Exploiting Instruction-Level Parallelism with Software Approach #2

E. J. Kim
Unrolled Loop (No Scheduling)

Clock cycle issued

Loop: L.D F0, 0(R1) 1 2
      ADD.D F4, F0, F2 3 4 5
      S.D F4, 0(R1) 6
      L.D F6, -8(R1) 7 8
      ADD.D F8, F6, F2 9 10 11
      S.D F8, -8(R1) 12
      L.D F10, -16(R1) 13 14
      ADD.D F12, F10, F2 15 16 17
      S.D F12, -16(R1) 18
      L.D F14, -24(R1) 19 20
      ADD.D F16, F14, F2 21 22 23
      S.D F16, -24(R1) 24
      DADDIU R1, R1, # -32 25 26
      BNE R1, R2, LOOP 27 28
## Loop Unrolling (Scheduling)

Clock cycle issued

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>L.D</td>
<td>F6, -8(R1)</td>
<td>2</td>
</tr>
<tr>
<td>L.D</td>
<td>F10, -16(R1)</td>
<td>3</td>
</tr>
<tr>
<td>L.D</td>
<td>F14, -24(R1)</td>
<td>4</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
<td>5</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8, F6, F2</td>
<td>6</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12, F10, F2</td>
<td>7</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16, F14, F2</td>
<td>8</td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
<td>9</td>
</tr>
<tr>
<td>S.D</td>
<td>F8, -8(R1)</td>
<td>10</td>
</tr>
<tr>
<td>DADDIU</td>
<td>R1, R1, # -32</td>
<td>11</td>
</tr>
<tr>
<td>S.D</td>
<td>F12, 16(R1)</td>
<td>12</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, LOOP</td>
<td>13</td>
</tr>
<tr>
<td>S.D</td>
<td>F16, 8(R1)</td>
<td>14</td>
</tr>
</tbody>
</table>
Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW)
Recall: Software Pipelining Example

<table>
<thead>
<tr>
<th>Before: Unrolled 3 times</th>
<th>After: Software Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) L.D F0,0(R1)</td>
<td>1) S.D 0(R1),F4 ; Stores M[i]</td>
</tr>
<tr>
<td>2) ADD.D F4,F0,F2</td>
<td>2) ADD.D F4,F0,F2 ; Adds to M[i-1]</td>
</tr>
<tr>
<td>3) S.D 0(R1),F4</td>
<td>3) L.D F0,-16(R1); Loads M[i-2]</td>
</tr>
<tr>
<td>4) L.D F6,-8(R1)</td>
<td>4) DSUBUI R1,R1,#8</td>
</tr>
<tr>
<td>5) ADD.D F8,F6,F2</td>
<td>5) BNEZ R1,LOOP</td>
</tr>
<tr>
<td>6) S.D -8(R1),F8</td>
<td></td>
</tr>
<tr>
<td>7) L.D F10,-16(R1)</td>
<td></td>
</tr>
<tr>
<td>8) ADD.D F12,F10,F2</td>
<td></td>
</tr>
<tr>
<td>9) S.D -16(R1),F12</td>
<td></td>
</tr>
<tr>
<td>10) DSUBUI R1,R1,#24</td>
<td></td>
</tr>
<tr>
<td>11) BNEZ R1,LOOP</td>
<td></td>
</tr>
</tbody>
</table>

- Symbolic Loop Unrolling
  - Maximize result-use distance
  - Less code space than unrolling
  - Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling
# Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -8(R1),F8</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -24(R1),F16</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -40(R1),F24</td>
<td></td>
<td></td>
<td>DSUBUI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>S.D -0(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>9</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)
Software Pipelining with Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/ branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,-48(R1)</td>
<td>ST 0(R1),F4</td>
<td>ADD.D F4,F0,F2</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F6,-56(R1)</td>
<td>ST -8(R1),F8</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>DSUBUI R1,R1,#24 2</td>
<td></td>
</tr>
<tr>
<td>L.D F10,-40(R1)</td>
<td>ST 8(R1),F12</td>
<td>ADD.D F12,F10,F2</td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Software pipelined across 9 iterations of original loop**
  - In each iteration of above loop, we:
    » Store to m,m-8,m-16 (iterations I-3,I-2,I-1)
    » Compute for m-24,m-32,m-40 (iterations I,I+1,I+2)
    » Load from m-48,m-56,m-64 (iterations I+3,I+4,I+5)

- **9 results in 9 cycles, or 1 clock per iteration**
- **Average: 3.3 ops per clock, 66% efficiency**

Note: Need fewer registers for software pipelining (only using 7 registers here, was using 15)

CPSC614
Lec 7.7
Trace Scheduling

- Parallelism across IF branches vs. LOOP branches?
- Two steps:
  - Trace Selection
    » Find likely sequence of basic blocks (trace) of (statically predicted or profile predicted) long sequence of straight-line code
  - Trace Compaction
    » Squeeze trace into few VLIW instructions
    » Need bookkeeping code in case prediction is wrong

- This is a form of compiler-generated speculation
  - Compiler must generate “fixup” code to handle cases in which trace is not the taken branch
  - Needs extra registers: undoes bad guess by discarding
Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

• **HW advantages:**
  - HW better at memory disambiguation since knows actual addresses
  - HW better at branch prediction since lower overhead
  - HW maintains precise exception model
  - HW does not execute bookkeeping instructions
  - Same software works across multiple implementations
  - Smaller code size (not as many nops filling blank instructions)

• **SW advantages:**
  - Window of instructions that is examined for parallelism much higher
  - Much less hardware involved in VLIW
  - More involved types of speculation can be done more easily
  - Speculation can be based on large-scale program behavior, not just local information
Superscalar v. VLIW

- Smaller code size
- Binary compatibility across generations of hardware
- Simplified Hardware for decoding, issuing instructions
- No Interlock Hardware (compiler checks?)
- More registers, but simplified Hardware for Register Ports (multiple independent register files?)
Problems with First Generation VLIW

- **Increase in code size**
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

- **Operated in lock-step; no hazard detection HW**
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict

- **Binary code compatibility**
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **IA-64**: instruction set architecture; EPIC is type
  - EPIC = 2nd generation VLIW?

- **Itanium™**: is name of first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 μ process

- 128 64-bit integer registers + 128 82-bit floating point registers
  - Not separate register files per functional unit as in old VLIW

- Hardware checks dependencies

- Predicated execution (select 1 out of 64 1-bit flags)
  => 40% fewer mispredictions?
IA-64 Registers

- The integer registers are configured to help accelerate procedure calls using a register stack
  - mechanism similar to that developed in the Berkeley RISC-I processor and used in the SPARC architecture.
  - Registers 0-31 are always accessible and addressed as 0-31
  - Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96)
  - The new register stack frame is created for a called procedure by renaming the registers in hardware;
  - a special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure

- 8 64-bit Branch registers used to hold branch destination addresses for indirect branches
- 64 1-bit predict registers
IA-64 Registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation is designed to ease the task of allocating of registers in software pipelined loops.
- When combined with predication, possible to avoid the need for unrolling and for separate prologue and epilogue code for a software pipelined loop:
  - makes the SW-pipelining usable for loops with smaller numbers of iterations, where the overheads would traditionally negate many of the advantages.
Conditional Instructions

• Predicated instructions
• Extension of instruction set
• Conditional instruction: an instruction that refers a condition, which is evaluated as part of the instruction execution
  – Condition is true: executed normally
  – False: no-op
  – ex) conditional move
Example

if (A == 0) { S = T; }

BNEZ R1, L
ADDU R2, R3, R0
L:

CMOVZ R2, R3, R1

conditional move only if the third operand is equal to zero

R1=A, R2=S, R3=T
• Conditional moves are used to change a control dependence into a data dependence.

• Handling multiple branches per cycle is complex. => Conditional moves provide a way of reducing branch pressure.

• A conditional move can often eliminate a branch that is hard to predict, increasing the potential gain.
Review of Caches
Who Cares About the Memory Hierarchy?

Processor-Memory Performance Gap: (grows 50% / year)

CPU 60%/yr.

DRAM 7%/yr.

“Moore’s Law”

“Less’ Law?”

• 1980: no cache in μproc; 1995 2-level cache on chip
  (1989 first Intel μproc with a cache on chip)
What is a cache?

- Small, fast storage used to improve average access time to slow memory.
- Exploits spacial and temporal locality
- In computer architecture, almost everything is a cache!
  - Registers “a cache” on variables - software managed
  - First-level cache a cache on second-level cache
  - Second-level cache a cache on memory
  - Memory a cache on disk (virtual memory)
  - TLB a cache on page table
  - Branch-prediction a cache on prediction information?

```
        Proc/Regs
       /     \     
   L1-Cache /       \ L2-Cache
  /       \       
Memory

Disk, Tape, etc.
```

Bigger Faster
Review: Cache performance

- Miss-oriented Approach to Memory Access:

\[
CPUtime = IC \times \left( CPI_{\text{Execution}} + \frac{\text{MemAccess}}{\text{Inst}} \times \text{MissRate} \times \text{MissPenalty} \right) \times \text{CycleTime}
\]

\[
CPUtime = IC \times \left( CPI_{\text{Execution}} + \frac{\text{MemMisses}}{\text{Inst}} \times \text{MissPenalty} \right) \times \text{CycleTime}
\]

- CPI_{\text{Execution}} includes ALU and Memory instructions

- Separating out Memory component entirely

  - AMAT = Average Memory Access Time
  - CPI_{\text{ALUOps}} does not include memory instructions

\[
CPUtime = IC \times \left( \frac{\text{AluOps}}{\text{Inst}} \times CPI_{\text{AluOps}} + \frac{\text{MemAccess}}{\text{Inst}} \times AMAT \right) \times \text{CycleTime}
\]

\[
AMAT = \text{HitTime} + \text{MissRate} \times \text{MissPenalty}
\]

\[
= \left( \text{HitTime}_{\text{Inst}} + \text{MissRate}_{\text{Inst}} \times \text{MissPenalty}_{\text{Inst}} \right) +
\left( \text{HitTime}_{\text{Data}} + \text{MissRate}_{\text{Data}} \times \text{MissPenalty}_{\text{Data}} \right)
\]
Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control

- Suppose that 10% of memory operations get 50 cycle miss penalty

- Suppose that 1% of instructions get same miss penalty

- CPI = ideal CPI + average stalls per instruction
  
  \[
  1.1 \text{(cycles/ins)} + \\
  [ 0.30 \text{(DataMops/ins)} \\
  \times 0.10 \text{(miss/DataMop)} \times 50 \text{(cycle/miss)}] + \\
  [ 1 \text{(InstMop/ins)} \\
  \times 0.01 \text{(miss/InstMop)} \times 50 \text{(cycle/miss)}] \\
  = (1.1 + 1.5 + .5) \text{ cycle/ins} = 3.1
  \]

- 58% of the time the proc is stalled waiting for memory!

- AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54
Unified vs Split Caches

- **Unified vs Separate I&D**

  ![Diagram](image)

- **Example:**
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%

- **Which is better (ignore L2 cache)?**
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that data hit has 1 stall for unified cache (only one port)

\[
\text{AMAT}_{\text{Harvard}} = 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) = 2.05 \\
\text{AMAT}_{\text{Unified}} = 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) = 2.24
\]
How to Improve Cache Performance?

\[ AMAT = HitTime + MissRate \times MissPenalty \]

1. Reduce the miss rate.
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.
Where to misses come from?

- **Classifying Misses: 3 Cs**
  
  - *Compulsory*—The first access to a block is not in the cache, so the block must be brought into the cache. Also called *cold start misses* or *first reference misses.*
    *(Misses in even an Infinite Cache)*
  
  - *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, *capacity misses* will occur due to blocks being discarded and later retrieved.
    *(Misses in Fully Associative Size X Cache)*
  
  - *Conflict*—If block-placement strategy is set associative or direct mapped, *conflict misses* (in addition to compulsory & capacity misses) will occur because a block can be discarded and later retrieved if too many blocks map to its set. Also called *collision misses* or *interference misses.*
    *(Misses in N-way Associative, Size X Cache)*
3Cs Absolute Miss Rate (SPEC92)

![Graph showing the relationship between cache size (KB) and miss rate for 1-way, 2-way, 4-way, and 8-way caches. The graph indicates that as cache size increases, the miss rate decreases for all cache ways. The graph also highlights the 'Conflict' and 'Capacity' regions, with the 'Compulsory' region showing the lowest miss rates.](image-url)
Larger Block Size (fixed size&assoc)

Block Size (bytes)

- 16
- 32
- 64
- 128
- 256

Miss Rate
- 0%
- 5%
- 10%
- 15%
- 20%
- 25%

- 1K
- 4K
- 16K
- 64K
- 256K

Increased Conflict Misses

Reduced compulsory misses
Traditional Four Questions for Memory Hierarchy Designers

• Q1: Where can a block be placed in the upper level? *(Block placement)*
  - Fully Associative, Set Associative, Direct Mapped
• Q2: How is a block found if it is in the upper level? *(Block identification)*
  - Tag/Block
• Q3: Which block should be replaced on a miss? *(Block replacement)*
  - Random, LRU
• Q4: What happens on a write? *(Write strategy)*
  - Write Back or Write Through (with Write Buffer)
Block Placement

- **Direct Mapped**
  - (block address) mod (number of blocks in cache)
- **Fully-associative**
- **Set associative**
  - (block address) mod (number of sets in cache)
  - n-way set associative
Simplest Cache: Direct Mapped

- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index

- Which one should we place in the cache?
- How can we tell which one is in the cache?
1 KB Direct Mapped Cache, 32B blocks

- For a $2^{N}$ byte cache:
  - The uppermost $(32 - N)$ bits are always the Cache Tag
  - The lowest $M$ bits are the Byte Select (Block Size = $2^{M}$)

<table>
<thead>
<tr>
<th>31</th>
<th>Cache Tag</th>
<th>Example: 0x50</th>
<th>9</th>
<th>Cache Index</th>
<th>4</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Valid Bit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stored as part of the cache “state”

<table>
<thead>
<tr>
<th>Cache Tag</th>
<th>Cache Index</th>
<th>Byte Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cache Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 31</td>
</tr>
<tr>
<td>**</td>
</tr>
<tr>
<td>Byte 1</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Ex: 0x01
Ex: 0x00
Two-way Set Associative Cache

- **N-way set associative**: N entries for each Cache Index
  - N direct mapped caches operate in parallel (N typically 2 to 4)

- **Example: Two-way set associative cache**
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result

![Diagram of Two-way Set Associative Cache]
Disadvantage of Set Associative Cache

- **N-way Set Associative Cache v. Direct Mapped Cache:**
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss

- **In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:**
  - Possible to assume a hit and continue. Recover later if miss.

---

Diagram:

- Valid
- Cache Tag
- Cache Data
- Cache Block 0
- Cache Tag
- Cache Data
- Cache Block 0
- Cache Tag
- Cache Data

- Adr Tag
- Compare
- Mux
- Sel1
- Sel0
- OR
- Hit
- Cache Block

Diagram shows the flow of data and comparison between different cache elements.
Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

```
Full Mapped               Direct Mapped        2-Way Assoc
(12 mod 8) = 4           (12 mod 4) = 0

01234567 01234567 01234567

Cache

01234567890123456789012345678901

Memory
```
Q2: How is a block found if it is in the upper level?

- **Tag on each block**
  - No need to check index or block offset
- **Increasing associativity shrinks index, expands tag**

<table>
<thead>
<tr>
<th>Block Address</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
</tr>
</tbody>
</table>
Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

<table>
<thead>
<tr>
<th>Assoc:</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>LRU</td>
<td>Ran</td>
<td>LRU</td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>
Q4: What happens on a write?

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.

- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?

- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location

- WT always combined with write buffers so that don’t wait for lower level memory
Write Buffer for Write Through

![Diagram of memory system with Processor, Cache, Write Buffer, and DRAM]

- **A Write Buffer is needed between the Cache and Memory**
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory

- **Write buffer is just a FIFO:**
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) $<< 1 / \text{DRAM write cycle}$

- **Memory system design:**
  - Store frequency (w.r.t. time) $\rightarrow 1 / \text{DRAM write cycle}$
  - Write buffer saturation
A Modern Memory Hierarchy

- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.

<table>
<thead>
<tr>
<th>Component</th>
<th>Speed (ns)</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Datapath</td>
<td>1s</td>
<td>100s</td>
</tr>
<tr>
<td>Control</td>
<td>10s</td>
<td>Ks</td>
</tr>
<tr>
<td>On-Chip Cache</td>
<td>100s</td>
<td>Ms</td>
</tr>
<tr>
<td>Second Level Cache (SRAM)</td>
<td>100s</td>
<td>Gs</td>
</tr>
<tr>
<td>Main Memory (DRAM)</td>
<td>10,000,000s</td>
<td>Ts</td>
</tr>
<tr>
<td>Secondary Storage (Disk)</td>
<td>10,000,000,000s</td>
<td></td>
</tr>
<tr>
<td>Tertiary Storage (disk/Tape)</td>
<td>10,000,000,000s</td>
<td></td>
</tr>
</tbody>
</table>
**DECstation 3100 cache**

- **Valid Tag Data**
- **Hit Data**

- **16 bits**
- **32 bits**

- **16K entries**

- **Byte offset**

- **block size = one word**
64KB Cache Using 4-Word Blocks

Address (showing in packet)

Hit

Data

16 Byte offset

2 Byte offset

Index

16

12

16 bits

128 bits

V

Tag

Data

Tag

Mux

32

32

32

32

4K entries
Block Size vs. Miss Rate

• Increasing block size: decreasing miss rate.
  - The instruction references have better spatial locality.

• Serious problem with just increasing the block size: miss penalty increases.
  - Miss penalty: the time required to fetch the block from the next lower level of the hierarchy and load it into the cache.
  - Miss penalty = (latency to the first word) + (transfer time for the rest of the block)