Chip Multiprocessors
Branch Prediction Preview
Computing the PC For non-branches (the wrong way)

- Non-branch instruction
  - $PC = PC + 4$
- When is PC ready?
Computing the PC for non-branches (the right way)

- Pre-decode in the fetch unit.
  - $PC = PC + 4$
- The PC is ready for the next fetch cycle.

```plaintext
add $s0, $t0, $t1
sub $t2, $s0, $t3
sub $t2, $s0, $t3
```
Computing the PC for Branches

• Branch instructions
  • \texttt{bne \$s1, \$s2, offset}
  • \texttt{if ($s1 \neq \$s2) \{ PC = PC + offset\} else \{PC = PC + 4;\}}

• When is the value ready?
Solution: Branch Prediction

• Can a processor tell the future?
• For non-taken branches, the new PC is ready immediately.
• Let’s just assume the branch is not taken
• Also called “branch prediction” or “control speculation”
• What if we are wrong?
• Branch prediction vocabulary
  • Prediction -- a guess about whether a branch will be taken or not taken
  • Misprediction -- a prediction that turns out to be incorrect.
  • Misprediction rate -- fraction of predictions that are incorrect.
Predict Not-taken

- We start the add, and then, when we discover the branch outcome, we *squash* it.
  - Also called “flushing the pipeline”
- All the work done for the squashed instructions is wasted!
And Branch Prediction Preview
Pipelining Recap

• Break up the logic with “pipeline registers” into pipeline stages
• Each stage can act on different instruction/data
• States/Control Signals of instructions are hold in pipeline registers (latches)
Pipelining Is Not Free
Pipelining Overhead

• Logic Delay (LD) -- How long does the logic take (i.e., the useful part)
• Set up time (ST) -- How long before the clock edge do the inputs to a register need be ready?
• Register delay (RD) -- Delay through the internals of the register.
• $CT_{\text{base}}$ -- cycle time before pipelining
  • $CT_{\text{base}} = LD + ST + RD$.
• $CT_{\text{pipe}}$ -- cycle time after pipelining $N$ times
  • $CT_{\text{pipe}} = ST + RD + LD/N$
  • Total time = $N \times ST + N \times RD + LD$
Pipelining Power

- Pipelining is a triple threat to power
  - Increased clock rate ($P \sim f^3$)
  - Pipeline register overheads
  - Increased cost of control hazards
Limit of OOO Superscalars

14 branches @ 80% accuracy = \(.8^{14} = 4.3\%\)
14 branches @ 90% accuracy = \(.9^{14} = 22\%\)
14 branches @ 95% accuracy = \(.95^{14} = 49\%\)
14 branches @ 99% accuracy = \(.99^{14} = 86\%\)
Multiprocessors

- Specifically, shared-memory multiprocessors have been around for a long time.
- Originally, put several processors in a box and provide them access to a single, shared memory.
- In the old days (< 2004), MPs were expensive and mildly exotic.
  - Big servers
  - Sophisticated users/data-center applications
Chip Multiprocessors (CMPS)

• Multiple processors on one die
• An easy way to spend xtrs
• Now common place
  • Laptops/desktops/game consoles/etc.
  • Less sophisticated users, all kinds of applications.
Big Core

- External Interface
- Instruction Fetch
- Instruction Cache (32 KB)
- TLB
- Data Cache (32 KB)
- Reorder Buffer, Instruction Queues, and Out-of-Order Logic
- On-Chip L2 Cache (256KB)
- Integer Unit
- Floating Point Unit
Small Cores

Diagram showing a layout of small cores with processors, caches, and other components.
Big Cores vs. Small Cores

The chart compares the Instruction Per Cycle (IPC) for Big Cores and Small Cores across various applications. The y-axis represents the IPC values ranging from 0 to 2.5, while the x-axis lists different applications such as compress, eqntott, m8kssim, MPsim, applu, apsi, swim, tomcatv, pmake, and Average. The chart shows a comparison between Small Core IPC (yellow bars) and Big Core IPC (blue bars) for each application.
Big Cores vs. Small Cores

% Misses/Instruction

- Small Core D-CACHE % MPCI
- Big Core D-CACHE % MPCI
Threads are Hard to Find

- To exploit CMP parallelism you need multiple processes or multiple “threads”
- **Processes**
  - Separate programs actually running (not sitting idle) on your computer at the same time.
  - Common in servers
  - Much less common in desktop/laptops
- **Threads**
  - Independent portions of your program that can run in parallel
  - Most programs are not multi-threaded.
- **We will refer to these collectively as “threads”**
  - A typical user system might have 1-8 actively running threads.
  - Servers can have more if needed (the sysadmins will hopefully configure it that way)
Parallel Programming is Hard

• Difficulties
  • Correctly identifying independent portions of complex programs
  • Sharing data between threads safely.
  • Using locks correctly
  • Avoiding deadlock

• There do not appear to be good solutions
  • We have been working on this for 30 years (remember, multi-processors have been around for a long time.)
  • It remains stubbornly hard.
Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```c
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
}
```

- If two threads execute this code, we would expect the shared_value to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.

Instructions in the two threads can be interleaved in any way.
Critical Sections and Locks

• By adding a lock, we can ensure that only one thread executes the critical section at a time.

```c
int shared_value = 0;
lock shared_value_lock;
void IncrementSharedVariable()
{
    acquire(shared_value_lock);
    int t = shared_value + 1;  // Line 1
    shared_value = t;          // line 2
    release(shared_value_lock);
}
```

• In this case we say shared_value_lock “protects” shared_value.
Locks are Hard

• The relationship between locks and the data they protect is not explicit in the source code and not enforced by the compiler
• In large systems, the programmers typically cannot tell you what the mapping is
• As a result, there are many bugs.
Locking Bug Example

```c
void Swap(int * a, lock * a_lock,
          int * b, lock * b_lock) {
    lock(a_lock);
    lock(b_lock);
    int t = a;
    a = b;
    b = t;
    unlock(a_lock);
    unlock(b_lock);
}
```

Thread 1

Swap(foo, foo_lock,
     bar, bar_lock);

Thread 2

Swap(bar, bar_lock,
     foo, foo_lock);

Thread 1 locks foo_lock, thread 2 locks bar_lock, both wait indefinitely for the other lock.
Finding, preventing, and fixing this kind of bug are all hard
End of CMPs