

Atividade de Laboratório 1

[Objetivos](#)

[Instalação dos Requisitos](#)

[Instalando em uma distribuição GNU/Linux](#)

[Instalando no Windows com WSL2](#)

[Utilizando o Repl.it](#)

[O processo de compilação](#)

[Exercício](#)

[Tarefa 1](#)

[Montagem](#)

[Simulação](#)

[Depuração](#)

[GNU makefile](#)

[Tarefa 2](#)

Objetivo

O objetivo desta atividade é a familiarização com as ferramentas e o ambiente de trabalho da disciplina.

OBS: Nesta disciplina, utilizaremos ferramentas de *software* (p.ex., o compilador CLANG) que estão instaladas nos computadores do IC-3. Caso você tenha interesse em instalar em seu computador, veja instruções no final deste documento, na seção de informações extras.

O processo de compilação

O processo de compilação de programa em linguagem C envolve 3 etapas principais:

- 1) **Compilação:** Cada arquivo de linguagem C é traduzido para código de linguagem de montagem (arquivos com a extensão .s).
- 2) **Montagem:** O montador (ou *assembler*) lê os arquivos em linguagem de montagem e produz um código-objeto (extensão .o no Linux). Note que um *software* complexo pode conter diversos arquivos de código fonte, o que irá levar a vários arquivos-objeto durante o processo de compilação. Assim, apesar de possuir código em linguagem de máquina, os arquivos-objeto não são executáveis, pois o código binário ainda está separado em diversos

arquivos-objeto e precisa ser "ligado" em um único arquivo, que contenha todo o código.

- 3) **Ligação:** O ligador lê diversos arquivos-objeto como entrada, os liga entre si, e também liga código de bibliotecas. O resultado é o executável final, o programa que pode ser executado pelo usuário.

A Figura 1 ilustra o processo de compilação de um *software* com dois arquivos fonte: `fonte1.c` e `fonte2.c`. Neste diagrama, o comando `gcc -S` invoca o compilador, o comando `as` invoca o montador e, por fim, o comando `ld` invoca o ligador.

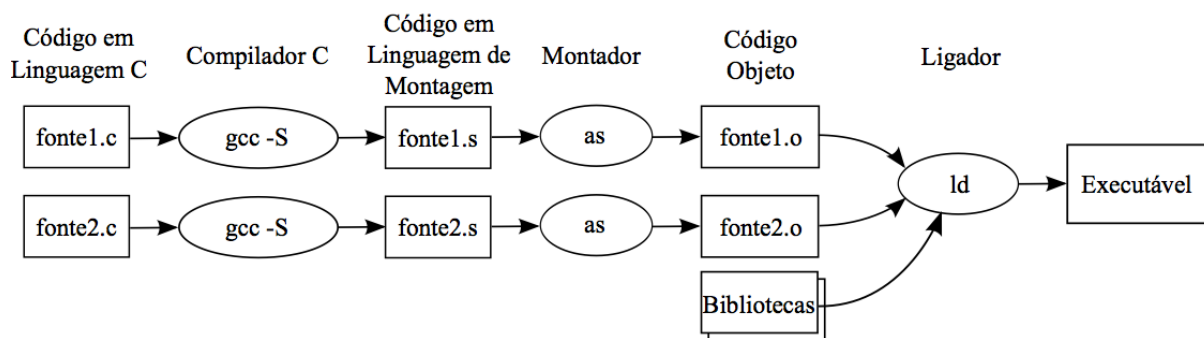


Figura 1: O processo de compilação de um programa utilizando as ferramentas da GNU.

Exercício

Você deverá percorrer o processo de compilação executando cada programa individualmente (compilador C, montador e ligador) até chegar no executável final do programa. Você deverá compilar o código fonte de um programa que está organizado em dois arquivos C, chamados de `arquivo1.c` e `arquivo2.c` com o seguinte conteúdo:

`arquivo1.c`

```
extern void write(int __fd, const void *__buf, int __n);
int main(void) {
    const char str[] = "Hello World!\n";
    write(1, str, 13);
    return 0;
}

void _start(){
    main();
}
```

arquivo2.c

```
void write(int __fd, const void *__buf, int __n){
    __asm__ __volatile__(
        "mv a0, %0          # file descriptor\n"
        "mv a1, %1          # buffer \n"
        "mv a2, %2          # size \n"
        "li a7, 64          # syscall write (64) \n"
        "ecall"
        : // Output list
        : "r"(__fd), "r"(__buf), "r"(__n) // Input list
        : "a0", "a1", "a2", "a7"
    );
}
```

Para realizar esta atividade, você deve, primeiramente, criar estes dois arquivos (com seus respectivos conteúdos) em uma pasta (diretório) em seu computador.

Você deverá fazer o processo de compilação uma vez manualmente (Tarefa 1), depois automatizar o processo com um *script* makefile (Tarefa 2). Para isso, você deve criar uma regra para cada arquivo intermediário, até chegar no arquivo final.

Tarefa 1

Além de percorrer todas as etapas de compilação manualmente, queremos também que nosso código seja executado em um simulador RISC-V. Como estamos usando computadores da família x86, vamos utilizar um ambiente de compilação cruzada (cross compiling), de modo que usaremos um montador e um linker que funcionam nas famílias x86, mas que geram código para a arquitetura RISC-V.

A seguir, explicaremos como executar cada etapa do processo de compilação de forma genérica. Note que, para esta tarefa: Você deverá **compilar** e **montar** cada arquivo C **separadamente**, e **ligar** os objetos gerados no final com o ligador.

Compilação

Para compilar um código-fonte C e produzir o código em linguagem de montagem .s do RISC-V, você deve executar o seguinte comando:

```
clang-12 --target=riscv32 -march=rv32g -mabi=ilp32d
-mno-relax arquivo_de_entrada.c -S -o arquivo_de_saida.s
```

Note o parâmetro na ferramenta clang: `--target=riscv32 -march=rv32g -mabi=ilp32d`. Esse parâmetro indica que queremos que seja gerado um objeto que não é da arquitetura do seu computador (provavelmente x86_64) mas sim para RISC-V de 32 bits.

Você pode verificar o conteúdo do arquivo `arquivo_de_saida.s` (produzido pelo comando acima) abrindo este arquivo em seu editor de texto favorito. Ele é um arquivo texto e contém o mesmo programa que você escreveu em C, porém transcrito para linguagem de montagem para o processador de arquitetura RISC-V RV32. Note que a linguagem de montagem faz referência a instruções (`add`, `mv`, *etc.*) e outros elementos específicos de cada tipo de processador e, consequentemente, é dependente da interface do mesmo.

Montagem

Para converter o código em linguagem de montagem para linguagem de máquina você pode usar o montador (*assembler*) da seguinte forma:

```
clang-12 --target=riscv32 -march=rv32g -mabi=ilp32d
-mno-relax arquivo_de_entrada.s -c -o arquivo_de_saida.o
```

Você não pode abrir o arquivo produzido (`arquivo_de_saida.o`) em seu editor de texto, pois é um arquivo binário. Para analisar esse arquivo, você precisa de programas especiais chamados "desmontadores", que interpretam o conteúdo do arquivo e convertem sua representação para texto. Você pode utilizar a ferramenta `objdump` para desmontar o arquivo binário em linguagem de máquina. Execute o comando

```
llvm-objdump-12 -D arquivo_de_saida.o
```

Tente comparar a saída produzida pelo desmontador (`llvm-objdump`) com o arquivo de entrada usado durante o processo de montagem (`arquivo_de_entrada.s`). Você perceberá que eles são diferentes, mas possuem diversas informações em comum (p.ex., listas de instruções a serem executadas pelo processador).

Ligação

Tendo os arquivos objeto em mãos, podemos executar o ligador (*linker*) através do seguinte comando:

```
ld.lld-12 arquivo_de_entrada.o -o executavel.x
```


Caso haja mais de um arquivo objeto, eles devem ser listados em sequência no ligador. Exemplo:

```
ld.lld-12 arquivo_de_entrada1.o arquivo_de_entrada2.o -o
executavel.x
```

Execução

Executar um programa escrito em linguagem de montagem do RISC-V exige o uso de um simulador (ou *hardware*) RISC-V, pois nossos computadores possuem processadores com conjunto de instruções da família de arquiteturas x86, sendo assim incompatíveis com código RISC-V. Nesta disciplina, utilizaremos o RISC-V ALE para simular um processador RISC-V.

Simulação

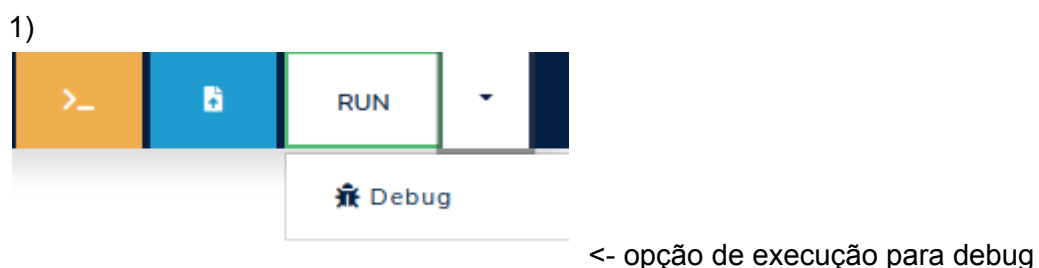
1. Acesse a página do [Simulador RISC-V](#) (caso já não esteja visualizando este enunciado através dela):
2. Clique em  (no canto superior direito) e selecione o executável
3. Clique em *Run* para iniciar a execução.
4. Observe a saída do seu código (se houver) no painel que irá abrir - verifique que a mensagem Hello world! foi impressa no terminal.

Note que, após o término da execução do seu programa, o simulador pode apresentar erros como “Error: Failed stop: 64 consecutive illegal instructions: 0”. Isso é esperado, já que o processador não sabe onde seu programa termina e tentará executar trechos de memória que não representam instruções válidas.

Depuração

Se você adicionar a *flag* -g aos comandos de compilação (clang) e ligação (lld) e, ao executar o simulador, escolher a opção debug ao invés de run, você poderá executar iterativamente sua aplicação e ver os resultados instrução por instrução.

Para executar uma aplicação no modo iterativo:



2)



^ terminal interativo

Os comandos do terminal interativo são:

```
run
    Run till interrupted.
until <address>
    Run until address or interrupted.
step [<n>]
    Execute n instructions (1 if n is missing).
peek <res> <addr>
    Print value of resource res (one of r, f, c, m) and address addr.
    For memory (m) up to 2 addresses may be provided to define a range
    of memory locations to be printed.
    examples: peek r x1    peek c mtval    peek m 0x4096
peek pc
    Print value of the program counter.
peek all
    Print value of all non-memory resources
poke res addr value
    Set value of resource res (one of r, c or m) and address addr
    Examples: poke r x1 0xff    poke c 0x4096 0xabcd
symbols
    List all the symbols in the loaded ELF file(s).
quit
    Terminate the simulator
```

Os principais comandos que irão te ajudar nos próximos labs e portanto você deve se familiarizar são:

- run (executa a aplicação);
- until (executa a aplicação até um endereço específico);
- step (executa as próximas n instruções);
- peek (escreve na tela o valor de um registrador ou endereço de memória); e
- poke (seta o valor de um registrador ou endereço de memória).

Por exemplo, para executar uma instrução do seu programa e escrever na tela o valor atual do registrador x10¹:

```
$ whisper /working/modelo.x --newlib --setreg sp=0x7FFFFFFC --interactive --isa acdfimsu
wasm streaming compile failed: TypeError: Failed to execute 'compile' on 'WebAssembly': Incorrect response MIME type. Expected 'application/wasm'.
falling back to ArrayBuffer instantiation
calling stub instead of sigaction()
>>> step 1
#1 0 000110b4      4505 r 0a      00000001 c.li      x10, 0x1
>>> peek r 10
0x00000001
>>> █
```

GNU makefile

O processo de desenvolvimento de *software* envolve diversas iterações de correções de *bugs* e recompilações. Entretanto, muitos destes projetos possuem uma quantidade grande de arquivos de programa e a compilação de todos os arquivos é um processo lento. Os arquivos objeto (.o) precisam ser ligados novamente para formar o novo binário, no entanto, apenas os arquivos modificados precisam ser recompilados. Dessa forma é importante ter um mecanismo automático para recompilar apenas os arquivos necessários. Para isso, existe uma modalidade de *script* específica para automatizar a compilação de *softwares*. O GNU makefile é um exemplo largamente utilizado no mundo GNU/Linux. Para instalá-lo em uma distribuição baseada no Debian, execute o seguinte comando:

```
sudo apt-get install build-essential
```

Para fazer o seu próprio *script* que irá orientar o GNU make a construir o seu programa, você deve criar um arquivo texto chamado Makefile, que deve estar na mesma pasta dos códigos-fonte, contendo regras para a criação de cada arquivo. Por exemplo, você pode criar regras para especificar como o arquivo .s (em linguagem de montagem) é criado (utilizando o compilador clang), especificar como os arquivos-objeto .o (códigos-objeto) são criados (utilizando o montador) e assim em diante. Exemplo de criação de regras:

```
ola.s: ola.c

        clang-12 --target=riscv32 -march=rv32g -mabi=ilp32d
-mno-relax ola.c -S -o ola.s

ola.o: ola.s

        clang-12 --target=riscv32 -march=rv32g -mabi=ilp32d
-mno-relax ola.s -c -o ola.o
```

¹ Registrador é um dispositivo de armazenamento interno do processador. As instruções de um programa podem gravar ou ler valores destes dispositivos. Durante o curso discutiremos como utilizar estes dispositivos em nossos programas.

Neste exemplo existem duas regras, nomeadas `ola.o` e `ola.s`. A regra `ola.o` deve corresponder ao arquivo que é produzido com essa regra. Os arquivos necessários para produzir o arquivo `ola.o` devem aparecer em uma lista (separada por espaços) após o caractere ":" (no nosso caso, `ola.s` é necessário para criar `ola.o`). Em seguida, você deve, na linha seguinte, usar uma tabulação (apertar a tecla `tab`) e digitar o comando que será executado no *shell* para produzir esse arquivo. No nosso exemplo, chamamos o compilador `gcc` para traduzir um arquivo em linguagem C para linguagem de montagem, e em outra regra, chamamos o montador para transformar um arquivo em linguagem de montagem `.s` em um arquivo-objeto `.o`. Note que você pode especificar como arquivo de entrada de uma regra o nome de outra regra, e esta outra regra será chamada antes para produzir o arquivo de entrada necessário. ATENÇÃO: O *script* não funcionará se não houver uma tabulação (`tab`) antes dos comandos "`clang-12 ...`". Não use espaços! Além disso, note que alguns editores de texto incluem espaços em vez do caracteres de tabulação quando a tecla `tab` é pressionada.

Você pode criar várias regras em um mesmo arquivo Makefile. Para executar o *script*, na linha de comando, digite `make nome-da-regra`. Por exemplo:

```
make nome-da-regra
```

O programa `make` irá executar os comandos associados à regra `nome-da-regra`, descrita no Makefile. Note que o programa `make` sempre lê o arquivo de nome Makefile na pasta em que você está e o usa como *script*. Se você não utilizar esse nome de arquivo (Makefile com "M" maiúsculo), o *script* irá falhar. Se você invocar o comando `make` sem parâmetros ele executará a primeira regra do arquivo Makefile.

Tarefa 2

Agora você criará o arquivo Makefile para gerar o programa da Tarefa 1 automaticamente. Testaremos as seguintes regras:

```
make arquivo1.s
```

```
make arquivo2.s
```

```
make arquivo1.o
```

```
make arquivo2.o
```

```
make executavel.x
```


Elas devem gerar, respectivamente: o código em linguagem de montagem do arquivo1 e do arquivo2; o objeto do arquivo1 e do arquivo2; o executável final.

Referência para Makefile:

- Guia em português: http://pt.wikibooks.org/wiki/Programar_em_C/Makefiles
- Manual original (em inglês):
<http://www.gnu.org/software/make/manual/make.html#Simple-Makefile>

Informações extras

Instalação dos requisitos em seu computador

Nesta disciplina, utilizaremos o compilador CLANG. O compilador está disponível nos laboratórios do IC, no entanto, caso deseje utilizá-lo em seu próprio computador, você pode tentar as seguintes opções:

1. Instalando em uma distribuição GNU/Linux

Você pode instalar uma distribuição Linux na sua máquina ou em uma máquina virtual utilizando, por exemplo, o Oracle [VirtualBox](#).

Em distribuições baseadas no Debian (e.g. [Ubuntu](#)), digite os seguintes comandos no terminal:

```
sudo apt update  
sudo apt install clang-12 lld-12
```

O sistema deve pedir sua senha (ou a senha do superusuário da máquina). Uma vez que o processo for concluído, você terá o ambiente configurado para a atividade de laboratório.

2. Instalando no Windows com WSL2

A Microsoft, a partir do Windows 10, permite que o usuário instale um ambiente GNU/Linux no Windows. Para isso, você pode seguir o tutorial: [Instalar o WSL no Windows 10](#). Sugerimos a instalação da distribuição Ubuntu 20.04 LTS. Há também diversos tutoriais no Youtube (e.g.: [1](#) e [2](#)). Também sugerimos a instalação do [Terminal do Windows](#) e do [VSCode](#).

Depois de instalado, inicie um terminal de sua distribuição WSL e digite os seguintes comandos (no caso das distribuições baseadas no Debian):

```
sudo apt update  
sudo apt install clang-12 lld-12
```

O sistema deve pedir sua senha (ou a senha do superusuário da máquina). Uma vez que o processo for concluído, você terá o ambiente configurado para a atividade de laboratório.

3. Utilizando o Repl.it

O [Replit](https://replit.com/) é um ambiente de desenvolvimento (IDE) online que lhe dá acesso a uma pequena máquina virtual no navegador. É possível neste ambiente para e editar e compilar programas com as ferramentas clang-12 e lld-12.

Para isso, você deve instalar as ferramentas clang-12 e lld-12. O processo de instalação pode ser diferente, dado que este ambiente tem evoluído nos últimos anos. Da última vez que fiz, utilizei os seguintes passos e comando para instalar o clang-12 e o lld-12. Faça login na plataforma, crie uma “repl” para a linguagem C e execute o seguinte comando no Shell:

```
sudo apt install clang-12 lld-12
```

O Replit salva seus arquivos automaticamente (desde que você esteja logado), mas os pacotes instalados são apagados quando a máquina virtual desliga ou reinicia. Assim, é necessário instalar o CLANG novamente (comando acima) sempre que a máquina for iniciada.