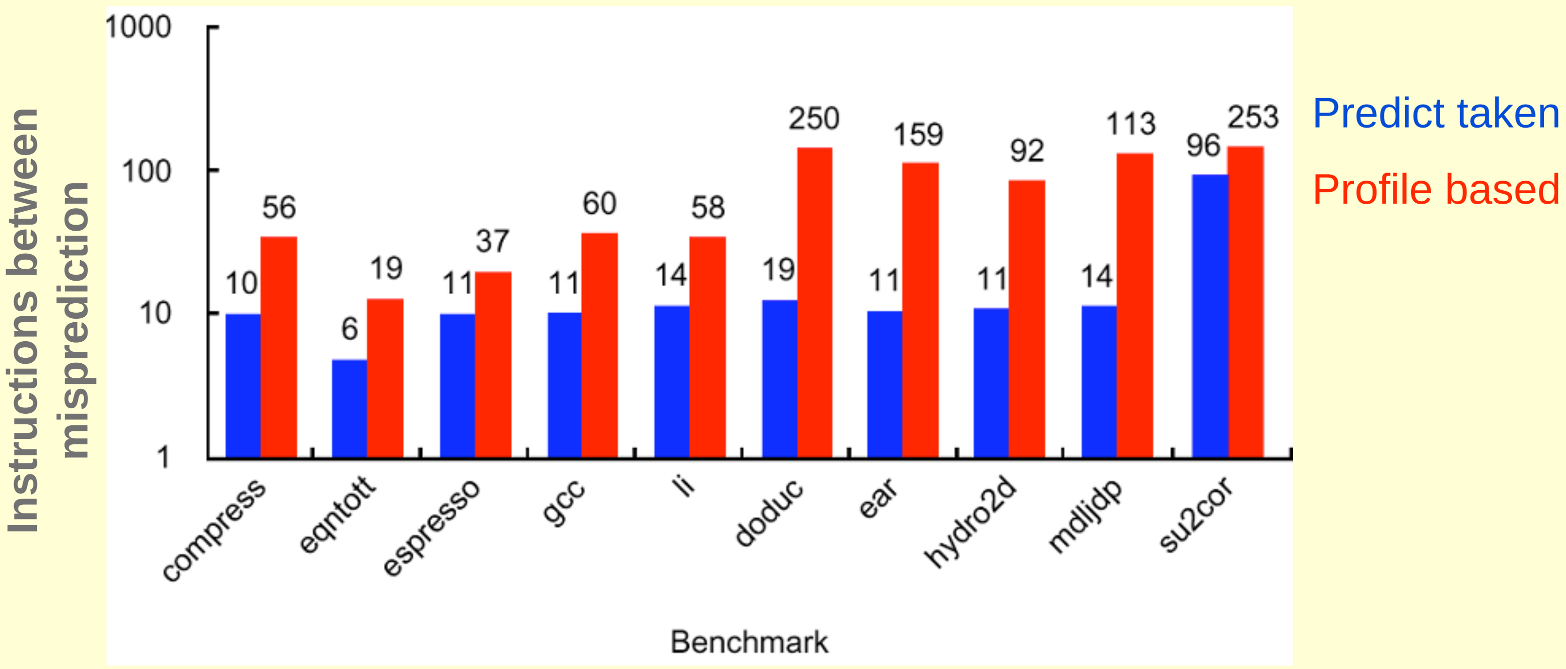
65% Taken; 52% Delay slots not usefully filled

Scheduling Scheme	***Branch Penalty***	***CPI***	***Pipeline Speedup***	***Speedup vs stall***
Stall pipeline	3.00	1.42	3.52	1.00
Predict taken	1.00	1.14	4.39	1.25
Predict not taken	1.00	1.09	4.58	1.30
Delayed branch	0.52	1.07	4.66	1.32

Slide: David Culler

Static Branch Prediction

- Examination of program behavior
  - Assume branch is usually taken based on statistics but misprediction rate still 9%-59%
- Predict on branch direction forward/backward based on statistics and code generation convention
  - Profile information from earlier program runs



Exception Types

- I/O device request
- Breakpoint
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Privilege violation
- Hardware and power failure

Exception Requirements

- Synchronous vs. asynchronous
  - I/O exceptions: Asyncronous
    - Allow completion of current instruction
  - Exceptions within instruction: Synchronous
    - Harder to deal with
- User requested vs. coerced
  - Requested predictable and easier to handle
- User maskable vs. unmaskable
- Resume vs. terminate
  - Easier to implement exceptions that terminate program execution

Stopping & Restarting Execution

- Some exceptions require restart of instruction
  - e.g. Page fault in MEM stage
- When exception occurs, pipeline control can:
  - Force a trap instruction into next IF stage
  - Until the trap is taken, turn off all writes for the faulting (and later) instructions
  - OS exception-handling routine saves faulting instruction PC

Stopping & Restarting Execution

- Precise exceptions
  - Instructions before the faulting one complete
  - Instructions after it restart
  - As if execution were serial
- Exception handling complex if faulting instruction can change state before exception occurs
- Precise exceptions simplifies OS
- Required for demand paging