MIPS ASSEMBLY

LECTURE 09-1

JIM FIX, REED COLLEGE CS2-S20

TODAY

- We'll look at MIPS32 assembly language programs.
- We'll run them using the SPIM simulator.
- I've posted the examples on-line, along with a copy of SPIM.

MIPS32 PROGRAM THAT SUMS FROM 1 TO 100

- Below is a MIPS32 program that computes the sum of the numbers from 1 to 100

```
1.      .globl main
2.      .text
3.
4.  main:
5.      li   $t0, 0           # sum = 0
6.      li   $t1, 1           # inc = 1
7.      li   $t2, 0           # count = 0
8.      li   $t3, 100         # last = 100
9.  loop:
10.     bgt $t3, $t2, done  # if last == count goto done
11.
12.     add $t2, $t2, $t1   # count += inc
13.     add $t0, $t0, $t2   # sum += count
14.     b   loop
15.
16. done:
17.     li   $v0, 0           # return 0
18.     jr   $ra              #
```

MIPS32 PROGRAM SNIPPET THAT SUMS FROM 1 TO 100

- Here is the "kernel" of the same program:

```
1.    li $t0, 0             # sum = 0
2.    li $t1, 1             # inc = 1
3.    li $t2, 0             # count = 0
4.    li $t3, 100           # last = 100
5.  loop:
6.    bgt $t3, $t2, done # if last == count goto done
7.    add$t2, $t2, $t1    # count += inc
8.    add$t0, $t0, $t2    # sum += count
9.    b  loop
10. done:
```

MIPS32 PROGRAM SNIPPET THAT PRINTS 1+2+...+100

- Here is the "kernel" of a similar program, and that outputs the result:

```
1.    li    $t0, 0            # sum = 0
2.    li    $t1, 1            # inc = 1
3.    li    $t2, 0            # count = 0
4.    li    $t3, 100          # last = 100
5.  loop:
6.    bgt   $t3, $t2, done    # if last == count goto done
7.    add   $t2, $t2, $t1     # count += inc
8.    add   $t0, $t0, $t2     # sum += count
9.    b     loop
10. done:
11.   li    $v0, 1            # print(sum)  -- syscall #1
12.   move  $a0, $t0          #
13.   syscall                 #
14.
```

- This MIPS32 program outputs a prompt, gets an input, sums up to that value.

```
1.          .data
2.  prompt:   .asciiz "Enter an integer: "
3.  feedback: .asciiz "The sum from 1 up to its value is "
4.  ending:   .asciiz ".\n"
5.
6.          .globl main
7.          .text
8.  main:
9.     li    $t0, 0           # sum = 0
10.    li    $t1, 1           # inc = 1
11.    li    $t2, 0           # count = 0
12.
13.    li      $v0, 4         # print(prompt) -- syscall #4
14.    la      $a0, prompt    #
15.    syscall                #
16.    li      $v0, 5         # last = input() -- syscall #5, result in $v0
17.    syscall                #
18.    move    $t3, $v0       #
19.
20. loop:
21.    bgt   $t3, $t2, done  # if last == count goto done
22.    add   $t2, $t2, $t1   # count += inc
23.    add   $t0, $t0, $t2   # sum += count
24.    b     loop
25.
```

- This MIPS32 program outputs a prompt, gets an input, sums up to that value.

```
20. loop:
21.     bgt   $t3, $t2, done  # if last == count goto done
22.     add   $t2, $t2, $t1   # count += inc
23.     add   $t0, $t0, $t2   # sum += count
24.     b     loop
25.
26. done:
27.     li      $v0, 4           # print(feedback)
28.     la      $a0, feedback  #
29.     syscall                  #
30.
31.     li      $v0, 1           # print(sum)
32.     move    $a0, $t0         #
33.     syscall                  #
34.
35.     li      $v0, 4           # print(ending)
36.     la      $a0, ending  #
37.     syscall                  #
38.
39.     li      $v0, 0           # return 0
40.     jr      $ra              #
```

A VARIANT OF SUMMING

- Here is a similar loop, but counts down, and skips summing each odd value…

```
1.          li      $t0, 0          # sum = 0
2.          move    $t2, $t3        # count = last
3.
4.  loop:
5.          beqz    $t2, done       # if count == 0 goto done
6.          andi    $t4, $t2, 0x1   # tmp = count % 2
7.          bgtz    $t4, skip_add   # if tmp > 0 go to skip_add
8.
9.  dont_skip_add:
10.         add     $t0, $t0, $t2   # sum += count
11.
12. skip_add:
13.         addi    $t2, $t2, -1    # count -= 1
14.         b       loop
15. done:
```

- This loop only sums the even elements...

```
1.  loop_head:
2.          li      $t0, 0          # sum = 0
3.          move    $t2, $t3        # count = last
4.  loop:
5.          beqz    $t2, done       # if count == 0 goto done

7.          andi    $t4, $t2, 0x1   # tmp = count % 2
8.          bgtz    $t4, skip_add   # if tmp > 0 go to skip_add
9.          addu    $t0, $t0, $t2   # sum += count
10. skip_add:
11.         addiu   $t2, $t2, -1    # count -= 1
12.         b       loop
```

A PROGRAM THAT COMPUTES A PRODUCT OF TWO INPUTS

```
1.  main:
2.     li       $v0,5
3.     syscall
4.     move     $t1,$v0
5.     li       $v0,5
6.     syscall
7.     move     $t2,$v0
8.  multiply:
9.     li       $t0,0
10. multiply_loop:
11.    beqz     $t2,report
12.    add     $t0,$t0,$t1
13.    addi    $t2,$t2,-1
14.    b        multiply_loop
15. report:
16.    li       $v0, 1
17.    move     $a0, $t0
18.    syscall
19.    li       $v0, 0
20.    jr       $ra
```

LOOP THAT COMPUTES THE PRODUCT OF TWO INTEGERS

- 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.      add     $t0, $t0, $t1   # product += x
6.      addi    $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.

LOOP THAT COMPUTES THE PRODUCT OF TWO INTEGERS

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

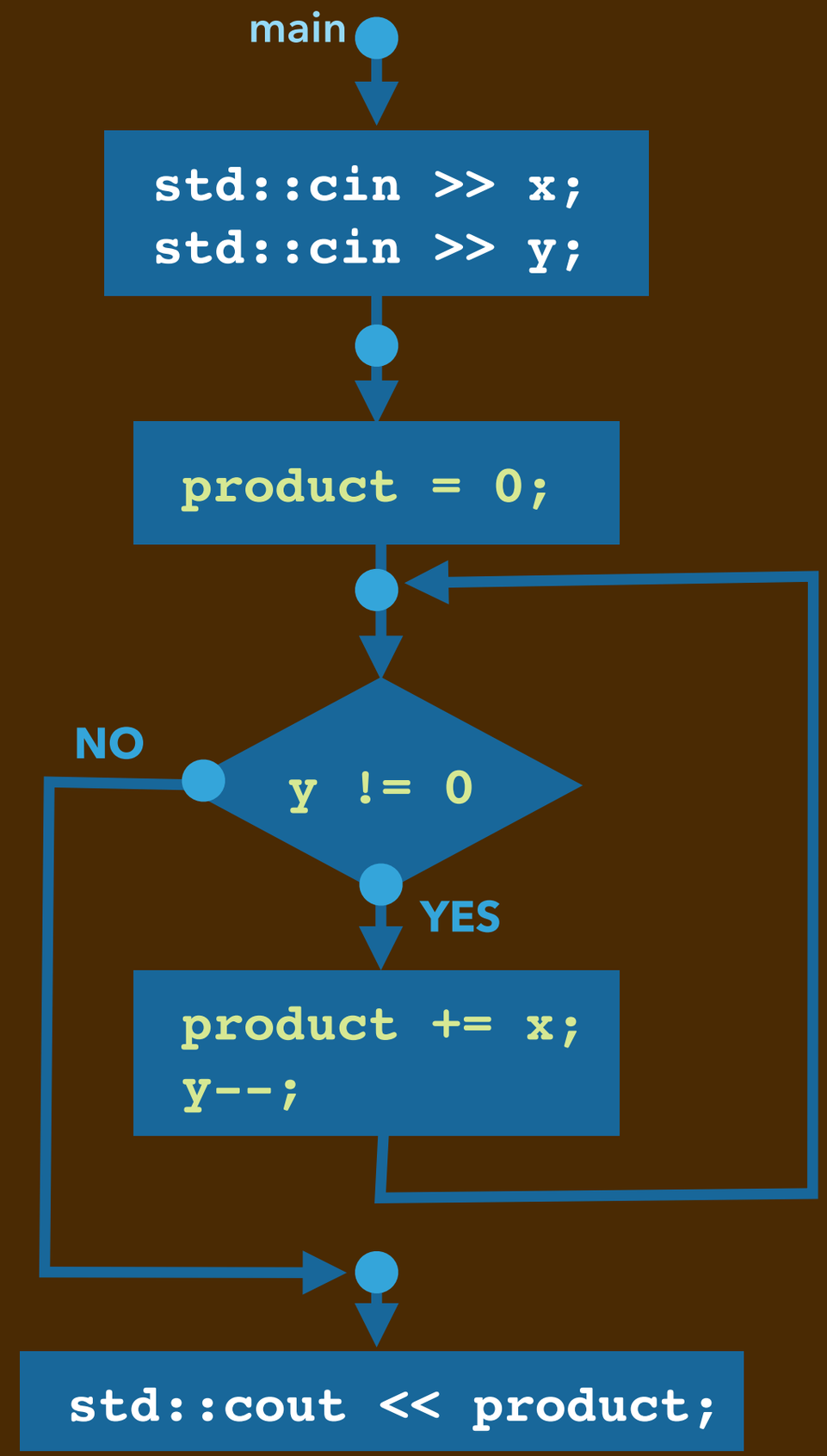
```
1.  int main() { int x,y, product;
2.     std::cin >> x;
3.     std::cin >> y;
4.     product = 0;
5.     while (y != 0) {
6.       product += x;
7.       y--;
8.     }
9.     std::cout << product;
10.    return 0;
11. }
```

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

FLOWCHART OF PRODUCT PROGRAM

```
1.  int main() { int x,y, product;
2.     std::cin >> x;
3.     std::cin >> y;
4.     product = 0;
5.     while (y != 0) {
6.        product += x;
7.        y--;
8.     }
9.     std::cout << product;
10.    return 0;
11. }
```

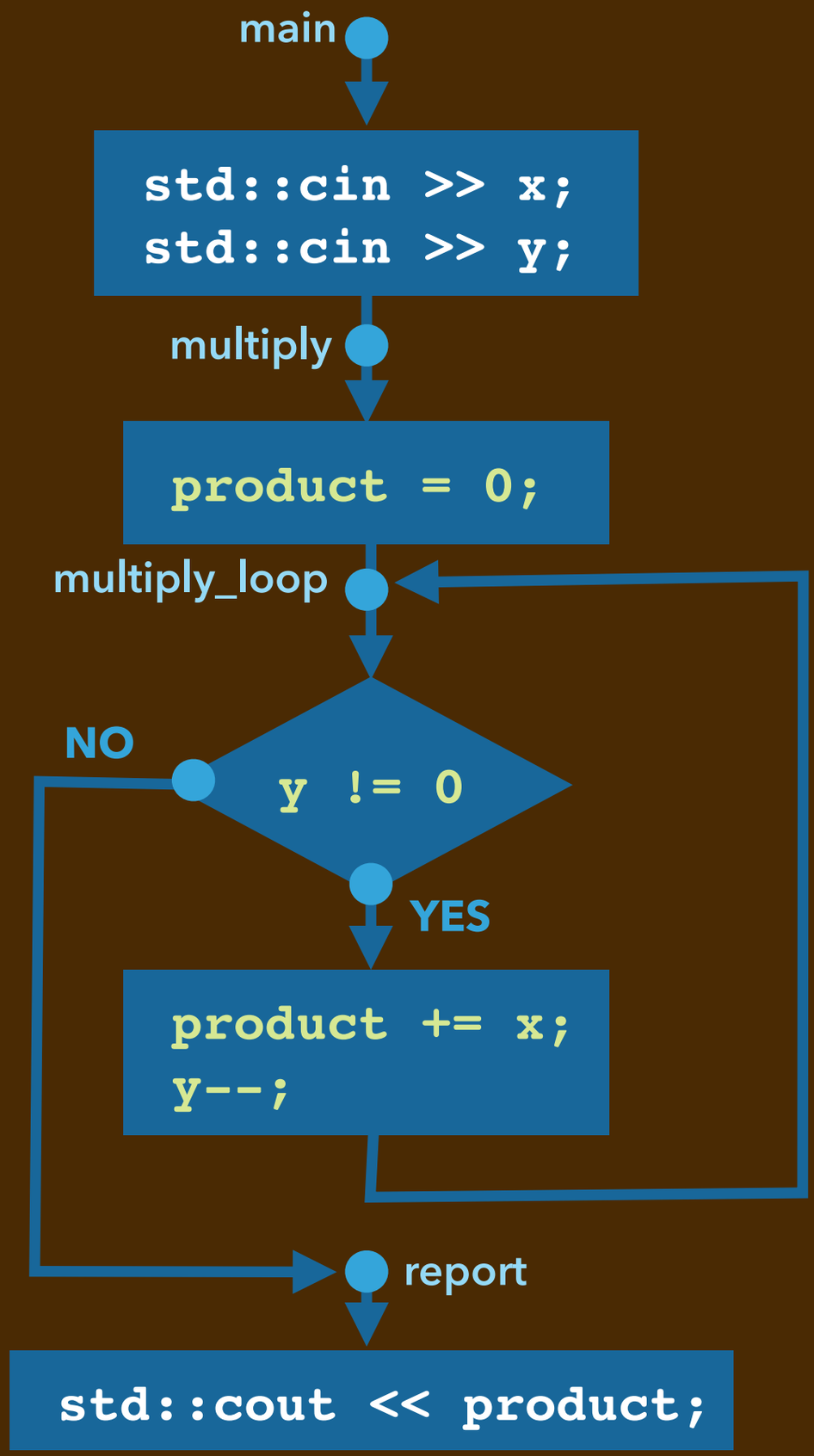
- Here is the "flow logic" of that code:
- Let's convert it to MIPS...



CONVERSION TO MIPS

```
1.  main:
2.     std::cin >> x;
3.     std::cin >> y;
4.  multiply:
5.     product = 0;
6.  multiply_loop:
7.     while (y != 0) {
8.        product += x;
9.        y--;
10.    }
11. report:
12.    std::cout << product;
13. end_main:
14.    return 0;
```

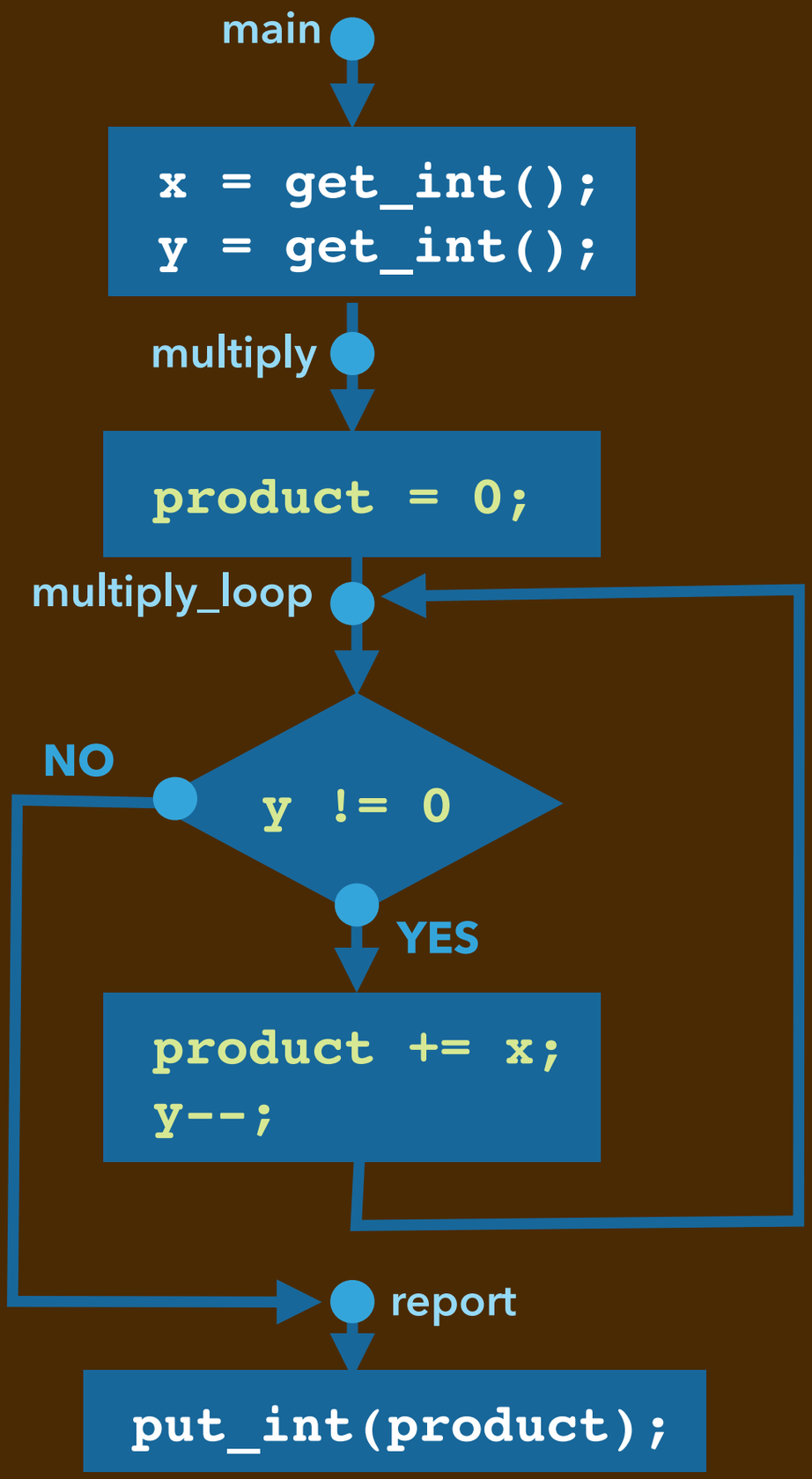
- Let's label the flow's targets.



CONVERSION TO MIPS

```
1.  main:
2.     x = get_int();
3.     y = get_int();
4.  multiply:
5.     product = 0;
6.  multiply_loop:
7.     while (y != 0) {
8.        product += x;
9.        y--;
10.    }
11. report:
12.    put_int(product);
13. end_main:
14.    return 0;
```

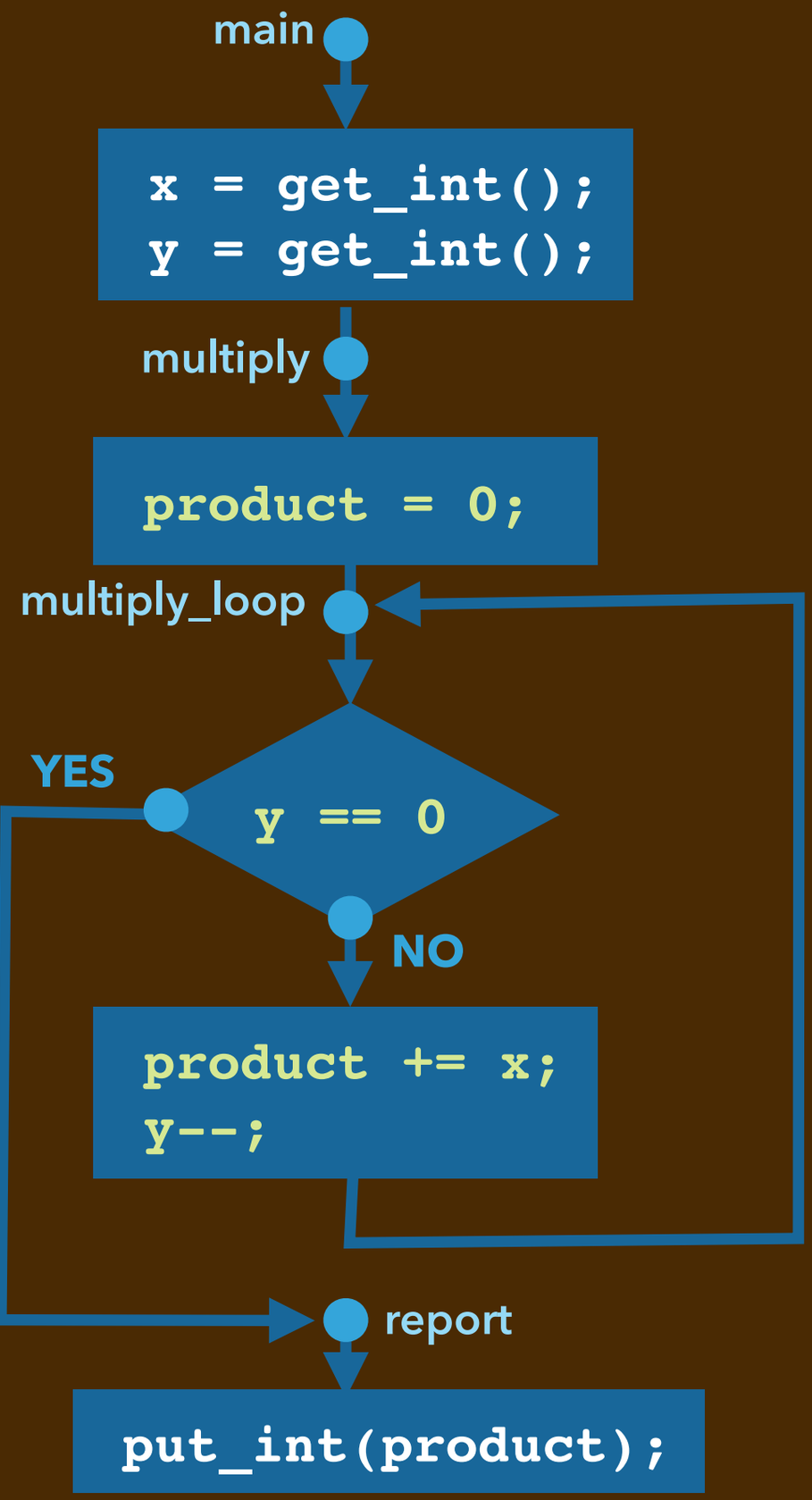
- Substitute system calls.



CONVERSION TO MIPS

```
1.  main:
2.     x = get_int();
3.     y = get_int();
4.  multiply:
5.     product = 0;
6.  multiply_loop:
7.     if (y == 0) goto report;
8.     product += x;
9.     y--;
10.    goto multiply_loop;
11. report:
12.    put_int(product);
13. end_main:
14.    return 0;
```

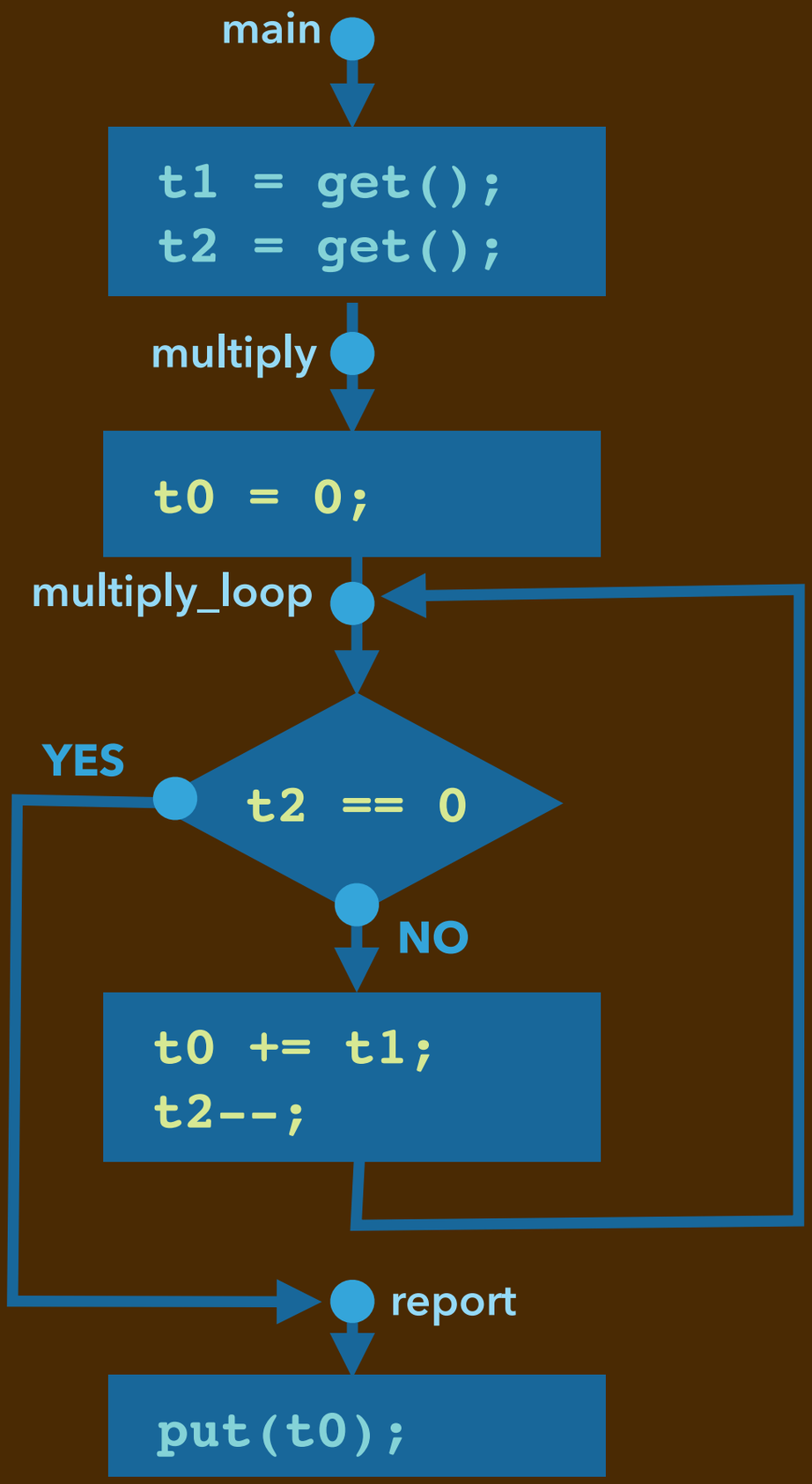
- Replace "structured" conditional statements with GOTOs.



CONVERSION TO MIPS

```
1.  main:
2.     t1 = get_int();
3.     t2 = get_int();
4.  multiply:
5.     t0 = 0;
6.  multiply_loop:
7.     if (t2 == 0) goto report;
8.     t0 += x;
9.     t1--;
10.    goto multiply_loop;
11. report:
12.    put_int(t0);
13. end_main:
14.    return 0;
```

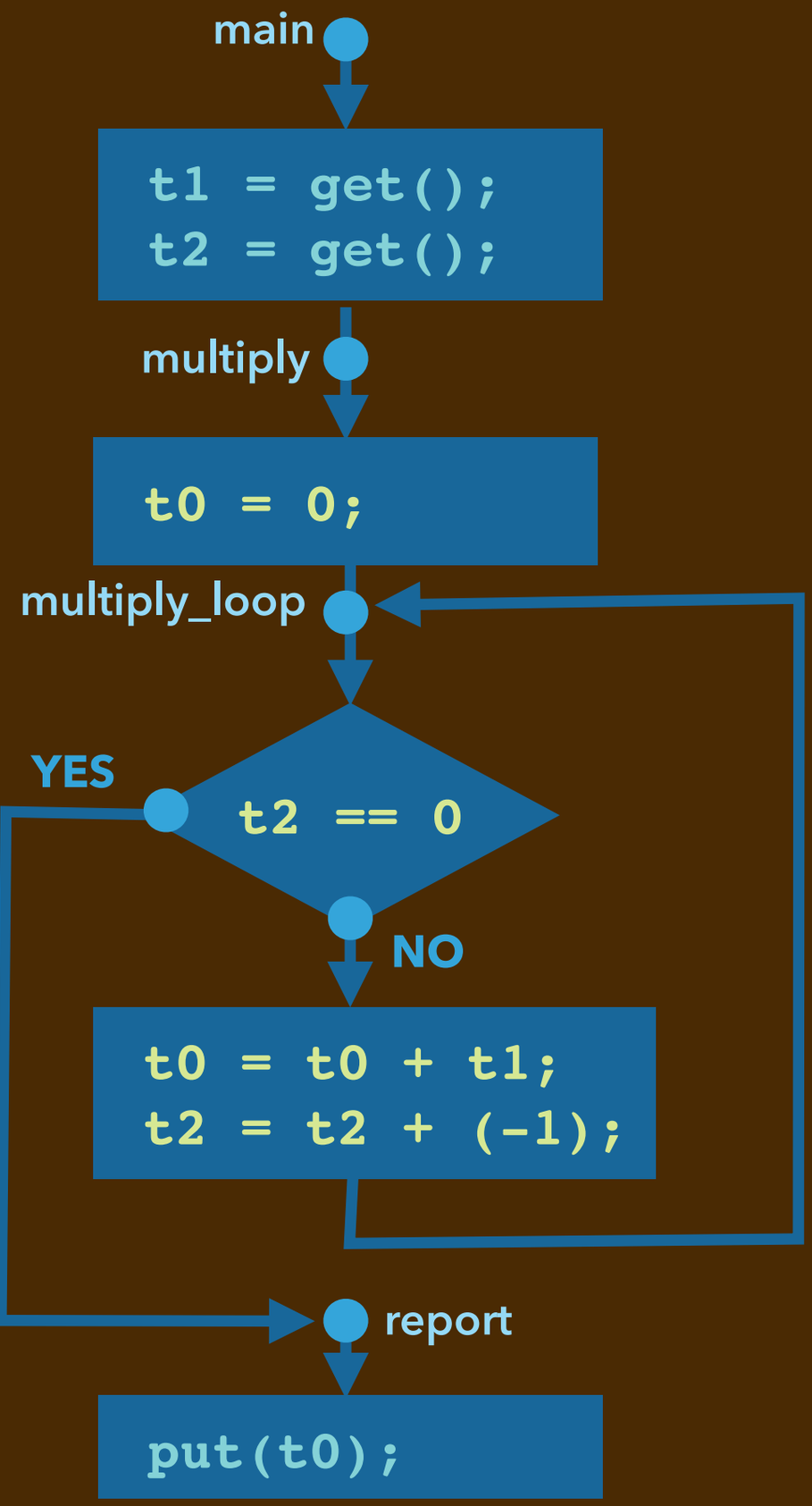
- Choose registers.



CONVERSION TO MIPS

```
1.  main:
2.     t1 = get_int();
3.     t2 = get_int();
4.  multiply:
5.     t0 = 0;
6.  multiply_loop:
7.     if (t2 == 0) goto report;
8.     t0 = t0 + t1;
9.     t1 = t1 + (-1);
10.    goto multiply_loop;
11. report:
12.    put_int(t0);
13. end_main:
14.    return 0;
```

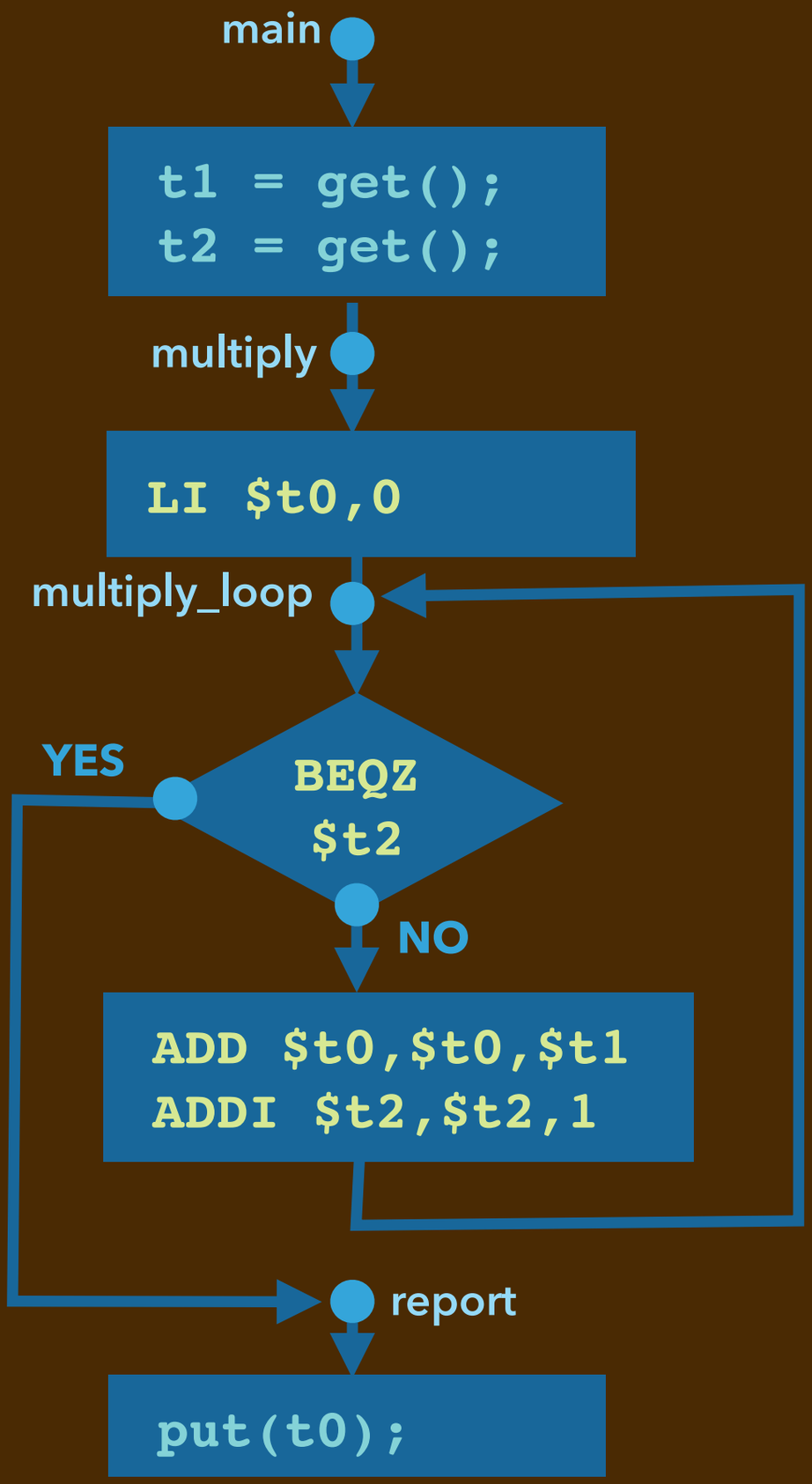
- Convert assignments to machine-like statements.



CONVERSION TO MIPS

```
1.  main:
2.     t1 = get_int();
3.     t2 = get_int();
4.  multiply:
5.     LI $t0,0
6.  multiply_loop:
7.     BEQZ $t2,report
8.     ADD $t0,$t0,$t1
9.     ADDI $t2,$t2,-1
10.    B multiply_loop
11. report:
12.    put_int(t0);
13. end_main:
14.    return 0;
```

- Change C statements to MIPS instructions.



LET'S REVIEW SOME OF THESE INSTRUCTIONS...

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

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

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

ADD

- Format:

  ADD *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
  - there's SUB 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.     ADD $t0,$t0,$t1
6.     ADDI $t2,$t2,-1
7.     B multiply_loop
8.  report:
```

ADD IMMEDIATE (UNTRAPPED)

- Format:

  ADDI *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 SUBI instruction)

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

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

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

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

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

```
1.  multiply:
2.     LI $t0,0
3.  multiply_loop:
4.     BEQZ $t2,report
5.     ADD $t0,$t0,$t1
6.     ADDI $t2,$t2,-1
7.     B multiply_loop
8.  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 $=$, $\neq$, $<$, $>$, $\leq$, $\geq$ tests with 0

LET'S REVIEW SOME OF THESE INSTRUCTIONS...

```
1.  multiply:
2.     LI $t0,0
3.  multiply_loop:
4.     BEQZ $t2,report
5.     ADD $t0,$t0,$t1
6.     ADDI $t2,$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...

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

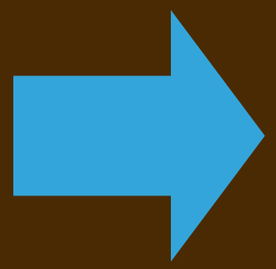
  ➡ Each has a number that identifies it.

  ➡ 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();
```



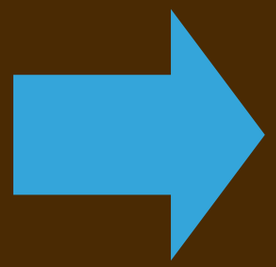
```
LI $v0,5
syscall
MOVE $t1,$v0
```

- 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
    ```

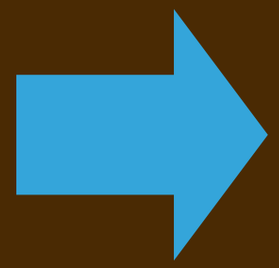
    would be like the C++ code

    ```
    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.***

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.
- The program "output100.s" below 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.s" program:

```
1.              .data
2.  report:     .asciiz "The value held in memory is "
3.  dot_eoln:   .asciiz ".\n"
4.  value:      .word 100
5.              .globl main
6.              .text
7.  main:
8.     la       $a0,report
9.     li       $v0,4
10.    syscall
11.    la       $t0,value
12.    lw       $a0,($t0)
13.    li       $v0,1
14.    syscall
15.    la       $a0,eoln
16.    li       $v0,4
17.    syscall
18.    li       $v0,0
19.    jr       $ra
```

LOADING A VALUE STORED AT AN ADDRESS

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

```
1.              .data
2.  report:     .asciiz "The value held in memory is "
3.  dot_eoln:   .asciiz ".\n"
4.  value:      .word 100
5.              .globl main
6.              .text
7.  main:
8.    la        $a0,report
9.    li        $v0,4
10.   syscall
11.   la        $t0,value
12.   lw        $a0,($t0)
13.   li        $v0,1
14.   syscall
15.   la        $a0,dot_eoln
16.   li        $v0,4
17.   syscall
18.   li        $v0,0
19.   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.s" program:

```
1.              .data
2.  report:     .asciiz "The value held in memory is "
3.  dot_eoln:   .asciiz ".\n"
4.  value:      .word 100
5.              .globl main
6.              .text
7.  main:
8.     la       $a0,report
9.     li       $v0,4
10.    syscall
11.    la       $t0,value
12.    lw       $a0,($t0)
13.    li       $v0,1
14.    syscall
15.    la       $a0,dot_eoln
16.    li       $v0,4
17.    syscall
18.    li       $v0,0
19.    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.s" program:

```
1.              .data
2.  report:     .asciiz "The value held in memory is "
3.  dot_eoln:   .asciiz ".\n"
4.  value:      .word 100
5.              .globl main
6.              .text
7.  main:
8.     la       $a0,report
9.     li       $v0,4
10.    syscall
11.    la       $t0,value
12.    lw       $a0,($t0)
13.    li       $v0,1
14.    syscall
15.    la       $a0,dot_eoln
16.    li       $v0,4
17.    syscall
18.    li       $v0,0
19.    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.  ADDI  $t0,$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.S")

```
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
```

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

```
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
```

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

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

```
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.S")

```
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.S")

```
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.S")

```
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.S")

```
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.S")

```
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
```

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

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

```
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
```

- 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)
```

LOADING FROM AND STORING TO 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 a "fetch."
  - ***source*** is a register holding the address where we're fetching the data in ***destination***

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 of ***source***