Lecture 14: Out-of-Order Execution (Dynamic Instruction Scheduling)
Announcements

- Lab due Friday (Feb 21)
- Homework 3 due next Wednesday (Feb 26)
- Exam coming up (before Spring Break)
Reminder: Lab Late Day Policy Adjustment

- Your total late days have increased to 7

- Each late day beyond all exhausted late days costs you 15% of the full credit of the lab
Reminder: A Note on Testing Your Code

- Testing is critical in developing any system

- You are responsible for creating your own test programs and ensuring your designs work for all possible cases

- That is how real life works also...
  - No one gives you all possible test cases, workloads, users, etc. beforehand
Readings for Past Few Lectures (I)

- P&H Chapter 4.9-4.11

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts


Readings for Past Few Lectures (II)

Readings Specifically for Today

  - More advanced pipelining
  - Interrupt and exception handling
  - Out-of-order and superscalar execution concepts

Readings for Next Lecture

- SIMD Processing
- Basic GPU Architecture
- Other execution models: VLIW, Dataflow


- Stay tuned for more readings...
Maintaining Precise State

- Reorder buffer
- History buffer
- Future register file
- Checkpointing

Readings
Registers versus Memory

- So far, we considered mainly registers as part of state

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)
Maintaining Speculative Memory State: Stores

- Handling out-of-order completion of memory operations
  - UNDOing a memory write more difficult than UNDOing a register write. Why?
  - One idea: Keep store address/data in reorder buffer
    - How does a load instruction find its data?
  - Store/write buffer: Similar to reorder buffer, but used only for store instructions
    - Program-order list of un-committed store operations
    - When store is decoded: Allocate a store buffer entry
    - When store address and data become available: Record in store buffer entry
    - When the store is the oldest instruction in the pipeline: Update the memory address (i.e. cache) with store data
Out-of-Order Execution
(Dynamic Instruction Scheduling)
Problem: A true data dependency stalls dispatch of younger instructions into functional (execution) units

Dispatch: Act of sending an instruction to a functional unit
Can We Do Better?

- What do the following two pieces of code have in common (with respect to execution in the previous design)?

```
IMUL R3 ← R1, R2  ADD R3 ← R3, R1
ADD R3 ← R3, R1  ADD R3 ← R3, R1
ADD R1 ← R6, R7  ADD R1 ← R6, R7
IMUL R5 ← R6, R8  IMUL R5 ← R6, R8
ADD R7 ← R3, R5  ADD R7 ← R3, R5
LD R3 ← R1 (0)
```

- Answer: First ADD stalls the whole pipeline!
  - ADD cannot dispatch because its source registers unavailable
  - Later independent instructions cannot get executed

- How are the above code portions different?
  - Answer: Load latency is variable (unknown until runtime)
  - What does this affect? Think compiler vs. microarchitecture
Preventing Dispatch Stalls

- Multiple ways of doing it
- You have already seen THREE:
  - 1. Fine-grained multithreading
  - 2. Value prediction
  - 3. Compile-time instruction scheduling/reordering
- What are the disadvantages of the above three?

- Any other way to prevent dispatch stalls?
  - Actually, you have briefly seen the basic idea before
    - Dataflow: fetch and “fire” an instruction when its inputs are ready
  - Problem: in-order dispatch (scheduling, or execution)
  - Solution: out-of-order dispatch (scheduling, or execution)
Out-of-order Execution (Dynamic Scheduling)

- **Idea:** Move the dependent instructions out of the way of independent ones
  - Rest areas for dependent instructions: Reservation stations

- Monitor the source “values” of each instruction in the resting area

- When all source “values” of an instruction are available, “fire” (i.e. dispatch) the instruction
  - Instructions dispatched in dataflow (not control-flow) order

- **Benefit:**
  - **Latency tolerance:** Allows independent instructions to execute and complete in the presence of a long latency operation
In-order vs. Out-of-order Dispatch

- In order dispatch + precise exceptions:

- Out-of-order dispatch + precise exceptions:

- 16 vs. 12 cycles
Enabling OoO Execution

1. Need to link the consumer of a value to the producer
   - Register renaming: Associate a “tag” with each data value
2. Need to buffer instructions until they are ready to execute
   - Insert instruction into reservation stations after renaming
3. Instructions need to keep track of readiness of source values
   - 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, need to dispatch the instruction to its functional unit (FU)
   - Instruction wakes up if all sources are ready
   - If multiple instructions are awake, need to select one per FU
Tomasulo’s Algorithm

- OoO with register renaming invented by Robert Tomasulo
  - Used in IBM 360/91 Floating Point Units

- What is the major difference today?
  - **Precise exceptions:** IBM 360/91 did NOT have this

- Variants used in most high-performance processors
  - Initially in Intel Pentium Pro, AMD K5
  - Alpha 21264, MIPS R10000, IBM POWER5, IBM z196, Oracle UltraSPARC T4, ARM Cortex A15
Two Humps in a Modern Pipeline

- **Hump 1**: Reservation stations (scheduling window)
- **Hump 2**: Reordering (reorder buffer, aka instruction window or active window)
General Organization of an OOO Processor


---

Tomasulo’s Machine: IBM 360/91

Load buffers from memory

Operation bus

FP FU

Reservation stations

FP registers

Store buffers to memory

Common data bus

From instruction unit
Register Renaming

- Output and anti dependencies are not true dependencies
  - WHY? The same register refers to values that have nothing to do with each other
  - They exist because not enough register ID’s (i.e. names) in the ISA

- The register ID is renamed to the reservation station entry that will hold the register’s value
  - Register ID → RS entry ID
  - Architectural register ID → Physical register ID
  - After renaming, RS entry ID used to refer to the register

- This eliminates anti- and output- dependencies
  - Approximates the performance effect of a large number of registers even though ISA has a small number
### Tomasulo’s Algorithm: Renaming

- Register rename table (register alias table)

<table>
<thead>
<tr>
<th>tag</th>
<th>value</th>
<th>valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R4</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R6</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R8</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>R9</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Tomasulo’s Algorithm

- If reservation station available before renaming
  - Instruction + renamed operands (source value/tag) inserted into the reservation station
  - Only rename if reservation station is available
- Else stall
- While in reservation station, each instruction:
  - Watches common data bus (CDB) for tag of its sources
  - When tag seen, grab value for the source and keep it in the reservation station
  - When both operands available, instruction ready to be dispatched
- Dispatch instruction to the Functional Unit when instruction is ready
- After instruction finishes in the Functional Unit
  - Arbitrate for CDB
  - Put tagged value onto CDB (tag broadcast)
  - Register file is connected to the CDB
    - Register contains a tag indicating the latest writer to the register
    - If the tag in the register file matches the broadcast tag, write broadcast value into register (and set valid bit)
  - Reclaim rename tag
    - no valid copy of tag in system!
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)
Exercise Continued

Pipeline structure

```
MUL R1, R2 -> R3
ADD R3, R4 -> R5
ADD R2, R6 -> R7
ADD R8, R9 -> R10
MUL R7, R10 -> R11
ADD R5, R11 -> R5

MUL takes 6 cycles
ADD takes 4 cycles

How many cycles total with data forwarding?
" " " " " " w1 " " ?
```
Exercise Continued

FD123456W
FD---D1234W
F------D1234W
FD1234W
FD---D123456W
F------D1234W

Execution timeline w/ scoreboard

FD123456W
FD E1234W
F D1234W
FD01234W
FD1234W
FD1234W
FD1234W
F D1234W

31 cycles
25 cycles
Exercise Continued

MUL  R3 ← R1, R2
ADD  R5 ← R3, R4
ADD  R7 ← R2, R6
ADD  R10 ← R8, R9
MUL  R11 ← R7, R10
ADD  R5 ← R5, R11

Tomosulo’s algorithm + full forwarding  20 cycles
How It Works

Register Alias Table

V Tag Value

of writer

Reservoir Stations for ADDER

SRC1

SRC2

V tag value

V tag value

V tag value

V tag value

X Y

Assume
dozer & multiplier have separate buses

Value tag

Value tag
Cycle 0

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

- Initial contents of the register status table
- Register statuses are all invalid
Cycle 2

**Cycle 2:** \( \text{MUL } R1, R2 \rightarrow R3 \) reads its sources from the RAT
- writes to its destination in the RAT (renames its destination)
- allocates a reservation station entry
- allocates a tag for its destination register
- places its sources in the reservation station entry that is allocated.

End of cycle 2:

<table>
<thead>
<tr>
<th></th>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

\( \text{MUL at } X \) becomes ready to execute
- \( \text{What if multiple instructions become ready at the same time?} \)
- both of its sources are valid in the reservation station \( X \)
Cycle 3:

- Cycle 3:
  -穆尔 X 状态转移
  -ADD R3, R4; RS 地址重新编号并插入 ADDER 记录器

End of cycle 3:

<table>
<thead>
<tr>
<th>R1</th>
<th>1 ~ 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>1 ~ 2</td>
</tr>
<tr>
<td>R3</td>
<td>0 ~ 4</td>
</tr>
<tr>
<td>R4</td>
<td>1 ~ 4</td>
</tr>
<tr>
<td>RS</td>
<td>0 ~</td>
</tr>
<tr>
<td>RB</td>
<td>1 ~ 6</td>
</tr>
<tr>
<td>RL</td>
<td>1 ~ 11</td>
</tr>
</tbody>
</table>

- ADD at a cannot be ready to execute because one of its sources is not ready
- It is waiting for the value with the tag X to be broadcast (by the mul m X)

Aside: Does the tag need to be associated with the RS entry of the producer?

Answer: No: Tag is a tag for the value that is communicated.

RS is a place to hold the address that enables data-flow. While they become ready, prese two are completely orthogonal.
Cycle 4

cycle 4:  
- ADD R2, R6 → R7 gets renamed and placed into RS

end of cycle 4:

- ADD at b becomes ready to execute
  (both sources are ready!)

- At cycle 5, it is sent to the adder out-of-program order
  → It is executed before the add in a
end of cycle 7:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
</tr>
<tr>
<td>R3</td>
<td>X</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
</tr>
<tr>
<td>R5</td>
<td>0</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
</tr>
<tr>
<td>R8</td>
<td>B</td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
</tr>
<tr>
<td>R11</td>
<td>Y</td>
</tr>
</tbody>
</table>

```
V
0 x ~ 1 ~ 4
1 ~ 2 1 ~ 6
1 ~ 8 1 ~ 9
0 a ~ 0 y ~
```

```
x1 ~ 1 | 1 | 1 ~ 2
4 ~ b ~ 0 c ~
```

* All 6 meters removed.
- Note what happened to R5
Cycle 8:

- MUL at X and ADD at b
  broadcast their tags and values

- RS entries waiting for these tags capture the values
  and set the Valid b accordingly

  \( \rightarrow (\text{What is needed in HW to accomplish this?}) \)

  CAM on tags that are broadcast for all RS
  entries & sources

- RAT entries waiting for these tags also capture the
  values and set the Valid b0s accordingly
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)

```
MUL   R3 ← R1, R2
ADD   R5 ← R3, R4
ADD   R7 ← R2, R6
ADD   R10 ← R8, R9
MUL   R11 ← R7, R10
ADD   R5 ← R5, R11
```
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)
Enabling OoO Execution, Revisited

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 select/schedule the instruction
Summary of OOO Execution Concepts

- Register renaming eliminates false dependencies, enables linking of producer to consumers
- Buffering enables the pipeline to move for independent ops
- Tag broadcast enables communication (of readiness of produced value) between instructions
- Wakeup and select enables out-of-order dispatch
An out-of-order engine dynamically builds the dataflow graph of a piece of the program
- which piece?

The dataflow graph is limited to the instruction window
- Instruction window: all decoded but not yet retired instructions

Can we do it for the whole program?
Why would we like to?
In other words, how can we have a large instruction window?
Can we do it efficiently with Tomasulo’s algorithm?
Dataflow Graph for Our Example

\[
\begin{align*}
&MUL \quad R3 \leftarrow R1, R2 \\
&ADD \quad R5 \leftarrow R3, R4 \\
&ADD \quad R7 \leftarrow R2, R6 \\
&ADD \quad R10 \leftarrow R8, R9 \\
&MUL \quad R11 \leftarrow R7, R10 \\
&ADD \quad R5 \leftarrow R5, R11
\end{align*}
\]
State of RAT and RS in Cycle 7

end of cycle 7:

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>~1</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>~2</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>~4</td>
</tr>
<tr>
<td>R5</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>~6</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>R8</td>
<td>1</td>
<td>~8</td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
<td>~9</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>C</td>
</tr>
<tr>
<td>R11</td>
<td>0</td>
<td>Y</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>

!![diagram]

* All 6 instructions removed.
- Note what happened to R5
Dataflow Graph

- **Nodes**: operations performed by the instruction
- **Arcs**: tags in Tamasulo'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
Questions to Ponder

- 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.
Registers versus Memory, Revisited

- So far, we considered register based value communication between instructions

- What about memory?

- What are the fundamental differences between registers and memory?
  - Register dependences known statically – memory dependences determined dynamically
  - Register state is small – memory state is large
  - Register state is not visible to other threads/processors – memory state is shared between threads/processors (in a shared memory multiprocessor)
Memory Dependence Handling (I)

- Need to obey memory dependences in an out-of-order machine
  - and need to do so while providing high performance

- Observation and Problem: Memory address is not known until a load/store executes

- Corollary 1: Renaming memory addresses is difficult
- Corollary 2: Determining dependence or independence of loads/stores need to be handled after their execution
- Corollary 3: When a load/store has its address ready, there may be younger/older loads/stores with undetermined addresses in the machine
When do you schedule a load instruction in an OOO 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
Food for Thought for You

- Many other design choices

- Should reservation stations be centralized or distributed?
  - What are the tradeoffs?

- Should reservation stations and ROB store data values or should there be a centralized physical register file where all data values are stored?
  - What are the tradeoffs?

- Exactly when does an instruction broadcast its tag?

- ...

...
More Food for Thought for You

- 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?
Recommended Readings


Other Approaches to Concurrency
(or Instruction Level Parallelism)
Approaches to (Instruction-Level) Concurrency

- Pipelining
- Out-of-order execution
- Dataflow (at the ISA level)
- SIMD Processing
- VLIW

- Systolic Arrays
- Decoupled Access Execute
Data Flow: Exploiting Irregular Parallelism
Remember: State of RAT and RS in Cycle 7

end of cycle 7:

<table>
<thead>
<tr>
<th>V</th>
<th>tag</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>1</td>
<td>~</td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>~</td>
</tr>
<tr>
<td>R3</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>R4</td>
<td>1</td>
<td>~</td>
</tr>
<tr>
<td>R5</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>R6</td>
<td>1</td>
<td>~</td>
</tr>
<tr>
<td>R7</td>
<td>0</td>
<td>b</td>
</tr>
<tr>
<td>R8</td>
<td>1</td>
<td>~</td>
</tr>
<tr>
<td>R9</td>
<td>1</td>
<td>~</td>
</tr>
<tr>
<td>R10</td>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>R11</td>
<td>0</td>
<td>Y</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>~</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>~</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>~</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>a</td>
<td>~</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>~</td>
<td>0</td>
<td>c</td>
</tr>
</tbody>
</table>

* All 6 multipliers removed.
- Note what happened to R5
Remember: Dataflow Graph

\[
\begin{align*}
\text{MUL} & \quad R_1, R_2 \rightarrow R_3 \quad \text{(x)} \\
\text{ADD} & \quad R_3, R_4 \rightarrow R_5 \quad \text{(a)} \\
\text{ADD} & \quad R_2, R_6 \rightarrow R_7 \quad \text{(b)} \\
\text{ADD} & \quad R_8, R_9 \rightarrow R_{10} \quad \text{(c)} \\
\text{MUL} & \quad R_7, R_{10} \rightarrow R_{11} \quad \text{(y)} \\
\text{ADD} & \quad R_5, R_{11} \rightarrow R_5 \quad \text{(d)}
\end{align*}
\]

\text{Nodes: operations performed by the instruction.}

\text{Arcs: tags in Tomasulo’s algorithm.}
Review: More on Data Flow

- 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

\{x = a + b; \\
y = b \times 7 \\
in \\
(x-y) \times (x+y)\}\}

- Values in dataflow graphs are represented as tokens

  token $< \text{ip}, \text{p}, \text{v} >$

  instruction ptr  port  data

- 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 = Data1 + Tag + Destination

Matching Area

Group = Data1 + Data2 + Tag + Destination

Instruction Fetch Area

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

Data Flow Proc. Element

Pool of Unmatched Tokens

OP → Dest.

New One

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

Program memory:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Destination(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>109</td>
<td>op1 120L</td>
</tr>
<tr>
<td>113</td>
<td>op2 120R</td>
</tr>
<tr>
<td>120</td>
<td>+ 141, 159L</td>
</tr>
<tr>
<td>141</td>
<td>op3 ...</td>
</tr>
<tr>
<td>159</td>
<td>op4 ... , ...</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:

<table>
<thead>
<tr>
<th>120R</th>
<th>Destination instruction address, Left/Right port</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.847</td>
<td>Value</td>
</tr>
</tbody>
</table>
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>

Wait-Match Unit
200, c, A
Instruction Fetch
207, c, Fetch, A
Execute Op
Fetch, A, 207, c
Form Tokens
Fetch, A, 207, c
Output
Fetch, A, 207, c
207, c, v
Network
207, c, v
Heap Memory Module Containing Address A
200, c, A
Program memory
200 Fetch 207
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
  - 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

- ISA level dataflow

- Microarchitecture-level dataflow:
  - Hwu and Patt, “HPSm, a high performance restricted data flow architecture having minimal functionality,” ISCA 1986.
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 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 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
Array vs. Vector Processors

Instruction Stream

LD  VR ← A[3:0]
ADD VR ← VR, 1
MUL VR ← VR, 2
ST  A[3:0] ← VR

ARRAY PROCESSOR

VECTOR PROCESSOR

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

Time

Same op @ same time

Different ops @ time

Different ops @ same space

Same op @ space

Space

82
SIMD Array Processing vs. VLIW

- VLIW
SIMD Array Processing vs. VLIW

- Array processor
Vector Processors

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors
  
  ```c
  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 \(\rightarrow\) 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., 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

Slide credit: Krste Asanovic
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
Memory Banking

- Example: 16 banks; can start one bank access per cycle
- Bank latency: 11 cycles
- Can sustain 16 parallel accesses if they go to different banks
Vector Memory System

Vector Registers

Address Generator

Memory Banks

Base

Stride

Slide credit: Krste Asanovic
Scalar Code Example

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

- Scalar code
  - MOV 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

304 dynamic instructions
Scalar Code Execution Time

- 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)
  - 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:
  
  ```
  MOVI VLEN = 50
  MOVI VSTR = 1
  VLD V0 = A
  VLD V1 = B
  VADD V2 = V0 + V1
  VSHFR V3 = V2 >> 1
  VST C = V3
  ```
  
  7 dynamic instructions
Vector Code Performance

- No chaining
  - i.e., output of a vector functional unit cannot be used as the input of another (i.e., no vector data forwarding)

- 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

- **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
Questions (I)

- What if # data elements > # elements in a vector register?
  - Need to break loops so that each iteration operates on # 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)
  - Use indirection to combine elements into vector registers
  - Called scatter/gather operations
Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
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</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.14</td>
<td>3.14</td>
</tr>
<tr>
<td>3</td>
<td>6.5</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>71.2</td>
<td>6.5</td>
</tr>
<tr>
<td>8</td>
<td>2.71</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>71.2</td>
<td>71.2</td>
</tr>
<tr>
<td></td>
<td>2.7</td>
<td>2.7</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]) then c[i] = a[i]
    else c[i] = b[i]

Steps to execute loop

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

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, consecutive accesses proceed in parallel

- **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

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

\[A_{6\times6} B_{6\times10} \rightarrow C_{4\times10}\] (dot products of rows & columns of A & B)

A: Load A₀ into a vector register V₁
   → each time you need to increment the address by 1 to access the next column
   → First matrix accesses have a stride of 1

B: Load B₀ into a vector register V₂
   → each time you need to increment by 10
   → stride of 10

Different strides can lead to bank conflicts.
   → How do you minimize them?
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
Remember: Array vs. Vector Processors

**ARRAY PROCESSOR**

**VECTOR PROCESSOR**

<table>
<thead>
<tr>
<th>PE0</th>
<th>PE1</th>
<th>PE2</th>
<th>PE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD0</td>
<td>LD1</td>
<td>LD2</td>
<td>LD3</td>
</tr>
<tr>
<td>AD0</td>
<td>AD1</td>
<td>AD2</td>
<td>AD3</td>
</tr>
<tr>
<td>MU0</td>
<td>MU1</td>
<td>MU2</td>
<td>MU3</td>
</tr>
<tr>
<td>ST0</td>
<td>ST1</td>
<td>ST2</td>
<td>ST3</td>
</tr>
</tbody>
</table>

**Instruction Stream**

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

**Time**

**Space**

- **Same op @ same time**
- **Different ops @ same space**

- **Different ops @ time**
- **Same op @ space**
Vector Instruction Execution

**ADDV C, A, B**

**Execution using one pipelined functional unit**


**Execution using four pipelined functional units**


Slide credit: Krste Asanovic
Vector Unit Structure

Memory Subsystem

Vector Registers

Lane

Functional Unit

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
- Complete 24 operations/cycle while issuing 1 short instruction/cycle

Slide credit: Krste Asanovic
for \( i=0; \ i < N; \ i++ \) 
\[ C[i] = A[i] + B[i]; \]

Vectorization is a compile-time reordering of operation sequencing
\[ \Rightarrow \] requires extensive loop dependence analysis

Slide credit: Krste Asanovic
Vector/SIMD Processing Summary

- Vector/SIMD machines 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
  - 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.


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).
MMX Example: Image Overlaying (I)

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

<table>
<thead>
<tr>
<th>PCMPSEQB MM1, MM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM1</td>
</tr>
<tr>
<td>MM3</td>
</tr>
<tr>
<td>MM1</td>
</tr>
</tbody>
</table>

Figure 9. Generating the selection bit mask.
MMX Example: Image Overlaying (II)

Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

Movie mm3, mem1 /* Load eight pixels from woman's image
Movie mm4, mem2 /* Load eight pixels from the blossom image
PcmpaQb mm1, mm3
Pand mm4, mm1
Pandn mm1, mm3
Por mm4, mm1

Figure 11. MMX code sequence for performing a conditional select.
Graphics Processing Units
SIMD not Exposed to Programmer (SIMT)
High-Level View of a GPU
Concept of “Thread Warps” and SIMT

- Warp: A set of threads that execute the same instruction (on different data elements) \(\rightarrow\) SIMT (Nvidia-speak)
- All threads run the same kernel
- Warp: The threads that run lengthwise in a woven fabric ...

![Diagram showing thread warps and SIMD pipeline](image-url)
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];

Scalar Sequential Code

Vectorized Code

Slide credit: Krste Asanovic
SIMT Memory Access

- Same instruction in different threads uses thread id to index and access different data elements

Let's assume N=16, blockDim=4 → 4 blocks

Slide credit: Hyesoon Kim
Sample GPU SIMT Code (Simplified)

CPU code

```c
for (ii = 0; ii < 100; ++ii) {
}
```

CUDA code

```c
// there are 100 threads
__global__ void KernelFunction(...) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    int varA = aa[tid];
    int varB = bb[tid];
    C[tid] = varA + varB;
}
```

Slide credit: Hyesoon Kim
Sample GPU Program (Less Simplified)

**CPU Program**

```c
void add matrix
    ( float *a, float* b, float *c, int N) {
    int index;
    for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j) {
            index = i + j*N;
            c[index] = a[index] + b[index];
        }
}

int main () {
    add matrix (a, b, c, N);
}
```

**GPU Program**

```c
__global__ add_matrix
    ( float *a, float *b, float *c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    int index = i + j*N;
    if (i < N && j < N)
        c[index] = a[index]+b[index];
}

int main() {
    dim3 dimBlock( blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N);
}
```
Latency Hiding with “Thread Warps”

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies

- **Register values of all threads stay in register file**

- **No OS context switching**

- **Memory latency hiding**
  - Graphics has millions of pixels

---

Slide credit: Tor Aamodt
Warp-based SIMD vs. Traditional SIMD

- Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

- Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
  - Essentially, it is SPMD programming model implemented on SIMD hardware
SPMD

- Single procedure/program, multiple data
  - This is a programming model rather than computer organization

- Each processing element executes the same procedure, except on different data elements
  - Procedures can synchronize at certain points in program, e.g. barriers

- Essentially, multiple instruction streams execute the same program
  - Each program/procedure can 1) execute a different control-flow path, 2) work on different data, at run-time
  - Many scientific applications programmed this way and run on MIMD computers (multiprocessors)
  - Modern GPUs programmed in a similar way on a SIMD computer
SPMD Execution on SIMD Hardware

- NVIDIA calls this “Single Instruction, Multiple Thread” ("SIMT") execution

Slide credit: Tor Aamodt
Control Flow Problem in GPUs/SIMD

- GPU uses SIMD pipeline to save area on control logic.
  - Group scalar threads into warps

- Branch divergence occurs when threads inside warps branch to different execution paths.

Slide credit: Tor Aamodt
Branch Divergence Handling (I)

Thread Warp

<table>
<thead>
<tr>
<th>Thread</th>
<th>Thread</th>
<th>Thread</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Stack

<table>
<thead>
<tr>
<th>TOS</th>
<th>Reconv. PC</th>
<th>Next PC</th>
<th>Active Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>E</td>
<td>E</td>
<td>1111</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>E</td>
<td>0110</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>1001</td>
</tr>
</tbody>
</table>

Time

Slide credit: Tor Aamodt
Branch Divergence Handling (II)

A;
if (some condition) {
  B;
} else {
  C;
}
D;

Control Flow Stack

<table>
<thead>
<tr>
<th></th>
<th>Next PC</th>
<th>Recv PC</th>
<th>Amask</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>--</td>
<td></td>
<td>1111</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td></td>
<td>1110</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td></td>
<td>0001</td>
</tr>
</tbody>
</table>

Execution Sequence

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Slide credit: Tor Aamodt
Dynamic Warp Formation

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)
- **Form new warp at divergence**
  - Enough threads branching to each path to create full new warps
Dynamic Warp Formation/Merging

- **Idea:** Dynamically merge threads executing the same instruction (after branch divergence)

Dynamic Warp Formation Example

Baseline

Dynamic Warp Formation

Legend

A
Execution of Warp x at Basic Block A

A
Execution of Warp y at Basic Block A

D
A new warp created from scalar threads of both Warp x and y executing at Basic Block D

A new warp created from scalar threads of both Warp x and y executing at Basic Block D

Slide credit: Tor Aamodt
What About Memory Divergence?

- Modern GPUs have caches
- Ideally: Want all threads in the warp to hit (without conflicting with each other)
- Problem: One thread in a warp can stall the entire warp if it misses in the cache.

- Need techniques to
  - Tolerate memory divergence
  - Integrate solutions to branch and memory divergence
NVIDIA GeForce GTX 285

- NVIDIA-speak:
  - 240 stream processors
  - “SIMT execution”

- Generic speak:
  - 30 cores
  - 8 SIMD functional units per core
NVIDIA GeForce GTX 285 “core”

Slide credit: Kayvon Fatahalian
**NVIDIA GeForce GTX 285 “core”**

- Groups of 32 **threads** share instruction stream (each group is a Warp)
- Up to 32 warps are simultaneously interleaved
- Up to 1024 thread contexts can be stored

Slide credit: Kayvon Fatahalian
30 cores on the GTX 285: 30,720 threads

Slide credit: Kayvon Fatahalian
VLIW and DAE
Remember: SIMD/MIMD Classification 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?

- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
SISD Parallelism Extraction Techniques

- We have already seen
  - Superscalar execution
  - Out-of-order execution

- Are there simpler ways of extracting SISD parallelism?
  - VLIW (Very Long Instruction Word)
  - Decoupled Access/Execute
VLIW
VLIW (Very Long Instruction Word)

- A very long instruction word consists of multiple independent instructions packed together by the compiler
  - Packed instructions can be logically unrelated (contrast with SIMD)

**Idea:** Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction

**Traditional Characteristics**

- Multiple functional units
- Each instruction in a bundle executed in lock step
- Instructions in a bundle statically aligned to be directly fed into the functional units
VLIW Concept

  - ELI: Enormously longword instructions (512 bits)
SIMD Array Processing vs. VLIW

- Array processor

![Diagram showing SIMD array processing vs. VLIW](image)
VLIW Philosophy

- Philosophy similar to RISC (simple instructions and hardware)
  - Except multiple instructions in parallel

- RISC (John Cocke, 1970s, IBM 801 minicomputer)
  - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
    - And, to reorder simple instructions for high performance
  - Hardware does little translation/decoding → very simple

- VLIW (Fisher, ISCA 1983)
  - Compiler does the hard work to find instruction level parallelism
  - Hardware stays as simple and streamlined as possible
    - Executes each instruction in a bundle in lock step
    - Simple → higher frequency, easier to design
VLIW Philosophy (II)

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.
Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors)
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones
VLIW Tradeoffs

- **Advantages**
  + No need for dynamic scheduling hardware → simple hardware
  + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
  + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

- **Disadvantages**
  -- Compiler needs to find N independent operations
    -- If it cannot, inserts NOPs in a VLIW instruction
    -- Parallelism loss AND code size increase
  -- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
  -- Lockstep execution causes independent operations to stall
    -- No instruction can progress until the longest-latency instruction completes
VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance
  - Too many NOPs (not enough parallelism discovered)
  - Static schedule intimately tied to microarchitecture
    - Code optimized for one generation performs poorly for next
  - No tolerance for variable or long-latency operations (lock step)

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)
  - Enable code optimizations
++ VLIW successful in embedded markets, e.g. DSP
DAE
Decoupled Access/Execute

- Motivation: Tomasulo’s algorithm too complex to implement
  - 1980s before HPS, Pentium Pro

- Idea: Decouple operand access and execution via two separate instruction streams that communicate via ISA-visible queues.

Decoupled Access/Execute (II)

- Compiler generates two instruction streams (A and E)
  - Synchronizes the two upon control flow instructions (using branch queues)

\[ q = 0.0 \]
\[ \text{Do } 1 \ k = 1, 400 \]
\[ x(k) = q + y(k) \times (r \times z(k+10) + t \times z(k+11)) \]

**Fig. 2a.** Lawrence Livermore Loop 1 (HYDRO EXCERPT)

<table>
<thead>
<tr>
<th>Access</th>
<th>Execute</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>AEQ + z + 10, A2</td>
<td>X4 + X2 *f AEQ</td>
</tr>
<tr>
<td>AEQ + z + 11, A2</td>
<td>X3 + X5 *f AEQ</td>
</tr>
<tr>
<td>AEQ + y, A2</td>
<td>X6 + X3 +f X4</td>
</tr>
<tr>
<td>A7 + A7 + 1</td>
<td>EAQ + AEQ *f X6</td>
</tr>
<tr>
<td>x, A2 + EAQ</td>
<td>.</td>
</tr>
<tr>
<td>A2 + A2 + A3</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

**Fig. 2b.** Compilation onto CRAY-1-like architecture

**Fig. 2c.** Access and execute programs for straight-line section of loop
Decoupled Access/Execute (III)

**Advantages:**

+ Execute stream can run ahead of the access stream and vice versa
  
  + If A takes a cache miss, E can perform useful work
  
  + If A hits in cache, it supplies data to lagging E
  
  + Queues reduce the number of required registers
  
+ Limited out-of-order execution without wakeup/select complexity

**Disadvantages:**

-- Compiler support to partition the program and manage queues
  
  -- Determines the amount of decoupling
  
-- Branch instructions require synchronization between A and E
  
-- Multiple instruction streams (can be done with a single one, though)
Astronautics ZS-1

- Single stream steered into A and X pipelines
- Each pipeline in-order


Astronautics ZS-1 Instruction Scheduling

- Dynamic scheduling
  - A and X streams are issued/executed independently
  - Loads can bypass stores in the memory unit (if no conflict)
  - Branches executed early in the pipeline
    - To reduce synchronization penalty of A/X streams
    - Works only if the register a branch sources is available

- Static scheduling
  - Move compare instructions as early as possible before a branch
    - So that branch source register is available when branch is decoded
  - Reorder code to expose parallelism in each stream
  - Loop unrolling:
    - Reduces branch count + exposes code reordering opportunities
Loop Unrolling

- Idea: Replicate loop body multiple times within an iteration
  + Reduces loop maintenance overhead
    - Induction variable increment or loop condition test
  + Enlarges basic block (and analysis scope)
    - Enables code optimization and scheduling opportunities
-- What if iteration count not a multiple of unroll factor? (need extra code to detect this)
-- Increases code size
Systolic Arrays
Why Systolic Architectures?

- **Idea:** Data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory.

- **Similar to an assembly line**
  - Different people work on the same car
  - Many cars are assembled simultaneously
  - Can be two-dimensional

- **Why?** Special purpose accelerators/architectures need
  - Simple, regular designs (keep # unique parts small and regular)
  - High concurrency → high performance
  - Balanced computation and I/O (memory access)
Systolic Architectures


Figure 1. Basic principle of a systolic system.
Systolic Architectures

- Basic principle: Replace a single PE with a regular array of PEs and carefully orchestrate flow of data between the PEs to achieve high throughput w/o increasing memory bandwidth requirements.

- Differences from pipelining:
  - Array structure can be non-linear and multi-dimensional
  - PE connections can be multidirectional (and different speed)
  - PEs can have local memory and execute kernels (rather than a piece of the instruction)
Systolic Computation Example

- Convolution
  - Used in filtering, pattern matching, correlation, polynomial evaluation, etc ...
  - Many image processing tasks

\[ \text{Given} \text{ the sequence of weights } \{w_1, w_2, \ldots, w_k\} \]
\[ \text{and the input sequence } \{x_1, x_2, \ldots, x_n\}, \]
\[ \text{compute} \text{ the result sequence } \{y_1, y_2, \ldots, y_{n+1-k}\} \]
\[ \text{defined by} \]
\[ y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1} \]
Systolic Computation Example: Convolution

- $y_1 = w_1x_1 + w_2x_2 + w_3x_3$
- $y_2 = w_1x_2 + w_2x_3 + w_3x_4$
- $y_3 = w_1x_3 + w_2x_4 + w_3x_5$

Figure 8. Design $W_1$: systolic convolution array (a) and cell (b) where $w_i$'s stay and $x_i$'s and $y_i$'s move systolically in opposite directions.
Worthwhile to implement adder and multiplier separately to allow overlapping of add/mul executions.
More Programmability

- Each PE in a systolic array
  - Can store multiple “weights”
  - Weights can be selected on the fly
  - Eases implementation of, e.g., adaptive filtering

- Taken further
  - Each PE can have its own data and instruction memory
  - Data memory → to store partial/temporary results, constants
  - Leads to stream processing, pipeline parallelism
    - More generally, staged execution
Pipeline Parallelism

Figure 1. (a) The code of a loop, (b) Each iteration is split into 3 pipeline stages: A, B, and C. Iteration i comprises Ai, Bi, Ci. (c) Sequential execution of 4 iterations. (d) Parallel execution of 6 iterations using pipeline parallelism on a three-core machine. Each stage executes on one core.
File Compression Example

Figure 3. File compression algorithm executed using pipeline parallelism
Systolic Array

- **Advantages**
  - Makes multiple uses of each data item → reduced need for fetching/refetching
  - High concurrency
  - Regular design (both data and control flow)

- **Disadvantages**
  - Not good at exploiting irregular parallelism
  - Relatively special purpose → need software, programmer support to be a general purpose model
The WARP Computer

- HT Kung, CMU, 1984-1988
- Linear array of 10 cells, each cell a 10 Mflop programmable processor
- Attached to a general purpose host machine
- HLL and optimizing compiler to program the systolic array
- Used extensively to accelerate vision and robotics tasks

The WARP Computer

Figure 1: Warp system overview
Figure 2: Warp cell data path
Systolic Arrays vs. SIMD

- Food for thought...
Some More Recommended Readings

- **Recommended:**