Review Session 1
Fri. Jan 19, 2:15 pm – 3:05 pm
Thornton 102

1. Agenda

- Announcements
- Homework hints
- MIPS instruction set
- Some PA1 thoughts
- MIPS procedure call convention and example

2. Announcements

- Homework 1 (can work in groups of 2):
  Due date extended to 5 pm on Thursday, January 25.
- The TA email list is ee108b-win0607-tas@lists.stanford.edu
- PA 1 (individual):
  Assigned: yesterday
  Due 11:59pm on Thursday, February 8, you have 3 weeks.
- Lab 1 (groups of 2):
  Due: Tuesday, January 30, you have 12 days.
- Office hours:
  Drew: Tuesdays 1pm – 3pm in P129
  Yi: Tuesdays 3pm – 5pm P129
- Access to P128 (computer cluster) and P129. Email the TA list for login info.

3. Homework hints

Problem 1:

- Part b: you can assume all 5000 elements are unequal.
- Part c:
  Total Execution Time = (total clk cyc.)/(clk cyc. rate)
- Part d: How does new instruction change the inner loop?
  Reduces instruction count, but cycle time has increased…

Problem 2:

- Go line by line of the C code and translate into assembly.
- Tail recursion.

Problem 3:

- Similar to problem 1.
- No more than 5 bugs.

Problem 4:

- Pseudo instructions.
- Small has 16 bits and Big has 32 bits
- Break big numbers into upper and lower halves
- You may use the notation upper(big) and lower(big) to extract a portion of a big number.

Please consult the bulletin board first for homework questions, email to the TA list if your question is not already answered.

4. MIPS instruction set

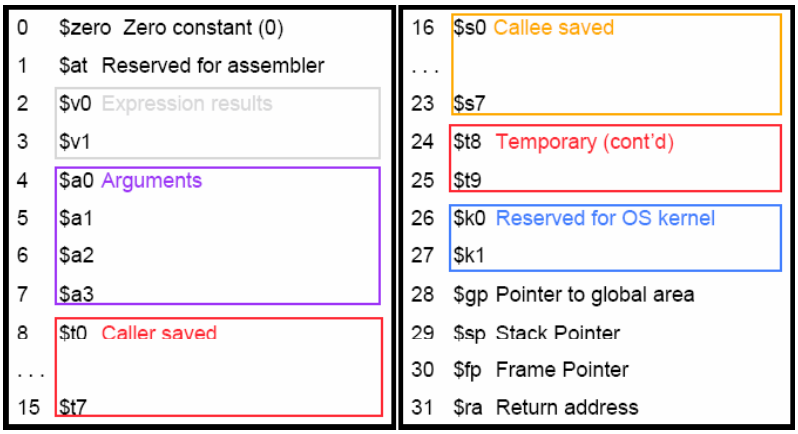
- R type instruction for arithmetic and logical operations. The fields are op, rs, rt, rd, shamt, funct
- I type instruction for operations involving immediates up to 16 bits wide. Load, store and branches are formatted as I type.
- J type instructions for jumping.
- Consult the text book's green card or pages 77 and 78 for a comprehensive listing of MIPS instructions.
- Important correction to column 1 of green card:
  Shift Left Logical `sll` R R[rd]=R[rt]<<shamt
  Shift Right Logical `srl` R R[rd]=R[rt]>>shamt

5. PA1 Thoughts

- Do the job of a compiler.
- Do not use tail recursion.
- Do use the Partition keyword.
- Know the procedure call conventions well.
- Use display command, xspim or pcspim to debug your code.

6. MIPS procedure call convention

Call convention summary (please follow this guideline when you do MIPS assembly coding for homeworks, PAs and in exams):



Example: a recursive program that evaluates factorial 10.

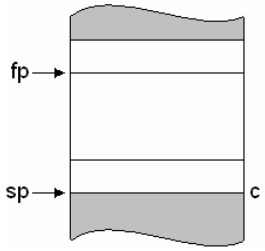
```
main ()
{
   printf ("The factorial of 10 is %d\n", fact (10));
}

int fact (int n)
{
   if (n < 1)
      return (1);
   else
      return (n * fact (n - 1));
}
```

```
.text                       # Indicate beginning of program
.globl main                 # main can be referenced from other files
main:
     subu      $sp, $sp,32    # Stack frame is 32 bytes long
     sw        $ra, 20($sp)   # Save return address
     sw        $fp, 16($sp)   # Save old frame pointer
     addiu     $fp, $sp,28    # Set up frame pointer
     li        $a0, 10        # Put argument (10) in $a0
     jal       fact           # Call factorial function
     la        $a0, $LC       # Load string address in $a0
     move      $a1, $v0       # Move fact result to $a1
     jal       printf         # Call the print function
     lw        $ra, 20($sp)   # Restore return address
     lw        $fp, 16($sp)   # Restore frame pointer
     addiu     $sp, $sp, 32   # Pop stack frame
     jr        $ra            # Return to caller

     .rdata                 # Read-only data
$LC:
     .ascii "The factorial of 10 is %d\n\000"
                            # Store the string in memory, but do not null-terminate it

.text
fact:
     subu      $sp, $sp, 32   # Stack frame is 32 bytes long
     sw        $ra, 20($sp)   # Save return address
     sw        $fp, 16($sp)   # Save frame pointer
     addiu     $fp, $sp, 28   # Set up frame pointer
     sw        $a0, 0($fp)    # Save argument (n)
     lw        $v0, 0($fp)    # Load n
     bgtz      $v0, iterate   # Branch if n > 0
     li        $v0, 1         # Return 1
     j         pop            # Jump to code to return
```

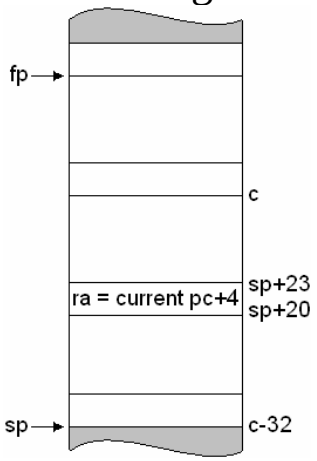


```
iterate:
     lw        $v1, 0($fp)    # Load n
     subu      $v0, $v1, 1    # Compute n - 1
     move      $a0, $v0       # Move value to $a0
     jal       fact           # Call factorial function

     lw        $v1, 0($fp)    # Load n
     mul       $v0, $v0,$v1   # Compute fact(n-1) * n

pop:                        # Result is in $v0
     lw        $ra, 20($sp)   # Restore $ra
     lw        $fp, 16($sp)   # Restore $fp
     addiu     $sp, $sp, 32   # Pop stack
     jr        $ra            # Return to caller
```

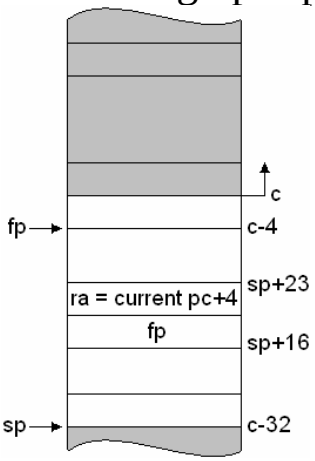
Originally



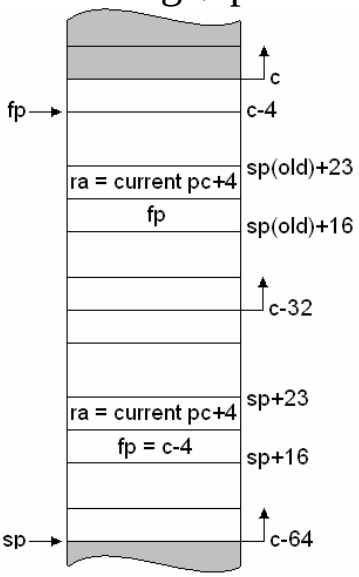
After storing $ra



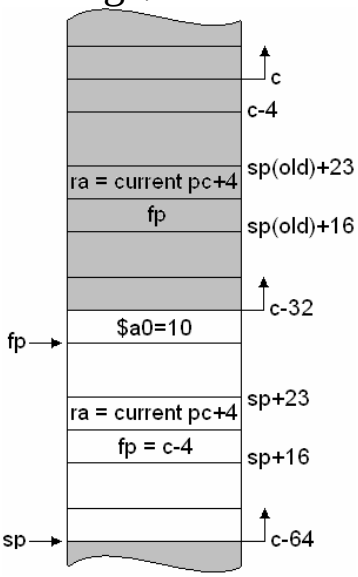
After setting up $fp



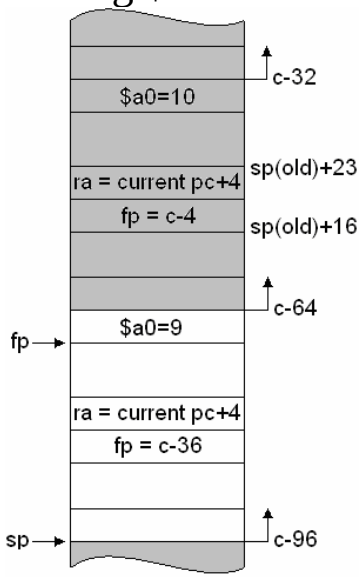
After saving $fp in fact



After saving $a0 in iteration 1



After saving $a0 in iteration 2



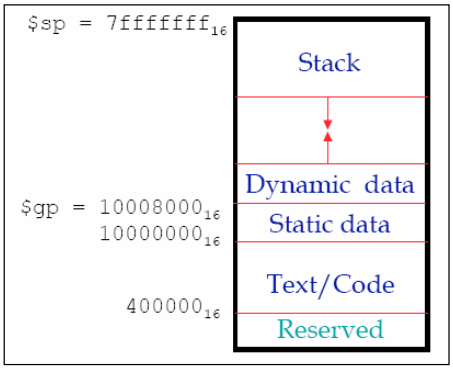
Aside 1:

The `.text` and `.globl` you see in the assembly code are assembler directives, which are there to tell the assembler how to translate a program but which do not translate to machine instructions. Names followed by a colon, such as `str` or `main`, are labels that name the next memory location.

Specifically data definitions start with the `.data` directive, code definition starts with `.text` directive – usually an assembly program would have a `main` as well as a number of subroutine definitions.

Example:

```
     .data          # data memory
foo: .word 0        # 32 bit variable

     .text          # program memory
     .align 2       # word alignment
     .globl main    # main is global

main:
     lw  $a0,  foo
```



Aside 2:
SPIM provides the "syscall" instruction to communicate with the system and do simple I/O. The specific I/O done depends on the value of some registers.

Example:

```
str:
   .asciiz "jon"  # Declare zero-delimited ascii string
   li $v0, 4      # load 4 in $v0 for print_string
                  # syscall
   la $a0, str    # load our string in the syscall
                  # argument register
   syscall        # make the call
```

In this case the system call code is 4 or print_string. Other system calls include:

Code 1: print_int with argument $a0 = integer
Code 2: print_float with argument $f12 = float
Code 3: print_double with argument $f12 = double
Code 4: print_string with argument $a0 = string
Code 5: read_int with integer result in $v0
Code 6: read_float with float result in $f0
:
:

The complete listing is available in Appendix A of the textbook (on CD).