

CS 203A
Advanced Computer Architecture

Lecture 2

Instruction Set Principles

Instructor: Jun Yang

9/28/2004

Lec. 2

1

Outline

- Instruction formats
- Types of operations
- Operand type and size
- Location of operands
- Addressing modes

9/28/2004

Lec. 2

2

Instruction Sets

- "ISA is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct program for that machine"
- What makes a good instruction set?
 - Implementability
 - Supports from low to high performance implementations

Easy, trap to sw to emulate complex instructions

Pipelining, parallelism, dynamic scheduling

- Programmability
 - Pre-1980: Human programmability; ISA semantically close to high-level language (HLL); but with different HLL ?
 - Post-1980: Compiler programmability; Primitive instructions from which solutions are synthesized;

9/28/2004

Lec. 2

3

Instruction Sets

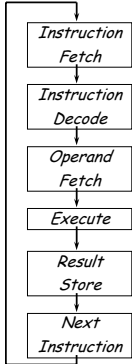
- What makes a good instruction set?
 - Upward/Forward/Backward compatibilities (make sure all written software works); business reality: software cost greater than hardware cost; forward compatibility: reserve trap hooks to emulate future ISA extensions

9/28/2004

Lec. 2

4

Instruction Set Architecture (ISA)



What must be specified?

- **Instruction Format, length, encoding**
 - How is it decoded?
- **Operations, data types and sizes**
 - What are supported?
- **Location of operands and result**
 - Where other than memory?
 - How many explicit operands?
 - How are memory operands located?
- **Control instruction**
 - Jumps, conditions, branches

9/28/2004

Lec. 2

5

Generic Examples of Instruction Format Widths

Fixed:

- 32 or 64 bits
- Easy pipelining/superscalar
- Less compact

Variable:

- More compact
- Harder (but doable) to superscalar/pipeline

Hybrid:

- Recently appeared
- e.g. in embedded processors, use 32-bit added 16-bit for code compaction

9/28/2004

Lec. 2

6

Effects of Instruction Formats on Code Size

- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions
- Recent embedded machines (ARM, MIPS) added optional mode to execute subset of 16-bit wide instructions (Thumb, MIPS16); per procedure decide performance or density
- Some architectures actually exploring on-the-fly decompression for more density.
 - PowerPC (IBM) compresses code in memory, decompresses it in instruction cache using special hardware.
 - Branch targets are handled through an address mapping table for uncompressed and compressed code.

9/28/2004

Lec. 2

7

Typical Operations (little change since 1960)

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

9/28/2004

Lec. 2

8

Top 10 80x86 Instructions

Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

° Simple instructions dominate instruction frequency

9/28/2004

Lec. 2

9

Operations for Media and Signal Processing

- SIMD (single-instruction multiple-data, vector)
 - A single instruction launch multiple narrower data operations (add, multiply, compare, shift, etc.). See Figure 2.17 on Pg. 110.
- For DSPs:
 - No exception handling for arithmetic overflow
 - Because of real-time application
 - Multiply-accumulate (MAC) instruction.
 - Key to dot product ops for vector and matrix multiplies.
 - Different rounding modes for wide accumulators.

9/28/2004

Lec. 2

10

Data Types and Sizes

- Distinction between s/w types and h/w types
 - s/w: type is property of a variable/constant
 - h/w: type is property of an operation
- Most common types:
 - Character (8 bits), half-word (16 bits), word (32 bits), doubleword (64 bits), single-precision (32 bits) double-precision (64 bits) fp.
 - Fixed point types
 - Signed (-2^{n-1} to $2^{n-1}-1$) - add, sub, mul, div, etc.
 - Unsigned (0 to 2^n-1) - addu, subu, mulu, divu, ... logic&bitwise ops
- Operands for Media and Signal Processing
 - Vertex (x, y, z-coordinates) and Pixels (RGBA) - 32 bits.

Fixed point

9/28/2004

Lec. 2

11

Location of Operands and Result

Most real machines are hybrids of these:

Stack: (mostly 60's and 70's, java bytecodes)

0 address add tos ← tos + next

- Good code density (tos implicit)
- Memory and pipelining bottlenecks

Accumulator (1 register): (pre 60's)

1 address add A acc ← acc + mem[A]

1+x address addx A acc ← acc + mem[A + x]

- Good code density
- Memory bottleneck

Register-Memory: (extended accumulator, 70's and 80's)

2 address add A B A ← A + B (mem[B])

3 address add A B C A ← B + C (mem[B/C])

- Good code density
- Asymmetric operands, asymmetric work per instruction

9/28/2004

Lec. 2

12

Location of Operands and Result

Register-register (60's and onwards):

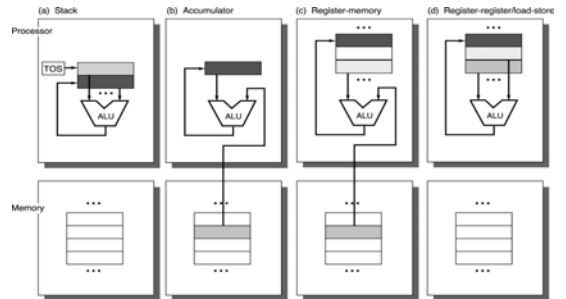
3 address `add Ra, Rb, Rc` $Ra \leftarrow Rb + Rc$
 `Load Ra, Rb` $Ra \leftarrow \text{mem}[Rb]$
 `store Ra, Rb` $\text{mem}[Rb] \leftarrow Ra$

- not good for code density
- Operand symmetry
- Deterministic length ALU operations
- Scheduling opportunities, etc.

Comparison:

Bytes per instruction? Number of instructions? Cycle per instruction?

Location of Operands and Result



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

General Purpose Registers (GPR) Dominate

- After 1980, almost 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 - shorter identifier

Two characteristics of GPR architectures

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS PowerPC, SPARC, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000
2	2	Memory-memory	VAX
3	3	Memory-memory	VAX

Advantages ? Disadvantages? (Read Figure 2.4 on Pg95!)

Memory Addressing

- Since 1980, almost every machine uses addresses to level of 8-bits (byte)
- Two questions for design of ISA:
 - Since one could read a 32-bit word as four loads of bytes from sequential byte address of 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?*

Endian-ness

- Order of bytes in words
- Little endian stores the most significant byte (MSB) in the byte with the littlest address
 - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
- Big endian stores the MSB in the byte with the biggest address
 - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

Store **Oxdeadbeef** to address 0x10000000:

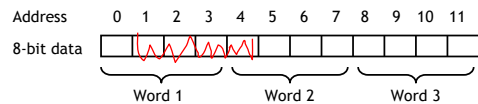
Address: 0x10000000 0x10000001 0x10000002 0x10000003

Big Endian: 0xef 0xbe 0xad 0xde

Little Endian: 0xde 0xad 0xbe 0xef

Memory alignment

- Keep in mind that memory is byte-addressable, so a 32-bit word actually occupies four contiguous locations (bytes) of main memory.



- The MIPS architecture requires words to be **aligned** in memory: 32-bit words must start at an address that is divisible by 4.
 - 0, 4, 8 and 12 are valid **word addresses**.
 - 1, 2, 3, 5, 6, 7, 9, 10 and 11 are *not* valid word addresses.
 - Unaligned memory accesses result in a **bus error**, which you may have unfortunately seen before.
- This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

Addressing Modes

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

9/28/2004

Lec. 2

21

Addressing Modes for Signal Processing

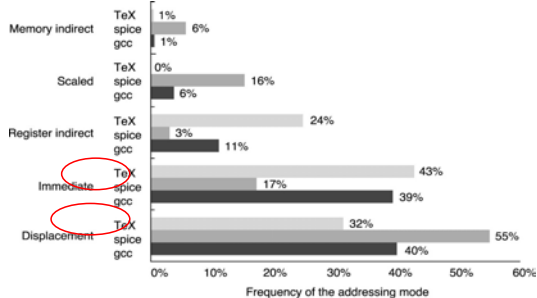
- Besides Immediate, Displacement, Register indirect and Direct modes, DSP supports instructions that perform **circular buffer access** and **Fast Fourier Transformation (FFT)**.
 - Due to frequency of these types of operations!

9/28/2004

Lec. 2

22

Usage of Addressing Mode



© 2003 Elsevier Science (USA). All rights reserved.

9/28/2004

Lec. 2

23

Addressing Mode Summary

- Data addressing modes that are important:
 - Register-register, displacement, immediate.
- Displacement size should be 12 to 16 bits
 - This range account for 75% to 99% of the displacement.
- Immediate size should be 8 to 16 bits
 - This range captures 50% to 80% of the immediates.

9/28/2004

Lec. 2

24