Jumps vs Branches

- Jump command takes you to the specified label
- Jumps, however, do not do any comparisons (they are unconditional)

Jump region (J)

- J can jump to a label that is within the current region

  - Main reason for having a separate Jump instruction instead of simply writing:
    beq $0, $0, jump_label?

  - **Reason**: separate jump instruction allows us to jump much farther than branch encoding.
    Will be discussed more in advanced course CSE120; no need to worry about it now!

Address	**Instruction**
8000	J 12000;
8004	Add $t4,$t4,$t5;
………	
………	
12000	Addi $t5, $t3, 10;

Jump Example

```
j blah_label
addi $t1, $0, 1
addi $t2, $0, 2
blah_label:
        addi $t3, $0, 3
```

Bkpt	Address	Code	Basic	Source
☐	0x00400000	0x08100003	j 0x0040000c	1: j blah_label
☐	0x00400004	0x20090001	addi $9,$0,0x00000001	2: addi $t1, $0, 1
☐	0x00400008	0x200a0002	addi $10,$0,0x0000...	3: addi $t2, $0, 2
☐	0x0040000c	0x200b0003	addi $11,$0,0x0000...	5: addi $t3, $0, 3

Only 2 command executed:

- j blah_label
- addi $t3, $0, 3

Jump Register (JR)

- JR uses the instruction address in a register
  - Example: JR $t0
  - Full 32-bit address in register

- JR instruction returns control to the caller. It copies the contents of $t0 into the PC (program counter), that keeps track of which instruction computer is currently executing

- Usually you think of this as "jumping to the address contained in $t0."

Address	**Instruction**
8000	Addi $t2, 12000
8004	Jr $t2
………	
………	
12000	Addi $t5, $t3, 10

Jumping (and Linking) : JAL

- Some jumps can store the address of the following instruction
  - Why?....Writing Functions **(will discuss more in later slides!)**
- This is known as “linking”
  - Similar behavior to J except next address (PC+4) is **automatically** remembered in $ra (return address) ($r31) register

Address	**Instruction**
8000	Jal 12000; // *PC=8000, therefore ($ra)=PC+4 = 8004*
8004	Add $t4, $t4,$t3;
………	
………	
12000	Addi $t5, $t3, 10
12004	Jr $ra

Jump and Link Example in MARS

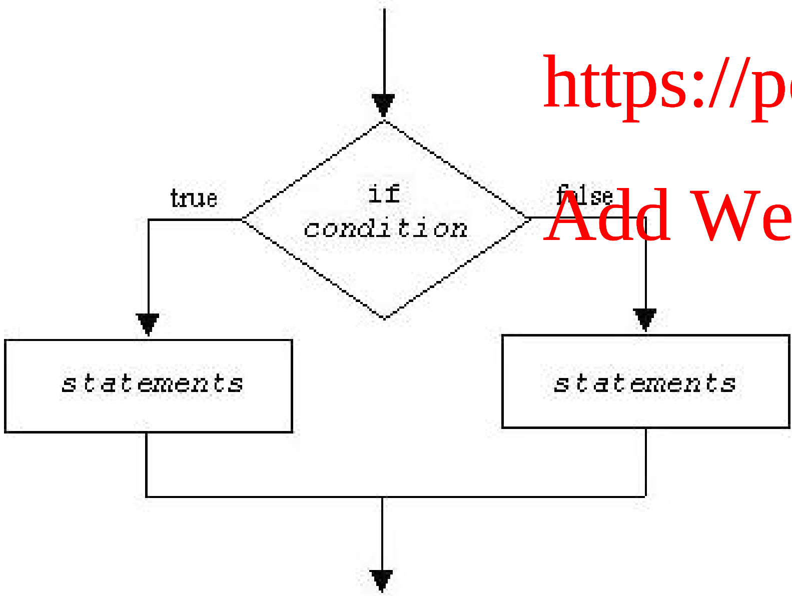
```
addi $s0, $0, 0              # set s0 to 0
jal increment_s0             # Jump to function increment s0
addi $s0, $0, 5              # Add 5 to S0
j end_prog                   # jump to end of the program
increment_s0:
        addi $s0, $s0, 1     # Increment s0 by 1
        jr $ra               # Go to next instruction after function call

end_prog:
        li $v0, 10           # set $v0 to 10
        syscall              # syscall with v0 = 10 means exit
```


Jump instructions summary

Instruction	Example	Meaning	Comments
jump	`j 1000`	go to address 1000	Jump to target address
jump register	`jr $t1`	go to address stored in $t1	For switch, procedure return
jump and link	`jal 1000`	$ra=PC+4; go to address 1000	Use when making procedure call. This saves the return address in $ra

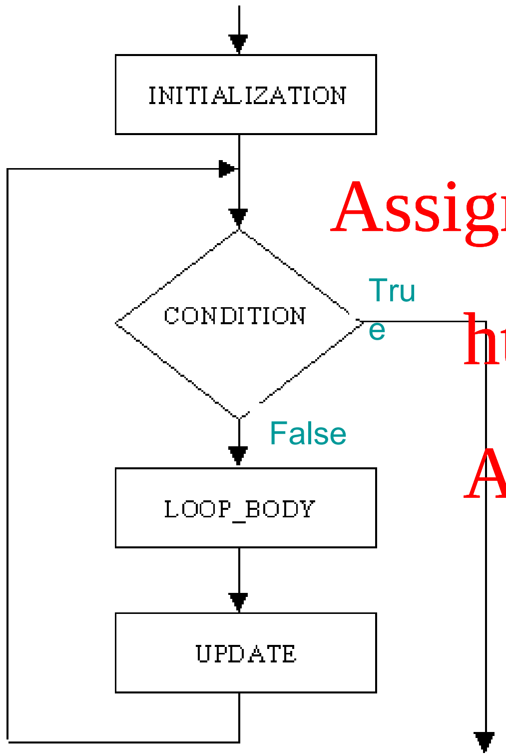
Program Control Flow

- You can conditionally execute sections of code using BEQ and B
- Why do you need the B?



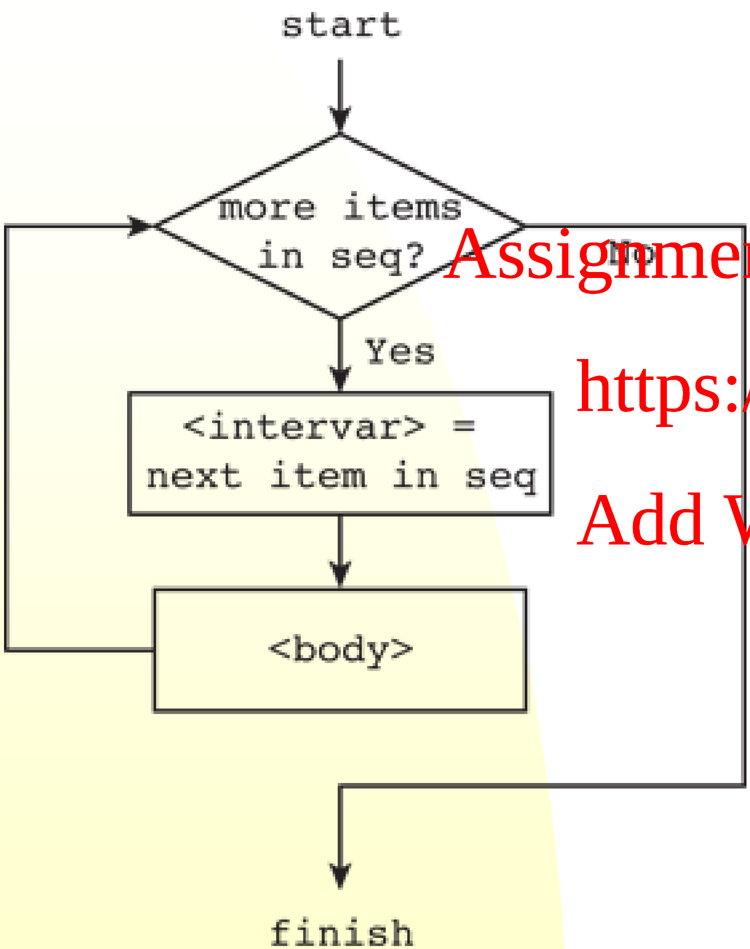

```
BEQ $t0, $t1, true_condition

false_condition:
        # statements
        B end_condition

true_condition:
        # statements

end_condition:
        # after branch
```

Branching in a Loop




```
init:
        li $t0, 0
loop_start:
        beq $t0, 10, loop_end
loop_body:
        move $a0, $t0
        li $v0, 1
        syscall
update:
        addi $t0, $t0, 1
        b loop_start
loop_end:
        nop
```

Iterating over a String




```
.text:
init:
        la $t0, hello_string
loop_start:
        lb $t2, ($t0)
        beq $t2, $zero, loop_end
loop_body:
        move $a0, $t2
        li $v0, 11
        syscall
update:
        addi $t0, $t0, 1
        b loop_start
loop_end:
        nop
.data
hello_string:
.asciiz "Hello World!"
```

Pseudo Instructions (or Pseudo-ops)

- The MIPS assembler contains many ”pseudo” instructions that look like instructions, but actually get mapped to other instructions.
  - These are intended to save the programmer time and make code more readable (fewer instructions in written code)
  - These remove “redundant” instructions (simpler processor)

Pseudo-op: NOT

- NOT can be implemented with NOR
  - Remember: x nor 0 = !x
  - Example:
    - NOT $t1, $t2
    - Gets mapped to: NOR $t1, $t2, $zero

Pseudo-Op: Rotate

- Rotate Left (ROL) and Rotate Right (ROR) are pseudo-ops
  - Use SLL/SLLV and SRL/SRLV to generate intermediate results
  - OR rotated bits back in
- Example:
  - ROL $t1, $t2, 3
- Translates to:
  - SRL $at, $t2, 29
  - SLL $t1, $t2, 3
  - OR $t1, $t1, $at

Pseudo-op: Load address (recommended!)

- LA $t1, label
  - There is no la instruction in MIPS!
  - Loads address of label into register:
    - LUI $t1, upper-16-bits
    - ORI $t1, lower-16-bits

Pseudo-op: Load Immediate

- Adding to zero is the same as load immediate
- Can put 16-bit value in instruction itself Load Immediate Signed
  - Example (using decimal notation):

    - LI $t1, 23

      - This is actually ADDIU $t1, $zero, 23

    - $t1 = 0 + 23
  - Can use hex notation:
    - LI $t1, 0x0F
    - $t1 = 0x0F
  - Can do negative immediate too
    - LI $t1, -23 is ADDI $t1, $zero, -23

Pseudo-op: Load 32-bit immediate

- There is a 32-bit load-immediate pseudo-op too!
  - Example:
    - LI $t1, 0x1234FFFF
  - Translates to:
    - LUI $t1, 0x1234
    - ORI $t1, 0xFFFF

Pseudo-op: Move

- To initialize a register
  - MOVE $t0, $zero
- Translates to
  - ADDU $t0, $zero, $zero

- To move a register
  - MOVE $t1, $t0
- Translates to
  - ADDU $t1, $t0, $zero

Using MIPS in MARS

- Registers are case sensitive
  - $A0 is not $a0 (gives error)
- Operations are case insensitive
  - ADD is same as add
- Use the symbolic register name like $t0
  - Not $r1 but $1 in MARS (gives error)
  - Recommend using $t0-$t7 for now… more later!

Table 1: Register Conventions

CPU Register	Symbolic Register	Usage
r0	zero	Always 0 (note 1)
r1	at	Assembler Temporary
r2 - r3	v0-v1	Function Return Values
r4 - r7	a0-a3	Function Arguments
r8 - r15	t0-t7	Temporary – Caller does not need to preserve contents
r16 - r23	s0-s7	Saved Temporary – Caller must preserve contents
r24 - r25	t8 - t9	Temporary – Caller does not need to preserve contents
r26 - r27	k0 - k1	Kernel temporary – Used for interrupt and exception handling
r28	gp	Global Pointer – Used for fast-access common data
r29	sp	Stack Pointer – Software stack
r30	s8 or fp	Saved Temporary – Caller must preserve contents OR Frame Pointer – Pointer to procedure frame on stack
r31	ra	Return Address (note 1)

Note 1: Hardware enforced, not just convention

System Call (syscall)

- Calls special system code to do things like…
  - Input from keyboard
  - Output to screen
  - Exit program
- Code for system call is in register $v0
- Arguments are (if needed) in $a0 and $a1

Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*

Hello World!

```
.text                        # Define the program instructions.

main:                        # Label to define the main program.
       li $v0,4              # Load 4 into $v0 to indicate a
                             # print string.
       la $a0, greeting      # Load the address of the greeting
                             # into $a0.
       syscall               # Print greeting. The print is
                             # indicated by $v0 having a value
                             # of 4, and the string to print
                             # is stored at the address in $a0.
       li $v0, 10            # Load a 10 (halt) into $v0.
       syscall               # The program ends.

.data                        # Define the program data.
greeting:
.asciiz "Hello World"        #The string to print (null
terminated!).
```

Conclusion

- You should know the instructions presented here
- There are some other less frequent ones
  - You don't need to remember them!
  - You should be able to understand them given the ISA information (e.g., the manual)!

Data Layout

References:

1) MIPS_Vol2.pdf

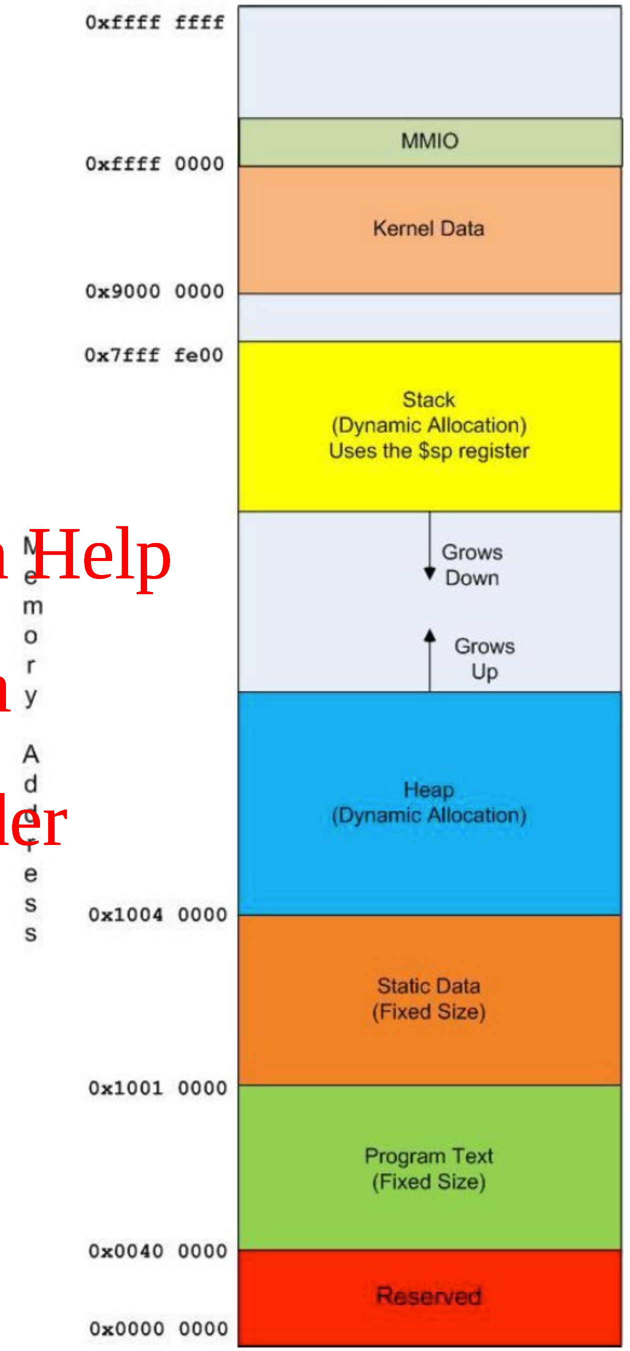
2) Intro to MIPS Assembly Language Programming

Where does the data come from?

- Data is arranged in memory
- Data can be initialized when a program starts (.data)
  - Must be stored in the program with the instructions
- Data may be not-initialized (or actually just zeroed) (.bss)
  - This doesn't need to be stored in the program itself
- Data may be created during execution
  - More later in the course!

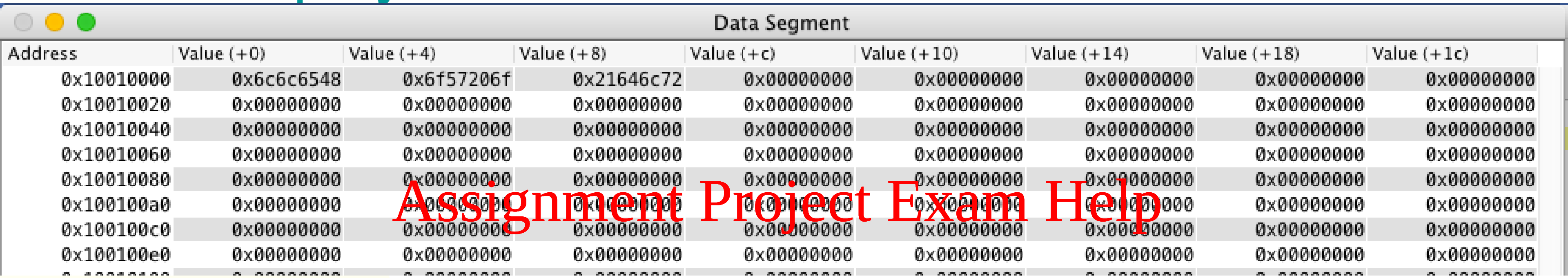
Code and Data Addresses

- The code (text) segment starts at 0x0040_0000
  - Each instruction is 4 bytes
- The data segment starts at 0x1001_0000
  - Data is placed at next adjacent location
- Both grow “up”
- What is the limit of my program size?
- What is the limit of my data size?



MARS Data Layout Viewer: Hello World

- Displayed as hexadecimal



Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x6c6c6548	0x6f57206f	0x21646c72	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

Data Declarations

```
.data                 # All of this will go in the "data" segment of memory
var1: .word 3         # create a single integer variable with initial value 3

array1: .byte 'a','b'      # create a 2-element character array
                           # initialized to ASCII a and b

half1:   .half 0x1234 # create 16-bit word

array2: .space 40          # allocate 40 consecutive bytes, with storage
                           # uninitialized

string1: .asciiz "Hello!\n" # string variable with end null

string2: .ascii  "Hello!\n" # string variable with NO end null
```

Data Segment Layout (Packing)

- Memory is arranged least significant byte first
  - .align directive can force alignment

```
.data
var1: .word 3
array1: .byte 'a','b'

half1: .half 0x1234
string1:.asciiz "Hello!\n"
```


Address	Value
0x1001_0000	03
0x1001_0001	00
0x1001_0002	00
0x1001_0003	00
0x1001_0004	a
0x1001_0005	b
0x1001_0006	34
0x1001_0007	12
0x1001_0008	H
0x1001_0009	e
0x1001_000A	l
0x1001_000B	l
0x1001_000C	o
0x1001_000D	!
0x1001_000E	\n
0x1001_000F	00

Computing Data Addresses

- If data segment starts at 0x1001_0000…
  - What is var1?
  - What is array1?
  - What is half1?

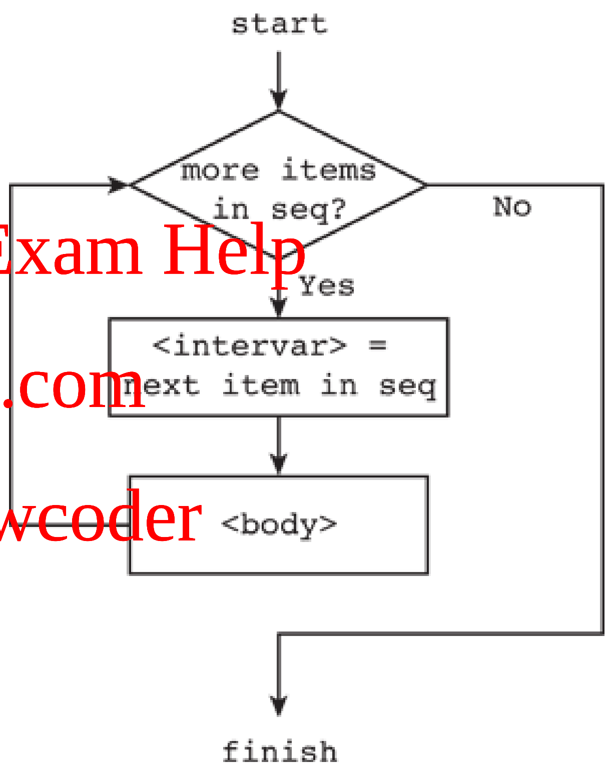
Label	Address	Value
var1	0x1001_0000	03
	0x1001_0001	00
	0x1001_0002	00
	0x1001_0003	00
array1	0x1001_0004	61 (a)
	0x1001_0005	62 (b)
half1	0x1001_0006	34
	0x1001_0007	12

```
.data
var1: .word 3
array1: .byte 'a','b'

half1:
 .half 0x1234
```

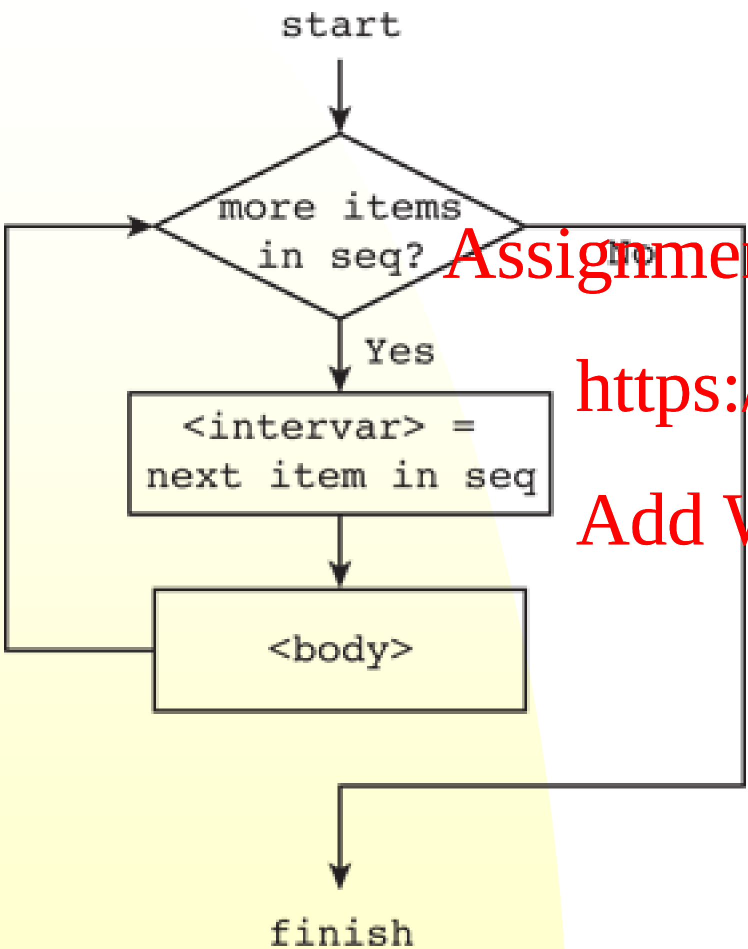
Spaces or new lines don't matter!

Arrays

- Arrays are a sequence of identical elements
  - Elements can be any size: byte, half word, word, or bigger
  - Strings are an array of characters
- Remember iterating over a sequence



Iterating over a String (Array of Characters)




```
.text:
init:
        la $t0, hello_string
loop_start:
        lb $t2, ($t0)
        beq $t2, 0, loop_end
loop_body:
        move $a0, $t2
        li $v0, 11
        syscall
update:
        addi $t0, $t0, 1
        b loop_start
loop_end:
        nop
.data
hello_string:
.asciiz "Hello World!"
```

Array of Integers (4 bytes)

```
.data
list:
.word 3, 0, 1, 2, 6, -2, 7, 3, 7

.text
la $t3, list            # put address of list into $t3

li $t2, 3               # put the index into $t2
sll $t2, $t2, 2         # Multiple index by 4 (1 word is 4 bytes)
                        # $t2 = 3 * 4 = 12

add $t1, $t2, $t3       # combine the two parts of the
address

lw $t4, 0($t1)          # get the value from the array cell
```

Address	Value
0x1001_0000	03
0x1001_0001	00
0x1001_0002	00
0x1001_0003	00
0x1001_0004	00
0x1001_0005	00
0x1001_0006	00
0x1001_0007	00
0x1001_0008	01
0x1001_0009	00
0x1001_000A	00
0x1001_000B	00
0x1001_000C	02
0x1001_000D	00
0x1001_000E	00
0x1001_000F	00
………..	……..