Assembly Cheatsheet

Every computer program needs to have several translation layers, one of them is assembly.
It lays between the high-level programming languages and the machine code.

Note that in this class we are using a simplified version of intel x86 assembly code. Also keep in mind that every computer architecture has a different set of instructions, and thus different assembly code.

Registers

Blocks of memory that we use when programming is assembly. In real life they are a bit more complicated that we what we use in class. The registers can be accessed using Address Modes.

Directive

> What is a directive? Any operation that we do in assembly, they are the "command" that we use while coding.

An example of a directive is `mov`, used to move or store memory from one place to another.

Address Modes

In class we look at 4 different types of address modes:

- **Direct** `mov A, 2`: Stored in `A`, the value given to by memory address `2`
- **Immediate** `mov A, #2`: Stored in `A` the value `2`.
- **Indexed** `mov A, 3[X]`: Stored in `A`, the value stored in address `3 + X`.
- **Indirect** `mov A, (2)`: Stored in `A` the value that is the address that is the value of address `2`. In other words, `2` has a number in it, this number is the address of the value that is stored in `A`.

Variable Initialization

Using the directives `db`, `dw`, `dd`, `d1` and `dt`, we can define variables in assembly usually at the top of the file or program. The following directives are used:

- `DB` define Byte: allocates 1 byte of memory
- `DW` define Word: allocates 2 bytes of memory
- `DD` define Doubleword: allocates 4 bytes of memory
- `DQ` define Quadword: allocates 8 bytes of memory
- `DT` define Ten bytes: allocates 10 bytes of memory

We usually will use `DD`, as it allocates the amount of bytes that an integer has, and most exercises work with integers.

Defining the variables at the top of the file, you can work with them directly, without needing to invoke registers.

MOV

The most basic instruction, used to move memory between the registers.
It is read as `move destination source`, thus it can be visualized using an arrow pointing to the left:

```
mov destination <- source
```

It will always take two operands. The destination and the source.

It can also be used on variables:

```
dd A
MOV R1 [A]
```

Means that we moved the variable `A` into `R1` (register one).

Arithmetical Operations (ADD, SUB, INC, MUL...)

There are several arithmetical (math) operations, of course. They work on the same principle as MOV, first determining the destination and then the source.

However, it is not `destination num1 num2`, instead the destination is one of the operands:

```
ADD destination, source == [ destination = (destination + source) ]
```

The value of the destination is overwritten. If we do not want this behavior, we should duplicate the data stored in the destination to not lose it.

```
MOV R1 [A]
MOV R2 [B]

MOV R3 R2

ADD R3 R1
```

In this case, we have 3 registers, R3 with the result of the operation and the other two with the operands. Note how we copied the value of the variable B into another register.

This operation can also be performed directly with the variables, as an example:

```
MOV R1, [C]
MUL R1, [D]
ADD R1, [B]
MOV [A], R1
```

Translation:

1. Store `C` in `R1`
2. Multiply `R1` with `D`
3. Add `B` into `R1`
4. Store content of `R1` in `A`
   Thus, the operation `A = B + C * D` was performed.
   Keep in mind that we have the initialize the variables before all of this:

```
A dd
B dd
C dd
D dd
```

Jump (Conditional Control Flow)

We need to use jumps into the instructions to get some control flow. If we want to do an `if` statement we need a way to jump some amount of code, skipping it if the condition is not meet.

In class, we are using another type of simplified notation/syntax for the jump on assembly:

- `JMP`: jump directly to location
- `JL`: jump if less than
- `JLE`: jump if less or equal than
- `JG`: jump if greater than
- `JGE`: jump if greater or equal than

First we would use a `CMP` instruction to specify the numbers that need to be compared. We need to specify as well, the place that we want to jump to.

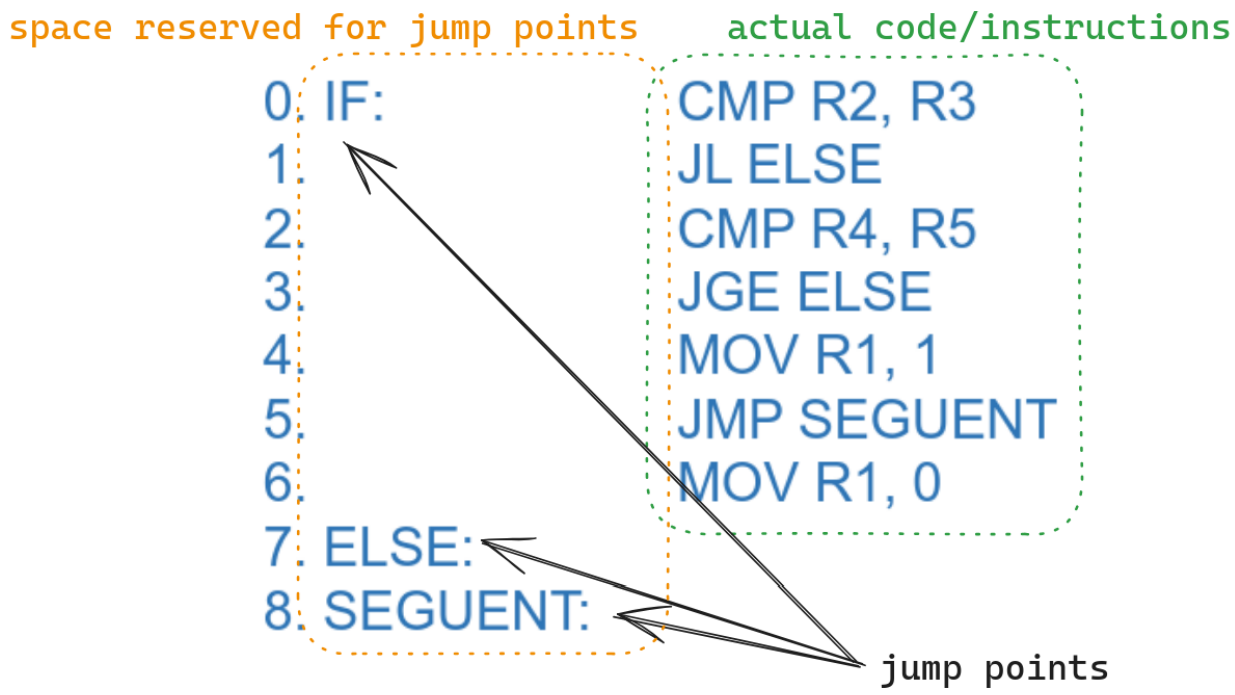
Example:

```
                CMP R1, R2
                JL else

                // more intruccions

                JMP end // jump directly to end

else:   // do something
                // more instruccions
```

```
end:
                // the program continues after the if/else statement
```

Jumps can also be performed without any condition, just do not use `CMP`.

Note how the actual code is indented to the right. This is done on purpose to get enough space to write the jump points.



Function Calls and the Stack

A set of Directives is define when dealing with function calls. To use them we need the Function Stack and the Stack Pointer (SP).

- `PUSH [(SP) <- value, SP <- SP-1]` Store at Stack
- `POP [SP <- SP+1, (SP) <- value]` Retrieve from Stack
- `CALL RoutineX [(SP) <- PC+1, SP <- SP-1]` Call Routine (function)
- `RET [SP <- SP+1, (PC) <- (SP)]` Return value from Routine

More in depth:

`PUSH` and `POP`

These two Directives go hand in hand, one cannot exist without the other. If we want to temporally take out the value from a register without loosing it, we use these functions.

Often there is the case where we must change a register in order to perform some operation, but in that register we have that we do not want to loose.

For example, in the real world the return of a function (Routine) always goes into the `EAX` register. If we do not want to loose the information that is has, we first use `PUSH` and then `POP`:

- `PUSH R1` - preserves the value of `R1`
- `POP R1` - restores the original value of `R1`
  Thus, we must always call first `PUSH` and then `POP`.

Definitions:

```
PUSH [(SP) <- value , SP <- SP-1]

POP [SP <- SP+1 , value <- (SP)]
```

Where value is the register that we give as input. Note that we are working with its content, no the address of the register.

In natural language:

- `PUSH`:

    1. Stores `value` into `(SP)`
    2. Decrease `SP` content by 1
- `POP`
    1. Increase `SP` content by 1
    2. Store `(SP)` in `value`

What is the function of `PS` (stack pointer) here:

> Note that when we are saving the value, we are doing so in the address given to by the pointer. Thus, to not overwrite the stored information, we must change the pointer.
>
> This pointer works as a way to keep track where the `value` is stored. We increased by one at the end of the `PUSH` to get a fresh address that has nothing important in it. If we do not do this, then in the next `PUSH` the information stored will be overwritten.

Image that we do the following:

```
PUSH R1
PUSH R2
```

If we `PUSH` two times, the pointer has been updated to times in a row. Therefore to get the value of `R1` we must first get the value of `R2`:

```
PUSH R1
PUSH R2

PUSH R2
PUSH R1
```

It will always work in reverse.

If we do not do `PUSH R2`, when using `PUSH R1` we are storing the contents that were on `R2` into `R1`. The pointer keeps track of where the information stored, not where did it come from (the register given as input).

Another what to think about it is in the form of questions where the operations performed gives the answer:

- `PUSH`:
    1. Stores `value` into `(SP)`. *Where do I need to store the `value` now?*
    2. Decrease `SP` content by 1. *Where do I need to store the information next time?*
- `POP`
    1. Store `(SP)` in `value`. *Where is the current information stored? I want to get it, store it into `value` please*
    2. Increase `SP` content by 1. *Where should I store the information next time?*

Here I changed the order of the operation, I believe it does not matter.

As a final note, this two functions can work using the `PS` in reverse, with `PUSH` increasing the pointer and `POP` decreasing it. It should not affect the functionality, as the only thing that they need is to coordinate the pointer.

`CALL` and `RET`

These Directives are used to call a subroutine or routine and get the return value from it. They also make use of the stack in order to store information.

The key idea is that a program counter (PC) is needs to return to the place where the routine was called, once it has been executed in order to procedure with the program. If you call a function at "line" 100, that it starts at "line" 300, then once its finished, the program need to return to line 101.

This is solved storing the `PC` after the call into the stack, then grabbing it back once the routine execution is finished.

Definitions:

```
CALL [(SP) <- PC+1 , SP <- SP-1]

RET [SP <- SP + 1, PC <- (SP)]
```

They have a very similar behavior that of `PUSH` and `POP`, as they are doing exactly the same but with `PC` instead of a register that is given as input.

Explanation in natural language:

- `CALL`
  1. Store `PC+1` into `(SP)`
  2. Decrease `SP` content by 1
- `RET`
  1. Increase `SP` content by 1
  2. Store `(SP)` in `PC`

The only thing that happens is that the program counter is saved somewhere, so that when the routine finished, the program can resume at the point where the routine was called.

Is like saving the current state of the program (execution) to do something else (the routine), and continue working in that state when we are finished.

Note that some functionally is missing from this functions that in the real world they perform. For example, when calling a routine, the `PC` is updated with the location of the routine.