MIPS ASSEMBLY

LECTURE 08-1

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

- ON-LINE COURSE LOGISTICS
- REVIEW OF MIPS ASSEMBLY, BUT DONE VIA…
- … MY SOLUTIONS TO HOMEWORK 07
- MIPS MEMORY OPERATIONS (WITH EXERCISES)

SOLUTION TO HOMEWORK 07 EXERCISE 1

- Below is the "kernel" of my MIPS code to multiply using repeated addition:

```
1. multiply:
2.     li      $t0, 0             # product = 0
3. multiply_loop:
4.     beqz    $t2, report        # if y == 0 goto report
5.     addu    $t0, $t0, $t1      # product += x
6.     addiu   $t2, $t2, -1       # y -= 1
7.     b       multiply_loop
8. report:
```

- It uses registers t0 to compute the product, repeatedly adding t1 to it.
- It starts with a product of 0, performs the addition at line 5 t2 times.

C++ SOLUTION TO HOMEWORK 07 EXERCISE 1

- Here is C++ code that mimics that MIPS code. The "kernel" loop is in **green**.

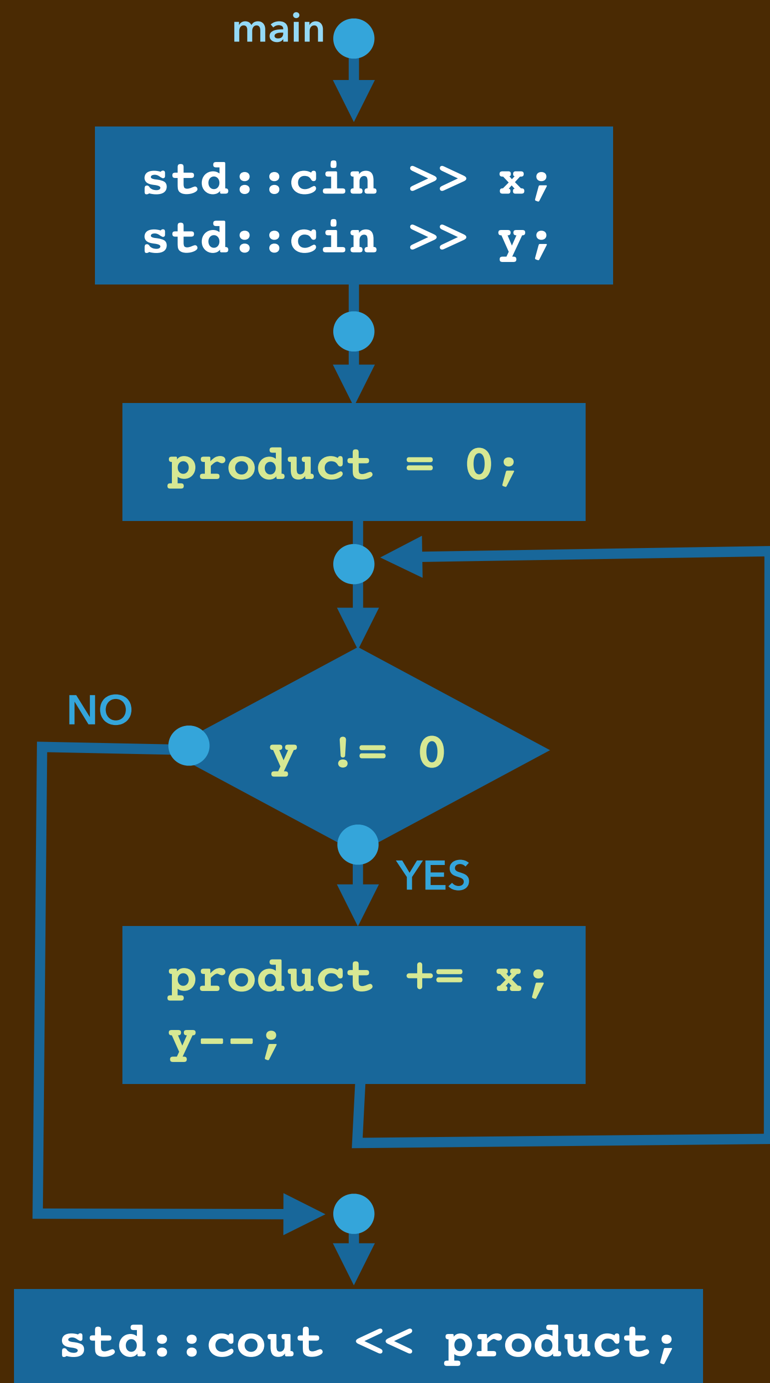
```
int main() { int x,y, product;
  std::cin >> x;
  std::cin >> y;
  product = 0;
  while (y != 0) {
    product += x;
    y--;
  }
  std::cout << product;
  return 0;
}
```

- Let's convert it to MIPS code, maybe how a compiler might...

FLOWCHART OF SOLUTION

```
int main() { int x,y, product;
  std::cin >> x;
  std::cin >> y;
  product = 0;
  while (y != 0) {
    product += x;
    y--;
  }
  std::cout << product;
  return 0;
}
```

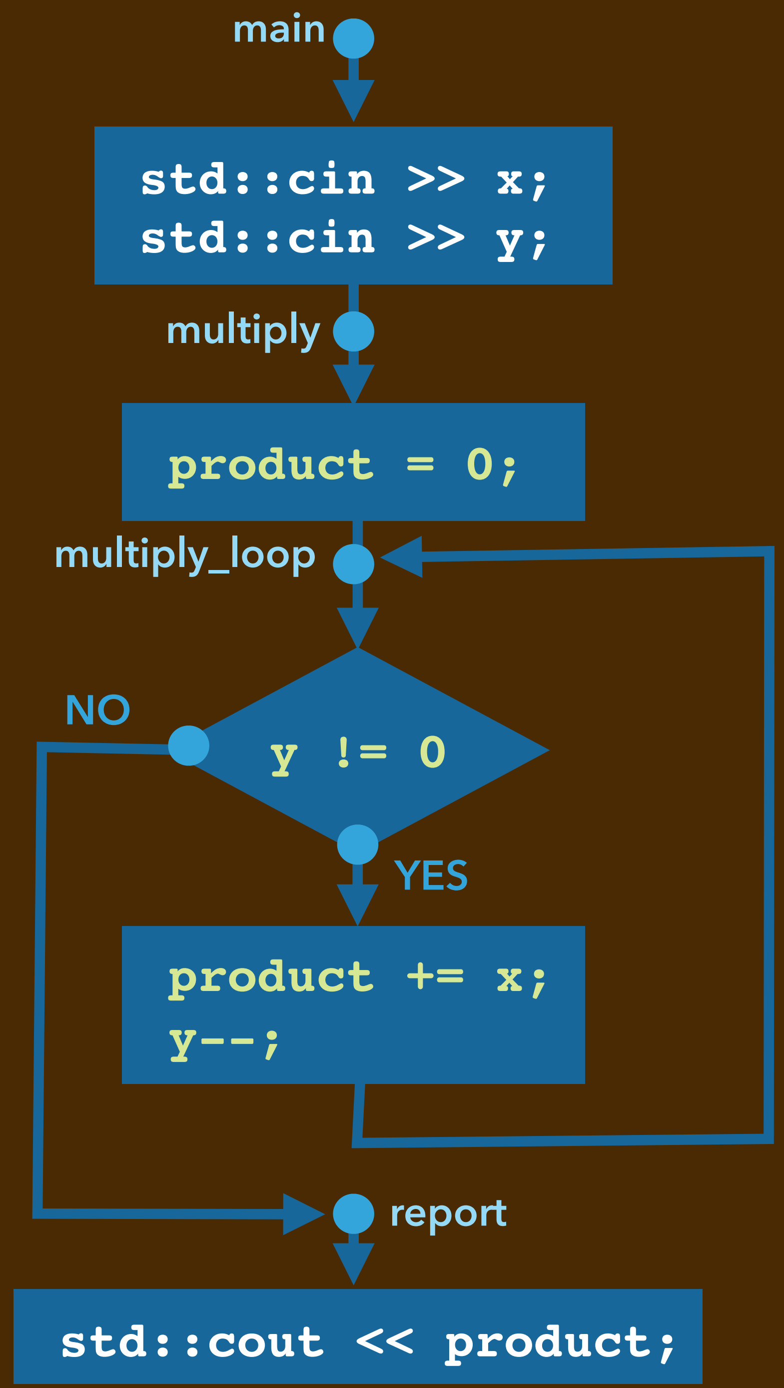
- Here is the "flow logic" of that code:
- Let's convert it to MIPS...



CONVERSION TO MIPS

```
main:
  std::cin >> x;
  std::cin >> y;
multiply:
  product = 0;
multiply_loop:
  while (y != 0) {
    product += x;
    y--;
  }
report:
  std::cout << product;
end_main:
  return 0;
```

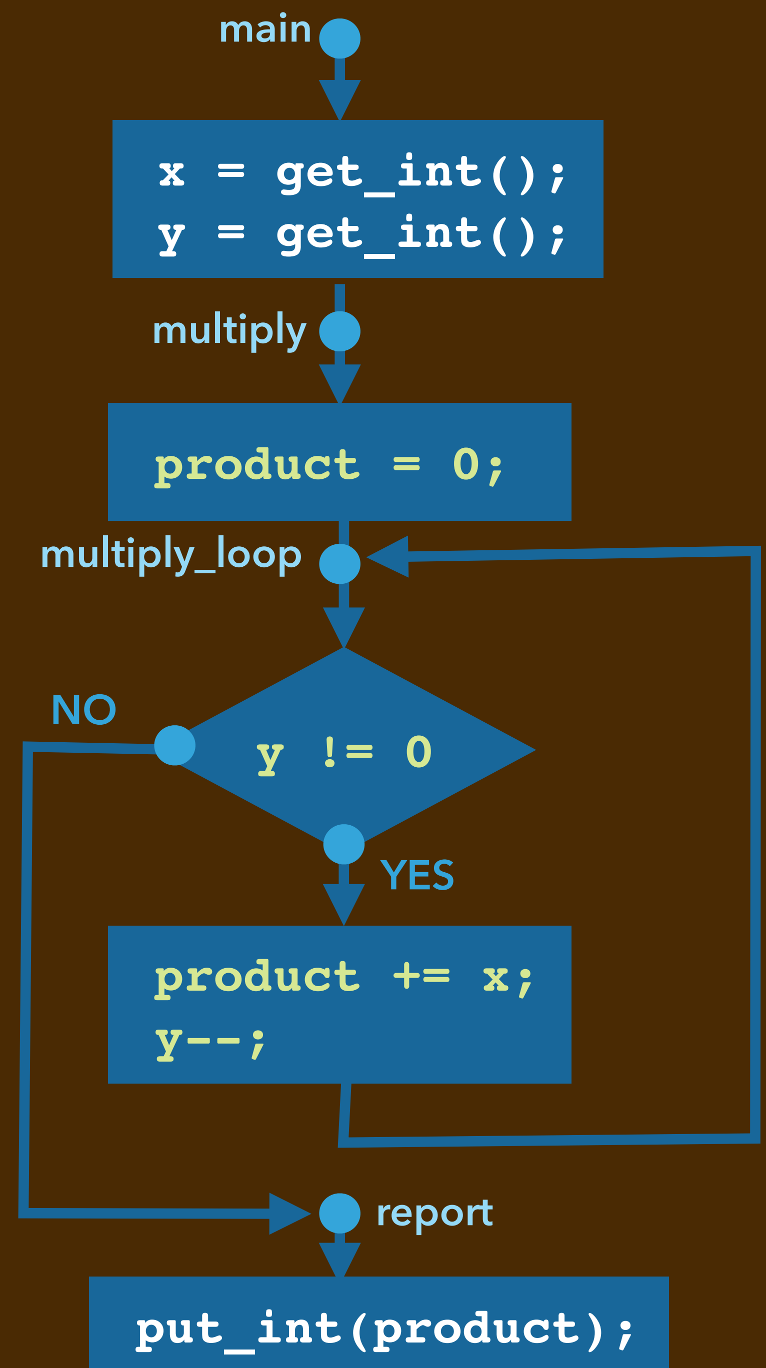
- Let's label the flow's targets.



CONVERSION TO MIPS

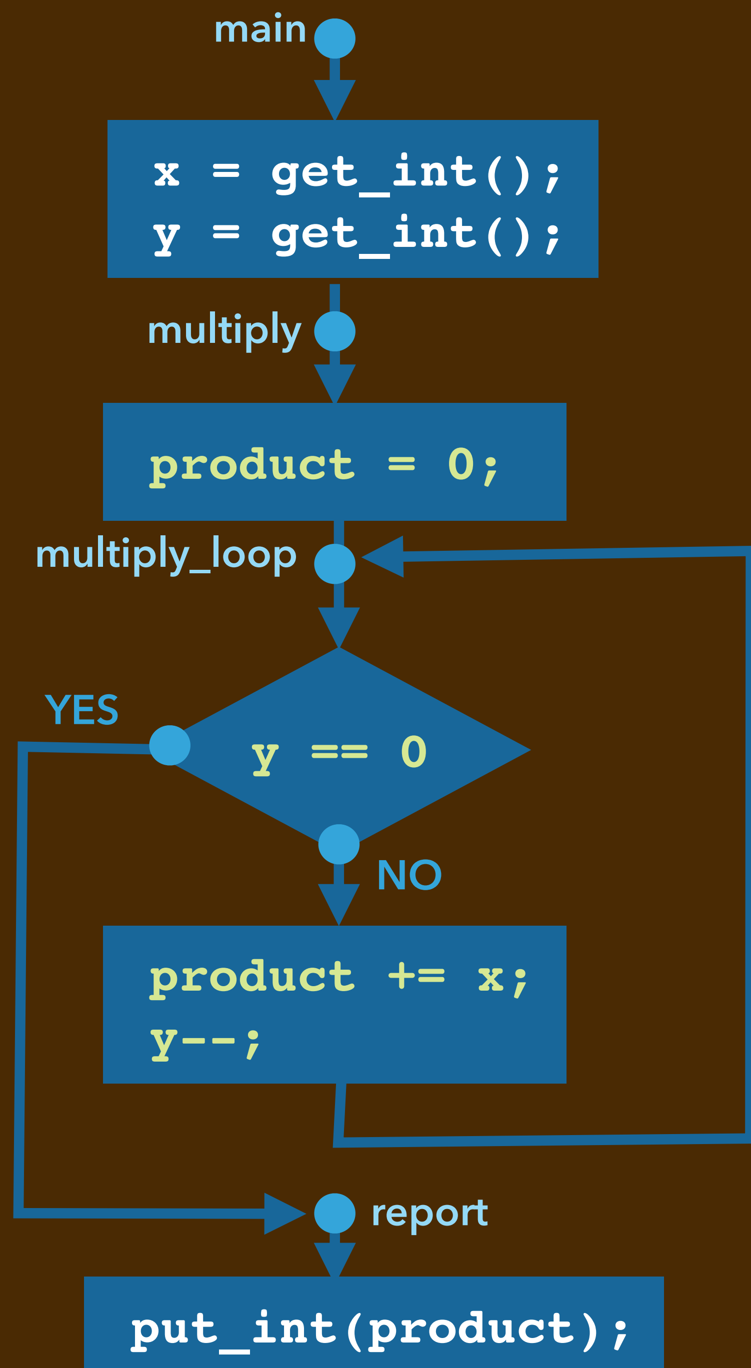
```
main:
  x = get_int();
  y = get_int();
multiply:
  product = 0;
multiply_loop:
  while (y != 0) {
    product += x;
    y--;
  }
report:
  put_int(product);
end_main:
  return 0;
```

- Substitute system calls.



CONVERSION TO MIPS

```
main:
   x = get_int();
   y = get_int();
multiply:
   product = 0;
multiply_loop:
   if (y == 0) goto report;
   product += x;
   y--;
   goto multiply_loop;
report:
   put_int(product);
end_main:
   return 0;
```

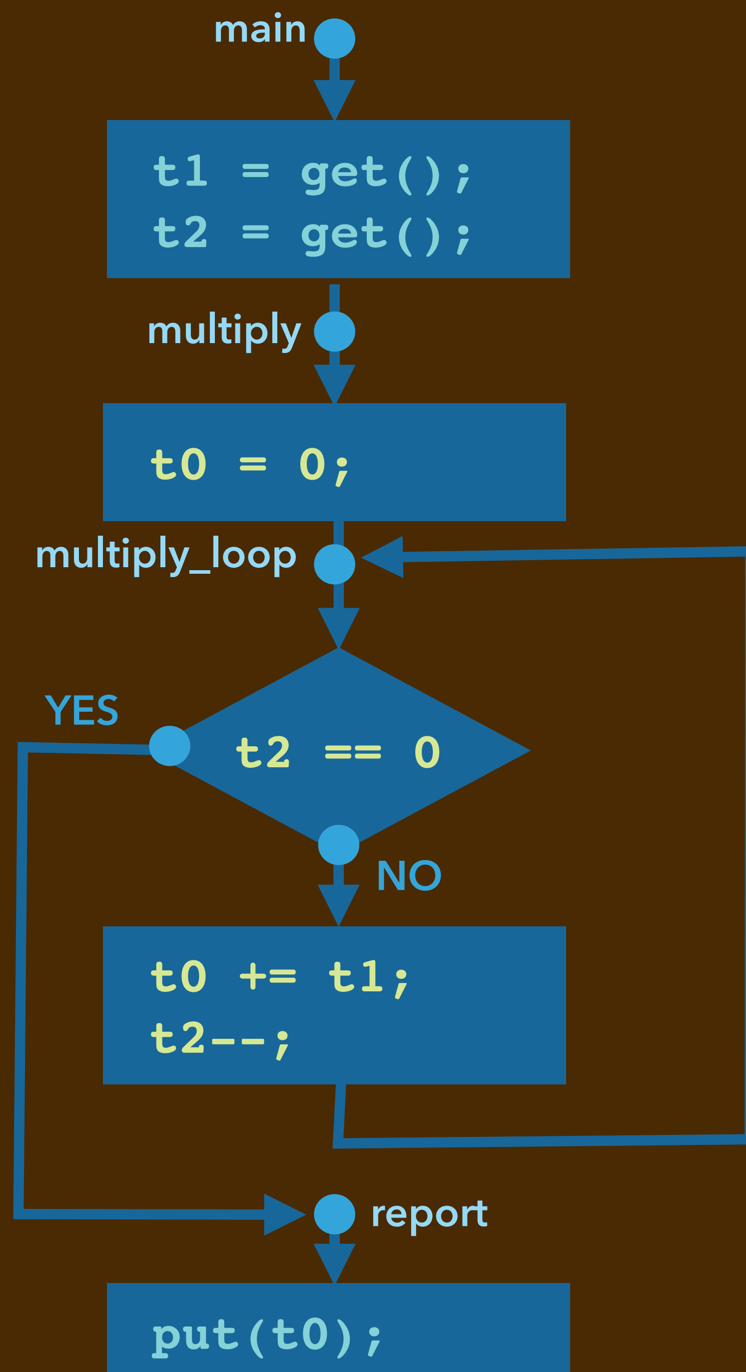


- Replace "structured" conditional statements with GOTOs.

CONVERSION TO MIPS

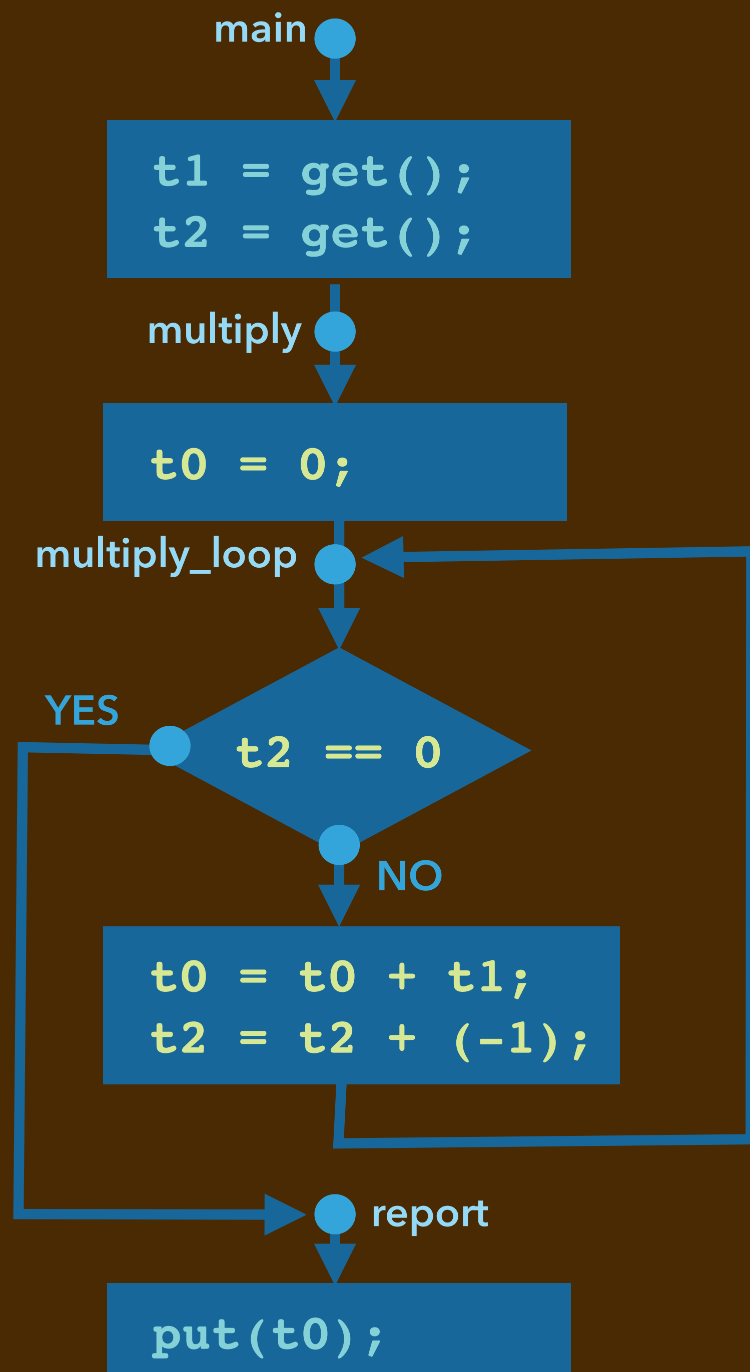
```
main:
    t1 = get_int();
    t2 = get_int();
multiply:
    t0 = 0;
multiply_loop:
    if (t2 == 0) goto report;
    t0 += x;
    t1--;
    goto multiply_loop;
report:
    put_int(t0);
end_main:
    return 0;
```

- Choose registers.



CONVERSION TO MIPS

```
main:
   t1 = get_int();
   t2 = get_int();
multiply:
   t0 = 0;
multiply_loop:
   if (t2 == 0) goto report;
   t0 = t0 + t1;
   t1 = t1 + (-1);
   goto multiply_loop;
report:
   put_int(t0);
end_main:
   return 0;
```

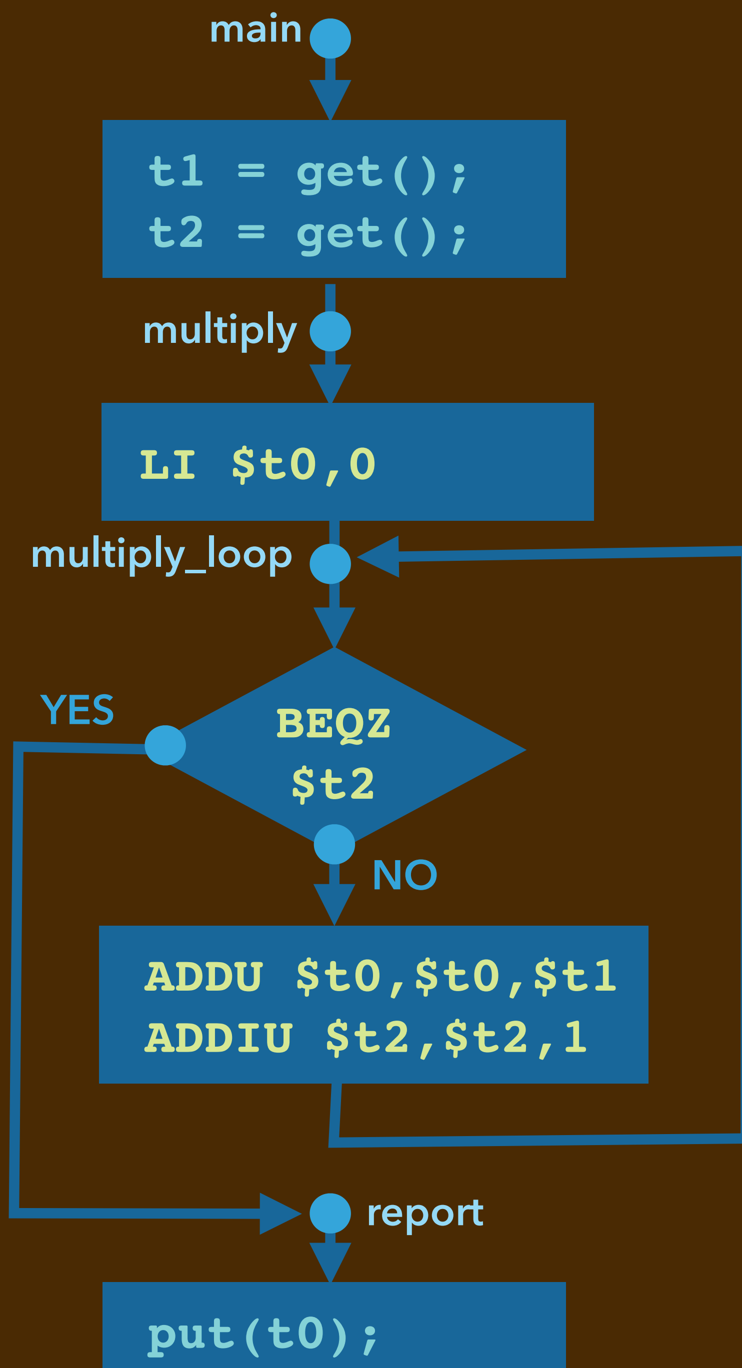


- Convert assignments to machine-like statements.

CONVERSION TO MIPS

```
main:
   t1 = get_int();
   t2 = get_int();
multiply:
   LI $t0,0
multiply_loop:
   BEQZ $t2,report
   ADDU $t0,$t0,$t1
   ADDIU $t2,$t2,-1
   B multiply_loop
report:
   put_int(t0);
end_main:
   return 0;
```

- Change C statements to MIPS instructions.



LET'S REVIEW SOME OF THESE INSTRUCTIONS…

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,1
  B multiply_loop
report:
```

LET'S REVIEW SOME OF THESE INSTRUCTIONS…

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

LOAD IMMEDIATE

- Format:

  LI *destination register* , *value*

- Meaning: load a register with a specific value.
- NOTES:
  - destination can be any MIPS register
  - value must be a 32-bit constant, including negative values using 2's complement encoding
  - "immediate" because bits of value are in the instruction's code

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
1. multiply:
2.   LI $t0,0
3. multiply_loop:
4.   BEQZ $t2,report
5.   ADDU $t0,$t0,$t1
6.   ADDIU $t2,$t2,-1
7.   B multiply_loop
8. report:
```

ADD (UNTRAPPED)

- Format:

  ADDU *destination*, *source1*, *source2*

- Meaning: add the contents of two registers, store sum the sum in a register
- NOTES:
  - destination can be any MIPS register
  - sources can be any registers
  - "untrapped" means errors (e.g. overflow) can be ignored by the machine
  - there's SUBU and bitwise AND, OR, XOR also

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
1. multiply:
2.   LI $t0,0
3. multiply_loop:
4.   BEQZ $t2,report
5.   ADDU $t0,$t0,$t1
6.   ADDIU $t2,$t2,-1
7.   B multiply_loop
8. report:
```

ADD IMMEDIATE (UNTRAPPED)

- Format:

  ADDIU *destination* **,** *source* **,** *value*

- Meaning: compute the some of a register's contents to a value, store the sum in a register
- NOTES:
  - source and destination can be any MIPS registers
  - value is a 16-bit constant, including negative values. Assumes 2's complement encoding.
  - (there's no SUBIU instruction)

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

BRANCH IF EQUAL TO ZERO

- Format:

  BEQZ *register*, *target-label*

- Meaning: go to the labeled instruction if a register's value is zero. Continue below if not.

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

BRANCH IF EQUAL TO ZERO

- Format:

  BEQZ *register* , *target-label*

- Meaning: go to the labeled instruction **if a register's value is zero**. Continue below if not.

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

BRANCH IF EQUAL TO ZERO

- Format:

  BEQZ *register* **,** *target-label*

- Meaning: go to the labeled instruction **if a register's value is zero**. Continue below **if not**.

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

```
product = 0;
while (y != 0) {
  product += x;
  y--;
}
```

BRANCH IF EQUAL TO ZERO

- Format:

  BEQZ *register* **,** *target-label*

- Meaning: go to the labeled instruction if a register's value is zero. Continue below if not.
- NOTE:
  - if "guarding" a while loop, the **condition** specifies when the loop should ***stop***, the opposite of the condition for continuing

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

BRANCH IF EQUAL TO ZERO

- Format:

  BEQZ *register* , *target-label*

- Meaning: go to the labeled instruction if a register's value is zero. Continue below if not.
- NOTES:
  - branch target can be above or below
  - there is BEQZ, BNEZ, BLTZ, BGTZ, BLEZ, BGEZ
  - these are =, ≠, <, >, ≤, ≥ tests with 0

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
1. multiply:
2.   LI $t0,0
3. multiply_loop:
4.   BEQZ $t2,report
5.   ADDU $t0,$t0,$t1
6.   ADDIU $t2,$t2,-1
7.   B multiply_loop
8. report:
```

BRANCH (UNCONDITIONALLY)

- Format:

  B *target-label*

- Meaning: go to the labeled instruction

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
multiply:
  LI $t0,0
multiply_loop:
  BEQZ $t2,report
  ADDU $t0,$t0,$t1
  ADDIU $t2,$t2,-1
  B multiply_loop
report:
```

BRANCH (UNCONDITIONALLY)

- Format:

  B *target-label*

- Meaning: go to the labeled instruction

WHAT ABOUT INPUT AND OUTPUT?

- There are several different ***system calls*** you can make that SPIM understands.
  - get an integer input from the console
  - print an integer output to the console
  - print a null-terminated character sequence out to the console

    ➡ Each has a number that identifies it.

    ➡ You tell the system which should run by setting register v0 to that number.

SYSTEM CALLS

- Getting an integer input is ***system call #5***.

```
t1 = get_int();
```

```
LI $v0,5
syscall
MOVE $t1,$v0
```

- Meaning of the MIPS code:
  - We load register v0 with the system call number.
  - We make the system call.
    - ➡A special sequence of instructions gets executed by the system to read input from the console. The integer read gets placed in register v0.
  - We copy (i.e. "move") its value to register t1, our storage for the first multiplicand.

SYSTEM CALLS

- Outputting an integer is ***system call #1***.

```
put_int(t0);
```

```
LI $v0,1
MOVE $a0,$t0
syscall
```

- Meaning of the MIPS code:
  - We load register v0 with the system call number.
  - We load register a0 with the "argument", the value we want the system to output.
    - ➡This has us copy (again, "move") the register for the product into a0.
  - We make the system call.
    - ➡A sequence of instructions gets executed to output the value in a0 to the console.

THE (POORLY NAMED) MOVE INSTRUCTION

MOVE

- Format:

  MOVE *destination-register* **,** *source-register*

- Meaning:
  copy the contents of the source register into the destination register
- NOTE: the value in the source register ***doesn't change***
  - For example the MIPS code

    ```
    MOVE $t0,$t1
    ```

    would be like the C++ code

    ```
    product = x;
    ```

RETURNING FROM MAIN

- Our (main) program must return an integer value to the system.

```
return 0;
```

➡

```
LI $v0,0
JR $ra
```

- We'll see (on Friday) how to write functions and procedures and use them.
- These two instructions hint at how that works. In short:
  - Functions, by convention, return a value by setting register v0 to that value.
  - A register named ra stores information about who called `main`.
    - ➡ This is called the ***return address*** of the "caller" code.
    - ➡ The `JR` instruction "jumps back" to that instruction.
- ***For now, just make sure your main code always ends with these two lines.***

THE FULL SOLUTION TO HOMEWORK 07 EXERCISE 1

```
main:
  li      $v0,5
  syscall
  move    $t1,$v0
  li      $v0,5
  syscall
  move    $t2,$v0
multiply:
  li      $t0,0
multiply_loop:
  beqz    $t2,report
  addu    $t0,$t0,$t1
  addiu   $t2,$t2,-1
  b       multiply_loop
report:
  li      $v0, 1
  move    $a0, $t0
  syscall
  li      $v0, 0
  jr      $ra
```

THE HOMEWORK 07 EXERCISE 2 SOLUTION LOOKS SIMILAR

```
main:
  li      $v0,5
  syscall
  move    $t1,$v0
  li      $v0,5
  syscall
  move    $t2,$v0
divide:
  li      $t0,0
divide_loop:
  blt     $t1,$t2,report
  addiu   $t0,$t0,1
  subu    $t1,$t1,$t2
  b       divide_loop
report:
...  # now $t0 contains the quotient, $t1 the remainder. We output them with two system calls.
  li      $v0, 0
  jr      $ra
```

C++ SOLUTION TO HOMEWORK 07 EXERCISE 2

- Here is the C++ for it. Again, the kernel of the code is **green**.

```
int main() { int number, divisor, quotient;
  std::cin >> number;
  std::cin >> divisor;
  quotient = 0;
  while (number >= divisor) {
    quotient++;
    number -= divisor;
  }
  std::cout << quotient << std::endl;
  std::cout << number << std::endl;
  return 0;
}
```

- It repeatedly subtracts the divisor until only a remainder is left.
- It counts the number of subtractions made. That count is the quotient.

SOME MORE INSTRUCTIONS WORTH HIGHLIGHTING

```
main:
   li       $v0,5
   syscall
   move     $t1,$v0
   li       $v0,5
   syscall
   move     $t2,$v0
divide:
   li       $t0,0
divide_loop:
   blt      $t1,$t2,report
   addiu    $t0,$t0,1
   subu     $t1,$t1,$t2
   b        divide_loop
report:
   # output $t0 and $t1
...
   li       $v0, 0
   jr       $ra
```

BRANCH IF LESS THAN

- Format:

  BLT *source1, source2 , target-label*

- Meaning: go to the labeled instruction if the first source register is less than the second. Continue below if not.
- NOTES:
  - there is BLT, BGT, BLE, BGE, BEQ, BNE
  - these are <, >, ≤, ≥, =, ≠

SOME MORE INSTRUCTIONS WORTH HIGHLIGHTING

```
main:
    li       $v0,5
    syscall
    move     $t1,$v0
    li       $v0,5
    syscall
    move     $t2,$v0
divide:
    li       $t0,0
divide_loop:
    blt      $t1,$t2,report
    addiu    $t0,$t0,1
    subu     $t1,$t1,$t2
    b        divide_loop
report:
...  # output $t0 and $t1
    li       $v0, 0
    jr       $ra
```

SUBTRACT (UNTRAPPED)

- Format:

  SUBU *destination* , *source1* , *source2*

- Meaning: subtract source2 from source1, store the difference in a register
- NOTES:
  - destination is a register
  - sources are registers
  - "untrapped" means errors (e.g. borrow) can be ignored by the machine
  - does not modify the source registers

THERE'S MORE THAN JUST CODE THERE

```
        # quotient.asm
        #
        # This code asks for two integers and then
        # outputs the quotient and remainder due to
        # their division.

        .data
prompt_num:   .asciiz "Enter an integer: "
prompt_div:   .asciiz "Enter an integer divisor: "
fdbk1:        .asciiz "Dividing "
fdbk2:        .asciiz " by "
fdbk3:        .asciiz " yields a quotient of "
fdbk4:        .asciiz " with a remainder of "
fdbk5:        .asciiz ".\n"

      .globl main
      .text
main:
      # Key:
      #      $t0 - the number to be divided
      #      $t1 - the divisor
      #      $t2 - the remainder
      #      $t3 - the quotient, a count
       li      $v0,5
       ...
```

SEGMENTS IN A PROGRAM MEMORY IMAGE

- In MIPS assembly we specify the contents of a program's ***text*** and ***data*** ***segments***:
  - `.text` contains the bytes of the executable code of the program
  - `.data` contains additional sequences of bytes, and for various types of values
    - SPIM loads all the bytes of a program's image, both the text and the data.
- For example, "output100.asm" holds two strings and an integer in its data:

```
            .data
report:     .asciiz "The value held in memory is "
dot_eoln:   .asciiz ".\n"
value:      .word 100
            .globl main
            .text
main:
  la        $a0, report
  ...
```

SEGMENTS IN A PROGRAM MEMORY IMAGE

- Here is the full "output100.asm" program:

```
            .data
report:     .asciiz "The value held in memory is "
dot_eoln:   .asciiz ".\n"
value:      .word 100
            .globl main
            .text
main:
    la      $a0,report
    li      $v0,4
    syscall
    la      $t0,value
    lw      $a0,($t0)
    li      $v0,1
    syscall
    la      $a0,eoln
    li      $v0,4
    syscall
    li      $v0,0
    jr      $ra
```

LOADING A VALUE STORED AT AN ADDRESS

- Here is the full "output100.asm" program:

```
              .data
report:       .asciiz "The value held in memory is "
dot_eoln:     .asciiz ".\n"
value:        .word 100
              .globl main
              .text
main:
      la      $a0,report
      li      $v0,4
      syscall
      la      $t0,value
      lw      $a0,($t0)
      li      $v0,1
      syscall
      la      $a0,eoln
      li      $v0,4
      syscall
      li      $v0,0
      jr      $ra
```

LOAD ADDRESS INTO A REGISTER

- Format:

  LA *destination* **,** *program-label*

- Meaning: loads a register with the address of a labelled item in the code
- NOTE: For a string (.asciiz), this is the address of the first letter of its sequence.

LOADING A VALUE STORED AT AN ADDRESS

- Here is the full "output100.asm" program:

```
            .data
report:     .asciiz "The value held in memory is "
dot_eoln:   .asciiz ".\n"
value:      .word 100
            .globl main
            .text
main:
  la        $a0,report
  li        $v0,4
  syscall
  la        $t0,value
  lw        $a0,($t0)
  li        $v0,1
  syscall
  la        $a0,eoln
  li        $v0,4
  syscall
  li        $v0,0
  jr        $ra
```

LOAD ADDRESS INTO A REGISTER

- Format:

  LA *destination* , *program-label*

- Meaning: loads a register with the address of a labelled item in the code
- NOTE: For an integer (.word), this is the address of the first of its four bytes.

LOADING A VALUE STORED AT AN ADDRESS

- Here is the full "output100.asm" program:

```
            .data
report:     .asciiz "The value held in memory is "
dot_eoln:   .asciiz ".\n"
value:      .word 100
            .globl main
            .text
main:
  la        $a0,report
  li        $v0,4
  syscall
  la        $t0,value
  lw        $a0,($t0)
  li        $v0,1
  syscall
  la        $a0,eoln
  li        $v0,4
  syscall
  li        $v0,0
  jr        $ra
```

LOAD A VALUE STORED AT AN ADDRESS

- Format:

  `LW` *destination*`,(`*source*`)`

- Meaning: loads a register with the (4-byte word) value stored at an address
- NOTE: ***source*** is a register that holds the address., i.e. a pointer to the loaded data.

LOADING A VALUE FROM MEMORY

LOAD A (FOUR BYTE) VALUE FROM AN ADDRESS IN MEMORY

LW *destination*, (*source*)

- NOTES:
  - This is sometimes called a "memory-to-register" transfer or "fetch."
  - *source* is a register holding the address of the data we're fecthing.
  - We can think of the source register as a pointer. So **LW** is like the C code:

```
int* source_pointer;
...
int destination_value = *source_pointer;
```

REGISTERS VERSUS MEMORY

- A MIPS processor has only 32 registers for storing calculated values.
  - It can also access a large memory using addresses.
- MIPS is a ***"load/store" computer architecture***.
  - It can only perform calculations on data stored in its registers
  - To modify a value stored in memory it must:
    1. ***load*** a value from memory by its address into a register
    2. modify that value, computing a new value, within registers
    3. ***store*** that changed value into memory at that same address

EXAMPLE: INCREMENTING A VALUE IN MEMORY

- Here is MIPS code that adds 10 to a location in memory:

```
1.  LA    $a0,value
2.  LW    $t0,($a0)
3.  ADDIU $t0,$t0,10
4.  SW    $t0,($a0)
```

STORING A VALUE TO MEMORY

STORE A (FOUR BYTE) VALUE TO AN ADDRESS IN MEMORY

`sw` *source*, (*destination*)

- NOTES:
  - This is sometimes called a "register-to-memory" transfer.
  - *destination* is a register holding the address where we're storing the data in the *source* register

WORDS VERSUS BYTES

- Recall that the fundamental quantity manipulated by a MIPS32 instruction set is 32-bits long.
  - ➡ Registers hold 32 bits, i.e 4 bytes, of data.
  - ➡ Integers are 32 bits, i.e. 4 bytes, long.
  - ➡ Addresses are 32 bits, i.e. 4 bytes, long.
- So in MIPS32, a ***word*** is 32 bits, i.e. 4 bytes.
  - ➡ The **LW** instruction reads 4 bytes of data from memory.
  - ➡ The **SW** instruction write 4 bytes of data out to memory.
- There are also the **LB** and **SB** instructions for accessing a *single byte* in memory.
  - ➡ NOTE: strings are sequences of characters, each character is a byte of data.

LOADING A BYTE AND STORING A BYTE

LOAD A BYTE'S VALUE FROM AN ADDRESS IN MEMORY

LB *destination*, (*source*)

- NOTES:
  - This is sometimes called a "memory-to-register" transfer or a "fetch."
  - *source* is a register holding the address where we're fetching the data in *destination*

STORE A BYTE'S VALUE TO AN ADDRESS IN MEMORY

SB *source*, (*destination*)

- NOTES:
  - This is sometimes called a "register-to-memory" transfer.
  - *destination* is a register holding the address where we're storing the data of *source*

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
    .data
hello_ptr: .asciiz "hello\n"
    .globl main
    .text
main:
print_hello:
    li $v0, 4
    la $a0, hello_ptr
    syscall
    j   change_string
print_bye:
    li $v0, 4
    la $a0, hello_ptr
    syscall
end_main:
    li $v0, 0
    jr $ra
change_string:
    la $t0, hello_ptr
    li $t1, 'b'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, 'y'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, 'e'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, '\n'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, 0
    sb $t1, ($t0)
    j print_bye
```

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
    .data
hello_ptr: .asciiz "hello\n"
    .globl main
    .text
main:
print_hello:
    li  $v0, 4
    la  $a0, hello_ptr
    syscall
    j   change_string
print_bye:
    li  $v0, 4
    la  $a0, hello_ptr
    syscall
end_main:
    li  $v0, 0
    jr  $ra
change_string:
    la $t0, hello_ptr
    li $t1, 'b'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, 'y'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, 'e'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, '\n'
    sb $t1, ($t0)
    addiu $t0, $t0, 1
    li $t1, 0
    sb $t1, ($t0)
     j print_bye
```

- The code jumps around to fit neatly on this slide...

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
1.    .data
2. hello_ptr: .asciiz "hello\n"
3.    .globl main
4.    .text
5. main:
6. print_hello:
7.   li $v0, 4
8.   la $a0, hello_ptr
9.   syscall
10.  j   change_string
11. print_bye:
12.  li $v0, 4
13.  la $a0, hello_ptr
14.  syscall
15. end_main:
16.  li $v0, 0
17.  jr $ra
18. change_string:
19.  la $t0, hello_ptr
20.  li $t1, 'b'
21.  sb $t1, ($t0)
22.  addiu $t0, $t0, 1
23.  li $t1, 'y'
24.  sb $t1, ($t0)
25.  addiu $t0, $t0, 1
26.  li $t1, 'e'
27.  sb $t1, ($t0)
28.  addiu $t0, $t0, 1
29.  li $t1, '\n'
30.  sb $t1, ($t0)
31.  addiu $t0, $t0, 1
32.  li $t1, 0
33.  sb $t1, ($t0)
34.  j print_bye
```

- It first prints the string "hello\n" **in lines 6-9**.

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
1.    .data
2. hello_ptr: .asciiz "hello\n"
3.   .globl main
4.   .text
5. main:
6. print_hello:
7.   li $v0, 4
8.   la $a0, hello_ptr
9.   syscall
10.  j   change_string
11. print_bye:
12.  li $v0, 4
13.  la $a0, hello_ptr
14.  syscall
15. end_main:
16.  li $v0, 0
17.  jr $ra
18. change_string:
19.  la $t0, hello_ptr
20.  li $t1, 'b'
21.  sb $t1, ($t0)
22.  addiu $t0, $t0, 1
23.  li $t1, 'y'
24.  sb $t1, ($t0)
25.  addiu $t0, $t0, 1
26.  li $t1, 'e'
27.  sb $t1, ($t0)
28.  addiu $t0, $t0, 1
29.  li $t1, '\n'
30.  sb $t1, ($t0)
31.  addiu $t0, $t0, 1
32.  li $t1, 0
33.  sb $t1, ($t0)
34.  j print_bye
```

- It then jumps to **line 18**.

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
1.     .data
2.  hello_ptr:  .asciiz "hello\n"
3.     .globl main
4.     .text
5.  main:
6.  print_hello:
7.     li  $v0, 4
8.     la  $a0, hello_ptr
9.     syscall
10.    j   change_string
11. print_bye:
12.    li  $v0, 4
13.    la  $a0, hello_ptr
14.    syscall
15. end_main:
16.    li  $v0, 0
17.    jr  $ra
18. change_string:
19.    la  $t0, hello_ptr
20.    li  $t1, 'b'
21.    sb  $t1, ($t0)
22.    addiu  $t0, $t0, 1
23.    li  $t1, 'y'
24.    sb  $t1, ($t0)
25.    addiu  $t0, $t0, 1
26.    li  $t1, 'e'
27.    sb  $t1, ($t0)
28.    addiu  $t0, $t0, 1
29.    li  $t1, '\n'
30.    sb  $t1, ($t0)
31.    addiu  $t0, $t0, 1
32.    li  $t1, 0
33.    sb  $t1, ($t0)
34.    j print_bye
```

- In Lines 19-30, it overwrites the bytes of the string with "**bye\n**"

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
1.     .data
2.  hello_ptr:  .asciiz "hello\n"
3.     .globl main
4.     .text
5.  main:
6.  print_hello:
7.     li  $v0, 4
8.     la  $a0, hello_ptr
9.     syscall
10.    j    change_string
11. print_bye:
12.    li  $v0, 4
13.    la  $a0, hello_ptr
14.    syscall
15. end_main:
16.    li  $v0, 0
17.    jr  $ra
18. change_string:
19.    la $t0, hello_ptr
20.    li $t1, 'b'
21.    sb $t1, ($t0)
22.    addiu  $t0, $t0, 1
23.    li $t1, 'y'
24.    sb $t1, ($t0)
25.    addiu  $t0, $t0, 1
26.    li $t1, 'e'
27.    sb $t1, ($t0)
28.    addiu  $t0, $t0, 1
29.    li $t1, '\n'
30.    sb $t1, ($t0)
31.    addiu  $t0, $t0, 1
32.    li $t1, 0
33.    sb $t1, ($t0)
34.    j print_bye
```

- It then jumps to line 11 to continue the work of **main**.

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
1.     .data
2.  hello_ptr:  .asciiz "hello\n"
3.     .globl main
4.     .text
5.  main:
6.  print_hello:
7.     li  $v0, 4
8.     la  $a0, hello_ptr
9.     syscall
10.    j   change_string
11. print_bye:
12.    li  $v0, 4
13.    la  $a0, hello_ptr
14.    syscall
15. end_main:
16.    li  $v0, 0
17.    jr  $ra
18. change_string:
19.    la $t0, hello_ptr
20.    li $t1, 'b'
21.    sb $t1, ($t0)
22.    addiu  $t0, $t0, 1
23.    li $t1, 'y'
24.    sb $t1, ($t0)
25.    addiu  $t0, $t0, 1
26.    li $t1, 'e'
27.    sb $t1, ($t0)
28.    addiu  $t0, $t0, 1
29.    li $t1, '\n'
30.    sb $t1, ($t0)
31.    addiu  $t0, $t0, 1
32.    li $t1, 0
33.    sb $t1, ($t0)
34.     j print_bye
```

- It prints the string at that same address using **lines 11-14**.

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
    .data
hello_ptr: .asciiz "hello\n"
   .globl main
   .text
main:
print_hello:
   li $v0, 4
   la $a0, hello_ptr
   syscall
   j   change_string
print_bye:
   li $v0, 4
   la $a0, hello_ptr
   syscall
end_main:
   li $v0, 0
   jr $ra
change_string:
   la $t0, hello_ptr
   li $t1, 'b'
   sb $t1, ($t0)
   addiu $t0, $t0, 1
   li $t1, 'y'
   sb $t1, ($t0)
   addiu $t0, $t0, 1
   li $t1, 'e'
   sb $t1, ($t0)
   addiu $t0, $t0, 1
   li $t1, '\n'
   sb $t1, ($t0)
   addiu $t0, $t0, 1
   li $t1, 0
   sb $t1, ($t0)
   j print_bye
```

- But the contents of the string have changed because of 5 **SB** instructions.

PRINTING STRINGS AND CHANGING STRINGS ("HELLOBYE.ASM")

```
    .data
hello_ptr:  .asciiz "hello\n"
    .globl main
    .text
main:
print_hello:
    li  $v0, 4
    la  $a0, hello_ptr
    syscall
    j   change_string
print_bye:
    li  $v0, 4
    la  $a0, hello_ptr
    syscall
end_main:
    li  $v0, 0
    jr  $ra
change_string:
    la $t0, hello_ptr
    li $t1, 'b'
    sb $t1, ($t0)
    addiu  $t0, $t0, 1
    li $t1, 'y'
    sb $t1, ($t0)
    addiu  $t0, $t0, 1
    li $t1, 'e'
    sb $t1, ($t0)
    addiu  $t0, $t0, 1
    li $t1, '\n'
    sb $t1, ($t0)
    addiu  $t0, $t0, 1
    li $t1, 0
    sb $t1, ($t0)
    j print_bye
```

- Notice we set the 5th character to 0. This *null terminates* that string.

SUMMARY: LOADING FROM AND STORING TO MEMORY

LOAD A WORD FROM MEMORY

`LW` *destination*, (*source*)

STORE A WORD TO MEMORY

`SW` *source*, (*destination*)

LOAD A BYTE FROM MEMORY

`LB` *destination*, (*source*)

STORE A BYTE TO MEMORY

`SW` *source*, (*destination*)

(OPTIONAL) EXERCISES FOR WEDNESDAY

- Change "hello.asm" so that it instead outputs this to the console

```
hello
ello
llo
lo
o
```

  - ➡Each line starts from a different place in the character string.

- Change "hello.asm" so that it instead outputs this to the console

```
hello
elloh
llohe
lohel
ohell
```

  - ➡Each line is a result of rotating the string's contents by one character.

- BONUS: have each instead operate on a string read as input from the console.
  - ➡See "string.asm" to see how to input a character string into memory.