COMPUTER ORGANIZATION (IS F242)

LECT 27: MIPS ARCHITECTURE

String Copy Example

- C code (naïve):
  - 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

```
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
```

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function. [int is 4 bytes]
- Swap procedure (leaf)

```
void swap(int v[], int k)
{
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

  - v in $a0, k in $a1, temp in $t0

The Procedure Swap

```
swap: sll $t1, $a1, 2   # $t1 = k * 4
      add $t1, $a0, $t1 # $t1 = v+(k*4)
                        #   (address of v[k])
      lw $t0, 0($t1)    # $t0 (temp) = v[k]
      lw $t2, 4($t1)    # $t2 = v[k+1]
      sw $t2, 0($t1)    # v[k] = $t2 (v[k+1])
      sw $t0, 4($t1)    # v[k+1] = $t0 (temp)
      jr $ra            # return to calling routine
```

The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i – 1;
         j >= 0 && v[j] > v[j + 1];
         j -= 1) {
      swap(v,j);
    }
  }
}
```

  - v in $a0, n in $a1, i in $s0, j in $s1

The Procedure Body

```
          move $s2, $a0           # save $a0 into $s2
          move $s3, $a1           # save $a1 into $s3
          move $s0, $zero         # i = 0
for1tst:  slt  $t0, $s0, $s3      # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
          beq  $t0, $zero, exit1  # go to exit1 if $s0 ≥ $s3 (i ≥ n)
          addi $s1, $s0, –1       # j = i – 1
for2tst:  slti $t0, $s1, 0        # $t0 = 1 if $s1 < 0 (j < 0)
          bne  $t0, $zero, exit2  # go to exit2 if $s1 < 0 (j < 0)
          sll  $t1, $s1, 2        # $t1 = j * 4
          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
          lw   $t3, 0($t2)        # $t3 = v[j]
          lw   $t4, 4($t2)        # $t4 = v[j + 1]
          slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
          beq  $t0, $zero, exit2  # go to exit2 if $t4 ≥ $t3
          move $a0, $s2           # 1st param of swap is v (old $a0)
          move $a1, $s1           # 2nd param of swap is j
          jal  swap               # call swap procedure
          addi $s1, $s1, –1       # j –= 1
          j    for2tst            # jump to test of inner loop
exit2:    addi $s0, $s0, 1        # i += 1
          j    for1tst            # jump to test of outer loop
```

move $s2/$s3	Move params
move $s0 … slt	Outer loop
beq … beq exit2	Inner loop
move $a0 … jal swap	Pass params & call
addi $s1 … j for2tst	Inner loop
exit2 … j for1tst	Outer loop

The Full Procedure

```
sort:    addi $sp,$sp, -20       # make room on stack for 5 registers
         sw $ra, 16($sp)         # save $ra on stack
         sw $s3,12($sp)          # save $s3 on stack
         sw $s2, 8($sp)          # save $s2 on stack
         sw $s1, 4($sp)          # save $s1 on stack
         sw $s0, 0($sp)          # save $s0 on stack
         …                       # procedure body
         …
         exit1: lw $s0, 0($sp)   # restore $s0 from stack
         lw $s1, 4($sp)          # restore $s1 from stack
         lw $s2, 8($sp)          # restore $s2 from stack
         lw $s3,12($sp)          # restore $s3 from stack
         lw $ra,16($sp)          # restore $ra from stack
         addi $sp,$sp, 20        # restore stack pointer
         jr $ra                  # return to calling routine
```

Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

Example: Clearing and Array

<pre>clear1(int array[], int size) { int i; for (i = 0; i < size; i += 1) array[i] = 0; }</pre>	<pre>clear2(int *array, int size) { int *p; for (p = &array[0]; p < &array[size]; p = p + 1) *p = 0; }</pre>
<pre> add $t0,$zero, $zero # i = 0 loop1: sll $t1,$t0,2 # $t1 = i * 4 add $t2,$a0,$t1 # $t2 = # &array[i] slt $t3,$t0,$a1 # $t3 = # (i < size) beq $t3,$zero,loop2 # if (…) # goto loop1 sw $zero, 0($t2) # array[i] = 0 addi $t0,$t0,1 # i = i + 1 j loop1 loop2:</pre>	<pre> add $t0,$a0,$zero # p = & array[0] sll $t1,$a1,2 # $t1 = size * 4 add $t2,$a0,$t1 # $t2 = # &array[size] loop3: slt $t3,$t0,$t2 # $t3 = #(p<&array[size]) beq $t3,$zero,loop4 # if (…) # goto loop2 sw $zero,0($t0) # Memory[p] = 0 addi $t0,$t0,4 # p = p + 4 j loop3 Loop4:</pre>

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer