Overview of Today's Lecture:

- The MIPS Assembly Language.
- MIPS Assembly Language Programming Conventions.
- The program Stack.
- Useful C techniques: "case" selection, "hash lookup"

Reading Assignment: Chapter 3, Appendix A
SPIM manual.

Integer to Hex in C

```c
char s[9];
char tr[]="0123456789ABCDEF";
void itohex(int I) {
    /* Convert I to a sequence of 8 hexadecimal digits in s */
    int j,k;
    s[8] = '\0';
    for (k=7; k>=0 k--) {
        j = I & 0xF; /* Save low-order 4 bits of I */
        s[k]=tr[j];
        I = I >> 4;
    }
}
```
Integer to Hex in SPIM

```assembly
.data
s: .space 8
.asciiz "\" # set s[8]='\0'
tr: .asciiz "0123456789ABCDEF"
.text
# itohex converts integer in $a0 to hex, result to $s[0..7]
itohex: la $t2, s # t2 = &s, to stop the loop
add $t0, $t2, 7 # t0 = &s[7] (k=7)
L1: andi $t1, $a0, 0xF # j = I & 0xF
    lb $t1, tr($t1) # j = tr[j]
    sb $t1, 0($t0) # s[k] = j
    srl $a0, $a0, 4 # I = I >> 4
    addi $t0, $t0, -1 # k--
    bge $t0, $t2, L1 # --> L1 if k>=0
    jr $ra # return to caller
```

MIPS: Software conventions for Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero constant 0</td>
</tr>
<tr>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>2</td>
<td>v0 expression evaluation</td>
</tr>
<tr>
<td>3</td>
<td>v1 function results</td>
</tr>
<tr>
<td>4</td>
<td>a0 arguments</td>
</tr>
<tr>
<td>5</td>
<td>a1</td>
</tr>
<tr>
<td>6</td>
<td>a2</td>
</tr>
<tr>
<td>7</td>
<td>a3</td>
</tr>
<tr>
<td>8</td>
<td>t0 temporary: caller saves</td>
</tr>
<tr>
<td>15-17</td>
<td>temporary: caller saves</td>
</tr>
<tr>
<td>16-17</td>
<td>callee saves</td>
</tr>
<tr>
<td>18-31</td>
<td>temporary (cont’d)</td>
</tr>
</tbody>
</table>

Memory Layout

- **Text segment**
- **Data segment**
- **Stack segment**
- **Reserved**
- **Static data**
- **Dynamic data**
- **Global area**
- **Frame pointer**
- **Return Address (HW)**
Example2

# Program to add together list of 9 numbers.

.text  # Code
.align 2
.globl main
main:  # MAIN procedure Entrance
    subu $sp, 40         # Push the stack
    sw $ra, 36($sp)    # Save return address
    sw $s3, 32($sp)    # Entry Housekeeping
    sw $s2, 28($sp)    # save registers on stack
    sw $s1, 24($sp)    #
    sw $s0, 20($sp)    #
    move $v0, $0         # initialize exit code to 0
    move $s1, $0         #
    la $s0, list       # Initialization
    la $s2, msg        #
    la $s3, list+36    #
    move $s1, $0         #
    la      $s0, list       # 
    la      $s2, msg        # / 
    la      $s3, list+36    #/
    move  $s1, $0  
    lw      $t6, 0($s0)      #
    addu    $s1, $s1, $t6    # Actual "work"  
    # SPIM I/O
    li      $v0, 4          #
    move    $a0, $s2        #  Print a string
    syscall                 #/
    li      $v0, 1          #
    move    $a0, $s1        #  > Print a number
    syscall                 #/
    li      $v0, 4          #
    la      $a0, nln        # > Print a string (eol)
    syscall                 #/
    addu    $s0, $s0, 4       # index update and
    bne     $s0, $s3, again   # end of loop
    move     $v0, $0         #
    lw      $s0, 20($sp)    # 
    lw      $s1, 24($sp)    # 
    lw      $s2, 28($sp)    # 
    lw      $s3, 32($sp)    # / load return address
    addu    $sp, 40         # Pup the stack
    jr      $ra             # exit(0) ;
.end    main            #  end of program

.data                   # Start of data segment
list:   .word   35, 16, 42, 19, 55, 91, 24, 61, 53
msg:    .asciiz "The sum is "
nln:    .asciiz "\n"
System call

- System call is used to communicate with the system and do simple I/O.
- Load system call code into Register $v0
- Load arguments (if any) into registers $a0, $a1 or $f12 (for floating point).
- Do: syscall
- Results returned in registers $v0 or $f0.

<table>
<thead>
<tr>
<th>code</th>
<th>service</th>
<th>Arguments</th>
<th>Result</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>print int</td>
<td>$a0</td>
<td>integer in $v0</td>
<td>(address)</td>
</tr>
<tr>
<td>2</td>
<td>print float</td>
<td>$f12</td>
<td>float in $f0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>print double</td>
<td>$f12</td>
<td>double in $f0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>read integer</td>
<td>$a0</td>
<td>integer in $v0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>read float</td>
<td>$a0</td>
<td>float in $f0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>read double</td>
<td>$a0</td>
<td>double in $f0</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>read string</td>
<td>$a0, $a1</td>
<td>integer in $v0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>sbrk</td>
<td>$a0</td>
<td>address in $v0</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>exit</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Details of the MIPS instruction set

- Register zero always has the value zero (even if you try to write it)
- Jump and link instructions put the return address PC+4 into the link register ra. (register $31)
- All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, ...)
- Immediate arithmetic and logical instructions extend operand 2 as follows:
  - Logical immediate values are zero extended to 32 bits
  - Arithmetic immediate values are sign extended to 32 bits
- The data loaded by the instructions lb and lh are extended as follows:
  - lbu, lhu are zero extended
  - lb, lh are sign extended
- Overflow can occur in these arithmetic and logical instructions:
  - add, sub, addi, div, divu?
  - It can not occur in addu, subu, addiu, and, or, xor, nor, shifts, multu, multi

Miscellaneous MIPS I instructions

- break A breakpoint trap occurs, transfers control to exception handler
- syscall A system trap occurs, transfers control to exception handler
- coprocessor instrs. Support for floating point
- TLB instructions Support for virtual memory: discussed later
- restore from exception Restores previous interrupt mask & kernel/user mode bits into status register
- load word left/right Supports misaligned word loads
- store word left/right Supports misaligned word stores
- Instructions not accessible through C:
  - ror, rol Rotate right and left
  - bccal Branch LT, or GE, and link
C programming: Many-way action selection

- switch(n)
  - Acts like a "computed" goto, which selects among several labels
    ```
    switch(n) {
      case 1: Action 1;
      break;
      ...
      case k: Action k;
    }
    ```
- "Indexed subroutine call"
  - Selects one of several different subroutines to call
    ```
    void A1(); A2(); ...; Ak();
    void *(choose[]) () = {A1, A2, ... , Ak};
    *(choose[n]) ();
    ```

Many-way selection in SPIM

- Many-way selection is implemented FAST by using a table of "target locations"
  ```
  all   $a0,$a0,2 # scale k by *4 to index words
  lw    $t1, tbl($a0) # t1 = tbl[k]
  jr    $t1 # goto *t1
  ```
- Indexed subroutine selection is done in about the same way
  ```
  The "jr" instruction is replaced by "jalr $t1"
  Each action returns, using "jr $r31"
  The "done" processing immediately follows the "jalr"
  ```

Hash tables in C

- A "hash lookup table" uses a particularly fast algorithm to compute a table index associated uniquely with a given object. "Object" could be a character string, or almost anything else.
- Method: Compute some arithmetic function h() which depends on the bits of the object O's internal representation. Use h() as the starting point for a circular linear search of the table, looking for a match with O, or an empty slot. Return the index of whichever you find.
- The table entries usually contain pointers to objects, so the table entry size need not be as large as the largest object stored.
- The table is not allowed to get more than about 80% full, so if O is not found, the search is guaranteed to stop at a null entry.
- Other forms of hash search exist, including one which follows a chain of pointers, headed by TBL[h()]. The pointer-following version uses more space than the circular linear search version. A search with better performance than the linear one can be built, by computing a "second hash function h2(O), and stepping h by h2 each time."
Hash table: C version

```c
unsigned int hash(char * str) {
    unsigned int k=0;
    while (*str) k = (k<<3) + *str;
    return k;
}
```

```c
#define SIZE (some prime number)
int strst, strfree;  char *tbl[SIZE]; str[10000];
int look() { /* Looks up str[0] in the hash table */
    char *p=&str[0];
    int h=hash(p), h2;
    h2 = (h%(SIZE-3))+1;  h=h%SIZE;
    while( tbl[h] && !strcmp(p,tbl[h] ) h = (h+=h2<SIZE) ? H : h-SIZE;
    if ( !tbl[h] ) { /* should add check for tbl 80% full here */
        tbl[h]=p; strst=strfree
    } else strfree =strst;
    return h;
}
```

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:

- **CALL** B
- **CALL** C
- **RET**
- **RET**

Some machines provide a memory stack as part of the architecture (e.g., VAX)
Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments (subroutine local variables & return address)
even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:

<table>
<thead>
<tr>
<th>SP Full?</th>
<th>Next Empty?</th>
<th>Little Big</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1 grows up</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0 grows down</td>
</tr>
</tbody>
</table>

How is empty stack represented?

- Little -> Big/Last Full
- Little/Next Empty -> Big

- **POP:** Read from Mem(SP)  Decrement SP
- **PUSH:** Increment SP  Write to Mem(SP)
Call-Return Linkage: Stack Frames (general case)

- Reference args at fixed (positive) offsets from FP
- Reference old registers and local variables at fixed (negative) offsets from FP
- Grows if arrays allocated in the stack

- Key fact: Each subroutine operates on its own Stack Frame, does not disturb other Stack Frames
- Many variations on stacks possible (up/down, last pushed / next)

- Compilers try to keep "register" variables in registers, not memory!

Call-Return Linkage: Stack Frames (MIPS)

- Arguments of this subroutine and local variables are addressed at fixed (≥0) offsets from SP
- Arguments of called subroutines addressed with fixed < 0 offsets from SP

- MIPS stack starts at 0x7fffffff and it grows down.
- MIPS stack frame doesn't grow, so FP is not needed, or computed.
- Calling routine computes called program's arguments in regs + memory

MIPS Function Calling Conventions

- First four arguments passed in registers.
Subroutines in perspective

- Non-recursive subroutines can store their local variables in FIXED memory locations.
  - The assembler lets you give names to locations in .data segment.
- Recursive subroutines must use new storage for each call.
  - Stack is convenient for local variables.
    - Storing variables on the stack is faster than in the .data segment.
      - To reference variable XYZ in .data, must execute 3 instructions:
        - lui $at, hi(XYZ)
        - ori $at, $at, lo(XYZ)
        - lw $t0, 0($at)
      - On the stack, 1 operation enough:
        - lw $t0, 4($sp)
- Using the stack is the "general" case; a compiler uses this method to avoid complicated analysis needed to decide if subroutine "non-recursive".
- "Conventions" suggest using the stack when calling compiled subroutines.

Radix Sort in C

```c
#include <stdio.h>

/* MSB radix sort -- a useful recursive routine */
/* msk must be 2**n, for 31>=n>=0. sort the integers in the range l..h inclusive on their low-order n bits */

/* Method: Pick an integer x from the range. If x&msk==0, move x to the lower end of the range; otherwise to the high end. Use l and h to keep track of those integers which have already been moved. Stop when l>v. Then recursively sort the low and the high parts of the range, separately. */

void rsrt(int *l, int *h, unsigned int msk) {
  int x, t, *l0, *h0;
  if (msk==0) return;
  if (h<l) return;
  l0 = l; h0 = h;
  x = *l;
  while (l<=h) {
    if (x&msk) {
      /* Move x to high part */
      t = *h;
      *h-- = x;
      x = t;
    } else {
      /* Move to low part */
      *l++ = x;
      x = *l;
    }
  }
  rsrt(l0, l-1, msk>>1);
  rsrt(h+1, h0, msk>>1);
}
```

Radix Sort in C (cont'd)

```c
void rsrt(int *l, int *h, unsigned int msk) {
  int x, t, *l0, *h0;
  if (msk==0) return;
  if (h<l) return;
  l0 = l; h0 = h;
  x = *l;
  while (l<=h) {
    if (x&msk) {
      /* Move x to high part */
      t = *h;
      *h-- = x;
      x = t;
    } else {
      /* Move to low part */
      *l++ = x;
      x = *l;
    }
  }
  rsrt(l0, l-1, msk>>1);
  rsrt(h+1, h0, msk>>1);
}
```
Radix Sort in C (main)

```c
int A[1000];
int main() {  
    int i, j;
    i = 0;
    while (i<1000) {  
        if ((j=scanf("%d", &A[i]))==EOF) break;
        i++;
    }
    rsrt(&A[0], &A[i-1], 1<<31);
    for (j=0; j<i; j++)
        printf( "%d\n", A[j]);
    return 0;
}
```

Radix Sort in SPIM

```c
/* MSB radix sort -- a useful recursive routine */
/* Input: 1 number per line, terminated by a negative number */
text
.globl rsrt
rsrt: subu $sp,$sp,20
    sw $ra,4($sp)
    sw $a0,8($sp)
    sw $a1,12($sp)
    sw $a2,16($sp) #void rsrt(int *l, int *h, unsigned
beqz $a2,retn # if (msk==0) return;
blt $a1,$a0,retn # if (h<l) return;
mov $t0,$a0
    mov $t1,$a1 # l0 = l; h0 = h;
    lw $t2,0($a0) # x = *l;
    b whl1 #}
    and $t3,$t2,$a2
    beqz $t3,low # if (x&msk) {
        lw $t3,0($a1) # t = *h;
        sw $t2,0($a1) # *h-- = x;
        move $t2,$t3 # x = t;
        b join # }
low: # else { # /* Move to low part */
    sw $t2,0($a0)
    lw $t2,0($a0) # x = *l;
    join: # }
    b join

whl1: and $t3,$t2,$a2
    beq $t3,low # if (x&msk) {  
    lw $t2,0($a1) # ...
    sw $t2,0($a0)
    lw $t2,0($a0) # ...  
    b join
    # and while (l<h) {
```

Radix Sort in SPIM (cont'd)
Radix Sort in SPIM (cont'd)

# Some thought needed: how do I preserve l and h over the recursive calls?
# Solution: h==l-1, so only one needs preservation: I'll put it in the AR
sw $a0,20($sp) # Save l (same as h+1 now)
lw $a0,8($sp) # original argument, or "l0"
srl $a2,$a2,1 # msk = msk >> 1
jal rsrt # rsrt(l0,l-1,msk>>1): (a1 is correct)
lw $a0,20($sp) # h1
lw $a1,12($sp) # h0
lw $a2,16($sp) # original msk
srl $a2,$a2,1 # msk = msk >> 1
jal rsrt # rsrt(h1,h0,msk>>1);
ret: # }
lw $ra,4($sp) # restore return address
addu $sp,$sp,20 # restore stack pointer
jr $ra # return to caller

.data
.align 2
A: .space 4000 #int A[1000];
nln: .asciiz "\n"

CPS104 MPS.

Radix Sort in SPIM (cont'd)

.text
.align 2
global main
main: subu $sp,$sp,4
sw $a0,4($sp) #int main() {
    # Keep &A[i] in $a0, &A[1000] in $t0 int i, j;
    la $a0,A # $a0 = $A[i=0]
    addu $t0,$a0,4000 # $t0 = &A[1000] for end test
    mwhl1: li $v0,5 # code for read_int
        syscall #
        bltz $v0,ml2 # break if # < 0 (Can't see EOF)
        sw $v0,0($a0) # A[i] = #
        addu $a0,$a0,4 # i++
        blt $a0,$t0,mwhl1 # end while (i<1000)
ml2: sw $a0,8($sp) # save &A[i] for printing
    subu $a1,$a0,4 # $a1 = &A[i-1]
    la $a0,A # $a0 = &A[0]
    lui $a2,0x8000 # $a2 = 1<<31
    jal rsrt # rsrt(&A[0],&A[i-1],1<<31);
    move $v0,$0
    lw $ra,4($sp)
    addu $sp,$sp,8 # for printing
    jr $ra # return 0; #