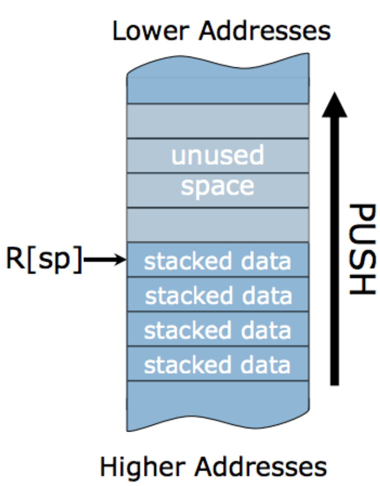
6.004 Tutorial Problems
L03 – Procedures and Stacks

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 `xi` onto stack

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

Pop value at top of stack into register `xi`

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

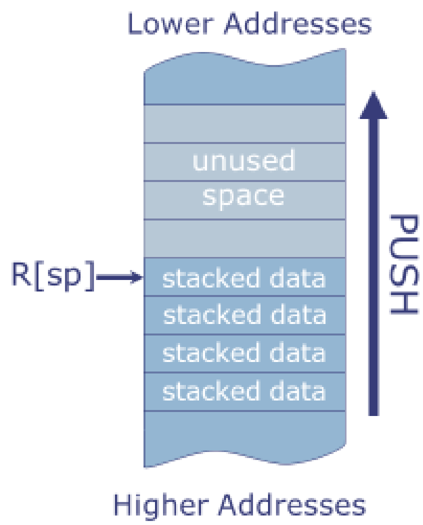
Assume `0(sp)` holds valid data.

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
```

```
// Beginning of your assembly code
greaterthan20:
        li t0, 0 // t0 ← count
        li t1, 0 // t1 ← index
        li t2, 20
loop:
        // We compile the for loop into something more like a while loop of the form:
        // while i < size:
        //     (body of loop goes here)
        //     i = i + 1

        ble a2, t1, endloop // if a2 (= size) <= t1 (= i), jump to endloop
            // (that is, stop the loop if i < size is false)
        slli t3, t1, 2 // t3 ← 4 × index
checka:
        add t4, a0, t3 // t4 ← a0 (= base of array a) + t3 = address of a[i]
        lw t5, 0(t4) // t5 = value at address t4 = value of a[i]
        ble t5, t2, checkb // if a[i] <= 20, then skip to checking b[i]
        addi t0, t0, 1 // increment count
checkb:
        add t4, a1, t3 // t4 ← a1 (= base of array b) + t3 = address of b[i]
        lw t5, 0(t4) // t5 = contents of address t4 = value of b[i]
        ble t5, t2, endcompare // if b[i] <= 20, then go to endcompare
        addi t0, t0, 1 // increment count
endcompare:
```

```
    addi t1, t1, 1 // increment index i
    j loop // restart loop from the condition check
endloop:
    mv a0, t0 // move count to a0, the register for holding the return value
    ret
```

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
    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

`addi sp, sp, -8` only allocates two words on the stack, `0(sp)` (i.e. `sp + 0`) and `4(sp)` (i.e. `sp + 4`). The address `8(sp)` belongs to the caller because it was greater than or equal to the value of `sp` in the caller, and using it violates calling convention. We can fix it just by replacing the `-8` with `-12`, and the `8` with `12` at the end:

```
function_A:
    addi sp, sp, -12
    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, 12
    ret
```

Everything else is correct. We save `a0` and `a1` onto the stack and restore them after calling `some_other_function`, since that function is allowed to overwrite them. Then we add them and put the result in `a0`, where it is returned to the caller.

(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
    sub a0, a0, a1
    xor a0, t0, a0
    jal foo
    addi t0, a0, 1
    xor a0, t0, a0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

yes ... **no**

Nothing is wrong here. `addi sp, sp, -4` allocates the address `0(sp)`, which we use to store `ra` and restore it so that it's OK when `ra` is overwritten by calling `foo`; and `sp` is also restored to the old value at the end. All used registers `a0`, `a1`, `t0` are caller-saved registers, so we are allowed to modify them without restoring them, but we also don't assume that `foo` preserves any of those registers when called, as we only need its return value, which appears in `a0`.

(C) ★
```
def function_C(x):
    foo(1, x)
    bar(2, x)
    baz(3, x)
    return 0

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

yes ... **no**

The code assumes that its argument `x` will stay in register `a1` as it calls functions `foo` and `bar`, because it needs to pass the same argument to `bar` and `baz`. However, those functions are allowed to overwrite `a1` by calling convention. Instead, we must store `x` in the stack and restore it when we need it again. (Note that we only need to store `x` once, and we can load it twice; that part of the stack belongs to this function, so neither `foo` nor `bar` is allowed to modify it. Also, we do *not* need to restore `a1` after returning from `baz`, because we don't need it any more and we aren't required to preserve it by calling convention.)

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

(D) `def function_D(x, y):`

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

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

yes ... no

If we want to use saved registers `s0` and `s1`, we must preserve them for our caller to abide by the calling convention. So we need to allocate additional space on the stack, store the initial values of the saved registers, and restore them before we return. Otherwise, this is a legal and reasonable use of `s0` and `s1` to store values (`x` and `y`) that we don't want the call to `foo` to overwrite.

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

An alternative would be to forgo the usage of `s0` and `s1` entirely, and simply store and restore `a0` and `a1` to/from the stack directly. We can choose any caller-saved registers other than `a0` to restore those values to; below we choose `t0` and `t1`.

```
function_D:
    addi sp, sp, -12
    sw ra, 0(sp)
```

```
sw a0, 4(sp)
sw a1, 8(sp)
li a0, 1
li a1, 2
jal foo
lw t0, 4(sp)
lw t1, 8(sp)
add a0, a0, t0
add a0, a0, t1
lw ra, 0(sp)
addi sp, sp, 12
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
```

```
mul:
   beqz a0, mul_done
   addi sp, sp, -8
   sw s0, 0(sp)
   sw ra, 4(sp)
   andi t0, a0, 1 // lowbit in t0
   mv s0, zero // p in s0
   beqz t0, lowbit_zero
   mv s0, a1
lowbit_zero:
   srli a0, a0, 1
   jal mul
   slli a0, a0, 1
   add a0, s0, a0
   lw s0, 0(sp)
   lw ra, 4(sp)
   addi sp, sp, 8
mul_done:
   ret
```

Errors (intentional, there may be unintentional ones too…):

1. `s0` and `ra` are saved and restored from different offsets – should be `lw ra, 4(sp); lw s0, 0(sp)`
2. `beqz a0, mul_done` should be before `sp` is decremented (or `mul_done` label should be moved up 3 instructions), because we need to make sure that the stack is preserved even if x == 0. That is, in any call to `mul`, the value of `sp` should be the same at the end as it was at the start, whether or not x == 0.
3. p cannot be in `t0` because it's caller-saved and used after call. The simplest fix is to store lowbit in `t0` and p in `s0` instead, which we did above. We could also use an `s1` register, although we'd have to save and restore it for our caller. Alternatively, we could add code before and after `jal mul` to save and restore `t0` in the stack.
4. `slli` and `srli` are switched (first one should be `srli` for `>>` (right-shift); seconds should be `slli` for `<<` (left-shift))
5. p should come from `a1` not `a0`.