Universidade Estadual de Londrina

PEDRO HENRIQUE MEDEIROS HERMES

REESCRITA DE PADRÕES

LONDRINA

2024

PEDRO HENRIQUE MEDEIROS HERMES

REESCRITA DE PADRÕES

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Wesley Attrot

LONDRINA

2024

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Sobrenome, Nome.
Título do Trabalho : Subtitulo do Trabalho / Nome Sobrenome. - Londrina, 2017.
100 f. : il.

Orientador: Nome do Orientador Sobrenome do Orientador.
Coorientador: Nome Coorientador Sobrenome Coorientador.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2017.
Inclui bibliografia.

1. Assunto 1 - Tese. 2. Assunto 2 - Tese. 3. Assunto 3 - Tese. 4. Assunto 4 - Tese. I. Sobrenome do Orientador, Nome do Orientador. II. Sobrenome Coorientador, Nome Coorientador. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

PEDRO HENRIQUE MEDEIROS HERMES

REESCRITA DE PADRÕES

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo
Membro da Banca – Sigla instituição

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro
Membro da Banca – Sigla instituição

Prof. Ms. Quarto Membro da Banca
Universidade/Instituição do Quarto
Membro da Banca – Sigla instituição

Londrina, XX de YY de 2024.

Este trabalho é dedicado às crianças adultas
que, quando pequenas, sonharam em se
tornar cientistas.

AGRADECIMENTOS

Aos meus pais, que fizeram o possível e o impossível para que eu alcançasse mais esse objetivo. Não existem palavras suficientes para expressar a minha imensa gratidão por vocês. Se um dia pude dizer que, finalmente, me formei cientista da computação, foi devido aos seus esforços e confiança depositados em mim.

"Não vos amoldeis às estruturas deste mundo, mas transformai-vos pela renovação da mente, a fim de distinguir qual é a vontade de Deus: o que é bom, o que Lhe é agradável, o que é perfeito.
(Bíblia Sagrada, Romanos 12, 2))

HERMES, PEDRO HENRIQUE. **Reescrita de padrões**. 2024. 33f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2024.

RESUMO

Palavras-chave: Latex. Template ABNT-DC-UEL. Editoração de texto.

HERMES, PEDRO HENRIQUE. **Title of the Work**. 2024. 33p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2024.

ABSTRACT

Keywords: Latex. ABNT-DC-UEL template. Text editoration.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

IR	*Intermediate representation*
AST	*Abstract syntax tree*
ISA	*Instruction Set Architecture*

SUMÁRIO

1 INTRODUÇÃO

Os compiladores são softwares responsáveis por realizar traduções de códigos escritos em linguagens de alto nível, como C/C++, Go, Rust, Fortran, etc., para linguagens de baixo nível que serão executadas em determinada arquitetura [1, 2, 3]. Para isso, todo o processo de compilação precisa ser quebrado em etapas menores, com uma única responsabilidade bem definida, que são agrupadas no *front-end*, *middle-end* e *back-end*.

O *front-end* é a etapa de análise constituída dos analisadores léxico, sintático e semântico. Seu objetivo é garantir que o código-fonte está em conformidade com as restrições gramaticais da linguagem em que foi escrito e que as sequências de instruções são semanticamente corretas, isto é, que são coerentes [1, 2, 3]. Ao final desse processo será emitida a representação intermediária equivalente ao código-fonte.

A Figura 1 mostra um exemplo das etapas contidas no *front-end*. A entrada consiste no código-fonte escrito em C/C++, Go, Fortran ou qualquer outra linguagem. Como saída, o *front-end* emite a representação intermediária que será processada nas demais etapas da compilação.

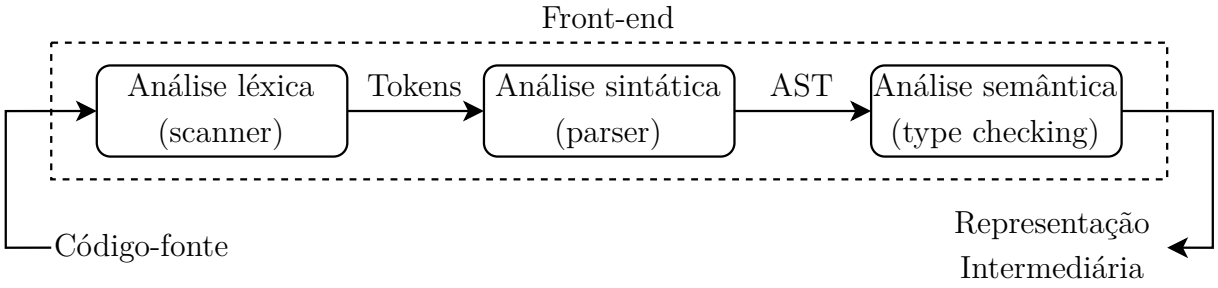


Figura 1 – *Front-end* do compilador. Fonte: adaptado de [1, 2, 3].

O *middle-end*, que é opcional mas amplamente utilizado, assume a responsabilidade de aplicar otimizações sobre a representação intermediária emitida pelo *front-end*. Tais otimizações podem estar mais ou menos relacionadas à determinada arquitetura dependendo do nível da representação intermediária, que pode ser de baixo, médio ou alto nível. Quando o baixo nível é escolhido, as otimizações são mais dependentes de arquitetura, enquanto no alto nível elas são menos dependentes de arquitetura [1, 2, 3].

Dentre o enorme conjunto de otimizações existentes, é possível encontrar *Dead-Code Elimination*, *Constant Folding*, *Constant Propagation*, *Partial-Redundancy Elimination*, etc. O compilador que realiza otimizações é nomeado "compilador otimizante" [1, 2, 3].

Por fim, o processo final da compilação, chamado "fase de síntese", ocorre no *back-end*, que assume a responsabilidade de tomar uma representação intermediária e gerar código para determinada arquitetura, chamado "código *assembly*". As principais etapas são o seletor de instruções, o alocador de registradores e o escalonador de instruções [1, 2, 3].

A Figura 2 mostra um exemplo das etapas contidas no *back-end*. A entrada consiste na representação intermediária previamente emitida pelo *front-end*. Entre uma etapa e outra, o *back-end* emite um código *assembly*, ao invés de outra representação intermediária. Por fim, a saída consiste no código de máquina inteiramente traduzido e otimizado para determinada arquitetura.

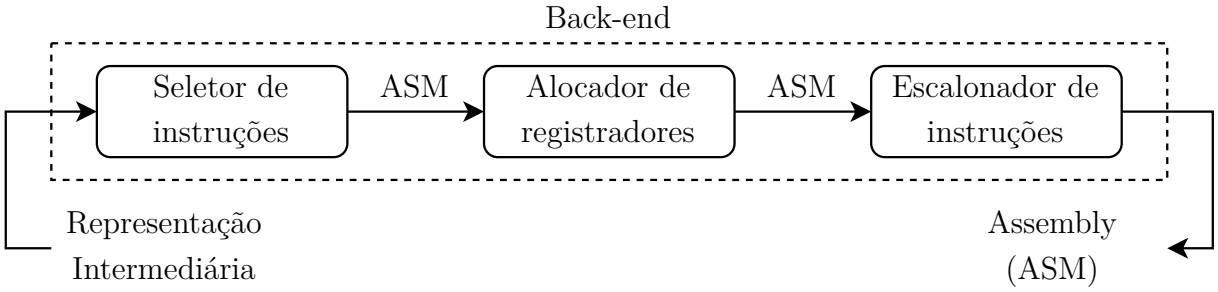


Figura 2 – *Back-end* do compilador. Fonte: adaptado de [1, 2, 3].

Na etapa de seleção de instruções, o compilador pode optar por traduzir comando por comando da representação intermediária para código de máquina ou utilizar padrões de subárvore mais complexos que resultam em código mais otimizado. Tal escolha está diretamente relacionada ao nível da representação intermediária [1, 2, 3]. Outras questões que influenciam a eficiência do código gerado é a quantidade de informação sobre a arquitetura alvo e os custos associados às instruções, que são difíceis de obter se não forem previamente disponibilizadas.

As representações intermediárias (IR, do inglês, *Intermediate Representation*), possuem a função de expressar com exatidão todo o código-fonte de forma que o compilador possa extrair conhecimento com eficiência [1, 2, 3]. Do ponto de vista computacional, é completamente inviável ao compilador iterar sobre o código-fonte puro, centenas ou milhares de linhas inúmeras vezes.

Uma IR básica em forma de árvore é chamada AST – *Abstract Syntax Tree*. A AST descreve toda a informação necessária sobre o código-fonte através de sua sintaxe em forma hierárquica. Por exemplo, uma operação de adição teria sua raiz representada pelo símbolo "+", ou pela palavra-chave "*sum*" [1, 2, 3]. A Figura 3 mostra uma AST para expressão matemática.

A abordagem para geração de código que utiliza os padrões de subárvore para

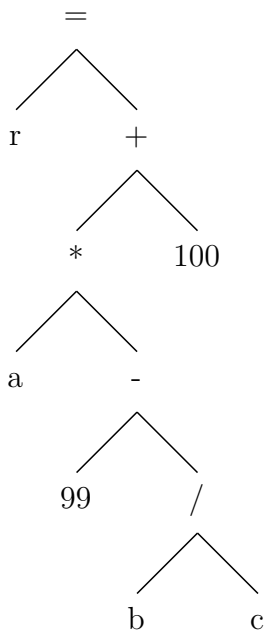


Figura 3 – Exemplo de AST para a expressão [r = a * (99 - b / c) + 100]. Fonte: autor.

tradução de código chama-se *Tree-Matching* [4]. Juntamente aos padrões estão contidas as informações de qual instrução deve ser emitida quando determinado padrão for encontrado na representação intermediária e qual é o custo computacional de se gerar tal instrução. A Figura 4 mostra um exemplo de subárvores e instruções associadas.

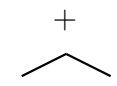
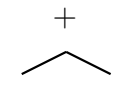
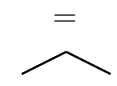
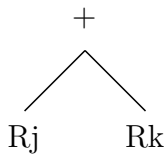
Instrução x86-64	**Subárvore**	**Custo**
add Rj, Ca		1
add Rj, Rk		1
mov Rj, Rk		1

Figura 4 – Padrões de subárvore. Fonte: adaptado de [4].

O funcionamento básico da abordagem consiste em iterar sobre a representação intermediária em formato de árvore e encontrar subárvores correspondentes da definição. Sempre que tal subárvore é encontrada ocorre a redução desse padrão para um único nó [4]. Por exemplo, caso o padrão



for encontrado na representação intermediária, a redução o fará ser representado como um único nó, no caso o nome do registrador onde a operação foi armazenada.

Quando um compilador faz bom uso de todas as instruções contidas no ISA (*Instruction Set Architecture*) do processador, o código gerado deixa de conter apenas instruções simples, chamadas "de uso geral", e passa a implementar extensões específicas que lidam melhor com determinadas situações [6, 7, 8]. O exemplo abaixo mostra o *assembly* para calcular a raiz quadrada de um valor *double* utilizando instruções AVX:

```
Sqrt proc
    push ebp
    mov ebp, esp

    vmovsd xmm0, real8 ptr [ebp + 8]
    vsqrtsd xmm1, xmm0, xmm0
    vmovsd real8 ptr [r], xmm1

    pop ebp
    ret
Sqrt endp
```

Outra operação comum que pode ser implementada através de extensões do conjunto de instruções é a soma de elementos inteiros contidos em dois vetores. O código abaixo mostra a função escrita em C/C++ e, em seguida, o código *assembly* equivalente implementado com funções AVX.

```
/* Código C/C++ */
void Sum(int *a, int *b, int *c) {
    for (int i = 0; i < VEC_SIZE; i++)
        c[i] = a[i] + b[i];
```

```
}

; Código assembly equivalente
AvxSum proc
    vmovdqa xmm0, xmmword ptr [rcx]
    vmovdqa xmm1, xmmword ptr [rdx]
    vpaddw xmm2, xmm0, xmm1
    vmovdqa xmmword ptr [r8], xmm2
    ret
AvxSum endp
```

Abaixo é apresentado o código *assembly* equivalente à função C/C++ anterior sem a utilização de instruções AVX. Nessa implementação, há um laço de repetição que itera sobre os elementos dos vetores, byte a byte, e realiza a soma.

```
Sum proc
    mov n, rcx
    mov rax, 0
    mov rbx, 0

_loop:
    cmp rax, [n]
    je _end

    mov cl, [rdx + rbx]
    add cl, [r8 + rbx]
    mov [r9 + rbx], cl
    add rbx, 1

    inc rax
    jmp _loop

_end:
    ret
Sum endp
```

Essa pesquisa tem por objetivo explorar a aplicação da técnica de reescrita de padrões [4] na substituição de instruções do conjunto de instruções de propósito geral por instruções mais eficientes, como MMX, SSE e AVX, com o objetivo de otimizar o desempenho de programas compilados [6, 7, 8]. Exemplos de códigos C/C++ e seus

respectivos *assembly*, com otimização e sem otimização, são discutidos no Capítulo 4, bem como detalhes do algoritmo de Alfred V. Aho [4], na Seção 4.1.

2 REPRESENTAÇÃO INTERMEDIÁRIA

Ao longo das várias etapas de tradução, o compilador adquire cada vez mais informação sobre o código que compila através das entradas passadas para cada uma de suas etapas. Esse fenômeno é descrito como "derivação de conhecimento" [1, 2, 3]. No entanto, um compilador não deve derivar conhecimento analisando o arquivo-fonte devido ao alto custo e complexidade computacional. Para resolver esse problema, foi proposta uma forma de expressar o código original, com exata equivalência, através de uma estrutura que se chama "representação intermediária".

A IR – *Intermediate Representation* é a estrutura responsável por conter toda a informação necessária para o sucesso da compilação, devendo ser suficientemente expressiva, mas ao mesmo tempo mantendo-se concisa [1, 2, 3]. Quase todas as etapas do compilador leem uma IR e produzem uma outra como resultado, sendo o *scanner* uma exceção.

É comum que alguns projetos de compiladores utilizem mais de uma IR para traduzir código, assim como diferentes formatos ou mesmo junções de múltiplas IRs. No entanto, um bom projeto deve se preocupar em empregar o formato adequado de IR para que o resultado seja o mais otimizado possível. Por exemplo, um programa que compila códigos C/C++ para *assembly* deve usar uma IR que expresse efetivamente informações sobre ponteiros, o que não seria necessário em tradutores de código Java [9].

2.1 As representações intermediárias como estruturas de dados

As árvores são estruturas de dados amplamente utilizadas na computação, sendo constituídas de nós em forma hierárquica rotulados "pais" e "filhos". Na compilação, a árvore como IR é chamada AST – *Abstract Syntax Tree* (ou em português, árvore sintática abstrata) sendo produzida como saída do analisador sintático, no *front-end* [1, 2, 3].

No Capítulo 1 é apresentado um exemplo de AST para expressão matemática simples. Porém, uma AST não é empregada apenas com essa finalidade, mas sim para representar trechos de código completos de uma linguagem alto nível. A Figura 5 mostra uma IR hierárquica para comando *while* em linguagem C/C++, onde o filho esquerdo da raiz corresponde à condição de repetição e o filho direito ao código executado pelo laço.

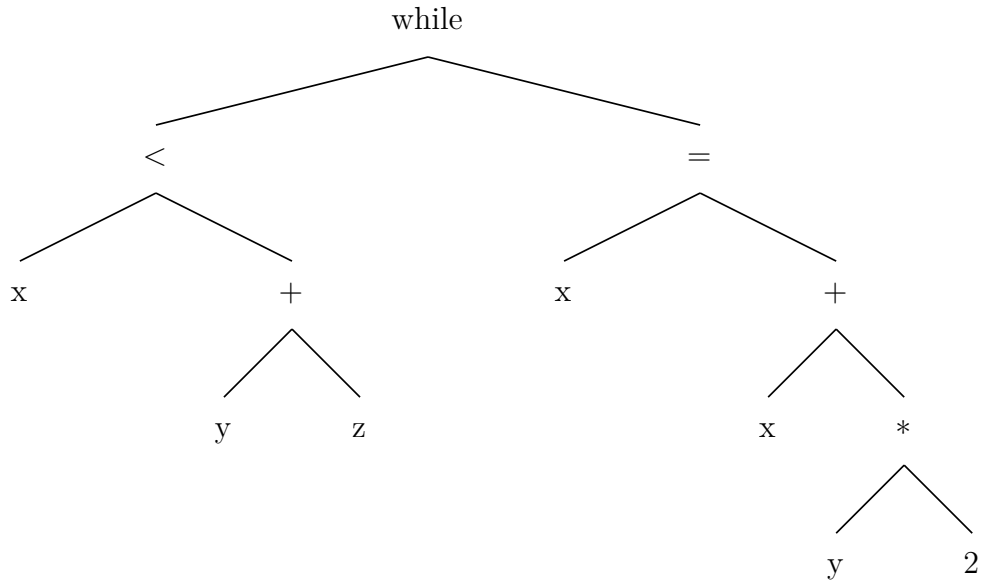


Figura 5 – Exemplo de AST alto nível para instrução *while* em linguagem C.

3 ARQUITETURA DO CONJUNTO DE INSTRUÇÕES

Posicionado entre o nível da microarquitetura e o nível do sistema operacional, o nível da arquitetura do conjunto de instruções é o responsável pelo interfaceamento entre o software o hardware [5, 10]. Em outras palavras, o conjunto de instruções diz aos desenvolvedores como interagir com determinado processador.

Assim como as IRs permitem que diferentes códigos-fonte sejam convertidos para uma mesma representação [1, 2, 3], o ISA – *Instruction Set Architecture* (o nome para arquitetura do conjunto de instruções, em inglês) assume papel semelhante. Isso permite que a CPU possa executar programas escritos em diferentes linguagens, desde que compiladas para o mesmo ISA [5, 10]. A Figura 6 mostra o nível ISA como intermediário entre software e hardware.

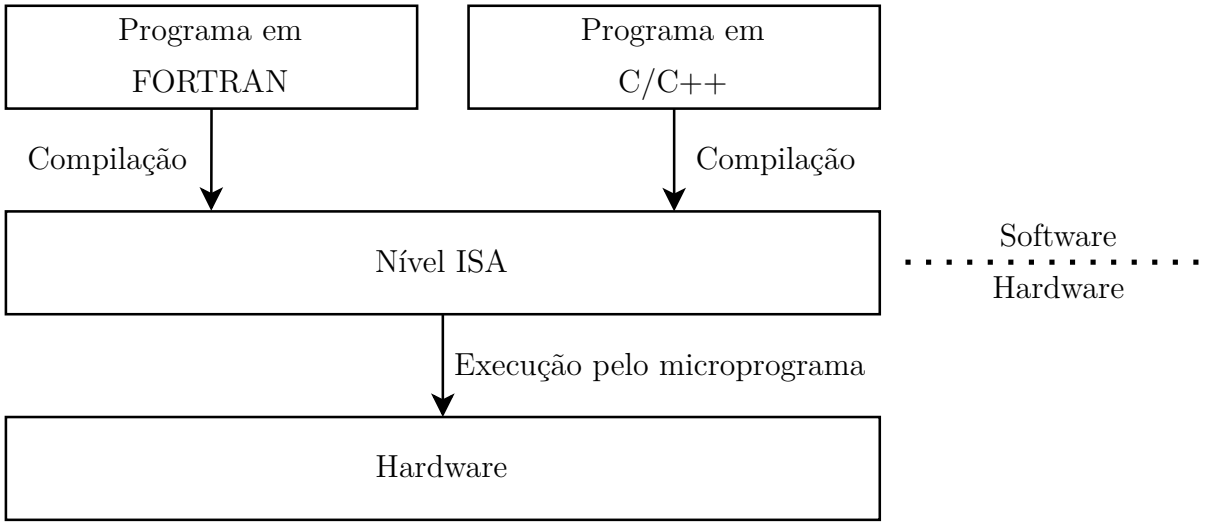


Figura 6 – Nível ISA como interface entre software e hardware. Fonte: [5].

Para um compilador é imprescindível o conhecimento de várias características do ISA alvo, como tipo de dados, quais registradores utilizar, quais são e como usar as instruções, e assim por diante [5, 10]. Dessa forma, os responsáveis pela criação do nível ISA disponibilizam manuais informando como usar as instruções corretamente, bem como informações sobre otimização de códigos pré-existentes [7, 11, 8].

É comum que, erroneamente, o ISA seja referenciado como "a arquitetura" [5, 10]. Por exemplo, a Intel e a AMD fabricam processadores que utilizam o mesmo conjunto de instruções x86-64 (uma extensão ou outra acaba variando), daí a razão para as mesmas aplicações funcionarem em ambos os processadores. No entanto, o nível da microarquitetura e a organização interna da CPU diferem. Isso é o que torna uma implementação

diferente da outra, afetando o desempenho do produto final.

Por fim, além das instruções de propósito geral do ISA, existem também aquelas mais especializadas que visam melhorar o desempenho de atividades específicas [6, 11]. São as chamadas "extensões do conjunto de instruções", e incluem MMX, SSE, AVX e muitas outras.

3.1 Extensão do conjunto de instruções

Com o passar do tempo, as aplicações se tornaram mais complexas e passaram a realizar operações mais intensivas na manipulação de informações. É o caso de processadores de imagens, processadores multimídia, aplicações 3D e demais. Para suprir tais demandas, novos conjuntos de instruções foram desenvolvidos e adicionados aos processadores. São as chamadas instruções SIMD.

SIMD – *Single Instruction, Multiple Data* é uma maneira de processar informações de forma mais eficiente através de paralelismo. Como o nome sugere, instruções SIMD são capazes de executar a mesma operação em múltiplos elementos ao mesmo tempo [6], sejam elas adições, subtrações, multiplicações, divisões, *shifts*, comparações e afins.

Para que o paralelismo ocorra, os valores precisam ser armazenados em registradores próprios das instruções SIMD. A Figura 7 mostra como valores de diferentes tamanhos são armazenados em registradores de 64-bits. É importante ressaltar que nem todas as extensões SIMD utilizam registradores de 64-bits, algumas delas implementam registradores de 128-bits ou 256-bits. No entanto, a forma como os valores são armazenados é equivalente.

O conjunto mais básico dentre os mencionados é o MMX, que adiciona oito novos registradores com capacidade de 64-bits, que vão de MM0 até MM7, destinados a executar operações SIMD com valores inteiros [6]. Isso significa que tais registradores podem armazenar um inteiro 64-bits, dois inteiros 32-bits, quatro inteiros 16-bits ou oito inteiros 8-bits.

Dentre as variadas instruções, há aquelas mais básicas como adição, subtração, multiplicação e divisão, e também há aquelas mais complexas, como *shifts* e conversões [6]. Por exemplo, a instrução pmaxub itera sobre os elementos contidos em determinados registradores e determina qual elemento é maior em cada um deles. A Figura 8 mostra o resultado da instrução pmaxub aplicada aos registradores mm0 e mm1.

Já o SSE – *Streaming SIMD Extensions*, por sua vez, possui funcionamento semelhante ao MMX e também adiciona oito novos registradores de 128-bits, que vão de XMM0 até XMM7. Sua utilidade difere do MMX ao incluir operações com valores de ponto flutuante de precisão única (*float*), ao invés de somente valores inteiros [6]. Poste-

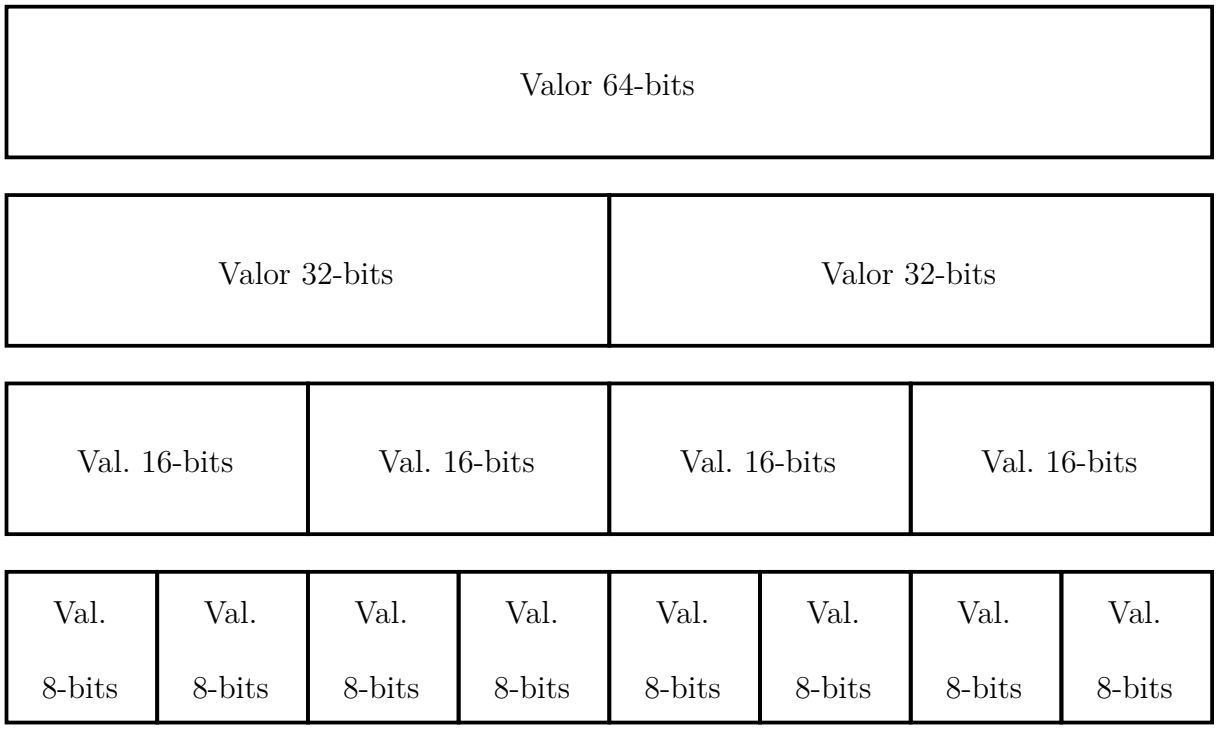


Figura 7 – Valores de diferentes tamanhos armazenados em registradores SIMD de 64-bits. Fonte: [6].

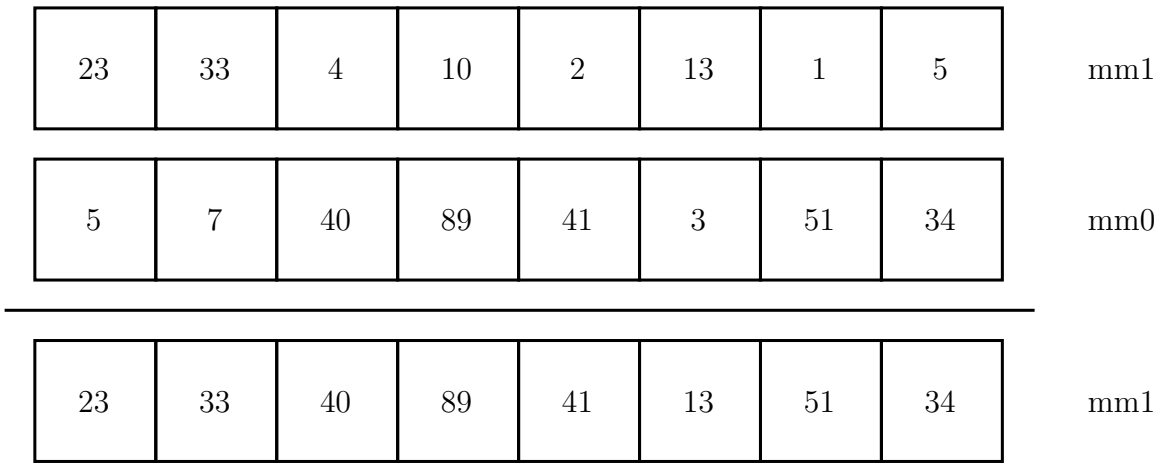


Figura 8 – Resultado da operação pmaxub sobre os registradores mm0 e mm1. Fonte: [6].

riormente, o SSE2 introduziu a capacidade de operar sobre valores de ponto flutuante de precisão dupla (*double*).

A principal vantagem do SSE sobre o MMX é que, quando utilizado para processar valores inteiros de 8-bits, suporta até 16 elementos, ao invés de apenas oito [6].

Por fim, a tecnologia AVX – *Advanced Vector Extensions*, de forma similar às anteriores, adiciona oito novos registradores de 256-bits, nomeados de YMM0 até YMM7

e estende as capacidades do SSE [6]. Porém, o AVX inova ao introduzir a sintaxe de três operandos, ao invés da conhecida sintaxe de dois operandos. O sucessor do AVX, o AVX-512, expandiu ainda mais suas capacidades ao introduzir a possibilidade de operar com valores de até 512-bits [6].

Essas são só algumas das tecnologias empregadas aos processadores atuais, porém há outras inúmeras que não serão abordadas, incluindo AES, XFR, EVP e demais.

4 REESCRITA DE PADRÕES

Códigos podem ser escritos de diferentes maneiras e ainda assim possuírem resultado equivalente entre eles. Porém, determinadas maneiras de programar são mais eficientes do que outras. O mesmo ocorre com os códigos *assembly* x86-64 gerados por um compilador.

Um código simples escrito em C/C++ para realizar a soma de elementos contidos em dois vetores e, posteriormente, armazená-la em um terceiro, poderia ser escrito da seguinte maneira:

```
void Sum(int *a, int *b, int *c) {
    for (int i = 0; i < VEC_SIZE; i++)
        c[i] = a[i] + b[i];
}
```

Se submetido à um compilador, o trecho acima poderia ser traduzido com exatidão para código *assembly* x86-64 utilizando apenas instruções de propósito geral. A saída seria a seguinte:

```
Sum proc
    mov n, rcx
    mov rax, 0
    mov rbx, 0

_loop:
    cmp rax, [n]
    je _end

    mov cl, [rdx + rbx]
    add cl, [r8 + rbx]
    mov [r9 + rbx], cl
    add rbx, 1

    inc rax
    jmp _loop

_end:
    ret
```

```
Sum endp
```

Em contrapartida, um compilador otimizante poderia empregar instruções da extensão AVX para realizar a mesma operação citada no código C/C++ acima. Como mencionado anteriormente, a extensão AVX provê conjunto de instruções focadas em manipulações de valores contidos em vetores. O mesmo código utilizando AVX poderia ser escrito da seguinte maneira:

```
AvxSum proc
    vmovdqa xmm0, xmmword ptr [rcx]
    vmovdqa xmm1, xmmword ptr [rdx]
    vpaddw xmm2, xmm0, xmm1
    vmovdqa xmmword ptr [r8], xmm2
    ret
AvxSum endp
```

Analisando as duas implementações lado a lado é possível notar como o método AVX é mais conciso e extremamente mais eficiente do que o código anterior, pois não é necessária a manipulação de endereços de memória manualmente, ou iterações em um laço de repetição.

4.1 Algoritmo para reescrita de padrões de árvore

5 TRABALHOS FUTUROS

6 CONCLUSÃO

REFERÊNCIAS

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi e Jeffrey D. Ullman. *Compiladores: Princípios, Técnicas e Ferramentas*. [S.l.]: Pearson, 2008.

[2] Keith D. Cooper e Linda Torczon. *Construindo Compiladores*. [S.l.]: Elsevier, 2014.

[3] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. [S.l.]: Morgan Kaufmann, 1997.

[4] Alfred V. Aho , Mahadevan Ganapathi e Steven W. K. Tjiang. Code Generation Using Tree Matching and Dynamic Programming. *ACM – Association for Computing Machinery*, 1989.

[5] Andrew S. Tanenbaum e Todd Austin. *Organização Estruturada de Computadores*. [S.l.]: Pearson, 2013.

[6] Daniel Kusswurm. *Modern x86 Assembly Language Programming – 32-bit, 64-bit, SSE and AVX*. [S.l.]: Apress, 2014.

[7] INTEL. *Intel 64 and IA-32 Architectures Software Developer Manuals*. 2024. Acesso em 23 de abril de 2024. Disponível em: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>.

[8] AMD. *AMD64 Architecture Programmer's Manual Volumes 1–5*. 2024. Acesso em 12 de maio de 2024. Disponível em: <https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf>.

[9] Andrew W. Appel. *Modern Compiler Implementation in Java*. [S.l.]: Cambridge University Press, 2002.

[10] William Stallings. *Arquitetura e Organização de Computadores*. [S.l.]: Pearson Education do Brasil Ltda., 2018.

[11] INTEL. *Intel 64 and IA-32 Architectures Optimization Reference Manual Volume 1*. 2024. Acesso em 9 de maio de 2024. Disponível em: <https://www.intel.com/content/www/us/en/content-details/821612/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html>.

Apêndices

APÊNDICE A – QUISQUE LIBERO JUSTO

Quisque facilisis auctor sapien. Pellentesque gravida hendrerit lectus. Mauris rutrum sodales sapien. Fusce hendrerit sem vel lorem. Integer pellentesque massa vel augue. Integer elit tortor, feugiat quis, sagittis et, ornare non, lacus. Vestibulum posuere pellentesque eros. Quisque venenatis ipsum dictum nulla. Aliquam quis quam non metus eleifend interdum. Nam eget sapien ac mauris malesuada adipiscing. Etiam eleifend neque sed quam. Nulla facilisi. Proin a ligula. Sed id dui eu nibh egestas tincidunt. Suspendisse arcu.

Anexos

ANEXO A – MORBI ULTRICES RUTRUM LOREM.

Sed mattis, erat sit amet gravida malesuada, elit augue egestas diam, tempus scelerisque nunc nisl vitae libero. Sed consequat feugiat massa. Nunc porta, eros in eleifend varius, erat leo rutrum dui, non convallis lectus orci ut nibh. Sed lorem massa, nonummy quis, egestas id, condimentum at, nisl. Maecenas at nibh. Aliquam et augue at nunc pellentesque ullamcorper. Duis nisl nibh, laoreet suscipit, convallis ut, rutrum id, enim. Phasellus odio. Nulla nulla elit, molestie non, scelerisque at, vestibulum eu, nulla. Ut odio nisl, facilisis id, mollis et, scelerisque nec, enim. Aenean sem leo, pellentesque sit amet, scelerisque sit amet, vehicula pellentesque, sapien.

Publicações principais do trabalho.

Publicações complementares.