

6.004 Tutorial Problems

L03 – Procedures and Stacks

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---

RISC-V Calling Conventions:

- Caller places arguments in registers a0–a7
- Caller transfers control to callee using `jal` (jump-and-link) to capture the return address in register ra. The following two instructions are equivalent:
 - `jal ra, label: R[ra] <= pc + 4; pc <= label`
 - `jal label` (pseudoinstruction for the above)
- Callee runs, and places results in registers a0 and a1
- Callee transfers control to caller using `jr` (jump-register) instruction. The following instructions are equivalent:
 - `jalr x0, 0(ra): pc <= R[ra]`
 - `jr ra` (pseudoinstruction for the above)
 - `ret` (pseudoinstruction for the above)

Push register x*i* onto stack

```
addi sp, sp, -4
sw xi, 0(sp)
```

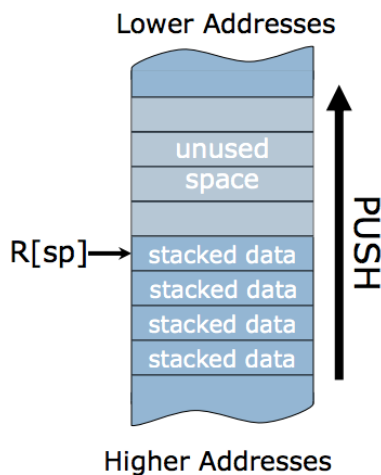
Pop value at top of stack into register x*i*

```
lw xi, 0(sp)
addi sp, sp, 4
```

Assume 0(sp) holds valid data.

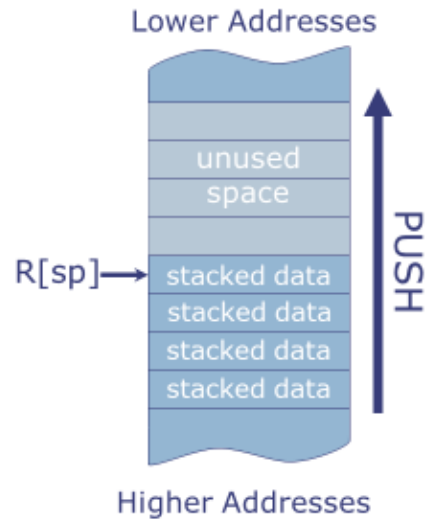
Stack discipline: can put anything on the stack, but leave stack the way you found it

- Always save **s** registers before using them
- Save **a** and **t** registers if you will need their value after procedure call returns.
- Always save **ra** if making nested procedure calls.



RISC-V Stack

- Stack is in memory → need a register to point to it
 - In RISC-V, stack pointer `sp` is `x2`
- Stack grows down from higher to lower addresses
 - Push decreases `sp`
 - Pop increases `sp`
- `sp` points to top of stack (last pushed element)
- Discipline: Can use stack *at any time*, but leave it as you found it!



February 12, 2020

MIT 6.004 Spring 2020

L03-19

Using the stack

Sample entry sequence

```
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
```

Corresponding Exit sequence

```
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, 8
```

February 12, 2020

MIT 6.004 Spring 2020

L03-20

Note: A small subset of essential problems are marked with a red star (★). We especially encourage you to try these out before recitation.

Problem 1.

Integer arrays **season1** and **season2** contain points Ben Bitdiddle had scored at each game over two seasons during his time at MIT Intramural Basketball Team. Please write a RISC-V assembly program which counts the number of games he scored more than 20 points. An equivalent Python program is given below. Note that the base addresses for arrays **season1** and **season2** along with their size are passed down to function **greaterthan20**.

```
import numpy as np

def main():
    season1 = np.array([18, 28, 19, 33, 25, 11, 20])
    season2 = np.array([30, 12, 13, 33, 37, 19, 22])
    result = greaterthan20(season1, season2, len(season1))
    print(result)

def greaterthan20(a, b, size):
    count = 0
    for i in range (size):
        if a[i] > 20:
            count += 1
        if b[i] > 20:
            count += 1
    return count
```

move a0 to
ret

// Beginning of your assembly code
greaterthan20:

```
li t0, 0 // t0 ← count
li t1, 0 // t1 ← index
li t2, 20

loop:
    j compare
loop:
    slli t3, t0, 2 // t3 = i * 4
    // t4: address of a[i]
```

check a:

```
add t4, a0, t3 // a0: base of array a.
lw t5, 0(t4) // the value of a[i]
ble t5, t2, checkb
addi t0, t0, 1
```

check b:

```
add t5, a1, t3 // a1: base of array b.
lw t6, 0(t5) // the value of b[i]
ble t6, t2, compare // 要实现的 compare 前 i 的 increment.
addi t0, t0, 1
```

end of

compare

```
addi t1, t1, 1 // i++
```

```
ble a2, t1, loop.
```

Problem 2.

For the following Python functions, does the corresponding RISC-V assembly obey the RISC-V calling conventions? If not, rewrite the function so that it does obey the calling conventions.

(A) `def function_A(a, b): ★`
 `some_other_function()`
 `return a + b`

```
function_A:
    addi sp, sp, -8 // error memory allocate.
    sw a0, 8(sp)
    sw a1, 4(sp)
    sw ra, 0(sp)
    jal some_other_function
    lw a0, 8(sp)
    lw a1, 4(sp)
    add a0, a0, a1
    lw ra, 0(sp)
    addi sp, sp, 8
    ret
```

yes ... no

(B) `def function_B(a, b):`
 `i = foo((a + b)^(a - b))`
 `return (i + 1)^i`

```
function_B:
    addi sp, sp, -4
    sw ra, 0(sp)
    add t0, a0, a1 // t0 = a+b
    sub a0, a0, a1 // a0 = a-b
    xor a0, t0, a0 // xor process.
    jal foo
    addi t0, a0, 1
    xor a0, t0, a0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

yes ... no

(C) def function_C(x): ★

```
foo(1, x)
bar(2, x)
baz(3, x)
return 0
```

function_C:

```
addi sp, sp, -8
sw ra, 0(sp)
mv a1, a0 → sw a1 4(sp)
li a0, 1
jal foo
li a0, 2 → lw a1 4(sp)
jal bar
li a0, 3 → lw a1 4(sp)
jal baz
li a0, 0
lw ra, 0(sp)
addi sp, sp, 8
ret
```

prevent nested procedures overwrite
X during executing of nested procedures

yes ... no

(D) def function_D(x, y):

```
i = foo(1, 2)
return i + x + y
```

function_D:

```
addi sp, sp, -12
sw ra, 0(sp)
mv s0, a0 → sw s0 4(sp)
mv s1, a1 → sw s1 8(sp)
li a0, 1
li a1, 2
jal foo
add a0, a0, s0
add a0, a0, s1
lw ra, 0(sp) → lw s1 8(sp)
addi sp, sp, 4 → lw s0 4(sp)
ret
```

Problem 3. ★

Our RISC-V processor does not have a multiply instruction, so we have to do multiplications in software. The Python code below shows a recursive implementation of multiplication by repeated addition of unsigned integers. Ben Bitdiddle has written and hand-compiled this function into the assembly code given below, but the code is not behaving as expected. Find the bugs in Ben's assembly code and write a correct version.

Python for unsigned multiplication

```
# x, y are unsigned integers
def mul(x, y):
    if x == 0:
        return 0
    else:
        lowbit = x & 1
        p = y if lowbit else 0
        return p + (mul(x >> 1, y) << 1)
```

Buggy assembly code

```
mul:
    addi sp, sp, -8
    sw s0, 0(sp)
    sw ra, 4(sp)
    beqz a0, mul_done
    andi s0, a0, 1 // lowbit in s0
    mv t0, zero // p in t0
    beqz s0, lowbit_zero
    mv t0, a0
lowbit_zero:
    slli a0, a0, 1
    jal mul
    srli a0, a0, 1
    add a0, t0, a0
    lw s0, 4(sp)
    lw ra, 0(sp)
    addi sp, sp, 8
mul_done:
    ret
```