CMSC 216
Introduction to Computer Systems
Assembly MIPS 3

Function implementation overview

C

caller

```
...
r = f(args);
...
```

callee

```
rtype f(params):
  local vars
  code
  return rval
```

Assembly (assuming args and rval passed on stack)

```
  ...
begin_call:
  put args on stack
  put space for rval on stack
  jump to f
ret_addr:
  copy rval to r
  shrink stack to what it
   was at begin_call
  ...
```

```
f:
  put local vars on stack
  code    # access args, local vars
  put rval in rval-space on stack
  jump to  ret_addr
```

How does f access
- the return address (**ret_addr**)
- **args**, rval-space, local vars
- ...

How does **f** access the return address

Assembly

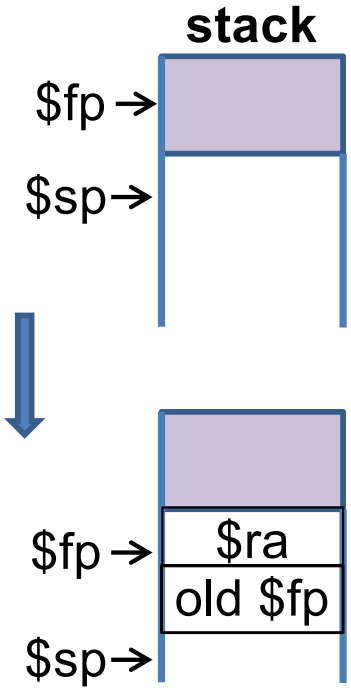
```
  ...
begin_call:
   ...
   jal f       # jump to f
ret_addr:

   ...
```

```
f:
  save $ra on stack

  ...

  restore $ra from stack
  jr $ra
```

- Caller does `jal f` to jump to **f** // puts the return address in **$ra**
- **f** pushes **$ra** on stack before doing anything else
- To return, **f** restores `$ra` from stack and does `jr $ra`

How does f access args, rval-space, local vars

- **f** can do offsets from the stack pointer (**$sp**) ... but **$sp** not fixed
- Instead, **f** uses another register, the **frame pointer** (**$fp**), to point to the saved **$ra** on stack, and does offsets from **$fp**
- **f** saves the old value of **$fp** on the stack just after the saved **$ra**
- To return: **f** sets **$sp** to **$fp**, restores **$fp** and **$ra**, and does **jr $ra**




```
f:                                          Prologue
  $sp = $sp - 8           // grow stack by 8 bytes
  mem[$sp + 8] = $ra      // save $ra (4 bytes)
  mem[$sp + 4] = $fp      // save $fp (4 bytes)
  $fp = $sp + 8           // location of saved $ra

  ...

                                            Epilogue
  $sp = $fp
  $fp = mem[$sp - 4]       // restore saved $fp
  $ra = mem[$sp]           // restore saved $ra
  jr $ra
```

Preserving registers across calls

- What about registers that the caller was using when it called f
  - caller can save them on the stack before the call and restore them after the return

  or
  - f can save them on the stack before using them and restore them before the return
- The convention is a bit of both
  - **Caller-saved** registers: $t0 - $t9
  - **Callee-saved** registers: $s0 - $s7, $ra, $fp (= $s8)

Passing args and rval in registers

- More efficient to pass values between caller and callee in registers instead of stack
- MIPS convention:
  - arguments in $a0–$a3 and, if more space is needed, the stack.
  - return value in $v0–$v1 and, if more space is needed, the stack.
- **In this course,** we may require all args to be passed on the stack, entirely avoiding $a0-$a3 (unless it's allowed explicitly)

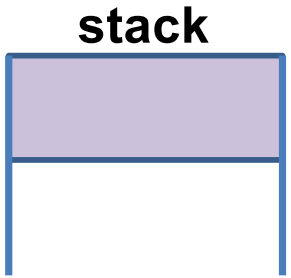
MIPS: Register usage conventions

Register Number	Register Name	Description
2-3	$v0 - $v1	(**v**alues) from expression evaluation and function results
4-7	$a0 - $a3	(**a**rguments) First four parameters for subroutine
8-15, 24-25	$t0 - $t9	**T**emporary variables. ***Caller-saved***
16-23	$s0 - $s7	**S**aved values representing final computed results. ***Callee-saved***
29	$sp	**S**tack **p**ointer
30	$fp (=$s8)	**F**rame **p**ointer (***Callee-saved***)
31	$ra	**R**eturn **a**ddress (***Callee-saved***)

Function implementation: all together

```
...
r = f(args);
...
```

```
rtype f(params) {
  body
  return rval
}
```



Call
- push caller-saved regs if needed
- put args into registers $a0-$a3
- push any additional args and rval-space on stack
- `jal f`

Prologue
- push $ra (return addr) on stack
- push $fp (frame pointer) on stack
- $fp ← addr of saved $ra on stack

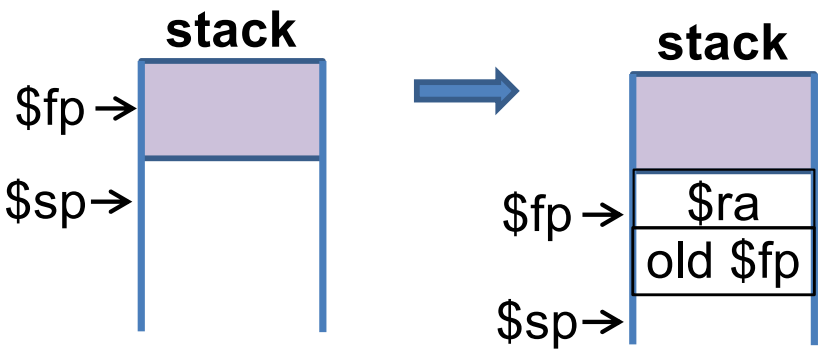
body
- push callee-saved regs if needed
- allocate local variables on stack
- access args, local vars and rval-space on stack by offset from $fp
- put rval in $v0-$v1 and, if needed, in stack

Return
- read rval from $v0-$v1 and, if needed, from stack
- store rval in destination `r`
- restore caller-saved regs and stack as it was at start of call

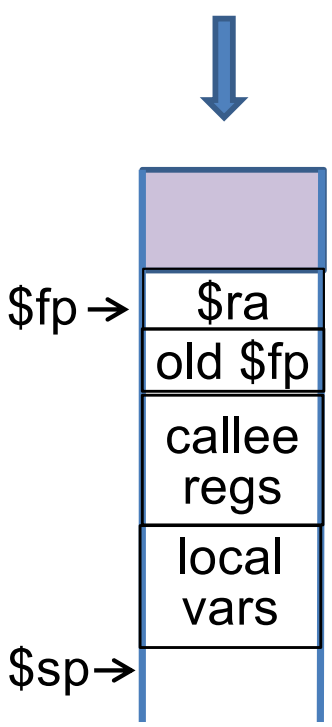
Epilogue
- restore $ra, $fp, $sp as at start of prologue
- `jr $ra`

Function structure

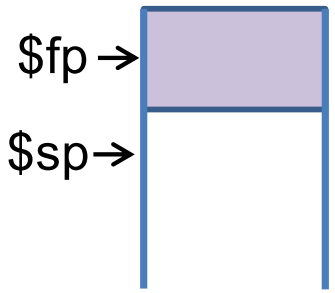
```
# PROLOGUE
subu $sp, $sp, 8    # grow stack 8B
sw   $ra, 8($sp)    # push $ra (4B)
sw   $fp, 4($sp)    # push $fp (4B)
addu $fp, $sp, 8    # set $fp to saved $ra
```




```
# BODY
grow stack for callee-saved regs, local vars
access args, rval-space and local vars by offset ($fp)
for each function call:
    grow stack for caller-saved regs, args, rval-space
    jal function
    copy rval, restore caller-saved regs
    shrink stack by caller-saved regs, args, rval-space
return: restore callee-saved regs
```

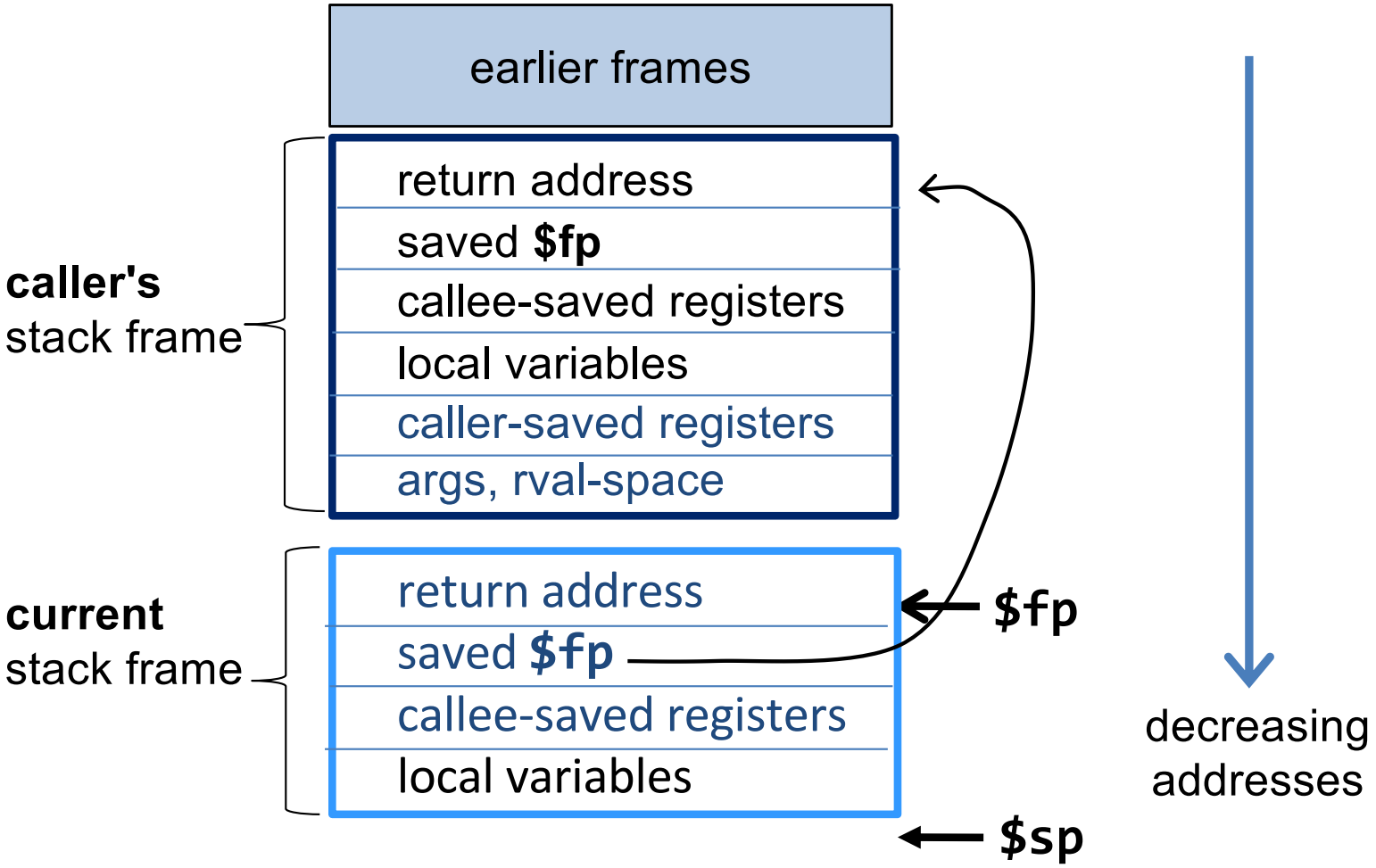



```
# EPILOGUE
move $sp, $fp       # restore $sp
lw   $ra, ($fp)     # restore saved $ra
lw   $fp, -4($sp)   # restore saved $fp
jr   $ra            # return to caller
```



Stack frames

- Each ***active*** function call (called and not returned yet) has a ***stack frame*** (aka ***activation record***) on the stack, starting at the saved caller's ***ra***.



Conventions in this course

- Obey the MIPS convention for passing args and return value
  - The only exception is if we explicitly require to pass **all** args in stack
- Args in stack are pushed in **right-to-left** order
  - eg, **`f(x, y)`**: push **`y`**, push **`x`** // **`x`** is between **`y`** and the stack top
- Obey the caller-saved and callee-saved convention
- Access function params and local variables by offsets from $fp
- For function call, use offsets from $sp to save args and rval-space,
- **`main()`** is a function, so has prologue and epilogue
  - omit them only if explicitly allowed (which would be ok if **`main()`** does not call a function)

Comments

- Caller-saved registers: $t0 - $t9
- Callee-saved registers: $s0 - $s7, $ra, $fp (= $s8)
- Because a callee can itself become a caller, args passed in registers ($a0-$a3) may need to be saved on the stack
- $sp points to the first free location past the top of the stack.
- Maximum value of $sp is **`0x7ffffffc`** (a word address)
- User program usually starts with a lower $sp than max because kernel pushes stuff (argc, argv, env) before calling the user program
- In some examples, we may initialize $sp to a specified value (usually **`0x7ffffffc`** for an initially empty stack)

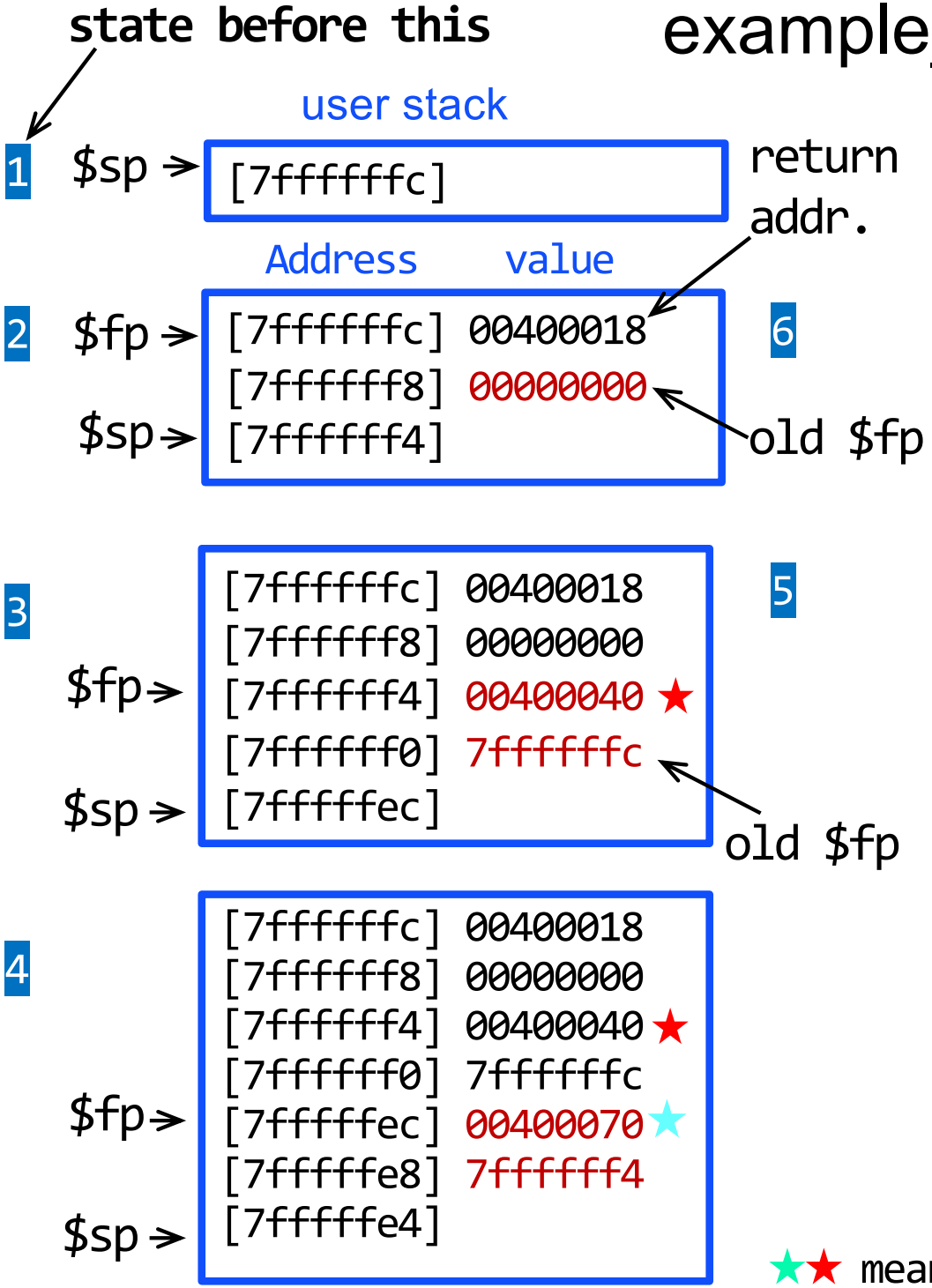
CMSC 216
Introduction to Computer Systems
Assembly MIPS 3 examples

example_f1: no local vars, no args, no rval

```
int main(void) {
  f();
  return 0;
}

void f(void) {
  printf("in f\n");
  g();
}

void g(void) {
  printf("in g\n");
}
```

```
        .data
strf:   .asciiz "in f\n"
strg:   .asciiz "in g\n"

        .text
main:   li   $sp, 0x7ffffffc # initially empty stack
        PROLOGUE
        jal   f              # call f
        EPILOGUE

f:      PROLOGUE
        # BODY
        <printf("%s", strf)>
        jal  g
        EPILOGUE

g:      PROLOGUE
        # BODY
        <printf("%s", strg)>
        EPILOGUE
```

example_f1.s




```
main:
   li   $sp, 0x7ffffffc
1  PROLOGUE
2  jal  f              # call f
6★ EPILOGUE

f: PROLOGUE
   <print_str(strf)>
3  jal  g              # call g
5★ EPILOGUE

g: PROLOGUE
4  <print_str(strg)>
   EPILOGUE
```

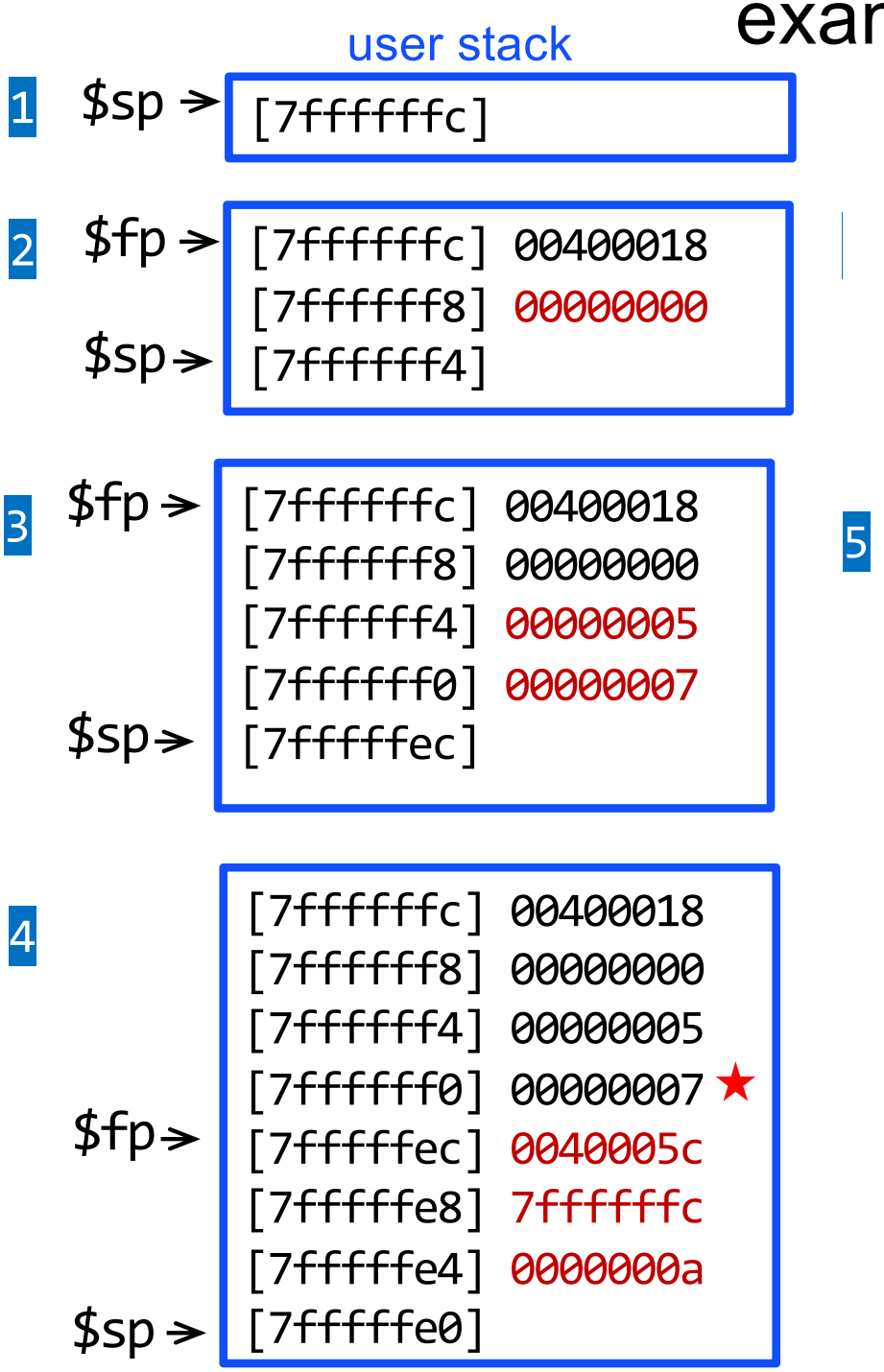
★★ means the first instructions in epilogues

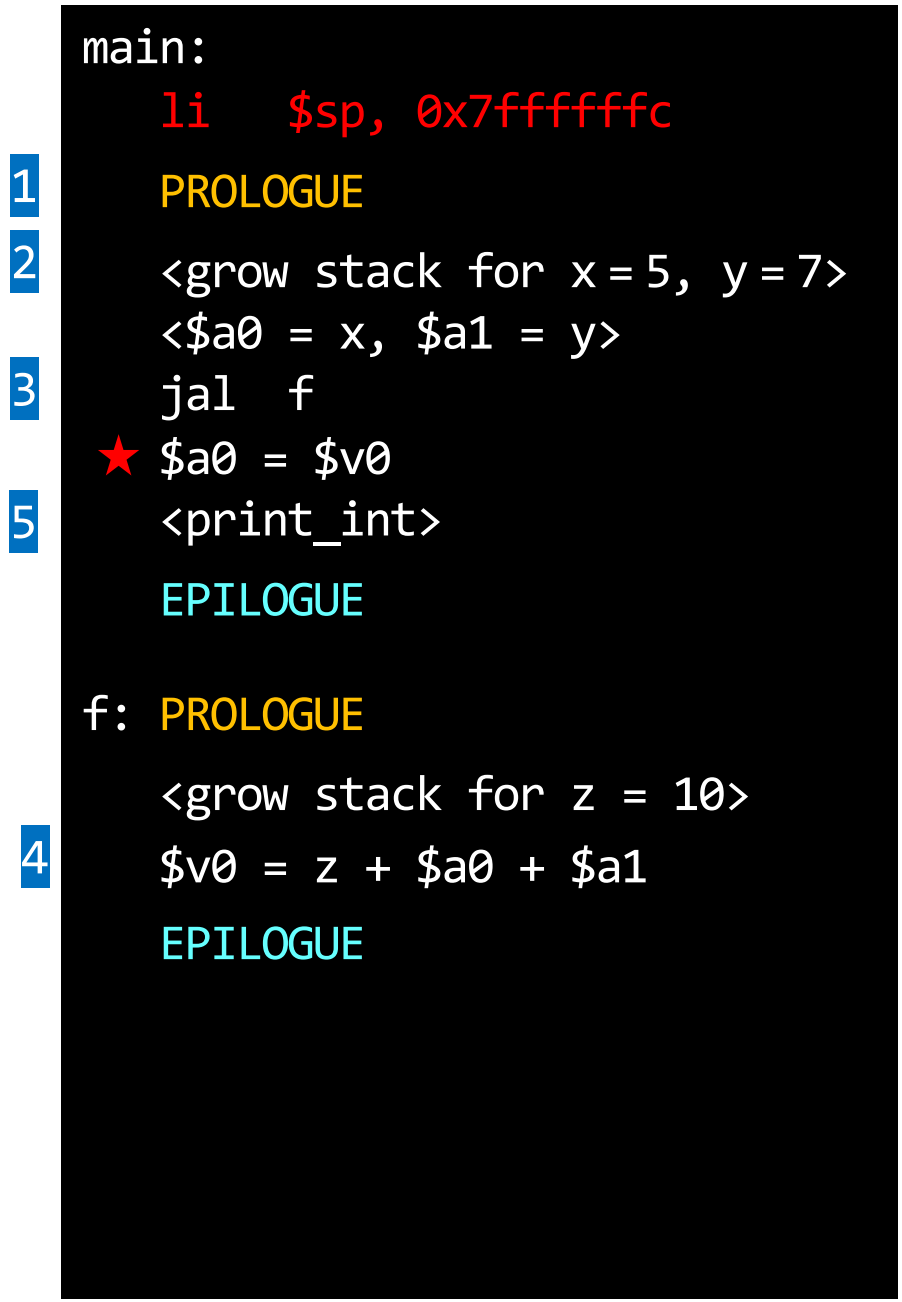
example_f2: local vars, args in regs, rval

```
int main(void) {
  int x, y;
  x = 5;
  y = 7;
  printf("%d\n", f(x,y));
}

void f(int a, int b) {
  int z;
  z = 10;
  return z + a + b;
}
```

```
        .text
main:   li   $sp, 0x7ffffffc # initially empty stack
        PROLOGUE
        subu $sp, $sp, 8       # grow stack for x,y
        li   $t0, 5            # x = 5
        sw   $t0, 8($sp)
        li   $t0, 7            # y = 7
        sw   $t0, 4($sp)
        # f(x,y): pass args x, y in a0, a1
        lw   $a0, -8($fp)      # $a0 = x (base $fp)
        lw   $a1, -12($fp)     # $a1 = y (base $fp)
        jal  f                 # call f
        <move result (in $v0) to $a0, print>
        EPILOGUE

f:      PROLOGUE
        # BODY
        <grow stack for z, set to 10>
        $v0 = z + $a0 + $a1
        EPILOGUE
```

example_f2.s


user stack
1
$sp ➔
[7ffffffc]
2
$fp ➔
[7ffffffc] 00400018
[7fffffff8] 00000000
$sp ➔
[7ffffff4]
3
$fp ➔
[7ffffffc] 00400018
[7fffffff8] 00000000
[7ffffff4] 00000005
[7ffffff0] 00000007
$sp ➔
[7fffffec]
4
[7ffffffc] 00400018
[7fffffff8] 00000000
[7ffffff4] 00000005
[7ffffff0] 00000007 ★
$fp ➔
[7fffffec] 0040005c
[7fffffe8] 7ffffffc
[7fffffe4] 0000000a
$sp ➔
[7fffffe0]


main:
li $sp, 0x7ffffffc
1
PROLOGUE
2
<grow stack for x = 5, y = 7>
<$a0 = x, $a1 = y>
3
jal f
★ $a0 = $v0
5
<print_int>
EPILOGUE
f: PROLOGUE
<grow stack for z = 10>
4
$v0 = z + $a0 + $a1
EPILOGUE

example_f2b: example_f2 with args in stack

```
int main(void) {
  int x, y;
  x = 5;
  y = 7;
  printf("%d\n", f(x,y));
}

void f(int a, int b) {
  int z;
  z = 10;
  return z + a + b;
}
```

```
main: li   $sp, 0x7ffffffc # initially empty stack
      PROLOGUE
      <grow stack for x = 5, y = 7>
      # call f(x,y)
      <load x, load y (using base $fp)>
      <grow stack for args>
      <push y, push x (using base $sp)>
      jal  f                  # call f
      <move result (in $v0) to $a0, print>
      EPILOGUE

f:    PROLOGUE
      <grow stack for z = 10>
      # using base $fp below
      lw   $t1, 4($fp)        # arg_a into $t1
      lw   $t2, 8($fp)        # arg_b into $t2
      lw   $t0, -8($fp)       # z into $t0
      $v0 = $t0 + $t1 + $t2
      EPILOGUE
```

example_f2b


```
main: li   $sp, 0x7ffffffc
      PROLOGUE
      <grow stack for x = 5, y = 7>
      # call f(x,y)
      <load x, load y (using base $fp)>
      <grow stack for args>
      <push y, push x (using base $sp)>
      jal  f                  # call f
      <move result (in $v0) to $a0, print>
      EPILOGUE

f:    PROLOGUE
      <grow stack for z = 10>
      # using base $fp below
      lw   $t1, 4($fp)        # arg_a into $t1
      lw   $t1, 8($fp)        # arg_b into $t2
      lw   $t0, -8($fp)       # z into $t0
      $v0 = $t0 + $t1 + $t2
      EPILOGUE
```

example_f3: recursion, local vars, args in regs, rval

```
int main() {
  int n = 4;
  printf("%d\n", f(n));
  return 0;
}

/* recursively return
   1*1 + ... + j*j */

int f(int j) {
  if (j == 1)
    return 1;
  else
    return j*j + f(j-1);
}
```

```
        .text
main:   li   $sp, 0x7ffffffc # init sp
        PROLOGUE
        <grow stack for n = 4>
        lw   $a0, -8($fp)    # arg = n
        jal  f               # $v0 = f(n)
        <printf("%d\n", result)>
        EPILOGUE

f:      PROLOGUE
        bne  $a0, 1, rec
        move $v0, $a0        # arg j == 1
        j    ret
rec:
        subu $sp, $sp, 4     # arg j > 1
        sw   $a0, 4($sp)     # save arg j
        sub  $a0, $a0, 1     # $a0 = arg j-1
        jal  f               # $v0 = f(j-1)
        lw   $t1, 4($sp)     # $t1 = j
        mul  $t1, $t1, $t1   # $t1 = j*j
        add  $v0, $v0, $t1   # $v0 = f(j-1) + j*j
ret:    EPILOGUE
```

example_f3.s

user stack

	[7ffffffc]	00400018
	[7ffffff8]	00000000
	[7ffffff4]	00000004
$fp1 →	[**7ffffff0**]	00400050 ★
	[7fffffec]	7ffffffc
$sp1 →	[7fffffe8]	00000004
$fp2 →	[**7fffffe4**]	004000a8 ★
	[7fffffe0]	**7ffffff0**
$sp2 →	[7fffffdc]	00000003
$fp3 →	[**7fffffd8**]	004000a8 ★
	[7fffffd4]	**7fffffe4**
$sp3 →	[7fffffd0]	00000002
$fp4 →	[**7fffffcc**]	004000a8 ★
	[7fffffc8]	**7fffffd8**
$sp4 →	[7fffffc4]	

```
        .text
main:   li   $sp, 0x7ffffffc # init sp
        PROLOGUE
        <grow stack for n = 4>
        lw   $a0, -8($fp)   # arg = n
        jal  f              # call f
   ★    <print result (in $v0), newline>
        EPILOGUE

f:      PROLOGUE
        bne  $a0, 1, rec    # if j != 1

        move $v0, $a0       # arg j == 1
        j    ret

rec:    subu $sp, $sp, 4    # arg j > 1
        sw   $a0, 4($sp)    # save arg j
        sub  $a0, $a0, 1

        jal  f              # $v0 = f(j-1)
   ★    lw   $t1, 4($sp)    # $t1 = j
        mul  $t1, $t1, $t1 # $t1 = j*j
        add  $v0, $v0, $t1 # $v0 = f(j)

ret:    EPILOGUE
```