CSE 240A:
Principles in Computer Architecture
Week 2
Performance / ISA

Jishen Zhao (https://cseweb.ucsd.edu/~jzhao/)

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

Review: Performance Metrics

- Performance metrics: Latency & throughput
- Comparing performance: Speedup
- Averaging performance:
  - Arithmetic mean
  - Harmonic mean
  - Geometric mean
- Measuring CPU performance: CPI (IPC)

CPI Example

- Assume a processor with instruction frequencies and costs
  - Integer ALU: 50%, 1 cycle
  - Load: 20%, 5 cycle
  - Store: 10%, 1 cycle
  - Branch: 20%, 2 cycle

- Which change would improve performance more?
  - A. "Branch prediction" to reduce branch cost to 1 cycle?
  - B. Faster data memory to reduce load cost to 3 cycles?
- Compute CPI
  - Base = 0.5*1 + 0.2*5 + 0.1*1 + 0.2*2 = 2 CPI
  - A = 0.5*1 + 0.2*5 + 0.1*1+ 0.2*1 = 1.8 CPI (1.11x or 11% faster)
  - B = 0.5*1 + 0.2*3 + 0.1*1 + 0.2*2 = 1.6 CPI (1.25x or 25% faster)
    - **B is the winner**

Today: Performance

- Metrics
  - Latency and throughput
  - Speedup
  - Averaging
- CPU Performance

Today: Performance / ISA

- Other CPU performance metrics
- Performance laws
  - Amdahl's law
  - Little's law
- Benchmarking
- Instruction set architecture (ISA)
  - What is ISA?
  - Instruction execution model
  - ISA design goals

Other CPU performance metrics

- MIPS
  - Million instructions per second <- IPC
- FLOPS
  - The number of floating point instructions per second
  - GFLOPS
  - Will visit again when we discuss GPU architectures

Example

Instruction	Percentage	CPI_M1	CPI_M2
ALU	50%	1	2
Branches	15%	2	1
Loads	20%	2	1
Stores	15%	1	1

- What is the average MIPS for M1 and M2?
  - Clock frequency is 1GHz →1 cycle is 1ns

Answer

Instruction	Percentage	CPI_M1	CPI_M2
ALU	50%	1	2
Branches	15%	2	1
Loads	20%	2	1
Stores	15%	1	1

- What is the average MIPS for M1 and M2?
  - Clock frequency is 1GHz →1 cycle is 1ns
  - M1: CPI = 0.5*1+0.15*2+0.2*2+0.15*1 = 1.35
    - 1 million instructions take $1.35*10^6$ ns, i.e., $1.35 * 10^{-3}$ s, to run
    - MIPS = $1 / (1.35*10^{-3}) = 741$
  - M2: CPI = 0.5*2 + 0.15*1 + 0.2*1+ 0.15*1 = 1.5
    - MIPS = $1 / (1.5*10^{-3}) = 667$

Example

Instruction	Percentage	CPI_M1	CPI_M2
ALU	50%	1	2
Branches	15%	2	1
Loads	20%	2	1
Stores	15%	1	1

- What is the peak MIPS for M1 and M2?
  - Clock frequency is 1GHz →1 cycle is 1ns

Example

Instruction	Percentage	CPI_M1	CPI_M2
ALU	50%	1	2
Branches	15%	2	1
Loads	20%	2	1
Stores	15%	1	1

- What is the peak MIPS for M1 and M2?
  - Clock frequency is 1GHz →1 cycle is 1ns
  - Both M1 and M2:
    - $CPI_{peak} = 1$
    - 1 instruction takes 1 cycle, i.e., 1ns = 10^{-9} s, to run
    - 1 million instructions take 10^{-3} s to run
    - MIPS = $(1*10^{-6})/(1*10^{-9}) = 1000$

Summary: Performance Metrics

- Performance metrics
  - Latency & throughput
- Comparing performance
  - Speedup
- Averaging performance
  - Arithmetic mean
  - Harmonic mean
  - Geometric mean
- Measuring CPU performance
  - CPI (IPC), MIPS, FLOPS, etc.

Performance Laws

Amdahl's Law

$$\frac{1}{(1-P)+\frac{P}{S}}$$

How much will an optimization improve performance?

P = proportion of running time affected by optimization

S = speedup

What if I speedup 25% of a program's execution by 2 times?

1.14x speedup

What if I speedup 25% of a program's execution by ∞?

1.33x speedup

Amdahl's Law for Parallelization

$$\frac{1}{(1-P)+\frac{P}{N}}$$

How much will parallelization improve performance?

P = proportion of parallel code

N = threads

"Max speedup", so N = ∞

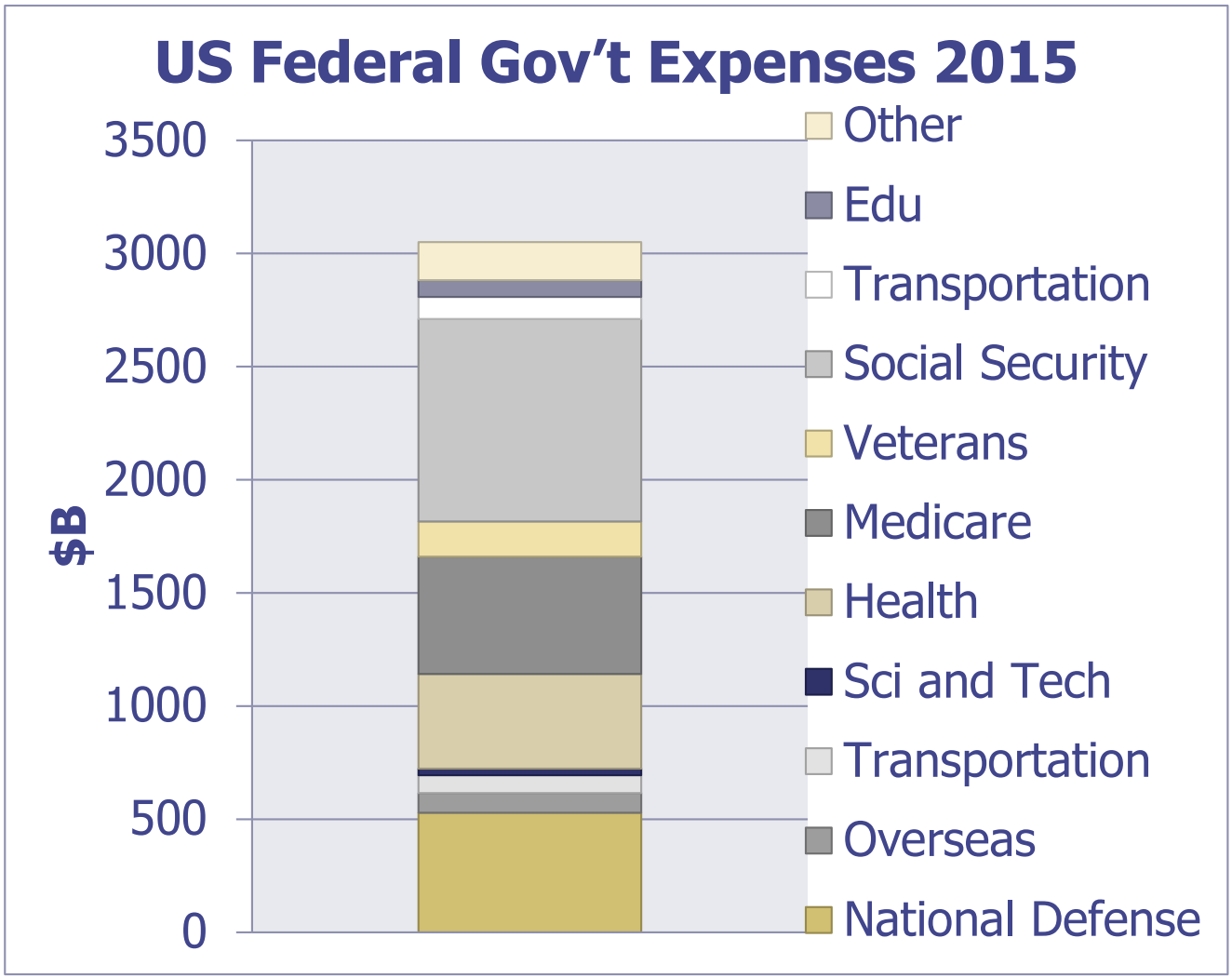
- What is the max speedup for a program that's 10% serial?

10x speedup

- What about 1% serial?

100x speedup

Amdahl's Law for the US Budget



https://en.wikipedia.org/wiki/2015_United_States_federal_budget

Scrapping Dept of Transportation ($81B) cuts budget by 2.6%

Example: Forget about the formula for a while

- Assume a program has the following instruction type breakdown:
  - 40% memory, latency is 4 cycles
  - 50% adds, latency is 2 cycles
  - 10% multiplies, latency is 16 cycles
- If you could pick one type of instructions to make twice as fast (half of latency) in the next-generation of this processor, which instruction type would you pick?

Answer

- Assume a program has the following instruction type breakdown:
  - 40% memory, latency is 4 cycles
  - 50% adds, latency is 2 cycles
  - 10% multiplies, latency is 16 cycles
- Pick one type of instructions to make twice as fast (half of latency)
  - Base CPI: 0.4*4 + 0.5*2 + 0.1*16 = 4.2
  - Make memory instructions 2x faster:
    - CPI_1 = 0.4*2 + 0.5*2 + 0.1*16 = 3.4
  - Make adds 2x faster: CPI_2 = (0.4*4 + 0.5*1 + 0.1*16 = 3.7
  - Make multiplies 2x faster: CPI_3 = (0.4*4 + 0.5*2 + 0.1*8 = 3.4

What we learned

- Amdahl's Law:

How much does
an optimization improve performance?

$$\frac{1}{(1-P)+\frac{P}{S}}$$


How long am I going to be in this line?
Use Little's Law!

Little's Law

$$L = \lambda W$$

L = items in the system

λ = average arrival rate

W = average wait time

- Assumption:
  - system is in steady state, i.e., average arrival rate = average departure rate
- No assumptions about:
  - arrival/departure/wait time distribution or service order (FIFO, LIFO, etc.)
- Works on **any** queuing system
- Works on **systems of systems**

Little's Law for Computing Systems

- Only need to measure two of L, λ and W
  - often difficult to measure L directly
- Describes how to meet performance requirements
  - e.g., to get high throughput (λ), we need either:
    - low latency per request (small W)
    - service requests in parallel (large L)
- Addresses many computer performance questions
  - sizing queue of L1, L2, L3 misses
  - sizing queue of outstanding network requests for 1 machine
    - or the whole datacenter
  - calculating average latency for a design

Benchmarking

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 or 2017)

- 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?

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

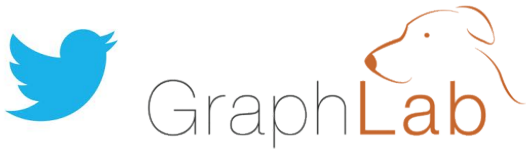
http://parsa.epfl.ch/cloudsuite/cloudsuite.html

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

Web Search
Apache Nutch

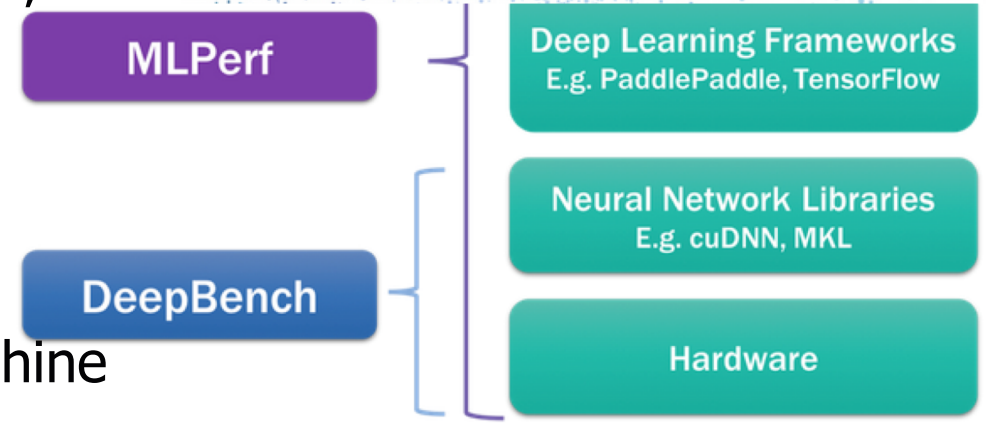
Web Serving
Nginx, PHP server

MLPerf – Machine learning benchmarks

- Measure system performance
  - Training and inference
  - From mobile devices to cloud services
- 74 supporting companies
- Frameworks
  - TensorFlow, Caffe2, PyTorch, PaddlePaddle
- Applications
  - Speech recognition, text to speech, image recognition, and machine translation

MLPerf

A broad ML benchmark suite for measuring performance of ML software frameworks, ML hardware accelerators, and ML cloud platforms.



Contributions by researchers from

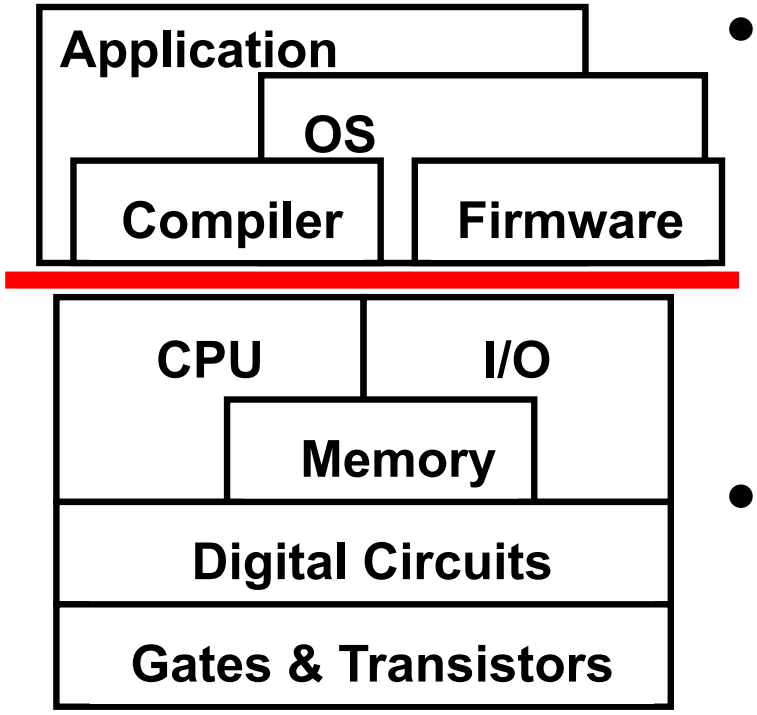
Harvard University, Stanford University, University of Arkansas, Littlerock, University of California, Berkeley, University of Illinois, Urbana Champaign, University of Minnesota, University of Texas, Austin, University of Toronto

Summary: Performance

- Performance metrics: Latency & throughput
- Comparing performance: Speedup
- Averaging performance:
  - Arithmetic mean
  - Harmonic mean
  - Geometric mean
- Measuring CPU performance: CPI (IPC)
- Performance laws
  - Amdahl's law
  - Little's law
- Benchmarks

Instruction Set Architecture (ISA)

ISA Overview

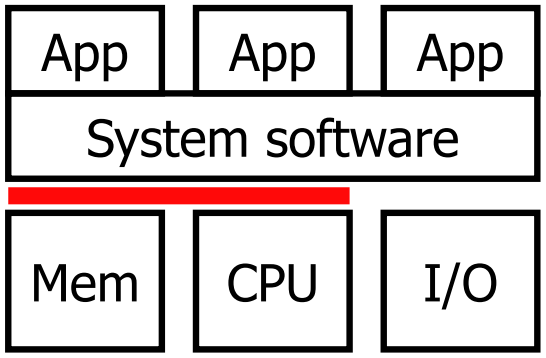


- What is an ISA?
  - An ISA includes a specification of the set of opcodes (machine language), and the native commands implemented by a particular processor. <Wikipedia>
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two "philosophies": CISC/RISC
    - Difference is blurring

- Good ISA…
  - Enables high-performance
  - At least doesn't get in the way
- Compatibility is a powerful force

Execution Model

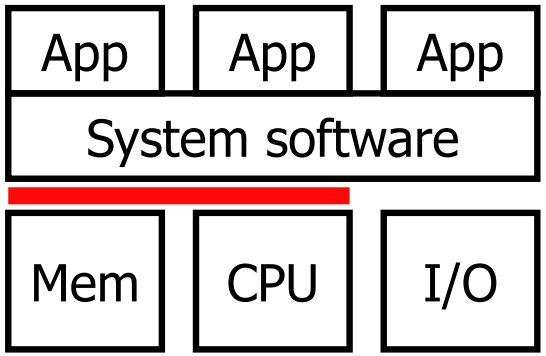
Program Compilation




```
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?

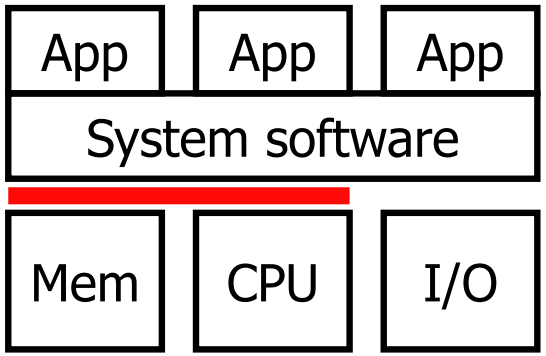
Assembly & Machine Language



- **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



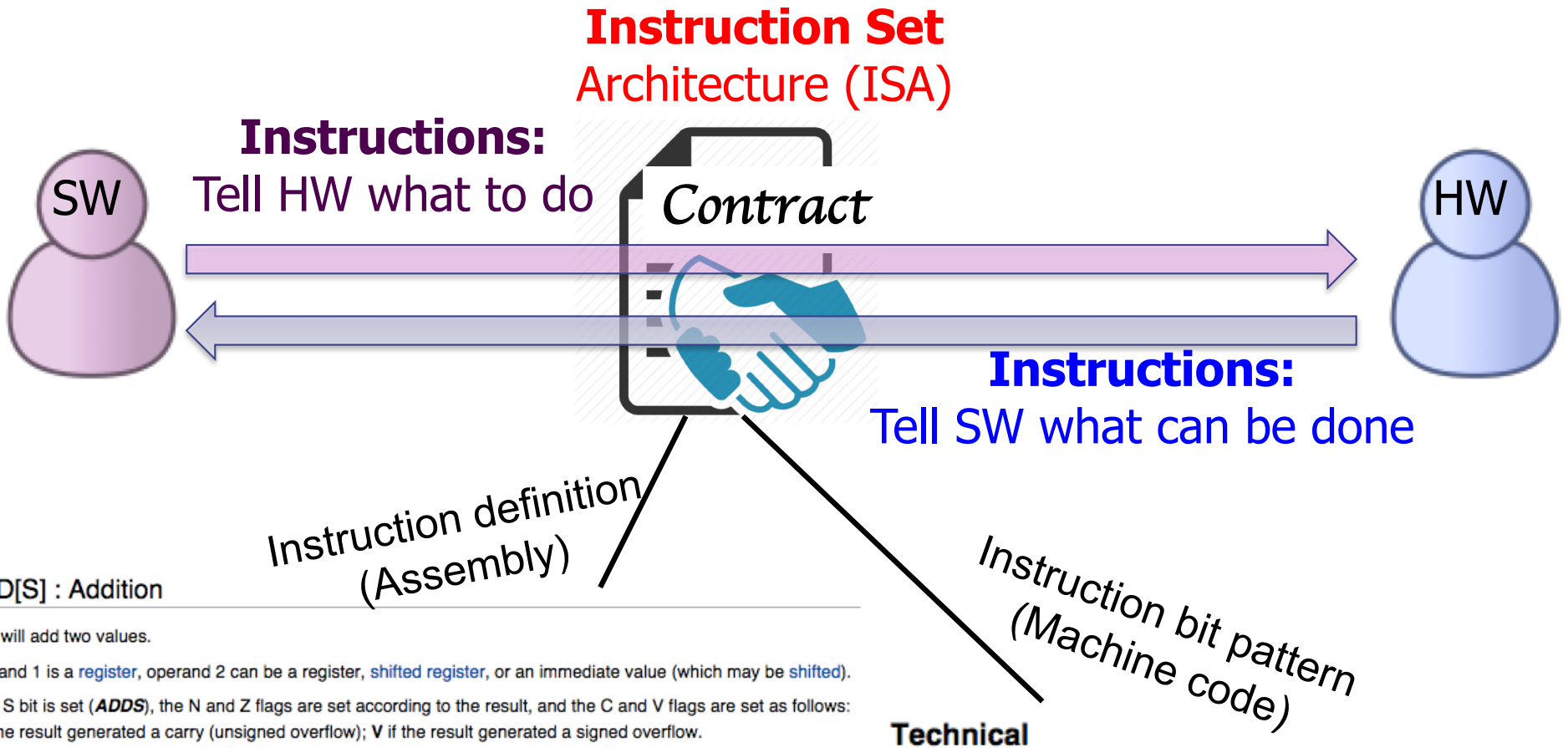
- **ARM**: example of real ISA
  - 32/64-bit operations
  - 32-bit insns
  - 63 registers
    - 31 integer, 32 floating point
  - ~100 different insns

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
```

What is an ISA?

What is ISA?



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), t

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

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

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
```

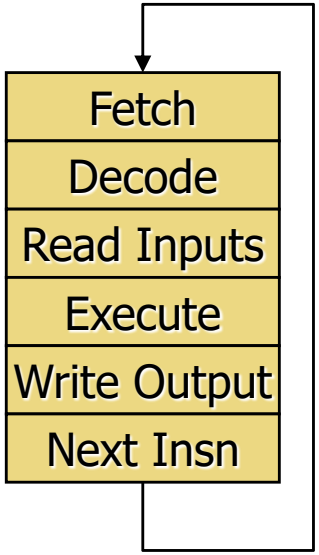
Technical

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

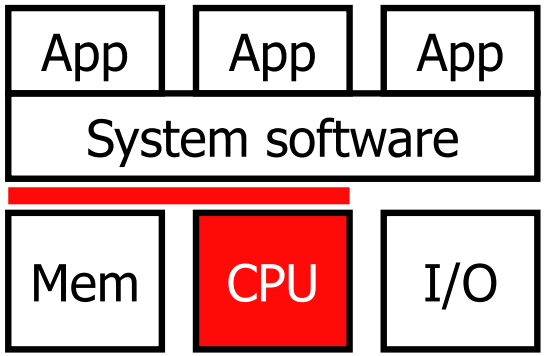
Note: If the I bit is *zero*, and bits 4 and 7 are both *one* (with bits 5,6 zero), the instruction is UMULL, not ADD.

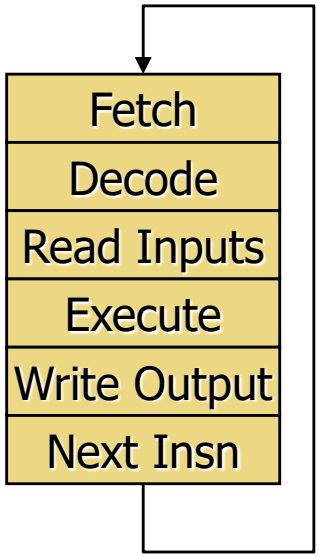
The Sequential Model



- **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

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")

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 MM location pointed to by A3 to the component of an array starting at MM 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
```

RISC vs. CISC

- 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**
  - Moore's Law 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
- More on RISC vs. CISC debate later…

Compiler Optimizations

- Primarily goal: reduce instruction count
  - Eliminate redundant computation, keep more things in registers
    + Registers are faster, fewer loads/stores
    – An ISA can make this difficult by having too few registers
- But also…
  - Reduce branches and jumps (later)
  - Reduce cache misses (later)
  - Reduce dependences between nearby insns (later)
    – An ISA can make this difficult by having implicit dependences
- How effective are these?
  + Can give 4X performance over unoptimized code
  – Collective wisdom of 40 years ("Proebsting's Law"): 4% per year
  - Funny but … shouldn't leave 4X performance on the table

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, …

Compatibility

- In many domains, ISA must remain compatible
  - IBM's 360/370 (the *first* "ISA family")
  - Another example: Intel's x86 and Microsoft Windows
    - x86 one of the worst designed ISAs EVER, but it survives
- **Backward compatibility**
  - New processors supporting old programs
    - **Hard to drop features**
    - Update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
  - Old processors supporting new programs
    - Include a "CPU ID" so the software can test for features
    - Add ISA hints by overloading no-ops (example: x86's PAUSE)
    - New firmware/software on old processors to emulate new insn

Translation and Virtual ISAs

- New compatibility interface: ISA + translation software
  - **Binary-translation** (static): transform static image, run native
  - **Emulation** (dynamic binary-translation): unmodified image, interpret each dynamic insn
    - Typically optimized with just-in-time (JIT) compilation
  - Examples: DEC FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
  - Performance overheads reasonable (many advances over the years)

- **Virtual ISAs**: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime), NVIDIA's "PTX"

Ultimate Compatibility Trick

- Support old ISA by…
  - …having a simple processor for that ISA somewhere in the system
  - How did PlayStation2 support PlayStation1 games?
    - Used PlayStation processor for I/O chip **& emulation**

What we learned

- What is ISA?
- Execution model:
  - Compilation
  - Assembly & machine language
- Instruction execution model
  - Registers, memory, PC
  - Instruction execution
- ISA design goals
  - Programmability
  - Performance/implementability
  - Compatibility