Stacks and Procedures: 1

12 points possible (ungraded)

Harry Hapless is a friend struggling to finish his Lab; knowing that you completed it successfully, he asks your help understanding the operation of the quicksort procedure, which he translated from the Python code given in the lab handout:

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

You recall from your lab that each of the three arguments and the local variable are 32-bit binary integers. You explain to Harry that quicksort returns no value, but is called for its effect on the contents of a region of memory dictated by its argument values. Harry asks some questions about the possible effect of the call **quicksort(0×1000, 0×10, 0×100)**:

0

9

0

$2F0$

$94C$

$F94$

1

2

3

4

8

0

$2F0$

$F24$

FCC

$2F0$

0

9

9
8
7
$2F0$
$F48$
$FF0$
$BP \rightarrow$ $2F0$
0
8
6
$SP \rightarrow$

```
quicksort:
        PUSH(LP)
        PUSH(BP)
        MOVE(SP, BP)
        PUSH(R1)
        PUSH(R2)
        PUSH(R3)
        PUSH(R4)

        LD(BP, -12, R1)
        LD(BP, -16, R2)
aa:     LD(BP, -20, R3)

        CMPLT(R2, R3, R0)
        BF(R0, qx)

        PUSH(R3)
        PUSH(R2)
        PUSH(R1)
        BR(partition, LP)
        DEALLOCATE(3)
        MOVE(R0, R4)
xx:
        SUBC(R4, 1, R0)
        PUSH(R0)
        PUSH(R2)
        PUSH(R1)
        BR(quicksort, LP)
        DEALLOCATE(3)

        PUSH(R3)
        ADDC(R4, 1, R0)
        PUSH(R0)
        PUSH(R1)
        BR(quicksort, LP)
bb:     DEALLOCATE(3)

qx:     POP(R4)
        POP(R3)
        POP(R2)
        POP(R1)
cc:     MOVE(BP, SP)
        POP(BP)
        POP(LP)
        JMP(LP)
```

1. Given the above call to **quicksort**, what is the region of memory locations (outside of the stack) that might be changed?
   Lowest memory address possibly effected: 0x

   Highest memory address possibly effected: 0x

   Harry's translation of quicksort to Beta assembly language appears above on the right.
2. What register did Harry choose to hold the value of the variable **pivotIndex**?
   Register holding pivotIndex value: R

   After loading and assembling this code in BSim, Harry has questions about its translation to binary.
3. Give the hex value of the 32-bit machine instruction with the tag **aa** in the program to the right.
   Hex translation of instruction at aa: 0x

Harry tests his code, which seems to work fine. He questions whether it could be shortened by simply eliminating certain instructions.

4. Would Harry’s quicksort continue to work properly if the instruction at **bb** were eliminated? If the instruction at **cc** were eliminated? Indicate which, if any, of these instructions could be deleted.

OK to delete instruction at bb?

○ Yes

○ No

OK to delete instruction at cc?

○ Yes

○ No

Harry runs his code on one of the Lab test cases, which executes a call to **quicksort(Y, 0, X)** via a **BR(quicksort, LP)** at address **0×948**. Harry halts its execution just as the instruction following the **xx** tag is about to be executed. The contents of a region of memory containing the topmost locations on the stack is shown to the right.

5. What are the arguments to the current quicksort call? Use the stack trace shown above to answer this question.

Arguments: array = 0x

left = 0x

right = 0x

6. What is the value **X** in the original call **quicksort(Y, 0, X)**?

Value of X in original call: 0x

7. What were the contents of R4 when the original call to **quicksort(Y, 0, X)** was made?
   Contents of R4 at original call: 0x

8. What is the address of the instruction tagged **bb:** in the program?
   HEX value of bb: 0x

Submit

Stacks and Procedures: 2

11 points possible (ungraded)

The following C program implements a function (ffo) of two arguments, returning an integer result. The assembly code for the procedure is shown on the right, along with a partial stack trace showing the execution of **ffo(0xDECAF,0)**. The execution has been halted just as the Beta is about to execute the instruction labeled **rtn**, i.e., the value of the Beta's program counter is the address of the first instruction in POP(R1). In the C code below, note that "v>>1" is a logical right shift of the value v by 1 bit.

```
// bit position of left-most 1
int ffo(unsigned v, int b) {
    if (v == 0) ???;
    else return ffo(v>>1,b+1);
}
```

1. Examining the assembly language for ffo, what is the appropriate C code for ??? in the C representation for ffo?
   C code for ???:

   ○ return v

○ return b

○ return 0

○ return ffo(v>>1,b)

$0x000F$
$0x001B$
$0x0208$
$0x012C$
$0x001B$
$0x0010$
$0x000D$
$0x0208$
$0x0140$
$0x000D$
$0x0011$
$0x0006$
$0x0208$
$0x0154$
$BP \rightarrow 0x0006$
$0x0012$
$0x0003$

```
ffo:     PUSH(LP)
         PUSH(BP)
         MOVE(SP,BP)
         PUSH(R1)

         LD(BP,-16,R0)
         LD(BP,-12,R1)
xxx:     BEQ(R1,rtn)

         ADDC(R0,1,R0)
         PUSH(R0)
         SHRC(R1,1,R1)
         PUSH(R1)
         BR(ffo,LP)
         DEALLOCATE(2)

rtn:     POP(R1)
         MOVE(BP,SP)
         POP(BP)
         POP(LP)
         JMP(LP)
```

2. What value will be returned from the procedure call ffo(3,100)?

   Value returned from procedure call ffo(3,100):

3. What are the values of the arguments in the call to ffo from which the Beta is about to return? Please express the values in hex or write “CAN'T TELL” if the value cannot be determined.

   Value of argument v or “CAN'T TELL”: 0x

   Value of argument b or “CAN'T TELL”: 0x

4. Determine the specified values at the time execution was halted. Please express each value in hex or write “CAN'T TELL” if the value cannot be determined.

   Value in R1 or “CAN'T TELL”: 0x

   Value in BP or “CAN'T TELL”: 0x

Value in LP or “CAN'T TELL”: 0x

Value in SP or “CAN'T TELL”: 0x

Value in PC or “CAN'T TELL”: 0x

5. What is the address of the BR instruction for the original call to ffo(0xDECAF,0)? Please express the value in hex or “CAN'T TELL”.
   Address of the original BR, or “CAN'T TELL”: 0x

6. A 6.004 student modifies ffo by removing the DEALLOCATE(2) macro in the assembly compilation of the ffo procedure, reasoning that the MOVE(BP,SP) will perform the necessary adjustment of stack pointer. She runs a couple of tests and verifies that the modified ffo procedure still returns the same answer as before. Does the modified ffo obey our procedure call and return conventions?
   Does modified ffo obey call/return conventions?
   Select an option

Submit

Stacks and Procedures: 3

13/13 points (ungraded)

It was mentioned in lecture that recursion became a popular programming construct following the adoption of the stack as a storage allocation mechanism, ca. 1960. But the Greek mathematician Euclid, always ahead of his time, used recursion in 300 BC to compute the greatest common divisor of two integers. His elegant algorithm, translated to C from the ancient greek, is shown below:

```
int gcd(int a, int b) {
   if (a == b) return a;
   if (a > b) return gcd(a-b, b);
   else return gcd(a, b-a);
}
```

The procedure **gcd(a, b)** takes two positive integers **a** and **b** as arguments, and returns the greatest positive integer that is a factor of both **a** and **b**.

Note that the base case for this recursion is when the two arguments are equal (== in C tests for equality), and that there are two recursive calls in the body of the procedure definition.

Although Euclid's algorithm has been known for millennia, a recent archeological dig has uncovered a new document which appears to be a translation of the above C code to Beta assembly language, written in Euclid's own hand. The Beta code is known to work properly, and is shown below.

```
gcd:    PUSH(LP)
        PUSH(BP)
        MOVE(SP, BP)
        PUSH(R1)
        PUSH(R2)
        LD(BP, -12, R0)
        LD(BP, -16, R1)
        CMPEQ(R0, R1, R2)
        BT(R2, L1)
        CMPLE(R0, R1, R2)
xxx:    BT(R2, L2)
        PUSH(R1)
        SUB(R0, R1, R2)
        PUSH(R2)
        BR(gcd, LP)
        DEALLOCATE(2)
        BR(L1)

L2:     SUB(R1, R0, R2)
        PUSH(R2)
        PUSH(R0)
        BR(gcd, LP)
        DEALLOCATE(2)

L1:     POP(R2)
        POP(R1)
yyy:    MOVE(BP, SP)
        POP(BP)
        POP(LP)
        JMP(LP)
```

1. Give the 32-bit binary translation of the **BT(R2,L2)** instruction at the label **xxx**

opcode (6 bits): 0b

011101 ✓

Rc (5 bits): 0b

11111 ✓

Ra (5 bits): 0b

00010 ✓

literal (16 bits): 0b

0000000000001000 ✓

2. One historian studying the code, a Greek major from Harvard, questions whether the **MOVE(BP, SP)** instruction at **yyy** is really necessary. If this instruction were deleted from the assembly language source and re-translated to binary, would the shorter Beta program still work properly?

Still works?

Yes ✓

```
main:   CMOVE(0x104, SP)
        PUSH(R17)
        PUSH(R18)
        BR(gcd, LP)
zzz:    HALT()
```

At a press conference, the archeologists who discovered the Beta code give a demonstration of it in operation. They use the test program shown above to initialize SP to hex **0×104**, and call gcd with two positive integer arguments from **R17** and **R18**. Unfortunately, the values in these registers have not been specified.

Address in Hex	**Data in Hex**
100 :	104
104 :	18
108 :	9
10C :	D8
110 :	D4
114 :	EF
118 :	BA
11C :	F
120 :	9
124 :	78
128 :	114
12C :	18
130 :	F
134 :	6
138 :	9

$13C: \quad 78$

$140: \quad 12C$

$144: \quad F$

$148: \quad 6$

$14C: \quad 6$

$150: \quad 3$

$154: \quad 58$

$158: \quad 144$

$SP \rightarrow 15C: \quad 6$

They start their program on a computer designed to approximate the computers of Euclid's day (think of Moore's law extrapolated back to 300 BC!), and let it run for a while. Before the call to gcd returns, they stop the computation just as the instruction at **yyy** is about to be executed, and examine the state of the processor.

They find that **SP** (the stack pointer) contains **0×15C**, and the contents of the region of memory containing the stack as shown **(in HEX)** to the right.

You note that the instruction at **yyy**, about to be executed, is preparing for a return to a call from gcd(a,b).

3. What are the values of **a** and **b** passed in the call to gcd which is about to return? Answer in HEX.

Args to current call: a=0x

3 ✓

b = 0x

6 ✓

4. What are the values of **a** and **b** passed in the *original* call to gcd, from registers **R17** and **R18**? Answer in HEX.

Args to original call: a=0x

9 ✓

b = 0x

18 ✓

5. What is the address corresponding to the tag **zzz**: of the **HALT()** following the original call to **gcd**?

Address of zzz: (HEX): 0x

D8 ✓

6. What is the address corresponding to the tag **L1**: in the assembly b for **gcd**?
   Address of L1: (HEX): 0x

   7C ✓

7. What value will be returned (in R0) as the result of the original call to **gcd**?
   Value returned to original caller: (HEX): 0x

   3 ✓

8. What was the value of R2 at the time of the original call to gcd?
   Original value in R2: (HEX): 0x

   BA ✓

Submit

Stacks and Procedures: 4

15/15 points (ungraded)

You are given the following listing of a C program and its translation to Beta assembly code:

```
int f(int x, int y)
  int a = x - 1; b = x + y;
  if (x == 0) return y;
  return f(a, ???)
```

```
f:    PUSH(LP)
      PUSH(BP)
mm:   MOVE(SP, BP)
      PUSH(R1)
      PUSH(R2)
      LD(BP, -16, R0)
yy:   LD(BP, -12, R1)
      BEQ(R1, xx)
      SUBC(R1, 1, R2)
      ADD(R0, R1, R1)
      PUSH(R1)
      PUSH(R2)
      BR(f, LP)
zz:   DEALLOCATE(2)
      LD(BP, -16, R1)
      ADD(R1, R0, R0)
      PUSH(R0)
ww:   PUSH(R2)
      BR(f, LP)
      DEALLOCATE(2)
xx:   POP(R2)
      POP(R1)
      POP(BP)
      POP(LP)
      JMP(LP)
```

1. Fill in the binary value of the **LD** instruction stored at the location tagged **yy** in the above program.

opcode (6 bits): 0b

011000 ✓

Rc (5 bits): 0b

00001 ✓

Ra (5 bits): 0b

11011 ✓

literal (16 bits): 0b

1111111111110100 ✓

2. Suppose the MOVE instruction at the location tagged **mm** were eliminated from the above program. Would it continue to run correctly?

Still works fine?

○ Yes

○ Can't Tell

● No

✓

3. What is the missing expression designated by **???** in the C program above.

○ b

○ y

● y + f(a,b)

○ f(a,b)

✓

The procedure f is called from location 0xFC and its execution is interrupted during a recursive call to f, just prior to the execution of the instruction tagged **xx**. The contents of a region of memory, including the stack, are shown to the left.

NB: All addresses and data values are shown in hex. The **BP** register contains **0×494**, and **SP** contains **0×49C**.

Address in Hex	**Contents in Hex**
448	2
44C	4
450	7
454	3
458	2
45C	100
460	D4
464	3
468	4
46C	5

470	1
474	50
478	???
$47C$	5
480	1
484	B
488	0
$48C$	70
490	$47C$
$BP \rightarrow 494$	5
498	0
$SP \rightarrow 49C$	

4. What are the arguments to the *most recent* active call to **f**?

Most recent arguments (HEX): x = 0x

0 ✓

y = 0x

B ✓

5. What value is stored at location **0×478**, shown as **???** in the listing to the left?

Contents 0×478 (HEX): 0x

464 ✓

6. What are the arguments to the *original* call to **f**?

Original arguments (HEX): x = 0x

2 ✓

y = 0x

3 ✓

7. What value is in the **LP** register?

Contents of LP (HEX): 0x

70 ✓

8. What value was in **R1** at the time of the original call?

Contents of R1 (HEX): 0x

3 ✓

9. What value is in **R0**?

   Value currently in R0 (HEX): 0x

   B ✓

10. What is the hex address of the instruction tagged **ww**

    Address of ww (HEX): 0x?

    64 ✓

Submit

Stacks and Procedures: 5

16/17 points (ungraded)

The **wfps** procedure determines whether a string of left and right parentheses is well balanced, much as your Turing machine of Lab 4 did. Below is the code for the **wfps** ("well-formed paren string") procedure in C, as well as its translation to Beta assembly code.

```
int STR[100];                 // string of parens

int wfps(int i,               // current index in STR
  int n)                      // LPARENs to balance

{ int c = STR[i];             // next character
  int new_n;                  // next value of n
  if (c == 0)                 // if end of string,
    return (n == 0);          //   return 1 iff n == 0
  else if (c == 1)            // on LEFT PAREN,
    new_n = n+1;              //    increment n
  else {                      // else must be RPAREN
    if (n == 0) return 0;     // too many RPARENS!
      xxxxx; }                // MYSTERY CODE!
  return wfps(i+1, new_n);    // and recurse.
}
```

```
STR:  . = .+4*100
wfps: PUSH(LP)
      PUSH(BP)
      MOVE(SP, BP)
      ALLOCATE(1)
      PUSH(R1)
      LD(BP, -12, R0)
      MULC(R0, 4, R0)
      LD(R0, STR, R1)
      ST(R1, 0, BP)
      BNE(R1, more)
      LD(BP, -16, R0)
      CMPEQC(R0, 0, R0)
rtn:  POP(R1)
      MOVE(BP, SP)
      POP(BP)
      POP(LP)
      JMP(LP)
more: CMPEQC(R1, 1, R0)
      BF(R0, rpar)
      LD(BP, -16, R0)
      ADDC(R0, 1, R0)
      BR(par)
rpar: LD(BP, -16, R0)
      BEQ(R0, rtn)
      ADDC(R0, -1, R0)
par:  PUSH(R0)
      LD(BP, -12, R0)
      ADDC(R0, 1, R0)
      PUSH(R0)
      BR(wfps, LP)
      DEALLOCATE(2)
      BR(rtn)
```

wfps expects to find a string of parentheses in the integer array stored at **STR**. The string is encoded as a series of **32-bit integers** having values of

- **1** to indicate a left paren,
- **2** to indicate a right paren, or
- **0** to indicate the end of the string.

These integers are stored in consecutive 32-bit locations starting at the address **STR**.

wfps is called with two arguments:

1. The first, **i**, is the index of the start of the part of **STR** that this call of **wfps** should examine. Note that indexes start at 0 in C. For example, if **i** is 0, then **wfps** should examine the entire string in **STR** (starting at the first character, or **STR[0]**). If **i** is

4, then **wfps** should ignore the first four characters and start examining **STR** starting at the fifth character (the character at **STR[4]**).

2. The second argument, **n**, is zero in the original call; however, it may be nonzero in recursive calls.

wfps returns 1 if the part of **STR** being examined represents a string of balanced parentheses if **n** additional left parentheses are prepended to its left, and returns 0 otherwise.

Note that the compiler may use some simple optimizations to simplify the assembly-language version of the code, while preserving equivalent behavior.

The C code is incomplete; the missing expression is shown as **xxxx**.

1. Fill in the binary value of the instruction stored at the location tagged **more** in the above assembly-language program.

   opcode (6 bits): 0b

   110100 ✓ Answer: 110100

   Rc (5 bits): 0b

   00000 ✓ Answer: 00000

   Ra (5 bits): 0b

   00001 ✓ Answer: 00001

   literal (16 bits): 0b

   0000000000000 ✓ Answer: 0000000000000001

   Explanation
   The instruction tagged **more** is **CMPEQC(R1, 1, R0)**.
   The opcode for the **CMPEQC** instruction is 110100.
   Register Rc is R0, or 00000 when encoded using 5 bits.
   Register Ra is R1, or 00001.
   The literal is 1 using 16 bits of binary.

2. Is the variable c from the C program stored as a local variable in the stack frame?

● Yes

○ No

✓

If so, give its (signed) offset from BP; else select “NA”.

○ $BP - 16$

○ $BP - 12$

○ $BP - 8$

● $BP + 0$

○ $BP + 4$

○ $BP + 8$

○ NA

✓

Explanation

The stack frame for **wfps** looks like this:

$$\begin{array}{rl} & n \\ & i \\ & LP \\ & BP \\ BP \rightarrow & c \\ & R1 \end{array}$$

This shows that to access the variable C, we want to access the location that BP points to without adding any offset.

3. Is the variable new_n from the C program stored as a local variable in the stack frame?

○ Yes

● No

✓

If so, give its (signed) offset from BP; else select “NA”.

○ $BP - 16$

○ $BP - 12$

○ $BP - 8$

○ $BP + 0$

○ $BP + 4$

○ $BP + 8$

● NA

✓

Explanation
new_n is not stored as a local variable in the stack frame. It is passed as the next value of n in the recursive calls to **wfps**.

4. What is the missing C source code represented by xxxxx in the given C program?

○ n = n + 1

○ n = n - 1

○ new_n = n + 1

● new_n = n - 1

○ new_n = n

✓

Explanation
The **rpar:** portion of code implements the final **else** statement. It first loads n into R0 and returns 0 if n == 0. Otherwise it subtracts 1 from n and pushes that onto the stack as the new value of variable n for the next recursive call. So new_n = n - 1.

The procedure **wfps** is called from an external procedure and its execution is interrupted during a recursive call to **wfps**, just prior to the execution of the instruction labeled **rtn**. The contents of a region of memory are shown below. At this point, **SP** contains 0×1D8, and **BP** contains 0×1D0.

NOTE: All addresses and data values are shown in hexadecimal.

Address in Hex Contents in Hex

188:	7
18C:	4A8
190:	0
194:	0
198:	458
19C:	D4
1A0:	1
1A4:	D8
1A8:	1
1AC:	1
1B0:	3B8
1B4:	1A0
1B8:	2
1BC:	1
1C0:	0
1C4:	2
1C8:	3B8
1CC:	1B8
BP⟶1D0:	2
1D4:	2
SP⟶1D8:	0

5. What are the arguments to the *most recent* active call to **wfps**?

Most recent arguments (HEX): i = 0x

2

✓ Answer: 2

n = 0x

0

✓ Answer: 0

Explanation

The remaining questions are easier to answer if we label the stack with the stack frame elements that each location represents.

Address in Hex	**Contents in Hex**	
188:	7	
18C:	4A8	
190:	0	n

194:	0	i
198:	458	LP
19C:	$D4$	BP
1A0:	1	c
1A4:	$D8$	R1
1A8:	1	n
1AC:	1	i
1B0:	$3B8$	LP
1B4:	$1A0$	BP
1B8:	2	c
1BC:	1	R1
1C0:	0	n
1C4:	2	i
1C8:	$3B8$	LP
1CC:	$1B8$	BP
BP→1D0:	2	c
1D4:	2	R1
SP→1D8:	0	

The arguments for the most recent call to **wfps** are at address 1C0 and 1C4, i = 2 and n = 0.

6. What are the arguments to the *original* call to **wfps**?

Original arguments (HEX): i = 0x

0 ✓ Answer: 0

n = 0x

0 ✓ Answer: 0

Explanation

There are three stack frame instances shown on the stack. The two that have LP = 0×3B8 are from the recursive call to **wfps**, whereas the top one where LP = 0×458 is from the original call to **wfps**. This means that the arguments for the original call are the ones at addresses 0×190 and 0×194, or i = 0 and n = 0.

7. What value is in **R0** at this point?

Contents of R0 (HEX): 0x

0 ✓ Answer: 0

Explanation

In the most recent active call to **wfps**, the arguments are i = 2 and n = 0. From the current stack frame we also see that the current c = 2 and its stored at address 0×1D0. Since c != 0, we enter the else if portion of the code which is at label **more**. There R0 is assigned to the result of checking if c == 1. Since it doesn't R0 is set to 0, and the code branches to label **rpar** to check for a right parenthesis. There n is loaded into R0, so again R0 = 0 and we then branch to label **rtn**. So R0 = 0 when execution is stopped.

8. How many parens (left and right) are in the string stored at STR (starting at index 0)? Give a number, or “**CAN’T TELL**” if the number can’t be determined from the given information.

Length of string, or “CAN’T TELL”:

○ 0

○ 1

○ 2

○ 3

● Can't Tell

✓

Explanation

There is no way to tell the actual number of parentheses in the string because we don't have the entire history of execution of the **wfps** routine.

9. What is the hex address of the instruction tagged **par**?

Address of par (HEX): 0x

39C ✓ Answer: 39C

Explanation

The LP value stored in the recursive calls to **wfps** points to the DEALLOCATE(2) instruction. This value is 0×3B8. Below is a table showing the addresses of the instructions beginning at label par and ending at the DEALLOCATE(2) instruction. Recall that PUSH instructions are actually a macro consisting of two instructions so they take up 8 bytes instead of 4.

39C3A43A83AC3B43B8

HEX address	instruction
	PUSH(R0)
	LD(BP, -12, R0)
	ADDC(R0, 1, R0)

PUSH(R0)

BR(wfps, LP)

DEALLOCATE(2)

So the address of **par** is 0×39C.

10. What is the hex address of the **BR** instruction that called **wfps** originally?

Address of original call (HEX): 0x

0×454

✗ Answer: 454

Explanation

Since the original LP = 0×458 and that points at the instruction immediately following the original branch instruction, that means that the original BR instruction is at 0×454.

Submit

ℹ Answers are displayed within the problem

Stacks and Procedures: 6

13 points possible (ungraded)

You've taken a summer internship with BetaSoft, the worlds largest supplier of Beta software. They ask you to help with their library procedure **sqr(j)**, which computes the square of a non-negative integer argument **j**. Because so many Betas don't have a multiply instruction, they have chosen to compute **sqr(j)** by adding up the first **j** odd integers, using the C code below and its translation to Beta assembly language to the left.

```
int sqr(j) {
   int s = 0;
   int k = j;
   while (k != 0) {
      s = s + nthodd(k);
      k = k - 1;
   }
   return s;
}

int nthodd(n) {
   if (n == 0) return 0;
   return ???;
}
```

You notice that the **sqr** procedure takes an integer argument j, and declares two local integer variables s and k (initialized to zero and j, respectively).

The body of **sqr** is a loop that is executed repeatedly, decrementing the value of k at each iteration, until k reaches zero. Each time through the loop, the local variable s incremented by the value of the kth odd integer, a value that is computed by an auxiliary procedure **nthodd**.

1. What is the missing expression shown as **???** in the C code defining **nthodd** above?
   What is the missing expression denoted ??? in above C code:

```
sqr:     PUSH (LP)
         PUSH (BP)
         MOVE (SP, BP)
         ALLOCATE(2)
         PUSH (R1)
         ST(R31, 0, BP)
         LD (BP, -12, R0)
         ST(R0, 4, BP)
loop:    LD(BP, 4, R0)
         BEQ(R0, done)
         PUSH(R0)
         SUBC(R0, 1, R0)
         ST(R0, 4, BP)
         BR(nthodd, LP)
         DEALLOCATE(1)
         LD(BP, 0, R1)
         ADD(R0, R1, R1)
         ST(R1, 0, BP)
         BR(loop)
done:    LD(BP, 0, R0)
         POP(R1)
         DEALLOCATE(2)
         MOVE(BP, SP)
         POP(BP)
         POP(LP)
         JMP(LP)

nthodd:  PUSH (LP)
         PUSH (BP)
         MOVE (SP, BP)
         LD (BP, -12, R0)
         BEQ(R0, zero)
         ADD(R0, R0, R0)
         SUBC(R0, 1, R0)
zero:    MOVE(BP, SP)
         POP(BP)
         POP(LP)
         JMP(LP)
```

2. What variable in the C code, if any, is loaded into R0 by the LD instruction tagged **loop**? Answer "none" if no such value is loaded by this instruction.
   Value loaded by instruction at loop:, or “none”:

   Using a small test program to run the above assembly code, you begin computing **sqr(X)** for some positive integer **X**, and stop the machine during its execution. You notice, from the value in the PC, that the instruction tagged **zero** is about to be executed. Examining memory, you find the following values in a portion of the area reserved for the Beta's stack.

F0: $F4$
F4: 5
F8: EC
FC: $D4$
100: 15
104: 1
108: $DECAF$
10C 2
110 $4C$
114 100
BP118: $\rightarrow$ 0

NB: All values are in HEX! Give your answers in hex, or write "CAN'T TELL" if you can't tell.

You notice that BP contains the value **0×118**.

3. What argument (in hex) was passed to the current call to **nthodd**? Answer "CAN'T TELL" if you can't tell.
   HEX Arg to nthodd, or "CAN'T TELL": 0x
4. What is the value **X** that was passed to the original call to **sqr(X)**? Answer "CAN'T TELL" if you can't tell.
   HEX Arg X to sqr, or "CAN'T TELL": 0x
5. What is the hex value in **SP?** Answer "CAN'T TELL" if you can't tell.
   HEX Value in SP, or "CAN'T TELL": 0x
6. What is the current value of the variable **k** in the C code for sqr? Answer "CAN'T TELL" if you can't tell.
   HEX Value of k in sqr, or "CAN'T TELL": 0x
7. The test program invoked **sqr(X)** using the instruction BR(sqr,LP). What is the address of that instruction? Answer "CAN'T TELL" if you can't tell.
   HEX Address of BR instruction that called sqr, or "CAN'T TELL": 0x

8. What value was in R1 at the time of the call to **sqr(X)**? Answer “CAN'T TELL” if you can’t tell.

   HEX Value in R1 at call to sqr, or “CAN'T TELL”: 0x

   Your boss at BetaSoft, Les Ismoore, suspects that some of the instructions in the Beta code could be eliminated, saving both space and execution time. He hands you an annotated listing of the code (shown below), identical to the original assembly code but with some added tags.

```
sqr:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        ALLOCATE(2)
        PUSH (R1)
        ST(R31, 0, BP)
        LD (BP, -12, R0)
        ST(R0, 4, BP)
loop:   LD(BP, 4, R0)
        BEQ(R0, done)
        PUSH(R0)
        SUBC(R0, 1, R0)
        ST(R0, 4, BP)
        BR(nthodd, LP)
        DEALLOCATE(1)
        LD(BP, 0, R1)
        ADD(R0, R1, R1)
        ST(R1, 0, BP)
        BR(loop)
done:   LD(BP, 0, R0)
        POP(R1)
q1:     DEALLOCATE(2)
        MOVE(BP, SP)
        POP(BP)
        POP(LP)
        JMP(LP)
nthodd: PUSH (LP)
q5:     PUSH (BP)
q2:     MOVE (SP, BP)
        LD (BP, -12, R0)
        BEQ(R0, zero)
        ADD(R0, R0, R0)
        SUBC(R0, 1, R0)
zero:   MOVE(BP, SP)
        POP(BP)
        POP(LP)
        JMP(LP)
```

Les proposes several optimizations, each involving just the deletion of one or more instructions from the annotated code. He asks, in each case, whether the resulting code would still work properly. For each of the following proposed deletions, select “OK” if the code would still work after the proposed deletion, or “NO” if not. For each question, **assume that the proposed deletion is the ONLY change** (i.e., you needn’t consider combinations of proposed changes).

9. Delete the instruction tagged **q1**.
   Proposed deletion OK or NO?

○ OK

○ NO

10. Delete the instruction tagged **q2**.
    Proposed deletion OK or NO?

    ○ OK

    ○ NO

11. Delete the instruction tagged **loop**.
    Proposed deletion OK or NO?

    ○ OK

    ○ NO

12. Delete the instruction tagged **zero**.
    Proposed deletion OK or NO?

    ○ OK

    ○ NO

After some back-and-forth with Les, he proposes to replace **nthodd** with a minimalist version that avoids much of the standard procedure linkage boilerplate:

```
nthodd: LD (SP, NNN, R0)
        BEQ(R0, zero)
        ADD(R0, R0, R0)
        SUBC(R0, 1, R0)
zero:   JMP(LP)
```

He's quite sure this code will work, but doesn't know the appropriate value for **NNN**.

13. What is the proper value for the constant **NNN** in the shortened version of **nthodd**?

Appropriate value for NNN (in decimal):

Submit

Stacks and Procedures: 7

15 points possible (ungraded)

You are given the following listing of a C program and its translation to Beta assembly code:

```
// Mystery function:
int f(int x, int y) {
  int a = (x+y) >> 1;
  if (a == 0) return y;
  else return ???;
}
```

```
f:    PUSH(LP)
      PUSH(BP)
      MOVE(SP, BP)
      PUSH(R1)
      PUSH(R2)
      LD(BP, -12, R1)
      LD(BP, -16, R0)
      ADD(R0, R1, R2)
      SRAC(R2, 1, R2)

xx:   BEQ(R2, bye)

      SUB(R1, R2, R1)
      PUSH(R1)
      PUSH(R0)

yy:   BR(f, LP)
      DEALLOCATE(2)
      ADD(R2, R0, R0)

bye:  POP(R2)
      POP(R1)
zz:   MOVE(BP, SP)
      POP(BP)
      POP(LP)
      JMP(LP)
```

(Recall that a >> b means a shifted b bits to the right, propagating – ie, preserving -- sign)

1. Fill in the binary value of the BR instruction stored at the location tagged **yy** in the above program.
   opcode (6 bits): 0b

   Rc (5 bits): 0b

   Ra (5 bits): 0b

   literal (16 bits): 0b

2. Suppose the MOVE instruction at the location tagged **zz** were eliminated from the above program. Would it continue to run correctly?
   Still works fine?
   - ○ YES
   - ○ NO

3. What is the missing expression designated by **???** in the C program above.
   - ○ f(y, a)
   - ○ a + f(y, x)
   - ○ a + f(y, x-a)
   - ○ f(x, -a)
   - ○ f(y, -a)

The procedure **f** is called from an external procedure and its execution is interrupted during a recursive call to **f**, just prior to the execution of the instruction tagged **bye**. The contents of a region of memory are shown below.

NB: All addresses and data values are shown in hex. The **BP** register contains **0×250**, **SP** contains **0×258**, and **R0** contains **0×5**.

```
204:    CC
208:    4
20C:    7
210:    6
214:    7
218:    E8
21C:    D4
220:    BAD
224     BABE
228     1
```

22C	6
230	54
234	
238	1
23C	6
240	3
244	1
248	54
24C	238
BP 250: $\rightarrow$	3
254	3
SP 258: $\rightarrow$	-1

4. What are the arguments to the *most recent* active call to **f**?
   Most recent arguments (HEX): x = 0x

   y = 0x
5. Fill in the missing value in the stack trace.
6. What are the arguments to the *original* call to **f**?
   Original arguments (HEX): x = 0x

   y = 0x
7. What value is in the **LP** register?
   Contents of LP (HEX): 0x
8. What value was in **R1** at the time of the original call?
   Contents of R1 (HEX): 0x

9. What value will be returned in R0 as the value of the original call? [HINT: You can figure this out without getting the C code right!].
   Value returned to original caller (HEX): 0x
10. What is the hex address of the instruction tagged **yy**?
    Address of yy (HEX): 0x

Submit

Stacks and Procedures: 8

15 points possible (ungraded)

You are given the following listing of a C program and its translation to Beta assembly code:

```
int f(int x, int y) {
  int a = (x+y) >> 2;
  if (a == 0) return x;
  else return y + f(a, x+a);
}
```

```
f:      PUSH(LP)
        PUSH(BP)
        MOVE(SP, BP)
        PUSH(R1)
        LD(BP, -12, R0)
        LD(BP, -16, R1)
        ADD(R0, R1, R1)
        SRAC(R1, 2, R1)
laby:   BEQ(R1, labx)

        ADD(R0, R1, R0)
        PUSH(R0)
        PUSH(R1)
        BR(f, LP)
        DEALLOCATE(2)

labz:   LD(BP, -16, R1)
        ADD(R1, R0, R0)

labx:   POP(R1)
        MOVE(BP, SP)
        POP(BP)
        POP(LP)
        JMP(LP)
```

(Recall that a >> b means a shifted b bits to the right, propagating – ie, preserving -- sign)

1. Fill in the binary value of the BEQ instruction stored at the location tagged **laby** in the above program.
   opcode (6 bits): 0b

   Rc (5 bits): 0b

   Ra (5 bits): 0b

literal (16 bits): 0b

2. Is a location reserved for the argument **x** in **f**'s stack frame? Give its (signed) offset from **BP**, or **NONE** if there is no such location.
   Offset of x (in decimal), or “NONE”:
3. Is a location reserved for the variable **a** in **f**'s stack frame? Give its (signed) offset from **BP**, or **NONE** if there is no such location.
   Offset of variable a, or “NONE”:

   The procedure **f** is called from an external procedure and its execution is interrupted during a recursive call to **f**, just prior to the execution of the instruction tagged **labz**. The contents of a region of memory are shown below.

   NB: All addresses and data values are shown in hex. The **SP** contains **0×1C8**.

```
184:    4
188:    7
18C:    3
190:    5
194:    D0
198:    D4
19C:    D8
1A0:    7
1A4     2
1A8     4C
1AC     19C
1B0     2
1B4     4
1B8     2
1BC     4C
```

1C0	$1B0$
1C4	2
SP1C8 $\rightarrow$	3

4. What are the arguments to the *most recent* active call to **f**?
   Most recent arguments (HEX): x = 0x

   y = 0x

5. What are the arguments to the *original* call to **f**?
   Original arguments (HEX): x = 0x

   y = 0x

6. What value is in the **BP** register?
   Contents of BP (HEX): 0x

7. What value is in **R1** prior to the execution of the **LD** at **labz**?
   Contents of R1 (HEX): 0x

8. What value will be loaded into **R1** by the instruction at **labz** if program execution continues?
   Contents of R1 (HEX): 0x

9. What is the hex address of the instruction tagged **labz**?
   Address of labz (HEX): 0x

10. What is the hex address of the **BR** instruction that called **f** originally?
    Address of original call (HEX): 0x

Submit

Discussion

Hide Discussion

Topic: 12. Procedures and Stacks / Tutorial: Stacks and Procedures

Add a Post

Show all posts ⌄ by recent activity ⌄

- ☑ STACKS AND PROCEDURES 6: Why Mem[0×100] == 15? According to Question D in this problem, the value X that was passed to the original call to sqr(X... **3**
- ☑ STACKS AND PROCEDURES: 8.G. Q: "What value is in R1 prior to the execution of the LD at labz?" A: "The value in R1 prior to exec... **6**
- 💬 Stacks and Procedures: 1-A. Highest memory address possibly effected Risking to sound pedantic, shouldn't I say the highest memory address effected is 0×1403, inste... **4**
- 💬 Hmm This tutorial has many problems This tutorial has exceptionally high number of problems as compared to other topics. Any partic... **7**
- 💬 [STAFF] Cannot grade problem 7 On checking the problem 7, it gave me "We're sorry, there was an error with processing your req... **3**
- ☑ Problem with Stacks and Procedures: 6 In the stack trace, I expected to see a value of R0 right after the value of R1 since we have a "P... **2**
- ☑ Typo in Stacks and Procedures 4? Is it just me, or does the stack of question set 4 have a small error? There is a small section at 0... **3**
- ☑ Stacks and Procedures 4: result of recursive call Per the explanation, the result of a recursive call to f is returned in R0. How does it get there? **3**
- ☑ TP12.2: Question D (value of R1 at halt time) In the answer it's stated that the value of R1 at the time the program is halted is 0×03, given pre... **4**
- ☑ Further optimisation of program in Stack and procedure: 2 In the program given in stack and procedure:2 under rtn label, if we remove the instruction MOV... **2**
- ☑ [STAFF] Stack discipline in beta assembly language In the case of recursion, how will the program know that stack discipline has been achieved? Is ... **2**
- ☑ Tutorial problem: stacks and procedures 1 - typo? The explanation for part E says: "This means that the BP is pointing immediately following a full ... **2**
- 💬 [STAFF] Typo on "CAN'T TELL" When stating problems, e.g. Problem 2, question C > Please express the values in hex or write "... **3**