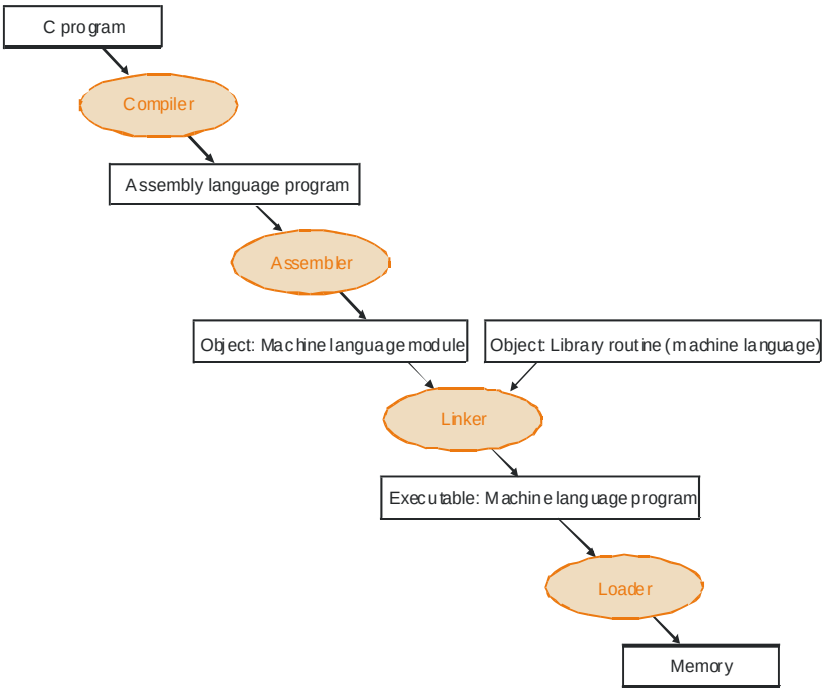
3.1 - Execução de um Programa

Um programa escrito em C deve ser traduzido para um programa executável. Os sistemas empregam quatro passos básicos lógicos para que um programa escrito em uma linguagem de alto nível deve passar até se tornar executável.



Um programa em linguagem de alto nível é **compilado** gerando um programa equivalente em linguagem de **montagem**, depois montado gerando um o **módulo-objeto** em linguagem de máquina.

3.2 - Compilador

Para os programas escritos em C podemos usar o compilador gcc que possui algumas opções de compilação:

-c → Compile or assemble the source files, but do not link. The linkin stage simply is not done. The ultimate output is in the form of a oject file for each source file. By default, the object file name for a source file is made by replacing the suffix .c, .i, .s, etc., with .o.
Esta compilação gera um arquivo objeto **.o**

Exemplo: `gcc -c arquitetura.c`
Arquivo de saída: `arquitetura.o`

-S →**Stop** after the stage of compilation proper; do **not assemble**. The output is in the form of an assembler code file for each non- assembler input file specified.

Exemplo: `gcc -S arquitetura.c`
Arquivo de saída: `arquitetura.s`

```
/* Este programa mostra o código em C*/
#include <stdio.h>
#define max 10
int a[max], i;

main ()
{
        for (i = 0; i <=  max; ++i){
                a[i]=i*2;
                printf("A(%d)=    %d    ",i,
a[i]);
                printf;
        }
}
```

```
/*Arquivo em assembler*/
/*arquitetura.s              */



        .file   "arquitetura.c"
        .def    ___main;        .scl    2;
        .type   32;     .endef
        .text
LC0:
        .ascii "A(%d)= %d \0"
        .align 2
.globl _main
        .def    _main; .scl     2;      .type
        32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $0, _i
L10:
        cmpl    $10, _i
        jle     L13
        jmp     L11
L13:
        movl    _i, %edx
        movl    _i, %eax
        addl    %eax, %eax
        movl    %eax, _a(,%edx,4)
        movl    $LC0, (%esp)
        movl    _i, %eax
        movl    %eax, 4(%esp)
        movl    _i, %eax
        movl    _a(,%eax,4), %eax
        movl    %eax, 8(%esp)
        call    _printf
        incl    _i
        jmp     L10
L11:
        leave
        ret
        .comm _a, 48   # 40
        .comm _i, 16   # 4
        .def    _printf;        .scl    2;
        .type   32;     .endef
```

-E → Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output.

```
$ gcc -E arquitetura.c  |more
# 1 "arquitetura.c"                # 1 "<built-in>"
```

```
# 1 "<command line>"
# 1 "arquitetura.c"

# 1 "/usr/include/stdio.h" 1
3
# 29 "/usr/include/stdio.h"
3
# 1 "/usr/include/_ansi.h" 1
3
# 15 "/usr/include/_ansi.h"
3
# 1 "/usr/include/newlib.h"
1 3
# 16 "/usr/include/_ansi.h"
2 3
# 1
"/usr/include/sys/config.h"
1 3

# 1
"/usr/include/machine/ieeefp
.h" 1 3
# 5
"/usr/include/sys/config.h"
2 3
# 17 "/usr/include/_ansi.h"
2 3
# 30 "/usr/include/stdio.h"
2 3
```

-o `file` → Place output in file. This applies regardless to whatever sort of output is being produced, whether it be an executable file an object file, an assembler file or preprocessed C code.
Exemplo: `gcc –o arquitetura.c`
Arquivo de saída: `a.out`

Since only one output file can be specified, it does not make sens to use -o when compiling more than one input file, unless you are producing an executable file as output.
Exemplo: `gcc arquitetura.c` **`–o`** `arquitetura.e`
Arquivo de saída: `arquitetura.e`

If -o is not specified, the default is to put an executable file i a.out, the object file for source.suffix in source.o, its assemble file in source.s, and all preprocessed C source on standard output.

-v Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.
Exemplo: `gcc –v`
Saída: `version 3.2 20020927 (prerelease)`

Otimizações: -O

-O1 → Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
Exemplo: `gcc –O1 -S arquiteturaopt1.c`
Arquivo de saída: `arquiteturaopt1.s`

Without –O → the compiler's goal is to **reduce the cost of compilation** and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

With –O → the compiler **tries to reduce code size and execution time** without performing any optimizations that take a great deal of compilation time.

Exemplo: `gcc –O -S arquitetura-O.c`
Arquivo de saída: `arquitetura-O.s`

- O2 → Optimize even more. GCC performs nearly all supported optimizations that do **not involve a space-speed tradeoff**. The compiler does **not perform loop unrolling or function inlining** when you specify -O2. As compared to -O, this option **increases both compilation time and the performance** of the generated code.

Exemplo: `gcc –O2 -S arquiteturaopt2.c`
Arquivo de saída: `arquiteturaopt2.s`

- O3 → Optimize yet more. -O3 turns on all optimizations specified by -O and also turns on the -finline-functions and -frename-registers options.

Exemplo: `gcc –O3 -S arquiteturaopt3.c`
Arquivo de saída: `arquiteturaopt3.s`

-O0 → Do not optimize.

Exemplo: `gcc –O0 -S arquiteturaopt0.c`
Arquivo de saída: `arquiteturaopt0.s`

-Os Optimize for size. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size.

Exemplo: `gcc –Os -S arquiteturaopt0s.c`
Arquivo de saída: `arquiteturaopt0s.s`

If you use multiple -O options, with or without level numbers, the last such option is the one that is effective.

Arquivo fonte (189 bytes)	Compilação	Saída	Tamanho (bytes)
arquitetura.c	gcc –c	arquitetura.o	688
arquitetura.c	gcc –S	arquitetura.s	681
arquitetura.c	gcc –o arquitetura.c	a.out	19069
------	gcc –v	version	---
arquiteturaopt1.c	gcc –O1 -S	arquiteturaopt1.s	551

arquiteturaopt2.c	gcc –O2 -S	arquiteturaopt2.s	647
arquiteturaopt3.c	gcc –O3 -S	arquiteturaopt3.s	685
arquiteturaopt0.c	gcc –O0 -S	arquiteturaopt0.s	685
arquiteturaopt0s.c	**arquiteturaopt0s.c**	**arquiteturaopt0s.s**	**516**
arquitetura-O.c	gcc –O -S	arquitetura-O.s	549

3.3 Montador

A função do montador é transformar a linguagem de montagem em código de máquina.
O montador transforma o programa em linguagem de montagem em um arquivo objeto, que é a combinação de instruções em linguagem de máquina, dados e informações para carregar o programa em memória.
São arquivos com extensão: **.s ou .asm**
O arquivo-objeto é composto de seis partes distintas:

1. Cabeçalho: descreve tamanho e posição do restante do arquivo.
2. Segmento de texto: código em linguagem de máquina.
3. Segmento de dados: dados para a execução do programa.
4. Informações sobre relocação: identifica as palavras de instrução e de dados que dependem os endereços absolutos por ocasião da carga do programa na memória.
5. Tabela de símbolos: contém os labels não definidos como as referencias externas.
6. Informações para análise de erros: associa instruções da máquina com os arquivos fonte em C para o depurador (*debbuger*).

```
/* arquitetura.s sem otimizações*/
        .file   "arquitetura.c"
        .def    ___main;        .scl    2;
        .type   32;     .endef
        .text
LC0:
        .ascii "A(%d)= %d \0"
        .align 2
.globl _main
        .def    _main;  .scl    2;      .type
        32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $0, _i
L10:
        cmpl    $10, _i
```

```
        jle     L13
        jmp     L11
L13:
        movl    _i, %edx
        movl    _i, %eax
        addl    %eax, %eax
        movl    %eax, _a(,%edx,4)
        movl    $LC0, (%esp)
        movl    _i, %eax
        movl    %eax, 4(%esp)
        movl    _i, %eax
        movl    _a(,%eax,4), %eax
        movl    %eax, 8(%esp)
        call    _printf
        incl    _i
        jmp     L10
L11:
        leave
        ret
        .comm   _a, 48      # 40
        .comm   _i, 16      # 4
        .def    _printf;  .scl      2;      .type
        32;     .endef
```

```
        pushl   %ebp
        movl    %esp, %ebp
        call    ___main
        movl    $0, _i
L6:
        movl    _i, %eax
        leal    (%eax,%eax), %edx
        pushl   %edx
        pushl   %eax
        pushl   $LC0
        movl    %edx, _a(,%eax,4)
        call    _printf
        movl    _i, %eax
        addl    $12, %esp
        incl    %eax
        movl    %eax, _i
        cmpl    $10, %eax
        jle     L6
        leave
        ret
        .comm   _a, 48      # 40
        .comm   _i, 16      # 4
        .def    _printf;  .scl      2;      .type
        32;     .endef
```

```
/* arquiteturaOs.s com otimização por tamanho*/
        .file   "arquiteturaopt0s.c"
        .def    ___main;        .scl    2;
        .type   32;     .endef
        .text
LC0:
        .ascii "A(%d)= %d \0"
        .align 2
.globl _main
        .def    _main;  .scl    2;      .type
        32;     .endef
_main:
```

```
/* arquitetura.s sem otimizações i686*/
        .file   "arquitetura.c"
        .def    ___main;  .scl      2;      .type
        32;     .endef
        .text
LC0:
        .ascii "A(%d)= %d \0"
        .align 2
.globl _main
        .def    _main;      .scl    2;      .type
        32;     .endef
_main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        movl    %eax, -4(%ebp)
```

```
        movl    -4(%ebp), %eax
        call    __alloca
        call    ___main
        movl    $0, _i
L10:
        cmpl    $10, _i
        jle     L13
        jmp     L11
L13:
        movl    _i, %edx
        movl    _i, %eax
        addl    %eax, %eax
        movl    %eax, _a(,%edx,4)
        movl    $LC0, (%esp)
        movl    _i, %eax
        movl    %eax, 4(%esp)
        movl    _i, %eax
        movl    _a(,%eax,4), %eax
```

```
        movl    %eax, 8(%esp)
        call    _printf
        incl    _i
        jmp     L10
L11:
        leave
        ret
        .comm   _a, 48      # 40
        .comm   _i, 16      # 4
        .def    _printf;    .scl    2;      .type
        32;     .endef
```

```
/* arquitetura.s sem otimizações SUN  */
        .file   "arquitetura.c"
        .section    ".rodata"
        .align 8
.LLC0:
        .asciz  "A(%d)= %d "
        .section    ".text"
        .align 4
        .global main
        .type   main, #function
        .proc   04
main:
        !#PROLOGUE# 0
        save    %sp, -112, %sp
        !#PROLOGUE# 1
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %g1
        st      %g0, [%g1]
```

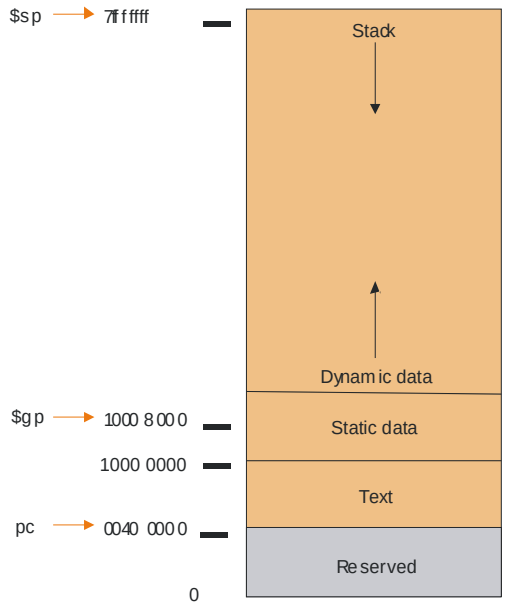
```
.LL2:
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %g1
        ld      [%g1], %g1
        cmp     %g1, 10
        ble     .LL5
        nop
        b       .LL3
         nop
.LL5:
        sethi   %hi(a), %g1
        or      %g1, %lo(a), %o4
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %g1
        ld      [%g1], %g1
        sll     %g1, 2, %o5
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %g1
        ld      [%g1], %g1
        add     %g1, %g1, %g1
        st      %g1, [%o4+%o5]
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %o4
        sethi   %hi(a), %g1
        or      %g1, %lo(a), %o3
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %g1
        ld      [%g1], %g1
        sll     %g1, 2, %o5
        sethi   %hi(.LLC0), %g1
        or      %g1, %lo(.LLC0), %o0
        ld      [%o4], %o1
        ld      [%o3+%o5], %o2
        call    printf, 0
         nop
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %o5
        sethi   %hi(i), %g1
        or      %g1, %lo(i), %g1
        ld      [%g1], %g1
        add     %g1, 1, %g1
        st      %g1, [%o5]
        b       .LL2
         nop
.LL3:
        mov     %g1, %i0
        ret
        restore
        .size   main, .-main
        .common a,40,4      .common i,4,4
        .ident  "GCC: (GNU) 3.3.2"
```

Ligador (Linker):

Tem a função de combinar os diversos módulos com as rotinas e bibliotecas resolvendo todas as referências entre eles.
Três passos compõem o trabalho do ligador:

1. Colocar os módulos de código e dados simbolicamente na memória.
2. Determinar os endereços dos labels de dados e instruções.
3. Resolver as referências internas e externas.

O ligador usa as informações de relocação e a tabela de símbolos de cada modulo-objeto para resolver todos os labels indefinidos.
Essas referências ocorrem em instruções de desvios condicional, incondicional e em endereços de dados. Ele encontra os endereços antigos e os substitui por novos, definindo quais os endereços de memória que cada um vai usar.



3.4 Carregador:

Aqui já temos o arquivo executável em disco, o SO deve preparar a máquina para executar o programa.
O carregador deve transferir o arquivo para a memória, no Unix isto é feito em seis passos:

1. Leitura do cabeçalho do arquivo executável: tamanho do texto e dados.
2. Criação de espaço para armazenar o código e dados.
3. Copiar os dados e as instruções para a memória.
4. Copiar os parâmetros para a pilha do programa principal.
5. Inicializar os registradores da máquina e posicionar o SP=primeiro endereço livre da memória.
6. Desviar para uma rotina de inicialização que copia os parâmetros nos registradores de argumento e chama a rotina principal do programa. Quando a rotina termina, a rotina de inicialização termina o programa executando uma chamada ao sistema do Unix → exit.

3.5 Processadores RISC e CISC

Durante grande parte da história dos computadores pessoais, os modelos predominantes de microprocessadores têm sido da Intel Corporation. O primeiro processador IBM PC foi o Intel 8086. As gerações de processadores Intel que o seguiram forma da família '86.

Todos eram versões mais elaboradas do 8086 original, mas com desempenho melhorado por uma de duas maneiras - operando mais rapidamente ou tratando mais dados simultaneamente. O 8088, por exemplo, operava a 4,7 milhões de oscilações por segundo - e alguns chips Pentium chegam a até 500 MHz. Já o 8088 podia tratar 8 bits de dados por vez, enquanto o 80486 trata 32 bits internamente. Mas apesar das alterações, os processadores Intel até o 80486 eram baseados em uma filosofia de projeto denominada

CISC, do inglês “complex instruction set computing”, ou computação por conjunto complexo de instruções.

O projeto **CISC** emprega comandos que **incorporam muitas pequenas instruções** para realizar uma única operação. É um facão no sentido de retalhar e fatiar dados e código. Uma alternativa de projeto, por comparação, funciona mais como um bisturi, cortando pedaços menores e mais delicados de dados e código.

O bisturi chama-se “reduced instructions set computing”, **RISC**, ou computação por **conjunto reduzido de instruções**. Projetos RISC são encontrados em processadores novos como o Alpha da DEC, RISC 6000 da IBM, o processador PowerPC e, até certo grau, os processadores Pentium da Intel. O RISC é um projeto menos complicado que utiliza diversas instruções simples para executar em menos tempo uma operação comparável à de um único processador CISC executando um grande e complicado comando.

Os chips RISC podem ser fisicamente menores que os chips CISC. E como possuem menos transistores, são geralmente mais baratos de produzir e menos propensos a sobreaquecimento. Tem havido muitas previsões de que o futuro dos processadores é um projeto RISC e isto está provavelmente correto. Mas não tem havido um movimento para grandes vendas de RISC devido a duas razões. A mais importante é manter a compatibilidade com o vasto número de programas aplicativos que tem sido escritos para funcionar com os processadores CISC anteriores da Intel. A segunda razão é que não se obtém o benefício completo da arquitetura RISC a menos que se esteja usando um sistema operacional e programas que tenham sido escritos e compilados especificamente para conseguir o melhor das operações RISC.

Alguns fabricantes de computadores estão oferecendo processadores RISC como forma de projetarem-se na tecnologia de ponta. Executam antigos programas CISC somente através da emulação de processadores CISC, o que anula as vantagens do RISC. Assim, a maioria dos fabricantes de PC têm permanecido com o projeto que é onde os investimentos estão. Ao mesmo tempo, criadores de software estão relutantes em converter seus programas para versões compiladas para RISC já que não há tantas pessoas que possuem PCs baseados em RISC. O mais provável é que os processadores continuem pela via evolucionária mais segura que a Intel está trilhando.

Eventualmente, vamos acabar usando arquitetura RISC, mas a maioria dos usuários não saberá quando seus computadores cruzarem a linha divisória entre os dois projetos.

Os novos processadores Pentium empregam uma arquitetura que mistura características das arquiteturas CISC e RISC, sendo muito difícil dizer qual das arquiteturas será a do futuro.

RISC	CISC
Instruções simples e rápidas (poucos ciclos de clock)	Instruções complexas e demoradas (muitos ciclos de clock)
Instruções executadas pelo hardware	Instruções executadas em microprograma
Poucas instruções	Muitas instruções
Compiladores complexos	Compiladores simples