Programming Universal Computers Instruction Sets

Lecture 5

Prof. Bienvenido Velez

What do we know?

From

Instruction Set Architecture

To

Processor Implementation

What Next?

How do we get here in the first place?

Instruction Set Design

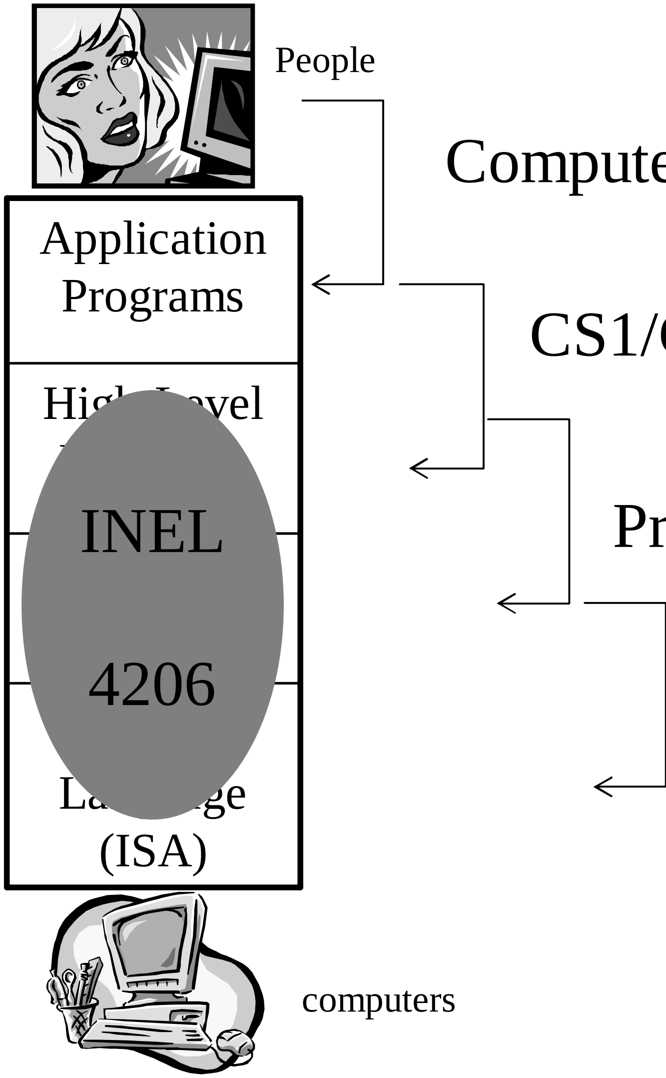
Outline

- Virtual Machines: Interpretation Revisited
- Example: From HLL to Machine Code
- Implementing HLL Abstractions
  - Control structures
  - Data Structures
  - Procedures and Functions

Virtual Machines (VM's)

Type of Virtual Machine	Examples	Instruction Elements	Data Elements	Comments
Application Programs	Spreadsheet, Word Processor	Drag & Drop, GUI ops, macros	cells, paragraphs, sections	Visual, Graphical, Interactive Application Specific Abstractions Easy for Humans Hides HLL Level
High-Level Language	C, C++, Java, FORTRAN, Pascal	if-then-else, procedures, loops	arrays, structures	Modular, Structured, Model Human Language/Thought General Purpose Abstractions Hides Lower Levels
Assembly-Level	SPIM, MASM	directives, pseudo-instructions, macros	registers, labelled memory cells	Symbolic Instructions/Data Hides some machine details like alignment, address calculations Exposes Machine ISA
Machine-Level (ISA)	MIPS, Intel 80x86	load, store, add, branch	bits, binary addresses	Numeric, Binary Difficult for Humans

Computer Science in Perspective



Computer Human Interaction, User Interfaces

CS1/CS2, Programming, Data Structures

Programming Languages, Compilers

Computer Architecture

INTERPRETATION
A **CORE** theme all throughout Computer Science

Computing Integer Division
Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

We ignore procedures and I/O for now

Easy I

A Simple Accumulator Processor

Instruction Set Architecture (ISA)

Instruction Set

Symbolic Name	Opcode	Action I=0	Symbolic Name	Action I=1
Comp	00 000	AC ? not AC	Comp	AC <- not AC
ShR	00 001	AC ? AC / 2	ShR	AC ? AC / 2
BrN*i*	00 010	AC < 0 ? PC ? X	BrN	AC < 0 ? PC ? MEM[X]
Jump*i*	00 011	PC ? X	Jump	PC ? MEM[X]
Store*i*	00 100	MEM[X] ? AC	Store	MEM[MEM[X]] ? AC
Load*i*	00 101	AC ? MEM[X]	Load	AC ? MEM[MEM[X]]
And*i*	00 110	AC ? AC and X	And	AC ? AC and MEM[X]
Add*i*	00 111	AC ? AC + X	Add	AC ? AC + MEM[X]

Computing Integer Division

Iterative C++ Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++
HLL

Easy-I
Assembly Language

Fall 2002

Computing Integer Division

Iterative C++ Version

Translate Data: Global Layout

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

```
0:      andi   0          # AC = 0
        addi   12
        storei 1000       # a = 12 (a  tored @ 1000)
        andi   0          # AC = 0
        addi   4
        storei 1004       # b = 4     tored @ 1004)
        andi   0          # AC =
        storei 1008       # resu    0 (result @ 1008)
```

Issues
- Memory allocation
- Data Alignment
- Data Sizing

C++
HLL

Easy-I
Assembly Language

Fall 2002

Computing Integer Division
Iterative C++ Version

Translate Code: Conditionals If-Then

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++
HLL

Easy-I
Assembly Language

```
0:      andi   0        # AC = 0
        addi   12
        storei 1000     # a = 12 (a stored @ 1000)
        andi   0        # AC = 0
        addi   4
        storei 1004     # b = 4 (b stored @ 1004)
        andi   0        # AC = 0
        storei 1008     # result = 0 (result @ 1008)
main:   loadi  1004     # compute a – b in AC
        comp            # using 2's complement add
        addi   1
        add    1000
        brni   exit     # exit if AC negative



exit:
```

Issues
- Must translate HLL boolean expression into ISA-level branching condition

Computing Integer Division

Iterative C++ Version

Translate Code: Iteration (loops)

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++
HLL

Easy-I
Assembly Language

```
0:        andi   0          # AC = 0
          addi   12
          storei 1000       # a = 12 (a stored @ 1000)
          andi   0          # AC = 0
          addi   4
          storei 1004       # b = 4 (b stored @ 1004)
          andi   0          # AC = 0
          storei 1008       # result = 0 (result @ 1008)
main:     loadi  1004       # compute a – b in AC
          comp              # using 2's complement add
          addi   1
          add    1000
          brni   exit       # exit if AC negative
loop:     loadi  1000
          brni   endloop




          jump   loop
endloop:
exit:
```

Fall 2002

Computing Integer Division

Iterative C++ Version

Translate Code: Arithmetic Ops

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++
HLL

Easy-I
Assembly Language

```
0:       andi   0         # AC = 0
         addi   12
         storei 1000      # a = 12 (a stored @ 1000)
         andi   0         # AC = 0
         addi   4
         storei 1004      # b = 4 (b stored @ 1004)
         andi   0         # AC = 0
         storei 1008      # result = 0 (result @ 1008)
main:    loadi  1004      # compute a – b in AC
         comp             # using 2's complement add
         addi   1
         add    1000
         brni   exit      # exit if AC negative
loop:    loadi  1000
         brni   endloop
         loadi  1004      # compute a – b in AC
         comp             # using 2's complement add
         addi   1
         add    1000      # Uses indirect bit I = 1



         jumpi  loop
endloop:
exit:
```

Computing Integer Division

Iterative C++ Version

Translate Code: Assignments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

C++
HLL

```
0:       andi   0         # AC = 0
         addi   12
         storei 1000      # a = 12 (a stored @ 1000)
         andi   0         # AC = 0
         addi   4
         storei 1004      # b = 4  (b stored @ 1004)
         andi   0         # AC = 0
         storei 1008      # result = 0 (result @ 1008)
main:    loadi  1004      # compute a – b in AC
         comp             # using 2's complement add
         addi   1
         add    1000
         brni   exit      # exit if AC negative
loop:    loadi  1000
         brni   endloop
         loadi  1004      # compute a – b in AC
         comp             # using 2's complement add
         addi   1
         add    1000      # Uses indirect bit I = 1
         storei 1000



         jump loop
endloop:
exit:
```

Easy-I
Assembly Language

Computing Integer Division

Iterative C++ Version

Translate Code: Increments

```
int a = 12;
int b = 4;
int result = 0;
main () {
  if (a >= b) {
    while (a >= 0) {
      a = a - b;
      result ++;
    }
  }
}
```

```
0:        andi   0          # AC = 0
          addi   12
          storei 1000       # a = 12 (a stored @ 1000)
          andi   0          # AC = 0
          addi   4
          storei 1004       # b = 4  (b stored @ 1004)
          andi   0          # AC = 0
          storei 1008       # result = 0 (result @ 1008)
main:     loadi  1004       # compute a – b in AC
          comp              # using 2's complement add
          addi   1
          add    1000
          brni   exit       # exit if AC negative
loop:     loadi  1000
          brni   endloop
          loadi  1004       # compute a – b in AC
          comp              # using 2's complement add
          addi   1
          add    1000       # Uses indirect bit I = 1
          storei 1000
          loadi  1008       # result = result + 1
          addi   1
          storei 1008
          jumpi  loop
endloop:
exit:
```

C++
HLL

Easy-I
Assembly Language

Computing Integer Division

Easy I Machine Code

Data

Address	Contents
1000	a
1004	b
1008	result

Challenge

Make this program as small and fast as possible

Program

Address	I Bit	Opcode (binary)	X (base 10)
0	0	00 110	0
2	0	00 111	12
4	0	00 100	1000
6	0	00 110	0
8	0	00 111	4
10	0	00 100	1004
12	0	00 110	0
14	0	00 100	1008
16	0	00 101	1004
18	0	00 000	unused
20	0	00 111	1
22	1	00 111	1000
24	0	00 010	46
26	0	00 101	1000
28	0	00 010	46
30	0	00 101	1004
32	0	00 000	unused
34	0	00 111	1
36	0	00 100	1000
38	0	00 101	1008
40	0	00 111	1
42	0	00 100	1008
44	0	00 011	26

Computing Integer Division

Iterative C++ Version

Revised Version

```
int a = 0;
int b = 4;
int result = 0;
main () {
  while (a >= b) {
    a = a - b;
    result ++;
  }
}
```

C++
HLL

Easy-I
Assembly Language

```
0:       andi   0           # AC = 0
         addi   12
         storei 1000        # a = 12 (a stored @ 1000)
         andi   0           # AC = 0
         addi   4
         storei 1004        # b = 4  (b stored @ 1004)
         andi   0           # AC = 0
         storei 1008        # result = 0 (result @ 1008)
main:
loop:    loadi  1004        # compute a – b in AC
         comp               # using 2's complement add
         addi   1
         add    1000
         brni   exit        # exit if AC negative
         loadi  1004        # compute a – b in AC
         comp               # using 2's complement add
         addi   1
         add    1000        # Uses indirect bit I = 1
         storei 1000
         loadi  1008        # result = result + 1
         addi   1
         storei 1008
         jumpi  loop
endloop:
exit:
```

Translating Conditional Expressions

```
int a = 0;
int b = 4;
int result = 0;
main () {
  while (a >= b) {
    a = a - b;
    result ++;
  }
}
```

Translating Logical Expressions

loop exit condition

~(a >= b) ? ~((a – b) >= 0)

? ((a – b) < 0)

What if Loop Exit Condition was:

~(a < b) ?

?

?

?

Computing Integer Division

Iterative C++ Version

Peephole Optimization

```
int a = 0;
int b = 4;
int result = 0;
main () {
  while (a >= b) {
    a = a - b;
    result ++;
  }
}
```

C++
HLL

Easy-I
Assembly Language

```
0:        andi   0          # AC = 0
          addi   12
          storei 1000       # a = 12 (a stored @ 1000)
          andi   0          # AC = 0
          addi   4
          storei 1004       # b = 4  (b stored @ 1004)
          andi   0          # AC = 0
          storei 1008       # result = 0 (result @ 1008)
main:
loop:     loadi  1004       # compute a – b in AC
          comp              # using 2’s complement add
          addi   1
          add    1000
          brni   exit       # exit if AC negative
          storei 1000
          loadi  1008       # result = result + 1
          addi   1
          storei 1008
          jumpi  loop
endloop:
exit:
```

The MIPS Architecture
ISA at a Glance

- Reduced Instruction Set Computer (RISC)
- 32 general purpose 32-bit registers
- Load-store architecture: Operands in registers
- Byte Addressable
- 32-bit address space

The MIPS Architecture

32 Register Set (32-bit registers)

Register #	Reg Name	Function
r0	r0	Zero constant
r4-r7	a0-a3	Function arguments
r1	at	Reserved for Operating Systems
r30	fp	Frame pointer
r28	gp	Global memory pointer
r26-r27	k0-k1	Reserved for OS Kernel
r31	ra	Function return address
r16-r23	s0-s7	Callee saved registers
r29	sp	Stack pointer
r8-r15	t0-t7	Temporary variables
r24-r25	t8-t9	Temporary variables
r2-r3	v0-v1	Function return values

The MIPS Architecture
Main Instruction Formats

Simple and uniform 32-bit 3-operand instruction formats

–R Format: Arithmetic/Logic operations on registers

opcode 6 bits	rs 5 bits	rt 5 bits	rd 5 bits	shamt 5 bits	funct 6 bits

–I Format: Branches, loads and stores

opcode 6 bits	rs 5 bits	rt 5 bits	Address/Immediate 16 bits

–J Format: Jump Instruction

opcode 6 bits	rs 5 bits	rt 5 bits	Address/Immediate 16 bits

The MIPS Architecture

Examples of <u>Native</u> Instruction Set

Instruction Group	Instruction	Function
Arithmetic/ Logic	add $s1,$s2,$s3	$s1 = $s2 + $s3
	addi $s1,$s2,K	$s1 = $s2 + K
Load/Store	lw $s1,K($s2)	$s1 = MEM[$s2+K]
	sw $s1,K($s2)	MEM[$s2+K] = $s1
Jumps and Conditional Branches	beq $s1,$s2,K	if ($s1=$s2) goto PC + 4 + K
	slt $s1,$s2,$s3	if ($s2<$s3) $s1=1 else $s1=0
	j K	goto K
Procedures	jal K	$ra = PC + 4; goto K
	jr $ra	goto $ra

The SPIM Assembler

Examples of Pseudo-Instruction Set

Instruction Group	Syntax	Translates to:
Arithmetic/ Logic	neg \$s1, \$s2	sub \$s1, \$r0, \$s2
	not \$s1, \$s2	nor \$17, \$18, \$0
Load/Store	li \$s1, K	ori \$s1, \$0, K
	la \$s1, K	lui \$at, 152 ori \$s1, \$at, -27008
	move \$s1, \$s2	
Jumps and Conditional Branches	bgt \$s1, \$s2, K	slt \$at, \$s1, \$s2 bne \$at, \$0, K
	sge \$s1, \$s2, \$s3	bne \$s3, \$s2, foo ori \$s1, \$0, 1 beq \$0, \$0, bar foo: slt \$s1, \$s3, \$s2 bar:

Pseudo Instructions: translated to native instructions by Assembler

The SPIM Assembler

Examples of <u>Assembler Directives</u>

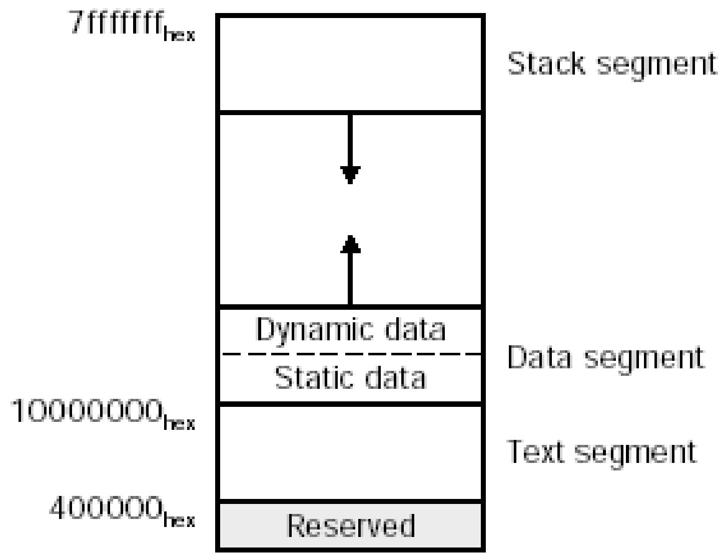
Group	Directive	Function
Memory Segmentation	.data <addr>	Data Segment starting at
	.text <addr>	Text (program) Segment
	.stack <addr>	Stack Segment
	.ktext <addr>	Kernel Text Segment
	.kdata <addr>	Kernel Data Segment
Data Allocation	x: .word <value>	Allocates 32-bit variable
	x: .byte <value>	Allocates 8-bit variable
	x: .ascii "hello"	Allocates 8-bit cell array
Other	.globl x	x is external symbol

Assembler Directives: Provide assembler additional info to generate machine code

Handy MIPS ISA References

- Appendix A: Patterson & Hennessy
- SPIM ISA Summary on class website
- Patterson & Hennessy Back Cover

The MIPS Architecture
Memory Model



Computing Integer Division

Iterative C++ Version

MIPS/SPIM Version

```
int a = 12;
int b = 4;
int result = 0;
main () {
    while (a >= b)
      a = a - b;
      result ++;
    }
  }
}
```

C++

MIPS Assembly Language

```
            .data                               # Use HLL program as a comment
x:          .word       12                      # int x = 12;
y:          .word       4                       # int y = 4;
res:        .word       0                       # int res = 0;
            .globl      main
            .text
main:       la          $s0, x                  # Allocate registers for globals
            lw          $s1, 0($s0)             #   x in $s1
            lw          $s2, 4($s0)             #   y in $s2
            lw          $s3, 8($s0)             #   res in $s3
while:      bgt         $s2, $s1, endwhile      # while (x >= y) {
            sub         $s1, $s1, $s2                       #   x = x - y;
            addi        $s3, $s3, 1             #   res ++;
            j           while                   # }
endwhile:
            la          $s0, x                  # Update variables in memory
            sw          $s1, 0($s0)
            sw          $s2, 4($s0)
            sw          $s3, 8($s0)
            jr          $ra
```

SPIM Assembler Abstractions

- Symbolic Labels
  - Instruction addresses and memory locations
- Assembler Directives
  - Memory allocation
  - Memory segments
- Pseudo-Instructions
  - Extend native instruction set without complicating arquitecture
- Macros

Implementing Procedures

- Why procedures?
  - Abstraction
  - Modularity
  - Code re-use
- Initial Goal
  - Write segments of assembly code that can be re-used, or “called” from different points in the main program.
  - KISS: Keep It Simple Stupid:
    - no parameters, no recursion, no locals, no return values

Procedure Linkage Approach I

- Problem
  – procedure must determine where to return after servicing the call
- Solution: Architecture Support
  – Add a jump instruction that saves the return address in some place known to callee
    - MIPS: jal instruction saves return address in register $ra
  – Add an instruction that can jump to return address
    - MIPS: jr instruction jumps to the address contained in its argument register

Computing Integer Division (Procedure Version)

```
int a = 0;
int b = 0;
int res = 0;
main () {
  a = 12;
  b = 5;
  res = 0;
  div();
  printf(“Res = %
}
void div(void) {
  while (a >= b) {
    a = a - b;
    res ++;
  }
}
```

C++

MIPS
Assembly Language

Fall 2002

```
            .data
x:          .word   0
y:          .word   0
res:        .word   0
pf1:        .asciiz "Result = "
pf2:        .asciiz "Remainder = "
            .globl      main
            .text
main:                                           # int main() {
                                                #   // main assumes registers sx
unused
            la          $s0, x                  #   x = 12;
            li          $s1, 12
            sw          $s1, 0($s0)
            la          $s0, y                  #   y = 5;
            li          $s2, 5
            sw          $s2, 0($s0)
            la          $s0, res    #   res = 0;
            li          $s3, 0
            sw          $s3, 0($s0)
            jal         d                       #   div();
            la          $s0, x                  #   // Must refresh registers from
memory
            li          $s1, 12
            lw          $s1, 0($s0)
            la          $s0, y                  #   y = 5;
            li          $s2, 5
            sw          $s2, 0($s0)
            la          $s0, res    #   res = 0;
            li          $s3, 0
            lw          $s3, 0($s0)
            la          $a0, pf1    #   printf("Result = %d \n");
            li          $v0, 4                  #   //system call to print_str
            syscall
            move        $a0, $s3
            li          $v0, 1                  #   //system call to print_int
            syscall
            la          $a0, pf2    #   printf("Remainder = %d \n");
            li          $v0, 4                  #   //system call to print_str
            syscall
            move        $a0, $s1
            li          $v0, 1                  #   //system call to print_int
            syscall
            jr          $ra                     #   return // TO Operating System
```

Computing Integer Division (Procedure Version)

Iterative C++ Version

```
int a = 0;
int b = 0;
int res = 0;
main () {
  a = 12;
  b = 5;
  res = 0;
  div();
  printf("Res = %d
}
void div(void) {
  while (a >= b) {
    a = a - b;
    res ++;
  }
}
```

C++

```
# div function
# PROBLEM: Must save args and registers before using them
d:                                              # void d(void) {
                                                #   // Allocate registers for globals
            la          $s0, x                  #   // x in $s1
            lw          $s1, 0($s0)
            la          $s0, y                  #   // y in $s2
            lw          $s2, 0($s0)
            la          $s0, res    #   // res in $s3
            lw          $s3, 0($s0)
while:      bgt         $s2, $s1, ewhile#   while (x <= y) {
            sub         $s1, $s1, $s2           #     x = x - y
            addi        $s3, $s3, 1 #     res ++
            j           while                   #   }
ewhile:                                         #   // Update variables in memory
            la          $s0, x
            sw          $s1, 0($s0)
            la          $s0, y
            sw          $s2, 0($s0)
            la          $s0, res
            sw          $s3, 0($s0)
enddiv:     jr          $ra                     #   return;
                                                # }
```

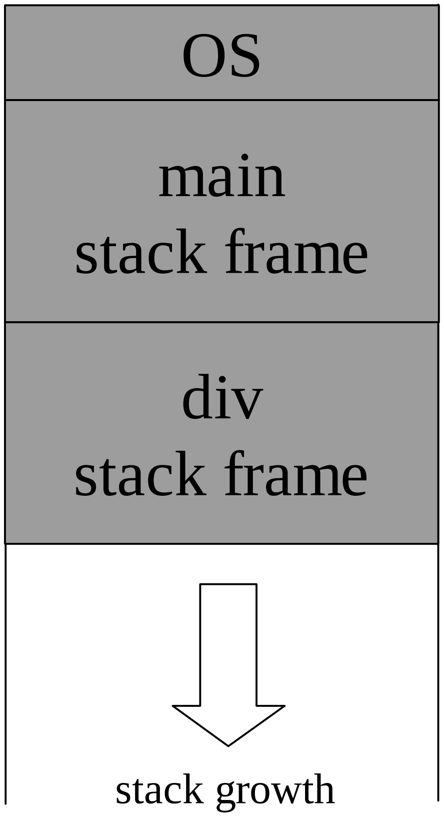
MIPS
Assembly Language

Pending Problems With Linkage Approach I

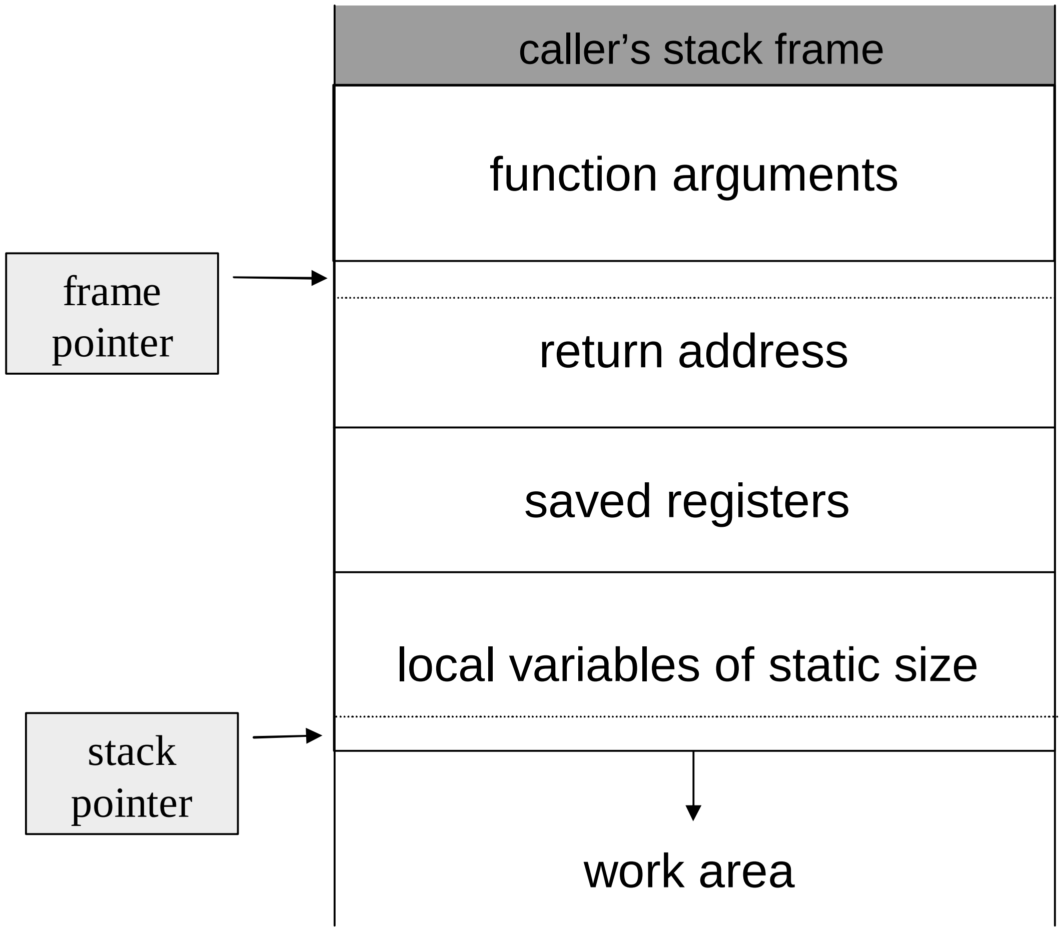
- Registers shared by all procedures
- Procedures should be able to call other procedures
- Lack of parameters forces access to globals
- Recursion requires multiple copies of local data
- Need a convention for returning function values

Solution: Use Stacks of Procedure Frames

- Stack frame contains:
  - Saved arguments
  - Saved registers
  - Return address
  - Local variables



Anatomy of a Stack Frame



Contract: Every function must leave the stack the way it found it

Example: Function Linkage using Stack Frames

```
int x = 0;
int y = 0;
int res = 0;
main () {
  x = 12;
  y = 5;
  res = div(x,y);
  printf("Res = %d",res);
}
int div(int a,int b) {
  int res = 0;
  if (a >= b) {
    res = div(a-b,b) + 1;
  }
  else {
    res = 0;
  }
  return res;
}
```

- Add return values
- Add parameters
- Add recursion
- Add local variables

MIPS: Procedure Linkage Summary

- First 4 arguments passed in \$a0-\$a3
- Other arguments passed on the stack
- Return address passed in \$ra
- Return value(s) returned in \$v0-\$v1
- Sx registers saved by callee
- Tx registers saved by caller