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

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>6</th>
<th>5</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Opx</td>
</tr>
</tbody>
</table>

Register-Immediate

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>Rs1</td>
<td>Rs2/Opx</td>
<td></td>
<td></td>
<td>immediate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Jump / Call

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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 (eg. 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

9/28/2020
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
5 Steps of MIPS Datapath

Instruction Fetch

取指

IR \leftarrow \text{mem}[PC];

PC \leftarrow PC + 4

Instr. Decode Reg. Fetch

解码

Execute Addr. Calc

执行

Memory Access

访存

Write Back

写回

Reg[IR_{rd}] \leftarrow Reg[IR_{rs}] \text{ op}_{IR_{fp}} Reg[IR_{rt}]
5 Steps of MIPS Datapath
Figure A.3, Page A-9

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

A <= Reg[IR_{rs}];
B <= Reg[IR_{rt}]

rslt <= A op_{IRop} B

WB <= rslt

Reg[IR_{rd}] <= WB
Inst. Set Processor Controller

IR \leftarrow \text{mem}[PC];
PC \leftarrow PC + 4

A \leftarrow \text{Reg}[IR_{rs}];
B \leftarrow \text{Reg}[IR_{rt}]

r \leftarrow A \text{ op}_{IR_{op}} B

WB \leftarrow r

Reg[IR_{rd}] \leftarrow WB

Irfetch

if bop(A,b)
PC \leftarrow \text{IR}_{jaddr}

PC \leftarrow PC + IR_{im}
br jmp

br

if bop(A,b)
PC \leftarrow IR_{jaddr}

PC \leftarrow PC + IR_{im}

jr

jmp

opFetch-DCD

r \leftarrow A \text{ op}_{IR_{op}} IR_{im}

r \leftarrow A + IR_{im}

rr

ri

LD

ST

r \leftarrow A + IR_{im}

WB \leftarrow \text{Mem}[r]

Reg[IR_{rd}] \leftarrow WB

Reg[IR_{rd}] \leftarrow WB

Reg[IR_{rd}] \leftarrow WB

JSR

JR
5 Steps of MIPS Datapath

Figure A.3, Page A-9

- **Data stationary control**
  - local decode for each instruction phase / pipeline stage
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
(Similar to Figure A.5, Page A-15)

How do you “bubble” the pipe?
Data Hazard on R1

Figure A.6, Page A-17

Time (clock cycles)

add r1, r2, r3

sub r4, r1, r3

and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11
Dependences and hazards

- Dependences are a program property:
  - If two instructions are data dependent they cannot execute simultaneously.
  - Existence of control-dependences means serialization.
  - Whether a dependence results in a hazard and whether that hazard actually causes a stall are properties of the pipeline organization.
  - Data dependences may occur through registers or memory.
Dependences and hazards

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the length of any stall is a property of the pipeline. A data dependence:

- Indicates that there is a possibility of a hazard.
- Determines the order in which results must be calculated, and
- Sets an upper bound on the amount of parallelism that can be exploited.
Dependencies

- Name dependencies
- Output dependence
- Anti-dependence
- Data
  - True dependence
- Control
Data dependences

- Data dependence, true dependence, and true data dependence are terms used to mean the same thing:

- An instruction j is data dependent on instruction i if either of the following holds:
  - instruction i produces a result that may be used by instruction j, or
  - instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

- Chains of dependent instructions.
Name dependences

- Output dependence:
  - When instruction I and j write the same register or memory location. The ordering must be preserved to leave the correct value in the register:
    - add r7,r4,r3
    - div r7,r2,r8

- Antidependence:
  - When instruction j writes a register or memory location that instruction i reads:
    - i: add r6,r5,r4
    - j: sub r5,r8,r11
Data Dependences through registers/memory

- Dependences through registers are easy:
  - `lw r10,10(r11)`
  - `add r12,r10,r8`
  - just compare register names.

- Dependences through memory are harder:
  - `sw r10,4 (r2)`
  - `lw r6,0(r4)`
  - `is r2+4 = r4+0 ? If so they are dependent, if not, they are not.`
Control dependences

- An instruction $j$ is control dependent on $i$ if the execution of $j$ is controlled by instruction $i$.
  - I: If $a < b$
    - j: $a = a + 1$; $j$ is control dependent on I.

- 1. An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.

- 2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.
Three Generic Data Hazards

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

  \[ \text{I: add } r_1, r_2, r_3 \]
  \[ \text{J: sub } r_4, r_1, r_3 \]

- Caused by a true dependence in the program.
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

- Caused by an “anti-dependence” in the program. 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\_j writes operand *before* Instr\_i writes it.

  ![Diagram](I: sub r1,r4,r3
  J: add r1,r2,r3
  K: mul r6,r1,r7)

- Caused by an “output dependence” in the program. 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

Time (clock cycles)

Instruction Order

add r1, r2, r3

sub r4, r1, r3

and r6, r1, r7

or r8, r1, r9

xor r10, r1, r11
What circuit detects and resolves this hazard?
Forwarding to Avoid LW-SW Data Hazard

Figure A.8, Page A-20

```
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

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?
Software Scheduling to Avoid Load Hazards

Try producing fast code for
\[
\begin{align*}
a &= b + c; \\
d &= e - f;
\end{align*}
\]
assuming \(a, b, c, d, e, \) and \(f\) in memory.

Slow code:

\[
\begin{align*}
\text{LW} & \quad Rb,b \\
\text{LW} & \quad Rc,c \\
\text{ADD} & \quad Ra,Rb,Rc \\
\text{SW} & \quad a,Ra \\
\text{LW} & \quad Re,e \\
\text{LW} & \quad Rf,f \\
\text{SUB} & \quad Rd,Re,Rf \\
\text{SW} & \quad d,Rd \\
\end{align*}
\]

Fast code:

\[
\begin{align*}
\text{LW} & \quad Rb,b \\
\text{LW} & \quad Rc,c \\
\text{LW} & \quad Re,e \\
\text{ADD} & \quad Ra,Rb,Rc \\
\text{LW} & \quad Rf,f \\
\text{SW} & \quad a,Ra \\
\text{SUB} & \quad Rd,Re,Rf \\
\text{SW} & \quad d,Rd \\
\end{align*}
\]

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
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 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

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

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>
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 \( \leq \) 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?
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

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

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

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

- Machine A is 1.33 times faster