

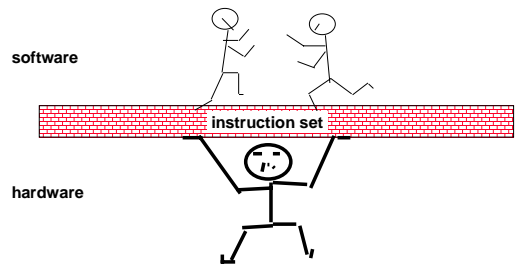


ECE 4680: Computer Architecture and Organization

Instruction Set Architecture

Different styles of ISA.
Basic issues when designing an ISA.
What a good ISA should be?

Instruction Set Design



An instruction is a binary code, which specifies a basic operation (e.g. add, subtract, and, or) for the computer

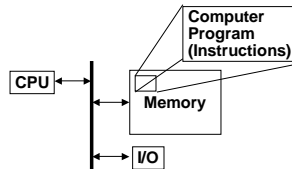
- **Operation Code:** defines the operation type
- **Operands:** operation source and destination

Instruction Set Architecture

Programmer's View

ADD 01010
SUBTRACT 01110
AND 10011
OR 10001
COMPARE 11010
.
.
.

Computer's View



Princeton (Von Neumann) Architecture

- Data and Instructions mixed in same memory ("stored program computer")
- Program as data (dubious advantage)
- Storage utilization
- Single memory interface

Harvard Architecture

- Data & Instructions in separate memories
- Has advantages in certain high performance implementations

Basic Issues in Instruction Set Design

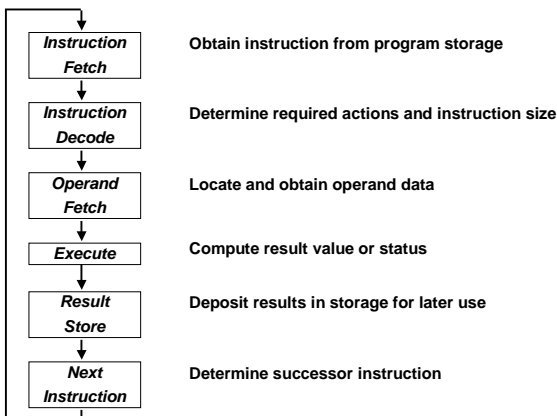
- What operations (and how many) should be provided
LD/ST/INC/BRN sufficient to encode any computation
But not useful because programs too long!
- How (and how many) operands are specified
Most operations are dyadic (eg, $A \leftarrow B + C$)
Some are monadic (eg, $A \leftarrow -B$)
- How to encode these into consistent instruction formats
Instructions should be multiples of basic data/address widths

Typical instruction set:

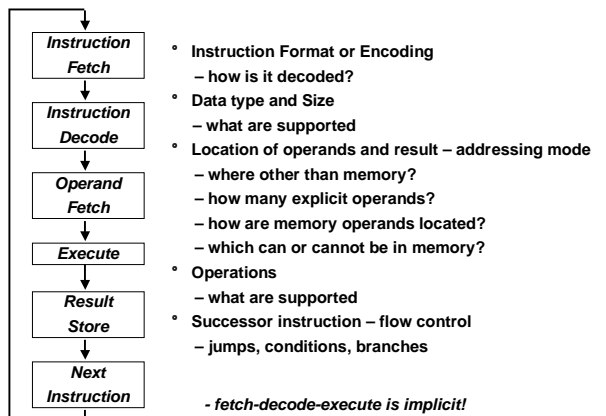
- 32 bit word
- basic operand addresses are 32 bits long
- basic operands, like integers, are 32 bits long
- in general case, instruction could reference 3 operands ($A := B + C$)

challenge: encode operations in a small number of bits!

Execution Cycle



What Must be Specified?



Topics to be covered

We will discuss the following topics which determines the Complexity of IS.

- Instruction Format or Encoding
 - how is it decoded?
- Data type and Size
 - what are supported
- Location of operands and result – addressing mode
 - where other than memory?
 - how many explicit operands?
 - how are memory operands located?
 - which can or cannot be in memory?
- Operations
 - what are supported
- Successor instruction – flow control
 - jumps, conditions, branches

Basic ISA Classes

Accumulator: (earliest machines)

1 address add A $acc \leftarrow acc + mem[A]$

1+x address addx A $acc \leftarrow acc + mem[A + x]$

Stack: (HP calculator, Java virtual machines)

0 address add $tos \leftarrow tos + next$

General Purpose Register: (e.g. Intel 80x86, Motorola 68xxx)

2 address add A B $EA(A) \leftarrow EA(A) + EA(B)$

3 address add A B C $EA(A) \leftarrow EA(B) + EA(C)$

Load/Store: (e.g. SPARC, MIPS, PowerPC)

3 address add Ra Rb Rc $Ra \leftarrow Rb + Rc$

load Ra Rb $Ra \leftarrow mem[Rb]$

store Ra Rb $mem[Rb] \leftarrow Ra$

Comparison:

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

Comparing Instructions

Comparing Number of Instructions

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

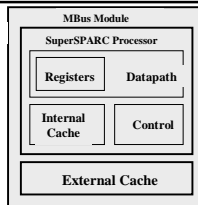
Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add C	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

General Purpose Registers Dominate

- Since 1975 all machines use general purpose registers (Java Virtual Machine adopts Stack architecture)
- 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)

More About register?

- integrated together in process chip.
- In top level in memory hierarchy.
- Faster to access and simpler to use.
- Special role in MIPS ISA: only registers not symbolic variables can be in instructions.
- The number of registers can not be too more, not be too less.
- Effective use of registers is a key to program performance.



Examples of Register Usage

Number of memory addresses per typical ALU instruction	Maximum number of operands per typical ALU instruction	Examples
0	3	SPARC, MIPS, Precision Architecture, Power PC
1	2	Intel 80x86, Motorola 68000
2	2	VAX (also has 3-operand formats)
3	3	VAX (also has 2-operand formats)

Example:

In VAX: **ADDL (R9), (R10), (R11)**
 $\text{mem}[\text{R9}] \leftarrow \text{mem}[\text{R10}] + \text{mem}[\text{R11}]$

In MIPS: `lw R1, (R10);` load a word
`lw R2, (R11)`
`add R3, R1, R2;` $\text{R3} \leftarrow \text{R1} + \text{R2}$
`sw R3, (R9);` store a word

Pros and Cons of Number of Memory Operands/Operands

Register-register: 0 memory operands/instr, 3 (register) operands/instr

- + Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute
- Higher instruction count than architectures with memory references in instructions. Some instructions are short and bit encoding may be wasteful.

Register-memory (1,2)

- + Data can be accessed without loading first. Instruction format tends to be easy to encode and yields good density.
- Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory Address in each instruction may restrict the number of registers. Clocks per instruction varies by operand location.

Memory-memory (3,3)

- + Most compact. Doesn't waste registers for temporaries.
- Large variation in instruction size, especially for three-operand instructions. Also, large variation in work per instruction. Memory accesses create memory bottleneck.

Summary on Instruction Classes

- o Expect new instructin set architecture to use general purpose register
- o Pipelining => Expect it to use load store variant of GPR ISA

Memory addressing

- **BYTE Addressing:**
 - Since 1980, almost every machine uses addresses to level of 8-bits
- **Two Questions for design of ISA**
 - For a 32-bit word, read it as four loads of bytes from sequential byte addresses or as one load work from a single byte address. How byte address map onto words ?
 - Can a word be placed on any byte boundary?

Addressing Objects

Big Endian: address of most significant **IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA**

Little Endian: address of least significant **Intel 80x86, DEC Vax**

Word:	3	2	1	0	little endian word 0:
	msb			lsb	
	0	1	2	3	big endian word 0:

Alignment: require that objects fall on address that is multiple of their size. (p 112)

BIG Endian versus Little Endian (P113 & A-46)

Example 1: Memory layout of a number #ABCD

In Big Endian: \rightarrow CD \$1001
 AB \$1000

In Little Endian: \rightarrow AB \$1001
 CD \$1000

Example 2: Memory layout of a number #FF00

Byte Swap Problem



Memory layout of a number of ABCDEFGH

Each system is self-consistent, but causes problems when they need communicate!

Addressing Modes

Addressing mode	Example	Meaning
Immediate	Add R4,#3	$R4 \leftarrow R4+3$
Register	Add R4,R3	$R4 \leftarrow R4+R3$
Register indirect	Add R4,(R1)	$R4 \leftarrow R4+Mem[R1]$
Displacement	Add R4,100(R1)	$R4 \leftarrow R4+Mem[100+R1]$
Indexed	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)	$R1 \leftarrow R1+Mem[Mem[R3]]$
Auto-increment	Add R1,(R2)+	$R1 \leftarrow R1+Mem[R2]; R2 \leftarrow R2+d$
Auto-decrement	Add R1,-(R2)	$R2 \leftarrow R2-d; R1 \leftarrow R1+Mem[R2]$
Scaled	Add R1,100(R2)[R3]	$R1 \leftarrow R1+Mem[100+R2+R3*d]$

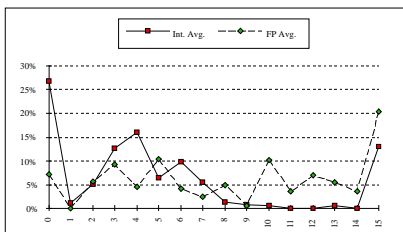
Addressing Mode:

- Addressing modes have the ability to significantly reduce instruction counts
- They also add to the complexity of building a machine

Addressing Mode Usage

- 3 programs avg, 17% to 43%
- Register deferred (indirect): 13% avg, 3% to 24%
 - Scaled: 7% avg, 0% to 16%
 - Memory indirect: 3% avg, 1% to 6%
 - Misc: 2% avg, 0% to 3%

Displacement Address Size



- Average of 5 programs from SPECint92 and Average of 5 programs from SPECfp92
- X-axis is in powers of 2: \Rightarrow addresses $> 2^3$ (8) and $< 2^4$ (16)
- 1% of addresses > 16 -bits

Immediate Size

- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits

Addressing Summary

- Data Addressing modes that are important:
Displacement, Immediate, Register Indirect
- Displacement size should be 12 to 16 bits
- Immediate size should be 8 to 16 bits

ECE4680 Lec 3 ISA-25

February 6, 2002

Typical Operations

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
Logical	not, and, or, set, clear
Shift	shift left/right, rotate left/right
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate

ECE4680 Lec 3 ISA-26

February 6, 2002

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

ECE4680 Lec 3 ISA-27

February 6, 2002

Methods of Testing Condition

- Condition Codes
Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.
ex: `add r1, r2, r3`
`bz label`
- Condition Register
Ex: `cmp r1, r2, r3`; compare r2 with r3, 0 or 1 is stored in r1
`bgt r1, label`; branch on greater
- Compare and Branch
Ex: `bgt r1, r2, label`; if $r1 > r2$, then go to label

ECE4680 Lec 3 ISA-28

February 6, 2002

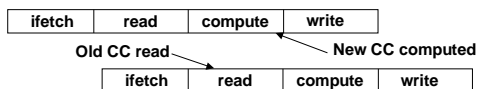
Condition Codes

Setting CC as side effect can reduce the # of instructions

X: . . SUB r0, #1, r0 BRP X	versus	X: . . SUB r0, #1, r0 CMP r0, #0 BRP X
--------------------------------------	--------	--

But also has disadvantages:

- not all instructions set the condition codes which do and which do not often confusing!
e.g., shift instruction sets the carry bit
- dependency between the instruction that sets the CC and the one that tests it: to overlap their execution, may need to separate them with an instruction that does not change the CC



ECE4680 Lec 3 ISA-29

February 6, 2002

Branches

--- Conditional control transfers

Four basic conditions:
N -- negative
Z -- zero
V -- overflow
C -- carry

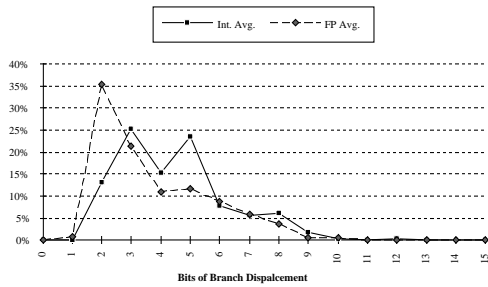
Sixteen combinations of the basic four conditions:

Always	Unconditional
Never	NOP
Not Equal	$\sim Z$
Equal	Z
Greater	$\sim[Z + (N \otimes V)]$
Less or Equal	$Z + (N \otimes V)$
Greater or Equal	$\sim(N \otimes V)$
Less	$N \otimes V$
Greater Unsigned	$\sim(C + Z)$
Less or Equal Unsigned	$C + Z$
Carry Clear	$\sim C$
Carry Set	C
Positive	$\sim N$
Negative	N
Overflow Clear	$\sim V$
Overflow Set	V

ECE4680 Lec 3 ISA-30

February 6, 2002

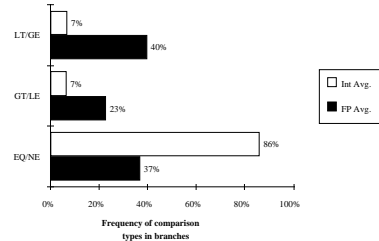
Conditional Branch Distance



- Distance from branch in instructions $2^i \Rightarrow \pm 2^{i-1}$
- 25% of integer branches are $> 2^2$

Conditional Branch Addressing

- PC-relative since most branches at least 8 bits suggested (± 128 instructions)
- Compare Equal/Not Equal most important for integer programs



Operation Summary

- Support these simple instructions, since they will dominate the number of instructions executed:

load,
store,
add,
subtract,
move register-register,
and,
shift,
compare equal, compare not equal,
branch (with a PC-relative address at least 8-bits long),
jump,
call,
return;

Data Types

Bit: 0, 1

Bit String: sequence of bits of a particular length

- 4 bits is a nibble
- 8 bits is a **byte**
- 16 bits is a **half-word**
- 32 bits is a **word**

Character:

- ASCII 7 bit code
- EBCDIC 8 bit code (IBM)
- UNICODE 16 bit code (Java)

Decimal:

- digits 0-9 encoded as 0000b thru 1001b
- two decimal digits packed per 8 bit byte

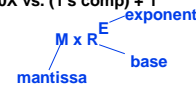
Integers:

- Sign & Magnitude: 0X vs. 1X
- 1's Complement: 0X vs. 1(-X)
- 2's Complement: 0X vs. (1's comp) + 1

Positive #'s same in all
First 2 have two zeros
Last one usually chosen

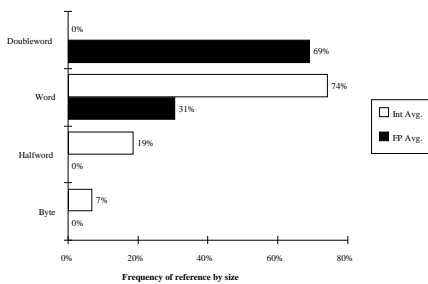
Floating Point:

- Single Precision
- Double Precision
- Extended Precision



How many +/- #'s?
Where is decimal pt?
How are +/- exponents represented?

Operand Size Usage

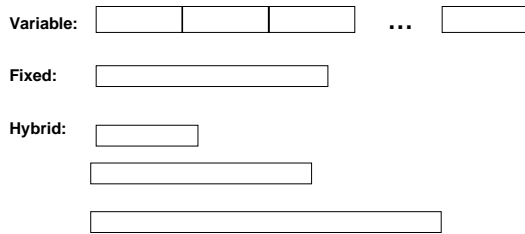


- Support these data sizes and types:
8-bit, 16-bit, 32-bit integers and
32-bit and 64-bit IEEE 754 floating point numbers

Instruction Format

- If have many memory operands per instructions and many addressing modes, need an Address Specifier per operand
- If have load-store machine with 1 address per instr. and one or two addressing modes, then just encode addressing mode in the opcode

Generic Examples of Instruction Formats



ECE4680 Lec 3 ISA-37

February 6, 2002

Summary of Instruction Formats

- If code size is most important, use variable length instructions
- If performance is most important, use fixed length instructions

ECE4680 Lec 3 ISA-38

February 6, 2002

Instruction Set Metrics

Design-time metrics:

- Can it be implemented, in how long, at what cost?
- Can it be programmed? Ease of compilation?

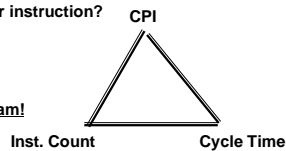
Static Metrics:

- How many bytes does the program occupy in memory?

Dynamic Metrics:

- How many instructions are executed?
- How many bytes does the processor fetch to execute the program?
- How many clocks are required per instruction?
- How "lean" a clock is practical?

Best Metric: Time to execute the program!



NOTE: this depends on instructions set, processor organization, and compilation techniques.

ECE4680 Lec 3 ISA-39

February 6, 2002

Lecture Summary: ISA

- Use general purpose registers with a load-store architecture;
- Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred;
- Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return;
- Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 64-bit IEEE 754 floating point numbers;
- Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size;
- Provide at least 16 general purpose registers plus separate floating-point registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist instruction set.

ECE4680 Lec 3 ISA-40

February 6, 2002