Procedures and Stacks

When entering numeric values in the answer fields, you can use integers (1000, 0x3E8, 0b1111101000), floating-point numbers (1000.0), scientific notation (1e3), engineering scale factors (1K), or numeric expressions (3*300 + 100).

Useful links:

- Introduction to BSim
- Summary of Instruction Formats (PDF)
- Beta Documentation (PDF)

Problem 1. Beta ISA

For each of the Beta instruction sequences shown below, indicate the values of the specified registers after the sequence has been executed by an unpipelined Beta. Consider each sequence separately and assume that execution begins at location 0 and halts when the HALT() instruction is about to be executed. Also assume that all registers have been initialized to 0 before execution begins.

Remember that even though the Beta reads and writes 32-bit words from memory, all addresses are *byte addresses*, i.e., the addresses of successive words in memory differ by 4.

You can find detailed descriptions of each Beta instruction in the "Beta Documentation" handout -- see link above.

Hint: You can enter answers in hex by specifying a "`0x`" prefix, *e.g.*, 16 could be entered as "`0x10`". Usually one would enter addresses, values in memory, etc. using hex.

A.
```
        . = 0
        AND(r31, r31, r0)
        CMPEQ(r31, r31, r1)
        ADD(r1, r1, r2)
        OR(r2, r1, r3)
        SHL(r1, r2, r4)
        HALT()
```

Value left in R0? 0
Value left in R1? 1
Value left in R2? 2
Value left in R3? 3
Address of 32-bit memory location containing OR instruction?
12

B.
```
        . = 0
        ADDC(r31, N, r0)
        LD(r0, 8, r1)
        SRAC(r1, 4, r2)
        ST(r2, 4, r0)
        HALT()

        . = 0x2000
  N:    LONG(0x12345678)
        LONG(0xDEADBEEF)
        LONG(0xEDEDEDED)
        LONG(0x00000004)
```

Value left in R0? 0x2000
Value left in R1? 0xEDEDEDED

Value left in R2? 0xFEDEDEDE

Address of 32-bit memory location written by ST? 0x2004

Value found in 32-bit memory location with address 0? 0b11000000000011111001000000

C.
```
        . = 0
        LD(r31, X, r0)
        CMPLE(r0, r31, r1)
        BNE(r1, L1, r31)
        ADDC(r31, 17, r2)
        BEQ(r31, L2, r31)
  L1:   SUB(r31, r0, r2)
  L2:   XORC(r2, 0xFFFF, r2)     // be careful here!
        HALT()

        . = 0x1CE8
  X:    LONG(0x87654321)
```

Value left in R0? 0x87654321

Value left in R1? 1

Value left in R2? 0x87654320

Value uasm assigns to L1? 20

Value found in 32-bit memory location with address 8? 0b011101111110000100000000

D.
```
        . = 0
        ADDC(r31, 0, r0)
        LD(r31, N, r1)
        BEQ(r31, L3, r31)
  L1:   ANDC(r1, 1, r2)
        BEQ(r2, L2, r31)
        ADDC(r0, 1, r0)
  L2:   SHRC(r1, 1, r1)
  L3:   BNE(r1, L1, r31)
        HALT()

        . = 0x2468
  N:    LONG(0x8F2E3D4C)
```

Value left in R0? 17

Value left in R1? 0

Number of times instruction labeled L2 is executed? 32

Suppose that the instructions above were relocated so that the first instruction were at location 0x100 instead of location 0. Assuming we then started execution at location 0x100 and we wanted the instructions to perform the same computation, which instruction encodings should be changed when relocating the program?

Instructions that need to be changed?

no instructions need to be changed

E.
```
  . = 0
  BEQ(r31, L1, r0)
  ADDC(r0, 0, r0)
  L1: LD(r0, 0, r1)
  HALT()
```

Value left in R0? 4

Value left in R1? 0b11000000000000000000000000

Problem 2. Design Problem: Quicksort

See the instructions below.

Use the Bsim instance below to enter your code. To complete this design problem, you should assemble your program, then run the simulation to completion. The built-in test will either report any discrepencies between the expected and actual outputs, or, if your code is correct, it will record the test passed.
To enable the online system to check this answer, first run the design tests provided by the tool.
BSim window is open

Suppose you have an array of N integers, and wish to sort them into ascending order. There are a variety of approaches you might use, ranging from the maligned *bubble sort* which takes $O(N^2)$ time, to faster approaches such as *quicksort* that averages $O(N \log N)$ time. Quicksort uses an elegantly simple *divide and conquer* approach to sorting an array:

1. Pick a *pivot* value from among the array elements, and remove that element from the array.
2. Partition the array into two smaller arrays, containing elements whose values are smaller or larger than the pivot value, respectively.
3. Call quicksort recursively to sort each of the two smaller arrays.
4. Finally, combine the sorted smaller-value array, the pivot element, and the sorted larger-value array into a single sorted result.

This description glosses over several details which may vary by implementation; for example, which of the two smaller arrays should contain elements that are *equal* to the pivot value.

A high-level Python approach to quicksort of a list is:

```
def qsort1(list):
  if list == []: return []
  else:
      pivot = list[0]       # arbitrarily choose first element
      lesser = qsort1([x for x in list[1:] if x < pivot])
      greater = qsort1([x for x in list[1:] if x >= pivot])
      return lesser + [pivot] + greater
```

Often it is preferable to sort an array *in place*, rather than allocating space for the resulting new array. A version of quicksort for in-place sorting is given in Wikipedia as

```
# in-place partition of subarray
#   left is the index of the leftmost element of the subarray
#   right is the index of the rightmost element of the
#     subarray (inclusive)
def partition(array,left,right):
    # choose middle element of array as pivot
    pivotIndex = (left+right) >> 1
    pivotValue = array[pivotIndex]

    # swap array[right] and array[pivotIndex]
    # note that we already store array[pivotIndex] in pivotValue
    array[pivotIndex] = array[right]

    # elements <= the pivot are moved to the left (smaller indices)
    storeIndex = left
    for i in xrange(left,right):  # don't include array[right]
        temp = array[i]
        if temp <= pivotValue:
            array[i] = array[storeIndex]
            array[storeIndex] = temp
            storeIndex += 1

    # move pivot to its final place
    array[right] = array[storeIndex]
```

```
    array[storeIndex] = pivotValue
    return storeIndex

def quicksort(array, left, right):
    if left < right:
        pivotIndex = partition(array,left,right)
        quicksort(array,left,pivotIndex-1)
        quicksort(array,pivotIndex+1,right)
```

Although this code is nominally in Python, it is substantially identical to C code.

Note that the code for both `quicksort` and `partition` identifies a range of consecutive elements in the `array` by two integers, `left` and `right`, that identify the array indices of the leftmost and rightmost elements within the range respectively. In contrast to usual C and Python practice, the indicated range includes both the left and right elements: thus, the only way to designate an empty (zero-length) subrange is by having `right`<`left`.

Your job is to translate this in-place version of quicksort to Beta assembly language. The `template.uasm` tab in the BSim editor contains the appropriate checkoff setup and dummy (empty) definitions for the procedures `quicksort` and `partition`. It also includes the above Python/C versions of the code as comments.

Arrays are stored in consecutive locations of memory. In this lab, we're dealing with arrays of integers, so each array element occupies a 32-bit word. Since the memory is byte addressed, consecutive elements of an integer array have addresses that differ by 4. Integer arrays are always *word aligned*, *i.e.*, the byte addresses of array elements have `00` as the low-order two address bits. When a program refers to an entire array, the value that gets passed around is the address of the first array element, *i.e.*, the address of `array[0]`. For example, here's a simple procedure that adds two consecutive array elements:

```
  def add_pair(array,i): return array[i] + array[i+1]
```

The corresponding assembly language code would be

```
add_pair:
     PUSH(LP)       // standard entry sequence
     PUSH(BP)
     MOVE(SP,BP)
     PUSH(R1)       // save registers we use below

     LD(BP,-12,R1)  // R1 = address of array[0]
     LD(BP,-16,R0)  // R0 = i

     // now we have to convert index i into the appropriate
     // address offset from the start of the array.  Since
     // each array element occupies 4 bytes, we multiply the
     // index by 4 to convert it to a byte offset.
     SHLC(R0,2,R0)  // shift left by 2 = multiply by 4
     ADD(R0,R1,R1)  // R1 = address of array[i]

     LD(R1,0,R0)    // R0 = array[i]
     LD(R1,4,R1)    // R1 = array[i+1]
     ADD(R1,R0,R0)  // R0 = array[i] + array[i+1]

     POP(R1)        // restore saved registers
     MOVE(BP,SP)    // standard exit sequence
     POP(BP)
     POP(LP)
     JMP(LP)
```

Setup: The `Quicksort` tab in the BSim window has been initialized with the code from the `template.uasm` tab. Look it over briefly; notice that the dummy procedures included are valid

(*e.g.*, they obey stack discipline and our linkage conventions) but devoid of any function.

Select the `Quicksort` tab in the BSim window, click *Assemble* to translate it to binary. If successful, you'll now see a window onto an operating (simulated) Beta running the binary translation of your program. The upper-left displays the contents of the Beta's registers, the lower-left a region of memory containing the translated code, and to the right two regions of memory (one near the top of the stack. Explore this view a bit: scroll the lower-left region and observe labels, hex values, and (where sensible) values decoded as Beta instructions. Most of these are from the checkoff program and associated infrastructure; but at the end of the non-zero contents, you'll see your empty partition and quicksort procedures.

Run the code, by hitting the *play* button; you'll see the Beta stepping through instructions, updating the display.

Hint: You can click on the "Split" button to configure the BSim window to show both your editor windows and the Beta simulation display. The simulator will highlight the line in your source code that generated the current Beta instruction, assuming that source line came from the currently selected editor window. You can watch the Beta scroll through your program making it easy to watch your code being executed.
When you get tired of watching this show, hit *pause* and then *fast forward* to execute without the tedious display updates. You should see some output from the checkoff program in the *console* window at the bottom of your screen.

Good news - you've passed the first two test cases! Looking at the console output, notice that Test 1 involves a sequence of 1 element, and Test 2 involves a sequence of length 2 thats already in order. These trivial test cases are already sorted, so your do-nothing quicksort (which, in its defense, at least does no damage) passes these tests. The third test, which actually needs some sorting, should fail.

TestCase: Lets focus on the failed test: change the `LONG(0)` in the location labeled *TestCase* near the beginning of your *Quicksort* file to read `LONG(3)`. This tells the checkoff program to run only the third test, making debugging that test a bit less tedious.

Our principal debugging tool is the *breakpoint*, a marker that can be inserted into our program at interesting points to cause it to pause and let us poke around.

Insert the line

```
.breakpoint
```

in your empty `quicksort` procedure, immediately following the line reading

```
// Fill in your code here...
```

and run it again. The simulation should stop when you hit the inserted breakpoint, allowing you to examine the state of the Beta and its memory at that point.

A. Looking at the state of the Beta, determine the value in `R20`.
What is the value in R20? `0x14`

In fact, you'll notice systematic values in registers; these are used by the checkoff program to check that you're properly saving/restoring registers.

Your next project, and the main task of this lab, is to fill in the code for the two procedures and get them working correctly. Although you may code these procedures however you like, we recommend paying careful attention to details in the Python/C versions supplied; seemingly minor variations (< rather than <=, including/excluding the rightmost element in a loop) can cause errors taking hours to debug.

While you may vary the order in which you approach the coding of these procedures, we suggest the following sequence of steps:

quicksort stub: Implement the first part of `quicksort`: the call to `partition`. Using a breakpoint in `partition`, ensure that its called with the proper arguments (identical to those of `quicksort`).

Try it on the first few test cases by varying the value in `TestCase`.

partition: Implement the code in `partition`, and debug it (again, using various test cases). Although this code is more complex than `quicksort`, it avoids the complication of recursive calls, making debugging easier. Once you're convinced `partition` works properly, move on to the next step.

Hint: The `partition` code involves a number of local variables. Although you can allocate these variables in the stack frame (as we have done in examples given in lecture), your code may be both smaller and more readable if you allocate registers to hold local variables. One convenient way to do this is to define symbolic names, *e.g.*,

```
p_array=R2          // base address of array (arg 0)
p_left=R3
p_right=R4
p_pivotIndex=R5     // Corresponds to PivotIndex in C program
p_pivotValue=R6
p_storeIndex=R7
...
```

Note that the `p_` prefixes are prepended to avoid name conflicts with other such assignments. Of course, you must remember to save and restore the values of any register you use in this (or any other) fashion!

Even if you choose to store these variables in the local stack frame, you will likely find symbolic names (defined as the variable offsets) easier to use than generic `Rx` register names.

Remember: A procedure can use `R0` with impunity, since the caller is expecting that to change to contain the return value. But if you use any other register (`R1`, `R2`, ...) it must have the same value after the procedure as it did when the procedure is called. For each register `Rx` that the procedure uses, there should be a `PUSH(Rx)` in the entry sequence and a `POP(Rx)` at the corresponding part of the exit sequence.

quicksort: Complete the `quicksort` procedure, and get it to run correctly on all the test cases by setting the value in `TestCase` to zero.

Stack crawling: The `left` and `right` values passed to `quicksort` are described as indices of `array` elements, implying that they range from zero through one less than the length of `array`. This constraint turns out to be not strictly true in all cases in our implementation. As our last exercise, we'll explore an exception that arises during execution of our `quicksort` implementation.

Insert a breakpoint in your `quicksort` procedure, after the stack frame has been set up and interesting values (*e.g.*, arguments) have been loaded into registers. Set `TestCase` to contain 13 (running only the last test case), and run your code.

When it stops at your breakpoint, check the value that was passed as the `right` argument. If its non-negative, click *fast forward* to continue until you hit the breakpoint again; click again, and keep clicking until you get to the breakpoint with a ***negative*** value for `right`.

At this point, you are several calls deep in the recursion of quicksort. By inspecting values in registers and on the stack, you can determine both the current state of the computation and the call history that led us here.

B. Find the two arguments to the current call:

Argument "left" in current call: 0

Argument "right" in current call: -1

C. What's the current value in element zero of the array?

Value in array[0]: 0xA

Note the relation between the element zero value and the other values in the array. If you look a bit at

the code and think about the behavior of your two procedures, you can see how the negative value arises and convince yourself that it does no harm to the computation. It is, however, a bit sloppy for the author of this code not to have documented this anomaly in an explanatory comment!

BP Chain: Observe where BP points into your stack frame, and the location of the saved LP value relative to this point. Scroll the disassembly window of BSim and find the two recursive calls within `quicksort`; note the hex locations of the instructions *following* each call. Write down these values; they will be saved LP values in stack frames associated with recursive calls, and can be used to distinguish recursive calls from the original call by the checkoff code.

D. By inspecting the saved LP value, determine whether the current recursive call to `quicksort` is from the first recursive call (sorting the array of smaller elements) or the second within `quicksort`.

Which recursive call? first (smaller elements)

E. Find the saved BP, which points to the stack frame for the prior call to `quicksort`; find the arguments to that call.

Argument "left" in prior call 0
Argument "right" in prior call 4

F. Follow the BP chain back to the original call, and report the recursion depth (number of active calls) when the negative argument is encountered. Note that a couple of frames at the bottom of the stack belong to procedures in the testing code -- we only want you to count frames belonging to quicksort. Think about how you can distinguish quicksort frames from test code frames.

Number of active quicksort calls 5