Sparc Assembly
Directives & Branching

CS 217

Sparc Assembly Review

<table>
<thead>
<tr>
<th>C Code</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = a + 5000;</td>
<td>set a,%i1</td>
</tr>
<tr>
<td></td>
<td>ld [%i1],%g1</td>
</tr>
<tr>
<td></td>
<td>set 5000,%g2</td>
</tr>
<tr>
<td></td>
<td>add %g1,%g2,%g1</td>
</tr>
<tr>
<td></td>
<td>set x,%i1</td>
</tr>
<tr>
<td></td>
<td>st %g1,[%i1]</td>
</tr>
</tbody>
</table>
Still to Learn

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

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, %0
  ld [%0], %1
  ```

  ```
  Text
  Data
  sum:
  ```
Initializing ASCII Data

- Special directives for ascii data

```
.byte 150, 145, 154, 154, 157, 0
.ascii "hello"
.byte 0
.ascii "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()
{
  ...
}
```
Example 2

```c
int a[100];

main()
{
    ...
}
```

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

\[
\begin{array}{c}
\text{branch} \\
\text{label}
\end{array}
\]

\[
\begin{array}{c|c|c}
00 & a & \text{cond} \\
31 & 29 & 28 \\
010 & & \text{disp22} \\
& 24 & 21
\end{array}
\]

• If branch is taken …
  - \( nPC = PC + 4 \times \text{signextend}(\text{disp22}) \)
  - 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)

\[
\begin{array}{c|c|c|c}
\ldots & \text{icc} & \ldots \\
31 & 29 & 25 & 23
\end{array}
\]

• 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: \( %g2, %g4, %g6 \)

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

- 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: \( src1, src2, dst \) \( dst = src1 \mid src2 \)
  - 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

<table>
<thead>
<tr>
<th>instruction</th>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>be</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>bne</td>
<td>!Z</td>
<td>!Z</td>
</tr>
<tr>
<td>bg bgu</td>
<td>!(Z</td>
<td>(N^V))</td>
</tr>
<tr>
<td>ble bleu</td>
<td>Z</td>
<td>(N^V)</td>
</tr>
<tr>
<td>bge bgeu</td>
<td>!(N^V)</td>
<td>!C</td>
</tr>
<tr>
<td>bl blu</td>
<td>N^V</td>
<td>C</td>
</tr>
</tbody>
</table>
Branching Examples

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

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

Branching Examples (cont)

- Loops
  
  ```
  for (i=0; i<n; i++)
      . . .
  ```
Branching Examples (cont)

• Loops (alternative implementation)
  for (i=0; i<n; i++)
    ...

```c
#define i %10
#define n %11
clr i
ba L2; nop
L1: ...
  inc i
L2: cmp i, n
  bl L1; nop
```

Example 2 (again)

```c
.section `.bss'
a: .skip 4 * 100

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

  clr %10
  L1: cmp %10, %11
  bge L2; nop
  ...
  sll %10, 2, %12
  ld [a + %12], %13
  ...
  inc %10
  ba L1; nop

L2:
  mov 1, %g1
  ta 0
  ret
  restore
```
More Control Transfer Instructions

- Control transfer instructions
  - instruction type addressing mode
    - `bicc` conditional branch PC-relative
    - `jmpl` jump and link register indirect
    - `rett` return from trap register indirect
    - `call` procedure call PC-relative
    - `ticc` traps register indirect (vectored)

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

- Branch instructions
  - \[ \text{op} \ a \ \text{cond} \ \text{op2} \ \text{disp22} \]
  - \( \text{nPC} = \text{PC} + \text{signextend}(\text{disp22}) \ll 2 \)

- Calls
  - \[ \text{op} \ \text{disp30} \]
  - \( \text{nPC} = \text{PC} + \text{signextend}(\text{disp30}) \ll 2 \)

  *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