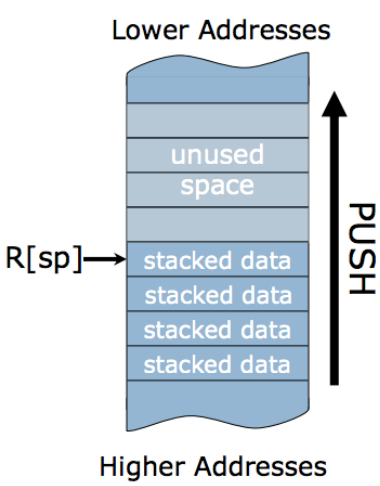
6.004 Tutorial Problems
L3 – Procedures and Stacks

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
  - jal ra, label: R[ra] <= pc + 4; pc <= label
  - jal label

- Callee runs, and places results in registers a0 and a1
- Callee transfers control to caller using jr (jump-register) instruction
  - ret: pc <= R[ra]
  - jr ra
  - jalr x0, 0(ra)



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.

Problem 1.

For the following C functions, does the corresponding RISC-V assembly obey the RISC-V calling conventions? If not, rewrite the function so that it does obey the calling conventions.

(A)

```
int function_A(int a, int b) {
   some_other_function();
   return a + b;
}
```

```
function_A:
   addi sp, sp, -8
   sw a0, 8(sp)
   sw a1, 4(sp)
   sw ra, 0(sp)
   jal some_other_function
   lw a0, 8(sp)
   lw a1, 4(sp)
   add a0, a0, a1
   lw ra, 0(sp)
   addi sp, sp, 8
   ret
```

yes ... no

```
function_A:
   addi sp, sp, -12
   sw a0, 8(sp)
   sw a1, 4(sp)
   sw ra, 0(sp)
   jal some_other_function
   lw a0, 8(sp)
   lw a1, 4(sp)
   add a0, a0, a1
   lw ra, 0(sp)
   addi sp, sp, 12
   ret
```

(B)

```
int function_B(int a, int b) {
   int i = foo((a + b) ^ (a - b));
   ret (i + 1) ^ i;
}
```

```
function_B:
   addi sp, sp, -4
   sw ra, 0(sp)
   add t0, a0, a1
   sub a0, a0, a1
   xor a0, t0, a0
   jal foo
   addi t0, a0, 1
```

```
    xor a0, t0, a0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

yes ... no

(C)

```
int function_C(int x) {
    foo(1, x);
    bar(2, x);
    baz(3, x);
    return 0;
}

function_C:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv a1, a0
    li a0, 1
    jal foo
    li a0, 2
    jal bar
    li a0, 3
    jal baz
    li a0, 0
    lw ra, 0(sp)
    addi sp, sp, 4
    ret
```

yes ... no

```
function_C:
    addi sp, sp, -8
    sw ra, 0(sp)
    mv a1, a0
    sw a1, 4(sp)
    li a0, 1
    jal foo
    lw a1, 4(sp)
    li a0, 2
    jal bar
    lw a1, 4(sp)
    li a0, 3
    jal baz
    li a0, 0
    lw ra, 0(sp)
    addi sp, sp, 8
    ret
```

(D)

```
int function_D(int x, int y) {
   int i = foo(1, 2);
   return i + x + y;
}
```

```
function_D:
   addi sp, sp, -4
   sw ra, 0(sp)
   mv s0, a0
   mv s1, a1
   li a0, 1
   li a1, 2
   jal foo
   add a0, a0, s0
   add a0, a0, s1
   lw ra, 0(sp)
   addi sp, sp, 4
   ret
```

yes ... no

```
function_D:
   addi sp, sp, -12
   sw ra, 0(sp)
   sw s0, 4(sp)
   sw s1, 8(sp)
   mv s0, a0
   mv s1, a1
   li a0, 1
   li a1, 2
   jal foo
   add a0, a0, s0
   add a0, a0, s1
   lw ra, 0(sp)
   lw s0, 4(sp)
   lw s1, 8(sp)
   addi sp, sp, 12
   ret
```

Problem 2.

Write assembly program that computes square of the sum of two numbers (i.e. sumSquare(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.

```
           C code for square of the sum of two numbers
unsigned int squareSum(unsigned int x, unsigned int 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
      mv s0, a0
      lw a0, 0(sp)
      lw a1, 4(sp)
      jal sum // same as jal ra, sum
      mv a1, s0
      jal mult // same as jal ra, mult
      lw s0, 8(sp)
      lw ra, 12(sp) // restore ra
      addi sp, sp, 16 // adjust stack pointer
      ret
```

Problem 3.

Our RISC-V processor does not have a multiply instruction, so we have to do multiplications in software. The C code below shows a recursive implementation of multiplication by repeated addition of unsigned integers (in C, `unsigned int` denotes an unsigned integer). Ben Bitdiddle has written and hand-compiled this function into the assembly code given below, but the code is not behaving as expected. Find the bugs in Ben's assembly code and write a correct version.

C code for unsigned multiplication

```
unsigned int mul(unsigned int x,
                 unsigned int y) {
  if (x == 0) {
    return 0;
  } else {
    unsigned int lowbit = x & 1;
    unsigned int p = lowbit? y : 0;
    return p + (mul(x >> 1, y) << 1);
}
```

Buggy assembly code

```
mul:
  addi sp, sp, -8
  sw s0, 0(sp)
  sw ra, 4(sp)
  beqz a0, mul_done
  andi s0, a0, 1  // lowbit in s0
  mv t0, zero  // p in t0
  beqz s0, lowbit_zero
  mv t0, a0
lowbit_zero:
  slli a0, a0, 1
  jal mul
  srli a0, a0, 1
  add a0, t0, a0
  lw s0, 4(sp)
  lw ra, 0(sp)
  addi sp, sp, 8
mul_done:
  ret
```

```
mul:
  beqz a0, mul_done
  addi sp, sp, -8
  sw s0, 0(sp)
  sw ra, 4(sp)
  andi t0, a0, 1  // lowbit in
t0
  mv s0, zero  // p in s0
  beqz t0, lowbit_zero
  mv s0, a1
lowbit_zero:
  srli a0, a0, 1
  jal mul
  slli a0, a0, 1
  add a0, s0, a0
  lw s0, 0(sp)
  lw ra, 4(sp)
  addi sp, sp, 8
mul_done:
 ret
```

Errors (intentional, there may be unintentional ones too…):

1. s0 and ra are saved and restored from different offsets – should be `lw ra, 4(sp); lw s0, 0(sp)`
2. `beqz a0, mul_done` should be before sp is decremented (or mul_done label should be moved up 3 instructions)
3. p cannot be in t0 because it's caller-saved and used after call; store lowbit in t0 and p in s0 instead, or use an s1 register, or add code before and after jal mul to save and restore t0.
4. Slli and srli are switched (first one should be slri, second slli)
5. p should come from a1 not a0.