LAB A: Integers and Binary Data v0.03

A1. Integer arithmetic (`add, addi, sub`)

We will begin with explaining how to instruct the RISC-V CPU to do some simple arithmetic, e.g. to calculate the sum of `2+3`.

(add)

The `add` instruction that uses `3` registers (`x5`, `x6`, and `x7` in the following example) does just that- it sums the `2` integer values in the registers `x6` and `x7` and stores the result in register `x5`:

```
      add   x5, x6, x7
```

(addi)

But how to store the values of `2` and `3` in the registers `x6` and `x7` respectively? The following form of the `addi` (add immediate) instruction stores the immediate value of `2`, provided as part of the instruction code, into the `x6` register:

```
      addi  x6, x0, 2
```

More precisely, the above `addi` instruction sums the value stored in the register specified as its second argument (in this case it is `x0` which always contains the value of `0`) with the immediate value specified as its third argument (`2` in this case) and stores the result (`2` in this case) in the register `x6`.

The assembly code using `3` registers (`x5`, `x6`, and `x7`) to calculate the sum of `2+3` could, therefore, be as follows:

```
      addi  x6, x0, 2
      addi  x7, x0, 3
      add   x5, x6, x7
```

Save the above example as a file named a1a.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5   t0   0x0000000000000005 5
x6   t1   0x0000000000000002 2
x7   t2   0x0000000000000003 3
```

An even shorter version using only `2` registers (`x5` and `x6`) could be as follows:

```
      addi  x6, x0, 2
      addi  x5, x6, 3
```

Save the above example as a file named a1b.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5   t0   0x0000000000000005 5
x6   t1   0x0000000000000002 2
```

Now let us instruct the RISC-V CPU to calculate the difference 2-3. One can easily obtain the correct result by calculating the sum 2+(-3) as follows:

```
      addi  x6, x0, 2
      addi  x7, x0, -3
      add   x5, x6, x7
```

Save the above example as a file named a1c.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5   t0   0xFFFFFFFFFFFFFFFF -1
x6   t1   0x0000000000000002 2
x7   t2   0xFFFFFFFFFFFFFFFD -3
```

(sub)

The above approach, however, works only for subtracting constants that are known at compile time. When subtracting values that are obtained at run time, e.g. input values and results of intermediate calculation, etc., the `sub` (subtract) instruction must be used:

```
addi  x6, x0, 2
addi  x7, x0, 3
sub   x5, x6, x7
```

Save the above example as a file named a1d.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

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

Exercise aex1a: Use a sequence of two `addi` instructions to calculate the value of `11-6` and store the result in `x5`. Save your solution as a file named aex1a.asm for possible future use.

Exercise aex1b: Use a sequence of `addi` instructions and a `sub` instruction to subtract `11` from `6` and store the result in `x5`. Save your solution as a file named aex1b.asm for possible future use.

Exercise aex1c: Calculate the expression (1024-512)-(256-128) and store the result in `x5`. Save your solution as a file named aex1c.asm for possible future use.

A2. Binary shifts (`slli, srli, srai`)

Now let us instruct the RISC-V CPU to calculate the following multiplication by an integer value that is a power of `2`:

```
17 × 8
```

(slli)

The instruction `slli` (shift left logical immediate) below takes the value in register `x6` and shifts its bits to the left while feeding zeros in the least significant bit. The number of shifts is determined by the provided immediate value (the immediate value of `0` produces no shift). The immediate value of `2` in our case produces `2` shifts which effectively multiplies the integer in `x6` by `4` (each `1`-bit shift to the left multiplies the number by `2`) and stores the result in `x5`.

```
slli  x5, x6, 2
```

The assembly code to load the value of `17` in `x6`, and then to multiply it by `4` and store the result in `x5` could, therefore, be as follows:

```
addi  x6, x0, 17
slli  x5, x6, 2
```

Save the above example as a file named a2a.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000000044 68
x6  t1   0x0000000000000011 17
```

(srli)

In a similar way the instruction `srli` (shift right logical immediate) can be used for division by a power of `2`, to calculate for example the following:

```
88 / 8
```

The corresponding assembly source code could be as follows:

```
      addi  x6, x0, 88
      srli  x5, x6, 3
```

Save the above example as a file named a2b.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x000000000000000B 11
x6  t1   0x0000000000000058 88
```

Now let us see what happens when we try to divide a negative value in a similar way:

```
      -88 / 8
```

The corresponding assembly source code could be as follows:

```
      addi  x6, x0, -88
      srli  x5, x6, 3
```

Save the above example as a file named a2c.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x1FFFFFFFFFFFFFF5 2305843009213693941
x6  t1   0xFFFFFFFFFFFFFFA8 -88
```

(srai)

The obtained result in the `x5` register shown above is obviously wrong. The problem is created by the `0` bits fed into the MSB part of the number during the shift. Indeed, the expected result is `-11`, which is a negative number that must have `1` as a most significant bit. Fortunately, this problem is easily solved by using the `srai` (shift right arithmetic immediate) instruction as follows:

```
      addi  x6, x0, -88
      srai  x5, x6, 3
```

Save the above example as a file named a2d.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0xFFFFFFFFFFFFFFF5 -11
x6  t1   0xFFFFFFFFFFFFFFA8 -88
```

Shift instructions are often used for extracting subsequences of bits. The following source code will, for example, move bits `[7:4]` of the value in `x6` to the `4` least significant bits `[3:0]` of `x5` nullifying all its other bits:

```
      addi  x6, x0, 0x123
      slli  x7, x6, 56
      srli  x5, x7, 60
```

The above code uses `slli` to move the `4` bits of interest to the MSB part if the register (bits `[63:60]`) followed by a `srli` to move the `4` bits of interest to the LSB part of the register (bits `[3:0]`), taking advantage of the fact that `srli` feeds `0` bits in the MSB during the shift to clear the bits `[63:4]` on the left.
Save the above example as a file named a2e.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000000002 2
x6  t1   0x0000000000000123 291
x7  t2   0x2300000000000000 2522015791327477760
```

Exercise aex2a: Calculate the value of the expression `(888/8-123*4)*2` and store the result in `x5`. Save your solution as a file named aex2a.asm for possible future use.

Exercise aex2b: Store the value of `0xffffffff00000000` in `x5` using only `addi` and `slli` instructions. Save your solution as a file named aex2b.asm for possible future use.

Exercise aex2c: Store the value of `0x0000123400000000` in `x5` using only `addi` and `slli` instructions. Save your solution as a file named aex2c.asm for possible future use.

A3. Logical operations (`andi`, `or`, `xori`)

(andi)

The following source code will extract bits `[7:4]` of the value in `x6` by directly masking out all other bits using the `andi` (and immediate) instruction. It will then move the bits to the LSB part by a `srli` instruction:

```
addi  x6, x0, 0x123
andi  x7, x6, 0x0f0
srli  x5, x7, 4
```

Save the above example as a file named a3a.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000000002 2
x6  t1   0x0000000000000123 291
x7  t2   0x0000000000000020 32
```

(or)

Another bitwise instruction, often used to combine subsequences of bits, is the `or` instruction:

```
addi  x6, x0, 0x123
andi  x6, x6, 0x0f0
addi  x7, x0, 0x456
andi  x7, x7, 0xf0f
or    x5, x6, x7
```

In the above example we combine the middle hexadecimal digit of the value `0x123` with the 1st and the 3rd hexadecimal digits of the value `0x456` and obtain in result `0x423` in `x5`.
Save the above example as a file named a3b.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000000426 1062
x6  t1   0x0000000000000020 32
x7  t2   0x0000000000000406 1030
```

(xori)

The `xori` (exclusive or) instruction calculates an exclusive `or` of its operands and can be used, for example, to flip sequences of bits. The following assembly source negates all the bits of the value in `x6` and stores the result in `x5`:

```
addi  x6, x0, 0x123
xori  x5, x6, -1
```

Save the above example as a file named a3c.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0xFFFFFFFFFFFFFEDC -292
x6  t1   0x0000000000000123 291
```

Exercise aex3a: Convert `-5` to `+5` by negating its bits and adding `1`. Save your solution as a file named aex3a.asm for possible future use.

Exercise aex3b: Calculate the value of `1234-(567+89)` without using the `sub` instruction. Save your solution as a file named aex3b.asm for possible future use.

Exercise aex3c: Rotate right by 4 bits the value of `0x0000000000000123`. The expected result is `0x3000000000000012`, i.e. all hexadecimal digits move right by one position while the rightmost one moves to the front. Save your solution as a file named aex3c.asm for possible future use.

A4. Loading larger values (`lui, EQU`)

Let us try to use the approach described in the previous sections for calculating the sum of `4098+3`.

Type the following instruction in the **Source** window and press the **Compile** button:

```
addi  x6, x0, 4098
```

The following error message is shown in the **Listing** window:

```
0x0000000000000000 ERROR: imm OUT OF RANGE [-2048,4095]     addi
rd,rs1,imm         addi    x6,x0,4098                  addi   x6, x0, 4098
```

It indicates that the immediate value we supplied as a third argument (`4098` in our case) is out of the allowed range. Indeed, as in the `addi` instruction there are only `12` bits for encoding the immediate values (see the bit allocation of the I-type instructions on the "Green Card") only unsigned integers in the range of `[0,4095]` or signed integers in the range of `[-2048,2047]` could be represented.

Note that the ALU itself does the addition using `64`-bit registers, so all we have to do is to find a way for putting the right values in the registers. One possible approach is to use more bits of the instruction for encoding larger immediate values. Since some bits are still needed for encoding the `rd` and the `opcode`, only immediate values represented by up to `20` bits can be encoded in this way (see the U-type instruction format in the "Green Card").

`(lui)`

The `lui&addi` method can be employed to store in a register a value represented by up to `32` bits. This method uses a sequence of `2` instructions, namely the `lui` (load upper immediate) instruction (stores the most-significant `20` bits) and the `addi` instruction (stores the least-significant `12` bits) as follows:

```
lui   x6, 1
addi  x6, x6, 2
```

Save the above example as a file named a4a.asm for possible future use.

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

```
x6  t1   0x0000000000001002 4098
```

Here is how we derived the constants `1` and `2` in the above `lui` and `addi` instructions. For the decimal value of `4098` we have

```
4098 = 4096+2 = (4096×1) +(256×0) + (16×0) + (1×2) = 0x1002
```

The `32` bit binary representation of the above hexadecimal value is, therefore, as follows

```
0000 0000 0000 0000 0001 0000 0000 0010
```

The most-significant `20` bits of the above value are its first `20` bits from the left which represent the value of `1` used in the above `lui` instruction as an immediate value:

```
0000 0000 0000 0000 0001
```

The least-significant `12` bits of the above value are its last `12` bits on the right which represent the value of `2` used in the above `addi` instruction as an immediate value:

```
0000 0000 0010
```

The assembly code to calculate the sum of `4098+3` could, therefore, be as follows:

```
lui   x6, 1
addi  x6, x6, 2
addi  x7, x0, 3
add   x5, x6, x7
```

Save the above example as a file named a4b.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000001005 4101
x6  t1   0x0000000000001002 4098
x7  t2   0x0000000000000003 3
```

An even shorter version that uses only `2` registers could be as follows:

```
lui   x6, 1
addi  x6, x6, 2
addi  x5, x6, 3
```

Save the above example as a file named a4c.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000001005 4101
x6  t1   0x0000000000001002 4098
```

What if we try to use the same approach for calculating the sum:

```
6146 + 3
```

It seems not to be so obvious how to derive the constants needed for the `lui` and the `addi` instructions manually, but the RVS can help us. The bitwise shift right operation (`>>`) can be used to extract the most-significant `20` bits of the value as follows:

```
6146 >> 12
```

And the bitwise `and` operation (`&`) can be used to extract the least-significant `12` bits as follows:

```
6146 & 0xfff
```

(EQU)

Note that the above calculations are carried out by the RVS at compile time so no machine instructions are actually generated. The calculated values can, however, be assigned to labels for possible future referencing using `EQU` (EQUIVALENT) assembler commands as follows:

```
b20:  EQU   6146 >> 12
b12:  EQU   6146 & 0xfff
```

Save the above example as a file named a4d.asm for possible future use.
Compile the above example. Check the resulting values of the labels `b12` and `b20` in the **Listing** window:

```
0x0000000000000802 b12
0x0000000000000001 b20
```

Based on the above values of `b20` and `b12` we can now easily see that the hexadecimal representation of `6146` is `0x1802` and its binary representation will, therefore, be as follows:

```
0000 0000 0000 0000 0001 1000 0000 0010
```

The most-significant `20` bits of the above value are its first `20` bits from the left which represent the value of `1` to use in the `lui` instruction as an immediate value:

```
0000 0000 0000 0000 0001
```

The least-significant `12` bits of the above value are its last `12` bits on the right which represent `2050` as an unsigned `12` bit value or `-2046` as a signed `12` bit value:

```
1000 0000 0010
```

To avoid possible confusion we will use the `12` bit hexadecimal notation (`0x802`) as an immediate value in the `addi` instruction.

The assembly code to calculate the sum of 6146+3 could, therefore, be as follows:

```
       lui   x6, 1
       addi  x6, x6, 0x802
       addi  x7, x0, 3
       add   x5, x6, x7
```

Save the above example as a file named a4e.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000000805 2053
x6  t1   0x0000000000000802 2050
x7  t2   0x0000000000000003 3
```

The above result is obviously wrong as we got `2053` in `x5`, instead of `6146+3=6149`.
The origin of the problem seems to be the value in `x6` which is shown as `2050` while it should actually be `6146`. But why did we get `2050` in the register `x6`? The reason is because in the RISC-V architecture the supplied immediate value in `addi` and similar instructions is always interpreted as *a signed integer* and is thus sign-extended accordingly. In our case, irrespectively of the way we specify the value, e.g. `2050`, `-2046`, or `0x802`, it will always be treated as `-2046`. Therefore, the first `2` instructions calculated the following expression which gave the actually obtained value in the x6 register:

```
       4096 - 2046 = 2050
```

In our case, what we really want is the `12` bit immediate value to be treated as an *unsigned integer* so that no sign extension is carried out and bits `[31-12]` are set to `0` but the RISC-V instruction set does not allow it. Adding `1` to `x5`, however, solves the problem (see the explanations on p.114 of the course textbook) so the first line of the source code could be changed as follows:

```
       lui   x6, 2
       addi  x6, x6, 0x802
       addi  x7, x0, 3
       add   x5, x6, x7
```

Save the modified source as a file named a4f.asm for possible future use.
Compile and run it. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000001805 6149
x6  t1   0x0000000000001802 6146
x7  t2   0x0000000000000003 3
```

The values calculated by the RVS can be directly referenced in the source as follows:

```
b20:  EQU   6146 >> 12
b12:  EQU   6146 & 0xfff
      lui   x6, b20 + 1
      addi  x6, x6, b12
      addi  x7, x0, 3
      add   x5, x6, x7
```

Save the above example as a file named a4g.asm for possible future use.
Compile and run the above example. Check the resulting values in the **Regs** window:

```
x5  t0   0x0000000000001805 6149
x6  t1   0x0000000000001802 6146
x7  t2   0x0000000000000003 3
```

An even shorter version could be as follows:

```
       lui   x6, (6146 >> 12) +1
       addi  x6, x6, 6146 & 0xfff
       addi  x5, x6, 3
```

Save the above example as a file named a4h.asm for possible future use.

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

```
x5   t0    0x0000000000001805 6149
x6   t1    0x0000000000001802 6146
x7   t2    0x0000000000000003 3
```

The last 2 examples illustrate how the assembler can relieve us from some manual calculations in the source text. Further enhancements are possible by introducing conditional assembly which could allow, for example, automating the process of adding `1` to the argument of the `lui` instruction depending on the sign of the value represented by the least significant `12` bits of the supplied constant.

Finally, what about constants that are represented by more than `32` bits? To load a `64` bit constant, for example, we could first split it into two `32` bit values, then employ `lui` to load the values into registers, and finally combine the two `32` bit values in one register.

```
c:     EQU   0x1234567811223344
       lui   x6, (c & 0xffffffff) >> 12
       addi  x6, x6, c & 0xfff
       lui   x7, c >> 44
       addi  x7, x7, (c & 0xfff00000000) >> 32
       slli  x7, x7, 32
       or    x5, x6, x7
```

Save the above example as a file named a4i.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    0x0000000011223344 287454020
x7   t2    0x1234567800000000 1311768464867721216
```

The value in the `x7` register was obtained by loading the necessary value in its lower `32` bits and then shifting the `x7` register bits to the left for 32 times through the `slli` instruction. The final value in the `x5` register was obtained by bitwise `or` of the values in the registers `x6` and `x7`.

The above example clearly shows that loading very large constants into registers by employing immediate constants embedded into the instructions is not quite trivial. In addition, the above code will require further modifications to account for the cases when the constant breaks into negative values.

Exercise aex4a: Use the `lui` and `addi` instructions to store the value of `8000` in `x6`, and then use the `addi` instruction to calculate the value of `8000-20` and store the result in `x5`. Save your solution as a file named aex4a.asm for possible future use.

Exercise aex4b: Use the `lui` and `addi` instructions to store the value of `-8000` in `x6`, and then use the `addi` instruction to calculate the value of `-8000+20` and store the result in `x5`. Save your solution as a file named aex4b.asm for possible future use.

Exercise aex4c: Use the `lui` and `addi` instructions to store the value of `23456` in `x6` and the value of `12345` in `x7`. Then use the `sub` instruction to calculate the value of `23456-12345` and store the result in `x5`. Save your solution as a file named aex4c.asm for possible future use.

Exercise aex4d: Use only `addi`, `lui`, `slli` and `add` instructions to store in `x5` the value of `0x1234587811223333`. Save your solution as a file named aex4d.asm for possible future use.