Lec 10: Assembler

Kavita Bala
CS 3410, Fall 2008
Computer Science
Cornell University

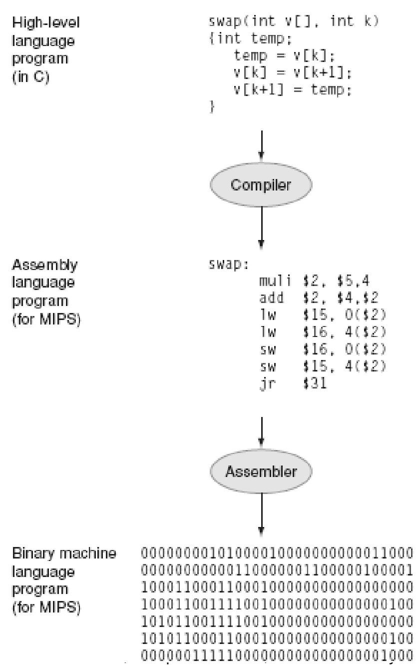
Announcements

- HW 2 is out
  - Due Wed after Fall Break
  - Robot-wide paths
- PA 1 is due next Wed
  - Don't use incrementor 4 times
- Ask us questions if in doubt

Examples

- A[12] = h + A[8]

- lw, $t0, 32($s3)
- add $t0, $s2, $t0
- sw $t0, 48($s3)



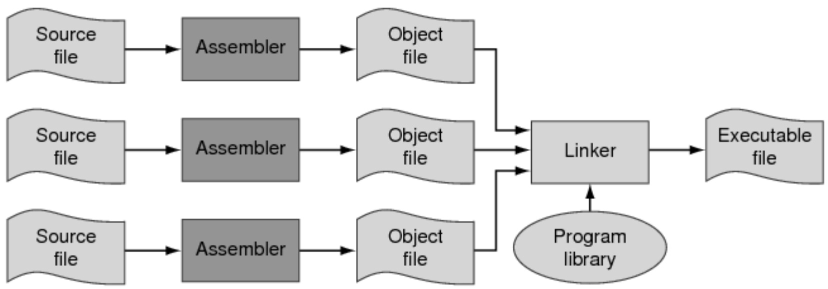


FIGURE A.1.1 The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Assembler

- Translates text assembly language to binary machine code
- Input: a text file containing MIPS instructions in human readable form
- Output: an object file (.o file in Unix, .obj in Windows) containing MIPS instructions in executable form

Assembly Language Instructions

- Arithmetic
  - ADD, ADDU, SUB, SUBU, AND, OR, XOR, NOR, SLT, SLTU
  - ADDI, ADDIU, ANDI, ORI, XORI, LUI, SLL, SRL, SLLV, SRLV, SRAV, SLTI, SLTIU
  - MULT, DIV, MFLO, MTLO, MFHI, MTHI
- Control Flow
  - BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ
  - J, JR, JAL, JALR, BLTZAL, BGEZAL
- Memory
  - LW, LH, LB, LHU, LBU
  - SW, SH, SB
- Special
  - SYSCALL, BREAK, SYNC, COPROC, LL, SC

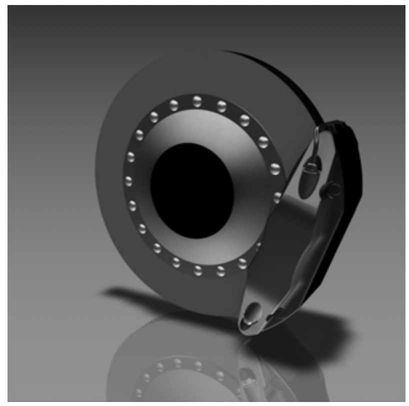
Assembly Language

- Assembly language is used to specify programs at a low-level

- Will I program in assembly?
  - I did, for kernel hacking
  - For performance (though compilers are getting better)
  - For highly time critical sections
  - For hardware without high level languages

Example: GPU Phong Shader

- Want to compute
  - out = N.L + (R.L)^n



Example: GPU Phong Shader

- ADD R0, c[3], -v[OPOS]   // L-P
- DP3 R1, R0, R0   // ||L-P||^2
- RSQ R2, R1.W   // 1/||L-P||
- MUL R0, R0, R2   // R0 = L

- DP3 R3, R0, v[NRML]   // R3 = N.L

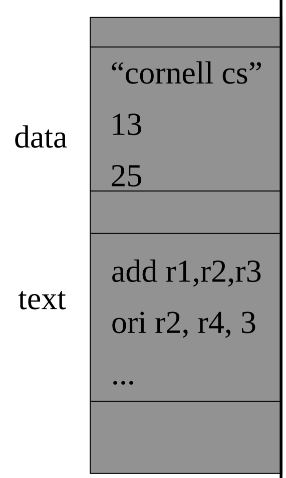
- ….   // Compute E
- DP3 R7, R4, v[NRML]   // E.N
- MUL R7, R7, c[6]   // 2 (E.N)
- MAD R8, R7, v[NRML], -R4 // 2 (E.N)N-E
- DP3 R9, R8, R0   // R.L
- LOG R10, R9.x   // LOG (R.L)
- MUL R9, c[5].x, R10.z   // n*(LOG(R.L))
- EXP R11, R9.z   // (R.L)^n
- …

Assembly Language

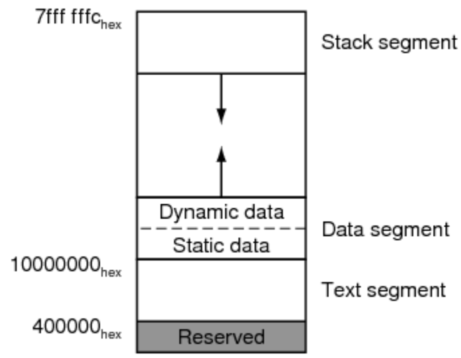
- Assembly language is used to specify programs at a low-level
- What does a program consist of?
  – MIPS instructions
  – Program data (strings, variables, etc)

Program Layout

- Programs consist of segments used for different purposes
  – Text: holds instructions
  – Data: holds statically allocated program data such as variables, strings, etc.



When you run the program



Assembling Programs

```
        .text
        .ent main
main:   la $4, Larray
        li $5, 15
        ...
        li $4, 0
        jal exit
        .end main
        .data
Larray:
        .long 51, 491, 3991
```

- Programs consist of a mix of instructions, pseudo-ops and assembler directives
- Assembler lays out binary values in memory based on directives

Example pseudo-ops

- blt = slt and bne
  - blt $s3, $s4, label

  Equivalent to
  - slt $at, $s3, $s4
  - bne $at, $zero, label

- Use register $at (assembler temporary) to compile this

Examples

- gcc –S helloWorld.c

- gcc –S add1To100.c

- gcc –S add1To100Sq.c

Add 1 to 100 Square

```
#include <stdio.h>

int main (int argc, char* argv[]) {

   int count = 0;
   int i = 0;
   for (i = 1; i <= 100; i++) { count += i*i; }
   printf ("The sum from 0 .. 100 is %d\n", count);
}
```



```
addiu    $29, $29, -32
sw       $31, 20($29)
sw       $4,  32($29)
sw       $5,  36($29)
sw       $0,  24($29)
sw       $0,  28($29)
lw       $14, 28($29)
lw       $24, 24($29)
multu    $14, $14
addiu    $8,  $14, 1
slti     $1,  $8, 101
sw       $8,  28($29)
mflo     $15
addu     $25, $24, $15
bne      $1,  $0, -9
sw       $25, 24($29)
lui      $4,  4096
lw       $5,  24($29)
jal      1048812
addiu    $4,  $4, 1072
lw       $31, 20($29)
addiu    $29, $29, 32
jr       $31
move     $2,  $0
```

y

```
        .text
        .align    2
        .globl    main
main:
        subu      $sp, $sp, 32
        sw        $ra, 20($sp)
        sd        $a0, 32($sp)
        sw        $0,  24($sp)
        sw        $0,  28($sp)
loop:
        lw        $t6, 28($sp)
        mul       $t7, $t6, $t6
        lw        $t8, 24($sp)
        addu      $t9, $t8, $t7
        sw        $t9, 24($sp)
        addu      $t0, $t6, 1
        sw        $t0, 28($sp)
        ble       $t0, 100, loop
        la        $a0, str
        lw        $a1, 24($sp)
        jal       printf
        move      $v0, $0
        lw        $ra, 20($sp)
        addu      $sp, $sp, 32
        jr        $ra


        .data
        .align    0
str:
        .asciiz   "The sum from 0 .. 100 is %d\n"
```

Procedure Calls

Procedures

- Enable code to be reused by allowing code snippets to be invoked

- Will need a way to
  - call the routine
  - pass arguments to it
    - fixed length
    - variable length
    - Recursive calls
  - return value to caller
  - manage registers

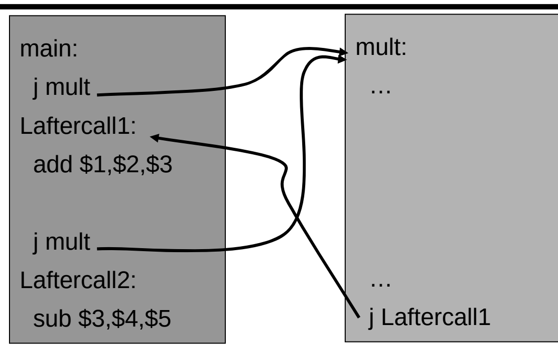
Take 1: Use Jumps

```
main:
  j mult
Laftercall1:
  add $1,$2,$3
```

```
mult:
  …

  …
  j Laftercall1
```

- Jumps and branches can transfer control to the callee (called procedure)
- Jumps and branches can transfer control back

Take 1: Use Jumps



- Jumps and branches can transfer control to the callee
- Jumps and branches can transfer control back
- What happens when there are multiple calls from different call sites?

Jump And Link

- JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register $31
- Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register $31

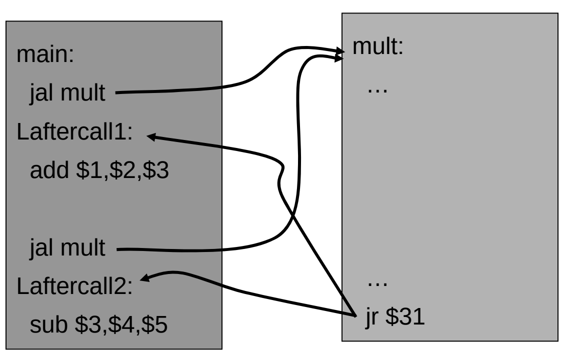
Take 2: JAL/JR

```
main:
  jal mult
Laftercall1:
  add $1,$2,$3

  jal mult
Laftercall2:
  sub $3,$4,$5
```

```
mult:
  …


  …
  jr $31
```

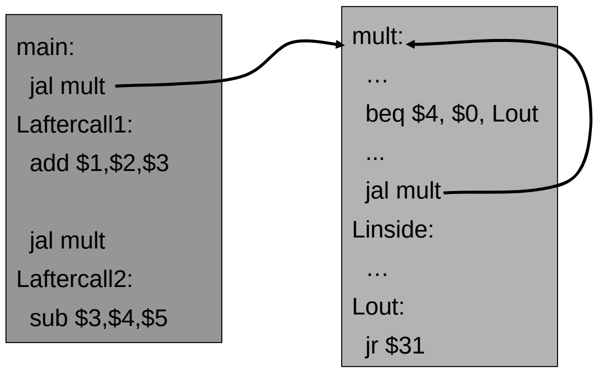
- JAL saves the PC in register $31
- Subroutine returns by jumping to $31

Take 2: JAL/JR



- JAL saves the PC in register $31
- Subroutine returns by jumping to $31
- What happens for recursive invocations?

Take 2: JAL/JR

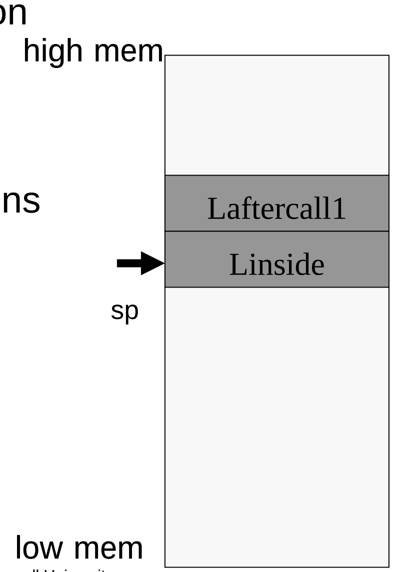


- Recursion overwrites contents of $31
- Need to save and restore the register contents

Call Stacks

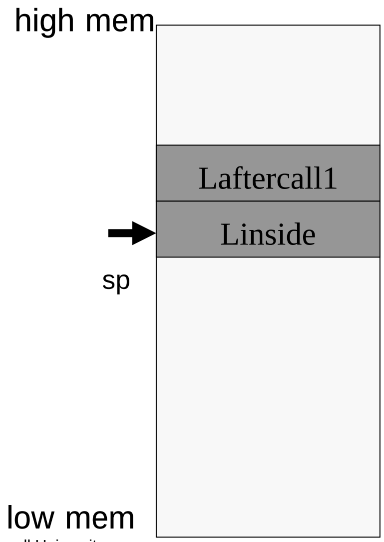
- A call stack contains activation records (aka stack frames)

- Each activation record contains
  - the return address for that invocation
  - the local variables for that procedure



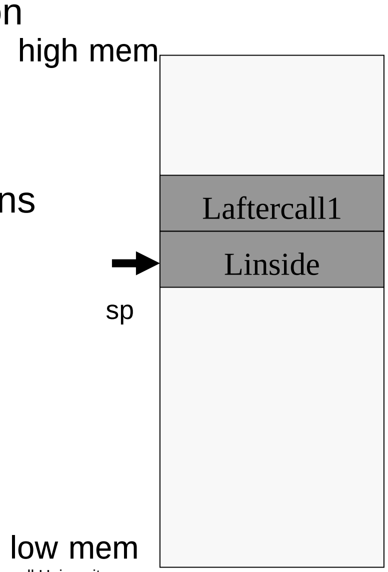
Call Stacks

- A stack pointer (sp) keeps track of the top of the stack
  - dedicated register ($29) on the MIPS
- Manipulated by push/pop operations
  - push: move sp down, store
  - pop: load, move sp up



Call Stacks

- A call stack contains activation records (aka stack frames)
- Each activation record contains
  - the return address for that invocation
  - the local variables for that procedure



Stack Growth

- Stacks start at a high address in memory
- Stacks grow down as frames are pushed on
  - Recall that the data region starts at a low address and grows up
  - The growth potential of stacks and data region are not artificially limited

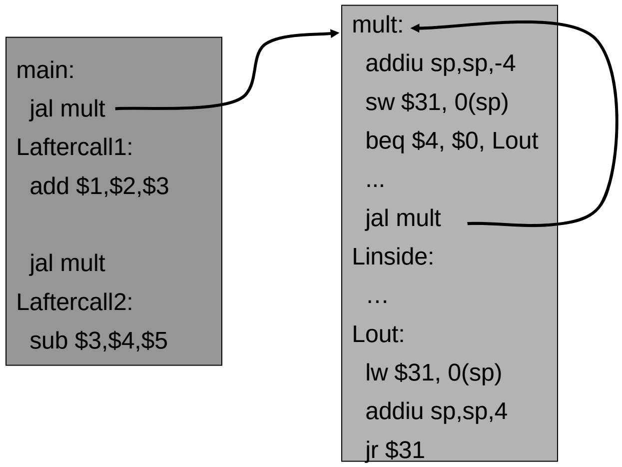
Take 3: JAL/JR with Activation Records

```
main:
  jal mult
Laftercall1:
  add $1,$2,$3

  jal mult
Laftercall2:
  sub $3,$4,$5
```

```
mult:
  addiu sp,sp,-4
  sw $31, 0(sp)
  beq $4, $0, Lout
  ...
  jal mult
Linside:
  …
Lout:
  lw $31, 0(sp)
  addiu sp,sp,4
  jr $31
```

- Stack used to save and restore contents of $31

Take 3: JAL/JR with Activation Records



- Stack used to save and restore contents of $31
- How about arguments?

Arguments & Return Values

- Need consistent way of passing arguments and getting the result of a subroutine invocation
- Given a procedure signature, need to know where arguments should be placed
  - `int min(int a, int b);`
  - `int subf(int a, int b, int c, int d, int e);`
  - `int isalpha(char c);`
  - `int treesort(struct Tree *root);`
  - `struct Node *createNode();`
  - `struct Node mynode();`
- Too many combinations of char, short, int, void *, struct, etc.
  - MIPS treats char, short, int and void * identically

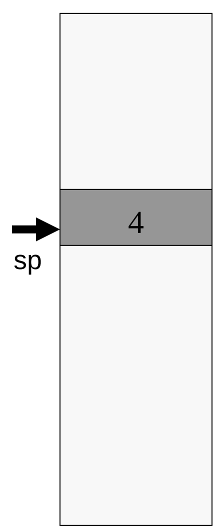
Simple Argument Passing

```
main:
  li a0, 6
  li a1, 7
  jal min
  // result in v0
```

- First four arguments are passed in registers
  - Specifically, $4, $5, $6 and $7, aka a0, a1, a2, a3
- The returned result is passed back in a register
  - Specifically, $2, aka v0

Many Arguments

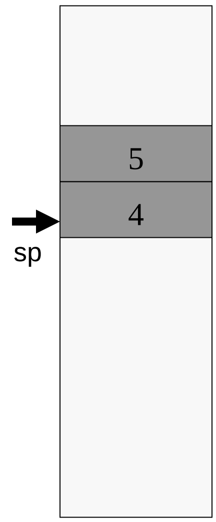
```
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  li $8, 4
  addiu sp,sp,-4
  sw $8, 0(sp)
  jal subf
  // result in v0
```



- What if there are more than 4 arguments?
- Use the stack for the additional arguments
  - "spill"

Many Arguments

```
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  addiu sp,sp,-8
  li $8, 4
  sw $8, 0(sp)
  li $8, 5
  sw $8, 4(sp)
  jal subf
  // result in v0
```



- What if there are more than 4 arguments?
- Use the stack for the additional arguments
  - "spill"

Variable Length Arguments

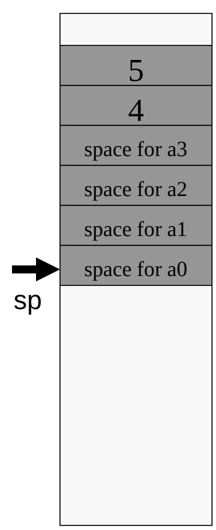
- `printf("Coordinates are: %d %d %d\n", 1, 2, 3);`
- Could just use the regular calling convention, placing first four arguments in registers, spilling the rest onto the stack
  - Callee requires special-case code
  - if(argno == 1) use a0, … else if (argno == 4) use a3, else use stack offset

Variable Length Arguments

- Best to use an (initially confusing but ultimately simpler) approach:
  - Pass the first four arguments in registers, as usual
  - Pass the rest on the stack
  - Reserve space on the stack for all arguments, including the first four
- Simplifies functions that use variable-length arguments
  - Store a0-a3 on the slots allocated on the stack, refer to all arguments through the stack

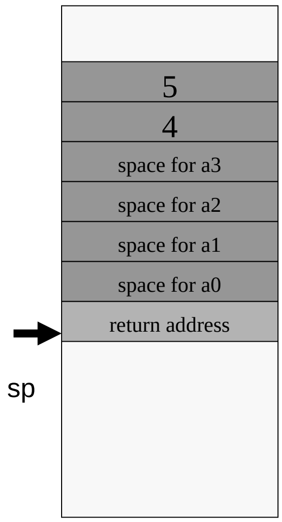
Register Layout on Stack

```
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  addiu sp,sp,-24
  li $8, 4
  sw $8, 16(sp)
  li $8, 5
  sw $8, 20(sp)
  jal subf
  // result in v0
```



- First four arguments are in registers
- The rest are on the stack
- There is room on the stack for the first four arguments, just in case

Frame Layout on Stack




```
blue() {
   pink(0,1,2,3,4,5);
}
pink() {
    ...
}
```


Pointers and Structures

- Pointers are 32-bits, treat just like ints
- Pointers to structs are pointers
- C allows passing whole structs
  - `int distance(struct Point p1, struct Point p2);`
  - Treat like a collection of consecutive 32-bit arguments, use registers for first 4 words, stack for rest
  - Inefficient and to be avoided, better to use

```
int
 distance(struct Point *p1, struct Point *p2);
```

Globals and Locals

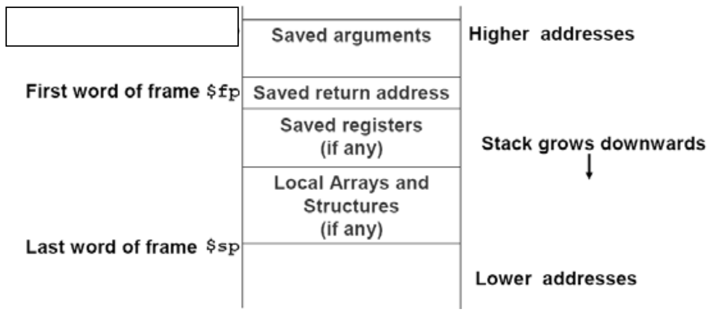
- Global variables are allocated in the “data” region of the program
  - Exist for all time, accessible to all routines
- Local variables are allocated within the stack frame
  - Exist solely for the duration of the stack frame
- Dangling pointers are pointers into a destroyed stack frame
  - C lets you create these, Java does not
  - int *foo() { int a; return &a; }

Frame Pointer

- It is sometimes cumbersome to keep track of location of data on the stack
  - The offsets change as new values are pushed onto and popped off of the stack
- Keep a pointer to the top of the stack frame
  - Simplifies the task of referring to items on the stack
- A frame pointer, $30, aka fp
  - Value of sp upon procedure entry
  - Can be used to restore sp on exit

© Kavita Bala, Computer Science, Cornell University

Frame Pointer



Register Usage

- Suppose a routine would like to store a value in a register
- Two options: caller-save and callee-save
- MIPS calling convention supports both

Register Usage

- Callee-save
  - Save it if you modify it
  - Assumes caller needs it
  - Save the previous contents of the register on procedure entry, restore just before procedure return
  - E.g. $31 (if you are a non-leaf… what is that?)

- Caller-save
  - Save it if you need it after the call
  - Assume callee can clobber any one of the registers
  - Save contents of the register before proc call
  - Restore after the call

Caller vs Callee tradeoff

- MIPS supports both

- Callee-save regs: $16-$23 (s0-s7)
- Caller-save regs: $8-$15,$24,$25 (t0-t9)

Callee-Save

```
mult:
 addiu sp,sp,-12
 sw $31,8(sp)
 sw $17, 4(sp)
 sw $16, 0(sp)
         …
 [use $17 and $16]
   …
 lw $31,8(sp)
 lw $17, 4(sp)
 lw $16, 0(sp)
 addiu sp,sp,12
```

- Assume caller is using the registers
- Save on entry, restore on exit
- Pays off if caller is actually using the registers, else the save and restore are wasted

Caller-Save

```
main:
 …
 [use $9 & $8]
 …
 addiu sp,sp,-8
 sw $9, 4(sp)
 sw $8, 0(sp)
 jal mult
 lw $9, 4(sp)
 lw $8, 0(sp)
 addiu sp,sp,8
 …
 [use $9 & $8]
```

- Assume registers are free for the taking
- But other subroutines will do the same
  - must protect values that will be used later
  - save and restore them before and after subroutine invocations
- Pays off if a routine makes few calls to other routines with values that need to be preserved

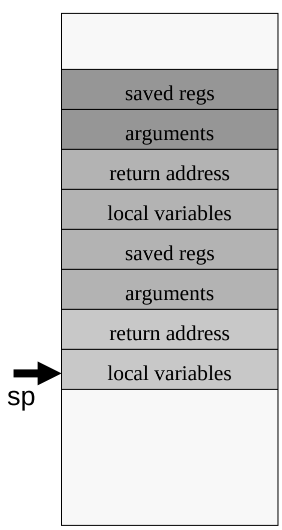
Leaf vs. non-leaf

- Leaf
  - Simple, fast
  - Don't save registers

- int f(int x, int y) {return (x+y);}

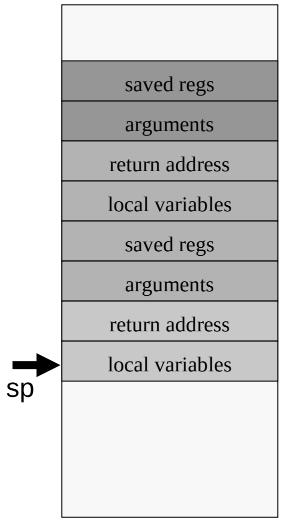
- f: add $v0, $a0, $a1 # add x and y
- j $ra # return

Frame Layout on Stack




```
blue() {
   pink(0,1,2,3,4,5);
}
pink() {
    orange(10,11,12,13,14);
}
```

Buffer Overflows




```
blue() {
   pink(0,1,2,3,4,5);
}
pink() {
    orange(10,11,12,13,14);
}
orange() {
         char buf[100];
         gets(buf); // read string, no check
}
```