L03: Performance; Instruction Set Architecture 1

Adapted from:

CS152: Computer Architecture and Engineering
Dave Patterson (www.cs.berkeley.edu/~pattrsn)

Performance

Recap: Measuring CPU performance

CPU time = seconds / program

CPU time = (cycles/program) x (sec/cycle)

CPU time = (cycles/instruction) x (instructions/program) x (sec/cycle)

CPI is the average number of clock cycles each instruction takes to execute

CPI

"Average cycles per instruction"

CPI = (CPU Time * Clock Rate) / Instruction Count
= Clock Cycles / Instruction Count

$$\text{CPU time} = \text{ClockCycleTime} * \sum_{i=1}^{n} \text{CPI}_i * I_i$$

Number of instructions of type *i*

$$\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i * F_i \quad \text{where} \quad F_i = \frac{I_i}{\text{Instruction Count}}$$

"instruction frequency"

Example (RISC processor)

Op	Freq	Cycles	CPI(i)
ALU	50%	1	.5
Load	20%	5	1.0
Store	10%	3	.3
Branch	20%	2	.4
			2.2

Typical Mix

Recap: Measuring CPU performance

CPU time = seconds / program

CPU time = (cycles/program) x (sec/cycle)

CPU time = (cycles/instruction) x
(instructions/program) x (sec/cycle)

CPI is the average number of clock cycles each instruction takes to execute

Performance is always relative to a specific program

Choosing programs to evaluate performance

- Use benchmarks
- But there is no ideal benchmark
- The best programs to use for benchmarks are real applications

Basis of Evaluation

	Pros	Cons
Actual Target Workload	• representative	• very specific • non-portable • difficult to run, or measure • hard to identify cause
Full Application Benchmarks	• portable • widely used • improvements useful in reality	• less representative
Small "Kernel" Benchmarks	• easy to run, early in design cycle	• easy to "fool"
Microbenchmarks	• identify peak capability and potential bottlenecks	• "peak" may be a long way from application performance

SPEC benchmarks

(Standard Performance Evaluation Corporation)

- **In use since the 1980s – current version CPU2000**
- **Eighteen application benchmarks (with inputs) reflecting a technical computing workload**
- **Eight integer**
  - **go, m88ksim, gcc, compress, li, ijpeg, perl, vortex**
- **Ten floating-point intensive**
  - **tomcatv, swim, su2cor, hydro2d, mgrid, applu, turb3d, apsi, fppp, wave5**
- **Must run with standard compiler flags**
  - **eliminate special undocumented incantations that may not even generate working code for real programs**

How do we speed things up?

- **For a given instruction set architecture, performance increases come from**
  - **Increases in clock rate [better technology]**
  - **Improvements in processor organisation that lower CPI [our main focus], and**
  - **Compiler enhancements that lower the instruction count or generate instructions with a lower average CPI**

How to we speed things up?

Invest resources where time is spent

Base Machine (Reg / Reg)

Op	Freq	Cycles	CPI(i)	% Time
ALU	50%	1	.5	23%
Load	20%	5	1.0	45%
Store	10%	3	.3	14%
Branch	20%	2	.4	18%
			2.2	

Typical Mix

How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

How does this compare with using branch prediction to shave a cycle off the branch time?

What if two ALU instructions could be executed at once?

Measuring speedup

- **Speedup due to enhancement *E*:**

$$speedup(E) = \frac{ex\_time(\ )}{ex\_time(E)} = \frac{performance(E)}{performance(\ )}$$

Amdahl's Law

"The performance enhancement possible with a given improvement is limited by the amount that the improved feature is used"

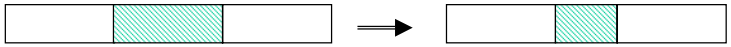
Amdahl's Law

- **Suppose that enhancement *E* accelerates a fraction *F* of the task by a factor *S* and the remainder of the task is unaffected. Then**

$$ex\_time(E) = ((1-F) + F/S) \times ex\_time(\ )$$

$$ex\ time\ after\ improv\ = \frac{ex\ time\ affected\ by\ improv}{amount\ of\ improv} + ex\ time\ unaffected$$

- **Speedup is limited by the amount that the improved feature is used**



$$speedup(E) = \frac{1}{(1-F) + F/S}$$

Example: Amdahl's Law

- **Suppose a program runs in 100 seconds on a machine, with multiply operations responsible for 80 seconds of this time. How much does the speed of the multiplication operation have to be improved for the program to run five times faster?**

$$\frac{100}{5} = \frac{80}{x} + 20$$

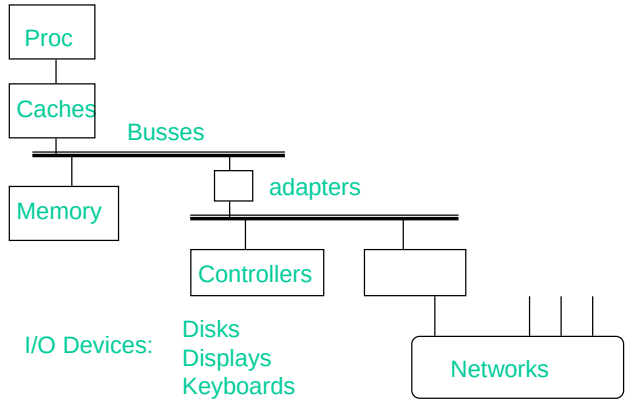
$$x \rightarrow \infty$$

Instruction Set Architecture 1

Review: Organisation

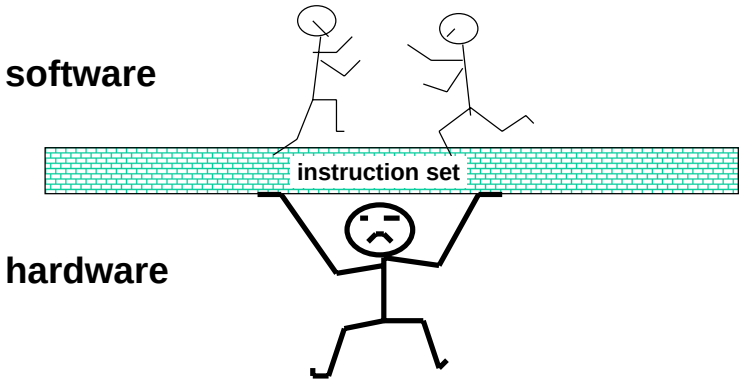
- All computers consist of five components
  - Processor: (1) datapath and (2) control
  - (3) Memory
  - (4) Input devices and (5) Output devices
- Not all "memory" are created equally
  - Cache: fast (expensive) memory are placed closer to the processor
  - Main memory: less expensive memory — we can have more
- Input and output (I/O) devices have the messiest organisation
  - Wide range of speed: graphics vs. keyboard
  - Wide range of requirements: speed, standard, cost ...
  - Least amount of research (so far)

Summary: Computer System Components



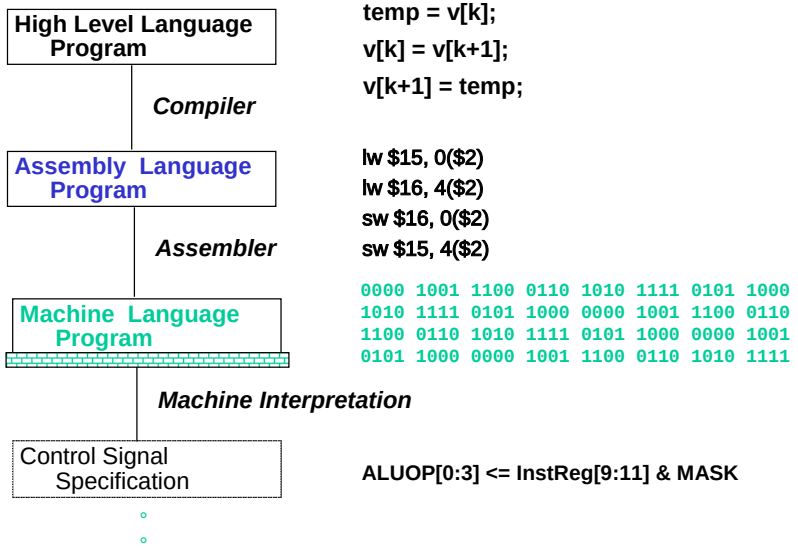
- All have interfaces & organisations

Review: Instruction Set Design

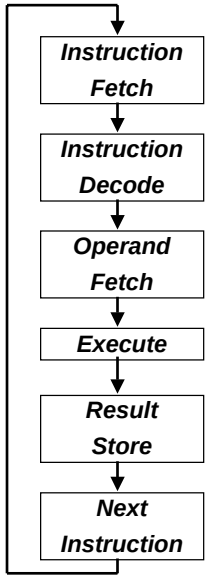


Which is easiest to change?

Levels of Representation



Execution Cycle



Obtain instruction from program storage

Determine required actions and instruction size

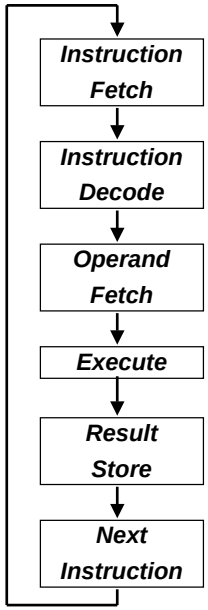
Locate and obtain operand data

Compute result value or status

Deposit results in storage for later use

Determine successor instruction

Instruction Set Architecture: What Must be Specified?



- Instruction Format or Encoding
  - how is it decoded?
- Location of operands and result
  - where other than memory?
  - how many explicit operands?
  - how are memory operands located?
  - which can or cannot be in memory?
- Data type and Size
- Operations
  - what are supported
- Successor instruction
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*

Basic ISA Classes

Accumulator (1 register):		
1 address	add A	acc ← acc + mem[A]
1+x address	addx A	acc ← acc + mem[A + x]
Stack:		
0 address	add	tos ← tos + next
General Purpose Register:		
2 address	add A B	EA(A) ← EA(A) + EA(B)
3 address	add A B C	EA(A) ← EA(B) + EA(C)
Load/Store:		
3 address	add Ra Rb Rc	Ra ← Rb + Rc
2 address	load Ra Rb	Ra ← mem[Rb]
	store Ra Rb	mem[Rb] ← Ra

Comparison:

Bytes per instruction? Number of Instructions? Cycles per instruction?

Comparing Number of Instructions

Code sequence for C = A + B for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

Instruction set design principles

- The four design principles:
  - Simplicity favours regularity
  - Smaller is faster
  - Good design demands a compromise
  - Make the common case fast

Example application of principles: MIPS

- MIPS instructions are all 32-bits large (simple because of regularity)
  - Arithmetic instructions always have three operands
  - Operands of arithmetic instructions are in registers
- MIPS has 32 registers each with 32 bits (smaller is faster)
- MIPS has three instruction formats (a compromise that makes for good design)
- Immediate values are provided in I-type instructions (making a common occurrence fast)

- MIPS has therefore become popular, and is used by NEC, Nintendo, SGI, and Sony, and is typical of ISAs designed since the early 1980s

General Purpose Registers Dominate

° Since mid 1970's all machines use general purpose registers

° Advantages of registers
  - registers are faster than memory
  - registers are easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack
  - registers can hold variables
    - memory traffic is reduced, so program is sped up (since registers are faster than memory)
    - code density improves (since register named with fewer bits than memory location)

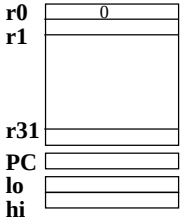
Summary on Instruction Classes

- Expect new instruction set architecture to use general purpose registers

- Pipelining => Expect it to use load store variant of GPR ISA because it is a clean abstraction with just one access to memory required per instruction

MIPS I Registers

- Programmable storage
  - 2^32 x bytes of memory
  - 31 x 32-bit GPRs (R0 = 0)
  - 32 x 32-bit FP regs (paired DP)
  - HI, LO, PC

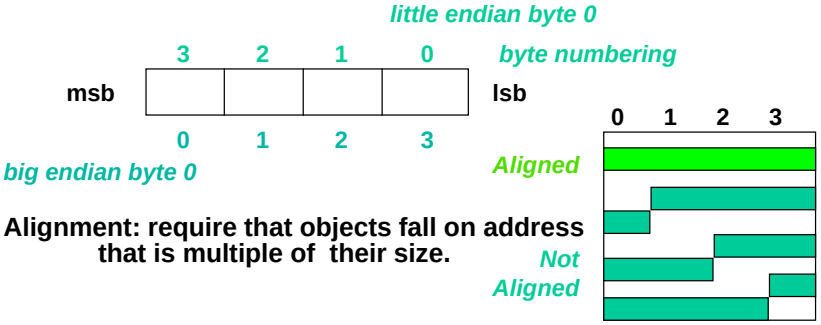


Memory Addressing

- Since 1980 almost every machine uses addresses to level of 8-bits (byte)
- 2 questions for design of ISA:
  - Since could read a 32-bit word as four loads of bytes from sequential byte addresses or as one load word from a single byte address, how do byte addresses map onto words?
  - Can a word be placed on any byte boundary?

Addressing Objects: Endianess and Alignment

- Big Endian: address of most significant byte = word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- Little Endian: address of least significant byte = word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Alignment: require that objects fall on address that is multiple of their size.

Possible addressing modes

Addressing mode	Example	Meaning
Register	Add R4,R3	R4← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2–d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]

Why Auto-increment/decrement? Scaled?