CAO Screenshots

Eduard Fekete

Character Count: 3170
Word Count: 337

December 25, 2025

1 Boolean, Logic

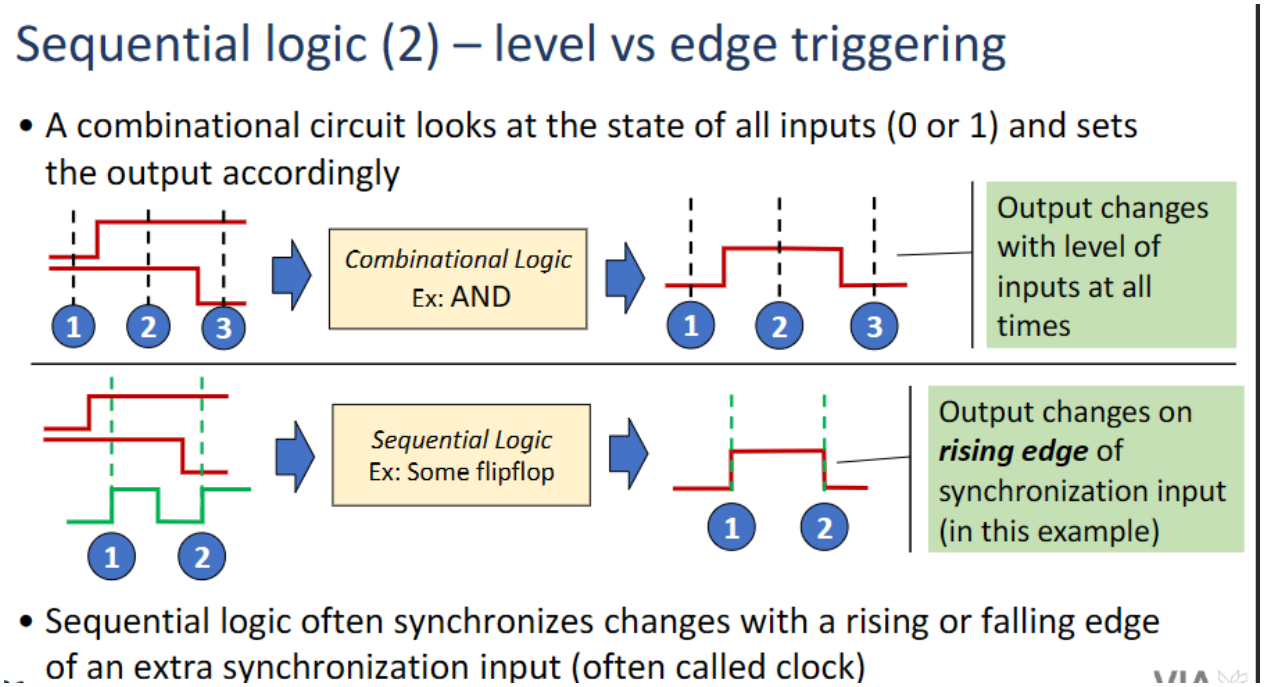


Figure 1: Level vs Edge triggering

Identity laws:	`A+0=A`	`A*1=A`
Null laws:	`A+1=1`	`A*0=0`
Idempotent laws:	`A+A=A`	`A*A=A`
Complement laws:	`A+!A=1`	`A*!A=0`
Double negation law:	`!!A=A`	
Commutative laws:	`A+B=B+A`	`A*B=B*A`
Associative laws:	`(A+B)+C=A+(B+C)`	`(A*B)*C=A*(B*C)`
Distributive laws:	`A*(B+C)=(A*B)+(A*C)`	`A+(B*C)=(A+B)*(A+C)`

Figure 2: Boolean Algebra Laws

1.1 Signed and Unsigned number ranges by bits

Bit Size (n)	Common Name	Unsigned Range (0 to $2^n - 1$)	Signed Range (Two's Complement) (-2^{n-1} to $2^{n-1} - 1$)
4	Nibble	0 to 15	−8 to 7
8	Byte	0 to 255	−128 to 127

Bit Size (n)	Common Name	Unsigned Range (0 to $2^n - 1$)	Signed Range (Two's Complement) (-2^{n-1} to $2^{n-1} - 1$)
16	Short / Word	0 to 65,535	−32,768 to 32,767
32	Int / DWord	0 to 4,294,967,295	−2,147,483,648 to 2,147,483,647
64	Long / QWord	0 to 18,446,744,073,709,551,615	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
128	Octaword / UUID	0 to 3.402×10^{38}	-1.701×10^{38} to 1.701×10^{38}
n	General	0 to $2^n - 1$	-2^{n-1} to $2^{n-1} - 1$

1.2 Multiplexers and Demultiplexers

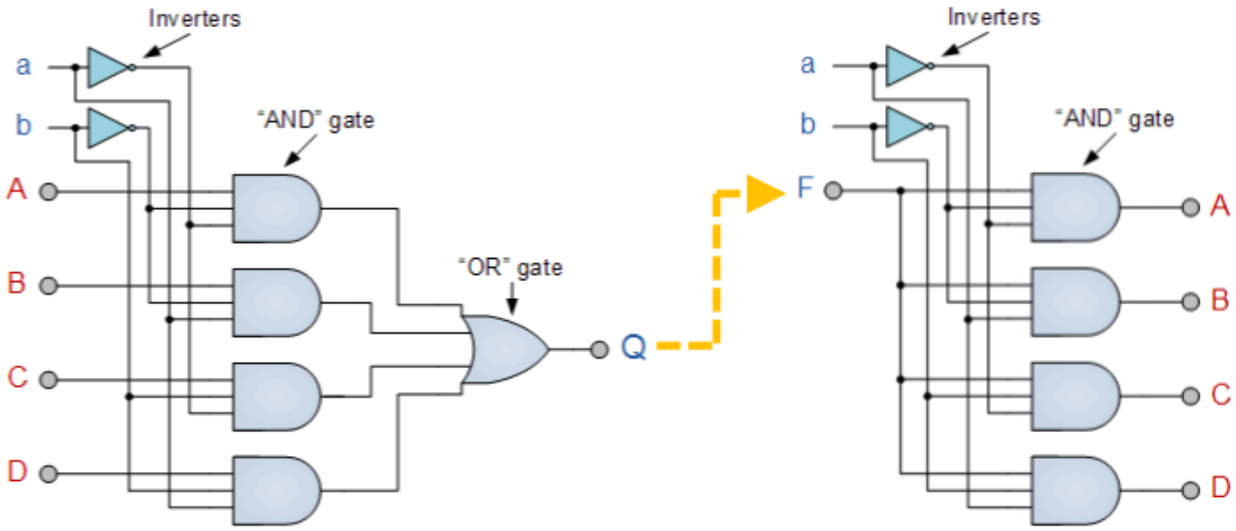


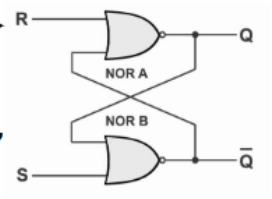
Figure 3: mux/demux

1.3 SR Flip Flops

The SR-flipflop (1) – the basic state core

Latch means "keep current state"

- The basic part of a flipflop is some kind of logic circuit with feedback, like this (this one is known as an SR-latch): → Let's have a look at it in Logisim (`SR_latch.circ`)



- Logisim flags some errors initially, but once we toggle it, we can set it with S (Q=1) and reset it with R (Q=0). Q and !Q should always be opposite. Here's what we see in Logisim: →

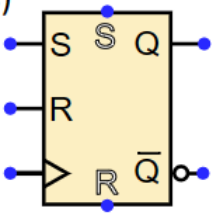
S	R	Q	!Q
0	0	Latch	Latch
0	1	0	1
1	0	1	0
1	1	0	0

- But there's some problems here:
  - There's some form of "initial not-well-defined" state until we set the first value
  - There's an "illegal output" when S=R: Q and !Q becomes the same

Figure 4: SR flip Flops #1

The SR-flipflop (2) – making a usable building block

- While the something like the SR-latch is at the core of all flipflops, real (usable) flipflops are enhanced to avoid undefined an initial state and other undesired behavior
- On top of that a generic flipflop building block also gets equipped with a synchronization input (often called clock), that provides extra control of state changes (flips or flops)
- If we just remove the undefined part and add a clock input, the result is an SR-flipflop: →



S	Set (synchronously on rising clock edge)
$\mathbb{S}$	Set asynchronously (at any time)
R	Reset (synchronously on rising clock edge)
$\mathbb{R}$	Reset asynchronously (at any time)
>	Clock
Q !Q	Outputs

- ... plus we add some async. inputs

Figure 5: SR flip Flops #2

The SR-flipflop (3) – a really usable building block ...

- The truth table for an RS-flipflop like looks like this:
- On top of that, the 𝕊 and ℝ inputs can be used to force a specific state at any time:
  - Asynchronous reset ℝ : When 0 or undefined, this input has no effect. When 1, the Q is pinned to 0 and the other inputs have no effect.
  - Asynchronous set 𝕊 : When 0 or undefined, this input has no effect. When 1, Q is pinned to 1 and the other inputs have no effect.
  - When both ℝ and 𝕊 are active, the Q is pinned to 0 since ℝ has priority.
- But there's still a line that may cause problems – we still need to take care that R and S are not set at the same time

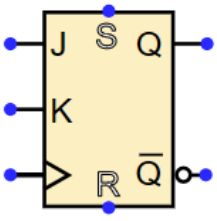
Clock	S	R	Q	!Q
⎍	0	0	Latch	Latch
⎍	0	1	0	1
⎍	1	0	1	0
⎍	1	1	0	0
X	0	0	Latch	Latch
X	0	1	Latch	Latch
X	1	0	Latch	Latch
X	1	1	Latch	Latch

Figure 6: SR flip Flops #3

1.4 JK Flip Flops

The JK-flipflop – a SR-flipflop deluxe

- If some extra circuitry is added an SR-flipflop, we can create a flipflop that is robust with all input combinations.
- The result is called an JK-flipflop, and it looks like an SR-flipflop, only the S/R inputs has been renamed after the inventor (**J**ack **K**ilby – who invented the integrated circuit - IC)



J	Set (synchronously on rising clock edge)
𝕊	Set asynchronously (at any time)
K	Reset (synchronously on rising clock edge)
ℝ	Reset asynchronously (at any time)
>	Clock
Q !Q	Outputs

Clock	J	K	Q	!Q
⎍	0	0	Latch	Latch
⎍	0	1	0	1
⎍	1	0	1	0
⎍	1	1	Toggle	Toggle
X	0	0	Latch	Latch
X	0	1	Latch	Latch
X	1	0	Latch	Latch
X	1	1	Latch	Latch

Figure 7: JK Flip Flop

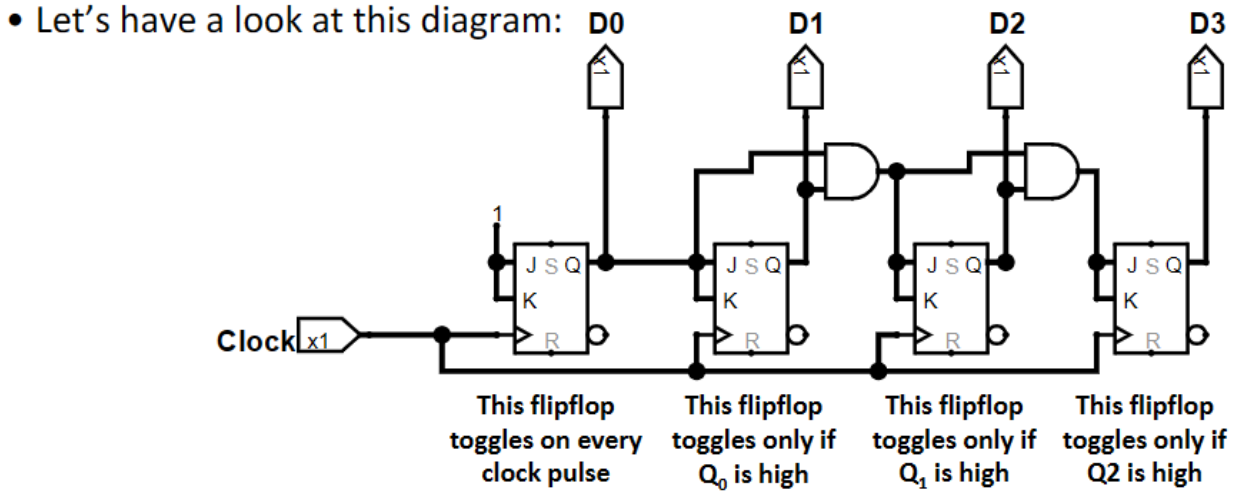


Figure 8: 4-bit counter

2 Atmega2560

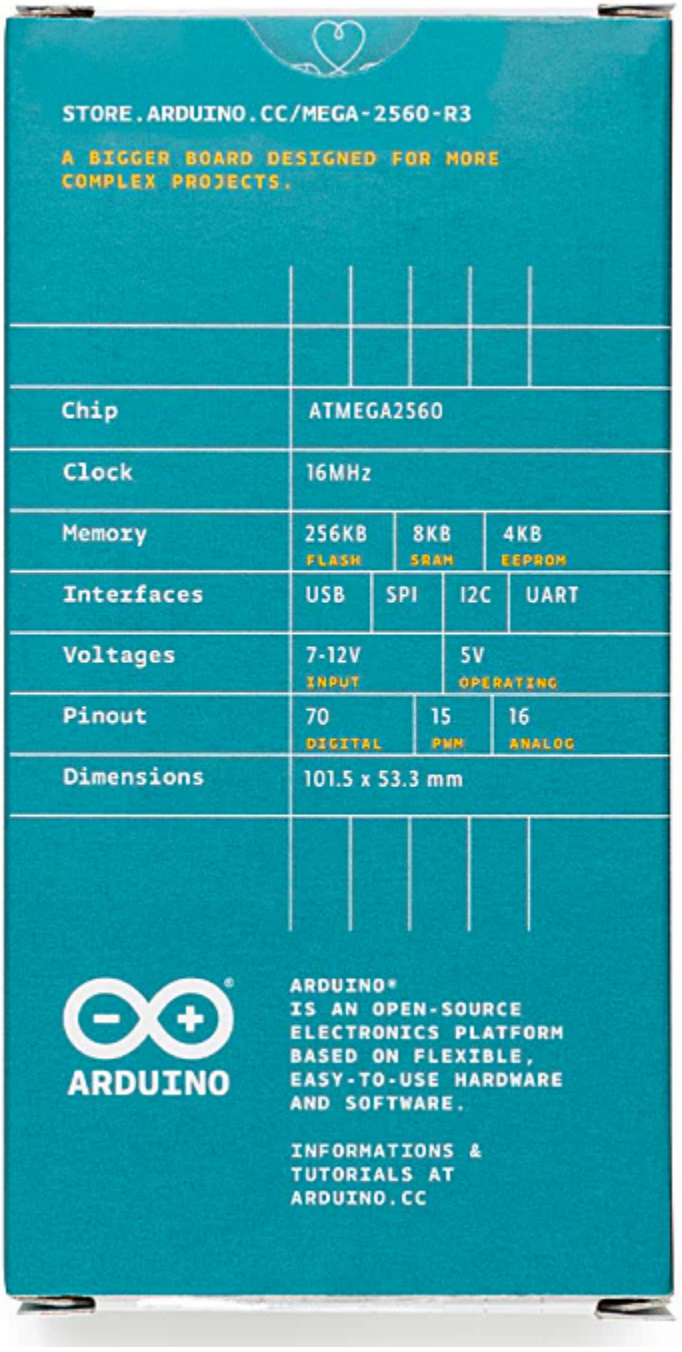


Figure 9: Packaging with specs

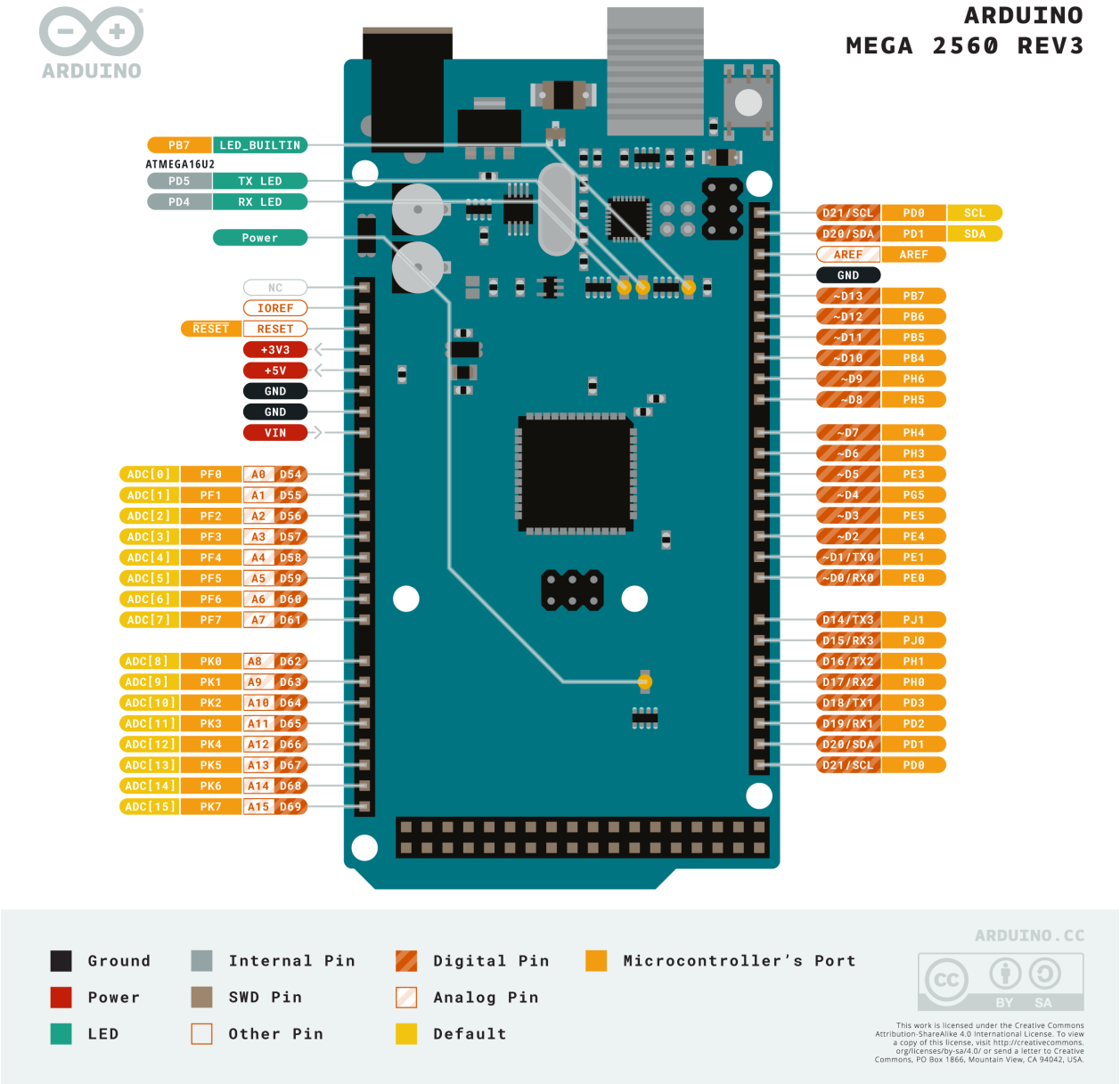


Figure 10: Arduino MEGA pins

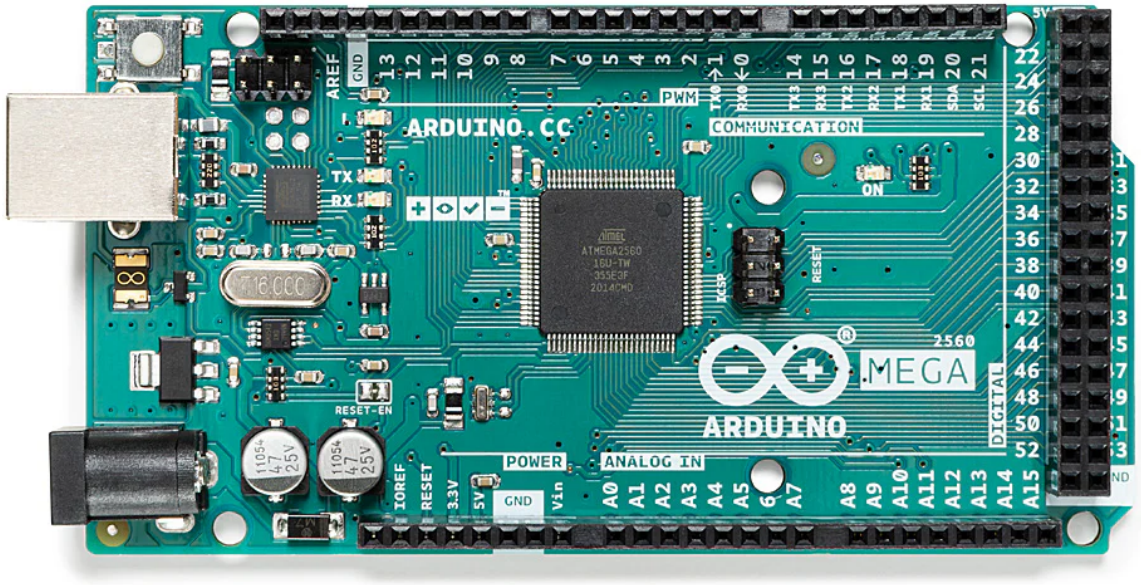
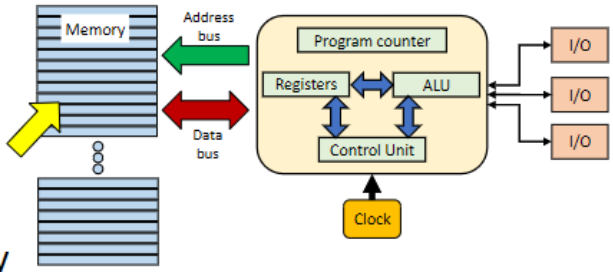


Figure 11: Arduino MEGA picture

Memory and storage types



- Memory is the primary storage medium for the CPU
- The CPU reads and writes to memory by setting an address on the address bus and the reads/writes to the memory word at the specified address over the data bus
- Memory comes in many forms, some of which are:
  - **RAM** (Random Access Memory) – fast read/write - large memory – volatile
  - **Flash** – fairly fast reads, slower writes – can be very large – nonvolatile – wears down with many write cycles – can come with very large capacities
  - **EEPROM** (Electrically Erasable and Programmable Read-Only Memory) – not so fast - can be both read and written – nonvolatile – used for small but important data sets (configurations etc.) – very cheap in small capacities

Figure 12: Memory and Storage types

3 General Stuff

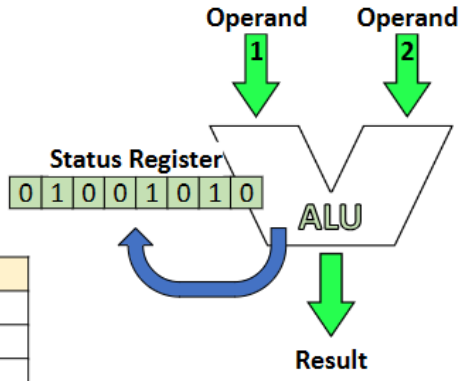
Special registers

- We have already seen some registers in our CPU model
  - Program counter – always points to (contains address of) next instruction
  - General purpose registers – used for general computations
- But there are other registers in a CPU, and that includes the Atmega2560
- ***Status register***
  - This register gets updated each time the ALU performs an operation
- ***Stack pointer***
  - This register points to an address in data memory (the RAM)
  - It is used to provide an easy way to use data memory for certain kinds of data
- We'll take a quick look at these registers in the next two slides, and get into more detail later in the course

Figure 13: Special Registers

Special register: the status register

- *Every time* the ALU performs an operation the bits of the status register gets updated to signal certain conditions, such as:
  - The result was negative/not negative
  - The result was zero/not-zero
  - A carry was generated from last operation
- The status register (SR) in the Atmega2560 has 8 bits, some of which are



Bit	*Name*	*If set (=1) Indicates*
0	Carry flag	There was a carry
1	Zero flag	Result was zero
2	Negative flag	Result was negative
3	2's compl. overflow	Overflow in 2's compl. operation

Figure 14: SREG

Special register: the stack pointer

- It's very convenient to be able to temporarily store data from registers into RAM and then retrieve it again later – the CPU does that with a *stack*
- A stack is an area of RAM set aside specifically for this purpose
- We store and retrieve data from the stack in a certain way: we store data with a ***push*** instruction and retrieve data with a ***pop*** instruction
- The stack grows from the end of data memory (high address) and up towards lower addresses
- The stack pointer always points to where the next data will be stored with a ***push***

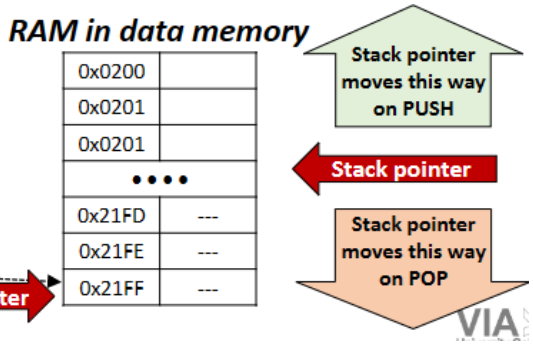


Figure 15: Stack Pointer

Anatomy of an Atmega2560 instruction (1) - **Size**

- An instruction is either 2 bytes (the most) or 4 bytes (only a few)
- All instructions have the same core layout, [opcode operand]:

Instruction size	Opcode	Operand(s) 0-2 operands
16	4-16 bits	0-12 bits
32	11-12	20-21

Let's take a quick look in the instruction set manual to see where this info came from

- Example, **LDI Rd,K** (load Immediate value into register):

16-bit Opcode:

1110	KKKK	dddd	KKKK
Opcode = 1110	First 4 bits of value	4 bits to specify register	Last 4 bits of value

- Notice that the bits for the operands are split across the 16-bit word
- The opcode are also sometimes split across the 16-bit word

Figure 16: Instruction Size

4 Control Flow

Calling an assembly subroutine, summary

Step	Caller	Subroutine
1	Put any arguments in registers or RAM	
2	Call subroutine (**CALL** instruction)	
3		Save registers (**PUSH** instruction(s))
4		Pull arguments from registers or RAM, do stuff
5		Place any output in register(s) or RAM
6		Restore registers (**POP** instruction(s))
7		Return (**RET** instruction)
8	Get return value from register or RAM and continue	

- In a high-level language the same things happen, but the compiler (or interpreter) takes care of all the nitty-gritty details

Figure 17: Subroutines call

Type	Instruction	Description
Unconditional control flow	rjmp k	Relative jump, PC←PC +/- 2048
	jmp <addr>	Absolute jump, PC←addr (full address range)
	call <addr>	Push ret.addr, PC←addr (full address range)
	rcall k	Push ret.addr, PC ←PC +/- 2048)
	ret	Pop address from stack, PC←address
Compare/test	cp Rd,Rr	Compare (Rd – Rr)
	cpi Rd,k	Compare with immediate (Rd – k)
Conditional control flow	breq k	Branch on equal (Z==1), PC←PC+64/-63
	brne k	Branch on not equal (Z==0)), PC←PC+64/-63
	brsh k	Branch on same/ higher (C==0)), PC ←PC+64/-63
	brlo k	Branch on lower (C==1)), PC←PC+64/-63

Figure 18: Instructions Table Control Flow

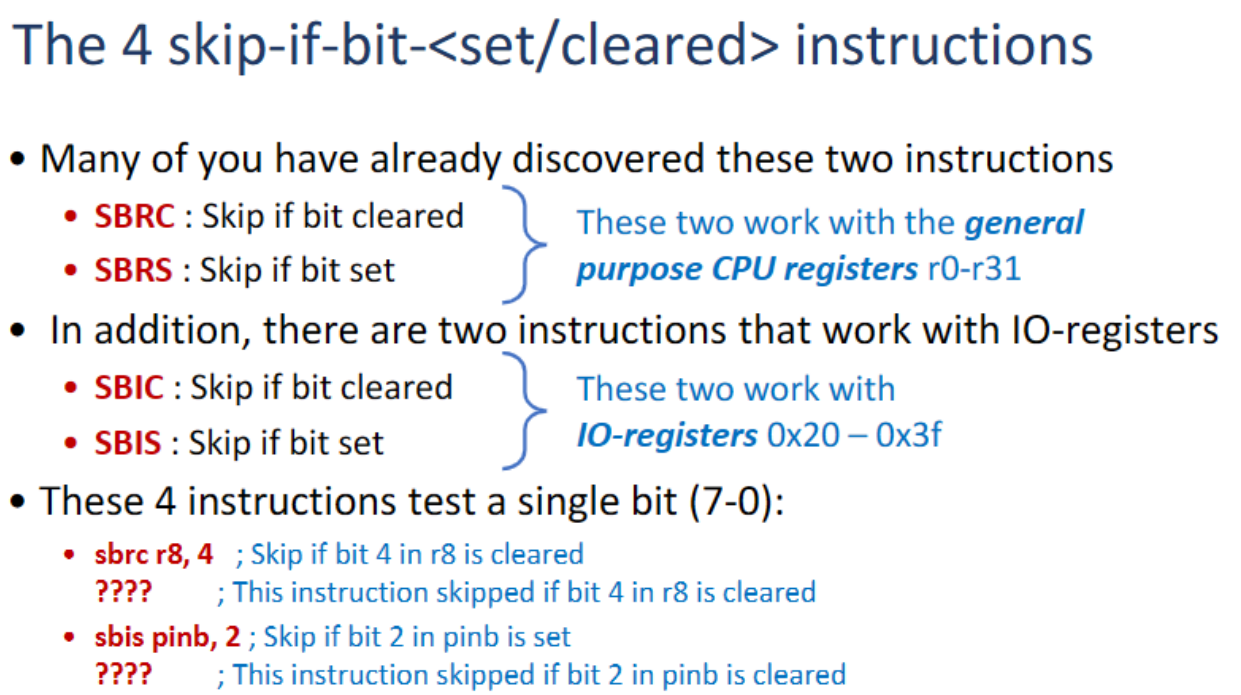


Figure 19: Skip if bit

5 I/O

Steps for using a port:

1. Choose the port you want to use, call it x
2. Set direction of port bits/pins in DDRx (only needed once)
3. - Write to PORTx to set output pins (high/low) (or to set pull-up for input pins)
   - Read from PINx to read current state of input pins (high/low)

Port example – writing to port (using the built-in led)

- The led on the Arduino is connected to portB pin 7. The **out** instruction writes the contents of a register to a port in data (I/O) space:

```
; Set bit7 on portb = output, bit 0..6 as input
ldi r16, 0b10000000
out ddrb, r16

ldi r16, 0b10000000 ; Turn on led
out portb, r16

ldi r16, 0b00000000  ; Turn off led
out portb, r16
```

1 Set direction (in/out) for the bits in portB – you only need to do this once

Once configured, you can set the value (high/low) for the port as many times you like 2+

Figure 20: Writing to port

Port example – reading from a port

- The **in** instruction reads the contents of a port in data (I/O) space and stores the result in a register:

```
; Set bit 0..6 on portb as input, bit7 as output
ldi r16, 0b10000000
out ddrb, r16

in r16, pinb ; Read pins of portb
```

1 Set direction (in/out) for the pins in portB – you only need to do this once

Once configured, you can read the value (high/low) of the port pins as many times you like 2+

- The value seen for pins configured as output pins, is the currently set output value

Figure 21: Reading from port

cbi & sbi – handy for flipping bits in ports

- There are two instructions designed mostly for setting and clearing bits in port A to G
- sbi – Set bit in I/O register
  - sbi porta, 7 ; Set bit 7 in porta
- cbi – Clear bit in I/O register
  - cbi porta, 0 ; Clear bit 0 in porta

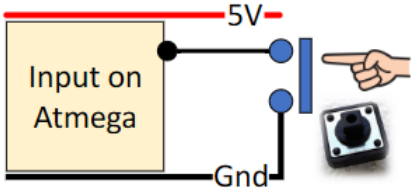
But luckily for us, m2560def.inc has already factored this in, so we can just use the nice named constants like porta without having to think about real I/O addresses

- These instructions read the current content of an I/O register and writes an updated value back
- cbi and sbi only works with registers in the address range 0x20 to 0x3F, because they, like in & out, implicitly add 0x20 to the I/O address
- So, I/O addresses must be supplied in the range 0x00-0x1F

Figure 22: cbi and sbi

What happens if an input is not connected?

- That means trouble!!
- If an input is not directly connected to a high or low voltage (Vcc or Gnd, +5v and zero for the Arduino), it will fluctuate randomly – if you read the input, if will sometimes be 1 (high) and sometimes 0 (low)
- An unconnected input is a no go!
- But what if we want to do something like this: We have a button, and we want the input to go low when the button pressed, will this work?



- No! It will set the input to 0/low when we press the button, but what's the value of the input when the button is NOT pressed?
- It's undefined, and connecting the input to +5V will not help

Shortcut when button pressed!

Figure 23: Input not connected

6 Memory

NAND flash reading, writing and erasing

- ***Reading:*** You can read individual bytes/words from a page (but always within the context of a specific page).
- ***Writing:*** NAND flash can only be written (programmed) at a page level. Before writing, the targeted page must be in an erased state (all 1s)—otherwise, the block containing the page must be erased.
- ***Erasing:*** Both NAND and NOR flash require erasing whole blocks (not individual bytes or pages). After erasure, all bits in a block are reset to 1.
- Before new data can be programmed into a page that already contains data, the current contents of the page plus the new data must all be copied to a new, erased page
- The microcontroller on the flash chip helps with all these shenanigans

Figure 24: NAND Flash specs

Flash wear – the weak spot of flash memory

- Every time a flash cell is written/erased it wears down a little
- The reason is simple:

1. It's common to use 10-20 volts to erase/write data (the chip produces this high voltage internally).
2. If we apply 20 volts between the substrate and the control gate, and the field is evenly distributed across the 2 insulator layers and the charge storage film, there's ~ ***6.5 V*** across the insulator the electrons must cross.
3. The insulator is around ***8 nm*** (nanometer, 10^{-9} meter)
4. 6.5 V / (8 x $10^{-9)}$ = 812500000 V/m ***= 812.5 MV/m*** (Megavolt/meter)
5. Shooting electrons through the insulator with almost 1000 MV/m tears the insulator

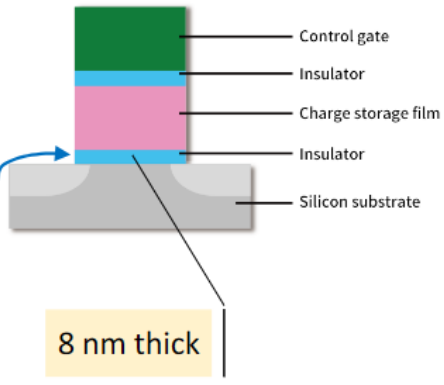


Figure 25: Flash stats

ATmega2560 EEPROM

- The ATmega2560 also has persistent storage in the shape of 4KB EEPROM
- It is not directly accessible in the data memory space, we need to go via a set of dedicated registers to read and write data (and these registers are of course accessible in the I/O space)

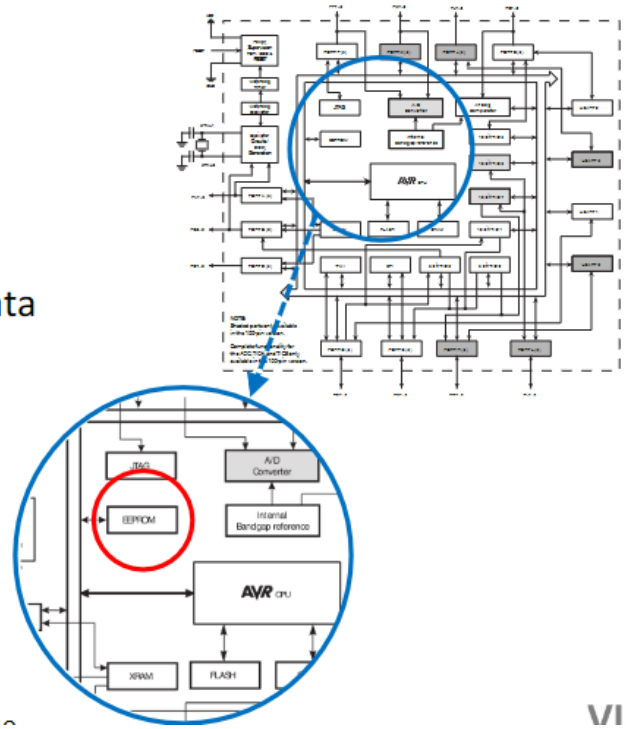


Figure 26: EEPROM

ATmega2560 EEPROM characteristics

- The EEPROM has 4096 bytes, each of which can be individually read and and written (no page/block structures here)
- When an EEPROM cell is erased, it ends up having only 1's (=0xff=255)
- The EEPROM is rated for at least 100.000 write/erase cycles
- It takes fairly long time to read and write to the EEPROM (typical write is around 3.3 ms
- A cell must be erased before it gets written
- Interrupts (which we will talk about next time) must be turned of while writing to the EEPROM
- The write process is very sensitive to supply voltage fluctuations
- EEPROM writes may fail, resulting in corrupted data being written

Figure 27: EEPROM specs

How to do an actual EEPROM read

- ***Reading***
  Before this code runs, r18:r17 must hold the address of the EEPROM cell to read (0-4095). When done, the content of the cell is in r16

```
read_eeprom_byte:
    ; Wait for completion of previous write
    sbic EECR, EEPE
    rjmp read_eeprom_byte
    ; Set up address (r18:r17) in address register
    out EEARH, r18
    out EEARL, r17
    ; Start eeprom read by writing EERE
    sbi EECR,EERE
    ; Read data from Data Register
    in r16, EEDR
```

Figure 28: EEPROM Read

How to do an actual EEPROM write

- ***Writing***
  Before this code runs, the data to be written must be in r16, and r18:r17 must hold the address of the EEPROM cell to write (0-4095)

```
save_eeprom_byte:
    ; Wait for completion of previous write
    sbic EECR,EEPE
    rjmp save_eeprom_byte
    ; Set up address (r18:r17) in address register
    out EEARH, r18
    out EEARL, r17
    ; Write data (r16) to Data Register
    out EEDR, r16
    ; Write logical one to EEMPE
    sbi EECR,EEMPE
    ; Start eeprom write by setting EEPE
    sbi EECR,EEPE
```

Figure 29: EEPROM Write

7 Interrupts

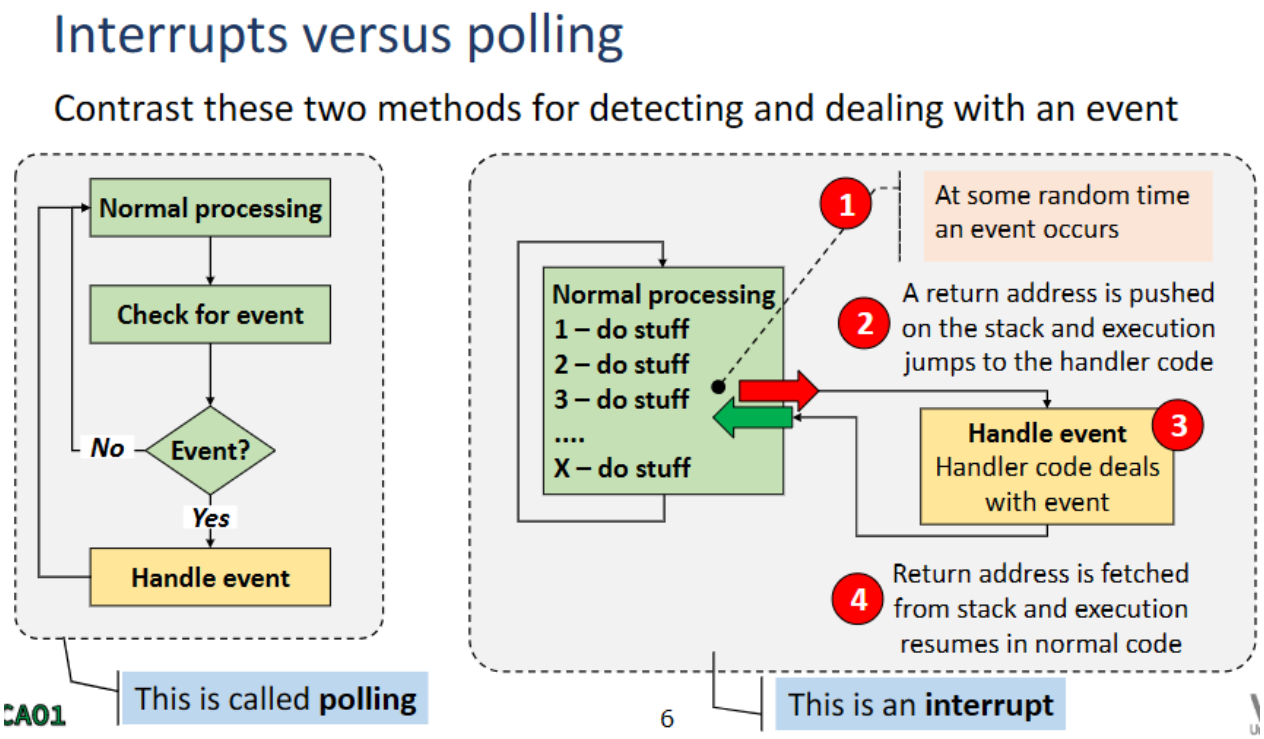


Figure 30: Interrupts vs Polling

	Polling	Interrupts
+	• Simple (to some extent – polling multiple events can get tricky)	• Very fast reaction time (around 8 clock cycles = 500 ns (nanoseconds) • Deterministic reaction time with a well-defined upper bound (max. 750 ns unless sleep mode is active) – no matter what happens in normal code • Does not waste CPU cycles
÷	• Wastes CPU cycles • Slower reaction time • Nondeterministic reaction time, not really with an upper bound • Reaction time dependent on what happens in normal code	• A bit more complex (but it's really just a forced call to a predefined routine)

Figure 31: pros cons inter. poll.

The interrupt vector table

Here's how the interrupt vector table looks on the AVR Atmega2560:

#	Interrupt	Address
1	Reset/power on	0x0000
2	External interrupt 0	0x0002
3	External interrupt 1	0x0004
4	External interrupt 2	0x0006
5-55	*(all sorts of interrupts)*	
56	USART3 Data Register Empty	0x006E
57	USART3 Tx Complete	0x0070

Notice these ...

- Power-on/reset is in fact handled like an interrupt
- We can also see that there's only 2 words set aside for each ISR
- The table can hold up to 57 entries for 57 different interrupts

Figure 32: Interrupt vector table

Code layout when using an interrupt vector table

```
.org 0x0000          // Reset/power-on
       rjmp main     // Jump to main loop
.org 0x000a          // INT4
       rjmp int4_isr // Jump to ISR
```

Interrupt vector table

```
.org 0x0072          // Make room for vector table
main:
       // Enable interrupts here
loop:
       nop
       rjmp loop
```

Main loop

The **.ORG** assembler directive instructs the assembler to put something at a specific address in code memory

```
int4_isr:
       // Handle interrupt here
       reti
```

Interrupt service routine

Figure 33: Interrupts example

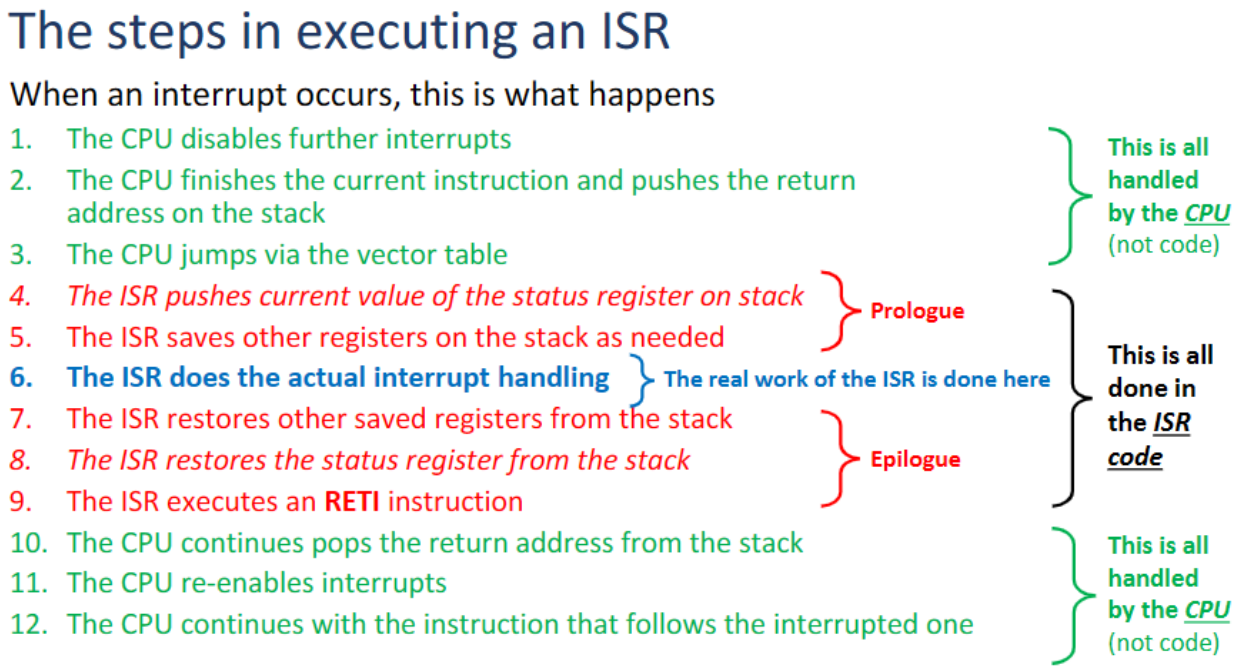


Figure 34: Interrupts timeline

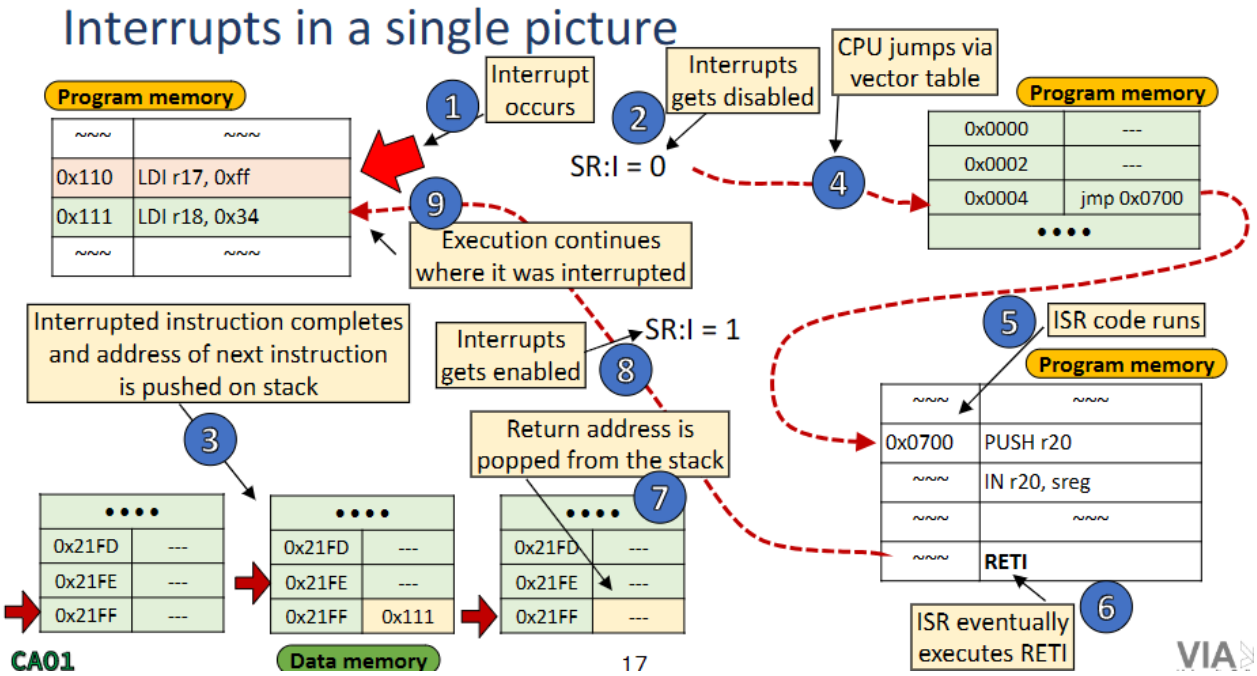


Figure 35: Interrupts path

Interrupts are enabled/disabled at two levels

1. All interrupts are enabled/disabled at a global level by setting or clearing the ***I bit in the status register***
   - `SEI` sets the I bit in SR and enables interrupts
   - `CLI` clears the I bit in SR and disables interrupts
2. In addition to that, each interrupt source (timer, external etc.) has a control register where interrupts for that specific source is enabled/disabled

- So, enabling any specific interrupt is always a 2-step process
  a) Setting up (enabling) the specific interrupt
  b) Enabling global interrupt

Figure 36: Enable/Disable interrupts

```
; --- Define some nice names for constants
#define BUTTON1 PD1
#define BUTTON2 PD0
#define LED1 PB7
#define LED2 PB6

.org 0x0000                ; --- Interrupt vector table
    jmp start
.org 0x0002
    jmp int0_isr

start:
    ; *** This is just the usual setup/init part ***
    call cao_usart0_init   ; Setup USART0
    ser r16                ; Setup portb as all output
    out DDRB ,r16
    clr r16                ; Setup portd as all input
    sbi PORTD, PD0         ; Use pull-up on pin 0
    sbi PORTD, PD1         ; and pin 1

    ldi r16,0b00000010     ; Set INT0 to trigger on falling edge
    sts EICRA,r16
    ldi r16,1<<INT0        ; Enable external interrupt INT0
    out EIMSK,r16
    sei                    ; Enable global interrupt
```

Figure 37: Example #1.1

```
int0_isr:
    push r16                ; Prologue, save state and registers
    in r16,sreg
    push r16

    sbic PIND, BUTTON2
    rjmp prologue           ; If button1 is not pressed, skip the rest
    ldi r16, 6
    call cao_delay_r16
    sbic PIND, BUTTON2
    rjmp prologue           ; Nothing to do unless button1 is still pressed
(debounce)
    sbic PORTB, LED2        ; Check current value of led1
    rjmp turn_led2_off
    sbi PORTB, LED2         ; a) If it's cleared, set it
    rjmp wait_release_button2
turn_led2_off:
    cbi PORTB, LED2         ; b) If it's set, clear it
wait_release_button2:
    sbis PIND, BUTTON2
    rjmp wait_release_button2 ; Don't move on until button1 is released again
prologue:                   ; Restore state and registers
    pop r16
    out sreg, r16
    pop r16
    reti

.INCLUDE "caolibrary.asm"
```

Figure 38: Example #1.2