LabB: Memory and Branches

B1 Assembly operations and commands (>>, &, EQU, DD)

It is often necessary to do calculations using different constants when writing assembly language programs. For example if we want to load a large constant into a register using the `lui&addi` method, we need to separate its `20` most significant bits from its `12` least significant bits. RVS offers a set of assembler operations that can be used for this purpose. Note that these operations help to build (assemble) the machine code and are not executed by the processor when running the program.

(>>)

The RVS bitwise operation *shift right* (`>>`) can be used to extract the most-significant `20` bits of the value as follows:

```
6146 >> 12
```

(&)

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

```
6146 & 0xfff
```

Note that the above calculations are carried out by the RVS at compile time so no machine instructions are actually generated.

(EQU)

Values calculated with assembly operations can be assigned to labels (`b20` and `b12` in the example below) for possible future referencing using the `EQU` (EQUIVALENT) assembler command as follows:

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

Save the above example as a file named b1a05.asm for possible future use and submit it.
Compile the above example. Check the resulting values of the labels `b12` and `b20` in the **Listing** window:

```
SYMBOL TABLE
0x0000000000000802 b12
0x0000000000000001 b20
```

Labels with their values (integer constants) appear in the assembler `SYMBOL TABLE` and are used for referring to the respective integer constants in the assembly language code. Labels are named integer constants and can be used as immediate values or addresses referring to different places in the computer memory.
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 immediate value of `1` to use in the `lui` instruction:

```
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
```

Now let us revisit the `lui&addi` method and try to calculate the sum `6146 + 3`. 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 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 b1b05.asm for possible future use and submit it.
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 the 20-bit value loaded with lui to x6, 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 b1c05.asm for possible future use and submit it.
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 b1d05.asm for possible future use and submit it.
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 b1e05.asm for possible future use and submit it.

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

```
x5  t0   0x0000000000001805 6149
x6  t1   0x0000000000001802 6146
```

The last 2 examples illustrate how the assembler can relieve us from some manual calculations (and thus possibilities for error) 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&addi` method 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 b1f05.asm for possible future use and submit it.

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 (ones at bit `11` or `43`).

(DD)

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

```
c:    DD    0x1234567811223344
```

Note that the above assembly command is processed at compile time and no machine instructions are actually generated. More details about the RVS `Define` commands (`DD, DW, DH, DB, DC, DM`) are provided in the RVS User Manual.

Note the difference between the `EQU` and the `DD` assembly language commands:

```
c1:   EQU   0x123
c2:   DD    0x123
```

Save the above example as a file named b1g05.asm for possible future use and submit it.

Compile the above example. Check the resulting values in the **Listing** window:

```
SYMBOL TABLE
0x0000000000000123 C1
0x0000000000000000 C2
```

The above `EQU` command assigns the value of `0x123` to the label `c1`. We can thus use the name `c1` instead of the constant `0x123` itself, e.g. in immediate instructions such as `addi`,

ori, lui, etc. On the other hand, the above DD command assigns to the label c2 the memory address where the constant 0x123 is stored. We can thus use the name c2 instead of the ***address*** of the constant 0x123, e.g. in memory instructions such as ld, sd, etc.

B2 Memory instructions and assembly language commands (ld, sd, DM, ORG)

(ld)

A constant value provided in a DD command 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 b2a05.asm for possible future use and submit it.
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 b2b05.asm for possible future use and submit it.
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 b2c05.asm for possible future use and submit it.
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 b2d05.asm for possible future use and submit it.
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 b2e05.asm for possible future use and submit it.
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:
```
0x0000000000000000 0x0000000000000123 291                    c:    DM
1
```

(ORG)

The `ORG` (ORIGIN) assembler command changes the compilation memory address. Data can be stored at a higher memory address, for example at `0x10000000`, as follows:
```
      ORG   0x10000000
c:    DD    0x1234567811223344
      ld    x5, c(x0)
```
Save the above example as a file named b2f05.asm for possible future use and submit it.
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 using the `lui&addi` method:
```
      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 b2g05.asm for possible future use and submit it.

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 b2h05.asm for possible future use and submit it.
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
```

B3 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 labelled by `src`, 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 ;negate
skip: sd    x5, dst(x0)
      ebreak      x0, x0, 0
dst:  DM    1
```

Save the above example as a file named b3a05.asm for possible future use and submit it.
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 in 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 ;0 marks the end
      sd    x5, dst(x6)
      addi  x6, x6, 8
      beq   x0, x0, loop
end:  ebreak      x0, x0, 0
dst:  DM    1
```

Note that the above `DM` assembly command reserves storage for a single integer right after the end of the program. The remaining integers are copied to the unused memory that follows.
Save the above example as a file named b3b05.asm for possible future use and submit it.
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 ;0 marks the end
      bge   x5, x0, skip
      sub   x5, x0, x5  ;negate
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 b3c05.asm for possible future use and submit it.
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
```