AVR ARCHITECTURE

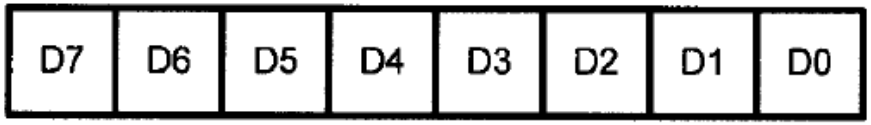
Hoda Roodaki
hroodaki@kntu.ac.ir

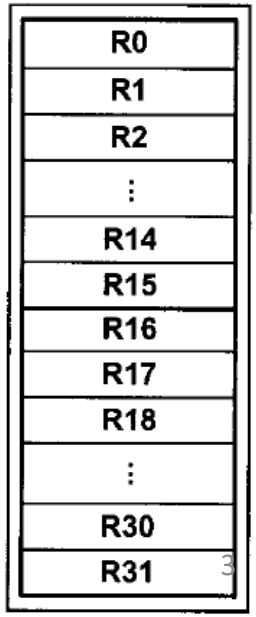
AVR ARCHITECTURE

- CPUs use registers to store data temporarily. To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data.
  - The General Purpose Registers
  - RAM memory inside the AVR and addressing mode.

THE GENERAL PURPOSE REGISTERS IN THE AVR

- CPUs use many registers to store data temporarily.
- To program in Assembly language, we must understand the registers and architecture of a given CPU and the role they play in processing data.
- AVR microcontrollers have many registers for arithmetic and logic operations.
  - Registers are used to store information temporarily.
    - That information could be a byte of data to be processed.
    - An address pointing to the data to be fetched.
  - The vast majority of AVR registers are 8-bit registers
  - In the ATMEG-1 AVR there is only one data type: 8-bit.
    - These range from the MSB (most-significant bit) D7 to the LSB (least-significant bit) DO.
  - With an 8-bit data type, any data larger than 8 bits must into 8-bit chunks before it is processed.





THE GENERAL PURPOSE REGISTERS IN THE AVR

- In AVR there are 32 general purpose registers.
  - They are R0-R31.
  - located in the lowest location of memory address.
  - All of these registers are 8 bits.
  - They can be used by all arithmetic and logic instructions.
  - To understand the use of the general purpose registers, we will show it in the context of two simple instructions: LDI and ADD.

LDI instruction

- LDI instruction
  - Copies 8-bit data into the general purpose registers. It has the following format

```
LDI Rd,K     ;load Rd (destination) with Immediate value K
             ;d must be between 16 and 31
```

  - K is an 8-bit value that can be 0-255 in decimal, or 00-FF in hex.
  - Rd is R16 to R31 (any of the upper 16 general purpose registers).
  - The I in LDI stands for “immediate”.
    - If we see the word "immediate" in any instruction, we are dealing with a value that must be provided right there with the instruction.

LDI instruction

- The following instruction loads the R20 register with a value of 0x25 (25 in hex).

```
LDI R20,0x25            ;load R20 with 0x25 (R20 = 0x25)
```

- The following instruction loads the R31 register with the value 0x87 (87 in hex)

```
LDI R31,0x87            ;load 0x87 into R31   (R31 = 0x87)
```

- We cannot load values into registers R0 to R15 using the LDI instruction.
  - For example; the following instruction is not valid

```
LDI R5,0x99             ;invalid instruction
```

LDI instruction

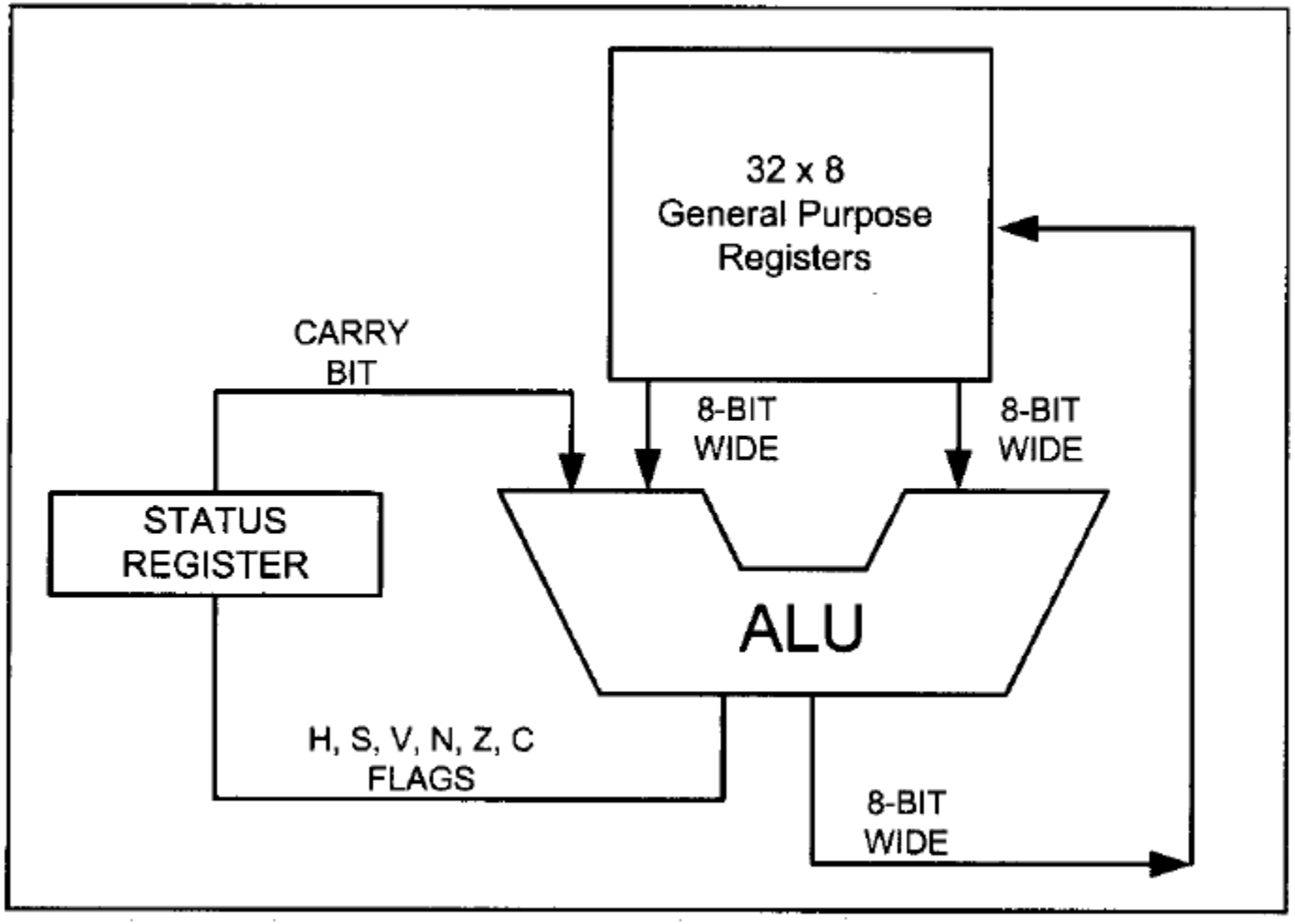
- The LDI loads the right operand into the left operand.
  - The destination comes first.
- To write a comment in Assembly language we use ';'.
  - It is the same as *'//'* in C language, which causes the remainder of the line of code to be ignored.
    - For instance, in the above examples the expressions mentioned after ';' just explain the functionality of the instructions to you, and do not have any effects on the execution of the instructions.

LDI instruction

- When programming the GPRs of the AVR microcontroller with an immediate value, the following points should be noted:
  - If we want to present a number in hex, we put a dollar sign($) or a 0x in front of it.
  - If we put nothing in front of a number, it is in decimal.
    - “LDI R16, 50", R16 is loaded with 50 in decimal.
    - "LDI R16, 0x50'', R16 is loaded with 50 in hex.
  - If values 0 to F are moved into an 8-bit register such as GPRs, the rest of the bits are assumed to be all zeros.
  - "LDI R16, 0x5" the result will be R16 = 0x05;
    - Rl6 = 00000101 in binary.
  - Moving a value larger than 255 (FF in hex) into the GPRs will cause an error.

```
LDI   R17, 0x7F2   ;ILLEGAL $7F2 > 8 bits ($FF)
```

AVR General Purpose Registers and ALU



ADD instruction

- The ADD instruction has the following format:

```
ADD   Rd,Rr  ;ADD Rr to Rd and store the result in Rd
```

  - The ADD instruction tells the CPU to add the value of Rr to Rd and put the result back into the Rd register.
    - To add two numbers such as 0x25 and 0x34, one can do the following:

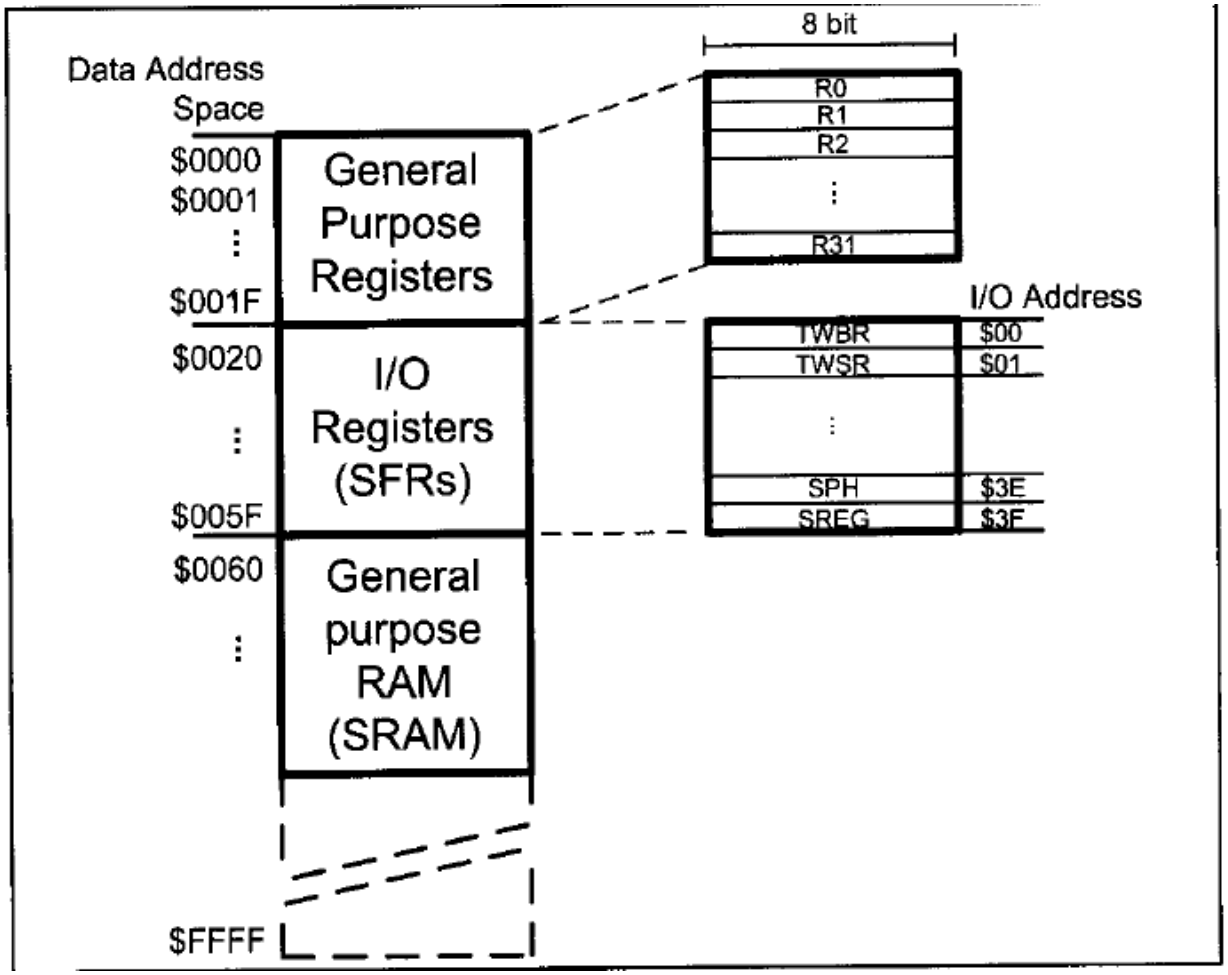
```
LDI R16,0x25      ;load 0x25 into R16
LDI R17,0x34      ;load 0x34 into R17
ADD R16,R17       ;add value R17 to R16 (R16 = R16 + R17)
```

    - Executing the above lines results in R16 = 0x59 (0x25 + 0x34 = 0x59)

THE AVR DATA MEMORY

- In AVR microcontrollers there are two kinds of memory space:
  - Code memory space
    - The program is stored in code memory space
  - Data memory space.
    - The data memory stores data.
    - The data memory is composed of three parts
      - GPRs (general purpose registers)
      - I/O memory
      - Internal data SRAM

The Data Memory for AVRs with No Extended I/O Memory



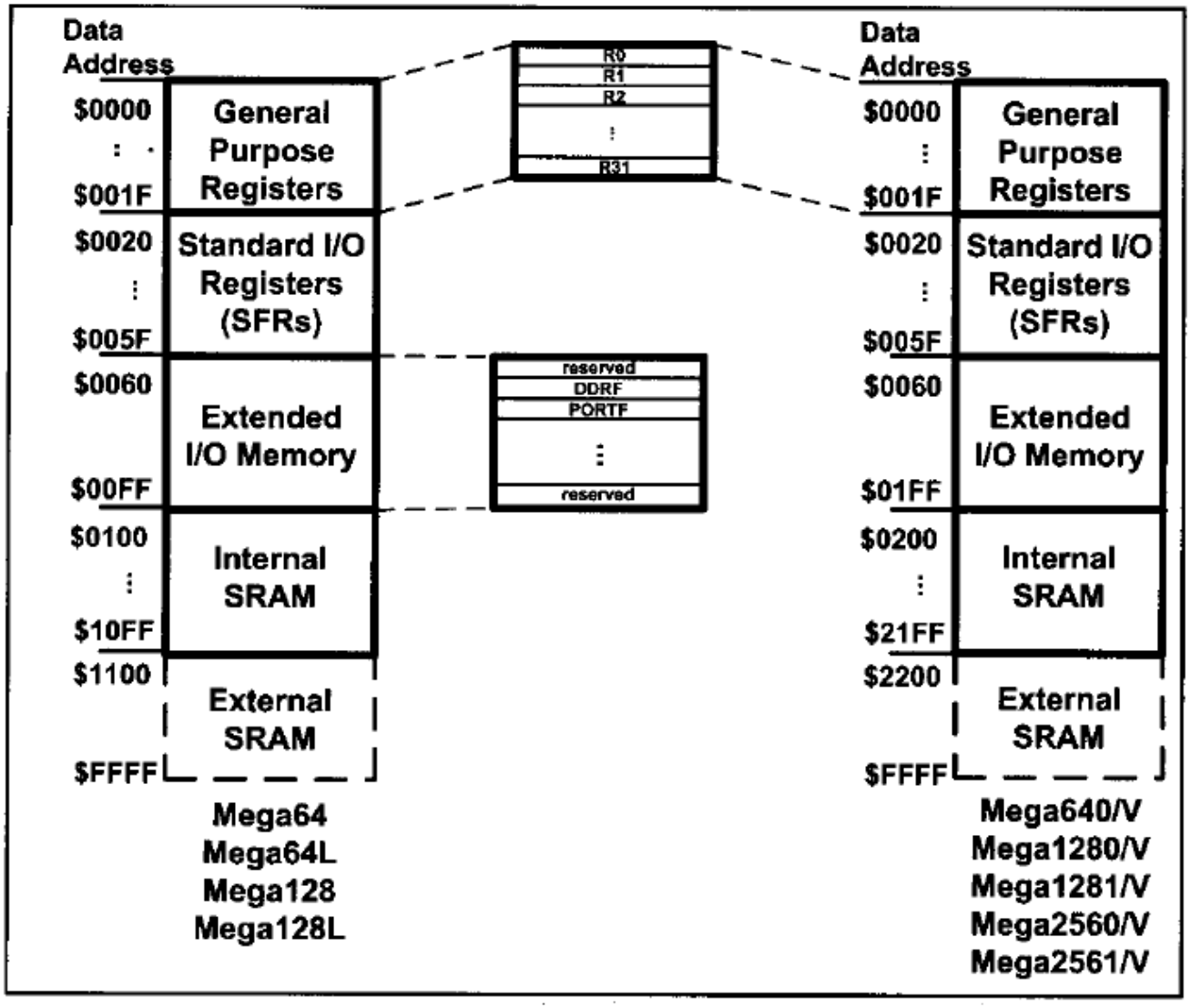
GPRs (general purpose registers)

- The GPRs use 32 bytes of data memory space.
  - They always take the address location $00-$1F in the data memory space, regardless of the AVR chip number.

I/O memory (SfRs)

- The I/O memory is dedicated to specific functions such as
  - Status register
  - Timers
  - Serial communication
  - I/O ports
  - ADC (Analog to Digital Convertor)
- The function of each I/O memory location is fixed by the CPU designer at the time of design because it is used for control of the microcontroller or peripherals.
- The AVR I/O memory is made of 8-bit registers.
  - The number of locations in the data memory set aside for I/O memory depends on the pin numbers and peripheral functions supported by that chip, although the number can vary from chip to chip even among members of the same family.
  - However, all of the AVRs have at least 64 bytes of l/O memory locations.
    - This 64-byte section is called *standard I/O memory.*
    - The I/O registers are called *SFRs (special function registers)* since each one is dedicated to a specific function.
    - In contrast to SFRs, the GPRs do not have any specific function and are used for storing general data.

The Data Memory for the AVRs with Extended I/O Memory



Internal data SRAM

- Internal data SRAM is widely used for storing data and parameters by AVR programmers and C compilers.
  - Each location of the SRAM can be accessed directly by its address.
  - Each location is 8 bits wide and can be used to store any data we want as long as it is 8-bit.
  - The size of SRAM can vary from chip to chip, even among members of the same family.

SRAM vs. EEPROM in AVR chips

- The AVR has an EEPROM memory that is used for storing data.
  - EEPROM does not lose its data when power is off, whereas SRAM does.
    - The EEPROM is used for storing data that should rarely be changed and should not be lost when the power is off (e.g., options and settings).
    - The SRAM is used for storing data and parameters that are changed frequently.
  - The three parts of the data memory (GPRs, SFRs, and the internal SRAM) are made of SRAM.

Data Memory Size for AVR Chips

	Data Memory (Bytes)	=	I/O Registers (Bytes)	+	SRAM (Bytes)	+	General Purpose Register
ATtiny25	224		64		128		32
ATtiny85	608		64		512		32
ATmega8	1120		64		1024		32
ATmega16	1120		64		1024		32
ATmega32	2144		64		2048		32
ATmega128	4352		64+160		4096		32
ATmega2560	8704		64+416		8192		32

USING INSTRUCTIONS WITH THE DATA MEMORY

- The instructions we have used so far worked with the immediate (constant) value of K and the GPRs.
- They also used the GPRs as their destination. We saw simple examples of using LDI and ADD.
- The AVR allows direct access to other locations in the data memory.

LDS instruction (LoaD direct from data Space)

```
LDS    Rd, K   ;load Rd with the contents of location K (0 ≤ d ≤ 31)
               ;K is an address between $0000 to $FFFF
```

- The LDS instruction tells the CPU to load (copy) one byte from an address in the data memory to the GPR.
  - After this instruction is executed, the GPR will have the same value as the location in the data memory.
  - The location in the data memory can be in any part of the data space; it can be one of the I/O registers, a location in the internal SRAM, or a GPR.
    - The "LDS R20, 0x1" instruction will copy the contents of location 1 (in hex) into R20.
      - Location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R1 to R20.

LDS instruction (LoaD direct from data Space)

- The following instruction loads R5 with the contents of location 0x200.
  - 0x200 is located in the internal SRAM:

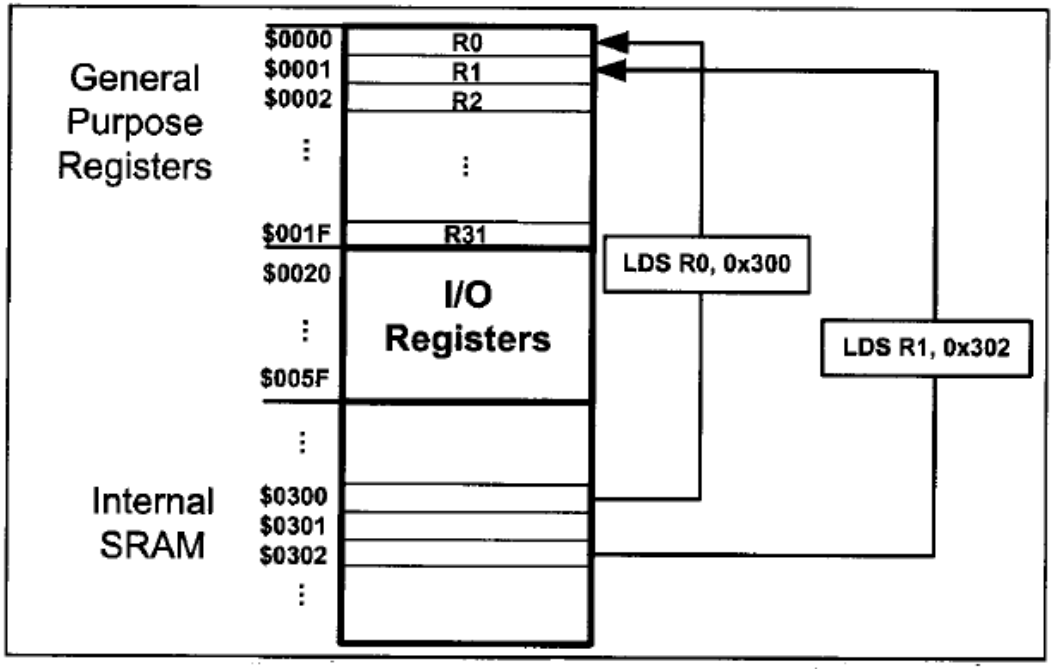
```
LDS R5,0x200 ;load R5 with the contents of location $200
```

  - The following program adds the contents of location 0x300 to location 0x302.
    - First it loads R0 with the contents of location 0x300.
    - R1 with the contents of location 0x302.
    - Adds R0 to R1.

```
LDS   R0, 0x300   ;R0 = the contents of location 0x300
LDS   R1, 0x302   ;R1 = the contents of location 0x302
ADD   R1, R0      ;add R0 to R1
```

LDS instruction (LoaD direct from data Space)

- "LDS R0, 0x300"
- "LDS R1, 0x302"



	R0	R1	Loc $300	Loc $302
Before LDS R0,0x300	?	?	α	β
After LDS R0,0x300	α	?	α	β
After LDS R1,0x302	α	β	α	β
After ADD R0, R1	α + β	β	α	β

STS instruction (STore direct to data Space)

```
STS   K, Rr ;store register into location K
            ;K is an address between $0000 to $FFFF
```

- The STS instruction tells the CPU to store (copy) the contents of the GPR to an address location in the data memory space.
  - After this instruction is executed, the location in the data space will have the same value as the GPR.
  - The location can be in any part of the data memory space; it can be one of the I/O registers, a location in the SRAM, or a GPR.
    - The “STS 0x1, R10" instruction will copy the contents of R10 into location 1.
    - Location 1 of the data memory is in the GPR part, and it is the address of R1. So, the instruction copies R10 to R1.

STS instruction (STore direct to data Space)

- The following instruction stores the contents of R25 to location 0x230.
  - 0x230 is located in the internal SRAM.

```
STS 0x230, R25    ;store R25 to data space location 0x230
```

```
LDI   R16, 0x55   ;R16 = 55 (in hex)
STS   0x38, R16   ;copy R16 to Port B (PORTB = 0x55)
STS   0x35, R16   ;copy R16 to Port C (PORTC = 0x55)
STS   0x32, R16   ;copy R16 to Port D (PORTD = 0x55)
```

STS instruction (STore direct to data Space)

- The following program will put 0x99 into locations 0x200-0x203 of the SRAM region in the data memory

```
LDI   R20, 0x99   ;R20 = 0x99
STS   0x200, R20  ;store R20 in loc 0x200
STS   0x201, R20  ;store R20 in loc 0x201
STS   0x202, R20
STS   0x203, R20 ;see the Mem. contents->
```

STS instruction (STore direct to data Space)

- The following program adds the contents of location 0x220 to location 0x221, and stores the result in location 0x221.

```
LDS   R30, 0x220  ;load R30 with the contents of location 0x220
LDS   R31, 0x221  ;load R31 with the contents of location 0x221
ADD   R31, R30    ;add R30 to R31
STS   0x221, R31  ;store R31 to data space location 0x221
```

Example 1

- State the contents of RAM locations $212 to $216 after the following program is executed:

```
LDI   R16, 0x99   ;load R16 with value 0x99
STS   0x212, R16
LDI   R16, 0x85   ;load R16 with value 0x85
STS   0x213, R16
LDI   R16, 0x3F   ;load R16 with value 0x3F
STS   0x214, R16
LDI   R16, 0x63   ;load R16 with value 0x63
STS   0x215, R16
LDI   R16, 0x12   ;load R16 with value 0x12
STS   0x216, R16
```

Example 1

- Solution:
  - After the execution of STS 0x212, R16 data memory location $212 has value 0x99.
  - After the execution of STS 0x213, R16 data memory location $213 has value 0x85.
  - After the execution of STS 0x214, R16 data memory location $214 has value 0x3F.
  - After the execution of STS 0x215 , R16 data memory location $215 has value 0x63.

Address	Data
$212	0x99
$213	0x85
$214	0x3F
$215	0x63
$216	0x12

Example 2

- State the contents of R20, R21, and data memory location 0x120 after the following program:

```
LDI   R20, 5       ;load R20 with 5
LDI   R21, 2       ;load R21 with 2
ADD   R20, R21     ;add R21 to R20
ADD   R20, R21     ;add R21 to R20
STS   0x120, R20   ;store in location 0x120 the contents of R20
```

Example 2

- The program loads R20 with value 5.
- Then it loads R21 with value 2.
- Then it adds the R21 register to R20 twice.
- At the end, it stores the result in location 0x120 of data memory.

Location	Data
R20	5
R21	
0x120	

After LDI R20, 5

Location	Data
R20	5
R21	2
0x120	

After LDI R21, 2

Location	Data
R20	7
R21	2
0x120	

After ADD R20, R21

Location	Data
R20	9
R21	2
0x120	

After ADD R20, R21

Location	Data
R20	9
R21	2
0x120	9

After STS 0x120, R20

I/O Registers of the ATmega32 and Their Data Memory Address Locations

Address		Name
Mem.	I/O	
$20	$00	TWBR
$21	$01	TWSR
$22	$02	TWAR
$23	$03	TWDR
$24	$04	ADCL
$25	$05	ADCH
$26	$06	ADCSRA
$27	$07	ADMUX
$28	$08	ACSR
$29	$09	UBRRL
$2A	$0A	UCSRB
$2B	$0B	UCSRA
$2C	$0C	UDR
$2D	$0D	SPCR
$2E	$0E	SPSR
$2F	$0F	SPDR
$30	$10	PIND
$31	$11	DDRD
$32	$12	PORTD
$33	$13	PINC
$34	$14	DDRC
$35	$15	PORTC

Address		Name
Mem.	I/O	
$36	$16	PINB
$37	$17	DDRB
$38	$18	PORTB
$39	$19	PINA
$3A	$1A	DDRA
$3B	$1B	PORTA
$3C	$1C	EECR
$3D	$1D	EEDR
$3E	$1E	EEARL
$3F	$1F	EEARH
$40	$20	UBRRC
		UBRRH
$41	$21	WDTCR
$42	$22	ASSR
$43	$23	OCR2
$44	$24	TCNT2
$45	$25	TCCR2
$46	$26	ICR1L
$47	$27	ICR1H
$48	$28	OCR1BL
$49	$29	OCR1BH
$4A	$2A	OCR1AL

Address		Name
Mem.	I/O	
$4B	$2B	OCR1AH
$4C	$2C	TCNT1L
$4D	$2D	TCNT1H
$4E	$2E	TCCR1B
$4F	$2F	TCCR1A
$50	$30	SFIOR
$51	$31	OCDR
		OSCCAL
$52	$32	TCNT0
$53	$33	TCCR0
$54	$34	MCUCSR
$55	$35	MCUCR
$56	$36	TWCR
$57	$37	SPMCR
$58	$38	TIFR
$59	$39	TIMSK
$5A	$3A	GIFR
$5B	$3B	GICR
$5C	$3C	OCR0
$5D	$3D	SPL
$5E	$3E	SPH
$5F	$3F	SREG

Note: Although memory address $20-$5F is set aside for I /O registers (SFR) we can access them as I /O locations with addresses starting at $00.

IN instruction (IN from I/O location)

- In the IN instruction, the I/O registers are referred to by their I/O addresses.
  - The “IN R20, 0x16" instruction will copy the contents of location $16 of the I/O memory (whose data memory address is 0x36) into R20.

IN instruction (IN from I/O location)

- The following instruction loads R19 with the contents of location 0x10 of the I/O memory:

```
IN R19,0x10          ;load R19 with location $10 (R19 = PIND)
```

- To work with the I/O registers more easily, we can use their names instead of their I/O addresses.
  - The following instruction loads R19 with the contents of PIND:

```
IN R19,PIND          ;load R19 with PIND
```

IN instruction (IN from I/O location)

- Notice that to be able to use the names of the I/O addresses instead of the I/O addresses we should include the proper header files (M32DEF.INC).
- The following program adds the contents of PIND to PINB, and stores the result in location 0x300 of the data memory:

```
IN    R1,PIND      ;load R1 with PIND
IN    R2,PINB      ;load R2 with PINB
ADD   R1, R2       ;R1 = R1 + R2
STS   0x300, R1    ;store R1 to data space location $300
```

IN vs. LDS

- We can use the LDS instruction to copy the contents of a memory location to a GPR.
  - This means that we can load an I/O register into a GPR, using the LDS instruction.
  - what is the advantage of using the IN instruction for reading the contents of I/O registers over using the LDS instruction?
    - The IN instruction has the following advantages:
      - The CPU executes the IN instruction faster than LDS.
        - » The IN instruction lasts 1 machine cycle, whereas LDS lasts 2 machine cycles.
        - » The IN is a 2-byte instruction, whereas LDS is a 4-byte instruction. This means that the IN instruction occupies less code memory.
        - » When we use the IN instruction, we can use the names of the I/O registers instead of their addresses.
        - » The IN instruction is available in all of the AVRs, whereas LDS is not implemented in some of the AVRs.

OUT instruction (OUT to I/O location)

```
OUT A, Rr ;store register to I/O location (0 ≤ r ≤ 31),(0 ≤A ≤ 63)
```

- The OUT instruction tells the CPU to store the GPR to the I/O register.
- After the instruction is executed, the I/O register will have the same value as the GPR.
  - The "out PORTD, R10" instruction will copy the contents of R10 into PORTD (location 12 of the I/O memory).
  - Notice that in the OUT instruction, the I/O registers are referred to by their I/O addresses (like the IN instruction).

OUT instruction (OUT to I/O location)

- The following program copies 0xE6 to the SPL register:

```
LDI   R20,0xE6    ;load R20 with 0xE6
OUT   SPL, R20    ;out R20 to SPL
```

- We must remember that we cannot copy an immediate value to an I/O register nor to an SRAM location.
- The following program copies PIND to PORTA:

```
IN    R0, PIND    ;load R20 with the contents of I/O reg PIND
OUT   PORTA, R0   ;out R20 to PORTA
```

Example

- Write a program to get data from the PINB and send it to the I/O register of PORT C continuously.

```
AGAIN:IN    R16, PINB    ;bring data from PortB into R16
      OUT   PORTC,R16    ;send it to Port C
      JMP   AGAIN        ;keep doing it forever
```

MOV instruction

- The MOV instruction is used to copy data among the GPR registers of R0-R31.

```
MOV   Rd,Rr          ;Rd = Rr (copy Rr to Rd)
                     ;Rd and Rr can be any of the GPRs
```

- The following instruction copies the contents of R20 to R10:

```
MOV   R10,R20        ;R10 = R20
```

  - For instance, if R20 contains 60, after execution of the above instruction both R20 and R10 will contain 60.

More ALU instructions involving the GPRs

- Add Instruction
  - The following program adds 0x19 to the contents of location 0x220 and stores the result in location 0x221.

```
LDI    R20, 0x19    ;load R20 with 0x19
LDS    R21, 0x220   ;load R21 with the contents of location 0x220
ADD    R21, R20     ;R21 = R21 + R20
STS    0x221, R21   ;store R21 to location 0x221
```

More ALU instructions involving the GPRs

- *INC instruction*
  - The INC instruction increments the contents of Rd by 1.

```
INC    Rd     ;increment the contents of Rd by one (0 ≤ d ≤ 31)
```

- The following instruction adds 1 to the contents of R2

```
INC    R2              ;R2 = R2 + 1
```

- The following program increments the contents of data memory location 0x430 by 1

```
LDS    R20, 0x430  ;R20 = contents of location 0x430
INC    R20         ;R20 = R20 + 1
STS    0x430, R20  ;store R20 to location 0x430
```

More ALU instructions involving the GPRs

- *SUB instruction*
  - The SUB instruction has the following format:

```
SUB   Rd,Rr          ;Rd = Rd - Rr
```

  - The SUB instruction tells the CPU to subtract the value of Rr from Rd and put the result back into the Rd register.
    - To subtract 0x25 from 0x34, one can do the following:

```
LDI    R20, 0x34    ;R20 = 0x34
LDI    R21, 0x25    ;R20 = 0x25
SUB    R20, R21     ;R20 = R20 - R21
```

    - The following program decrements the contents of R10, by 1:

```
LDI    R16, 0x1     ;load 1 to R16
SUB    R10, R16     ;R10 = R10 - R16
```

More ALU instructions involving the GPRs

- *DEC instruction*
  - The DEC instruction has the following format:

```
DEC   Rd            ;Rd = Rd - 1
```

  - The DEC instruction decrements (subtracts 1 from) the contents of Rd and puts the result back into the Rd register.
    - The following instruction subtracts 1 from the contents of R10: `DEC  R10  ;R10 = R10 - 1`
    - In the following program, we put the value 3 into R30. Then the value in R30 is decremented.

```
LDI   R30, 3      ;R30 = 3
DEC   R30         ;R30 has 2
DEC   R30         ;R30 has 1
DEC   R30         ;R30 has 0
```

More ALU instructions involving the GPRs

- *COM instruction*
  - The "COM Rd" instruction complements (inverts) the contents of Rd and places the result back into the Rd register.
    - In the following program, we put 0x55 into R16 and then send it to the SFR location of PORTB. Then the content of R16 is complemented, which becomes AA in hex.
      - The 01010101 (0x55) is inverted and becomes 10101010 (0xAA).

```
LDI   R16,0x55     ;R16 = 0x55
OUT   PORTB, R16   ;copy R16 to Port B SFR (PB = 0x55)
COM   R16          ;complement R16          (R16 = 0xAA)
OUT   PORTB, R16   ;copy R16 to Port B SFR (PB = 0xAA)
```

More ALU instructions involving the GPRs

- Example
  - Write a simple program to toggle the I/O register of PORT B continuously forever.

```
        LDI   R20, 0x55   ;R20 = 0x55
        OUT   PORTB, R20  ;move R20 to Port B SFR (PB = 0x55)
L1:     COM   R20         ;complement R20
        OUT   PORTB, R20  ;move R20 to Port B SFR
        JMP   L1          ;repeat forever (see Chapter 3 for JMP)
```

More ALU instructions involving the GPRs

- ALU Instructions Using, Two GPRs

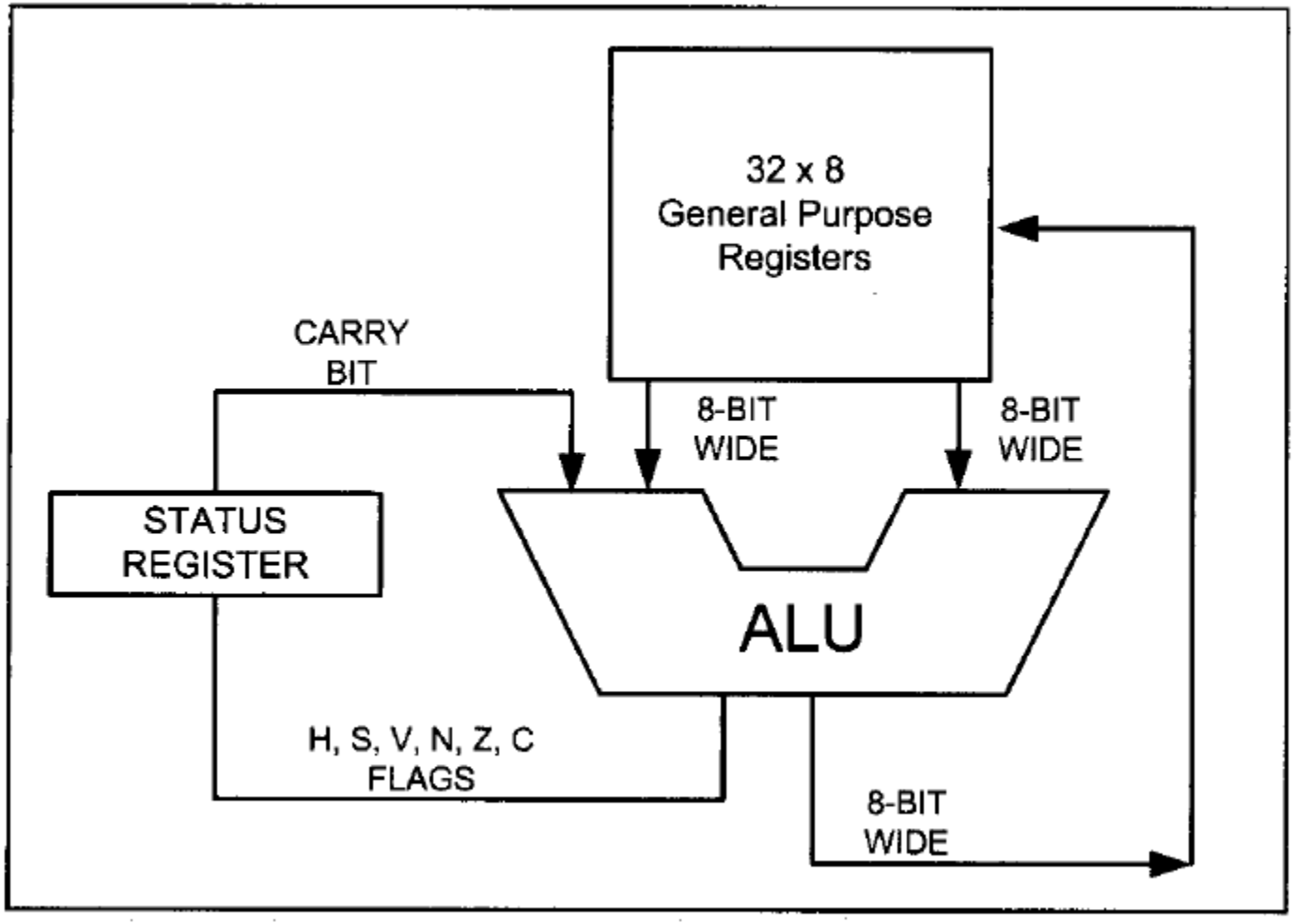
Instruction		
ADD	Rd, Rr	ADD Rd and Rr
ADC	Rd, Rr	ADD Rd and Rr with Carry
AND	Rd, Rr	AND Rd with Rr
EOR	Rd, Rr	Exclusive OR Rd with Rr
OR	Rd, Rr	OR Rd with Rr
SBC	Rd, Rr	Subtract Rr from Rd with carry
SUB	Rd, Rr	Subtract Rr from Rd without carry

More ALU instructions involving the GPRs

- Some Instructions Using a GPR as Operand

Instruction		
CLR	Rd	Clear Register Rd
INC	Rd	Increment Rd
DEC	Rd	Decrement Rd
COM	Rd	One's Complement Rd
NEG	Rd	Negative (two's complement) Rd
ROL	Rd	Rotate left Rd through carry
ROR	Rd	Rotate right Rd through carry
LSL	Rd	Logical Shift Left Rd
LSR	Rd	Logical Shift Right Rd
ASR	Rd	Arithmetic Shift Right Rd
SWAP	Rd	Swap nibbles in Rd

AVR General Purpose Registers and ALU



AVR DATA FORMAT AND DIRECTIVES

- AVR data type
  - The AVR microcontroller has only one data type.
    - 8 bits, and the size of each register is also 8 bits.
    - It is the job of the programmer to break down data larger than 8 bits (00 to 0xFF, or 0 to 255 in decimal) to be processed by the CPU.
    - The data types used by the AVR can be positive or negative.

Data format representation

- There are four ways to represent a byte of data in the AVR assembler.
  - Hex
  - Binary
  - Decimal
  - ASCII formats

Data format representation

- *Hex numbers*
  - There are two ways to show hex numbers:
  - Put 0x (or 0X) in front of the number like this.
  - Put $ in front of the number.
- *Binary numbers*
  - There is only one way to represent binary numbers in an AVR assembler as follows:
    - LDI Rl6,0bl0011001
    - The upper case B will also work.

Data format representation

- *Decimal numbers*
  - To indicate decimal numbers in an AVR assembler we simply use the decimal (e.g., 12) and nothing before or after it. Here are some examples of how to use it:

```
LDI  R17, 12  ;R17 = 00001100 or 0C in hex
SUBI R17, 2   ;R17 = 12 - 2 = 10 where 10 is equal to 0x0A
```

- *ASCII characters*
  - To represent ASCII data in an AVR assembler we use single quotes as follows

```
LDI R23,'2'  ;R23 = 00110010 or 32 in hex
```

Assembler directives

- While instructions tell the CPU what to do, directives (also called *pseudo-instructions)* give directions to the assembler.
  - The LDI and ADD instructions are commands to the CPU.
  - .EQU, .DEVICE, and .ORG are directives to the assembler.
  - The directives help us develop our program easier and make our program legible (more readable) .

Assembler directives

- *.EQU (equate)*
  - This is used to define a constant value or a fixed address.
  - The .EQU directive does not set aside storage for a data item, but associates a constant number with a data or an address label so that when the label appears in the program, its constant will be substituted for the label.
    - The following uses .EQU for the counter constant, and then the constant is used to load the R21 register:

```
.EQU   COUNT = 0x25
       ...   ....
       LDI   R21, COUNT          ;R21 = 0x25
```

    - When executing the above instruction "LDI R21, COUNT", the register R21 will be loaded with the value 25H.

Assembler directives

- What is the advantage of using .EQU?
  - Assume that a constant (a fixed value) is used throughout the program, and the programmer wants to change its value everywhere. By the use of .EQU, the programmer can change it once and the assembler will change all of its occurrences throughout the program.
  - This allows the programmer to avoid searching the entire program trying to find every occurrence.
    - We mentioned earlier that we can use the names of the I/O registers instead of their addresses (e.g., we can write "OUT PORTA,R20" instead of "OUT 0x1B,R20"). This is done with the help of the .EQU directive.
    - In include files such as M32DEF.INC the I/O register names are associated with their addresses using the .EQU directive.
      - For example, in M32DEF.INC the following pseudo-instruction exists, which associates 0x1B (the address of PORTB) with the PORTB

```
.EQU   PORTB = 0x1B
```

Assembler directives

- *.SET*
  - This directive is used to define a constant value or a fixed address.
  - In this regard, the .SET and .EQU directives are identical.
    - The only difference is that the value assigned by the .SET directive may be reassigned later.

```
.SET F = 0x114    ; set F to point to an SRAM location
LDS r0, F                    ; load location into r0
.SET F = F + 1    ; increment (redefine) F. This would be illegal if using .EQU
LDS r1, F                    ; load next location into r1
```

Assembler directives

- *.ORG (origin)*
  - The .ORG directive is used to indicate the beginning of the address. It can be used for both code and data.

```
.DSEG ; Start data segment
    .ORG 0x120; Set SRAM address to hex 120
.CSEG
    .ORG 0x10 ; Set Program Counter to hex 10
```

- *.INCLUDE directive*
  - The .include directive tells the AVR assembler to add the contents of a file to our program
    - like the #include directive in C language.

Rules for labels in Assembly language

- By choosing label names that are meaningful, a programmer can make a program much easier to read and maintain.
- There are several rules that names must follow.
  - Each label name must be unique.
  - The names used for labels in Assembly language programming consist of alphabetic letters in both uppercase and lowercase, the digits 0 through 9, and the special characters question mark (?), period(.), at(@), underline(_), and dollar sign($).
  - The first character of the label must be an alphabetic character. In other words, it cannot be a number.
  - Every assembler has some reserved words that must not be used as labels in the program. Foremost among the reserved words are the mnemonics for the instructions.
    - For example, "LDI" and "ADD" are reserved because they are instruction mnemonics.