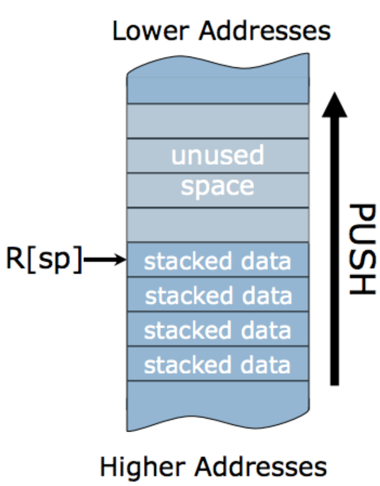
6.004 Worksheet Questions
L03 – Procedures and Stacks I

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
```

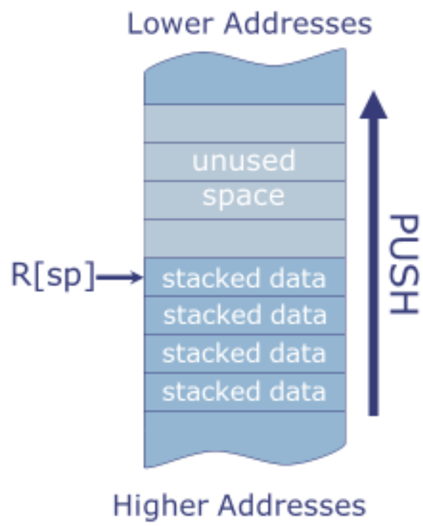
Assume `0(sp)` holds valid data.

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:
        // We are translating the for loop of the original greaterthan20 code into something more easily
        translated into assembly code, like a while loop of the form:
        // while i < size:
        //      (body of loop goes here)
        //      i = i + 1

        ble a2, t1, endloop // if a2 (= size) <= t1 (= i), jump to endloop
                            // (that is, stop the loop if i < size is false)
        slli t3, t1, 2      // t3 ← 4 × index
checka:
        add t4, a0, t3      // t4 ← a0 (= base of array a) + t3
                            // (a0 + t3 = address of a[i])
        lw t5, 0(t4)        // t5 = value at address t4 = value of a[i]
        ble t5, t2, checkb  // if a[i] <= 20, then skip to checking b[i]
        addi t0, t0, 1      // increment count
checkb:
        add t4, a1, t3      // t4 ← a1 (= base of array b) + t3
                            // (a1 + t3 = address of b[i])
```

```
      lw t5, 0(t4)           // t5 = contents of address t4 = value of b[i]
      ble t5, t2, endcompare // if b[i] <= 20, then go to endcompare
      addi t0, t0, 1         // increment count
endcompare:
      addi t1, t1, 1         // increment index i
      j loop                 // restart loop from the condition check
endloop:
      mv a0, t0              // move count to a0, the register for holding the
                             // return value
      ret
```

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

`addi sp, sp, -8` only allocates two words on the stack, `0(sp)` (i.e. `sp + 0`) and `4(sp)` (i.e. `sp + 4`). Therefore, using the address `8(sp)` violates the calling convention because we have not allocated space for a third word. We can fix it just by replacing the `-8` with `-12`, and the `8` with `12` at the end:

```
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
```

Everything else is correct. We save `a0` and `a1` onto the stack and restore them after calling `some_other_function`, since they are caller saved, and that function is allowed to overwrite them. Then we add them and put the result in `a0`, where it is returned to the caller.

(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

Nothing is wrong here. `addi sp, sp, -4` allocates the address `0(sp)`, which we use to store and restore `ra`, so that it's OK when `ra` is overwritten by calling `foo`. `sp` is also restored to the old value at the end. All used registers `a0`, `a1`, `t0` are caller-saved registers, so we are allowed to modify them. Since we also don't assume that `foo` preserves any of those registers when called, as we only need its return value, which appears in `a0`. Thus, we do not need to restore `a0`, `a1`, `t0` or save them to the stack.

(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

The code assumes that its argument `x` will stay in register `a1` as it calls functions `foo` and `bar`, because it needs to pass the same argument to `bar` and `baz`. However, those functions are allowed to overwrite `a1` by calling convention, since `a1` is a caller saved register. Instead, we must store `x` in the stack and restore it when we need it again. (Note that we only need to store `x` once, and we can load it twice; that part of the stack belongs to this function, so neither `foo` nor `bar` is allowed to modify it. Also, we do *not* need to restore `a1` after returning from `baz`, because we don't need it anymore and we aren't required to preserve it by calling convention.)

```
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

If we want to use callee saved registers `s0` and `s1`, we must preserve them for our caller to abide by the calling convention. So we need to allocate additional space on the stack, store the initial values of the saved registers, and restore them before we return. Otherwise, this is a legal and reasonable use of `s0` and `s1` to store values (`x` and `y`) that we don't want the call to `foo` to overwrite.

```
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
```

An alternative would be to forgo the usage of `s0` and `s1` entirely, and simply store and restore `a0` and `a1` to/from the stack directly. We can choose any caller-saved registers other than `a0` to restore those values to; below we choose `t0` and `t1`.

```
function_D:
    addi sp, sp, -12
    sw ra, 0(sp)
    sw a0, 4(sp)
    sw a1, 8(sp)
    li a0, 1
    li a1, 2
    jal foo
    lw t0, 4(sp)
    lw t1, 8(sp)
    add a0, a0, t0
    add a0, a0, t1
    lw ra, 0(sp)
    addi sp, sp, 12
    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. 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
```

```
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), because we need to make sure that the stack is preserved even if x == 0. That is, in any call to `mul`, the value of `sp` should be the same at the end as it was at the start, whether or not x == 0.
3. p cannot be in `t0` because `t0` is caller-saved and used after call. The simplest fix is to store lowbit in `t0` and p in `s0` instead, which we did above. We could also use an `s1` register, although we'd have to save and restore it for our caller. Alternatively, we could add code before and after `jal mul` to save and restore `t0` in the stack.
4. `slli` and `srli` are switched (first one should be `srli` for `>>` (right-shift); second should be `slli` for `<<` (left-shift))
5. p should come from `a1` not `a0`.

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)

<table>
<tr>
<td>

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

</td>
<td>

Value left in x5: 0x______4_________

Value left in x6: 0x______8_________

Address of label L2: 0x_____10_________

</td>
</tr>
</table>

- pc = 0x0: jal L1, x5 sets x5 to the address of the instruction after "jal L1, x5", which is 4, and then jumps to L1
- pc = 0x8: j L2 jumps to L2
- pc = 0x10: jr x5 jumps to the instruction at address 4, which is "jal end, x6"
- pc = 0x4: jal end, x6 sets x6 to the address of the instruction after that "jal end, x6", which is 8
- pc = 0x14: the program stops

L2 is at 0x10 because it is four words after the first word, which is at 0x0, so it's 4 × 4.

(B)

<pre>. = 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</pre>
	Value left in x7: 0x____87654321____
	
	Value left in x8: 0x_____608________
	
	Value left in x9: 0x____12345678____

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
```

The program loops through the given words starting from address 0x604. It copies each word into the previous word in memory, and then continues looping if that word was less than or equal to 0. 0x87654321 is less than 0 because its most significant bit is set in two's complement, but 0x12345678 is not, so the program stops looping when x8 = 0x608 and x9 = 0x12345678. x7 is still equal to its original value, 0x600 until the last line, but loading 0x600 at the end gives 0x87654321 because the first iteration of the loop overwrote it.

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
```

```
def map(x):
  return ______2**x ________
```

(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

  addi sp, sp, -20
  sw a1, 0(sp)
  sw s2, 8(sp)
  sw s3, 12(sp)
  sw ra, 16(sp)

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

  sw t1, 4(sp)

  lw a0, 0(s2)
  call map
  sw a0, 0(s2)

  lw a1, 0(sp)
  lw t1, 4(sp)
```

Prep Phase:
- Stack pointer must be decreased by 20
- Registers a1, ra, s2, s3 must be saved
Inside Loop
- Register t1 must be saved before map call (sw of a1 not needed because it doesn't change)
- Register a1, t1 must be loaded again before arithmetic
End phase:
- Registers s2, s3, ra must be restored from stack
- Stack Pointer must be restored by +20
	`addi s2, s2, 4`
	`addi t1, t1, 1`
	`j loop`
	`end:`
	`mv a0, s3`
	
	`lw s2, 8(sp)`
	`lw s3, 12(sp)`
	`lw ra, 16(sp)`
	`addi sp, sp, 20`
	
	`ret`

Problem 6. From Past Quizzes ★

For each of the following code snippets, provide the value left in each register **after executing the entire code snippet** (i.e., when the processor reaches the instruction at the `end` label), or specify **CAN'T TELL** if it is impossible to tell the value of a particular register. The code snippets are independent of each other.

(A)

```
code_start:
  li x1, 0x26
  lui x2, 0x24
  blt x2, x1, L1
  addi x1, x1, 1
L1:
  add x1, x1, zero
end:
```

1) li x1, 0x26 stores value 0x26 in register x1
2) lui x2, 0x24 loads 0x24 into the upper portion of the register, shifting the value by 12 bits making the value of x2 into 0x24000
3) if x2 is less than x1, we branch to L1. 0x24000 is not less than 0x26, so we do not branch to L1
4) We add 1 to x1 so x1 is 0x27
5) We add 0 to x1, so x1 is still 0x27

Therefore x1 = 0x27, x2 = 0x24000, and we can't tell what pc is because we do not know the starting value of the pc of the code

x1: (0x) ______27____________

x2: (0x) ____24000___________

pc: (0x) ___CAN'T TELL_____

(B)

```
  . = 0x100
  li x4, 0x6
  addi x5, zero, 0xC00
  slli x4, x4, 8
  or x6, x4, x5
end:
```

1. li x4, 0x6 loads value 0x6 into x4
2. addi x5, zero, 0xC00 also acts like li and stores 0xFFFFFC00 into x5. 0xC00 is in twos complement notation. When the 12 bit 0xC00 is stored into x5, its MSB is sign extended so the value in x5 is 0xFFFFFC00, not 0xC00
3. shift x4 logically left by 8, making the value in x4 = 0x600
4. bitwise or x4 and x5 and store in x6. This gives us x6 = 0xFFFFFE00
5. by the end the pc has traveled 4 lines, which at 4 bytes per line means the address has to add 16, pc = 0x110

x4: (0x) ______600______________

x5: (0x) _____FFFFFC00________

x6: (0x) ______FFFFFE00________

pc: (0x) ________110____________

(C)

```
  . = 0x100
  addi x7, zero, 0x204
  li x8, 3
  lw x9, -4(x7)
  sw x8, 4(x7)
end:

. = 0x200
.word 0x01010101
.word 0xAAAAAAAA
.word 0x77777777
```

1. addi x7, zero, 0x204 acts similar to li and stores 0x204 into x7
2. load 3 into x8
3. load the word at position 0x200 into x9, making x9 = 0x01010101
4. save the word in x8 into 0x208, making the word at address 0x208 into 3

x9: (0x) ______01010101______

Which address in memory is written to: (0x) ______208__________

What value is written to memory: (0x) ______3____________

Problem 7. From Past Quizzes

(A) The box below shows the C code for a function `func` and an incorrect implementation in RISC-V assembly. While this implementation follows the logic of the corresponding C code correctly, it fails to follow the RISC-V 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 `func` follow the calling convention. You may not need to use all the spaces provided. You should not modify any of the instructions already provided.

Note all values (x and count) are **signed 32-bit integers.** `func` uses two other functions, `check` and `change`, shown to right, which follow the RISC-V calling convention. Your answer should still follow calling convention **even if the `check` and `change` functions are modified** to perform something else (that follows the calling convention).

```
check:
  not a0, a0
  andi a0, a0, 0x1
  ret

change:
  srli a0, a0, 1
  ret
```

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

```
// C code
// int func(int x) {
//   int count = 0;
//   while (check(x)) {
//     count += 1;
//     x = change(x); }
//   return count; }
```

```
func:
  li t1, 0
while:
  mv s1, a0
  call check
  beqz a0, end
  addi t1, t1, 1
  mv a0, s1
  call change
  j while
end:
  mv a0, t1
  ret
```

```
func:
  li t1, 0

  addi sp, sp, -12
  sw ra, 0(sp)
  sw s1, 4(sp)

while:
  mv s1, a0

  sw t1, 8(sp)

  call check
```

	`lw t1, 8(sp)`
	
	`beqz a0, end`
	`addi t1, t1, 1`
	`mv a0, s1`
	
	`sw t1, 8(sp)`
	
	`call change`
	
	`lw t1, 8(sp)`
	
	`j while`
	`end:`
	`mv a0, t1`
	
	`lw ra, 0(sp)`
	`lw s1, 4(sp)`
	`addi sp, sp, 12`
	
	`ret`

(B) Can you make this code more efficient by changing one of the registers s1, t1, or a0 to use a different register? If so, explain which register should be changed and why. If not, explain why not.

If t1 was replaced by an s register, then it would only need to be stored/restored on/from the stack once rather than every time through the while loop.