Instruction Set Architecture and Principles

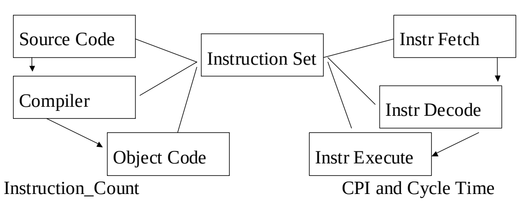
Chapter 2

Instruction Sets

- What's an instruction set?
  - – Set of all instructions understood by the CPU
  - – Each instruction directly executed in hardware
- Instruction Set Representation
  - – Sequence of bits, typically 1-4 words
  - – May be variable length or fixed length
  - – Some bits represent the op code, others represent the operand

Instruction Set Affects CPU Performance

- Recall
  - – ExecTime = Instruction_Count * CPI * Cycle_Time
  - – Instruction Set is at the heart of the matter!



Classes of Instruction Set Architectures

- Stack Based
  - – Implicitly use the top of a stack
    - PUSH X, PUSH Y, ADD, POP Z
    - Z = X + Y
- Accumulator Based
  - – Implicitly use an accumulator
    - LOAD X, ADD Y, STORE Z
- GPR – General Purpose Registers
  - – Operands are explicit and may be memory or registers

• LOAD R1, X	or	LOAD R1, X
• LOAD R2, Y		ADD R1, Y
• ADD R1, R2, R3		STORE R1, Z
• STORE R1, Z		

Comments on Classifications of ISA

- Stack-based generates short instructions (one operand) but programming may be complex, lots of stack overhead
- Accumulator-based also has complexities for juggling different data values
- GPR – most common today
  - Allows large number of registers to exist to hold variables
  - Compiler gets the job today of allocating variables to registers

Comparison Details

	PRO	CON
STACK	Simple Address format Effective decode Short Instruction → high code density	Stack bottleneck Lack of random access Many instr's needed for some code
ACC	Short Instrucions → High code density	Lots of memory traffic
GPR	Lots of code generation options Efficiencies possible	Larger code size Possibly complex effective address calculations

How Many Operands?

- Two or Three?
  - Two : Source and Result
  - Three: Source 1, Source 2, and Result
- Tradeoffs
  - Two operand ISA requires more temporary instructions (e.g. Z = X + Y can't be done in one instruction)
  - Three operand ISA supports fewer instructions but increases the instruction complexity and size
- Also must consider the types of operands allowed
  - Register to Register, Register to Memory, Memory to Memory
  - Instruction density, memory bottlenecks, CPI variations!

Register-Register (0,3)

- (m, n) means m memory operands and n total operands in an ALU instruction
  - Pure RISC, register to register operations
  - Advantages
    - Simple, fixed length instruction encodings
    - Decode is simple
    - Uniform CPI
  - Disadvantages
    - Higher Instruction Count
    - Some instructions are short and bit encodings may be wasteful

Register-Memory (1,2)

- Register – Memory ALU Architecture
- In later evolutions of RISC and CISC
- Advantages
  - Data can be accessed without loading first
  - Instruction format easy to encode
  - Good instruction density
- Disadvantages
  - Source operand also destination, data overwritten
  - Need for memory address field may limit # registers
  - CPI varies by operand location

Memory-Memory (3,3)

- True memory-memory ALU model, e.g. full orthogonal CISC architecture
- Advantages
  - Most compact instruction density, no temporary registers needed
- Disadvantages
  - Memory access create bottleneck
  - Variable CPI
  - Large variation in instruction size
  - Expensive to implement
- Not used in today's architectures

Memory Addressing

- What is accessed - byte, word, multiple words?
  - today's machine are byte addressable, due to legacy issues
- But main memory is organized in 32 - 64 byte lines
  - matches cache model
  - Retrieve data in, say, 4 byte chunks
- Alignment Problem
  - accessing data that is not aligned on one of these boundaries will require multiple references
    - E.g. fetching 16 bit integer at byte offset 3 requires two four byte chunks to be read in (line 0, line 1)
  - Can make it tricky to accurately predict execution time with mis-aligned data
  - Compiler should try to align! Some instructions auto-align too

Big-Endian vs. Little-Endian

- How is data stored?
  - E.g. given 0xABCD
- Big-Endian
  - Store MSByte first (AB CD)
  - Hex dumps a little easier to read
  - Intel
- Little-Endian
  - Store LSByte first (CD AB)
  - May still get right value when reading different word sizes
  - Motorola
- Computers internally know how data is stored, so does it matter?
  - Yes, networks and sending data byte by byte
  - May need functions, like htons
  - Some systems have an Endian control bit to select

Addressing Modes

- The addressing mode specifies the address of an operand we want to access
  - Register or Location in Memory
  - The actual memory address we access is called the **effective address**
- Effective address may go to memory or a register array
  - typically dependent on location in the instruction field
  - multiple fields may combine to form a memory address
  - register addresses are usually simple - needs to be fast
- Effective address generation is important and should be fast!
  - Falls into the common case of frequently executed instructions

Memory Addressing

Mode	Example	Meaning	When used
Register	Add R4, R3	Regs[R4]←Regs[R4]+ Regs[R3]	Value is in a register
Immediate	Add R4, #3	Regs[R4] ← Regs[R4] + 3	For constants
Displacement	Add R4, 100(R1)	Regs[R4] ← Regs[R4] + Mem[100+Regs[R1]]	Access local variables
Indirect	Add R4, (R1)	Regs[R4]←Regs[R4]+ Mem[Regs[R1]]	Pointers
Indexed	Add R3, (R1+R2)	Regs[R3]←Mem[Regs [R1]+Regs[R2]]	Traverse an array
Direct	Add R1, $1001	Regs[R1] ← Regs[R1] + Mem[1001]	Static data, address constant may be large

Memory Addressing

Mode	Example	Meaning	When used
Memory Indirect	Add R1, @(R3)	Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]	*p if R3=p
Autoinc	Add R1, (R2)+	Regs[R1]←Regs[R1]+ Mem[Regs[R2]], Regs[R2]←Regs[R2]+1	Stepping through arrays in a loop
Autodec	Add R1, (R2)-	Regs[R1]←Regs[R1]+ Mem[Regs[R2]], Regs[R2]←Regs[R2]-1	Same as above. Can push/pop for a stack
Scaled	Add R1, 100(R2)[R3]	Regs[R1]← Regs[R1]+ Mem[100+Regs[R2] + Regs[R3] * d]	Index arrays by a scaling factor, e.g. word offsets

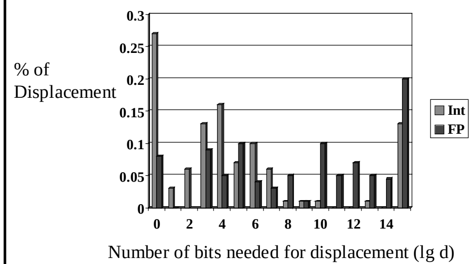
Which modes are used?

- VAX supported all modes!
- Dynamic traces, frequency of modes collected
  - Memory Indirect
    - TeX: 1%, SPICE: 6%, gcc: 1%
  - Scaled
    - TeX: 0%, SPICE: 16%, gcc: 6%
  - Register Deferred
    - TeX: 24%, SPICE: 3%, gcc: 11%
  - Immediate
    - TeX: 43%, SPICE: 17%, gcc: 39%
  - Displacement
    - TeX: 32%, SPICE: 55%, gcc: 40%
- Results say: support displacement, immediate, fast!
- WARNING! Role of the compiler here?

Displacement Addressing Mode

- According to data, this is a common case – so optimize for it
- Major question – size of displacement field?
  - If small, may fit into word size, better instruction density
  - If large, allows larger range of accessed data
  - To resolve, use dynamic traces once again to see the size of displacement actually used

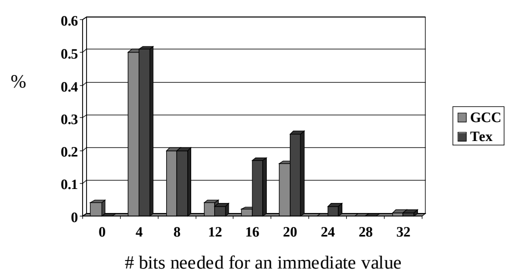
Displacement Traces



Immediate Addressing Mode

- Similar issues as with displacement; how big should the operands be? What size data do we use in practice?
- Tends to be used with constants
  - Constants tend to be small?
- What instructions use immediate addressing?
  - Loads: 10% Int, 45% FP
  - Compares: 87% Int, 77% FP
  - ALU: 58% Int, 78% FP
  - All Instructions: 35% Int, 10% FP

Immediate Addressing Mode



Instruction Set Optimization

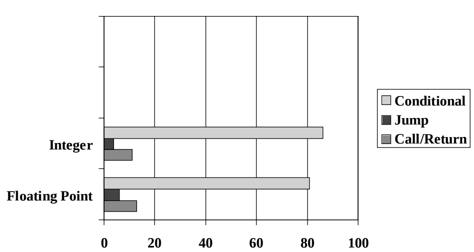
- See what instructions are executed most frequently, make sure they are fast!
- Intel x86:

– Load	22%
– Conditional Branch	20%
– Compare	16%
– Store	12%
– Add	8%
– AND	6%
– SUB	5%
– Move Reg To Reg	4%
– Call	1%
– Ret	1%

Control Flow

- Transfers or instructions that change the flow of control
- Jump
  - unconditional branch
  - How is target specified? How far away from PC?
- Branch
  - when condition is used
  - How is condition set?
- Calls
  - Where is return address stored?
  - How are parameters passed?
- Returns
  - How is the result returned?
- What work does the linker have to do?

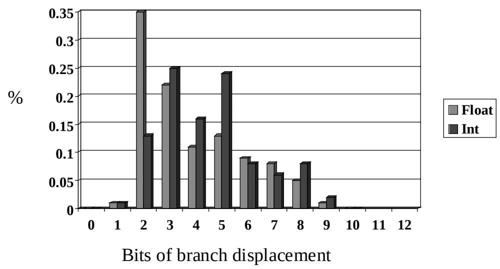
Biggest Deal is Conditional Branch



Branch Address Specification

- Effective address of the branch target is known at compile time for both conditional and unconditional branches
  - as a register containing the target address
  - as a PC- relative offset
- Consider word length addresses, registers, and instructions
  - full address desired? Then pick the register option.
  - BUT - setup and effective address will take longer.
  - if you can deal with smaller offset then PC relative works
  - PC relative is also position independent - so simple linker duty, preferred when possible
- Do more measurements to see what's possible!

Branch Distances



Condition Testing Options

Name	Test	Pro	Con
Condition Code	Special PSW bits set by ALU	Conditions may be set for "free"	Extra state to maintain, constrain ordering of instructions
Condition Register	Comparison result put in register, test register	Simple, less ordering constraints	Uses up a register
Compare and Branch	Compare is part of branch	One instruction instead of two for a branch	May be too much work per instruction

What is compared?

- < , >=
  - Int: 7% Float: 40%
- >, <=
  - Int: 7% Float: 23%
- ==, !=
  - Int: 86% Float: 37%

- Over 50% of integer compares were to test for equality with 0

Branch Direction

- GCC
  - Backward branches: 24%
  - Branches taken: 54%
- Spice
  - Backward branches: 31%
  - Branches taken: 51%
- TeX
  - Backward branches: 17%
  - Branches taken: 54%
- Most backward branches are loops
  - taken about 90%
- Branch statistics are both compiler and application dependent
- Loop optimizations may have large effect

- We'll see the role of this later with branch prediction and pipelining

Operand Type and Size

- Operands may have many types, how to distinguish which?
- Annotate with a tag interpreted by hardware
  - Not used anymore today
- The opcode also encodes the type of the operand
  - Amounts to different instructions per type
- Typical types
  - character – byte (UNICODE?)
  - short integer – two bytes, 2's complement
  - integer - one word, 2's complement
  - float - one word - IEEE 754
  - double - two words - IEEE 754
  - BCD or packed decimal

Most Frequently Used Operand Types

- Double word
  - TeX 0%, Spice 66%, GCC 0%
- Word
  - TeX 89%, Spice 34%, GCC 91%
- Halfword
  - TeX 0%, Spice 0%, GCC 4%
- Byte
  - TeX 11%, Spice 0%, GCC 5%
- Move underway now to 64 bit machines
  - BCD likely to go away
  - larger offsets and immediates is likely
  - usage of 64 and 128 bit values will increase

Encoding the Instruction Set

- How to actually store, in binary, the instructions we want
  - Depends on previous discussion, operands, addressing modes, number of registers, etc.
  - Will affect code size, and CPI
- Tradeoffs:
  - Desire to have many registers, many addressing modes
  - Desire to have the average instruction size small
  - Desire to encode into lengths the hardware can easily and efficiently handle
    - Fixed or variable length?
    - Remember data delivered in blocks of cache line sizes

Instruction Set Encoding Options

Variable (e.g. VAX)

OpCode and # of ops	Operand 1	Operand 2	…	Operand N

Fixed (e.g. DLX, SPARC, PowerPC)

OpCode	Operand 1	Operand 2	Operand 3

Hybrid (e.g. x86, IBM 360)

OpCode	Operand 1	Operand 2	Operand 3
OpCode	Operand 1	Operand 2	
OpCode			

Instruction Size? Complexity?

Role of the Compiler

- Role of the compiler is critical
  - Difficult to program in assembly, so nobody does it
  - Certain ISA's make assembly even more difficult to optimize
  - Leave it to the compiler
- Compiler writer's primary goal:
  - correctness
- Secondary goal:
  - speed of the object code
- More minor goals:
  - speed of the compilation
  - debug support
  - Language interoperability

Compiler Optimizations

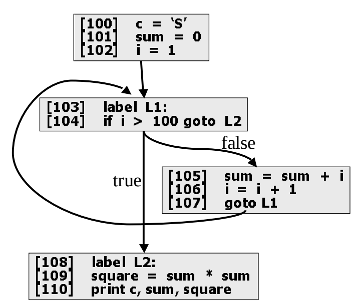
- High-Level
  - Done on source with output fed to later passes
  - E.g. procedure call changed to inline
- Local
  - Optimize code only within a basic block (sequential fragment of code)
  - E.g. common subexpressions – remember value, replace with single copy. Replace variables with constants where possible, minimize boolean expressions
- Global
  - Extend local optimizations across branches, optimize loops
  - E.g., remove code from loops that compute same value on each pass and put it before the loop. Simplify array address calculations.

Compiler Optimizations (cont)

- Register Allocation
  - What registers should be allocated to what variables?
  - NP Complete problem using graph coloring. Must use an approximation algorithm
- Machine-dependent Optimization
  - Use SIMD instructions if available
  - Replace multiply with shift and add sequence
  - Reorder instructions to minimize pipeline stalls

Example of Register Allocation

```
c = 'S' ;
sum = 0 ;
i = 1 ;
while ( i <= 100 ) {
   sum =  sum + i ;
   i = i + 1 ;
}
square = sum * sum;
print c, sum, square;
```



Example : Register Allocation

- Assume only two registers available, R1 and R2. What variables should be assigned, if any?

Variable	Register
c	?
sum	?
i	?
square	?

```
c = 'S' ;
sum = 0 ;
i = 1 ;
while ( i <= 100 ) {
    sum =  sum + i ;
    i = i + 1 ;
}
square = sum * sum;
print c, sum, square;
```

Example : Register Allocation

- Sum and I should get priority over variable C
- Reuse R2 for variables I and square since there is no point in the program where both variables are **simultaneously live.**

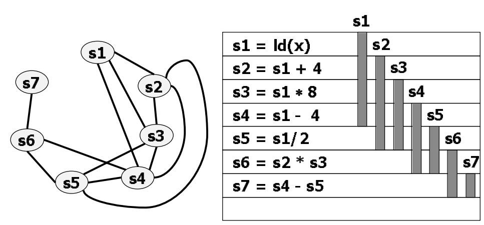
Variable	#Uses	Register
c	1	none
sum	103	R1
i	301	R2
square	1	R2

```
c = 'S' ;
sum = 0 ;
i = 1 ;
while ( i <= 100 ) {
    sum =  sum + i ;
    i = i + 1 ;
}
square = sum * sum;
print c, sum, square;
```

Register Allocation: Constructing a Graph

- A node is a variable (may be temporary) that is a candidate for register allocation
- An edge connects two nodes, v1 and v2, if there is some statement in the program where variables v1 and v2 are simultaneously live, meaning they would interfere with one another
- Once this graph is constructed, we try to color it with k colors, where k = number of free registers. Coloring means no connecting nodes may be the same color. The coloring property ensures that no two variables that interfere with each other are assigned the same register.

Register Allocation Example



What is a valid coloring?
Can we use the same register for s4 that we use for s1?

Impact of Compiler Technology can be Large

Optimization	% Faster
Procedure Integration	10%
Local Optimizations Only	5%
Local + Register Allocation	26%
Local + Global + Register	63%
Everything	81%

Stanford UCode Compiler Optimization on Fortran/Pascal Programs
Clear benefit to compiler technology and optimizations!

Compiler Take-Aways

- ISA should have at least 16 general-purpose registers
  - Use for register allocation, simplifies graph coloring
- Orthogonality (all addressing modes for all operations) simplifies code generation
- Provide primitives, not solutions
  - E.g., a solution to match a language construct may only work with one language (see 2.9)
  - Primitives can be optimized to create a solution
- Bind as many values as possible at compile-time, not run-time