Today's class

• Pipelining
  – Speed up
  – How it is done
  – Hazards
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?

![Diagram showing sequential laundry process and time taken for each load]
• Pipelined laundry takes 3.5 hours for 4 loads
Key Definitions

*Pipelining* is a key implementation technique used to build fast processors. It allows the execution of multiple instructions to overlap in time.

A pipeline within a processor is similar to a car assembly line. Each assembly station is called a *pipe stage* or a *pipe segment*.

The *throughput* of an instruction pipeline is the measure of how often an instruction exits the pipeline.
Speed up

• Speedup for 4 loads
  – \[ \frac{\text{Old\_execution\_time}}{\text{New\_execution\_time}} = 4 \frac{(\text{wash\_time} + \text{dry\_time} + \text{fold\_time})}{(\text{wash\_time} + 4 \text{dry\_time} + \text{fold\_time})} \]

• Speedup for N loads
  – \[ N \frac{(\text{wash\_time} + \text{dry\_time} + \text{fold\_time})}{(\text{wash\_time} + N \text{dry\_time} + \text{fold\_time})} \]
  \[ \Rightarrow \max\_\text{speedup} = \frac{(\text{wash\_time} + \text{dry\_time} + \text{fold\_time})}{\text{dry\_time}} \]
  \[ \Rightarrow \max\_\text{speedup} = \frac{\text{execution\_time\_of\_one}}{\text{slowest\_stage}} \]
Pipeline Stages

We can divide the execution of an instruction into the following 5 “classic” stages:

- **IF**: Instruction Fetch
- **ID**: Instruction Decode, register fetch
- **EX**: Execution
- **MEM**: Memory Access
- **WB**: Register write Back
Pipeline Throughput and Latency

Consider the pipeline above with the indicated delays. We want to know what is the *pipeline throughput* and the *pipeline latency*.

**Pipeline throughput**: instructions completed per second.

**Pipeline latency**: how long does it take to execute a single instruction in the pipeline.
Pipeline Throughput and Latency

Pipeline throughput: how often an instruction is completed.

\[
\text{Throughput} = \frac{1 \text{ instr}}{\max \{ \text{lat(IF)}, \text{lat(ID)}, \text{lat(EX)}, \text{lat(MEM)}, \text{lat(WB)} \}}
\]

\[
= \frac{1 \text{ instr}}{\max \{5 \text{ ns}, 4 \text{ ns}, 5 \text{ ns}, 10 \text{ ns}, 4 \text{ ns} \}}
\]

\[
= \frac{1 \text{ instr}}{10 \text{ ns}} \quad \text{(ignoring pipeline register overhead)}
\]

Pipeline latency: how long does it take to execute an instruction in the pipeline.

\[
L = \text{lat(IF)} + \text{lat(ID)} + \text{lat(EX)} + \text{lat(MEM)} + \text{lat(WB)}
\]

\[
= 5 \text{ ns} + 4 \text{ ns} + 5 \text{ ns} + 10 \text{ ns} + 4 \text{ ns} = 28 \text{ ns}
\]
Pipeline Throughput and Latency

Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction.

We are in trouble! The latency is not constant. This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.
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
Other Definitions

• Pipe stage or pipe segment
  – A decomposable unit of the fetch-decode-execute paradigm

• Pipeline depth
  – Number of stages in a pipeline

• Machine cycle
  – Clock cycle time

• Latch
  – Per phase/stage local information storage unit
Design Issues

• Balance the length of each pipeline stage

\[
\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}
\]

• Problems
  – Usually, stages are not balanced
  – Pipelining overhead
  – Hazards (conflicts)

• Performance (throughput \rightarrow CPU performance equation)
  – Decrease of the CPI
  – Decrease of cycle time
### MIPS Instruction Formats

#### I Format

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs\textsubscript{1}</th>
<th>rd</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>16</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

#### R Format

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>rs\textsubscript{1}</th>
<th>rs\textsubscript{2}</th>
<th>rd</th>
<th>Shamt/function</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
<td>10</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>20</td>
<td>21</td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

#### J Format

<table>
<thead>
<tr>
<th></th>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

*Fixed-field decoding*
1st and 2nd Instruction cycles

• Instruction fetch (IF)
  \[ \text{IR} \leftarrow \text{Mem}[\text{PC}]; \]
  \[ \text{NPC} \leftarrow \text{PC} + 4 \]

• Instruction decode & register fetch (ID)
  \[ \text{A} \leftarrow \text{Regs}[\text{IR}_{6..10}]; \]
  \[ \text{B} \leftarrow \text{Regs}[\text{IR}_{11..15}]; \]
  \[ \text{Imm} \leftarrow ((\text{IR}_{16})^{16} \# \# \text{IR}_{16..31}) \]
3rd Instruction cycle

- Execution & effective address (EX)
  - Memory reference
    - ALUOutput $\leftarrow A + \text{Imm}$
  - Register - Register ALU instruction
    - ALUOutput $\leftarrow A \ func \ B$
  - Register - Immediate ALU instruction
    - ALUOutput $\leftarrow A \ op \ \text{Imm}$
  - Branch
    - ALUOutput $\leftarrow \text{NPC} + \text{Imm}; \ Cond \ (A \ op \ 0)$
4th Instruction cycle

- Memory access & branch completion (MEM)
  - Memory reference
    - PC ← NPC
    - LMD ← Mem[ALUOutput] (load)
    - Mem[ALUOutput] ← B (store)
  - Branch
    - if (cond) PC ← ALUOutput; else PC ← NPC
5th Instruction cycle

• Write-back (WB)
  – Register - register ALU instruction
    • $\text{Regs}[IR_{16..20}] \leftarrow \text{ALUOutput}$
  – Register - immediate ALU instruction
    • $\text{Regs}[IR_{11..15}] \leftarrow \text{ALUOutput}$
  – Load instruction
    • $\text{Regs}[IR_{11..15}] \leftarrow \text{LMD}$
5 Steps of MIPS Datapath

1. Instruction Fetch
2. Instruction Decode
3. Register Fetch
4. Execute Address Calculation
5. Memory Access

Next PC

4

Address

Adder

Memory

Next SEQ PC

RS1

RS2

RD

Imm

Reg File

Sign Extend

Zero?

ALU

Data Memory

LMD

MUX

WB Data
5 Steps of MIPS Datapath

- Data stationary control
  - local decode for each instruction phase / pipeline stage
## Basic Pipeline

<table>
<thead>
<tr>
<th>Instr #</th>
<th>Clock number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$i$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 1$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 2$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 3$</td>
<td>IF</td>
</tr>
<tr>
<td>$i + 4$</td>
<td>IF</td>
</tr>
</tbody>
</table>
Pipelined Datapath

IF/ID
- PC
- Instr. Cache

ID/EX
- Mux
- Regs
- Add Mux
- Sign extend

EX/MEM
- Zero?
- ALU Mux
- Data Cache

MEM/WB
- Mux
Performance limitations

• Imbalance among pipe stages
  – limits cycle time to slowest stage

• Pipelining overhead
  – Pipeline register delay
  – Clock skew

• Clock cycle > clock skew + latch overhead

• Hazards
Data Hazards
Data Hazards

Data hazards occur because of dependencies on operands between instructions

- **DADD R1, R2, R3**
- **DSUB R4, R1, R5**
- **AND R6, R1, R7**
- **OR R8, R1, R9**
- **XOR R10, R1, R11**

R1 is produced by the first instruction and used as an operand in all the remaining instructions but the computed value is not stored in R1 until clock cycle 5.

A data hazard arises if one operand is not yet available when needed by another instruction in the pipeline.

© 2003 Elsevier Science (USA). All rights reserved.
To Resolve Data Hazards

• We prefer not to stall later instructions
  – especially since the value needed is ready earlier in the pipeline just not yet available in R1

• *Forwarding* will allow a stage to send a computed value on to the location(s) in the pipeline waiting for it
  – forwarding is also called bypassing or short-circuiting

• Forwarding is performed by hardware connected to the latches

• Forwarding can occur
  – through the register file between instructions in WB and ID stages
  – between one instruction as it leaves the EX or MEM stage and the next instruction as it enters the EX or MEM stage
Forwarding Hardware

ALU output to ALU input is used if the next instruction is an ALU operation that uses the result of the current ALU operation.

Memory output to ALU input is used if the current instruction is a Load and the next instruction is an ALU operation that uses the loaded value.

Here, we add further paths that allow a value to be forwarded from ALU or memory into the A or B register or into data memory.
Forwarding in our Example

DADD’s result forwarded to one of DSUB’s ALU inputs

DADD’s result forwarded to one of AND’s ALU inputs one cycle later

DADD’s result forwarded to OR through the register file
Example Requiring a Stall

• Now consider the following sequence:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, 0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

– LW instruction computes the effective address in EX stage and fetches the datum in MEM stage
– this value cannot be forwarded until the end of the MEM stage whereas DSUB needs R1 at the beginning of its EX stage
– since the value does not become available until the end of that clock cycle, DSUB cannot proceed into its EX stage

• NOTE: DSUB actually gets a value for R1 from the register file, but it is an old version of R1, no longer valid because of the LW instruction
– the DSUB instruction must stall prior to its EX stage
The stall allows the LW instruction to obtain the datum from memory in its MEM stage and forward it onto DSUB’s EX stage:
- DSUB will have retrieved the old value of R1 in its ID stage and this will be replaced before DSUB proceeds into its EX stage.
- Notice that neither AND nor OR must be stalled because R1 can be forwarded to them as we have seen previously.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1, 0(R2)</td>
<td></td>
<td></td>
<td>stall</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>AND R6, R1, R7</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>OR R8, R1, R9</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>
RAW hazards

• Read after write
  – instruction i begins execution before instruction j but
  – instruction j tries to read a source operand before instruction i writes it
  – so, j gets the old value

• This is the most common form of data hazard
  – including all of the examples we looked at previously

• We overcome this hazard by forwarding or stalling
  – as soon as instruction i has computed value, it is forwarded to the proper
    stage for instruction j to use it

<table>
<thead>
<tr>
<th>DADD R1, R2, R3</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSUB R4, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

DSUB instruction tries to read R1 before it is written by DADD instruction

Notice if our pipeline were longer, forwarding might not be enough to solve our problem
WAW hazards

- **Write after write**
  - instruction i begins execution before instruction j but
  - instruction j tries to write its result to a location before it has been written to by instruction i
    - the writes are performed in the wrong order (recall that they are writing to the same register, so the earlier instruction’s result winds up in the register, not the latter instruction’s result)
    - this is *only possible* in a pipeline where writes may occur in more than one stage or instructions can complete out of order

- **Consider a pipeline in which loads take two memory cycles and ALU operations skip the MEM stage entirely**
  - we could then have a situation where the latter instruction writes before the earlier instruction writes
WAR hazard

• Write after read
  – instruction i begins execution before instruction j but
  – instruction j tries to write its result to a location before instruction i can read it as a source operand
    • the source operand is replaced before it can be read
  – this \textit{cannot happen} in MIPS because all reads are in the ID stage and all writes in the WB stage
  – this hazard could occur in a pipeline where there are multiple stages in which reads and writes can occur (although this is still rare since reads usually occur before writes in a pipeline)
  – consider as an example, autoincrement/decrement addressing and a two-memory load where the register storing the address is not read until the second stage
    • the autoincrementing writes before the load reads

<table>
<thead>
<tr>
<th>SW 0(R1, R2)</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM1</th>
<th>MEM2</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DADD R2, R3, R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recall that register writes happen before register reads in the same cycle
Impact of Stalls in MIPS

• Suppose that 30% of the instructions are loads and half the time the instruction following the load depends on the result of the load
  – that is, 50% of all loads cause a RAW hazard
  – in MIPS, this hazard creates a single-cycle delay

• How much faster is the ideal pipelined machine which does not delay the pipeline?

• Solution:
  – the ideal machine has a CPI of 1 since there are no stalls
  – if an instruction which follows a Load is stalled, its CPI becomes 2, so half of the instructions following Loads have a CPI of 1 and half have a CPI of 2
    • average CPI for an instruction following a load is 1.5
  – with 30% loads, effective CPI = \((.7 \times 1 + .3 \times 1.5) = 1.15\)
    • the ideal machine is 1.15 times faster.
Scheduling for Data Hazards

• RAW data hazards will still be prevalent (whenever a loaded value is needed immediately afterward)

• We can optimize a compiler to alter the order of MIPS instructions to eliminate the stalls caused by this data hazard
  – this techniques is known as pipeline (or instruction) scheduling or *compiler scheduling*
  – this can be accomplished by moving a load further to the beginning of a set of code or an instruction that uses the loaded value further to the end of a set of code

• The simplest pipeline scheduling algorithms try to use other instructions in the same block of code to fill in a stall
  – graphing data dependencies allows the compiler to select neutral instructions (that is, instructions that can be moved without causing a logical error to be introduced)
  – this is easy to perform in the MIPS pipeline because it is a short pipeline and the EX and MEM stages in close proximity
  – this becomes more complicated in longer pipelines
## Scheduling Examples

| Instruction | Registers | IF | ID | EX  | MEM | WB |
|-------------|-----------|----|----|-----|-----|----|-----|
| LW          | R1, 0(R2) | IF | ID | EX  | MEM | WB |       |
| DADD        | R1, R3, R4| IF | ID | Stall | EX | MEM | WB |       |
| LW          | R5, 4(R2) | IF | Stall | ID | EX  | MEM | WB |       |
| DADD        | R5, R1, R5| IF | ID | Stall | EX | MEM | WB |       |
| SW          | R5, 8(R2) | IF | Stall | ID | EX  | MEM | WB |       |

Here, we have 2 stalls resulting from LW followed by using the loaded values, we can schedule the code so that the stalls are not needed by moving an independent instruction between the load and when it is needed.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>R1, 0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>R5, 4(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>DADD</td>
<td>R1, R3, R4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>DADD</td>
<td>R5, R1, R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>R5, 8(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

In this example however, there are no independent instructions that can be moved after the LWs, so we are stuck with the stalls.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Registers</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LW</td>
<td>R2, 0(R1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
<tr>
<td>LW</td>
<td>R3, 0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>Stall</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>DADD</td>
<td>R3, R3, #1</td>
<td>IF</td>
<td>Stall</td>
<td>ID</td>
<td>Stall</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>SW</td>
<td>R3, 0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
</tr>
</tbody>
</table>

This code is equivalent to (*p)++;
Control Hazards
Control Hazards

• Currently, branches are determined in the EX stage and the PC is changed (if it is) in the MEM stage
  – by the time this decision is made, three more instructions have already been fetched into the pipeline
    • if the branch is taken, these instructions should not have been fetched
    • this is known as a control hazard

Branch determined (MUX not shown in this figure)

Later instructions make it into the pipeline that shouldn’t be here because of the branch
Reducing Branch Penalty

- Compile time solutions
  - Decide on a hardware action
  - Compiler tries to use this knowledge

1. Pipeline freeze (or flush)
2. Predicted-not-taken
3. Predicted-taken
4. Delayed branch
Pipeline Freeze

- Hold or delete all instructions after a branch until the target address is known.
  - Simple to implement
  - Results in 1 cycle stall for MIPS
  - Longer stalls for other pipeline architectures
Predicted-not-taken

• Execute successor instructions in sequence
• “Squash” instructions in pipeline if branch actually taken
• Must be careful not to alter state of registers until actual branch target is known
• Only slightly more complicated than pipeline freeze to implement
• Compiler can modify loops to favor branches not taken
Predicted-taken

• Treat every branch as taken
• As soon as branch is decoded and target address is computed, begin fetching at the target
  – No advantage for MIPS because target address is not known any earlier than branch outcome
  – Only makes sense for machines that compute target address before determining branch outcome
Delayed Branch

• Execute instruction after branch no matter what
• Fetch subsequent instruction depending on branch outcome
  branch
  sequential successor instruction
  branch target if taken
• Compiler’s job is to put a useful instruction as the sequential successor instruction
• Otherwise a NOP is used
Performance with Branches

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

For the schemes just mentioned, penalty is at most 1 cycle. Penalty is more for deeper pipelines.