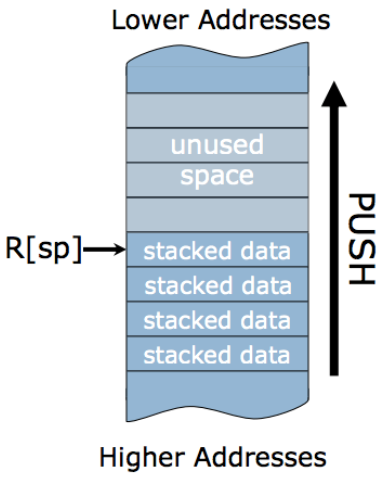
6.004 Tutorial Problems
L04B – Procedures and Stacks II

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
```

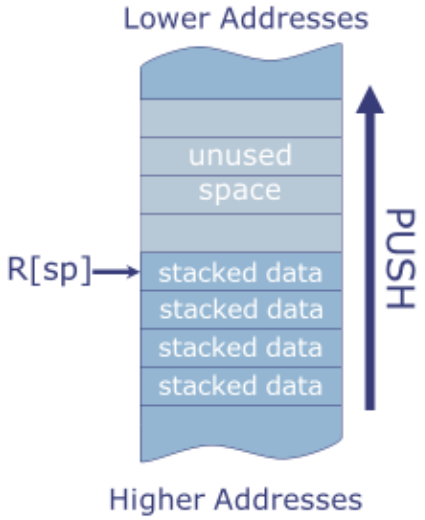
Assume `0(sp)` holds valid data.

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.

```
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
```

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

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 ???: _____

(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_____

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 t1, 0x40006004
        sw a0 0(t1)
        jr ra
```

lw a0 0x4000 0000

(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)

        call func_B

        lw ra, 0(sp)

        lw t0, 4(sp)

        addi sp, sp, 8
```

```
add a0, a0, t0

ret
```

(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.

python:

```
for i in range(5, 1, -1)
    print(i)
```

assembly:

```
loops:
    li to 5
    li t1 0x4000 0004
    sw to t1
    addi to to -1
    beqz to end.
    jr loops:
end:
```

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
```

	1
	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?

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

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

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

(E) What is the address of the instruction that called `array_sum`?