Syllabus of Embedded System

Unit 2: Programming for Embedded Systems (5 Hours)

2.1 Overview of AVR Architecture
2.2 Embedded C Programming for Microcontroller

- Introduction to C for Embedded Systems
- Data Types, Control Structures, and pointers
- Memory Management
- AVR Interrupt handling
- Input and output ports interfacing on AVR
- Timers and Counters in AVR
- Serial Communication in AVR

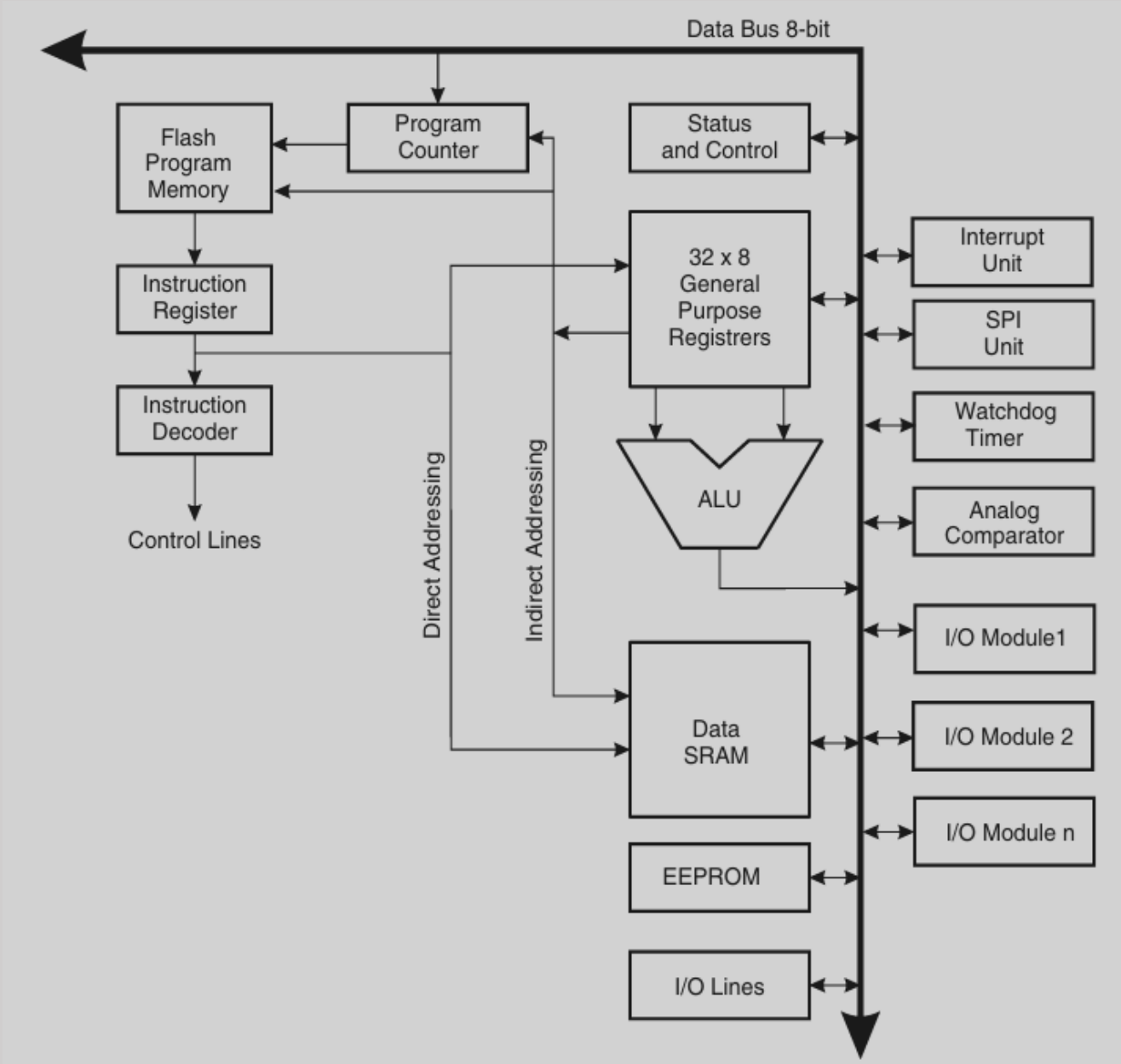
Overview of AVR Architecture

AVR is a family of 8-bit RISC microcontrollers developed by Atmel (now Microchip).

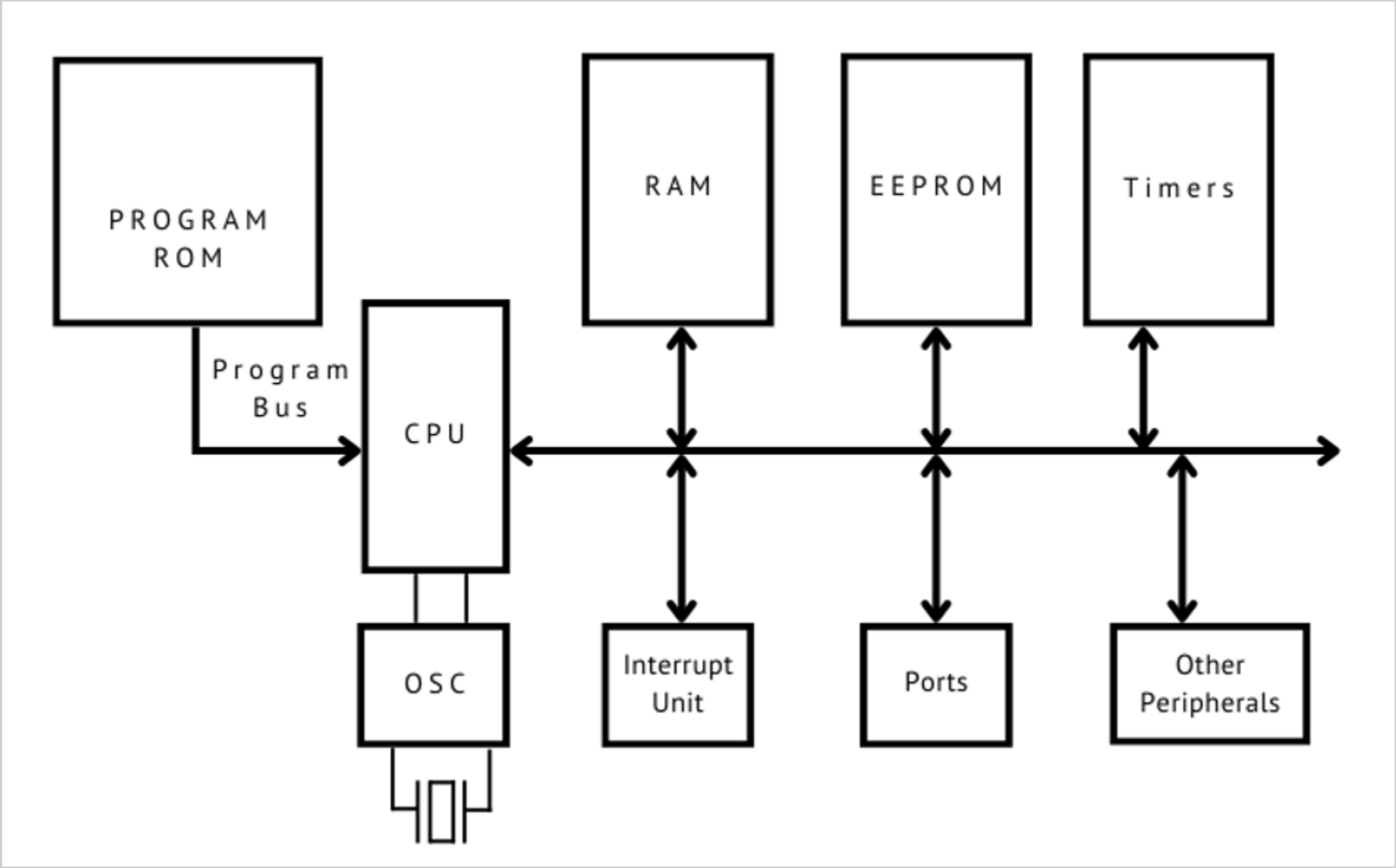
It was one of the first microcontroller families to use on-chip Flash memory for program storage.

The name AVR stands for Alf-Egil Bogen and Vegard Wollan's RISC processor.

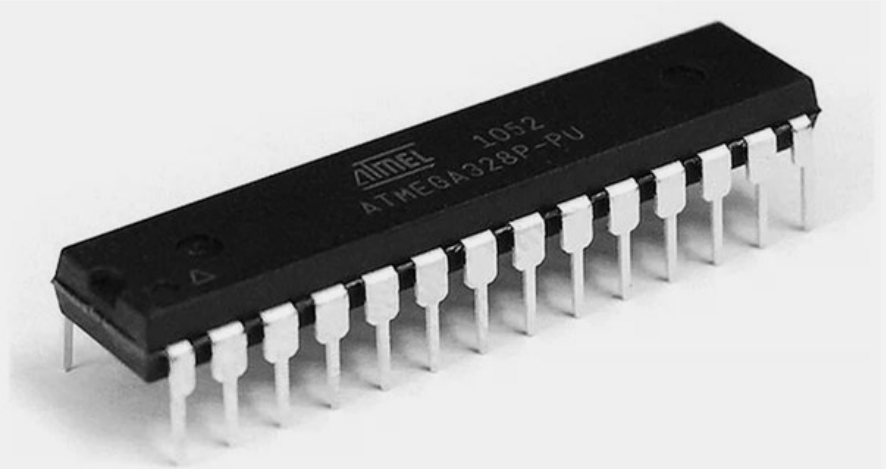
It is based on a **Harvard architecture**, which separates program and data memory.



Overview of AVR Architecture



Overview of AVR Architecture



RISC Architecture: (Reduced Instruction Set Computer). Features a rich instruction set where most instructions execute in a single clock cycle.

High Performance: Achieves high throughput (approaching 1 MIPS per MHz) by executing instructions quickly and efficiently.

Modified Harvard Architecture: Uses separate buses and memories for Program (Flash) and Data (SRAM), allowing simultaneous access and improving speed.

32 General-Purpose Registers: A large register file, with all registers directly connected to the ALU (Arithmetic Logic Unit).

Overview of AVR Architecture

Memory Architecture

32KB Flash (Program)

In-System Programmable (ISP) non-volatile memory used for storing the program code. It can be reprogrammed thousands of times.

2KB SRAM (Data)

Static RAM. This is volatile memory used for storing variables, stack, and data during runtime. All data is lost when power is off.

1KB EEPROM (Data)

Non-volatile memory used for storing data that must be saved when power is off, such as configuration settings or calibration data.

Overview of AVR Architecture

Memory Architecture

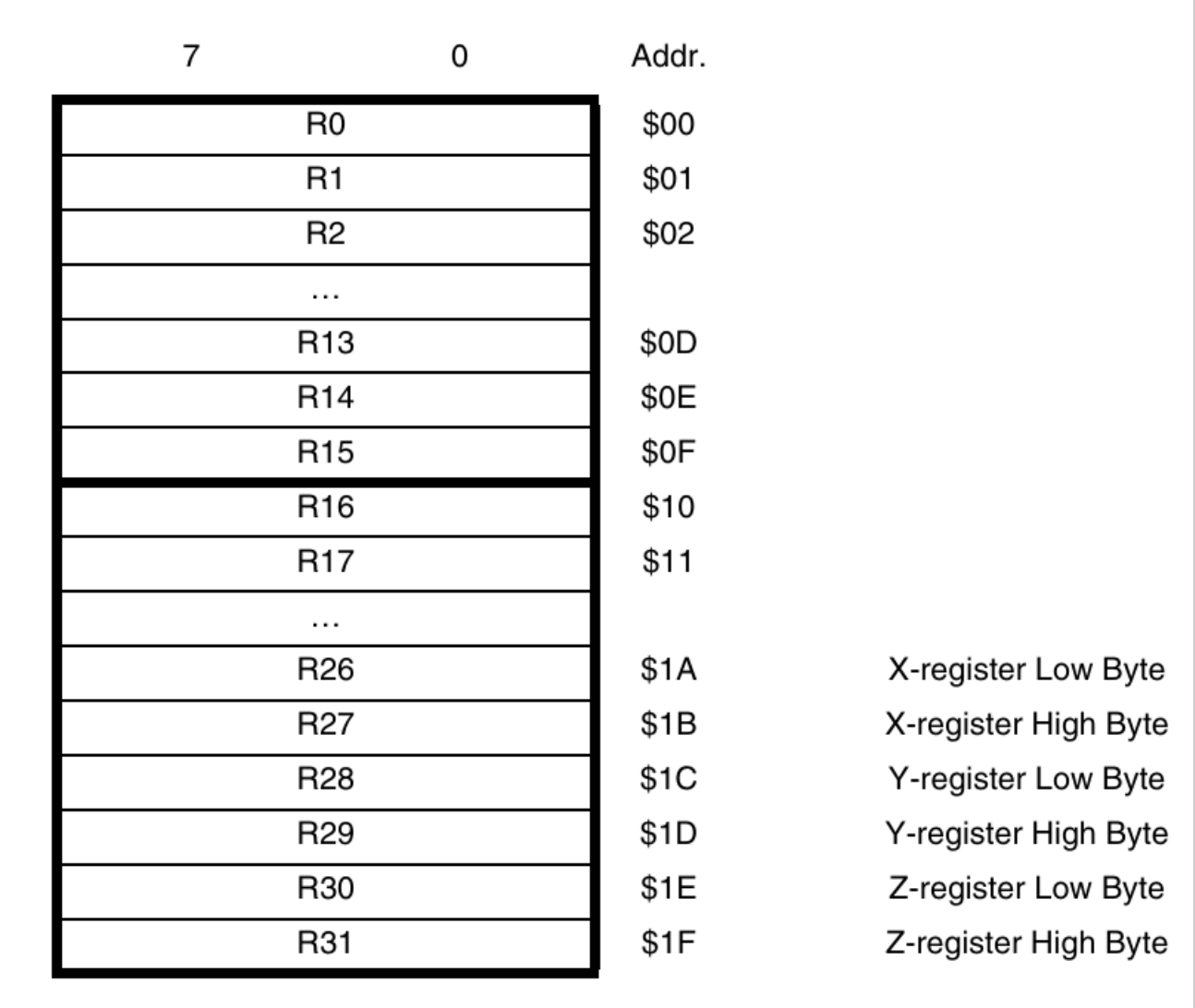
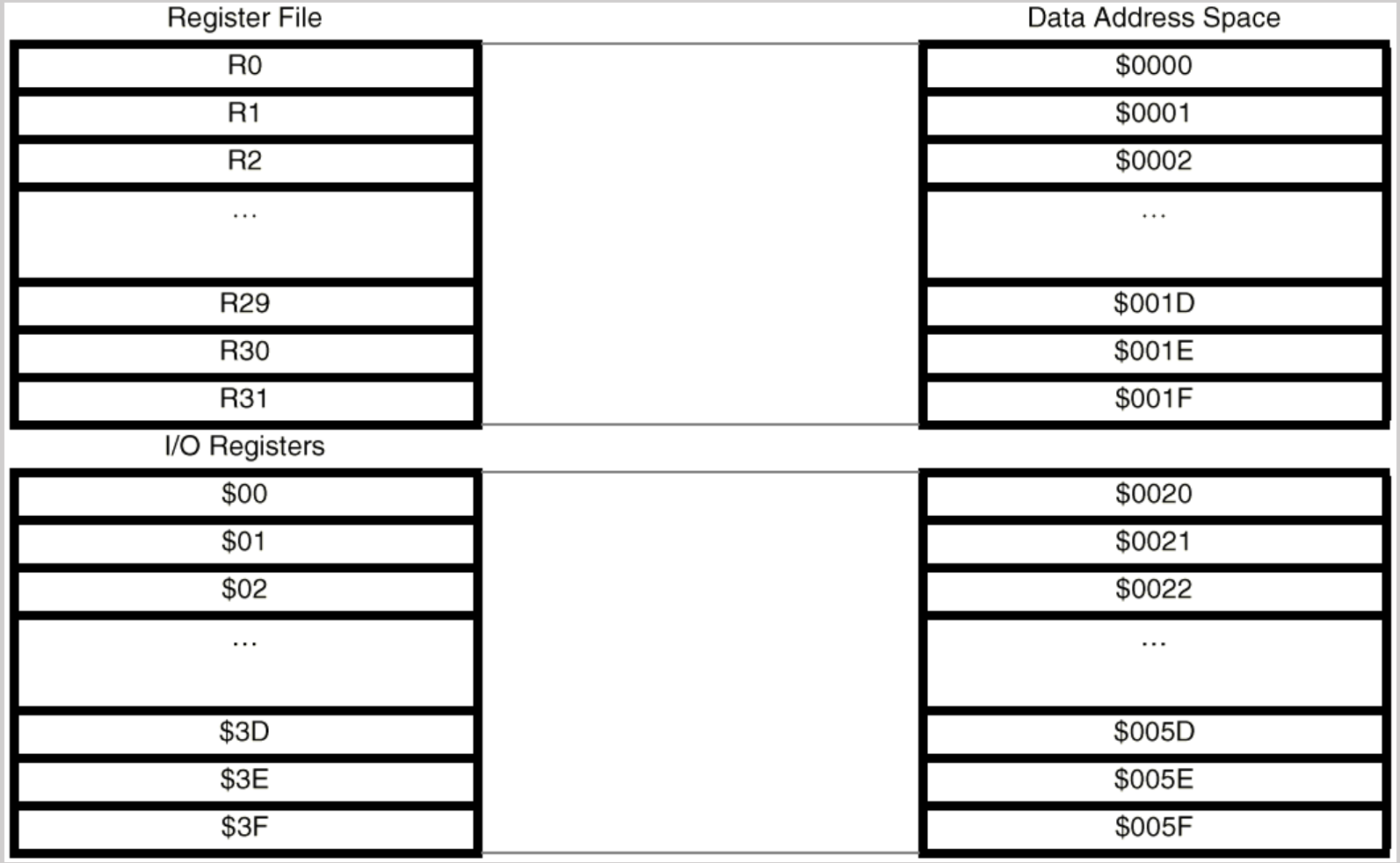


fig: 32x8 register

Overview of AVR Architecture

Memory Architecture



Overview of AVR Architecture

Memory Architecture

$1D ($3D)	EEDR	EEPROM Data Register								page 39
$1C ($3C)	EECR	-	-	-	-	-	EEMWE	EEWE	EERE	page 39
$1B ($3B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	page 54
$1A ($3A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	page 54
$19 ($39)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	page 54
$18 ($38)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	page 56
$17 ($37)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	page 56
$16 ($36)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	page 56
$15 ($35)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	page 61
$14 ($34)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	page 61
$13 ($33)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	page 61
$12 ($32)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	page 63
$11 ($31)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	page 63
$10 ($30)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	page 63
$0F ($2F)	SPDR	SPI Data Register								page 44
$0E ($2E)	SPSR	SPIF	WCOL	-	-	-	-	-	-	page 44
$0D ($2D)	SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	page 43

Overview of AVR Architecture

CPU and Register File

Arithmetic Logic Unit (ALU)

The high-performance ALU is directly connected to all 32 general-purpose registers. This allows it to access two independent registers, perform an operation, and write the result back in a single clock cycle.

General Purpose Registers

The 32 x 8-bit registers (R0-R31) are the heart of the AVR core. This fast-access register file is the heart of the CPU's operation, reducing the need to access the slower data SRAM.

Overview of AVR Architecture

I/O ports

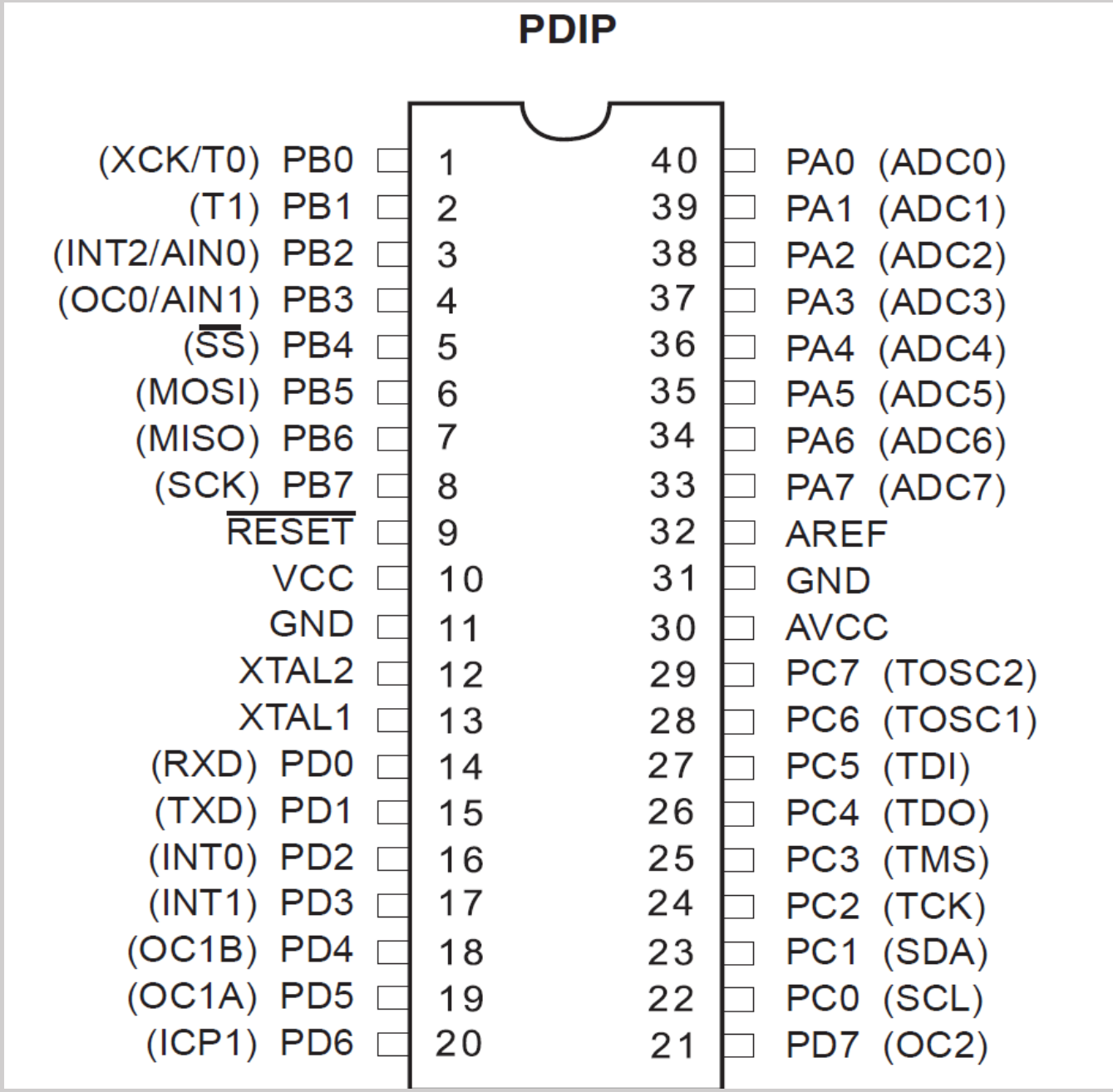
The ATmega32 features four 8-bit I/O ports: **PORTA, PORTB, PORTC, and PORTD** (totaling 32 I/O lines).

Each pin is bi-directional and can be configured as either an input or an output using three key registers:

DDRx: Data Direction Register (sets pin as input/output).

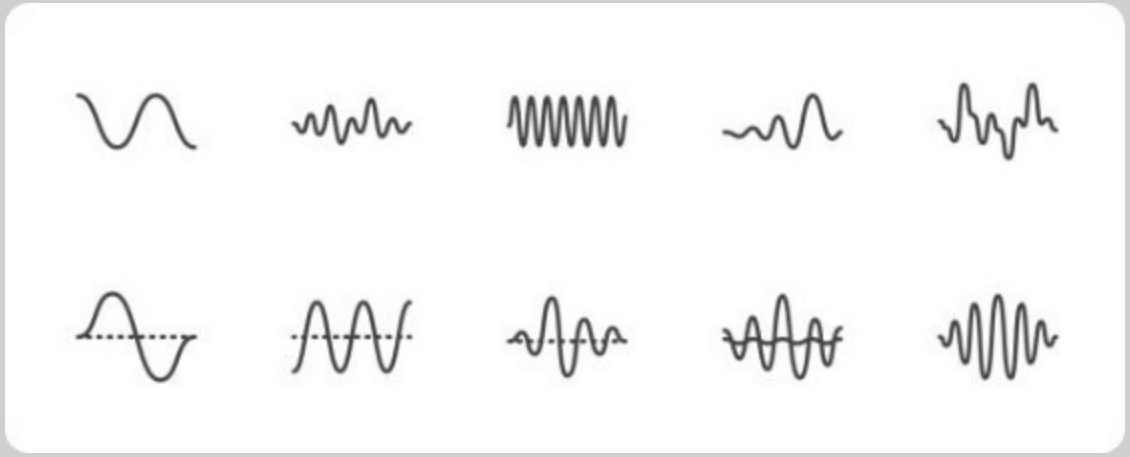
PORTx: Data Register (writes output data or enables pull-up).

PINx: Input Pins Address (reads the logic level from the pin).



Overview of AVR Architecture

Timers and ADC



Timers/Counters: Features three timers (two 8-bit, one 16-bit). Used for timing events, generating PWM (Pulse Width Modulation), and counting external events.

Analog-to-Digital Converter (ADC): A 10-bit, 8-channel ADC. This converts real-world analog signals (like from sensors) into digital values the MCU can understand.

Overview of AVR Architecture

Communication System

USART

(Universal Synchronous/Asynchronous Receiver/Transmitter) For serial communication with PCs or other devices (e.g., RS-232).

SPI

(Serial Peripheral Interface) A fast, synchronous, full-duplex communication protocol often used for high-speed peripherals like SD cards.

TWI (I^2C)

(Two-Wire Interface) A multi-master, multi-slave serial bus (identical to I^2C) commonly used for connecting sensors and peripherals.

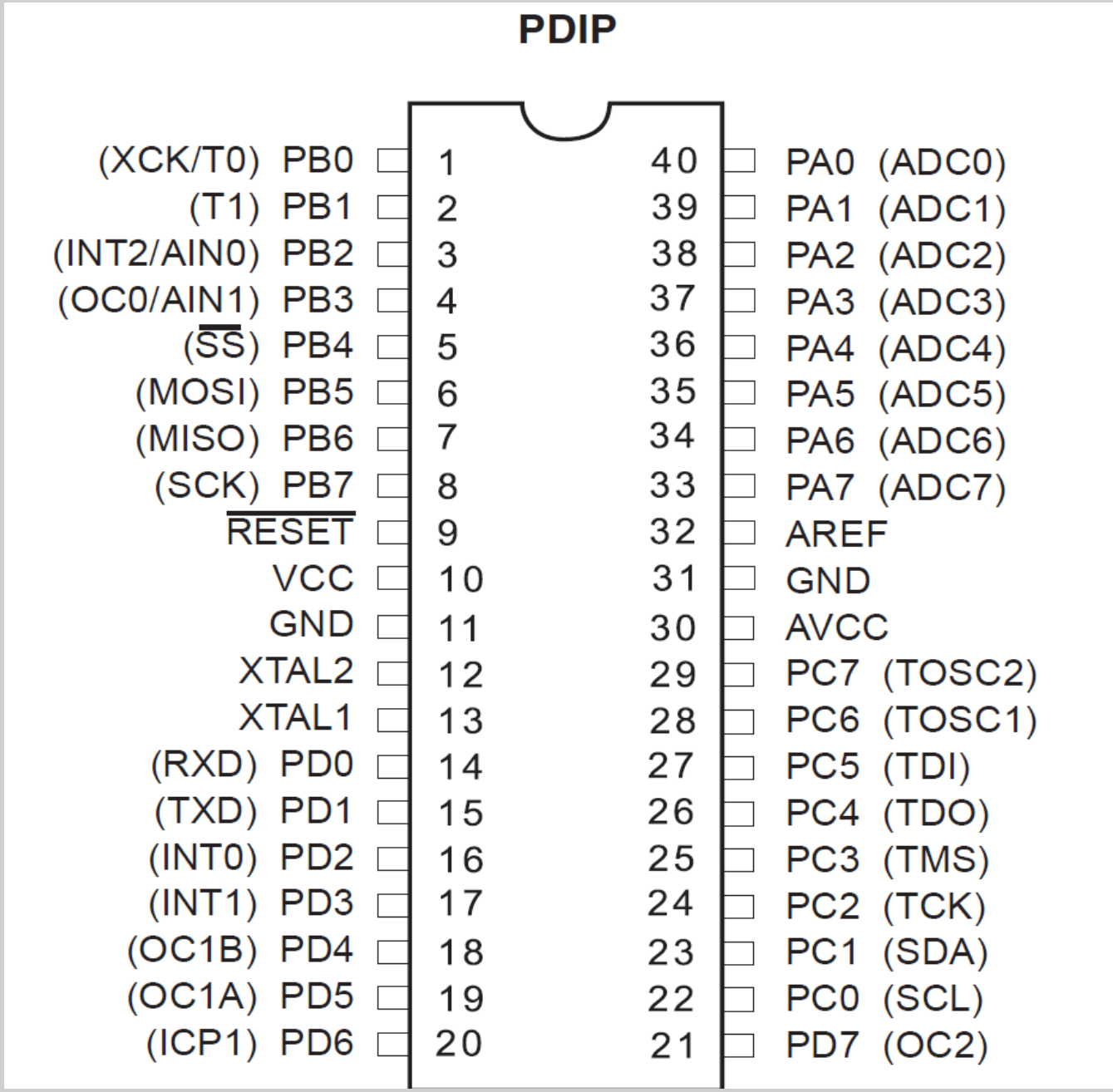
Overview of AVR Architecture

Interrupt System

Interrupts allow the MCU to respond to critical events (like a button press or data arrival) without halting the main program.

The ATmega32 has a powerful interrupt structure with multiple sources, including:

- External Interrupts (INT0, INT1, INT2)
- Timer Overflows/Compares
- ADC Conversion Complete
- USART, SPI, and TWI events



Introduction to C for embedded system

Why C for embedded systems?

Close to hardware but still high-level

Efficient and generates small, fast machine code

Full control over memory

Direct access to registers

Introduction to C for embedded system

No printf(), no screen, no keyboard

Everything is done through:

- Registers
- Peripherals
- External interfaces (LEDs, sensors, serial)

No dynamic memory (avoid malloc, free)

- Bad for small RAM microcontrollers.

main() never ends

- There is no operating system to return to.

Timing is critical

- Delays, timers, interrupts are essential.

Data types, Control structures and Pointers

C Type	Size on ATmega32	Range	Notes
char	1 byte (8-bit)	–128 to 127	Most operations are 8-bit, very fast
unsigned char / uint8_t	1 byte	0 to 255	Most common for registers
int / signed int	2 bytes (16-bit)	–32768 to 32767	Default integer type
unsigned int	2 bytes	0 to 65535	Good for timer values
long	4 bytes		Slower—avoid unless needed
float	4 bytes		VERY slow on AVR, avoid in ISR & tight loops

Preferred types are uint8_t, uint16_t, uint32_t

Data types, Control structures and Pointers

If-else

& is ANDing and << is SHIFTing.

| is ORing

```
if (PINB & (1 << PB0)) {
    PORTC |= (1 << PC0);
} else {
    PORTC &= ~(1 << PC0);
}
```

Data types, Control structures and Pointers

switch-case

Used in

- Finite state machine
- Menu systems

```
switch(mode) {
    case 0: run_motor(); break;
    case 1: stop_motor(); break;
    case 2: reverse_motor(); break;
}
```

Data types, Control structures and Pointers

while(1)

while(1) is the backbone of main()

```
while (1) {
    read_sensor();
    update_output();
}
```

Data types, Control structures and Pointers

for loops

Used for

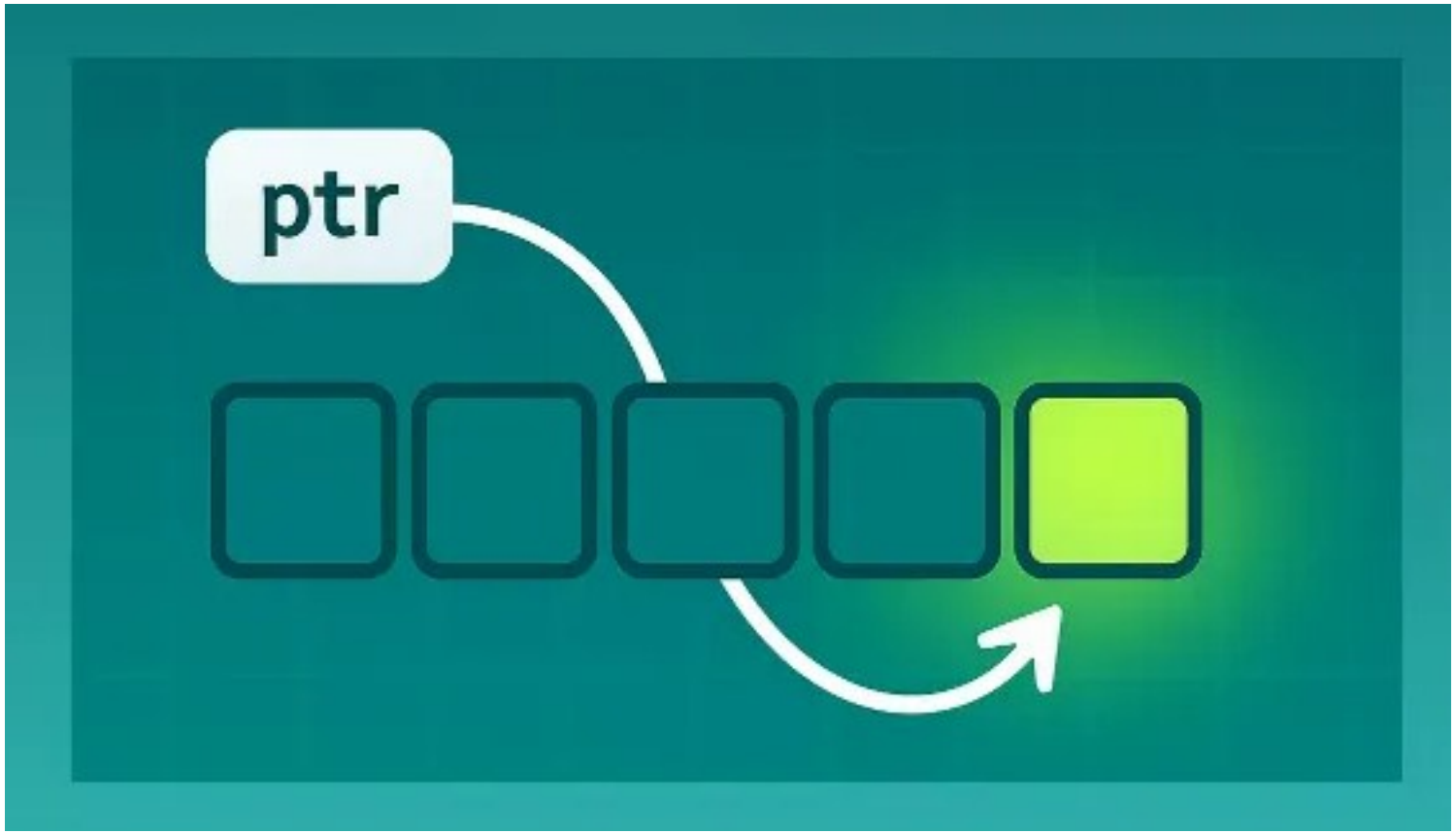
- Short delays
- Repetitive GPIO patterns

```
for (uint8_t i = 0; i < 8; i++) {
    PORTA = (1 << i);
    _delay_ms(200);
}
```

Data types, Control structures and Pointers

pointers

A pointer in C is a variable that stores the address of another variable.



Data types, Control structures and Pointers

pointers

Use of pointers

- Access hardware registers directly
- Manipulate port registers dynamically
- Pass variables to functions by reference

```
uint8_t x = 10;
uint8_t *ptr = &x;
```

Data types, Control structures and Pointers

Registers are memory addresses that control hardware.

avr/io.h defines all register name like PORTA, DDRA, PINA, etc

|= , &=~ are used to set/clear bits.

```
DDRC |= (1 << PC0);  // Set PC0 as output
PORTC &= ~(1 << PC0); // Set PC0 LOW
```

Data types, Control structures and Pointers

volatile

Use volatile when:

- Variable is a hardware register
- Variable is modified by an interrupt
- Variable is changed by hardware (ADC, Timer, UART)
- Variable is shared between main code and ISR
- Variable might change without your program writing to it

```
volatile uint8_t *portA = (uint8_t*)0x1B;
*portA = 0xFF;  // Write to PORTA
```

Data types, Control structures and Pointers

#include <avr/io.h> : Gives access to all Atmega registers

main(): program starts here

while(1): program starts here

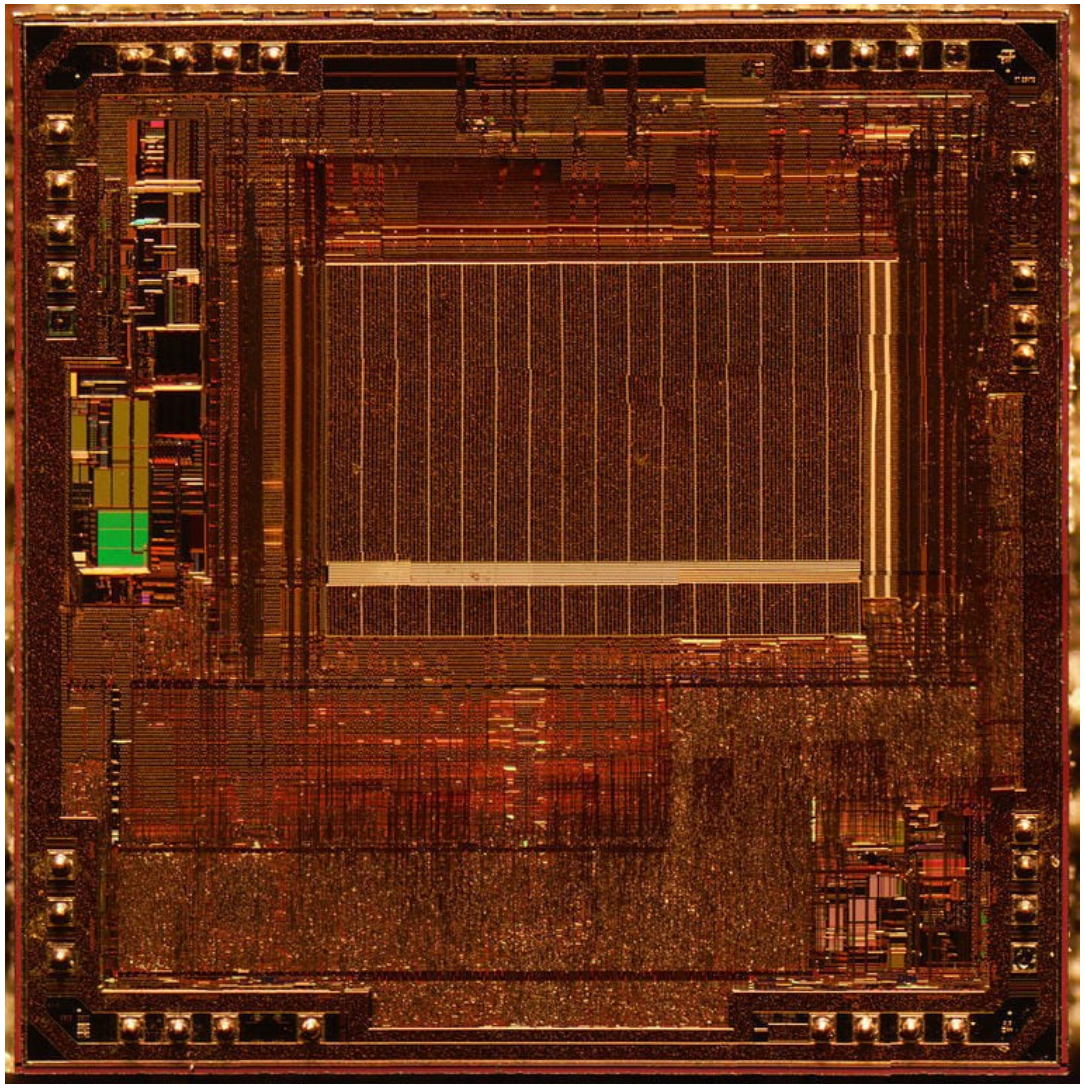
```
#include <avr/io.h>

int main(void) {
    DDRA = 0xFF;        // Configure PORTA as
output
    PORTA = 0b10101010; // Output pattern
    while (1) {
        // Endless loop
    }
}
```

Flash Memory Management

Flash memory is read-only during normal operation and is used to store the program code.

- Use **compiler optimization** settings (e.g., -Os in GCC) to reduce code size.
- Store constant data (e.g., lookup tables, strings) in flash using the PROGMEM directive.
- Avoid frequent flash rewrites as they have limited write cycles.
- Use bootloaders to simplify firmware updates without external programming tools.
- Utilize segment analysis tools to track flash usage.

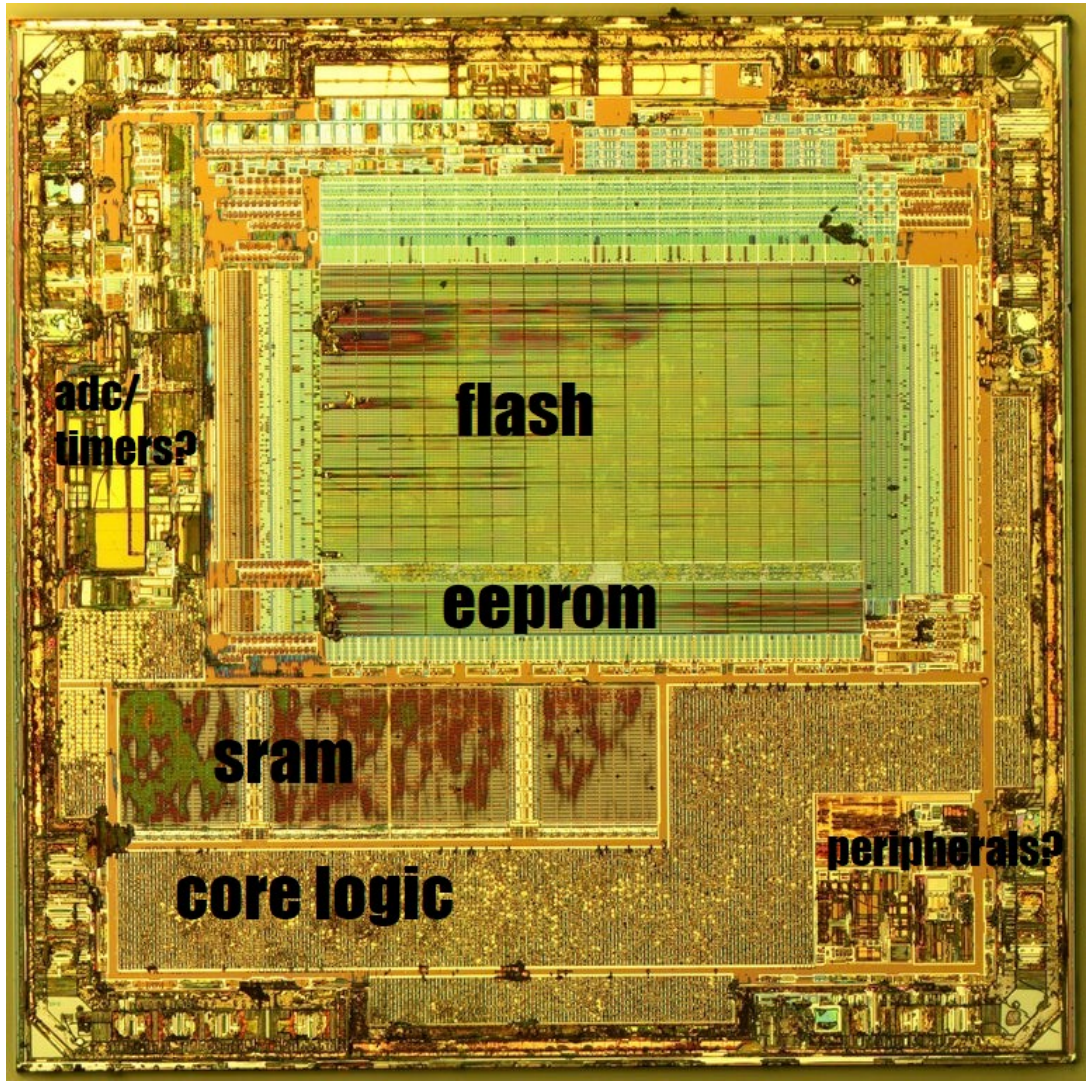


Memory management

SRAM Management

SRAM is used for runtime data, including global and local variables, stacks, and buffers.

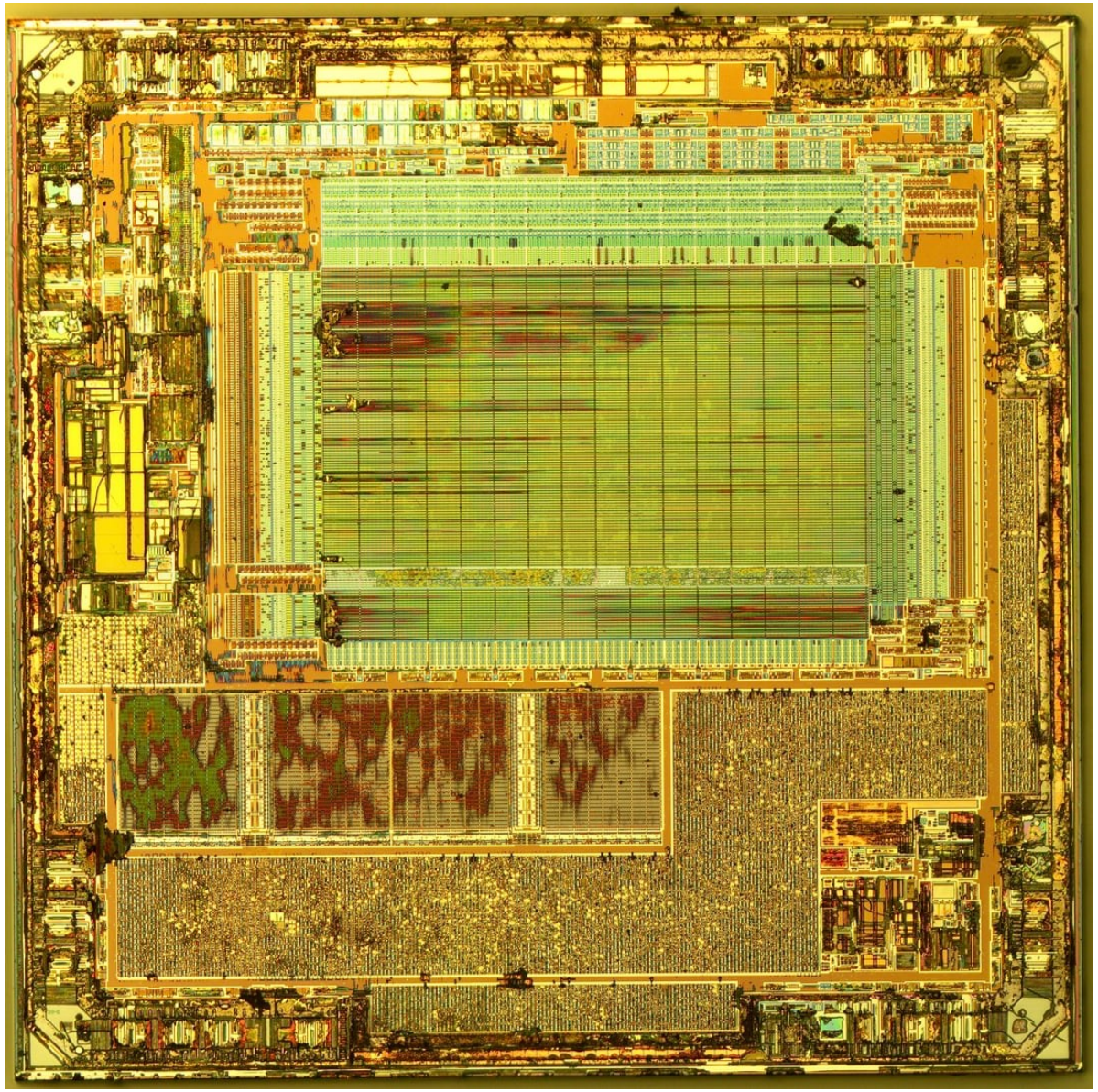
- Minimize global variables to reduce memory overhead.
- Use **stack-efficient coding practices**, such as reusing local variables.
- Avoid dynamic memory allocation (malloc/free) unless necessary, as it can lead to fragmentation.
- Leverage **data packing** techniques, like using smaller data types or bit-fields.
- Use static code analysis tools to detect potential SRAM overflows.



EEPROM Management

EEPROM provides non-volatile storage for configuration or data that must persist after a reset or power loss.

- Use the EEPROM library functions (eeprom_read_byte, eeprom_write_byte) for safe access.
- Implement wear leveling to extend EEPROM lifespan.
- Store frequently updated data in RAM and write it to EEPROM periodically.
- Perform read-after-write operations to ensure data integrity.
- Avoid writing to EEPROM in performance-critical sections due to slow write speeds.



AVR Interrupt Handling

Interrupt Vs Polling

Whenever any device need the microcontroller, the device notifies it by sending an interrupt signal.

Microcontroller continuously monitors the status of the device in polling method. When status is met, it performs the service.

We can assign priority to different interrupt differently.

The polling methods checks all devices in a round robin fashion.

Interrupt method can mask a device request for service.

Polling method wastes much of microcontroller's time by polling devices that do not need service.

AVR Interrupt Handling

Steps in executing an interrupt

It finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack.
It jumps to a fixed location in memory called the *interrupt vector table*. The interrupt vector table directs the microcontroller to the address of the interrupt service routine (ISR).
The microcontroller starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then it starts to execute from that address.

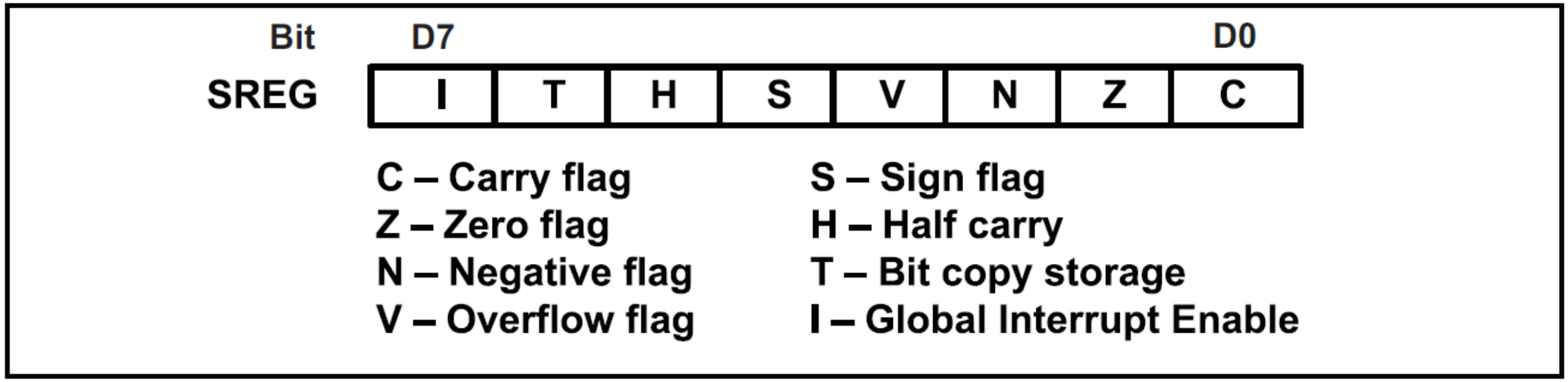
AVR Interrupt Handling

Sources of interrupts

- There are at least two interrupts set aside for each of the timers, one for overflow and another for compare match.
- Three interrupts are set aside for external hardware interrupts. INT0, INT1 and INT2.
- Serial communication's USART has three interrupts, one for receive and two interrupts for transmit.
- The SPI interrupts.
- The ADC.

AVR Interrupt Handling

Bits of Status Register (SREG)



Bit D7 (I) of the SREG register must be set to HIGH to allow the interrupt to happen.

If I=1, each interrupt is enabled by setting to HIGH the interrupt enable flag bit for that interrupt.

Input and Output ports interfacing on AVR

I/O Port Registers

DDRx: Data Direction Registers
These are 8-bit registers.
These are used to configure the pins of the ports as input or output.
Writing a one to the bits in this register sets those specific pins as output pins.
Writing a zero to the bits in this register sets those specific pins as input pins.
All bits in these registers can be read as well as written to.
The initial value of these bits is zero.

Input and Output ports interfacing on AVR

I/O Port Registers

PORTx: Data Registers
These are 8-bit registers.
These are used to put the pins of the ports in a logic HIGH or logic LOW state.
Writing a one to the bits in this register puts a HIGH logic (5V) on those pins.
Whereas writing a zero to the bits in this register puts a LOW logic (0V) on those pins.
All bits in these registers can be read as well as written to.
The initial value of these bits is zero.

Input and Output ports interfacing on AVR

I/O Port Registers

PINx: Input Pins Address Registers
These are 8-bit registers.
These are used to read the values on the specific pins of the port.
These bits are read-only bits and cannot be written to.

Input and Output ports interfacing on AVR

```
//Seven segment display
#include <avr/io.h>
#include <util/delay.h>

uint8_t digits[10] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};

int main(void) {
DDRD = 0xFF;

    while (1) {
        for (uint8_t i = 0; i < 10; i++) {
            PORTD = digits[i];
            _delay_ms(1000);
        }
    }
}
```

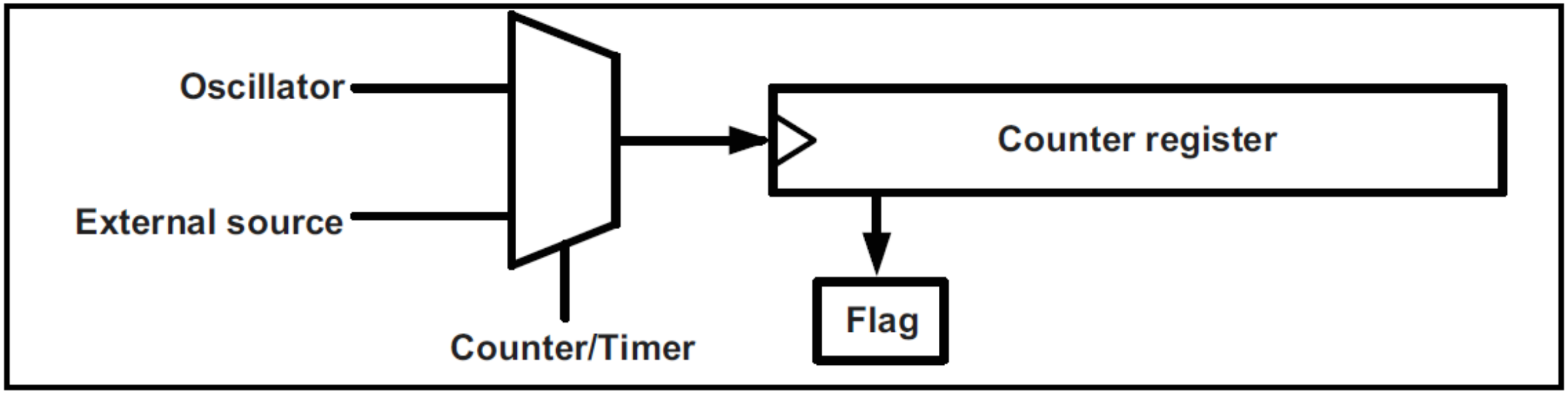
Input and Output ports interfacing on AVR

Create a program that reads an 8-bit value from PORTA and performs:

- Extract bits 2-5 and display on PORTB
- Count number of set bits in PORTA and show on PORTC (0-8)

Timer and Counter in AVR

General view of Counters and Timers

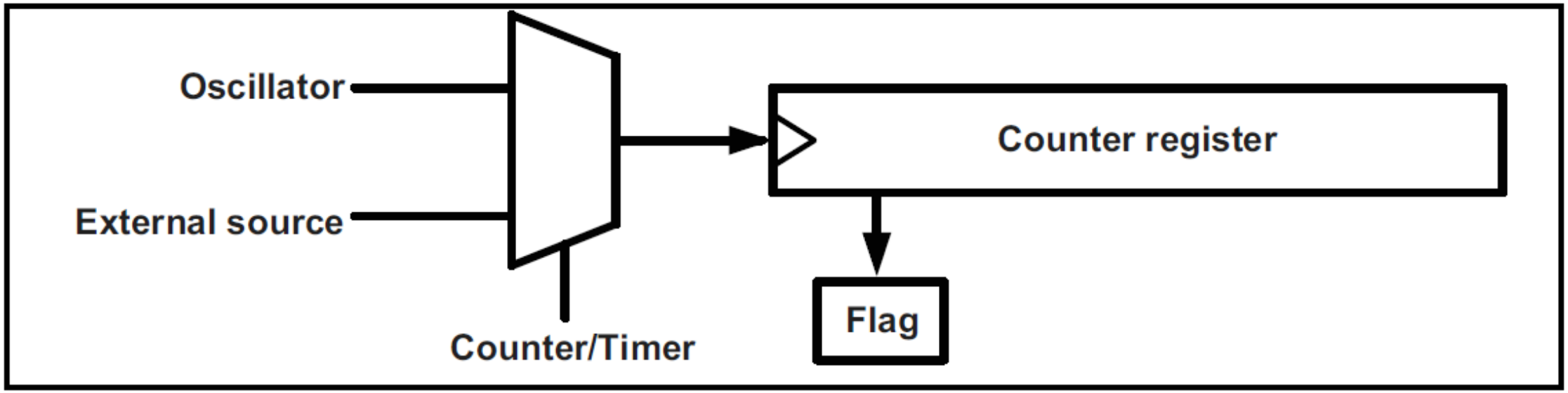


When we want to count an event, we connect the external event source to the clock pin of the counter register.

When we want to generate time delays, we connect the oscillator to the clock pin of the counter.

Timer and Counter in AVR

Ways to generate time delays



1. Clear the counter at the start time and wait until the counter reaches a certain number.
2. Load the counter register and wait until the counter overflows and the flag is set.

Timer and Counter in AVR

The AVR has one to six timers/counters named as Timers 0, 1, 2, 3, 4 and 5 depending on the family member.

In AVR some of the timers/counters are 8-bit and some are 16-bit.

ATmega32, there are three timers: Timer0(8-bit), Timer1(16-bit) and Timer2(8-bit).

Every timer needs a clock pulse to tick. The clock source can be internal(for timers) or external(for counters).

Timer and Counter in AVR

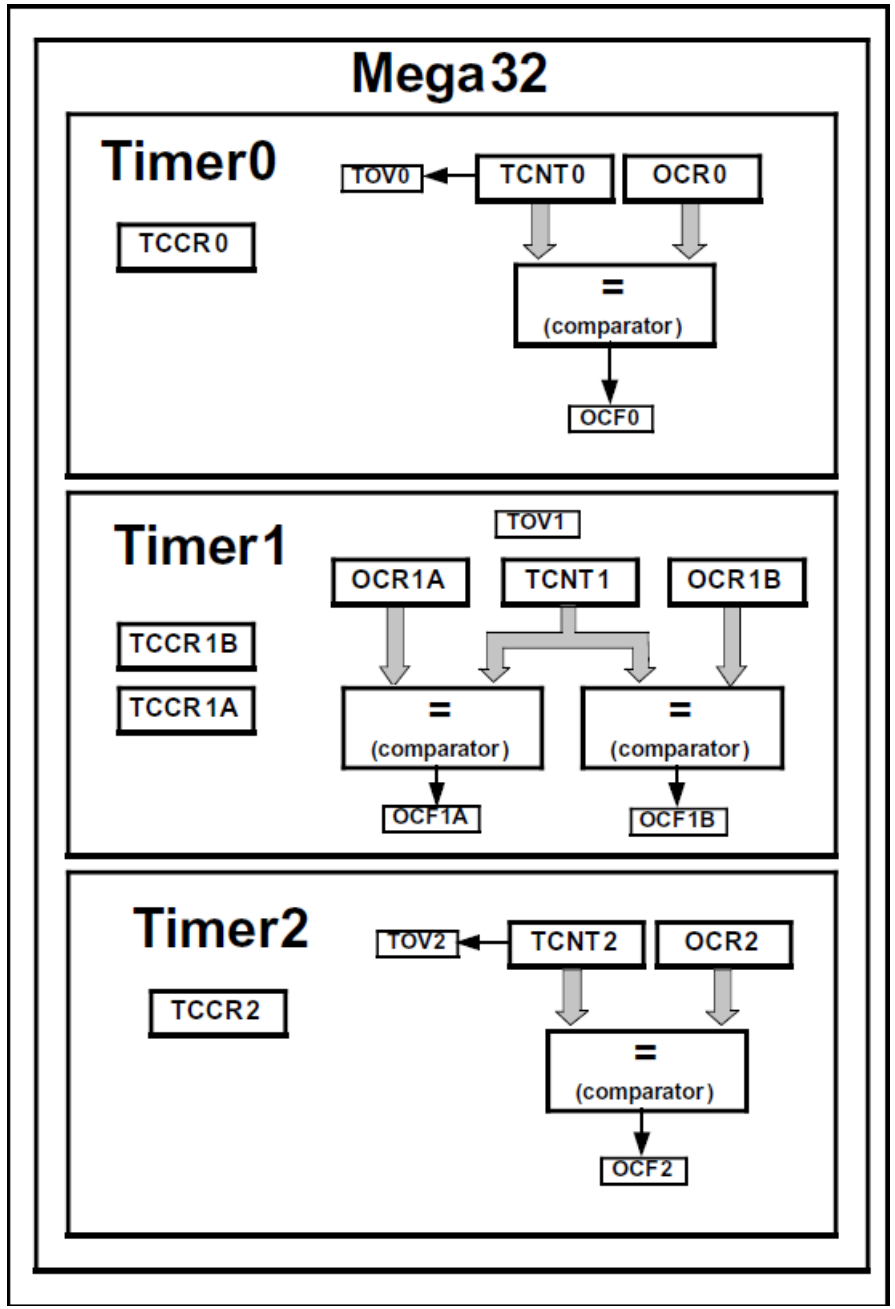
TCNTn: Timer/Counter Register

TOVn: Timer Overflow

TCCRn: Timer/Counter control register

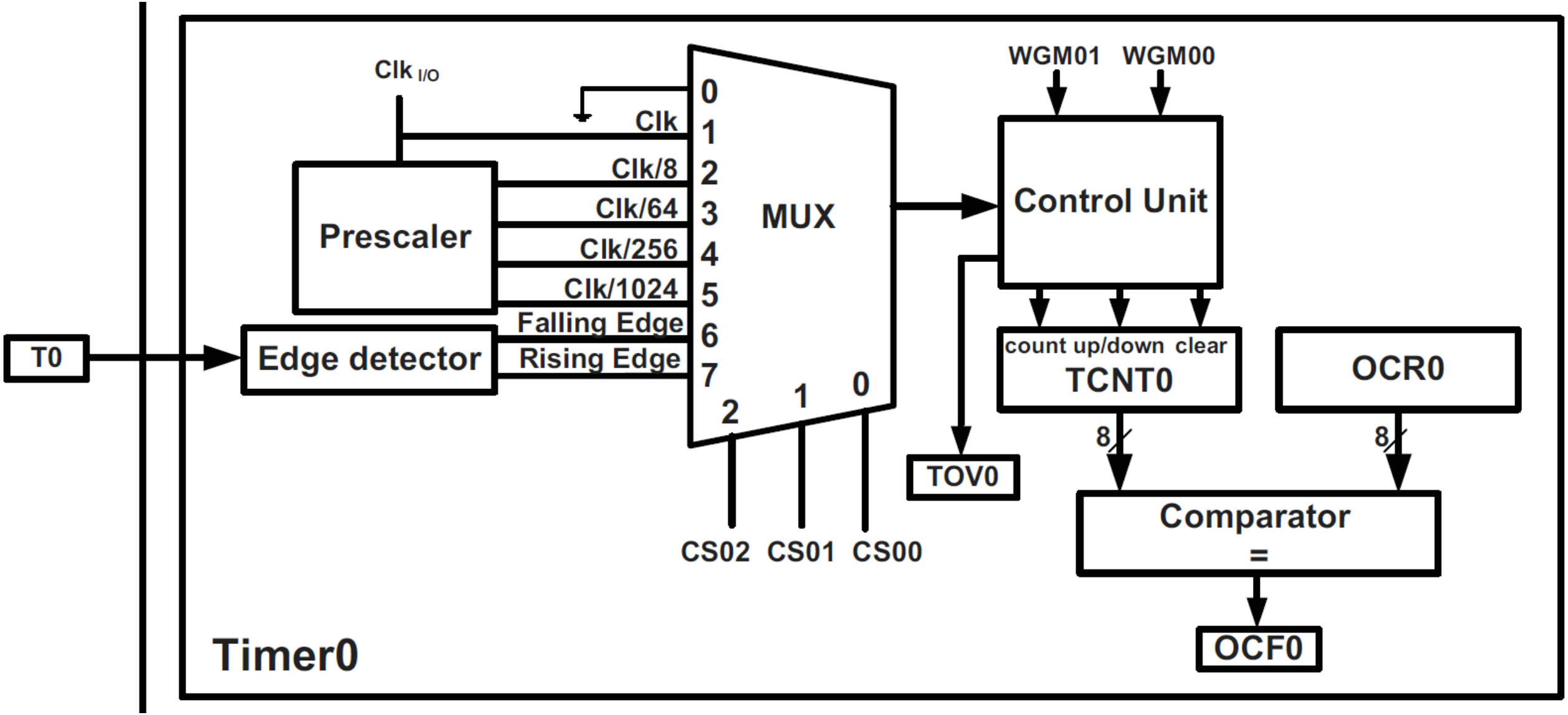
OCRn: Output Compare Register

OCFn: Output Compare Flag



Timer and Counter in AVR

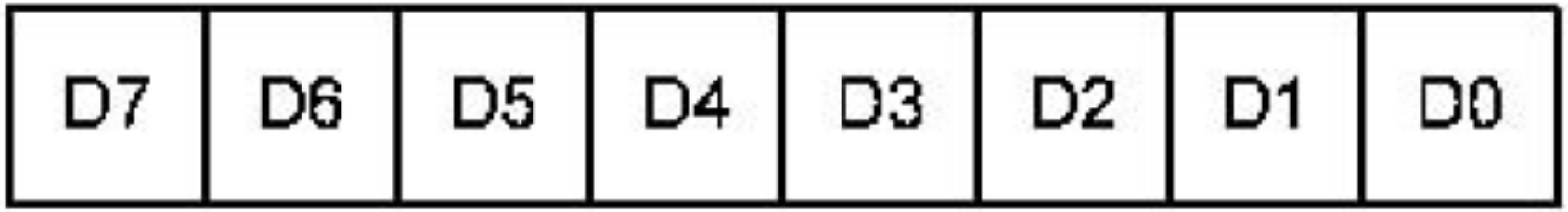
Timer0 programming



Timer and Counter in AVR

Timer0 programming

TCNT0: Timer/Counter Register0 is 8-bit register



TCCR0: Timer/Counter Control Register

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

FOC0: Force compare match: This is a write-only bit, which can be used while generating a wave. Writing 1 to it causes the wave generator to act as if a compare match had occurred.

Timer and Counter in AVR

Timer0 programming

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

WGM00, WGM01

WGM00	WGM01	Timer0 mode selector bits
0	0	Normal
0	1	CTC (Clear Timer on Compare Match)
1	0	PWM, Phase correct
1	1	Fast PWM

COM01:00 Compare Output Mode: These bits control the waveform generator.

Timer and Counter in AVR

Timer0 programming

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

CS02:00

D2	D1	D0	Timer0 Clock Selector
0	0	0	No Clock Source
0	0	1	Clk (No Prescaling)
0	1	0	Clk/8
0	1	1	Clk/64
1	0	0	Clk/256
1	0	1	Clk/1024
1	1	0	External clock source on T0 pin. Clock on falling edge.
1	1	1	External clock source on T0 pin. Clock on rising edge.

Timer and Counter in AVR

Timer0 programming

TIFR: Timer/Counter Interrupt Flag Register

7	6	5	4	3	2	1	0
OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0

TOV0: Timer0 overflow flag bit

0 = Timer0 did not overflow

1 = Timer0 has overflowed (from $FF to $00)

OCF0: Timer0 output compare flag bit

0 = compare match did not occur

1 = compare match occurred

Note: In AVR, when we want to clear a given flag of register, we write 1 to it and 0 to the other bits.

Timer and Counter in AVR

Timer0 Normal Mode Programming

STEP1: Load the TCNT0 register with the initial count value.

STEP2: Load the value into the TCCR0 register for required control setting.

STEP3: Keep monitoring the timer overflow flag (TOV0). Get out of the loop when TOV0 becomes high.

STEP4: Stop the timer by disconnecting the clock source.

STEP5: Clear the TOV0 flag for the next round.

Timer and Counter in AVR

```
#include <avr/io.h>

void T0delay();

int main(void)
{
		DDRB = 0xFF;			/* PORTB as output*/

		while(1)			/* Repeat forever*/
		{
			PORTB=0x55;
			T0delay();	/* Give some delay */
			PORTB=0xAA;
			T0delay();
		}
}

void T0delay()
{
		TCNT0 = 0x25;					/* STEP1:Load TCNT0*/
		TCCR0 = 0x01;			/*STEP2: Timer0, normal mode, no pre-scalar */

		while((TIFR&0x01)==0);  /* STEP3: Wait for TOV0 to roll over */
		TCCR0 = 0;			 /* STEP4: Disconnect clock source*/
		TIFR = 0x01;			/* STEP5: Clear TOV0 flag*/
}
```

Timer and Counter in AVR

Timer0 CTC Mode Programming

7	6	5	4	3	2	1	0
FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00

COM01, COM00

COM01	COM00	Description
0	0	The normal port operation
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

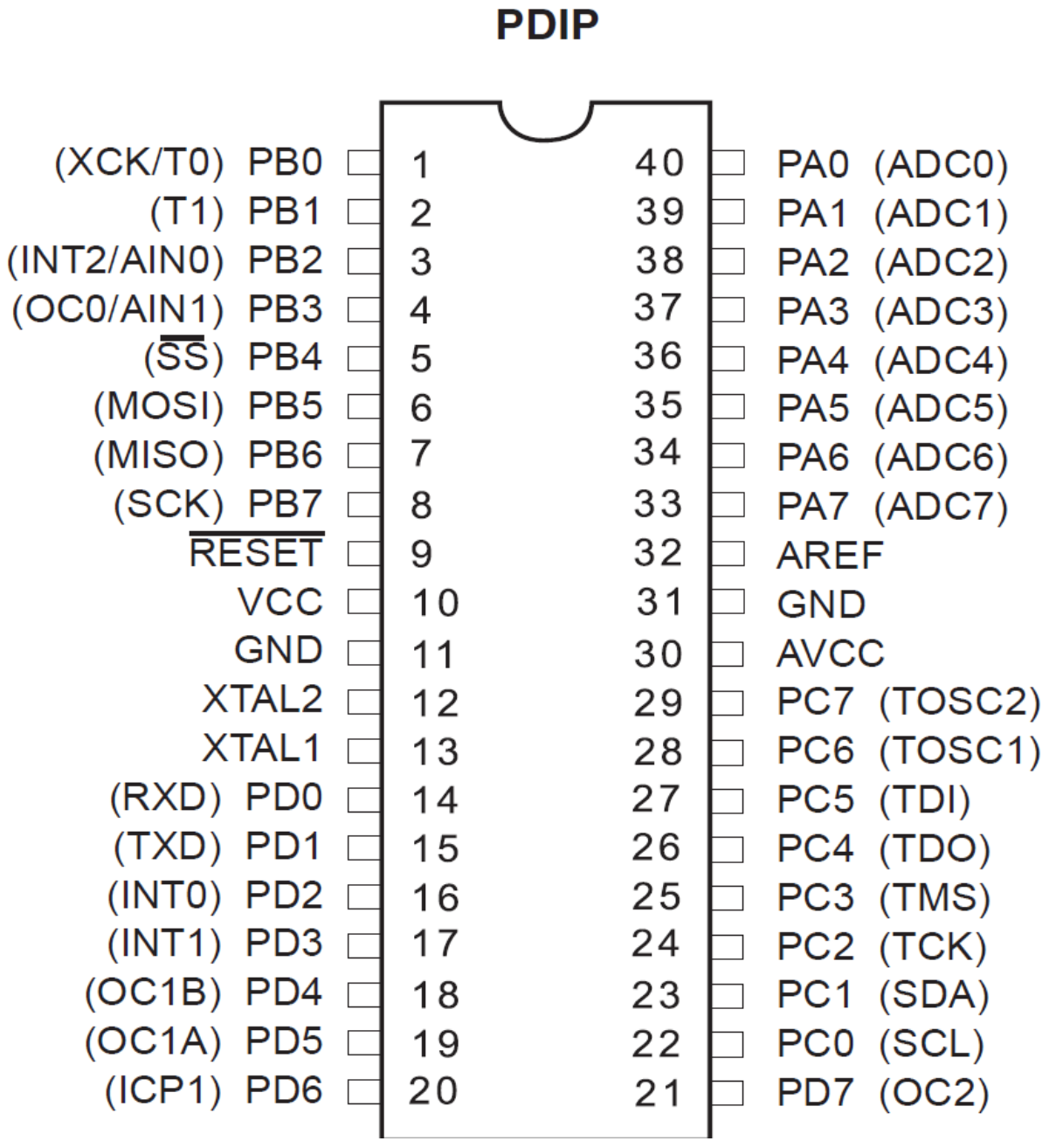
Timer and Counter in AVR

Timer0 CTC Mode Programming

STEP1: Configure OC0 pin (PB3) as output.

STEP2: Load the value into the TCCR0 register for required control setting.

STEP3: Load the required value in OCR0.



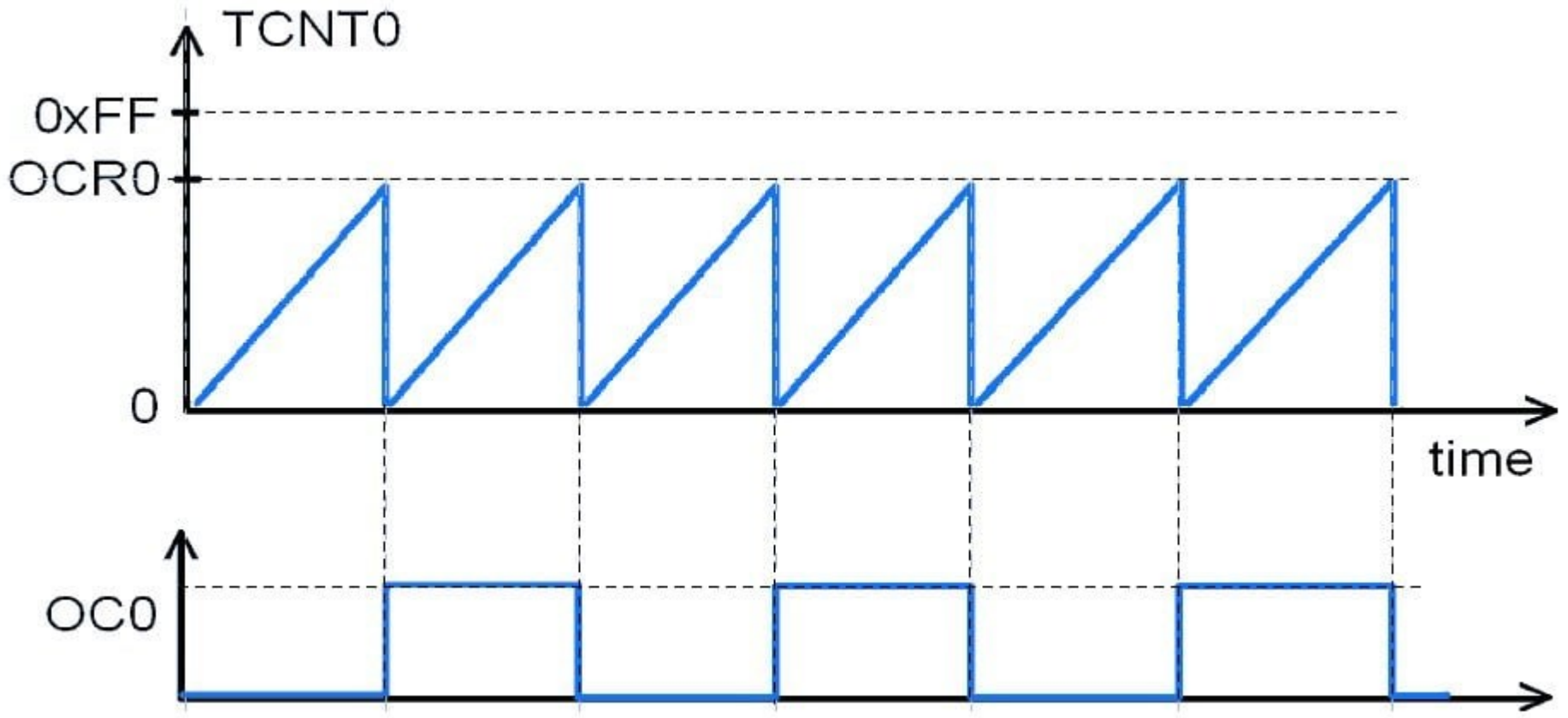
Timer and Counter in AVR

Normal mode waveform generation program

```
#include "avr/io.h"
int main ( )
{
        DDRB = DDRB|(1<<3);          /* PB3 (OC0) as output */
        TCCR0 = 0x19;                /* CTC mode, toggle on compare match,
                        clk- no pre-scaling */
        OCR0 = 200;                  /* compare value */
        while (1);
}
```

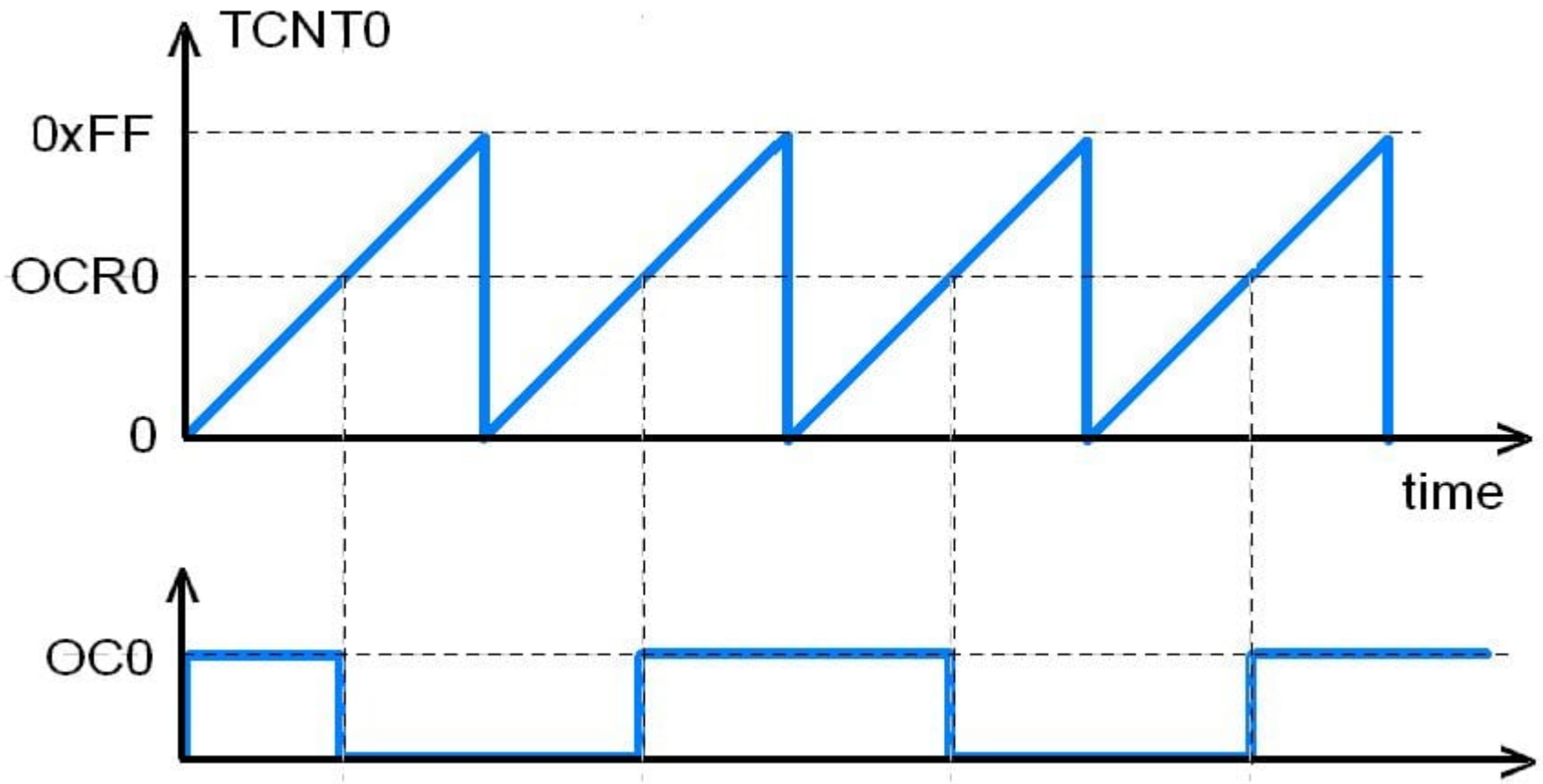
Timer and Counter in AVR

CTC mode waveform generation program



Timer and Counter in AVR

Normal mode waveform generation program



Syllabus of Embedded System

Unit 2: Programming for Embedded Systems (5 Hours)

2.1 Overview of AVR Architecture
2.2 Embedded C Programming for Microcontroller

- Introduction to C for Embedded Systems
- Data Types, Control Structures, and pointers
- Memory Management
- AVR Interrupt handling
- Input and output ports interfacing on AVR
- Timers and Counters in AVR
- Serial Communication in AVR