Instruction Set Architecture (ISA) Design

Overview
- Classify Instruction set architectures
- Look at how applications use ISAs
- Examine a modern RISC ISA (DLX)
- Measurement of ISA usage in real computers

Classification Categories
- How are operands stored in CPU?
  - accumulator-based
  - stack-based
  - register-set based
- Memory addressing
- Branch architecture
- Instruction encoding
Accumulator Architectures

- Single register A
- One explicit operand per instruction
  - two 'working' instruction types: op and store
    - A ← A op M
    - A ← A op *M
    - *M ← A
  - two addressing modes
    - direct addressing, *M
    - immediate, M
- Attributes:
  - short instructions possible
  - minimal internal state; simple internal design
  - high memory traffic; many loads and stores
- The most popular early architecture: IBM 7090, DEC PDP-8, MOS 6502

Stack Architectures

- No registers
- No explicit operands in ALU instructions; one in push/pop
- A = B + C * D
  - push b
  - push c
  - push d
  - mul
  - add
  - pop a
- Advantages:
  - short instructions possible: implicit stack address references
  - Compiler is easy to write
Stack Architectures (2)

- Disadvantages
  - Code is inefficient
    - fix by: random access to stacked values
  - Stack is a bottleneck
    - fix by: with a stack-cache
- Promoted in the 60’s; limited popularity:
  - Burroughs B5500/6500, HP 3000/70, Java VM

Register-Set based Architectures

- “General Purpose Registers” = GPRs
- Registers are like an explicitly-managed cache for holding recently used values
- The dominant architecture: CDC 6600, IBM 360/370 (RX), PDP-11, 68000, all RISC machines, etc.
- Advantages:
  - Allows fast access to temporary values
  - Permits clever compiler optimization
  - Reduced traffic to memory
- Disadvantages:
  - Longer instructions (than accumulator designs)
- GPR classification
  - Number operands in ALU instruction
  - Number of operands in memory
Register-to-Register

- No memory addresses (load/store architecture)
  - typically 3-operand ALU ops
    - Bigger encoding, but simplifies register allocation
  - Simple fixed-length instructions
  - easily pipelined
  - higher instruction count
  - Examples: CDC6600, CRAY-1, most RISCs

\[
C = A + B
\]

| LOAD  | R1 <- A |
| LOAD  | R2 <- B |
| ADD   | R3 <- R1 + R2 |
| STORE | C <- R3 |

Register-to-Memory

- One memory address
  - typically 2-operand ALU ops
  - instruction length and/or cycle time varies
  - result destroys an operand
  - harder to pipeline
  - Examples: IBM 360/370

\[
C = A + B
\]

| LOAD  | R1 <- A |
| ADD   | R1 <- R1 + B |
| STORE | C <- R1 |
Memory-to-Memory

- Three memory addresses,
  - No register wastage
  - Large variation in work/instruction

\[ C = A + B \]

Measurements to support ISA design

- Given alternatives, how can we decide which is best?
- Measure real applications/compilers
- Ten SPEC92 Benchmarks for load/store machine measurements
  - Integer: espresso, li, eqntott, compress, gcc
  - Floating point: doduc, mdjdp2, ear, su2cor, hydro2d
- Three SPEC89 Benchmarks for VAX measurements
  - Tex, Spice2g6, gcc
- Measurements may vary
  - Dependent on specific benchmark, ISA and compiler
  - Design of ISAs needs a wide class of benchmarks including OS
  - Special purpose applications may have different requirements (eg. embedded control, DSP, graphics)
  - We can still draw conclusions about general purpose apps
Program Usage of Addressing Modes

double x[100];
void foo(int a) {
    int j;
    for (j=0; j<10; j++)
        x[j] = 3 + a*x[j-1];
    bar(a);
}

Addressing Modes

- Addressing modes ultimately specify a constant, a register, or a location in memory
  - Register R1
  - Immediate #123
  - Direct (12345)
  - Register indirect (R1)
  - Displacement 100(R1)
  - Indexed (R1+R2) or 12345(R1)
  - Scaled 100(R2+4*R3)
  - Memory indirect @(R2) or @(any mode)
  - Auto-increment (R2)+
  - Auto-decrement -(R2)

- PC-relative forms (or PC may be a general register, like VAX R15)

- Complicated modes reduce instruction count at the cost of complex implementations
What Memory addressing modes are most common?

- Measured on the VAX
- Register operands account for 51% of all references
- Displacement-style mode is most common

How big are address displacements

- 12 bits captures 75%
- 16 bits captures 99%
How common are immediates?

Percentage of operations that use immediates

<table>
<thead>
<tr>
<th>Loads</th>
<th>Compares</th>
<th>ALU ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>20%</td>
<td>80%</td>
<td>40%</td>
</tr>
</tbody>
</table>

- TeX
- Spice
- GCC

How big are immediate fields?

Bits needed for the immediate value

- TeX
- Spice
- GCC
What size data is accessed?

Original DEC Alpha AXP had no byte or halfword data access instructions; saves having an alignment network in a critical data path. A good choice?

Control flow operations

- Four types of control flow instructions:
  - Conditional branches: 84%
  - Unconditional jumps: 4%
  - Procedure calls/returns: 12%

- Ways to specify the destination address:
  - PC relative
    - uses fewer instruction bits
    - position independent
  - Any of the addressing modes for data instructions
    - Procedure return requires some kind of register-indirect
Conditional Branch Instructions (1)

- Condition code
  - record ALU status from each op
    - Z, N, C, O
  - single copy, modified by most instructions
  - special small "register" is set by all operations
  - sometimes set free, as a side effect of necessary instruction
  - sometimes set when unwanted!
  - compare and branch must be kept together
  - single flag register can be a serial bottleneck
  - Examples: 360/370, 80x86, VAX, SPARC, PowerPC

Conditional Branch Instructions (2)

- Condition in general registers
  - record ALU status from compare in a register
    - R1 ← COMP (R2, R3)
  - alternatively use condition registers
  - test the register later for branch
    - BGE R1, LOOP
  - allows separation of test and branch
  - enables parallel execution
    - R1 ← COMP (R2, R3)
    - R4 ← COMP (R5, R3)
    - BGE R1, DST1
    - BGE R4, DST2
  - may require additional instruction; "uses up" a register
  - Examples: MIPS, DEC Alpha

- Preceding integer compares
  - Most are EQ or NE (86%) vs. GT/LE(7%) or LT/GE (7%)
  - Most are immediate compares (87%) of which most are
    compares to zero (75%)
Conditional Branch Instructions (3)

- Compare-and-branch
  - Combination compare/branch: no explicit condition encoding
    - \texttt{BGE R1, R2, LOOP}
  - saves an instruction but requires a lot of work
  - Examples: VAX, MIPS

How far do branches go?

![Graph showing the number of bits needed to specify distance between target and branch]

- Integer
- Flt. pt.
The role of compiler technology

- Most programming is now done in a high-level language
- Increased role of the compiler in determining performance
- Correct and efficient compilers are hard to write, so design ISAs that make it as easy as possible to write an optimizing compiler

- Optimizations:
  - High level: source code manipulation (eg: inline procedures)
  - Local: within straight-line code (eg: common subexpression elimination, constant propagation, strength reduction)
  - Global: across branches (eg: copy propagation, loop code motion, register allocation)
  - Machine-dependent (eg: pipeline scheduling)

A simple loop

```c
int A[100], B[100], C;

main ()
{
    int i;
    c=10;
    for (i=0; i<100; i++)
        A[i] = B[i] + C;
}
```
Unoptimized code

C=10;
    li r14, 10
    sw r14, C
for (i=0; i<100; i++)
    sw r0, 4(sp)
$33:  
    A[i] = B[i] + C
    lw r14, 4(sp)
    mul r15, r14, 4
    lw r24, B(r15)
    lw r25, C
    addu r6, r24, r25
    lw r16, 4(sp)
    mul r17, r16, 4
    sw r8, A(r17)
    lw r9, 4(sp)
    addu r10, r9, 1
    sw r10, 4(sp)
    blt r10, 100, $33
j  $31

12 instructions per iteration

Optimized code

C=10;
    li r14, 10
    sw r14, C
for (i=0; i<100; i++)
    la r3, A
    la r4, B
    la r6, B+400
$33:  
    A[i] = B[i] + C
    lw r14, 4(sp)
    addu r15, r14, 10
    sw r15, 0(r3)
    addu r3, r3, 4
    addu r4, r4, 4
    bltu r4, r6, $33
j  $31

6 instructions per iteration
4 fewer loads due to code motion, register allocation, constant propagation
2 fewer multiplies due to induction variable elimination, strength reduction

Can you do better by hand?
What is the effect of optimization on the instruction mix?

- Decreases:
  - Total instruction count
  - number of memory references
  - number of ALU operations
- Does NOT generally decrease:
  - number of branch operations, which are thus increased as a percentage of all instructions

Good ISA features for an optimizing compiler

- Regularity
  - Make operations, data types, and addressing modes all orthogonal
- Provide primitives, not complex solutions
  - Do not match high-level language features directly -- it's been tried many times before and generally doesn't work!
- Simplify tradeoffs
  - Provide one efficient mechanism
- Allow constants to be used everywhere
  - Allows constant propagation
- Provide enough registers
  - Enables sophisticated register allocation schemes
- Provide a way to communicate hints
  - about memory hierarchy
  - about branch guesses