Author: Arsalan Doost Zanganeh

Student No: 401521228

---

# Q1:

To calculate the checksum and perform the error check, we typically use the **simple checksum method** (adding the values and taking the two's complement) or a **modulo operation**. Below are the steps for both datasets:

---

**Dataset A: 0x65, 0x09, 0x95**

1. **Step 1: Calculate the sum of the values**
   Add the values in hexadecimal:
   0x65 + 0x09 + 0x95 = 0x103
2. **Step 2: Extract only the lowest byte**
   The sum is 0x103. Extract the lower byte:
   0x103 mod 256 = 0x03
3. **Step 3: Calculate the checksum**
   The checksum is the **two's complement** of the sum's lower byte:
   Two’s complement = 256 − 0x03 = 0xFD

**Checksum = 0xFD**

4. **Step 4: Verify the checksum**
   Add all values, including the checksum, to check if the result is 0x00:
   0x65 + 0x09 + 0x95 + 0xFD = 0x00

**Error Check: Passed**

**Dataset B: 0x71, 0x69, 0x31, 0x88**

1. **Step 1: Calculate the sum of the values**
   Add the values in hexadecimal:
   0x71 + 0x69 + 0x31 + 0x88 = 0x193
2. **Step 2: Extract only the lowest byte**
   The sum is 0x193. Extract the lower byte:
   0x193 mod 256 = 0x93
3. **Step 3: Calculate the checksum**
   The checksum is the **two's complement** of the sum's lower byte:
   Two's complement = 256 − 0x93 = 0x6D

**Checksum = 0x6D**

4. **Step 4: Verify the checksum**
   Add all values, including the checksum, to check if the result is 0x00:
   0x71 + 0x69 + 0x31 + 0x88 + 0x6D = 0x200
   0x200 mod 256 = 0x00

**Error Check: Passed**

---

**Final Results:**

- **Dataset A: Checksum = 0xFD, Error Check = Passed**
- **Dataset B: Checksum = 0x6D, Error Check = Passed**

# Q2:

To determine the contents of the registers R20, R30 and R31 after executing the program, let's analyze it step by step.

---

**Program Analysis**

1. **Definitions:**
    - .ORG0 : The program starts at memory location 0.
    - .EQU DATA_ADDR = (OUR_DATA<<1): The label OUR_DATA (at program memory address 0x100) is multiplied by 2 because addresses in Flash memory (program memory) are word-addressed, and each instruction is 2 bytes. Thus, DATA_ADDR = 0x200
2. **Instructions:**
    - **LDIR30,LOW(DATA_ADDR):**
      Load the lower byte of DATA_ADDR = 0x200 into R30
      R30=0x00
    - **LDI R31, HIGH(DATA_ADDR):**
      Load the higher byte of DATA_ADDR = 0x200 into R31
      R31=0x02
    - **LPM R20, Z:**
      Use the Z-register ( composed of R31:R30 ) to load a byte from program memory into R20
      Z = 0x0200, which points to the first byte at OUR_DATA

      OUR_DATA contains the string **'M', 'P', ' ', 'I', 'U', 'S', 'T'**
      The first byte is the ASCII value of 'M', which is **0x4D**
      R20=0x4D

---

**Register Values:**

- R20=0x4D = 0x4D ('M')
- R30=0x00 = 0x00 (lower byte of DATA_ADDR)
- R31=0x02 = 0x02 (higher byte of DATA_ADDR)

# Q3:

To check if bit 6 of port B is set, the **SBIS** (Skip if Bit in I/O Register is Set) instruction is used. This instruction skips the next instruction if the specified bit in the specified register is **1**.

**Required Instruction:**

SBIS PINB, 6

**Explanation:**

1. **PINB:** This is the input register for port B, which shows the status of the pins.
2. **6:** Refers to bit 6 of the PINB register.
3. If bit 6 of PINB is **set** (logic 1), the next instruction is skipped.

**Example Code:**

```
SBIS PINB, 6     ; If bit 6 of PINB is set, skip the next instruction.
RJMP BIT_IS_SET  ; If bit 6 is set, jump to the BIT_IS_SET label.
```

---

# Q4:

The range of numbers represented by a signed integer depends on its bit width. For an **N-bit signed integer**, the range is:

−(2N−1) to +(2N−1−1)-\left(2^{N-1}\right) \, \text{to} \, +\left(2^{N-1} - 1\right)

This is because one bit is reserved for the sign (positive or negative).

---

**Common Ranges:**

1. **8-bit signed integer:**

   −128 to +127

2. **16-bit signed integer:**

   −32,768 to +32,767

3. **32-bit signed integer:**

   −2,147,483,648 to +2,147,483,647-2,147,483,648

4. **64-bit signed integer:**

   −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

---

**Example for ATmega32:**

The ATmega32 microcontroller typically uses **8-bit signed integers** in its registers. Therefore, the range is:

−128 to +127

---

# Q5:

A **macro** is a preprocessor directive in assembly programming that defines a reusable block of code with a name. When the macro is called in the program, it is replaced by the defined block of code during the assembly process, before the actual machine code is generated. Macros are similar to functions but do not involve call and return overhead, as their code is directly expanded inline.

---

**Key Features of Macros:**

1. **Reusability:** Allows writing repetitive tasks only once and reusing them multiple times.
2. **Parameterization:** Macros can accept parameters, making them flexible for various inputs.
3. **Code Expansion:** The macro's code is directly inserted into the program where the macro is invoked.
4. **Simplification:** Makes the program easier to read and maintain by avoiding repetitive code.

---

**Applications of Macros:**

1. **Simplifying Repeated Code:** Reduces the need to rewrite common operations like setting/clearing bits or delays.
2. **Improving Readability:** Gives meaningful names to blocks of code, making the program easier to understand.

3. **Parameterization:** Allows creating generic routines that work for multiple registers or values.
4. **Reducing Errors:** Minimizes the chance of mistakes in repetitive code by centralizing it in one macro definition.

---

**Benefits:**

- **Efficiency in Development:** Speeds up writing and debugging code.
- **No Overhead:** Unlike subroutines, macros don't involve stack operations for calls and returns.

**Limitations:**

- **Code Size:** Expanding macros inline can increase program size.
- **No Debug Information:** Macros don’t exist in the final binary, making debugging harder.

---

# Q6:

**Timers vs Counters**

- **Timer:**
  A timer is a hardware module in microcontrollers that increments at regular intervals based on an internal clock source (e.g., the system clock or a prescaled version of it). Timers are used for measuring time intervals, generating delays, or creating periodic interrupts.
- **Counter:**
  A counter is similar to a timer, but it increments (or decrements) based on external events (e.g., pulses on a specific pin). Counters are typically used for counting external events like button presses, pulses from sensors, or encoder signals.

**Differences Between Timer and Counter**

| Feature | Timer | Counter |
|---|---|---|
| **Clock Source** | Internal clock (e.g., system clock) | External events (e.g., pin toggles) |
| **Use Case** | Time measurement, delays, PWM | Event counting |
| **Prescaler** | Can divide the internal clock rate | Typically not needed |
| **Trigger Source** | Internal | External input pins |

---

**Timers/Counters in ATmega32**

The ATmega32 microcontroller has **three Timer/Counter modules**, which can operate as either timers or counters depending on the configuration:

1. **Timer/Counter 0:**
    - **Size:** 8-bit
    - **Clock Source:** Internal or external
    - **Features:** Fast PWM, Phase Correct PWM, CTC, Normal mode.
2. **Timer/Counter 1:**
    - **Size:** 16-bit
    - **Clock Source:** Internal or external
    - **Features:** More advanced, supports Input Capture mode for event timing.
3. **Timer/Counter 2:**
    - **Size:** 8-bit
    - **Clock Source:** Internal or external
    - **Features:** Similar to Timer 0 but with asynchronous operation capability using an external crystal.

---

**Key Features of ATmega32 Timers/Counters**

- **Timer 0 and Timer 2 (8-bit):**
    - Can count up to 255 before rolling over.

- Useful for generating shorter delays or fast PWM signals.

- **Timer 1 (16-bit):**
  - Can count up to 65,535 before rolling over.
  - Ideal for precise timing or longer delays.
- **Modes of Operation:**
  - Normal Mode: Counts from 0 to maximum, then resets.
  - CTC (Clear Timer on Compare): Resets the timer when it matches a specified value.
  - PWM Modes: Generate pulse-width modulated signals.

---

**Applications**

- **Timer:**
  - Delay generation.
  - Creating square wave outputs (PWM).
  - Measuring time between events using Input Capture.
- **Counter:**
  - Counting external pulses or events.
  - Tracking sensor or encoder outputs.

---

# Q7:

**Role of the TOV Flag (Timer/Counter Overflow Flag)**

The **TOV (Timer/Counter Overflow)** flag is a special bit in the **status register (TIFR)** of the ATmega32 microcontroller that indicates when a timer or counter has overflowed. This flag is used to signal that the timer or counter has reached its maximum count value and then rolled over (reset back to zero).

**When Does TOV Get Activated?**

- The TOV flag gets **set** (activated) when the timer or counter reaches its maximum value and overflows.
- For an **8-bit timer** (Timer 0 or Timer 2), it overflows after reaching 255, and for a **16-bit timer** (Timer 1), it overflows after reaching 65,535.
- This means when the timer or counter counts up to its maximum value and then resets to zero, the TOV flag will be set, signaling that the timer has overflowed.

**Example:**

- **For 8-bit Timer (e.g. Timer 0):**
    - When Timer 0 reaches 255 and the next clock pulse occurs, it overflows back to 0, and the **TOV0** flag is set.
- **For 16-bit Timer (e.g. Timer 1):**
    - When Timer 1 reaches 65,535 and overflows back to 0, the **TOV1** flag is set.

---

**Clearing the TOV Flag:**

The TOV flag is automatically cleared when the corresponding interrupt service routine (ISR) is executed, or it can be manually cleared by writing a **1** to it (using the TIFR register).

---

# Q8:

To enable **Compare Match A** mode in **Timer1** of the ATmega32, you need to configure the **Timer/Counter Control Registers** (TCCR1A and TCCR1B) appropriately. The **Compare Match A** mode is used to generate an interrupt or take some action when the value in the timer reaches a specific compare value (OCR1A).

---

**Steps to Activate Compare Match A Mode for Timer1:**

1. **Set the appropriate bits in the TCCR1A and TCCR1B registers.**

- **TCCR1A (Timer/Counter Control Register A)**:
  Set the **COM1A1** bit (bit 7) and **COM1A0** bit (bit 6) to configure the behavior of the **OC1A** pin (Output Compare A). In **Compare Match A** mode, the pin will toggle, clear, or set based on the desired behavior.

- **TCCR1B (Timer/Counter Control Register B)**:
  Set the **WGM12** bit (bit 4) to 1 to enable **CTC (Clear Timer on Compare Match)** mode, which is necessary for Compare Match A operation.

2. **Set the compare value (OCR1A)**:
   The **OCR1A** register holds the value to compare against the timer value. When the timer reaches this value, a Compare Match A event occurs.

---

**Required Register Settings:**

- **TCCR1A**:
  - COM1A1 = 1 (non-normal mode behavior for OC1A pin)
  - COM1A0 = 0 (output is cleared on compare match)
  - WGM10 = 0, WGM11 = 0 (these bits are irrelevant for this mode, as we're using CTC mode)
- **TCCR1B**:
  - WGM12 = 1 (CTC mode, Compare Match A mode enabled)
  - WGM13 = 0 (irrelevant in this case)
  - CS12, CS11, CS10 (prescaler bits to control timer frequency)

---

**Example Code:**

```
; Initialize Timer1 in Compare Match A mode with CTC
LDI    R16, (1 << WGM12)        ; Set WGM12 in TCCR1B (CTC mode)
OUT    TCCR1B, R16
LDI    R16, (1 << COM1A1)       ; Set COM1A1 in TCCR1A (OC1A clear on compare match)
OUT    TCCR1A, R16
LDI    R16, 0xFF                ; Load compare value (OCR1A) (e.g., 255 for a fast interrupt)
OUT    OCR1A, R16
LDI    R16, (1 << CS10)         ; Set prescaler to 1 (no prescaling)
OUT    TCCR1B, R16
```

**Explanation:**

1. **TCCR1A**:
    - COM1A1 is set to 1, COM1A0 is set to 0 to clear the OC1A pin when a compare match occurs (this could also be set to toggle or set depending on the application).
2. **TCCR1B**:
    - WGM12 is set to 1 to enable CTC mode for Timer1, meaning the timer will reset when it matches OCR1A.
3. **OCR1A**:
    - Set the compare value (e.g. 255), and the timer will compare the timer count to this value.
4. **Prescaler**:
    - The prescaler is set to 1 (no division), so the timer counts at the system clock speed.

---

**Result:**

When Timer1 reaches the value in **OCR1A**, it triggers a **Compare Match A** event, which can be used to generate an interrupt or take action like toggling an output pin, depending on how the **COM1A1** and **COM1A0** bits are configured.

---

# Q9:

Both **RET** and **RETI** are instructions used in assembly to return from a function or interrupt service routine (ISR), but they are used in different contexts, particularly when dealing with interrupts.

---

**RET (Return from Subroutine)**

- **Purpose:** The RET instruction is used to return from a normal subroutine (a regular function call in the program).
- **Operation:** It pops the return address from the stack and returns control to the instruction after the CALL or function invocation.

- **Usage:** Used in regular function calls or non-interrupt routines.

**RETI (Return from Interrupt)**

- **Purpose:** The RETI instruction is used to return from an interrupt service routine (ISR).
- **Operation:** It works similarly to RET, but in addition to popping the return address from the stack, it also:
    - **Enables global interrupts** by setting the Global Interrupt Flag (I).
    - **Restores the interrupt flag** to its previous state, ensuring the interrupt system works correctly after returning from the ISR.
- **Usage:** Used specifically when returning from an interrupt service routine.

---

**Key Differences:**

| Feature | RET | RETI |
|---|---|---|
| **Context** | Used in normal subroutines | Used in interrupt service routines (ISRs) |
| **Global Interrupts** | Does not affect global interrupts | Restores the Global Interrupt Flag (I) |
| **Purpose** | Return from a function | Return from an interrupt service routine (ISR) |
| **Operation** | Pops the return address from the stack | Pops the return address and enables interrupts |

---

# Q10:

**What is a Prescaler?**

A **prescaler** is a mechanism used in timers and counters to divide the input clock frequency by a fixed value, effectively slowing down the rate at which the timer or counter increments. It allows you to adjust the frequency of the timer or counter by selecting a division factor. The prescaler is especially useful when you need a longer time interval or when you want to reduce the frequency of timer events.

In microcontrollers like the ATmega32, the prescaler is typically set using specific bits in the timer control registers (e.g., TCCR0, TCCR1B).

**How Does a Prescaler Work?**

When the prescaler is applied, the timer's clock source is divided by the prescaler value, which means the timer's counter increments at a slower rate than the original clock. This allows for longer counting periods and greater flexibility in generating time delays or handling events.

For example, if the system clock is 16 MHz and a prescaler of 8 is selected, the timer will increment every 8 clock cycles instead of every 1 clock cycle.

**Prescaler Example for Timer0 in ATmega32**

Suppose the system clock is 16 MHz, and you want to use Timer0 to create a delay. The prescaler will allow you to control the frequency at which the timer counts.

**Steps to Set Prescaler for Timer0:**

1. The system clock is 16 MHz, so each clock cycle is 1/16,000,000 seconds (62.5 ns).
2. If you set a prescaler of 64, the timer will increment every 64 clock cycles.
3. Therefore, each timer tick will occur every 64×62.5 ns=4 μs64 \times 62.5 \, \text{ns} = 4 \, \mu\text{s}.

**Setting the Prescaler in Code:**

```
LDI    R16, (1 << CS01) | (1 << CS00)   ; Set prescaler to 64 (CS01 = 1, CS00 = 1)
OUT    TCCR0, R16                       ; Apply the prescaler to Timer0
```

In this case, the **CS01** and **CS00** bits are set to 1, which configures the prescaler to 64.

---

**Example: Creating a Delay Using Timer0 with a Prescaler**

Let's say you want to create a 1-second delay using Timer0. Given a system clock of 16 MHz and a prescaler of 64:

1. **Clock frequency with prescaler = 16 MHz / 64 = 250,000 Hz.**
2. The timer will increment every $\frac{1}{250000}$ = 4 μs.
3. To create a 1-second delay, you need the timer to count for $\frac{1\ \text{second}}{4\ \mu\text{s}}$ = 250,000 ticks

So, you would set the **OCR0** register to 250,000 (in an appropriate 8-bit format) to make the timer overflow after 1 second.

---

**Prescaler Values in ATmega32:**

The ATmega32 allows several prescaler options, including values like 1, 8, 64, 256, and 1024. These values can be set using the control bits in the timer's register.