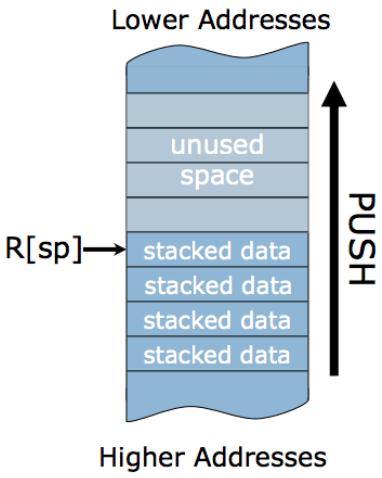
6.004 Tutorial Problems
L04A – Procedures and Stacks II

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 three instructions are equivalent:
  - `jal ra, label: R[ra] <= pc + 4; pc <= label`
  - `jal label` (pseudoinstruction for the above)
  - `call 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
```

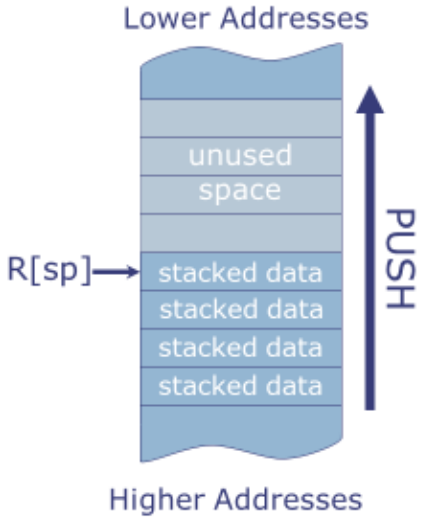
Assume `0(sp)` holds valid data.

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!



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
```

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.

Write assembly program that computes square of the sum of two numbers (i.e. squareSum(x,y) = $(x + y)^2$) and follows RISC-V calling convention. Note that in your assembly code you have to call assembly procedures for **`mult`** and **`sum`**. They are not provided to you, but they are fully functional and obey the calling convention.

Python code for square of the sum of two numbers

```
def squareSum(x, y):
    return mult(sum(x, y), sum(x, y))

# start of the assembly code
squareSum:
```

```
addi  sp sp -4
sw    ra 0(sp)  // sw ra => sp
call  sum
addi  sp sp -4
sw    ra 0(sp)
addi  a0 a0 a1  //
lw    ra 0(sp)
addi  sp sp 4
ret
call  mult
addi  sp sp -16
sw    ra 0(sp)
sw    s2 12(sp)
li    s1 0  //
li    s2 0  // counter.
sw    s0 4(sp)
sw    s1 8(sp)  //  a0 => s0
mv    s0 a0
loops:
      blt  s1 s2 end
      addi s2 s2 1
      add  a0 a0 a0   // a0 = a0 + a0 + ... + a0   (n terms)
end:
      lw   ra 0(sp)
      lw   s0 4(sp)
      lw   s1 8(sp)
      lw   s2 12(sp)
      addi sp sp 16
      ret
      lw   ra 0(sp)
      ret  // results in
```

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.

Write assembly program that computes square of the sum of two numbers (i.e. squareSum(x,y) = $(x + y)^2$) and follows RISC-V calling convention. Note that in your assembly code you have to call assembly procedures for **`mult`** and **`sum`**. They are not provided to you, but they are fully functional and obey the calling convention.

Python code for square of the sum of two numbers

```
def squareSum(x, y):
    return mult(sum(x, y), sum(x, y))
```

```
# start of the assembly code
squareSum:
        addi  sp  sp -16
        sw    ra  0(sp)
        sw    a0  4(sp)
        sw    a1  8(sp)
        sw    s0  12(sp)    // s0 = a0 + a1

        call  sum
        mv    s0  a0        // store sum in s0
        lw    a0  4(sp)
        lw    a1  8(sp)
        call  sum
        mv    a1  s0        // sum, in a1
        call  mult
        lw    s0  8(sp)
        lw    ra  0(sp)
        addi  sp  sp  16
        ret
```

Problem 2. ★

The following C program computes the log base 2 of its argument. The assembly code for the procedure is shown on the right, along with a stack trace showing the execution of ilog2(10). The execution has been halted just as it's about to execute the instruction labeled "rtn:" The SP label on the stack shows where the SP is pointing to when execution halted.

```
/* compute log base 2 of arg */
int ilog2(unsigned x) {
    unsigned y;
    if (x == 0) return 0;
    else {
        /* shift x right by 1 bit */
        y = x >> 1;
        return ilog2(y) + 1;
    }
}
```

```
ilog2: beqz a0, rtn
       addi sp, sp, -8
       sw s0, 4(sp)
       sw ra, 0(sp)
       srli s0, a0, 1
       mv a0, s0
       jal ra, ilog2
       addi a0, a0, 1
       lw ra, 0(sp)
       lw s0, 4(sp)
       addi sp, sp, 8

rtn:   jr ra
```

(A) Please fill in the values for the two blank locations in the stack trace shown on the right. Please express the values in hex.

Fill in values (in hex!) for 2 blank locations

	0x93
	0x240
	0x1
SP→	0x240
	0x5
	0x1108
	0x37

(B) What are the values in a0, s0, sp, and pc at the time execution was halted? Please express the values in hex or write "CAN'T TELL".

Value in a0: 0x 2 **in s0: 0x** 2

Value in sp: 0x can't tell **in pc: 0x** 250

(C) What was the address of the original ilog2(10) function call?

Original ilog2(10) address: 0x 1108

Problem 3. ★

You are given an incomplete listing of a C program (shown below) and its translation to RISC-V assembly code (shown on the right):

```
int fn(int x) {
   int lowbit = x & 1;
   int rest = x >> 1;
   if (x == 0) return 0;
   else return ???;
}
```

```
fn: addi sp, sp, -12
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw ra, 8(sp)
    andi s0, a0, 1
    srai s1, a0, 1
yy: beqz a0, rtn
    mv a0, s1
    jal ra, fn
    add a0, a0, s0

rtn: lw s0, 0(sp)
     lw s1, 4(sp)
     lw ra, 8(sp)
     addi sp, sp, 12
     jr ra
```

(A) What is the missing C source corresponding to ??? in the above program?

C source code: _____

The procedure **`fn`** is called from an external procedure and its execution is interrupted just prior to the execution of the instruction tagged '**`yy:`**'. The contents of a region of memory during one of the **recursive calls** to **`fn`** are shown on the left below. If the answer to any of the below problems cannot be deduced from the provided information, write "CAN'T TELL".

```
fn: addi sp, sp, -12
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw ra, 8(sp)
    andi s0, a0, 1
    srai s1, a0, 1
yy: beqz a0, rtn
    mv a0, s1
    jal ra, fn
    add a0, a0, s0
rtn: lw s0, 0(sp)
    lw s1, 4(sp)
    lw ra, 8(sp)
    addi sp, sp, 12
    jr ra
```

	0x1
0x1D0	???
	0x4C ra
SP→	0x1 s0
	0x11 s1
	0x4C ra
	0x1 s0
	0x23 s1
	0x4C ra
	0x3 orig s0
	0x22 orig s1
	0xC4 orig ra

(B) What was the argument to the most recent call to **`fn`**?

Most recent argument (HEX): x= 0x11

(C) What is the missing value marked **`???`** for the contents of location 1D0?

Contents of 1D0 (HEX): Can't Tell

(B) What is the hex address of the instruction tagged **rtn:?**

Address of rtn (HEX): 0x50

(C) What was the argument to the *first recursive* call to **`fn`**?

First recursive call argument (HEX): x= 0x23

(D) What is the hex address of the *jal* instruction that called **`fn`** *originally*?

Address of original call (HEX): 0x4C

(E) What were the contents of s1 at the time of the *original* call?

Original s1 contents (HEX): 0x22

(F) What value will be returned to the *original* caller if the value of *a0* at the time of the original call was 0x47?

Return value for original call (HEX): 4