Chapter 6: Architecture

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`.

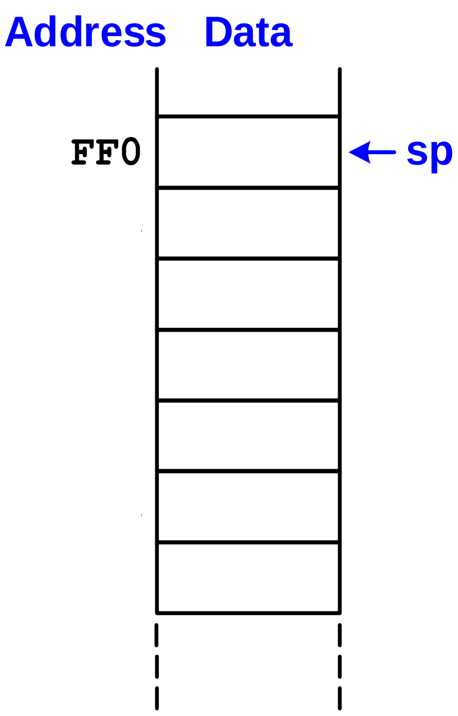
Recursive Functions

```
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
```

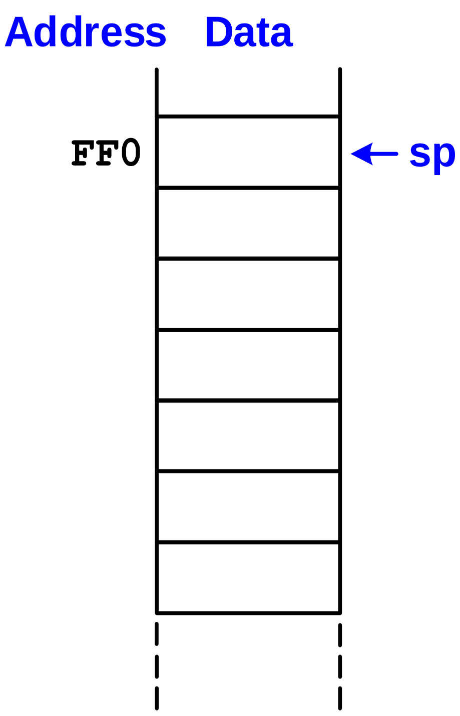
PC+4 = 0x8528 when factorial is called recursively.

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, imm`$_{20:0}$)
    - **rd** = PC+4; PC = PC + **imm**
  - jump and link register (`jalr rd, rs, 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
    - $\texttt{rd = PC + \{imm}_{31:12}\texttt{, 12'b0\}}$
- Pseudoinstruction: $\texttt{call imm}_{31:0}$
  - Behaves like `jal imm`, but allows 32-bit immediate offset

$\texttt{auipc ra, imm}_{31:12}$

$\texttt{jalr ra, ra, imm}_{11:0}$

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` `addi s8, s8, 0xDEF`
`bgt s1, t3, L3`	`blt t3, s1, L3`
`bgez t2, L7`	`bge t2, zero, L7`
`call L1`	auipc ra, $imm_{31:12}$ jalr ra, ra, $imm_{11:0}$
`ret`	`jalr zero, ra, 0`

See Appendix B for more pseudoinstructions.