1. Data Hazards and Forwarding
2. Control Hazard Solution
3. How to Handle Exception
1. How to implement “data forwarding”?
2. How to detect load-use hazard? How to stall pipeline?
3. How to resolve branch in the decode stage?
4. How to flush pipeline?
Data Dependence Detection & Forwarding

sub r2, r1, r3
and r12, r2, r5
or r13, r6, r2
add r14, r2, r2
sw r15, 100($r2)
add r14, r2, r2  
or r13, r6, r2  
and r12, r2, r5  
sub r2, r1, r3
How to detect dependency between (sub, and)?

Instr. Order

Time (clock cycles)

sub r2, r1, r3 and r12, r2, r5 or r13, r6, r2
add r14, r2, r2
sw r15, 100($r2)

1a: EX/MEM. RegisterRd = ID/EX.RegisterRs
1b: EX/MEM. RegisterRd = ID/EX.RegisterRt
sw r15, 100($r2)  
add r14, r2, r2  
or r13, r6, r2  
and r12, r2, r5  
sub r2, r1, r3
How to detect dependency between (sub, or)?

2a: MEM//WB.RegisterRd = ID/EX.RegisterRs (sub & or)
2b: MEM/WB.RegisterRd = ID/EX.RegisterRt
Hazard conditions:

- 1a: EX/MEM. RegisterRd = ID/EX.RegisterRs (sub & and)
- 1b: EX/MEM. RegisterRd = ID/EX.RegisterRt
- 2a: MEM/WB.RegisterRd = ID/EX.RegisterRs (sub & or)
- 2b: MEM/WB.RegisterRd = ID/EX.RegisterRt
- RegWrite signal of WB Control field
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- EX/MEM.RegisterRd <> $0
- MEM/WB.RegisterRd <> $0

How to forward data?
Resolving Hazards by Forwarding

- Use the value in pipeline registers rather than waiting for the WB stage to write the register file.
  - EX/MEM.Aluout
  - MEM/WB.Aluout

<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</td>
<td>10/-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 $15, 100(2)$

use the value stored in EX/MEM

use the value stored in MEM/WB
Forwarding Logic

- **Forwarding**: input to ALU from any pipe reg.
  - Add multiplexors to ALU input
  - Forwarding Control will be in EX
## Forwarding Control

<table>
<thead>
<tr>
<th>Mux control</th>
<th>Source</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForwardA = 00</td>
<td>ID/EX</td>
<td>The first ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardA = 10</td>
<td>EX/MEM</td>
<td>The first ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardA = 01</td>
<td>MEM/WB</td>
<td>The first ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
<tr>
<td>ForwardB = 00</td>
<td>ID/EX</td>
<td>The second ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardB = 10</td>
<td>EX/MEM</td>
<td>The second ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardB = 01</td>
<td>MEM/WB</td>
<td>The second ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
</tbody>
</table>

---

### b. With forwarding

[Diagram showing the flow of data with forwarding units.]
**Forwarding Control**

<table>
<thead>
<tr>
<th>Mux control</th>
<th>Source</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForwardA = 00</td>
<td>ID/EX</td>
<td>The first ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardA = 10</td>
<td>EX/MEM</td>
<td>The first ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardA = 01</td>
<td>MEM/WB</td>
<td>The first ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
<tr>
<td>ForwardB = 00</td>
<td>ID/EX</td>
<td>The second ALU operand comes from the register file.</td>
</tr>
<tr>
<td>ForwardB = 10</td>
<td>EX/MEM</td>
<td>The second ALU operand is forwarded from the prior ALU result.</td>
</tr>
<tr>
<td>ForwardB = 01</td>
<td>MEM/WB</td>
<td>The second ALU operand is forwarded from data memory or an earlier ALU result.</td>
</tr>
</tbody>
</table>

1. **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.RegisterRs))
   ForwardB = 10

2. **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 = 01
1. **EX hazard**

   if (EX/MEM.RegWrite
       and (EX/MEM.RegisterRd ≠ 0)
       and (EX/MEM.RegisterRd=ID/EX.RegisterRs))

   ForwardA = 10
2. MEM hazard
if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0) and (MEM/WB.RegisterRd=ID/Ex.RegisterRt))
ForwardB = 01
Forwarding Control (cont.)

\[
\begin{align*}
\text{inst1} & \quad \text{add} \; $1,$1,$2; & \quad \text{IF ID EX MEM WB} \\
\text{inst2} & \quad \text{add} \; $1,$1,$3; & \quad \text{IF ID EX MEM WB} \\
\text{inst3} & \quad \text{add} \; $1,$1,$4; & \quad \text{IF ID EX MEM WB} \\
\end{align*}
\]

......

Which instruction should forward its results to instruction 3?

**MEM hazard condition becomes**

\[
\begin{align*}
\text{if (MEM/WB.RegWrite} & \quad \text{and (MEM/WB.RegisterRd} 
eq 0) \\
& \quad \text{and (Ex/MEM.RegisterRd} \neq \text{ID/Ex.RegisterRs} \\
& \quad \text{and (MEM/WB.RegRd} = \text{ID/Ex.RegisterRs})) \quad \text{ForwardA} = 01
\end{align*}
\]

\[
\begin{align*}
\text{if (MEM/WB.RegWrite} & \quad \text{and (MEM/WB.RegRd} 
eq 0) \\
& \quad \text{and (Ex/MEM.RegisterRd} \neq \text{ID/Ex.RegisterRt} \\
& \quad \text{and (MEM/WB.RegRd} = \text{ID/Ex.RegisterRt})) \quad \text{ForwardB} = 01
\end{align*}
\]
Datapath with Forwarding
Example

Show how forwarding works with this instruction sequence (with dependencies highlighted):

- sub $2, $1, $3
- and $4, $2, $5
- or $4, $4, $2
- add $9, $4, $2
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.

If (ID/EX.MemRead and ((ID/EX.RegisterRt = IF/ID.RegisterRs) or (ID/EX.RegisterRt = IF/ID.RegisterRt)))

stall the pipeline
Hazard Detection and Stall

- If (ID/EX.MemRead and
  ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
  (ID/EX.RegisterRt = IF/ID.REgisterRt)))
  **stall the pipeline**

- Stall the pipeline
  - Preventing instructions in the IF and ID stages from making progress
    - Preserve the PC and IF/ID pipeline registers
  - We need to do nothing in EX at CC4, MEM at CC5, WB at CC6
    - Deasserting all nine control signals in the EX, MEM and WB stage

![Program execution order and pipeline stages diagram]
Stall/Bubble in the Pipeline

Program execution order (in instructions)

lws, 20($1)

and becomes nop

and $4, $2, $5 stalled in ID

or $8, $2, $6 stalled in IF

add $9, $4, $2

Or, more accurately...
Hazard Detection Unit

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

Show how hazard detection unit works with this instruction sequence (with dependencies highlighted):

\[
\begin{aligned}
\text{lw} & \; \text{\$2, 20\text{(\$1)}} \\
\text{and} & \; \text{\$4, \$2, \$5} \\
\text{or} & \; \text{\$4, \$4, \$2} \\
\text{add} & \; \text{\$9, \$4, \$2}
\end{aligned}
\]

\{ 
- load-use data hazard \\
- Forwarding
\}
Clock 5
Clock 7
Control Hazard Solutions

- Stall: wait until decision is clear

- Impact: 3 clock cycles per branch instruction
  => slow
Control Hazard Solution (1): Reducing the Delay of Branches (Example: BEQZ, BNEZ)
Data Hazards for Branches

- If a comparison register is a destination of 2nd or 3rd preceding ALU instruction

\[
\begin{align*}
&\text{add } \$1, \$2, \$3 &\quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
&\text{add } \$4, \$5, \$6 &\quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
&\quad \vdots &\quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
&\text{beq } \$1, \$4, \text{target} &\quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB}
\end{align*}
\]

- Can resolve using forwarding
Data Hazards for Branches

- If a comparison register is a destination of preceding ALU instruction or 2nd preceding load instruction
  - Need 1 stall cycle

lw $1, addr
add $4, $5, $6
beq stalled
beq $1, $4, target
Data Hazards for Branches

- If a comparison register is a destination of immediately preceding load instruction
  - Need 2 stall cycles

```
lw  $1, addr          IF  ID  EX  MEM  WB
beq  stalled         IF  ID
beq  stalled         ID
beq  $1, $0, target   ID  EX  MEM  WB
```
Impact: 2 clock cycles per branch instruction
=> slow
Control Hazard Solutions (2)

- Predict: guess one direction then back up if wrong
  - Predict not taken

- Impact: 1 clock cycles per branch instruction if right, 2 if wrong (right - 50% of time)

- More dynamic scheme: history of 1 branch (- 90%)
# Predict branch not taken

<table>
<thead>
<tr>
<th>Branch Inst (i)</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst i+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst i+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst i+4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
</tbody>
</table>

Correct Prediction: Zero Cycle Branch Penalty!

<table>
<thead>
<tr>
<th>Branch Inst (i)</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst i+1</td>
<td>IF</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
<td>nop</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

Incorrect Prediction - waste one cycle
How to flush pipeline?
Zero the instruction field of the IF/ID pipeline register
sll $0, $0, 0
Dynamic Branch Prediction

- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check (saves HW, but may not be right branch)

32-bit branch address

Last n-bit

| 00  | NT (0) |
| 01  | T (1)  |
| 10  | NT (0) |
| 11  | T (1)  |

branch history table = $2^n$

Example: --00 T, --01 NT, --10 T, --11 T, --00 NT, --01 NT, --10 NT, --11 T,
Dynamic Branch Prediction

Problem: in a loop, 1-bit BHT will cause 2 mispredictions (avg is 9 iterations before exit):
- End of loop case, when it exits instead of looping as before
- First time through loop on next time through code, when it predicts exit instead of looping
- Only 80% accuracy even if loop 90% of the time

L1: :::
    ::
    ::
    ::
    SUBI R1, #8
    BNEZ R1, L1
Solution: 2-bit scheme which changes prediction only if get misprediction \textit{twice}:
Exercise

- Branch outcome of a single branch
  - T T T N N N T T T

- How many instances of this branch instruction are mis-predicted with a 1-bit predictor?

- How many instances of this branch instruction are mis-predicted with a 2-bit predictor?
Problems with predicted taken?
  – Need to calculate target address
  – Solution: branch target buffer

Correlating predictor
  – A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches

Tournament branch predictor
  – A branch predictor with multiple prediction for each branch and a selection mechanism that chooses which predictor to enable for a given branch
### Exceptions

<table>
<thead>
<tr>
<th>inst i</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst i+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
</tr>
<tr>
<td>inst i+2</td>
<td>IF</td>
<td>ID</td>
<td></td>
<td></td>
<td>EX</td>
</tr>
<tr>
<td>inst i+3</td>
<td>IF</td>
<td></td>
<td>ID</td>
<td></td>
<td>IF</td>
</tr>
<tr>
<td>inst i+4</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td>IF</td>
</tr>
</tbody>
</table>

Faulting instruction - overflow

**Steps to handle exceptions:**

- Flush the instruction in the IF, ID and EX stages.
- Let all preceding instructions complete if they can.
- EPC = address of (offending instruction) + 4
- Call the OS to handle the exception
  - PC = 0x40000040
- Return from the exception handler
  - PC = EPC - 4
Datapath with Controls to Handle Exceptions
Example: Handling Exception

Given the following instruction sequence:
- \(40_{\text{hex}}\) sub $11, $2, $4
- \(44_{\text{hex}}\) and $12, $2, $5
- \(48_{\text{hex}}\) or $13, $2, $6
- \(4C_{\text{hex}}\) add $1, $2, $1
- \(50_{\text{hex}}\) slt $15, $6, $7
- \(54_{\text{hex}}\) lw $16, 50($7)

Assume the instruction to be invoked on an exception begin like this
- \(40000040_{\text{hex}}\) sw $25, 1000($0)
- \(40000044_{\text{hex}}\) sw $26, 1004($0)

Show what happens in the pipeline if an overflow exception occurs in the add instruction.
Example: Handling Exception

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>40hex sub</td>
<td>$11, $2, $4</td>
<td></td>
</tr>
<tr>
<td>44hex and</td>
<td>$12, $2, $5</td>
<td></td>
</tr>
<tr>
<td>48hex or</td>
<td>$13, $2, $6</td>
<td></td>
</tr>
<tr>
<td>4Chex add</td>
<td>$1, $2, $1</td>
<td></td>
</tr>
<tr>
<td>50hex slt</td>
<td>$15, $6, $7</td>
<td></td>
</tr>
<tr>
<td>54hex lw</td>
<td>$16, 50($7)</td>
<td></td>
</tr>
</tbody>
</table>

1. The overflow is detected when “add” is in the EXE stage.
2. Save PC+4 (50) in EPC
3. Assert IF.Flush, ID. Flush, and EX.Flush

lw $16, 50($7) → slt $15, $6, $7 → add $1, $2, $1 → or $13, ... → and $12, ...

Clock 5
1. Fetch stage: the first instruction of the exception routine
2. Instruction prior to the add instruction complete
Parallelism via Instruction: Multiple Issue

- **Static multiple issue**
  - Compiler groups instructions to be issued together
  - Packages them into “issue slots”
  - Compiler detects and avoids hazards

- **Dynamic multiple issue**
  - CPU examines instruction stream and chooses instructions to issue each cycle
  - Compiler can help by reordering instructions
  - CPU resolves hazards using advanced techniques at runtime

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>
Speculation

- “Guess” what to do with an instruction
  - Start operation as soon as possible
  - Check whether guess was right
    - If so, complete the operation
    - If not, roll-back and do the right thing

- Common to static and dynamic multiple issue

- Examples
  - Speculate on branch outcome
    - Roll back if path taken is different
  - Speculate on load
    - Roll back if location is updated
Compiler/Hardware Speculation

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include “fix-up” instructions to recover from incorrect guess

- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation
Static Multiple Issue

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required

- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - \( \Rightarrow \) Very Long Instruction Word (VLIW)

- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies with a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with nop if necessary
MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction withnop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>ALU/branch</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>n + 4</td>
<td>Load/store</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>n + 8</td>
<td>ALU/branch</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>n + 12</td>
<td>Load/store</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>n + 16</td>
<td>ALU/branch</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
<tr>
<td>n + 20</td>
<td>Load/store</td>
<td>IF  ID  EX  MEM  WB</td>
</tr>
</tbody>
</table>
Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    - add $t0, $s0, $s1
    - load $s2, 0($t0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required
Scheduling Example

Schedule this for dual-issue MIPS

Loop: lw $t0, 0($_s1)  # $t0=array element
      addu $t0, $t0, $s2  # add scalar in $s2
      sw $t0, 0($_s1)    # store result
      addi $_s1, $_s1,-4 # decrement pointer
      bne $_s1, $zero, Loop  # branch $_s1!=0

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>lw $t0, 0($_s1)</td>
<td>1</td>
</tr>
<tr>
<td>addi $_s1, $_s1,-4</td>
<td>nop</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>nop</td>
<td>3</td>
</tr>
<tr>
<td>bne $_s1, $zero, Loop</td>
<td>sw $t0, 4($_s1)</td>
<td>4</td>
</tr>
</tbody>
</table>

IPC = 5/4 = 1.25 (c.f. peak IPC = 2)
Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead

- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name
Loop:  lw   $t0, 0($s1)
       addu  $t0, $t0, $s2
       sw    $t0, 0($s1)
       addi  $s1, $s1, –4
       bne   $s1, $zero, Loop
Unrolled Loop That Minimizes Stalls

Renaming

Loop:

lw $t0, 0($s1)
addu $t0, $t0, $s2
sw $t0, 0($s1)

lw $t0, -4($s1)
addu $t0, $t0, $s2
sw $t0, -4($s1)

lw $t0, -8($s1)
addu $t0, $t0, $s2
sw $t0, -8($s1)

lw $t0, -12($s1)
addu $t0, $t0, $s2
sw $t0, -12($s1)

addi $s1, $s1, -16
bne $s1, $zero, Loop

Loop:

lw $t0, 0($s1)
addu $t0, $t0, $s2
sw $t0, 0($s1)

lw $t1, -4($s1)
addu $t1, $t0, $s2
sw $t1, -4($s1)

lw $t2, -8($s1)
addu $t2, $t0, $s2
sw $t2, -8($s1)

lw $t3, -12($s1)
addu $t3, $t0, $s2
sw $t3, -12($s1)

addi $s1, $s1, -16
bne $s1, $zero, Loop
## Loop Unrolling Example

<table>
<thead>
<tr>
<th></th>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td>addi $s1, $s1, -16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>addu $t0, $t0, $s2</td>
<td>lw $t1, -4($s1)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>addu $t1, $t1, $s2</td>
<td>lw $t2, -8($s1)</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 0($s1)</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>addu $t3, $t3, $s2</td>
<td>sw $t1, -4($s1)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw $t2, -8($s1)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lw $t3, -12($s1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>addu $t3, $t3, $s2</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sw $t3, -12($s1)</td>
<td>10</td>
</tr>
</tbody>
</table>

**Original version**

lw $t0, 0($s1)
addu $t0, $t0, $s2
sw $t0, 0($s1)

lw $t1, -4($s1)
addu $t1, $t0, $s2
sw $t1, -4($s1)

lw $t2, -8($s1)
addu $t2, $t0, $s2
sw $t2, -8($s1)

lw $t3, -12($s1)
addu $t3, $t0, $s2
sw $t3, -12($s1)

addi $s1, $s1, -16
bne $s1, $zero, Loop

Chapter 4 — The Processor — 63
Loop Unrolling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loop:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,-16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>sw $t2, 8($s1)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>lw $t3, 4($s1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>addu $t3, $t3, $s2</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td></td>
</tr>
</tbody>
</table>

**Original version**

Chapter 4 — The Processor — 64
## Loop Unrolling Example

### Loop:

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $s1, $s1, –16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>sw $t2, 8($s1)</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>lw $t3, 4($s1)</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>addu $t3, $t3, $s2</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
</tr>
</tbody>
</table>

### Another optimization opportunity!!

```
lw $t0, 0($s1)
addu $t0, $t0, $s2
sw $t0, 0($s1)
lw $t1, -4($s1)
addu $t1, $t0, $s2
sw $t1, -4($s1)
lw $t2, -8($s1)
addu $t2, $t0, $s2
sw $t2, -8($s1)
lw $t3, -12($s1)
addu $t3, $t0, $s2
sw $t3, -12($s1)
addi $s1, $s1, –16
bne $s1, $zero, Loop
```
Loop Unrolling Example

<table>
<thead>
<tr>
<th>ALU/branch</th>
<th>Load/store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,–16</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>lw $t1, 12($s1)</td>
<td>2</td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td>lw $t2, 8($s1)</td>
<td>3</td>
</tr>
<tr>
<td>addu $t1, $t1, $s2</td>
<td>lw $t3, 4($s1)</td>
<td>4</td>
</tr>
<tr>
<td>addu $t2, $t2, $s2</td>
<td>sw $t0, 16($s1)</td>
<td>5</td>
</tr>
<tr>
<td>addu $t3, $t3, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td>nop</td>
<td>sw $t2, 8($s1)</td>
<td>7</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td>sw $t3, 4($s1)</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>lw $t0, 0($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addu $t0, $t0, $s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $t0, 0($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t1, –4($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addu $t1, $t0, $s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $t1, –4($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t2, –8($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addu $t2, $t0, $s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $t2, –8($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw $t3, –12($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addu $t3, $t0, $s2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sw $t3, –12($s1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>addi $s1, $s1,–16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- IPC = 14/8 = 1.75
  - Closer to 2, but at cost of registers and code size
Dynamic Multiple Issue

- "Superscalar" processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU
Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order

Example

```
lw   $t0, 20($s2)
addu $t1, $t0, $t2
sub  $s4, $s4, $t3
slti $t5, $s4, 20
```
- Can start sub while addu is waiting for lw
Dynamically Scheduled CPU

Instruction fetch and decode unit

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reservation station

Reorders buffer for register writes

Commit unit

Out-of-order execute

In-order issue

Preserves dependencies

Hold pending operands

Results also sent to any waiting reservation stations

In-order commit

Can supply operands for issued instructions
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicatable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
Does Multiple Issue Work?

The BIG Picture

- Yes, but not as much as we’d like
- Programs have real dependencies that limit ILP
- Memory delays and limited bandwidth
  - Hard to keep pipelines full
- Speculation can help if done well
## Power Efficiency

- Complexity of dynamic scheduling and speculations requires power
- Multiple simpler cores may be better

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order/Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600MHz</td>
<td>31</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core</td>
<td>2006</td>
<td>2930MHz</td>
<td>14</td>
<td>4</td>
<td>Yes</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>UltraSparc III</td>
<td>2003</td>
<td>1950MHz</td>
<td>14</td>
<td>4</td>
<td>No</td>
<td>1</td>
<td>90W</td>
</tr>
<tr>
<td>UltraSparc T1</td>
<td>2005</td>
<td>1200MHz</td>
<td>6</td>
<td>1</td>
<td>No</td>
<td>8</td>
<td>70W</td>
</tr>
</tbody>
</table>
Concluding Remarks

- ISA influences design of datapath and control
- Datapath and control influence design of ISA
- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards: structural, data, control
- Multiple issue and dynamic scheduling (ILP)
  - Dependencies limit achievable parallelism
  - Complexity leads to the power wall