

# Code Generation for Function Calls and Typing Functions

Ludovic Henrio<sup>1</sup>

Master 1, ENS de Lyon et Dpt Info, Lyon1

November 15, 2022



---

<sup>1</sup>Slides borrowed from Matthieu Moy and Laure Gonnord

## 1 Code “generation” for function calls

- Function Calls and Return
- Stack, local variables, parameters
- Local Variables
- Register Saving&Restoring
- Parameter Passing
- RiscV Calling Conventions
- Summary

## 2 Typing functions

## 1 Code “generation” for function calls

- **Function Calls and Return**
- Stack, local variables, parameters
- Local Variables
- Register Saving&Restoring
- Parameter Passing
- RiscV Calling Conventions
- Summary

# Function calls: goal

```
void sub_prog() {  
    /* ... */  
    return; /* back to after call to sub_prog */  
}  
  
void sub_prog2() {  
    /* ... */  
} /* back to after call to sub_prog2 */  
  
int main(void) {  
    sub_prog(); /* jumps to start of sub_prog */  
    sub_prog2(); /* same for sub_prog2 */  
    return 0;  
}
```

# First attempt: jump

```
sub_prog:
    li t1, 42
    j after_sub_prog

sub_prog2:
    li t1, 43
    j after_sub_prog2

.globl main
main:
    j sub_prog # Jump
after_sub_prog:

    j sub_prog2
after_sub_prog2:

    ret # end of main
```

# First attempt: jump

```
sub_prog:
    li t1, 42
    j after_sub_prog

sub_prog2:
    li t1, 43
    j after_sub_prog2

.globl main
main:
    j sub_prog # Jump
after_sub_prog:

    j sub_prog2
after_sub_prog2:

    ret # end of main
```

## Question



What's the problem?

## Second attempt: a register for the return address

```
sub_prog:
    li t0, 42
    jr ra

    .globl main
main:
    ## la = Load address
    la ra, after_sub_prog
    j sub_prog
after_sub_prog:
    ## Second call to the
    ## same sub-program
    la ra, after_sub_prog2
    j sub_prog
after_sub_prog2:

    ret # end of main
```

## Second attempt: a register for the return address

```
sub_prog:
    li t0, 42
    jr ra

    .globl main
main:
    ## la = Load address
    la ra, after_sub_prog
    j sub_prog
after_sub_prog:
    ## Second call to the
    ## same sub-program
    la ra, after_sub_prog2
    j sub_prog
after_sub_prog2:

    ret # end of main
```

### Question



What's the limitation?



## Second attempt: a register for the return address

```
sub_prog:
    li t0, 42
    jr ra

    .globl main
main:
    ## la = Load address
    la ra, after_sub_prog
    j sub_prog
after_sub_prog:
    ## Second call to the
    ## same sub-program
    la ra, after_sub_prog2
    j sub_prog
after_sub_prog2:

    ret # end of main
```

### Question



What about recursive calls?

# Recursive or Nested Calls?

```
nested_sub_prog:
    li t0, 43
    jr ra

sub_prog:
    li t0, 42
    ## Override ra set in main
    la ra, after_sub_prog2
    j nested_sub_prog
after_sub_prog2:
    ## return to main... or not.
    jr ra

.globl main
main:
    la ra, after_sub_prog
    j sub_prog
after_sub_prog:
    ret
```

# Recursive or Nested Calls?

```
nested_sub_prog:
    li t0, 43
    jr ra

sub_prog:
    li t0, 42
    ## Override ra set in main
    la ra, after_sub_prog2
    j nested_sub_prog
after_sub_prog2:
    ## return to main... or not.
    jr ra

    .globl main
main:
    la ra, after_sub_prog
    j sub_prog
after_sub_prog:
    ret
```

## Warning

It's not sufficient, but before solving the problem let's simplify notations...

# Meta-instruction `call` & `ret` in RiscV

- Meta-instruction `call label`:
  - Equivalent to `la ra, next-addr + j label`
  - Loads return address into register `ra`, jumps to `label`
  - Return address = address right after the `call`
- Meta-instruction `ret`:
  - Equivalent to `jalr zero, ra, 0`
  - Jumps to the address contained in `ra`

```
sub_prog:
    li t0, 42
    call nested_sub_prog
    ## return to main?
    ret

.globl main
main:
    call sub_prog
    ret
```

# Meta-instruction `call` & `ret` in RiscV

- Meta-instruction `call label`:
  - Equivalent to `la ra, next-addr + j label`
  - Loads return address into register `ra`, jumps to `label`
  - Return address = address right after the `call`
- Meta-instruction `ret`:
  - Equivalent to `jalr zero, ra, 0`
  - Jumps to the address contained in `ra`

```
sub_prog:
    li t0, 42
    call nested_sub_prog
    ## return to main?
    ret

.globl main
main:
    call sub_prog
    ret
```

## Question



Nice, but does not change our problem!

# Digression

- In some instruction sets (e.g. Intel), `call/ret` save and restore address on stack. Then, it “works magically”.
- RISC-V = Reduced Instruction Set Chip, version V  $\Rightarrow$  we need more work by hand.

# Saving return address on stack

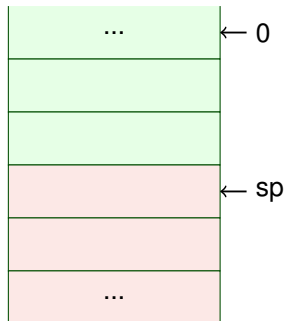
```
    ## Should save/restore too  
nested_sub_prog:  
    li t0, 43  
    ret
```

```
sub_prog:  
    ## push ra on stack  
    addi sp, sp, -8  
    sd ra, 0(sp)  
  
    ## Function's body  
    li t0, 42  
    call nested_sub_prog  
  
    ## pop ra from stack  
    ld ra, 0(sp)  
    addi sp, sp, 8  
    ret
```

```
    .globl main  
main:  
    addi sp, sp, -8  
    sd ra, 0(sp)  
  
    call sub_prog  
  
    ld ra, 0(sp)  
    addi sp, sp, 8  
    ret
```

# Stack during function calls

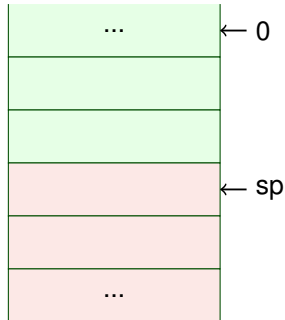
Before call + save ra



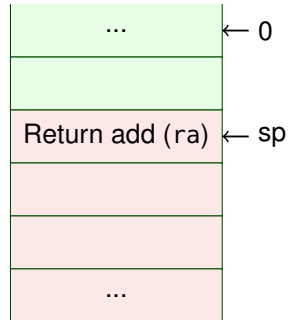


# Stack during function calls

Before call + save ra



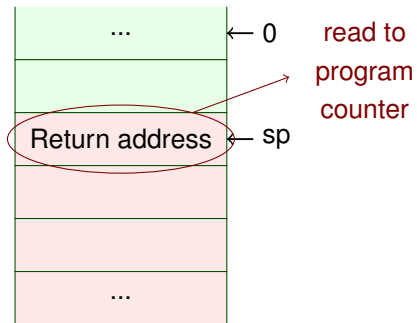
After call + save ra



return address = address following call

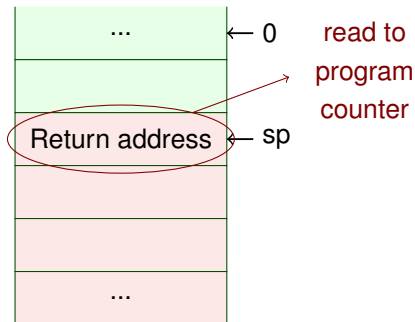
# Stack and ret instruction

Before ret + restore ra

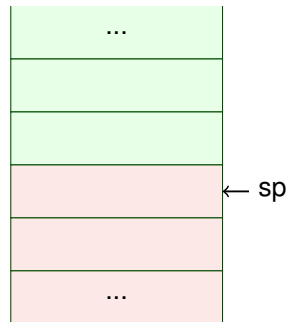


# Stack and ret instruction

Before ret + restore ra



After ret + restore ra ( $\approx$  "pop ra")



# call and ret : Example

```

    ## Should save/restore too
nested_sub_prog:
    li t0, 43
    ret

sub_prog:
    ## push ra on stack
    addi sp, sp, -8
    sd ra, 0(sp)

    ## Function's body
    li t0, 42
    call nested_sub_prog

    ## pop ra from stack
    ld ra, 0(sp)
    addi sp, sp, 8
    ret

```

```

$ spike -d pk ./a.out
: until pc 0 10162 <- sub_prog
: reg 0 sp
0x000000007f7e9b48
: reg 0 ra
0x00000000000010188
: mem 0 000000007f7e9b48
0x00000000000010106
: mem 0 000000007f7e9b40
0x00000000000000000
core 0: 0x00010162 c.addi sp, -8
core 0: 0x00010164 c.sdsp ra, 0(sp)
: reg 0 sp
0x000000007f7e9b40
: reg 0 ra
0x00000000000010188
: mem 0 000000007f7e9b48
0x00000000000010106
: mem 0 000000007f7e9b40
0x00000000000010188
[...]
```

# call and ret : Example

```

## Should save/restore too
nested_sub_prog:
    li t0, 43
    ret

sub_prog:
    ## push ra on stack
    addi sp, sp, -8
    sd ra, 0(sp)

    ## Function's body
    li t0, 42
    call nested_sub_prog

    ## pop ra from stack
    ld ra, 0(sp)
    addi sp, sp, 8
    ret

```

```

$ spike -d pk ./a.out
[...]
: reg 0 ra
0x00000000000010188
[...]
core 0: 0x00010166 li    t0, 42
core 0: 0x0001016a jal   pc -0xe
core 0: 0x0001015c li    t0, 43
core 0: 0x00010160 ret
: reg 0 ra
0x0000000000001016e
core 0: 0x0001016e c.ldsp ra, 0(sp)
core 0: 0x00010170 c.addi sp, 8
: reg 0 ra
0x00000000000010188
core 0: 0x00010172 ret
core 0: 0x00010188 li    a0, 101

```

## 1 Code “generation” for function calls

- Function Calls and Return
- **Stack, local variables, parameters**
- Local Variables
- Register Saving&Restoring
- Parameter Passing
- RiscV Calling Conventions
- Summary

# Local variables: first (failed) attempt without stack

```
int f() {  
    a = ...;  
  
    ... a ...;  
}
```

```
f:  
    mv ..., t0  
  
    ... t0 ...  
    ret
```

```
int g() {  
    b = ...;  
  
    ... b ...;  
}
```

```
g:  
    mv ..., t0  
  
    ... t0 ...  
    ret
```

- What if f calls g?
- What if we have more variables than registers?
- What if we were in real C and have &x operator?
- $\Rightarrow$  Not viable as-is.

# Variables accessible by a function

```
int global;  
int main() {  
    int local;  
    ...  
};
```

- Global variables
- Local variables
- Parameters ( $\approx$  local variables set by caller)



# Variables accessible by a function

```
int global;  
int main() {  
    int local;  
    ...  
};
```

- **Global variables**

⇒ Exist in 1 and only 1 sample. Easy management with labels pointing to static data.

- Local variables

- Parameters ( $\approx$  local variables set by caller)

# Variables accessible by a function

```
int global;  
int main() {  
    int local;  
    ...  
};
```

- Global variables

⇒ Exist in 1 and only 1 sample. Easy management with labels pointing to static data.

- Local variables

- Parameters ( $\approx$  local variables set by caller)

⇒ Only exist when function is being called

## 1 Code “generation” for function calls

- Function Calls and Return
- Stack, local variables, parameters
- **Local Variables**
- Register Saving&Restoring
- Parameter Passing
- RiscV Calling Conventions
- Summary

## Address of Local Variables

Forget about register allocation for now, and assume all variables are stored in memory. Registers are used for temporaries. More on that later.

# Local Variables $\neq$ Global Variables

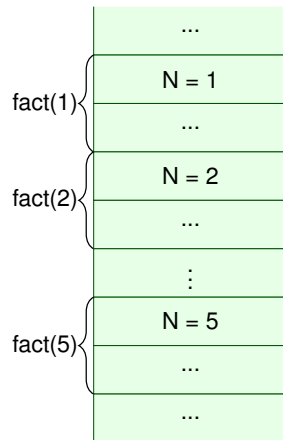
```
int fact(int N) {  
    int res;  
    if (N <= 1) {  
        res = 1;  
    } else {  
        res = N;  
        res = res * fact(N - 1);  
    }  
    return res;  
}
```

- `fact(5)` calls `fact(4)` which calls `fact(3)` ...  $\Rightarrow$  several values for `N` at the same time in memory.

# Local Variables $\neq$ Global Variables

```
int fact(int N) {  
    int res;  
    if (N <= 1) {  
        res = 1;  
    } else {  
        res = N;  
        res = res * fact(N - 1);  
    }  
    return res;  
}
```

- fact(5) calls fact(4) which calls fact(3) ...  $\Rightarrow$  several values for N at the same time in memory.



# Address of Local Variables

- Absolute address:
  - ⇒ impossible, address isn't constant
- Relative to sp:
  - ⇒ Possible<sup>2</sup>, but painful: sp may change too often.
- Solution: address relative to the frame pointer fp
  - fp is set when entering a function
  - ... and restored before return

---

<sup>2</sup>done by gcc -fomit-frame-pointer for example

# Management of sp / fp: Function Calls

```
main: # ...
```

```
    call f
```

```
    # ...
```

```
f:  addi sp, sp, -32
```

```
    sd ra, 0(sp)
```

```
    sd fp, 8(sp)
```

```
    addi fp, sp, 32
```

```
    # Body of f: loc2 = loc1
```

```
    ld t0, -8(fp)
```

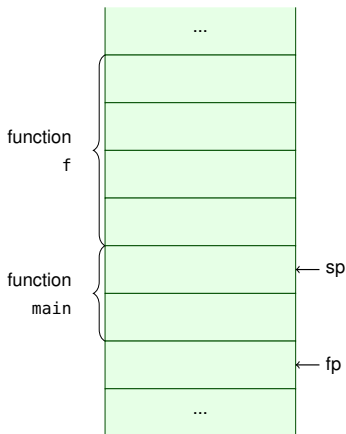
```
    sd t0, -16(fp)
```

```
    ld fp, 8(sp)
```

```
    ld ra, 0(sp)
```

```
    addi sp, sp, 32
```

```
    ret
```





# Management of sp / fp: Function Calls

```
main: # ...
```

```
    call f
```

```
    # ...
```

```
f:  addi sp, sp, -32
```

```
    sd ra, 0(sp)
```

```
    sd fp, 8(sp)
```

```
    addi fp, sp, 32
```

```
    # Body of f: loc2 = loc1
```

```
    ld t0, -8(fp)
```

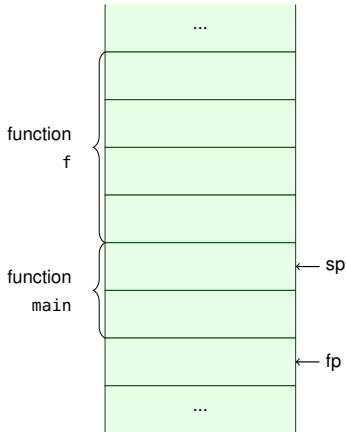
```
    sd t0, -16(fp)
```

```
    ld fp, 8(sp)
```

```
    ld ra, 0(sp)
```

```
    addi sp, sp, 32
```

```
    ret
```



# Management of sp / fp: Function Calls

```
main: # ...
```

```
    call f
```

```
    # ...
```

```
f:  addi sp, sp, -32
```

```
    sd ra, 0(sp)
```

```
    sd fp, 8(sp)
```

```
    addi fp, sp, 32
```

```
    # Body of f: loc2 = loc1
```

```
    ld t0, -8(fp)
```

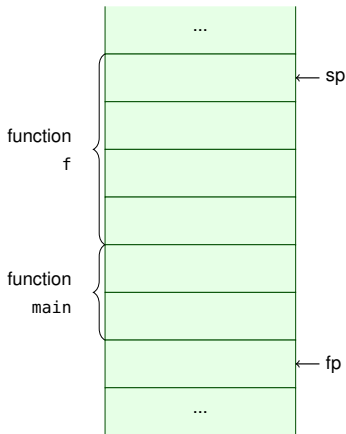
```
    sd t0, -16(fp)
```

```
    ld fp, 8(sp)
```

```
    ld ra, 0(sp)
```

```
    addi sp, sp, 32
```

```
    ret
```



# Management of sp / fp: Function Calls

```
main: # ...
```

```
    call f
```

```
    # ...
```

```
f:  addi sp, sp, -32
```

```
    sd ra, 0(sp)
```

```
    sd fp, 8(sp)
```

```
    addi fp, sp, 32
```

```
    # Body of f: loc2 = loc1
```

```
    ld t0, -8(fp)
```

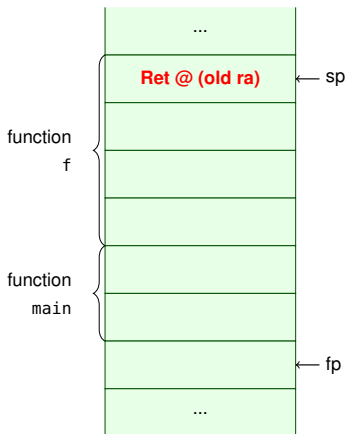
```
    sd t0, -16(fp)
```

```
    ld fp, 8(sp)
```

```
    ld ra, 0(sp)
```

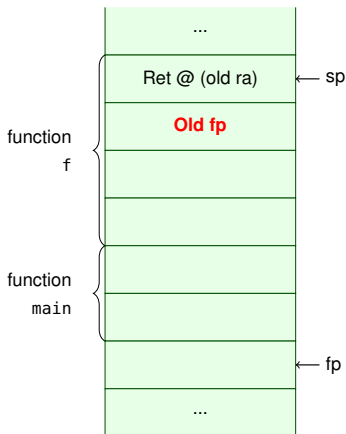
```
    addi sp, sp, 32
```

```
    ret
```



# Management of sp / fp: Function Calls

```
main: # ...  
      call f  
      # ...  
  
f:    addi sp, sp, -32  
      sd ra, 0(sp)  
      sd fp, 8(sp)  
      addi fp, sp, 32  
      # Body of f: loc2 = loc1  
      ld t0, -8(fp)  
      sd t0, -16(fp)  
  
      ld fp, 8(sp)  
      ld ra, 0(sp)  
      addi sp, sp, 32  
      ret
```



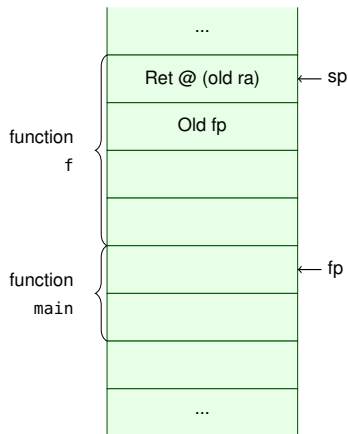
# Management of sp / fp: Function Calls

```

main: # ...
      call f
      # ...

f:    addi sp, sp, -32
      sd ra, 0(sp)
      sd fp, 8(sp)
      addi fp, sp, 32
      # Body of f: loc2 = loc1
      ld t0, -8(fp)
      sd t0, -16(fp)

      ld fp, 8(sp)
      ld ra, 0(sp)
      addi sp, sp, 32
      ret
  
```



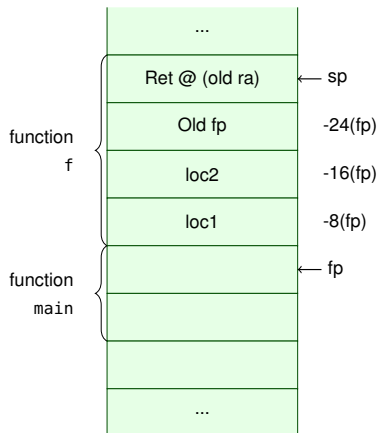
# Management of sp / fp: Function Calls

```

main: # ...
      call f
      # ...

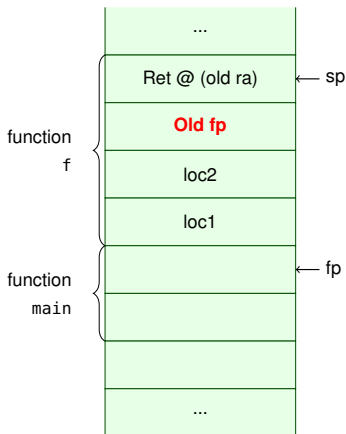
f:    addi sp, sp, -32
      sd ra, 0(sp)
      sd fp, 8(sp)
      addi fp, sp, 32
      # Body of f: loc2 = loc1
      ld t0, -8(fp)
      sd t0, -16(fp)

      ld fp, 8(sp)
      ld ra, 0(sp)
      addi sp, sp, 32
      ret
  
```



# Management of sp / fp: Return from Functions

```
main: # ...  
      call f  
      # ...  
  
f:    addi sp, sp, -32  
      sd ra, 0(sp)  
      sd fp, 8(sp)  
      addi fp, sp, 32  
      # Body of f: loc2 = loc1  
      ld t0, -8(fp)  
      sd t0, -16(fp)  
  
      ld fp, 8(sp)  
      ld ra, 0(sp)  
      addi sp, sp, 32  
      ret
```



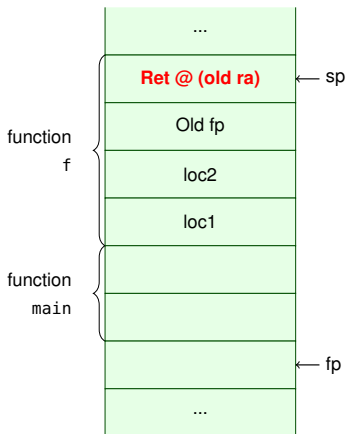
# Management of sp / fp: Return from Functions

```

main: # ...
      call f
      # ...

f:    addi sp, sp, -32
      sd ra, 0(sp)
      sd fp, 8(sp)
      addi fp, sp, 32
      # Body of f: loc2 = loc1
      ld t0, -8(fp)
      sd t0, -16(fp)

      ld fp, 8(sp)
      ld ra, 0(sp)
      addi sp, sp, 32
      ret
  
```





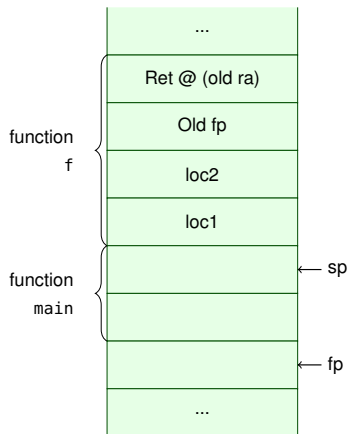
# Management of sp / fp: Return from Functions

```

main: # ...
      call f
      # ...

f:    addi sp, sp, -32
      sd ra, 0(sp)
      sd fp, 8(sp)
      addi fp, sp, 32
      # Body of f: loc2 = loc1
      ld t0, -8(fp)
      sd t0, -16(fp)

      ld fp, 8(sp)
      ld ra, 0(sp)
      addi sp, sp, 32
      ret
  
```



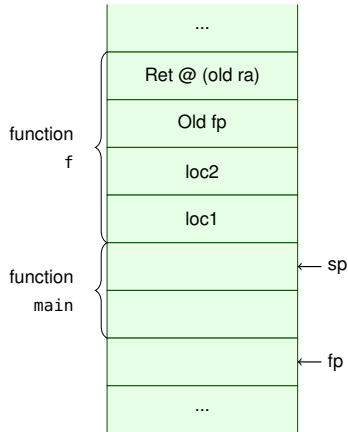
# Management of sp / fp: Return from Functions

```

main: # ...
      call f
      # ...

f:    addi sp, sp, -32
      sd ra, 0(sp)
      sd fp, 8(sp)
      addi fp, sp, 32
      # Body of f: loc2 = loc1
      ld t0, -8(fp)
      sd t0, -16(fp)

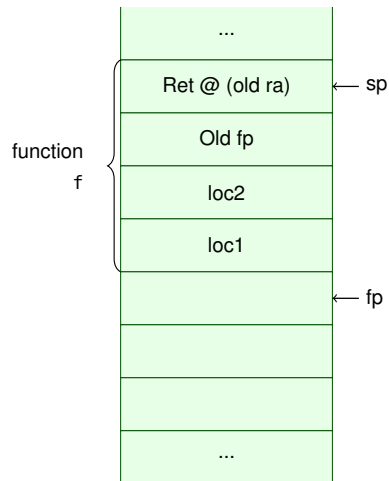
      ld fp, 8(sp)
      ld ra, 0(sp)
      addi sp, sp, 32
      ret
  
```



# Stack Frame

**Stack frame** = set of data accessible by a function

- Created on function call
- “Destroyed” on function return



## 1 Code “generation” for function calls

- Function Calls and Return
- Stack, local variables, parameters
- Local Variables
- **Register Saving&Restoring**
- Parameter Passing
- RiscV Calling Conventions
- Summary

# Register Saving&Restoring

- Problem: Register = global variable in the CPU

```
li t0, 42
call f # may use t0
sd t0, ... # may not be 42 anymore
```

- When to save?
  - By the caller, before the call:  
⇒ Restored by caller, after the call.
  - By the callee, at the beginning of the function:  
⇒ Restored by the callee, before the end of function (ret).

# Register Saving&restoring by the caller

*# Register saving before the call*

```
sd t0, 16(sp)
```

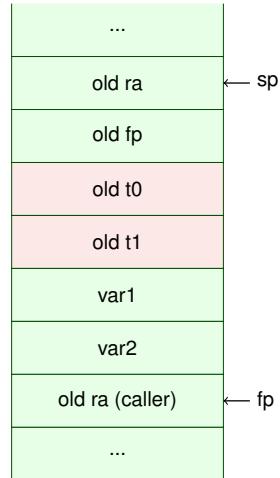
```
sd t1, 24(sp)
```

```
call f
```

*# Restoring, after the call*

```
ld t1, 24(sp)
```

```
ld t0, 16(sp)
```



# Save&Restore by the Callee

```
f: addi sp, sp, -32
   sd ra, 0(sp)
   sd fp, 8(sp)
   addi fp, sp, 32
```

*# Save*

```
sd s1, 16(sp)
sd s2, 24(sp)
```

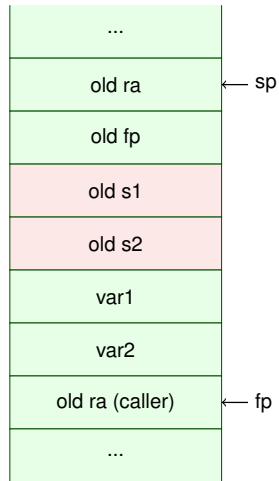
*# Body of f*

*# Restore*

```
ld s2, 24(sp)
ld s2, 16(fp)
```

*# return from f*

```
ld fp, 8(sp)
ld ra, 0(sp)
addi sp, sp, 32
ret
```



# Register Saving Conventions

Caller/Callee saving: what shall we chose?

- Caller and callee need to follow the same convention!
- Convention can be different for different registers:
  - “**scratch**” (aka volatile, aka caller-saved)  $\Rightarrow$  the callee doesn't have to save/restore. The callee needs to save/restore if needed.
  - “**non scratch**” (aka callee-saved)  $\Rightarrow$  The callee must save any register it uses.



# Register Saving Conventions

Caller/Callee saving: what shall we chose?

- Caller and callee need to follow the same convention!
- Convention can be different for different registers:
  - “**scratch**” (aka volatile, aka caller-saved)  $\Rightarrow$  the callee doesn't have to save/restore. The callee needs to save/restore if needed.  
**In RiscV:**  $ra, t_i, i \in 0..6$ .  $t$  = temporary
  - “**non scratch**” (aka callee-saved)  $\Rightarrow$  The callee must save any register it uses.  
**In RiscV:**  $sp, fp, s_i, i \in 1..11$  (remember that  $s0$  is another name for  $fp$ ).  $s$  = saved.

# Save&Restore: Big Picture

*f: # ... (room for at least 160 bytes)*

`sd s1, 16(fp)`

`sd s2, 24(sp)`

*# ...*

`sd s11, 96(sp)`

*# Call function g*

`sd t0, 104(sp)`

`sd t1, 112(sp)`

*# ...*

`sd t6, 152(sp)`

`call g`

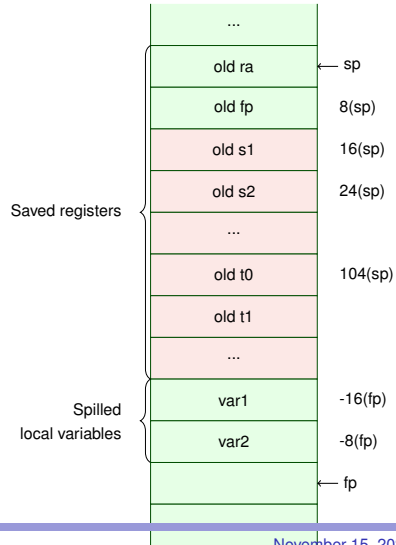
`ld t1, 112(sp)`

`ld t0, 104(sp)`

`ld s2, 24(sp)`

`ld s1, 16(fp)`

*# ...*



# Hmm, Save&Restore Everything, Really?

- Save&Restore everything: each stack-frame = 160 bytes + actual variables (not counting floating-point registers).
- Easy optimization:
  - Save  $s_i$  registers only if used somewhere in the function.
  - Save  $t_i$  registers only if live at the call site.
  - Don't save  $t_i$  when no function is called.
- In the lab: go for brute-force, save everything!

# Local Variables Vs Registers

- Remember register allocation: variables can be (cleverly) allocated to registers, not just in memory...
- ... but we need one instance of variable per stack-frame.

# Local Variables Vs Registers

- Remember register allocation: variables can be (cleverly) allocated to registers, not just in memory...
- ... but we need one instance of variable per stack-frame.
- Registers are saved (by caller or callee)  $\Rightarrow$  virtually one register instance per stack frame  $\Rightarrow$  allocate variables just like before for each function, it works.

# Local Variables Vs Registers

- Remember register allocation: variables can be (cleverly) allocated to registers, not just in memory...
- ... but we need one instance of variable per stack-frame.
- Registers are saved (by caller or callee)  $\Rightarrow$  virtually one register instance per stack frame  $\Rightarrow$  allocate variables just like before for each function, it works.
- In real-life, constructs like C's & (address of) operator may also force allocation in the stack.

## 1 Code “generation” for function calls

- Function Calls and Return
- Stack, local variables, parameters
- Local Variables
- Register Saving&Restoring
- **Parameter Passing**
- RiscV Calling Conventions
- Summary

# Parameter Passing

- Function parameter = local variable assigned from callee
- Same as local variables: in stack or in register
- RiscV: pass parameters using registers  $a_i$  ( $i \in 0..7$ ), use stack only if more than 8 parameters.
- Return value = value in the callee, assigned from caller
- RiscV: return value in  $a0$



# Parameter Passing: Example

```
f: # ...  
  
    # x = g(42, 666)  
    li a0, 42  
    li a1, 666  
    call g  
    ld ``x'', a0  
    # ...  
  
g: # ...  
    # use a0 and a1 to  
    # access parameters.
```

# Parameter Passing + Saving&Restoring

What doesn't not work

```
f: # ...  
  
    # g(h(), i(), j())  
    call h  
    mv a0, a0 # or not  
    call i  
    mv a1, a0 # Oops, call i just destroyed a0  
    call j # may destroy a1 too  
    mv a2, a0  
    call g  
    # ...  
  
g: # ...  
    # use a0 and a1 to  
    # access parameters.
```

# Parameter Passing + Saving&Restoring

*# g(h(), i(), j())*

*# First, Evaluate args:*

call h

mv temp\_0, a0

call i

mv temp\_1, a0

call j

mv temp\_2, a0

*# Then, do pass arguments:*

mv a0, temp\_0

mv a1, temp\_1

mv a2, temp\_2

*# Save registers*

sd t0, 104(sp)

sd t1, 112(sp)

call g

mv temp\_3, a0

# Pass-by-Value, Pass-by-Address

- Pass-by-value: copy the value of the argument to pass it to the callee
  - Original value is unmodified
  - Done for scalar types in C (`void f(int x)`)
- Pass-by-address: give the address of the argument to the callee
  - Modifications done in the callee are visible to the caller
  - Avoids cost of copy for large data
  - Done manually in C (`void f(int *x)`) or by the compiler when using references (`void f(int &x)`)
  - Arrays are always passed by address in C

## Example: increment(int &x) {x++;}

increment:

```
addi sp, sp, -16
sd ra, 0(sp)
sd fp, 8(sp)
addi fp, sp, 16
```

*# Read args to temporaries*

```
mv t0, a0
```

*# Body*

```
lw t1, (t0)
addi t1, t1, 1
sw t1, (t0)
```

```
ld fp, 8(sp)
ld ra, 0(sp)
addi sp, sp, 16
ret
```

main:

```
addi sp, sp, -32
sd ra, 0(sp)
sd fp, 8(sp)
addi fp, sp, 32
sd s1, 16(sp)
```

*# int x = 42; x stored at -8(fp)*

```
li t0, 42
sd t0, -8(fp)
```

*# increment(x)*

```
addi a0, fp, -8 # Address of x
call increment
```

*# return x;*

```
ld a0, -8(fp)
```

## 1 Code “generation” for function calls

- Function Calls and Return
- Stack, local variables, parameters
- Local Variables
- Register Saving&Restoring
- Parameter Passing
- **RiscV Calling Conventions**
- Summary

# Calling Conventions, aka Application Binary Interface

- **Calling convention** (ABI, Application Binary Interface) = conventions imposed for data representation and allocation in memory.
- Needed to get the caller and callee to work together (e.g. caller compiled with GCC, callee compiled with LLVM, will it work?<sup>3</sup>)
- May impose:
  - A number of system calls and how to perform them
  - Memory addresses usable by a program
  - Registers usage convention
  - Stack conventions

---

<sup>3</sup>Same question when mixing GCC and your compiler ...

# RiscV ABI

- Format of a stack frame as described here.
- Stack pointer always multiple of 16 (add 8 bytes of padding if needed)
- Parameters passed in  $a_i$
- Return value passed through  $a_0$
- $t_i, i \in 0..6$  are caller-saved
- $s_i, i \in 1..11$  are callee-saved

See “RISC-V User-Level ISA, Chapter 18, Calling Convention”

<https://riscv.org/wp-content/uploads/2015/01/riscv-calling.pdf>



## 1 Code “generation” for function calls

- Function Calls and Return
- Stack, local variables, parameters
- Local Variables
- Register Saving&Restoring
- Parameter Passing
- RiscV Calling Conventions
- **Summary**

# Code for a Function Body

```
f: addi sp, sp, -XXX # Adjust XXX to size of stack frame
    sd ra, 0(sp)
    sd fp, 8(sp)
    addi fp, sp, XXX # Adjust

# Save s registers
sd s1, 16(sp)
sd s2, 24(sp)
# ...
sd s11, 96(sp)

# Read args to temporaries
mv temp_0, a0
mv temp_1, a1
# ...

# Body of f

# Restore s registers
ld s1, 16(sp)
ld s2, 24(sp)
# ...
ld s11, 96(sp)

# Evaluate return val
# in temp_41
# ...
mv a0, temp_41

ld fp, 8(sp)
ld ra, 0(sp)
addi sp, sp, XXX # Adjust
ret
```

# Code for a Function Call

```
# Evaluate args:
# ... save t registers
call h
# ... restore t registers
mv temp_0, a0
call i # + save and restore registers
mv temp_1, a0
call j # + save and restore registers
mv temp_2, a0

# Do pass arguments
mv a0, temp_0
mv a1, temp_1
mv a2, temp_2
```

```
# Save t registers
sd t0, 104(sp)
sd t1, 112(sp)
# ...
sd t6, 152(sp)
# Actual call
call g

# Restore t registers
ld t0, 104(sp)
ld t1, 112(sp)
# ...
ld t6, 152(sp)
# Get return value
mv temp_3, a0
```

- 1 Code “generation” for function calls
- 2 Typing functions

# Mini-While Syntax 1/2

Expressions:

$$e ::= c \mid e + e \mid e \times e \mid \dots$$

Mini-while:

$S(Smt) ::=$	$x := expr$	assign
	$  \quad x := f(e_1, \dots, e_n)$	simple function call
	$  \quad skip$	do nothing
	$  \quad S_1; S_2$	sequence
	$  \quad \text{if } b \text{ then } S_1 \text{ else } S_2$	test
	$  \quad \text{while } b \text{ do } S \text{ done}$	loop

## Mini-While Syntax 2/2

**[NEW]** Programs with function definitions and global variables

$Prog ::= D \text{ FunDef } Body$	Program
$Body ::= D; S$	Function/main body
$D ::= \text{var } x : \tau   D; D$	Variable declaration
$FunDef ::= \tau \text{ } f(x_1 : \tau_1, \dots, x_n : \tau_n) \text{ } Body; \text{return } e$ $\quad   \text{ FunDef FunDef}$	Function def

Note/discussion: to simplify syntax and semantics:

- 1) function call is not an expression but a special statement.
- 2) return only appears at the end of the function definition (enforced by syntax).

## An example

```
int x
int f(int x, bool b) {
  int y;
  x:=1;
  y:=3;
  if b then x:=x+1 else x:=y;
  return x+1;
}
{
  int y;
  x:=0;
  y:=0;
  y:=f(3, False);
  y:=f(True); // Syntax OK ; we should prevent this by typing
}
```

## OLD Type System (1/2)

From declarations we infer  $\Gamma : Var \rightarrow Basetype$  with the two following rules:

$$\overline{var\ x : t \rightarrow_d [x \mapsto t]}$$

$$\frac{D_1 \rightarrow_d \Gamma_1 \quad D_2 \rightarrow_d \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{D_1; D_2 \rightarrow_d \Gamma_1 \cup \Gamma_2}$$

Then a typing judgment for expressions is  $\Gamma \vdash e : \tau \in Basetype$ . Statements have no type.

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}}$$



# OLD Type System (2/2)

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash S_1; S_2}$$

$$\frac{D \rightarrow \Gamma \quad \Gamma \vdash S}{\emptyset \vdash D; S}$$

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e}$$

$$\frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash S_1 \quad \Gamma \vdash S_2}{\Gamma \vdash \text{if } b \text{ then } S_1 \text{ else } S_2}$$

$$\frac{\Gamma \vdash b : \text{bool} \quad \Gamma \vdash S}{\Gamma \vdash \text{while } b \text{ do } S \text{ done}}$$

## Function table (types)

First we extract the list of function signatures:

$$\frac{}{\tau f(x_1 : \tau_1, \dots, x_n : \tau_n) \rightarrow_f [f \mapsto (\tau_1, \dots, \tau_n \rightarrow \tau)]}$$

$$\frac{Fundef_1 \rightarrow_f \Gamma_1 \quad Fundef_2 \rightarrow_f \Gamma_2 \quad Dom(\Gamma_1) \cap Dom(\Gamma_2) = \emptyset}{Fundef_1 \quad Fundef_2 \rightarrow_f \Gamma_1 \cup \Gamma_2}$$

# Type judgements and typing program

Typing of statements has now the form :  $\Gamma, \Gamma_f \vdash S$

Where  $\Gamma$ : map that defines the variable types,  $\Gamma_f$ : function map,  $S$  statement. To type a program we type all function bodies:

$$\begin{array}{c}
 D \rightarrow_d \Gamma_g \quad \text{Fundef} \rightarrow_f \Gamma_f \quad D_m \rightarrow_d \Gamma_m \\
 \Gamma_g + \Gamma_m, \Gamma_f \vdash S \quad \forall (\tau \ f(x_1 : \tau_1, \dots, x_n : \tau_n) \ D_f; S_f; \text{return } e) \in \text{Fundef}. \\
 \hline
 \Gamma_g + \Gamma_l \vdash e : \tau \wedge \Gamma_g + \Gamma_l, \Gamma_f \vdash S_f \text{ with } x_1 : \tau_1; \dots; x_n : \tau_n; D_f \rightarrow_d \Gamma_l \\
 \hline
 \vdash D \ \text{Fundef} \ D_m; S
 \end{array}$$

$\Gamma_g + \Gamma_l$  overrides  $\Gamma_g$  with  $\Gamma_l$ , i.e.  $(\Gamma_g + \Gamma_l)(x)$  is  $\Gamma_l(x)$  if it is defined and  $\Gamma_g(x)$  else.

# Typing function call

CALL

$$\frac{\Gamma_f(f) = \tau_1, \dots, \tau_n \rightarrow \tau \quad \forall i \in [1..n]. \Gamma \vdash e_i : \tau_i \quad \Gamma \vdash x : \tau}{\Gamma, \Gamma_f \vdash x := f(e_1, \dots, e_n)}$$

Typing rules for other statement are unchanged (except the additional  $\Gamma_f$  parameter)

# An example

Type the following program (or not):

```
int x
int f(int x, bool b) {
  int y;
  x:=1;
  y:=3;
  if b then x:=x+1 else x:=y;
  return x+1;
}
{
  int y;
  x:=0;
  y:=0;
  y:=f(3, False);
  y:=f(True);
}
```

## Typing functions in Mini-C

- Typing of function calls should check that parameters are of the good type but also the number of parameters is good
- The table of function types is not pre-populated (single pass), hence the need to have function declaration.
- There is function definition and function declaration without body: if both are present coherence should be checked (same types for parameter and return).
- Do not forget to check return type.
- Typing MiniC is quite easy, producing meaningful error message is harder!

**Look at typing rule and suggest a few error messages**

# Conclusion

We have seen:

- How a function call and return is encoded in assembler
- With stack manipulation, stack pointer and frame pointer
- A simple type system for functions

Next course will be:

- Function semantics (many different versions)
- Type safety.