

CPS 104
Computer Organization
Lecture 6: MIPS ISA and Assembler

Robert Wagner

CPS104 MPS.1

©RW Fall 2000

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.

CPS104 MPS.2

©RW Fall 2000

Integer to Hex in 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;
  }
}
```

CPS104 MPS.3

©RW Fall 2000

Integer to Hex in SPIM

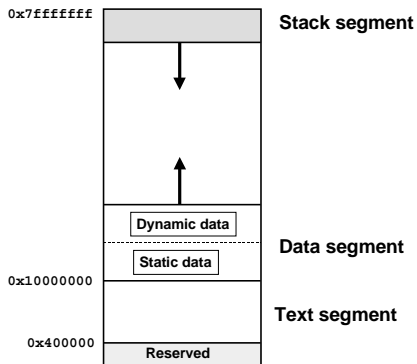
```

.data
s: .space 8
.asciiz "" # set s[8]='\0'
tr: .ascii "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 = l & 0xF
     lb $t1,tr($t1) # j = tr[l]
     sb $t1,0($t0) # s[k] = j
     srl $a0,$a0,4 # l = l >> 4
     addi $t0,$t0,-1 # k--
     bge $t0,$t2,L1 # --> L1 if k>=0
     jr $ra # return to caller
    
```

MIPS: Software conventions for Registers

0	zero constant 0	16	s0 callee saves
1	at reserved for assembler	...	
2	v0 expression evaluation &	23	s7
3	v1 function results	24	t8 temporary (cont'd)
4	a0 arguments	25	t9
5	a1	26	k0 reserved for OS kernel
6	a2	27	k1
7	a3	28	gp Pointer to global area
8	t0 temporary: caller saves	29	sp Stack pointer
...		30	fp frame pointer
15	t7	31	ra Return Address (HW)

Memory Layout



Example2

```
# Program to add together list of 9 numbers.

        .text                # Code
        .align 2
        .globl main

main:
subu    $sp, 40              #\ Push the stack
sw      $ra, 36($sp)        #\ Save return address
sw      $s3, 32($sp)        # \
sw      $s2, 28($sp)        # > Entry Housekeeping
sw      $s1, 24($sp)        # / save registers on stack
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  #/
```

CPS104 MPS.7

©RW Fall 2000

Example2 (cont.)

```
#                Main code segment

again:
        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
```

CPS104 MPS.8

©RW Fall 2000

Example2 (cont.)

```
#                Exit Code

        move   $v0, $0        #\
        lw    $s0, 20($sp)    # \
        lw    $s1, 24($sp)    # \
        lw    $s2, 28($sp)    # \ Closing Housekeeping
        lw    $s3, 32($sp)    # / restore registers
        lw    $ra, 36($sp)    # / load return address
        addu  $sp, 40         # / Pop the stack
        jr    $ra             #/ exit(0) ;
        .end  main           # end of program

#                Data Segment

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

CPS104 MPS.9

©RW Fall 2000

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.

code	service	Arguments	Result	comments
1	print int	\$a0		
2	print float	\$f12		
3	print double	\$f12		
4	print string	\$a0		(address)
5	read integer		integer in \$v0	
6	read float		float in \$f0	
7	read double		double in \$f0	
8	read string	\$a0=buffer, \$a1=length		
9	sbrk	\$a0=amount	address in \$v0	
10	exit			

CPS104 MPS.10

©RW Fall 2000

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:
 - lb, 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, mult,

CPS104 MPS.11

©RW Fall 2000

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

CPS104 MPS.12

©RW Fall 2000

C programming: Many-way action selection

○ C provides 2 FAST methods for choosing one of many possible "next actions"

- 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"

```
        sll    $a0,$a0,2    # scale k by *4 to index words
        lw     $t1, tbl($a0) # t1 = tbl[k]
        jr     $t1         # goto *t1

T1:     add    $v0,$a1,$a2  # f(l,j) = l + j
        done
T2:     sub    $v0,$a1,$a2  # f(l,j) = l-j
        done
...
done:   ...                # print $v0
        ...
        .data
tbl:    .word  T1,T2      # Each word holds an address in the program
```

○ 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

```

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

#define SIZE (some prime number)
int strst, strfree; char *tbl[SIZE]; str[10000];
int look() { /* Looks up str[strst]. If not found, sets strst=strfree, and
tbl[k]=&str[strst], where k is the first null pointer found in tbl.
Returns index in tbl where a pointer to str[strst] is located. */

char *p=&str[strst];
int h=hash(p), h2;

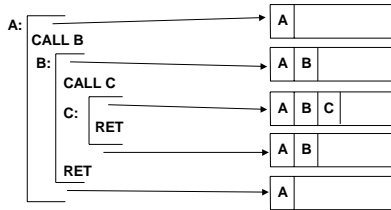
h2 = (h%(SIZE-3))+1; h=h%SIZE;
while( tbl[h] && !strcmp(p,tbl[h]) h = (h+=h2<SIZE) ? H : h-SIZE;

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:



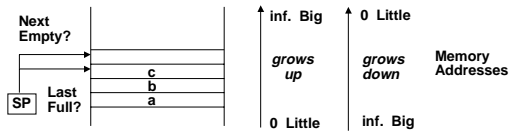
Some machines provide a memory stack as part of the architecture (e.g., VAX)

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:



How is empty stack represented?

Little -> Big/Last Full

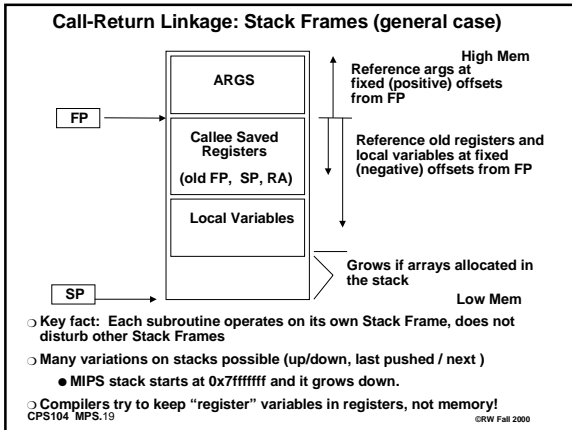
Little -> Big/Next Empty

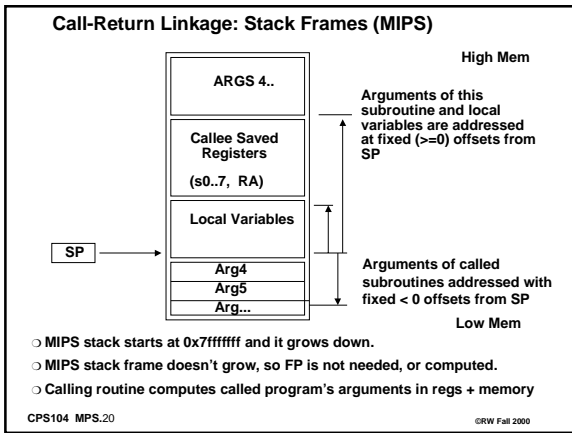
POP: Read from Mem(SP)
Decrement SP

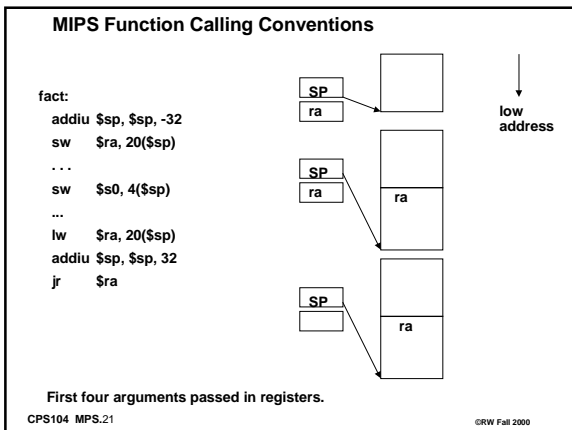
POP: Decrement SP
Read from Mem(SP)

PUSH: Increment SP
Write to Mem(SP)

PUSH: Write to Mem(SP)
Increment SP







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

CPS104 MPS.22

©RW Fall 2000

Radix Sort in C

```
#include <stdio.h>

/* MSB radix sort -- a useful recursive routine */
/* msk must be 2**n, for 3l>=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>h. Then recursively sort the low and the high parts of
the range, separately.
*/
```

CPS104 MPS.23

©RW Fall 2000

Radix Sort in C (cont'd)

```
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);
}
```

CPS104 MPS.24

©RW Fall 2000

Radix Sort in C (main)

```
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;
}
```

CPS104 MPS.25

©RW Fall 2000

Radix Sort in SPIM

```
/* MSB radix sort -- a useful recursive routine */
# Input: l 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
                    # int msk) {
     beqz $a2, retrn # if (msk==0) return;
     blt  $a1, $a0, retrn # if (h < l) return;
     move $t0, $a0
     move $t1, $a1 # l0 = l; h0 = h;
     lw  $t2, 0($a0) # x = *l;
     b   whtst # while (l <= h) {
```

CPS104 MPS.26

©RW Fall 2000

Radix Sort in SPIM (cont'd)

```
whll: and  $t3, $t2, $a2
     beqz $t3, low # if (x & msk) {
#     lw  $t3, 0($a1) # /* Move x to high part */
     sw  $t2, 0($a1) # t = *h;
     subu $a1, $a1, 4 # *h-- = x;
     move $t2, $t3 # x = t;
     b   join
low: # else { # /* Move to low part */
     sw  $t2, 0($a0)
     addu $a0, $a0, 4 # *l++ = x;
     lw  $t2, 0($a0) # x = *l;
join: # }
whtst: # }
     ble $a0, $a1, whll # end while (l <= h) {
```

CPS104 MPS.27

©RW Fall 2000

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 "10"
srl $a2,$a2,1 # msk = msk >> 1
jal rsrt # rsrt(l0,l-1,msk>>1); ($a1 is correct)
lw $a0,20($sp) # h+1
lw $a1,12($sp) # h0
lw $a2,16($sp) # original msk
srl $a2,$a2,1 # msk = msk >> 1
jal rsrt # rsrt(h+1,h0,msk>>1);
# }
retrn:
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];
nl: .asciiz "\n"
```

CPS104 MPS.28

©RW Fall 2000

Radix Sort in SPIM (cont'd)

```
.text
.align 2
.globl main
main: subu $sp,$sp,4
sw $ra,4($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
mwhll: li $v0,5 # code for read_int
syscall #
bltz $v0,m12 # break if # < 0 (Can't see EOF)
sw $v0,0($a0) # A[i] = #
addu $a0,$a0,4 # i++
blt $a0,$t0,mwhll # end while (i<1000)
m12: 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);
```

CPS104 MPS.29

©RW Fall 2000

Radix Sort in SPIM (cont'd)

```
la $a1,A # j = &A[0]
lw $t0,8($sp) # $t0 = &A[i] for end test
mfl1: lw $a0,0($a1) # Arg to print_int = *$a1
li $v0,1 # code for print_int
syscall # print_int(A[j])
la $a0,nl # address of new_line character string
li $v0,4 # code for print_string
syscall # print_string("\n")
addu $a1,$a1,4 #
blt $a1,$t0,mfl1 # end for (j=0; j<i; j++)

move $v0,$0
lw $ra,4($sp)
addu $sp,$sp,8
jr $ra # return 0;
# }
```

CPS104 MPS.30

©RW Fall 2000
