Evolution of Instruction Sets

Single Accumulator (EDSAC 1950)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

High-level Language Based
(B5000 1963)

Stack architectures
(B6500 1963, HP3000/70)

General Purpose Register Machines

Complex Instruction Sets
(Vax, Intel 432 1977-80)

Load/Store Architecture
(CDC 6600, Cray 1 1963-76)

RISC
(Mips,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

(1)

Basic CPU storage options

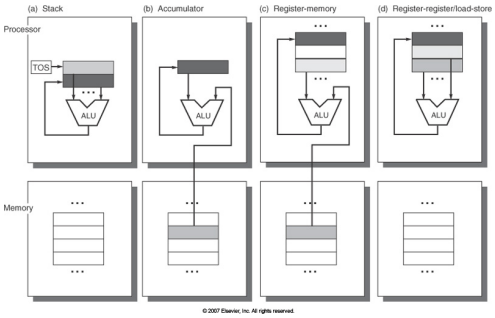
C = A + B in different storage schemes:

1. ***Accumulator***

```
load   A
add    B
Store  C
```

2. ***Stack***

```
push   A
push   B
add
pop    C
```



3. ***Memory-Memory***

```
add   C, A, B
```

4. ***Register-Memory***

```
load  R1, A
add   R2, R1, B
store R2, C
```

5. ***Register-Register***

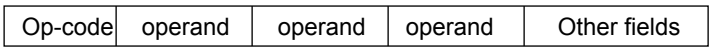
```
load  R1, A
load  R2, B
add   R3, R2, R1
store R3, C
```

What is the effect on: speed, memory traffic, encoding, program length?
What determines the number of registers?

(2)

- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems

- Issues in instruction set design:
  - operand storage in CPU (stack, registers, accumulator)
  - number of operands in an instruction (fixed or variable number)
  - type and size of operands (how is operand type determined)
  - addressing modes,
  - allowed operations and the size of op-codes,
  - size of each instruction.



(3)

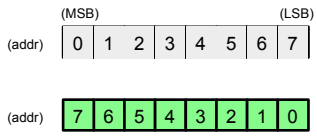
Memory addressing

- Most modern machines are *byte-addressable* -- yet most memory and cache traffic is in terms of words.

- A natural alignment problem (the start of a word or a double word).
  - Compiler is responsible
  - hardware does the checking

- How are bytes addressed within a word?
  - Big Endian -- byte 0 is the MSB (IBM, MIPS, SPARC)
  - Little Endian -- bye 0 is the LSB (vax, intel 80x86)

  Problem when we deal with serial communication and I/O devices.



(4)

Addressing Modes

1) Register
   operand = content of register = (R) — Add R1, R2
2) Immediate
   operand = in instruction = C — Add R1, #54
3) Register indirect
   operand = in memory = Mem[(R)]
   address = content of register — Add R1, (R2)
4) Displacement (or base)
   operand in memory = Mem[(R) + Base]
   address = content of register + base — Add R1, 54(R2)
5) Indexed
   operand in memory = Mem[(R) + (IR)]
   address = content of R+ content of IR — Add R1, (R2+R3)

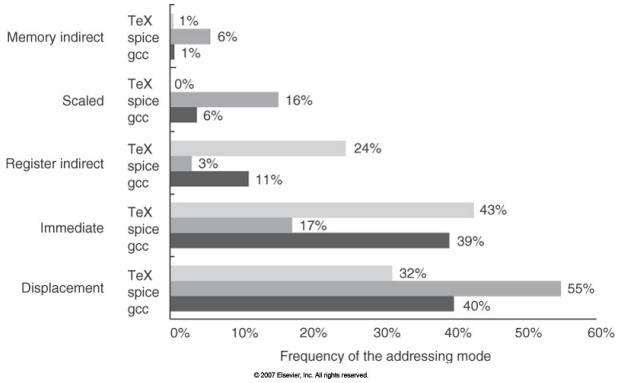
Note: (R) means content of R and Mem[A] means content of memory address A. (5)

6) Direct (absolute)
   operand in memory = Mem[C]
   address = a constant in the instruction — Add R1, (1000)
7) Memory indirect
   operand in memory = Mem[Mem[(R)]]
   address = the content of Mem[(R)] — Add R1, @(R2)
8) Auto-increment (or decrement)
   operand in memory = Mem[(R)]
   The content of R is incremented — Add R1, (R2)+
9) Scaled
   operand in memory = Mem[C+(R)+(IR)*d] — Add R1, 100(R2)(R3)

Which addressing mode is most suitable for:
local variables, stack operations, array operations, pointers, branch addresses, branch condition evaluation.

(6)

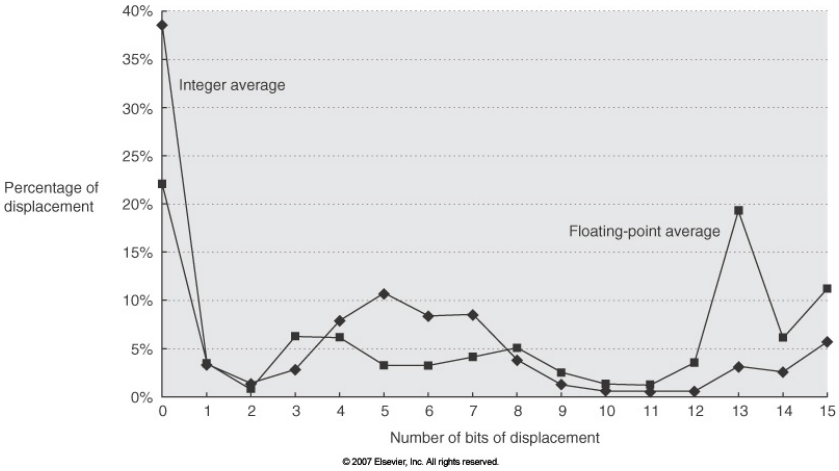
Popularity of the addressing mechanisms



- Can we use only a few (the most popular) addressing modes?
- Why would we want to use only a few addressing modes?
- The Immediate mode is mostly used for loads, compares and ALU operations.
- How many bits should we use in the Immediate and Displacements modes?
- DSP's have *modulo* addressing (for circular buffer management) and *bit-reverse* addressing (for FFTs)

(7)

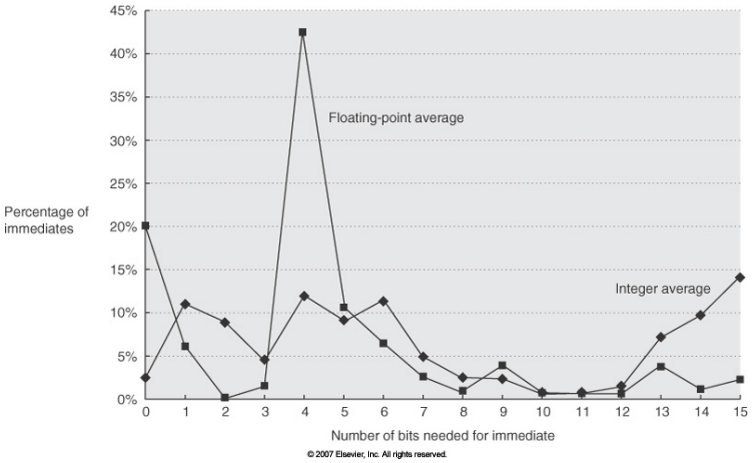
Displacement size



(8)

Immediate size

On Average, around 20% of instructions contain immediate operands



(9)

Operations

- **Arithmetic/logical**: add, sub, mult, div, shift (arith,logical), and, or, not, xor …
- **Data movement**: copy, move, load, store, ..
- **Control:** branch, jump, call, return, trap, …
- **System**: OS and memory management (ignore for now)
- Floating point:
- **Decimal**: legacy from COBOL
- **String**: move, copy, compare, search
- **Graphics**: pixel operations, compression, ...

- In Media and Signal processing, **partitioned** (or **paired**) operations are common (example: add the two half-words of a word).
- **Saturating arithmetic** avoids interruption for overflow or underflow conditions – useful for DSP's under real-time constrains.
- **Multiply-accumulate** operations are very useful for dot products in DSPs.

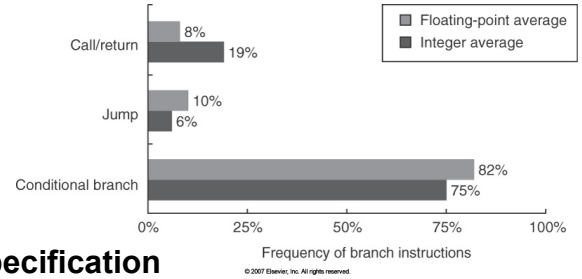
(10)

Frequent Operations

Rank	80x86 instruction	Integer average (% 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%

(11)

Branch instructions



Branch address specification

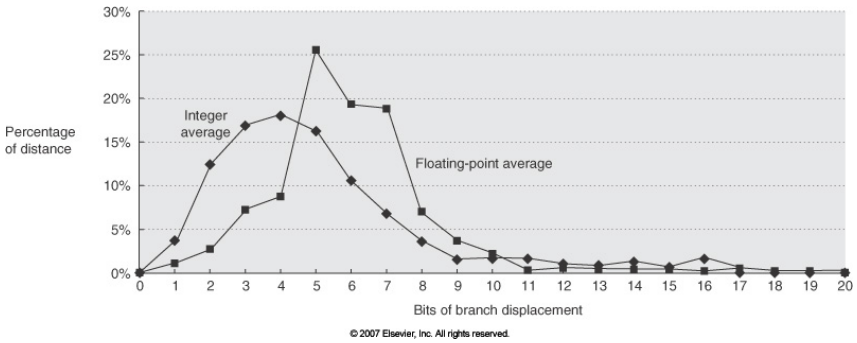
1) PC relative: -- makes the code position independent,
-- reduces the number of bits for target specification
-- target should be known at compile-time

2) put the target address in a register:
-- less restrictions on the range of branch address
-- useful for “switch” statements and function pointers
-- loaded at run-time (shared libraries and virtual functions)

(12)

Branch instructions

- Branch distance is usually not very large.
- Branch conditions are usually simple equality/inequality comparisons
- More than 80% of comparisons use immediate constants,
- A majority of comparisons is with “zero”,



(13)

Specification of branch conditions

- Use condition codes (flags usually set by hardware)
- Use condition registers
- compare and branch instructions
- Predicated instructions (operations guarded by a predicate)

C Source:

if (a < b) c++ else c+=1+b

Assume a -> r1
b -> r2
c -> r3

Unpredicated

```
    cmp r1,r2
    bge L0
    bra L1
L0: add r3,r2,r3
L1: add r3,1,r3
```

Predicated

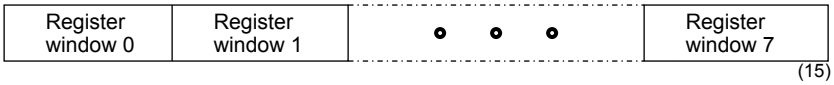
```
cmplt r1,r2,p1
add    r3,1,r3
add_p p1,r3,r2,r3
```

(14)

Procedure call and Return

- At a minimum the return address should be saved
  - Use branch and link instructions
  - Need to use a stack for nested calls
- Registers may have to be saved (by hardware or software)
- Registers can be saved by the caller or the callee,
- May mark some registers as “temporary”,
- Pass arguments in registers or on stack

- May use multiple register files
  - 24 of the 32 registers in the SPARC are in a register window and 8 are globals
  - The number of register windows depends on implementation
  - SAVE, RESTORE move windows forward or backward
  - On window overflow, save register on a stack
  - On window underflow, reload registers from stack



(15)

Encoding the instruction set

- Need to include op-code, operands, and maybe other fields
- Variable # of operands may call for variable instruction length
- Variable instruction length may reduce the code size,
- Fixed instruction length is easier to decode and faster to execute
- May use variable length op-code (why ?)
- How do you specify the addressing mode?

Examples:

- The VAX:
  – can have any number of operands, each may use any addressing modes,
  – Each operand uses a 4-bit specifier + 4-bit register address + one possible byte or word for displacement/immediate.

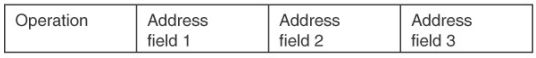
- RISC instructions use a fixed # of operands and specific addressing modes,

- Intel and IBM 360/370 use a hybrid approach (a few instruction lengths)
- IBM *Code_pack* keeps compressed programs in memory (good/bad??)

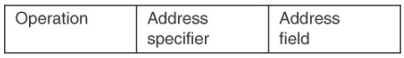
(16)

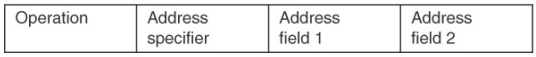
Encoding the instruction set

(a) Variable (e.g., Intel 80x86, VAX)



(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)





(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

(17)

Role of compilers

- Compilers are multi-phase: Front-end, high-level optimization, global optimization and code generation.
- The goals of a compiler are: correctness, speed of compiled code, speed of compilation, debugging support, …
- Compiler can do better optimization when instructions are simple
- Allocation of variables:
  - registers are used for temporaries, and possibly parameter passing
  - stacks are used for activation records and local variables
  - a global data area (may be bottom of stack) is used for globals
  - a heap is used for dynamically declared data

(18)

A "Typical" RISC

- Fixed format instruction
- General purpose registers -- some have overlapping register windows.
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement (no indirection)
- Simple branch conditions -- use PC relative mode for branching.
- Hardwired control (as opposed to micro-programmed control)
- Pipelined execution (one instruction issue every clock tick),
- Delayed Branches and pipeline stalls.

RISC II (Berkeley) had 39 instructions, 2 addressing modes and 3 data types,
Vax had 304 instructions, 16 addressing modes, 14 data types,

RISC II programs were 30% larger than Vax programs but 5 times as fast.
The RISC compiler were 9 times faster than the Vax compiler.

(19)

The MIPS architecture

- 32, 64-bit general purpose registers (register 0 is hardwired to “0”)
  - called R0, … , R31.
- 32, 64-bit floating point registers (each can hold a 32-bit single precision or a 64-bit double precision value)
  - called F0, F1, … , F31 (or F0, F2, … , F30).
- A special register for floating point status,
- Only immediate and displacement addressing modes (16-bit field)
- Byte addressable memories with 64-bit addresses.
- 32-bit instructions

Data transfer operations: LB, LBU, SB, SH, SW, SD, S.S, S.D, …

Arithmetic/logical operations: DADD, DADDI, DADDU, DADDIU, DSLL, DSLT, DSUB, …

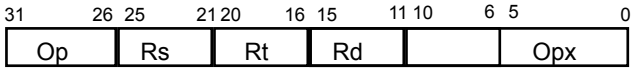
Control operations: BEQZ, BNE, J, JR, JAL, JALR, …

Floating point operations: ADD.S, ADD.D, ADD.PS, MULT.S, MADD.S, ...

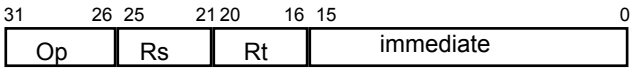
(20)

MIPS instruction format

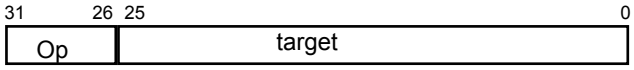
Register-Register (R-type) – used mainly for ALU operations



Register-Immediate (I-type) – used mainly for load/store and branch operations



Jump / Call (J-type)



(21)

The Intel 80x86 architecture

- 1971 - Intel 4004 (4-bit architecture)
- 1972 - Intel 8008 (8-bit architecture)
- 1974 - Intel 8080 (larger ISA, 16-bit address space, single accumulator, only 6 VLSI chips)
- 1974 - Intel 8086 (16-bit architecture, 16-bits dedicated registers)
- 1980 - Intel 8087 (floating point co-processor)
- 1982 - Intel 80286 (24-bit address space but has a compatible mode)
- 1985 - Intel 80386 (32-bit architecture and address space, 32 GPRs, paging and segmentation hardware),
- 1989 - Intel 80486
- 1992 - Intel Pentium
- 1996 - Pentium 2 (233-366 MHz, 512 KB L2 cache)
- 1999 - Pentium 3 (100-133 MHz, 512KB L2 cache)
- 2000 - Pentium 4 (1.3-3.6 GHz, 256KB – 2MB L2 cache)
- 2005 - Pentium D (2.66-3.73 GHz, 2–4 MB L2 cache)
- 2007 - Pentium dual core (1.6-2.7 GHz, 1-2MB L2 cache)
- 2010 - Nahalem (up to 3GHz, up to 8 cores, up to 30MB L3)

(22)

The Intel original 80x86 architecture

- Internal registers but mostly for dedicated uses.
- 16-bit architecture, but can get 20-bit address using segmentation
- addressing modes:
  - absolute
  - register indirect (BX, SI, DI in 16-bit modes, extended registers in 32-bit mode)
  - base mode (BX, SI, DI, SI + displacement which is 8, 16 bits , or 8, 16, 32 bits)
  - indexed BX+SI, BX+DI, BP+SI, BP+DI
  - based indexed (indexed+ 8 or 16 bit displacement)
  - based plus scaled indexed (on 386, scale = 0,1,2,3 , restrictions on register use is removed)
  - based with scaled index and displacement.

- Op-code byte usually indicates the operand type and the addressing mode. Some instructions use a *postbyte* which contains addressing mode information.

(23)