MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register
### Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>
### Pipeline Performance

**Single-cycle ($T_c = 800ps$)**

Program execution order (in instructions)

- `lw $1, 100($0)`
- `lw $2, 200($0)`
- `lw $3, 300($0)`

### Pipelined ($T_c = 200ps$)

Program execution order (in instructions)

- `lw $1, 100($0)`
- `lw $2, 200($0)`
- `lw $3, 300($0)`

Diagram showing the timing and execution order for both single-cycle and pipelined operations.
Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - Time between instructions_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}

- If not balanced, speedup is less

- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease
Pipelining and ISA Design

- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
- Load/store addressing
  - Can calculate address in 3rd stage, access memory in 4th stage
- Alignment of memory operands
  - Memory access takes only one cycle
Hazard

- Situations that prevent starting the next instruction in the next cycle

**Structure hazards**
- A required resource is busy

**Data hazard**
- Need to wait for previous instruction to complete its data read/write

**Control hazard**
- Deciding on control action depends on previous instruction
Structure Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to stall for that cycle
    - Would cause a pipeline “bubble”

- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
Data Hazards

- An instruction depends on completion of data access by a previous instruction

- add $s0, $t0, $t1
- sub $t2, $s0, $t3
Forwarding (aka Bypassing)

- Use result when it is computed
- Don’t wait for it to be stored in a register
- Requires extra connections in the datapath

Program execution order (in instructions)

- add $s0, $t0, $t1
- sub $t2, $s0, $t3

Diagram of pipeline stages:

- IF: Instruction Fetch
- ID: Instruction Decode
- EX: Execution
- MEM: Memory Access
- WB: Write Back

Timeline:

- Time: 200, 400, 600, 800, 1000
Load-Use Data Hazard

- Can’t always avoid stalls by forwarding
  - If value not computed when needed
  - Can’t forward backward in time!

Program execution order (in instructions)

lw $s0, 20($t1)

sub $t2, $s0, $t3
Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for $A = B + E; \ C = B + F;$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Original Execution</th>
<th>Modified Execution</th>
<th>Cycle Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t1, 0($t0)</td>
<td>lw $t2, 4($t0)</td>
<td>lw $t1, 0($t0)</td>
<td>13 cycles</td>
</tr>
<tr>
<td>lw $t2, 4($t0)</td>
<td>add $t3, $t1, $t2</td>
<td>lw $t2, 4($t0)</td>
<td></td>
</tr>
<tr>
<td>add $t3, $t1, $t2</td>
<td>sw $t3, 12($t0)</td>
<td>add $t3, $t1, $t2</td>
<td></td>
</tr>
<tr>
<td>sw $t3, 12($t0)</td>
<td>lw $t4, 8($t0)</td>
<td>sw $t3, 12($t0)</td>
<td></td>
</tr>
<tr>
<td>lw $t4, 8($t0)</td>
<td>add $t5, $t1, $t4</td>
<td>lw $t4, 8($t0)</td>
<td></td>
</tr>
<tr>
<td>add $t5, $t1, $t4</td>
<td>sw $t5, 16($t0)</td>
<td>add $t5, $t1, $t4</td>
<td></td>
</tr>
<tr>
<td>sw $t5, 16($t0)</td>
<td></td>
<td>sw $t5, 16($t0)</td>
<td>11 cycles</td>
</tr>
</tbody>
</table>
Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch

- In MIPS pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage
Stall on Branch

- Wait until branch outcome determined before fetching next instruction

Diagram showing the execution order of instructions and the time taken for each stage.
Branch Prediction

- Longer pipelines can’t readily determine branch outcome early
  - Stall penalty becomes unacceptable

- Predict outcome of branch
  - Only stall if prediction is wrong

- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay
MIPS with Predict Not Taken

Prediction correct

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

Program execution order (in instructions)

Time

200 400 600 800 1000 1200 1400

Instruction fetch Reg ALU Data access Reg

200 ps

Instruction fetch Reg ALU Data access Reg

Prediction incorrect

add $4, $5, $6
beq $1, $2, 40
lw $3, 300($0)

Program execution order (in instructions)

Time

200 400 600 800 1000 1200 1400

Instruction fetch Reg ALU Data access Reg

200 ps

Instruction fetch Reg ALU Data access Reg

or $7, $8, $9

400 ps

bubble bubble bubble bubble bubble bubble
More-Realistic Branch Prediction

- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken

- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history
Pipelining improves performance by increasing instruction throughput
- Executes multiple instructions in parallel
- Each instruction has the same latency

Subject to hazards
- Structure, data, control

Instruction set design affects complexity of pipeline implementation
MIPS Pipelined Datapath

Right-to-left flow leads to hazards
Pipeline registers

- Need registers between stages
- To hold information produced in previous cycle
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - “Single-clock-cycle” pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. “multi-clock-cycle” diagram
    - Graph of operation over time
- We’ll look at “single-clock-cycle” diagrams for load & store
IF for Load, Store, ...
ID for Load, Store, …
EX for Load
WB for Load

Wrong register number
Corrected Datapath for Load
EX for Store

Chapter 4 — The Processor — 27
MEM for Store
WB for Store
Multi-Cycle Pipeline Diagram

Form showing resource usage

Time (in clock cycles)

CC 1  CC 2  CC 3  CC 4  CC 5  CC 6  CC 7  CC 8  CC 9

Program execution order (in instructions)

lw $10, 20($1)

sub $11, $2, $3

add $12, $3, $4

lw $13, 24($1)

add $14, $5, $6
Multi-Cycle Pipeline Diagram

Traditional form

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>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execution</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data access</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write back</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add $14, $5, $6</td>
<td>Instruction fetch</td>
<td>Instruction decode</td>
<td>Execution</td>
<td>Memory</td>
</tr>
<tr>
<td>Lw $13, 24 ($1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add $12, $3, $4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub $11, $2, $3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lw $10, 20($1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pipelined Control (Simplified)
Pipelined Control

- Control signals derived from instruction
- As in single-cycle implementation
Pipelined Control
Data Hazards in ALU Instructions

Consider this sequence:

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

We can resolve hazards with forwarding

- How do we detect when to forward?
Dependencies & Forwarding

<table>
<thead>
<tr>
<th>Value of register $2$:</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></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>
</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)
Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register

- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt

- Data hazards when
  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRt
Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not $zero
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0
Forwarding Paths

b. With forwarding
Forwarding Conditions

- **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 = 01
Double Data Hazard

- Consider the sequence:
  
  \[
  \text{add } $1, $1, $2 \\
  \text{add } $1, $1, $3 \\
  \text{add } $1, $1, $4 \\
  \]

- Both hazards occur
  
  - Want to use the most recent

- Revise MEM hazard condition
  
  - Only fwd if EX hazard condition isn’t true
Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
      and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
      and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
      and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
      and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
      and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01
Datapath with Forwarding

Chapter 4 — The Processor — 44
Load-Use Data Hazard

Program execution order (in instructions)

- lw $2, 20($1)
- and $4, $2, $5
- or $8, $2, $6
- add $9, $4, $2
- slt $1, $6, $7

Need to stall for one cycle
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs, IF/ID.RegisterRt
- Load-use hazard when
  - ID/EX.MemRead and
    - ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
      (ID/EX.RegisterRt = IF/ID.RegisterRt))
- If detected, stall and insert bubble
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
- 1-cycle stall allows MEM to read data for \texttt{lw}
  - Can subsequently forward to EX stage
Stall/Bubble in the Pipeline

Program execution order (in instructions)

- `lw $2, 20($1)`
- `and` becomes `nop`
- `and $4, $2, $5`
- `or $8, $2, $6`
- `add $9, $4, $2`

Stall inserted here
Stall/Bubble in the Pipeline

Program execution order (in instructions)

Iw $2, 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...
Datapath with Hazard Detection
Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure
Branch Hazards

- If branch outcome determined in MEM

Program execution order (in instructions)

```
40 beq $1, $3, 28
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)
```

Flush these instructions (Set control values to 0)
Reducing Branch Delay

- Move hardware to determine outcome to ID stage
  - Target address adder
  - Register comparator
- Example: branch taken
  
  ```
  36: sub $10, $4, $8
  40: beq $1, $3, 7
  44: and $12, $2, $5
  48: or $13, $2, $6
  52: add $14, $4, $2
  56: slt $15, $6, $7
  ... 
  72: lw $4, 50($7)
  ```
Example: Branch Taken

and $12, $2, $5

beq $1, $3, 7

sub $10, $4, $8

before<1>

before<2>

Clock 3
Example: Branch Taken

\[ \text{lw} \quad S4, 50(S7) \]

\[ \text{Bubble (nop)} \]

\[ \text{beq} \quad S1, S3, 7 \]

\[ \text{sub} \quad S10, \ldots \]

before\(<1\> \]
Data Hazards for Branches

- If a comparison register is a destination of 2\textsuperscript{nd} or 3\textsuperscript{rd} 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} \\
\ldots & \quad \text{IF} \quad \text{ID} \quad \text{EX} \quad \text{MEM} \quad \text{WB} \\
\text{beq} & \ $1$, $4$, 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 2\textsuperscript{nd} 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
beq stalled
beq stalled
beq $1, $0, target
```
Dynamic Branch Prediction

- In deeper and superscalar pipelines, branch penalty is more significant
- Use dynamic prediction
  - Branch prediction buffer (aka branch history table)
  - Indexed by recent branch instruction addresses
  - Stores outcome (taken/not taken)
- To execute a branch
  - Check table, expect the same outcome
  - Start fetching from fall-through or target
  - If wrong, flush pipeline and flip prediction
1-Bit Predictor: Shortcoming

- Inner loop branches mispredicted twice!

outer: ...
    ...
inner: ...
    ...
    beq ..., ..., inner
    ...
    beq ..., ..., outer

- Mispredict as taken on last iteration of inner loop
- Then mispredict as not taken on first iteration of inner loop next time around
2-Bit Predictor

- Only change prediction on two successive mispredictions
Calculating the Branch Target

- Even with predictor, still need to calculate the target address
  - 1-cycle penalty for a taken branch

- Branch target buffer
  - Cache of target addresses
  - Indexed by PC when instruction fetched
    - If hit and instruction is branch predicted taken, can fetch target immediately
Exceptions and Interrupts

- “Unexpected” events requiring change in flow of control
  - Different ISAs use the terms differently

- Exception
  - Arises within the CPU
    - e.g., undefined opcode, overflow, syscall, …

- Interrupt
  - From an external I/O controller

- Dealing with them without sacrificing performance is hard
Handling Exceptions

- In MIPS, exceptions managed by a System Control Coprocessor (CP0)
- Save PC of offending (or interrupted) instruction
  - In MIPS: Exception Program Counter (EPC)
- Save indication of the problem
  - In MIPS: Cause register
  - We’ll assume 1-bit
    - 0 for undefined opcode, 1 for overflow
- Jump to handler at 8000 00180
Handler Actions

- Read cause, and transfer to relevant handler
- Determine action required
- If restartable
  - Take corrective action
  - use EPC to return to program
- Otherwise
  - Terminate program
  - Report error using EPC, cause, …
Exceptions in a Pipeline

- Another form of control hazard
- Consider overflow on add in EX stage
  
  ```
  add $1, $2, $1
  ```

  - Prevent $1 from being clobbered
  - Complete previous instructions
  - Flush add and subsequent instructions
  - Set Cause and EPC register values
  - Transfer control to handler

- Similar to mispredicted branch
  - Use much of the same hardware
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)
Scheduling Static Multiple Issue

- 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 with nop

<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>
MIPS with Static Dual Issue
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>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 Unrolling Example

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

<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><strong>nop</strong></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, $t4, $s2</td>
<td>sw $t1, 12($s1)</td>
<td>6</td>
</tr>
<tr>
<td><strong>nop</strong></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>
</tbody>
</table>
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
Speculation

- Predict branch and continue issuing
  - Don’t commit until branch outcome determined
- Load speculation
  - Avoid load and cache miss delay
    - Predict the effective address
    - Predict loaded value
  - Load before completing outstanding stores
  - Bypass stored values to load unit
  - Don’t commit load until speculation cleared
Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predicable
  - 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
- Some dependencies are hard to eliminate
  - e.g., pointer aliasing
- Some parallelism is hard to expose
  - Limited window size during instruction issue
- 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>
The Opteron X4 Microarchitecture

§4.11 Real Stuff: The AMD Opteron X4 (Barcelona) Pipeline

72 physical registers
The Opteron X4 Pipeline Flow

- For integer operations

- FP is 5 stages longer
- Up to 106 RISC-ops in progress

- Bottlenecks
  - Complex instructions with long dependencies
  - Branch mispredictions
  - Memory access delays
Fallacies

- Pipelining is easy (!)
  - The basic idea is easy
  - The devil is in the details
    - e.g., detecting data hazards

- Pipelining is independent of technology
  - So why haven’t we always done pipelining?
  - More transistors make more advanced techniques feasible
  - Pipeline-related ISA design needs to take account of technology trends
    - e.g., predicated instructions