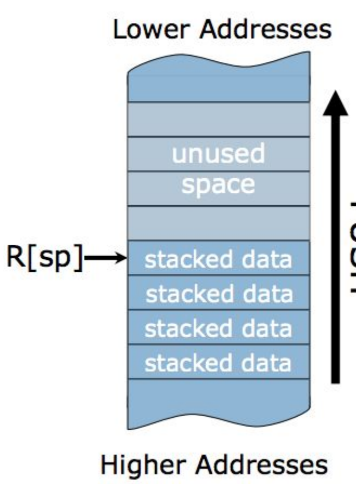
6.004 Tutorial Problems
L04 – Procedures and Stacks II

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 (pc stands for program counter, the memory address of the current/next instruction):
  - `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
```

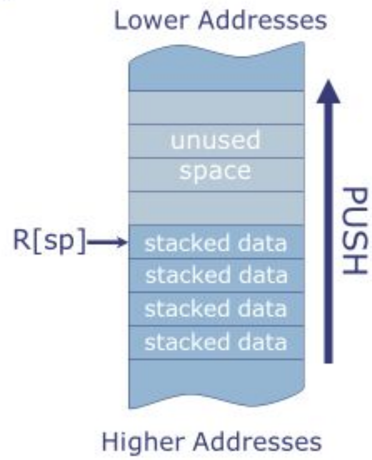
Assume `0(sp)` holds valid data.

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!



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.

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.

a0 a1

```
/* compute square sum of args */
unsigned squareSum(unsigned x, unsigned y) {
    unsigned z = sum(x, y);
    return mult(z, z);
}

// start of assembly code
```

a1 a0 a0 a1

① Figure registers to store in mem.

- ra

```
addi sp, sp, -4
sw ra 0(sp)
jal sum
mv a1 a0
jal mult

lw ra 0(sp)
addi sp, sp, 4
ret
```

a0

★ i log 2 stack ↑ 2

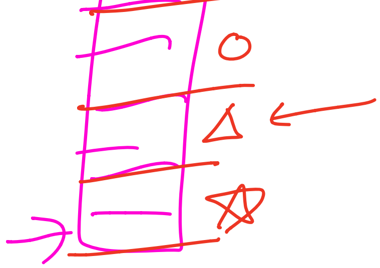
Δ i log 2 stack ↑ 2

O i log 2 stack ↑ 2

stack ↓ 2

stack #2

stack 2



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.

ilog(0x5)

y = s0

```
/* 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
0x240   addi a0, a0, 1
0x244   lw ra, 0(sp)
0x248   lw s0, 4(sp)
0x24C   addi sp, sp, 8
0x250 rtn: jr ra
```

0+1+1

0x5 shifted right by 1 bit

recursion

(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

0x0	
0x93	
0x240	ra
0x1	s0
0x240	ra
0x2	s0
SP→ 0x240	ra
0x5	s0
0x1108	ra
0x37	s0

s0

(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 1104

0x4

main()

0x1104? ilog2

0x1108 ra

ilog2 ilog2 ilog2 return

0x1108 − 0x4 = 0x1104

0b1001100 0b1001111 0b1000001

0b 0000001 0b 000...
0000000 000001

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 ???;
}
```

anding
s0 = a0 &1
lowbit = x &1

rest = s1

a0

```
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
```

a0 =

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

C source code: fn(rest) + lowbit

return fn(0) + ~~s0(?)~~

rest

lowbit

funct1 0b1001100

base +0 0b100110

+0 0b10011

funct1 +1 0b1001

+1 0b100

+0 0b10

+0 0b1

3 +1 0b0000

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".

garbage

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

Sp Sp first function call

(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

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

Address of rtn (HEX): _______ 0x50

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

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

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

Address of original call (HEX): _______ 0xC0

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

Original s1 contents (HEX): _______ 0x22

(H) 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): _______

main
s0 s1

0xC0 jal fn
0xC4 ☆

fn
fn
fn

next to ☆

a<0
if t0<0
return

int a = x - y
t0 = a0 - a1

Problem 4. ★

The following C program implements a function H(x,y) of two arguments, which returns an integer result. The assembly code for the procedure is shown on the right.

a0

a1

```
int H(int x, int y) {
   int a = x - y;
   if (a < 0) return x;
   else return ???;
}
```

a0

```
H:     sub t0, a0, a1
       bltz t0, rtn
       addi sp, sp, -4
       sw ra, 0(sp)
       mv a0, t0
       jal H
       lw ra, 0(sp)
       addi sp, sp, 4
rtn:   jr ra
```

a0 =

The execution of the procedure call **H(0x68, 0x20)** has been suspended just as the processor is about to execute the instruction labeled "rtn:" **during one of the recursive calls to H**. A *partial* trace of the stack at the time execution was suspended is shown to the right below.

(A) Examining the assembly language for H, what is the appropriate C code for ??? in the C representation for H?

C code for ???: H(a, y)

a0 a1

(B) Please fill in the values for the blank locations in the stack dump shown on the right. Express the values in hex or write "---" if value can't be determined. For all following questions, suppose that during the initial (non-recursive) call to H, sp pointed to the memory location containing 0x0010.

Fill in the blank locations with values (in hex!) or "---"

	0x007c
sp →	
	0x001c
	0x0010
	0x00e0
	0x00ec

(C) Determine the specified values at the time execution was suspended. Please express each value in hex or write "CAN'T TELL" if the value cannot be determined.

Value in a0 or "CANT TELL": 0x_______________

Value in a1 or "CANT TELL": 0x_______________

Value in ra or "CANT TELL": 0x_______________

Value in sp or "CANT TELL": 0x_______________

Address of the initial call instruction to H: 0x______________

From past quizzes:

Problem 4. Stack Detective (16 points)

Below is the Python code for a recursive implementation of binary search, which finds the index at which an element should be inserted into a sorted array to ensure that the array is still sorted after the insertion. To the right is a not so elegant, but valid, implementation of the function using RISC-V assembly.

```
/* find where to insert element in arr */
unsigned binary_search(int[] arr,
                        unsigned start,
                        unsigned end,
                        int element){
      if (start == end){
            return end;
      }
      mid = (start + end) / 2;
      if (element < arr[mid]){
            end = mid;
      } else {
            start = mid + 1;
      }
      return binary_search(arr, start, end,
      element);
}
```

```
binary_search: addi sp, sp, -8
               sw ra, 4(sp)
               sw s0, 0(sp)
               mv s0, a2
               beq a1, a2, done
               add t0, a1, a2
               srli t0, t0, 1
               slli t1, t0, 2
               add t1, a0, t1
               lw t1, 0(t1)
if:            bge __________
               mv a2, t0
               j recurse
else:          addi t0, t0, 1
               mv a1, t0
recurse:       call binary_search
               mv s0, a0
done:          mv a0, s0
               lw s0, 0(sp)
               lw ra, 4(sp)
L1:            addi sp, sp, 8
               ret
```

(A) (2 points) What should be in the blank on the line labeled **if** to make the assembly implementation match the Python code?

if: bge ____________________________

(B) (2 points) How many words will be written to the stack before the program makes each recursive call to the function **binary_search**?

Number of words pushed onto stack before each recursive call? ______________

The program's initial call to function **binary_search** occurs outside of the function definition via the instruction '**call binary_search**'. The program is interrupted *during a recursive call* to **binary_search**, *just prior* to the execution of '**addi sp, sp, 8**' at label **L1**. The diagram on the right shows the contents of a region of memory. All addresses and data values are shown in hex. **The current value in the SP register is 0xEB0 and points to the location shown in the diagram.**

Memory Contents

	Address	Data
	0xEA4	0x0
	0XEA8	0x5
	0XEAC	0xC4
SP→	0XEB0	0x6
	0xEB4	0xC4
	0xEB8	0x6
	0xEBC	0xC4
	0xEC0	0xA
	0xEC4	0xC4
	0xEC8	0x3E
	0xECC	0xCA4
	0xED0	0xCED

(C) (4 points) What were the values of arguments `arr` and `end` at the beginning of *the initial call* to **binary_search**? Write CAN'T TELL if you cannot tell the value of an argument from the stack provided.

Arguments at beginning of this call : `arr` = **0x**_______________

`end` = **0x**_______________

(D) (4 points) What are the values in the following registers right when the execution of **binary_search** is interrupted? Write CAN'T TELL if you cannot tell.

Current value of s0: 0x_______________

Current value of ra: 0x_______________

(E) (2 points) What is the hex address of the '**call binary_search**' instruction that made the *initial call* to **binary_search**?

Address of instruction that made initial call to binary_search: 0x _______________

(F) (2 points) What is the hex address of the `ret` instruction?

Address of `ret` **instruction: 0x** _______________