MIPS FUNCTION CALLS

LECTURE 10-2

JIM FIX, REED COLLEGE CS2-S20

FUNCTION CALLS IN MIPS

The MIPS ystem calls hint at a more general mechanism we need, namely...

Q: How do we mimic C++'s function calling mechanism in MIPS?

A: By following the MIPS function calling conventions and stack discipline.

OUTLINE:

- SOME SIMPLE C++ EXAMPLES
- CALL/RETURN WITH JAL/JR ; PARAMETER PASSING
- CREATE/PUSH AND TAKE-DOWN/POP OF STACK FRAME
- EXAMINE CONVENTIONS FOR SAVING REGISTERS' VALUES ON THE FRAME

LEAF CALLS IN C++

- We considered this C++ program:

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.     int lo = two_digits(C,D);
14.     int n = times100(hi) + lo;
15.     cout << n << endl;
16. }
```

LEAF CALLS IN C++

- Let's rename the variables with register names...

```
1.  int two_digits(int a0, int a1) {
2.      return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.      return 100 * a0;
6.  }
7.  int main(void) { int s0, s1, s2, s3;
8.      cin >> s0;
9.      cin >> s1;
10.     cin >> s2;
11.     cin >> s3;
12.     int s0 = two_digits(s0,s1);
13.     int s1 = two_digits(s2,s3);
14.     int v0 = times100(s0) + s1;
15.     cout << v0 << endl;
16. }
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

LEFT: C++

```
1.  int two_digits(int a0,a1) {
2.    return 10 * a0 + a1;
3.  }
4.  int times100(int a0) {
5.    return 100 * a0;
6.  }
7.  int main(void) {
8.    int s0,s1,s2,s3;
9.    cin >> s0;
10.   cin >> s1;
11.   cin >> s2;
12.   cin >> s3;
13.   int s0 = two_digits(s0,s1);
14.   int s1 = two_digits(s2,s3);
15.   int v0 = times100(s0) + s1;
16.   cout << v0 << endl;
17. }
```

RIGHT: MIPS FOR MAIN

call sites

```
main:
... # syscalls to get s0-s3
    ...
    move $a0,$s0
    move $a1,$s1
    jal  two_digits
    move $s0,$v0

    move $a0,$s2
    move $a1,$s3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    add  $a0,$v0,$s1
    ...
... # syscall to output $a0
```

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.
  - We use the **JAL** instruction to "*jump and link*" to a **labelled code line**.

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.
  - We use the **JAL** instruction to "*jump and link*" to a **labelled code line**.
  - This saves the **line after the jump** into a register named **ra**.

CALLING A FUNCTION IN MIPS

- To mimic the C++ call `int s0 = two_digits(s0,s1);`
- ...we write this MIPS code:

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

the "return address" for this call site is here

- NOTES:
  - We pass parameters using the argument registers **a0-a3**
  - We extract the return value from register **v0**.
  - We use the **`JAL`** instruction to "*jump and link*" to a **labelled code line**.
  - This saves the **line after the jump** into a register named **ra**.

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

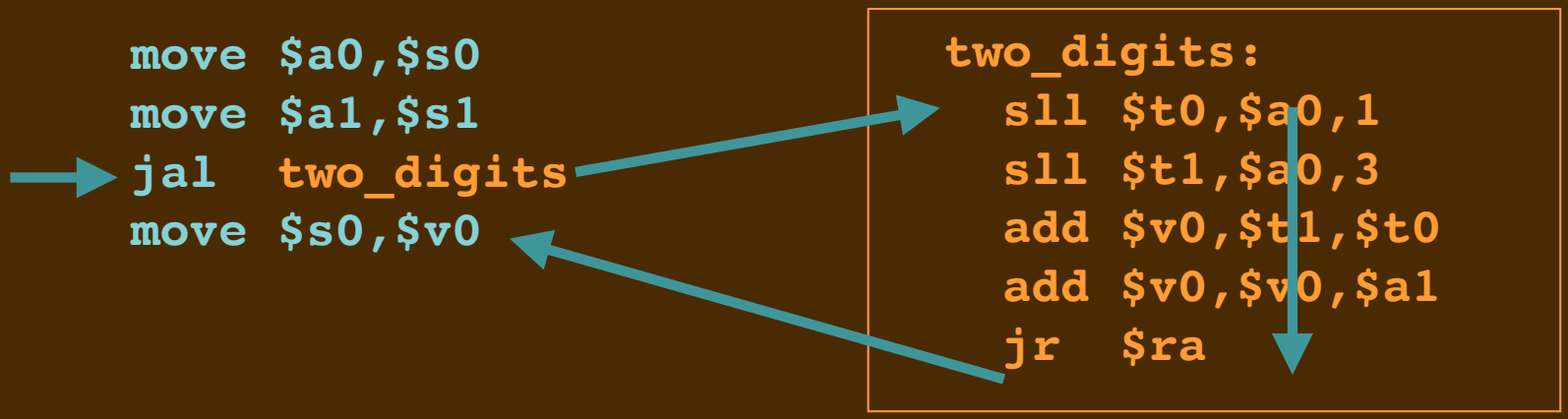
```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:

CALLING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.



- NOTES:
  - These steps are the "jump and link" followed by the "jump back" (return).

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - It grabs its two parameters from **a0** and **a1**.

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - It grabs its two parameters from **a0** and **a1**.
  - It computes its result and puts it into **v0**.

~~CALLING A FUNCTION IN MIPS~~ WRITING A FUNCTION IN MIPS

- The ***callee*** two_digits can assume the ***caller*** followed those conventions.

```
move $a0,$s0
move $a1,$s1
jal  two_digits
move $s0,$v0
```

```
two_digits:
  sll $t0,$a0,1
  sll $t1,$a0,3
  add $v0,$t1,$t0
  add $v0,$v0,$a1
  jr  $ra
```

- NOTES:
  - It grabs its two parameters from **a0** and **a1**.
  - It computes its result and puts it into **v0**.
  - It jumps back to the caller using the address value stored in **ra**.

JUMPING FOR CALL AND RETURN

- There are two "jump" instructions used to call and return from functions:
  - **`JAL`** ***label***
    - ➡This jumps to the callee code at that ***label***.
    - ➡It saves the *return address* into register **$ra**
    - ➡The return address is for the caller's instruction just below the **`JAL`**.
  - **`JR`** **$ra**
    - ➡This jumps from the callee back to the instruction below the call site.
    - ➡The caller then continues executing.

SPECIAL REGISTERS IN MIPS

- There are several conventions for registers in MIPS:
  - Registers **$a0**-**$a3** hold the arguments for the call. (The first 16 bytes.)
  - Registers **$v0** and **$v1** hold the result of the call.
  - Register **$ra** holds the return address of the call.
  - Registers **$fp** and **$fp** mark the top and bottom of the stack frame.

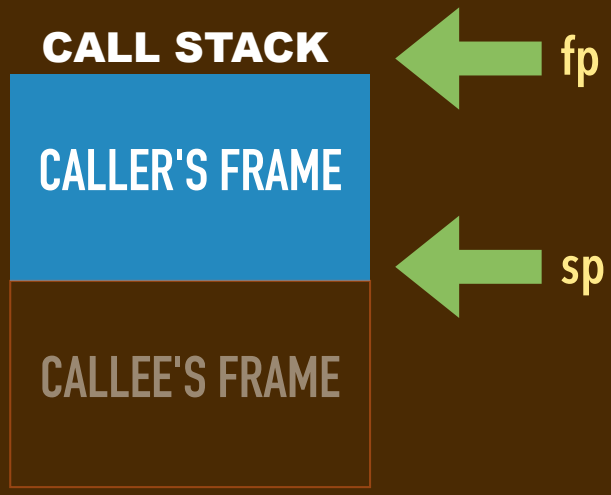
STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

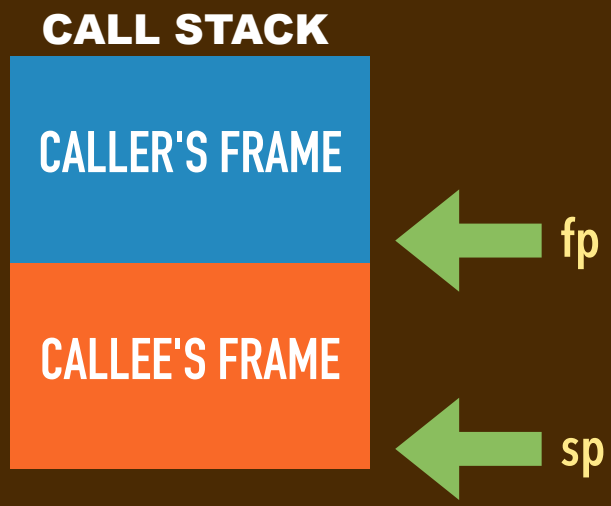
BEFORE THE CALL



STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

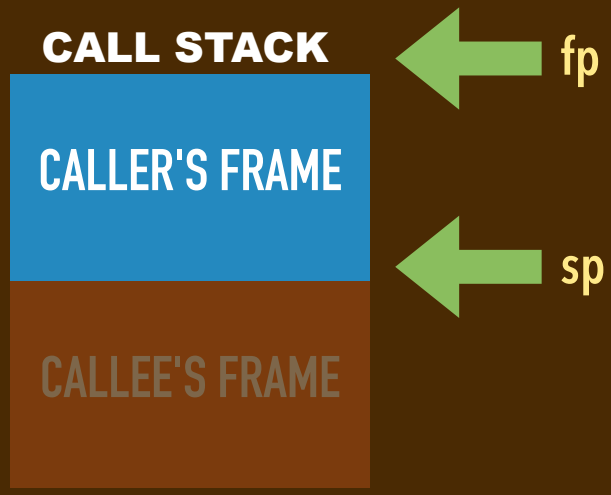
DURING THE CALL



STACK FRAME DISCIPLINE

- The MIPS calling conventions designate that...
  - register **fp** points to the byte just above the top of a function's frame.
  - register **sp** points to the byte just at the bottom of a function's frame
- ...and that the callee ***preserve the caller's frame***.

AFTER THE CALL



STACK FRAME DISCIPLINE (CONT'D)

- The MIPS calling conventions designate that...
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
  - (some say they should be ***double-word aligned*** (multiples of 8)

NON-LEAF FUNCTION

- Let's instead consider this code. The function **four_digits** is not a leaf call.

```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int four_digits(int w,int x,int y,int z) {
8.      return times100(two_digits(w,x)) + two_digits(y,z);
9.  }
10. int main(void) { int A, B, C, D;
11.     cin >> A;
12.     cin >> B;
13.     cin >> C;
14.     cin >> D;
15.     cout << four_digits(A,B,C,D) << endl;
16. }
```

- We're going to work to convert this and the earlier example into MIPS code.

FOUR_DIGITS IN MIPS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

this prologue code sets up a stack frame

this epilogue code takes down the stack frame

FOUR_DIGITS IN MIPS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

this code calls two_digits

FOUR_DIGITS IN MIPS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

saves registers
sets arguments

FOUR_DIGITS IN MIPS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

gets return value

CODE STRUCTURE

- Every call site has a prologue and an epilogue:
  - The caller's prologue saves registers and sets up arguments.
  - Its epilogue gets the return value and restores saved registers.
- Every function's code has a prologue and an epilogue:
  - The callee's prologue sets up its frame, saves registers, grabs arguments.
  - Its epilogue restores registers, takes down the frame, sets the return value.

CODE STRUCTURE

- Every call site has a prologue and an epilogue:
  - The caller's prologue saves registers and sets up arguments.
  - Its epilogue gets the return value and restores saved registers.
- Every function's code has a prologue and an epilogue:
  - The callee's prologue sets up its frame, saves registers, grabs arguments.
  - Its epilogue restores registers, takes down the frame, sets the return value.

CODE STRUCTURE

- Every call site has a prologue and an epilogue:
  - The caller's prologue saves registers and sets up arguments.
  - Its epilogue gets the return value and restores saved registers.
- Every function's code has a prologue and an epilogue:
  - The callee's prologue sets up its frame, saves registers, grabs arguments.
  - Its epilogue restores registers, takes down the frame, sets the return value.

CALLEE-SAVED REGISTERS

- The MIPS calling conventions designate that...
  - Registers need to be preserved with a function call. No clobbering!
- Some registers are "***callee-saved***"
  - ➡ The function called must save the values of these registers on the stack before using them.
  - ➡ It must restore their values from the stack before it retuns to the caller.
  - These registers' values are guaranteed to be preserved with a function call.

CALLER-SAVED REGISTERS

- The MIPS calling conventions designate that...
  - Registers need to be preserved with a function call. No clobbering!
- Some registers are "***caller-saved***"
  - ➡ The caller saves these on the stack before calling a function.
  - ➡ The caller restores them from the stack after the call.
  - These registers' values may not be preserved with a function call.

MIPS CALLING CONVENTIONS SUMMARY: THE CALLER

- PROLOGUE: Before the caller calls a function...
  - It saves caller-saved registers (a0-a3, t0-t9) onto its stack frame.
  - It places the parameters into registers a0-a3.
  - It pushes 5th, 6th, etc parameters onto the bottom of its stack frame.
- Using **`JAL`** saves a return address to register ra.
- EPILOGUE: After the function is called...
  - The caller restores registers it has saved, if needed.
  - It extracts the return value from register v0 and v1.

MIPS CALLING CONVENTIONS SUMMARY: THE CALLEE

- PROLOGUE: When a function is called...
  - It saves callee-saved registers (fp, sp, ra, s0-s7) onto its stack frame.
  - It extracts argument registers a0-a3 and from slots just above its frame.
  - It normally sets fp to the old sp, subtracts an offset from sp.
    - ➡ The offset it chooses is the callee's frame size. It has to be a multiple of 8.
- EPILOGUE: Before a function returns...
  - It puts the return value into register v0 and v1.
  - It restores registers for the caller, including fp, sp, and ra.
- It then performs `JR $RA` to return control back to the caller.

FOUR_DIGITS IN MIPS

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-20($fp)
    sw   $a3,-24($fp)
    jal  two_digits
    move $t0,$v0
    sw   $t0,-12($fp)
    lw   $a0,-20($fp)
    lw   $a1,-24($fp)
    jal  two_digits
    move $t1,$v0
    sw   $t1,-16($fp)
    lw   $t0,-12($fp)
    move $a0,$t0
    jal  times100
    lw   $t1,-16($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

FOUR_DIGITS IN MIPS WITH SOME CLEAN-UP

```
four_digits:
    sw   $ra,-4($sp)
    sw   $fp,-8($sp)
    move $fp,$sp
    addi $sp,$sp,-32
    sw   $a2,-16($fp)
    sw   $a3,-20($fp)
    jal  two_digits
    sw   $v0,-12($fp)
    lw   $a0,-16($fp)
    lw   $a1,-20($fp)
    jal  two_digits
    lw   $a0,-12($fp)
    sw   $v0,-12($fp)
    jal  times100
    lw   $t1,-12($fp)
    add  $v0,$v0,$t1
    addi $sp,$sp,32
    lw   $fp,-8($sp)
    lw   $ra,-4($sp)
    jr   $ra
```

RESOURCES

- Check out https://godbolt.org/
  - Lets you edit a program and inspect `gcc` output with target MIPS32.
  - Note: uses Intel assembly rather than AT&T assembly.
  - Note: uses different register names
    - ➡ $2, $3 are $v0, $v1; $4-$7 are $a0-$a3; $29-$31 are $sp,$fp,$ra
    - ➡ $8-$15,$24,$25 are $t0-$t9; $16-$23 are $s0-$s7; $0 is $zero
- Seems to have a few different calling conventions.
  - Mine are culled from a few sources:
    - ➡ https://courses.cs.washington.edu/courses/cse410/09sp/examples/MIPSCallingConventionsSummary.pdf
    - ➡ http://ellard.org/dan/www/Courses/cs50-asm.pdf

RESOURCES

- You might also take a look at Intel x86 assembly
  - This command generates a **`program.s`**

```
g++ -S program.cc
```