MIPS Assembly Language Programming: Part 1
Let's Learn More MIPS Instructions!

• So far we've seen:
  • Addition (add $d, $s, $t) and Subtraction (sub $d, $s, $t)
  • Load Immediate & Skip (lis $d) for loading constants
  • Jump Register (jr $s) to jump (set PC) to another memory location
  • Load Word (lw $t, i($s)) and Store Word (sw $t, i($s)) for memory access
  • The .word directive (for encoding non-instruction words in our program)

• Today we'll learn:
  • Multiplication, Division and Modulo/Remainder
  • Less-Than Comparison
  • Conditional Branching (lets us implement conditionals and loops!)
Multiplication

• **Assembly language notation:**
  
  - `mult $s, $t` Signed values
  - `multu $s, $t` Unsigned values

• **Machine language encodings:**
  
  - `mult 000000 sssss tttttt 00000 00000 011000`
  - `multu 000000 sssss tttttt 00000 00000 011001`

• Multiplies the numbers in $s$ and $t$.
  
  - Where does the result get stored?
  - Why do we need two versions here, but not for addition and subtraction?
Getting Multiplication Results

• When adding two 32-bit numbers, the result is at most 33 bits.
  • In our simplified MIPS, we ignore overflow.
  • The full version of MIPS provides two add instructions, one which raises an exception if overflow occurs, and one that ignores overflow.

• But when multiplying two 32-bit numbers, the result could need up to 64 bits to represent.

• Treating overflow as an error is undesirable in this case, but ignoring it and discarding the upper 32 bits may also be undesirable.

• Multiplication instructions in MIPS store the lower 32 bits of the result in the lo register, and the upper 32 bits in the hi register.
Move From Lo/Hi

• **Assembly language notation:**
  - `mflo $d`  Moves the value from lo into $d
  - `mfhi $d`  Moves the value from hi into $d

• **Machine language encodings:**
  - `mflo 000000 00000 00000 dddddd 00000 010010`
  - `mfhi 000000 00000 00000 dddddd 00000 010000`

• When writing assembly language programs in this course, it is safe to just use `mflo` to get multiplication results – we will not worry about multiplication overflow in assignments.
Division

• **Assembly language notation:**
  
  \[
  \text{div} \; \$s, \; \$t \quad \text{Signed values}
  \]
  
  \[
  \text{divu} \; \$s, \; \$t \quad \text{Unsigned values}
  \]

• **Machine language encodings:**

  \[
  \text{div} \; 000000 \; sssss \; ttttt \; 00000 \; 00000 \; 011010
  \]
  
  \[
  \text{divu} \; 000000 \; sssss \; ttttt \; 00000 \; 00000 \; 011011
  \]

• These instructions compute the *quotient* and *remainder* simultaneously, storing the quotient in **lo** and the remainder in **hi**.
Notes about (Signed) Division

• The remainder can be *negative* – similar to the modulo operator in C/C++ (as opposed to mathematical modulo).

• The quotient $q$ and remainder $r$ are solutions to this equation:
  - $s = (t \cdot q) + r$, where $|t \cdot q| \leq s$ and $|r| < s$

• The $t \cdot q$ part is always bounded by $s$ in absolute value, and the remainder makes up for any missing part.
  - If $s$ is positive, then: $(t \cdot q) \leq s$, so $r$ must be positive.
  - If $s$ is negative, then: $(t \cdot q) \geq s$, so $r$ must be negative.

• **So the sign of the remainder matches the sign of $s$.**
  - Easy way to remember: if $t$ is larger than $s$, the quotient is 0 and the equation becomes $s = r$, so the signs must match.
Comparison

• **Assembly language notation:**
  
  slt  \$d, \$s, \$t \quad \text{Sets } \$d \text{ to } 1 \text{ if } \$s < \$t, 0 \text{ otherwise}
  
  sltu \$d, \$s, \$t \quad \text{slt is for signed values, sltu for unsigned}

• **Machine language encodings:**
  
  slt  000000 sssss sssss sssss sssss sssss 00000 101010
  
  sltu 000000 sssss sssss sssss sssss sssss 00000 101011

• Consider the 32-bit word 0xFFFFFFFF = 111…11.

• In unsigned this is $2^{32} - 1$, but in two’s complement it’s -1.

• So comparing this value with 0 would give opposite results for slt/sltu.
Conditional Branching

• **Assembly language notation:**
  
  beq $s, $t, i  
  Branch with offset i if $s == $t
  
  bne $s, $t, i  
  Branch with offset i if $s != $t

• **Machine language encodings:**
  
  beq 000100 sssss ttttt iiiii iiiii iiiii iiiii
  
  bne 000101 sssss ttttt iiiii iiiii iiiii iiiii

• The offset value i is encoded in 16-bit two’s complement.

• What does "Branch with offset i" mean?
Conditional Branching, Explained

• Recall: The jr $s (Jump Register) instruction sets PC to the value in $s.
• The branch instructions increment PC by i words, where i is the 16-bit immediate operand.
• Example: If $3 is zero, set $3 = $1, otherwise, set $3 = $2.

   bne $3, $0, 2
   add $3, $1, $0
   beq $0, $0, 1
   add $3, $2, $0
   jr $31
Conditional Branching, Explained

• Recall: The jr $s (Jump Register) instruction sets PC to the value in $s.
• The branch instructions increment PC by i words, where i is the 16-bit immediate operand.
• Example: If $3 is zero, set $3 = $1, otherwise, set $3 = $2.

```assembly
bne $3, $0, 2  ← When this bne executes
add $3, $1, $0  ← PC is here
beq $0, $0, 1
add $3, $2, $0
jr $31
```
Conditional Branching, Explained

• Recall: The jr $s (Jump Register) instruction sets PC to the value in $s.
• The branch instructions increment PC by i words, where i is the 16-bit immediate operand.
• Example: If $3 is zero, set $3 = $1, otherwise, set $3 = $2.

```plaintext
bne $3, $0, 2          ← If $3 != 0, PC += 8 (2 words)
add $3, $1, $0
beq $0, $0, 1
add $3, $2, $0          ← PC is now here
jr $31
```
Conditional Branching, Explained

• Recall: The jr $s (Jump Register) instruction sets PC to the value in $s.
• The branch instructions increment PC by $i$ words, where $i$ is the 16-bit immediate operand.
• Example: If $3$ is zero, set $3 = 1$, otherwise, set $3 = 2$.

```
bne $3, $0, 2  \leftarrow \text{If } 3 = 0, \text{ do not branch} 
add $3, $1, $0  \leftarrow \text{PC stays here} 
beq $0, $0, 1 
add $3, $2, $0 
jr $31
```
Conditional Branching, Explained

• Recall: The jr $s (Jump Register) instruction sets PC to the value in $s.
• The branch instructions increment PC by i words, where i is the 16-bit immediate operand.
• Example: If $3 is zero, set $3 = $1, otherwise, set $3 = $2.

  bne $3, $0, 2
  add $3, $1, $0
  beq $0, $0, 1  ← When this beq executes
  add $3, $2, $0  ← PC is here
  jr $31
Conditional Branching, Explained

• Recall: The jr $s (Jump Register) instruction sets PC to the value in $s.

• The branch instructions *increment* PC by i words, where i is the 16-bit immediate operand.

• Example: If $3 is zero, set $3 = $1, otherwise, set $3 = $2.

  bne $3, $0, 2
  add $3, $1, $0
  beq $0, $0, 1  \leftarrow \text{Since } $0 == $0, \text{ PC += 4 (1 word)}
  add $3, $2, $0
  jr $31  \leftarrow \text{PC is now here}
Loops with Branching

• Branch offsets can be negative, which lets us implement loops.
• Example: A MIPS program that sums the numbers from 1 to n, where $2 starts out holding the value of n.

```
add $3, $0, $0
add $3, $3, $2
lis $1
.word -1
add $2, $2, $1
bne $2, $0, -5
jr $31
```

Pseudocode version:

```
$3 = 0
repeat
    $3 += $2
    $1 = -1
    $2 += $1
until $2 == 0
```
Loops with Branching

- Notice we load the value -1 into $1 on every iteration of the loop.
- This is wasteful because the value doesn’t change. It would be more efficient to move this code outside of the loop.

```
add $3, $0, $0
add $3, $3, $2
lis $1
.word -1
add $2, $2, $1
bne $2, $0, -5
jr $31
```

Pseudocode version:

```
$3 = 0
repeat
    $3 += $2
    $1 = -1
    $2 += $1
until $2 == 0
```
Loops with Branching

• We moved it out of the loop... or did we?
• We did not change the branch offset! It is still -5, so the loop still includes the code we moved.

```
add $3, $0, $0
lis $1
.word -1
add $3, $3, $2
add $2, $2, $1
bne $2, $0, -5
jr $31
```

Pseudocode version:

```
$3 = 0
$1 = -1
repeat
    $3 += $2
    $2 += $1
until $2 == 0
```
Loops with Branching

• Now we have successfully moved it out of the loop.
• Updating the branch offsets every time you change the length of a loop is a hassle. Fortunately, there is a better way.

```
add $3, $0, $0
lis $1
.word -1
add $3, $3, $2
add $2, $2, $1
bne $2, $0, -3
jr $31
```

Pseudocode version:

```
$3 = 0
$1 = -1
repeat
    $3 += $2
    $2 += $1
until $2 == 0
```
Branching with Labels

• When working in *assembly language*, instead of using numeric offsets, we can use **labels** to specify the location to branch to.

<table>
<thead>
<tr>
<th>(Without labels)</th>
<th>(With labels)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bne $3, $0, 2</td>
<td>bne $3, $0, nonZero</td>
</tr>
<tr>
<td>add $3, $1, $0</td>
<td>add $3, $1, $0</td>
</tr>
<tr>
<td>beq $0, $0, 1</td>
<td>beq $0, $0, skip</td>
</tr>
<tr>
<td>add $3, $2, $0</td>
<td>nonZero: add $3, $2, $0</td>
</tr>
<tr>
<td>jr $31</td>
<td>skip: jr $31</td>
</tr>
</tbody>
</table>
Branching with Labels

- When working in *assembly language*, instead of using numeric offsets, we can use **labels** to specify the location to branch to.

  (Without labels)                        (With labels)
  add $3, $0, $0             add $3, $0, $0
  lis $1                       lis $1
  .word -1                      .word -1
  add $3, $3, $2               loop: add $3, $3, $2
  add $2, $2, $1               add $2, $2, $1
  bne $2, $0, -3             bne $2, $0, loop
  jr $31                        jr $31
Example: Absolute Value

• $1$ contains a two's complement integer.
• Write a program that computes the absolute value of this integer and store its unsigned representation in $3$.

```
add $3, $0, $1 ; Copy $1 to $3
slt $2, $1, $0 ; $2 = 1 if $1 is negative, 0 otherwise
beq $2, $0, nonNegative
sub $3, $0, $3 ; Negate the value in $3
nonNegative:
jr $31
```
Example: Sum of Bits

- Compute the sum of bits of the value in $1$, and store the result in $3$.
- To get the **rightmost bit** of a binary value, take the value modulo 2.
- To **shift the value right** by one bit, divide it by 2.
- Our program will be based on the following pseudocode:

```plaintext
$3 = 0$
while($1 != 0):
    lo = $1 / 2$
    hi = $1 % 2$
    $1 = lo$
    $3 += hi$
```
Example: Sum of Bits

• Compute the sum of bits of the value in $1$, and store the result in $3$.

  $3 = 0$
  while($1 != 0)$:
    lo = $1 / 2$
    hi = $1 \% 2$
    $1 = lo$
    $3 += hi$
Example: Sum of Bits

• Compute the sum of bits of the value in $1$, and store the result in $3$.

```
add $3$, $0$, $0$
while($1$ != $0$):
  lo = $1$ / 2
  hi = $1$ % 2
  $1$ = lo
  $3$ += hi
```
Example: Sum of Bits

- Compute the sum of bits of the value in $1$, and store the result in $3$.

  ```plaintext
  add $3, $0, $0
  while($1 != 0):
    lo = $1 / 2
    hi = $1 % 2
    $1 = lo
    $3 += hi
  ```
Example: Sum of Bits

• Compute the sum of bits of the value in $1, and store the result in $3.

    add $3, $0, $0
    loop:
    beq $1, $0, end
    lo = $1 / 2
    hi = $1 % 2
    $1 = lo
    $3 += hi
    beq $0, $0, loop
    end:
Example: Sum of Bits

• Compute the sum of bits of the value in $1$, and store the result in $3$.

```
add $3, $0, $0
loop:
  beq $1, $0, end
  lo = $1 / 2
  hi = $1 % 2
  $1 = lo
  $3 += hi
  beq $0, $0, loop
end:
```
Example: Sum of Bits

- Compute the sum of bits of the value in $1, and store the result in $3.

  ```assembly
  add $3, $0, $0
  lis $2
  .word 2
  loop:
  beq $1, $0, end
  divu $1, $2
  $1 = lo
  $3 += hi
  beq $0, $0, loop
  end:
  ```
Example: Sum of Bits

• Compute the sum of bits of the value in $1, and store the result in $3.
  
  add $3, $0, $0
  lis $2
  .word 2
  loop:
  beq $1, $0, end
    divu $1, $2
    $1 = lo
    $3 += hi
  beq $0, $0, loop
  end:
Example: Sum of Bits

• Compute the sum of bits of the value in $1, and store the result in $3.

  add $3, $0, $0
  lis $2
  .word 2
  loop:
  beq $1, $0, end
    divu $1, $2
    mflo $1
  $3 += hi
  beq $0, $0, loop
  end:
Example: Sum of Bits

- Compute the sum of bits of the value in $1, and store the result in $3.

  ```assembly
  add $3, $0, $0
  lis $2
  .word 2
  loop:
  beq $1, $0, end
     divu $1, $2
     mflo $1
     $3 += hi
  beq $0, $0, loop
  end:
  ```
Example: Sum of Bits

• Compute the sum of bits of the value in $1, and store the result in $3.

  add $3, $0, $0
  lis $2
  .word 2
  loop:
  beq $1, $0, end
    divu $1, $2
    mflo $1
    mfhi $5
    add $3, $3, $5
  beq $0, $0, loop
  end:
Example: Sum of Bits

- Compute the sum of bits of the value in $1, and store the result in $3.

  ```
  add $3, $0, $0
  lis $2
  .word 2
  loop:
  beq $1, $0, end
  divu $1, $2
  mflo $1
  mfhi $5
  add $3, $3, $5
  beq $0, $0, loop
  end: jr $31
  ```
Example: Array Loops

- $1$ contains the address of an array $A$ and $2$ contains its size $n$.
Express the following loop in assembly, using $5$ to hold the index $i$.

```assembly
for(int i = 0; i < n; ++i) { A[i] = 0; }
```

```
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds $i
for: slt $6, $5, $2 ; $6 is 1 if $i < n, 0 otherwise
    beq $6, $0, end ; Go to end of loop if $i < n is false
    mult $5, $4
    mflo $6 ; $6 = $i * 4
    add $6, $1, $6 ; $6 = address of $A + ($i * 4) = address of $A[i]
    sw $0, 0($6) ; $A[i] = 0
    add $5, $5, $11 ; $i += 1
    beq $0, $0, for ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Instead of multiplying by $4$, can increment a separate counter by $4$:
    
    ```
    lis $11
    .word 1
    lis $4
    .word 4
    add $5, $0, $0 ; $5 holds $i$
    for: slt $6, $5, $2 ; $6 is 1 if $i < n$, 0 otherwise
      beq $6, $0, end ; Go to end of loop if $i < n$ is false
      mult $5, $4
      mflo $6 ; $6 = i * 4
      add $6, $1, $6 ; $6 = address of $A + (i * 4) = address of $A[i]$
      sw $0, 0($6) ; $A[i] = 0$
      add $5, $5, $11 ; i += 1
      beq $0, $0, for ; back to top of loop
    end:
    ```
Example: Array Loops

• $1 contains the address of an array A and $2 contains its size n. There are a lot of ways to write a loop that zeroes out the array.
  • Instead of multiplying by 4, can increment a separate counter by 4:

```
lis $11
   .word 1
lis $4
   .word 4
add $5, $0, $0   ; $5 holds i
for: slt $6, $5, $2   ; $6 is 1 if i < n, 0 otherwise
    beq $6, $0, end   ; Go to end of loop if i < n is false
    add $6, $1, $6    ; $6 = address of A + (i * 4) = address of A[i]
    sw $0, 0($6)     ; A[i] = 0
    add $5, $5, $11   ; i += 1
    beq $0, $0, for   ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Instead of multiplying by 4, can increment a separate counter by 4:

```assembly
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds $i$

for: slt $6, $5, $2 ; $6 is 1 if $i < n$, 0 otherwise
    beq $6, $0, end ; Go to end of loop if $i < n$ is false
    add $6, $1, $6 ; $6 = address of $A + (i * 4) = address of $A[i]
    sw $0, 0($6) ; $A[i] = 0
    add $5, $5, $11 ; $i += 1
    beq $0, $0, for ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Instead of multiplying by 4, can increment a separate counter by 4:

```assembly
lis $11
    .word 1
lis $4
    .word 4
add $5, $0, $0  ; $5 holds i
add $7, $0, $0  ; $7 holds i * 4
for:    slt $6, $5, $2  ; $6 is 1 if i < n, 0 otherwise
            beq $6, $0, end  ; Go to end of loop if i < n is false
            add $6, $1, $7  ; $6 = address of A + (i * 4) = address of A[i]
            sw $0, 0($6)       ; A[i] = 0
            add $5, $5, $11   ; i += 1
            add $7, $7, $4    ; $7 += 4
            beq $0, $0, for   ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Instead of comparison with slt, can decrement $2$ until it reaches 0:

```assembly
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds i
add $7, $0, $0 ; $7 holds i * 4
for: slt $6, $5, $2 ; $6 is 1 if i < n, 0 otherwise
  beq $6, $0, end ; Go to end of loop if i < n is false
  add $6, $1, $7 ; $6 = address of A + (i * 4) = address of A[i]
  sw $0, 0($6) ; A[i] = 0
  add $5, $5, $11 ; i += 1
  add $7, $7, $4 ; $7 += 4
  beq $0, $0, for ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  
  • Instead of comparison with slt, can decrement $2$ until it reaches 0:
    
    ```
    lis $11
    .word 1
    lis $4
    .word 4
    add $5, $0, $0 ; $5 holds $i$
    add $7, $0, $0 ; $7 holds $i * 4$
    for: beq $6, $0, end ; Go to end of loop if $i < n$ is false
      add $6, $1, $7 ; $6 = address of $A + (i * 4) = address of $A[i]$
      sw $0, 0($6) ; $A[i] = 0$
      add $5, $5, $11 ; $i += 1$
      add $7, $7, $4 ; $7 += 4$
    beq $0, $0, for ; back to top of loop
    end:
    ```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$.

There are a lot of ways to write a loop that zeroes out the array.

• Instead of comparison with $\text{slt}$, can decrement $2$ until it reaches $0$:

```assembly
lis $11
.word 1
lis $4
.word 4
add $5, $0, $0 ; $5 holds $i$
add $7, $0, $0 ; $7 holds $i \times 4$
for: beq $2, $0, end ; Go to end of loop if $n == 0$
   add $6, $1, $7 ; $6 = address of $A + (i \times 4) = \text{address of } A[i]$
   sw $0, 0($6) ; $A[i] = 0$
   add $5, $5, $11 ; i += 1$
   add $7, $7, $4 ; $7 += 4$
   sub $2, $2, $11 ; n -= 1$
   beq $0, $0, for ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Instead of comparison with slt, can decrement $2$ until it reaches $0$:

```assembly
lis $11
.word 1
lis $4
.word 4
; $5 is no longer used!
add $7, $0, $0 ; $7 holds i * 4
for: beq $2, $0, end ; Go to end of loop if n == 0
  add $6, $1, $7 ; $6 = address of A + (i * 4) = address of A[i]
  sw $0, 0($6) ; A[i] = 0
  add $7, $7, $4 ; $7 += 4
  sub $2, $2, $11 ; n -= 1
  beq $0, $0, for ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Can modify the address in $1$ directly instead of using a temporary register:

```assembly
  lis $11
  .word 1
  lis $4
  .word 4

  add $7, $0, $0 ; $7 holds $i \times 4
  for:  beq $2, $0, end ; Go to end of loop if $n == 0
       add $6, $1, $7 ; $6 = address of $A + ($i \times 4) = address of $A[i]
       sw $0, 0($6) ; $A[i] = 0

       add $7, $7, $4 ; $7 += 4
       sub $2, $2, $11 ; $n -= 1
       beq $0, $0, for ; back to top of loop

  end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Can modify the address in $1$ directly instead of using a temporary register:

```assembly
lis $11
.word 1
lis $4
.word 4
add $7, $0, $0 ; $7 holds $i \times 4
for: beq $2, $0, end ; Go to end of loop if $n == 0
    sw $0, 0($1) ; On iteration $i$, set $A[i] = 0$
add $1, $1, $4 ; $1 = address of $A[i+1]$
sub $2, $2, $11 ; $n = 1$
beq $0, $0, for ; back to top of loop
end:
```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Can modify the address in $1$ directly instead of using a temporary register:
    
    ```
    lis $11
    .word 1
    lis $4
    .word 4
    ; $7 is no longer used!
    for:  beq $2, $0, end  ; Go to end of loop if $n == 0
          sw $0, 0($1)    ; On iteration $i$, set $A[i] = 0$
          add $1, $1, $4   ; $1 = address of $A[i+1]$
          sub $2, $2, $11  ; $n -= 1$
          beq $2, $0, for  ; back to top of loop
    end:
    ```
Example: Array Loops

• $1$ contains the address of an array $A$ and $2$ contains its size $n$. There are a lot of ways to write a loop that zeroes out the array.
  • Can modify the address in $1$ directly instead of using a temporary register:

```assembly
lis $11
.word 1
lis $4
.word 4
for:   beq $2, $0, end ; Go to end of loop if $n == 0
       sw $0, 0($1)    ; On iteration $i$, set $A[i] = 0$
       add $1, $1, $4  ; $1 = address of $A[i+1]$
       sub $2, $2, $11 ; $n -= 1$
       beq $0, $0, for ; back to top of loop
end:
```