# Discussion 5
# RISC-V Calling Convention

Kunchang Guo

# Calling and Returning Functions

```
main:
  # …
  li a0, 0
  call fun
  # …

fun:
  # …
  ret
```

- What does `call fun` mean? (usually translated to `auipc + jalr` [1])

```
ra := pc + 4            # save address as the next instruction to $ra
pc := address of fun    # jump to fun
```

- What does `ret` mean? (translated to `jalr` [1])

```
pc := ra                # jump back
```

[1]: RISC-V User Level ISA

# Calling and Returning Functions

Consider a buggy example [2]:

```
   main:
1    li a0, 10
2    li a1, 11
3    call sum_then_double
4    #...
   sum_then_double:
5    call sum
6    slli a0, a0, 1
7    ret
   sum:
8    add a0, a0, a1
9    ret
```

What's wrong with this code snippet?

[2]: 6.1810 RISC-V Calling Convention

# Calling and Returning Functions

Consider a buggy example [2]:

```
   main:
1    li a0, 10
2    li a1, 11
3    call sum_then_double
4    #...
   sum_then_double:
5    call sum
6    slli a0, a0, 1
7    ret
   sum:
8    add a0, a0, a1
9    ret
```

What's wrong with this code snippet?

Line number pointed by `$ra` and `$pc`:

| | call | | call | ret |
|---|---|---|---|---|
| | main -> | sum_then_double -> | sum -> | sum_then_double … |
| `$ra` | 4 | 6 | 6 | |
| `$pc` | 5 | 8 | 6 | |

`$ra` would have been overwritten between function calls.
When executing line 7 (`ret`), `$pc` will be set to line 6 (`$ra`) again.
-> This program never returns back to `main`!

How can functions interoperate properly?

[2]: 6.1810 RISC-V Calling Convention

# Calling and Returning Functions

Fixing the bug:

```
sum_then_double:
  # save $ra onto the stack
  addi sp, sp, -4
  sw ra, 0(sp)

  call sum
  slli a0, a0, 1

  # restore $ra
  lw ra, 0(sp)
  addi sp, sp, 4

  ret
```

```
sum:
  add a0, a0, a1
  ret

main:
  li a0, 10
  li a1, 11
  call sum_then_double
  #...
```

# Calling and Returning Functions

Fixing the bug:

```
sum_then_double:
  # save $ra onto the stack
  addi sp, sp, -4
  sw ra, 0(sp)

  call sum
  slli a0, a0, 1

  # restore $ra
  lw ra, 0(sp)
  addi sp, sp, 4

  ret
```

```
sum:
  add a0, a0, a1
  ret

main:
  li a0, 10
  li a1, 11
  call sum_then_double
  #...
```

We need a **contract** that governs
- How to transfer function arguments
- How to return values back
- Who saves which registers
- How to manage the stack
- …

# RISC-V Calling Convention Register Usage

| Register | ABI Name | Saver | Description |
|---|---|---|---|
| x0 | zero | - | Hard-wired zero |
| x1 | ra | Caller | Return address |
| x2 | sp | Callee | Stack pointer |
| x3 | gp | - | Global pointer |
| x4 | tp | - | Thread pointer |
| x5-7 | t0-2 | Caller | Temporaries |
| x8 | s0/fp | Callee | Saved register/frame pointer |
| x9 | s1 | Callee | Saved register |
| x10-11 | a0-1 | Caller | Function arguments/return values |
| x12-17 | a2-7 | Caller | Function arguments |
| x18-27 | s2-11 | Callee | Saved registers |
| x28-31 | t3-6 | Caller | Temporaries |

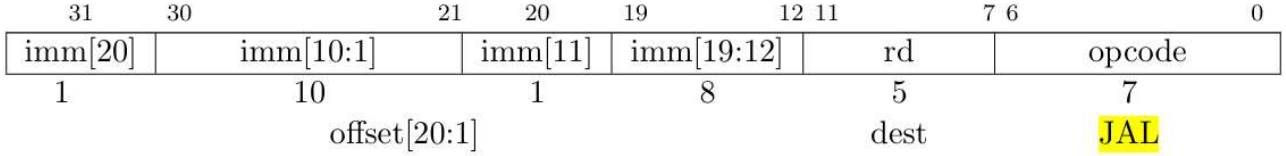
**Who saves what?**

Caller-Saved:
- Temporary (t0-t6)
- Argument/Return (a0-a7,ra)

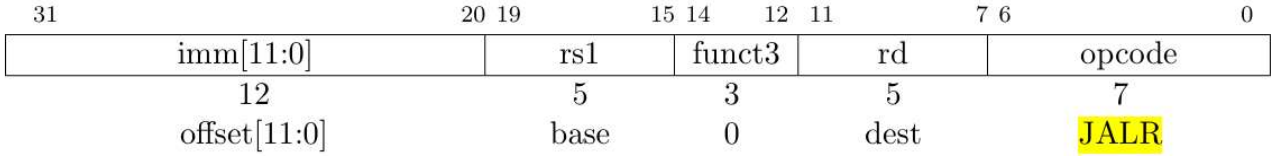
Callee-Saved:
- Saved registers (s0-s11)
- Stack pointer (sp)

# Unconditional Jumps

**JAL** (jump and link):
- `jal rd offset`
  - `rd := pc+4`
  - `pc := pc+sign_ext(offset)`
- imm: 21-bit signed (multiple of 2 bytes)
- targets ranges +/- 1MiB




**JALR** (jump and link register)
- `jalr rd rs offset`
  - `rd := pc+4`
  - `pc := rs+sign_ext(offset)`
    (with least-significant bit set to 0)
- imm: 12-bit signed
- `lui+jalr` / `auipc+jalr` can target 4GiB space



# Unconditional Jumps

**JAL** (jump and link):
- `jal rd offset`
  - `rd := pc+4`
  - `pc := pc+sign_ext(offset)`
- imm: 21-bit signed (multiple of 2 bytes)
- targets ranges +/- 1MiB

**JALR** (jump and link register)
- `jalr rd rs offset`
  - `rd := pc+4`
  - `pc := rs+sign_ext(offset)`
    (with least-significant bit set to 0)
- imm: 12-bit signed
- **`lui+jalr`** / **`auipc+jalr`** can target 4GiB space

Pseudo Instructions [1]:

```
j   offset  – jal  zero, offset
jal offset  – jal  ra,   offset

jr   rs     – jalr zero, rs, 0
jalr rs     – jalr ra,   rs, 0

call offset –
 - auipc t1, %hi(offset)
 - jalr  ra, t1, %lo(offset)

ret         – jalr zero, ra, 0
```

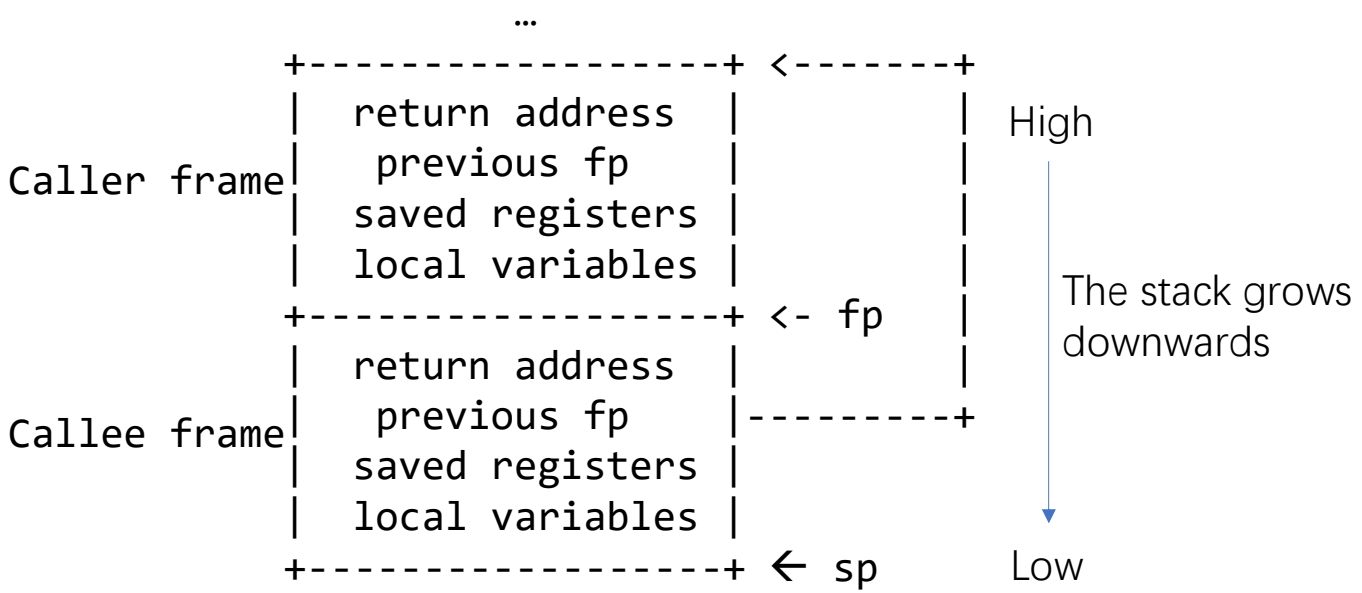
[1]: RISC-V User Level ISA

# The Stack

**Parameter passing**
- a0-a7
- additional parameters on stack

**Register saving**:
- Caller-saved: t0-t6, a0-a7, ra
- Callee-saved: s0-s11, sp

**Alignment**:
- must be 16-byte aligned
- ensures efficient memory access




Frame pointer:
- stable reference point to access stack-allocated variables
- maintains the linked chain of stack frames

# Caller & Callee

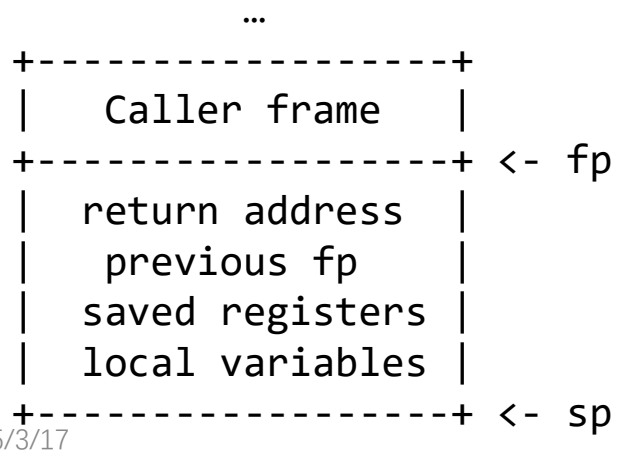
**Caller** –
Function that invokes another function
- **`sum_then_double`** (caller) -> `sum`
- saves caller-saved registers if needed

```
sum_then_double:
  addi    sp,sp,-32   # reserve stack space
  sw      ra,28(sp)   # save return address
  sw      s0,24(sp)   # save frame pointer
  addi    s0,sp,32    # change frame pointer
  sw      a0,-20(s0)  # save parameter 1
  sw      a1,-24(s0)  # save parameter 2
  lw      a1,-24(s0)
  lw      a0,-20(s0)
  call    sum
  mv      a5,a0       # a5 := return value of sum
  slli    a5,a5,1
  mv      a0,a5
  lw      ra,28(sp)   # restore return address
  lw      s0,24(sp)   # restore frame pointer
  addi    sp,sp,32    # release stack space
  jr      ra
```

Prologue

Body

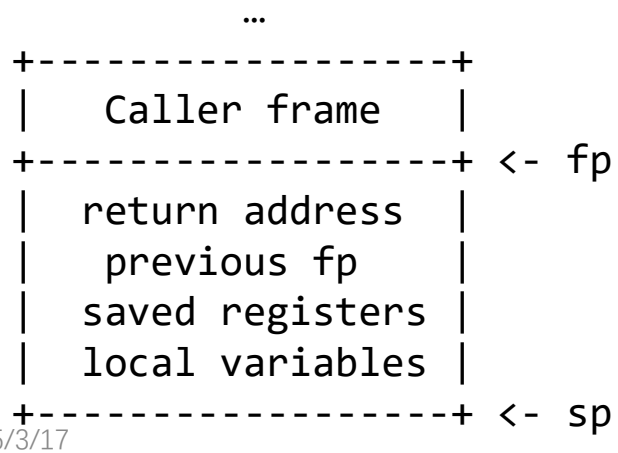
Epilogue




Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

**Caller** –
Function that invokes another function
- **`sum_then_double`** (caller) **`->`** `sum`
- saves caller-saved registers if needed
- prepares arguments (a0-a7)
  with additional parameters on stack



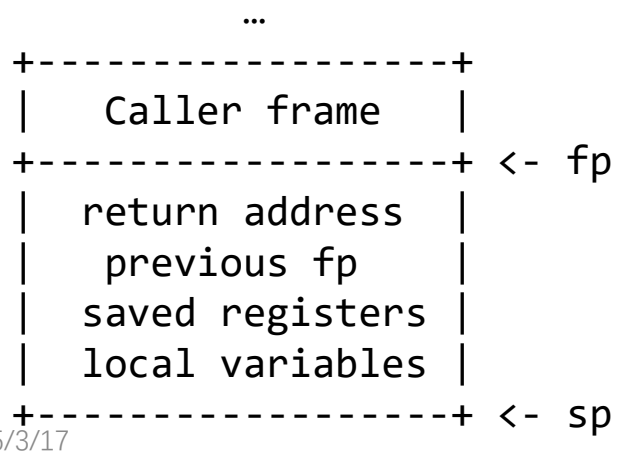

Prologue / Body / Epilogue

```
sum_then_double:
    addi    sp,sp,-32   # reserve stack space
    sw      ra,28(sp)   # save return address
    sw      s0,24(sp)   # save frame pointer
    addi    s0,sp,32    # change frame pointer
    sw      a0,-20(s0)  # save parameter 1
    sw      a1,-24(s0)  # save parameter 2
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    call    sum
    mv      a5,a0       # a5 := return value of sum
    slli    a5,a5,1
    mv      a0,a5
    lw      ra,28(sp)   # restore return address
    lw      s0,24(sp)   # restore frame pointer
    addi    sp,sp,32    # release stack space
    jr      ra
```

Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

**Caller** –
Function that invokes another function
- **`sum_then_double`** (caller) -> `sum`
- saves caller-saved registers if needed
- prepares arguments (a0-a7)
  with additional parameters on stack
- sets return address (ra)
  and jumps to function




```
sum_then_double:
  addi    sp,sp,-32  # reserve stack space
  sw      ra,28(sp)  # save return address
  sw      s0,24(sp)  # save frame pointer
  addi    s0,sp,32   # change frame pointer
  sw      a0,-20(s0) # save parameter 1
  sw      a1,-24(s0) # save parameter 2
  lw      a1,-24(s0)
  lw      a0,-20(s0)
  call    sum
  mv      a5,a0      # a5 := return value of sum
  slli    a5,a5,1
  mv      a0,a5
  lw      ra,28(sp)  # restore return address
  lw      s0,24(sp)  # restore frame pointer
  addi    sp,sp,32   # release stack space
  jr      ra
```

Prologue

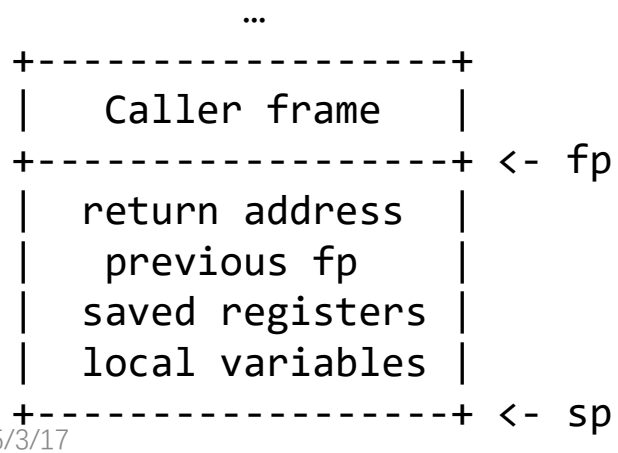
Body

Epilogue

Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

**Caller** –
Function that invokes another function
- **`sum_then_double`** (caller) -> `sum`
- saves caller-saved registers if needed
- prepares arguments (a0-a7)
  with additional parameters on stack
- sets return address (ra)
  and jumps to function
- restore caller-saved registers
  after callee returned




Prologue
Body
Epilogue

```
sum_then_double:
    addi    sp,sp,-32   # reserve stack space
    sw      ra,28(sp)   # save return address
    sw      s0,24(sp)   # save frame pointer
    addi    s0,sp,32    # change frame pointer
    sw      a0,-20(s0)  # save parameter 1
    sw      a1,-24(s0)  # save parameter 2
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    call    sum
    mv      a5,a0       # a5 := return value of sum
    slli    a5,a5,1
    mv      a0,a5
    lw      ra,28(sp)   # restore return address
    lw      s0,24(sp)   # restore frame pointer
    addi    sp,sp,32    # release stack space
    jr      ra
```

Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

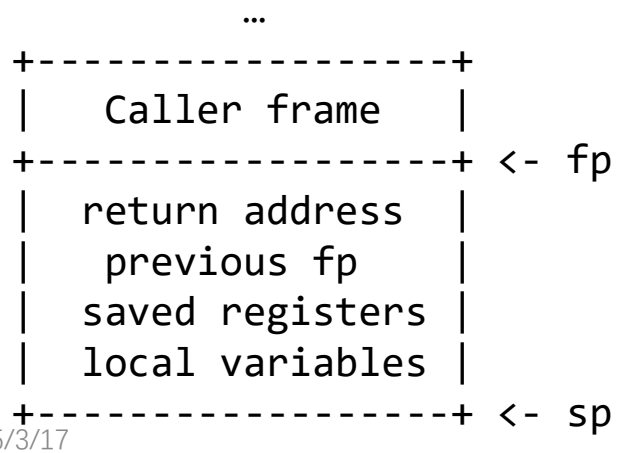
**Callee** –
Function that invoked by another function
- `main` -> **`sum_then_double`** (callee)
- saves callee-saved registers

```
sum_then_double:
    addi    sp,sp,-32   # reserve stack space
    sw      ra,28(sp)   # save return address
    sw      s0,24(sp)   # save frame pointer
    addi    s0,sp,32    # change frame pointer
    sw      a0,-20(s0)  # save parameter 1
    sw      a1,-24(s0)  # save parameter 2
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    call    sum
    mv      a5,a0       # a5 := return value of sum
    slli    a5,a5,1
    mv      a0,a5
    lw      ra,28(sp)   # restore return address
    lw      s0,24(sp)   # restore frame pointer
    addi    sp,sp,32    # release stack space
    jr      ra
```

Prologue

Body

Epilogue




Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

**Callee** –
Function that invoked by another function
- `main` -> **`sum_then_double`** (callee)
- saves callee-saved registers
- executes function logic

```
sum_then_double:
  addi    sp,sp,-32  # reserve stack space
  sw      ra,28(sp)  # save return address
  sw      s0,24(sp)  # save frame pointer
  addi    s0,sp,32   # change frame pointer
  sw      a0,-20(s0) # save parameter 1
  sw      a1,-24(s0) # save parameter 2
  lw      a1,-24(s0)
  lw      a0,-20(s0)
  call    sum
  mv      a5,a0      # a5 := return value of sum
  slli    a5,a5,1
  mv      a0,a5
  lw      ra,28(sp)  # restore return address
  lw      s0,24(sp)  # restore frame pointer
  addi    sp,sp,32   # release stack space
  jr      ra
```

Prologue

Body

Epilogue

```
                …
  +------------------+
  |   Caller frame   |
  +------------------+ <- fp
  |  return address  |
  |   previous fp    |
  |  saved registers |
  |  local variables |
  +------------------+ <- sp
```

Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

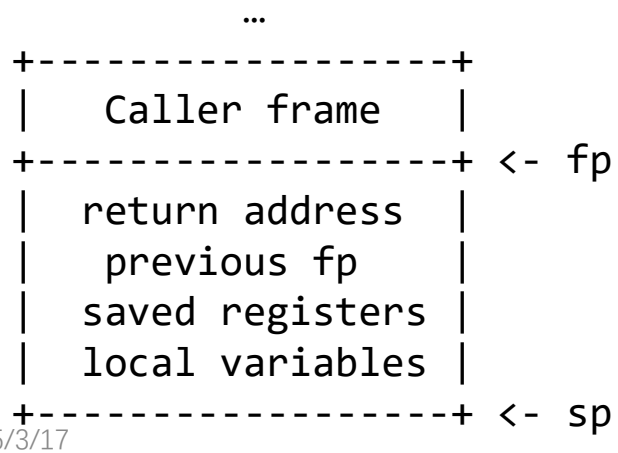
**Callee** –
Function that invoked by another function
- `main` -> **`sum_then_double`** (callee)
- saves callee-saved registers
- executes function logic
- restores callee-saved registers

```
sum_then_double:
  addi    sp,sp,-32  # reserve stack space
  sw      ra,28(sp)  # save return address
  sw      s0,24(sp)  # save frame pointer
  addi    s0,sp,32   # change frame pointer
  sw      a0,-20(s0) # save parameter 1
  sw      a1,-24(s0) # save parameter 2
  lw      a1,-24(s0)
  lw      a0,-20(s0)
  call    sum
  mv      a5,a0      # a5 := return value of sum
  slli    a5,a5,1
  mv      a0,a5
  lw      ra,28(sp)  # restore return address
  lw      s0,24(sp)  # restore frame pointer
  addi    sp,sp,32   # release stack space
  jr      ra
```

Prologue

Body

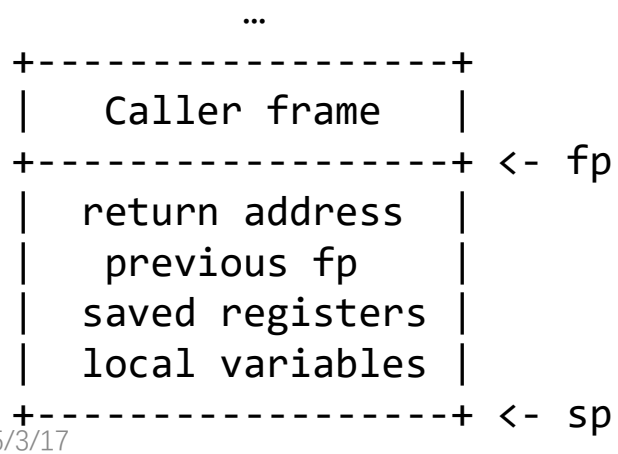
Epilogue




Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Caller & Callee

**Callee** –
Function that invoked by another function
- `main` -> **`sum_then_double`** (callee)
- saves callee-saved registers
- executes function logic
- restores callee-saved registers
- returns by jumping to ra




Prologue / Body / Epilogue

```
sum_then_double:
    addi    sp,sp,-32   # reserve stack space
    sw      ra,28(sp)   # save return address
    sw      s0,24(sp)   # save frame pointer
    addi    s0,sp,32    # change frame pointer
    sw      a0,-20(s0) # save parameter 1
    sw      a1,-24(s0) # save parameter 2
    lw      a1,-24(s0)
    lw      a0,-20(s0)
    call    sum
    mv      a5,a0       # a5 := return value of sum
    slli    a5,a5,1
    mv      a0,a5
    lw      ra,28(sp)   # restore return address
    lw      s0,24(sp)   # restore frame pointer
    addi    sp,sp,32    # release stack space
    jr      ra
```

Compiled by RISC-V(32 bit) gcc 14.2.0 Compiler Explorer

# Takeaways

- Calling conventions are contracts enabling interoperability
- Register responsibilities are clearly defined:
  - a0-a7: Arguments and return values
  - ra: Return address (caller-saved)
  - sp: Stack pointer (callee-saved)
  - t0-t6: Temporaries (caller-saved)
  - s0-s11: Saved registers (callee-saved)
- Stack management rules for reliable function nesting
- Unconditional jumps details

**For more details, see** riscv-spec.pdf

- Arguments on stack
- Passing/returning complex values
  - Aggregated structure
- Soft-float calling convention

...

# Exam Examples

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  # prologue. Tips: the minimum stack space
required for saving registers, disregard
stack alignment requirements, i.e., the
stack can be of any size (2 points, 1 for
correct number (consistent with your next
question's choice, e.g. if you select "EF"
in the next question and fill-8 in this
blank, you are correct!), 1 for
positive/negative)
  addi sp,sp,___
  # please select below all the registers
that must be saved here in the stack before
function call. (2 points, 0.25 for each
option A-G)
  sw ____
A. a0.          E. s0.
B. a1.          F. s1.
C. t0.          G. ra.
D. t1.          H. sp.
  # if c<0, jump to foo_true to return
  blt a0,x0,foo_true
  …
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  # prologue. Tips: the minimum stack space
required for saving registers, disregard
stack alignment requirements, i.e., the
stack can be of any size (2 points, 1 for
correct number (consistent with your next
question's choice, e.g. if you select "EF"
in the next question and fill-8 in this
blank, you are correct!), 1 for
positive/negative)
  addi sp,sp,-12
  # please select below all the registers
that must be saved here in the stack before
function call. (2 points, 0.25 for each
option A-G)
  sw E,F,G
A. a0.          E. s0.
B. a1.          F. s1.
C. t0.          G. ra.
D. t1.          H. sp.
  # if c<0, jump to foo_true to return
  blt a0,x0,foo_true
  …
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  addi sp,sp,-12
  sw s0,s1 and sp
  # if c<0, jump to foo_true to return
  blt a0,x0,foo_true
foo_false:
  mv s0,a0 # put c in s0 for further use
  li a0,__ # fill in the blank here to pass
the parameter to the malloc function that
to be called (1 point)
  call malloc
  …
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  addi sp, sp, -12
  sw s0,s1 and sp
  # if c<0, jump to foo_true to return
  blt a0,x0,foo_true
foo_false:
  mv s0,a0 # put c in s0 for further use
  li a0,8  # fill in the blank here to pass
the parameter to the malloc function that
to be called (1 point)
  call malloc
  …
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  addi sp,sp,-12
  sw s0,s1 and sp
  # if c<0, jump to foo_true to return
  blt a0,x0,foo_true
foo_false:
  mv s0,a0 # put c in s0 for further use
  li a0,8
  call malloc
  mv s1,a0 # put n in s1 for further use
  # calculate c-1 and pass the parameter to
foo function for recursive call (3points)
  addi _,_,_
  call foo
  # write the return value into n->next (2
points)
  _ _,_
  # write c into n->c (Tips: c is char
type.) (2points)
  _ _,_
  …
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  addi sp,sp,-12
  sw s0,s1 and sp
  # if c<0, jump to foo_true to return
  blt a0,x0,foo_true
foo_false:
  mv s0,a0 # put c in s0 for further use
  li a0,8
  call malloc
  mv s1,a0 # put n in s1 for further use
  # calculate c-1 and pass the parameter to
foo function for recursive call (3points)
  addi a0,s0,-1
  call foo
  # write the return value into n->next (2
points)
  sw a0,4(s1)
  # write c into n->c (Tips: c is char
type.) (2points)
  sb s0,0(s1)
  …
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  addi sp,sp,-12
  sw s0,s1 and sp
  blt a0,x0,foo_true
foo_false:
  mv s0,a0
  li a0,8
  call malloc
  mv s1,a0
  addi a0,s0,-1
  call foo
  sw a0,4(s1)
  sb s0,0(s1)
  mv a0,s1
  j foo_exit
foo_true:
  add a0,x0,x0
foo_exit:
  lw …
  # restore the stack pointer (2points)
  addi sp,sp,_
  ret
```

# SP2024 Midterm I No. 6

Assume a `struct` type defined as follows has the following layout, `c` at address 0 (or address x) and `next` at 4 (or address x+4) because of alignment. The size of `struct node` is then 8-byte.

```
struct node {unsigned char c, struct node *next};
struct node* foo(char c) {
  struct node *n;
  if (c < 0) return 0;
  n = malloc(sizeof(struct node));
  n->next = foo(c- 1);
  n->c = c;
  return n; }
```

Please **complete the RV32I assembly** implementing the `foo` function according to the comments and instructions below following **calling conventions**. Since we are calling other functions, assume local variable `c` is put in `s0` and `n` in `s1` so that we can still use these values after function call.

```
foo:
  addi sp,sp,-12
  sw s0,s1 and sp
  blt a0,x0,foo_true
foo_false:
  mv s0,a0
  li a0,8
  call malloc
  mv s1,a0
  addi a0,s0,-1
  call foo
  sw a0,4(s1)
  sb s0,0(s1)
  mv a0,s1
  j foo_exit
foo_true:
  add a0,x0,x0
foo_exit:
  lw …
  # restore the stack pointer (2points)
  addi sp,sp,12
  ret
```

# Project 1.1 Questions