Lecture 04: ISA Principles

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
## Contents

1. Introduction
2. Classifying Instruction Set Architectures
3. Memory Addressing
4. Type and Size of Operands
5. Operations in the Instruction Set
6. Instructions for Control Flow
7. Encoding an Instruction Set
8. Crosscutting Issues: The Role of Compilers
9. RISC-V ISA

- **Supplement**
  - MIPS ISA
  - RISC vs CISC
  - Compiler compilation stages
  - ISA Historical
    - Appendix L
  - Comparison of ISA
    - Appendix K
1 Introduction

**Instruction Set Architecture** – the portion of the machine visible to the assembly level programmer or to the compiler writer

- To use the hardware of a computer, we must *speak* its language
- The words of a computer language are called *instructions*, and its vocabulary is called an *instruction set*
sum.s for X86

.float sum(int N, float X[], float a) {
    int i;
    float result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}

- http://www.hep.wisc.edu/~pinghc/x86AssmTutorial.htm
- https://en.wikibooks.org/wiki/X86_Assembly/SSE
sum.s for RISC-V

```assembly
.file "sum.s"
.text
.align 2
.globl sum
.type sum, @function

sum:
    add    sp, sp, -48
    sd     s0, 40(sp)
    add    s0, sp, 48
    sw     a0, -36(s0)
    sd     a1, -48(s0)
    fsw    fa2, -40(s0)
    sw     zero, -24(s0)
    sw     zero, -20(s0)
    j      .L2
.L3:
    lw     a5, -20(s0)
    sll    a5, a5, 2
    ld     a4, -48(s0)
    add    a5, a4, a5
    flw    fa4, 0(a5)
    flw    fa5, -40(s0)
    fmul.s fa5, fa4, fa5
    flw    fa4, -24(s0)
    fadd.s fa5, fa4, fa5
    fsw    fa5, -24(s0)
    lw     a5, -20(s0)
    addw   a5, a5, 1
    sw     a5, -20(s0)
.L2:
    lw     a4, -20(s0)
    lw     a5, -36(s0)
    blt    a4, a5,.L3
    flw    fa5, -24(s0)
    fmv.s  fa0, fa5
    ld     s0, 40(sp)
    add    sp, sp, 48
    jr      ra
.size   sum, .-sum
.ident   "GCC: (GNU) 6.1.0"
```

float sum(int N, float X[], float a) {
    int i;
    float result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}

https://riscv.org/

2 Classifying Instruction Set Architectures

| Operand storage in CPU                  | Where are they other than memory |
| # explicit operands named per instruction | How many? Min, Max, Average |
| Addressing mode                        | How the effective address for an operand calculated? Can all use any mode? |
| Operations                             | What are the options for the opcode? |
| Type & size of operands                 | How is typing done? How is the size specified? |

*These choices critically affect number of instructions, CPI, and CPU cycle time*
ISA Classification

• Most basic differentiation: internal storage in a processor
  – Operands may be named explicitly or implicitly
• Major choices:
  1. In an **accumulator architecture** one operand is *implicitly* the accumulator => similar to calculator
  2. The operands in a **stack architecture** are *implicitly* on the top of the stack
  3. The **general-purpose register architectures** have only *explicit* operands – either registers or memory location
Register Machines

- How many registers are sufficient?
- General-purpose registers vs. special-purpose registers
  - *compiler flexibility and hand-optimization*
- Two major concerns for arithmetic and logical instructions (ALU)
  1. Two or three operands
     - \( X + Y \Rightarrow X \)
     - \( X + Y \Rightarrow Z \)
  2. How many of the operands may be memory addresses (0 – 3)

<table>
<thead>
<tr>
<th>Number of memory addresses</th>
<th>Maximum number of operands allowed</th>
<th>Type of Architecture</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>Load-Store</td>
<td>Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Register-Memory</td>
<td>IBM 360/370, Intel 80x86, Motorola 68000, TI TMS32C54x</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Memory – memory</td>
<td>VAX (also has 3 operand formats)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Memory - memory</td>
<td>VAX (also has 2 operand formats)</td>
</tr>
</tbody>
</table>

Hence, register architecture classification (# mem, # operands)
(0, 3): Register-Register

- ALU is Register to Register – also known as pure Reduced Instruction Set Computer (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
  - except for memory instructions of course
  - but there are only 2 of them - load and store

**Disadvantages**
- instruction count tends to be higher
- some instructions are short - wasting instruction word bits
(1, 2): Register-Memory

Evolved RISC and also old CISC
• new RISC machines capable of doing speculative loads
• predicated and/or deferred loads are also possible

Advantages
– data access to ALU immediate without loading first
– instruction format is relatively simple to encode
– code 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
  • if memory target is in L0 cache then not so bad
  • if not - life gets miserable
(2, 2) or (3, 3): Memory-Memory

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 code
- doesn’t waste registers for temporary values
  - good idea for use once data - e.g. streaming media

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
  - register file reduces memory accesses if reused

Not used today
Summary: Tradeoffs for the ISA Classes

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-register (0,3)</td>
<td>Simple, fixed length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute.</td>
<td>Higher instruction count than architectures with memory references in the instructions. More instructions and lower instruction density leads to larger programs.</td>
</tr>
<tr>
<td>Register-memory (1,2)</td>
<td>Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density</td>
<td>Operands are not equivalent since a source operand is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.</td>
</tr>
<tr>
<td>Memory-memory (2,2) or (3,3)</td>
<td>Most compact. Does not waste registers for temporaries.</td>
<td>Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today)</td>
</tr>
</tbody>
</table>
3 Memory Addressing

• Objects have **byte addresses**
  – the number of bytes counted from the beginning of memory

• **Object Length:**
  – bytes (8 bits), half words (16 bits),
  – words (32 bits), and double words (64 bits).
  – The type is implied in opcode, e.g.,
    • LDB – load byte
    • LDW – load word, etc

• **Byte Ordering**
  – Little Endian: puts the byte whose address is xx00 at the least significant position in the word. (7,6,5,4,3,2,1,0)
  – Big Endian: puts the byte whose address is xx00 at the most significant position in the word. (0,1,2,3,4,5,6,7)
  • Problem occurs when exchanging data among machines with different orderings
Interpreting Memory Addresses

• Alignment Issues
  – Accesses to objects larger than a byte must be aligned. An access to an object of size $s$ bytes at byte address $A$ is aligned if $A \mod s = 0$.
  – Misalignment causes hardware complications, since the memory is typically aligned on a word or a double-word boundary
  – Misalignment typically results in an alignment fault that must be handled by the OS

• Hence
  – byte address is anything - never misaligned
  – half word - even addresses - low order address bit $= 0$ (XXXXXXX0) else trap
  – word - low order 2 address bits $= 0$ (XXXXXX00) else trap
  – double word - low order 3 address bits $= 0$ (XXXXX000) else trap
### Aligned/Misaligned Addresses

<table>
<thead>
<tr>
<th>Width of object</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte (byte)</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>2 bytes (half word)</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>2 bytes (half word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
<td>Aligned</td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
<tr>
<td>4 bytes (word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>Aligned</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
<tr>
<td>8 bytes (double word)</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
<td>Misaligned</td>
</tr>
</tbody>
</table>

**Figure A.5** Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.
Addressing Modes

• How architecture specify the effective address of an object?
  – **Effective address**: the actual memory address specified by the addressing mode.
  – E.g. Mem[R[R1]] refers to the contents of the memory location whose location is given the contents of register 1 (R1).

• Addressing Modes:
  – Register.
  – Immediate
  – Displacement
  – Register indirect,.......
<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
<th>When used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct or absolute</td>
<td>Add R1,(1001)</td>
<td>Regs[R1] ← Regs[R1] + Mem[1001]</td>
<td>Sometimes useful for accessing static data; address constant may need to be large.</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Add R1,@(R3)</td>
<td>Regs[R1] ← Regs[R1] + Mem[Mem[Regs[R3]]]</td>
<td>If R3 is the address of a pointer p, then mode yields *p.</td>
</tr>
<tr>
<td>Autoincrement</td>
<td>Add R1,(R2)+</td>
<td>Regs[R1] ← Regs[R1] + Mem[Regs[R2]]</td>
<td>Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>Add R1, -(R2)</td>
<td>Regs[R2] ← Regs[R2] - d</td>
<td>Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.</td>
</tr>
<tr>
<td>Scaled</td>
<td>Add R1,100(R2)[R3]</td>
<td>Regs[R1] ← Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]</td>
<td>Used to index arrays. May be applied to any indexed addressing mode in some computers.</td>
</tr>
</tbody>
</table>

**Figure A.6** Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable d designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A-36.
Addressing Mode Impacts

- Instruction counts
- Architecture Complexity
- CPI
Summary of use of memory addressing modes

Figure A.7  Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see Section A.8. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost
Displacement values are widely distributed

Add $R4, 100(R1)$  
$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$

Accessing local variables (+ simulates register indirect, direct addressing modes).

---

Impact instruction length

---

![Graph showing distribution of displacement values](image-url)
Displacement Addressing Mode

• Benchmarks show
  – 12 bits of displacement would capture about 75% of the full 32-bit displacements
  – 16 bits should capture about 99%

• Remember:
  – optimize for the common case. Hence, the choice is at least 12-16 bits

• For addresses that do fit in displacement size:
  
  Add R4, 10000 (R0)

• For addresses that don’t fit in displacement size, the compiler must do the following:
  
  Load R1, 1000000
  Add R1, R0
  Add R4, 0 (R1)
Immediate Addressing Mode

• Used where we want to get to a numerical value in an instruction

• Around 25% of the operations have an immediate operand

At high level:

\[
a = b + 3;
\]
\[
\text{if ( } a > 17 \text{ )}
\]
\[
\text{goto Addr}
\]

At Assembler level:

\[
\text{Load R2, #3}
\]
\[
\text{Add R0, R1, R2}
\]
\[
\text{Load R2, #17}
\]
\[
\text{CMPBGT R1, R2}
\]
\[
\text{Load R1, Address (R1)}
\]
About 25% of data transfer and ALU operations have an immediate operand.

Add R4, #3


For constants.

Impact instruction length

**Figure A.9** About one-quarter of data transfers and ALU operations have an immediate operand. The bottom bars show that integer programs use immediates in about one-fifth of the instructions, while floating-point programs use immediates in about...
Number of bits for immediate

- 16 bits would capture about 80% and 8 bits about 50%.

Impact instruction length

Add R4, #3


For constant
Summary: Memory Addressing

- A new architecture expected to support at least: displacement, immediate, and register indirect
  - represent 75% to 99% of the addressing modes
- The size of the address for displacement mode to be at least 12-16 bits
  - capture 75% to 99% of the displacements
- The size of the immediate field to be at least 8-16 bits
  - capture 50% to 80% of the immediates

Processors rely on compilers to generate codes using those addressing modes
How is the type of an operand designated?

• The type of the operand is usually encoded in the *opcode*
  – e.g., LDB – load byte; LDW – load word

• Common operand types: (imply their sizes)
  - Character (8 bits or 1 byte)
  - Half word (16 bits or 2 bytes)
  - Word (32 bits or 4 bytes)
  - Double word (64 bits or 8 bytes)
  - Single precision floating point (4 bytes or 1 word)
  - Double precision floating point (8 bytes or 2 words)
    - Characters are almost always in ASCII
    - 16-bit Unicode (used in Java) is gaining popularity
    - Integers are two’s complement binary
    - Floating points follow the IEEE standard 754

• Some architectures support *packed decimal*: 4 bits are used to encode the values 0-9; 2 decimal digits are packed into each byte
Figure A.11 Distribution of data accesses by size for the benchmark programs. The double-word data type is used for double-precision floating point in floating-point programs and for addresses, since the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.
Summary: Type and Size of operands

• 32-architecture supports 8-, 16-, and 32-bit integers, 32-bit and 64-bit IEEE 754 floating-point data.

• A new 64-bit address architecture supports 64-bit integers

• Media processor and DSPs need wider accumulating registers for accuracy.
### 5 Operations in the Instruction Set

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic and logical</td>
<td>Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Loads-stores (move instructions on computers with memory addressing)</td>
</tr>
<tr>
<td>Control</td>
<td>Branch, jump, procedure call and return, traps</td>
</tr>
<tr>
<td>System</td>
<td>Operating system call, virtual memory management instructions</td>
</tr>
<tr>
<td>Floating point</td>
<td>Floating-point operations: add, multiply, divide, compare</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, decimal multiply, decimal-to-character conversions</td>
</tr>
<tr>
<td>String</td>
<td>String move, string compare, string search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Pixel and vertex operations, compression/decompression operations</td>
</tr>
</tbody>
</table>

**Figure A.12** Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any.
### Top 10 instructions for the 80x86

<table>
<thead>
<tr>
<th>Rank</th>
<th>80x86 instruction</th>
<th>Integer average (% total executed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>22%</td>
</tr>
<tr>
<td>2</td>
<td>conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>3</td>
<td>compare</td>
<td>16%</td>
</tr>
<tr>
<td>4</td>
<td>store</td>
<td>12%</td>
</tr>
<tr>
<td>5</td>
<td>add</td>
<td>8%</td>
</tr>
<tr>
<td>6</td>
<td>and</td>
<td>6%</td>
</tr>
<tr>
<td>7</td>
<td>sub</td>
<td>5%</td>
</tr>
<tr>
<td>8</td>
<td>move register-register</td>
<td>4%</td>
</tr>
<tr>
<td>9</td>
<td>call</td>
<td>1%</td>
</tr>
<tr>
<td>10</td>
<td>return</td>
<td>1%</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td><strong>96%</strong></td>
</tr>
</tbody>
</table>

**Figure A.13** The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.
6 Instructions for Control Flow

• Control instructions change the flow of control:
  – instead of executing the next instruction, the program branches to the address specified in the branching instructions

• They break the pipeline
  – Difficult to optimize out
  – AND they are frequent

• Four types of control instructions
  – Conditional branches
  – Jumps – unconditional transfer
  – Procedure calls
  – Procedure returns
Breakdown of control flow instructions

- Conditional branches
- Jumps – unconditional transfer
- Procedure calls
- Procedure returns

**Issues:**
- Where is the target address? How to specify it?
- Where is return address kept? How are the arguments passed? (calls)
- Where is return address? How are the results passed? (returns)
Addressing Modes for Control Flow Instructions

- PC-relative (Program Counter)
  - supply a displacement added to the PC
    - Known at compile time for jumps, branches, and calls (specified within the instruction)
  - the target is often near the current instruction
    - requiring fewer bits
    - independently of where it is loaded (position independence)

- Register indirect addressing – dynamic addressing
  - The target address may not be known at compile time
  - Naming a register that contains the target address
    - Case or switch statements
    - Virtual functions or methods in C++ or Java
    - High-order functions or function pointers in C or C++
    - Dynamically shared libraries
Branch distances

Figure A.15 Branch distances in terms of number of instructions between the target and the branch instruction. The most frequent branches in the integer programs are to targets that can be encoded in 4 to 8 bits. This result tells us that short displacement fields often suffice for branches and that the designer can gain some encoding density by having a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store computer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer instructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits needed for the displacement may increase if the computer has variable-length instructions to be aligned on any byte boundary. The programs and computer used to collect these statistics are the same as those in Figure A.8.
## Conditional Branch Options

<table>
<thead>
<tr>
<th>Name</th>
<th>Examples</th>
<th>How condition is tested</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition code (CC)</td>
<td>80x86, ARM, PowerPC, SPARC, SuperH</td>
<td>Tests special bits set by ALU operations, possibly under program control.</td>
<td>Sometimes condition is set for free.</td>
<td>CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.</td>
</tr>
<tr>
<td>Condition register</td>
<td>Alpha, MIPS</td>
<td>Tests arbitrary register with the result of a comparison.</td>
<td>Simple.</td>
<td>Uses up a register.</td>
</tr>
<tr>
<td>Compare and branch</td>
<td>PA-RISC, VAX</td>
<td>Compare is part of the branch. Often compare is limited to subset.</td>
<td>One instruction rather than two for a branch.</td>
<td>May be too much work per instruction for pipelined execution.</td>
</tr>
</tbody>
</table>

**Figure A.16** The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.
Comparison Type vs. Frequency

- Most loops go from 0 to n.
- Most backward branches are loops – taken about 90%
Procedure Invocation Options

• Procedure calls and returns
  – control transfer
  – state saving; the return address must be saved
  Newer architectures require the compiler to generate stores and loads for each register saved and restored

• Two basic conventions in use to save registers
  – caller saving: the calling procedure must save the registers that it wants preserved for access after the call
    • the called procedure need not worry about registers
  – callee saving: the called procedure must save the registers it wants to use
    • leaving the caller unrestrained

most real systems today use a combination of both
  • Application binary interface (ABI) that set down the basic rules as to which register be caller saved and which should be callee saved
7. Encoding an Instruction Set

• Opcode: specifying the operation
• # of operand
  – addressing mode
  – address specifier: tells what addressing mode is used
  – Load-store computer
    • Only one memory operand
    • Only one or two addressing modes

• The architecture must balancing several competing forces when encoding the instruction set:
  – # of registers && Addressing modes
  – Size of registers && Addressing mode fields; Average instruction size && Average program size.
  – Easy to handle in pipeline implementation.
x86 vs. Alpha Instruction Formats

• **x86:**

<table>
<thead>
<tr>
<th>Instruction Prefixes</th>
<th>Opcode</th>
<th>ModR/M</th>
<th>SIB</th>
<th>Displacement</th>
<th>Immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-, 2-, or 3-byte opcode</td>
<td>1 byte (if required)</td>
<td>1 byte (if required)</td>
<td>Address displacement of 1, 2, or 4 bytes or none</td>
<td>Immediate data of 1, 2, or 4 bytes or none</td>
</tr>
</tbody>
</table>

- **Mod**: 7-6
- **Reg/Opc**: 5-2
- **R/M**: 1-0

• **Alpha**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Number</th>
<th>PALcode Format</th>
<th>Branch Format</th>
<th>Memory Format</th>
<th>Operate Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opcode</td>
<td>RA</td>
<td>Disp</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>RA</td>
<td>RB</td>
<td>Disp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opcode</td>
<td>RA</td>
<td>RB</td>
<td>Function</td>
<td>RC</td>
<td></td>
</tr>
</tbody>
</table>
Instruction Length Tradeoffs

• **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
Uniform vs Non-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
Three basic variation instruction encoding: variable length, fixed length, and hybrid

<table>
<thead>
<tr>
<th>Operation and no. of operands</th>
<th>Address specifier</th>
<th>Address field 1</th>
<th>Address specifier</th>
<th>Address field n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Variable (e.g., Intel 80x86, VAX)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address field 1</th>
<th>Address field 2</th>
<th>Address field 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier 1</th>
<th>Address specifier 2</th>
<th>Address field</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Address specifier</th>
<th>Address field 1</th>
<th>Address field 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The length of 80x86 (CISC) instructions varies between 1 and 17 bytes. The length of most RISC ISA instructions are 4 bytes. X86 program are generally smaller than RISC ISA.

To reduce RISC code size

---

**Figure A.18** Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with address specifier determining the addressing mode and the length of the specifier that operand. It generally enables the smallest code representation, since
Reduced Code Size in RISCs

• Hybrid encoding – support 16-bit and 32-bit instructions in RISC, eg. ARM Thumb, MIPS 16 and RISC-V
  – narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and two-address format rather than the classic three-address format
  – claim a code size reduction of up to 40%

• Compression in IBM’s CodePack
  – Adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss
  – The instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk
  – Claim code reduction 35% - 40%
  – PowerPC create a Hash table in memory that map between compressed and uncompressed address. Code size 35%~40%

• Hitachi’s SuperH: fixed 16-bit format
  – 16 rather than 32 registers
  – fewer instructions
Summary of Instruction Encoding

• Three choices
  – Variable, fixed and hybrid
  – Note the differences of hybrid and variable

• Choices of instruction encoding is a tradeoff between
  – For performance: fixed encoding
  – For code size: variable encoding

• How hybrid encoding is used in RISC to reduce code size
  – 16bit and 32bit

• In general, we see:
  – RISC: fixed or hybrid
  – CISC: variable
8 The Role of Compilers

• Almost all programming is done in high-level languages.
  – An ISA is essentially a compiler target.

• See backup slides for the compilation stage by most compiler, e.g. gcc

• Compiler goals:
  – All correct programs execute correctly
  – Most compiled programs execute fast (optimizations)
  – Fast compilation
  – Debugging support
Figure A.19 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A pass is simply one phase in which the compiler reads and transforms the entire program. (The term phase is often used interchangeably with pass.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.
Optimization Types

• High level – done at or near source code level
  – If procedure is called only once, put it in-line and save CALL
  – more general case: if call-count < some threshold, put them in-line

• Local – done within straight-line code
  – common sub-expressions produce same value – either allocate a register or replace with single copy
  – constant propagation – replace constant valued variable with the constant
  – stack height reduction – re-arrange expression tree to minimize temporary storage needs

• Global – across a branch
  – copy propagation – replace all instances of a variable A that has been assigned X (i.e., A=X) with X.
  – code motion – remove code from a loop that computes same value each iteration of the loop and put it before the loop
  – simplify or eliminate array addressing calculations in loops
Optimization Types

• Machine-dependent optimizations – based on machine knowledge
  – strength reduction – replace multiply by a constant with shifts and adds
    • would make sense if there was no hardware support for MUL
    • a trickier version: $17 \times = \text{arithmetic left shift 4 and add}$

• pipelining scheduling – reorder instructions to improve pipeline performance
  – dependency analysis
  – branch offset optimization - reorder code to minimize branch offsets
## Major types of optimizations

<table>
<thead>
<tr>
<th>Optimization name</th>
<th>Explanation</th>
<th>Percentage of the total number of optimizing transforms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-level</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure integration</td>
<td>Replace procedure call by procedure body</td>
<td>N.M.</td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common subexpression elimination</td>
<td>Replace two instances of the same computation by single copy</td>
<td>18%</td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Replace all instances of a variable that is assigned a constant with the constant</td>
<td>22%</td>
</tr>
<tr>
<td>Stack height reduction</td>
<td>Rearrange expression tree to minimize resources needed for expression evaluation</td>
<td>N.M.</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global common subexpression elimination</td>
<td>Same as local, but this version crosses branches</td>
<td>13%</td>
</tr>
<tr>
<td>Copy propagation</td>
<td>Replace all instances of a variable A that has been assigned X (i.e., A = X) with X</td>
<td>11%</td>
</tr>
<tr>
<td>Code motion</td>
<td>Remove code from a loop that computes same value each iteration of the loop</td>
<td>16%</td>
</tr>
<tr>
<td>Induction variable elimination</td>
<td>Simplify/eliminate array addressing calculations within loops</td>
<td>2%</td>
</tr>
<tr>
<td><strong>Processor-dependent</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strength reduction</td>
<td>Many examples, such as replace multiply by a constant with adds and shifts</td>
<td>N.M.</td>
</tr>
<tr>
<td>Pipeline scheduling</td>
<td>Reorder instructions to improve pipeline performance</td>
<td>N.M.</td>
</tr>
<tr>
<td>Branch offset optimization</td>
<td>Choose the shortest branch displacement that reaches target</td>
<td>N.M.</td>
</tr>
</tbody>
</table>

Figure A.20 Major types of optimizations and examples in each class. These data tell us about the relative frequency of occurrence of various optimizations. The third column lists the static frequency with which some of the
Complier Optimizations – Change in IC

Figure A.21 Change in instruction count for the programs lucas and mcf from the SPEC2000 as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (software pipelining), and global register allocation. Level 3 adds procedure integration. These experiments were performed on Alpha compilers.
Compiler Based Register Optimization

• Compiler assumes small number of registers (16-32)
  – Optimizing use is up to compiler
  – HLL programs have no explicit references to registers
  – usually – is this always true?

• Compiler Approach
  – Assign symbolic or virtual register to each candidate variable
  – Map (unlimited) symbolic registers to real registers
  – Symbolic registers that do not overlap can share real registers
  – If you run out of real registers some variables
    • Spilling
Graph Coloring

• Given a graph of nodes and edges
  – Assign a color to each node
  – Adjacent nodes have different colors
  – Use minimum number of colors
  
    https://en.wikipedia.org/wiki/Graph_coloring

• Registration allocation
  – Nodes are symbolic registers
  – Two registers that are live in the same program fragment are joined by an edge
  – Try to color the graph with $n$ colors, where $n$ is the number of real registers
  – Nodes that can not be colored are placed in memory
Iron-code Summary

• Section A.2—Use general-purpose registers with a load-store architecture.
• Section A.3—Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register indirect.
• Section A.4—Support these data sizes and types: 8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.
  – Now we see 16-bit FP for deep learning in GPU
• Section A.5—Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and shift.
• Section A.6—Compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
• Section A.7—Use fixed instruction encoding if interested in performance, and use variable instruction encoding if interested in code size.
• Section A.8—Provide at least 16 general-purpose registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist IS
  – Often use separate floating-point registers.
  – The justification is to increase the total number of registers without raising problems in the instruction format or in the speed of the general-purpose register file. This compromise, however, is not orthogonal.
# Real World ISA

<table>
<thead>
<tr>
<th>Arch</th>
<th>Type</th>
<th># Oper</th>
<th># Mem</th>
<th>Data Size</th>
<th># Regs</th>
<th>Addr Size</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>Reg-Reg</td>
<td>3</td>
<td>0</td>
<td>64-bit</td>
<td>32</td>
<td>64-bit</td>
<td>Workstation</td>
</tr>
<tr>
<td>ARM</td>
<td>Reg-Reg</td>
<td>3</td>
<td>0</td>
<td>32/64-bit</td>
<td>16</td>
<td>32/64-bit</td>
<td>Cell Phones, Embedded</td>
</tr>
<tr>
<td>MIPS</td>
<td>Reg-Reg</td>
<td>3</td>
<td>0</td>
<td>32/64-bit</td>
<td>32</td>
<td>32/64-bit</td>
<td>Workstation, Embedded</td>
</tr>
<tr>
<td>SPARC</td>
<td>Reg-Reg</td>
<td>3</td>
<td>0</td>
<td>32/64-bit</td>
<td>24-32</td>
<td>32/64-bit</td>
<td>Workstation</td>
</tr>
<tr>
<td>TI C6000</td>
<td>Reg-Reg</td>
<td>3</td>
<td>0</td>
<td>32-bit</td>
<td>32</td>
<td>32-bit</td>
<td>DSP</td>
</tr>
<tr>
<td>IBM 360</td>
<td>Reg-Mem</td>
<td>2</td>
<td>1</td>
<td>32-bit</td>
<td>16</td>
<td>24/31/64</td>
<td>Mainframe</td>
</tr>
<tr>
<td>x86</td>
<td>Reg-Mem</td>
<td>2</td>
<td>1</td>
<td>8/16/32/64-bit</td>
<td>4/8/24</td>
<td>16/32/64</td>
<td>Personal Computers</td>
</tr>
<tr>
<td>VAX</td>
<td>Mem-Mem</td>
<td>3</td>
<td>3</td>
<td>32-bit</td>
<td>16</td>
<td>32-bit</td>
<td>Minicomputer</td>
</tr>
</tbody>
</table>
The details is to trade-off