Lecture test in 2 weeks

Write a short program in Assembler doing ….

You will be using your computer with all the programs you want as well as any notes or texts. You will submit your program as a text file – just email it to me when the time is up (no later). 10% penalty per minute!

Its all on the honor system, you will be expected not to communicate with anybody. Identical programs will be marked at 0.

Today:

- Announcements
- General info finding strategy
- Microcontroller programming in assembler – final parts
- Activities 1 and 2
- Intro to programming in C

Announcements:

Please submit on Canvas lab notes and annotated programs as a pdf file before the beginning of lab 3

Labs Marking scheme:

Completeness and quality of procedure/circuits/code etc. used to perform all the completed activities	5
General Report Structure: objectives defined, clear enough statements of what is being done and why, what the results were. General clarity of the notes	2
Above and beyond. Evidence of exploration above and beyond the specific tasks requested in the manual.	1

Labs 1&2 will be worth 4 points, labs 3&4 and 5&6 8 points each pair.

Plese make sure that you upload all the software needed for programing in C

Addressing Modes

- Immediate mode
- Syntax: #N
- The word following the instruction contains the immediate constant N.
- Examples:

```
mov.b #01000001b, & P1DIR
mov.b #65, & P1DIR
mov.b #0x41, & P1DIR
```

- Source address can use this format (destination cannot)

Addressing Modes

- Absolute mode
- Syntax: &ADDR
- The word following the instruction contains the absolute address.
- Example:

  mov.b #11110111b, &P1DIR

  The destination address uses this format

Addressing Modes

- Register mode
- Syntax Rn
- Register contents are operand
- Rn can be PC, SP, SR, CG2, R4 ...R16
- Example

  bis.w #CPUOFF,SR

  Destination address is the CPU status register SR

Addressing Modes

As/Ad	Addressing Mode	Syntax	Description
00/0	Register mode	Rn	Register direct. Register contents are operand
01/1	Indexed mode	X(Rn)	Indexed mode. The operand is in memory at address Rn+X.
01/1	Symbolic mode	ADDR	Symbolic mode. (PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.
01/1	Absolute mode	&ADDR	Absolute Mode. The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.
10/−	Indirect register mode	@Rn	Register indirect. Rn is used as a pointer to the operand. (same as 0(Rn))
11/−	Indirect autoincrement	@Rn+	Register autoincrment. Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.
11/−	Immediate mode	#N	Immediate Mode. The word following the instruction contains the immediate constant N. Indirect autoincrementmode @PC+ is used.

General info finding strategy

Documentation: read the manual!
 - not always so easy. Which manual?

There are two major documents relevant for the microcontroller:
 - Family reference guide (slau144) [eg cpu instructions]
 - chip data sheet (slas735) [eg what pin can do what]
Both contain much more info than we need!

Use:
-Table of contents
-Keyword searching

Also:
- Course lab manual – general instruction, what tasks are required
- OS specific set-up/user guide – how to set up computers, what command to type to compile/assemble programs and load on to Launchpad board.

Activity 1 due on Canvas next week at the beginning of the lecture

- Write commands which will configure all pins of port 1 as inputs, and move the value from port 1 to register R7. Finally, write the binary number which will be in the 16 bit register R7 after these operations assuming that all 8 pins of port 1 were connected to 3V.
- Port P1 registers:

- P1REN ; Port P1 up/down resistor enable
- P1SEL ; Port P1 selection
- P1DIR ; Port P1 direction
- P1OUT ; Port P1 output
- P1IN ; Port P1 input

Activity 2 (due on Canvas next week at the beginning of the lecture)

What are the values of R7 and the
Z, N, and C bits after the following commands
(assuming they were all 0 initially)

	Z = 0	N = 0	C = 0	R7 = 0
mov.w #0xF0F0, R7	Z = ?	N = ?	C = ?	R7 = ?
add.w #0xF000, R7	Z = ?	N = ?	C = ?	R7 = ?
sub.w #0xE0F0, R7	Z = ?	N = ?	C = ?	R7 = ?

Notice: before completing activity 2

You have to look at the detailed description of each command to find out how it affects the Status Register's bits. The descriptions are found in

Microprocessor family slau144j MSP430x2xx

Stack and Stack Pointer

- A part of the RAM usually starting at the top (highest address 0x400) of the available RAM used to store values which we will use later (PUSH, PUSH.b and POP, POP.b commands) or for storing values of registers during subroutines and interrupts.

- The size of the RAM limits the number of nested subroutines and interrupts – we can not allow the area where we keep the variables to overlap with the stack. This is a common cause of “crashing” the computer or microprocessor.
- Stack pointer (SP register) indicates the lowest occupied position in the stack

```
#include "msp430g2553.inc"
org 0xf800
RESET:
        mov.w #0x0400, sp
        mov.w #WDTPW|WDTHOLD,&WDTCTL
        mov.b #11110111b, &P1DIR
        mov.b #01000001b, &P1OUT
        mov.b #00001000b, &P1IE
        mov.w #0x0041, R7
        mov.b R7, &P1OUT
        EINT
        bis.w #CPUOFF,SR
PUSH:
        xor.w #0000000001000001b, R7
        mov.b R7,&P1OUT
        bic.b #00001000b, &P1IFG
        reti
org 0xffe4
dw PUSH
org 0xfffe
dw RESET
```

Interrupts

- Are triggered by an external event (eg an input going low, a timer overflowing, number of counts exceeding some preset value etc.)

- PC (program counter) and SR (status register) are s regular flow can resume when the interrupt finishes

Interrupts

- When interrupt occurs the current microprocessor’s activity stops and the interrupt service routine (ISR) is started
- The address of the ISR has to be stored in the specific location in the memory. This address is called an interrupt vector. They are located in the flash memory area 0xFFE0 to 0xFFFF. The addresses of the interrupt vectors are listed in the msp430g2553 manual.

Interrupts

- ```
  #include "msp430g2x31.inc"
  CPUOFF equ 0x0010
  org 0xf800
  RESET:
          mov.w #0x280, sp
          mov.w #WDTPW|WDTHOLD,&WDTCTL
          mov.b #11110111b, &P1DIR
          mov.b #01000001b, &P1OUT
          mov.b #00001000b, &P1IE
          mov.w #0x0041, R7
          mov.b R7, &P1OUT
          EINT
          bis.w #CPUOFF,SR
  PUSH:
          xor.w #0000000001000001b, R7
          mov.b R7,&P1OUT
          bic.b #00001000b, &P1IFG
          reti
  org 0xffe4
  dw PUSH
  org 0xfffe
  dw RESET
  ```
  ```

Interrupts

- The event setting an interrupt is in fact setting a bit in a specific register. This bit is called an interrupt flag. For example a PORT1.3 interrupt, sets bit 3 in the P1IFG register at the address 0x026. The ISR must clear this flag!

- Each interrupt service routine has to end with RETI command.

```
#include "msp430g2x31.inc"
CPUOFF equ 0x0010
org 0xf800
RESET:
        mov.w #0x280, sp
        mov.w #WDTPW|WDTHOLD,&WDTCTL
        mov.b #11110111b, &P1DIR
        mov.b #01000001b, &P1OUT
        mov.b #00001000b, &P1IE
        mov.w #0x0041, R7
        mov.b R7, &P1OUT
        EINT
        bis.w #CPUOFF,SR
PUSH:
        xor.w #0000000001000001b, R7
        mov.b R7,&P1OUT
        bic.b #00001000b, &P1IFG
        reti
org 0xffe4
dw PUSH
org 0xfffe
dw RESET
```

Interrupts

- When the interrupt occurs:
  - status register and program counter are pushed onto the stack.
  - the program counter is loaded from the interrupt vector
  - the ISR (pointed to by the interrupt vector) executes
  - the ISR must clear the interrupt flag [some clear themselves]
  - reti pops the status register and program counter off the stack so the program can continue

  Save/restore registers? In assembly you need to worry about this yourself. In a C program, this is handled for you.

```
#include "msp430g2553.inc"
org 0xf800
RESET:
        mov.w #0x400, sp
        mov.w #WDTPW|WDTHOLD,&WDTCTL
        mov.b #11110111b, &P1DIR
        mov.b #01000001b, &P1OUT
        mov.b #00001000b, &P1IE
        mov.w #0x0041, R7
        mov.b R7, &P1OUT
        EINT
        bis.w #CPUOFF,SR
PUSH:
        xor.w #0000000001000001b, R7
        mov.b R7,&P1OUT
        bic.b #00001000b, &P1IFG
        reti
org 0xffe4
dw PUSH
org 0xfffe
dw RESET
```

Enable pin P1.3 to produce interrupts

Global enable interrupts (DINT will disable)

Calling Subroutines

```
        Move.b #00001111,
…………………………….
        mov.w #60000, r9
        CALL  #Delay
..............................
Delay:
        dec r9
        jnz Delay
        RET
```

3.4.6.32 MOV

MOV[.W] Move source to destination

MOV.B Move source to destination

Syntax

```
MOV src,dst  or  MOV.W src,dst
MOV.B src,dst
```

Operation src → dst

Description The source operand is moved to the destination.

The source operand is not affected. The previous contents of the destination are lost.

Status Bits Status bits are not affected.

Mode Bits OSCOFF, CPUOFF,and GIE are not affected.

Example The contents of table EDE (word data) are copied to table TOM. The length of the tables must be 020h locations.

```
        MOV   #EDE,R10              ; Prepare pointer
        MOV   #020h,R9              ; Prepare counter
Loop    MOV   @R10+,TOM-EDE-2(R10)  ; Use pointer in R10 for both tables
        DEC   R9                    ; Decrement counter
        JNZ   Loop                  ; Counter not 0, continue copying
        ......                      ; Copying completed
        ......
        ......
```

Example The contents of table EDE (byte data) are copied to table TOM. The length of the tables should be 020h locations

```
        MOV     #EDE,R10              ; Prepare pointer
        MOV     #020h,R9              ; Prepare counter
Loop    MOV.B   @R10+,TOM-EDE-1(R10)  ; Use pointer in R10 for
                                      ; both tables
        DEC   R9                      ; Decrement counter
        JNZ   Loop                    ; Counter not 0, continue
                                      ; copying
        ......                        ; Copying completed
        ......
        ......
```

Notes

- No arithmetic on port registers
- The r1-r4 registers are reserved (program counter and so on). Have a look at the CPU registers
- Interrupt vectors are port not pin specific

Why program in Assembly?

Full control of every detail of program flow and memory organization

Speed!

Smallest, most compact programs

Why program in Assembly?

Full control of every detail of program flow and memory organization

Speed!

Smallest, most compact programs

Why not?

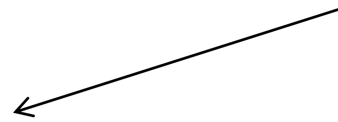
Need to take care of every detail

Hard to debug

Tedious

Instruction set is CPU specific

Programming in C



include header file, similar to .include in assembly. Defines symbols like P1OUT

```
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

Programming in C

```c
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

Declare a global variable.
Global variables can be used anywhere in the program. The volatile keyword tells the compiler that the variable might change unexpectedly (eg in an interrupt) so it should store the variable in RAM, not just in a register.

A variable declared within a set of braces can only be accessed within those braces.

Programming in C

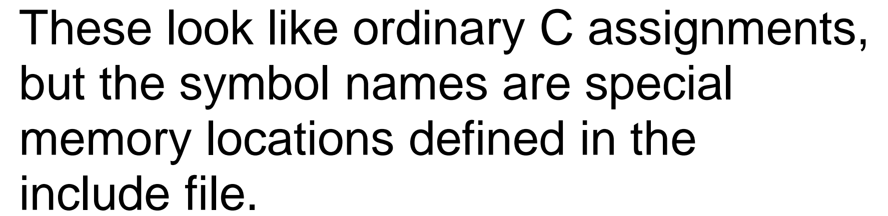
Every C program must have a routine called main. The compiler generates the code necessary for the address of the main routine to go into the reset vector. The (void) says that no parameters are passed to the function.

```
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

Programming in C




```c
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

Programming in C

```
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

Turn on the bits to ensure that P1.6 and P1.0 are outputs.
|= is an operator . This statement Is equivalent to:
P1DIR = P1DIR | 0x41;
where | is the bitwise OR operation.

Programming in C

```
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

for(initialization ; condition ; increment expression)

Programming in C

```
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

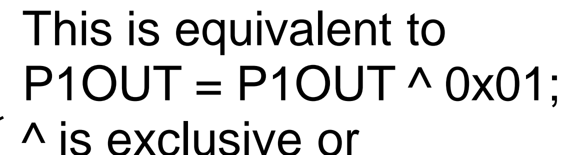
Test if i is 0. Note that equality is tested with ==
A single = is an assignment.

Programming in C

```c
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```



Programming in C

```
#include <msp430.h>

volatile unsigned int i=0;
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;
        P1DIR |= 0x41;

        for(;;){
                for ( i = 0 ; i < 20000 ; i++ ){
                        if ( i  ==  0 )
                                P1OUT ^= 0x01;
                        if ( i  == 6000)
                                P1OUT ^= 0x40;
                }
        }
}
```

```
if ( i == 0 ){
        a = 2;
        b = 3;
}
```

braces

vs

```
if ( i  == 0 )
        a = 2;
        b = 3;
```

```
if (i == 0) a = 2;
b = 3;
```

Never do this:

```
if ( i  ==  0 );
a = 2;
```

tabbing is helpful for readability. Most useful editors will help tabbing

The compiler itself ignores whitespace – it's just for readability

```
if ( i == 0 ){
        a = 2;
        b = 3;
}
```

braces

vs

```
if ( i  == 0 )
        a = 2;
        b = 3;
```

executed even if
i != 0

```
if (i == 0) a = 2;
b = 3;
```

tabbing is helpful
for readability.
Many useful
editors will help
tabbing

Never do this:

```
if ( i  ==  0 );
a = 2;
```

The compiler itself ignores whitespace – it's just for readability

Programming in C

Operators:

=, +, -, *, /

% - modulus

& - bitwise AND

| - bitwise OR

^ - bitwise XOR

~ - bitwise NOT

<< - bitshift left

>> - bitshift right

Comparison:

==, <, >, !=   if (i < 3), if (i != 3)

&& - logical AND   if (i == 1 && j == 2)

|| - logical OR   if (i == 1 || j == 2)

! logical NOT