Lab 4: main goals

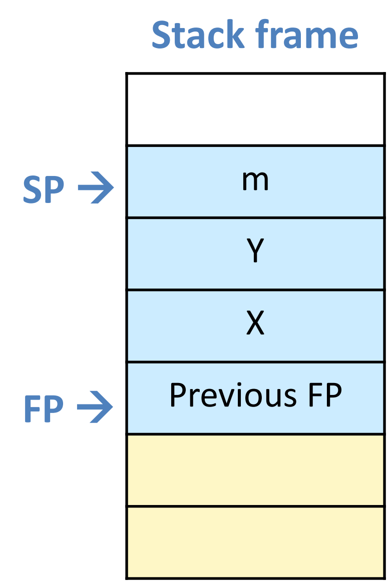
- Analyze the problems that appear when using several source files within a project, and understand how to resolve the symbols.
  - Some files written in C, some others in assembly
- Understand the differences between local and global variables.
- Understand the relation between our own written C program and the machine code generated by a *gcc* compiler.
- Being able to use both variables and functions defined in C from an assembly code, and vice versa.
- Have an initial contact with the representation of structured data types in high level languages.

Translating C functions into assembly

Using local variables in assembly (ref. page 3 of the Lab script)

```c
int Highest(int X, int Y){
    int m;
    if(X>Y)
    m = X;
    else
    m = Y;
    return m;
  }
```

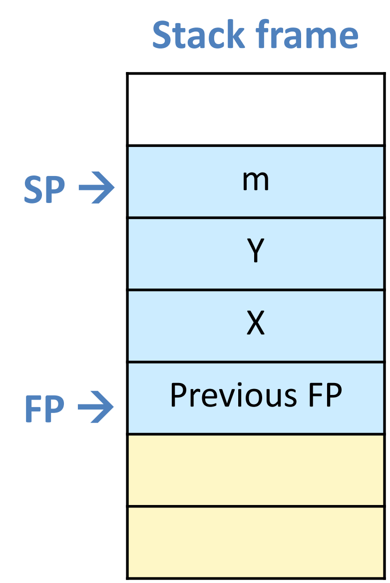
```
Highest:
  push {fp}
  mov fp, sp
  sub sp, sp, #12  @ reserve space for X, Y and m
// X is stored in fp-4, Y in fp-8 and m in fp-12
  str r0, [fp,#-4] @ initialize X with the first parameter
  str r1, [fp,#-8] @ initialize Y with the second parameter

  @ if( X > Y )
  ldr r0, [fp,#-4] @ r0 ← X
  ldr r1, [fp,#-8] @ r1 ← Y
  cmp r0, r1
  ble ELS
  @ then
  ldr r0, [fp,#-4] @ m = X;
  str r0, [fp,#-12]
  b RET
  @ else
 ELS:
  ldr r0, [fp,#-8] @ m = Y;
  str r0, [fp,#-12]
  @ return m
 RET:
  ldr r0, [fp,#-12] @ return value
  mov sp, fp
  pop {fp}
  mov pc, lr
```



Translating C functions into assembly

Using local variables in assembly (ref. page 3 of the Lab script)

```
// A more readable version of the same function
.equ X, -4
.equ Y, -8
.equ m, -12
Highest:
  push {fp}
  mov fp, sp
  sub sp, sp, #12
  str r0, [fp,#X] @ initialize X with the first parameter
  str r1, [fp,#Y] @ initialize Y with the second parameter

  @ if( X > Y )
  ldr r0, [fp,#X] @ r0 ← X
  ldr r1, [fp,#Y] @ r1 ← Y
  cmp r0, r1
  ble ELS
  @ then
  ldr r0, [fp,#X] @ m = X;
  str r0, [fp,#m]
  b RET
  @ else
ELS:
  ldr r0, [fp,#Y] @ m = Y;
  str r0, [fp,#m]
  @ return m
RET:
  ldr r0, [fp,#m] @ return value
  mov sp, fp
  pop {fp}
  mov pc, lr
```



Compiler optimizations

Compiling the same code with different optimization levels (ref. page 5 of the Lab script)

```c
#define N 10
int A[N];
int B[N];
int C,D,i;

int main(void) {
  for (i=0;i<N-1;i+=2) {
     A[i]   = B[i] + C;
     A[i+1] = B[i] - D;
  }
  return 0;
}
```

Compilation without optimizations.
gcc –O0

```
main:
    push {fp}
    add fp, sp, #0
    ldr r3, =i
    mov r2, #0
    str r2, [r3]
    b COND
LOOP:
    ldr r3, =i
    ldr r2, [r3]
    ldr r3, =i
    ldr r1, [r3]
    ldr r3, =B
    ldr r1, [r3, r1, lsl #2]
    ldr r3, =C
    ldr r3, [r3]
    add r1, r1, r3
    ldr r3, =A
    str r1, [r3, r2, lsl #2]
    ldr r3, =i
    ldr r3, [r3]
    add r2, r3, #1
    ldr r3, =i
    ldr r1, [r3]
    ldr r3, =B
    ldr r1, [r3, r1, lsl #2]
    ldr r3, =D
    ldr r3, [r3]
    sub r1, r1, r3
    ldr r3, =A
    str r1, [r3, r2, lsl #2]
    ldr r3, =i
    ldr r3, [r3]
```

A[i]= B[i] + C

A[i+1]= B[i] - D

Compilation with optimizations.
gcc –O2

```
main:
    ldr r3, =C
    push {r4, r5, r6}
    ldr ip, =A        @ ip <- A
    ldr r6, [r3]      @ r6 <- C
    ldr r3, =D
    ldr r4, =B
    ldr r5, [r3]      @ r5 <- D
    mov r2, ip        @ r2 <- &A[0]
    mov r3, #0        @ r3 es i*4
LOOP:
    ldr r1, [r4, r3]  @ r1 <- B[i]
    add r0, r6, r1
    str r0, [ip, r3]  @ A[i] <- r0
    add r3, r3, #8    @ 4*i <- 4*(i + 2)
    sub r1, r1, r5
    cmp r3, #40
    str r1, [r2, #4]  @ *(r2+1) = r1
    add r2, r2, #8    @ salta 2 enteros
    bne LOOP
    ldr r3, =i
    mov r2, #10
    str r2, [r3]
    mov r0, #0
    pop {r4, r5, r6}
    bx  lr
```

Observe: variable i is only updated after loop exit

fc²

Accessing byte-sized data

New machine instructions for loading / storing bytes (ref. page 6 of the Lab script)

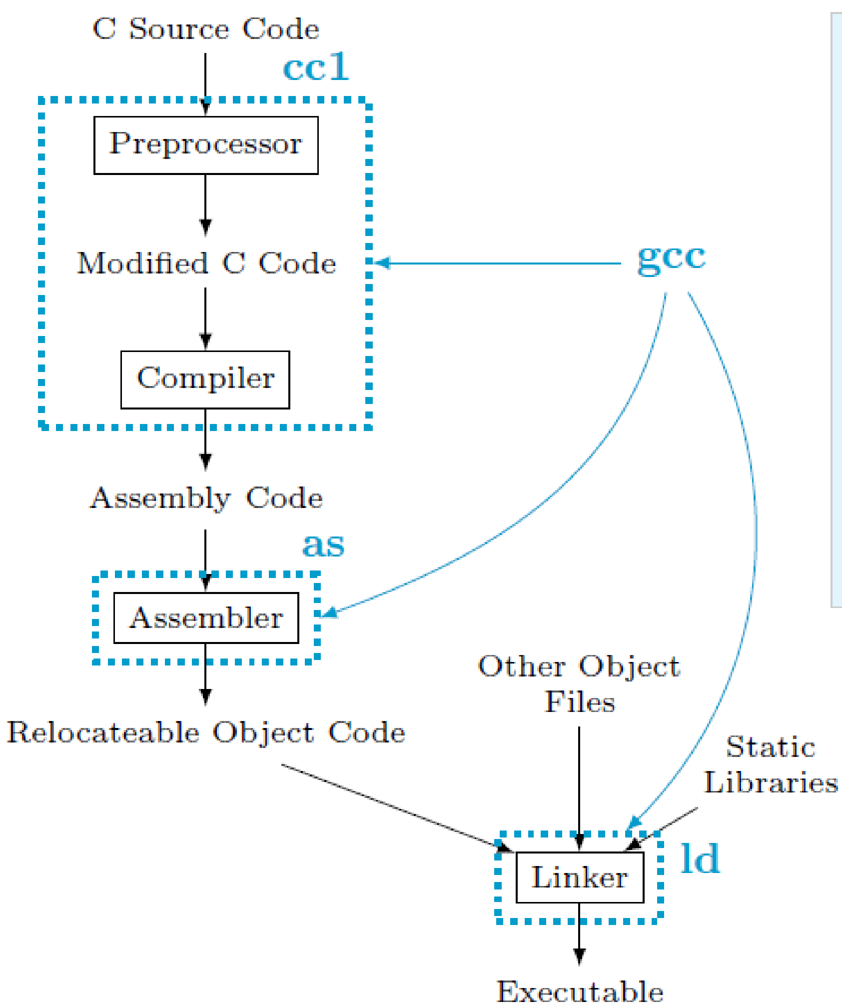
- **LDRB**: load an unsigned integer of size 1 byte. The datum is copied in the **8 least significant bits** of the destination register, and **the rest is filled with 0s**.
- **STRB**: store in memory an integer of size 1 byte, obtained from the **8 least significant bits** of the source register.
- No restrictions on the address: alignment restrictions do not apply.

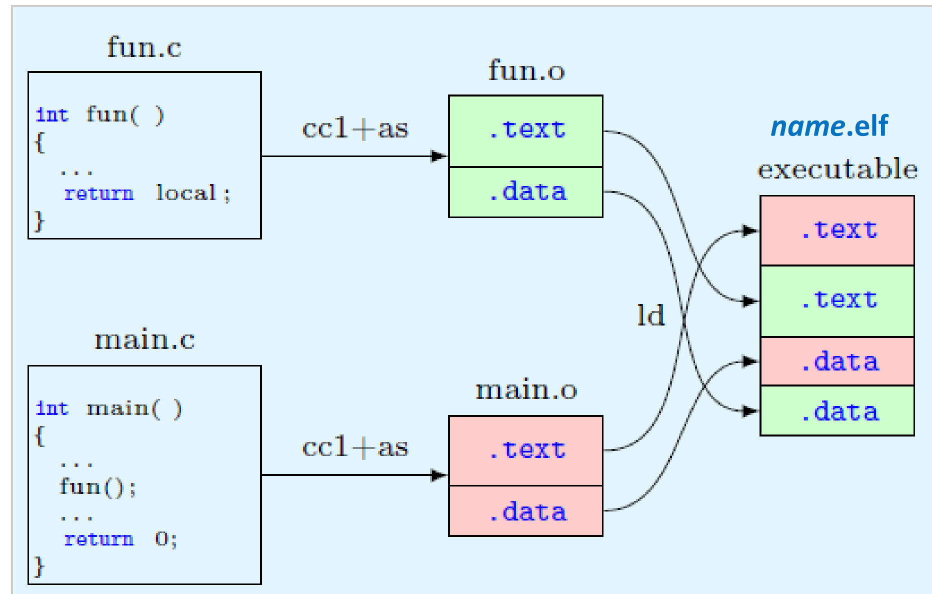
Some examples:

```
LDRB R5, [R9]            @ Loads into the 8 least significant  bits of R5, 8 bits
                         @ from Mem(R9), and resets the remaining bits of R5.
LDRB R3, [R8, #3]        @ Loads into the 8 least significant bits of R3, 8 bits from
                         @ Mem(R8+3), and resets the remaining bits of R3.
STRB R4, [R10, #0x200]   @ Stores into Mem(R10+0x200) the 8 least significant bits of R4.
STRB R10, [R7, R4]       @ Stores into Mem(R7+R4) the 8 least significant bits of R10.
```

Combining several sources

Several C and assembly functions can be merged to create a single executable file (*name*.elf) (ref. slide 72 of module 8)





- What is an "external symbol"?
- How the references to external symbols are resolved?
- What is the symbol table?
- What is the memory map?

Global symbols in C

Using external functions and variables in C (ref. pages 8-9 of the Lab script)

- In C language, all functions and global variables are exported global symbols by default.
- In order to use a function which is defined in **another file**, a forward declaration of the function must be done.
  - For example, for using function FOO, defined in another file, which receives or returns no parameters, we must use the following forward declaration before invoking it:

    ```
    extern void FOO(void);
    ```

    where the **extern** modifier is optional.

- In the case of global variables, a forward declaration must also be done for using a variable defined in another file. For example,

    ```
    extern int aux;
    ```

    If the **extern** modifier is not used, the variable is considered as a COMMON symbol.

fc2

Global symbols in assembly

Declaring and using global symbols in assembly (ref. pages 9-10 of the Lab script)

- Unlike C, in assembly language symbols are local by default, i.e. invisible from another file. For turning them into global symbols we must export them using the **.global** directive.
- Moreover, when the programmer wants to use a symbol defined in another file, the **.extern** directive must be used.
- Example:

```
    .extern FOO;     @ an extern symbol is made visible.
    .global start;   @ a local symbol is exported.

start:
    bl FOO;
    …
```

Launching a C program

In our *bare metal* system (i.e. without any operating system) with need some mechanism to initialize the stack and run the "main" function (ref. page 13 of the Lab script)

Table 2. Example of boot code

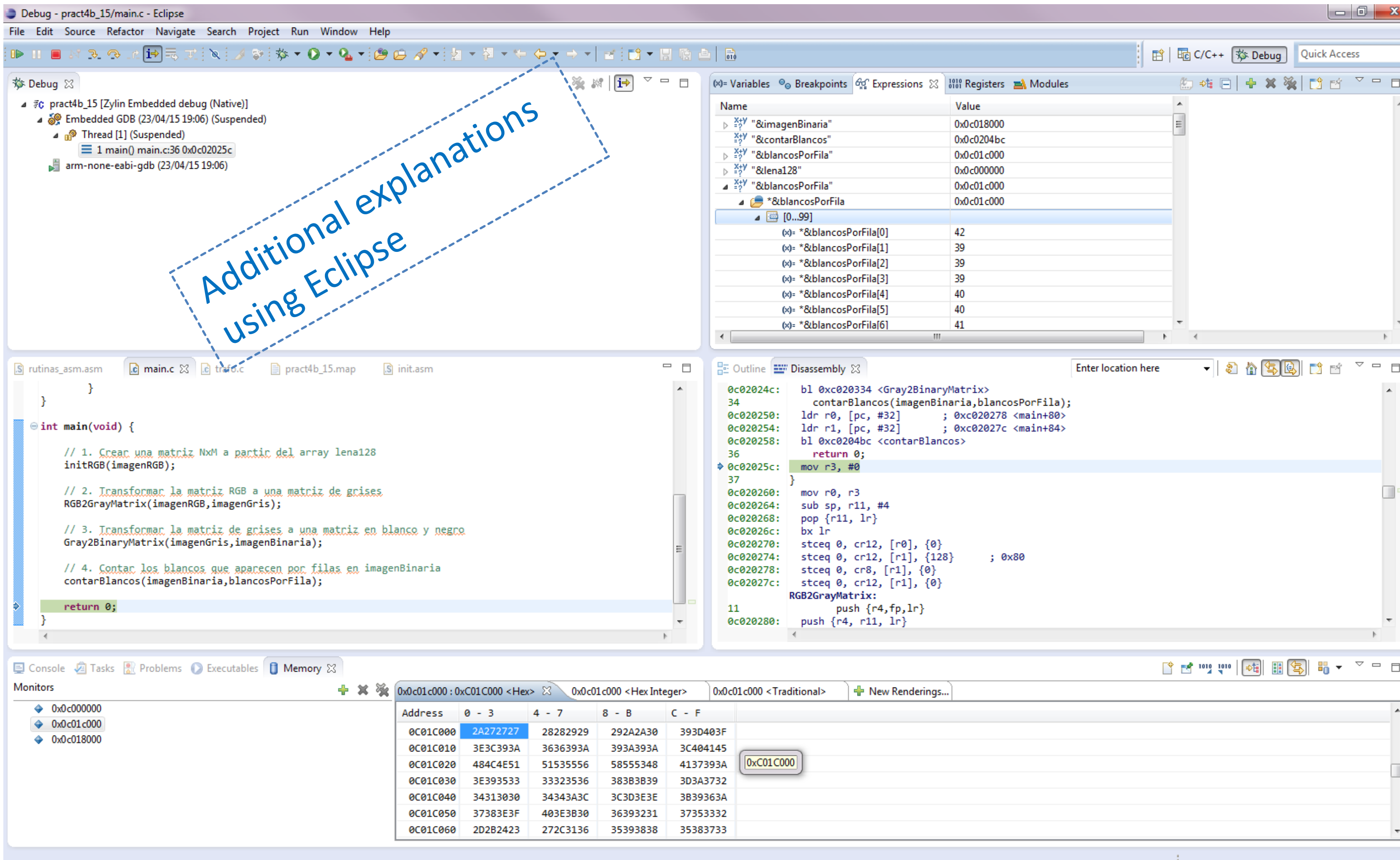
```
    .extern main
    .extern _stack
    .global start

start:
    ldr sp,=_stack
    mov fp,#0

    bl main

End:
    b End
.end
```

File "init.asm"

fc2

Debugging a C program in *Eclipse*

We have additional tools to deal with programs written in C: the Expressions and the Variables windows (ref. page 16 of the Lab script)

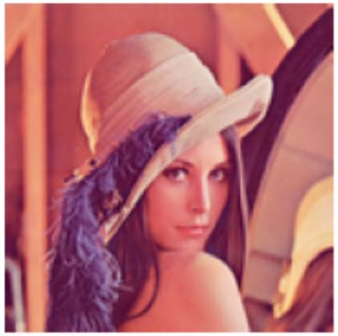


Development of the lab session

Given a color image composed of 128x128 pixels, obtain a grayscale version and a pure Black&White version (ref. page 17 of the Lab script)

Lena128.c
A text file with 3x128x128 elements of unsigned char type.

initRGB



RGBImage
A matrix of 128x128 elements. Each element is a *struct* composed of 3 bytes (R,G,B) to represent a color pixel

RGB2GrayMatrix

grayImage
A matrix of 128x128 elements. Each element is one byte to represent a gray pixel (gray intensity)

gray2BinaryMatrix

binaryImage
A matrix of 128x128 elements. Each element is one byte with the value 0 or 255

Structure of the image in memory

It is declared as an array of pixels. Each pixel is a "struct" composed of three unsigned bytes (ref. page 17 of the Lab script)

```
typedef struct _pixel_RGB_t {
    unsigned char R;
    unsigned char G;
    unsigned char B;
} pixelRGB;
```

```
#define N 128
#define M 128
// defining the color image
pixelRGB RGBImage[N][M];
// defining the gray image
unsigned char grayImage [N][M];
```

Pixels and Colors: Examples

R	G	B
79	129	189
218	102	50
70	246	22
225	137	127

```
RGBImage [0][0]
RGBImage [70][32]
```

R	G	B
139	98	151

Initializing the RGB image

The input data is "lena128 [ ]", which is given as a vector of 3×2^{14} elements.

Each element of lena128 is a byte (**unsigned char** in C)

Every triplet of lena128 represents a pixel of the color image.

```
unsigned char lena128[] = {
225, 137, 127, 226, 135, 123, 227, 136, 123, 223, 134, 116,...}
```

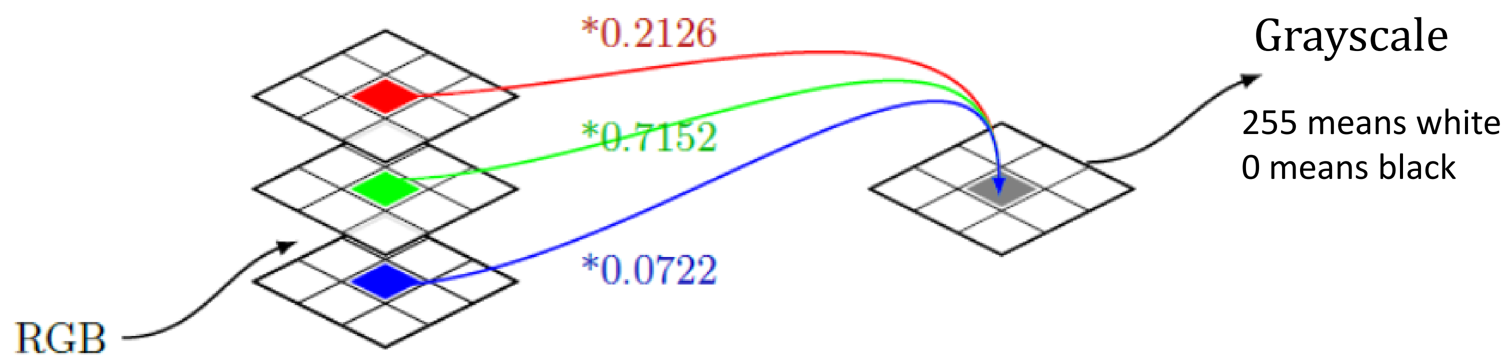
The first 128 triplets represent the first row of RGBImage, the next 128 the second, and so on…

```
// A C function to initialize the RGBImage
void initRGB(pixelRGB m[N][M]) {
    int i,j;

    for (i=0;i<N;i++)
        for (j=0; j<M; j++) {
            m[i][j].R = lena128[(i*M + j)*3];
            m[i][j].G = lena128[(i*M + j)*3 + 1];
            m[i][j].B = lena128[(i*M + j)*3 + 2];
        }
}
```

Transforming RGB into grayscale

The gray value for a given color pixel can be computed as (ref. page 18 of the Lab script):

```
gray = 0.2126*pixel.R + 0.7152*pixel.G + 0.0722*pixel.B;
```



- However, this involves floating point arithmetic. To avoid this we can multiply the constants by a large enough number, HN, and finally divide the result by HN.
- Additionally, if we choose HN as power of two, the division can be made just by shifting. Therefore, we implement the following computation:

```
gray = (3483*orig.R + 11718*orig.G + 1183*orig.B) / 16384;
```

Transforming into a binary image

We define an intermediate number (threshold) between 0 and 255 (ref. page 19 of the Lab script)

Gray shades above the threshold are transformed into “white”. The ones below (or equal to) the threshold are transformed into “black”

We have defined the threshold as 127

```
for (i=0;i<N;i++)
    for (j=0;j<M;j++)
        if (grayImage[i][j] > threshold)
            binaryImage[i][j]= 255;
        else
            binaryImage[i][j]= 0;
```

Structure of the main program

(ref. page 19 of the Lab script)

```c
int main(void) {
    // 1. Create an NxM matrix from the array lena128
    initRGB(RGBImage);

    // 2. Transform the RGB matrix into a grayscale matrix
    RGB2GrayMatrix(RGBImage,grayImage);

    // 3. Transform the grayscale matrix into B&W
    gray2BinaryMatrix(grayImage,binaryImage);

    // 4. Count the white pixels in each row of binaryImage
    countWhites(binaryImage,whitesPerRow);

    return 0;
}
```

Memory allocation of data and code

We can always find the addresses that the linker has assigned to each function and data structure just by analyzing the contents of the file <project_name>.map, which is in the "Debug" folder.

Prac4a.map

```
Allocating common symbols

Common symbol       size              file

binaryImage         0x4000            ./main.o
grayImage           0x4000            ./main.o
RGBImage            0xc000            ./main.o
whitesPerRow        0x80              ./main.o
```

fc²

Memory allocation of data and code

(Cont'd)

```
 .data               0x0c000000     0xc000

                    Start Addr.       size
  *(.data)
  .data              0x0c000000         0x0 ./init.o
  .data              0x0c000000     0xc000 ./lena128.o
                     0x0c000000                  lena128
  .data              0x0c00c000         0x0 ./main.o
  .data              0x0c00c000         0x0 ./routines_asm.o
  .data              0x0c00c000         0x0 ./trafo.o
 ......
 ......


*(COMMON)
 COMMON             0x0c00c000    0x14080 ./main.o
                    0x0c00c000                  binaryImage
                    0x0c010000                  grayImage
                    0x0c014000                  RGBImage
                    0x0c020000                  whitesPerRow
```

Memory allocation of data and code

(Cont'd)

```
                  Start Addr.      size

.text             0x0c020080       0x51c
 *(.text)
 .text            0x0c020080        0x14 ./init.o
                  0x0c020080                start
 .text            0x0c020094         0x0 ./lena128.o
 .text            0x0c020094       0x1ec ./main.o
                  0x0c020094                initRGB
                  0x0c020228                main
 .text            0x0c020280        0x38 ./routines_asm.o
                  0x0c020280                rgb2gray
 .text            0x0c0202b8       0x2e4 ./trafo.o
                  0x0c0202b8                RGB2GrayMatrix
                  0x0c020400                gray2BinaryMatrix
                  0x0c0204e0                countWhites
```