AVR Microcontroller

Microprocessor Course

Aban 1393

AVR Microcontroller

A brief history of the AVR microcontroller

The basic architecture of AVR was designed by two students of Norwegian Institute of Technology (NTH), Alf-Egil Bogen and Vegard Wollan, and then was bought and developed by Atmel in 1996.

You may ask what AVR stands for; AVR can have different meanings for different people! Atmel says that it is nothing more than a product name, but it might stand for Advanced Virtual RISC, or Alf and Vegard RISC (the names of the AVR designers).

AVR Microcontroller

The AVR is an 8-bit RISC single-chip microcontroller with Harvard architecture that comes with some standard features such as

- on-chip program (code) ROM,
- Data RAM,
- Data EEPROM, timers and
- I/O ports

Some AVRs have some additional features like ADC, PWM, and different kinds of serial interface such as USART, SPI, I2C (TWI), CAN, USB, and so on.

AVR Microcontroller

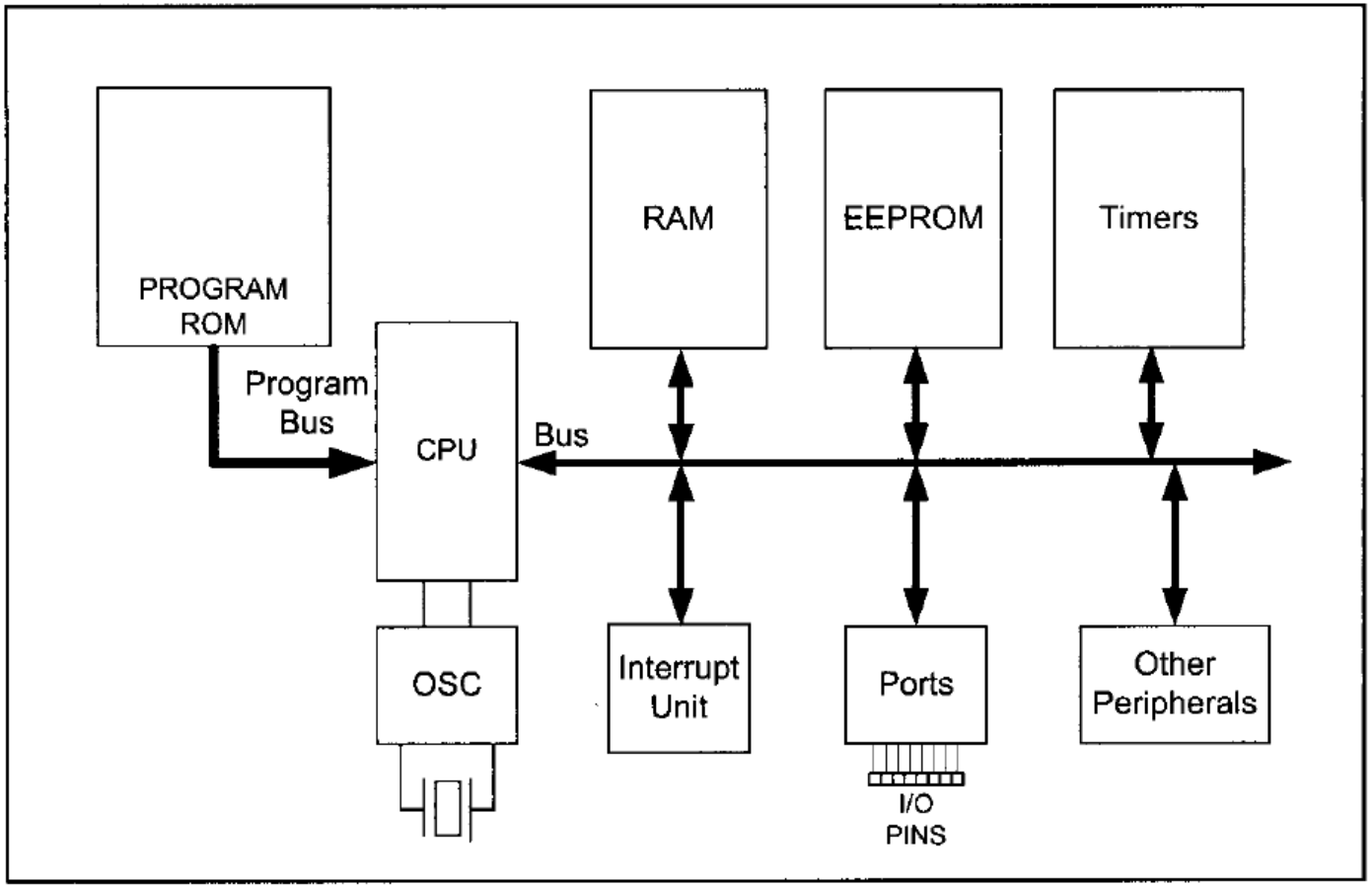


Figure 1-2. Simplified View of an AVR Microcontroller

AVR Microcontroller

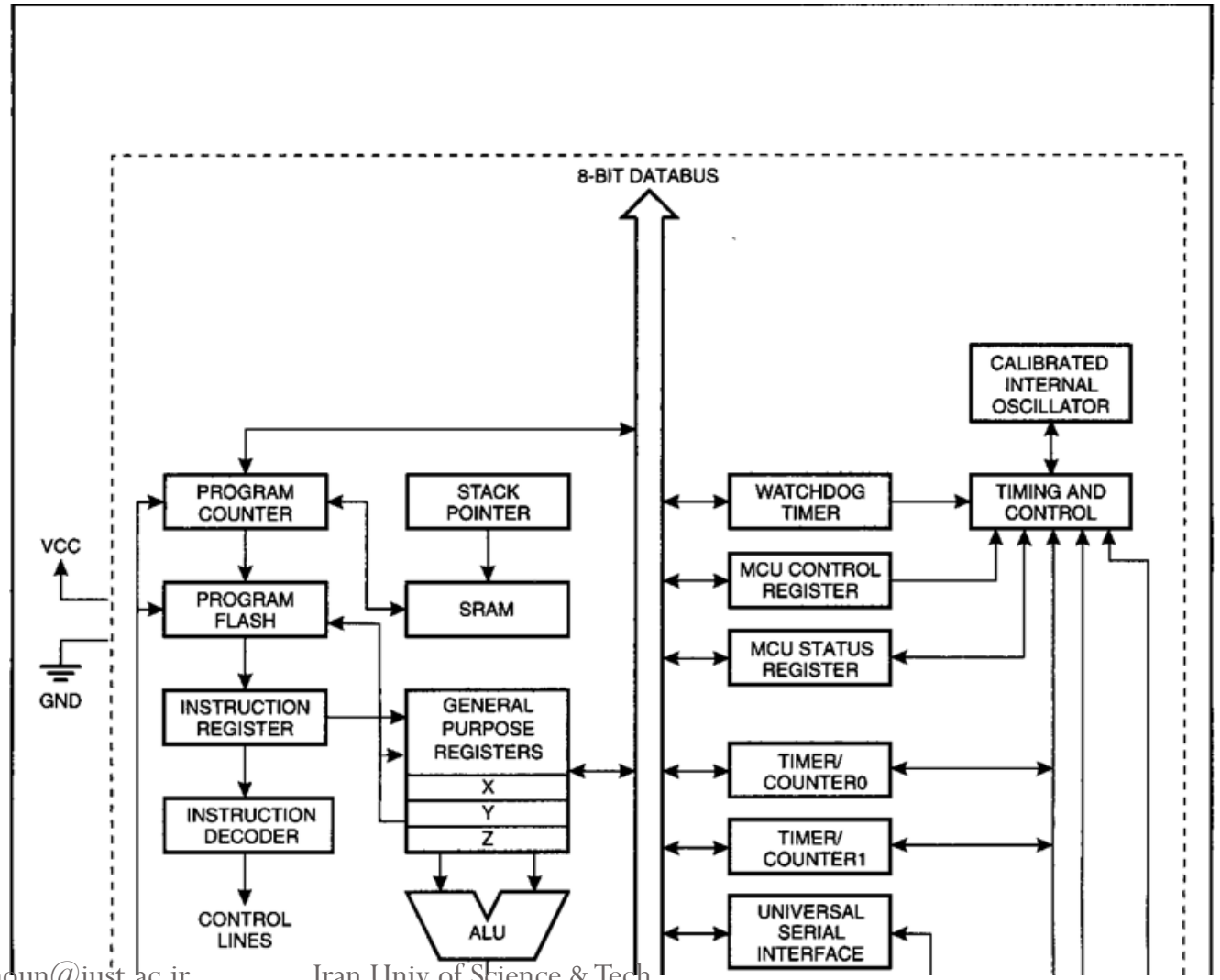
- ***AVR microcontroller program ROM***

Although the AVR has 8M (megabytes) of program (code) ROM space, not all family members come with that much ROM installed.

The program ROM size can vary from 1K to 256K at the time of this writing, depending on the family member.

The AVR was one of the first microcontrollers to use on-chip Flash memory for program storage. The Flash memory is ideal for fast development because

AVR Microcontroller AT tiny25 Block Diagram



AVR Microcontroller AT tiny25 Block Diagram

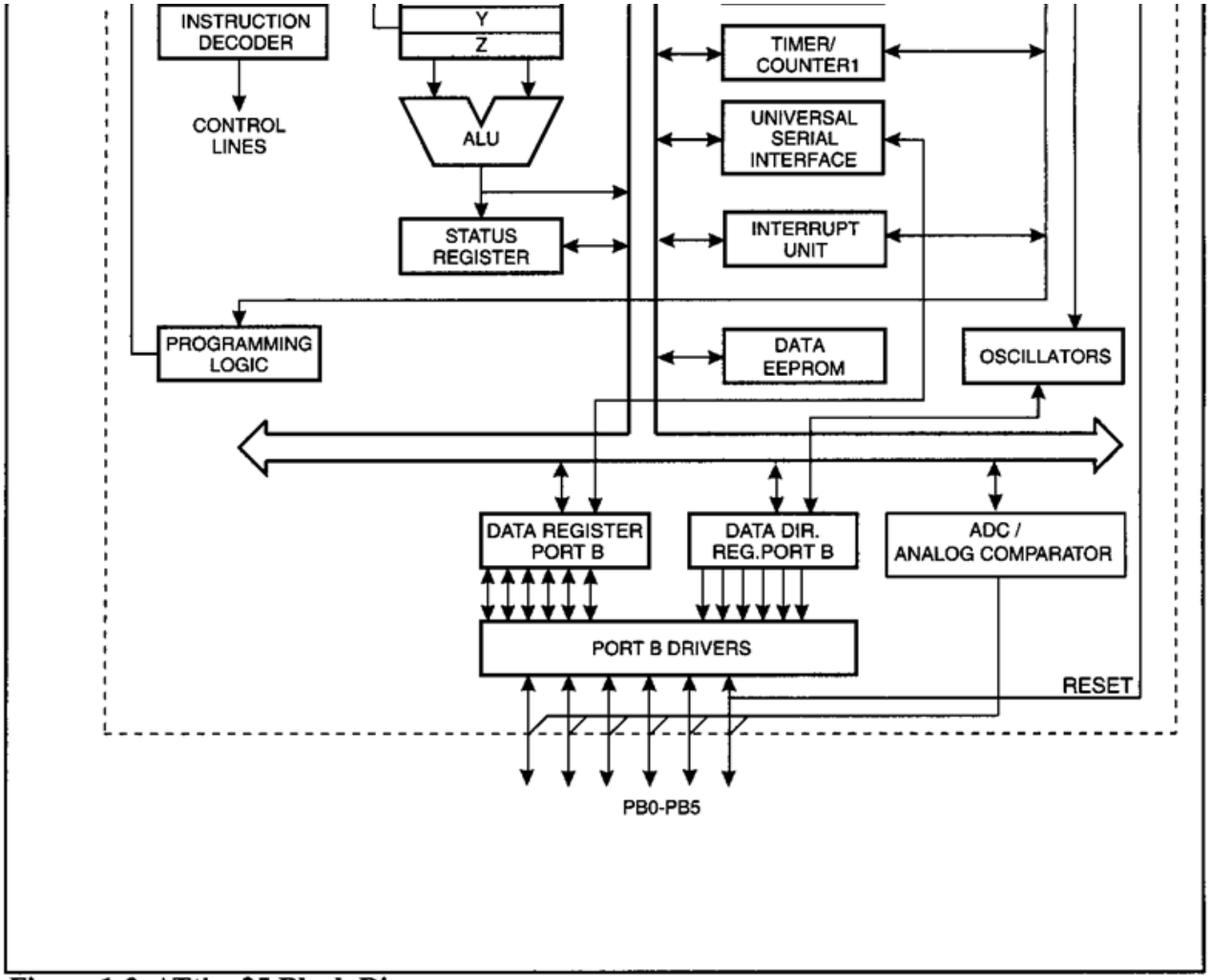
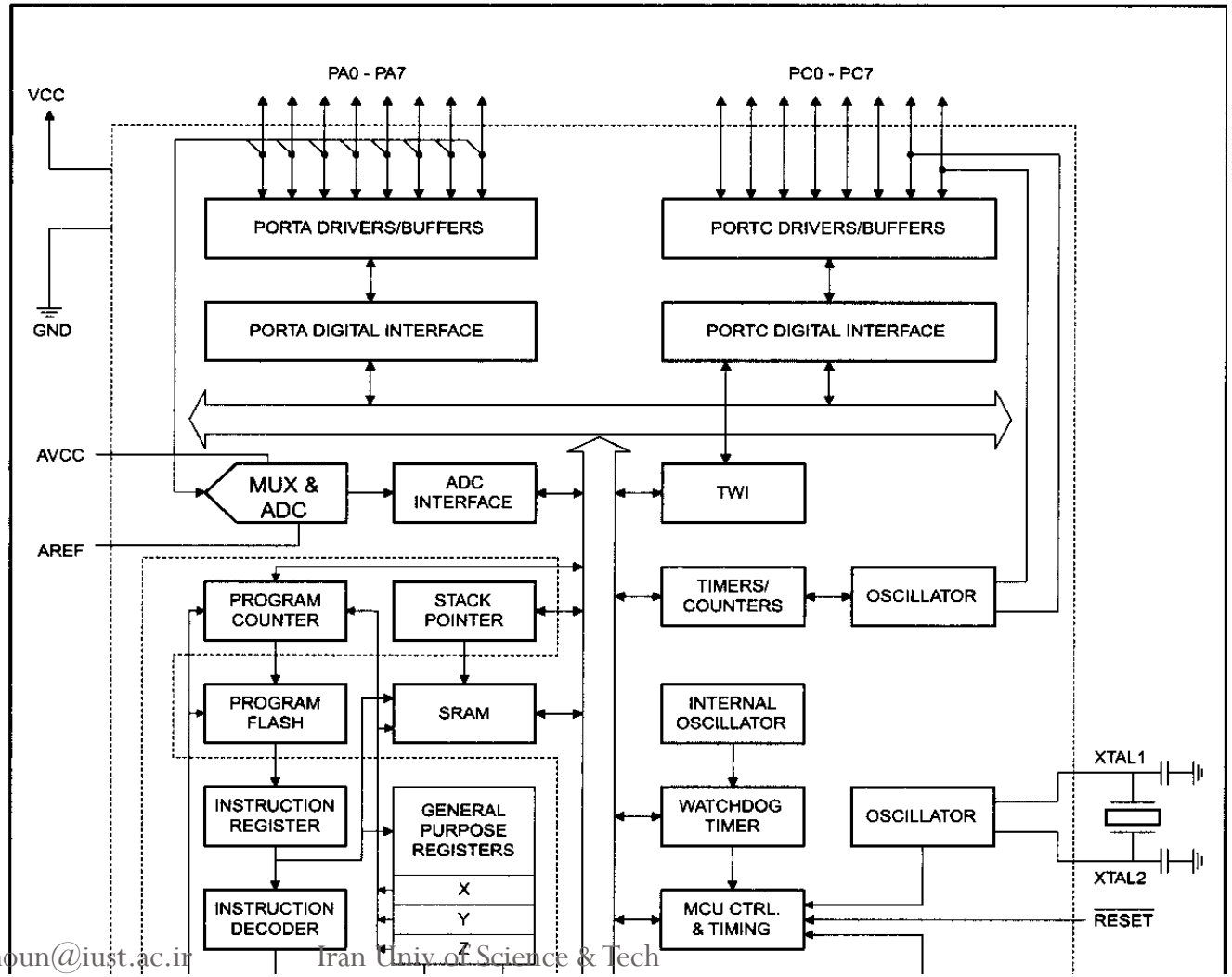


Figure 1-3. ATtiny25 Block Diagram

AVR Microcontroller ATmega32 Block Diagram



AVR Microcontroller ATmega32 Block Diagram

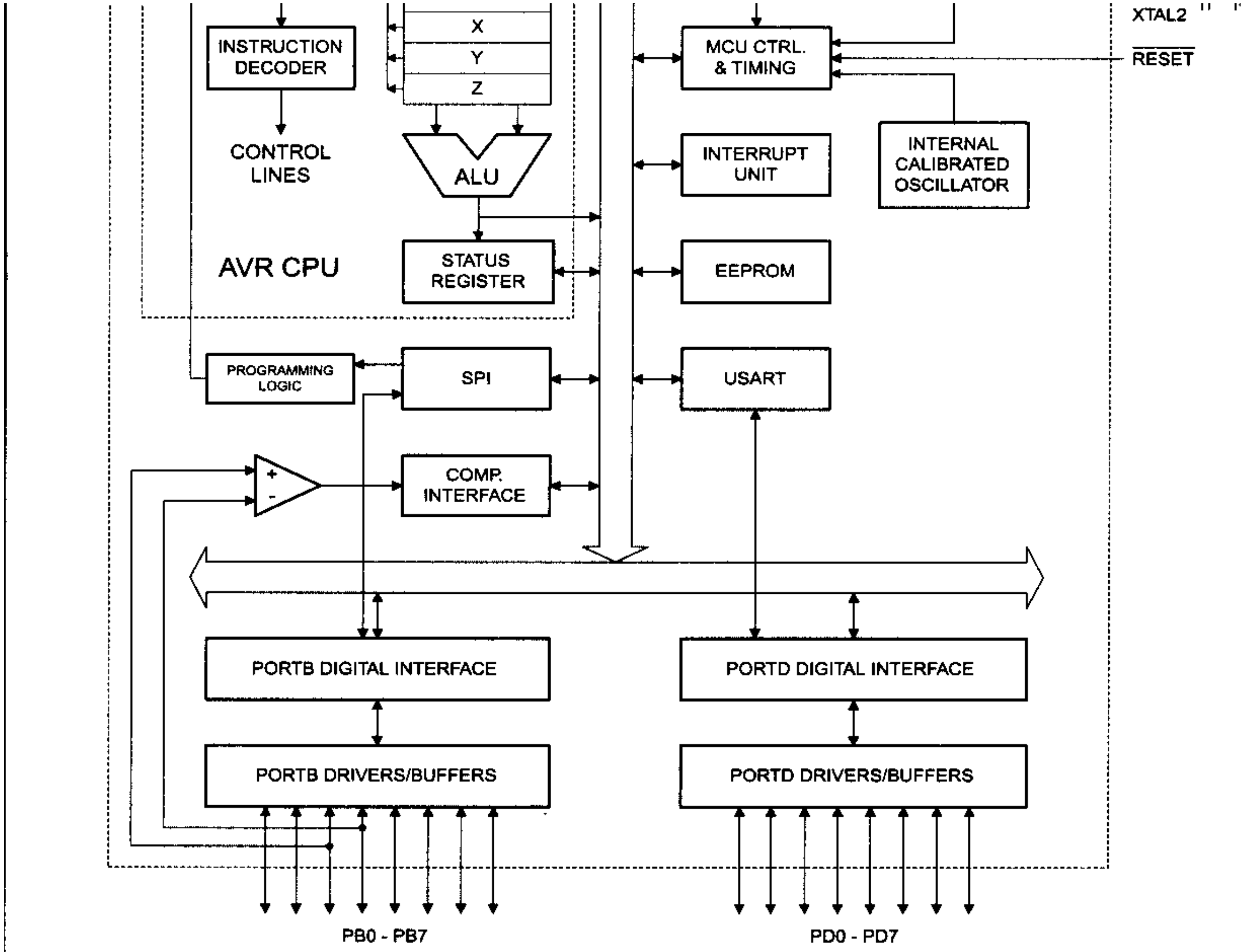


Figure 1-4. ATmega32 Block Diagram

AVR Microcontroller ATmega32 Block Diagram

AVR microcontroller data RAM and EEPROM

While ROM is used to store program (code), the RAM space is for data storage. The AVR has a maximum of 64K bytes of data RAM space. Not all of the family members come with that much RAM. As we will see, the data RAM space has three components:

- general-purpose registers,
- I/O memory,
- internal SRAM.

There are 32 general-purpose registers in all of the AVRs, but the SRAM's size and the I/O memory's size varies from chip to chip. On the Atmel website, whenever the size of RAM is mentioned the internal SRAM size is meant. The internal SRAM space is used for a read/write scratch pad. In AVR, we also have a small amount of EEPROM to store critical data that does not need to be changed very often.

AVR Microcontroller ATmega32 Block Diagram

AVR microcontroller I/O pins

The AVR can have from 3 to 86 pins for I/O. The number of I/O pins depends on the number of pins in the package itself. The number of pins for the AVR package goes from 8 to 100 at this time. In the case of the 8-pin AT90S2323, we have 3 pins for I/O, while in the case of the 100-pinATmegal280, we can use up to 86 pins for I/O.

AVR microcontroller peripherals

Most of the AVRs come with ADC (analog-to-digital converter), timers, and USART (Universal Synchronous Asynchronous Receiver Transmitter) as standard peripherals. As you will see, the ADC is 10-bit and the number of ADC channels in AVR chips varies and can be up to 16, depending on the number of pins in the package.

The AVR can have up to 6 timers besides the watchdog timer. The USART peripheral allows us to connect the AVR-based system to serial ports such as the COM port of the x86 IBM PC. Most of the AVR family members come with the PC and SPI buses and some of them have USB or CAN bus as well.

AVR Microcontroller ATmega32 Block Diagram

Table 1-2: Some Members of the Classic Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
AT90S2313	2K	128	128	15	0	2	SOIC20, PDIP20
AT90S2323	2K	128	128	3	0	1	SOIC8, PDIP8
AT90S4433	4K	128	256	20	6	2	TQFP32, PDIP28

Notes:

1. All ROM, RAM, and EEPROM memories are in bytes.
2. Data RAM (general-purpose RAM) is the amount of RAM available for data manipulation (scratch pad) in addition to the register space.

AVR Microcontroller

AVRs are generally classified into four groups:

- Mega
- Tiny
- Special purpose
- Classic

In this course we cover the Mega family because these microcontrollers are widely used. Also we will focus on ATmega32 since it is powerful, widely available, and comes in DIP packages, which makes it ideal for educational purposes.

AVR Microcontroller

Classic AVR (ATSOSxxxx)

This is the original AVR chip, which has been replaced by newer AVR chips. Table 1-2 shows some members of the Classic AVR that are not recommended for new designs.

Mega AVR (ATmegaxxxx)

These are powerful microcontrollers with more than 120 instructions and lots of different peripheral capabilities, which can be used in different designs. See Table 1-3. Some of their characteristics are as follows:

- Program memory: 4K to 256K bytes
- Package: 28 to 100 pins
- Extensive peripheral set
- Extended instruction set: They have rich instruction sets.

AVR Microcontroller ATmega32 Block Diagram

Table 1-3: Some Members of the ATmega Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATmega8	8K	1K	0.5K	23	8	3	TQFP32, PDIP28
ATmega16	16K	1K	0.5K	32	8	3	TQFP44, PDIP40
ATmega32	32K	2K	1K	32	8	3	TQFP44, PDIP40
ATmega64	64K	4K	2K	54	8	4	TQFP64, MLF64
ATmega1280	128K	8K	4K	86	16	6	TQFP100, CBGA

Notes:

1. All ROM, RAM, and EEPROM memories are in bytes.
2. Data RAM (general-purpose RAM) is the amount of RAM available for data manipulation (scratch pad) in addition to the register space.
3. All the above chips have USART for serial data transfer.

AVR Microcontroller ATmega32 Block Diagram

Tiny AVR (ATtinyxxxx)

As its name indicates, the microcontrollers in this group have less instructions and smaller packages in comparison to mega family. You can design systems with low costs and power consumptions using the Tiny AVRs. See Table 1-4. Some of their characteristics are as follows:

- Program memory: 1K to 8K bytes
- Package: 8 to 28 pins
- Limited peripheral set
- Limited instruction set: The instruction sets are limited. For example, some of them do not have the multiply instruction.

AVR Microcontroller ATmega32 Block Diagram

Table 1-4: Some Members of the Tiny Family

Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATtiny13	1K	64	64	6	4	1	SOIC8, PDIP8
ATtiny25	2K	128	128	6	4	2	SOIC8, PDIP8
ATtiny44	4K	256	256	12	8	2	SOIC14, PDIP14
ATtiny84	8K	512	512	12	8	2	SOIC14, PDIP14

AVR Microcontroller ATmega32 Block Diagram

Special purpose AVR

The ICs of this group can be considered as a subset of other groups, but their special capabilities are made for designing specific applications. Some of the special capabilities are: USB controller, CAN controller, LCD controller, Zigbee, Ethernet controller, FPGA, and advanced PWM.

Table 1-5: Some Members of the Special Purpose Family

Part Num.	Code ROM	Data RAM	Data EEPROM	Max I/O pins	Special Capabilities	Timers	Pin numbers & Package
AT90CAN128	128K	4K	4K	53	CAN	4	LQFP64
AT90USB1287	128K	8K	4K	48	USB Host	4	TQFP64
AT90PWM216	16K	1K	0.5K	19	Advanced PWM	2	SOIC24
ATmega169	16K	1K	0.5K	54	LCD	3	TQFP64, MLF64

AVR product number scheme

All of the product numbers start with AT, which stands for Atmel. Now, look at the number located at the end of the product number, from left to right, and find the biggest number that is a power of 2.

This number most probably shows the amount of the microcontroller's ROM. For example, in ATmega1280 the biggest power of 2 that we can find is 128; so it has 128K bytes of ROM. In ATtiny44, the amount of memory is 4K, and so on. Although this rule has a few exceptions such as AT90PWM216, which has 16K of ROM instead of 2K, it works in most of the cases.

Other microcontrollers

There are many other popular 8-bit microcontrollers besides the AVR chip. Among them are the 805 1, HCS08, PIC, and Z8. The AVR is made by Atmel Corp, as seen in Table 1-6. Microchip produces the PIC family. Freescale (formerly Motorola) makes the HCS08 and many of its variations. Zilog produces the 28 microcontroller. The 8051 family is made by Intel and a number of other companies. To contrast the ATmega32 with the 8052 chip and PIC, examine Table 1-7.

Table 1-7: Comparison of 8051, PIC18 Family, and AVR (40-pin package)

Feature	8052	PIC18F452	ATmega32
Program ROM	8K	32K	32K
Data RAM (maximum space)	256 bytes	2K	2K
EEPROM	0 bytes	256 bytes	1K
Timers	3	4	3
I/O pins	32	35	32

AVR Microcontroller ATmega32 Block Diagram

For a comprehensive treatment of the 8051, HCS12, and PIC microcontrollers, see "The 8051 Microcontroller and Embedded Systems," "HCS12 Microcontroller and Embedded Systems," and "PIC Microcontroller and Embedded Systems" by Mazidi, et al.

Table 1-6: Some of the Companies that Produce Widely Used 8-bit Microcontrollers

Company	Web Site	Architecture
Atmel	http://www.atmel.com	AVR and 8051
Microchip	http://www.microchip.com	PIC16xxx/18xxx
Intel	http://www.intel.com/design/mcs51	8051
Philips/Signetics	http://www.semiconductors.philips.com	8051
Zilog	http://www.zilog.com	Z8 and Z80
Dallas Semi/Maxim	http://www.maxim-ic.com	8051
Freescale Semi	http://www.freescale.com	68HC11/HCS08

See http://www.microcontroller.com for a complete list.

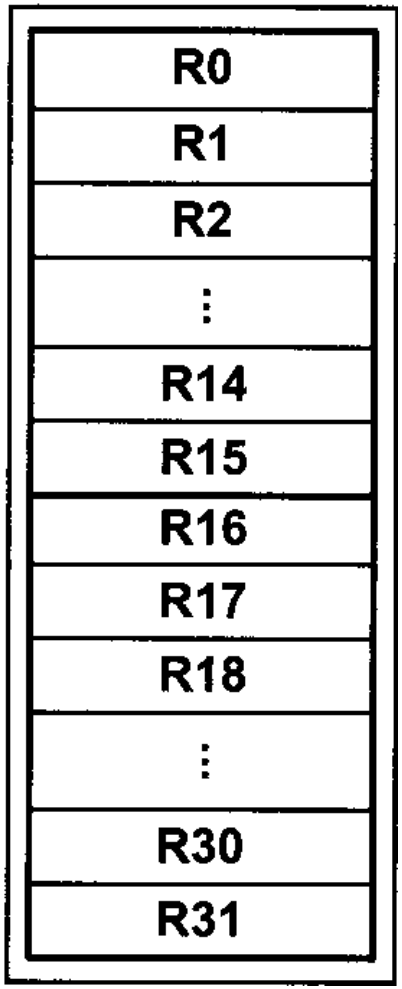
AVR Microcontroller

THE GENERAL PURPOSE REGISTERS IN THE AVR

CPUs use many registers to store data temporarily. In this section we look at the general purpose registers (GPRs) of the AVR and we demonstrate the use of GPRs with simple instructions such as LDI and ADD.

AVR microcontrollers have many registers for arithmetic and logic operations. In the CPU, registers are used to store information temporarily. That information could be a byte of data to be processed, or an address pointing to the data to be fetched. In the AVR there is only one data type: 8-bit.

In AVR there are 32 general purpose registers. They are RO-R31 and are located in the lowest location of memory address. All of these registers are 8 bits.



AVR Microcontroller

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 (Load Immediate)

Simply stated, the 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, and Rd is R16 to R31 (any of the upper 16 general purpose registers). The I in LDI stands for "immediate

AVR Microcontroller

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)
```

The following instruction loads R25 with the value 0x15 (15 in hex and 21 in decimal).

```
LDI R25,0x79      ;load 0x79 into R25 (R25 = 0x79)
```

Note: We cannot load values into registers RO to R15 using the LDI instruction.
For example the following instruction is not valid:

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

AVR Microcontroller

When programming the GPRs of the AVR microcontroller with an immediate value, the following points should be noted:

1. 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.

   For example, in "**`LDI R16,50`**", R16 is loaded with 50 in decimal, whereas in "**`LDI R16,0x50`**", R16 is loaded with 50 in hex.

2. 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.

   For example, in "**`LDI R16,0x5`**" the result will be R16 = 0x05; that is, R16 = 00000101 in binary.

3. Moving a value larger than 255 (FF in hex) into the GPRs will cause an error.

```
LDI R17,Ox7F2   ;ILLEGAL $7F2 > 8 bits (SFF)
```

AVR Microcontroller

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)

AVR Microcontroller

This figure shows the general purpose registers (GPRs) and the ALU in AVR.

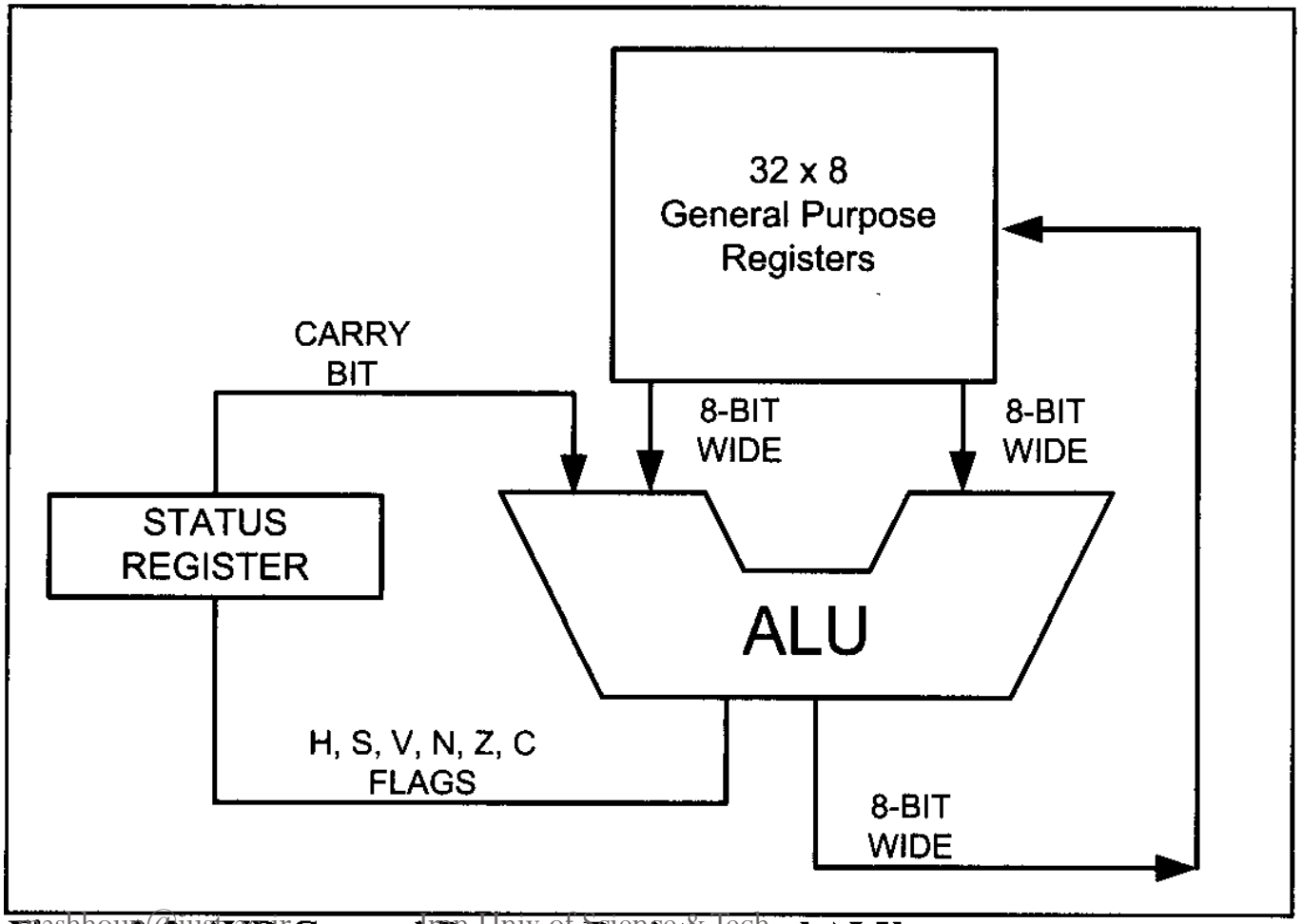


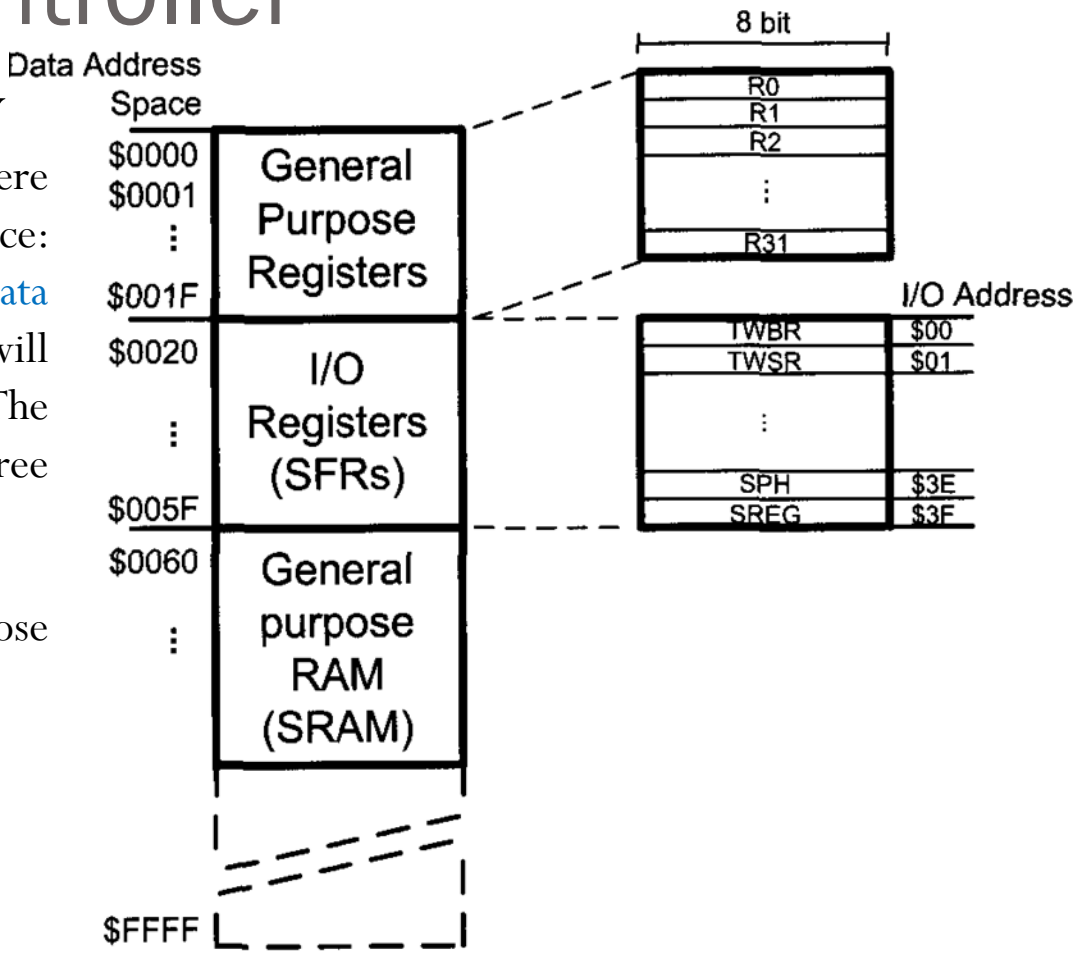
Figure 2-2. AVR General Purpose Registers and ALU

AVR Microcontroller

THE AVR DATA MEMORY

In AVR microcontrollers there are two kinds of memory space: code memory space and data memory space. here, we will discuss the data memory space. The data memory is composed of three parts:

1. GPRs (general purpose registers),
2. I/O memory, and
3. internal data SRAM.



mashhoun@iust.ac.ir Iran Univ of Science & Tech

AVR Microcontroller

GPRs (general purpose registers)

General Purpose, 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, and so on. 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 I/O memory locations. This 64-byte section is called ***standard I/O memory***.

AVR Microcontroller

In AVRs with more than 32 110 pins (e.g., ATmega64, ATmega128, and ATmega256) there is also an extended I/O memory, which contains the registers for controlling the extra ports and the extra peripherals.

In other microcontrollers the I/O registers are called ***SFR (Special Function Registers)*** since each one is dedicated to a specific function.

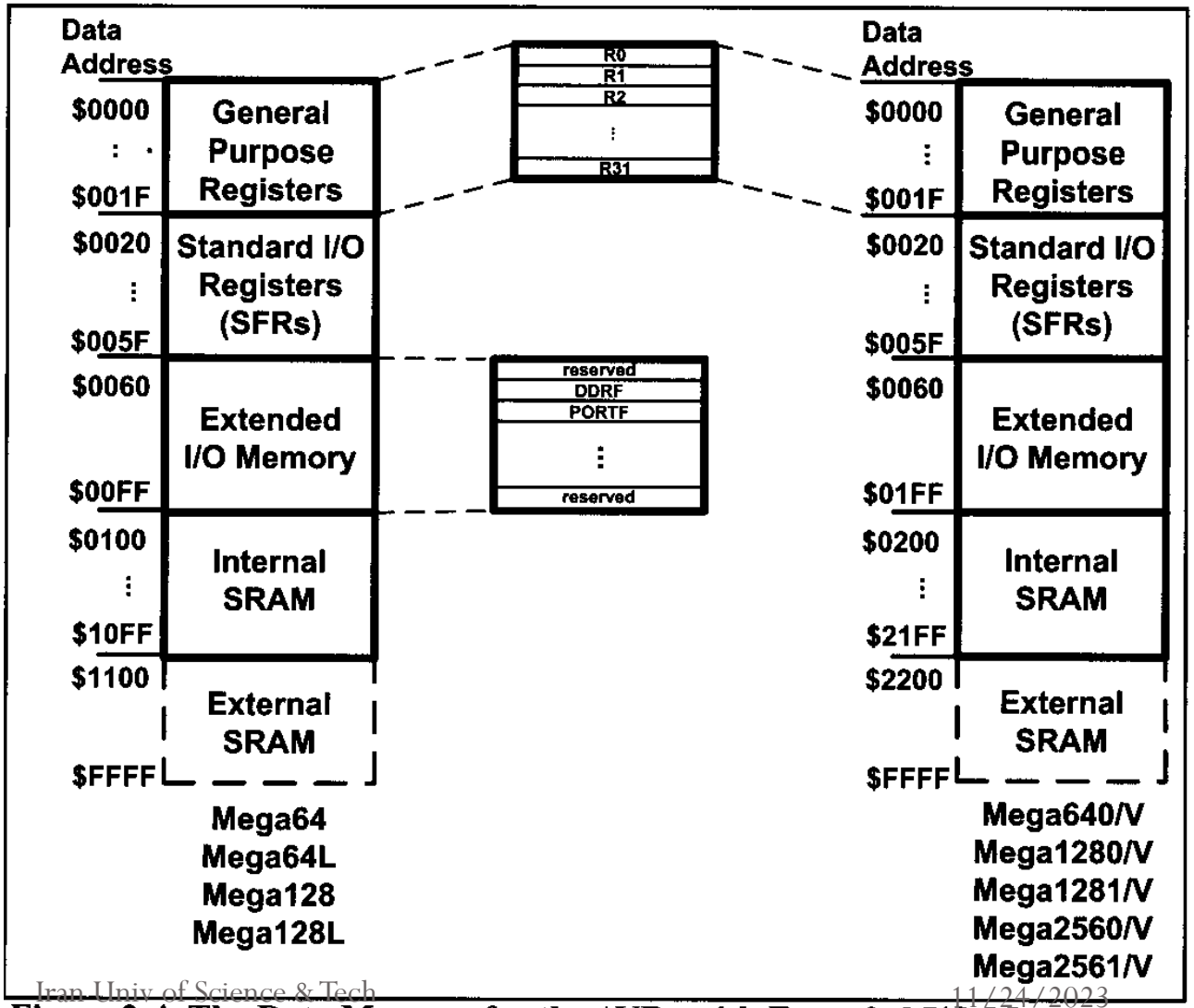


Figure 2-4. The Data Memory for the AVRs with Extended I/O Memory

AVR Microcontroller

Internal data SRAM

Internal data SRAM is widely used for storing data and parameters by AVR programmers and C compilers. Generally, this is called ***scratch pad. Each location*** of the SRAM can be accessed directly by its address. We will use these locations in future chapters to store data brought into the CPU via VO and serial ports. Each location is 8 bits wide and can be used to store any data we want as long as it is 8-bit.

Again, the size of SRAM can vary from chip to chip, even among members of the same family. **See Table 2-1 for a comparison of the data memories of various** AVR chips.

Table 2-1: 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

AVR Microcontroller

SRAM vs. EEPROM in AVR chips

The AVR has an EEPROM memory that is used for storing data. As told before, EEPROM does not lose its data when power is off, whereas SRAM does. So, 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); whereas 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. The EEPROM memory of AVR chips is covered later.

In AVR datasheets, EEPROM refers to the EEPROM's size, and SRAM is the internal SRAM size. By adding the sizes of GPR, SFRs (I/O registers), and SRAMs we get the data memory size. See Table in previous slide.

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 earlier. The AVR allows direct access to other locations in the data memory. Here we show the instructions accessing various locations of the data memory. This is one of the most important sections for mastering the topic of AVR Assembly language programming.

AVR Microcontroller

LDS instruction (LoaD direct from data Space)

```
LDS Rd, K        ;load Rd with the contents of location K 10 S d % 311
                 ;K is an address between $0000 to SFFFF
```

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.

For example, the "**`LDS R20,0x1`**" instruction will copy the contents of location 1 (in hex) into R20.

AVR Microcontroller

The following instruction loads R5 with the contents of location 0x200. As you can see in Figure 2-3, 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. To do so, first it loads RO with the contents of location 0x300 and R1 with the contents of location 0x302, then adds R0 to R1:

```
LDS R0,0x300      ;RO = the contents of location Ox300
LDS R1,0x302      ;R1 = the contents of location 0x302
ADD R1,R0         ;add RO to R1
```

AVR Microcontroller

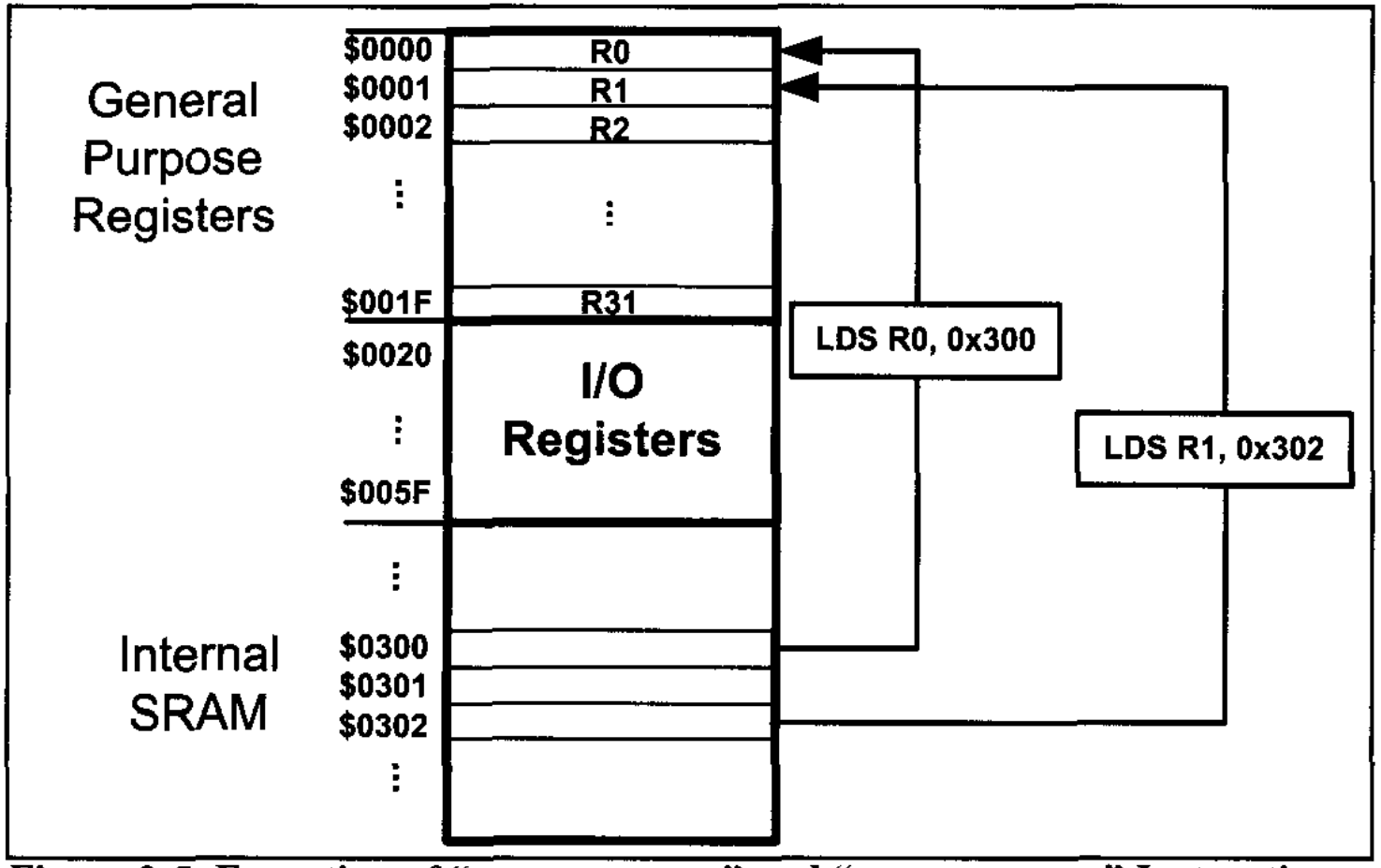


Figure 2-5. Execution of "LDS R0,0x300" and "LDS R1,0x302" Instructions

AVR Microcontroller

Figure 2-6 shows the contents of RO, R1 and locations 300 and 302 of data memory before and after the execution of each of the instructions, assuming that locations $300 and $302 contain α and β respectively.

	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

Figure 2-6. The Contents of R0, R1, and Locations $300 and $302

AVR Microcontroller

STS instruction (STore direct to data Space)

```
STS K,Rr        ;store register into location K
                ;K is an address between $0000 t o SFFFF
```

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.

For example, the "`STS 0x1,R10`" instruction will copy the contents of R10 into location 1.

AVR Microcontroller

The following instruction stores the contents of R25 to location 0x230.

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

The following program first loads the R16 register with value 0x55, then moves this value around to VO registers of ports B, C, and D. As shown in Figure 2-7, the addresses of PORTB, PORTC, and PORTD are 0x38, 0x35, and 0x32, respectively:

```
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)
```

As we saw in Figure 2-3, PORTB, PORTC, and PORTD are part of the special function registers in the I/O memory. They can be connected to the I/O pins of the AVR microcontroller as we will see later.

AVR Microcontroller

We can also store the contents of a GPR into any location in the SRAM region of the 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           ;store R20 in loc 0x202
STS 0x203,R20           ;store R20 in loc 0x203
```

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
```

AVR Microcontroller

Notice that you cannot copy (store) an immediate value directly into the SRAM location in AVR. This must be done via the GPRs.

IN instruction (IN from I/O location)

```
IN Rd,A     ;load an I/O location to the GPR (0 ≤ d ≤31),(0 ≤ A ≤ 63)
```

The IN instruction tells the CPU to load one byte from an I/O register to the GPR. After this instruction is executed, the GPR will have the same value as the I/O register. For example, the "**`IN R20, 0x16`**" instruction will copy the contents of location 16 (in hex) of the I/O memory into R20.

AVR Microcontroller

As you can see in Figure 2-7, each location in I/O memory has two addresses: I/O address and data memory address.

- Each location in the data memory has a unique address called the ***data memory address***.
- ***Each I/O register has a relative address in comparison to the*** beginning of the I/O memory; this address is called ***the I/O address***.

See Figure 2-3. You see the list of I/O registers in Figure 2-7.

AVR Microcontroller

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.

Figure 2-7. I/O Registers of the ATmega32 and Their Data Memory Address Locations

AVR Microcontroller

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,Ox3F              ;load R16 with value Ox3F
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
```

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 Ox3F;
after the execution of **`STS 0x215,R16`** data memory location $215 has value 0x63;
and so on.

AVR Microcontroller

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 t o R20
STS 0x120,R20      ;store in location 0x120 the contents of R20
```

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.

AVR Microcontroller

In the IN instruction, the I/O registers are referred to by their I/O addresses. For example, the "**IN R20,0x16**" instruction will copy the contents of location $16 of the I10 memory (whose data memory address is 0x36) into R20. As shown in Figure 2-7,110 address 0x16 belongs to PINB, so the instruction copies the contents of PINB to R20.

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. For example, the following instruction loads R19 with the contents of PIND:

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

AVR Microcontroller

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. The details of I/O ports are discussed in Chapter 4.

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 + RZ
STS 0x300, R1   ;store R1 to data space location $300
```

AVR Microcontroller

IN vs. LDS

we can load an I/O register into a GPR, using the LDS instruction. So, what is the advantage of using the IN instruction over using the LDS instruction? The IN instruction has the following advantages:

1- The CPU executes the IN instruction faster than LDS. The IN instruction lasts 1 machine cycle, whereas LDS lasts 2 machine cycles.
2- The IN is a 2-byte instruction, whereas LDS is a 4-byte instruction.
3- When we use the IN instruction, we can use the names of the I/O registers instead of their addresses.
4. The IN instruction is available in all of the AVRs, whereas LDS is not implemented in some of the AVRs.

Notice that in using the IN instruction we can access only the standard I/O memory, while we can access all parts of the data memory using the LDS instruction.

AVR Microcontroller

OUT instruction (OUT to 110 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. For example, 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).

AVR Microcontroller

The following program copies OxE6 to the SPL register:

```
LDI R20,OxE6      ;load R20 with OxE6
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 1/O reg PIND
OUT PORTA,R0      ;out R20 to PORTA
```

AVR Microcontroller

In Example 2-3 we use **JMP** to repeat an action indefinitely. **JMP** is similar to "goto" in the C language.

Example 2-3

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

Solution:

```
AGA1N:   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. It has the following format:

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

For example, 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

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
```

AVR Microcontroller

INC instruction

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

The INC instruction increments the contents of Rd by 1. For example, 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
```

AVR Microcontroller

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 subtracts 5 from the contents of location 0x300 and stores the result in location 0x320:

AVR Microcontroller

SUB instruction

The following program subtracts 5 from the contents of location 0x300 and stores the result in location 0x320:

```
LDS R0,0x300      ;RO = contents of location 0x300
LDI R16,0x5       ;R16 = 0x5
SUB R0,R16        ;R0 = R0 - R16
STS 0x320,R0      ;store the contents of R0 to location 0x320
```

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

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

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. For example, 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
```

AVR Microcontroller

Table 2-2: 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

Rd and Rr can be any of the GPRs. See Chapter 5 for examples of the instructions in Table 2-2.

AVR Microcontroller

Table 2-3: 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

Chapters 3 through 6 will show how to use the instructions in Table 2-3.

AVR Microcontroller

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 (OxAA).

```
LDI R16,0x55      ;R16 = 0x55
OUT PORTB,R16     ;copy R16 to PortB SFR (PB = 0x55)
COM R16           ;complement R16 (R16 = OxAA)
OUT PORTB,R16     ;copy R16 to PortB SFR (PB = OxAA)
```

AVR Microcontroller

Example 2-4

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

Solution:

```
        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)
```

AVR STATUS REGISTER

The flag register in the AVR is called the status register (**SReg**). Here we discuss various bits of this register and provide some examples of how it is altered.

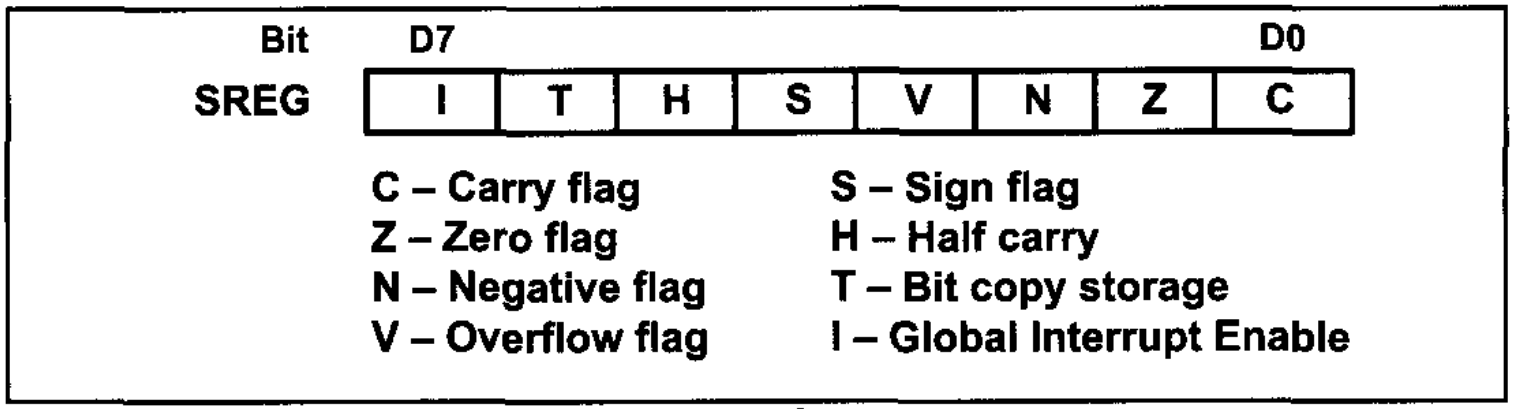


Figure 2-8. Bits of Status Register (SREG)

The status register is an 8-bit register. It is also referred to as the flag register. See Figure 2-8 for the bits of the status register.

AVR Microcontroller

STATUS REGISTER

C, the carry flag

This flag is set whenever there is a carry out from the D7 bit. This flag bit is affected after an 8-bit addition or subtraction.

Z, the zero flag

The zero flag reflects the result of an arithmetic or logic operation. If the result is zero, then Z = 1. Therefore, Z = 0 if the result is not zero.

N, the negative flag

Binary representation of signed numbers uses D7 as the sign bit. The negative flag reflects the result of an arithmetic operation. If the D7 bit of the result is zero, then N = 0 and the result is positive. If the D7 bit is one, then N = 1 and the result is negative. The negative and V flag bits are used for the signed number arithmetic operations.

V, the overflow flag

This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations while the overflow flag is used to detect errors in signed arithmetic operations.

S, the Sign flag

This flag is the result of Exclusive-ORing of N and V flags.

H, Half carry flag

If there is a carry from D3 to D4 during an ADD or SUB operation, this bit is set; otherwise, it is cleared. This flag bit is used by instructions that perform BCD (binary coded decimal) arithmetic. In some microprocessors this is called the AC flag (Auxiliary Carry flag).

ADD instruction and the status register

Next we examine the impact of the ADD instruction on the flag bits C, H, and Z of the status register. Some examples should clarify their meanings.

Examine Example ***2-5 to see the impact of the DEC instruction on selected flag bits. See*** also Examples 2-6 through 2-8 to see the impact of the ADD instruction on selected flag bits.

Example 2-5

Show the status of the Z flag during the execution of the following program:

```
LDI   R20,4        ;R20 = 4
DEC   R20          ;R20 = R20 - 1
DEC   R20          ;R20 = R20 - 1
DEC   R20          ;R20 = R20 - 1
DEC   R20          ;R20 = R20 - 1
```

AVR Microcontroller
STATUS REGISTER

Solution:

The Z flag is one when the result is zero. Otherwise, it is cleared (zero). Thus:

After	Value of R20	The Z flag
LDI R20, 4	4	0
DEC R20	3	0
DEC R20	2	0
DEC R20	1	0
DEC R20	0	1

AVR Microcontroller
STATUS REGISTER

Example 2-6

Show the status of the C, H, and Z flags after the addition of 0x38 and 0x2F in the following instructions:

```
LDI   R16, 0x38
LDI   R17, 0x2F
ADD   R16, R17     ;add R17 to R16
```

Solution:

	$38	0011 1000
+	$2F	0010 1111
	$67	0110 0111 R16 = 0x67

C = 0 because there is no carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the D4 bit.
Z = 0 because the R16 (the result) has a value other than 0 after the addition.

AVR Microcontroller
STATUS REGISTER

Example 2-7

Show the status of the C, H, and Z flags after the addition of 0x9C and 0x64 in the following instructions:

```
LDI   R20, 0x9C
LDI   R21, 0x64
ADD   R20, R21     ;add R21 to R20
```

Solution:

$9C	1001 1100	
+ $64	0110 0100	
$100	0000 0000	R20 = 00

C = 1 because there is a carry beyond the D7 bit.
H = 1 because there is a carry from the D3 to the D4 bit.
Z = 1 because the R20 (the result) has value 0 in it after the addition.

AVR Microcontroller
STATUS REGISTER

Example 2-8

Show the status of the C, H, and Z flags after the addition of 0x88 and 0x93 in the following instructions:

```
LDI   R20, 0x88
LDI   R21, 0x93
ADD   R20, R21     ;add R21 to R20
```

Solution:

$ 88	1000 1000	
+ $ 93	1001 0011	
$11B	0001 1011	R20 = 0x1B

C = 1 because there is a carry beyond the D7 bit.
H = 0 because there is no carry from the D3 to the D4 bit.
Z = 0 because the R20 has a value other than 0 after the addition.

AVR Microcontroller
STATUS REGISTER

Not all instructions affect the flags

Some instructions affect all the six flag bits C, H, Z, S, V, and N (e.g.,ADD). But some instructions affect no flag bits at all. The load instructions are in this category. And some instructions affect only some of the flag bits. The logic instructions (e.g., AND) are in this category.

Table 2-4: Instructions That Affect Flag Bits

Instruction	C	Z	N	V	S	H
ADD	X	X	X	X	X	X
ADC	X	X	X	X	X	X
ADIW	X	X	X	X	X	
AND		X	X	X	X	
ANDI		X	X	X	X	
CBR		X	X	X	X	
CLR		X	X	X	X	

AVR Microcontroller

STATUS REGISTER

COM	X	X	X	X	X	
DEC		X	X	X	X	
EOR		X	X	X	X	
FMUL	X	X				
INC		X	X	X	X	
LSL	X	X	X	X		X
LSR	X	X	X	X		
OR		X	X	X	X	
ORI		X	X	X	X	
ROL	X	X	X	X		X
ROR	X	X	X	X		
SEN			1			
SEZ		1				
SUB	X	X	X	X	X	X
SUBI	X	X	X	X	X	X
TST		X	X	X	X	

Note: X can be 0 or 1. (See Chapter 5 for how to use these instructions.)

Flag bits and decision making

There are instructions that will make a conditional jump (branch) based on the status of the flag bits. Table 2-5 provides some of these instructions.

Table 2-5: AVR Branch (Jump) Instructions Using Flag Bits

Instruction	Action
BRLO	Branch if C = 1
BRSH	Branch if C = 0
BREQ	Branch if Z = 1
BRNE	Branch if Z = 0
BRMI	Branch if N = 1
BRPL	Branch if N = 0
BRVS	Branch if V = 1
BRVC	Branch if V = 0

AVR DATA FORMAT AND DIRECTIVES

The AVR microcontroller has only one data type. It is 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 OxFF, or 0 to 255 in decimal) to be processed by the CPU.

Data format representation

There are four ways to represent a byte of data in the AVR assembler. The numbers can be in hex, binary, decimal, or ASCII formats.

AVR Microcontroller

AVR DATA FORMAT AND DIRECTIVES

Hex numbers

There are two ways to show hex numbers:

1. Put 0x (or 0X) in front of the number like this: LDI R16,0x99
2. Put $ in front of the number, like this: LDI R22,$99

Here are a few lines of code that use the hex format:

```
LDI  R28,$75          ;R28 = 0x75
SUBI R28,0x11         ;R28 = 0x75 - 0x11 = 0x64
SUBI R28,0X20         ;R28 = 0x64 - 0x20 = 0x44
ANDI R28,0xF          ;R28 = 0x44 - 0x0F = 0x35
```

Binary numbers

There is only one way to represent binary numbers in an AVR assembler. It is as follows:

```
LDI R16,0b10011001          ;R16 = 10011001 or 99 in hex
```

The uppercase B will also work. Here are some examples of how to use it:

```
LDI  R23,0b00100101         ;R23 = $25
SUB1 R23,OB00010001         ;R23 = $25 - $11 = $14
```

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 OC in hex
SUBI R17,2        ;R17 = 12 - 2 = 10 where 10 is equal to OxOA
```

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
```

This is the same as other assemblers such as the 805 1 and x86. Here are some more examples:

```
LDI R20,'9'       ;R20 = 0x39,which is hex number for ASCII '9'
SUBI R20,'1'      ;R20 = 0x39 - 0x31 = 0x8
                  ;(31 hex is for ASCII '1')
```

To represent a string, double quotes are used; and for defining ASCII strings, we use the .DB (define byte) directive.

Assembler directives

While instructions tell the CPU what to do, directives (also called pseudo instructions) give directions to the assembler. For example, the LDI and ADD instructions are commands to the CPU, but .EQU, .DEVICE, and .ORG are directives to the assembler.

.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
```

AVR Microcontroller

AVR DATA FORMAT AND DIRECTIVES

As 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 0xlB,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 0xlB (the address of PORTB) with the PORTB.

```
.EQU PORTB = OxlB
```

.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.

Using .EQU for fixed data assignment

To get more practice using .EQU to assign fixed data, examine the following:

```
                ;in hexadecimal
.EQU DATA1 = 0x39           ;one way to define hex value
.EQU DATA2 = $39            ;another way to define hex value

                ;in binary
.EQU DATA3 = 0b00110101     ;binary (35 in hex)

                ;in decimal
.EQU DATA4 = 39             ;decimal numbers (27 in hex)

                ;in ASCII
.EQU DATA5 = '2'            ;ASCII characters
```

We use .DB to allocate code ROM memory locations for fixed data such as ASCII strings.

Using .EQU for SFR address assignment

.EQU is also widely used to assign SFR addresses. Examine the following code:

```
.EQU COUNTER = 0x00          ;counter value 00
.EQU PORTB = 0x18            ;SFR Port B address
LDI R16,COUNTER              ;R16 = 0x00
OUT PORTB,R16                ;Port B (loc 0x18) now has 00 too
```

. ORG (origin)

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

Using .EQU for RAM address assignment

Another common usage of .EQU is for the address assignment of the internal SRAM. Examine the following rewrite of an earlier example using .EQU:

```
.EQU SUM = 0x120 ;assign RAM loc to SUM
LDI R20,5        ;load R20 with 5
LDI R21,2        ;load R21 with 2
ADD R20,R21      ;R20 = R20 t R21
ADD R20,R21      ;R20 = R20 + R21
STS SUM,R20      ;store the result in loc 0x120
```

This is especially helpful when the address needs to he changed in order to use a different AVR chip for a given project. It is much easier to refer to a name than a number when accessing RAM address locations.

AVR Microcontroller

AVR DATA FORMAT AND DIRECTIVES

.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). In Table 2-6, you see the files that you must include whenever you want to use any of the AVRs.

For example, when you want to use ATmega32, you must write the following instruction at the beginning of your program:

```
.INCLUDE    "M32DEF.INC"
```

Table 2-6: Some of the Common AVRs and Their Include Files

ATMEGA		ATTINY		Special Purpose	
ATmega8	m8def.inc	ATtiny11	tn11def.inc	AT90CAN32	can32def.inc
ATmega16	m16def.inc	ATtiny12	tn12def.inc	AT90CAN64	can64def.inc
ATmega32	m32def.inc	ATtiny22	tn22def.inc	AT90PWM2	pwm2def.inc
ATmega64	m64def.inc	ATtiny44	tn44def.inc	AT90PWM3	pwm3def.inc
ATmega128	m128def.inc	ATtiny85	tn85def.inc	AT90USB646	usb646def.inc
ATmega256	m256def.inc				
ATmega2560	m2560def.inc				

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.

- First, 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 0,and dollar sign ($).
- The first character of the label must be an alphabetic character.
- 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. In addition to the mnemonics there are some other reserved words.

INTRODUCTION TO AVR ASSEMBLY ROGRAMMING

A program that consists of 0s and 1s is called machine language. In the early days of the computer, programmers coded programs in machine language. Although the hexadecimal system was used as a more efficient way to represent binary numbers, the process of working in machine code was still cumbersome for humans. Eventually, Assembly languages were developed, which provided mnemonics for the machine code instructions, plus other features that made programming faster and less prone to error.

Assembly language programs must be translated into machine code by a program called an assembler. Assembly language is referred to as a low-level language because it deals directly with the internal structure of the CPU. To program in Assembly language, the programmer must know all the registers of the CPU and the size of each, as well as other details.

Structure of Assembly language

An Assembly language program consists of, among other things, a series of lines of Assembly language instructions.

An Assembly language instruction consists of a mnemonic, optionally followed by one or two operands. The operands are the data items being manipulated, and the mnemonics are the commands to the CPU, telling it what to do with those items.

An Assembly language instruction consists of four fields:

```
[ label:] mnemonic [ operands] [ ;comment]
```

AVR Microcontroller

AVR DATA FORMAT AND DIRECTIVES

```
        ;AVR Assembly Language Program To Add Some Data.
        ;store SUM in SRAM location 0x300.

        .EQU  SUM   = 0x300      ;SRAM loc $300 for SUM

        .ORG 00                  ;start at address 0
        LDI R16, 0x25            ;R16 = 0x25
        LDI R17, $34             ;R17 = 0x34
        LDI R18, 0b00110001      ;R18 = 0x31
        ADD R16, R17             ;add R17 to R16
        ADD R16, R18             ;add R18 to R16
        LDI R17, 11              ;R17 = 0x0B
        ADD R16, R17             ;add R17 to R16
        STS SUM, R16             ;save the SUM in loc $300
HERE:   JMP HERE                 ;stay here forever
```

Program 2-1: Sample of an Assembly Language Program

AVR Microcontroller

AVR DATA FORMAT AND DIRECTIVES

Regarding the above format, the following points should be noted:

1. The label field allows the program to refer to a line of code by name. The label field cannot exceed a certain number of characters. Check your assembler for the rule.
2. The Assembly language mnemonic (instruction) and operand(s) fields together perform the real work of the program and accomplish the tasks for which the program was written. In Assembly language statements such as

```
LDI   R23,$55
ADD   R23,R19
SUB1 R23,$67
```

ADD and LDI are the mnemonics that produce opcodes; the "$55" and "$67" are the operands.

AVR Microcontroller

AVR DATA FORMAT AND DIRECTIVES

Regarding the above format, the following points should be noted:

3. The comment field begins with a semicolon comment indicator ";". Comments may be at the end of a line or on a line by themselves.

4. Notice the label "HERE" in the label field in Program 2-1. In the JMP the AVR is told to stay in this loop indefinitely. If your system has a monitor program you do not need this line and should delete it from your program.

AVR Microcontroller

ASSEMBLING AN AVR PROGRAM

Now that the basic form of an Assembly language program has been given, the next question is:

How it is created, assembled, and made ready to run?

The steps to create an executable Assembly language program (Figure 2-9) are outlined as follows:

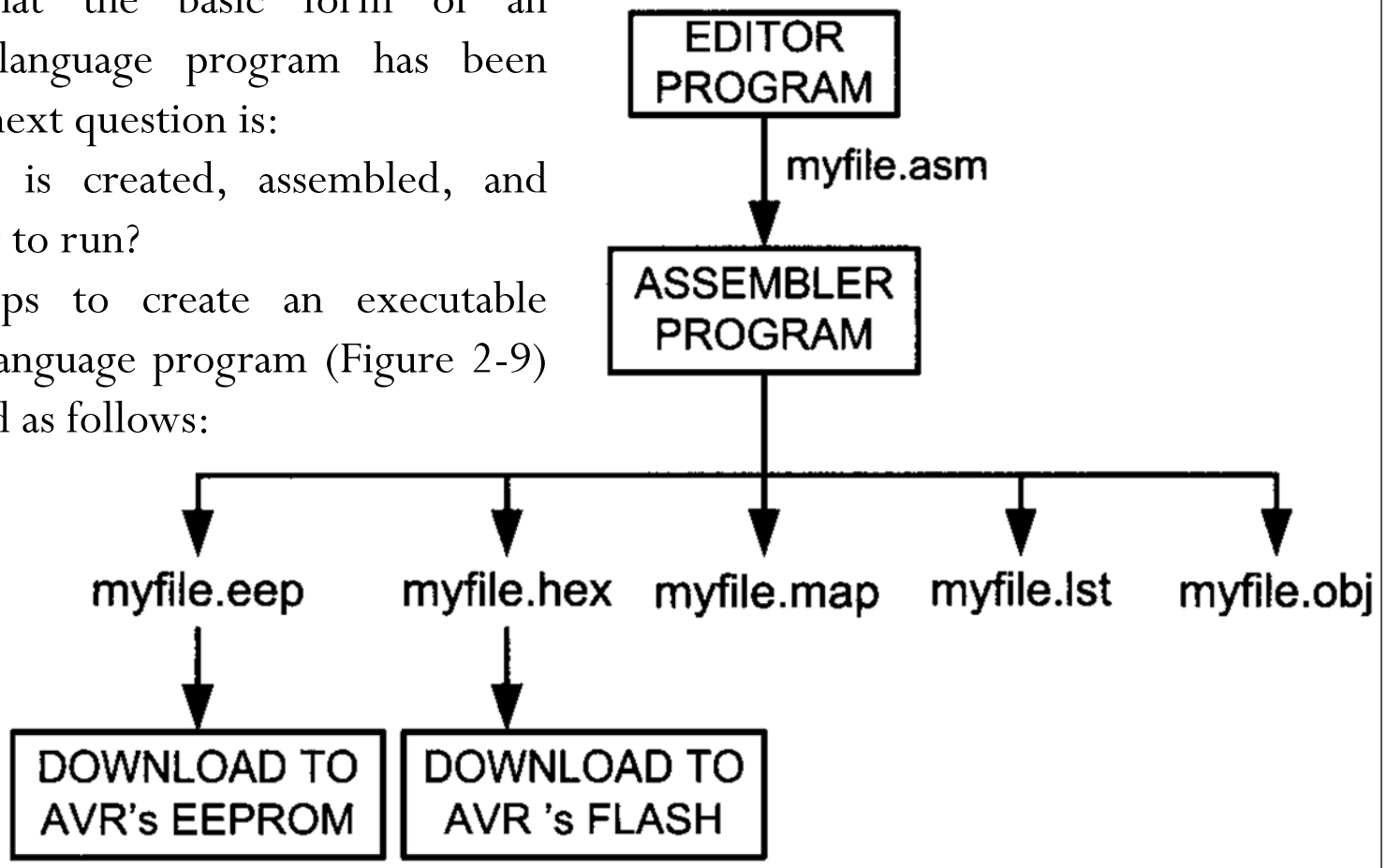


Figure 2-9. Steps to Create a Program

AVR Microcontroller

ASSEMBLING AN AVR PROGRAM

1. First we use a text editor to type in a program similar to Program 2- 1. In the case of the AVR microcontrollers, we use the AVRStudio IDE, which has a text editor, assembler, simulator, and much more all in one software package. It is an excellent development software that supports all the AVR chips and is free.
   For assemblers, the file names follow the usual DOS conventions, but the source file has the extension "asm". The "asm" extension for the source file is used by an assembler in the next step.
2. The asm source file containing the program code created in step 1 is fed to the AVR assembler. The assembler produces an object file, a hex file, an eeprom file, a list file, and a map file. The object file has the extension "obj", the hex file extension is "hex", the list file extension is "lst", the map file extension is "map", and the eeprom file has the extension "eep". After a successful link, the hex file is ready to be burned into the AVR's program ROM and is downloaded into the AVR Trainer.

AVR Microcontroller
ASSEMBLING AN AVR PROGRAM

1. First we use a text editor to type in a program similar to Program 2- 1.
2. The "asm" source file containing the program code created in step 1 is fed to the AVR assembler. The assembler produces an object file, a hex file, an eeprom file, a list file, and a map file. The object file has the extension "obj", the hex file extension is "hex", the list file extension is "lst", the map file extension is "map", and the eeprom file has the extension "eep".

   After a successful link, the hex file is ready to be burned into the AVR's program ROM and is downloaded into the AVR Trainer. We can write the eeprom file into the AVR's EEPROM to initialize the EEPROM.

More about asm and object files

The asm file is also called the source file and must have the "asm" extension. The object file is used as input to a simulator or an emulator.

Before we can assemble a program to create a ready-to-run program, we must make sure that it is error free. The AVR Studio IDE provides us error messages and we examine them to see the nature of syntax errors. The assembler will not assemble the program until all the syntax errors are fixed. A sample of an error message is shown in Figure 2-10.

```
AVRASM: AVR macro assembler 2.1.2 (build 99 Nov  4 2005 09:35:05)
Copyright (C) 1995-2005 ATMEL Corporation

F:\AVR\Sample\Sample.asm(7): error: Invalid register
F:\AVR\Sample\Sample.asm(8): error: Operand(s) out of range in 'ldi r17,0x3432'
F:\AVR\Sample\Sample.asm(9): error: Undefined symbol: R38
F:\AVR\Sample\Sample.asm(9): error: Invalid register
F:\AVR\Sample\Sample.asm(16): No EEPROM data, deleting F:\AVR\Sample\Sample.eep

Assembly failed, 4 errors, 0 warnings
```

Figure 2-10. Sample of an AVR Error

AVR Microcontroller
ASSEMBLING AN AVR PROGRAM

"map" files

The map file shows the labels defined in the program together with their values. Examine Figure 2-11. It shows the Map file of Program 2-1.

```
AVRASM ver. 2.1.2   F:\AVR\Sample\Sample.asm Sun Apr 06 23:39:32 2008


EQU   SUM           00000300
CSEG  HERE          00000009
```

Figure 2-11. Map File of Program 2-1

AVR Microcontroller
ASSEMBLING AN AVR PROGRAM

"lst" files

list shows the binary and source code; it also shows which instructions are used in the source code, and the amount of memory the program uses. See Figure 2-12.

```
AVRASM ver. 2.1.2  F:\AVR\Sample\Sample.asm Tue Mar 11 11:28:34 2008

                ;store SUM in SRAM location 0x300.
                   .DEVICE ATMega32
                   .EQU  SUM   = 0x300     ;SRAM loc $300 for SUM

                   .ORG 00                 ;start at address 0
000000 e205        LDI R16, 0x25           ;R16 = 0x25
000001 e314        LDI R17, $34            ;R17 = 0x34
000002 e321        LDI R18, 0b00110001     ;R18 = 0x31
000003 0f01        ADD R16, R17            ;add R17 to R16
000004 0f02        ADD R16, R18            ;add R18 to R16
000005 e01b        LDI R17, 11             ;R17 = 0x0B
000006 0f01        ADD R16, R17            ;add R17 to R16
000007 9300 0300   STS SUM, R16            ;save the SUM in loc $300
000009 940c 0009 HERE: JMP HERE            ;stay here forever

RESOURCE USE INFORMATION
------------------------
...
Memory use summary [bytes]:
Segment    Begin     End       Code    Data    Used    Size    Use%
---------------------------------------------------------------------
[.cseg] 0x000000 0x000016      22       0      22 unknown       -
[.dseg] 0x000060 0x000060       0       0       0 unknown       -
[.eseg] 0x000000 0x000000       0       0       0 unknown       -

Assembly complete, 0 errors, 0 warnings
```

Figure 2-12. List File of Program 2-1

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

Program counter in the AVR

The most important register in the AVR microcontroller is the PC. The program counter is used by the CPU to point to the address of the next instruction to be executed. As the CPU fetches the opcode from the program ROM, the program counter is incremented automatically to point to the next instruction.

In AVR microcontrollers each Flash memory location is 2 bytes wide. For example, in ATmega32, whose Flash is 32K bytes, the Flash is organized as 16Kx 16, and its program counter is 14 bits wide (2^{14} = 16K memory locations).

The ATmega64 has a 15-hit program counter, so its Flash has 32K locations (2^{15}=32K), with each location containing 2 bytes (32K x 2 bytes = 64K bytes).

AVR Microcontroller

PROGRAM COUNTER and ROM SPACE

In the case of a 16-bit program counter, the code space is 64K (2^{16} = 64K), which occupies the 0000-$FFFF address range. The program counter in the AVR family can be up to 22 bits wide. This means that the AVR family can access program addresses 000000 to $3FFFFF, a total of 4M locations. Because each Flash location is 2 bytes wide, the AVR can have a maximum of 8M bytes of code. However, at the time of this writing, none of the members of the AVR family have the entire 8M bytes of on-chip ROM installed.

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

Table 2-7: AVR On-chip ROM Size and Address Space

	On-chip Code ROM (Bytes)	Code Address Range (Hex)	ROM Organization
ATtiny25	2K	00000–003FF	1K × 2 bytes
ATmega8	8K	00000–00FFF	4K × 2 bytes
ATmega32	32K	00000–03FFF	16K × 2 bytes
ATmega64	64K	00000–07FFF	32K × 2 bytes
ATmega128	128K	00000–0FFFF	64K × 2 bytes
ATmega256	256K	00000–1FFFF	128K × 2 bytes

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

ROM memory map in the AVR family

It must be noted that while the first location of program ROM inside the AVR has the address of 000000, the last location can be different depending on the size of the ROM on the chip.

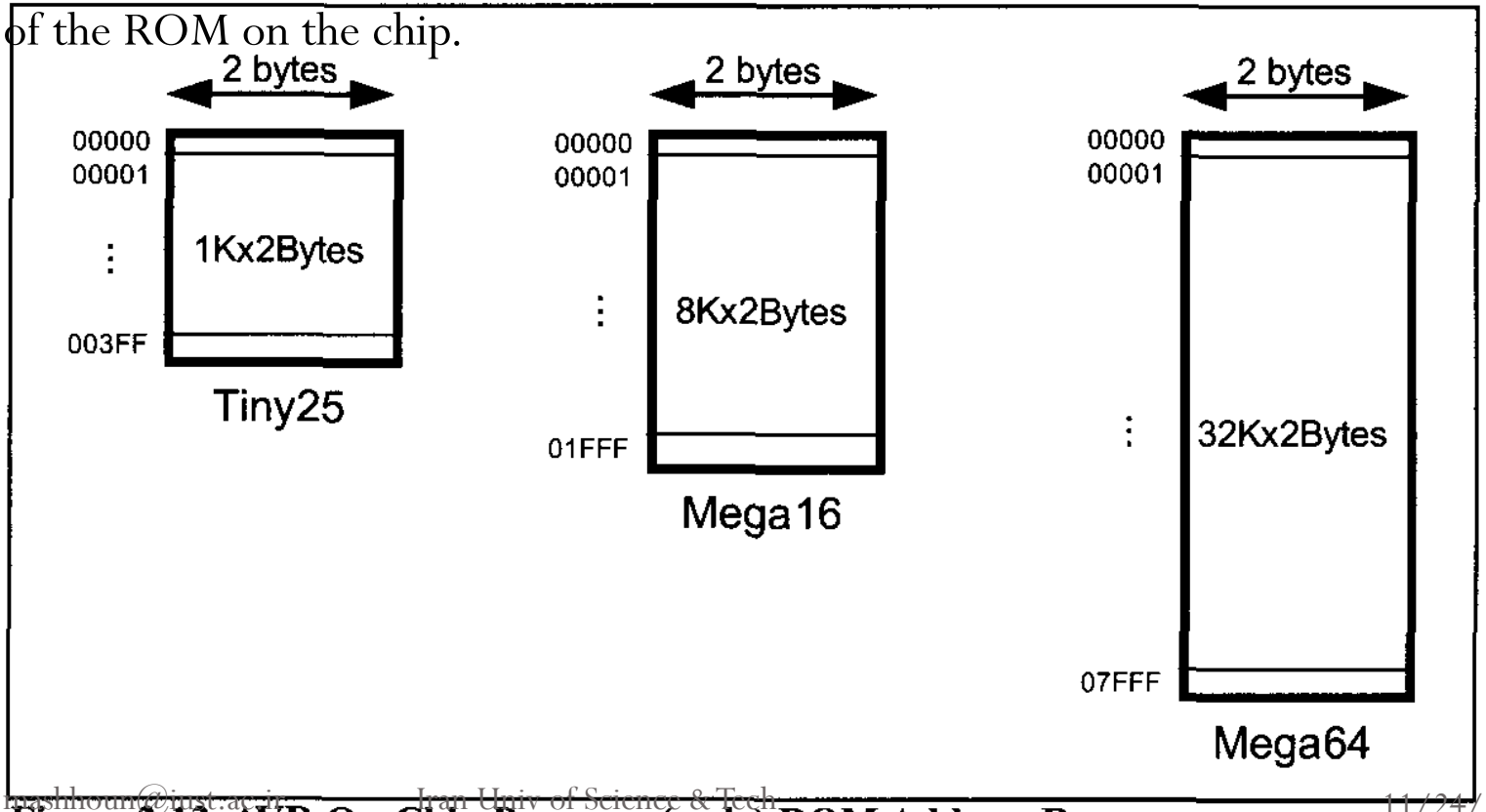


Figure 2-13. AVR On-Chip Program (code) ROM Address Range

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

Where the AVR wakes up when it is powered up

In the AVR microcontrollers, the microcontroller wakes up at memory address 0000 when it is powered up.

When the AVR is powered up, the

$$PC = 00000$$

Placing code in program ROM

The list shows that address 0000 contains E205, which is the opcode for moving a value into R16, and the operand to be moved to R16. Therefore, the instruction

"LDI R16, 0x25"

has a machine code of "E205", where E is the opcode and 205 is the operand.

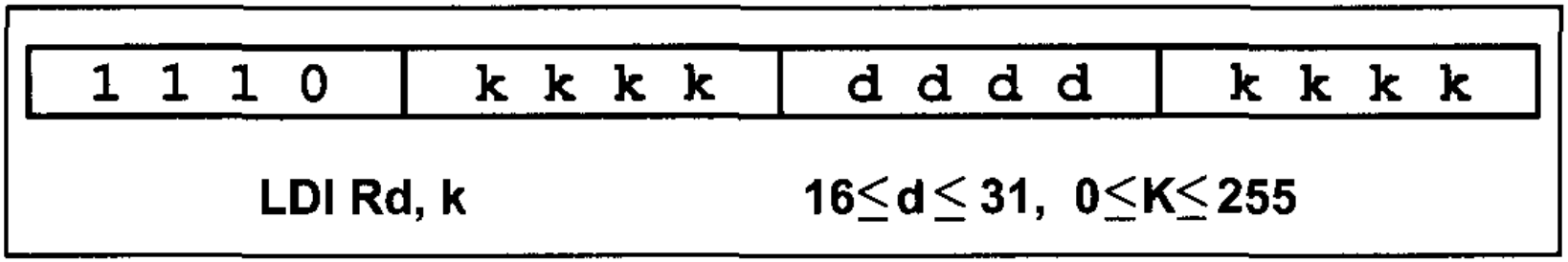


Figure 2-14. The Machine Code for Instruction "LDI Rd, k" in Binary

Placing code in program ROM

Similarly, the machine code "E3 14" is located in ROM memory location 0001 and represents the opcode and the operands for the instruction "LDI R17, $34".

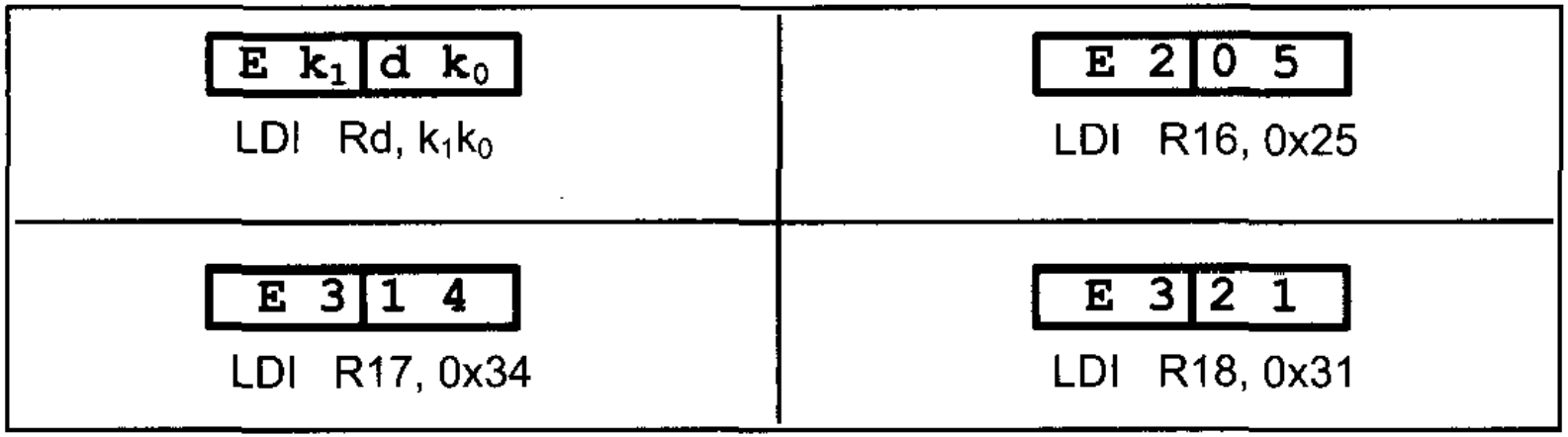


Figure 2-15. The Machine Code for Instruction "LDI Rd, k" in Hex

the same way, machine code "E321" is located in memory location 0002 and represents the opcode and the operand for the instruction LDI R18, 0b00110001

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

The memory location 0003 has the machine code of 0F01, which is the opcode and the operands for the instruction "ADD R16, R17".

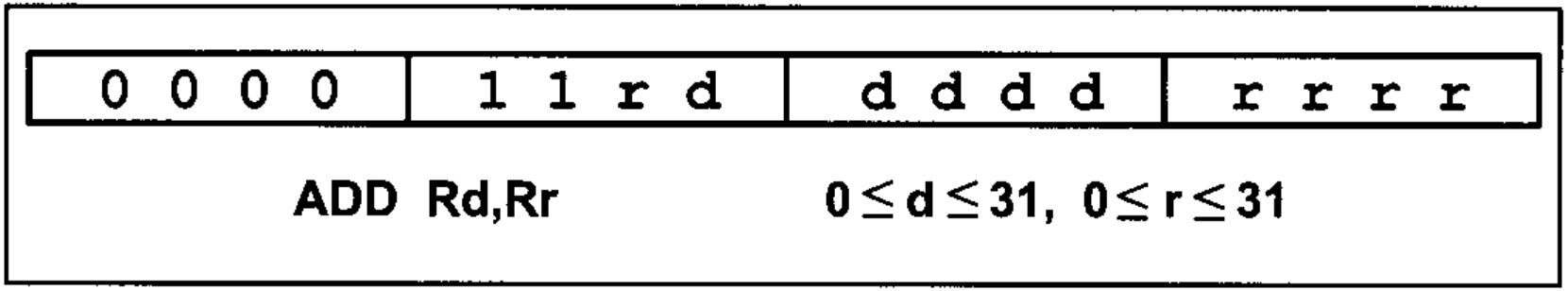


Figure 2-16. The Machine Code for Instruction "ADD Rd,Rr" in Binary

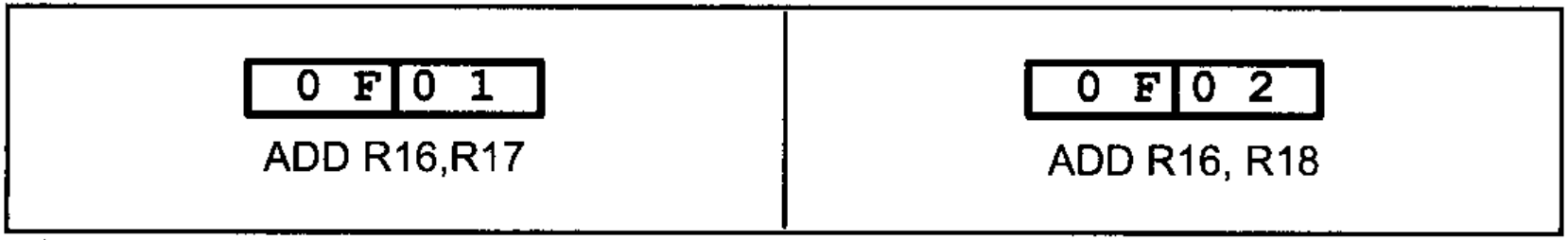


Figure 2-17. The Machine Code for Instruction "ADD Rd,Rr" in Hex

Similarly, the machine code "0F02" is located in memory location 0004 and represents the opcode and the operands for the instruction "ADD R16, R18".

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

The memory location 0005 has the opcode and operand for the "LDI R17,11" instruction.

The memory location 0006 has the opcode and operand for the "ADD R16, R17" instruction.

The opcode for instruction "STS SUM, R16" is located at address 00007 and its address of 0x300 at address 00008.

The opcode for "JMP HERE" and its target address are located in locations 00009 and 0000A.

While all the instructions in this program are 2-byte instructions, the JMP and STS instructions are 4-byte instructions.

Executing a program instruction by instruction

Assuming that the above program is burned into the ROM of an AVR chip, the following is a step-by-step description of the action of the AVR upon applying power to it:

1. The instruction LDI R16,0x25 is executed Then the program counter is incremented to point to 00001, which contains code E314, the machine code for the instruction "LDI R17, 0x34".
2. Upon executing the machine code E3 14, the value 0x34 is loaded to R17. Then the program counter is incremented to 0002.
3. ROM location 0002 has the machine code for instruction "LDI R18, 0x31". This instruction is executed and now PC = 0003.
4. This process goes on until all the instructions up to "ADD R16, R17" are fetched and executed. Notice that all the above instructions are 2-byte instructions; that is, each one takes two bytes of ROM (one word).

Executing a program instruction by instruction

5. Now PC = 0007 points to the next instruction, which is "STS SUM, R16". This is a 2-word (4-byte) instruction. It takes addresses of 07 and 08. When the instruction is executed, the content of R16 is stored into memory location 0x300. After the execution of this instruction, PC = 0009.
6. Now PC = 0009 points to the next instruction, which is "JMP HERE". This is a 2-word (4-byte) instruction. It takes addresses of 09 and OA. After the execution of this instruction, PC = 0009. This keeps the program in an infinite loop.

AVR Microcontroller

PROGRAM COUNTER and ROM SPACE

ROM width in the AVR

To bring in more information (code or data) into the CPU, AVR increased the width of data bus to 16 bits. In other words, the AVR is word addressable. For the AVR, the internal data bus between the code ROM and the CPU is 16 bits wide, as shown data size.

The widening of the data path between the program ROM and the CPU is another way in which the AVR designers increased the processing power of the AVR family.

Another reason to make the code ROM 16 bits wide is to match it with the instruction width of the AVR because the vast majority of the instructions are 2-byte instructions.

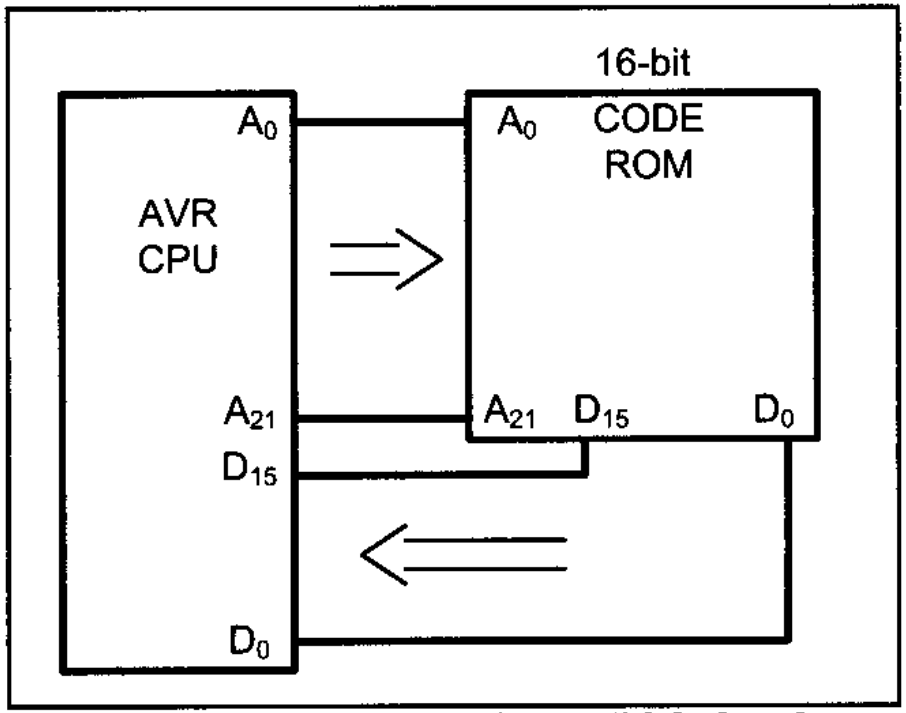


Figure 2-18. Program ROM Width for the AVR

AVR Microcontroller

PROGRAM COUNTER and ROM SPACE

The AVR designers have made all instructions either 2-byte or 4-byte; there are no 1-byte or 3-byte instructions, as is the case with the x86 and 8051 chips.

This is part of the RISC architectural philosophy. It must also be noted that the data memory SRAM in the AVR microcontroller is still 8-bit, and it is byte-addressable.

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

Harvard architecture in the AVR

AVR uses Harvard architecture, which means that there are separate buses for the code and the data memory. The Program Bus provides access to the Program Flash ROM whereas the Data Bus is used for bringing data to the CPU.

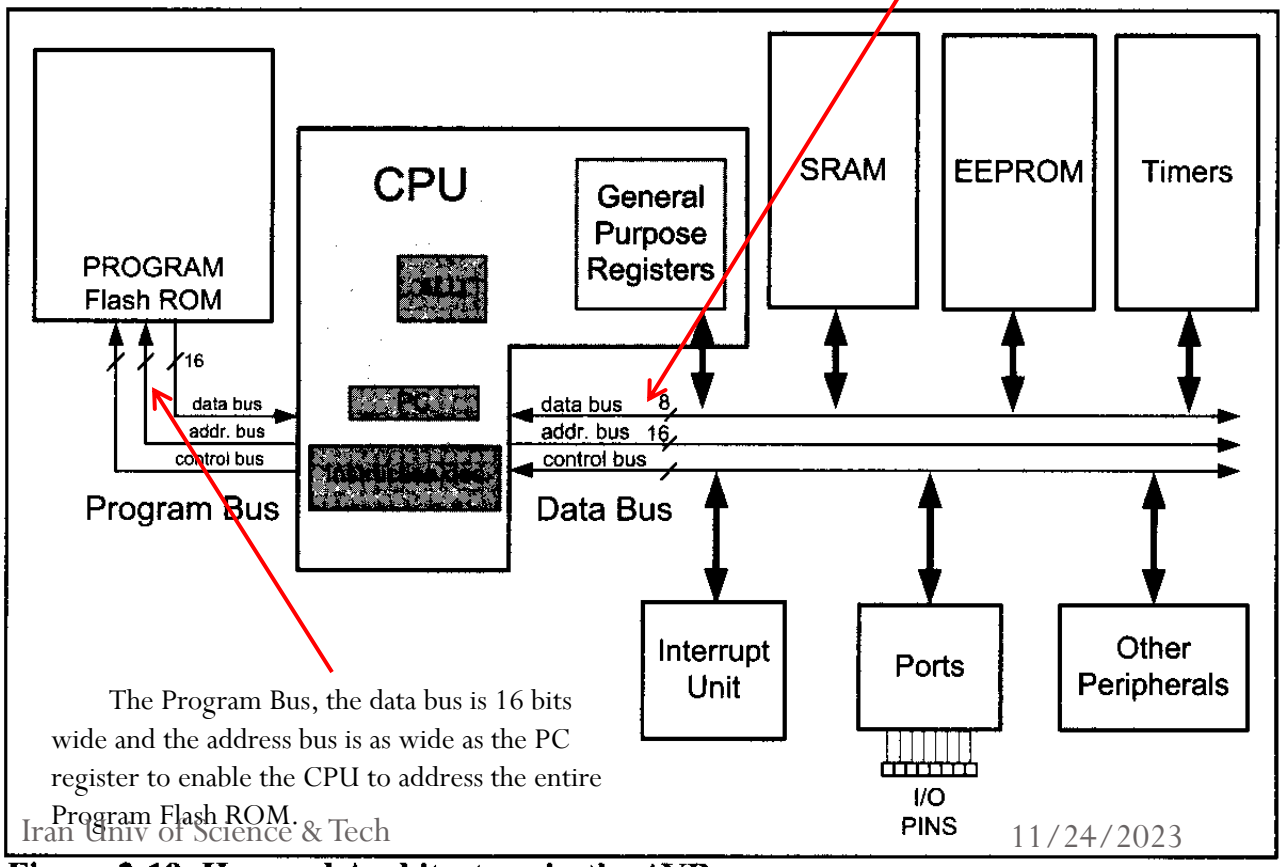


Figure 2-19. Harvard Architecture in the AVR

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

When the CPU wants to execute the

```
LDS Rn,k
```

instruction, it puts k on the address bus of the Data Bus, and receives data through the data bus.

For example, to execute

```
LDS R20, 0x90
```

the CPU puts 0x90 on the address bus. The location $90 is in the SRAM. Thus, the SRAM puts the contents of location $90 on the data bus. The CPU gets the contents of location $90 through the data bus and puts it in R20.

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

The "**`STS k,Rn`**" instruction is executed similarly. The CPU puts k on the address bus and the contents of Rn on the data bus. The unit whose address is on the address bus receives the contents of data bus.

For example, to execute the

```
STS $100,R30
```

instruction the CPU puts the contents of R30 on the data bus and $100 on the address bus.

Because $100 is bigger than $60, the address belongs to SRAM; thus SRAM gets the contents of the data bus and puts it in location $100 of the SRAM.

Little endian vs. big endian war

little endian

The low byte goes to the low memory location, and the high byte goes to the high memory address.

big endian

In the big endian method, the high byte goes to the low address, whereas in the little endian method, the high byte goes to the high address and the low byte to the low address.

All Intel microprocessors and many microcontrollers use the little endian convention. Freescale (formerly Motorola) microprocessors, along with some mainframes, use big endian.

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

Instruction size of the AVR

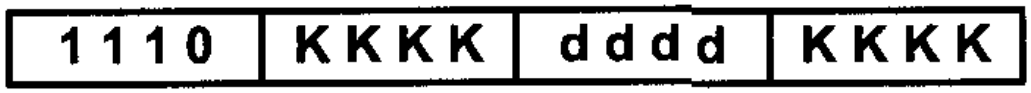
Almost all the instructions in the AVR are 2-byte instructions.

The exceptions are STS, JMP, and a few others.

LDI instruction formation

The LDI is a 2-byte (16-bit) instruction. Of the 16 bits, the first 4 bits are set aside for the opcode, the second and the fourth 4 bits are used for the value of 00 to $FF, and the third 4 bits present the destination register.

```
LDI Rd, K    ;load register Rd with value K
```



$16 \le d \le 31,\ 0 \le K \le 255$

AVR Microcontroller
PROGRAM COUNTER and ROM SPACE

ADD instruction formation

The ADD is a 2-byte (16-bit) instruction. Of the 16 bits, the first 6 bits are set aside for the opcode, and the other 10 bits represent the source and the destination registers. This is shown below

```
ADD Rd, Rr    ;Add Rd to Rr
```

$0 \le d \le 31,\ 0 \le r \le 31$

STS instruction formation

The STS is a 4-byte (32-bit) instruction. Of the 32 bits, the first 16 bits are set aside for the opcode and the address of the source, and the other 16 bits are used for the address of the destination. This is shown below:

```
STS k, Rr    ;Store register Rr in memory location k
```

1001	001r	rrrr	0000
kkkk	kkkk	kkkk	kkkk

$0 \le r \le 31, 0 \le k \le 65535$

LDS instruction formation

The LDS is a 4-byte (32-bit) instruction. Of the 32 bits, the first 16 bits are set aside for the opcode and the destination register, and the other 16 bits are used for the address of the source memory location. This is shown below.

```
LDS Rd, k    ;Load from memory location k to register Rd
```

1001	000d	dddd	0000
kkkk	kkkk	kkkk	kkkk

$0 \le d \le 31, 0 \le k \le 65535$

IN instruction formation

The IN is a 2-byte (16-bit) instruction. Of the 16 bits, the first 5 bits are set aside for the opcode, and the other 11 bits are used for the address of the source memory location, and destination register. This is shown below.

```
IN Rd, A   ;load from Address A of I/O memory into register Rd
```

1011	0AAd	dddd	AAAA

$0 \le d \le 31, 0 \le A \le 63$

OUT instruction formation

The OUT is a 2-byte (16-bit) instruction. Of the 16 bits, the first 5 bits are set aside for the opcode, and the other 11 bits are used for the address of the source memory location and destination register. This is shown below.

```
OUT A, Rr   ;Store register Rr in I/O memory location A
```

$0 \le d \le 31, 0 \le A \le 63$

JMP instruction formation

The JMP is a 4-byte (32-bit) instruction. Of the 32 bits, only 10 bits are set aside for the opcode, and the rest (22 bits) are used for the target address of the JMP. This is shown below.

The 22-bit address gives us 4M of address space; so, it can address all of the ROM space.

```
JMP k    ;Jump to address k
```

1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

$0 \le k \le 4M$

AVR Microcontroller

RISC ARCHITECTURE IN THE AVR

There are three ways available to microprocessor designers to increase the processing power of the CPU:

1. Increase the clock frequency of the chip. One drawback of this method is that the higher the frequency, the more power and heat dissipation. Power and heat dissipation is especially a problem for hand-held devices
2. Use Harvard architecture by increasing the number of buses to bring more information (code and data) into the CPU to be processed.
3. Change the internal architecture of the CPU and use what is called RISC architecture.

Atmel used all three methods to increase the processing power of the AVR microcontrollers.

RISC architecture

Since the 1960s, in all mainframes and minicomputers, designers put as many instructions as they could think of into the CPU. Some of these instructions performed complex tasks. Because microprocessors used such a large number of instructions, many of which performed highly complex activities, they came to be known as CISC (Complex Instruction Set Computer) processors.

According to several studies in the 1970s, many of these complex instructions etched into CPUs were never used by programmers and compilers. The huge cost of implementing a large number of instructions (some of them complex) into the microprocessor, plus the fact - , . . that a good portion of the transistors on the chip are used by the instruction decoder, made some designers think of simplifying and reducing the number of instructions. As this concept developed, the resulting processors came to be known as RISC (Reduced Instruction Set Computer).

AVR Microcontroller

RISC ARCHITECTURE IN THE AVR

Features of RISC

The following are some of the features of RISC as implemented by the AVR microcontroller

1. **RISC processors have a fixed instruction size.** Therefore, the CPU can decode the instructions quickly. This is like a bricklayer working with bricks of the same size as opposed to using bricks of variable sizes. In the last section we saw how the AVR uses 2-byte instructions with very few 4-byte instructions.
2. **Large number of registers**. All RISC architectures have at least 32 registers. Of these 32 registers, only a few are assigned to a dedicated function. One advantage of a large number of registers is that it avoids the need for a large stack to store parameters.

AVR Microcontroller

RISC ARCHITECTURE IN THE AVR

Features of RISC

3. **Small instruction set**. RISC processors have only basic instructions such as ADD, SUB, MUL, LOAD, STORE, AND, OR, EOR, CALL, JUMP, and so on. The limited number of instructions is one of the criticisms leveled at the RISC processor because it makes the job of Assembly language programmers much more tedious and difficult compared to CISC Assembly language programming. In the ATmega we have around 130 instructions

4. **More than 95% of instructions are executed with only one clock cycle**, in contrast to CISC instructions.

AVR Microcontroller

RISC ARCHITECTURE IN THE AVR

Features of RISC

5) **RISC processors have separate buses for data and code**. In all the x86 processors, like all other CISC computers, there is one set of buses for the address and another set of buses for data carrying opcodes and operands in and out of the CPU

   In RISC processors, there are four sets of buses:

   (1) a set of data buses for carrying data in and out of the CPU,

   (2) a set of address buses for accessing the data,

   (3) a set of buses to carry the opcodes, and

   (4) a set of address buses to access the opcodes.

   The use of separate buses for code and data operands is commonly referred to as Harvard architecture.

AVR Microcontroller

RISC ARCHITECTURE IN THE AVR

Features of RISC

6. Because CISC has such a large number of instructions, each with so many different addressing modes, microinstructions are used to implement them. The implementation of microinstructions inside the CPU employs more than 40-60% of transistors in many CISC processors. RISC instructions, however, due to the small set of instructions, are implemented using the hardwire method. Hardwiring of RISC instructions takes no more than 10% of the transistors.
7. **RISC uses load/store architecture**. In CISC, data can be manipulated while it is still in memory. For example, in instructions such as "ADD Reg, Memory", the microprocessor must bring the contents of the external memory location into the CPU, add it to the contents of the register, then move the result back to the external memory location. The problem is there might be a delay in accessing the data from external memory. Then the whole process would be stalled, preventing other instructions from proceeding in the pipeline.

Features of RISC

In concluding this discussion of RISC processors, it is interesting to note that RISC technology was explored by the scientists at IBM in the mid-1970s, but it was David Patterson of the University of California at Berkeley who in 1980 brought the merits of RISC concepts to the attention of computer scientists.

It must also be noted that in recent years CISC processors such as the Pentium have used some RISC features in their design. This was the only way they could enhance the processing power of the x86 processors and stay competitive. Of course, they had to use lots of transistors to do the job, because they had to deal with all the CISC instructions of the x86 processors and the legacy software of DOS/Windows.

AVR Microcontroller

VIEWING REGISTERS AND MEMORY WITH AVR STUDIO IDE

The AVR microcontroller has great tools and support systems, many of them free or inexpensive. AVR Studio is an assembler and simulator provided for free by Atmel Corporation and can be downloaded from the www.atmel.com website.

Many assemblers and C compilers come with a simulator. Simulators allow us to view the contents of registers and memory after executing each instruction (single-stepping). It is strongly recommended to use a simulator to single-step some of the programs in this chapter and future chapters. Single-stepping a program with a simulator gives us a deeper understanding of microcontroller architecture, in addition to the fact that we can use it to find the errors in our programs.

See the following website for a tutorial on using AVR Studio:

http://www.MicroDigitalEd.com

AVR Microcontroller

VIEWING REGISTERS AND MEMORY WITH AVR STUDIO IDE

Figures 2-21 through 2-23 show screenshots for AVR simulators from AVR Studio.

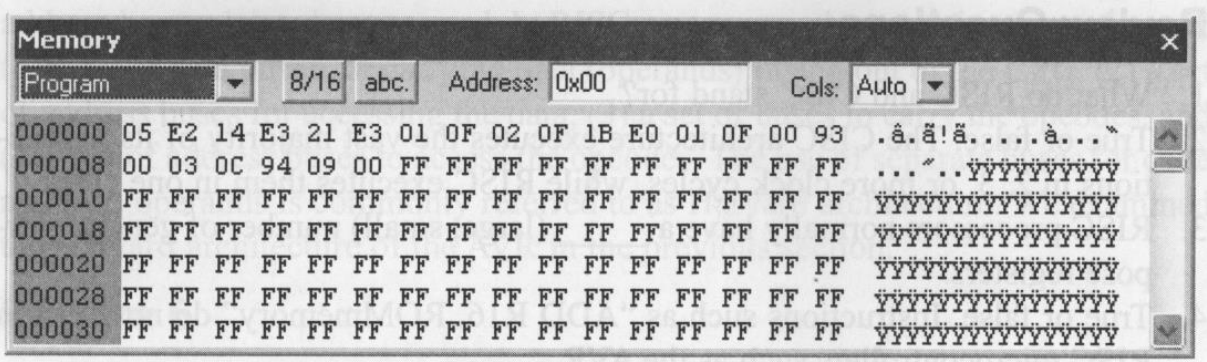


Figure 2-21. Data Memory Window in AVR Studio IDE

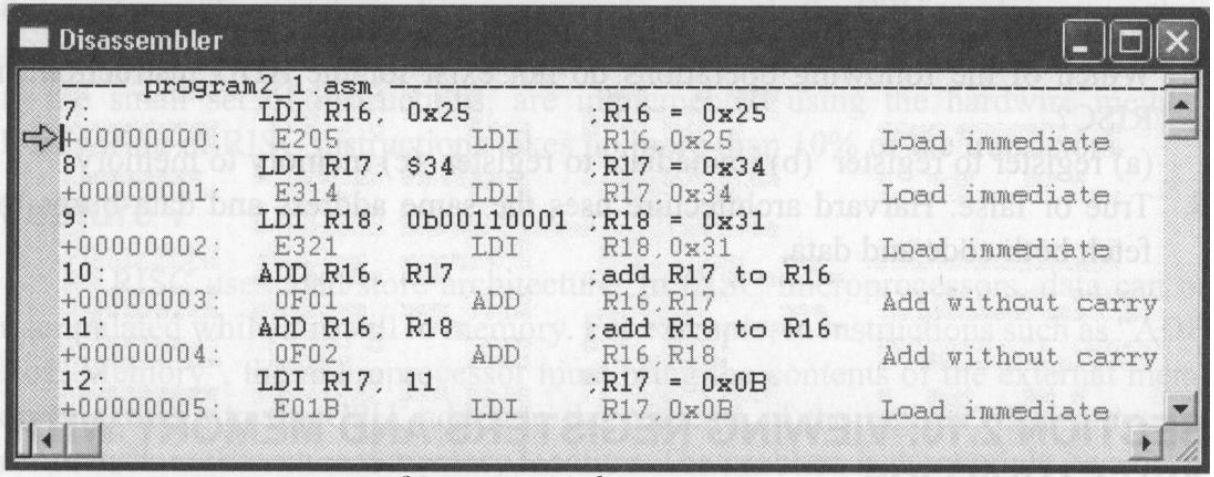
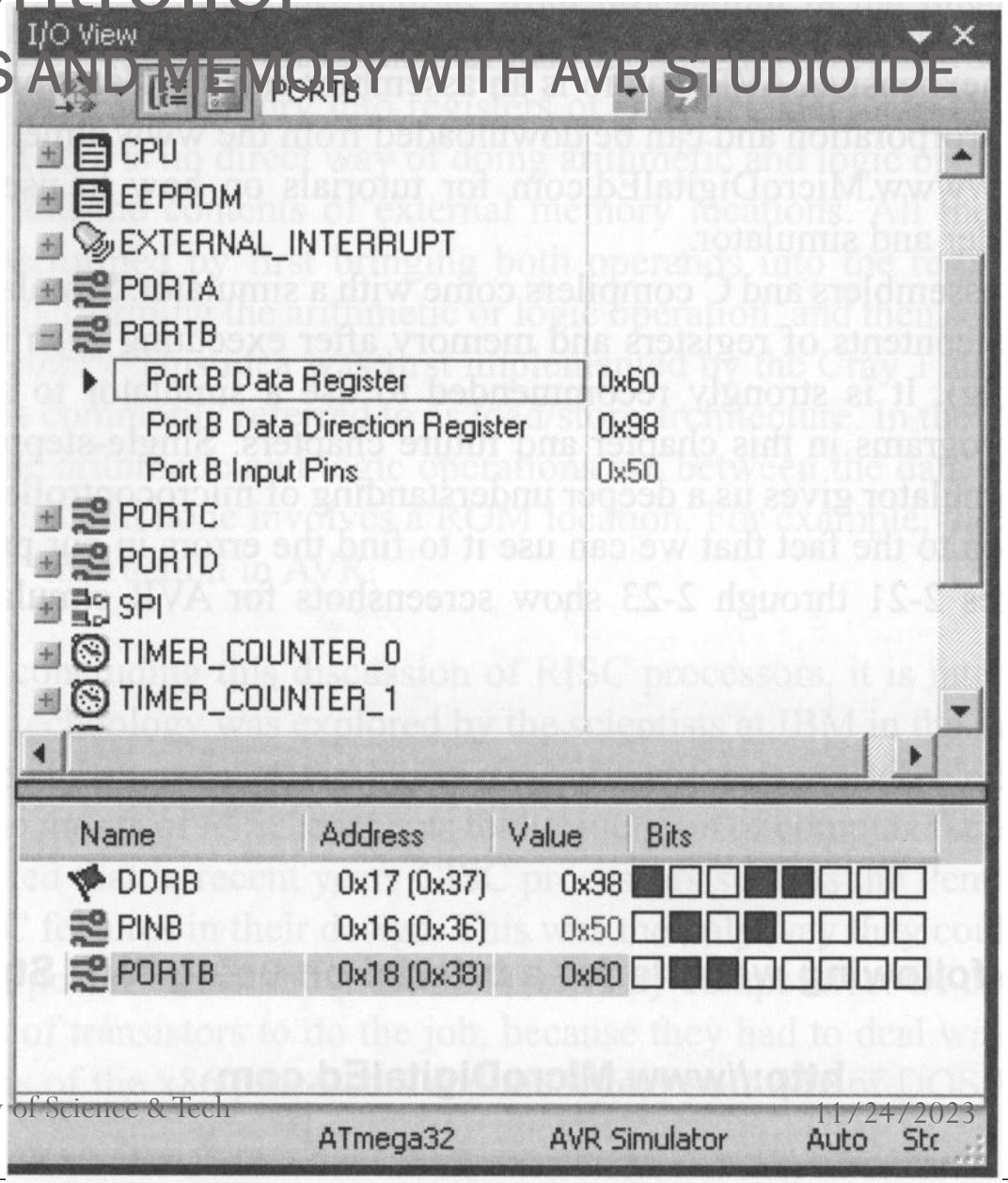


Figure 2-22. Program ROM (Disassembler) Window in AVR Studio IDE

AVR Microcontroller

VIEWING REGISTERS AND MEMORY WITH AVR STUDIO IDE

Figures 2-21 through 2-23 show screenshots for AVR simulators from AVR Studio.



mashhoun@iust.ac.ir Iran Univ of Science & Tech