COMP-573A Microcomputers

SPARC Architecture v8-v9

slides by Alexandre Denault

Design Goals

SPARC was designed as a target for optimizing compilers and easily pipelined hardware implementations. SPARC implementations provide exceptionally high execution rates and short time-to-market development schedules.

The SPARC Architecture Manual, Version 8

A bit of history …

- 1984: A team of Sun engineers (including Bill Joy) decide to create the SPARC architecture, mostly based on the work of Patterson.
- 1986: SPARC V7 specification is published.
- 1986: First SPARC processor is implemented by Sun/Fujistu.
- 1989: SPARC International is founded and the SPARC Architecture becomes an open standard.
- 1990: SPARC V8 specification is published.
- 1993: SPARC V9 specification is published.

Why create SPARC International?

- To establish a strong set of standards and avoid vendor dependent solutions.
  - Control of the SPARC architecture is in the hands of an independent, non-profit organization, SPARC International, whose membership is open to everyone.
- To protect the SPARC label and test the various implementation for conformance.
  - SPARC International has developed the *SPARC Application Conformance Toolkit*, which can test systems for conformance against the *SPARC Compliance Definition*.

The SPARC Architecture

- Load and store architecture. Operations are always done over registers.
- Offers a large number of registers using a “register window” scheme.
- Instruction set has only 72 basic instructions.
- Passes arguments using registers and the stack.
- Optimizes branch instruction using a delay slot.

Memory Architecture

- Memory is never directly addressed, so we do not need to concern ourselves with the memory architecture.

General Purpose Registers

- The SPARC uses a “register scheme” to manage the large number of registers available to the the programmer.
- A SPARC processor can contain anywhere between 52 and 524 general purpose registers.
- At any moment, a programmer has access to 32 of these registers.
- 8 of these registers are global, thus available from any function.
- The 24 other registers are part of the register sliding window.

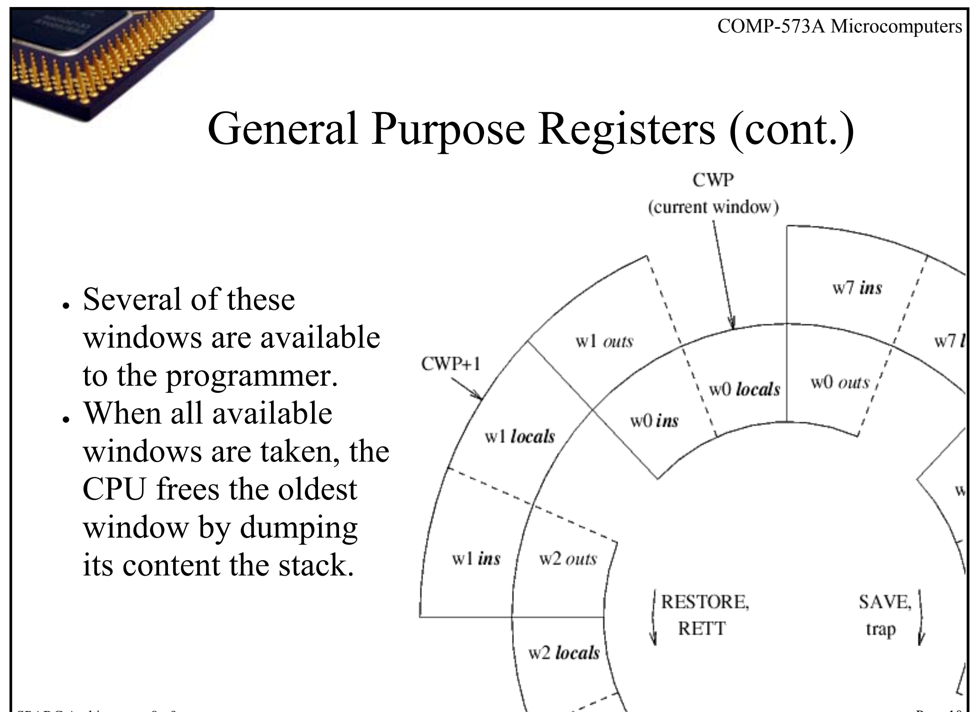
General Purpose Registers (cont.)

- A sliding window contains three types of registers:
  - **Input registers** : Arguments are passed to a function using these registers.
  - **Local registers** : The programmer can use these registers to store any local data.
  - **Output registers** : When calling a function, the programmer puts his argument in these registers.
- Programmers have access to 8 registers of each type.
- However, some of these registers have a special purpose and should not be used to store local data.

General Purpose Registers (cont.)

- When a function is called, the register window "slides".
  - Output registers become input registers
  - A new set of local registers and output registers are provided.

window (CWP + 1)	window CWP
r[31] : *ins* r[24]	
r[23] : *locals* r[16]	
r[15] : *outs* r[8]	r[31] : *ins* r[24]
	r[23] : *locals* r[16]
	r[15] : *outs* r[8]



General Purpose Registers (cont.)

- Increasing the number of register on the processor doesn't change the architecture, it just increases the number of available sliding windows.
- This type of architecture is very effective in situations with many function calls.
- When the OS must change context (change of running process), all the registers must be dumped to memory and the previous register values must be restores.
- The identity of the current sliding window is kept by 5 bits in the status register.

General Purpose Registers (cont.)

- As mentionned previously, some registers have special purposes:

```
Global Register     Note  Purpose
%g0     (r00)             always zero
%g1     (r01)       [1]   temporary value
%g2     (r02)       [2]   global 2
%g3     (r03)       [2]   global 3
%g4     (r04)       [2]   global 4
%g5     (r05)             reserved for SPARC ABI
%g6     (r06)             reserved for SPARC ABI
%g7     (r07)             reserved for SPARC ABI

Global Register           Note      Purpose
%l0-%l7  (r16)-(r23)      [3]       local 0-7

[1] assumed by caller to be destroyed (volatile) across a
    procedure call
[2] should not be used by SPARC ABI library code
[3] assumed by caller to be preserved across a procedure  call
```

```
Out Register        Note  Purpose
%o0      (r08)      [3]   outgoing parameter 0 / return
                          value from callee
%o1      (r09)      [1]   outgoing parameter 1
%o2      (r10)      [1]   outgoing parameter 2
%o3      (r11)      [1]   outgoing parameter 3
%o4      (r12)      [1]   outgoing parameter 4
%o5      (r13)      [1]   outgoing parameter 5
%o6,%sp  (r14)      [1]   stack pointer
%o7      (r15)      [1]   temporary value / address of CALL
                          instruction

In Register         Note  Purpose
%i0      (r24)      [3]   incoming parameter 0 / return
                               value to caller
%i1      (r25)      [3]   incoming parameter 1
%i2      (r26)      [3]   incoming parameter 2
%i3      (r27)      [3]   incoming parameter 3
%i4      (r28)      [3]   incoming parameter 4
%i5      (r29)      [3]   incoming parameter 5
%i6,%fp  (r30)      [3]   frame pointer
%i7      (r31)      [3]   return address - 8
```


The Stack

Stack area	Address
free space	
64 bytes reserved to save i0-i7 and l0-l7 in case of interrupt	%sp
4 by. hidden return	%sp + 64
24 byte reserved for first six arguments	
other parameters (after six)	%sp + 92
temporary space (if needed)	%sp + 92 + 4 * p
local variables	%fp – 4 * n
	%fp

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 92
  - If a function with more than 6 parameters will be used, add 4 bytes for each extra parameter (over 6).
  - Add 4 bytes for each local variable you want to create.
  - Add any amount of temporary space you want (multiple of 4 bytes).
  - The final number must be a multiple of 8 (pad with an extra 4 bytes of temporary space if it is not).

The Stack (cont.)

- The first 64 bytes of the reserve space is used to save the values of %i0-%i7 and %l0-%l7 if we run out of sliding windows.
- Normally, the return value of a function will be found in %o0. However, if a structure value will be return, the value of the structure to fill should be place in the hidden return slot before the call.
- When a function is called, the first six parameters are placed in the registers %o0-%o5. However, some operations cannot be executed over registers. Arguments can alternatively be passed on the stack using these last 6 reserved bytes.

The Stack (cont.)

- When calling a function with more than 6 parameters, additional parameters should be placed on the stack.
- Local variables and temporary space are easier to address using the frame pointer (fp).

Addressing Memory

- Register Indirect with Index

  The effective address is calculated by adding the contents two integer registers. Used for array access.

```
ld   [r1+r2], r3
```

- Register Indirect with Displacement

  The effective address is calculated by adding a 13 bit signed integer constant to a register. Used with pointers to structures or to access the stack.

```
st   r3, [r1+12]
```

Instruction Set

- Unlike Intel x86, instructions only come in one 32 bit flavor (except for floating-point operations).
- Most operation (other than load and store) can only be done over registers.
- A completely different set of instructions and registers is provided for floating point arithmetic.
- Some instructions are mnemonics for other instructions (alias). Notable examples include: set, not, neg, call, jmp and cmp.

Load and Store

- As mentioned before, the SPARC is a load and store architecture. Many operation cannot be done over memory.
  - The logic behind this is simple : it is often faster to load, execute your operation and store than to provide one huge instruction to do all three operations.
- Brackets [ ] are used to refer to the content of memory at a particular address.

```
st          r3,[r1]   --   Store the value of r3 at
                      -- memory address r1
ld          [r1],r3   --   Load the content of memory
                      -- at r1 into register r3
```

- The move instruction should be use to move data from one register to another.

Loading Constants

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

```
sethi   %hi(.LLC0), %o0         --set 22 first bits
or      %o0, %lo(.LLC0), %o0     --set 10 last bits
```

- The %hi and %lo keywords allow use to isolate a specific part of an address without the need to use bit shifting.
- The set instruction can be use instead. It is not real SPARC instruction but a keyword interpreted by the compiler. The final machine code will still use the two mentioned instructions.

```
set FPzero, %i4
set string1, %o0
```

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.
- Logical operations such as AND or NOT are not signed.
- Unlike Intel x86, most arithmetic and logic operation require a destination register.

```
add r1, r2, r3          -- r3 = r1 + r2
sub r1, r2, r3          -- r3 = r1 - r2
and r1, r2, r3          -- r3 = r1 AND r2
neg r1, r2              -- r2 = - r1
```

Multiplication and Division

- Unlike most arithmetic operations, multiplication and division are available in signed and unsigned mode.
- Integer multiplication and division use a special %y register.
  - For multiplication, the %y register is used to old the higher order bits of the results.
    For division, the %y register is used to old the higher order bits of the number behind divided. After the division, the remainder can be found in the %y register.

Multiplication and Division (cont.)

```
smul    %r2, %r3, %r2      -- r2 * r3 --> y & r2
udiv    %r2, %r3, %r2      -- y & r2 / r3 --> r2
                           -- remainder in y
```

- Remember to clear %y register before a multiplication or a division if you are not using it to store higher-order bytes.
  - **Warning!** It takes three cycles to access (read/write) the %y register. That means there must be at least 3 instructions (cycle) between any two instructions that use the %y register.

Floating-point instructions

- A SPARC V8 processor is equipped with 32 floating-point registers. These can be use to hold:
  - 32 single-precision number (1 register each)
  - 16 double-precision number (2 registers each)
  - 8 quad-precision number (4 registers each)
  - Any combination of the above ...
- Operations over these three types of floating-point numbers are provided.
- Operations to convert integer to floating-point or vice-versa are also provided.

Floating-point instructions (cont.)

- Operations to move date from general purpose registers to floating-point registers are not allowed.
- If you receive float-point as arguments (%i0-%i5), you will need to put them in memory temporarily to transfer them to a floating-point register.
- The type of rounding used is determined by two bytes in the Floating-Point State Register (30 and 31).
- Special purpose mathematical instructions are provided on chip (such as square root).

```
st %i2,[%fp-8]            -- Save temp to mem
ld [%fp-8],%f3            -- f3 = i2
fadds %f2,%f3,%f5         -- f5 = f2 + f3
```

Jump Instructions

- Jump instructions are similar to those found on the Intel x86.
- However, every branch instruction on the SPARC architecture has a delay slot.
  - The instruction place after a branch instruction will be execute before the branch. The instruction in a delay slot must only take 1 cycle to complete.
  - If the annul bit is set on a conditionnal branch (",a" is added to the instruction), then the instruction in the delay slot will not be executed if the branch is not taken.

```
call dofunction,0
add  #i1, #i2, %i3    <- Executed before call

bne,a label           <- Annul bit
add  #i1, #i2, %i3
```

Stack Instructions

- As mentioned before, the programmer must, at the beginning of a function, allocate his stack frame.
- This can be done using the save instruction.
- Calling the save instruction also slides the register window.
- A stack frame must never be smaller than 96 bytes.
- The programmer must deallocate the stack frame at the end of his function using the restore instruction.

```
save    %sp, -112, %sp -- Save stack frame of 112
                       --   bytes and slides the
                       --   register window
...
restore                -- Deallocate stack frame
                       -- and slides back the
                       --   register window
```

Functions Instructions

- The **call** instruction can be use to branch the execution of our program to a new function.
- The **ret** instruction can be used to return the execution of our program to the previous function.
- Please note that the call and ret instruction are both branching instruction, thus they have delay slots.

Instruction Reference

- Here are some of the most common Sparc assembly instructions.
- The descriptions were taken from “The SPARC Architecture Manual, Version 8” and “The SPARC Architecture Manual, Version 9”

Move Register

SPARC

Syntax:

mov reg2 or imm11, regd

Operation:

regd ← reg2 or imm11

Description:

This instruction copies an integer register to another integer register. It does not modify any condition codes.

Load Word

SPARC

Syntax:

ld [address], regd

Operation:

regd ← [address]

Description:

The load word instruction copies a word from memory into r[d]. The effective address for a load instruction is either "r[1] + r[2]" or "r[1] + sign_ext (simm13)".

A successful load instruction operates atomically.

The ld instruction can also be used to load single floating-point numbers to floating point registers. However, to load double or quad floating-point numbers, the ldd and ldq instructions must be used.

Store Word

Syntax:

st regrd, [address]

Operation:

[address] ← regd

Description:

The store integer instructions copies the whole 32-bit integer register into memory. The effective address for a store instruction is either “r[1] + r[2]” or “r[1] + sign_ext (simm13)”.

A successful store instruction operates atomically.

The st instruction can also be used to store single floating-point numbers to memory. However, to store double or quad floating-point numbers, the std and stq instructions must be used.

Add

Syntax:

add reg1, reg2 or imm13, regd

Operation:

regd ← reg1 + reg2 or imm13

Description:

The add instruction computes “r[1] + r[2]” or “r[1] + sign_ext(simm13)”, and write the sum into r[d].

Subtraction

Syntax:

sub reg1, reg2 or imm13, regd

Operation:

regd $\leftarrow$ reg1- reg2 or imm13

Description:

The sub instruction computes “reg[1] - reg[2]” or “reg[1] - sign_ext(simm13)”, and write the sum into reg[d].

Signed Multiplication

Syntax:

smul reg1, reg2 or imm13, regd

Operation:

Y:regd $\leftarrow$ reg1 * reg2 or imm13

Description:

The multiply instruction performs 32-bit by 32-bit multiplication, producing 64-bit results. It writes the 32 most significant bits of the product into the Y register and the 32 least significant bits into r[*d*].

A signed multiply assumes signed integer word operands and computes a signed integer doubleword product.

Unsigned Multiplication

Syntax:

umul reg1, reg2 or imm13, regd

Operation:

Y:regd ← reg1 * reg2 or imm13

Description:

The multiply instruction performs 32-bit by 32-bit multiplication, producing 64-bit results. It writes the 32 most significant bits of the product into the Y register and the 32 least significant bits into r[*d*].

An unsigned multiply assumes unsigned integer word operands and computes an unsigned integer doubleword product.

Signed Division

Syntax:

sdiv reg1, reg2 or imm13, regd

Operation:

regd ← y:reg1 / reg2 or imm13

Description:

The divide instruction performs a 64-bit by 32-bit division, producing a 32-bit result. The integer quotient are sign-or zero-extended to 32 bits and are written into r[d]. On some processors, the remainder can be found in the y register.

Signed division rounds an inexact quotient toward zero. For example, –7 / 4 equals the rational quotient of –1.75, which rounds to –1 (not –2) when rounding toward zero.

Unsigned Division

SPARC

Syntax:

udiv reg1, reg2 or imm13, regd

Operation:

regd ← y:reg1 / reg2 or imm13

Description:

The divide instruction performs a 64-bit by 32-bit division, producing a 32-bit result. This operation assumes reg1 and reg2 to be unsigned words. The integer quotient are written into r[d]. On some processors, the remainder can be found in the y register.

Unsigned division rounds an inexact rational quotient toward zero.

Logical And

SPARC

Syntax:

and reg1, reg2 or imm13, regd

Operation:

regd ← reg1 & reg2 or imm13

Description:

This instruction implements the bitwise logical AND operation.

Other available logical operators include:

- andn : And Not
- or : Inclusive Or
- orn : Inclusive Or Not
- xor : Exclusive Or
- xorn : Exclusive Or Not

Set High

SPARC

Syntax:

sethi %hi (imm22), regd

Operation:

regd ← 0

[10:31]regd ← imm22

Description:

SETHI zeroes the least significant 10 bits of regd, and replaces bits 31 through 10 of regd with the value from its imm22 field.

SETHI does not affect the condition codes.

Branch

Syntax:

ba{,a} label

Operation:

PC ← PC + (4 * sign_ext(disp22))

Description:

BA (Branch Always) causes an unconditional PC-relative, delayed control transfer to the address “PC + (4 * sign_ext(disp22)).”

If the annul field of the branch instruction is 1, the delay instruction is annulled (not executed). If the annul field is 0, the delay instruction is executed.

Note that the annul bit has a **different** effect on conditional branches than it does on unconditional branches.

Integer Conditional Branch

SPARC

Syntax:

bcc{,a} label

Operation:

If cc == true then

PC ← PC + (4 * sign_ext(disp22))

Description:

Conditional Bicc instructions evaluate the 32-bit integer condition codes (cc), according to the cond field of the instruction, producing either a TRUE or FALSE result. If TRUE, the branch is taken, that is, the instruction causes a PC-relative, delayed control transfer to the target address. If FALSE, the branch is not taken.

If a conditional branch is taken, the delay instruction is always executed regardless of the value of the annul field. If a conditional branch is not taken and the a (annul) field is 1, the delay instruction is annulled (not executed).

Call and Link

SPARC

Syntax:

call label

Operation:

r[15] ← PC

PC ← PC + (4 * sign_ext(disp30))

Description:

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to address PC + (4 * sign_ext(disp30)). Since the word displacement (disp30) field is 30 bits wide, the target address lies within a range of -2^{31} to $+2^{31} - 4$ bytes.

The CALL instruction also writes the value of PC, which contains the address of the CALL, into r[15] (out register 7). The value written into r[15] is visible to the instruction in the delay slot.

Save Caller's Window

SPARC

Syntax:

save {reg1, reg2_or_imm13, regd}

Operation:

save register window

regd ← reg1 + reg2 or imm13

Description:

The SAVE instruction provides the routine executing it with a new register window. The out registers from the old window become the in registers of the new window. The contents of the out and the local registers in the new window are zero or contain values from the executing process; that is, the process sees a clean window.

Furthermore, SAVE behave like normal ADD instructions, except that the source operands r[rs1]and/or r[rs2] are read from the **old** window (that is, the window addressed by the original CWP) and the sum is written into r[rd] of the **new** window (that is, the window addressed by the new CWP).

Restore Caller's Window

SPARC

Syntax:

restore {reg1, reg2_or_imm13, regd}

Operation:

save register window

regd ← reg1 + reg2 or imm13

Description:

The RESTORE instruction restores the register window saved by the last SAVE instruction executed by the current process. The in registers of the old window become the out registers of the new window. The in and local registers in the new window contain the previous values.

Furthermore, RESTORE behave like normal ADD instructions, except that the source operands r[rs1]and/or r[rs2] are read from the **old** window (that is, the window addressed by the original CWP) and the sum is written into r[rd] of the **new** window (that is, the window addressed by the new CWP).

Return

SPARC

Syntax:

ret

Operation:

PC ← r[31]

Description:

The CALL instruction causes an unconditional, delayed, PC-relative control transfer to the address held in r[31]. This effectively ends a function call and returns control flow to its previous function.

Floating-Point Addition

SPARC

Syntax:

fadd(s,d,q) freg1, freg2, fregd

Operation:

fregd ← freg1 + freg2

Description:

The floating-point add instructions add the floating-point register(s) specified by the reg1 field and the floating-point register(s) specified by the reg2 field, and write the sum into the floating-point register(s) specified by the regd field.

Rounding is performed as specified by the FSR.RD field.

Floating-Point Subtraction

SPARC

Syntax:

fsub(s,d,q) freg1, freg2, fregd

Operation:

fregd ← freg1 - freg2

Description:

The floating-point subtract instructions subtract the floating-point register(s) specified by the *reg2* field from the floating-point register(s) specified by the *reg1* field, and write the difference into the floating-point register(s) specified by the *rd* field.

Rounding is performed as specified by the FSR.RD field.

Floating-Point Compare

SPARC

Syntax:

fcmp(s,d,q) freg1 , freg2

Operation:

compare freg1, freg2

Description:

These instructions compare the f register(s) specified by the freg1 field with the f register(s) specified by the freg2 field, and set the floating-point condition codes.

Convert Floating-Point to Integer SPARC

Syntax:

f(s,d,q)toi freg2, fregd

Operation:

fregd ← (integer)freg2

Description:

FsTOi, FdTOi, and FqTOi convert the floating-point operand in the floating-point register(s) specified by freg2 to a 32-bit integer in the floating-point register specified by fregd.

The result is always rounded toward zero; that is, the rounding direction (RD) field of the FSR register is ignored.

Convert Integer to Floating-Point SPARC

Syntax:

fito(s,d,q) freg2, fregd

Operation:

fregd ← (float)freg2

Description:

FiTOs, FiTOd, and FiTOq convert the 32-bit signed integer operand in floating-point register(s) specified by freg2 into a floating-point number in the destination format. All write their result into the floating-point register(s) specified by fregd.

FiTOs, FiTOd, and FiTOq round as specified by the FSR.RD field.

Floating-Point Move

SPARC

Syntax:

fmov(s,d,q) freg2, fregd

Operation:

fregd ← freg2

Description:

FMOV copies the source to the destination unaltered.

This instruction do not round.

Floating-Point Negate

SPARC

Syntax:

fneg(s,d,q) freg2, fregd

Operation:

fregd ← -freg2

Description:

FNEG copies the source to the destination with the sign bit complemented.

This instruction do not round.

Floating-Point Absolute Value SPARC

Syntax:

fabs(s,d,q) freg2, fregd

Operation:

fregd ← [0:0]0 :: [1:31] freg2

Description:

FABS copies the source to the destination with the sign bit cleared.

This instruction do not round.

Floating-Point Multiply SPARC

Syntax:

fmul(s,d,q) freg1, freg2, fregd

Operation:

fregd ← freg1 * freg2

Description:

The floating-point multiply instructions multiply the contents of the floating-point register(s) specified by the freg1 field by the contents of the floating-point register(s) specified by the freg2 field, and write the product into the floating-point register(s) specified by the fregd field.

Rounding is performed as specified by the FSR.RD field.

Floating-Point Divide

SPARC

Syntax:

fdiv(s,d,q) freg1, freg2, fregd

Operation:

fregd ← freg1 / freg2

Description:

The floating-point divide instructions divide the contents of the floating-point register(s) specified by the freg1 field by the contents of the floating-point register(s) specified by the freg2 field, and write the quotient into the floating-point register(s) specified by the fregd field.

Rounding is performed as specified by the FSR.RD field.

Floating-Point Square Root

SPARC

Syntax:

fsqrt(s,d,q) freg2, fregd

Operation:

fregd ← $freg2^{1/2}$

Description:

These instructions generate the square root of the floating-point operand in the floating-point register(s) specified by the freg2 field, and place the result in the destination floating-point register(s) specified by the fregd field.

Rounding is performed as specified by the FSR.RD field.

No Operation

Syntax:

nop

Operation:

r[0] ← 0

Description:

The NOP instruction changes no program-visible state (except the PC and nPC).

Struct. of an Assembly Prog.

```
.section    ".rodata"
        .align 8
.LLC0:
        .asciz  "Hello World"
```

Constants Declaration

```
.section       ".text"
        .align 4
```

Code Alignment

```
        .global main
        .type    main,#function
        .proc    04
```

Function Declaration

```
main:
        save     %sp, -112, %sp
        sethi    %hi(.LLC0), %o0
        call     printf, 0
        or       %o0, %lo(.LLC0), %o0
        ret
        restore
```

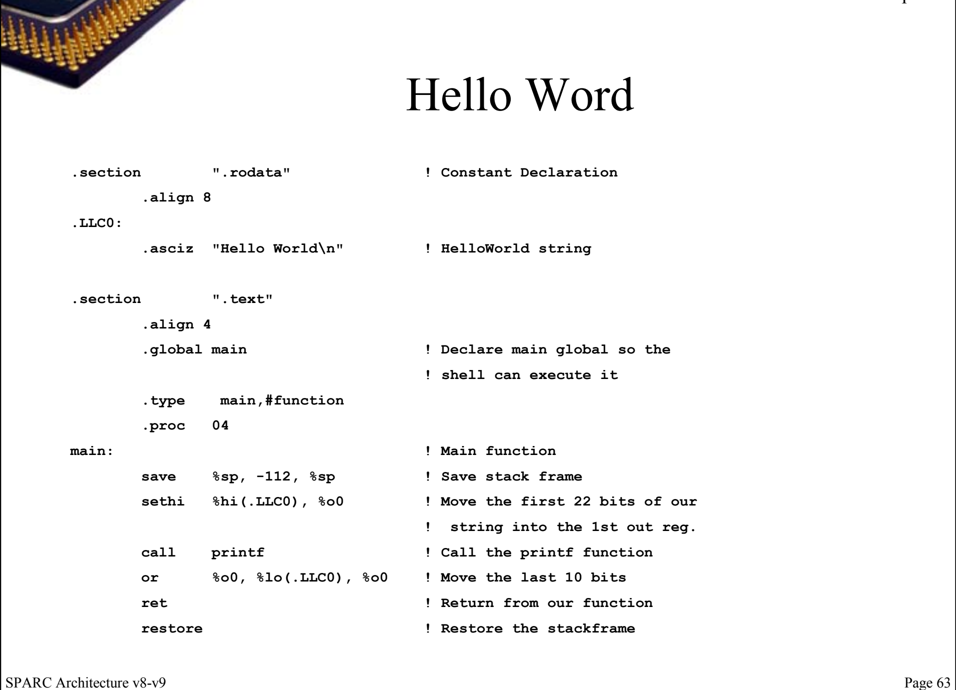
Function Code

Constants Declaration

- Constants should be declare as the beginning of the file (though gcc sometimes puts them at the end).
- Very similar to Intel x86.
- Most common type of constants are
  - ascii
  - asciz (null terminated)
  - byte (1 byte)
  - word (4 byte)
  - single (float)
  - double (more precise float)

Code Alignment

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



Temporary Variables and Arithmetic

```
.section    ".text"
      .align 4
      .global main                ! Declare main global so the
                                  ! shell can execute it
      .type    main,#function
      .proc   04
main:                             ! Main function
      save    %sp, -128, %sp      ! Save stack frame
      mov     5, %o0              ! temp = 5
      st      %o0, [%fp-20]       ! x = temp
      mov     6, %o0              ! temp = 6
      st      %o0, [%fp-24]       ! y = temp
      mov     7, %o0              ! temp = 7
      st      %o0, [%fp-28]       ! z = temp
      ld      [%fp-20], %o0       ! temp1 = x
      ld      [%fp-24], %o1       ! temp2 = y
      add     %o0, %o1, %o0       ! temp1 = temp1 + temp2
      st      %o0, [%fp-28]       ! z = temp1
      ld      [%fp-28], %i0       ! Prepare to return z
      ret                         ! Return from our function
      restore                     ! Restore the stackframe
```

If statement

C Code:

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

Assembler:

```
 mov     5, %o0                 /* i = 5 */
        cmp     %o0, 0                 /* temp = i - 0 */
        bne     .Outside_If            /* if i != 0 goto Outside_If */
        nop
       /* Inside If */
.Outside_If:
```


For statement

C Code:

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

Assembler:

```
 st      %g0, [%fp-20]          /* j = 0 */
.Begin_For:
        ld      [%fp-20], %o0          /* temp = j */
        cmp     %o0, 14                /* temp2 = j - 14 */
        ble     .Inside_For            /* if temp2 <= 0 goto Inside_For */
        nop
        b       .Outside_For           /* goto outside for */
        nop
.Inside_For:
       /* Inside For */
        ld      [%fp-20], %o0          /* temp = j */
        add     %o0, 1, %o1            /* j = j + 1 */
        st      %o1, [%fp-20]          /* j = temp */
        b       .Begin_For
        nop
.Outside_For:
        mov     0, %i0                 /* Prepare to return 0 */
```

Evolution of the SPARC

- The V9 architecture is the successor to the V8 architecture studied in class.
- As mentioned before, it was release in 1993, just 3 years after the release of the V8 architecture.
- The V9 architecture provides several enhancement over the V8 architecture:
  - 64- bit virtual address
  - 64-bit integer data
  - Addition of 32 single floating-point registers (or 16 double)
  - Improved parallelism (ex: 4 fp operations simultaneously)
  - New instructions (ex: 64-bit multiply and divided)

Evolution of the SPARC (cont.)

- Branches on register value (eliminating the need to compare)
- Conditional moves (removes the need for many branches)
- The V9 architecture has many fault tolerance / parallelism features built-in such as compare and swap instructions.
- The V9 achieves all this, and remains binary compatible with all previous SPARC architecture.

References

- SPARC International Inc.
  http://www.sparc.com/
- The SPARC Architecture Manual Version 8
- The SPARC Architecture Manual Version 9
- A Laboratory Manual for the SPARC Revision: 1.2
  http://www.cs.unm.edu/%7Emaccabe/classes/341/labman/labman.html
- Rice Universisty Comp 320, Fall 2000, Subset of SPARC V8/V9 Assembly Language
  http://www.owlnet.rice.edu/%7Ecomp320/2001/assignments/sparc_subset.html
- SPARC stack frame information
  http://compilers.iecc.com/comparch/article/91-04-038
- Understanding stacks and registers in the SPARC architecture(s)
  http://www.sics.se/%7Epsm/sparcstack.html