MIPS examples

- We’ve learned all of the important features of the MIPS instruction set architecture, so now it’s time for some examples!
  – First we’ll see a nested function, which calls another function.
  – Next up is a demonstration of recursion.
  – Finally we’ll work with some C-style strings.
- This week’s sections will discuss how structures and objects can be stored and manipulated in MIPS. You’ll also get some debugging practice, which will be useful for the assignment that will appear later today.

©2001-2003 Howard Huang

Combinations

- Writing a function to compute combinations based on the fact code from last time will illustrate the implementation of nested functions.
- A mathematical definition of combinations is:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- The corresponding C or Java code is shown below.

```
int comb(int n, int k)
{
   return fact(n) / fact(k) / fact(n-k);
}
```

- Looks easy, right?

Look before you leap!

```
int comb(int n, int k)
{
  return fact(n) / fact(k) / fact(n-k);
}
```

- We use MIPS registers to represent the argument and return values.
  - Arguments n and k are passed in caller-saved registers $a0 and $a1.
  - The return value (an integer) is placed in $v0.
- The arguments n and k are used multiple times—in particular, *after* the function calls fact(n) and fact(k). We'll have to save the original values of $a0 and $a1, to ensure they aren't overwritten by fact.
- comb is a nested function, which means it also acts as a callee. We must also preserve callee-saved registers such as $ra properly.
- The return expression involves two divisions and requires one temporary register in MIPS. We'll use $s0 for illustrative purposes.

Callee-saved registers

- To summarize, comb will need to preserve and restore four registers:

  $ra $s0 $a0 $a1

- comb is the *callee* in relation to whoever calls it (e.g., main).
  - Thus, comb is responsible for preserving the callee-saved registers $ra and $s0, which it will modify.
  - It's easiest to save them at the beginning of the function, and to restore them right before returning.

```
comb:
    sub   $sp, $sp, 8
    sw    $ra, 0($sp)
    sw    $s0, 4($sp)

    ... main body ...

    lw    $ra, 0($sp)
    lw    $s0, 4($sp)
    addi  $sp, $sp, 8
    jr    $ra
```

Caller-saved registers

- However, comb is the *caller* in relation to the fact function.
  - This means comb must store the caller-saved registers $a0 and $a1 before it calls fact, and restore them afterwards.
  - We'll allocate two additional words on the stack for $a0 and $a1. They'll be restored as necessary in the function.
- Is it possible to preserve $a0 and $a1 in any of the other registers, instead of the stack?

```
comb:
    sub  $sp, $sp, 16
    sw   $ra, 0($sp)
    sw   $s0, 4($sp)
    sw   $a0, 8($sp)
    sw   $a1, 12($sp)

    ... main body ...

    lw   $ra, 0($sp)
    lw   $s0, 4($sp)
    addi $sp, $sp, 16
    jr   $ra
```

The rest of comb

- The call fact(n) is easy.
  - The value n is also the first argument of comb, so it is already in $a0.
  - The result is saved in register $s0.

```
# fact(n)
jal   fact
move  $s0, $v0
```

- For fact(k):
  - We have to load k from memory, in case fact(n) overwrote $a1.
  - $s0 is updated with fact(n) / fact(k).

```
# fact(k)
lw    $a0, 12($sp)
jal   fact
div   $s0, $s0, $v0
```

- For fact(n-k):
  - We have to restore both n and k.
  - The final result of comb ends up in $s0.

```
# fact(n-k)
lw    $a0, 8($sp)
lw    $a1, 12($sp)
sub   $a0, $a0, $a1
jal   fact
div   $s0, $s0, $v0
```

The whole kit and caboodle

- The whole function is shown on the right.
- Don't forget to return the result! We must move $s0 to $v0 before restoring registers and returning.
- That's a lot of work for a simple one-line C function.

```
comb:
    sub   $sp, $sp, 16
    sw    $ra, 0($sp)
    sw    $s0, 4($sp)
    sw    $a0, 8($sp)
    sw    $a1, 12($sp)
    jal   fact
    move  $s0, $v0
    lw    $a0, 12($sp)
    jal   fact
    div   $s0, $s0, $v0
    lw    $a0, 8($sp)
    lw    $a1, 12($sp)
    sub   $a0, $a0, $a1
    jal   fact
    div   $s0, $s0, $v0
    move  $v0, $s0
    lw    $ra, 0($sp)
    lw    $s0, 4($sp)
    addi  $sp, $sp, 16
    jr    $ra
```

Famous Fibonacci Function

- The Fibonacci sequence is often expressed recursively:

$$fib(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

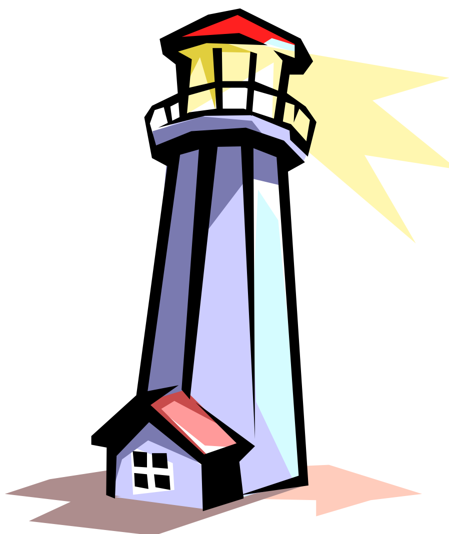
- This is easy to convert into a C program.

```
int fib(int n)
{
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

- The translation to MIPS is not bad if you understood the first example.

Some observations about fib

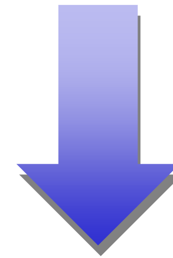
- This function is similar to comb in some ways.
  - fib calls another function (itself), so we will have to save $ra.
  - We need to save the argument n across the first recursive call.
  - We need a temporary register for the result of the first call.
- A recursive function also acts as *both* caller and callee.
  - Calling the same function guarantees that the same registers will be used, and overwritten if we're not careful.
  - So to be careful, we have to save *every* register that is used.



The base case

- The base case of the recursion is easy.
- If $a0 is less than 1, then we just return it. (We won't worry about testing for valid inputs here.)
- This part of the code does not involve any function calls, so there's no need to preserve any registers.

```
int fib(int n)
{
  if (n <= 1)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```



```
fib:
      bgt   $a0, 1,recurse
      move  $v0, $a0
      jr    $ra
```

Doing the recursion

- First save $ra and the argument $a0. An extra word is allocated on the stack to save the result of fib(n-1).

```
recurse:
    sub   $sp, $sp, 12
    sw    $ra, 0($sp)
    sw    $a0, 4($sp)
```

- The argument n is already in $a0, so we can decrement it and then "jal fib" to implement the fib(n-1) call. The result is put into the stack.

```
    addi  $a0, $a0, -1
    jal   fib
    sw    $v0, 8($sp)
```

- Retrieve n, and then call fib(n-2).

```
    lw    $a0, 4($sp)
    addi  $a0, $a0, -2
    jal   fib
```

- The results are summed and put in $v0.

```
    lw    $v1, 8($sp)
    add   $v0, $v0, $v1
```

- We only need to restore $ra before popping our frame and saying bye-bye.

```
    lw    $ra, 0($sp)
    addi  $sp, $sp, 12
    jr    $ra
```

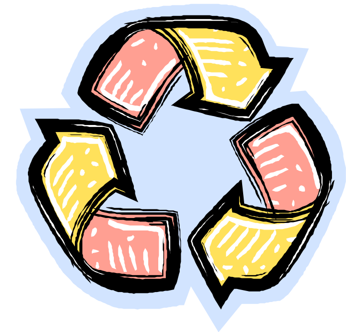
The complete fib

```
fib:
   bgt   $a0, 1, recurse
   move  $v0, $a0
   jr    $ra

recurse:
   sub   $sp, $sp, 12
   sw    $ra, 0($sp)
   sw    $a0, 4($sp)

   addi  $a0, $a0, -1
   jal   fib
   sw    $v0, 8($sp)
   lw    $a0, 4($sp)
   addi  $a0, $a0, -2
   jal   fib
   lw    $v1, 8($sp)
   add   $v0, $v0, $v1

   lw    $ra, 0($sp)
   addi  $sp, $sp, 12
   jr    $ra
```



Representing strings

- A C-style string is represented by an array of bytes.
  - Elements are one-byte ASCII codes for each character.
  - A 0 value marks the end of the array.

32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	”	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	’	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	I	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o	127	del

String manipulation

- For example, “Harry Potter” can be stored as a 13-byte array.

72	97	114	114	121	32	80	111	116	116	101	114	0
H	a	r	r	y		P	o	t	t	e	r	\0

- We can convert a string to uppercase by manipulating the ASCII values.
  - Lowercase letters a-z have ASCII codes from 97 to 122.
  - Uppercase letters A-Z range from 65 to 90.
  - A lowercase letter can be converted to uppercase by subtracting 32 from the ASCII code (e.g., 97 – 32 = 65).

Two versions of ToUpper

- Both of these loop through a string, subtracting 32 from lowercase letters, until they reach the terminating 0.
- The first one accesses letters by indexing the array str, and incrementing the index on each loop iteration.
- The second one uses a pointer that produces a letter when it's dereferenced. The pointer is incremented by one on every loop iteration.

```
void ToUpper(char str[])
{
  int i = 0;
  while (str[i] != 0) {
    if (str[i] >= 97 && str[i] <= 122)
      str[i] = str[i] - 32;
    i++;
  }
}
```

```
void ToUpper(char *str)
{
  char *s = str;
  while (*s != 0) {
    if (*s >= 97 && *s <= 122)
      *s = *s - 32;
    s++;
  }
}
```

Array version

- A direct translation of the array version means each iteration of the loop must re-compute the address of str[i]—requiring two additions.

```
toupper:
    li   $t0, 0            # $t0 = i
loop:
    add  $t1, $t0, $a0     # $t1 = &str[i]
    lb   $t2, 0($t1)       # $t2 = str[i]
    beq  $t2, $0, exit     # $t2 = 0?
    blt  $t2, 97, next     # $t2 < 97?
    bgt  $t2, 122, next    # $t2 > 122?
    sub  $t2, $t2, 32      # convert
    sb   $t2, 0($t1)       # and store back
next:
    addi $t0, $t0, 1       # i++
    j    loop
exit:
    jr   $ra
```

- Here we work with bytes, but if the array contained integers, this address computation would require a multiplication as well.

Pointer version

- Instead, we could use a register to hold the exact address of the current element. Each time through the loop, we'll increment this register to point to the next element.

```
toupper:
    lb   $t2, 0($a0)       # $t2 = *s
    beq  $t2, $0, exit     # $t2 = 0?
    blt  $t2, 97, next     # $t2 < 97?
    bgt  $t2, 122, next    # $t2 > 122?
    sub  $t2, $t2, 32      # convert
    sb   $t2, 0($a0)       # and store back
next:
    addi $a0, $a0, 1       # s++
    j    toupper
exit:
    jr   $ra
```

- With an array of words, we would have to increment the pointer by 4 on each iteration. But then we would only need one addition, instead of two additions and a multiplication.

Side by side

```
toupper:
    li   $t0, 0
loop:
    add  $t1, $t0, $a0
    lb   $t2, 0($t1)
    beq  $t2, $0, exit
    blt  $t2, 97, next
    bgt  $t2, 122, next
    sub  $t2, $t2, 32
    sb   $t2, 0($t1)
next:
    addi $t0, $t0, 1
    j    loop
exit:
    jr   $ra
```

```
toupper:



    lb   $t2, 0($a0)
    beq  $t2, $0, exit
    blt  $t2, 97, next
    bgt  $t2, 122, next
    sub  $t2 $t2, 32
    sb   $t2, 0($a0)
next:
    addi $a0, $a0, 1
    j    toupper
exit:
    jr   $ra
```

- Another similar example is in Section 3.11 of Hennessy and Patterson.

Summary

- These three examples demonstrate that writing large, modular programs in assembly language is difficult!
  - It's hard to figure out how to best use the limited number of registers. (Register allocation is an important problem in writing compilers.)
  - We must always follow the MIPS function call conventions regarding the passing and returning of values and preservation of registers.
  - There is only one addressing mode that must be used for all memory accesses, whether to arguments or stack-allocated space.
- You'll get some good programming practice on the machine problem.