# Function Calls

# Function Calls

- **Caller:** calling function (in this case, `main`)
- **Callee:** called function (in this case, `sum`)

**C Code**

```c
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

# Simple Function Call

**C Code**

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

**RISC-V assembly code**

```
0x00000300 main:   jal  simple       # call
0x00000304         add  s0, s1, s2
...                ...

0x0000051c simple: jr   ra           # return
```

**`void` means that `simple` doesn't return a value**

**`jal simple`:**

`ra` = PC + 4 (0x00000304)

jumps to `simple` label (PC = 0x0000051c)

**`jr ra`:**

PC = `ra` (0x00000304)

# Function Calling Conventions

- **Caller:**
  - passes **arguments** to callee
  - jumps to callee
- **Callee:**
  - **performs** the function
  - **returns** result to caller
  - **returns** to point of call
  - **must not overwrite** registers or memory needed by caller

# RISC-V Function Calling Conventions

- **Call Function:** jump and link (`jal func`)
- **Return** from function: jump register (`jr ra`)
- **Arguments**: `a0 – a7`
- **Return value**: `a0`

# Input Arguments & Return Value

## C Code

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);   // 4 arguments
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // return value
}
```

# Input Arguments & Return Value

**RISC-V assembly code**

```
# s7 = y
main:
. . .
addi a0, zero, 2   # argument 0 = 2
addi a1, zero, 3   # argument 1 = 3
addi a2, zero, 4   # argument 2 = 4
addi a3, zero, 5   # argument 3 = 5
jal  diffofsums    # call function
add  s7, a0, zero # y = returned value
. . .
# s3 = result
diffofsums:
add  t0, a0, a1    # t0 = f + g
add  t1, a2, a3    # t1 = h + i
sub  s3, t0, t1    # result = (f + g) - (h + i)
add  a0, s3, zero # put return value in a0
jr   ra            # return to caller
```

# Input Arguments & Return Value

**RISC-V assembly code**

```
# s3 = result
diffofsums:
  add  t0, a0, a1   # t0 = f + g
  add  t1, a2, a3   # t1 = h + i
  sub  s3, t0, t1   # result = (f + g) - (h + i)
  add  a0, s3, zero # put return value in a0
  jr   ra           # return to caller
```

- `diffofsums` overwrote 3 registers: `t0`, `t1`, `s3`
- `diffofsums` can use *stack* to temporarily store registers

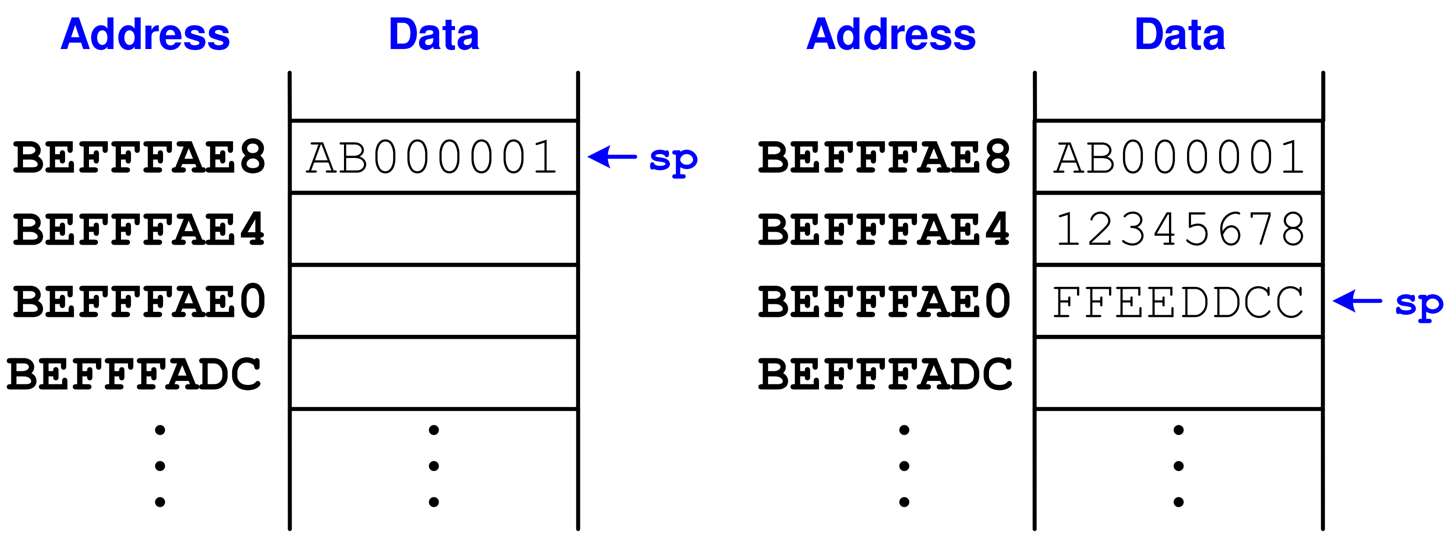
# The Stack

# The Stack

- Memory used to temporarily save variables
- Like stack of dishes, last-in-first-out (LIFO) queue
- ***Expands***: uses more memory when more space needed
- ***Contracts***: uses less memory when the space is no longer needed

# The Stack

- Grows down (from higher to lower memory addresses)
- Stack pointer: `sp` points to top of the stack




Make room on stack for **2 words**.

# How Functions use the Stack

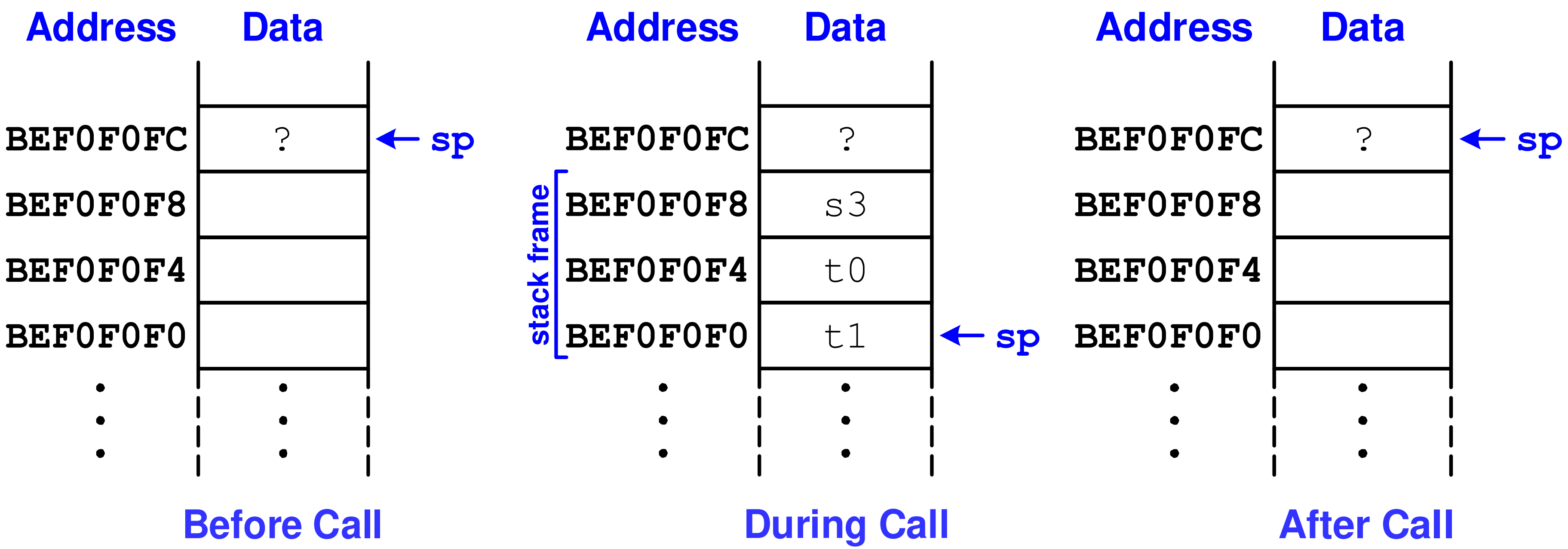
- Called functions must have no unintended side effects
- But `diffofsums` overwrites 3 registers: `t0`, `t1`, `s3`

```
# RISC-V assembly
# s3 = result
diffofsums:
  add  t0, a0, a1   # t0 = f + g
  add  t1, a2, a3   # t1 = h + i
  sub  s3, t0, t1   # result = (f + g) - (h + i)
  add  a0, s3, zero # put return value in a0
  jr   ra           # return to caller
```

# Storing Register Values on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -12        # make space on stack to
                          # store three registers
  sw   s3, 8(sp)          # save s3 on stack
  sw   t0, 4(sp)          # save t0 on stack
  sw   t1, 0(sp)          # save t1 on stack
  add  t0, a0, a1         # t0 = f + g
  add  t1, a2, a3         # t1 = h + i
  sub  s3, t0, t1         # result = (f + g) - (h + i)
  add  a0, s3, zero       # put return value in a0
  lw   s3, 8(sp)          # restore s3 from stack
  lw   t0, 4(sp)          # restore t0 from stack
  lw   t1, 0(sp)          # restore t1 from stack
  addi sp, sp, 12         # deallocate stack space
  jr   ra                 # return to caller
```

# The Stack During `diffofsums` Call



# Preserved Registers

| **Preserved**<br>***Callee-Saved*** | **Nonpreserved**<br>***Caller-Saved*** |
|---|---|
| `s0-s11` | `t0-t6` |
| `sp` | `a0-a7` |
| `ra` | |
| stack above `sp` | stack below `sp` |

# Storing Saved Registers on the Stack

```
# s3 = result
diffofsums:
  addi sp, sp, -4        # make space on stack to
                         # store one register
  sw   s3, 0(sp)         # save s3 on stack
  add  t0, a0, a1        # t0 = f + g
  add  t1, a2, a3        # t1 = h + i
  sub  s3, t0, t1        # result = (f + g) - (h + i)
  add  a0, s3, zero      # put return value in a0
  lw   s3, 0(sp)         # restore $s3 from stack
  addi sp, sp, 4         # deallocate stack space
  jr   ra                # return to caller
```

# Optimized `diffofsums`

```
# a0 = result
diffofsums:
  add   t0, a0, a1    # t0 = f + g
  add   t1, a2, a3    # t1 = h + i
  sub   a0, t0, t1    # result = (f + g) - (h + i)
  jr    ra            # return to caller
```

# Non-Leaf Function Calls

## Non-leaf function:

a function that calls another function

```
func1:
  addi sp, sp, -4    # make space on stack
  sw   ra, 0(sp)     # save ra on stack
  jal  func2
  ...
  lw   ra, 0(sp)     # restore ra from stack
  addi sp, sp, 4     # deallocate stack space
  jr   ra            # return to caller
```

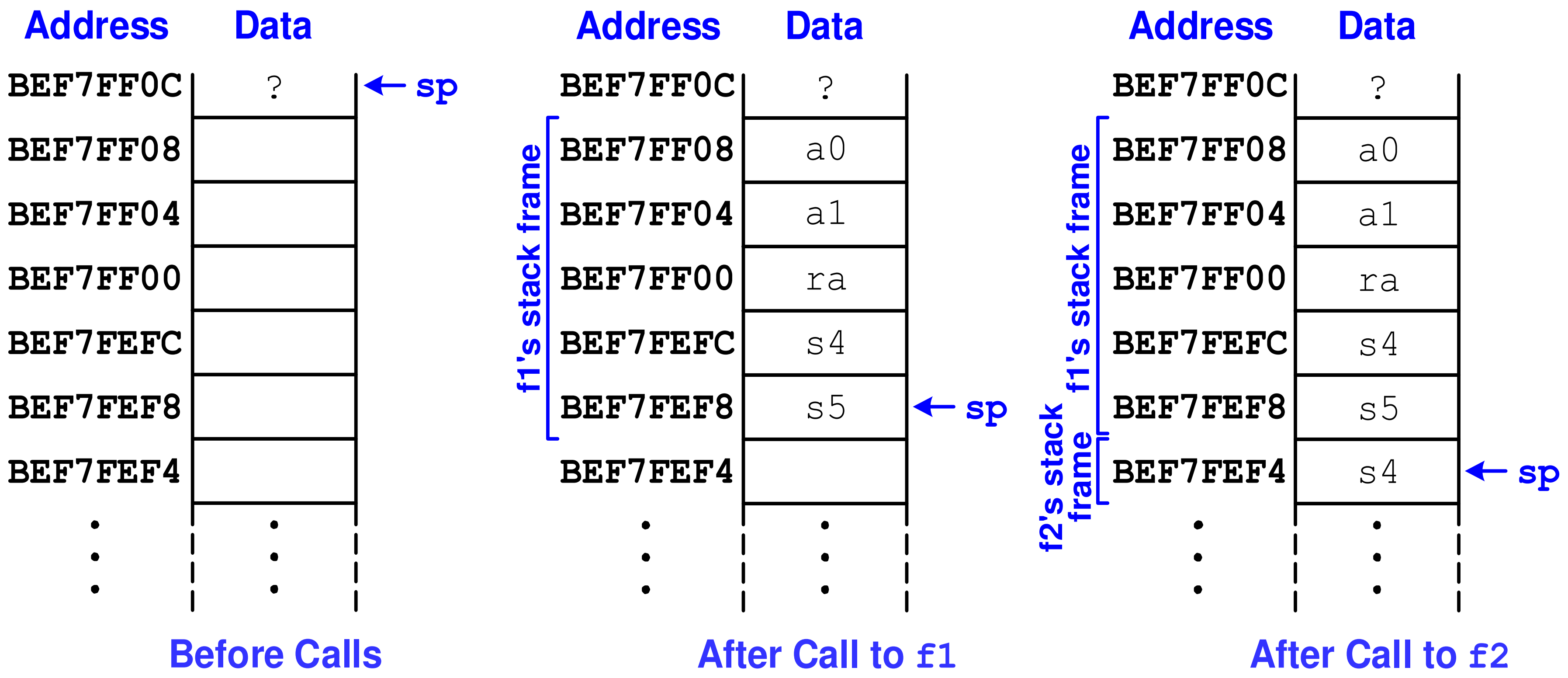
Must preserve **ra** before function call.

# Non-Leaf Function Call Example

```
# f1 (non-leaf function) uses s4-s5 and needs a0-a1 after call to f2
f1:
  addi sp, sp, -20    # make space on stack for 5 words
  sw   a0, 16(sp)
  sw   a1, 12(sp)
  sw   ra, 8(sp)      # save ra on stack
  sw   s4, 4(sp)
  sw   s5, 0(sp)
  jal  func2
  ...
  lw   ra, 8(sp)      # restore ra (and other regs) from stack
  ...
  addi sp, sp, 20     # deallocate stack space
  jr   ra             # return to caller

# f2 (leaf function) only uses s4 and calls no functions
f2:
  addi sp, sp, -4     # make space on stack for 1 word
  sw   s4, 0(sp)
  ...
  lw   s4, 0(sp)
  addi sp, sp, 4      # deallocate stack space
  jr   ra             # return to caller
```

# Stack during Function Calls



# Function Call Summary

- **Caller**
  - Save any needed registers (`ra`, maybe `t0-t6/a0-a7`)
  - Put arguments in `a0-a7`
  - Call function: `jal callee`
  - Look for result in `a0`
  - Restore any saved registers

- **Callee**
  - Save registers that might be disturbed (`s0-s11`)
  - Perform function
  - Put result in `a0`
  - Restore registers
  - Return: `jr ra`

# Recursive Functions

# Recursive Function Example

- Function that **calls itself**
- When converting to assembly code:
  - In the first pass, treat recursive calls as if it’s calling a different function and ignore overwritten registers.
  - Then save/restore registers on stack as needed.

# Recursive Function Example

- **Factorial function:**
  - factorial(n) = n!
    = n*(n-1)*(n-2)*(n-3)…*1
  - **Example:** factorial(6) = 6!
    = 6*5*4*3*2*1
    = 720

# Recursive Function Example

**High-Level Code**

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return (n*factorial(n-1));
}
```

**Example: n = 3**

```
factorial(3): returns 3*factorial(2)
factorial(2): returns 2*factorial(1)
factorial(1): returns 1
```

**Thus,**

```
factorial(1): returns 1
factorial(2): returns 2*1 = 2
factorial(3): returns 3*2 = 6
```

# Recursive Function Example

**High-Level Code**

```
int factorial(int n) {




  if (n <= 1)
    return 1;



  else
    return (n*factorial(n-1));
}
```

**RISC-V Assembly**

```
factorial:
```

**Pass 1.** Treat as if calling another function. Ignore stack.
**Pass 2.** Save overwritten registers (needed after function call) on the stack before call.

# Recursive Function Example

### High-Level Code

```
int factorial(int n) {




  if (n <= 1)
    return 1;


  else
    return (n*factorial(n-1));
}
```

**Pass 1.** Treat as if calling another function. Ignore stack.
**Pass 2.** Save overwritten registers (needed after function call) on the stack before call.

### RISC-V Assembly

```
factorial:
  addi sp, sp, -8    # save regs
  sw   a0, 4(sp)
  sw   ra, 0(sp)
  addi t0, zero, 1   # temporary = 1
  bgt  a0, t0, else  # if n>1, go to else
  addi a0, zero, 1   # otherwise, return 1
  addi sp, sp, 8     # restore sp
  jr   ra            # return
else:
  addi a0, a0, -1    # n = n - 1
  jal  factorial     # recursive call
  lw   t1, 4(sp)     # restore n into t1
  lw   ra, 0(sp)     # restore ra
  addi sp, sp, 8     # restore sp
  mul  a0, t1, a0    # a0=n*factorial(n-1)
  jr   ra            # return
```

**Note:** `n` is restored from stack into `t1` so it doesn't overwrite return value in `a0`.

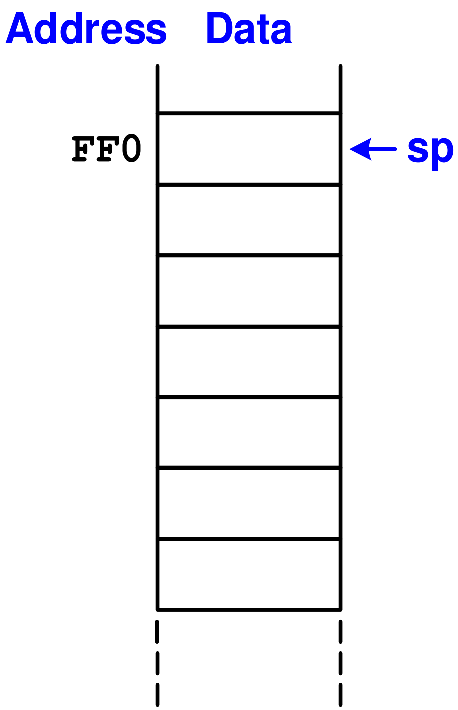
# Recursive Functions

```
0x8500 factorial: addi sp, sp, -8     # save registers
0x8504            sw   a0, 4(sp)
0x8508            sw   ra, 0(sp)
0x850C            addi t0, zero, 1    # temporary = 1
0x8510            bgt  a0, t0, else   # if n > 1, go to else
0x8514            addi a0, zero, 1    # otherwise, return 1
0x8518            addi sp, sp, 8      # restore sp
0x851C            jr   ra             # return
0x8520 else:      addi a0, a0, -1     # n = n - 1
0x8524            jal  factorial      # recursive call
0x8528            lw   t1, 4(sp)      # restore n into t1
0x852C            lw   ra, 0(sp)      # restore ra
0x8530            addi sp, sp, 8      # restore sp
0x8534            mul  a0, t1, a0     # a0 = n*factorial(n-1)
0x8538            jr   ra             # return
```

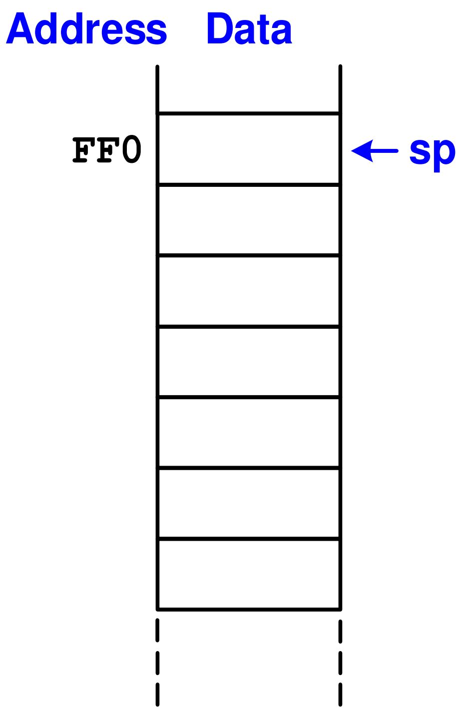
**PC+4 = 0x8528** when factorial is called recursively.

# Stack During Recursive Function

When **`factorial(3)`** is called:




Before Calls




After Recursive Calls

# More on Jumps & Pseudoinstructions

# Jumps

- RISC-V has two types of unconditional jumps
  - Jump and link (`jal rd,` $\texttt{imm}_{20:0}$)
    - **rd** = PC+4; PC = PC + **imm**
  - jump and link register (`jalr rd, rs,` $\texttt{imm}_{11:0}$)
    - **rd** = PC+4; PC = [**rs**] + SignExt(**imm**)

# Pseudoinstructions

- **Pseudoinstructions** are not actual RISC-V instructions but they are often more convenient for the programmer.
- Assembler converts them to real RISC-V instructions.

# Jump Pseudoinstructions

- RISC-V has four jump psuedoinstructions
  - `j    imm     jal  x0, imm`
  - `jal  imm     jal  ra, imm`
  - `jr   rs      jalr x0, rs, 0`
  - `ret          jr   ra` (i.e., `jalr x0, ra, 0`)

# Labels

- Label indicates where to jump
- Represented in jump as immediate offset
  - **imm** = # bytes past jump instruction
  - In example, below, **imm** = (51C-300) = 0x21C
  - `jal simple = jal ra, 0x21C`

RISC-V assembly code

```
0x00000300 main:   jal  simple       # call
0x00000304         add  s0, s1, s1
...                ...


0x0000051c simple: jr   ra           # return
```

# Long Jumps

- The immediate is limited in size
  - 20 bits for `jal`, 12 bits for `jalr`
  - Limits how far a program can jump
- Special instruction to help jumping further
  - `auipc rd, imm`: add upper immediate to PC
    - rd = PC + {$imm_{31:12}$, 12'b0}
- Pseudoinstruction: call $imm_{31:0}$
  - Behaves like `jal imm`, but allows 32-bit immediate offset

    auipc ra, $imm_{31:12}$

    jalr ra, ra, $imm_{11:0}$

# More RISC-V Pseudoinstructions

| Pseudoinstruction | RISC-V Instructions |
|---|---|
| `j label` | `jal zero, label` |
| `jr ra` | `jalr zero, ra, 0` |
| `mv t5, s3` | `addi t5, s3, 0` |
| `not s7, t2` | `xori s7, t2, -1` |
| `nop` | `addi zero, zero, 0` |
| `li s8, 0x56789DEF` | `lui s8, 0x5678A`<br>`addi s8, s8, 0xDEF` |
| `bgt s1, t3, L3` | `blt t3, s1, L3` |
| `bgez t2, L7` | `bge t2, zero, L7` |
| `call L1` | auipc ra, $imm_{31:12}$<br>jalr ra, ra, $imm_{11:0}$ |
| `ret` | `jalr zero, ra, 0` |

See Appendix B for more pseudoinstructions.