Pipelining

Dr. Soner Onder
CS 4431
Michigan Technological University
A "Typical" RISC ISA

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  - base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3
Example: MIPS (- MIPS)

Register-Register

31 26 25 2120 16 15 1110 6 5 0

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2</th>
<th>Rd</th>
<th>Opx</th>
</tr>
</thead>
</table>

Register-Immediate

31 26 25 2120 16 15 0

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rd</th>
<th>immediate</th>
</tr>
</thead>
</table>

Branch

31 26 25 2120 16 15 0

<table>
<thead>
<tr>
<th>Op</th>
<th>Rs1</th>
<th>Rs2/Opx</th>
<th>immediate</th>
</tr>
</thead>
</table>

Jump / Call

31 26 25 0

<table>
<thead>
<tr>
<th>Op</th>
<th>target</th>
</tr>
</thead>
</table>
Datapath vs Control

Datapath: Storage, FU, interconnect sufficient to perform the desired functions
- Inputs are Control Points
- Outputs are signals

Controller: State machine to orchestrate operation on the data path
- Based on desired function and signals
Approaching an ISA

- **Instruction Set Architecture**
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL on *architected registers* and memory
- Given technology constraints assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function units to do all the required operations
  - Possible additional storage (e.g., MAR, MBR, …)
  - Interconnect to move information among regs and FUs
- Map each instruction to sequence of RTLs
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller
Pipelining: Its Natural!

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes
Sequential Laundry

Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?
Pipelined Laundry
Start work ASAP

- Pipelined laundry takes 3.5 hours for 4 loads
Pipelining Lessons

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup

![Diagram showing pipeline stages and task orders with times and durations]
5 Steps of MIPS Datapath

Figure A.2, Page A-8

Instruction Fetch

Instr. Decode Reg. Fetch

Execute Addr. Calc

Memory Access

Write Back

Next PC

Address

Memory

Adder

4

Next SEQ PC

Instr. Reg File

RD

RS1

RS2

Imm

Sign Extend

Reg File

ALU

Zero?

Data Memory

LMD

MUX

MUX

MUX

MUX

MUX

Next PC

IR $\leftarrow$ mem[PC];

PC $\leftarrow$ PC + 4

Reg[IR_{rd}] $\leftarrow$ Reg[IR_{rs}] \text{ op}_{IRop} \text{ Reg[IR_{rt}]}
5 Steps of MIPS Datapath

Figure A.3, Page A-9

IR <= mem[PC];
PC <= PC + 4

A <= Reg[IRrs];
B <= Reg[IRrt]

rslt <= A opIRop B

WB <= rslt

Reg[IRrd] <= WB
Inst. Set Processor Controller

IR <= mem[PC];
PC <= PC + 4

A <= Reg[IR$_{rs}$];
B <= Reg[IR$_{rt}$]

Ifetch

opFetch-DCD

PC <= IR$_{jaddr}$

if bop(A,b)
PC <= PC+IR$_{im}$

br

PC <= IR$_{jaddr}$

r <= A op$_{IR_{op}}$ B

RR

r <= A op$_{IR_{op}}$ IR$_{im}$

RI

r <= A + IR$_{im}$

JR

jmp

PC <= PC+IR$_{im}$

LD

ST

if bop(A,b)

PC <= PC+IR$_{im}$

br

PC <= IR$_{jaddr}$

St

PC <= PC+IR$_{im}$

LD

ST

WB <= r

Reg[IR$_{rd}$] <= WB

WB <= Mem[r]

Reg[IR$_{rd}$] <= WB

Reg[IR$_{rd}$] <= WB

9/23/2009
5 Steps of MIPS Datapath

- Data stationary control
  - local decode for each instruction phase / pipeline stage
Visualizing Pipelining

Figure A.2, Page A-8
Pipelining is not quite that easy!

- 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

Figure A.4, Page A-14
How do you “bubble” the pipe?
**Speed Up Equation for Pipelining**

\[
\text{CPI}_{\text{pipelined}} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}
\]

\[
\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

For simple RISC pipeline, CPI = 1:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory (“Harvard Architecture”)
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed

\[
\text{SpeedUp}_A = \frac{\text{Pipeline Depth}}{1 + 0} \times \left(\frac{\text{clock}_\text{unpipe}}{\text{clock}_\text{pipe}}\right) = \text{Pipeline Depth}
\]

\[
\text{SpeedUp}_B = \frac{\text{Pipeline Depth}}{1 + 0.4 \times 1} \times \left(\frac{\text{clock}_\text{unpipe}}{\text{clock}_\text{unpipe} / 1.05}\right) = \left(\frac{\text{Pipeline Depth}}{1.4}\right) \times 1.05 = 0.75 \times \text{Pipeline Depth}
\]

\[
\frac{\text{SpeedUp}_A}{\text{SpeedUp}_B} = \frac{\text{Pipeline Depth}}{(0.75 \times \text{Pipeline Depth})} = 1.33
\]

- Machine A is 1.33 times faster
Data Hazard on R1

Figure A.6, Page A-17

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
Three Generic Data Hazards

- **Read After Write (RAW)**
  Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it

  \[
  \begin{align*}
  I & : \text{add } r1, r2, r3 \\
  J & : \text{sub } r4, r1, r3
  \end{align*}
  \]

- **Caused by a “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

  ![Instruction Sequence Diagram]

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

  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)**
  Instr\textsubscript{J} writes operand before Instr\textsubscript{I} writes it.

  \[ \text{I: sub } r1, r4, r3 \]
  \[ \text{J: add } r1, r2, r3 \]
  \[ \text{K: mul } r6, r1, r7 \]

- 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
Forwarding to Avoid Data Hazard

Figure A.7, Page A-19

Intr. Order

**Time (clock cycles)**

add \(r_1, r_2, r_3\)

sub \(r_4, r_1, r_3\)

and \(r_6, r_1, r_7\)

or \(r_8, r_1, r_9\)

xor \(r_{10}, r_1, r_{11}\)
What circuit detects and resolves this hazard?
Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20

**Instr. Order**

- add \( r1, r2, r3 \)
- lw \( r4, 0(r1) \)
- sw \( r4, 12(r1) \)
- or \( r8, r6, r9 \)
- xor \( r10, r9, r11 \)

**Time (clock cycles)**
Data Hazard Even with Forwarding

Figure A.9, Page A-21

Time (clock cycles)

lw r1, 0(r2)
sub r4, r1, r6
and r6, r1, r7
or r8, r1, r9
Data Hazard Even with Forwarding
(Similar to Figure A.10, Page A-21)

How is this detected?
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

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?
How do you do it?
Where is the “commit”?
Branch Stall Impact

- If CPI = 1, 30% branch, Stall 3 cycles => new CPI = 1.9!
- 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
Pipelined MIPS Datapath

Figure A.24, page A-38

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

#4: Delayed Branch

- Define branch to take place **AFTER** a following instruction

```
branch instruction
  sequential successor\_1
  sequential successor\_2
  ........
  sequential successor\_n
branch target if taken
```

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this
Scheduling Branch Delay Slots (Fig A.14)

A. From before branch

- add $1, $2, $3
- if $2=0 then
- delay slot
- if $2=0 then
- add $1, $2, $3

B. From branch target

- sub $4, $5, $6
- add $1, $2, $3
- if $1=0 then
- delay slot
- if $1=0 then
- add $1, $2, $3
- sub $4, $5, $6

C. From fall through

- add $1, $2, $3
- if $1=0 then
- delay slot
- if $1=0 then
- sub $4, $5, $6

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the sub instruction may need to be copied, increasing IC
- In B and C, must be okay to execute sub when branch fails
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 processor go to deeper pipelines and multiple issue, the branch delay grows and need 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</th>
<th>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>
<td>1.0</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.20</td>
<td>4.2</td>
<td>1.33</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.40</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.10</td>
<td>4.5</td>
<td><strong>1.45</strong></td>
</tr>
</tbody>
</table>
Problems with Pipelining

- **Exception**: An unusual event happens to an instruction during its execution
  - Examples: divide by zero, undefined opcode
- **Interrupt**: Hardware signal to switch the processor to a new instruction stream
  - Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting)
- **Problem**: It must appear that the exception or interrupt must appear between 2 instructions ($I_i$ and $I_{i+1}$)
  - The effect of all instructions up to and including $I_i$ is totally complete
  - No effect of any instruction after $I_i$ can take place
- The interrupt (exception) handler either aborts program or restarts at instruction $I_{i+1}$
Precise Exceptions in Static Pipelines

Key observation: architected state only change in memory and register write stages.
And In Conclusion: Control and Pipelining

- Control VIA State Machines and Microprogramming
- Just overlap tasks; easy if tasks are independent
- Speed Up ≤ Pipeline Depth; if ideal CPI is 1, then:

\[
\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}
\]

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Exceptions, Interrupts add complexity
Handling multi-cycle operations

- How would the pipeline should be changed if some instructions need more than a single cycle to complete their execution?
- What are the consequences in terms of hazards?