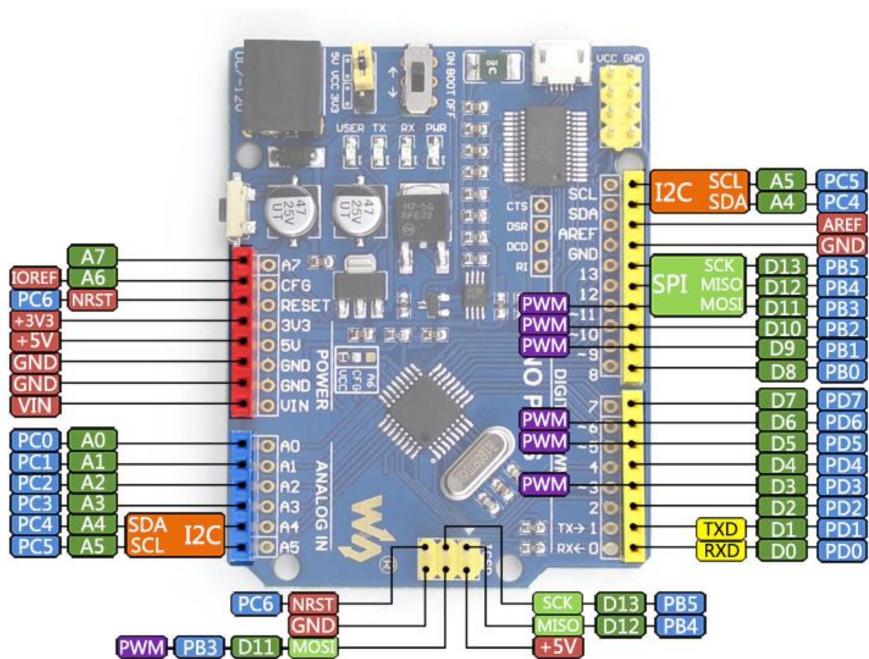
LESSON 1

ALL ABOUT ARDUINO ASSEMBLY PROGRAMMING

ATMega328P Features



First, it has 6 Analog input pins, and its memory can go up to 32kb. Also, it has 2kb of SRAM and up to 1kb of EEPROM.

Then, its clock speed stands at 16 Megahertz, and it has a total of 14 I/O pins.

The chip has 3 ports, B,C, & D. Port D gives us access to all 8 bits while B & C provide 6 bits each.

Internal Registers

ATMega32 Programmer Model: Registers (GPRs)

"Six of the 32 registers can be used as three 16-bit indirect address register pointers for Data Space addressing –enabling efficient address calculations. One of the these address pointers can also be used as an address pointer for look up tables in Flash Program memory. These
added function registers are the 16-bit X-register, Y-register and Z-register, described later."

7 0	Addr.	
R0	$00	
R1	$01	
R2	$02	
...		
R13	$0D	
R14	$0E	
R15	$0F	
R16	$10	
R17	$11	
...		
R26	$1A	X-register Low Byte
R27	$1B	X-register High Byte
R28	$1C	Y-register Low Byte
R29	$1D	Y-register High Byte
R30	$1E	Z-register Low Byte
R31	$1F	Z-register High Byte

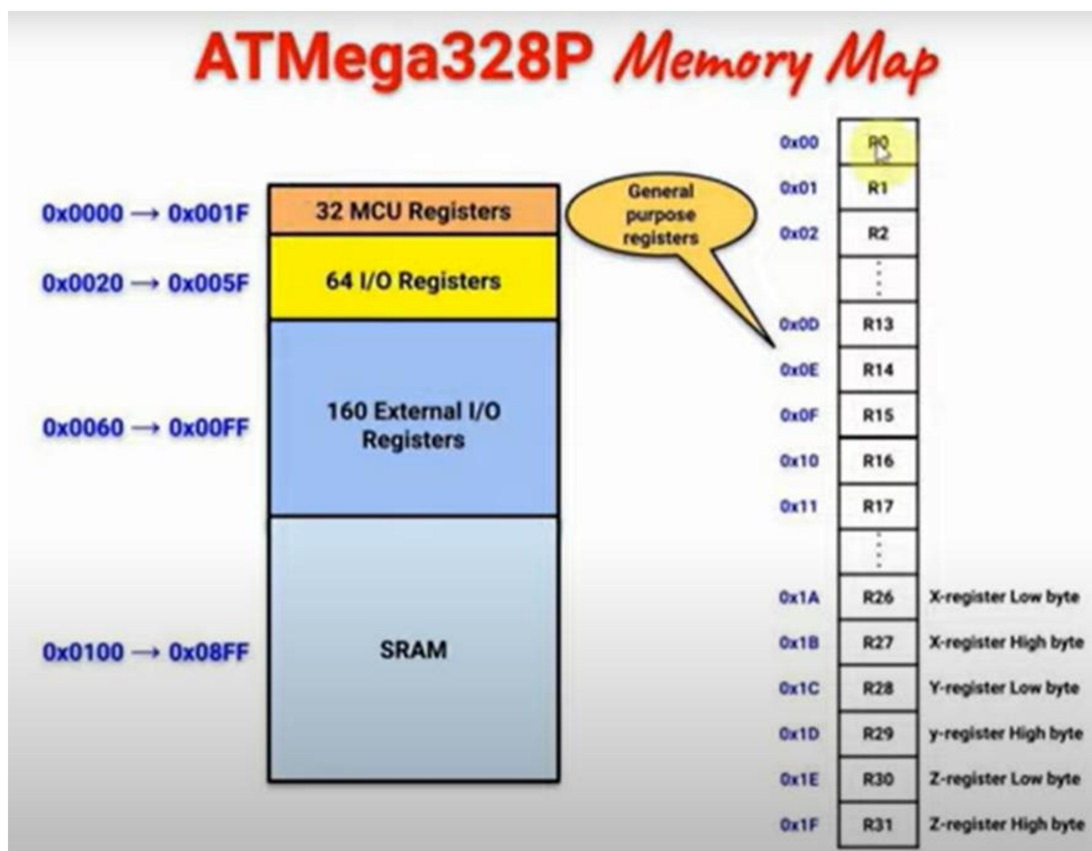
20112012-I Module 2/33

Instruction Set

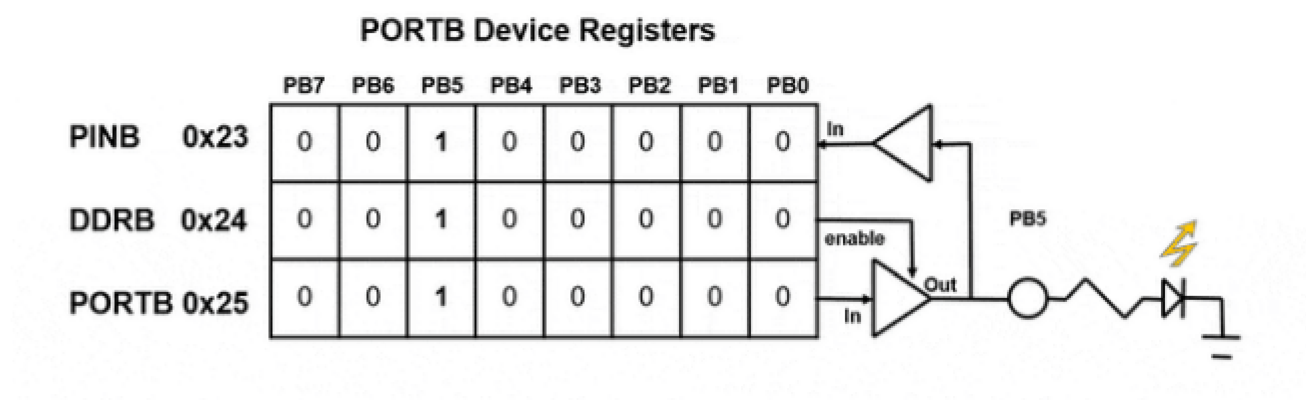
LOGIC	ADD	SUB	AND	OR	MUL
BRANCH	JMP	RCALL	RET	CPI	BREQ
DATA TRANSFER	MOV	IN	OUT	LDI	STS
BIT & BIT TEST	ROL	CBI	CLC	SWAP	POR
CONTROL	BREAK	NOP	SLEEP	WDR	SBIC

Memory Map

- There are 2 bytes for general purpose registers from R0-R31.
- There are 64 I/O R0-R31
- R0-R15 are split by function
- R16-R31 are able to accept direct bit storage
- R0-R25 are 8-bit
- R26-R30 can store 16bits with the H & L bytes stored in each pair as shown.



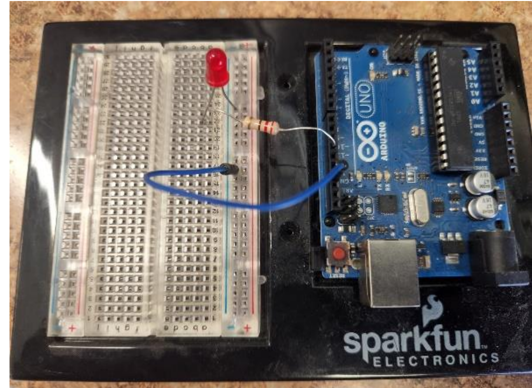
The bits from our program will be made into ***0***'s and ***1***'s that control voltages. We control which pins are output. ***PORTB*** is a block of 8 memory circuits located at address ***0x25*** and it outputs the binary value of the number ***PORTB***. ***PINB*** is the input and sees the voltages at ***PORTB***. ***DDRB*** is the Data Direction Register for PORTB at address ***0x24***.



This is a diagram of the circuits inside the microcontroller chip. ***PB0*** is the first pin of ***PORTB*** and it is pin ***D8*** on the Uno Board. ***PB5*** has the built-in LED circuit connected. In the Arduino IDE we would turn on PB5 as an output with the command ***pinMode(LED_BUILTIN, OUTPUT).***

When output pin ***PB5*** to the Arduino Uno has ***DDRB = 1*** then the output circuit for that pin is enabled. When ***DDRB = 0*** then the circuit allows the pin to connect to register ***PINB*** and the voltage at the pin is an input.

Arduino IDE & Assembly Programming Model. Blink On-Board LED.



To allow an assembly program to compile in the IDE we need two files. They must be named the same and reside in the project folder.

1. Create a new sketch and name it. Let's say LED.ino.
2. In the top right corner of the IDE click the down arrow and choose New Tab.
3. Name the new file LED.S (capital S). This will contain our assembly code. The file MUST be named .S so the code is pre-compiled, and the correct codes will be linked together properly. If you used a .s (lower case) the code will not pre-compile. Pre-compiling means all the sections of the app are identified and located in memory so they can be assembled and linked with any necessary libraries.
4. In the .INO file, add this code. We are using a small sketch to manage our app.

```
1//-------------------------
2 // C Code for Blinking LED
3 //-------------------------
4  extern "C"               //the following will be located externally to this ino file
5 {
6  void start();            //define our function prototypes here.
7  void led(byte);
8 }
9 //--------------------------------------------------
10 void setup()
11 {
12  start();
13 }
14 //--------------------------------------------------
15 void loop()
16 {
17  led(1);
18  delay(200);
19  led(0);
20  delay(200);
21  }
```

Line 4: This tells the IDE compiler we are using code outside/externally to this C code sketch. It defines the two assembly functions we will be using: namely **start()** and **led(byte)**.

Lines 6 & 7: These are the prototypes of our functions and tells the compiler what the signature of the functions will be. In other words, besides its name, it defines the parameters and data types the functions will use. We must define our functions in C/C++ before using them.

Line 10: The void setup() manages the app just like a regular sketch. The difference is, we have encapsulated our functions into another file which happens to be assembly code. We could do the same thing with C++ code to separate our code into logical sections.

Setup() calls the start() function inside the .S file to initialize our app. Notice how we are simply mixing C/C++ code we are accustomed to with a block of assembly instructions.

Line 15: As usual, the loop() function will make our program run in a loop. But in this case, the led() function is in the .S file. The LED function accepts an integer as a parameter. This value will be used to turn the on-board LED ON and OFF.

Line 17: This calls the LED(1) an passes 0x01 (0000 0001) as an argument to turn LED ON.

Line 19: This calls the LED function again but passes a 0 as the argument to turn the LED OFF.

Lines 18 & 20: Here we are using the standard delay() function we are accustomed to in our sketches. This is not the most efficient way to create a delay, however as we will see soon. But it demonstrates the mixing of the two languages and keeps things simple for the moment.

Now, open the .S tab so we can write our assembly code. **Notice comments start with a ";" not "//".**

```
1.  /*
2.  ;_________________________________
3.  ; Assembly Code Used to toggle the
4.    On-Board LED on Pin 13
5.  ;_________________________________
6.  NOTES:
7.  Use actual pin# to toggle LED. In this case PB5. Don't use 6 even though it is the 6th bit!
8.
9.  */
10. ;tells where to start storing code in IO registers
11. #define __SFR_OFFSET 0x00
12. #include "avr/io.h"         ;this is used to upload code to the UNO using AVRDUDE which comes with the IDE
13.
14. .global start               ;declare global variables so it can be accessed fro INO code
15. .global led
16.
17. ;--------------------------------------------------------------------------
18. start:
19.    SBI   DDRB, 5            ;SBI(Set Bit in IO Register) sets PB5 (D13) to output mode so we can write.
20.    RET                      ;return to setup() function
21. ;--------------------------------------------------------------------------
22. led:                        ;the argument sent to led()in the INO code is stored in R24.
23.    CPI   R24, 0x00          ;CPI(Compare I/O bit) to 0. IF R24 == 1, THEN Jump to line 18 to turn it ON, ELSE jump to
    line 17.
24.    BREQ  ledOFF             ;BREQ(Branch If Equal to 0). If R24 is equal to 0 jump to subroutine ledOFF
25.    SBI   PORTB, 5           ;SBI (Set Bit in IO Register) D13 to high
26.    RET                      ;return to loop() function
27. ;--------------------------------------------------------------------------
28. ledOFF:
29.    CBI   PORTB, 5           ;set D13 to low
30.    RET                      ;return to loop() function
31.
32. /*
33.  * Let's look at IF-ELSE logic in Lines 22-26.
34.  * 1. We send a 1 (0x01) the first time through the loop in the INO code to turn the LED ON. This value is stored
    in R24.
35.  * 2. Line 23 compares this value to 0. If R24 != 0, which it isn't, the program jumps to line 25 where it
```

36. * sets PB5 HIGH and turns the led ON.
37. 3. Now we call LED again but pass a 0 (0x00) to turn it off. Line 23 compares what we sent (0 in this case)
38. * and once again compares it to 0 (0x00). Now line 23 evaluates to true so it executes line 24.
39. * 4. When line 24 branches to ledOFF(), line 29 clears bit PB5 (D13) and makes it 0 (0x00) which turns the LED off.

Equivalent C++ code:

```
if(R24 == 1)      //we sent a 1 to the led() function to turn it on

 {

   SBI PORTB,5;    //turn it on

 }

else

 ledOFF;          //turn it off because we sent a 0 to turn it off
```

Keep in mind that these codes are being called/controlled by the loop function inside the INO file. The return (RET) statements direct the execution instructions to jump/go back to where these functions were called just like standard logic in high-level lanaguages!