Basic Concepts: Instruction Execution

Each SPARC instruction is one word long (32-bits)
Info is encoded into each instruction.

```
01001101 10101010 10111001 00101101
```

OpCode Register Literal Data

Instructions are stored in memory with data.
Instructions are always word aligned.
Registers
"PC" - the program counter

The "fetch-increment-execute" loop:

```
PC = 0
loop
  instr = MEMORY [PC]
  PC = PC + 4
  Execute instr...
    • fetch operands
    • perform computations
    • store results (includes modifying PC)
endLoop
```

© Harry H. Porter, 2006

Machine Architecture Variations

Stack Architectures

Easy to compile to
Lots of memory accesses
Used mostly for interpreters

```
LVALUE z
PUSH   x
PUSH   y
ADD
ASSIGN
```

Accumulator Architectures

One general purpose register

```
LOAD  x
ADD   y
STORE z
```

CISC / Two-Operand Architectures

Several general-purpose registers
Two operand fields in instructions
The result overwrites one operand

```
LOAD  x,R3
ADD   y,R3
STORE R3,z
```

RISC / Load Store / Three-operand Architectures

Lots of general-purpose registers
Instructions have 3 operand fields
Each instruction is either
Computation
Memory access
Operands for computation must be in registers
Many instructions execute in a single clock cycle

```
LOAD  x,R1
LOAD  y,R2
ADD   R1,R2,R3
STORE R3,z
```

© Harry H. Porter, 2006

SPARC Registers

General-Purpose Registers

32 registers
32-bits (4bytes) each — *Well, okay to assume 32-bits...*
Divided into 4 sets of 8 registers

Global	`%g0, %g1, ... %g7`
Local	`%l0, %l1, ... %l7`
In	`%i0, %i1, ... %i7`
Out	`%o0, %o1, ... %o7`

Available operations:
- Integer arithmetic: add, sub, mul, div, cmp
- Logical: and, or, not, shift-left, shift-right

Floating-Point Registers

32 registers `%f0, %f1, ... %f31`
Available operations:
- Floating-point arithmetic: add, sub, mul, div, cmp
- Integer-to-floating conversion

© Harry H. Porter, 2006

Other Registers

Program Counter (PC)

32-bits

Integer Condition Code Register

4-bits
For integer operations

Floating-Point Condition Code Register

4-bits
For floating-point operations

"Y" Register

32-bits
Used for integer mulitply
and divide operations

```
                3A0F 2C33
          x     0145 0000
     ----------------------
     0049 B543  1CBF 0000
```

Y reg *Normal Reg*

Other Registers

Lots - ignore them

© Harry H. Porter, 2006

Example SPARC Instructions

```
sub     %g2,%g7,%g5
```

Operand 1 (reg1)

Operand 2 (reg2)

Result / Destination (regD)

```
1000 1010 0010 0000 1000 0000 0000 0111
```

regD — reg1 — reg2

```
sub     %g2,7,%g5
```

```
1000 1010 0010 0000 1010 0000 0000 0111
```

regD — reg1 — Literal data value

© Harry H. Porter, 2006

13-Bit Immediate Values

The instruction includes a 13-bit signed value.

Range: `-4096 .. 4095`

This value is "sign-extended" to 32-bits.

Example:

```
---- ---- ---- ---- ---0 0000 0000 0111
```

```
0000 0000 0000 0000 0000 0000 0000 0111
```

© Harry H. Porter, 2006

13-Bit Immediate Values

The instruction includes a 13-bit signed value.

Range: `-4096 .. 4095`

This value is "sign-extended" to 32-bits.

Example:

```
---- ---- ---- ---- ---1 1111 1111 1001
```

```
1111 1111 1111 1111 1111 1111 1111 1001
```

© Harry H. Porter, 2006

Notation

```
sub     reg1,reg2_or_immed,regD
```

reg1, reg2, regD:

Any one of the 32 general-purpose integer registers (5 bits)

immed

13-bit signed integer value

Must be between -4096 and 4095

Signed-extended to 32-bits before being used

Syntax: Full "C"-like expressions

Character literals	`'m'`	*All equal*
	`"m"`	
Hex	`0x6d`	
Decimal	`109`	
Octal	`0155`	
Expressions	`64 + (3 * 'm')`	
Symbols	`x`	*Assembly-time constants, not runtime variables!*

© Harry H. Porter, 2006

Assembler Syntax

One instruction per line
Labels - on same line, or on line alone

Spaces are okay, but not normally used

```
label:
        sub     %g3,%g5,%g7         ! Comments
        add     %g7,56,%g7          !
        add     %12,34,%12          !
```

Tab Tab Tab

Note the Commenting style

Example

```
        ld      myVal,%12           ! myVal = myVal + 78
        add     %12,78,%12          ! .
        st      %12,myVal           ! .
```

The destination is always on the right.

(Not quite legal SPARC)

© Harry H. Porter, 2006

Instructions

Arithmetic

add
sub
Signed: smul, sdiv
Unsigned: umul, udiv

Logical

and
or
xor
andn
orn
xnor

Shifting

sll
srl
sra

These do not modify the condition code Register.

Arithmetic

addcc
subcc
smulcc
sdivcc
umulcc
udivcc

Logical

andcc
orcc
xorcc
andncc
orncc
xnorcc

These do modify the condition code register.

Will set:
Z=1
if result is zero
N=1
if result is neg
etc.

© Harry H. Porter, 2006

Logical Functions

```
and
or
xor    =   x != y
andn   =   x and (not y)
orn    =   x or (not y)
xnor   =   x = y
```

x	y	and	or	xor	andn	orn	xnor
0	0	0	0	0	0	1	1
0	1	0	1	1	0	0	0
1	0	0	1	1	1	1	0
1	1	1	1	0	0	1	1

These instructions work on all 32 bits at once:

```
and     %g4,%g5,%g6

%g4 ->  0011 1100 ... 1010
%g5 ->  1010 1101 ... 1001
        ----------------
%g6 ->  0010 1100 ... 1000
```

© Harry H. Porter, 2006

To turn on bits in a word...

Use the "**or**" instruction and a "mask" word

```
or      x,mask,result
```

Turn on bits in x wherever the mask has a 1 bit

Example: Turn on every other bit in 3A0F

```
0011 1010 0000 1111 <--- 3A0F
0101 0101 0101 0101 <--- mask
-------------------
0111 1111 0101 1111 <--- result
```

To turn off bits in a word...

Use the "**and**" instruction and a mask

```
and     x,mask,result
```

Turn off bits in x wherever the mask has a 0 bit

To flip (or "toggle") bits in a word...

Use the "**xor**" instruction and a mask

```
xor     x,mask,result
```

Change the bits in x wherever the mask has a 1 bit

© Harry H. Porter, 2006

Shifting Instructions

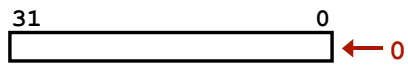
sll

"Shift Left Logical" **<<**

```
sll    reg1,reg2_or_immed,regD
```

Number of bits 0..31

A fast way to multiply by 2^N...

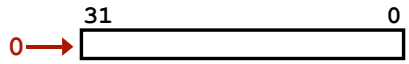


Example: Multiply by 32 (= 2^5)

```
sll    x,5,result
         0000 0000 0000 0011   = 3
         0000 0000 0110 0000   = 64+32 = 96
```

srl

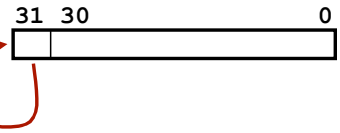
"Shift Right Logical" **>>>**




```
srl    reg1,reg2_or_immed,regD
```

sra

"Shift Right Arithmetic" **>>**




```
sra    reg1,reg2_or_immed,regD
```

A fast way to divide by 2^N, rounding toward **-∞**...

```
sra    x,N,result
```

© Harry H. Porter, 2006

Testing

cmp **reg1,reg2_or_immed**

Compare operand1 to operand2

Set integer condition codes accordingly

The next instruction will normally be a conditional branch

Example:

```
cmp    %g3,73      ! if x <= 73 goto loop
ble    loop        ! .
```

Branch if the condition codes indicate op1 ≤ op2

© Harry H. Porter, 2006

How to turn high-level code into assembly code:

```
if (x >= 73) && (y < 98) then
  b = c + d
endIf
```

Step 1: Convert LOOPs and IFs into GOTOs (possibly reversing the tests):

```
    if x<73 then goto elseLabel
    if y>=98 then goto elseLabel
    b = c + d
elseLabel:
```

Assume:
x in %l3
y in %l2
b in %g3
c in %g4
d in %g5

Step 2: Turn into assembly code.
Keep the operand order and tests the same!

```
    cmp     %l3,73        ! if x >= 73
    bl      elseLabel     ! .
    cmp     %l2,98        ! .  and y < 98
    bge     elseLabel     ! .
    add     %g4,%g5,%g3   !   b = c + d
elseLabel:                ! endIf
```

NOTE: The comments look like source code, including indentation!

© Harry H. Porter, 2006

Unconditional Branch (the "goto" instruction)

```
ba    label
```

Conditional Branches

For Signed Values

```
bl    label
ble   label
bg    label
bge   label
```

For Unsigned Values

```
blu   label
bleu  label
bgu   label
bgeu  label
```

Equality Testing

```
be    label
bne   label
```

<same>
<same>

© Harry H. Porter, 2006

The Condition Code Register

4 bits:

- **N** 1 = negative
- **Z** 1 = zero
- **V** 1 = overflow
- **C** 1 = carry out

Set after arithmetic operations

`addcc`, `subcc`, ...

Reflect the result

Instructions to test the bits individually

Instruction		Will branch if...
`bneg`	`label`	`N=1`
`bpos`	`label`	`N=0`
`bz`	`label`	`Z=1`
`bnz`	`label`	`Z=0`
`bvs`	`label`	`V=1`
`bvc`	`label`	`V=0`
`bcs`	`label`	`C=1`
`bcc`	`label`	`C=0`

© Harry H. Porter, 2006

The Delay Slot

Due to pipelining...

All branch instructions take 1 extra instruction to go into effect

The instruction following the branch is executed before the branch happens!!!

© Harry H. Porter, 2006

The Delay Slot

Due to pipelining...

All branch instructions take 1 extra instruction to go into effect

The instruction following the branch is executed before the branch happens!!!

Option 1: Put a "nop" instruction in the "delay slot"

```
cmp     %l3,73
bl      elseLabel
nop
```

© Harry H. Porter, 2006

The Delay Slot

Due to pipelining...

All branch instructions take 1 extra instruction to go into effect

The instruction following the branch is executed before the branch happens!!!

Option 1: Put a "nop" instruction in the "delay slot"

```
cmp     %l3,73
bl      elseLabel
nop
```

Very tricky to do correctly!

Option 2: Figure out how to put a real, useful instruction in the "delay slot".

```
ld      myVar,%l3       ! var = var - 1
sub     %l3,1,%l3       ! .
st      %l3,myVar       ! .
cmp     %l3,73          ! if var < 73
bl      elseLabel       ! .  goto elseLabel
nop                     ! .
```

© Harry H. Porter, 2006

The Delay Slot

Due to pipelining...

All branch instructions take 1 extra instruction to go into effect

The instruction following the branch is executed before the branch happens!!!

Option 1: Put a "nop" instruction in the "delay slot"

```
cmp     %l3,73
bl      elseLabel
nop
```

Very tricky to do correctly!

Option 2: Figure out how to put a real, useful instruction in the "delay slot".

```
ld      myVar,%l3        ! var = var - 1
sub     %l3,1,%l3        ! .

cmp     %l3,73           ! if var < 73
bl      elseLabel        ! .  goto elseLabel
st      %l3,myVar        ! .  (delay)
```

© Harry H. Porter, 2006

- Figuring out how to rearrange the code to fill the delay slot is difficult & error-prone
- Study Chapter 2 (in "Paul") for examples.
- Project 7:
  You can practice filling the delay slots
  Get the program right first!!!
- Our Compiler
  Will not make this important optimization.
- See how smart the "C" compiler is.

© Harry H. Porter, 2006

Optimizing Assembly Code

A typical loop:

```
while x1 <= 17 do
  x1 = x1 + x2;
  x3 = x3 + 1;
end
```

Assume variables in registers:

```
x1 ⇒ %l1
x2 ⇒ %l2
x3 ⇒ %l3
```

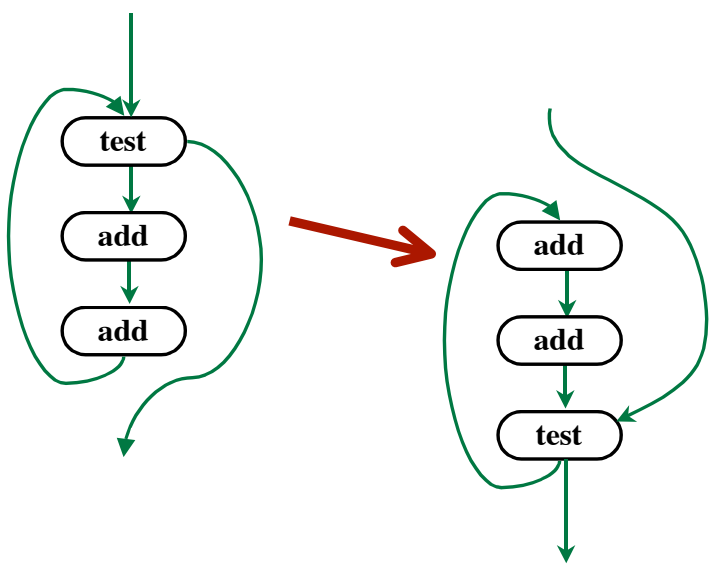
Translation into SPARC:

```
test:
        cmp     %l1,17
        bg      done
        nop
        add     %l1,%l2,%l1
        add     %l3,1,%l3
        ba      test
        nop
done:
```

*Execution Time: N*7*

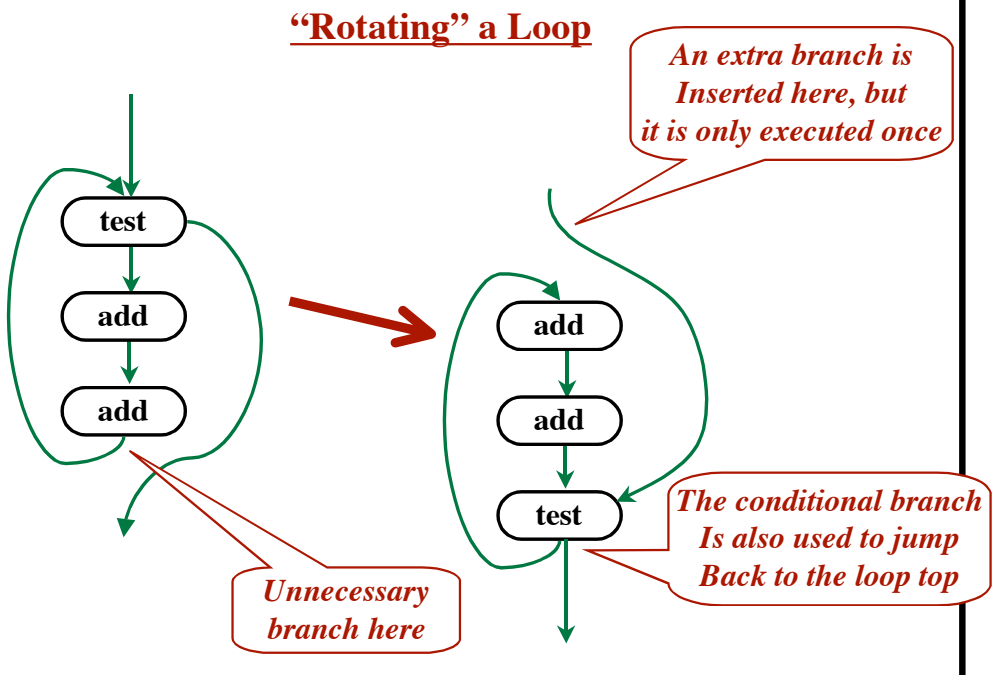
© Harry H. Porter, 2006

"Rotating" a Loop



© Harry H. Porter, 2006

"Rotating" a Loop



© Harry H. Porter, 2006

Optimizing Assembly Code

A typical loop:

```
while x1 <= 17 do
  x1 = x1 + x2;
  x3 = x3 + 1;
end
```

Assume variables in registers:

```
x1 ⇒ %l1
x2 ⇒ %l2
x3 ⇒ %l3
```

Translation into SPARC:

```
test:
        cmp     %l1,17
        bg      done
        nop
        add     %l1,%l2,%l1
        add     %l3,1,%l3
        ba      test
        nop
done:
```

Execution Time:
*N*7*

© Harry H. Porter, 2006

Optimizing Assembly Code

A typical loop:

```
while x1 <= 17 do
  x1 = x1 + x2;
  x3 = x3 + 1;
end
```

Assume variables in registers:

```
x1 ⇒ %l1
x2 ⇒ %l2
x3 ⇒ %l3
```

Translation into SPARC:

```
test:
        cmp     %l1,17
        bg      done
        nop
        add     %l1,%l2,%l1
        add     %l3,1,%l3
        ba      test
        nop
done:
```

```
        ba      test
        nop
loop:
        add     %l1,%l2,%l1
        add     %l3,1,%l3
test:
        cmp     %l1,17
        ble     loop
        nop
```

*Execution Time: N*5*

© Harry H. Porter, 2006

Optimizing Assembly Code

A typical loop:

```
while x1 <= 17 do
  x1 = x1 + x2;
  x3 = x3 + 1;
end
```

Assume variables in registers:

```
x1 ⇒ %l1
x2 ⇒ %l2
x3 ⇒ %l3
```

Translation into SPARC:

```
        ba      test
        cmp     %l1,17
loop:
        add     %l1,%l2,%l1
        add     %l3,1,%l3
        cmp     %l1,17
test:
        ble     loop
        nop
```

```
        ba      test
        nop
loop:
        add     %l1,%l2,%l1
        add     %l3,1,%l3
test:
        cmp     %l1,17
        ble     loop
        nop
```

*Execution Time: N*5 (1 cycle saved, total)*

© Harry H. Porter, 2006

How to fill a delay slot

```
        ble     target
        nop
        mul     ...
        ...
        div
target: add     %l0,%l1,%l2
        sub     ...
```

Copy the target instruction into the delay slot, and branch to 2nd instruction

```
        ble     target
        add     %l0,%l1,%l2
        mul     ...
        ...
        div
        add     %l0,%l1,%l2
target: sub     ...
```

© Harry H. Porter, 2006

How to fill a delay slot

```
        ble     target
        nop
        mul     ...
        ...
        div
target: add     %l0,%l1,%l2
        sub     ...
```

Copy the target instruction into the delay slot, and branch to 2nd instruction

```
        ble     target
        add     %l0,%l1,%l2
        mul     ...
        ...
        div
        add     %l0,%l1,%l2
target: sub     ...
```

Problem:

The "add" is executed even when the branch is NOT taken!

Solution:

"Annulled Branches"

© Harry H. Porter, 2006

Annulled Branches

Assumption:

- Loops end with a conditional branch
- The branch is back to the loop-top
- Loops execute many times
- Goal: Speed up highly repetitive loops
- The branch is taken more often than not
- Goal: Optimize the "branch-is-taken" case

Approach:

- Execute the delay instruction when branch is taken
- Add some support for the case when branch not taken

  May execute a little slower, but...

© Harry H. Porter, 2006

Annulled Branches

One bit in conditional branch instructions

The "annul" bit

If the bit is "0" (The branch is not "annulled")

The instruction in the delay slot is always executed.

Syntax:

```
bge     label
```

If the bit is "1" (The branch is "annulled")

Branch Taken

Instruction in delay slot is executed

Branch Not Taken

Instruction in delay slot is NOT executed

Syntax:

```
bge,a   label
```

© Harry H. Porter, 2006

Optimizing Assembly Code

A typical loop:

```
while x1 <= 17 do
  x1 = x1 + x2;
  x3 = x3 + 1;
end
```

Assume variables in registers:

```
x1 ⇒ %l1
x2 ⇒ %l2
x3 ⇒ %l3
```

Translation into SPARC:

*Execution Time: N*4*

```
        ba      test
        cmp     %l1,17
loop:
        add     %l1,%l2,%l1
        add     %l3,1,%l3
        cmp     %l1,17
test:
        ble     loop
        nop
```

```
        ba      test
        cmp     %l1,17
loop:
        add     %l3,1,%l3
        cmp     %l1,17
test:
        ble,a   loop
        add     %l1,%l2,%l1
```

© Harry H. Porter, 2006

Pseudo-Ops

```
        .byte     35      ! 0x23
        .half     35      ! 0x0023
        .word     35      ! 0x00000023
```

The value can be specified many ways (hex, decimal, ascii, expressions,...)

```
        .word     0x3a0f12d8
        .half     (123+0x0F00)<<5
        .byte     'a'
```

A list of values may be used:

```
        .word     25,78,0x44000000+'a'    ! fills 3 words
```

Floating-Point values may be placed in memory:

```
        .single   0r12.34      ! 4-byte floating point value
        .double   0r+1234e-2   ! 8-byte floating-point value
```

Labels will often be used:

```
 myVar: .word     0xffffabcd
```

© Harry H. Porter, 2006

```
.ascii      "abcdef"
```

Will initialize N bytes of storage, filling it with character data.

"C" strings are terminated with 0x00.

```
.asciz      "abcdef"
```

Will initialize N+1 bytes of storage, putting 0x00 after the final byte.

```
.skip       3500
```

Will skip 3500 bytes, leaving them uninitialized.

© Harry H. Porter, 2006

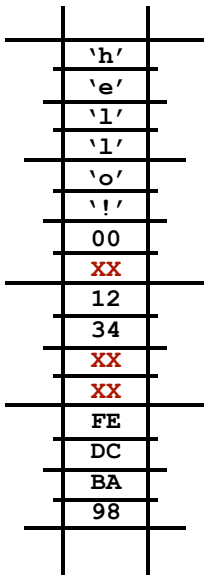
The "align" Pseudo-Op

```
.align      2
.align      4
.align      8
```

Will skip as many bytes as necessary to get onto the indicated alignment boundary.

Example:

```
.asciz      "hello!"
.align      2
.half       0x1234
.align      4
.word       0xfedcba98
```



© Harry H. Porter, 2006

Symbols

Each label is a symbolic name for an address

```
loop:    ...
         ...
         ba      loop
         nop
```

By default, each symbol is local to one ".s" file.

```
         .global     symbol
```

Makes "symbol" available to the linker and debugger as an "external symbol".

```
         .global     main
main:    ...
```

To use an externally defined symbol, nothing special is needed.

```
         call        printf
```

The assembler will not compain if "printf" is not defined in this .s file.
The linker will resolve the symbol.

If not defined in any .o file ⇒ Linker error: "unknown symbol"

© Harry H. Porter, 2006

Segments (in Unix)

`.data` ← Put most data here
Will be placed in read-write pages

`.text` ← Put code and constant data here
Will be read-only pages

`.bss` ← Put uninitialized data here
Read-write pages will be allocated

© Harry H. Porter, 2006

Unix Commands

`gcc myProg.s -c`

Looks at the .s extension
Calls assembler
`-c` means produce a ".o" file

`gcc myProg.o` ***...plus other .o files...*** **`-o myProg`**

Looks at the .o extensions
Calls the linker
"**`-o xxx`**" means produce an executable with name "xxx"

`myProg`

Loads the program into memory and executes it.

`gcc -S samplePgm.c`

To see what the "C" compiler produces.
Creates "samplePgm.s"

© Harry H. Porter, 2006

Integer Multiplication and Division

No multiply or divide instructions in early versions of SPARC

Place operand 1 in %o0
Place operand 2 in %o1
Call a subroutine
Find the result in %o0

Example:

```
ld    x,%o0      ! z = x * y
ld    y,%o1      ! .
call  .mul       ! .
nop              ! .
st    %o0,z      ! .
```

Available subroutines:

Signed and Unsigned Versions

```
.mul   }
.umul  } %o0 x %o1 ⇒ %o0
.div   }
.udiv  }
.rem   } %o0 ÷ %o1 ⇒ %o0
.urem  }
```

© Harry H. Porter, 2006

Loading / Storing / Moving

```
ld      address,regD
st      reg1,address
mov     reg1,regD
```

Data always moves to the left

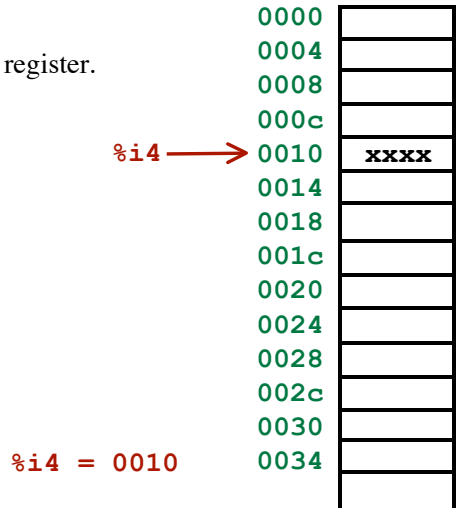
These each move one word (32 bits).
"reg" and "regD" may be any integer register.

Address

```
[reg]
[reg+offset]
[reg+reg]
```

Examples

```
ld      [%i4],%i6
```



© Harry H. Porter, 2006

Loading / Storing / Moving

```
ld      address,regD
st      reg1,address
mov     reg1,regD
```

Data always moves to the left

These each move one word (32 bits).
"reg" and "regD" may be any integer register.

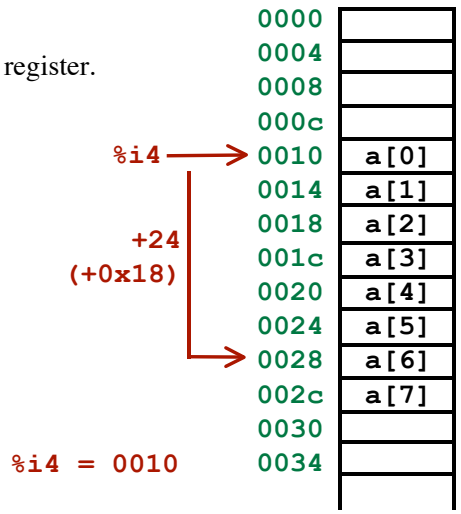
Address

```
[reg]
[reg+offset]
[reg+reg]
```

Examples

```
ld      [%i4],%i6
ld      [%i4+24],%i6
```

A constant 13-bit value



© Harry H. Porter, 2006

Loading / Storing / Moving

```
ld      address,regD
st      reg1,address
mov     reg1,regD
```

Data always moves to the left

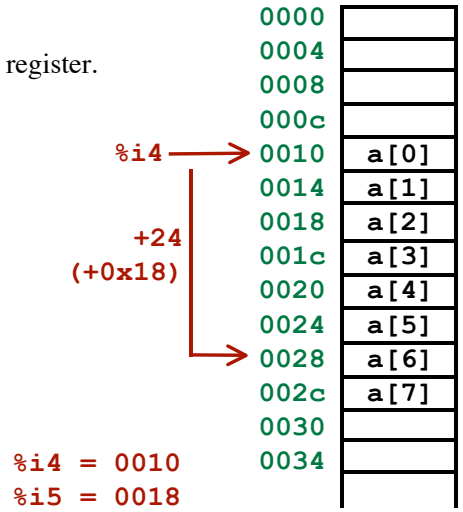
These each move one word (32 bits).
"reg" and "regD" may be any integer register.

Address

```
[reg]
[reg+offset]
[reg+reg]
```

Examples

```
ld      [%i4],%i6
ld      [%i4+24],%i6
ld      [%i4+%i5],%i6
```



© Harry H. Porter, 2006

%g0

Special register: `%g0`

Reading from it?
Always zero

Writing to it?
Data is discarded

© Harry H. Porter, 2006

Synthetic Instructions

```
mov    reg_or_immed,regD
```

Programmer codes this

```
or     %g0,reg_or_immed,regD
```

Assembler produces this

```
not    reg1,regD
xnor   reg1,%g0,regD
```

```
cmp    reg1,reg2_or_immed
subcc  reg1,reg2_or_immed,%g0
```

© Harry H. Porter, 2006

Loading Immediate Data into a Register

Option 1:

```
mov  reg_or_immed,regD
```

The immediate value will be encoded in 13 bits.
... And sign-extended to to 32-bits when used.
The range:
-4096 .. 4095

```
---- ---- ---- ---- ---1 0000 0101 0111
```

```
1111 1111 1111 1111 1111 0000 0101 0111
```

© Harry H. Porter, 2006

The "sethi" Instruction

```
sethi     immed_22,regD
```

A 22-bit value included in instruction
Loaded into the high-order (most significant) 22 bits of regD
The low-order 10 bits are cleared (to zero)

Built-in macros in the assembler

`%hi( X )`
Returns the high-order 22 bits of X
`%lo( X )`
Returns the low-order 10 bits of X

Option 2:

```
sethi     %hi(value),regD
or        regD,%lo(value),regD
```

© Harry H. Porter, 2006

The "set" Synthetic Instruction

Option 3:

```
set       value,regD
```

Any 32-bit value
Expands into two instructions

```
sethi     %hi(value),regD
or        regD,%lo(value),regD
```

Example:

```
set     myVar,%l4
ld      [%l4],%l5
```

© Harry H. Porter, 2006

The "set" Synthetic Instruction

Option 3:

```
set     value,regD
```

Any 32-bit value
Expands into two instructions

```
sethi   %hi(value),regD
or      regD,%lo(value),regD
```

Example:

```
set     myVar,%l4
ld      [%l4],%l5
```

The Delay Slot

```
cmp     ...
ble     loopLabel
add     ...
sub     ...
```

The "delay slot"

Do not put "set" in the delay slot.

Actually, for some values, "set" will expand to only one instruction.
Still, do not put "set" in the delay slot.

© Harry H. Porter, 2006

Synthetic Instructions

Shorthand		*What Gets Assembled*	
tst	reg	orcc	reg,%g0,%g0
clr	regD	or	%g0,%g0,regD
btst	reg_or_immed,reg	andcc	reg,reg_or_immed,%g0
bset	reg_or_immed,regD	or	regD,reg_or_immed,regD
bclr	reg_or_immed,regD	andn	regD,reg_or_immed,regD
btog	reg_or_immed,regD	xor	regD,reg_or_immed,regD
mov	reg_or_immed,regD	or	%g0,reg_or_immed,regD
not	reg1,regD	xnor	reg1,%g0,regD
cmp	reg1,reg2_or_immed	subcc	reg1,reg2_or_immed,%g0
nop		sethi	0,%g0

Mask, with selected bits set to 1

© Harry H. Porter, 2006

Registers

Four groups of 8 registers each

Global	`%g__`
Local	`%l__`
In	`%i__`
Out	`%o__`

Local `%l0, %l1, %l2, ... %l7`

Used by this routine any way it wants
Will be saved automatically during subroutine calls

Global `%g0, %g1, %g2, ... %g7`

`%g0` is special (= zero), can not be modified
Used for "global" data, visible to all routines
Not saved during subroutine calls

In `%i0, %i1, %i2, ... %i7`
Out `%o0, %o1, %o2, ... %o7`

Used in passing parameters to/from subroutines.
The "Calling Conventions"
Efficient parameter passing

© Harry H. Porter, 2006

SPARC Calling Conventions

Consider calling a subroutine
The "*caller*" calls the "*callee*".
From the perspective of the current routine...

"In" Registers

Arguments to this routine are found in

`%i0`	`%i1`	`%i2`	`...`	`%i5`
↑	↑	↑		↑
arg1	arg2	arg3	...	arg6

Fewer than 6 arguments? Use only as many as needed.
Additional arguments? Must be passed on the stack.
This routine will put the "returned value" into `%i0` before returning (if any)

`%i6`

Has a special name: `%fp` (the "frame pointer")

`%i7`

Has a special use
Holds a pointer to the "call" instruction which called this routine
Will be used by the "return" instruction

© Harry H. Porter, 2006

SPARC Calling Conventions

"Out" Registers

Used to pass arguments to routines we will call.

Just before the "call", arguments to the subroutine are put into

```
%o0, %o1, %o2, ... %o5
```

arg1 arg2 arg3 ... arg6

Fewer than 6 arguments? Use only as many as needed.

Additional arguments? Must be passed on the stack.

If the callee returns a value...

This routine will find it in `%o0`

`%o6`

Has a special name: `%sp` (the "stack pointer")

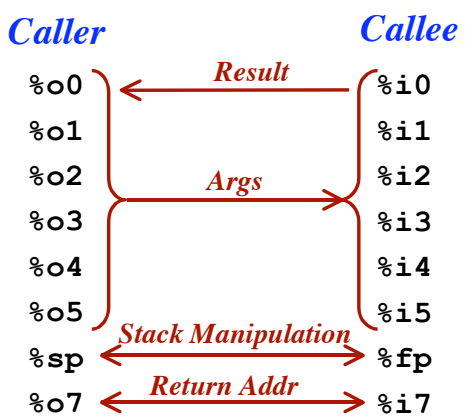
`%o7`

Has a special use

When a "call" instruction is executed...

the address of the "call" instruction will be placed in `%o7`

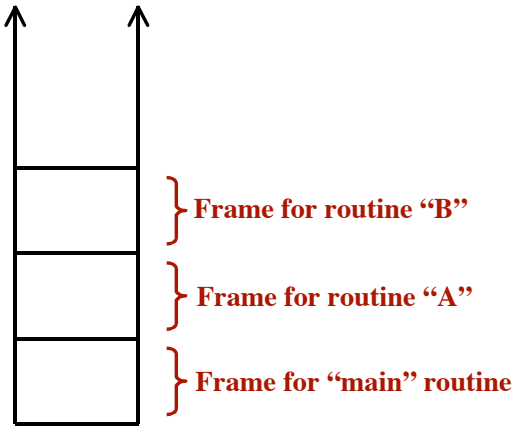
© Harry H. Porter, 2006



© Harry H. Porter, 2006

Subroutine Calling Stack

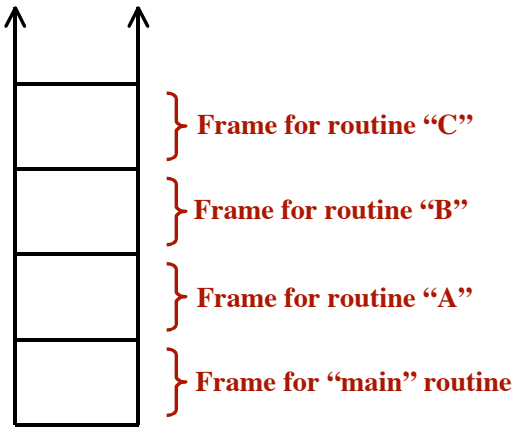
Local variables for each routine...
kept in the stack frame
(Also called "activation record")
The stack of frames is located in main memory



© Harry H. Porter, 2006

Subroutine Calling Stack

Local variables for each routine...
kept in the stack frame
(Also called "activation record")
The stack of frames is located in main memory



© Harry H. Porter, 2006

The SPARC Idea

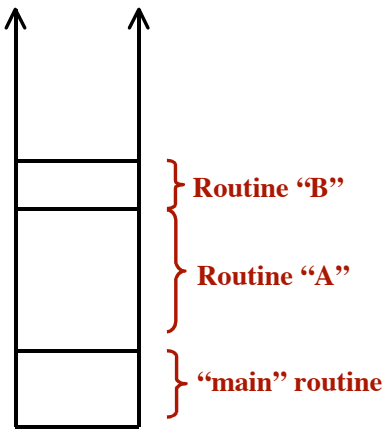
Goal:

- Fast subroutine calling
- Keep the calling stack in registers
  - But each frame has a different size...

Idea:

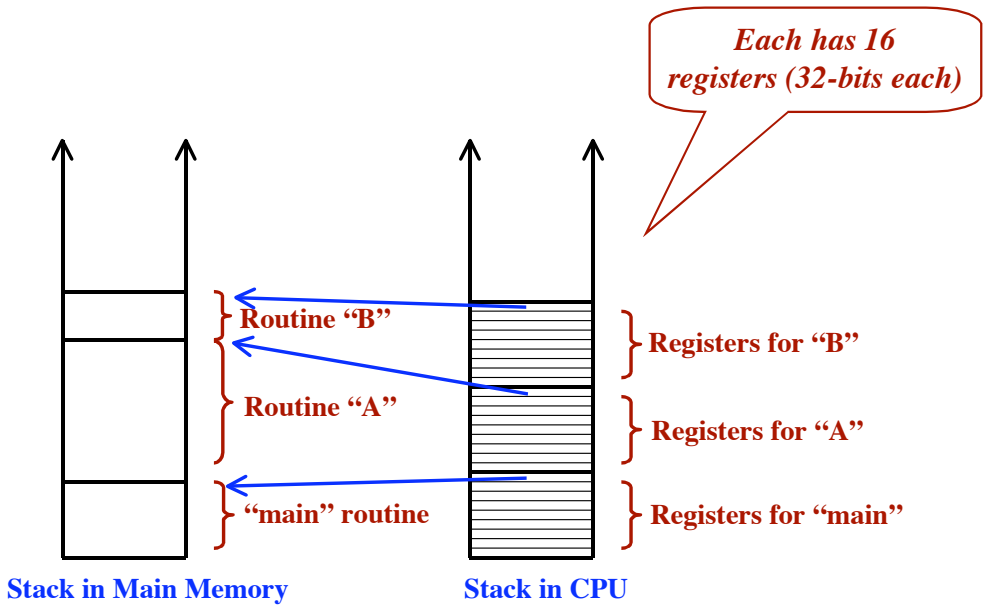
- Cache part of the frame in registers
- Create a stack of frames in the CPU
  - Avoid main memory most of the time!!!

© Harry H. Porter, 2006

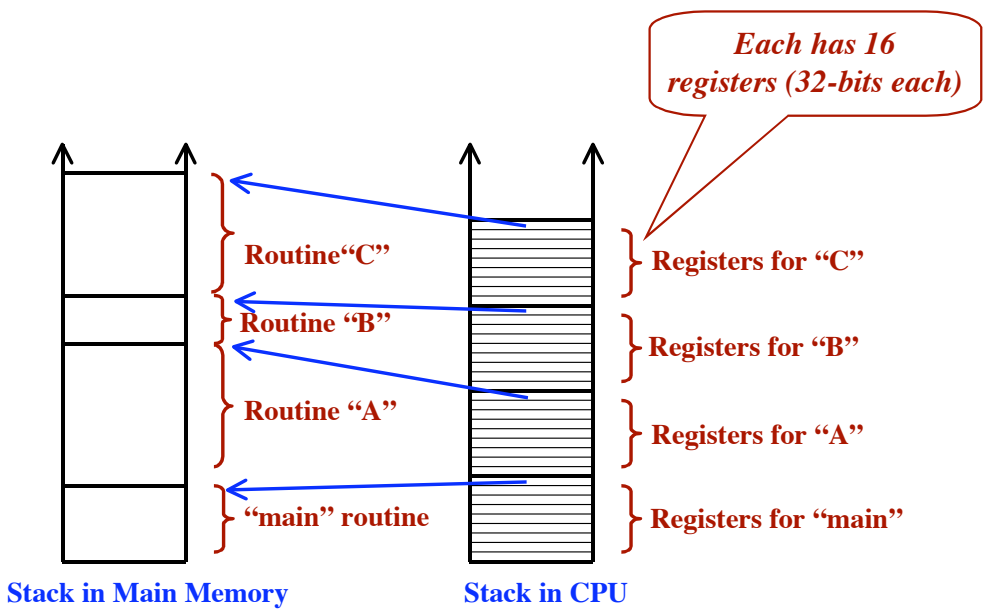


Stack in Main Memory

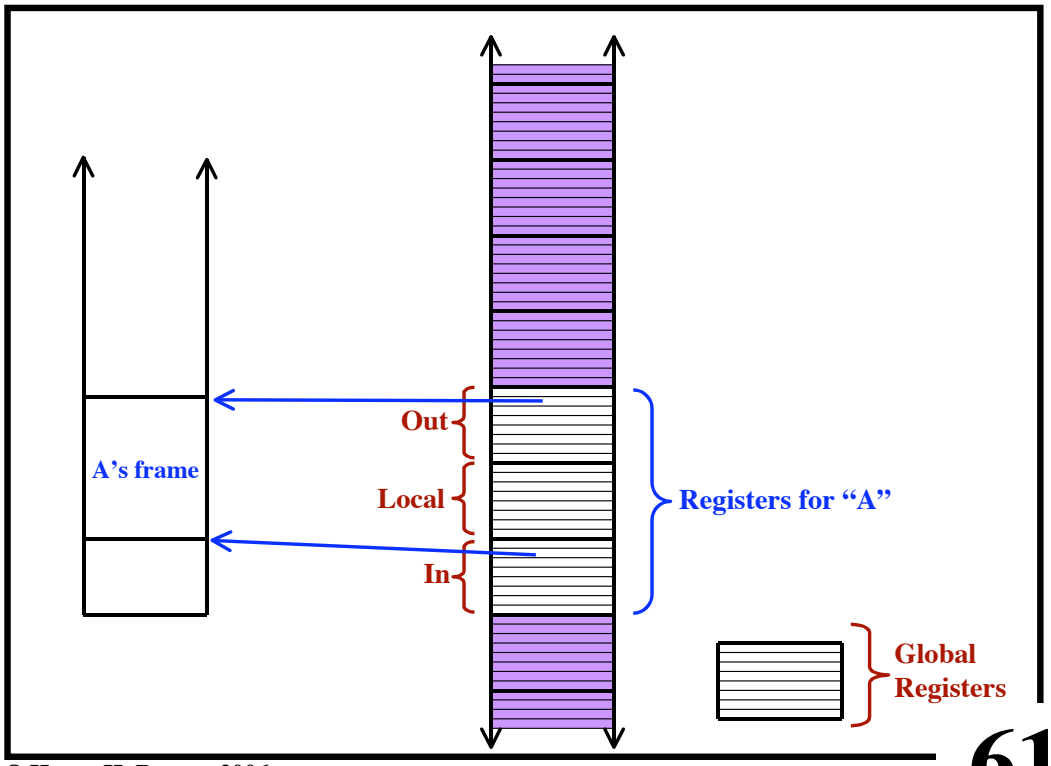
© Harry H. Porter, 2006



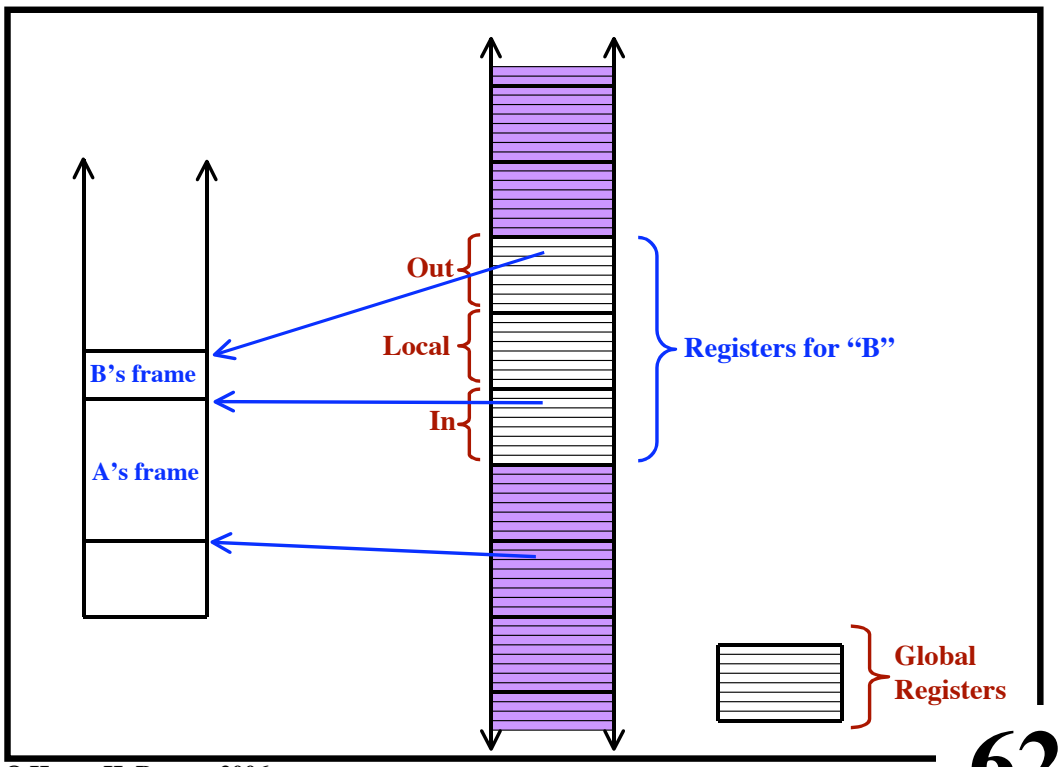
© Harry H. Porter, 2006



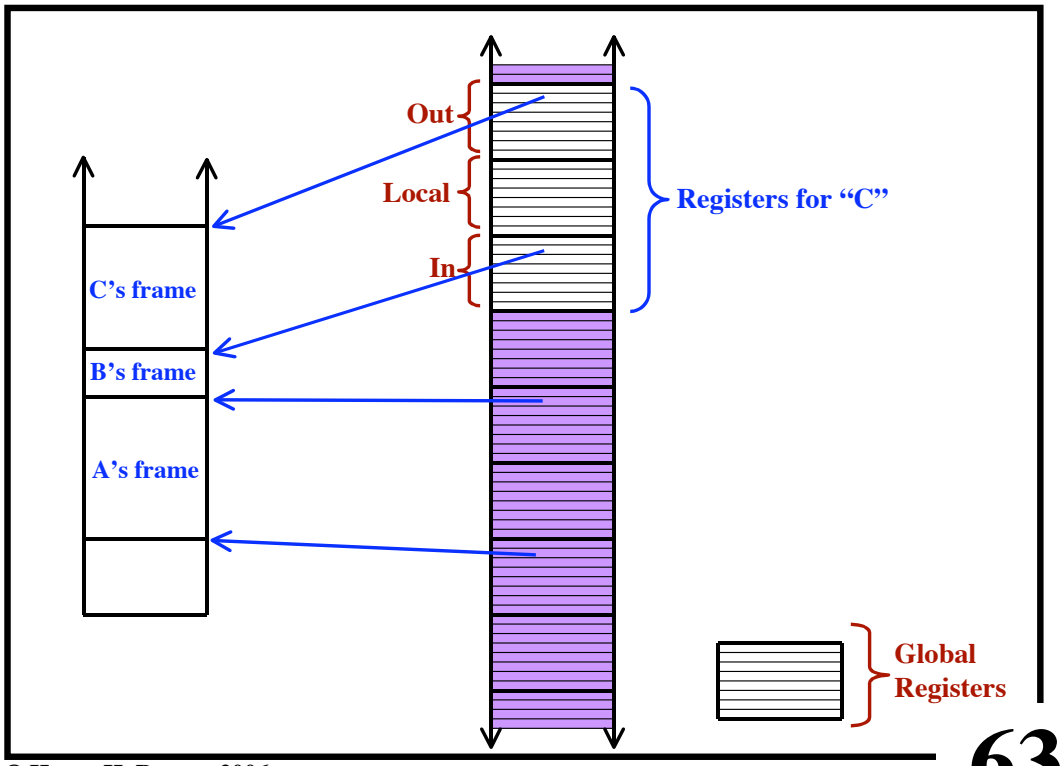
© Harry H. Porter, 2006



© Harry H. Porter, 2006



© Harry H. Porter, 2006



© Harry H. Porter, 2006

- Lots of on-chip registers
- Each routine gets a new set of 16 registers
  Access to stack (i.e., to main memory) is reduced
- Arguments can be passed in registers
  (Most of the time)
- Return addresses are stored in registers (on-chip)
- Relevant instructions

```
call
ret
save
restore
```

These instructions manipulate the register stack

Typical Usage

```
...
  call    foo
...

foo:
  save    ...
  ...
  restore ...
  ret
```

© Harry H. Porter, 2006