Instruction Set Architectures: History and Issues

Many slides taken from Dr. Srinivasan Parthasarathy. Some figures from our text. Any errors are my own…

Schedule of things to do

- HW5 posted.
  - Due on 4/23 (last day of class) @10pm – 24 hours late with only 5% off.
- MS3 meeting out later today.
- Project is due on Saturday 4/20 at 9pm.
  - Last synth job can still be running, but don't rely on it.
  - No code changes after this.
- Oral and written reports are due on Tuesday 4/23.
  - Brief directions for both posted shortly.
  - Final written report is due at 9pm on Piazza
  - Oral reports are during the day.
  - Will watch at least 2 other talks.
    - Can go to all if you wish.

Computer Architecture's Changing Definition

- 1950s to 1960s:
  - Computer Architecture Course =
    - Computer Arithmetic
- 1970s to 1980s:
  - Computer Architecture Course =
    - Instruction Set Design (especially ISA appropriate for compilers)
- 1990s+
  - Computer Architecture Course =
    - Design of CPU (microarchitecture)
    - Design of memory system & I/O system
    - Multiprocessor/multi-thread issues

Instruction Set Architecture: The interface between hardware and software

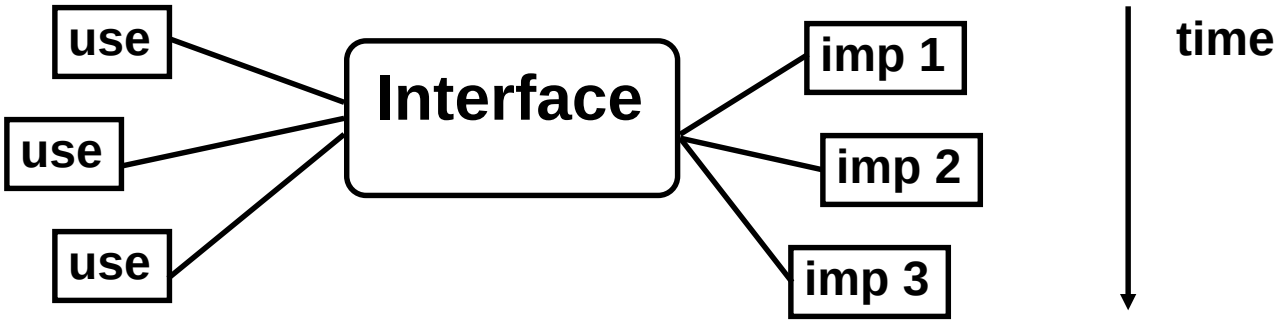
- Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine.

- The instruction set architecture is also the machine description that a hardware designer must understand to design a correct implementation of the computer.

Interface Design

A good interface:

- **Lasts through many implementations (portability, compatibility)**
- **Is used in many different ways (generality)**
- **Provides *convenient* functionality to higher levels**
- **Permits an *efficient* implementation at lower levels**



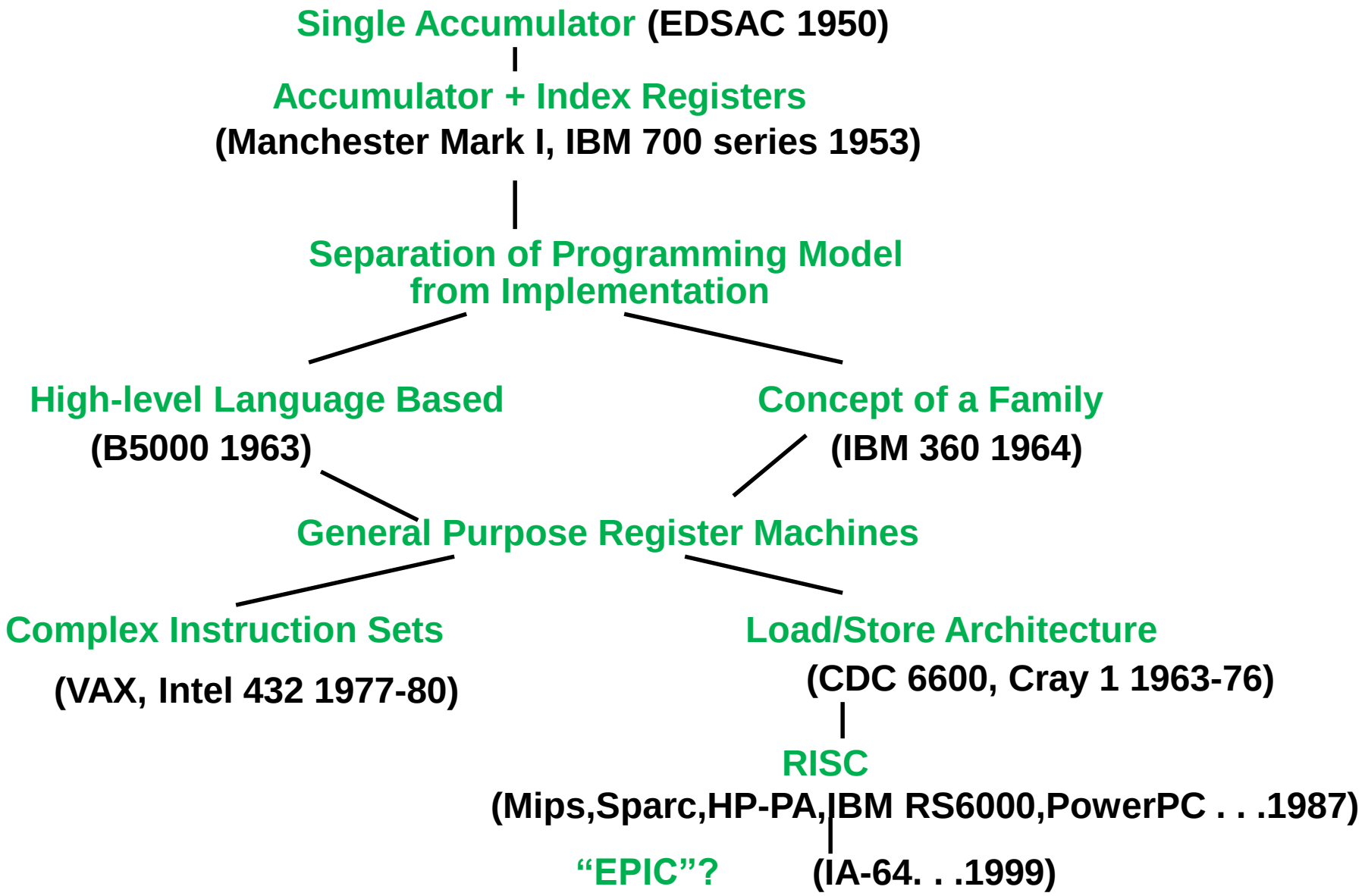
Today's outline

- History of ISA design

- Overview of ISA options
  - Classification into 0,1,2,3 address machines
  - Addressing modes
  - Other issues

- Sum-up.

Evolution of Instruction Sets



Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- And they in turn influence these

Today's outline

- History of ISA design

- Overview of ISA options
  - Classification into 0,1,2,3 address machines
  - Addressing modes
  - Other issues

- Sum-up.

What Are the Components of an ISA?

- Sometimes known as *The Programmer's Model* of the machine
- Storage cells
  - General and special purpose registers in the CPU
  - Many general-purpose cells of same size in memory
  - Storage associated with I/O devices
- The machine instruction set
  - The instruction set is the entire repertoire of machine operations
  - Makes use of storage cells, formats, and results of the fetch/execute cycle
    - e.g.., register transfers

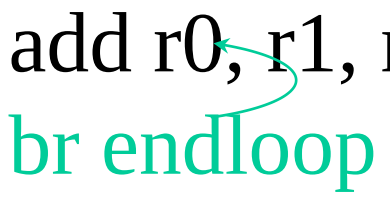
What Must an Instruction Specify?(I)

Data Flow ←

- Which operation to perform **add r0, r1, r3**
  - Ans: Op code: add, load, branch, etc.
- Where to find the operands: **add r0, r1, r3**
  - In CPU registers, memory cells, I/O locations, or part of instruction
- Place to store result **add r0, r1, r3**
  - Again CPU register or memory cell

What Must an Instruction Specify?(II)

- Location of next instruction



  - Almost always memory cell pointed to by program counter—PC
- Sometimes there *is* no operand, or no result, or no next instruction.
  - Can you think of examples?

Instructions Can Be Divided into 3 Classes (I)

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - *Load*—source is memory and destination is register
  - *Store*—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - *Add*, *Sub*, *Shift*, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - *Br Loc*, *Brz Loc2*,—unconditional or conditional branches

Classifying ISAs

Accumulator (before 1960):

1 address	add A	acc <– acc + mem[A]

Stack (1960s to 1970s):

0 address	add	tos <– tos + next

Memory-Memory (1970s to 1980s):

2 address	add A, B	mem[A] <– mem[A] + mem[B]
3 address	add A, B, C	mem[A] <– mem[B] + mem[C]

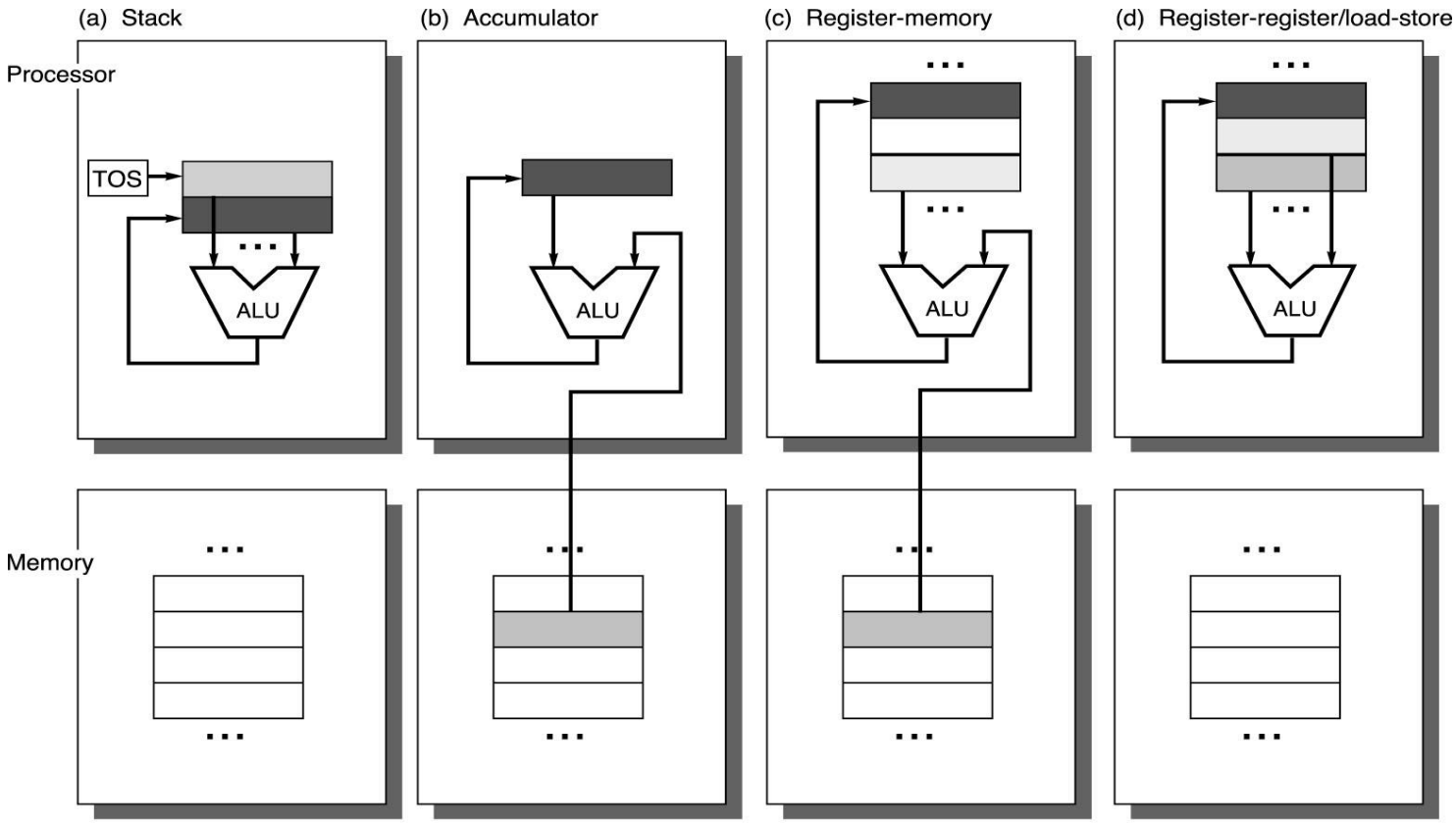
Register-Memory (1970s to present):

2 address	add R1, A	R1 <– R1 + mem[A]
	load R1, A	R1 <_ mem[A]

Register-Register (Load/Store) (1960s to present):

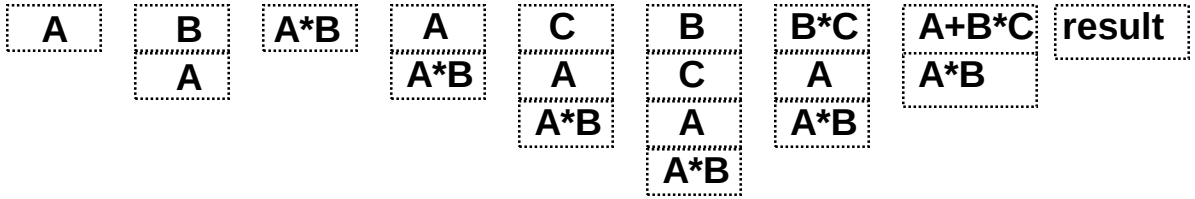
3 address	add R1, R2, R3	R1 <– R2 + R3
	load R1, R2	R1 <– mem[R2]
	store R1, R2	mem[R1] <– R2

Classifying ISAs



Stack Architectures

- Instruction set:
  - add, sub, mult, div, . . .
  - push A, pop A
- Example: A*B - (A+C*B)
  - push A
  - push B
  - mul
  - push A
  - push C
  - push B
  - mul
  - add
  - sub



Stacks: Pros and Cons

- Pros
  - Good code density (implicit operand addressing→ top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures
- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional instructions like SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures
  - How about an OoO machine?

Accumulator Architectures

- **Instruction set:**
  - **add A, sub A, mult A, div A, . . .**
  - **load A, store A**

- **Example: A*B - (A+C*B)**

  load B | **mul C** | **add A** | **store D** | **load A** | **mul B** | **sub D**

B	B*C	A+B*C	A+B*C	A	A*B	result

Accumulators: Pros and Cons

- **Pros**
  - **Very low hardware requirements**
  - **Easy to design and understand**
- **Cons**
  - **Accumulator becomes the bottleneck**
  - **Little ability for parallelism or pipelining**
  - **High memory traffic**
  - **How about an OoO machine?**

Memory-Memory Architectures

- **Instruction set:**

  (3 operands) **add A, B, C** **sub A, B, C** **mul A, B, C**

- **Example: A*B - (A+C*B)**
  - **3 operands**

    mul D, A, B

    mul E, C, B

    add E, A, E

    sub E, D, E

Memory-Memory: Pros and Cons

- **Pros**
  - **Requires fewer instructions (especially if 3 operands)**
  - **Easy to write compilers for (especially if 3 operands)**

- **Cons**
  - **Very high memory traffic (especially if 3 operands)**
  - **Variable number of clocks per instruction (especially if 2 operands)**
  - **With two operands, more data movements are required**
  - **How about an OoO machine?**

Register-Memory Architectures

- **Instruction set:**

  add R1, A **sub R1, A** **mul R1, B**

  load R1, A **store R1, A**

- **Example: A*B - (A+C*B)**

```
load R1, A
mul R1, B        /*    A*B          */
store R1, D
load R2, C
mul R2, B        /*    C*B          */
add R2, A        /*    A + CB       */
sub R2, D        /*    AB - (A + C*B) */
```

Memory-Register: Pros and Cons

- **Pros**
  - **Some data can be accessed without loading first**
  - **Instruction format easy to encode**
  - **Good code density**

- **Cons**
  - **Operands are not equivalent (poor orthogonality)**
  - **Variable number of clocks per instruction**
  - **May limit number of registers**

Load-Store Architectures

- **Instruction set:**

  add R1, R2, R3 sub R1, R2, R3 mul R1, R2, R3
  load R1, R4 store R1, R4

- **Example: A*B - (A+C*B)**

```
load R4, &A
load R5, &B
load R6, &C
mul R7, R6, R5          /*   C*B          */
add R8, R7, R4          /*   A + C*B      */
mul R9, R4, R5          /*   A*B          */
sub R10, R9, R8         /*   A*B - (A+C*B) */
```

Load-Store: Pros and Cons

- **Pros**
  - **Simple, fixed length instruction encoding**
  - **Instructions take similar number of cycles**
  - **Relatively easy to pipeline**

- **Cons**
  - **Higher instruction count**
  - **Not all instructions need three operands**
  - **Dependent on good compiler**
    - Need to schedule registers well at the least.

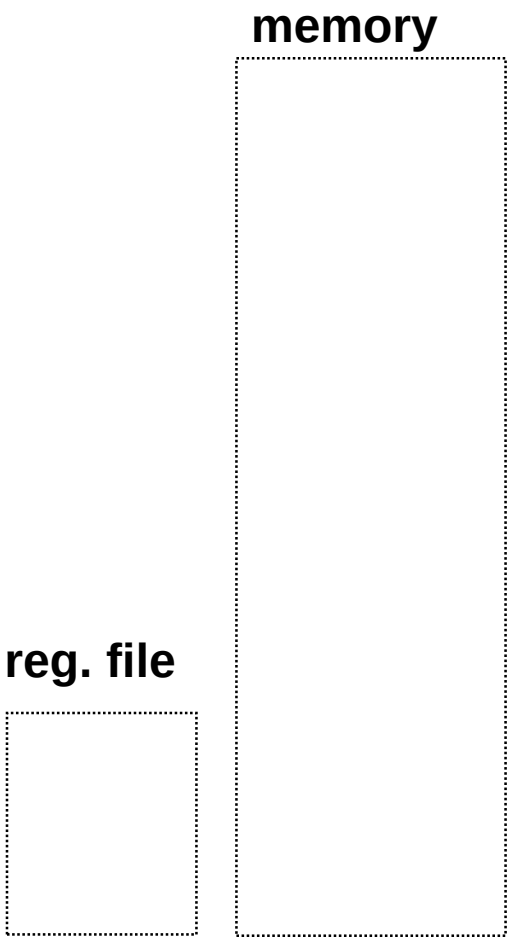
Comparing Code Density

Stack	Accum.	Reg-Mem	Load/Store
push A	load B	load R1, A	load R4, &A
push B	mul C	mul R1, B	load R5, &B
mul	add A	store R1, D	load R6, &C
push A	store D	load R2, C	mul R7, R6, R5
push C	load A	mul R2, B	add R8, R7, R4
push B	mul B	add R2, A	mul R9, R4, R5
mul	sub D	sub R2, D	sub R10, R9, R8
add			
sub			

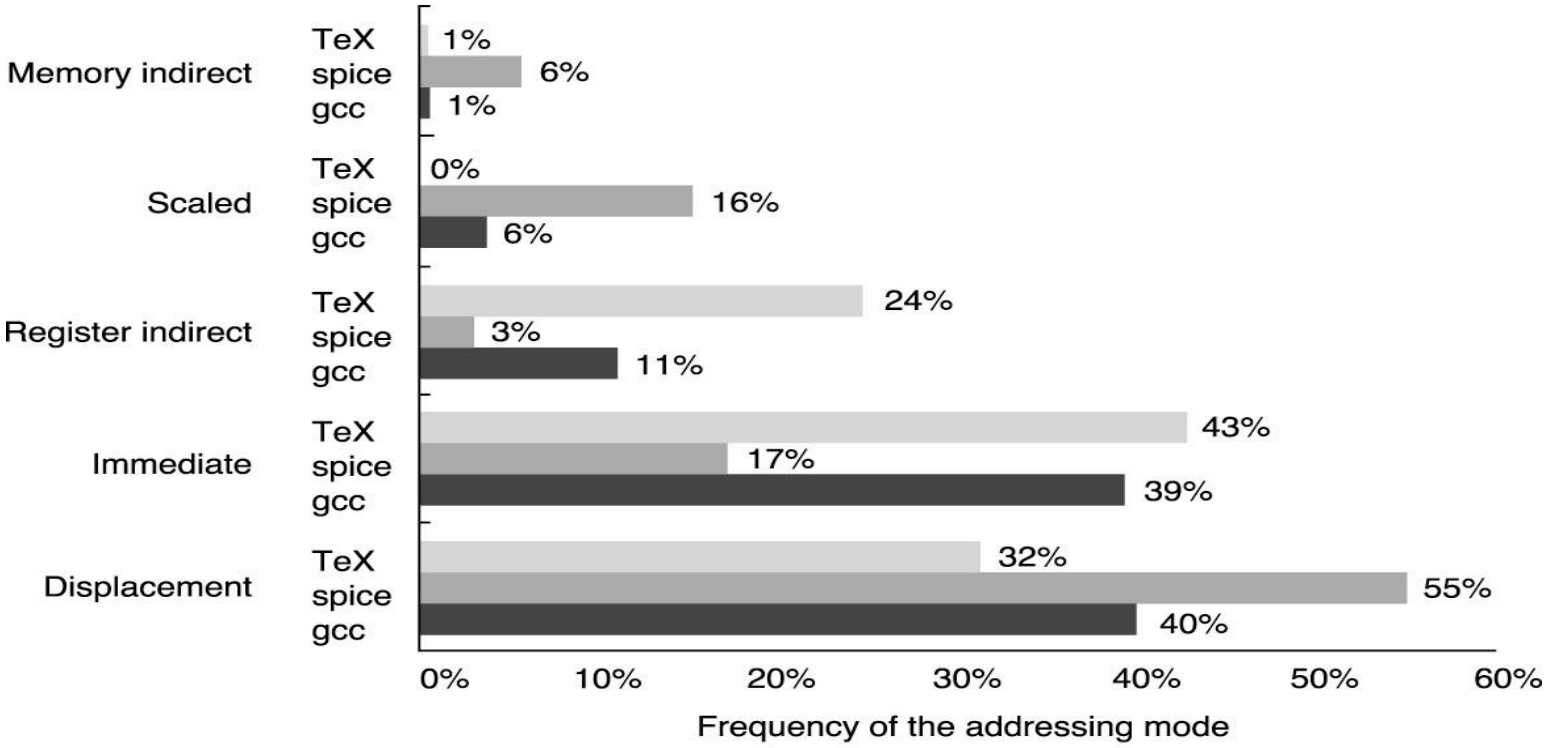
If we need 5 bits to specify a register, 16 bits to specify a memory location and 8 bits to specify the opcode, how many bits do we use for each scheme?

Types of Addressing Modes (VAX)

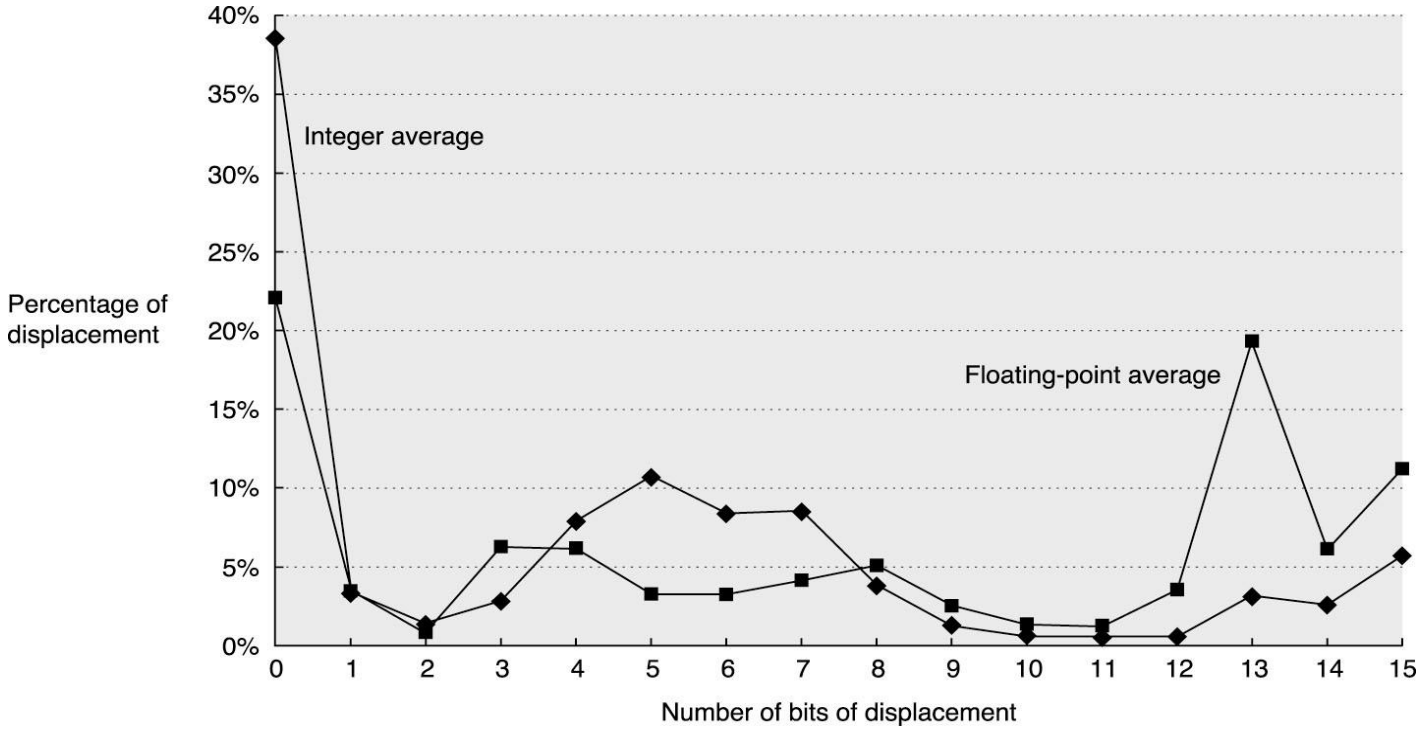
1. Register direct	Ri
2. Immediate (literal)	#n
3. Displacement	M[Ri + #n]
4. Register indirect	M[Ri]
5. Indexed	M[Ri + Rj]
6. Direct (absolute)	M[#n]
7. Memory Indirect	M[M[Ri]]
8. Autoincrement	M[Ri++]
9. Autodecrement	M[Ri - -]
10. Scaled	M[Ri + Rj*d + #n]



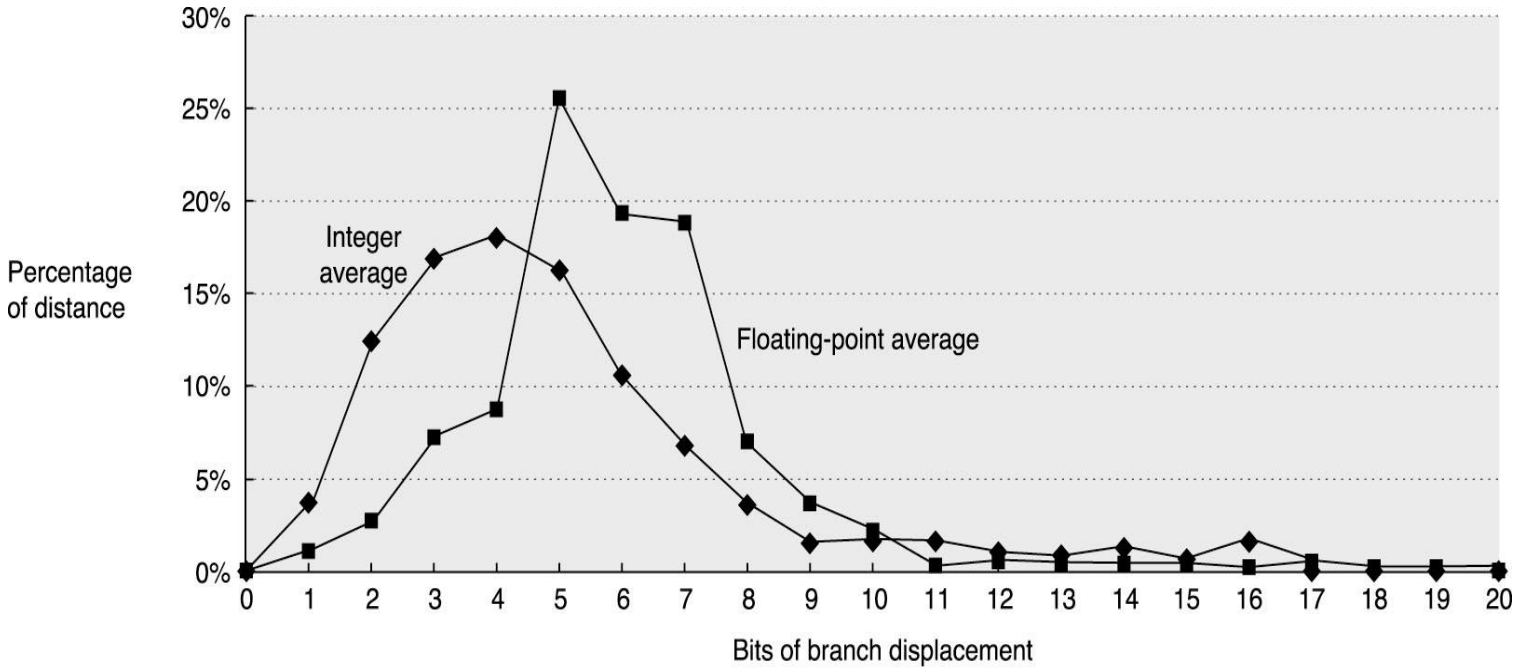
Summary of Use of Addressing Modes



Distribution of Displacement Values



Branch Distances (in terms of number of instructions)



Registers vs. Memory

- **Advantages of Registers**
  - **Faster than cache (no addressing mode or tags)**
  - **Deterministic (no misses)**
  - **Can replicate (multiple read ports)**
  - **Short identifier (typically 3 to 8 bits)**
  - **Reduce memory traffic**

- **Disadvantages of Registers**
  - **Need to save and restore on procedure calls and context switch**
  - **Can't take the address of a register (for pointers)**
  - **Fixed size (can't store strings or structures efficiently)**
  - **Generally limited in number**

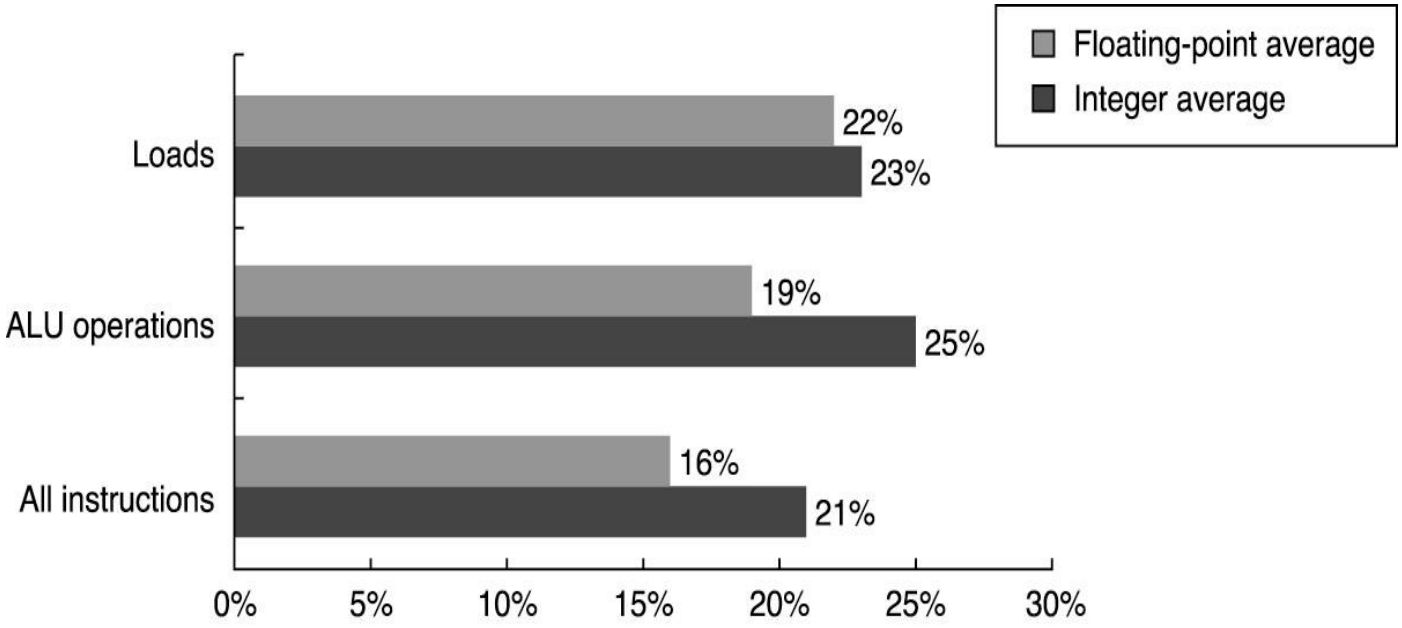
How about something other than registers and memory?

- **We have two namespaces.**
  - **Why not 3?**
- **What other name spaces might make sense?**
-

Alignment Issues

- If the architecture does not restrict memory accesses to be aligned then
  - Software is simple
  - Hardware must detect misalignment and make 2+ memory accesses
  - Expensive detection logic is required
  - All references can be made slower
- Sometimes unrestricted alignment is required for backwards compatibility
- If the architecture restricts memory accesses to be aligned then
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned
- Since we want to make the common case fast, having restricted alignment is often a better choice, unless compatibility is an issue

Frequency of Immediate Operands



80x86 Instruction Frequency (SPECint92, Fig. 2.16)

Rank	*Instruction*	*Frequency*
1	load	22%
2	branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	register move	4%
9	call	1%
10	return	1%
Total		96%

Today's outline

- History of ISA design
- Overview of ISA options
  - Classification into 0,1,2,3 address machines
  - Addressing modes
  - Other issues
- Sum-up.

Encoding an Instruction Set

- What are the metrics of goodness?
  - $T_{Program}$ is always the main measure.
  - But what goes into that?
    - Number of instructions
    - Time it takes to execute each instruction
      - Complexity of instruction
        - » Decode
        - » Execute
      - Size if code total (yes, that double counts number of instructions to some extent)
      - Impact on parallelization

Encoding an Instruction Set

- Some impacts are pretty obvious
  - If you need fewer bits for a given program, you can expect a higher Icache hit rate.
  - If instructions aren't regular (which field selects the input registers, variable instruction word lenght) you can expect a longer decode time.
- Some aren't
  - Discuss how the ISA might make a superscalar out-of-order processor difficult to build.

Encoding an Instruction Set

- Consider a load/store machine that uses immediates
  - 5 bits for registers, 16 bits for immediates.
    - What percent of a 32-bit ISA encoding does a 3 register argument instruction use?
    - An instruction using two registers and an immediate?
  - What would be the downside to using a 12-bit immediate? The upside?

Encoding an Instruction Set

- A desire to have as many registers and addressing modes as possible
- The impact of size of register and addressing mode fields on the average instruction size and hence on the average program size
- A desire to have instruction encode into lengths that will be easy to handle in the implementation