CI Input and Output (ecall, DC)

The values that we discussed so far were all constants, known in advance (at compile time). Generic computations, however, usually employ values that are not known at compile time but will be available later, at runtime. Such values are often provided to the system through some input facilities such as system calls or memory mapped channels.

ecall

In RVS an `ecall` (environment call) can be used to input an integer value (see the RVS- IOsyscalls manual for more details) as follows:

```
ecall x5, x0, 5 ;integer
```

The generic source code that sums 2 values input by the user at runtime could, therefore, be as follows:

```
ecall x6, x0, 5 ;integer
ecall x7, x0, 5 ;integer
add   x5, x6, x7
```

Save the above example as a file named **c3a.asm** for possible future use.Compile and run the above example. The following data as shown in the OUT window was used for testing:

```
1
2
```

And here are the resulting values in the `Regs` window:

```
x5 t0   0x0000000000000003 3
x6 t1   0x0000000000000001 1
x7 t2   0x0000000000000002 2
```

The above source code placed the resulting sum in the x5 register, so we could check the result in the `Regs` window as shown above. Another way would be to output the x5 value so that the user can see it directly in the OUT window as follows:

```
ecall x6, x0, 5 ;integer
ecall x7, x0, 5 ;integer
add   x5, x6, x7
ecall x0, x5, 0
```

Save the above example as a file named **c3b.asm** for possible future use. Compile and run the above example. The following data as shown in the OUT window was used for testing:

```
1
2
3
```

Note that the result is now shown directly on the last line in the OUT window as shown above, so we don't need to look for it in the Regs window. The above code inputs 2 values, calculates their sum, outputs it, and finishes. To run it again, just press the START button in the Listing window of the RVS. Repeated running of the code in a loop can be organized using the `beq` instruction in the following way:

```
loop:   ecall   x6, x0, 5 ;integer
        ecall   x7, x0, 5 ;integer
        add     x5, x6, x7
        ecall   x0, x5, 0
        beq     x0, x0, loop
```

Save the above example as a file named **c3c.asm** for possible future use. Compile and run the above example.

DC

The simple summation code demonstrated so far provides no information to the user about its purpose and use. Such information could be provided through messages (sequences of characters or strings) shown in the OUT window. A sequence of characters can be defined using the DC (Define Characters) assembler commands as follows:

```
c1: DC  "integer:"
c2: DC  "sum:"
s1: DC  "Inputs two integers\nand prints the sum.\0"
```

The first 2 sequences of characters above are short enough to fit in a 64 bit register (up to 8 bytes) and can thus be used as prompts in the Read Services (e.g. the read_integer ecall) and printed by the print_characters ecall. The third sequence of characters which is longer than 8 bytes and can be printed only by the print_string ecall.The source code below extends the previous example with information messages and prompts:

```
c1:     DC  "integer:"
c2:     DC  "sum:"
s1:     DC  "Inputs two integers\nand prints the sum.\0"
        ld  x28, c1(x0)
        ld  x29, c2(x0)
        addi    x30, x0, s1
        ecall   x0, x30, 4 ;info string
loop:   ecall   x6, x28, 5 ;integer
        ecall   x7, x28, 5 ;integer
        add     x5, x6, x7
        ecall   x1, x29, 3 ;"sum:"
        ecall   x0, x5, 0 ;the result
        beq x0, x0, loop
```

Save the above example as a file named c3d.asm for possible future use. Compile and run the above example. Here is a sample dialog: Inputs two integers and prints the sum.

```
c1:     DC  "integer:"
c2:     DC  "sum:"
s1:     DC  "Inputs two integers\nand prints the sum.\0"
        ld  x28, c1(x0)
        ld  x29, c2(x0)
        addi    x30, x0, s1
        ecall   x0, x30, 4 ;info string
loop:   ecall   x6, x28, 5 ;integer
        ecall   x7, x28, 5 ;integer
        add     x5, x6, x7
        ecall   x1, x29, 3 ;"sum:"
        ecall   x0, x5, 0 ;the result
        beq x0, x0, loop
```

Save your solution as a file named **cex3a.asm** for possible future use.

C2 The stack

The stack is a fundamental data structure with LIFO (Last-In-First-Out) data policy supporting **push** and **pop** operations. The push operation adds an element to the stack and the pop operation removes the most recently added element. Traditionally stacks are organized to grow downward i.e. newly added elements are placed at lower memory addresses with respect to the earlier added elements. The `x2` register is often used as a stack pointer, so it is aliased to `sp`. When in use, this register must always point to the top of the stack.

Now let's define a stack starting at the memory address of 0x10000 and initialize the sp accordingly.

```
STACK: EQU  0x100000
       lui  sp, STACK>>12
```

Compile and run the above sequence. Check the value of the label STACK in the Listing:

```
0x0000000000100000  STACK
```

Compare it with the value stored in the `sp` register:

```
x2  sp   0x0000000000100000 1048576
```

Pushing the value of 1 in the stack can be done as follows.

```
addi    x5, x0, 1
sd      x5, 0(sp)
addi    sp, sp, -8
```

Compile and run the above sequence. Check the value in the `sp` register:

```
x2  sp   0x00000000000FFFF8 1048568
```

Check the values stored in the stack area of the memory:

```
0x0000000000100000 0x0000000000000001 1
```

Now let's push 2 in the stack:

```
addi    x5, x0, 2
sd      x5, 0(sp)
addi    sp, sp, -8
```

Compile and run the above sequence. Check the value in the `sp` register:

```
x2  sp   0x00000000000FFFF0 1048560
```

Check the values stored in the stack area of the memory:

```
0x00000000000ffff8 0x0000000000000002 2
0x0000000000100000 0x0000000000000001 1
```

Now let's pop the top element of the stack:

```
addi    sp, sp, 8
ld      x5, 0(sp)
```

Compile and run the above sequence. Check the values in the `sp` and the `x5` registers:

```
x2  sp  0x00000000000FFFF8 1048568
x3  gp  0x0000000000000000 0
x4  tp  0x0000000000000000 0
x5  t0  0x0000000000000002 2
```

To illustrate the LIFO data policy of the stack we will create a program that first inputs a sequence of integer numbers pushing them into the stack in the order of input and then outputs the numbers in a reverse order by popping them from the stack one by one.

```
s1:     DC "No of ints:\0"
s2:     DC "Int"
s3:     DC ":"
s4:     DC " "
STACK:  EQU 0x100000
        lui     sp, STACK>>12   ;
        add     x5, x0, s1      ;
        ecall   x1, x5, 4       ;out question
        ecall   x5, x0, 5       ;inp No of ints
        addi    x6, x0, 1       ;counter
loop1:  ld      x7, s2(x0)
        ecall   x1, x7, 3       ;out Int
        ecall   x1, x6, 0       ;out index
        ld      x7, s3(x0)      ;
        ecall   x8, x7, 5       ;out :, in #
        sd      x8, 0(sp)       ;push
        addi    sp, sp, -8      ;push
        addi    x6, x6, 1       ;counter
        bge     x5, x6, loop1   ;
        addi  x6, x0, 1         ;counter
loop2: ld       x7, s2(x0)
        ecall   x1, x7, 3       ;out Int
        ecall   x1, x6, 0       ;out index
        ld      x7, s3(x0)      ;
        ecall   x1, x7, 3       ;out :, in #
        addi    sp, sp, 8       ;pop
        ld      x8, 0(sp)       ;pop
        ecall   x0, x8, 0       ;out :, in #
        addi    x6, x6, 1       ;counter
        bge     x5, x6, loop2
```

Save the above example as a file named **c2a05.asm** for possible future use and submit it. Compile and run the above example. Check the result in the OUT window:

```
No of ints:3
Int1:1
Int2:4
Int3:5
Int1:5
Int2:4
Int3:1
```

C3 Leaf and non-leaf procedures (jalr, jal)

Another very common use of the stack is in function and procedure calls. We will begin with writing some procedures for redacting texts. The following template program defines a sample text as a string (a NULL-terminated sequence of characters) using the DC assembler command, outputs it, and finishes the execution by ebreak:

```
str1: DC    "sampled text\0"    ;
      addi  x6, x0, str1        ;output
      ecall x0, x6, 4
      ebreak x0, x0, 0          ;finish
```

Save the above example as a file named **c3a05.asm** for possible future use and submit it. Compile and run the above example. Check the result in the OUT window:

```
sampled text
```

We will use the above program as a placeholder when developing and testing the character manipulation procedures that follow. Now let us consider removing the final "d" in the first word of our sample string. To do this we can write a procedure that deletes a single character in a string.

jalr

When the procedure accomplishes its task it can use the jalr instruction to jump back to the return address stored in register x1 at call time. Here is the implementation of a simple delch1 procedure that deletes a character at a specified position in a string:

```
delch1: lb      x5, 0(a2)
loop1:  beq     x5, x0, end1    ;
        lb      x5, 1(a2)       ;
        sb      x5, 0(a2)       ;
        addi    a2, a2, 1       ;
        jal     x0, loop1       ;
end1:   jalr    x0, 0(x1)       ;return
```

The above procedure takes a single argument from the register `a2` which holds the address of the character to be removed. Note that we have to loop through all the characters after the one designated for deletion and move them forward by one position to fill the gap.

jal

We have prepared a placeholder program that defines the test string `str1`, then loads the address of the character to delete in register `a2`, and finally employs the `jal` (jump and link) instruction to store the return address in register `x1` and jump to the first instruction of the `delch1` procedure. You can insert the following placeholder program in front of the procedure for testing:

```
str1: DC    "sampled text\0"          ;
      addi  a2, x0, str1+6      ;ch addr
      jal   x1, delch1
      addi  x6, x0, str1        ;output
      ecall x0, x6, 4
      ebreak x0, x0, 0          ;finish
```

Save the above example as a file named **c3b05.asm** for possible future use and submit it. Compile and run the above example. Check the result in the OUT window:

```
sampled text
```

We can write a more generic *delch* procedure that is not limited to deleting a single character. In this case `a2` register can again hold the address of the character to be deleted and the register a3 can hold the number of characters to be deleted. As *delch1* fills the gap after every deletion of a character, the address of

the next-to-be-deleted character will remain the same. Therefore, in the *delch* procedure we just call *delch1* the number of times indicated in a3:

```
delch:  jal x1, delch1          ;
        addi    a3, a3, -1          ;
        bne     a3, x0, delch   ;
        jalr    x0, 0(x1)       ;return
```

The above procedure, however, is not going to work properly. The reason is that it is a non-leaf procedure (a procedure that calls one or more other procedures) and thus needs to save and restore certain values. One obvious problem is that in the last line we attempt to use the return address provided in `x1` at call time, but it has been overwritten in the first line of the code. Another problem is that *delch1* changes the value in a3 so this value must be saved before each call and restored afterwards. The first thought that comes to mind is to try to use some of the other registers for saving and restoring the values. While this might be fairly easy to do in our simple case (try it as an exercise), the available registers will soon be exhausted when nested procedure calls with more parameters are employed. The fundamental solution to this problem is to save and restore to the RAM where we have much more space. Using a stack (LIFO memory access policy) is particularly suitable for this purpose since each procedure needs the temporary storage only while active (since invocation until return.) To streamline the use of the registers and their store and restore a calling convention has been established (see the table **REGISTER NAME, USE, CALLING CONVENTION** in the Green Card.) Here is the stack version of the *delch* procedure:

```
delch: sd   x1, 0(sp)           ;push
       sd   s0, -8(sp)          ;push
       sd   s1, -16(sp)         ;push
       addi sp, sp, -24         ;push
       addi s0, a2, 0           ;
       addi s1, a3, 0           ;
       bge  x0, s1, end2
loop2: jal  x1, delch1          ; j delch1
       addi a2, s0, 0
       addi s1, s1, -1
       bne  s1, x0, loop2
end2:  addi sp, sp, 24          ;pop
       ld   x1, 0(sp)           ;pop
       ld   s0, -8(sp)          ;pop
       ld   s1, -16(sp)         ;pop
       jalr x0, 0(x1)           ;return
```

In the above procedure we use the stored registers s0 and s1 to hold the values of the parameters a2 and a3. As s0 and s1 are saved registers, by convention, the callee must preserve their values. Therefore, *delch1* must preserve the values in s0 and s1 (so that the caller delch can count on them staying intact) but *delch* itself must also preserve those values (so that the callers of delch can count on them staying intact.) Similarly, the callee is responsible by convention for saving and restoring the return address in x1. Following the convention we push the values of x1, s0 and s1 into the stack at the beginning of delch and pop them back at the end. You can insert the following placeholder program in front of the procedure for testing:

```
str1: DC "sampled text\0"
STACK: EQU      0x100000        ;stack
       lui      sp, STACK>>12
       addi     a2, x0, str1+6  ;chaddr
       addi     a3, x0, 6       ;#ch
       jal      x1, delch
       addi     x6, x0, str1    ;output
       ecall    x0, x6, 4
       ebreak   x0, x0, 0       ;finish
```

In the above example we deleted 6 characters starting from the ending 'd' of the first word. Try deletions starting at different positions and with different lengths. Save the above example as a file named **c3c05.asm** for possible future use and submit it. In a similar way we can create a procedure *insch1* that inserts a character at a given address in a string. The procedure can have two arguments, the insertion address in `a2` and the character in `a3`.

```
insch1: lb  x5, 0(a2)       ;
        sb  a3, 0(a2)       ;
        addi a3, x5, 0      ;
        addi a2, a2, 1      ;
        bne a3, x0, insch1  ;
        sb  a3, 0(a2)       ;
        jalr  x0, 0(x1)     ;
```

Note how the procedure loops to the end of the string moving all the characters at and after the insertion point backward by one position in order to open space for the inserted character.

What other procedures for operations on strings might be useful? For example, do we need a procedure to replace a character? In fact, replacing a character in a string takes just one instruction:

```
sb a3, 0(a2)
```

The above example assumes that the address of the character to replace is in `a2` and the new character is in `a3`. We can also empty a string by setting its first character to `0`:

```
sb x0, 0(a2)
```

Recursion

Recursive procedures are non-leaf procedures as they call themselves by definition. We will, therefore, use the stack for saving and restoring the return addresses and other values as necessary. Note that we initialize the stack this time differently. We write the value of 0 into the sp register and will assume that sp points to the last used address in the stack (this is different from our previous examples where we assumed that the sp points to the first empty/available address in the stack.) One of the simplest possible recursive procedures is perhaps the one that calculates the factorials as shown below.

```
n        0,1,2,3, 4,  5,  6,   7,    8,     9,     10
fact(n)  1,1,2,6,24,120,720,5040,40320,362880,3628800
```

The sample assembly program could be as follows:

```
        addi    sp, x0, 0       ;sp initialization
        addi    a0, x0, 5       ;n=5
        jal     x1, fact        ;call fact
        ebreak  x0, x0, 0
fact:   blt      x0, a0, recu   ;if(0<a0) recursion
        addi     a0, x0, 1       ;if(a0<=0)return 1
        jalr     x0, 0(x1)       ;return
recu:   sd       x1,  -8(sp)     ;push ra
        sd       a0,  -16(sp)    ;push a0
        addi         sp, sp, -16     ;adjust sp
        addi     a0, a0, -1      ;a0=a0-1
        jal      x1,  fact       ;recursive call
        addi     sp, sp, 16      ;adjust sp
        ld       x1, -8(sp)      ;pop ra
        ld       x5, -16(sp)     ;pop a0
        mul      a0, x5, a0      ;fact(a0)=a0*fact(a0-1)
        jalr     x0,  0(x1)      ;return
```

Save the above example as a file named **c4a05.asm** for possible future use and submit it. The `fact` procedure in the above example either returns the value of `1` for $a0 \leq 0$ or calls itself recursively with the value of a0 decreased by `1`. Note how we calculate the return value in `a0` by multiplying the current value of `a0` restored from the stack to `x5` with the returned from the recursive call value in `a0`. For better understanding, here is the pseudo code of the above procedure:

```
   fact (n) {
        if (n <= 0) return 1;
            return n * fact (n - 1);
    }
```

The above procedure is not *tail recursive* since the call to itself is *not* the last thing it does (in fact it multiplies the result from the recursive call by `n`.) We can, however rewrite the procedure as follows:

```
  fact (n, accumulator) {
        if (n == 0) return accumulator;
            return fact (n - 1, n * accumulator);
}
```

Now the above `fact` procedure makes a function call (to itself) with no further calculations (i.e., a tail call). This function is *tail recursive*. Rewriting a recursive procedure into a tail recursive form can help eliminate the recursion. The assembly version of the above pseudo code could be as follows:

```
      addi a1, x0, 1        ;accumulator
      addi a0, x0, 5        ;n=5
      jal  x1, fact         ;call fact
      ebreak x0, x0, 0
fact: blt   x0, a0, recu    ;if(0<a0) recursion
      add   a0, x0, a1      ;if(a0<=0)return accumulator
      jalr  x0, 0(x1)       ;return
recu: mul   a1, a1, a0      ;fact(a0)=a0*fact(a0-1)
      addi  a0, a0, -1      ;a0=a0-1
      jal   x0,  fact       ;jump instead of a recursive call
```

Save the above example as a file named **c4b05.asm** for possible future use and submit it. Note that the use of the stack in the above procedure is minimal as we only need to store in it the return address.