Eidgenössische Technische Hochschule Zürich

Programming in Systems (37-023)

Programming in Assembler

Basics of Operating Systems

Models of Computer Architecture

Lecturer today:

Prof. Thomas M. Stricker

Text-/Reference-Books:

R.P.Paul: SPARC Architecture... and C

Sun SPARC V8 Manual and K&R C Reference

Topics of Today:

- **Standard/Leaf Subroutines**
- **SPARC Calling Convention**
- **Arguments on the Stack**
- **Pointer as Arguments**

Subroutines

- Basic tool for structured and object oriented programming.
- Functions in C - procedures in Oberon - Methods in C++/Java.
- Code pieces that can be called again with different arguments:
  -> code reuse, more compact code.
- Called with a jump instruction from main program or from subroutine itself.
- Subroutines are transparent when it comes to register use. Content of registers must be saved on the stack for subroutine call.
- Arguments to the subroutine are handled like local variables.
- Open subroutines are like a macro, where arguments are substituted.

Calling a subroutine (Basics)

- The address of the calling instruction in the “main program” is saved. It is the so called return address.
- The SPARC architecture saves that address into %o7 and passes it as %i7 to the subroutine (details later).
- Calls to subroutines are **delayed** like branches.
- The actual destination for the return is therefore %o7 + 8 (two 32bit instructions after the call).
- If the name of the subroutine is given by a label (some fixed location in the code) the call is made with a call instruction:

call <name>

- stores the %pc in %o7 and provides a delay slot

- If the address of a subroutine is calculated at run time the invocation is done by

jmpl <source_reg>, <dest_reg>

- This is necessary for calling the contents of a function variable or for calling virtual methods of classes (inheritance)
- Calls a function a the address provided by the register <source_reg>.
- Saves the %pc in <dest_reg>

jmpl %o0, %o7

is semantically equivalent to

call (%o0) !this is illegal syntax

- The return from a subroutine is also done with the jump&link mechanism as:

jmpl %i7 + 8, %g0 /* discards the %pc */

which is equivalent to the **ret** instruction

Save / Restore (temporary definition)

save %sp, -64, %sp

- stores the registers %l0-%l7, %i0-%i7 to the stack (into the 64 bytes provided)
- copies the regs %o0-%o7- > %i0-%i7:
- does an addition (mostly used to get more space on the stack).

restore

- copies the regs %o0-%o7 <- %i0-%i7:
- load the contents of the registers %l0-%l7, %i0-%i7 back from the stack.
- also performs an add instructions which is normally left unused (except for gcc which uses it when it can!)

  e.g. restore %g2,12,%g2

Typical Subroutine Call

```
1   main:
2         ...
3         call mysubr
4         nop                    !or something meaningful
5         ...

6   mysubr:
7         save %sp,-112,%sp
8         ...
9         ret
10        restore                !fills delay slot of ret
```


Passing of Arguments

- Must allow arguments that are calculated as expressions.
- Must allow recursion.

  So hard coding is not a solution!

- Passing all arguments by the stack is slow since this must go by main memory (or at least by the stacks).
- Therefore the SPARC architecture uses the registers %o0,..,%o5 to pass arguments across subroutine calls.

  note that %o6 holds the stack pointer and %o7 the return address.

- If more than six arguments must be passed the stack is used
- for larger data structure, the stack is used.

SPARC Calling Convention

- Determines the organization of the stack frame during subroutine calls.
- Allows separate compilation and linking. This permits C programs that call assembly routines.
- Stack is organized as follows:
  - **64 bytes** for the storage of the register contents at a save instruction.
  - **4 bytes** for a struct pointer, that permits the return of structs
  - **24 bytes** for the first six parameters that are usually passed in registers (the space is still reserved for GDB)
  - **4 bytes** for padding (alignment)
  - **n bytes** for the local variables of the subroutines.
- Don’t forget: The stack pointer must always be doubleword aligned

Example

- General form

```
save   %sp, -(64 + 4 + 24 + n) & -8, %sp
```

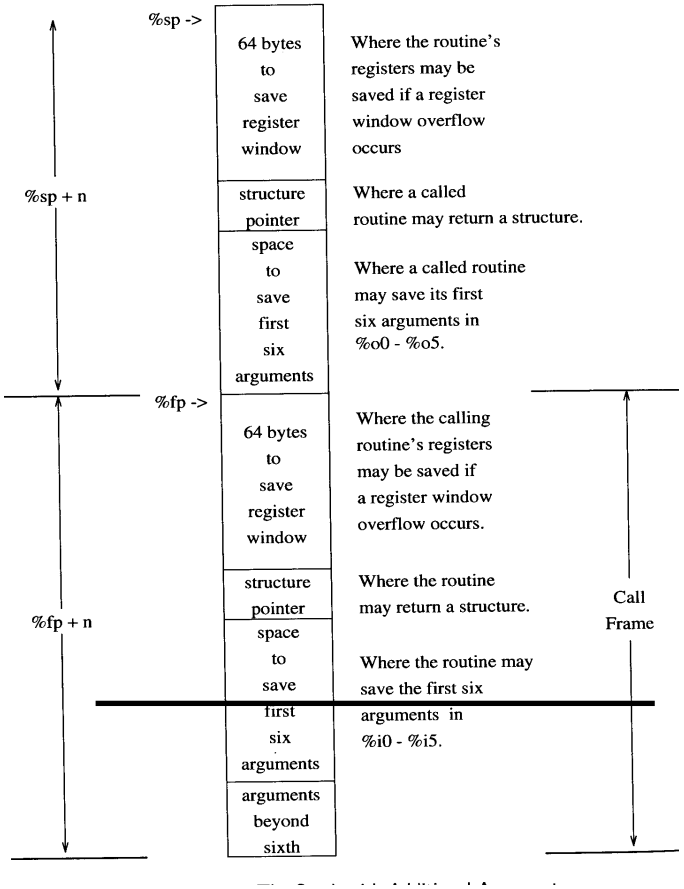
- Example

```
vector()
int a,b;
char d;

save   %sp, -(64 + 4 + 24 + 9) & -8, %sp
```

- Addressing convention:
  - The addresses of local variables are negative with respect to %fp.
- All other data in the stack are positive with respect to %sp.

Picture of the stack



Return from a Subroutine

- Subroutines, that return arguments are called functions:
- Example in C:

```
int example(int a, int b, char c);

main() {
  int r;

  r = example(3, 5, 4);
  printf("%d\n", r);
}

int example(int a, int b, char c)
{
  int x, y;
  short ary[128];

  register int i, j;

  x = a + b;
  i = c + 64;
  ary[i] = c + a;
  y = x * a;
  j = x + i;
  return x + y;
}
```


Example in Assembler:

```
                                        ! a_r in %i0
                                        ! b_r in %i1
                                        ! c_r in %i2
   x_s = -4
   y_s = -8
   ary_s = -256
                                        !i_r in %l0
                                        !j_r in %l1
_example:
   save   %sp, -360, %sp
   add    %i0, %i1, %o0                 !x = a + b
   st     %o0, [%fp + x_s]
   add    %i2, 64, %l0                  !i = c + 64
   add    %i0, %i2, %o0                 !ary[i] = c + a
   sll    %l0, 1, %o1
   add    %fp, ary_s, %o2
   sth    %o0, [%o1 + %o2]
   ld     [%fp + x_s], %o0              !y = x * a
   mov    %i0, %o1
   smul   %o0, %o1, %o0
   st     %o0, [%fp + y_s]
   ld     [%fp + x_s], %o0              !j = x + i
   add    %l0, %o0, %l1
   ld     [%fp + x_s], %o0
   ld     [%fp + y_s], %o1              !return x + y
   ret
   restore  %o0, %o1, %o0
```

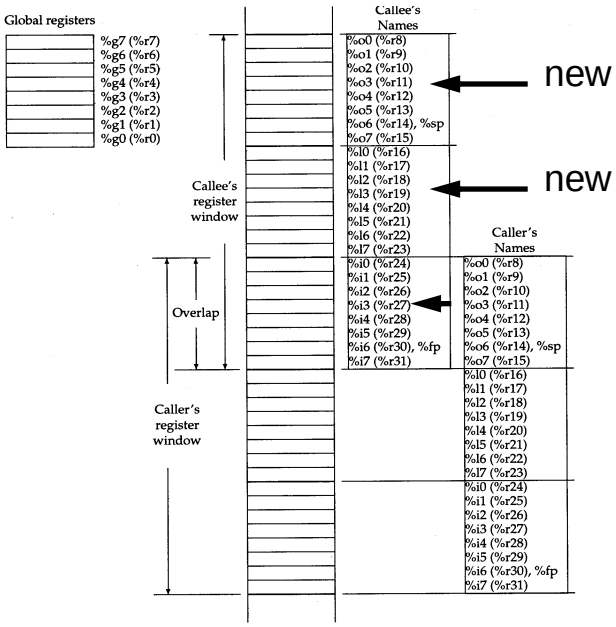
Leaf Subroutines

- Do not contain any further subroutine calls. The call can be optimized. Save/ restore instructions can be avoided.
- Leaf subroutines must use the register set of the calling routine.
- Restrictions: Use of %o0 to %o5 and %g0 to %g1 is permitted.
- Call works as for subroutines - but with %o7 instead of %i7. Therefore return to %o7+8!

```
   retl              ==        jmpl  %07+8, %g0

1  foo:   add      %o1, %o0, %o0
2         add      %o2, %o0, %o0
3         add      %o3, %o0, %o0
4         add      %o4, %o0, %o0
5         add      %o5, %o0, %o0
6         ld       [%sp + arg7_s], %o1
7         add      %o1, %o0, %o0
8         ld       [%sp + arg8_s], %o1
9         retl
10        add      %o1, %o0, %o0
```


Register renaming



- Save
- Restore

Register Names

- Alternative names of registers refer to the use in subroutine calls:

```
%gx    Global Register
%lx    Local Register in Subroutine
%ix    Input Register in Subroutine
%ox    Output Register in Subroutine


%f0-%f31:     %f0-%f31
%r0-%r7:      %g0-%g7
%r8-%r15:     %o0-%o7
%r16-%r23:    %l0-%l7
%r24-%r31:    %i0-%i7

%sp:          %r14, %o6
%ret:         %r15, %o7
%fp:          %r30, %i6
```


SPARC calling convention

	%i7 (%r31)	return address – 8 †
	%fp, %i6 (%r30)	frame pointer †
	%i5 (%r29)	incoming param 6 †
in	%i4 (%r28)	incoming param 5 †
	%i3 (%r27)	incoming param 4 †
	%i2 (%r26)	incoming param 3 †
	%i1 (%r25)	incoming param 2 †
	%i0 (%r24)	incoming param 1 / return value to caller †
	%l7 (%r23)	local 7 †
	%l6 (%r22)	local 6 †
	%l5 (%r21)	local 5 †
local	%l4 (%r20)	local 4 †
	%l3 (%r19)	local 3 †
	%l2 (%r18)	local 2 †
	%l1 (%r17)	local 1 †
	%l0 (%r16)	local 0 †
	%o7 (%r15)	temporary value / address of CALL instruction ‡
	%sp, %o6 (%r14)	stack pointer †
	%o5 (%r13)	outgoing param 6 ‡
out	%o4 (%r12)	outgoing param 5 ‡
	%o3 (%r11)	outgoing param 4 ‡
	%o2 (%r10)	outgoing param 3 ‡
	%o1 (%r9)	outgoing param 2 ‡
	%o0 (%r8)	outgoing param 1 / return value from callee ‡
	%g7 (%r7)	global 7 (SPARC ABI: use reserved)
	%g6 (%r6)	global 6 (SPARC ABI: use reserved)
	%g5 (%r5)	global 5 (SPARC ABI: use reserved)
global	%g4 (%r4)	global 4 (SPARC ABI: global register variable §)
	%g3 (%r3)	global 3 (SPARC ABI: global register variable §)
	%g2 (%r2)	global 2 (SPARC ABI: global register variable §)
	%g1 (%r1)	temporary value ‡
	%g0 (%r0)	0
state	%y	Y register (used in multiplication/division) ‡
	(*icc* field of %psr)	Integer condition codes ‡
	(*fcc* field of %fsr)	Floating-point condition codes ‡
	(*ccc* field of %csr)	Coprocessor condition codes ‡
	%f31	floating-point value ‡
floating point	:	:
	:	:
	%f0	floating-point value ‡

Calling Convention

SPARC

- %g0 NULL
- %g1 temp, caller saved
- %g2..g4 global, callee must save
- %g5..g7 global, reserved
- %l0..%l7 local, callee must save with “save” (unless leaf proc.)
- %i0..%i5 in parametes, caller must save (unless leaf proc.)
- %o0..%o5 local, callee must save with “save”(unless leaf proc.)
- %o6, %o7 stack-pointer, return(tmp)
- %i6,%i7 frame-pointer, ret-addr
- %f0..%f31 caller saves

Alternatives: Simple Calling Conventions

Other processors:

- MIPS
- 68000
- iWarp

Register contents are always stored on the stack:

- Argument registers
- Caller saved registers
- Callee saved registers
- Scratch registers
- Interrupt registers

Pointer as Arguments (C)

- Call by reference paradigm (as contrast to call by value)
- Need to pass pointers as arguments to subroutines.
- Classical example swap(x,y)

```
swap(int *x, int *y)
{
    int temp;

    temp = *x;
    *x = *y;
    *y = temp;
}
```


Pointer as Arguments (assembly language)

```
    x_s = -4      !int x                  !local variables
    y_s = -8      !int y

_main:                                    !bsp54.s
    save   %sp, -104, %sp                 !2 words for arguments
    mov    5, %o0
    st     %o0, [%fp + x_s]                    !x = 5
    mov    7, %o0
    st     %o0, [%fp + y_s]                    !y = 7
    add    %fp, x_s, %o0                  !pointer to x in %o0
    call   _swap
    add    %fp, y_s, %o1                  !pointer to y in %o1
    ret
    restore
    ta     0

    .global _swap                         ! a leaf routine
_swap:
    ld     [%o0], %o2                          !%o2 = x
    ld     [%o1], %o3                          !%o3 = y
    st     %o2, [%o1]
    retl
    st     %o3, [%o0]
```

Passing Arguments on the Stack

- Needed when more than 6 arguments are to be passed
- Caller must make space on the stack before calling subroutine

```
foo(1,2,3,4,5,6,7,8)

int foo(int a1, int a2, int a3, int a4, int a5,
int a6, int a7, int a8)
{
return a1+a2+a3+a4+a5+a6+a7+a8;
}
```

- Reservation of space for arg 7-8 by

```
add    %sp, -2*4 & -8, %sp
```

- a7 and a8 will be written at %sp+92 and %sp+96 to the stack (arg8_s =92, arg7_s=96)

Assembly Code

```
_main:                                          !bsp52.s
       save%  sp, -96, %sp
       add    %sp, -2 * 4 & -8, %sp
       mov    8, %o0
       st     %o0, [%sp + arg8_s]
       mov    7, %o0
       st     %o0, [%sp + arg7_s]
       mov    6, %o5
       mov    5, %o4
       mov    4, %o3
       mov    3, %o2
       mov    2, %o1
       call   _foo
       mov    1, %o0
       sub    %sp, -2 * 4 & -8, %sp
       mov    1, %g1
       ta     0

_foo:
    save   %sp, -96, %sp
    ld     [%fp + arg8_s], %o0
    ld     [%fp + arg7_s], %o1        !the 7th argument
    add    %o0, %o1, %o0
    add    %i5, %o0, %o0             !the sixth argument
    add    %i4, %o0, %o0              !the fifth argument
    add    %i3, %o0, %o0            !the fourth argument
    add    %i2, %o0, %o0              !the third argument
    add    %i1, %o0, %o0         !the second argument
    ret
    restore %i0, %o0, %o0             !the first argument
```


Returns of “structs”

- Structs can be returned by functions:

Example in C:

```
/* bsp50.c */

struct point {
  int x, y;
}

struct point zero() {
  struct point local;

  local.x = 0;
  local.y = 0;

  return local;
}

main()
{

  struct point x1, x2;

  x1 = zero();
  x2 = zero();
}
```


Assembly Code:

```
   !tkisem version bsp51.s
   !local variables
   x1 = -8
   x2 = -16
_main:
    save   %sp, -112, %sp              !double aligned
    mov    1, %l1
    st     %l1, [%fp+x1]               !x1.x = 1 for test
    mov    2, %l1
    st     %l1, [%fp+x1+4]             !x1.y = 2 for test
    add    %fp, x1, %o0          !%fp -8 is pointer to x1
    call   zero
    st     %o0, [%sp + 64]         !struct pointer adress
    add    %fp, x2, %o0
    call   zero
    st     %o0, [%sp + 64]
    ret
    restore
    ta     0

    x = 0
    y = 4
    .global zero
zero:
    save   %sp, -104, %sp
    ld     [%fp + 64], %o0         !get pointer into %o0
    st     %g0, [%o0 + x]
    st     %g0, [%o0 + y]
           ret
           restore
```