MIPS32 Assembly Language Basics
(doing arithmetic)

- To do following C++ expression in MIPS:

  a = b + c + d – e
- Translate it into multiple instructions:

  (assume **a**, **b**, **c**, **d** & **e** are in **$t1**, **$t2**, **$t3**, **$t4** & **$t5**)

```
add $t0, $t2, $t3
add $t0, $t0, $t4
sub $t1, $t0, $t5
```

- Common pattern:
  - A line of HLL code translates into several lines of AL code

MIPS32 Assembly Language Basics
(doing arithmetic)

- C++ expression:

  a = (b + c) – (d + e)
- MIPS instructions:

  (assume **a**, **b**, **c**, **d** & **e** are in **$t1**, **$t2**, **$t3**, **$t4** & **$t5**)

```
add $t0, $t2, $t3
add $t1, $t4, $t5
sub $t1, $t0, $t1
```

- Remark:
  - Convenient/easy to have spare registers for use as temporaries

MIPS32 Assembly Language Basics
(doing arithmetic: recall from earlier)

- C++ expression:

```
A = B + C * D – E + F + A
```

- Equivalent instructions:

```
mult   T,C,D   ;T = C*D
add    T,T,B   ;T = B+C*D
sub    T,T,E   ;T = B+C*D-E
add    T,T,F   ;T = B+C*D-E+F
add    A,T,A   ;A = B+C*D-E+F+A
```

- Slide #18 of **006 ComputerOrg&DesignOverview01**
- Illustrating 3-address machines (to which MIPS belongs)
- Should be straightforward to translate above into actual MIPS code

MIPS32 Assembly Language Basics
(doing arithmetic – Textbook e.g., §2.4, p.13)

- C++ expression:

```
$a2 = sqrt($a0*$a0 + $a1*$a1)
```

- Equivalent instructions:
(assume results of computation don't exceed 32 bits)
(assume there's library function **`sqrt`** that we can call: receives argument through **`$a0`** and returns result through **`$v0`**)

```
mult $a0, $a0       # square $a0
mflo $t0            # $t0 = lower 32-bits of product
mult $a1, $a1       # square $a1
mflo $t1            # $t1 = lower 32-bits of product
add $a0, $t0, $t1   # $a0 = $t0 + $t1
jal sqrt            # call square root function
move $a2, $v0       # result of sqrt returned in $v0
```

MIPS32 Assembly Language Basics
(doing arithmetic – Textbook e.g., §2.4, p.13)

- C++ expression:

```
$a2 = sqrt($a0*$a0 + $a1*$a1)
```

- Equivalent instructions:
  (assume results of computation don't exceed 32 bits)
  (assume there's library function **`sqrt`** that we can call: receives argument through **`$a0`** and returns result through **`$v0`**)

```
mul $t0, $a0, $a0   # $t0 = lower 32-bits of $a0 sq'd
mul $t1, $a1, $a1   # $t1 = lower 32-bits of $a1 sq'd
add $a0, $t0, $t1   # $a0 = $t0 + $t1
jal sqrt            # call square root function
add $a2, $v0, $0    # result of sqrt returned in $v0
```

using **`mul`** and **`add`** instead of **`mult/mflo`** and **`move`**

MIPS32 Assembly Language Basics
(doing arithmetic – Textbook e.g., §2.4, p.14)

- C++ expression:

```
$v0 = PI * $t8 * $t8
```

- Equivalent instructions:
  (assume results of computation don't exceed 32 bits)

```
li $t0, 31415       # PI scaled up by 10,000
mult $t8, $t8       # radius squared
mflo $t1            # $t1 = lower 32-bits of product
mult $t1, $t0       # multiply $t1 by PI
mflo $v0            # $v0 = lower 32-bits of product
li $t1, 10000       # load scale factor 10,000
div $v0, $t1        # divide $v0 by scale factor
mflo $v0            # $v0 = truncated integer result
```

MIPS32 Assembly Language Basics
(doing arithmetic – Textbook e.g., §2.4, p.14)

- C++ expression:

```
$v0 = PI * $t8 * $t8
```

- Equivalent instructions:
  (assume results of computation don't exceed 32 bits)

```
li $t0, 31415       # PI scaled up by 10,000
mul $t1, $t8, $t8   # $t1 = lower 32-bits of $t8 sq'd
mul $v0, $t1, $t0   # $v0 = lower 32-bits of $t1 * PI
li $t1, 10000       # load scale factor 10,000
div $v0, $t1        # divide $v0 by scale factor
mflo $v0            # $v0 = truncated integer result
```

using `mul` instead of `mult/mflo`
pseudoinstruction to replace last 2 statements: `div $v0, $v0, $t1`

MIPS32 Assembly Language Basics
(accessing array - "fixed-offset" case)

```
...
############################ data segment ################################
        .data
list:   .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7
############################ code segment ################################
        .text
        .globl main
main:
        ...

        la $t3, list      # put address of list into $t3
        lw $t4, 0($t3)    # put value of list[0] into $t4
        lw $t5, 4($t3)    # put value of list[1] into $t5
        lw $t6, 8($t3)    # put value of list[2] into $t6

        ...
```

MIPS32 Assembly Language Basics
(accessing array - "variable-offset" case)

```
...
############################ data segment ################################
        .data
list:   .word 3, 0, 1, 2, 6, -2, 4, 7, 3, 7
############################ code segment ################################
        .text
        .globl main
main:
        ...
        addi $t7, $0, 0     # initialize counter $t7 to 0
        addi $t6, $0, 10    # initialize end_counter $t6 to 10
        la $t0, list        # $t0 has address of 1st element
loop:   lw $a0, 0($t0)      # $a0 has integer $t0 points at
        ####### code processing array element (value now in $a0) #######
        addi $t7, $t7, 1    # increment counter $t7
        beq $t7, $t6, done  # go to done if all elements are processed
        addi $t0, $t0, 4    # $t0 has address of next element
        j loop              # repeat processing for next element
done:
        ...
```


MIPS32 Assembly Language Basics
(caveat when accessing memory)

- Common pitfall
  - Wrongly think that sequential word addresses differ by 1
  - Wrongly think that address of next word can be found by incrementing address in register by 1
    - Instead of by word size in bytes
  - Don't let *pointer arithmetic* of C/C++ fool you
- Also to remember
  - For `lw` and `sw`, sum of base address and offset must be *multiple of 4* to be *word aligned*
  - Be wary of bus error

MIPS32 AL Basics (doing selection)

```
if (i == j)
    ++i;
--j;
```

```
        if (i != j) goto endif;
        ++i;
endif:
        --j;
```

```
# assume i & j are in $t1 & $t2
        bne $t1, $t2, endif       # branch if !(i == j)
        addi $t1, $t1, 1          # ++i
endif:
        addi $t2, $t2, -1         # --j
```


MIPS32 AL Basics (doing selection)

```
if (i == j)
    ++i;
else
    --j;
j += i;
```

```
        if (i != j) goto else;
        ++i;
        goto endif;
else:
        --j;
endif:
        j += i;
```

```
# assume i & j are in $t1 & $t2
        bne $t1, $t2, else        # branch if !(i == j)
        addi $t1, $t1, 1          # ++i
        j endif                   # jump over else (DON'T forget this!!!)
else:   addi $t2, $t2, -1         # --j
endif:
        add $t2, $t2, $t1         # j += i
```

MIPS32 AL Basics (doing selection)

```
if (i == j && i == k)
   ++i;
else
   --j;
j = i + k;
```

```
         if (i != j) goto else;
         if (i != k) goto else;
         ++i;
         goto endif;
else:
         --j;
endif:
         j = i + k;
```

```
# assume i, j & k are in $t1, $t2 & $t3
        bne $t1, $t2, else       # branch if !(i == j)
        bne $t1, $t3, else       # branch if !(i == k)
        addi $t1, $t1, 1         # ++i
        j endif                  # jump over else
else:   addi $t2, $t2, -1        # --j
endif:
        add $t2, $t1, $t3        # j = i + k
```


MIPS32 AL Basics (doing selection)

```
if (i == j || i == k)
   ++i;
else
   --j;
j = i + k;
```

```
         if (i == j) goto good;
         if (i != k) goto else;
good:
         ++i;
         goto endif;
else:
         --j;
endif:
         j = i + k;
```

```
# assume i, j & k are in $t1, $t2 & $t3
        beq $t1, $t2, good       # branch if (i == j)
        bne $t1, $t3, else       # branch if !(i == k)
good:   addi $t1, $t1, 1         # ++i
        j endif                  # jump over else
else:   addi $t2, $t2, -1        # --j
endif:
        add $t2, $t1, $t3        # j = i + k
```

MIPS32 AL Basics (doing selection)

```
if (i == j && i == k)
   ++i;
else
   --j;
j = i + k;
```

```
        cumu = (i == j);
        next = (i == k);
        cumu = cumu && next;
        if ( ! cumu) goto else;
        ++i;
        goto endif;
else:
        --j;
endif:
        j = i + k;
```

```
# assume next, i, j, k, cumu are respectively in $t0, $t1, $t2, $t3, $t9
        seq $t9, $t1, $t2         # cumu = (i == j) ? 1 : 0
        seq $t0, $t1, $t3         # next = (i == k) ? 1 : 0
        and $t9, $t9, $t0         # cumu = cumu && next
        beqz $t9, else            # if ( ! cumu) goto else
        addi $t1, $t1, 1          # ++i
        j endif                   # goto endif
else:   addi $t2, $t2, -1         # --j
endif:
        add $t2, $t1, $t3         # j = i + k
```


MIPS32 AL Basics (doing selection)

```
if ( (c >= '0' && c <= '9') ||
     (c >= 'A' && c <= 'Z') ||
     (c >= 'a' && c <= 'z')
   )
   cout << "c is alnum";
else
   cout << "c is not alnum";
cout << endl;
```

brute-force
&&
unoptimized

```
        exp1 = (c >= '0');
        exp2 = (c <= '9');
        cumu = exp1 && exp2;
        exp1 = (c >= 'A');
        exp2 = (c <= 'Z');
        exp1 = exp1 && exp2;
        cumu = cumu || exp1;
        exp1 = (c >= 'a');
        exp2 = (c <= 'z');
        exp1 = exp1 && exp2;
        cumu = cumu || exp1;
        if ( ! cumu) goto else;
        cout << "c is alnum";
        goto endif;
else:
        cout << "c is not alnum";
endif:
        cout << endl;
```

```
# assume c, exp1, exp2, cumu are respectively in $t0, $t1, $t2, $t9
        sge $t1, $t0, '0'     # exp1 = (c >= '0')
        sle $t2, $t0, '9'     # exp2 = (c <= '9')
        and $t9, $t1, $t2     # cumu = exp1 && exp2
        sge $t1, $t0, 'A'     # exp1 = (c >= 'A')
        sle $t2, $t0, 'Z'     # exp2 = (c <= 'Z')
        and $t1, $t1, $t2     # exp1 = exp1 && exp2
        or $t9, $t9, $t1      # cumu = cumu || exp1
        sge $t1, $t0, 'a'     # exp1 = (c >= 'a')
        sle $t2, $t0, 'z'     # exp2 = (c <= 'z')
        and $t1, $t1, $t2     # exp1 = exp1 && exp2
        or $t9, $t9, $t1      # cumu = cumu || exp1
        beqz $t9, else        # if ( ! cumu) goto else
        (do syscall here)     # cout << "c is alnum"
        j endif               # goto endif
else:   (do syscall here)     # cout << "c is not alnum"
endif:
        (do syscall here)     # cout << endl
```

MIPS32 Assembly Language Basics
(doing selection)

```
switch (i)
{
   case 1: ++i;
   case 2: i += 2;
           break;
   case 3: i += 3;
}
```

```
# assume i is in $t1
                 addi $t4, $0, 1          # set $t4 (temp holder) to 1
                 bne $t1, $t4, c2_cond    # case 1 false, branch to case 2 cond
                 j c1_body                # case 1 true, branch to case 1 body
c2_cond:         addi $t4, $0, 2          # set $t4 (temp holder) to 2
                 bne $t1, $t4, c3_cond    # case 2 false, branch to case 3 cond
                 j c2_body                # case 1 true, branch to case 2 body
c3_cond:         addi $t4, $0, 3          # set $t4 (temp holder) to 3
                 bne $t1, $t4, all_done   # case 3 false, branch to all_done
                 j c3_body                # case 1 true, branch to case 3 body
c1_body:         addi $t1, $t1, 1         # case 1 body: ++i
c2_body:         addi $t1, $t1, 2         # case 2 body: i += 2
                 j all_done               # break
c3_body:         addi $t1, $t1, 3         # case 3 body: i += 3
all_done:
```


MIPS32 Assembly Language Basics
(doing selection)

```
         $s0 = 32;
top:     cout << "Input a value from 1 to 3: ";
         cin >> $v0;
         switch ($v0)
         {
            case 1: $s0 = $s0 << 1;
                    break;
            case 2: $s0 = $s0 << 2;
                    break;
            case 3: $s0 = $s0 << 3;
                    break;
            default: goto top;
         }
         cout << $s0;
```

```
                 .data
                 .align 2
jumptable:       .word top, case1, case2, case3
prompt :         .asciiz "\n\n Input a value from 1 to 3: "
                 .text
top:
...
```

MIPS32 AL Basics (doing selection)

```
top:
                 li $v0, 4               # code to print a string
                 la $a0, prompt
                 syscall
                 li $v0, 5               # code to read an integer
                 syscall
                 blez $v0, top           # default for less than one
                 li $t3, 3
                 bgt $v0, $t3, top       # default for greater than 3
                 la $a1, jumptable       # load address of jumptable
                 sll $t0, $v0, 2         # compute word offset
                 add $t1, $a1, $t0       # form a pointer into jumptable
                 lw $t2, 0($t1)          # load an address from jumptable
                 jr $t2                  # jump to specific case “switch”
case1:           sll $s0, $s0, 1         # shift left logical one bit
                 b output
case2:           sll $s0, $s0, 2         # shift left logical two bits
                 b output
case3:           sll $s0, $s0, 3         # shift left logical three bits
output:          li $v0, 1               # code to print an integer is 1
                 move $a0, $s0           # pass argument to system in $a0
                 syscall                 # output results
```


MIPS32 Assembly Language Basics (remember DeMorgan)

- When translating pseudocode to assembly language, we often have to *branch on* ***negated condition***
- Good to know about *negation of various conditions*:

Condition	*Negated Condition*
`x > y`	`x <= y`
`x >= y`	`x < y`
`x < y`	`x >= y`
`x <= y`	`x > y`
`<cond1> && <cond2>`	`! <cond1> || ! <cond2>`
`<cond1> || <cond2>`	`! <cond1> && ! <cond2>`

MIPS32 Assembly Language Basics
(doing loops)

```
while (i < k)
{
   ++k;
   i = i * 2;
}
```

```
begin:   if (i < k)
         {
            ++k;
            i = i * 2;
            goto begin;
         }
```

```
# assume i & k are in $t1 & $t3
begin:            bge $t1, $t3, end         # branch if !(i < k)
                  addi $t3, $t3, 1          # ++k
                  add $t1, $t1, $t1         # i = i * 2
                  j begin
end:
```


MIPS32 Assembly Language Basics
(doing loops)

```
for (<init>; <cond>; <update>)
{
   <loop_body>
}
```

```
         <init>
         for (; <cond>; )
         {
            <loop_body>
            <update>
         }
```

```
         <init>
begin:   if ( <cond> )
         {
            <loop_body>
update:     <update>
            goto begin
         }
```

for what special case do we need to jump to update?

(e.g.)

```
k = 0;
for (i = 1; i <= j; ++i)
   k = k + i;
```

```
         k = 0;
         i = 1;
begin:   if (i <= j)
         {
            k = k + i;
            ++i;
            goto begin;
         }
```

```
# assume i, j & k are in $t1, $t2 & $t3
         add $t3, $zero, $zero
         addi $t1, $zero, 1
begin:   bgt $t1, $t2, end
         add $t3, $t3, $t1
         addi $t1, $t1, 1
         j begin
end:
```

MIPS32 Assembly Language Basics

(last e.g. modified)

(doing loops)

```
for (<init>; <cond>; <update>)
{
    <loop_body>
}
```

```
        <init>
        for (; <cond>; )
        {
            <loop_body>
            <update>
        }
```

```
        <init>
begin:  if ( <cond> )
        {
            <loop_body>
update:     <update>
            goto begin
        }
```

for what special case do we need to jump to update?

```
k = 0;
for (i = 1; i <= j; ++i)
{
   if (i == k) continue;
   k = k + i;
}
```

```
          k = 0;
          i = 1;
begin:    if (i <= j)
          {
             if (i == k) goto update;
             k = k + i;
update:      ++i;
             goto begin;
          }
```

```
# assume i, j & k are in $t1, $t2 & $t3
          add $t3, $zero, $zero
          addi $t1, $zero, 1
begin:    bgt $t1, $t2, end
          beq $t1, $t3, update
          add $t3, $t3, $t1
update:   addi $t1, $t1, 1
          j begin
end:
```


MIPS32 Assembly Language Basics

(doing loops)

```
do
{
    ++k;
    i = i * 2;
}
while (i < k);
```

```
          {
begin:        ++k;
              i = i * 2;
          }
          if (i < k)
              goto begin;
```

```
# assume i & k are in $t1 & $t3
begin:         addi $t3, #t3, 1        # ++k
               add $t1, $t1, $t1       # i = i * 2
               bge $t1, $t3, end       # end loop if !(i < k)
               j begin
end:
```

MIPS32 Assembly Language Basics
(doing loops)

Textbook example (§2.7, p.15)

```
$a0 = 0;
for ( $t0 = 10; $t0 > 0; $t0 = $t0 - 1)
{
   $a0 = $a0 + $t0;
}
```

```
                 li $a0, 0                 # $a0 = 0
                 li $t0, 10                # initialize loop counter to 10
loop:
                 add $a0, $a0, $t0
                 addi $t0, $t0, -1         # decrement loop counter
                 bgtz $t0, loop            # if ($t0 > 0) branch to loop
```

What don't you like about it?

MIPS32 Assembly Language Basics
(doing loops)

Textbook example (§2.7, p.15)

```
$a0 = 0;
for ( $t0 = 10; $t0 > 0; $t0 = $t0 - 1)
{
   $a0 = $a0 + $t0;
}
```

```
                 li $a0, 0                 # $a0 = 0
                 li $t0, 10                # initialize loop counter to 10
                 j loop_cond               # test condition 1st like it should
loop:
                 add $a0, $a0, $t0
                 addi $t0, $t0, -1         # decrement loop counter
loop_cond:       bgtz $t0, loop            # if ($t0 > 0) branch to loop
```

semantically correct @ cost of 1 instruction

MIPS32 Assembly Language Basics

(testing conditions with set and logical instr$^{n}_{s}$)

set & logical operations → more general but less compact

```
if ($t0 < $t1) $v0 = 0;                    # if1
if ($t0 > $t1 && $t0 < $t2) $v1 = 1;       # if2
if ($t1 < $t2 || $t1 > $t3) $v0 = 2;       # if3
```

```
if1:                slt $a0, $t0, $t1         # $a0 set to 1 if $t0 < $t1 else 0
                    beq $a0, $0, if2          # skip to if2 if $a0 = 0
                    add $v0, $0, $0           # $v0 = 0
if2:                slt $a0, $t1, $t0         # $a0 set to 1 if $t1 < $t0 else 0
                    slt $a1, $t0, $t2         # $a1 set to 1 if $t0 < $t2 else 0
                    and $a0, $a0, $a1         # $a0 has 1 if above 2 true else 0
                    beq $a0, $0, if3          # skip to if3 if $a0 = 0
                    addi $v1, $0, 1           # $v1 = 1
if3:                slt $a0, $t1, $t2         # $a0 set to 1 if $t1 < $t2 else 0
                    slt $a1, $t3, $t1         # $a1 set to 1 if $t3 < $t1 else 0
                    or $a0, $a0, $a1          # $a0 has 1 if at least 1 true else 0
                    beq $a0, $0, end          # skip to end if $a0 = 0
                    addi $v0, $0, 2           # $v1 = 2
end:
```


MIPS32 Assembly Language Basics

(testing conditions with set and logical instr$^{n}_{s}$)

set & logical operations → good for complex compound cond$^{n}_{s}$

```
if ( ($t0 < $t1 && $t0 > $t2) || ($t1 < $t2 && $t1 > $t3) )
   $v0 = 1;
else
   $v0 = 2;
```

```
                    slt $a0, $t0, $t1         # $a0 set to 1 if $t0 < $t1 else 0
                    sgt $a1, $t0, $t2         # $a1 set to 1 if $t0 > $t2 else 0
                    and $a0, $a0, $a1         # $a0 has 1 if above 2 true else 0
                    bne $a0, $0, set1         # with ||, can short-circuit here
                    slt $a1, $t1, $t2         # $a1 set to 1 if $t1 < $t2 else 0
                    sgt $a2, $t1, $t3         # $a2 set to 1 if $t1 > $t3 else 0
                    and $a1, $a1, $a2         # $a1 has 1 if above 2 true else 0
                    or $a0, $a0, $a1          # $a0 has 1 if whole thing true else 0
                    beq $a0, $0, set2         # whole thing not true
set1:               addi $v0, $v0, 1
                    j done
set2:               addi $v0, $v0, 2
done:
```

Basics *Extra*

MIPS32 Assembly Language Basics *Extra*

(swap 2 registers with **no additional storage**)

- Say registers are **$t1** and **$t2**:

```
xor $t1, $t1, $t2
xor $t2, $t2, $t1
xor $t1, $t1, $t2
```

- Example:

```
$t1: 1101...0111
$t2: 0101...1010
$t1: 1000...1101   after 1st xor
$t2: 1101...0111   after 2nd xor
$t1: 0101...1010   after 3rd xor
```

MIPS32 Assembly Language Basics *Extra*

(avoid **in-loop jumps** when translating loops)

```
for (<init>; <test>; <update>)
{
    <body>
}
```

```
                 <init>
                 goto forTest;
forBody:         {
                     <body>
                     <update>
                 }
forTest:         if (<test>) goto forBody;
```

```
while (<test>)
{
    <body>
}
```

```
                 goto whileTest;
whileBody:       {
                     <body>
                 }
whileTest:       if (<test>) goto whileBody;
```

```
do
{
    <body>
}
while (<test>);
```

```
whileBody:       {
                     <body>
                 }
whileTest:       if (<test>) goto whileBody;
```


MIPS32 Assembly Language Basics *Extra*

(avoid **in-loop jumps** when translating loops)

```
sum = 0;
for ( i = 10; i > 0; --i)
{
    sum = sum + i;
}
```

```
                 li $a0, 0                 # $a0 is sum (init to 0)
                 li $t0, 10                # $t0 is i (init to 10)
begFor:          blez $t0, endFor          # end loop if (i <= 0)
                 add $a0, $a0, $t0         # sum = sum + i
                 addi $t0, $t0, -1         # --i
                 j begFor                  # do next iteration
endFor:
```

total instructions executed = **2 + 4*10 + 1 = 43**

Do you see this one?

MIPS32 Assembly Language Basics *Extra*

(avoid **in-loop jumps** when translating loops)

```
sum = 0;
for ( i = 10; i > 0; --i)
{
   sum = sum + i;
}
```

```
                 sum = 0;
                 i = 10;
                 goto forTest;
                 {
forBody:            sum = sum + i;
                    --i;
                 }
forTest:         if (i > 0) goto forBody;
```

```
                 li $a0, 0                  # $a0 is sum (init to 0)
                 li $t0, 10                 # $t0 is i (init to 10)
                 j forTest                  # go test condition
forBody:         add $a0, $a0, $t0          # sum = sum + i
                 addi $t0, $t0, -1          # --i
forTest:         bgtz $t0, forBody          # do for body if (i > 0)
```

total instructions executed = **3 + 3*10 = 33**

MIPS32 Assembly Language Basics *Extra*

(avoid **in-loop jumps** when translating loops)

```
sum = 0;
i = 10;
while (i > 0)
{
   sum = sum + i;
   --i;
}
```

```
                 li $a0, 0                  # $a0 is sum (init to 0)
                 li $t0, 10                 # $t0 is i (init to 10)
begWhile:        blez $t0, endWhile         # end loop if (i <= 0)
                 add $a0, $a0, $t0          # sum = sum + i
                 addi $t0, $t0, -1          # --i
                 j begWhile                 # do next iteration
endWhile:
```

total instructions executed = **2 + 4*10 + 1 = 43**

MIPS32 Assembly Language Basics *Extra*

(avoid **in-loop jumps** when translating loops)

```
sum = 0;
i = 10;
while (i > 0)
{
   sum = sum + i;
   --i;
}
```

```
             sum = 0;
             i = 10;
             goto whileTest;
             {
whileBody:      sum = sum + i;
                --i;
             }
whileTest:   if (i > 0) goto whileBody;
```

```
                  li $a0, 0              # $a0 is sum (init to 0)
                  li $t0, 10             # $t0 is i (init to 10)
                  j whileTest            # go test condition
whileBody:        add $a0, $a0, $t0      # sum = sum + i
                  addi $t0, $t0, -1      # --i
whileTest:        bgtz $t0, whileBody    # do while body if (i > 0)
```

total instructions executed = **3 + 3*10 = 33**

MIPS32 Assembly Language Basics *Extra*

(**pointer advantage** when traversing array)

```
void zeroOut1(int a[], int size)
{
    int i = 0;
    while (i < size)
    {
        a[i] = 0;
        ++i;
    }
}
```

```
void zeroOut2(int* a, int size)
{
    int* p = a;           // p = &a[0]
    while (p < a + size) // p < &a[size]
    {
        *p = 0;
        p++;
    }
}
```

```
zeroOut1:     move $t0, $0                # $t0 is i init to 0
              j whileTest                 # go test condition
whileBody:    sll $t1, $t0, 2             # $t1 has 4i
              add $t1, $t1, $a0           # $t1 now has address of a[i]
              sw $0, 0($t1)               # a[i] = 0
              addi $t0, $t0, 1            # ++i
whileTest:    blt $t0, $a1, whileBody     # do while body if i < size

              jr $ra
```

Why 6 and not 5?

total instructions executed = **6*size + 3**

MIPS32 Assembly Language Basics *Extra*

(pointer advantage when traversing array)

```
void zeroOut1(int a[], int size)
{
    int i = 0;
    while (i < size)
    {
        a[i] = 0;
        ++i;
    }
}
```

```
void zeroOut2(int* a, int size)
{
    int* p = a;          // p = &a[0]
    while (p < a + size) // p < &a[size]
    {
        *p = 0;
        p++;
    }
}
```

```
zeroOut2:      move $t0, $a0            # $t0 is p init to &a[0]
               sll $t1, $a1, 2          # $t1 has 4*size
               add $t1, $t1, $a0        # $t1 now has &a[size]
               j whileTest              # go test condition
whileBody:     sw $0, 0($t0)            # *p = 0
               addi $t0, $t0, 4         # p++ (pointer-arithmetically)
whileTest:     blt $t0, $t1, whileBody  # do while body if (p < &a[size])

               jr $ra
```

Why 4 and not 3?

total instructions executed = **4*size + 5**

MIPS32 Assembly Language Basics *Extra*

(things to note when optimizing/comparing code)

- # of lines of code (instructions) leaves much to be desired:
  - A pseudoinstruction (in general) incurs several *true* instructions
  - Different (true) instructions have different time/space costs
  - A not-in-loop instruction will only be executed once
  - An in-loop instruction will (in general) be executed many times
- Registers are premium commodity
  - Other things being equal, smaller register footprint = better
- Some spaghetti dishes are healthier/taste better than others
  - In the "goto" world of AL programmers, spaghetti is staple food
  - "healthier/tasted better" ⇔ better structured/more readable/…

MIPS32 Assembly Language Basics *Extra*
(address vs value)

- A MIPS register can hold any 32-bit quantity
  - That value can be a signed integer, an unsigned integer, a pointer (memory address), *etc.*
- If you write

  `add $t2, $t1, $t0`

  then **`$t0`** and **`$t1`** must contain values
- If you write

  `lw $t2, 0($t0)`

  then **`$t0`** must contain a memory address
- If you mix these up, trouble befalls!

MIPS32 Assembly Language Basics *Extra*
(don't let pointer arithmetic fool you)

- Common error made by many AL programmers:
  - Assuming address of next word can be found by incrementing address in register by 1 instead of by word size in bytes
- For **`lw`** and **`sw`** in particular:
  - Sum of base address and offset must be multiple of 4
    - (to be word aligned)
- Example 1:
  - C++: **`g = h + A[8];`**
  - MIPS assembly:
    - **`g`** → **`$s1`**, **`h`** → **`$s2`** and base address of **`A`** → **`$s3`** assumed

```
lw $t0, 32($s3)
add $s1, $s2, $t0
```

- Example 2:
  - C++: **`A[8] = h + A[5];`**
  - MIPS assembly:
    - **`h`** → **`$s2`** and base address of **`A`** → **`$s3`** assumed

```
lw $t0, 20($s3)
add $t0, $s2, $t0
sw $t0, 32($s3)
```