More ISA

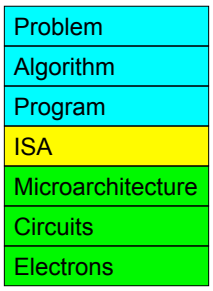
Property of ISA vs. Uarch?

- ADD instruction's opcode
- Number of general purpose registers
- Number of cycles to execute the MUL instruction
- Whether or not the machine employs pipelined instruction execution

- Remember
  - Microarchitecture: Implementation of the ISA under specific design constraints and goals

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
  - Time to Market



- Design point determined by the "Problem" space (application space), or the intended users/*market*

3

Many Different ISAs Over Decades

- x86
- PDP-x: Programmed Data Processor (PDP-11)
- VAX
- IBM 360
- CDC 6600
- SIMD ISAs: CRAY-1, Connection Machine
- VLIW ISAs: Multiflow, Cydrome, IA-64 (EPIC)
- PowerPC, POWER
- RISC ISAs: Alpha, MIPS, SPARC, ARM

- What are the fundamental differences?
  - E.g., how instructions are specified and what they do
  - E.g., how complex are the instructions

4

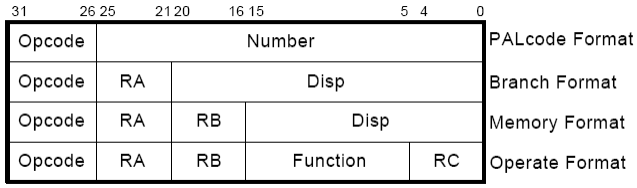
Real World Instruction Sets

Arch	Type	# Oper	# Mem	Data Size	# Regs	Addr Size	Use
Alpha	Reg-Reg	3	0	64-bit	32	64-bit	Workstation
ARM	Reg-Reg	3	0	32/64-bit	16	32/64-bit	Cell Phones, Embedded
MIPS	Reg-Reg	3	0	32/64-bit	32	32/64-bit	Workstation, Embedded
SPARC	Reg-Reg	3	0	32/64-bit	24-32	32/64-bit	Workstation
TI C6000	Reg-Reg	3	0	32-bit	32	32-bit	DSP
IBM 360	Reg-Mem	2	1	32-bit	16	24/31/64	Mainframe
x86	Reg-Mem	2	1	8/16/32/ 64-bit	4/8/24	16/32/64	Personal Computers
VAX	Mem-Mem	3	3	32-bit	16	32-bit	Minicomputer

5

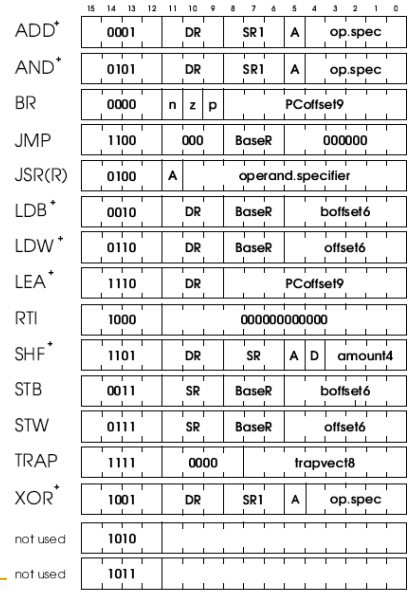
Instruction

- Basic element of the HW/SW interface
- Consists of
  - opcode: what the instruction does
  - operands: who it is to do it to
  - Example from Alpha ISA:



6

Set of Instructions, Encoding, and Spec

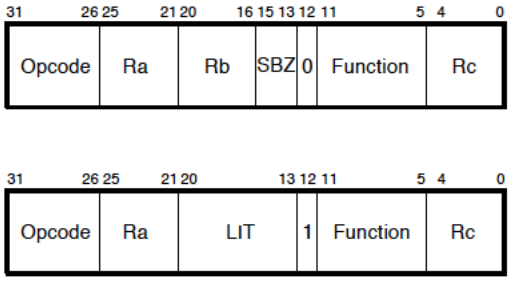


- Example from LC-3b ISA
- x86 Manual
- Aside: concept of "bit steering"
  - A bit in the instruction determines the interpretation of other bits
- Why unused instructions?

7

Bit Steering in Alpha

Figure 3–4: Operate Instruction Format



If bit <12> of the instruction is 0, the Rb field specifies a source register operand.

If bit <12> of the instruction is 1, an 8-bit zero-extended literal constant is formed by bits <20:13> of the instruction. The literal is interpreted as a positive integer between 0 and 255 and is zero-extended to 64 bits. Symbolically, the integer Rbv operand is formed as follows:

8

What Are the Elements of An ISA?

- Instruction processing style
  - Specifies the number of "operands" an instruction "operates" on and how it does so
  - 0, 1, 2, 3 address machines
    - 0-address: stack machine (push A, pop A, op)
    - 1-address: accumulator machine (ld A, st A, op A)
    - 2-address: 2-operand machine (one is both source and dest)
    - 3-address: 3-operand machine (source and dest are separate)
  - Tradeoffs?
    - Larger operate instructions vs. more executed operations
    - Code size vs. execution time vs. on-chip memory space

9

An Example: Stack Machine

+ Small instruction size (no operands needed for operate instructions)
  - Simpler logic
  - Compact code

+ Efficient procedure calls: all parameters on stack
  - No additional cycles for parameter passing

-- Computations that are not easily expressible with "postfix notation" are difficult to map to stack machines
  - Cannot perform operations on many values at the same time (only top N values on the stack at the same time)
  - Not flexible

10

An Example: Stack Machine (II)

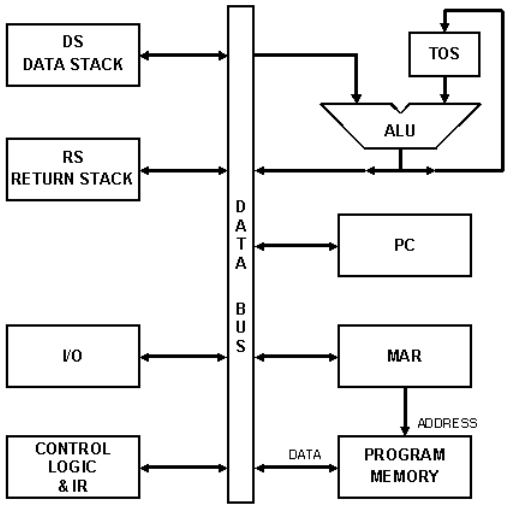


Figure 3.1 -- The canonical stack machine.

Koopman, "Stack Computers: The New Wave," 1989. http://www.ece.cmu.edu/~koopman/stack_computers/sec3_2.html

11

An Example: Stack Machine Operation

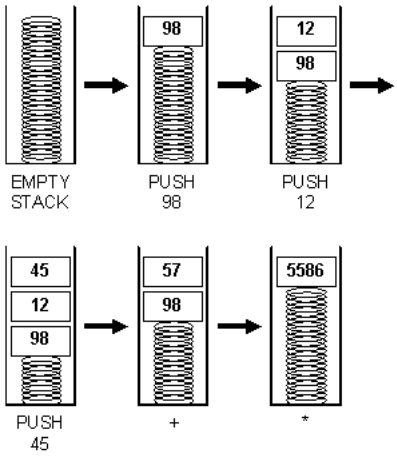


Figure 3.2 -- An example stack machine.

Koopman, "Stack Computers: The New Wave," 1989. http://www.ece.cmu.edu/~koopman/stack_computers/sec3_2.html

12

Other Examples

- PDP-11: A 2-address machine
  - PDP-11 ADD: 4-bit opcode, 2 6-bit operand specifiers
  - Why? Limited bits to specify an instruction
  - Disadvantage: One source operand is always clobbered with the result of the instruction
    - *How do you ensure you preserve the old value of the source?*
- X86: A 2-address (memory/memory) machine
- Alpha: A 3-address (load/store) machine
- MIPS?

13

What Are the Elements of An ISA?

- Instructions
  - Opcode
  - Operand specifiers (addressing modes)
    - How to obtain the operand? Why are there different addressing modes?
- Data types
  - Definition: Representation of information for which there are instructions that operate on the representation
  - Integer, floating point, character, binary, decimal, BCD
  - Doubly linked list, queue, string, bit vector, stack
    - VAX: INSQUEUE and REMQUEUE instructions on a doubly linked list or queue; FINDFIRST
    - Digital Equipment Corp., "VAX11 780 Architecture Handbook," 1977.
    - X86: SCAN opcode operates on character strings; PUSH/POP

14

Data Type Tradeoffs

- What is the benefit of having more or high-level data types in the ISA?
- What is the disadvantage?
- Think compiler/programmer vs. microarchitect
- Concept of semantic gap
  - Data types coupled tightly to the semantic level, or complexity of instructions
- Example: Early RISC architectures vs. Intel 432
  - Early RISC: Only integer data type
  - Intel 432: Object data type, capability based machine

15

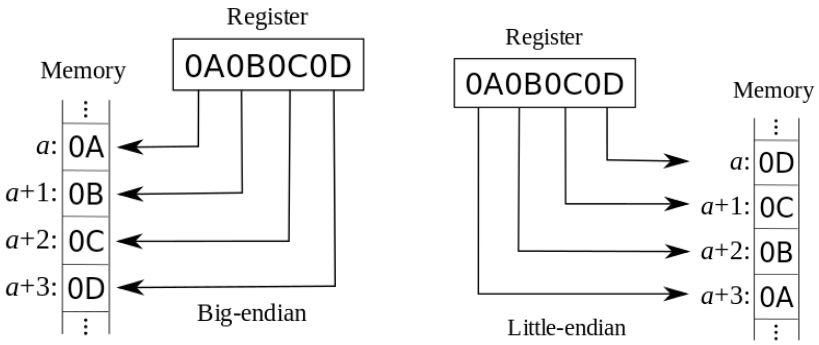
What Are the Elements of An ISA?

- Memory organization
  - Address space: How many uniquely identifiable locations in memory
  - Addressability: How much data does each uniquely identifiable location store
    - Byte addressable: most ISAs, characters are 8 bits
    - Bit addressable: Burroughs 1700. Why?
    - 64-bit addressable: Some supercomputers. Why?
    - 32-bit addressable: First Alpha
    - Food for thought
      - How do you add 2 32-bit numbers with only byte addressability?
      - How do you add 2 8-bit numbers with only 32-bit addressability?
      - Big endian vs. little endian? MSB at low or high byte.
  - Support for virtual memory

16

Big-endian vs. Little-endian

- **Big-endian** systems store the *most significant byte* of a word in the *smallest address*.
- **Little-endian** systems, in contrast, store the *least significant byte* in the *smallest address*.



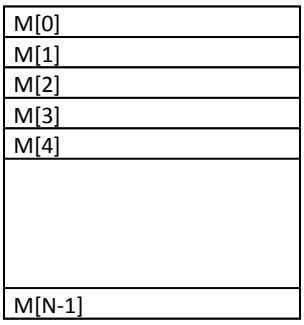
17

What Are the Elements of An ISA?

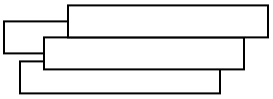
- Registers
  - How many
  - Size of each register
- Why is having registers a good idea?
  - Because programs exhibit a characteristic called data locality
  - A recently produced/accessed value is likely to be used more than once (temporal locality)
    - Storing that value in a register eliminates the need to go to memory each time that value is needed

18

Programmer Visible (Architectural) State



Memory
array of storage locations
indexed by an address



Registers
- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter
memory address
of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

19

Aside: Programmer Invisible State

- Microarchitectural state
- Programmer cannot access this directly

- E.g. cache state
- E.g. pipeline registers

20

Evolution of Register Architecture

- Accumulator
  - a legacy from the "adding" machine days
- Accumulator + address registers
  - need register indirection
  - initially address registers were special-purpose, i.e., can only be loaded with an address for indirection
  - eventually arithmetic on addresses became supported
- General purpose registers (GPR)
  - all registers good for all purposes
  - grew from a few registers to 32 (common for RISC) to 128 in Intel IA-64

21

Instruction Classes

- Operate instructions
  - Process data: arithmetic and logical operations
  - Fetch operands, compute result, store result
  - Implicit sequential control flow
- Data movement instructions
  - Move data between memory, registers, I/O devices
  - Implicit sequential control flow
- Control flow instructions
  - Change the sequence of instructions that are executed

22

What Are the Elements of An ISA?

- Load/store vs. memory/memory architectures
  - Load/store architecture: operate instructions operate only on registers
    - E.g., MIPS, ARM and many RISC ISAs
  - Memory/memory architecture: operate instructions can operate on memory locations
    - E.g., x86, VAX and many CISC ISAs

23

What Are the Elements of An ISA?

- Addressing modes specify how to obtain the operands
  - Absolute LW rt, 10000
    use immediate value as address
  - Register Indirect: LW rt, (r_{base})
    use GPR[r_{base}] as address
  - Displaced or based: LW rt, offset(r_{base})
    use offset+GPR[r_{base}] as address
  - Indexed: LW rt, (r_{base}, r_{index})
    use GPR[r_{base}]+GPR[r_{index}] as address
  - Memory Indirect LW rt ((r_{base}))
    use value at M[GPR[r_{base}]] as address
  - Auto inc/decrement LW Rt, (r_{base})
    use GRP[r_{base}] as address, but inc. or dec. GPR[r_{base}] each time

24

What Are the Benefits of Different Addressing Modes?

- Another example of programmer vs. microarchitect tradeoff
- Advantage of more addressing modes:
  - Enables better mapping of high-level constructs to the machine: some accesses are better expressed with a different mode → reduced number of instructions and code size
    - Think array accesses (autoincrement mode)
    - Think indirection (pointer chasing)
- Disadvantage:
  - More work for the compiler
  - More work for the microarchitect

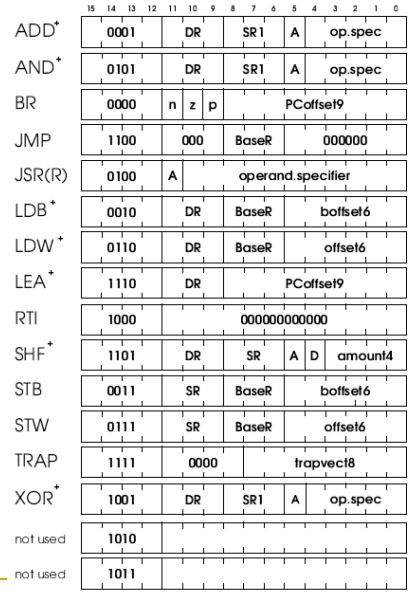
25

ISA Orthogonality

- Orthogonal ISA:
  - All addressing modes can be used with all instruction types
  - Example: VAX
    - (~13 addressing modes) x (>300 opcodes) x (integer and FP formats)
- Who is this good for?
- Who is this bad for?

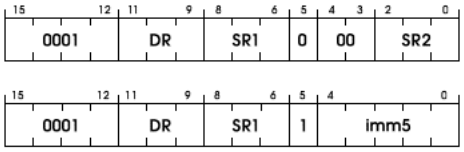
26

Is the LC-3b ISA Orthogonal?



LC-3b: Addressing Modes of ADD

Encodings

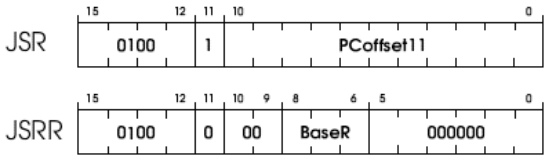


Operation

```
if (bit[5] == 0)
  DR = SR1 + SR2;
else
  DR = SR1 + SEXT(imm5);
setcc();
```

LC-3b: Addressing Modes of of JSR(R)

Encodings



Operation

```
R7 = PC† ;
if (bit[11] == 0)
  PC = BaseR;
else
  PC = PC† + LSHF(SEXT(PCoffset11), 1);
```

Description

First, the incremented PC is saved in R7. This is the linkage back to the calling routine. Then, the PC is loaded with the address of the first instruction of the subroutine, causing an unconditional jump to that address. The address of the subroutine is obtained from the base register (if bit[11] is 0), or the address is computed by sign-extending bits [10:0] to 16 bits, left-shifting the result one bit, and then adding this value to the incremented PC (if bit[11] is 1).

29

Another Question

- Does the LC-3b ISA contain complex instructions?

30

Complex vs. Simple Instructions

- Complex instruction: An instruction does a lot of work, e.g. many operations
  - Insert in a doubly linked list
  - Compute FFT
  - String copy
- Simple instruction: An instruction does small amount of work, it is a primitive using which complex operations can be built
  - Add
  - XOR
  - Multiply

31

Complex vs. Simple Instructions

- Advantages of Complex instructions
  + Denser encoding → smaller code size → better memory utilization, saves off-chip bandwidth, better cache hit rate (better packing of instructions)
  + Simpler compiler: no need to optimize small instructions as much
- Disadvantages of Complex Instructions
  - Larger chunks of work → compiler has less opportunity to optimize (limited in fine-grained optimizations it can do)
  - More complex hardware → translation from a high level to control signals and optimization needs to be done by hardware

32

ISA-level Tradeoffs: Semantic Gap

- Where to place the ISA? Semantic gap
  - Closer to high-level language (HLL) → Small semantic gap, complex instructions
  - Closer to hardware control signals? → Large semantic gap, simple instructions
- RISC vs. CISC machines
  - RISC: Reduced instruction set computer
  - CISC: Complex instruction set computer
    - FFT, QUICKSORT, POLY, FP instructions?
    - VAX INDEX instruction (array access with bounds checking)

33

ISA-level Tradeoffs: Semantic Gap

- Some tradeoffs (for you to think about)
- 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

34

X86: Small Semantic Gap: String Operations

- An instruction operates on a string
  - Move one string of arbitrary length to another location
  - Compare two strings
- Enabled by the ability to specify repeated execution of an instruction (in the ISA)
  - Using a "prefix" called REP prefix
- Example: REP MOVS instruction
  - Only two bytes: REP prefix byte and MOVS opcode byte (F2 A4)
  - Implicit source and destination registers pointing to the two strings (ESI, EDI)
  - Implicit count register (ECX) specifies how long the string is

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 Semantic Gap Examples in VAX

- FIND FIRST
  - Find the first set bit in a bit field
  - Helps OS resource allocation operations
- SAVE CONTEXT, LOAD CONTEXT
  - Special context switching instructions
- INSQUEUE, REMQUEUE
  - Operations on doubly linked list
- INDEX
  - Array access with bounds checking
- STRING Operations
  - Compare strings, find substrings, …
- Cyclic Redundancy Check Instruction
- EDITPC
  - Implements editing functions to display fixed format output
- Digital Equipment Corp., "VAX11 780 Architecture Handbook," 1977-78.

37

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*

38

How High or Low Can You Go?

- Very large semantic gap
  - Each instruction specifies the complete set of control signals in the machine
  - Compiler generates control signals
  - Open microcode (John Cocke, circa 1970s)
    - Gave way to optimizing compilers
- Very small semantic gap
  - ISA is (almost) the same as high-level language
  - Java machines, LISP machines, object-oriented machines, capability-based machines

39

A Note on ISA Evolution

- ISAs have evolved to reflect/satisfy the concerns of the day
- Examples:
  - Limited on-chip and off-chip memory size
  - Limited compiler optimization technology
  - Limited memory bandwidth
  - Need for specialization in important applications (e.g., MMX)
- Use of translation (in HW and SW) enabled underlying implementations to be similar, regardless of the ISA
  - Concept of dynamic/static interface
  - Contrast it with hardware/software interface

40

Effect of Translation

- One can translate from one ISA to another *ISA* to change the semantic gap tradeoffs
- Examples
  - Intel's and AMD's x86 implementations translate x86 instructions into programmer-invisible microoperations (simple instructions) in hardware
  - Transmeta's x86 implementations translated x86 instructions into "secret" VLIW instructions in software (code morphing software)
- Think about the tradeoffs

41

ISA-level Tradeoffs: Instruction Length

- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)
- Variable length: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
    - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently
- Tradeoffs
  - Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ISA extensibility and expressiveness
  - Performance? Smaller code vs. imperfect decode

42

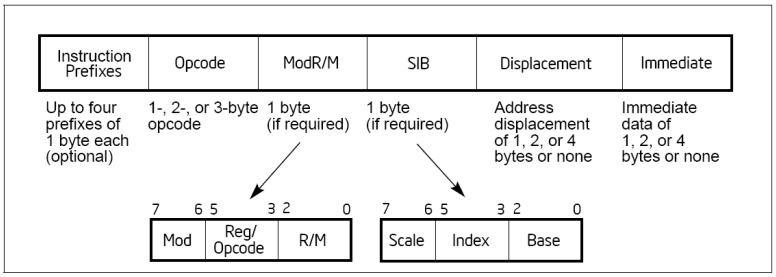
ISA-level Tradeoffs: Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - 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

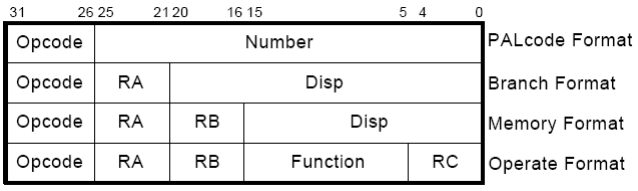
43

x86 vs. Alpha Instruction Formats

- x86:



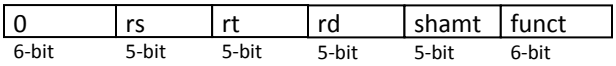
- Alpha:



44

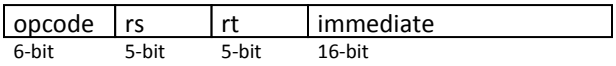
MIPS Instruction Format

- R-type, 3 register operands



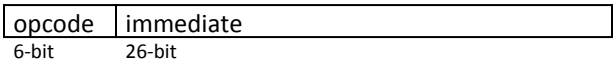
R-type

- I-type, 2 register operands and 16-bit immediate operand



I-type

- J-type, 26-bit immediate operand



J-type

- Simple Decoding
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b'00)
  - format and fields easy to extract in hardware

45

A Note on Length and Uniformity

- Uniform decode usually goes with fixed length
- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths

46

A Note on RISC vs. CISC

- Usually, …
- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes
- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

47

ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file
- Large number of registers:
  - + Enables better register allocation (and optimizations) by compiler → fewer saves/restores
  - -- Larger instruction size
  - -- Larger register file size

48

What Are the Elements of An ISA?

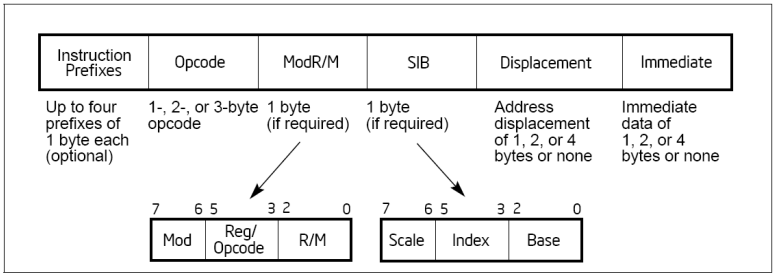
- How to interface with I/O devices
  - Memory mapped I/O
    - A region of memory is mapped to I/O devices
    - I/O operations are loads and stores to those locations
  - Special I/O instructions
    - IN and OUT instructions in x86 deal with ports of the chip
  - Tradeoffs?
    - Which one is more general purpose?

What Are the Elements of An ISA?

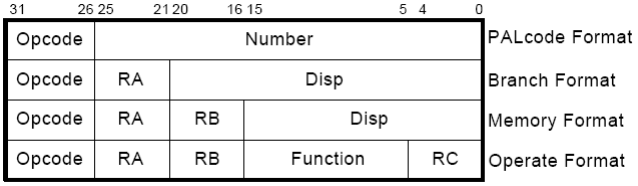
- Privilege modes
  - User vs supervisor
  - Who can execute what instructions?
- Exception and interrupt handling
  - What procedure is followed when something goes wrong with an instruction?
  - What procedure is followed when an external device requests the processor?
  - Vectored vs. non-vectored interrupts (early MIPS)
- Virtual memory
  - Each program has the illusion of the entire memory space, which is greater than physical memory

x86 vs. Alpha Instruction Formats

- x86:



- Alpha:



51

Other Example ISA-level Tradeoffs

- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- …

52

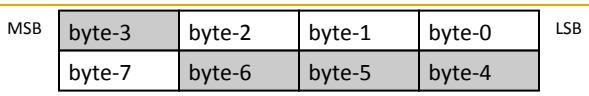
Back to Programmer vs. (Micro)architect

- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

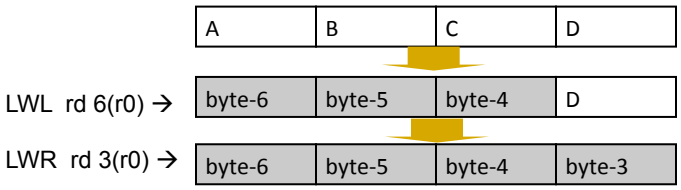
- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
  - Compile/programmer needs to align data

53

MIPS: Aligned Access



- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the "infrequent" case



  - LWL/LWR is slower
  - Note LWL and LWR still fetch within word boundary

54

X86: Unaligned Access

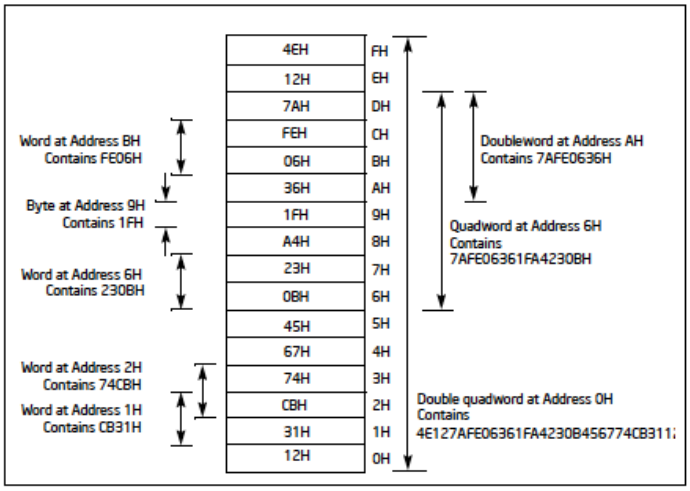


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

55

Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment
- Cons of having no restrictions on alignment
- Filling in the above: an exercise for you…

56

Machine Code

```
00100111101111011111111111100000
10101111101111110000000000010100
10101111101001000000000000100000
10101111101001010000000000100100
10101111101000000000000000011000
10101111101000000000000000011100
10001111101011100000000000011100
10001111101110000000000000011000
00000001110011100000000000011001
00100101110010000000000000000001
00101001000000010000000001100101
10101111101010000000000000011100
00000000000000000001111000000010010
00000011000011111110010000100001
00010100001000001111111111110111
10101111101110010000000000011000
00111100000001000001000000000000
10001111101001010000000000011000
00001100000100000000000011101100
00100100100001000000010000110000
10001111101111110000000000010100
00100111101111010000000000100000
00000011111000000000000000001000
00000000000000000001000000100001
```

MIPS machine language code for a routine to compute and print the sum of the squares of integers between 0 and 100.

57

MIPS

The same routine written in assembly language. However, the code for the routine does not label registers or memory locations nor include comments.

```
sw      $4,  32($29)
sw      $5,  36($29)
sw      $0,  24($29)
sw      $0,  28($29)
lw      $14, 28($29)
lw      $24, 24($29)
multu   $14, $14
addiu   $8,  $14, 1
slti    $1,  $8, 101
sw      $8,  28($29)
mflo    $15
addu    $25, $24, $15
bne     $1,  $0, -9
sw      $25, 24($29)
lui     $4,  4096
lw      $5,  24($29)
jal     1048812
addiu   $4,  $4, 1072
lw      $31, 20($29)
addiu   $29, $29, 32
```

58

MIPS

The same routine written in assembly language with labels, but no comments.

```
        .text
        .align  2
        .globl  main
main:
        subu    $sp, $sp, 32
        sw      $ra, 20($sp)
        sd      $a0, 32($sp)
        sw      $0,  24($sp)
        sw      $0,  28($sp)
loop:
        lw      $t6, 28($sp)
        mul     $t7, $t6, $t6
        lw      $t8, 24($sp)
        addu    $t9, $t8, $t7
        sw      $t9, 24($sp)
        addu    $t0, $t6, 1
        sw      $t0, 28($sp)
        ble     $t0, 100, loop
        la      $a0, str
        lw      $a1, 24($sp)
        jal     printf
        move    $v0, $0
        lw      $ra, 20($sp)
        addu    $sp, $sp, 32
        jr      $ra

        .data
        .align  0

str:    .asciiz "The sum from 0 .. 100 is %
```

59

The routine written in the C programming

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
     int i;
     int sum = 0;

     for (i = 0; i <= 100; i = i + 1) sum = sum + i *
     printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

60

The routine written in Ruby

```
(1..100).inject{|s,n| s + n * n}
```