

inst.eecs.berkeley.edu/~cs61c  
**UCB CS61C : Machine Structures**

**Lecture 11 – Introduction to MIPS Procedures II & Logical Ops**


**2013-02-15**

**Sr Lecturer SOE**  
**Dan Garcia**

**VIRTUAL HUMANS...**

Prof Paul Debevec (UC Berkeley PhD 1996) at USC has been working to create virtual humans to keep alive the memory AND INTERACTIONS w/people into a 3D hologram. He is recording the Holocaust survivors, who tell their story, answering 500 questions about themselves. They're in a race against time...

[www.washingtonpost.com/national/holograms-seen-as-tools-to-teach-future-generations-about-holocaust-retel1-survivors-stories/2013/02/02/558cab32-6d59-11e2-8f4f-2ab96162ba9\\_story\\_1.html](http://www.washingtonpost.com/national/holograms-seen-as-tools-to-teach-future-generations-about-holocaust-retel1-survivors-stories/2013/02/02/558cab32-6d59-11e2-8f4f-2ab96162ba9_story_1.html)

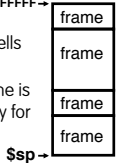


### Review

- Functions called with **jal**, return with **jr \$ra**.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
- Instructions we know so far...
  - Arithmetic: **add, addi, sub, addu, addiu, subu**
  - Memory: **lw, sw, lb, sb**
  - Decision: **beq, bne, slt, slti, sltu, sltiu**
  - Unconditional Branches (Jumps): **j, jal, jr**
- Registers we know so far
  - All of them!
  - There are CONVENTIONS when calling procedures!

### The Stack (review)

- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames

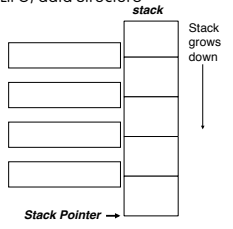


### Stack

- Last In, First Out (LIFO) data structure

```

main ()
{ a(0);
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
  
```

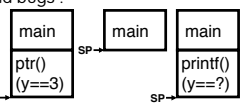


### Who cares about stack management?

- Pointers in C allow access to deallocated memory, leading to hard-to-find bugs!

```

int *ptr () {
  int y;
  y = 3;
  return &y; }
main () {
  int *stackAddr, content;
  stackAddr = ptr();
  content = *stackAddr;
  printf("%d", content); /* 3 */
  content = *stackAddr;
  printf("%d", content); /* 13451514 */
}
  
```



### Memory Management

- How do we manage memory?
- Code, Static storage are easy: they never grow or shrink
- Stack space is also easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time

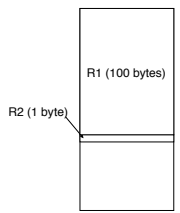
### Heap Management Requirements

- Want **malloc()** and **free()** to run quickly.
- Want minimal memory overhead
- Want to avoid *fragmentation*\* – when most of our free memory is in many small chunks
  - In this case, we might have many free bytes but not be able to satisfy a large request since the free bytes are not contiguous in memory.

\* This is technically called *external fragmentation*

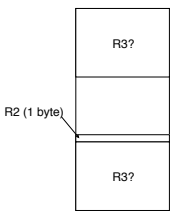
### Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
  - Request R3 for 50 bytes



### Heap Management

- An example
  - Request R1 for 100 bytes
  - Request R2 for 1 byte
  - Memory from R1 is freed
    - Memory has become fragmented!
    - We have to keep track of the two *freespace* regions
  - Request R3 for 50 bytes
    - We have to search the data structures holding the freespace to find one that will fit! Choice here...



## Administrivia

- Project (P1 1) due Sunday @ 23:59:59pm
  - Quick Peer Instruction question: how are you doing on part 1 of the project?
    - 10, 20% done
    - 20, 40% done
    - 40, 60% done
    - 60, 80% done
    - 80, 100% done
- TAs, anything?



## Register Conventions (1/4)

- Caller**: the calling function
- Callee**: the function being called
- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**jal**) and which may be changed.



## Register Conventions (2/4) – saved

- \$0**: No Change. Always 0.
- \$s0-\$s7**: Restore if you change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.
- \$sp**: Restore if you change. The stack pointer must point to the same place before and after the **jal** call, or else the caller won't be able to restore values from the stack.
- HINT -- All saved registers start with S!



## Register Conventions (2/4) – volatile

- \$ra**: Can Change. The **jal** call itself will change this register. Caller needs to save on stack if nested call.
- \$v0-\$v1**: Can Change. These will contain the new returned values.
- \$a0-\$a3**: Can change. These are volatile argument registers. Caller needs to save if they are needed after the call.
- \$t0-\$t9**: Can change. That's why they're called temporary: any procedure may change them at any time. Caller needs to save if they'll need them afterwards.



## Register Conventions (4/4)

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a **jal** call.
  - Function E must save any S (saved) registers it intends to use before garbling up their values, and restore them after done garbling
- Remember: caller/callee need to save only temporary/saved registers they are using, not all registers.



## Peer Instruction

```
r: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
...   ### PUSH REGISTER(S) TO STACK?
jal e # Call e
...   # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to caller of r

e: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to r
```

What does **r** have to push on the stack before "jal e"?

- 1 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 2 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 3 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 4 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)
- 5 of (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra)



## Peer Instruction Answer

```
r: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
...   ### PUSH REGISTER(S) TO STACK?
jal e # Call e
...   # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to caller of r

e: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
jr $ra # Return to r
```

What does **r** have to push on the stack before "jal e"?

- |           | Saved                                | Volatile! -- need to push |
|-----------|--------------------------------------|---------------------------|
| a) 1 of   | (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra) |                           |
| b) 2 of   | (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra) |                           |
| c) 3 of   | (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra) |                           |
| ☺ d) 4 of | (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra) |                           |
| e) 5 of   | (\$s0, \$sp, \$v0, \$t0, \$a0, \$ra) |                           |



## "And in Conclusion..."

- Register Conventions: Each register has a purpose and limits to its usage. Learn these and follow them, even if you're writing all the code yourself.
- Logical and Shift Instructions
  - Operate on bits individually, unlike arithmetic, which operate on entire word.
  - Use to isolate fields, either by masking or by shifting back and forth.
  - Use shift left logical, **sll**, for multiplication by powers of 2
  - Use shift right logical, **srl**, for division by powers of 2 of unsigned numbers (unsigned int)
  - Use shift right arithmetic, **sra**, for division by powers of 2 of signed numbers (int)
- New Instructions: **and**, **andi**, **or**, **ori**, **sll**, **srl**, **sra**



## Bonus slides

- These are extra slides that used to be included in lecture notes, but have been moved to this, the "bonus" area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

# Bonus



## Bitwise Operations

- So far, we've done arithmetic (**add**, **sub**, **addi**), mem access (**lw** and **sw**), & branches and jumps.
- All of these instructions view contents of register as a single quantity (e.g., signed or unsigned int)
- New Perspective: View register as 32 raw bits rather than as a single 32-bit number
  - Since registers are composed of 32 bits, wish to access individual bits (or groups of bits) rather than the whole.
- Introduce two new classes of instructions
  - Logical & Shift Ops



## Logical Operators (1/3)

- Two basic logical operators:
  - AND: outputs 1 only if all inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output

a	b	a AND b	a OR b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

a	a AND b	a OR b
0	0	b
1	b	1



## Logical Operators (2/3)

- Logical Instruction Syntax:
  - 2,3,4
  - where
    - operation name
    - register that will receive value
    - first operand (register)
    - second operand (register) or immediate (numerical constant)
- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware



## Logical Operators (3/3)

- Instruction Names:
  - and**, **or**: Both of these expect the third argument to be a register
  - andi**, **ori**: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
  - C: Bitwise AND is `&` (e.g., `z = x & y;`)
  - C: Bitwise OR is `|` (e.g., `z = x | y;`)



## Uses for Logical Operators (1/3)

- Note that **anding** a bit with **0** produces a **0** at the output while **anding** a bit with **1** produces the original bit.
- This can be used to create a mask.
  - Example:
 

```

1011 0110 1010 0100 0011 1101 1001 1010
mask: 0000 0000 0000 0000 0000 1111 1111 1111
                    
```

    - The result of **anding** these:
 

```

0000 0000 0000 0000 0000 1101 1001 1010
                    
```

 mask last 12 bits



## Uses for Logical Operators (2/3)

- The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting to all 0s).
- Thus, the **and** operator can be used to set certain portions of a bitstring to **0s**, while leaving the rest alone.
  - In particular, if the first bitstring in the above example were in `$t0`, then the following instruction would mask it:
 

```
and $t0, $t0, 0xFFF
```



## Uses for Logical Operators (3/3)

- Similarly, note that **oring** a bit with **1** produces a **1** at the output while **oring** a bit with **0** produces the original bit.
- Often used to force certain bits to 1s.
  - For example, if `$t0` contains `0x12345678`, then after this instruction:
 

```
ori $t0, $t0, 0xFFFF
```

 ... `$t0` will contain `0x1234FFFF`
    - (i.e., the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).



## Example: Fibonacci Numbers 1/8

- The Fibonacci numbers are defined as follows:
 
$$F(n) = F(n - 1) + F(n - 2)$$

$$F(0) \text{ and } F(1) \text{ are defined to be } 1$$
- In scheme, this could be written:
 

```

(define (Fib n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (else (+ (Fib (- n 1))
                  (Fib (- n 2))))))
                    
```



## Example: Fibonacci Numbers 2/8

- Rewriting this in C we have:
 

```

int fib(int n) {
  if(n == 0) { return 1; }
  if(n == 1) { return 1; }
  return (fib(n - 1) + fib(n - 2));
}
                    
```



### Example: Fibonacci Numbers 3/8

- Now, let's translate this to MIPS!
- You will need space for three words on the stack
- The function will use one \$s register, \$s0
- Write the Prologue:

```
fib:
addi $sp, $sp, -12 # Space for three words
sw $ra, 8($sp)    # Save return address
sw $s0, 4($sp)    # Save s0
```



### Example: Fibonacci Numbers 4/8

- Now write the Epilogue:

```
fib:
lw $s0, 4($sp)    # Restore $s0
lw $ra, 8($sp)    # Restore return address
addi $sp, $sp, 12 # Pop the stack frame
jr $ra            # Return to caller
```



### Example: Fibonacci Numbers 5/8

- Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
if(n == 0) { return 1; } /*Translate Me!*/
if(n == 1) { return 1; } /*Translate Me!*/
return (fib(n - 1) + fib(n - 2));
}
```

```
addi $v0, $zero, 1 # $v0 = 1
beq $a0, $zero, fin #
addi $t0, $zero, 1 # $t0 = 1
beq $a0, $t0, fin #
```

Continued on next slide. . .



### Example: Fibonacci Numbers 6/8

- Almost there, but be careful, this part is tricky!

```
int fib(int n) {
return (fib(n - 1) + fib(n - 2));
}
```

```
addi $a0, $a0, -1 # $a0 = n - 1
sw $a0, 0($sp)   # Need $a0 after jal
jal fib          # fib(n - 1)
lw $a0, 0($sp)   # restore $a0
addi $a0, $a0, -1 # $a0 = n - 2
```



### Example: Fibonacci Numbers 7/8

- Remember that \$vo is caller saved!

```
int fib(int n) {
return (fib(n - 1) + fib(n - 2));
}
```

```
add $s0, $v0, $zero # Place fib(n - 1)
# somewhere it won't get
# clobbered
jal fib              # fib(n - 2)
add $v0, $v0, $s0  # $v0 = fib(n-1) + fib(n-2)
```

To the epilogue and beyond. . .



### Example: Fibonacci Numbers 8/8

- Here's the complete code for reference:

```
fib: addi $sp, $sp, -12      lw $a0, 0($sp)
sw $ra, 8($sp)            addi $a0, $a0, -1
sw $s0, 4($sp)            add $s0, $v0, $zero
addi $v0, $zero, 1       jal fib
beq $a0, $zero, fin      add $v0, $v0, $s0
addi $t0, $zero, 1       fin: lw $s0, 4($sp)
beq $a0, $t0, fin        lw $ra, 8($sp)
addi $a0, $a0, -1        sw $a0, 0($sp)
sw $a0, 0($sp)           addi $sp, $sp, 12
jal fib                   jr $ra
```



### Bonus Example: Compile This (1/5)

```
main() {
int i,j,k,m; /* i-m: $s0-$s3 */
...
i = mult(j,k); ...
m = mult(i,i); ...
}
int mult (int mcand, int mlrier){
int product;
product = 0;
while (mlrier > 0) {
product += mcand;
mlrier -= 1; }
return product;
}
```



### Bonus Example: Compile This (2/5)

```
__start:
...
add $a0, $s1, $0 # arg0 = j
add $a1, $s2, $0 # arg1 = k
jal mult        # call mult
add $s0, $v0, $0 # i = mult()
add $a0, $s0, $0 # arg0 = i
add $a1, $s0, $0 # arg1 = i
jal mult        # call mult
add $s3, $v0, $0 # m = mult()
...
j __exit      main() {
int i,j,k,m; /* i-m: $s0-$s3 */
...
i = mult(j,k); ...
m = mult(i,i); ... }
```



### Bonus Example: Compile This (3/5)

- Notes:
  - main function ends with a jump to \_\_exit, not jr \$ra, so there's no need to save \$ra onto stack
  - all variables used in main function are saved registers, so there's no need to save these onto stack



### Bonus Example: Compile This (4/5)

```

mult:
Loop: add $t0,$0,$0 # prod=0
      slt $t1,$0,$a1 # mlr > 0?
      beq $t1,$0,Fin # no->Fin
      add $t0,$t0,$a0 # prod+=mc
      addi $a1,$a1,-1 # mlr-=1
      j Loop # goto Loop
Fin:   add $v0,$t0,$0 # $v0=prod
      jr $ra # return
      int mult (int mcand, int mlr){
      int product = 0;
      while (mlr > 0) {
      product += mcand;
      mlr -= 1;
      }
      return product;
      }
    
```



### Bonus Example: Compile This (5/5)

- Notes:
  - no `jal` calls are made from `mult` and we don't use any saved registers, so we don't need to save anything onto stack
  - temp registers are used for intermediate calculations (could have used `s` registers, but would have to save the caller's on the stack.)
  - `$a1` is modified directly (instead of copying into a temp register) since we are free to change it
  - result is put into `$v0` before returning (could also have modified `$v0` directly)



### Parents leaving for weekend analogy (1/5)

- Parents (main) leaving for weekend
- They (caller) give keys to the house to kid (callee) with the rules (calling conventions):
  - You can trash the temporary room(s), like the den and basement (registers) if you want, we don't care about it
  - BUT you'd better leave the rooms (registers) that we want to save for the guests untouched. "these rooms better look the same when we return!"
- Who hasn't heard this in their life?



### Parents leaving for weekend analogy (2/5)

- Kid now "owns" rooms (registers)
- Kid wants to use the saved rooms for a wild, wild party (computation)
- What does kid (callee) do?
  - Kid takes what was in these rooms and puts them in the garage (memory)
  - Kid throws the party, trashes everything (except garage, who ever goes in there?)
  - Kid restores the rooms the parents wanted saved after the party by replacing the items from the garage (memory) back into those saved rooms



### Parents leaving for weekend analogy (3/5)

- Same scenario, except before parents return and kid replaces saved rooms...
- Kid (callee) has left valuable stuff (data) all over.
  - Kid's friend (another callee) wants the house for a party when the kid is away
  - Kid knows that friend might trash the place destroying valuable stuff!
  - Kid remembers rule parents taught and now becomes the "heavy" (caller), instructing friend (callee) on good rules (conventions) of house.



### Parents leaving for weekend analogy (4/5)

- If kid had data in temporary rooms (which were going to be trashed), there are three options:
  - Move items directly to garage (memory)
  - Move items to saved rooms whose contents have already been moved to the garage (memory)
  - Optimize lifestyle (code) so that the amount you've got to shlep stuff back and forth from garage (memory) is minimized.
    - Mantra: "Minimize register footprint"
- Otherwise: "Dude, where's my data?!"



### Parents leaving for weekend analogy (5/5)

- Friend now "owns" rooms (registers)
- Friend wants to use the saved rooms for a wild, wild party (computation)
- What does friend (callee) do?
  - Friend takes what was in these rooms and puts them in the garage (memory)
  - Friend throws the party, trashes everything (except garage)
  - Friend restores the rooms the kid wanted saved after the party by replacing the items from the garage (memory) back into those saved rooms



### Shift Instructions (review) (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

```

0001 0010 0011 0100 0101 0110 0111 1000
      |-----|
0000 0000 0001 0010 0011 0100 0101 0110
    
```

  - Example: shift left by 8 bits

```

0001 0010 0011 0100 0101 0110 0111 1000
      |-----|
0011 0100 0101 0110 0111 1000 0000 0000
    
```



### Shift Instructions (2/4)

- Shift Instruction Syntax:
  - 1, 2, 3, 4
  - ...where
    - 1) operation name
    - 2) register that will receive value
    - 3) first operand (register)
    - 4) shift amount (constant < 32)
- MIPS shift instructions:
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending



### Shift Instructions (3/4)

- Example: shift right arithmetic by 8 bits

⇒ 0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift right arithmetic by 8 bits

⇒ 1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

*Cal*

CMSC 111 Introduction to MIPS / Procedures 1 & Logical Ops (4)

March, Spring 2018 © UCSD

### Shift Instructions (4/4)

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

`a *= 8;` (in C)  
would compile to:

`sll $s0,$s0,3` (in MIPS)

- Likewise, shift right to divide by powers of 2  
(rounds towards  $-\infty$ )
  - remember to use `sra`

*Cal*

CMSC 111 Introduction to MIPS / Procedures 1 & Logical Ops (5)

March, Spring 2018 © UCSD