Lecture 7

CMPEN 331

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:

The first step is to load save[i] into a temporary register, we need to have its address first. We have to multiply the index i by 4.

```
      Loop: sll $t1, $s3, 2        # Temp register $t1= i*4
#To get the address of save[i], we need to add $t1 and
  the
#base of save in $s6
      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
  Exit: …
```

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
```

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

In Support of Branch Instructions

- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1    then
                      # $t0 = 1         else
                      # $t0 = 0
```

- Instruction format (R format):

0	16	17	8		0x24

```
slti $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltui $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

Aside: More Branch Instructions

- Can use `slt`, `beq`, `bne`, and the fixed value of 0 in register `$zero` to create other conditions
  - less than `blt $s1, $s2, Label`

```
slt   $at, $s1, $s2        #$at set to 1 if $s1 < $s2
bne   $at, $zero, Label
```

  - less than or equal to `ble $s1, $s2, Label`
  - greater than `bgt $s1, $s2, Label`
  - great than or equal to `bge $s1, $s2, Label`

- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - That is why, the assembler needs a reserved register (`$at`)

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

Procedure Calling

- Steps required
  1. Place parameters in registers
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call

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 (calling program)
- \$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)

Register Specifications

- $t0 - $t9: temporary register that are not preserved by the callee (called procedure) on a procedure call

- $s0 - $s7: saved registers that must be preserved on a procedure call (if used, the callee saves and restore them)

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`

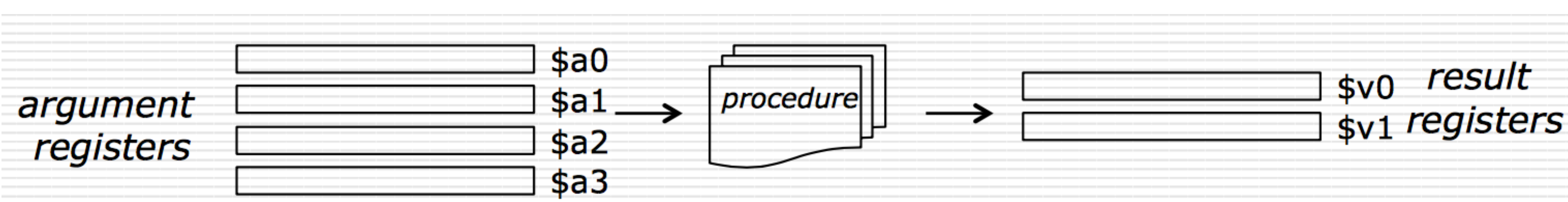
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code: (Leaf procedure: procedures that don't call others)

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

  - Arguments g, h, i, 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
   add  $t1, $a2, $a3       Procedure body
   sub  $s0, $t0, $t1
   add  $v0, $s0, $zero     Result
   lw   $s0, 0($sp)
   addi $sp, $sp, 4         Restore $s0
   jr   $ra                 Return back to
                            calling routine
```

Lecture 8

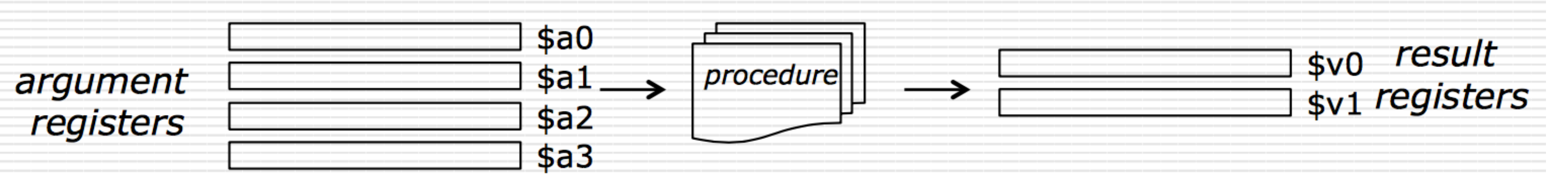
CMPEN 331

Leaf Procedure Example

- C code: (Leaf procedure: procedures that don’t call others)

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

  - Arguments g, h, i, 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` `sw   $s0, 0($sp)`	Save $s0 on stack
`add  $t0, $a0, $a1` `add  $t1, $a2, $a3` `sub  $s0, $t0, $t1`	Procedure body
`add  $v0, $s0, $zero`	Result
`lw   $s0, 0($sp)` `addi $sp, $sp, 4`	Restore $s0
`jr   $ra`	Return back to calling routine

Non-Leaf Procedures

- 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);
}
```

  - Argument n in $a0
  - Result in $v0

Steps for Calling a Procedure

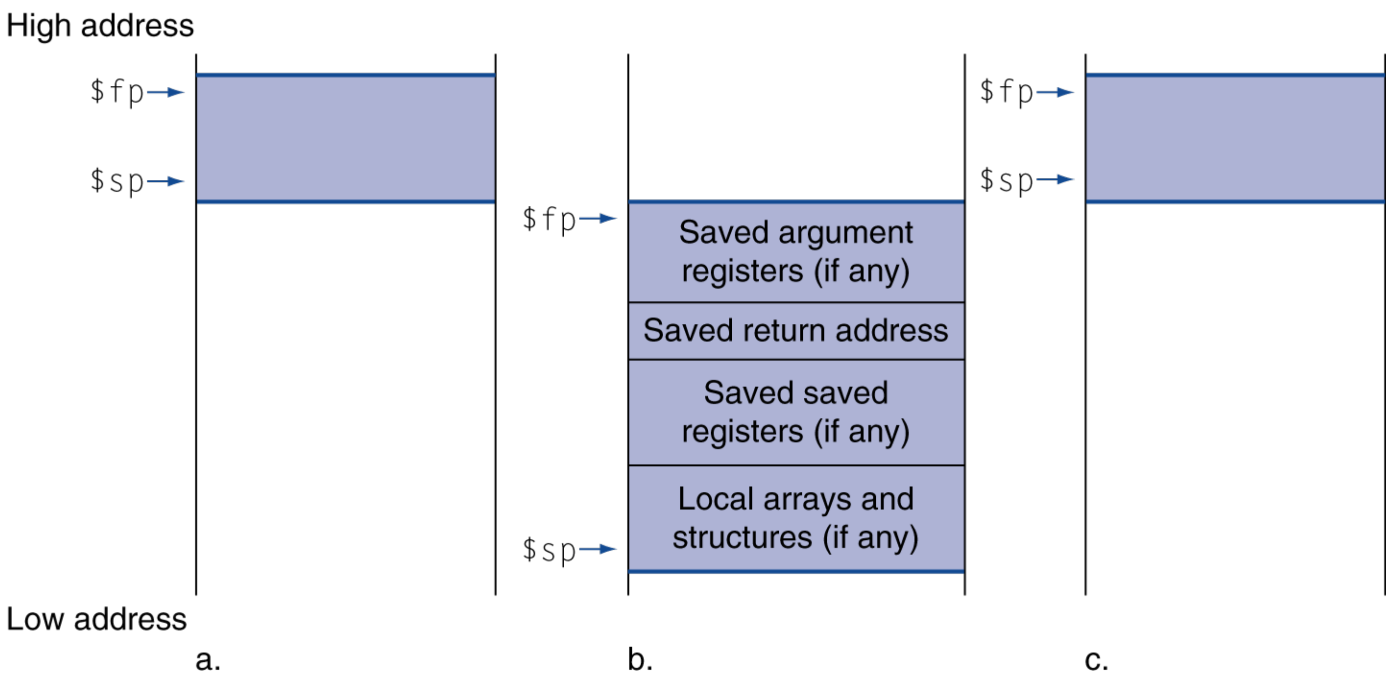
- Save necessary values onto stack
- Assign arguments if any
- jal call
- Restore values from stack

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
```

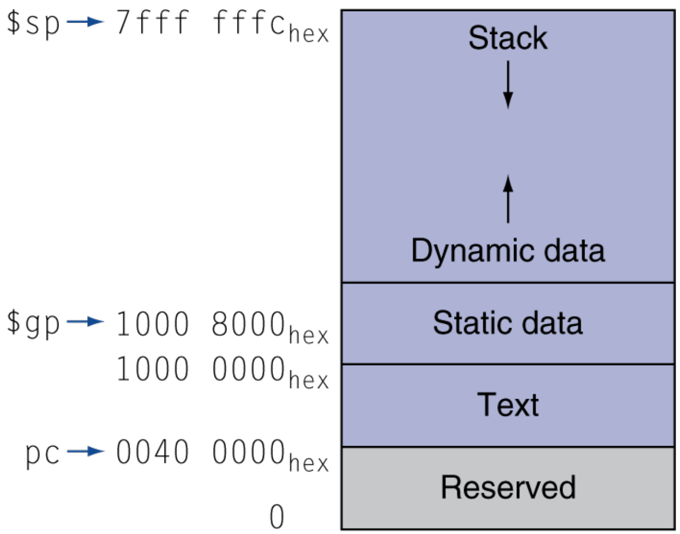
Local Data on the Stack



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

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



Byte Operations

- Could use bitwise operations
- MIPS byte load/store
- `lb`:load a byte from memory, placing it in the rightmost 8 bits of a register

```
lb rt, offset(rs) # read byte from source

lbu rt, offset(rs)

sb rt, offset(rs)
```

String Copy Example

- C code:
  - Null-terminated string

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

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- 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
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
- lui: load upper immediate

```
lui rt, constant
```

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0
- Load 32 bit constant into register $s0

0000 0000 0011 1101	0000 1001 0000 0000
61 in decimal	2304 in decimal

Instruction	Upper 16 bits	Lower 16 bits
`lui $s0, 61`	0000 0000 0011 1101	0000 0000 0000 0000
`ori $s0, $s0, 2304`	0000 0000 0111 1101	0000 1001 0000 0000

Jump Addressing

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction

op	address
6 bits	26 bits

Branch Addressing

- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- PC (program counter)-relative addressing
  - Target address = PC + branch address

Target Addressing Example

- Loop code from earlier example
  - Assume Loop at location 80000

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

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

```
        beq $s0,$s1, L1
                ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```