Machine-Level Programming: Procedures
CS 347

Jan. 25, 1996

Topics

- Register Conventions
- Run time stack management
- Supporting recursion
- Floating Point registers & instructions

MIPS Register Conventions

Number	Name	Usage
$0	$0	Constant 0
$1	$at	Reserved Temp.
$2	$v0	Return Values
$3	$v1	
$4	$a0	Procedure arguments
$5	$a1	
$6	$a2	
$7	$a3	
$8	$t0	Caller Save Temporaries: May be overwritten by called procedures
$9	$t1	
$10	$t2	
$11	$t3	
$12	$t4	
$13	$t5	
$14	$t6	
$15	$t7	

Number	Name	Usage
$16	$s0	Callee Save Temporaries: May not be overwritten by called procedures
$17	$s1	
$18	$s2	
$19	$s3	
$20	$s4	
$21	$s5	
$22	$s6	
$23	$s7	
$24	$t8	Caller Save Temp
$25	$t9	
$26	$k0	Reserved for Operating Sys
$27	$k1	
$28	$gp	Global Pointer
$29	$sp	Stack Pointer
$30	$s8	Callee Save Temp
$31	$ra	Return Address

Stack-Based Languages

Languages that support recursion

- E.g.: C, Pascal

Stack Allocated in *Frames*

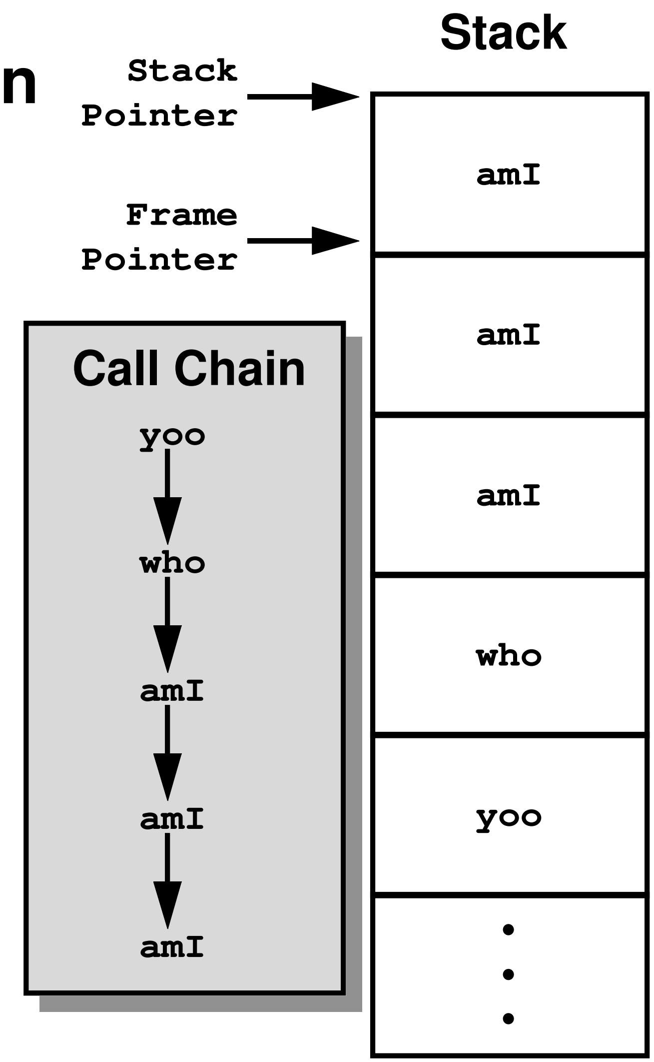
- State for procedure invocation
  – Return point, arguments, locals

Code Example

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    •
    •
    amI();
    •
    •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```



Saving Conventions

When procedure `yoo` calls `who`

- **`yoo` is the *caller***
- **`who` is the *callee***

Caller Save

- **If `yoo` wants value in register before & after `who` returns**
  - Save it on the stack
  - Use callee save register

Callee Save

- **If `who` wants to use register**
  - Save current register value on stack, restore when returning
  - Use caller save register

Saving Examples

Caller Save

- Caller must save/restore
- Callee can clobber

```
yoo:
    li $8, 17
      • • •
    sw $8, 20($sp)
    jal who
    lw $8, 20($sp)
      • • •
    mov $2, $8
    j $31
```

```
who:
    li $8, 57
      • • •
    j $31
```

Callee Save

- Callee must save/restore

```
yoo:
    li $16, 17
      • • •
    jal who
      • • •
    mov $2, $16
    j $31
```

```
who:
    sw $16, 20($sp)
    li $16, 57
      • • •
    lw $16, 20($sp)
    j $31
```

What's wrong with above code?

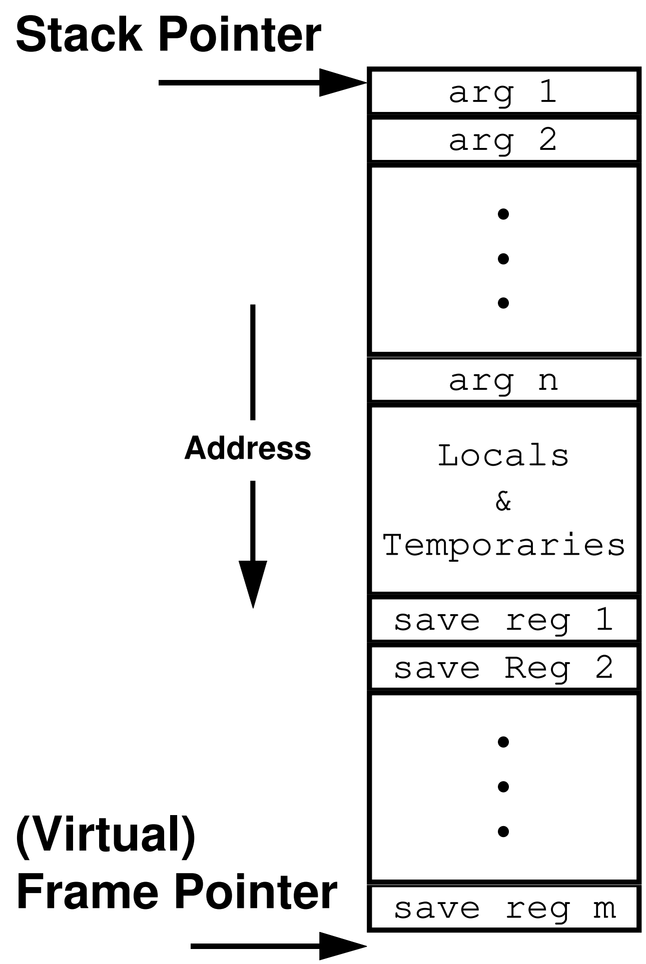
MIPS Stack Frame

Conventions

- **Agreed by all program/compiler writers**
  - Allows linking between different compilers
  - Allows symbolic debugging tools

Run Time Stack

- **Save context**
  - Registers
- **Storage for local variables**
- **Parameters to called functions**
- **Required to support recursion**



Stack Frame Requirements

Procedure Categories

- **Leaf procedures that do not use stack**
  - Do not call other procedures
  - Can fit all temporaries in caller-save registers
- **Leaf procedures that use stack**
  - Do not call other procedures
  - Need stack for temporaries
- **Non-leaf procedures**
  - Must use stack (to save $31 at the least)

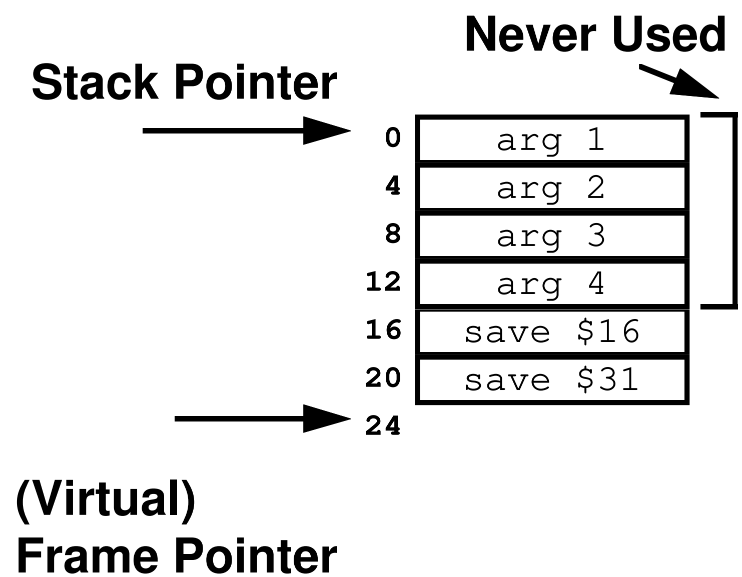
Stack Frame Structure

- **Must be multiple of 8 bytes**
- **Must have space for saving argument registers $4, …, $7**
  - Even if don't store anything there
  - Used by symbolic debuggers

Optimized Stack Frame Example

C Code

```
/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```



Compiled -O: Frame Setup

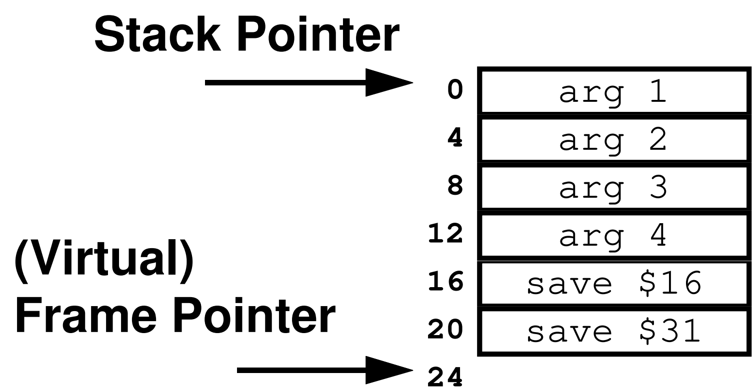
```
rfact:
        .frame   $sp,24,$31
        .mask    0x80010000,-4
        .fmask   0x00000000,0
        subu     $sp,$sp,24
        sw       $31,20($sp)
        sw       $16,16($sp)
```

- **Stack frame 24 bytes**
- **Virtual frame ptr at sp + 24**
- **Save registers $31 and $16**
- **Starting at sp + 24 - 4**
- **No floating pt. regs. used**

Opt. Stack Frame Example (Cont.)

C Code

```
/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```



Compiled -O: Body

```
     move  $16,$4         # Save x
     slt   $2,$16,2       # if x < 2
     BNE   $2,$0,$L3      # then goto $L3
     li    $2,0x00000001 # with return val 1
     JAL   rfact          # Call rfact
     subu  $4,$16,1       # on x-1
     mult  $16,$2         # x * return val
     mflo  $2             # as return val
```

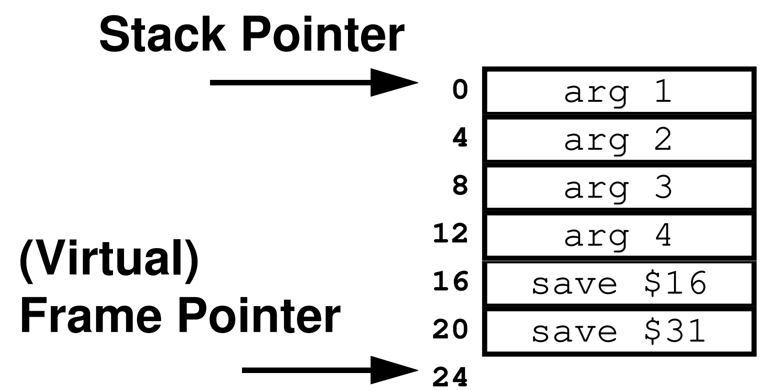
Opt. Stack Frame Example (Cont.)

C Code

```
/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```

Compiled -O: Completion

```
$L3:
    lw   $31,20($sp)
    lw   $16,16($sp)
    addu $sp,$sp,24
    j    $31
```



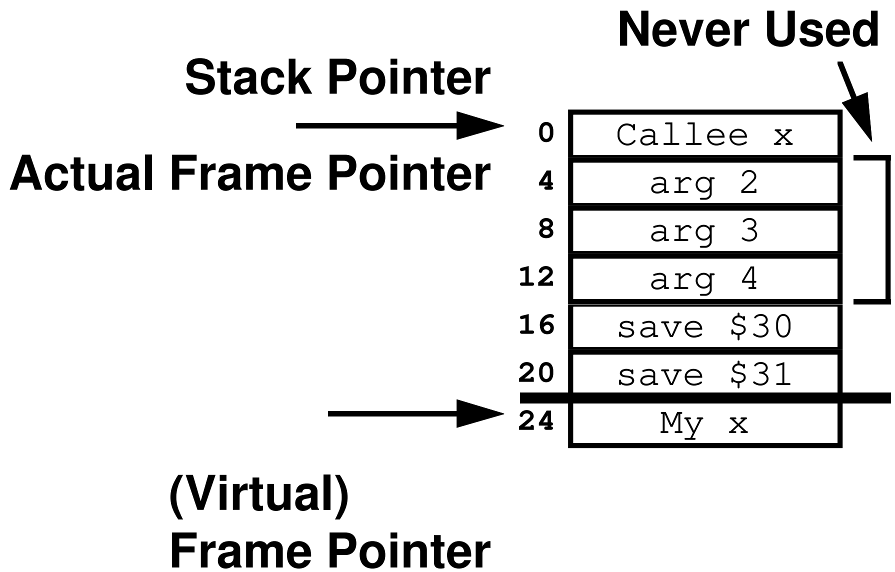
Unopt. Stack Frame Example:

C Code

```
/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```

Compiled: Frame Setup

```
rfact:
    .frame  $fp,24,$31
    .mask   0xc0000000,-4
    .fmask  0x00000000,0
    subu    $sp,$sp,24
    sw      $31,20($sp)
    sw      $fp,16($sp)
    move     $fp,$sp
    sw       $4,24($fp)
```

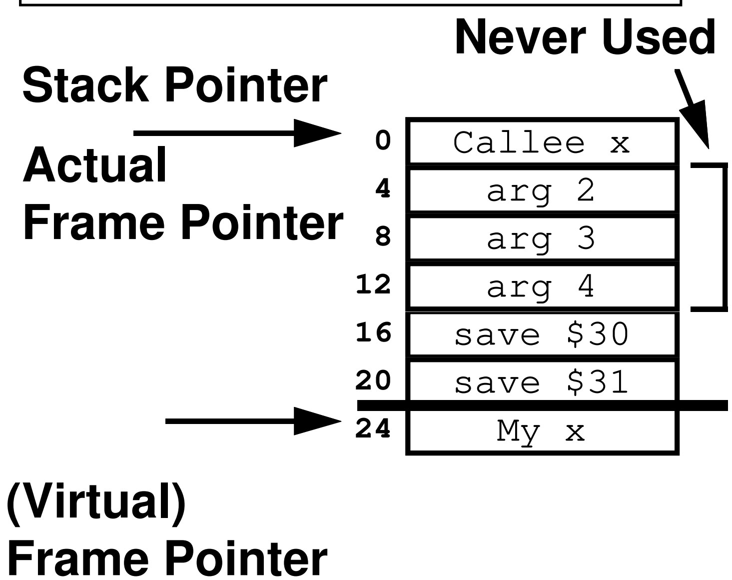


- Stack frame 24 bytes
- Virtual frame ptr at fp + 24
- Save registers $31 and $30 ($fp)
- Starting at fp + 24 - 4
- No floating pt. regs. used

Unopt. Stack Frame Example (Cont.)

C Code

```
/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```



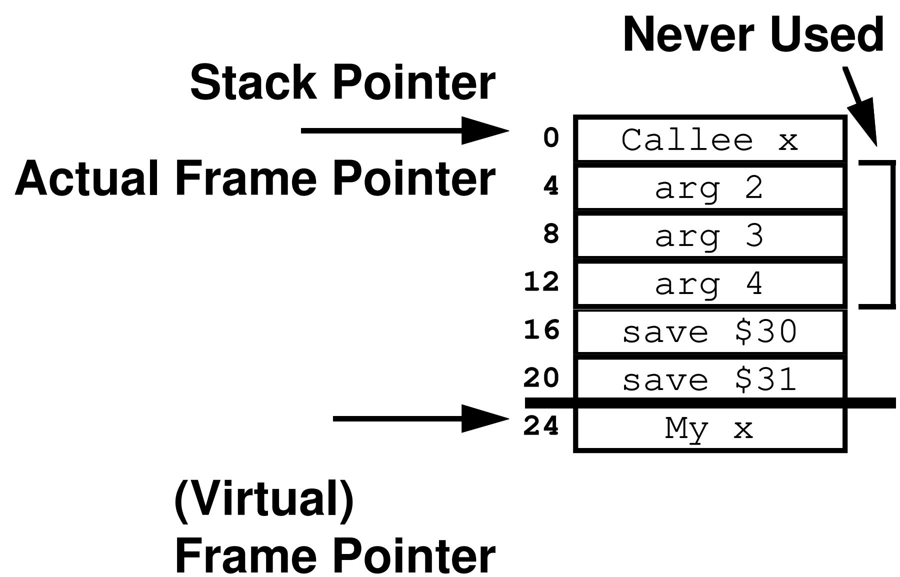
Compiled: Body

```
   lw      $2,24($fp)     # Get argument x
   slt     $3,$2,2        # If !(x < 2)
   beq     $3,$0,$L2      # then goto $L2
   li      $2,0x00000001  # return val 1
   j       $L1            # goto $L1
$L2:
   lw      $3,24($fp)     # Get argument x
   subu    $2,$3,1        # Subtract 1
   move    $4,$2          # assign to arg1
   jal     rfact          # Call rfact
   lw      $3,24($fp)     # Get argument x
   mult    $2,$3          # x * return val
   mflo    $2             # is return val
   j       $L1
```

Unopt. Stack Frame Example (Cont.)

C Code

```
/* Recursive factorial */
int rfact(int x)
{
  if (x <= 1)
    return 1;
  return x * rfact(x-1);
}
```



Compiled: Completion

```
$L1
  move    $sp,$fp # don't trust $sp here
  lw      $31,20($sp)
  lw      $fp,16($sp)
  addu    $sp,$sp,24
  j       $31
```

Local Storage Example

C Code

```
char *stuff_it
  (char c0, char c1, char c2,
   char c3, char c4, char c5,
   char c6)
{
  char buf[7]; /* Not enough! */
  int i = 0;
  char *result;
  buf[i++] = c0;
  buf[i++] = c1;
  buf[i++] = c2;
  buf[i++] = c3;
  buf[i++] = c4;
  buf[i++] = c5;
  buf[i++] = c6;
  buf[i++] = 0;
  result = (char *) malloc(i);
  strcpy(result, buf);
  return result;
}
```

```
char *do_it()
{
  return stuff_it
    ('1','5','-','3','4','7','!');
}
```

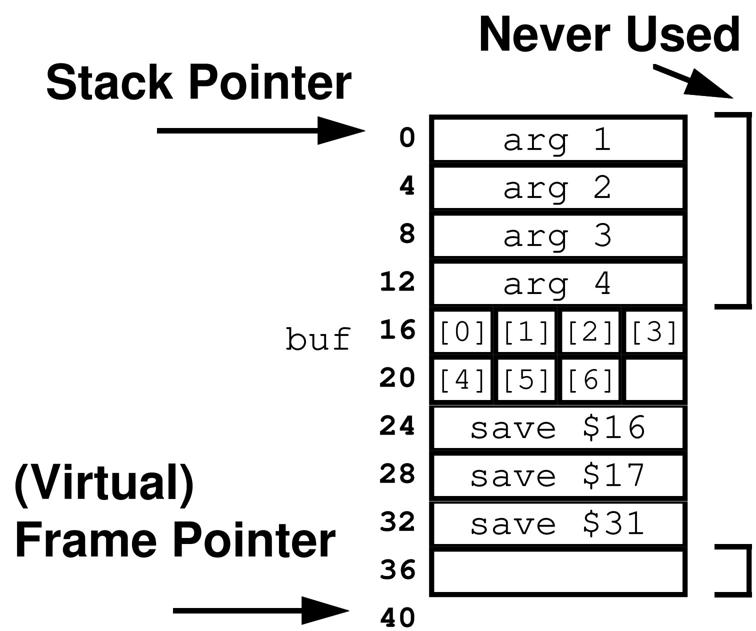
Local Storage Example

C Code: Initial

```
char *stuff_it
  (char c0, char c1, char c2,
   char c3, char c4, char c5,
   char c6)
{
  char buf[7]; /* Not enough! */
  int i = 0;
  char *result;
```

Compiled -O: Frame Setup

```
stuff_it:
  .frame  $sp,40,$31
  .mask   0x80030000,-8
  .fmask  0x00000000,0
  .subu   $sp,$sp,40
  sw      $31,32($sp)
  sw      $17,28($sp)
  sw      $16,24($sp)
```



- Stack frame 40 bytes
- Virtual frame ptr at sp + 40
- Save registers $31, $17, $16
- Starting at sp + 24 - 8
- No floating pt. regs. used

Local Storage Example (Cont.)

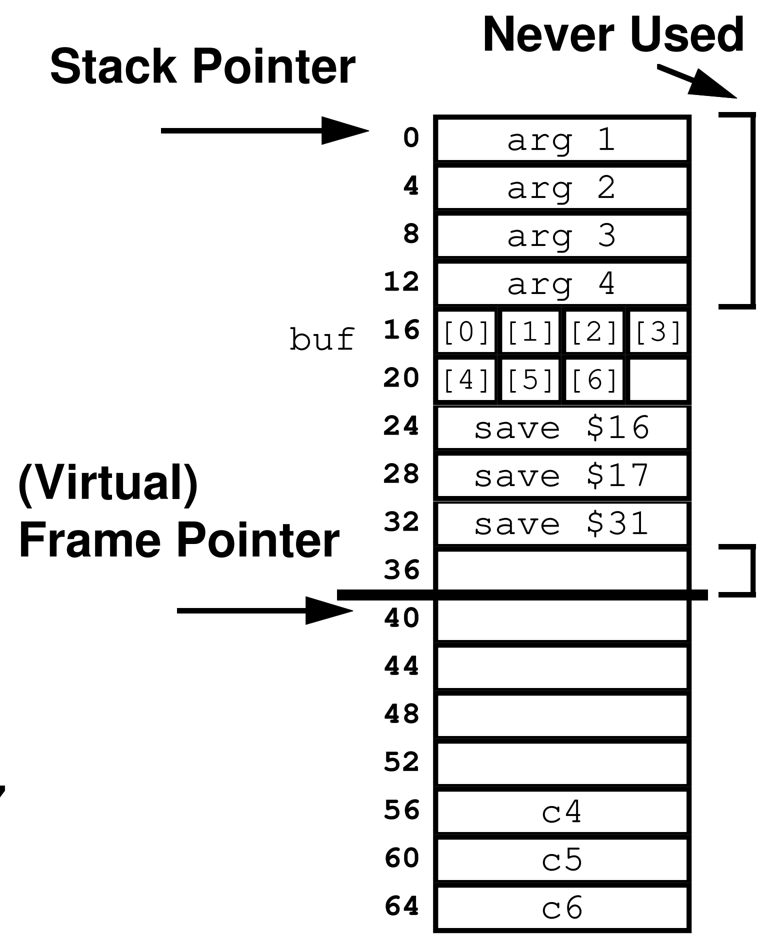
C Code: Argument Retrieval

```
char *stuff_it
  (char c0, char c1, char c2,
   char c3, char c4, char c5,
   char c6)
```

Compiled -O: Args 5–7

```
   lbu     $2,56($sp)
   lbu     $3,60($sp)
   lbu     $8,64($sp)
```

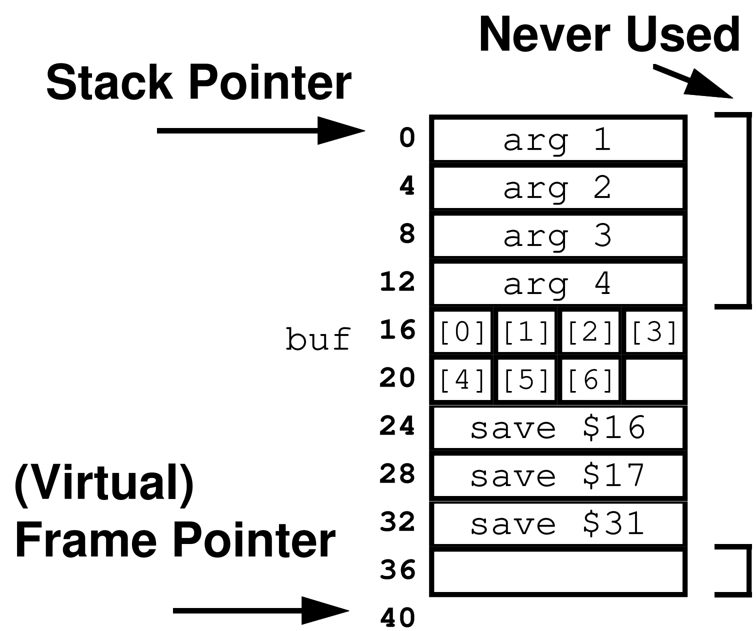
- **`c0` … `c3` passed in registers $4 … $7**



Local Storage Example (Cont.)

C Code: Local Setup

```
buf[i++] = c0;
buf[i++] = c1;
buf[i++] = c2;
buf[i++] = c3;
buf[i++] = c4;
buf[i++] = c5;
buf[i++] = c6;
buf[i++] = 0;
```



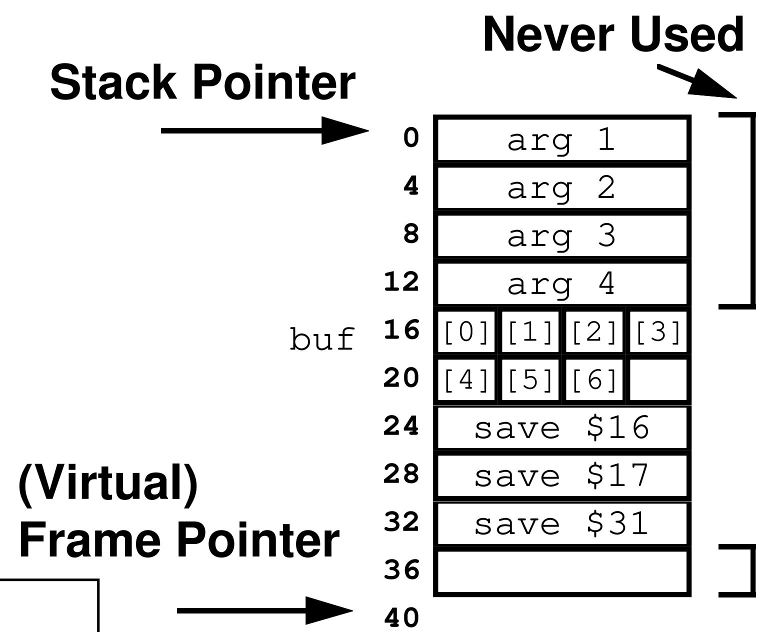
Compiled -O

```
addu    $17,$sp,16   # $17 = &buf
sb      $4,16($sp)   # buf[0] = c0
sb      $5,17($sp)   # buf[1] = c1
sb      $6,18($sp)   # buf[2] = c2
sb      $7,19($sp)   # buf[3] = c3
sb      $0,23($sp)   # buf[7] = 0  !!
sb      $2,20($sp)   # buf[4] = c4
sb      $3,21($sp)   # buf[5] = c5
sb      $8,22($sp)   # buf[6] = c6
```

Local Storage Example (Cont.)

C Code: Calls

```
result = (char *) malloc(i);
strcpy(result, buf);
```



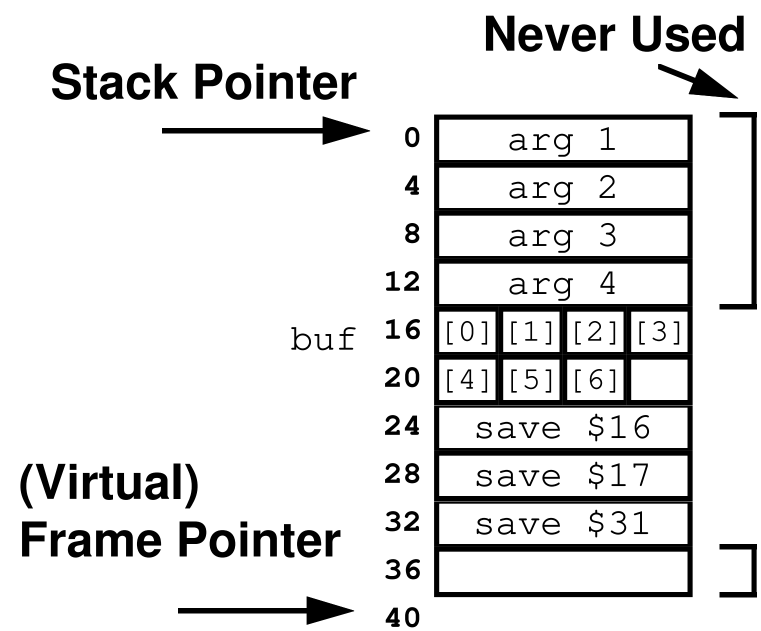
Compiled -O

```
li      $4,0x00000008   # i = 8
jal     malloc          # malloc(i)
move    $16,$2          # $16 = result
move    $4,$16
JAL     strcpy          # strcpy(result,
move    $5,$17          #   , buf)
move    $2,$16          # return val result
```

Local Storage Example (Cont.)

Completion

```
        lw      $31,32($sp)
        lw      $17,28($sp)
        lw      $16,24($sp)
        addu    $sp,$sp,40
        j       $31
```



Floating Point Unit

Implemented as Coprocessor 1

- Hardware to add, multiply, and divide
- Floating point data registers
- Various control & status registers

Floating Point Values

- Single precision (C `float`): 32 bits
- Double precision (C `double`): 64 bits

Floating Point Data Registers

- 32 registers, each 4 bytes
- Allocated in pairs
  - Even for single precision

`$f0`	`$f1`	Return Values
`$f2`	`$f3`	
`$f4`	`$f5`	Caller Save Temporaries: May be overwritten by called procedures
`$f6`	`$f7`	
`$f8`	`$f9`	
`$f10`	`$f11`	
`$f12`	`$f13`	Procedure arguments
`$f14`	`$f15`	
`$f16`	`$f17`	Caller Save Temporaries:
`$f18`	`$f19`	
`$f20`	`$f21`	Callee Save Temporaries: May not be overwritten by called procedures
`$f22`	`$f23`	
`$f24`	`$f25`	
`$f26`	`$f27`	
`$f28`	`$f29`	
`$f30`	`$f31`	

Floating Point Code Example

Compute Inner Product of Two Vectors

- **Single precision**

```
float inner_prodF
  (float x[], float y[], int n)
{
  int i;
  float result = 0.0;
  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
inner_prodF:
  subu    $sp,$sp,8 # Allocate stack frame
  mtc1    $0,$f4    # result = 0.0
  BLEZ    $6,$L3    # if n <= 0 goto $L3
  move    $3,$0     # i = 0
$L5:
  l.s     $f0,0($4) # $f0 = x[i]
  l.s     $f2,0($5) # $f2 = y[i]
  mul.s   $f0,$f0,$f2  # $f0 = x[i] * y[i]
  addu    $5,$5,4   # y+i++
  add.s   $f4,$f4,$f0  # result += $f0
  addu    $3,$3,1   # i++
  slt     $2,$3,$6  # if i < n
  BNE     $2,$0,$L5 # then loop
  addu    $4,$4,4   # with x+i++
$L3:
  mov.s   $f0,$f4   # return val = result
  addu    $sp,$sp,8 # Deallocate frame
  j       $31       # return
```

Double Precision

```
double inner_prodD
  (double x[], double y[],
   int n)
{
  int i;
  double result = 0.0;
  for (i = 0; i < n; i++) {
    result += x[i] * y[i];
  }
  return result;
}
```

```
inner_prodD:
  # result in $f4,5
  mtc1    $0,$f4    # result = 0.0
  mtc1    $0,$f5
  subu    $sp,$sp,8 # allocate stack frame
  BLEZ    $6,$L8    # if n <= 0 goto $L8
  move    $3,$0     # i = 0
$L10:
  l.d     $f2,0($4) # $f2,3 = x[i]
  l.d     $f0,0($5) # $f0,1 = y[i]
  mul.d   $f2,$f2,$f0 # $f2,3 *= y[i]
  addu    $5,$5,8   # y+i++
  add.d   $f4,$f4,$f2 # result *= $f2,f3
  addu    $3,$3,1     # i++
  slt     $2,$3,$6    # if i < n
  BNE     $2,$0,$L10  # then loop
  addu    $4,$4,8     # with x+i++
$L8:
  mov.d   $f0,$f4     # return val = result
  addu    $sp,$sp,8   # deallocate frame
  j       $31         # return
```

Disassembled

```
0x400250 <inner_prodD>:         mtc1 $zero,$f4
0x400254 <inner_prodD+4>:       mtc1 $zero,$f5
0x400258 <inner_prodD+8>:       addiu $sp,$sp,-8
0x40025c <inner_prodD+12>:      BLEZ $a2,0x400290 <inner_prodD+64>
0x400260 <inner_prodD+16>:      move $v1,$zero
0x400264 <inner_prodD+20>:      lwc1 $f2,0($a0)
0x400268 <inner_prodD+24>:      lwc1 $f3,4($a0)
0x40026c <inner_prodD+28>:      lwc1 $f0,0($a1)
0x400270 <inner_prodD+32>:      lwc1 $f1,4($a1)
0x400274 <inner_prodD+36>:      addiu $v1,$v1,1
0x400278 <inner_prodD+40>:      mul.d $f2,$f2,$f0
0x40027c <inner_prodD+44>:      slt $v0,$v1,$a2
0x400280 <inner_prodD+48>:      addiu $a1,$a1,8
0x400284 <inner_prodD+52>:      add.d $f4,$f4,$f2
0x400288 <inner_prodD+56>:      BNE $v0,$zero,0x400264 <inner_prodD+20>
0x40028c <inner_prodD+60>:      addiu $a0,$a0,8
0x400290 <inner_prodD+64>:      mov.d $f0,$f4
0x400294 <inner_prodD+68>:      JR $ra
0x400298 <inner_prodD+72>:      addiu $sp,$sp,8
```