COMP-573A Microcomputers

PowerPC Architecture 6xx

slides by Alexandre Denault

A bit of history …

- The original idea for the PowerPC architecture came from IBM's *Power* architecture (introduced in the Risc/6000)
- At that time, IBM was interested in finding business partners to expand *Power*'s market.
- IBM approached Apple, who was currently looking at new Risc solutions.
- Motorola, who has extensive knowledge of the embedded market was also brought into the deal.
- Thus was born the Apple-IBM-Motorola alliance.

Addition to the Power Architecture

- Big or little-endian ordering (Power has little-endian, while most of Motorola's chip were big-endian)
- Single and double precision floating-point arithmetic
- 64-bit architecture, backward compatible to 32-bit

The PowerPC architecture

- Architecture resembles a mix between Sparc Risc and Motorola Cisc.
- Architecture has different implementation levels (so the chip does not need to be fully implemented for embedded solutions).
- Load and store architecture. Operations are always done over registers.
- Offers a large number of mnemonics that increase the number of instructions without increasing the number of on-chip instruction.
- Passes arguments using registers and the stack.

Memory Architecture

- Like Sparc …
  - Memory is never directly addressed, so we do not need to concern ourselves with the memory architecture.
  - Registers in the PowerPC architecture are 32-bit, allowing us to address 4 gigabytes of virtual memory.

General Purpose Registers

- The Power PC uses a flat-scheme of 32 general purpose registers.
- Some of these registers have special tasks assigned to them:
  - r0 Volatile register which may be modified during function linkage
  - r1 Stack frame pointer, always valid
  - r2 System-reserved register
  - r3-r4 Volatile registers used for parameter passing and return values
  - r5-r10 Volatile registers used for parameter passing
  - r11-r12 Volatile registers which may be modified during function linkage

General Purpose Registers (cont.)

- r13 Small data area pointer register
- r14-r30 Registers used for local variables
- r31 Used for local variables or "environment pointers“

- Non-volatile registers (ex: r0, r14-r31) must be saved before they are used. At the end of a function call, they must be restored to their original values.
- When working with *gcc/gas*, registers are referenced as numbers (ex: 1,2,3, etc). Great care must be used not to confuse a reference to register 1 and the numeral 1.

General Purpose Registers (cont.)

- References to the register 0 is sometimes interpreted as the numeral 0 by some functions, even if the parameter was supposed to be a register. When unsure, it is best to avoid using variable 0.
- The PowerPC specification is the best resource to identify the expected arguments of each function.

Floating-Point Registers

- The Power PC architecture offers 32 floating-point registers with 64-bit precision.
- Either single precision or double precision floating-point numbers can be stored in these registers.
- Instructions to load and store double precision floating-point numbers transfers 64-bit of data without conversion.

Floating-Point Registers (cont.)

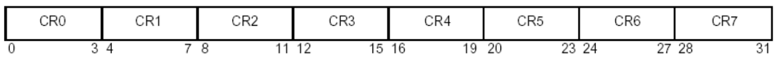
- Instructions to load from memory single precision floating point numbers will convert the numbers to double precision format before storing them in the register.
- Instructions to store to memory a single precision number from a register are provided.
- Unlike the Sparc architecture, floating-point arguments **are** passed in the floating-point registers.

Floating-Point Registers (cont.)

- Floating have special purposes assign to them:
  - f0 Volatile register
  - f1 Volatile register used for parameter passing and return values
  - f2-f8 Volatile registers used for parameter passing
  - f9-f13 Volatile registers
  - f14-f31 Registers used for local variables
- Like mentioned previously, the value of non-volatile registers must be saved if they are used in a function.

Conditional Register

- This 32-bit register is divided into 8 groups of 4-bit registers.



- This allows the programmer to maintain 8 different results of “condition” that can be used for branching.
- The first parameter of a “compare” or a “branch” instruction is the CR window number (0 – 7).

Conditional Register (cont.)

- The Condition Register can be used for both integer and floating point operations.
- The CR0 is often use to store the implicit (condition) results of an integer operation.
- The CR1 is often use to store the implicit (condition) results of an integer operation.
- All CR windows can be used to store the result of a compare operation.

Conditional Register (cont.)

- In these three scenarios, the signification of the four condition bit changes.
- Certain instructions allow the programmer to copy CR windows or to manipulate them at the bit level.
- CR2, CR3 and CR4 are nonvolatile (value on entry must be preserved on exit)

Floating-Point Status and Control Register

- The Floating-Point Status and Control Register (FPSCR) allows the programmer to:
  - Recording exceptions generated by FP operations
  - Recording the type of the result produced by a FP operation
  - Controlling the rounding mode used by FP operations
  - Enabling or disabling the reporting of exceptions
- The first 24 are bits are status bits. The 12 other bits are used to control FP operations

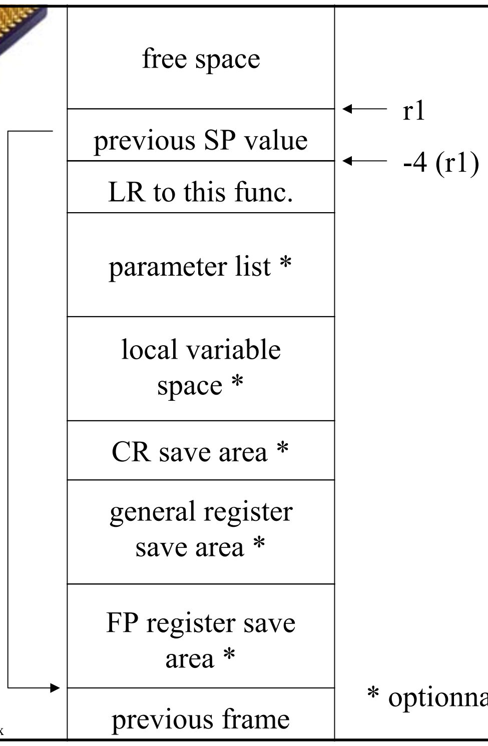
FP Status and Control Register (cont.)

- Some of the status bits are sticky. Once set, these bits will remained set until cleared by the appropriate operation.
- Note: An instruction allows you to copy bits from the FPSCR to a CR windows. These bits can be used to control branching (tricky).

Link Register

- The Link Register is used to record the return address of an instruction when the program branches to a function.
- This return address is automatically written to LR when the branch to function operation is used.
- If a programmer wants to branch to another function, it his responsibility to save the content of the LR register (since the content will be overwritten when he branches to the next function).

The Stack



* optionnal component

The Stack (cont.)

- When creating a function, the programmer must calculate the size of the stack frame he will need.
  - Start with a base value of 8
  - If function with more than 8 integer parameters and/or 8 floating-point parameters will be used, add 4 bytes for each extra parameter integer parameters and add 8 bytes for each extra floating-point parameter.
  - Add 4 bytes for each local variable you want to create (8 if you want that local variable to be a double precision floating-point number).

The Stack (cont.)

- CR2, CR3 and CR4 are considered non-volatile, so an additional 4 byte must be added to save them if they will be modified.
- Several general purpose registers and floating-point registers are considered non-volatile. For each non-volatile general purpose register which is modified, add 4 bytes. For each non-volatile floating-point register which is modified, add 8 bytes.
- The final number must be a multiple of 16 (pad with an extra bytes of temporary space if it is not).

The Stack (cont.)

- Data in the stack is most often addressed using r1 (stack pointer) or r31 (frame pointer).
- However, in the PowerPC architecture, the frame pointer does not need to be defined and often carries the same value as the stack pointer.
- As previously mentioned, before a function is called, the current value of LR must be save.
- It is important to note that the current value of the LR register must be saved in previous stack frame (in the reserved space).

The Stack (cont.)

- Under Linux, the format of the stack is defined in the SYSTEM V ABI PowerPC Processor Supplement document.
- Other OS, such as MacOs X have a different stack format.

Addressing Memory

- Register Indirect : (rA)
  The effective address is stored in a register.
- Register Indirect with Index : (rA) + rB
  The effective address is calculated by adding the contents two integer registers.
- Register Indirect with Offset : (rA) + offset
  The effective address is calculated by adding a signed integer constant to a register.

Instruction Set

- Like Sparc, all instructions have a fixed length of 32-bit.
- Most operation (other than load and store) can only be done over registers.
- In “gas”, registers are referenced like absolute values. It is necessary to know the format of each instruction to differentiate register reference from absolute values.

Load and Store

- As mentioned before, the PowerPC is a load and store architecture. Many operation cannot be done over memory.
- Parenthesis () are used to refer to the content of memory at a particular address.
- As mentioned before, offset can be added to a memory reference to target specific blocks of memory.
- Constants are loaded into registers with the mnemonic li, which is equivalent to an addition with 0.

```
li    4,8   -- Load the value 8 in register4.
```

Load and Store (cont.)

- The load word and zero (stw) instruction can be used to copy a value from memory to register (note: if 0 is used as a destination register, the value 0 will be used instead).

```
lwz   3,8(1)    -- load the value at memory address
             -- register1 + 8 to register 3
```

- A popular variation is the load word and zero with update instruction which will update the destination register with the address in memory where the value was stored.

```
lwzu  3,8(1)     --  load the value at memory address
             -- register1 + 8 to register 3
                 -- value of register1 is changed to reg1 + 8
```

Load and Store (cont.)

- The store word (stw) instruction can be used to copy a value from register to memory (note: if 0 is used as a destination register, the value 0 will be used instead).

```
stw   3,8(1)-- Store the value of register3 at
            -- memory address register1 + 8
```

- A popular variation is the store word with update instruction which will update the destination register with the address in memory where the value was stored.

```
stwu   3,8(1) -- Store the value of register3 at
              -- memory address register1 + 8 and
              -- value of register1 is changed to reg1 + 8
```

Arithmetic and Logic Instructions

- The SPARC architecture uses 2's complement representation for signed integer values.
- Most arithmetic instruction such as add and subtraction are signed.
- Some instructions, such as addi will use the value 0 if register 0 is used as an argument.

```
add      1, 2, 3      -- r1 = r2 + r3
addi     1, 2, 3      -- r1 = r2 + 3
addi     1, 0, 3      -- r1 = 3
sub      1, 2, 3      -- r1 = r2 - r3
subf     1, 2, 3      -- r1 = r3 - r2
and      1, 2, 3      -- r1 = r2 AND r3
neg      1, 2         -- r1 = - r2
```

Multiplication

- Multiplication of 32-bit numbers much be executed with two instruction if the full 64-bit result is needed: high-order [0-31] and low-order [32-63]
- There are two types of low-order multiplication, multiply low-word (mullw) and multiply low immediate (mulli).
- Since there is no difference in the last 32-bits of a signed/unsigned multiplication, there is no need for two types of multiply low-word instruction.
- However, high-order multiplication do come in sign/unsigned flavor (mulhw and mulhwu).

Division

- The division operation is available in signed and unsigned format (divw and divwu).
- The quotient is saved in the destination register.
- There are no instruction to obtain the remainder of a division. It must be calculated manually using a division, a multiplication and a subtraction.

```
mullw    1, 3, 4      -- r1 = (r3 * r4) bit [32-63]
mulhw    2, 3, 4      -- r2 = (r3 * r4) bit [0-31]
mullwi   1, 2, 3      -- r1 = r2 * 3
divw     1, 2, 3      -- r1 = r2 / r3
divwu    1, 2, 3      -- r1 = r2 / r3
```

Floating-point instructions

- A PowerPC processor has 32 floating point registers.
- Numbers in FP registers are stored as double precision floating-point numbers. (Single precision numbers are converted to double when stored in registers).
- The PowerPC offers both single precision and double precision operations.
- When executing single precision operation, the calculations are executed as double precision and rounded down to single precision.
- The PowerPC offer special instructions to execute polynomial calculation.

Loading Constants

- Constants need to be loaded using 2 instructions (since their 32 bit address is too large to fit in an single instruction).

```
lis 3,.LC0@ha         --set 16 first bits
la 3,.LC0@l(3)        --set 16 last bits
```

- The @ha and @l keywords allow use to isolate a specific part of an address without the need to use bit shifting.

Code Alignment

- Some architecture require specific aligning of instructions in memory
- PowerPC also requires an alignment of a least 3.

Instruction Reference

- Here are some of the most common PowerPC assembly instructions.
- The descriptions were taken from the “PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors”

Load Word and Zero

Syntax:

```
lwz rD,d(rA)
```

Operation:

```
if rA = 0
    then b ← 0
    else b ← (rA)
EA ← b + EXTS(d)
rD ← MEM(EA, 4)
```

Description:

EA is the sum (rA|0) + d. The word in memory addressed by EA is loaded into rD.

Load Word and Zero with Upda

Syntax:

```
lwzu rD,d(rA)
```

Operation:

```
EA ← (rA) + EXTS(d)
rD ← MEM(EA, 4)
rA ← EA
```

Description:

EA is the sum (rA) + d. The word in memory addressed by EA is loaded into rD.

EA is placed into rA.

If rA = 0, or rA = rD, the instruction form is invalid.

Load Immediate

Syntax:
  li rD,value

Operation:
  rD ← value + 0

Description:
  The “li” instruction is a mnemonics for “addi rD,0,value”.

Load Address

Syntax:
  la rD,disp(rA)

Operation:
  rD ← (rA) + disp

Description:
  The “la” instruction is a mnemonics for “addi rD,rA,disp”.

Load Immediate Shifted

Syntax:

```
lis rD,value
```

Operation:

```
rD ← (SIMM || (16)0)
```

Description:

The “lis” instruction is a mnemonics for “addis rD,0,value”.

Store Word

Syntax:

```
stw rS,d(rA)
```

Operation:

```
if rA = 0
    then b ← 0
    else b ← (rA)
EA ← b + EXTS(d)
MEM(EA, 4 ← ¬ rS
```

Description:

EA is the sum (rA|0) + d. The contents of rS are stored into the word in memory addressed by EA.

Store Word with Update

Syntax:

stwu rS,d(rA)

Operation:

EA ← (rA) + EXTS(d)

MEM(EA, 4) ← (rS)

rA ← EA

Description:

EA is the sum (rA) + d. The contents of rS are stored into the word in memory addressed by EA.

EA is placed into rA.

If rA = 0, the instruction form is invalid.

Move Register

Syntax:

mr rA,rS

Operation:

rA ← (rS) OR (rs)

Description:

The “mr” instruction is a mnemonics for “or rA,rS,rS”.

Move from Link Register

Syntax:

mflr rD

Operation:

rD ← SPR(8)

Description:

The “mflr” instruction is a mnemonics for “mfspr rD,8”.

Move to Link Register

Syntax:

mtlr rD

Operation:

SPR(8) ← (rS)

Description:

The “mtlr” instruction is a mnemonics for “mtspr 8,rD”.

Addition

Syntax:

add rD,rA,rB

Operation:

```
rD ← (rA) + (rB)
```

Description:

The sum (rA) + (rB) is placed into rD.

Add Immediate

Syntax:

addi rD,rA,SIMM

Operation:

```
if rA = 0
    then rD ← EXTS(SIMM)
    else rD ← (rA) + EXTS(SIMM)
```

Description:

The sum (rA|0) + sign extended SIMM is placed into rD.

NOTE: addi uses the value 0, not the contents of GPR0, if rA = 0.

Subtract From

Syntax:

add rD,rA,rB

Operation:

rD ← ¬(rA) + (rB) + 1

Description:

The sum ¬ (rA) + (rB) + 1 is placed into rD. (equivalent to (rB)-(rA))

Logical And

Syntax:

and rA,rS,rB

Operation:

rA ← (rS) & (rB)

Description:

The contents of rS are ANDed with the contents of rB and the result is placed into rA.

Two’s Complement Negation

Syntax:

neg rD,rA

Operation:

rD ← ¬ (rA) + 1

Description:

The value 1 is added to the one’s complement of the value in rA, and the resulting two’scomplement is placed into rD.

Note: If rA contains the most negative 32-bit number (0x8000_0000), the result is the most negative number.

Multiply Low Word

Syntax:

mullw rD,rA,rB

Operation:

prod[0–63] ← (rA) * (rB)

rD ← prod[32-63]

Description:

The 32-bit operands are the contents of rA and rB. The low-order 32-bits of the 64-bit product (rA) * (rB) are placed into rD.

The low-order 32-bits of the product are independent of whether the operands are regarded as signed or unsigned 32-bit integers.

This instruction can be used with mulhw to calculate a full 64-bit product.

NOTE: This instruction may execute faster on some implementations if rB contains the operand having the smaller absolute value.

Multiply Low Immediate

Syntax:

mulli rD,rA,SIMM

Operation:

prod[0–63] ← (rA) * EXTS(SIMM)

rD ← prod[32-63]

Description:

The first operand is (rA). The second operand is the sign-extended value of the SIMM field.

The low-order 32-bits of the 64-bit product of the operands are placed into rD.

Both the operands and the product are interpreted as signed integers. The low-order 32-bits of the product are calculated independently of whether the operands are treated as signed or unsigned 32-bit integers.

This instruction can be used with mulhd*x* or mulhw*x* to calculate a full 64-bit product.

Multiply High Word

Syntax:

mulhw rD,rA,rB

Operation:

prod[0–63] ← (rA) * (rB)

rD ← prod[0–31]

Description:

The 64-bit product is formed from the contents of rA and rB. The high-order 32 bits of the 64-bit product of the operands are placed into rD.

Both the operands and the product are interpreted as signed integers.

This instruction may execute faster on some implementations if rB contains the operand having the smaller absolute value.

Multiply High Word Unsignec

Syntax:

mulhwu rD,rA,rB

Operation:

prod[0–63] ← (rA) * (rB)

rD ← prod[0–31]

Description:

The 32-bit operands are the contents of rA and rB. The high-order 32 bits of the 64-bit product of the operands are placed into rD.

This instruction may execute faster on some implementations if rB contains the operand having the smaller absolute value.

Divide Word

Syntax:

divw rD,rA,rB

Operation:

dividend ← (rA)

divisor ← (rB)

rD ← dividend ¸ divisor

Description:

The dividend is the contents of rA. The divisor is the contents of rB. The remainder is not supplied as a result. Both the operands and the quotient are interpreted as signed integers. The quotient is the unique signed integer that satisfies the equation—dividend = (quotient * divisor) + r where 0 r < |divisor| (if the dividend is non-negative), and –|divisor| < r 0 (if the dividend is negative).

If an attempt is made to perform either of the divisions—0x8000_0000 ¸ -1 or <anything> ¸ 0, then the contents of rD are undefined.

Floating Addition (Double)

Syntax:

fadd frD,frA,frB

Operation:

frD ← (frA) + (frB)

Description:

The floating-point operand in frA is added to the floating-point operand in frB. If the mostsignificant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

If a carry occurs, the sum's significand is shifted right one bit position and the exponent is increased by one. FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Floating Multiplication (Doubl

Syntax:

fmul frD,frA,frC

Operation:

frD ← (frA) * (frC)

Description:

The floating-point operand in register frA is multiplied by the floating-point operand in register frC. If the most-significant bit of the resultant significand is not a one, the result is normalized.

The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Floating Multiply-Add (Double

Syntax:

fmadd frD,frA,frC,frB

Operation:

frD ← ((fra) * (frC)) + (frB)

Description:

The floating-point operand in register frA is multiplied by the floating-point operand in register frC. The floating-point operand in register frB is added to this intermediate result.

If the most-significant bit of the resultant significand is not a one, the result is normalized. The result is rounded to double-precision under control of the floating-point rounding control field RN of the FPSCR and placed into frD.

FPSCR[FPRF] is set to the class and sign of the result, except for invalid operation exceptions when FPSCR[VE] = 1.

Floating Move Register

Syntax:

fmr frD,frB

Operation:

frD ← (frB)

Description:

The content of register frB is placed into frD.

Compare Words

Syntax:

```
cmpw CRn,rA,rB
```

Operation:

```
a ← (rA)
b ← (rB)
if a < b then c ← 0b100
    else if a > b then c ← 0b010
    else c ← 0b001
CRn ← c
```

Description:

The contents of rA are compared with the contents of rB, treating the operands as signed integers. The result of the comparison is placed into CR field CRn.

The “cmpw” instruction is a mnemonics for “cmp 3,0,rA,rB”.

Compare Word Immediate

Syntax:

```
cmpwi CRn ,rA,value
```

Operation:

```
a ← (rA)
if a < EXTS(value) then c ← 0b100
    else if a > EXTS(value) then c ← 0b010
    else c ← 0b001
CRn ← c
```

Description:

The contents of rA are compared with the sign-extended value of the value field, treating the operands as signed integers. The result of the comparison is placed into CR field CRn.

The “cmpw” instruction is a mnemonics for “cmp 3,0,rA,rB”.

Branch

Syntax:

b target_addr

Operation:

NIA ← CIA + EXTS(LI || 0b00)

Description:

Target_addr specifies the branch target address.

The branch target address is the sum of LI || 0b00 sign-extended plus the address of this instruction.

Branch Not Equal

Syntax:

bne CRn,target

Operation:

```
if ctr_ok & cond_ok
    then NIA ←  CIA + EXTS(BD || 0b00)
```

Description:

The “bne” instruction is a mnemonics for “bc CRn,10,target”.

Target_addr specifies the branch target address.

The branch target address is the sum of LI || 0b00 sign-extended plus the address of this instruction.

Branch to Function

Syntax:

bl target_addr

Operation:

NIA ← CIA + EXTS(LI || 0b00)

LR ← CIA + 4

Description:

Target_addr specifies the branch target address.

The branch target address is the sum of LI || 0b00 sign-extended plus the address of this instruction.

The effective address of the instruction following the branch instruction is placed into the link register.

Return from Branch

Syntax:

blr

Operation:

NIA ← LR

Description:

The effective address of the instruction following the branch instruction is taken from the link register.

Hello Word

```
.section        .rodata                 ! Constant Declaration
        .align 2
.LC0:
        .string "Hello World"          ! HelloWorld string

.section        ".text"
        .align 2
        .globl main                    ! Declare main global so the shell can execute it
.type   main,@function                 ! Main function
main:
        stwu 1,-16(1)                  ! Allocate a stackframe of 16 bytes
        mflr 0                         ! Move link register to register0
        stw 0,20(1)                    ! Store link register in previous stack frame
        lis 3,.LC0@ha                  ! Load first 16 bits of address of string
        la 3,.LC0@l(3)                 ! Load last 16 bits of address of string
        crxor 6,6,6                    ! Needed for ABI compliance
        bl printf                      ! Call to printf (branch)
        lwz 0,20(1)                    ! Bring back link register from stack frame
        mtlr 0                         ! Move original link register add in link register
        addi 1,1,16                    ! Deallocate stackframe
        blr                            ! Return (branch return)
```

Temporary Variables and Arithmetic

```
main:                                  ! Main function
        stwu 1,-48(1)                  ! Allocate stack frame
        stw 31,44(1)                   ! Save value of register31 (non-volatile)
        mr 31,1                        ! Move SP in register 31
        stw 3,8(31)                    ! Save first arg
        stw 4,12(31)                   ! Save second arg
        li 0,5                         ! temp = 5
        stw 0,16(31)                   ! x = temp
        li 0,6                         ! temp = 6
        stw 0,20(31)                   ! y = temp
        li 0,7                         ! temp = 7
        stw 0,24(31)                   ! z = temp
        lwz 9,16(31)                   ! temp2 = x
        lwz 0,20(31)                   ! temp = y
        add 0,9,0                      ! temp = temp2 + temp
        stw 0,24(31)                   ! z = temp1
        lwz 0,24(31)                   ! temp = z
        mr 3,0                         ! Prepare to return z
        lwz 11,0(1)                    ! Get old sp
        lwz 31,-4(11)                  ! Save old value back to 31
        mr 1,11                        ! Restore old sp to register1
        blr                            ! Return
```

If statement

C Code:

```
if( i == 0 ) {
/* Inside if */
}
/* Outside if */
```

Assembler:

```
          lwz 0,8(31)
          cmpwi 0,0,0               ! Store result in CR0, compare value of register0
                                    ! With the value 0
          bne 0,.L2                 ! Jump, depending on value of CR0
          /* Inside if */
  .L2:
          /* Outside if */
  .L3:
          lwz 0,12(31)
```

For statement

C Code:

```
for (j = 0; j < 15; j++) {
/* Inside for */
}
/* Outside for */
```

Assembler:

```
        stw 0,12(31)            ! j = 0
.start_for:
        lwz 0,12(31)            ! temp = j
        cmpwi 0,0,14            ! if ( j = 14 )
        ble 0,.in_for           ! then goto in_for
        b .exit_for             ! else goto exit_for
.in_for:
         /* Inside for */
        lwz 9,12(31)            ! temp = j
        addi 0,9,1              ! temp = temp + 1
        stw 0,12(31)            ! j = temp
        b .start_for            ! goto start_for
.exit_for:
        /* Outside for */
```

Evolution of the PowerPC

- The initial PowerPC included plans for a 64-bit PowerPC processor.
- This means application compiled for a 32-bit processor will still function on a 64-bit processor.
- The G5, Apple's newest line of computer, is equipped with a 64-bit PowerPC chip.
- The G5 processor is based on IBM's Power4 architecture.

Reference

- PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors
- Balance of Power: Introducing PowerPC Assembly Language
  (http://www.mactech.com/articles/develop/issue_21/21balance.html)
- System V R4 ABI, PowerPC edition
- The calling sequence and stack frame for Linux
  (http://math-atlas.sourceforge.net/devel/atlas_contrib/node93.html)
- Register usage for Linux
  (http://math-atlas.sourceforge.net/devel/atlas_contrib/node92.html)