Assignment and Exam Reminders

- **Lab 4: Due March 6**
  - Control flow and branch prediction

- **Lab 5: Due March 22**
  - Data cache

- **HW 4: March 18**
- **Exam: March 20**

- Finish the labs early
- You have almost a month for Lab 5
Announcements

- Please turn in your feedback form: Very Important

- No office hours (for me) today
IA-64: A “Complicated” VLIW ISA

Recommended reading:
EPIC – Intel IA-64 Architecture

- Gets rid of lock-step execution of instructions within a VLIW instruction
- Idea: More ISA support for static scheduling and parallelization
  - Specify dependencies within and between VLIW instructions (explicitly parallel)

+ No lock-step execution
+ Static reordering of stores and loads + dynamic checking
-- Hardware needs to perform dependency checking (albeit aided by software)
-- Other disadvantages of VLIW still exist

IA-64 Instructions

- **IA-64 “Bundle” (~EPIC Instruction)**
  - Total of 128 bits
  - Contains three IA-64 instructions
  - Template bits in each bundle specify dependencies within a bundle

- **IA-64 Instruction**
  - Fixed-length 41 bits long
  - Contains three 7-bit register specifiers
  - Contains a 6-bit field for specifying one of the 64 one-bit predicate registers
Groups of instructions can be executed safely in parallel
- Marked by “stop bits”

Bundles are for packaging
- Groups can span multiple bundles
  - Alleviates recompilation need somewhat
### Template Bits

- **Specify two things**
  - Stop information: Boundary of independent instructions
  - Functional unit information: Where should each instruction be routed

<table>
<thead>
<tr>
<th>Template</th>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>01</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>02</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>03</td>
<td>M-unit</td>
<td>I-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>04</td>
<td>M-unit</td>
<td>L-unit</td>
<td>X-unit(^a)</td>
</tr>
<tr>
<td>05</td>
<td>M-unit</td>
<td>L-unit</td>
<td>X-unit(^a)</td>
</tr>
<tr>
<td>06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>09</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0A</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0B</td>
<td>M-unit</td>
<td>M-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0C</td>
<td>M-unit</td>
<td>F-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0D</td>
<td>M-unit</td>
<td>F-unit</td>
<td>I-unit</td>
</tr>
<tr>
<td>0E</td>
<td>M-unit</td>
<td>M-unit</td>
<td>F-unit</td>
</tr>
<tr>
<td>0F</td>
<td>M-unit</td>
<td>M-unit</td>
<td>F-unit</td>
</tr>
<tr>
<td>10</td>
<td>M-unit</td>
<td>I-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>11</td>
<td>M-unit</td>
<td>I-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>12</td>
<td>M-unit</td>
<td>B-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>13</td>
<td>M-unit</td>
<td>B-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>B-unit</td>
<td>B-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>17</td>
<td>B-unit</td>
<td>B-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>18</td>
<td>M-unit</td>
<td>M-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>19</td>
<td>M-unit</td>
<td>M-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>1A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1C</td>
<td>M-unit</td>
<td>F-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>1D</td>
<td>M-unit</td>
<td>F-unit</td>
<td>B-unit</td>
</tr>
<tr>
<td>1E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^a\) X-unit is an additional functional unit that can be used in certain templates.
Three Things That Hinder Static Scheduling

- Dynamic events (static unknowns)
- Branch direction
- Load hit miss status
- Memory address

Let’s see how IA-64 ISA has support to aid scheduling in the presence of statically-unknown load-store addresses
Idea: Support unsafe code motion

- **ld.s** (speculative load) fetches *speculatively* from memory. i.e. any exception due to **ld.s** is suppressed.
- If **ld.s r1** did not cause an exception then **chk.s r1** is a NOP, else a branch is taken (to execute some compensation code).
Non-Faulting Loads and Exception Propagation in IA-64

- Idea: Support unsafe code motion
  - Load and its use

- Load data can be speculatively consumed (use) prior to check
- “speculation” status is propagated with speculated data
- Any instruction that uses a speculative result also becomes speculative itself (i.e. suppressed exceptions)
- *chk.s* checks the entire dataflow sequence for exceptions
Aggressive ST-LD Reordering in IA-64

- **Idea:** Reorder LD/STs in the presence of unknown address

- **ld.a** (advanced load) starts the monitoring of any store to the same address as the advanced load

- If no aliasing has occurred since *ld.a, ld.c* is a NOP

- If aliasing has occurred, *ld.c* re-loads from memory
Aggressive ST-LD Reordering in IA-64

- Idea: Reorder LD/STs in the presence of unknown address
  - Load and its use

inst 1
inst 2
...
ld r1=[x]
use=r1

potential aliasing

ld.a r1=[x]
inst 1
inst 2
use=r1
...
...
st [?]....
chk.a X
...
ld r1=[a]
use=r1
What We Covered So Far in 447

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

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

- Static Instruction Scheduling
Agenda for the Rest of 447

- The memory hierarchy
- Caches, caches, more caches (high locality, high bandwidth)
- Virtualizing the memory hierarchy
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory
- Multiprocessors
- Coherence and consistency
- Interconnection networks
- Multi-core issues
Readings for Today and Next Lecture

- Memory Hierarchy and Caches
- Cache chapters from P&H: 5.1-5.3
- Memory/cache chapters from Hamacher+: 8.1-8.7
- An early cache paper by Maurice Wilkes
Memory (Programmer’s View)
Abstraction: Virtual vs. Physical Memory

- **Programmer** sees virtual memory
  - Can assume the memory is “infinite”

- **Reality:** Physical memory size is much smaller than what the programmer assumes

- **The system** (system software + hardware, cooperatively) maps virtual memory addresses are to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

+ Programmer does not need to know the physical size of memory nor manage it → A small physical memory can appear as a huge one to the programmer → Life is easier for the programmer

-- More complex system software and architecture

A classic example of the programmer/(micro)architect tradeoff
You need a larger level of storage to manage a small amount of physical memory automatically.

- Physical memory has a backing store: disk

We will first start with the physical memory system.

For now, ignore the virtual → physical indirection.

- As you have been doing in labs

We will get back to it when the needs of virtual memory start complicating the design of physical memory...
Idealism

Instruction Supply

Pipeline (Instruction execution)

Data Supply

- Zero-cycle latency
- Infinite capacity
- Zero cost
- Perfect control flow

- No pipeline stalls

- Zero-cycle interconnect (operand communication)

- Enough functional units
- Zero latency compute

- Zero-cycle latency
- Infinite capacity
- Infinite bandwidth
- Zero cost
The Memory Hierarchy
Memory in a Modern System
Ideal Memory

- Zero access time (latency)
- Infinite capacity
- Zero cost
- Infinite bandwidth (to support multiple accesses in parallel)
The Problem

- Ideal memory’s requirements oppose each other

- Bigger is slower
  - Bigger $\rightarrow$ Takes longer to determine the location

- Faster is more expensive
  - Memory technology: SRAM vs. DRAM vs. Disk vs. Tape

- Higher bandwidth is more expensive
  - Need more banks, more ports, higher frequency, or faster technology
Memory Technology: DRAM

- Dynamic random access memory
- Capacitor charge state indicates stored value
  - Whether the capacitor is charged or discharged indicates storage of 1 or 0
  - 1 capacitor
  - 1 access transistor

- Capacitor leaks through the RC path
  - DRAM cell loses charge over time
  - DRAM cell needs to be refreshed
Memory Technology: SRAM

- Static random access memory
- Two cross coupled inverters store a single bit
  - Feedback path enables the stored value to persist in the “cell”
  - 4 transistors for storage
  - 2 transistors for access
Memory Bank Organization and Operation

- Read access sequence:
  1. Decode row address & drive word-lines
  2. Selected bits drive bit-lines
     - Entire row read
  3. Amplify row data
  4. Decode column address & select subset of row
     - Send to output
  5. Precharge bit-lines
     - For next access
SRAM (Static Random Access Memory)

Read Sequence
1. address decode
2. drive row select
3. selected bit-cells drive bitlines
   (entire row is read together)
4. differential sensing and column select
   (data is ready)
5. precharge all bitlines
   (for next read or write)

Access latency dominated by steps 2 and 3
Cycling time dominated by steps 2, 3 and 5
- step 2 proportional to $2^m$
- step 3 and 5 proportional to $2^n$
DRAM (Dynamic Random Access Memory)

- Bits stored as charges on node capacitance (non-restorative)
  - bit cell loses charge when read
  - bit cell loses charge over time

Read Sequence

1~3 same as SRAM

4. a “flip-flopping” sense amp amplifies and regenerates the bitline, data bit is mux’ ed out

5. precharge all bitlines

Destructive reads

Charge loss over time

Refresh: A DRAM controller must periodically read each row within the allowed refresh time (10s of ms) such that charge is restored
DRAM vs. SRAM

- **DRAM**
  - Slower access (capacitor)
  - Higher density (1T 1C cell)
  - Lower cost
  - Requires refresh (power, performance, circuitry)
  - Manufacturing requires putting capacitor and logic together

- **SRAM**
  - Faster access (no capacitor)
  - Lower density (6T cell)
  - Higher cost
  - No need for refresh
  - Manufacturing compatible with logic process (no capacitor)
The Problem

- **Bigger is slower**
  - SRAM, 512 Bytes, sub-nanosec
  - SRAM, KByte~MByte, ~nanosec
  - DRAM, Gigabyte, ~50 nanosec
  - Hard Disk, Terabyte, ~10 millisec

- **Faster is more expensive (dollars and chip area)**
  - SRAM, < 10$ per Megabyte
  - DRAM, < 1$ per Megabyte
  - Hard Disk < 1$ per Gigabyte
  - These sample values scale with time

- **Other technologies have their place as well**
  - Flash memory, PC-RAM, MRAM, RRAM (not mature yet)
Why Memory Hierarchy?

- We want both fast and large

- But we cannot achieve both with a single level of memory

- Idea: Have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the fast(er) level(s)
The Memory Hierarchy

Move what you use here

With good locality of reference, memory appears as fast as and as large as.

Backup everything here

Fast small

Big but slow

Faster per byte

Cheaper per byte
Memory Hierarchy

- Fundamental tradeoff
  - Fast memory: small
  - Large memory: slow
- Idea: Memory hierarchy

- Latency, cost, size, bandwidth
Locality

- One’s recent past is a very good predictor of his/her near future.

- **Temporal Locality**: If you just did something, it is very likely that you will do the same thing again soon
  - since you are here today, there is a good chance you will be here again and again regularly

- **Spatial Locality**: If you did something, it is very likely you will do something similar/related (in space)
  - every time I find you in this room, you are probably sitting close to the same people
Memory Locality

- A “typical” program has a lot of locality in memory references
  - typical programs are composed of “loops”

- Temporal: A program tends to reference the same memory location many times and all within a small window of time

- Spatial: A program tends to reference a cluster of memory locations at a time
  - most notable examples:
    - 1. instruction memory references
    - 2. array/data structure references
Caching Basics: Exploit Temporal Locality

- **Idea:** Store recently accessed data in automatically managed fast memory (called cache)
- **Anticipation:** the data will be accessed again soon

- **Temporal locality** principle
  - Recently accessed data will be again accessed in the near future
  - This is what Maurice Wilkes had in mind:
    - “The use is discussed of a fast core memory of, say 32000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.”
Caching Basics: Exploit Spatial Locality

- **Idea:** Store addresses adjacent to the recently accessed one in automatically managed fast memory
  - Logically divide memory into equal size blocks
  - Fetch to cache the accessed block in its entirety
- **Anticipation:** nearby data will be accessed soon

- **Spatial locality** principle
  - Nearby data in memory will be accessed in the near future
    - E.g., sequential instruction access, array traversal
  - This is what IBM 360/85 implemented
    - 16 Kbyte cache with 64 byte blocks
The Bookshelf Analogy

- Book in your hand
- Desk
- Bookshelf
- Boxes at home
- Boxes in storage

- Recently-used books tend to stay on desk
  - Comp Arch books, books for classes you are currently taking
  - Until the desk gets full

- Adjacent books in the shelf needed around the same time
  - If I have organized/categorized my books well in the shelf
Caching in a Pipelined Design

- The cache needs to be tightly integrated into the pipeline
  - Ideally, access in 1-cycle so that dependent operations do not stall
- High frequency pipeline \(\Rightarrow\) Cannot make the cache large
  - But, we want a large cache AND a pipelined design
- Idea: Cache hierarchy
A Note on Manual vs. Automatic Management

- **Manual:** Programmer manages data movement across levels
  -- too painful for programmers on substantial programs
  - “core” vs “drum” memory in the 50’s
  - still done in some embedded processors (on-chip scratch pad SRAM in lieu of a cache)

- **Automatic:** Hardware manages data movement across levels, transparently to the programmer
  ++ programmer’s life is easier
  - the average programmer doesn’t need to know about it
    - You don’t need to know how big the cache is and how it works to write a “correct” program! (What if you want a “fast” program?)
Automatic Management in Memory Hierarchy


**Slave Memories and Dynamic Storage Allocation**

**M. V. WILKES**

**Summary**

The use is discussed of a fast core memory of, say, 32,000 words as a slave to a slower core memory of, say, one million words in such a way that in practical cases the effective access time is nearer that of the fast memory than that of the slow memory.

- “By a slave memory I mean one which automatically accumulates to itself words that come from a slower main memory, and keeps them available for subsequent use without it being necessary for the penalty of main memory access to be incurred again.”
A Modern Memory Hierarchy

Memory Abstraction

- **Register File**: 32 words, sub-nsec
- **L1 cache**: ~32 KB, ~nsec
- **L2 cache**: 512 KB ~ 1MB, many nsec
- **L3 cache**: ..... 
- **Main memory (DRAM)**: GB, ~100 nsec
- **Swap Disk**: 100 GB, ~10 msec

- **Manual/Compiler register spilling**
- **Automatic HW cache management**
- **Automatic demand paging**
Hierarchical Latency Analysis

- For a given memory hierarchy level \( i \) it has a technology-intrinsic access time of \( t_i \). The perceived access time \( T_i \) is longer than \( t_i \).
- Except for the outer-most hierarchy, when looking for a given address there is
  - a chance (hit-rate \( h_i \)) you “hit” and access time is \( t_i \)
  - a chance (miss-rate \( m_i \)) you “miss” and access time \( t_i + T_{i+1} \)
  - \( h_i + m_i = 1 \)
- Thus
  \[
  T_i = h_i \cdot t_i + m_i \cdot (t_i + T_{i+1})
  \]
  \[
  T_i = t_i + m_i \cdot T_{i+1}
  \]

\( h_i \) and \( m_i \) are defined to be the hit-rate and miss-rate of just the references that missed at \( L_{i-1} \).
Hierarchy Design Considerations

- Recursive latency equation
  \[ T_i = t_i + m_i \cdot T_{i+1} \]

- The goal: achieve desired \( T_1 \) within allowed cost

- \( T_i \approx t_i \) is desirable

- Keep \( m_i \) low
  - increasing capacity \( C_i \) lowers \( m_i \), but beware of increasing \( t_i \)
  - lower \( m_i \) by smarter management (replacement::anticipate what you don’t need, prefetching::anticipate what you will need)

- Keep \( T_{i+1} \) low
  - faster lower hierarchies, but beware of increasing cost
  - introduce intermediate hierarchies as a compromise
90nm P4, 3.6 GHz

- L1 D-cache
  - \( C_1 = 16K \)
  - \( t_1 = 4 \text{ cyc int} / 9 \text{ cycle fp} \)
- L2 D-cache
  - \( C_2 = 1024 \text{ KB} \)
  - \( t_2 = 18 \text{ cyc int} / 18 \text{ cyc fp} \)
- Main memory
  - \( t_3 = \sim 50\text{ns} \text{ or } 180 \text{ cyc} \)

Notice
- best case latency is not 1
- worst case access latencies are into 500+ cycles
Cache Basics and Operation
Cache

- Generically, any structure that “memoizes” frequently used results to avoid repeating the long-latency operations required to reproduce the results from scratch, e.g. a web cache

- Most commonly in the on-die context: an automatically-managed memory hierarchy based on SRAM
  - memoize in SRAM the most frequently accessed DRAM memory locations to avoid repeatedly paying for the DRAM access latency
Caching Basics

- Block (line): Unit of storage in the cache
  - Memory is logically divided into cache blocks that map to locations in the cache

- When data referenced
  - HIT: If in cache, use cached data instead of accessing memory
  - MISS: If not in cache, bring block into cache
    - Maybe have to kick something else out to do it

- Some important cache design decisions
  - Placement: where and how to place/find a block in cache?
  - Replacement: what data to remove to make room in cache?
  - Granularity of management: large, small, uniform blocks?
  - Write policy: what do we do about writes?
  - Instructions/data: Do we treat them separately?
Cache Abstraction and Metrics

- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT) = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Aside: Can reducing AMAT reduce performance?