CMPE 110: Computer Architecture

Week 2
Performance / ISA

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

[Adapted in part from Jose Renau, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

Reminder

- Midterm 1
  - Wednesday Oct. 5, in class
  - Location: classroom + overflow room
  - Five questions
  - Lecture notes up to Monday Oct. 3: Performance, ISA

Review: Performance

- Performance metrics: Latency & throughput
- Comparing performance: Speedup
- Averaging performance:
  - Arithmetic mean
  - Harmonic mean
  - Geometric mean
- Measuring CPU performance: CPI (IPC)
- Amdahl's Law
  - How much does an optimization improve performance?

Today:

- Benchmarking
- Instruction set architecture (ISA)
  - What is ISA?
  - Execution model
    - Program execution model
    - Instruction execution model
  - ISA design goals

Processor Performance and Workloads

- Q: what does performance of a chip mean?
- A: nothing, there must be some associated workload
  - **Workload**: set of tasks someone (you) cares about

- **Benchmarks**: standard workloads
  - Used to compare performance across machines
  - Either are or highly representative of actual programs people run

- **Micro-benchmarks**: non-standard workloads
  - Tiny programs used to evaluate certain aspects of performance
  - Not representative of complex behaviors of real applications
  - Examples: binary tree search, towers-of-hanoi, 8-queens, etc.

SPEC CPU 2006

- Latency SPEC
  - For each benchmark
    - Take odd number of latency samples
    - Choose median
    - Take speedup (latency ratio of reference machine / your machine)
  - Take “average” (mean) of ***ratios*** over all benchmarks
- Throughput SPEC
  - Run multiple benchmarks in parallel on multiple-processor system
- Recent (latency) leaders
  - SPECint: Intel Xeon E3-1280 v3 (63.7)
  - SPECfp: Intel Xeon E5-2690 2.90 GHz (96.6)

GeekBench

- Set of cross-platform multicore benchmarks
  - Can run on iPhone, Android, laptop, desktop, etc
- Tests integer, floating point, memory, memory bandwidth performance
- GeekBench stores all results online
  - Easy to check scores for many different systems, processors
- Pitfall: Workloads are simple, may not be a completely accurate representation of performance
  - We know they evaluate compared to a baseline benchmark

GeekBench Numbers

- Desktop (4 core Ivy bridge at 3.4GHz): 11456
- Laptop:
  - MacBook Pro (13-inch) - Intel Core i7-3520M 2900 MHz (2 cores) - 7807
- Phones:
  - iPhone 5 - Apple A6 1000 MHz (2 cores) – 1589
  - iPhone 4S - Apple A5 800 MHz (2 cores) – 642
  - Samsung Galaxy S III (North America) - Qualcomm Snapdragon S3 MSM8960 1500 MHz (2 cores) - 1429

PARSEC

What is PARSEC?

- Princeton Application Repository for Shared-Memory Computers
- Benchmark Suite for Chip-Multiprocessors
- Started as a cooperation between Intel and Princeton University, many more have contributed since then
- Freely available at:

http://parsec.cs.princeton.edu/

Objective of PARSEC

- Multithreaded Applications
  - Future programs must run on multiprocessors
- Emerging Workloads
  - Increasing CPU performance enables new applications
- Diverse
  - Multiprocessors are being used for more and more tasks
- State-of-Art Techniques
  - Algorithms and programming techniques evolve rapidly

Workloads

Program	Application Domain	Parallelization
Blackscholes	Financial Analysis	Data-parallel
Bodytrack	Computer Vision	Pipeline NEW!
Canneal	Engineering	Data-parallel NEW!
Dedup	Enterprise Storage	Pipeline
Facesim	Animation	Data-parallel
Ferret	Similarity Search	Pipeline
Fluidanimate	Animation	Data-parallel
Freqmine	Data Mining	Data-parallel
Raytrace NEW!	Visualization	Data-parallel
Streamcluster	Data Mining	Data-parallel
Swaptions	Financial Analysis	Data-parallel
Vips	Media Processing	Data-parallel
X264	Media Processing	Pipeline

CloudSuite

A Suite for Emerging Scale-out Applications

http://parsa.epfl.ch/cloudsuite/cloudsuite.html

Data Analytics
Machine learning

Data Caching
Memcached

Data Serving
Cassandra NoSQL

Graph Analytics
TunkRank

Media Streaming
Apple Quicktime Server

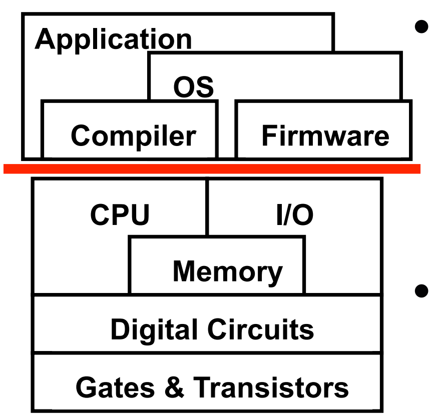
SW Testing as a Service
Symbolic constraint solver
Cloud9

Web Search
Apache Nutch

Web Serving
Nginx, PHP server

Instruction Set Architecture (ISA)

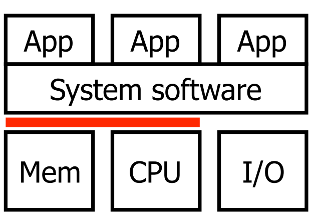
ISA Overview – Instruction Set Architecture



- What is an ISA?
  - An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor. <Wikipedia>
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two "philosophies": CISC/RISC
    - Difference is blurring
- Good ISA…
  - Enables high-performance
  - At least doesn't get in the way

Execution Model

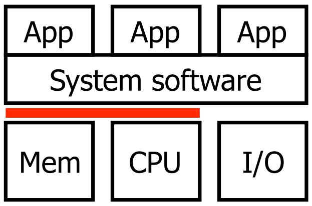
Program Compilation




```
int array[100], sum;
void array_sum() {
   for (int i=0; i<100;i++) {
      sum += array[i];
   }
}
```

- **Program** written in a "high-level" programming language
  - C, C++, Java, C#
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Compiler also optimizes
  - Compiler is itself a program…who compiled the compiler?

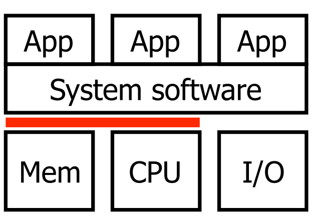
Assembly & Machine Language



- **Assembly language**
  - Human-readable representation
- **Machine language**
  - Machine-readable representation
  - 1s and 0s (often displayed in "hex")
- **Assembler**
  - Translates assembly to machine

Machine code	*Assembly code*
x9A00	CONST R5, #0
x9200	CONST R1, array
xD320	HICONST R1, array
x9464	CONST R2, sum
xD520	HICONST R2, sum
x6640	LDR R3, R1, #0
x6880	LDR R4, R2, #0
x18C4	ADD R4, R3, R4
x7880	STR R4, R2, #0
x1261	ADD R1, R1, #1
x1BA1	ADD R5, R5, #1
x2B64	CMPI R5, #100
x03F8	BRn array_sum_loop

Example Assembly Language & ISA

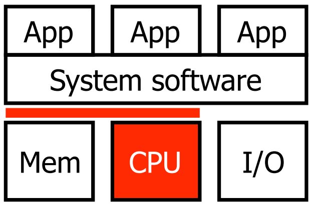


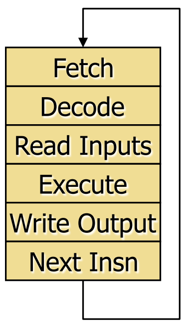
- **ARM**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 63 registers
    - 31 integer, 32 floating point
  - ~100 different insns

Example code is ARM, but all ISAs are pretty similar

```
cmp r1, #0
   push     {r4}
   ble      .L4
   movs     r3, #0
   subs     r2, r0, #4
   mov      r0, r3
.L3:
   adds     r3, r3, #1
   ldr      r4, [r2, #4]!
   cmp      r3, r1
   add      r0, r0, r4
   bne      .L3
.L2:
   pop      {r4}
   bx       lr
.L4:
   movs     r0, #0
   b .L2
```

Instruction Execution Model





Instruction → Insn

- A computer is just a finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Called "instruction pointer" in x86
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)
- **Program is just "data in memory"**
  - Makes computers programmable ("universal")

What is an ISA? -- In more detail

What Is An ISA?

- **ISA (instruction set architecture)**
  - A well-defined hardware/software interface
  - The **"contract"** between software and hardware
    - **Functional definition** of storage locations & operations
      - Storage locations: registers, memory
      - Operations: add, multiply, branch, load, store, etc
    - **Precise description** of how to invoke & access them
- Not in the contract: non-functional aspects
  - How operations are implemented
  - Which operations are fast and which are slow and when
  - Which operations take more power and which take less
- Instructions (Insns)
  - Bit-patterns hardware interprets as commands

ARM ADD Documentation

ADD[S] : Addition

ADD will add two values.

Operand 1 is a register, operand 2 can be a register, shifted register, or an immediate value (which may be shifted).

If the S bit is set (***ADDS***), the N and Z flags are set according to the result, and the C and V flags are set as follows: **C** if the result generated a carry (unsigned overflow); **V** if the result generated a signed overflow.

ADD is useful for basic addition. Use ADC to perform addition with the Carry flag considered.

Syntax

```
ADD<suffix>  <dest>, <op 1>, <op 2>
```

Function

```
dest = op_1 + op_2
```

Technical

The instruction bit pattern is as follows:

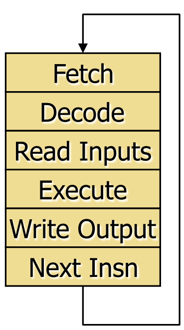
31 - 28	27	26	25	24 - 21	20	19 - 16	15 - 12	11 - 0
condition	0	0	I	0 1 0 0	S	op_1	dest	op_2/shift

Note: If the I bit is *zero*, and bits 4 and 7 are both *one* (with bits 5,6 zero), the instruction is UMULL, not ADD.

A Language Analogy for ISAs

- Communication
  - Person-to-person → software-to-hardware
- Similar structure `adds r3, r3, #1`
  - Narrative → program
  - Sentence → insn
  - Verb → operation (add, multiply, load, branch)
  - Noun → data item (immediate, register value, memory value)
  - Adjective → addressing mode
- Many different languages, many different ISAs
  - Similar basic structure, details differ (sometimes greatly)
- Key differences between languages and ISAs
  - Languages evolve organically, many ambiguities, inconsistencies
  - ISAs are explicitly engineered and extended, unambiguous

The Sequential Model



- **Basic structure of all modern ISAs**
  - Often called Von Neumann, but in ENIAC before
- **Program order**: total order on dynamic insns
  - Order and **named storage** define computation
- Convenient feature: **program counter (PC)**
  - Insn itself stored in memory at location pointed to by PC
  - Next PC is next insn unless insn says otherwise
- Processor logically executes loop at left
- **Atomic**: insn finishes before next insn starts
  - Implementations can break this constraint physically
  - But must maintain illusion to preserve correctness

ISA Design Goals

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Performance/Implementability**
  - Easy to design high-performance implementations?
  - Easy to design low-power implementations?
  - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain as languages, programs, and technology evolve?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2, Core i7, …

Programmability

- Easy to express programs efficiently?
  - For whom?
- Before 1980s: **human**
  - Compilers were terrible, most code was hand-assembled
  - Want high-level coarse-grain instructions
    - As similar to high-level language as possible
- After 1980s: **compiler**
  - Optimizing compilers generate much better code that you or I
  - Want low-level fine-grain instructions
    - Compiler can't tell if two high-level idioms match exactly or not
- This shift changed what is considered a "good" ISA…

Implementability

- Every ISA can be implemented
  - Not every ISA can be implemented efficiently
- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)
- Certain ISA features make these difficult
  - – Variable instruction lengths/formats: complicate decoding
  - – Special-purpose registers: complicate compiler optimizations
  - – Difficult to interrupt instructions: complicate many things

What Makes a Good ISA?

- **Programmability**
  - Easy to express programs efficiently?
- **Performance**/**Implementability**
  - Easy to design high-performance implementations?
  - Easy to design low-power implementations?
  - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain as languages, programs, and technology evolve?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2, Core i7, …

Recall: CPU Performance Equation

- Latency = seconds / program =
  - (insns / program) * (cycles / insn) * (seconds / cycle)
  - **Insns / program**: insn count
    - Impacted by program, compiler, ISA
  - **Cycles / insn**: CPI
    - Impacted by program, compiler, ISA, micro-arch
  - **Seconds / cycle**: clock period (Hz)
    - Impacted by micro-arch, technology
- For low latency (better performance) minimize all three
  – Difficult: often pull against one another
  - Example we have seen: RISC vs. CISC ISAs
    - ± RISC: low CPI/clock period, high insn count
    - ± CISC: low insn count, high CPI/clock period

Example: Instruction Granularity

Execution time =
(instructions/program) * (seconds/cycle) * (cycles/instruction)

- **CISC** (Complex Instruction Set Computing) **ISAs**
  - Big heavyweight instructions (lots of work per instruction)
  - \+ Low "insns/program"
  - – Higher "cycles/insn" and "seconds/cycle"
    - We have the technology to get around this problem
- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - Minimalist approach to an ISA: simple insns only
  - \+ Low "cycles/insn" and "seconds/cycle"
  - – Higher "insn/program", but hopefully not as much
    - Rely on compiler optimizations

CISC vs. RISC

- CISC (M68000):

```
Add      (A3)+,   100(A2)
```

Add the content of memory location pointed to by A3 to the component of an array starting at memory address 100. The index number of the component is in A2. The content of A3 is then automatically incremented by 1.

- RISC (MIPS):

```
Lw    $t0, 0($s3)
Lw    $t1, 100 ($s2)
Add   $t2, $t0, $t1
Sw    $t2, 100($s3)
Addi  $s3, $s3, 1
```

The Debate

- RISC argument
  - CISC is fundamentally handicapped
  - For a given technology, RISC implementation will be better (faster)
    - Current technology enables single-chip RISC
    - When it enables single-chip CISC, RISC will be pipelined
    - When it enables pipelined CISC, RISC will have caches
    - When it enables CISC with caches, RISC will have next thing...

- CISC rebuttal
  - CISC flaws not fundamental, can be fixed with **more transistors**
  - Technology advancement will narrow the RISC/CISC gap (true)
    - Good pipeline: RISC = 100K transistors, CISC = 300K
    - By 1995: 2M+ transistors had evened playing field
  - Software costs dominate, **compatibility** is paramount