CEG 3136 – Computer Architecture II
Lab1 – Introduction to Microprocessor Programming

Objectives:

1) Become familiar with the Debug-12 Monitor.
2) Introduce assembler programming.
   a) Assemble and run assembler code
   b) Learn basic principles of debugging.
   c) Apply different addressing modes.
3) Design and Implement a simple software module.

PreLab:

- Review (a) the lab manual, (b) the Alarm System Simulation Software Design and code, and (c) this document.
- Prepare a draft of the lab report and develop the Delay Module.
- Marks are assigned for the pre-lab.

Equipment Used:

- Windows PC
- Dragon 12 Plus Trainer

Components Used:

- None

Observation:

As 3rd year students, you should be showing up to the labs prepared. This means, at the least, that you have read the lab manual and lab material, compiled your software, and drafted your lab report before you enter SITE 2061. Many headaches will be avoided if you see the entire contents of the labs so that you know where the exercises are leading. It will be very difficult to complete the lab during the allotted time if you arrive unprepared. Read the lab carefully - there is no point in doing something that isn't required. In particular, you should prepare the following:

1. Note the changes you are to make to the code for completing Part 1.
2. For Part 2, try to identify the bugs by reviewing what the code is doing.
3. Develop and assemble the Delay Module for Part 3.
4. Draft your lab report. You can create tables that can be filled out during the lab for Parts 1 and 2. Develop your design for Part 3 and record it in your draft lab report. You are required to show your draft lab report (and assembled code) to the TA at the start of the lab session.

Part 1 – Using the D-Bug12 Monitor

a. Using the D-Bug12 Monitor, assemble the following program (with the D-Bug12 ASM command) at address $2000 (Note: type a period <.> to get out of the ASM mode). As you are entering the program, complete item b below. After assembling the program and verifying the code, run the program by typing **G 2000**. Note that the PSHA instruction will be located at address $200F. This address is used by the BNE branching instruction to create a loop.

Address	**Mnemonic**	**Opcode**	**Comments**
$2000	LDD	#$3A	Load ASCII code for ":"
	LDX	$EE86	Load the vector for *putchar* routine
	JSR	0,X	Print what's in B on terminal
	LDX	$EE84	Load the vector for *getchar* routine
	JSR	0,X	Get a new character in B
	LDAA	#3	Initialize loop counter
$200F	PSHA		Save the counter on stack
	PSHB		Save contents of B on stack
	LDD	#$20	Load B with a space
	LDX	$EE86	Load the vector for *putchar* routine
	JSR	0,X	Print it on terminal
	PULB		Get original character
	LDX	$EE86	Load the vector for *putchar* routine
	JSR	0,X	Print it on terminal
	PULA		Retrieve the counter
	DECA		Decrement loop counter
	BNE	$200F	If counter <> 0, repeat
	SWI		Return to the monitor

b. As you are providing the instructions to the assembler, record in the table below, the machine code in the "Contents" column with the given address in the Address column. See the first entry in the table as an example. As you are entering the instructions, try to predict the machine code before the assembling it. In your report break down the machine instruction into the opcode and operand for each of the following instructions. Explain in your report the addressing mode being used by each of the three following instructions.

LDX $EE86
JSR 0,X
BNE $200F

Address	**Content**	**Instruction**	**Description**
2000	CC003A	LDD #$3A	Load ASCII code for ":"
		LDX $EE86	Load the vector for *putchar* routine
		JSR 0,X	Print what's in B on terminal
		LDX $EE84	Load the vector for *getchar* routine
		JSR 0,X	Get a new character in B
		LDAA #3	Initialize loop counter
		PSHA	Save the counter on stack
		PSHB	Save contents of B on stack
		LDD #$20	Load B with a space
		LDX $EE86	Load the vector for *putchar* routine
		JSR 0,X	Print it on terminal
		PULB	Get original character
		LDX $EE86	Load the vector for *putchar* routine
		JSR 0,X	Print it on terminal
		PULA	Retrieve the counter
		DECA	Decrement loop counter
		BNE $200F	If counter <> 0, repeat
		SWI	Return to the monitor

c. In your report, explain what the program does? Use C pseudo-code to illustrate.
d. Change the ":" prompt to a ">" and change the separator character from a space " " to a comma ";".
e. Change the loop counter so it prints exactly 15 of them. Demonstrate to the TA your new program.
f. In your report provide the listing of the modified code using a similar table as in item b.

Part 2 – Assembler

In Part 1, the method of entering code into the HCS12 with the D-Bug12 simple assembler is a pretty poor one considering other available tools. The experience is valuable to gain insight on the process of assembling a program. Usually, more powerful assemblers like MiniIDE are used. You should familiarize yourself with MiniIDE as it will be used for the rest of this lab and for the lab 2.

a. In this part, you will be assembling a program that has been coded (see the file "alarmSimul.asm" and other assembler files). This program will let you simulate the alarm system using the Dragon-12 card as described in the design document *CEG3131AlarmSimulDesign.pdf*. This document also provides a description of the software design.
b. The following equates in the source file specifies the locations for the code, data and stack.

```
; Memory map equates
STACK:    EQU  $2000
CODE:     EQU  $0400
VARIABLES:    EQU       $2500
```

The program has been designed to run in EEPROM (programmable read-only memory) found at address $0400. Variables are stored in RAM at address $2500 (it will be initialised when the program is started). Constant data, such as message strings, that are not modified are found at the end of the code. The stack pointer is initialised to $2000 and thus the stack starts at address $1FFF. These "equates" are defined in the file *sections.inc* which also defines the sections *globalVar*, *code_section,* and *globConstant*. Sections are discussed in Module 3; you may also consult documentation on the Assembler directives SECTION and SWITCH.

c. Start MiniIDE and load the program provided (file *alarmSimul.asm*). Assemble the program which creates an S19 file (the file *alarmSimul.S19*). Download the S19 file into the Dragon-12 board by following the instructions in Appendix A.
d. The assembler also creates the file *alarmSimul.lst*. This file shows the assembled machine code that corresponds to the assembler source code. Use it to follow the operation of the code in the Dragon-12 board.
e. Run the Debug-12 command "G 0400"; this starts the alarm simulation program. The program has two problems:
    - It accepts all codes. For example you can arm and disarm the system with a code of your choice. A bug exists in the subroutine `isCodeValid` (in the Armed module).
    - Delays have not been implemented in the software (that is the Delay module does not exist). You will deal with this second problem in Part 3 of the lab.

f. Before dealing with the bug in `isCodeValid`, let's learn how to trace the execution of the software. Hit the reset button to bring you back to the Debug-12 monitor. By default, the D-Bug12 monitor uses hardware breakpoints. Only 2 breakpoints can be set in this mode.

g. Examine the listing of the assembled code in the file alarmSimul.lst. The file provides the machine code for the instructions of the source code and the addresses where these instructions are located. Use this file to follow the execution of the program. Trace the first few instructions as follows:
   - Set the PC to point to the first instruction of the program located at address $0400 by simply entering PC 0400. Record the contents of the CPU registers in the table below.
   - Now step through one instruction by typing "T" (this will be the LDS #STACK instruction that initialises the stack pointer). Record the values found in the CPU registers in the table below. Don't forget to follow the program execution in the LST file.
   - The next instruction is a BSR that jumps to a subroutine. Type "T" to execute this instruction. The PC will be changed to point to the first instruction in this subroutine (at address *inithw*). Record the values found in the CPU registers.
   - Trace the next two instructions (do not forget to record the register values after the execution of each instruction). Examine the effect of the various instructions on the contents of the CPU registers. If you trace a third instruction (BCLR Clksel,x,%10000000), the system will hang. This instruction updates a register that affects the system clock which changes the operation of the serial communications port. To recover from this problem, hit the reset button.

h. Tracing is fine, but impractical to get to points in the program beyond many loops and many subroutines. The D-Bug12 monitor allows setting breakpoints that permits the execution of a program until the breakpoint (an address) is encountered. Try the following. Set a breakpoint at the address of the "movw #0,alarmCode" instruction (in the *inithw* subroutine) with the command "BR <ADDR>", where <ADDR> is the address where the instruction is located (example $0450). Confirm this address by examining the alarmSimul.lst file. Now start the execution of the program with "G $0400". The CPU executes the program until the given address is seen. The monitor then takes control of the CPU and provides the contents of the CPU registers.
   - Trace two instructions (these should be MOVW instructions). What registers change, if any? What addressing mode was used in these instructions? Verify this effect with the D-Bug12 command MD to examine the contents of RAM. Record your observations in your report.
   - Trace the next instruction that is "rts". Explain in your report the effect of this instruction. Always record values found in the registers and notice how their contents change.

i. It's time to debug the problem with `isCodeValid` (Armed Module). Set a breakpoint to the start of the `isCodeValid` subroutine. You should have taken the time to understand how this subroutine works (examine the equivalent C function). If you have an idea of the problem, then trace the subroutine to verify your hypothesis. Note that at anytime you can change the values of registers and memory before continuing your execution. Once you have verified your hypotheses or discovered the bug, change the original code and download the newly

assembled code. Test the updated software and demonstrate to your TA. In your report, explain how you discovered the bug and corrected it.

D						
A	**B**	**X**	**Y**	**SP**	**PC**	**CCR**

Part 3 – Developping the Delay Module

In this third part of the lab you will develop the Delay module, which consists of two subroutines:

a. `pollDelay`: This subroutine delays for 1 milli-second. To create longer delays, it uses a counter that is decremented each time it is called. When the counter reaches 0, it returns a TRUE value (i.e. 1); otherwise it returns a FALSE value (i.e. 0). The subroutine is designed to create long delays (for example 10 seconds) with the counter while allowing the calling routine to get control every millisecond.
b. `setdelay`: This subroutine is used to set the counter before `pollDelay` is called and thus has one parameter (which is passed in the D register). For example, if 100000 is passed to `setdelay`, then the counter used by `pollDelay` is initialised to this value. Thus after 10000 calls to `pollDelay` (about 10 seconds), it returns TRUE.

The above approach is used since the calling routines in the *Armed* module, must be able to check certain conditions during delays. For example, when the alarm system has been armed, during the 10 second delay, an alarm code can be entered to disarm the system. Thus input from the user must be checked (using `pollgetchar`) during this delay. See the `enableAlarm` subroutine (and C function) for more details.

Before coming to the lab session, it is essential that you design and code the Delay module.

a. To design the module, determine the algorithm to create a 1 ms delay. The basic idea is to define how long instructions take and to create a loop to execute enough instructions enough times to cause the 1 ms delay. DO NOT use the timer - we shall do this in lab 4. Instructions such as NOP and BNE are useful for this purpose. You will need to determine the number of CPU cycles an instruction takes and the length of these cycles. See the CPU12 reference guide for more details. You may wish to consider creating a new subroutine, e.g. delay1ms, which creates the 1 ms delay. The polldelay subroutine will simply call delay1ms.
b. Record your design of the module by collecting all information required and describing your approach in a draft of your report. Create the C functions as a means of detailing your algorithms for both `pollDelay` and `setdelay`. See the design document *CEG3136AlarmSimulDesign.pdf* for examples.
c. After you have completed your design, complete the *delay.asm* file using MiniIDE and translate the C functions into assembler subroutines. Do assemble your program BEFORE coming to the lab session.
d. You will note that in the *armed.asm* file, calls are made to `pollDelay` and `setdelay`. During the lab session, after completing Part 2 of the lab, re-assemble your program and debug your software to verify that the following delays are properly implemented:
    - when the system is armed (between the "Arming" and "Armed" messages).
    - when the system is disarmed when the front door is opened (i.e. the letter 'a' is typed); that is between the "Disarmed" message and the main menu.
    - between displaying the "@" characters when the alarm is sounded.
e. Demonstrate that delays are working in all three cases to your TA.

Some notes:

Note that assemblers are not identical, so although 99% of your core code will work, switching between them might require some code rewriting (this is why we recommend the use of the MiniIDE assembler).

- The assemblers can be picky, for instance, any instruction needs to have at least 1 space from the leftmost of a new line, while labels (and EQU equates) must be left justified. And this might not be true for some other assemblers. The point we're trying to get across is that some errors you can't figure out might be due to the formatting.
- Additional strange formatting: It is possible that one of the assemblers supports both "JSR 0, X" and "JSR 0,X" while the other one will only work with "JSR 0,X". The difference is the space after the ",".

Finally, additional info on the LOAD command from the DBug-12 Monitor. This is important for all compiled code. Usually, you should write an ORG statement in your .asm code file so that the assembler can know where in RAM your code will exist. Our most basic one is the ORG $2000 statement which we see on line 1. However the ORG statement is optional. You could write your code without an ORG statement and simply use LOAD $2000 to load the info into memory starting at address $2000. Using the ORG statement is recommended and then just LOAD. Care must be taken when arguments for both statements or neither statement:

1. You put an ORG $2000 and use LOAD $2000
2. You don't put an ORG statement, and you only use LOAD to load the program

In both those cases, you may get a message akin to "Can't write to target memory" and some garbage will appear on your screen. What just happened? In Case 1, your program believes it start at address $2000. However the LOAD command says "don't start writing at $0000, but at $2000". Thus the software tries to insert itself into the RAM at $2000+$2000 = $4000 which is reserved memory and not allowed. In the case of the Dragon-12, this can have drastic consequences since the D-Bug12 resides at address $4000 and writing anything there destroys D-Bug12.

In the second case, without an ORG statement the program believes it starts at address $0000, and without telling LOAD otherwise that's where it will try to go. However this is also reserved memory, and this causes an error.

What-to-do LIST

PreLab related part:

a. Take the time to review this document and understand the work you need to do before and in the lab.
b. Take the time to review the *alarmSimul* assembler code and the design document. In particular, try to understand how the program starts, and how the initialisation subroutine *inithw* works. Also look at the *isCodeValid* subroutine. This subroutine contains a bug(s) which you need to correct.
c. Do design (include the design in the draft lab report), code and assemble the Delay Module.
d. Prepare a draft of the lab report and show it to the TA at the start of the lab session. Assemble the *alarmSimul* software (including the Delay Module).

In the lab:

a. Follow the instructions in Parts 1, 2 and 3. Careful to record all data required for the report as you follow these instructions. For Part 2 you will need to consult the Annex on how to download the machine code into EEPROM in order to debug the `isCodeValid` subroutine. For Part 3, you will be debugging the Delay module. You will make 3 demonstrations to the TA. The first is the modified routine in Part 1, the second is the debugged *isCodeValid* subroutine, and the third is the implementation of delays in the software.

In your report:

a. For Part 1, provide the two tables requested; the description of the addressing modes for the given instructions and the C function that shows the logic of the program.
b. For Part 2, provide the table requested and answer all questions posed in (g)/(h). Also report on your work discovering and correcting the bug as requested in (i).
c. Provide design documentation for the Delay module. Note that you will submit *delay.asm* with your report.

Appendix A – Loading the EEPROM

Recall the memory map for the 68HCS12 microcontroller:

0000-03ff - I/O register
0400-0fff - On-chip EEPROM
1000-3bff - On chip RAM (for user)
3c00-3fff - On chip RAM (for D-Bug12)
4000-ffff - D-Bug12 program.

Software is loaded into the micro-controller by downloading an S19 file. The MiniIDE assembler produces S19 files and can be used to download the file. D-Bug12 provides a utility (LOAD command) to receive the file and store the program into memory (either RAM or EEPROM). Use to following steps to load the alarmSimul program into EEPROM:

1. Generate the file *AlarmSimul.s19* using the assembler.
2. Go to the assembler IDE Terminal window and type the D-Bug12 command LOAD.
3. Now download the *AlarmSimul.s19* file using the IDE download function (in MiniIDE, click on the Download icon or select the "Download File…" item in the Terminal menu).
4. A number of asterisks will appear in the Terminal window. When the download is complete, the ">" will appear.

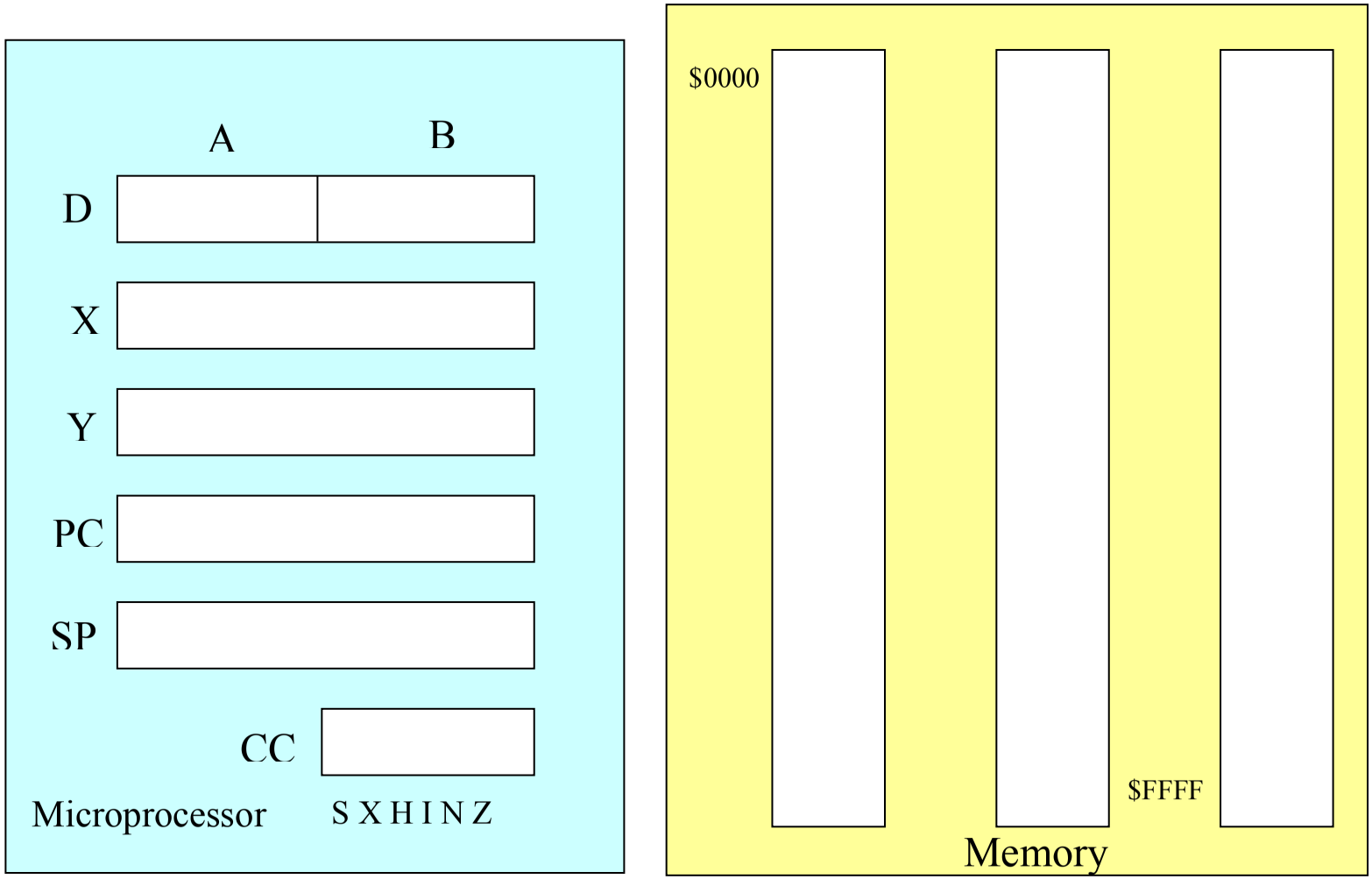
Type "G 0400" to start the program. You will be able to turn off the power to the trainer card without loosing your program.

**********IMPORTANT**********

When you have completed your Lab, run the command BULK, This will erase all of the EEPROM.

****

Appendix B – Programming Model



Appendix B – ASCII Codes

ASCII is the American Standard Code for Information Interchange. It is a 7-bit code. Many 8-bit codes (such as ISO 8859-1, the Linux default character set) contain ASCII as their lower half. The international counterpart of ASCII is known as ISO 646. The following table contains the 128 ASCII characters.

ASCII TABLE

Oct	Dec	Hex	Char	Oct	Dec	Hex	Char	Oct	Dec	Hex	Char	Oct	Dec	Hex	Char
0	0	0	NUL	40	32	20	SPACE	100	64	40	@	140	96	60	`
1	1	1	SOH	41	33	21	!	101	65	41	A	141	97	61	a
2	2	2	STX	42	34	22	"	102	66	42	B	142	98	62	b
3	3	3	ETX	43	35	23	#	103	67	43	C	143	99	63	c
4	4	4	EOT	44	36	24	$	104	68	44	D	144	100	64	d
5	5	5	ENQ	45	37	25	%	105	69	45	E	145	101	65	e
6	6	6	ACK	46	38	26	&	106	70	46	F	146	102	66	f
7	7	7	BEL	47	39	27	'	107	71	47	G	147	103	67	g
10	8	8	BS	50	40	28	(	110	72	48	H	150	104	68	h
11	9	9	HT	51	41	29	)	111	73	49	I	151	105	69	i
12	10	0A	LF	52	42	2A	*	112	74	4A	J	152	106	6A	j
13	11	0B	VT	53	43	2B	+	113	75	4B	K	153	107	6B	k
14	12	0C	FF	54	44	2C	,	114	76	4C	L	154	108	6C	l
15	13	0D	CR	55	45	2D	-	115	77	4D	M	155	109	6D	m
16	14	0E	SO	56	46	2E	.	116	78	4E	N	156	110	6E	n
17	15	0F	SI	57	47	2F	/	117	79	4F	O	157	111	6F	o
20	16	10	DLE	60	48	30	0	120	80	50	P	160	112	70	p
21	17	11	DC1	61	49	31	1	121	81	51	Q	161	113	71	q
22	18	12	DC2	62	50	32	2	122	82	52	R	162	114	72	r
23	19	13	DC3	63	51	33	3	123	83	53	S	163	115	73	s
24	20	14	DC4	64	52	34	4	124	84	54	T	164	116	74	t
25	21	15	NAK	65	53	35	5	125	85	55	U	165	117	75	u
26	22	16	SYN	66	54	36	6	126	86	56	V	166	118	76	v
27	23	17	ETB	67	55	37	7	127	87	57	W	167	119	77	w
30	24	18	CAN	70	56	38	8	130	88	58	X	170	120	78	x
31	25	19	EM	71	57	39	9	131	89	59	Y	171	121	79	y
32	26	1A	SUB	72	58	3A	:	132	90	5A	Z	172	122	7A	z
33	27	1B	ESC	73	59	3B	;	133	91	5B	[	173	123	7B	{
34	28	1C	FS	74	60	3C	<	134	92	5C	\	174	124	7C	|
35	29	1D	GS	75	61	3D	=	135	93	5D	]	175	125	7D	}
36	30	1E	RS	76	62	3E	>	136	94	5E	^	176	126	7E	~
37	31	1F	US	77	63	3F	?	137	95	5F	_	177	127	7F	DEL