Sparc Assembly
Directives & Branching

CS 217

Sparc Assembly Review

C Code

```
x = a + 5000;
```

Assembly code

```
set a,%i1
ld [%i1],%g1

set 5000,%g2

add %g1,%g2,%g1

set x,%i1
st %g1,[%i1]
```

?

Still to Learn

- How do “define variables”
- How to implement control structures
- How to define and call functions
- Other details

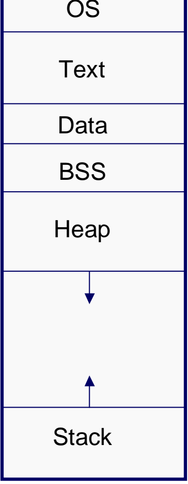
Assembler Directives

Assembler Directives

- Identify sections
- Allocate/initialize memory
- Make symbols externally visible

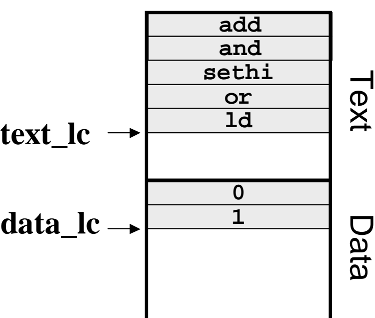
Identifying Sections

- Text (.section ".text")
  - Contains code (instructions)
  - Default section
- Read-Only Data (.section ".rodata")
  - Contains constants
- Read-Write Data (.section ".data")
  - Contains user-initialized global variables
- BSS (.section ".bss")
  - Block starting symbol
  - Contains zero-initialized global variables



Sections (cont)

- Each section has own location counter
  - Location counter is updated when assembler processes directive or instruction



Allocating memory

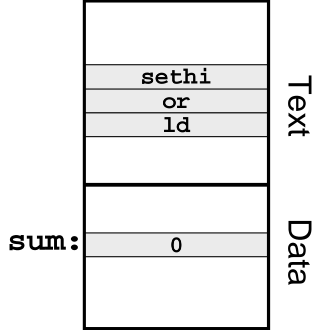
- Increment location counter by nbytes
  - `.skip nbytes`

```
        .section ".bss"
var1:   .skip 16

        .section ".data"
var2:   .skip 4
```

Initializing memory

- Increment location counter and initialize data
  - `.byte    byteval1 [, byteval2 ...]`
  - `.half    halfval1 [, halfval2 ...]`
  - `.word    wordval1 [, wordval2 ...]`

```
        .section ".data"
sum:    .word 0

        .section ".text"
        set sum, %o0
        ld [%o0], %i1
```



Initializing ASCII Data

- Special directives for ascii data

```
.byte  150, 145, 154, 154, 157, 0

.ascii "hello"
.byte 0

.asciz "hello"
```

Making Symbols Externally Visible

- Mark variables as global
  - `.global`

```
        .section ".data"
        .align 4
        .global month
month:  .word jan, feb, mar, apr, may, jun
        .word jul, aug, sep, oct, nov, dec
  jan:  .asciz "January"
  feb:  .asciz "February"
  mar:  .asciz "March"
  apr:  .asciz "April"
  may:  .asciz "May"
  jun:  .asciz "June"
  jul:  .asciz "July"
  ...
```

Making Symbols Externally Visible

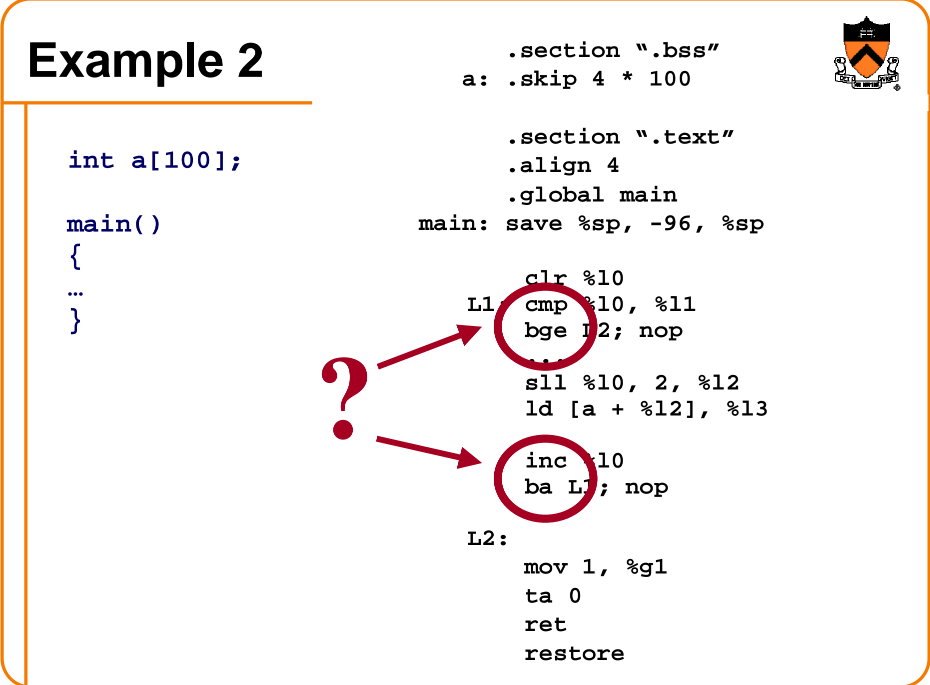
- Mark functions as global
  - **`.global`**

```
         .section ".rodata"
fmt:     .asciz "Hello, world\n"

         .section ".text"
         .align 4
         .global main
main:    save %sp, -96, %sp

         set fmt, %o0
         call printf
         nop

         mov 1, %g1
         ta 0
         ret
         restore
```

Example 1

```
struct example {
  int a, b;
  char d;
  short x, y;
  int u, v;
};

struct example a =
{
  1, 2,
  'C',
  4, 5,
  6, 7
};

main()
{
…
}
```

```
         .section ".data"
a:       .word 1, 2
         .byte `C'
         .align 2
         .half 3, 4
         .align 4
         .word 6, 7

         .section ".text"
         .align 4
         .global main
main:    save %sp, -96, %sp

         set a, %l0
         ld [%l0 + 0], %l1
         ld [%l0 + 4], %l2
         ldub [%l0 + 8], %l3
         ldsh [%l0 + 10], %l4

         mov 1, %g1
         ta 0
         ret
         restore
```



Branches

- Instructions normally fetched and executed from sequential memory locations
  - PC is the address of the current instruction
  - nPC is the address of the next instruction
  - nPC = PC + 4
- Branches and control transfer instructions change nPC to something else
  - ba *label* nPC = label
  - bge *label* if (last compare was "greater or equal")
    nPC = label
    else
    nPC = PC + 4

Branch Instructions

- Set program counter (PC) conditionally

b [a / n / .. / z] {,a} *label*

00	a	cond	010	disp22
31	29	28	24	21

- If branch is taken …
  - nPC = PC + 4 * signextend(disp22)
  - target is a PC-relative address
  - where PC is the address of the branch instruction
- Decision to branch is based on integer condition codes

Integer Condition Codes

- Processor State Register (PSR)

. . .	icc	. . .
31	23	20

- Integer condition codes (icc)
  - N **set if the last ALU result was negative**
  - Z **set if the last ALU result was zero**
  - V **set if the last ALU result was overflowed**
  - C **set if the last ALU instruction that modified the** icc **caused a carry out of, or a borrow into, bit 31**

Carry and Overflow Condition Codes

- If the carry bit is set
  - the last addition resulted in a carry, or
  - the last subtraction resulted in a borrow

  - Used for multi-word addition

```
addcc  %g3,%g5,%g7
addxcc %g2,%g4,%g6
```

the most significant word is in the even register

(%g6,%g7) = (%g2,%g3) + (%g4,%g5)

- If the overflow bit is set
  - result of subtraction (or signed-addition) doesn't fit

CC Instructions

- Arithmetic operations
  - addcc ***src1, src2, dst*** dst = src1 + src2
  - subcc ***src1, src2, dst*** dst = src1 - src2
- Logical operations
  - andcc ***src1, src2, dst*** *dst = src1 & src2*
  - orcc ***src1, src2, dst*** *dst = src1 | src2*
  - xorcc ***src1, src2, dst*** *dst = src1 ^ src2*
- Synthetic instructions
  - tst *reg* orcc reg,%g0,%g0
  - btst *bits,reg* andcc reg,bits,%g0
  - cmp *src1,src2* subcc src1,src2,%g0
  - cmp *src,value* subcc src,value,%g0

Branch Instructions

- Unconditional branches (and synonyms)
  - ba jmp **branch always**
  - bn nop **branch never**
- Raw condition-code branches
  - bnz !Z
  - bz Z
  - bpos !N
  - bneg N
  - bcc !C
  - bcs C
  - bvc !V
  - bvs V

Branching Instructions (cont)

- Comparison branches

instruction	signed	unsigned
be	Z	Z
bne	!Z	!Z
bg bgu	!(Z | (N^V))	!(C | Z))
ble bleu	Z | (N^V)	C | Z
bge bgeu	!(N^V)	!C
bl blu	N^V	C

Branching Examples

- if-then-else

```
if (a > b)
    c = a;
else
  c = b;
```

```
     #define a %l0
     #define b %l1
     #define c %l2
     cmp a,b
     ble L1; nop
     mov a,c
     ba L2; nop
L1:  mov b,c
L2:  ...
```

Branching Examples (cont)

- Loops

```
for (i=0; i<n; i++)
    . . .
```

```
     #define i %l0
     #define n %l1
     clr i
L1:  cmp i,n
     bge L2; nop
     . . .
     inc i
     ba L1; nop
L2:
```

Branching Examples (cont)

- Loops (alternative implementation)

for (i=0; i<n; i++)
    . . .

```
     #define i %l0
     #define n %l1
     clr i
     ba L2; nop
L1:  . . .
     inc i
L2:  cmp i,n
     bl L1; nop
```

Example 2 (again)

```
int a[100];

main()
{
…
}
```

```
        .section ".bss"
a:      .skip 4 * 100

        .section ".text"
        .align 4
        .global main
main:   save %sp, -96, %sp

        clr %l0
L1:     cmp %l0, %l1
        bge L2; nop
        ...
        sll %l0, 2, %l2
        ld [a + %l2], %l3
        ...
        inc %l0
        ba L1; nop

L2:
        mov 1, %g1
        ta 0
        ret
        restore
```

More Control Transfer Instructions

- Control transfer instructions

instruction	type	addressing mode
b*icc*	conditional branch	PC-relative
jmpl	jump and link	register indirect
rett	return from trap	register indirect
call	procedure call	PC-relative
t*icc*	traps	register indirect (vectored)

PC-relative addressing is like register displacement addressing that uses the PC as the base register

More Control Transfer Instructions

- Branch instructions

op	a	cond	op2	disp22

nPC = PC + signextend(disp22) << 2

- Calls

op	disp30

nPC = PC + signextend(disp30) << 2

position-independent code does not depend on where it's loaded; uses PC-relative addressing

Summary

- Assembly language directives
  - Define sections
  - Allocate memory
  - Provide labels for variables, branches, etc.
  - Control whether labels are externally visible
- Branch instructions
  - Set nPC conditionally based on integer condition codes
  - ICCs set by arithmetic and logical instructions
  - Branch addresses are relative (at most 22 bits away)
- Other control transfer instructions
  - Described in next few lectures