UNIVERSIDADE FEDERAL DE MINAS GERAIS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
DCC008 – SOFTWARE BÁSICO – TB

Trabalho Prático I

Montador para a máquina *Swombat*

Alunos: Christian Vieira
Matheus Vargas
Matheus de Paula
Pedro Mariano

Professor: Daniel Macedo
Monitor: Daniel Vieira

Belo Horizonte
15 de maio de 2017

Sumário

Lista de Figuras

Lista de implementações

Lista de Tabelas

List of Algorithms

1 Introdução

Salomon (SALOMON, 1993) considera um *assembler* como um tradutor que traduz instruções provenientes de uma linguagem simbólica em instruções de linguagem de máquina para uma determinada arquitetura alvo, sendo a tradução feita de um para um.

Uma das razões para o estudo de *assemblers* reside no fato de que as operações de um *assembler* refletem a arquitetura do computador alvo uma vez que a linguagem de montagem é fortemente dependente da organização interna do computador. Características arquiteturais tais como o tamanho da palavra de instrução/dados, formato numérico, codificação interna de caracteres, registradores de propósitos específicos e gerais, organização da memória, tipos de endereçamento e instruções, etc; afetam o modo de como as instruções são escritas e como o *assembler* lida com as instruções e diretivas associadas.

O assemblador considerado neste trabalho é o assemblador de dois passos, descrito no livro texto adotado na disciplina de *Software Básico*. Um assemblador de dois passos consiste em uma implementação que faz a leitura do código fonte em linguagem simbólica duas vezes, a primeira passagem para a resolução da *referência antecipada* em que as definições de símbolos e rótulos de declarações são coletadas e armazenadas em uma tabela. A segunda passagem todos os valores simbólicos são conhecidos, não restando nenhuma *referência antecipada*, então, cada declaração lida pode ser montada e traduzida. O método de duas passagens apesar de requer uma passagem extra, é um método de implementação simples. Maiores detalhes sobre o processo de assemblagem por duas passagens podem ser obtidos em (TANENBAUM; AUSTIN, 2013) [cap. 7].

Tipicamente um assemblador moderno possui duas entradas e saídas. A primeira entrada é a que ativa o assemblador e especifica os parâmetros e o nome do arquivo de entrada. A segunda entrada é o arquivo fonte caminho (contendo código fonte em linguagem simbólica de máquina e diretivas de montagem). A primeira saída de um assemblador típico é o arquivo objeto o qual contém as instruções montadas, ou seja, um programa em linguagem de máquina a ser posteriormente carregado e executado na máquina alvo. A segunda saída típica de um assemblador é o arquivo de listagem o qual contém informações relevantes do processo de montagem tal como as instruções de máquina e diretivas (SALOMON, 1993).

1.1 Objetivo

O objetivo deste trabalho concerne o desenvolvimento de um montador *(assembler)*, um *software* capaz de realizar a tradução de um programa escrito em uma linguagem de montagem para uma linguagem de máquina específica, neste caso, a *Swombat*, a qual será simulada através do *software* simulador *CPUSim*<http://www.cs.colby.edu/djskrien/CPUSim/>.

Espera-se na execução deste trabalho praticar os conceitos desenvolvidos na disciplina de *Software Básico* tais como o processo de tradução de linguagem simbólica para linguagem de máquina, uso de um *software* simulador de máquina alvo (*CPUSim* simulando a máquina *Swombat*), estruturas de dados adequadas para manipulação de *arrays* associativos, processo de *tokenização*, análise léxica, análise sintática e tradução para a máquina alvo.

2 Implementação

A implementação do assemblador para a máquina *Swombat* foi baseada nos conceitos do montador de dois passos descrito em (TANENBAUM; AUSTIN, 2013) [cap. 7]. Basicamente, o primeiro passo consiste na *tokenização* do arquivo contendo a implementação em linguagem simbólica, resolvedo a questão da *referência posterior* através da montagem das *tabelas de símbolos* e *pseudo-instruções*. As tabelas foram elaboradas utilizando *arrays associativos* do tipo *hash*, facilitando tanto a inserção de símbolos quanto à pesquisa destes. A segunda passagem é realizada de forma que sempre que um símbolo com referência seja encontrado, a referência é pesquisada em uma das tabelas conforme o caso (*tabela de símbolo* ou *tabela de pseudo-instruções*) e então toda a instrução é montada e inserida no arquivo de saída. A composição do programa em sua totalidade poderá ser observada nos blocos da figura 1 a qual ilustra os blocos e o inter-relacionamento deles (nem todos os blocos foram implementados no trabalho).

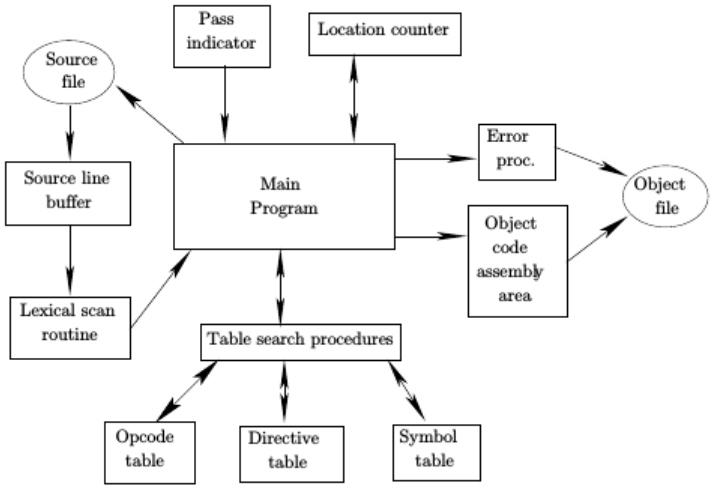

Figura 1 Composição do assemblador para a máquina *Swombat* (SALOMON, 1993)

2.1 A gramática do assemblador *Swombat*

A *Gramática Livre de Contexto Livre (CFG)* exibida abaixo especifica a sintaxe aceita pelo assemblador da máquina *Swombat*. Entretanto, por ser um trabalho didático e com pouco tempo para a execução, somente parte da gramática descrita abaixo foi passível de implementação, conforme descrito nas especificações do trabalho prático (MACEDO; VIEIRA, 2017).

```
[fontsize=\footnotesize]
Program                    -> [CommentsAndEOLs] EquMacroIncludeGlobalPart InstructionPart EOF
CommentsAndEOLs            -> ([Comment] EOL)+
Comment                    -> ";" <any-sequence-of-characters-not-including-EOF-or-EOL>
EquMacroIncludeGlobalPart  -> ((EquDeclaration | MacroDeclaration |
                              Include | Global) CommentsAndEOLs)*
Include                    -> ".include" <string-of-characters-not-including-EOL-or-EOF>
EquDeclaration             -> Symbol "EQU" Operand
MacroDeclaration           -> "MACRO" Symbol [Symbol ([","] Symbol)*]
                               CommentsAndEOLs InstructionPart "ENDM"
Global                     -> ".global" Symbol
InstructionPart            -> ((RegularInstructionCall | DataPseudoinstructionCall |
                                AsciiPseudoinstructionCall | Include) CommentsAndEOLs)*
RegularInstructionCall     -> (Label CommentsAndEOLs)* [Label] Symbol [Operand ([","] Operand)*]
DataPseudoinstructionCall  -> (Label CommentsAndEOLs)* [Label] ".data" Operand [","] ( Operand |
                              [Operand [","]] "[" [Operand ([","] Operand)*] "]")
AsciiPseudoinstructionCall -> (Label CommentsAndEOLs)* [Label] ".ascii" String
Label                      -> Symbol ":"
Operand                    -> Symbol | Literal
Literal                    -> [ "-" | "+" ] ( ( 0-9 )+ | "0x"( 0-9a-fA-F )+ |
                               "0b"( 0 | 1 )+ | <single-quoted-character>)
```

2.2 Inserção de dados da *pseudo-instrução: ".data"*

A tomada da decisão para armazenamento dos dados oriundos do uso da *pseudo-instrução: ".data"* foi dada levando em consideração que a posição de memória *254* é reservada como um endereço de *IO*. Assim, definiu-se que os dados da *pseudo-instrução: ".data"* ficariam armazenados logo após a instrução de término do programa (instrução: `exit`). Posteriormente, foi observado que a política adotada pelo assemblador do *CPUSim* foi a mesma, o que deu maior embasamento à política de alocação implementada.

3 Ambiente de desenvolvimento

O ambiente de desenvolvimento utilizado para a composição, compilação, execução e testes é dado abaixo:

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=16.04
DISTRIB_CODENAME=xenial
DISTRIB_DESCRIPTION="Ubuntu 16.04.2 LTS"
gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.4)
```

Conforme poderá ser visto no arquivo `makefile` (no diretório `src`), o compilador utilizado foi o compilador *C++ GCC* padrão `C++11`, o uso da diretiva `-std=c++11` especifica o padrão e versão da linguagem utilizada.

3.1 Compilação

Para compilar o programa, basta ir ao diretório onde encontra-se o código fonte (diretório `assemler`)e através do utilitário `make`, executar o comando. Automaticamente o utilitário fará a leitura do arquivo `makefile` o qual tem uma espécie de *"receita"* para a compilação do programa.

3.2 Execução do programa

A execução do programa é dada da seguinte forma (ambiente *GNU/Linux*):

```
$ .\mont <file_in.a> <file_out.mif>
```

Ao executar o comando acima, o programa: `mont` será invocado, tendo como argumento o arquivo de entrada: `file_in.a` e opcionalmente um nome para o arquivo de saída `file_out.mif`. Caso usuário não especifique um arquivo de saída, um nome *default* é dado para o arquivo de saída (`a.mif`), neste caso, a forma de invocação do programa é: `$ .\assembler <file_in.a>`. Caso não ocorra nenhum erro de montagem um arquivo com a extensão `*.mif` será dado. Caso ocorra alguma anomalia, uma mensagem de erro é dada na saída de erros padrão (`stderr`) e a execução do programa é encerrada.

3.3 Avaliação de alocação/desalocação de recursos

Afim de avaliar a correta alocação e desalocação de recursos bem como vazamento de memória, a ferramenta de *software valgrind* (<http://valgrind.org/>) (SEWARD et al., 2016) foi utilizada, os resultados obtidos inicialmente apresentavam vazamentos de memória. Após a verificação e identificação das anormalidades, as correções foram realizadas e o resultado da execução mostrou o correto gerenciamento dos recursos utilizados ao longo da execução do programa (assemblador). A saída do verificador é dada abaixo:

```
[fontsize=\footnotesize]
chris@chris:~/work/ufmg/software-basico_dcc008/2017-1/tps/tp1/assembler$
valgrind --tool=memcheck --leak-resolution=high --leak-check=full
--track-origins=yes --show-reachable=yes --show-leak-kinds=all
./mont ../tst/iterativeFibonacci.a ../tst/mont_iterativeFibonacci.mif
==21167== Memcheck, a memory error detector
==21167== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==21167== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==21167== Command: ./mont ../tst/iterativeFibonacci.a ../tst/mont_iterativeFibonacci.mif
==21167==
==21167==
==21167== HEAP SUMMARY:
==21167==     in use at exit: 72,704 bytes in 1 blocks
==21167==   total heap usage: 219 allocs, 218 frees, 91,034 bytes allocated
==21167==
==21167== 72,704 bytes in 1 blocks are still reachable in loss record 1 of 1
==21167==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==21167==    by 0x4EC3EFF: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21)
==21167==    by 0x40104E9: call_init.part.0 (dl-init.c:72)
==21167==    by 0x40105FA: call_init (dl-init.c:30)
==21167==    by 0x40105FA: _dl_init (dl-init.c:120)
==21167==    by 0x4000CF9: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==21167==    by 0x2: ???
==21167==    by 0xFFEFFFEA6: ???
==21167==    by 0xFFEFFFEAD: ???
```

```
==21167==    by 0xFFEFFFEC9: ???
==21167==
==21167== LEAK SUMMARY:
==21167==    definitely lost: 0 bytes in 0 blocks
==21167==    indirectly lost: 0 bytes in 0 blocks
==21167==      possibly lost: 0 bytes in 0 blocks
==21167==    still reachable: 72,704 bytes in 1 blocks
==21167==         suppressed: 0 bytes in 0 blocks
==21167==
==21167== For counts of detected and suppressed errors, rerun with: -v
==21167== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Conforme esperado, não foi encontrado nenhum tipo de vazamento de memória ou falha de segmentação, confirmando assim a alocação e desalocação correta de recursos dinâmicos utilizados ao longo da execução do programa. A informação *still reachable: 72,704 bytes in 1 blocks* é devida ao esquema de captura de excessões da *linguagem C++*, não tendo nenhuma relação com situações indesejadas do gerenciamento de recursos efetuados (maiores informações são dadas em: (ZUBKOV, 2017)).

4 Resultados

Para testarmos o montador criamos dois programas simples na linguagem de montagem da máquina *Swombat*, apresentados nas seguintes subseções:

4.1 Programa 1 - Fatorial usando a Pilha

O primeiro programa é uma função do cálculo do fatorial usando a pilha interna da máquina. O número que deseja-se calcular o fatorial é dado através da leitura da entrada padrão (teclado), o fatorial é então calculado e o valor impresso na saída padrão (monitor). O valor máximo do fatorial calculado é o 8. Acima deste valor, por restrições da máquina, o resultado é imprevisível. Opcionamente poderia ser feito um programa de maior complexidade, mas devido ao tempo escasso, optou-se por implementar o algoritmo de forma mais simples, porém suficiente para verificar o funcionamento correto da implementação e do assemblador.

4.2 Programa 2 - Fibonacci Iterativo

Este programa calcula a função de *Fibonacci* de forma iterativa. O valor a ser calculado é dado na entrada padrão (teclado) e o valor calculado é impresso na saída padrão (monitor). Assim como no caso anterior, a implementação é limitada a poucos casos, dado a limitação da máquina alvo.

4.3 Programa 3 - Exemplo - Multiplicação de dois inteiros por soma sucessiva

O terceiro e último programa em linguagem de máquina foi inicialmente dado como integrante do pacote de *software* para avaliação e testes do "ambiente" *CPUSim*. Apesar de não necessariamente ser um exemplo para teste no assemblador escrito, o mesmo foi assemblado (utilizando o assemblador construído) e executado no *software* de simulação. Em linhas gerais, o programa realiza a multiplicação de dois inteiros através de soma sucessiva (sem o uso da instrução de máquina de multiplicação).

As telas contendo as simulações bem como as implementações dos programas de testes podem ser observadas no Apêndice:A, seções: A.1, A.2 e **??**. Figuras: 2, 3 e **??**. Observa-se que todos os resultados apresentados estão em conformidade com os requerimentos pretendidos dos algoritmos (Programa 1: Fatorial usando a Pilha, Programa 2: Fibonacci Iterativo e por fim o Programa 3: multiplicação de dois inteiros por soma sucessiva) implementados em linguagem de montagem.

5 Limitações conhecidas

As limitações atuais do assemblador implementado são dadas abaixo:

- Suporte limitado a pseudo-instruções: atualmente há suporte para somente a pseudo-instrução: `.data`, em que a mesma deve ter o formato: `label .data n m` em que n e m são respectivamente o tamanho (em *bytes*) a ser utilizado na memória e o dado a ser inserido na memória

- Geração de arquivo de saída: atualmente somente o formato `*.mif` é dado, tendo como base fixa (*radix*) tanto para o endereço de memória quanto para instruções/dados, além de não oferecer a inclusão de comentários tal qual pode ser encontrado ao se gerar um arquivo de saída no *software CPUSim*
- Parser primitivo na linha de argumentos: o parser atual suporta somente a leitura de um arquivo de entrada e um de saída, não oferecendo possibilidade para assemblagem múltiplas de arquivos.
- Símbolos/seções não alcançáveis: não há suporte para identificação e eliminação de *label's*, sub-rotinas e seções de dados não alcançáveis.
- Símbolos/seções múltiplas: não há identificação de *label's* em duplicidade (indicação de erro), ou de seções de programa e dados iguais (os quais podem ser otimizados)
- analisador léxico/sintático: a análise léxica e sintática da linguagem suportada pelo assemblador necessita de maiores ajustes e implementações quanto a checagem de erros e separação explícita entre os módulos constituintes, suportando desta forma a especificação completa da linguagem simbólica descrita na gramática da linguagem de montagem da máquina *Swombat*

6 Futuras implementações

Com o intuito de dar continuidade ao projeto de forma que o assemblador esteja em conformidade com a gramática apresentada da linguagem de montagem da máquina *Swombat*, os seguintes itens deverão ser acrescentados e/ou modificados na implementação:

- Uso de uma estrutura de dados associativa do tipo *hash* perfeito ou minimamente perfeito (WIKIPEDIA, 2016a) para as tabelas de símbolos, *opcodes*, pseudo-instrução, etc. Atualmente a estrutura de dados utilizada é o `unordered_map` que é uma *hash* implementada na*STL (Standard Template Library)* (WIKIPEDIA, 2016b)
- Re-escrita da implementação de forma a torná-la mais amigável a futuros implementadores ou mesmo manutenção. O estilo orientado à objetos seria o mais apropriado
- Uso de *strings* ao estilo *C++* e não *C* ou mesmo a mistura de ambos os estilos
- Parser para argumentos provenientes da entrada padrão (para invocar o assemblador com maiores opções na linha de comandos)
- Opção de *dump* no arquivo assemblado de forma a exibir o conteúdo de uma determinada área da memória (seção de texto, dados, constantes, etc)
- Geração de um arquivo `*.lst` (arquivo de listagem) contendo mesclagem da listagem simbólica (instruções, label's e diretivas)
- Identificação e eliminação de *label's* e sub-rotinas não alcançáveis
- Identificação de *label's* duplicados
- Suporte a assemblagem múltipla de arquivos
- Suporte para geração de arquivos de saída no formato `*.mif` e intel `*.hex`. No caso do formato `*.mif`, deverá ter opções de base utilizada *(radix)* e inclusão de comentários oriundos do código fonte (código em linguagem simbólica)
- Nas mensagens de erros, retirar caminho absoluto dos nomes dos arquivos, melhorando assim a interface com o usuário
- Inserir nas mensagens de erros a coluna correspondente de onde o erro foi encontrado, imprimir a linha e abaixo da linha de texto (código em assembly), o caracter `^` posicionando onde o erro foi encontrado

7 Conclusões

O *trabalho prático* propiciou maior entendimento da composição (programação) e organização (estruturação) de um pequeno assemblador para uma máquina específica *(Swombat)*. Seguindo as orientações contidas no livro texto da disciplina (TANENBAUM; AUSTIN, 2013)[cap. 7] referente à composição de um assemblador de dois passos.

Objetivando princípios de *engenharia de software* o assemblador foi escrito de forma multimodular (vários arquivos) de forma a ter baixo acoplamento entre as partes integrantes do projeto e alta reusabilidade, aproveitando a capacidade de expressão da linguagem utilizada *(Linguagem C++)*. As principais estruturas de dados utilizadas (tabela de dispersão ou tabela *hash*, lista, *strings*, etc) foram as disponíveis da biblioteca padrão da linguagem e também da *STL*. A compreensão da manipulação e alocação das es-

truturas inicialmente foi tida como difícil, dado o paradigma de visualizar mentalmente as estrutura de dados utilizadas e como manipulá-las durante a composição do código fonte. A ferramenta de *debbug* e checagem de vazamento de memória *Valgrind* <http://valgrind.org/> foi utilizada com o intuito de checar as alocações e desalocações de recursos ao longo do programa. A saída verificada pela ferramenta *Valgrind* não mostrou qualquer sinal de violação de extravasamento de memória ou ainda recursos não usados mas que ainda permanecem alocados.

O desenvolvimento do trabalho ocorreu normalmente, sem grandes percalços, uma vez que a especificação do trabalho foi dada de forma clara e objetiva, tendo ainda à disposição de um *software* simulador da máquina *Swombat (CPUSpim)* para comprovação do funcionamento do assemblador (o qual traduz linguagem simbólica para linguagem de máquina) na máquina alvo.

8 Referências Bibliográficas

MACEDO, D.; VIEIRA, D. *Trabalho Prático 1 - Montador*. 2017. [Online; acessado em 15/05/2017]. Disponível em: <https://virtual.ufmg.br/20171/pluginfile.php/134447/mod_forum/attachment/39853/2017-1_TP1%20-%20Software%20Ba%CC%81sico-%20V3.pdf>.

SALOMON, D. *Assemblers and Loaders*. 1993. [Online; acessado em 07/10/2016]. Disponível em: <http://www.davidsalomon.name/assem.advertis/asl.pdf>.

SEWARD, J. et al. *Valgrind*. 2016. [Online; acessado em 07/10/2016]. Disponível em: <http://valgrind.org>.

TANENBAUM, A. S.; AUSTIN, T. *Structured Computer Organization*. 6. ed. [S.l.]: Pearson, 2013. ISBN 0-13-291652-5.

WIKIPEDIA. *Perfect hash function*. 2016. [Online; acessado em 07/10/2016]. Disponível em: <https://en.wikipedia.org/wiki/Perfect_hash_function>.

WIKIPEDIA. *Unordered associative containers (C++)*. 2016. [Online; acessado em 07/10/2016]. Disponível em: <https://en.wikipedia.org/wiki/Unordered_associative_containers_(C%2B%2B)>.

ZUBKOV, S. *How can I avoid still reachable memory-leak in C++ exception handling?* 2017. [Online; acessado em 15/05/2017]. Disponível em: <https://www.quora.com/How-can-I-avoid-still-reachable-memory-leak-in-C++-exception-handling>.

Apêndice A Execução dos programas assemblados no simulador *CPUSim*

A.1 Programa 1 - Fatorial usando a Pilha

```
1  loadi A1 IO
2  loadi A0 _one
3  loadi A3 _one
4  call _fat_n
5  exit
6  _fat_n: jmpz A1 _fat_zero
7  push A1
8  subtract A1 A3
9  pop A2
10 multiply A0 A2
11 call _fat_n
12 return
13 _fat_zero: storei A0 IO
14 return
15 _one:    .data 2 1
```

Listagem 1 Listagem do programa em assembly que calcula o fatorial

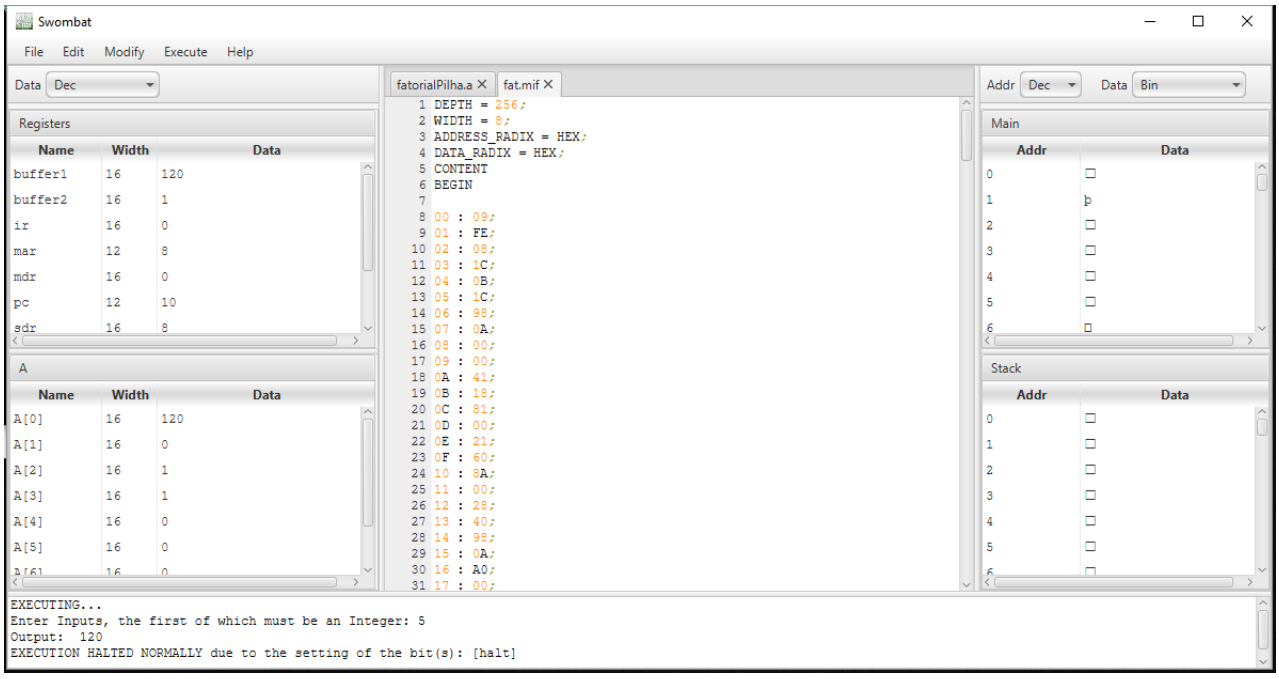

Figura 2 Resultado da simulação do programa que calcula o fatorial de um número

A.2 Programa 2 - Fibonacci Iterativo

```
1  loadi A0 IO ; n-simo termo da sequencia
2  loadi A1 _one
3  loadi A2 _one
4  loadi A3 _two
5  loadi A6 _one
6  _fib_1: loadi A7 _one
7  subtract A7 A0
8  jmpz A7 _print_fib1
9  _fib_2: loadi A7 _two
10 subtract A7 A0
11 jmpz A7 _print_fib1
12 call _fibonacci_n
13 jump _eop
14 _print_fib1: storei A1 IO
15 _eop: exit
16 _fibonacci_n: add A4 A1
17 add A4 A2
18 move A2 A1
19 move A1 A4
20 clear A4
21 subtract A0 A6
22 subtract A3 A0
23 jmpz A3 _finally
24 loadi A3 _two
25 jump _fibonacci_n
26 _finally: storei A1 IO
27 return
28 _one:    .data 2 1
29 _two:    .data 2 2
```

Listagem 2 Listagem do programa em assembly que a soma de *Fibonacci de forma iterativa*

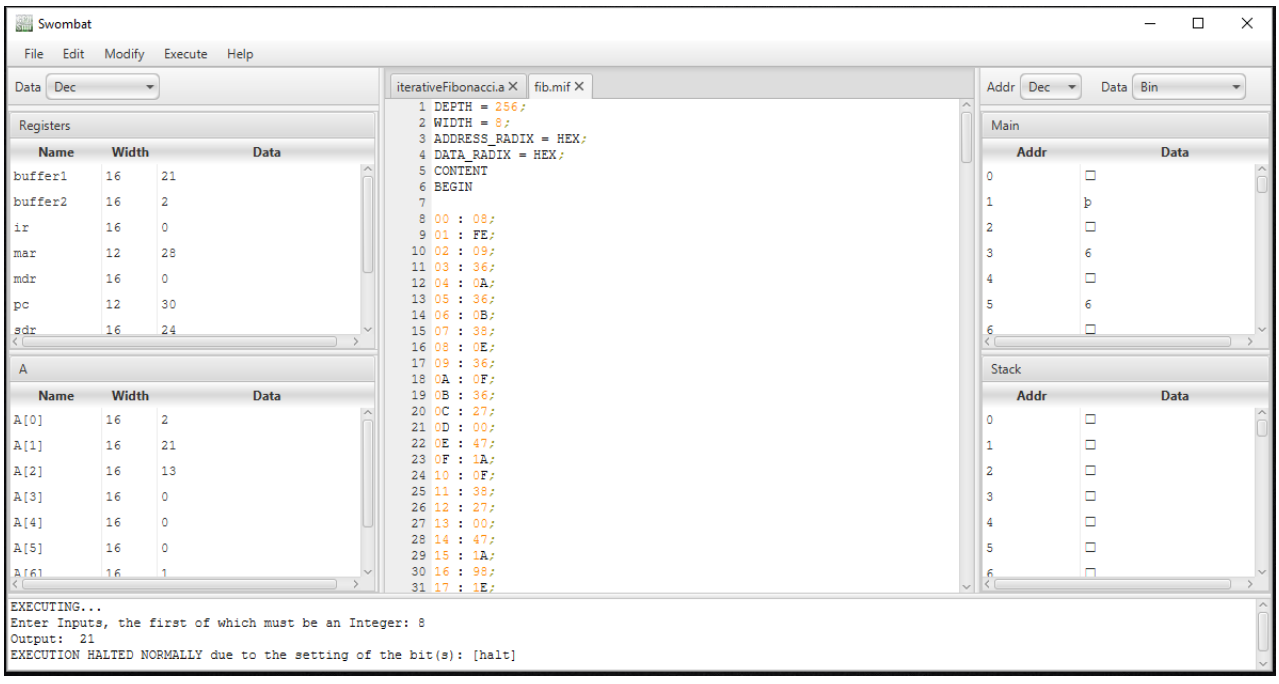

Figura 3 Resultado da simulação do programa que calcula a série de *Fibonacci*

A.3 Programa Exemplo: Multiplicação de dois inteiros por soma sucessiva

```
1                loadi A0 -5
2                loadi A3 _neg_one      ; load -1 -> A3
3                loadi A0 IO            ; read n -> A0
4                loadi A1 IO            ; read m -> A1 jmpn A1 _fix_sign     ; if m < 0 jump to
                     fix_sign
5                jump _clearA2          ; jump to clear
6  _fix_sign:    move A2 A1             ; fix_sign: m -> A2
7                subtract A1 A1         ; A1 = A1 - A1 = 0
8                subtract A1 A2         ; A1 = A1 - A2 = -m (= abs(m))
9                storei A3 _sign        ; store -1 into _sign
10 _clearA2:     subtract A2 A2         ; clear A2
11 _Start:       jmpz A1 _D_one         ; Start: jump to D_one if m = 0.
12               add A2 A0              ; add n to the sum in A2
13               loadi A7 _one
14               subtract A1 A7
15               jump _Start            ; go back to Start
16 _D_one:       loadi A0 _sign         ; D_one: load the _sign into A0
17               jmpn A0 _neg           ; if _sign < 0 jump to neg
18               jump _pos              ; jump to pos
19 _neg:         subtract A1 A2         ; neg: A1 = A1-A2 = -sum
20               move A2 A1             ; copy A1 into A2
21 _pos:         storei A2 IO           ; pos: output the final sum in A2
22               exit                   ; stop
23 _neg_one:     .data 2 -1             ; _neg_one: constant -1
24 _sign:        .data 2 1              ; _sign: (1 or -1)
25 _one:         .data 2 1              ; constant 1
```

Listagem 3 Listagem do programa em assembly que a multiplicação de dois inteiros por soma sucessiva

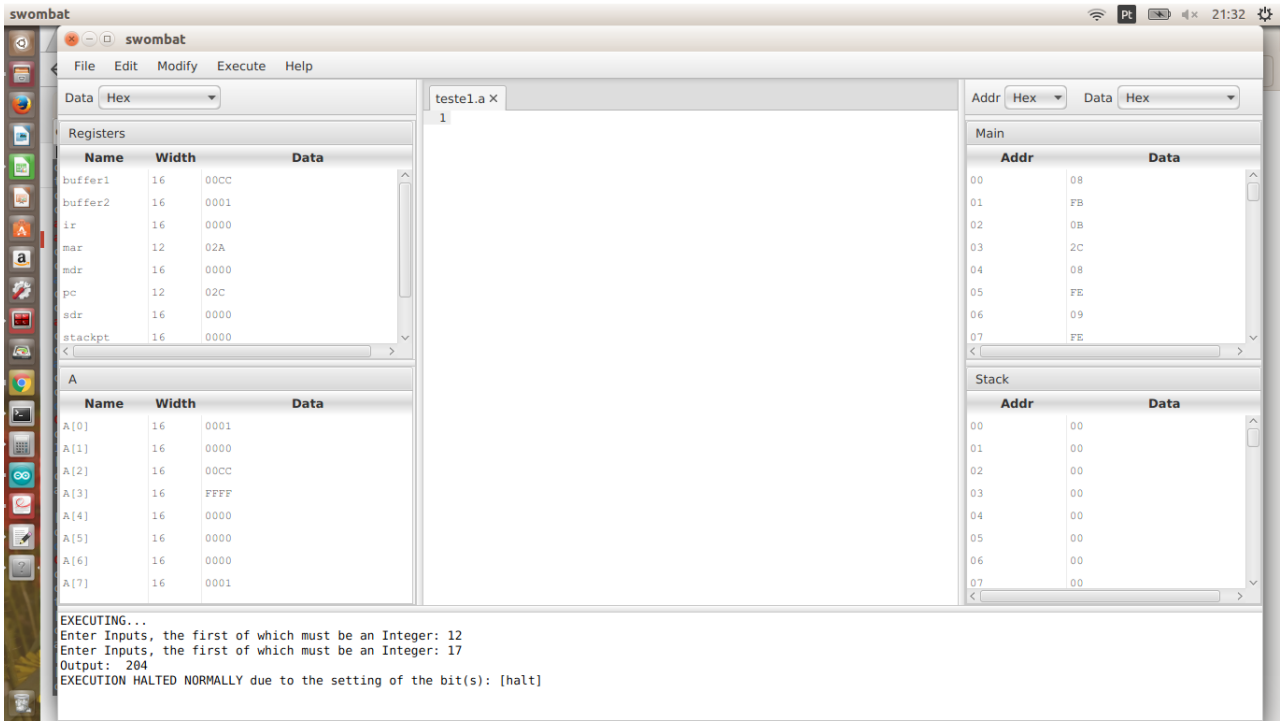

Figura 4 Simulação do programa que efetua a multiplicação de dois inteiros por soma sucessiva