CMPE110 Lecture 06
Instruction Set Architecture III

Heiner Litz

https://canvas.ucsc.edu/courses/12652

Announcements

Review

RISC-V Constants

Often want to specify a constant operand in the instruction

Constant == immediate == literal == offset

Use the `addi` instruction

`addi dst, src1, immediate`

The immediate is a 12 bit signed value between -2^{11} and $2^{11}-1$

Example:

```
C: a++;
RISC-V: addi x1, x1, 1    # a = a + 1
```

Data Transfer Instructions: Loads

Data transfer instructions have three parts

- Operator name (defines the transfer size as well)
- Destination register
- Base register address and constant offset

```
ld dst, offset(base)
```

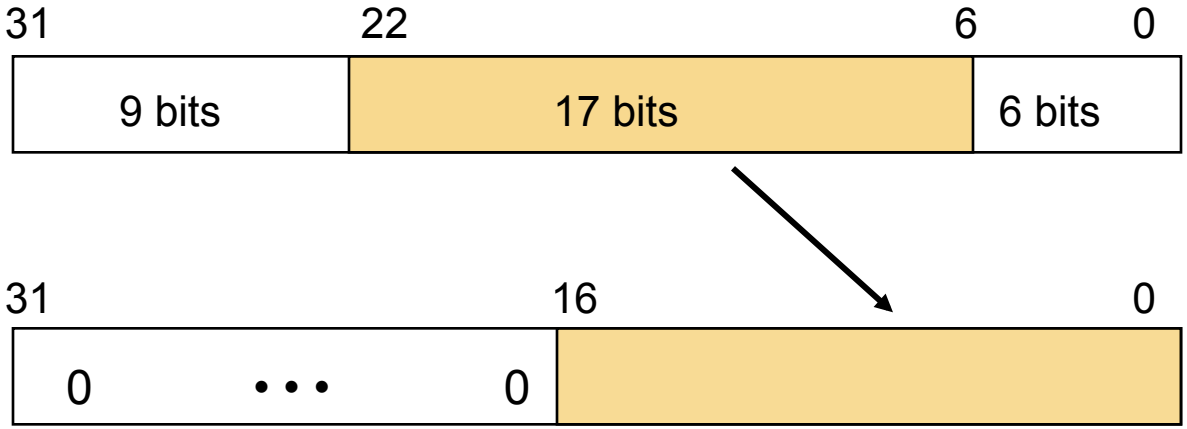
Offset value is a 12-bit uunsigned constant (immediate)

Instruction Set Architecture III

- Extra tricks
- Changing flow control
- Procedure calls

Extra tricks: Logical Operators

Bitwise operators often useful for bit manipulation




```
sll xt0, xt3, 9  # shift xt3 left by 9, store in xt0
srl xt0, xt0, 15 # shift xt0 right by 15
```

Tip: always operate unsigned except for arithmetic shifts

Extra Tricks: Loading a 32 bit Constant

RISC-V only has 12 bits of immediate value

- Could load from memory
- But still have to generate memory address

Use `lui` and `ori` to load `0xdeadbeef` into `xa0`

```
lui xa0, 0xdeadb            # xa0 = 0xdeadb000
ori xa0, xa0, 0xeef         # xa0 = 0xdeadbeef
```

Changing Flow Control

Changing Control Flow

Programs have complex flow control

- If-then-else and case statements
- Loops (if, while, …)
- Function/method/procedure calls

Control flow operations: two types

- Conditional branch instructions are known as branches
- Unconditional changes in the control flow are called jumps

The target of the branch/jump is a label

- Label identifies a location in a program

RISC-V Conditional Branch

The simplest conditional test is the `beq` instruction for equality

```
beq reg1, reg2, label
```

Consider the code

```
        if (a == b) goto L1;
        a = a + c;
L1:     // Continue
```

Use the `beq` instruction

```
        beq x1, x2, label  (label is an immediate added to PC)
        add x1, x1, x2
        …
L1:     # Continue
```

RISC-V Unconditional Jump

The `j` instruction jumps to a label

```
j label
```

This is essentially a goto statement

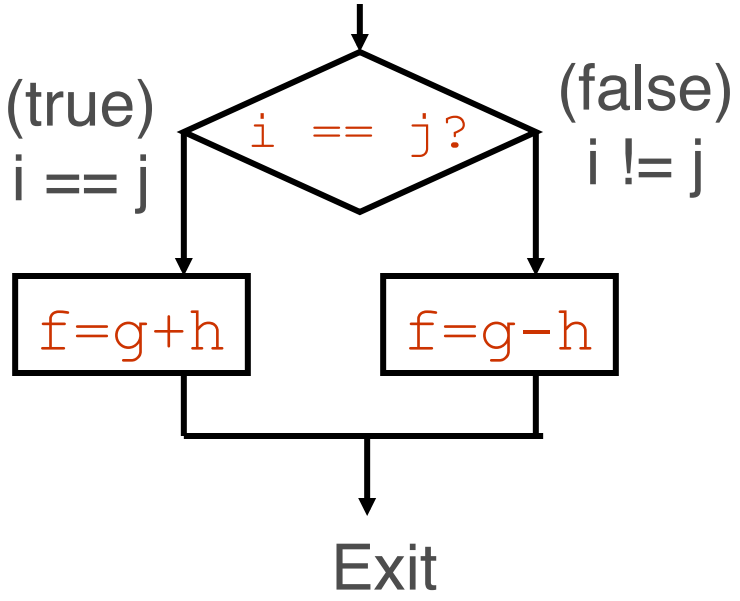
PC relative addressing (what is a label?)

- Labels have to be 0s and 1s as well ☺
- Assembler translates labels into immediate
- Immediate is added to PC
- Enables jump within (+/-) 2^18 of current PC

If-then-else Example

Consider the code

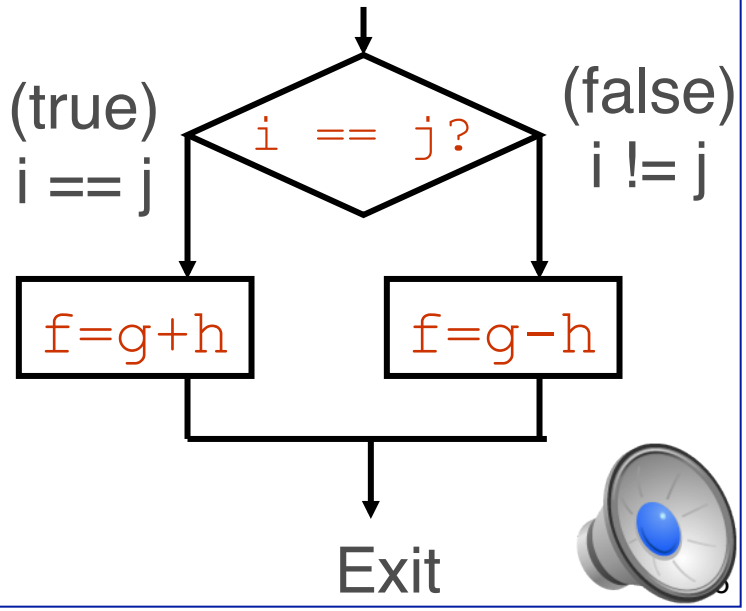
```
if (i == j) f = g + h;
else f = g - h;
```



If-then-else Example

Create labels and use equality instruction

```
        beq x3, x4, True    # Branch if i == j
False:  subu x0, x1, x2     # f = g - h
        j Exit              # Go to Exit
True:   add x0, x1, x2      # f = g + h
Exit:
```



RISC-V Jumps & Branches

Instruction	Example	Meaning
jump	j L	goto L
jump register	jr x1	goto value in x1
jump and link	jal L	goto L and set xra
jump and link register	jalr x1	goto x1 and set xra

xra = return address register

branch equal	beq x1, x2, L	if (x1 == xs2) goto L
branch not eq	bne x1, x2, L	if (x1 != x2) goto L
branch l.t. 0	bltz x1, L	if (x1 < 0) goto L
branch l.t./eq 0	blez x1, L	if (x1 <= 0) goto L
branch g.t. 0	bgtz x1, L	if (x1 > 0) goto L
branch g.t./eq 0	bgez x1, L	if (x1 >= 0) goto L

Branch on Other Comparisons

How do we branch on other comparisons?

<, >, ≤, ≥

Two instruction solution

A comparison instruction with binary outputs (e.g., slt = set less than)

An equality/inequality branch on the comparison output

Consider the following C code

```
if (f < g) goto Less;
```

Solution

```
slt xt0, xs0, xs1          # xt0 = 1 if xs0 < xs1
bne xt0, xzero, Less       # Goto Less if xt0 != 0
```

RISC-V Comparisons

Instruction	Example	Meaning	Comments
set less than	`slt x1, x2, x3`	x1 = (x2 < x3)	comp less than signed
set less than imm	`slti x1, x2, 100`	x1 = (x2 < 100)	comp w/const signed
set less than uns	`sltu x1, x2, x3`	x1 = (x2 < x3)	comp < unsigned
set l.t. imm. uns	`sltiu x1, x2, 100`	x1 = (x2 < 100)	comp < const unsigned

A < B = B > A
!(A < B) = A ≥ B

C

```
if (a < 8)
```

Assembly

```
slti xv0,xa0,8              # xv0 = a < 8
beq  xv0,xzero, Exceed      # goto Exceed if xv0 == 0
```

While loop in C

Consider a `while` loop

```
while (A[i] == k)
        i = i + j;
```

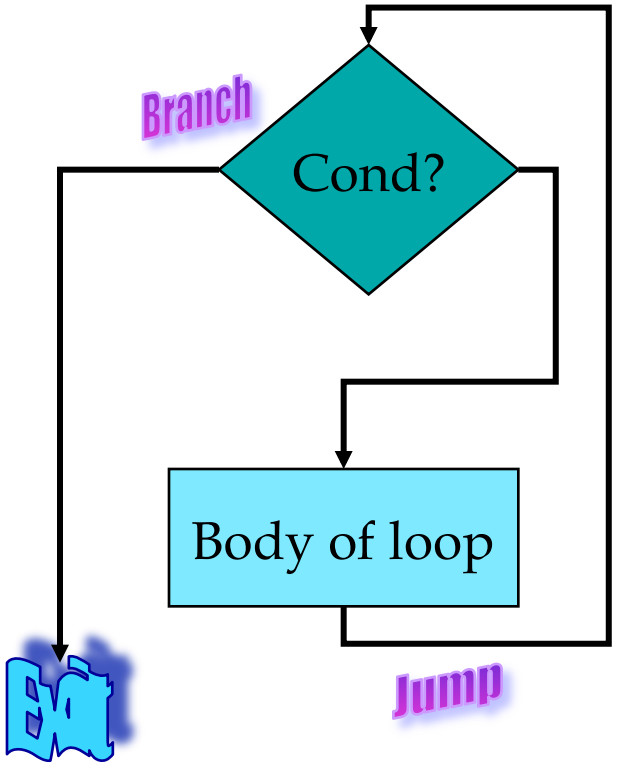
Any idea?

assume `i in x0, j in x1, k in x2, &A[0] in x3`

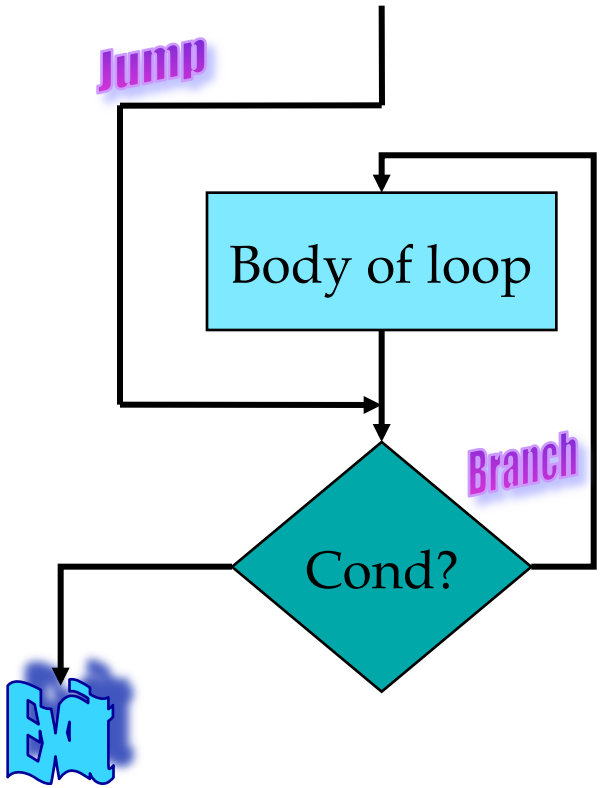
```
Loop:  sll xt0, x0, 2          # xt0 = 4 * i
       addu xt1, xt0, x3       # xt1 = &(A[i])
       lw xt2, 0(xt1)          # xt2 = A[i]
       bne xt2, x2, Exit       # goto Exit if !=
       addu x0, x0, x1         # i = i + j
       j Loop                  # goto Loop
Exit:
```

Improve Loop Efficiency

Two branches/iteration:



Better structure:



Improved Loop Solution

Remove extra jump from loop body

```
        j Cond                  # goto Cond
Loop:   addu xs0, xs0, xs1      # i = i + j
Cond:   sll xt0, xs0, 2         # xt0 = 4 * i
        addu xt1, xt0, xs3      # xt1 = &(A[i])
        lw xt2, 0(xt1)          # xt2 = A[i]
        beq xt2, xs2, Loop      # goto Loop if ==
Exit:
```

Reduced loop from 6 to 5 instructions

Even small improvements important if loop executes many tim

For Loops?

- How do you implement a for loop?

While Loop


```
while (Test)
    Body
```

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

For Loop


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

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

For Loop Example


```
/* Sum an array of numbers a of length n */
int sum_array(int *a, int n) {
  int result = 0;
  int i ;
  for (i=0; i<n; i++) {
    result += a[i];
  }
  return result;
}
```

For Loop Example


```
/* Sum an array of numbers a
of length n */
int sum_array(int *a, int n)
{
  int result = 0;
  int i ;
  for (i=0; i<n; i++) {
    result += a[i];
  }
  return result;
}
```

```
/* Sum an array of numbers */
int sum_array(int *a, int n) {
  int result = 0;
  int i ;
  i=0 ;
  goto test ;
loop:
  result += a[i] ;
  i++ ;
test:
  if(i<n) goto loop ;

  return result;
}
```

For Loop Example


```
# xa0=a, xa1=n, xv0=result
# xt0=i
sum_array:
   add xv0,x0,x0    # result = 0 ;
   add xt0,x0,x0    # i=0 ;
   j test ;
loop:
   sll xt1,xt0,2    # xt1=i*4
   add xt1,xa0,xt1  # &(a[i])
   lw  xt1,0(xt1)   # a[i]
   add xv0,xt1,xv0  # result += a[i]
   addi xt0,xt0,1   # i++
test:
   slt xt1,xt0,xa1  # xt1 = (i<n)
   bne xt1,x0,loop  # if(i<n) goto loop
```

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 to lookup branch target
   - Use jr to unconditionally jump to address stored in a register
     - jr dest # Jump to xdest
   - Avoids conditional branches
   - Possible when cases are small integer constants and dense

- C compiler picks 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

jtab:

Targ0
Targ1
Targ2
• • •
Targn-1

Jump Targets

Targ0: Code Block 0

Targ1: Code Block 1

• • •

Targn-1: Code Block *n*–1

Implementation

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

Switch Statement Code


```
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**

```
        slt    xt0, xa0, x0        # xt0 = 1 if op < 0
        bne    xt0, x0, Exit       # if op < 0 goto Exit
        slti   xt0, xa0, 6         # xt0 = 1 if op < 6
        beq    xt0, x0, Exit       # if op >= 6 goto Exit
        slli   xt1, xa0, 2         # xt1 = 4 * op
        add    xt2, xt1, xt4       # xt2 =&(JumpTable[op])
                                   # assumes xt4=JumpTable
        lw     xt3, 0(xt2)         # xt3 = JumpTable[op]
        jr     xt3                 # jump to JumpTable[op]
```

Homework: C Example

Simple C procedure: sum_pow2(b,c) = 2^{b+c}

```
1:int sum_pow2(int b, int c)
2:{
3:    int pow2 [8] = {1, 2, 4, 8, 16, 32, 64, 128};
4:    int  a, ret;
5:    a = b + c;
6:    if (a < 8)
7:        ret = pow2[a];
8:    else
9:        ret = 0;
10:   return(ret);
11:}
```

Homework: sum_pow2 Assembly


```
sum_pow2:                               # xa0 = b, xa1 = c
        addu  xa0,xa0,xa1               # a = b + c, xa0 = a
        slti  xt0,xa0,8                 # xt0 = a < 8
        beq   xt0,xzero, Exceed         # goto Exceed if xt0 == 0
        addiu xt1,xsp,8                 # xt1 = pow2 address
        sll   xt0,xa0,2                 # xt0 = a*4
        addu  xt0,xt0,xt1               # xt0 = pow2 + a*4
        lw    xt0,0(xt0)                # xt0 = pow2[a]
        j     Return                    # goto Return
Exceed: addu   xt0,xzero,xzero          # xt0 = 0

Return: jr ra                           # return sum_pow2
```

Branching Far Away

If the target is more than -2^{11} to $2^{11}-1$ words away, then the compiler inverts condition and inserts an unconditional jump

Consider the example where L1 is far away

```
beq xs0, xs1, L1          # goto L1 if Sx0=xs1
```

Can be rewritten as

```
        bne xs0, xs1, L2  # Inverted
        j L1              # Unconditional jump
L2:
```

Compiler must be careful not to cross 1 GB boundaries with jump instructions

More on this in a couple of slides

Procedure Call and Return

- Procedures are required for structured programming
  - Aka: functions, methods, subroutines, …
- Implementing procedures in assembly requires several things
  - 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
- Just a single new instruction needed
  - A variation of jump (jump and link)
  - The rest is software (using instructions we already know)

Procedure Call Steps

1. Place parameters in a place where the procedure can access them
2. Transfer control to the procedure
   - Some kind of jump
3. Allocate the memory resources needed for the procedure
4. Perform the desired task (procedure body)
5. Place the result value in a place where the calling program can access it
6. Free the memory allocated in (3)
7. Return control to the point of origin

Call and Return Implementation

- Call: jump & link instruction (`jal target`)
  - Store PC+4 in register $31 ($ra)
  - Jump to target
- Return: jr $ra
- Arguments: $a0 - $a3
  - By software convention
- Return values: $v0, $v1
  - By software convention

Complications

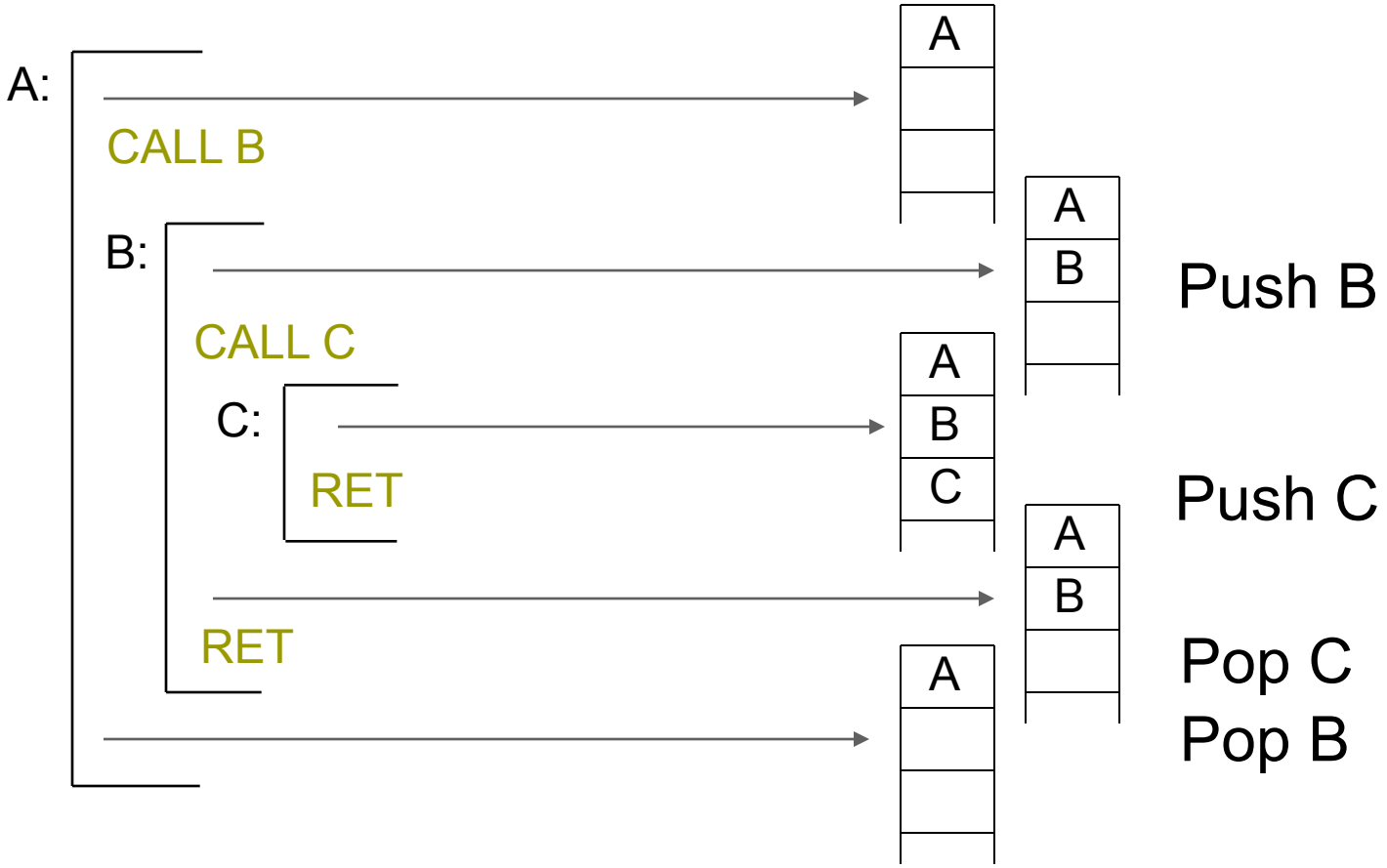
- What if function has >4 arguments?
  - Or an unknown number of arguments
- What if function returns >8 bytes?
- What if the function calls another function
  - A() → B() → C() →
  - Recursion: A() → A() → A() →

Stack-Based Languages

- Languages that support recursion (C, C++, Java, …)
  - Code must be “reentrant”
    - Multiple simultaneous instantiations of single procedure
  - Need someplace to store state of each instantiation
    - Arguments, local variables, return pointer
- Stack discipline (last in, first out)
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does → LIFO
    - Stack allocated in frames

Nested Stacks

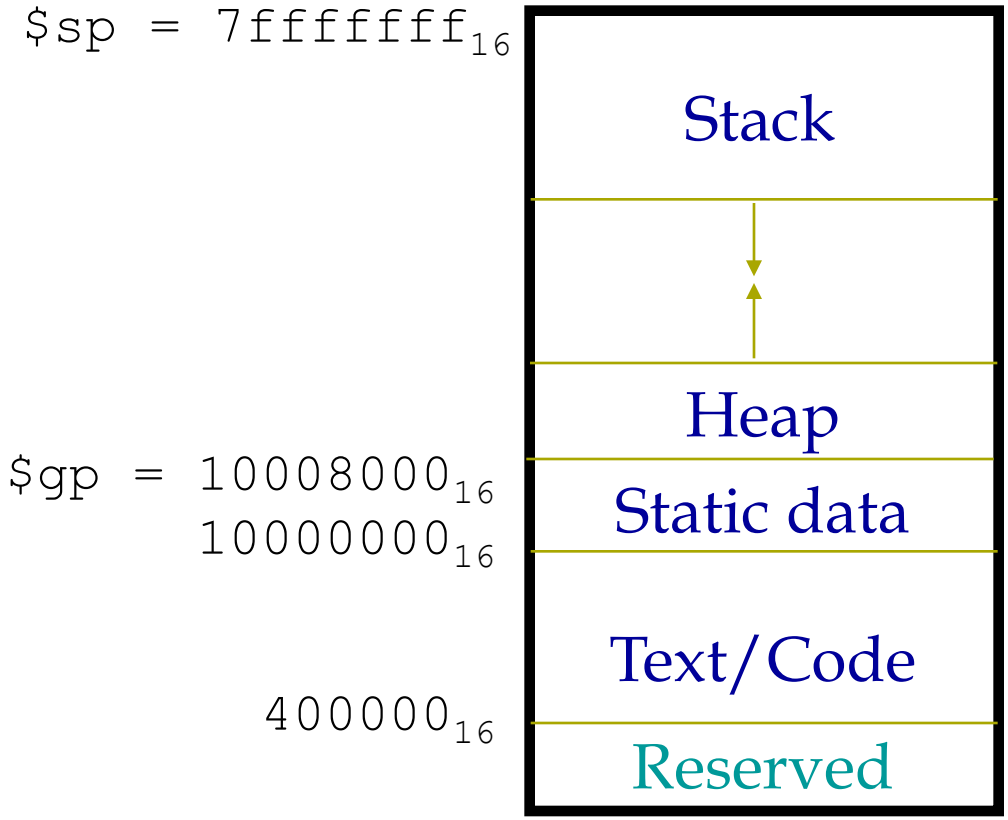
- The stack grows downward and shrinks upward



Call/Return Conventions

- Need common rules and layouts
  - To allow multi-procedure programs to work robustly
  - Procedures may be compiled and distributed separately
- Common conventions for MIPS
  - Stack grows downwards
  - Entire procedure frame is pushed and popped
    - Rather than single elements
  - Conventions for register use
    - More on this soon
- No special connection to HW here
  - All implemented with regular instructions
  - Alternative conventions would probably work as well

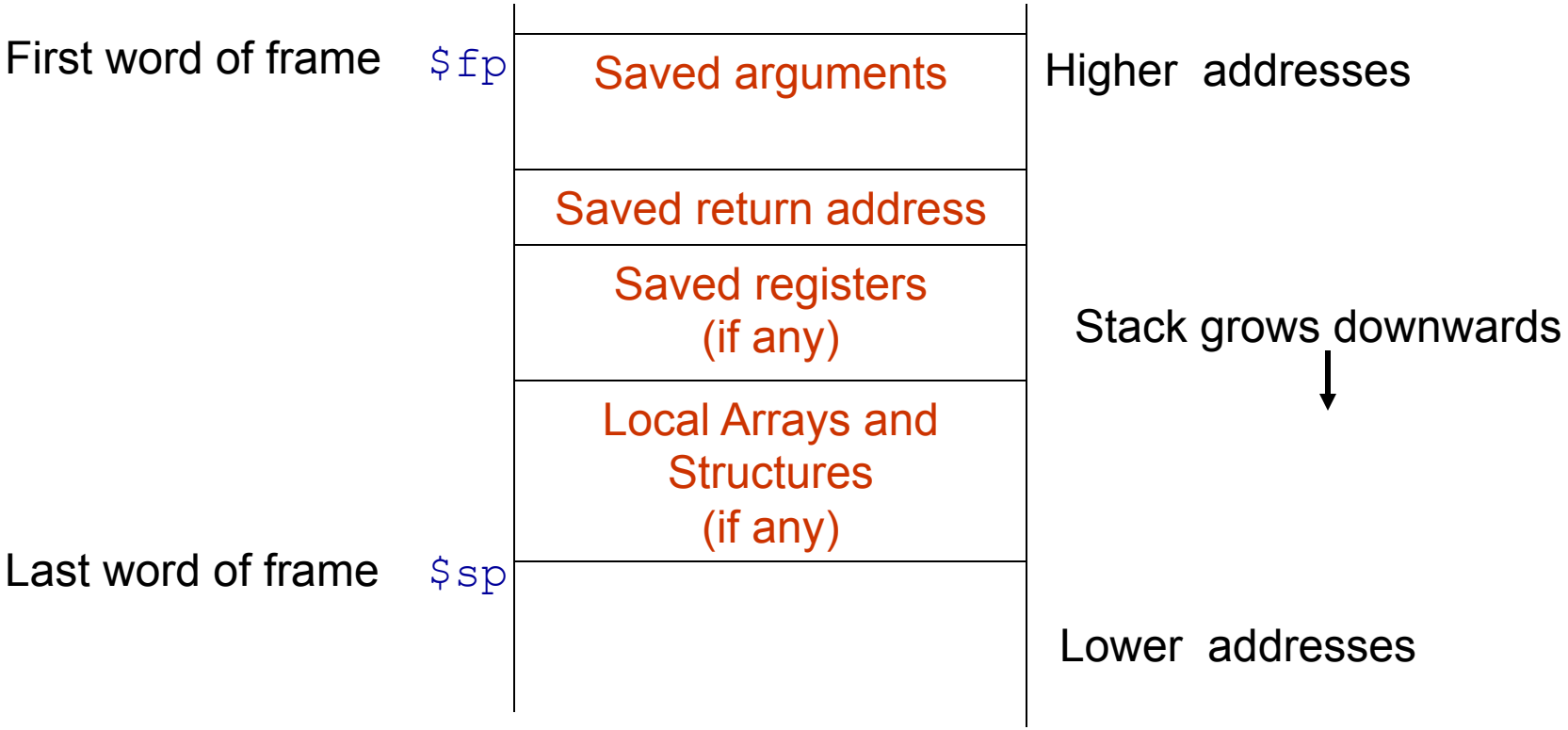
MIPS Storage Layout (Software Convention)



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

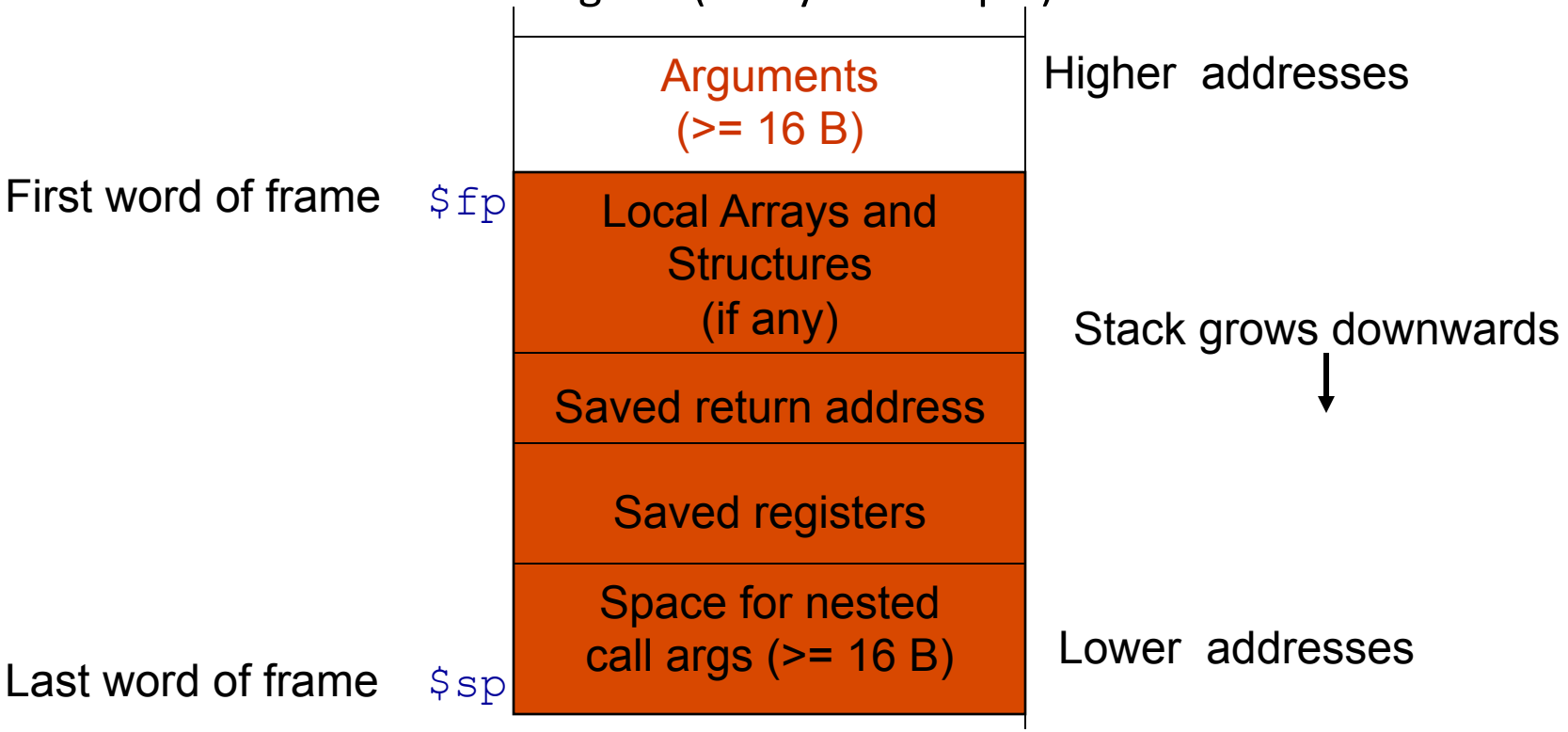
Procedure Activation Record or Frame

- Each procedure creates an activation record on 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 Calling Convention

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	Temporaries: caller saved
. . .		
15	$t7	

16	$s0	Temporaries: 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

Callee saved

Caller vs. Callee Saved Registers

Callee Saved	Caller Saved
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)

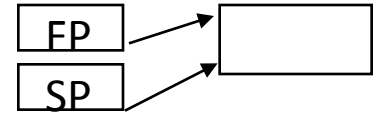
- Callee saved registers (preserved for caller)
  - Save register values on stack prior to use
  - Restore registers before return
- Caller saved registers (not preserved for caller)
  - Do what you please and expect callees to do likewise
  - Should be saved by the caller if needed after procedure call

Call and Return: the Details

- Caller
  - Save caller-saved registers `$a0-$a3`, `$t0-$t9` as needed
  - Load arguments in `$a0-$a3`, rest passed on stack
  - Execute `jal`
- Callee setup
  1. Allocate memory for new frame (`$sp` = `$sp` - frame)
  2. Save callee-saved registers `$s0-$s7`, `$fp`, `$ra` **as needed**
  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`
- Caller
  - Restore any caller-saved registers as needed

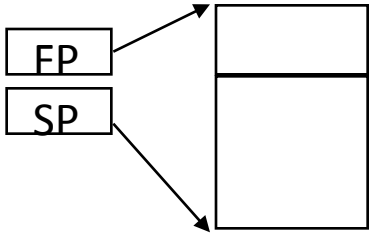
Calling Convention Steps

Before call:



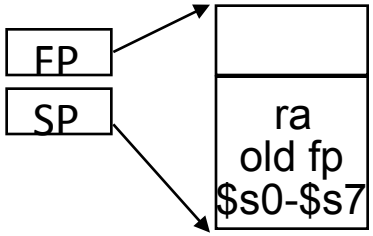
First four arguments passed in registers

Callee setup; step 1



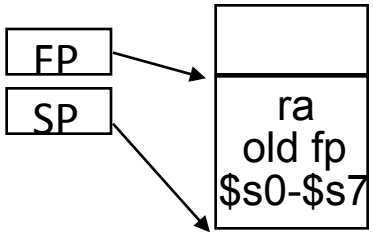
Adjust SP

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)  # Save $ra
        sw      $fp, 24($sp)  # Save $fp
        addiu   $fp, $sp, 28  # Set new $fp
        addiu   $a0, $a0, 1   # num + 1
        jal     bar           # call bar
        lw      $fp, 24($sp)  # Restore $fp
        lw      $ra, 28($sp)  # Restore $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      # n < 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    $a0, 20($sp)     # Save n
        addui $a0, $a0, -1     # n - 1
        jal   fact             # Call recursive
link:   lw    $a0, 20($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
```