Assembly Language

(Read professor Roumani's Lecture notes and sample codes on assembly language. This is just a supplement to those.)

In this course we will use SPIM (the MIPS simulator) to write assembly programs.

Instructions:

- Instructions → Genuine instructions
- Pseudo Instructions → Replaced by real instructions
- Directive Instructions → Help the assembler run the instructions

Every line in a MIPS program has the following:

```
Label:      instruction operands    #sh-like comments
```

(Labels are optional. Leave a column for labels, whether you write them or not.)

Numbers and Strings

- Numbers: Decimal, Octal (starting with 0) and Hexadecimal (starting with 0x)
- Strings: "between quotes", Java-like special characters (\t, \n, etc.) are supported.

Note: In assembly we should store variables in registers and memory. We always use registers unless they are full. SPIM has 32 registers and they all have nicknames.

div & mult

Division using *div* returns both the quotient and the remainder and stores them in two new non-programmable registers LO and HI. (The former is the quotient, the later is the remainder.)

Multiplication using *mult* is done the same way except this time the low-order word is stored in LO and the high-order word is stored in HI.

We can not access registers LO and HI directly, but we can assign their values to other variables using `mflo` and `mfhi`:

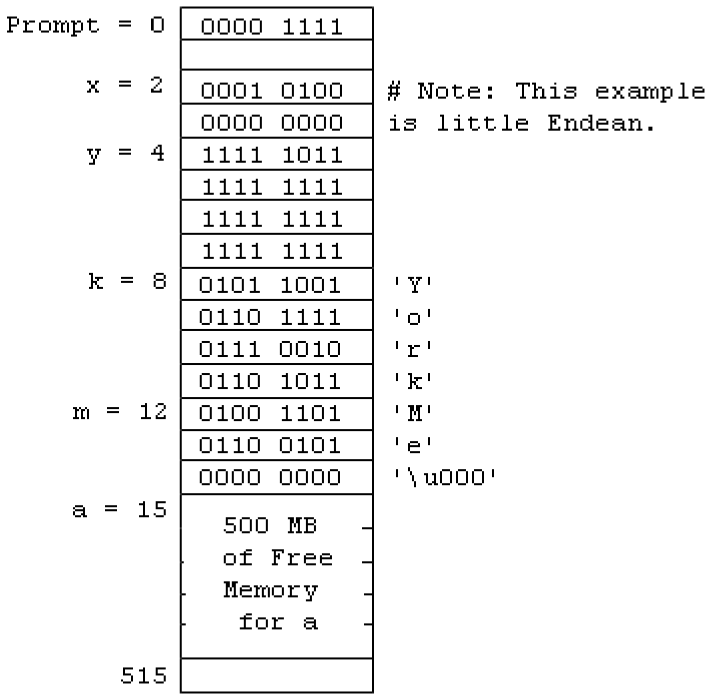
```
mflo $t2    # t2 = lo
mfhi $t2    # t2 = hi
```

Strings are too large for registers → we have to store them in DRAM:

This is how we store data in memory. We can then call them using the labels just like variable names (later labels are converted to memory address automatically). In the comments you can see the Java syntax.

```
            .text
            #instructions go here.
            .data
prompt:     .byte       7           # byte prompt = 7;
x:          .half       20          # short x = 20;
y:          .word       -5          # int y = -5;
k:          .ascii      "York"      # String k = "York";
m:          .asciiz     "Me"        # String m = "Me\u000";
a:          .space      500         # 500 MB of uninitialized memory
```

This is how they will be stored in memory: (Remember the rules for storing data in memory!)



Jump (j, jr) vs. Branches (beq, bne, etc.):

Branches are conditional jumps.

beq = Branch Equal to

bne = Branch Not Equal to

bltz/bgtz = Branch less than/greater than zero

blez/bgez = Branch less than/greater than or equal to zero

In assembly there are no blocks: We can not run a few instructions for each state of the branch. Instead we jump to labels. See the following example (and its Java equivalent):

```
main: add $s1, $0, 5                  Java:
      add $s2, $0, 7                  int s1, s2, s3;
      slt $t0, $s1, $s2               s1 = 5;
      bne $t0, $0, less               s2 = 7;
      addi $s3, $0, -20               if ( s1 < s2 )
      j done                             s3 = 20;
less: addi $s3, $0, 20                else
      j done   # not necessary           s3 = -20;
done:
```

Note that there is no such thing as *less than* in MIPS (except for zero) so we used `slt` (set less than) which sets t0 to 1 if s1 < s2 and 0 otherwise

The jump family:

```
j skip         # Jump to label "skip"
```

- `j` puts the address of label in register PC

    (PC contains the next instruction that will be executed.)

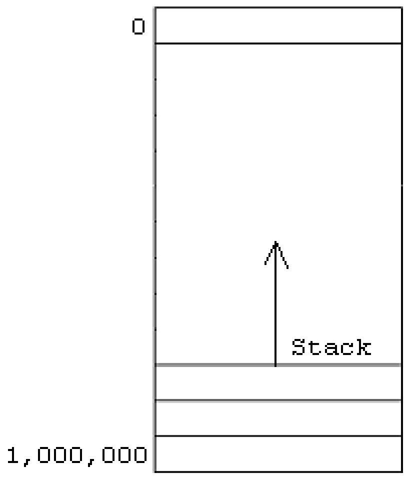
- `jal` does the same thing, but it also stores the address of the next instruction into register `$ra`.

- There is only one register `$ra` and it is already used by the operating system. (Remember the last line of our programs is `jr $ra`.) So in order to use it again we have to store it's value in another register (say `$s0`).

The Stack

What if we need to write a recursive code? We won't have enough registers.

The end of the memory is used as a stack:



Register `$sp` points to the top of the stack (first empty byte):

```
sw    $ra, 0($sp)        # push $ra into the stack
addi  $sp, $sp, -4       # we must change the value of sp

addi  $sp, $sp, 4
lw    $ra, 0($sp)        # pop from the stack into $ra
```