EE108B
Lecture
MIPS Assembly Language III

Christos Kozyrakis
Stanford University
http://eeclass.stanford.edu/ee108b

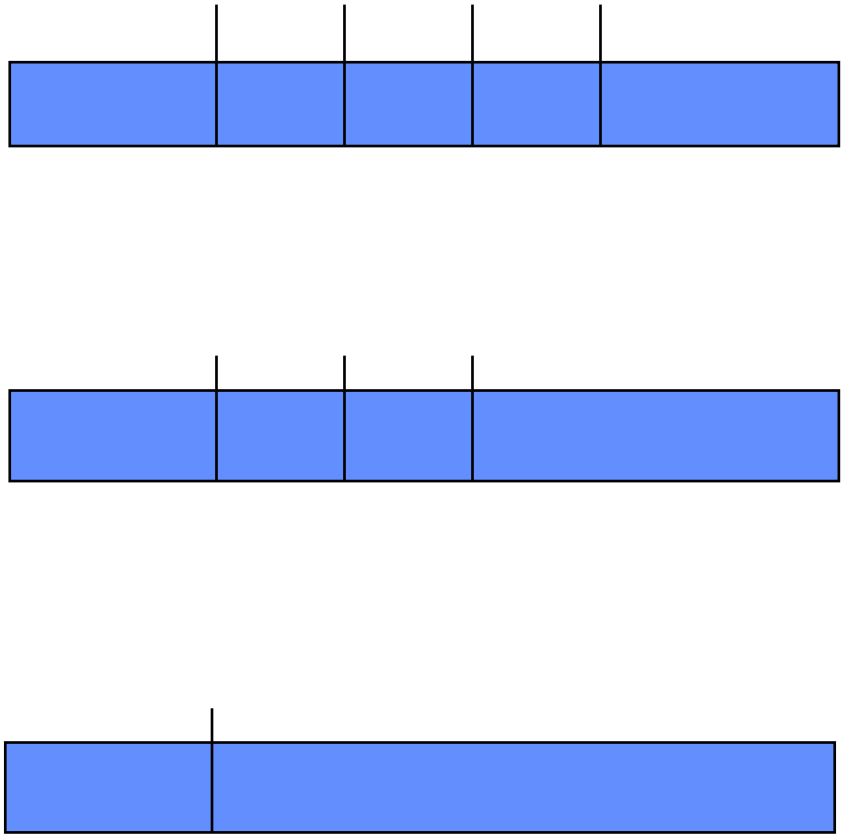
Announcements

- PA1 available, due on Thursday 2/8
  - Work on you own (no groups)
- Homework #1 due on Tue 1/23, 5pm outside Gates 310
- Lab #1 is available
  - Due on Tuesday 1/30, midnight
- EE108B Review Session
  - Friday: 2:15pm-3:05pm in Thornton 102

Review of MIPS Assembly Language II

- Memory data transfer
  - Data alignment
- Control transfer instructions
  - Branches and jumps
- Machine language
  - Encoding assembly instructions
  - MIPS instruction formats
    - R-format

MIPS Machine Instruction Review: Instruction Format Summary



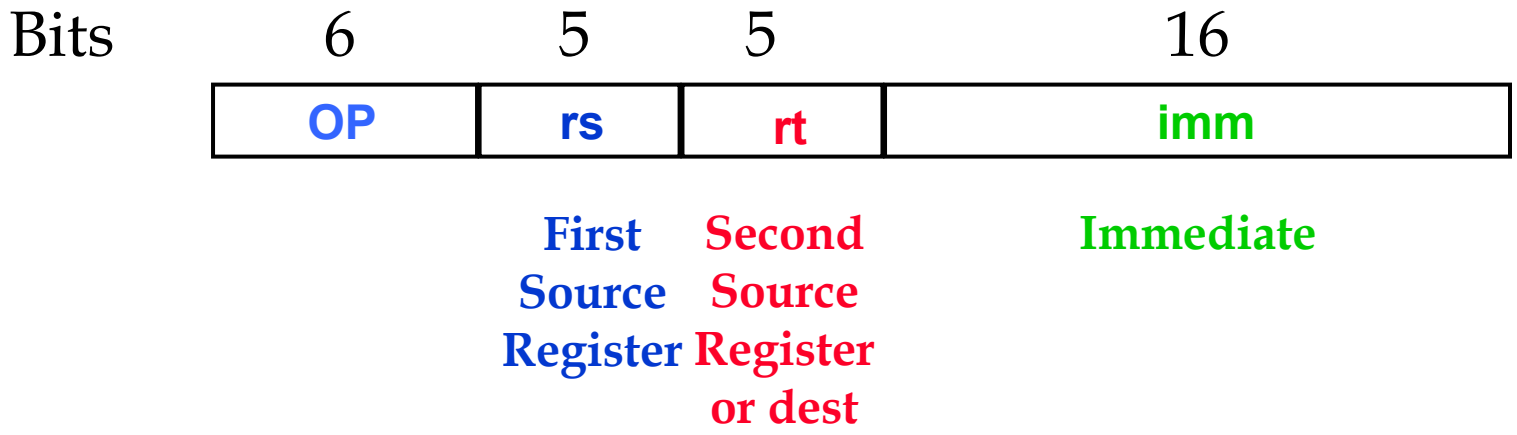
Today's Menu

- Reading 2.5, 2.7, 2.9

- I & J format
- Introduce a few more instructions
  - Logical operations
  - jal, jr
- For loops
- Switch statements
- Procedure calls

I-Format

- The immediate instruction format
  - Uses different opcodes for each instruction
  - Immediate field is signed (positive/negative constants)
  - Used for loads and stores as well as other instructions with immediates (addi, lui, etc.)
  - Also used for branches

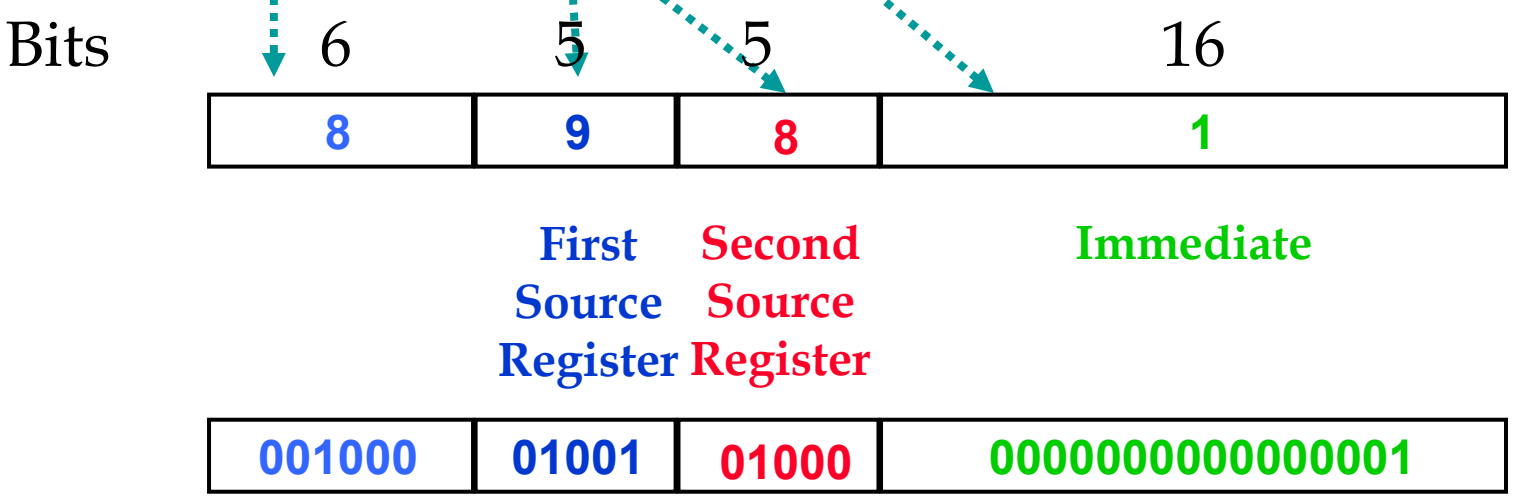


I-Format Example

- Consider the `addi` instruction

```
addi $t0, $t1, 1          # $t0 = $t1 + 1
```

- Fill in each of the fields



Another I-Format Example

- Consider the while loop

```
Loop:   addu $t0, $s0, $s0          # $t0 = 2 * i
        addu $t0, $t0, $t0          # $t0 = 4 * i
        add $t1, $t0, $s3           # $t1 = &(A[i])
        lw $t2, 0($t1)              # $t2 = A[i]
        bne $t2, $s2, Exit          # goto Exit if !=
        addu $s0, $s0, $s1          # i = i + j
        j Loop                      # goto Loop
Exit:
```

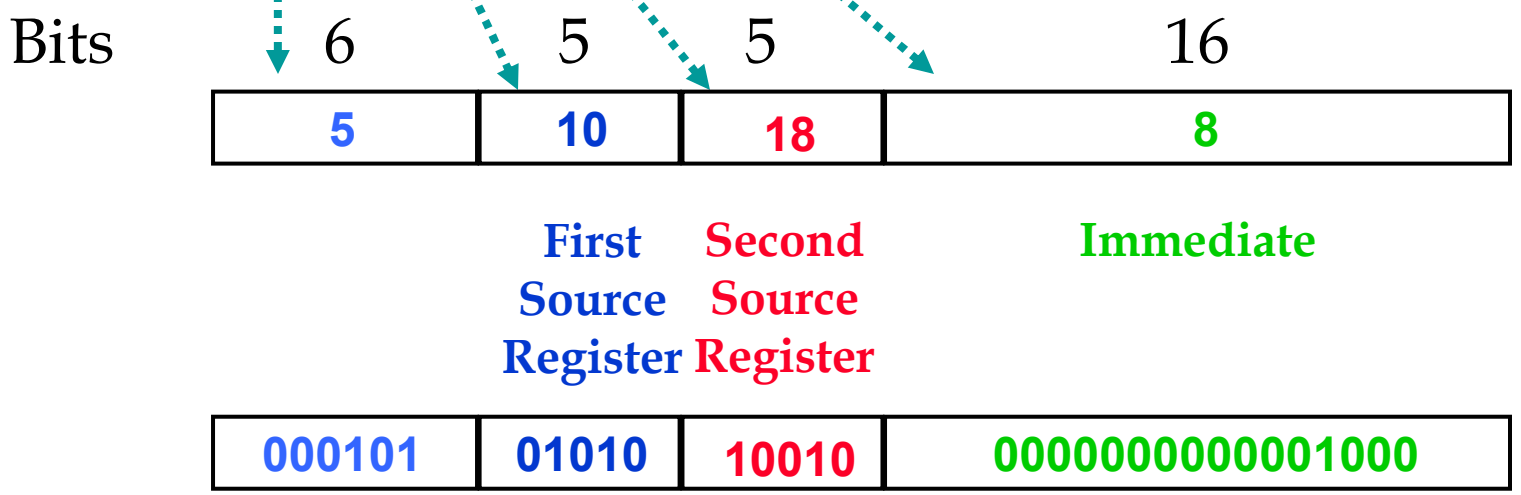
- Pretend the first instruction is located at address 80000

I-Format Example (Incorrect)

- Consider the `bne` instruction

```
bne $t2, $s2, Exit    # goto Exit if $t2 != $s2
```

- Fill in each of the fields



- This is not the optimum encoding

PC Relative Addressing

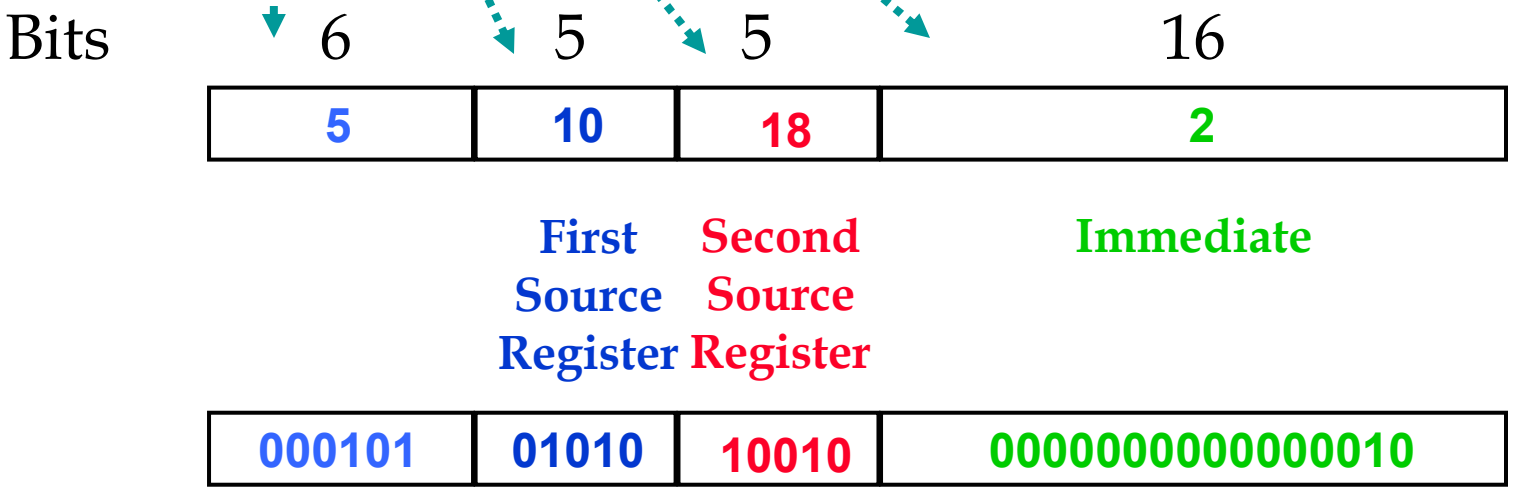
- How can we improve our use of immediate addresses when branching?
- Since instructions are always 32 bits long and word addressing requires alignment, every address must be a multiple of 4 bytes
- Therefore, we actually branch to the address that is PC + 4 + 4 $\times$ immediate

I-Format Example

- Re-consider the `bne` instruction

```
bne $t2, $s2, Exit  # goto Exit if $t2 != $s2
```

- Use PC-Relative addressing for the immediate



Branching Far Away

- If the target is greater than -2^{15} to $2^{15}-1$ words away, then the compiler inverts the condition and inserts an unconditional jump
- Consider the example where L1 is far away

```
beq $s0, $s1, L1              # goto L1 if S$0=$s1
```

- Can be rewritten as

```
        bne $s0, $s1, L2      # Inverted
        j L1                  # Unconditional jump
L2:
```

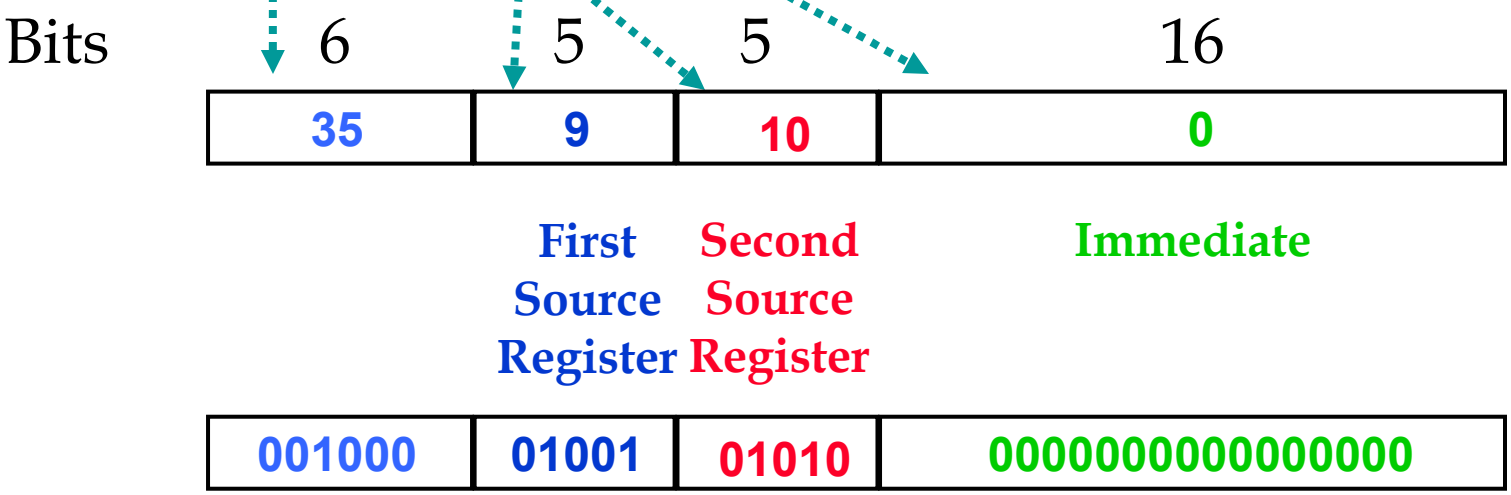
- Compiler must be careful not to cross 256 MB boundaries with jump instructions

I-Format Example: Load/Store

- Consider the `lw` instruction

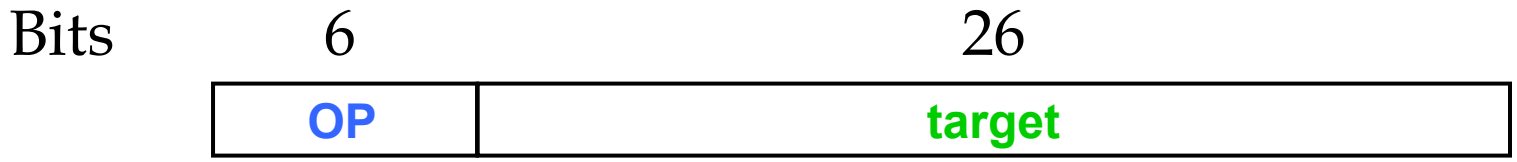
```
lw $t2, 0($t1)          # $t2 = Mem[$t1]
```

- Fill in each of the fields



J-Format

- The jump instruction format
  - Different opcodes for each instruction
  - Examples include j and jal instructions
  - Absolute addressing since long jumps are common
  - Based on word addressing (target $\times$ 4)
  - Pseudodirect addressing where 28 bits from target, and remaining 4 bits come from upper bits of PC



Jump Target Address

$$\texttt{Jump PC} = \texttt{PC}_{31..28}\texttt{||target||00}$$

```
sum_pow2:                                   # $a0 = b, $a1 = c
          addu  $a0,$a0,$a1                 # a = b + c, $a0 = a
          bltz  $a0, Exceed                 # goto Exceed if $a0 < 0
          slti  $v0,$a0,8                   # $v0 = a < 8
          beq   $v0,$zero, Exceed           # goto Exceed if $v0 == 0
          addiu $v1,$sp,8                   # $v1 = pow2 address
          sll   $v0,$a0,2                   # $v0 = a*4
          addu  $v0,$v0,$v1                 # $v0 = pow2 + a*4
          lw    $v0,0($v0)                  # $v0 = pow2[a]
          b     Return                      # goto Return
Exceed:   addu  $v0,zero,zero               # $v0 = 0

Return:   jr    ra                          # return sum_pow2
```

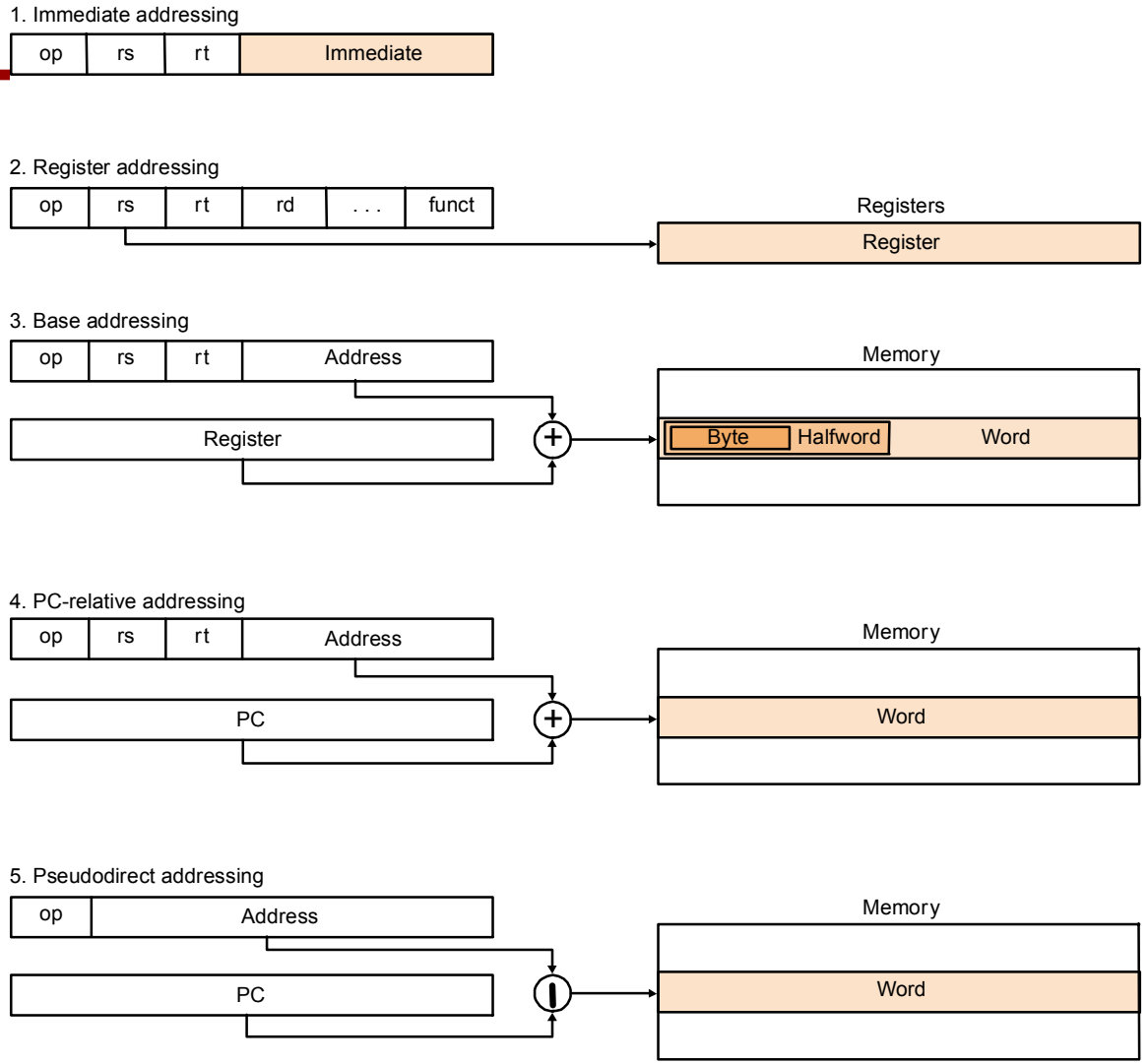
sum_pow2 Revised Machine and DisAssembly

```
sum_pow2:
0x400a98:  00 85 20 21  addu a0,a0,a1
0x400a9c:  04 80 00 08  bltz a0,0x400abc
0x400aa0:  28 82 00 06  slti v0,a0,6
0x400aa4:  10 40 00 06  beq v0,zero,0x400abc
0x400aa8:  27 a3 00 08  addiu v1,sp,8
0x400aac:  00 04 10 80  sll v0,a0,2
0x400ab0:  00 43 10 21  addu v0,v0,v1
0x400ab4:  8c 42 00 00  lw v0,0(v0)
0x400ab8:  10 00 00 01  j 0x400ac0
0x400abc:  00 00 10 21  addu v0,zero,zero
0x400ac0:  03 e0 00 08  jr ra
```

Addressing Modes Summary

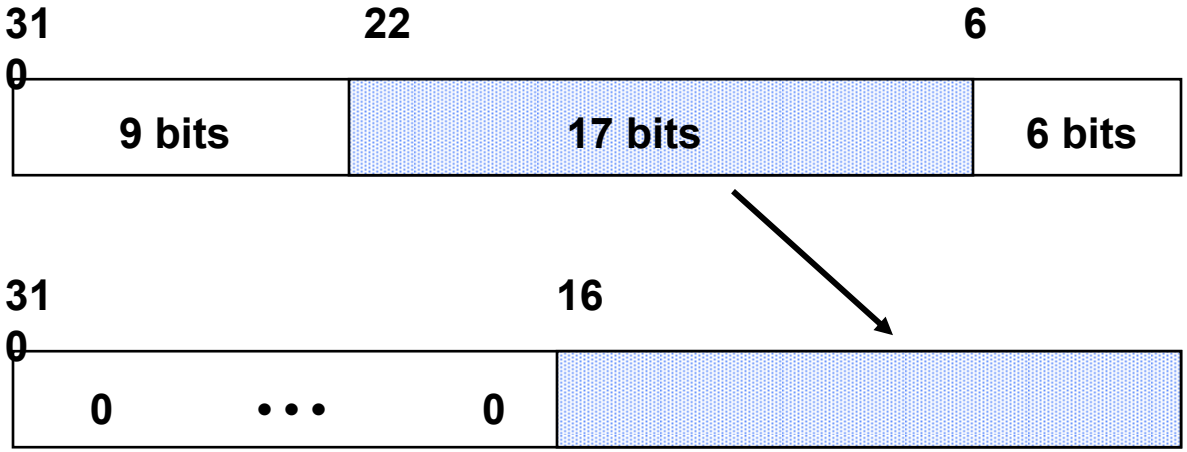
- Register addressing
  - Operand is a register (e.g. ALU)
- Base/displacement addressing (ex. load/store)
  - Operand is at the memory location that is the sum of
  - a base register + a constant
- Immediate addressing (e.g. constants)
  - Operand is a constant within the instruction itself
- PC-relative addressing (e.g. branch)
  - Address is the sum of PC and constant in instruction (e.g. branch)
- Pseudo-direct addressing (e.g. jump)
  - Target address is concatenation of field in instruction and the PC

Addressing Modes Summary



Logical Operators

- Bitwise operators often useful for bit manipulation




```
sll $t0, $t3, 9  # shift $t3 left by 9, store in $t0
srl $t0, $t0, 15 # shift $t0 right by 15
```

- Always operate unsigned except for arithmetic shifts

MIPS Logical Instructions

Instruction	Example	Meaning	Comments
and	and $1, $2, $3	$1 = $2 & $3	Logical AND
or	or $1, $2, $3	$1 = $2 | $3	Logical OR
xor	xor $1, $2, $3	$1 = $2 □ $3	Logical XOR
nor	nor $1, $2, $3	$1 = ~($2 | $3)	Logical NOR
and immediate	andi $1, $2, 10	$1 = $2 & 10	Logical AND w. constant
or immediate	ori $1, $2, 10	$1 = $2 | 10	Logical OR w. constant
xor immediate	xori $1, $2, 10	$1 = ~$2 & ~10	Logical XOR w. constant
shift left log	sll $1, $2, 10	$1 = $2 << 10	Shift left by constant
shift right log	srl $1, $2, 10	$1 = $2 >> 10	Shift right by constant
shift rt. Arith	sra $1, $2, 10	$1 = $2 >> 10	Shift rt. (sign extend)
shift left var	sllv $1, $2, $3	$1 = $2 << $3	Shift left by variable
shift right var	srlv $1, $2, $3	$1 = $2 >> $3	Shift right by variable
shift rt. arith	srav $1, $2, $3	$1 = $2 >> $3	Shift right arith. var
load upper imm	lui $1, 40	$1 = 40 << 16	Places imm in upper 16b

Loading a 32 bit Constant

- MIPS only has 16 bits of immediate value
- Could load from memory but still have to generate memory address

- Use `lui` and `ori` to load `0xdeadbeef` into $`a0`
  - **`lui $a0, 0xdead`** **`# $a0 = dead0000`**
  - **`ori $a0, $a0, 0xbeef`** **`# $a0 = deadbeef`**

For Loop Example

```
/* Compute x raised to nonnegative power p */
int ipowr_for(int x, unsigned p) {
int result;
  for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1)
      result *= x;
    x = x*x;
  }
  return result;
}
```

- Algorithm
  - Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots 2^{n-1}p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1^{\,2} \cdot (z_2^{\,2})^{\,2} \cdot \ldots \cdot (\ldots((z_{n-1}^{\,2})^{\,2})\ldots)^{\,2}$

    $z_i = 1$ **when** $p_i = 0$

    $z_i = x$ **when** $p_i = 1$

    ***n–1* times**
  - Complexity $O(\log p)$

Example

3^{10}

$= 3^2 * 3^8$

$= 3^2 * ((3^2)^{\,2})^{\,2}$

While Loop Transformation

- While loop review

```
while (Test)
    Body
```

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
```

For Loop Transformation

- Similar to while loop

```
for (Init; Test; Update)
    Body
```

```
  Init;
  goto test;
loop:
  Body
  Update ;
test:
  if (Test)
    goto loop;
```

For Example

```
int result;
for (result = 1;
     p != 0;
     p = p>>1) {
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

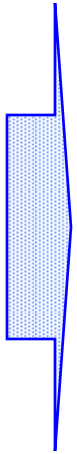
Init `result = 1`

Test `p != 0`

Update `p = p >> 1`

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```



```
  result = 1;
  goto test;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
test:
  if (p != 0)
    goto loop;
```

MIPS assembly is left as an exercise

multiply covered in Chap 3

Switch Statements

```
typedef enum {ADD, MULT, MINUS, DIV,
   MOD, BAD}  op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

- Implementation Options:

1. Series of conditional branches
   - Good if few cases
   - Slow if many

2. Jump Table
   - Lookup branch target
     - Use the MIPS jr instruction to unconditionally jump to address stored in a register
     - jr dest # Jump to $dest
   - Avoids conditional branches
   - Possible when cases are small integer constants

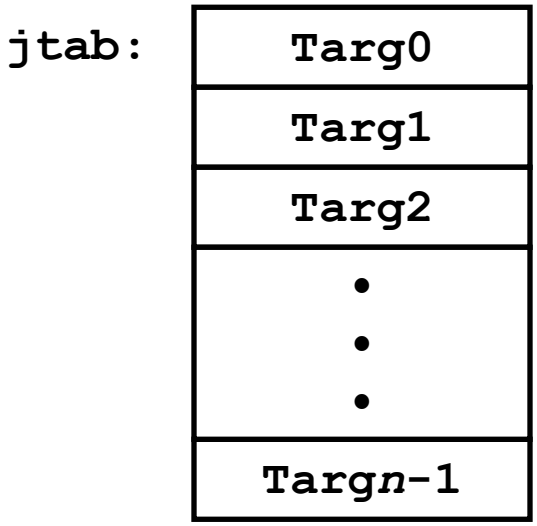
- C compiler e.g. gcc
  - Picks one based on case structure

Jump Table Structure

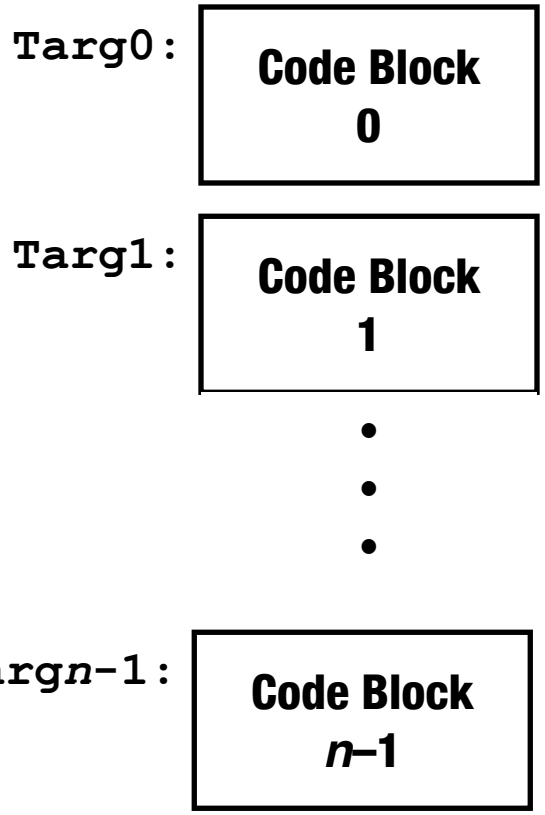
Switch Form

```
switch(op) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table



Jump Targets



Approx. Translation

```
target = JTab[op];
goto *target;
```

Switch Statement Example

```
typedef enum
  {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;
char unparse_symbol(op_type op)
{
  switch (op) {
    • • •
  }
}
```

Enumerated Values

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

Setup:

```
bltz    $a0, Exit           # if op < 0 goto Exit
slti    $t0, $a0, 6         # $t0 = 1 if op < 6
beq     $t0, $zero, Exit    # if op >= 6 goto Exit
slli    $t1, $a0, 2         # $t1 = 4 * op
add     $t2, $t1, $t4       # $t2 =&(JumpTable[op])
                            # assumes $t4=JumpTable
lw      $t3, 0($t2)         # $t3 = JumpTable[op]
jr      $t3                 # jump to JumpTable[op]
```

Jump Table

```
JumpTable:
  .word L0   #Op = 0
  .word L1   #Op = 1
  .word L2   #Op = 2
  .word L3   #Op = 3
  .word L4   #Op = 4
  .word L5   #Op = 5
```

Advantage of Jump Table

Can do ***k***-way branch

Targets

```
L0:
    ori $v0, $zero, 43    # '+'
    j Exit
L1:
    ori $v0, $zero, 42    # '*'
    j Exit
L2:
    ori $v0, $zero, 45    # '-'
    j Exit
L3:
    ori $v0, $zero, 47    # '/'
    j Exit
L4:
    ori $v0, $zero, 37    # '%'
    j Exit
L5:
    ori $v0, $zero, 63    # '?'
Exit:                     # end of switch
```

Procedure Call and Return

- Procedures are required for structured programming
  - Aka: functions, methods, subroutines, …
- Implementing procedures in assembly requires several things to be done
  - Memory space must be set aside for local variables
  - Arguments must be passed in and return values passed out
  - Execution must continue after the call
- Procedure Steps
  1. Place parameters in a place where the procedure can access them
  2. Transfer control to the procedure
  3. Acquire the storage resources needed for the procedure
  4. Perform the desired task
  5. Place the result value in a place where the calling program can access it
  6. Return control to the point of origin

Call and Return

- To jump to a procedure, use the `jal` or `jalr` instructions

```
jal target          # Jump and link to label
jalr $dest          # Jump and link to $dest
```

- Jump and link
  - The *program counter* (PC) stores the address of the currently executing instruction
  - The "jump and link" instructions stores the next instruction address in `$ra` before transferring control to the target/destination
  - Therefore, the address stored in `$ra` is PC + 4
- To return, use the `jr` instruction

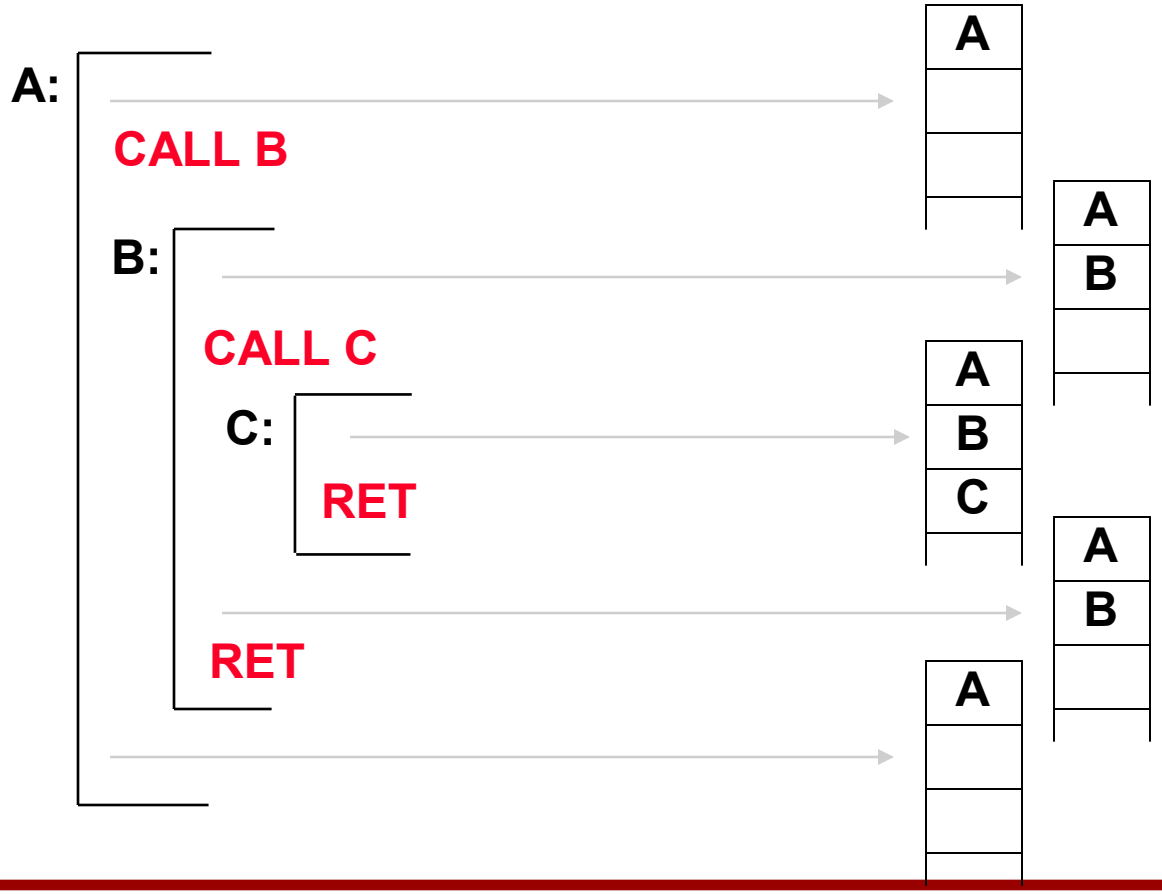
```
jr $ra
```

Stack-Based Languages

- Languages that Support Recursion
  - e.g., C, Java, Pascal, …
  - Code must be "Reentrant"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments, local variables, return pointer
- Stack Discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
  - LIFO
- Stack Allocated in Frames
  - State for single procedure instantiation

Nested Stacks

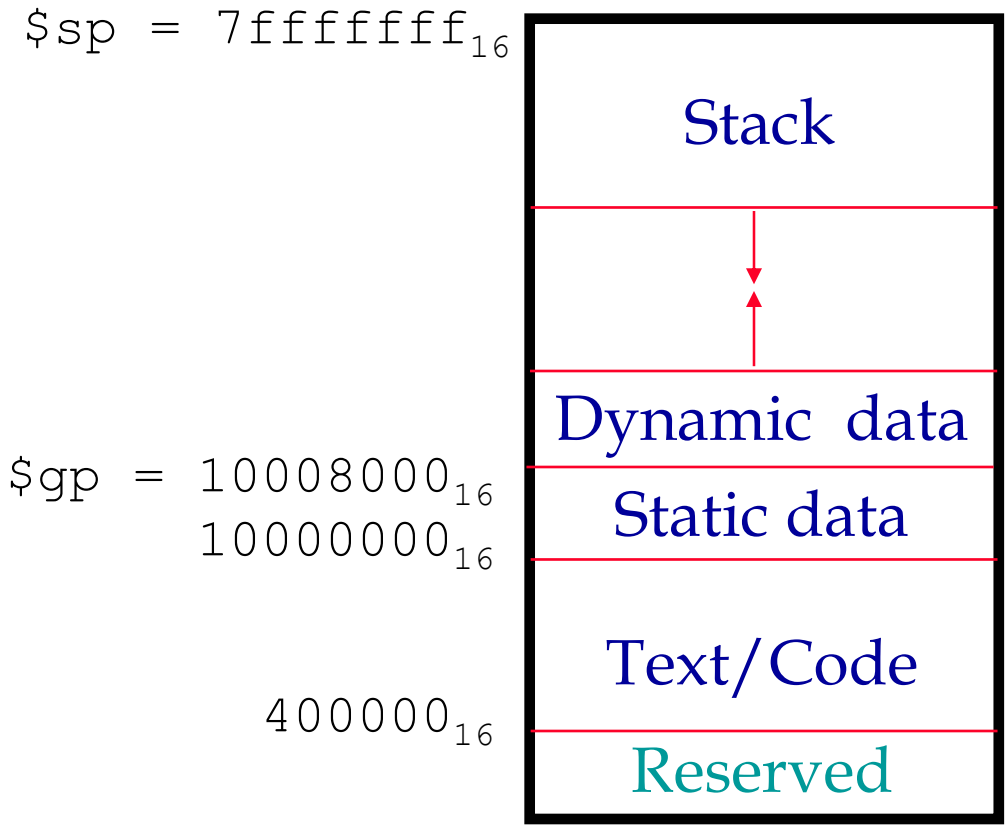
- The stack grows downward and shrinks upward



Stacks

- Data is pushed onto the stack to store it and popped from the stack when not longer needed
  - MIPS does not support in hardware (use loads/stores)
  - Procedure calling convention requires one
- Calling convention
  - Common rules across procedures required
  - Recent machines are set by software convention and earlier machines by hardware instructions
- Using Stacks
  - Stacks can grow up or down
  - Stack grows down in MIPS
  - Entire stack frame is pushed and popped, rather than single elements

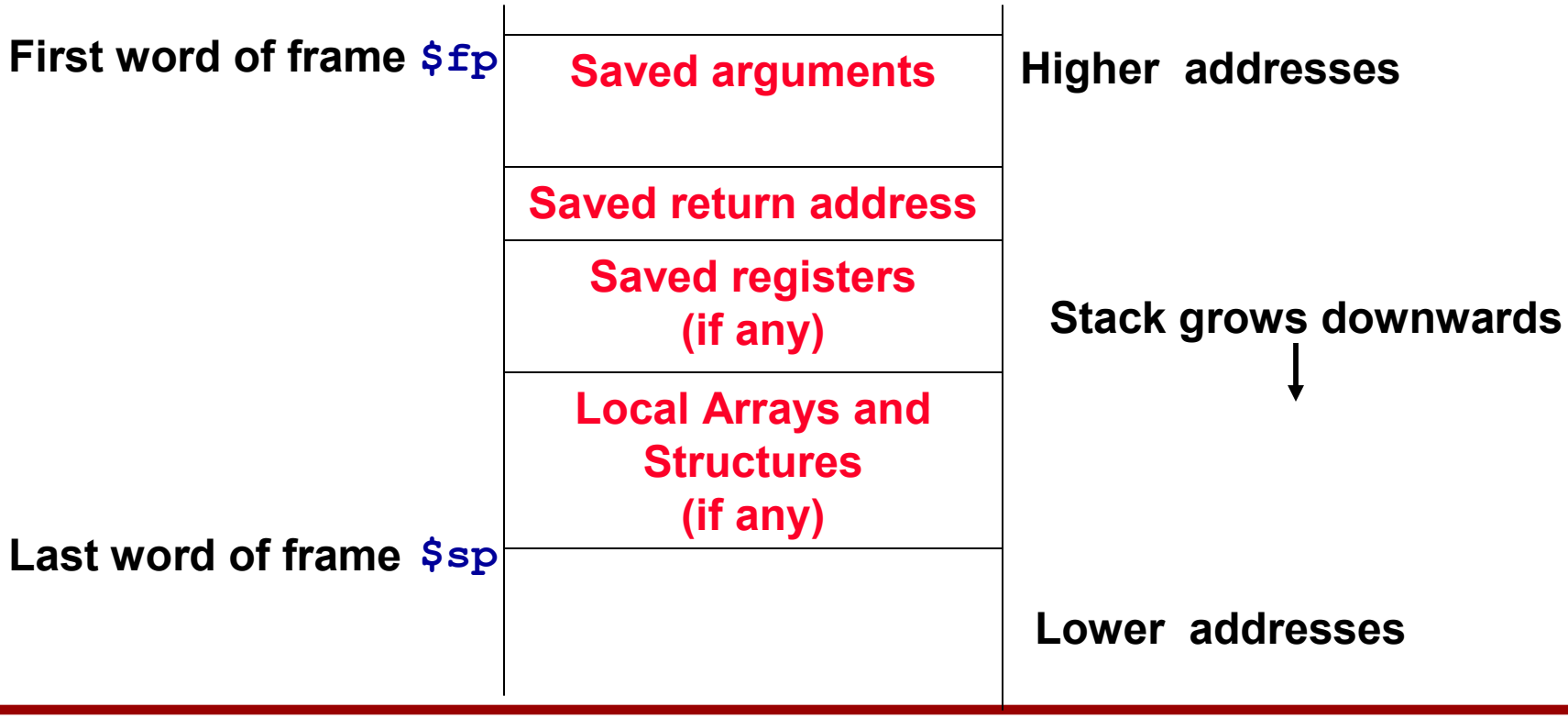
MIPS Storage Layout



Stack and dynamic area grow towards one another to maximize storage use before collision

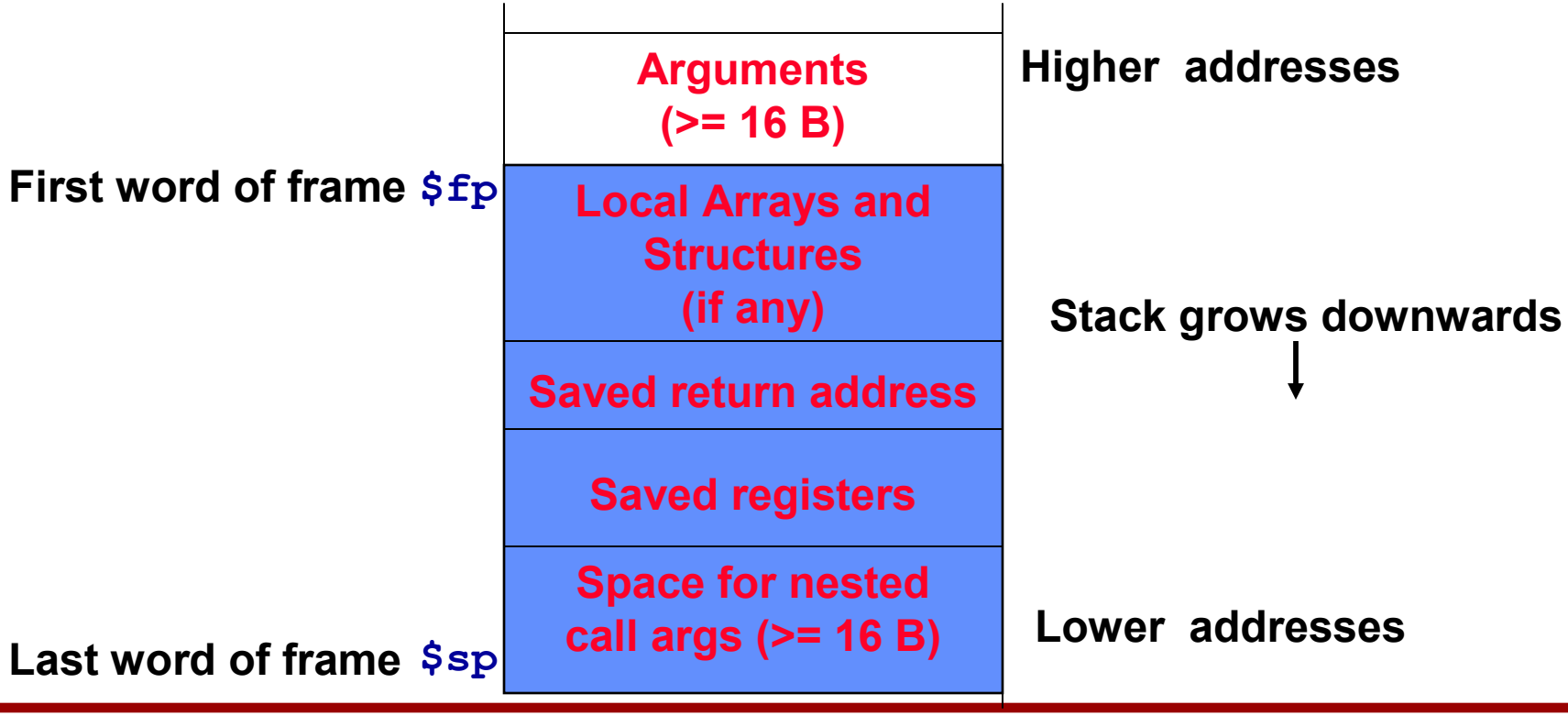
Procedure Activation Record (Frame) textbook

- Each procedure creates an activation record on the stack



Procedure Activation Record (Frame) SGI/GCC Compiler

- Each procedure creates an activation record on the stack
  - At least 32 bytes by convention to allow for $a0-$a3, $ra, $fp and be double double word aligned (16 byte multiple)



Register Assignments

- Use the following calling conventions

Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

Register Assignments (cont)

0	$zero	Zero constant (0)
1	$at	Reserved for assembler
2	$v0	Expression results
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Caller saved
. . .		
15	$t7	

16	$s0	Callee saved
. . .		
23	$s7	
24	$t8	Temporary (cont'd)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Pointer to global area
29	$sp	Stack Pointer
30	$fp	Frame Pointer
31	$ra	Return address

Caller vs. Callee Saved Registers

Preserved	Not Preserved
Saved registers ($s0-$s7)	Temporary registers ($t0-$t9)
Stack/frame pointer ($sp, $fp, $gp)	Argument registers ($a0-$a3)
Return address ($ra)	Return values ($v0-$v1)

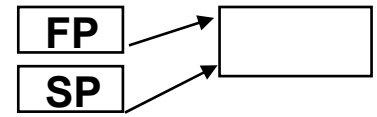
- Preserved registers (Callee Save)
  - Save register values on stack prior to use
  - Restore registers before return
- Not preserved registers (Caller Save)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

Call and Return

- Caller
  - Save caller-saved registers `$a0`-`$a3`, `$t0`-`$t9`
  - Load arguments in `$a0`-`$a3`, rest passed on stack
  - Execute `jal` instruction
- Callee Setup
  1. Allocate memory for new frame (`$sp` = `$sp` - frame)
  2. Save callee-saved registers `$s0`-`$s7`, `$fp`, `$ra`
  3. Set frame pointer (`$fp` = `$sp` + frame size - 4)
- Callee Return
  - Place return value in `$v0` and `$v1`
  - Restore any callee-saved registers
  - Pop stack (`$sp` = `$sp` + frame size)
  - Return by `jr $ra`

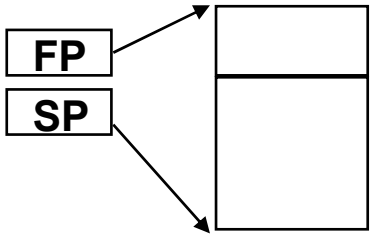
Calling Convention Steps

Before call:



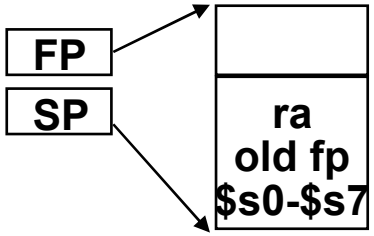
First four arguments passed in registers

Callee setup; step 1



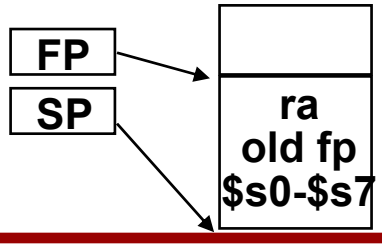
Adjust SP

Callee setup; step 2



Save registers as needed

Callee setup; step 3



Adjust FP

Simple Example

```
int foo(int num)
{
   return(bar(num + 1));
}

int bar(int num)
{
   return(num + 1);
}
```

```
foo:
        addiu   $sp, $sp, -32   # push frame
        sw      $ra, 28($sp)    # Store $ra
        sw      $fp, 24($sp)    # Store $fp
        addiu   $fp, $sp, 28    # Set new $fp
        addiu   $a0, $a0, 1     # num + 1
        jal     bar             # call bar
        lw      $fp, 24($sp)    # Load $fp
        lw      $ra, 28($sp)    # Load $ra
        addiu   $sp, $sp, 32    # pop frame
        jr      $ra             # return

bar:
        addiu   $v0, $a0, 1     # leaf procedure
        jr      $ra             # with no frame
```

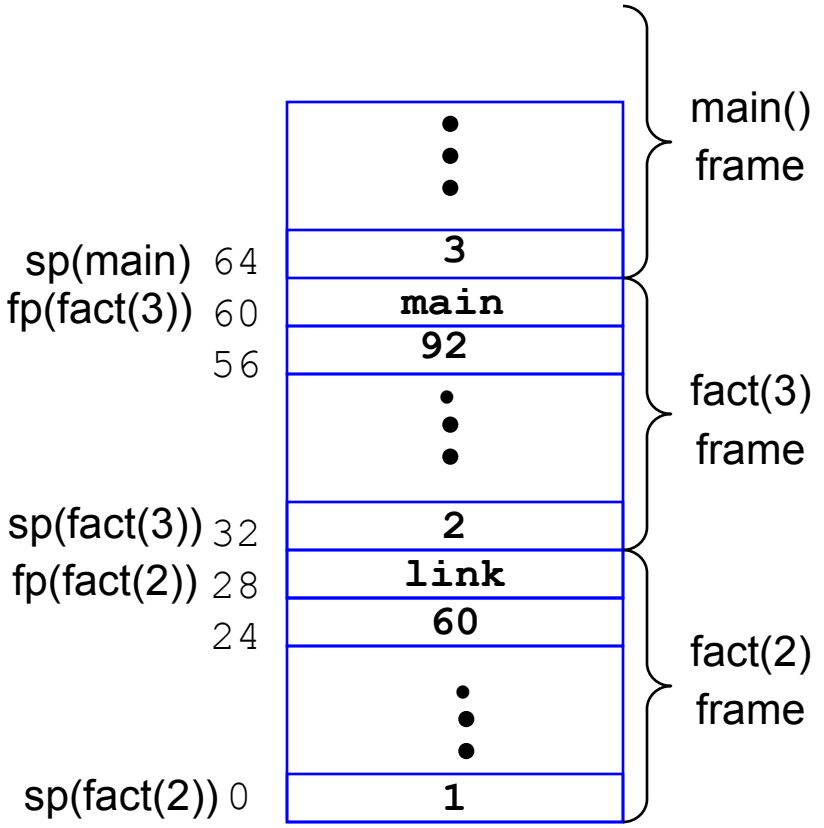
Factorial Example

```
int fact(int n)
{
   if (n <= 1)
      return(1);
   else
      return(n*fact(n-1));
}
```

```
fact:   slti   $t0, $a0, 2        # a0 < 2
        beq    $t0, $zero, skip   # goto skip
        ori    $v0, $zero, 1      # Return 1
        jr     $ra                # Return
skip:   addiu  $sp, $sp, -32      # $sp down 32
        sw     $ra, 28($sp)       # Save $ra
        sw     $fp, 24($sp)       # Save $fp
        addiu  $fp, $sp, 28       # Set up $fp
        sw     $a0, 32($sp)       # Save n
        addui  $a0, $a0, -1       # n - 1
        jal    fact               # Call recursive
link:   lw     $a0, 32($sp)       # Restore n
        mul    $v0, $v0, $a0      # n * fact(n-1)
        lw     $ra, 28($sp)       # Load $ra
        lw     $fp, 24($sp)       # Load $fp
        addiu  $sp, $sp, 32       # Pop stack
        jr     $ra                # Return
```

Running Factorial Example

```
main()
{
    fact(3);
}
```



Stack just before call of `fact(1)`

Factorial Again
Tail Recursion

```
int fact_t (int n, int val)
{
  if (n <= 1)
    return val;
  return
    fact_t(n-1, val*n);
}
```

```
fact_t(n, val)
{
   • • •
  return
   fact_t(Nexpr, Vexpr)
}
```

```
int fact(int n)
{
  return fact_t(n, 1);
}
```

- Form
  - Directly return value returned by recursive call
- Consequence
  - Can convert into loop

Removing Tail Recursion

```
fact_t(n, val)
{
 start:
   • • •
  val = Vexpr;
  n = Nexpr;
  goto start;
}
```

```
int fact_t (int n, int val)
{
 start:
  if (n <= 1)
    return val;
  val = val*n;
  n = n-1;
  goto start;
}
```

- Effect of Optimization
  - Turn recursive chain into single procedure (leaf)
  - Eliminate procedure call overhead
  - No stack frame needed
  - Constant space requirement
    - Vs. linear for recursive version

Assembly Code for fact_t

```
int fact_t (int n, int val)
{
 start:
  if (n <= 1)
    return val;
  val = val*n;
  n = n-1;
  goto start;
}
```

```
fact_t: addui  $v0, $a1, $zero    # $v0 = $a1
start:  slti   $t0, $a0, 2        # n < 2
        beq    $t0, $zero, skip   # if n >= 2 goto skip
        jr     $ra                # return
skip:   mult   $v0, $v0, $a0      # val = val * n
        addiu  $a0, $a0, -1       # n = n - 1
        j      start              # goto start
```

Pseudoinstructions

- Assembler expands *pseudoinstructions*

```
move $t0, $t1                # Copy $t1 to $t0

addu $t0, $zero, $t1         # Actual instruction
```

- Some pseudoinstructions need a temporary register
  - Cannot use `$t`, `$s`, etc. since they may be in use
  - The `$at` register is reserved for the assembler

```
blt $t0, $t1, L1     # Goto L1 if $t0 < $t1

slt $at, $t0, $t1    # Set $at = 1 if $t0 < $t1
bne $at, $zero, L1   # Goto L1 if $at != 0
```