GNU -> GCC (compilador)

as ensamblador -> codi maquina mes entendible -> Add 16, 17

codi màquina -> son "xorros" de 16 bits

per fer servir l'assemblador -> codi: avr-gcc

correu -> sd.aguila@upc.edu

horari atenció -> dimarts 12 - 13:30
dimecres 12 - 13:30

35% examen final
15% parcial
50% laboratori

si va millor el final -> 50%

de la bibliografia -> important el primer collas!

interpretii donem el codi el tradueix linia a linia sins poderlo executar mai mes sencer; el tradueix tot de cop a un de sa ix

compilador nivell

tambe el passa a cada nivell

en AVR es interpretació, es va linia per linia de la ROM.

Sistema operatiu:
(intermediari entre programes i hardware. És una màquina virtual.
tradueix i ofereix algo que entengui la màquina!
drivers hardware: acaba decidir què és el que s'ha de fer)

avr-gcc → macroeina, compilador de C aquí el preparem per
poder executar

format elf. genèric i executable

Es fa compilació creuada → el pc compila un
programa per l'arduino

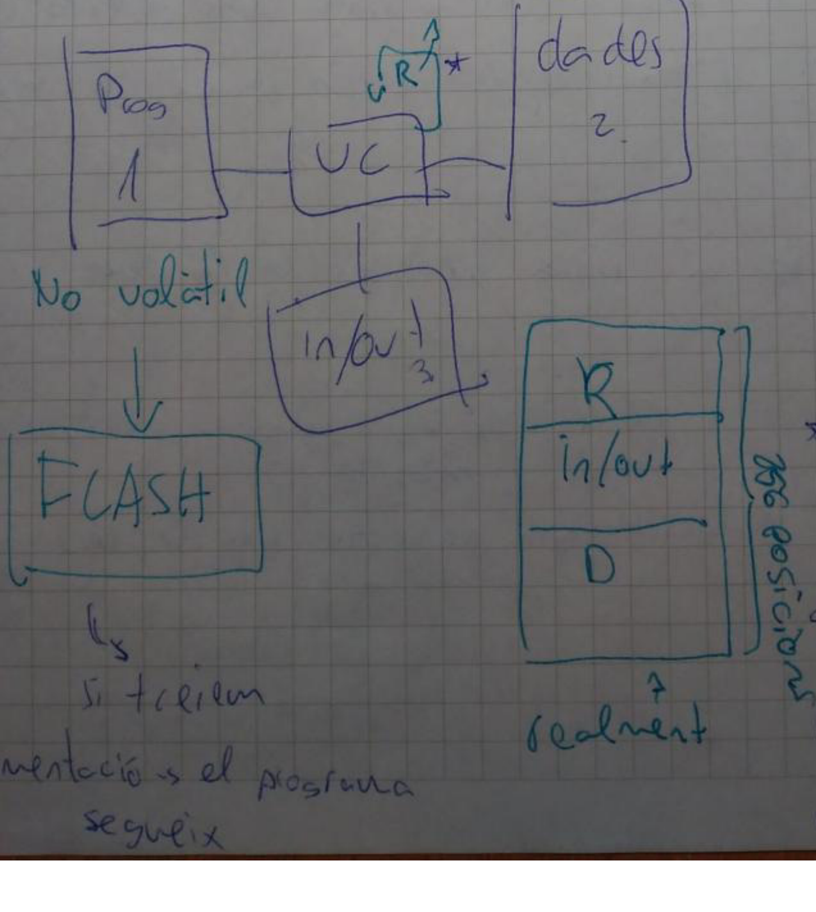
El format elf no van be per l'arduino

hem d'utilitzar avr-objcopy o avr-objdump

Main → adreça d'on està la nop!

main:
↓
nop

Memòries 32 registres



1. per accedir-hi → jump

2. per accedir → ld, st, etc.

3 in/out

* 2 privilegiats, només es poden fer operacions aritmètiques amb aquest.

posició 15 de memòria de dades → és r15

un dels bits de la adreça 5 de la memoria in/out, està connectat al LED. Només un bit, no tot el byte.

7 6 5 4 3 2 1 0

00 10 0000 → 0x20

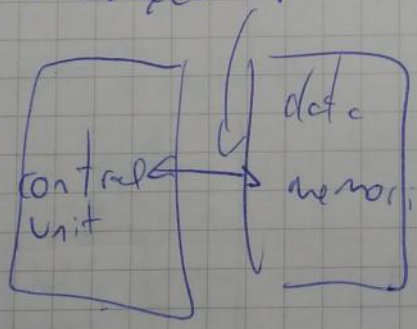
~~0000 1000~~

En un computador → bits ↔ dades
↳ programa per les dades

CPU → central processing unit

AVR → arquitectura harvard

hi ha bus d'adreces (per llegir memoria) i bus de dades



amb la harvard pot accedir a les dues memòries alhora

RAM ordinador → 4, 8, 16 GB

Un PC de 32 bits → 4 GB (2^{32}) = 4 GB

si hi poses més memòria, et saltaran bits

si té més de 4 GB, segur que haurà de tenir 64 bits

Eeprom (?) memòria entrada sortida

Atmel capitals 7 i 8

STATUS REGISTER

I -> interrupcions

T -> És un comodí -> podem ser-lo com vulguem, per exemple per guardar un bit en un moment concret

H -> Half carry

S -> Sign bit s signe

V -> complement a dos

N -> negatiu -> és el bit de més pes a la sortida de ALU

Z i C

Registres X, Y, Z

assumptar r27:r26 → 29,28 → 31,30

STACK POINTER

Registre que guarda ~~adreces~~ És per ser una pila una adreça de memòria

té una llargada de 16 bits, ja que guarda una adreça

$1\,1\,1\,1\,1\,1_0 \rightarrow 2^6 - 1$
$2^5\,2^4\,2^3\,2^2\,2^1\,2$

Memòria 0000 FFFF = 2^{14} = 16K
de programa

16000 posicions de 2 bytes → 32K bytes!
en memòries de words

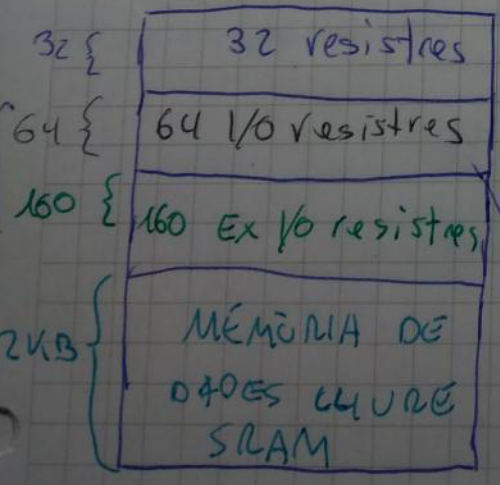
part final de program memory hi ha el boot flash section → és un programa que sempre hi és i s'executa. El que fa és mirar si en el moment de fer un reset hi ha alguna entrada USB. És un mecanisme d'arrancada

sist d'arrancada d'un PC → carrega el sistema operatiu del disc dur a la RAM

és com un programador de la secció, si connectem USB i clicken reset → ens permet reprogramar dins la memòria

MEMÒRIA DE DADES

és de 2K Bytes tot i que l'arquitectura permet 64K!



no són accessibles des de la memòria entrada/sortida, sino via memòria de dades

són 160 resistres extra

via in/out, lds/sts només via lds/sts

10001111111 = 2303

MEMÒRIA ENTRADA/SORTIDA

Register summary → resum de resistres

Memòria in/out posició 0
" data mem posició 32

Nom
in/out -> 5 -> PORTB → PORTB 5 -> LED
in/out -> 4 -> DDRB

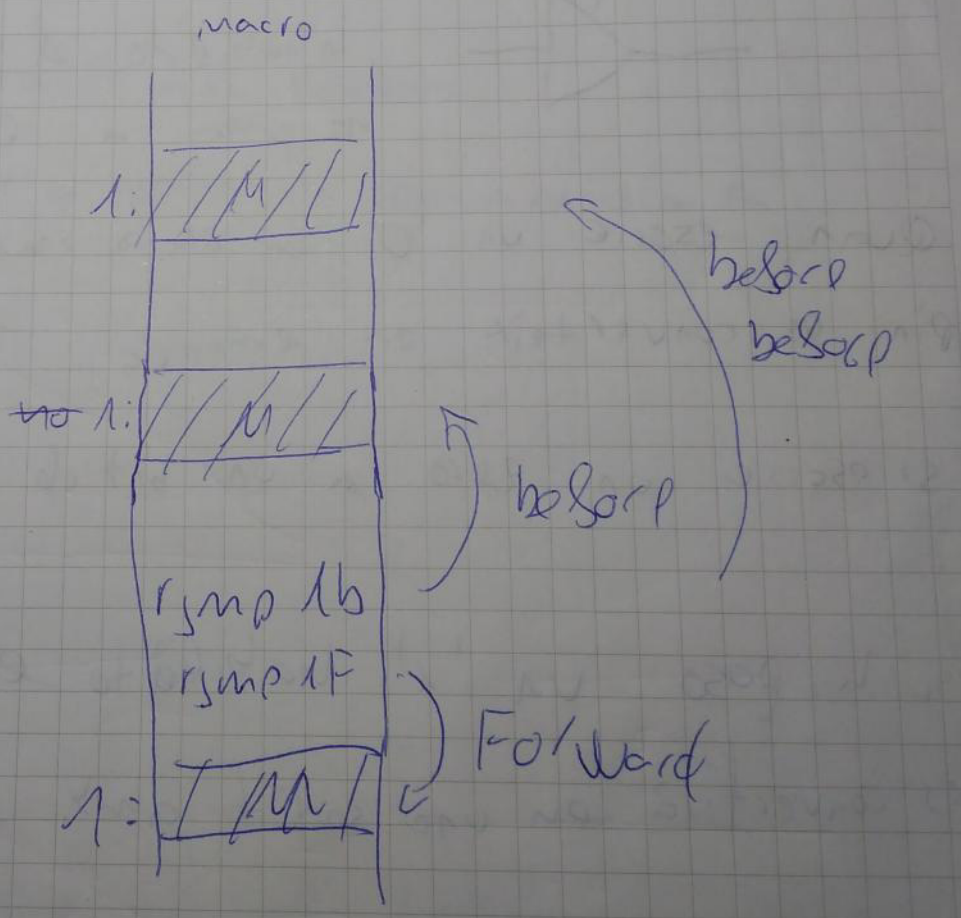
Hi ha resistres per accedir a la EEPROM

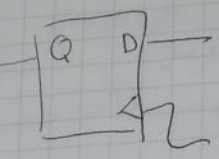
2 bytes per accedir a eeprom → com a màxim la eeprom tindrà 2^{16}.

LABORATORI PRÀCTICA 2- Prou 3

macro: Més línies de programa en total
Assemblador copia i enganya la macro allè on es crida

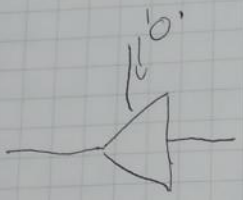
Subrutina Més temps total en executar-ho tot
cau quan es fiaba la crida, sen va al tios de codi i
llavors torna



DD → 

pot escriure en el DD → o '0' o '1'

Fem escric '0' → Q=D='0'



inhabilito el bosser, la entrada no passa a la sortida

Quan escric un '0' a DD → immediatament el pin es converteix en entrada

si escric un zero a un bit de DD → es converteix en entrada

si li poso un '1' habilito el bosser: es convertirà en una sortida amb el valor del PORT!

Per activar pull up → cal que sigui entrada, el valor del PORT sigui 1, i que el PUD sigui 0.

(PUD) - pull up disable → si li poso un 1 → deshabilito el pull up

el bit 4 del registre de dades nº x55 (0x37 dels registres i/o)

PORT → és només per posar sortides
PIN → és per les entrades (si ho configuro com a sortida, amb el resistor pin podria

Mirar la sortida → És un valor elèctric que hi ha al pin

TEORIA

subrutina → ho executa la cpu d'aquesta manera
Macro → ho transforma l'assemblador

com que ho fa l'assemblador, ei indiquem així:

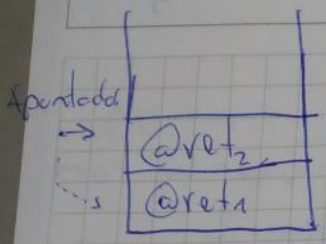
```
.macro waitabit tot
:
.endm
```

Perquè un tros de codi sigui subrutina, cal que es cridi a través d'un rcall, ecall, etc.
La CPU interpreta el codi màquina

La instrucció call ocupa 2 words

quan fem xcall, es guarda la posició on estem +2 i es guarda a dins de la stack

Quan fas un ret, el PC mira on la última posició de la pila, no és que es borri la última posició, sinó que l'apuntador de la pila baixa una posició, llavors perd significat.



si s'omplen totes les posicions hi haurà → stack overflow.

on esta la pila?

Memòria de dades

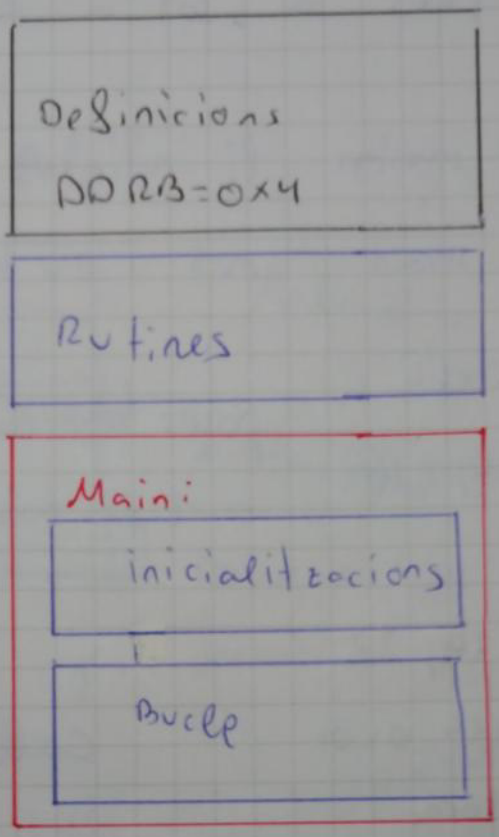
* Stack pointer (apuntador) comensa des de baix de tot a la memòria i va pujant, podria ser tota la memòria de dades

* l'stack pointer està a dins de la memòria de dades també

L'stack pointer, ha salts de dos en dos perquè la memòria és de 8 bits, i l'adreça és de 16, ocupa dos posicions.

una de les inicialitzacions dels start files, és comensar l'SP a on toca

ESTRUCTURA DE PROGRAMA



A definicions puc posar operacions amb constants
↳ ex. 4+3 → el càlcul el farà l'assemblador

Si vull sumar amb la CPU → Add

pas 80 → esquema ports
614 → registres

DDRB → primer el poses a .set a dalt del programa
C
D

llavors al main li indiques quins bits del registre seran inputs i outputs:

ldi r16, 0xFF
out DDRB-0, r16

poses tots els bits del registre a 1 → tots outputs

~~test~~ delay: LDI r19, 23
torna: SUBI r19, 0x01
BRNE torna
RET

{
call delay
{

subrutina → delay

Registre d'estat → on hi han els flags

Aquest codi modifica el resistre r19 posant-lo a 0, independentment del valor que hi havia a r19

si després no es modifica el flag z, es mantindrà a 1

Per ser un comptador → he modificat r19 i resistre d'estat.

A la pila hi guardaré els resistres que no vull que es vegin alterats

Push → posa a l'últim nivell Push Rr

Pop → treu de l'últim nivell Pop Rd