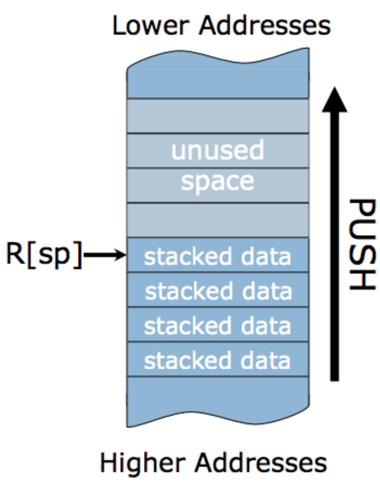
6.004 Fall 2020 Tutorial Problems
L03 – Procedures and Stacks

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`. The following two 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)

- 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
```

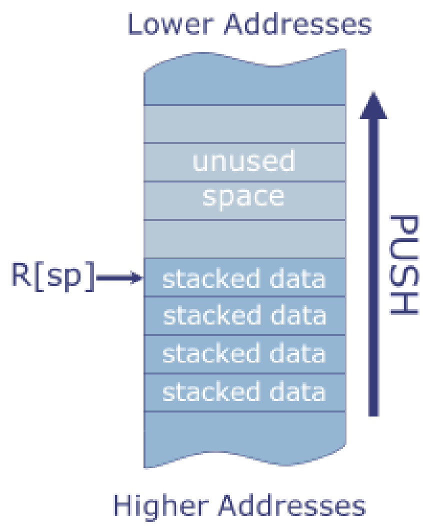
Assume `0(sp)` holds valid data.

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.

Integer arrays **season1** and **season2** contain points Ben Bitdiddle had scored at each game over two seasons during his time at MIT Intramural Basketball Team. Please write a RISC-V assembly function **greaterthan20** which counts the number of games he scored more than 20 points. An equivalent C function and a sample use case are given below. Note that the base addresses for arrays **season1** and **season2** along with their size are passed down to function **greaterthan20**.

```
int greaterthan20(int a[], int b[], int size) {
    int count = 0;
    for (int i = 0; i < size; ++i) {
        if (a[i] > 20)
            count += 1;
        if (b[i] > 20)
            count += 1;
    }
    return count;
}

int main() {
    int season1[] = {18, 28, 19, 33, 25, 11, 20};
    int season2[] = {30, 12, 13, 33, 37, 19, 22};
    int result = greaterthan20(season1, season2, 7);
}
```

```
// Beginning of your assembly code
greaterthan20:
        li t0, 0 // t0 ← count
        li t1, 0 // t1 ← index
        li t2, 20
loop:
```

Problem 2.

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

(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

(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

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

```c
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
```

Problem 4.

For each RISC-V instruction sequence below, provide the hex values of the specified registers after each sequence has been executed. **Assume that all registers are initialized to 0 prior to each instruction sequence.** Each instruction sequence begins with the line (`. = 0x0`) which indicates that the first instruction of each sequence is at address 0. Assume that each sequence execution ends when it reaches the `unimp` instruction.

(A)

```
. = 0x0
     jal x5, L1
     jal x6, end
L1:  j L2
     jal x6, end
L2:  jr x5
end: unimp
```

Value left in x5: 0x______________

Value left in x6: 0x______________

Address of label L2: 0x______________

(B)

```
. = 0x0
      li x7, 0x600
      mv x8, x7
loop: addi x8, x8, 4
      lw x9, 0(x8)
      sw x9, -4(x8)
      blez x9, loop
      lw x7, 0(x7)
end:  unimp
```

Value left in x7: 0x______________

Value left in x8: 0x______________

Value left in x9: 0x______________

The code above refers to certain locations in memory. Assume that the first 4 memory locations starting from address 0x600 have been initialized with the following 4 words.

```
. = 0x600
// First 4 words at address 0x600
.word 0x60046004
.word 0x87654321
.word 0x12345678
.word 0x00000001
```

Problem 5.

(A) Please fill in the blank to make the Python code have the same functionality as the assembly code. The part in the blank should be a mathematical expression of `x` alone using only Python mathematical operations of `+`, `-`, `*`, `/`, `//` (integer division), or `**` (power).

```
map:
  li a1, 1
  sll a0, a1, a0
  ret
```

```python
def map(x):
  return ____________________
```

(B) The code below that calls `map` violates calling convention. Please add appropriate instructions (**either Increment/Decrement stack pointer, Load word from stack, or Save word to stack only**) into the blank spaces on the right to make it follow the calling convention. You may not need to use all the spaces provided.

Your answer should still follow calling convention **even if the map function is modified** to perform something else (that follows the calling convention).

For full credit, you should **only save registers that must be saved onto the stack and avoid unnecessary loads and stores** while following the calling conventions.

```
//pseudocode:
// def array_process(array, size):
//   for i in range(size):
//     array[i] = map(array[i])
//   return array
array_process:
  li t1, 0
  mv s2, a0
  mv s3, a0
loop:
  beq t1, a1, end
  lw a0, 0(s2)
  call map
  sw a0, 0(s2)
  addi s2, s2, 4
  addi t1, t1, 1
  j loop
end:
  mv a0, s3
  ret
```

```
array_process:
  li t1, 0

  ____________________

  ____________________

  ____________________

  ____________________

  ____________________

  ____________________

  mv s2, a0
  mv s3, a0
loop:
  beq t1, a1, end

  ____________________

  ____________________

  lw a0, 0(s2)
```

	<pre> call map
	  sw a0, 0(s2)
	
	  ____________________
	
	  ____________________
	  addi s2, s2, 4
	  addi t1, t1, 1
	  j loop
	end:
	  mv a0, s3
	  ____________________
	
	  ____________________
	
	  ____________________
	
	  ____________________
	
	  ____________________
	  ret</pre>