Reminder: Homeworks

- Homework 4
  - Due Today

- Homework 5
  - Out today
  - Due April 2
  - Topics: Out-of-order execution, dataflow, vector processing, memory, caches
Reminder: Lab Assignments

- **Lab Assignment 4**
  - **Due this Friday (March 23)**
  - Implementing control flow and branch prediction in your pipelined simulator
  - You can get *extra credit* by optimizing your design to be the fastest
    - Just make sure you have one of the Top 3 lowest execution times

- **Lab Assignment 5**
  - Will be out Wednesday
  - **Due April 6**
Don’t Forget: Feedback Sheet for Us

- You can still turn this in
  - Until the end of this weekend (March 25)
Midterm I Grades

- **Average:** 175 / 335 (+40 bonus)
- **Median:** 170 / 335 (+40 bonus)
- **Minimum:** 98 / 335 (+40 bonus)
- **Maximum:** 281 / 335 (+40 bonus)
Midterm Solutions

- Posted online
  - Check and make sure you understand everything
  - Remember: this is just the first midterm; there is one more
    - And then the final

- TAs will cover all solutions during discussion sessions this week and early next week
Midterm I: Per-Question Statistics

I: Potpourri

II: Fine-Grained Multithreading

III: Branch Prediction

IV: Mystery Instruction

V: Out-of-Order Execution

VI: Value Prediction
Midterm I: Per-Question Statistics

VII: What Does This Logic Do?

VIII: Bonus (CISC)
Homework 3 Grades

Average: 91.46
Median: 95.5
Max: 104
Min: 57
Max Possible Points: 105
Total number of students: 49
Lab 3 Grades

Average: 185.5
Median: 190
Max: 248
Min: 90

Max Possible Points (w/o EC): 215
Total number of students: 49
Prizes for Lab 3 Competition

- 8 students had fully-correct designs
- 4 students shared the “fastest” design
  - Earned 10% extra credit
  - Top 3 will soon earn a book each

<table>
<thead>
<tr>
<th>Student</th>
<th>Cycles</th>
<th>Cycle Time (ns)</th>
<th>Execution Time (ns)</th>
<th>Ranking</th>
<th>Relative Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcai</td>
<td>29997</td>
<td>10.554</td>
<td>316588</td>
<td>1st Place</td>
<td>1</td>
</tr>
<tr>
<td>jasonli1</td>
<td>28693</td>
<td>11.109</td>
<td>318750</td>
<td>2nd Place</td>
<td>1.01</td>
</tr>
<tr>
<td>thuberty</td>
<td>28303</td>
<td>11.687</td>
<td>330777</td>
<td>3rd Place</td>
<td>1.04</td>
</tr>
<tr>
<td>jwloh</td>
<td>31151</td>
<td>10.73</td>
<td>334250</td>
<td>4th Place</td>
<td>1.06</td>
</tr>
<tr>
<td>twomack</td>
<td>28653</td>
<td>18.705</td>
<td>535954</td>
<td>5th Place</td>
<td>1.69</td>
</tr>
<tr>
<td>dhavals</td>
<td>29045</td>
<td>18.653</td>
<td>541776</td>
<td>6th Place</td>
<td>1.71</td>
</tr>
<tr>
<td>dliang</td>
<td>31327</td>
<td>27.632</td>
<td>865627</td>
<td>7th Place</td>
<td>2.73</td>
</tr>
<tr>
<td>pmpatel</td>
<td>57819</td>
<td>20.074</td>
<td>1160658</td>
<td>8th Place</td>
<td>3.67</td>
</tr>
</tbody>
</table>
Prizes for Lab 3 Competition

- Rui Cai
- Jason Lin
- Tyler Huberty
Readings for Next Time

- Cache chapters from P&H: 5.1-5.3
- Memory/cache chapters from Hamacher+: 8.1-8.7
- First cache paper by Maurice Wilkes, 1965
Review of Last Lecture

- Precise exceptions
  - Reorder buffer
  - Future file
  - In-order pipeline with precise exceptions

- Out-of-order execution
  - Tomasulo’s algorithm
Today

- Wrap up out-of-order execution
- Data flow
- Vector processing
Out-of-Order Execution
(Dynamic Instruction Scheduling)
Enabling OoO Execution

1. Link the consumer of a value to the producer
   - Register renaming: Associate a “tag” with each data value

2. Buffer instructions until they are ready
   - Insert instruction into reservation stations after renaming

3. Keep track of readiness of source values of an instruction
   - Broadcast the “tag” when the value is produced
   - Instructions compare their “source tags” to the broadcast tag
     → if match, source value becomes ready

4. When all source values of an instruction are ready, dispatch the instruction to functional unit (FU)
   - Wakeup and schedule the instruction
Review: An Exercise

Assume ADD (4 cycle execute), MUL (6 cycle execute)

Assume one adder and one multiplier

How many cycles

- in a non-pipelined machine
- in an in-order-dispatch pipelined machine with imprecise exceptions (no forwarding and full forwarding)
- in an out-of-order dispatch pipelined machine imprecise exceptions (full forwarding)
An Exercise, with Precise Exceptions

- Assume ADD (4 cycle execute), MUL (6 cycle execute)
- Assume one adder and one multiplier
- How many cycles
  - in a non-pipelined machine
  - in an in-order-dispatch pipelined machine with reorder buffer (no forwarding and full forwarding)
  - in an out-of-order dispatch pipelined machine with reorder buffer (full forwarding)
Out-of-Order Execution with Precise Exceptions

- **Idea:** Use a reorder buffer to reorder instructions before committing them to architectural state

- An instruction updates the register alias table (essentially a future file) when it completes execution

- An instruction updates the *architectural register file* when it is the oldest in the machine and has completed execution
Out-of-Order Execution with Precise Exceptions

- **Hump 1:** Reservation stations (scheduling window)
- **Hump 2:** Reordering (reorder buffer, aka instruction window or active window)
Summary of OoO Execution Concepts

- Renaming eliminates false dependencies

- Tag broadcast enables value communication between instructions → dataflow

- An out-of-order engine dynamically builds the dataflow graph of a piece of the program
  - which piece?
    - Limited to the instruction window
    - Instruction window: the set of instructions that are currently in the machine, i.e. decoded but not yet retired
  - Can we do it for the whole program? Why would we like to?
    - How can we have a large instruction window efficiently?
Questions

- Why is OoO execution beneficial?
  - What if all operations take single cycle?
  - **Latency tolerance**: OoO execution tolerates the latency of multi-cycle operations by executing independent operations concurrently

- What if an instruction takes 500 cycles?
  - How large of an instruction window do we need to continue decoding?
  - How many cycles of latency can OoO tolerate?
  - **What limits the latency tolerance scalability of Tomasulo’s algorithm?**
    - **Active/instruction window size**: determined by register file, scheduling window, reorder buffer
Food for Thought for You (I)

- How can you implement branch prediction in an out-of-order execution machine?
  - Think about branch history register and PHT updates
  - Think about recovery from mispredictions
    - How to do this fast?

- How can you combine superscalar execution with out-of-order execution?
  - These are different concepts
  - Concurrent renaming of instructions
  - Concurrent broadcast of tags

- How can you combine superscalar + out-of-order + branch prediction?
How do you handle load and store execution in an out-of-order execution engine?

- Problem: A younger load can have its address ready before an older store’s address is known.
- Known as the memory disambiguation problem or the unknown address problem.

Approaches:
- **Conservative:** Stall the load until all previous stores have computed their addresses (or even retired from the machine).
- **Aggressive:** Assume load is independent of unknown-address stores and schedule the load right away.
- **Intelligent:** Predict (with a more sophisticated predictor) if the load is dependent on the/any unknown address store.
Handling of Store-Load Dependencies

- A load’s dependence status is not known until all previous store addresses are available.

- How does the OOO engine detect dependence of a load instruction on a previous store?
  - Option 1: Wait until all previous stores committed (no need to check)
  - Option 2: Keep a list of pending stores in a store buffer and check whether load address matches a previous store address

- How does the OOO engine treat the scheduling of a load instruction wrt previous stores?
  - Option 1: Assume load dependent on all previous stores
  - Option 2: Assume load independent of all previous stores
  - Option 3: Predict the dependence of a load on an outstanding store
Memory Disambiguation (I)

- Option 1: Assume load dependent on all previous stores
  + No need for recovery
  -- Too conservative: delays independent loads unnecessarily

- Option 2: Assume load independent of all previous stores
  + Simple and can be common case: no delay for independent loads
  -- Requires recovery and re-execution of load and dependents on misprediction

- Option 3: Predict the dependence of a load on an outstanding store
  + More accurate. Load store dependencies persist over time
  -- Still requires recovery/re-execution on misprediction
  - Alpha 21264: Initially assume load independent, delay loads found to be dependent
Memory Disambiguation (II)


- Predicting store-load dependencies important for performance
- Simple predictors (based on past history) can achieve most of the potential performance
Many other design choices

For example: Should reservation stations be centralized or distributed?
- What are the tradeoffs?
Recommended Readings


Data Flow:
Exploiting Irregular Parallelism
Dataflow Graph for Our Example

- In the last lecture

MUL R3 ← R1, R2
ADD R5 ← R3, R4
ADD R7 ← R2, R6
ADD R10 ← R8, R9
MUL R11 ← R7, R10
ADD R5 ← R5, R11
# State of RAT and RS in Cycle 7

**end of cycle 7:**

<table>
<thead>
<tr>
<th>V tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>~1</td>
</tr>
<tr>
<td>R2</td>
<td>~2</td>
</tr>
<tr>
<td>R3</td>
<td>~4</td>
</tr>
<tr>
<td>R4</td>
<td>~6</td>
</tr>
<tr>
<td>R5</td>
<td>~8</td>
</tr>
<tr>
<td>R6</td>
<td>~9</td>
</tr>
<tr>
<td>R7</td>
<td>~9</td>
</tr>
<tr>
<td>R8</td>
<td>~9</td>
</tr>
<tr>
<td>R9</td>
<td>~9</td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>1</td>
<td>~4</td>
</tr>
<tr>
<td>1</td>
<td>~2</td>
<td>1</td>
<td>~6</td>
</tr>
<tr>
<td>1</td>
<td>~8</td>
<td>1</td>
<td>~9</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>~0</td>
<td>y</td>
</tr>
</tbody>
</table>

**X**

- All 6 meters removed.
- Note what happened to RS5.
Dataflow Graph

Nodes: operations performed by the instruction

Arrows: tags in Tamosulo's algorithm

- MUL R1, R2 → R3 (x)
- ADD R3, R4 → R5 (a)
- ADD R2, R6 → R7 (b)
- ADD R8, R9 → R10 (c)
- MUL R7, R10 → R11 (y)
- ADD R5, R11 → R5 (d)
Restricted Data Flow

- An out-of-order machine is a “restricted data flow” machine
  - Dataflow-based execution is restricted to the microarchitecture level
  - ISA is still based on von Neumann model (sequential execution)

- Remember the data flow model (at the ISA level):
  - Dataflow model: An instruction is fetched and executed in data flow order
  - i.e., when its operands are ready
  - i.e., there is no instruction pointer
  - Instruction ordering specified by data flow dependence
    - Each instruction specifies “who” should receive the result
    - An instruction can “fire” whenever all operands are received
In a data flow machine, a program consists of data flow nodes.

- A data flow node fires (fetched and executed) when all its inputs are ready
  - i.e. when all inputs have tokens

Data flow node and its ISA representation
Data Flow Nodes

*Conditional

*Relational

*Barrier Synch
A small set of dataflow operators can be used to define a general programming language.
Dataflow Graphs

Values in dataflow graphs are represented as tokens

An operator executes when all its input tokens are present; copies of the result token are distributed to the destination operators

no separate control flow
Example Data Flow Program
Control Flow vs. Data Flow

\[
\begin{align*}
a &:= x + y \\
b &:= a \times a \\
c &:= 4 - a
\end{align*}
\]

Figure 2. A comparison of control flow and dataflow programs. On the left a control flow program for a computer with memory-to-memory instructions. The arcs point to the locations of data that are to be used or created. Control flow arcs are indicated with dashed arrows; usually most of them are implicit. In the equivalent dataflow program on the right only one memory is involved. Each instruction contains pointers to all instructions that consume its results.
Data Flow Characteristics

- Data-driven execution of instruction-level graphical code
  - Nodes are operators
  - Arcs are data (I/O)
  - As opposed to control-driven execution
- Only real dependencies constrain processing
- No sequential I-stream
  - No program counter
- Operations execute asynchronously
- Execution triggered by the presence of data
A Dataflow Processor

Token = Data + Tag + Destination

Matching Area

Group = Data1 + Data2 + Tag + Destination

Instruction Fetch Area

Execution Package = Data1 + Data2 + OpCode + Tag + Destination

Data Flow Proc. Element

Token = Data + Tag + Destination
MIT Tagged Token Data Flow Architecture

- **Wait–Match Unit**: try to match incoming token and context id and a waiting token with same instruction address
  - **Success**: Both tokens forwarded
  - **Fail**: Incoming token $\rightarrow$ Waiting Token Mem, bubble (no-op forwarded)
TTDA Data Flow Example

**Conceptual**

**Encoding of graph**

<table>
<thead>
<tr>
<th>Program memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Opcode</strong></td>
</tr>
<tr>
<td>109</td>
</tr>
<tr>
<td>113</td>
</tr>
<tr>
<td>120</td>
</tr>
<tr>
<td>141</td>
</tr>
<tr>
<td>159</td>
</tr>
</tbody>
</table>

**Re-entrancy ("dynamic" dataflow):**

- Each invocation of a function or loop iteration gets its own, unique, "Context"
- Tokens destined for same instruction in different invocations are distinguished by a context identifier

**Encoding of token:**

A "packet" containing:

- 120R 6.847 Destination instruction address, Left/Right port Value

- Ctxt 6.847 Context Identifier Value
TTDA Data Flow Example
TTDA Data Flow Example

Conceptual:

Heap Memory

Encoding of graph:

Program memory:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>Fetch, 207</td>
</tr>
<tr>
<td>207</td>
<td>op1, ...</td>
</tr>
</tbody>
</table>
Manchester Data Flow Machine

- Matching Store: Pairs together tokens destined for the same instruction
- Large data set → overflow in overflow unit
- Paired tokens fetch the appropriate instruction from the node store
Data Flow Advantages/Disadvantages

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

- Disadvantages
  - No precise state
    - Interrupt/exception handling is difficult
    - Debugging very difficult (no precise state)
  - Bookkeeping overhead (tag matching)
  - Too much parallelism? (Parallelism control needed)
    - Overflow of tag matching tables
  - Implementing dynamic data structures difficult
Data Flow Summary

- Availability of data determines order of execution
- A data flow node fires when its sources are ready
- Programs represented as data flow graphs (of nodes)

- 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
  - Hwu and Patt, “HPSm, a high performance restricted data flow architecture having minimal functionality,” ISCA 1986.
Further Reading on Data Flow

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

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

- SIMD exploits instruction-level parallelism
  - Multiple instructions 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
  - **Vector processor**: Instruction operates on multiple data elements in consecutive time steps
SIMD Array Processing vs. VLIW

- VLIW
SIMD Array Processing vs. VLIW

- Array processor

```
Program Counter
```

```
add VR, VR, 1
```

```
VLEN = 4
```

```
add VR[0],VR[0],1 add VR[1],VR[1],1 add VR[2],VR[2],1 add VR[3],VR[3],1
```

```
Instruction Execution
```

```
PE PE PE PE
```
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
We did not cover the following slides in lecture. These are for your preparation for the next lecture.
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 $\rightarrow$ no hardware interlocking within a vector
  - No control flow within a vector
  - Known stride allows prefetching of vectors into 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

+ Highly regular memory access pattern
  - Interleaving multiple banks for higher memory bandwidth
  - Prefetching

+ 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 Registers

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

- Use deep pipeline (= fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent

\[ V3 \leftarrow v1 \ast v2 \]

*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