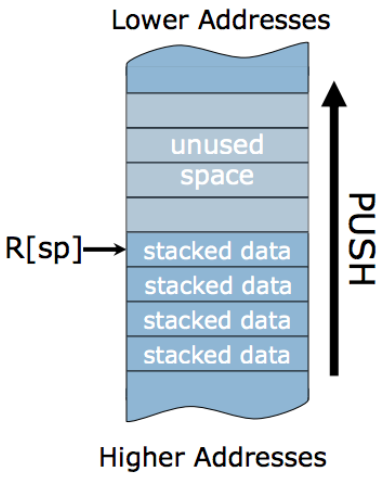
6.004 Tutorial Problems
L04A – Procedures and Stacks II

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
```

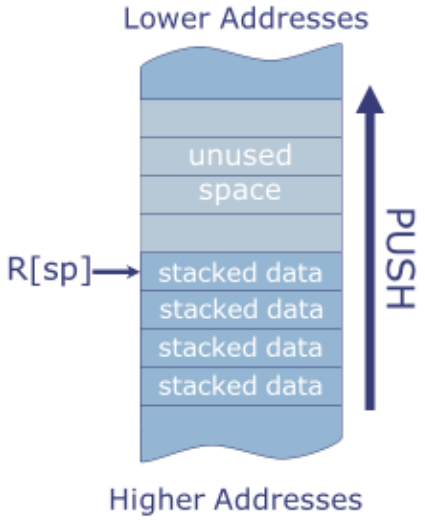
Assume `0(sp)` holds valid data.

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, -16 // adjust stack pointer
      sw a0, 0(sp) // a0 -> x
      sw a1, 4(sp) // a1 -> y
      sw s0, 8(sp) // Store s0 before using it
      sw ra, 12(sp) // Store ra since it will be overwritten
      jal sum // same as jal ra, sum: call sum(x, y)
      mv s0, a0 // store the result of the first call to sum
      lw a0, 0(sp) // restore x
      lw a1, 4(sp) // restore y
      jal sum // same as jal ra, sum: call sum(x, y)
      mv a1, s0 // retrieve the result of the first call
      jal mult // same as jal ra, mult: call mult(sum(…), sum(…))
      lw s0, 8(sp) // restore the caller's s0
      lw ra, 12(sp) // restore ra
      addi sp, sp, 16 // adjust stack pointer
      ret
```

The above solution compiles this function to follow the Python code strictly, without using any of our own understanding of what `mult` and `sum` mean. We call `sum` twice since in general, it might give different results the two times it's called, or it might have side effects such as printing something to the screen. Many other solutions are possible.

The code can be much simplified if we can assume sum has no side effects and returns the same result both times:

```
squareSum:
      addi sp, sp, -4 // adjust stack pointer
      sw ra, 0(sp) // Store ra since it will be overwritten
      jal sum // same as jal ra, sum: call sum(x, y)
      mv a1, a0 // copy the result of sum to a1
      jal mult // same as jal ra, mult: call mult(sum(…), sum(…))
      lw ra, 0(sp) // restore ra
      addi sp, sp, 4 // adjust stack pointer
      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	ra
	0x1	s0
	0x240	ra
	0x2	s0
SP→	0x240	ra
	0x5	s0
	0x1108	ra
	0x37	s0

By analyzing the program's assembly code, we can that in each iteration the stack pointer is decremented by 8, and then the values of s0 and ra are stored in the two words that were just allocated. Since we're about to execute the 'jr ra' instruction, we just loaded ra and s0 off the stack, and then added 8 to the stack pointer. Thus, the two blanks must be the s0 and ra the current function call saved to the stack originally.

If we work backwards from 'jr ra', we see that we also must have come back from a recursive call to ilog2, given the 'jal ra, ilog2'. Thus, the two slots above the blanks are the s0 and ra of that recursive call. This gives us some more info, that the address of the 'addi a0, a0, 1' instruction is 0x240, and that the recursive call is given 0x1 as its argument, which can be deduced by seeing that the argument a0 is just copied from s0 right before the recursive call. In each recursive call, a0 is shifted to the right by one, so in the current function call, the value of a0 must have been 0x2 or 0x3. Since the original call had an argument of 10, we are also in a recursive call! This means the ra that we save to the stack is also 0x240, hence the first blank.

Finally, we can go to the two slots below the blanks to see what our parent call placed there. Since it placed 0x5 as s0's saved value, our argument would have been 0x5 >> 1, or 0x2. This gives us the second blank.

(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_____ in s0: 0x_____

Value in sp: 0x_____ in pc: 0x_____

a0 = 2, s0 = 2, sp = Can't tell, pc = 0x250

To find a0, we need to know where the base case occurred. We know the two slots above the blank are for the next recursive call. Since it has an argument of 1, its next recursive call gets an argument of 0, meaning it skips recursing and just returns. The function returns to our child call, which adds 1 to a0 and then returns to our current call, which also adds 1 to a0. Thus, a0 = 2.

We know we just loaded s0 back from the stack, so we can work backward to see should "un-add" 8 to see where the stack was, then load s0 from 4 after, which points to the blank which we filled with 0x2.

At no point in the program does the value of sp get saved to the stack, so there's no way for us to tell where the stack lives in memory.

Since we know that the return address of the recursive call is 0x240, we can count forward instruction by instruction, adding 0x4 each time to find the pc of the 'jr ra' instruction, which is at 0x250

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

Original ilog2(10) address: 0x_____

0x1104

We know that the two stack entries 0x240 and 0x5 were from the current call's parent call. Since its argument was thus 0x5, it still is not the original call. Thus, the next two entries must be from *its* parent call. However, we now find completely different values for s0 and ra. This must mean that these were the values saved when ilog2 started running, since the ra of 0x1108 points to some distant other part of memory. We can then tell that this is the data saved by the original call. If the return address of that original call is 0x1108, then the original call occurred at 0x1104, since it added 4 to its PC to get the return address.

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: _____

fn(rest) + lowbit

By taking a close look at the assembly, we can see that s0 is set to x & 1, since the first argument x is located in the a0 register. Thus, s0 contains 'lowbit'. The same can be done for s1, which contains 'rest'. On the line labeled 'yy', we branch forward to 'rtn' if a0, or x, is equal to 0. Thus, 'rtn' should correspond to the 'then' block of the if in the source code, and 'yy' corresponds to the 'else' block. In 'yy', we recursively call fn with s1 moved into a0 for the argument, then add s0 to the result. In the source code, this corresponds to fn(rest) + lowbit. Then from here, the program just cleans up and returns, so this is the result.

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".

From the context we can learn a lot of information about what's on the stack. If we look backwards a bit from the point that execution is stopped, we see that we just recently subtracted 12 from sp, which allocated 3 words of space on the stack, and then stored s0, s1, and ra there. Thus, sp currently points to the first of those 3 words stored.

Additionally, since execution was stopped prior to the 'jal ra, fn' recursive call, the 3 words above the current sp are junk, since we never created a recursive call to store those values, they are left over from something else.

Lastly, since we know we are in a recursive call, the 0x4C return address must be to the 'add a0, a0, s0' instruction right after the recursive call. Additionally, since each call to fn saves 3 words to the stack, each layer of the call has an activation frame of size 3, allowing us to reconstruct the previous activation frames, which are below the current one. We see that the activation frame below the current one also contains the same return address, so it too is a recursive call. However, the frame below that has a different ra, so it must be the frame of the original call to fn.

Junk	0x1
Junk	???
Junk	0x4C
s0 SP→	0x1
s1 / rest	0x11
ra	0x4C
s0 / lowbit	0x1
s1 / rest	0x23
ra	0x4C
orig s0	0x3
orig s1	0x22
orig ra	0xC4

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

Most recent argument (HEX): x=_____0x11

The argument is moved from s1 to a0 right before the recursive call is made, so that value is also still in s1, since we haven't modified s1 yet. This means the s1 value saved to the stack is the same as the argument, which in this case is 0x11.

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

Contents of 1D0 (HEX): _____CAN'T TELL

We cannot tell what the value of ??? is because those entries are not part of a recursive call to fn, since we haven't reached the 'jal ra, fn' yet. Those values could be leftover from anything, and thus have any random value.

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

Address of rtn (HEX): _____0x50

As discussed before, we know that the recursive return address is 0x4C, the address of the 'add a0, a0, s0' instruction. We can get the address of the instruction labelled **rtn**, which is the very next instruction, by simply adding 4 to this address to get 0x50.

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

First recursive call argument (HEX): x=_____0x23

We were able to use the stack to discover that the lowermost activation frame is that of the original call to fn, which makes the frame above it that of the *first recursive call*. We can then use our knowledge that the saved s1 value is the same as the argument to find out that the argument to this call was 0x23.

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

Address of original call (HEX): _____0xC0

The return address that we find in the original call's stack frame is 0xC4. If this is the address that the original call will eventually return to, then the address of the instruction that made that call would be 4 before it, meaning 0xC4 – 4 = 0xC0

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

Original s1 contents (HEX): _____0x22

Just like all the recursive calls, the original call saved the value s1 initially had on the stack. By reconstructing the frames for each call, we discovered that this was the 0x22 value.

(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): _____0x4

counts the number of 1's in original number

This problem doesn't actually rely on the contents of the stack, just understanding what the assembly code is actually doing. As long as x is not 0, the program continues to recurse. Then, as it comes back up the recursive stack, it adds 'lowbit' each time, which is 1 if the lowest bit of the current x is 1, with x getting shifted each time. So the program effectively looks at each bit of x and totals how many ones it finds. 0x47 in binary is 0b01000111, which means the program will find 4 ones.