FINAL REVIEW PART1

Howard Mao
Pipelining
“Iron Law” of Processor Performance

Instructions per program depends on source code, compiler technology, and ISA

Cycles per instructions (CPI) depends on ISA and \( \mu \)-architecture

Time per cycle depends upon the \( \mu \)-architecture and base technology

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>CPI</th>
<th>cycle time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcoded</td>
<td>&gt;1</td>
<td>short</td>
</tr>
<tr>
<td>Single-cycle unpipelined</td>
<td>1</td>
<td>long</td>
</tr>
<tr>
<td>Pipelined</td>
<td>1</td>
<td>short</td>
</tr>
</tbody>
</table>
Instructions interact with each other in pipeline

- An instruction in the pipeline may need a resource being used by another instruction in the pipeline → structural hazard

- An instruction may depend on something produced by an earlier instruction
  - Dependence may be for a data value → data hazard
  - Dependence may be for the next instruction’s address → control hazard (branches, exceptions)

- Handling hazards generally introduces bubbles into pipeline and reduces ideal CPI > 1
Interlocking Versus Bypassing

```
add x1, x3, x5
sub x2, x1, x4
```

Instruction interlocked in decode stage

Bypass around ALU with no bubbles
Example Bypass Path
[ Assumes data written to registers in a W cycle is readable in parallel D cycle (dotted line). Extra write data register and bypass paths required if this is not possible. ]
Exception Handling 5-Stage Pipeline

1. **PC**
2. **Inst. Mem**
3. **D**
4. **Decode**
5. **E**
6. **M**
7. **Data Mem**
8. **W**

- **Illegal Opcode**
- **Overflow**
- **Data address Exceptions**
- **Asynchronous Interrupts**
- **Select Handler PC**
- **Kill F Stage**
- **Kill D Stage**
- **Kill E Stage**
- **Commit Point**
- **EPC**
- **Kill Writeback**
In-Order Superscalar Pipeline

- Fetch two instructions per cycle; issue both simultaneously if one is integer/memory and other is floating point
- Inexpensive way of increasing throughput, examples include Alpha 21064 (1992) & MIPS R5000 series (1996)
- Same idea can be extended to wider issue by duplicating functional units (e.g. 4-issue UltraSPARC & Alpha 21164) but regfile ports and bypassing costs grow quickly
Out-of-Order Processor
Scoreboard for In-order Issues

Busy[FU#] : a bit-vector to indicate FU’s availability.
(FU = Int, Add, Mult, Div)
These bits are hardwired to FU's.

WP[reg#] : a bit-vector to record the registers for which writes are pending.
These bits are set by Issue stage and cleared by WB stage

Issue checks the instruction (opcode dest src1 src2) against the scoreboard (Busy & WP) to dispatch

FU available?  Busy[FU#]
RAW? WP[src1] or WP[src2]
WAR? cannot arise
WAW? WP[dest]
Renaming Structures

Renaming table & regfile

Reorder buffer

Replacing the tag by its value is an expensive operation

- Instruction template (i.e., tag t) is allocated by the Decode stage, which also associates tag with register in regfile
- When an instruction completes, its tag is deallocated
IBM 360/91 Floating-Point Unit
R. M. Tomasulo, 1967

Distribute instruction templates by functional units

load buffers (from memory)

store buffers (to memory)

Common bus ensures that data is made available immediately to all the instructions waiting for it.
Match tag, if equal, copy value & set presence “p”.

Floating-Point Regfile
In-Order Commit for Precise Traps

- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state
**“Data-in-ROB” Design**
*(HP PA8000, Pentium Pro, Core2Duo, Nehalem)*

<table>
<thead>
<tr>
<th></th>
<th>Opcode</th>
<th>p</th>
<th>Tag</th>
<th>Src1</th>
<th>p</th>
<th>Tag</th>
<th>Src2</th>
<th>p</th>
<th>Reg</th>
<th>Result</th>
<th>Except?</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Managed as circular buffer in program order, new instructions dispatched to free slots, oldest instruction committed/reclaimed when done ("p" bit set on result)
- Tag is given by index in ROB (Free pointer value)
- In dispatch, non-busy source operands read from architectural register file and copied to Src1 and Src2 with presence bit "p" set. Busy operands copy tag of producer and clear “p” bit.
- Set valid bit “v” on dispatch, set issued bit “i” on issue
- On completion, search source tags, set “p” bit and copy data into src on tag match. Write result and exception flags to ROB.
- On commit, check exception status, and copy result into architectural register file if no trap.
- On trap, flush machine and ROB, set free=oldest, jump to handler
Data Movement in Data-in-ROB Design

- Read operands during decode
- Write sources in dispatch
- Read operands at issue
- Write results at commit
- Read results for commit
- Bypass newer values at dispatch
- Write results at completion
Unified Physical Register File
(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement

Diagram:
- Decode Stage
  - Register Mapping
  - Read operands at issue
- Unified Physical Register File
- Committed Register Mapping
  - Write results at completion
Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.

- During decode, store buffer slot allocated in program order
- Stores split into “store address” and “store data” micro-operations
- “Store address” execution writes tag
- “Store data” execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit
Conservative O-o-O Load Execution

\[
\text{sd } x1, (x2) \\
\text{ld } x3, (x4)
\]

- Can execute load before store, if addresses known and \( x4 \neq x2 \)
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don’t execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)
Address Speculation

\[ \text{sd } x1, \ (x2) \]
\[ \text{ld } x3, \ (x4) \]

- Guess that \( x4 \neq x2 \)
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find \( x4 == x2 \), squash load and all following instructions

\Rightarrow \text{Large penalty for inaccurate address speculation}
Branch Prediction
Overall probability a branch is taken is \(~60\text{-}70\%)\) but:

\begin{figure}
\centering
\begin{tikzpicture}
    \node[decision] (d1) {
        \text{backward}
    };
    \node[decision, right of=d1] (d2) {
        \text{forward}
    };
    \path[->] (d1) edge node {90\%} (d2);
    \path[->] (d2) edge node {50\%} (d1);
\end{tikzpicture}
\end{figure}

ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110
\begin{itemize}
    \item \texttt{bne0 (preferred taken)}
    \item \texttt{beq0 (not taken)}
\end{itemize}

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64
\begin{itemize}
    \item typically reported as \(~80\%)\) accurate
Branch Prediction Bits

- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!

**BP state:**

\[(\text{predict } \text{take/} \neg \text{take}) \times (\text{last prediction } \text{right/} \text{wrong})\]
Branch History Table (BHT)

4K-entry BHT, 2 bits/entry, ~80-90% correct predictions
Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)
Branch Target Buffer (BTB)

- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded
Subroutine Return Stack

Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```plaintext
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```

Push call address when function call executed

Pop return address when subroutine return decoded

```
&fd()
&fc()
&fb()
```

$k$ entries (typically $k=8$-16)
Multithreading
Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

One way is to interleave execution of instructions from different program threads on same pipeline

Interleave 4 threads, T1-T4, on **non-bypassed** 5-stage pipe

- T1: LD x1, 0(x2)
- T2: ADD x7, x1, x4
- T3: XORI x5, x4, 12
- T4: SD 0(x7), x5
- T1: LD x5, 12(x1)

Prior instruction in a thread always completes write-back before next instruction in same thread reads register file
Thread Scheduling Policies

- **Fixed interleave** (*CDC 6600 PPUs, 1964*)
  - Each of N threads executes one instruction every N cycles
  - If thread not ready to go in its slot, insert pipeline bubble

- **Software-controlled interleave** (*TI ASC PPUs, 1971*)
  - OS allocates S pipeline slots amongst N threads
  - Hardware performs fixed interleave over S slots, executing whichever thread is in that slot

- **Hardware-controlled thread scheduling** (*HEP, 1982*)
  - Hardware keeps track of which threads are ready to go
  - Picks next thread to execute based on hardware priority scheme
Coarse-Grain Multithreading

- Tera MTA designed for supercomputing applications with large data sets and low locality
  - No data cache
  - Many parallel threads needed to hide large memory latency

- Other applications are more cache friendly
  - Few pipeline bubbles if cache mostly has hits
  - Just add a few threads to hide occasional cache miss latencies
  - Swap threads on cache misses
Superscalar Machine Efficiency

- Instruction issue
- Time

Issue width

- Completely idle cycle, (vertical waste)
- Partially filled cycle, i.e., IPC < 4, (horizontal waste)
Cycle-by-cycle interleaving removes vertical waste, but leaves some horizontal waste.
What is the effect of splitting into multiple processors?

- reduces horizontal waste,
- leaves some vertical waste, and
- puts upper limit on peak throughput of each thread.
SMT adaptation to parallelism type

For regions with high thread-level parallelism (TLP) entire machine width is shared by all threads.

For regions with low thread-level parallelism (TLP) entire machine width is available for instruction-level parallelism (ILP).
Icount Choosing Policy

Fetch from thread with the least instructions in flight.

Why does this enhance throughput?
Summary: Multithreaded Categories
CS152 Administrivia

- Lab 5 due Friday April 27
- Friday April 27, discussion will cover multiprocessor review
- Final exam, Tuesday May 8, 306 Soda, 11:30am-2:30pm
CS252 Administrivia

- Project presentations, May 2nd, 2:30pm-5pm, 511 Soda
- Final project papers, May 11th
FINAL REVIEW PART2

Donggyu Kim
Memory Hierarchy

Why does it matter?
Processor-DRAM Gap (latency)

Performance

Time


μProc 60%/year

CPU

Processor-Memory Performance Gap: (growing 50%/yr)

DRAM 7%/year

Four-issue 3GHz superscalar accessing 100ns DRAM could execute 1,200 instructions during time for one memory access!
Physical Size Affects Latency

- Signals have further to travel
- Fan out to more locations
- More complex address decoder
Memory Hierarchy

- **capacity**: Register $\ll$ SRAM $\ll$ DRAM
- **latency**: Register $\ll$ SRAM $\ll$ DRAM
- **bandwidth**: on-chip $>>$ off-chip

On a data access:

- *if* data $\in$ fast memory $\Rightarrow$ low latency access (*SRAM*)
- *if* data $\notin$ fast memory $\Rightarrow$ high latency access (*DRAM*)
Performance Models

- **Average Memory Access Time (AMAT)**
  - Hit Time + Miss Rate × Miss Penalty
- **Iron Law**
  \[
  \frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}
  \]
  \[
  CPI_{total} = CPI_{CPU} + \sum_{e: \text{event}} Misses/\text{Instruction}_e \times \text{MissPenalty}_e
  \]
- **Goal**
  - Reduce the hit time
  - Reduce the miss rate
  - Reduce the miss penalty
  - *Increase the cache bandwidth*
    → effectively decrease the hit time
Causes of Cache Misses: The 3+1 C’s

Compulsory: first reference to a line (a.k.a. cold start misses)
   – misses that would occur even with infinite cache

Capacity: cache is too small to hold all data needed by the program
   – misses that would occur even under perfect replacement policy

Conflict: misses that occur because of collisions due to line-placement strategy
   – misses that would not occur with ideal full associativity

Coherence: misses that occur when a line is invalidated due to a remote write to another word (false sharing)
   – misses that would not occur without shared lines
Effect of Cache Parameters on Performance

- Larger cache size
  + reduces capacity and conflict misses
  - hit time will increase

- Higher associativity
  + reduces conflict misses
  - may increase hit time

- Larger line size
  + reduces compulsory misses
  - increases conflict & **coherence** misses and miss penalty
Multilevel Caches

**Problem**: A memory cannot be large and fast

**Solution**: Increasing sizes of cache at each level

Local miss rate = misses in cache / accesses to cache

Global miss rate = misses in cache / CPU memory accesses

Misses per instruction = misses in cache / number of instructions
Prefetching

- Speculate on future instruction and data accesses and fetch them into cache(s)
  - Instruction accesses easier to predict than data accesses

- Varieties of prefetching
  - Hardware prefetching
  - Software prefetching
  - Mixed schemes

- Issues
  - Timeliness – not late and not too early
  - Cache and bandwidth pollution
Hardware Instruction Prefetching

Instruction prefetch in Alpha AXP 21064

- Fetch two lines on a miss; the requested line (i) and the next consecutive line (i+1)
- Requested line placed in cache, and next line in instruction stream buffer
- If miss in cache but hit in stream buffer, move stream buffer line into cache and prefetch next line (i+2)
Reducing Write Hit Time (or Bandwidth)

**Problem:** Writes take two cycles in memory stage, one cycle for tag check plus one cycle for data write if hit

**Solutions:**
- Design data RAM that can perform read and write in one cycle, restore old value after tag miss
- Fully-associative (CAM Tag) caches: Word line only enabled if hit
- Pipelined writes: Hold write data for store in single buffer ahead of cache, write cache data during next store’s tag check
Write Buffer to Reduce Read Miss Penalty

Processor is not stalled on writes, and read misses can go ahead of write to main memory

**Problem:** Write buffer may hold updated value of location needed by a read miss

**Simple solution:** on a read miss, wait for the write buffer to go empty

**Faster solution:** Check write buffer addresses against read miss addresses, if no match, allow read miss to go ahead of writes, else, return value in write buffer

→ *This breaks sequential consistency (SC)*!
## Cache Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Hit Time</th>
<th>Miss Penalty</th>
<th>Miss Rate</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small and simple caches</td>
<td>Decrease</td>
<td>–</td>
<td>Increase (Capacity)</td>
<td>Decrease</td>
</tr>
<tr>
<td>Multi-level caches</td>
<td>–</td>
<td>Increase</td>
<td>Decrease</td>
<td>Increase</td>
</tr>
<tr>
<td>Pipelined writes</td>
<td>Decrease</td>
<td>–</td>
<td>–</td>
<td>Increase</td>
</tr>
<tr>
<td>Write buffer</td>
<td>–</td>
<td>Decrease</td>
<td>–</td>
<td>Increase</td>
</tr>
<tr>
<td>Sub-blocks</td>
<td>–</td>
<td>Decrease</td>
<td>Increase (Compulsory)</td>
<td>Increase</td>
</tr>
<tr>
<td>Code optimization</td>
<td>–</td>
<td>–</td>
<td>Decrease</td>
<td>–</td>
</tr>
<tr>
<td>Compiler prefetching</td>
<td>–</td>
<td>–</td>
<td>Decrease (Compulsory, But possible cache / bandwidth pollution)</td>
<td>Increase (non-blocking cache, additional instructions)</td>
</tr>
<tr>
<td>Hardware prefetching (Stream buffer)</td>
<td>–</td>
<td>–</td>
<td>Decrease (Compulsory)</td>
<td>Increase</td>
</tr>
<tr>
<td>Victim cache</td>
<td>–</td>
<td>Increase</td>
<td>Decrease (Conflict, compared to direct-mapped)</td>
<td>Decrease (Compared to set-associative)</td>
</tr>
</tbody>
</table>
Virtual Memory

Why do we care hardware support?
Page Tables Live in Memory

Virtual Address Space
Pages for Job 1

Virtual Address Space
Pages for Job 2

Physical
Memory
Pages

How many memory accesses per instruction?

Simple linear page tables are too large, so hierarchical page tables are commonly used (see later)

Common for modern OS to place page tables in kernel’s virtual memory (page tables can be swapped to secondary storage)
• Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient
Translation-Lookaside Buffers (TLB)

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

- TLB hit $\Rightarrow$ *Single-Cycle Translation*
- TLB miss $\Rightarrow$ *Page-Table Walk to refill*

```
(VPN = virtual page number)
(PPN = physical page number)
```

```
<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>D</th>
<th>tag</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

virtual address

VPN
offset

physical address

PPN
offset

hit?```
Address Translation: putting it all together

Virtual Address

TLB Lookup

- miss
  - Page Table Walk
    - the page is
      - $\notin$ memory
        - Page Fault (OS loads page)
      - $\in$ memory
        - Update TLB

- hit
  - Protection Check
    - denied
      - Protection Fault
    - permitted
      - Physical Address (to cache)

Where?

SEGFAULT
Handling a TLB Miss

Software (MIPS, Alpha)

- TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged “untranslated” addressing mode used for walk.
- Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.

Hardware (SPARC v8, x86, PowerPC, RISC-V)

- A memory management unit (MMU) walks the page tables and reloads the TLB.
- If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page Fault exception for the original instruction.

NOTE: A given ISA can use either TLB miss strategy
Virtual-Address Caches

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- aliasing problems due to the sharing of pages (-)
- maintaining cache coherence (-)

Alternative: place the cache before the TLB (StrongARM)
Concurrent Access to TLB & Cache (Virtual Index/Physical Tag)

Index L is available without consulting the TLB

\[ \textit{cache and TLB accesses can begin simultaneously!} \]

Tag comparison is made after both accesses are completed

\[ \text{Condition: } L + b \leq k \]
VLIW Machine
Simple HW + Smart Compilers
Performance Models, Again

\[
\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}
\]

\[
CPI_{\text{total}} = CPI_{\text{CPU}} + \sum_{e: \text{event}} \text{Misses/Instruction}_e \times \text{MissPenalty}_e
\]

• Goal
  • Take advance of instruction-level parallelism (ILP)
    • Limited by data dependencies
    • Varies across applications
  • Reschedule instructions
    • By hardware (OoO processors)
    • By software (compiler optimizations)
Out-of-Order Control Complexity: MIPS R10000

[SGI/MIPS Technologies Inc., 1995]
VLIW: Very Long Instruction Word

- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- HW doesn’t check data dependencies across instructions (no pipeline stalls)
VLIW Compiler Responsibilities

- Schedule operations to maximize parallel execution

- Guarantees intra-instruction parallelism

- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs
for (i=0; i<N; i++)

Loop Execution

Compile

loop:    fld f1, 0(x1)
         add x1, 8
         fadd f2, f0, f1
         fsd f2, 0(x2)
         add x2, 8
         bne x1, x3,

How many FP ops/cycle?

1 fadd / 8 cycles = 0.125
Scheduling Loop Unrolled Code

Unroll 4 ways

| loop: | fld f1, 0(x1) | fld f2, 8(x1) | fld f3, 16(x1) | fld f4, 24(x1) | add x1, 32 | fadd f5, f0, f1 | fadd f6, f0, f2 | fadd f7, f0, f3 | fadd f8, f0, f4 | fsd f5, 0(x2) | fsd f6, 8(x2) | fsd f7, 16(x2) | fsd f8, 24(x2) | add x2, 32 | bne x1, x3, loop |

Schedule

<table>
<thead>
<tr>
<th>Int1</th>
<th>Int2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36
for (i=0; i<N; i++)

Compile

loop:
    fld f1, 0(x1)
    add x1, 8
    fadd f2, f0, f1
    fsd f2, 0(x2)
    add x2, 8
    bne x1, x3,

loop
Software Pipelining

How many FLOPS/cycle?

1 fadd / 3 cycles = 0.333
### Software Pipelining + Loop Unrolling

#### Unroll 4 ways first

**loop:**
- `fld f1, 0(x1)`
- `fld f2, 8(x1)`
- `fld f3, 16(x1)`
- `fld f4, 24(x1)`
- `add x1, 32`
- `fadd f5, f0, f1`
- `fadd f6, f0, f2`
- `fadd f7, f0, f3`
- `fadd f8, f0, f4`
- `fsd f5, 0(x2)`
- `fsd f6, 8(x2)`
- `fsd f7, 16(x2)`
- `add x2, 32`
- `fsd f8, -8(x2)`
- `bne x1, x3, loop`

#### Table:

<table>
<thead>
<tr>
<th></th>
<th>Int1</th>
<th>Int2</th>
<th>M1</th>
<th>M2</th>
<th>FP+</th>
<th>FPx</th>
</tr>
</thead>
<tbody>
<tr>
<td>prolog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop</td>
<td>fld f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f3</td>
<td></td>
<td></td>
<td></td>
<td>fadd f7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fadd f8</td>
</tr>
<tr>
<td></td>
<td>fadd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>add x2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>epilog</td>
<td>fld f4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fsd f7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>bne</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>fadd f8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**How many FLOPS/cycle?**

4 fadds / 4 cycles = 1
Trace Scheduling [Fisher, Ellis]

- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use *profiling feedback* or compiler heuristics to find common branch paths
- Schedule whole “trace” at once
- Add fixup code to cope with branches jumping out of trace
Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity (compiler runtime)

Despite several attempts, VLIW has failed in general-purpose computing arena (so far).
  - More complex VLIW architectures are close to in-order superscalar in complexity, no real advantage on large complex apps.

Successful in embedded DSP market
  - Simpler VLIWs with more constrained environment, friendlier code.
SIMD vs. Vector vs. GPU
Towards Data-Level Parallelism (DLP)
Performance Models, Again

\[
\frac{Time}{Program} = \frac{Instructions}{Program} \times \frac{Cycles}{Instruction} \times \frac{Time}{Cycle}
\]

\[
CPI_{total} = CPI_{CPU} + \sum_{e : \text{event}} Misses/\text{Instruction}_e \times \text{MissPenalty}_e
\]

• Goal
  • Take advantage of data-level parallelism
    • Same operations on large data arrays
    • Present in scientific/machine-learning applications
  • Increase the bandwidth of each instruction
    • Less dynamic instruction counts while keeping CPI & cycle time constant
Vector Programming Model

**Scalar Registers**

- x31
- x0

**Vector Registers**

- v31
- v0

**Vector Length Register**

- v1

**Vector Arithmetic Instructions**

- vadd v3, v1, v2

**Vector Load and Store Instructions**

- vlds v1, x1, x2

**Memory**

- Base, x1
- Stride, x2
## Vector Code Example

<table>
<thead>
<tr>
<th># C code</th>
<th># Scalar Code</th>
<th># Vector Code</th>
</tr>
</thead>
</table>
| for (i=0; i<64; i++)
  C[i] = A[i] + B[i]; | li x4, 64
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fadd.d f3,f1,f2
  fsd f3, 0(x3)
  addi x1, 8
  addi x2, 8
  addi x3, 8
  subi x4, 1
  bnez x4, loop | li x4, 64
setvl x5, x4
vld v1, x1
vld v2, x2
vadd v3,v1,v2
vst v3, x3 |
Vector Instruction Set Advantages

- **Compact**
  - one short instruction encodes N operations

- **Expressive**
  - Unit-stride load/store for a contiguous block
  - Strided load/store for a known pattern
  - Indexed load/store for indirect accesses

- **Scalable**
  - Same code on more parallel pipelines (lanes)
T0 Vector Microprocessor (UCB/ICSI, 1995)

Built by Prof. Krste Asanovic

Vector register elements striped over lanes
Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
HW/SW Techniques for Vector Machine

• Chaining
  • Bypassing the results to the dependent instructions to start the following instructions as soon as possible

• Stripmining
  • Compiler optimization that breaks loops to fit in vector registers

• Vector masks
  • To support conditional code for vectorizable loop

• Vector reduction
  • For reduction operations across vector elements

• Gather/Scatter
  • To support indirect (indexed) memory operations for vectorizable loop
RISC-V Vector ISA

- Enjoy Lab4?
- Ready for another coding question?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vld vd, offset(rs1), vm</code></td>
<td><code>vd[i] := mem[(rs1) + offset + i]</code></td>
</tr>
<tr>
<td><code>vst vs3, offset(rs1), vm</code></td>
<td><code>mem[(rs1) + offset + i] := vs3[i]</code></td>
</tr>
<tr>
<td><code>vlds vd, offset(rs1), rs2, vm</code></td>
<td><code>vd[i] := mem[(rs1) + offset + i * rs2]</code></td>
</tr>
<tr>
<td><code>vstse vs3, offset(rs1), rs2, vm</code></td>
<td><code>mem[rs1 + offset + i * (rs2)] := vs3[i]</code></td>
</tr>
<tr>
<td><code>vldx vd, offset(rs1), vs2, vm</code></td>
<td><code>vd[i] := mem[(rs1) + offset + vs2[i]]</code></td>
</tr>
<tr>
<td><code>vstx vs3, offset(rs1), vs2, vm</code></td>
<td><code>mem[(rs1) + offset + vs2[i]] := vs3[i]</code></td>
</tr>
<tr>
<td><code>vadd vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] + vs2[i]</code></td>
</tr>
<tr>
<td><code>vsub vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] - vs2[i]</code></td>
</tr>
<tr>
<td><code>vmul vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] * vs2[i]</code></td>
</tr>
<tr>
<td><code>vdiv vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] / vs2[i]</code></td>
</tr>
<tr>
<td><code>vrem vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] % vs2[i]</code></td>
</tr>
<tr>
<td><code>vmax vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := max(vs1[i], vs2[i])</code></td>
</tr>
<tr>
<td><code>vmin vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := min(vs1[i], vs2[i])</code></td>
</tr>
<tr>
<td><code>vsl vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] &lt;&lt; vs2[i]</code></td>
</tr>
<tr>
<td><code>vsr vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] &gt;&gt; vs2[i]</code></td>
</tr>
<tr>
<td><code>vseq vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] == vs2[i] ? 1 : 0</code></td>
</tr>
<tr>
<td><code>vne vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] != vs2[i] ? 1 : 0</code></td>
</tr>
<tr>
<td><code>vslt vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] &lt; vs2[i] ? 1 : 0</code></td>
</tr>
<tr>
<td><code>vsgte vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs1[i] &gt;= vs2[i] ? 1 : 0</code></td>
</tr>
<tr>
<td><code>vaddi vd, vs1, imm, vm</code></td>
<td><code>vd[i] := vs1[i] + imm</code></td>
</tr>
<tr>
<td><code>vsl i vd, vs1, imm, vm</code></td>
<td><code>vd[i] := vs1[i] &lt;&lt; imm</code></td>
</tr>
<tr>
<td><code>vsti vd, vs1, imm, vm</code></td>
<td><code>vd[i] := vs1[i] &gt;&gt; imm</code></td>
</tr>
<tr>
<td><code>vmadd vd, vs1, vs2, vs3, vm</code></td>
<td><code>vd[i] := vs1[i] * vs2[i] + vs3[i]</code></td>
</tr>
<tr>
<td><code>vmsub vd, vs1, vs2, vs3, vm</code></td>
<td><code>vd[i] := vs1[i] * vs2[i] - vs3[i]</code></td>
</tr>
<tr>
<td><code>vmadd vd, vs1, vs2, vs3, vm</code></td>
<td><code>vd[i] := -(vs1[i] * vs2[i] + vs3[i])</code></td>
</tr>
<tr>
<td><code>vmsub vd, vs1, vs2, vs3, vm</code></td>
<td><code>vd[i] := -(vs1[i] * vs2[i] - vs3[i])</code></td>
</tr>
<tr>
<td><code>vslide vd, vs1, rs2, vm</code></td>
<td><code>vd[i] := 0 ≤ (rs2) + i &lt; VL ? vs1[(rs2) + i] : 0</code></td>
</tr>
<tr>
<td><code>vinsert vd, vs1, rs2, vm</code></td>
<td><code>vd[rs2[i]] := (rs1)</code></td>
</tr>
<tr>
<td><code>vexTRACT rd, vs1, rs2, vm</code></td>
<td><code>(rd) := (i for i in range(0, VL) if LSB(vsi[1]) == i + [-1] &amp; 0)</code></td>
</tr>
<tr>
<td><code>vMfirst rd, vs1</code></td>
<td><code>(rd) := len[i for i in range(0, VL) if LSB(vsi[1]) == 1])</code></td>
</tr>
<tr>
<td><code>vselect vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := vs2[i] &lt; VL ? vs1[vs2[i]] : 0</code></td>
</tr>
<tr>
<td><code>vmerge vd, vs1, vs2, vm</code></td>
<td><code>vd[i] := LSB(vsi[1]) == vs2[i] : vs1[i]</code></td>
</tr>
</tbody>
</table>

Table 2: RISC-V Vector Instructions
Packed SIMD vs. Vectors?

• Big problem of SIMD
  • *Vector lengths are encoded in instructions*
  • Need different instructions for different vector lengths (recall 61C labs)
  • Rewrite your code for wider vectors!
  • Vector: the VL register + stripmining
## GPU vs. Vectors?

<table>
<thead>
<tr>
<th></th>
<th><strong>GPU</strong></th>
<th><strong>Vector</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Programing Model</strong></td>
<td>SPMD (CUDA, OpenCL) Should reason threads</td>
<td>C code + Auto Vectorization SPMD (OpenCL) (RISC-V Vector Assembly)</td>
</tr>
<tr>
<td><strong>Execution Model</strong></td>
<td>SIMT (Single instruction Multiple threads)</td>
<td>SIMD</td>
</tr>
<tr>
<td><strong>Memory Hierarchy</strong></td>
<td>Scratchpad + Caches</td>
<td>Transparent Caches</td>
</tr>
<tr>
<td><strong>Memory Operations</strong></td>
<td>Scatter/Gather + Hardware Coalescing Unit</td>
<td>Unit-stride, Strided, Scatter/gather</td>
</tr>
<tr>
<td><strong>Conditional Execution</strong></td>
<td>Thread Mask + Divergence Stack</td>
<td>Vector mask</td>
</tr>
<tr>
<td><strong>Hardware Complexity</strong></td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
Summary

• Memory hierarchy
  • To cope with the speed gap between CPU and main memory
  • Various cache optimizations

• Virtual memory (paging)
  • No programs run in bare-metal
  • Each program runs as if it has its own contiguous memory space
  • OS + HW (TLB, virtually-indexed physically-tagged)

• Very Long Instruction Word (VLIW)
  • Simple HW + very smart compilers
  • Loop unrolling, SW pipelining, trace scheduling
  • Predictability: variable memory latency, branches, exceptions

• Data-level Parallelism
  • Vector Processor vs. SIMD vs. GPU
  • HW & SW techniques for vector machines
  • RISC-V Vector Programming