Appendices

A Useful instructions to implement linear and circular buffer action

- `MOV` **source, destination**

  Source ⇒ destination

  Where, "source" could be a value, auxiliary register, accumulator or a temporary register. "Destination" could be an auxiliary register, accumulator or a temporary register.

- `ADD` **source, destination**

  destination = destination + source.

  Where, "source" could be a value, auxiliary register, accumulator or a temporary register. "Destination" could be an auxiliary register, accumulator or a temporary register.

- `SUB` **source, destination**

  destination = destination - source.

- `MAC` **mul 1, mul 2,** `ACx`

  This single instruction performs both multiplication and accumulation operation. "x" can take values from 0 to 3.

  `ACx` = `ACx` + (mul_1 * mul_2);

  mul_1 and mul_2 could be an `ARn` register, temporary register or accumulator. (But both cannot be accumulators).

- `BSET ARnLC`

  Sets the bit `ARnLC`.

  The bit `ARnLC` determines whether register `ARn` is used for linear or circular addressing.

  `ARnLC`=0; Linear addressing.

  `ARnLC=1`; Circular addressing.

  By default it is linear addressing. "n" can take values from 0 to 7.

- `BCLR ARnLC`

  Clears the bit `ARnLC`.

- `RPT #count`

  The instruction following the `RPT` instruction is repeated "count+1" no of times.

- `RPTB label`

  Repeat a block of instructions. The number of times the block has to be repeated is stored in the register `BRC0`. Load the value "count-1" in the register `BRC0` to repeat the loop "count" number of times. The instructions after `RPTB` up to label constitute the block. The instruction syntax is as follows

  Load "count−1" in `BRC0`
  `RPTB` label
  · · · block of instructions · · ·
  Label: last instruction

  The usage of the instruction is shown in the sample asm code.

- `RET`

  The instruction returns the control back to the calling subroutine. The program counter is loaded with the return address of the calling sub-routine. This instruction cannot be repeated.

B Important points regarding assembly language programming

- Give a tab before all the instructions while writing the assembly code.
- In Immediate addressing, numerical value itself is provided in the instruction and the immediate data operand is always specified in the instruction by a `#` followed by the number(ex: `#0011h`). But the same will not be true when referring to labels (label in your assembly code is nothing more than shorthand for the memory address, ex: `firbuff` in your sample codes data section). When we write `#firbuff` we are referring to memory address and not the value stored in the memory address.
- Usage of `dbl` in instruction `MOV dbl(*(#_inPtr)), XAR6`

  `inPtr` is a 32 bit pointer to an `Int16` which has to be moved into a 23 bit register. The work of `dbl` is to convert this 32 bit length address to 23 bit address. It puts bits `inPtr(32:16)` ⇒ `XAR6(22:16) and inPtr (15:0)` ⇒ `XAR6(15:0)`

  Example: In c code, the declaration `Int16 *inPtr` creates a 32 bit pointer `inPtr` to an `Int16` value. Then the statement `MOV dbl(*(#_inPtr)),XAR6` converts the 32 bit value of `inPtr` into 23 bit value. If `inPtr` is having a value 0x000008D8 then `XAR6` will have the value 0008D8 and `AR6` will have the value 08D8. So any variable which is pointed by `inPtr` will be stored in the memory location 08D8. We can directly access the value of variable pointed by `inPtr` by using `*AR6` in this case.
- If a register contains the address of a memory location, then to access the data from that memory location, `*` operator can be used.
- `MOV *AR1+, *AR2+`

  The above instruction will move "the contents pointed" by `AR1` to `AR2` and then increment contents in `AR1`, `AR2`.
- To view the contents of the registers, go to `view` ⇒ `registers` ⇒ `CPU register`.
- To view the contents of the memory, go to `view` ⇒ `memory` ⇒ enter the address or the name of the variable.

C Some assembly language directives

- `.global`: This directive makes the symbols global to the external functions.
- `.set`: This directive assigns the values to symbols. This type of symbols is known as *assembly time constants*. These symbols can then be used by source statements in the same manner as a numeric constant. Ex. `Symbol .set value`
- `.word`: This directive places one or more 16-bit integer values into consecutive words in the current memory section. This allows users to initialize memory with constants.
- `.space(expression)`: The `.space` directive advances the location counter by the number of bytes specified by the value of expression. The assembler fills the space with zeros.
- `.align`: The `.align` directive is accompanied by a number (X). This number (X) must be a power of 2. That is 2, 4, 8, 16, and so on. The directive allows you to enforce alignment of the instruction or data immediately after the directive, on a memory address that is a multiple of the value X. The extra space, between the previous instruction/data and the one after the `.align` directive, is padded with NULL instructions (or equivalent, such as `MOV EAX, EAX`) in the case of code segments, and NULLs in the case of data segments.