CS152 – Computer Architecture and Engineering

Fall 2004

Lecture 10: Basic MIPS Pipelining Review

John Lazzaro
(www.cs.berkeley.edu/~lazzaro)

Dave Patterson
(www.cs.berkeley.edu/~patterson)

[Adapted from Mary Jane Irwin’s slides www.cse.psu.edu/~cg431 ]
Recap last lecture

Customers: measure to buy
Architects: measure for design

Architects: measure for design

Tools: Performance Equation, CPI

Seconds Program = Instructions Program

Cycles Instruction

Seconds Cycle

Amdahl’s Law’s lesson: Balance

Speedup whole = \frac{1}{1 - (\% \text{ affected/Speedup}_{\text{part}})}

Energy:

E_{0->1} = \frac{1}{2} C V_{dd}^2
E_{1->0} = \frac{1}{2} C V_{dd}^2
The Five Stages of Load Instruction

- **Cycle 1**: IFetch: Instruction Fetch and Update PC
- **Cycle 2**: Dec: Registers Fetch and Instruction Decode
- **Cycle 3**: Exec: Execute R-type; calculate memory address
- **Cycle 4**: Mem: Read/write the data from/to the Data Memory
- **Cycle 5**: WB: Write the result data into the register file
Pipelined MIPS Processor

- Start the \textbf{next} instruction while still working on the current one
  - improves \textit{throughput} or \textit{bandwidth} - total amount of work done in a given time (average instructions per second or per clock)
  - instruction \textit{latency} is not reduced (time from the start of an instruction to its completion)

- pipeline clock cycle (pipeline stage time) is limited by the slowest stage
- for some instructions, some stages are \textit{wasted} cycles
Single Cycle, Multiple Cycle, vs. Pipeline

Single Cycle Implementation:

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>Clk</td>
</tr>
<tr>
<td>Load</td>
<td>Store</td>
</tr>
<tr>
<td>Waste</td>
<td></td>
</tr>
</tbody>
</table>

Multiple Cycle Implementation:

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
<th>Cycle 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
<td>Clk</td>
</tr>
<tr>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
<td>lw</td>
</tr>
<tr>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>IFetch</td>
</tr>
<tr>
<td>sw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-type</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Implementation:

<table>
<thead>
<tr>
<th>lw</th>
<th>IFetch</th>
<th>Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>R-type</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
</tbody>
</table>

“wasted” cycles
Multiple Cycle v. Pipeline, Bandwidth v. Latency

Multiple Cycle Implementation:

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
<th>Cycle 7</th>
<th>Cycle 8</th>
<th>Cycle 9</th>
<th>Cycle 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
</tr>
<tr>
<td>lw</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pipeline Implementation:

<table>
<thead>
<tr>
<th>lw</th>
<th>IFetch</th>
<th>Dec</th>
<th>Exec</th>
<th>Mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
<tr>
<td>R-type</td>
<td>IFetch</td>
<td>Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>WB</td>
</tr>
</tbody>
</table>

- **Latency** per lw = 5 clock cycles for both
- **Bandwidth** of lw is 1 per clock clock (IPC) for pipeline vs. 1/5 IPC for multicycle
- Pipelining improves instruction bandwidth, not instruction latency
Pipelining the MIPS ISA

What makes it easy

- all instructions are the same length (32 bits)
  - easier to fetch in 1st stage and decode in 2nd stage
- few instruction formats (three) with symmetry across formats
  - can begin reading register file in 2nd stage
- memory operations can occur only in loads and stores
  - can use the execute stage to calculate memory addresses
- each MIPS instruction writes at most one result and does so near the end of the pipeline

What makes it hard

- structural hazards: what if we had only one memory?
- control hazards: what about branches?
- data hazards: what if an instruction’s input operands depend on the output of a previous instruction?
MIPS Pipeline Datapath Modifications

- What do we need to add/modify in our MIPS datapath?
  - registers between pipeline stages to isolate them

![Datapath Diagram]

System Clock

IF: IFetch
ID: Dec
EX: Execute
MEM: MemAccess
WB: WriteBack

Instruction Memory

Data Memory

Register File

ALU

Memory

Register File

ALU

Address

Write Data

Write Data

Write Data

Write Back

System Clock
Graphically Representing MIPS Pipeline

- Can help with answering questions like:
  - how many cycles does it take to execute this code?
  - what is the ALU doing during cycle 4?
  - is there a hazard, why does it occur, and how can it be fixed?
Why Pipeline? For Throughput!

Once the pipeline is full, one instruction is completed every cycle.

Time to fill the pipeline
Administrivia

- Lab 2 demo Friday, due Monday
  - Feedback on team effort
  - How did it work? Change before pipeline?

- Reading Chapter 6, sections 6.1 to 6.4 for today, 6.5 to 6.9 for next 2 lectures

- Midterm Tue Oct 12 5:30 - 8:30 in **101 Morgan**
  (you asked for it)
  - Northwest corner of campus, near Arch and Hearst
  - Midterm review Sunday Oct 10, 7 PM, 306 Soda
  - Bring 1 page, handwritten notes, both sides
  - Nothing electronic: no calculators, cell phones, pagers, …
  - Meet at LaVal’s Northside afterwards for Pizza
Important Observation

- Each functional unit can only be used once per instruction (since 4 other instructions executing)

- If each functional unit used at different stages then leads to hazards:
  - Load uses Register File’s Write Port during its 5th stage
  - R-type uses Register File’s Write Port during its 4th stage

  2 ways to solve this pipeline hazard.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td>Mem</td>
<td>Wr</td>
</tr>
<tr>
<td>R-type</td>
<td>Ifetch</td>
<td>Reg/Dec</td>
<td>Exec</td>
<td></td>
<td>Wr</td>
</tr>
</tbody>
</table>
Solution 1: Insert “Bubble” into the Pipeline

- Insert a “bubble” into the pipeline to prevent 2 writes at the same cycle
  - The control logic can be complex.
  - Lose instruction fetch and issue opportunity.

- No instruction is started in Cycle 6!
Solution 2: Delay R-type’s Write by One Cycle

- Delay R-type’s register write by one cycle:
  - Now R-type instructions also use Reg File’s write port at Stage 5
  - Mem stage is a NOP stage: nothing is being done.

![Pipeline Diagram]

Clock

Cycle 1 | Cycle 2 | Cycle 3 | Cycle 4 | Cycle 5 | Cycle 6 | Cycle 7 | Cycle 8 | Cycle 9
---|---|---|---|---|---|---|---|---
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type
Ifetch | Reg/Dec | Exec | Mem | R-type

Cal

CS 152 L10 Pipeline Intro (14)  Fall 2004 © UC Regents
Can Pipelining Get Us Into Trouble?

- **Yes:** Pipeline Hazards
  - **structural hazards:** attempt to use the same resource by two different instructions at the same time
  - **data hazards:** attempt to use data before it is ready
    - instruction source operands are produced by a prior instruction still in the pipeline
    - load instruction followed immediately by an ALU instruction that uses the load operand as a source value
  - **control hazards:** attempt to make a decision before condition has been evaluated
    - branch instructions

- Can always resolve hazards by **waiting**
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards
A Single Memory Would Be a Structural Hazard

Time (clock cycles)

Reading data from memory

Reading instruction from memory
How About Register File Access?

Potential read before write data hazard
How About Register File Access?

Can fix register file access hazard by doing reads in the second half of the cycle and writes in the first half.

Potential read before write data hazard
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

Which are read before write data hazards?
Register Usage Can Cause Data Hazards

- Dependencies backward in time cause hazards

- Add r1, r2, r3
- Subtract r4, r1, r5
- AND r6, r1, r7
- OR r8, r1, r9
- XOR r4, r1, r5

Read before write data hazards
Loads Can Cause Data Hazards

- Dependencies backward in time cause hazards

- **Load-use data hazard**

- `lw r1,100(r2)`
- `sub r4,r1,r5`
- `and r6,r1,r7`
- `or r8, r1, r9`
- `xor r4,r1,r5`
One Way to “Fix” a Data Hazard

add r1, r2, r3

Can fix data hazard by waiting – stall – but affects throughput

sub r4, r1, r5

and r6, r1, r7
Another Way to “Fix” a Data Hazard

Can fix data hazard by forwarding results as soon as they are available to where they are needed.

Instruction Order:

- add r1, r2, r3
- sub r4, r1, r5
- and r6, r1, r7
- or r8, r1, r9
- xor r4, r1, r5
Another Way to “Fix” a Data Hazard

Can fix data hazard by forwarding results as soon as they are available to where they are needed.

add r1, r2, r3
sub r4, r1, r5
and r6, r1, r7
or r8, r1, r9
xor r4, r1, r5
Forwarding with Load-use Data Hazards

lw r1,100(r2)
sub r4,r1,r5
and r6,r1,r7
or r8, r1, r9
xor r4,r1,r5

Will still need one stall cycle even with forwarding
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards

Diagram:

- beq
- lw
- Inst 3
- Inst 4
One Way to “Fix” a Control Hazard

Can fix branch hazard by waiting – stall – but affects throughput.

beq
stall
stall
stall
lw
Inst 3
Corrected Datapath to Save RegWrite Addr

- Need to preserve the destination register address in the pipeline state registers
Corrected Datapath to Save RegWrite Addr

- Need to preserve the destination register address in the pipeline state registers (Bug in COD 1st edition!)

![Diagram of corrected datapath to save RegWrite Addr](image)
MIPS Pipeline Control Path Modifications

- All control signals can be determined during Decode
  - and held in the state registers between pipeline stages
Control Settings

<table>
<thead>
<tr>
<th>EX Stage</th>
<th>MEM Stage</th>
<th>WB Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg Dst</td>
<td>ALU Op1</td>
<td>ALU Op0</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

Q: Why not show write enable for pipeline registers?
A: Written every clock cycle (like PC)

Q: Why not show control for IF and ID stages?
A: Control same for all instructions in IF and ID stages: fetch instruction, increment PC
Other Pipeline Structures Are Possible

- What about (slow) multiply operation?
  - let it take two cycles

- What if the data memory access is twice as slow as the instruction memory?
  - make the clock twice as slow or …
  - let data memory access take two cycles (and keep the same clock rate)
Sample Pipeline Alternatives (for ARM ISA)

- **ARM7**
  - (3-stage pipeline)
  - 3 stages: IM → Reg → EX
  - PC update, IM access, decode reg access, ALU op, DM access, shift/rotate, commit result (write back)

- **StrongARM-1**
  - (5-stage pipeline)
  - 5 stages: IM → Reg → ALU → DM → Reg

- **XScale**
  - (7-stage pipeline)
  - 7 stages: IM1 → IM2 → Reg → SHFT → ALU → DM1 → DM2
  - PC update, BTB access, start IM access, decode reg 1 access, ALU op, DM write reg write, IM access, shift/rotate reg 2 access, start DM access exception
Suppose a big data cache results in a data cache latency of 2 clock cycles and a 6-stage pipeline. (Pipelined, so can do 1 access / clock cycle.) What is the impact?

1. Instruction bandwidth is now 5/6-ths of the 5-stage pipeline
2. Instruction bandwidth is now 1/2 of the 5-stage pipeline
3. The branch delay slot is now 2 instructions
4. The load-use hazard can be with 2 instructions following load
5. Both 3 and 4: branch delay and load-use now 2 instructions
6. None of the above
Suppose a big data cache results in a data cache latency of 2 clock cycles and a 6-stage pipeline. (Pipelined, so can do 1 access / clock cycle.) What is the impact?

1. Instruction bandwidth is now 5/6-ths of the 5-stage pipeline
2. Instruction bandwidth is now 1/2 of the 5-stage pipeline
3. The branch delay slot is now 2 instructions
4. The load-use hazard can be with 2 instructions following load
5. Both 3 and 4: branch delay and load-use now 2 instructions
6. None of the above
Suppose a big I cache results in an I cache latency of 2 clock cycles and a 6-stage pipeline. (Pipelined, so can do 1 access / clock cycle.) What is the impact?

1. Instruction bandwidth is now 5/6-ths of the 5-stage pipeline
2. Instruction bandwidth is now 1/2 of the 5-stage pipeline
3. The branch delay slot is now 2 instructions
4. The load-use hazard can be with 2 instructions following load
5. Both 3 and 4: branch delay and load-use now 2 instructions
6. None of the above
Suppose a big I cache results in an I cache latency of 2 clock cycles and a 6-stage pipeline. (Pipelined, so can do 1 access / clock cycle.) What is the impact?

1. Instruction bandwidth is now 5/6-ths of the 5-stage pipeline
2. Instruction bandwidth is now 1/2 of the 5-stage pipeline
3. The branch delay slot is now 2 instructions
4. The load-use hazard can be with 2 instructions following load
5. Both 3 and 4: branch delay and load-use now 2 instructions
6. None of the above
Suppose we use with a 4 stage pipeline that combines memory access and write back stages for all instructions but load, stalling when there are structural hazards. Impact?

1. The branch delay slot is now 0 instructions
2. Most loads cause stall since often a structural hazard on reg. writes
3. Most stores cause stall since they have a structural hazard
4. Both 2 & 3: most loads&stores cause stall due to structural hazards
5. Most loads cause stall, but there is no load-use hazard anymore
6. Both 2 & 3, but there is no load-use hazard anymore
7. None of the above
Suppose we use with a 4 stage pipeline that combines memory access and write back stages for all instructions but load, stalling when there are structural hazards. Impact?

1. The branch delay slot is now 0 instructions

2. Most loads cause stall since often a structural hazard on reg. writes

3. Most stores cause stall since they have a structural hazard

4. Both 2 & 3: most loads & stores cause stall due to structural hazards

5. Most loads cause stall, but there is no load-use hazard anymore

6. Both 2 & 3, but there is no load-use hazard anymore

7. None of the above

Q: Why not say every load stalls?

A: Not all next instructions write in Wr stage
Designing a Pipelined Processor

- Go back and examine your data path and control diagram
- Associate resources with states
  - Be sure there are no structural hazards: one use / clock cycle
- Add pipeline registers between stages to balance clock cycle
  - Amdahl’s Law suggests splitting longest stage
- Resolve all data and control dependencies
  - If backwards in time in pipeline drawing to registers
    => data hazard: forward or stall to resolve them
  - If backwards in time in pipeline drawing to PC
    => control hazard: we’ll see next time
- Assert control in appropriate stage
- Develop test instruction sequences likely to uncover pipeline bugs
  - If you don’t test it, it won’t work
Brain storm on bugs (if time permits)

- Where are bugs likely to hide in a pipelined processor?
  1.
  2.
  ...

- How can you write tests to uncover these likely bugs?
  1.
  2.
  ...

- Once it passes a test, never need to run it again in the design process?
Summary

- All modern day processors use pipelining

- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
  - Multiple tasks operating simultaneously using different resources

- Potential speedup = Number of pipe stages

- Pipeline rate limited by slowest pipeline stage
  - Unbalanced lengths of pipe stages reduces speedup
  - Time to “fill” pipeline and time to “drain” it reduces speedup

- Must detect and resolve hazards
  - Stalling negatively affects throughput

- Next time: pipeline control, including hazards