Computer architectures LAB 3

Word instructions

Addressing Modes

Addressing made easy

Supplementary examples & exercises.

L. Bruynseels ludo.bruynseels@kuleuven.be

# Goals of this lab session

* Work with word instructions
* Addressing modes.
* Understand the stack and function calls: examples of indirect & index addressing.

# Word instructions

16-bits registers: PC, SP, X, Y

Word instructions accept a word (16 bits) as destination and/or source

Examples: ADDW, CLRW, CPW, CPLW, INCW, LDW , POPW, PUSHW, ..

For detailed information: see PM0044, Programming Manual.

# Addressing modes

Addressing modes play an important role in the performance of a computer. Also, a good understanding of addressing modes helps in understanding high level language concepts like scope of a variable and calling conventions (eg passing parameters by value or by reference)

* Immediate : LDA , #5 => the number ‘5’ is loaded into register A
* Direct: LDA, 5 => The value in memory address ‘5’ is loaded into register A
* Indexed:

LD A, (3,X) => The value in memory address X+3 is loaded in register A (X must be preloaded with a valid base address)

LD A, (2,SP) => The value in memory address SP+2 is loaded in register A

* Pointer: LD A, [pointer] => ‘pointer’ contains an address. At this address, the value we want can be found.

# Exercise 1 – Stack pointer

## Clone (download) <https://github.com/ludobruynseels/GroepT-CA-LAB3.git>

* Set ‘Stackpointer’ as active project.
* Set ‘Simulator’ as Debug Instrument.

=> You can study this exercise without the board.

=> The problem with the last example (recursion) is always there but only visible when testing with the simulator.

## Example1

* + Java equivalent:

**byte c = optellen(0x20, 0x0B)**

* + Step with the debugger through every step and watch carefully what happens on the stack.

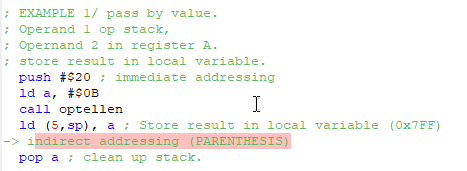
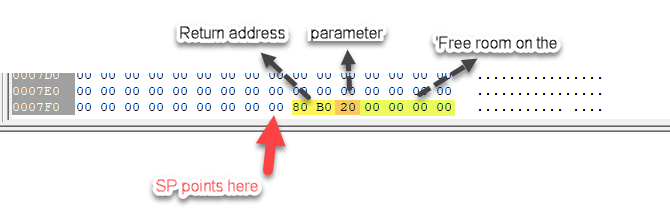
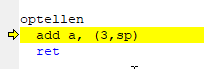


Figure 1 The stack just after we enter ‘optellen’.

After the call, the result of the addition is stored on the stack. This is how **local variables** work => scope limited to enclosing block {…work…} in C, C++, Java, C#.

Afbeelding met tafel

Automatisch gegenereerde beschrijving

Figure 2: The stack after we return from the call

## Example 2: Result is stored in address cResult1.

Step through de code with the debugger and watch carefully what happens on the stack and in memory address cResult1.

## Example 3: pointer

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Figure 3: Excerpt from file ./debug/main.lst

Adres of pointer = $8082

Content of memory cell $8082 = $8080

Content of memory cell $8080 = $0A

* LD A, [pointer] will load $0A into register A (Accumulator)

## Example 4: pass by value + direct addressing:

* Step through de code with the debugger and watch carefully what happens on the stack and in memory address cResult1.
* Attention to the notation: there is no ‘#’

## Example 5: pass by reference : we load the address of the variable.

* Step through de code with the debugger and watch carefully what happens on the stack and in memory address cResult1.
* Attention to the notation: ‘#’ makes the difference. (the devil is in the details!)

## Example 7: Recursion

* This code fragment calculates = N\*(N+1)/2 with recursion. In the example N = 127.
* Try the find the maximum value for which this call works. YOU CAN ONLY FIND OUT IF YOU THE SIMULATOR. If you run it on the board, all disaster remains under water and invisible.
* Can you explain this?

Tip: look in file Stackpointer/Include files/mapping.inc

## Quiz

What is the function of this code fragment that appears in every main.asm as generated by STVD by default?

Afbeelding met tekst

Automatisch gegenereerde beschrijving

Figure 4: Excerpt from startup code.

# Exercise 2 – Nightrider II

Right click on on the ‘nightriderii’ project and set as active project.

TIM3 generates on interrupt every x seconds. The interrupt service routine (ISR) and the timer are properly configured.

Your task is the complete the isr such that a new pattern is loaded every time the interrupt occurs. This pattern is written to the output ports. You should see changing pattern on the LED’s. Once all the pattern have been used, restart.

Think of patterns as an array of bitpatterns: patters[numsteps]. Numsteps is the number of patterns in the array.

patterns [0] is at memory addres numSteps.

patterns [1] is at memory address numSteps + 1. And so on…

In this case, you cannot use SP as register but X will do.

LDW X, #2

lD A, (patterns,X) => loads patterns[2]

LD PD\_ODR, A => loads the pattern to the output ports.

Attention 1: you must keep your loop counter save in a memory location between subsequent interrupts. You may not assume that any register holds it value between interrupts.

Attention 2: run this on the board, not on the simulator.

Attention 3: set the jumper correctly.

You can edit the source file and design you own sequence.

# Exercise 3 – Scale (toonladder)

* Right click on the ‘scale’ project and set as active project.
* Connect the speaker by putting the jumper in the right position.
* Basically, it is the same as the previous exercise (ie play every note in the array ‘notes’ ) but now you must also configure the TIM2 in order to generate the correct frequency. (there are 8 notes). C4 is the central C (ut – do) on the piano.
* To configure TIM2:
  + tones[i] -> TIM2\_ARR
  + tones[i] / 2 -> TIM2\_CCR1
  + divide by 2 == shift to right in binary.

LDW x, tones[i]

LD A, XH

LD TIM2\_ARRH, A

etc…

* To calculate extra tones: divide 2MHz by the desired frequency.
* Examine the role of TIM3\_PSCR and TIM3\_ARRH, TIM3\_ARRL Look in the reference manual. These registers may be used as TIMx\_PSCR. This is because there are more timers TIM1, TIM2, … and this functionality is the same for all timers.

# Exercise 4 – Song

Right click on the ‘song project and set as active project.

Suggestion:

Make an array with as much elements as there are notes in the tune

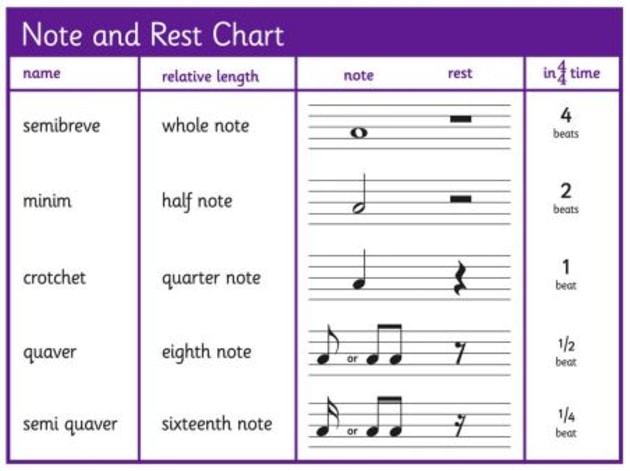
First try to play the right notes in the right order. The length can be adjusted afterwards.

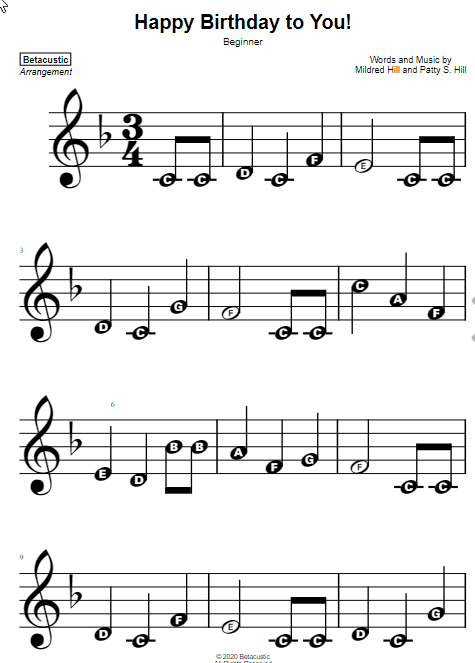
This is a general rule in programming in general and in in assembly in particucal: don’t try to do too much at once. Split the work in many small tasks that can be tested and debugged individually.

The duration of a note can be implemented in 2 ways:

1) change the value of ARR in TIM3

2) look at TIM3 as a metronome that ticks away 8th notes. So, an 8th note last 1 tick, a crotchet 2 ticks, a minim 4 ticks. => relative length in table hereunder.





|  |  |  |
| --- | --- | --- |
| Tone | Index in Tones[] | Relative length |
| C | 0 | 1 |
| C | 0 | 1 |
| D | 1 | 2 |
| C | 0 | 2 |
| F | 2 | 2 |
| E | 3 | 4 |
| C | 0 | 1 |
| C | 0 | 1 |
| D | 1 | 2 |
| C | 0 | 2 |
| G | 4 | 2 |
| F | 3 | 4 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |