HPC VT 2013

Machine-dependent Optimization
• Choose good **data structures**
• Reduce number of operations
• Use cheap operations – strength reduction
• Avoid too many small function calls – **inlining**
• Use compiler **optimization flags**
• Help the compiler! Use **const** and **restrict**
Overview

• Instruction Cycle
  • Overview
  • Pipelining
  • Loop Unrolling
  • Loop Fusion

• Memory hierarchy
  • Overview
  • Registers
  • Data Locality
Basic operation:

- Modern computers are of LOAD-STORE type
- These machines store operands in “registers”
- That's a small scratch memory very close to the arithmetical units
- Data must be loaded from memory into a register, operated upon, and then stored back
Basic types of machine instructions:

- Data movement between memory and registers: “move”
- Arithmetic and logical operations: add, multiply, bitwise ops, ...
- Conditions and jumps: “goto”, conditional goto
- Procedure calls: “call” and “return”

Start:
```assembly
mov dx
mov bx, dx
mov b[ bx ], 5
mov ah, 0ah
inc bx
shl ax, 1
mov dx, 00
mov dl, [ bx ]
sub dx, 48
add ax, dx
loop usedigit
cmp n1, 00
jnz second
mov n1, ax
```
Execution in a CPU

"Machine Code"

"Data"

CPU
Architecture – too complex
CPU -- a simpler model
Instruction stages, example

1. Instruction fetch (IF)
   - Fetch instruction from registers

2. Instruction decode/register fetch (ID)
   - Decode instruction, read data for the operation

3. Execution (EX)
   - Execute on Arithmetic-Logic Unit

4. Memory access (MEM)
   - Access the memory (if necessary)

5. Write-back (WB)
   - Write to registers
add \ r3=r1+r2

Stage 1 (IF): fetch instruction
Stage 2 (ID): decode to determine the add, read registers $r1$ and $r2$
Stage 3 (EX): add the values
Stage 4 (MEM): nothing
Stage 5 (WB): write result into $r3$
Five common steps

Fetch Instruction | Read Register | ALU | Memory Access | Write Register

Execution time \(= N_{\text{instructions}} \times T_p\)
How "long" is a CPU cycle?

1982: 5MHz clock
1 cycle --> 200ns

2002: 3GHz clock
1 cycle --> 0.3ns

2012: 3GHz clock
1 cycle --> 0.3ns
Observations

- All steps are performed in succession
- Steps are waiting for work most of the time
- Logic is idling
- The CPI (cycles per instruction) is at least 5
Observations

- All steps are performed in succession
- Steps are waiting for work most of the time
- Logic is idling
- The CPI (cycles per instruction) is at least 5
- Solution: use pipelining
  - Start a new instruction each cycle
  - Each clock cycle becomes a pipe stage
  - Maximize use of logic
### Pipelining

Pipelining allows several instruction steps to execute simultaneously.

In this example, up to 5 x faster execution.

Requires that instructions are *independent*.

<table>
<thead>
<tr>
<th>Inst1</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inst2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst4</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst6</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Inst7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

---

-----> time ----->

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
</table>
Pipelines

Execution time = $N_{\text{instructions}} \times T_s$
Pipeline speedup

- For a 5-stage pipeline we can execute one instruction per cycle if the pipe is full.
- In general, a pipe of length n will give a speedup of n if we ignore startup cost.
- Although: *latency stays the same*.
- Same technique used in fabrication industry:
  - Cars, mobile phones etc.
Problems with pipelines:
Data hazards

- Works fine if we can start a new instruction every clock cycle
- Instructions must be independent
- If the pipeline detects a hazard it inserts “bubbles” to wait for dependencies

E.g. “write after read”

```
add  r1=r2+r3
mul  r4=r1*4
```
How to get more independent instructions?

• Loop unrolling – rewrite loop to do more work in each loop iteration

• Loop fusion – turn two (or more) loops into one
for(i = 0; i < N; i++) {
   // do something related to i
}

-->

for(i = 0; i < N; i+=3) {
   // do something related to i
   // do something related to i+1
   // do something related to i+2
}

// Afterwards: take care of remaining part, if any
// (needed in case N does not divide evenly by 3)
Loop fusion

for(i = 0; i < N; i++) {
    // do “work A” related to i
}

for(i = 0; i < N; i++) {
    // do “work B” related to i
}

-->

for(i = 0; i < N; i++) {
    // do “work A” related to i
    // do “work B” related to i
}
Problems with pipelines: Control hazards

- Most prominent: Branch hazards
- A branch can make instructions halfway through the pipe unnecessary
- The pipe needs to be flushed

*Branch prediction tries to take care of this.*
Programming implications: Avoiding branches

- Branching can be expensive
- Don’t repeat checking the same condition

```c
for (i = 0; i < 42; i++)
    if (a) dosomething(i);

if (a)
    for (i = 0; i < 42; i++)
        dosomething(i);
```
What compilers can do for you

• Compilers know which resources are available and how long instructions take
• Compilers schedule the instructions
• Tries to minimize pipeline bubbles
• Keep all resources busy (increase parallelism)
• Sometimes we will have bubbles due to hazards
• Branch prediction
• Out-of-order execution
• **Check Hardware-Performance-Counters**
Today: \(~10\text{-}20\) stages and \(4\text{-}6\) pipes

- + Shorter cycletime (more MHz)
- + ILP (parallel pipelines)
- - Branch delay even more expensive
- - Even harder to find "enough" independent instructions.
Multiple pipelines

Execution time = $N_{\text{instructions}} \times T_s / N_{\text{Pipelines}}$

Problems of the Pipeline remain!
Model of CPU with several pipelines
Modern memory: ~150 CPU cycles access time

- + Shorter cycle time (more MHz)
- - Memory access even more expensive.

- Memory access is slow.
Implications for pipelines

- Waiting for data stalls the pipeline
- We will need a lot of registers to hide this latency
- Solution: cache memories
- A cache memory is a smaller SRAM (faster) memory which acts as a temporary storage to hide the main memory latencies
MEMORY HIERARCHY
Example of Memory Hierarchy

**L0:** CPU registers hold words retrieved from L1 cache.

**L1:**
- on-chip L1 cache (SRAM)
- L1 cache holds cache lines retrieved from the L2 cache memory.

**L2:**
- off-chip L2 cache (SRAM)
- L2 cache holds cache lines retrieved from main memory.

**L3:** main memory (DRAM)
- Main memory holds disk blocks retrieved from local disks.

**L4:**
- local secondary storage (local disks)
- Local disks hold files retrieved from disks on remote network servers.

**L5:**
- remote secondary storage (distributed file systems, Web servers)

---

Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices
Memory Bus

- CPU chip
  - Cache memories
  - Register file
  - ALU
- Bus interface
- System bus
- Memory bus
- I/O bridge
- Main memory
Registers

- Registers are a scarce resource
  - Store words (32/64 bit)
  - Special registers for floating-point, SIMD (128/256/512 bits!)
- The compiler tries to maximize the usage of the registers
  - Called register allocation
- If you run out of registers, you must temporarily store results back to memory and then retrieve them again
  - Register spill
  - Degrades performance
There are two keywords that control register allocation from C

The `register` keyword suggests (forces?) a variable to a register

– Used for heavily accessed variables
– Today, most compilers can figure this out themselves
– You cannot take the address of something that is stored in a register

The `volatile` keyword forces the results to be written back to memory

– Used in low-level and parallel programming
// tell compiler that a should be stored in a register
register int a = 23;

// force b to be read from and written back to memory
volatile int b = 43;
Caches
Caches

- Several levels of cache with varying latency
- L1 (approx. 10-100 kB) 1-5 cycles
- L2 (256 kB-) 10-20 cycles
- L3 (> 2 MB), slower
- Caches organized into "cache lines", approx. 128 bytes each
Cache optimization

Try to increase the chances that the needed memory locations are already in cache
Reorder data accesses to improve data locality
Modern prefetchers are great, can help.
for(i=0; i<N; ++i){
    ...
}

-->

for(j=0; j<N; j+=B)
    for(i=j; i<min(N, j+B); ++i){
        ....
    }

General Caching Concepts

- “Cache hit”
- “Cache miss”
- Cache Performance Metrics: "hit rate" or "miss rate"
Caches and performance

- Caches are extremely important for performance
- Level 1 latency is usually 1 or 2 cycles
- Work well for problems with nice locality properties
- Caching can be used in other areas as well, example: web-caching (proxies)
- Modern CPUs have two or three levels of cache
- Most of the chip area is used for caches
How caches work -- cache lines

- Cache divided into **cache lines**
- When memory is read into cache, a whole cache line is always read at the same time
- Good if we have data locality: nearby memory accesses will be fast
- Typical cache line size: 64-128 bytes
How do caches work?

- **"Direct-mapped cache"**: Hash function maps to a single place for every unique address
- **"Fully associative cache"**: Cache space is divided into n sets. Address is distributed modulo n. k-way set associative (k = 2...24)
- **"Set associative cache"**: Maps to any slot. Hard to build in practice
To synchronize data in hierarchies caches can either be:

• **Write-through**
  - Reflect change immediately
  - L1 is often write-through

• **Write-back**
  - Synchronize all data at a given signal
  - Less traffic, but other drawbacks instead
SSE SIMD instructions

- Single instruction, multiple data (SIMD)
- Streaming SIMD Extensions (SSE)
  --> "SSE SIMD" instructions
- Allows increased performance by doing the same operation for several data entries at once.
- As with pipelining, requires independent instructions.
- Tricky to use in your own code. Intrinsics. The compiler can (hopefully) do this for us
- Heavily used in optimized linear algebra libraries