MUP

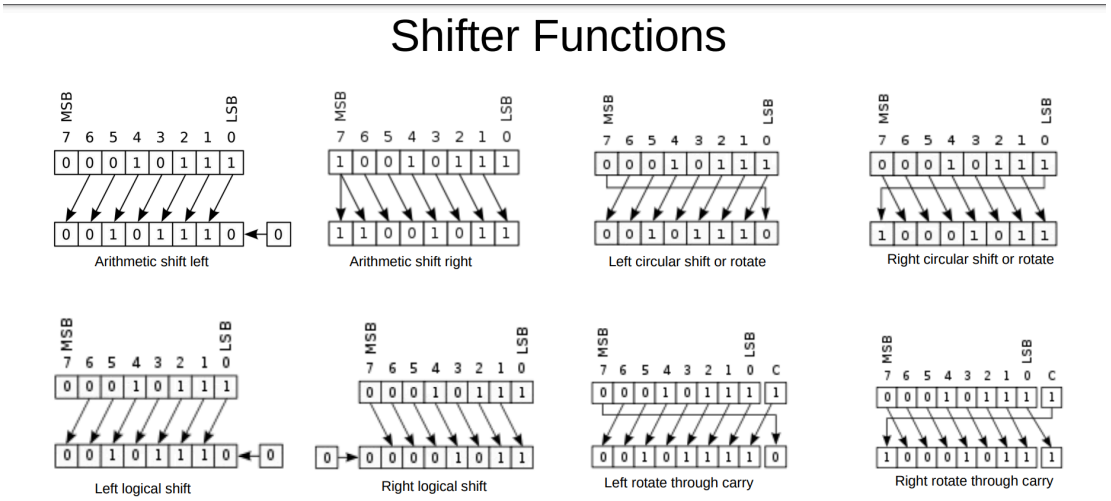
https://docs.google.com/spreadsheets/d/1LeLxZPGiMB_ZDZggbZp3P7fK516pXYhVgZA__djNkWM/edit?gid=0#gid=0

ARM- ASSEMBLY LANG
MANI THINGS: AVR PROG...
all register

https://eng.libretexts.org/Bookshelves/Computer_Science/Programming_Languages/Introduction_to_Assembly_Language_Programming:_From_Soup_to_Nuts:_ARM_Edition_(Kann)/04:_New_Page/4.04:_New_Page

cpu architecture - What does a 'Split' cache means. And how is it useful(if it is)? - Stack Overflow

1. ALU logical operation: AND, OR, XOR, etc
2. ALU arithmetic operation: ADD, MUL, ETC



Memory

- **PROM** is programmed by the **user** and cannot be altered after programming.

- **EPROM** can only be fully erased (not selectively by byte) and reprogrammed.
- **EEPROM** allows selective erasure and rewriting of individual bytes.
- **SRAM** is constructed using **flip-flops**, and **DRAM** is made using **capacitors**.
- **SRAM** is faster and doesn't need refreshing, while **DRAM** is cheaper and has higher storage density.
- **SRAM** is volatile but can be made non-volatile with a **battery backup**.

EU

1. **What is the Instruction Address Generator (IAG)?**

The IAG is the engine or logic responsible for determining the next instruction's memory address. It keeps track of program flow and updates the **Program Counter (PC)**, which always holds the memory address of the instruction to be executed next.

- In **sequential execution**, the PC just moves to the next instruction in memory, typically by adding a fixed value (like 4 bytes for each instruction).
- In **non-sequential execution** (e.g., jumps, branches, subroutine calls), the PC needs to be updated to point to a different location in memory.

2. **Program Flow in Normal and Jump Conditions**

- **Normal Execution**: After executing an instruction, the PC is updated to the next instruction address by adding a fixed value (usually 4 for 32-bit instructions).
- **Jump/Branch Execution**: When a jump or branch occurs, the PC needs to be updated to a new address instead of simply moving to the next instruction. This address could be based on:
  - A **branch offset** (given in the instruction),
  - A **fixed jump** (like in subroutine calls),
  - Or other control signals (e.g., interrupts, function calls).

3. **How Does the IAG Work?**

The IAG performs its task using the following components:

- **Adder**: Adds two inputs to determine the next address.

- **PC**: One of the adder inputs is always connected to the current value of the PC, which is the address of the current instruction.
- **Multiplexer (MuxINC)**: This multiplexer selects the second input to the adder. It can choose between:
    1. **Constant 4** (for normal sequential execution),
    2. **Branch offset** (for jump/branch instructions).
       The branch offset comes from the
       immediate field in the **Instruction Register (IR)** and is sign-extended to 32 bits.
- **PC Update Mechanism**: The result of the addition is either the next instruction (PC + 4) or a new address (PC + branch offset), and this result can be written back to the PC via another multiplexer.

4. **MuxPC and Subroutine Handling**

- **MuxPC**: This multiplexer selects whether to update the PC with the value from the adder (i.e., the next instruction address) or from a **register RA** (for subroutine calls).
- **Register RA**: When a subroutine is called, the PC needs to jump to the starting address of the subroutine. This starting address is stored in RA.
- **Subroutine Linkage**: When jumping to a subroutine, the current PC (which points to the calling instruction) is saved in a temporary register (PC-Temp). This allows the program to return to the correct location after the subroutine finishes.

5. **Return from Subroutine**

- The return address from the subroutine (stored in PC-Temp) is typically managed via the **stack** in modern microprocessors. The **stack pointer** holds the return address, and when the subroutine finishes, the PC is restored to the return address.

6. **Control Signals**

- **INC_select**: This signal controls the MuxINC to select either a fixed constant (4) or the branch offset, depending on whether the program is continuing sequentially or branching.
- **PC_select**: This signal controls MuxPC, deciding whether to update the PC from the adder (normal program flow) or from the value in RA (when

handling subroutines or interrupt service routines).

- **PC_enable**: This signal enables the loading of the new address into the PC.

Summary:

✅

- The **IAG** updates the PC during normal and jump conditions.
- It uses an **adder** and a series of multiplexers to either add a constant value (for sequential execution) or a branch offset (for jump instructions).
- The **PC** can be updated either with the next sequential address or an address from **register RA** (for subroutine calls).
- **Control signals** (INC_select, PC_select, and PC_enable) guide how the IAG behaves in different situations.

🤠CONTROL UNIT

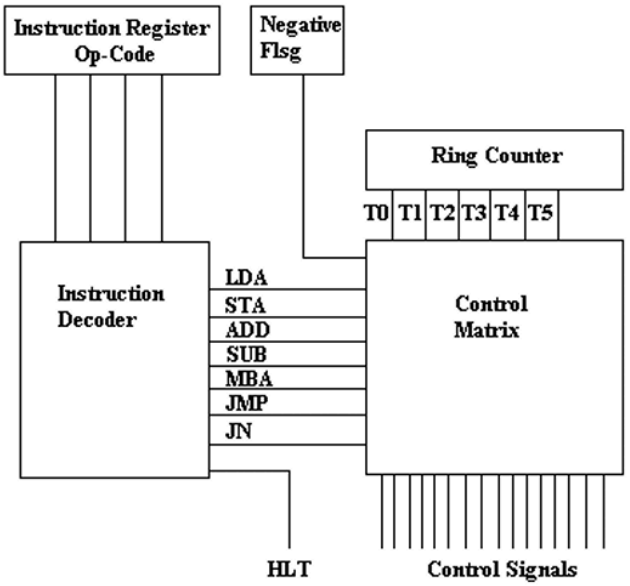
Sequence Breakdown:

1. **Fetch Instruction**:
   - The **PC** provides the address of the next instruction.
   - This address is sent to the **MAR**, which is then used by the memory system to fetch the instruction.
2. **Execute Instruction**: Meanwhile, the **PC** may be updated to hold the address of the next instruction.
   - If the instruction involves accessing memory (e.g., a load or store operation), the **AGU** calculates the **effective address** of the data operand.

- This calculated address is sent to the **MAR**, and memory accesses the data based on this address.

Hardwired Control vs. Microprogrammed Control:

- **Hardwired Control**: Involves fixed logic circuits and combinational logic to generate control signals. It is fast but difficult to modify.
- **Microprogrammed Control**: Uses a sequence of microinstructions stored in memory (control store) to generate control signals. This approach is slower but more flexible and easier to update

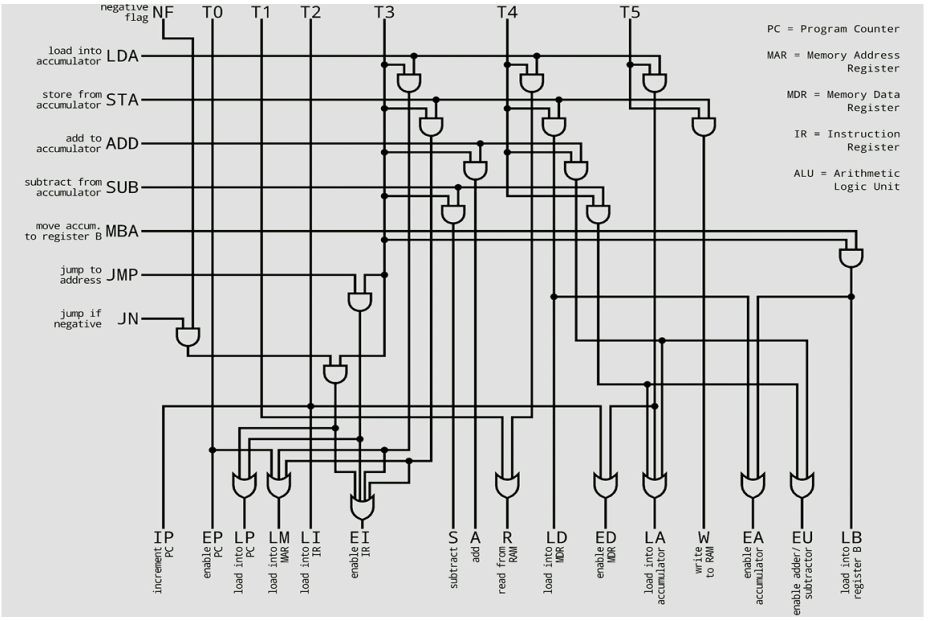


Constant length OP-Codes are assumed

HARDWIRED CONTROL UNIT

Hardwired Control Unit

Matrix part



✅

- **MAR** acts as the interface between the processor and memory during read or write operations.

Sequence Breakdown:

1. **Fetch Instruction**:
   - **PC - - >** next instruction address.
   - This address is sent to the **MAR**, which is then used by the memory system to fetch the instruction.
2. **Execute Instruction**: Meanwhile, the **PC** may be updated to hold the address of the next instruction.
   - If the instruction involves accessing memory (e.g., a load or store operation), the **AGU** calculates the **effective address** of the data operand.
   - This calculated address is sent to the **MAR**, and memory accesses the data based on this address.

TUT-3

1. This problem is about the addressing modes. Given the following:
   (a) Each mode eld is 8 bits
   (b) Internal memory 32, 8-bit registers
   (c) 1 kbyte of main memory with word size 2 bytes
   compute,
   (a) how many distinct unique memory locations, each of the following ve type of addressing, can
   address?
   i. immediate
   ii. direct
   iii. indirect
   iv. direct register
   v. indirect register
   (b) the range of values of operands each of the above addressing could handle

i. NILL | 2^8
ii. 2^8 | 2^16 [only dependent on word size]
iii. 2^9 | 2^16
iv. 32 | 2^8
v. 2^8 | 2^16

[generally, it should have been 2^5 but these 2^5 location have 8- bit address and these 8 bits can point to 2^8 unique loc]

✅ FLOATING POINT REPRES.

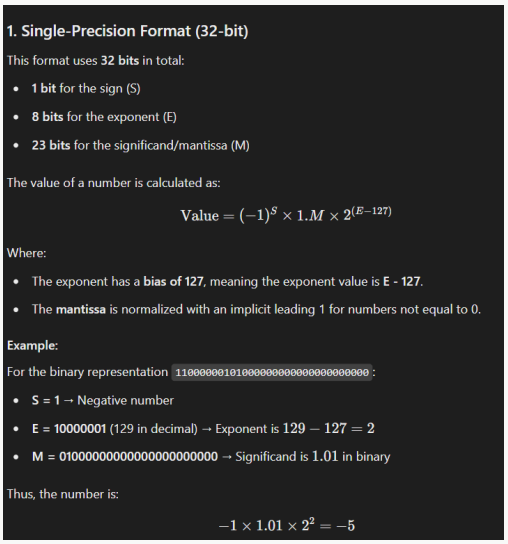
In normalized floating-point representation, the leading bit of the mantissa is always assumed to be 1

32 BIT —→ 1- 8E - 23M

- 127 bias

64 BIT —→ 1 - 11E - 52 M

- 1023 bias



1. Single-Precision Format (32-bit)

This format uses **32 bits** in total:

- **1 bit** for the sign (S)
- **8 bits** for the exponent (E)
- **23 bits** for the significand/mantissa (M)

The value of a number is calculated as:

$$\text{Value} = (-1)^S \times 1.M \times 2^{(E-127)}$$

Where:

- The exponent has a **bias of 127**, meaning the exponent value is **E - 127**.
- The **mantissa** is normalized with an implicit leading 1 for numbers not equal to 0.

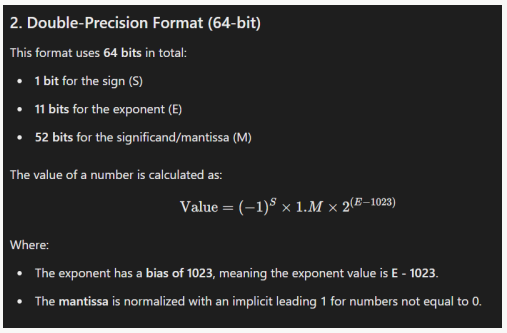
Example:

For the binary representation `11000000101000000000000000000000`:

- **S = 1** → Negative number
- **E = 10000001** (129 in decimal) → Exponent is $129 - 127 = 2$
- **M = 01000000000000000000000** → Significand is 1.01 in binary

Thus, the number is:

$$-1 \times 1.01 \times 2^2 = -5$$



2. Double-Precision Format (64-bit)

This format uses **64 bits** in total:

- **1 bit** for the sign (S)
- **11 bits** for the exponent (E)
- **52 bits** for the significand/mantissa (M)

The value of a number is calculated as:

$$\text{Value} = (-1)^S \times 1.M \times 2^{(E-1023)}$$

Where:

- The exponent has a **bias of 1023**, meaning the exponent value is **E - 1023**.
- The **mantissa** is normalized with an implicit leading 1 for numbers not equal to 0.

PERIPHERALS

Port Mapped I/O:

- **Dedicated I/O instructions**: Uses specific CPU instructions like `IN` and `OUT` to communicate with I/O devices.
- **Separate address space**: I/O devices have their own address space, distinct from the system's main memory.
- **Example**: Common in Intel x86 and x86-64 processors. Uses

Memory Mapped I/O:

- **Shared address space**: same address space as main memory, allowing them to be accessed just like memory locations.
- **Fewer CPU instructions**: Uses regular CPU instructions for memory access, making it simpler and faster.
- **Efficient design**: Reduces the complexity and size of the CPU,

instructions like `INS`, `OUTS`, etc., to transfer data between I/O ports and CPU registers.

- **Limited CPU interaction**: Requires multiple instructions for simple operations (e.g., adding a constant to a port involves reading, modifying, and writing back).
- **Isolated I/O**: Sometimes referred to as "isolated I/O" because the I/O and memory spaces are separate.

making it cheaper, faster, and more power-efficient.

- **Common in embedded systems**: Found in systems like AVR, ARM, and most microcontrollers, following the principles of RISC.
- **Versatility**: All addressing modes for memory can also be used for I/O.

ADV:

- less logic

DISADV:

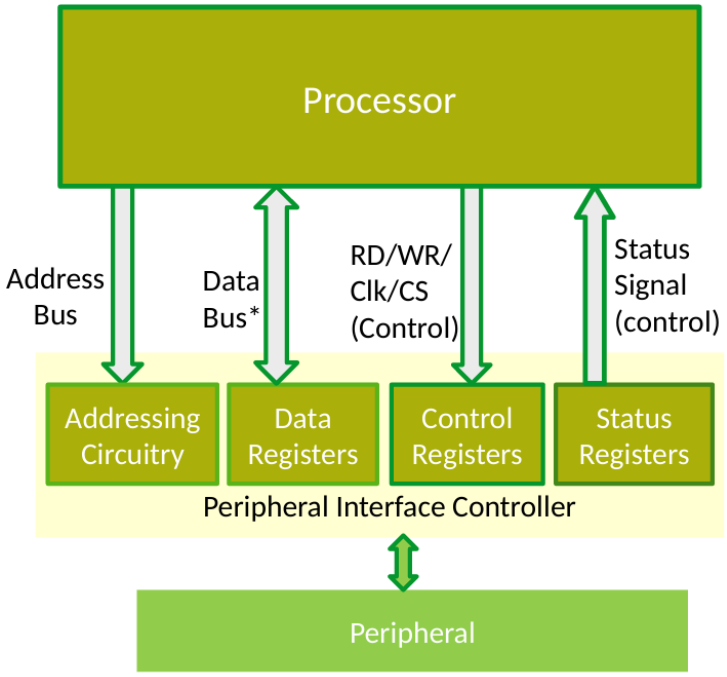
- More instructions are needed for simple tasks

From the processor point of view–

PIC are a bunch of registers which can be read or written to–

When?

- Polling or Interrupt



- Status — processor always read
- control reg — pro. always writes
- data reg — to send or receive
- Keyboard controller — column data reg, row data reg → concatenates them by DEVICE DRIVER

polling

- Average typing speed → 12 stroke/key press per sec
- very high speed of 50 strokes /sec
- Assume a minimum of 7 msec press time and 13 msec before another key is pressed

|

Assume a processor reads keyboard registers every 1 msec

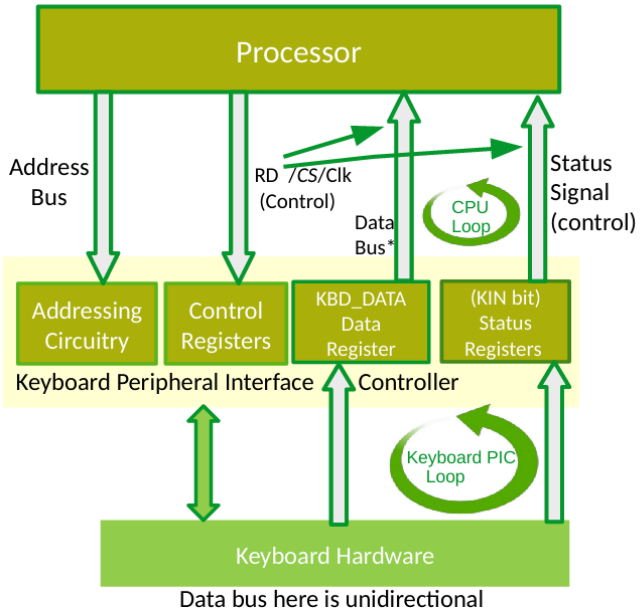
|

At least 7 successive ONES for key presses and at least 13 successive ZEROES before next key is pressed

- Keyboard pic reg

	FUNCT	

KBD_DATA	hold ascii of keypress	sends actual data
KBD_STATUS	KIN (input): 1 if key_pre not read	If it is still `1`, it means the data in `KBD_DATA` is valid
	KIRQ (INTR req): 1 if intr raised.	Indicates to the CPU that there's pending work
KBD_CONT	KIE (intr enable): 1-enable intr	CPU will handle the keyboard through interrupts rather than polling



INTERRUPT

- handle external event through Interrupt subroutine (ISR) → reads status / data registers of PIC
- Interrupt Controller chip (like a Peripheral Controller) sometime used to connect multiple interrupts → define priorities for each interrupt and turn on / off any Interrupt

INTR

- CPU does if intr enabled
- more time, power

peripheral → processor → memory

DMA

- CPU grants permission to direct access memory
- less time

peri → memory

- CPU → internal bus | DMA → system bus

▼ INTERRUPT PROCESS

The explanation you provided covers the concept of interrupts in microprocessor systems, specifically focusing on how interrupts are handled when a peripheral requests the processor's attention. Let's break down this concept in more detail, with an emphasis on the steps involved and the reasons behind each one:

Interrupt Handling Process:

Interrupts are essential in microprocessors and microcontrollers because they allow the system to respond to events as they occur, without needing to constantly check for those events (polling). When an interrupt is triggered, the microprocessor temporarily halts its current operations to attend to the interrupt, ensuring that time-sensitive tasks are handled promptly. The process of handling an interrupt involves several key steps:

1. Interrupt Request (IRQ) Signal:

- **What Happens:** A device or peripheral sends a signal to the microprocessor, indicating that it requires attention. This is the interrupt request (IRQ) signal.
- **Why It's Important:** The IRQ signal acts as a notification that an event requiring immediate attention has occurred, such as data being ready to read from a sensor, a timer overflow, or an external button press.

2. Current Instruction Completion:

- **What Happens:** The microprocessor does not immediately jump to the interrupt handler but finishes executing the current instruction in its pipeline.
- **Why It's Important:** Microprocessors execute instructions in a pipeline, and interrupting this process midway could cause data corruption or unexpected behavior. The microprocessor ensures that the current instruction is completed properly before handling the interrupt.

3. Interrupt Acknowledgment:

- **What Happens:** After completing the current instruction, the microprocessor acknowledges the interrupt. This acknowledgment might involve sending a signal to the interrupting device, informing it that the interrupt has been received and is being processed.
- **Why It's Important:** Acknowledging the interrupt confirms that the microprocessor is now aware of the interrupt request and will proceed to process it. This also prevents the interrupting device from repeatedly sending interrupt signals if it's waiting for confirmation.

4. Interrupt Vector:

- **What Happens:** The microprocessor identifies the specific interrupt source. Each interrupt type (whether it's from an external device or an internal event) has a unique memory address called an interrupt vector, which points to the start of the Interrupt Service Routine (ISR).
- **Why It's Important:** The interrupt vector ensures that the processor knows exactly where to go to find the appropriate ISR. Different interrupts might be associated with different vectors, so the system can handle multiple interrupts in a structured way.

5. Context Saving:

- **What Happens:** Before the microprocessor jumps to the ISR, it saves its current state, which typically includes the values of the Program Counter (PC), registers, and other critical state information.
- **Why It's Important:** Saving the context ensures that the microprocessor can return to the exact state it was in when the interrupt is over. This step is crucial because the interrupt could change important data or registers, and without saving the state, the processor would lose its place in the original program.

6. Jump to ISR:

- **What Happens:** After saving the context, the microprocessor jumps to the address specified by the interrupt vector and begins executing the ISR.
- **Why It's Important:** This step directs the processor to the correct location in memory to execute the code that will handle the interrupt.

The ISR contains the instructions necessary to address the interrupting event (e.g., reading data from a sensor or resetting a flag).

7. Execution of ISR:

- **What Happens:** The microprocessor executes the instructions in the ISR. This code is responsible for handling the interrupt, which could involve reading or writing data, resetting hardware flags, or performing any action required by the interrupting device.
- **Why It's Important:** The ISR ensures that the event that triggered the interrupt is properly handled. It might involve simple tasks like clearing an interrupt flag or more complex operations like communicating with external peripherals.

8. Context Restoration:

- **What Happens:** After the ISR completes, the microprocessor restores the saved context, which includes restoring the program counter (PC) and the state of the registers.
- **Why It's Important:** Restoring the context is essential for returning the processor to its exact state before the interrupt occurred. Without this step, the processor would lose its place in the main program, leading to errors or undefined behavior.

9. Return from Interrupt (RETI):

- **What Happens:** The microprocessor executes a special instruction, such as "RETI" (Return from Interrupt), which causes the processor to pop the saved context from the stack and resume execution from the point where the interrupt occurred.
- **Why It's Important:** The RETI instruction ensures that the processor can resume normal operation after handling the interrupt. It also signals the end of the interrupt handling process.

10. Continuation of Execution:

- **What Happens:** The microprocessor continues executing the main program, picking up right where it left off before the interrupt was triggered.

- **Why It's Important:** This allows the main program to proceed as if the interrupt had not occurred, with all registers and the program counter restored to their prior values. The system continues running smoothly without any loss of data or functionality.

Interrupt Handling and Interrupt Vector Table:

An interrupt vector table is a data structure in memory that holds the addresses of all the interrupt service routines (ISRs) for the various interrupts. Each interrupt type has a corresponding entry in this table, which the microprocessor uses to quickly locate the correct ISR when an interrupt occurs. The vector table is typically set up during the initialization of the system.

Types of Interrupts:

Interrupts can be triggered by different sources:

- **External Interrupts**: These come from hardware peripherals, like sensors, timers, or external devices connected to the microprocessor.
- **Internal Interrupts**: These can be generated by the microprocessor itself in response to internal events, such as exceptions (e.g., division by zero) or timers reaching specific values.
- **Software Interrupts**: These are generated by software (e.g., system calls in an operating system).

Summary:

Interrupts allow a microprocessor to handle asynchronous events efficiently. The interrupt handling process ensures that the microprocessor can attend to these events without wasting processing time on polling. Each step, from the interrupt request to the return from the ISR, ensures that the microprocessor's state is preserved and that the appropriate action is taken in response to the interrupt, allowing the system to continue its normal operations seamlessly.

WHOLE SUMM

✅ **Sequence in Summary:**

1. Data fetched → MBR.
2. Instruction decoded by CU.
3. If operation → ALU, perform calculation.
4. If I/O → Data sent to or received from I/O device.
5. Result stored back in memory or a register.

AVR ASSEMBLY LANG

- NOP ⟶ No operation ⟶do nothing for one clock cycle but pc++ ⟶ move to the next instruction.
- ADC —— So if there was a carry from adding the LSBs, it will automatically be included in the addition of the MSBs.
- PUSH — pushes the current value into the stack.

  POP — retrieves the value from stack after interrupt.
- `BRGE` (Branch if Greater or Equal) checks if the result is **greater than or equal to zero**.

  `BRNE` (Branch if Not Equal).
- `BRCC` (Branch if No CARRY)

▼ what's the difference between load - lpm when we tell LPM R16, Z+ and store st something...

1. Loading Data from Flash vs. SRAM

- `LPM` **(Load Program Memory)**:
  - `LPM` is used to load data from **Flash memory** (program memory) into a register.
  - This is important because in AVR, Flash memory is read-only during runtime, so you need `LPM` to read constants or data tables stored there.
  - **Example:** `LPM R16, Z+` means "load the byte at the program memory address pointed to by `Z` into `R16`, then increment `Z`." This is helpful for reading consecutive bytes in Flash.
- `LD` **(Load Direct from Data Space)**:
  - `LD` is used to load data from **SRAM (data memory)**, which is where variable data and temporary values are stored during runtime.
  - You can think of `LD` as loading values from RAM, where your program stores data dynamically.

2. Storing Data in SRAM

- `ST` **(Store to SRAM)**:
  - `ST` writes data from a register into **SRAM**.
  - This instruction is the reverse of `LD`, where you're taking a register's value and saving it to a specific address in SRAM.
  - **Example:** `ST X+, R16` stores the content of `R16` to the address in `X`, then increments `X` for the next store operation.

Key Differences Between `LPM`, `LD`, and `ST`

Instruction	Source	Destination	Use Case
`LPM`	Flash memory	Register	Reading constants from Flash
`LD`	SRAM	Register	Loading data from SRAM

`ST`	Register	SRAM	Storing data to SRAM

Why Use `LPM` Instead of `LD` for Flash?

Flash memory and SRAM are located at different addresses and serve different purposes in AVR architecture. `LPM` allows you to access Flash (read-only) for constants and lookup tables, while `LD` and `ST` are for RAM-based data.

Example to Illustrate:

Let's say you want to:

1. Read a constant stored in Flash memory (`NUM`) and add it to another number.
2. Store the result in SRAM for later use.

Here's what the code might look like:

```
; Setup `Z` to point to Flash address of NUM
LDI ZL, LOW(NUM<<1)
LDI ZH, HIGH(NUM<<1)

; Load from Flash to register
LPM R1, Z+      ; Load the first byte of `NUM` into R1 from Flash
LPM R2, Z       ; Load the next byte of `NUM` into R2 from Flash

; Perform some operations (e.g., add)
ADD R1, R2      ; Add the two bytes together, result in R1

; Store result to SRAM
LDI XL, LOW(0x0060)
LDI XH, HIGH(0x0060)
ST X, R1        ; Store the result from R1 into SRAM at 0x0060
```

- `LPM` reads from Flash into `R1` and `R2`.
- `ST` stores the computed result in SRAM.

This illustrates how `LPM` is crucial for Flash data access, while `ST` and `LD` are suited for working with SRAM.

▼ IS Storing in SRAM is necessary

- **Storing in SRAM** is necessary if you want to preserve the result for future use or if you're dealing with a multi-step operation. If your task only requires the immediate result (e.g., to display or use in the next step), then you could bypass storing it in memory and simply keep it in registers.

▼ DDRX , PORTX, PINX

- DDRX — Direction input or output
- PORTX — display high or low value in assigned output pin
- PINX — read pin input

✅ DDRX , PORTX, PINX

The code you've provided involves working with the **PORTB**, **DDRB**, and **PINB** registers on an AVR microcontroller. Let's break down what each of these registers does and how the code interacts with them.

Code Breakdown:

```
LDI R16, 0x01       ; Load the value 0x01 into register R16
OUT DDRB, R16       ; Output the value in R16 to the DDRB regi
ster

LDI R16, 0x01       ; Load the value 0x01 into register R16 again
OUT PORTB, R16      ; Output the value in R16 to the PORTB re
gister
```

DDRB (Data Direction Register for PORTB):

- **Purpose**: Determines the direction of the pins on PORTB. Each bit in the DDRB register corresponds to a pin on PORTB.
  - **1** = Output (the pin will be used as an output).
  - **0** = Input (the pin will be used as an input).
- **In the code**:
  - `LDI R16, 0x01` loads the value `0x01` (which is binary `00000001`) into register `R16`.
  - `OUT DDRB, R16` writes this value to `DDRB`. This means **pin 0** of PORTB is set as an output, and all other pins on PORTB remain as inputs (since the corresponding bits are `0`).

PORTB (Data Register for PORTB):

- **Purpose**: Determines the value that is output on the pins of PORTB when those pins are set as output. If a pin is set as an output, writing to PORTB will drive the pin high (1) or low (0).
  - **1** = High (logic level 1) on the corresponding pin.

    - **0** = Low (logic level 0) on the corresponding pin.
- **In the code**:
    - `LDI R16, 0x01` loads the value `0x01` (binary `00000001`) into register `R16`.
    - `OUT PORTB, R16` writes this value to `PORTB`. This means **pin 0** of PORTB is driven high (`1`), and all other pins remain low (`0`).

PINB (Input Pins Register for PORTB):

- **Purpose**: It reads the current state of the pins on PORTB. For input pins, this register reflects the logic level present on the pin (high or low). For output pins, it reflects the last value written to PORTB.
    - **1** = Logic high (1) on the pin.
    - **0** = Logic low (0) on the pin.

However, **PINB** isn't directly involved in the code you provided, so we don’t need to consider it here for this question.

Summary of What Happens to PORTB, DDRB, and PINB:

1. **DDRB** is configured such that **pin 0** of PORTB is set to output (`1`), while all other pins on PORTB are set to input (`0`).
2. **PORTB** is set such that **pin 0** is driven high (`1`), and all other pins are driven low (`0`).
3. **PINB** is not explicitly used in this code, but if it were to be read, it would reflect the current state of the pins, with **pin 0** showing high (`1`) and the rest showing low (`0`), based on the output written to PORTB.

Answer to Your Question:

Yes, **DDRB** determines the direction (input or output) of each pin, and **PORTB** determines the value (high or low) on the output pins. In this case, **PORTB** will "display" or output the value `0x01` on pin 0 (high, or logic 1), while **DDRB** will configure pin 0 as an output. The

role of **PORTB** is to set the output value, while **DDRB** configures the direction.

▼ DIFFERENCE BETWEEN SBI AND OUT DDRX

- DDRX — output or input
- SBI — if ddrx in output sets a particular pin eg. B2 ,3,4
- CBI — for input

✅ The `SBI` (Set Bit in I/O Register) instruction directly sets specific bits in a register, offering a shorthand for bit manipulation.

Comparing `OUT DDRx, R16` and `SBI`:

- `OUT DDRx, R16`: Configures pin direction (input/output) in the DDRx register.
- `SBI`: Sets a bit in registers like PORTx or PINx, modifying pin state without changing direction. assuming you have already set the pin as output

Using `SBI` and `CBI` Without `OUT DDRx`:

- Effective only when the pin is already configured as an output.
- No meaningful effect on input pins.

Example: Setting Pin 2 with `SBI`:

1. Configure as output:

```
LDI R16, 0x04
OUT DDRC, R16
```

2. Turn on:

```
SBI PORTC, 2
```

In summary: `OUT DDRx, R16` sets pin direction, while `SBI PORTx, bit_position` sets the pin state for outputs. `SBI` affects PORTx only after DDRx configuration.

▼ What Happens if `RET` is Used Instead of `RETI`:

- `RET` **(Return from Subroutine)**: When executed, `RET` just returns from the function (subroutine). If used in an interrupt service routine (ISR), it would **not properly clear the interrupt flag**, meaning the interrupt would not be acknowledged, and subsequent interrupts may not be handled.

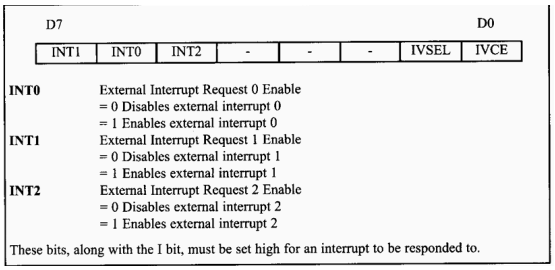
- `RETI` **(Return from Interrupt)**: This instruction not only returns from the ISR but also clears the interrupt flag and enables further interrupts. This is necessary for proper interrupt handling in AVR. If `RET` were used instead of `RETI`, the interrupt flag would not be cleared, leading to potential interrupt handling issues.

▼ interrupt address

For the ATmega32, the external interrupt vector table is defined as follows:

bitwise in GICR - General Interrupt Control Register

1. **INT0**: Address `0x0002` — bit 6
2. **INT1**: Address `0x0004` — bit 7
3. **INT2**: Address `0x0006` — bit 8



D7							D0
INT1	INT0	INT2	-	-	-	IVSEL	IVCE

INT0 External Interrupt Request 0 Enable
= 0 Disables external interrupt 0
= 1 Enables external interrupt 0

INT1 External Interrupt Request 1 Enable
= 0 Disables external interrupt 1
= 1 Enables external interrupt 1

INT2 External Interrupt Request 2 Enable
= 0 Disables external interrupt 2
= 1 Enables external interrupt 2

These bits, along with the I bit, must be set high for an interrupt to be responded to.

▼ interrupt register

`MCUCR` (Microcontroller Control Register) configures settings in AVR microcontrollers, including external interrupt behavior and sleep modes. For external interrupts, it controls triggering conditions for `INT0` and `INT1`.

Key Bits in `MCUCR` for External Interrupts

In ATmega32, `MCUCR` sets trigger conditions for `INT0` and `INT1` pins:

- **ISC00 and ISC01**: Control `INT0` triggering
  - `00` – Low level
  - `01` – Any logical change
  - `10` – Falling edge
  - `11` – Rising edge
- **ISC10 and ISC11**: Control `INT1` triggering (same options as above)

Example: Configuring `INT1` for Rising Edge

```
LDI R16, (1 << ISC11) | (1 << ISC10)
OUT MCUCR, R16
```

`MCUCR` also controls sleep mode configuration, though less relevant for basic interrupt setups.

`ISC11` and `ISC10` are bits in the `MCUCR` of AVR microcontrollers that control `INT1` interrupt behavior. They set the trigger conditions for `INT1` as follows:

- **00**: Low level
- **01**: Any logical change
- **10**: Falling edge
- **11**: Rising edge

These settings allow precise control over interrupt timing. To set `INT1` for rising edge detection:

```
LDI R16, (1 &lt;&lt; ISC11) | (1 &lt;&lt; ISC10)
OUT MCUCR, R16
```

Yes, in addition to `MCUCR`, there are several other important registers related to interrupts in AVR microcontrollers (like ATmega32). These registers control interrupt enabling, configuration, and flags. Here are some key interrupt-related registers:

1. MCUCR (Microcontroller Control Register)

Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- Controls interrupt sense control for external interrupts `INT0` and `INT1` (like `ISC11`, `ISC10` for `INT1`).
- Sets how interrupts should be triggered (low level, logical change, falling edge, or rising edge).

2. MCUCSR (Microcontroller Control and Status Register)

- Contains the `ISC2` bit, which controls the sense control for `INT2` (on the falling or rising edge).
- Holds flags for reset sources (like external reset, power-on reset, etc.).

3. GICR (General Interrupt Control Register)

Bit	7	6	5	4	3	2	1	0
	INT1	INT0	INT2	–	–	–	IVSEL	IVCE
Read/Write	R/W	R/W	R/W	R	R	R	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

- Used to enable external interrupts `INT0`, `INT1`, and `INT2`.
- Bits `INT0`, `INT1`, and `INT2` are set to 1 to enable the respective interrupts.
- Example:

```
LDI R16, (1 << INT1)     ; Enable INT1
OUT GICR, R16            ; Write to GICR
```

4. GIFR (General Interrupt Flag Register)

- Contains flags for external interrupts `INT0`, `INT1`, and `INT2`.
- These flags are set when an interrupt is triggered and cleared when the corresponding interrupt vector is executed.
- Bits `INTF0`, `INTF1`, and `INTF2` represent the flags for `INT0`, `INT1`, and `INT2`, respectively.

5. SREG (Status Register)

- Holds the `I` (Global Interrupt Enable) bit, which enables or disables all interrupts globally.
- To enable global interrupts, you need to set the `I` bit (using the `SEI` instruction).
- To disable global interrupts, clear the `I` bit (using the `CLI` instruction).
- Example:

```
SEI     ; Enable global interrupts
CLI     ; Disable global interrupts
```

✅ **Summary of Key Interrupt Registers**

- **MCUCR**: Controls interrupt sensing for `INT0` and `INT1`.
- **MCUCSR**: Controls interrupt sensing for `INT2` and holds reset flags.
- **GICR**: Enables or disables `INT0`, `INT1`, and `INT2`.
- **GIFR**: Holds the interrupt flags for `INT0`, `INT1`, and `INT2`.
- **SREG**: Global interrupt enable/disable via the `I` bit.

These registers allow fine control over the triggering, enabling, and handling of interrupts in AVR microcontrollers.

Vector No.	Program Address[2]	Source	Interrupt Definition
1	$000[1]	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$002	INT0	External Interrupt Request 0
3	$004	INT1	External Interrupt Request 1
4	$006	INT2	External Interrupt Request 2
5	$008	TIMER2 COMP	Timer/Counter2 Compare Match
6	$00A	TIMER2 OVF	Timer/Counter2 Overflow
7	$00C	TIMER1 CAPT	Timer/Counter1 Capture Event
8	$00E	TIMER1 COMPA	Timer/Counter1 Compare Match A
9	$010	TIMER1 COMPB	Timer/Counter1 Compare Match B
10	$012	TIMER1 OVF	Timer/Counter1 Overflow
11	$014	TIMER0 COMP	Timer/Counter0 Compare Match
12	$016	TIMER0 OVF	Timer/Counter0 Overflow
13	$018	SPI, STC	Serial Transfer Complete
14	$01A	USART, RXC	USART, Rx Complete
15	$01C	USART, UDRE	USART Data Register Empty
16	$01E	USART, TXC	USART, Tx Complete
17	$020	ADC	ADC Conversion Complete
18	$022	EE_RDY	EEPROM Ready
19	$024	ANA_COMP	Analog Comparator
20	$026	TWI	Two-wire Serial Interface
21	$028	SPM_RDY	Store Program Memory Ready

▼ SUMMARY

✅ SUMMARY

1. Interrupts

- **INT0, INT1, INT2**: External interrupts where INT0 and INT1 use MCUCR to set interrupt sense (ISCxx bits) for various edge types. For INT2, instead of MCUCR, **MCUCSR** is used, specifically ISC2 (since it's a single bit) to configure the edge (falling or rising).
- **Global Interrupt Enable (SREG Register)**: The `SEI` instruction enables global interrupts, while `CLI` disables them.
- **Interrupt Service Routine (ISR)**: Use `PUSH` and `POP` to save and restore `SREG` in the ISR to preserve the status register.

2. Registers for I/O Configuration

- **DDRx**: Data Direction Register for setting pin as input or output.
- **PORTx**: Used for setting high/low voltage on a port.
- **PINx**: Used to read the status of a pin.

3. Timers and Counters

- **TCCR0, TCNT0, TIMSK**: Basic control registers for Timer 0 (similar for Timer 1, 2) to set modes, enable interrupts, and configure prescalers.
- **Overflow Handling**: Timers can trigger interrupts on overflow, controlled by setting bits in TIMSK.

4. Bitwise Operations and Commands

- **ORI, ANDI, SBI, CBI**: For manipulating bits. Example: `ORI R16, (1 << ISC01) | (1 << ISC00)` sets the ISC01 and ISC00 bits in `R16`.
- **SBI (Set Bit in I/O Register) and CBI (Clear Bit in I/O Register)**: Handy for toggling bits directly without changing other bits in the register.

5. Stack Operations

- **PUSH and POP**: Essential for saving/restoring registers (especially `SREG`) in interrupts or subroutine calls.
- **Stack Pointer (SP)**: Set up initially with `SPL` and `SPH` to ensure a stack exists in memory.

6. Delays

- Nested loops (like in your code) create delays based on counter decrements. This is an essential concept for timing without timers.

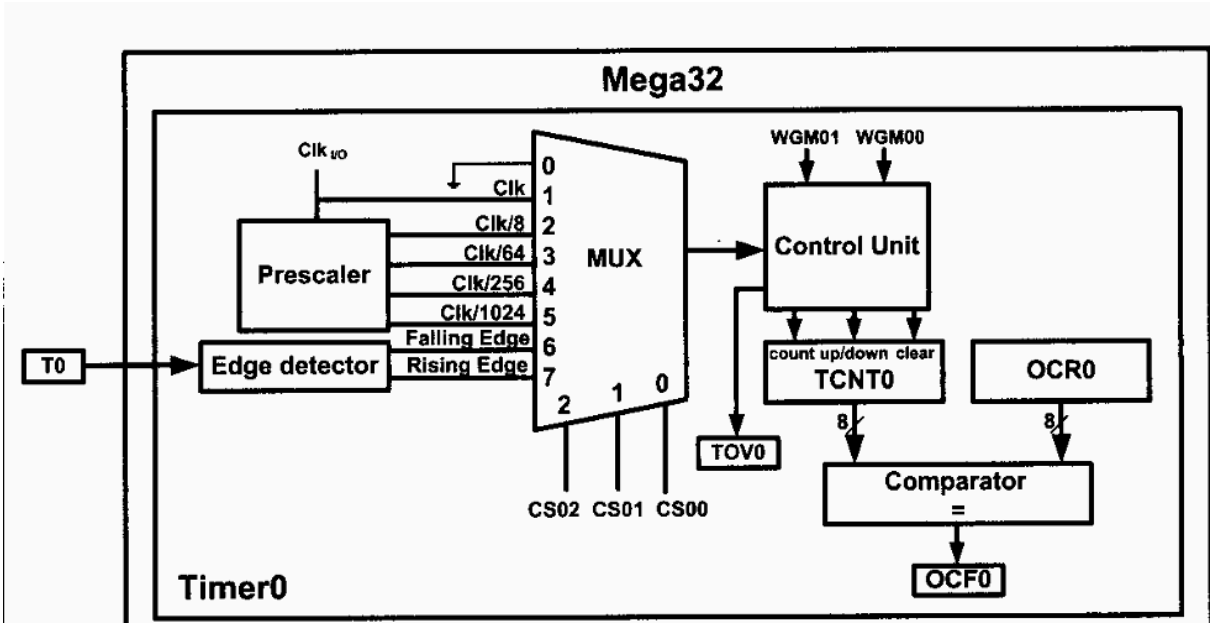
7. Common Pitfalls

- Forgetting to enable global interrupts (`SEI`) after configuring interrupts.
- Not preserving `SREG` in ISR with `PUSH` and `POP`.
- Misconfiguring pin direction in DDRx for intended input/output use.

ARM- ASSEMBLY LANG

- `DCB`: Defines data in **bytes** (8 bits).
- `DCW` directive is used to define a **word** (2 bytes)
- `DCD`: Defines data in **double words** (32 bits, or 4 bytes).

MANI THINGS: AVR PROG...



Why Use **0x01** for TCCR0?

In AVR microcontrollers, **TCCR0** is used to configure various aspects of **Timer0**. The relevant bits for setting the prescaler are **CS02**, **CS01**, and **CS00**. Here's how these bits work:

CS02	CS01	CS00	Timer0 Clock Source
0	0	0	Timer stopped
0	0	1	No prescaling
0	1	0	Prescaler = 8
0	1	1	Prescaler = 64
1	0	0	Prescaler = 256
1	0	1	Prescaler = 1024

In this program, **TCCR0 = 0x01** sets **CS00** to `1` and **CS01** and **CS02** to `0`. This corresponds to the **no prescaler** setting.

- if no prescaling → increment 1 clock cycle
- if prescaler == 8 → increment 8 clock cycles.

avr_qs_endsem.pdf

https://prod-files-secure.s3.us-west-2.amazonaws.com/1b1c5101-9651-4202-b9d0-cf7582ac6ad3/478b9ca8-dfe4-42fc-917f-264a93887762/avr_qs_endsem.pdf

avr_9.pdf

https://prod-files-secure.s3.us-west-2.amazonaws.com/1b1c5101-9651-4202-b9d0-cf7582ac6ad3/6b000ca7-dd5c-4613-9533-8bc44a875b29/avr_9.pdf

Example 10-1

Show the instructions to (a) enable (unmask) the Timer0 overflow interrupt and Timer2 compare match interrupt, and (b) disable (mask) the Timer0 overflow interrupt, then (c) show how to disable (mask) all the interrupts with a single instruction.

Solution:

```
(a)   LDI R20,(1<<TOIE0)|(1<<OCIE2) ;TOIE0 = 1, OCIE2 = 1
      OUT TIMSK,R20  ;enable Timer0 overflow and Timer2 compare match
      SEI                           ;allow interrupts to come in

(b)   IN    R20,TIMSK               ;R20 = TIMSK
      ANDI  R20,0xFF^(1<<TOIE0)     ;TOIE0 = 0
      OUT   TIMSK,R20               ;mask (disable) Timer0 interrupt
```

We can perform the above actions with the following instructions, as well:

```
      IN    R20,TIMSK          ;R20 = TIMSK
      CBR   R20,1<<TOIE0       ;TOIE0 = 0
      OUT   TIMSK,R20          ;mask (disable) Timer0 interrupt
                               .
(c)   CLI                      ;mask all interrupts globally
```

Notice that in part (a) we can use "`LDI, 0x81`" in place of the following instruction: "`LDI R20,(1<<TOIE0)|(1<<OCIE2)`"

Example 10-2

What is the difference between the RET and RETI instructions? Explain why we cannot use RET instead of RETI as the last instruction of an ISR.

Solution:

Both perform the same actions of popping off the top bytes of the stack into the program counter, and making the AVR return to where it left off. However, RETI also performs the additional task of setting the I flag, indicating that the servicing of the interrupt is over and the AVR now can accept a new interrupt. If you use RET instead of RETI as the last instruction of the interrupt service routine, you simply block any new interrupt after the first interrupt, because the I would indicate that the interrupt is still being serviced.

Example 10-7

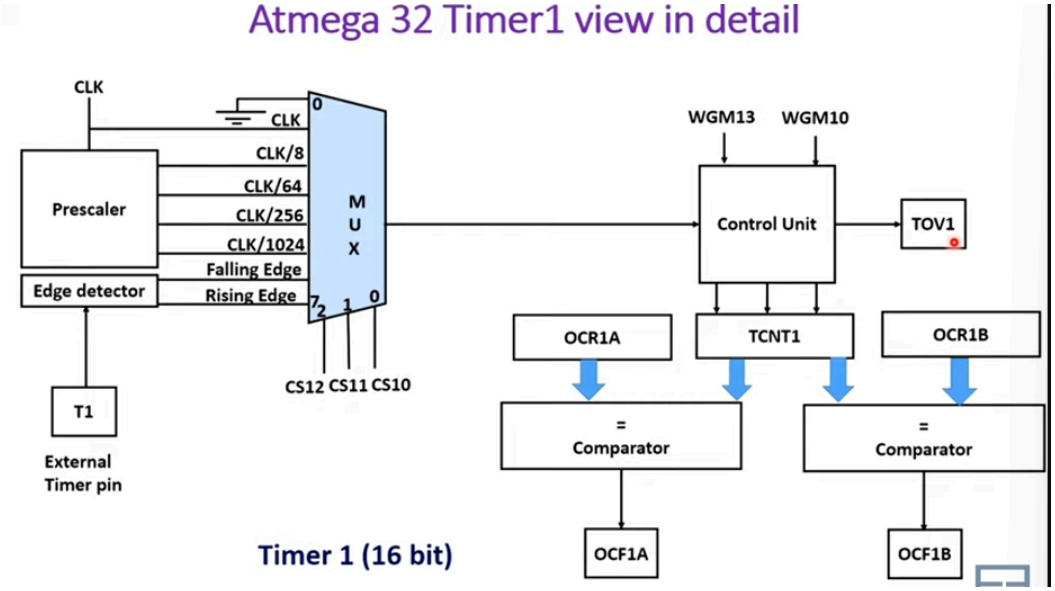
Rewrite Example 10-5, so that whenever INT0 goes low, it toggles PORTC.3 only once.

Solution:

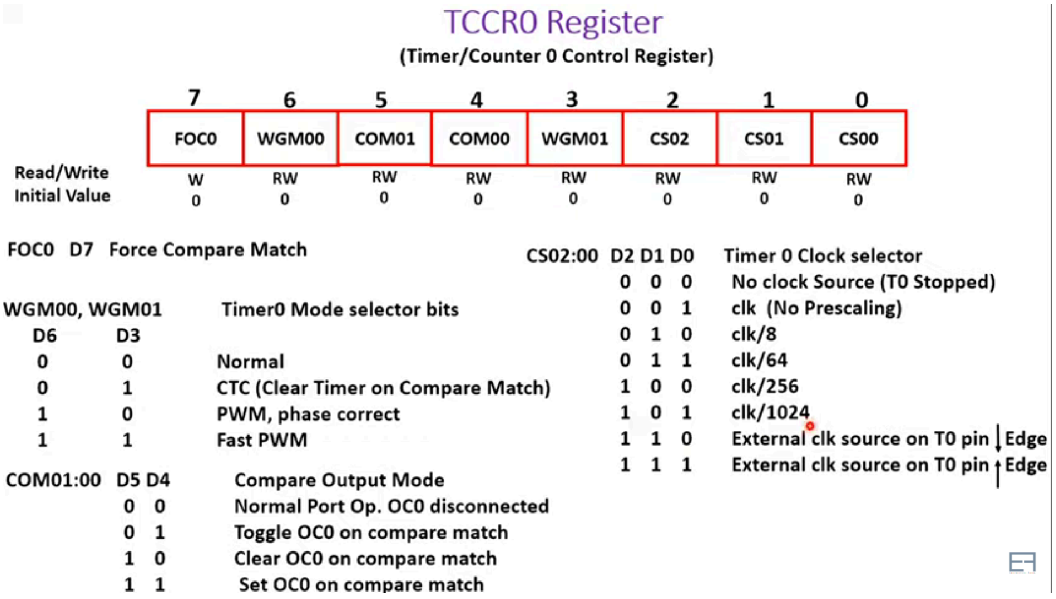
```
.INCLUDE "M32DEF.INC"
.ORG 0                          ;location for reset
      JMP   MAIN
.ORG 0x02                       ;location for external interrupt 0
      JMP   EX0_ISR
MAIN: LDI   R20,HIGH(RAMEND)
      OUT   SPH,R20
      LDI   R20,LOW(RAMEND)
      OUT   SPL,R20             ;initialize stack
      LDI   R20,0x2             ;make INT0 falling edge triggered
      OUT   MCUCR,R20
      SBI   DDRC,3              ;PORTC.3 = output
      SBI   PORTD,2             ;pull-up activated
      LDI   R20,1<<INT0         ;enable INT0
      OUT   GICR,R20
      SEI                       ;enable interrupts
HERE: JMP   HERE
EX0_ISR:
      IN    R21,PORTC
      LDI   R22,0x08            ;00001000 for toggling PC3
      EOR   R21,R22
      OUT   PORTC,R21
      RETI
```

all register

- ▼ timer0
- ▼ timer1



▼ TCCR0



▼ TCCR1A , TCCR1B

WGM13:WGM10

Mode	WGM13	WGM12	WGM11	WGM10	Mode of Operation	TOP	Update of OCR1x	TOV1 Flag Set
0	0	0	0	0	Normal	OxFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	Ox00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	Ox01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	Ox03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM. 8-bit	Ox00FF	TOP	TOP
6	0	1	1	0	Fast PWM, 9-bit	Ox01FF	TOP	TOP
7	0	1	1	1	Fast PWM, 10-bit	Ox03FF	TOP	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	(Reserved)	—	—	—
14	1	1	1	0	Fast PWM	ICR1	TOP	TOP
15	1	1	1	1	Fast PWM	OCR1A	TOP	TOP

TCCR1A and TCCR1B Control Registers

	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	FOC1A	FOC1B	WGM11	WGM10
Read/Write	RW	RW	R	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0

TCCR1A Control Register

	7	6	5	4	3	2	1	0
	ICNC1	ICES1	-	WGM13	WGM12	CS12	CS11	CS10
Read/Write	RW	RW	R	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0

TCCR1B Control Register

▼ TIFR

TIFR Register

Bit	7	6	5	4	3	2	1	0
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0
Read/Write	RW	RW	RW	RW	RW	RW	RW	RW
Initial Value	0	0	0	0	0	0	0	0

OCF2	Timer2 Output Compare Match flag
TOV2	Timer2 Overflow flag
ICF1	Input Capture flag
OCF1A	Timer1 Output Compare A Match flag
OCF1B	Timer1 Output Compare B Match flag
TOV1	Timer1 Overflow flag
OCF0	Timer0 Output Compare Match flag
TOV0	Timer0 Overflow flag

▼ TIMSK

Bit	7	6	5	4	3	2	1	0
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

TIMSK Register

▼ MCUCR

▼ GICR

▼