Instruction Set Architecture (ISA)

- ISA provides the level of abstraction for both the hardware and software
- Interface between the software that runs on a computer and the hardware that executes it
Instruction Set Architecture (ISA)

- **Functional definition** of operations, modes, and storage locations supported by hardware
- **Precise description** of the ways to invoke and access them

- No guarantees regarding
  - How operations are implemented
  - Which operations are fast, which are slow, and when
  - Which operations take more power

In early days, …

- In 1950s, each new model had entirely different instruction set
  - Programmed at machine code or assembly level
- IBM introduced FORTRAN in 1957
  - Much easier to write programs
  - Code wasn’t much slower
  - Possible to use a new machine without reprogramming
  - Algol, PL/I, and other “high-level languages” followed
- Computer makers weren’t so happy
  - Needed to write new compilers (and OS’s) for each new model
    - Written in assembly code
    - Portable compilers didn’t exist
- IBM360 architecture, the first ISA used for multiple models (huge success!)
  - 6 models introduced in 1964
  - Evolved to 370 (virtual addressing) and 390 (32-bit address)
ISA for different segments

- Instruction sets for all three segments are very similar
- Desktops
  - Equal emphasis for integer and floating point,
  - Little regard for code size and power
- Servers
  - High performance, high throughput
  - Many do not need floating-point
- Embedded
  - Emphasis on low cost and power – code size is important
  - Floating-point may be optional
- Desktops and embedded also care about multimedia applications
  - Special media extension instructions

Where are instructions stored?

Harvard Architecture

Von Neumann Architecture

Current Architectures
Crafting an ISA

- Designing an ISA is both an art and a science
- What makes a good ISA?
  - Completeness
  - Orthogonality
  - Regularity and simplicity
  - Compactness
  - Ease of programming
  - Ease of implementation
    - High performance implementation
    - Recently, low-power, low-cost, high-availability, etc.
  - Compatibility
    - x86 (IA-32) generations: 8086, 80286, 80386, 80486, Pentium, Pentium II, Pentium III, Pentium 4, …

Key ISA decisions

- Operations
  - How many?
  - What kinds?
- Operands
  - How big?
  - How many?
  - Locations?
  - How to specify?
- Instruction format
  - How many formats?
  - Instruction length?
Operand locations

- Three types of computers
  - Stack
  - Accumulator
  - Registers
- Two types of register machines
  - Register-memory
    - Most operands can be either in registers or in memory
  - Load-store
    - Most operations deal with registers only
    - Explicit load and store instructions move data between registers and memory

How many operands?

**Stack**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 address</td>
<td>add</td>
<td>tos ← tos + next</td>
</tr>
</tbody>
</table>

**Accumulator**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 address</td>
<td>add</td>
<td>A</td>
</tr>
</tbody>
</table>

**Register-memory**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 addresses</td>
<td>add</td>
<td>Ra, B</td>
</tr>
<tr>
<td>3 addresses</td>
<td>add</td>
<td>Ra, Rb, C</td>
</tr>
</tbody>
</table>

**Load/Store**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3 addresses</td>
<td>add</td>
<td>Ra, Rb, Rc</td>
</tr>
<tr>
<td></td>
<td>load</td>
<td>Ra, Rb</td>
</tr>
<tr>
<td></td>
<td>store</td>
<td>Ra, Rb</td>
</tr>
</tbody>
</table>
Operand locations for four ISA classes

Example

\[
C = A + B
\]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Accumulator</th>
<th>Reg (reg-mem)</th>
<th>Reg (load-store)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push A</td>
<td>Load A</td>
<td>Load R1, A</td>
<td>Load R1, A</td>
</tr>
<tr>
<td>Push B</td>
<td>Add B</td>
<td>Add R3, R1, B</td>
<td>Load R2, B</td>
</tr>
<tr>
<td>Add</td>
<td>Store C</td>
<td>Store R3, C</td>
<td>Add R3, R1, R2</td>
</tr>
<tr>
<td>Pop C</td>
<td></td>
<td></td>
<td>Store R3, C</td>
</tr>
</tbody>
</table>

Registers: fast, exploit locality, reduced memory traffic, easier to re-order
Addressing mode

- Register: add R1, R2, R3
- Immediate: add R1, R2, 3
- Displacement: load R1, 16(R2)
- Register indirect: load R1, (R3)
- Indexed: load R1, R2(R3)
- Auto increment: load R1, (R2)+
- Auto decrement: load R1, (R2)-
- Direct/absolute: load R1, (256)
- Memory indirect: load R1, @(R2)
- Scaled: load R1, (R2)[R3]

Addressing mode (2)

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Example instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Add R1, R2, R3</td>
<td>Regs[R1] ← Regs[R2] + Regs[R3]</td>
</tr>
<tr>
<td>Immediate</td>
<td>Add R1, R2, #3</td>
<td>Regs[R1] ← Regs[R2] + 3</td>
</tr>
<tr>
<td>Displacement</td>
<td>Load R1,16(R2)</td>
<td>Regs[R1] ← Mem[16+Regs[R2]]</td>
</tr>
<tr>
<td>Register indirect</td>
<td>Load R1, (R3)</td>
<td>Regs[R1] ← Mem[Regs[R3]]</td>
</tr>
<tr>
<td>Indexed</td>
<td>Load R1, R2(R3)</td>
<td>Regs[R1] ← Mem[Regs[R2]+Regs[R3]]</td>
</tr>
<tr>
<td>Direct/absolute</td>
<td>Load R1, (256)</td>
<td>Regs[R1] ← Mem[256]</td>
</tr>
<tr>
<td>Memory indirect</td>
<td>Load R1, @(R2)</td>
<td>Regs[R1] ← Mem[Mem[Regs[R2]]]</td>
</tr>
</tbody>
</table>

- More addressing modes → low instruction counts, more complexity (CISC-like)
- Most common modes: immediate and displacement
- Displacement and immediate values: often require fewer than 8 bits, but also often require 16 bits
Frequency of the addressing mode

<table>
<thead>
<tr>
<th>Addressing Mode</th>
<th>Tex</th>
<th>Spice</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory indirect</td>
<td>1%</td>
<td>6%</td>
<td>1%</td>
</tr>
<tr>
<td>Scaled</td>
<td>0%</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>Register indirect</td>
<td>3%</td>
<td></td>
<td>11%</td>
</tr>
<tr>
<td>Immediate</td>
<td></td>
<td>17%</td>
<td>39%</td>
</tr>
<tr>
<td>Displacement</td>
<td></td>
<td>32%</td>
<td>55%</td>
</tr>
</tbody>
</table>

Number of bits of displacement

© 2003 Elsevier Science (USA). All rights reserved.
Number of bits needed for immediate

Little and big endian

Consider a 64-bit quantity, composed of bytes 7-0 (MSB-LSB) (B7, B6, B5, B4, B3, B2, B1, B0)

Little-endian format:
Advantage: easier to organize bytes, half-words, words, double words, etc. into registers (Alpha, x86)

Big-endian format:
Advantage: values are stored in the order they are printed out, the sign is available early (Motorola)
Big-endian

\[ R_1 = 0x01020304 \]

After R1 is stored into memory (0xAA00):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAA00</td>
<td>01</td>
</tr>
<tr>
<td>0xAA01</td>
<td>02</td>
</tr>
<tr>
<td>0xAA02</td>
<td>03</td>
</tr>
<tr>
<td>0xAA03</td>
<td>04</td>
</tr>
</tbody>
</table>

Big-endian may be in registers too!

Bit 0 is the left-most bit (MSB).

Little-endian

\[ R_1 = 0x01020304 \]

After R1 is stored into memory (0xAA00):

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xAA00</td>
<td>04</td>
</tr>
<tr>
<td>0xAA01</td>
<td>03</td>
</tr>
<tr>
<td>0xAA02</td>
<td>02</td>
</tr>
<tr>
<td>0xAA03</td>
<td>01</td>
</tr>
</tbody>
</table>
How many registers?

- **VAX** – 16 registers
  - R15 is program counter (PC)
    - Loading R15 is a jump instruction
- **x86** – 8 general purpose registers
  - Fine print: some restrictions apply
  - Plus floating point and special purpose registers
- Most RISC’s have 32 int and 32 floating point registers
  - Plus some special purpose ones
    - PowerPC has 8 four-bit “condition registers”, a “count register” (to hold loop index), and others.
- IA-64 has 128 integer, 128 floating point, and 64 “predicate” registers

### Operands in an instruction

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-Register (3 ops, 0 mem)</td>
<td>Simple, fixed-length, simple code-generation, easy pipelining and parallelism extraction</td>
<td>High instruction count and code size</td>
<td>Alpha, MIPS, ARM, PowerPC, SPARC</td>
</tr>
<tr>
<td>Register-Memory (2 ops, 1 mem)</td>
<td>Can access data without doing a load, small code size</td>
<td>One of the operands is destroyed, instruction latency is variable</td>
<td>Intel 80x86, Motorola 68000</td>
</tr>
<tr>
<td>Memory-Memory (2 ops, 2 mems)</td>
<td>Most compact code size, doesn’t waste registers</td>
<td>Variation in instruction length (hard to decode), frequent memory accesses, variable instruction latency</td>
<td>VAX</td>
</tr>
<tr>
<td>Memory-Memory (3 ops, 3 mems)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What operations?

• Arithmetic
  – Add, subtract, multiply, divide
• Logic
  – And, or, shift, rotate, etc.
• Load/store
• Control flow
  – Unconditional branch
  – Conditional branch
  – Function call/return
• Floating point operations
• Permutation

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic/Logical</td>
<td>Add, sub, and, or, mul, div</td>
</tr>
<tr>
<td>Data transfer</td>
<td>Loads/stores</td>
</tr>
<tr>
<td>Flow control</td>
<td>Branch, jump, call, return</td>
</tr>
<tr>
<td>System</td>
<td>OS call, virtual memory management, cache</td>
</tr>
<tr>
<td>Floating point</td>
<td>FP add, sub, mul, div</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal add, sub, mult, decimal to character conversions</td>
</tr>
<tr>
<td>String</td>
<td>Move, compare, search</td>
</tr>
<tr>
<td>Graphics</td>
<td>Compression/decompression, vertex/pixel ops</td>
</tr>
</tbody>
</table>
Types of branches

- Conditional branch: `beq R1, R2, label`
- Unconditional branch: `jump label`
- Procedure call: `call fun1`
- Procedure return: `return`

Branch distance
Branch instruction decisions

- How is the destination of a branch specified?
- How is the condition of the branch specified?
- What should be done at function calls/returns?
- What about interrupt?

Specifying the target address

- Absolute address
- PC-Relative: target address = PC + distance
  - Needs fewer bits to encode
  - Independent of how/where the compiled code is linked
  - The displacement needs 4-8 bits typically
- Register-indirect jumps: the address is not known at compile-time and has to be computed at run-time
  - Procedure returns
  - Case statements
  - Virtual functions
  - Function pointers
  - Dynamically shared libraries
Specifying branch conditions

- **Condition codes**
  - Processor status bits are set as a side-effect
    
    sub R1, R2, R3
    
    bz foo

- **Condition register**
  
  cmp R1, R2, R3 ; or cmp.gt
  
  bgt R1, label

- **Compare and branch**
  
  bgt R1, R2, label

<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,</td>
<td>Tests special bits set</td>
<td>Sometimes condition is set for</td>
<td>CC is extra state. Instructions cannot be re-ordered</td>
</tr>
<tr>
<td></td>
<td>PowerPC, SPARC</td>
<td>by ALU ops</td>
<td>free</td>
<td></td>
</tr>
<tr>
<td>Condition Register</td>
<td>Alpha, MIPS</td>
<td>Comparison sets register</td>
<td>Simple</td>
<td>Register pressure</td>
</tr>
<tr>
<td>Compare and branch</td>
<td>PA-RISC, VAX</td>
<td>Comparison is part of</td>
<td>One instruction instead of two</td>
<td>Complex pipelines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the branch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Function call/return**

- Need to maintain a stack of return addresses
  - In memory or in hardware
- Can save all registers together or this can be done selectively
- Who is responsible for saving registers?
  - Caller saving
    - Global register has to be made available to other procedures
    - Only saves values that it cares about
  - Callee saving
    - Saves only as many registers as it needs
    - Some registers can be overwritten
  - A combination of both is typically employed
### Common operations

<table>
<thead>
<tr>
<th>80x86 instruction</th>
<th>Integer average (% total executed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>22%</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>20%</td>
</tr>
<tr>
<td>Compare</td>
<td>16%</td>
</tr>
<tr>
<td>Store</td>
<td>12%</td>
</tr>
<tr>
<td>Add</td>
<td>8%</td>
</tr>
<tr>
<td>And</td>
<td>6%</td>
</tr>
<tr>
<td>Sub</td>
<td>5%</td>
</tr>
<tr>
<td>Move register-register</td>
<td>4%</td>
</tr>
<tr>
<td>Call/Return</td>
<td>2%</td>
</tr>
</tbody>
</table>

### Instruction set encoding

- Instruction bits are precious resources
- Operations are easy to encode efficiently
  - The key issues are the number of operands and their addressing modes
- Few addressing modes
  - Low complexity in decoding and pipelining, but greater code size
- Fixed instruction lengths
  - Low complexity in decoding, but greater code size
Instruction lengths

(a) Variable (e.g., VAX, Intel 80x86)

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

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

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

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Example: instruction layout for MIPS

I-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>Immediate</th>
</tr>
</thead>
</table>

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt = rs op immediate)
Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

Register-register ADDI operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, ...
Read/write special registers and moves

J-type instruction

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Offset added to PC</th>
</tr>
</thead>
</table>

Jump and jump and link
Trap and return from exception
**Compiler**

- Compiler is primary customer of ISA
- Register allocation is a huge contribute to performance
- Compiler writer’s job made easier when ISA has
  - Regularity
  - Primitives, not solutions
  - Simple trade-offs
- Compiler wants
  - Simplicity over power

---

**Compiler optimization**

<table>
<thead>
<tr>
<th>Dependencies</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language dependent; machine independent</td>
<td>Front end per language</td>
</tr>
<tr>
<td>Somewhat language dependent; largely machine independent</td>
<td>Intermediate representation</td>
</tr>
<tr>
<td>Small language dependencies; machine dependencies slight (e.g., register counts/types)</td>
<td>High-level optimizations</td>
</tr>
<tr>
<td>Highly machine dependent; language independent</td>
<td>Global optimizer</td>
</tr>
<tr>
<td></td>
<td>Code generator</td>
</tr>
</tbody>
</table>

© 2003 Elsevier Science (USA), All rights reserved.
Effect of compiler optimization

Register allocation issues

- Graph coloring: determine when variables are live and avoid allocating the same register to variables that are live simultaneously

- Stack: local variables
  - Easy to allocate registers for

- Global data area: statically declared objects
  - Most of them are arrays or other aggregate data structure
  - May be accessed from multiple places (aliasing)
  - Difficult to allocate to registers

- Heap: dynamically created objects, accessed with pointers
  - Difficult to allocate to registers
## RISC vs. CISC

**Complex Instruction Set Computer:**
- Hardware is faster → implement everything in hardware
  - Rich instruction set
  - Complex decoding
  - Complex analysis to identify dependences

**Reduced Instruction Set Computer:**
- Focus on the most frequently used operations
  - Easy to extract parallelism
  - Simpler hardware, easy to achieve higher clock speeds

### Reduced instruction set computer (RISC)

- Relatively few instructions
- Instructions have the same length
- Relatively few instruction formats
- Instructions can be executed in one cycle
  - Except for a few instructions like MUL
  - Good for pipeline design
- Relatively few addressing mode
- Operands are stored in registers
  - except for LOAD/STORE instructions
- A large number of registers
Popular processors

- Intel
  - IA32: CISC
  - IA64: RISC
    - Many features borrowed from HP PA-RISC processors
  - XScale: RISC was ARM’s StrongARM2
- AMD
  - Athlon: RISC
    - Uses technologies in DEC Alpha 21064
    - Converts x86 instructions into fixed-length MicroOPs.
  - Hammer: Athlon with 64-bit extension
- Freescale and IBM
  - PowerPC: RISC (PowerPC, PowerMac, …)
- ARM (Advanced RISC Machines Ltd.)
  - RISC processor core
- SPARC
  - RISC processors designed by Sun, TI, and Fujitsu

RISC vs. CICS

<table>
<thead>
<tr>
<th></th>
<th>RISC</th>
<th>CISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of instructions</td>
<td>&lt; 100</td>
<td>several hundreds</td>
</tr>
<tr>
<td>Length of instructions</td>
<td>fixed</td>
<td>variable</td>
</tr>
<tr>
<td>Instruction format</td>
<td>few</td>
<td>many</td>
</tr>
<tr>
<td>Execution cycle</td>
<td>normally one cycle</td>
<td>variable</td>
</tr>
<tr>
<td>Addressing mode</td>
<td>&lt; 4</td>
<td>many</td>
</tr>
<tr>
<td>Operands</td>
<td>in registers</td>
<td>can be memory to memory</td>
</tr>
<tr>
<td>Number of registers</td>
<td>many (&gt; 30)</td>
<td>few (2 – 16)</td>
</tr>
<tr>
<td>Hardware implementation</td>
<td>efficient</td>
<td>expensive</td>
</tr>
</tbody>
</table>
Dealing with code size in RISC

- Some hybrid versions allow for 16 and 32-bit instructions (40% reduction in code size) – useful for embedded apps

- Compress instructions in memory
  - More hardware complexity

- Reducing the register file size can also reduce the instruction length
  - However, applications need more registers

SPEC89 benchmarks
Key points

- Modern ISA’s typically sacrifice power and flexibility for regularity and simplicity
  - Trade off code density for instruction level parallelism
- Instruction bits are very limited
  - Particularly in a fixed-length instruction format
- Registers are critical to performance
  - IA-64 has 128 registers
- Displacement addressing mode handles the majority of memory reference needs

A little bit more about the history

- In the 1960s, stack architectures were considered a good match for high-level languages
- In the 1970s, ISAs were enriched to make the compiler’s job easier – CISC
  - Software costs were a concern
- In the 1980s, there was a push for simpler architectures – high clock speed and high parallelism – RISC
- In the 1990s, multimedia extensions
- Application-Specific Instruction Set Processor
  - Choose only the instructions critical to the performance of target applications
    - Save die area
    - Reduce power consumption
Case study: PowerPC instruction set architecture

- Integer instructions
- Load and Store instructions
- Flow control instructions
- Trap instructions
- Processor control instructions
- Memory synchronization instructions
- Memory control instructions
- System linkage instructions

Simplified PowerPC block diagram
Registers in a 32-bit PowerPC core

- General-purpose registers
  - 32 registers (R0 to R31)
- Condition registers
  - CR0 to CR7, each 4 bits: LT, GT, EQ, and SO
- XER register
  - Summary overflow (SO), Overflow (OV), and Carry (CA)
- Link register (LR)
- Count register (CTR)
- Other registers
  - Floating-point registers, time base facility, configuration, memory management, exception handling, etc.

General-purpose registers (GPR)

- R0 through R31 are in general-purpose register file
  - R0 is not always 0
  - R0 is considered 0 in some instructions

```
addi   R1, R0, 10
```

You can use any GPR, but follow calling conventions if your codes will be linked with other codes
Condition register (CR)

- A 32-bit register used for testing and branching
- Grouped into eight 4-bit fields: CR0 – CR7
- Can be accessed with instructions such as crand and cror

Example setting CR0

```assembly
cmpw   R1, R2   ; == cmp 0, 1, R1, R2
R1:    10    5    10
R2:    5     10   10
CR0:   010x  100x  001x
       R1 > R2    R1 < R2    R1 == R2

How can you check <=, >=, and !=?
```
Integer instructions

Arithmetic instructions:
\[
\text{add} \quad R1, R2, R3 \quad ; \quad R1 = R2 + R3
\]

Compare instructions:
\[
\text{cmp} \quad cr3,0,R1, R2 \quad ; \text{result will be CR3}
\]
\[
\text{cmpw} \quad cr3, R1, R2 \quad ; \text{Mnemonics}
\]

Logical instructions:
\[
\text{or} \quad R1, R2, R3 \quad ; \quad R1 = R2 \lor R3
\]

Rotate and shift instructions:
\[
\text{rlwnm} \quad R1,R2,R3,0,31 \quad ; \quad R1 = R2 \ll R3
\]
\[
\text{rotlw} \quad R1,R2,R3 \quad ; \text{Mnemonics}
\]

Load and store instructions

Three addressing modes for Load and Store

Register indirect addressing:
\[
\text{lwz} \quad R1, (R2) \quad ; \quad R1 = *R2
\]
\[
; \quad EA = R2
\]

Register indirect with immediate index addressing:
\[
\text{lwz} \quad R1, 4(R2) \quad ; \quad R1 = *(R2 + 4)
\]
\[
; \quad EA = R2 + 4
\]
\[
; \quad \text{R0 is always 0 in lwz}
\]

Register indirect with index addressing:
\[
\text{lwz} \quad R1, R2, R3 \quad ; \quad R1 = *(R2 + R3)
\]
\[
; \quad EA = R2 + R3
\]
Load and store instructions (2)

- Use three addressing modes to load and store
  - byte (8-bit), half word (16-bit), word (32-bit)
- The base register may be updated

Branch and flow control instructions

- Branch instructions
  - \( bx : b, ba, bl, bla \)
    - Absolute address or relative address?
    - Save the next instruction address in LR?
- Branch conditional instructions
  - \( bcx : bc, bca, bcl, bcla \)
- Branch conditional to link register
  - \( bclr \), can be used as return
- Branch conditional to count register
Count register (CTR)

- A 32-bit register that holds a loop count
- Can be decremented by using proper BO fields in conditional branch instructions
  - Bit 2 in the BO field

<table>
<thead>
<tr>
<th>BO</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is FALSE.</td>
</tr>
<tr>
<td>0001b</td>
<td>Decrement the CTR, then branch if the decremented CTR = 0 and the condition is FALSE.</td>
</tr>
<tr>
<td>0010b</td>
<td>Branch if the condition is FALSE.</td>
</tr>
<tr>
<td>0011b</td>
<td>Decrement the CTR, then branch if the decremented CTR = 0 and the condition is TRUE.</td>
</tr>
<tr>
<td>0100b</td>
<td>Decrement the CTR, then branch if the decremented CTR ≠ 0 and the condition is TRUE.</td>
</tr>
<tr>
<td>0101b</td>
<td>Branch if the condition is TRUE.</td>
</tr>
<tr>
<td>1200b</td>
<td>Decrement the CTR, then branch if the decremented CTR ≠ 0.</td>
</tr>
<tr>
<td>1201b</td>
<td>Decrement the CTR, then branch if the decremented CTR = 0.</td>
</tr>
<tr>
<td>1211b</td>
<td>Branch always.</td>
</tr>
</tbody>
</table>

* Bits are ignored and should be cleared, as they may be assigned a meaning in a future version of the architecture.

Table 8-7. BO Operand Encodings

Example: loop using CTR

```
main:
   li   R1, 10
   mtctr R1 ; set the count register
again:
   ...... ; repeated instructions
   bdnz again ; CTR = CTR - 1
                               ; go to again if CTR != 0
exit:
   ; CTR = 10
   ; do {
   ;     } while (CTR)

* bdnz is listed in Table F-4 (PE: page F-7)
```
Call functions using link register (LR)

```assembly
main:
or R1, R2, R3
bl foo ; call foo
add R1, R1, R2 ;

foo:
add R1, R1, R1 ;
blr ; return ; branch to the link register
```

Nested function calls

```assembly
main:
bl bar
nop

bar:
bl foo
nop
......
blr

foo:
.....
blr
```
**Calling convention (Linux)**

- R0: scratch registers
- R1: stack pointer
- R2: system reserved
- R3 – R10: parameter and result passing
  - R3 and R4 are also return values
- R11 – R12: scratch registers
- R13: global pointer to the small data area
- R14 – R31: global integer registers
  - R31 may be used as environment pointers

**Integer exception register**

- Summary overflow (SO), Overflow (OV), and Carry (CA)
- Affected by instructions such as `addo` and `addco`
- CA is set and used by `addc`, `adde`, and other instructions

![Figure 4-2. XER Register](image)
mfspr and mtspr instructions

- Move from special-purpose registers (to GPR)
- Move to special-purpose registers (from GPR)

Mnemonics !!!

\[
\begin{align*}
mfxer & \quad R1 \quad ; \quad mfspr \quad R1, 1 \\
mflr & \quad R1 \quad ; \quad mfspr \quad R1, 8 \\
mfctr & \quad R1 \quad ; \quad mfspr \quad R1, 9 \\
mtxer & \quad R1 \quad ; \quad mtspr \quad 1, R1 \\
mtlr & \quad R1 \quad ; \quad mtspr \quad 8, R1 \\
mtctr & \quad R1 \quad ; \quad mtspr \quad 9, R1 \\
\end{align*}
\]