CONT…

- **PWM – Pulse width modulation** has many application in digital communication , power electronics, **auto intensity control of street lights**, **speed control of dc motor** and variable pwm to generate analog signal from digital signals, **digital to analog converter.**
- **USART**-**universal synchronous asynchronous receiver-transmitter**, is one of the most used device-to-device communication protocols.
- The **Serial Peripheral Interface** (**SPI**) is a synchronous serial communication interface specification used for short-distance communication, primarily in embedded systems.
- The three wires carry input data to that slave and output data from the slave and the timing clock.
- The developers from Motorola labeled the three wires MOSI (**master out/slave in**), MISO (**master in/slave out**), and SCK (**serial clock**).
- The **I2C** is known as inter-IC, which is developed by Philips for interfacing with the peripheral devices.
- A I2C bus is a bidirectional two-wired serial bus which is used to transport the data between integrated circuits.

CONT…

- The I2C is a serial bus protocol consisting of two signal lines such as SCL and SDL lines which are used to communicate with the devices.
- The SCL stands for a ‘**serial clock line**’ and this signal is always driven by the ‘master device’.
- The SDL stands for the ‘**serial data line**’, and this signal is driven by either the master or the I2C peripherals. Both these SCL and SDL lines are in open-drain state when there is no transfer between I2C peripherals.
- **PHY**-physical layer provides an electrical, mechanical, and **procedural interface to the transmission medium**.
- **TAG** (named after the Joint Test Action Group which codified it) is an industry standard for verifying designs and testing printed circuit boards after manufacture.
- JTAG implements standards for on-chip instrumentation in electronic design automation (EDA) as a complementary tool to digital simulation.
- A CAN DBC file (CAN database) is a **text file that contains information for decoding raw CAN bus data to 'physical values'**.

CONT…

- A **general-purpose input/output** (GPIO) is an uncommitted digital signal pin on an integrated circuit or electronic circuit board which may be used as an input or output, or both, and is controllable by software.
- GPIOs have no predefined purpose and are unused by default.
- **Embedded SRAM** is the **static random access memory** (SRAM) that an embedded system uses for its memory needs. SRAM is among the fastest memory available but is also expensive. Embedded engineers use it for critical functions that need fast and reliable memory.

The AVR Microcontroller

- **Choosing a Microcontroller**

There are five major 8-bit microcontroller from microchip technology:

1) Freescale semiconductors (formerly Motorola)
2) Intels 8051
3) Atmels AVR
4) PIC

- ➢ one microcontroller is not compatible with others and each microcontroller have unique instruction set, register set, etc
- ➢There are also 16-bit and 32bit microcontrollers by various chip makers and hence a designer must choose by certain criteria's
- ➢Thee are three criteria's in choosing microcontroller

1) Meeting the computing need of task at hand efficiently and cost effectively.
2) Availability of the software and hardware development tools such as compiler and assembler
3) Wide availability and reliable source of the microcontroller

Cont…

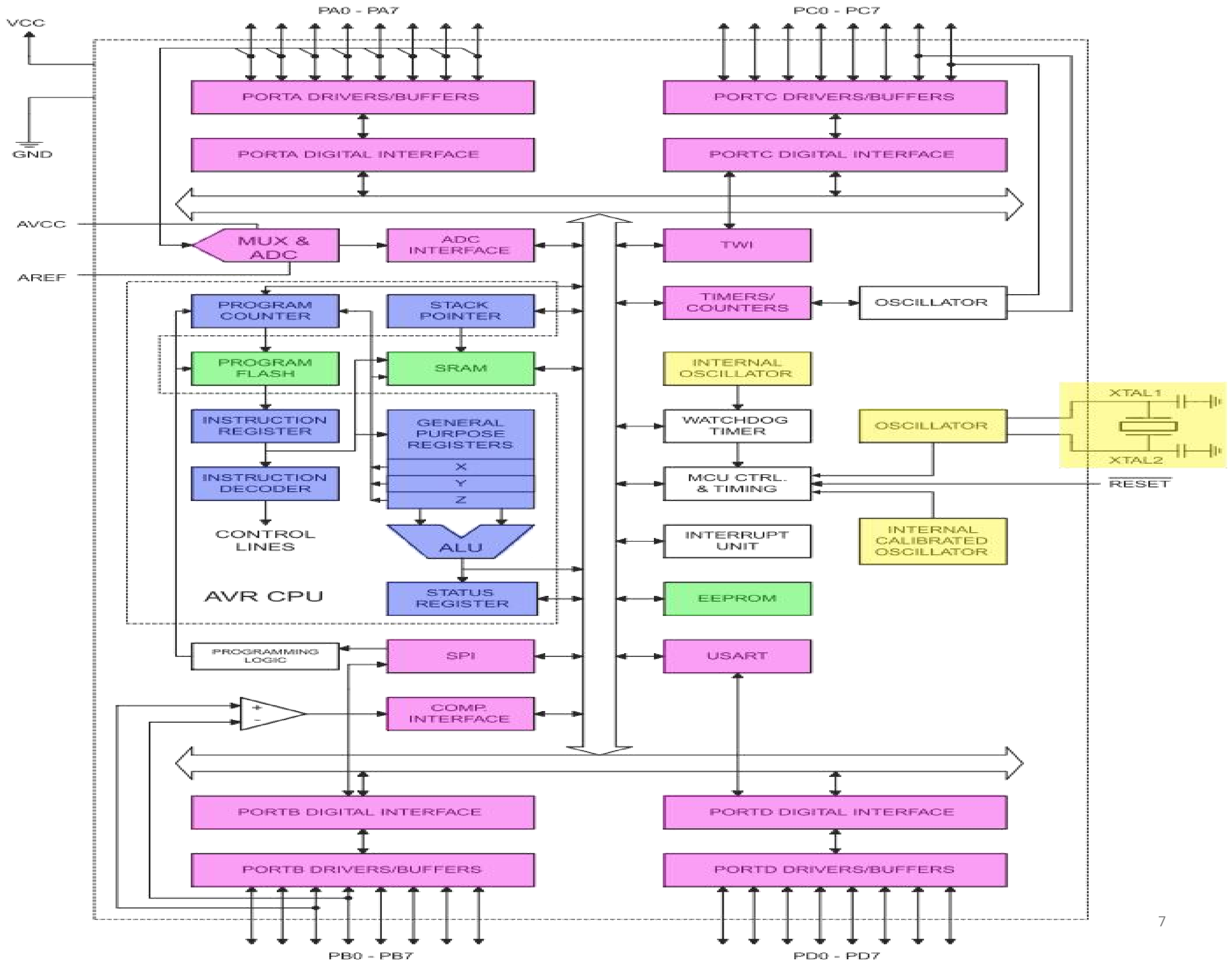
- The AVR microcontroller designed by two Norwegian institute of technology students; Alf-Egil Bowen and Vegard Wollan and then bought by atmel.
- Thus AVR stands for Advanced Virtual RISC **or** Alf and Vegard RISC
- There are many kinds of AVR microcontroller with different properties.
- Except AVR32, which is 32-it, all AVR are 8-bit
- AVR are generally classified into four broad groups:
  1) Mega
  2) Tiny
  3) Special purpose and
  4) Classic
- But, this course is focused on mega specially Atmega32 microcontroller

ATmega32 microcontroller Architecture

- The AVR microController is based on the advanced Reduced Instruction Set Computer (RISC) architecture.
- ATmega32 microController is a low power CMOS technology based controller. Due to RISC architecture AVR microcontroller can execute 1 million of instructions per second if cycle frequency is 1 MHz provided by crystal oscillator.

Key Features of ATmega32 microcontroller are:-

- 2 Kilo bytes of internal Static RAM
- 32 X 8 general working purpose registers
- 32 Kilo bytes of in system self programmable flash program memory.
- 1024 bytes EEPROM
- Programmable serial USART
- 8 Channel, 10 bit ADC
- One 16-bit timer/counter with separate prescaler, compare mode and capture mode.
- Available in 40 pin DIP, 44-pad QFN/MLF and 44-lead QTFP
- Two 8-bit timers/counters with separate prescalers and compare modes
- 32 programmable I/O lines
- In system programming by on-chip boot program
- Master/slave SPI serial interface
- 4 PWM channels
- Programmable watch dog timer with separate on-chip oscillator


PA0 - PA7
PC0 - PC7
VCC
GND
PORTA DRIVERS/BUFFERS
PORTC DRIVERS/BUFFERS
PORTA DIGITAL INTERFACE
PORTC DIGITAL INTERFACE
AVCC
MUX & ADC
ADC INTERFACE
TWI
AREF
PROGRAM COUNTER
STACK POINTER
TIMERS/ COUNTERS
OSCILLATOR
PROGRAM FLASH
SRAM
INTERNAL OSCILLATOR
XTAL1
INSTRUCTION REGISTER
GENERAL PURPOSE REGISTERS
X
Y
Z
WATCHDOG TIMER
OSCILLATOR
XTAL2
INSTRUCTION DECODER
MCU CTRL. & TIMING
RESET
CONTROL LINES
ALU
INTERRUPT UNIT
INTERNAL CALIBRATED OSCILLATOR
AVR CPU
STATUS REGISTER
EEPROM
PROGRAMMING LOGIC
SPI
USART
+
-
COMP. INTERFACE
PORTB DIGITAL INTERFACE
PORTD DIGITAL INTERFACE
PORTB DRIVERS/BUFFERS
PORTD DRIVERS/BUFFERS
PB0 - PB7
PD0 - PD7

Cont…

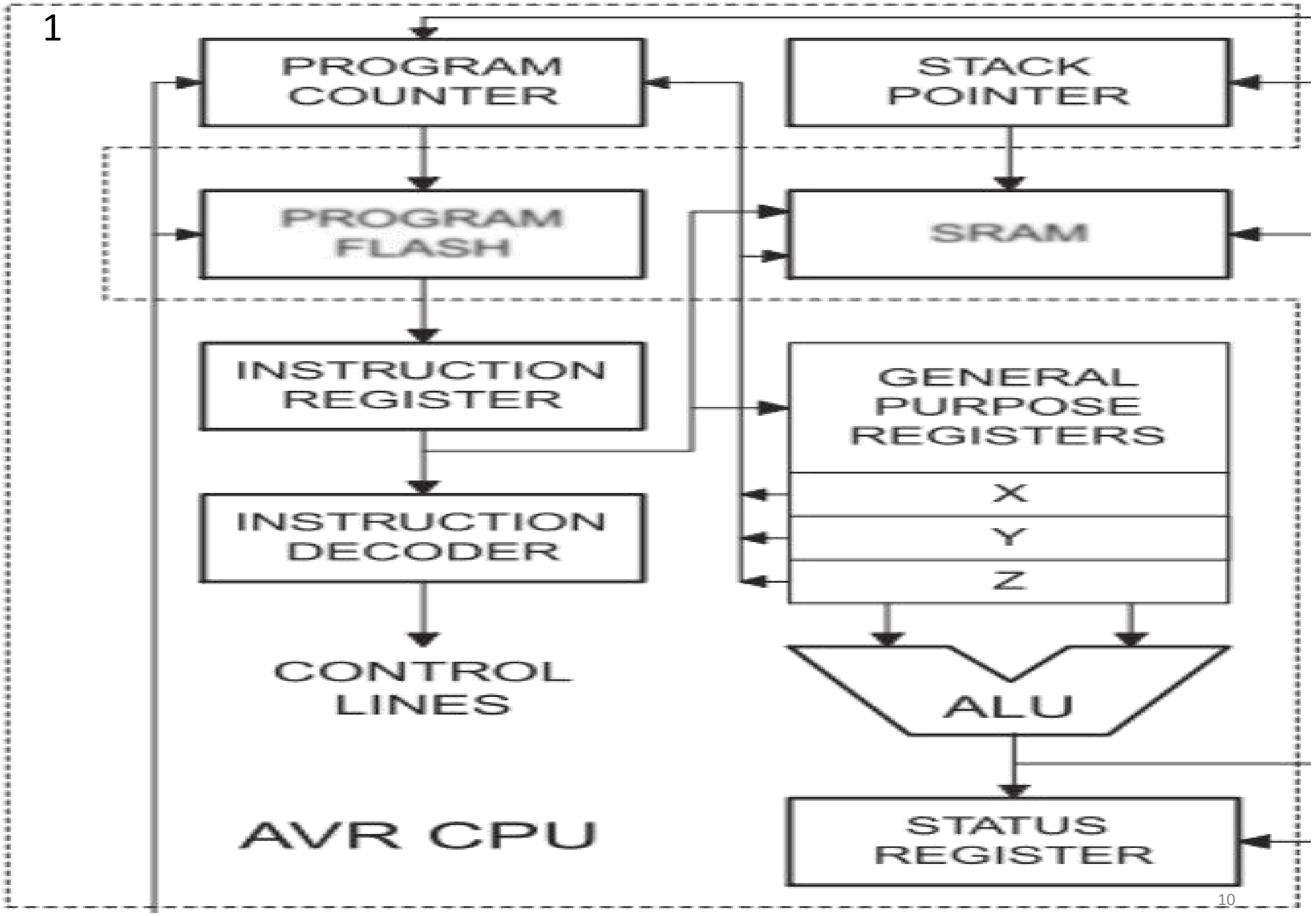
- The CPU components are shaded blue.
- The memory components are shaded green.
- The clock components are shaded in orange.
- The I/O components are shaded in purple.
- **<u>ATmega32 Highlights</u>**
- Native data size is 8 bits (1 byte).
- Uses 16-bit data addressing allowing it to address $2^{16} = 65536$ unique addresses.
- Has three separate on-chip memories
  - 2KiB SRAM
    - 8 bits wide
    - used to store data
  - 1KiB EEPROM
    - 8 bits wide
    - used for persistent data storage
  - 32KiB Flash
    - 16 bits wide
    - used to store program code
- I/O ports A-D
  - Digital input/output
  - Analog input
  - Serial/Parallel
  - Pulse accumulator

Cont…

- **Programmers Model**
- languages like C, C++, and Java require little, if any, knowledge of the CPU's internal configuration.
- Assembly language requires knowledge of the internals of the CPU since we are operating at a lower level.
- Machine language is the native language of the CPU
  - Consists only of 1's and 0's.
  - Is what we actually download to the controller's program memory.
  - Is stored in the .hex file generated by AVR studio.
- Program hierarchy:
  - **High-level** language gets converted into **assembly** language by a compiler.
  - **Assembly** language gets converted into **machine** language by an assembler.
  - AVR Studio acts as our assembler for this course.

CPU and Registers

1

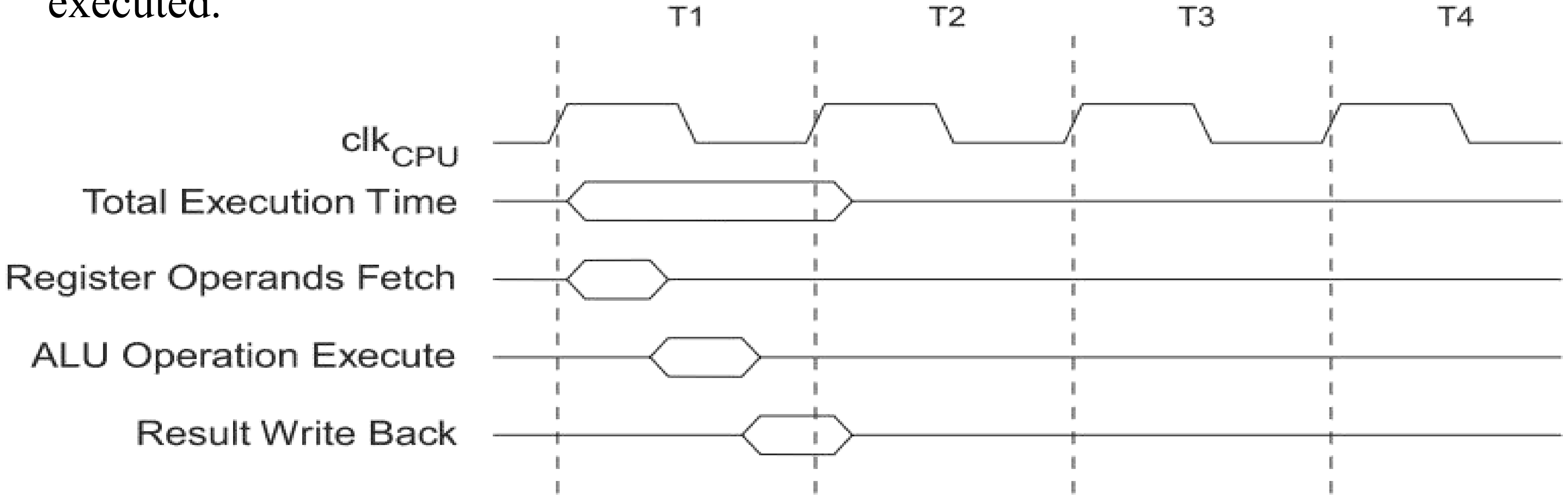


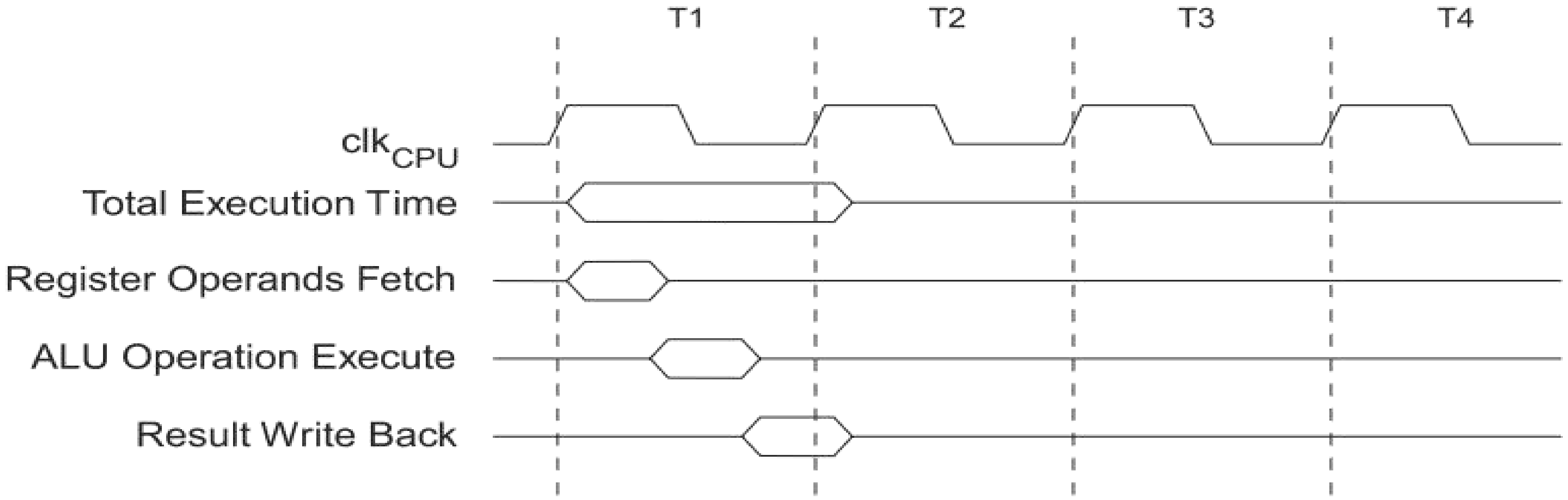
Cont..

Steps to execution

- **Program Counter** fetches program instruction from memory
  - The PC stores a program memory address that contains the location of the next instruction.
  - The PC is initialized to 0x0 on reset/power up.
    - When we placed the bootloader on the chip, we set a fuse that enables a reset vector to load the PC with the address where the bootloader program starts.
  - When the program begins, the PC must contain the address of the first instruction in the program.
  - Program instructions are stored in consecutive program memory locations.
  - The PC is automatically incremented after each instruction.
  - Note: There are jump instructions that can modify the PC (e.g., the PC must change when calling or returning from some other routine).
- Places instruction in **Instruction Register**.
- **Instruction Decoder** determines what the instruction is.
- **Arithmetic Logic Unit** executes the instruction.:

The CPU clock determines the timing of when instructions are fetched and executed.





Cont…

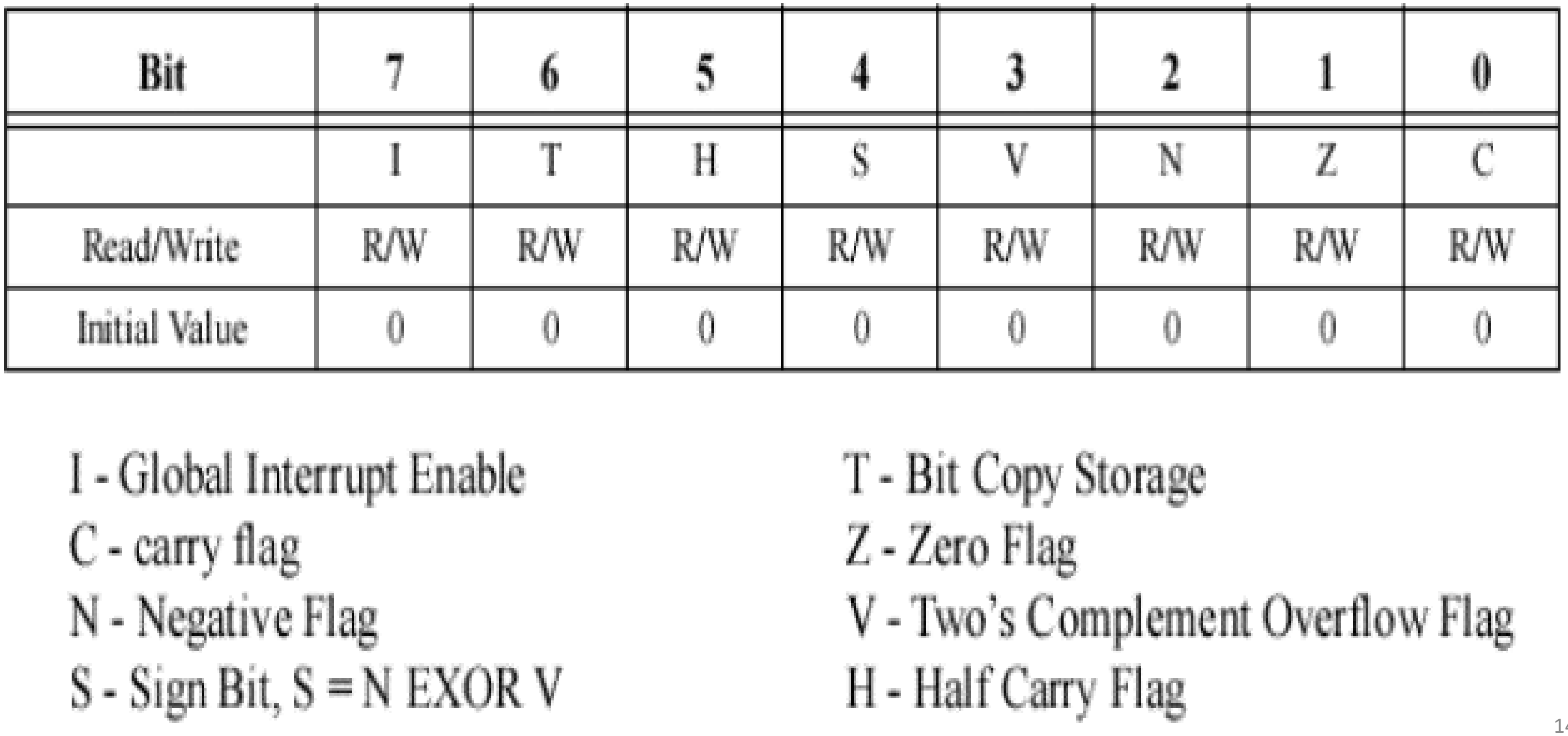
Registers

- Directly accessed by the CPU/ALU — very fast.
- Registers contain:
  - Address of the next instruction to fetch from program memory (PC).
  - Machine instruction to be executed (IR).
  - "Input" data to be operated on by the ALU.
  - "Output" data resulting from an ALU operation.
- **General Purpose Registers**
- There are 32x8 bit general purpose registers, **R0-R31**.
- **X**, **Y**, and **Z** are 16 bit registers that overlap **R26-R31**.
  - Used as address pointers.
  - Or to contain values larger than 8 bits (i.e., >255).
  - Only register **Z** may be used for access to program memory.
- Certain operations cannot be performed on the first 16 registers, **R0-R15**
  - *LDI*
  - *ANDI*
  - etc...
- *CLR* **Rx** does work on all registers.

Cont..

Other Registers

- Stack pointer (SP)
  - Is 16 bits wide.
  - Stores return address of subroutine/interrupt calls.
  - May be used to store temporary data and local variables.
- Status Register (SREG)



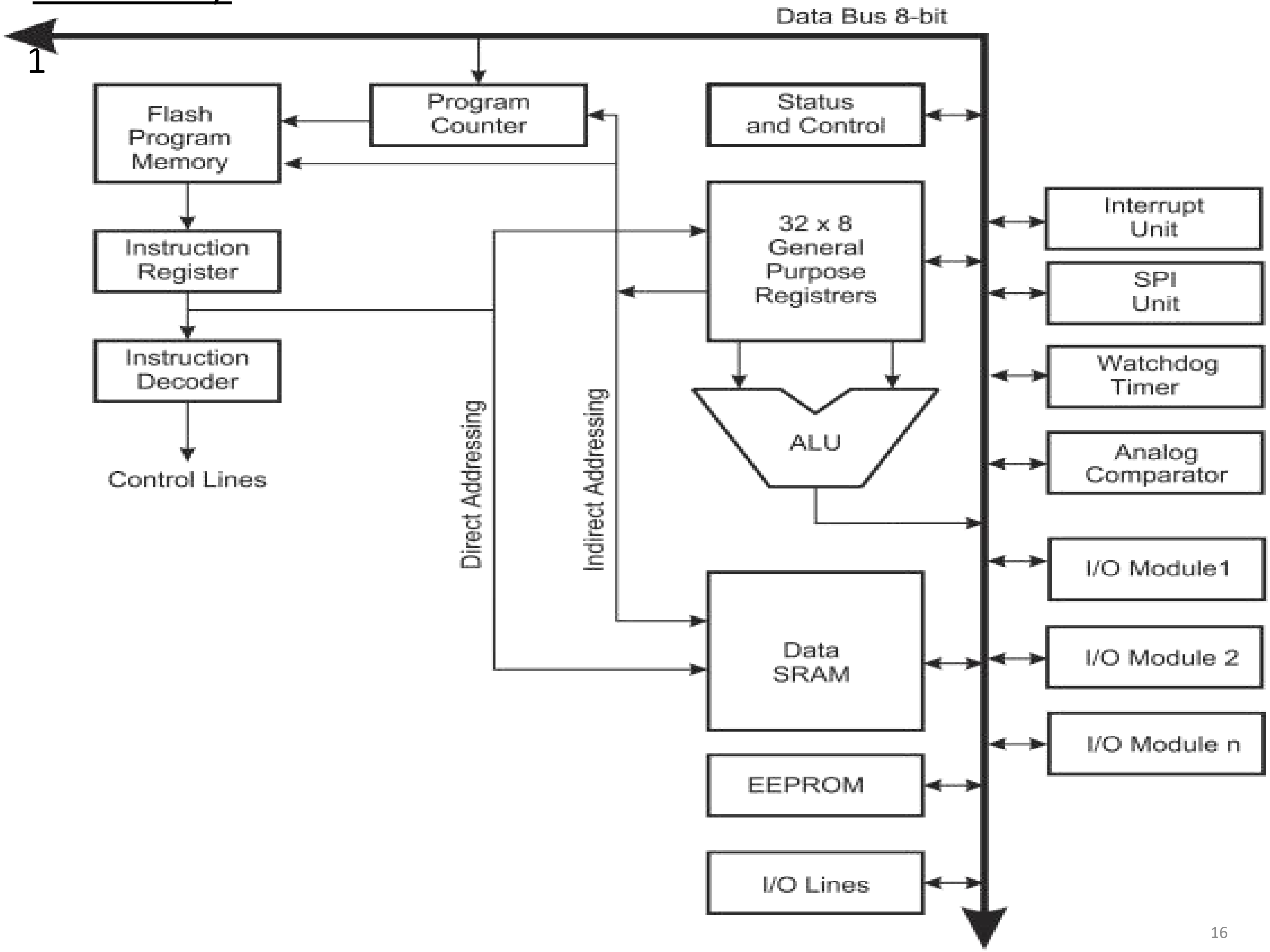
Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

I - Global Interrupt Enable
C - carry flag
N - Negative Flag
S - Sign Bit, S = N EXOR V
T - Bit Copy Storage
Z - Zero Flag
V - Two's Complement Overflow Flag
H - Half Carry Flag

Cont…

- Is 8 bits wide.
- Contains information associated with the results of the most recent ALU operation.
  - **Carry flag (C**) set if CY from most signficant bit (MSB).
    - Often used when adding numbers that are larger than 8 bits. Here we need to use an add with carry instruction (*ADC*) to add the next most significant byte.
  - **Zero flag (Z)** set if result is zero.
    - *BREQ* instruction says branch if zero flag is set.
    - *BRNE* instruction says branch if zero flag is not set.
  - **Negative flag (N)** set if MSB is one.
    - Arithmetic instructions change the flags
    - Data transfer instructions do not change the flag.
    - The instruction set documentation identifies which flags each instruction may modify.
  - **Overflow flag (V)** set if 2's complement overflow occurs.
    - An overflow occurs if you get the wrong sign for your result, e.g., $64_{10} + 64_{10} = -128_{10}$ ($01000000_2 + 01000000_2 = 10000000_2$).
    - Note: whenever the carry into the MSB and the carry out don't match, we have an overflow.
  - **Sign bit (S) s** = N EXOR V.
  - **Half carry flag (H**) is set when carry occurs from b_3 to b_4.
    - Used with binary coded decimal (BCD) arithmetic.

Memory



Data Bus 8-bit
1
Flash Program Memory
Program Counter
Status and Control
Instruction Register
Instruction Decoder
Control Lines
32 x 8 General Purpose Registrers
ALU
Direct Addressing
Indirect Addressing
Data SRAM
EEPROM
I/O Lines
Interrupt Unit
SPI Unit
Watchdog Timer
Analog Comparator
I/O Module1
I/O Module 2
I/O Module n

Cont…

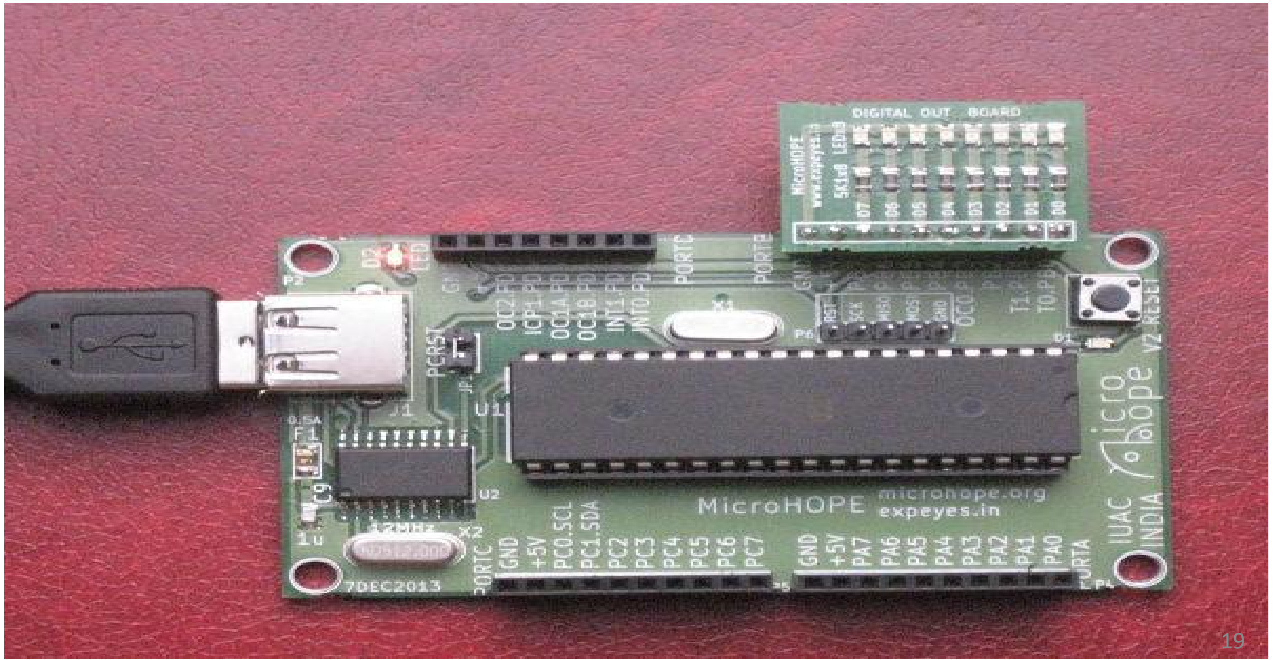
- The address bus is 16 bits wide.
- The data bus is 8 bits wide.
- Program memory is stored on Flash from 0x0000 to 0x3FFF (*F_END*).
  - Our boot loader is loaded in the last 1KiB of the Flash memory.
- SRAM is used for:
  - 32 General Purpose registers from 0x00 to 0x1F.
  - 64 I/O ports from 0x20 to 0x5F.
  - User data from 0x60 to 0x85F (*RAMEND*).
  - Note: the bootloader uses the last 32 bytes of data memory, we will consider *RAMEND*-0x20 as the end of data RAM.
- ROM for user data is stored on EEPROM.
  - ATmega32 has 1KiB of storage from 0x000 to 0x3FF (*E_END*).

Assembly language Programming with ATmega32 Instruction Set

- To code in assembler, one should have some idea about the architecture of the target hardware.
- It is enough to assume that the AVR micro-controller appears to the programmer as a set of General Purpose Registers (GPRs: R1 to R31), Special Functions Registers (SFRs) that controls the peripherals, some data memory (2kbytes of SRAM for Atmega32).
- All of them are in the data address space.
- We also have Program memory and EEPROM in different address spaces.
- Assembly language programming involves moving data between GPRs, SFRs and RAM, and performing arithmetic and logical operations on the data.
- We have four I/O ports (A,B,C and D, controlled by 12 SFRs) that can be used for providing input data to our programs and display the output.
- In order to do that, we need to use switches and LEDs connected to these ports. The following examples use the Digital Output Board (having 8 LEDs) to display program results. The figure shows the board plugged into port B.

Cont…

- Let us start with a small program shown below (**immed.S** is included in the examples provided.)
-
```
    .section .text    ; denotes code section
    .global main
main:
    ldi r16, 255      ; load r16 with 255
    out 0x17, r16  ; send R16 to DDRB using I/O address
    out 0x18, r16  ; and to PORTB also
    .end
```
Clicking on 'Assemble' and then 'Upload' should make all the LED light up.



Cont…

General Purpose Registers

- We are already familiar with the Special Function Registers (including DDRB and PORTB) that were used to configure and control various features of the microcontroller.
- In addition to these ATmega32 has 32 general purpose registers (32 here is a coincidence.
- The 32 in ATmega32 refers to 32 kB of flash memory available for programming.
- All AVR micro-controllers, even ATmega8 or ATmega16 have 32 GPRs).
- Any numerical value that needs to be used in the program needs to be first loaded into one of the GPRs. So, if you want to load 0xff into DDRB, you first need to load 0xff into a GPR and then copy the content of the GPR into DDRB.
- This might seem like an unnecessary restriction to us who have been used to writing DDRB=0xff in C, but it is a necessary consequence of the streamlined hardware design of the processor which C hides from us.
- Even though the 32 registers R0-31 are called "general purpose", there are special uses for some of them, which will be discussed later.

Cont…

Instructions

- What we could do intuitively with an assignment operator (=) in C requires the use of more than one instruction.
- **LDI** (Load Immediate) : used to load a constant value to one of the registers R16-31 (that's a restriction. load immediate can't work with R1 ro R15)
- **OUT** (output to any Special Function Register) : The SFRs are mapped to the locations 0 to 3Fhex. 0x17 and 0x18 are the I/O mapped addresses of the registers DDRB and PORTB respectively.
- The SFRs are also mapped into the memory space to locations 20hex to 5Fhex.
- Due to this reason you can use **STS** (Store Direct to SRAM) instruction instead of **OUT** but to a different address.
- **OUT 0x17, R16** and **STS 0x37, R16** achieves the same result but the former is compact.

Adding two numbers

- The code listed below (add.S) adds two numbers and displays the result on the LEDs connected to port B.
- Instead of remembering the addresses of DDRB and PORTB, we have included the file 'avr/io.h' that contains all the Register names. Naming the program with **.S** (capital S instead of small s **)** invokes the pre-processor, that also allows expressions like (1 << PB3) to be used. (add.S)

Cont…

- #include <avr/io.h>

```
        .section .text    ; denotes code section
        .global main
main:
      LDI     R16, 255        ; load R16 with 255
      STS    DDRB, R16     ; set all bits of port B as output
      LDI    R16, 2       ; load R16 with 2
      LDI    R17, 4       ; load R17 with 4
      ADD  R16, r17     ; R16 <- R16 + R17
      STS    PORTB, R16     ; result to port B
      .END
```

- Running this program lights LEDs D2 and D3.
- **The Status Register**

	7	6	5	4	3	2	1	0
SREG	I	T	H	S	V	N	Z	C
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	0	0	0	0	0	0	0	0

- *Bit 0 : CarryBit 1 : ZeroBit 2 : NegativeBit 3 : Two's complement overflowBit 4 : Sign bit, exclusive OR of N and VBit 5 : Half Carry*

Cont….

- Let us modify the previous program to evaluate 255 + 1.
- The result will be shown on port B and the status flag register SREG on port A. (carry.S)
- #include <avr/io.h>

```
        .section .text    ; denotes code section
        .global main
main:
      LDI     R16, 255
      STS    DDRB, R16        ; All bits of port B as output
      STS    DDRA, R16        ; All bits of port A as output
      LDI    R17, 1           ; load R17 with 1,  R16 already has 255
      ADD  R16,  R17          ; R16 <- R16 + r17
      STS    PORTB, R16       ; Sum to port B
      LDS   R16, SREG         ; Load the Status register
      STS    PORTA, R16       ; display it on port A
      .END
```

Cont…

- The Carry, Zero and Half Carry bits will be set on port B.
- Exercise 1: Load R16 and R17 with two numbers and study the results and status flags generated by the following operations.
- *COM   R16 ; Complement*
- *NEG   R16 ; 2's complement*
- *TST   R16 ; test for zero or minus*
- *AND R16, R17; bitwise AND*
- *OR   R16, R17 ; bitwise OR*
- *ADD R16, R17 ; summingExercise 2: Add a number and it's 2's complement, do the same with 1's complement and compare the results*
- *LDI   R16, 10   ; load number*
- *MOV   R17, R16 NEG   R16; 2's complement*
- *ADD R17, R16*

Cont…

Moving Data

- *To manipulate data, we need to bring them into the GPRs (R1 to R31) and the results should be put back into the memory locations.*
- *There are different modes of transferring data between the GPRs and the memory locations, as explained below.*
- *Register Direct: MOV R1, R2 ; copies R2 to R1 .*
- *Two GPRs are involved in this.*
- *There are also operations that involves a single register like, INC R1*
- *I/O Direct : For moving data between the GPRs and the SFRs, since the SFRs can be accessed as I/O addresses.*
- *OUT 0x17, R1 copies R1 to DDRB.*
- *Please note that the I/O address is 20hex less than the memory mapped address (0x37) of the same SFR. (io-direct.S)*
- *Immediate : This mode can be used for transferring a number to any register from R16 to R31, like : LDI R17, 200.*
- *The data is provided as a part of the instruction. (immed.S)*
- *Data Direct: In this mode, the address of the memory location containing the data is specified, instead of the data itself.*
- *LDS R1, 0x60 moves the content of memory location 0x60 to R1. STS 0x61, R1 copies R1 to location 0x61. (data-direct.s)*

Cont…

- Data Indirect : In the previous mode, the address of the memory location is part of the instruction word.
- Here the address of the memory location is taken from the contents of the X, Y or Z registers.
- X, Y and Z are 16 bit registers made by combining two 8 bit registers (X is R26 and R27; Y is R28 and R29; Z is R30 and R31. This is required for addressing memory above 255. (data-indirect.s)
- LDI R26, 0x60 ; address of location 0x0060 toX
- LDI R27, 0x00
- LD   R16, X ; load R16 with the content of memory location pointed to by X
- This mode has several variations like pre and post incrementing of the register or adding an offset to it. Refer to the data book for details.

Cont…

Programs having Data

- Programs generally have variables, sometimes with initialized data.
- The expects them inside the .data segment. The following example shows how to access a data variable using direct and indirect modes. (data-direct-var.S)
-

```
#include <avr/io.h>
        .section .data ; data section starts here
var1:   .byte  0xEE; initialized global variable var1 .section .text ; code
section
        .global     __do_copy_data    ; initialize global variables
        .global     __do_clear_bss    ; and setup stack pointer
        .global main
main:
  LDS  R1, var1; load R1 using data direct mode
  STS   DDRA, R1; display R1 on port A
  STS   PORTA, R1
   LDI   R26, lo8(var1); load the lower and
   LDI   R27, hi8(var1); higher bytes of the address of var1 to X
   LD    R16, X  ; Load R16 using data-indirect mode,  data from where
X is pointing to
        STS   DDRB, R16 ; display R16 on port B
        STS   PORTB, R16;
        .end
```

- The lines .global   __do_copy_data and .global   __do_clear_bss tells the assembler to insert code for initializing the global variables, which is a must.

Cont…

- **Jumps and Calls**
- The programs written so far has an execution flow from the beginning to the end, without any branching or subroutine calls, generally required in all practical programs. The execution flow can be controlled by CALL and JMP instructions. (call-jump.S)

```
#include <avr/io.h>
        .section .text    ; code section starts
disp: ; our  subroutine
        STS    PORTB, R1; display R1 on port B
        INC    R1 ; increments it
        RET    ; and return
        .global main
main:
        LDI,   R16, 255
        STS   DDRB, R16
        MOV  R1, R16
loop:
        RCALL  disp          ; relative call
        CALL    disp          ; direct call
        RJMP    loop
        .end
```

- *The main program calls the subroutine in a loop, the data is incremented on each call. Use an oscilloscope to view the voltage waveform at each LEDs.*

Cont…

Output of the Assembler

- The code we write are translated by the assembler into machine language instructions.
- Then it is passed on to the Linker to decide the locations to which code and data are to be stored before executing it.
- The code is stored into the Program memory.
- Even though the processor starts from location zero on a reset, the linker places the addresses of the interrupt vectors there, then some initialization code and after that our code is placed. You can explore the .lst output to know the details.

Interrupts, the asynchronous Calls

- There are situations where the uC should respond to external events, stopping the current program temporarily.
- This is done using Interrupts, that are external signals, either from the I/O pins or from from some of the peripheral devices.
- On receiving an interrupt signal, the processor stores the current Program Counter to the memory location pointed to by the Stack Pointer and jumps to the corresponding interrupt vector location (For example, the processor will jump to location 0x0002 (0x0004 if you count them as bytes), if external interrupt pin INT0 is activated, provided the interrupt is enabled by the processor beforehand. (interrupt.S).
- Connect PD2 to ground momentarily and watch the LEDs.

Cont…

```
• .section .text    ; denotes code section
•          .global __vector_1 ; INT0_vect
• __vector_1:                ; Interrupt Service Routine of INT0. Called when PD2 is LOW.
•        INC R1
•        OUT   0x18, R1
•        RETI                ; return from interrupt
•        .global main
• main:
•         LDI     R16, 255
•         OUT  0x17, R16   ; DDRB
•         OUT  0x12, R16   ; enable Port D pull-up resistors
•         LDI    R16, 0x40   ; enable
•         OUT  0x3b, r16    ; interrupt INT0
•         CLR  R1
•         SEI  ; enable interrupts globally
• loop:
•         RJMP loop  ; infinite loop
•         .end
```

What is an Interrupt?

- An interrupt is a signal (an “interrupt request”) generated by some event external to the CPU.
- Causes CPU to stop currently executing code and jump to separate piece of code to deal with the event.
- Interrupt handling code often called an ISR (“Interrupt Service Routine”).
- When ISR is finished, execution returns to code running prior to interrupt.
- Varying number of ISRs may be supported ▫ Typically between 1-16 depending on platform
- • Multiple events may be mapped to a single routine i.e. any PORTA pin change
- Interrupt events may have priorities
- ISR is implemented inside a function with no parameters and no return value (void)
- Typically keep interrupt routines shorter than 15-20 lines of code

Interrupt Sources

- **Hardware Interrupts**
- Commonly used to interact with external devices or peripherals
- Microcontroller may have peripherals on chip
- **Software Interrupts**
- Triggered by software commands, usually for special operating system tasks i.e. switching between user and kernel space, handling exceptions
- Common hardware interrupt sources
- Input pin change
- Hardware timer overflow or compare-match
- Peripherals for serial communication
- UART, SPI, I2C – Rx data ready, tx ready, tx complete
- ADC conversion complete
- Watchdog timer timeout

Coding Interrupts in AVR C

- Include interrupt macros and device vector definitions
- #include <avr/interrupt.h>
- #include <avr/io.h>
- Define ISR using macro and appropriate vector name: by default the
- compiler determines all registers that will be modified and saves them
- (prologue code) and restores them for you (epilog)
- ISR(UART0_RX_vect){
- // ISR code to execute here
- }
- Somewhere in main or function code
- sei(); //Enable global interrupts
- Enable specific interrupts of interest in corresponding control registers. Ex:
- UCSRB |= (1<<RXCIE); //enable interrupt