CPE 390: Microprocessor Systems

Spring 2018

Lecture 10

I/O and Interrupts

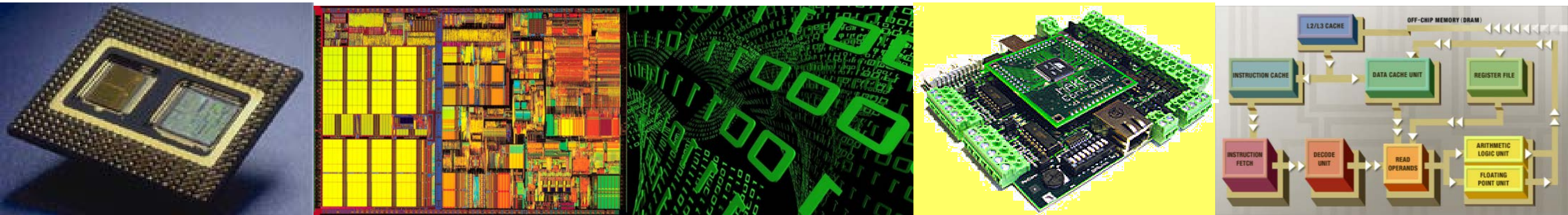
Bryan Ackland

Department of Electrical and Computer Engineering

Stevens Institute of Technology

Hoboken, NJ 07030

Adapted from HCS12/9S12 An Introduction to Software and Hardware Interfacing   Han-Way Huang, 2010



Input/Output

- So far, we have only considered programs that do computation on data stored in memory
  - For a microprocessor to perform a useful task, it must interact with the outside world
- Input/Output (I/O) devices (peripherals) are electronic components that facilitate the exchange of data between the microprocessor and its external environment

Input Devices

Keyboard
Mouse
Switches
A/D converter
Push-button
Real-time clock

Output Devices

Display
Printer
LEDs
7-segment display
D/A converter
Speaker

I/O Devices

Hard disk
Ethernet
Timers
USB port
Co-processor
Bluetooth

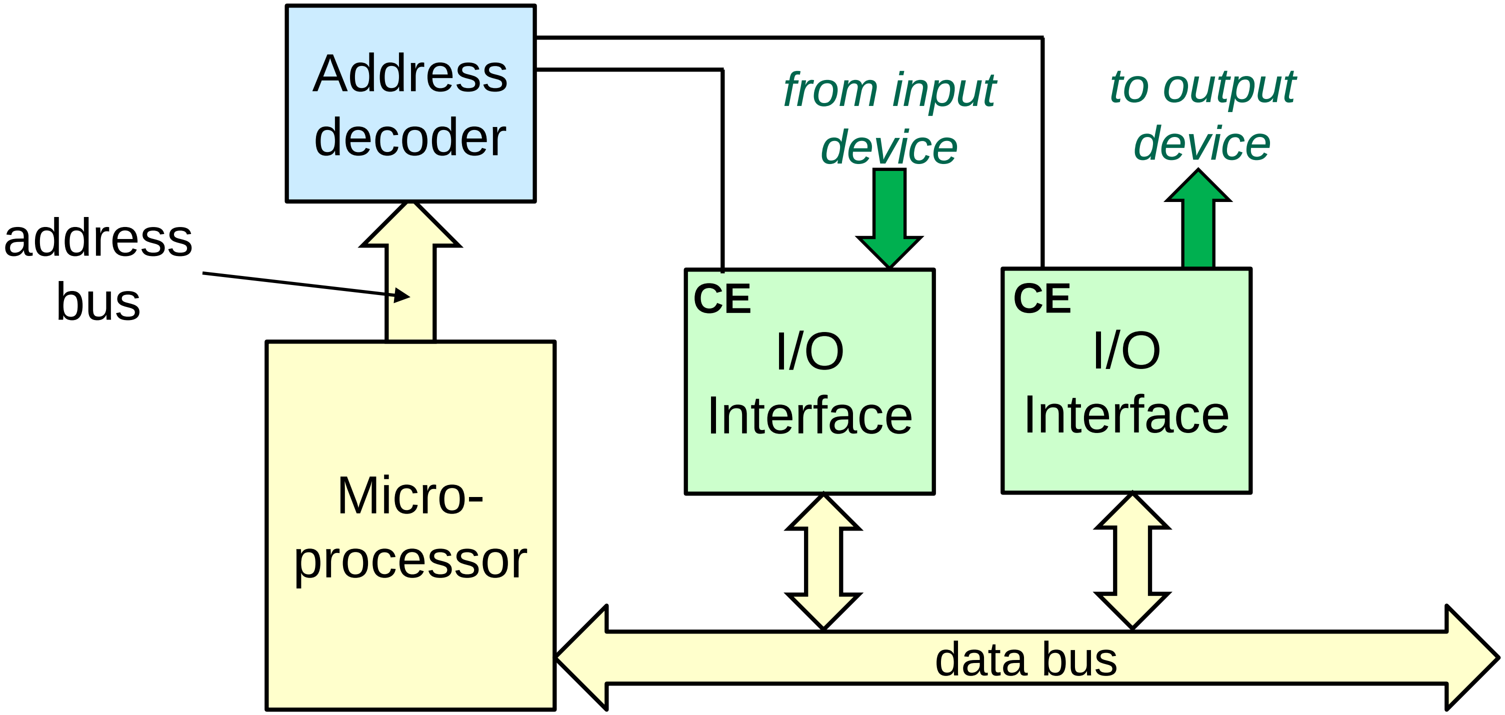
Two Common I/O Schemes

- Isolated I/O
  - Dedicated instructions for I/O operations
  - Separate address space for I/O devices
  - I/O devices do not occupy limited memory address space
  - I/O address decoder can be simplified because address space is much smaller
  - Example: Intel x86 architecture

- Memory Mapped I/O
  - I/O devices use same address & data bus as memory
  - Can use same instructions used to access memory
  - Much more flexibility in accessing I/O devices
  - More susceptible to programming errors (confusing memory and I/O addresses)
  - Example: HCS12 architecture

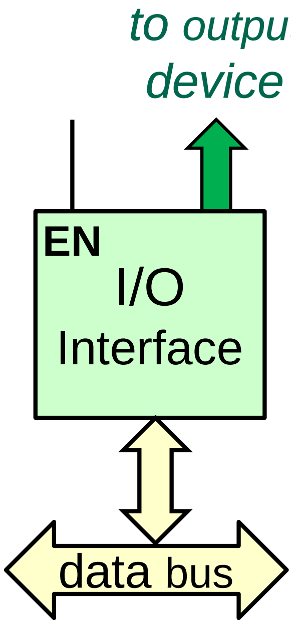
Memory Mapped I/O

- Speed and electrical characteristics of I/O devices are different from CPU
  - usually much slower than the CPU
  - cannot connect them directly (like memory)
- I/O devices usually attached to address & data buses through an I/O interface



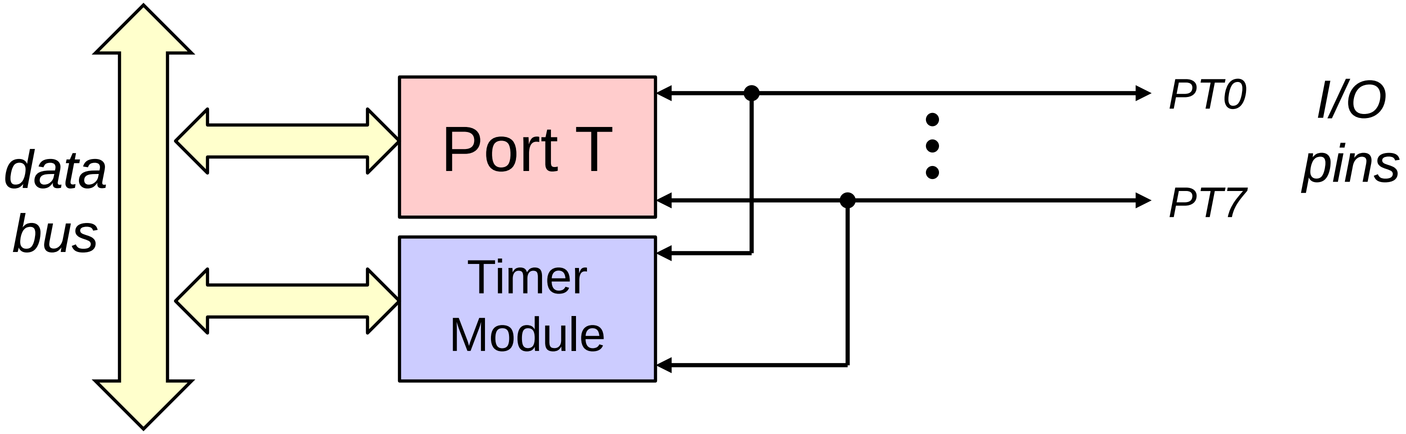
I/O Interface

- I/O interface acts as a buffer between I/O device and CPU
- Has data pins connected to microprocessor data bus and I/O port pins connected to I/O device
- Has “enable” pin which, when asserted, allows the interface to respond to a data transfer request
- Each I/O interface contains registers that provide:
  - data buffering (holds data until ready to be transferred)
  - control (e.g. allows CPU to determine data direction)
  - status (e.g. lets CPU know when data is ready)
  - each register appears to CPU as a memory location that can be read and/or written

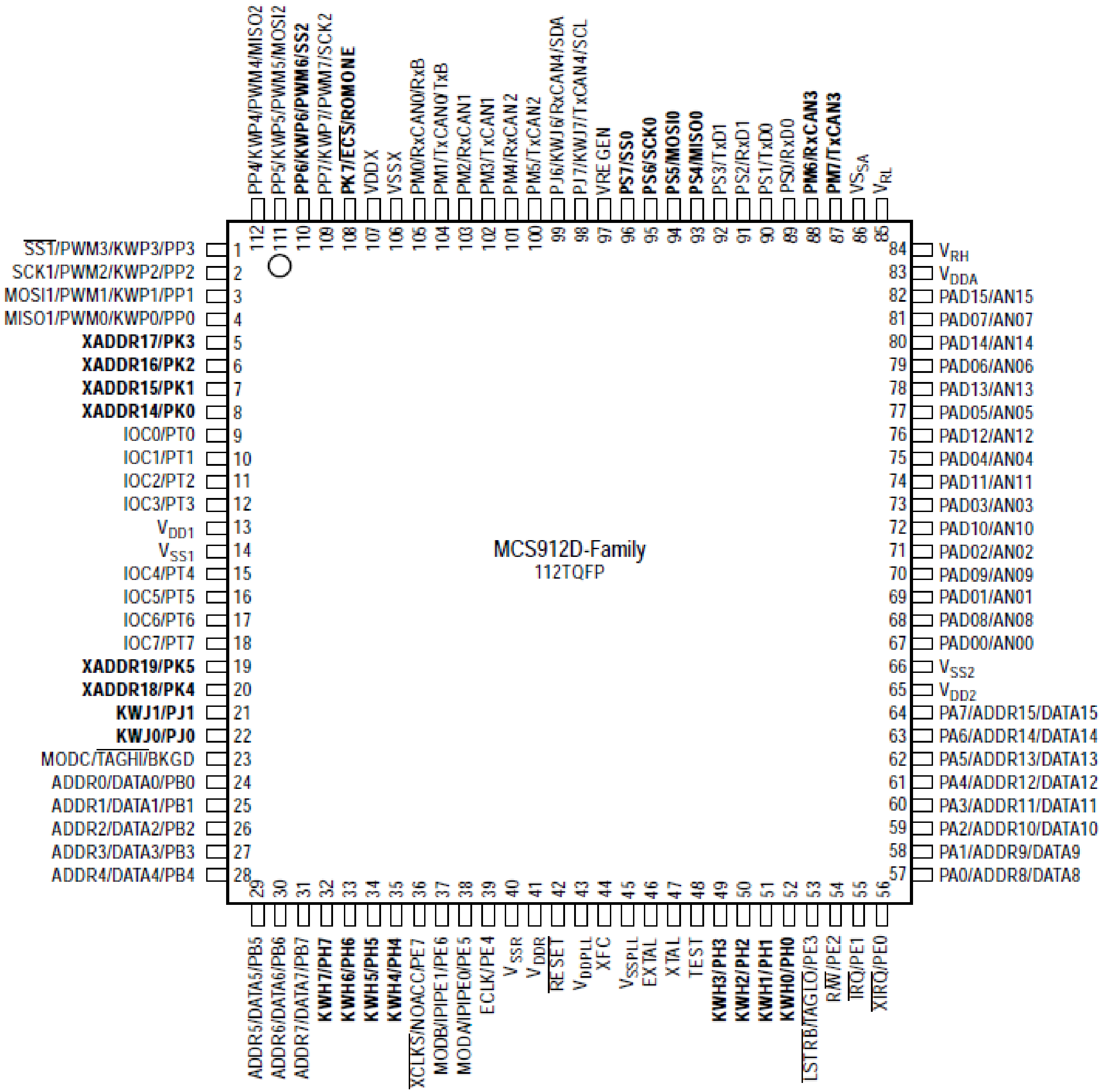


HCS12 I/O Interface

- HCS12 is more than a microprocessor – it's a microcontroller
  - CPU + on-chip memory + on-chip I/O devices and interfaces
- On HCS12, I/O interfaces are called I/O Ports
- HCS12 devices may have anywhere from:
  - 48-144 I/O signal pins
  - connected to the CPU via 3-12 on-chip I/O ports
- Most I/O pins on HCS12 are dual function
  - can function as simple parallel I/O port pin OR
  - as I/O pin of dedicated peripheral, for example:



HCS12 Pinout Example



HCS12 I/O Ports

Port Name	#pins	Pin Names	Data Register Name	Associated Peripheral Function
A	8	PA7~PA0	PTA	Address & Data Bus
B	8	PB7~PB0	PTB	Address & Data Bus
E	8	PE7~PE0	PTE	Bus control & Interrupt
H	8	PH7~PH0	PTH	Expanded address
J	8	PJ7~PJ0	PTJ	SPI Serial Interface
K	8	PK7~PK0	PTK	Expanded address
M	8	PM7~PM0	PTM	CAN and I^2C
P	8	PP7~PP0	PTP	PWM & SPI
S	8	PS7~PS0	PTS	Serial Interface
T	8	PT7~PT0	PTT	Timer
AD0,1	16	PAD15~PAD0	PORTAD0,1	A/D Converter(s)

I/O Port Addresses

- Number of registers associated with each I/O port
- Each register has a separate (memory) address
- These registers are assigned addresses (mapped) in the range $0000 ~ $03FF
- For example:
  - data register associated with Port A is assigned address $0000
  - data register associated with Port B is assigned address $0001
- Rather than having to remember all these addresses, a name is associated with each I/O register
- These names can be found in file hcs12.inc
  - included on CD in text book
- Can be made available in your assembly program by adding this line at the beginning of your .asm file:

```
INCLUDE    “hcs12.inc”
```

First Few Lines of “hcs12.inc”

```
PORTA       equu    0       ; port a = address lines a8 - a15
PTA         equ     0       ; alternate name for PORTA
PORTB       equ     1       ; port b = address lines a0 - a7
PTB         equ     1       ; alternate name for PORTB
DDRA        equ     2       ; port a direction register
DDRB        equ     3       ; port b direction register

PORTE       equ     8       ; port e = mode, irq and control signals
PTE         equ     8       ; alternate name for PORTE
DDRE        equ     9       ; port e direction register
PEAR        equ     $a      ; port e assignments
MODE        equ     $b      ; mode register
PUCR        equ     $c      ; port pull-up control register
RDRIV       equ     $d      ; port reduced drive control register
EBICTL      equ     $e      ; e stretch control

INITRM      equ     $10     ; ram location register
INITRG      equ     $11     ; register location register
INITEE      equ     $12     ; eeprom location register
MISC        equ     $13     ; miscellaneous mapping control
```

Simple I/O: Ports A and B

- Ports A and B are simple 8-bit parallel I/O ports
  - Each bit can be individually set to be an input or output
  - Two registers associated with each of these ports:

- **Data Register: PTA and PTB**
  - use normal instructions to move data to and from these registers as if they were memory locations, e.g:

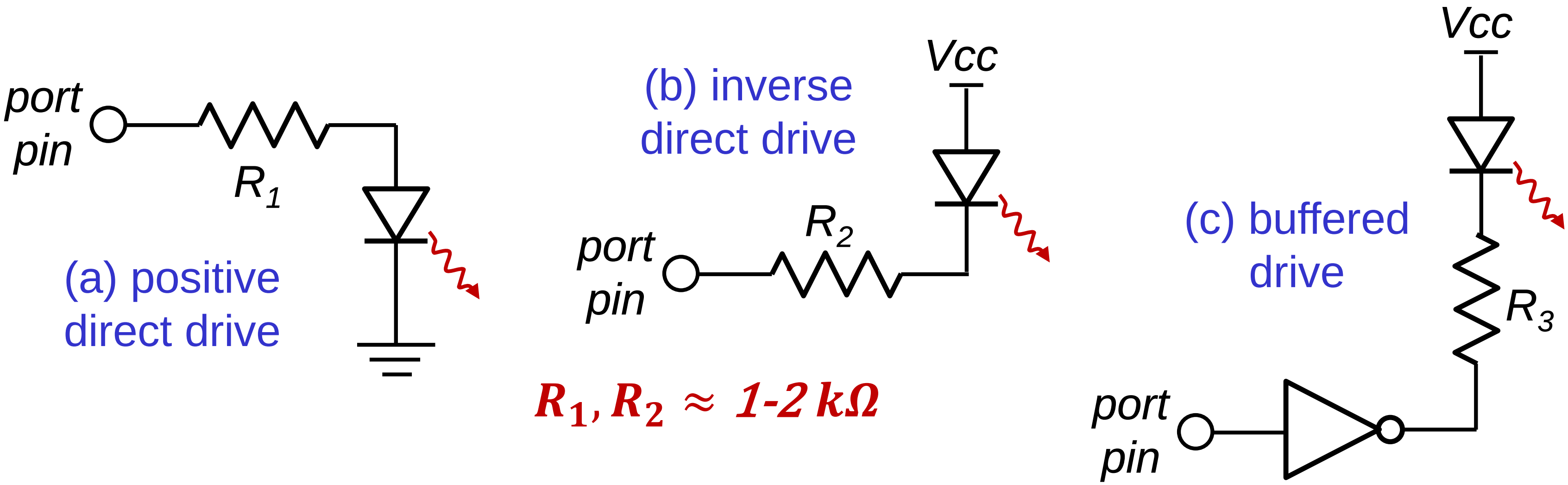
```
movb        #$35, PTA        ;outputs $35 to Port A

ldaa        PTB              ;inputs a byte from Port B into acc. A
```

- **Data Direction Register: DDRA and DDRB**
  - sets direction of each bit in data register (0 = input; 1 = output)

```
movb        #$FF, DDRA       ;configure Port A for output

movb        #$AA, DDRB       ;even pins for input, odd pins for output
```

Simple I/O: Interfacing with LEDs

- A light emitting diode (LED) is often a convenient way to display value of a single bit of information
- LED is illuminated by passing a few mA of forward current through the diode
  - typically this requires a forward bias of 1-2V
  - usually driven by a voltage source and current limiting resistor
  - port I/O pin can be used as the voltage source



(a) positive direct drive

(b) inverse direct drive

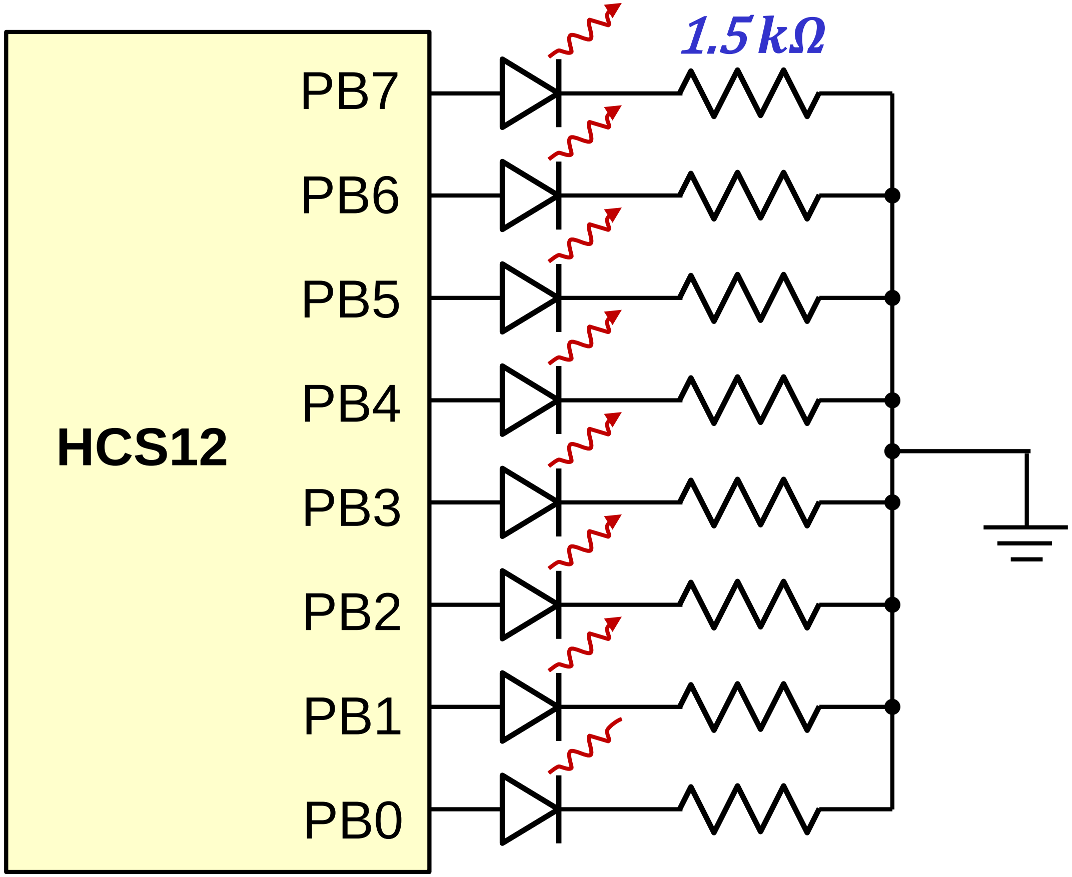
(c) buffered drive

$R_1, R_2 \approx$ *1-2 kΩ*

- Buffered drive used for higher power LEDs

Example: Bouncing LED Display

- Use the HCS12 Port B to drive eight LEDs. Light each of them half a second in turn in one direction, and then in the other direction. (Assume you have a 100ms time delay subroutine available for use)

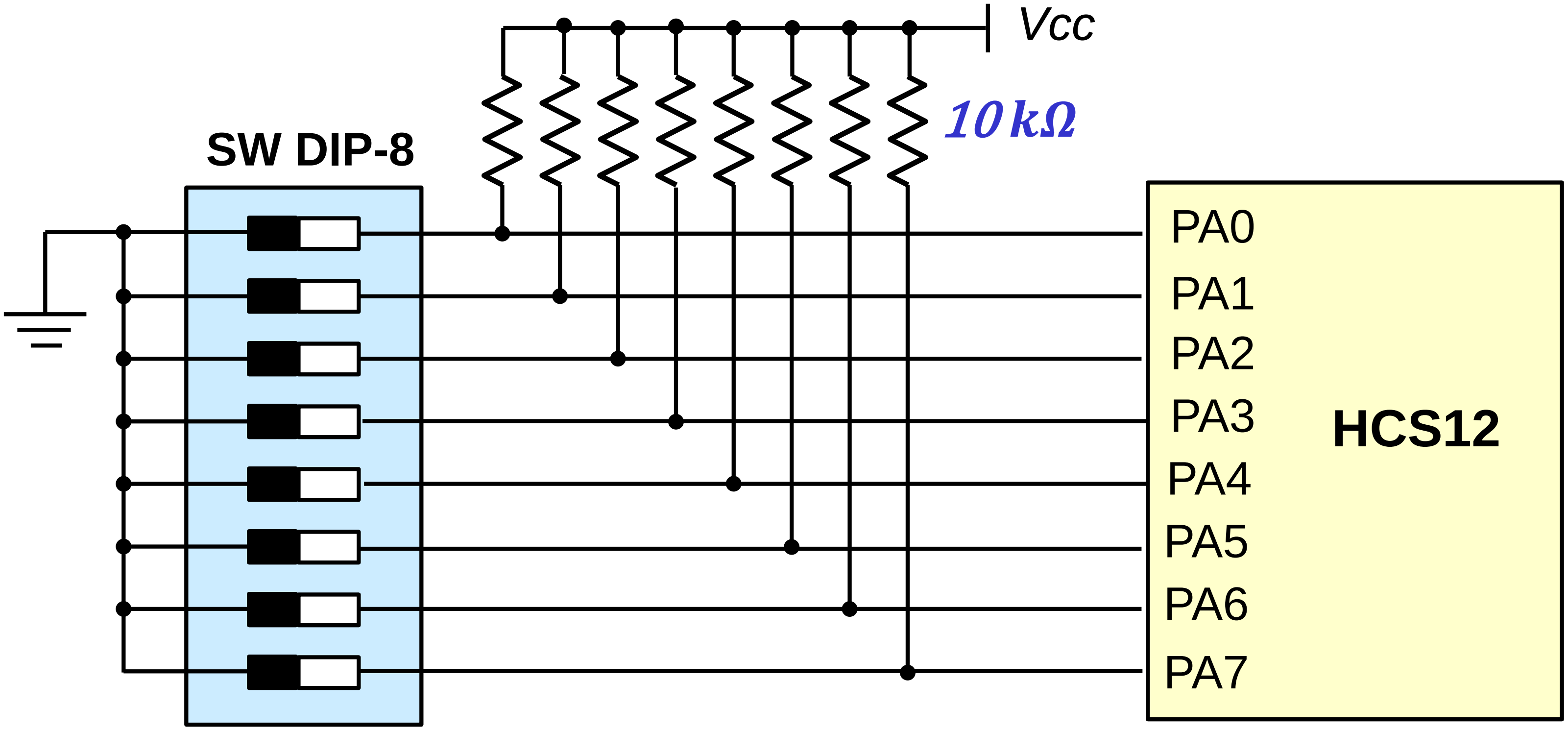


Example: Bouncing LED Display - Code

```
           include   hcs12.inc

           ORG       $800
led_tab:   DC.B      $80,$40,$20,$10,$08,$04,$02
           DC.B      $01,$02,$04,$08,$10,$20,$40

lpcnt:     DS.B      1                    ;pattern counter

           ORG       $4000
           movb      #$FF, DDRB           ; configure port B for output
forever:   movb      #14, lpcnt           ; initialize pattern count
           ldx       #led_tab             ; X is pointer to pattern
led_lp:    movb      1, x+, PTB           ; turn on one LED
           ldy       #5                   ;wait for 500 mS
           jsr       delayby100ms
           dec       lpcnt                ;end of pattern table?
           bne       led_lp
           bra       forever              ;start from beginning
```

Interfacing with DIP Switches

- Pull-up resistors hold input at ‘1’ unless connected by switch to ground (‘0’)

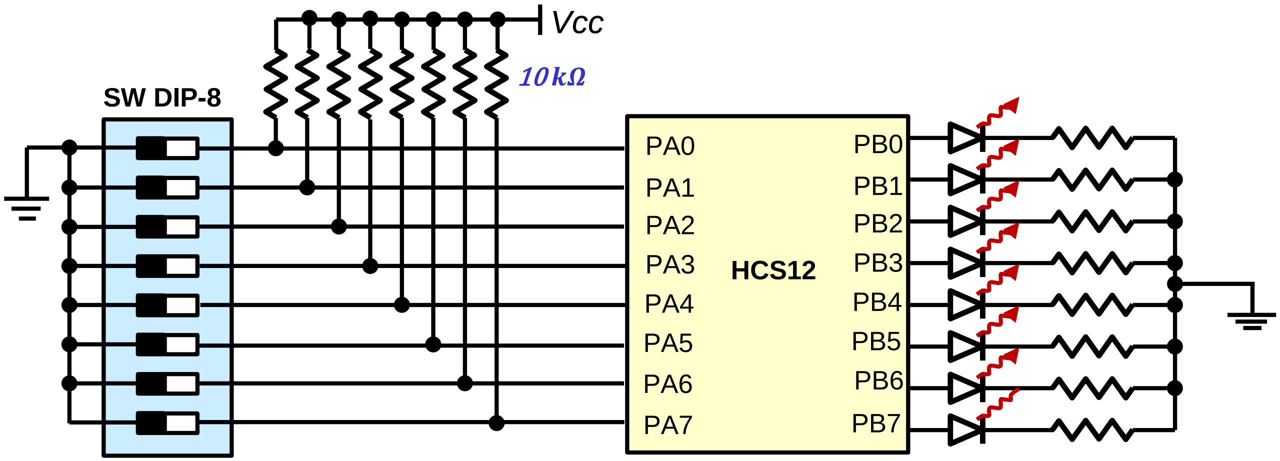


- To read data into accumulator A:

```
movb    #0, DDRA        ;configure port A for input
ldaa    PTA             ;read into accumulator A
```

Interfacing with DIP Switches

Write a program starting at address $4000 that would allow us to use the 8 switches on port A to individually control the 8 LEDs on port B



Interrupts

- **What is an interrupt?**
  - A special event that requires the CPU to stop normal program execution and perform some service related to the event
  - The source of the interrupt can be internal to the CPU (e.g. illegal op-code, divide-by-0), an on-chip peripheral (e.g. timer) or external to the chip (e.g. I/O completion, low-battery signal)
- **Why are interrupts used?**
  - Allow processor to respond to events that cannot easily be anticipated by normal program without huge amount of condition and status checking that would make the code very slow and difficult to understand/debug
- **Function of interrupts include:**
  - Coordinating I/O activities (without polling of I/O devices)
  - Performing time critical operations in a real-time system
  - Providing a graceful way to exit from errors
  - Reminding the CPU to perform routine tasks (e.g. updating real-time clock, scheduling in an operating system)

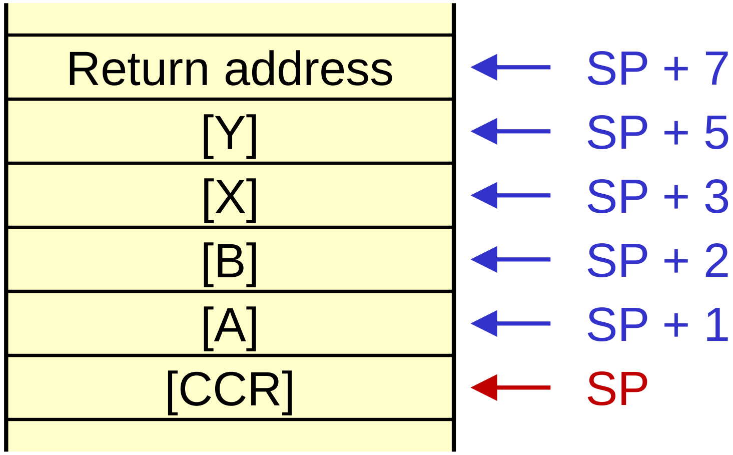
Interrupt Sequence

- **What happens when an interrupt occurs?**

1. CPU completes execution of current instruction
2. Disables interrupts (to prevent a further interrupt)
3. Saves current value of program counter on stack
4. Saves current CPU status on stack
   - includes accumulators, registers and CCR
5. Identifies source of interrupt and resolves starting address of interrupt service routine that services that source
6. Executes interrupt service routine
   - until it encounters an **rti** (return from interrupt) instruction
7. Restores CPU status from stack
8. Restores program counter from stack
   - which allows resumption of interrupted program
9. Enables interrupts

Stack Order on Interrupt

- When the CPU services an interrupt, it automatically saves all CPU registers (except SP):



Return address	SP + 7
[Y]	SP + 5
[X]	SP + 3
[B]	SP + 2
[A]	SP + 1
[CCR]	SP

- Return from interrupt **rti** instruction terminates the interrupt service routine
  - restores all CPU registers
  - continue to execute interrupted program unless there is another interrupt pending
- How can an ISR communicate with main program?

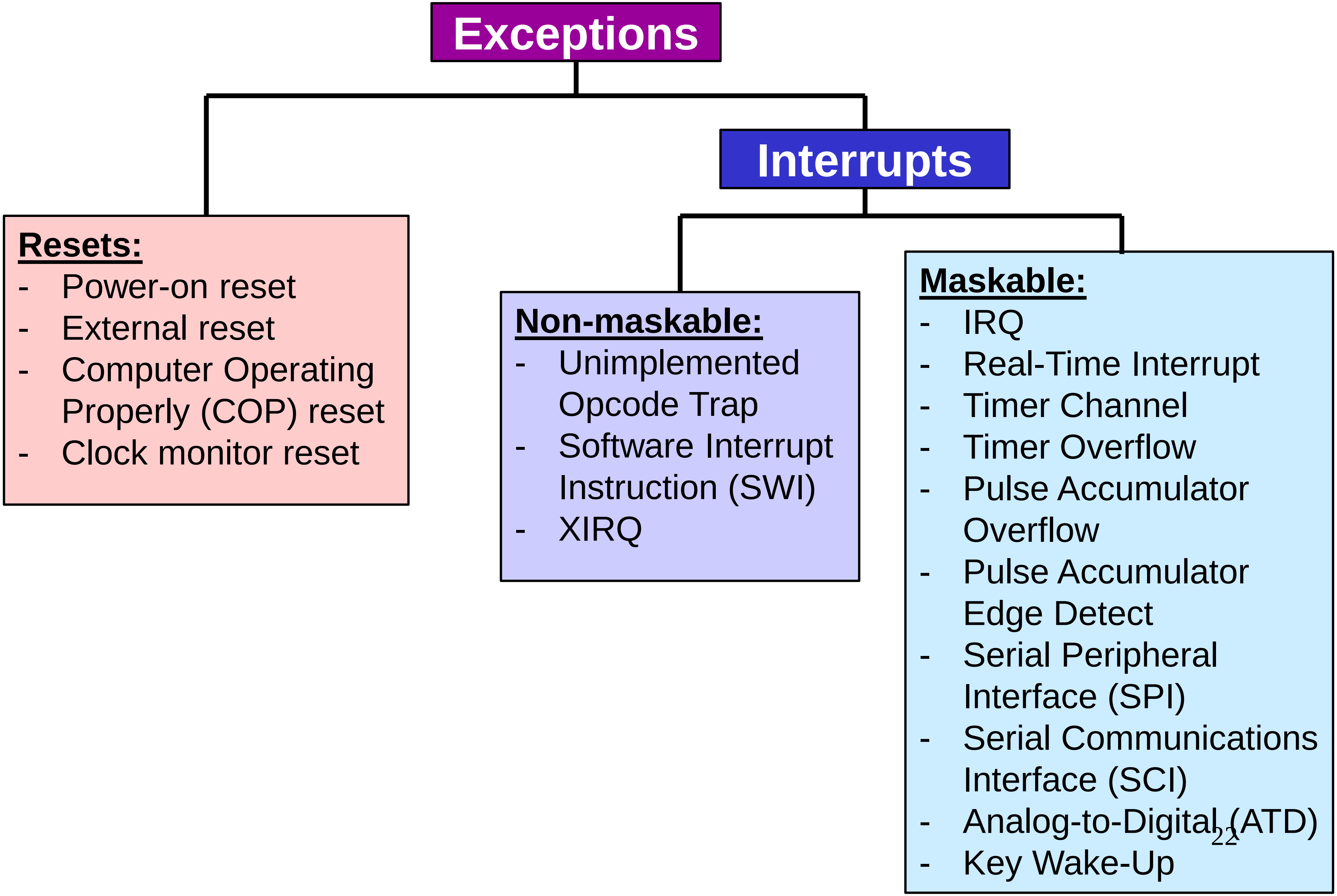
Maskable Interrupts

- A maskable interrupt is one which the normal program can choose to ignore
- A maskable interrupt source must be enabled by the software before it can interrupt the CPU
  - by setting an interrupt enable flag
- A maskable interrupt can be disabled at any time by the software
- Two levels of interrupt enabling capability:
  - global interrupt flag (in the CPU which enables/disables all maskable interrupts)
  - local interrupt flag (in the I/O interface that enables/disables a particular interrupt source)
- Examples include the $\overline{IRQ}$ pin and all on-chip peripheral function interrupts

Non-Maskable Interrupts

- A non-maskable interrupt is one which cannot be ignored (disabled) by the normal program
- There are three non-maskable interrupts:
  - external $\overline{XIRQ}$ pin
  - unimplemented op-code trap
  - software interrupt (swi) instruction
- There are also three exceptions (similar to interrupts) that are non-maskable
  - power-on reset
  - external reset (the $\overline{reset}$ pin)
  - computer operating properly (COP) reset – a watchdog timer
  - clock monitor reset
- Exceptions do not normally return to interrupted program
- Why would you want to have an interrupt be non-maskable?

HCS12 Exception Processing System



Exceptions

Interrupts

Resets:

- Power-on reset
- External reset
- Computer Operating Properly (COP) reset
- Clock monitor reset

Non-maskable:

- Unimplemented Opcode Trap
- Software Interrupt Instruction (SWI)
- XIRQ

Maskable:

- IRQ
- Real-Time Interrupt
- Timer Channel
- Timer Overflow
- Pulse Accumulator Overflow
- Pulse Accumulator Edge Detect
- Serial Peripheral Interface (SPI)
- Serial Communications Interface (SCI)
- Analog-to-Digital (ATD)
- Key Wake-Up

Interrupt Vector Table

- How does the CPU know where to find the appropriate interrupt service routine?
- Associated with each interrupt source is an interrupt vector – the starting address of the service routine
- These vectors are stored in the interrupt vector table
- Some different approaches to determining vectors:
  - Predefined vectors (no need for table) – e.g. Intel 8051
  - Allocate table to a predefined memory location - e.g. HCS12
  - Interrogate interrupting hardware for an index or pointer to service routine (e.g. Freescale 68000, Intel x86)
- HCS12 locates interrupt vector table at memory addresses $FF8A – $FFFE
  - each vector requires two bytes
  - in lab sessions, this is mapped down to $0F8A – $0FFE

Sample of Interrupt Vector Table

Vector Address	Interrupt Source	CCR Mask
$FFFE	Reset	none
$FFFC	Clock monitor	none
$FFFA	COP reset	none
$FFF8	Bad opcode	none
$FFF6	SWI	none
$FFF4	XIRQ	X
$FFF2	IRQ	I
$FFF0	Real-Time Interrupt	I
$FFEE	ECT channel 0	I
$FFEC	ECT channel 1	I
$FFEA	ECT channel 2	I
$FFE8	ECT channel 3	I
$FFE6	ECT channel 4	I

Vector Address	Interrupt Source	CCR Mask
$FFE4	ECT channel 5	I
$FFE2	ECT channel 6	I
$FFE0	ECT channel 7	I
$FFDE	ECT overflow	I
$FFDC	Pulse accA overflow	I
$FFDA	Pulse accA edge	I
$FFD8	SPI0	I
$FFD6	SCI0	I
$FFD4	SCI1	I
$FFD2	ATD0	I
$FFD0	ATD1	I
$FFCE	Port J	I
$FFCC	Port H	I

see Table 6.1 in Textbook for complete table

Interrupt Priority

- What happens if two sources interrupt at the same time? Who should get service?
- Interrupts are prioritized according to their vector table address.
  - Higher the vector table address, the higher the priority
  - Highest priority interrupt is the hardware reset at $FFFE
- Can raise any one of the maskable interrupts to the highest priority (of the maskable interrupts) by setting the HPRIO register to the least significant 8-bits of that interrupt's vector table address.
  - relative priorities of other sources remain the same
- If there are multiple interrupts pending:
  - CPU first services highest priority interrupt
  - Once this has been serviced, the **rti** instruction will then transfer control to next highest priority interrupt

Interrupt Programming

- Three steps in adding interrupt service to a program:
  1. *Write the interrupt service routine.*
     - Can be thought of as a (asynchronous) subroutine call.
     - Keep as short as possible to minimize overhead – only do those tasks that have to be done immediately.
     - May or may not return to interrupted program (with **rti** instruction) depending on nature of interrupt
  2. *Initialize interrupt vector table*
     - Load starting address of interrupt service routine into appropriate vector table address
  3. *Enable the interrupts at run-time*

- An interrupt incurs significant overhead in saving and restoring the CPU registers & accumulators
  - Minimum overhead is 17 to 20 E-clock cycles. With an 8 MHz E-clock, this is a little over 2 μs.

IRQ Interrupt

- $\boldsymbol{\overline{IRQ}}$ is an external maskable interrupt on HCS12
- Maskable interrupts (i.e. $\overline{IRQ}$ and all peripheral function interrupts) are globally enabled by clearing the I-bit in the CCR

7	6	5	4	3	2	1	0
S	X	H	I	N	Z	V	C

- The I-bit is set (i.e. interrupts are disabled) at system reset
- The I-bit can be cleared (i.e. interrupts enabled) or set (interrupts disabled) at any time by user software
- When an interrupt occurs, the I-bit is automatically set by the processor to prevent further interrupts
  - The I-bit is then normally cleared when the **rti** instruction restores the CCR

Interrupt Control Register

- The operation of the $\overline{\boldsymbol{IRQ}}$ interrupt is further controlled by the Interrupt Control Register IRQCR. It has two active control bits:

7	6	5	4	3	2	1	0
IRQE	IRQEN	0	0	0	0	0	0

IRQEN: IRQ enable bit

1 = IRQ pin interrupt enabled *(reset condition)*

0 = IRQ pin interrupt disabled

This is the local interrupt enable for IRQ. It does not affect other maskable interrupts

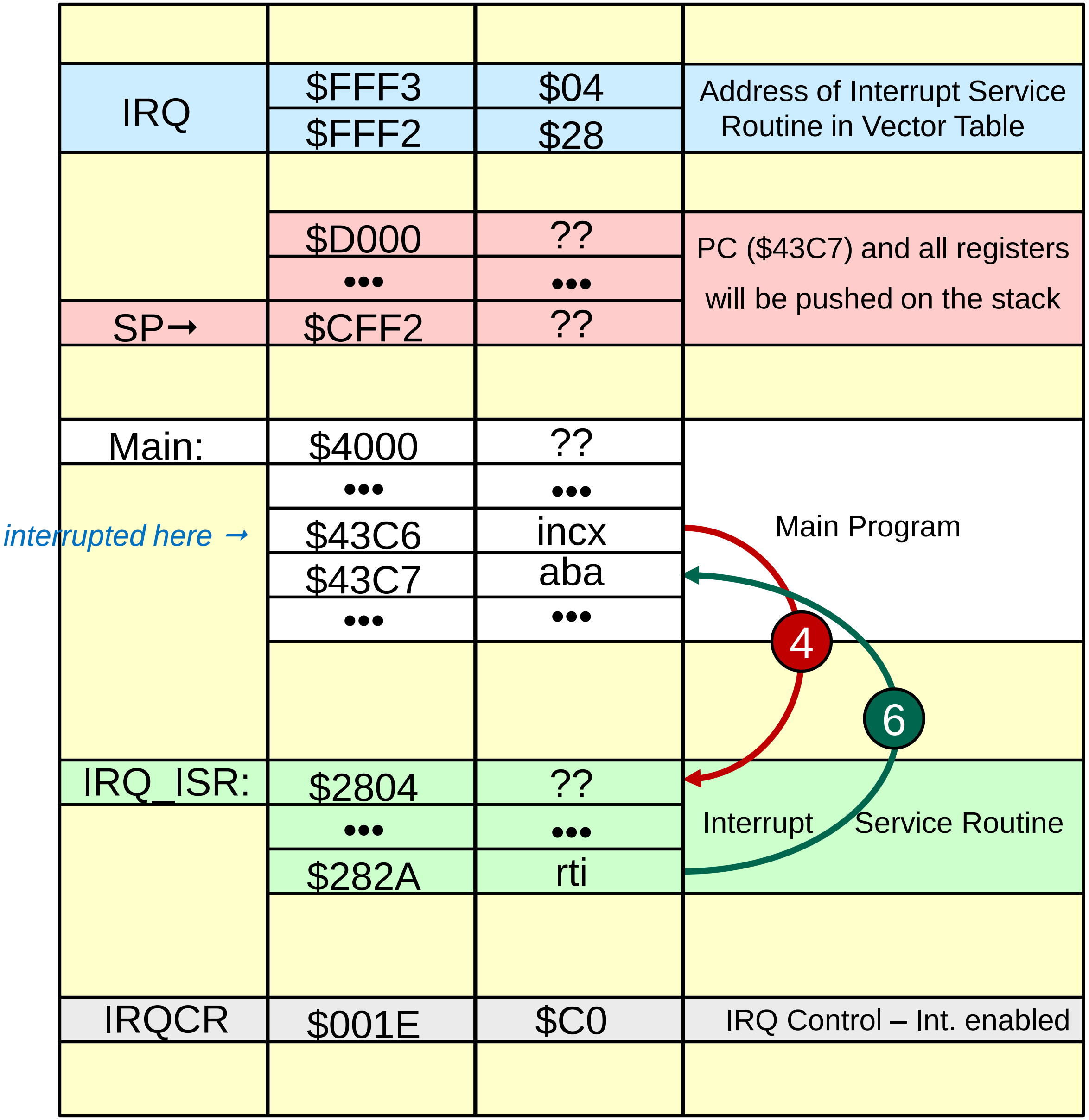
IRQE: IRQ edge-sensitive bit

1 = IRQ responds only to falling edge

0 = IRQ responds to low (‘0’) level *(reset condition)*

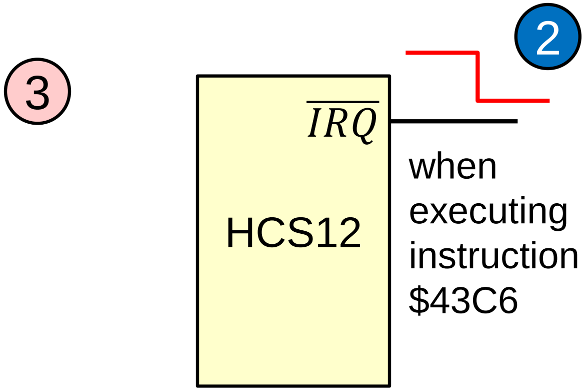
- ➢ *Level sensitive IRQ allows multiple sources to be connected to this pin (IRQ pin in internally pulled high), but need to make sure that signal is de-asserted before returning from interrupt*
- ➢ *Edge sensitive IRQ means we don’t have to worry about duration of IRQ pulse but makes the pin more noise sensitive*

Picturing an Interrupt



IRQ	$FFF3	$04	Address of Interrupt Service Routine in Vector Table
	$FFF2	$28	
	$D000	??	PC ($43C7) and all registers will be pushed on the stack
	•••	•••	
SP→	$CFF2	??	
Main:	$4000	??	Main Program
	•••	•••	
interrupted here →	$43C6	incx	
	$43C7	aba	
	•••	•••	
IRQ_ISR:	$2804	??	Interrupt Service Routine
	•••	•••	
	$282A	rti	
IRQCR	$001E	$C0	IRQ Control – Int. enabled

1 Interrupts have been enabled by clearing I bit in CCR (global) and setting IRQCR to $C0 (local)



when executing instruction at $43C6

4 CPU transfers control to address of interrupt service routine stored in interrupt table

5 rti instruction restores registers by pulling their values off the stack

6 rti instruction loads PC with return address from stack

Example: Interrupt Driven Bouncing LEDs

- Suppose we have a 2 Hz digital square wave connected to pin $\overline{IRQ}$ of an HCS12 microcontroller. Rewrite the “Bouncing LEDs” code to use this waveform to time the updating of the LED pattern, rather than using a software time delay.

```
          include   hcs12.inc
          ORG       $800
pindex:   DS.B      1                           ;pattern index
led_tab:  DC.B      $80,$40,$20,$10,$08,$04,$02
          DC.B      $01,$02,$04,$08,$10,$20,$40

          ORG       $4000
          movw      #IRQIS, UserIRQ             ; set up interrupt vector
          movb      #$FF, DDRB                  ; configure port B for output
          movb      #0, pindex                  ; initialize pattern index
          movb      led_tab, PTB                ; output initial pattern
          movb      #$C0, IRQCR                 ; enable IRQ and edge triggering
          cli                                   ; enable maskable interrupts
forever:  bra       forever                     ; wait for interrupt
```

Example: Interrupt Service Routine

```
;Interrupt Service Routine

IRQIS:    ldab    pindex          ; get current index
          incb                    ; increment index
          cmpb    #14             ; are we past end of table?
          bne     pdxok
          clrb                    ; if so, reset index
pdxok:    stab    pindex          ; save updated index
          ldx     #led_tab        ; X is table address
          movb    b, x, PTB       ; send indexed pattern to LEDs
          rti                     ; return to main program
```

XIRQ Interrupt

- $\boldsymbol{\overline{XIRQ}}$ is an external level sensitive non-maskable interrupt
- The $\overline{XIRQ}$ is enabled by clearing the X-bit in the CCR

7	6	5	4	3	2	1	0
S	X	H	I	N	Z	V	C

- The X-bit is set (i.e. interrupt is disabled) at system reset
- The X-bit can be cleared (i.e. interrupt enabled) at any time by user software
- Once cleared, the X-bit <u>cannot be set by user software</u>
  - in other words, once enabled, this interrupt cannot be disabled
- When an $\overline{XIRQ}$ interrupt occurs, both the X-bit and I-bit are automatically set by the processor to prevent further interrupts
  - They are normally restored to their previous values when the **rti** instruction restores the CCR
  - Note that the X-bit is not set when a maskable interrupt occurs
  - The $\overline{XIRQ}$ pin can interrupt a maskable interrupt service routine

Other Non-maskable Interrupts

- **Unimplemented Opcode Trap**
  - An opcode uses 1 or 2 bytes of memory – 511 possible opcodes
  - Of these, only 309 of these are used
  - If PC encounters an unused code, interrupt occurs through vector address $FFF8:$FFF9
  - Usually does not make sense to return from such an interrupt
- **Software Interrupt Instruction**
  - This are commonly used by operating system to implement a system call or debug monitor to insert a breakpoint into user code
  - In case of breakpoint, software interrupt causes transfer of control to monitor program which allows a user to examine memory and registers
  - Can continue interrupted program by executing an **rti** instruction

Computer Operating Properly (COP)

- COP is a free-running watch-dog timer
  - checks that user programming is still running correctly
  - will time-out and generate interrupt if not regularly re-armed
  - used to automatically re-start code following software crash
- When COP is enabled, user program must write the sequence {$55, $AA} into the ARMCOP register on a regular basis
  - Failure to write the sequence within the specified time or writing any other code to ARMCOP will generate COP interrupt which resets CPU
- Time-out period can be set using register COPCTL
  - anywhere from 2^{14} to 2^{24} crystal oscillator clock cycles
  - 4 msec to 4 sec with a 4MHz crystal
  - COPCTL register described in Fig. 6.17 of text

Real-Time Interrupt

- The Real-Time Interrupt (RTI) is an on-chip peripheral that can be programmed to interrupt CPU at regular intervals.
  - used to periodically update I/O, time multiplex displays, maintain real-time clock, activate process scheduler etc.
  - uses three registers to set up and control interrupt sequence

- **CRG Interrupt Enable Register (CRGINT)**

7	*6*	*5*	*4*	*3*	*2*	*1*	*0*
RTIE	0	0	LCKIE	0	0	SCMIE	0

RTIE: Real-time clock interrupt enable (‘0’: disabled, ‘1’: enabled)

Note: CRGINT and CRGFLG are registers belonging to Clock and Reset Generation Unit (CRG). They control a number of clock and PLL features – we are only concerned with RTI functionality

Real-Time Interrupt Registers

- **CRG Flag Register (CRGFLG)**

7	6	5	4	3	2	1	0
RTIF	PORF	0	LOCKIF	LOCK	TRACK	SCMIF	SCM

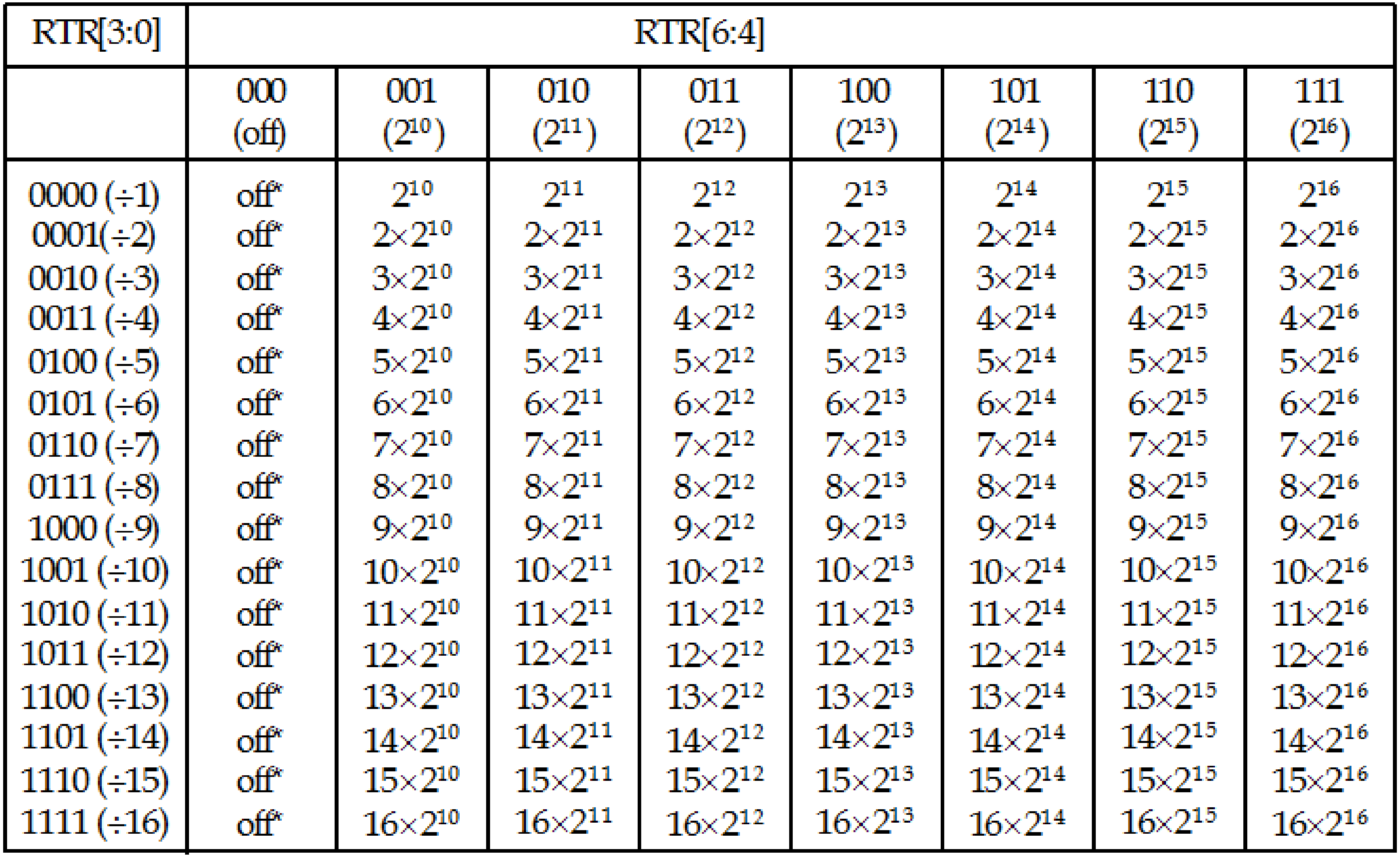
RTIF: Indicates RTI time-out has occurred.
When RTIE =‘1’, an interrupt will occur when this flag is set.
RTIF must be reset in ISR by writing a ‘1’ to it.

- **RTI Control Register (RTICTL)**

7	6	5	4	3	2	1	0
0	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0

RTR[6:0]: selects interrupt period anywhere from 2^{10} to 2^{20} crystal oscillator clock periods

RTI Period (in units of OSC period)



RTR[3:0]	RTR[6:4]							
	000 (off)	001 (2^{10})	010 (2^{11})	011 (2^{12})	100 (2^{13})	101 (2^{14})	110 (2^{15})	111 (2^{16})
0000 (÷1)	off*	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001(÷2)	off*	2×2^{10}	2×2^{11}	2×2^{12}	2×2^{13}	2×2^{14}	2×2^{15}	2×2^{16}
0010 (÷3)	off*	3×2^{10}	3×2^{11}	3×2^{12}	3×2^{13}	3×2^{14}	3×2^{15}	3×2^{16}
0011 (÷4)	off*	4×2^{10}	4×2^{11}	4×2^{12}	4×2^{13}	4×2^{14}	4×2^{15}	4×2^{16}
0100 (÷5)	off*	5×2^{10}	5×2^{11}	5×2^{12}	5×2^{13}	5×2^{14}	5×2^{15}	5×2^{16}
0101 (÷6)	off*	6×2^{10}	6×2^{11}	6×2^{12}	6×2^{13}	6×2^{14}	6×2^{15}	6×2^{16}
0110 (÷7)	off*	7×2^{10}	7×2^{11}	7×2^{12}	7×2^{13}	7×2^{14}	7×2^{15}	7×2^{16}
0111 (÷8)	off*	8×2^{10}	8×2^{11}	8×2^{12}	8×2^{13}	8×2^{14}	8×2^{15}	8×2^{16}
1000 (÷9)	off*	9×2^{10}	9×2^{11}	9×2^{12}	9×2^{13}	9×2^{14}	9×2^{15}	9×2^{16}
1001 (÷10)	off*	10×2^{10}	10×2^{11}	10×2^{12}	10×2^{13}	10×2^{14}	10×2^{15}	10×2^{16}
1010 (÷11)	off*	11×2^{10}	11×2^{11}	11×2^{12}	11×2^{13}	11×2^{14}	11×2^{15}	11×2^{16}
1011 (÷12)	off*	12×2^{10}	12×2^{11}	12×2^{12}	12×2^{13}	12×2^{14}	12×2^{15}	12×2^{16}
1100 (÷13)	off*	13×2^{10}	13×2^{11}	13×2^{12}	13×2^{13}	13×2^{14}	13×2^{15}	13×2^{16}
1101 (÷14)	off*	14×2^{10}	14×2^{11}	14×2^{12}	14×2^{13}	14×2^{14}	14×2^{15}	14×2^{16}
1110 (÷15)	off*	15×2^{10}	15×2^{11}	15×2^{12}	15×2^{13}	15×2^{14}	15×2^{15}	15×2^{16}
1111 (÷16)	off*	16×2^{10}	16×2^{11}	16×2^{12}	16×2^{13}	16×2^{14}	16×2^{15}	16×2^{16}

Example: Real Time Clock

- Use the RTI facility to maintain a 24-hour real-time clock. Time should be maintained in three locations *hours*, *mins* and *secs* at locations $6000, $6001 and $6002 respectively. Assume crystal oscillator is 4.096 MHz

```
          include   hcs12.inc

          ORG       $6000
hours:    DS.B      1                   ; global time variables
mins:     DS.B      1
secs:     DS.B      1
RTC_cnt:  DS.B      1                   ; count 10ms interrupts

          ORG       $4000
          movw      #rtisr, UserRTI     ; set up interrupt vector
          movb      #$39, RTICTL        ; set RTI timeout to 40,960 cycles
          movb      #$80, CRGINT        ; enable rti (local) interrupt
          cli                           ; enable maskable interrupts
          bra       stuff               ; go do useful work
```

Example: Real Time Clock ISR

```
;Interrupt Service Routine

rtisr:     movb     #$80, CRGFLG     ; clear RTI interrupt flag
           inc      RTC_cnt          ; increment index
           ldaa     #100
           cmpa     RTC_cnt          ; check for 100 x 10ms
           beq      up_secs
           rti
up_secs:   clr      RTC_cnt
           inc      secs             ; increments secs
           ldaa     #60
           cmpa     secs             ; check for 60 secs
           beq      up_mins
           rti
```

Example: Real Time Clock ISR (cont.)

```
up_mins:   clr     secs
           inc     mins
           cmpa    mins            ; check for 60 mins
           beq     up_hrs
           rti
up_hrs:    clr     mins
           inc     hours
           ldaa    #24
           cmpa    hours           ; check for 24 hours
           bne     done
           clr     hours
done:      rti
```