ISA

Instruction Set Architecture

- ❑ What the heck is an instruction set anyway?
  - Just a set of instructions
  - Each instruction is *directly* executed by the CPU's hardware
- ❑ How is it represented?
  - By a binary format since the hardware only *grok's* bits
  - Typical physical blobs are bits, bytes, words, n-words
  - Word size is typically 16, 32, 64 bits today
  - Options - fixed or variable length formats
  - Fixed - each instruction encoded in same size field - typically 1 word
  - Variable - half-word, whole-word, multiple word instructions are possible

A Note on Measurements

- ❑ We're taking the quantitative approach
- ❑ BUT measurements will vary:
  - due to application selection or application mix
  - due to the particular compiler being used
  - also dependent on compiler optimization selection
- ❑ And the target ISA
- ❑ Hence the measurements we'll talk about
  - are useful to understand the method
  - are a typical yet small sample derived from benchmark codes
- ❑ To do it for real
  - you would want lots of real applications
  - Plus - your compiler and ISA

Instruction Characteristics

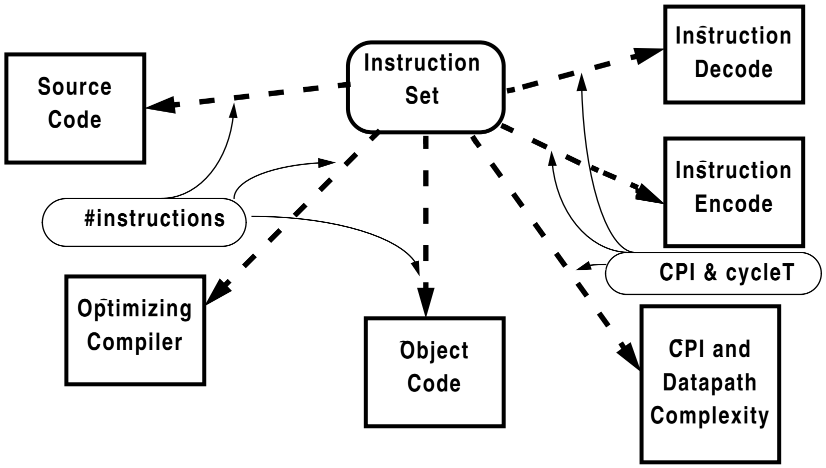
- ❑ Usually a simple operation
  - Which operation is identified by the *op-code* field
- ❑ But operations require operands - 0, 1, or 2
  - To identify where they are, they must be addressed
    - Address is to some piece of storage
    - Typical storage possibilities are main memory, registers, or a stack
    - Each has it's own particular organization
- ❑ 2 options explicit or implicit addressing
  - Implicit - the op-code implies the address of the operands
    - ADD on a stack machine - Pops the top 2 elements of the stack, then pushes the result
    - HP calculators work this way
    - Can you spot the pro's and con's of this model?
  - Explicit - the address is specified in some field of the instruction
    - Note the potential for 3 addresses - 2 operands + the destination
    - What are the advantages of addressing registers vs. memory?

What Operations are Needed

- ❑ Arithmetic + Logical
  - ADD, SUB, MULT, DIV, SHIFT - logical and arith, AND, OR, XOR, NOT
- ❑ Data Transfer - copy, load, store
- ❑ Control - branch, jump, call, return, trap
- ❑ System - OS and memory management
  - We'll ignore these for now - but remember they are needed
- ❑ Floating Point
  - Same as arithmetic but usually take bigger operands
- ❑ Decimal - if you go for it what else do you need?
  - Legacy from COBOL and the commercial application domain
- ❑ String - move, compare, search

Instruction Set Design

The problem: XEQ_t = #Instructions x CPI x cycle-time
Iset influences everything!



Classifying Instruction Set Architectures

Based on CPU internal storage options
AND operand-arity

Operand Storage in CPU	Where are they other than memory
Number of explicit operands named per instruction	How many? Min, Max - maybe even average
Addressing Modes	How is the effective address for an operand calculated? Can all operands use any mode?
Operations	What are the options for the opcode?
Type and size of operands	How is typing done? How is the size specified

These choices critically affect - #instructions, CPI, and cycle time

Basic CPU Storage Options

Sadly only one survived - too bad compilers are hard

Storage Type	Examples	Explicit operands per ALU instruc.	Result Destination	Explicit operand access method
Stack	B5500, B6500 HP2116B HP 3000/70	0	Stack	Push & Pop Stack
Accumulator	PDP-8 Motorola 6809 + ancient ones	1	Accumulator	Acc = Acc + mem
Register Set	IBM 360 DEC VAX + all modern micro's	2 or 3	Registers or Memory	Rx = Ry + mem (3) Rx = Rx + Ry (2) Rx = Rx + Rz (3)

register-register, register-memory, and memory-memory (gone) options

C = A + B

- Stack

```
Push A
Push B
ADD
Pop C
```

- Accumulator

```
Load A
Add B
Store C
```

- Register

```
Load R1, A
Add R1, B
Store C, R1
```

- Consider - instruction encoding?
- Consider what a search or sort would look like

Form and Function are always Related

Consider the class pro's and con's

Machine T ype	Advantages	Disadvantages
Stack	Simple effective address Short instructions Good code density Simple I-decode	Lack of random access. Efficient code is difficult to generate. Stack is often a bottleneck.
Accumulator	Minimal internal state Fast context switch Short instructions Simple I-decode	Very high memory traffic
Register	Lots of code generation options. Efficient code since compiler has numerous useful options.	Longer instructions. Possibly complex effective address generation. Size and structure of register set has many options.

Modern Choice - GPR Machines

Why?

- To some extent - it's IBM disease
  - Dominant manufacturer in this part of the wars
  - Compiler expertise momentum followed this track
  - Consider RISC vs. Stack option w.r.t datapath issues
  - Consider a cached stack vs. a modern GPR machine w.r.t. cache locality issues
- Still it was probably the right choice - sigh!
  - Software has considerably more momentum than hardware
  - Software exists for decades
  - Hardware is pitched every 2 to 4 years
- Compiler technology is important - today it's well advanced for GPR - in particular RISC

Comparison is Instructive

- Look at fundamental differences in datapath and control-path
- Assumptions - datapath
  - Only direct addressing
  - 8 bit opcode for all (not true but difference is minor)
  - 16 registers for GPR machine
  - 16-bit memory address field
  - Ignore byte and half-word accesses for simplicity
  - Pick a 32-bit data-word for yucks
  - T ri-state bus style rather than mux and demux style
- Assumptions - control (ignore condix-branch)
  - Micro-code like descriptions
  - FSM control is the only control style used today

Instruction Formats

Accumulator - 24 bit instruction

8-bit OpCode	16-bit Memory Address

Stack - also 24 bit but most instructions will be 8-bit (pack-em)

8-bit OpCode	16-bit Memory Address

Register - 2 explicit operands (3 explicit is obvious) - 28 bits

8-bit OpCode	4-bit Reg-addr	16-bit Memory Address

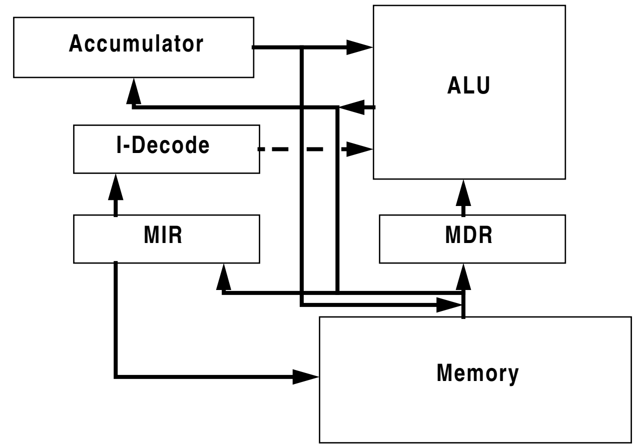
- Could pack but instruction word allignment would be a problem.

Some Datapath Abbreviations

- IR - instruction register
- MAR - memory address register
- MDR - memory data register
- ALU - Arithmetic & Logical Unit

Accumulator Datapath

No wonder they did this initially



Accumulator Control

Loads

- Read Memory, Enable Memory to Accumulator
- Load Accumulator

Stores

- Enable Accumulator to MBUS
- Write Memory

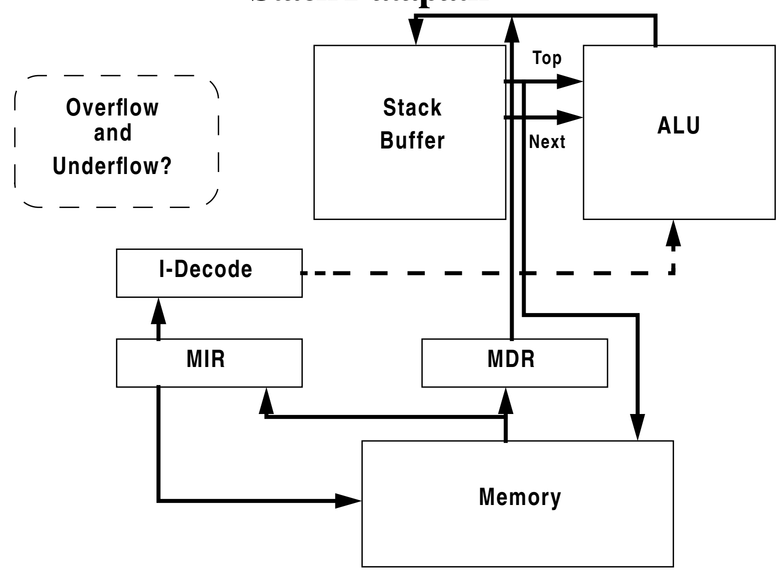
ALU Op's

- Read Memory
- Enable ALU to Accumulator
- Load Accumulator

Branch - just like an IFetch but with PC as address source

- Read Memory
- Enable Memory to MIR
- Load MIR

Stack Datapath



Stack Control (over -simplified)

Loads

- Read Memory
- Push

Stores

- Enable Top to MBUS, Write Memory
- Pop

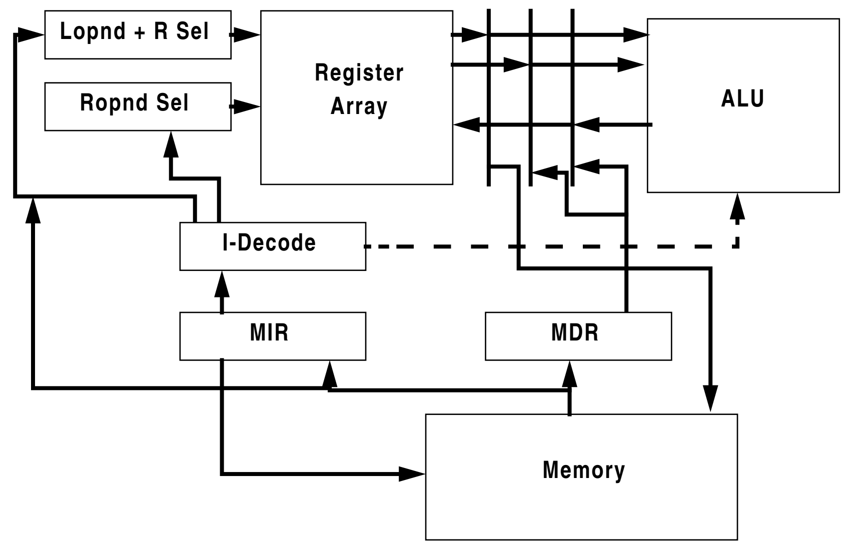
ALU Op's

- Load Top or Next
- Pop or not

Branch - just like an IFetch but with PC as address source

- Read Memory
- Enable Memory to MIR
- Load MIR

Simple GPR Datapath



GPR Control

- ❑ Loads - just like accumulator but select Reg.
- ❑ Stores - just like accumulator but select Reg.
- ❑ Branch - same as all the rest
- ❑ ALU OPs - whoa!
  - Select Left operand and result register
  - Decide whether you want memory or a register for the right operand
  - Note minimum of 3 busses between the Register array and the ALU

Random IC Wisdom - thanks to Carver Mead

An IC is a device where if you are lucky you sneak a few transistors in under the wires.

Still GPR Machines W on

- ❑ How many R' s vs. which structure is the debate of the day
- ❑ Lot's of options have gone to press

# Memory Ops per typical ALU instruction	Max ALU operands allowed	Examples
0	2 3	IBM RT-PC SPARC, MIPS, HP-PA, PowerPC, ALPHA
1	2 3	PDP-10, M6800, IBM 360, Intel 90x86 IBM 360RS
2	2 3	PDP-11, National 32x32, IBM 360SS, VAX NEC S1
3	3	VAX - blech!

- ❑ Hence Register (mem, operand) classification

Register -Register (0, 3) Pro' s + Con' s

- ❑ (m,n) m memory operands, n total operands in ALU instruction
- ❑ ALU is Register to Register -- i.e. pure RISC
- ❑ Advantages:
  - Simple fixed length instruction encoding
  - Decode is simple since instruction types are small
  - Simple code generation model
  - Instruction CPI tends to be very uniform
- ❑ Disadvantages:
  - Instruction count tends to be higher
  - Some instructions are short - wasting instruction word bits

Register -Memory (1, 2) P + C' s

- ❑ Evolved RISC and also old CISC - go figure?
  - New RISC machines capable of doing speculative and/or deferred loads
- ❑ Register - Memory ALU Architecture
- ❑ Advantages:
  - Data access to ALU immediate without loading first
  - Instruction format is relatively simple
  - Density is improved over Register (0, 3) model
- ❑ Disadvantages:
  - Operands are not equivalent - source operand may be destroyed
  - Need for memory address field may limit # of registers
  - CPI will vary

Memory-Memory (3, 3) P + C' s

- ❑ True Memory - Memory ALU Architecture
- ❑ True and most complex CISC model
  - currently extinct and likely to remain so
  - more complex memory actions are likely to appear but not directly linked to the ALU
- ❑ Advantages:
  - Most compact
  - Doesn' t waste registers for temporary values
- ❑ Disadvantages:
  - Large variation in instruction size - may need a shoe-horn
  - Large variation in CPI - i.e. work per instruction
  - Exacerbates the infamous memory bottleneck

Memory Addressing

- ❑ All architectures clearly need some way to address memory

A number of questions naturally arise

- ❑ What is accessed - byte, word, multiple words
  - for no good reason many of today' s machine are byte addressable
  - but the main memory is organized in 32 - 64 Byte lines to match cache model
- ❑ Hence there is a natural alignment problem
  - accessing a word or double-word which crosses 2 lines
    - requires 2 references
  - automagic alignment is possible but hides the number of references
    - also therefore hides an important case of CPI bloat
    - hence a bad idea

The Lilliputian W ars

Thanks to Prof. James Finnegan
Oceanview University
Oceanview Kansas

❑ When we address bytes - who gets to be first?

❑ Same gig with bit numbers - less of an ISA issue

❑ Big Endian - byte 0 is the MSB

❑ Little Endian - byte 0 is the LSB

❑ Wouldn' t be a big deal if we didn' t have byte serial communication and I/O devices

Other Alignment Issues

❑ Common convention is to expect aligned data objects

- compiler is responsible for keeping this straight
- hardware just checks - e.g. generates a trap if alignment restrictions are violated
- implies that op-code is type specific
  - LDB - load byte, LDW - load word, etc.

❑ Hence

- byte address is anything - never misaligned
- half word - even addresses - low order address bit = 0 else trap
- word - low order 2 address bits = 0 else trap
- double word - low order 3 adress bits = 0 else trap

Addressing Modes

❑ An important aspect of ISA design

- has major impact on both the HW complexity and the IC
- HW complexity affects the CPI and the cycle time

❑ Basically a set of mappings

- from address specified to address used
- address used = *effective address*
- Effective address may go to memory or to a register array
  - which is typically dependent on it' s location in the instruction field
  - in some modes multiple fields are combined to form a memory address
  - register addresses are usually more simple - e.g. they need to be fast
- effective address generation is an important focus
  - since it is the common case - e.g. every instruction needs it
  - it must also be fast

❑ Note tower of Babel effect

- address mode names are vendor specific

Typical Address Modes I

Mode	Example Instruction	Meaning	Use
Register	Add R4, R3	Regs[R4] <- Regs[R4] + Regs[R3]	All RISC ALU operations
Immediate	Add R4, #3	Regs[R4] <- Regs[R4] + 3	for small constants - problems?
Displacement	Add R4, 100(R1)	Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]	accessing local variables
Register deferred or Indirect	Add R4, (R1)	Regs[R4] <- Regs[R4} + Mem[Regs[R1]]	pointers
Indexed	Add R3, (R1 + R2)	Regs[R3] <- Regs [R3] + Mem[Regs[R1] + Regs[R2]]	array access - R1 is the base, R2 is the index
Direct or absolute	Add R1, (1001)	Regs[R1] <- Regs[R1] + Mem[1001]	

Typical Address Modes II

Mode	Example Instruction	Meaning	Use
Memory Indirect or Memory Deferred	Add R1, @R3	Regs[R1] <- Regs[R1] + Mem[Mem[Regs[3]]]	If R3 holds a pointer address, then result is the full dereferenced pointer
Autoincrement in this case post increment note symmetry with autodec	Add R1, (R2) +	Regs[R1] <- Regs[R1] + Mem[Regs[R2]]; Regs[R2] <- Regs[R2] + d	Array walks - if element of size d is accessed then pointer increments auto
Autodecrement in this case predecrement	Add R1, - (R2)	Regs[R2] <- Regs[R2] - d; Regs[R1] <- Regs[R1] + Mem[Regs[R2]];	array walks, with autoinc useful for stack implementation
Scaled	Add R1, 100 (R2) [R3]	Regs[R1] <- Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]	array access - may be applied to indexed addressing in some machines

Some Mind Games

That might just show up on an exam for example

- Take a program in one mode and convert it to a new program with a new mode as a possibility
- Show the effective address computation path for some mode
- Consider implementing some common data structures
- Consider common idioms - matrix multiply , etc.
- For the fragment which modes would be best
- Which would win for XEQ time ignoring cost issues
- Given a set of modes, show the Inst. formats

Benchmarks show Mode Importance

Based on a VAX which supported everything - SHEESH!
All Frequencies are based on dynamic traces

- Memory Indirect
  - T eX = 1%, Spice = 6%, GCC = 1%
- Scaled
  - T eX = 0%, Spice = 16%, GCC = 6%
- Register Deferred
  - T eX = 24%, Spice = 3%, GCC = 1 1%
- Immediate
  - T eX = 43%, Spice = 17%, GCC = 39%
- Displacement
  - T eX = 32%, Spice = 55%, GCC = 40%

Determining Field Size

REMEMBER - Optimize for the Common Case

- Analyze your programs with dynamic traces
- Then prioritize:
  - Displacement field size - e.g. see Fig. 2.7
  - Immediate or Literal field size - Figs 2.8 & 2.9
- Determine which modes appear to win - Fig. 2.6
  - Consider the instruction encoding' s that might work
  - Build the datapath model and consider CPI and cycle time
  - Evaluate expected code density
- Also consider register amounts
  - Structured or general purpose
  - How do these issues affect the above?

Control Instructions are a Big Deal

- ❏ Jumps - unconditional transfer
- ❏ Conditional Branches
  - How is condition code set?
  - How is target specified? How far away is it?
- ❏ Calls
  - How is target specified? How far away is it?
  - Where is return address kept?
  - How are the arguments passed? Callee vs. Caller save!
- ❏ Returns
  - Where is the return address? How far away is it?
  - How are the results passed?

Keep in mind what implications are made on the linker!

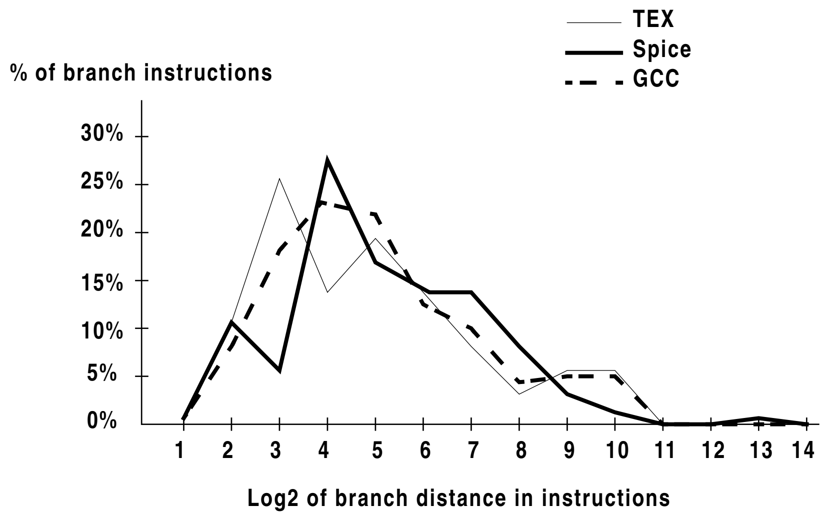
Branch Address Specification

- ❏ Known at compile time for uncondix and condix branches - hence specified in the instruction
  - 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
- ❏ How do you find out what works?
  - Start by measuring your programs of course.

Branch Stats - 90% are PC Relative

- ❏ Call/Return
  - T eX = 16%, Spice = 13%, GCC = 10%
- ❏ Jump
  - T eX = 18%, Spice = 12%, GCC = 12%
- ❏ Conditional
  - T eX = 66%, Spice = 75%, GCC = 78%
- ❏ What else do you need to know?
  - % of each in overall instruction frequency
  - + target range distribution
  - Loop vs. Exit probability - is the Baskett bit worth it?
  - What is the frequency of conditions tested?
  - Real backward vs. forward branches
  - Effect of exceptions

Branch Distances on MIPS 2000



Condition Testing Options

Option	How Tested	Advantages	Disadvantages
Condition Code or CC	**Special "PSW" bits set by ALU, possibly via direct instruction set**	**Often the CC is set for free**	**Adds state, constrains the instruction ordering**
Condition Register	**Comparison results put in a register**	**Simple Less instruction ordering constraints.**	**Uses up a register**
Compare and Branch	**Compare is part of the branch instruction and condition is used to determine the branch decision**	**One instruction rather than two**	**May be too much work If so then either CPI or cycle time may be extended**

What kinds of compares do Branches Use?

Also note the value range and probability of immediate data

- LT/GE
  - Tex 25%, Spice 0%, GCC 11%
- GT/LE
  - Tex 3%, Spice 25%, GCC 0%
- EQ/NE
  - Tex 72%, Spice 75%, GCC 89%
- Compares that use immediates
  - Tex 83%, Spice 92%, GCC 84%
- Notice - on the average 50% of the compares are a simple EQ or NE with 0!!!!

Direction, Frequency, and real Change

Program	% of Backward Branches	% Branches Taken	% of all control instructions that modify the PC
GCC	**26%**	**54%**	**63%**
Spice	**31%**	**51%**	**63%**
TeX	**17%**	**54%**	**70%**
Average	**25%**	**53%**	**65%**

- Key points:
  - Most backward branches are loops - taken about 90%
  - Branch statistics are both compiler and application dependent
  - Any loop optimizations may have large effect

Operand Type & Size

- Specified by instruction or by hardware tag
  - tagged machines are extinct - perhaps they shouldn't be
- Typical types: assume word=32 bits
  - character - byte - ASCII or EBCDIC (IBM) - 4 per word
  - short integer - 2-bytes, 2's complement
  - integer - one word - 2's complement
  - float - one word - usually IEEE 754 these days
  - double precision float - 2 words - IEEE 754
  - BCD or packed decimal - 4-bit values packed 8 per word
- Instructions will be needed for common conversions - software can do the rare ones

Operand Type & Size

- Specified by instruction or by hardware tag
  - tagged machines are extinct - perhaps they shouldn't be
- Typical types: assume word=32 bits
  - character - byte - ASCII or EBCDIC (IBM) - 4 per word
  - short integer - 2-bytes, 2's complement
  - integer - one word - 2's complement
  - float - one word - usually IEEE 754 these days
  - double precision float - 2 words - IEEE 754
  - BCD or packed decimal - 4-bit values packed 8 per word
- Instructions will be needed for common conversions - software can do the rare ones

Data Access Patterns

- 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%
- The future - as we go to 64 bit machines
  - decimal's future is unclear
  - larger offsets, immediates, etc. is likely
  - usage of 64 and 128 bit values will increase

University of Utah
Department of Computer Science
CS561