Macros

- Uses:
  - to improve readability of your program
  - to eliminate repetitive code construction
  - to reduce overhead associated with subroutines
- Register usage must be carefully considered
- Register Approaches:
  - New Pseudo Instruction: use the $at register -- but <u>don't</u> use any pseudo instructions
  - Marshalling: use only the registers provided as arguments
  - Inlined- subroutine: Utilize: $a0, $a1, $a2, $a3, $v0, and $v1
- Syntax:

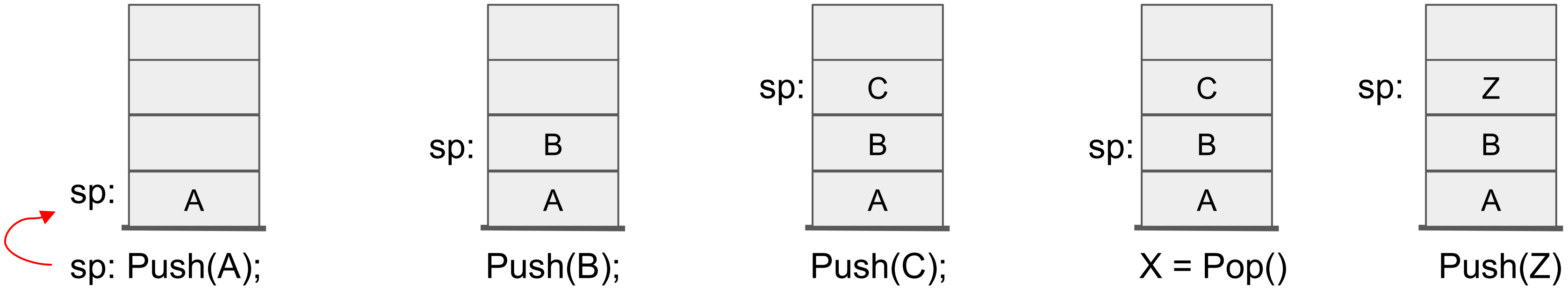
```
.macro  <name>(%arg1 .. %argn)
    <list of native instructions>
.end_macro
```

Stack Operations

Push(a) ⇔
sp = sp + 1
sp[0] = a

x = Pop() ⇔
x = sp[0]
sp = sp - 1

- Stack is an abstract data structure
- The stack is an array of <u>words</u>
- Operations:
  - Push: Push(A), Push(B), Push(C)
  - Pop: X = Pop();
  - Push: Push(Z);



But the MIPS Way

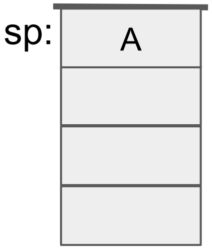
```
Push(a) ⇔
    sp = sp - 1
    sp[0] = a
```

```
x = Pop() ⇔
        x = sp[0]
        sp = sp + 1
```

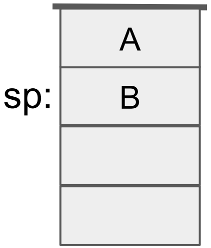
- Stack is an abstract data structure
- Operations:
  - Push: Push(A), Push(B), Push(C)
  - Pop: X = Pop();
- sp: points to the current top of stack

```
Push(a) ⇔
    subi $sp, $sp, 4
    sw $a0, 0($sp)
```

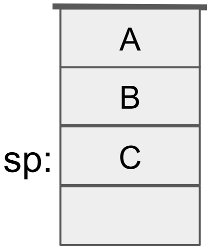
```
x = Pop() ⇔
        lw $v0, 0($sp)
        addi $sp, $sp, 4
```



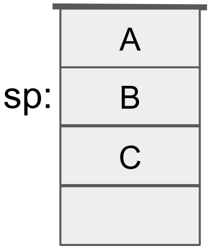
Push(A);



Push(B);



Push(C);



X = Pop()

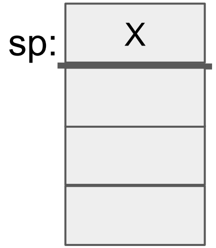
Multiple Pushes / Pops

```
Push(a) ⇔
    sp = sp - 1
    sp[0] = a
```

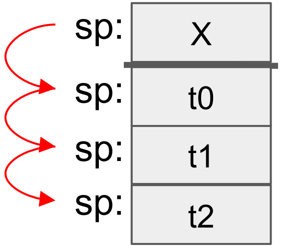
```
x = Pop() ⇔
        x = sp[0]
        sp = sp + 1
```

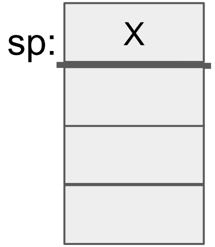
```
Push(a) ⇔
    subi $sp, $sp, 4
    sw $a0, 0($sp)
```

```
x = Pop() ⇔
        lw $v0, 0($sp)
        addi $sp, $sp, 4
```

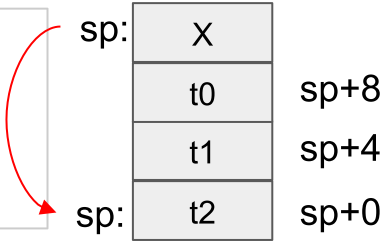



```
Push(t0);
Push(t1);
Push(t2);
```






```
subi $sp, $sp, 12
sw $t0, 8($sp)
sw $t1, 4($sp)
sw $t2, 0($sp)
```




```
t0 = Pop();
t1 = Pop();
t2 = Pop();
```

```
lw $t0, 8($sp)
lw $t1, 4($sp)
lw $t2, 0($sp)
addi $sp, $sp, 12
```

Printf: `print(variable); println("string");`

- Java Prototype
  - `public PrintStream printf(String format, Object... args)`
- Java Example
  - `printf("the value of x is %d", x);`
- Format Specifier,
  - %conversion
- Format Conversions:
  - d: decimal, u: unsigned decimal
  - o: octal
  - x: hexadecimal
  - c: character
  - s: string
  - f: floating point
  - ~~t: binary (bi"t")~~
- MIPS Macros:
  - print_d, print_u
  - print_o
  - print_x
  - print_c
  - print_s
  - ~~print_f~~
  - print_t

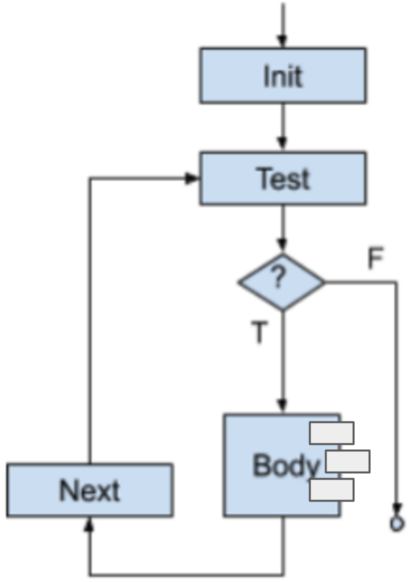
Control Flow Graph

- A graphic representation of the representation between basic blocks
- A basic block:
  - a list of instructions with
  - a single entry point (starting point)
  - a single exit point (last instruction)
- Such representations model the behavior of our code
- Recall the while loop, and other control structures

- What about subroutines calls

  (subroutine: general term for …

  methods, functions, procedures, etc.)



While Loop

Three Address Code (TAC)

- A generic assembly language in which <u>all</u> instructions have <u>at most</u> three addresses

- An address references either
    - a register location
    - a memory location
    - an immediate value, e.g., “2”

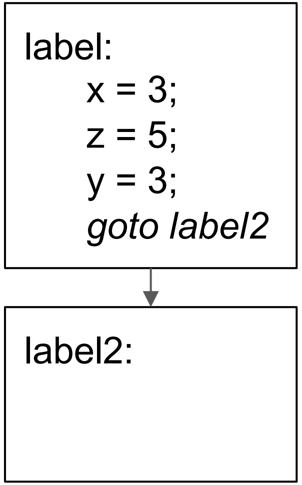
- Immediate values are stored as part of the instruction in a location

Examples:

1. a = y + x
2. a = y
3. a = x + 2
4. b = d * 2 + y
    - t0 = d * 2
    - t1 = t0 + y
    - b = t1

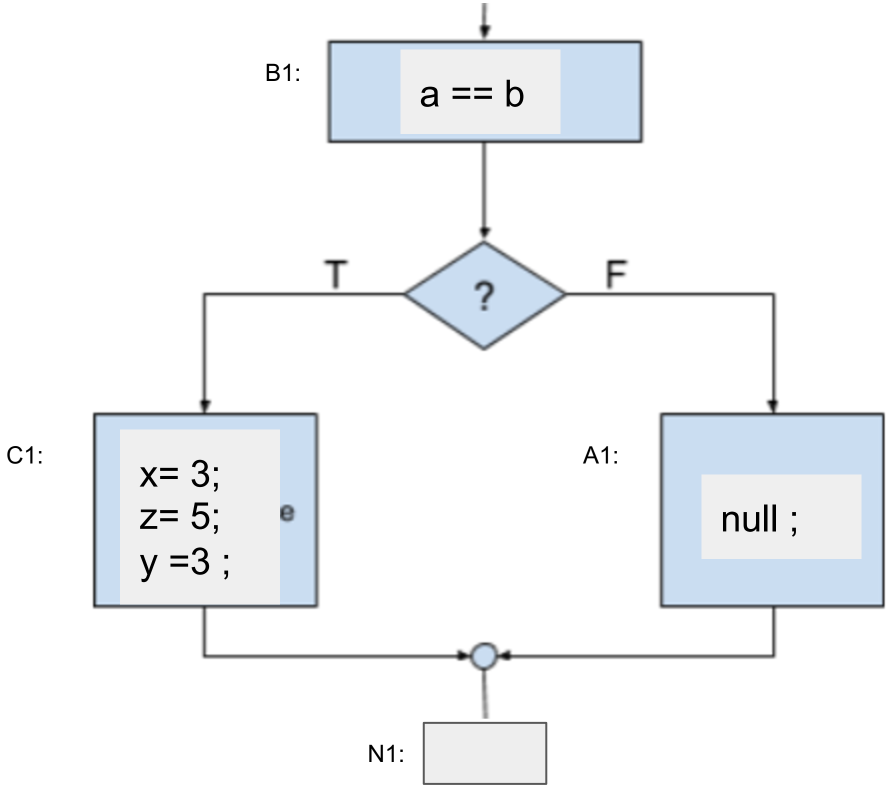
Basic Blocks

- A number of instructions in which there is
  - a single entry point (via a label), and
  - a single exit point (via a goto)
- All programs can be broken down into a set of basic blocks
- A control flow graph determines which a basic block is executed.
- Standard control flow graphs
  - if-then-else and all other variants (e.g., switch)
  - while, do-while and all other variants
  - for loop and all other variants
  - call-return



Code Flow: If-then-else

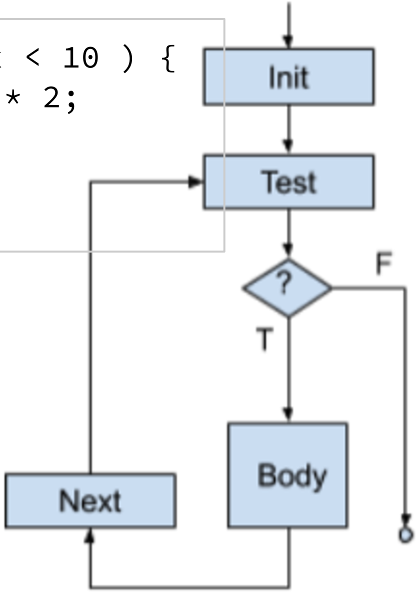
```
if (          ){

} else {

}
```




```
B1:

    if true goto C1
    goto A1

C1:

A1:

N1:
```

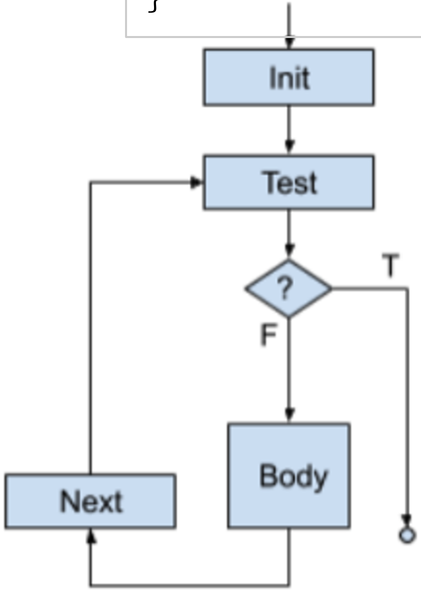
Control Flow: Loops

```
while ( x < 10 ) {
    a = x * 2;

    x ++;
}
```



While Loop

```
until ( x>= 10 {
  a = x * 2;

  x ++;
}
```



Until Loop

```
do {
    a = x * 2;

    x ++;
} while ( x < 10);
```

Do While Loop

```
for ( i = 0; i < 10 ; i++ ) {
   a = x * 2
}
```