Stacks and procedures

Problem 1. ★ Consider the following implementation of an algorithm for finding the greatest common divisor of two integers:

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

The C compiler has compiled this procedure into the following code for an unpipelined Beta processor:

```
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)
        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)
        MOVE (BP, SP)
        POP (BP)
        POP (LP)
        JMP (LP)
```

A. The program above contains the instruction LD(BP,-16,R1). Explain what the compiler was trying to do when in generated this instruction.

The compiler is loading the value of the second argument ("b") so it could compute "a == b".

B. What are the contents of the memory location holding the instruction BR(L1)?

BR(L1) is encoded as 0x77FF0007. Remember that PUSH is actually a macro that expands into *two* Beta instructions, so there are 7 instructions between L1 and L2.

C. When the instruction labeled "L1" is executed, what is the best characterization of the contents of R0?

L1 is just at the beginning of the procedure return sequence, so at that point R0 should contain the value to be returned to the caller.

D. Looking at the code, a student suggests that both DEALLOCATE instructions could be eliminated since deallocation is performed implicitly by the MOVE(BP,SP) instruction in the exit sequence. After calling gcd, would it be possible to tell if the DEALLOCATE instructions had been removed?

Yes, it would be possible to tell: even though the stack is reset correctly by the MOVE(BP,SP) instruction, that happens after the POP(R1) and POP(R2) instructions, so they will not restore the values of R1 and R2 from the values pushed onto the stack during the entry sequence.

E. How many words of stack are needed to execute gcd(24,16)? Don't forget to include the stack space occupied by the arguments in the initial call.

gcd(24,16) => gcd(8,16) => gcd(8,8), at which point no more recursive calls are made. So three nested procedure calls are executed, each with a 6-word stack frame (2 args, LP, BP, saved R1, saved R2), for a total of 18 words of stack space.

F. During execution of gcd(28,70), the Beta processor is halted and the contents of portions of the stack are found to contain the following:

```
      ???
      0x00000594
      0x00001234
      0x00000046
      0x0000002A
      0x0000000E
      0x0000001C
      0x00000594
      0x0000124C
 BP-->0x0000002A
      0x0000000E
 SP-->0x00001254
      0x0000000E
```

What is the value of the second argument ("b") to the current call to gcd?

The second argument is located at Mem[Reg[BP] - 16] = 0xE.

G. What is the value in the BP register at the time the stack snapshot was taken?

0x1264 = (old BP stored in Mem[Reg[BP] - 4]) + (size of one stack frame)
= 0x124C + (6 words = 24 bytes = 0x18 bytes)
= 0x124C + 0x18

H. What is the correct value for "???" above?

gcd(28,70) => gcd(28,42) => gcd(28,14), at which point the snapshot was taken. So "???" is the first arguments to the preceding call, i.e., 28.

I. What is the address of the POP(R2) instruction?

All calls in the execution of gcd(28,70) have a < b, so all the recursive calls are from the second BR(gcd,LP). Using the value of LP stored at Mem[Reg[BP] - 8] which points at the DEALLOCATE instruction, the POP(R2) is stored at location 0x594 + 4 = 0x598.

J. At the time the stack snapshot was taken, what is the significance of the value 0x1254 in the location at <SP>?

Since SP points to the first *unused* location on the stack, the value in the location SP points to has no significance to the calculation, it's just ununsed stack space.

K. The stack snapshot was taken just after the execution of a particular instruction. Could the snapshot have been taken just after the execution of the PUSH(R1) instruction near the beginning of the gcd procedure?

No. Given the values of BP and SP, the PUSH(R2) instruction must have already been executed.

Problem 2. Consider the following function written in Beta assembly language:

```
foo:  PUSH (LP)
      PUSH (BP)
      MOVE (SP, BP)
      PUSH (R1)
      LD (BP, -12, R1)
B1:   BNE (R1, cmp, R31)
B2:   BR (rtn)
loop: SHLC (R1, 1, R1)
cmp:  CMPLT (R1, R31, R0)
      BF (R0, loop)
rtn:  MOVE (R1, R0)
      POP (R1)
      POP (BP)
      POP (LP)
      JMP (LP)
```

A. What are the contents of the memory location labeled B1?

0x7BE10002

B. If the branch instruction at the location labeled B2 is replaced with a NOP, characterize the input arguments that will cause foo to loop indefinitely.

If you enter the loop with R1 (initialized from the input argument) set to 0, it will loop forever.

C. How many times will the instruction labeled loop be executed when foo is called as follows:

```
CMOVE (1024, R0)
PUSH (R0)
CMOVE (-256, R0)
PUSH (R0)
BR (foo, LP)
DEALLOCATE (2)
```

Zero. The argument (-256) is less than zero, so the CMPLT instruction returns 1 (true) and the following BF does not branch, so the loop is never executed.

D. Which of the following instructions could replace the instruction at the memory location labeled cmp without changing the behavior of the function?

a) CMPLTC (R1, 0, R0)
b) CMPLEC (R1, -1, R0)
c) SRAC (R1, 31, R0)
d) SHRC (R1, 31, R0)
e) All of the above
f) None of the above

e. They all leave 1 in R0 if and only if R1 is negative.

E. Describe the behavior of the function foo if the instruction POP (LP) is replaced with DEALLOCATE (1).

foo still works correctly; actually it's a cycle faster while leaving the registers and memory contents the same as the unmodified program. LP isn't restored, but it wasn't changed by foo!

F. Suppose that function foo begins execution, and is subsequently interrupted, and the following indications of the processor's state are observed:

```
R0 = 0               Mem[0x1000] = 0x00001024
R1 = 0xbba00000      Mem[0x1004] = 0x000000e0
                     Mem[0x1008] = 0x00000001
                     Mem[0x100c] = 0x800000cc
      .              Mem[0x1010] = 0x00001004
      .              Mem[0x1014] = 0x00001000
      .              Mem[0x1018] = 0x00001774
                     Mem[0x101c] = 0x80000084
```

```
                    Mem[0x1020] = 0x00001014
BP = 0x00001024     Mem[0x1024] = 0x00001000
LP = 0x80000084     Mem[0x1028] = 0xafadcafe
SP = 0x00001028     Mem[0x102c] = 0xabadcafe
```

What is the value of the label foo?

You can't determine the value of the label foo since the procedure doesn't call any subroutines iteself and leave a PC somewhere we can examine.

G. What argument was the function foo called with in this instance?

First argument is stored at Mem[Reg[BP] - 12] = 0x00001774.

H. Assuming that foo was itself called from some other function, what was the first argument to that function?

The BP for the function that called foo is at Mem[Reg[BP] - 4] = 0x1014. The first argument for that function is at Mem[0x1014 - 12] = 0x00000001.

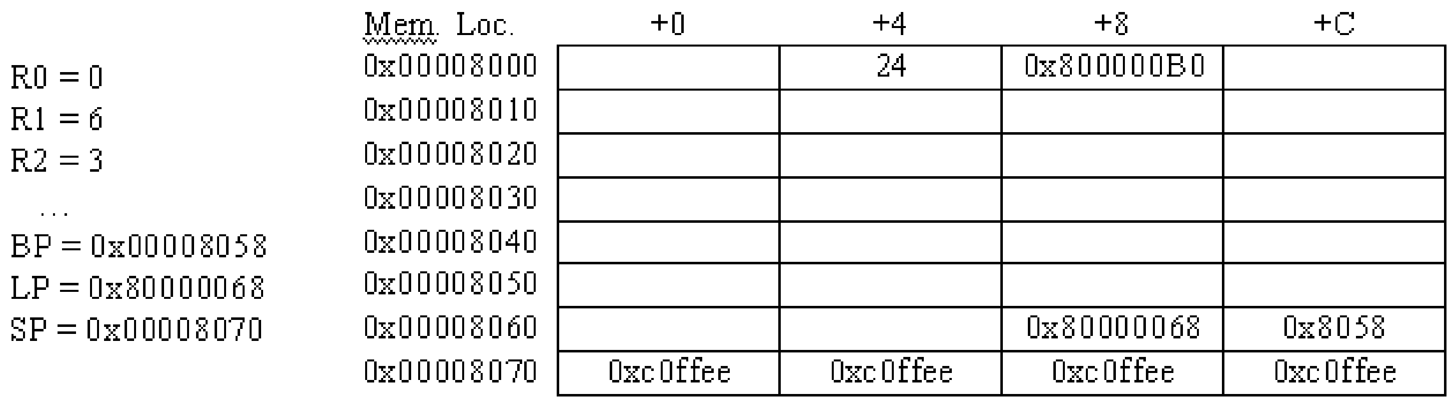
Problem 3. Consider yet another C function for computing the greatest common denominator between two numbers, x and y:

```
int gcd(int x, int y)
 {
   if (x == y) return x;
   if (y > x)
     y = y - x;
   else
     x = x - y;
   return gcd(x, y);
 }
```

It compiles into the following Beta code:

```
gcd:    PUSH (LP)
        PUSH (BP)
        MOVE (SP, BP)
        PUSH (R1)
        PUSH (R2)
        LD (BP, -12, R1)
        LD (BP, -16, R2)
        CMPEQ (R2, R1, R0)
        BF (R0, ifxgty)
        MOVE (R1, R0)
        BR (done)
ifxgty: CMPLE (R2, R1, R0)
        BT (R0, else)
        SUB (R2, R1, R2)
        BR (call)
else:   SUB (R1, R2, R1)
call:   PUSH (R2)
        PUSH (R1)
        BR (gcd, LP)
        DEALLOCATE (2)
done:   POP (R2)
        POP (R1)
        POP (BP)
        POP (LP)
        JMP (LP)
```

Assume the function gcd is called with two positive non-zero arguments while the stack pointer set to 0x8000 (i.e. it's two arguments are placed in 0x8000, and 0x8004). Sometime thereafter, the Beta is stopped and the contents of various memory locations and registers are examined. Here is a summary of what is seen:



R0 = 0
R1 = 6
R2 = 3
...
BP = 0x00008058
LP = 0x80000068
SP = 0x00008070

Mem. Loc.	+0	+4	+8	+C
0x00008000		24	0x800000B0	
0x00008010				
0x00008020				
0x00008030				
0x00008040				
0x00008050				
0x00008060			0x80000068	0x8058
0x00008070	0xc0ffee	0xc0ffee	0xc0ffee	0xc0ffee

The contents of the blank table entries are unknown.

A. Fill in as many blank spaces as possible in the table above and indicate those locations whose contents cannot be determined from the supplied information.

The table below shows what we can deduce about the contents of the stack from what we have been given. Here are some notes on how we did this:

- By looking at the code, we can tell that each stack frame would be 6 words long. Tracing the frames from the beginning, we can mark out the stack frames and thus know what purpose each memory location serves (e.g., holding LP, etc.)
- Since all but the first stack frame belongs to a recursive call from gcd to itself, and since there is only one place in gcd where it calls itself, we know that, except in the first frame, the saved LP should have the same value in all the frames: 0x80000068. Similarly, knowing the stack frame structure allows us to know where BP will be pointing in each stack frame. This lets us know the value of BP that will be saved in the next stack frame. This way we can determine the BP values saved in all the frames except the first.
- To determine the arguments, we need to work backwards. First, we note that the values of R1 and R2 when the program was stopped tell us the arguments x and y, respectively, of the current invocation of gcd. This

is because the previous invocation used R1 and R2 for pushing the arguments to the recursive call. Thus, we know 0x8060 contains y=3, and 0x8064 contains x=6. We further know that the arguments to the previous invocation were y=9 and x=6 because the fact that R0 was 0 when we stopped the program tells us that in the previous invocation, y was greater than x and that the new y (i.e., 3) was the difference between the old y (i.e., 9) and the old x (i.e., 6).

- Unfortunately, knowing that the third invocation to gcd was gcd(x=6,y=9) does not tell us for sure what the arguments to the invocation before that one were. They could have been (6,15) or (15,9). Fortunately, it is not too hard to draw a binary tree of the possible arguments for each call to gcd. If we do this, we will find that of the 8 possible argument pairs for the top-level call to gcd, only (x=24,y=33) has 24 as its first argument.

- Finally, after filling-in the arguments, we can trace through the code and determine the values in the R1 and R2 slots. We can do this more quickly by noticing the relation between the pushed R1 and R2 values and the arguments before them. Note however, that this pattern is not a general rule, but just occurs because gcd uses R1 and R2 when PUSHing arguments.

- We cannot know the values in memory locations 0x800C to 0x8014 because we don't know exactly what main (gcd's original caller) was doing with BP, R1 and R2 before it called gcd. For example, main does not have to use R1 and R2 to PUSH the arguments, so we cannot determine R1 and R2 as we did for the recursive calls.

```
Mem. Loc.    Contents    Comments

0x8000       33          initial y
0x8004       24          initial x
0x8008       0x800000B0  main's return addr
0x800C       unknown     main's BP
0x8010       unknown     main's R1
0x8014       unknown     main's R2
------------------------------------------
0x8018       9           y
0x801C       24          x
0x8020       0x80000068  gcd's return addr
0x8024       0x8010      previous gcd's BP
0x8028       24          previous gcd's R1
0x802C       9           previous gcd's R2
------------------------------------------
0x8030       9           y
0x8034       15          x
0x8038       0x80000068  gcd's return addr
0x803C       0x8028      previous gcd's BP
0x8040       15          previous gcd's R1
0x8044       9           previous gcd's R2
------------------------------------------
0x8048       9           y
0x804C       6           x
0x8050       0x80000068  gcd's return addr
0x8054       0x8040      previous gcd's BP
0x8058       6           previous gcd's R1
```

```
0x805C       9           previous gcd's R2
------------------------------------------
0x8060       3           y
0x8064       6           x
0x8068       0x80000068  gcd's return addr
0x806C       0x8058      previous gcd's BP
0x8070       0xC0FFEE    free
0x8074       0xC0FFEE    free
0x8078       0xC0FFEE    free
0x807C       0xC0FFEE    free
```

B. Using the information supplied, determine what is the address of the entry point of gcd? What initial arguments was it called with? What value would this particular call have returned if it had been allowed to complete?

We know from the state of the machine when it was stopped that in a recursive call, LP is set to 0x80000068 by the BR(gcd,LP) instruction inside gcd. This gives us the address of the DEALLOCATE(2) instruction following the BR. From here, we count 25 instructions backwards to gcd, keeping in mind that PUSH and POP are macros composed of 2 instructions each. Thus, we know that the address of gcd must be 25*4 = 100 = 0x64 bytes before 0x80000068, that is, 0x80000004. (Actually, the real address of gcd is 0x04. The 8 in the most significant digit means that the code was running in supervisor mode. More on this in later lectures.)

If the function was allowed to complete, it would have returned 3 as the value of gcd(24,33).

C. Which instruction was about to be executed when the machine was stopped?

The instruction about to be executed when the machine was stopped was the MOVE(SP,BP) instruction. Since the last item on the stack is the value of BP, we know that the machine was stopped after PUSH(BP), but before PUSH(R1). We know that it was stopped before MOVE(SP,BP) because BP has not been set to SP yet. We also know that we could not have stopped the machine during the POP sequence because otherwise, R0 would not be 0.

D. If the machine was restarted in the same state as when it was stopped, how many more times would gcd have been called? Given the particular call to gcd whose state is summarized above, what is the greatest value that the stack pointer register (SP) will take on during the execution of this call.

If we let the program continue to run, it would make one more recursive call to gcd(3,3), which will then return a 3. This means we would add 2 more words to the stack to complete the stack frame of gcd(6,3) and add another 6 words for the frame of gcd(3,3). Thus, the greatest value that the stack pointer register (SP) will take on would be 0x8090.

E. Suppose gcd is called with two non-zero positive arguments, x and y. What is the maximum depth that the stack can grow? Give your answer in terms of x and y.

As a worst-case upper bound, note that the worst case happens when either x or y is 1. In this case, we recurse as many times as the non-1 number, or in other words, the number of levels is max(x,y). This is not a tight bound, however. A tighter upper bound would be max(x,y)/gcd(x,y). This is because the number of calls is equal to the number of subtractions it takes to get from the larger number to gcd(x,y), where each subtraction is at least as large as gcd(x,y). This is not the tightest upper bound possible yet, however. For example, in this specific case, it gives us 33/3 = 11 levels instead of the actual 6. In any case, the maximum stack depth would be (# of levels)*24 bytes, since each stack frame has 6 words * 4 bytes/word = 24 bytes.

Problem 4. Consider the following C function, which compiles to the Beta assembly code shown to the right:

```
int fib(int n)
{
    if (n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

```
fib:    PUSH (LP)           | Entry.1
        PUSH (BP)           | Entry.2
        MOVE (SP, BP)       | Entry.3
        PUSH (R1)           | Entry.4
        PUSH (R2)           | Entry.5
        LD (BP, -12, R2)
        CMPLEC (R2, 1, R0)
        BT (R0, L2)
        SUBC (R2, 1, R0)
        PUSH (R0)           | Code.1
        BR (fib, LP)        | Code.2
        DEALLOCATE (1)      | Code.3
        MOVE (R0, R1)       | Code.4
        SUBC (R2, 2, R0)    | Code.5
        PUSH (R0)           | Code.6
        BR (fib, LP)        | Code.7
        DEALLOCATE (1)      | Code.8
        ADD (R1, R0, R0)
        BR (L3)
L2:     MOVE (R2, R0)
L3:     POP (R2)            | Return.1
        POP (R1)            | Return.2
        POP (BP)            | Return.3
        POP (LP)            | Return.4
        JMP (LP)            | Return.5
```

A. How many memory locations are allocated on the stack for each call to fib?

There are 5 memory locations allocated for each call. Fib has one argument and 4 additional locations are allocated by the entry sequence (locations for saving LP, BP, R1, R2).

B. Consider the following code substitution for the series of instructions labeled Entry.[1-5] in fib:

```
fib:    ALLOCATE (4)
        ST (LP, -16, SP)
        ST (BP, -12, SP)
        ST (R1, -8, SP)
        ST (R2, -4, SP)
        ADDC (SP, -8, BP)
```

Does the new code fragment perform the same function as the block that it replaces? Does it require more or fewer Beta instructions? Is the given sequence safe (could it be interrupted at any point and still operate correctly)? What are the advantages and disadvantages of this substitution?

The code sequence is equivalent. To see why imagine all the PUSHes expanded into their component instructions (ADDC followed by ST). The ALLOCATE(4) above is equivalent to the 4 separate ADDCs embedded in the PUSHes; the four ST instructions writes the same information onto the stack in the same order as the original sequence. The new sequence is shorter by 3 instructions and is "safe" since the SP is incremented before the STs happen.

New sequence advantages: fewer instructions, faster execution. Disadvantages: harder to understand (?).

C. Which of the following three Beta assembly code fragments could transparently replace the series of instructions labeled Return.[1-5]?

```
L3: LD (SP, -16, LP) L3: DEALLOCATE (4)  L3: LD (SP, -4, R2)
    LD (SP, -12, BP)     LD (SP, 12, R2)     LD (SP, -8, R1)
    LD (SP, -8, R1)      LD (SP, 8, R1)      LD (SP, -12, BP)
    LD (SP, -4, R2)      LD (SP, 4, BP)      LD (SP, -16, LP)
    DEALLOCATE (4)       LD (SP, 0, LP)      DEALLOCATE (4)
    JMP (LP)             JMP (LP)            JMP (LP)
```

The middle sequence is not safe since the space on the stack is deallocated before we've finished using the contents of those locations. Either the left or right sequences could transparently replace the original return sequence.

D. How do these versions compare to the original in terms of the number of instructions used? Are there any other advantages or disadvantages of these substitutions?

Either the left or right sequence is shorter (and hence faster) than the original sequence.

E. Suppose the following code was substituted for the assembly language block labeled Code.[1-8]:

```
PUSH (R0)
BR (fib, LP)
MOVE (R0, R1)
SUBC (R2, 2, R0)
ST (R0, -4, SP)
BR (fib, LP)
DEALLOCATE (1)
```

Is this resulting code faster? Safe?

The new code sequence is safe and 2 instructions faster since both the SUBC inside of DEALLOCATE and the subsequent ADDC inside of PUSH have been eliminated.

Problem 5.

A. Using our procedure linkage convention, how far (in terms of addresses) can a callee be from its caller? What parts of our convention are impacted by this limitation (the calling sequence, the entry sequence, or the return sequence)? How might our convention be extended to allow for calls to more distant procedures?

   The entry point of the callee must be reachable by a branch, i.e., it must be within approximately 2^15 words of the call instruction. Only the calling sequence has this address built-in, all the other parts of our calling convention use values taken from 32-bit registers or memory locations. We could overcome this limitation by using LDR to load the target address into a register and then JMP using the register.

B. Our current calling convention requires all addresses to be resolved at compile time. Suppose that we would like to be able to call special libraries of functions whose addresses are unknown until the program is loaded into memory. How could we modify our procedure linkage convention to accommodate this capability?

   Object-oriented languages often require this sort of mechanism since the actual target of the call can't be determined at compile time. Now the target address needs to be computed (say by looking it up in a method table associated with the object's type) and then using JMP to transfer control to the method. All the other elements of the convention can stay as they are.

Problem 6.

A. Which of the following Beta assembly language macros can always be used for loading the Nth argument passed to a function using our standard procedure linkage convention. Assume argument 1 is the first argument and no more than 2048 arguments are passed.
   1. .macro ARG(N, RC) LD(SP, -8 - 4*N, RC)
   2. .macro ARG(N, RC) LD(BP, -12 + 4*N, RC)
   3. .macro ARG(N, RC) LD(SP, (N-1)*4-12, RC)
   4. .macro ARG(N, RC) LD(BP, -4*(N+2), RC)
   5. .macro ARG(N, RC) LD(RC, -4*N+8, BP)

   Macro 1. Won't work since it uses a fixed offset from SP to access a particular argument. PUSHes and POPs within a procedure (eg, from pushing args for a nested procedure call) will change the position of SP relative to the base of the stack frame, so a fixed offset won't work everywhere in the procedure.

   Macro 2. The offset calculation is wrong, e.g., for N = 1 the computed offset is -8 instead of the required -12.

   Macro 3. The offset calculation is wrong again, e.g., for N = 2 the computed offset is -8 instead of the required -16.

   Macro 4. This is correct.

   Macro 5. The offset calculation is incorrect and RC/BP are reversed in the call to LD.

Problem 7. In this problem, we take a quick look at one of several alternatives to the instruction set architecture family represented by the Beta. We consider here the design of a pure stack machine, an architectural approach which has been used frequently in virtual machines (e.g., abstract machine architectures used to communicate instruction sequences to compilers) and occasionally in real machines such as the Burroughs B6700. The idea of a pure stack machine is to use the processor's stack as the only programmer-visible processor state. Unlike the Beta, there are no registers for explicit use by instructions: nearly all instructions pop their source operands from the stack, and push their result onto the stack. Although the implementation of a stack machine may use dedicated registers (such as an SP and PC), such registers are never explicitly addressed by code. As a result, most instructions are coded simply as an opcode - they need no fields to specify source and destination operands. In our hypothetical stack machine, such instructions consist of a single byte. LOAD and STORE instructions are exceptions, and provide a way to access data from memory. Each is represented as an opcode followed by a 4-byte memory address; LOAD pushes the contents of the indicated address onto the stack, while STORE pops the top word from the stack and stores it into the supplied address. On such a machine, the program

```
        LOAD(A)
        NEG
        LOAD(B)
        MUL
        LOAD(C)
        DIV
        STORE(D)



A:      LONG(123456)
B:      LONG(456789)
C:      LONG(234567)
D:      LONG(0)
```

would compute -A*B/C and store the result in D. Note that the single-byte instruction NEG pops and negates its argument, pushing the result back onto the stack; DIV and MUL each pop two operands and push the result back onto the stack. Note that variables and constants are stored at arbitrary locations in main memory.

A. How many bytes are taken by the above code (exclusive of the four data locations)?

Excluding the four data locations, we are interested in the amount of bytes taken from the LOAD(A) to the STORE (D). There are 7 instructions, and each instruction takes 1 byte, except for LD's and ST's which take an additional 4 bytes to index the address. Therefore, this takes 23 bytes.

B. Rewrite the above code sequence for the Beta, again assuming that A, B, C, and D can be arbitrary 32-bit memory locations. How many bytes will the Beta instruction sequence occupy?

A,B,C,D are the addresses of the memory locations for variables a,b,c, and d.

```
        BR(skip)
aload:  LONG(A)
bload:  LONG(B)
cload:  LONG(C)
dload:  LONG(D)
skip:   LDR(aload, R0)     | Load address of a into R0
```

```
        LD(R0,R0)           | Load value of variable a into R0
        LDR(bload, R1)      | Load address of b into R1
        LD(R1,R1)           | Load value of variable b into R1
        LDR(cload, R2)      | Load address of c into R2
        LD(R2,R2)           | Load value of variable c into R2
        LDR(dload, R3)      | Load address of d into R3

        SUB(R31,R0,R4)      | calculate -A
        MUL(R4,R1,R5)       | calculate -A*B
        DIV(R5,R2,R6)       | calculate -A*B/C
        ST(R6,R3)           | write D back.
```

Since each instruction is 4 bytes long, and we have 16 instructions (and embedded data), this takes up 64 bytes. This is over twice as many bytes as the stack machine code.

C. Now rewrite your Beta code assuming that A, B, C, and D are stored in the local stack frame as local variables, i.e., they are addressed using BP with the appropriate offset. How many bytes does your new code require?

If A, B, C, and D are in the stack, we can use R27, the BP register, to retrieve A, B, and C:

```
    LD(BP, 0, R0)     | Load memory location value "A" in R0
    LD(BP, 4, R1)     | Load memory location value "B" in R1
    LD(BP, 8, R2)     | Load memory location value "C" in R2
    MULC(R0, -1, R0) | Perform the -A operation
    MUL(R0, R1, R1)   | Perform the -A*B operation.
    DIV(R1, R2, R2)   | Perform the -A*B/C operation
    ST(R2, 12, BP)    | Store computation in memory location D
```

Now, the code is only 7 lines long, of 4 bytes/instruction => 28 bytes. This is still more than the stack implementation, but less than part B.

D. Stack machines commonly provide special load and store instructions optimized for access to data (such as local variables) stored near the top of the stack. Typically such instructions address memory locations computed using an instruction-stream constant and the current value of the SP.

Propose specific SP-relative instructions, LOAD-SR and STORE-SR, that allow access to locations stored near the top of the stack. Specify how each instruction is coded, including an opcode and any instruction-stream constant necessary. Describe the details of each instruction's operation in the same way instructions are documented in the Beta Documentation

```
LOAD-SR
Usage: LOAD-SR(literal), where literal is a one byte unsigned offset.
PC = PC + 4
EA = Reg[SP] - 4*literal
MEM[SP] = Mem[EA]
Reg[SP] = Reg[SP] + 4
```

The effective address EA is computed by subtracting from the stack pointer four times the 8-bit displacement literal. The location in memory specified by EA is read and then pushed onto the stack. We use an unsigned format because we don't need to represent both positive and negative offsets.

```
STORE-SR
Usage: STORE-SR(literal), where literal is a one byte unsigned offset.
PC = PC + 4
EA = Reg[SP] - 4*literal
MEM[EA] = MEM[SP-4]
Reg[SP] = Reg[SP] - 4
```

The effective address EA is computed by subtracting from the stack pointer four times the 8-bit displacement literal. The top element of the stack is popped off and written into the effective address.

E. Recode the above sequence for a stack machine, assuming that A, B, C, and D are located near the top of the stack. How many bytes are taken?

New code assuming that a, b, c and d have been pushed onto the stack, in that order.

```
    LOAD-SR(4)  | push a onto the top of the stack. (note, this changes how
                 | far away b,c, and d are!)
    NEG
    LOAD-SR(4)   | bring b to the top of the stack
    MUL
    LOAD-SR(3)   | bring c to the top of the stack
    DIV
    STORE-SR(2)  | write d back to its original place on the stack.
```

This code now takes 11 bytes. Notice how it is somewhat troublesome to keep track of where variables are at every point in time. This is why we typically use a BP register; it simplifies coding sequences like this tremendously by providing a fixed reference position.

F. How would you go about choosing the size of the constant for a serious design?

To choose the size of the constant for a serious design, we would have to analyze our system by running some benchmarks and see how far most away from memory most loads and stores are at. For example, if we could reach 40% of our source and destination addresses with 1 byte, but 90% with two bytes, then we would probably opt to have the constant be two bytes. However, if in 1 byte we could reach 40% of our source and destination addresses, but with even two, three and even four bytes, reach 50%, then it would not be worth it to increase the number of bytes.

G. Propose simple instructions that allow space for variables (like A, B, C, and D) to be allocated on the stack and subsequently deallocated when they are no longer needed.

We want to make a stack version of our beta allocate and deallocate. Therefore:

```
ALLOCATE:
Usage: ALLOCATE <1 byte unsigned literal, the # of words to allocate>
SP = SP + 4*literal
PC = PC+4
```

```
DEALLOCATE:
Usage: DEALLOCATE <1 byte unsigned literal, the # of words to deallocate>
SP = SP - 4*literal
PC = PC +4
```

Note: we used unsigned literals since we don't need to represent both positive #s AND negative #'s, and we can get twice the range by using unsigned numbers.

It would be acceptable if the argument to allocate/deallocate was taken from the stack, rather than an argument.

H. Typical stack machines also hide the PC from explicit program access, providing CALL and RETURN instructions to be used for procedure calls. At minimum, such instructions copy values between the PC and the stack.

Devise simple, single-byte CALL and RETURN instructions that allow a procedure having no arguments to be called and to return to its caller. Describe the operation of each instruction.

If we do not need to pass any arguments, then we do not need to push arguments onto the stack, and in fact, all we care about is that we can go to another procedure, and that it returns to us. Therefore:

```
CALL:
Usage: CALL <4 byte absolute (unsigned) address>
Mem[SP] = PC + 4
SP = SP + 4
PC = absolute address
```

It would also be acceptable if the call had no argument and took the address from the stack.

```
RETURN:
Usage: RETURN
SP = SP - 4
PC = Mem[SP]
```

I. Finally, consider using your SP-relative load and store instructions to implement stack frames on our little stack machine. Your job is to devise and describe conventions to be followed by caller and callee allowing (i) arguments to be passed from the caller to the callee, (ii) local variables to be allocated and used by the callee, and (iii) a single value to be returned from the callee to the caller.

To be concrete, consider the following procedure and its call:

```
int F(int a, int b)
 { int x;
   x = a*b;
   return (x + x*b)
 }
...
```

```
z = F(23, 41);
...
```

The natural place to return the value of the procedure is on the stack, but it is awkward to allocate this location in the caller because it blocks access to the return value. How does your proposed convention deal with this problem?

We have to make sure the return value on the stack is accessible to the callee as well as placed before the return value on the stack. If the callee always knows how many values it has pushed onto the stack, and it knows the last two things on the stack before it was called is the return value followed by the return address, then it can use STORE-SP to store the value onto the return value.

J. Diagram the layout of the stack using your conventions. Show the location of stacked arguments, local variables, the return value, and any temporaries used by the caller.

STACK layout:

```
argn
argn-1

arg1
return location
return value
local argn
local argn-1

local arg1
temporaries
```

K. Give a general template for the code sequence necessary to call an arbitrary procedure. Illustrate using the specific call in the above code, but indicate how it generalizes to arbitrary calls.

Template:

```
LOAD(ARGn)
LOAD(ARGn-1)
...
LOAD(ARG1)
ALLOCATE(1)  | Allocate space for return value
CALL(callee) | using our instruction above
DEALLOCATE(1) | Get rid of the return value
DEALLOCATE(n) | Get rid of arguments
```

In our case:

```
LOADC(41) | arguments in reverse order
LOADC(23)
ALLOCATE(1)
```

```
CALL(F)
DEALLOCATE(1)
DEALLOCATE(2) | these could be merged into one but are separated for clarity.
```

L. Give a template for the code of the called routine. Illustrate by translating the procedure F, above, to stack machine code using your conventions.

Template:

```
ALLOCATE(m) | Allocate space for local variables

<insert useful code here, including a STORE-SR to
put the correct value into the space on the stack reserved
for the return value.>

DEALLOCATE(m) | Deallocate local variables
RETURN
```

This is somewhat unpleasant, due to the fact that every time we do almost anything, we change the value of the stack pointer (since every operation modifies the stack in some way), which changes the offsets for the variables.

Drawing a careful diagram of the stack while working this problem is essential.

```
F:
    ALLOCATE(1)   | reserve room for x
    LOAD-SR(4)    | fetch a
    LOAD-SR(6)    | fetch b
    MUL
    STORE-SR(2)   | write a*b into x
    LOAD-SR(1)    | load X again
    LOAD-SR(6)    | load b
    MUL
    LOAD-SR(2)    | load X again
    ADD
    STORE-SR(4)   | write the return value
    DEALLOCATE(1) | get rid of our local variable X
    RETURN
```

We see from this example the value of a BP register.