CPU Pipelining Issues

What have you been beating your head against?

This pipe stuff makes my head hurt!
Pipelining

Improve performance by increasing instruction throughput

Program execution order (in instructions)

lw $1, 100(0)$
lw $2, 200(0)$
lw $3, 300(0)$

Program execution order (in instructions)

lw $1, 100(0)$
lw $2, 200(0)$
lw $3, 300(0)$

Ideal speedup is number of stages in the pipeline. Do we achieve this?
Pipelining

What makes it easy

- all instructions are the same length
- just a few instruction formats
- memory operands appear only in loads and stores

What makes it hard?

- structural hazards: suppose we had only one memory
- control hazards: need to worry about branch instructions
- data hazards: an instruction depends on a previous instruction

Individual Instructions still take the same number of cycles
But we’ve improved the through-put by increasing the number of simultaneously executing instructions
# Structural Hazards

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Reg Read</th>
<th>ALU</th>
<th>Data Access</th>
<th>Reg Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst Fetch</td>
<td>Reg Read</td>
<td>ALU</td>
<td>Data Access</td>
<td>Reg Write</td>
</tr>
<tr>
<td>Inst Fetch</td>
<td>Reg Read</td>
<td>ALU</td>
<td>Data Access</td>
<td>Reg Write</td>
</tr>
<tr>
<td>Inst Fetch</td>
<td>Reg Read</td>
<td>ALU</td>
<td>Data Access</td>
<td>Reg Write</td>
</tr>
</tbody>
</table>
Data Hazards

Problem with starting next instruction before first is finished

dependencies that “go backward in time” are data hazards

<table>
<thead>
<tr>
<th>Program execution order (in instructions)</th>
<th>Time (in clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub $2, $1, $3</td>
<td>CC 1: 10</td>
</tr>
<tr>
<td>and $12, $2, $5</td>
<td>CC 2: 10</td>
</tr>
<tr>
<td>or $13, $6, $2</td>
<td>CC 3: 10</td>
</tr>
<tr>
<td>add $14, $2, $2</td>
<td>CC 4: 10</td>
</tr>
<tr>
<td>sw $15, 100($2)</td>
<td>CC 5: 10–20</td>
</tr>
<tr>
<td></td>
<td>CC 6: –20</td>
</tr>
<tr>
<td></td>
<td>CC 7: –20</td>
</tr>
<tr>
<td></td>
<td>CC 8: –20</td>
</tr>
<tr>
<td></td>
<td>CC 9: –20</td>
</tr>
</tbody>
</table>

Value of register $2: 10 10 10 10 10–20 –20 –20 –20 –20
Software Solution

Have compiler guarantee no hazards

Where do we insert the “nops”?

\[
\begin{align*}
\text{sub} & \quad $2, $1, $3 \\
\text{and} & \quad $12, $2, $5 \\
\text{or} & \quad $13, $6, $2 \\
\text{add} & \quad $14, $2, $2 \\
\text{sw} & \quad $15, 100($2)
\end{align*}
\]

Problem: this really slows us down!
Forwarding

Use temporary results, don't wait for them to be written register file forwarding to handle read/write to same register ALU forwarding
Can't always forward

Load word can still cause a hazard:

an instruction tries to read a register following a load instruction that writes to
the same register.

Thus, we need a hazard detection unit to “stall” the instruction
We can stall the pipeline by keeping an instruction in the same stage.
When we decide to branch, other instructions are in the pipeline!

We are predicting “branch not taken”

need to add hardware for flushing instructions if we are wrong
Improving Performance

Try to avoid stalls! E.g., reorder these instructions:

\[
\text{lw } \$t0, \ 0(\$t1) \\
\text{lw } \$t2, \ 4(\$t1) \\
\text{sw } \$t2, \ 0(\$t1) \\
\text{sw } \$t0, \ 4(\$t1)
\]

Add a “branch delay slot”

the next instruction after a branch is always executed
rely on compiler to “fill” the slot with something useful

Superscalar: start more than one instruction in the same cycle
Dynamic Scheduling

The hardware performs the “scheduling”
  hardware tries to find instructions to execute
  out of order execution is possible
  speculative execution and dynamic branch prediction

All modern processors are very complicated
  Pentium 4: 20 stage pipeline, 6 simultaneous instructions
  PowerPC and Pentium: branch history table
  Compiler technology important
Pipeline Summary (I)

• Started with unpipelined implementation
  – direct execute, 1 cycle/instruction
  – it had a long cycle time: mem + regs + alu + mem + wb

• We ended up with a 5-stage pipelined implementation
  – increase throughput (3x???)
  – delayed branch decision (1 cycle)
    Choose to execute instruction after branch
  – delayed register writeback (3 cycles)
    Add bypass paths (6 x 2 = 12) to forward correct value
  – memory data available only in WB stage
    Introduce NOPs at IR<sup>ALU</sup>, to stall IF and RF stages
    until LD result was ready
Pipeline Summary (II)

Fallacy #1: Pipelining is easy
Smart people get it wrong all of the time!

Fallacy #2: Pipelining is independent of ISA
Many ISA decisions impact how easy/costly it is to implement pipelining (i.e. branch semantics, addressing modes).

Fallacy #3: Increasing Pipeline stages improves performance
Diminishing returns. Increasing complexity.
RISC = Simplicity???

“The P.T. Barnum World’s Tallest Dwarf Competition”

World’s Most Complex RISC?

- VLIWs, Super- Scalars
- Primitive Machines with direct implementations
- Complex instructions, addressing modes
- Addressing features, eg index registers
- Generalization of registers and operand coding
- Pipelines, Bypasses, Annulment, …, …

Comp 411
Memory Hierarchy

Why are you dressed like that? Halloween was weeks ago!

It makes me look faster, don’t you think?

• Memory Flavors
• Principle of Locality
• Program Traces
• Memory Hierarchies
• Associativity

(Study Chapter 5)
What Do We Want in a Memory?

<table>
<thead>
<tr>
<th></th>
<th>Capacity</th>
<th>Latency</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>1000’s of bits</td>
<td>10 ps</td>
<td>$$$$$</td>
</tr>
<tr>
<td>SRAM</td>
<td>1’s Mbytes</td>
<td>0.2 ns</td>
<td>$$$</td>
</tr>
<tr>
<td>DRAM</td>
<td>10’s Gbytes</td>
<td>10 ns</td>
<td>$</td>
</tr>
<tr>
<td>Hard disk*</td>
<td>10’s Tbytes</td>
<td>10 ms</td>
<td>$</td>
</tr>
<tr>
<td>Want?</td>
<td>100 Gbytes</td>
<td>0.2 ns</td>
<td>cheap</td>
</tr>
</tbody>
</table>

* non-volatile

* non-volatile
Tricks for Increasing Throughput

The first thing that should pop into your mind when asked to speed up a digital design...

**PIPELINING**

Synchronous DRAM (SDRAM) ($25 per Gbyte)

Double-clocked Synchronous DRAM (SDRAM)
Hard Disk Drives

Typical drive:
- Average latency = 4 ms (7200 rpm)
- Average seek time = 8.5 ms
- Transfer rate = 140 Mbytes/s (SATA)
- Capacity = 1.5 T byte
- Cost = $149 (10¢ G byte)
Quantity vs Quality...

Your memory system can be
• BIG and SLOW... or
• SMALL and FAST.

We've explored a range of
device-design trade-offs.

Is there an
ARCHITECTURAL solution
to this DELIMA?

$/GB

Access Time

- SRAM (5000$/GB, 0.2 ns)
- DRAM (25$/GB, 5 ns)
- DISK (0.10$/GB, 10 mS)
- DVD Burner (0.06$/G, 150ms)
Managing Memory via Programming

- In reality, systems are built with a mixture of all these various memory types

- How do we make the most effective use of each memory?
  - We could push all of these issues off to programmers
    - Keep most frequently used variables and stack in SRAM
    - Keep large data structures (arrays, lists, etc) in DRAM
    - Keep bigger data structures on disk (databases) on DISK

- It is harder than you think... data usage evolves over a program's execution
Best of Both Worlds

What we REALLY want: A BIG, FAST memory! (Keep everything within instant access)

We’d like to have a memory system that
• PERFORMS like 10 GBytes of SRAM; but
• COSTS like 1-4 Gbytes of slow memory.

SURPRISE: We can (nearly) get our wish!

KEY: Use a hierarchy of memory technologies:
Key IDEA

• Keep the most often-used data in a small, fast SRAM (often local to CPU chip)

• Refer to Main Memory only rarely, for remaining data.

The reason this strategy works: LOCALITY

Locality of Reference:

Reference to location X at time t implies that reference to location X+ΔX at time t+Δt becomes more probable as ΔX and Δt approach zero.
Cache

cache (kash)

n.

A hiding place used especially for storing provisions.
A place for concealment and safekeeping, as of valuables.
The store of goods or valuables concealed in a hiding place.

*Computer Science.* A fast storage buffer in the central processing unit of a computer. In this sense, also called cache memory.

v. tr. cached, caching, caches.

To hide or store in a cache.
Cache Analogy

You are writing a term paper at a table in the library.

As you work you realize you need a book.

You stop writing, fetch the reference, continue writing.

You don’t immediately return the book, maybe you’ll need it again.

Soon you have a few books at your table and no longer have to fetch more books.

The table is a CACHE for the rest of the library.
Typical Memory Reference Patterns

MEMORY TRACE – A temporal sequence of memory references (addresses) from a real program.

TEMPORAL LOCALITY – If an item is referenced, it will tend to be referenced again soon.

SPATIAL LOCALITY – If an item is referenced, nearby items will tend to be referenced soon.
Working set: a set $S$ which changes slowly w.r.t. access time.

$S$ is the set of locations accessed during $\Delta t$.

Working set size, $|S|$
Exploiting the Memory Hierarchy

Approach 1 (Cray, others): Expose Hierarchy

- Registers, Main Memory,
  Disk each available as storage alternatives;
- Tell programmers: “Use them cleverly”

Approach 2: Hide Hierarchy

- Programming model: SINGLE kind of memory, single address space.
- Machine AUTOMATICALLY assigns locations to fast or slow memory, depending on usage patterns.
Why We Care

CPU performance is dominated by memory performance. More significant than:
- ISA, circuit optimization, pipelining, super-scalar, etc

TRICK #1: How to make slow MAIN MEMORY appear faster than it is.
Technique: CACHEING

TRICK #2: How to make a small MAIN MEMORY appear bigger than it is.
Technique: VIRTUAL MEMORY
The Cache Idea: Program-Transparent Memory Hierarchy

GOALS:
1) Improve the average access time

\[ t_{ave} = \alpha t_c + (1 - \alpha)(t_c + t_m) = t_c + (1 - \alpha)t_m \]

\( \alpha \) HIT RATIO: Fraction of refs found in CACHE.
\( 1-\alpha \) MISS RATIO: Remaining references.

2) Transparency (compatibility, programming ease)

Challenge:
To make the hit ratio as high as possible.

Cache contains TEMPORARY COPIES of selected main memory locations... eg. Mem[100] = 37
How High of a Hit Ratio?

Suppose we can easily build an on-chip static memory with a 0.8 nS access time, but the fastest dynamic memories that we can buy for main memory have an average access time of 10 nS. How high of a hit rate do we need to sustain an average access time of 1 nS?

\[ \alpha = 1 - \frac{t_{ave} - t_c}{t_m} = 1 - \frac{1 - 0.8}{10} = 98\% \]

WOW, a cache really needs to be good?
**Cache**

Sits between CPU and main memory

Very fast table that stores a TAG and DATA

**TAG** is the memory address

**DATA** is a copy of memory at the address given by TAG

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>17</td>
</tr>
<tr>
<td>1040</td>
<td>1</td>
</tr>
<tr>
<td>1032</td>
<td>97</td>
</tr>
<tr>
<td>1008</td>
<td>11</td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>17</td>
</tr>
<tr>
<td>1004</td>
<td>23</td>
</tr>
<tr>
<td>1008</td>
<td>11</td>
</tr>
<tr>
<td>1012</td>
<td>5</td>
</tr>
<tr>
<td>1016</td>
<td>29</td>
</tr>
<tr>
<td>1020</td>
<td>38</td>
</tr>
<tr>
<td>1024</td>
<td>44</td>
</tr>
<tr>
<td>1028</td>
<td>99</td>
</tr>
<tr>
<td>1032</td>
<td>97</td>
</tr>
<tr>
<td>1036</td>
<td>25</td>
</tr>
<tr>
<td>1040</td>
<td>1</td>
</tr>
<tr>
<td>1044</td>
<td>4</td>
</tr>
</tbody>
</table>
Cache Access

On load we look in the TAG entries for the address we’re loading
Found → a *Hit*, return the DATA
Not Found → a *Miss*, go to memory for the data and put it and the address (TAG) in the cache

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>17</td>
</tr>
<tr>
<td>1040</td>
<td>1</td>
</tr>
<tr>
<td>1032</td>
<td>97</td>
</tr>
<tr>
<td>1008</td>
<td>11</td>
</tr>
</tbody>
</table>

Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>17</td>
</tr>
<tr>
<td>1004</td>
<td>23</td>
</tr>
<tr>
<td>1008</td>
<td>11</td>
</tr>
<tr>
<td>1012</td>
<td>5</td>
</tr>
<tr>
<td>1016</td>
<td>29</td>
</tr>
<tr>
<td>1020</td>
<td>38</td>
</tr>
<tr>
<td>1024</td>
<td>44</td>
</tr>
<tr>
<td>1028</td>
<td>99</td>
</tr>
<tr>
<td>1032</td>
<td>97</td>
</tr>
<tr>
<td>1036</td>
<td>25</td>
</tr>
<tr>
<td>1040</td>
<td>1</td>
</tr>
<tr>
<td>1044</td>
<td>4</td>
</tr>
</tbody>
</table>
Cache Lines

Usually get more data than requested (Why?)

A **LINE** is the unit of memory stored in the cache
usually much bigger than 1 word, 32 bytes per line is common
bigger LINE means fewer misses because of spatial locality
but bigger LINE means longer time on miss

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>17</td>
</tr>
<tr>
<td>1040</td>
<td>1</td>
</tr>
<tr>
<td>1032</td>
<td>97</td>
</tr>
<tr>
<td>1008</td>
<td>11</td>
</tr>
</tbody>
</table>
Finding the TAG in the Cache

A 1MByte cache may have 32k different lines each of 32 bytes. We can’t afford to sequentially search the 32k different tags. ASSOCIATIVE memory uses hardware to compare the address to the tags in parallel but it is expensive and 1MByte is thus unlikely.
Finding the TAG in the Cache

A 1MByte cache may have 32k different lines each of 32 bytes.

We can’t afford to sequentially search the 32k different tags.

**ASSOCIATIVE** memory uses hardware to compare the address to the tags in parallel but it is expensive and 1MByte is thus unlikely.

**DIRECT MAPPED CACHE** computes the cache entry from the address:
- multiple addresses map to the same cache line
- use TAG to determine if right

Choose some bits from the address to determine the Cache line:
- low 5 bits determine which byte within the line
- we need 15 bits to determine which of the 32k different lines has the data

which of the $32 - 5 = 27$ remaining bits should we use?
Direct-Mapping Example

- With 8 byte lines, the bottom 3 bits determine the byte within the line
- With 4 cache lines, the next 2 bits determine which line to use

1024d = 10000000000b → line = 00b = 0d
1000d = 01111101000b → line = 01b = 1d
1040d = 10000010000b → line = 10b = 2d
Direct Mapping Miss

• What happens when we now ask for address 1008?

1008d = 01111110000b → line = 10b = 2d

but earlier we put 1040d there...

1040d = 10000010000b → line = 10b = 2d

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>44</td>
</tr>
<tr>
<td>1000</td>
<td>17</td>
</tr>
<tr>
<td>1008</td>
<td>11</td>
</tr>
<tr>
<td>1016</td>
<td>29</td>
</tr>
</tbody>
</table>
Miss Penalty and Rate

The **MISS PENALTY** is the time it takes to read the memory if it isn’t in the cache.  
50 to 100 cycles is common.

The **MISS RATE** is the fraction of accesses which **MISS**.

The **HIT RATE** is the fraction of accesses which **HIT**.

MISS RATE + HIT RATE = 1

Suppose a particular cache has a **MISS PENALTY** of 100 cycles and a **HIT RATE** of 95%. The CPI for load on HIT is 5 but on a **MISS** it is 105. What is the average CPI for load?

Average CPI = 10

\[ 5 \times 0.95 + 105 \times 0.05 = 10 \]

Suppose **MISS PENALTY** = 120 cycles?

then CPI = 11 (slower memory doesn’t hurt much)
Some Associativity can help

Direct-Mapped caches are very common but can cause problems...

SET ASSOCIATIVITY can help.

Multiple Direct-mapped caches, then compare multiple TAGS

- 2-way set associative = 2 direct mapped + 2 TAG comparisons
- 4-way set associative = 4 direct mapped + 4 TAG comparisons

Now array size == power of 2 doesn’t get us in trouble

But

- slower
- less memory in same area
- maybe direct mapped wins...
What about store?

What happens in the cache on a store?

WRITE BACK CACHE \(\rightarrow\) put it in the cache, write on replacement
WRITE THROUGH CACHE \(\rightarrow\) put in cache and in memory

What happens on store and a MISS?

WRITE BACK will fetch the line into cache
WRITE THROUGH might just put it in memory