Agenda for Today & Next Few Lectures

- Single-cycle Microarchitectures
- Multi-cycle and Microprogrammed Microarchitectures
- Pipelining
- Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...
- Out-of-Order Execution
- Issues in OoO Execution: Load-Store Handling, ...
- Alternative Approaches to Instruction Level Parallelism
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- **SIMD Processing (Vector and array processors, GPUs)**
- VLIW
- Decoupled Access Execute
- Systolic Arrays
Reminder: Announcements

- Lab 3 due this Friday (Feb 20)
  - Pipelined MIPS
  - Competition for high performance
    - You can optimize both cycle time and CPI
    - Document and clearly describe what you do during check-off

- Homework 3 due Feb 25
  - A lot of questions that enable you to learn the concepts via hands-on exercise
  - Remember this is all for your benefit (to learn and prepare for exams)
    - HWs have very little contribution to overall grade
    - Solutions to almost all questions are online anyway
    - But I would still like you to do them (for your benefit)!
Homework 3.1: Feedback Form

- Due Monday Feb 23

- I would like your feedback on the course
- Easy to fill in
- Can submit anonymously, if you wish
- Worth 0.25% of your grade
- Need to get checked off after submitting to get your grade points
  - Can email
  - If anonymous, show that you are turning in and have a TA check you off
Readings for Today


Recap of Last Lecture

- OoO Execution as Restricted Data Flow
- Memory Disambiguation or Unknown Address Problem
- Memory Dependence Handling
  - Conservative, Aggressive, Intelligent Approaches
- Load Store Queues
- Design Choices in an OoO Processor
- Combining OoO+Superscalar+Branch Prediction
- Example OoO Processor Designs

- Data Flow (at the ISA level) Approach to Concurrency
  - Characteristics
  - Supporting dynamic instances of a node: Tagging, Context IDs, Frames
  - Example Operation
  - Advantages and Disadvantages
  - Combining Data Flow and Control Flow: Getting the Best of Both Worlds
Reminder: Intel Pentium 4 Simplified

Figure 2. Stages of the Alpha 21264 instruction pipeline.

Review: Data Flow: Exploiting Irregular Parallelism
Review: Pure Data Flow Pros and Cons

- Advantages
  - Very good at exploiting irregular parallelism
  - Only real dependencies constrain processing

- Disadvantages
  - Debugging difficult (no precise state)
    - Interrupt/exception handling is difficult (what is precise state semantics?)
  - Implementing dynamic data structures difficult in pure data flow models
  - Too much parallelism? (Parallelism control needed)
  - High bookkeeping overhead (tag matching, data storage)
  - Instruction cycle is inefficient (delay between dependent instructions), memory locality is not exploited
Can we get the best of both worlds?

Two possibilities

- Model 1: Keep control flow at the ISA level, do dataflow underneath, preserving sequential semantics
- Model 2: Keep dataflow model, but incorporate some control flow at the ISA level to improve efficiency, exploit locality, and ease resource management
  - Incorporate threads into dataflow: statically ordered instructions; when the first instruction is fired, the remaining instructions execute without interruption in control flow order (e.g., one can pipeline them)
Review: Data Flow Summary

- Data Flow at the ISA level has not been (as) successful

- Data Flow implementations under the hood (while preserving sequential ISA semantics) have been very successful
  - Out of order execution
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- **SIMD Processing** (Vector and array processors, GPUs)
- VLIW
- Decoupled Access Execute
- Systolic Arrays
SIMD Processing: Exploiting Regular (Data) Parallelism
Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
Data Parallelism

- Concurrency arises from performing the **same operations on different pieces of data**
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors

- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)

- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel

- SIMD exploits instruction-level parallelism
  - Multiple “instructions” (more appropriately, operations) are concurrent: instructions happen to be the same
SIMD Processing

- Single instruction operates on multiple data elements
  - In time or in space
- Multiple processing elements

- Time-space duality

  - **Array processor**: Instruction operates on multiple data elements at the same time using different spaces

  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps using the same space
Array vs. Vector Processors

**ARRAY PROCESSOR**

<table>
<thead>
<tr>
<th>Time</th>
<th>LD0</th>
<th>LD1</th>
<th>LD2</th>
<th>LD3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD0</td>
<td>AD1</td>
<td>AD2</td>
<td>AD3</td>
<td></td>
</tr>
<tr>
<td>MU0</td>
<td>MU1</td>
<td>MU2</td>
<td>MU3</td>
<td></td>
</tr>
<tr>
<td>ST0</td>
<td>ST1</td>
<td>ST2</td>
<td>ST3</td>
<td></td>
</tr>
</tbody>
</table>

**VECTOR PROCESSOR**

<table>
<thead>
<tr>
<th>Space</th>
<th>LD</th>
<th>ADD</th>
<th>MUL</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD1</td>
<td>AD0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD2</td>
<td>AD1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD3</td>
<td>AD2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AD3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MU3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Same op @ same time**
- **Different ops @ same space**
- **Different ops @ time**
- **Same op @ space**
SIMD Array Processing vs. VLIW

- **VLIW**: Multiple independent operations packed together by the compiler
SIMD Array Processing vs. VLIW

- Array processor: Single operation on multiple (different) data elements

Instruction Execution:
- add VR[0],VR[0],1
- add VR[1],VR[1],1
- add VR[2],VR[2],1
- add VR[3],VR[3],1

Program Counter -> add VR, VR, 1
VLEN = 4
Vector Processors

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors
  
  ```
  for (i = 0; i<=49; i++)
      C[i] = (A[i] + B[i]) / 2
  ```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values

- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
  - Stride: distance between two elements of a vector
Vector Processors (II)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element

- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking within a vector
  - No control flow within a vector
  - Known stride allows prefetching of vectors into registers/cache/memory
Vector Processor Advantages

+ No dependencies within a vector
  - Pipelining, parallelization work well
  - Can have very deep pipelines, no dependencies!

+ Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth requirements

+ Highly regular memory access pattern
  - Can interleave vector data elements across multiple memory banks for higher memory bandwidth (to tolerate memory bank access latency)
  - Prefetching a vector is relatively easy

+ No need to explicitly code loops
  - Fewer branches in the instruction sequence
Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)
  ++ Vector operations
-- Very inefficient if parallelism is irregular
  -- How about searching for a key in a linked list?

To program a vector machine, the compiler or hand coder must make the data structures in the code fit nearly exactly the regular structure built into the hardware. That’s hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtleties of the application area. Often the rewriting is

Vector Processor Limitations

-- Memory (bandwidth) can easily become a bottleneck, especially if
  1. compute/memory operation balance is not maintained
  2. data is not mapped appropriately to memory banks
Vector Processing in More Depth
Vector Registers

- Each **vector data register** holds $N \times M$-bit values
- **Vector control registers**: VLEN, VSTR, VMASK
- Maximum VLEN can be $N$
  - Maximum number of elements stored in a vector register
- **Vector Mask Register** (VMASK)
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g., $VMASK[i] = (V_k[i] == 0)$
Vector Functional Units

- Use deep pipeline to execute element operations → fast clock cycle

- Control of deep pipeline is simple because elements in vector are independent

\[ V_1 \times V_2 \rightarrow V_3 \]

*Six stage multiply pipeline*
Vector Machine Organization (CRAY-1)

- CRAY-1

- Scalar and vector modes
- 8 64-element vector registers
- 64 bits per element
- **16 memory banks**
- 8 64-bit scalar registers
- 8 24-bit address registers
Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements

- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now

- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle

- Question: How do we achieve this with a memory that takes more than 1 cycle to access?

- Answer: Bank the memory; interleave the elements across banks
Memory Banking

- Memory is divided into banks that can be accessed independently; banks share address and data buses (to minimize pin cost).
- Can start and complete one bank access per cycle.
- Can sustain N parallel accesses if all N go to different banks.

![Diagram showing Memory Banking with banks and MDR/MAR connections to CPU and data/address buses.](Picture credit: Derek Chiou)
Vector Memory System

- Next address = Previous address + Stride
- If stride = 1 & consecutive elements interleaved across banks & number of banks >= bank latency, then can sustain 1 element/cycle throughput
Scalar Code Example

- For $I = 0$ to $49$
  - $C[i] = (A[i] + B[i]) / 2$

- Scalar code (instruction and its latency)

  MOVI R0 = 50  
  MOVA R1 = A  
  MOVA R2 = B  
  MOVA R3 = C  

  X: LD R4 = MEM[R1++]  
      LD R5 = MEM[R2++]  
      ADD R6 = R4 + R5  
      SHFR R7 = R6 >> 1  
      ST MEM[R3++] = R7  

  DECBNZ --R0, X

344 dynamic instructions
Scalar Code Execution Time (In Order)

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined: $2 \times 11$ cycles
  - $4 + 50 \times 40 = 2004$ cycles

- Scalar execution time on an in-order processor with 16 banks (word-interleaved: consecutive words are stored in consecutive banks)
  - First two loads in the loop can be pipelined
  - $4 + 50 \times 30 = 1504$ cycles

- Why 16 banks?
  - 11 cycle memory access latency
  - Having 16 (>11) banks ensures there are enough banks to overlap enough memory operations to cover memory latency
Vectorizable Loops

- A loop is **vectorizable** if each iteration is independent of any other

- For I = 0 to 49
  - C[i] = (A[i] + B[i]) / 2

- Vectorized loop (each instruction and its latency):
  
  \[
  \begin{align*}
  &\text{MOVI VLEN = 50} & 1 & \text{7 dynamic instructions} \\
  &\text{MOVI VSTR = 1} & 1 \\
  &\text{VLD V0 = A} & 11 + \text{VLN} - 1 \\
  &\text{VLD V1 = B} & 11 + \text{VLN} - 1 \\
  &\text{VADD V2 = V0 + V1} & 4 + \text{VLN} - 1 \\
  &\text{VSHFR V3 = V2 >> 1} & 1 + \text{VLN} - 1 \\
  &\text{VST C = V3} & 11 + \text{VLN} - 1
  \end{align*}
  \]
Basic Vector Code Performance

- Assume no chaining (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation

- One memory port (one address generator)
- 16 memory banks (word-interleaved)

- 285 cycles
Vector Chaining

- **Vector chaining**: Data forwarding from one vector functional unit to another

```
LV v1
MULV v3, v1, v2
ADDV v5, v3, v4
```
Vector Code Performance - Chaining

- **Vector chaining**: Data forwarding from one vector functional unit to another

  1 1 11 49 11 49

- **182 cycles**

  These two VLDs cannot be pipelined. WHY?

  VLD and VST cannot be pipelined. WHY?

  **Strict assumption**: Each memory bank has a single port (memory bandwidth bottleneck)
Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank

- 79 cycles
- 19X perf. improvement!
Questions (I)

- What if number of data elements > number of elements in a vector register?
  - Idea: Break loops so that each iteration operates on number of elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
  - Called vector stripmining

- What if vector data is not stored in a strided fashion in memory? (irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into vector registers
  - Called scatter/gather operations
Want to vectorize loops with indirect accesses:

```c
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

Indexed load instruction (\textit{Gather})

```assembly
LV vD, rD       # Load indices in D vector
LVI vC, rC, vD  # Load indirect from rC base
LV vB, rB       # Load B vector
ADDV.D vA,vB,vC # Do add
SV vA, rA       # Store result
```
Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle sparse matrices
- Vector loads and stores use an index vector which is added to the base register to generate the addresses

<table>
<thead>
<tr>
<th>Index Vector</th>
<th>Data Vector (to Store)</th>
<th>Stored Vector (in Memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.14</td>
<td>Base+0 3.14</td>
</tr>
<tr>
<td>2</td>
<td>6.5</td>
<td>Base+1 X</td>
</tr>
<tr>
<td>6</td>
<td>71.2</td>
<td>Base+2 6.5</td>
</tr>
<tr>
<td>7</td>
<td>2.71</td>
<td>Base+3 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+4 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+5 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+6 71.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base+7 2.71</td>
</tr>
</tbody>
</table>
Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop:   if (a[i] != 0) then b[i]=a[i]*b[i]
goto loop
```

- Idea: **Masked operations**
  - VMASK register is a bit mask determining which data element should not be acted upon
    ```
    VLD V0 = A
    VLD V1 = B
    VMASK = (V0 != 0)
    VMUL V1 = V0 * V1
    VST B = V1
    ```
  - Does this look familiar? This is essentially **predicated execution**.
Another Example with Masking

for (i = 0; i < 64; ++i)
    if (a[i] >= b[i])
        c[i] = a[i]
    else
        c[i] = b[i]

Steps to execute the loop in SIMD code

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>VMASK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>-5</td>
<td>-4</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>-7</td>
<td>-8</td>
<td>1</td>
</tr>
</tbody>
</table>
Masked Vector Instructions

Simple Implementation
- execute all N operations, turn off result writeback according to mask

Density-Time Implementation
- scan mask vector and only execute elements with non-zero masks

Which one is better? Tradeoffs?

Slide credit: Krste Asanovic
Some Issues

- Stride and banking
  - As long as they are *relatively prime* to each other and there are enough banks to cover bank access latency, we can sustain 1 element/cycle throughput

- Storage of a matrix
  - **Row major**: Consecutive elements in a row are laid out consecutively in memory
  - **Column major**: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column
Matrix multiplication

A & B, both in row major order

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
6 & 7 & 8 & 9 & 10 & 11
\end{array}
\quad \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19
\end{array}
\]

\[A_{4\times2} \cdot B_{8\times10} \Rightarrow C_{4\times10} \quad \text{(dot products of rows \& columns of A \& B)}\]

A: Load Ao into a vector register \(V1\)

\(\Rightarrow\) each time you need to increment the address by 1 to access the next column

\(\Rightarrow\) first matrix accesses have a stride of 1

B: Load Bo into a vector register \(V2\)

\(\Rightarrow\) each time you need to increment by 10

\(\Rightarrow\) stride of 10

Different strides can lead to bank conflicts.

\(\Rightarrow\) How do you minimize them?
Minimizing Bank Conflicts

- More banks

- Better data layout to match the access pattern
  - Is this always possible?

- Better mapping of address to bank
  - E.g., randomized mapping
Array vs. Vector Processors, Revisited

- Array vs. vector processor distinction is a “purist’s” distinction

- Most “modern” SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
  - GPUs are a prime example we will cover in a bit more detail
Remember: Array vs. Vector Processors

**Instruction Stream**
- **LD**  VR ← A[3:0]
- **ADD** VR ← VR, 1
- **MUL** VR ← VR, 2
- **ST**  A[3:0] ← VR

**Time**
- **LD0**
- **AD0**
- **MU0**
- **ST0**
- **LD1**
- **AD1**
- **MU1**
- **ST1**
- **LD2**
- **AD2**
- **MU2**
- **ST2**
- **LD3**
- **AD3**
- **MU3**
- **ST3**

**Array Processor**
- **Same op @ same time**
  - LD0, LD1, LD2, LD3
  - AD0, AD1, AD2, AD3
  - MU0, MU1, MU2, MU3
  - ST0, ST1, ST2, ST3

**Vector Processor**
- **Different ops @ time**
  - LD0, AD0, MU0, ST0
  - LD1, AD1, MU1, ST1
  - LD2, AD2, MU2, ST2
  - LD3, AD3, MU3, ST3

**Space**
- Same op @ space
Vector Instruction Execution

VADD A,B → C

Execution using one pipelined functional unit


Execution using four pipelined functional units


Vector Unit Structure

- **Functional Unit**
- **Partitioned Vector Registers**
- **Lane**
- **Memory Subsystem**

Elements 0, 4, 8, ...

Elements 1, 5, 9, ...

Elements 2, 6, 10, ...

Elements 3, 7, 11, ...

Slide credit: Krste Asanovic
Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle
Automatic Code Vectorization

Vectorization is a compile-time reordering of operation sequencing
⇒ requires extensive loop dependence analysis

Scalar Sequential Code

Vector Instruction

Vectorized Code

Iter. 1

Iter. 2

Slide credit: Krste Asanovic
Vector/SIMD Processing Summary

- Vector/SIMD machines are good at exploiting **regular data-level parallelism**
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)

- Performance improvement limited by vectorizability of code
  - Scalar operations limit vector machine performance
  - Remember Amdahl’s Law
  - CRAY-1 was the fastest SCALAR machine at its time!

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD
SIMD Operations in Modern ISAs
Intel Pentium MMX Operations

- Idea: One instruction operates on multiple data elements simultaneously
  - Ala array processing (yet much more limited)
  - Designed with multimedia (graphics) operations in mind

No VLEN register
Opcode determines data type:
- 8 8-bit bytes
- 4 16-bit words
- 2 32-bit doublewords
- 1 64-bit quadword

Stride always equal to 1.

MMX Example: Image Overlaying (I)

- **Goal:** Overlay the human in image 1 on top of the background in image 2

**Figure 8. Chroma keying: image overlay using a background color.**

**PCMPEQB MM1, MM3**

<table>
<thead>
<tr>
<th>MM1</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
<th>Blue</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM3</td>
<td>X7!=blue</td>
<td>X6!=blue</td>
<td>X5=blue</td>
<td>X4=blue</td>
<td>X3!=blue</td>
<td>X2!=blue</td>
<td>X1=blue</td>
<td>X0=blue</td>
</tr>
</tbody>
</table>

| MM1  | 0x0000 | 0x0000 | 0xFFF  | 0xFFF  | 0x0000 | 0x0000 | 0xFFF | 0xFFF |

**Figure 9. Generating the selection bit mask.**
Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

Figure 11. MMX code sequence for performing a conditional select.