

ARC: Computer Architecture

tanguy.risset@insa-lyon.fr
Lab CITI, INSA de Lyon
Version du March 16, 2023

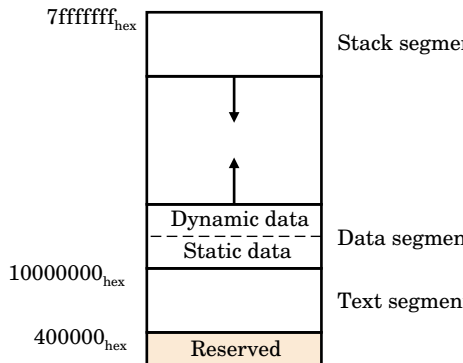
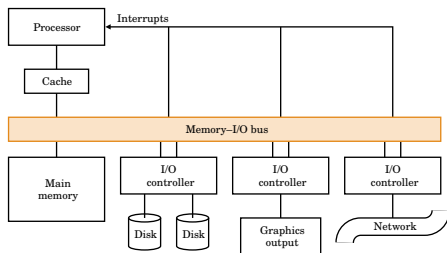
Tanguy Risset

March 16, 2023

Du langage à l'exécution



Rappels d'architecture

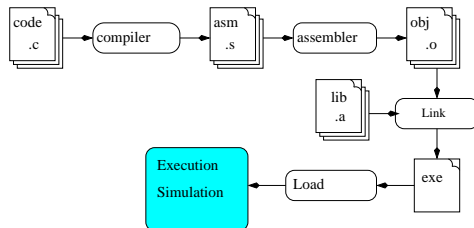


Architecture view from the programmer

- Modern systems allow
 - To run multiple independent programs in parallel (process)
 - To access memory space larger than physical memory available (virtual memory)
- For the programmer: all this is transparent
 - Only one program runs with very large memory available
- The processor view memory contains:
 - The code to execute
 - Static data (size known at compile time)
 - Dynamic data (size known at runtime: the heap, and the space needed for the execution itself: the battery)
- The programmer sees only the data (static and dynamic)

compilation process

- the complete process will translate a C program into code executable (loading and execution will take place later).



- We often call *compilation* the set compiler + assembler
- The gcc compiler also includes an assembler and linking process (accessible by options)

Your compilation process

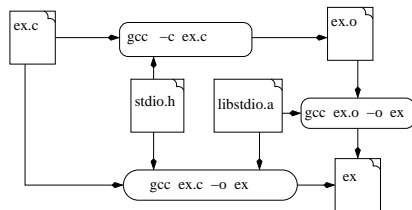
- The programmer:
 - Write a program (say a C program: `ex.c`)
 - Compiles it to an object program `ex.o`
 - links it to obtain an executable `ex`

content of `ex.c`

```
#include <stdio.h>

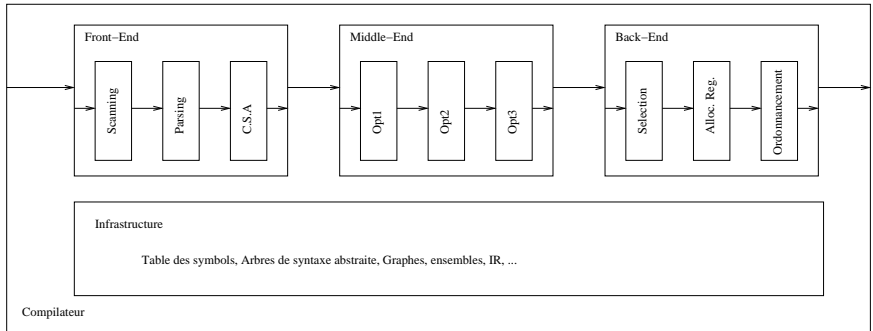
int main()
{
    printf("hello World\n");

    return(0);
}
```



Zooming on “compilation”

- The compilation process is divided in 3 phases:

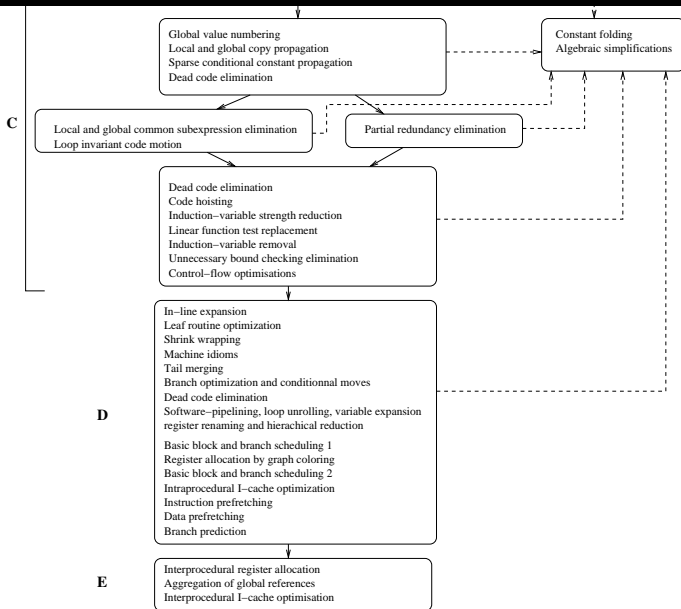


Compilation: the front-end

- The front end of an embedded code compiler uses the same techniques as traditional compilers (we can want to include assembler parts directly)
- Parsing LR(1): the parser is usually generated with dedicated *metacompilation* tools such as Flex et bison for GNU

Compilation: The middle-end

- Some phases of optimizations are added, they can be very calculative
- Some example of optimisation independent of the target machine architecture
 - Elimination of redundant expressions
 - dead code elimination
 - constant propagation
- Warning: optimization can hinder the understanding of the assembler (use the -O0 options with `gcc`)



Compilation: The back-end

- The code generation phase is dedicated to architecture target. Retargetable compilation techniques are used for architectural families.
- The most important steps important are:
 - Code selection
 - Register allocation
 - instruction scheduling

- The gcc command runs several program depending on the options
 - The pre-processor cpp
 - The compiler cc1
 - The assembleur gas
 - The Linker ld
- gcc -v allow to visualize the different programs called by gcc

The pre-processor `cpp` or `gcc -E`

- the task of the pre-processor are :
 - elimination of comments,
 - inclusion of source files
 - macro substitution (`#define`)
 - conditionnal compilation.
- Example:

ex1.c

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))  
...  
f=MAX(3,b);
```

ex1.i

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))  
...  
f=((3) > (b) ? (3) : (b));
```

The compiler cc1 or gcc -S

- generate assembly code
- `gcc -S main.c -o main.S`
- Exemple :

```
void main()
{ int i;
  i=0;

  while (1)
  {
    i++;
    nop();
  }
}
```

Assembly code generated (for MSP430)



```
mov    #2558, SP        ; stack initialization de la pile
mov    r1, r4           ; r4 <- SP
mov    #0, 0(r4)        ; i initialization
inc    0(r4)            ; i++
nop                     ; nop();
jmp    $-6              ; unconditionnal jump (PC-6):
incd   SP               ;
br     #0x1158           ;
```

Assembly code produce by mspgcc -S

```
.text
.p2align 1,0
.global      main
.type        main,@function

main:
/* prologue: frame size = 2 */
.L__FrameSize_main=0x2
.L__FrameOffset_main=0x6
    mov      #(__stack-2), r1
    mov      r1,r4
/* prologue end (size=3) */
    mov      #llo(0), @r4
.L2:
    add      #llo(1), @r4
    nop
    jmp      .L2
/* epilogue: frame size=2 */
    add      #2, r1
    br       #__stop_progExec__
/* epilogue end (size=3) */
/* function main size 14 (8) */
```


Assembler as ou gas

- transform an assembly code into object code (binaire representation of symbolic assembly code)
- Option `-c` of `gcc` allow to combine compilation et assembly
`gcc -c main.c -o main.o`

Linking: ld

- Produce the executable (a.out by default) from object code of programs and library used
- There are two ways to use libraries in a program
 - Dynamic or shared libraries (default option): the code of the library is not included in the executable, the system dynamically loads the code of the library in memory when calling the program. We need than only *one* version of the library in memory even if several programs use the same library. The library must be installed on the machine, before running the code.
 - Static libraries: the code of the library is included in the executable. The executable file is bigger but you can run it on a machine on which the library is not installed.

Binary file manipulation

Some usefull command:

- `nm`

Allow to know symboles (i.e. label: function names) used in an object file or executable

```
trisset@hom\$ nm fib.elf | grep main
000040c8 T main
```

- `objdump` allow to anlayze a binary file. For instance it can get correspondance between binary representation and assembly code

```
trisset@hom$ objdump -f fib
fib:          file format elf32-msp430
architecture: msp:43, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00001100
```