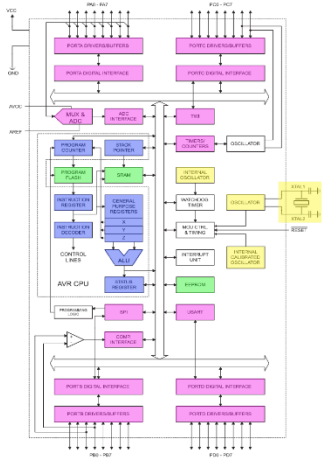
به نام خدا

آشنایی با زبان اسمبلی AVR

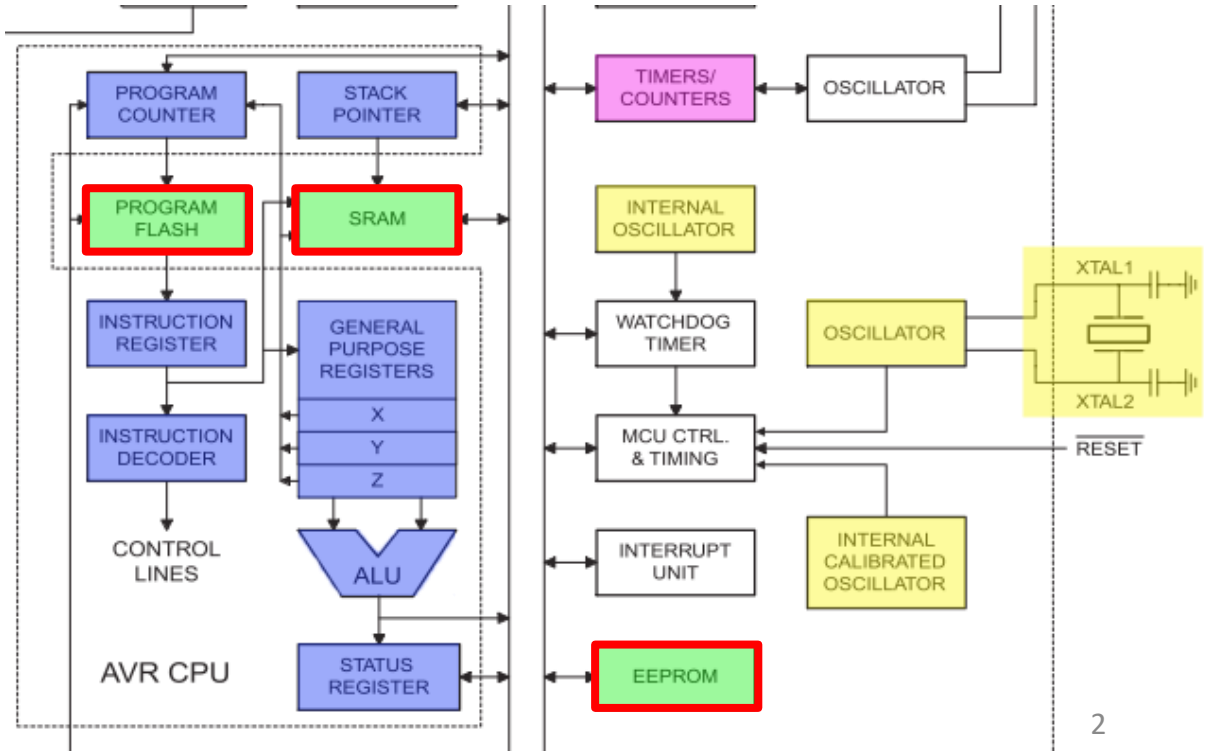
کار با حافظه

Dr. Aref Karimiafshar
A.karimiafshar@ec.iut.ac.ir

کار با حافظه



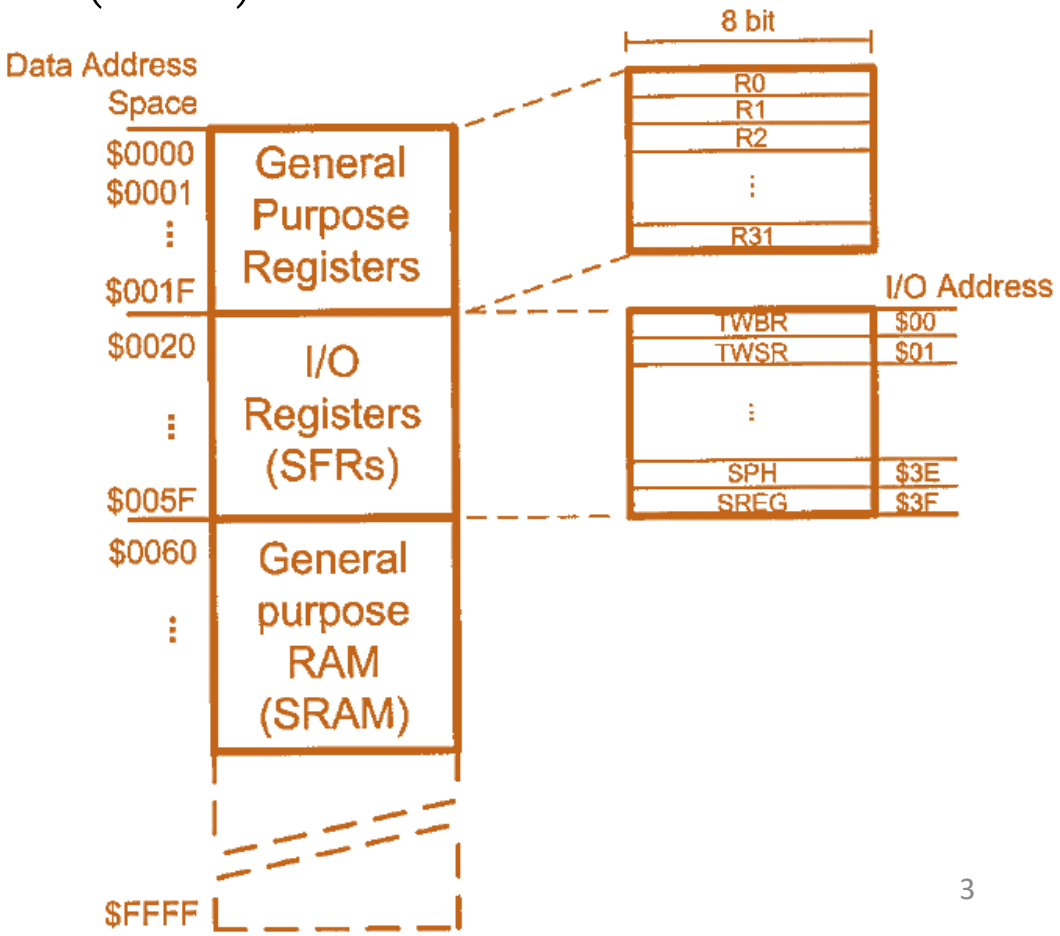
- Two kinds of memory space in AVR:
  - Code memory space
    - Stores our program
  - Data memory space
    - Stores data



به صورت کلی میکروکنترلر avr دو نوع فضای حافظه در اختیار ما قرار میده:
فضای حافظه مرتبط با کد: که توی اون برنامه ذخیره میشه
و فضای حافظه مرتبط با داده: که توی اون داده ها چه به صورت دائم و چه به صورت موقت و برای انجام محاسبات ذخیره میشه و در انجام محاسبات به ALU کمک خواهد کرد
توش شکل زیر:
حافظه ها با رنگ سبز مشخص شده
حافظه برنامه رو داریم که از نوع فلش است : حافظه فلش برای ذخیره سازی برنامه استفاده میشه
در واقع همون محلی است که داده ها و کد برنامه ذخیره میشه و از روی اون تک تک دستورات وارد CPU میشن و اجرا میشن
حافظه SRAM رو داریم : برای ذخیره سازی موقت داده ها استفاده میشه
و حافظه EEPROM : برای ذخیره دائمی اطلاعات استفاده میشه

حافظه داده

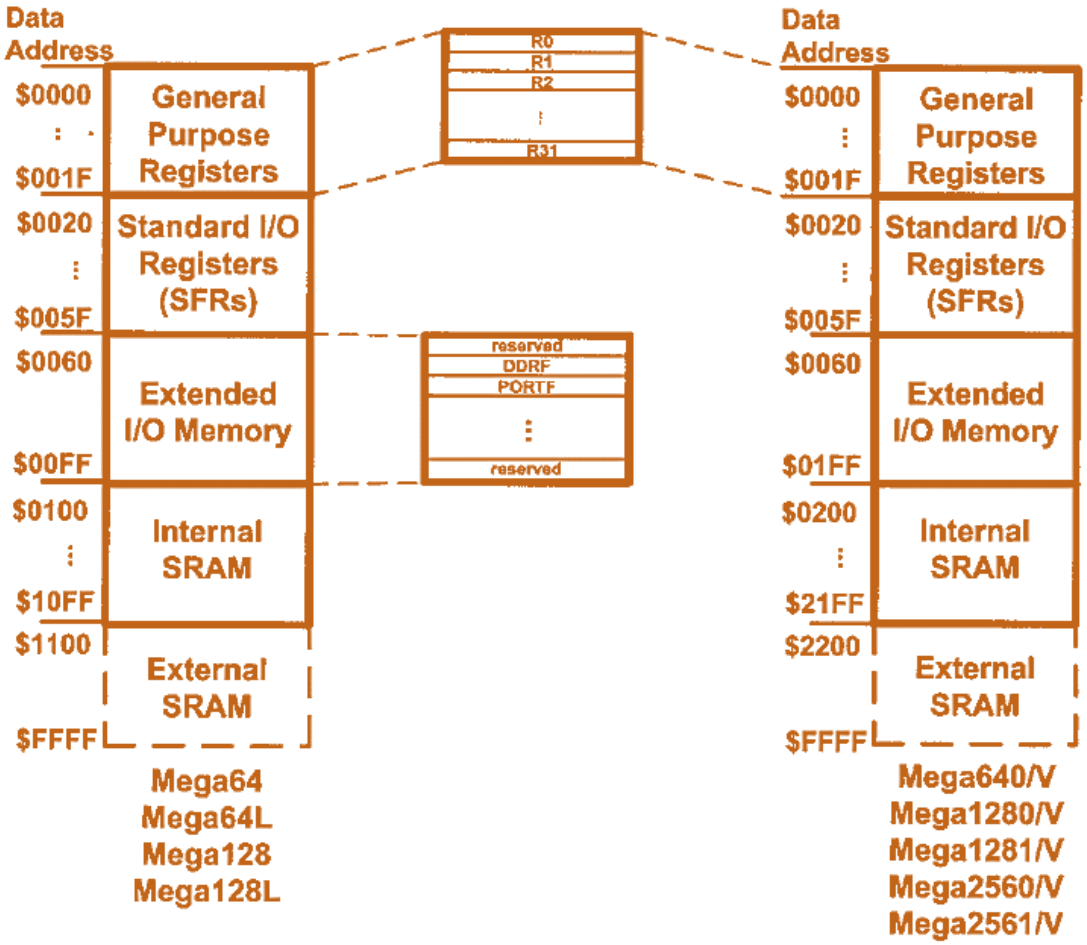
- Data memory space
  - General Purpose Registers (GPRs)
    - 32x8 registers
  - I/O memory
    - Status register
    - Timer
    - Serial communication
    - I/O ports
    - ADC
    - …
  - Internal data SRAM



به صورت کلی :
فضای کلی مرتبط با حافظه ی داده رو می تونیم به سه قسمت تقسیم کنیم:
رجیسترهای همه منظوره
فضای مرتبط با I/O که توی این قسمت رجیستر وضعیت داریم و یکسری رجیستر مرتبط با تایمرها رو داریم و عملکردهای سریال و رجیسترهای پورت ها و ADC و ..
یک قسمت از این فضای داده مرتبط میشه با خود SRAM اون چیزی که به صورت onchip و داخلی در میکروکنترلر وجود داره
پس فضای داده ما یک فضای پیوسته هستش که در ابتدای این فضا رجیسترها قرار دارد ینی 32 تا رجیستر همه منظوره داریم که از صفر تا 1F اینا نیاز به ادرس دارن
بعد از این فضا رجیسترهای I/O قرار دارند و بعد SRAM و بعد یک SRAM خارجی اگر بخوایم به این سیستم اضافه بکنیم

حافظه I/O

- I/O memory
  - Special Function Registers (SFRs)
  - Number of locations in data mem. set aside for I/O mem depends on the pin numbers and peripheral functions
  - However, all AVRs have at least 64 bytes of I/O mem
    - Called standard I/O mem
  - AVRs with more than 32 I/O pins
    - Have an extended I/O mem



حافظه I/O:
این قسمت شامل رجیسترهای خاص منظوره میشه که برای یکسری عملکردهای خاصی دیده شدن به صورت خیلی خاص اگر بخوایم بگیم: رجیستر وضعیت یک رجیستری که در این فضا قرار داره تعداد محل هایی که توی این قسمت از حافظه کنار گذاشته میشه وابسته هستش به نوع اون میکروکنترلر و امکانات جانبی و عملکردهای جانبی و تعداد پایه هایی که اون میکروکنترلر قرار در اختیار قرار بده
ه صورت کلی و جنرال در اکثریت این میکروکنترلرهای AVR ما 64 بایت فضا به صورت مجزا برای I/O در نظر گرفتیم که به این قسمت میگن استاندارد I/O ینی در واقع یکسری رجیسترهای معروف و مشخصی هستند که در بین همه این خانواده های AVR وجود دارند
حالا ممکنه یکسری خانواده ها و یکسری میکروکنترلر هایی امکانات بیشتری داشته باشند که این ها رو به صورت Extended I/O memory در ادامه این فضای قرار میدیم
پس توی فضای I/O یکسری رجیسترهای عمومی تری داریم که توی همه خانواده های AVR مشترک هستش ینی standard I/O register و یکسری رجیسترهای خاص تری داریم که وابسته به نوع امکانات و تعداد پایه ها در بعضی از میکروکنترلرها وجود داره ینی extendedn I/O memory
توی خود این میکروکنترلرهایی که فضای extendend دارند ممکنه این متفاوت باشه مثلا توی mega64 این extendend اش از 60 هگز تا FF هگز است ولی مثلا توی سری های mega640 این از 60 هگز تا 1F هگز هستش که این باز وابسته به نوع امکانات و تعداد پایه های و رجیسترهای مورد نیازی که برای عملیات های مرتبط با اننها نیاز است این فضا تخصیص داده میشه

SRAM حافظه

- Internal data SRAM
  - Storing data and parameters
    - Called scratch pad
  - Each location of SRAM can be accessed directly by its address
  - Each location is 8-bit wide
  - Size of SRAM can vary from chip to chip

Data Memory Size for AVR Chips

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

در ادامه فضای داده:
SRAM: اون فضایی هستش که پارامترها و داده های ما به صورت موقت ذخیره میشن و CPU , ALU با اونها در ارتباطه مثل چک نویس می مونه که ما یکسری محاسبات می خوایم انجام بدیم و اینجا یادداشت می کنیم
هر محلی از SRAM توسط یک ادرس به صورت مشخص ادرس دهی می شه و عرض داده ای که SRAM در AVR در اختیار ما قرار میده 8 بیت هستش اما سایز و اندازه این SRAM وابسته به تراشه ای که داریم استفاده می کنیم متفاوت است
توی جدول روبه رو سایز SRAM رو برای چندتا تراشه مختلف از AVR گفته
گفتیم فضای داده به سه بخش تقسیم میشه که توی جدول هم می تونیم اینو ببینیم

LDS – Load Direct from Data Space

- Loads one byte from the data space to a register.

(i) Rd ← (k)

	Syntax:	Operands:	Program Counter:
(i)	LDS Rd,k	0 ≤ d ≤ 31, 0 ≤ k ≤ 65535	PC ← PC + 2

32-bit Opcode:

1001	000d	dddd	0000
kkkk	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words 2 (4 bytes)

Cycles 2

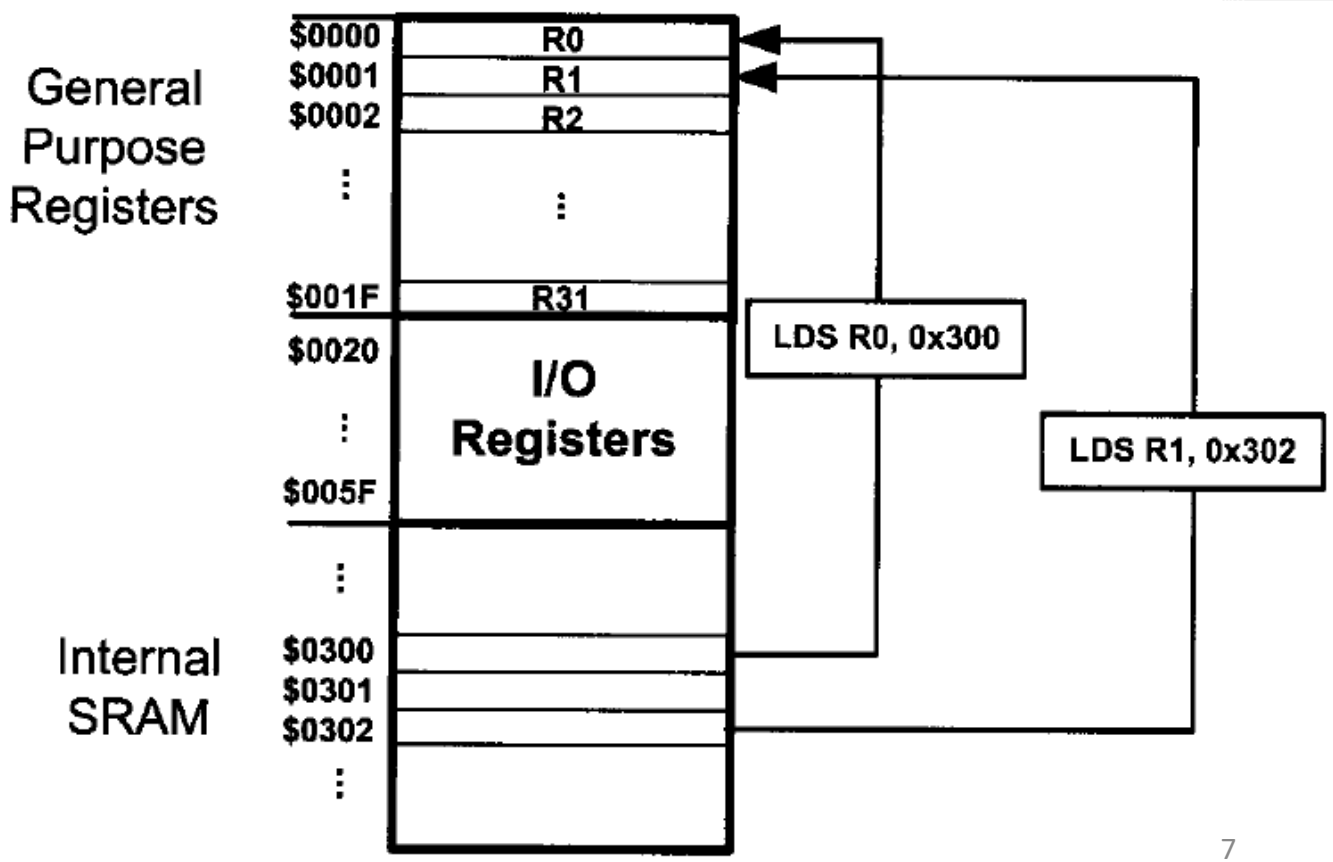
برای کار کردن با حافظه و اینکه یک مقداری رو در حافظه ذخیره بکنیم از دستور روبه رو می ریم و این دستور میاد یک بایت رو از یک ادرسی توی حافظه می خونه و میاد داخل یک رجیستر قرار میده

این K اون محلی هستش که داره به اون خونه مورد نظر ما از حافظه اشاره می کنه و وقتی پرانتز می ذاریم ینی محتوای K رو ینی (K) داخل رجیستر Rd قرار بده

K می تونه یک عدد 16 بیتی باشه

LDS – Example

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



مثال:
مثلا می خوایم محتوای خونه 300 هگز رو برداریم و بریزیم توی R0 و محتوای خونه 302 حافظه رو بریزیم توی رجیستر R1
اتفاقی که می افته اینه که وقتی که دستور اول رو اجرا میکنیم میاد محتوای خونه 300 هگز رو می ریزه توی R0
فضای داده ما یک همچین فضای پیوسته ای هستش
ما از قسمت SRAM داده رو داده رو برمیداریم و می ریزیم توی رجیسترهای همه منظوره چرا؟
چون عملگرها و دستوراتی که داریم با رجیسترها کار می کنند توی معماری ریسک و AVR

STS – Store Direct to Data Space

- Stores one byte from a Register to the data space.

(i) (k) ← Rr

	Syntax:	Operands:	Program Counter:
(i)	STS k,Rr	0 ≤ r ≤ 31, 0 ≤ k ≤ 65535	PC ← PC + 2

32-bit Opcode:

1001	001d	dddd	0000
kkkk	kkkk	kkkk	kkkk

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words 2 (4 bytes)

Cycles 2

این دستور میاد محتوای یک رجیستر رو در یک خونه ای از حافظه که اینجا میشه K قرار میده

I/O Registers

- Each location in I/O memory has two addresses
  - I/O address
  - Data memory address

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

دستوراتی که برای کار با I/O ها داریم:
توی فضای I/O ما از 0 تا 64 میایم ادرس دهی میکنیم و میایم به صورت مجزا بهشون یکسری ادرس میدیم
پس مثلا اگر بخوایم توی فضای I/O کار بکنیم وقتی بگیم 00 این معادل اینکه توی فضای دیتا ممووری خونه 20 هگز رو بریم دسترسی پیدا بکنیم و همینطور برای اینکه کار با اینها راحت تر باشه بهشون یکسری اسم هم نسبت دادن

IN - Load an I/O Location to Register

- Loads data from the I/O Space (Ports, Timers, Configuration Registers, etc.) into register Rd in the Register File.

(i) $Rd \leftarrow I/O(A)$

	Syntax:	Operands:	Program Counter:
(i)	IN Rd,A	$0 \le d \le 31, 0 \le A \le 63$	$PC \leftarrow PC + 1$

16-bit Opcode:

1011	0AAd	dddd	AAAA

Status Register (SREG) and Boolean Formula

I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words	1 (2 bytes)
Cycles	1

این دستور مختص کار با فضای I/O هستش و میاد یک مقداری رو از ورودی می گیره ینی از رجیسترهای مرتبط با فضای I/O میگیره و در یک رجیستر همه منظوره قرار میده پس مشخص میکنیم که از کدوم رجیستر توی فضای I/O می خوایم بخونیم و اونو می خونیم و توی یک رجیستر همه منظوره می ریزیم

IN - Load an I/O Location to Register

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

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

The following program adds the contents of PIND to PINB, and stores the result in location 0x300 of the data memory:

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

برای اینکه کار با این دستور و این رجیسترها ساده تر باشه بهشون یک اسم می دن مثلا به جای اینکه 10 هگز ورودی I/O بذاریم از PIND استفاده میکنیم
مثال:
از پورت D یک ورودی می خونیم و داخل R1 می ریزیم و پورت B می خونیم و توی R2 می ریزیم و محتوای این دوتارو جمع می کنیم و توی خونه 300 هگز قرار می دیم

IN vs. LDS

- IN is faster than LDS
  - IN lasts 1 MC, LDS lasts 2 MC
- IN occupies less memory
  - IN is 2-Byte instruction, LDS is 4-Byte instruction
- When we use IN
  - We can use the names of I/O registers
- IN is available in all AVRs, LDS implemented in some

هر دوی این ها میاین یک خونه از حافظه رو برمیدارن توی فضای دیتا مموری اسپیس ما و می ریزن توی رجیستر همه منظوره
اما تفاوت هایی با هم دارند:
1- IN سریعتر انجام می شه چون دستور IN توی یک سیکل ماشین انجام میشه و LDS توی دوتا
2- دستور IN فضای کمتری هم مصرف میکنه چون اپکدش 16 بیتی است ولی LDS یک اپکد 32 بیتی یا 4 بایتی داشت
3- وقتی که داریم با دستور IN کار میکنیم می تونیم از نام رجیسترها استفاده بکنیم ولی این قضیه برای دستور LDS صدق نمیکنه
4- دستور IN توی همه سری های AVR وجود داره ولی LDS توی بعضی از این سری ها پیاده سازی شده
پس اگر دستوری نیاز داریم که بخواد با I/O کار بکنه این هم از طریق IN , هم از طریق LDS امکان پذیره ولی به صورت خاص IN برای کار با رجیسترهای I/O طراحی شده و خیلی سریعتر و فضای کمتری اشغال میکنه پس تا زمانی که نیازی به LDS پیش نیومده ضرورتی نداره از اون استفاده بکنیم

OUT – Store Register to I/O Location

- Stores data from register Rr in the Register File to I/O Space (Ports, Timers, Configuration Registers, etc.).

(i)	I/O(A) ← Rr		
	Syntax:	Operands:	Program Counter:
(i)	OUT A,Rr	0 ≤ r ≤ 31, 0 ≤ A ≤ 63	PC ← PC + 1

16-bit Opcode:

1011	1AAr	rrrr	AAAA

Status Register (SREG) and Boolean Formula

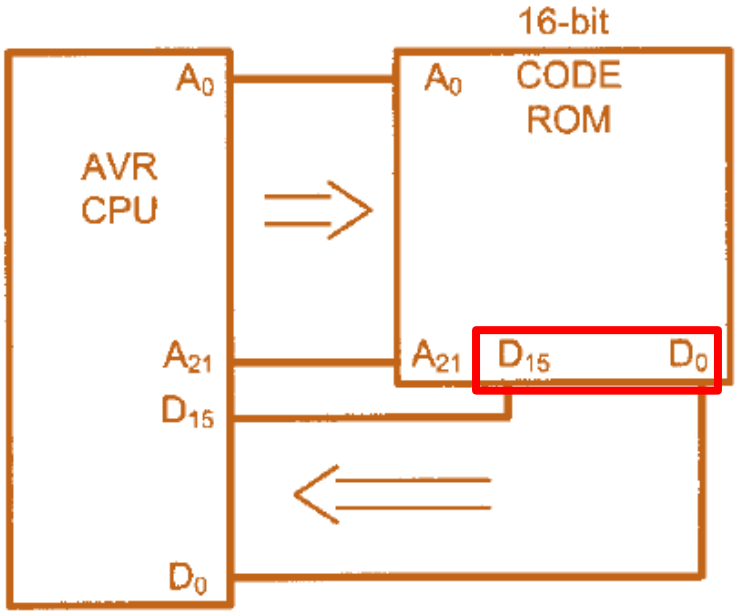
I	T	H	S	V	N	Z	C
–	–	–	–	–	–	–	–

Words	1 (2 bytes)
Cycles	1

این دستور میاد یک مقداری رو در رجیسترهای I/O ذخیره میکنه
R یک رجیستر همه منظوره است و A یکی از اون رجیسترهای خاص منظوره مرتبط با I/O ما هستش

Program ROM Space

- Program counter
  - Point to the address of next instruction
- Program counter width
  - The wider program counter, the wider address space
- In AVR microcontroller
  - Each Flash mem. location is 2 bytes
  - Example: ATmega32 → 32KB Flash
    - Organized as 16Kx16
    - PC → 14 bits wide



حافظه برنامه و فضای ادرس مرتبط با کد:
اولین المانی که تاثیرگذار است و با اون کاری میکنیم که دستیابی پیدا بکنیم به فضای ادرس Program counter است : که یک رجیستر که ادرس دستورات در اون قرار میگیره و با استفاده از این ادرس می تونیم به دستورات مختلف در حافظه ارجاع پیدا بکنیم
عرض این pc یا Program counter که تعیین میکنه ما به چه فضایی از حافظه بتونیم دسترسی پیدا بکنیم هرچه عرض این PC ما بزرگتر باشه ما می تونیم به فضای ادرس بیشتری ارجاع بدیم
نکته: این به صورت فلش پیاده سازی میشه و هر محلی از اون شامل 2 بایت هستش ینی با هر ادرسی که روی باس قرار میگیره به 2 بایت ازاین حافظه دسترسی پیدا خواهیم کرد
پس وقتی میگیم که ATmega32 ما 32 کیلو بایت فلش داره این سازمان و چینش حافظه اون به صورت 2 بایت کنار هم دیگه است ینی میشه 32 خونه 8 بیتی که اگر بخوایم به صورت 16 بیتی سازماندهی بشه میشه 16K خونه 16 بیتی و این دلیلش هم در این مسئله ریشه داره که عرض اکثریت دستورات در سری های مختلف AVR با توجه به اینکه ریسک هستن 16 بیت هستش ینی اپکدهای 16 بیتی ما داریم پس منطقیه که عرض حافظه برنامه 16 بیتی در نظر بگیریم
در مورد ATmega32 که 32 کیلوبایت فلش داره وسازمان اون به صورت 16k در 16 بیت هستش ما فقط به 14 بیت نیاز داریم که بتونیم به این فضا ادرس دهی بکنیم
نکته ای که در مورد حافظه برنامه توی avr وجود داره ما باس مجزا برای اون داریم با توجه به اینکه معماری هاردوارد داریم
ما ادرس رو که می تونه تا 22 بیت باشه توی PC قرار میدیم و این ارجاع پیدا میکنه به حافظه ما و یک خونه رو مشخص میکنه و عرض داده ای هم که به ما به عنوان خروجی میده و دستور مارو مشخص میکنه 16 بیت هستش که اون همون 16 بیت اپکد ما هست که میاد وارد CPU میشه و دیکد میشه و قراره اون دستور مارو اجرا بکنه

Program ROM Space

AVR On-chip ROM Size and Address Space

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

- In AVR microcontroller
  - PC can be up to 22 bits wide
  - Access program address 000000 to $3FFFFF
    - Total of 4M locations ---> 8M bytes on-chip ROM

در سری های مختلف تراشه های AVR ما میزان متفاوتی از رام رو می بینیم که در درون اون ها تعبیه شده مثلا atmega8 ما 8 کیلوبایت داره
توی میکروکنترلرهای avr ما Program counter چیزی که درنظر گرفته حداکثر 22 بیت هستش که این می تونه یک فضایی از 000000 هگز تا 1fffff هگز به ما ادرس دهی کنه
و چون خونه ها به صورت 16 بیتی یا 2 بایتی سازماندهی میشن چیزی نزدیک به 8 مگابایت می تونیم حافظه حداکثر داشته باشیم

Assembler Directives

- Give directions to the assembler
- Directives help us
  - Develop our program easier
  - Make our program legible (more readable)

- Directives
  - .EQU
  - .SET
  - .ORG
  - …

```
.EQU COUNTER = 0x00
.EQU PORTB = 0x18
LDI  R16, COUNTER
OUT  PORTB, R16
```

دستورات add , or , and ,... این ها مستقیما توسط cpu اجرا میشن ولی Directives ها شبه کدهایی هستند که فقط مورد استفاده assembler قرار میگیرند و قرار نیست توسط cpu اجرا بشن و کاری رو بخواین انجام بدن
به صورت واضح و خیلی شفاف میشه اینه که یکسری راهنمایی هایی رو در اختیار assembler قرار میده و این ها باعث میشه که ما بتونیم با سهولت بیشتری کد بزنیم و کد ما خوانا تر بشه
نکته: Directives ها با نقطه شروع میشن

Assembler Directives

.EQU

- Define a constant value
- Does not set aside storage for a data item
- Associate a constant number with a label

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

.SET

- Define a constant value
- Like .EQU; difference → may be reassigned later

EQU : با استفاده از این Directives می تونیم یک نام برای یک مقدار ثابتی در نظر بگیریم ینی یک لیبل داریم درست میکنیم و به جای اینکه مستقیم مقدار 0X25 در کد استفاده بکنیم از COUNT استفاده میکنیم و این به ما کمک میکنه اگر خواستیم مقدار رو عوض بکنیم نیاز نباشه همه برنامه رو تغییر بدیم و فقط کافیه مقدار روبه روی COUNT رو تغییر بدیم
نکته: فضای حافظه ای برای Directives کنار گذاشته نمیشه و ابتدای برنامه این لیبل با مقدار منتاظر با اون جایگزین میشه و بقیه کاری که دیگه قراره assembler انجام بده

.SET: برای داده ما لیبل تعریف میکنه مثل EQU با این تفاوت که ممکنه در ادامه مقدار دهی مجدد بشه

Assembler Directives

.ORG

- Indicate the beginning of the address

.INCLUDE

- Add the contents of a file to our program

- When you want to use ATmega32
  - You must write the following at the beginning of your program

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

ORG: مشخص میکنه که شروع کد ما از چه ادرسی باشه

INCLUDE: می تونیم محتوای یک فایل رو به برنامه اضافه بکنیم مثلا وقتی که بخوایم از ATmega32 استفاده بکنیم و برای اون برنامه نویسی بکنیم میایم اینو اینکلود میکنیم ینی یک فایل از پیش تعریف شده رو به برناممون اضافه میکنیم

Assembler Directives

.DB

- Allocate program memory in byte-sized chunks
- The number can be:
  - Binary `.DB 0xb0101`
  - Decimal `.DB 28`
  - Hex `.DB 0xA`
  - ASCII `.DB 's'`

.DW

- Allocate program memory in word-sized chunks

DB: یک قسمتی از حافظه رو به صورت مشخص با یک مقداری پر میکنه و این به صورت بایت انجام میشه - این مقدار ها به صورت بایت توی حافظه ذخیره میشن و وقتی که ما داریم این کارو برای حافظه برنامه انجام می دیم می دونیم که طول اون 16 بیت یا 2 بایت هستش بنابراین اتفاقی که می افته با هر بار اختصاص دادن DB یک قسمت از اون رو پر میکنیم و زمانی که دوباره از DB استفاده میکنیم قسمت دومش هم پر میشه و وقتی که 16 بیت ما تمام شد می ریم سراغ 16 بیت بعد و اگر تعداد زوج باشه مثل اینجا که الان مثال زده همه خونه پر میشه ولی اگر تعداد فرد بود مثلا سه تا باشه خانه اول 2 بایتش پر میشه و خانه دوم فقط یه بایتش پر میشه و بقیش با صفر پر خواهد شد

DB 0X0101 : اگر مشخص نکرده باشیم که از چه خونه ای میخوایم این کارو انجام بدیم مثلا از خونه صفر می ره و این مقدار توش ذخیره میشه

DW: این هم مثل بالایی است با این تفاوت که به صورت word انجام میشه

Assembler Directives

.ESEG

- Variable will be located in EEPROM

```
.ESEG
    .DB 0b0101
    .DB 0xE
```

.CSEG

- Variable will be located in Code memory

```
.CSEG
    .DB 0b0101
    .DB 0xE
```

ESEG: این به ما کمک میکنه که تعاریفی که برای مقادیر ثابت داریم در حافظه EEPROM قرار بگیره ینی اگر قبل از DB از این دستور استفاده بکنیم این مقادیر می رن و در حافظه EEPROM ذخیره میشن

CSEG: اگر از این دستور استفاده بکنیم این مقادیر می رن و در حافظه کد برنامه ذخیره میشن

Assembler Directives

.DSEG

- Variable will be located in SRAM

```
.DSEG
    .DB 0b0101
    .DB 0xE
```

.EXIT

- Stop the execution

DSEG: اگر از این دستور استفاده بکنیم این مقادیر به صورت ثابت در حافظه داده ذخیره میشن

EXIT: این باعث میشه که اجرای دستورات متوقف بشه و ادامه برنامه تداوم پیدا نکنه

Assembly Language Ins. Format

- Assembly language instructions consist of four fields:

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

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

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

به صورت کلی:
فرمتی که تا اینجا باهاش اشنا شدیم اینه که یک خط کد ما می تونه با یک لیبل شروع بشه و با :
خاتمه پیدا میکنه و بعد سیمبل دستور رو داریم مثل ADD ,.. و بعد اپرندهارو داریم و بعد می تونیم
کامنت بذاریم با ;

EEPROM حافظه

- Data in SRAM will be lost if power is disconnected
- EEPROM can save stored data even if power is cut off
- Accessing EEPROM in AVR
  - Three I/O registers, directly related to EEPROM
    - EECR (EEPROM Control Register)
    - EEDR (EEPROM Data Register)
    - EEARH : EEARL (EEPROM Address Register High-Low)

Size of EEPROM Memory in ATmega Family

Chip	Bytes	Chip	Bytes	Chip	Bytes
ATmega8	512	ATmega16	512	ATmega32	1024
ATmega64	2048	ATmega128	4096	ATmega256RZ	4096
ATmega640	4096	ATmega1280	4096	ATmega2560	4096

حافظه EEPROM
نکته: داده هایی که داخل SRAM ذخیره میشن با قطع برق و خاموش کردن دستگاه پاک میشن ینی داده ها به صورت موقت اونجا ذخیره میشن ولی EEPROM می تونه داده هارو به صورت دائمی اونجا ذخیره بکنه که با قطع شدن برق هم داده ها از EEPROM پاک نمیشن
نحوه کار و استفاده از EEPROM:
درباره EEPROM سه رجیستر وجود داره که مستقیما با EEPROM کار میکنند:
EECR: دستورات و فرمان های کنترلی در خودش جا میده
EEDR: داده ای که قراره بین EEPROM و فضای خارج از اون صورت بگیره از طریق این رجیستر انجام میشه
EEARH: برای مشخص کردن ادرس اون محلی از حافظه EEPROM هستش که قراره داده توش بنویسییم یا بخونیم

رجیسترهای کار با EEPROM

- EEDR (EEPROM Data Register)
  - To write data to EEPROM, you have to write it to EEDR
    - Then transfer it to EEPROM
  - To read data from EEPROM, you have to read from EEDR

- EEARH : EEARL (EEPROM Address Register High-Low)
  - Together make a 16-bit reg. to address each location in EEPROM
    - 10 bits are used in Atmega32

Bit	15	14	13	12	11	10	9	8
EEARH	-	-	-	-	-	-	EEAR9	EEAR8
EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
Bit	7	6	5	4	3	2	1	0

اگر بخوایم کار بکنیم با این EEPROM باید با اون رجیسترهای سه گانه ای که درباره اونها صفحه قبل گفتیم استفاده بکنیم و از طریق اونها کارهارو انجام بدیم
EEDR: هر داده ای که بخوایم به حافظه EEPROM منتقل بکنیم از طریق این رجیستر انجام میشه ینی اگر بخوایم بنویسیم توی EEPROM اول داده را باید توی این رجیستر بنویسیم و بعد از توی این رجیستر منتقل میشه به EEPROM یا اگر بخوایم از حافظه EEPROM بخونیم اول مقدار اون خانه حافظه از EEPROM به این رجیستر منتقل میشه و بعد از این رجیستر به رجیسترهای همه منظوره یا حافظه قابل انتقاله
این رجیستر مثل یک پلی بین حافظه EEPROM و CPU یا حافظه های دیگه نقش داره

EEAR: از دو قسمت high , low تشکیل شده که قسمت high اش با H مشخص میشه ینی EEARH و قسمت low اش با L ینی EEARL
این دو قسمت با همدیگه تشکیل 16 بیت ادرس رو میدن که امکان ادرس دهی فضای EEPROM را به ما خواهد داد
مثلا ATmega32 که گفتیم 1024 بایت حافظه EEPROM داره ما فقط به 10 بیت این رجیستر نیاز داریم

رجیسترهای کار با EEPROM

- EECR (EEPROM Control Register)
  - Select the kind of operation to perform
    - Start
    - Read
    - Write

  - EERE→ EEPROM Read Enable
  - EEWE→ EEPROM Write Enable

EECR: این رجیستر به ما کمک میکنه که اون عملی که قصد داریم انجام بدیم رو با فعال کردن یکسری بیت هایی در این رجیستر مشخص بکنیم
مثلا زمانی که بخوایم بخونیم بیت صفر را فعال می کنیم و زمانی که بخوایم بنویسیم بیت 1 را فعال میکنیم و ....

نوشتن EEPROM

To write on EEPROM the following steps should be followed. Notice that steps 2 and 3 are optional, and the order of the steps is not important. Also note that you cannot do anything between step 4 and step 5 because the hardware clears the EEMWE bit to zero after four clock cycles.

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Write new EEPROM data to EEDR (optional).
4. Set the EEMWE bit to one (in EECR register).
5. Within four clock cycles after setting EEMWE, set EEWE to one.

نوشتن توی EEPROM:
مراحلش:
1- اول صبر میکنیم که اگر نوشتنی از قبل هست این کارش به اتمام برسه و بیت EEWE صفر بشه
2- بعد می تونیم ادرس و داده ای که قراره مورد استفاده قرار بگیره برای نوشتن رو توی حافظه های ادرس و دیتا قرار بدیم (ترتیب اینها خیلی مهم نیست ینی اول می تونیم داده را در رجیستر دیتا ینی EEDR و بعد ادرس در رجیستر داده ینی EEAR قرار بدیم) میشه شماره 2 و 3
4- بعد از اینکه این ها مقدارشون مشخص شد بیت EEMWE فعال میشه
5- و بعد از فعال شدن اون بیت 4 سیکل ساعت طول میکشه که EEWE فعال بشه و اون مقدار در حافظه نوشته بشه

سناریویی که اینجا مطرح کردیم نیاز به برنامه نویسی داره که صفحه بعد گفته...

نوشتن EEPROM

Write an AVR program to store 'G' into location 0x005F of EEPROM .

Solution:

```
.INCLUDE "M16DEF.INC"

WAIT:                ;wait for last write to finish
SBIC   EECR,EEWE     ;check EEWE to see if last write is finished
RJMP   WAIT          ;wait more
LDI    R18,0         ;load high byte of address to R18
LDI    R17,0x5F      ;load low byte of address to R17
OUT    EEARH, R18    ;load high byte of address to EEARH
OUT    EEARL, R17    ;load low byte of address to EEARL
LDI    R16,'G'       ;load 'G' to R16
OUT    EEDR,R16      ;load R16 to EEPROM Data Register
SBI    EECR,EEMWE    ;set Master Write Enable to one
SBI    EECR,EEWE     ;set Write Enable to one
```

مثال:
یک ATmega16 داریم و میخوایم توی حافظه اش بنویسیم
یک حلقه می ذاریم که چک بکنه که ایا اخرین عملیات نوشتن به اتمام رسیده یا نه پس میاد بیت eewe رو چک میکنه و اگر صفر بود دستور بعدی خودش رو پرش میکنه و میاد سراغ LDI

خواندن از EEPROM

To read from EEPROM the following steps should be taken. Note that step 2 is optional.

1. Wait until EEWE becomes zero.
2. Write new EEPROM address to EEAR (optional).
3. Set the EERE bit to one.
4. Read EEPROM data from EEDR.

خواندن از EEPROM:
1- باید صبر بکنیم که اخرین نوشتن تموم بشه
2- بعد باید ادرسی که قراره از اون بخونیم رو مشخص میکنیم و توی EEAR قرار میدیم
3- و بعد بیت خواندن را فعال میکنیم ینی EERE
4- عملیات خواندن انجام میشه

خواندن از EEPROM

Write an AVR program to read the content of location 0x005F of EEPROM into PORTB.

Solution:

```
.INCLUDE "M16DEF.INC"
      LDI   R16,0xFF
      OUT   DDRB,R16
WAIT:                     ;wait for last write to finish
      SBIC  EECR,EEWE     ;check EEWE to see if last write is finished
      RJMP  WAIT          ;wait more
      LDI   R18,0         ;load high byte of address to R18
      LDI   R17,0x5F      ;load low byte of address to R17
      OUT   EEARH, R18    ;load high byte of address to EEARH
      OUT   EEARL, R17    ;load low byte of address to EEARL
      SBI   EECR,EERE     ;set Read Enable to one
      IN    R16,EEDR      ;load EEPROM Data Register to R16
      OUT   PORTB,R16     ;out R16 to PORTB
```

مثال:

معماری RISC

RISC processors have a fixed instruction size. In a CISC microcontroller such as the 8051, instructions can be 1, 2, or even 3 bytes. For example, look at the following instructions in the 8051:

```
CLR   C                        ;clear Carry flag, a 1-byte instruction
ADD   Accumulator, #mybyte      ;a 2-byte instruction
LJMP  target_address            ;a 3-byte instruction
```

RISC uses load/store architecture. In CISC microprocessors, data can be manipulated while it is still in memory. For example, in instructions such as "ADD Reg, Memory", the microprocessor must bring the contents of the external memory location into the CPU, add it to the contents of the register, then move the result back to the external memory location. The problem is there might be a delay in accessing the data from external memory.

معماری RISC :
AVR برمبنای معماری RISC هستش

نکته اول: تمام دستورات اندازشون ثابت و مشخص است ینی تقریبا تمام دستورات AVR ما 16 بیتی بود ینی اپکد 16 بیتی داشت این برخلاف میکروکنترل های CISC هستش ینی دستوراتی با عرض متفاوت داره مثلا 8051 می تونه دستورات 1 یا 2 یا 3 بایت باشه پس طول متغییری دارن و میتونه کار رو بعضا متفاوت جلو بده

نکته دوم: معمولا RISC از ساختار load/store استفاده میکنه ینی اگر بخوایم یه کاری رو انجام بدیم به رجیسترهای همه منظوره منتقل میکنیم و اون عملیات بر روی اونها انجام میشه ولی در یکروپروسسورها و به صورت کلی طراحی هایی که براساس CISC انجام میشه این اتفاق می تونه بین رجیستر و مموری هم انجام بشه و طبیعتا این یه مقداری فرایندهای پیچیده تری داره

پایان

موفق و پیروز باشید