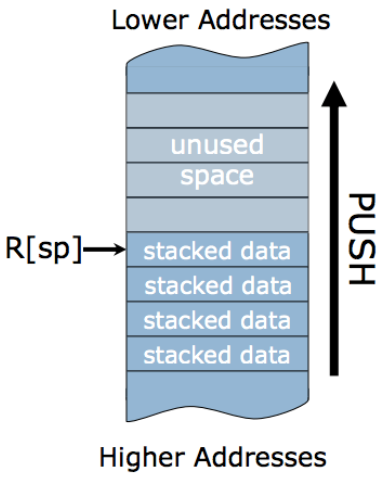
6.004 Tutorial Problems
L04B – Procedures and Stacks II

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
```

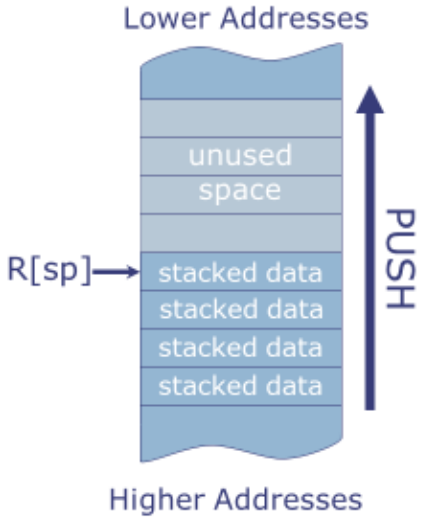
Assume `0(sp)` holds valid data.

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. ★

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.

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

```
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
```

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)

Line 2 of the assembly contains a bltz instruction which branches if the value stored in t0 is less than 0. If this branch is not taken (i.e. a > 0,) we fall through and continue with the addi instruction. Thus, line 3 in the assembly is the start of the "else" block. We see the stack pointer being decremented by 4 and the return address (ra) being pushed onto the stack with the sw instruction. This is the typical entry sequence in preparation for a function call. Next, an mv instruction copies the value in t0 into a0 before calling the function H (recall that jal H is the same as the call H instruction.) By the calling convention, the arguments passed to H are placed in registers a0 and a1. At this point, we see that the value in a0 is t0 (from the mv instruction.) t0 currently contains the value of a. The a1 register has not been changed and has the same value as when H was first called. Thus, it contains the second argument passed to H in the initial (non-recursive) call, so it contains y. The jal H instruction is therefore calling H(a, y).

(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 →	0x007c
	0x001c
	0x0010
	0x00e0
	0x00ec

In the initial call to H (let's call this Call 1), sp starts at the location with 0x0010. As it is about to call H (i.e. just before the jal instruction,) it moves the stack pointer up by 4. Recall that lower memory addresses are drawn at the top of the diagram. Then, it pushes ra onto the stack. We see that this value is 0x001c, which is the value of the ra register during the first call to H. This is the value set by original caller. We don't know which function called H, but we do know that to return to that function, we can simply jump back to 0x001c. Also, we know that at this point, x = 0x68, y = 0x20, and a = x – y = 0x48. This is greater than 0, so we take the else branch and call H again recursively with H(a, y) = H(0x48, 0x20), (Call 2).

Now, when we call H with the jal instruction, we will overwrite the value in ra with the address of the lw instruction and then jump to the H label. At this point, x = 0x48, y = 0x20, and a = x – y = 0x28. ra currently equals the address of the lw instruction (we don't know what this is yet.) Once again, a >= 0, so we take the else branch and call H again with H(a, y) = H(0x28, 0x20), (Call 3). To do so, we once again move sp by -4 and push ra onto the stack with the mv instruction. This is the blank slot currently. We don't know what this is yet. The jal instruction jumps to H and sets ra to the address of the lw instruction.

We repeat the same procedure as above, noting that a = 0x28 – 0x20 = 0x08. This is >= 0, so we take the else branch, move sp by 4, and push ra (the address of the lw instruction.) Observe that this is 0x007c. We now know the address of the lw instruction – it's 0x007c. This was exactly the same value that was pushed before Call 3. Therefore, the blank slot in the middle of the stack has value 0x007c. Continuing, we call H again with H(0x08, 0x20), (Call 4).

In Call 4, our subtraction produces a negative result, so the bltz instruction jumps to the jr ra instruction and we do not store anything else on the stack. At this point, we will simply be popping off the values on the stack as we return through each of the recursive calls. Therefore, we do not have any additional information about values above 0x007c in the stack. The call to ret jumps us back to ra which points to the lw instruction. This causes us to load the previous ra that was stored on the stack and jump back to that. This continues until we jump back to 0x001c (the ra set by the function that called us in the initial call (Call 1.) Since we do not know what that function does with the stack, we similarly do not have any information about the contents at the bottom of the stack (below 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__8_____

Value in a1 or "CANT TELL": 0x__20_____

Value in ra or "CANT TELL": 0x__7C_____

Value in sp or "CANT TELL": 0x_CAN'T TELL____

Address of the initial call instruction to H: 0x_0018_____

The answers to these questions can be found through the analysis in part (B). Based on the position of sp, we can see that execution was halted while returning from Call 3 as described in the previous section. We have just popped off the top-most 0x007c value on the stack and loaded that into ra. Thus value at ra is 0x7c. At this point, a0 and a1 have not been modified since the final call to H. Thus they contain the values of the final arguments passed to H in Call 4, which

are 0x08 and 0x20 respectively. We have no information about where the stack sits in memory. We only decremented and incremented sp relative to its original value (which is unknown.) Therefore, we do not know the value of sp.

From the analysis in part (B), we know that before we make Call 2, we store the value of ra which is the return address for the function that made Call 1 (the initial call.) This is 4 greater than the address of the actual instruction that called us (since jal links in the next instruction rather than the actual instruction that does the calling.) Thus the address of the instruction that called is us 0x001c – 4 = 0x0018.

Problem 2.
Consider the following memory map of the MMIO interface for a RISC-V processor:

Address	Read/Write	Description
0x4000 0000	WRITE-ONLY	Out to console (prints ASCII character)
0x4000 0004	WRITE-ONLY	Out to console (prints decimal number)
0x4000 0008	WRITE-ONLY	Out to console (prints hexadecimal number)
0x4000 4000	READ-ONLY	In from console (gets signed word)
0x4000 5000	READ-ONLY	Cycle counter (gets number of instructions executed since simulation start)
0x4000 6000	READ-ONLY	Performance counter (gets number of instructions executed while counter on)
0x4000 6004	WRITE-ONLY	0 to turn performance counting off, 1 to turn performance counting on. Starts off with value 0.

(A) Modify the following assembly code to print out the return value of the function in hexadecimal in addition to returning it.

```
main:


        slli a0, a0, 2


        add a0, a0, a1
        li t0, 0x40000008
        sw a0, 0(t0)
        jr ra
```

The memory address that handles hexadecimal console printing is 0x40000008. By writing a value to that memory address with sw, we can trigger the hardware to print to the console.

(B) Modify the following assembly code to track how instructions are executed by `func_B`. Print this value in decimal to the console.

```
func_A:
        mv t0, a0
        andi t0, t0, 1
        addi sp, sp, -8
        sw t0, 4(sp)
        sw ra, 0(sp)
        li t1, 0x40006004
        li t2, 1
        sw t2, 0(t1)
        call func_B
        li t1, 0x40006004
        sw zero, 0(t1)
        lw ra, 0(sp)
        lw t0, 4(sp)
        addi sp, sp, 8
        add a0, a0, t0
        li t1, 0x40006000
        lw t2, 0(t1)
```

```
        li t1, 0x40000004
        sw t2, 0(t1)
        ret
```

We use the performance counter which can be turned on and off and only counts when it is turned on. Turn the performance counter on when calling func_B and off after it is called to count the number of cycles spent in func_B. This is achieved by writing 1 and 0 respectively to 0x40006004. Read this count from 0x40006000 and write it out to 0x40000004 which will print it in decimal.

(C) Write a short assembly program to print the integers in descending order from 5 through 1 (inclusive) to the console with each number on its own line.

```
print_num:
        li a0, 5
        li a1, 0x4000000
        li a2, 0xA // newline character
loop:
        sw a0, 4(a1) // print decimal number
        sw a2, 0(a1) // print newline to ascii
        addi a0, a0, -1
        bnez a0, loop
return:
        ret
```

Problem 3. ★

Consider the following program that (naively) sums an array recursively. Assume that the base address of the array is passed in register a0 and the length of the array is passed in a1. For all problems in this section, assume that the value at memory address 0x4010 is 1.

Python:

```
def array_sum(p, length):
    if length == 1:
        return p[0]
    return p[0] + array_sum(p[1:], length - 1)
```

RISC-V Assembly:

```
array_sum:
    lw t1, 0(a0)
    addi a1, a1, -1
    beqz a1, return
    addi a0, a0, 4
    addi sp, sp, -8
    sw t1, 4(sp)
    sw ra, 0(sp)
    jal array_sum
    lw ra, 0(sp)
    lw t1, 4(sp)
    addi sp, sp, 8
    add t1, t1, a0
return:
    mv a0, t1
    jr ra
```

	0x58
	0x02
	???
	0x06
SP →	0x58
	0x04
	0x24
	0x03

(A) The procedure call `array_sum(0x4000, 5)` has been halted at the `return` label, just before the processor is about to execute the mv instruction **during one of the recursive calls to array_sum**. The incomplete stack at this point is shown on the right. What should the value at ??? be?

0x58

From the assembly, we know that before each recursive call to array_sum with the jal instruction, we decrement sp by 8 to store t1 and ra on the stack. Because we are currently at the return label, we have just popped ra and t1 off the stack (the two lw instructions.) Thus the ??? represents the return address for the current function call that was saved to the stack.

Looking further back into the assembly, we see that we are returning after a recursive call to array_sum. This tells us that the two slots above the blank form the stack frame for the recursive call, with the top value (0x58) being the ra and the bottom value (0x02) being the stored value of t1. This tells us that 0x58 is the address of the lw instruction underneath the jal instruction. This tells us the value of ??? since the unknown value is also the return address that is stored by a recursive call to array_sum. Thus ??? = 0x58.

(B) What is the value of the second element (index 1) of the array?

0x04

The other value the assembly is storing onto the stack is t1. If we look at where t1 is set, we see that the first line of assembly loads the value at the address in a0 into t1. a0 is the first argument passed into array_sum which is the base address of the array we are summing. By loading this value from memory, we are loading the first element of the array into t1. In each recursive call, we pass the address of the next element in the array as the first argument. Therefore, the stack actually shows each element of the array as its loaded in!

The find the first element of the array, we must find the activation record that belongs to the initial call to array_sum. We can find this by looking at the return addresses. The first recursive call to array_sum will need to store the return address that was set by its caller (i.e. some external function that called array_sum.) All 0x58 values we see on the stack are return addresses set by recursive calls, but 0x24 is also a return address that was saved before the recursive call. Because this address does not fall within our code segment, we know that it belongs to a different caller function. Thus the 0x24 and 0x3 values make up the activation record saved before the first recursive call. This tells us that 0x3 is the 0-th element in the array and 0x04 is the 1st element.

(C) If the value at memory location 0x4010 is 1, what is the sum of all the elements in the array?
0x10 = 16

We know based on the original procedure call to array_sum(0x4000, 5) that the address of the last element of the array is stored at 0x4010. This also happens to be the only value that we cannot deduce from the stack because the last call to array_sum hits a base case and thus does not interact with the stack at all. All other values we can see on the stack, separated by stored ra values. 0x03 + 0x04 + 0x06 + 0x02 + 0x01 = 0x10.

(D) What is the value of `t1` at the current halted location?
0x9

At the current location, we have just finished executing add t1, t1, a0. In this instruction, a0 contains the return value of the recursive call to array_sum. t1 contains the value in the current index of the array we are at. Thus at our current halted location, t1 contains the cumulative sum of the array. We can from the stack that we have already summed the last three values of the array, so 0x06 + 0x02 + 0x01 = 0x9.

(E) What is the address of the instruction that called `array_sum`?
0x24 – 4 = 0x20

Based off the discussion in section (B), we know that the return address saved by the calling function is 0x24. Because we always save the next instruction into ra when calling another function, the address of the actual instruction that called array_sum will be four less than the saved ra. Thus 0x24 – 4 = 0x20.