LOGISTICS

- I've started using **Slack** to handle questions/discussion for this course
  - I've invited you to join the "Reed Computer Science" group
  - I've added you to a csci221-s20-discuss channel.
  - ***Try it out!***

- I'd like to develop and publish a schedule of **Zoom** office hours
  - I emailed you a **WhenIsGood** poll seeking the best time slots.
  - ***Please*** follow the link on that email! (Now is a fine time!)

FUNCTIONS IN MIPS

LECTURE 08–3

JIM FIX, REED COLLEGE CS2–S20

TODAY'S PLAN

System 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
- DIGRESSION: MULTIPLICATION WITH BIT OPS
- 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

RECALL: FUNCTIONS IN C++

- Consider 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. }
```

RECALL: FUNCTIONS IN C++

- Consider 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. }
```

RECALL: FUNCTIONS IN C++

- Consider this C++ program:

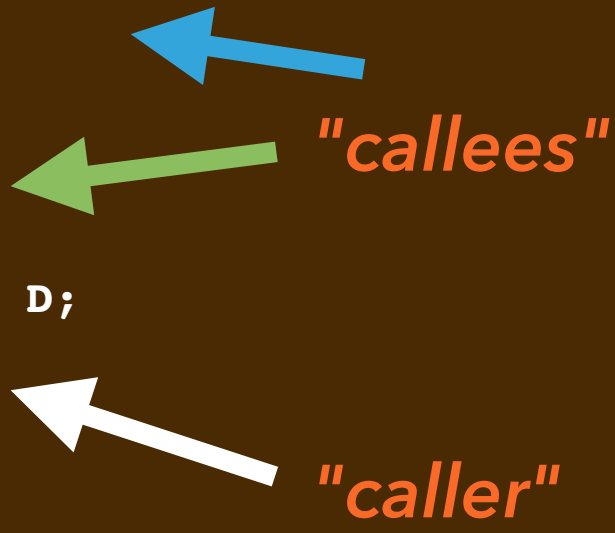
```
1.  int two_digits(int tens, int ones) {
2.      return 10 * tens + ones;
3.  }
4.  int times100(int number) {
5.      return 100 * number;
6.  }
7.  int main(void) { int A, B, C, D;
8.      cin >> A;
9.      cin >> B;
10.     cin >> C;
11.     cin >> D;
12.     int hi = two_digits(A,B);
13.    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

"caller"

RECALL: FUNCTIONS IN C++

- Consider 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. }
```



RECALL: FUNCTIONS IN C++

- Consider 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. }
```

This program takes in four digits as input, and then computes and outputs a four digit integer with those digits.

RECALL: FUNCTIONS IN C++

- Consider 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. }
```

CONSOLE

```
1 5
2 1
3 3
4 7
5
```

This program takes in four digits as input, and then computes and outputs a four digit integer with those digits.

RECALL: FUNCTIONS IN C++

- Consider 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. }
```

CONSOLE

```
1 5
2 1
3 3
4 7
5 5137
```

This program takes in four digits as input, and then computes and outputs a four digit integer with those digits.

RECALL: FUNCTIONS IN C++

- Here is a different version:

```
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. }
```

RECALL: FUNCTIONS IN C++

- Here is a different version:

```
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.

FIRST, A DIGRESSION...

- My plan is to talk about call/return and the call stack.

- Before we begin that, consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?

FIRST, A DIGRESSION...

- My plan is to talk about call/return and the call stack.

- Before we begin that, consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition.

FIRST, A DIGRESSION...

- My plan is to talk about call/return and the call stack.

- Before we begin that, consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*

FIRST, A DIGRESSION...

- My plan is to talk about call/return and the call stack.

- Before we begin that, consider these two expressions

```
return 10 * tens + ones;

return 100 * number;
```

- Q: How do we perform those multiplications in MIPS?
- A1: Repeated addition. *Not how multiplication is perfomed. Too slow.*
- A2: Use the MIPS **`MULT`** instruction, along with **`MFLO`** and **`MFHI`**

ANSWER 3: USE BITS SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL**: multiplying by two will shift the bits of a number left:

```
   111    <= binary for the value 7
  1110    <= binary for the value 2*7=14
111000    <= binary for the value 8*7=56
```

ANSWER 3: USE BITS SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL**: multiplying by two will shift the bits of a number left:

```
111     <= binary for the value 7
1110    <= binary for the value 2*7=14
111000  <= binary for the value 8*7=56
```

- **Q**: So how might we multiply by 10?

ANSWER 3: USE BITS SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
111      <= binary for the value 7
1110     <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10x = (2+8)x = 2x + 8x$

ANSWER 3: USE BITS SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
111      <= binary for the value 7
1110     <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$
- **A:** We can multiply by 2, then by 8, and sum the two results.
- I.E...

ANSWER 3: USE BITS SHIFTING OPERATIONS

- Using built-in multiplication is fine, but there is another way, too.
- **RECALL:** multiplying by two will shift the bits of a number left:

```
111      <= binary for the value 7
1110     <= binary for the value 2*7=14
111000   <= binary for the value 8*7=56
```

- **Q:** So how might we multiply by 10?
- NOTE: $10\,x = (2+8)\,x = 2\,x + 8\,x$
- **A:** We can multiply by 2, then by 8, and sum the two results.
- **I.E.** We can shift right one bit and also shift left three bits. Then add.

ANSWER 3: USE BITS SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3
addu $t0,$t1,$t2
```

ANSWER 3: USE BITS SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3                tmp = tmp * 10
addu $t0,$t1,$t2
```

- It has the effect of multiplying t0 by 10.

ANSWER 3: USE BITS SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3          tmp = tmp * 10
addu $t0,$t1,$t2
```

- It has the effect of multiplying t0 by 10.
- **Q**: So how might we multiply by 100?

ANSWER 3: USE BITS SHIFTING OPERATIONS

- The code below uses the **SLL** instruction to do exactly that with t0:

```
sll  $t1,$t0,1
sll  $t2,$t0,3
addu $t0,$t1,$t2
```

```
tmp = tmp * 10
```

- It has the effect of multiplying t0 by 10.
- **Q**: So how might we multiply by 100?
- NOTE: 100 = 64 + 32 + 4
- **A**: We shift 2, 5, and 6 places left. Add.

MULTIPLICATION BY 100

- The code below multiplies t0 by 100:

```
sll  $t1,$t0,2
sll  $t2,$t0,5
sll  $t3,$t0,6
addu $t0,$t1,$t2
addu $t0,$t0,$t3
```

MULTIPLICATION BY 100

- The code below multiplies t0 by 100:

```
sll  $t1,$t0,2
sll  $t2,$t0,5
sll  $t3,$t0,6
addu $t0,$t1,$t2
addu $t0,$t0,$t3
```

- **EXERCISE 1:** do the above using only t0 and one other register.

MULTIPLICATION BY 100

- The code below multiplies t0 by 100:

```
sll  $t1,$t0,2
sll  $t2,$t0,5
sll  $t3,$t0,6
addu $t0,$t1,$t2
addu $t0,$t0,$t3
```

- **EXERCISE 1:** do the above using only t0 and one other register.
- **EXERCISE 2:** redo HW07 Exercise 1 using shifting and adding
  - Assume that the two values are non-negative and fit into 15 bits.

MULTIPLICATION BY 100

- The code below multiplies t0 by 100:

```
sll  $t1,$t0,2
sll  $t2,$t0,5
sll  $t3,$t0,6
addu $t0,$t1,$t2
addu $t0,$t0,$t3
```

- **EXERCISE 1:** do the above using only t0 and one other register.
- **EXERCISE 2:** redo HW07 Exercise 1 using shifting and adding
  - Assume that the two values are non-negative and fit into 15 bits.
  - Use the bitwise **`AND`** instruction to inspect a bit of one multiplicand.

BACK TO OUR EXAMPLE C++

```
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) {
8.      int A, B, C, D;
9.      cin >> A;
10.     cin >> B;
11.     cin >> C;
12.     cin >> D;
13.     int hi = two_digits(A,B);
14.     int lo = two_digits(C,D);
15.     int n = times100(hi) + lo;
16.     cout << n << endl;
17. }
```

BACK TO OUR EXAMPLE C++. NAME VARIABLES AS REGISTERS

```
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. }
```

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:
... # put inputs into 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
    addu $a0,$v0,$s1
... # 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

```
main:
... # put inputs into 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
    addu $a0,$v0,$s1
... # 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

```
main:
... # put inputs into 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
    addu $a0,$v0,$s1
... # 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

```
main:
... # put inputs into 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
    addu $a0,$v0,$s1
... # 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

```
main:
... # put inputs into 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
    addu $a0,$v0,$s1
... # 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:
... # put inputs into 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
    addu $a0,$v0,$s1
... # 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

- 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
  addu $v0,$t1,$t0
  addu $v0,$v0,$a1
  jr   $ra
```

- NOTES:

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
    addu $v0,$t1,$t0
    addu $v0,$v0,$a1
    jr   $ra
```

- NOTES:

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
    addu $v0,$t1,$t0
    addu $v0,$v0,$a1
    jr   $ra
```

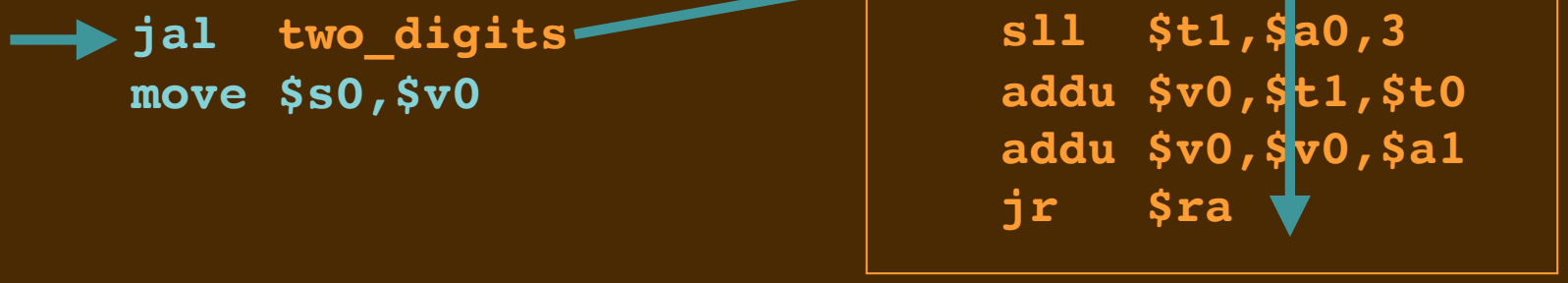
- NOTES:

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
```

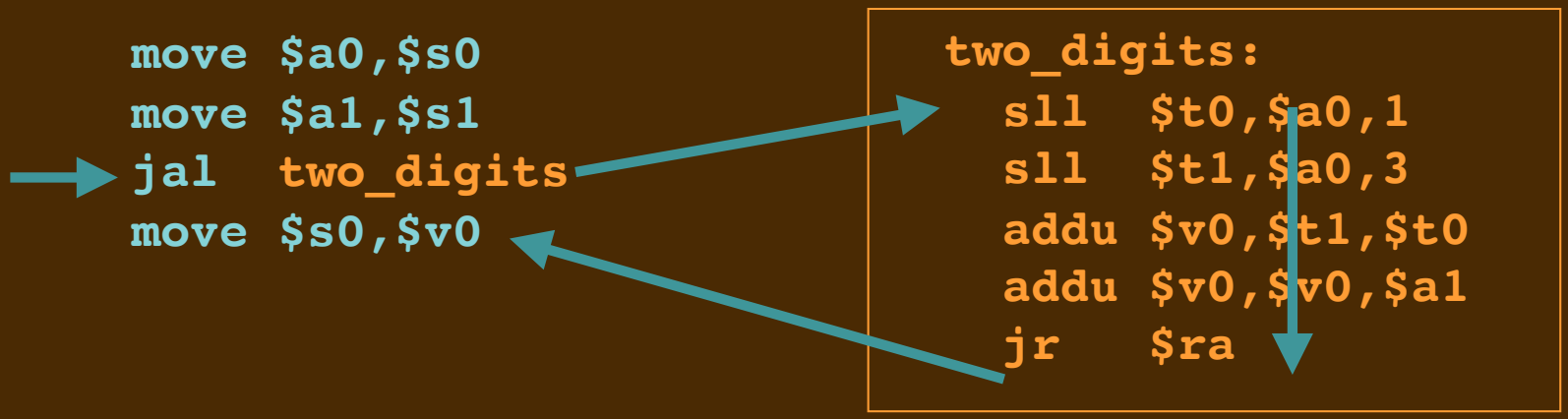
```
two_digits:
    sll  $t0,$a0,1
    sll  $t1,$a0,3
    addu $v0,$t1,$t0
    addu $v0,$v0,$a1
    jr   $ra
```



- NOTES:

CALLING A FUNCTION IN MIPS

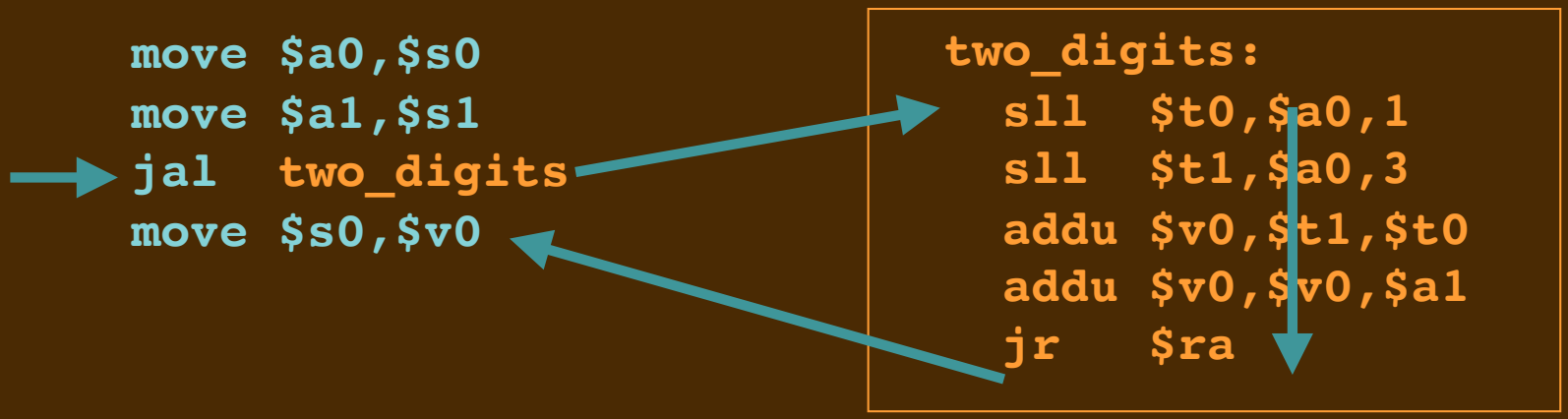
- The *callee* two_digits can assume the *caller* followed those conventions.



- 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
  addu $v0,$t1,$t0
  addu $v0,$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
  addu $v0,$t1,$t0
  addu $v0,$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
  addu $v0,$t1,$t0
  addu $v0,$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
  addu $v0,$t1,$t0
  addu $v0,$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**.

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used **t0**-**t3** in main, rather than **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
```

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

- ...

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used **t0**-**t3** in main, rather than **s0**-**s3**...:

```
    move $a0,$t0
    move $a1,$t1
    jal  two_digits
    move $t0,$v0

    move $a0,$t2
    move $a1,$t3
    jal  two_digits
    move $t1,$v0
...
    move $a0,$t0
    jal  times100
```

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

- ...

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used t0-t3 in main, rather than s0-s3...:

```
    move $a0,$t0
    move $a1,$t1
    jal  two_digits
    move $t0,$v0

    move $a0,$t2
    move $a1,$t3
    jal  two_digits
    move $t1,$v0
...
    move $a0,$t0
    jal  times100
```

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

THE PROBLEMS WITH REGISTER CHOICE

- Suppose we had instead used t0-t3 in main, rather than s0-s3...:

```
    move $a0,$t0
    move $a1,$t1
    jal  two_digits
    move $t0,$v0

    move $a0,$t2
    move $a1,$t3
    jal  two_digits
    move $t1,$v0
...
    move $a0,$t0
    jal  times100
```

@!$%$

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

!!!!!

- ...then the second call to two_digits would "**clobber**" the value held in **t0**.

THE PROBLEMS WITH REGISTER CHOICE

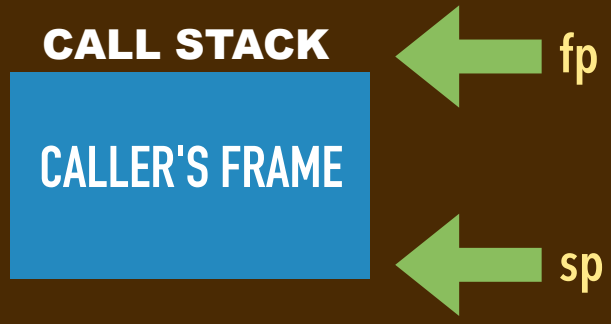
- We only have a fixed number of registers to work with.
  - ➡ Store some function variables in memory. Use ***function frames***.
- As we know, C++ allows us to have an arbitrarily deep number of calls.
  - Because of recursion, a function could have several outstanding calls.
  - Because of recursion, we cannot predict the depth of the calls.
    - ➡ We manage a ***call stack*** of function frames.

THE PROBLEMS WITH REGISTER CHOICE

- We only have a fixed number of registers to work with.
  - ➡ Store some function variables in memory. Use ***function frames***.
- As we know, C++ allows us to have an arbitrarily deep number of calls.
  - Because of recursion, a function could have several outstanding calls.
  - Because of recursion, we cannot predict the depth of the calls.
    - ➡ Manage a ***call stack*** of function frames.
- We still have the problem of avoiding register conflicts.
  - ➡ Adopt conventions to guide us in ***saving/restoring*** registers with calls.

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
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
- ...and that the callee ***preserve the caller's frame***.

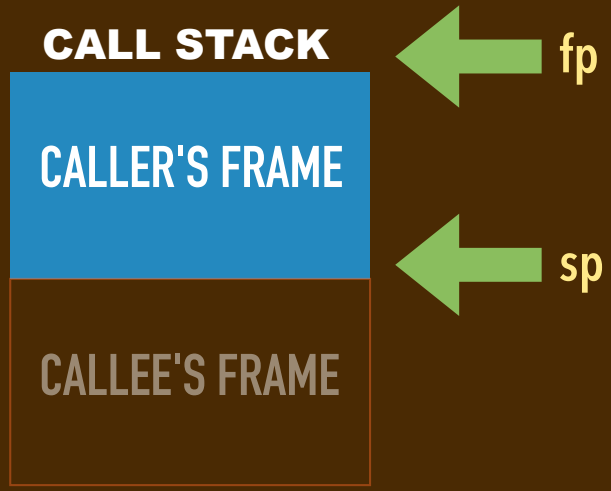


BEFORE THE CALL

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
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
- ...and that the callee ***preserve the caller's frame***.

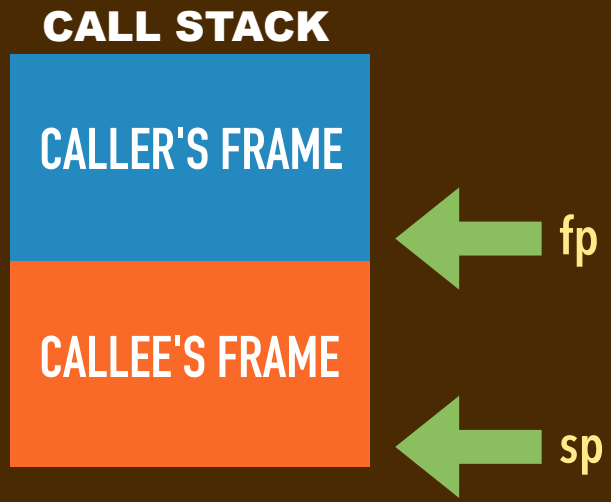
BEFORE THE CALL



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
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
- ...and that the callee ***preserve the caller's frame***.

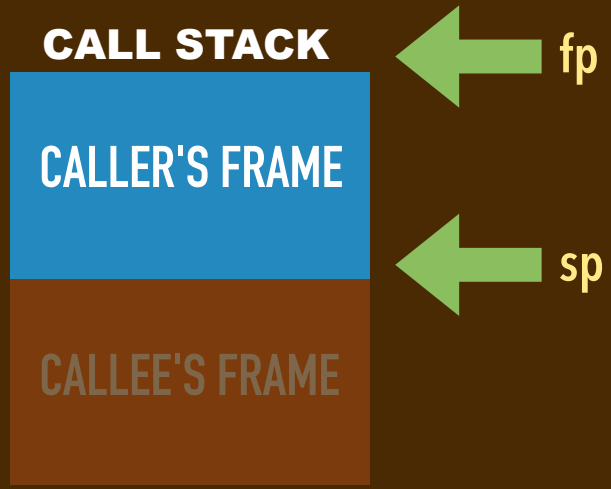
DURING THE CALL



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
  - the frame size should be at least 32 bytes
  - the addresses in fp and sp should be ***word-aligned*** (multiples of 4).
- ...and that the callee ***preserve the caller's frame***.

AFTER THE CALL



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.

FOUR_DIGITS IN C++

```
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. }
```

FOUR_DIGITS IN MIPS (VERSION 1)

```
four_digits:
    move $s0,$a0
    move $s1,$a1
    move $s2,$a2
    move $s3,$a3

    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
    addu $v0,$v0,$s1

    jr $ra
```

FOUR_DIGITS IN MIPS (VERSION 2) USING ARGUMENTS DIRECTLY

```
four_digits:
    # Forward $a0 and $a1 to the call to two_digits.
    jal  two_digits
    move $s0,$v0

    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0

    move $a0,$s0
    jal  times100
    addu $v0,$v0,$s1

    jr $ra
```

FOUR_DIGITS IN MIPS (VERSION 3) USING STACK FRAME

```
four_digits:
    sw    $ra,-4($sp) # Save $ra so we can make calls too.
    sw    $fp,-8($sp) # Save caller's frame pointer.
    move  $fp,$sp     # Set up our frame pointer (frame top).
    addiu $sp,$sp,-32 # Set up the bottom of our frame.
    jal  two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    addu $v0,$v0,$s1  # Set up return value.
    addiu $sp,$sp,32  # Restore caller's frame bottom.
    lw    $fp,-8($sp) # Restore caller's frame pointer.
    lw    $ra,-4($sp) # Restore caller's return address.
    jr    $ra         # Return.
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp) # By convention, S registers must be preserved
    sw    $s1,-16($sp) # They are "callee-saved" registers.
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal   two_digits
    move $s1,$v0
    move $a0,$s0
    jal   times100
    addu $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw    $s0,-12($sp) #
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

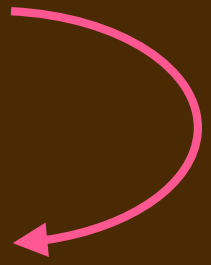
```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp) # By convention, S registers must be preserved
    sw    $s1,-16($sp) # They are "callee-saved" registers.
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move $s0,$v0       # This is because we use them down here.
    move $a0,$a2
    move $a1,$a3
    jal   two_digits
    move $s1,$v0
    move $a0,$s0
    jal   times100
    addu $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw    $s0,-12($sp) #
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp) # By convention, S registers must be preserved
    sw    $s1,-16($sp) # They are "callee-saved" registers.
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move $s0,$v0       # This is because we use S down here.
    move $a0,$a2
    move $a1,$a3
    jal   two_digits
    move $s1,$v0       # and here
    move $a0,$s0       # and here
    jal   times100
    addu $v0,$v0,$s1   # and here.
    addiu $sp,$sp,32
    lw    $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw    $s0,-12($sp) #
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp) # By convention, S registers must be preserved
    sw    $s1,-16($sp) # They are "callee-saved" registers.
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move  $s0,$v0
    move  $a0,$a2
    move  $a1,$a3
    jal   two_digits
    move  $s1,$v0
    move  $a0,$s0
    jal   times100
    addu  $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw    $s0,-12($sp) #
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```



- *Even better, we can assume two_digits follows this same convention.*

➡*It saves/restores S registers too.*

FOUR_DIGITS IN MIPS (VERSION 4) SAVING S REGISTERS

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp) # By convention, S registers must be preserved
    sw    $s1,-16($sp) # They are "callee-saved" registers.
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move $s0,$v0
    move $a0,$a2
    move $a1,$a3
    jal   two_digits
    move $s1,$v0
    move $a0,$s0
    jal   times100
    addu $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp) # Restore $s0 and $s1 for the caller.
    lw    $s0,-12($sp) #
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```


- *As does times100.*
- *As the callee, it preserves S registers.*

FOUR_DIGITS IN MIPS (VERSION 4) PROBLEM!

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp)
    sw    $s1,-16($sp)
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move  $s0,$v0
    move  $a0,$a2
    move  $a1,$a3
    jal   two_digits
    move  $s1,$v0
    move  $a0,$s0
    jal   times100
    addu  $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp)
    lw    $s0,-12($sp)
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

@!$%????

- Unfortunately we cannot assume that a2 and a3 are preserved by two_digits.

FOUR_DIGITS IN MIPS (VERSION 4) PROBLEM!

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp)
    sw    $s1,-16($sp)
    move  $fp,$sp
    addiu $sp,$sp,-32
    jal   two_digits
    move  $s0,$v0
    move  $a0,$a2
    move  $a1,$a3
    jal   two_digits
    move  $s1,$v0
    move  $a0,$s0
    jal   times100
    addu  $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp)
    lw    $s0,-12($sp)
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

@?$%!!!!

- Unfortunately we cannot assume that a2 and a3 are preserved by two_digits.
- We have to assume they're clobbered.

FOUR_DIGITS IN MIPS (VERSION 5) SAVING A REGISTERS

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    sw    $s0,-12($sp)
    sw    $s1,-16($sp)
    move  $fp,$sp
    addiu $sp,$sp,-32
    sw    $a2,-20($fp) # Save the arguments. They are caller-saved.
    sw    $a3,-24($fp) #
    jal  two_digits
    move $s0,$v0
    lw   $a0,-20($fp)  # Restore them after the call.
    lw   $a1,-24($fp)  #
    jal  two_digits
    move $s1,$v0
    move $a0,$s0
    jal  times100
    addu $v0,$v0,$s1
    addiu $sp,$sp,32
    lw    $s1,-16($sp)
    lw    $s0,-12($sp)
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

FOUR_DIGITS IN MIPS (VERSION 6) USING T REGISTERS

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

FOUR_DIGITS IN MIPS (VERSION 6) USING T REGISTERS

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

@?$%!!!!

@?$%!!!!

- Problem: The MIPS calling conventions designate T registers as "caller-saved."

FOUR_DIGITS IN MIPS (VERSION 6) USING T REGISTERS

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    move  $fp,$sp
    addiu $sp,$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)
    addu $v0,$v0,$t1
    addiu $sp,$sp,32
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

FOUR_DIGITS IN MIPS (VERSION 6) WITH SOME CLEAN-UP

```
four_digits:
    sw    $ra,-4($sp)
    sw    $fp,-8($sp)
    move  $fp,$sp
    addiu $sp,$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)
    addu  $v0,$v0,$t1
    addiu $sp,$sp,32
    lw    $fp,-8($sp)
    lw    $ra,-4($sp)
    jr    $ra
```

MIPS CALLING CONVENTIONS SUMMARY: THE CALLER

- 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.
- After the function is called...
  - The caller restores registers it has saved, if needed.
  - It extracts the return value from register v0.

MIPS CALLING CONVENTIONS SUMMARY: THE CALLEE

- When a function is called...
  - It saves callee-saved registers (fp, sp, ra, s0-s9) onto its stack frame.
  - It extracts argument registers a0-a3 and from slots just above its frame
- Before a function returns...
  - It puts the return value into register v0.
  - It restores registers for the caller, including fp, sp, and ra.
- It then performs `JR $RA` to return control back to the caller.

MIPS CALLING CONVENTIONS SUMMARY

ANY QUESTIONS????