

## PowerPC Architecture and Assembly Language

An *instruction set architecture* (ISA) specifies the programmer-visible aspects of a processor, independent of implementation

- number, size of registers
- precise function, encoding of instructions
- examples:
  - x86 (aka IA-32, the 32-bit Intel Architecture)
  - MIPS, SPARC, ARM, 68000, 68HC11, Z-80, 8085, ...

The PowerPC ISA was jointly defined by IBM, Apple, and Motorola in 1991

- used by Apple for Power Macintosh systems
- based on IBM POWER ISA used in RS/6000 workstations
- 32-bit and 64-bit versions defined; we'll only consider 32-bit version (implemented by MPC823)

Key “RISC” features:

- fixed-length instruction encoding (32 bits)
- 32 general-purpose registers, 32 bits each
- only load and store instructions access external memory and devices

## A Simple Example

Here's a little code fragment that converts an (infinite) uppercase ASCII string, stored in memory, to lower case:

```
loop:  lbz    r4, 0(r3)
       addi   r4, r4, 'a' - 'A'      # 0x20
       stb    r4, 0(r3)
       addi   r3, r3, 1
       b     loop
```

Let's look at what this does, instruction by instruction:

```
lbz r4, 0(r3)
```

loads a byte and zero-extends it to 32 bits

the *effective address* is  $(r3) + 0$

In the notation of the data book:

$$r4 \leftarrow (24)0 \parallel \text{MEM}(r3) + 0, 1)$$

## Simple Example, cont'd

addi r4, r4, 0x20

add an immediate value

$r4 \leftarrow (r4) + 0x20$

stb r4, 0(r3)

store a byte

again, the effective address is  $(r3) + 0$

$MEM((r3) + 0, 1) \leftarrow r4[24-31]$

addi r3, r3, 1

$r3 \leftarrow (r3) + 1$

b loop

branch to label 'loop'

machine language actually encodes offset (-16)

## Loads & Stores in General

Loads and store opcodes start with 'l' and 'st'. The next character indicates the access size, which can be *byte*, *halfword* (2 bytes), or *word* (4 bytes).

The effective address can be computed in two ways:

1. "register indirect with immediate index"  
aka base + offset, base + displacement  
  
written "d(rA)", effective address is  $(rA) + d$   
  
d is a 16-bit signed value

2. "register indirect with index",  
aka indexed, register + register  
  
written "rA,rB", effective address is  $(rA) + (rB)$   
  
must append 'x' to opcode to indicate index  
  
e.g.: stbx r4, r5, r6

**catch:** if rA (but *not* rB) is r0 in either of these forms, the processor will use the *value* 0 (*not* the contents of r0).

## Loads & Stores cont'd

- Zeroing vs. algebraic (loads only)

Contrast: `lha r4, 0(r3)`

`lhz r4, 0(r3)`

The algebraic option is:

1. not allowed for byte loads
2. illegal for word loads on 32-bit implementations

- Update mode

e.g.: `lwzu r4, 1(r3)`

$EA \leftarrow (r3) + 1$   
 $r4 \leftarrow \text{MEM}(EA, 4)$   
 $r3 \leftarrow EA$

## Load/Store Miscellany

- Unaligned accesses are OK, but slower than aligned

- PowerPC is *big-endian*

- Summary:

|                    |                    |                    |                    |                    |                    |                    |
|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|--------------------|
| <code>lbz</code>   | <code>lhz</code>   | <code>lha</code>   | <code>lwz</code>   | <code>stb</code>   | <code>sth</code>   | <code>stw</code>   |
| <code>lbzx</code>  | <code>lhzx</code>  | <code>lhax</code>  | <code>lwzx</code>  | <code>stbx</code>  | <code>sthx</code>  | <code>stwx</code>  |
| <code>lbzu</code>  | <code>lhzu</code>  | <code>lhau</code>  | <code>lwzu</code>  | <code>stbu</code>  | <code>sthu</code>  | <code>stwu</code>  |
| <code>lbzux</code> | <code>lhzux</code> | <code>lhaux</code> | <code>lwzux</code> | <code>stbux</code> | <code>sthux</code> | <code>stwux</code> |

- Miscellaneous
  - integer doubleword
  - floating-point
  - multiple
  - string
  - byte-reversed
  - reservations

## Arithmetic & Logical Instructions

Most have two versions:

### 1. register-register

ex: `add r1, r2, r3` means  $r1 \leftarrow (r2) + (r3)$

### 2. immediate (*i* suffix)

ex: `addi r1, r2, 5` means  $r1 \leftarrow (r2) + 5$

Immediate operands are limited to 16 bits. (Why?)

Immediates are expanded to 32 bits for processing. Arithmetic operations (add, subtract, multiply, divide) *sign extend* the immediate, while logical operations (and, or, etc.) *zero extend* the immediate. What is the range of a sign-extended 16-bit immediate?

What is the range of a zero-extended 16-bit immediate?

## Arith. & Logical (cont'd)

A few instructions (add, and, or, xor) have a third version:

### 3. immediate shifted (*is* suffix)

ex: `addis r1, r2, 5` means  $r1 \leftarrow (r2) + (5 \parallel 0x0000)$

- `andis`, `oris`, `xoris` let you twiddle bits in the upper half of a register in one instruction.
- The primary use of `addis` is to load a value outside the 16-bit immediate range.
  - funky `ld/st r0` rule applies (`addi` and `addis` *only*)
- “simplified mnemonics”:
  - `lis`
  - `li`

## Aside: Dealing w/32-bit immediates

Two ways to put a full 32-bit value in a register:

```
lis    r3, 5
ori    r3, r3, 99
```

or

```
lis    r3, 5
addi   r3, r3, 99
```

When are these *not* equivalent?

Assembler suffixes:

- @h
- @ha
- @l

## Arithmetics (cont'd)

Subtraction instruction is 'subf': subtract from

```
subf   r3, r4, r5    means    r3 ← r5 - r4
```

- Why is this actually useful?

- More “simplified mnemonics”:

- sub
- subi

Negation:

```
neg    r3, r4        means    r3 ← -r4
```

Numerous other add/sub variants deal with carry flag (XER[CA]) for extended precision.

## Multiply/Divide

Classic problem: product of two 32-bit numbers is 64 bits.

- lower 32 bits
  - mulli
  
- mullw
  
- upper 32 bits
  - mulhw
  
- mulhwu

Divides:

- divw
  
- divwu

## Logicals

About what you'd expect:

- and
- or
- xor

Plus a few more (no immediate forms):

- nand
  
- nor
  - simplified mnemonic: not
  
- eqv
  
- andc
  
- orc

And on the bleeding edge:

- extsb, extsh
  
  
- cntlzw

## Example Revisited

Here's a more complete version of the example that:

- initializes the address
- stops at the end of the string

```
string: .ascii "BIFF\0"
```

```
main: lis    r3, string@h
      ori    r3, r3, string@l
```

```
loop: lbz    r4, 0(r3)
      cmpwi r4, 0
      beq    done
      addi  r4, r4, 'a' - 'A'    # 0x20
      stb   r4, 0(r3)
      addi  r3, r3, 1
      b     loop
```

```
done: b     done
```

## New Instructions

```
cmpwi r4, 0
```

compare word immediate

sets three condition code bits (in CR register):

- LT
- GT
- EQ

```
beq done
```

branch if equal

branches iff (EQ == 1)

## Condition Codes in General

Four compare instructions:

- `cmpw, cmpwi`
- `cmplw, cmplwi`

Also, any arithmetic/logical instruction *may* set the condition codes as a side effect, if you append a '.' to the opcode.

```
and.   r3, r4, r5
```

is equivalent to

```
and    r3, r4, r5
cmpwi  r3, 0
```

Exceptions: the following instructions *do not exist*

- `addi., addis.`
- `andi, andis`
- `ori., oris., xori., xoris.`

## Conditional Branches

Can branch on any one condition bit true or false:

- `blt`
- `bgt`
- `beq`
- `bge (also bnl)`
- `ble (also bng)`
- `bne`

Any number of instructions that do not affect the condition codes may appear between the condition-code setting instruction and the branch.

## The Count Register (CTR)

A special register just for looping.

```
li    r4, 100
mtctr r4
loop: lwzu r5, 4(r6)
      add  r7, r7, r5
      bdnz loop
```

mtctr: move to CTR  
requires register (no immediate form)  
mfctr also available

bdnz: branch decrement not zero

```
CTR ← CTR-1
branch iff (CTR == 0)
```

condition codes are unaffected

- can combine condition code test:

```
bdnzt eq,loop
```

```
CTR ← CTR-1
branch iff ((CTR == 0) && (EQ == 1))
```

variations:

- bdnzf
- bdz
- bdzf, bdnzf

## The Hairy Truth

- There is a fourth condition code bit (SO, for “summary overflow”)
- There are eight condition registers (CR0-CR7)
  - total of 32 condition bits
  - compares & branches use CR0 by default
  - dotted arithmetic/logicals always use CR0
  -
- There are 1,024 possible conditional branches
- All the compares and conditional branches we’ve discussed are “simplified mnemonics”... “see Appendix F”!

## Shifts and Rotates

All shifts & rotates have '.' variants

### Shifts

- slw, srw, slwi, srwi: shift (left|right) word (immediate)
  
- slwi, srwi are “simplified mnemonics” for special case of ‘rlwinm’ rotate
  
- sraw, srawi: shift right algebraic word (immediate)

## Rotates

All rotates have two steps:

1. Rotate left by specified amount
  - same as rotate right by  $(32 - n)$
2. Combine result with mask
  - mask specified by beginning & ending bit positions (called MB and ME)
  - bits MB through ME are 1, others are 0
  
  - if  $(MB > ME)$ , the 1s “wrap around”
  
  - rlwinm: rotate left word immediate then AND with mask  
rlwinm rD, rS, Rotate, MaskBegin, MaskEnd
  
  - rlwnm: rotate left word then AND with mask
    - like rlwinm, but rotate count in register (not immediate)
  
  - rlwimi: rotate left word immediate then mask insert

## Rotate Examples

“simplified mnemonics”:

`rotrw rA, rS, rB` = `rlwnm rA, rS, rB, 0, 31`

`rotlwi rA, rS, n` = `rlwinm rA, rS, n, 0, 31`

`rotrwi rA, rS, n` = `rlwinm rA, rS, 32-n, 0, 31`

`slwi rA, rS, n` = `rlwinm rA, rS, n, 0, 31-n`

`extlwi rA, rS, n, b` = `rlwinm rA, rS, b, 0, n-1`

- extract n bits starting at position b, left justify

`extrwi rA, rS, n, b` = `rlwinm rA, rS, b+n, 32-n, 31`

- extract n bits starting at position b, right justify

## Rotate Examples cont'd

`inslwi rA, rS, n, b` = `rlwimi rA, rS, 32-b, b, b+n-1`

- take n bits from the left end of rS, insert in rA starting at position b
- leaves other bits of rA unchanged

`insrwi rA, rS, n, b` = `rlwimi rA, rS, 32-(b+n), b, b+n-1`

- same, but take bits from the right end of rS

`rlwinm` is also useful for simple masking (i.e. rotate count = 0)