Week 05

MIPS Instruction Set

1/46

The MIPS processor implements a base set of instructions, e.g.

- `lw`, `sw`, `add`, `sub`, `and`, `or`, `sll`, `beq`, ...

Augmented by a set of pseudo-instructions, e.g.

- `move`, `rem`, `la`, `li`, `blt`, ...

Each pseudo-instruction maps to one or more base instructions, e.g.

```
Pseudo-instruction       Base instruction(s)

li $t0, Const            ori  $t0, $0, Const

la $r0, Addr             lui  $t0, Addr[31:16]
                         ori  $r0, $t0, Addr[15:0]

move $r0, $r1            addu $r0, $0, $r1
```

... MIPS Instruction Set

2/46

In describing instructions:

Syntax	Semantics
`$Reg`	as source, the content of the register, `reg[Reg]`
`$Reg`	as destination, value is stored in register, `reg[Reg] = value`
`Label`	references the associated address (in C terms, `&Label`)
`Addr`	any expression that yields an address (e.g. `Label($Reg)`)
`Addr`	as source, the content of memory cell `memory[Addr]`
`Addr`	as destination, value is stored in `memory[Addr] = value`

Effectively ...

- treat registers as `unsigned int reg[32]`
- treat memory as `unsigned char mem[`2^{32}`]`

... MIPS Instruction Set

3/46

Examples of data movement instructions:

```
la   $t1,label    # reg[t1] = &label
lw   $t1,label    # reg[t1] = memory[&label]
sw   $t3,label    # memory[&label] = reg[t3]
                  # &label must be 4-byte aligned
lb   $t2,label    # reg[t2] = memory[&label]
sb   $t4,label    # memory[&label] = reg[t4]
move $t2,$t3      # reg[t2] = reg[t3]
lui  $t2,const    # reg[t2][31:16] = const
```

Examples of bit manipulation instructions:

```
and  $t0,$t1,$t2  # reg[t0] = reg[t1] & reg[t2]
and  $t0,$t1,Imm  # reg[t0] = reg[t1] & Imm[t2]
                  # Imm is a constant (immediate)
or   $t0,$t1,$t2  # reg[t0] = reg[t1] | reg[t2]
```

```
xor   $t0,$t1,$t2  # reg[t0] = reg[t1] ^ reg[t2]
neg   $t0,$t1      # reg[t0] = ~ reg[t1]
```

... MIPS Instruction Set

4/46

Examples of arithmetic instructions:

```
add  $t0,$t1,$t2  # reg[t0] = reg[t1] + reg[t2]
                  #   add as signed (2's complement) ints
sub  $t2,$t3,$t4  # reg[t2] = reg[t3] + reg[t4]
addi $t2,$t3, 5   # reg[t2] = reg[t3] + 5
                  #   "add immediate" (no sub immediate)
addu $t1,$t6,$t7  # reg[t1] = reg[t6] + reg[t7]
                  #   add as unsigned integers
subu $t1,$t6,$t7  # reg[t1] = reg[t6] + reg[t7]
                  #   subtract as unsigned integers
mult $t3,$t4      # (Hi,Lo) = reg[t3] * reg[t4]
                  #   store 64-bit result in registers Hi,Lo
div  $t5,$t6      # Lo = reg[t5] / reg[t6] (integer quotient)
                  # Hi = reg[t5] % reg[t6] (remainder)
mfhi $t0          # reg[t0] = reg[Hi]
mflo $t1          # reg[t1] = reg[Lo]
                  # used to get result of MULT or DIV
```

... MIPS Instruction Set

5/46

Examples of testing and branching instructions:

```
seq  $t7,$t1,$t2   # reg[t7] = 1 if (reg[t1]==reg[t2])
                   # reg[t7] = 0 otherwise (signed)
slt  $t7,$t1,$t2   # reg[t7] = 1 if (reg[t1] < reg[t2])
                   # reg[t7] = 0 otherwise (signed)
slti $t7,$t1,Imm   # reg[t7] = 1 if (reg[t1] < Imm)
                   # reg[t7] = 0 otherwise (signed)

j    label         # PC = &label
jr   $t4           # PC = reg[t4]
beq  $t1,$t2,label # PC = &label if (reg[t1] == reg[t2])
bne  $t1,$t2,label # PC = &label if (reg[t1] != reg[t2])
bgt  $t1,$t2,label # PC = &label if (reg[t1] > reg[t2])
bltz $t2,label     # PC = &label if (reg[t2] < 0)
bnez $t3,label     # PC = &label if (reg[t3] != 0)
```

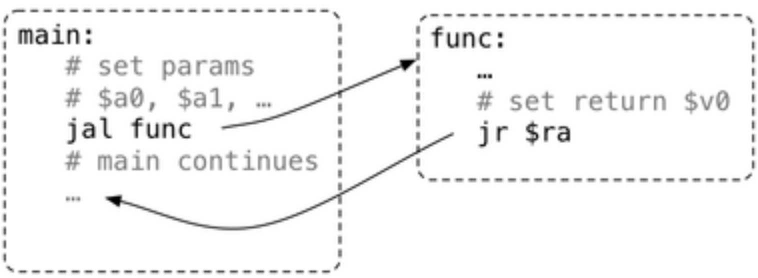
After each branch instruction, execution continues at new PC location

... MIPS Instruction Set

6/46

Special jump instruction for invoking subroutines

```
jal  label         # make a subroutine call
                   # save PC in $ra, set PC to &label
                   # use $a0,$a1 as params, $v0 as return
```



... MIPS Instruction Set

7/46

SPIM interacts with stdin/stdout via `syscalls`

Service	Code	Arguments	Result
print_int	1	`$a0` = integer	
print_float	2	`$f12` = float	
print_double	3	`$f12` = double	
print_string	4	`$a0` = char *	
read_int	5		integer in `$v0`
read_float	6		float in `$f0`
read_double	7		double in `$f0`
read_string	8	`$a0` = buffer, `$a1` = length	string in buffer (including "\n\0")

... MIPS Instruction Set

Directives (instructions to assembler, not MIPS instructions)

```
    .text        # following instructions placed in text
    .data        # following objects placed in data

    .globl       # make symbol available globally

a:  .space 18    # uchar a[18];  or  uint a[4];
    .align 2     # align next object on 2^2-byte addr

i:  .word 2      # unsigned int i = 2;
v:  .word 1,3,5  # unsigned int v[3] = {1,3,5};
h:  .half 2,4,6  # unsigned short h[3] = {2,4,6};
b:  .byte 1,2,3  # unsigned char b[3] = {1,2,3};
f:  .float 3.14  # float f = 3.14;

s:  .asciiz "abc"
                 # char s[4] {'a','b','c','\0'};
t:  .ascii "abc"
                 # char s[3] {'a','b','c'};
```

MIPS Programming

Writing directly in MIPS assembler is difficult:

- develop the solution in C, using
  - registers and `.data` objects as global vars
- translate each C statement to several MIPS instructions

Example:

```
int x = 5;      x:  .word 5
int y = 3;      y:  .word 3
int z;          z:  .space 4
                    ...
                    lw  $t1, x
                    lw  $t2, y
z = x + y;          add $t0, $t1, $t2
                    sw  $t0, z
```

... MIPS Programming

Beware: registers are shared by all parts of the code.

One function can overwrite value set by another function

```
int x;   // first global variable
int y;   // second global variable

int main(void)            int f(int n)
{                         {
   x = 5;                    y = 1;
   y = f(x);                 for (x = 1; x <= n; x++)
   printf("...",x,y);           y = y * x;
   return 0;                 return y;
}                         }
```

After the function, `x == 6` and `y == 120`

It is sheer coincidence that `y` has the correct value.

... MIPS Programming

11/46

Need to be careful managing registers

- follow the conventions implied by register names
- preserve values that need to be saved across function calls

Within a function

- you manage register usage as you like
- typically making use of `$t?` registers

When making a function call

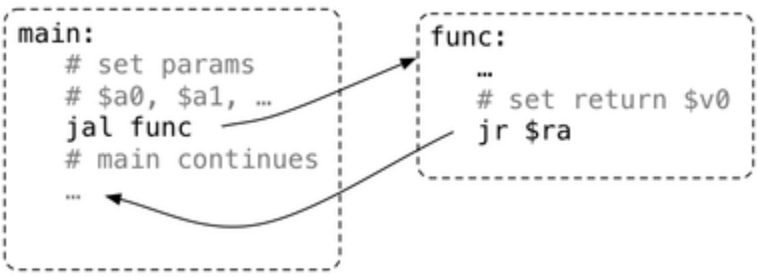
- you transfer control to a separate piece of code
- which may change the value of any non-preserved register
- `$s?` registers must be preserved by function
- `$a?`, `$v?`, `$t?` registers may be modified by function

Function/Subroutine Calls

12/46

Simple function-call protocol:

- load argument values into `$a0`, `$a1`, ...
- invoke `jal`: loads PC into `$ra`, jumps to function
- function puts return values in `$v0`, `$v1`
- returns to caller using `jr $ra`



... Function/Subroutine Calls

13/46

More detail on the function call protocol (assume function `f()`):

- on entry, save the value of `$ra`
- on entry, save the value of any `$s?` registers modified by `f()`
- use the values in `$a?` as input parameters (e.g. `f(2,5)`)
- ... perform the required computation ...
- set the values of `$v0` and `$v1` as returned values
- on exit, restore the saved values of `$s?` registers
- on exit, restore the value of `$ra`

Note that register `$ra` is overwritten by any function call `f()` makes.

... Function/Subroutine Calls

14/46

Example simple function call protocol:

```
# set up arguments
  li $a0, 2
  li $a1, 3
# set $ra and jump
  jal func
# return here
  ...


  .data
fun_ret:
  .space 4
fun_safe:
  .space 32
```

```
func:
   # save return address
   sw  $ra, fun_ret
   # save values of $s0..$s7
   sw  $s0, fun_safe+0
   sw  $s1, fun_safe+4
   ...
   # perform function code
   # might involve calling
   #       other functions
   # leave result in $v0
   # restore $s0..$s7
   lw  $s0, fun_safe+0
   ...
   # restore return address
   lw  $ra, fun_ret
   jr  $ra
```

But even this is not adequate e.g. recursive functions

Exercise 1: Subroutine to Print a number

15/46

In the `addr.s` example ...

- printing results was tedious and repetitive

Encapsulate the instructions in a function behaving like:

```
void print(int n)
{
    printf("%d", n);
    printf("\n");
}
```

and use this to simplify the code in `addr.s`

Control Structures

16/46

C provides expression evaluation and assignment, e.g.

- `x = (1 + y*y) / 2;    z = 1.0 / 2;` ...

MIPS provides register-register operations, e.g.

- `move` R_d,R_s, `li` R_d,Const, `add`, `div`, `and`, ...

C provides a range of control structures

- sequence (`;`), `if`, `while`, `for`, `break`, `continue`, ...

MIPS provides testing/branching instructions

- `seq`, `slti`, `sltu`, ..., `beq`, `bgtz`, `bgezal`, ..., `j`, `jr`, `jal`, ...

We need to render C's structures in terms of testing/branching

Sequence is easy S_1 ; S_2 → mips(S_1) mips(S_2)

... Control Structures

17/46

Simple example of assignment and sequence:

```
int x;          x: .space 4
int y;          y: .space 4

x = 2;             li   $t0, 2
                   sw   $t0, x

y = x;             lw   $t0, x
                   sw   $t0, y

y = x+3;           lw   $t0, x
                   addi $t0, 3
                   sw   $t0, y
```

... Control Structures

Expression evaluation involves

- describing the process as a sequence of binary operations
- managing data flow between the operations

Example:

```
# x = (1 + y*y) / 2
# assume x and y exist as labels in .data
  lw   $t0, y            # t0 = y
  mul  $t0, $t0, $t0     # t0 = t0*t0
  addi $t0, $t0, 1       # t0 = t0+1
  li   $t1, 2            # t1 = 2
  div  $t0, $t1          # Lo = t0/t1 (int div)
  mflo $t0               # t0 = Lo
  sw   $t0, x            # x = t0
```

It is useful to minimise the number of registered involved in the evaluation

---

Conditional Statements

Conditional statements (e.g. `if`)

```
                              if_stat:
if (Cond)                       t0 = evaluate Cond
   { Statements1 }              beqz $t0, else_part
else                            execute Statements1
   { Statements2 }              j    end_if
                              else_part:
                                execute Statements2
                              end_if:
```

... Conditional Statements

Example of if-then-else:

```
int x;          x: .space 4
int y;          y: .space 4
char z;         z: .space 1

x = getInt();      li   $v0, 5
                   syscall
                   move $t0, $v0

y = getInt();      li   $v0, 5
                   syscall
                   move $t1, $v0

if (x == y)        bne  $t0, $t1, printN
   z = 'Y';      printY:
                   li   $a0, 'Y'
                   j    print
```

```
else                printN:
   z = 'N';           li    $a0, 'N'
                      j     print     # redunant
                    print:
putChar(z);           li    $v0, 11
                      syscall
```

Exercise 2: Mapping `if`

21/46

Translate the following C statement to MIPS

```
if (mark < 50)
   grade = 'F';      // i.e. FL
else if (mark < 65)
   grade = 'P';      // i.e. PS
else if (mark < 75)
   grade = 'C';      // i.e. CR
else if (mark < 85)
   grade = 'D';      // i.e. DN
else
   grade = 'H';      // i.e. HD
```

Assume that `mark` and `grade` are defined in `.data`

... Conditional Statements

22/46

Could make `switch` by first converting to `if`

```
switch (Expr) {              tmp = Expr;
case Val1:                   if (tmp == Val1)
   Statements1 ; break;         { Statements1; }
case Val2:                   else if (tmp == Val2
case Val3:                            || tmp == Val3
case Val4:                            || tmp == Val4)
   Statements2 ; break;         { Statements2; }
case Val5:                   else if (tmp == Val5)
   Statements3 ; break;         { Statements3; }
default:                     else
   Statements4 ; break;         { Statements4; }
}
```

... Conditional Statements

23/46

Jump table: an alternative implementation of `switch`

- works best for small, dense range of case values (e.g. 1..10)

```
                           jump_tab:
                              .word c1, c2, c2, c2, c3
                           switch:
                                t0 = evaluate Expr
switch (Expr) {                 if (t0 < 1 || t0 > 5)
case 1:                            jump to default
   Statements1 ; break;         dest = jump_tab[(t0-1)*4]
case 2:                         jump to dest
case 3:                    c1:  execute Statements1
case 4:                         jump to end_switch
   Statements2 ; break;    c2:  execute Statements2
case 5:                         jump to end_switch
   Statements3 ; break;    c3:  execute Statements3
default:                        jump to end_switch
   Statements4 ; break;    default:
}                               execute Statements4
                           end_switch:
```

Boolean Expressions

Boolean expressions in C are short circuit

```
(Cond1 && Cond2 && ... && Condn)
```

Evaluates by

- evaluate $\texttt{Cond}_1$; if 0 then return 0 for whole expression
- evaluate $\texttt{Cond}_2$; if 0 then return 0 for whole expression
- ...
- evaluate $\texttt{Cond}_n$; if 0 then return 0 for whole expression
- otherwise, return 1

In C, any non-zero value is treated as true; MIPS tends to use 1 for true

C99 standard defines return value for booleans expressions as 0 or 1

... Boolean Expressions

Similarly for disjunctions

```
(Cond1 || Cond2 || ... || Condn)
```

Evaluates by

- evaluate $\texttt{Cond}_1$; if !0 then return 1 for whole expression
- evaluate $\texttt{Cond}_2$; if !0 then return 1 for whole expression
- ...
- evaluate $\texttt{Cond}_n$; if !0 then return 1 for whole expression
- otherwise, return 1

In C, any non-zero value is treated as true; MIPS tends to use 1 for true

C99 standard defines return value for booleans expressions as 0 or 1

Exercise 3: Implementing Conjunctions

Implement the following in MIPS assembler

```
if (x != 0 && y != 0 && x > y)
   { statements1; }
else
   { statements2; }
```

Assume that `x` and `y` are labels defined in `.data`

Iteration Statements

Iteration (e.g. `while`)

```
                           top_while:
while (Cond) {                t0 = evaluate Cond
   Statements;                beqz $t0,end_while
}                             execute Statements
                              j    top_while
                           end_while:
```

Treat `for` as a special case of `while`

```
                                     i = 0
for (i = 0; i < N; i++) {            while (i < N) {
    Statements;                         Statements;
}                                       i++;
                                     }
```

Exercise 4: Mapping `while`

28/46

Implement the following in MIPS assembler

```
i = 1;
while (i < 20) {
   sum = sum + i;
   i++;
}
```

Assume that `i` and `sum` are defined in `.data`

... Iteration Statements

29/46

Example of iteration over an array:

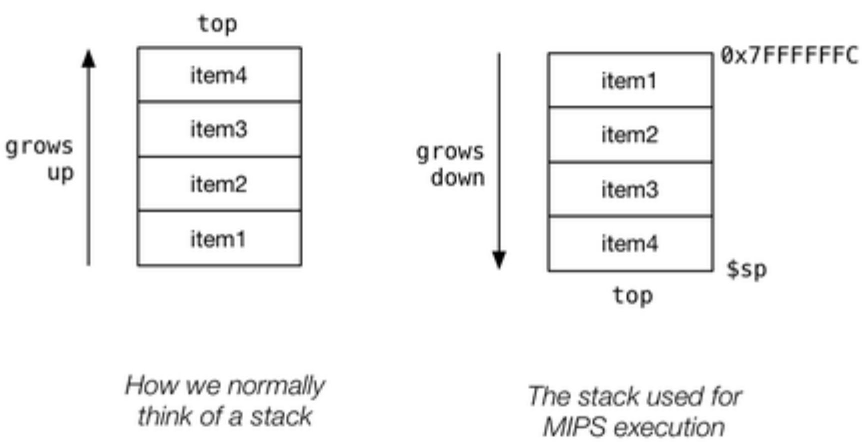
```
int sum, i;                 sum: .word 4        # use reg for i
int a[5] = {1,3,5,7,9};     a:   .word 1,3,5,7,9
...                              ...
sum = 0;                         li   $t0, 0    # i = 0
                                 li   $t1, 0    # sum = 0
                                 li   $t2, 4    # max index
for (i = 0; i < N; i++)     for: bgt  $t0, $t2, end_for
                                 move $t3, $t0
                                 mul  $t3, $t3, 4
   sum += a[i];                  add  $t1, $t1, a($t3)
printf("%d",sum);                addi $t0, $t0, 1    # i++
                                 j    for
                        end_for: sw   $t1, sum
                                 move $a0, $t1
                                 li   $v0, 1
                                 syscall        # printf
```

Function/Subroutine Calls

30/46

The function call protocol we showed earlier was a simple special case.

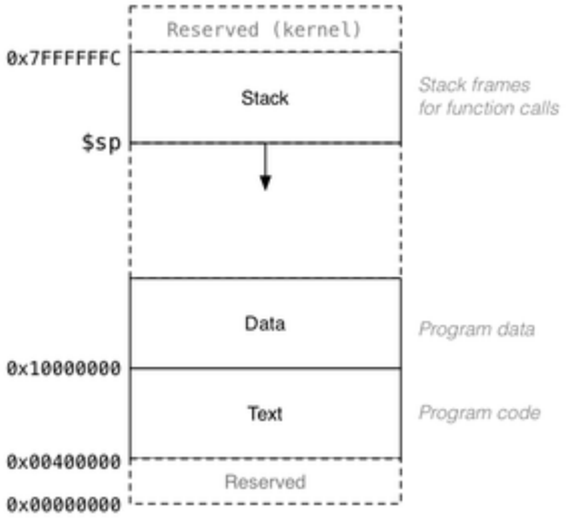
The general case is handled using the MIPS stack.



... Function/Subroutine Calls

31/46

Reminder: MIPS memory usage



... Function/Subroutine Calls

32/46

Reminder: register usage conventions when `f()` calls `g()`:

- caller saved registers (saved by `f()`)
  - `f()` tells `g()` "If there is anything I want to preserve in these registers, I have already saved it before calling you"
  - `g()` tells `f()` "Don't assume that these registers will be unchanged when I return to you"
  - e.g. `$t0` .. `$t9`, `$a0` .. `$a3`, `$ra`
- callee saved registers (saved by `g()`)
  - `f()` tells `g()` "I assume the values of these registers will be unchanged when you return"
  - `g()` tells `f()` "If I need to use these registers, I will save them first and restore them before returning"
  - e.g. `$s0` .. `$s7`, `$sp`, `$fp`

... Function/Subroutine Calls

33/46

Each function allocates a small section of the stack (a *frame*)

- used for: saved registers, local variables, parameters to callees
- created in the function *prologue* (pushed)
- removed in the function *epilogue* (popped)

Why we use a stack:

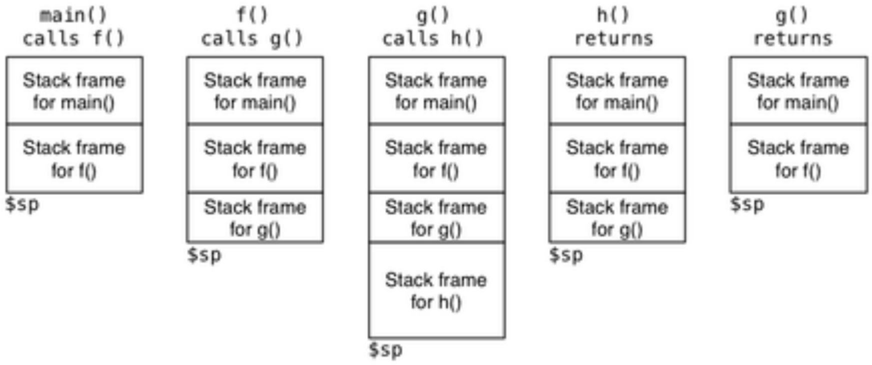
- function `f()` calls `g()` which calls `h()`
- `h()` runs, then finishes and returns to `g()`
- `g()` continues, then finishes and returns to `f()`

i.e. last-called, first-exits (last-in, first-out) behaviour

... Function/Subroutine Calls

34/46

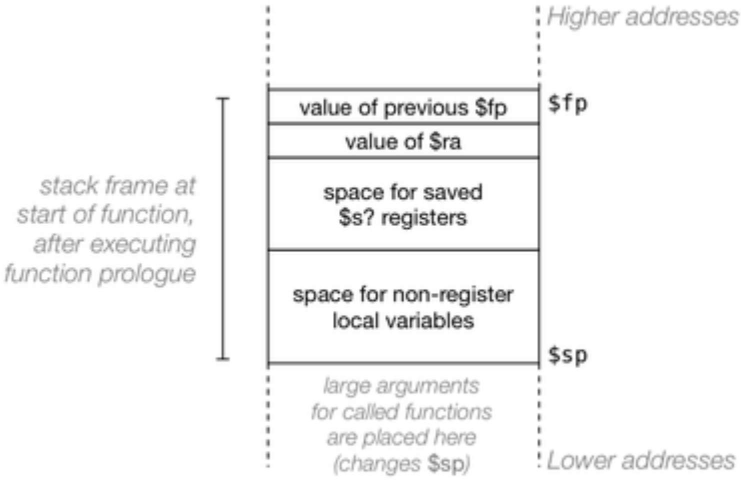
How stack changes as functions are called and return:



... Function/Subroutine Calls

35/46

Contents of a typical stack frame:



Aside: MIPS Branch Delay Slots

36/46

The real MIPS architecture is "pipelined" to improve efficiency

- one instruction can start before the previous one finishes

For branching instructions (e.g. `jal`) ...

- instruction following branch is executed before branch completes

To avoid potential problems use **`nop`** immediately after branch

A problem scenario, and its solution (branch delay slot):

```
# Implementation of print(compute(42))
li   $a0, 42                li   $a0, 42
jal  compute                jal  compute
move $a0, $v0               nop
jal  print                  move $a0,$v0
                            jal  print
```

Since SPIM is not pipelined, the `nop` is not required

Function Calling Protocol

37/46

Before one function calls another, it needs to

- place 64-bit double args in `$f12` and `$f14`
- place 32-bit arguments in the `$a0..$a3`
- if more than 4 args, or args larger than 32-bits ...

   - push value of all such args onto stack
- save any non-`$s?` registers that need to be preserved
   - push value of all such registers onto stack
- `jal` address of function (usually given by a label)

Pushing onto stack from `$t0` means:

```
addi $sp, $sp, -4
sw   $t0, ($sp)
```

... Function Calling Protocol

38/46

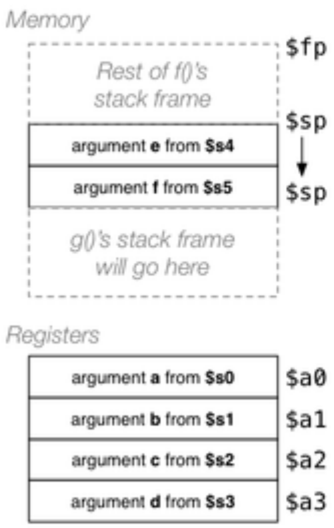
Example: function `f()` calls function `g(a,b,c,d,e,f)`

```
int f(...)
{
   // variables happen to be stored
   // in registers $s0, $s1, ..., $s5
   int a,b,c,d,e,f;
   ...
   a = g(a,b,c,d,e,f);
   ...
}
int g(int u,v,w,x,y,z)
{
   return u+v+w*w*x*y*z;
}
```

... Function Calling Protocol

39/46

MIPS version of function call:




```
# … rest of f()'s code
# a = g(a,b,c,d,e,f)
# …
# set up args
move $a0, $s0
move $a1, $s1
move $a2, $s2
move $a3, $s3
sw   $s4, -4($sp)
sw   $s5, -8($sp)
addi $sp, $sp, -8
# call g()
jal  g
# place return value in $s0
move $s0, $v0
```

Exercise 5: Simple Function call

40/46

Write MIPS code to implement the function call in ...

```
char a[100];

int main(void)
{
   fgets(a, 99, stdin);
   printf("%d\n", mylength(a,99));
   return 0;
}
int mylength(char *s, int n)
{
   int i = 0;
```

```
    int *end = &s[n];
    while (s < end && *s != '\0')
       { s++; i++; }
    return i;
}
```

Structure of Functions

41/46

Functions in MIPS have the following general structure:

```
# start of function
FuncName:
   # function prologue
   # sets up stack frame
   # saves relevant registers
   ...
   # function body
   # performs computation
   # leaving result in $v0
   ...
   # function epilogue
   # restores registers
   # cleans up stack frame
   jr  $ra
```

Function Prologue

42/46

Before a function starts working, it needs to ...

- create a stack frame for itself (change `$fp` and `$sp`)
- save the return address in the stack frame
- save any `$s?` registers that it plans to change

We can determine the initial size of the stack frame via

- 4 bytes for saved `$fp` + 4 bytes for saved `$ra`
- + 4 bytes for each saved `$s?`

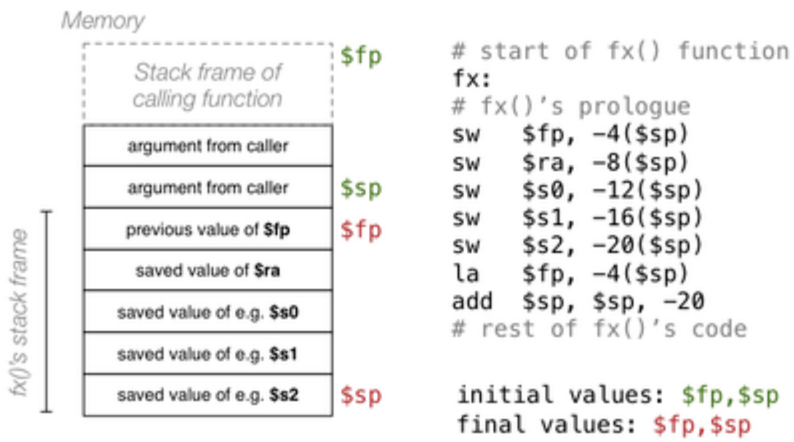
Changing `$fp` and `$sp` ...

- new `$fp` = old `$sp` - 4
- new `$sp` = old `$sp` - size of frame (in bytes)

... Function Prologue

43/46

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`



Function Epilogue

44/46

Before a function returns, it needs to ...

- place the return value in `$v0` (and maybe `$v1`)
- pop any pushed arguments off the stack
- restore the values of any saved `$s?` registers
- restore the saved value of `$ra` (return address)
- remove its stack frame (change `$fp` and `$sp`)
- return to the calling function (`jr $ra`)

Locations of saved values computed relative to `$fp`

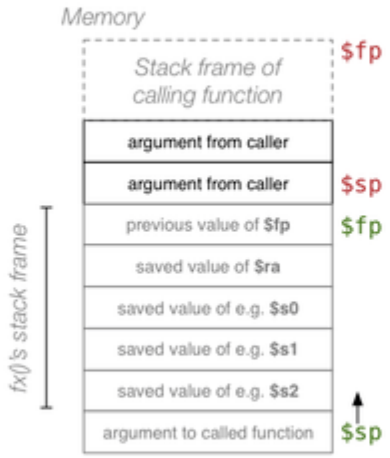
Changing `$fp` and `$sp` ...

- new `$sp` = old `$fp` + 4
- new `$fp` = `memory[`old `$fp]`

... Function Epilogue

45/46

Example of function `fx()`, which uses `$s0`, `$s1`, `$s2`




```
# end of fx() function
lw   $v0, return value
# fx()'s epilogue
lw   $s2, -16($fp)
lw   $s1, -12($fp)
lw   $s0, -8($fp)
lw   $ra, -4($fp)
# remove stack frame
la   $sp, 4($fp)
lw   $fp, ($fp)
# return
jr   $ra

initial values: $fp,$sp
final values: $fp,$sp
```

Exercise 6: Function to sum values in array

46/46

Implement a MIPS version of the following:

```
int array[10] = {5,4,7,6,8,9,1,2,3,0};

int main(void)
{
   printf("%d\n", sumOf(array,0,9));
   return 0;
}

int sumOf(int a[], int lo, int hi)
{
   if (lo > hi)
      return 0;
   else
      return a[lo] + sumOf(a,lo+1,hi);
}
```

Produced: 24 Aug 2017