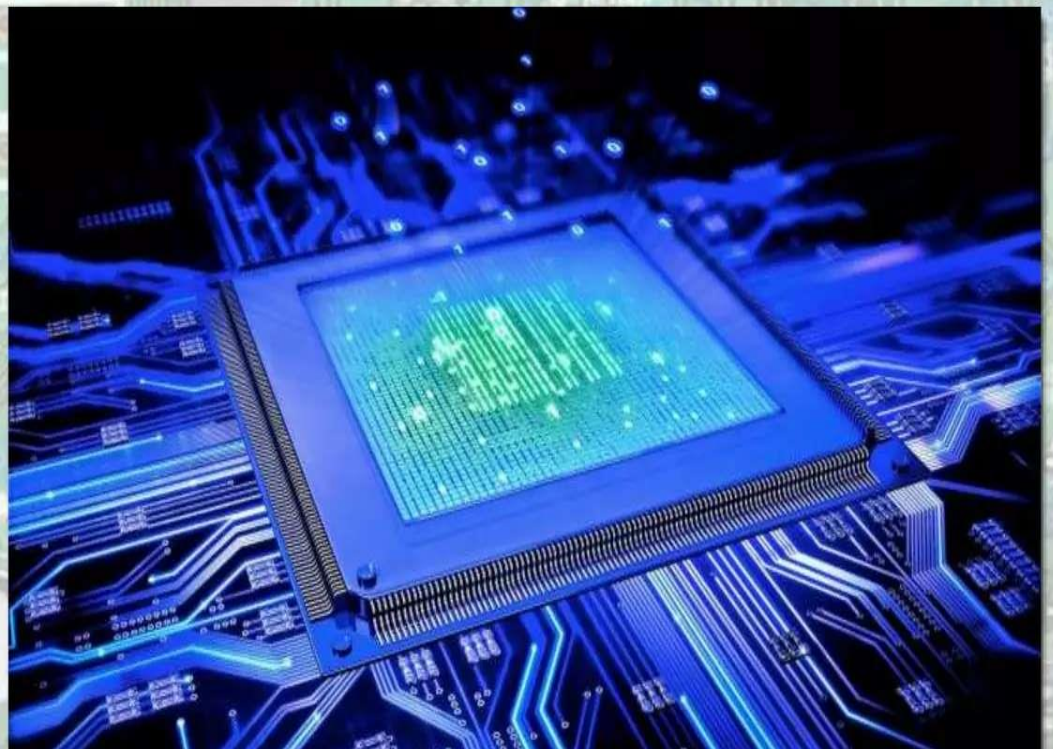
Embedded Systems

5041



Microcontroller

- A Microcontroller is a **programmable digital processor with necessary peripherals.**
- Both microcontrollers and microprocessors are **complex sequential digital circuits** meant to carry out **job according to the program / instructions**
- A microcontroller can be compared to a Swiss knife with **multiple functions incorporated in the same IC.**

AVR Microcontrollers & Assembly Language Programming

MODULE I

AVR Microcontrollers & Assembly Language Programming

- **AVR microcontrollers and its architecture**
  - You learn about the basic features and architecture of avr microcontrollers
- **AVR assembly language programming**
  - You learn the instruction set and assembly language programming(ALP) using those instructions

Microprocessors vs. Microcontrollers

MICROPROCESSORS	MICROCONTROLLERS
❑ Need external memory for program/ data storage	❑ Microcontrollers have on chip memory for storage of data/program
❑ Requires additional interfacing ICs to interface peripherals with microprocessor	❑ Does not need much interfacing Ics. It can be thought of as a microprocessor with built-in peripherals.
❑ Microprocessor can have much higher computing power than microcontrollers.	❑ In general microcontrollers are developed for less computationally complex applications
❑ Microprocessors can run at a clock speed much higher than that of microcontrollers	❑ Clock speed of microcontrollers is limited to a few tens of MHz.

Popular manufacturers of microcontroller

- Altera
- Analog Devices
- Atmel
- Freescale
- Intel
- Microchip(PIC)
- National Semiconductors
- Texas Instruments

How do you choose a microcontroller for your project from this long list ?

MICROCONTROLLER SELECTION

See the list: List of common microcontrollers

What is AVR ?

- **AVR** is a family of microcontrollers developed by **Atmel**.
- These are modified Harvard architecture 8-bit RISC single-chip microcontrollers.
- The basic architecture of AVR was designed by two students of Norwegian Institute of Technology and was bought by Atmel

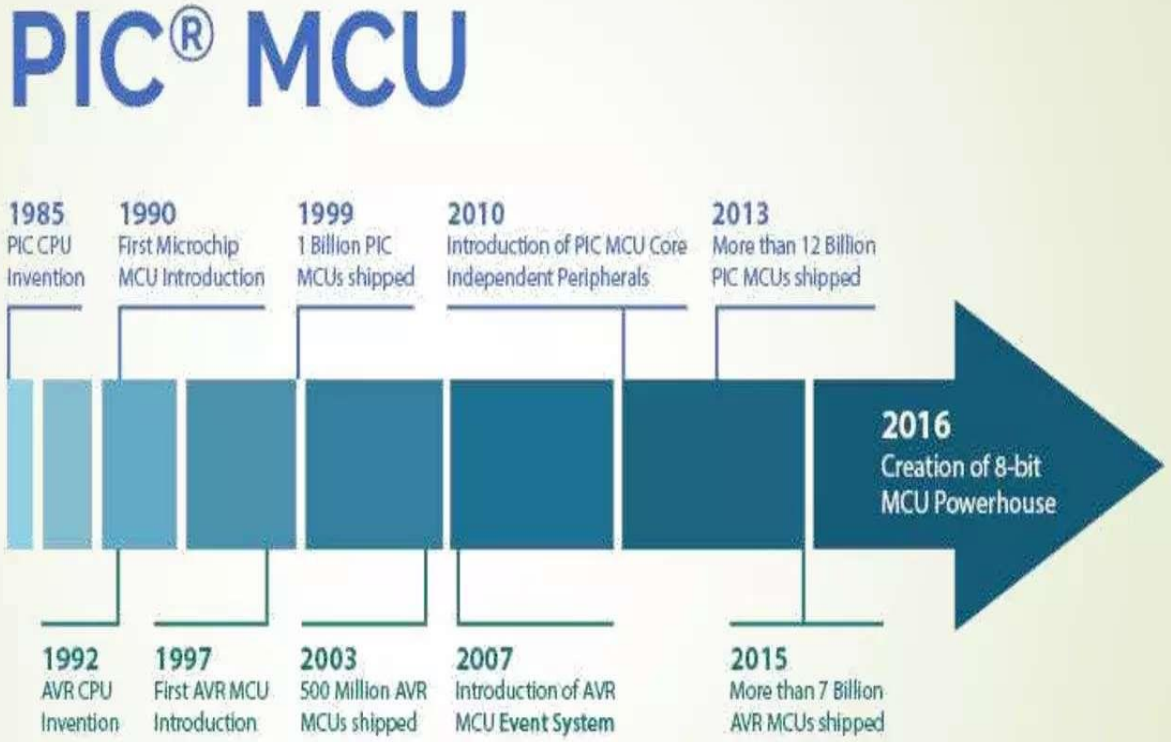
What is a Harvard architecture and RISC??

About Atmel

- Atmel is an American based designer and manufacturer of semiconductors
- It serves application including consumer, communication, automotive, computer networking, medical and aerospace.
- In 2016 Microchip acquired Atmel
- Atmel is now knows as,

ATMEL VISIT

Evolution of AVR microcontroller



Features of AVR microcontrollers

Standard features

- 8-bit RISC single-chip microcontroller
- Harvard architecture
- On-chip program ROM, data RAM, data EEPROM
- Timers and I/O ports

Additional features

- ADC, PWM
- Different serial interfaces like USART, SPI, I2C, CAN, USB etc.

AVR Family

AVR Product Families

tinyAVR® MCUs

tinyAVR microcontrollers (MCUs) are optimized for applications that require performance, power efficiency and ease of use in a small package. All tinyAVR devices are based on the same architecture and compatible with other AVR® devices. The integrated ADC, DAC, Comparators, EEPROM memory and brown-out detector let you build applications without adding external components. tinyAVR devices also offer Flash memory and on-chip debug for fast, secure and cost-effective in-circuit upgrades that significantly cut your

megaAVR® MCUs

megaAVR microcontrollers (MCUs) are the ideal choice for designs that need some extra muscle. For applications requiring large amounts of code, megaAVR devices offer substantial program and data memories with performance up to 20 MIPS. Meanwhile, innovative Atmel picoPower® technology helps minimize power consumption. All megaAVR devices offer self-programmability for fast, secure, cost-effective in-circuit upgrades. You can even upgrade the Flash memory while running your application.

AVR® XMEGA MCUs

AVR XMEGA microcontrollers deliver the best possible combination of real-time performance, high integration and low power consumption for 8/16-bit MCU applications.

AVR MCU List

AVR Family

AVR can be classified into three groups

- Tiny
- Mega
- xmega

AVR Family

tinyAVR

- 0.5–16 kB program memory
- 6–32-pin package
- Limited peripheral set

AVR Family

megaAVR

- 4–512 kB program memory
- 28–100-pin package
- Extended instruction set (multiply instructions and instructions for handling larger program memories)
- Extensive peripheral set

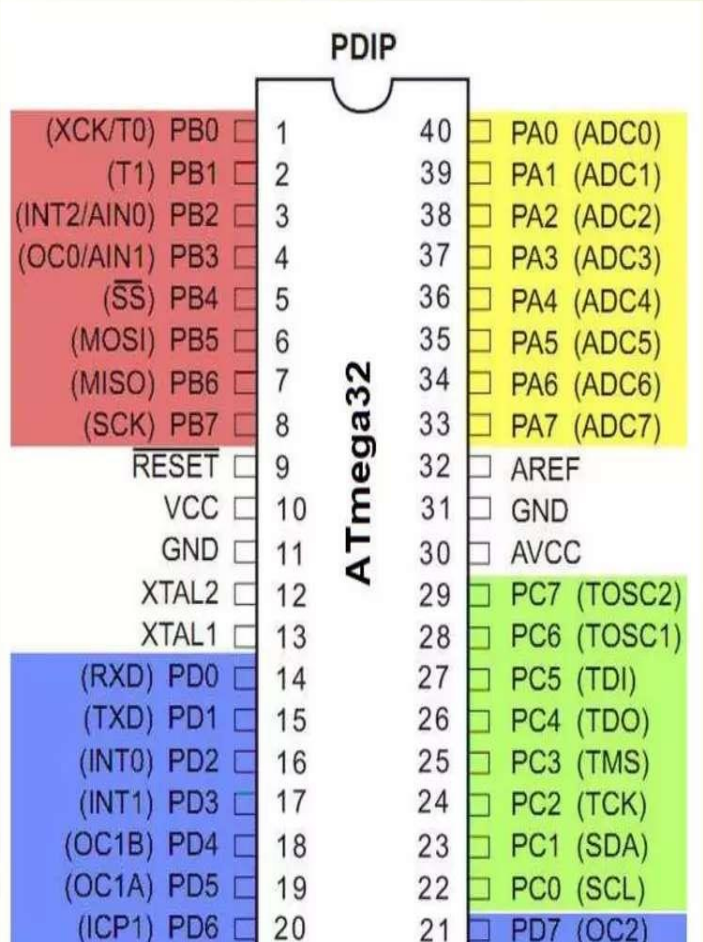
AVR Family

xmegaAVR

- 16–384 kB program memory
- 44–64–100-pin package (A4, A3, A1)
- Extended performance features, such as DMA, “Event System”, and cryptography support.
- Extensive peripheral set with ADCs

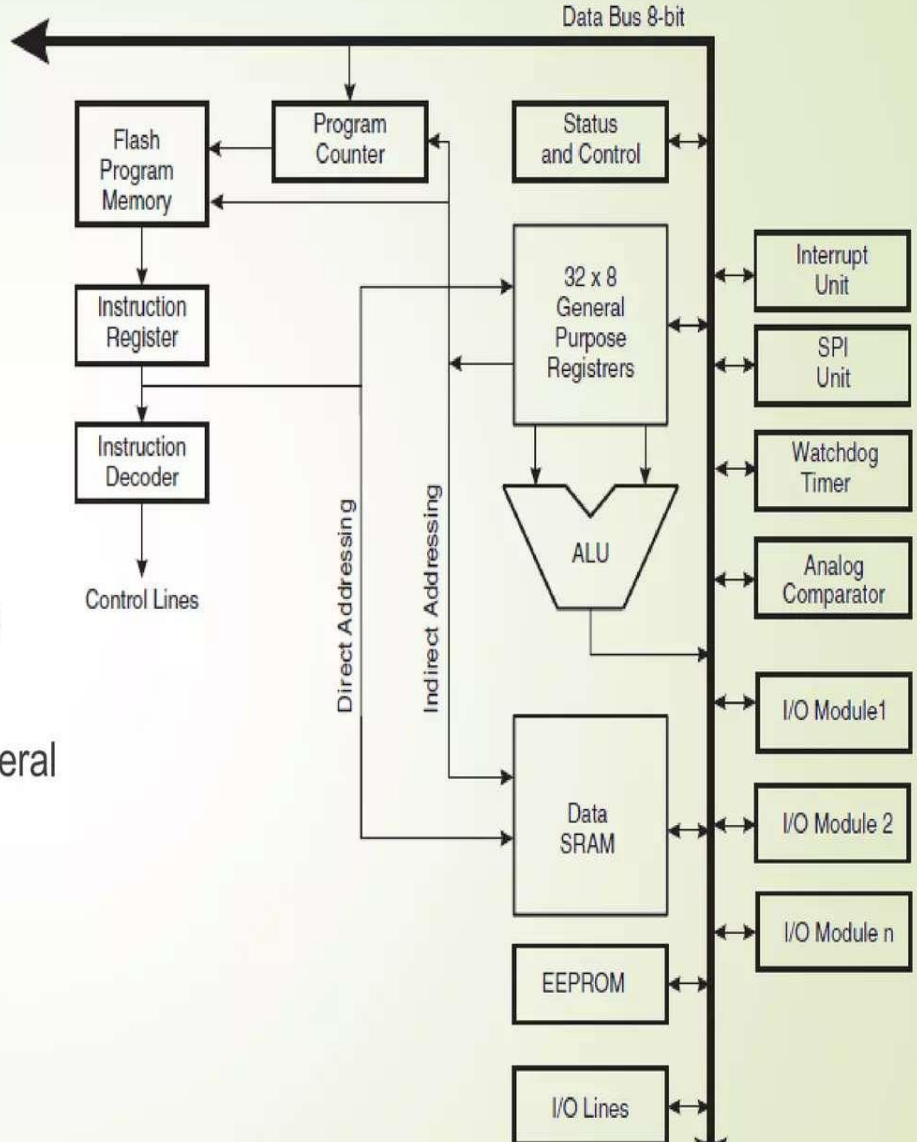
Pinout of ATmega 32

- PA0-PA7- PortA
- PB0-PB7- PortB
- PC0-PC7- PortC
- PD0-PD7- PortD
- ADC0-ADC7- Analog to Digital Converter
- XTAL1-XTAL2- External crystal oscillator
- TXD,RXD, XCK-USART pins
- INT0,INT1-External interrupt pins
- AIN0,AIN1- Analog comparator



AVR architecture

- Harvard architecture-supports parallelism
- Separate memories for program and data
- Next instruction is fetched while current is executed
- Fast access register file with 32x8-bit general purpose register,

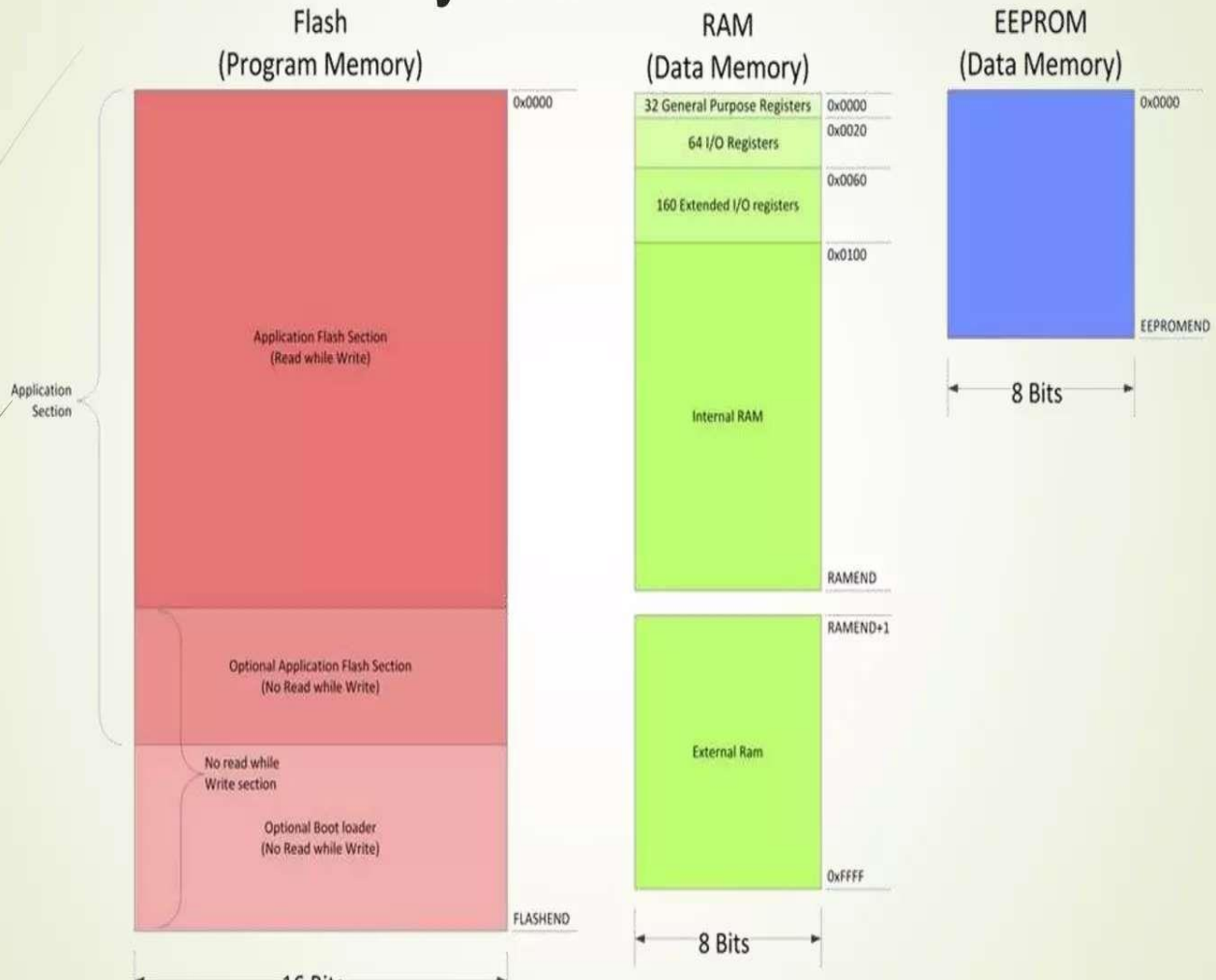


AVR Memory architecture

An AVR microcontroller has three types of memory

- Flash : Non volatile memory for storing the program. ATmega32 has **32KB of Program memory.**
- RAM : Volatile memory for storing the runtime state of the program being executed. ATmega32 has **2KB bytes of RAM memory.**
- EEPROM : Electrically Erasable Programmable ROM. Is a non volatile memory. Individual bytes can be erased and rewritten. Used to store semi permanent data. ATmega32 has **1KB of data EEPROM.**

AVR Memory architecture

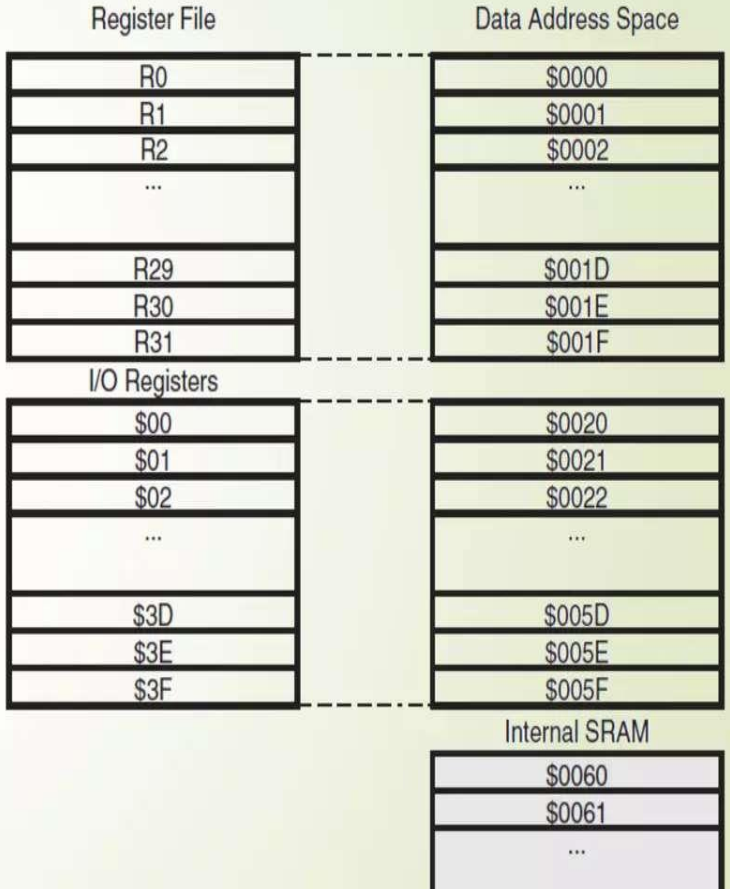


AVR Memory architecture

Data RAM

The data memory comprises of three sections

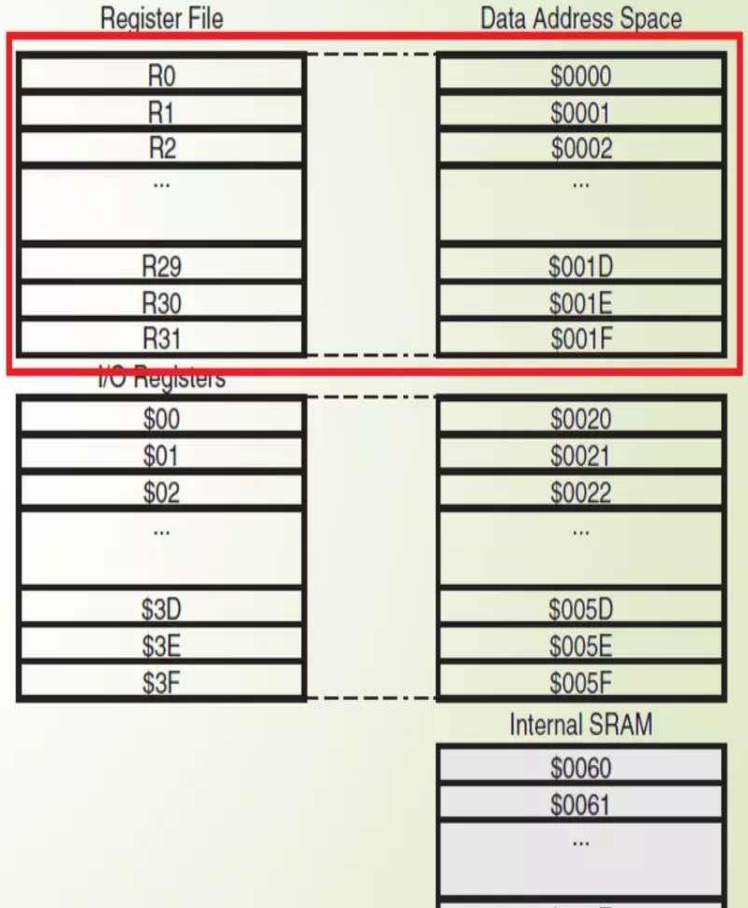
- **G**eneral **P**urpose **R**egisters (GPRs)
- I/O memory(Special function Registers)
- Internal data SRAM



Data RAM

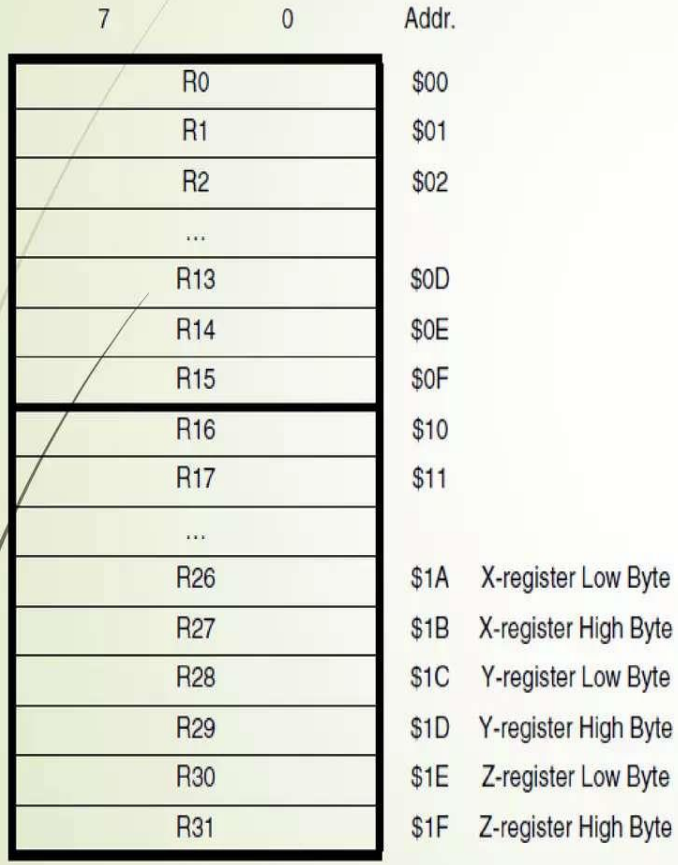
General Purpose Registers(GPRs)

- There are **32** General purpose registers in ATmega32
- **Numbered from R0 to R31**
- Located in the lowest location of memory address
- All registers are 8 bit
- Used by CPU to store data temporarily
- Can be used by all arithmetic and logic operations



Data RAM

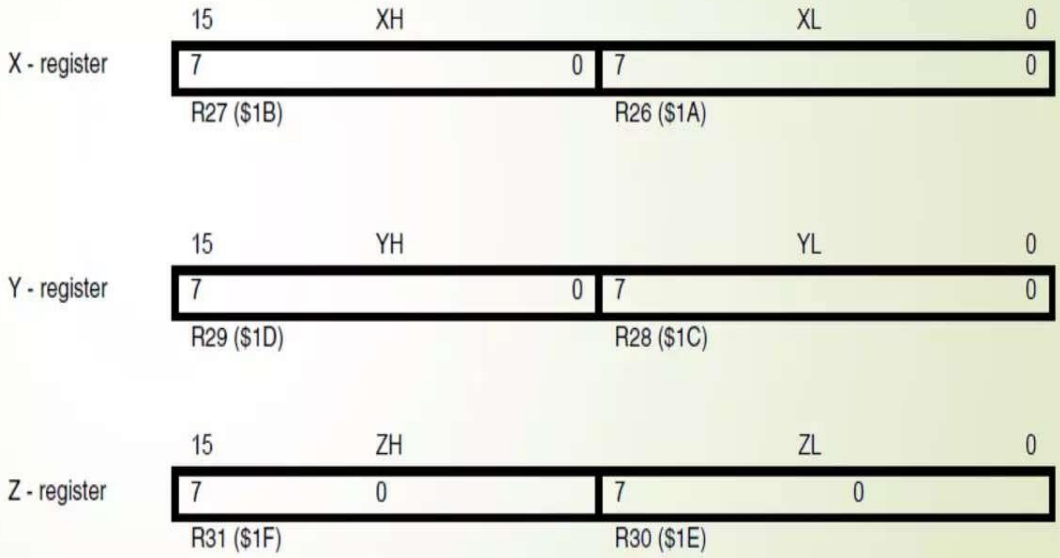
General Purpose Registers(GPRs)



- The GPRs take location address from $00-$FF in the data memory space regardless of the AVR chip number.
- Register operations are faster.
- Fetching operands from register, performing the operation and storing the result in memory happens in one clock cycle
- Registers from R26 to R31 can be paired as three 16 bit registers

AVR General Purpose Registers

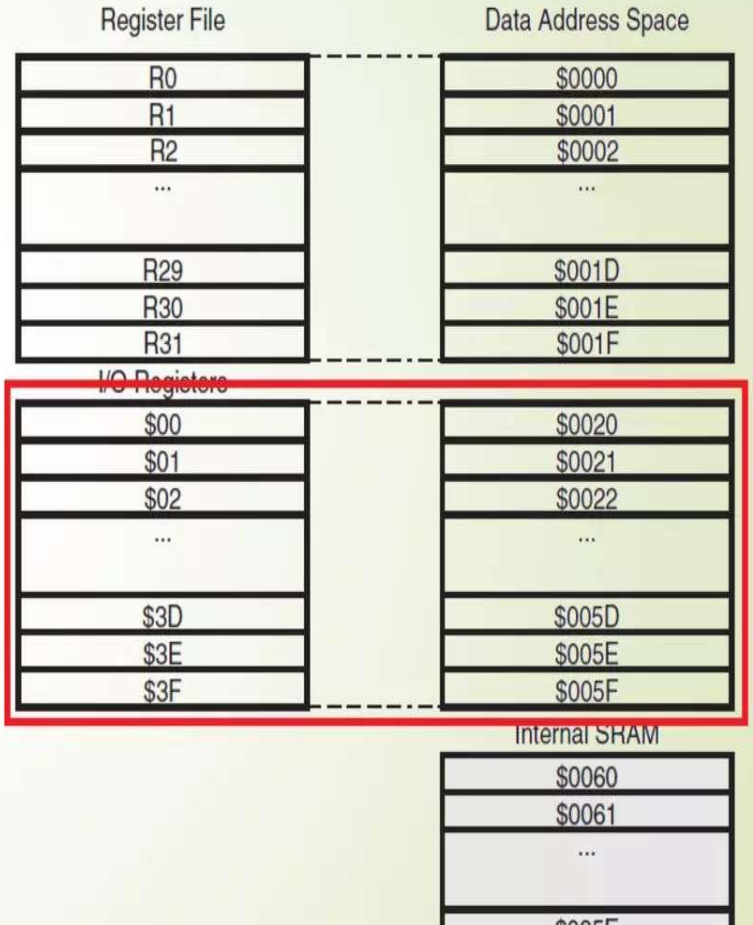
- R26-R27 → X register
- R28-R29 → Y register
- R30-R31 → Z register
- These registers can be used for storing (16-bit) address of memory locations for indirect addressing of data



Data RAM

I/O Memory

- Dedicated to special functions such as status register, timer, serial communication etc.
- Function of an I/O register and its address is fixed by the CPU designer and cannot be changed.
- Number of I/O registers can vary with different microcontroller
- However, all AVRs have at least 64bytes of I/O memory.
- ATmega32 has 64 bytes of I/O memory



Data RAM

I/O Memory

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

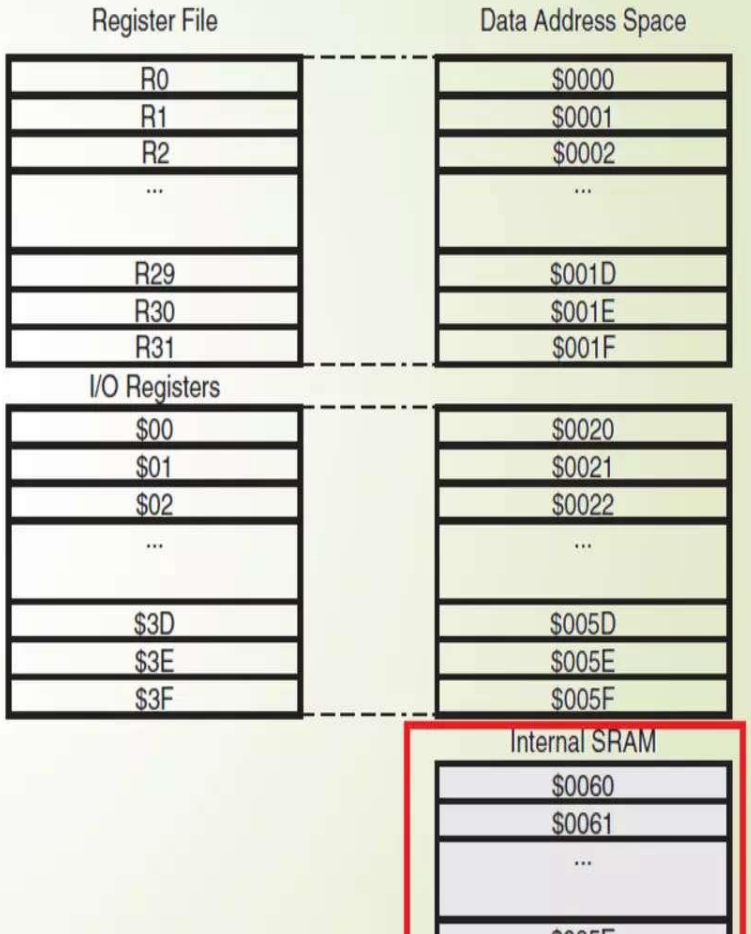
Address comparison

Status Register

Data RAM

Internal data SRAM

- Used for storing data and parameters
- Generally called '*scratch pad*'
- Each location can be accessed by its address.
- 8 bit wide.
- Any data of 8 biy wide can be stored in it, may it be data or address.
- Size of SRAM can vary from chip to chip.
- ATmega32 has 2KB of data SRAM.



AVR Memory

Memory of typical microcontrollers

	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

ATmega32

Parameter Name	Value
Program Memory Type	Flash
Program Memory (KB)	32
CPU Speed (MIPS)	16
RAM Bytes	2,048
Data EEPROM (bytes)	1024
Digital Communication Peripherals	1-UART, 1-SPI, 1-I2C
Capture/Compare/PWM Peripherals	1 Input Capture, 1 CCP, 4PWM
Timers	2 x 8-bit, 1 x 16-bit
Comparators	1
Temperature Range (C)	-40 to 85
Operating Voltage Range (V)	2.7 to 5.5
Pin Count	44
Cap Touch Channels	16

ATmega32 Data Sheet

AVR Assembly language programming

AVR Assembly language programming

- Instruction set
- Programming steps
- AVR Studio 5

Instruction Set

- Data Transfer instructions
- Arithmetic and Logic instructions
- Branch instructions
- Bit and Bit-Test instructions
- MCU control instructions

Data transfer instructions
MOV

- To copy data from one register to another register
- Data can be copied only between GPRs (R0-R31)
- Syntax : **MOV Rd, Rr**
- Example : MOV R10, R20

Data transfer instructions
LDI

- Load immediate data
- To copy 8-bit (immediate)data into General purpose registers
- The word immediate indicates a value that must be provided right there with the instruction
- Syntax : **LDI Rd, k**
- k is an 8-bit value that can be 0-255 in decimal
- Rd is R16 to R31
- Example : LDI R18, $4A

Beware...!

You cannot use LDI instruction to load immediate data to registers R0 to R15

Data transfer instructions
LDS

- Load Direct from Data Space
- To load 1 byte from an address in the data memory to General purpose register
- Syntax : **LDS Rd, K**
- Load Rd with **the content of location k**
- Rd is R0 to R31
- Example : LDS R5, $202
- Location in data memory can be any part of the data space. It can be an I/O register, data SRAM or a GPR.

Address	Data
$200	$13
$201	$FE
$202	**$6C**
$203	$0D
$204	$E3
$205	$19
$206	$BB

Data transfer instructions
STS

- Store Direct to Data Space
- Syntax : **STS K, Rr**
- The content of register Rr is copied to the address location K.
- Rd is R0 to R31
- Example : STS $202, R4

Beware..!

You cannot copy an immediate value directly into SRAM location, but only via a GPR...!

What happens when you execute an instruction STS 0x1, R10 ???

Address	Data
$201	$FE
$202	$3D
$203	$0D
$204	$E3

Address	Data
$201	$FE
$202	**$A9**
$203	$0D
$204	$E3

Data Memory address Vs I/O address

- Each location in I/O memory has two addresses- **I/O address** and **Data memory address**
- The data memory address of an I/O location is unique
- I/O memory address is the address relative to the beginning of I/O, called I/O address
- In ATmega32, the data memory address can vary from $0000-$085F, but I/O address can range from $00 to $3F

I/O memory

Data transfer instructions

IN

- Load an I/O location data to GPR
- Syntax : **IN Rd, A**
- 0 ≤ A ≤ 63 (or $00-$3F)
- Rd is R0 to R31
- Example : IN R04, 0x3F

If both LDS and IN instructions copy data from data memory to GPR, then what is the difference between them?

BEFORE

R04

I/O Address	Data
$3C	$FE
$3D	$3D
$3E	$0D
$3F	$E3

AFTER

R04
$E3

I/O Address	Data
$3C	$FE
$3D	$3D
$3E	$0D
$3F	$E3

Data transfer instructions
OUT

- Store the GPR into an I/O register
- Syntax : **OUT A, Rr**
- 0 ≤ A ≤ 63 (or $00-$3F)
- Rr is R0 to R31
- Example : OUT $3C, R04

I/O Address	Data
$3C	$FE
$3D	$3D
$3E	$0D
$3F	$E3

I/O Address	Data
$3C	$AA
$3D	$3D
$3E	$0D
$3F	$E3

Data transfer instructions

LD

- Load Indirect
- To load 1 byte from an address pointed by the GPR pair(X,Y,Z)
- Syntax : **LD Rd,m**
- Load Rd with **the content of address pointed by m**
- **m** can be X,Y or Z
- Example : LD R5, Y
- Location in data memory can be any part of the data space. It can be an I/O register, data SRAM or a GPR.

Address	Data
$0232	$13
$0233	$FE
$0234	$6C
$0235	**$0D**
$0236	$E3
$0237	$19
$0238	$BB

Data transfer instructions

ST

- Load Indirect
- To load 1 byte from an address pointed by the GPR pair(X,Y,Z)
- Syntax : **ST m, Rr**
- Store the content of Rr to **the address pointed by m**
- **m** can be X,Y or Z
- Example : ST Y, R05
- Location in data memory can be any part of the data space. It can be an I/O register, data SRAM or a GPR.

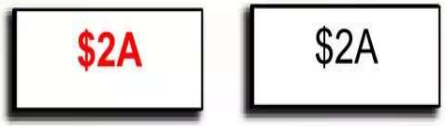
R27=YH	R26=YL
$02	$33

Address	Data
$0232	$13
$0233	**$0D**
$0234	$6C
$0235	$0D
$0236	$E3
$0237	$19
$0238	$BB

Data transfer instructions

Example No.1

```
LDI     R20, $18
MOV     R19,  R20
LDI     R20, $2A
MOV     R19, R20
STS         $237, R19
```

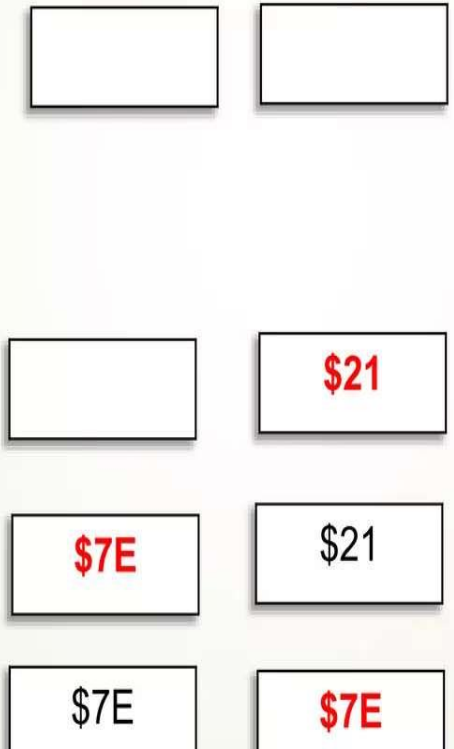


Address	Data
$0232	$13
$0233	$21
$0234	$6C
$0235	$0D
$0236	$E3
$0237	$2A
$0238	$BB

Data transfer instructions

Example No.2

```
LDS     R23, $233
LDI     R22,  $7E
MOV     R23, R22
STS     $237,R23
```



Address	Data
$0232	$13
$0233	$21
$0234	$6C
$0235	$0D
$0236	$E3
$0237	$7E
$0238	$BB

Data transfer instructions

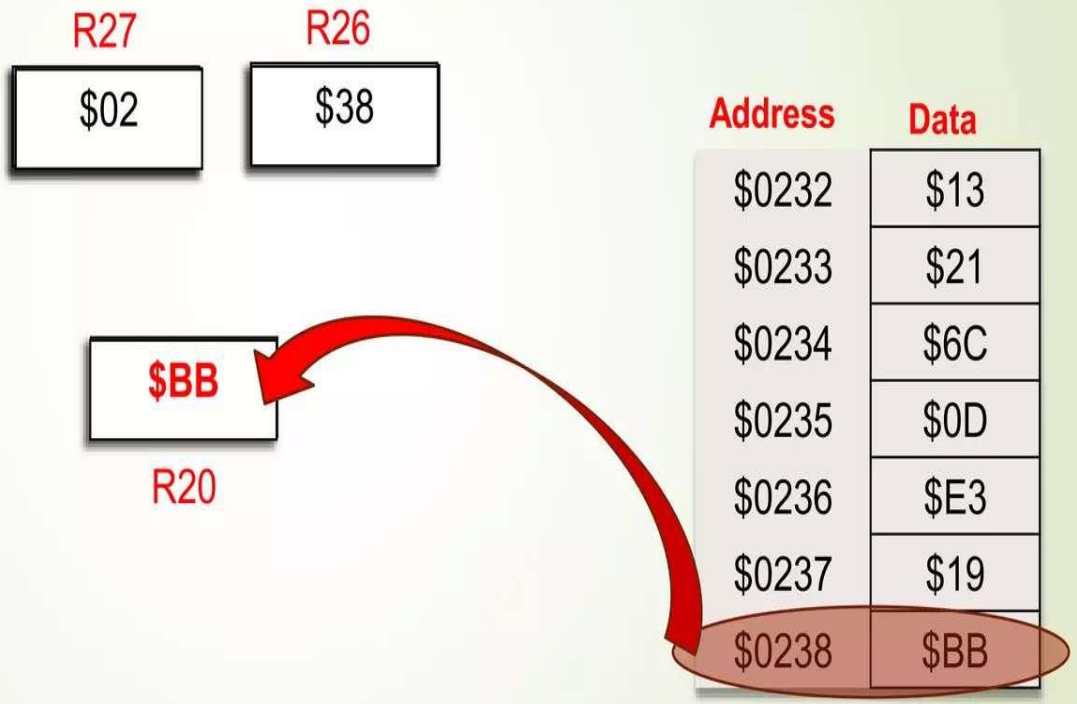
Example No.3

```
LDI   R28, $38
LDI   R29, $02
ST    Y, R20
```

Data transfer instructions

Example No.4

```
LDI     R26, $38
LDI     R27, $02
LD      R28, X
```



Data transfer instructions

Find the errors in the following instructions

LDI R20, $202	A 12 bit number cannot go into an 8 bit register
LDS R19, R16	LDS is used to load data from memory, not from another register-syntax error
LDI R09, $55	LDI cannot load an immediate data into registers from R0-R15
LDS R21, y	Wrong instruction. Use LD instead of LDS

AVR Status Register-SREG

- Flag register to indicate arithmetic conditions such as carry bit
- Flag register in AVR is called **Status Register – SREG**
- An 8-bit register
- In ATmega32 SREG has data memory address of $5F and I/O address of $3F
- The function of each bit is as follows...

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

SREG($5F)

I/O Memory

AVR Status Register-SREG

C- Carry flag

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

- C flag is set whenever there is a carry out from the D7th bit
- The flag is affected after an 8 bit addition or subtraction
- Is used to detect errors in unsigned arithmetic operation

AVR Status Register-SREG

Z- Zero flag

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

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

AVR Status Register-SREG

N- Negative flag

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

- Negative flag reflects the result of an arithmetic operation
- If D7th bit of the result is zero then N=0
- If the D7th bit is 1 then N=1

AVR Status Register-SREG

V- Overflow flag

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

- Whenever the result of a signed number operation is too large, and the high-order bit overflows into the sign bit(D7th bit), V flag is set.
- Used to detect errors in signed arithmetic operation

AVR Status Register-SREG

S- Sign flag

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

- This flag is the result of EX-Oring of N and V flags.

AVR Status Register-SREG

H- Half carry flag

D7	D6	D5	D4	D3	D2	D1	D0
I	T	H	S	V	N	Z	C

- Sets whenever there is a carry from D3 to D4 during ADD or SUB operation.
- Used in BCD arithmetic.
- Function same as that of Auxiliary carry flag in some microprocessors

Arithmetic and logic instructions
ADD

- Adds the content of two registers
- Syntax : **ADD Rd, Rr**
- Flags affected: C,Z,H, S,V,N
- *Example:* ADD R21,R22

	R21	R22
BEFORE	$15	$25
AFTER	**$3A**	$25

Arithmetic and logic instructions
ADC

- Adds the content of two registers along with carry
- Syntax : **ADC Rd, Rr**
- Flags affected: C,Z,H, S,V,N
- *Example:* ADC R21,R22

	R21	R22
BEFORE	$15	$25
AFTER	$3B	$25

I	T	H	S	V	N	Z	C
							1

Arithmetic and logic instructions
SUB

- Subtracts the content of source register from destination register and stores it back to destination register
- Syntax : **SUB Rd, Rr**
- Flags affected: Z,C,N,V,H
- *Example:* SUB R21,R22

	R21	R22
BEFORE	$34	$13
AFTER	**$11**	$13

Arithmetic and logic instructions
SUBI

- Subtracts a constant from register content
- Syntax : **SUB Rd, k**
- Flags affected: Z,C,N,V,H
- *Example:* SUB R16,$0A

	R16
BEFORE	$3A
AFTER	**$30**

Arithmetic and logic instructions
SBC

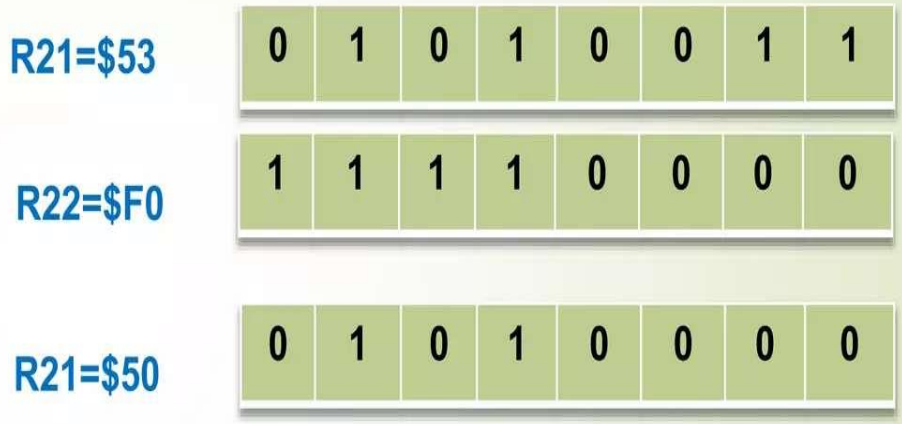
- Subtract the content of source register and carry from destination register
- Rd=Rd-Rr-C
- Syntax : **SBC Rd, Rr**
- Flags affected: C,Z,H, S,V,N
- *Example:* SBC R21,R22

	R21	R22
BEFORE	$25	$15
AFTER	$0F	$15

I	T	H	S	V	N	Z	C
							1

Arithmetic and logic instructions
AND

- Logical AND registers
- Syntax : **AND Rd, Rr**
- Flags affected: Z,V,N
- *Example:* AND R21,R22



Arithmetic and logic instructions

ANDI

- Logical AND register and a constant
- Syntax : **ANDI Rd, k**
- k is an 8 bit number
- Flags affected: Z,V,N
- *Example:* ANDI R21,$17

R21=$53

0	1	0	1	0	0	1	1

$17

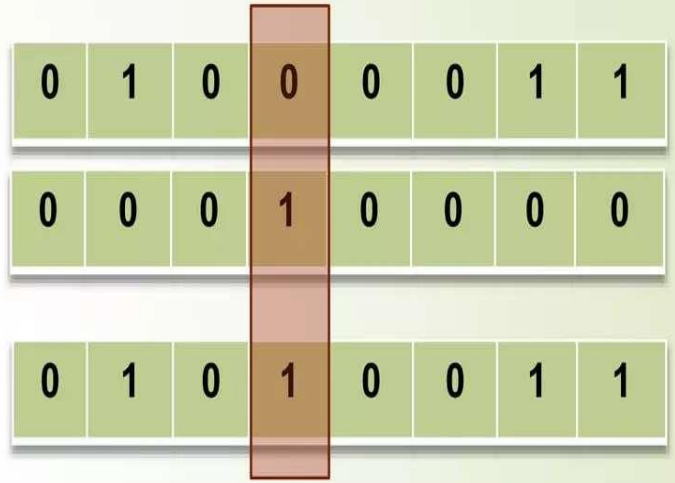
0	0	0	1	0	1	1	1

R21=$13

0	0	0	1	0	0	1	1

Arithmetic and logic instructions
OR

- Logical OR registers
- Syntax : **OR Rd, Rr**
- Flags affected: Z,V,N
- *Example:* OR R21,R22



Arithmetic and logic instructions
ORI

- Logical OR register and a constant
- Syntax : **ORI Rd, k**
- K is an 8 bit number
- Flags affected: Z,V,N
- *Example:* ORI R21,$17

R21=$53

0	1	0	1	0	0	1	1

$17

0	0	0	1	0	1	1	1

R21=$57

0	1	0	1	0	1	1	1

Arithmetic and logic instructions
EOR

- Exclusive ORing of registers
- Syntax : **EOR Rd, Rr**
- Flags affected: Z,V,N
- *Example:* EOR R21,R22

R21=$D3

1	1	0	1	0	0	1	1

R22=$A A

1	0	1	0	1	0	1	0

R21=$79

0	1	1	1	1	0	0	1

Arithmetic and logic instructions
COM

- 1's complement of registers
- Syntax : **COM Rd**
- Flags affected: Z,C,V,N
- *Example:* COM R21

R21=$55

BEFORE

0	1	0	1	0	1	0	1

R21=$AA

AFTER

1	0	1	0	1	0	1	0

Arithmetic and logic instructions
NEG

- 2's complement of registers
- Syntax : **NEG Rd**
- Flags affected: Z,C,V,N,H
- *Example:* NEG R21

R21=$55

BEFORE

0	1	0	1	0	1	0	1

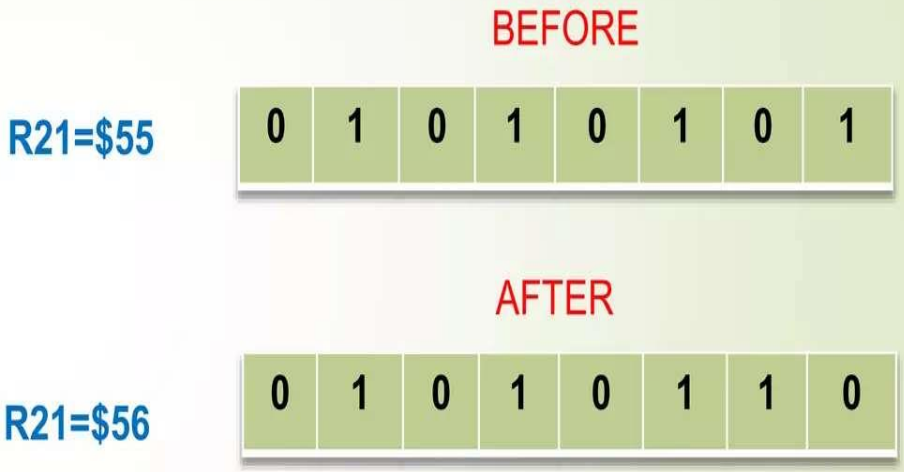
R21=$AB

AFTER

1	0	1	0	1	0	1	1

Arithmetic and logic instructions
INC

- Increment register value by 1
- Syntax : **INC Rd**
- Flags affected: Z,V,N
- *Example:* INC R21



Arithmetic and logic instructions
DEC

- Decrement register value by 1
- Syntax : **DEC Rd**
- Flags affected: Z,V,N
- *Example:* DEC R21

BEFORE

R21=$55

0	1	0	1	0	1	0	1

AFTER

R21=$54

0	1	0	1	0	1	0	0

Arithmetic and logic instructions
CLR

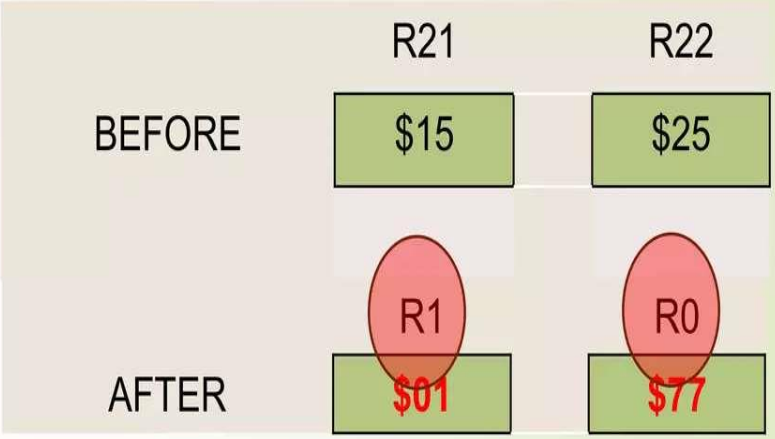
- Clear register
- Syntax : **CLR Rd**
- Flags affected: Z,V,N
- *Example:* CLR R21

Arithmetic and logic instructions
SER

- Set register
- Syntax : **SER Rd**
- Flags affected: None
- *Example:* SER R21

Arithmetic and logic instructions
MUL

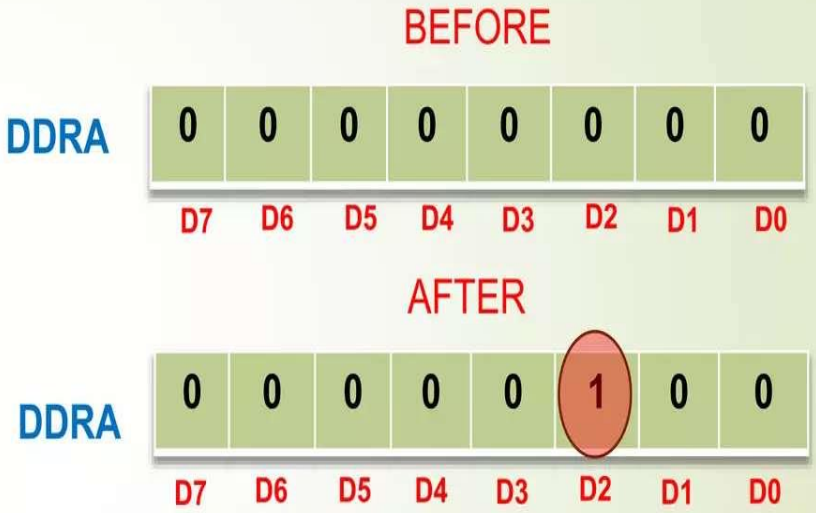
- Multiplies the content of two registers and stores the result in R0 and R1
- Syntax : **MUL Rd, Rr**
- Flags affected: C,Z
- *Example:* MUL R21,R22



Bit and bit-test instructions

SBI

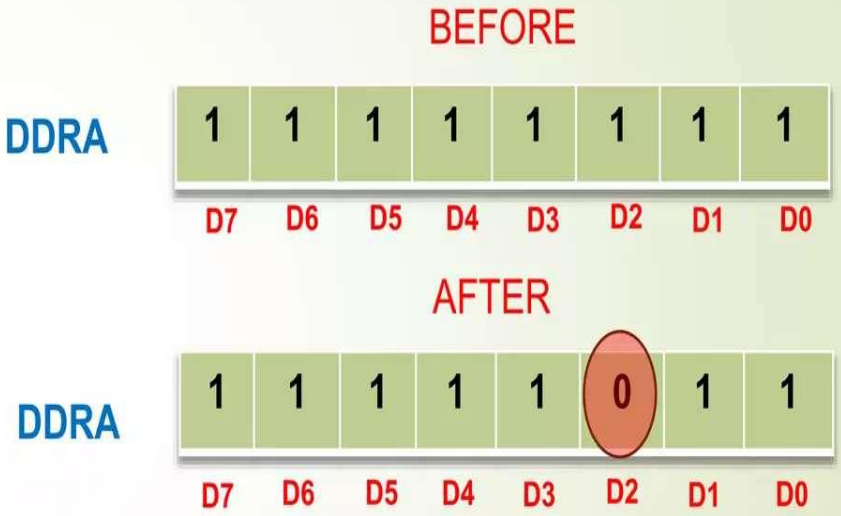
- Set bit in I/O register
- Syntax : SBI P, b
- Flags affected: None
- Example SBI DDRA, 2



Bit and bit-test instructions

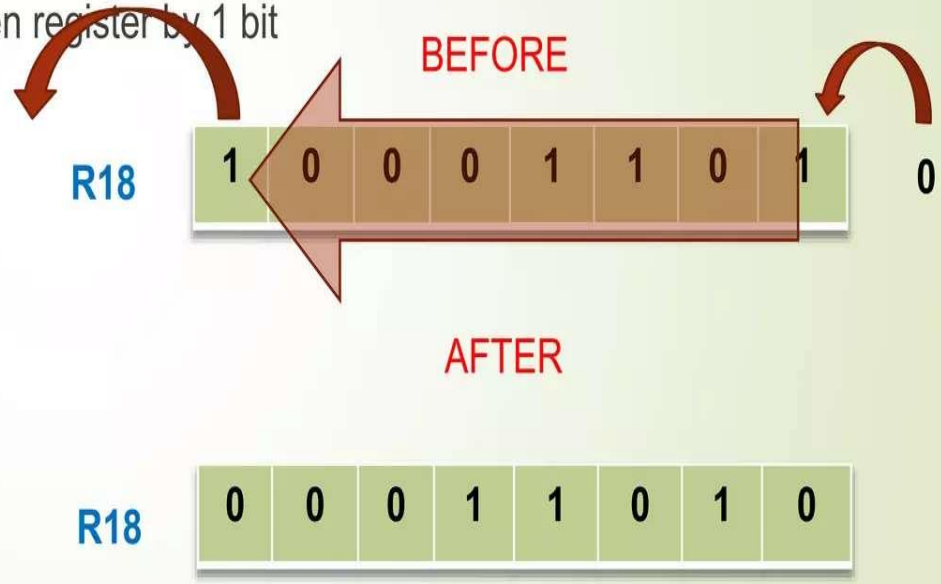
CBI

- Clear bit in I/O register
- Syntax : CBI P, b
- Flags affected: None
- Example CBI DDRA, 2



Bit and bit-test instructions
LSL

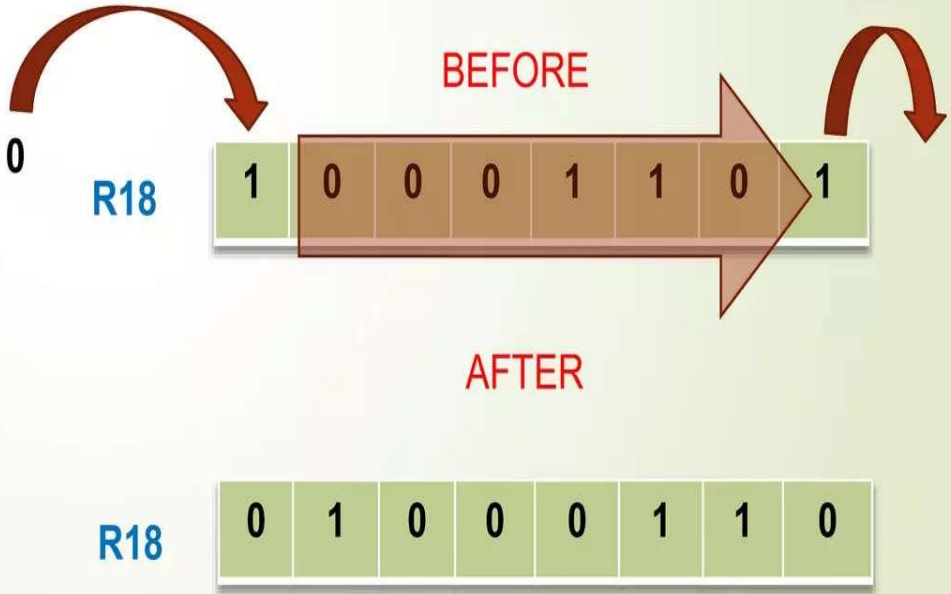
- Logical Shift Left
- Shifting the content of given register by 1 bit
- Syntax : LSL Rd
- Flags affected: Z,C,N,V
- Example LSL R18



Bit and bit-test instructions

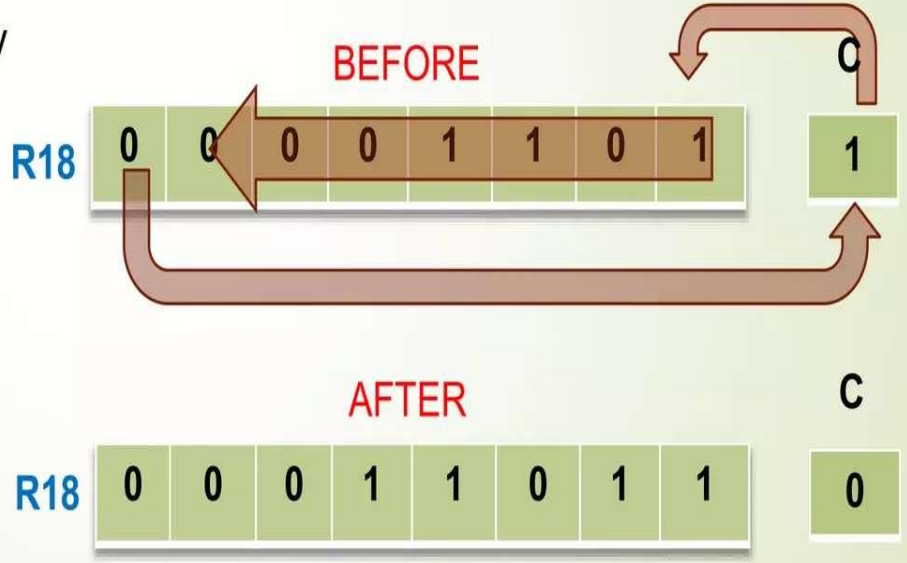
LSR

- Logical Shift Right
- Shifting the content of register right by 1 bit
- Syntax : LSR Rd
- Flags affected: Z,C,N,V
- Example LSR R18



Bit and bit-test instructions
ROL

- Rotate left through Carry
- Syntax : ROL Rd
- Flags affected: Z,C,N,V
- Example ROL R18

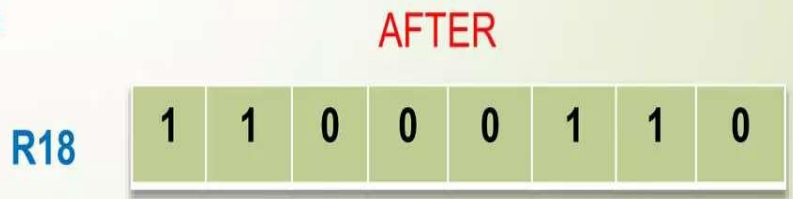


Bit and bit-test instructions

ASR

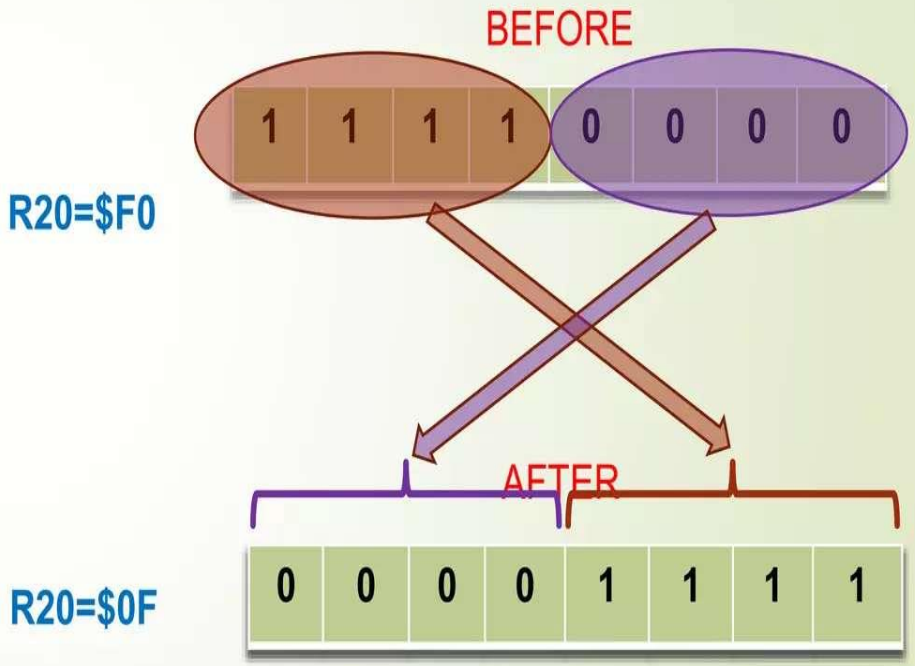
- Aritmetic Shift Right
- Shifting the content of register right by 1 bit
- Syntax : LSR Rd
- Example LSR R18
- Flags affected: Z,C,N,V **0**

- **The D7th bit is filled with the same value of D7th bit just before shifting**



Bit and bit-test instructions
SWAP

- Swap nibbles.
- When this instruction is executed on a register the lower 4 bits and upper 4 bits are exchanged ie, D3-D0 bits become D7-D4 and D7-D4 bits become D3-D0
- Syntax : SWAP Rd
- Flags affected: None
- Example: SWAP R20



Bit and bit-test instructions
SEC

- Set carry
- The carry bit in the status register is cleared
- No operands
- Flags affected : C
- Syntax : CLC

Bit and bit-test instructions
CLC

- Clear carry
- The carry bit in the status register is cleared
- No operands
- Flags affected : C
- Syntax : CLC

Instructions we learnt so far...

Data Transfer Instruction	Arithmetic and Logic instructions	Arithmetic and Logic instructions	Bit and Bit-Test instructions	MCU Control Instructions
MOV	ADD	COM	SBI	NOP
LDI	ADC	NEG	CBI	
LDS	SUB	INC	LSL	
STS	SUBI	DEC	LSR	
IN	SBC	CLR	ROL	
OUT	AND	SER	ASR	
LD	ANDI	MUL	SWAP	
ST	OR		SEC	
MOV	ORI		CLC	
	EOR			

MCU control instructions
NOP

- No operation is done
- No operand
- No flag is affected
- One unit of clock cycle is used out
- Can generate a delay

Some Examples...

Example No. 5

```
LDI R17, $33
LDI R18, $21
INC R17
INC R17
DEC R18
ADD R17,R18
STS $65, R17
```

Find the contents of the registers and Location $65

Some Examples...

Example No. 6

LDI R17, $9F

LDI R18, $61

CLR R19

ADD R17, R18

ROL R17

Find the contents of the registers

Some Examples...

Example No. 7

```
LDI R17, $90
LDI R18, $30
SWAP R17
SWAP R18
MUL R17, R18
MOV R17,R0
MOV R18,R1
```

Find the contents of the registers

Some Examples...

Example No. 8

LDI R17, $90

LDI R18, $30

ASR R17

ASR R18

COM R17

NEG R18

Some Examples...

Example No. 8

Write an assembly language program for subtracting two 8 bit numbers and store the difference in memory

Some Examples...

Example No. 9

Write an assembly language program for multiplying two 8 bit numbers and store the product in memory

Some Examples...

Example No. 10

Write an assembly language program to add two 16-bit numbers and store the sum and carry in memory

Branching instructions

- To change the sequential flow of program execution
- Classified as two types
  - Conditional Branching Instructions
  - Unconditional Branching Instructions

Conditional Branching instructions
BRNE

- Branch if NOT EQUAL
- It can be read as 'branch if not equal **to zero**'
- Checks whether Z flag is set(Z=1) and **branches if not (Z=0)**

```
Back:   ……
        ……
        ……
        DEC R16
        BRNE Back
        ……
```

Conditional Branching instructions
BREQ

- Branch if EQUAL
- It can be read as ‘branch if **equal to zero**’
- Checks whether Z flag is set(Z=1) and **branches if Z=1**

Back:	…….
	…….
	…….
	EOR R20, R21
	BREQ Back
	…….

Conditional Branching instructions
BRLO

- Branch if lower
- Checks the C flag and **branches if C=1**
- Remember the CP instruction
  - CP Rd, Rr
  - Rd=Rd-Rr
- After a compare instruction CP for unsigned numbers, if C=1, it means left hand operand was lower than the right hand operand

Back:	.....
	.....
	.....
	ADD R20, R21
	BRLO Back
	.....

Conditional Branching instructions
BRSH

- Branch if same or higher
- Checks the C flag and **branches if C=0**

```
Back:   .....
        .....
        .....
        ADD R20, R21
        BRSH Back
        .....
```

Conditional Branching instructions

Some other instructions

Mnemonic	Operands	Description	
BRLT	**K**	Branch if less than (signed)	Branch if S=1
BRGE	**k**	Branch if greater than or equal	Branch if S=0
BRVS	**k**	Branch if overflow flag is set	Branch of V=1
BRVC	**k**	Branch if overflow flag cleared	Branch if V=0

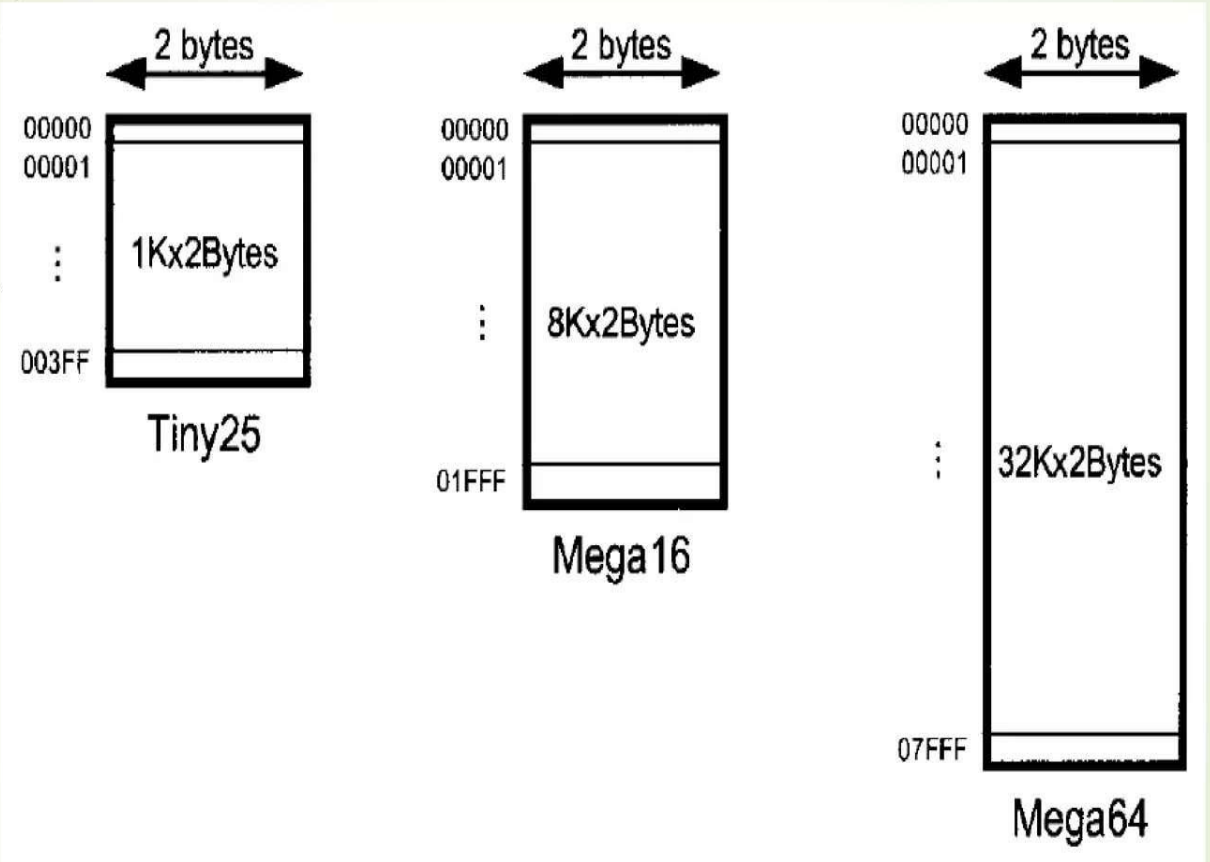
Program Counter

- A register used by the CPU to point to the address of the next instruction to be executed
- As CPU fetches a new instruction, PC is automatically incremented to point to the next instruction
- Wider the PC, the more memory locations a controller can address.

Program Counter
(Cntd.)

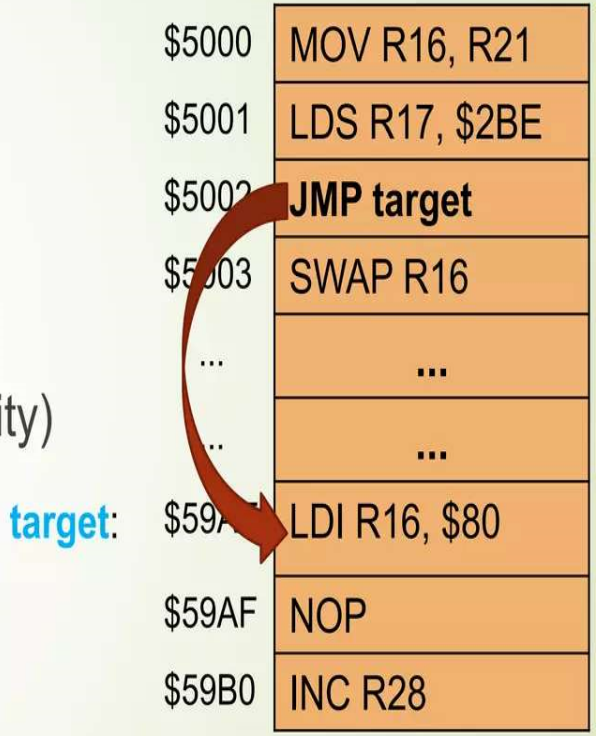
- In AVR, each memory location is two bytes wide
- In ATmega32 the 32KB memory is organised as 16K X 16
- And hence the PC is 14 bits wide
- The code address can range from 0000-$7FFF
- The PC in the AVR family can be upto 22 bit wide, hence it can access utmost 4M locations. Since each location is 2B wide, the total program memory can be 8MB.

Program Counter
(Cntd.)



Branching instructions
JMP

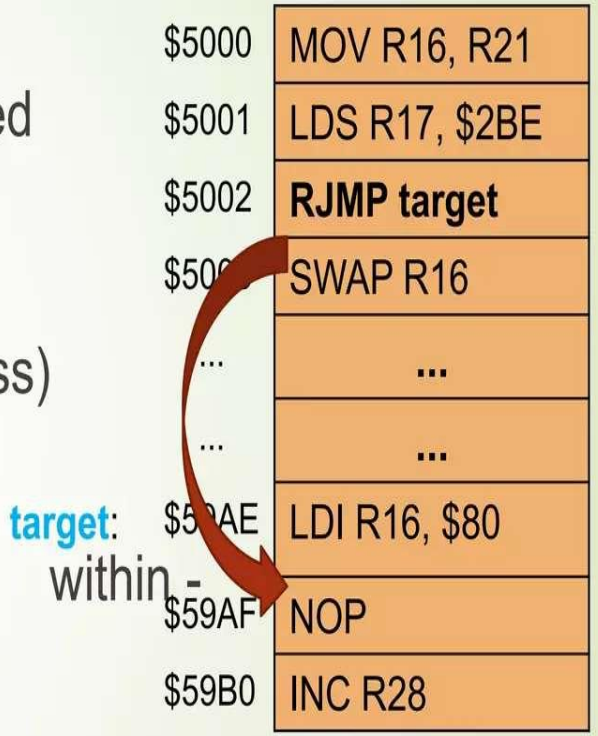
- Direct jump: The program control jumps to the address specified in the instruction
- Control is transferred unconditionally
- 4-byte instruction(32 bit)
- 10 bit for opcode, 22 bit for address(4M addressibility)
- *Syntax:* JMP k
- Example: JMP Target
- Where, 'Target' is the label of the instruction to which we intend to jump to. The assembler replaces this with the target absolute address



Branching instructions

RJMP

- Relative jump: The target address is calculated with respect to the value in the program counter
- 2-byte instruction (4-bit opcode, 12 bits address)
- *Syntax: R*JMP k
- K ranges from 000-$FFF i.e target must be within -2048 to +2047
- Example: RJMP Target
- Preferred over JMP since it takes less ROM space



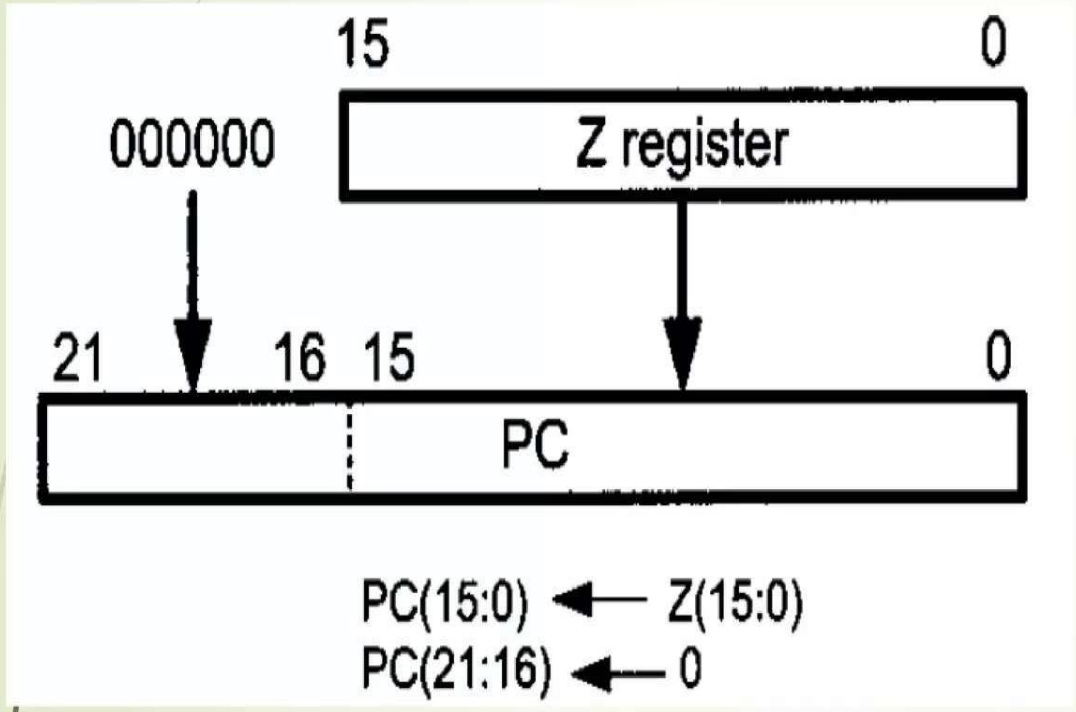
Branching instructions

IJMP

- Indirect jump: The target address is calculated with respect to the value in the program counter
- When executed PC is loaded with the content of Z register, hence jumps to the location pointed by Z register
- *Syntax:* IJMP
- In other jump instructions, the target address is static, whereas in IJMP, the target address can be made dynamic

Branching instructions

IJMP



Z	$59	$AE

	$5000	MOV R16, R21
	$5001	LDS R17, $2BE
	$50	IJMP
	$5003	SWAP R16
	...	...
	...	...
target:	$59	LDI R16, $80
	$59AF	NOP
	$59B0	INC R28

Call instructions

- CALL is used to call a subroutine
- In AVR there are four Call instructions
  - CALL - Long call
  - RCALL - Relative Call
  - ICALL - Indirect call to Z
  - EICALL - Extended indirect call to Z

Branching instructions(Cntd.)

CALL

- 4-byte instruction
- 10 bits opcode and 22 bits address of target subroutine
- CALL can be used to call subroutines placed anywhere in the 4M address space

1001	010k_{21}	$k_{20}k_{19}k_{18}k_{17}$	111k_{16}
$k_{15}k_{14}k_{13}k_{12}$	$k_{11}k_{10}k_9k_8$	$k_7k_6k_5k_4$	$k_3k_2k_1k_0$

$0 \leq k \leq 3FFFFF$

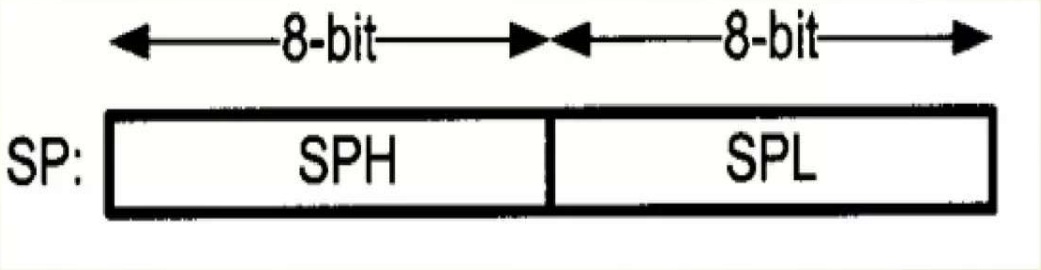
8 bit	8 bit	8 bit	8 bit
1001010k_{21}	k_{20}...k_{17}111k_{16}	k_{15} ... k_8	k_7 ... k_0

Stack

- Stack is a section of RAM used to store information temporarily. The information could be data or address.
- Stack follows LIFO
- If the stack pointer(SP) is a register inside the CPU that can point to the currently accessible location of the stack
- In I/O memory there are two registers SPL and SPH, with SPH holds the high byte of SP and SPL holds the low bytes of SP.

Stack

- SP must be wide enough to address the entire RAM. For AVRs with more than 256 bytes of RAM, SP must be wider than 8 bits.
- Inserting and retrieving data are done through PUSH and POP instructions



Stack
PUSH

- PUSH: Storing of CPU information is done through Push instruction.
- To push the content of the register Rr onto stack
- PUSH Rr
- Rr can be any GPR(R0-R31)
- Example: PUSH R10

Stack
POP

- POP: retrieving data back from stack to a register
- When pop is executed the SP is incremented by one and top location of stack is copied to register
- POP Rr
- Rr can be any GPR (R0-R31)
- Example: POP R10

Initializing Stack

- When AVR is powered up, SP contains 0, the address of R0
- We need to initialize SP to make it point to some other location in RAM
- In AVR stack grows from higher memory to lower memory
- It is convention to initialize stack to the uppermost memory location

Initializing Stack

- In AVR, **RAMEND** represents the address of the last RAM location
- To make SP point to the last RAM location, just load RAMEND to SP
- RAMEND is of two byte hence
  - Load high byte of RAMEND to SPH
  - Load low byte of RAMEND to SPL

What happens when a ‘call’ is made?

RET instruction and role of Stack

- When a subroutine is called, the processor first saves the address of the instruction just below the CALL instruction on the stack and
- Transfers control to that subroutine.
- The address of the next instruction is broken into units of 8 bits and stored in the stack
- When an **RET** instruction is executed the top location of the stack is copied back to the PC and stack pointer is incremented

Calling a subroutine-Example

Toggle all the bits of PORTB with a time delay

```
.INCLUDE "M32DEF.INC"
.ORG 0
      LDI R16,HIGH(RAMEND)    ;load SPH
      OUT SPH,R16
      LDI R16,LOW(RAMEND)     ;load SPL
      OUT SPL,R16

BACK:
      LDI  R16,0x55     ;load R16 with 0x55
      OUT  PORTB,R16    ;send 55H to port B
      CALL DELAY        ;time delay
      LDI  R16,0xAA     ;load R16 with 0xAA
      OUT  PORTB,R16    ;send 0xAA to port B
      CALL DELAY        ;time delay
```

Calling a subroutine-Example

Toggle all the bits of PORTB with a time delay

```
;------  this is the delay subroutine
       .ORG 0x300          ;put time delay at address 0x300
DELAY:
       LDI   R20,0xFF      ;R20 = 255,the counter
AGAIN:
       NOP                 ;no operation wastes clock cycles
       NOP
       DEC   R20
       BRNE AGAIN          ;repeat until R20 becomes 0
       RET                 ;return to caller
```

The upper limit of stack

- We can define stack anywhere in the general purpose memory
- Though, we must not define stack in the register memory(00-$1F) or in the I/O memory($20-5F)
- So, the SP must be set to point above $60

Calling a subroutine-Example

Arithmetic Calculator (lab experiment)

- write an assembly language program to fetch two 8-bit (unsigned) numbers from memory, perform arithmetic operation as specified by an option number stored in the third memory location and store the result back to memory