## 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 semantics, encoding of instructions

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
- MPC823 implements 32-bit version, no floating point

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; all others operate only on registers

Not-so-RISC features:

- several special-purpose registers
- some really strange instructions (like rlwimi)

## A Simple Example

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

```
loop:   lbz     r4, 0(r3)
        addi    r4, r4, 0x20            # 'a' - 'A'
        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 ← (24)0 || MEM( (r3) + 0 , 1)

## Simple Example, cont'd

addi r4, r4, 0x20

add an immediate value

r4 ← (r4) + 0x20

stb r4, 0(r3)

store a byte

again, the effective address is (r3) + 0

MEM( (r3) + 0 , 1) ← r4[24-31]

addi r3, r3, 1

r3 ← (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 indexed

   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 (use extsb instruction)
  2. illegal for word loads on 32-bit implementations

- Update mode

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

EA ← (r3) + 1
r4 ← MEM(EA, 4)
r3 ← EA

## Load/Store Miscellany

- Unaligned accesses are OK, but slower than aligned

- PowerPC is *big-endian*

- Summary:

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

- 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 ← (r2) + (r3)

2. immediate (i suffix)

   ex: addi r1, r2, 5 means r1 ← (r2) + 5

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

Immediates are *always* 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 ← (r2) + (5 || 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

- subfic is immediate version; 'c' indicates carry flag is set
- sub, subi are "simplified mnemonics"
- neg (negate)
- Numerous other add/sub variants deal with carry flag (XER[CA]) for extended precision.

Multiply:

- classic problem: product of two 32-bit numbers is 64 bits
- mulli, mullw generate lower 32 bits of product
- mulhw, mulhwu generate upper 32 bits of product

Divide:

- divw, divwu for signed/unsigned division

## Logicals, Shifts, and Rotates

Basics (register-register or register-immediate):

- and, or, xor

Plus a few more (no immediate forms):

- nand, nor, not
- eqv (not xor)
- andc, orc (complement second argument)

And on the bleeding edge:

- cntlzw

Shifts:

- slw, srw, slwi, srwi: shift (left|right) word (immediate)
- sraw, srawi: shift right algebraic word (immediate)

Rotates:

- rotlw, rotlwi, rotrwi: shift (left|right) word (immediate)
- no rotrw: must rotate left by 32 - *n* (use subfic)
- all are simplified mnemonics for rlwnm/rlwinm...

## Rotates In Their Full Glory

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

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

## 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, 0x20          # 'a' - 'A'
        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, shift or rotate 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
- bdzt, bdzf

## 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
  - can do boolean ops directly on CR bits
- There are 1,024 possible conditional branches
- All the compares and conditional branches we’ve discussed are “simplified mnemonics”... “see Appendix F”!