# PROCEDURE CALLS

Mahdi Nazm Bojnordi

Assistant Professor

School of Computing

University of Utah

# Overview

- Notes
  - Homework 3 is due tonight
    - Verify your uploaded file before the deadline
  - A full list of MIPS instructions can be find in the textbook—e.g., page 64.
    - Please do not invent an new instruction format
- This lecture
  - Procedure calls

# Recall: Control Instructions

- Determine which instruction to be executed next
  - Conditional branch: Jump to instruction L1 if register1 equals register2
    - beq  register1, register2, L1
    - bne, slt (set-on-less-than), slti
  - Unconditional branch: Jump to instruction L1
    - J  L1
    - Jr  $s0 (jump table; long jumps and case statements)

# Functions and Procedures

- Example C code: Bubble sort

```
4   for (c = 0 ; c < n - 1; c++) {
5     for (d = 0 ; d < n - c - 1; d++) {
6       if (array[d] > array[d+1]) {
7         swap       = array[d];
8         array[d]   = array[d+1];
9         array[d+1] = swap;
10      }
11    }
12  }
```

# Functions and Procedures

- Example C code: Bubble sort

```
4   for (c = 0 ; c < n - 1; c++) {
5     for (d = 0 ; d < n - c - 1; d++) {
6       if (array[d] > array[d+1]) {
7         swap       = array[d];
8         array[d]   = array[d+1];
9         array[d+1] = swap;
10      }
11    }
12  }
```

**Caller**

```
4   for (c = 0 ; c < n - 1; c++) {
5     for (d = 0 ; d < n - c - 1; d++) {
6       process();
7     }
8   }
```

**Callee**

```
13  process() {
14    if (array[d] > array[d+1]) {
15      swap       = array[d];
16      array[d]   = array[d+1];
17      array[d+1] = swap;
18    }
19  }
```

# Procedure Calls

- How to implement function/procedure calls
  - Using Jumps

```
main:
                j myFunction
Laftercall1: add $1,$2,$3
                …
```

```
myFunction:
                …
                j Laftercall1
```

# Procedure Calls

- How to implement function/procedure calls
  - Using Jumps
    - What happens if there are multiple calls?

```
main:
                j myFunction
Laftercall1: add $1,$2,$3
                …
                j myFunction
Laftercall2:  sub $3,$4,$5
```

```
myFunction:
                …
                j Laftercall1
```

# Procedure Calls

- How to implement function/procedure calls
  - Using Jumps
    - What happens if there are multiple calls?
  - Using JAL (Jump-and-Link) and JR
    - Store the next instruction's address in $ra

```
main:
                jal myFunction
Laftercall1: add $1,$2,$3
                …
                jal myFunction
Laftercall2:  sub $3,$4,$5
```

```
myFunction:
                …
                jr  $ra
```

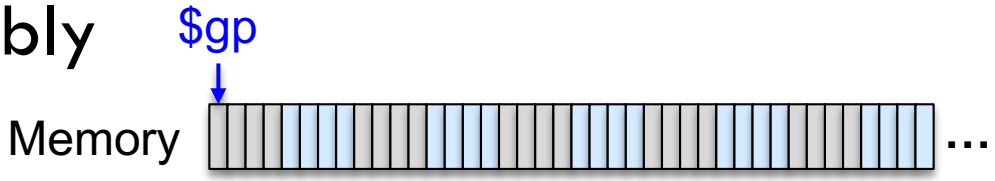
# Example Code

- Convert to Assembly

```
int  x, array[100];


main() {
   int i;
   x = 5;
   for(i=0; i < 100; ++i) {
       increment(i);
   }
}


void increment(int d) {
   int i = 12;
   array[d] = x + i;
}
```

# Example Code

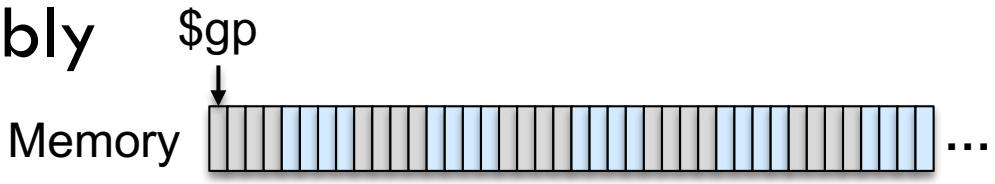
- Convert to Assembly




```
int  x, array[100];


main() {
   int i;
   x = 5;
   for(i=0; i < 100; ++i) {
       increment(i);
   }
}


void increment(int d) {
   int i = 12;
   array[d] = x + i;
}
```

# Example Code

- Convert to Assembly



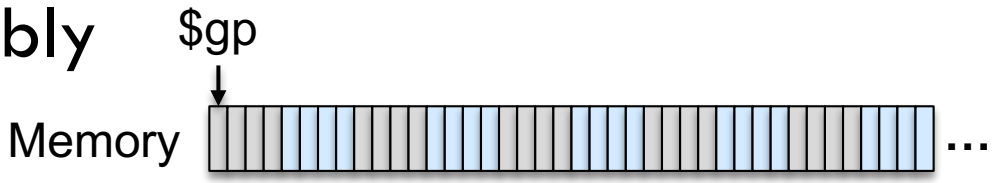

```
int  x, array[100];


main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}


void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

$s0

```
main:
```

# Example Code

- Convert to Assembly



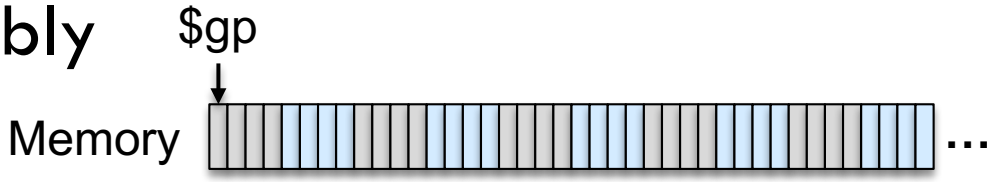

```
int  x, array[100];


main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}


void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

$s0

```
main:
        addi    $t0, $zero, 5
        sw      $t0, 0($gp)
```

# Example Code

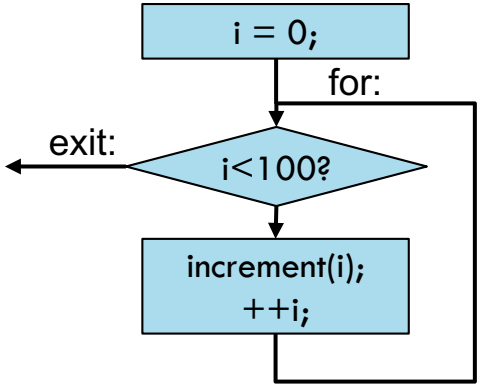
- Convert to Assembly




```
int  x, array[100];

main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}

void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

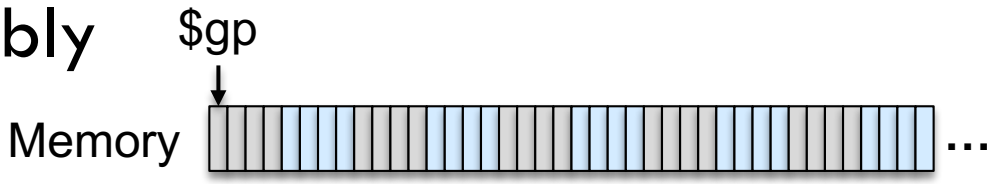
$s0

```
main:
        addi    $t0, $zero, 5
        sw      $t0, 0($gp)
        addi    $s0, $zero, 0
for:    slti    $t1, $s0, 100
        beq     $t1, $zero, exit
        jal     increment
        addi    $s0, $s0, 1
        j       for
exit:
```



# Example Code

- Convert to Assembly




```
int  x, array[100];

main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}

void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

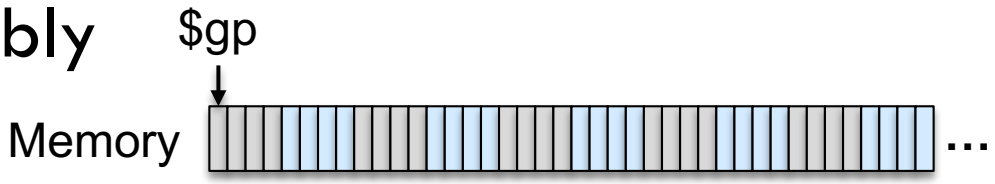
$s0

$s0

```
main:
        addi    $t0, $zero, 5
        sw      $t0, 0($gp)
        addi    $s0, $zero, 0
for:    slti    $t1, $s0, 100
        beq     $t1, $zero, exit
        jal     increment
        addi    $s0, $s0, 1
        j       for
exit:

increment:
        addi    $s0, $zero, 12
```

# Example Code

- Convert to Assembly




```
int  x, array[100];

main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}

void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

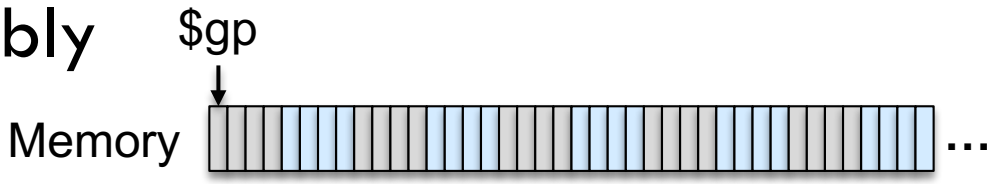
$s0

$s0

```
main:
            addi    $t0, $zero, 5
            sw      $t0, 0($gp)
            addi    $s0, $zero, 0
for:        slti    $t1, $s0, 100
            beq     $t1, $zero, exit
            jal     increment
            addi    $s0, $s0, 1
            j       for
exit:

increment:
            addi    $s0, $zero, 12
            lw      $t0, 0($gp)
            add     $t1, $t0, $s0
            sw      $t1, 4($gp)
            jr      $ra
```

# Example Code

- Convert to Assembly




```
int  x, array[100];

main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}

void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

$s0

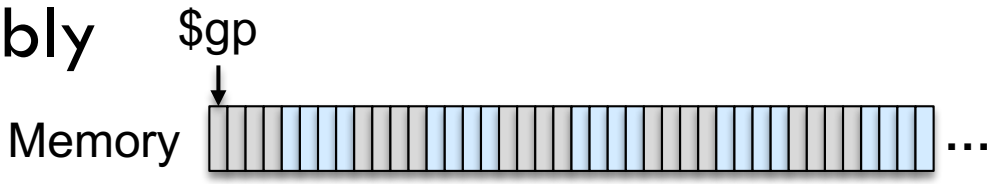
$s0

```
main:
        addi    $t0, $zero, 5
        sw      $t0, 0($gp)
        addi    $s0, $zero, 0
for:    slti    $t1, $s0, 100
        beq     $t1, $zero, exit
        jal     increment
        addi    $s0, $s0, 1
        j       for
exit:

increment:
        addi    $s0, $zero, 12
        lw      $t0, 0($gp)
        add     $t1, $t0, $s0
        sw      $t1, 4($gp)
        jr      $ra
```

What are the issues?

# Example Code

- Convert to Assembly




```
int  x, array[100];

main() {
    int i;
    x = 5;
    for(i=0; i < 100; ++i) {
        increment(i);
    }
}

void increment(int d) {
    int i = 12;
    array[d] = x + i;
}
```

$s0 → i

$s1 → i

$a0 → d

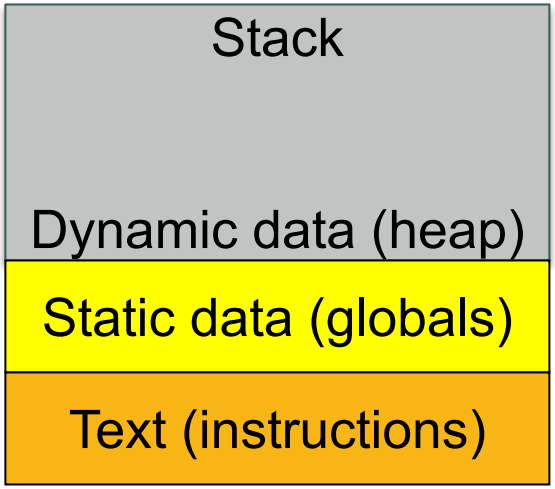
address of array[d] : 4*d + $gp + 4

```
main:
        addi    $t0, $zero, 5
        sw      $t0, 0($gp)
        addi    $s0, $zero, 0
for:    slti    $t1, $s0, 100
        beq     $t1, $zero, exit
        add     $a0, $s0, $zero
        jal     increment
        addi    $s0, $s0, 1
        j       for
exit:
increment:
        addi    $s1, $zero, 12
        lw      $t0, 0($gp)
        add     $t1, $t0, $s1
        sll     $t2, $a0, 2
        add     $t2, $t2, $gp
        sw      $t1, 4($t2)
        jr      $ra
```

# Memory Organization

- Activation record: the space allocated on stack by a procedure including saved values and data local to the procedure
  - frame pointer ($fp) points to the start of the record
  - stack pointer ($sp) points to the end
    - variable addresses are specified relative to $fp as $sp may change during the execution of the procedure
- $gp points to area in memory that saves global variables
- Dynamically allocated storage (with malloc()) is placed on the heap

```
int  x, array[100];

main() {
   int i;
   x = 5;
   for(i=0; i < 100; ++i) {
       increment(i);
   }
}
```




```
void increment(int d) {
   int i = 12;
   array[d] = x + i;
}
```

# Registers

- The 32 MIPS registers are partitioned as follows.
  - Register 0 : $zero always stores the constant 0
  - Register 1 : $at reserved for pseudo instructions
  - Registers 2-3 : $v0, $v1 return values of a procedure
  - Registers 4-7 : $a0-$a3 input arguments to a procedure
  - Registers 8-15 : $t0-$t7 temporaries
  - Registers 16-23: $s0-$s7 variables
  - Registers 24-25: $t8-$t9 more temporaries
  - Registers 28 : $gp global pointer
  - Registers 29 : $sp stack pointer
  - Registers 30 : $fp frame pointer
  - Registers 31 : $ra return address

# Call/Return Memory Management

- Before/after executing the jal, the caller/callee must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into the stack space
- Arguments are copied into $a0-$a3; the jal is executed
- After the callee creates stack space, it updates the value of $sp
- Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented
- On return, the caller/callee brings in stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

# Example: Procedure Call

- int leaf_example (int g, int h, int i, int j) {
  - int f ;
  - f = (g + h) – (i + j);
  - return f;
- }

# Example: Procedure Call

- int leaf_example (int g, int h, int i, int j) {
  - int f ;
  - f = (g + h) – (i + j);
  - return f;
- }

```
g: $a0
h: $a1      f: $s0
i: $a2      temp: $t0, $t1
j: $a3      return: $v0
```

# Example: Procedure Call

- int leaf_example (int g, int h, int i, int j) {
  - int f ;
  - f = (g + h) – (i + j);
  - return f;
- }

```
g: $a0
h: $a1      f: $s0
i: $a2      temp: $t0, $t1
j: $a3      return: $v0
```

```
leaf_example: addi $sp, $sp, -12
              sw $t1, 8($sp)
              sw $t0, 4($sp)
              sw $s0, 0($sp)
              add $t0, $a0, $a1
              add $t1, $a2, $a3
              sub $s0, $t0, $t1
              add $v0, $s0, $zero
              lw $s0, 0($sp)
              lw $t0, 4($sp)
              lw $t1, 8($sp)
              addi $sp, $sp, 12
              jr $ra
```

# Saving Convention

- Caller saved
  - $t0-$t9: the callee won't bother saving these, so save them if you care
  - $ra: it's about to get over-written
  - $a0-$a3: so you can put in new arguments
- Callee saved
  - $s0-$s7: these typically contain "valuable" data