Instruction Set Architecture

Instruction Set Architecture is the HW/SW interface

- Agreement between programmer and hardware
- Defines the visible state of the system
- Defines how each instruction changes that state

Programmers use ISA to model HW

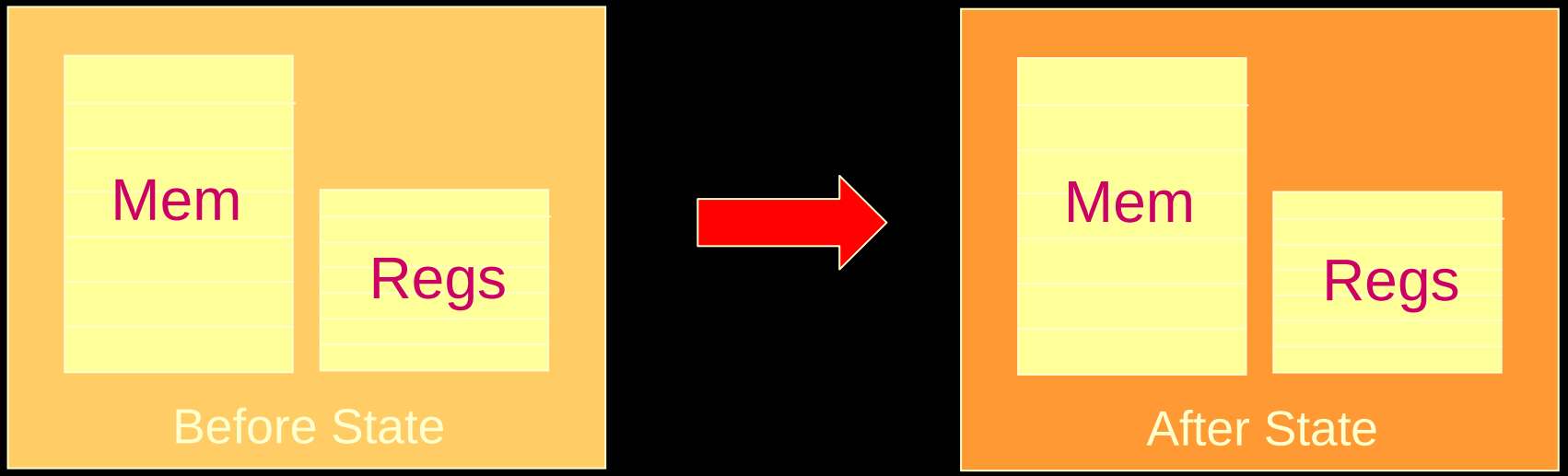
- Simulators
- Performance estimation

Designers use ISA as definition of correctness

ISA defines instructions and encodings but also...

ISA Overview

Instruction format
Instruction types
Address mode
Operands



Machine State, PC
Memory Organization
Register Organization

Architecture vs. Implementation

Architecture defines the “what”

- What is the programmer visible state
- What happens on each instruction

Implementation defines the “how”

- The sequence of steps
- The time it takes

Why separate architecture and implementation?

- Compatibility (VAX, ARM)
- Longevity (x86 -- 11 generations!)
- Amortize research investment
- Retain software investment (SW is more important??)

Architecture Families

Many architectures “grow” with time

Companies make families of chips that run the same programs

- Binary compatibility

8088, 8086, 80286, 80386, 80486, Pentium, Pentium MMX, Pentium II, Pentium III, Pentium 4

68000, 68008, 68010, 68020, 68030, 68040, 68060

R2000, R3000, R6000, R4000, R8000, R5000, R10000, R12000, R14000, R18000

Chips in same family do have different ISAs

- But cores are the same
- Need to recompile to see new ISA benefits

Architecture or Implementation

- Number of GP registers
- Width of the data bus
- Binary representation of the instruction
- Number of cycles a floating point add takes
- Number of cycles processor must wait after a load before it can use the data
- Floating point format supported
- Size of the instruction cache
- Number of instructions that issue each cycle
- Number of addressing modes

Classifying ISAs

Type of internal storage is basic differentiation

- Stack, accumulator, or registers

Number of operands (in parentheses) is tied closely

- Stack (0 or 1) `push A; push B; add; pop C`
  - Stack is the *implicit* operand
- Accumulator (1) `load A; add B; store C`
  - Accumulator is *implicit* operand
- General purpose registers
  - Register-memory `load r1,A; add r1,B, store r1,C`
  - Register-register `load r1,A; load r2,B, add r3,r2,r1; store r3, C`
- Memory-memory `add C,A,B` (ancient history)

Load-store Architectures

Virtually every machine designed since 1980

- Only loads/stores access memory

Why?

- Registers are faster than memory
- Registers are easier for compilers

Ex: stack architectures must process left to right

Registers can

- Process in any order
- Hold variables
- Improve code density

Disadvantage?

- Encoding! (Code size)

Compiler Effects

Compiler wants many general purpose registers

- Less *register pressure*
- Interchangeability (few special-purpose regs)

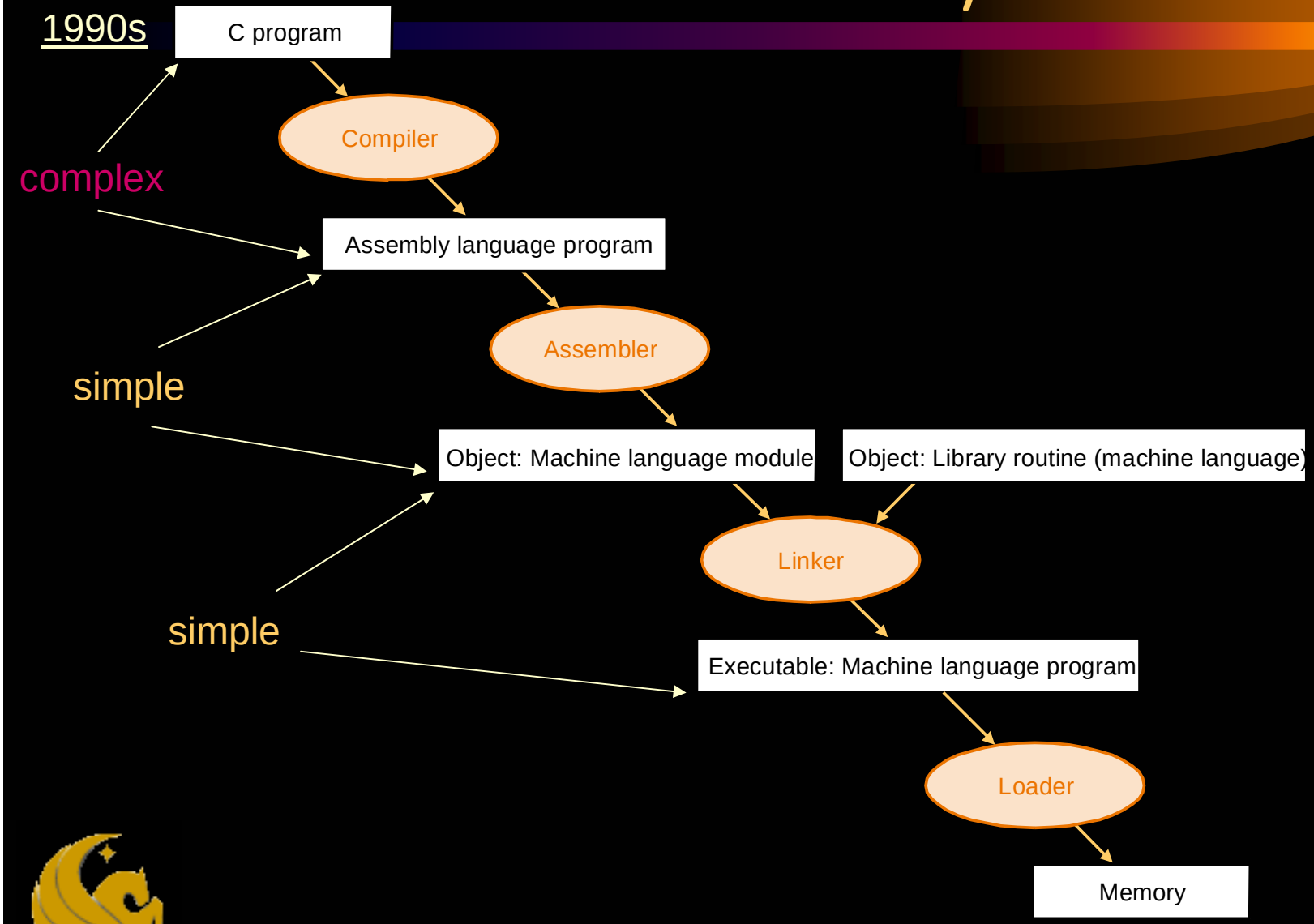
Compiler wants orthogonality (regularity)

- Law of least surprise
- No strange side effects
- Consistent addressing modes

3 more items on the compiler wish list

- Provide primitives, not solutions
- Simplify trade-offs of alternatives
- Make compile-time constants fast

Semantic Gap



Memory Addressing

Byte addressing

- Since 1980 every machine can address 8-bit bytes
- MIPS memory is linear array of 2^{32} bytes (32-bit addresses)

But natural load size is not a byte

- Typically a word (4 bytes)
- Or a double word (8 bytes)
- Most also support half words (2 bytes)

Questions:

- How do byte addresses map into words?
  - Byte order
- How can words be positioned in memory?
  - Alignment

Byte Ordering

Two Conventions

- Big Endian, specify address of most significant byte
- Little Endian, specify address of least significant byte

No technical significance to distinction (just religious!)

- Big Endian: Amiga, 68K Macs, IBM RS6K, SGI, Sun
- Little Endian: Alpha, DEC, Vax, x86
- Recently many processors are “bimodal”
  - MIPS, PowerPC (both mostly Big Endian)

Names based on Gulliver’s Travels

(http://lamicounter.epfl.ch/users/erik/litt/endianne.html)

Memory Alignment

Alignment

- Object located at address that is a multiple of its size

Important performance effect

- Also logical simplification
  - Removes complexity of sequencing memory references
  - Especially difficult when crossing cache lines or virtual pages

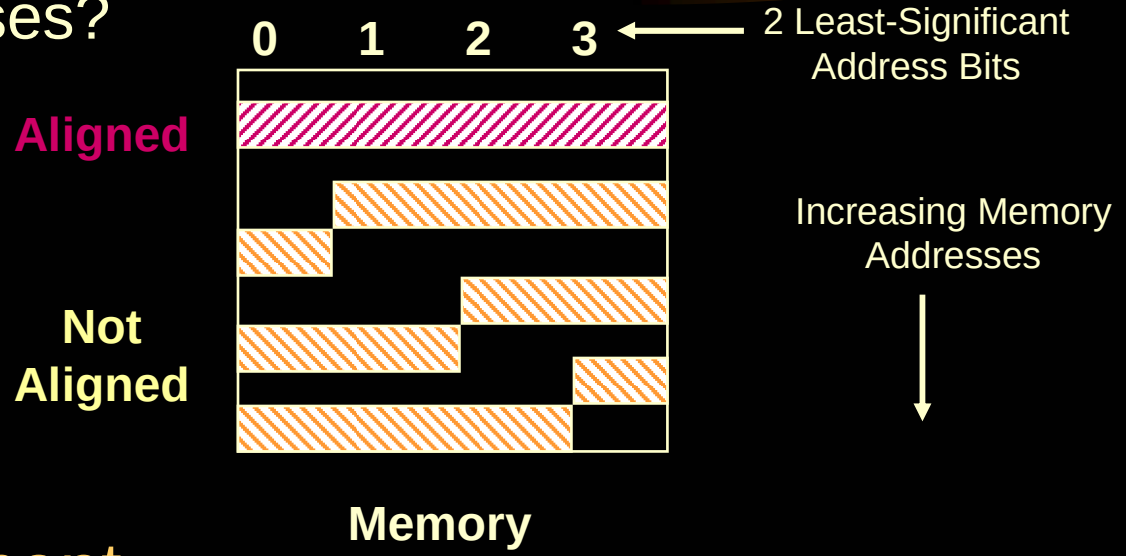
Historically

- Early machines (IBM 360 in 1964) require alignment
- Restriction removed in 70s: too hard for programmers!
- RISC: reintroduced for performance and simplicity
  - Memory is cheaper

Alignment: Example

32-bit word

- 2 accesses?



Word alignment

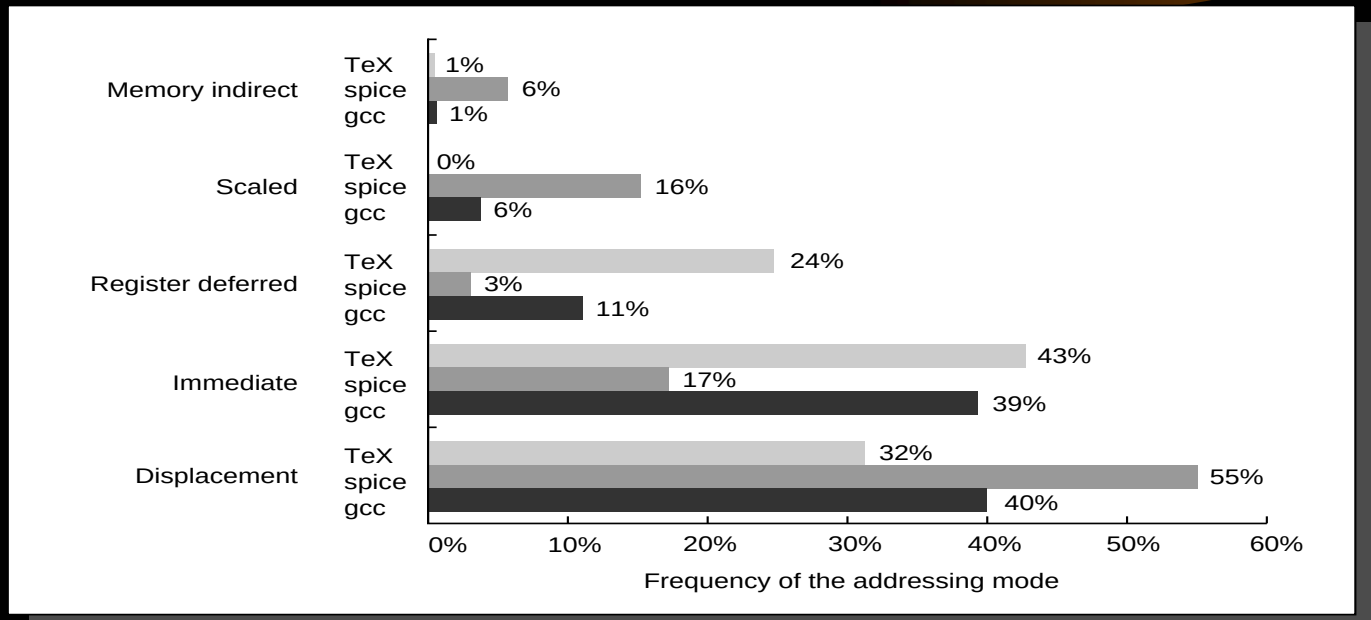
- xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|00
- xxxxxxxxxxxxxxxxxxxxxxxxxxxxx|000
- (important trick later on for instruction encoding)

Addressing Modes

Register	`add r1,r2`
Immediate	`add r1,0x4`
Displacement	`add r1,100(r2)`
Register Indirect	`add r1,(r2)`
Indexed	`add r1,(r2+r3)`
Direct	`add r1,(0x3428)`
Memory Indirect	`add r1,@(r2)`
Auto Increment	`add r1,(r2)+`
Auto Decrement	`add r1,-(r2)`

How Many are Needed?

VAX had them all!



99% of all addressing modes

How Many Bits?

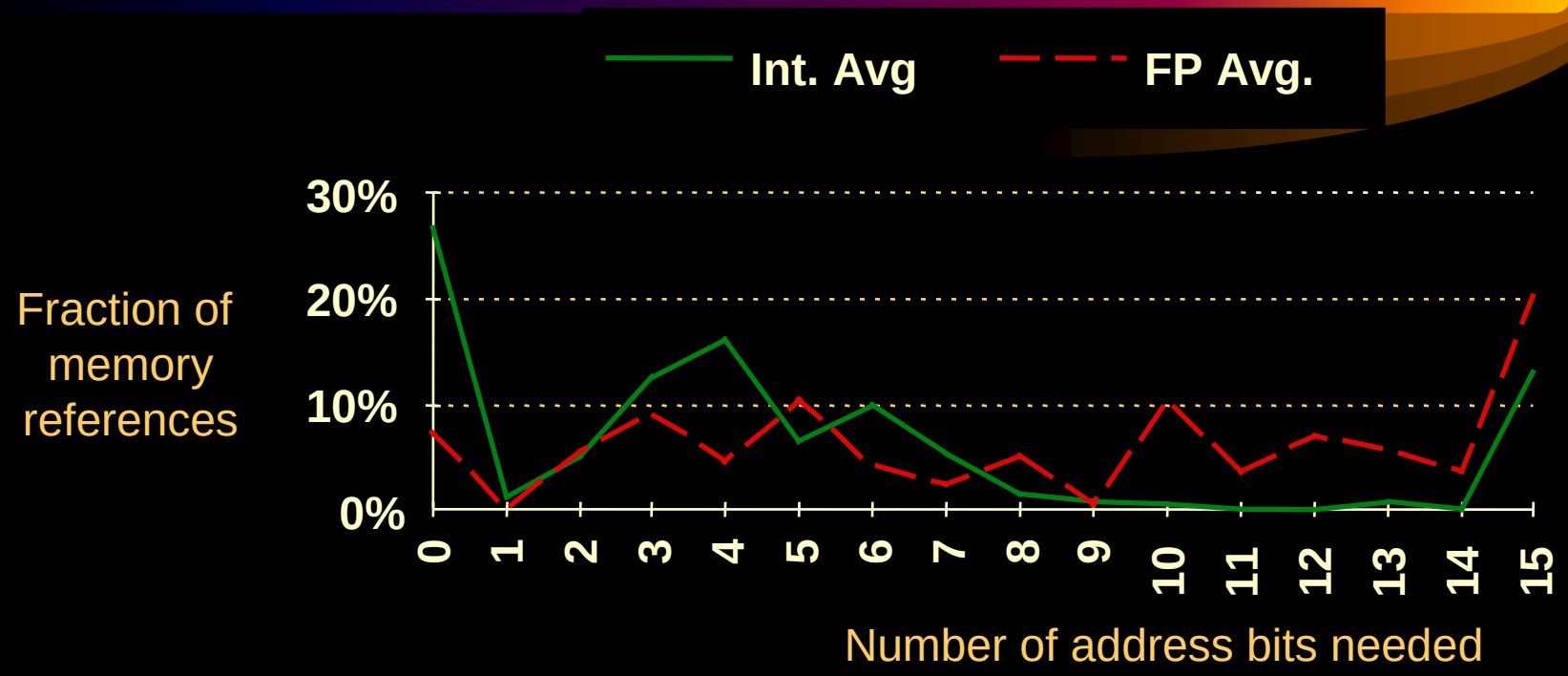
How big are immediates?

- 50% to 60% fit within 8 bits
- 75% to 80% fit within 16 bits
- Assuming sign extension!

What about displacements?

- Another study from the MIPS architecture...

Displacements



Average of 5 SPECfp and 5 SPECint programs

- 1% of addresses need > 16 bits
- 12-16 bits sufficient

Instruction Types

Arithmetic and logical

- Add, and, or, sub, xor, shift, …

Data movement

- Loads/stores
- Register to register

Control

- Conditional branches
- Jumps + syscall
- Compares
- Procedure call/return

Floating point

Misc. junk (MMX, graphics, string, BCD)

Control Instructions

Conditional branches

- `branch <cond> <target>`
- Typical conditions are eq and ne
  - `bne r1,r2,target`
  - `beq r3,r7,target`

Unconditional jumps

- `jump <target>`

Targets are often *PC-relative*

- Fewer bits (target is typically close by)
- Position independence

Jump Register

How do you return from a procedure call?

- Return address is not known at compile time!

Use jump register:

- `jr r31`
- Combine with jump-and-link: `jal ProcedureName`

Jump register also used for

- Case or switch statements
- Virtual functions
- Dynamically linked libraries
- Anything where target is not known at compile time

Condition Codes…or Not

Condition codes

- Bits set by ALU about the most recently computed result
- \+ "Free" comparison
- \- Extra state, constrain possible inst ordering, hard to pipeline

Compare and branch

- Compare is part of the branch (limited to a subset)
- \+ One instruction rather than two, no extra state
- \- Cycle time concerns (too much work)
  - MIPS pipeline treats these compares specially

Procedure Calls

How to preserve registers across a procedure call?

- Save registers to the stack
- All of them?

Can establish a *calling convention*

- In software
- pact as to which registers need saving and by whom

Caller-saved registers

- registers saved by caller if live. callee may use at will

Callee-saved regisers

- registers saved by the callee if used

Instruction Encoding

How many gp registers?

- Need $\log_2 N$ bits per register, called *register specifier*

How many addressing modes?

How many opcodes and operands?

Trade-offs:

- Instruction size vs. ease of decoding
- Instruction cache effects
- Program size

Fixed length vs. variable length instructions

Fixed vs. Variable Length

Variable length instructions

- Give more efficient encodings
- No bits wasted for unused fields in instruction
- Can frequency code common operations
- Examples: VAX, Intel x86
- But, can make implementation difficult
  - Sequential determination of each operand!

Compromise: a few good formats

- Can be either fixed length or a few (3) variable length
- Most RISC machines used fixed-length encoding
- Operand locations are easy to find
- Important for pipelining and quick comparisons