University of Windsor

ELECTRICAL AND COMPUTER ENGINEERING
ELEC3270: MICROPROCESSORS

ASSIGNMENT 5: INTRODUCTION TO MINIIDE & THRSIM11, AND INSTRUCTION DELAYS

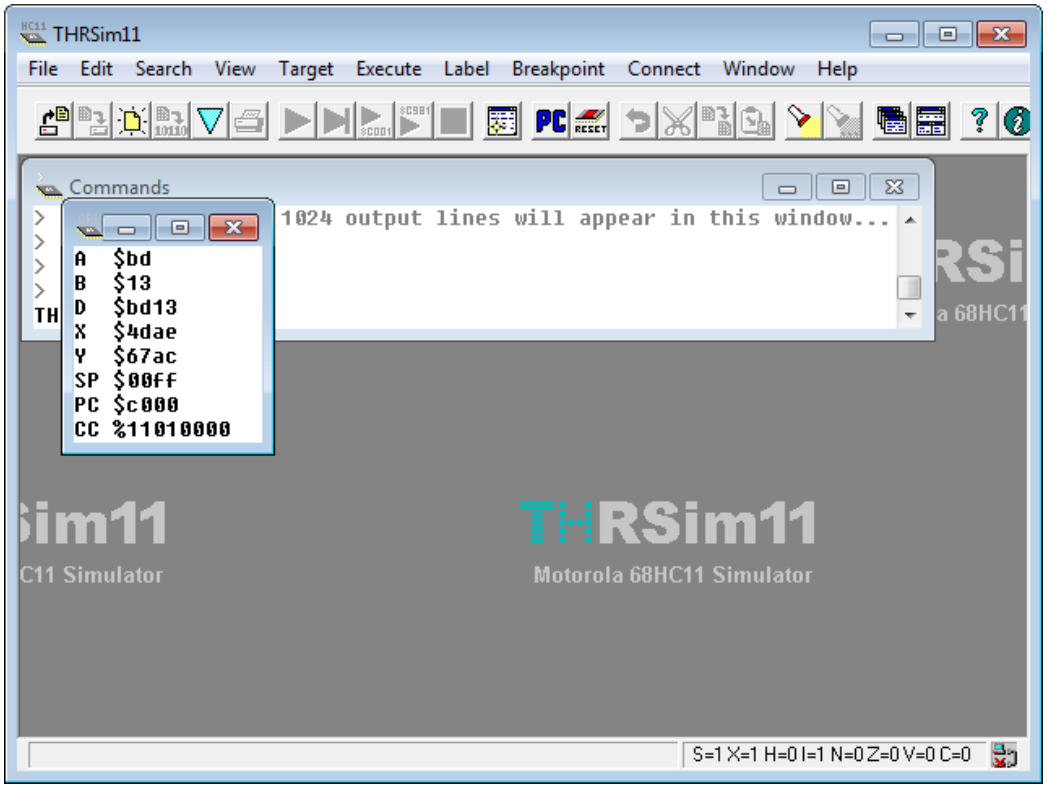
OBJECTIVE: To download & install software, and to assemble and simulate your first 68HC11 program.

SECTIONS:

A. Download and Install MiniIDE & THRSim11 (5%)
B. Generate an Arbitrary Delay using a Single Loop (35%)
C. Generate a 1 M Cycle Delay using a Nested Loop (60%)

A. DOWNLOAD AND INSTALL MINIIDE & THRSIM11

We will be using two separate programs in this course. MiniIDE is an Integrated Development Environment (IDE) and Assembler for the 68HC11 and 68HC12 microcontrollers. It will allow us to generate the machine readable code the 68HC11 will run. Download MiniIDE from the course site ("http://courses.muscedere.com/ELEC32701/miniide.msi"), and run the MSI to install the software.
Since we don't have access to any lab equipment this semester, we will test our code virtually by use of a simulator. THRSim11 is a free simulator that offers the necessary functionality we need. It has an internal assembler, however it is not as advanced as MiniIDE, so we will use MiniIDE for the majority of our code development. Download THRSim11 from the course site ("http://courses.muscedere.com/ELEC32701/thrsim11_5.30_max_setup.exe"), and run the installer. When asked where to install THRSim11, enter "C:\Program Files\THRSim11" – Make sure there is no " (x86)" in the entry or you will have issues with later labs. You can always uninstall the software and reinstall it with the proper destination in case you made this error.
When the THRSim11 installation is completed it will ask if you want to run it; you may answer yes.
We need to make a few minor configuration changes to THRSim11. Once it opens you will see the following window on the bottom right. Go to the "File", "Options", "Simulator..." menu and check "Reset 68HC11 after loading", and press OK. Go to the "File", "Options", "Memory Configuration..." menu. You will be asked to exit THRSim11; answer yes or exit to the future prompts. When the Memory Configuration Tool appears, change the "Memory Config" dropdown to "68HC11E9", and press OK. THRSim11 should restart. Unfortunately this tool does not remember our window layout once we exit it so you will often have to move windows to your liking. Since this tool was created before the Windows 7 Aero visual look, some of the windows are too narrow and



their title is not easily visible. A consequence of this is they cannot easily be moved. To see the titles and to move them, just expand the window horizontally. The window on top of the "Commands" window is one of them. Expand this window to see the title is "CPU registers" and now you can move it.
Download the "ELEC3270-Assignment5.zip" file and extract the files with your SID. In THRSim11, go to the "File", "Open..." menu. You will have to change "Files of type" to "Machine code (*.s19)" and go to the folder which contains your files. Select the "SID_5a.s19" file. A "Disassembler" window should appear showing the interpreted opcode mnemonics and operands. Go to the line "$D00A", select it with the mouse and right click on it and select "Add Breakpoint...", or simply just press F5, this will add a break point which is where the simulator will pause. Go to the "View", "Number of Clock Cycles". A new windows should appear showing a value of 2. If it is not 2, you did not change the setting properly as indicated earlier. Go back and do it again. The value of 2 here indicates that the CPU required 2 clock cycles to fetch the program starting address of $D000 from the memory address $FFFE-$FFFF ($FFFE contains $D0, and $FFFF contains $00). This is how the CPU knows where to go when it is reset. Now go to the "Execute", "Run" menu, or press F9; this will start the simulation. You will see the disassembly lines changing green to show the current instruction the simulator is running. When it gets to $D00A, it will stop. Notice the cycle count is increasing. The simulation runs much faster if the "Disassembly" windows is the selected one. Once the simulation has stopped, record the number of cycles in the "SID_5a.txt" file; only one line containing this value **and only this value** is necessary.
This code is a nest loop which simply delays the CPU an arbitrary number of clock cycles. In the next parts of the lab we will explore how we can develop code which will execute in an exact number of cycles.

B. Generate an Arbitrary Delay using a Single Loop

For this portion of the laboratory you have to modify some code which will repeatedly flash an LED on and off for an arbitrary number of cycles; the number of cycles is based on your SID. We can use the principle from part A to do this. If we look at the code, we will see something like this:

```
            ldaa  #$40    ; Set ACCA to 64 decimal        - 2 cycles
loop        nop           ; No operation                  - 2 cycles
            deca          ; Decrement ACCA                - 2 cycles
            bne   loop    ; Loop back if ACCA is not zero - 3 cycles
```

Initially ACCA is set to "#$40", this means the immediate value of $40, which is hexadecimal, or 64 decimal; this takes 2 cycles to run. Our next instruction is "no operation", it wasn't in the code in part A, but it doesn't matter as it affects nothing, but it does take 2 cycles to run. Next we decrement or subtract 1 from ACCA and store it back into ACCA; this all takes 2 cycles as well to run. Lastly we branch back to label "loop" (which is where the "nop" is) only if the ZERO flag is not set. Remember, the ZERO flag is set whenever the ALU in the 68HC11 results in a zero. So the "DECA" instruction will set the ZERO flag when it sets ACCA to zero. This instruction takes 3 cycles regardless of the outcome. Since ACCA is initially 64 and we branch only when ACCA is not zero, this loop will repeat 64 times. Therefore the number of cycles consumed is $2 + 64 \cdot (2+2+3)$ which is 450. If, for example, the clock to our CPU was 2 MHz, each cycle would be 500 ns (1/(2 MHz) is 500 ns). Therefore this code generates EXACTLY a 225 μs delay in only 6 bytes of code. There are some limitations however as ACCA is only 8-bits and can never exceed 255. With this in mind, the largest number of cycles this could ever consume is $2 + 255 \cdot (2+2+3)$ which is 1787 – not a very large number. We could always add more NOP instructions, but that would result in larger code. An alternative is to use the INDX register, which is 16-bits and therefore can hold larger values (up to 65535) which could generate more than 500K cycle delays depending on the number of NOPs inside the loop. We could also use other instructions (multiply for example) which consume more time, but be sure that their result doesn't affect the loop.
In THRSim11, go to the "File", "Open..." menu. You will have to change "Files of type" to "THRAss11 source (*.asm)" and go to the folder which contains your files. Select the "SID_5b.asm" file. A new window should appear. The first comment line indicates to you how many cycles you need to delay it by. **All of the instruction delays must be accounted for, including the ones the toggle the LED and jump back to the beginning.**
Each instruction includes a comment detailing how many cycles it requires to run. Later in the code you will see "LDX #@"; here you change the "@" to the calculated value you need. You may also need to add in some extra

delay cycles. We can use any number of instructions which can waste time. Two in particular that do nothing are "NOP" (consumes 2 cycles) and "BRN" which is "branch never" (consumes 3 cycles). BRN requires an operand (address) so we can use "*" which is the current location in memory of the current instruction. This is ideal as branches need locations that are close to the current instruction. You may need a combination of these to achieve your goal. There is no instruction with a delay of 1 cycle, so if that is the situation you find yourself in than you will need to lower your initial value of INDX and add in more instructions later to compensate. When your code is ready, you can go to "File", "Assemble" or just press CTRL-A. If there are any errors, the system will let you know. Otherwise a new window will appear, "SID_5b.lst". An "LST" file is a human readable file which shows the results of the assembly (values of symbols) as well as the sequential instruction addresses, machine code bytes, and mnemonic opcodes and operands. You should place a break point on the first line so we can see how many cycles it will take to run until it loops back to that same address. Check the number of cycles at this point and record it. Now press F9 to start the simulation and when it stops, it will be on the same line. The difference between the current number of cycles and the one you recorded earlier is the cycle count between toggling the LED. If you press F9, the difference between the new value and previous one should be the same. Our code is therefore consistent in its operation. Microcontrollers are very reliable in their timing. Running this code will never takes any less or more time; it will always be the same. Depending on your application, timing can be very important.

C. Generate a 1 M Cycle Delay using a Nested Loop

To achieve a delay of 1 million cycles assuming you are using the same code in part B, would require INDX to be initially set to about 125,000; this value is too large for 16-bits. To practically solve this you need to use a nested loop, which is a loop in a loop. Again we have to watch out for the limitations here on the registers we choose. ACCA & ACCB are both 8-bits (0 to 255), and INDX & INDY are 16-bits (0 to 65535). However, remember that INDY offers us the same operations as INDX but with the penalty of an extra code byte and an extra operating cycle. It is up to which registers you want to use and how. Since we are only designing a delay loop; all that is important is that time it takes to run. We therefore have a single outcome controlled by 2 registers. This is similar to the problem, "what two positive numbers add up to 1 thousand"; there are many possible combinations. **To allow for some variation in our results, you will be required to set the initial value of one of your control registers (either INDX or INDY) to some specific value which is provided in the first comments of the "SID_5c.asm" file.** It is up to you to figure out what the other control registers should be set to and which register to use (ACCA, ACCB, INDX, or INDY). To further constrain your code development, you will need to **try to minimize the size of your code** to the fewest bytes possible; so please don't use 20 NOPs to get a 40 cycle delay. You don't even need to use NOPs if you like. **In order to achieve full marks, you must document (as comments in your code) how you obtained exactly the 1 million cycle second delay.** *Your grade will be determined by the deviation from 1 million cycles.*
Your code should operate like the following pseudo code:

```
start:  toggle LED
        set Y = 50
loop1:  set X = 500
loop2:  decrement X; loop back to loop2 when X is not zero
        decrement Y; loop back to loop1 when Y is not zero
        set a = 5 to deal with delays that cannot be accommodated with X and Y
loop3:  decrement A; loop back to loop3 when A is not zero
        any other small delays which cannot be accommodated with A
        go back to start
```

This code loops $50 \cdot 500 + 5$ times, but the number of cycles depends on each instruction and the number of times it runs.

When you have completed all parts of the lab, ZIP the "SID_5a.txt", "SID_5b.asm" and "SID_5c.asm" into a single ZIP file (SID_5.zip) and upload them to Blackboard.