Chapter 3
Interrupts and Resets
ECE 3120

Dr. Mohamed Mahmoud

http://iweb.tnitech.edu/mmahmoud/
mmahmoud@tntech.edu
3.1 Stack

3.2 Subroutines

3.3 Interrupts

3.4 Clock setting
What is stack?
- Stack is a section of the memory reserved for special use.
- Elements can be accessed (read/write) from only its top
- **Last-in-first-out** (LIFO) data structure

Much like a variable-length array
Stack overflow: The processor writes data into the stack too many times so that it writes in a location outside the area allocated to the stack.

Stack underflow: The processor reads data from the stack too many times so that it reads from a location outside the area allocated to the stack.

No hardware enforces the stack boundary.

The software must check the stack overflow and underflow to make sure that the program will not crash.
How the stack can be implemented?

- The stack pointer (SP) points at the top byte of the stack
- A stack grows from a high address toward lower address

- **Push operation** adds a new item on the top of the stack (write).
  1- SP is decremented by the number of bytes to be written to point to the new top of the stack
  2- Write on the location pointed by SP

- **Pull (Pop) operation** reads (= remove) an element from stack.
  1- Read the byte (or word) pointed by SP.
  2- SP is incremented by the number of bytes read to point to the new top of the stack

- Proper initialization of the SP is the responsibility of the programmer.
- It is also the programmer responsibility to keep the SP within the stack area
Pushing and pulling a byte

**Initial stack**

- $1005: EF
- $1006: B1
- $1007: F2
- $1008: A5

SP points at the top of stack
SP = $1005

### After executing pula

1. $A = [SP] = EF
2. $SP = SP + 1 = $1006

**A new top of stack**

- $1004: C1
- $1005: EF
- $1006: B1
- $1007: F2
- $1008: A5

### After executing psha and $a = C1

1. $SP = SP - 1 = $1004
2. $[SP] = A = C1

**A new top of stack**

- $1004: C1
- $1005: EF
- $1006: B1
- $1007: F2
- $1008: A5
Pushing and pulling a word

Initial stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1005</td>
<td>EF</td>
</tr>
<tr>
<td>$1006</td>
<td>B1</td>
</tr>
<tr>
<td>$1007</td>
<td>F2</td>
</tr>
<tr>
<td>$1008</td>
<td>A5</td>
</tr>
</tbody>
</table>

SP points at the top of stack
SP = $1005

After executing pulx

1- $X = [SP]$: [SP+1] = EFB1
2- SP = SP + 2 = $1007

A new top of stack

After executing pshx and $X = 3F5C

1- SP = SP - 2 = $1003
2- [SP]: [SP+1] = X = 3F5C

A new top of stack
- Push and pull instruction (except for pulc) do not affect on CCR bits
- Push and pull all the registers except the SP.
- There is no push or pull for memory

### Table 4.1 HCS12 push and pull instructions and their equivalent load and store instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Function</th>
<th>Equivalent instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>psha</td>
<td>push A into the stack</td>
<td>staa 1, -SP</td>
</tr>
<tr>
<td>pshb</td>
<td>push B into the stack</td>
<td>stab 1, -SP</td>
</tr>
<tr>
<td>pshc</td>
<td>push CCR into the stack</td>
<td>none</td>
</tr>
<tr>
<td>pshd</td>
<td>push D into stack</td>
<td>std 2, -SP</td>
</tr>
<tr>
<td>pshx</td>
<td>push X into the stack</td>
<td>stx 2, -SP</td>
</tr>
<tr>
<td>pshy</td>
<td>push Y into the stack</td>
<td>sty 2, -SP</td>
</tr>
<tr>
<td>pula</td>
<td>pull A from the stack</td>
<td>ldaa 1, SP+</td>
</tr>
<tr>
<td>pulb</td>
<td>pull B from the stack</td>
<td>ldab 1, SP+</td>
</tr>
<tr>
<td>pulc</td>
<td>pull CCR from the stack</td>
<td>none</td>
</tr>
<tr>
<td>puld</td>
<td>pull D from the stack</td>
<td>ldd 2, SP+</td>
</tr>
<tr>
<td>pulx</td>
<td>pull X from the stack</td>
<td>ldx 2, SP+</td>
</tr>
<tr>
<td>puly</td>
<td>pull Y from the stack</td>
<td>ldy 2, SP+</td>
</tr>
</tbody>
</table>
- Move instruction can be used to push/pull an immediate value or a memory to/from the stack as follows:

```assembly
movb #val,1,-SP ; push the 8 bit val 1 into stack
movw #val2,2,-SP ; push the 16 bit val 2 into the stack
movb 1,SP+,a ; pull 8 bits and store in a
movw 2,SP+,val2 ; pull 16 bits and store in x
```

**Manipulating SP**

- lds instruction can be used to initialize SP at the beginning of a program, e.g., lds #$3000

- leas can be used to manipulate stack pointer without pushing/popping values.

- leas with negative offsets creates space on the top of the stack, e.g.,
  leas 2,-sp ; sp = sp-2

- leas with positive offsets, removes data from the top of the stack, e.g.,
  leas 2,+sp ; sp = sp+2.
Example: What is the stack contents after the execution of the following instruction sequence.

```
lds #$1006 ;SP = 1006 initial value
ldaa #$20
psha ;SP = SP -1 = 1005
ldab #28
pshb ;SP = SP -1 = 1004
ldx #FF3A
pshx
```

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1002</td>
<td>FF</td>
</tr>
<tr>
<td>$1003</td>
<td>3A</td>
</tr>
<tr>
<td>$1004</td>
<td>28</td>
</tr>
<tr>
<td>$1005</td>
<td>20</td>
</tr>
<tr>
<td>$1006</td>
<td></td>
</tr>
</tbody>
</table>
```

SP = $1002.
Another example: See how SP and the stack contents change after push and pull operations.

1- The stack returns to its initial contents when we pull the same amount of data we pushed

2- when the order of the pull operations is the opposite of that of the push operations, the registers get their initial values.

If you need to use registers but you do not want to lose their contents, save them in stack, use then and then return the original values.
Common uses of stacks

1- Temporary data storage
   - Registers are limited resources.
   - If you want to use a register but you need its value for a later use, you can
     save the value temporarily on the stack and retrieve it later. See previous slide

     \[
     \text{psha} \quad ; \text{save the contents of A} \\
     \quad ; \text{use A in computations} \\
     \text{pula} \quad ; \text{restore the initial content of A}
     \]

2- Subroutine calls (interrupt service) and parameter passing
Details in next section.

3- Swap
Stack contents are retrieved in the reverse order from which they were
placed onto the stack.

\[
\begin{align*}
\text{psha} \\
\text{pshb} \\
\text{pula} \\
\text{pulb}
\end{align*}
\]

\{ \text{Swap a and b} \}
4- Reverse data structure

Write a code to reverse the order of an array with N elements stored starting from a memory location $1000. That means the element number N should be number 1 and the element number N-1 should be number 2 and so on.

```
ldx #$0FFF
ldab #00  ; b is a loop index
Loop1: incb
       cmpb #N
       bg start_pull
       ldaa b,x
       psha
       bra Loop1

start_pull:

ldab #00  ; b is a loop index
Loop2: incb
       cmpb #N
       bg _end
       pula
       staa b,x
       bra Loop2

_end:
```

Push the array elements

Pop the elements in reverse order and store them in the array
Outline

3.1 Stack

3.2 Subroutines

3.3 Interrupts

3.4 Clock setting
What is a Subroutine?
- It is common that the same sequence of instructions need to be executed in several places of the program.
- If we write the same code each time it is needed the program size will be large.
- Subroutine is a group of instructions that is written in memory only once and can be called multiple times from anywhere in the memory.

Why subroutines?
- Save memory
- Improve reusability of code: Subroutines can be used into different programs.
- Better organization: A complicated program can be more organized with subroutines.

However, program execution time increases.
A sequence of instructions

A mechanism is needed to:
1- Call
2- Return
3- Pass data
4- Return data (results)

Without subroutines
Write the code each time it is needed

The program flow with using a subroutine
1- Calling a subroutine

```assembly
bsr <opr> ; branch to subroutine
```

<opr> is specified in relative addressing mode and is in the range of -128 to +127 bytes from the instruction immediately after bsr.

```assembly
jsr <opr> ; jump to subroutine
```

<opr> is specified in direct, extended, indexed, and indexed indirect mode. The subroutine being called can be within the range of 64 kB.

How bsr and jsr work?

1- Save the return address that points to the next instruction after bsr or jsr.
   Push PC onto the stack. PC contains the address of the instruction following the jsr or bsr

2- jump to the subroutine. Load PC with the subroutine starting address

At the beginning of a subroutine, the top of the stack always contains the return address.
2- Returning from a subroutine

- The last instruction in a subroutine.
- It returns to the instruction immediately after the calling instruction
- Restore the return address from stack and load it in PC

When executing rts, the top of stack should contain the returning address, otherwise the program will crash.

To ensure that,
the amount of pushed data in a subroutine = the amount of popped data

```
  rts ;Return from subroutine
```

The stack after executing bsr
and before executing rts
The difference between branch instructions and subroutine calls

**Branch instructions:** PC = the target location address

**Call instructions**
1- Save PC (return address) in stack
2- PC = the target location address
Nested subroutines

Main program

Sub1:
- bsr Sub2
  - rts

Sub2:
- bsr Sub3
  - rts

Sub3:
- rts

Stack initially

After bsr Sub1

After bsr Sub2

After bsr Sub3

After rts of Sub1

After rts of Sub2

After rts of Sub3
A good programming habit that can reduce the chance of errors

Sub2:

```
psha
pshb
pshc
pushx
ldda $1000
psha
```

Local variable:
Use A, B, the flags, X and location $1000

```
pula
sta $1000
pulx
pulc
pulb
Pula
rts
```

Store the register (and memory) you use in the subroutine

Notice the order

This can be done in the program instead of the subroutine. Use push instructions before bsr and pull instructions after bsr.

Sub1 does not change the values stored in flags, registers or memory
3- Passing/returning data

- Subroutines usually expect inputs from the caller of the subroutine, and the caller usually expects the subroutine to return certain results.

Parameter passing techniques:

1- Passing parameters using memory:

The caller sets memory locations and the callee reads them.

Example:

```
movb #2,$1000
movb #4,$1001
bsr addition
```

The caller stores the data in memory locations, e.g., $1000 and $1001

```
addition:
  ldaa $1000
  adaa $1001
```

The subroutine reads the data from memory locations, e.g., $1000 and $1001
2- Passing parameters using registers
The caller sets registers and the callee reads them

Example:
```
ldaa #2
ldab #4
bsr
addition
```

```
addition:
aba
```

The caller stores the data in registers, e.g., a and b
The subroutine reads the data from the registers

3- Passing parameters using stack
The caller pushes the parameters in the stack and the callee reads them

Example:
```
ldaa #2
psha
ldaa #3
psha
bsr addition
leas 2,+sp
```

The caller pushes the data in stack before calling the subroutine
The stack is cleaned after the subroutine call
Parameter returning techniques:

1- Returning parameters using memory:
   To return results, the subroutine places the results in memory and the caller reads them.

2- Returning parameters using registers:
   The subroutine places the results in registers and the caller reads them.

3- Returning parameters using stack
   The caller creates a room of certain number of bytes in the stack before making call, the callee stores the results in these bytes.
In the Caller:

```
leas 3,-sp ; space for 3 bytes
bsr addition
pula
Staa $1000
pula
pulb
```

In the subroutine:

```
addition:

    tfr sp,X
    staa 2,+x ; set the first return value
    stab 1,+x ; set the second return value
    aba
    stab 1,+x ; set the third return value
    rts
```
Example: Draw the stack frame for the following program segment

```
ldd  #$1234  (1)
pshd
ldx  #$4000 (2)
pshx
jsr  sub_xyz (3)
...
```

```
sub_xyz: pshd (4)
pshx (5)
pshy (6)
leas -10,sp (7)
...
```

Why? This is a space for 10 bytes. They can be used as local variable inside the subroutine.

**Remember: when executing rts, the top of the stack must be the return address**
Write a subroutine to create a time delay that is a multiple of 100 ms. The multiple is passed in y.

Delay:  
  pshx ; save x

Eloop3:  
  ldx #60000 ; 2 E cycles

Iloop3:  
  psha ; 2 E cycles
  pula ; 3 E cycles
  psha ; 2 E cycles
  pula ; 3 E cycles
  psha ; 2 E cycles
  pula ; 3 E cycles
  psha ; 2 E cycles
  pula ; 3 E cycles
  psha ; 2 E cycles
  pula ; 3 E cycles
  psha ; 2 E cycles
  pula ; 3 E cycles
  nop ; 1 E cycle
  nop ; 1 E cycle
  dbne x, Iloop3 ; 3 E cycles
  dbne y, Eloop3 ; 3 E cycles
  pulx ; restore x
  rts

100ms
Example on using lookup tables

Write a subroutine “Square” to square an input number. The number to be squared (assumed unsigned) is passed in A. The result is returned in A. If the result is too big to fit in A, return $FF in A

What is lookup table?
Pre-calculate and store all the squared values (there are not that many), and then look up the correct answer based on the input number

Lookup tables applications
Very useful when the function to be computed is very complicated (e.g., sine, log, …), or represents some calibration values (e.g., temperature as a function of voltage)

1- Calculate the lookup table values
2- Using indexed addressing, it is easy to get the right value from the table.
   If X = the starting address of the table, and A = the input number (i.e., 0 to 15), then ldaa a,x will fetch the square value
Pseudocode

If (A > 15)
    Load A with $FF
Else
    Load A with the value in the table, at the location indexed by X+A
End

lda #7
jsr Square ;Call subroutine to compute the square of A
;A = 49, the square of 7
-----------
Square: pshx

ldx #Square_table ;x = Starting address of the lookup
cmpa #15 ;Check if A > 15
bhi tooBig ;Branch if A > 15
ldaa a,x ;A <= 15, so get the result
bra _End

tooBig: ldaa #$FF ;Signal overflow with $FF
_End: pulx
rts

Square_table dc.b 0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,255
Write a subroutine called BCDtoASCII to convert the two BCD numbers stored in B to two ASCII code numbers in D (A:B). If the numbers in B are not BCD, return FF in A.

Pseudocode
Since the number is BCD, we expect numbers from 0 to 9 so to convert to ASCII we need to add $30

1- Separate the two numbers and store them in A and B
   1.1 Clear A
   1.2 Shift D to left 4 times. The most significant BCD number is in A
   1.3 Shift B to right 4 times, now B = the least significant BCD number

2- If (the numbers in A and B are not BCD) then A = FF and end
   Else Add $30 for A and B to convert them to ASCII

BCDtoASCII:

;1- separate the two numbers. Store them in A and B
clr a  ;1.1 Clear A
lsld ; shift D 4 times. A = least significant BCD
lsld ;
lsld ;
lsld ;
lsld ;
.; shift B 4 times. B = most significant BCD
lsrb
lsrb
lsrb
lsrb

;2- check if the A and B are BCD
cmpa #10
bhs not_BCD ; branch if higher or same
cmpb #10
bhs not_BCD ; branch if higher or same

; The two branches are not taken so A and B are BCD
; Convert to ASCII by adding $30
adda #$30
addb #$30
bra _end ; skip the not-bcd statement

; not BCD -------------------------------
not_BCD: ldaa #FF

_end:
    rts
Write a subroutine called HextoASCII to convert the two Hex numbers stored in A to two ASCII code numbers in A and B

- Since the numbers are in hex, they are from 0 to 15.
- For the numbers from 0 to 9, add $30 to convert to ASCII
- For the numbers from 10 to 15, add $37 to convert to ASCII why?
- The ASCII of A, B, .. = $41, $42, ...

Pseudocode
1- Separate the two numbers and store them in A and B
2- Convert the two numbers to ASCII
   2.1 **If** (A <= 9) **then** A = A + $30 **else** A = A + $37
   2.1 **If** (B <= 9) **then** B = B + $30 **else** B = B + $37

No need to check if the number is hex because all the numbers from 0 to 15 are hex
HexToASCII:
; 1- separate the two numbers. Store them in A and B
    tba ; a = b
    lsra ; Shift right A 4 times. A = The most significant hex
    lsra ; 
    lsra ; 
    lsra ; 
    andb #$0F ; B = The least significant hex

; Convert A to ASCII
    cmpa #10
    bhs digit_A_F ; it is digit from A to F
    ; Branch is not taken so it is a number 0 to 9
    adda #$30
    bra work_on_B

digit_A_F: adda #$37

; Convert B to ASCII
work_on_B: cmpb #10
    bhs digit_A_F_1
    addb #$30
    bra _end

digit_A_F_1: addb #$37

_end: rts

Equivalent shorter code

adda #$30
cmpa #10
blo work_on_B
adda #$7
Write a subroutine called BinarytoASCII to store the binary numbers stored in A as ASCII's and store the result in memory locations pointed by X to X+7. Do not change the contents of A or X.

Ex. if A = 11010001, the locations from X to X+7 should have $31, $30, $30, $30, $31, $30, $31, $31. The ASCII's of 1 and 0 are $31 and $30.

Steps:
1- Shift right A
2- If the carry is 1, store 31 else store 30
3- Repeat the steps 1 and 2 eight times.

BinarytoASCII:
   ; save values of a, b and x
   psha
   pshb
   pshx

   clrb       ; b is the loop index

Loop:  lsra
       bcs store_one ; branch if the carry is set
       movb #$30,b,x ; branch is not taken so the bit is 0
       bra _loopend  ; skip next statement
store_one: movb #$31,b,x

_loopend: incb
    cmpb #8
    blo Loop

; restore the original values of x, b and a
pulx
pulb
pula
rts
Write a subroutine that returns FF if two strings are identical. The beginning addresses of the strings and the result are passed on stack. Each string ends by 0.

```assembly
leas 1,-sp ;reserve a location for the result
ldx #string2
pshx
ldx #string1
pshx
bsr check_strings
leas 4,+sp
pula ; a = the result
```

Pass the strings’ starting addresses and allocate a byte for the return

Remove the passed arguments and read the result
check_strings:
    pushx
    pushy
    movb #$FF,6,sp ;Result = FF assume the two strings are identical

    movw 2,sp,x ;x = the beginning address of String1
    movw 4,sp,y ;y = the beginning address of String2

Loop:
    ldaa 0,x+ ;a = the first byte of String1, x = x+1
    ldab 0,y+ ;b = the first byte of String2, y = y+1
    cba
    bne not_identical ;String1<>String2 if one byte is not equal.
    ; the loop ends when a = b = 0
    cmpa #0
    beq _end
    bra Loop

not_identical: movb #$00,6,sp ;Result = 00 the strings are different

_end:
    puly
    pulx
    rts
Write a subroutine called bin2dec that converts a 16-bit binary number stored in D into a BCD ASCII string. The pointer to the buffer that holds the string is passed in Y. The string must be terminated by Null (0).

This program is the improvement of the one given in chapter 2. Try to identify the differences.

Ex., If $D = \%0000 0010 0110 1100$ (or 620 decimal)

Steps:
1- Push the Null character (ASCII of $00$) in the stack.
2- Divide D by 10, $D =$ quotient and push (the remainder + $30$) in stack
3- Repeat step 2 as long as the quotient $<> 0$
4- Pull from stack and store in buffer

Why stack? We get the least significant digit first but it should be stored last. The stack is used to reverse the order of the resultant numbers. It may difficult to store the numbers directly in the buffer because we do not know how many digits will be resulted before doing the conversion.
org 1000
buffer rmb 6
org 3000
ldd #620 ;set d
ldy #buffer ;set y
jsr bin2dec
-------------

Bin2dec:
pshx ;save X in stack
movb #0,1,-SP ;push NULL character into stack
Divloop: ldx #10 ;divide the number by 10
       idiv ;D/10, D = remainder and X = quotient
       addb #$30 ;convert the remainder to ASCII code
       pshb ;push it into the stack
       xgdx ;swap quotient to D
       cpd #0 ;if quotient=0, prepare to pull out
       beq revloop ;the decimal digit characters
       bra divloop ;quotient <> 0, divide by 10
revloop: pula ;start to reverse the string from stack
       staa 1,y+ ;save ASCII string in the buffer
       cmpa #0 ;reach the NULL pushed previously?
       beq done ;if yes, then done
       bra revloop ;continue to pup
Done: pulx ; restore the index register X
       rts
- When a subroutine needs very few local variables, the subroutine can use registers to hold them.
- However, when there are not enough registers to hold local variable, the subroutine will need to assign local variables to the stack
- This program is an example on using the stack to store the local variables of a subroutine.
- In the program, it is assumed that the number in D is unsigned, how the program should change if the number is signed??

1- Check if the number is positive or negative
2- If the number is positive, the previous subroutine can be used as is
3- If the number is negative:
   (1) add the ASCII of ‘-’ at the beginning of the string
   (2) Negate the number and then use the previous subroutine as is

To practice:-

(1) Modify the previous subroutine assuming the number in D is signed.
(2) Do the reverse of the previous program. Convert a string of the ASCII codes of a signed decimal number to binary.
Outline

3.1 Stack

3.2 Subroutines

3.3 Interrupts

3.4 Clock setting
An interrupt is an event that causes the CPU to stop the program execution briefly and run a subroutine, called Interrupt Service Routine (ISR), related to the event. After executing the ISR, the CPU resumes the program from the point it left the program.

- The ISR code should take some necessary actions appropriate to the event.
- Serving an interrupt means executing its subroutine.
- An interrupt can happen anywhere in the program at any time, but the CPU waits for the current instruction to be completed before responding to the interrupt.

```
org $2000
Entry:
  movb #10,$1000
  movb #15,$1001
  movb #20,$1002
  ldaa $1000
  adda $1001
  adda $1002
  staa $1003
```

ISR

```
RTI
```
- An example of an event: A sensor outputs a pulse when a train approaches a crossing
- This signal causes an interrupt and the MCU run the relevant routine that lowers the crossing gate

An interrupt means an event has occurred and a relevant action should be taken
An interrupt is caused by a hardware event, something going wrong, or software initiated interrupts.

1- **Internal**: On-chip peripheral devices such as the timers, parallel ports,..

2- **External**: Off-chip circuits connected to hardware pins IRQ or XIRQ

3- **Interrupt-like cases**:
   - RESET (power on)
   - Software errors that prevent the instruction from being completed can cause software interrupts.
     - **Examples**: divide-by-0, unimplemented opcode (there are 202 unimplemented opcodes (16-bit opcode))
     - This interrupt simply outputs error message and abort the program.
Two ways to detect events (interrupt request signals):

1- **Polling**
   - To avoid missing an interrupt signal, the CPU must continuously read the interrupt pin. **The CPU can’t do anything else** (e.g., sensing and controlling other devices)
   - However, **faster response to events**. Interrupt consumes time in saving and restoring of registers, etc.

2- **Interrupt**
   - Better utilization of CPU, i.e., CPU doesn’t waste time in polling
   - Remind the CPU to perform routine tasks every programmed timeout.
   - I/O operations can be performed more efficiently. I/O operations are usually slow. Interrupt can prevent the CPU from being tied up during the I/O operations, i.e., **CPU does not wait until I/O device is ready**.
   - Fast enough for performing time critical applications such as real-time data sample systems and power failure
- **Maskable interrupts**: Can be ignored by the CPU. A program can ask the CPU to respond (or ignore) a maskable interrupt.

- **Non-maskable interrupts**: Can’t be disabled or ignored by the CPU.

1- **Maskable interrupts**

- Includes /IRQ pin and all peripheral devices.

- Depending on the application and situation, some interrupts may not be desired or needed and should be prevented from affecting the CPU. e.g., a section of the program needs to be uninterrupted

2- **Two levels of interrupt enabling capability**

1- **Global masking capability**

- When none of the maskable interrupts are desirable, the processor can disable them by setting the global interrupt mask flag (I in the CCR)

To set flag I (disable all maskable interrupts): sei or orcc #%00010000

To clear flag I (enable all maskable interrupts): cli or andcc #%11101111
- By default, the I bit is set to 1 during reset. An instruction should be written to clear the I bit to enable maskable interrupts.

2- A local interrupt masking capability

Each interrupt source has an enable bit to enable/disable this interrupt source.

2- Nonmaskable interrupts

- /XIRQ pin interrupt and unimplemented opcode trap.

- Non-maskable interrupts are often used to deal with major system failures, such as loss of power.

- The ISR responding to a power failure can save the contents of the registers in a nonvolatile memory when the voltage drops below a threshold.

- After reset, X bit is set to 1 which means nonmaskable interrupts are disabled.

- As long as the X bit remains set, interrupt service requests made via the XIRQ pin are not recognized.
- After minimal system initialization, software can clear the X bit of the CCR register to enable the XIRQ interrupt using andcc #\%10111111 ; or #$BF

- Once enabled, software cannot reset the X bit, and hence it becomes "nonmaskable"

- Enabling non-maskable interrupts before a system is initialized can lead to undesired interrupts. The X bit provides a mechanism for enabling non-maskable interrupts after a system is stable.

- The X bit is not affected by maskable interrupts.
Interrupt Flag Bit

- The interrupt signal sets a hardware flag (i.e., a flip-flop) to request service. (different from the CCR flags)
- The CPU can tell that an event occurred when this flip flop is one. It doesn’t have to catch the input pulse when it happens.
- If the CPU is busy at the time of interrupt occurrence, the system can memorize the events.
- The interrupt flag bit should be cleared when the interrupt is served. As long as this bit is set, there is an interrupt that is not served yet.
- An interrupt is called pending when it is active, but not served yet.
  – Before executing an instruction, the hardware checks if there is an interrupt, i.e., one interrupt flag is set.
- If there is no interrupt, the CPU executes the next instruction in the main program, otherwise it executes the related ISR.
- Pending Interrupt Requests will be served after returning from serving the currently executed interrupt.
  – While the I bit is set (disabled), maskable interrupts can become pending and are remembered, but they will not be served until the I bit is cleared.
Interrupt signal path

When enabled, takes control of CPU, and does these things:
1. Save registers on stack
2. Fetch address of ISR
3. Disable interrupts

The I-bit is the "gatekeeper" for all the maskable interrupts

In addition to the internal subsystems, /IRQ is an external pin that can generate an interrupt

The "/" means that the signal is asserted when low

These switches are implemented using AND gates

- There are many devices that can cause interrupts
- Each sets a flag
- Each has its own interrupt enable switch
- As shown in previous slide, an interrupt happens if (1) the global interrupt is enabled, (2) a device local interrupt is enabled, and the device flag is set.

- **Interrupt Overrun**: Second interrupt occurs before processing of first complete, i.e., CPU misses an event.

- An interrupt service routine should be as short as possible to avoid interrupt overrun.

![Diagram]

**Diagram Explanation**:
- **Interrupt enable**: Set/clear by the program.
- **Interrupt flag**: Set by device when it wants interrupt, cleared by program when the interrupt is served.
- **I bit in CCR**: Indicates if there is an interrupt.

1. **Interrupt request from one device**
2. **Interrupt request from different devices**
- To set up IRQ interrupt, the programmer should store the ISR starting address (vector) at the designated location as follows:

```assembly
 Org $FFF2
 dc.w #IRQ_ISR
 movw #IRQ_ISR, $FFF2
```

OR

- The label at the beginning of the IRQ interrupt service routine is IRQ_ISR.

- To serve an interrupt, the CPU needs to know the starting address of the interrupt service routine. This address is called **Interrupt Vector**
- The interrupt vector of each interrupt source should be stored at a predefined fixed memory location called **Vector address**.
- **Vector table**: A table where all interrupt vectors are stored. CPU needs to access this table to obtain the interrupt vector.
- Programmer cannot change vector address, but certainly can change the interrupt vector.
- All HCS12 interrupt vectors are stored in a table located at $FF80 to $FFFF. See Table 1.6.
- From Table 1.6, the unimplemented opcodes share the same vector $FFF8:$FFF9.
- The interrupt vector address of IRQ is $FFF2 and $FFF3.

- **To set up IRQ interrupt, the programmer should store** the ISR starting address (vector) at the designated location as follows:

```assembly
 Org $FFF2
dc.w #IRQ_ISR
```

- The label at the beginning of the IRQ interrupt service routine is IRQ_ISR.
Contains the interrupt vectors of all interrupts

When IRQ interrupt occurs, the CPU fetches the ISR starting address from $FFF2:$FFF3

Vector address: $FFF2:$FFF3
Interrupt vector: 4103

We cannot change $FFF2:$FFF3 but we can change 4103

CPU should resolve the starting address of the interrupt service routine to jump to the interrupt subroutine: PC = ISR starting address
Table 6.1 Interrupt vector map

<table>
<thead>
<tr>
<th>Vector address</th>
<th>Interrupt source</th>
<th>CCR mask</th>
<th>Local Enable</th>
<th>HPRIO value to elevate to highest I bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFE</td>
<td>Reset</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$FFFC</td>
<td>Clock monitor reset</td>
<td>none</td>
<td>COPCTL(CME,FCME)</td>
<td></td>
</tr>
<tr>
<td>$FFFA</td>
<td>COP failure reset</td>
<td>none</td>
<td>COP rate selected</td>
<td></td>
</tr>
<tr>
<td>$FF8</td>
<td>Unimplemented instruction trap</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$FF6</td>
<td>SWI</td>
<td>none</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$FF4</td>
<td>XIRQ</td>
<td>X bit</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$FF2</td>
<td>IRQ</td>
<td>1 bit</td>
<td>INTCR(IRQEN)</td>
<td>$F2</td>
</tr>
<tr>
<td>$FF0</td>
<td>Real time interrupt</td>
<td>1 bit</td>
<td>RTICL(RTIE)</td>
<td>$F0</td>
</tr>
<tr>
<td>$FFEE</td>
<td>Timer channel 0</td>
<td>1 bit</td>
<td>TMSK1(C0l)</td>
<td>$EE</td>
</tr>
<tr>
<td>$FFEC</td>
<td>Timer channel 1</td>
<td>1 bit</td>
<td>TMSK1(C1l)</td>
<td>$EC</td>
</tr>
<tr>
<td>$FFEA</td>
<td>Timer channel 2</td>
<td>1 bit</td>
<td>TMSK1(C2l)</td>
<td>$EA</td>
</tr>
<tr>
<td>$FFE8</td>
<td>Timer channel 3</td>
<td>1 bit</td>
<td>TMSK1(C3l)</td>
<td>$E8</td>
</tr>
<tr>
<td>$FFE6</td>
<td>Timer channel 4</td>
<td>1 bit</td>
<td>TMSK1(C4l)</td>
<td>$E6</td>
</tr>
<tr>
<td>$FFE4</td>
<td>Timer channel 5</td>
<td>1 bit</td>
<td>TMSK1(C5l)</td>
<td>$E4</td>
</tr>
<tr>
<td>$FFE2</td>
<td>Timer channel 6</td>
<td>1 bit</td>
<td>TMSK1(C6l)</td>
<td>$E2</td>
</tr>
<tr>
<td>$FFE0</td>
<td>Timer channel 7</td>
<td>1 bit</td>
<td>TMSK1(C7l)</td>
<td>$E0</td>
</tr>
<tr>
<td>$FFDE</td>
<td>Timer overflow</td>
<td>1 bit</td>
<td>TMSK2(TOI)</td>
<td>$DE</td>
</tr>
<tr>
<td>$FFDC</td>
<td>Pulse accumulator overflow</td>
<td>1 bit</td>
<td>PACTL(PAOVI)</td>
<td>$DC</td>
</tr>
<tr>
<td>$FFDA</td>
<td>Pulse accumulator input edge</td>
<td>1 bit</td>
<td>PACTL(PAI)</td>
<td>$DA</td>
</tr>
<tr>
<td>$FFD8</td>
<td>SPI serial transfer complete</td>
<td>1 bit</td>
<td>SPOCR1(SPIE)</td>
<td>$D8</td>
</tr>
<tr>
<td>$FFD6</td>
<td>SCI0</td>
<td>1 bit</td>
<td>SC0CR2(TIE,TCIE,RIE,ILIE)</td>
<td>$D6</td>
</tr>
<tr>
<td>$FFD4</td>
<td>SCI1</td>
<td>1 bit</td>
<td>SC1CR2(TIE,TCIE,RIE,ILIE)</td>
<td>$D4 (1,3,4)</td>
</tr>
<tr>
<td>$FFD2</td>
<td>ATD0 or ATD1</td>
<td>1 bit</td>
<td>ATDxCTL2(ASCIIE)</td>
<td>$D2</td>
</tr>
<tr>
<td>$FFD0</td>
<td>MSCAN 0 wakeup</td>
<td>1 bit</td>
<td>C0RIER(WUPIE)</td>
<td>$D0 (1*,2,2*)</td>
</tr>
<tr>
<td>$FFCE</td>
<td>Key wakeup J or H</td>
<td>1 bit</td>
<td>KWIJE[7:0] and KWIEN[7:0]</td>
<td>$CE (1,3,4)</td>
</tr>
</tbody>
</table>
## Table 6.1 Interrupt vector map

<table>
<thead>
<tr>
<th>Vector address</th>
<th>Interrupt source</th>
<th>CCR mask</th>
<th>Local Enable</th>
<th>HPIO value to elevate to highest 1 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFCC</td>
<td>Modulus down counter underflow</td>
<td>I bit</td>
<td>MCCTL(MCZI)</td>
<td>$C C</td>
</tr>
<tr>
<td>$FFCA</td>
<td>Pulse accumulator B overflow</td>
<td>I bit</td>
<td>PBCTL(PBOVI)</td>
<td>$C A</td>
</tr>
<tr>
<td>$FFC8</td>
<td>MSCAN 0 errors</td>
<td>I bit</td>
<td>C0RIER(RWRNIE,TWRNIE, RERRIE, TERRIE, BOFFIE, OVRIE)</td>
<td>$C8 (2*,3,4)</td>
</tr>
<tr>
<td>$FFC6</td>
<td>MSCAN 0 receive</td>
<td>I bit</td>
<td>C0RIER(RXFIE)</td>
<td>$C6 (2*,3,4)</td>
</tr>
<tr>
<td>$FFC4</td>
<td>MSCAN 0 transmit</td>
<td>I bit</td>
<td>C0TCR(TXEIE[2:0])</td>
<td>$C4 (2*,3,4)</td>
</tr>
<tr>
<td>$FFC2</td>
<td>CGK lock and limp home</td>
<td>I bit</td>
<td>PLLCR(LOCKIE, LHIE)</td>
<td>$C2 (3,4)</td>
</tr>
<tr>
<td>$FFC0</td>
<td>IIC Bus</td>
<td>I bit</td>
<td>IBCR(IBIE)</td>
<td>$C0 (4)</td>
</tr>
<tr>
<td>$FFBE</td>
<td>MSCAN 1 wakeup</td>
<td>I bit</td>
<td>C1RIER(WUPIE)</td>
<td>$BE (4)</td>
</tr>
<tr>
<td>$FFBC</td>
<td>MSCAN 1 errors</td>
<td>I bit</td>
<td>C1RIER(RWRNIE, TWRNIE, RERRIE, TERRIE, BOFFIE, OVRIE)</td>
<td>$BC (4)</td>
</tr>
<tr>
<td>$FFBA</td>
<td>MSCAN 1 receive</td>
<td>I bit</td>
<td>C1RIER(RXFIE)</td>
<td>$BA (4)</td>
</tr>
<tr>
<td>$FFB8</td>
<td>MSCAN 1 transmit</td>
<td>I bit</td>
<td>C1TCR(TXEIE[2:0])</td>
<td>$B8 (4)</td>
</tr>
<tr>
<td>$FFB6</td>
<td>Reserved</td>
<td>I bit</td>
<td></td>
<td>$B6</td>
</tr>
<tr>
<td>$FF80-$FFB5</td>
<td>Reserved</td>
<td>I bit</td>
<td></td>
<td>$80-$BB4</td>
</tr>
</tbody>
</table>

**Note:**
1. Available in 812 A4
2. Used as BDLC interrupt vector for 912B32, 912B32. 2*. Available in 912BC32
3. Available in D60
4. Available in DG128 (DT128)
It is possible that several interrupts would be pending at the same time. The CPU has to decide which interrupt to serve first → the interrupts should be prioritized.

- Microprocessor cannot serve more than one interrupt at the same time.
- An interrupt with highest priority receives service before interrupts at lower priorities.
- HCS12 prioritizes interrupts in hardware.
- In the table 6.1, the interrupts that have higher vector addresses are at higher priorities.
- RESET has highest priority
- /XIRQ has higher priority than any other hardware interrupts.
- /IRQ has the highest priority among the maskable interrupts.
- Timer channel 0 has higher priority than the timer channel 1.

We have limited capability to program the maskable interrupts by raising one of the maskable interrupts to the highest level within the group of maskable interrupts so that it can get quicker service.
- The relative priorities of the other sources remain the same
- The priorities of reset and nonmaskable interrupts are not programmable.
- To raise a maskable interrupt source to the highest priority, write the low byte of the vector address of this interrupt to the highest priority interrupt register (HPRIO).
- Example: to raise the capture timer channel 7 interrupt to the highest priority, write the value of $E0 to the HPRIO
- The address of HPRIO is $001F.  
  \[ \text{movb} \ #E0, \ \text{HPRIO} \]
- No need to memorize the address, it is defined in “mc9s12dg256.inc” file as follow:  HPRIO equ $001F
- Include the file, after that you can use the name “HPRIO”

\[
\text{include 'mc9s12dg256.inc'} \\
\text{movb} \ #E0, \ \text{HPRIO}
\]

![Figure 6.1 Highest priority I interrupt register](image)
1. When an event takes place, it sets a flag bit to interrupt the CPU.

2. This interrupt request is served if:
   
   (1) I bit and local interrupt enable bit are enabled in case maskable
       interrupts and X bit is enabled in case of nonmaskable interrupt
   
   (2) It is the only interrupt or the highest priority interrupt if there are
       several interrupts pending for service

3. The CPU recognizes the interrupt request and responds to it when it
   completes the execution of the current instruction

4. The CPU automatically pushes all registers (except SP) on the stack (9 bytes total).
   This includes the return address stored in
   PC and CCR register do not mess up stack!
5. The CPU prevents further interrupts from occurring till the ISR is done by setting the I bit (and also setting the X bit if the interrupt was caused by the XIRQ signal).

   - By default the MCU avoids nested interrupts
   - During serving a maskable interrupt, other maskable interrupts cannot be served but non-maskable interrupts can happen.
   - XIRQ interrupt is disabled during a system reset and upon entering the service routine of another XIRQ interrupt.

6. Resolve the interrupt vector and transfer control to the interrupt service routine (ISR). PC = the ISR starting address

7. Cancel the interrupt request by clearing the interrupt flag. Without this step the interrupt will be executed over and over again and the main program would never execute again

8. Execute the ISR instructions.

   Use all the CPU registers without fear of interfering with the main program, but for memory locations, it is the programmer’s responsibility to ensure that the ISR does not change memory locations used by the main program
9. The last instruction in an ISR is always “RTI” (return from interrupt)
   - RTI retrieves the registers that were present before the interrupt occurred from the stack.
   - This will automatically:
     (1) Enables X and I bits because the CCR is stacked before the X and I bits are disabled
     (2) transfers control back to the main program since the stacked PC contains the return address.

10. Resume the interrupted code from the place where the main program was interrupted.
Interrupt programming

1- Main program

1.1 Initializing the interrupt vector

```
Org $FFF2 ;$FFF2 is the vector table address
dc.w #IRQ_ISR ;#IRQISR is the starting address of the IRQ ISR, store it in the vector address
```

1.2 Initializing the stack pointer

```
lds #$4000
```

1.3 Configure the subsystems, e.g., respond to level or falling edge, ports are input or output, etc.

1.4 Enable interrupt

- Enable the local interrupts for each of the subsystems
- By default, the I and X bits are set to 1 during reset. Clear the I and X bits to enable maskable and nonmaskable interrupts.
- It is important not to enable the interrupts before the initialization is done
2- The interrupt subroutine

1- Clear the interrupt flag bit

2- Must end with RTI instruction (not RTS!)

The Overhead of Interrupts

- Although interrupt mechanism provides many advantages, it also involves some overhead including: instructions to initialize the interrupt, fetching the interrupt vector, push/pop registers, and executing RTI.

- That is why polling can respond quicker to events.

- Interrupt latency varies according to the number of cycles required to complete the current instruction, and waiting time for serving higher priority interrupts if there are
Interrupts versus subroutines

- Both leave the main program to execute a sequence of code and return back to where it left the program.

- A subroutine call is expected but interrupt is not expected in the sense that it can happen at any time.

- A subroutine needs a call instruction but an interrupt is called when an event happens.

- Different instructions to return to the main program (RTI and RTS).

- Subroutine call instruction saves the return address in stack and upload PC with the subroutine starting address, but interrupts: (1) saves all registers in stack, (2) upload PC with the ISR starting address, (3) disable interrupt to avoid nested interrupts.

- To return from a subroutine: PC = the top of stack.

- To return from an ISR: pull all registers including PC, enable interrupt.
- The only external maskable interrupt for the HCS12.

- **When does the interrupt occur?**

1- **Level-sensitive**
   - External interrupt is generated as long as the pin is low
   - Pros: Allows multiple external interrupt sources to be tied to this pin. Whenever one of the interrupt sources is low, an interrupt request will be detected.
   - Cons: The user should make sure that the IRQ signal is de-asserted (goes high) before the HCS12 exits the interrupt service routine if there are no other pending interrupt request.

2- **Edge-sensitive**
   - External interrupt generated when there is a falling edge in IRQ pin
   - Pros: User does not need to be concerned about the duration of the assertion time of the IRQ signal
   - Cons: Not suitable for noisy environment because every falling edge caused by noise will be recognized as an interrupt.
The IRQ interrupt is configured by programming the interrupt control register (IRQCR) (also known as INTCR). The address of IRQCR is $001E.

- IRQ interrupt has a local enable bit (IRQEN) which is bit 6 of the IRQCR register. IRQEN = 1 enabled and IRQEN = 0 disabled.

- IRQ interrupt can be edge-triggered or level-triggered. The triggering method is selected by programming the IRQE bit of the IRQCR. IRQE = 0 respond to low level, and IRQE = 1 responds to falling edge.

- Contents of the IRQCR register are 01000000 on reset.

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQE</td>
<td>IRQEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Reset:

<table>
<thead>
<tr>
<th></th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQE</td>
<td>IRQEN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**IRQE** -- IRQ edge sensitive only bit
- IRQE can be written once in normal mode. In special modes, it can be written any time, but the first write is ignored.
  - 1 = IRQ pin responds only to falling edge
  - 0 = IRQ pin responds to low level.

**IRQEN** -- IRQ enable bit
- IRQEN bit can be written any time in all modes. The IRQ pin has an internal pullup.
  - 1 = IRQ pin interrupt enabled
  - 0 = IRQ pin interrupt disabled

Figure 6.2 Interrupt control register (IRQCR)
- A reset is nonmaskable.
- The vector interrupt of Reset is $FFFE:$FFFE
- The reset service routine is stored in nonvolatile memory
- Do not save registers because CPU won’t return back to the main program
- The reset initializes some CPU registers, flip-flops, and the control registers in I/O interface chips in order for the computer to function properly.
- **Two types of resets:**
  1. Power-on reset: when the power is on
  2. Manual reset: when a signal on a /RESET pin is asserted. to get out of most error conditions. Power does not go down,
- After reset, both I and X bits in the CCR register are set. This disables interrupts to allow initializing the interrupts
- After a reset, the CPU always starts at a defined address.
The IRQ pin of HCS12 is connected to a 1-Hz digital waveform and Port B (a parallel port on the microprocessor) is connected to eight LED’s. Write a program to count the number of the input cycles at /IRQ pin and output it to Port B. The ISR should respond to a falling edge on the /IRQ pin.

In main program:
1. Write ISR starting address into the IRQ vector address
2. Configure Port B as output port
3. Configure IRQ to respond to negative edge
4. Enable interrupts including IRQ local interrupt and global maskable interrupts
5. Run endless loop while awaiting interrupt

In the ISR:
1. Increment the count
2. Send Count to LEDs
include 'mc9s12dg256.inc';include register equates

org $1500

count rmb 1

org $2000

lds #$5000 ; initialize the SP
clr count ; start count at 0

movb #$FF,DDRB ; configure Port B as output by sending 1's to its direction register DDR

movw #IRQISR,Virq ; Put ISR starting address in vector location ; Virq is the vector of IRQ

movb #$C0,INTCR ; enable IRQ interrupt and select edge triggering(b7 and b6 of INTCR register)

movb #$00,PORTB ; Initial the port all LEDs are off

cli ; global enable of interrupts

forever: bra forever ; just waiting for interrupt to occur

;IRQ service routine
IRQISR:

inc count ; increment count

movb count,PORTB ; send count to LEDs

rti
Outline

3.1 Stack

3.2 Subroutines

3.3 Interrupts

3.4 Clock setting
The microcontroller needs clock signal to function. The clock signal has the form of square waveform.

How to produce clock signal? The user can choose between:

1- Using external crystal
   - The external crystal is connected between the EXTAL (pin 46) and XTAL (pin 46).
   - The crystal is connected to on-chip circuit that can generate the clock.

2- Using external oscillator
   - The external clock source provided by the oscillator is connected to the EXTAL pin (46).
   - The /XCLKS (pin 36) must be grounded to select the external clock signal (EXTAL).

   As shown in next slide’s figure, OSCCLK is the oscillator clock (either from crystal or external oscillator).
   - By programming PLLSEL bit, OSCCLK may bypass the PLL or go through the PLL circuit.
   - The PLL circuit has the capability to multiply incoming signal frequency and stabilize its output signal frequency.
External oscillator connections (/XCLKS = 1)

Common crystal connections (/XCLKS = 1)

Figure 6.15 HCS12 clock generation circuit
- Either the OSCCLK or the PLLCLK can be chosen as the system clock SYSCLK.

- SYSCLK is divided by 2 to derive the bus clock to control the instruction execution and peripheral operation (E-clock from timing).

Phase Locked Loop

- The frequency of the PLLCLK is controlled by registers synthesizer (SYNR) and reference divide (REFDY) using the following equation:

\[
\text{PLLCLK} = 2 \times \text{OSCCLK} \times \frac{\text{SYNR} + 1}{\text{REFDV} + 1}
\]

![Figure 6.8 The CRG synthesizer register (SYNR)](image)

![Figure 6.9 The CRG reference divider register (REFDV)](image)
- Maximum bus frequency (E-clock) should be 25Mhz. We cannot exceed this limit.

- Therefore, the maximum PLLCLK is 50MHz

SYNR: 0 to 63 \[ \text{PLLCLK}_{\text{max}} = 2 \text{ OSCCLK} \left( \frac{64}{1} \right) = 128 \text{ OSCCLK} \leq 50\text{MHz} \]

REFDV: 0 to 15 \[ \text{PLLCLK}_{\text{min}} = 2 \text{ OSCCLK} \left( \frac{1}{16} \right) = \left( \frac{1}{8} \right) \text{ OSCCLK} \]

- The synthesizer (SYNR) and reference divider (REFDV) registers are programmed to get the desired frequency.
PLL operation is controlled by four registers: PLLCTL, CRGFLG, and CLKSEL.

1- The PLL control register (PLLCTL)

```
    7  6  5  4  3  2  1  0
    CME PLLON  AUTO  ACQ  0  PRE  PCE SCME
```

PLLON: phase lock loop on bit
- 0 = PLL is turned off
- 1 = PLL is turned on. If AUTO bit is set, the PLL will lock automatically.

AUTO: automatic bandwidth control bit
- 0 = automatic mode control is disabled and the PLL is under software control, using ACQ bit.
- 1 = high bandwidth filter is selected

```
movb #$60,PLLCTL
```

Write %0110 0000 to the PLLCTL to enable the PLL and let it lock automatically.

2- CRG Flags register (CRGFLG)

```
RTIF PORF 0 LOCKIF LOCK TRACK SCMIF SCM
```

- LOCK — Lock Status Bit
  - 1 = PLL VCO is within the desired tolerance of the target frequency.
  - 0 = PLL VCO is not within the desired tolerance of the target frequency.

```
pll1: brclr CRGFLG,#$08,pll1 ;wait for PLL to lock
```
3- CLKSEL

<table>
<thead>
<tr>
<th>PLLSEL</th>
<th>PSTP</th>
<th>SYSWAI</th>
<th>ROAWAI</th>
<th>PLLWAI</th>
<th>CWAI</th>
<th>RTIWAI</th>
<th>COPWAI</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

PLLSEL: PLL select bit
- 0 = system clocks are derived from OSCCLK
- 1 = system clocks are derived from PLLCLK

PSTP: pseudo stop bit
This bit controls the functionality of the oscillator during the stop mode.
- 0 = oscillator is disabled in stop mode
- 1 = oscillator continues to run in stop mode (pseudo mode). The oscillator amplitude is reduced.

```
movb    #$80,CLKSEL     ;select PLLCLK
movb    #$40,CLKSEL     ;select OSCCLK
```
There is a system that derives its bus clock from the PLL circuit and an crystal oscillator of 8 MHz is selected. The desired bus clock is 24 MHz. Write an assembly subroutine to perform the configuration.

The frequency of OSCCLK is 8 MHz.
SYNCLK frequency = 2 x 24 = 48 MHz = PLL clock

48 MHz = 2 x 8 MHz x \( \frac{SYNR + 1}{REFDV + 1} \)

\( \frac{SYNR+1}{(REFDV +1)} = 3 \)

One solution is to set SYNR and REFDV to 2 and 0

```assembly
SYNR equ $0034
REFDV equ $0035
PLLCTL equ $003A
CRGFLG equ $0037
CLKSEL equ $0039

SetClk8: movb #$02,SYNR ; SYNR = 2 , PLLOSC = 48 MHz
movb #$00,REFDV ; REFDV = 0
movb #$60,PLLCTL ; enable PLL

plll: brclr CRGFLG,#$08,plll ;wait until PLL locks into the
      ; target frequency
movb #$80,CLKSEL ;clock derived from PLL PLLCLK
;movb #$40,CLKSEL ;clock derived from select OSCCLK
rts
```
Real-time interrupt (RTI)

- The RTI can be used to generate a hardware interrupt periodically at a fixed rate using a periodic timer, e.g., the interrupt is generated every 100 ms.
Programming RTI

- RTI interrupt enable bit (RTIE) is bit 7 in CRGINTE
- Set this bit to enable RTI interrupts

\[ \text{bset CRGINTE, } \#80 \quad ; \text{RTIE} = 1 \text{ (enables RTI interrupts)} \]

- Interrupt flag (RTIF): RTI requests interrupt service by setting the flag RTIF in CRGFLG register. This flag will be set at a fixed rate.

- It is the programmer responsibility to clear this flag when the interrupt is served.

\[ \text{movb } \#80, \text{CRGFLG } ; \text{clear RTIF by writing 1.} \]
How it works?

- By using a counter, an interrupt is generated after counting a programmable number of clock cycles.
- If the clock is accurate the interrupt happens at fixed rate.

Pro: RTI is a high-priority interrupt (just below IRQ)

Con: Time periods are not flexible

- The period (in units of OSCCLK cycles) is \((N+1) \times 2^{(M+9)}\)

\[
\begin{align*}
N &= RTR[3:0] \\
M &= RTR[6:4]
\end{align*}
\]

M = 0 means RTI off

- The source clock for the RTI is OSCCLK
- The address of RTICTL is $003B

The possible interrupt periods (in number of oscillator clocks) are listed in Table 6.4.
For all giving times, OSCCLK = 8MHz

Table 6.4 RTI interrupt period (in units of OSCCLK cycle)

<table>
<thead>
<tr>
<th>RTR[3:0]</th>
<th>000 (off)</th>
<th>001 (2^10)</th>
<th>010 (2^11)</th>
<th>011 (2^12)</th>
<th>100 (2^13)</th>
<th>101 (2^14)</th>
<th>110 (2^15)</th>
<th>111 (2^16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 (÷1)</td>
<td>off*</td>
<td>2^10</td>
<td>2^11</td>
<td>2^12</td>
<td>2^13</td>
<td>2^14</td>
<td>2^15</td>
<td>2^16</td>
</tr>
<tr>
<td>0001 (÷2)</td>
<td>off*</td>
<td>2×2^10</td>
<td>2×2^11</td>
<td>2×2^12</td>
<td>2×2^13</td>
<td>2×2^14</td>
<td>2×2^15</td>
<td>2×2^16</td>
</tr>
<tr>
<td>0010 (÷3)</td>
<td>off*</td>
<td>3×2^10</td>
<td>3×2^11</td>
<td>3×2^12</td>
<td>3×2^13</td>
<td>3×2^14</td>
<td>3×2^15</td>
<td>3×2^16</td>
</tr>
<tr>
<td>0011 (÷4)</td>
<td>off*</td>
<td>4×2^10</td>
<td>4×2^11</td>
<td>4×2^12</td>
<td>4×2^13</td>
<td>4×2^14</td>
<td>4×2^15</td>
<td>4×2^16</td>
</tr>
<tr>
<td>0100 (÷5)</td>
<td>off*</td>
<td>5×2^10</td>
<td>5×2^11</td>
<td>5×2^12</td>
<td>5×2^13</td>
<td>5×2^14</td>
<td>5×2^15</td>
<td>5×2^16</td>
</tr>
<tr>
<td>0101 (÷6)</td>
<td>off*</td>
<td>6×2^10</td>
<td>6×2^11</td>
<td>6×2^12</td>
<td>6×2^13</td>
<td>6×2^14</td>
<td>6×2^15</td>
<td>6×2^16</td>
</tr>
<tr>
<td>0110 (÷7)</td>
<td>off*</td>
<td>7×2^10</td>
<td>7×2^11</td>
<td>7×2^12</td>
<td>7×2^13</td>
<td>7×2^14</td>
<td>7×2^15</td>
<td>7×2^16</td>
</tr>
<tr>
<td>0111 (÷8)</td>
<td>off*</td>
<td>8×2^10</td>
<td>8×2^11</td>
<td>8×2^12</td>
<td>8×2^13</td>
<td>8×2^14</td>
<td>8×2^15</td>
<td>8×2^16</td>
</tr>
<tr>
<td>1000 (÷9)</td>
<td>off*</td>
<td>9×2^10</td>
<td>9×2^11</td>
<td>9×2^12</td>
<td>9×2^13</td>
<td>9×2^14</td>
<td>9×2^15</td>
<td>9×2^16</td>
</tr>
<tr>
<td>1001 (÷10)</td>
<td>off*</td>
<td>10×2^10</td>
<td>10×2^11</td>
<td>10×2^12</td>
<td>10×2^13</td>
<td>10×2^14</td>
<td>10×2^15</td>
<td>10×2^16</td>
</tr>
<tr>
<td>1010 (÷11)</td>
<td>off*</td>
<td>11×2^10</td>
<td>11×2^11</td>
<td>11×2^12</td>
<td>11×2^13</td>
<td>11×2^14</td>
<td>11×2^15</td>
<td>11×2^16</td>
</tr>
<tr>
<td>1011 (÷12)</td>
<td>off*</td>
<td>12×2^10</td>
<td>12×2^11</td>
<td>12×2^12</td>
<td>12×2^13</td>
<td>12×2^14</td>
<td>12×2^15</td>
<td>12×2^16</td>
</tr>
<tr>
<td>1100 (÷13)</td>
<td>off*</td>
<td>13×2^10</td>
<td>13×2^11</td>
<td>13×2^12</td>
<td>13×2^13</td>
<td>13×2^14</td>
<td>13×2^15</td>
<td>13×2^16</td>
</tr>
<tr>
<td>1101 (÷14)</td>
<td>off*</td>
<td>14×2^10</td>
<td>14×2^11</td>
<td>14×2^12</td>
<td>14×2^13</td>
<td>14×2^14</td>
<td>14×2^15</td>
<td>14×2^16</td>
</tr>
<tr>
<td>1110 (÷15)</td>
<td>off*</td>
<td>15×2^10</td>
<td>15×2^11</td>
<td>15×2^12</td>
<td>15×2^13</td>
<td>15×2^14</td>
<td>15×2^15</td>
<td>15×2^16</td>
</tr>
<tr>
<td>1111 (÷16)</td>
<td>off*</td>
<td>16×2^10</td>
<td>16×2^11</td>
<td>16×2^12</td>
<td>16×2^13</td>
<td>16×2^14</td>
<td>16×2^15</td>
<td>16×2^16</td>
</tr>
</tbody>
</table>
- The value in RTICTL determines how often the periodic interrupt occurs in terms of external clock periods.

- The number in the table means the RTI occurs in every ‘the number’ clock cycles.

- Ex.

  - RTICTL = 0100 1001, M = 0100 = 4 and N = 1001 = 9 from the table an interrupts will occur every 10 x 2^13 clocks.

  - If OSCCLK=8MHz, an interrupt is generated every (10 x 2^13)/(8MHz) = 10.24 ms. That means teh RTI ISR will be called 97.66 times every one second.

- Note that if the RTICTL register is not configured, the interrupt will not occur even if the RTIE bit is set.
Example: Set up the RTI system to generate periodic timeouts at intervals of approximately 2 ms given that OSCCLK = 16MHz

\[
2 \text{ ms} = \frac{\text{Number of counts}}{16 \text{ MHz}} \quad \iff \quad \text{Number of counts} = 32,000
\]

From the table, when \( N = 0 \) and \( M = 6 \), the number of counts = 32,768 This is the closest number of counts to 32,000

The actual time interval = 2.048ms

So \( \text{RTR}[3:0] = 0, \text{RTR}[6:4] = 6 \)

– Code:

```
movb #01100000,RTICTL
```
Example: Program the RTI system to flash an LED at a rate of about 1 per second, i.e., turn on for one second, then turn off for 1 sec. OSCCLK = 8 MHz.

- One second is a very long time in computers.
- The slowest RTI timeout rate is 8/second
  - So we only want to take an action every 8\textsuperscript{th} timeout
- We will have to keep a count of timeouts
  - Every time we sense a timeout, we increment the counter
  - When the counter reaches 8, we reset the counter to zero and toggle the LED

Main program

- Load the RTI ISR vector into locations $\text{FFFF}:\text{FFFF}$
  - Set up PT0 for output (bit 0 in register DDRT)
  - Set up RTI rate (bits RTR6:RTR0 in register RTICTL)
  - Enable RTI interrupts (set RTIE bit in register CRGINT)
  - Initialize count to zero (count is a variable in fixed global memory)
  - Turn on interrupt system (e.g., the CLI instruction)
  - Go into an infinite loop
RTI_ISR
- Clear the flag (write 1 to the RTIF bit in CRGFLG)
- Increment count
- If count is equal to 8, toggle PT0 and clear count
- Execute the RTI instruction to return from interrupt

```
org 1000
rti_count ds.b 1

org $FFF0
dc.w RTI_ISR ;load RTI ISR vector

; main program
org 2000
movb #1,DDRT ;set up PT0 for output
movb #$7F,RTICTL ;set up slowest RTI rate
bset CRGINT,#$80 ; RTIE=1 (enables RTI interrupts)
lds #$4000 ; initialize the stack pointer
andcc #%10111111 ; enable nonmaskable interrupts
cli ; enable interrupt systems
Loop: bra Loop ; wait for interrupt
```
RTI_ISR:
    movb #$80,CRGFFLG ;clear RTIF by writing a 1 to it.
    inc rti_count
    ldaa rti_count
    cmpa #8 ;check if count =8
    bne RTI_done ;if no, we are done
    clr rti_count ;if yes clear count and
    ldaa PTT ; --- toggle PT0
    eora #1
    staa PTT
RTI_done: rti
Thank You!

Questions