Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2019

# RISC-V Architecture II

# RISC-V:
# Control Transfer Operations

Chap. 2.7, 2.10

# Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

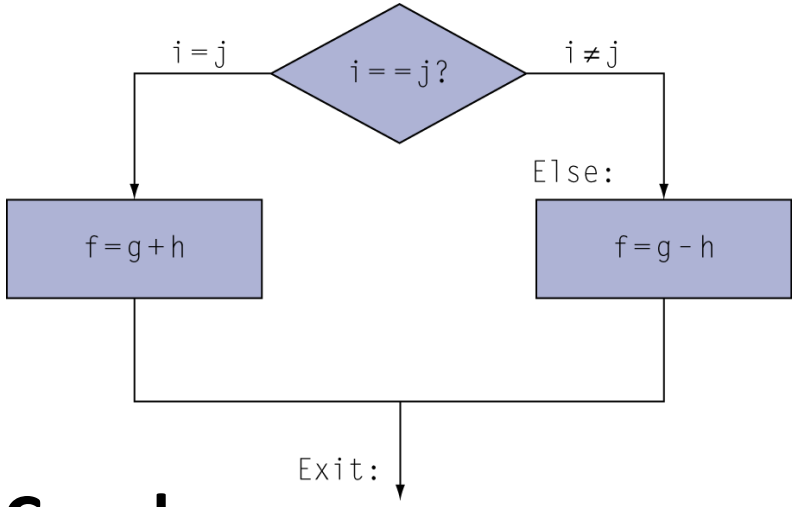
```
beq rs1, rs2, L1
```

  - if (`rs1 == rs2`), branch to instruction labeled `L1`

```
bne rs1, rs2, L1
```

  - if (`rs1 != rs2`), branch to instruction labeled `L1`

# Compiling If Statements




**C code:**

```
if (i == j)
  f = g + h;
else
  f = g – h;
```

**Compiled RISC-V code:**

```
// i in x22, j in x23
// f in x19, g in x20, h in x21

      bne  x22, x23, L1
      add  x19, x20, x21
      beq  x0, x0, Exit   // unconditional
L1:   sub  x19, x20, x21
Exit: ...
```

**Assembler calculates addresses**

# Compiling Loop Statements

**C code:**

```
while (A[i] == k)
  i += 1;
```

**Compiled RISC-V code:**

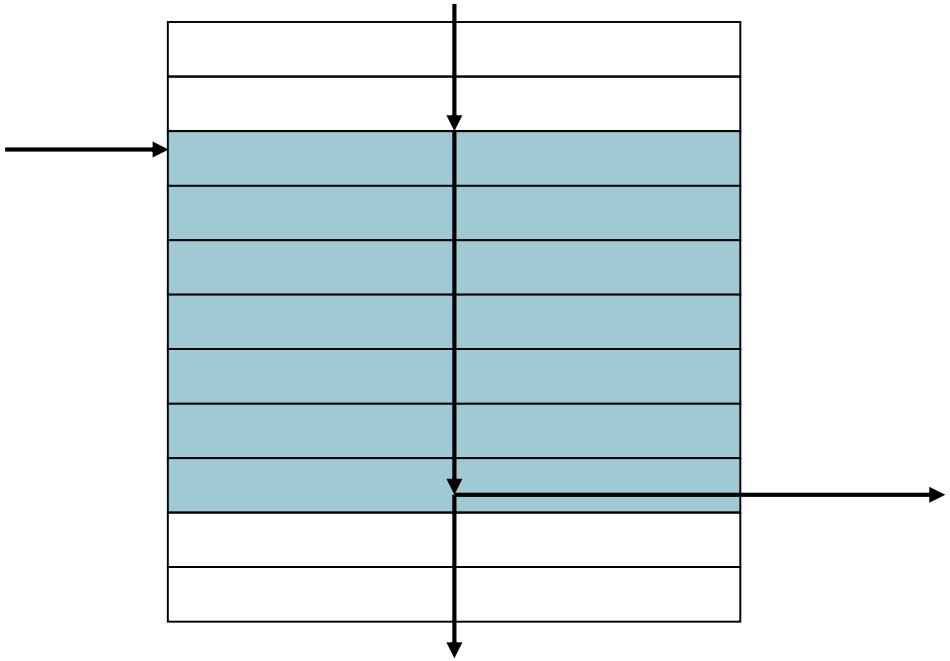
```
 // i in x22, k in x24
 // address of A[] in x25

Loop: slli  x10, x22, 3
      add   x10, x10, x25
      ld    x9, 0(x10)
      bne   x9, x24, Exit
      addi  x22, x22, 1
      beq   x0, x0, Loop
 Exit: ...
```

# Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)

- A compiler identifies basic blocks for optimization

- An advanced processor can accelerate execution of basic blocks



# More Conditional Operations

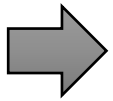
```
blt rs1, rs2, L1
```

- if (rs1 < rs2), branch to instruction labeled L1

```
bge rs1, rs2, L1
```

- if (rs1 >= rs2), branch to instruction labeled L1

```
if (a > b)
  a += 1;
```



```
        bge   x23, x22, Exit
        addi  x22, x22, 1
Exit:   ...
```

# Signed vs. Unsigned Comparison

- Signed comparison: `blt`, `bge`
- Unsigned comparison: `bltu`, `bgeu`

- Example

| | |
|---|---|
| x22 | 11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111 |
| x23 | 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000001 |

```
blt   x22, x23, Exit
```

**= Go to Exit if -1 < 1**

```
bltu  x22, x23, Exit
```

**= Go to Exit if $2^{64}$-1 < 1**

# Target Addressing

- Target addresses are always aligned to 2 bytes (i.e., even addresses)
  - Some of instructions can be encoded with 16 bits (with C extension)
  - PC-relative
- Branch addressing
  - Most branch targets are near branch: forward or backward
  - Target address = PC + SignExt(12-bit immediate value << 1)
- Jump addressing
  - Jump and link (`jal`) target uses 20-bit immediate for larger range
  - Target address = PC + SignExt(20-bit immediate value << 1)
  - For long jumps:
    (e.g., 32-bit absolute address)

| | |
|---|---|
| `lui`: | load address [31:12] to temp register |
| `jalr`: | add address [11:0] and jump to target |

# Control Transfer Instructions

| Instruction | Type | Example | Meaning |
|---|---|---|---|
| **Branch equal** | **SB** | `beq rs1, rs2, imm12` | `if (R[rs1] == R[rs2])`<br>`  pc = pc + SignExt(imm12 << 1)` |
| **Branch not equal** | **SB** | `bne rs1, rs2, imm12` | `if (R[rs1] != R[rs2])`<br>`  pc = pc + SignExt(imm12 << 1)` |
| **Branch greater than or equal** | **SB** | `bge rs1, rs2, imm12` | `if (R[rs1] >= R[rs2])`<br>`  pc = pc + SignExt(imm12 << 1)` |
| **Branch greater than or equal unsigned** | **SB** | `bgeu rs1, rs2, imm12` | `if (R[rs1] >=u R[rs2])`<br>`  pc = pc + SignExt(imm12 << 1)` |
| **Branch less than** | **SB** | `blt rs1, rs2, imm12` | `if (R[rs1] < R[rs2])`<br>`  pc = pc + SignExt(imm12 << 1)` |
| **Branch less than unsigned** | **SB** | `bltu rs1, rs2, imm12` | `if (R[rs1] <u R[rs2])`<br>`  pc = pc + SignExt(imm12 << 1)` |
| **Jump and link** | **UJ** | `jal rd, imm20` | `R[rd] = PC + 4`<br>`PC = PC + SignExt(imm20 << 1)` |
| **Jump and link register** | **I** | `jalr rd, imm12(rs1)` | `R[rd] = PC + 4`<br>`PC = (R[rs1] + SignExt(imm12)) & (~1)` |

# Conditional Branch Example

```
long max (long x, long y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

```
long goto_max (long x, long y)
{
    if (x <= y)
        goto done;
    y = x;
done:
    return y;
}
```

```
    # x is in a0
    # y is in a1

max:
    ble     a0, a1, L1      # if (x <= y) goto L1
    addi    a1, a0, 0       # a1 = x
L1:
    addi    a0, a1, 0       # a0 = a1
    ret
```

# Do-While Loop Example

```
long fact_do (long x) {
    long result = 1;
    do {
        result *= x;
        x = x - 1;
    } while (x > 1);
    return result;
}
```

```
long fact_do (long x) {
    long result = 1;
Loop:
    result = result * x;
    x = x - 1;
    if (x > 1) goto Loop;
    return result;
}
```

```
    # x is in a0

fact_do:
    addi    a5, a0, 0       # a5 = x (x)
    addi    a0, zero, 1     # a0 = 1 (result)
L2:
    mul     a0, a0, a5      # result *= x
    addi    a5, a5, -1      # x = x - 1
    addi    a4, zero, 1     # a4 = 1
    bgt     a5, a4, L2      # if (x > 1) goto L2
    ret
```

# Do-While Loop

- General "Do-While" translation

**C Code**

```
do
  Body
  while (Test);
```

**Goto Version**

```
Loop:
  Body
  if (Test)
    goto Loop
```

- *Body* can be any C statement
  - Typically compound statement:

```
{
    Statement1;
    Statement2;
      …
    Statementn;
}
```

- *Test* is expression returning integer:
  = 0 interpreted as false, ≠ 0 interpreted as true

# While Loop Example (1)

```
long fact_while (long x) {
    long result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

```
long fact_while (long x) {
    long result = 1;
Loop:
    if (x <= 1) goto Exit;
    result = result * x;
    x = x - 1;
    goto Loop;
Exit:
    return result;
}
```

**gcc with –Og option**

```
     # x is in a0

fact_while:
     addi    a5, a0, 0        # a5 = x (x)
     addi    a0, zero, 1      # a0 = 1 (result)
L2:
     addi    a4, zero, 1      # a4 = 1
     ble     a5, a4, L4       # if (x <= 1) goto L4
     mul     a0, a0, a5       # result *= x
     addi    a5, a5, -1       # x = x - 1
     beq     zero, zero, L2   # goto L2
L4:
     ret
```

# While Loop Example (2)

```
long fact_while (long x) {
    long result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```

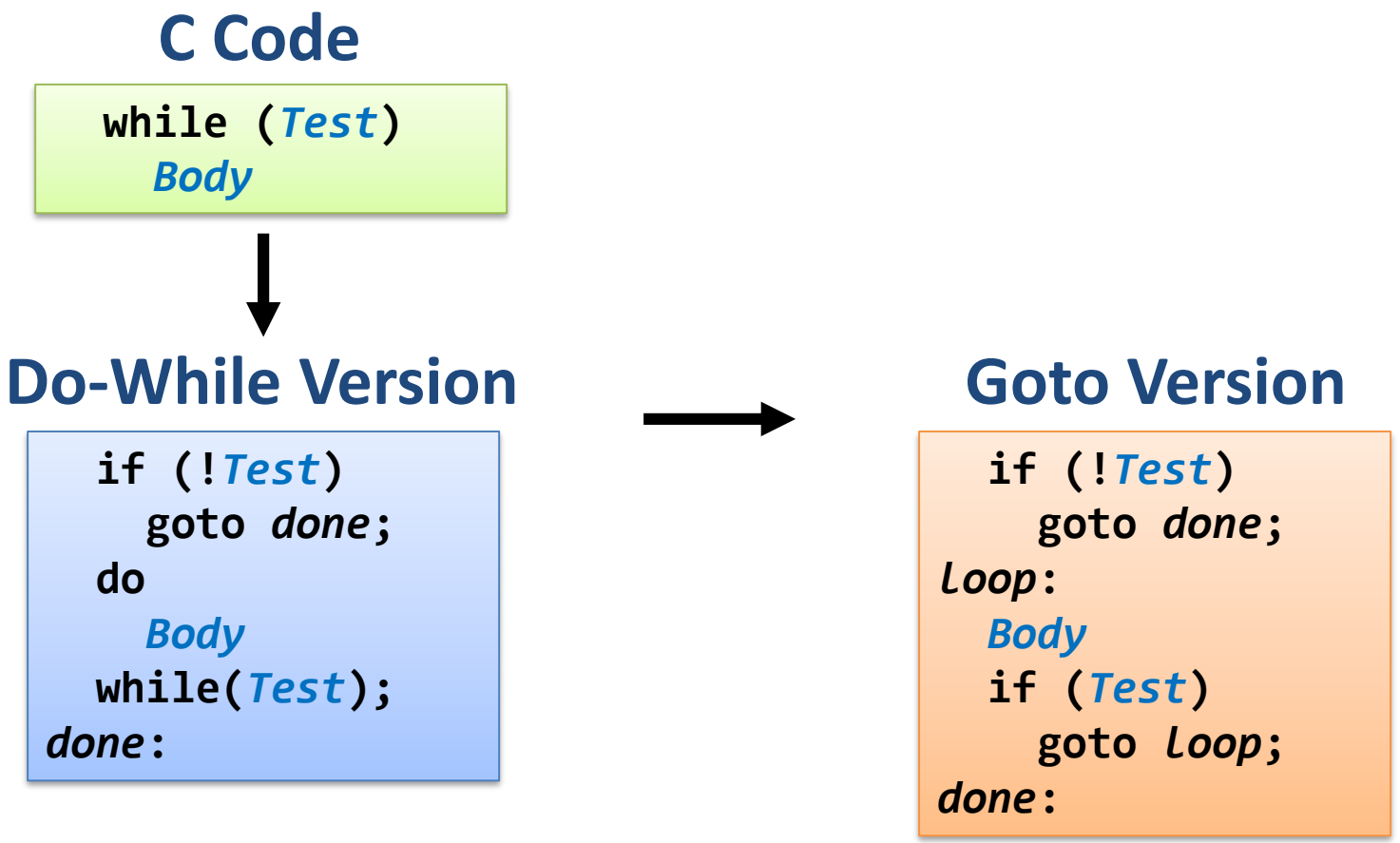
```
long fact_while2 (long x) {
    long result = 1;
    if (x <= 1) goto Exit;
Loop:
    result = result * x;
    x = x - 1;
    if (x != 1) goto Loop;
Exit:
    return result;
}
```

**gcc with –O2 option**

```
     # x is in a0

fact_while2:
     addi    a5, a0, 0        # a5 = x (x)
     addi    a4, zero, 1      $ a4 = 1
     addi    a0, zero, 1      # a0 = 1 (result)
     ble     a5, a4, L4       # if (x <= 1) goto L4
L3:
     mul     a0, a0, a5       # result *= x
     addi    a5, a5, -1       # x = x - 1
     bne     a5, a4, L3       # if (x != 1) goto L3
L4:
     ret
```

# While Loop

- General "While" translation



# For Loop

## For Version

```
for (Init; Test; Update)
    Body
```

## While Version

```
Init;
while (Test) {
    Body
    Update;
}
```

## Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do {
    Body
    Update;
  } while (Test)
done:
```

## Goto Version

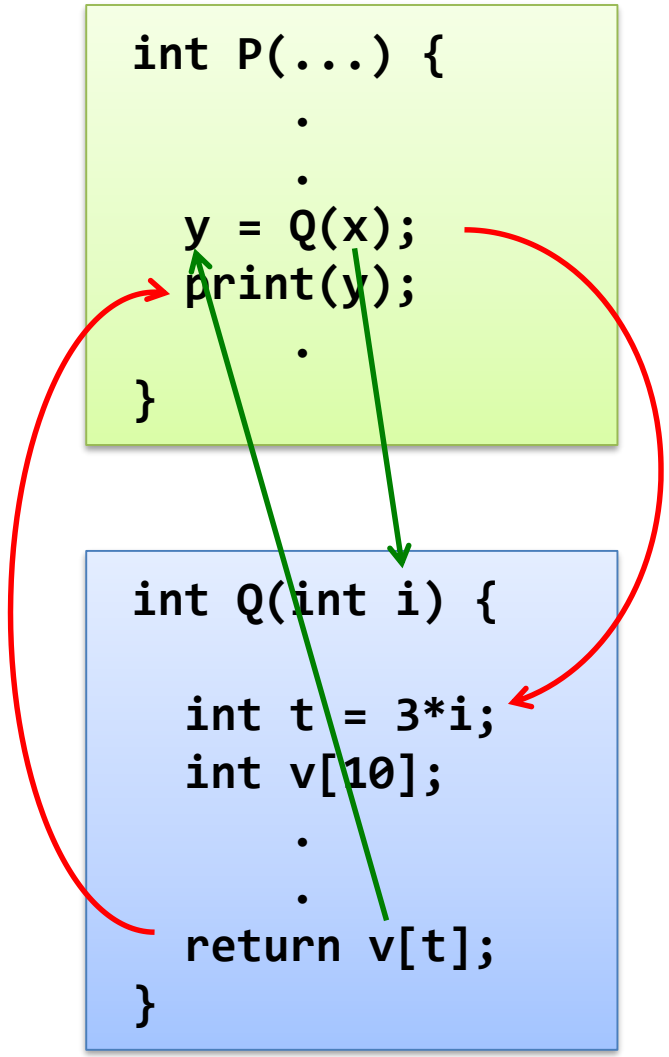
```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update;
  if (Test)
    goto loop;
done:
```

# RISC-V:
# Procedure Call

Chap. 2.8

# Mechanisms in Procedures

- Passing control
  - To beginning of procedure code
  - Back to return point
- Passing data
  - Procedure arguments
  - Return value
- Memory management
  - Allocate during procedure execution
  - Deallocate upon return
- All implemented with machine instructions

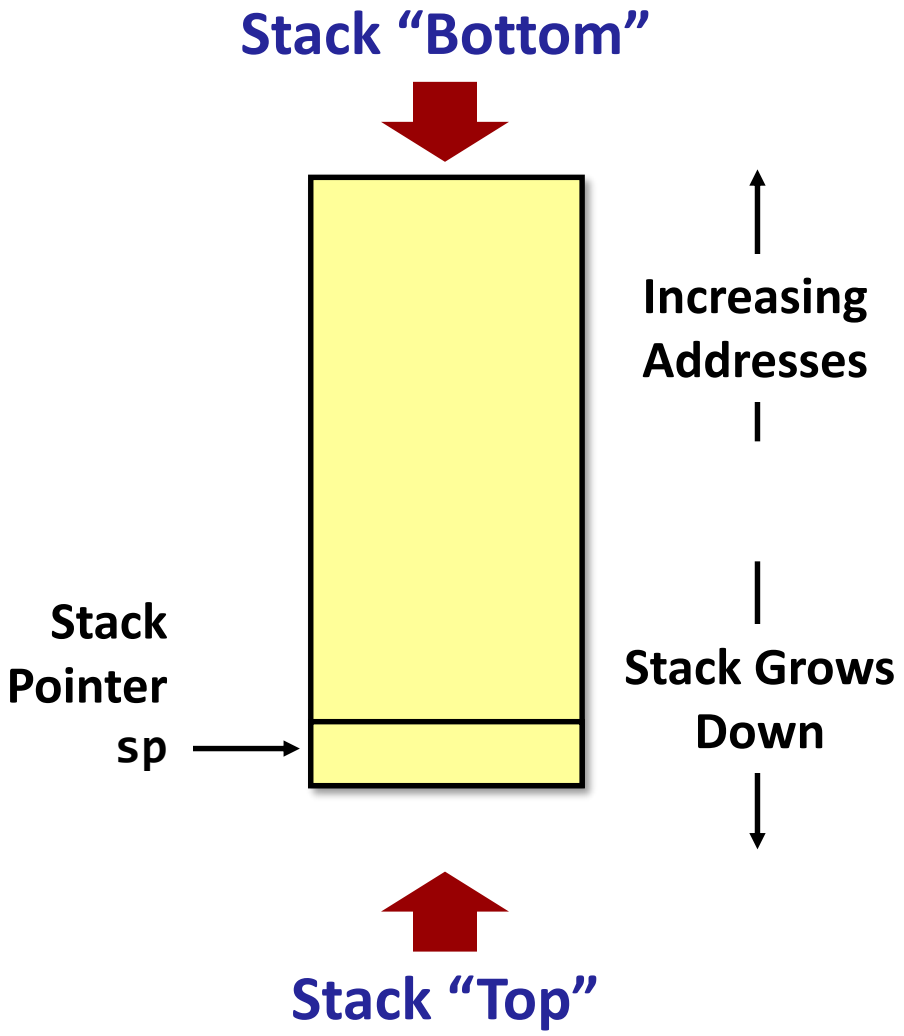


# Procedure Calling in RISC-V

- Place parameters in registers `x10` to `x17` (or `a0` to `a7`)
- Transfer control to procedure, saving the return address in `ra`
- Acquire storage for procedure
- Perform procedure's operations
- Place result in register `a0` (and `a1`) for caller
- Return to the next instruction of call (address in `ra`)

# RISC-V Stack

- Region of memory managed with stack discipline
  - Last-In, First-Out (LIFO)
  - No explicit push/pop operations
  - Load/store instructions used to access stack memory
- Grows toward lower addresses
- Register **sp (x2)** contains lowest stack address
  - Address of "top" element



# Procedure Call Instructions

- Procedure call: jump and link
  - Address of following instruction put in x1
  - Jumps to target address

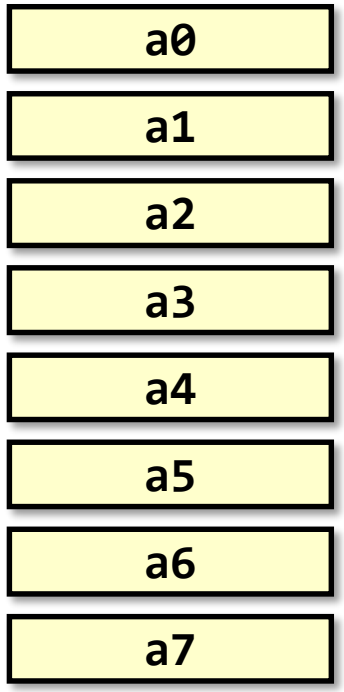
```
jal  ra, func
```

- Procedure return: jump and link register
  - Like `jal`, but jumps to 0 + address in x1
  - Use `x0` as `rd` (`x0` cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

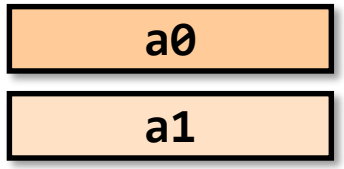
```
jalr x0, 0(ra)
```

# Passing Arguments

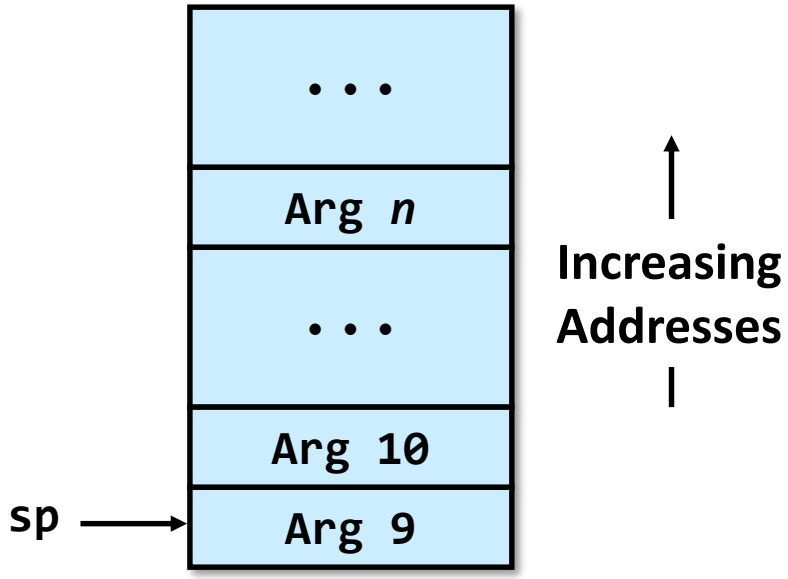
- First 8 arguments:




- Return value




- Remaining arguments:
  - Push the rest on the stack in reverse order
  - Only allocate stack space when needed

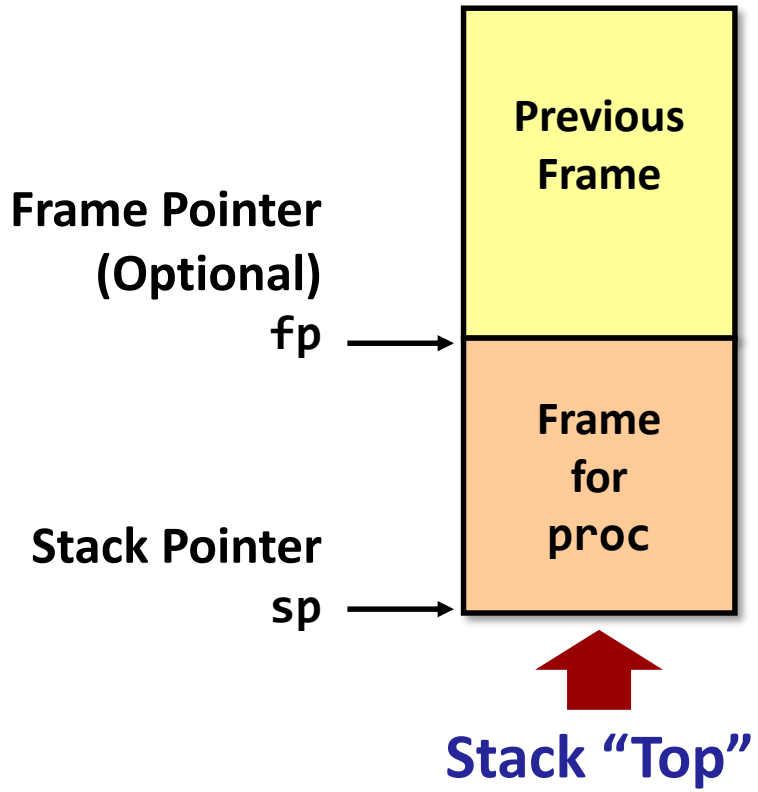


# Stack-based Languages

- Languages that support recursion (e.g. C, C++, Pascal, Java)
  - Code must be "Reentrant"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments, local variables, return address
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in *frames*
  - State for single procedure instantiation

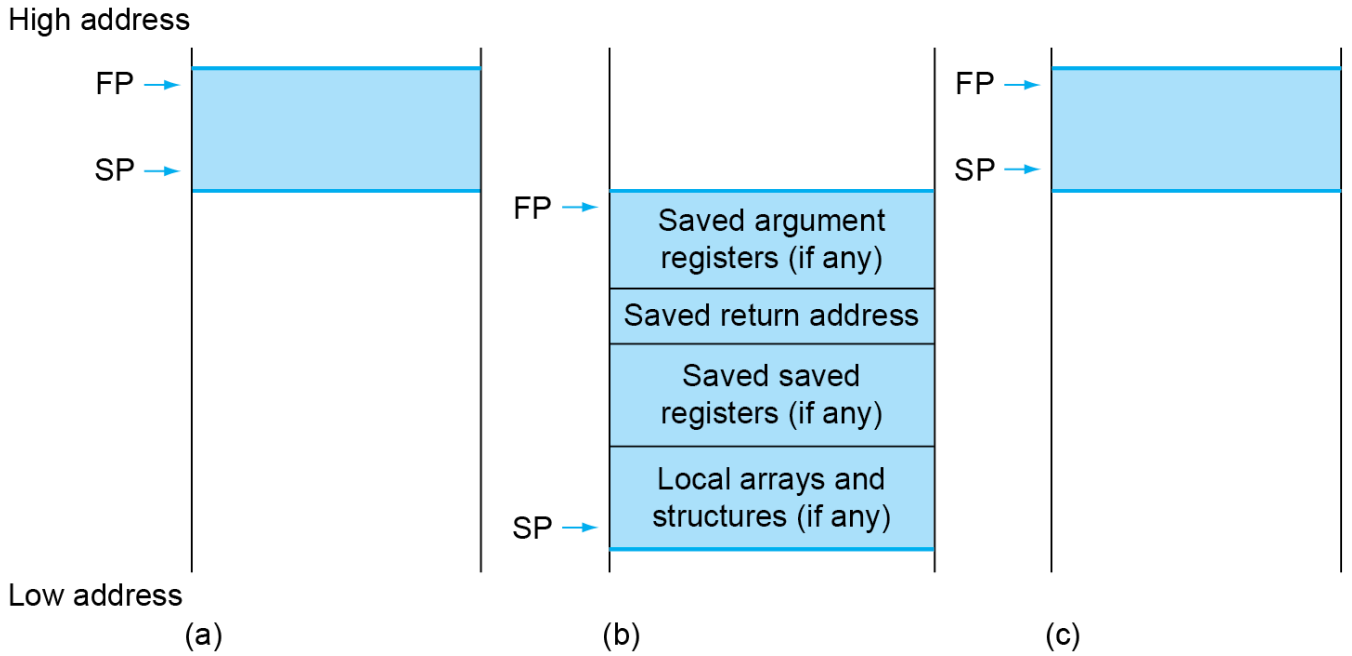
# Stack Frame

- Contents
  - Return information
  - Arguments
  - Local variables & temp space
- Management
  - "Set-up" code: space allocated when enter procedure
  - "Finish" code: deallocate when return
  - Stack pointer **sp** indicates stack top
  - Optional frame pointer **fp** indicates start of current frame



# Local Data on the Stack

- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage



# Leaf Procedure Example (1)

```
long leaf(long g,
          long h,
          long i,
          long j)
{
    long f;
    f = (g + h) - (i + j);

    return f;
}
```

g in x10
h in x11
i in x12
j in x13

```
leaf:

    add     x5, x10, x11   ; x5 <- g + h
    add     x6, x12, x13   ; x6 <- i + j
    sub     x20, x5, x6    ; x20 <- x5 + x6
    addi    x10, x20, 0    ; x10 <- x20

    jalr    x0, 0(x1)      ; return
```

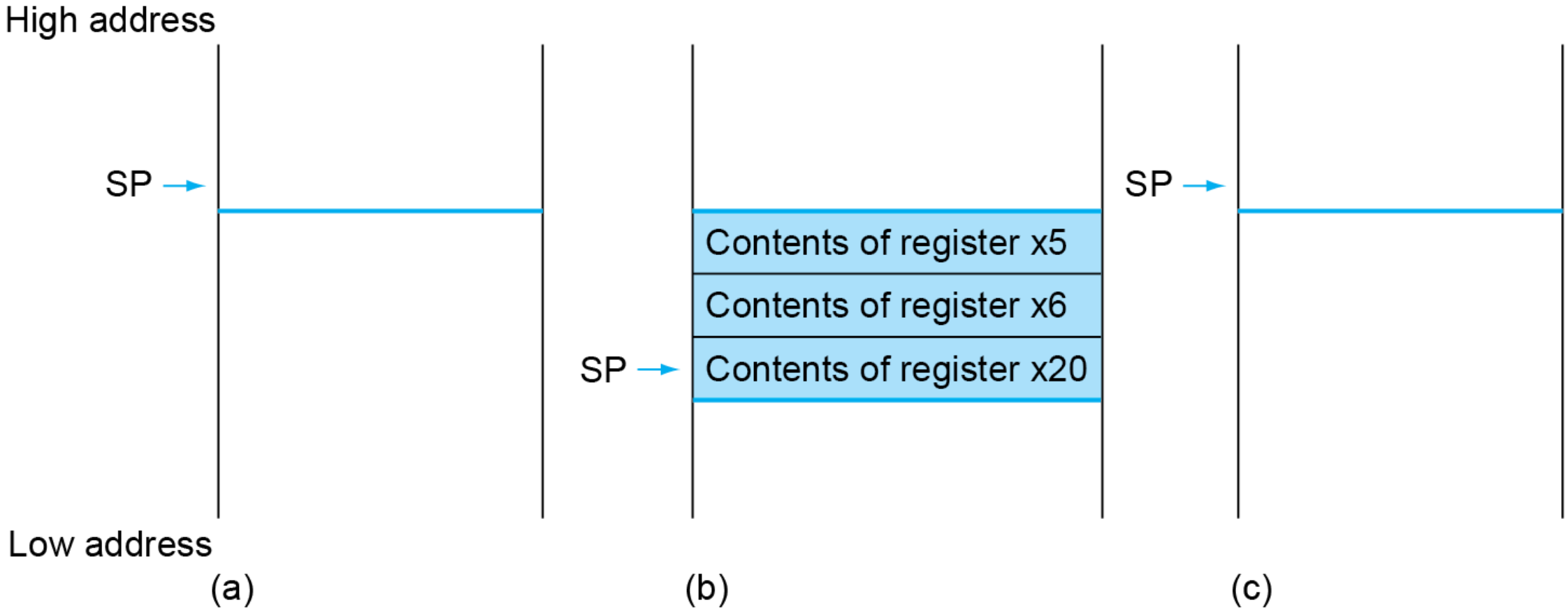
# Leaf Procedure Example (2)

```
long leaf(long g,
          long h,
          long i,
          long j)
{
    long f;
    f = (g + h) - (i + j);

    return f;
}
```

g in x10
h in x11
i in x12
j in x13

```
leaf:
    addi   sp,sp,-24    ; make space on stack
    sd     x5,16(sp)    ; save x5
    sd     x6,8(sp)     ; save x6
    sd     x20,0(sp)    ; save x20
    add    x5,x10,x11   ; x5 <- g + h
    add    x6,x12,x13   ; x6 <- i + j
    sub    x20,x5,x6    ; x20 <- x5 + x6
    addi   x10,x20,0    ; x10 <- x20
    ld     x20,0(sp)    ; restore x20
    ld     x6,8(sp)     ; restore x6
    ld     x5,16(sp)    ; restore x5
    addi   sp,sp,24     ; adjust stack
    jalr   x0,0(x1)     ; return
```

# Local Data on the Stack



# Register Saving Problem

- When procedure **yoo()** calls **who()**:
  - **yoo()** is the caller, **who()** is the callee
- Can register be used for temporary storage?

```
yoo:
    . . .
    addi x5, zero, 419
    jal  ra, who
    add  x5, x5, x6
    . . .
    ret
```

```
who:
    . . .
    add x5, x10, x11
    . . .
    ret
```

  - Contents of register **x5** overwritten by **who()**

# Register Saving Conventions

- "Caller saved" registers
  - Caller saves temporary values in its frame before the call
  - Contents of these registers can be modified as a result of procedure call
  - RISC-V: `a0` – `a7`, `t0` – `t6` (x10 – x17, x5 – x7, x28 – x31)

- "Callee saved" registers
  - Callee saves temporary values in its frame before using
  - Callee restores them before returning to caller
  - The contents of these registers are preserved across a procedure call
  - RISC-V: `s0` – `s11` (x8 – x9, x18 – x27)

# Leaf Procedure Example (Revisited)

```
long leaf(long g,
          long h,
          long i,
          long j)
{
    long f;
    f = (g + h) - (i + j);

    return f;
}
```

g in a0
h in a1
i in a2
j in a3

```
leaf:

    add     t0,a0,a1      ; x5 <- g + h
    add     t1,a2,a3      ; x6 <- i + j
    sub     a0,t0,t1      ; x20 <- x5 + x6

    jalr    x0,0(ra)      ; return
```

# Non-Leaf Procedures

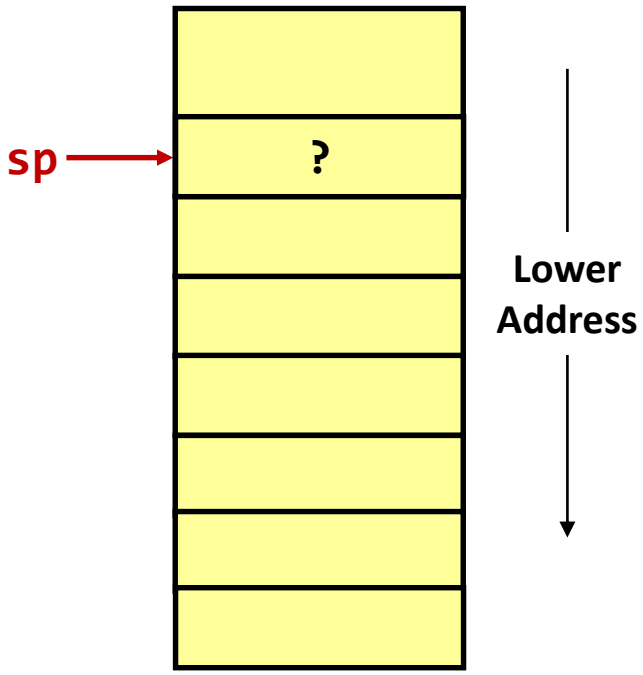
- Procedures that call other procedures
- For nested call, caller needs to save on the stack
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

# Non-Leaf Procedure Example

```
long fact(long n)
{
  if (n < 1)
    return 1;
  else
    return n * fact(n-1);
}
```

```
fact:
     addi    sp, sp, -16    ; make space for 16bytes
     sd      ra, 8(sp)      ; save return address
     sd      a0, 0(sp)      ; save n
     addi    t0, a0, -1     ; t0 <- n - 1
     bge     t0, zero, L1   ; if (t0 >= 0), goto L1
     addi    a0, zero, 1    ; a0 <- 1 (retval)
     addi    sp, sp, 16     ; adjust stack
     jalr    zero, 0(ra)    ; return
L1:
     addi    a0, a0, -1     ; a0 <- n - 1
     jal     ra, fact       ; call fact(n-1)
     addi    t1, a0, 0      ; t1 <- fact(n-1)
     ld      a0, 0(sp)      ; restore n
     ld      ra, 8(sp)      ; restore return address
     addi    sp, sp, 16     ; adjust stack pointer
     mul     a0, a0, t1     ; a0 <- n * t1 (retval)
     jalr    zero, 0(ra)    ; return
```

# Example: fact(2)

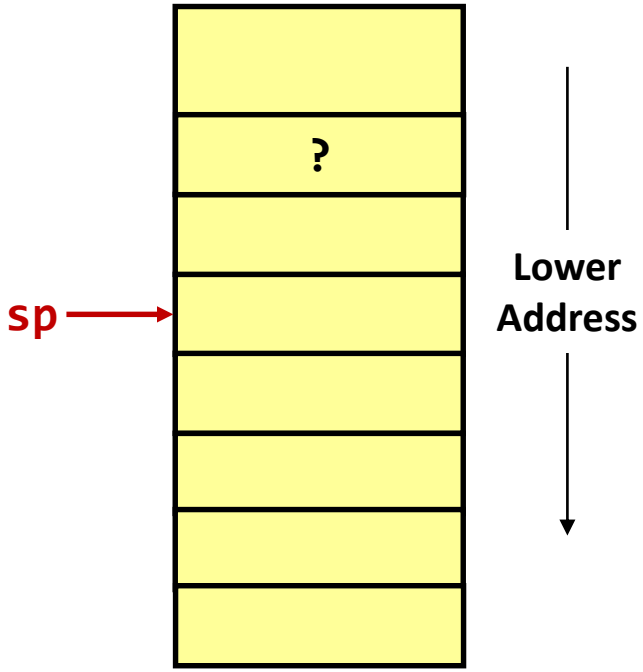



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | ? |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



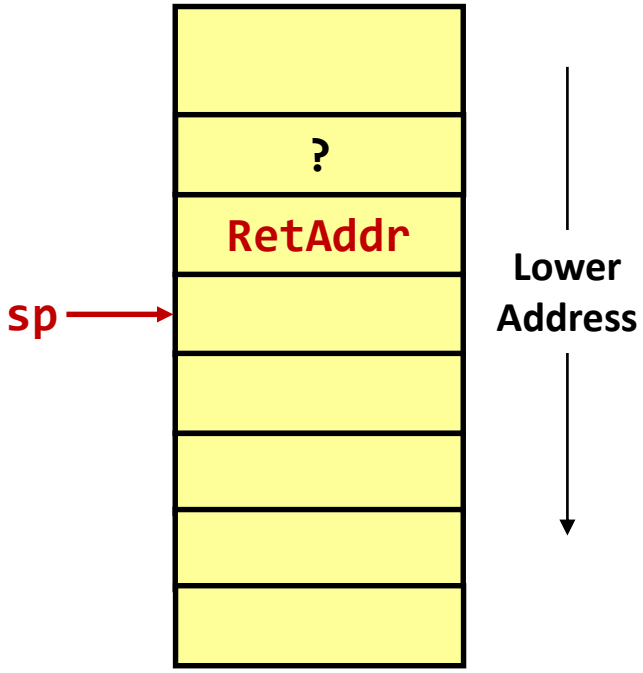

Lower Address

| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | ? |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



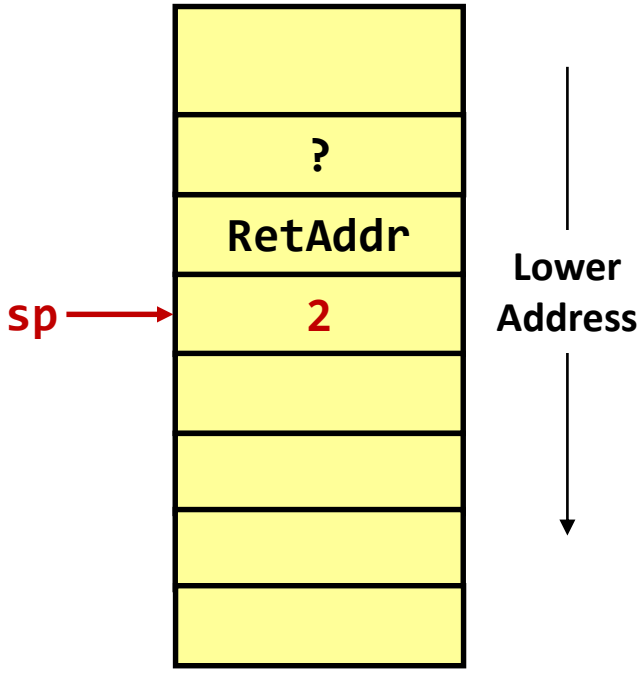

Lower Address

| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | ? |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

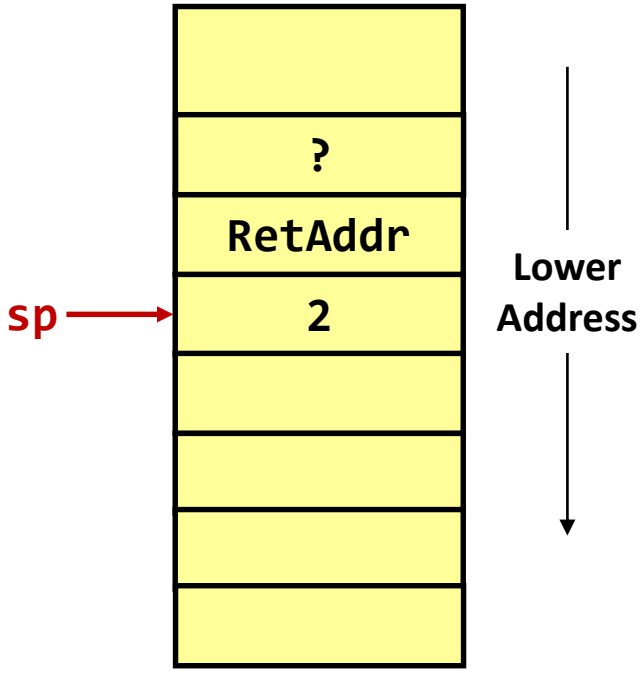



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | ? |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

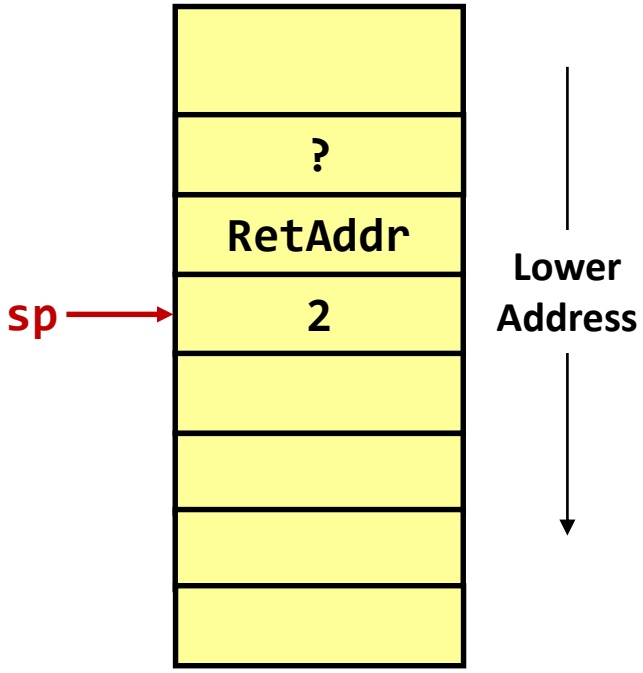



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

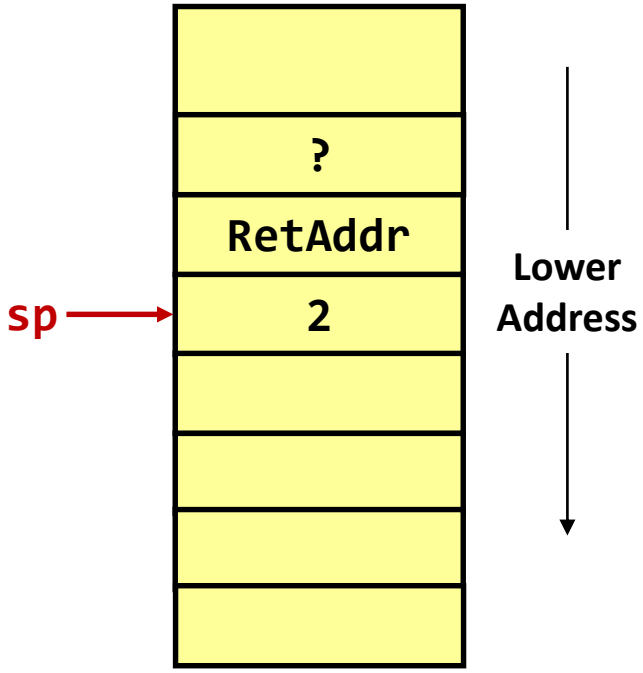



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
     addi    sp, sp, -16
     sd      ra, 8(sp)
     sd      a0, 0(sp)
     addi    t0, a0, -1
     bge     t0, zero, L1
     addi    a0, zero, 1
     addi    sp, sp, 16
     jalr    zero, 0(ra)
L1:
     addi    a0, a0, -1
     jal     ra, fact
A:   addi    t1, a0, 0
     ld      a0, 0(sp)
     ld      ra, 8(sp)
     addi    sp, sp, 16
     mul     a0, a0, t1
     jalr    zero, 0(ra)
```

# Example: fact(2)

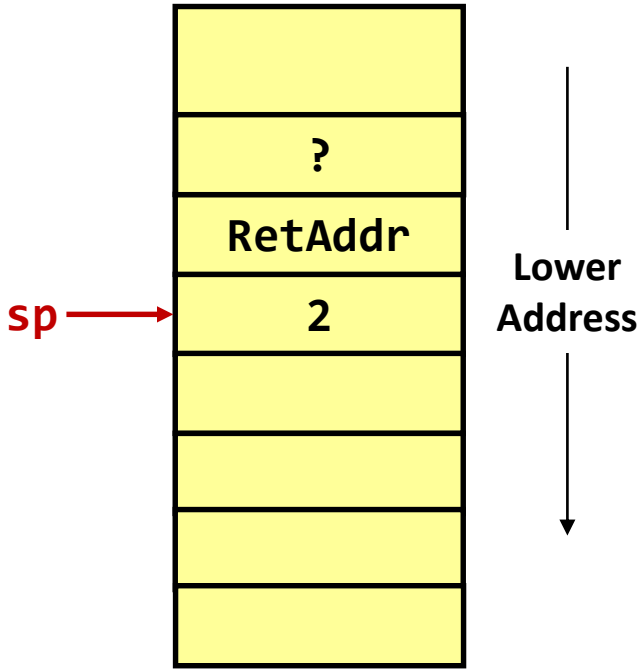



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 1 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

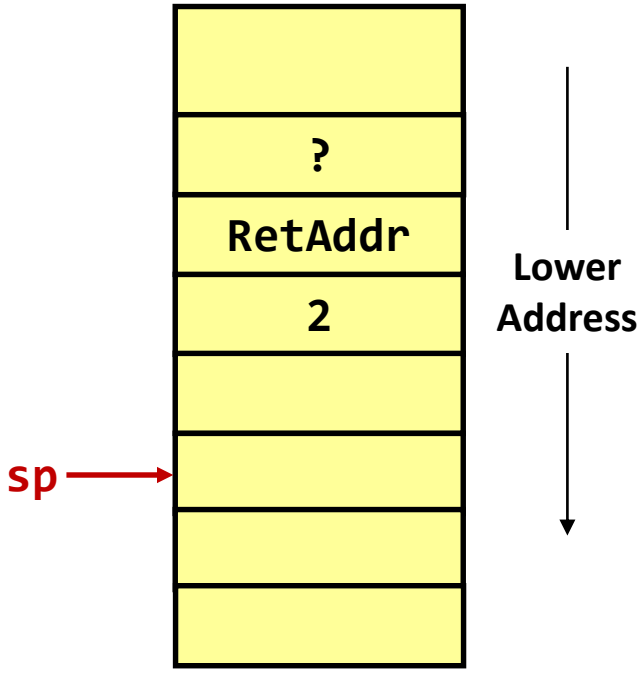



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



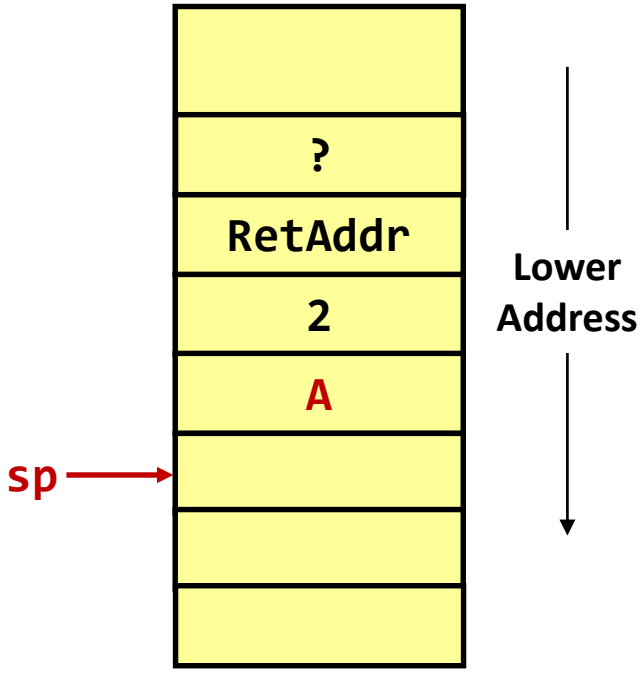

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



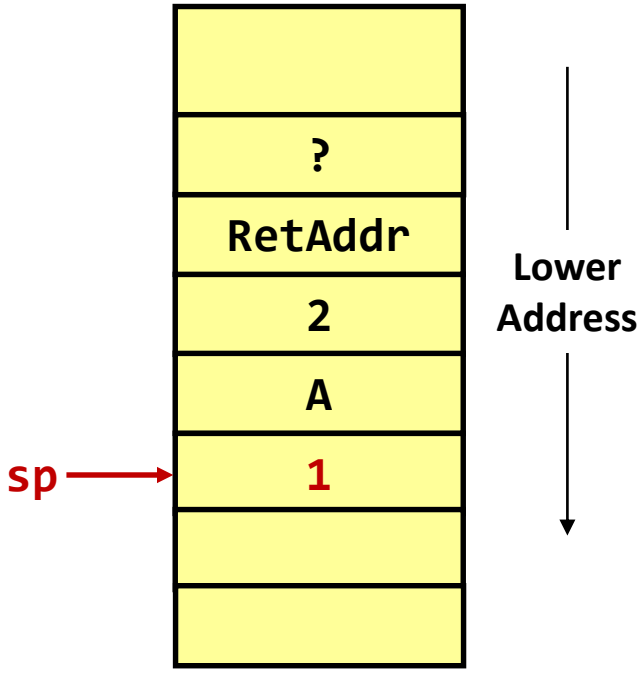

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



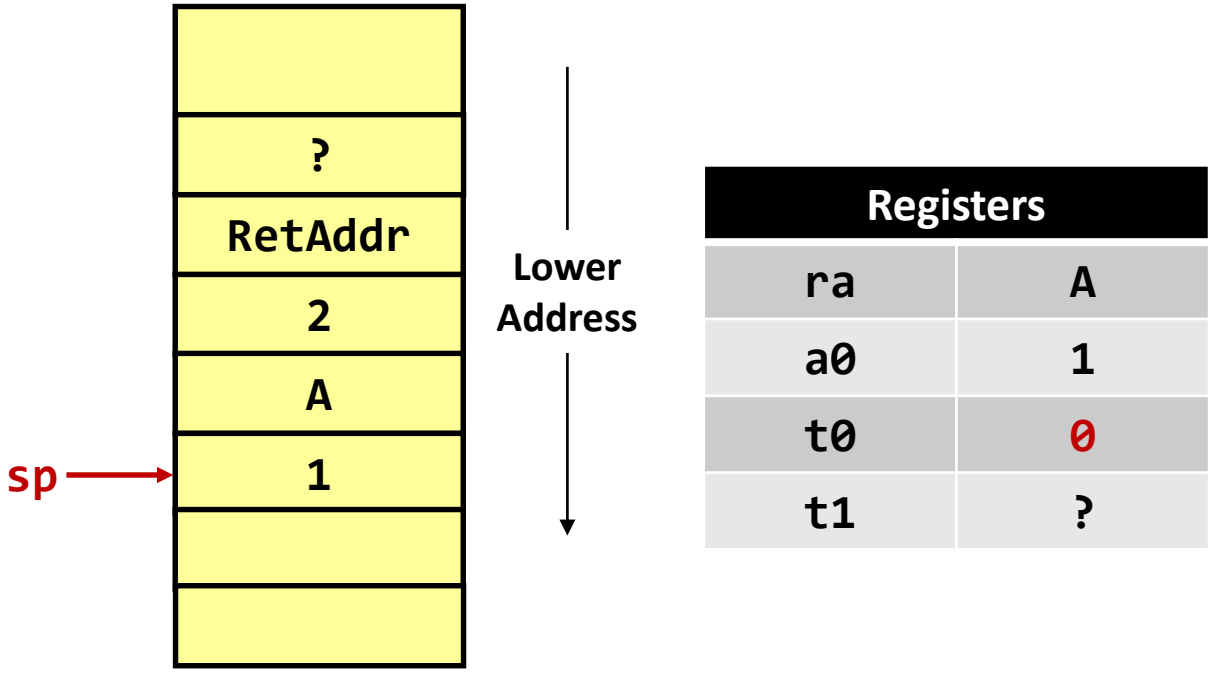

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | 1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

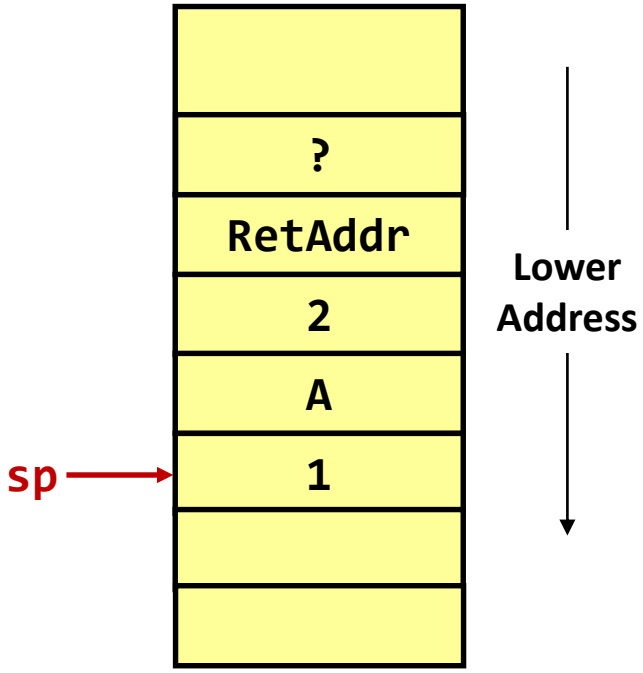
# Example: fact(2)




| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | 0 |
| t1 | ? |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
pc ->   bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

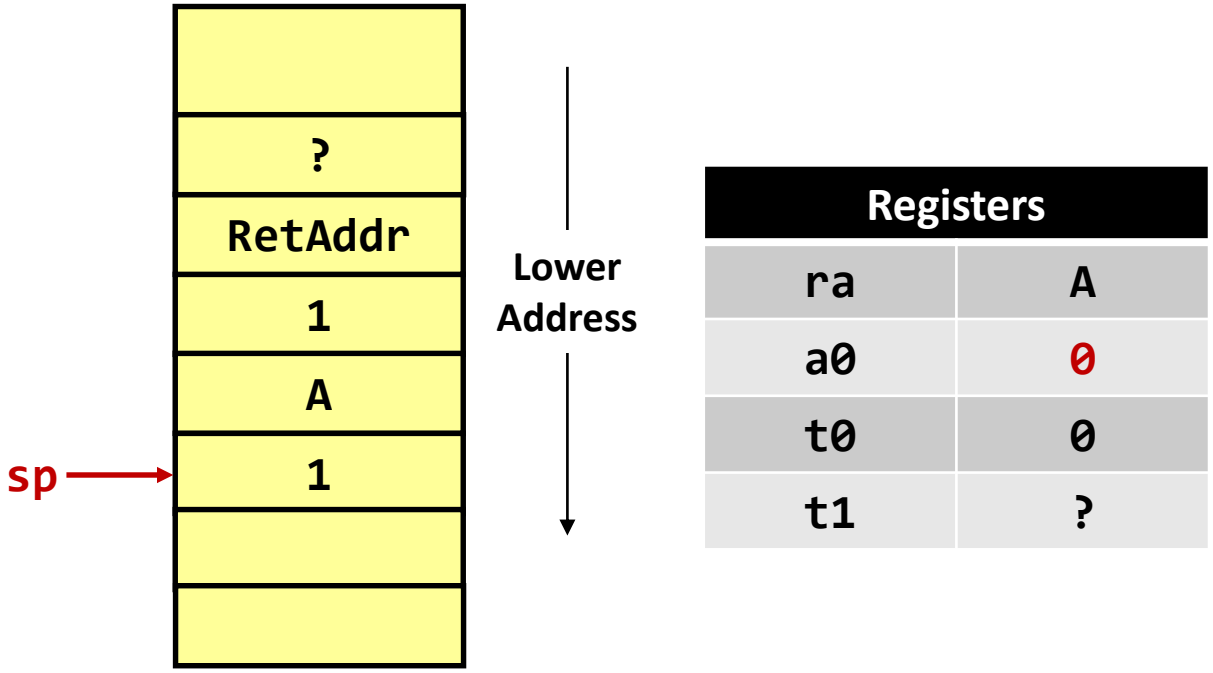



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | 0 |
| t1 | ? |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

pc

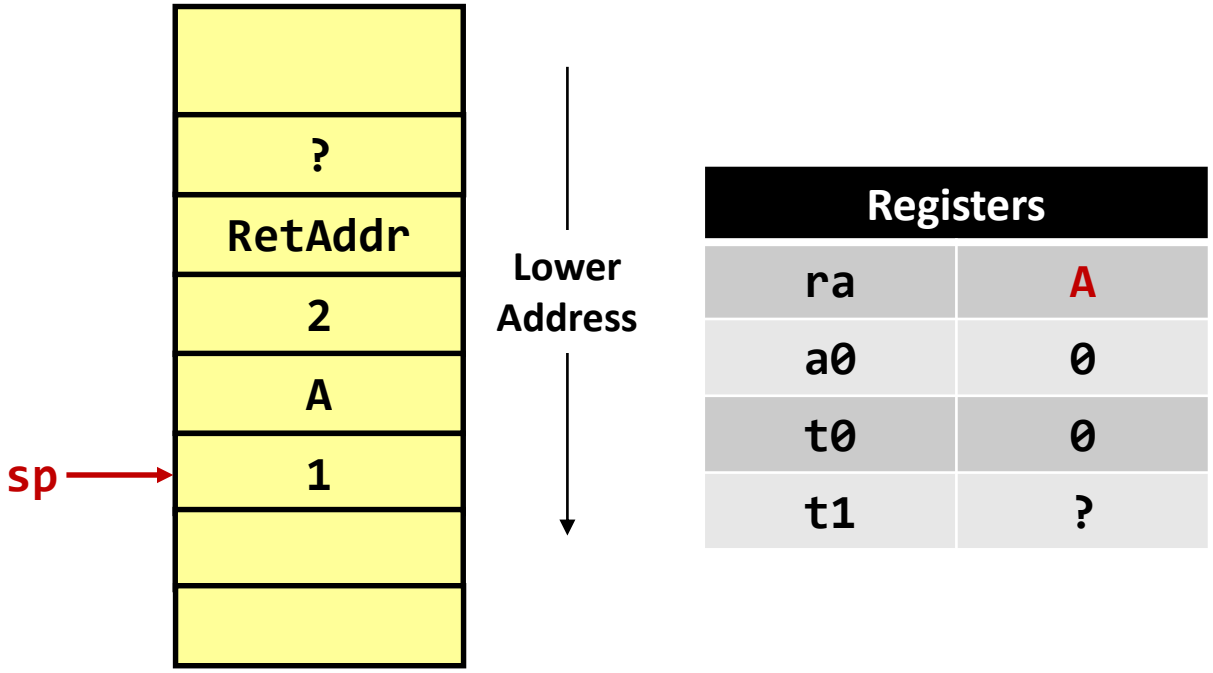
# Example: fact(2)




| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | 0 |
| t1 | ? |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
pc ->   jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

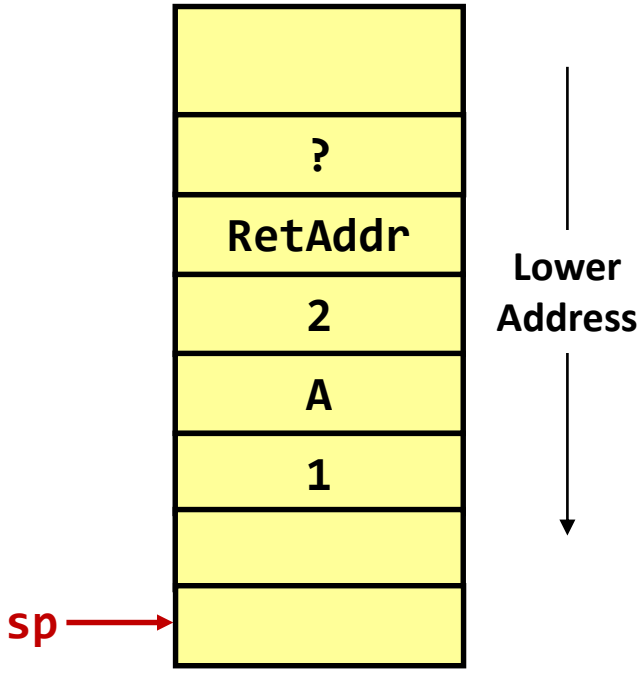



| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | 0 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



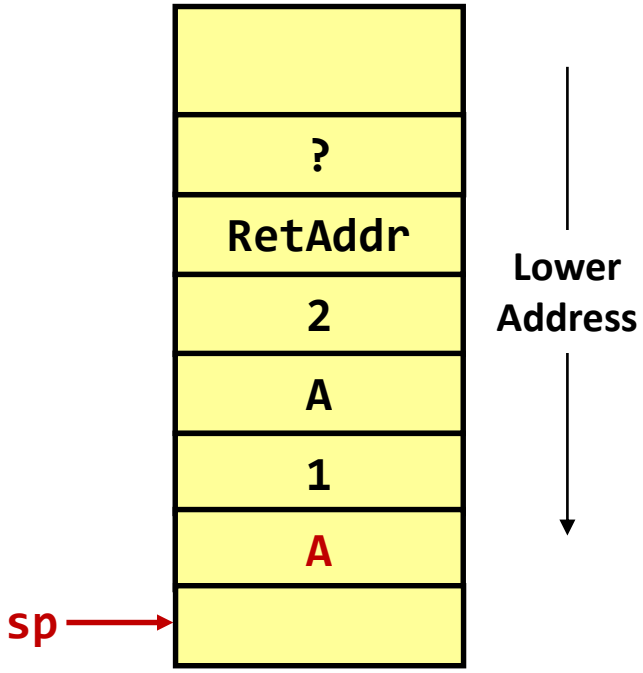

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | 0 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



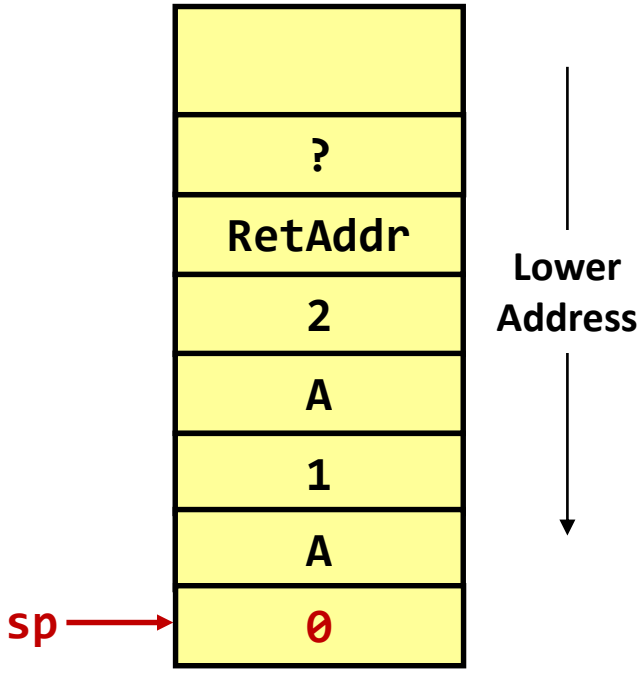

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | 0 |
| t1 | ? |

pc

```
fact:
      addi    sp, sp, -16
      sd      ra, 8(sp)
      sd      a0, 0(sp)
      addi    t0, a0, -1
      bge     t0, zero, L1
      addi    a0, zero, 1
      addi    sp, sp, 16
      jalr    zero, 0(ra)
L1:
      addi    a0, a0, -1
      jal     ra, fact
 A:   addi    t1, a0, 0
      ld      a0, 0(sp)
      ld      ra, 8(sp)
      addi    sp, sp, 16
      mul     a0, a0, t1
      jalr    zero, 0(ra)
```

# Example: fact(2)

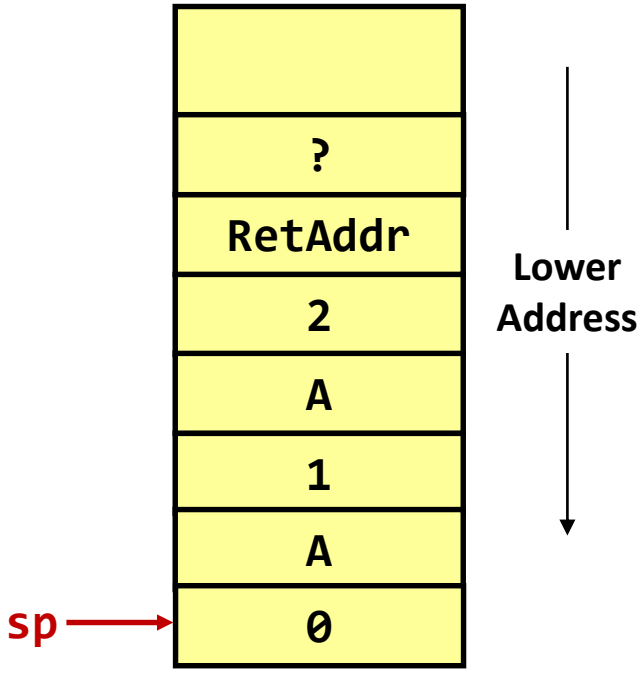



| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | 0 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



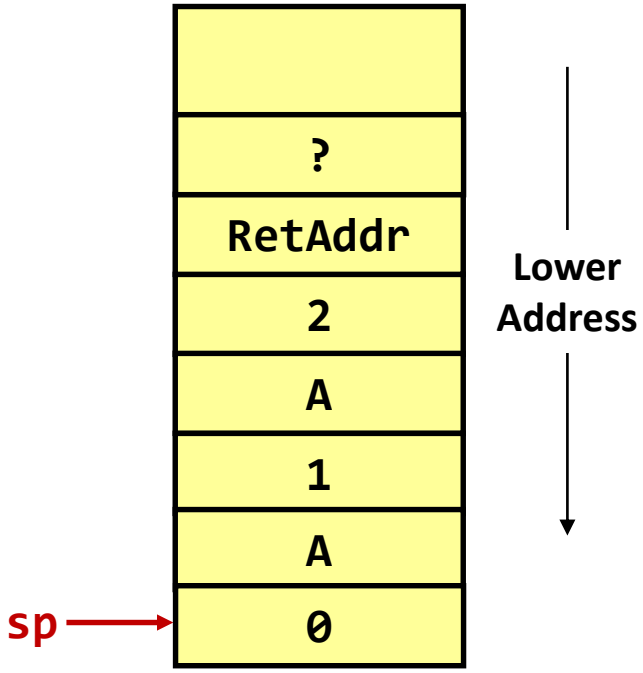

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | -1 |
| t1 | ? |

pc →

```
fact:
     addi    sp, sp, -16
     sd      ra, 8(sp)
     sd      a0, 0(sp)
     addi    t0, a0, -1
     bge     t0, zero, L1
     addi    a0, zero, 1
     addi    sp, sp, 16
     jalr    zero, 0(ra)
L1:
     addi    a0, a0, -1
     jal     ra, fact
A:   addi    t1, a0, 0
     ld      a0, 0(sp)
     ld      ra, 8(sp)
     addi    sp, sp, 16
     mul     a0, a0, t1
     jalr    zero, 0(ra)
```

# Example: fact(2)

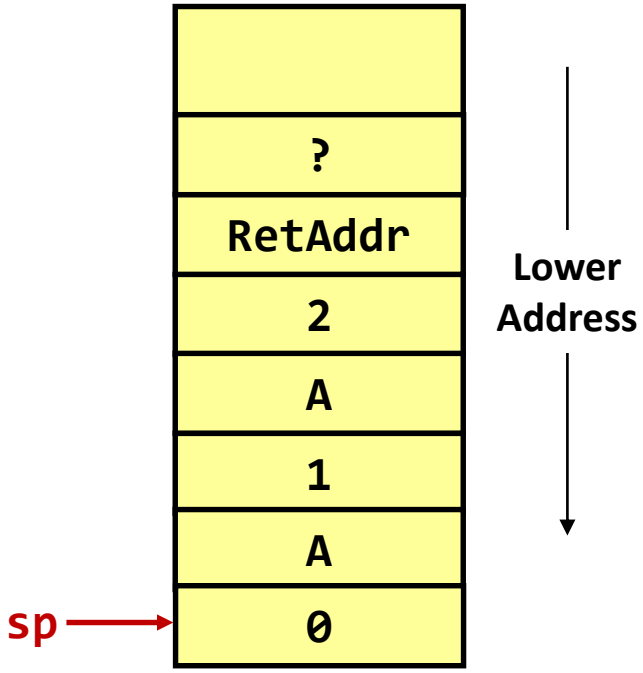



| Registers | |
|---|---|
| ra | A |
| a0 | 0 |
| t0 | -1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



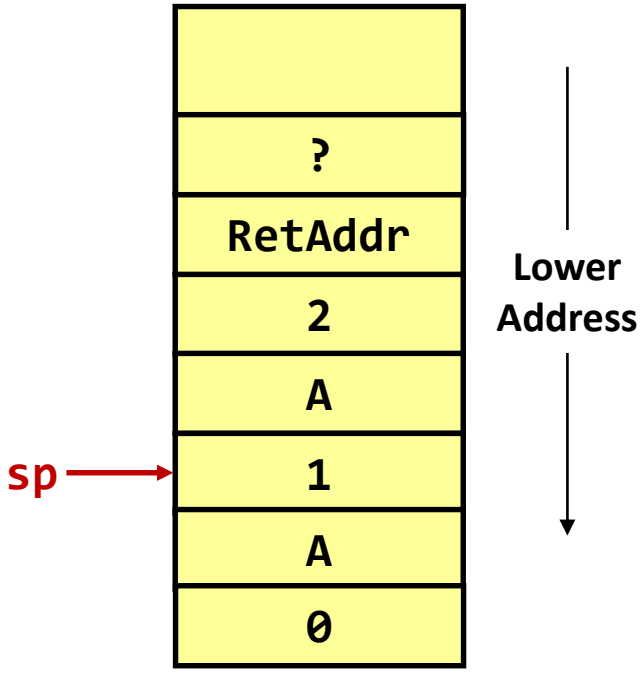

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | ? |

pc →

```
fact:
     addi    sp, sp, -16
     sd      ra, 8(sp)
     sd      a0, 0(sp)
     addi    t0, a0, -1
     bge     t0, zero, L1
     addi    a0, zero, 1
     addi    sp, sp, 16
     jalr    zero, 0(ra)
L1:
     addi    a0, a0, -1
     jal     ra, fact
A:   addi    t1, a0, 0
     ld      a0, 0(sp)
     ld      ra, 8(sp)
     addi    sp, sp, 16
     mul     a0, a0, t1
     jalr    zero, 0(ra)
```

# Example: fact(2)

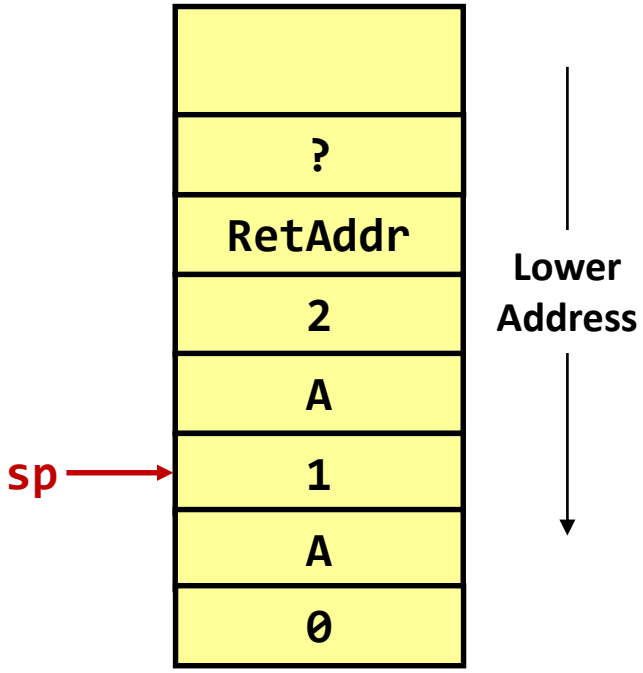



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | ? |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)



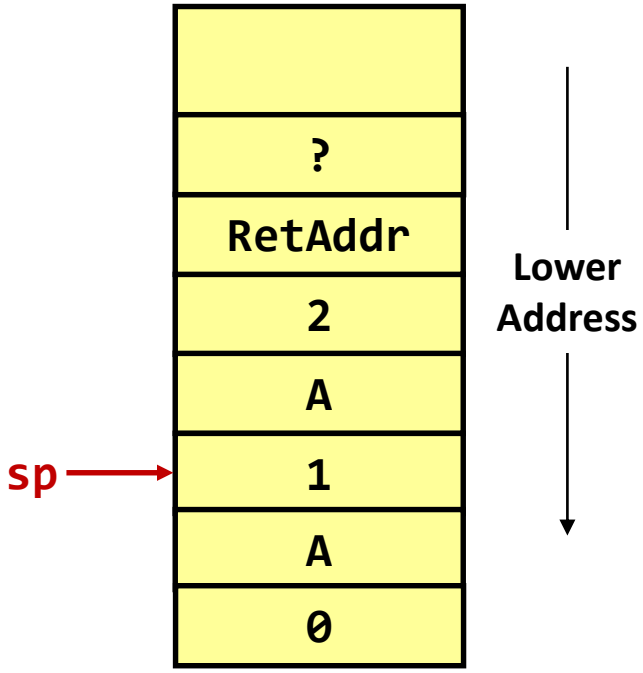

Lower Address

| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | ? |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

pc → A:

# Example: fact(2)

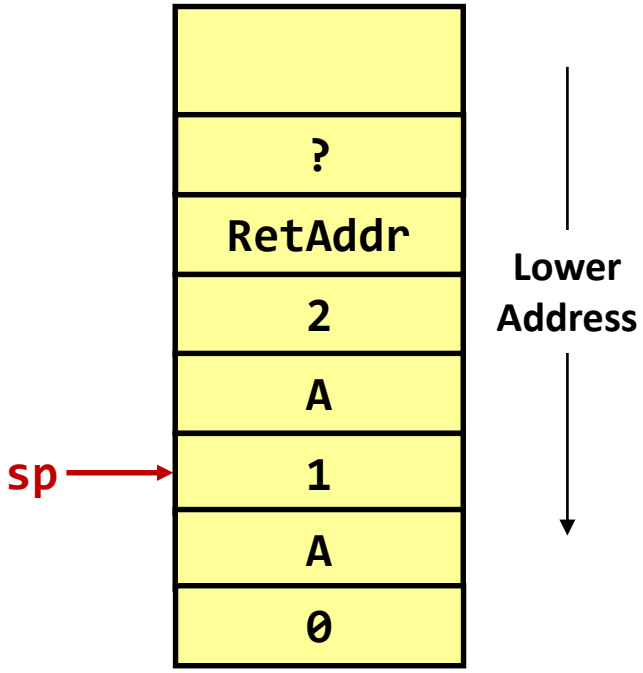



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

pc →

# Example: fact(2)

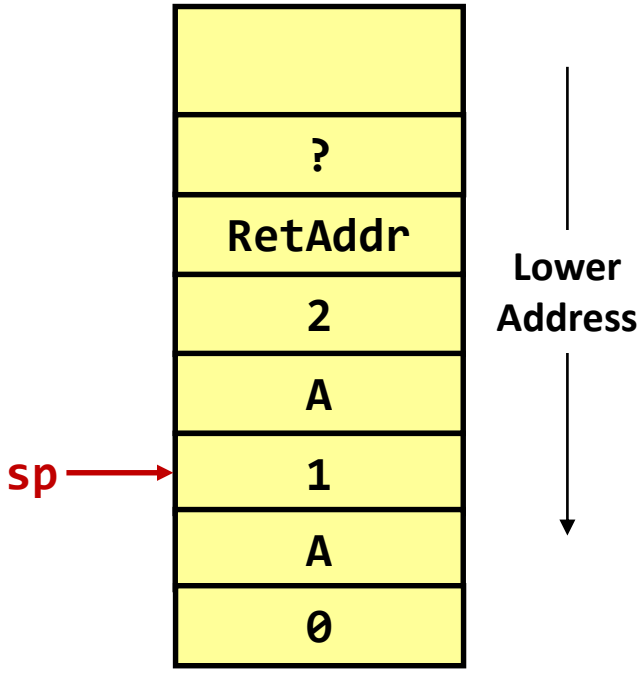



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

pc →

# Example: fact(2)

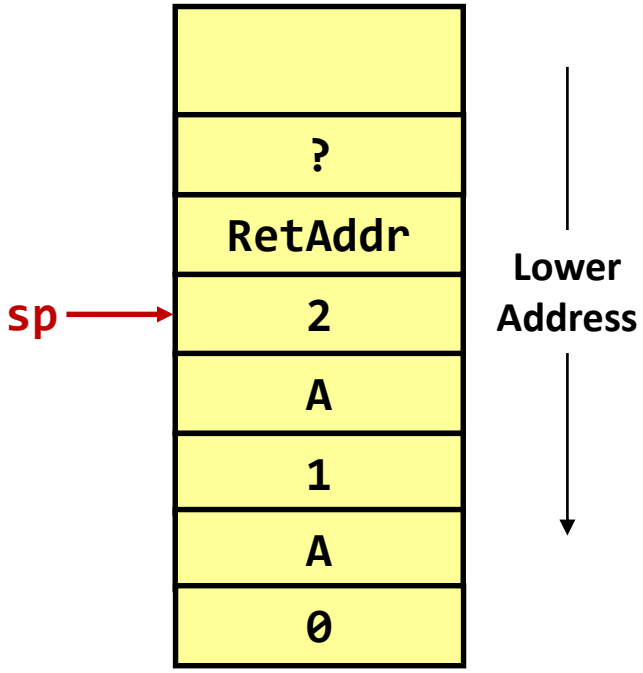



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

pc →

# Example: fact(2)

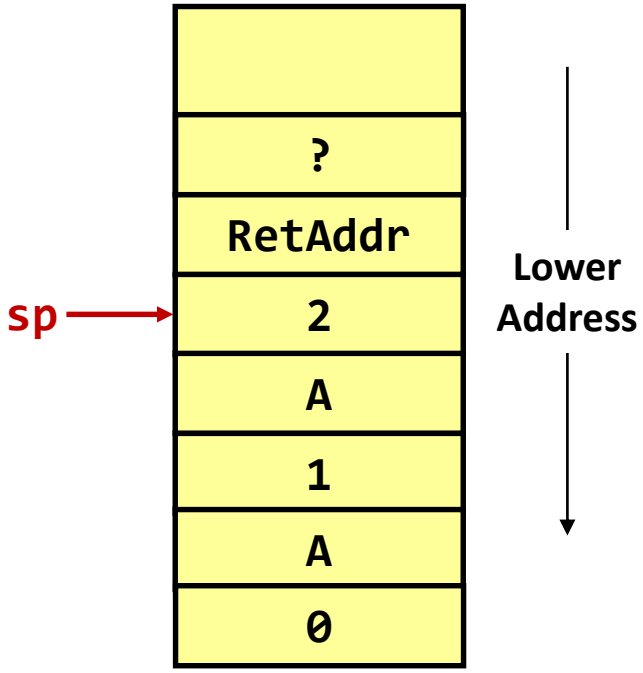



| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
      addi    sp, sp, -16
      sd      ra, 8(sp)
      sd      a0, 0(sp)
      addi    t0, a0, -1
      bge     t0, zero, L1
      addi    a0, zero, 1
      addi    sp, sp, 16
      jalr    zero, 0(ra)
L1:
      addi    a0, a0, -1
      jal     ra, fact
 A:   addi    t1, a0, 0
      ld      a0, 0(sp)
      ld      ra, 8(sp)
      addi    sp, sp, 16
      mul     a0, a0, t1
      jalr    zero, 0(ra)
```

pc →

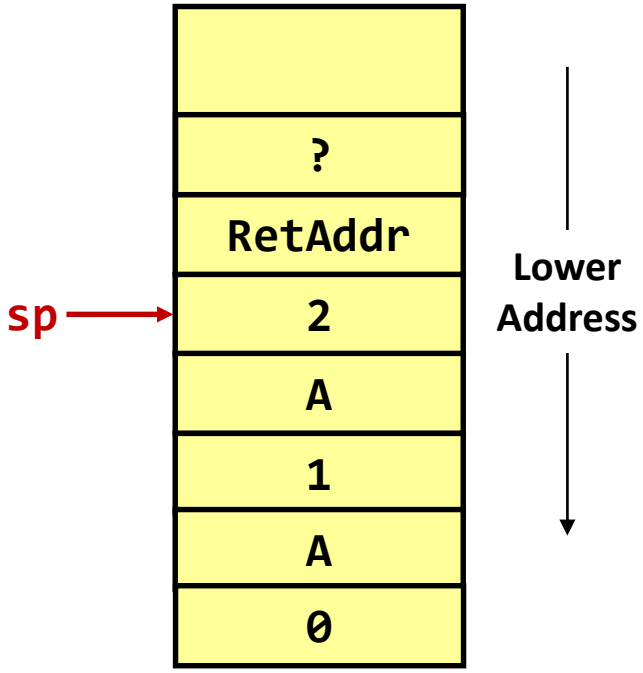
# Example: fact(2)




| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
pc ->   jalr    zero, 0(ra)
```

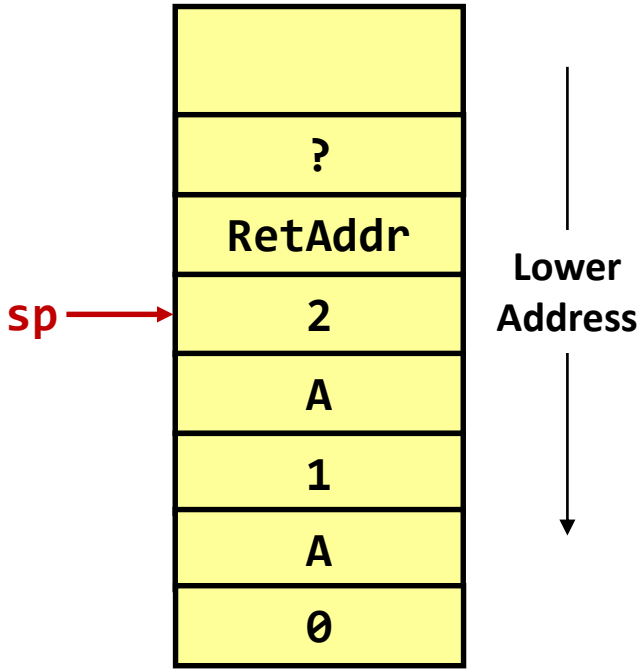
# Example: fact(2)




| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
pc ->   A:
        addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

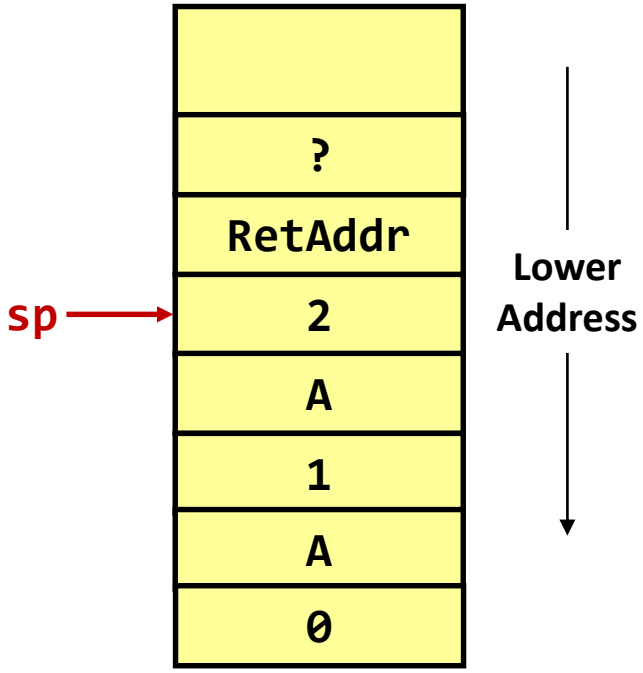
# Example: fact(2)




| Registers | |
|---|---|
| ra | A |
| a0 | 1 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
pc ->   ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

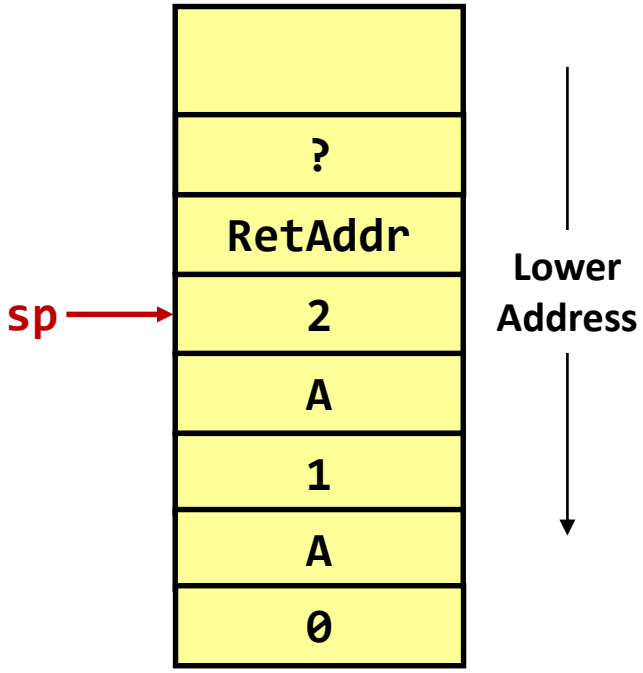
# Example: fact(2)




| Registers | |
|---|---|
| ra | A |
| a0 | 2 |
| t0 | -1 |
| t1 | 1 |

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
pc ->   ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

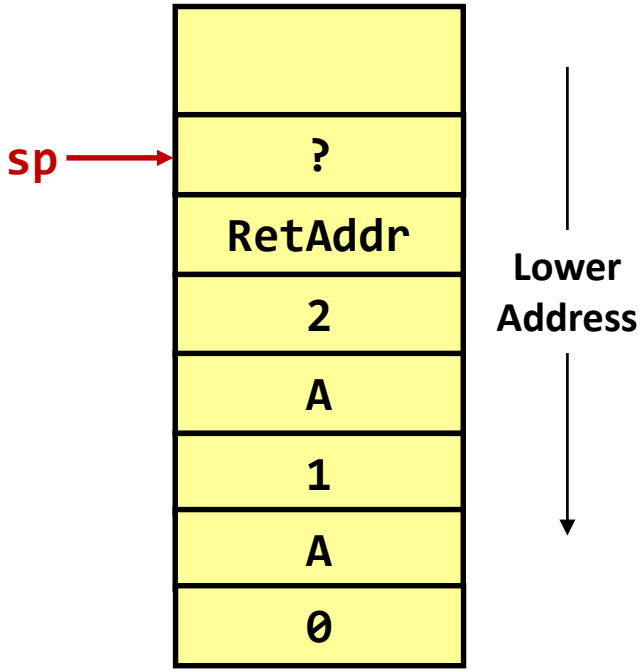



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | -1 |
| t1 | 1 |

pc →

```
fact:
        addi    sp, sp, -16
        sd      ra, 8(sp)
        sd      a0, 0(sp)
        addi    t0, a0, -1
        bge     t0, zero, L1
        addi    a0, zero, 1
        addi    sp, sp, 16
        jalr    zero, 0(ra)
L1:
        addi    a0, a0, -1
        jal     ra, fact
A:      addi    t1, a0, 0
        ld      a0, 0(sp)
        ld      ra, 8(sp)
        addi    sp, sp, 16
        mul     a0, a0, t1
        jalr    zero, 0(ra)
```

# Example: fact(2)

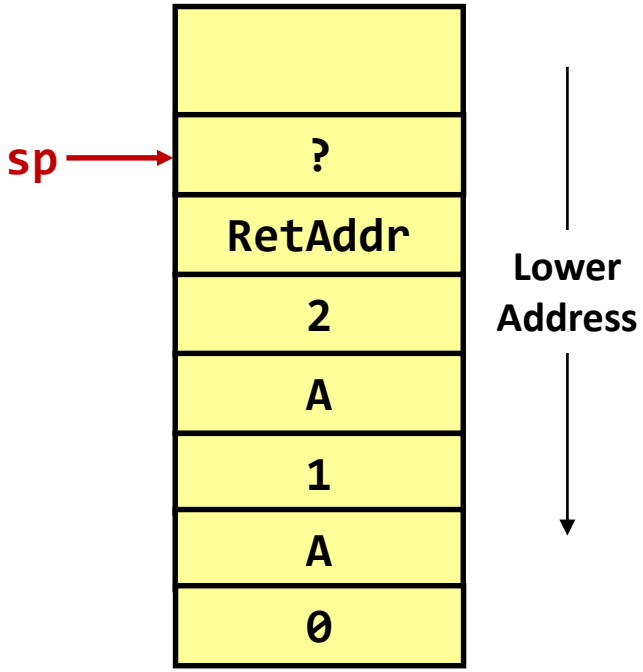



| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | -1 |
| t1 | 1 |

pc →

```
fact:
      addi    sp, sp, -16
      sd      ra, 8(sp)
      sd      a0, 0(sp)
      addi    t0, a0, -1
      bge     t0, zero, L1
      addi    a0, zero, 1
      addi    sp, sp, 16
      jalr    zero, 0(ra)
L1:
      addi    a0, a0, -1
      jal     ra, fact
 A:   addi    t1, a0, 0
      ld      a0, 0(sp)
      ld      ra, 8(sp)
      addi    sp, sp, 16
      mul     a0, a0, t1
      jalr    zero, 0(ra)
```

# Example: fact(2)




| Registers | |
|---|---|
| ra | RetAddr |
| a0 | 2 |
| t0 | -1 |
| t1 | 1 |

```
fact:
      addi    sp, sp, -16
      sd      ra, 8(sp)
      sd      a0, 0(sp)
      addi    t0, a0, -1
      bge     t0, zero, L1
      addi    a0, zero, 1
      addi    sp, sp, 16
      jalr    zero, 0(ra)
L1:
      addi    a0, a0, -1
      jal     ra, fact
 A:   addi    t1, a0, 0
      ld      a0, 0(sp)
      ld      ra, 8(sp)
      addi    sp, sp, 16
      mul     a0, a0, t1
pc    jalr    zero, 0(ra)
```

# Assembler Pseudo-Instructions

| Pseudo-instruction | Base instruction(s) | Meaning |
|---|---|---|
| `li rd, imm` | `addi rd, x0, imm` | Load immediate |
| `la rd, symbol` | `auipc rd, D[31:12]+D[11]`<br>`addi rd, rd, D[11:0]` | Load absolute address where D = `symbol` – `pc` |
| `mv rd, rs` | `addi rd, rs, 0` | Copy register |
| `not rd, rs` | `xori rd, rs, -1` | One's complement |
| `neg rd, rs` | `sub rd, x0, rs` | Two's complement |
| `bgt{u} rs, rt, offset` | `blt{u} rt, rs, offset` | Branch if > (u: unsigned) |
| `ble{u} rs, rt, offset` | `bge{u} rt, rs, offset` | Branch if ≥ (u: unsigned) |
| `b{eq\|ne}z rs, offset` | `b{eq\|ne} rs, x0, offset` | Branch if { = \|≠ } |
| `b{ge\|lt}z rs, offset` | `b{ge\|lt} rs, x0, offset` | Branch if { ≥ \| < } |
| `b{le\|gt}z rs, offset` | `b{ge\|lt} x0, rs, offset` | Branch if { ≤ \| > } |
| `j offset` | `jal x0, offset` | Unconditional jump |
| `call offset` | `jal ra, offset` | Call subroutine (near) |
| `ret` | `jalr x0, 0(ra)` | Return from subroutine |
| `nop` | `addi x0, x0, 0` | No operation |

# Putting It All Together

Chap. 2.13, 2.14

# swap()

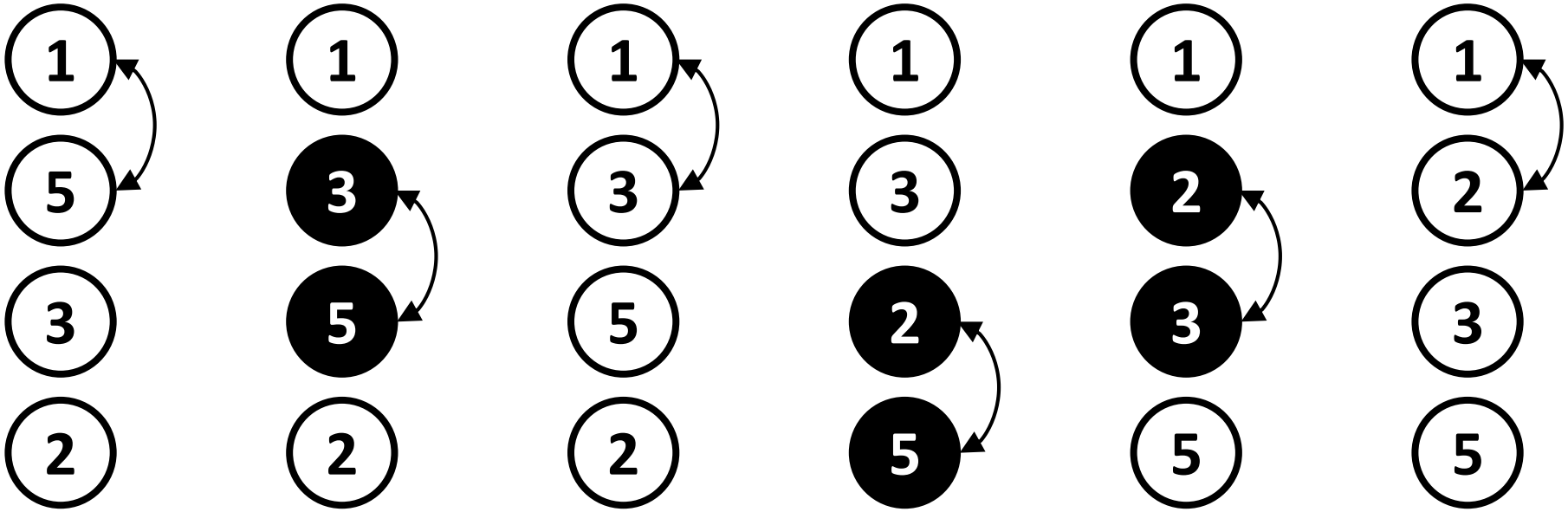
- Swap v[k] and v[k+1]
- Leaf function

```
void swap(long long v[],
          long long k)
{
   long long temp;

   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
   ; v is in a0
   ; k is in a1

swap:
   slli   t1, a1, 3    ; t1 = k * 8
   add    t1, a0, t1   ; v = v + t1
   ld     t0, 0(t1)    ; t0 = v[k]
   ld     t2, 8(t1)    ; t2 = v[k+1]
   sd     t2, 0(t1)    ; v[k] = t2
   sd     t0, 8(t1)    ; v[k+1] = t0
   ret
```

# sort()

```
void sort(long long v[], size_t n) {
    size_t i, j;
    for (i = 1; i < n; i++)
        for (j = i - 1; j >= 0 && v[j] > v[j+1]; j--) {
            swap(v, j);
}
```



# sort()  1/3

```
sort:
    addi    sp, sp, -40             ; make space for 5 registers
    sd      ra, 32(sp)              ; save ra (return address)
    sd      s6, 24(sp)              ; save s6 (will be used for n)
    sd      s5, 16(sp)              ; save s5 (will be used for v)
    sd      s4, 8(sp)               ; save s4 (will be used for j)
    sd      s3, 0(sp)               ; save s3 (will be used for i)

    mv      s5, a0                  ; s5 = v
    mv      s6, a1                  ; s6 = n
    li      s3, 0                   ; s3 = 0 (i)

Outer:
    bge     s3, s6, OuterExit       ; if (i >= n) goto OuterExit
    addi    s4, s3, -1              ; s4 = i - 1 (j)
```

# sort() 2/3

```
Inner:
    blt     s4, zero, InnerExit      ; if (j < 0) goto InnerExit
    slli    t0, s4, 3                ; t0 = j * 8
    add     t0, s5, t0               ; t0 = v + j * 8
    ld      t1, 0(t0)                ; t1 = v[j]
    ld      t2, 8(t0)                ; t2 = v[j+1]
    ble     t1, t2, InnerExit        ; if (v[j] <= v[j+1]) goto InnerExit

    mv      a0, s5                   ; a0 = v
    mv      a1, s4                   ; a1 = j
    jal     ra, swap                 ; call swap

    addi    s4, s4, -1               ; j = j - 1
    j       Inner
```

# sort()  3/3

```
InnerExit:
    addi   s3, s3, 1                ; i = i + 1
    j      Outer                    ; goto Outer

OuterExit:
    ld     s3, 0(sp)                ; restore s3
    ld     s4, 8(sp)                ; restore s4
    ld     s5, 16(sp)               ; restore s5
    ld     s6, 24(sp)               ; restore s6
    ld     ra, 32(sp)               ; restore ra
    addi   sp, sp, 40               ; adjust stack
    ret
```

# Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address

- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity

# Example: Clearing an Array

```
void clear1(long long A[],
            size_t n) {
  size_t i;
  for (i = 0; i < n; i++)
    A[i] = 0;
}
```

```
clear1:
  li    t0, 0          ; i = 0
Loop:
  slli  t1, t0, 3      ; t1 = i * 8
  add   t2, a0, t1     ; t2 = A + i*8
  sd    zero, 0(t2)    ; A[i] = 0
  addi  t0, t0, 1      ; i++
  blt   t0, a1, Loop   ; if (i < n)
                       ;   goto Loop
```

```
void clear2(long long *A,
            long long n) {
  long long *p;
  for (p = A; p < A + n; p++)
    *p = 0;
}
```

```
clear2:
  mv    t0, a0         ; p = A
  slli  t1, a1, 3      ; t1 = n * 8
  add   t2, a0, t1     ; t2 = A + n*8
Loop:
  sd    zero, 0(t0)    ; *p = 0
  addi  t0, t0, 8      ; p++
  bltu  t0, t2, Loop   ; if (p<A+n*8)
                       ;   goto Loop
```

# Comparison: Arrays vs. Pointers

- Multiply "strength reduced" to shift

- Array version requires shift to be inside loop
  - Part of index calculation for incremented *i*

- Compiler can achieve same effect as manual use of pointers
  - Compiling `clear1()` with `-O1` generates pointer-based code
  - Induction variable elimination
  - Better to make program clearer and safer

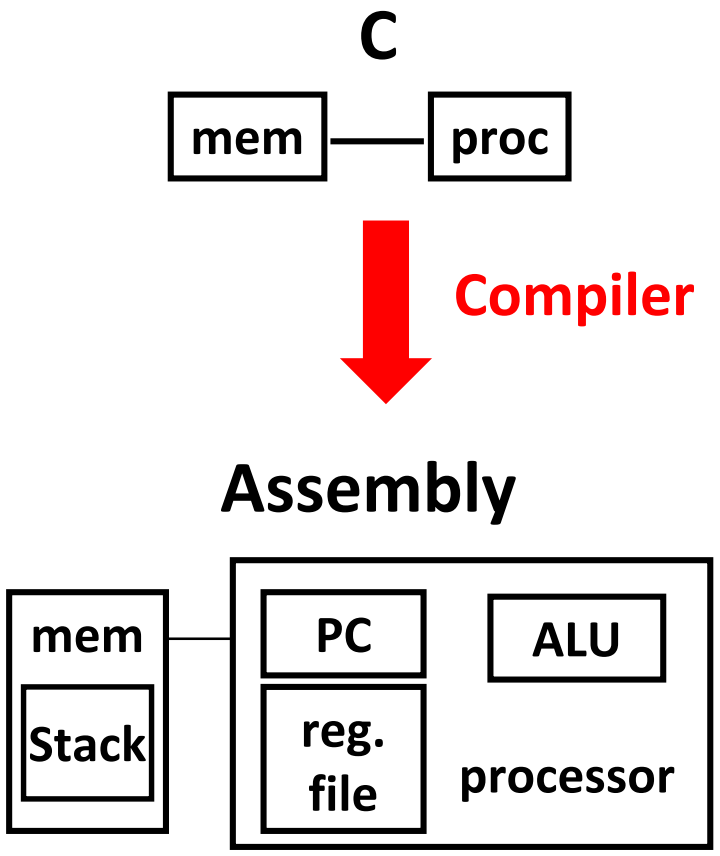
# Machine-level Programming

- Assembly code is textual form of binary object code

- Low-level representation of program
  - Explicit manipulation of registers
  - Simple and explicit instructions
  - Minimal concept of data types
  - Many C control constructs must be implemented with multiple instructions

# Summary

## Machine Models




## Data

1) char
2) int, float
3) double
4) struct, array
5) pointer

1) byte
2) 2-byte halfword
3) 4-byte word
4) 8-byte double word
5) contiguous byte allocation
6) address of initial byte

## Control

1) loops
2) conditionals
3) switch
4) Proc. call
5) Proc. return

1) branch/jump
2) call
3) ret