Pipelining hazards, Parallel Data, Threads
Review

Software

- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”

- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

Hardware

Harness Parallelism & Achieve High Performance

- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”

- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads

- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

Hardware descriptions

All gates functioning in parallel at same time

Today’s Lecture

Logic Gate

Smart Phone

Warehouse Scale Computer

Computer

Core

…

Core

Memory

(Core)

Input/Output

Instruction Unit(s)

Functional Unit(s)

Main Memory

A₀+B₀A₁+B₁A₂+B₂A₃+B₃

- Functional Unit(s)
- Instruction Unit(s)
- Main Memory
- Input/Output
- Memory
- Core
- …
- Core
- Warehouse Scale Computer
- Smart Phone
- Today’s Lecture
- Logic Gate
Control Path
Pipelined Control
Hazards

Situations that prevent starting the next logical instruction in the next clock cycle

1. Structural hazards
   – Required resource is busy (e.g., roommate studying)

2. Data hazard
   – Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)

3. Control hazard
   – Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)
3. 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

• BEQ, BNE in MIPS pipeline

• Simple solution Option 1: *Stall* on every branch until have new PC value
  – Would add 2 bubbles/clock cycles for every Branch! (~ 20% of instructions executed)
Stall => 2 Bubbles/Clocks

Where do we do the compare for the branch?
Control Hazard: Branching

• Optimization #1:
  – Insert special branch comparator in Stage 2
  – As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  – Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  – Side Note: means that branches are idle in Stages 3, 4 and 5

Question: What’s an efficient way to implement the equality comparison?
One Clock Cycle Stall

Time (clock cycles)

Instr. Order

Instr 1

Instr 2

Instr 3

Instr 4

Branch comparator moved to Decode stage.

beq

Instr 4
Control Hazards: Branching

• Option 2: *Predict* outcome of a branch, fix up if guess wrong
  – Must cancel all instructions in pipeline that depended on guess that was wrong
  – This is called “flushing” the pipeline

• Simplest hardware if we predict that all branches are NOT taken
  – Why?
Control Hazards: Branching

• Option #3: Redefine branches
  – Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  – New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the branch-delay slot)

• Delayed Branch means we always execute inst after branch

• This optimization is used with MIPS
Example: Nondelayed vs. Delayed Branch

Nondelayed Branch

\[
\begin{align*}
&\text{or } \$8, \$9, \$10 \\
&\text{add } \$1, \$2, \$3 \\
&\text{sub } \$4, \$5, \$6 \\
&\text{beq } \$1, \$4, \text{Exit} \\
&\text{xor } \$10, \$1, \$11
\end{align*}
\]

Exit:

Delayed Branch

\[
\begin{align*}
&\text{add } \$1, \$2, \$3 \\
&\text{sub } \$4, \$5, \$6 \\
&\text{beq } \$1, \$4, \text{Exit} \\
&\text{or } \$8, \$9, \$10 \\
&\text{xor } \$10, \$1, \$11
\end{align*}
\]

Exit:
Control Hazards: Branching

• Notes on Branch-Delay Slot
  – Worst-Case Scenario: put a no-op in the branch-delay slot
  – Better Case: place some instruction preceding the branch in the branch-delay slot—as long as the changed doesn’t affect the logic of program
    • Re-ordering instructions is common way to speed up programs
    • Compiler usually finds such an instruction 50% of time
    • Jumps also have a delay slot ...
Greater Instruction-Level Parallelism (ILP)

• Deeper pipeline (5 => 10 => 15 stages)
  – Less work per stage ⇒ shorter clock cycle
• Multiple issue “superscalar”
  – Replicate pipeline stages ⇒ multiple pipelines
  – Start multiple instructions per clock cycle
  – CPI < 1, so use Instructions Per Cycle (IPC)
  – E.g., 4GHz 4-way multiple-issue
    • 16 BIPS, peak CPI = 0.25, peak IPC = 4
  – But dependencies reduce this in practice
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
Superscalar Laundry: Parallel per stage

- More resources, HW to match mix of parallel tasks?

<table>
<thead>
<tr>
<th>Time</th>
<th>6 PM</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>1</th>
<th>2 AM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Order</td>
<td>(light clothing)</td>
<td>(dark clothing)</td>
<td>(very dirty clothing)</td>
<td>(light clothing)</td>
<td>(dark clothing)</td>
<td>(very dirty clothing)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Pipeline Depth and Issue Width

### Intel Processors over Time

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
<tr>
<td>Core 2 Conroe</td>
<td>2006</td>
<td>2930 MHz</td>
<td>14</td>
<td>4</td>
<td>2</td>
<td>75W</td>
</tr>
<tr>
<td>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
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
Scheduling Static Multiple Issue

• Compiler must remove some/all hazards
  – Reorder instructions into issue packets
  – No dependencies within a packet
  – Possibly some dependencies between packets
    • Varies between ISAs; compiler must know!
  – Pad issue packet 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>
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>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
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>nop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lw  $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Scheduling Example

• Schedule this for dual-issue MIPS

```plaintext
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></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
# Scheduling Example

- Schedule this for dual-issue MIPS

```plaintext
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></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
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 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>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, $t4, $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>
</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

  \[
  \begin{align*}
  &\text{lw} \quad \text{\$t0, 20(\$s2)} \\
  &\text{addu} \quad \text{\$t1, \$t0, \$t2} \\
  &\text{subu} \quad \text{\$s4, \$s4, \$t3} \\
  &\text{slti} \quad \text{\$t5, \$s4, 20}
  \end{align*}
  \]
  – Can start \text{subu} while \text{addu} is waiting for \text{lw}
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
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 (Branch Prediction)
    • Roll back if path taken is different
  – Speculate on load
    • Roll back if location is updated
Pipeline Hazard: Matching socks in later load

- A depends on D; stall since folder tied up;
Out-of-Order Laundry: Don’t Wait

- A depends on D; rest continue; need more resources to allow out-of-order
# Out Of Order Intel

- All use OOO since 2001

<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>Core 2 Yorkfield</td>
<td>2008</td>
<td>2930 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>4</td>
<td>95W</td>
</tr>
<tr>
<td>Core i7 Gulftown</td>
<td>2010</td>
<td>3460 MHz</td>
<td>16</td>
<td>4</td>
<td>Yes</td>
<td>6</td>
<td>130W</td>
</tr>
</tbody>
</table>
“And in Conclusion..”

• Pipelining is an important form of ILP
• Challenge is (are?) hazards
  – Forwarding helps w/many data hazards
  – Delayed branch helps with control hazard in 5 stage pipeline
  – Load delay slot / interlock necessary
• More aggressive performance:
  – Longer pipelines
  – Superscalar
  – Out-of-order execution
  – Speculation
The Flynn Taxonomy,
Intel SIMD Instructions
Great Idea #4: Parallelism

- **Parallel Requests**
  Assigned to computer
  e.g. search “Garcia”

- **Parallel Threads**
  Assigned to core
  e.g. lookup, ads

- **Parallel Instructions**
  > 1 instruction @ one time
  e.g. 5 pipelined instructions

- **Parallel Data**
  > 1 data item @ one time
  e.g. add of 4 pairs of words

- **Hardware descriptions**
  All gates functioning in parallel at same time

Software

Hardware

Warehouse
Scale
Computer

Leverage Parallelism & Achieve High Performance

Computer

Core

... Core

Memory

Input/Output

Instruction Unit(s)

Functional
Unit(s)

A₀+B₀ A₁+B₁ A₂+B₂ A₃+B₃

Cache Memory

Logic Gates

Smart Phone

We are here
Agenda

• Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Loop Unrolling
Hardware vs. Software Parallelism

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>Serial</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Matrix Multiply written in MatLab running on an Intel Pentium 4</td>
<td>Windows Vista Operating System running on an Intel Pentium 4</td>
<td></td>
</tr>
<tr>
<td>Parallel</td>
<td>Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)</td>
<td>Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)</td>
<td></td>
</tr>
</tbody>
</table>

- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
- *Flynn's Taxonomy* is for parallel hardware
Flynn’s Taxonomy

### Data Streams

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD: Intel Pentium 4</td>
<td>SIMD: SSE instructions of x86</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: No examples today</td>
<td>MIMD: Intel Xeon e5345 (Clovertown)</td>
</tr>
</tbody>
</table>

- **SIMD and MIMD most commonly encountered today**
- **Most common parallel processing programming style:** Single Program Multiple Data (“SPMD”)
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions (will see later in Thread Level Parallelism)
- **SIMD:** specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)
Single Instruction/Single Data Stream

- Sequential computer that exploits no parallelism in either the instruction or data streams
- Examples of SISD architecture are traditional uniprocessor machines
Multiple Instruction/Single Data Stream

- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)

- MISD no longer commonly encountered, mainly of historical interest only
Single Instruction/Multiple Data Stream

- Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)
Multiple Instruction/Multiple Data Stream

- Multiple autonomous processors simultaneously executing different instructions on different data
- MIMD architectures include multicore and Warehouse Scale Computers
Agenda

• Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Loop Unrolling
Agenda

• Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Loop Unrolling
SIMD Architectures

• **Data-Level Parallelism (DLP):** Executing one operation on multiple data streams

• **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

\[ y[i] := c[i] \times x[i], \quad 0 \leq i < n \]

• **Sources of performance improvement:**
  – One instruction is fetched & decoded for entire operation
  – Multiplications are known to be independent
  – Pipelining/concurrency in memory access as well
“Advanced Digital Media Boost”

- To improve performance, Intel’s SIMD instructions
  - Fetch one instruction, do the work of multiple instructions
  - MMX (MultiMedia eXtension, Pentium II processor family)
  - SSE (Streaming SIMD Extension, Pentium III and beyond)
Example: SIMD Array Processing

for each $f$ in array:
    $f = \sqrt{f}$

for each $f$ in array {
    load $f$ to the floating-point register
    calculate the square root
    write the result from the register to memory
}

for every 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
SSE Instruction Categories for Multimedia Support

<table>
<thead>
<tr>
<th>Instruction category</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned add/subtract</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Saturating add/subtract</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Max/min/minimum</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Average</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Shift right/left</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
</tbody>
</table>

- Intel processors are **CISC (complicated instrs)**
- SSE-2+ supports wider data types to allow $16 \times 8$-bit and $8 \times 16$-bit operands
Intel Architecture SSE2+ 128-Bit SIMD Data Types

- Note: in Intel Architecture (unlike MIPS) a word is 16 bits
  - Single precision FP: Double word (32 bits)
  - Double precision FP: Quad word (64 bits)
XMM Registers

- Architecture extended with eight 128-bit data registers
  - 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
  - e.g. 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>127</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XMM0</td>
<td></td>
</tr>
</tbody>
</table>
SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>MOV {H/L} {PS/PD} xmm, mem/xmm</td>
<td>MUL{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

{SS} Scalar Single precision FP: 1 32-bit operand in a 128-bit register
{PS} Packed Single precision FP: 4 32-bit operands in a 128-bit register
{SD} Scalar Double precision FP: 1 64-bit operand in a 128-bit register
{PD} Packed Double precision FP, or 2 64-bit operands in a 128-bit register
## SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>MOV {H/L} {PS/PD} xmm, mem/xmm</td>
<td>MUL{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

- **xmm**: one operand is a 128-bit SSE2 register
- **mem/xmm**: other operand is in memory or an SSE2 register
- **{A}**: 128-bit operand is aligned in memory
- **{U}**: means the 128-bit operand is unaligned in memory
- **{H}**: means move the high half of the 128-bit operand
- **{L}**: means move the low half of the 128-bit operand
Example: Add Single Precision FP Vectors

Computation to be performed:

```
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

**SSE Instruction Sequence:**

```
movaps address-of-v1, %xmm0
  // v1.w | v1.z | v1.y | v1.x -> xmm0
addps  address-of-v2, %xmm0
  // v1.w+v2.w | v1.z+v2.z | v1.y+v2.y | v1.x+v2.x
     -> xmm0
movaps %xmm0, address-of-vec_res
```

move from mem to XMM register, memory aligned, packed single precision

add from mem to XMM register, packed single precision

move from XMM register to mem, memory aligned, packed single precision
Example: Image Converter (1/5)

- Converts BMP (bitmap) image to a YUV (color space) image format:
  - Read individual pixels from the BMP image, convert pixels into YUV format
  - Can pack the pixels and operate on a set of pixels with a single instruction

- Bitmap image consists of 8-bit monochrome pixels
  - By packing these pixel values in a 128-bit register, we can operate on $128/8 = 16$ values at a time
  - Significant performance boost
Example: Image Converter (2/5)

- FMADDPS – Multiply and add packed single precision floating point instruction
- One of the typical operations computed in transformations (e.g. DFT or FFT)

\[ P = \sum_{n=1}^{N} f(n) \times x(n) \]
Example: Image Converter (3/5)

- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):

  ```c
  for (int i = 0; i < 4; i++)
      p = p + src1[i] * src2[i];
  ``

1) Regular x86 instructions for the inner loop:
   ```
   fmul [...]  
   faddp [...]  
   ```
   - Instructions executed: 4 * 2 = 8 (x86)
Example: Image Converter (4/5)

- FP numbers f(n) and x(n) in src1 and src2; p in dest;
- C implementation for N = 4 (128 bits):

```c
for (int i = 0; i < 4; i++)
    p = p + src1[i] * src2[i];
```

2) SSE2 instructions for the inner loop:

```c
// xmm0=p, xmm1=src1[i], xmm2=src2[i]
mulps %xmm1,%xmm2 // xmm2 * xmm1 -> xmm2
addps %xmm2,%xmm0 // xmm0 + xmm2 -> xmm0
```

- Instructions executed: 2 (SSE2)
Example: Image Converter (5/5)

- FP numbers $f(n)$ and $x(n)$ in src1 and src2; $p$ in dest;
- C implementation for $N = 4$ (128 bits):

\[
\text{for (int } i = 0; i < 4; i++) \\
\quad p = p + \text{src1}[i] \times \text{src2}[i];
\]

3) SSE5 accomplishes the same in \textbf{one} instruction:

\[\text{fmaddps} \ %\text{xmm0}, \ %\text{xmm1}, \ %\text{xmm2}, \ %\text{xmm0}\]
\[
\quad \text{// xmm2 } \times \text{ xmm1 } + \text{ xmm0 } \rightarrow \text{ xmm0}
\]
\[
\quad \text{// multiply xmm1 x xmm2 packed single,}
\]
\[
\quad \text{// then add product packed single to sum in xmm0}
\]
Agenda

• Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Loop Unrolling
Data Level Parallelism and SIMD

• SIMD wants adjacent values in memory that can be operated in parallel.

• Usually specified in programs as loops:

\[
\text{for}(i=0; \ i<1000; \ i++) \\
\quad x[i] = x[i] + s;
\]

• How can we reveal more data level parallelism than is available in a single iteration of a loop?
  
  – *Unroll the loop* and adjust iteration rate.
Looping in MIPS

Assumptions:
- $s0 \rightarrow$ initial address (beginning of array)
- $s1 \rightarrow$ scalar value s
- $s2 \rightarrow$ termination address (end of array)

Loop:

```
lw   $t0,0($s0)
addu $t0,$t0,$s1     # add s to array element
sw   $t0,0($s0)       # store result
addiu $s0,$s0,4      # move to next element
bne  $s0,$s2,Loop     # repeat Loop if not done
```
Loop Unrolled

Loop:
```
lw    $t0,0($s0)
addu  $t0,$t0,$s1
sw    $t0,0($s0)
lw    $t1,4($s0)
addu  $t1,$t1,$s1
sw    $t1,4($s0)
lw    $t2,8($s0)
addu  $t2,$t2,$s1
sw    $t2,8($s0)
lw    $t3,12($s0)
addu  $t3,$t3,$s1
sw    $t3,12($s0)
addiu $s0,$s0,16
bne   $s0,$s2,Loop
```

NOTE:

1. Using different registers eliminate stalls

2. Loop overhead encountered only once every 4 data iterations

3. This unrolling works if
   \[ \text{loop\_limit \ mod \ 4} = 0 \]
Loop Unrolled Scheduled

Note: We just switched from integer instructions to single-precision FP instructions!

Loop:

\[
\begin{align*}
\text{lwc1} & \quad \text{\$t0,0($s0)} \\
\text{lwc1} & \quad \text{\$t1,4($s0)} \\
\text{lwc1} & \quad \text{\$t2,8($s0)} \\
\text{lwc1} & \quad \text{\$t3,12($s0)} \\
\text{add.s} & \quad \text{\$t0,$t0,$s1} \\
\text{add.s} & \quad \text{\$t1,$t1,$s1} \\
\text{add.s} & \quad \text{\$t2,$t2,$s1} \\
\text{add.s} & \quad \text{\$t3,$t3,$s1} \\
\text{swc1} & \quad \text{\$t0,0($s0)} \\
\text{swc1} & \quad \text{\$t1,4($s0)} \\
\text{swc1} & \quad \text{\$t2,8($s0)} \\
\text{swc1} & \quad \text{\$t3,12($s0)} \\
\text{addiu} & \quad \text{\$s0,$s0,16} \\
\text{bne} & \quad \text{\$s0,$s2,Loop}
\end{align*}
\]

4 Loads side-by-side:  
Could replace with 4 wide SIMD Load

4 Adds side-by-side:  
Could replace with 4 wide SIMD Add

4 Stores side-by-side:  
Could replace with 4 wide SIMD Store
Loop Unrolling in C

• Instead of compiler doing loop unrolling, could do it yourself in C:

```c
for(i=0; i<1000; i++)
    x[i] = x[i] + s;
```

```c
for(i=0; i<1000; i=i+4) {
    x[i]   = x[i]   + s;
    x[i+1] = x[i+1] + s;
    x[i+2] = x[i+2] + s;
    x[i+3] = x[i+3] + s;
}
```

What is downside of doing this in C?
Generalizing Loop Unrolling

• Take a loop of \( n \) iterations and perform a \( k \)-fold unrolling of the body of the loop:
  – First run the loop with \( k \) copies of the body \( \text{floor}(n/k) \) times
  – To finish leftovers, then run the loop with 1 copy of the body \( n \mod k \) times
Review

• Flynn Taxonomy of Parallel Architectures
  – **SIMD**: *Single Instruction Multiple Data*
  – **MIMD**: *Multiple Instruction Multiple Data*
  – **SISD**: Single Instruction Single Data
  – **MISD**: Multiple Instruction Single Data (unused)

• Intel SSE SIMD Instructions
  – One instruction fetch that operates on multiple operands simultaneously
  – 64/128 bit XMM registers
  – (SSE = *Streaming SIMD Extensions*)

• Threads and Thread-level parallelism
Intel SSE Intrinsics

• Intrinsics are C functions and procedures for putting in assembly language, including SSE instructions
  – With intrinsics, can program using these instructions indirectly
  – One-to-one correspondence between SSE instructions and intrinsics
Example SSE Intrinsics

Intrinsics:

- Vector data type: 
  \_m128d

- Load and store operations:
  \_mm_load_pd
  MOVAPD/aligned, packed double
  \_mm_store_pd
  MOVAPD/aligned, packed double
  \_mm_loadu_pd
  MOVUPD/unaligned, packed double
  \_mm_storeu_pd
  MOVUPD/unaligned, packed double

- Load and broadcast across vector
  \_mm_load1_pd
  MOVSD + shuffling/duplicating

- Arithmetic:
  \_mm_add_pd
  ADDPD/add, packed double
  \_mm_mul_pd
  MULPD/multiple, packed double

Corresponding SSE instructions:
Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[ C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j} \]

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix}
= 
\begin{bmatrix}
C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
= 
\begin{bmatrix}
C_{1,1} = 1*1 + 0*2 = 1 & C_{1,2} = 1*3 + 0*4 = 3 \\
C_{2,1} = 0*1 + 1*2 = 2 & C_{2,2} = 0*3 + 1*4 = 4
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply

• Using the XMM registers
  – 64-bit/Double precision/two doubles per XMM reg

\[
\begin{array}{ccc}
C_1 & C_{1,1} & C_{1,2} \\
C_2 & C_{2,1} & C_{2,2} \\
\end{array}
\]

\[
\begin{array}{ccc}
A & A_{1,i} & A_{2,i} \\
\end{array}
\]

\[
\begin{array}{ccc}
B_1 & B_{i,1} & B_{i,1} \\
B_2 & B_{i,2} & B_{i,2} \\
\end{array}
\]

Stored in memory in Column order
Example: 2 x 2 Matrix Multiply

- Initialization

<table>
<thead>
<tr>
<th></th>
<th>C1</th>
<th>C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example: 2 x 2 Matrix Multiply

• Initialization

<table>
<thead>
<tr>
<th></th>
<th>C_1</th>
<th></th>
<th>C_2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

• l = 1

_A_1,1   _A_2,1

_mm_load_pd: Load 2 doubles into XMM reg, Stored in memory in Column order

_B_1,1   _B_1,1

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
### Example: 2 x 2 Matrix Multiply

- **First iteration intermediate result**

<table>
<thead>
<tr>
<th></th>
<th>(0 + A_{1,1}B_{1,1})</th>
<th>(0 + A_{2,1}B_{1,1})</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₁</td>
<td>(0 + A_{1,1}B_{1,2})</td>
<td>(0 + A_{2,1}B_{1,2})</td>
</tr>
</tbody>
</table>

\[
c1 = \texttt{_mm_add_pd}(c1, \texttt{_mm_mul_pd}(a, b1));
\]
\[
c2 = \texttt{_mm_add_pd}(c2, \texttt{_mm_mul_pd}(a, b2));
\]

SSE instructions first do parallel multiplies and then parallel adds in XMM registers.

- **I = 1**

<table>
<thead>
<tr>
<th></th>
<th>A₁,₁</th>
<th>A₂,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  _mm_load_pd: Stored in memory in Column order.

<table>
<thead>
<tr>
<th></th>
<th>B₁,₁</th>
<th>B₁,₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B₁,₂</th>
<th>B₁,₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₂</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  _mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
Example: 2 x 2 Matrix Multiply

• First iteration intermediate result

<table>
<thead>
<tr>
<th></th>
<th>0+A_{1,1}B_{1,1}</th>
<th>0+A_{2,1}B_{1,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C₂</td>
<td>0+A_{1,1}B_{1,2}</td>
<td>0+A_{2,1}B_{1,2}</td>
</tr>
</tbody>
</table>

c1 = _mm_add_pd(c1, _mm_mul_pd(a, b1));
c2 = _mm_add_pd(c2, _mm_mul_pd(a, b2));
SSE instructions first do parallel multiplies and then parallel adds in XMM registers

• I = 2

<table>
<thead>
<tr>
<th></th>
<th>A_{1,2}</th>
<th>A_{2,2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

_mm_load_pd: Stored in memory in Column order

<table>
<thead>
<tr>
<th></th>
<th>B_{2,1}</th>
<th>B_{2,1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>B₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B₂</td>
<td>B_{2,2}</td>
<td>B_{2,2}</td>
</tr>
</tbody>
</table>

_mm_load1_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
Example: 2 x 2 Matrix Multiply

- Second iteration intermediate result

\[
\begin{array}{c|c}
C_{1,1} & C_{2,1} \\
\hline
A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
\hline
A_{1,1}B_{1,2} + A_{1,2}B_{2,2} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \\
\end{array}
\]

\[
c1 = \_mm\_add\_pd(c1, \_mm\_mul\_pd(a,b1));
c2 = \_mm\_add\_pd(c2, \_mm\_mul\_pd(a,b2));
\]

SSE instructions first do parallel multiplies and then parallel adds in XMM registers

- \( l = 2 \)

\[
\begin{array}{c|c}
A_{1,2} & A_{2,2} \\
\end{array}
\]

\_mm\_load\_pd: Stored in memory in Column order

\[
\begin{array}{c|c}
B_{2,1} & B_{2,1} \\
B_{2,2} & B_{2,2} \\
\end{array}
\]

\_mm\_load1\_pd: SSE instruction that loads a double word and stores it in the high and low double words of the XMM register (duplicates value in both halves of XMM)
Example: 2 x 2 Matrix Multiply

Definition of Matrix Multiply:

\[ C_{i,j} = (A \times B)_{i,j} = \sum_{k=1}^{2} A_{i,k} \times B_{k,j} \]

\[
\begin{bmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{bmatrix}
\times
\begin{bmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{bmatrix}
= \begin{bmatrix}
C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\times
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
= \begin{bmatrix}
C_{1,1} = 1\times1 + 0\times2 = 1 & C_{1,2} = 1\times3 + 0\times4 = 3 \\
C_{2,1} = 0\times1 + 1\times2 = 2 & C_{2,2} = 0\times3 + 1\times4 = 4
\end{bmatrix}
\]
Example: 2 x 2 Matrix Multiply
(Part 1 of 2)

#include <stdio.h>
// header file for SSE compiler intrinsics
#include <emmintrin.h>

// NOTE: vector registers will be represented in comments as v1 = [ a | b]
// where v1 is a variable of type __m128d and a, b are doubles

int main(void) {
    // allocate A,B,C aligned on 16-byte boundaries
    double B[4] __attribute__((aligned (16)));
    double C[4] __attribute__((aligned (16)));
    int lda = 2;
    int i = 0;
    // declare several 128-bit vector variables
    __m128d c1,c2,a,b1,b2;

    // Initialize A, B, C for example
    /* A =
        (note column order!)
        1 0
        0 1
     */

    /* B =
        (note column order!)
        1 3
        2 4
     */
    B[0] = 1.0; B[1] = 2.0; B[2] = 3.0; B[3] = 4.0;

    /* C =
        (note column order!)
        0 0
        0 0
     */
    C[0] = 0.0; C[1] = 0.0; C[2] = 0.0; C[3] = 0.0;
// used aligned loads to set
// c1 = [c_11 | c_21]
c1 = _mm_load_pd(C+0*lda);
// c2 = [c_12 | c_22]
c2 = _mm_load_pd(C+1*lda);

for (i = 0; i < 2; i++) {
    /* a =
       i = 0: [a_11 | a_21]
       i = 1: [a_12 | a_22]
    */
a = _mm_load_pd(A+i*lda);
    /* b1 =
       i = 0: [b_11 | b_11]
       i = 1: [b_21 | b_21]
    */
b1 = _mm_load1_pd(B+i+0*lda);
    /* b2 =
       i = 0: [b_12 | b_12]
       i = 1: [b_22 | b_22]
    */
b2 = _mm_load1_pd(B+i+1*lda);
    /* c1 =
       i = 0: [c_11 + a_11*b_11 | c_21 + a_21*b_11]
       i = 1: [c_11 + a_21*b_21 | c_21 + a_22*b_21]
    */
c1 = _mm_add_pd(c1,_mm_mul_pd(a,b1));
    /* c2 =
       i = 0: [c_12 + a_11*b_12 | c_22 + a_21*b_12]
       i = 1: [c_12 + a_21*b_22 | c_22 + a_22*b_22]
    */
c2 = _mm_add_pd(c2,_mm_mul_pd(a,b2));
}

// store c1,c2 back into C for completion
_mm_store_pd(C+0*lda,c1);
_mm_store_pd(C+1*lda,c2);

// print C
printf("%g,%g\n%g,%g\n",C[0],C[2],C[1],C[3]);
return 0;
Inner loop from gcc –O -S

L2: movapd (%rax,%rsi), %xmm1 //Load aligned A[i,i+1]->m1
    movddup (%rdx), %xmm0   //Load B[j], duplicate->m0
    mulpd %xmm1, %xmm0     //Multiply m0*m1->m0
    addpd %xmm0, %xmm3     //Add m0+m3->m3
    movddup 16(%rdx), %xmm0 //Load B[j+1], duplicate->m0
    mulpd %xmm0, %xmm1     //Multiply m0*m1->m1
    addpd %xmm1, %xmm2     //Add m1+m2->m2
    addq $16, %rax         // rax+16 -> rax (i+=2)
    addq $8, %rdx          // rdx+8 -> rdx (j+=1)
    cmpq $32, %rax         // rax == 32?
    jne L2                 // jump to L2 if not equal
    movapd %xmm3, (%rcx)   //store aligned m3 into C[k,k+1]
    movapd %xmm2, (%rdi)   //store aligned m2 into C[l,l+1]
You Are Here!

**Software**

- Parallel Requests
  Assigned to computer
  e.g., Search “Katz”

- Parallel Threads
  Assigned to core
  e.g., Lookup, Ads

**Hardware**

- Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

- Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- Hardware descriptions
  All gates functioning in parallel at same time

Harness Parallelism & Achieve High Performance
Background: Threads

- A *Thread* stands for “thread of execution”, is a single stream of instructions
  - A program / process can *split*, or *fork* itself into separate threads, which can (in theory) execute simultaneously.
  - An easy way to describe/think about parallelism

- A single CPU can execute many threads by *Time Division Multiplexing*

- **Multithreading** is running multiple threads through the same hardware
Parallel Processing:
Multiprocessor Systems (MIMD)

- Multiprocessor (MIMD): a computer system with at least 2 processors

1. Deliver high throughput for independent jobs via job-level parallelism
2. Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel processing program

Now use term **core** for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant
Transition to Multicore

parallel_app

sequential_app

transistors

number_of_cores

frequency

typical_power

Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond
Multiprocessors and You

• Only path to performance is parallelism
  – Clock rates flat or declining
  – SIMD: 2X width every 3-4 years
    • 128b wide now, 256b 2011, 512b in 2014?, 1024b in 2018?
    • Advanced Vector Extensions are 256-bits wide!
  – MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, ...

• A key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  – Scheduling, load balancing, time for synchronization, overhead for communication
<table>
<thead>
<tr>
<th>Year</th>
<th>Cores</th>
<th>SIMD bits /Core</th>
<th>Core * SIMD bits</th>
<th>Peak DP FLOPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>2</td>
<td>128</td>
<td>256</td>
<td>4</td>
</tr>
<tr>
<td>2005</td>
<td>4</td>
<td>128</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>2007</td>
<td>6</td>
<td>128</td>
<td>768</td>
<td>12</td>
</tr>
<tr>
<td>2009</td>
<td>8</td>
<td>128</td>
<td>1024</td>
<td>16</td>
</tr>
<tr>
<td>2011</td>
<td>10</td>
<td>256</td>
<td>2560</td>
<td>40</td>
</tr>
<tr>
<td>2013</td>
<td>12</td>
<td>256</td>
<td>3072</td>
<td>48</td>
</tr>
<tr>
<td>2015</td>
<td>14</td>
<td>512</td>
<td>7168</td>
<td>112</td>
</tr>
<tr>
<td>2017</td>
<td>16</td>
<td>512</td>
<td>8192</td>
<td>128</td>
</tr>
<tr>
<td>2019</td>
<td>18</td>
<td>1024</td>
<td>18432</td>
<td>288</td>
</tr>
<tr>
<td>2021</td>
<td>20</td>
<td>1024</td>
<td>20480</td>
<td>320</td>
</tr>
</tbody>
</table>