EE2016 Microprocessor Theory & Lab, Aug – Nov. 2024

Week4: AVR Microcontrollers

Dr. R. Manivasakan, OSWM Lab, EED, IIT, Madras

Organization of AVR Topics

- Criteria for choosing microcontrollers
- Microcontrollers available in market
- AVR Microcontrollers – Introduction
- AVR Microcontrollers – Hardware
  - Pinout
  - Internal hardware architecture (Harvard & RISC)
  - AVR memory (Program & Data)
  - AVR – CU
  - AVR - EU
  - AVR I/O
- AVR Microcontrollers – ISA
  - Assembling an AVR program
  - AVR Data format & Directives
  - Memory Instructions
  - AVR Branching, call & timing, delay instructions
  - AVR I/O programming
  - AVR ALU instructions
  - AVR Interrupt programming

Criteria for choosing Microcontrollers

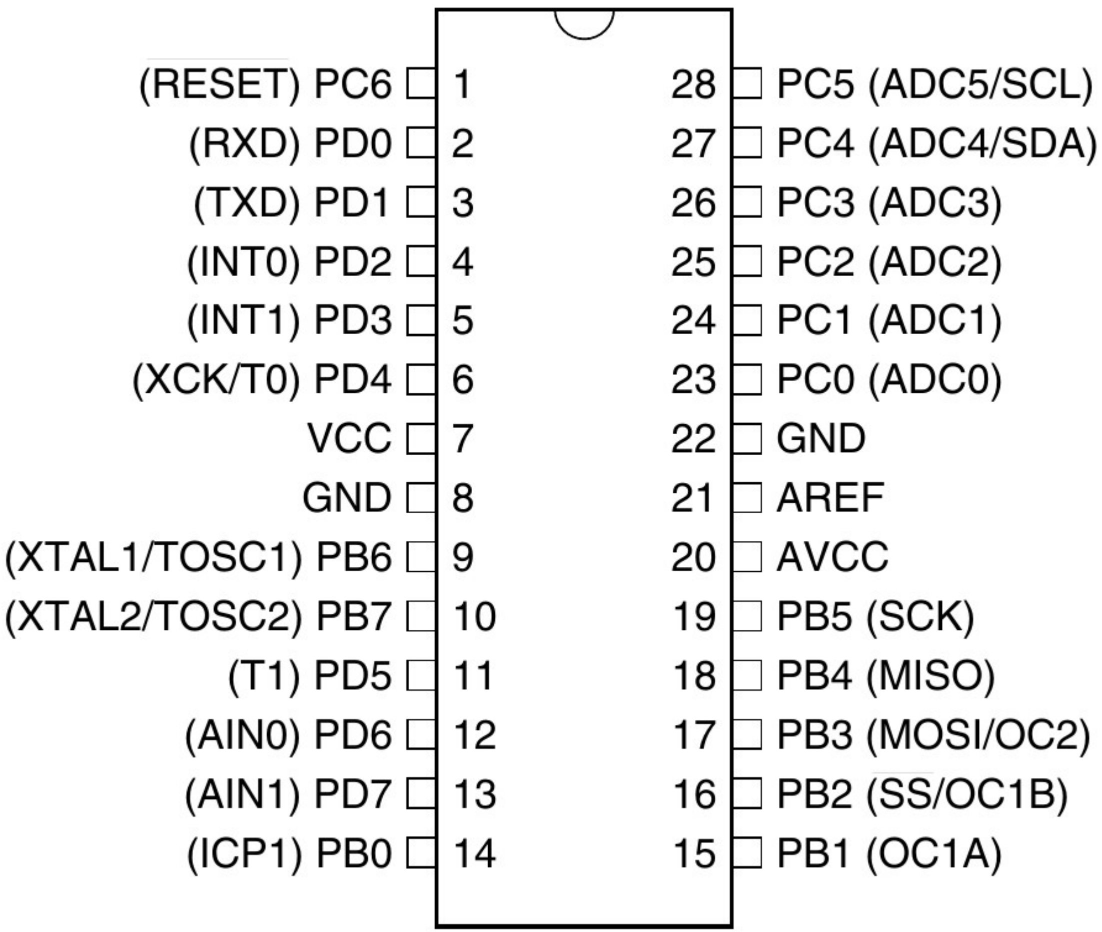
- Must meet the task at hand efficiently and cost effectively
  - Computing needs: 8-bit, 16-bit, 32-bit muC?
  - Speed – highest speed supported
  - Packaging, DIP (dual-in-line) or QFP (quad-flat-package)
    - Important in space, assembling, prototyping the end product
  - Power consumption – crictical for battery powered products
  - Amount of ROM & RAM on chip
  - Number of I/O pins & the timer on-chip
  - Ease of upgrade to higher performance or lower-power consumption versions
  - Cost per unit
- Ease of developing a product around it
- Ready availability in needed quantities

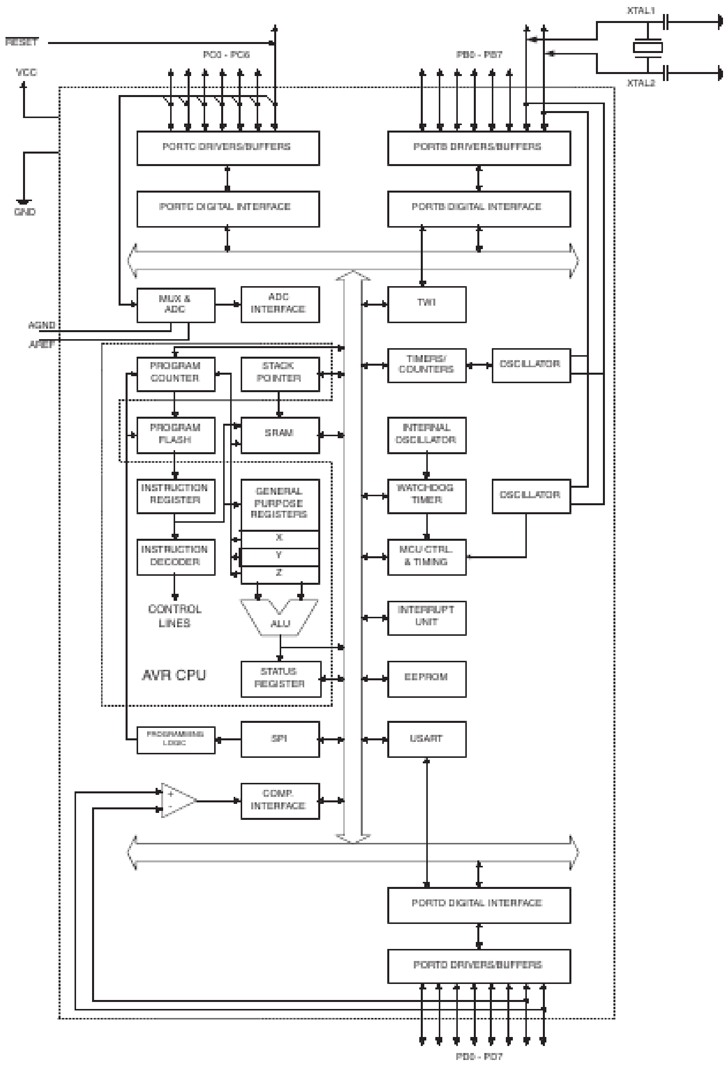
AVR Microcontrollers

- 8-bit (data bus width is 8 bits) with Harvard architecture
- Apart from AVR32, rest are 8-bit
- Not 100% compatibility within the family
  - Must recompile the codes
- Mega, tiny, classic & special purpose
- DIP
- Flash memory
- Interfaces such as USART, SPI, I2C, CAN, USB

Hardware Architecture of AVR

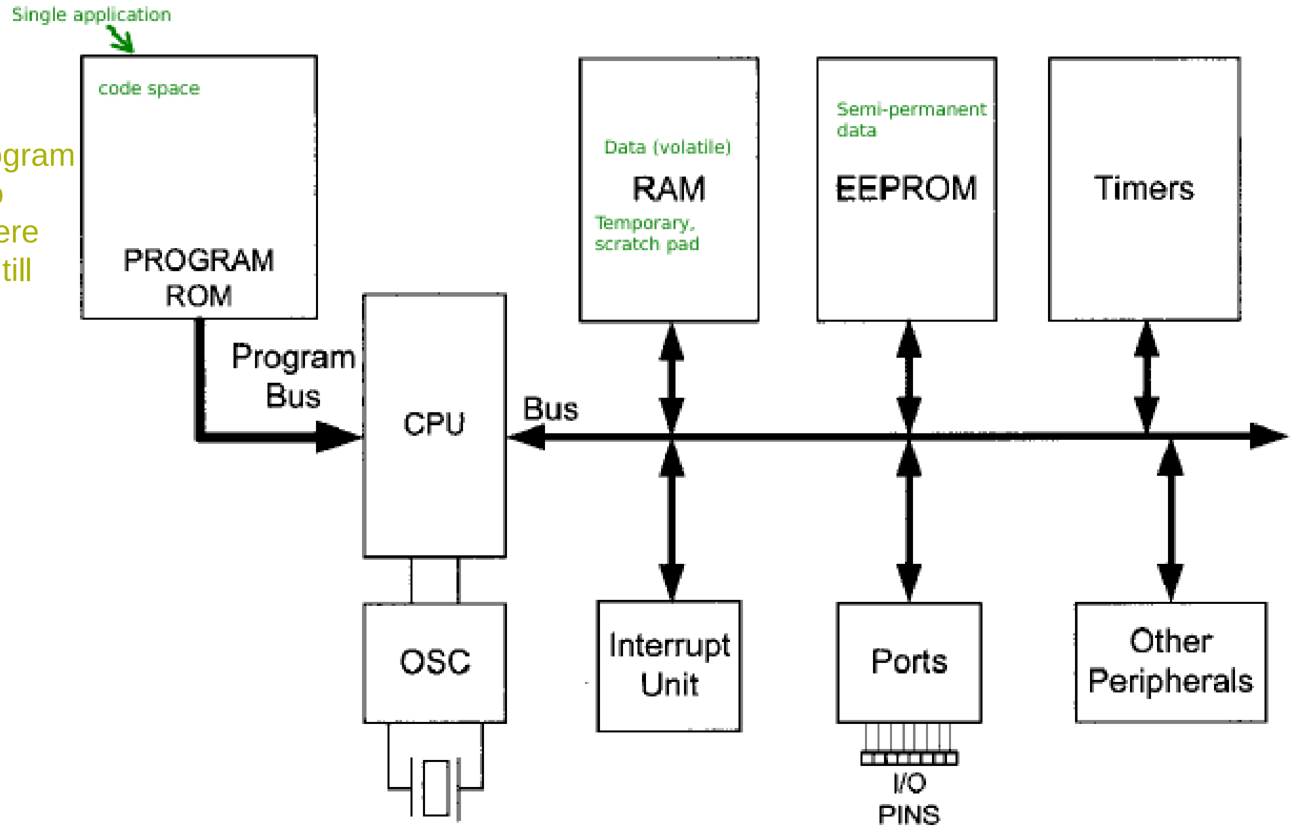
Atmega8 Pinout (Architecture: Block Diagram)





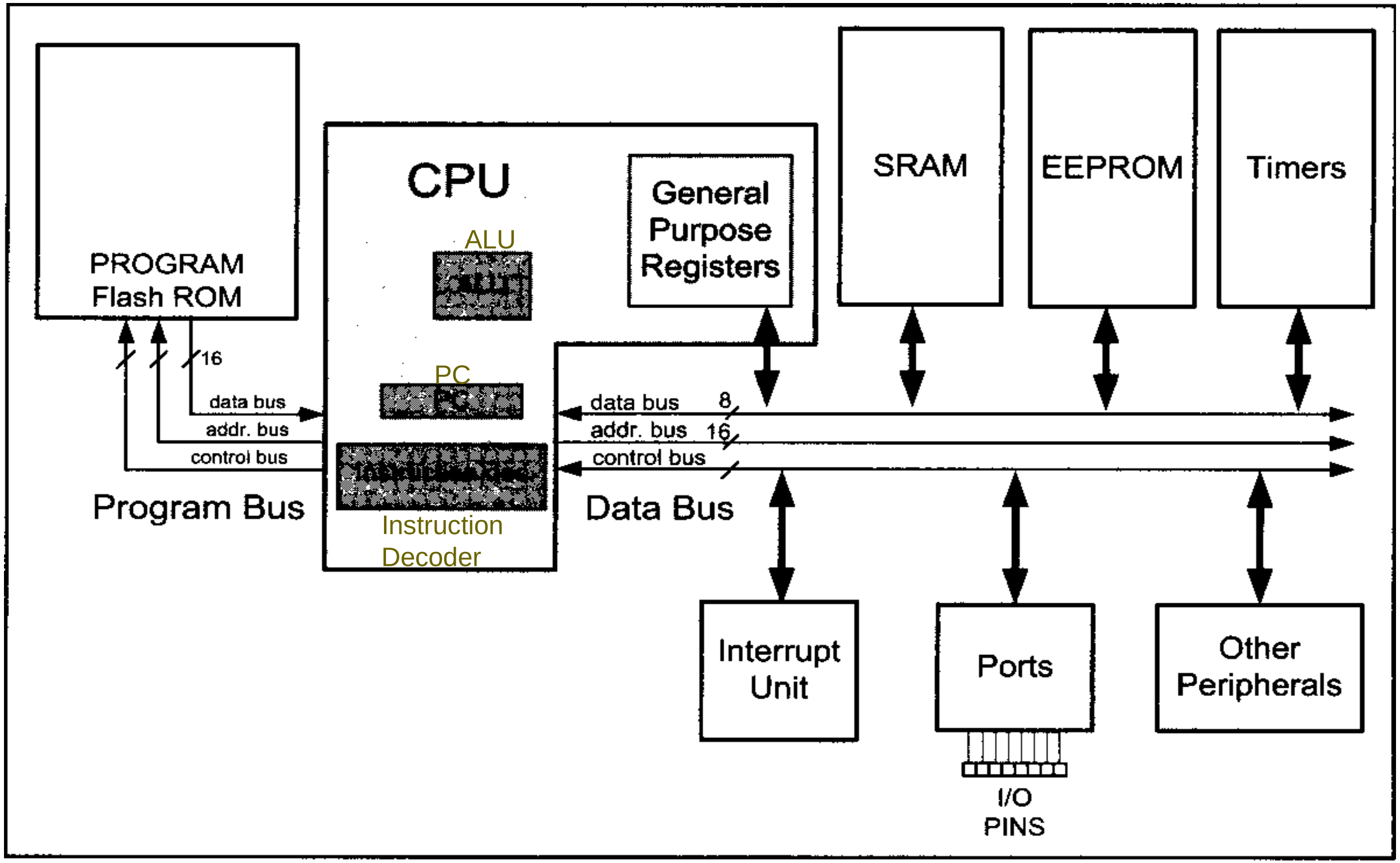
Simplified model for AVR Microcontroller

Once the program is written into ROM, it is there permanently till the life of the product



Harvard Architecture in AVR

- Little endian
  - High byte goes to high memory address while low byte goes to low memory address
- Big endian
  - Otherway round



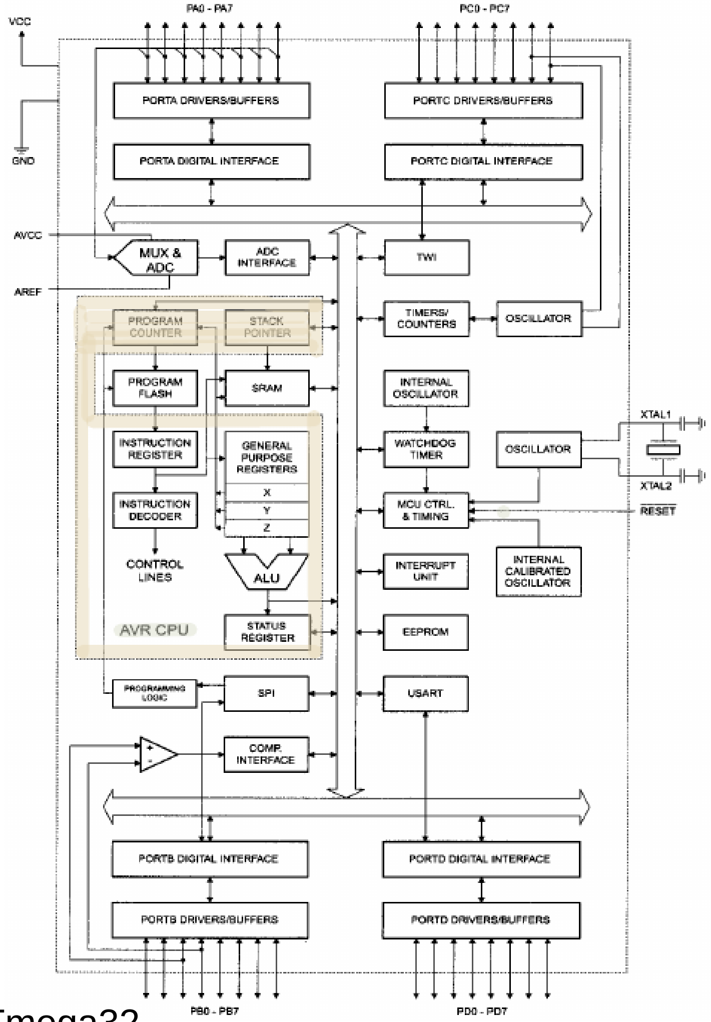
RISC Architecture in AVR

- RISC processors have a fixed instruction size
  - Fixed size instructions are easier to decode
- Large number of registers (in internal memory)
  - Speeds up the computations
- Small instruction set
  - Overlap among instructions is minimal
- More than 95% of instructions are executed with only one clock cycle
- RISC processors have separate buses for code & data
  - Improves substantially the operational speed due to space diversity and improves the security of the processor as a whole
- Atmel’s UC3 is the first floating point processing (FPU) core (2010).

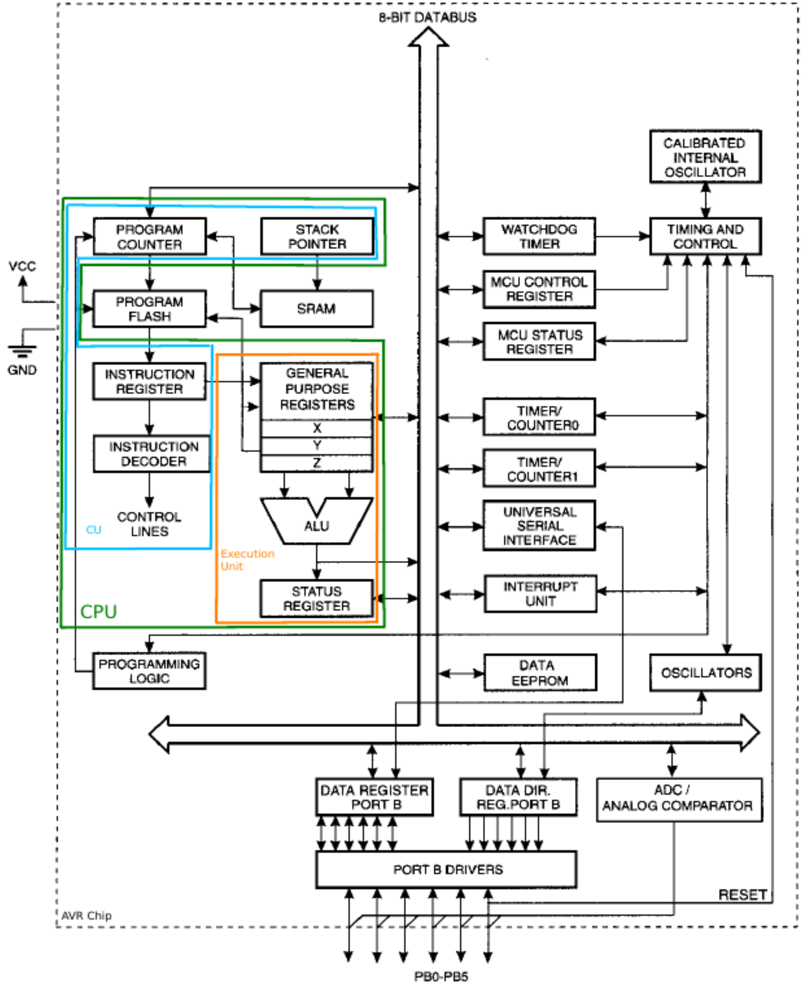
RISC Architecture in AVR

- RISC uses load / store architecture (LSA)
  - Def: An LSA is an ISA, that divides instructions into two categories: memory access (load and store between main memory and internal registers) operations and ALU operations (which only occur between registers)
  - Only LOAD and STORE instructions can access memory
  - In contrast, in CISC other instructions also can access memory
    - Eg. intel x86 (pure CISC)
    - Some architectures are hybrid: does incorporate some RISC principles, it also includes some CISC-like (Complex Instruction Set Computer) features
    - Eg PowerPC and SPARC
  - Eg. for RISC based muC / muP are RISC-V, ARM and MIPS
  - LSA differs from a register-memory architecture (RMA) (for example, a CISC ISA such as x86) in which one of the operands for the ADD operation may be in main memory, while the other is in a (an internal) register or both in MM.
  - Two concepts come into play here, in RISC with LSA: (Joe Zbiciak, quora.com)
  - **Orthogonality:** Orthogonality means you can combine operations together with minimal restrictions. For example, suppose bucket A contains “operand addressing modes” and bucket B contains “mathematical operations”. A fully-orthogonal architecture would let you combine any operand addressing mode from bucket A with any mathematical operation from bucket B.
  - **Minimalism:** Break down a given more common operation (while implementing an engineering task), into one or more fundamental operations (which cant be further broken up). Consider set ‘M’ of all possible such fundamental operations. If a given element (fundamental) operation could be realized by any other in M, then throw it out of the set. The set ‘S’, so obtained finally would hold minimal number of fundamental operations. This set is our RISC ISA.
    - Now, any given engineering task could be realised in terms of minimal number of such fundamental operations.
    - The fundamental operation should be simple enough to only require 1 execute cycle in the pipeline.
    - If you need to add two numbers, if at all possible you’d like to limit yourself to the fundamental ways in which you can add two numbers, without considering where the operands come from.

AVR Microcontrollers

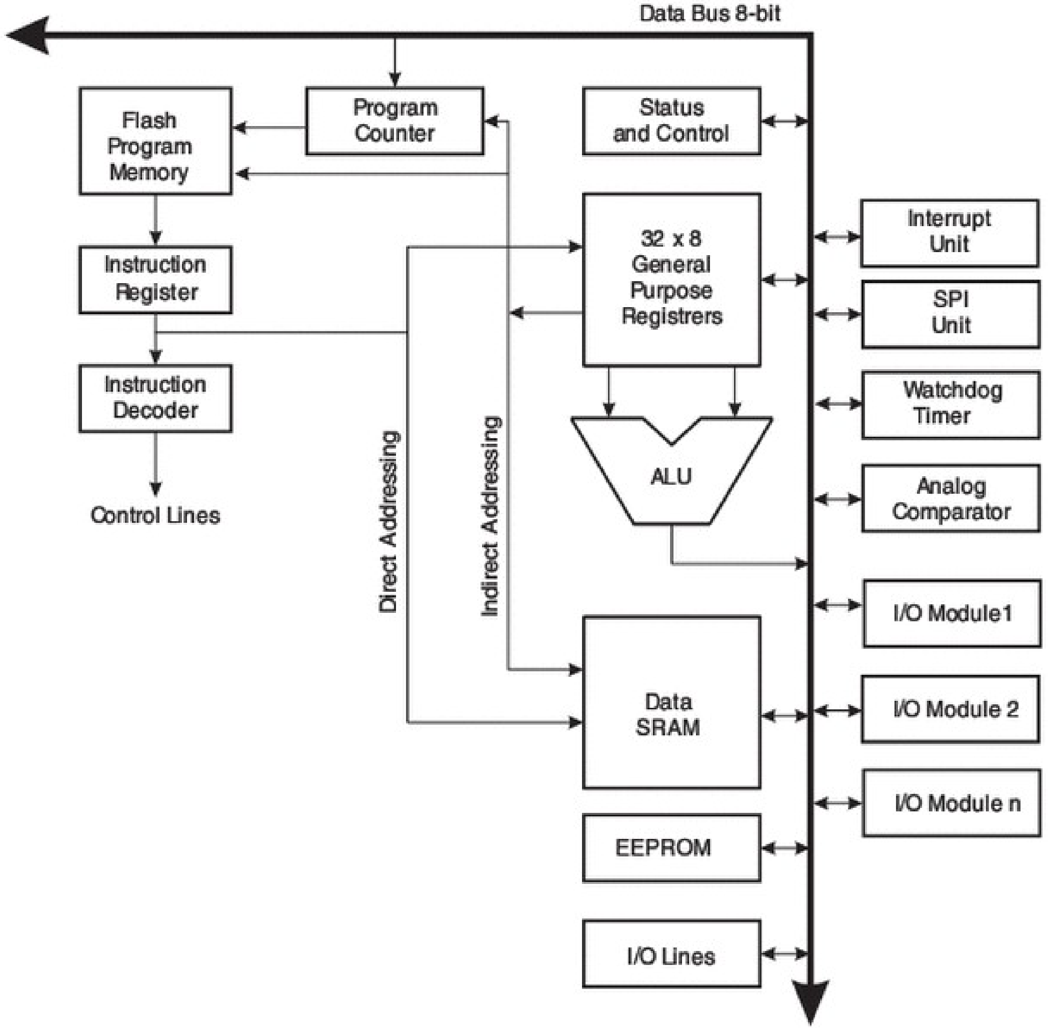


ATmega32

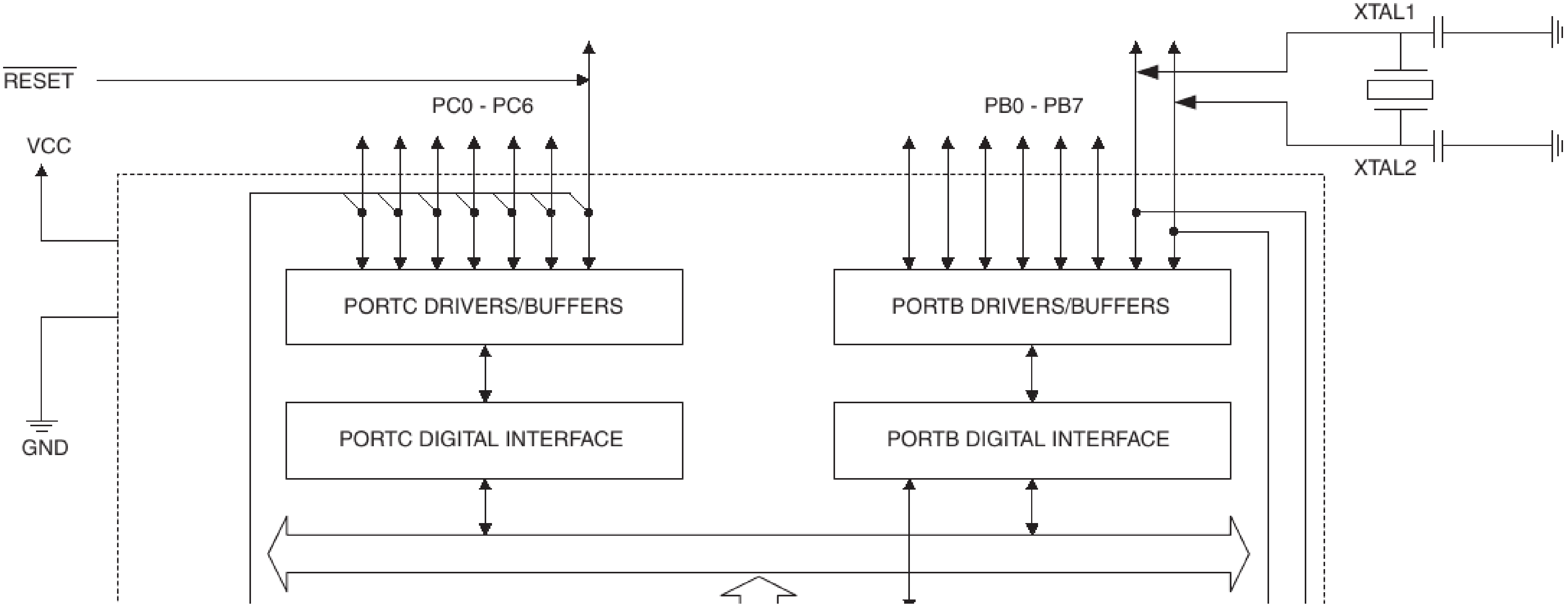


ATiny25

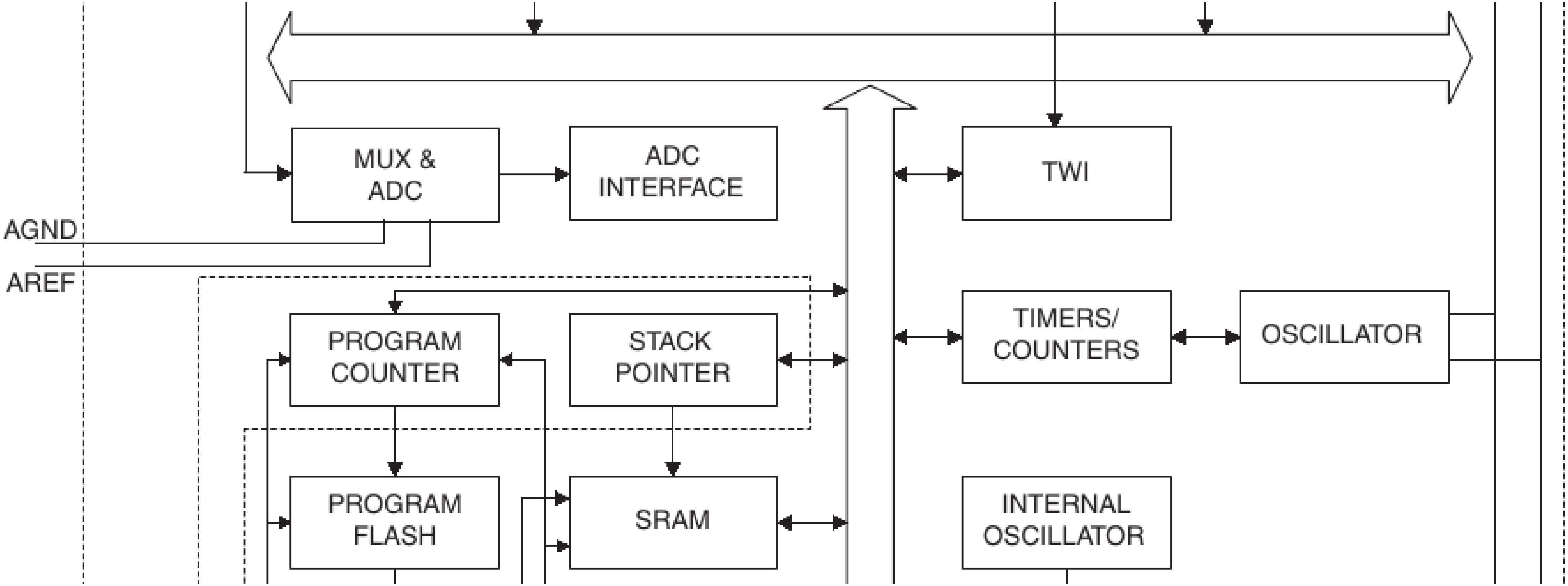
Atmega32A Architecture: Block Diagram



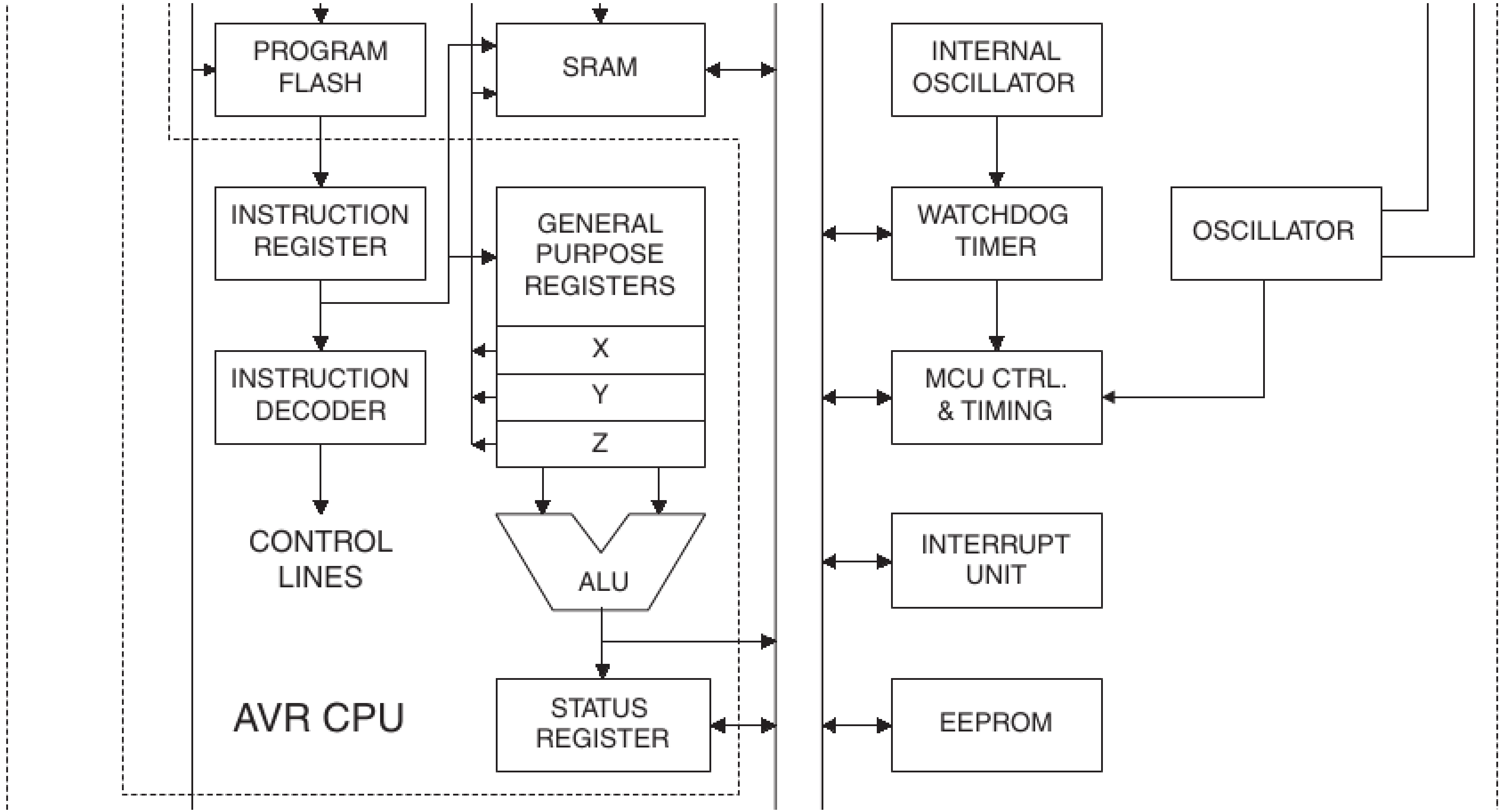
Atmega8L Architecture: Block Diagram



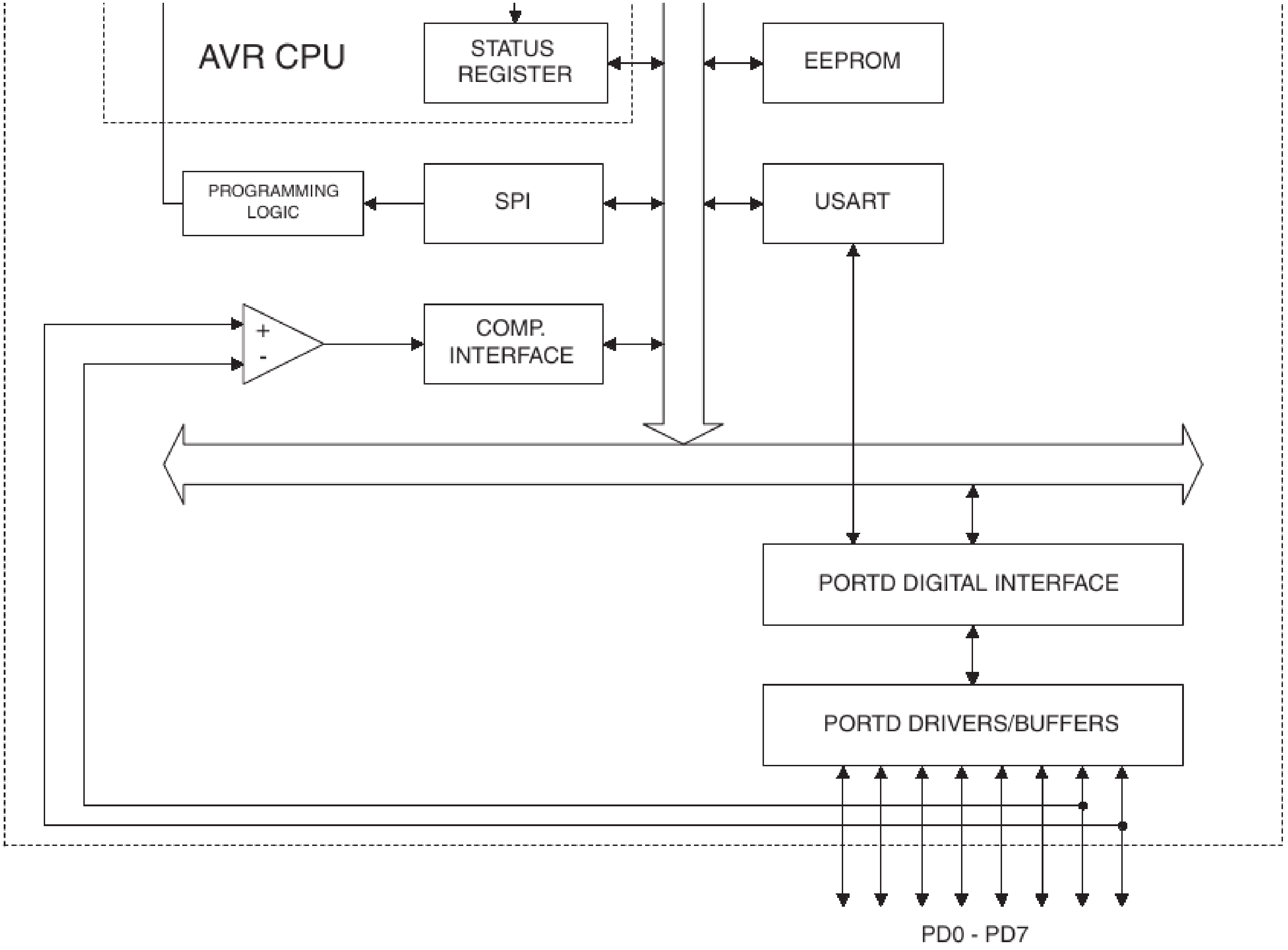
Atmega8L Architecture: Block Diagram



Atmega8L Architecture: Block Diagram



Atmega8L Architecture: Block Diagram



Semiconductor Memory & its Technology

- Semiconductor Memory
  - Capacity, organization, access time (speed) and cost.
- CPU needs memory
  - (a) fixed & permanent
  - (b) temporary information that can change w.r.t time eg: OS, application packages etc
  - Fixed or permanent – ROM
  - Temporary – RAM
  - ROM & RAM faster – Primary
  - HDD slower - Secondary
- Read Only Memory (ROM)
  - Non-Volatile
  - Mask ROM
    - Contents are programmed by the manufacturer
    - Not user programmable
      - Programmable – user cant burn information

- Read Only Memory (ROM) - Contd
  - Programmable ROM
    - User can program it
    - One time programmable
  - Erasable Programmable ROM
    - Erasable (1000 times?), but takes 20 minutes
    - All contents are erased
    - Take out the chip from system board, place it on EPROM erasure equipment, erase, reprogram and put it back into the chip
  - Electronically Erasable PROM (EEPROM)
    - Selectively a byte can be erased in EEPROM in less than a sec
    - Cost per bit in EEPROM is higher as compared to EPROM
  - Flash EPROM
    - Erasable entire content in less than a sec
    - Blocks in flash, in which one can erase selectively a block

Memory Technology

- Random Access Memory (RAM)
  - Volatile
  - Static RAM (SRAM)
    - Flip-flops --> volatile
    - Dont require refreshing
  - Non-Volatile RAM (NV RAM)
    - Non-Volatile
    - Anatomy (Hardware)
      - Extremely power efficient SRAM cells built of CMOS
      - Uses an internal lithium battery as a backup energy source
      - Uses an intelligent control circuitry to switch the backup source when the external power supply fails.

- Random Access Memory (RAM)
  - Dynamic Random Access Memory (DRAM)
    - Capacitors to store bits
    - need to be refreshed (due to leakage)
    - Advantages
      - High density
      - Cheaper cost per bit
      - Lower power consumption per bit
    - Disadvantages
      - Refreshed periodically
      - Cant be accessed during refreshing
      - DRAM are not preferred for real-time and low latency applications

Memory in AVR Family of Microcontrollers

- AVR Microcontroller ROM
  - Holds program code
  - On-chip flash memory (1st MuC to have!)
    - 8 MB memory size
- Random Access Memory (RAM)
  - For storing data, max 64 KB
  - 3 components
    - 32 general purpose registers,
    - I/O memory
    - internal SRAM

Atmega Memory

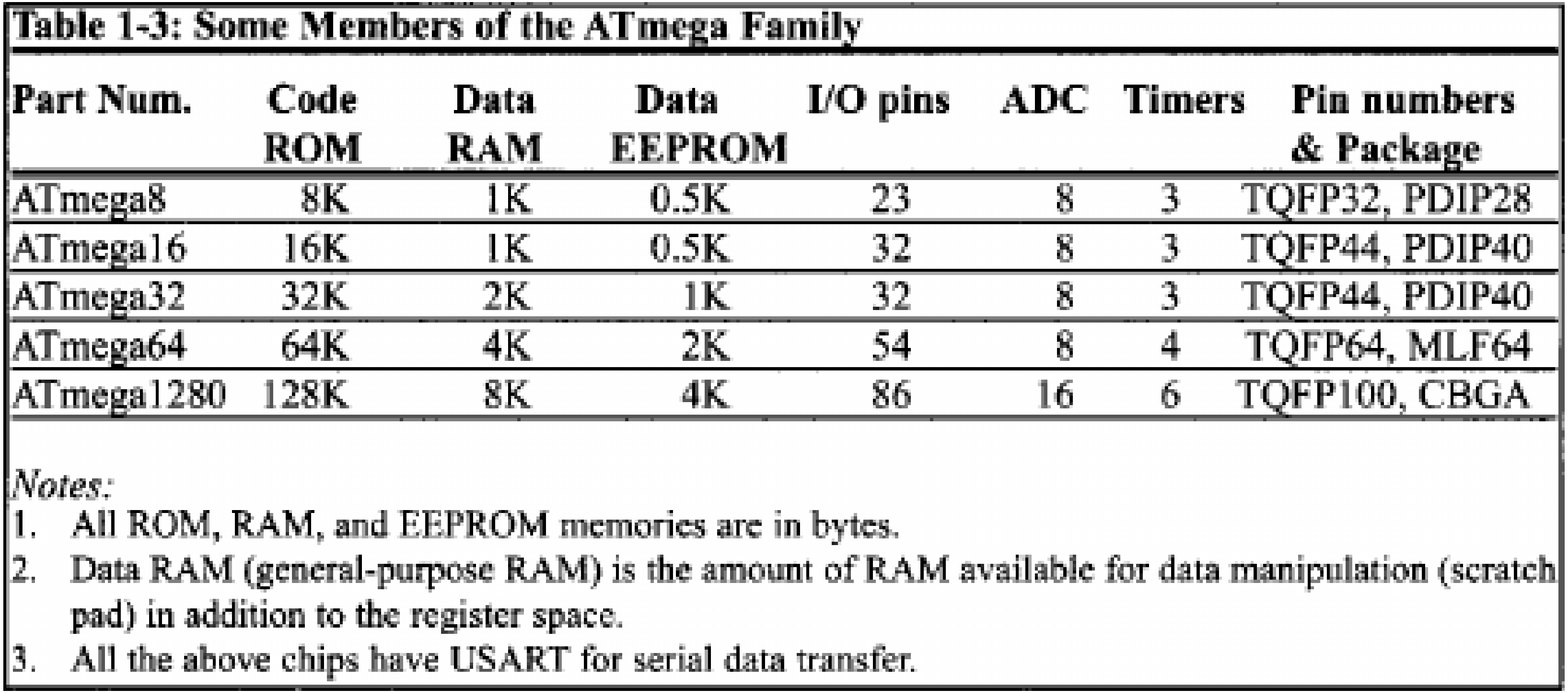


Table 1-3: Some Members of the ATmega Family

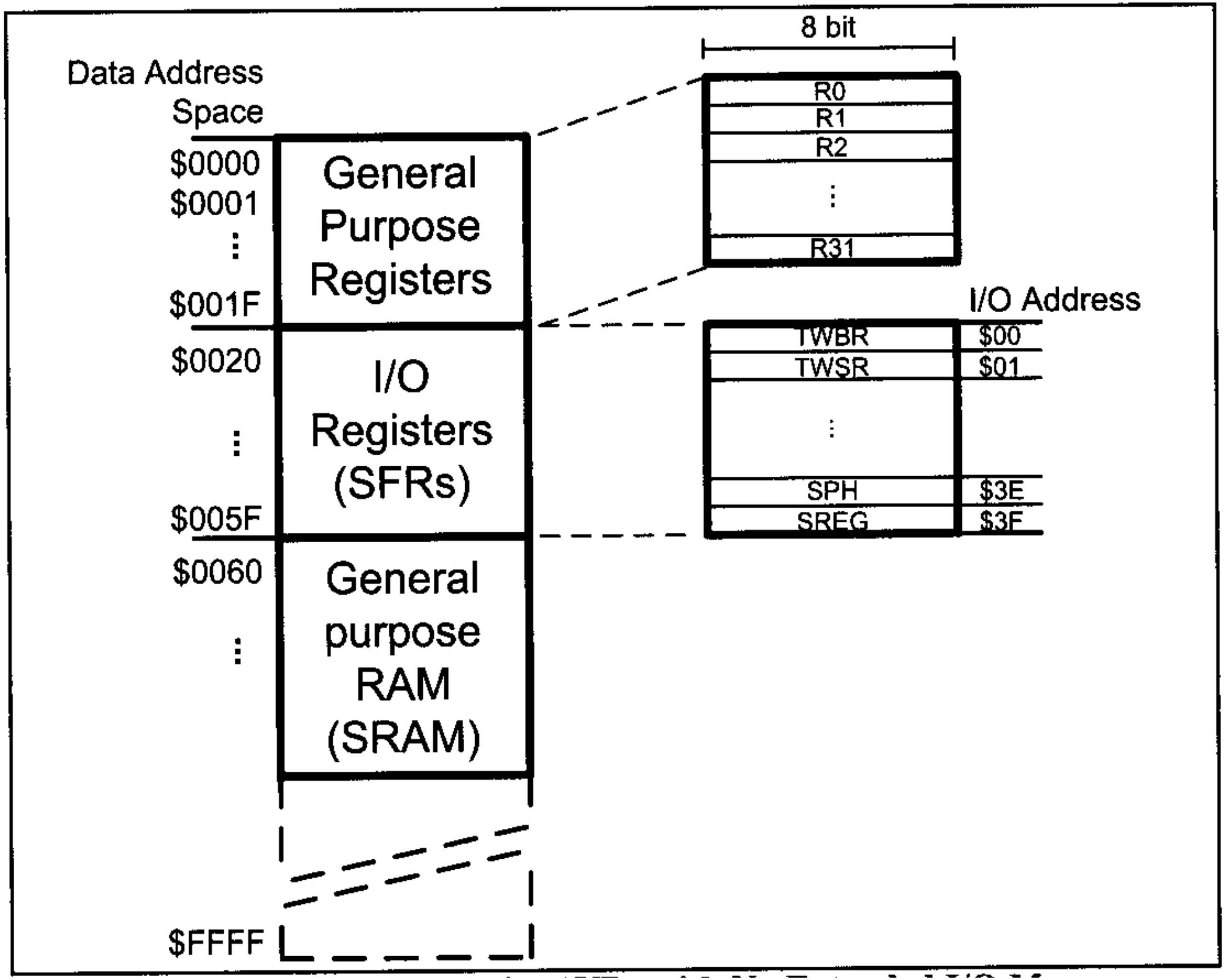
Part Num.	Code ROM	Data RAM	Data EEPROM	I/O pins	ADC	Timers	Pin numbers & Package
ATmega8	8K	1K	0.5K	23	8	3	TQFP32, PDIP28
ATmega16	16K	1K	0.5K	32	8	3	TQFP44, PDIP40
ATmega32	32K	2K	1K	32	8	3	TQFP44, PDIP40
ATmega64	64K	4K	2K	54	8	4	TQFP64, MLF64
ATmega1280	128K	8K	4K	86	16	6	TQFP100, CBGA

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.

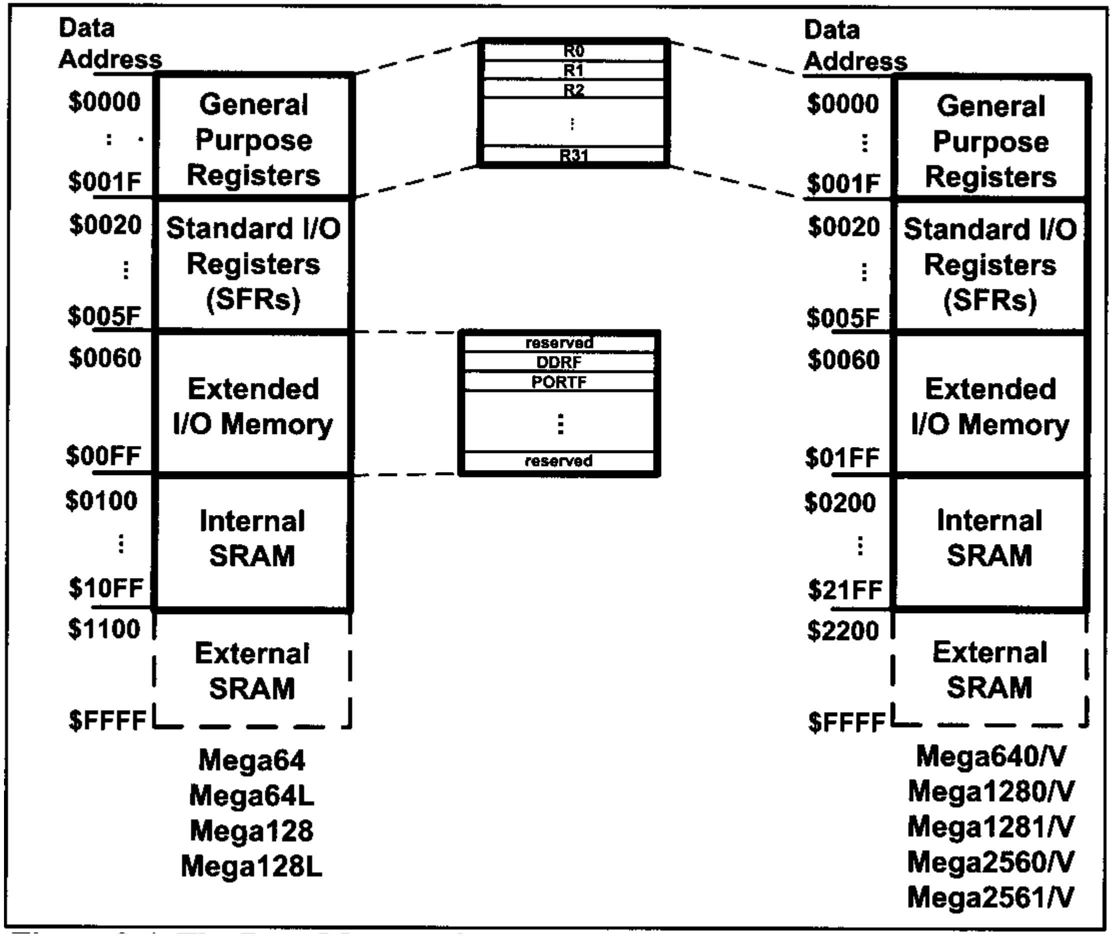
Data Memory in AVRs

- Data Memory
  - GPRs
  - I/O memory
  - Internal data SRAM
- GPRs
  - 32 8-bit registers
  - Address: $0000 - $001F
- Internal SRAM
  - SRAM is expensive than DRAM & is faster
  - SRAM used typically in cache
- I/O memory - Special Function Registers (SFRs)
  - Status registers
  - Timers
  - I/O ports
  - Serial communication ports
  - ADC



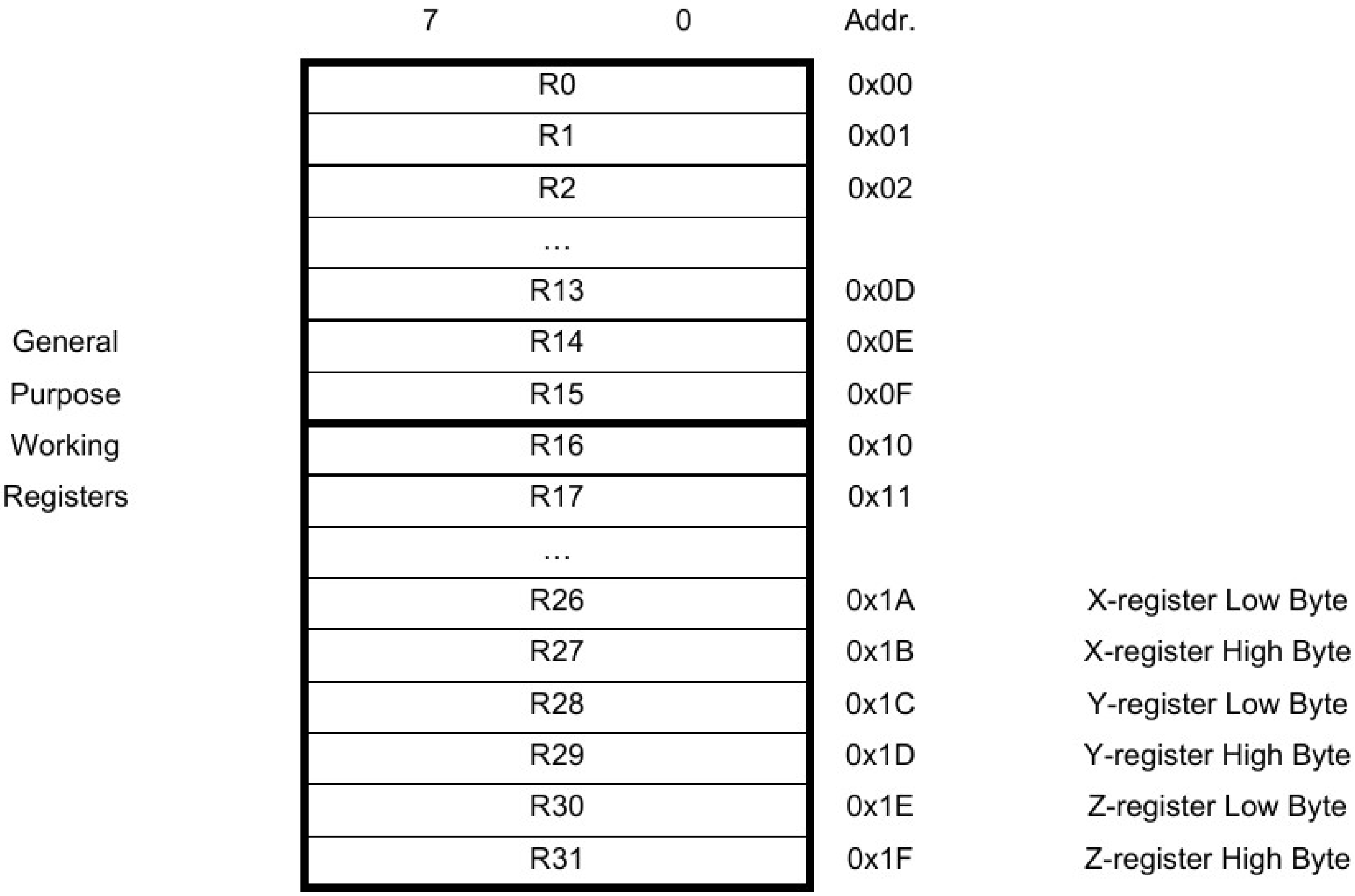
Data Memory for AVRs

- Standard I/O Registers
  - 8-bit registers – 64 in number
  - Total memory, 64 bytes
  - Also, called Special Function Registers (SFRs)
    - I/O port registers (Port A, Port B etc), interrupt I/O pins, bitrate registers, communication status registers, timer, ALU status register etc
- Internal data SRAM
  - Scratch pad (cache)
  - Used to store data brought into CPU via I/O & serial ports
  - Data bus 8-bits wide
  - GPRs, SFRs & internal SRAM are made of SRAM
- EEPROM in AVR
  - Semi permanent memory eg. “Check engine” signal in ECU
  - Unless cleared using OBD interface



GPRs & Pointer Registers

- General Purpose Registers (GPRs)
  - 8-bit registers
  - R0 – R31 (total of 32 registers)
  - Pointer Registers
    - R26 – R31 generally used as pointers
    - R27,R26 pair holds 16 bit address of a memory location in main memory (which could address the memory block of size 2^16 X word size)
    - Ditto for R29,R28 and R31,R30 pairs

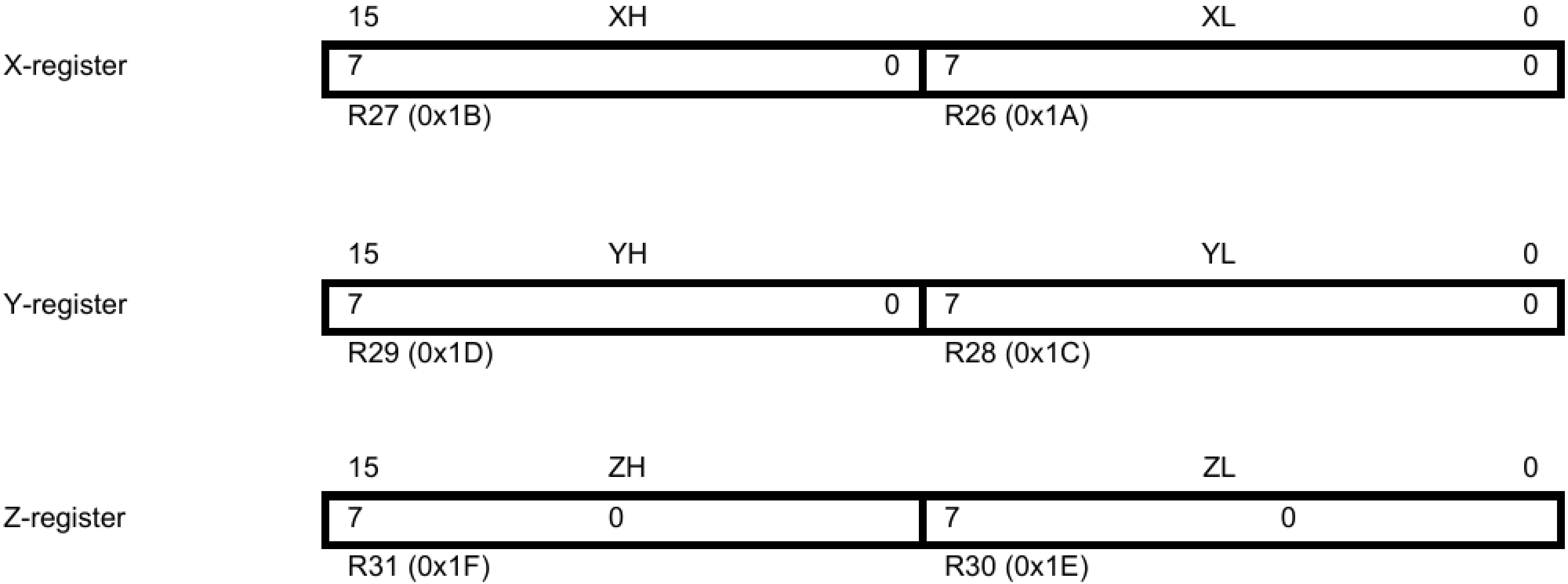


General Purpose Working Registers

7 0	Addr.	
R0	0x00	
R1	0x01	
R2	0x02	
…		
R13	0x0D	
R14	0x0E	
R15	0x0F	
R16	0x10	
R17	0x11	
…		
R26	0x1A	X-register Low Byte
R27	0x1B	X-register High Byte
R28	0x1C	Y-register Low Byte
R29	0x1D	Y-register High Byte
R30	0x1E	Z-register Low Byte
R31	0x1F	Z-register High Byte

Pointer Registers: X, Y & Z

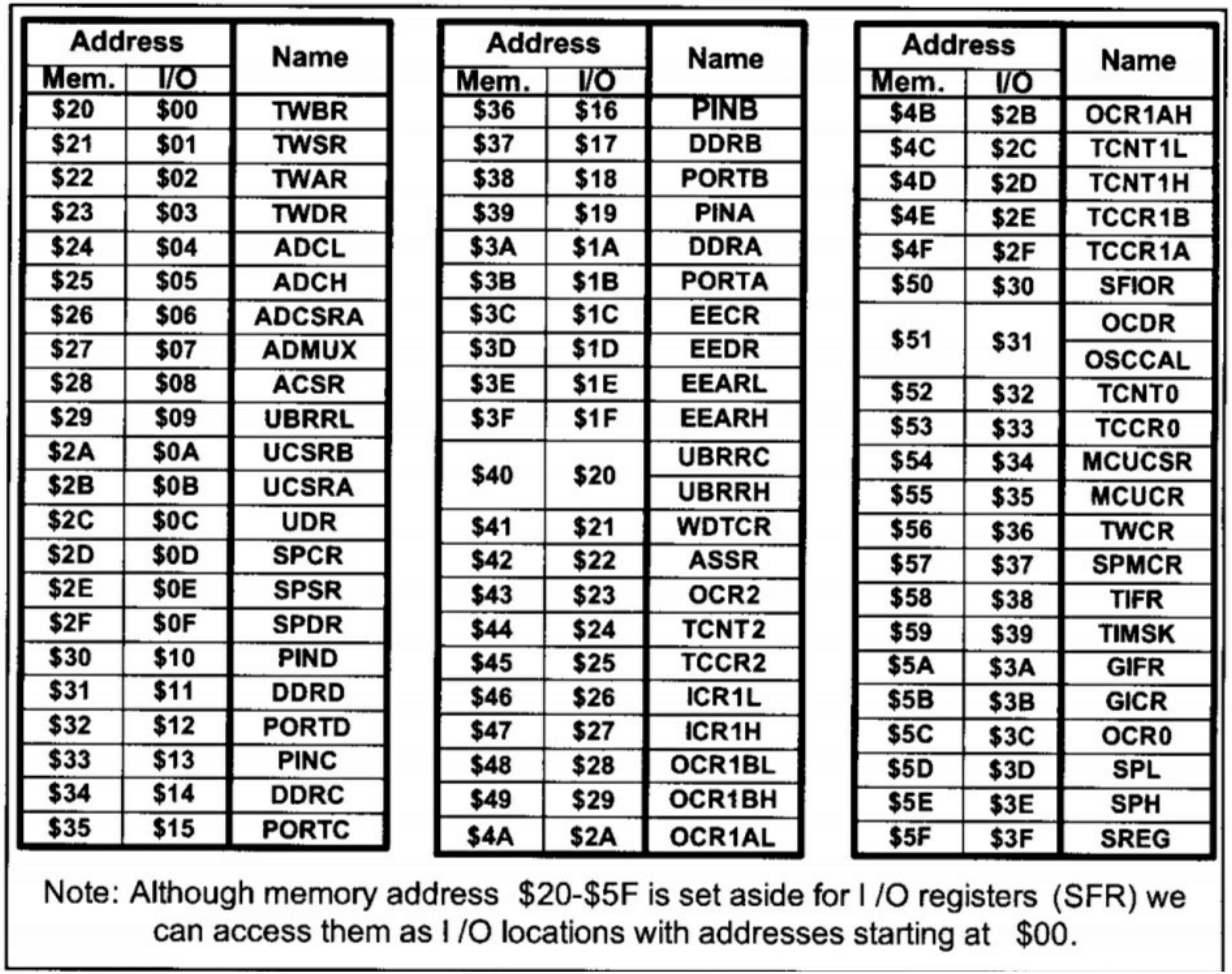
Figure 4. The X-register, Y-register and Z-Register



	15	XH	XL	0
X-register	7	0	7	0
	R27 (0x1B)		R26 (0x1A)	
	15	YH	YL	0
Y-register	7	0	7	0
	R29 (0x1D)		R28 (0x1C)	
	15	ZH	ZL	0
Z-register	7	0	7	0
	R31 (0x1F)		R30 (0x1E)	

I/O Registers

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

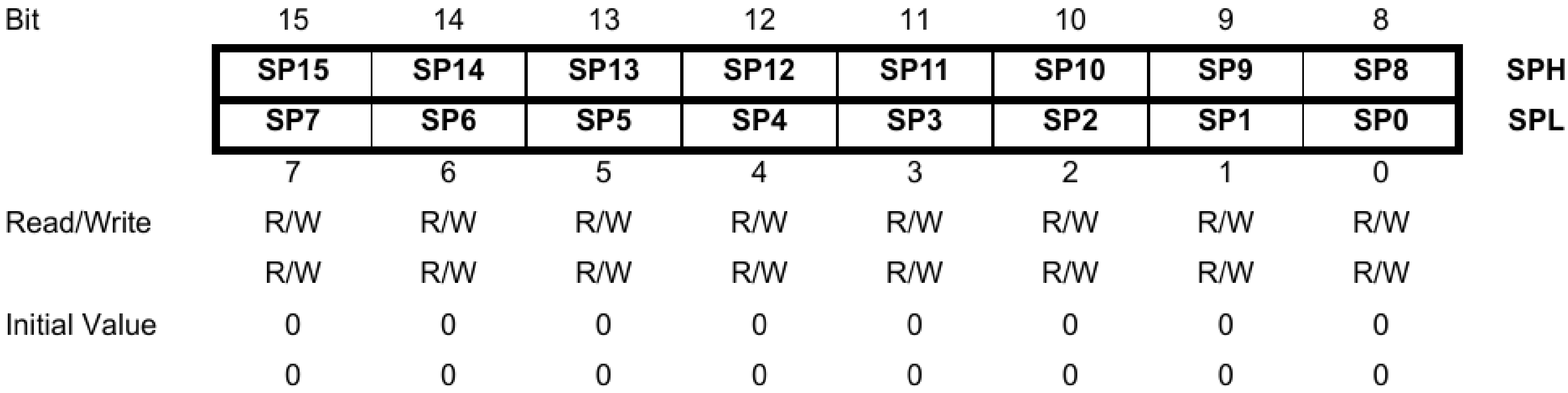
Address		Name
Mem.	I/O	
$36	$16	PINB
$37	$17	DDRB
$38	$18	PORTB
$39	$19	PINA
$3A	$1A	DDRA
$3B	$1B	PORTA
$3C	$1C	EECR
$3D	$1D	EEDR
$3E	$1E	EEARL
$3F	$1F	EEARH
$40	$20	UBRRC UBRRH
$41	$21	WDTCR
$42	$22	ASSR
$43	$23	OCR2
$44	$24	TCNT2
$45	$25	TCCR2
$46	$26	ICR1L
$47	$27	ICR1H
$48	$28	OCR1BL
$49	$29	OCR1BH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

Stack Pointer – Registers in Control Unit

- Stack Pointer is a 16-bit register in the I/O registers area (internal memory) with SPH in address 0x5D and SPL in address 0x5E
- Stack is used to point to the instruction to be executed next, before the PC jumps to the ISR during interrupt or the subroutine call.

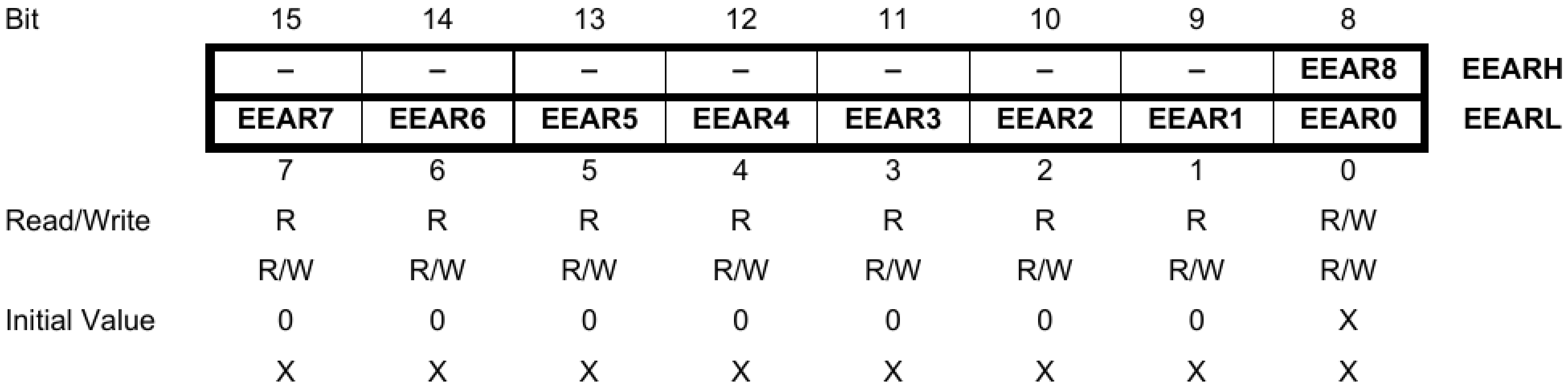


Bit	15	14	13	12	11	10	9	8	
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

I/O Registers

- EEPROM – Electrically eraesable programmable ROM

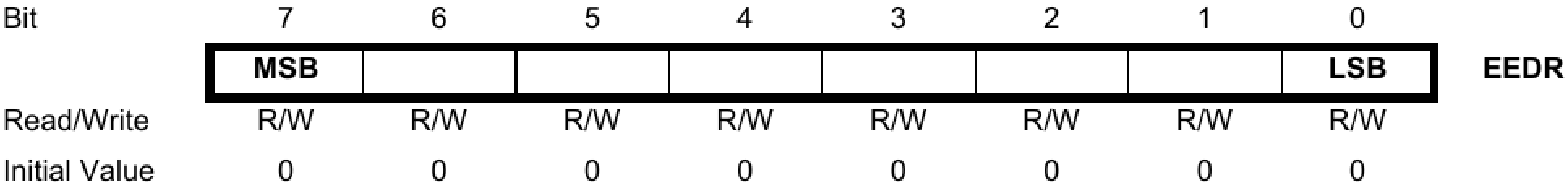
- EEPROM Address Register



Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	–	EEAR8	EEARH
	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0	EEARL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	X	
	X	X	X	X	X	X	X	X	

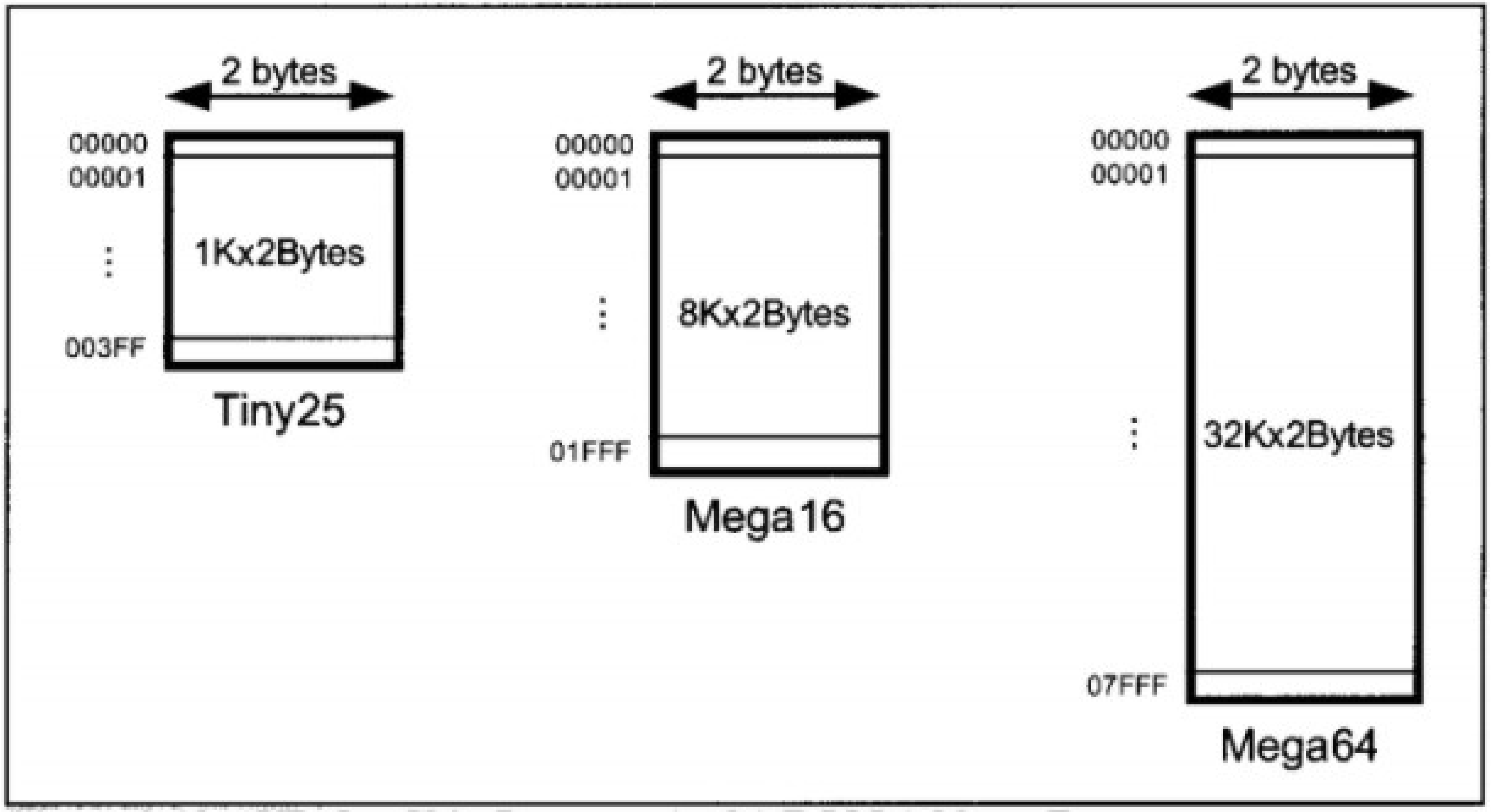
I/O Registers

- EEPROM – Electrically eraesable programmable ROM
- EEPROM Address Register



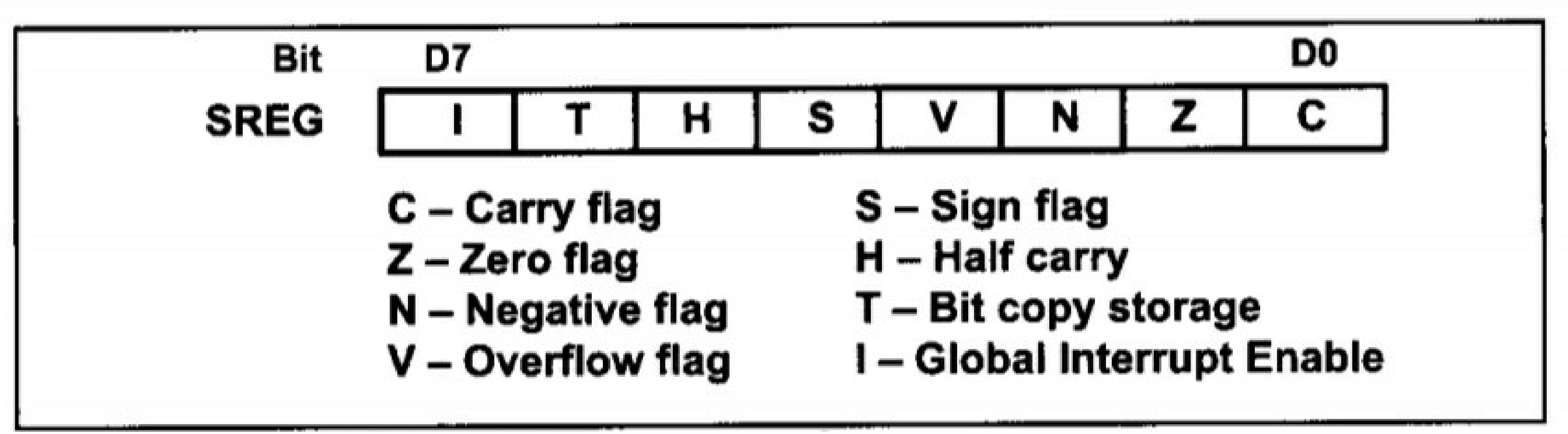
Bit	7	6	5	4	3	2	1	0	
	MSB							**LSB**	**EEDR**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

AVR on-chip Program code ROM Address Range



AVR ALU Status Register

- Status Register (SREG) of Flag Register
  - Conditional flags C, Z, N, V, S, H, T and I
    - Indicate some conditions which could be used in conditional branch instructions.
  - S, the sign bit
    - Exclusive-OR-ing of N & V flags
  - H, half carry flag. It indicates when a carry or borrow has been generated out of the least significant four bits of the accumulator register following the execution of an arithmetic instruction. It is primarily used in decimal (BCD) arithmetic instructions.

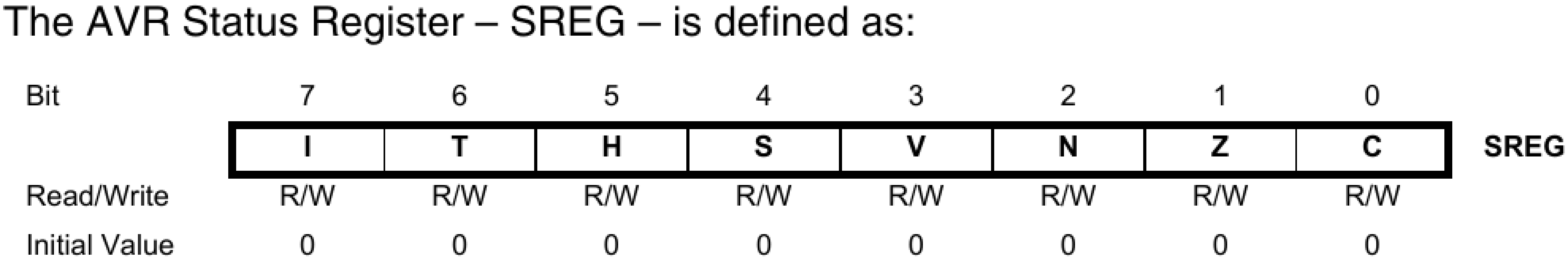


Bit	D7							D0
SREG	I	T	H	S	V	N	Z	C

C – Carry flag
Z – Zero flag
N – Negative flag
V – Overflow flag

S – Sign flag
H – Half carry
T – Bit copy storage
I – Global Interrupt Enable

Atmega8L AVR Status Register (SREG)



The AVR Status Register – SREG – is defined as:

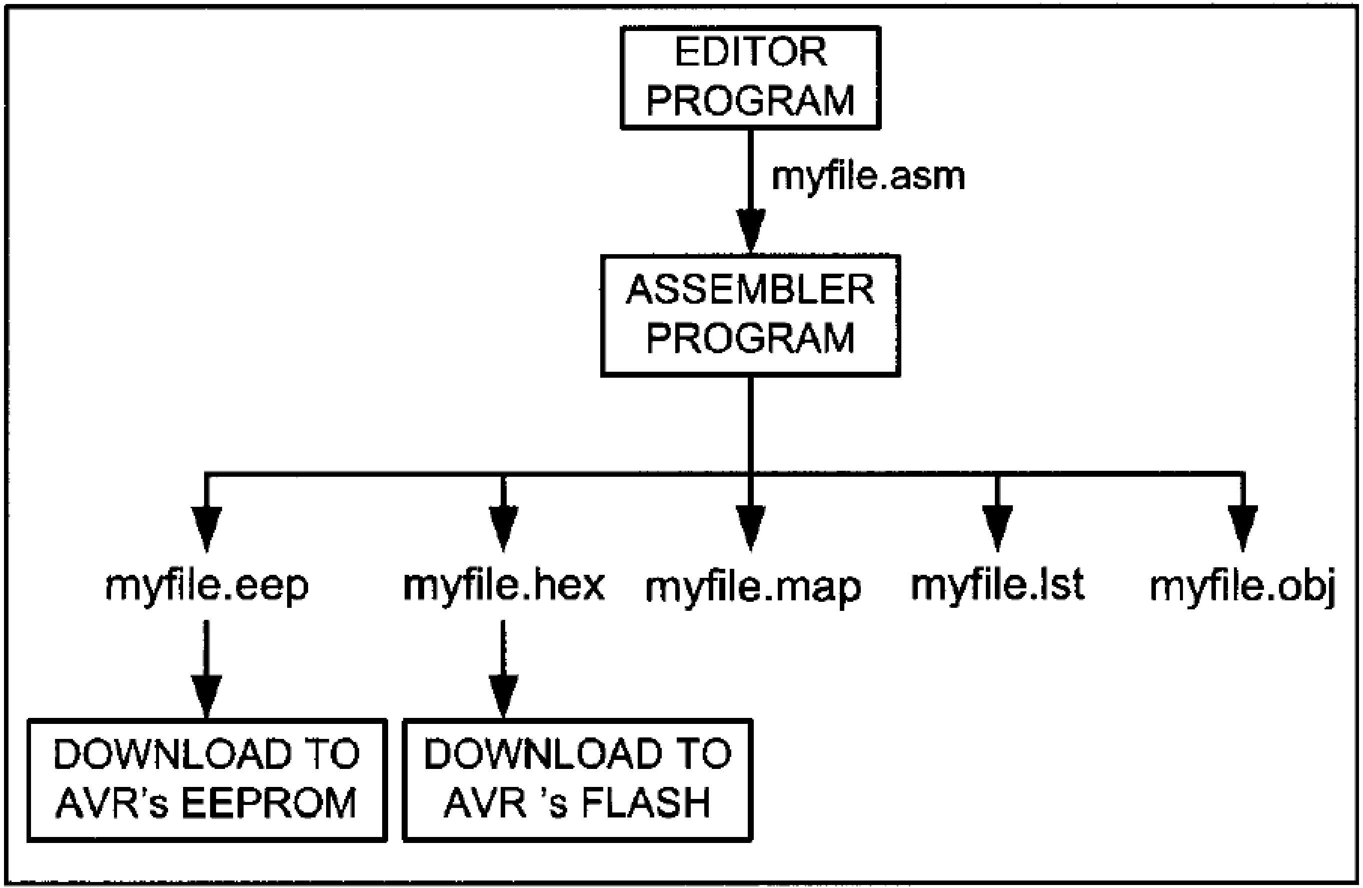
Bit	7	6	5	4	3	2	1	0	
	I	**T**	**H**	**S**	**V**	**N**	**Z**	**C**	**SREG**
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**
- **Bit 6 – T: Bit Copy Storage**
- **Bit 5 – H: Half Carry Flag**
- **Bit 4 – S: Sign Bit, S = N ⊕ V**
- **Bit 3 – V: Two’s Complement Overflow Flag**
- **Bit 2 – N: Negative Flag**
- **Bit 1 – Z: Zero Flag**
- **Bit 0 – C: Carry Flag**

- AVR ISA Architecture & Assembly Language Programming

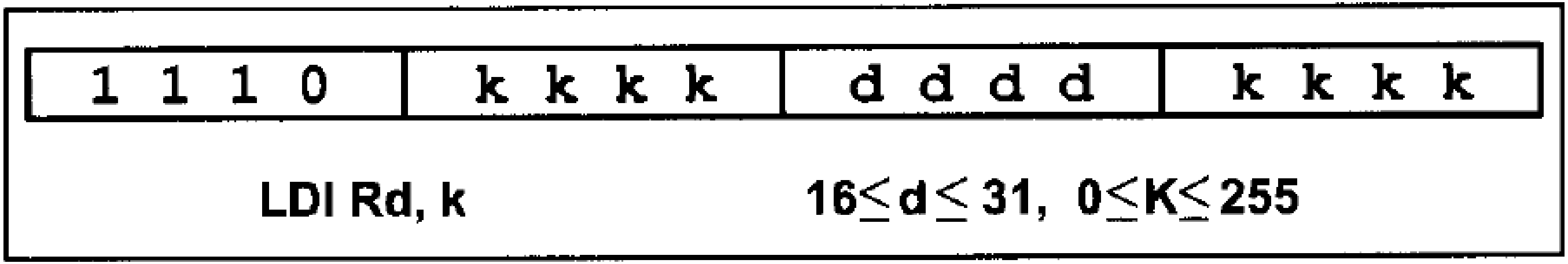
ISA and Assembly Programming

- Instruction Set Architecture (ISA)
  - Instructions
- Assembly Programming
  - Program consists of instructions
- Software Development
  - Edit the text file of assembler program
  - Compile it
  - Burn the object file into AVR’s flash.



ISA and Assembly Programming

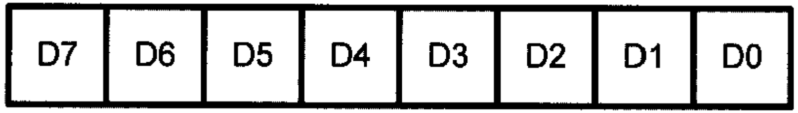
- Machine Code for Instruction “LDI Rd, k
- Op code LDI has machine code 1110



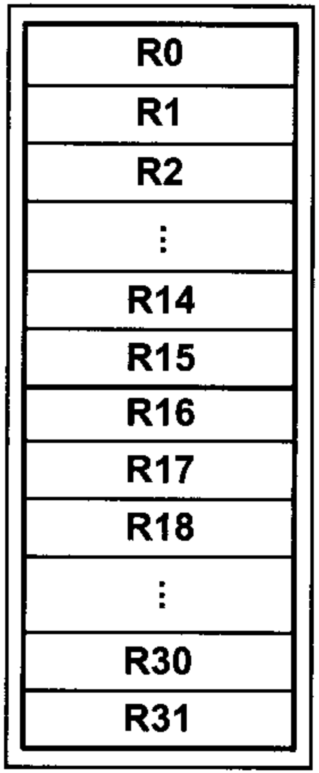
1 1 1 0	k k k k	d d d d	k k k k

LDI Rd, k $16 \leq d \leq 31$, $0 \leq K \leq 255$

AVR: General Purpose Registers

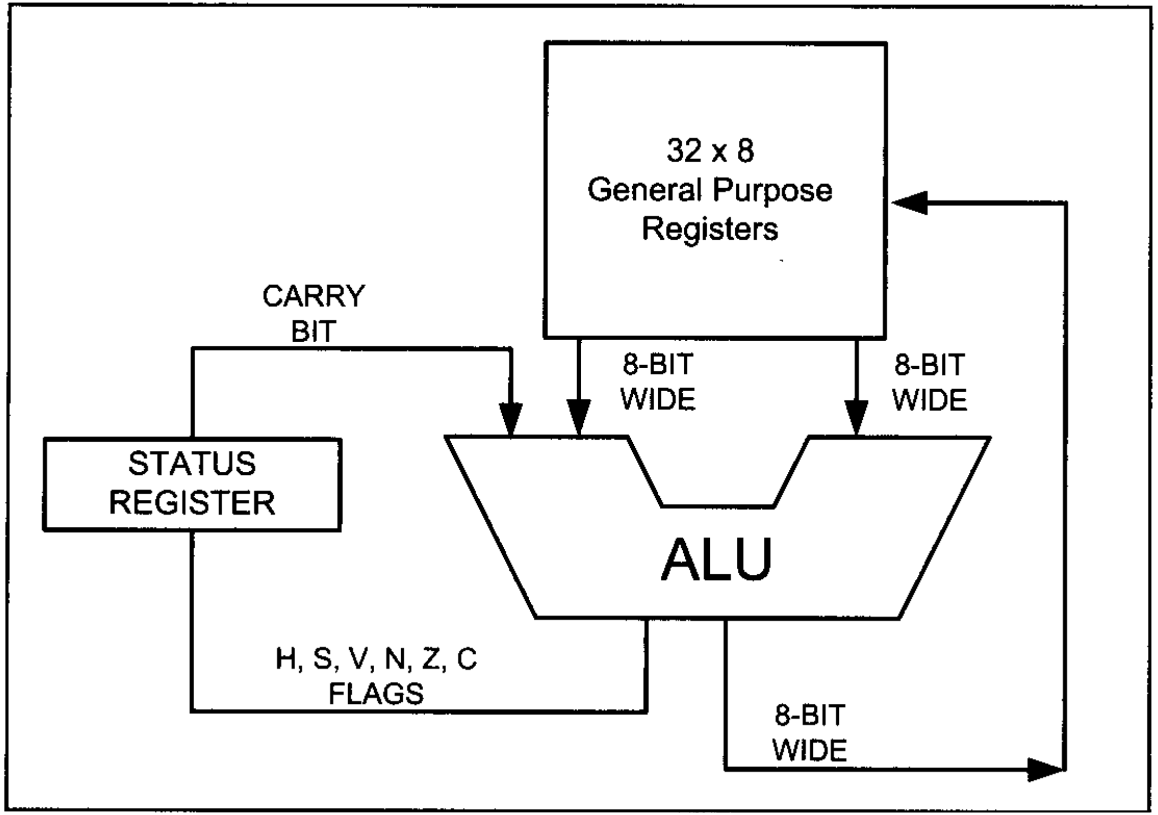


- GPRS in AVR ~ accumulators in other muPs
- GPRs are used to store information temporarily (eg: intermediate results in a computation)
- 8-bit registers D7 through D0
- `MOV R0, R15;` Copy contents of R15 to R0
- `LDI Rd, K`; load Rd (destination) with immediate value of K (decimal)
- `LDI R5, 50;` not allowed. LDI to registers R16 – R31 are allowed
- `LDI R17, 50;` decimal
- `LDI R17, 0x50;` hexadecimal some use $50



General Purpose Registers & ALU

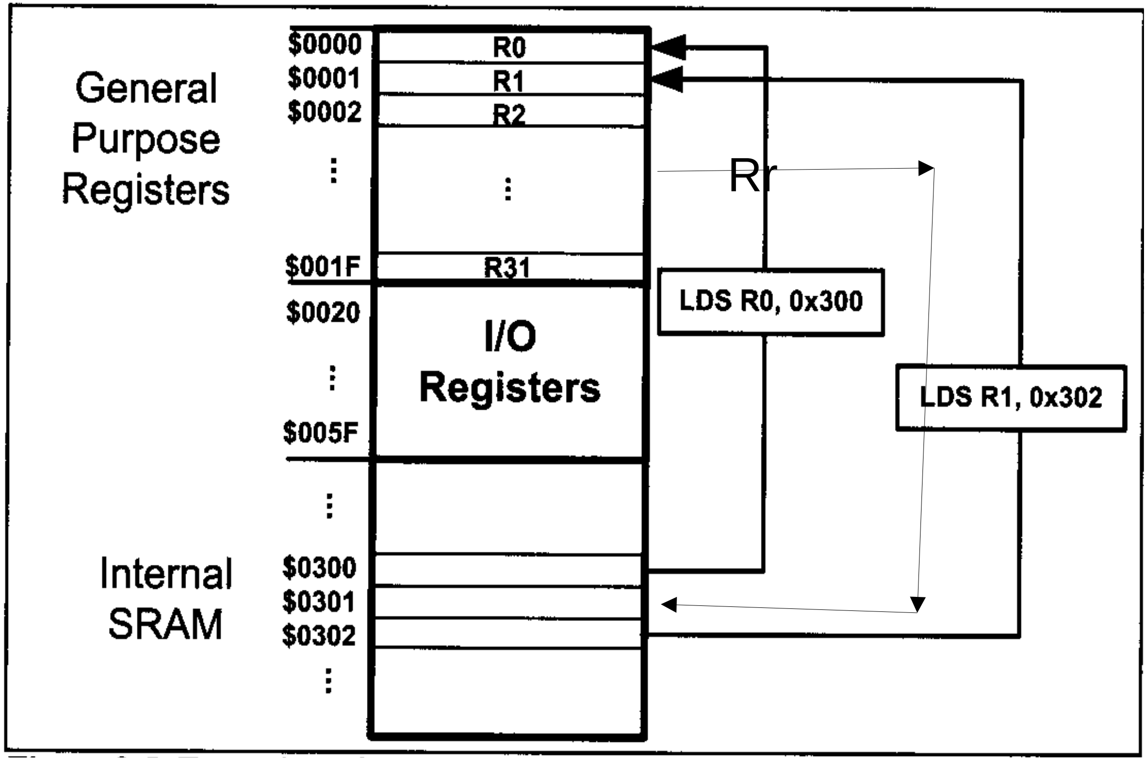
- Add Instruction
- `ADD Rd,Rr;` Add Rr to Rd & result in Rd



Data Memory Instructions

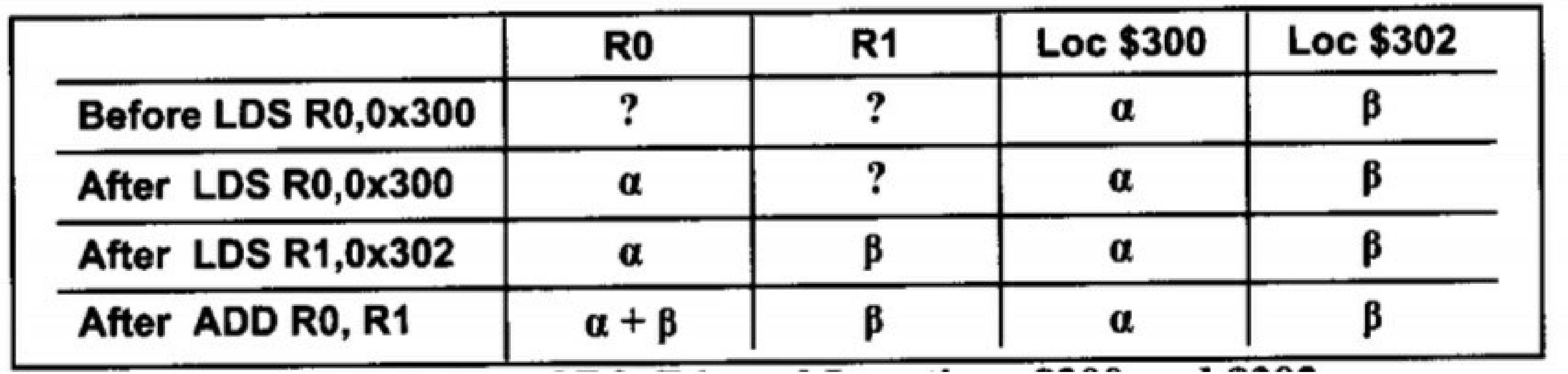
- `LDS Rd, K` ; Load Rd 0<=d<=31 with contents of data location K
- Eg Program

-
  ```
  LDS R0, 0x300;
  LDS R1, 0x302;
  ADD R0, R1;
  ```
- `STS K, Rr;` Store register into data location K



Data Memory Instructions

- `STS K,Rr;` Store contents of register into data location `K`, (0<=r<=31)
- `IN Rd,A;` Load an I/O location to GPR (0<=d<=31), (0<=A<=63)
- `OUT A,Rr;` Store register to I/O location (0<=r<=31),(0<=A<=63)
- Rr, Rd are internal registers



	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	$\alpha + \beta$	β	α	β

Some More Instructions

- `LPM Rd, A;` Load from program memory location A
- `BRCC LineNo;` Branch if clear carry [Go to ‘LineNo’ if carry is zero]
- `ST X, Rd;` Store the data in register Rd to a mm location pointed by X
- `NOP;` No operation

AVR Program

```
        .CSEG
        LDI ZL,LOW(NUM<<1)
        LDI ZH,HIGH(NUM<<1)
        LDI XL,0x60
        LDI XH,0x00
        LDI R16,00
        LPM R0, Z+
        LPM R1, Z
        ADD R0, R1
        BRCC abc
        LDI R16, 0x01
abc:ST X+, R0
    ST X, R16
    NOP
    NUM: .db 0xD3,0x5F;
```

IN Versus LDS

- IN lasts for 1 machine cycle while LDS lasts for 2 machine cycle
- IN ~ 2 byte instruction while LDS is 4 byte instruction
- IN instruction is available in ALL AVRs, while LDS ?
- When we use the IN instruction, we can use the names of the I/O registers instead of their addresses

Instructions that Affect Flag Bits

- Some Instructions affect all the six flag bits C, H, Z, S, V and N
- Some instructions affect no flag at all.
  - Eg. Load instructions
- Some instructions
  -

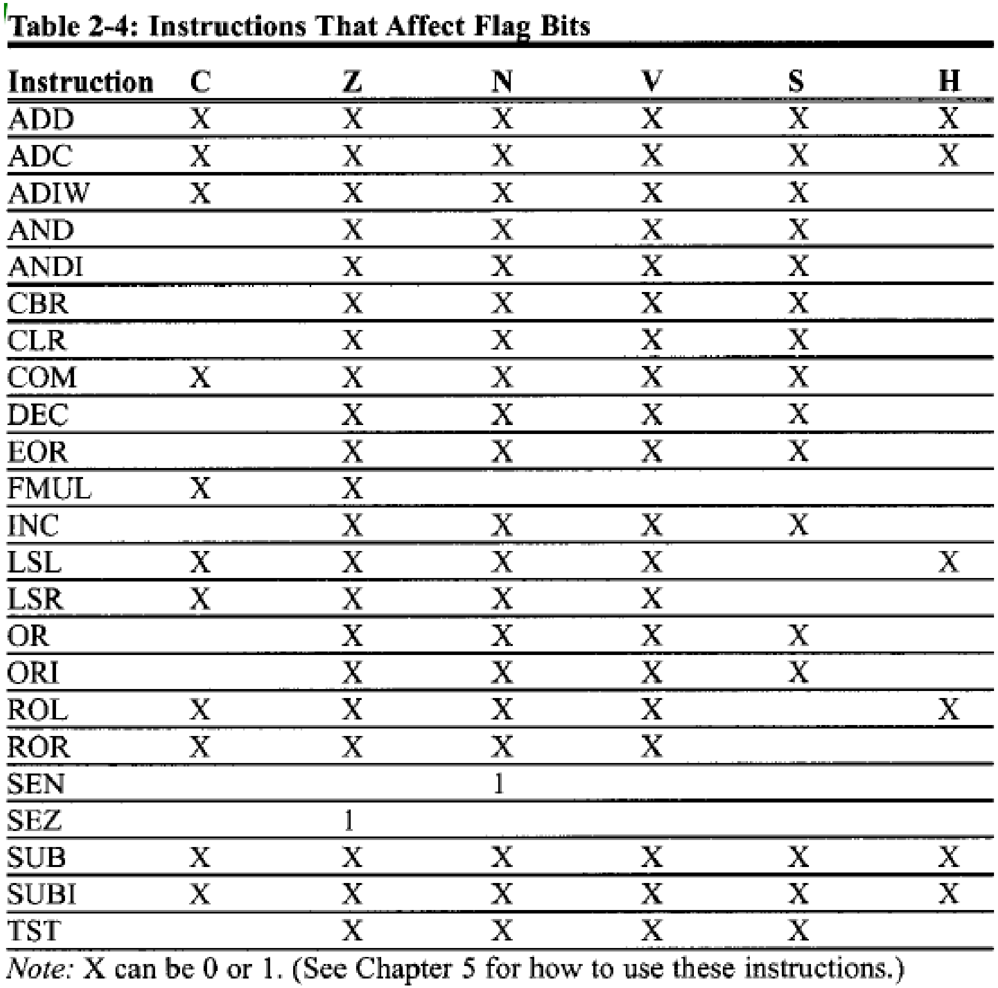
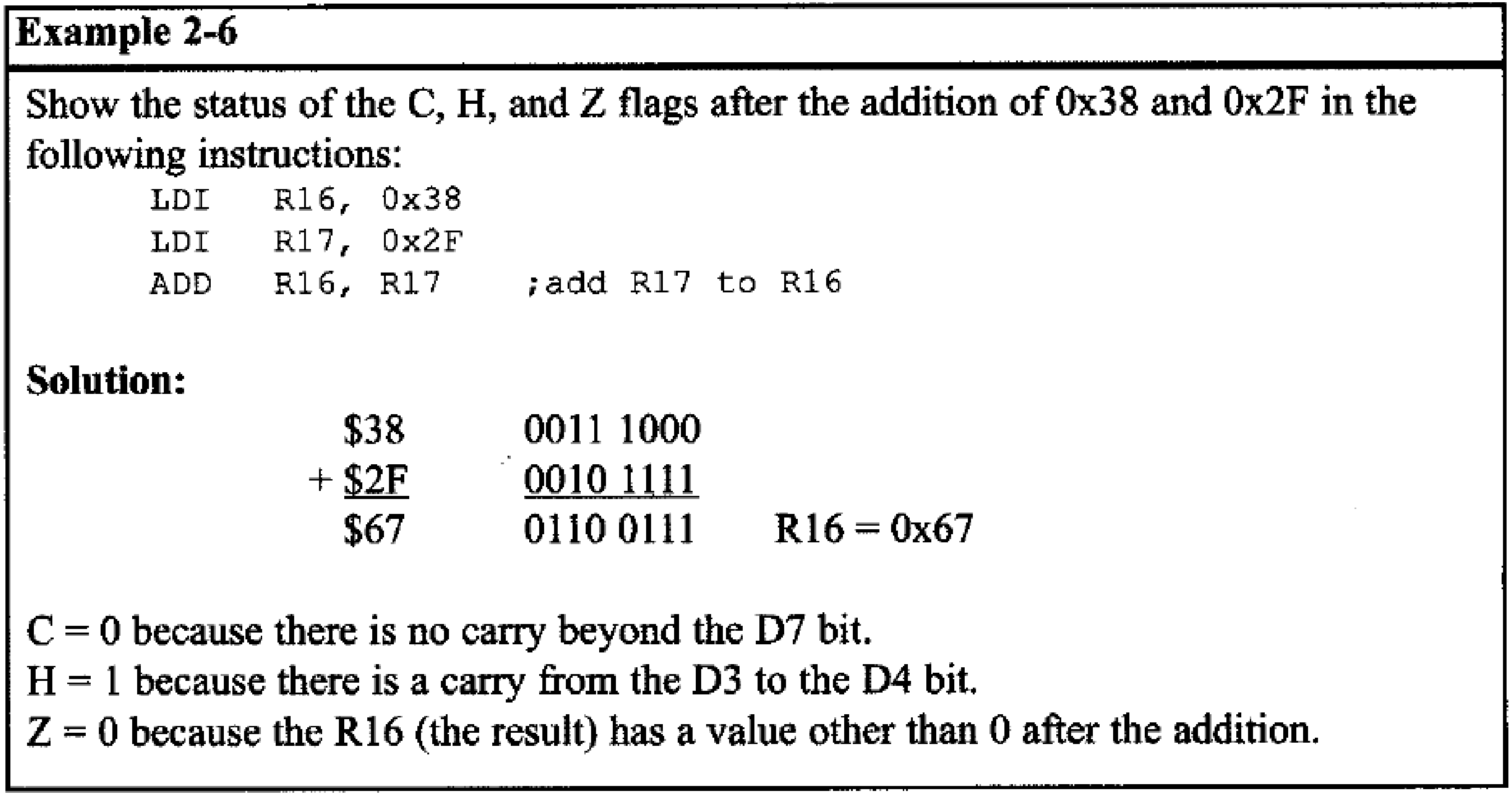


Table 2-4: Instructions That Affect Flag Bits

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	
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.)

Status Register & ADD Instruction



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:

$$\begin{array}{rll} \$38 & 0011\ 1000 & \\ +\ \underline{\$2F} & \underline{0010\ 1111} & \\ \$67 & 0110\ 0111 & \mathrm{R16} = \mathrm{0x67} \end{array}$$

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 Branch Instructions Conditional on Flag Bits

- Flag bits and decision making
  - Conditional flags C, Z, N, V, S, H, T and I
    - Indicate some conditions which could be used in conditional branch instructions.
  - S, the sign bit
    - Exclusive-OR-ing of N & V flags
  - H, half carry flag It indicates when a carry or borrow has been generated out of the least significant four bits of the accumulator register following the execution of an arithmetic instruction. It is primarily used in decimal (BCD) arithmetic instructions.

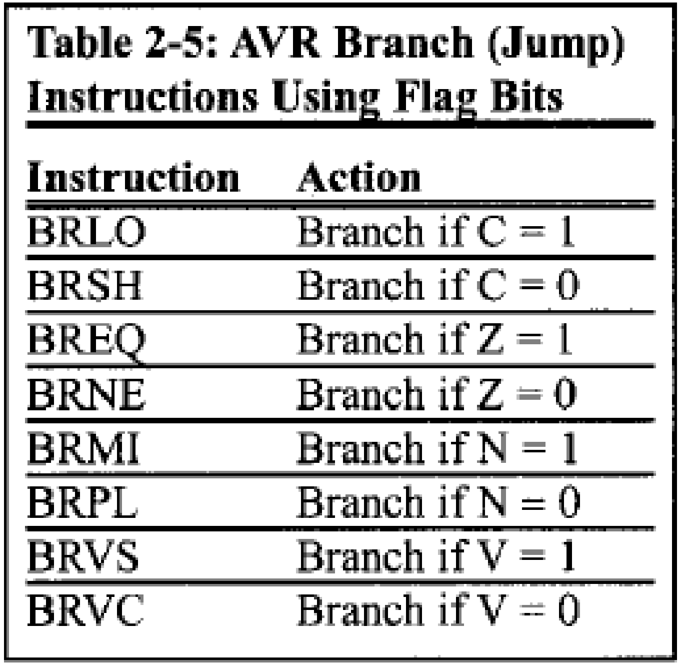
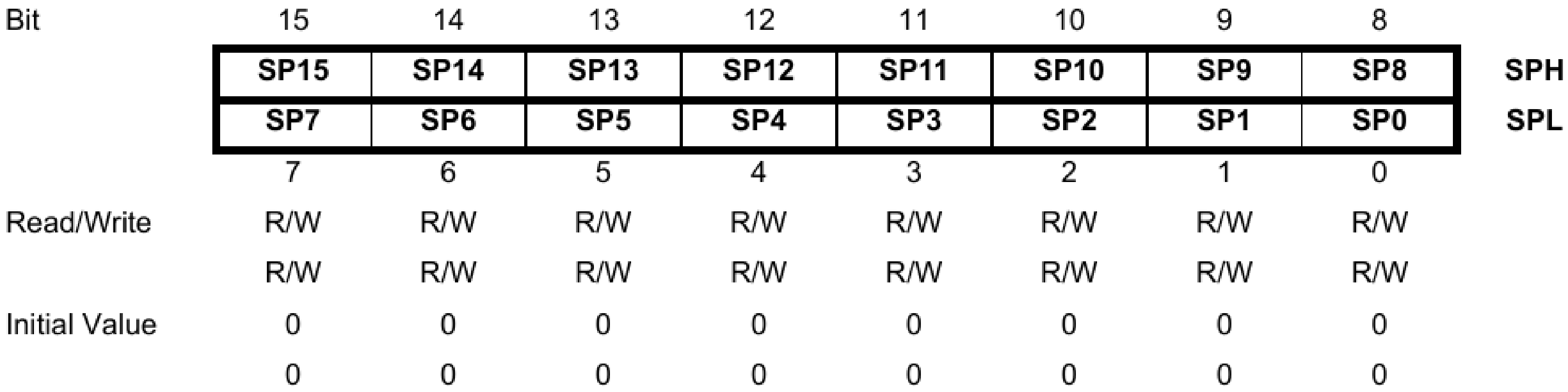


Table 2-5: AVR Branch (Jump) Instructions Using Flag Bits

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

Stack Pointer – During Interrupt & Subroutine

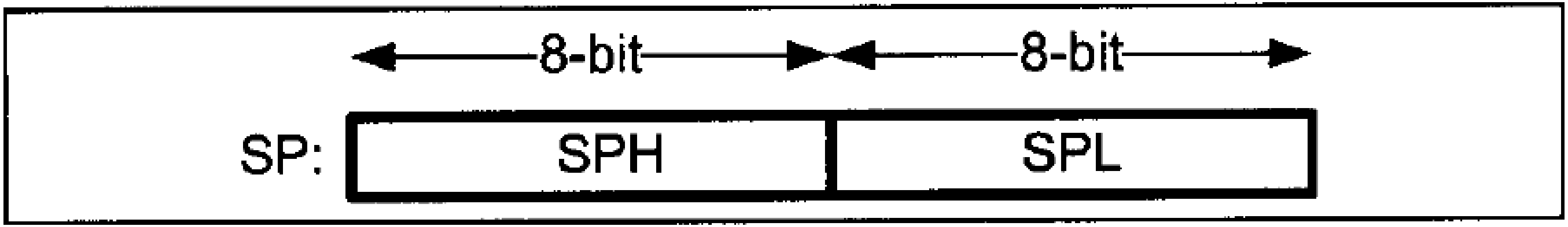
- Stack Pointer is a 2 byte register implemented through SPH in address 0x5D and SPL in address 0x5E in the I/O registers area of the internal memory.
- Stack is used to point to the instruction to be executed next, before the PC jumps to the ISR during interrupt or the subroutine call. This is implemented through “PUSH”
- Once the ISR is served or subroutine is served, the PC fetches back the address of instruction (from where it left) through the command “POP”.



Bit	15	14	13	12	11	10	9	8	
	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Stack Pointer

- Stack Pointer
  - PUSH
    - The next instruction to be executed is ‘pushed’ in
  - POP
    - The stored instructions are “Pop”ped out in the order opposite to the order in which it was stored
    - --> First IN, last OUT queue
  - Used during the interrupt and subroutine part of the program



Stack and Stack Pointer

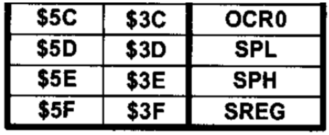
- Stack Memory
  - A piece of memory which could be external RAM (a part of main memory) or could be internal memory, whose access is restricted / different than that of conventional way of accessing
  - Stack memory is a subset of memory, in which one end (say, TOP - address) is fixed, the other (say BOTTOM, address) could be variable
  - When there is no interrupt being served, then stack pointer points to the top end.
  - As and when the Interrupt arrives, the (a) PC values (b) flag /status register values (c) context registers are PUSHed into the stack memory from top, in that order and correspondingly the stack pointer also decremented as shown in illustration.
  - Interrupts can be nested also
  - When the top most (inner most in the nest) interrupt (which is currently getting served), when the corresponding ISR is served, then the above three values are POPped up while the stack pointer moves up.
- Stack Pointer
  - 2 byte pointer pointing to the variable end of the stack memory
- Assembly syntax
  - PUSH
    - The next instruction to be executed is 'pushed' in
  - POP
    - The stored instructions are "Pop"ped out in the order opposite to the order in which it was stored
    - --> First IN, last OUT queue
  - Used during the interrupt and subroutine part of the program

INT 0

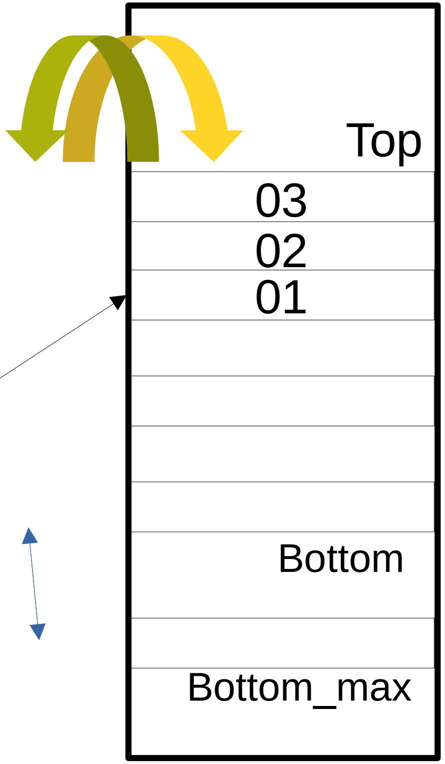
03 – Context register value
02 – Status register value
01 – PC value

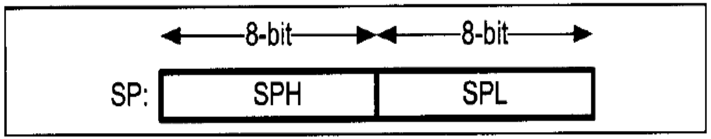
INT 1

13 – Context register value
12 – Status register value
11 – PC value



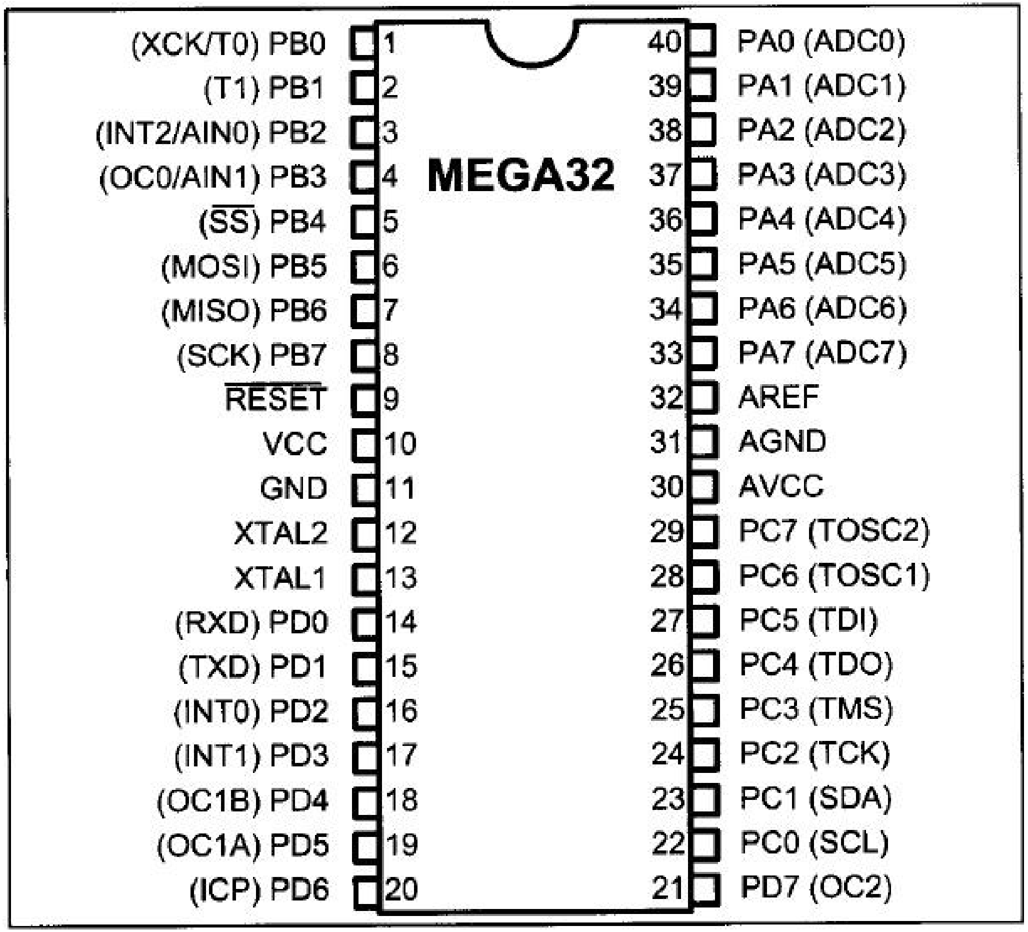
$5C	$3C	OCR0
$5D	$3D	SPL
$5E	$3E	SPH
$5F	$3F	SREG



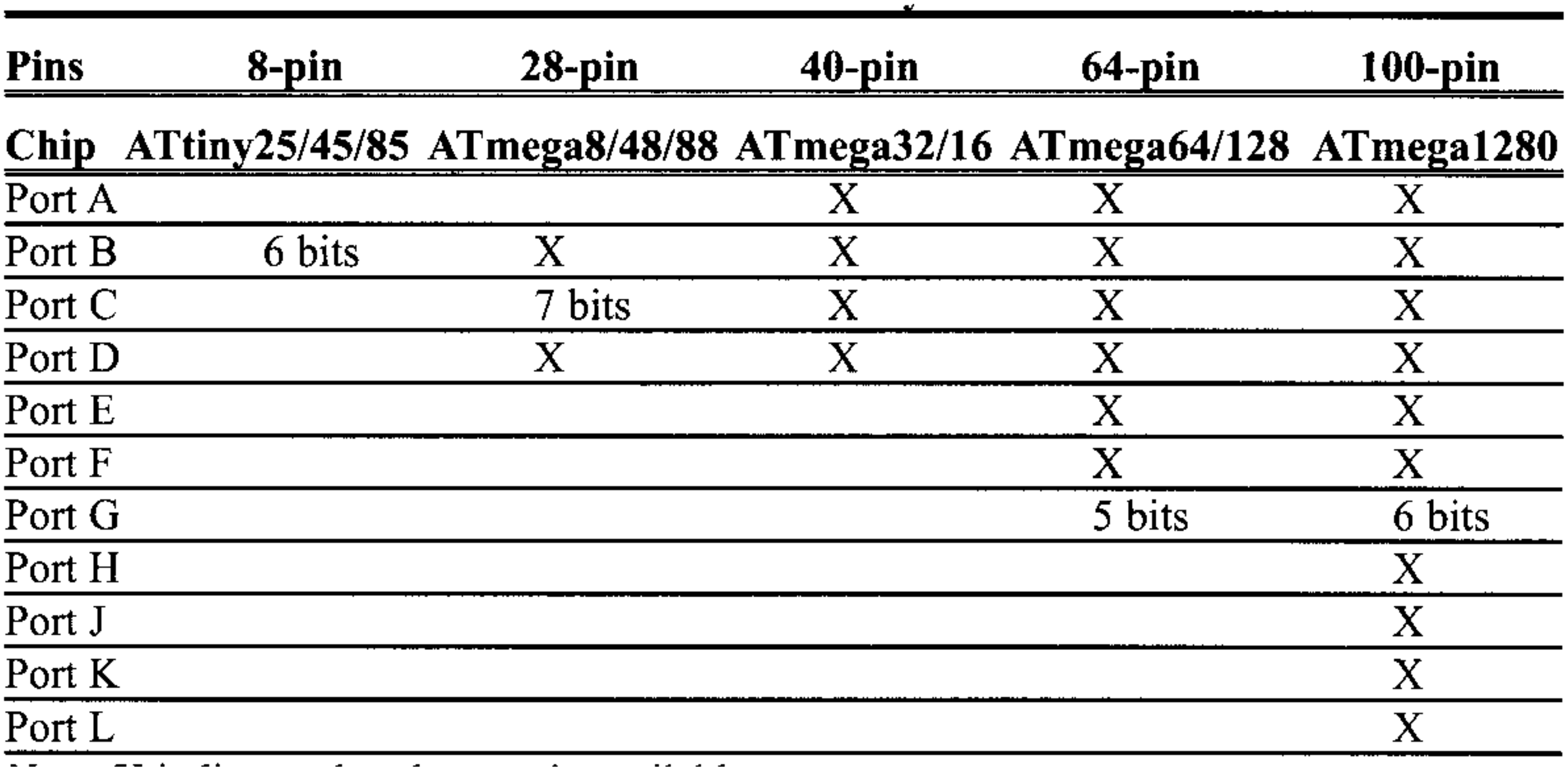


Chapter 4: AVR I/O Programming

Chapter 4: AVR I/O Programming



Chapter 4: AVR I/O Programming



Pins	8-pin	28-pin	40-pin	64-pin	100-pin
Chip	**ATtiny25/45/85**	**ATmega8/48/88**	**ATmega32/16**	**ATmega64/128**	**ATmega1280**
Port A			X	X	X
Port B	6 bits	X	X	X	X
Port C		7 bits	X	X	X
Port D		X	X	X	X
Port E				X	X
Port F				X	X
Port G				5 bits	6 bits
Port H					X
Port J					X
Port K					X
Port L					X

Chapter 4: AVR I/O Programming

Data General Interrupt Control Register - GICR

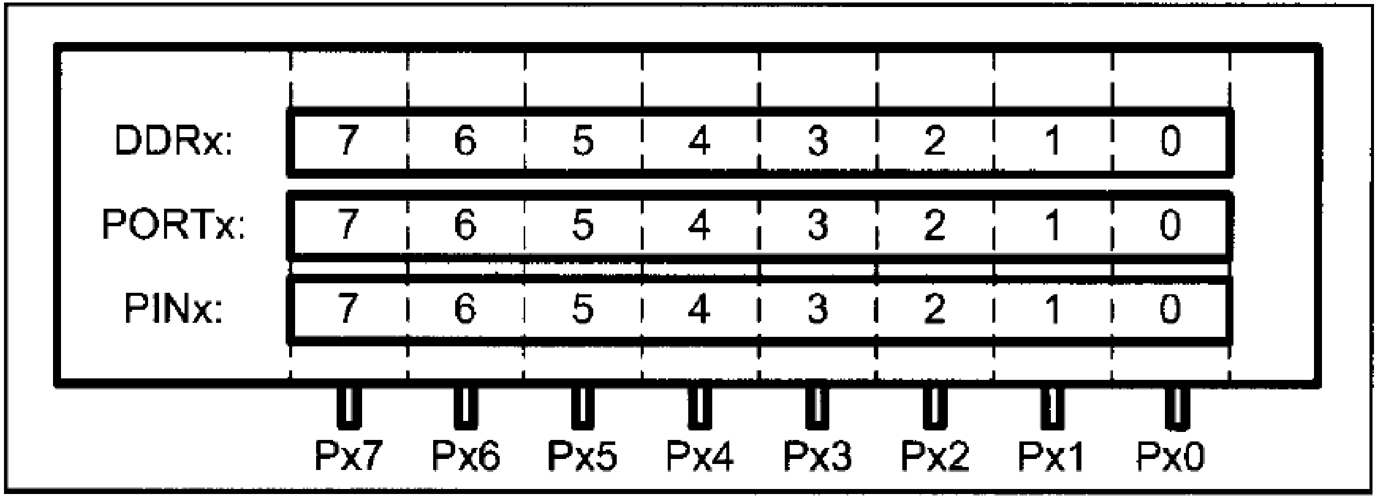
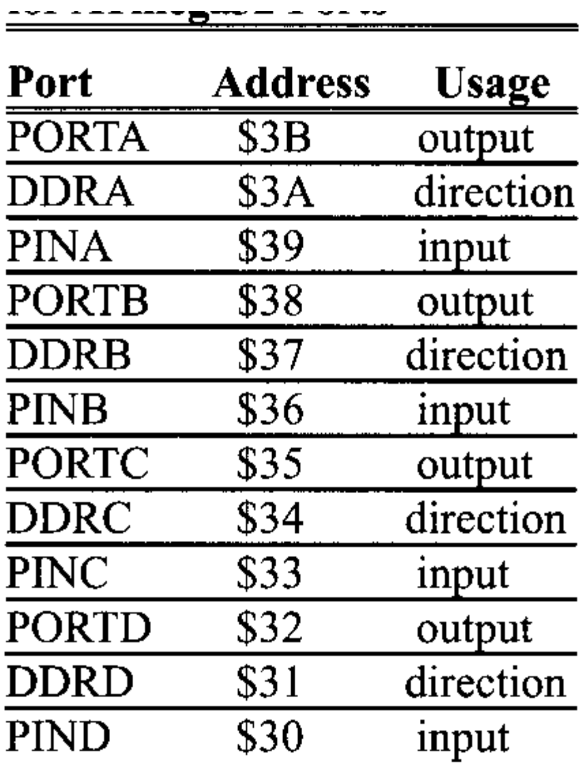
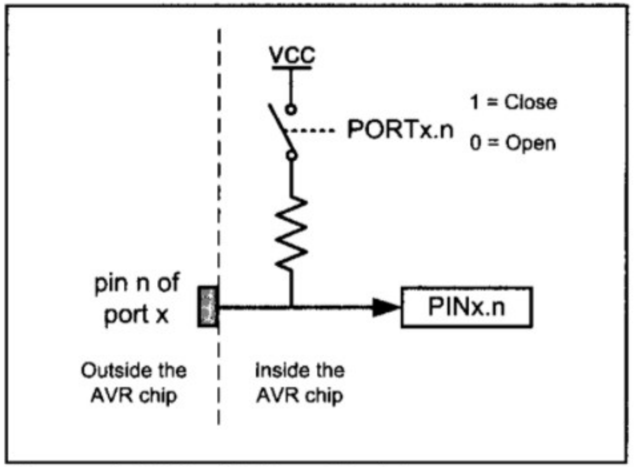


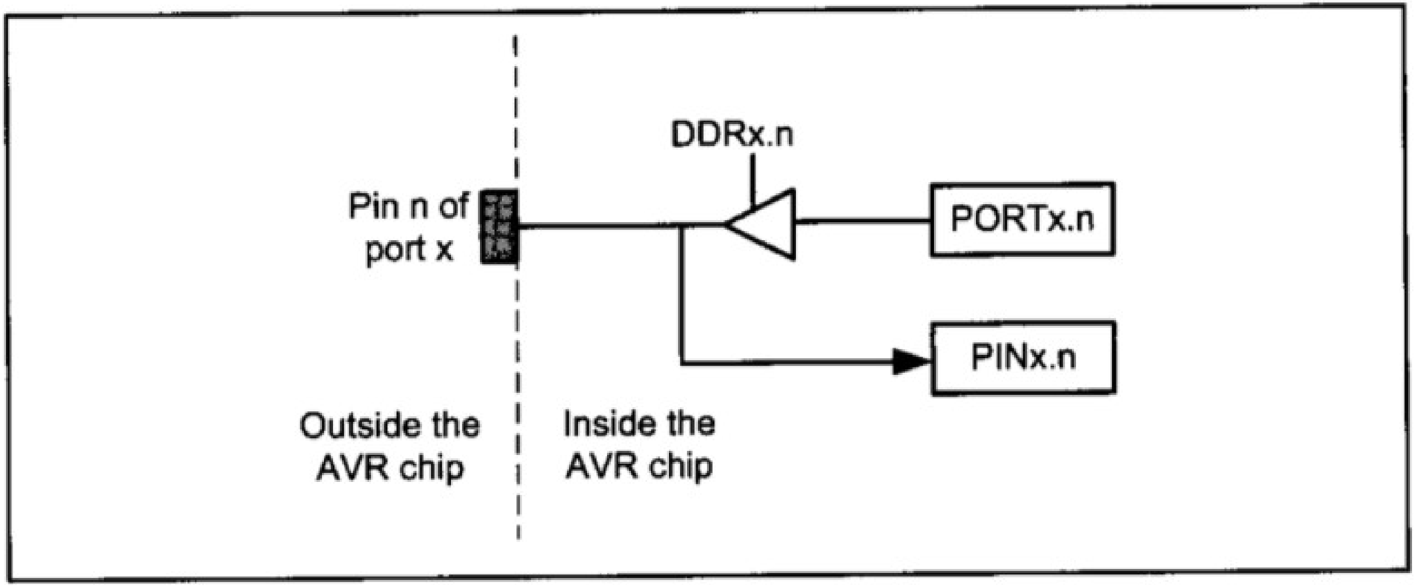
Figure 4-2. Relations Between the Registers and the Pins of AVR



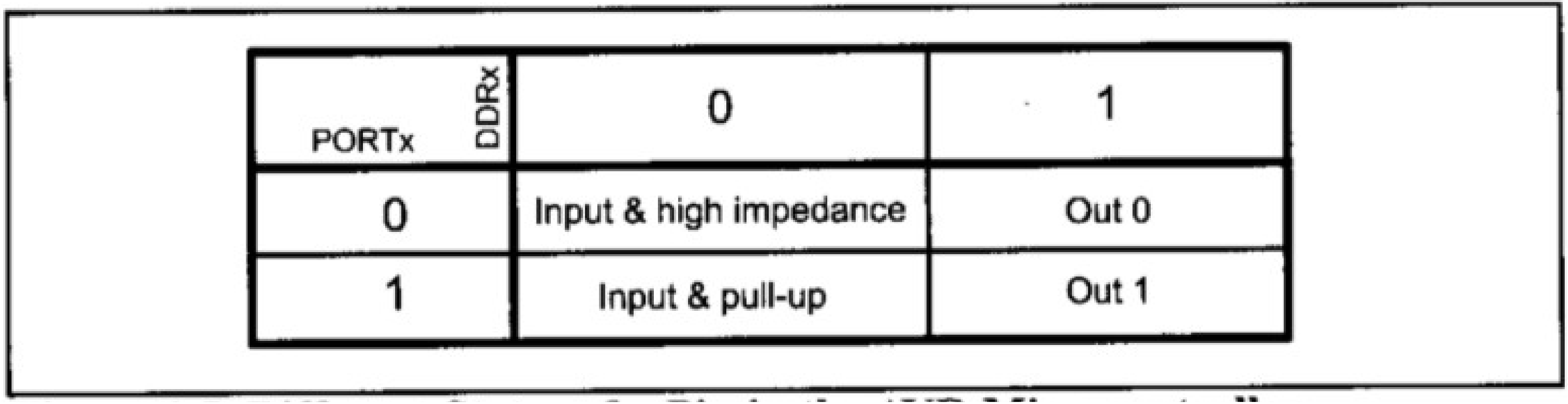
Port	Address	Usage
PORTA	$3B	output
DDRA	$3A	direction
PINA	$39	input
PORTB	$38	output
DDRB	$37	direction
PINB	$36	input
PORTC	$35	output
DDRC	$34	direction
PINC	$33	input
PORTD	$32	output
DDRD	$31	direction
PIND	$30	input

Chapter 4: AVR I/O Programming





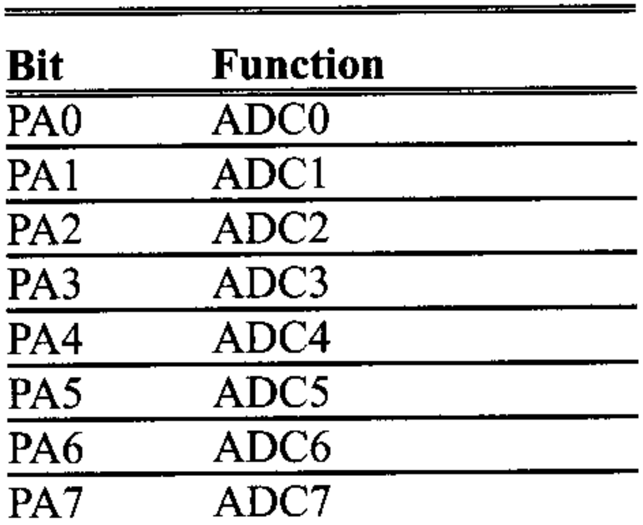
Chapter 4: AVR I/O Programming



PORTx / DDRx	0	1
0	Input & high impedance	Out 0
1	Input & pull-up	Out 1

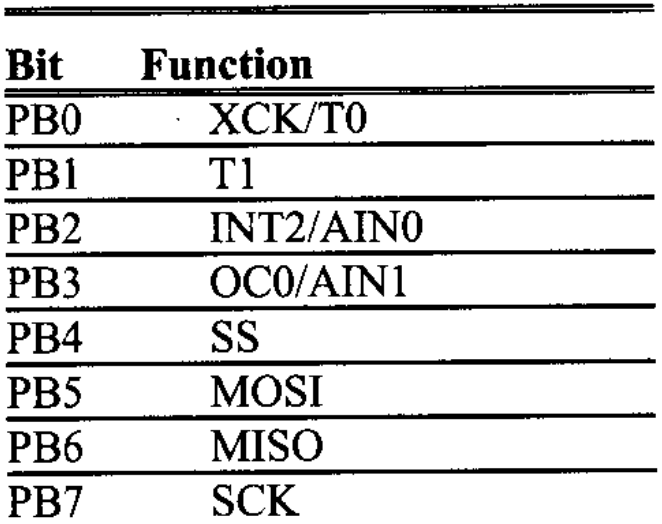
Different States of a Pin in the AVR Microcontroller

Chapter 4: AVR I/O Programming



Bit	Function
PA0	ADC0
PA1	ADC1
PA2	ADC2
PA3	ADC3
PA4	ADC4
PA5	ADC5
PA6	ADC6
PA7	ADC7

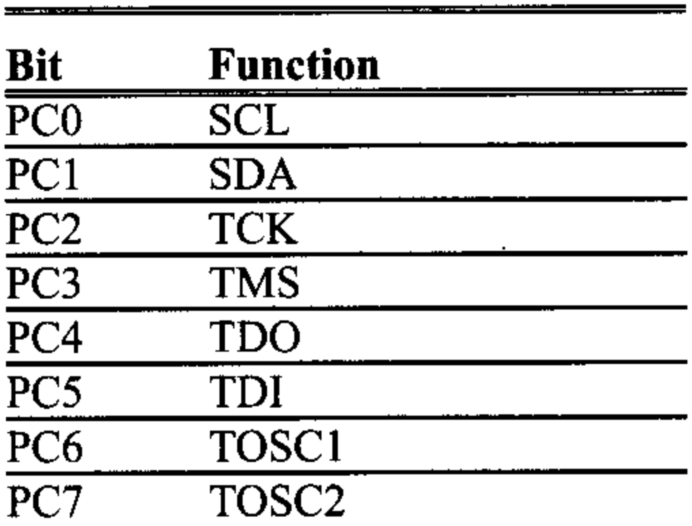
Table 4-3. Port A Alternate Functions



Bit	Function
PB0	XCK/T0
PB1	T1
PB2	INT2/AIN0
PB3	OC0/AIN1
PB4	SS
PB5	MOSI
PB6	MISO
PB7	SCK

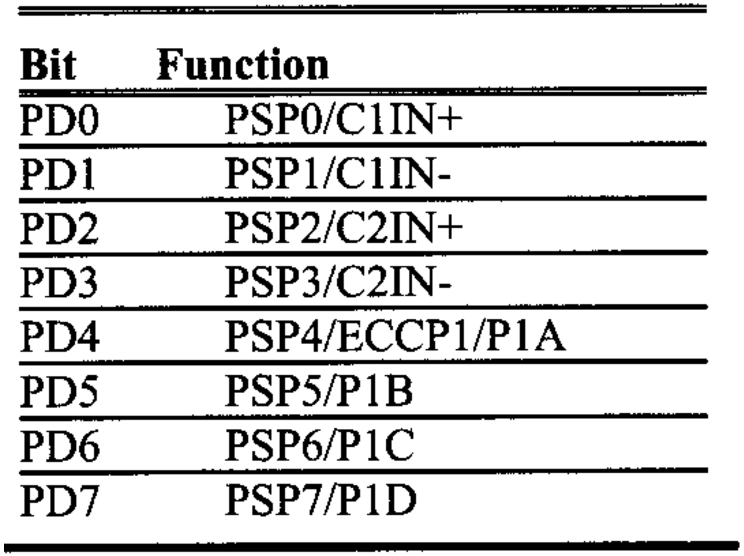
Table 4-4. Port B Alternate Functions

Chapter 4: AVR I/O Programming



Bit	Function
PC0	SCL
PC1	SDA
PC2	TCK
PC3	TMS
PC4	TDO
PC5	TDI
PC6	TOSC1
PC7	TOSC2

Table 4-5. Port C Alternate Functions

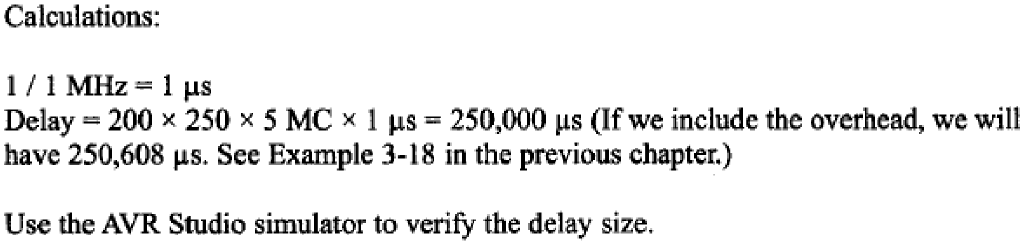


Bit	Function
PD0	PSP0/C1IN+
PD1	PSP1/C1IN-
PD2	PSP2/C2IN+
PD3	PSP3/C2IN-
PD4	PSP4/ECCP1/P1A
PD5	PSP5/P1B
PD6	PSP6/P1C
PD7	PSP7/P1D

Table 4-6. Port D Alternate Functions

Chapter 4: I/O Programming

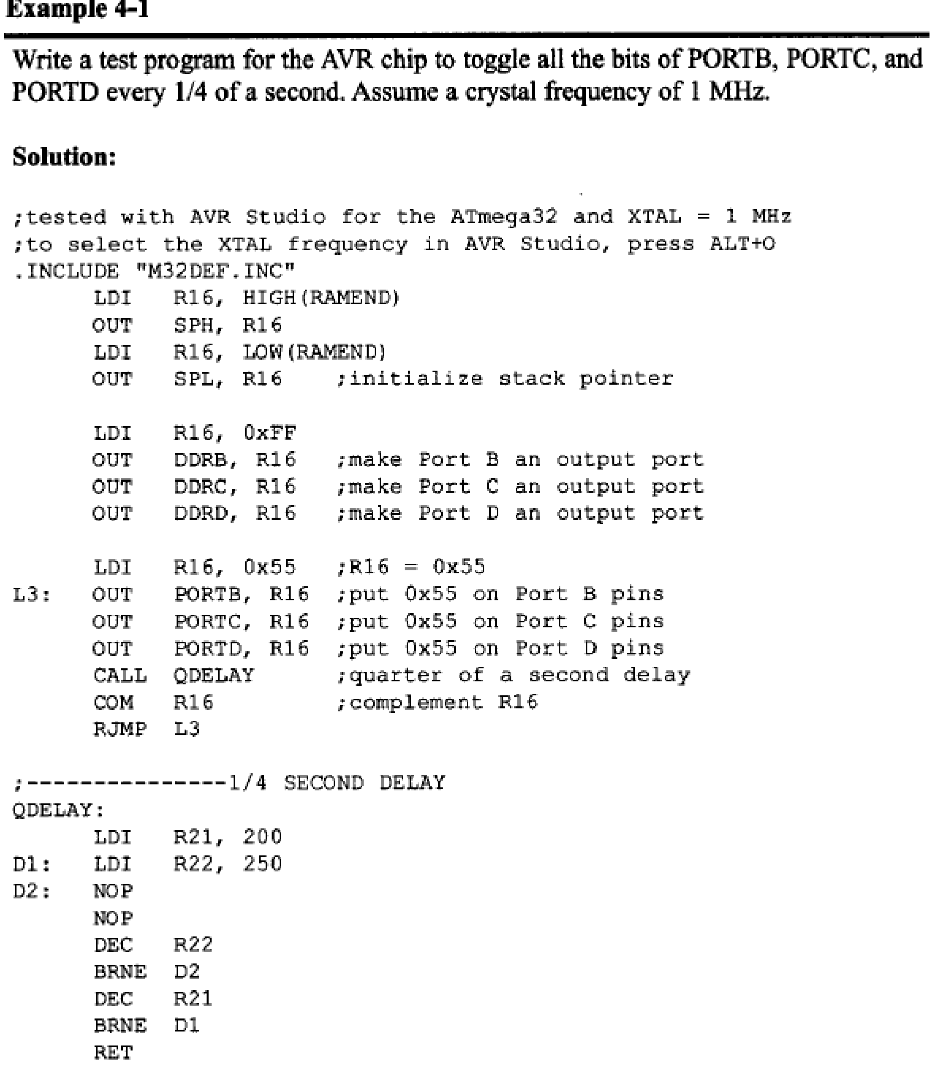
- `CALL SbRtne;` Long call to a Subroutine "SbRtne". SbRtne is an integer (say, k) indicating the line number within the program memory. Devices with 16 bits PC, 0<=k<=64K
- `COM R16;` 1's complement of R16
- `RJMP L3;` Relative JUMP with respect to current PC value.
- `DEC R21;` Decrement the content of R21 by one unit
- `BRNE D2;` Branch if Not Equal
- `RET;` Return to main program



Calculations:

1 / 1 MHz = 1 μs
Delay = 200 × 250 × 5 MC × 1 μs = 250,000 μs (If we include the overhead, we will have 250,608 μs. See Example 3-18 in the previous chapter.)

Use the AVR Studio simulator to verify the delay size.



Example 4-1

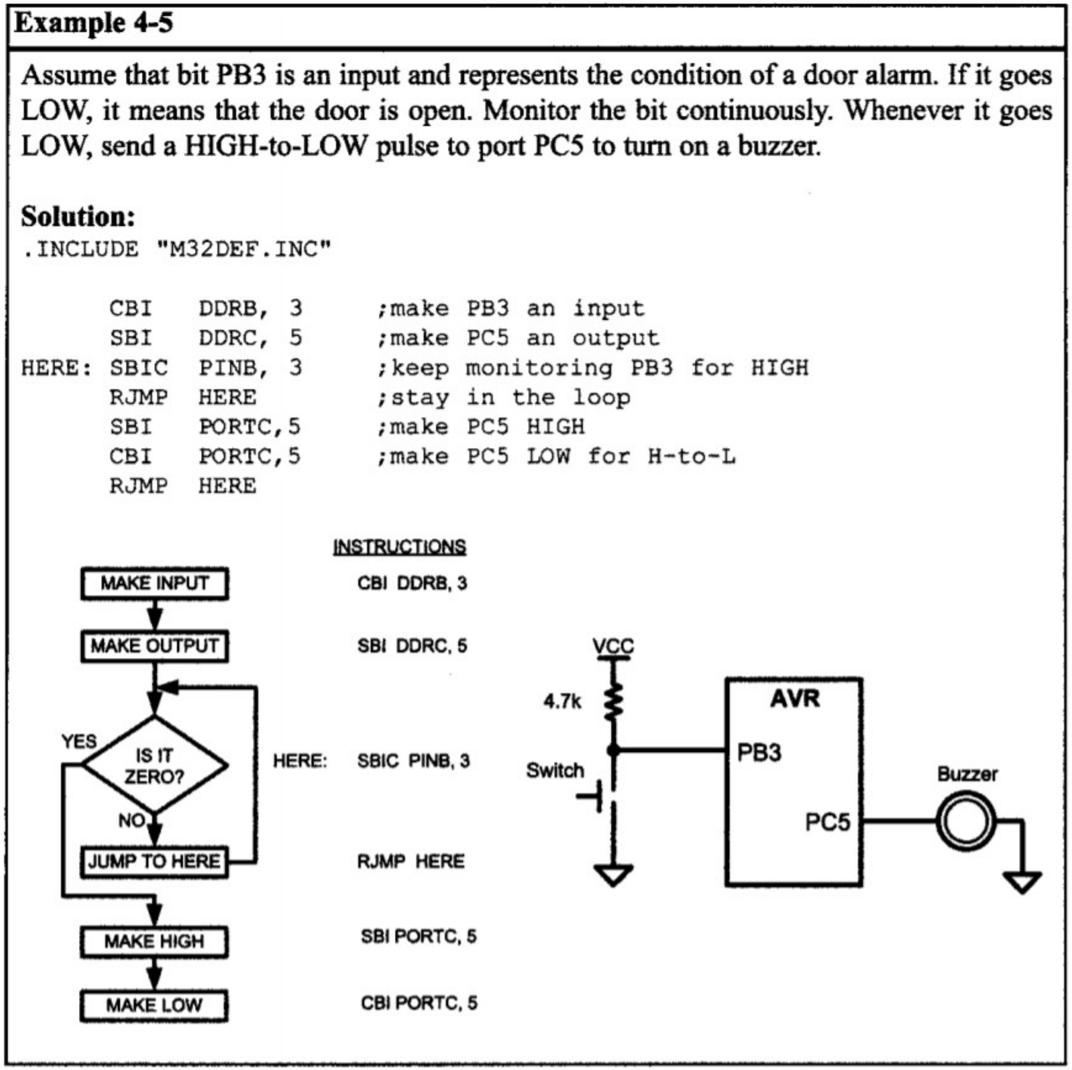
Write a test program for the AVR chip to toggle all the bits of PORTB, PORTC, and PORTD every 1/4 of a second. Assume a crystal frequency of 1 MHz.

Solution:

```
;tested with AVR Studio for the ATmega32 and XTAL = 1 MHz
;to select the XTAL frequency in AVR Studio, press ALT+O
.INCLUDE "M32DEF.INC"
      LDI   R16, HIGH(RAMEND)
      OUT   SPH, R16
      LDI   R16, LOW(RAMEND)
      OUT   SPL, R16    ;initialize stack pointer

      LDI   R16, 0xFF
      OUT   DDRB, R16   ;make Port B an output port
      OUT   DDRC, R16   ;make Port C an output port
      OUT   DDRD, R16   ;make Port D an output port

      LDI   R16, 0x55   ;R16 = 0x55
L3:   OUT   PORTB, R16  ;put 0x55 on Port B pins
      OUT   PORTC, R16  ;put 0x55 on Port C pins
      OUT   PORTD, R16  ;put 0x55 on Port D pins
      CALL  QDELAY      ;quarter of a second delay
      COM   R16         ;complement R16
      RJMP  L3

;---------------1/4 SECOND DELAY
QDELAY:
      LDI   R21, 200
D1:   LDI   R22, 250
D2:   NOP
      NOP
      DEC   R22
      BRNE  D2
      DEC   R21
      BRNE  D1
      RET
```

Chapter 4: AVR I/O Programming

- `SBI IO-Reg, #bit;` Set the bit to 1 in the given I/O register
- `CBI IO-Reg, #bit;` Clear Bit in I/O Register
- `SBIC IO-Reg, #bit;` Skip the next instruction if given bit = 0 in the given I/O register.



Example 4-5

Assume that bit PB3 is an input and represents the condition of a door alarm. If it goes LOW, it means that the door is open. Monitor the bit continuously. Whenever it goes LOW, send a HIGH-to-LOW pulse to port PC5 to turn on a buzzer.

Solution:

```
.INCLUDE "M32DEF.INC"

        CBI    DDRB, 3      ;make PB3 an input
        SBI    DDRC, 5      ;make PC5 an output
HERE:   SBIC   PINB, 3      ;keep monitoring PB3 for HIGH
        RJMP   HERE         ;stay in the loop
        SBI    PORTC,5      ;make PC5 HIGH
        CBI    PORTC,5      ;make PC5 LOW for H-to-L
        RJMP   HERE
```

Chapter 10: AVR Interrupt Programming

Background Information To Interrupt Programming

- Interfacing peripherals to microprocessor
  - Address mapped
  - Port mapped
  - Access
    - Polling
    - Interrupt
- Interrupt
  - PUSH into stack pointer
  - Interrupt Service Routine (ISR)
  - POP out of stack pointer and continue

AVR External Interrupt Programming

- Interrupt Service Routine (ISR) per every interrupt
  - Fixed location where the ISR is stored
  - Group of memory locations set aside to hold the addresses of ISRs --> **interrupt vector table**
- Interrupt Sources
  - Interrupts for each Timers
    - Overflow timer
    - Compare match
  - **External Hardware Interrupt**
    - PD2 (INT0)
    - PD3 (INT1)
    - PB2 (INT2)
  - Serial communication’s USART has three interrupts
  - SPI interrupts
  - ADC’s interrupt

AVR Interrupt Vector Table

- External Interrupt request 0
  - ROM location 0002 corresponding to INT0 (hardware pin PD2, PORTD.2)
- External Interrupt request 1
  - ROM location 0004 corresponding to INT1 (hardware pin PD3, PORTD.3)
- External Interrupt request 2
  - ROM location 0006 corresponding to INT2 (hardware pin PB2, PORTB.2)

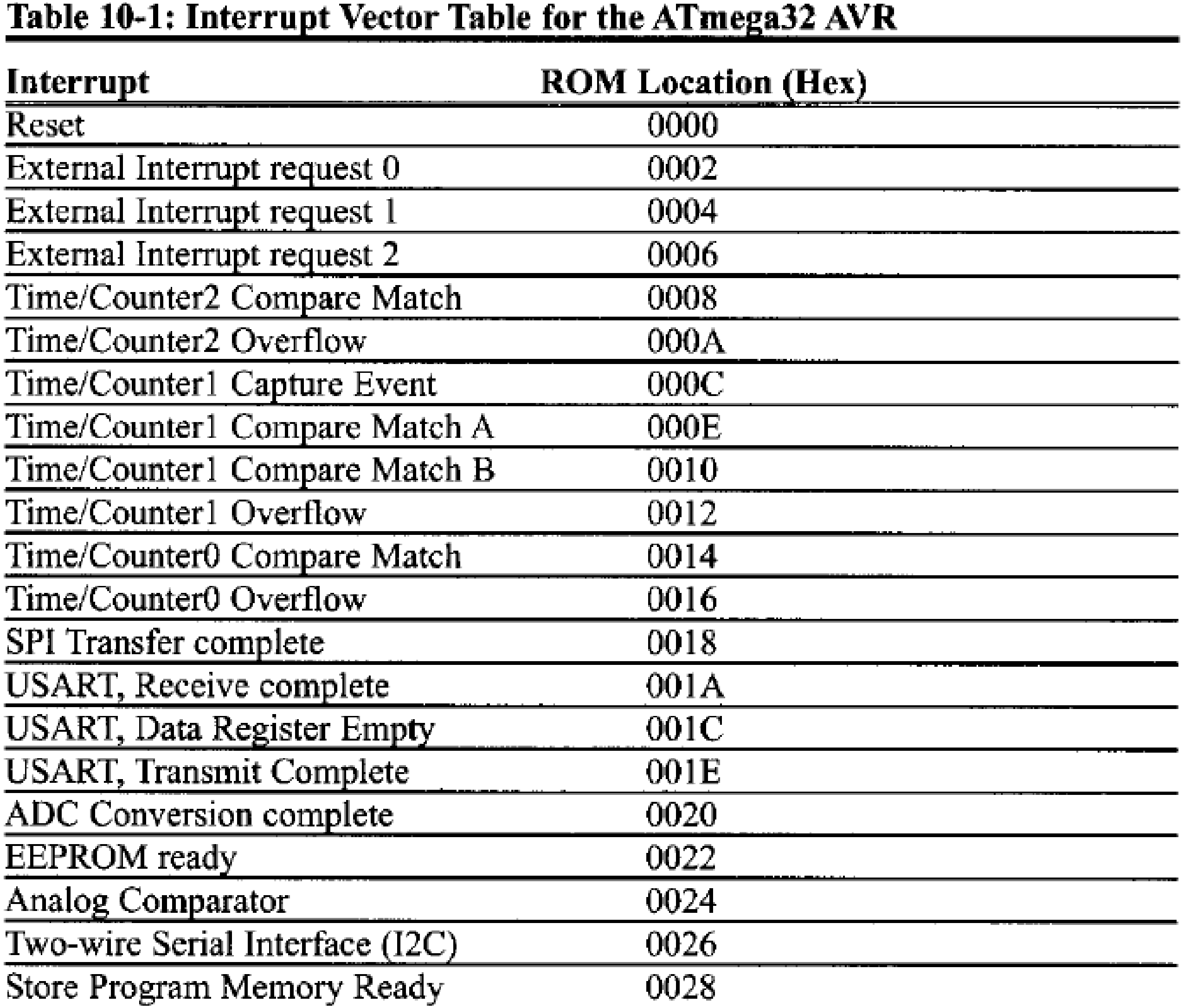
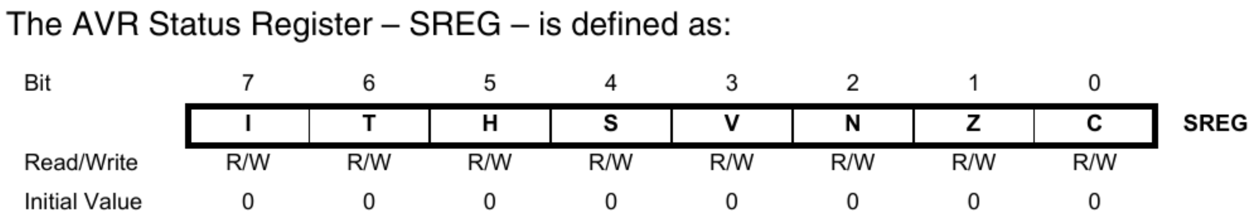


Table 10-1: Interrupt Vector Table for the ATmega32 AVR

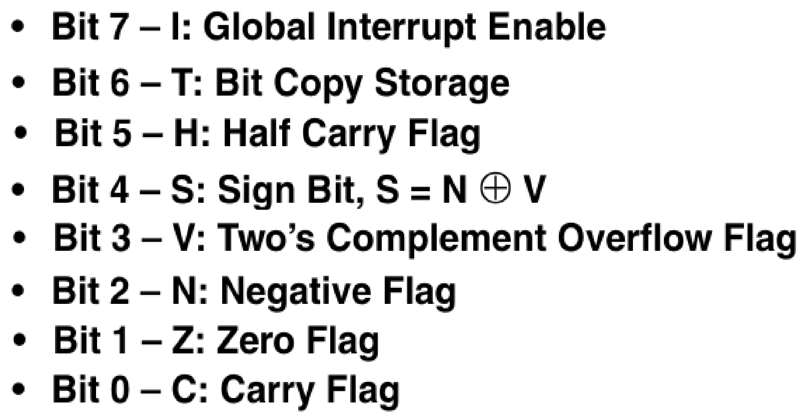
Interrupt	ROM Location (Hex)
Reset	0000
External Interrupt request 0	0002
External Interrupt request 1	0004
External Interrupt request 2	0006
Time/Counter2 Compare Match	0008
Time/Counter2 Overflow	000A
Time/Counter1 Capture Event	000C
Time/Counter1 Compare Match A	000E
Time/Counter1 Compare Match B	0010
Time/Counter1 Overflow	0012
Time/Counter0 Compare Match	0014
Time/Counter0 Overflow	0016
SPI Transfer complete	0018
USART, Receive complete	001A
USART, Data Register Empty	001C
USART, Transmit Complete	001E
ADC Conversion complete	0020
EEPROM ready	0022
Analog Comparator	0024
Two-wire Serial Interface (I2C)	0026
Store Program Memory Ready	0028

Atmega8L AVR Status Register (SREG)



The AVR Status Register – SREG – is defined as:

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



- **Bit 7 – I: Global Interrupt Enable**
- **Bit 6 – T: Bit Copy Storage**
- **Bit 5 – H: Half Carry Flag**
- **Bit 4 – S: Sign Bit, S = N ⊕ V**
- **Bit 3 – V: Two's Complement Overflow Flag**
- **Bit 2 – N: Negative Flag**
- **Bit 1 – Z: Zero Flag**
- **Bit 0 – C: Carry Flag**

- Enabling / Disabling an Interrupt through D7 of SREG (the 'I' bit) --> enabling / disabling **interrupts globally**
- By reset all would be disabled (masked)
  - CLI (Clear Interrupt) makes I = 0
  - **Enabling Interrupt**
    - Set I = 1 of SREG register must be set to HIGH to allow interrupts to happen.
    - Done through SEI (Set Interrupt) instruction
- During interrupt, after PUSH, 'I' bit is reset to '0'.

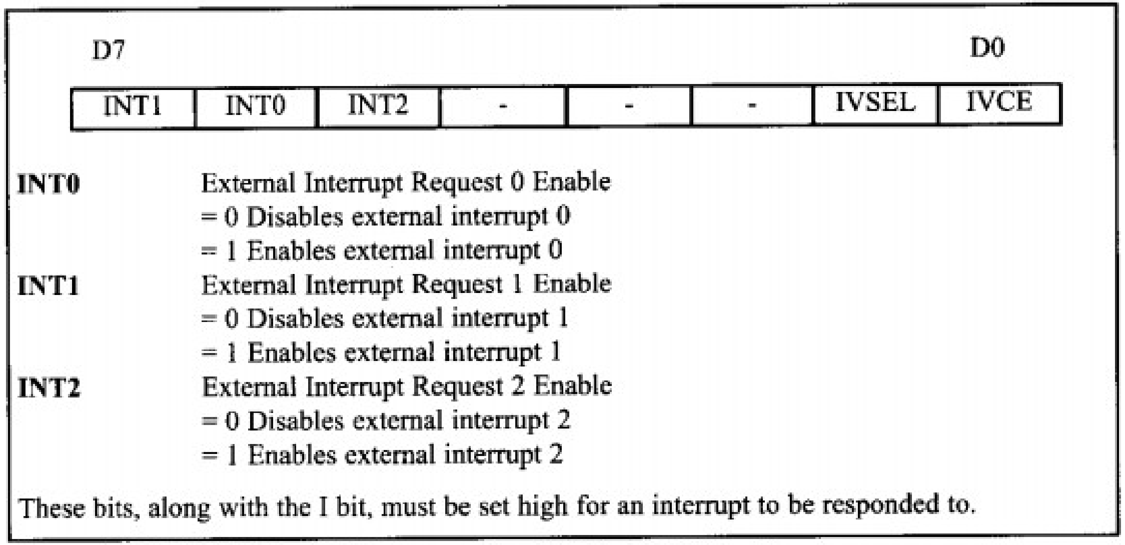
- Difference between RET and RETI in ISR
  - RETI – after POPing, I = 1, while RET still I = 0

External Interrupt: GICR (General Interrupt Control Register)

- Enable external hardware interrupt
  - Done by setting INTx in GICR register
  - Eg.

```
LDI  R20, 0x40
OUT  GICR, R20
```

- INT0 is the low level triggered interrupt by default
  - Low signal to PD2, the muC would be interrupted and jump to 0x 0002 location in vector table to service the corresponding ISR
- D



D7 D0

INT1	INT0	INT2	-	-	-	IVSEL	IVCE

INT0 External Interrupt Request 0 Enable
= 0 Disables external interrupt 0
= 1 Enables external interrupt 0

INT1 External Interrupt Request 1 Enable
= 0 Disables external interrupt 1
= 1 Enables external interrupt 1

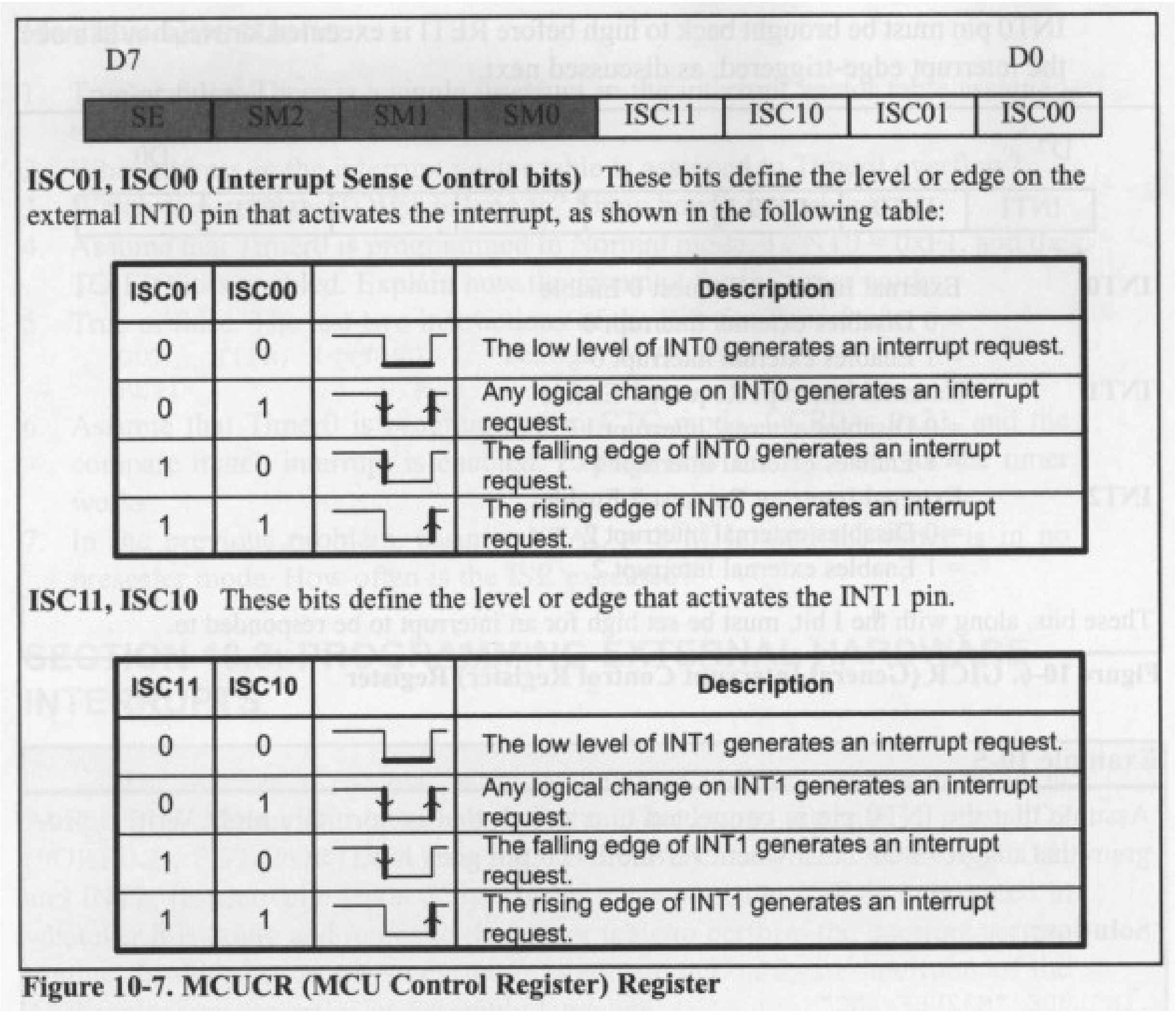
INT2 External Interrupt Request 2 Enable
= 0 Disables external interrupt 2
= 1 Enables external interrupt 2

These bits, along with the I bit, must be set high for an interrupt to be responded to.

Figure 10-6. GICR (General Interrupt Control Register) Register

Interrupt Triggering Mechanism in AVR

- Types of activation for external hardware Interrupt
  - Level triggered
  - Edge triggered
- INT0 and INT1 are either edge or level triggered
- INT2 is necessarily edge triggered
- The way to configure INT0 and INT1 for triggering is given here



D7							D0
SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00

ISC01, ISC00 (Interrupt Sense Control bits) These bits define the level or edge on the external INT0 pin that activates the interrupt, as shown in the following table:

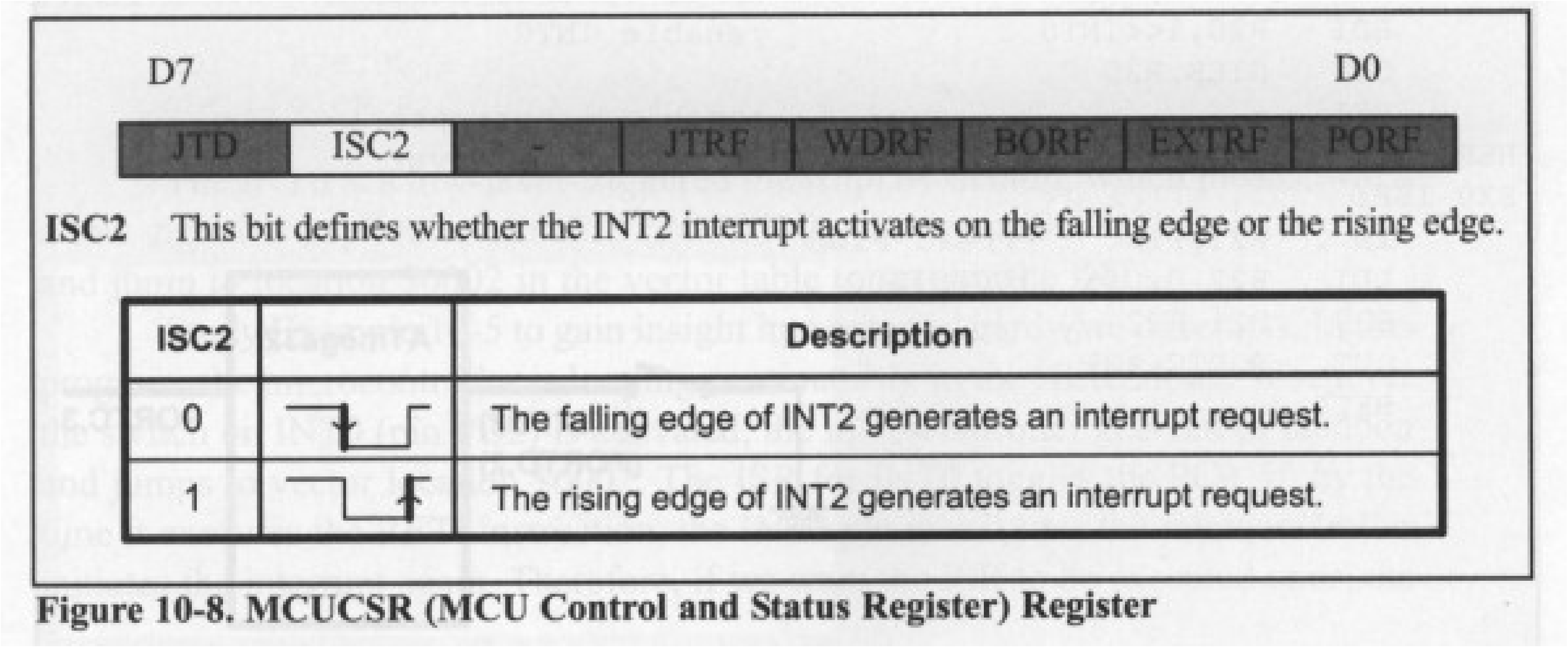
ISC01	ISC00		Description
0	0		The low level of INT0 generates an interrupt request.
0	1		Any logical change on INT0 generates an interrupt request.
1	0		The falling edge of INT0 generates an interrupt request.
1	1		The rising edge of INT0 generates an interrupt request.

ISC11, ISC10 These bits define the level or edge that activates the INT1 pin.

ISC11	ISC10		Description
0	0		The low level of INT1 generates an interrupt request.
0	1		Any logical change on INT1 generates an interrupt request.
1	0		The falling edge of INT1 generates an interrupt request.
1	1		The rising edge of INT1 generates an interrupt request.

Figure 10-7. MCUCR (MCU Control Register) Register

Configuring the INT2 using D6 in MCUCSR



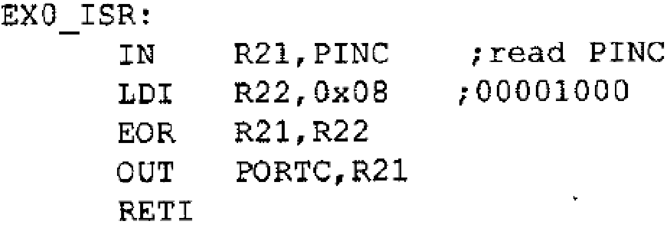
D7							D0
JTD	ISC2	-	JTRF	WDRF	BORF	EXTRF	PORF

ISC2 This bit defines whether the INT2 interrupt activates on the falling edge or the rising edge.

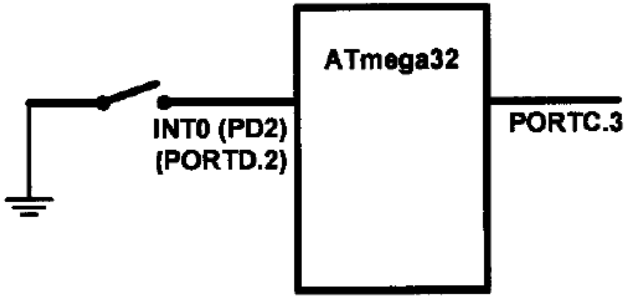
ISC2		Description
0	↓	The falling edge of INT2 generates an interrupt request.
1	↑	The rising edge of INT2 generates an interrupt request.

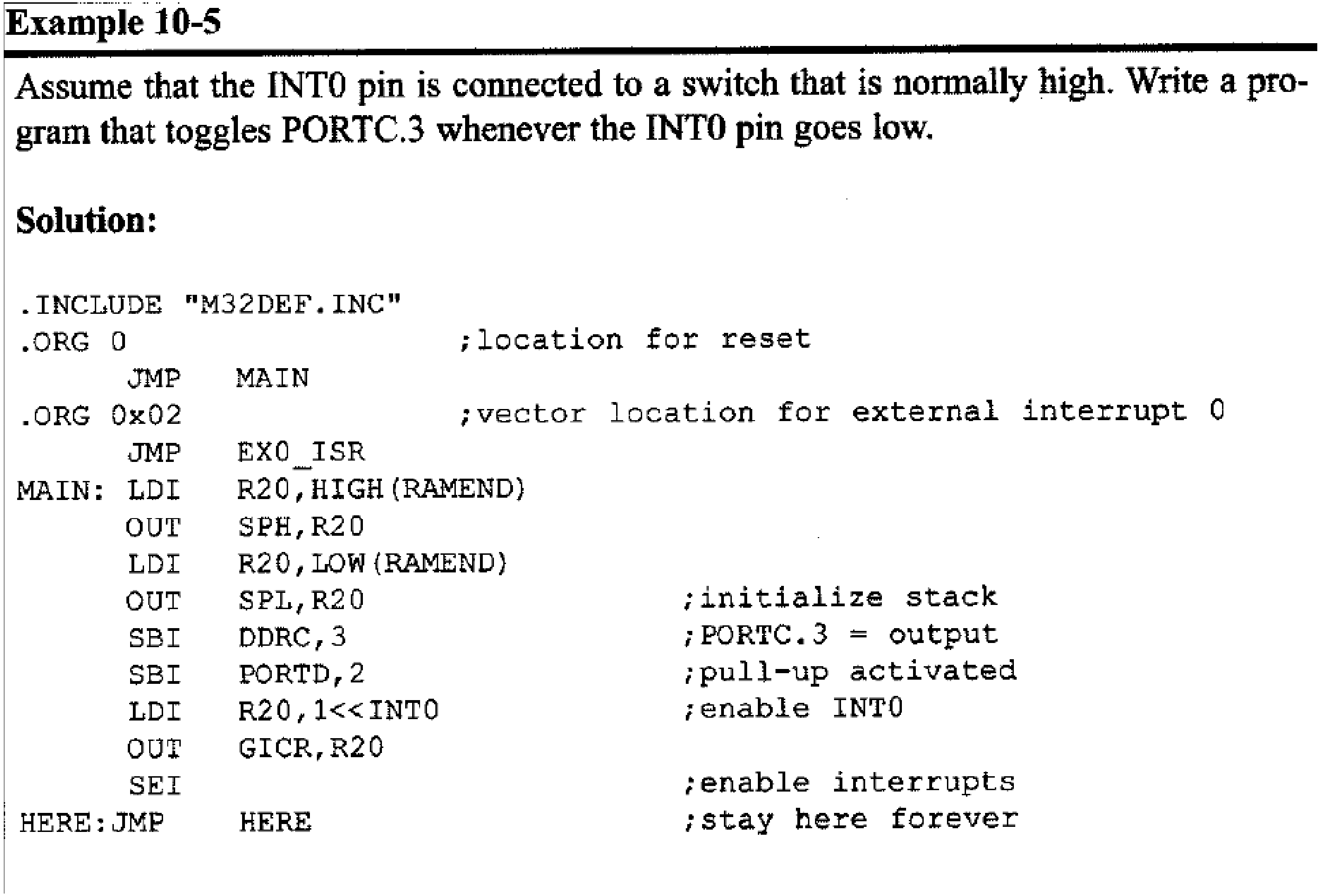
Figure 10-8. MCUCSR (MCU Control and Status Register) Register

AVR External Interrupt Program - An Example



```
EX0_ISR:
      IN     R21,PINC     ;read PINC
      LDI    R22,0x08    ;00001000
      EOR    R21,R22
      OUT    PORTC,R21
      RETI
```





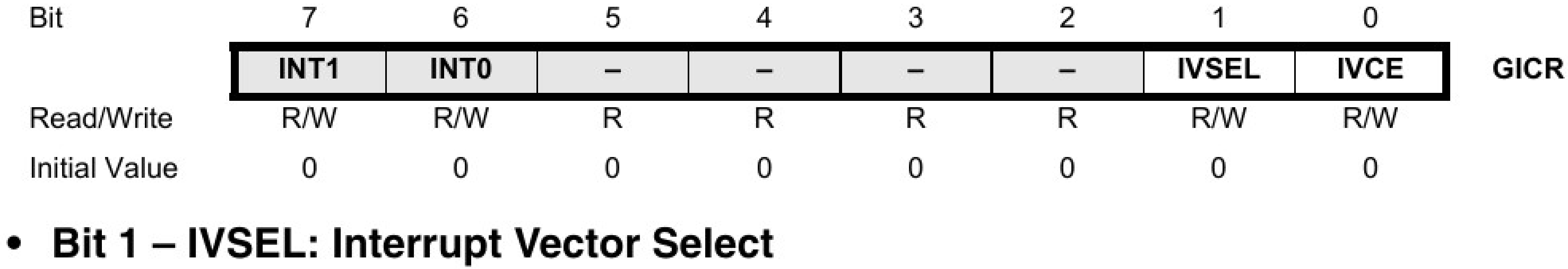
Example 10-5

Assume that the INT0 pin is connected to a switch that is normally high. Write a program that toggles PORTC.3 whenever the INT0 pin goes low.

Solution:

```
.INCLUDE "M32DEF.INC"
.ORG 0                     ;location for reset
      JMP   MAIN
.ORG 0x02                  ;vector location for external interrupt 0
      JMP   EX0_ISR
MAIN: LDI   R20,HIGH(RAMEND)
      OUT   SPH,R20
      LDI   R20,LOW(RAMEND)
      OUT   SPL,R20                     ;initialize stack
      SBI   DDRC,3                      ;PORTC.3 = output
      SBI   PORTD,2                     ;pull-up activated
      LDI   R20,1<<INT0                 ;enable INT0
      OUT   GICR,R20
      SEI                               ;enable interrupts
HERE:JMP    HERE                        ;stay here forever
```

Data General Interrupt Control Register - GICR

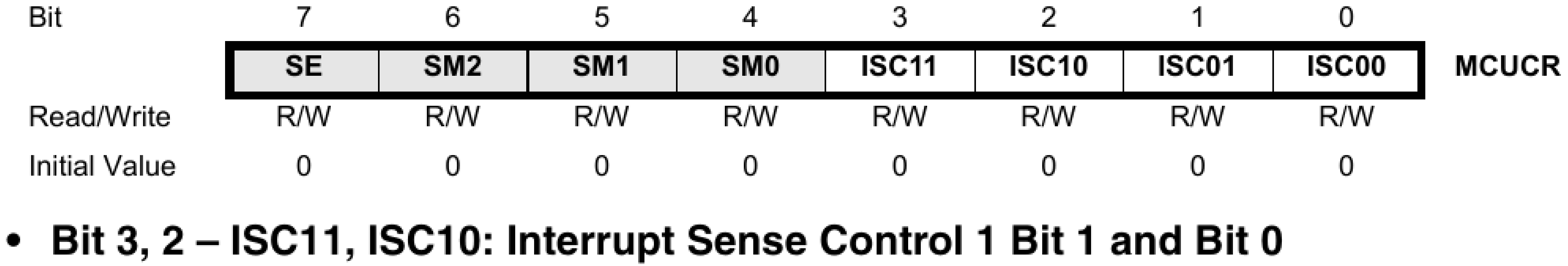


Bit	7	6	5	4	3	2	1	0	
	INT1	INT0	–	–	–	–	IVSEL	IVCE	GICR
Read/Write	R/W	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 1 – IVSEL: Interrupt Vector Select**

MCU Control Register (MCUCR)

➢

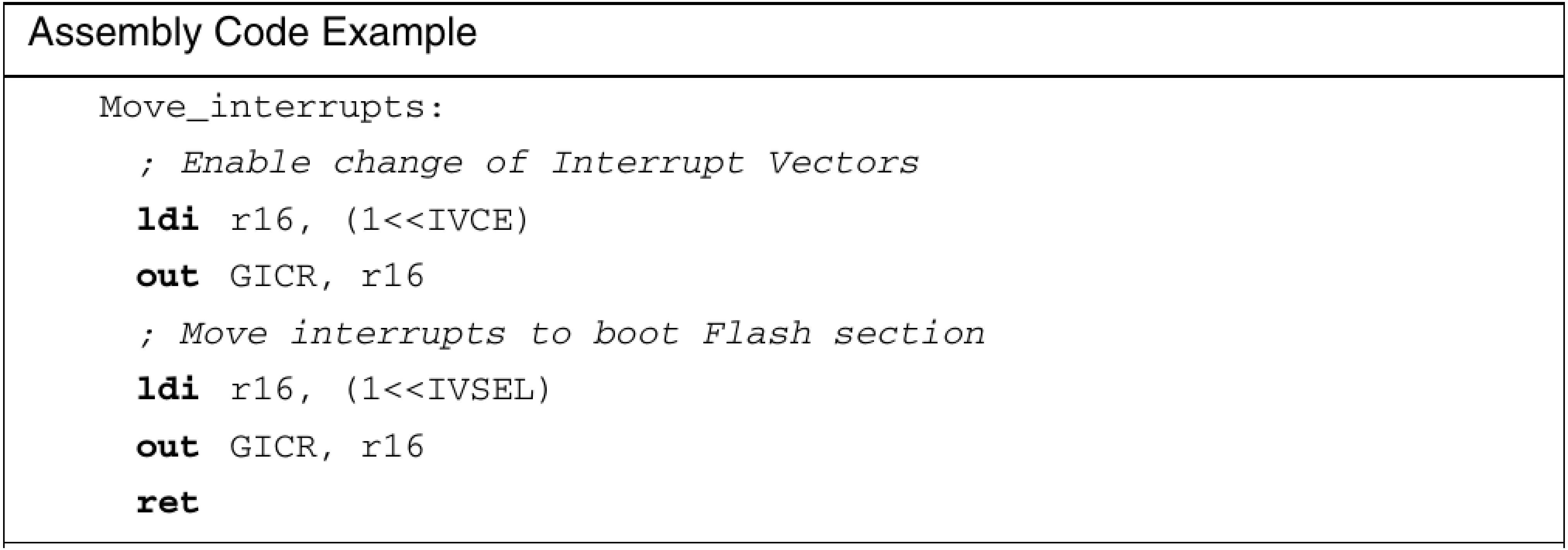


Bit	7	6	5	4	3	2	1	0	
	SE	SM2	SM1	SM0	ISC11	ISC10	ISC01	ISC00	MCUCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 3, 2 – ISC11, ISC10: Interrupt Sense Control 1 Bit 1 and Bit 0**

Interrupt Through C - Interfacing

- Interrupt through C - Interfacing

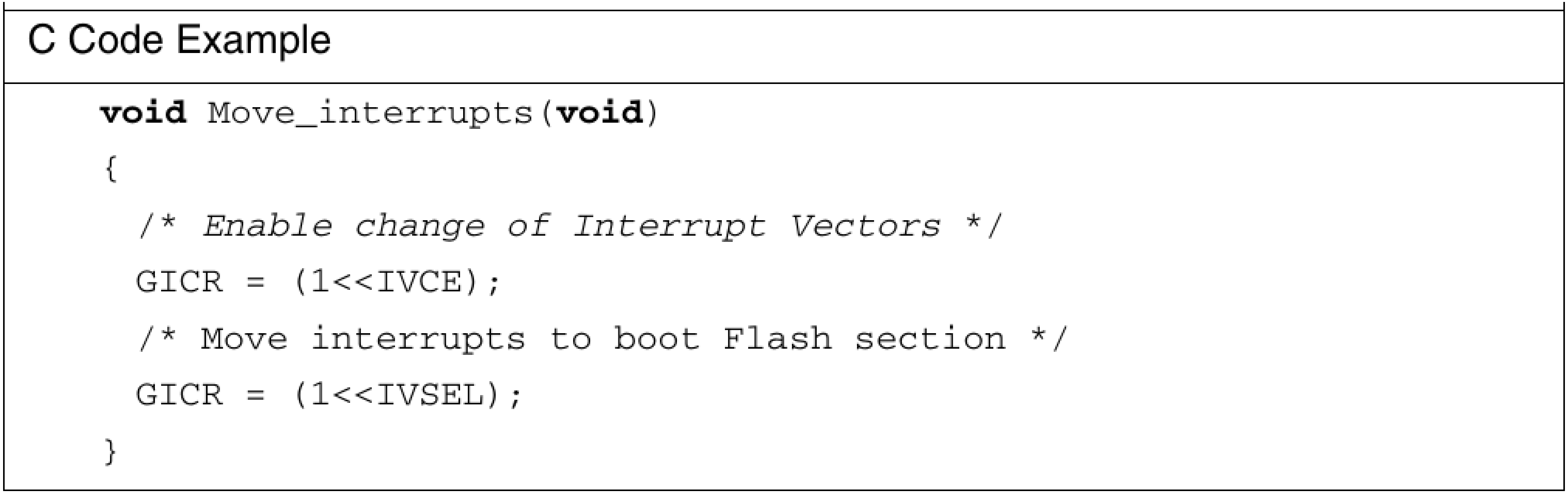


Assembly Code Example

```
Move_interrupts:
  ; Enable change of Interrupt Vectors
  ldi r16, (1<<IVCE)
  out GICR, r16
  ; Move interrupts to boot Flash section
  ldi r16, (1<<IVSEL)
  out GICR, r16
  ret
```

GPRs & Pointer Registers

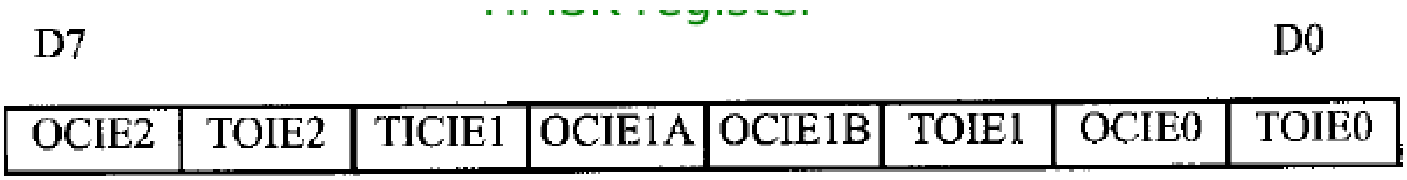
- Interrupt in AVR



C Code Example

```
void Move_interrupts(void)
{
  /* Enable change of Interrupt Vectors */
  GICR = (1<<IVCE);
  /* Move interrupts to boot Flash section */
  GICR = (1<<IVSEL);
}
```

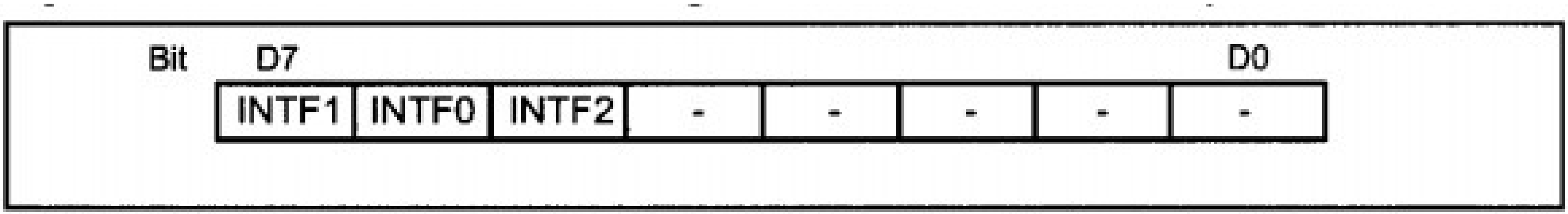
Chapter 10: AVR Interrupt Programming



TIMSK register

D7							D0
OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0

Chapter 4



Bit	D7							D0
	INTF1	INTF0	INTF2	-	-	-	-	-

Figure 10-9. GIFR (General Interrupt Flag Register) Register