Instruction Set Architecture

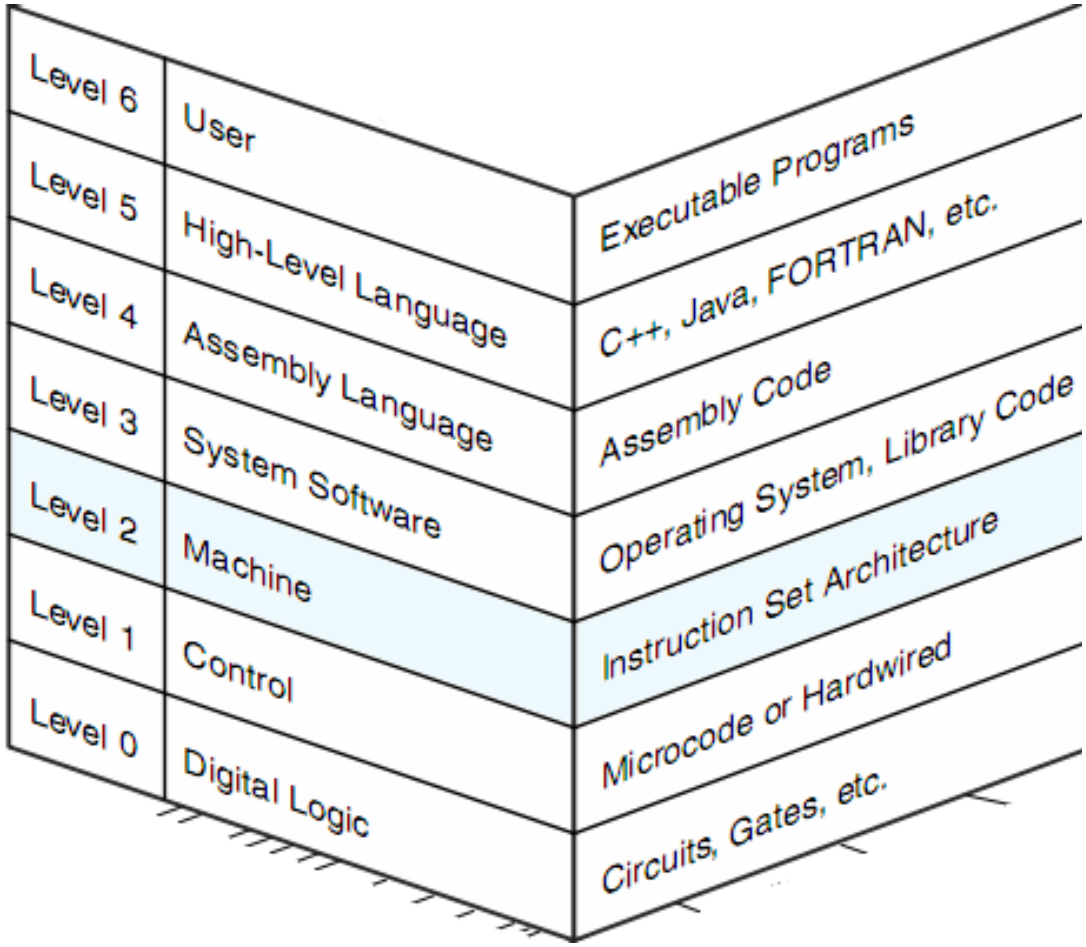
Design and Tradeoffs

Purpose and Scope

- What is ISA design?
  - Where does ISAs fit into Computer Architecture?
  - Instruction format + trade-offs
    - Byte-Ordering
      - Little endian/big endian
    - Instruction length
      - Fixed/Variable
    - Number of opcodes
      - 0,1,2,3 opcodes

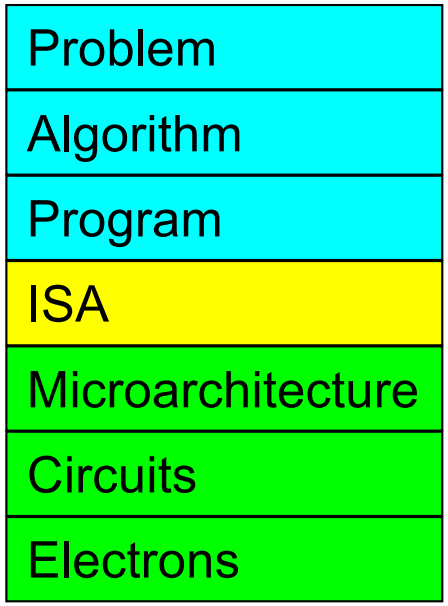
Instruction Set Architecture

- Computer Architecture = Hardware + ISA
- Instruction Set Architecture
  - Interface between all the **software** that runs on the machine and the **hardware** that executes it
  - Allows you to talk to the machine



Levels of Transformation

- ISA
  - Agreed upon interface between software and hardware
    - SW/compiler assumes, HW promises
  - What the software writer needs to know to write system/user programs
- Microarchitecture
  - Specific implementation of an ISA
  - Not visible to the software
- Microprocessor
  - **ISA, uarch**, circuits
  - "Architecture" = ISA + microarchitecture



ISA vs. Microarchitecture

- What is part of ISA vs. Uarch?
  - Gas pedal: interface for “acceleration”
  - Internals of the engine: implements “acceleration”
  - Add instruction vs. Adder implementation
- Implementation (uarch) can be various as long as it satisfies the specification (ISA)
  - Bit serial, ripple carry, carry lookahead adders
  - x86 ISA has many implementations: 286, 386, 486, Pentium, Pentium Pro, …
- Uarch usually changes faster than ISA
  - Few ISAs (x86, SPARC, MIPS, Alpha) but many uarchs
  - *Why?*

ISA

- Instructions
  - Opcodes, Addressing Modes, Data Types
  - Instruction Types and Formats
  - Registers, Condition Codes
- Memory
  - Address space, Addressability, Alignment
  - Virtual memory management
- Call, Interrupt/Exception Handli
- Access Control, Priority/Privileg
- I/O
- Task Management
- Power and Thermal Management
- Multi-threading support, Multiprocessor support

Intel® 64 and IA-32 Architectures
Software Developer's Manual

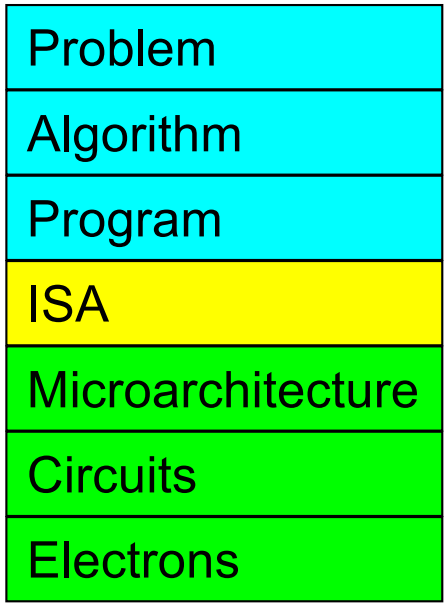
Volume 1:
Basic Architecture

Microarchitecture

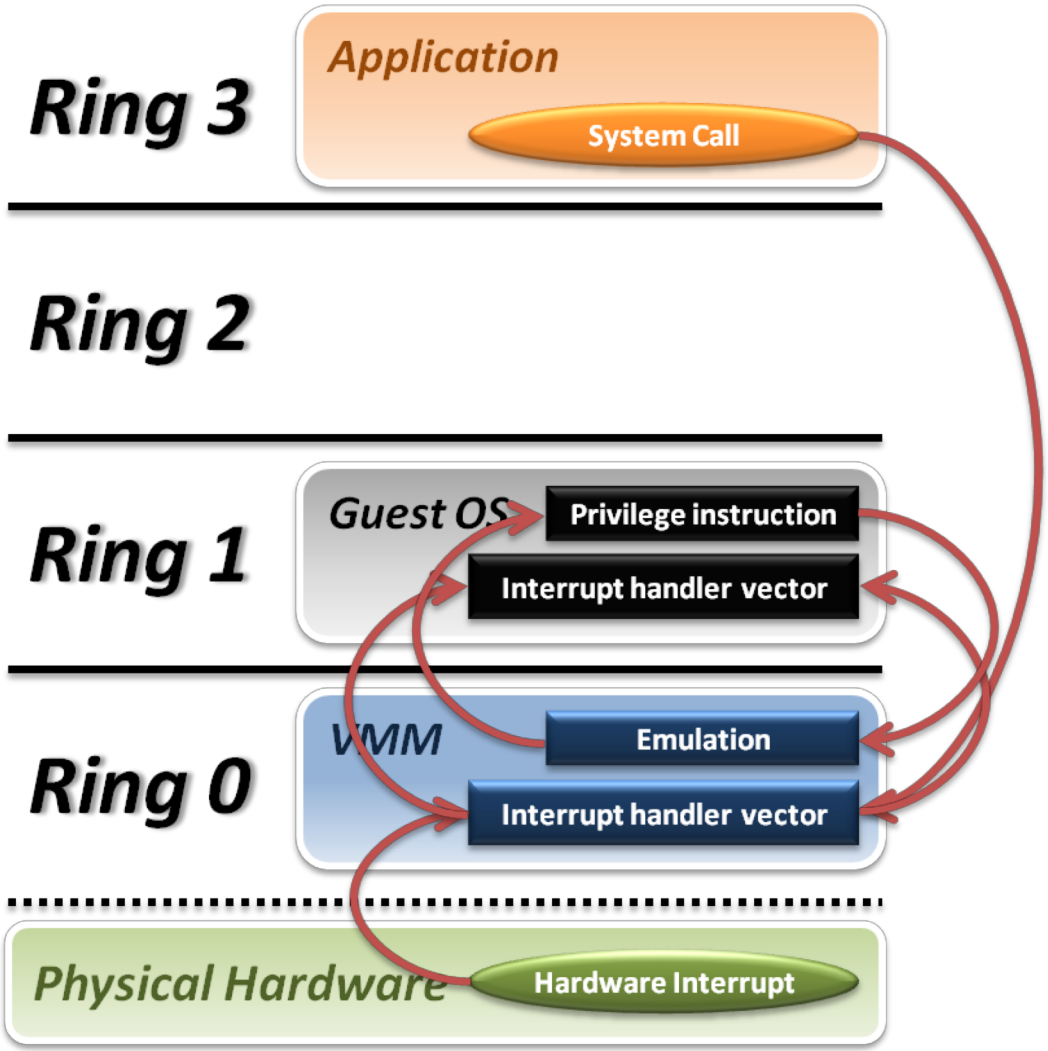
- Implementation of the ISA under specific design constraints and goals
- Anything done in hardware without exposure to software
  - Pipelining
  - In-order versus out-of-order instruction execution
  - Memory access scheduling policy
  - Speculative execution
  - Superscalar processing (multiple instruction issue?)
  - Clock gating
  - Caching? Levels, size, associativity, replacement policy

Design Point

- A set of design considerations and their importance
  - leads to tradeoffs in both ISA and uarch
- Considerations
  - Cost
  - Performance
  - Maximum power consumption
  - Energy consumption (battery life)
  - Availability
  - Reliability and Correctness (or is it?)
  - Time to Market



Example: ISA Security



Tradeoffs: Soul of Computer Architecture

- ISA-level tradeoffs
- Uarch-level tradeoffs
- System and Task-level tradeoffs
  - How to divide the labor between hardware and software

ISA-level Tradeoffs: Semantic Gap

- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) or closer to hardware control signals? → Complex vs. simple instructions
  - RISC vs. CISC vs. HLL machines
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)
  - Tradeoffs:
    - Simple compiler, complex hardware vs. complex compiler, simple hardware
      - Caveat: Translation (indirection) can change the tradeoff!
    - Burden of backward compatibility
    - Performance?
      - Optimization opportunity: Example of VAX INDEX instruction: who (compiler vs. hardware) puts more effort into optimization?
      - Instruction size, code size

X86: Small Semantic Gap: String Operations

REP MOVS DEST SRC

```
IF AddressSize = 16
    THEN
        Use CX for CountReg;
    ELSE IF AddressSize = 64 and REX.W used
        THEN Use RCX for CountReg; FI;
    ELSE
        Use ECX for CountReg;
FI;
WHILE CountReg ≠ 0
    DO
        Service pending interrupts (if any);
        Execute associated string instruction;
        CountReg ← (CountReg – 1);
        IF CountReg = 0
            THEN exit WHILE loop; FI;
        IF (Repeat prefix is REPZ or REPE) and (ZF = 0)
        or (Repeat prefix is REPNZ or REPNE) and (ZF = 1)
            THEN exit WHILE loop; FI;
    OD;
```

```
DEST ← SRC;
IF (Byte move)
    THEN IF DF = 0
        THEN
            (R|E)SI ← (R|E)SI + 1;
            (R|E)DI ← (R|E)DI + 1;
        ELSE
            (R|E)SI ← (R|E)SI – 1;
            (R|E)DI ← (R|E)DI – 1;
        FI;
    ELSE IF (Word move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 2;
            (R|E)DI ← (R|E)DI + 2;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 2;
            (R|E)DI ← (R|E)DI – 2;
        FI;
    ELSE IF (Doubleword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 4;
            (R|E)DI ← (R|E)DI + 4;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 4;
            (R|E)DI ← (R|E)DI – 4;
        FI;
    ELSE IF (Quadword move)
        THEN IF DF = 0
            (R|E)SI ← (R|E)SI + 8;
            (R|E)DI ← (R|E)DI + 8;
            FI;
        ELSE
            (R|E)SI ← (R|E)SI – 8;
            (R|E)DI ← (R|E)DI – 8;
        FI;
FI;
```

How many instructions does this take in MIPS?

Small versus Large Semantic Gap

- CISC vs. RISC
  - Complex instruction set computer → complex instructions
    - Initially motivated by "not good enough" code generation
  - Reduced instruction set computer → simple instructions
    - John Cocke, mid 1970s, IBM 801
      - Goal: enable better compiler control and optimization
- RISC motivated by
  - Memory stalls (no work done in a complex instruction when there is a memory stall?)
    - When is this correct?
  - Simplifying the hardware → lower cost, higher frequency
  - Enabling the compiler to optimize the code better
    - Find fine-grained parallelism to reduce stalls

Small versus Large Semantic Gap

- John Cocke's RISC (large semantic gap) concept:
  - Compiler generates control signals: open microcode

- Advantages of Small Semantic Gap (Complex instructions)
  + Denser encoding → smaller code size → saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + Simpler compiler

- Disadvantages
  - Larger chunks of work → compiler has less opportunity to optimize
  - More complex hardware → translation to control signals and optimization needs to be done by hardware

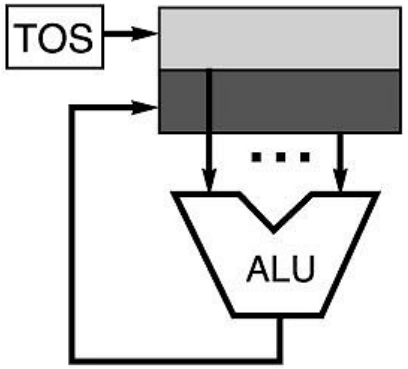
- Read Colwell et al., "Instruction Sets and Beyond: Computers, Complexity, and Controversy," IEEE Computer 1985.

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

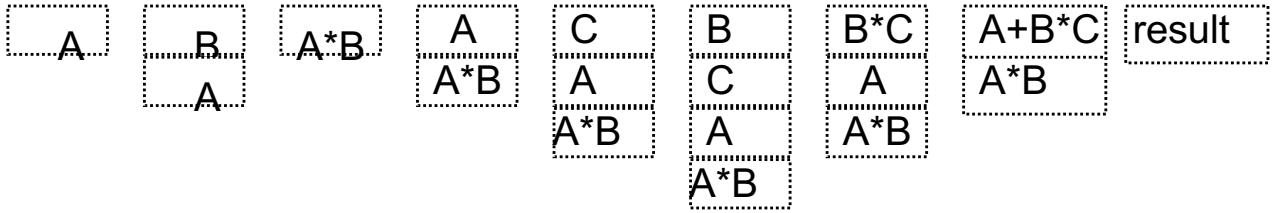
  push C

  push B

  mul

  add

  sub



Stacks: Pros and Cons

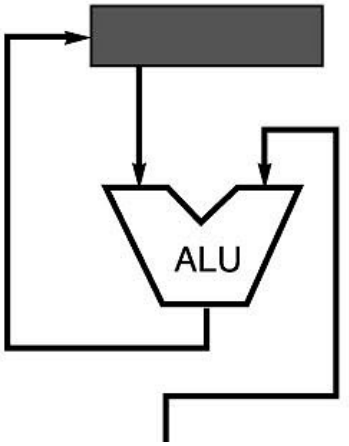
- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures
- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional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

Accumulator Architectures

- Instruction set:

  add A, sub A, mult A, div A, . . .

  load A, store A



acc = acc +,-,*,/ mem[A]

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

  load B

  mul C

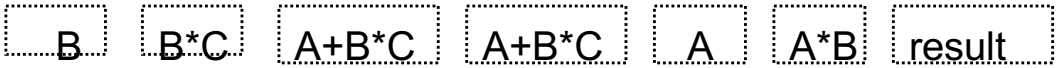
  add A

  store D

  load A

  mul B

  sub D



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

Memory-Memory Architectures

- Instruction set:

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

  (2 operands) add A, B sub A, B mul A, B

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

    mul D, A, B
    mul E, C, B
    add E, A, E
    sub E, D, E

  2 operands

    mov D, A
    mul D, B
    mov E, C
    mul E, B
    add E, A
    sub E, D

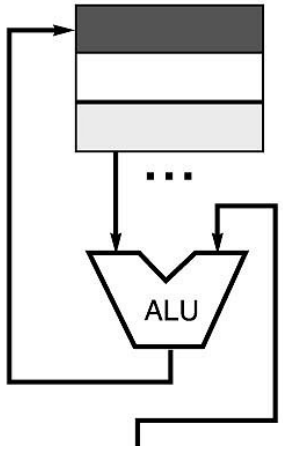
Memory-Memory: Pros and Cons

- 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
  - With two operands, more data movements are required

Register-Memory Architectures

- Instruction set:

```
add R1,  A        sub R1, A        mul R1, B
load R1, A        store R1, A
```



R1 = R1 +,-,*,/ mem[B]

- 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)  */
```

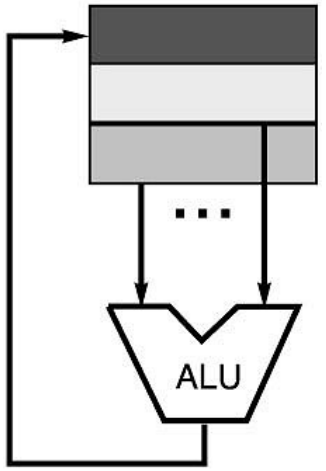
Memory-Register: Pros and Cons

- Pros
  - Some data can be accessed without loading first
  - Instruction format easy to encode
  - Good code density
- Cons
  - Operands are not equivalent (poor orthogonal)
  - 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, &Astore R1, &A move R1, R2



R3 = R1 +,-,*,/ R2

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

```
load R1, &A
load R2, &B
load R3, &C
mul R7, R3, R2          /*    C*B          */
add R8, R7, R1          /*    A + C*B      */
mul R9, R1, R2          /*    A*B          */
sub R10, R9, R8         /*    A*B - (A+C*B) */
```

Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline and make superscalar
- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Instruction Formats

- When we design ISAs we can look at..
  - Byte-Ordering
  - Instruction Length
  - Number of Opcodes

..and their various points and weaknesses..

80x86 Instruction Frequency

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%**

Relative Frequency of Control Instructions

Operation	SPECint92	SPECfp92
Call/Return	13%	11%
Jumps	6%	4%
Branches	81%	87%

• Design hardware to handle branches quickly, since these occur most frequently

Byte-Ordering

- How to store data consisting of multiple bytes on a byte-addressable machine?
  - Little Endian
    - **Least significant** byte stored at **lowest** byte address
  - Big Endian
    - **Most significant** byte stored at **lowest** byte address

Address	Example Address	Value
Base + 0	1001	..
Base + 1	1002	..
Base + 2	1003	..
Base + ..	..	..

Byte-Ordering

- Ex. Represent the String

APPLE

		Address				
		Base + 0	Base + 1	Base + 2	Base + 3	Base + 4
Byte-Order	Little Endian	E	L	P	P	A
	Big Endian	A	P	P	L	E

Byte-Ordering

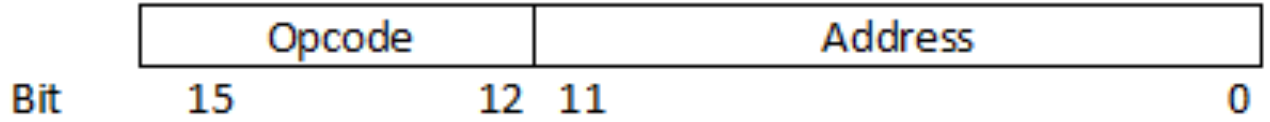
- Little Endian
  - Good for:
    - High-precision arithmetic faster and easier
    - 32 to 16 bit conversion faster (no addition needed)
- Big Endian
  - Good for:
    - Easier to read hex dumps
    - Faster String operations

Byte-Ordering

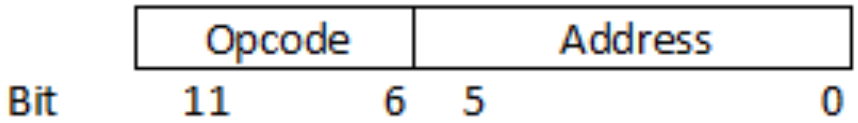
- Examples of **Little Endian**
  - BMP
  - RTF
  - MSPaint
- Examples of **Big Endian**
  - JPEG
  - Adobe Photoshop
  - MacPaint

Instruction Length

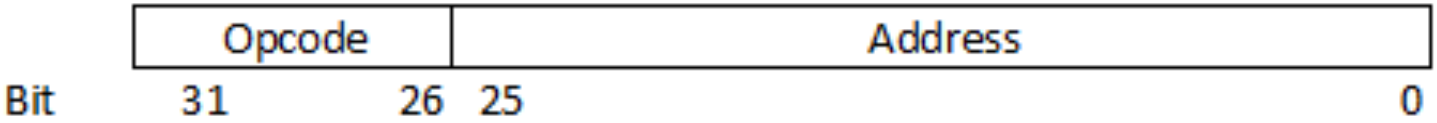
- Fixed Length
  - Ex. MARIE is a fixed length instruction set consisting of 16-bits



- Variable Length
  - 12-bits



  - 36-bits



Instruction length

- Fixed Length
  - Pro: Decodes faster (Not exactly)
    - Less Complexity
  - Con: Wastes Space
    - Opcodes that do not require operands such as MARIE's *halt* makes no use of its address space.
    - Additionally, instructions must be **word aligned**. Creates **gaps** in memory

Instruction Length

- Variable Length
  - Pro: Saves storage space (Not exactly)
    - Instructions take up only as much space as needed
    - Instructions must be **word aligned** in main memory
    - Therefore instructions of varying lengths will create **gaps** in main memory
  - Con: Complex to decode

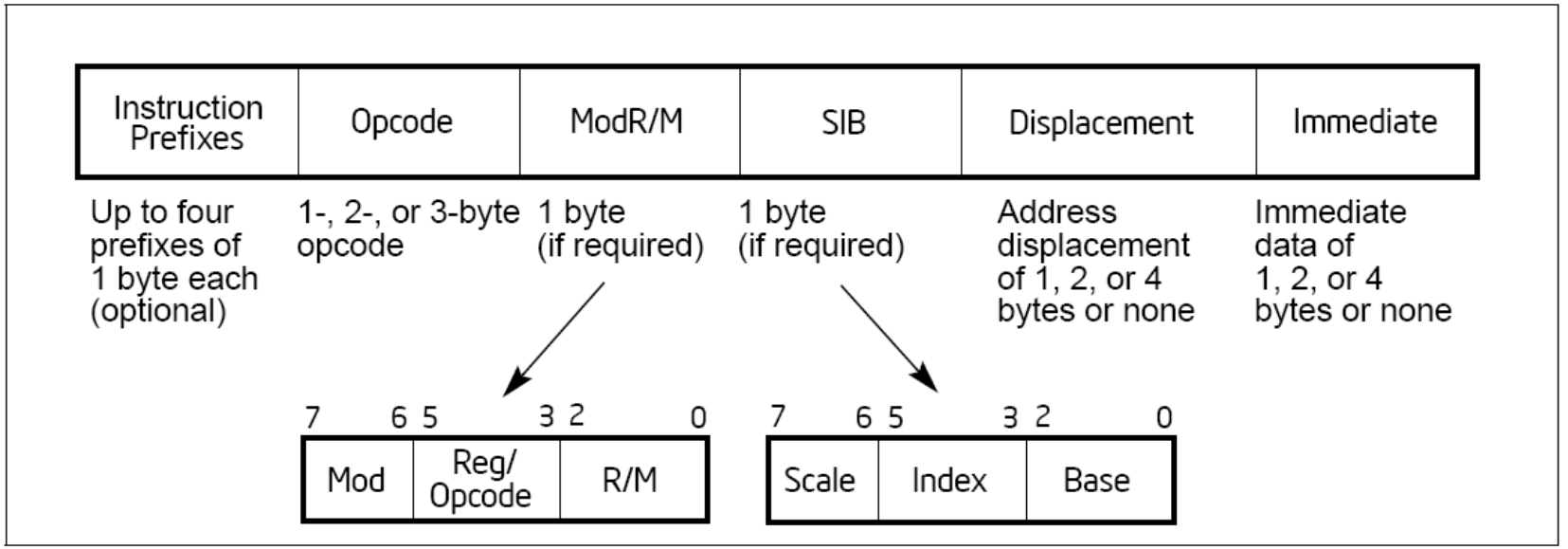
ISA-level Tradeoffs: Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, …
  - Many "RISC" ISAs: Alpha, MIPS, SPARC
  - \+ Easier decode, simpler hardware
  - \+ Enables parallelism: generate target address before knowing the instruction is a branch
  - -- Restricts instruction format (fewer instructions?) or wastes space

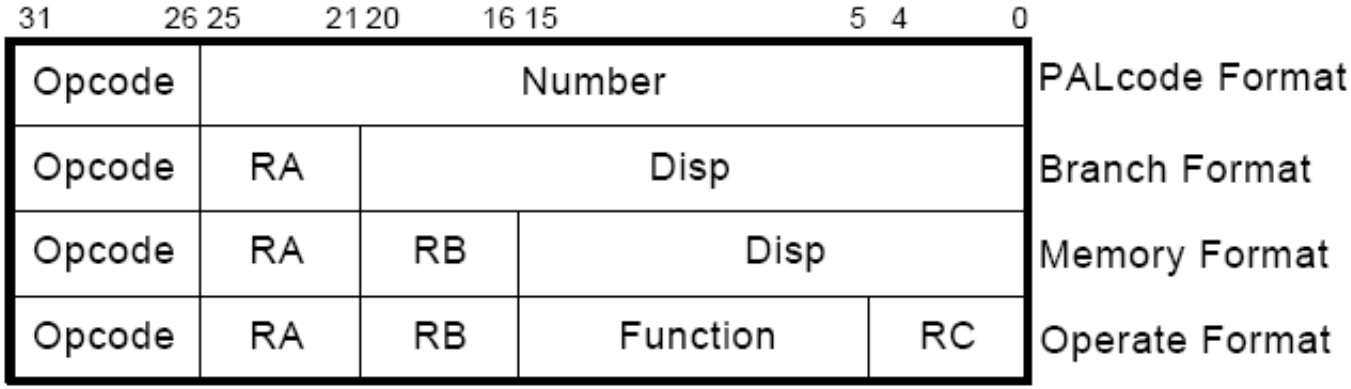
- Non-uniform decode
  - E.g., opcode can be the 1st-7th byte in x86
  - \+ More compact and powerful instruction format
  - -- More complex decode logic (e.g., more adders to speculatively generate branch target)

x86 vs. Alpha Instruction Formats

- x86:



- Alpha



Number of Operands

- Common Instruction Formats
  - OPCODE + 0 addresses
  - OPCODE + 1 (usually a memory address)
  - OPCODE + 2 (registers, or register + memory address)
  - OPCODE + 3 (registers, or combinations of registers and memory)

Number of Operands

- Most common instruction formats use zero, one, two, or three operands
- Ex. Instructions use zero or one operands
  - Zero operand

    Halt
  - One operand

    Add X

Number of Operands

- Example of architectures using three, two, one, zero operands
  - We want to evaluate

$$Z = (X + Y) * (W + U)$$

Number of Operands

- Three operands in **general-purpose register** architecture

Add R1, X, Y	*//R1 = X + Y*
Add R2, W, U	*//R2 = W + U*
Mult Z, R1, R2	*//Z = R1 + R2*

Number of Operands

- **Two** operands in **general-purpose register** architecture

Load R1, X	*//R1 = X*
Add R1, Y	*//R1 += Y*
Load R2, W	*//R2 = W*
Add R2, U	*//R2 += U*
Mult R2, R1	*//R2 *= R1*
Store Z, R2	*//Z = R2*

Number of Operands

- **One** Operand in **accumulator** architecture

Load X	*//AC = X*
Add Y	*//AC += Y*
Store Temp	*//Temp = AC*
Load W	*//AC = W*
Add U	*//AC += U*
Mult Temp	*// AC *= Temp*
Store Z	*//Z = AC*

Number of Operands

- **Zero** Operands in **stack** architecture

Push X	*//X*
Push Y	*//XY*
Add	*//XY+*
Push W	*//XY+W*
Push U	*//XY+WU*
Add	*//XY+WU+*
Mult	*//XY+WU+**
Pop Z	*//Z = XY+WU+**

Number of Operands

- Less operands ->
  - shorter decode time
  - Longer programs
- More operands ->
  - Complex operations requires longer decode
  - Complex operations raises complexity of ISA
  - Shorter programs

Conclusions

- Various trade-offs in the design of ISAs
  - Byte-order
    - Little endian/big endian
  - Instruction length
    - Fixed: decodes faster, but wastes space
    - Variable: slower, not necessarily saves space
  - Operands
    - Less operands -> longer programs + shorter decode + simpler ISA implementation