Pipelining

- Reconsider the data path we just did
- Each instruction takes from 3 to 5 clock cycles
- However, there are parts of hardware that are idle many times
- We can reorganize the operation
- Make each hardware block independent
  - 1. Instruction Fetch Unit
  - 2. Register Read Unit
  - 3. ALU Unit
  - 4. Data Memory Read/Write Unit
  - 5. Register Write Unit
- Units in 3 and 5 cannot be independent, but operations can be
- Let each unit just do its required job for each instruction
- If for some instruction, a unit need not do anything, it can simply perform a noop
Gain of Pipelining

- Improve performance by increasing instruction throughput
- Ideal speedup is number of stages in the pipeline
- Do we achieve this? No, why not?
Pipelining

• What makes it easy
  – all instructions are the same length
  – just a few instruction formats
  – memory operands appear only in loads and stores

• What makes it hard?
  – structural hazards: suppose we had only one memory
  – control hazards: need to worry about branch instructions
  – data hazards: an instruction depends on a previous instruction

• We’ll study these issues using a simple pipeline
• Other complication:
  – exception handling
  – trying to improve performance with out-of-order execution, etc.
Basic Idea

- What do we need to add to actually split the datapath into stages?
Pipelined Data Path

Can you find a problem even if there are no dependencies?
What instructions can we execute to manifest the problem?
Corrected Data Path
Execution Time

- Time of n instructions depends on
  - Number of instructions n
  - # of stages k
  - # of control hazard and penalty of each step
  - # of data hazards and penalty for each
- Time = n + k - 1 + load hazard penalty + branch penalty
- Load hazard penalty is 1 or 0 cycle
  - depending on data use with forwarding
- branch penalty is 3, 2, 1, or zero cycles depending on scheme
Design and Performance Issues With Pipelining

- Pipelined processors are not EASY to design
- Technology affect implementation
- Instruction set design affect the performance, i.e., beq, bne
- More stages do not lead to higher performance
Pipeline Operation

- In pipeline one operation begins in every cycle
- Also, one operation completes in each cycle
- Each instruction takes 5 clock cycles (k cycles in general)
- When a stage is not used, no control needs to be applied
- In one clock cycle, several instructions are active
- Different stages are executing different instructions
- How to generate control signals for them is an issue
Graphically Representing Pipelines

• Can help with answering questions like:
  – how many cycles does it take to execute this code?
  – what is the ALU doing during cycle 4?
  – use this representation to help understand datapaths
## Instruction Format

<table>
<thead>
<tr>
<th></th>
<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>LW</td>
<td>REG 1</td>
<td>REG 2</td>
<td>LOAD ADDRESS</td>
<td>OFFSET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW</td>
<td>REG 1</td>
<td>REG 2</td>
<td>STORE ADDRESS</td>
<td>OFFSET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R-TYPE</td>
<td>REG 1</td>
<td>REG 2</td>
<td>DST</td>
<td>SHIFT AMOUNT</td>
<td>ADD/AND/OR/SLT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ/BNE</td>
<td>REG 1</td>
<td>REG 2</td>
<td>BRANCH ADDRESS</td>
<td>OFFSET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>JUMP</td>
<td>ADDRESS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Operation for Each Instruction

<table>
<thead>
<tr>
<th></th>
<th>LW:</th>
<th>SW:</th>
<th>R-Type:</th>
<th>BR-Type:</th>
<th>JMP-Type:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong></td>
<td>READ INST</td>
<td>1. READ INST</td>
<td>1. READ INST</td>
<td>1. READ INST</td>
<td>1. READ INST</td>
</tr>
<tr>
<td><strong>2.</strong></td>
<td>READ REG 1</td>
<td>READ REG 1</td>
<td>2. READ REG 1</td>
<td>2. READ REG 1</td>
<td>2. READ REG 1</td>
</tr>
<tr>
<td></td>
<td>READ REG 2</td>
<td>READ REG 2</td>
<td>3. ADD REG 1 +</td>
<td>3. OPERATE on</td>
<td>3. SUB REG 2</td>
</tr>
<tr>
<td></td>
<td><strong>OFFSET</strong></td>
<td><strong>OFFSET</strong></td>
<td><strong>OFFSET</strong></td>
<td><strong>REG 1 / REG 2</strong></td>
<td>from REG 1</td>
</tr>
<tr>
<td><strong>3.</strong></td>
<td>ADD REG 1 +</td>
<td>3. ADD REG 1 +</td>
<td>4.</td>
<td>4.</td>
<td>4.</td>
</tr>
<tr>
<td></td>
<td>OFFSET</td>
<td>OFFSET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>4.</strong></td>
<td>READ MEM</td>
<td>4. WRITE MEM</td>
<td>5. WRITE DST</td>
<td>5. WRITE DST</td>
<td>5. WRITE DST</td>
</tr>
<tr>
<td><strong>5.</strong></td>
<td>WRITE REG2</td>
<td>5.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipeline Data Path Operation
Fetch Unit

Branch Address

Jump Register Address

Jump Address

MUX

MUX

MUX

4

ADD

NPC

INST

INST MEMORY

INST 31-00

IA
Register Fetch Unit
ALU Operation and Branch Logic

![ALU Operation and Branch Logic Diagram]
Memory and Write back Stage
Pipeline Data Path Operation
Dependencies

- Problem with starting next instruction before first is finished
  - dependencies that “go backward in time” are data hazards

Program execution order:
(in instructions)

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)

Value of register $2:

<table>
<thead>
<tr>
<th></th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (in clock cycles)</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

- Dependencies that “go backward in time” are data hazards.
A program with data dependencies

- Consider the following program

  ```
  add $t0, $t1, $t2
  add $t1, $t0, $t3
  and $t2, $t4, $t0
  or $t3, $t1, $t0
  slt $t4, $t2, $t3
  ```

- Problem with starting next instruction before first is finished
  - dependencies that “go backward in time” are data hazards
add $t0, $t1, $t2

add $t1, $t0, $t3

and $t2, $t4, $t0

or $t3, $t1, $t0

slt $t4, $t2, $t3
Solution: Software No-ops/Hardware Bubbles

• Have compiler guarantee no hazards
• Where do we insert the “no-ops”?

```
sub $2, $1, $3
and $12, $2, $5
or $13, $6, $2
add $14, $2, $2
sw $15, 100($2)
```

Problem: this really slows us down!

– Also, the program will always be slow even if a techniques like forwarding is employed afterwards in newer version

• Hardware can detect dependencies and insert no-ops in hardware
  – Hardware detection and no-op insertion is called stalling
  – This is a bubble in pipeline and waste one cycle at all stages
  – Need two or three bubbles between write and read of a register
Hazard Detection Unit

- Stall by letting an instruction that won’t write anything go forward
Stalling

- Hardware detection and no-op insertion is called stalling
- We stall the pipeline by keeping an instruction in the same stage

Program execution order (in instructions)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
<th>CC 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $2, 20($1)</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td>IM</td>
</tr>
<tr>
<td>and $4, $2, $5</td>
<td>IM</td>
<td>Reg</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>IM</td>
</tr>
<tr>
<td>or $8, $2, $6</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
<td>IM</td>
</tr>
<tr>
<td>add $9, $4, $2</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
</tr>
<tr>
<td>slt $1, $6, $7</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
<td>DM</td>
<td>Reg</td>
<td>IM</td>
<td>Reg</td>
</tr>
</tbody>
</table>

Diagram showing the pipeline stages and instructions.
Stalled Operation (no write before read)

add $t0, $t1, $t2
add $t1, $t0, $t3
add $t1, $t0, $t3
add $t1, $t0, $t3
Stalled Operation (write before read)

add $t0, $t1, $t2

add $t1, $t0, $t3

and $t2, $t4, $t0
Detecting Hazards for Forwarding

- **EX hazard**
  - If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd != 0) and (EX/MEM.REGISTERRd = ID/EX.RegisterRs)) ForwardA = 10
  - If ((EX/MEM.RegWrite) and (EX/MEM.RegisterRd != 0) and (EX/MEM.RegisterRd = ID/EX.RegisterRt)) ForwardB = 10

- **MEM hazard**
  - If ((MEM/WB.RegWrite) and (MEM/WB.RegisterRd != 0) and (MEM/WB.REGISTERRd = ID/EX.RegisterRs)) ForwardA = 01
  - If ((MEM/WB.RegWrite) and (MEM/WB.RegisterRd != 0) and (MEM/WB.REGISTERRd = ID/EX.RegisterRt)) ForwardB = 10

- In case of `lw` followed by a `sw` instruction, forwarding will not work. This is because data in MEM stage are still being read
  - Plan on adding forwarding in MEM stage of put a stall/ bubble

- In case of `lw` followed by an instruction that uses the value
  - One has to add an stall
Forwarding

- Use temporary results, don’t wait for them to be written
  - register file forwarding to handle read/write to same register
  - ALU forwarding
  - May also need forwarding to memory (think!!)

<table>
<thead>
<tr>
<th>Time (in clock cycles)</th>
<th>CC 1</th>
<th>CC 2</th>
<th>CC 3</th>
<th>CC 4</th>
<th>CC 5</th>
<th>CC 6</th>
<th>CC 7</th>
<th>CC 8</th>
<th>CC 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of register $2$ :</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10/20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
<td>-20</td>
</tr>
<tr>
<td>Value of EX/MEM :</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Value of MEM/WB :</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-20</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Program execution order (in instructions)

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, CON $2

What if this $2$ was $13$?
Forwarding
Can't always forward

• **Load word can still cause a hazard:**
  – an instruction tries to read a register following a load instruction that writes to the same register.

• Thus, we need a hazard detection unit to “stall” the load instruction
Branch Hazards

- When we decide to branch, other instructions are in the pipeline!

- We are predicting “branch not taken”
  - need to add hardware for flushing instructions if we are wrong
Improving Performance

• Try and avoid stalls! E.g., reorder these instructions:

```assembly
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

• Add a “branch delay slot”
  – the next instruction after a branch is always executed
  – rely on compiler to “fill” the slot with something useful

• Superscalar: start more than one instruction in the same cycle
Other Issues in Pipelines

• Exceptions
  – Errors in ALU for arithmetic instructions
  – Memory non-availability
• Exceptions lead to a jump in a program
• However, the current PC value must be saved so that the program can return to it back for recoverable errors
• Multiple exception can occur in a pipeline
• Preciseness of exception location is important in some cases
• I/O exceptions are handled in the same manner
Handling Branches

• Branch Prediction
  – Usually we may simply assume that branch is not taken
  – If it is taken, then we flush the pipeline
    • Clear control signals for instruction following branch

• Delayed branch
  – Fill instructions that need to be executed even if branch occur
  – If none available fill NOOPs

• Reduce delay in resolving branches
  – Compare at register stage
  – Branch prediction table
    • PC value (for branch) and next address
    • One or two bits to store what should be prediction
Two State vs Four State Branch Prediction

- Two state model

- Four State Model
Pipeline with Early Branch Resolution/Exception
Superscalar Architecture
A Modern Pipelined Microprocessor
Important Facts to Remember

• Pipelined processors divide the execution in multiple steps
• However pipeline hazards reduce performance
  – Structural, data, and control hazard
• Data forwarding helps resolve data hazards
  – But all hazards cannot be resolved
  – Some data hazards require bubble or noop insertion
• Effects of control hazard reduced by branch prediction
  – Predict always taken, delayed slots, branch prediction table
  – Structural hazards are resolved by duplicating resources
Pipeline control

• We have 5 stages. What needs to be controlled in each stage?
  – Instruction Fetch and PC Increment
  – Instruction Decode / Register Fetch
  – Execution
  – Memory Stage
  – Write Back

• How would control be handled in an automobile plant?
  – a fancy control center telling everyone what to do?
  – should we use a finite state machine?
Pipeline Control
Pipeline Control

- Pass control signals along just like the data

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Execution/Address Calculation stage control lines</th>
<th>Memory access stage control lines</th>
<th>stage control lines</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reg Dst ALU Op1 ALU Op0 ALU Src Branch Mem Read Mem Write</td>
<td>Reg write Mem to Reg</td>
<td></td>
</tr>
<tr>
<td>R-format</td>
<td>1 1 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>lw</td>
<td>0 0 0 0 0 1 0 1 0 1</td>
<td>0 1 0 1</td>
<td>1 1</td>
</tr>
<tr>
<td>sw</td>
<td>X 0 0 0 1 0 0 0 1 0</td>
<td>0 0 1 0</td>
<td>0 X</td>
</tr>
<tr>
<td>beq</td>
<td>X 0 1 0 0 1 0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 X</td>
</tr>
</tbody>
</table>

IF/ID

ID/EX

EX/MEM

MEM/WB

Control

Instruction

WB

M

EX
Data Path with Control
Flushing Instructions