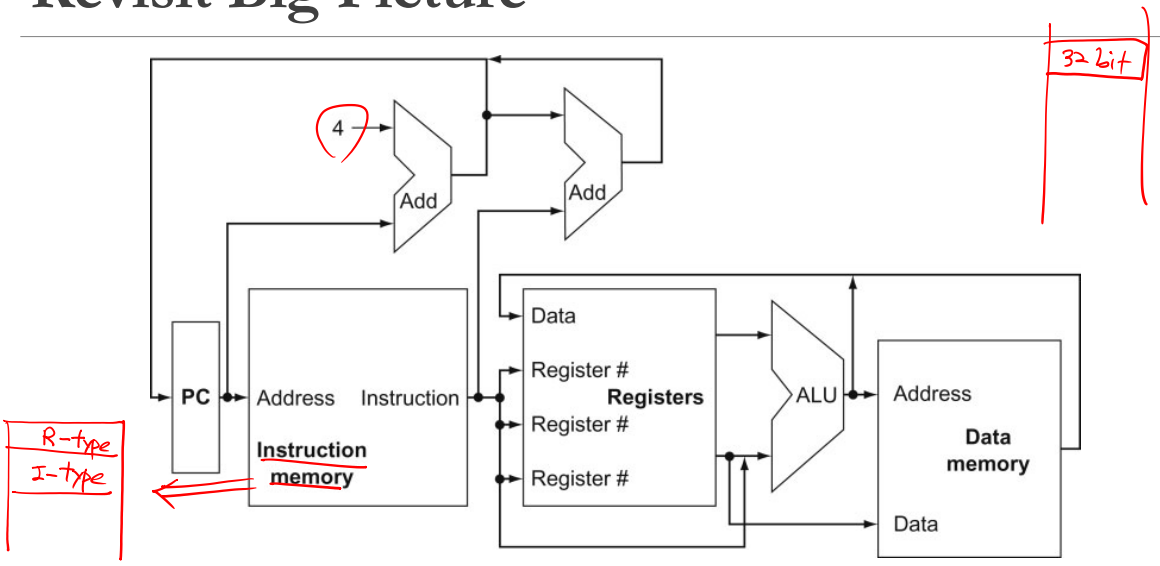
Revisit Big Picture



Compiling Loop Statements

Memory

- C code:
  - while (save[i] == k) i += 1;
  - i in $s3, k in $s5, address of save in $s6
- Compiled MIPS code:

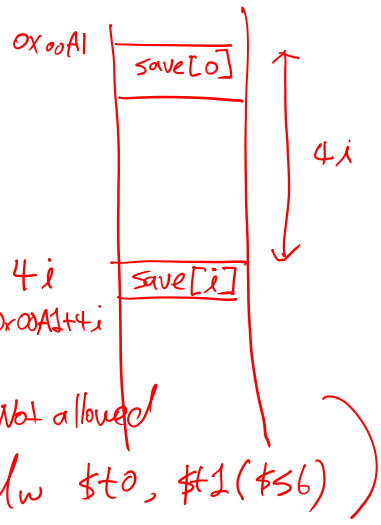
```
Loop:  sll  $t1, $s3, 2
       add  $t1, $t1, $s6
       lw   $t0, 0($t1)
       bne  $t0, $s5, Exit
       addi $s3, $s3, 1
       j    Loop
Exit: …
```

0x00A1

$t1 = 4i

$t1 = 0x00A1+4i

srl / sra



More Conditional Operations

- Set result to 1 if a condition is true
  - Otherwise, set to 0
- 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

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

Signed vs. Unsigned

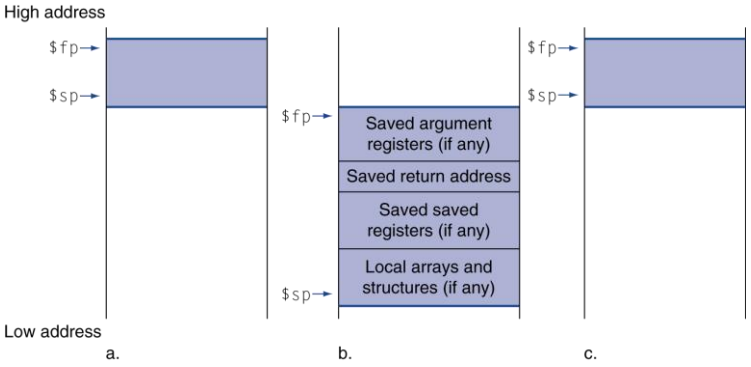
- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - slt $t0, $s0, $s1 # signed
    - –1 < +1 ➔ $t0 = 1
  - sltu $t0, $s0, $s1 # unsigned
    - +4,294,967,295 > +1 ➔ $t0 = 0

Procedure Calling

- Steps required
  1. Main routine (caller) place parameters in registers where the procedure (callee) can access them
     - $a0 – $a3: arguments (reg's 4 – 7)
  2. Caller transfers control to the callee (jal Dest)
  3. Callee acquires the storage resources needed
  4. Callee performs the operations
  5. Callee places result in register where the caller can access it
     - $v0, $v1: result values (reg's 2 and 3)
  6. Callee returns cotrol to the caller (jr $ra)

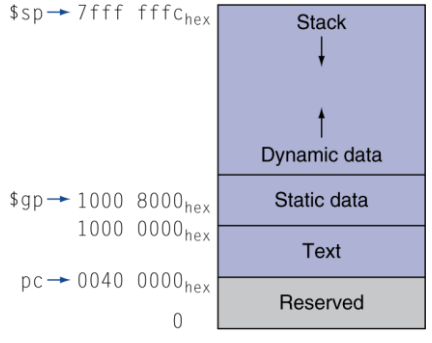
Local Data on the Stack

- A procedure frame (aka activation record) contains the procedure's saved registers and local variables.
  - The frame pointer ($fp) points to the first word of the procedure frame.
  - $fp is initialized using $sp on a call and $sp is restored using $fp on a return.



Memory Layout

- 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

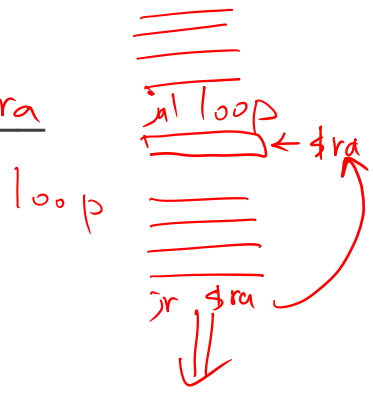


Register Usage

- $a0 – $a3: arguments (reg's 4 – 7)
- $v0, $v1: result values (reg's 2 and 3)
- $t0 – $t9: temporaries
  - Can be overwritten by callee
- $s0 – $s7: saved
  - Must be saved/restored by callee
- $gp: global pointer for static data (reg 28)
- $sp: stack pointer (reg 29)
- $fp: frame pointer (reg 30)
- $ra: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link
  ```
  jal ProcedureLabel
  ```
  - Address of following instruction put in $ra
  - Jumps to target address
- Procedure return: jump register
  ```
  jr $ra
  ```
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

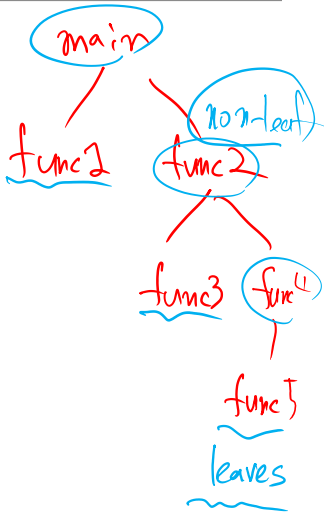


Leaf Procedure Example

- C code:

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

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



Leaf Procedure Example

- MIPS code:

`leaf_example:`	
`addi $sp, $sp, -4`	Save $s0 on stack
`sw $s0, 0($sp)`	
`add $t0, $a0, $a1`	Procedure body
`add $t1, $a2, $a3`	
`sub $s0, $t0, $t1`	
`add $v0, $s0, $zero`	Result
`lw $s0, 0($sp)`	Restore $s0
`addi $sp, $sp, 4`	
`jr $ra`	Return

- C code:

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

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

9/12