Sr Lecturer SOE
Dan Garcia

`inst.eecs.berkeley.edu/~cs61c`

UCB CS61C : Machine Structures

Lecture 11 – Introduction to MIPS Procedures II & Logical Ops

2014-09-24

STUDENTS SAVED BY STRESS-DETECTION APP

Students at Dartmouth allowed researchers to silently measure their stress level (via noting their irregular sleep patterns, reduced social activity (calls, and social networking). Two students who exhibited high stress levels were allowed to get incompletes, since the researchers noted their stress was too high.

`www.newscientist.com/article/mg22329863.300-failing-students-saved-by-stressdetecting-app.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`**

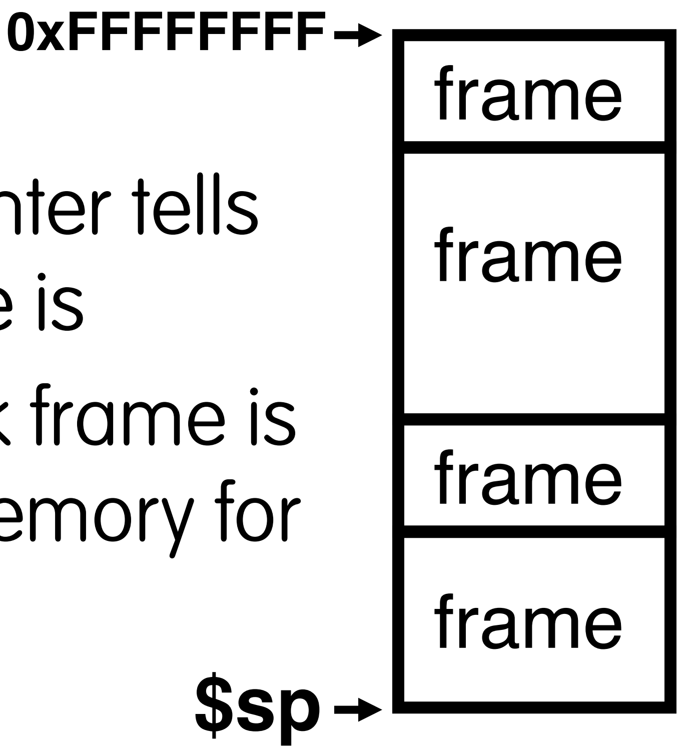
  Memory: **`lw, sw, lb, sb`**

  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)

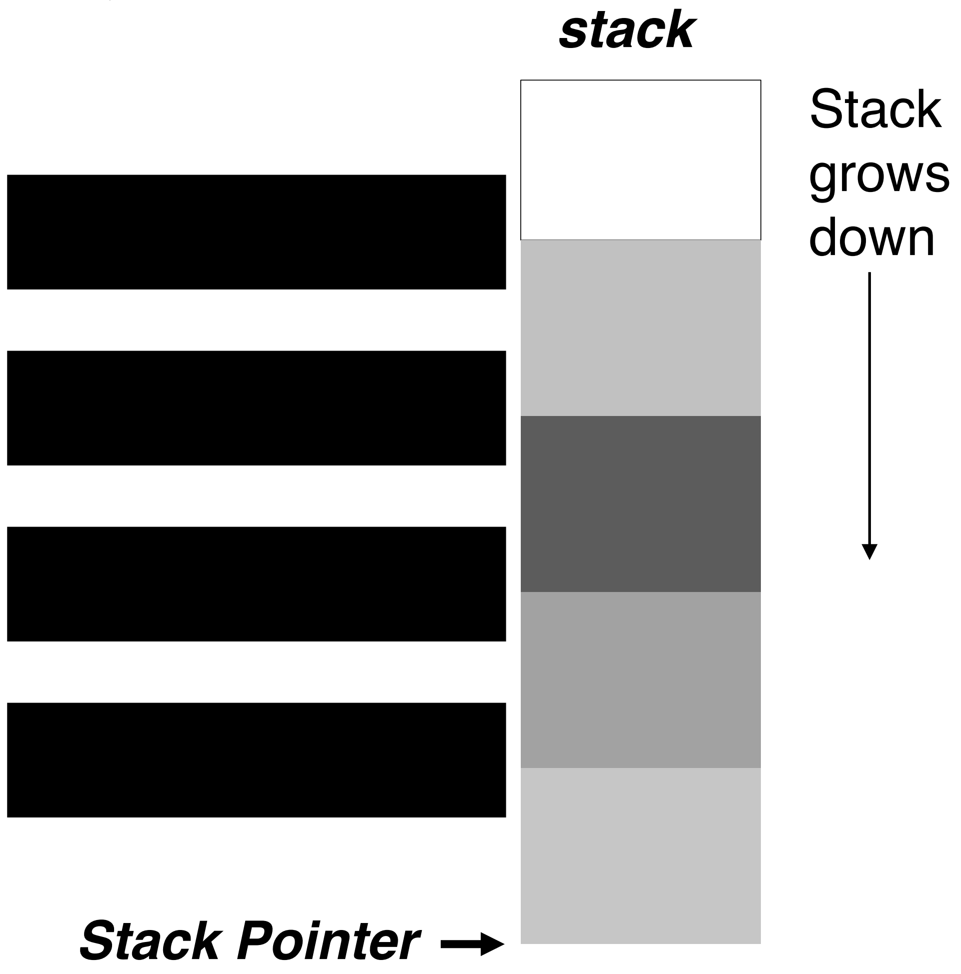
- Stack frame includes:
  - Return "instruction" address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer tells where bottom of stack frame is
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames



Stack

- Last In, First Out (LIFO) data structure

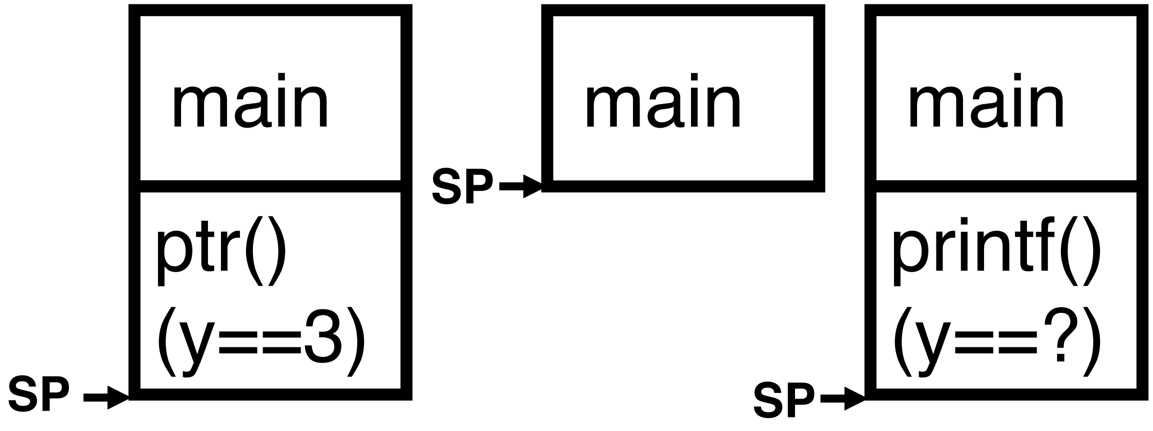
```
main ()
{ a(0);
}
  void a (int m)
  { b(1);
  }
    void b (int n)
    { c(2);
    }
      void c (int o)
      { d(3);
      }void d (int p)
        {
        }
```



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

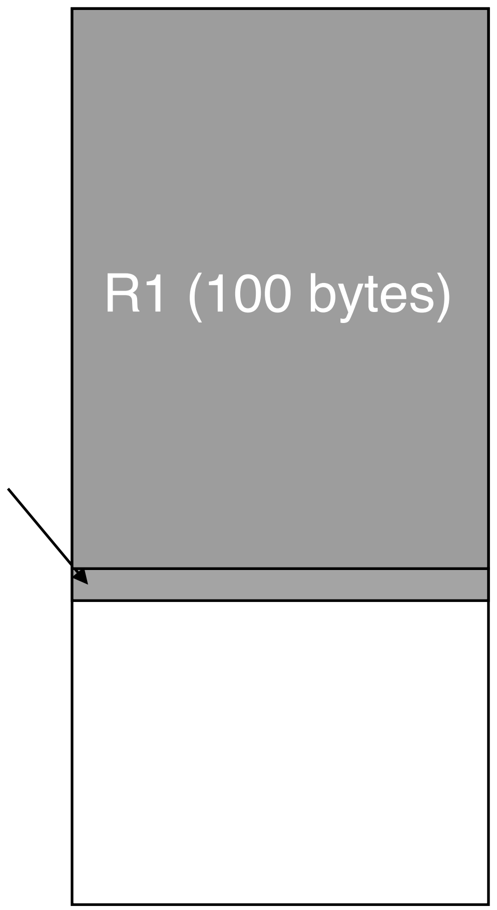
Heap Management Requirements

- 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 fragmention*

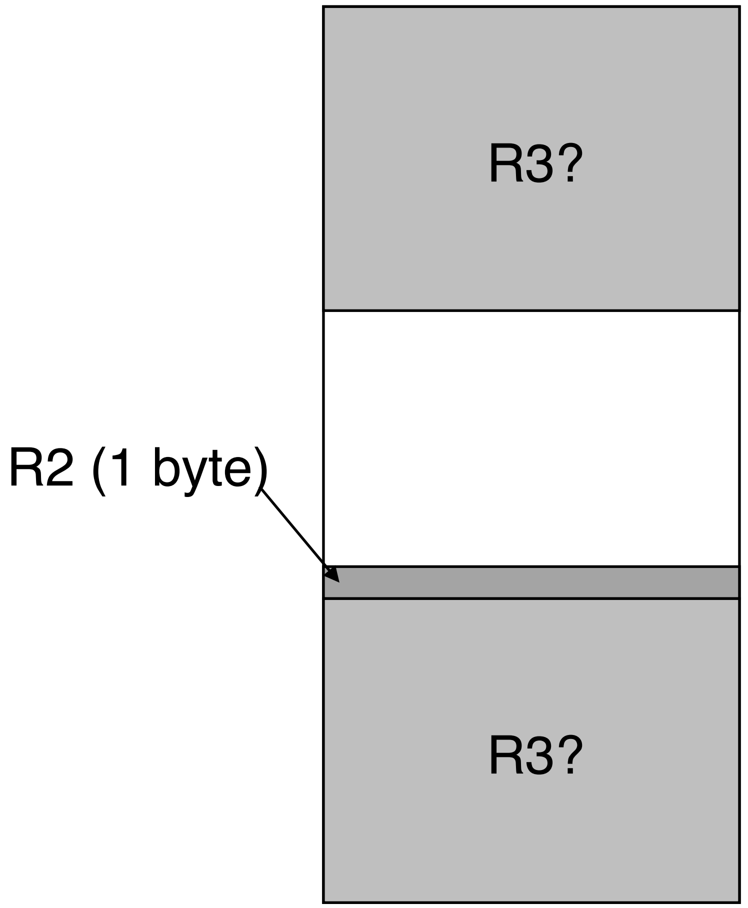
Heap Management

- 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...



Register Conventions (1/4)

- Calle<u>R</u>: the calling function
- Calle<u>E</u>: 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-$s7`**: Restore if you change. Very important, that's why they're called <u>saved</u> registers. If the <u>callee</u> changes these in any way, it must restore the original values before returning.
- **`$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: calle<u>r</u>/calle<u>e</u> 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`**"?

```
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 <u>shift left logical,</u> **`sll`**, for multiplication by powers of 2
  - Use <u>shift right logical,</u> **`srl`**, for division by powers of 2 of unsigned numbers (**`unsigned int`**)
  - Use <u>shift right arithmetic,</u> **`sra`**, for division by powers of 2 of signed numbers (**`int`**)
- New Instructions: **`and, andi, or, ori, sll, srl, sra`**