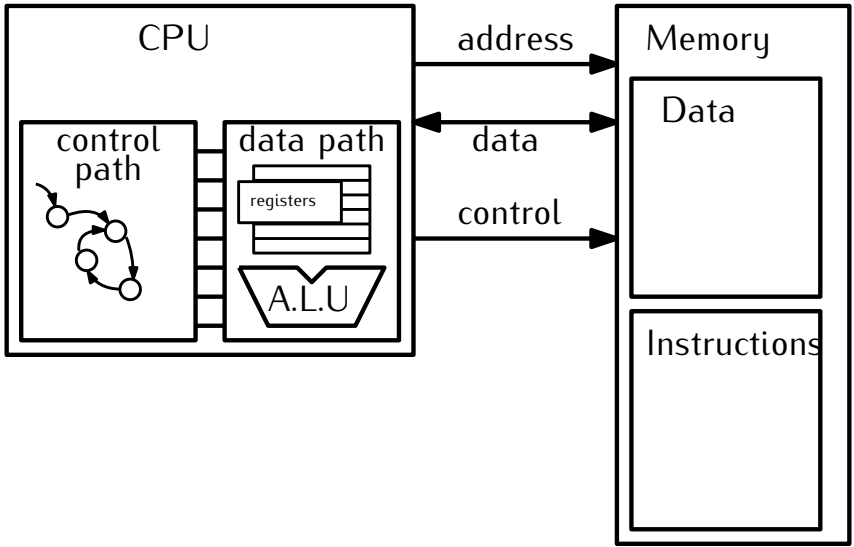
Input/Output - MMIO - Interrupts
–Computer Organization–

Lionel Morel

Computer Science and Information Technologies - INSA Lyon

Fall-Winter 2024-25

Von Neumann Architecture (from last time)



msp430 - The ABI - Instruction Set

Mnemonic		Description	Operation	V	N	Z	C
ADC(.B)	dst	Add C to destination	dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Add source to destination	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Add source and C to destination	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	AND source and destination	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Clear bits in destination	.not.src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Set bits in destination	src .or. dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bits in destination	src .and. dst	0	*	*	*
BR	dst	Branch to destination	dst → PC	–	–	–	–
CALL	dst	Call destination	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)	dst	Clear destination	0 → dst	–	–	–	–
CLRC		Clear C	0 → C	–	–	–	0
CLRN		Clear N	0 → N	–	0	–	–
CLRZ		Clear Z	0 → Z	–	–	0	–
CMP(.B)	src,dst	Compare source and destination	dst – src	*	*	*	*
DADC(.B)	dst	Add C decimally to destination	dst + C → dst (decimally)	*	*	*	*
DADD(.B)	src,dst	Add source and C decimally to dst.	src + dst + C → dst (decimally)	*	*	*	*
DEC(.B)	dst	Decrement destination	dst – 1 → dst	*	*	*	*
DECD(.B)	dst	Double-decrement destination	dst – 2 → dst	*	*	*	*
DINT		Disable interrupts	0 → GIE	–	–	–	–
EINT		Enable interrupts	1 → GIE	–	–	–	–
INC(.B)	dst	Increment destination	dst +1 → dst	*	*	*	*

Example of ASM programs - micro-machine vs msp430

Micro-machine

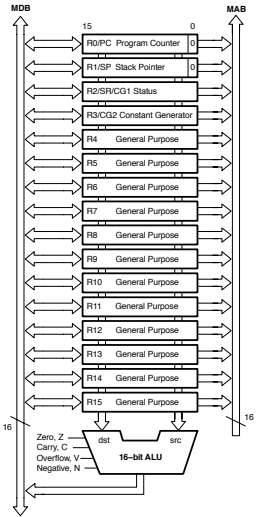
```
max: *100 -> A
     *101 -> B
     B-A ?
     JR +2 IFN
     B    -> A
     A    -> *102
```

msp430

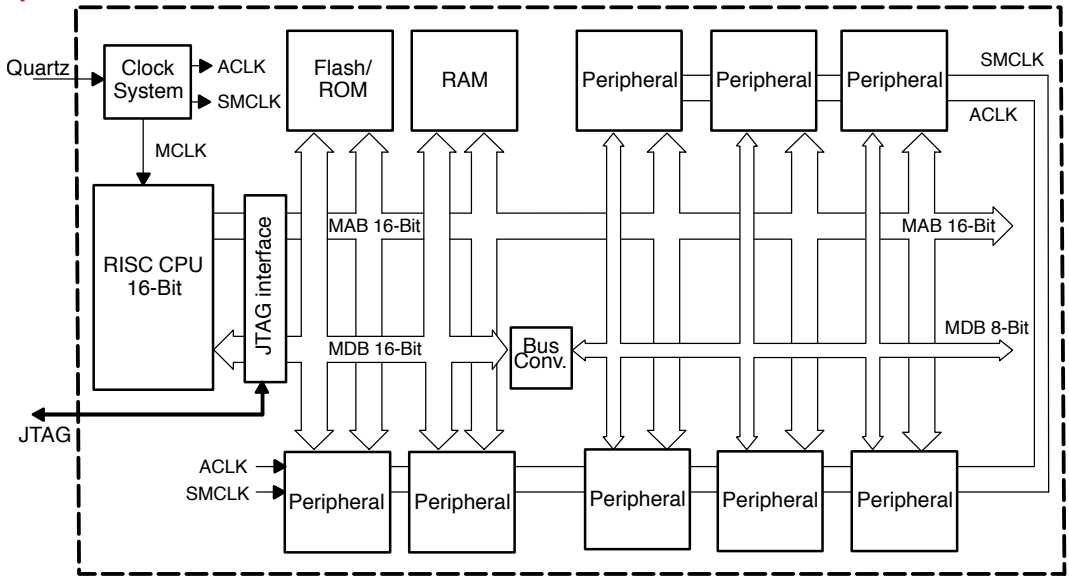
```
 .section .init9

 main:
      mov.b &0x1000, r4
      mov.b &0x1001, r5
      cmp r5, r4
      jle end
      mov.b r5, r4
end:
      mov.b r4, &&0x1001
```

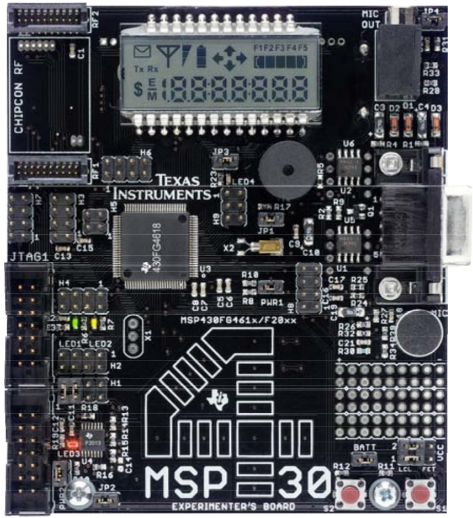
msp430 - The CPU



msp430 - Architecture



msp430 - Experimental Platform



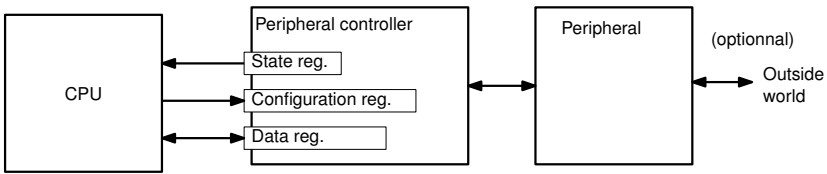
Mechanisms

- I/O Controller
- Memory-Mapped I/O
- Polling
- Interrupts

Input/Output as seen from the CPU

A peripheral is seen as a set of **registers** that can be used to **exchange information between CPU and peripheral**

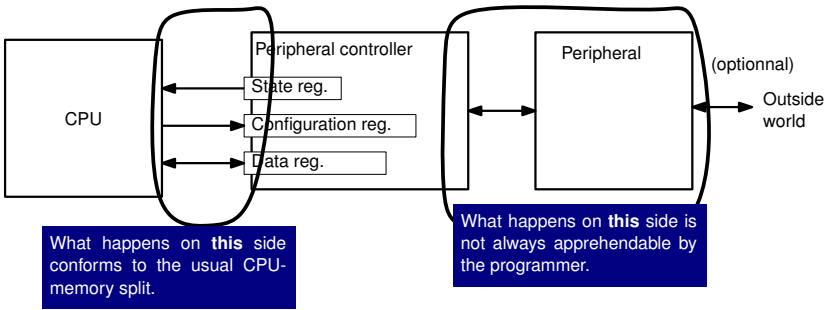
- State registers (Read-only)
- Control registers (Write-only)
- Data registers (Read-Write)



Input/Output as seen from the CPU

A peripheral is seen as a set of **registers** that can be used to **exchange information between CPU and peripheral**

- State registers (Read-only)
- Control registers (Write-only)
- Data registers (Read-Write)



msp430 - I/O register example - Timer_A (1/4)

Consider the Timer_A peripheral:

15.1 Timer_A Introduction

Timer_A is a 16-bit timer/counter with three or five capture/compare registers. Timer_A can support multiple capture/compares, PWM outputs, and interval timing. Timer_A also has extensive interrupt capabilities. Interrupts may be generated from the counter on overflow conditions and from each of the capture/compare registers.

Timer_A features include:

- Asynchronous 16-bit timer/counter with four operating modes
- Selectable and configurable clock source
- Three or five configurable capture/compare registers
- Configurable outputs with PWM capability
- Asynchronous input and output latching
- Interrupt vector register for fast decoding of all Timer_A interrupts

The block diagram of Timer_A is shown in Figure 15–1.

msp430 - I/O registers example - Timer_A (2/4)

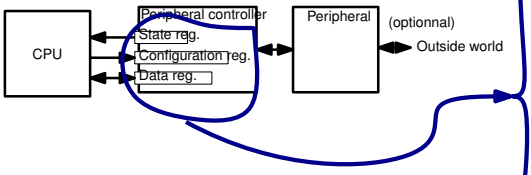


Table 15–3.Timer_A3 Registers

Register	Short Form	Register Type	Address	Initial State
Timer_A control Timer0_A3 Control	TACTL/ TA0CTL	Read/write	0160h	Reset with POR
Timer_A counter Timer0_A3 counter	TAR/ TA0R	Read/write	0170h	Reset with POR
Timer_A capture/compare control 0 Timer0_A3 capture/compare control 0	TACCTL0/ TA0CCTL	Read/write	0162h	Reset with POR
Timer_A capture/compare 0 Timer0_A3 capture/compare 0	TACCR0/ TA0CCR0	Read/write	0172h	Reset with POR
Timer_A capture/compare control 1 Timer0_A3 capture/compare control 1	TACCTL1/ TA0CCTL1	Read/write	0164h	Reset with POR
Timer_A capture/compare 1 Timer0_A3 capture/compare 1	TACCR1/ TA0CCR1	Read/write	0174h	Reset with POR
Timer_A capture/compare control 2 Timer0_A3 capture/compare control 2	TACCTL2/ TA0CCTL2	Read/write	0166h	Reset with POR
Timer_A capture/compare 2 Timer0_A3 capture/compare 2	TACCR2/ TA0CCR2	Read/write	0176h	Reset with POR
Timer_A interrupt vector Timer0_A3 interrupt vector	TAIV/ TA0IV	Read only	012Eh	Reset with POR

msp430 - I/O registers example - Timer_A (3/4)

For each “register” associated to a peripheral, the documentation tells you:

- its size
- the size and meaning of each bitfield
- their initial value (at boot)
- if you can read and/or write them

TACTL, Timer_A Control Register

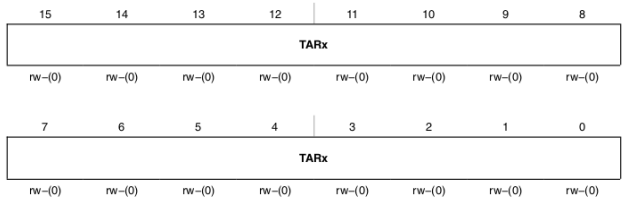
15	14	13	12	11	10	9	8
Unused						TASSELx	
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)

7	6	5	4	3	2	1	0
IDx		MCx		Unused	TACLR	TAIE	TAIFG
rw–(0)	rw–(0)	rw–(0)	rw–(0)	rw–(0)	w–(0)	rw–(0)	rw–(0)

Unused	Bits 15-10	Unused
TASSELx	Bits 9-8	Timer_A clock source select 00 TACLK 01 ACLK 10 SMCLK 11 Inverted TACLK
IDx	Bits 7-6	Input divider. These bits select the divider for the input clock. 00 /1 01 /2 10 /4 11 /8
MCx	Bits 5-4	Mode control. Setting MCx = 00h when Timer_A is not in use conserves power. 00 Stop mode: the timer is halted 01 Up mode: the timer counts up to TACCR0 10 Continuous mode: the timer counts up to 0FFFFh 11 Up/down mode: the timer counts up to TACCR0 then down to 0000h
Unused	Bit 3	Unused
TACLR	Bit 2	Timer_A clear. Setting this bit resets TAR, the clock divider, and the count direction. The TACLR bit is automatically reset and is always read as zero.
TAIE	Bit 1	Timer_A interrupt enable. This bit enables the TAIFG interrupt request. 0 Interrupt disabled 1 Interrupt enabled
TAIFG	Bit 0	Timer_A interrupt flag 0 No interrupt pending 1 Interrupt pending

msp430 - I/O registers example - Timer_A (4/4)

TAR, Timer_A Register



TARx	Bits 15-0	Timer_A register. The TAR register is the count of Timer_A.

I/O registers

- Hey!!! But ...
- ... these are **no CPU registers** ....
- So we can't write:

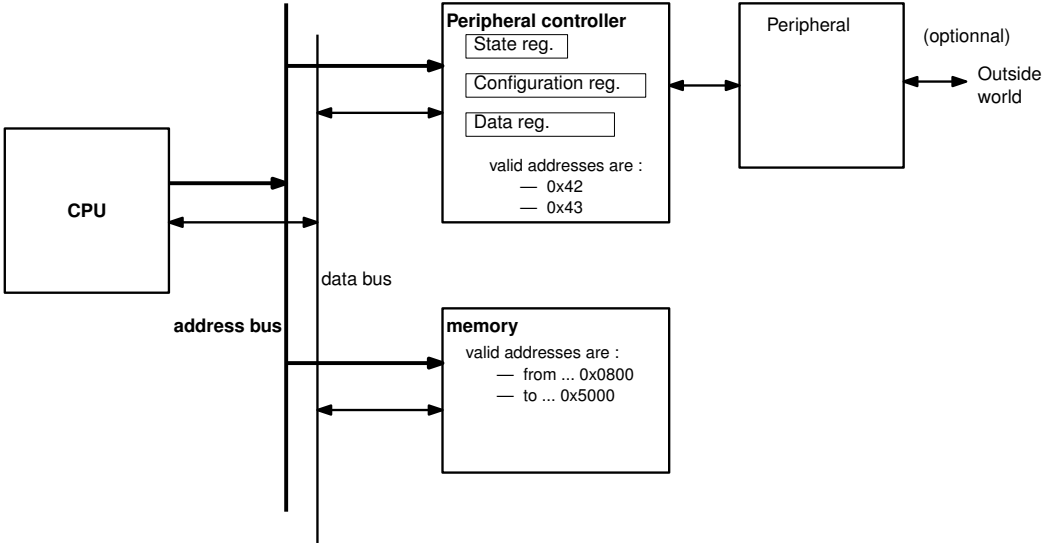
  `mv #42, TACTL`

  like we write `mv #42, R5`
- ... So how do we access those ????

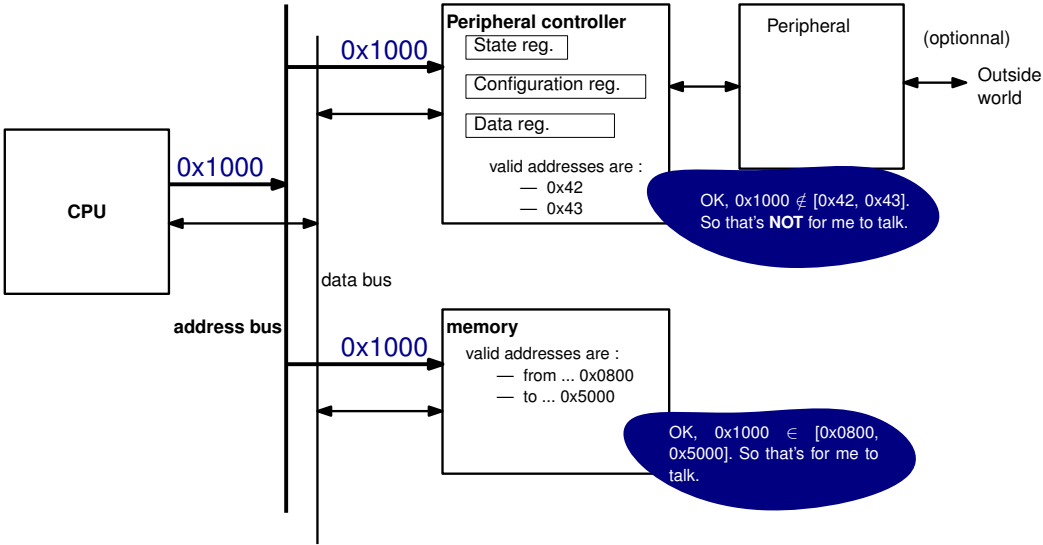
Memory-Mapped I/O

- Devices and memory share the **same address space**
- Some parts of the memory address space correspond to **real memory cells**
- Some parts of the memory address space correspond to peripheral controllers' registers.
- Any CPU instruction that can access memory can be used to **transfer data to/from an I/O device**

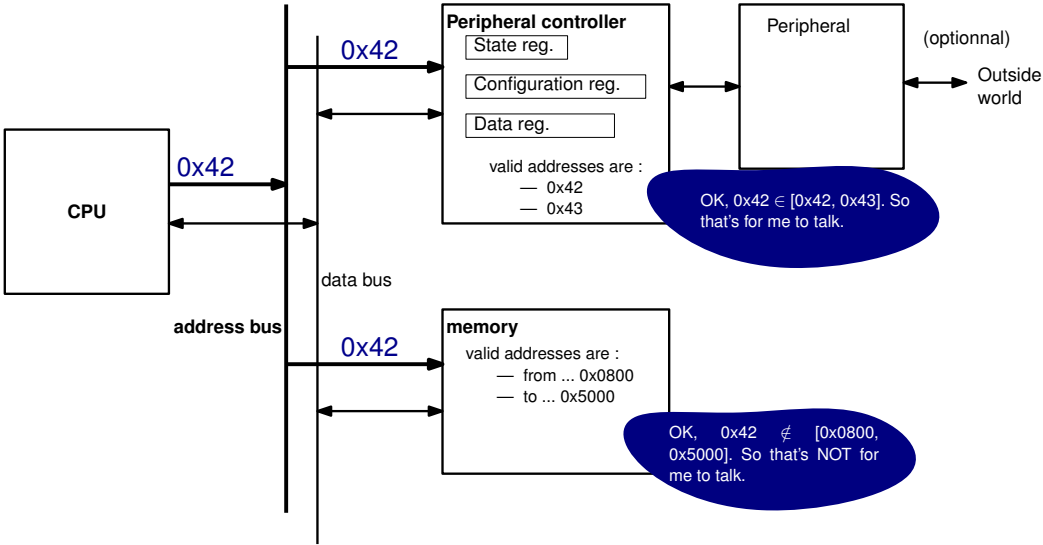
Memory-Mapped I/O - How it works



Memory-Mapped I/O - How it works



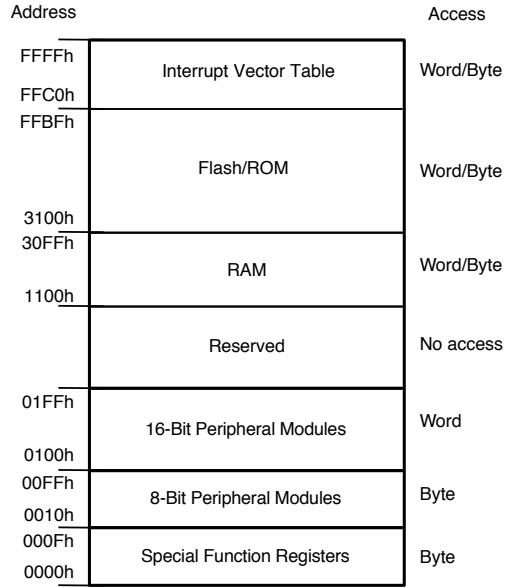
Memory-Mapped I/O - How it works



msp430 - Memory Layout

Most peripherals on the msp430 can be accessed through MMIOs

- The address space is logically split amongst the different peripherals
- Here the address space is 2^{16} bytes large.
- But the “real” memory is split between:
  - RAM from 0x1100 to 0x30ff
  - Flash from 0x3100 to 0xffbf
  - Notation Warning: 30FFh $\equiv$ 0x30FF



msp430 - MMIO

Table 15–3.Timer_A3 Registers

Register	Short Form	Register Type	Address	Initial State
Timer_A control Timer0_A3 Control	TACTL/ TA0CTL	Read/write	0160h	Reset with POR
Timer_A counter Timer0_A3 counter	TAR/ TA0R	Read/write	0170h	Reset with POR
Timer_A capture/compare control 0 Timer0_A3 capture/compare control 0	TACCTL0/ TA0CCTL	Read/write	0162h	Reset with POR
Timer_A capture/compare 0 Timer0_A3 capture/compare 0	TACCR0/ TA0CCR0	Read/write	0172h	Reset with POR
Timer_A capture/compare control 1 Timer0_A3 capture/compare control 1	TACCTL1/ TA0CCTL1	Read/write	0164h	Reset with POR
Timer_A capture/compare 1 Timer0_A3 capture/compare 1	TACCR1/ TA0CCR1	Read/write	0174h	Reset with POR
Timer_A capture/compare control 2 Timer0_A3 capture/compare control 2	TACCTL2/ TA0CCTL2	Read/write	0166h	Reset with POR
Timer_A capture/compare 2 Timer0_A3 capture/compare 2	TACCR2/ TA0CCR2	Read/write	0176h	Reset with POR
Timer_A interrupt vector Timer0_A3 interrupt vector	TAIV/ TA0IV	Read only	012Eh	Reset with POR

The TACTL registers is accessed (in R/W mode) by using address 0x0160

mv #42, &0x0160

(Warning: the #42 value here is probably meaningless for TACTL!)

Different Types of Peripherals

- So far, we have talked about a specific “Timer” peripherals
- Depending on the platform, many different types of peripherals might be avaible
- eg, on our msp430:
  - Clock modules
  - Flash-Memory controller
  - Hardware multiplier
  - DMA controllers
  - Watchdog Timer
  - Real Time Clock
  - USART peripheral serial (UART or SPI modes)
  - LCD controller
  - DAC and ACD
  - etc.
  - **General Purpose Input-Output** devices, GPIOs

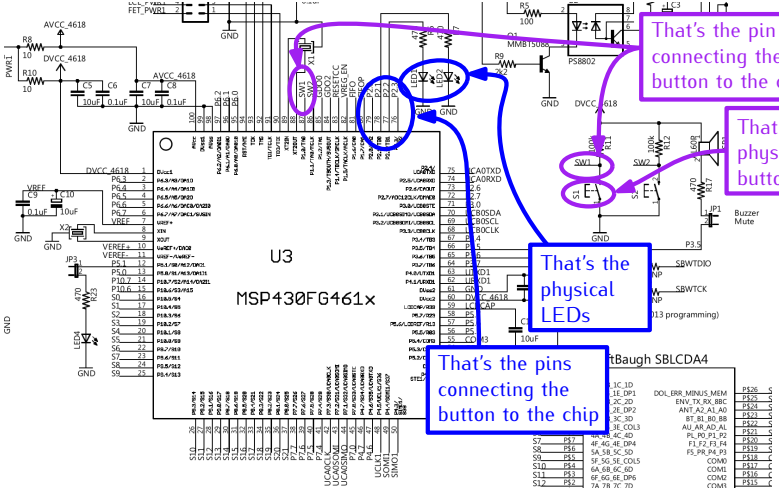
GPIO: Definition - Inside the msp430 chip

- **General-Purpose Input/Output**
- One **pin** that can be configured by software as Input or Output
- Pins are grouped by packs of 8, called **ports**
- `P1.7`, ..., `P1.0` are the eight pins grouped inside port `P1`
- On our board, `P1.0` is the pin connected to the switch button 1
- `P1.1` is the pin connected to the switch button 2
- `P1` is controlled through 3 8bits registers:
  - `P1IN` used to read data from the 8 pins
  - `P1OUT` used to write data to the 8 pins
  - `P1DIR` used to configure the 8 pins as inputs or outputs

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	–
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC

These are memory-mapped too!!

msp430 - Buttons and LEDs - Outside the msp430 chip



Polling[1]

Definition

= SW regularly sampling the activity of a HW element

- Sometimes called “Busy-wait” synchronization
- Example: let’s switch the led ON when the button has been pressed

```
loop1:
  mov.b #3, R15          ;; these lines compare
  mov.b &32, R14         ;; the bits that are
  add.b R15, R14, R14    ;; set to ONE whenever
  cmp.b R14, R15         ;; the button is pressed
  jeq loop1              ;; as long as button NOT pressed,
                         ;; actively look for its state

  mov.b #2, &49          ;; do this only when out from loop
```

[1] In French: Attente Active

Polling - Limits

- As the name suggests, software keeps the CPU “busy waiting for something to happen”
- So while we’re waiting, the CPU doesn’t do anything else!
- What if we integrated dealing with outside events right within the CPU?

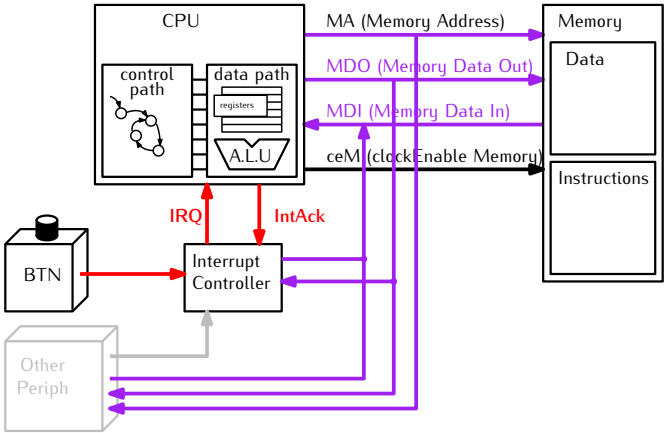
Interrupts - HW Principle

- We need a mechanism for:
  - A device to signal it wants the attention of the CPU
  - The CPU to stop what it's doing
  - When over, the CPU needs to move back to what it was doing before
- This requires:
  - A mechanism for the device to raise a **flag**
  - The CPU to check for those flags regularly
  - The CPU to be able to jump from the currenly executed code to a piece of code dedicated to handling that device's requests
  - The CPU to be able to jump back to what the CPU was doing previously

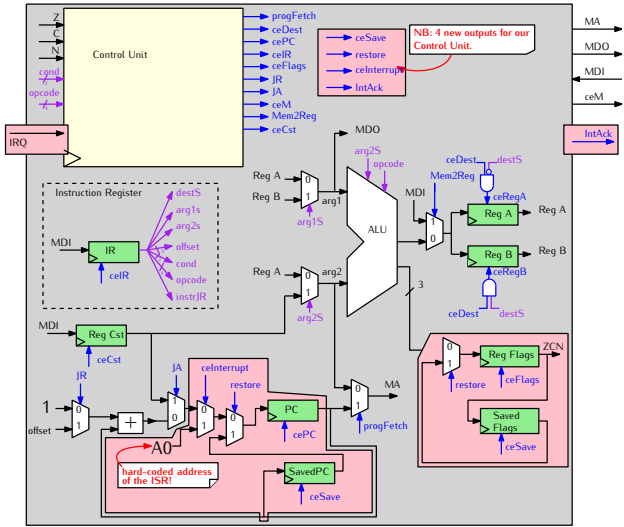
In a sense, the CPU incorporates the busy-wait procedure ... Much more efficient... !!

Interrupts in the Micro-Machine: Principle

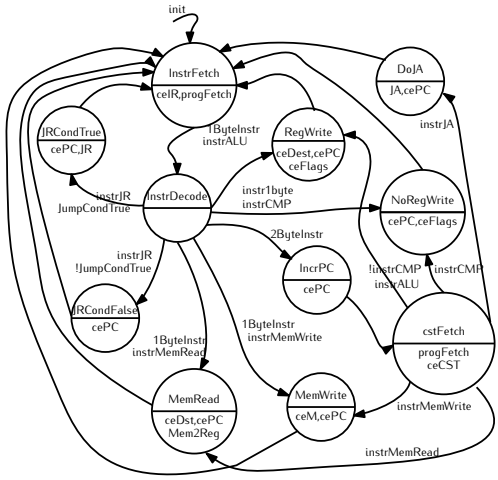
- Only one device: a special button called “Signal”
- No Interrupt Vector (IRQ is enough in this case)
- The Interrupt Service Routine is located at `0xA0`



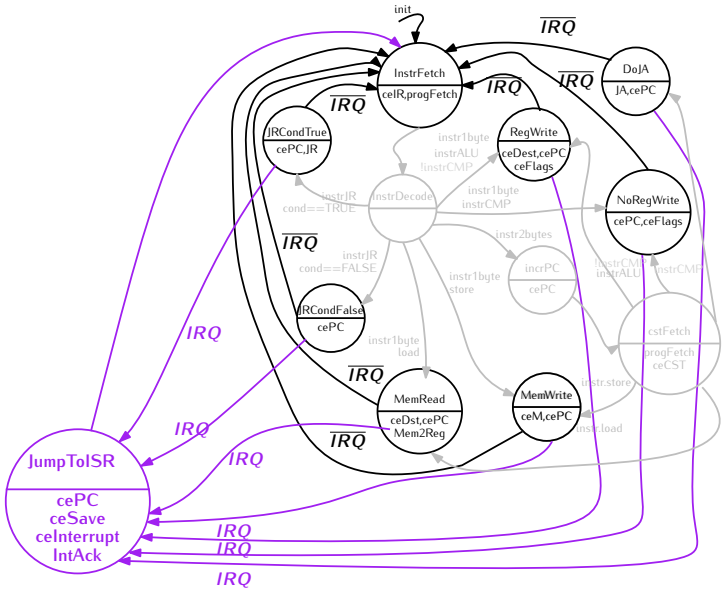
Interrupts in the Micro-Machine: New Datapath



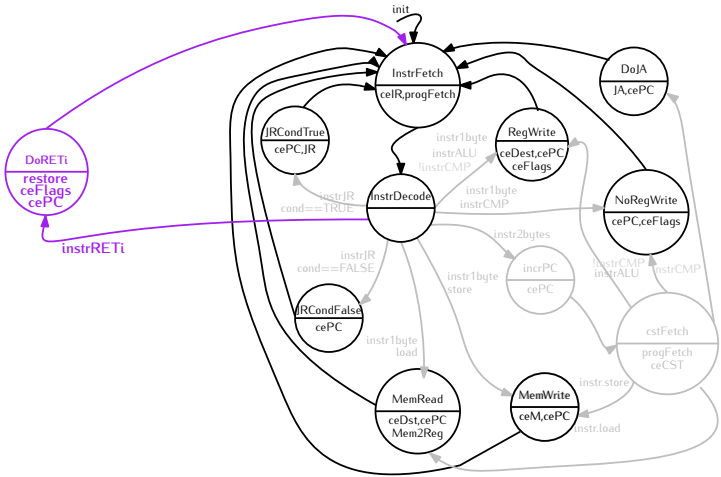
Interrupts in the Micro-Machine: New Control Unit



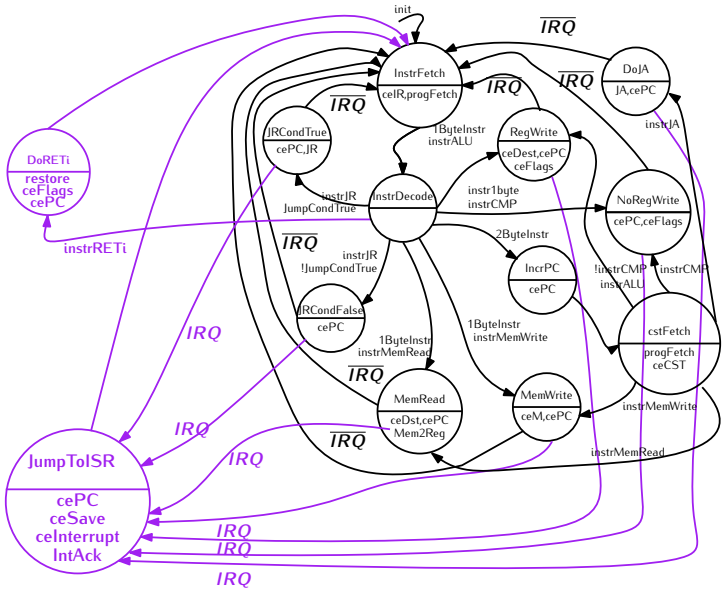
Interrupts in the Micro-Machine: New Control Unit



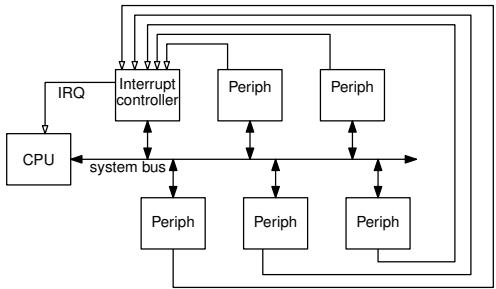
Interrupts in the Micro-Machine: New Control Unit



Putting it all together: the complete Control Unit



Interrupts - HW Principle



- CPU can’t have one pin for each separate device.
- We somewhat need to multiplex

Interrupts: SW side

```
myisr:
  /* instructions dealing */
  /*   with your device */
  mov.b #42, 0x31
  ...
  /* go back to "wherever */
  /* we were before" */
  reti
```

```
.section .init9
main:
  /* init the red LED */
  mov.b #1, &50
  /* turn it off */
  mov.b #0, &49
  /* turn it on */
  mov.b #2, &49

loop:
   jmp loop
```

Principle

- For each device, an identifier exists called "**Inverrupt Vector**"
- This IV is associated to an **Interrupt Routine**, here named `myisr`
- When **IV** is received, the currently-executing **instruction I_i** is finished, and the CPU goes to execute `myisr`
- When `myisr` executes `reti`, program resumes at **instruction I_{i+1}**

Interrupts - Working Principle and Vocabulary (1/2)

The device

- Emits a request towards the **Interrupt Controller** (IC)

The **IC**

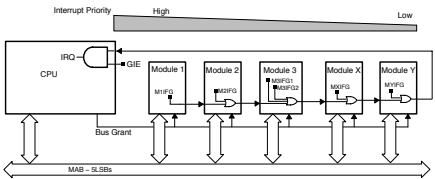
- Sorts requests
- Raises an **Interrupt Request** (IRQ) to the CPU.
- This is a **dedicated signal** wire between IC and CPU

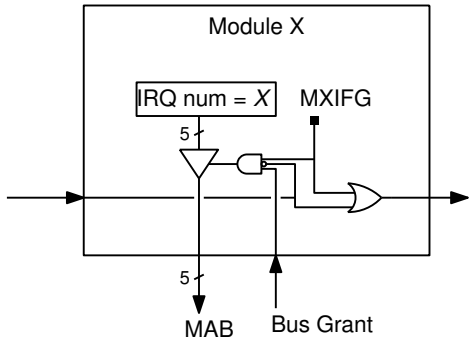
Interrupts - Working Principle and Vocabulary (2/2)

The **CPU**

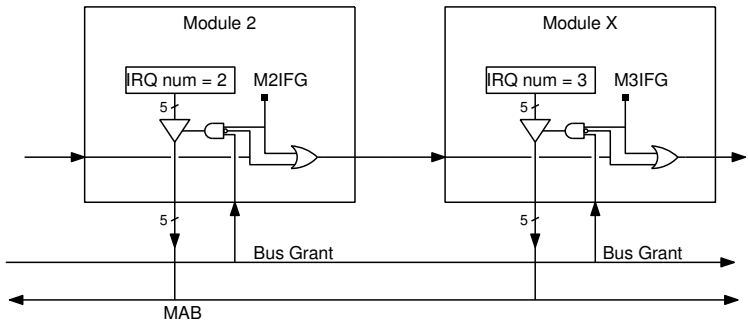
- Receives IRQ with identifier, named **Interrupt Vector** (IV)
  - BTW: Expects an **Interrupt Service Routine** (ISR) to be associated to that IV
- **Saves** the address of the instruction to come back to
  `[Some Place]` ← `PC+4`
- **Jumps to the ISR**
- Executes it
  - `Fetch, Decode, Execute`, ...,
  - "Well that's just code!"
- When end of ISR is reached, it should include a **special instruction to bring the CPU back to where it was before**
- This instruction is called a **Return from interrupt**, in msp430 `reti`

msp430: Interrupts

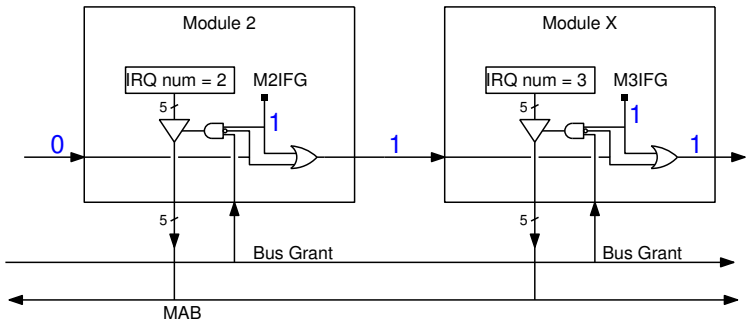




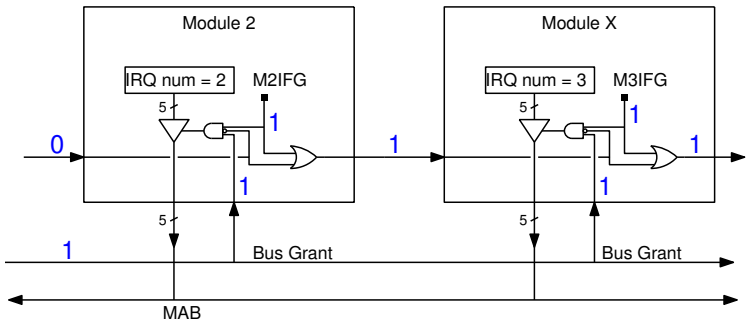
msp430: Interrupts



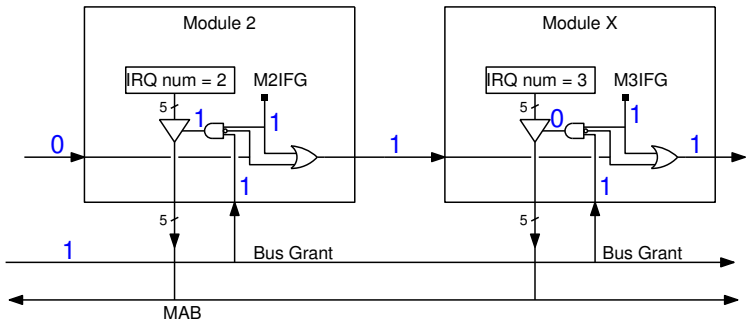
msp430: Interrupts



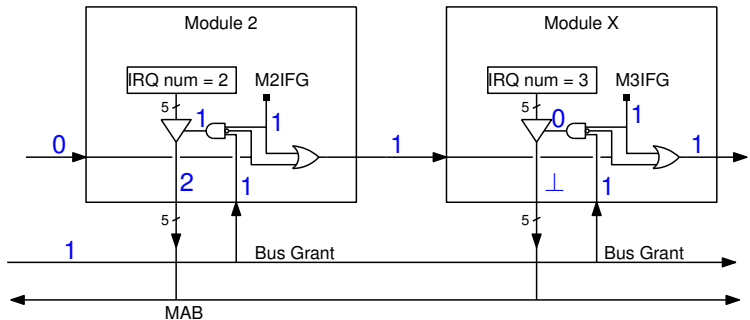
msp430: Interrupts



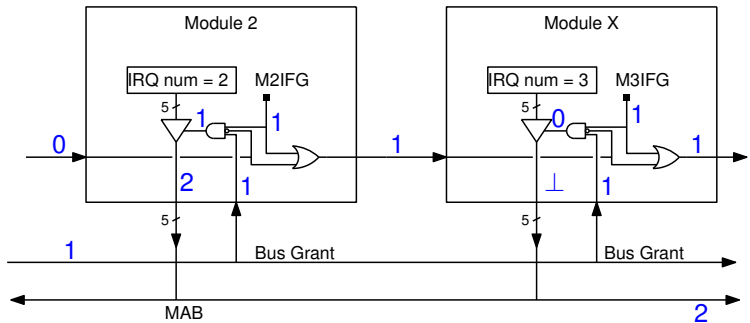
msp430: Interrupts



msp430: Interrupts



msp430: Interrupts



msp430: Interrupt Processing (1/5)

Interrupt Acceptance

The interrupt latency is six cycles, starting with the acceptance of an interrupt request and lasting until the start of execution of the first instruction of the interrupt-service routine, as shown in Figure 2–6. The interrupt logic executes the following:

1) Any currently executing instruction is completed.

2) The PC, which points to the next instruction, is pushed onto the stack.

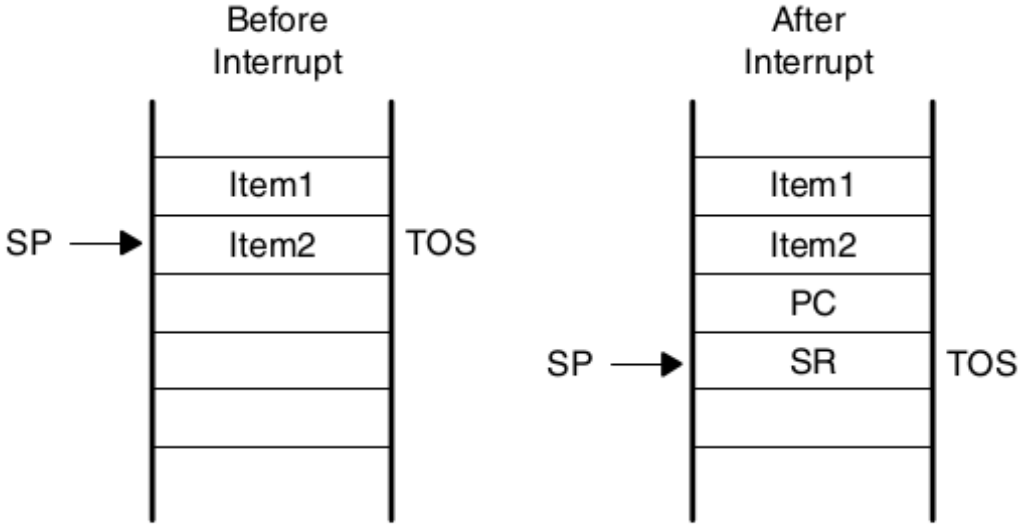
3) The SR is pushed onto the stack.

4) The interrupt with the highest priority is selected if multiple interrupts occurred during the last instruction and are pending for service.

5) The interrupt request flag resets automatically on single-source flags. Multiple source flags remain set for servicing by software.

6) The SR is cleared with the exception of SCG0, which is left unchanged. This terminates any low-power mode. Because the GIE bit is cleared, further interrupts are disabled.

msp430: Interrupt Processing (2/5)



msp430: Interrupt Processing (3/5)

Return From Interrupt

The interrupt handling routine terminates with the instruction:

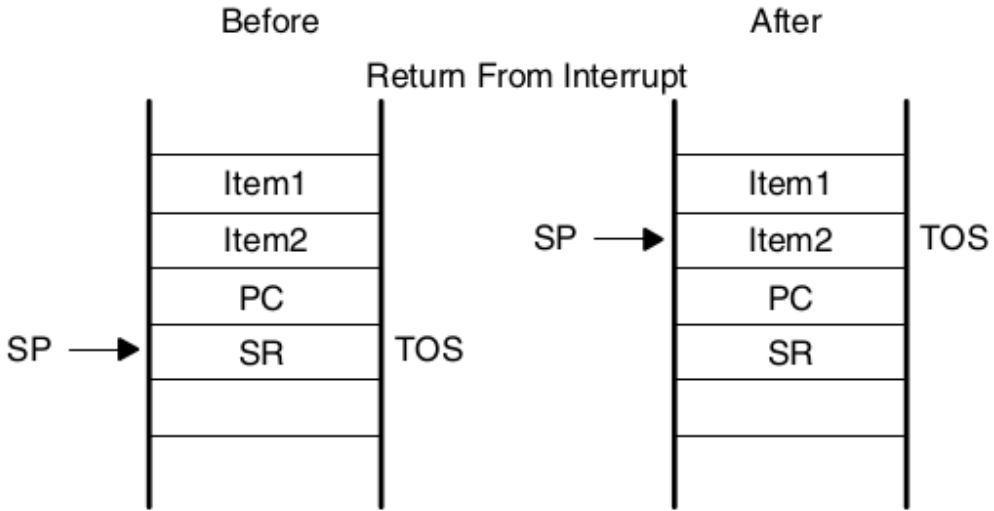
`RETI` (return from an interrupt service routine)

The return from the interrupt takes 5 cycles to execute the following actions and is illustrated in Figure 2–7.

1) The SR with all previous settings pops from the stack. All previous settings of GIE, CPUOFF, etc. are now in effect, regardless of the settings used during the interrupt service routine.

2) The PC pops from the stack and begins execution at the point where it was interrupted.

msp430: Interrupt Processing (4/5)



msp430: Interrupt Processing (5/5)

2.2.4 Interrupt Vectors

The interrupt vectors and the power-up starting address are located in the address range 0FFFFh to 0FFE0h as described in Table 2–1. A vector is programmed by the user with the 16-bit address of the corresponding interrupt service routine. Some devices may contain more interrupt vectors. See the device-specific data sheet for the complete interrupt vector list.

Table 2–1. Interrupt Sources, Flags, and Vectors

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-up, external reset, watchdog, flash password	WDTIFG KEYV	Reset	0FFFEh	15, highest
NMI, oscillator fault, flash memory access violation	NMIIFG OFIFG ACCVIFG	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	14
Device-specific			0FFFAh	13
Device-specific			0FFF8h	12
Device-specific			0FFF6h	11
Watchdog timer	WDTIFG	maskable	0FFF4h	10
Device-specific			0FFF2h	9
Device-specific			0FFF0h	8
Device-specific			0FFEEh	7

Ça serait quand même bien de leur toucher deux mots ....

- sur la hierarchie mémoire (plutôt ici)
- sur le vliw (plutôt dans le chapitre sur la construction du proc)
- le pipeline (mais c'est déjà un peu le cas dans le chapitre sur la construction du proc)