Pipelining – Appendix A

5 Steps of MIPS Datapath

Visualizing Pipelining

Figure A.2, Page A-8

Hazards

• Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - Structural hazards: HW cannot support this combination of instructions (single person to fold and put clothes away)
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline (missing sock)
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).
One Memory Port/Structural Hazards

Speed Up Equation for Pipelining

Example: Dual-port vs. Single-port
Three Generic Data Hazards

- **Read After Write (RAW)**
  - Instr I, tries to read operand before Instr J, writes it
  - \(I): \text{add } r1, r2, r3\)
  - \(J): \text{sub } r4, r1, r3\)
- Caused by a “Dependence” (in compiler nomenclature). This hazard results from an actual need for communication.
No Hazard here

Three Generic Data Hazards

- **Write After Read (WAR)**
  - Instruction $J$ writes operand before Instruction $I$ reads it.
  - Called an “anti-dependence” by compiler writers. 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

Three Generic Data Hazards

- **Write After Write (WAW)**
  - Instruction $J$ writes operand before Instruction $I$ writes it.
  - Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”.
  - Can’t happen in MIPS 5 stage pipeline because:
    - All instructions take 5 stages, and
    - Writes are always in stage 5
  - Will see WAR and WAW in more complicated pipes

Coping with RAW Hazards: Forwarding

- Addresses RAW data hazard
- The idea:
  - When a RAW hazard is present, instead of waiting until the first instruction actually writes the new value to the register...
  - ...take the updated value directly when it is available in the first instruction’s execution (and inject it as one of the inputs to the later instruction’s EX stage)
- Obviously, this requires modifying the hardware design
  - Add control logic to detect the hazard
  - Datapath to forward the updated value to earlier stages
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

HW Change for Forwarding

NextPC
 Registers
Multiplexer
Immediate
ALU
Multiplexer
Data Memory
MEM/WR

Forwarding to Avoid LW-SW Data Hazard

Time (clock cycles)

Instr. Order

add r1, r2, r3
lw r4, 0(r1)
sw r4, 12(r1)
or r8, r6, r9
xor r10, r9, r11
Data Hazard Even with Forwarding

Time (clock cycles)

Instruction Order

lw r1, 0(r2)

sub r4, r1, r6

and r6, r1, r7

or r8, r1, r9

Data Hazard Even with Forwarding

Time (clock cycles)

Instruction Order

lw r1, 0(r2)

sub r4, r1, r6

and r6, r1, r7

or r8, r1, r9

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:

Fast code:

```asm
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
LW Re,e
SW d,Rd
```

```asm
LW Rb,b
LW Rc,c
ADD Ra,Rb,Rc
SW a,Ra
```

Compiler optimizes for performance. Hardware checks for safety.

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

What do you do with the 3 instructions in between?

If CPI = 1, 30% branch,
Stall 3 cycles => new CPI = 1.3!
Goals of a branch stall reduction strategy

Determine if the branch is taken sooner, AND

Compute the “taken” branch address earlier

One approach

• We could change the ISA to substitute beqz (branch if reg=0) and bnez (branch if reg <> 0) for the beq and bne instructions.
• Then:
  – Move Zero test to ID/RF stage
  – Adder to calculate new PC in ID/RF stage
  – 1 clock cycle penalty for branch versus 3

For general Branches: 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 actually 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

Pipelined MIPS Datapath

Figure A.24, page A-38

• Interplay of instruction set design and cycle time.
Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER a following instruction
- branch instruction sequential successor sequential successor sequential successor branch target if taken
- Introduce a delay by changing the program to insert n instructions that would:
  (a) have to be executed whether or not the branch is taken
  (b) would not produce an incorrect result
- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Delayed Branch

• Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
• Delayed Branch downside: As processors go to deeper pipelines and multiple issue, the branch delay grows and needs more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Scheduling Branch Delay Slots (Fig A.14)
A. From before branch
B. From branch target
C. From fall through

Delayed Branch

• Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
• Delayed Branch downside: As processors go to deeper pipelines and multiple issue, the branch delay grows and needs more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

Evaluating Branch Alternatives

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

Assume 4% unconditional branch, 6% conditional branch-untaken, 10% conditional branch-taken

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI speedup v. unpipelined</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.60</td>
<td>3.1</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
</tr>
</tbody>
</table>

Pipeline depth = 1 + Branch frequency \times Branch penalty