LAB B: Memory, Branches, and I/O

B1 Data in memory (`DD, ld, sd, DM, ORG`)

(DD)

Constant values can be stored in the memory at compile time using the `DD` (Define Double) assembly command:

```
c:    DD    0x1234567811223344
```

(ld)

Then the constant value can be loaded from the memory to a register at run time using the `ld` (load double) instruction:

```
c:    DD    0x1234567811223344
      ld    x5, c(x0)
```

Save the above example as a file named b1b.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
      x5  t0   0x1234567811223344 1311768465155175236
```

Also check the resulting values in the **Mem** window:

```
0x0000000000000000 0x1234567811223344 1311768465155175236  c:    DD
0x1234567811223344
```

The advantage of the above approach is that there are no limitations about the bit-size of the values in memory (e.g. 64 bit values can be specified directly.) Since those values are not part of the generated instructions (they are not immediate instruction arguments) they can be freely used at different places in the code as needed. The values stored in memory by the compiler can also be modified at runtime by appropriate machine instructions.
The value of the label `c:` in the above example denotes the address (`0` in this case) of the constant stored in memory at runtime by the compiler. This address is then used as an immediate value in the `ld` instruction. The actual address is calculated by summing the value of the base register (in our case `x0` which always contains `0`) with the offset which is the immediate value. This provides for very easy access to values stored in the beginning of the memory e.g. at addresses in the range of `[0,4095]` that can be represented by the `12` bits of the offset only.

(sd)

The `sd` (store double) instruction works in a way opposite to the `ld` instruction. It stores a value from a register into the memory at a specified address.
Type the following instruction in the **Source** window:

```
      sd    x0, 0(x0)
```

Save the above example as a file named b1c.asm for possible future use.
Compile and run the above example. The following error message appears in the **Exec** window:

```
ERROR: memw: #not data adr=0x0000000000000000 len=8
mem(0x0000000000000000)=sd {x0 x0 0x000} {    sd    x0, 0(x0)}
```

The above instruction attempted to write the value of 0 in the memory at address 0, thus overwriting the compiled code as shown in the **Listing** window:

```
0x0000000000000000 S  0000000 00000 00000 011 00000 0100011 sd      x0
x0 0x000       sd      x0,0(x0)                 sd     x0, 0(x0)
```

Let us change the memory address to 4 to point exactly after the compiled instruction.

```
      sd    x0, 4(x0)
```

Save the above example as a file named b1d.asm for possible future use.
Compile and run the above example. The following error message appears in the **Exec** window:

```
ERROR: memw: #alignment problem adr=0x0000000000000004 len=8
```

The above instruction attempted to write an 8-byte word starting at address 4 which is not divisible by 8 which caused the alignment error.
Let us now change the memory address to 8 to point at the first 8-byte word boundary after the compiled instruction:

```
      sd    x0, 8(x0)
```

Save the above example as a file named b1e.asm for possible future use.
Compile and run the above example. Check the resulting value in the **Mem** window:

```
0x0000000000000008 0x0000000000000000 0
0.000000E+000
```

The above line in the **Mem** window indicates that the value of 0 was stored in the memory at address 8.

(DM)

In the following example `addi` stores the value of `0x123` in `x5`. Then `sd` stores the value in `x5` to the memory at the address defined by the label `c`. Storage space of 1 double-word (64 bits) is reserved by the `DM` (Define Memory) assembly command:

```
c:    DM    1
      addi  x5, x0, 0x123
      sd    x5, c(x0)
```

Save the above example as a file named b1f.asm for possible future use.
Compile and run the above example. Check the value of the label `c` in the **Listing** window:

```
0x0000000000000008 START
0x0000000000000000 c
```

Note the value of the START label (8 in this case) that points to the address of the first executable instruction.
In the **Mem** window check the stored value at the address specified by the label c:

```
0x0000000000000008 0x0000000000000123 291                    1.437731E-
321 c:    DM     1
```

(ORG)

Now let us consider the case when some data must be stored at a higher memory address, for example at `0x10000000` as specified by the `ORG` (ORIGIN) assembler command below:

```
      ORG   0x10000000
c:    DD    0x1234567811223344
      ld    x5, c(x0)
```

Save the above example as a file named b1g.asm for possible future use.
Compile the above example and check the **Mem** window to confirm that the constant is properly stored:

```
0x0000000010000000 0x1234567811223344 1311768465155175236  c:    DD
0x1234567811223344
```

In the **Listing** window, however, you will see the following error message:

```
0x0000000010000008 ERROR: imm OUT OF RANGE [-2048,4095]     ld
rd,rs1,imm        ld      x5,c(x0)                  ld     x5, c(x0)
```

Obviously, larger addresses that require more than `12` bits have to be loaded in registers. The following source code illustrates it:

```
      ORG   0x10000000
c:    DD    0x1234567811223344
```

```
      lui   x6, c >> 12
      addi  x6, x6, c & 0xfff
      ld    x5, 0(x6)
```
Save the above example as a file named b1h.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:
```
x5  t0   0x1234567811223344 1311768465155175236
x6  t1   0x0000000010000000 268435456
```
Also check the **Mem** window to confirm that the constant is properly stored:
```
0x0000000010000000 0x1234567811223344 1311768465155175236  c:    DD
0x1234567811223344
```
Another possible approach is to store the (very) large address in the memory at compile time as follows:
```
a:    DD    c
      ORG   0x1000000000000000
c:    DD    0x1234567811223344
      ld    x6, a(x0)
      ld    x5, 0(x6)
```
Save the above example as a file named b1i.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:
```
x5  t0   0x1234567811223344 1311768465155175236
x6  t1   0x1000000000000000 1152921504606846976
```
Also check the **Mem** window to confirm that the constant is properly stored:
```
0x0000000000000000 0x1000000000000000 1152921504606846976  a:    DD
c
0x1000000000000000 0x1234567811223344 1311768465155175236  c:    DD
0x1234567811223344
```

Exercise bex1a: Instruct the assembler to store the values of `1024, 2048, 4096, 8192` in the beginning of the memory using the `DD` assembler command. Sum them, calculate their average using a shift instruction, and store the result in memory right after the compiled code. Save your solution as a file named bex1a.asm for possible future use.

Exercise bex1b: Instruct the compiler to store the values of `0x2222333344445555` and `0x1111222233334444` at addresses `0x1000100010001000` and `0x1000100010001100` respectively. At run time load the values in the registers `x6` and `x7` respectively then calculate the sum, the difference, and the bitwise `or` and `xor` of the two values, storing the results in the registers `x28-x31` respectively. Save your solution as a file named bex1b.asm for possible future use.

Exercise bex1c: Instruct the compiler to store the values of `a=0xAAAABBBBCCCCDDDD` and `b=0x4444333322221111` at addresses `0` and `8` respectively. Calculate `a+b`, `a-b`, `b-a`, `a AND b`, `a OR b`, `a XOR b`, `NOT a`, `NOT b`, then store the results in consecutive double words starting at address 16. Save your solution as a file named bex1c.asm for possible future use.

B2 Branches (bge,beq,bne,jal,jalr,bltu,blt,slt,slti)

(bge)

Calculating the absolute value of an integer is carried out by i) checking whether it is negative and if so ii) negating the value:

```
      If ( i < 0 ) i = -i
```

The above can be implemented in assembler through the conditional branch instruction `bge` (Branch if Greater or Equal). The following sample code takes an integer value from memory address 0, calculates its absolute, and stores the result in the memory right after the compiled code:

```
src:  DD    -3
      ld    x5, src(x0)
      bge   x5, x0, skip
      sub   x5, x0, x5
skip: sd    x5, dst(x0)
      ebreak      x0, x0, 0
dst:  DM    1
```

Save the above example as a file named b2a.asm for possible future use.

Compile and run the above example. Check the resulting values in the **Mem** window:

```
0x0000000000000000 0xfffffffffffffffd -3
0x0000000000000020 0x0000000000000003 3
```

For skipping the negation (implemented by subtraction from 0) the branch instruction `bge` was used the code. This instruction has as a parameter a label (named `skip` in our case) that denotes the instruction to branch to. Note that the value of the label is an absolute address which is used by the assembler to calculate the offset to the target instruction as a relative value with respect to the `bge` instruction. The offset value of `4` in the compiled code in the **Listing** window indicates that the target instruction is `4` pairs of bytes (which is `2` instructions of `4` bytes) after the `beq` instruction:

```
0x000000000000000c SB 0000000 00000 00101 101 00100 1100011 bge     x5
x0 0x004       bge     x5,x0,4                 bge    x5, x0, skip
```

(beq)

In the following example we illustrate the use of the `beq` (Branch if Equal) instruction to organize a loop. The code reads non-zero integers starting from address 0 in the memory and copies them to the memory starting right after the compiled code. The copy process finishes when a 0 value is encountered (the 0 value is not copied):

```
src:  DD    -1, 5, -3, 7, 0
      add   x6, x0,     x0
loop: ld    x5, src(x6)
      beq   x5, x0, end
      sd    x5, dst(x6)
      addi  x6, x6, 8
      beq   x0, x0, loop
end:  ebreak      x0, x0, 0
dst:  DM    1
```

Save the above example as a file named b2b.asm for possible future use.

Compile and run the above example. Check the resulting values in the **Mem** window:

```
0x0000000000000000 0xffffffffffffffff -1
0x0000000000000008 0x0000000000000005 5
0x0000000000000010 0xfffffffffffffffd -3
0x0000000000000018 0x0000000000000007 7
0x0000000000000020 0x0000000000000000 0
0x0000000000000048 0xffffffffffffffff -1
```

```
0x0000000000000050 0x0000000000000005 5
0x0000000000000058 0xfffffffffffffffd -3
0x0000000000000060 0x0000000000000007 7
```
We can easily combine the code from the two previous examples to change the source values to their absolutes in the process of the copying:
```
src:  DD    -1, 5, -3, 7, 0
      add   x6, x0,     x0
loop: ld    x5, src(x6)
      beq   x5, x0, end
      bge   x5, x0, skip
      sub   x5, x0, x5
skip: sd    x5, dst(x6)
      addi  x6, x6, 8
      beq   x0, x0, loop
end:  ebreak      x0, x0, 0
dst:  DM    1
```
Save the above example as a file named b2c.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Mem** window:
```
0x0000000000000000 0xffffffffffffffff -1
0x0000000000000008 0x0000000000000005 5
0x0000000000000010 0xfffffffffffffffd -3
0x0000000000000018 0x0000000000000007 7
0x0000000000000020 0x0000000000000000 0
0x0000000000000050 0x0000000000000001 1
0x0000000000000058 0x0000000000000005 5
0x0000000000000060 0x0000000000000003 3
0x0000000000000068 0x0000000000000007 7
```
(jal)
The offset based method of encoding the branch targets makes it easy to jump to addresses in the vicinity of the branch instruction, e.g. in the range `[PC-4096,PC+4095]` that can be encoded with the `12` bits of the immediate value. Jumps further away e.g. to offsets encoded by up to `20` bits can be achieved by the `jal` (jump and link) instruction:
```
      jal   x0, loop
```
The above instruction, however, is non-conditional but it can be easily combined with conditional branches to extend their jump range:
```
      beq   x5, x6, target
      add   x0, x0, x0
```
(bne)
The above example could be rewritten using the `bne` (branch on non-equal) instruction as follows:
```
      bne   x5, x6, skip
      jal   x0, target
skip: add   x0, x0, x0
```
(jalr)
Note that the above approach still limits the jump range to `[PC-524288,PC+524287]`. For jumps to even more distant addresses the `jalr` instruction which takes the destination address from a register can be used. The `jal` and `jalr` instructions are revisited in LabD where their use for procedure invocation and consequent return is discussed.

Addition and subtraction of large values may lead to an overflow (a value too large to fit in 64 bits) We only need, however, just one extra bit to handle such situations.

(bltu)

Overflow checking for unsigned addition requires only a single additional branch instruction `bltu` (branch on less than unsigned) after the addition:

```
add t0, t1, t2
bltu t0, t1, overflow
```

For signed addition, if one operand's sign is known, overflow checking requires only a single branch instruction `blt` (branch on less than) after the addition. This covers the common case of addition with an immediate operand.

(blt)

For a positive immediate value (denoted by `+imm`) the overflow check could be:

```
addi  t0, t1, +imm
blt   t0, t1, overflow
```

And for a negative immediate value (denoted by `-imm`) the overflow check could be:

```
addi  t0, t1, -imm
blt   t1, t0, overflow
```

(slt, slti)

For general signed addition, three additional instructions after the addition are required, leveraging the observation that the sum should be less than one of the operands if and only if the other operand is negative.

```
add   t0, t1, t2
slti  t3, t2, 0
slt   t4, t0, t1
bne   t3, t4, overflow
```

Exercise bex2a: Given a sequence of non-zero integers followed by 0, find the biggest integer in the sequence and place the result in x5. Use the DD assembler command to store in the beginning of the memory the initial test sequence of `-1, 55, -3, 7, 0`. Save your solution as a file named bex2a.asm for possible future use.

Exercise bex2b: Given a sequence of non-zero integers followed by 0, find the smallest integer in the sequence then swap it with the integer in the beginning of the sequence. Use the DD assembler command to store in the beginning of the memory the initial test sequence of `121, 33, -5, 242, -45, -12, 0`. Save your solution as a file named bex2b.asm for possible future use.

Exercise bex2c: An n-dimensional integer vector can be represented by a sequence of n integers. Use DD assembler commands to store in the beginning of the memory the sample sequences `1, 5, -7, 23, -5` and `3, -2, 4, 11, -7`. Sum the corresponding two vectors and store the resulting sequence in the memory right after the code. Save your solution as a file named bex2c.asm for possible future use.

B3 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 b3a.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 b3b.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 b3c.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 b3d.asm for possible future use.

Compile and run the above example.

Here is a sample dialog:

```
Inputs two integers
and prints the sum.
integer:7
integer:-2
sum:5
integer:4
integer:3
sum:7
integer:Cancelled
```

Exercise bex3a: Ask the user for his name then greet him as per the following dialog:

```
What is your name?
John
Hello John!
```

Save your solution as a file named bex3a.asm for possible future use.

Exercise bex3b: Create a program that i) asks the user to enter an integer value, ii) asks the user to enter a memory address, and then it iii) stores the entered integer value in the memory at the entered address. The above process continues in a loop until cancelled. Save your solution as a file named bex3b.asm for possible future use.

Exercise bex3c: A simple telephone directory can be constructed by DC assembler commands as follows:

```
dir:  DC    "John"
      DC    "11111"
      DC    "Nick"
      DC    "22222"
      DC    "Sara"
      DC    "11111"
      DC    "Nick"
      DC    "33333"
      DD    0
```

Create a program that i) asks the user to enter a phone or a name, ii) searches the directory, and iii) reports the found entries. The above process continues in a loop until cancelled.
The dialog could be as follows:

```
Enter a phone or a name
to search for:John
John 11111
Enter a phone or a name
to search for:11111
John 11111
Sara 11111
Enter a phone or a name
to search for:Nick
Nick 22222
Nick 33333
Enter a phone or a name
to search for:Cancelled
```

Save your solution as a file named bex3c.asm for possible future use.