Lecture 4

Instruction Set Architecture(2)

Recap

- Instruction set architecture
  - RISC vs. CISC
  - MIPS/ARM/x86
- Instructions:
  - Arithmetic instruction: add, sub, …
  - Data transfer instruction: lw, sw, lh, sh, …
  - Logical instruction: and, or, …
  - Conditional branch beq, bne, …
- Basic concepts:
  - Operands: register vs. memory vs. immediate
  - Numeric representation: signed, unsigned, sign extension
  - Instruction format: R-format vs. I-format

Today's topic

- More control instructions
- Procedure call/return

Control Instructions: if else

- Conditional branch: Jump to instruction L1 if register1 equals to register2: beq register1, register2, L1
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  j L1
  jr $s0

Convert to assembly:

```
if (i == j)                    bne  $s3, $s4, Else
   f = g+h;                    add  $s0, $s1, $s2
else                           j    Exit
   f = g-h;             Else:  sub  $s0, $s1, $s2
                        Exit:
```

Loop

Convert to assembly:

```
while   (save[i] == k)
     i += 1;
```

i and k are in $s3 and $s5 and base of array save[] is in $s6

```
Loop:  sll    $t1, $s3, 2
       add    $t1, $t1, $s6
       lw     $t0, 0($t1)
       bne    $t0, $s5, Exit
       addi   $s3, $s3, 1
       j      Loop
Exit:
```

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
```

Example

Convert to assembly:

Convert to assembly:

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

```
        slt   $t0, $s3, $s4
        beq   $t0, $zero, Else
        add   $s0, $s1, $s2
        j     Exit
Else:   sub   $s0, $s1, $s2
Exit:
```

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

Ans: ABC

Question

- C has many statements for decisions and loops, while MIPS has few. Which of the following correctly explain this imbalance?
  A. More decision statements make code easier to read and understand.
  B. Fewer decision statements simplify the task of the underlying layer that is responsible for execution.
  C. More decision statements mean fewer lines of code, which generally reduces coding time.
  D. More decision statements mean fewer lines of code, which generally results in the execution of fewer operations.

Pseudo Instructions

- blt $s0, $s1, Label
  - If s0<s1, jump to Label

slt $s2, $s0, $s1

bne $s2, $zero, Label

- bgt $s0, $s1, Label
  - If s0>s1, jump to Label
- ble $s0, $s1, Label
  - If s0<=s1, jump to Label
- beqz $s0, Label
  - If s0==0, jump to Label
- li $t0, 5
  - Load immediate, t0 = 5
- move $t0, $s0
  - t0 = s0

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

Branch Instruction Design

- Why blt, bge are not supported in hardware?
- 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

两个合起来导致一个指令的时间太长，而时钟信号周期要适应最长的指令，因而导致其他很快的指令在同周期内做完没事可做，而长指令尚未结束。

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!

Procedures

- A procedure or function is one tool used by the programmers to structure programs
  - Benefit: easy to understand, reuse code
- We can think of a procedure like a spy
  - acquires resources → performs task → covers his tracks → returns back with desired result
- When the procedure is executed (when the caller calls the callee), there are six steps
  - parameters (arguments) are placed where the callee can see them
  - control is transferred to the callee
  - acquire storage resources for callee
  - execute the procedure
  - place result value where caller can access it
  - return control to caller

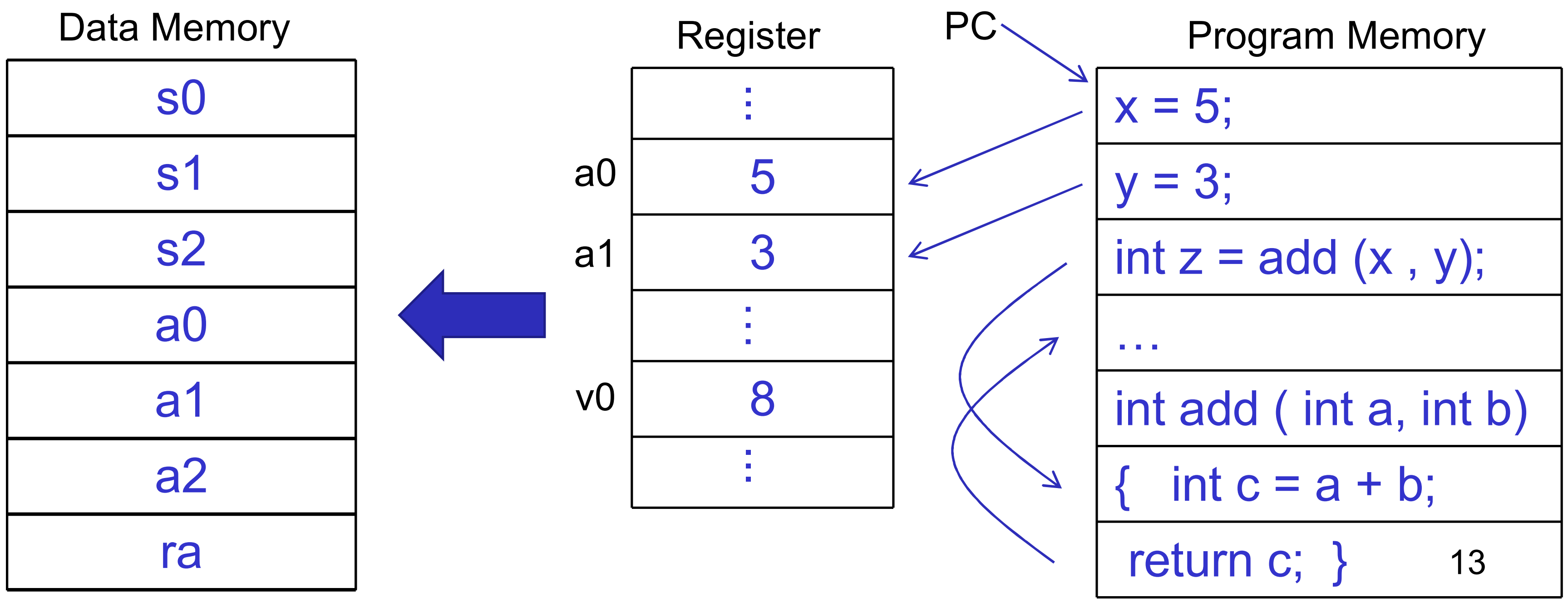
Procedure Calling

Caller:

```
int x = 5;
int y = 3;
int z = add (x , y);
x = x + 7;
…
```

callee:

```
int add ( int a, int b)
{
    int c = a + b;
    return c;
}
```



Registers Used during Procedure Calling

- The registers are used to hold data between the caller and the callee
  - $a0 - $a3: four argument registers to pass parameters
  - $v0 - $v1: two value registers to return the values
  - $ra: one return address register to return to the point of origin in the caller

Jump and Link

为啥要ra记录pc+4呢，而不是直接跳到函数下面的label呢，因为函数被调用的地方可能不止一次，所以不能跳到固定label而得跳到

- *program counter* (PC)
  - A special register maintains the address of the instruction currently being executed
- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure's address (the PC is accordingly set to this address)

jal NewProcedureAddress

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction
- How do we return control back to the caller after completing the callee procedure?

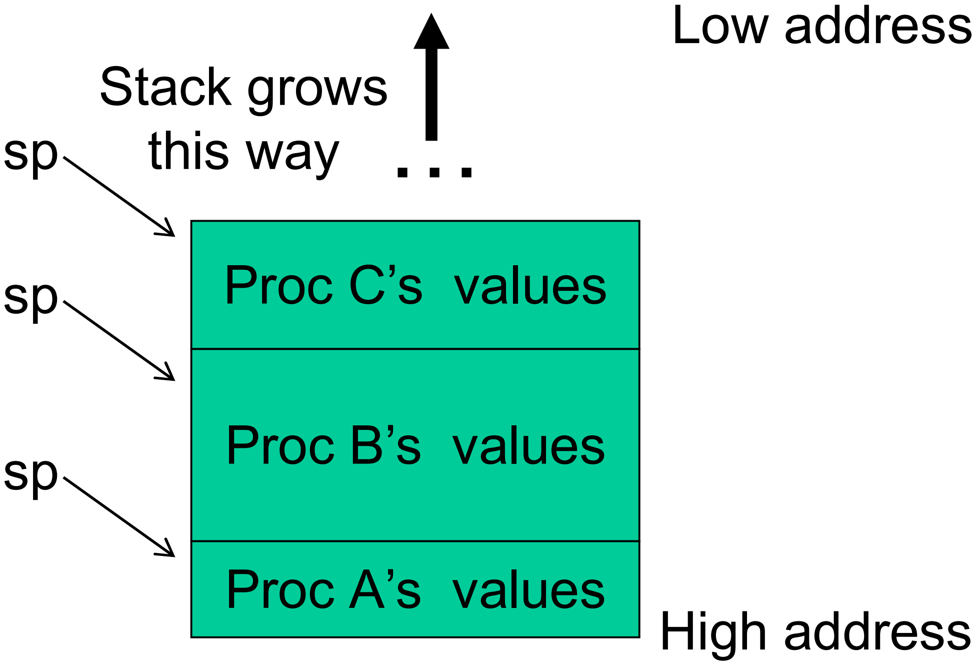
Registers

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

The Stack

在过程调用中，过程中使用的寄存器会在过程结束后恢复初值，而这是由Stack来实现的

The registers for a procedure are volatile, it disappears every time we switch procedures. Therefore, a procedure's values in the registers are backed up in memory on a stack




```
Proc  A

        call  Proc B
              …
              call Proc C
                …
              return
        return
return
```

Storage Management on a Call/Return

- A new procedure must create space for all its variables on the stack
- Before executing the jal, the caller must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into its own stack space
- Arguments are copied into $a0-$a3; the jal is executed
- After the callee creates stack space, it updates the value of $sp
- Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented
- On return, the caller may bring in its stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

Example 1- leaf procedure

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

The caller has saved:
g→$a0,
h→$a1,
i→$a2,
j→$a3,
return address → $ra

```
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
```

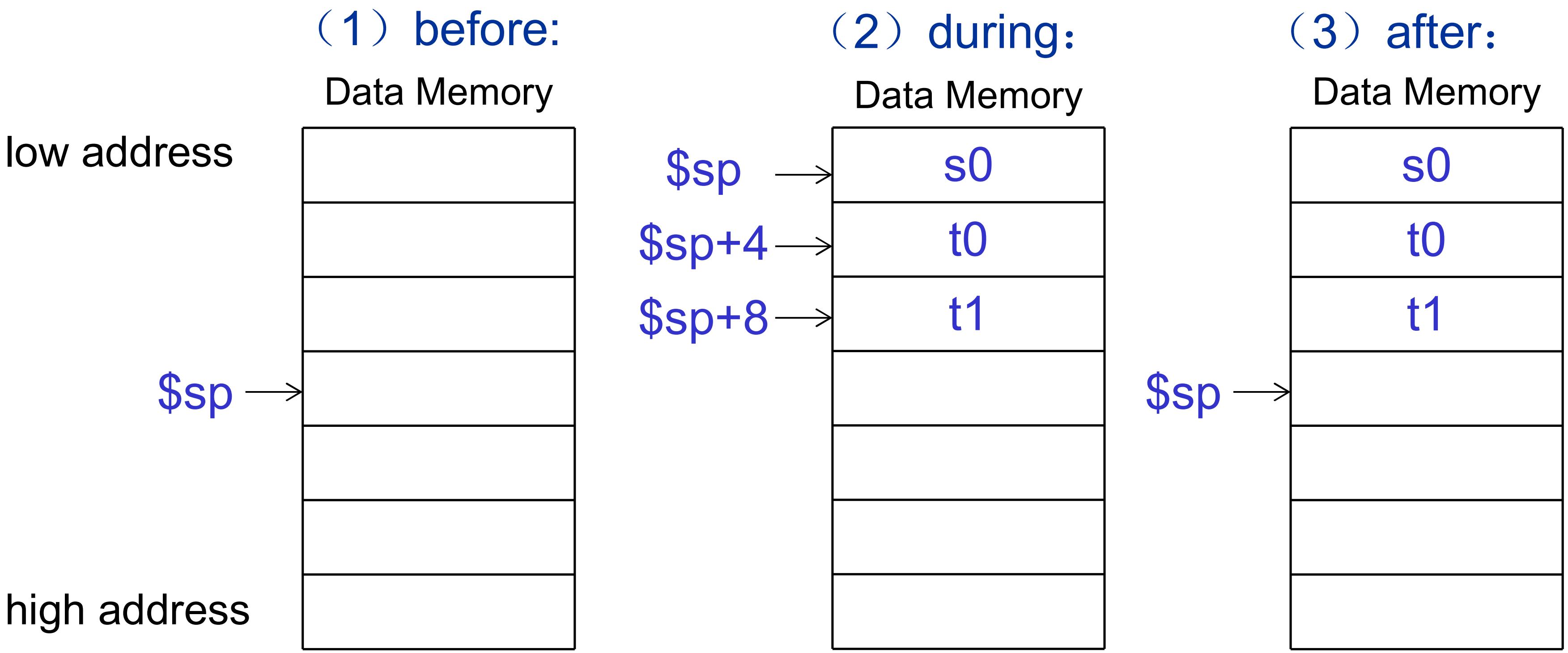
Save t0,t1,s0
Protect environment

Procedure body

Restore t0 t1 s0

Return result

Data in the stack in example 1



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

Example 2 – non-leaf procedure

- 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

Example 2 – non-leaf procedure

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

```
fact:
    addi     $sp, $sp, -8
    sw       $ra, 4($sp)
    sw       $a0, 0($sp)
    slti     $t0, $a0, 1
    beq      $t0, $zero, L1
      addi   $v0, $zero, 1
      addi   $sp, $sp, 8
      jr     $ra
L1:
    addi     $a0, $a0, -1
    jal      fact
    lw       $a0, 0($sp)
    lw       $ra, 4($sp)
    addi     $sp, $sp, 8
    mul      $v0, $a0, $v0
    jr       $ra
```

Compare n<1

Return 1

Fact(n-1)

Return n*fact(n-1)

Notes:

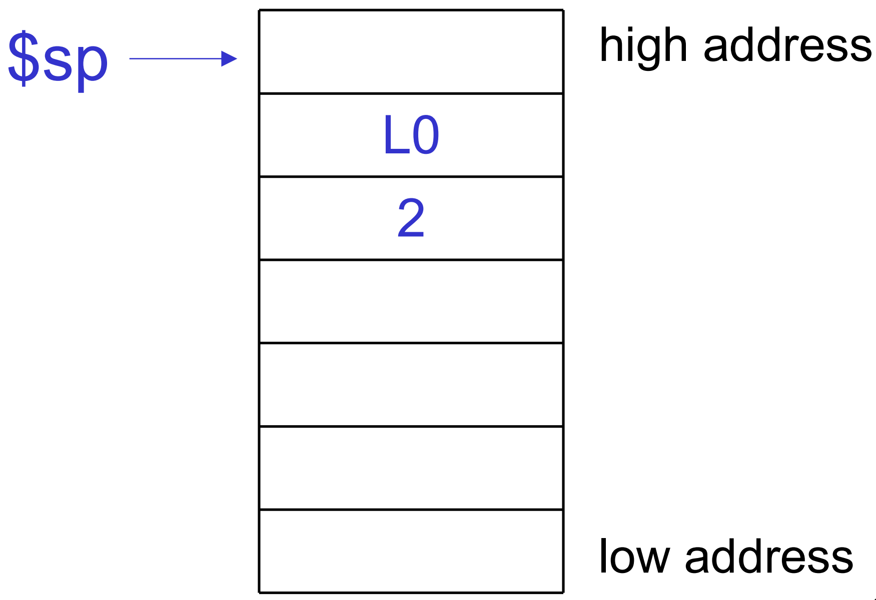
The caller saves $a0 and $ra in its stack space.

Temps are never saved.

Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

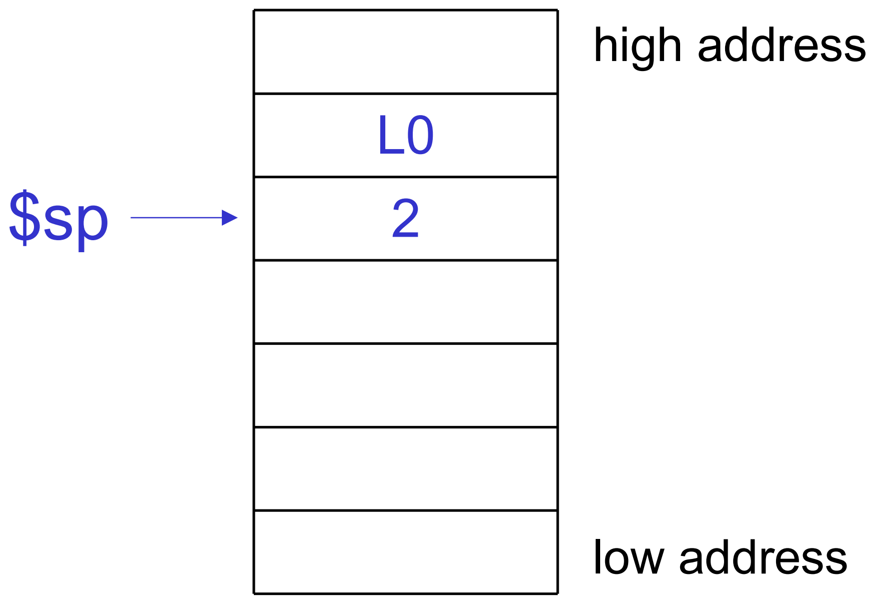
a0=2



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

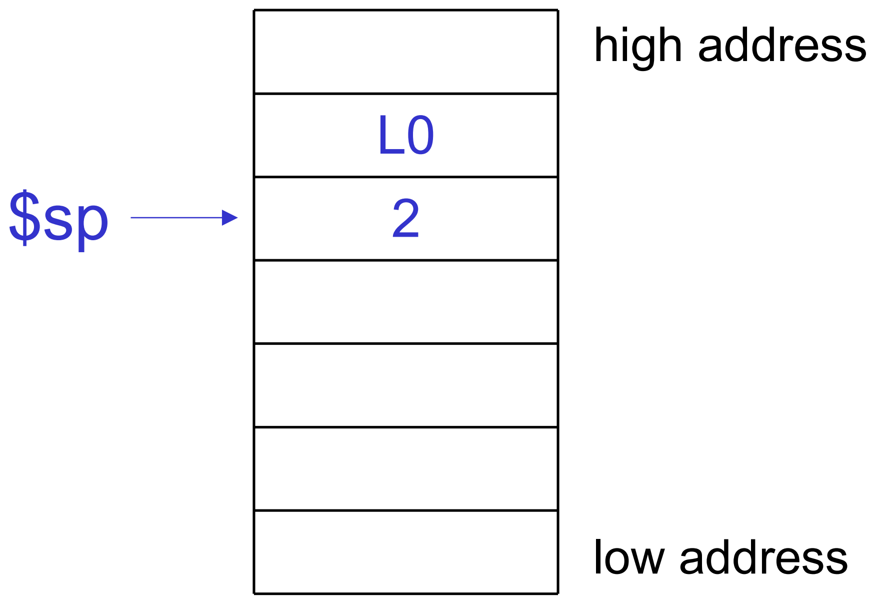
a0=2 t0=0



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

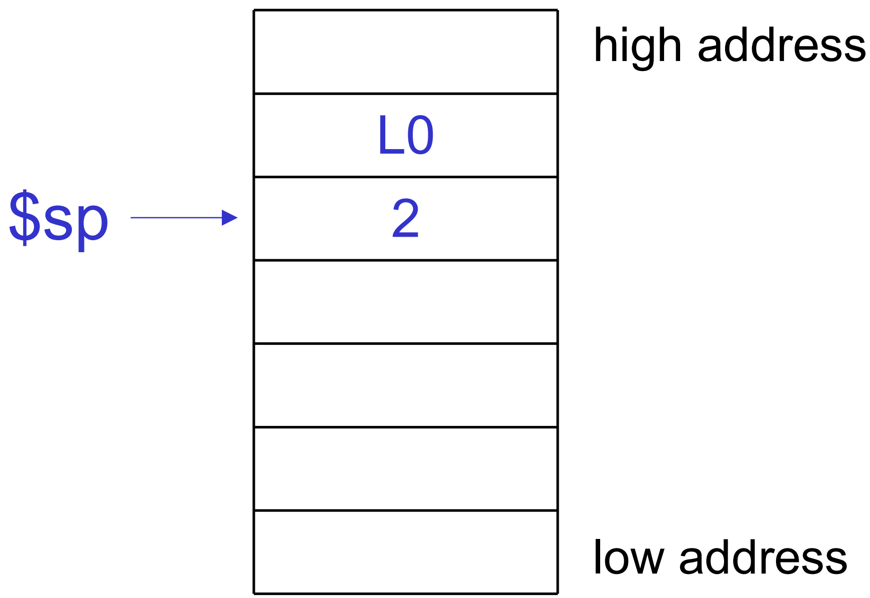
a0=1 t0=0



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

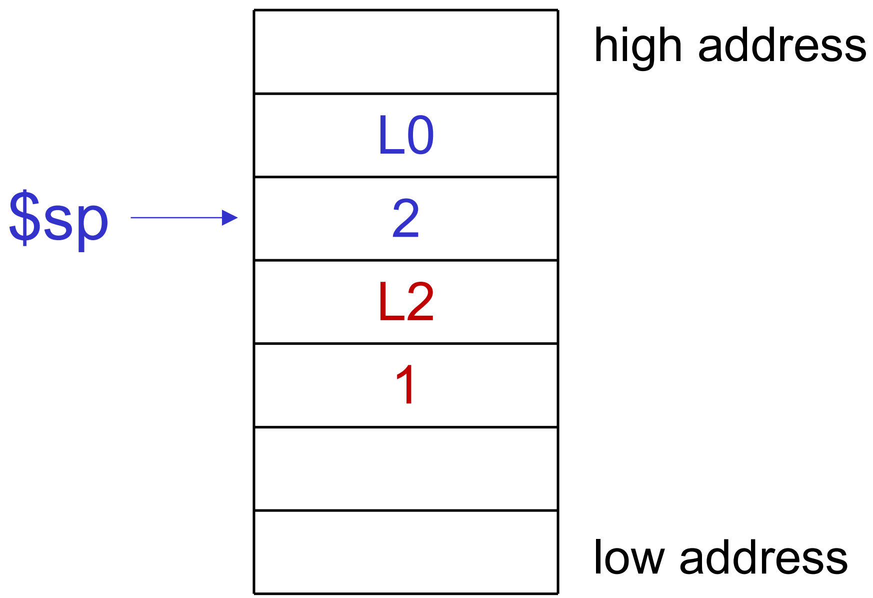
a0=1 t0=0 ra=L2



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

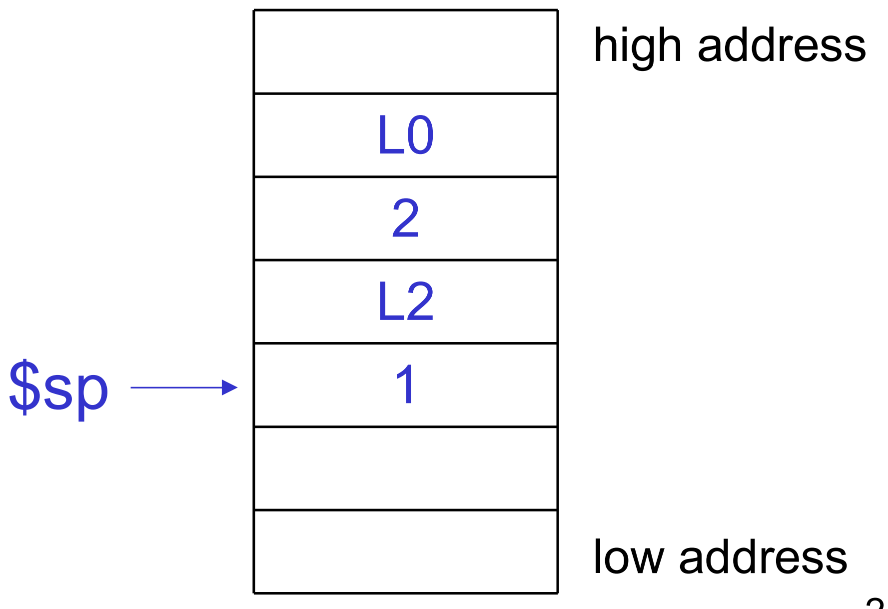
a0=1   t0=0   ra=L2



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

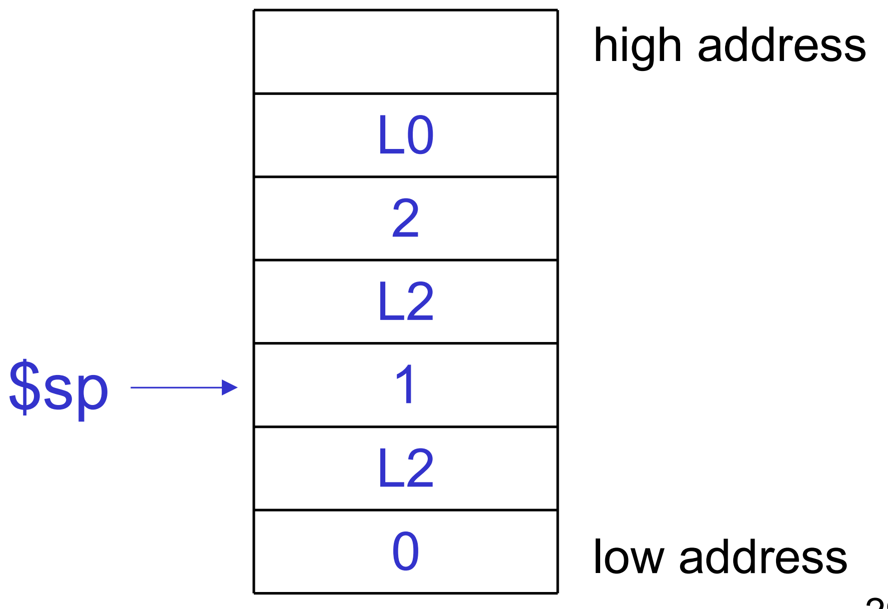
a0=0 t0=0 ra=L2



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

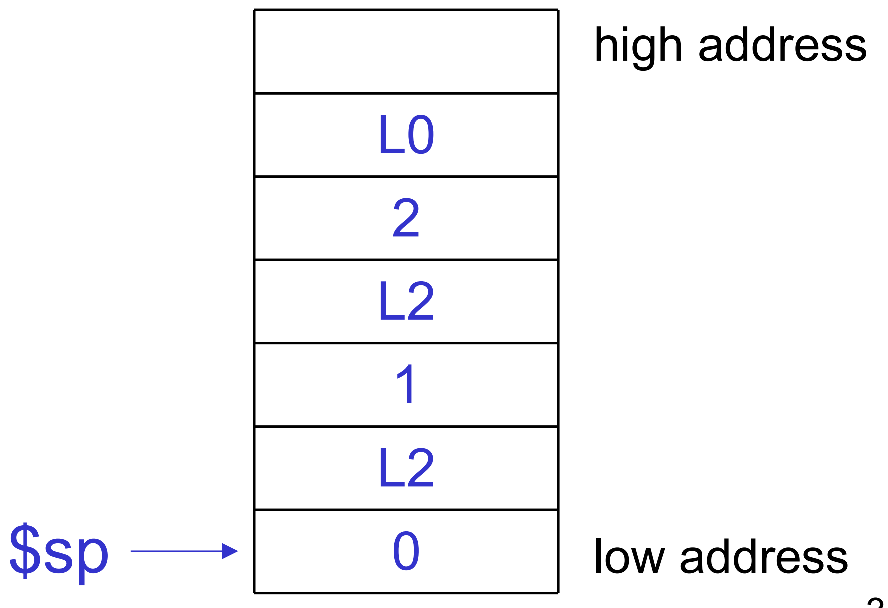
a0=0 t0=1 ra=L2



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

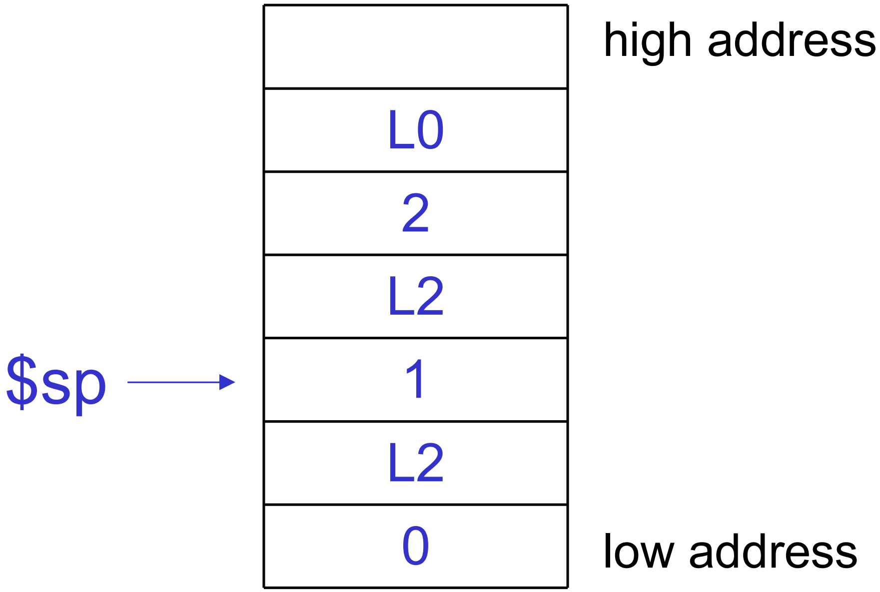
a0=0 t0=1 ra=L2 v0=1



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

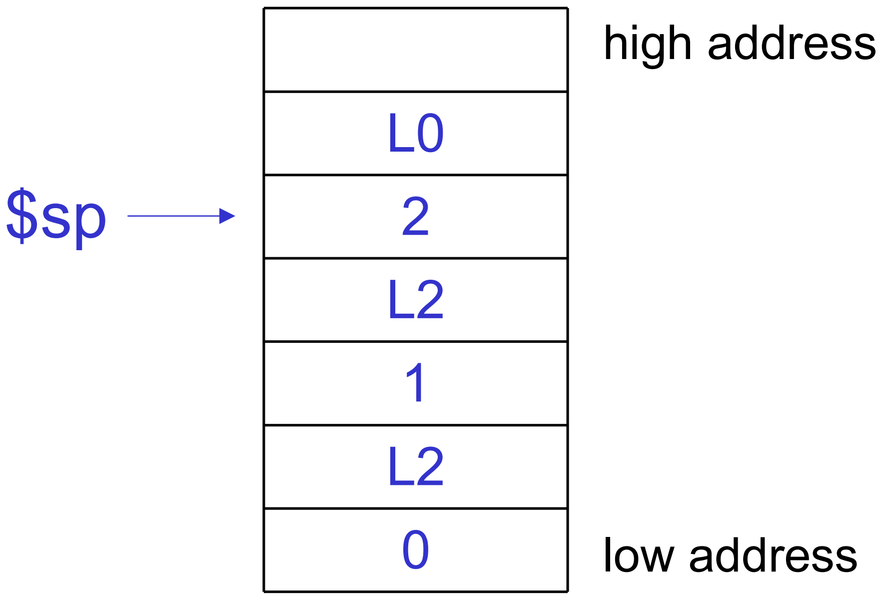
a0=1 t0=1 ra=L2 v0=1



Example 2 – non-leaf procedure

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
      addi  $v0, $zero, 1
      addi  $sp, $sp, 8
      jr    $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
L2: lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

a0=2   t0=1 ra=L0 v0=2



Saving Conventions

- Caller saved: Temp registers \$t0-\$t9 (the callee won't bother saving these, so save them if you care), \$ra (it's about to get over-written), \$a0-\$a3 (so you can put in new arguments)

- Callee saved: \$s0-\$s7 (these typically contain "valuable" data)

Saving Conventions

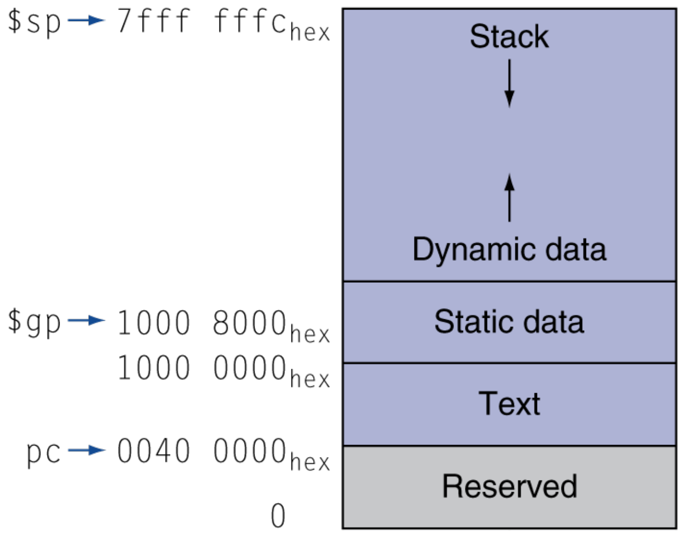
- Caller saved: Temp registers \$t0-\$t9 (the callee won't bother saving these, so save them if you care), \$ra (it's about to get over-written), \$a0-\$a3 (so you can put in new arguments)

- Callee saved: \$s0-\$s7 (these typically contain "valuable" data)

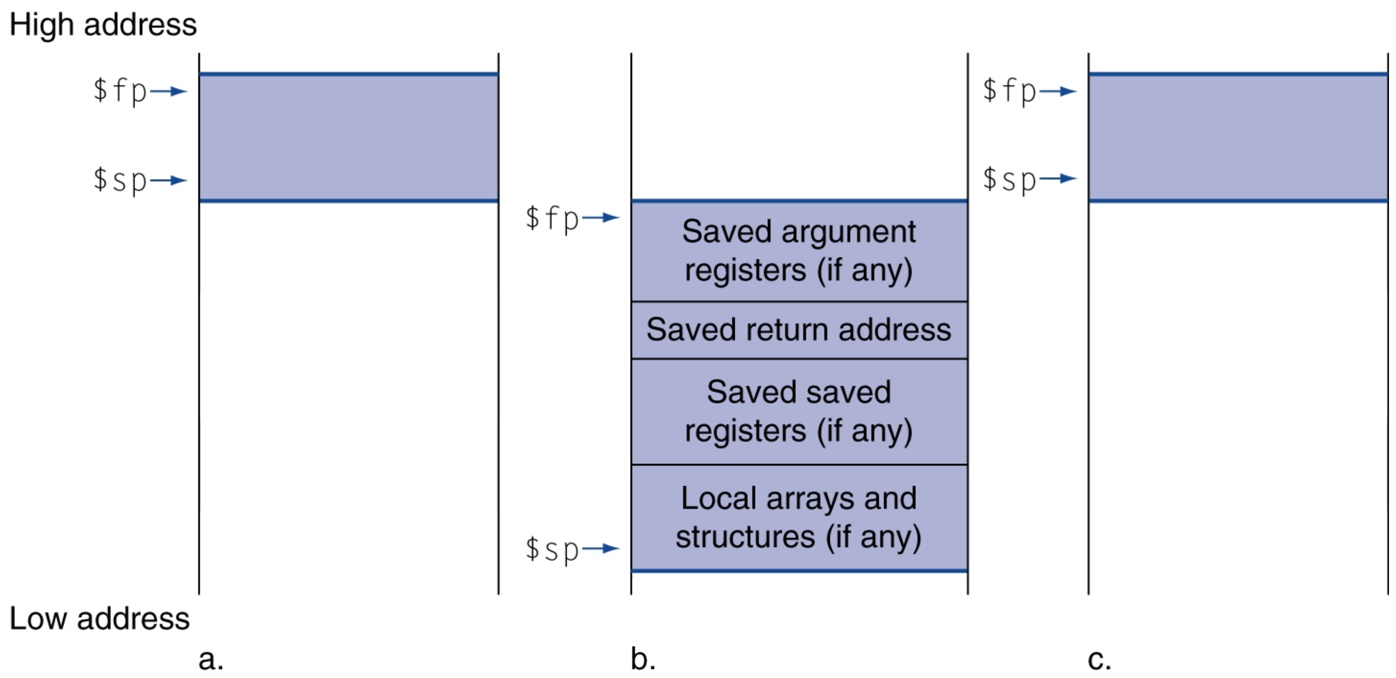
Memory Layout

冯诺依曼结构：指令和数据存在同一片内存
$gp->Static data:类似程序里的全局变量，一般$gp是放在Static data数据区一半的位置，这样往上4000hex往下4000hex
stack是从上往下写的，Dynamic data(heap)是从下往上写的。
$fp:用来界定stack的边界，位置固定不变

- 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



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

Homework #2

- Chapter 2：2.10, 2.12, 2.14, 2.16, 2.19, 2.23
- Due on Mar. 15

Check Yourself

- Given the importance of registers, what is the rate of increase in the number of registers in a chip over time?
  1. Very fast: They increase as fast as Moore's law, which predicts doubling the
  number of transistors on a chip every 18 months.
  2. Very slow: Since programs are usually distributed in the language of the
  computer, there is inertia in instruction set architecture, and so the number of registers increases only as fast as new instruction sets become viable

- What is the decimal value of this 64-bit two's complement number?
  1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
  1111 1111 1111 1000two
  1) –4ten
  2) –8ten
  3) –16ten
  4) 18,446,744,073,709,551,609ten

- What MIPS instruction does this represent? Choose from one of the four options below.

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

- 1. sub $t0, $t1, $t2
  2. add $t2, $t0, $t1
  3. sub $t2, $t1, $t0
  4. sub $t2, $t0, $t1

- Which operations can isolate a field in a word?
  1. AND
  2. A shift left followed by a shift right

- 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.

- Why does C provide two sets of operators for AND (& and &&) and two sets of operators for OR (| and ||), while MIPS doesn't?
  1. Logical operations AND and OR implement & and |, while conditional branches implement && and ||.
  2. The previous statement has it backwards: && and || correspond to logical operations, while & and | map to conditional branches.
  3. They are redundant and mean the same thing: && and || are simply inherited from the programming language B, the predecessor of C.