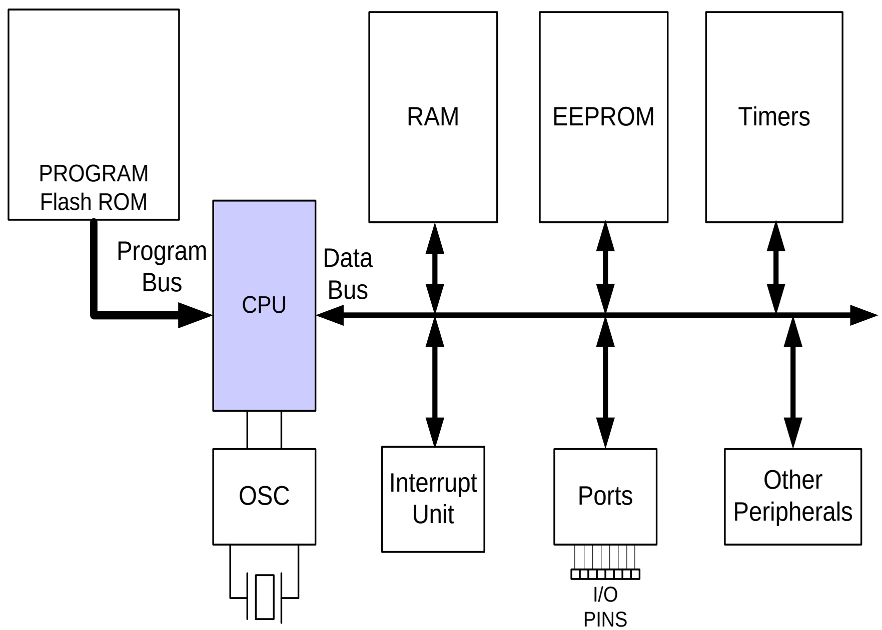
EIE3105: Hardware Connections and AVR Programming in C

Dr. Lawrence Cheung

Semester 1, 2021/22

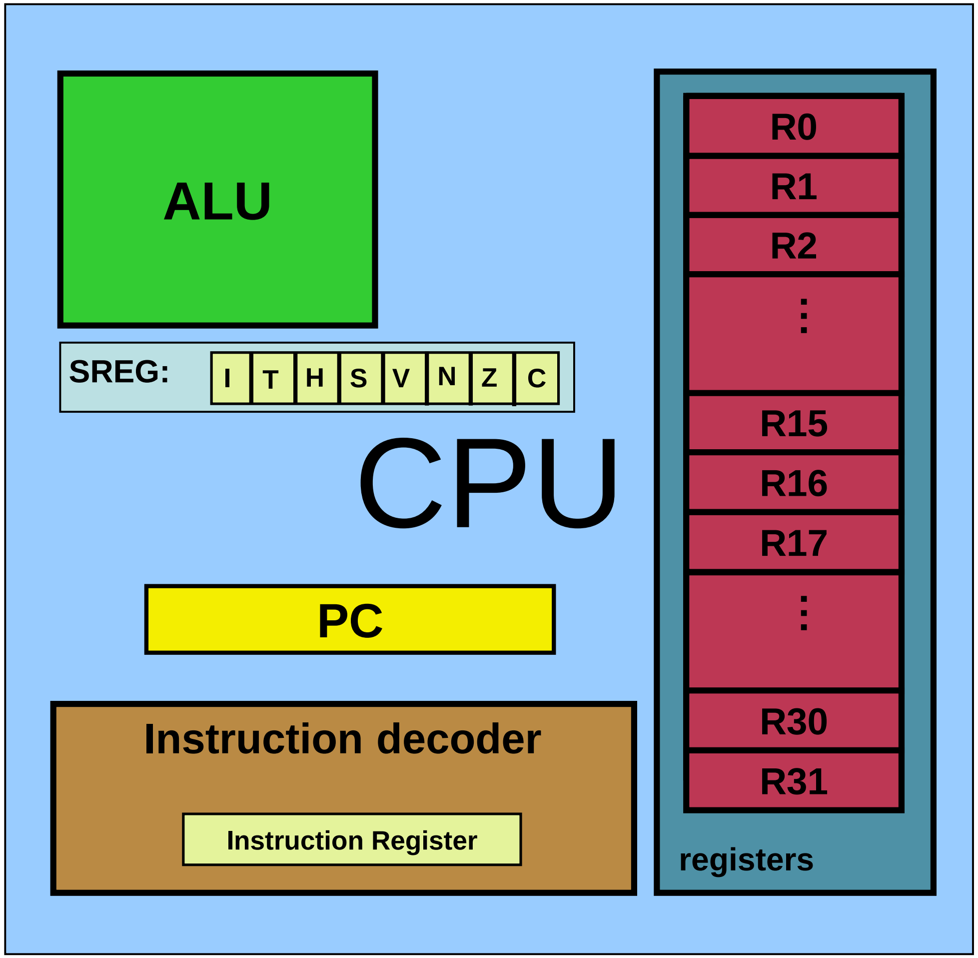
Topics

- AVR's CPU and architecture
- Data Address Space
- RISC Architecture
- I/O Programming
- C Programming
- Hardware Connections
- Hex File Format

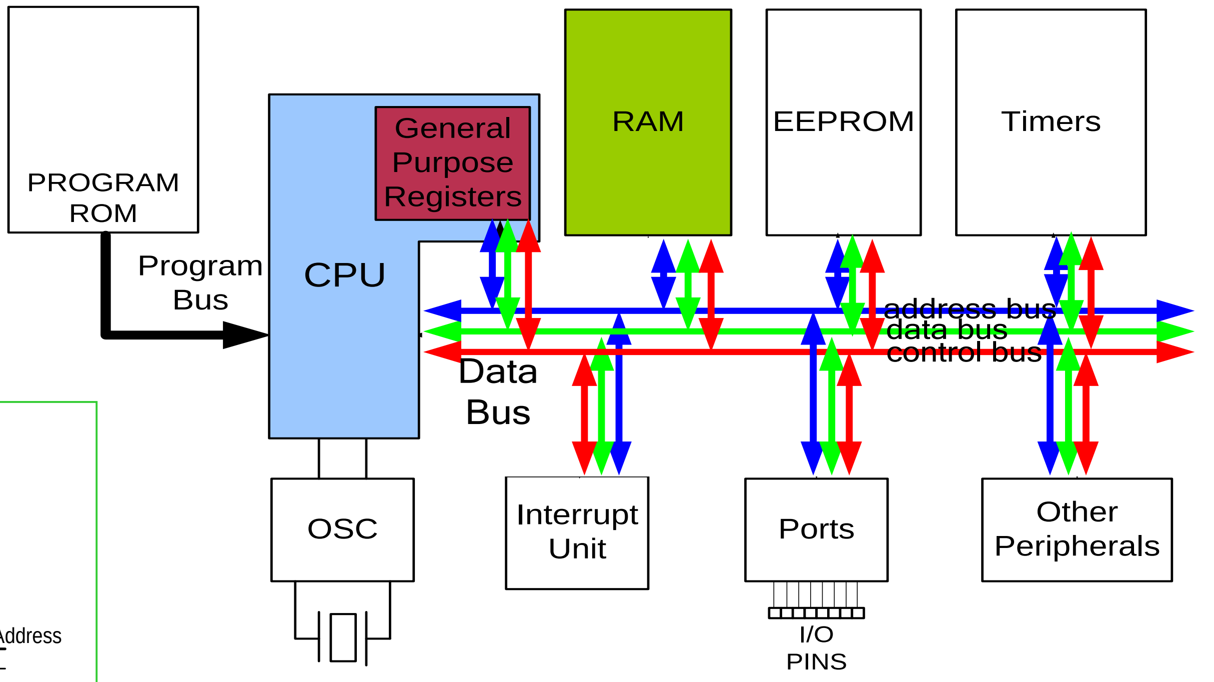


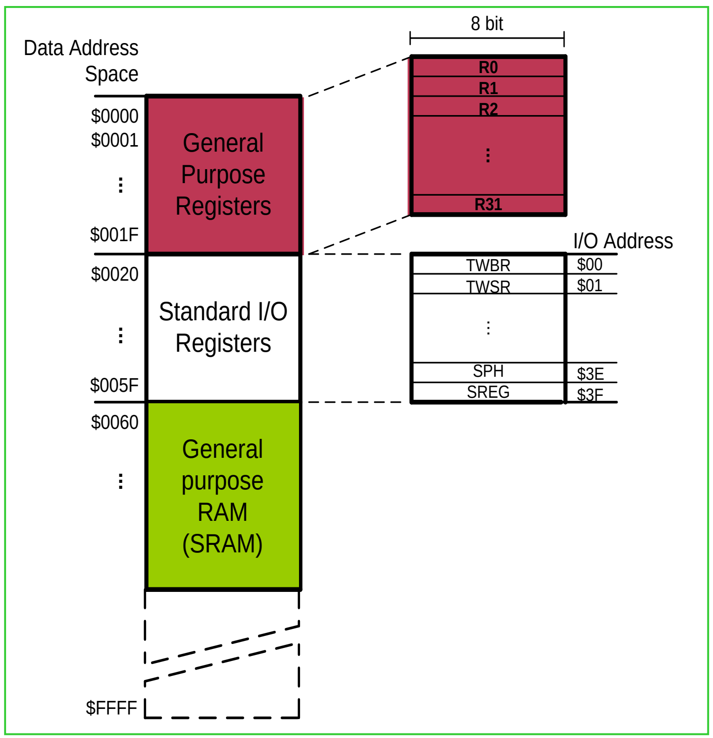
AVR's CPU

- AVR's CPU
  - ALU
  - 32 General Purpose registers (R0 to R31)
  - PC register
  - Instruction decoder

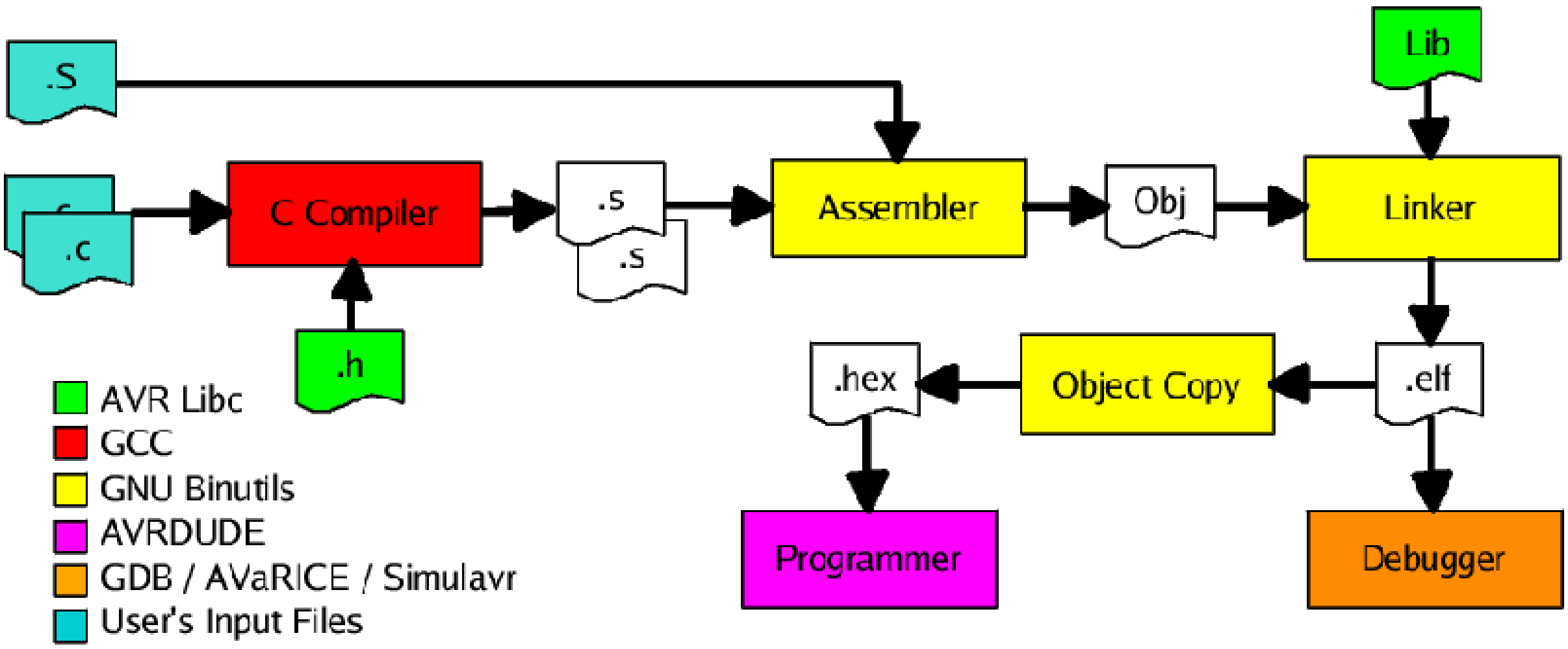


Data Address Space





C Compiler


.S
.c
.c
C Compiler
.s
.s
Assembler
Obj
Lib
Linker
.h
.hex
Object Copy
.elf
Programmer
Debugger
AVR Libc
GCC
GNU Binutils
AVRDUDE
GDB / AVaRICE / Simulavr
User's Input Files

C Compiler

- .s file: intermediate file (automatically generated source file)
- .S file: file to store your own assemble sources
- ELF: The elf file usually has text and data sections. Text section is the program code that you wrote. Data section is the initialized global/static variable.

Some simple instructions

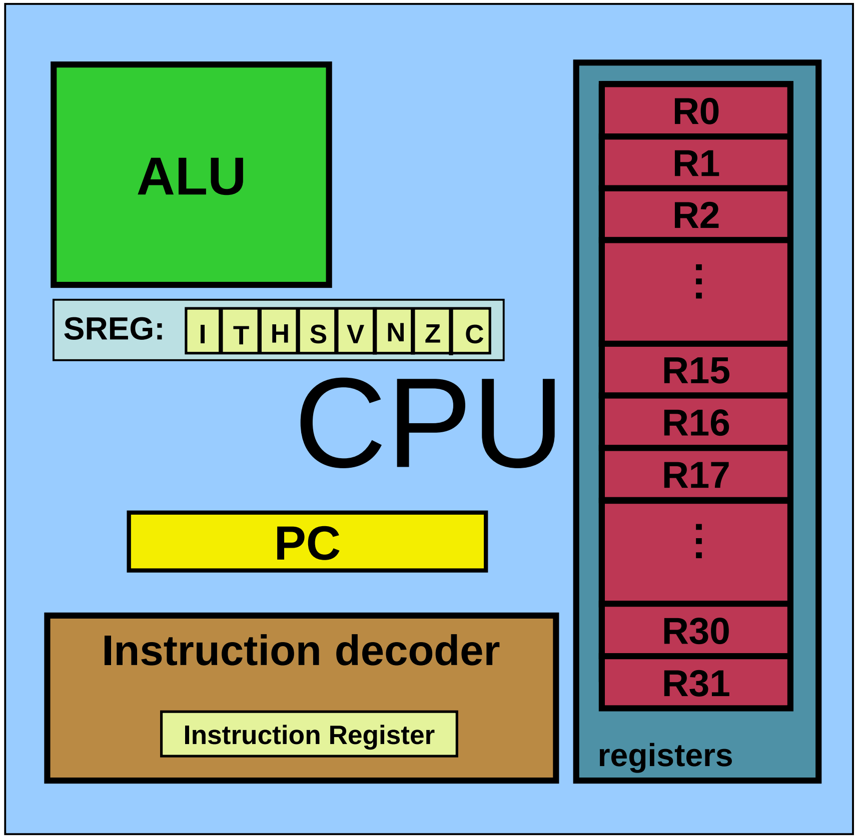
- LDI (**L**oad **I**mmediate)
  - Load values into general purpose registers (GPRs).
  - Format: LDI Rd, k
    - Rd = k
  - Examples:

LDI R16, 53 ;R16 = 53

LDI R19, 132 ;R19 = 132

LDI R23, 0x27 ;R23 = 0x27

;in hex

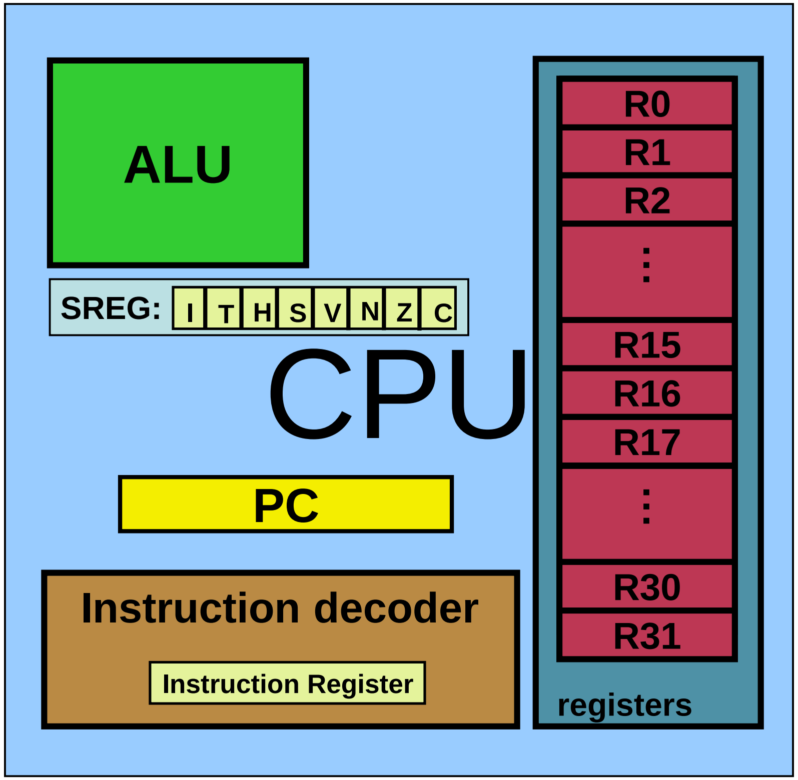


Some simple instructions

- There are some instructions for doing Arithmetic and logic operations; such as: ADD, SUB, MUL, AND, etc.
- ADD Rd, Rr
  - Rd = Rd + Rr
  - Examples:

    ADD R25, R9 ;R25 = R25 + R9

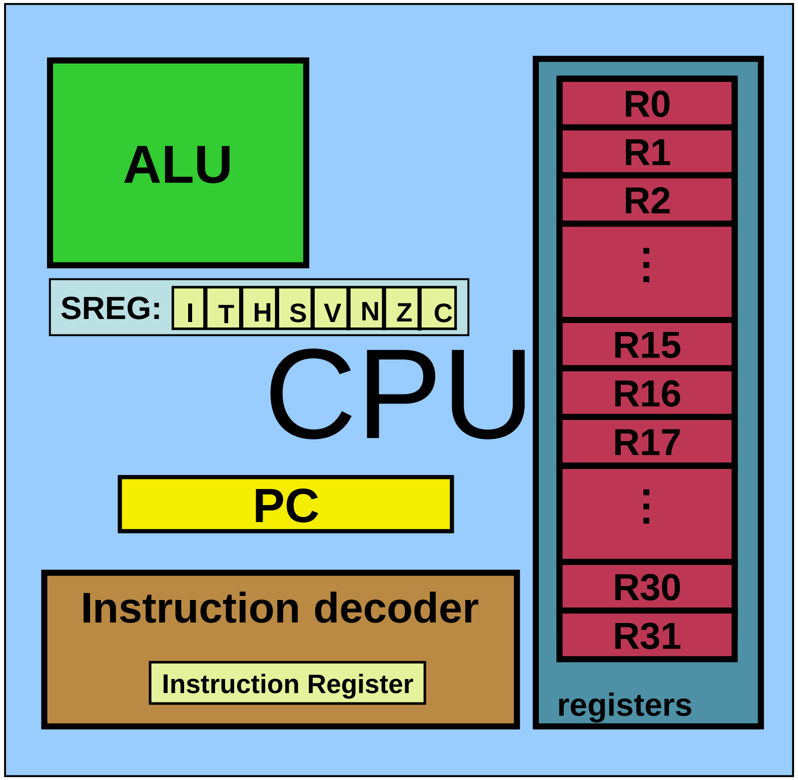
    ADD R17, R33;R17 = R17 + R30



A simple program

- Write a program that calculates 19 + 95

```
LDI R16, 19     ;R16 = 19
LDI R20, 95     ;R20 = 95
ADD R16, R20    ;R16 = R16 + R20
```



A simple program

- Write a program that calculates 19 + 95 + 5

```
LDI     R16, 19         ;R16 = 19
LDI     R20, 95         ;R20 = 95
LDI     R21, 5          ;R21 = 5
ADD     R16, R20        ;R16 = R16 + R20
ADD     R16, R21        ;R16 = R16 + R21
```

```
LDI     R16, 19         ;R16 = 19
LDI     R20, 95         ;R20 = 95
ADD     R16, R20        ;R16 = R16 + R20
LDI     R20, 5          ;R20 = 5
ADD     R16, R20        ;R16 = R16 + R20
```

Some simple instructions

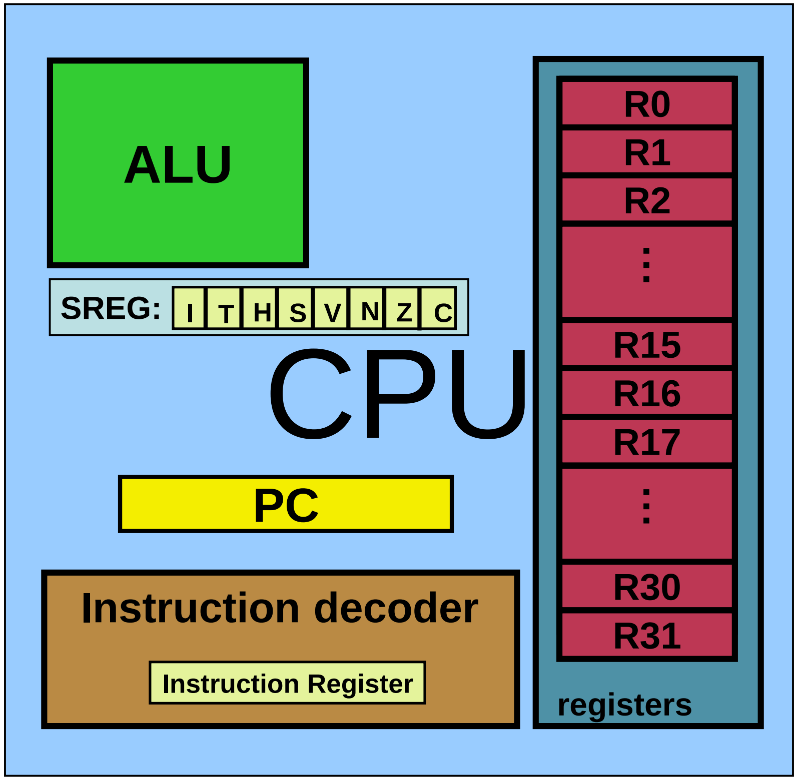
- SUB Rd, Rr
  - Rd = Rd – Rr
  - Examples:

    ;R25 = R25 – R9

    SUB R25, R9

    ;R17 = R17 – R30

    SUB R17, R30

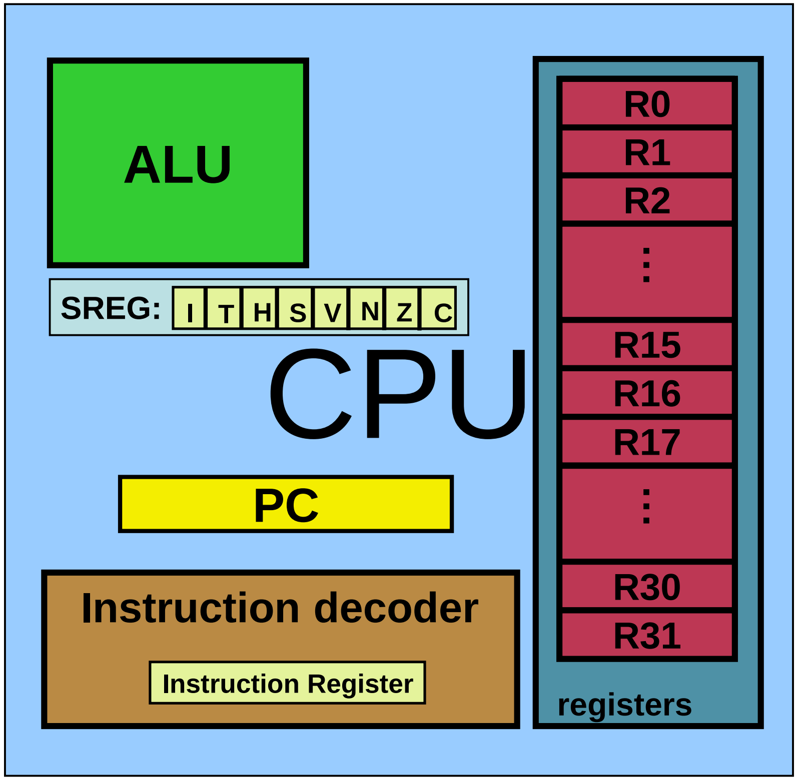


Some simple instructions

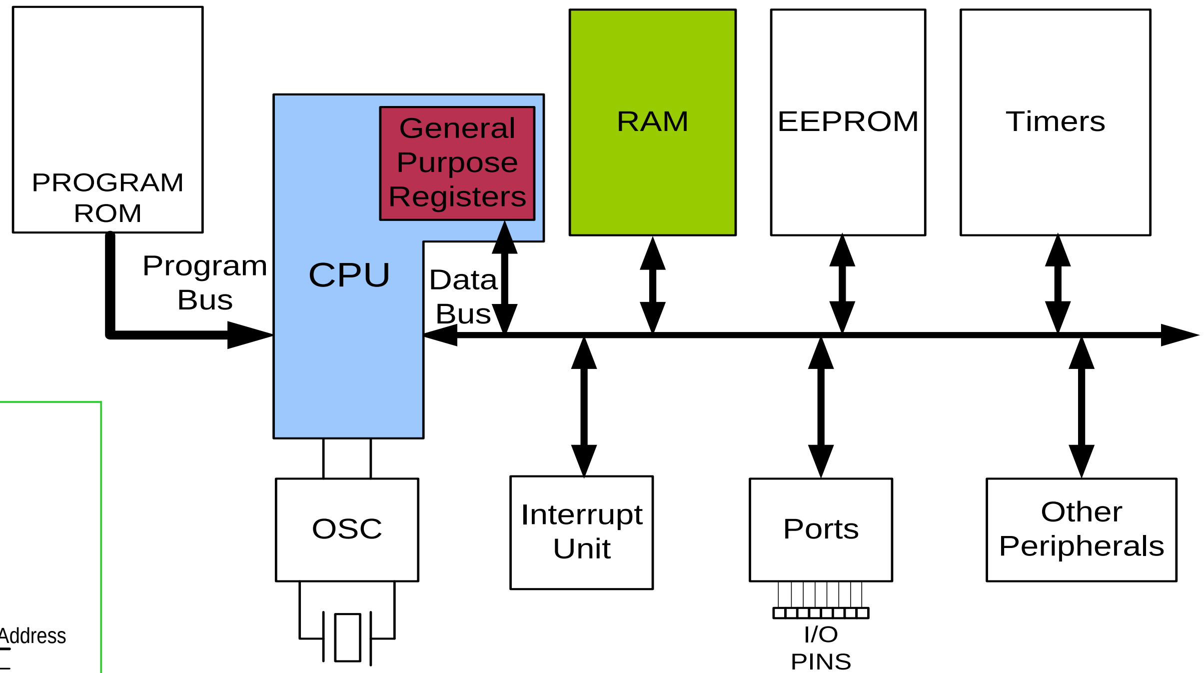
- INC Rd
  - Rd = Rd + 1
  - Example:

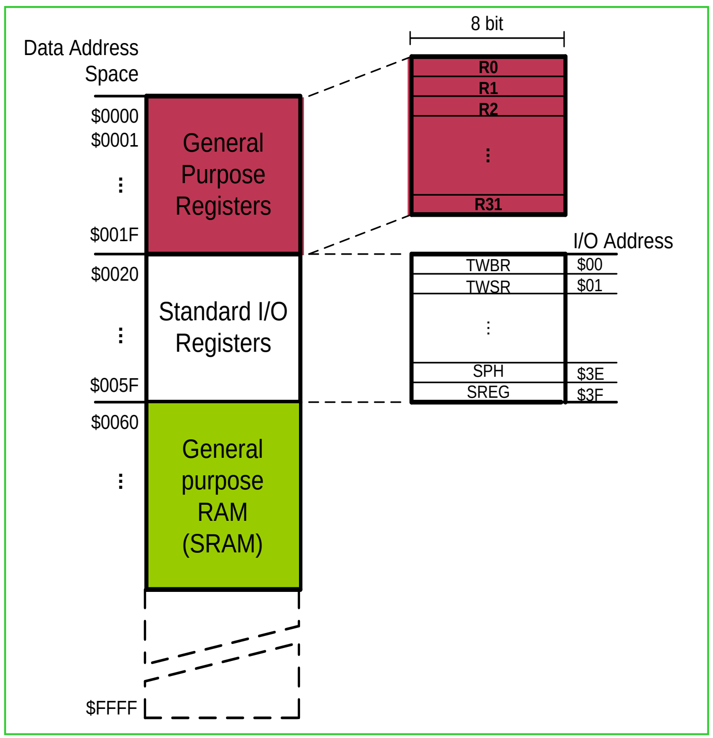
    INC R25 ;R25 = R25 + 1
- DEC Rd
  - Rd = Rd – 1
  - Example:

    DEC R23 ;R23 = R23 - 1

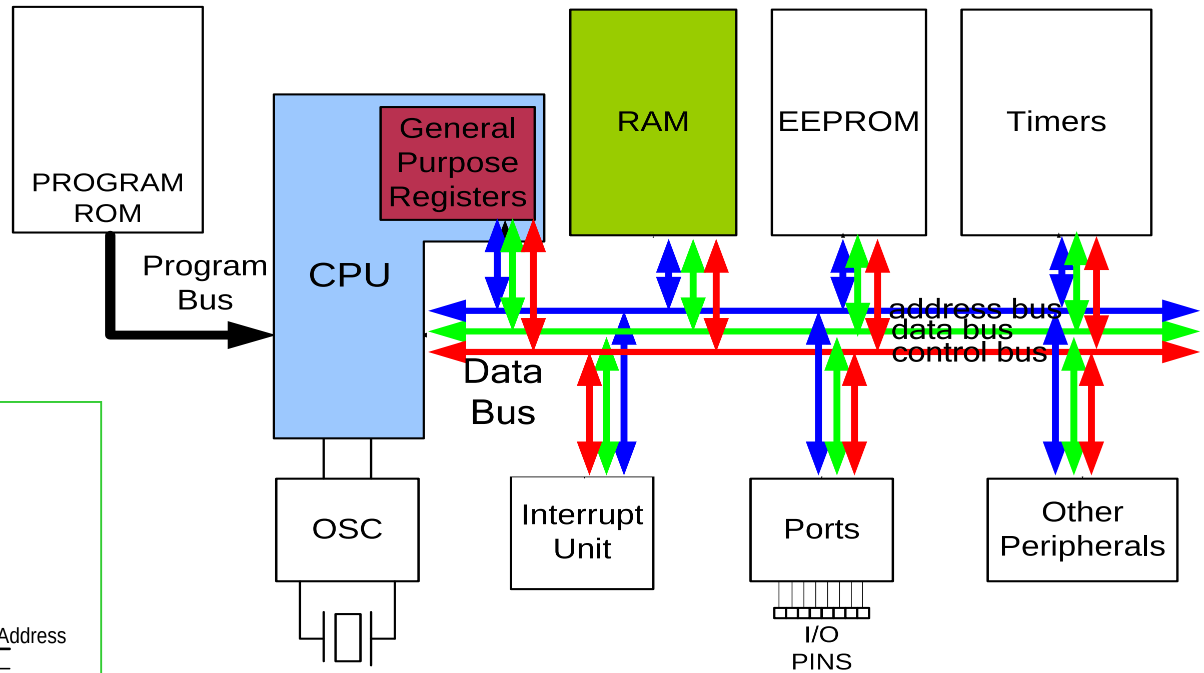


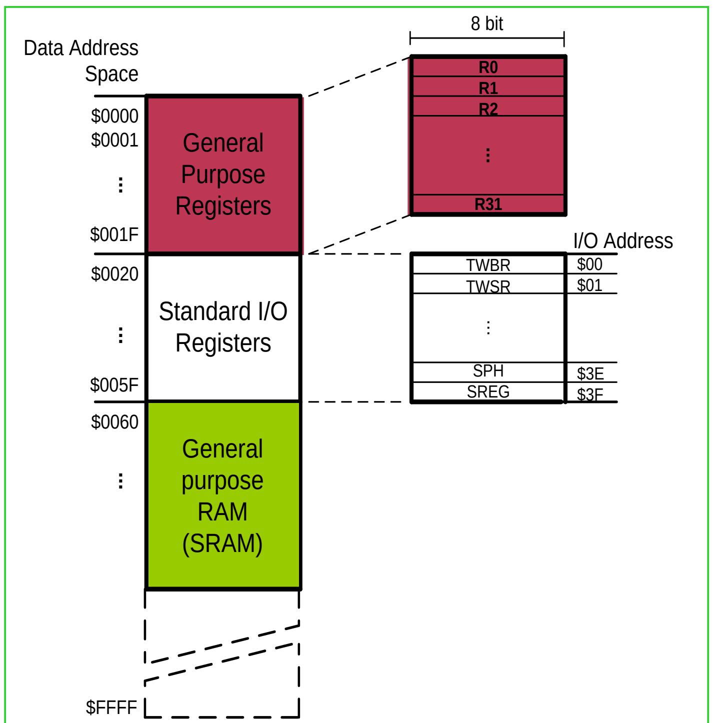
Data Address Space





Data Address Space





Data Address Space

LDS (Load direct from data space)

LDS Rd, addr ;Rd = [addr]

Example:

LDS R1, 0x60

STS (Store direct to data space)

STS addr,Rd ;[addr]=Rd

Example:

STS 0x60, R1 ;[0x60] = R15

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

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

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

Data Address Space

Example: Write a program that stores 55 into location 0x80 of RAM.

Solution:

```
LDI  R20, 55     ;R20 = 55
STS  0x80, R20   ;[0x80] = R20 = 55
```

Example: Write a program that copies the contents of location 0x80 of RAM into location 0x81.

Solution:

```
LDS  R20, 0x80        ;R20 = [0x80]
STS  0x81, R20        ;[0x81] = R20 = [0x80]
```

Example 1

- Write down the program codes of the following action: Add contents of location 0x90 to contents of location 0x95 and store the result in location 0x313.

Example 1

Data Address Space

Example: What does the following instruction do?

```
LDS  R20,2
```

Answer:

It copies the contents of R2 into R20; as 2 is the address of R2.

Example 2

- Write down the program codes of the following action: Store 0x53 into the SPH register. The address of SPH is 0x5E.

Example 2

Data Address Space

IN (IN from IO location)

IN Rd, IOaddress ;Rd = [addr]

Example:

IN R1, 0x3F ;R1 = SREG

IN R17, 0x3E ;R17 = SPH

OUT (OUT to IO location)

OUT IOAddr, Rd ;[addr]=Rd

Example:

OUT 0x3F, R12 ;SREG = R12

OUT 0x3E, R15 ;SPH = R15

Using Names of IO registers

Example:

```
OUT   SPH,R12    ;OUT  0x3E,R12
IN    R15,SREG   ;IN   R15,0x3F
```

Example 3

- Write down the program codes of the following action: Add the contents of the PINC IO register to the contents of PIND and store the result in location 0x90 of the SRAM.

Example 3

Some simple instructions

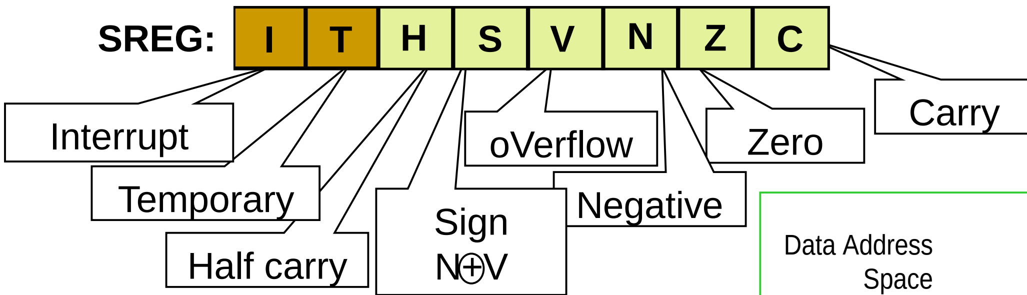
- MOV Rd, Rr
  - Rd = Rr (copy Rr to Rd)
  - Rd and Rr can be any of the GPRs
  - Example:

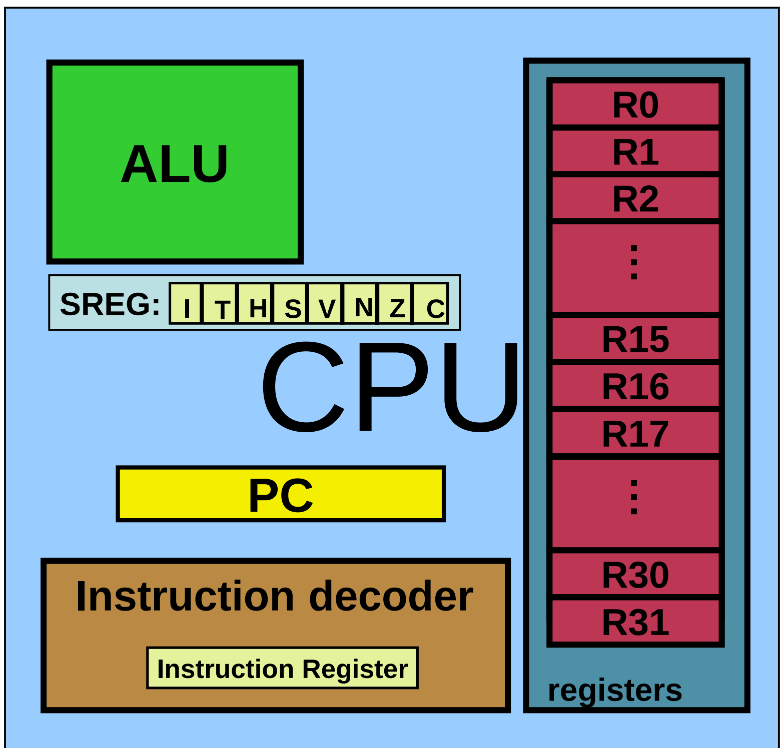
```
MOV    R10, R20                ;R10 = R20
```

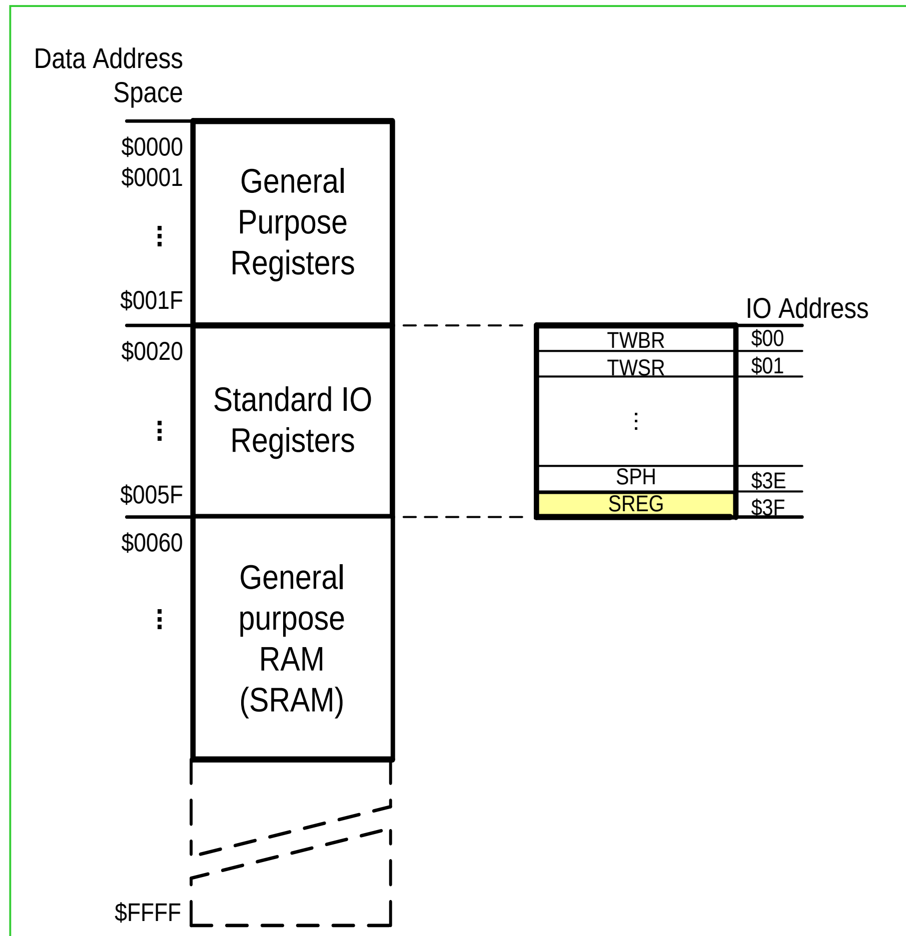
- COM Rd
  - Complement the content of Rd
  - Example:

```
LDI    R16, 0x55        ;R16 = 0x55
COM    R16              ;complement R16 (R16 = 0xAA)
```

Status Register (SREG)







Status Register (SREG)

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

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

Solution:

		1	
	$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.

Status Register (SREG)

Example: 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:

		1	
	\$9C	1001 1100	
+	\$64	0110 0100	
	\$100	1 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 a value 0 in it after the addition.

Status Register (SREG)

Example: Show the status of the C, H, and Z flags after the subtraction of 0x23 from 0xA5 in the following instructions:

```
LDI     R20, 0xA5
LDI     R21, 0x23
SUB     R20, R21        ;subtract R21 from R20
```

Solution:

	$A5	***1010 0101***	
-	***$23***	***0010 0011***	
	$82	***1000 0010***	***R20 = $82***

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 0 because the R20 has a value other than 0 after the subtraction.
H = 0 because there is no borrow from D4 to D3.

Example 4

- Example: Show the status of the C, H, and Z flags after the subtraction of 0x73 from 0x52 in the following instructions:
  - LDI R20, 0x52
  - LDI R21, 0x73
  - SUB R20, R21 ;subtract R21 from R20

Example 4

Status Register (SREG)

Example: Show the status of the C, H, and Z flags after the subtraction of 0x9C from 0x9C in the following instructions:

```
LDI    R20, 0x9C
LDI    R21, 0x9C
SUB    R20, R21        ;subtract R21 from R20
```

Solution:

	$9C	**1001 1100**	
-	**$9C**	**1001 1100**	
	$00	**0000 0000**	**R20 = $00**

C = 0 because R21 is not bigger than R20 and there is no borrow from D8 bit.
Z = 1 because the R20 is zero after the subtraction.
H = 0 because there is no borrow from D4 to D3.

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

Assembler Directives

- .EQU *name = value*
  - *Example:*

```
 .EQU    COUNT = 0x25
 LDI     R21, COUNT               ;R21 = 0x25
 LDI     R22, COUNT + 3           ;R22 = 0x28
```

- .SET *name = value*
  - *Example:*

```
.SET     COUNT = 0x25
 LDI     R21, COUNT               ;R21 = 0x25
 LDI     R22, COUNT + 3           ;R22 = 0x28
.SET     COUNT = 0x19
 LDI     R21, COUNT               ;R21 = 0x19
```

Assembler Directives

- .INCLUDE "*filename.ext*"

Table 2-6: Some of the common AVRs and their include files

MEGA		TINY		Special Purpose	
Mega8	m8def.inc	Tiny11	tn11def.inc	90CAN32	can32def.inc
Mega16	m16def.inc	Tiny12	tn12def.inc	90CAN64	can64def.inc
Mega32	m32def.inc	Tiny22	tn22def.inc	90PWM2	pwm2def.inc
Mega64	m4def.inc	Tiny44	tn44def.inc	90PWM3	pwm3def.inc
Mega128	m128def.inc	Tiny85	tn85def.inc	86RF401	at86rf401def.inc
Mega256	m256def.inc				
Mega2560	m2560def.inc				

Assembler Directives

M32def.inc

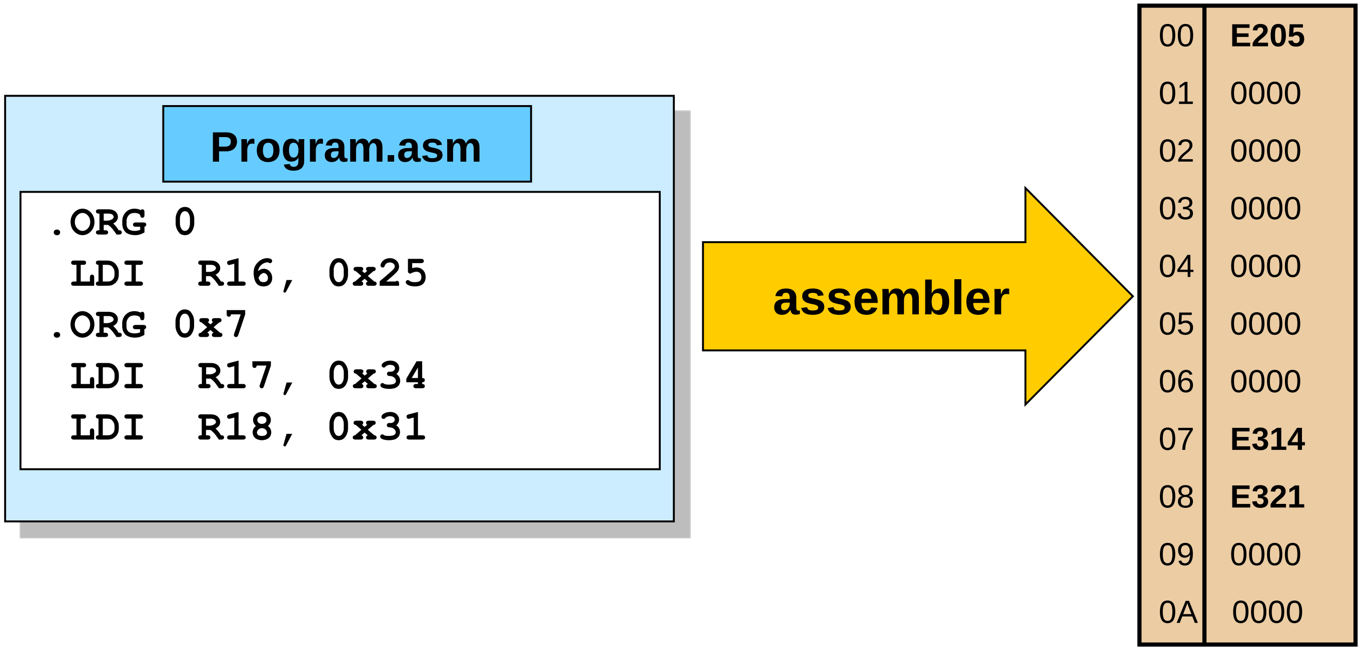
```
.equ    SREG    = 0x3f
.equ    SPL     = 0x3d
.equ    SPH     = 0x3e
....
.equ    INT_VECTORS_SIZE = 42    ; size in words
```

Program.asm

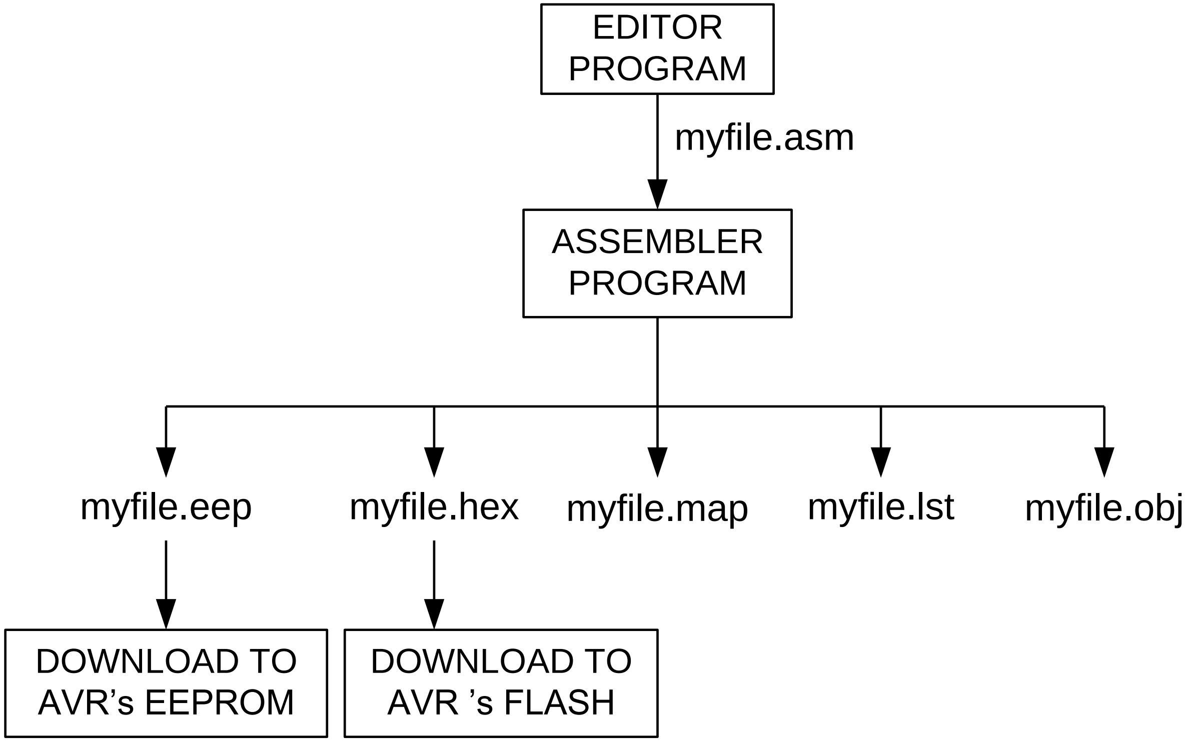
```
.INCLUDE "M32DEF.INC"
    LDI   R20, 10
    OUT   SPL, R20
```

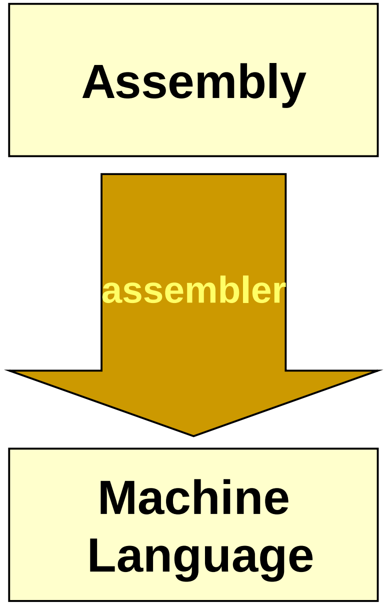
Assembler Directives

- .ORG *address*

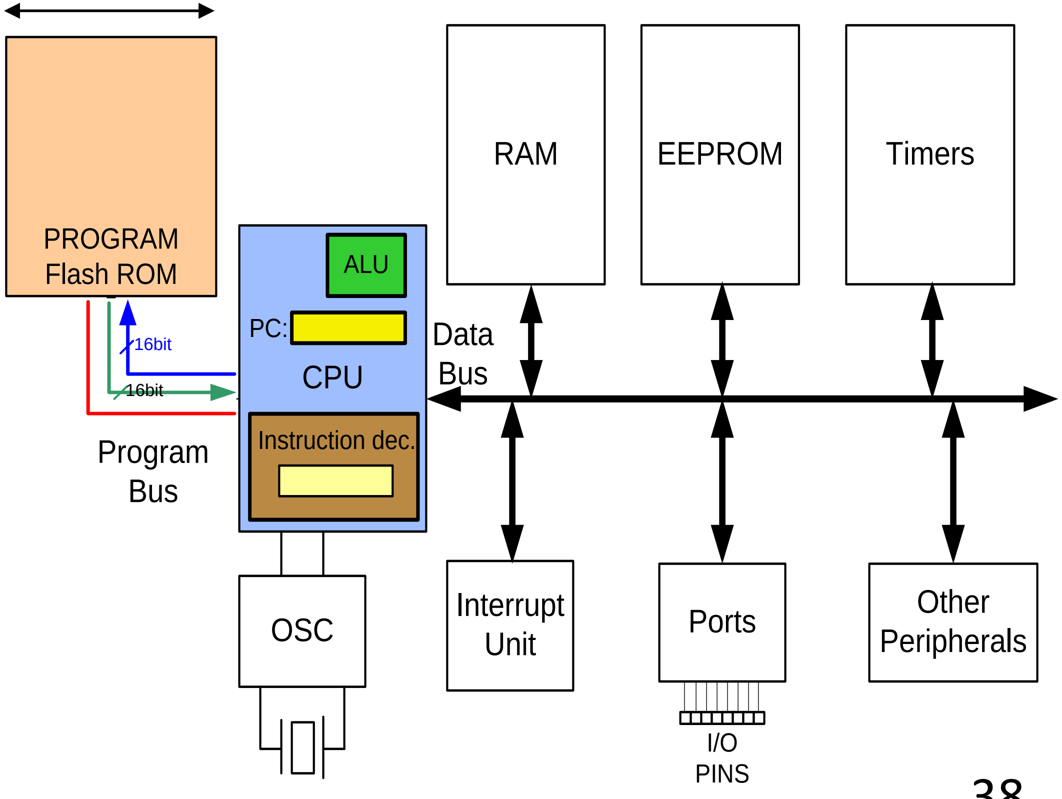


Assembler


EDITOR PROGRAM
myfile.asm
ASSEMBLER PROGRAM
myfile.eep
myfile.hex
myfile.map
myfile.lst
myfile.obj
DOWNLOAD TO AVR's EEPROM
DOWNLOAD TO AVR 's FLASH


Assembly
assembler
Machine Language

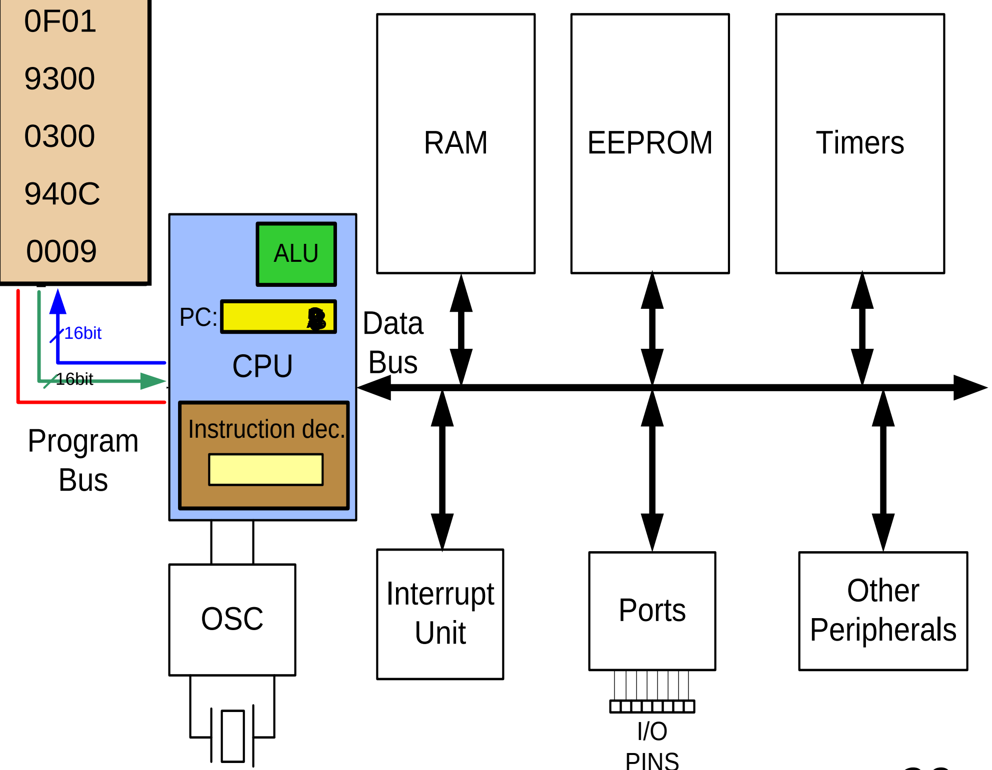
Flash memory and PC register



Flash memory and PC register

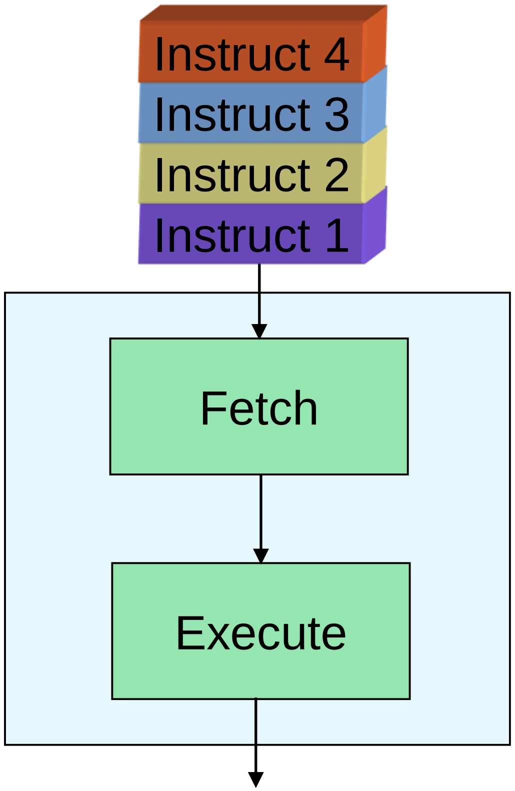
```
     LDI   R16, 0x25
     LDI   R17, $34
     LDI   R18, 0x31
     ADD   R16, R17
     ADD   R16, R18
     LDI   R17, 11
     ADD   R16, R17
     STS   SUM, R16
HERE:JMP  HERE
```

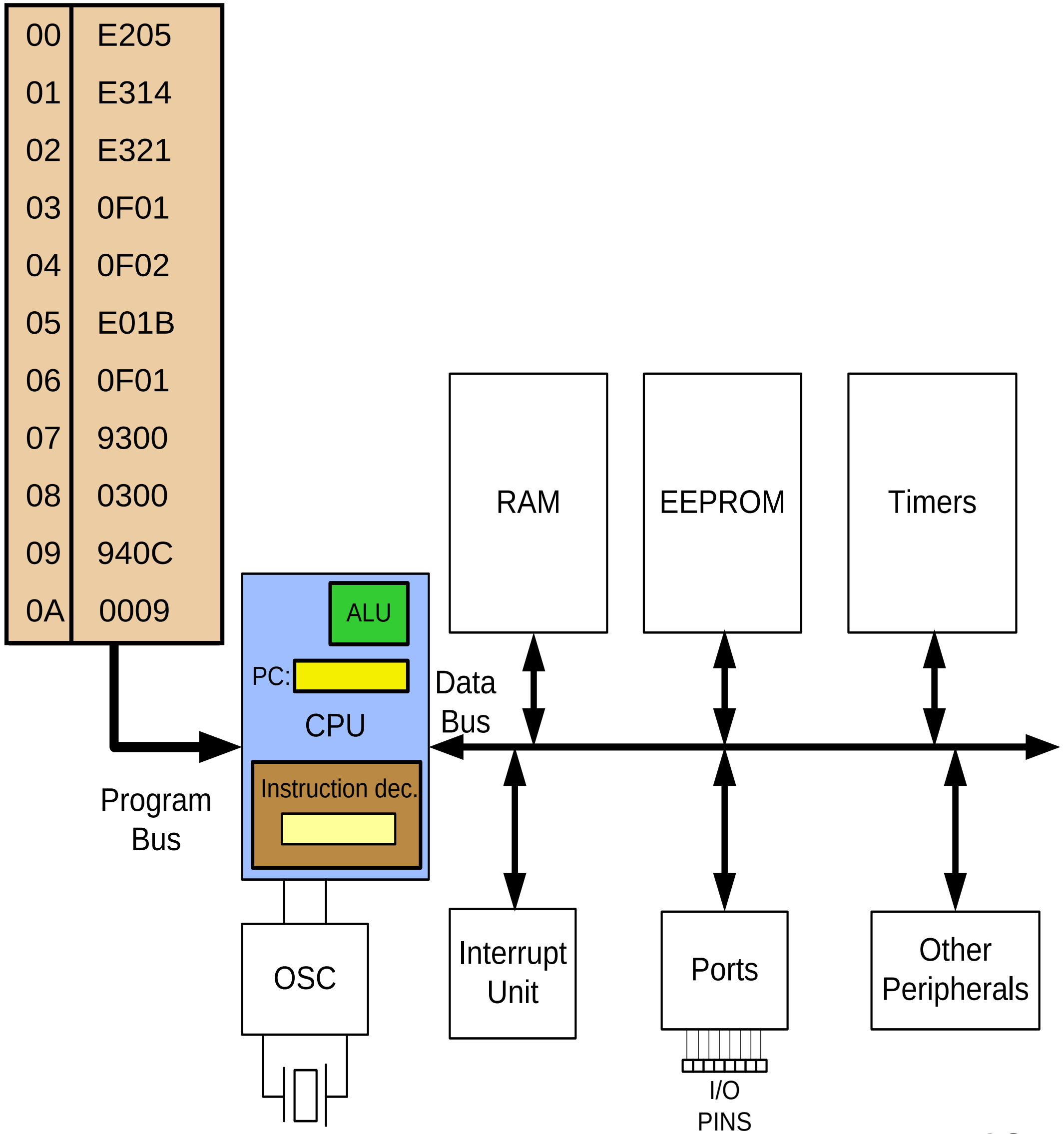
00	E205
01	E314
02	E321
03	0F01
04	0F02
05	E01B
06	0F01
07	9300
08	0300
09	940C
0A	0009



Fetch and execute

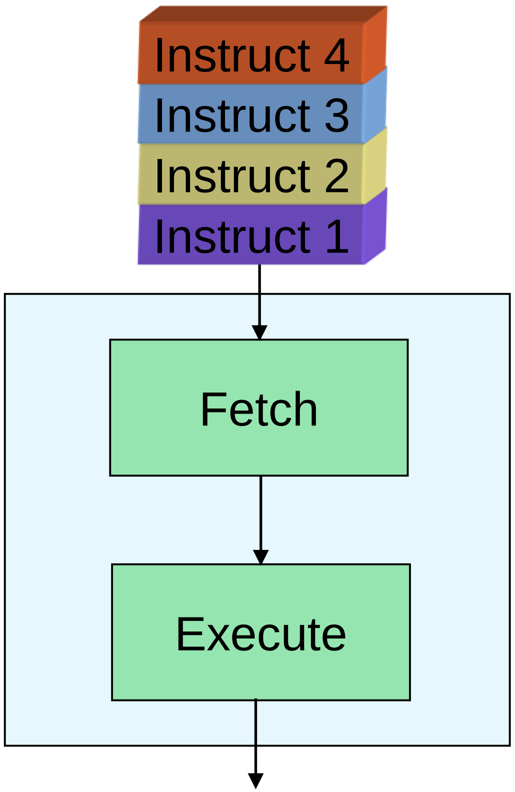
- Old Architectures

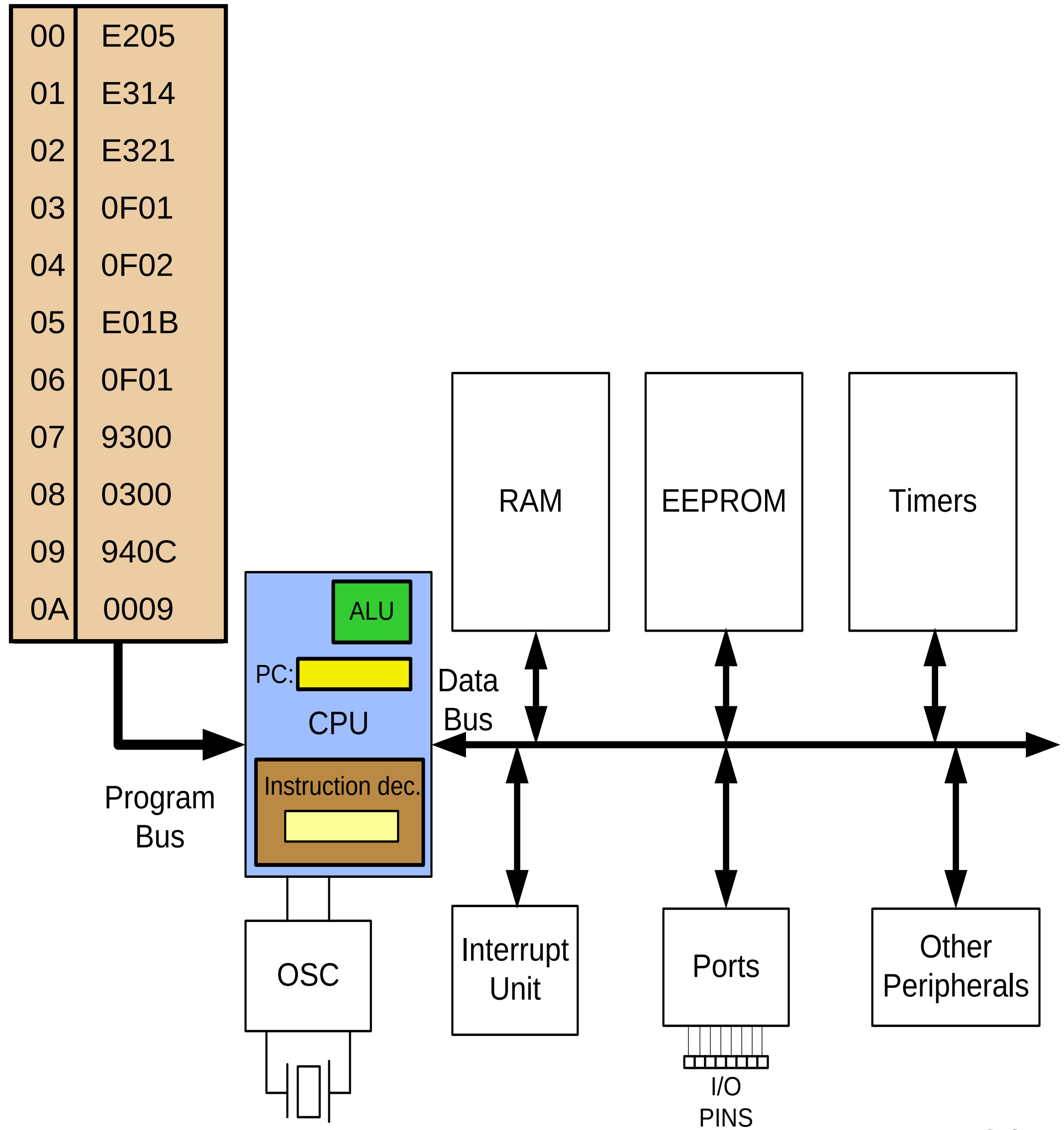




Pipelining

- Pipelining





How to speed up the CPU

- Increase the clock frequency
  - More frequency → More power consumption and more heat
  - Limitations
- Change the architecture
  - Pipelining
  - RISC

Changing the architecture: RISC vs. CISC

- CISC (Complex Instruction Set Computer)
  - Put as many instructions as you can into the CPU.
- RISC (Reduced Instruction Set Computer)
  - Reduce the number of instructions, and use your facilities in a more proper way.

RISC architecture

- Feature 1
  - RISC processors have a fixed instruction size. It makes the task of instruction decoder easier.
    - In AVR the instructions are 2 or 4 bytes.
  - In CISC processors instructions have different lengths
    - Example: in 8051
      - CLR C ; a 1-byte instruction
      - ADD A, #20H ; a 2-byte instruction
      - LJMP HERE ; a 3-byte instruction

RISC architecture

- Feature 2: reduce the number of instructions
  - Pros: Reduce the number of used transistors.
  - Cons:
    - Can make the assembly programming more difficult.
    - Can lead to using more memory.

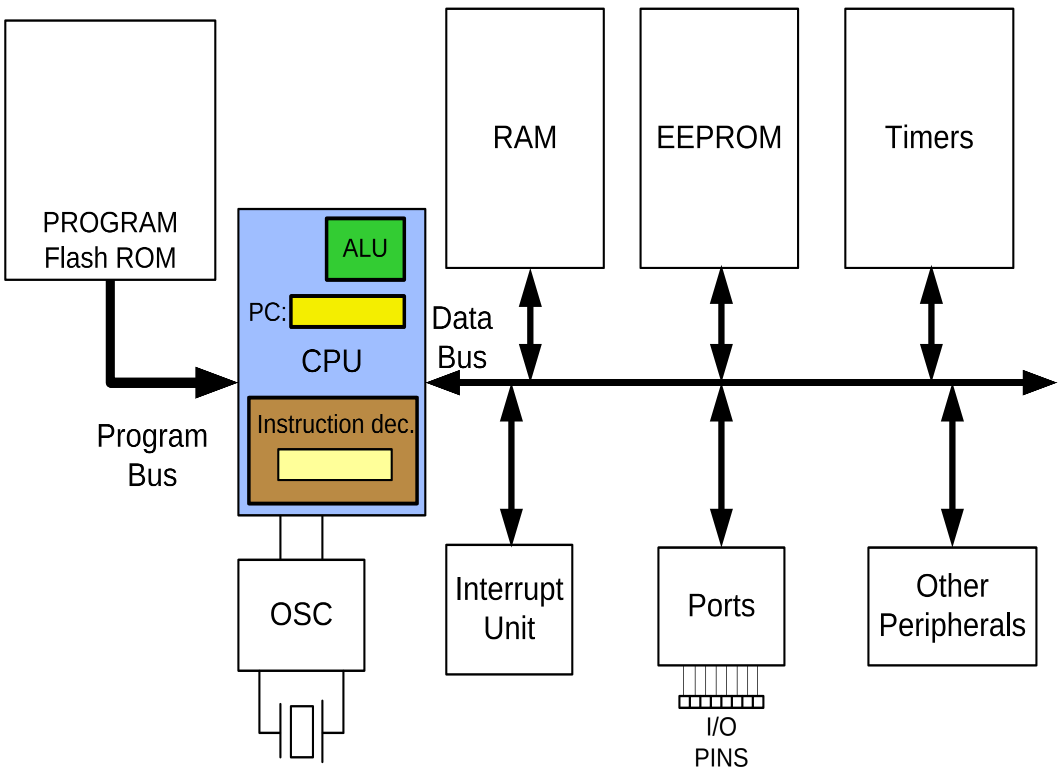
RISC architecture

- Feature 3: limit the addressing mode
  - Advantage
    - hardwiring
  - Disadvantage
    - Can make the assembly programming more difficult.

RISC architecture

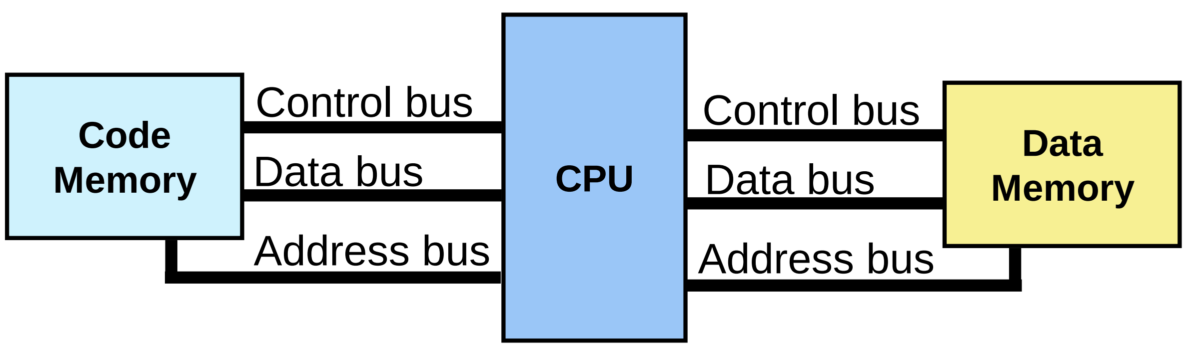
- Feature 4: Load/Store

```
LDS R20, 0x200
LDS R21, 0x220
ADD R20, R21
STS 0x230, R20
```



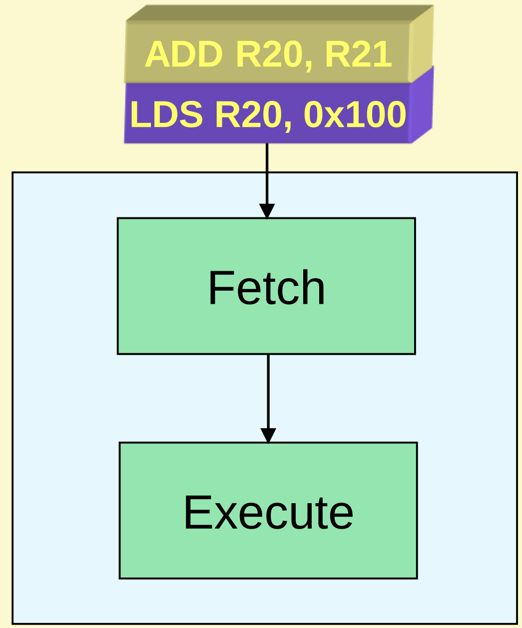
RISC architecture

- Feature 5 (Harvard architecture): separate buses for opcodes and operands
  - Advantage: opcodes and operands can go in and out of the CPU together.
  - Disadvantage: leads to more cost in general purpose computers.



RISC architecture

```
LDS R20, 0x100 ;R20 = [0x100]
ADD R20, R21   ;R20 = R20 + R21
```



RISC architecture

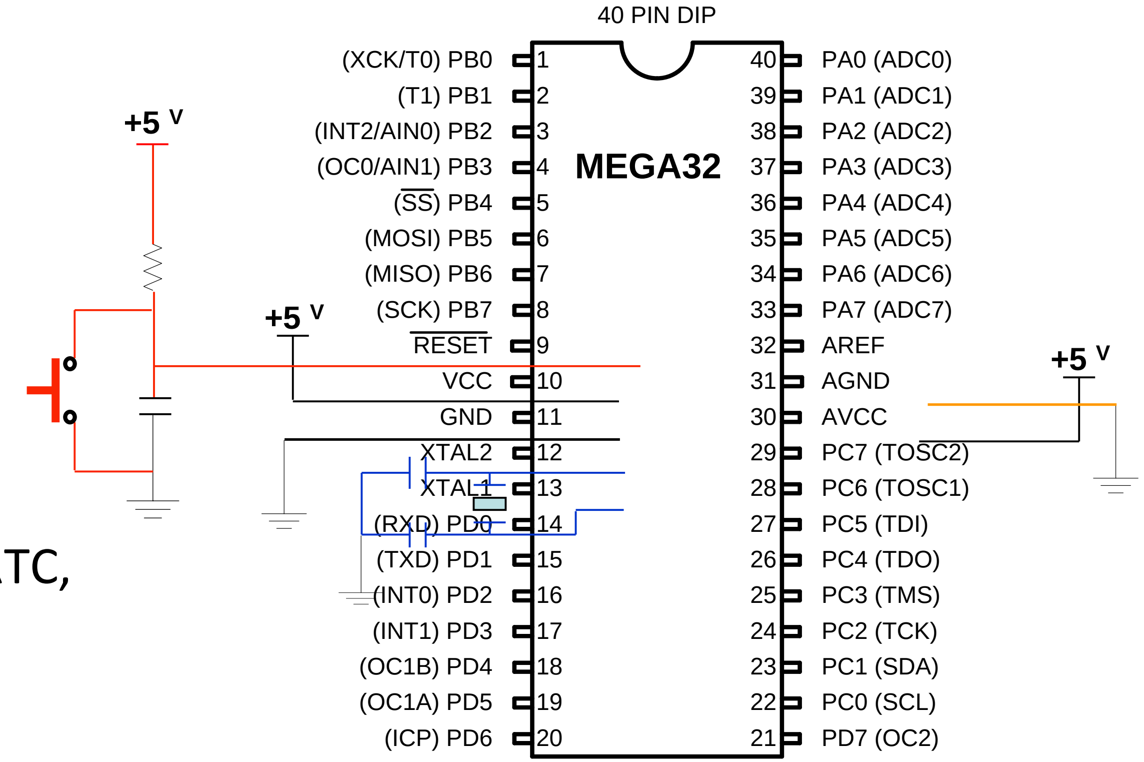
- Feature 6: more than 95% of instructions are executed in 1 machine cycle.

RISC architecture

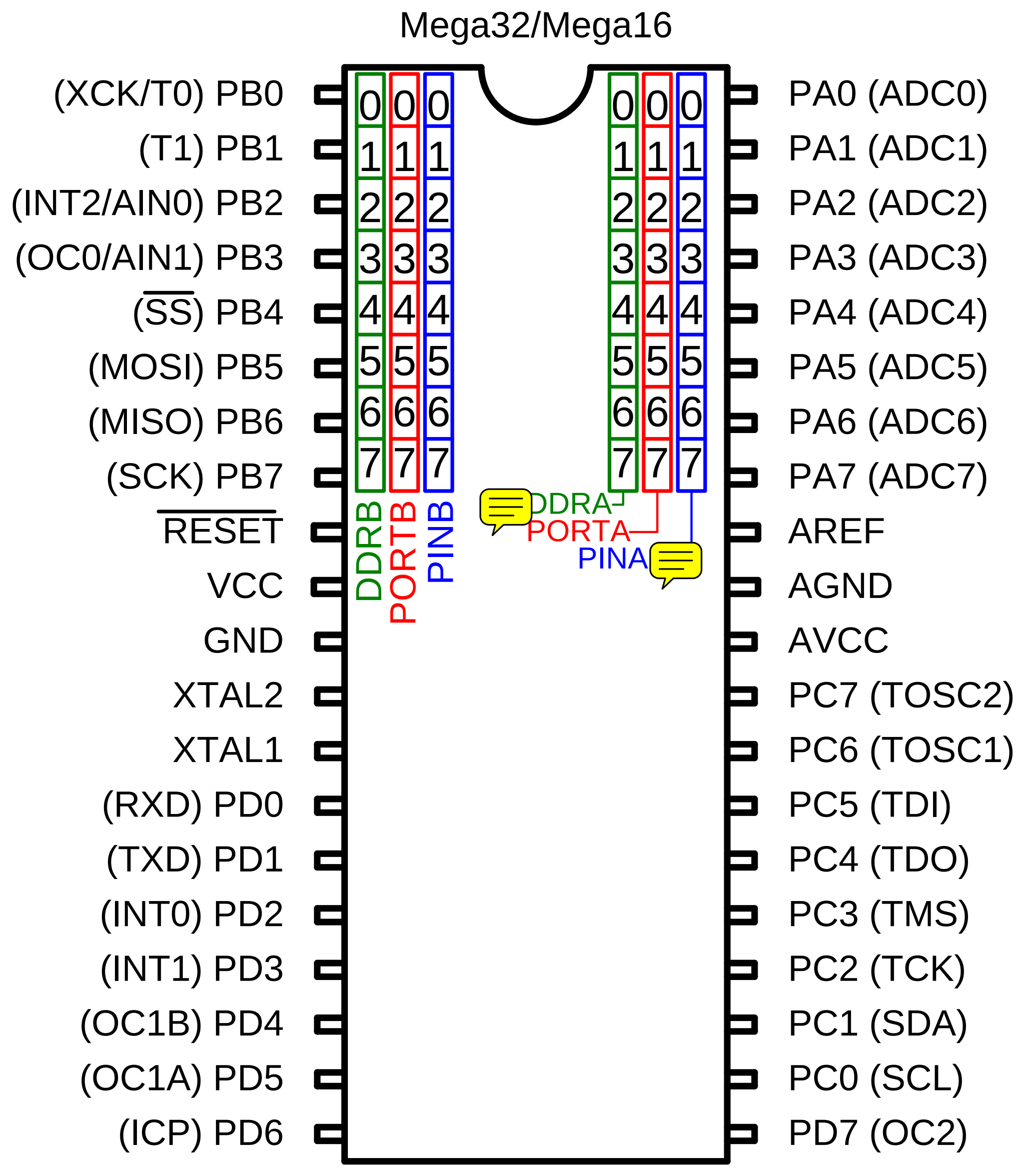
- Feature 7
  - RISC processors have at least 32 registers. It decreases the need for stack and memory usages.
    - In AVR there are 32 general purpose registers (R0 to R31).

ATmega16/mega32 pinout

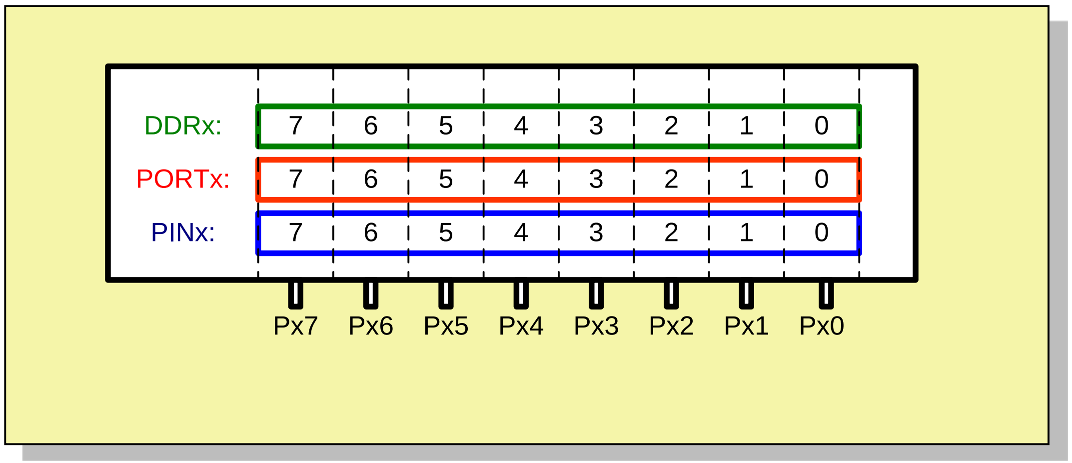
1. Vital Pins:
   1. Power
      - VCC
      - Ground
   2. Crystal
      - XTAL1
      - XTAL2
   3. Reset
2. I/O pins
   - PORTA, PORTB, PORTC, and PORTD
3. Internal ADC pins
   - AREF, AGND, AVCC

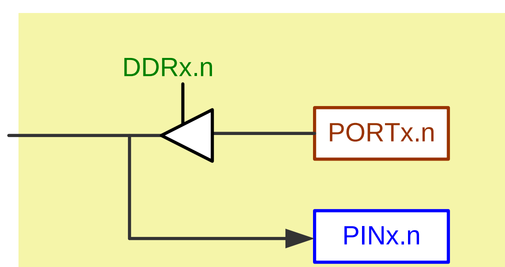


The structure of I/O pins



The structure of I/O pins





PORTx \ DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

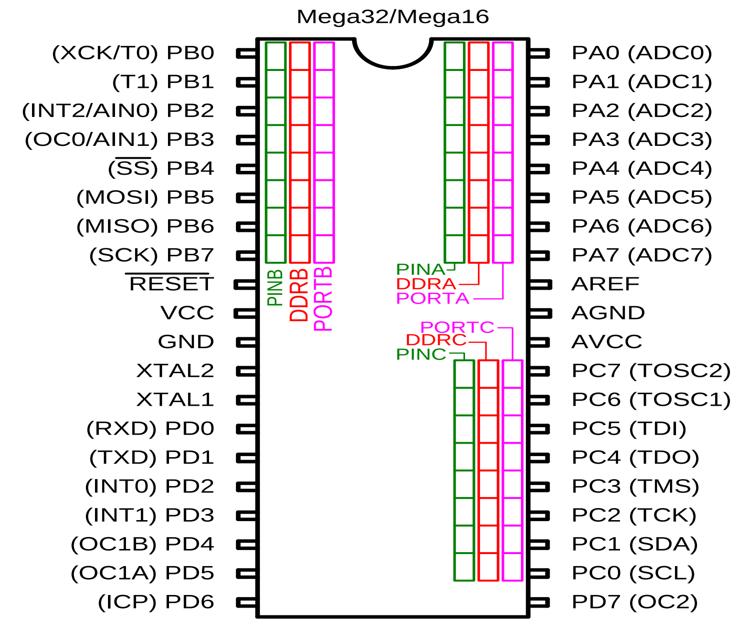
Example 5

- Write a program that makes all the pins of PORTA one.

DDRA:	1	1	1	1	1	1	1	1
PORTA:	1	1	1	1	1	1	1	1

```
DDRA = 0xFF;

PORTA = 0xFF;
```



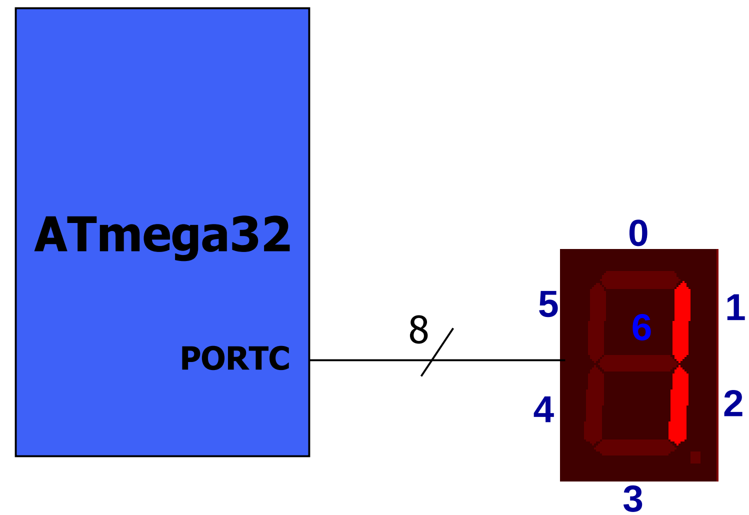
PORTx \ DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

Example 6

- A 7-segment is connected to PORTC. Display 1 on the 7-segment.

DDRC:	1	1	1	1	1	1	1	1
PORTC:	0	0	0	0	0	1	1	0

```
DDRC = 0xFF;
PORTC = 0x06;
```



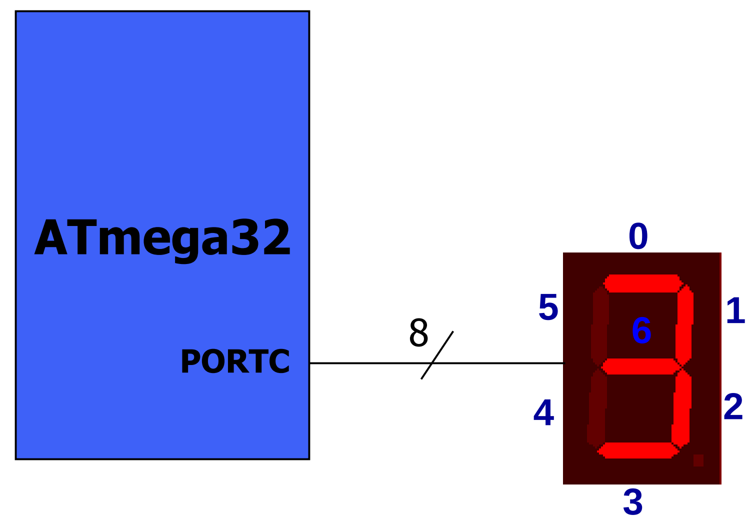
PORTx \ DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

Example 7

- A 7-segment is connected to PORTC. Display 3 on the 7-segment.

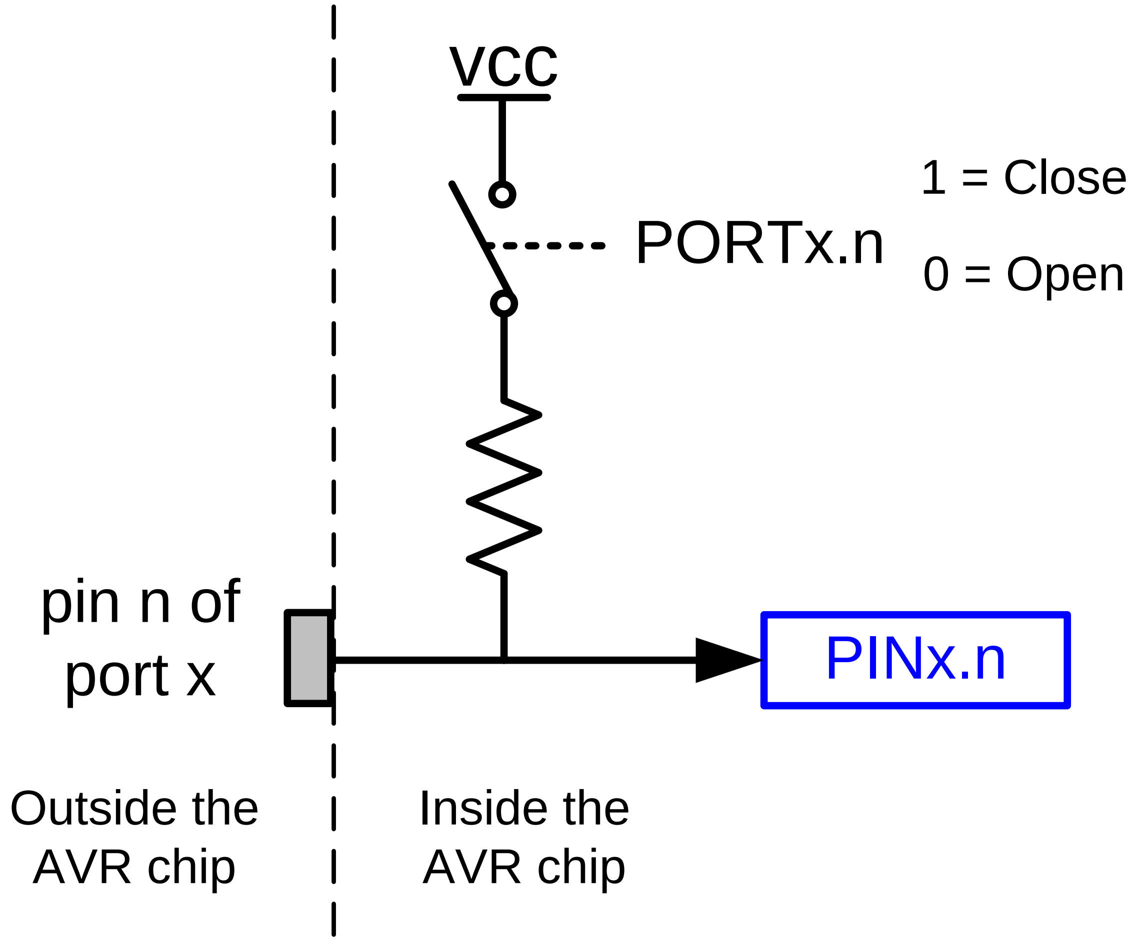
DDRC:	1	1	1	1	1	1	1	1
PORTC:	0	1	0	0	1	1	1	1

```
DDRC = 0xFF;
PORTC = 0x4F;
```

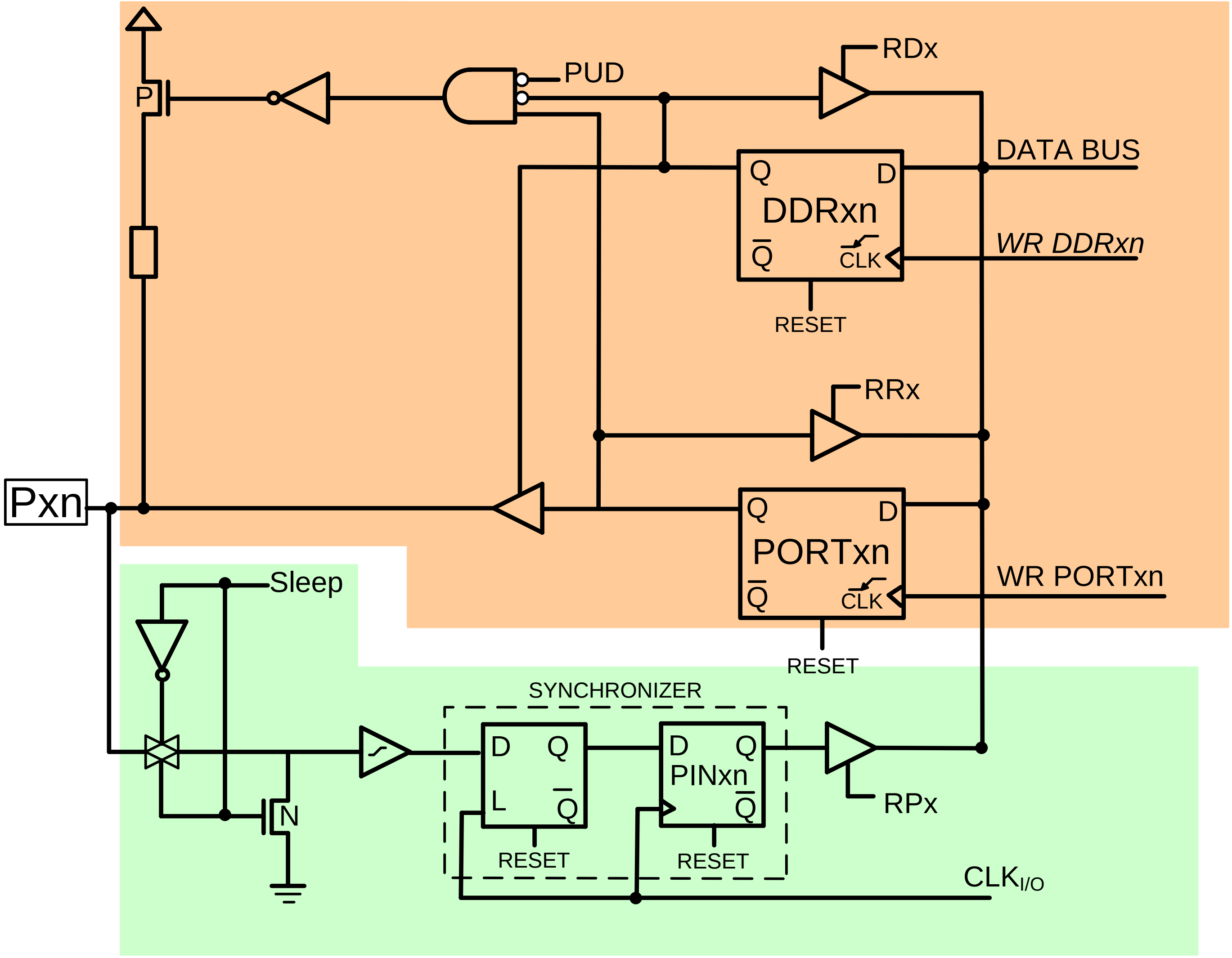


PORTx \ DDRx	0	1
0	high impedance	Out 0
1	pull-up	Out 1

Pull-up resistor



The structure of I/O pins



Assembly Program

- Write a program that continuously sends out to Port C the alternating values of 0x55 and 0xAA.

```
     .INCLUDE "M32DEF.INC"
     LDI  R16,0xFF     ;R16 = 11111111 (binary)
     OUT  DDRC,R16     ;make Port C an output port
L1:  LDI  R16,0x55     ;R16 = 0x55
     OUT  PORTC,R16    ;put 0x55 on Port C pins
     LDI  R16,0xAA     ;R16 = 0xAA
     OUT  PORTC,R16    ;put 0xAA on Port C pins
     RJMP L1
```

C Programming

- So difficult to write an assembly program!
- That is why we focus on C programming only.

Data Types

- Use unsigned whenever you can
- unsigned char instead of unsigned int if you can

Table 7-1: Some Data Types Widely Used by C compilers

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
char	8-bit	–128 to +127
unsigned int	16-bit	0 to 65,535
int	16-bit	–32,768 to +32,767
unsigned long	32-bit	0 to 4,294,967,295
long	32-bit	–2,147,483,648 to +2,147,483,648
float	32-bit	±1.175e-38 to ±3.402e38
double	32-bit	±1.175e-38 to ±3.402e38

Data Types

- Example: Write an AVR C program to send values 00 – FF to Port B.

```
#include <avr/io.h>          //standard AVR header
int main(void)
{
        unsigned char z;
        DDRB = 0xFF;        //PORTB is output
        for (z = 0; z <= 255; z++)
                PORTB = z;
        return 0;
}
// Notice the program never exists the for loop because if you
// increment an unsigned char variable when it is 0xFF, it will
// become zero.
```

Time Delays

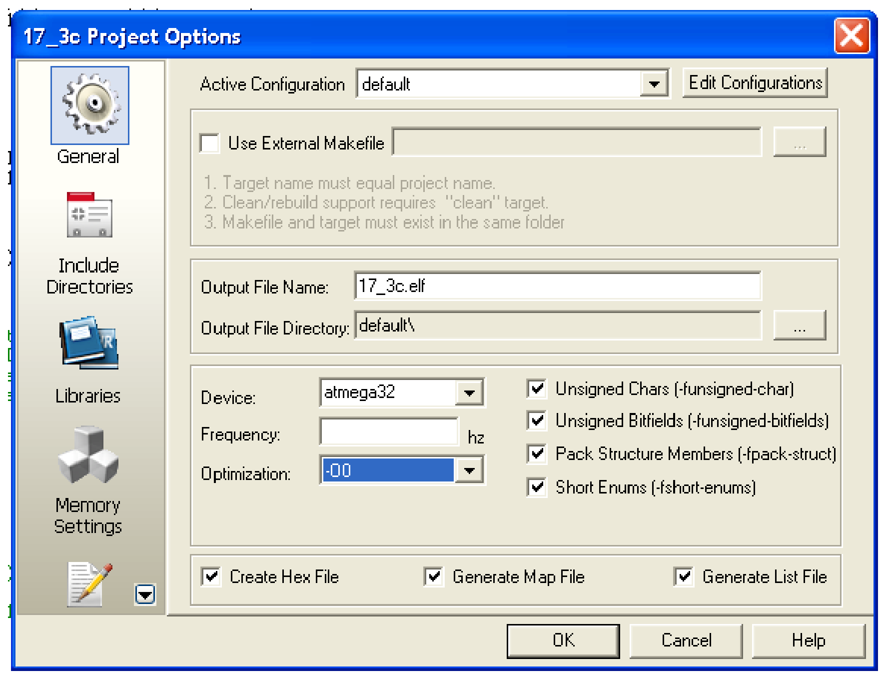
- You can use a for loop to make time delay.
- If you use for loop,
  - The clock frequency can change your delay duration!
  - The compiler has a direct effect on delay duration!
  - You MUST set the optimization level to O0!

```
void delay100ms(void){

        unsigned int i ;
        for(i=0; i<42150; i++);
}
```

Time Delays


File
Project
Build
Edit
View
Tools
Debug
Window
Help
ce Disabled
AUTO
GCC


17_3c Project Options
General
Include Directories
Libraries
Memory Settings
Active Configuration
default
Edit Configurations
Use External Makefile
1. Target name must equal project name.
2. Clean/rebuild support requires "clean" target.
3. Makefile and target must exist in the same folder
Output File Name:
17_3c.elf
Output File Directory:
default\
Device:
atmega32
Frequency:
hz
Optimization:
-O0
Unsigned Chars (-funsigned-char)
Unsigned Bitfields (-funsigned-bitfields)
Pack Structure Members (-fpack-struct)
Short Enums (-fshort-enums)
Create Hex File
Generate Map File
Generate List File
OK
Cancel
Help

Time Delays

- You can use pre-defined functions to make a time delay.
  - In WinAVR: first you should include:

```
#include <util/delay.h>
```

  - Then you can use:

```
delay_ms(1000);
delay_us(1000);
```

- It is compiler dependent, not hardware dependent.

Time Delays

- To overcome the portability problem, you can use macro or wrapper function. So to change the compiler you need to change only a simple function.

```
 void delay_ms(int d)
{
        _delay_ms(d);
}
```

Class Exercise 1

- Write an AVR C program to get a byte of data from Port C. If it is less than 100, send it to Port B; otherwise, send it to Port D. The above process will do it repeatedly.

Class Exercise 1 (Your work)

Class Exercise 1 (Answer)

I/O programming

- Byte size I/O programming in C

```
DDRB = 0xFF;
while (1) {
        PORTB = 0xFF ;
        delay100ms();
        PORTB = 0x55 ;
        delay100ms();
}
```

- Different compilers have different syntax for bit manipulations!

Masking is the best way

Logical Operations

1110 1111 && 0000 0001 = True AND True = True

1110 1111 || 0000 0000 = True OR False = True

!(1110 1111) = Not (True) = False

Bit-Wise logical operators

Table 7-3: Bit-wise Logic Operators for C

		AND	OR	EX-OR	Inverter
A	B	A&B	A|B	A^B	Y=~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

```
  1110 1111
& 0000 0001
-------------
  0000 0001
```

```
  1110 1111
| 0000 0001
-------------
  1110 1111
```

```
~ 1110 1011
-------------
  0001 0100
```

Shift operations

- data >> number of bits to be shifted right
- data << number of bits to be shifted left

1110 0000 >> 3

0001 1100

0000 0001 << 2

0000 0100

must be '0'

Setting a bit in a Byte to 1

- We can use | operator to set a bit of a byte to 1.

```
    xxxx xxxx                  xxxx xxxx
|   0001 0000      OR     |    1 << 4
    -------------              -------------
    xxx1 xxxx                  xxx1 xxxx
```

```
PORTB |= ( 1 << 4);     //Set bit 4 (5th bit) of PORTB
```

Clearing a bit in a Byte to 0

- We can use & operator to clear a bit of a byte to 0.

	xxxx xxxx			xxxx xxxx
&	1110 1111	OR	&	~(1 << 4)
	-------------			-------------
	xxx0 xxxx			xxx0 xxxx

```
PORTB &= ~( 1 << 4);    //Clear bit 4 (5th bit) of PORTB
```

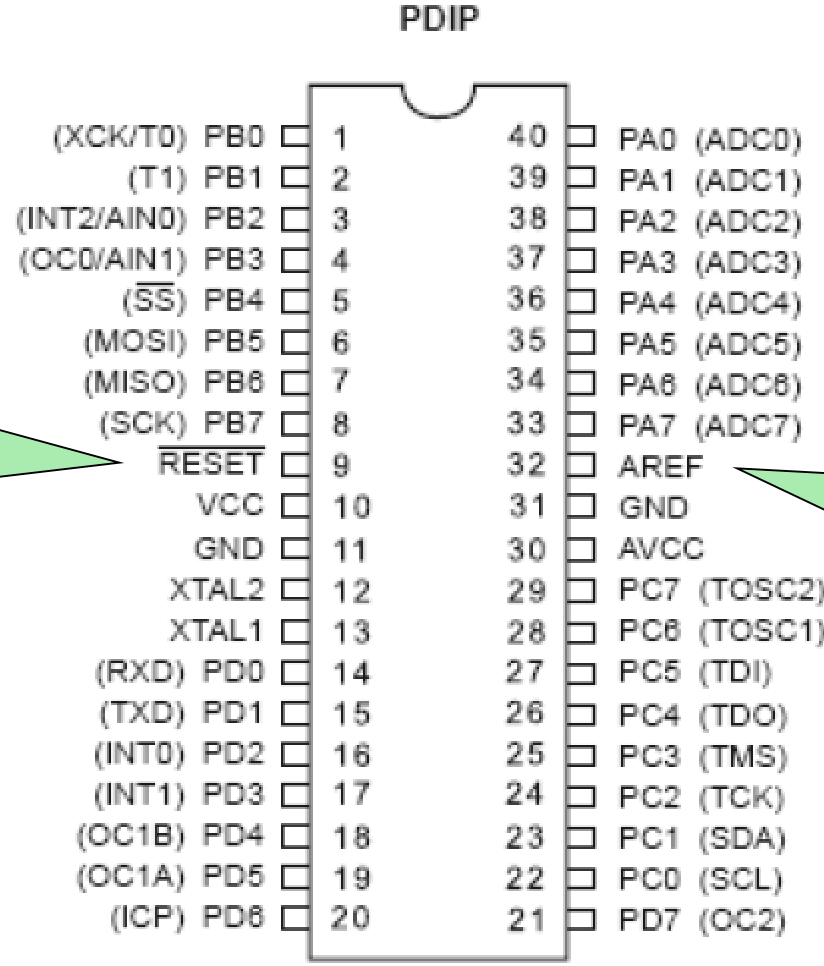
Checking a bit in a Byte

- We can use & operator to see if a bit in a byte is 1 or 0.

	xxxx xxxx			xxxx xxxx
&	0010 0000	OR	&	(1 << 5)
	-------------			-------------
	00x0 0000			00x0 0000

```
if (PINC & (1 << 5))    // check bit 5 (6th bit) of PINC
```

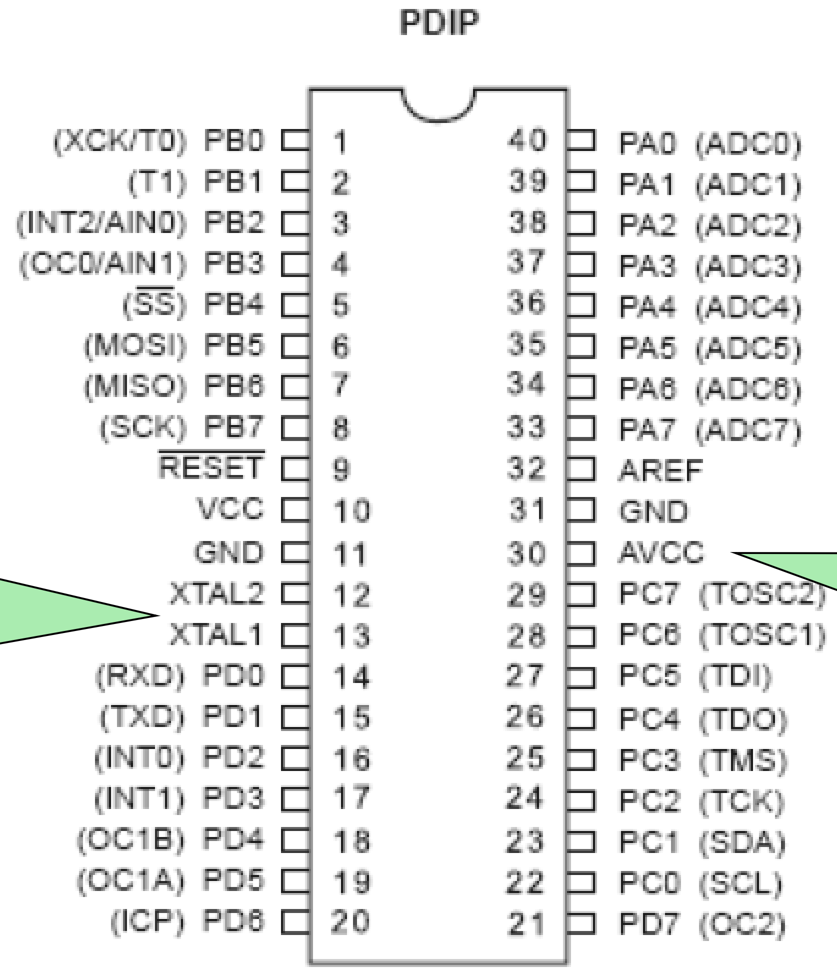
ATmega 32 pins



Clears all the registers and restart the execution of program

Reference voltage for ADC

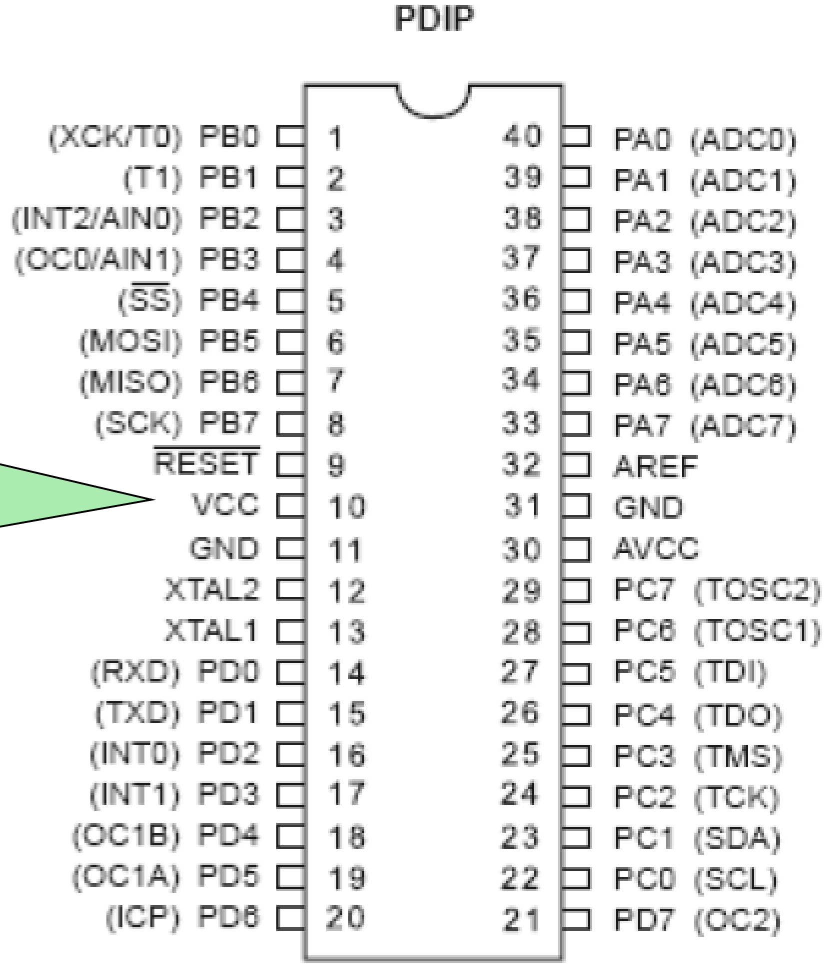
ATmega 32 pins



These pins are used to connect external crystal or RC oscillator

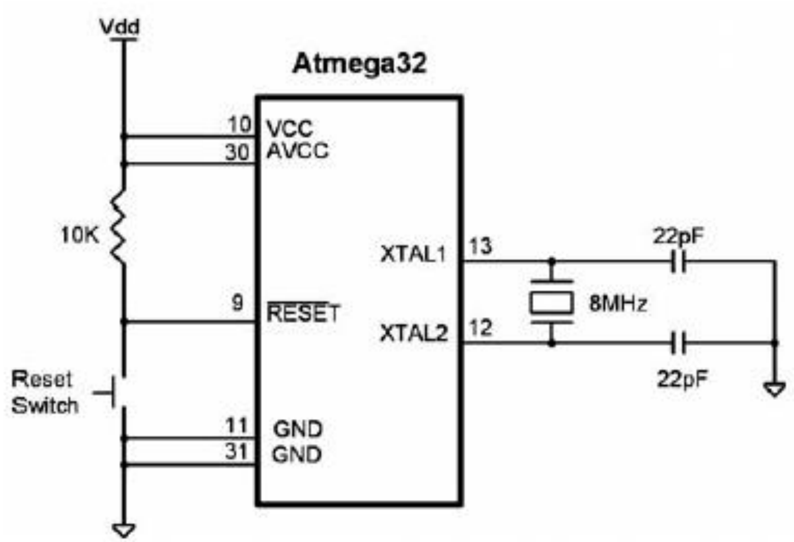
Supply voltage for ADC and portA. Connect it to VCC

ATmega 32 pins



Provides supply voltage to the chip. It should be connected to +5

AVR simplest connection



Fuse bytes of ATmega 32

Table 8-6 Fuse High Byte

Fuse High	Bit No.	Description	Defult Value Byte
OCDEN	7	Enable OCD	1 (unprogrammed)
JTAGEN	6	Enable JTAG	0 (programmed)
SPIEN	5	Enable SPI Serial Program and Data Downloading	0 (programmed)
CKOPT	4	Oscilator options	1 (unprogrammed)
EESAVE	3	EEPROM memory is preserved through the Chip Erase	1 (unprogrammed)
BOOTSZ1	2	Select boot size	0 (programmed)
BOOTSZ0	1	Select boot size	0 (programmed)
BOOTRST	0	Select reset vector	1 (unprogrammed)

Fuse bytes of ATmega 32

Table 8-7 Fuse Low Byte

Fuse High Byte	Bit No.	Description	Defult Value
BODLEVEL	7	Brown-out Detector trigger level	1
BODEN	6	Brown-out Detector enable	1
SUT1	5	Select start-up time	1
SUT0	4	Select start-up time	0
CKSEL3	3	Select Clock source	0
CKSEL2	2	Select Clock source	0
CKSEL1	1	Select Clock source	0
CKSEL0	0	Select Clock source	1

Clock source in ATmega 32

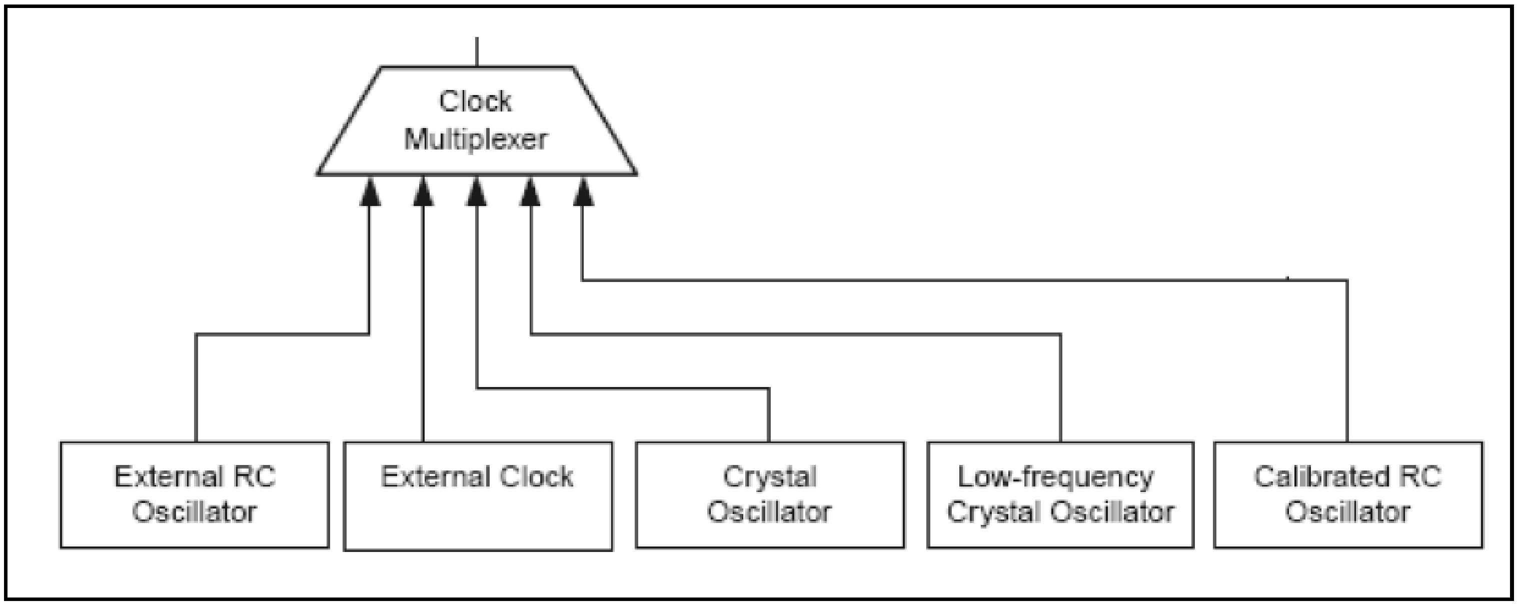


Figure 8-4. Atmega32 Clock Sources

Clock source in ATmega 32

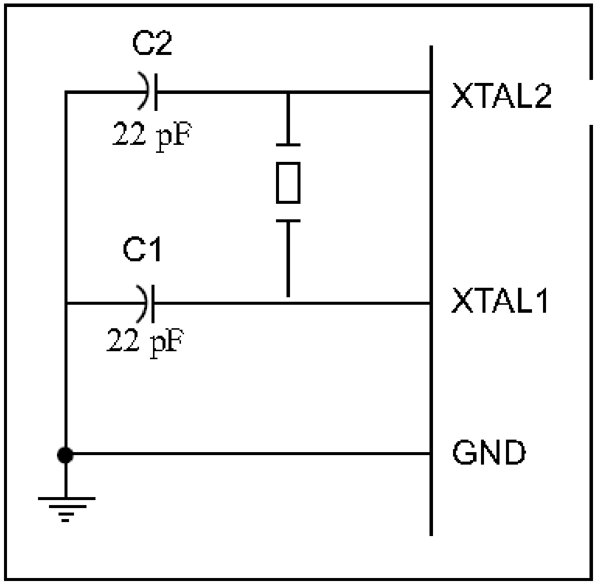


Figure 8-6a. XTAL1–XTAL2 Connection to Crystal Oscillator

CKOPT	CKSEL3..1	Frequency (MH)
1	101	0.4 - 0.9
1	110	0.9 - 3.0
1	111	3.0 - 8.0
0	101, 110, 1111	1.0<

Clock source in ATmega 32

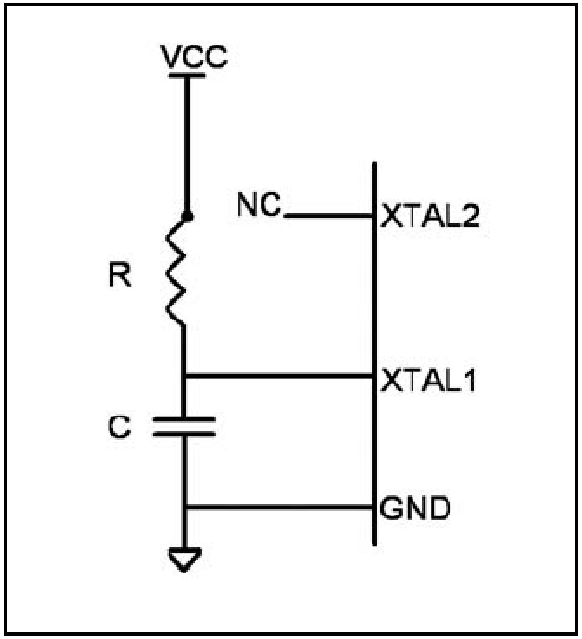


Figure 8-5 External RC

Table 8-8 Internal RC Oscilator Operation Modes

CKSEL0..3	Frequency
0001	1 MH
0010	2 MH
0011	4 MH
0100	8 MH

Table 8-9 External RC Oscilator Operation Modes

CKSEL0..3	Frequency(MH)
0101	<0.9
0110	0.9- 3.0
0111	3.0- 8.0
1000	8.0- 12.0

Power on Reset and Burn on Detection

Table 8-11: Startup time for crystal oscilator and recommanded usage

CKSEL0	SUT1..0	Start-UpTime From Power Down	Delay From Reset(VCC=5)	Recommended Usage
0	00	258CK	4.1	Ceramic resonator, fast rising power
0	01	258CK	65	Ceramic resonator, slowly rising power
0	10	1K CK	-	Ceramic resonator, BOD enabled
0	11	1K CK	4.1	Ceramic resonator, fast rising power
1	00	1K CK	65	Ceramic resonator, slowly rising power
1	01	16K CK	-	Crystal Oscilator, BOD enabled
1	10	16K CK	4.1	Crystal Oscilator, fast rising power
1	11	16K CK	65	Crystal Oscilator, slowly rising power

Golden Rule of Fuse bits

- If you are using an external crystal with a frequency more than 1MHz, you can set all of the CKSEL3, CKSEL2, CKSEL1, SUT1 and SUT0 to 1 and clear CKOPT to 0.

Inside an HEX file

```
:020000020000FC
:1000000008E00EBF0FE50DBF05E5009508BB0E9497
:100010000A00FBCF40E158EC6AEF000000006A954F
:0C002000E1F75A95C9F74A95B1F7089529
:00000001FF


Separating the fields, we get the following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH          CC
:02 0000 02 0000                                      FC
:10 0000 00 08E00EBF0FE50DBF05E5009508BB0E94          97
:10 0010 00 0A00FBCF40E158EC6AEF000000006A95          4F
:0C 0020 00 E1F75A95C9F74A95B1F70895                  29
:00 0000 01                                           FF
```

Figure 8-7. Intel Hex File Test Program with the Intel Hex Option

Inside an HEX file

```
:020000020000FC
:100000008E00EBF0FE50DBF05E5009508BB0E9497
:10001000A00FBCF40E158EC6AEF000000006A954F
:0C002000E1F75A95C9F74A95B1F7089529
:00000001FF
```

Each line starts with a colon.

how many bytes are in the line.

Separati… the fields … get the following:

```
:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH          CC
:02 0000 02 0000                                      FC
:10 0000 00 08E00EBF0FE50DBF05E5009508BB0E94          97
:10 0010 00 0A00FBCF40E158EC6AEF000000006A95          4F
:0C 0020 00 E1F75A95C9F74A95B1F70895                  29
:00 0000 01                                           FF
```

Figure 8-7. Intel Hex File Test Program with the Intel Hex Option

Inside an HEX file

This is a 16-bit address; The loader places the first byte of data into this memory address. It can address 64k locations

Type of line:
00: there are more lines to come after This line.
01: this is the last line
02: Segment address

```
:0200
:100000000      0FE50DBF05E5009508BB0E9497
:10001000      CF40E158EC
:0C002000      A95C9F74A95
:0000000

Separat ng the fields, we        following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH         CC
:02 0000 02 0000                                     FC
:10 0000 00 08E00EBF0FE50DBF05E5009508BB0E94         97
:10 0010 00 0A00FBCF40E158EC6AEF000000006A95         4F
:0C 0020 00 E1F75A95C9F74A95B1F70895                 29
:00 0000 01                                          FF
```

Figure 8-7. Intel Hex File Test Program with the Intel Hex Option

Inside an HEX file

```
:020000020000FC
:100000008E00EBF0FE50DBF05E5009508BB0E9497
:100010000A00FBCF40E158EC6AEF000000006A954F
:0C002000E1F75A95C9F74A95B1F7089529
:00000001FF

Separating the fields we get the following:

:BB AAAA TT HHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHH       CC
:02 0000 02 0000                                    FC
:10 0000 00 08E00EBF0FE50DBF05E5009508BB0E94        97
:10 0010 00 0A00FBCF40E158EC6AEF000000006A95        4F
:0C 0020 00 E1F75A95C9F74A95B1F70895                29
:00 0000 01                                         FF
```

Real Data

Checksum

Figure 8-7. Intel Hex File Test Program with the Intel Hex Option

Inside an HEX file

- Verify the checksum byte for line 4:
  - 0C + 00 +20 + 00 + E1 + F7 + 5A + 95 + C9 + F7 + 4A + 95 + B1 + F7 + 08 + 95 = 7D7H
  - The carry is dropped and checksum = 100H – D7H = 29H.
- Verify also the information is not corrupted:
  - 0C + 00 +20 + 00 + E1 + F7 + 5A + 95 + C9 + F7 + 4A + 95 + B1 + F7 + 08 + 95 + ***29*** = 800H
  - The carry is dropped and the result = 00H.

Class Exercise 2

- The first line of the content of a .hex file is shown below:

  :0200000803E

- Find the checksum.

Class Exercise 2 (Your work)

Class Exercise 2 (Answer)

Reference Readings

- Chapter 3, 5, 8, and 9 – *The AVR Microcontroller and Embedded Systems : Using Assembly and C*, M. A. Mazidi, S. Naimi, and S. Naimi, Pearson, 2014.

End