CS202 : COMPUTER ORGANIZATION

Lecture 3
Instruction Set Architecture (2)

2023 Spring

Today’s Agenda

- Recap
- Context
  - More control instructions
  - Procedure call 函数调用
  - Character data
- Reading: Textbook, Section 2.8, 2.9

Recap

- Lecture2
  - MIPS
    - Introduction
      - CISC
      - RISC
    - Register file
      - 32-32bits registers
      - $s0-$s7, $t0-$t9
    - Composition of Instruction
      - operator
      - operand
    - Design principles (regularity, small, common case, compromise
    - Big endian
      - MSByte at LSAddress
  - Instruction Type
    - Arithmetic
      - immediate
    - Data transfer
      - byte offset
    - Logical
    - Branch/Jump
    - Instruction format
      - R-type
      - I-type
  - Data representation
    - unsigned/signed binary
    - 2's complement

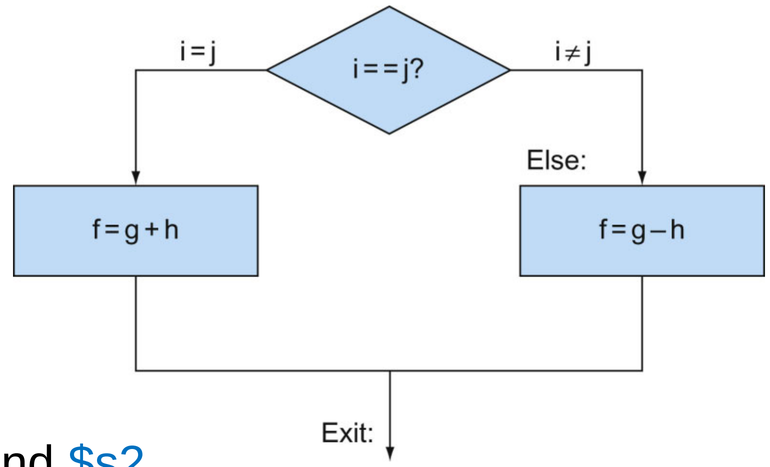
一个寄存器32位

数据在寄存器

Compiling If Statements

- C code

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



i and j are in $s3 and $s4,
f,g and h are in $s0, $s1 and $s2

- Compiled MIPS code:

```
      bne $s3, $s4, Else  # go to Else if i ≠ j
      add $s0, $s1, $s2   # f=g+h, skipped if i ≠ j
      j   Exit            # go to Exit
Else: sub $s0, $s1, $s2   # f=g-h, skipped if i = j
Exit:
```

Assembler calculates addresses

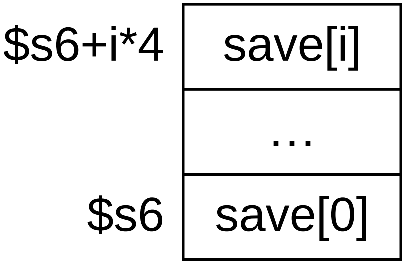
Compiling Loop Statements

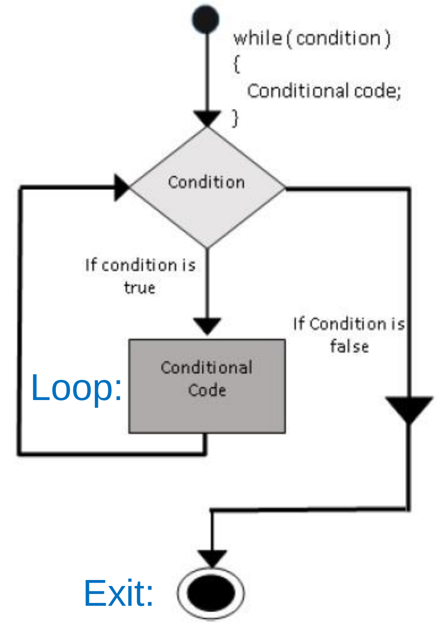
- C code:

  while (save[i] == k)

    i += 1;

- i in $s3, k in $s5, address of save in $s6



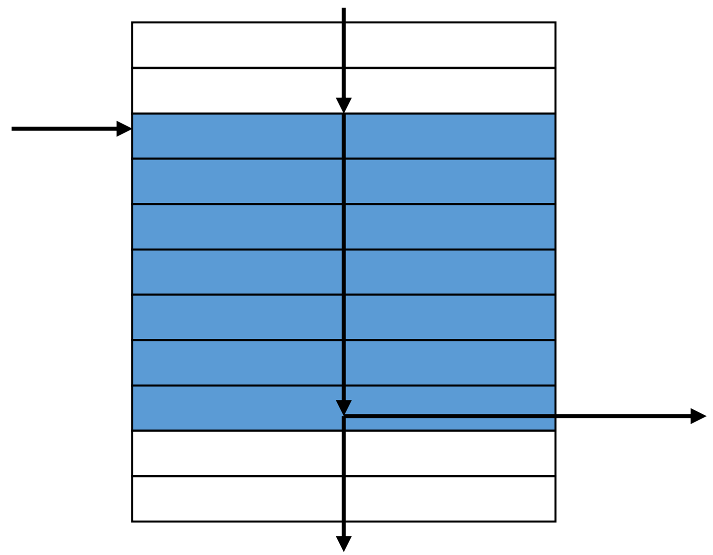


- Compiled MIPS code:

```
Loop: sll   $t1, $s3, 2     # Temp reg $t1 = i * 4
      add   $t1, $t1, $s6   # $t1 = address of save[i]
      lw    $t0, 0($t1)     # Temp reg $t0 = save[i]
      bne   $t0, $s5, Exit  # go to Exit if save[i]≠k
      addi  $s3, $s3, 1     # i = i + 1
      j     Loop            # go to Loop
Exit:
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



  - A compiler identifies basic blocks for optimization
  - An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- How to compile:
  - If (a < b) …, else, …
- *slt rd, rs, rt*
  - if (rs < rt) rd = 1; else rd = 0;
- *slti rt, rs, constant*
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt    $t0, $s1, $s2    # if ($s1 < $s2)
bne    $t0, $zero, L    #   branch to L
```

Compiling If Statements

- C code

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

i and j are in $s3 and $s4,
f,g and h are in $s0, $s1 and $s2

- Compiled MIPS code:

```
        slt $t0, $s3, $s4       #if ($s3<$s4) $t0=1, else $t0=0
        beq $t0, $zero, Else    #if ($t0==0) goto Else
        add $s0, $s1, $s2       # f=g+h, skipped if i >= j
        j Exit                  # go to Exit
Else:   sub $s0, $s1, $s2       f=g-h, # skipped if i < j
Exit:
```

Pseudo-instructions

	Pseudoinstruction	Usage
Copy	Move	`move regd,regs`
	Load address	`la regd,address`
	Load immediate	`li regd,anyimm`
Arithmetic	Absolute value	`abs regd,regs`
	Negate	`neg regd,regs`
	Multiply (into register)	`mul regd,reg1,reg2`
	Divide (into register)	`div regd,reg1,reg2`
	Remainder	`rem regd,reg1,reg2`
	Set greater than	`sgt regd,reg1,reg2`
	Set less or equal	`sle regd,reg1,reg2`
	Set greater or equal	`sge regd,reg1,reg2`
Shift	Rotate left	`rol regd,reg1,reg2`
	Rotate right	`ror regd,reg1,reg2`
Logic	NOT	`not reg`
Memory access	Load doubleword	`ld regd,address`
	Store doubleword	`sd regd,address`
Control transfer	Branch less than	`blt reg1,reg2,L`
	Branch greater than	`bgt reg1,reg2,L`
	Branch less or equal	`ble reg1,reg2,L`
	Branch greater or equal	`bge reg1,reg2,L`

Pseudo-instructions Example

- Register move
  - Format: `move reg2,reg1`
  - Equivalent to: `add reg2,$zero,reg1`
- Load immediate
  - Format: `li reg, value`
  - If value fits in 16 bits: `addi reg,$zero,value (ori)`
  - Otherwise: `lui reg, upper 16 bits of value`
    `ori reg,$zero,lower 16 bits`
- Load address
  - Format: `la reg, value`
  - Equivalent to: `lui $at, 0x1001`
    `ori $s0, $at, 16`
- Branch less than
  - Format: `blt reg1, reg2, Label`
  - Equivalent to: `slt reg3, reg1, reg2`
    `bne reg2, $zero, Label`

There is no such instructions in hardware,
The assembler translates them into a combination of real instructions

Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

Question

- C has many statements for decisions and loops, while MIPS has few. Which of the following do or do not explain this imbalance? Why?

1. More decision statements make code easier to read and understand.
2. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
3. More decision statements mean fewer lines of code, which generally reduces coding time.
4. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

Signed vs. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - slt $t0, $s0, $s1 # signed
    - $-1 < +1 \Rightarrow$ $t0 = 1
  - sltu $t0, $s0, $s1 # unsigned
    - $+4{,}294{,}967{,}295 > +1 \Rightarrow$ $t0 = 0

The register contains bits without meaning.
Are the bits represents a signed number or unsigned one?
See the instruction!

Procedure Calling

- A procedure or function is one tool used by the programmers to structure programs
  - Benefit: easy to understand, reuse code
- Six Steps of Calling a Function
  1. Put 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. Put the result value in a place where the calling program can access it.
  6. Return control to the point of origin, since a procedure can be called from several points in a program.

Procedure Calling-Question1

- Step 1, 5 and 6: Where should we put the arguments and return values?
  - Registers way faster than memory, so use them whenever possible
    - $a0 - $a3: four argument registers to pass parameters
    - $v0 - $v1: two value registers to return the values
    - $ra: one return address register used to save where a function is called from so we can get back
  - If need extra space, use memory (the stack!)

Procedure Calling-Question2

- Step 2 and 6：How do we Transfer Control?
  - Procedure call: jump and link
    jal ProcedureLabel
    - Address of following instruction put in $ra
    - Jumps to target address
    - Used by Caller
  - Procedure return: jump register
    jr $ra
    - Copies $ra to program counter
    - Can also be used for computed jumps
      - e.g., for case/switch statements
    - Used by Callee
  - PC(Program Counter)
    - A special register maintains the address of the instruction currently being executed

Caller's Code

- C code

```
int main()
{  . . .
    sum = leaf_example(a,b,c,d)
    . . .
}
```

- MIPS code: a, …, d in $s0, …, $s3, and sum in $s4

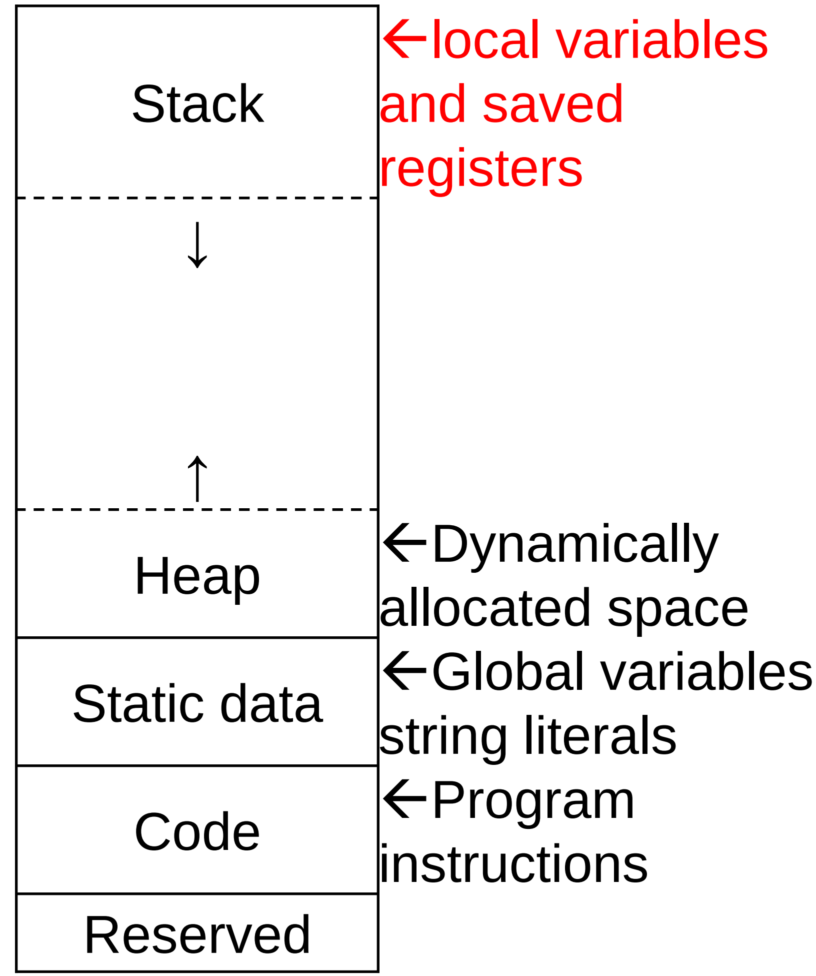
```
add   $a0, $zero, $s0
add   $a1, $zero, $s1
add   $a2, $zero, $s2
add   $a3, $zero, $s3
jal   leaf_example
add   $s4, $0, $v0
```

Procedure Calling-Question3

- Step 3: What's the Local storage for variables?
  - We use stack to save registers
  - $sp (stack pointer) holds the address at the bottom of the stack
    - Decrement it (recall stack grows downwards)
    - Then use store word to write to a variable
    - To "clean up", just increment the stack pointer

Memory Layout vs Stack

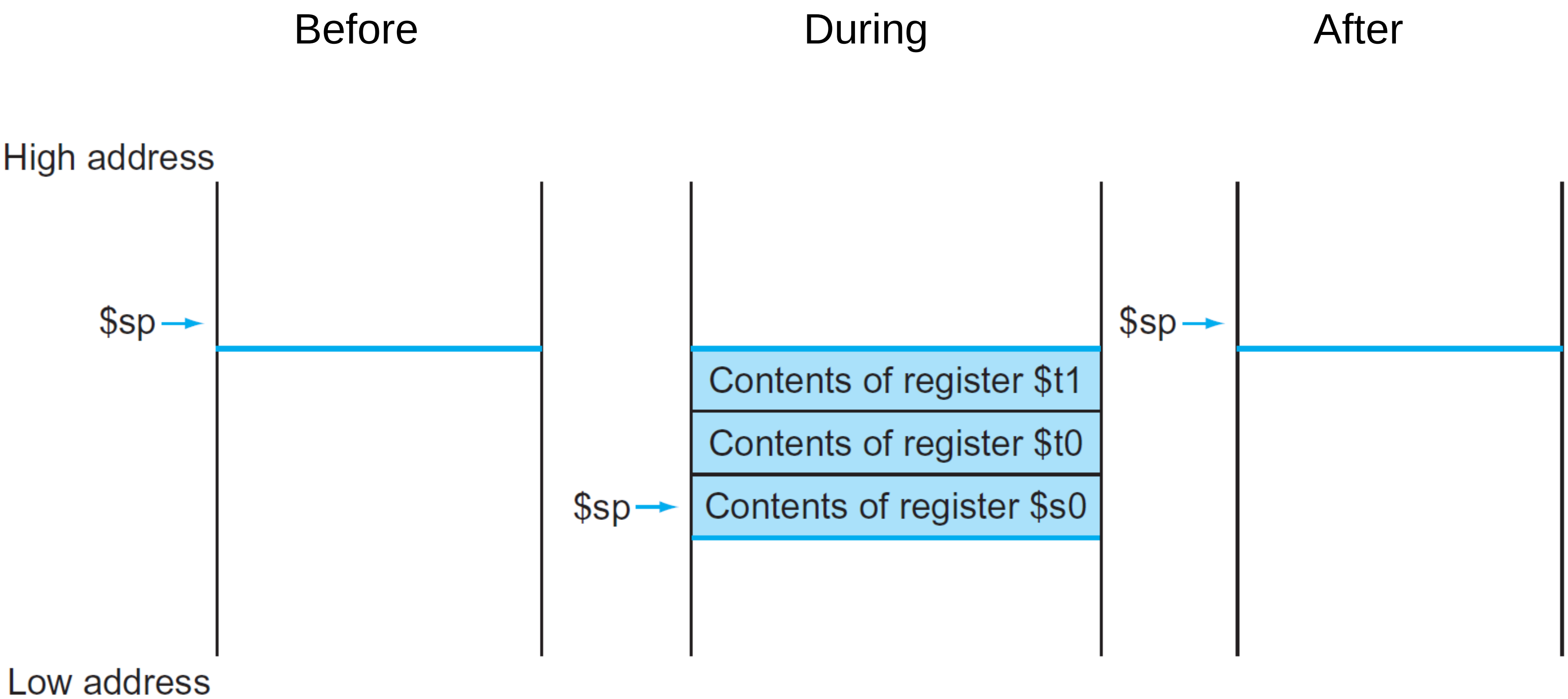
High address



Low address

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage

Stack Before, During, After Call



Leaf Procedure Example

被调用的函数里面不再调用其它函数

- C code:

```
int leaf_example (int g, int h, int i, int j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Register Assignment
  - Arguments g, …, j in $a0, …, $a3
  - f in $s0 (hence, need to save $s0 on stack)
  - Result in $v0

Return address	$ra
Procedure address	Labels
Arguments	$a0, $a1, $a2, $a3
Return value	$v0, $v1
Local variables	$s0, $s1, …, $s7
Note the use of register conventions	

Leaf Procedure Example

- MIPS code:

$a0, $a1, $a2, $s3

```
int leaf_example (int g, int h, int i, int j)
  { int f;
    f = (g + h) - (i + j);
    return f;
  }
```

$s0 $t0 $t1

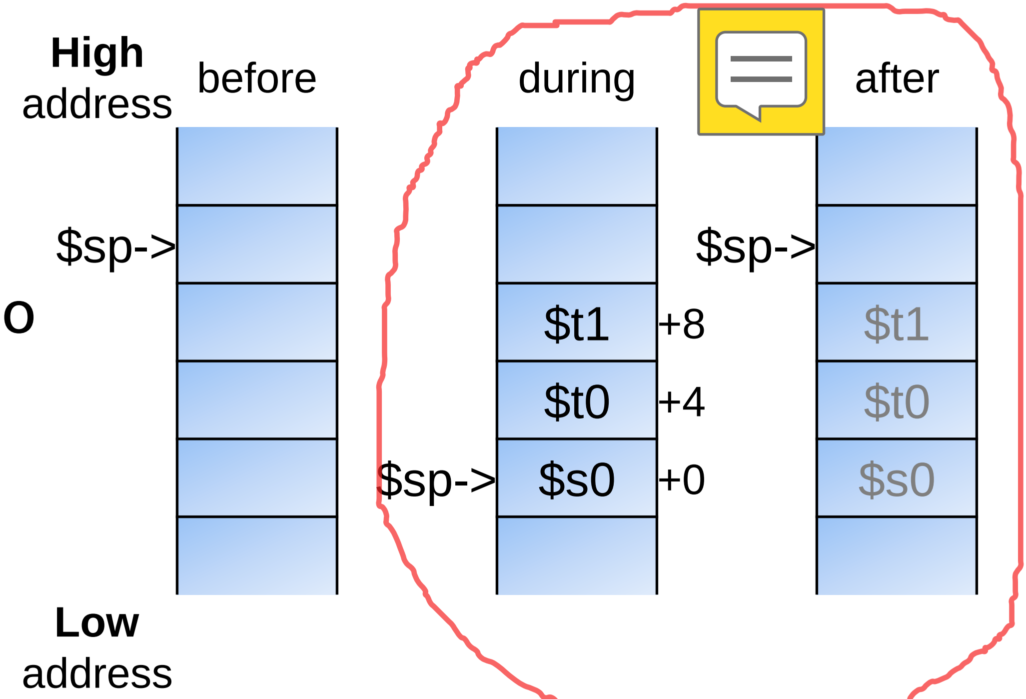
```
leaf_example:
    addi $sp, $sp, -12
    sw   $t1, 8($sp)
    sw   $t0, 4($sp)
    sw   $s0, 0($sp)
    add  $t0, $a0, $a1
    add  $t1, $a2, $a3
    sub  $s0, $t0, $t1
    add  $v0, $s0, $zero
    lw   $s0, 0($sp)
    lw   $t0, 4($sp)
    lw   $t1, 8($sp)
    addi $sp, $sp, 12
    jr   $ra
```

push

compute

return value

pop



Leaf Procedure Example(cont.)

- Optimized MIPS code:

```
leaf_example:
   addi $sp, $sp, -4
   sw   $s0, 0($sp)
   add  $t0, $a0, $a1
   add  $t1, $a2, $a3
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero
   lw   $s0, 0($sp)
   addi $sp, $sp, 4
   jr   $ra
```

To avoid too many memory operations:

- \$t0 -\$t9: temporary registers are not preserved by the callee
- \$s0 -\$s7: saved registers must be preserved by the callee if used

- Even better version?

```
leaf_example:
  add  $t0, $a0, $a1
  add  $t1, $a2, $a3
  sub  $v0, $t0, $t1  #don't put f in $s0
  jr   $ra
```

Procedure Calling-Question 4

- Step 4: Function Calling Conventions?
  - Register Conventions as a contract between the Caller and the Callee
  - Caller: function making the call, using jal
  - Callee: function being called
- If both the Caller and Callee obey the procedure conventions, there are significant benefits
  - People who have never seen or even communicated with each other can write functions that work together
  - Recursion functions work correctly

Caller’s Rights, Callee’s Rights

- Callees’ rights:
  - Right to use VAT registers freely
  - Right to assume arguments are passed correctly
- To ensure callees’s right, caller saves registers:
  - Return address $ra
  - Arguments $a0, $a1, $a2, $a3
  - Return value $v0, $v1
  - $t Registers $t0 - $t9
- Callers’ rights:
  - Right to use S registers without fear of being overwritten by callee
  - Right to assume return value will be returned correctly
- To ensure caller’s right, callee saves registers:
  - $s Registers $s0 - $s7

Contract in Function Calls (1)

- Caller's responsibilities (how to call a function)
- Slide $sp down to reserve memory:
  e.g., addi $sp, $sp, -8
- Save $ra on stack because jal might overwrites it:
  e.g., sw $ra, 4 ($sp)
- If still need their values after function call, save $v, $a, $t on stack or copy to $s registers
- Put first 4 words of arguments in $a0-3, additional arguments go on stack
- jal to the desired function
- Receive return values in $v0, $v1
- Undo first steps:

  e.g., lw $t0, 0($sp)

  lw $ra, 4($sp)

  addi $sp, $sp, 8

Contract in Function Calls (2)

- Callee’s responsibilities (i.e. how to write a function)
- If using $s or big local structures, slide $sp down to reserve memory:

  e.g., addi $sp, $sp, -48
- If using $s, save before using:

  e.g., sw $s0, 44($sp)
- Receive arguments in $a0-3, additional arguments on stack
- Run the procedure body
- If not void, put return values in $v0,1
- If applicable, undo first two steps:

  e.g., lw $s0, 44($sp)
  addi $sp, $sp, 48
- jr $ra

MIPS register conventions

Name	Register number	Usage	Preserved on call?
$zero	0	The constant value 0	n.a.
$v0–$v1	2–3	Values for results and expression evaluation	no
$a0–$a3	4–7	Arguments	no
$t0–$t7	8–15	Temporaries	no
$s0–$s7	16–23	Saved	yes
$t8–$t9	24–25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

Non-Leaf Procedures 嵌套调用

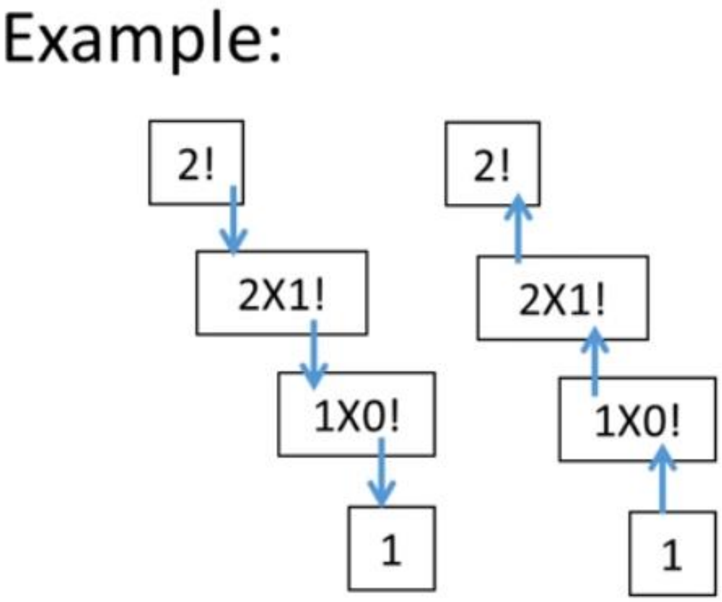
- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

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

- Register Assignment
  - Argument n in $a0
  - Result in $v0



Non-Leaf Procedure Example

- MIPS code:

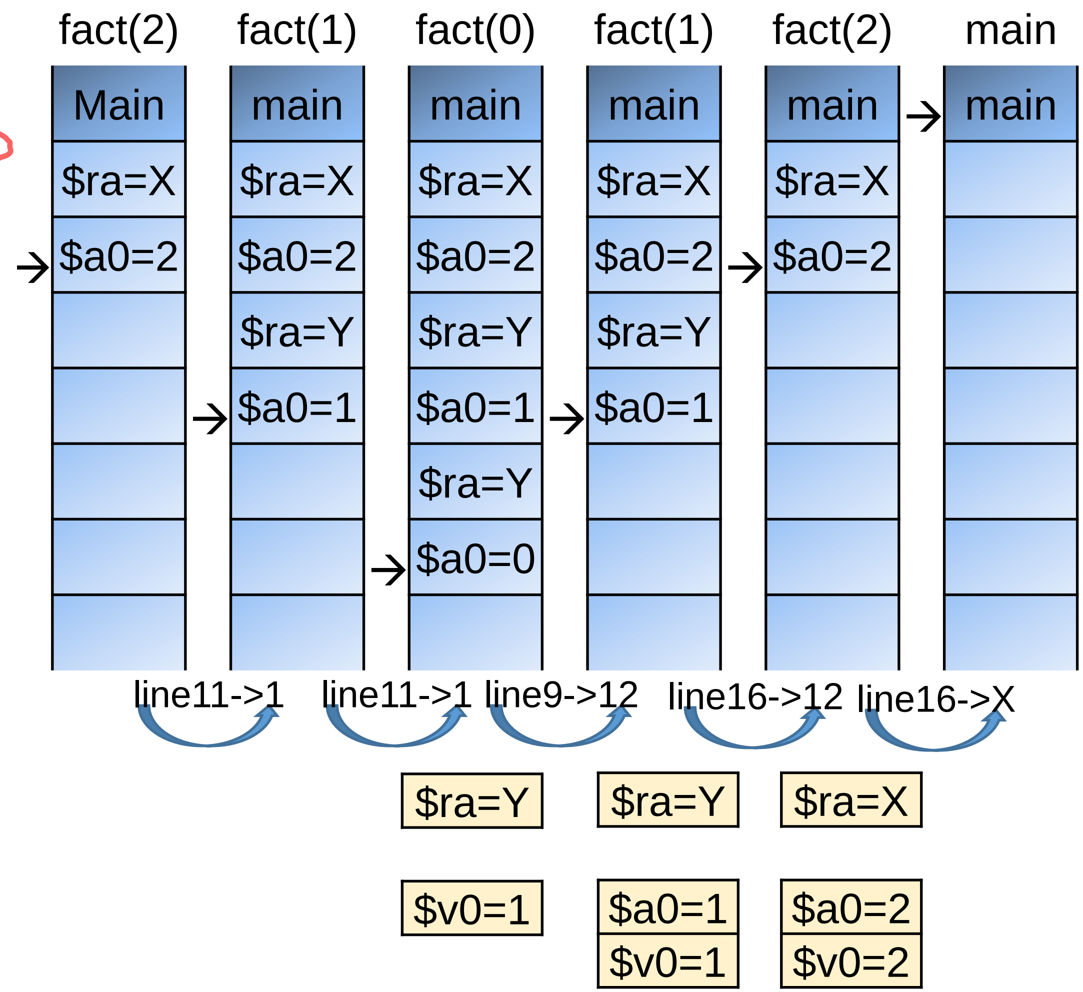
```
fact:
    addi $sp, $sp, -8     # adjust stack for 2 items
    sw   $ra, 4($sp)      # save return address
    sw   $a0, 0($sp)      # save argument
    slti $t0, $a0, 1      # test for n < 1
    beq  $t0, $zero, L1
    addi $v0, $zero, 1    # if so, result is 1
    addi $sp, $sp, 8      # pop 2 items from stack
    jr   $ra              # and return
L1: addi $a0, $a0, -1     # else decrement n
    jal  fact             # recursive call
    lw   $a0, 0($sp)      # restore original n
    lw   $ra, 4($sp)      # and return address
    addi $sp, $sp, 8      # pop 2 items from stack
    mul  $v0, $a0, $v0    # multiply to get result
    jr   $ra              # and return
```

Example details

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

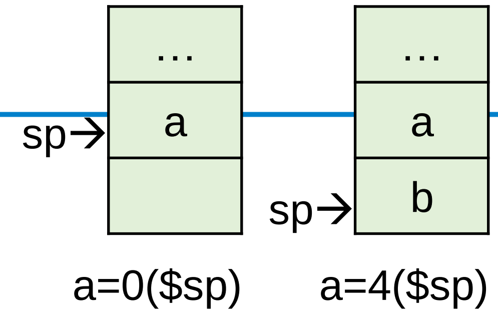
- MIPS code: suppose n=2

```
1:fact:
2:      addi  $sp, $sp, -8
3:      sw    $ra, 4($sp)
4:      sw    $a0, 0($sp)
5:      slti  $t0, $a0, 1
6:      beq   $t0, $zero, L1
7:      addi  $v0, $zero, 1
8:      addi  $sp, $sp, 8
9:      jr    $ra
10:L1:  addi  $a0, $a0, -1
11:     jal   fact
12:Y:   lw    $a0, 0($sp)
13:     lw    $ra, 4($sp)
14:     addi  $sp, $sp, 8
15:     mul   $v0, $a0, $v0
16:     jr    $ra
```

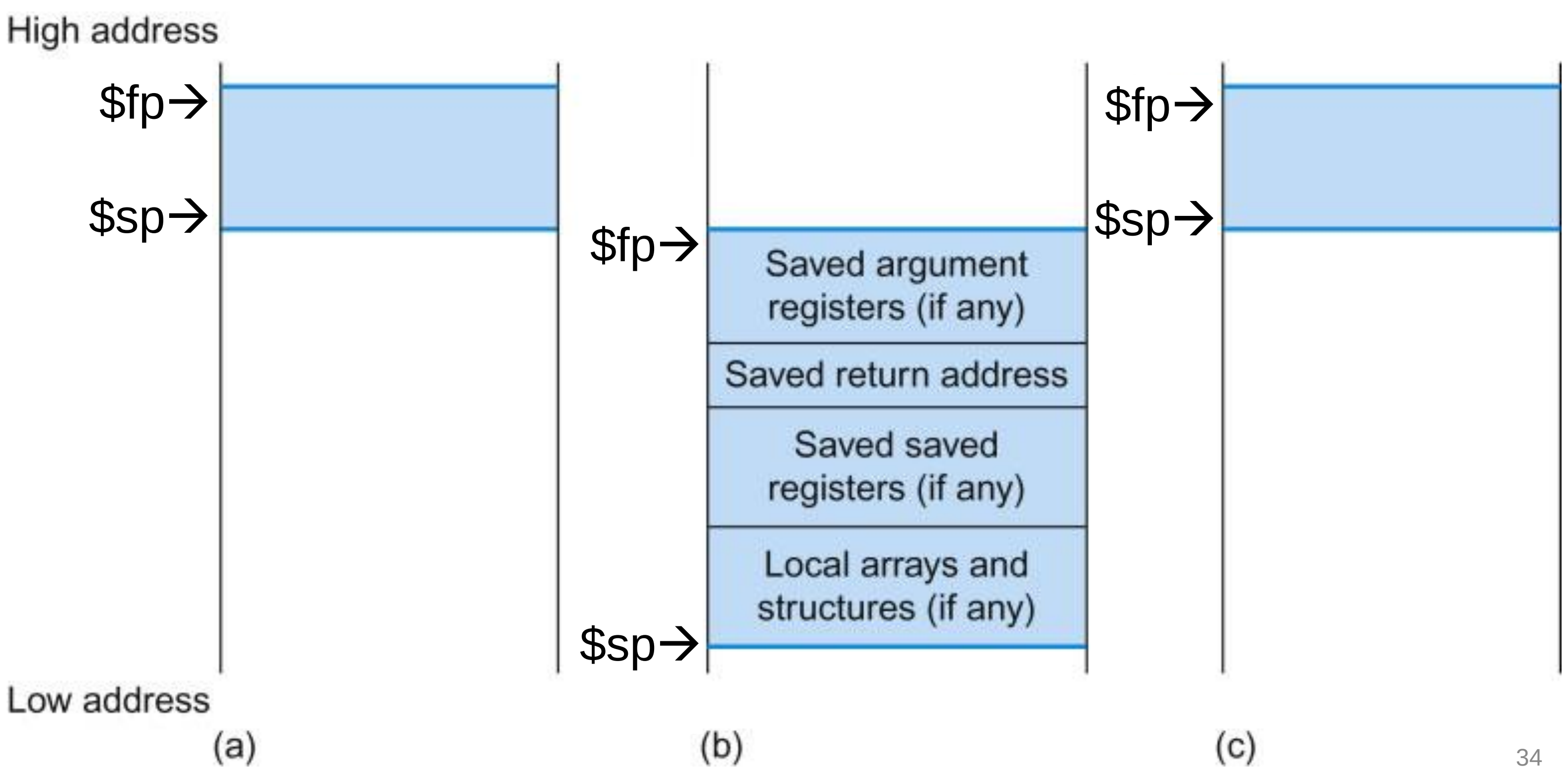


X: return address of main
Y: the address of line 12

Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage



Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world’s alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
- String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

- Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

- Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

- Store just rightmost byte/halfword

String Copy Example

```
void strcpy (char x[], char y[])
{ int i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

- C code (naïve):
  - Null-terminated string
  - i in $s0
  - Addresses of
    x, y in $a0, $a1
- MIPS code:

```
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw   $s0, 0($sp)       # save $s0
    add  $s0, $zero, $zero # i = 0
L1: add  $t1, $s0, $a1     # addr of y[i] in $t1
    lbu  $t2, 0($t1)       # $t2 = y[i]
    add  $t3, $s0, $a0     # addr of x[i] in $t3
    sb   $t2, 0($t3)       # x[i] = y[i]
    beq  $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j    L1                # next iteration of loop
L2: lw   $s0, 0($sp)       # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr   $ra               # and return
```

Summary

- Control instructions
  - beq, bne
  - slt+beq/bne
  - psudo: blt, bgt, ble, bge
- Procedure call
  - Caller vs callee, jal vs jr
  - Leaf procedure vs non-leaf procedure
  - Memory layout, stack， push vs pop
  - Register convention
- Character data
  - lb/lbu,lh/lhu
  - sb, sh