Universidade de Brasília
Instituto de Exatas
Departamento de Ciência da Computação

Relatório do Laboratório 1
Assembly RISC-V

CIC0099 - Organização e Arquitetura de Computadores

Turma 02 - Prof. Dr. Ricardo Pezzuol Jacobi

Grupo B4

Henrique Morcelles Salum	232003008
Athos Calixto Muniz	211068261
Gabriela Fernanda Rodrigues Costa	180120859
Cauê Araújo Euzebio	211028195
Lucas Silva Nóbrega	180035096

Sumário

1 Introdução

O laboratório 1 de OAC é separado em três grandes experimentos - esses, por sua vez, subdivididos em diversas tarefas - e visa familiarizar o aluno ao Assembly RISC-V e ao *RISC-V Assembler and Runtime Simulator* (RARS). Notadamente, o primeiro experimento concentra-se no RARS; os outros dois, no RISC-V.

Mais detalhadamente, no primeiro experimento, analisa-se o desempenho de um código de ordenação fornecido; o segundo experimento requer o uso de um compilador cruzado de C para RISC-V, com o objetivo de gerar e corrigir códigos em Assembly; o terceiro experimento propõe o desenvolvimento de um algoritmo para a transformada discreta de Fourier em RISC-V.

2 Métodos e Análise

2.1 Primeiro Experimento

É fornecido o programa, em Assembly, `sort.s`, de ordenamento de um vetor. Considere um processador RISC-V com frequência de clock 50MHz e CPI=1 e os vetores de entrada de n elementos $V_0[n] = \{1,\ 2,\ \ldots,\ n\}$, já ordenado, e $V_1[n] = \{n,\ n-1,\ \ldots,\ 1\}$, inversamente ordenado.

2.1.1 Tarefa I

Enunciado

Para o procedimento `sort`, escreva as equações dos tempos de execução, $t_0[n]$ e $t_1[n]$, em função de n.

Primeiro, devemos lembrar que o tempo de execução pode ser calculado pela seguinte equação:

$$t_{exec} = \frac{I \cdot CPI}{f_{clock}} \tag{1}$$

Note que já temos CPI e f_{clock}, só resta descobrir I (a quantidade de instruções executadas).

O RARS pode ser utilizado para esse fim: no menu, escolhemos Tools > Instruction Counter, conectamos essa ferramenta ao código, montamos, adicionamos pontos de quebra na chamada do subprocedimento `SORT` e na linha seguinte e executamos o programa. Realizando esse passo a passo para vários vetores ordenados e inversamente ordenados de

diferentes tamanhos n, observou-se que

$$I_0[n] = \begin{cases} 18, & \text{se } n = 0 \\ 10n + 13, & \text{se } n \geq 1 \end{cases}$$

e

$$I_1[n] = 9n^2 - 4n + 18, \text{ se } n \geq 0$$

De acordo com a Equação (1), temos que

$$t_0[n] = \begin{cases} 0,36, & \text{se } n = 0 \\ 0,2n + 0,26, & \text{se } n \geq 1 \end{cases} \quad [\mu s] \tag{2}$$

e

$$t_1[n] = 0,18n^2 - 0,08n + 0,36, \text{ se } n \geq 0 \quad [\mu s] \tag{3}$$

2.1.2 Tarefa II

Enunciado

Para $n = \{10,\ 20,\ 30,\ 40,\ 50,\ 60,\ 70,\ 80,\ 90,\ 100\}$, plote (em escala!) as duas curvas, $t_0[n]$ e $t_1[n]$, em um mesmo gráfico $n \times t$.

Utilizando as Equações (2) e (3), fizemos o programa a seguir, em Fortran, que calcula $t_0[n]$ e $t_1[n]$ para os n de interesse e escreve esses dados em um arquivo.

```
1  program exec_time
2      implicit none
3      integer :: iunit, i
4
5      open(newunit=iunit,file='oac_t0.dat',status='replace')
6      do i = 1, 10
7          write(iunit, *) i*10, t0(i*10)
8      end do
9      close(iunit)
10
11     open(newunit=iunit,file='oac_t1.dat',status='replace')
12     do i = 1, 10
13         write(iunit, *) i*10, t1(i*10)
14     end do
15     close(iunit)
16 contains
17     double precision function t0(n)
18         integer, intent(in) :: n
19         t0 = 0.26d0 + 0.2d0 * n
20     end function
21     double precision function t1(n)
22         integer, intent(in) :: n
23         t1 = 0.18d0*n*n - 0.08d0*n + 0.36d0
24     end function
25 end program exec_time
```

Código 1: Programa em Fortran para calcular $t_0[n]$ e $t_1[n]$ para os argumentos pedidos

A partir dos dados obtidos, plotamos o gráfico a seguir com gnuplot.

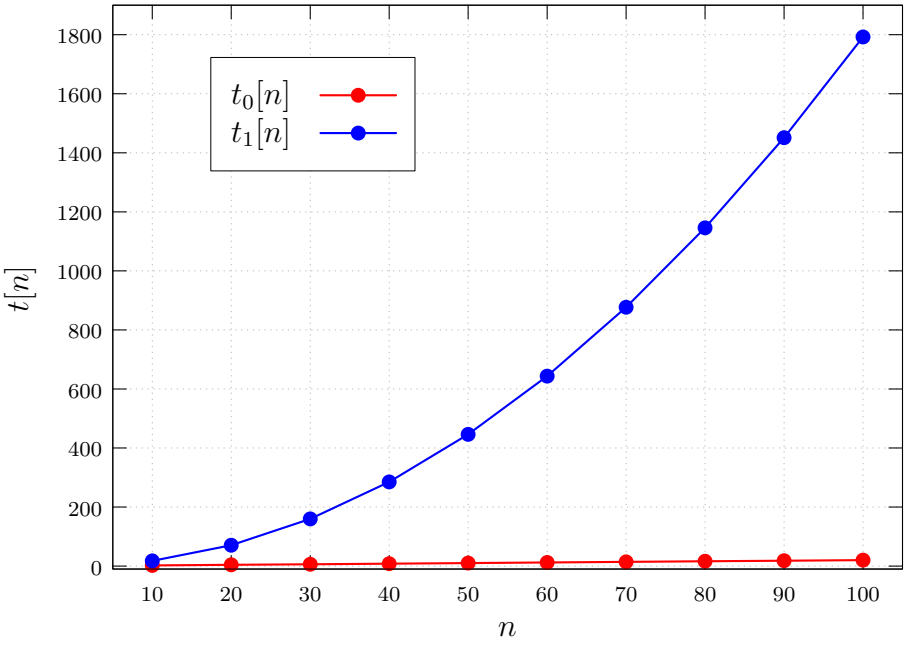

Figura 1: Comparação entre $t_1[n]$ e $t_0[n]$

Perceba que o gráfico de $t_1[n]$ se distancia tanto do de $t_0[n]$ conforme n cresce que o segundo chega a parecer constante em $t = 0$. Esse resultado é coerente, pois $t_1[n]$ é uma função de segundo grau e $t_0[n]$, de primeiro. De fato, no limite, temos

$$\lim_{n\to\infty} \frac{t_0[n]}{t_1[n]} = \lim_{n\to\infty} \frac{0,2n + 0,26}{0,18n^2 - 0,08n + 0,36} = 0$$

2.2 Segundo Experimento

O segundo experimento tem como objetivo compreender a compilação cruzada e a geração de código Assembly a partir de C, com foco em RISC-V. Utilizamos o site Compiler Explorer para compilar os códigos em C com diferentes diretivas.

2.2.1 Tarefa I

Enunciado

Dado o programa sortc.c, compile-o com a diretiva -O0 e obtenha o arquivo sortc.s. Indique as modificações necessárias no código Assembly gerado para que possa ser executado corretamente no RARS.

A primeira vista, percebe-se que o compilador não utiliza as diretivas `.data` e `.text`, que são necessárias para a montagem no RARS. Além disso, no simulador, as intruções começam a ser executadas a partir do topo da região `.text`, dessa forma, seria necessário deslocar o procedimento `main` ou chamá-lo no início dessa seção, além de incluir uma *syscall* com `a7=10` para terminar o programa. Ademais, sem incluir a biblioteca padrão

do C, funções como `printf` e `putchar` não podem ser utilizadas; seguindo a dica fornecida no roteiro, utilizamos a função `show`, definida no programa `sort.s`.

Com as correções acima, o programa já pode ser montado pelo RARS, porém, ainda há um erro em tempo de execução: `Error in : Instruction load access error`. O erro provavelmente se deve à má utilização do par `lui` + `addi` para acessar a memória, pois sua substituição por `la` resolveu o problema, tornando o programa totalmente funcional.

2.2.2 Tarefa II

Enunciado

Compile o programa `sortc_mod.c` e, com a ajuda do RARS, monte uma tabela comparativa com o número total de instruções executadas pelo programa todo, e o tamanho em bytes dos códigos em linguagem de máquina gerados para cada diretiva de otimização da compilação {`-O0`, `-O3`, `-Os`}. Compare ainda com os resultados obtidos no item 1.1) com o programa `sort.s`, que foi implementado diretamente em Assembly. Analise os resultados obtidos usando o mesmo vetor de entrada.

Diretriz	**Instruções executadas**	**Tamanho (bytes)**
-O0	10.081	496
-Os	3.905	312
-O3	2.181	268
`sort.s`	3.746	296

Tabela 1: Instruções executadas para cada diretiva de compilação

Os dados de desempenho para as diferentes diretrizes de compilação fornecem observações interessantes sobre as estratégias de otimização do GCC. Ao comparar os resultados é possível observar relações entre eficiência e tamanho do código.

A diretiva -O0, que compila sem otimização, é a que gera um código com mais instruções executadas (10.081) e o maior tamanho em bytes (496).

A *flag* -Os, que prioriza a redução do tamanho do código, reduziu o número de instruções (3.905) e o tamanho do código (312 bytes) significativamente em relação à -O0, mas não conseguiu superar em nenhum dos quesitos as otimizações de -O3 ou o código escrito direto em Assembly `sort.s`.

A *flag* -O3, que representa o nível máximo de otimização, mostrou resultados mais eficientes, executando menos instruções que todas as outras opções (2181) e com o menor tamanho (268 bytes). Esse desempenho é alcançado com técnicas como eliminação de código redundante e reordenação de instruções, e evidencia a eficácia da otimização do compilador em gerar código enxuto.

O código `sort.s`, escrito manualmente em Assembly, apresentou 3746 instruções e um tamanho de 296 bytes, ficando entre -Os e -O3. Este resultado demonstra que programar

em Assembly pode ser eficiente, mas, para bater de frente com compiladores modernos e otimizados, é necessário conhecimento e grande esforço.

2.2.3 Tarefa III

Enunciado

Pesquise na internet e explique as diferenças entre as otimizações `-O0`, `-O1`, `-O2`, `-O3` e `-Os`.

- -O0 (Sem otimização): Configuração padrão do compilador, quando nenhuma otimização é especificada. Ao compilar com -O0, o código gerado é praticamente uma tradução direta do código fonte para assembly, o que facilita a depuração. A estrutura do código original é preservada, mantendo todas as variáveis e evitando rearranjos ou remoções de código, para que possa ser feito um mapeamento um-para-um;

- -O1: Otimização com o objetivo de reduzir tamanho do código e tempo de execução sem aumentar o tempo de compilação. Entre as otimizações estão a remoção de código que nunca são executados (código morto), e simplificação de expressões;

- -O2: Otimiza mais ainda, incluindo todas as otimizações de -O1 e ativando outras como inline de funções pequenas para eliminar overhead de chamadas e análise de fluxo de dados. Aumenta tanto o tempo de compilação quanto o desempenho do código gerado;

- -O3: Todas as otimizações de -O2 são ativadas, além de outras transformações que maximizam o desempenho. Esta diretriz aplica técnicas como inlining de funções, unrolling de loops, onde o compilador replica sistematicamente o corpo dos loops para reduzir significativamente o overhead de controle e vetorização. Esta abordagem agressiva apresenta entretanto um *trade-off* significativo, o código gerado pode ter um tempo de compilação maior, e um aumento em tamanho que acaba impactando negativamente a utilização da memória cache;

- -Os: Otimização projetada para sistemas embarcados ou com recursos limitados de memória. -Os aplica otimizações de -O2 que não aumentam o tamanho do código e desativa as que tendem a expandir o código. São utilizadas técnicas como eliminação de código não utilizado, seleção de instruções mais curtas e reorganização de código, visando gerar um executável menor.

2.3 Terceiro Experimento

O terceiro experimento consiste em implementar, em Assembly RISC-V, a Transformada Discreta de Fourier (DFT), que realiza uma mudança de base entre duas bases ortonormais de dimensão N: a base temporal (amostras) composta por N impulsos unitários deslocados $\delta[n-m],\ m \in \{0,\ldots,N-1\}$ para a base frequencial composta por N exponenciais complexas $e^{-i\frac{2\pi}{N}kn},\ k \in \{0,\ldots,N-1\}$. A DFT é dada pela seguinte fórmula:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}kn} \tag{4}$$

em que $x[n]$ é o n-ésimo termo do sinal (sequência) x no domínio do tempo, $X[k]$ é k-ésimo termo desse sinal no domínio da frequência e N é a largura do sinal.

2.3.1 Tarefa I

Enunciado

Escreva um procedimento que receba um ângulo em radianos θ (em fa0) e retorne $\cos\theta$ (em fa0) e $\sin\theta$ (em fa1).

Diferentemente das linguagens de alto nível, não existem funções trigonométricas nativas em Assembly. Consequentemente, é necessário utilizar séries de Taylor, séries polinomiais capazes de aproximar funções mais complicadas. No limite, com um somatório de infinitos termos, essas séries se igualam à função original, pelo menos em algum intervalo. As séries de Maclaurin para o cosseno e o seno são as seguintes:

$$\cos\theta = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!}\theta^{2n} \qquad \sin\theta = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}\theta^{2n+1}$$

Ingenuamente, poderíamos pensar em implementar o procedimento pedido por meio de dois loops separados, uma função que calcula o fatorial e outra que calcula exponenciais. Essa abordagem, porém, é excessivamente cara. A fim de otimizar o algoritmo é interessante perceber a seguinte relação:

$$\begin{aligned}
e^{i\theta} &= \sum_{n=0}^{\infty} \frac{(i\theta)^n}{n!} = \frac{(i\theta)^0}{0!} + \frac{(i\theta)^1}{1!} + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \ldots \\
&= 1 + i\theta - \frac{\theta^2}{2!} - i\frac{\theta^3}{3!} + \frac{\theta^4}{4!} + i\frac{\theta^5}{5!} - \ldots \\
&= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \ldots\right) + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \ldots\right) \\
&= \left(\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!}\theta^{2n}\right) + i\left(\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!}\theta^{2n+1}\right) \\
&= \cos\theta + i\sin\theta
\end{aligned}$$

Ou seja, além de termos demonstrado a igualdade de Euler, percebemos que se pode utilizar um único loop para calcular simultaneamente seno e cosseno (o loop referente à série da exponencial complexa), basta definir, pela paridade do índice do somatório, em qual registrador vamos acumular o resultado em cada iteração (`fa0` para índices pares e `fa1` para índices ímpares).

Teoricamente, a implementação do algoritmo `SINCOS` conforme descrito acima (pela série de Taylor) seria suficiente. No entanto, na prática, apesar da série de Taylor convergir para qualquer ângulo, a aproximação por um número finito de termos é imprecisa para ângulos grandes e não basta aumentar o número de iterações (de termos da série) pois o cálculo de potências elevadas pode causar *overflow*. Dessa forma, a fim de melhorar as aproximações, modulamos o ângulo de forma que ele fique sempre no intervalo $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. De acordo com a figura a seguir, isso preserva o seno, mas altera o sinal do cosseno; precisamos, portanto, criar uma *flag* para inverter o cosseno.

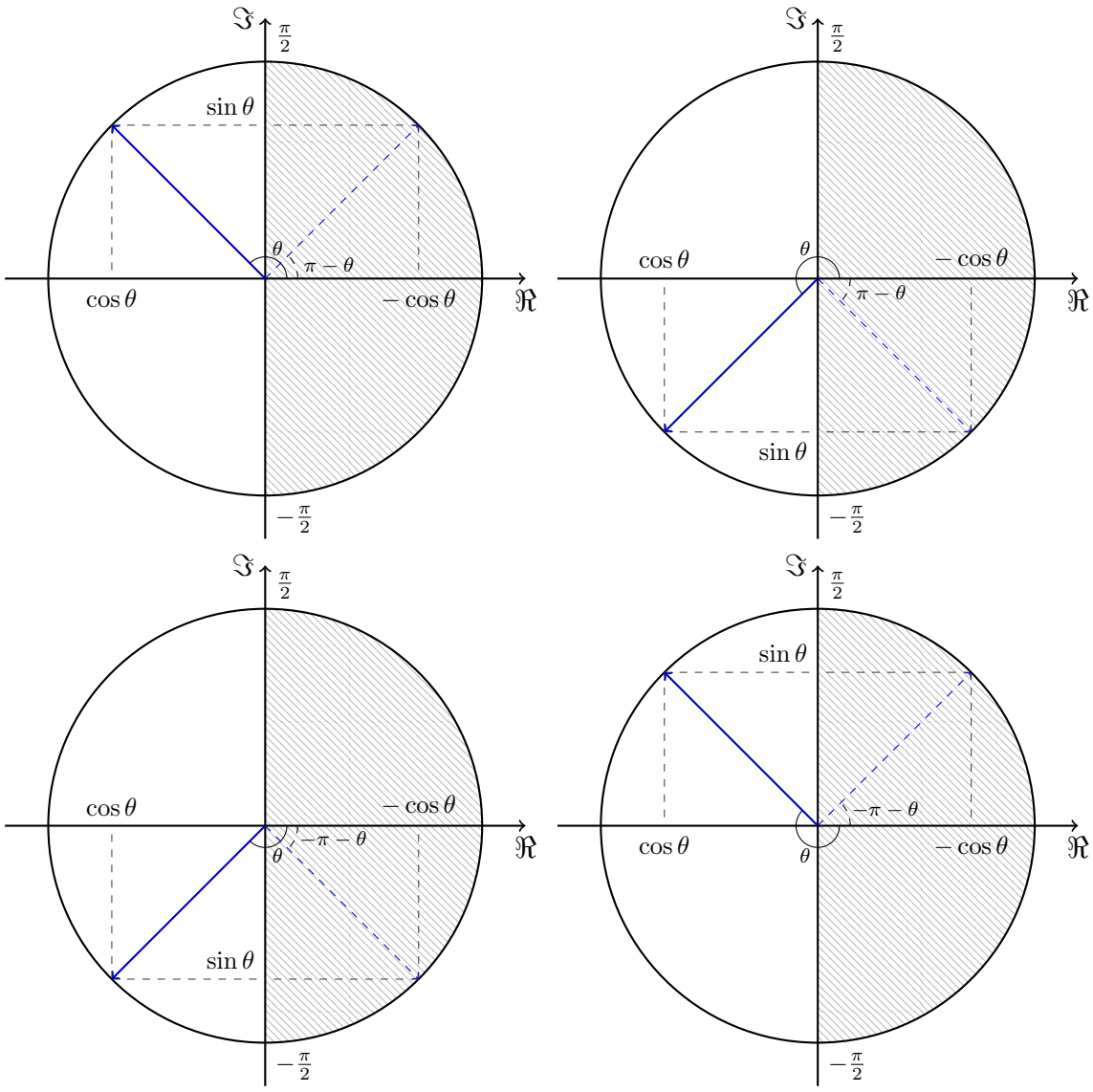

Figura 2: Representação gráfica de quatro casos de interesse da normalização realizada

2.3.2 Tarefa II

Enunciado

Escreva um procedimento em Assembly RISC-V com a seguinte definição

```
void DFT(float *x, float *X_real, float *X_imag, int N)
```

que, dado o endereço do vetor $x[n]$ de floats (em `a0`) de tamanho N na memória, os endereços dos espaços reservados para o vetor complexo $X[k]$ (parte real e parte imaginária) (em `a1` e `a2`) e o número de pontos N (em `a3`), calcule a DFT de N pontos de $x[n]$ e coloque o resultado no espaço alocado para $X_{real}[k]$ e $X_{imag}[k]$.

Tendo em mãos o procedimento da última tarefa, a implementação da DFT torna-se conceitualmente simples. O procedimento consiste em criar um loop que invoca a função `SINCOS` - que retorna as componentes real e imaginária de $e^{-i\frac{2\pi}{N}kn}$ nos registradores `fa0` e `fa1`, respectivamente - multiplicar esses valores por $x[n]$ e armazenar os resultados nas posições de memória correspondentes. A complexidade reside apenas no gerenciamento operacional dos registradores, pois, como temos procedimentos aninhados, é necessário utilizar a *stack*.

2.3.3 Tarefa III

Enunciado

Escreva um programa `main` que defina no `.data` o vetor $x[n]$, o espaço para o vetor $X[K]$, o valor de N e chame o procedimento DFT.

```
.data
N:      .word   8
x:      .float  1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0
X_real: .float  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
X_imag: .float  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
.text
...
jal DFT
...
```

A seguir, apresente no console a saída dos N pontos no formato:

```
x[n]    X[k]
1.0     8.0+0.0i
1.0     0.0+0.0i
1.0     0.0+0.0i
1.0     0.0+0.0i
1.0     0.0+0.0i
1.0     0.0+0.0i
1.0     0.0+0.0i
1.0     0.0+0.0i
```

Essa `main` consiste em uma chamada à função DFT e um loop para exibir *floats* e caracteres no console (além de uma pequena lógica condicional para não exibir o '+'

quando a parte imaginária é negativa). O resultado, exibido a seguir, não ficou igual ao do exemplo do enunciado, em que todos os termos de $X[k]$, com excessão do primeiro, são 0.0, mas acreditamos que isso se deva a dois fatores: à imperfeição da aproximação por polinômios de Taylor e à precisão de máquina. De fato, como os maiores resultados foram da ordem de 10^{-6}, essas são justificativas razoáveis.

x[n]	X[k]
1.0	8.0+0.0i
1.0	-7.748604E-7+0.0i
1.0	-6.727769E-7+2.3841858E-7i
1.0	3.5762787E-7-4.7683716E-7i
1.0	0.0-7.1525574E-7i
1.0	-2.7418137E-6-1.1324883E-6i
1.0	-1.6589261E-6+1.9073486E-6i
1.0	-2.0861626E-6-1.1920929E-7i

2.3.4 Tarefa IV

Enunciado

Calcule a DFT dos seguintes vetores $x[n]$, com $N = 8$

```
x1: .float  1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
x2: .float  1.0, 0.7071, 0.0, -0.7071, -1.0, -0.7071, 0.0, 0.7071
x3: .float  0.0, 0.7071. 1.0,  0.7071, 0.0, -0.7071, -1.0, -0.7071
x4: .float  1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0
```

Utilizando o código desenvolvido nas tarefas anteriores, calculamos a DFT de cada um dos sinais. Os resultados estão exibidos a seguir, de acordo com a ordem do enunciado.

x_1

x[n]	X[k]
1.0	1.0+0.0i
0.0	1.0+0.0i
0.0	1.0+0.0i
0.0	1.0+0.0i
0.0	1.0+0.0i
0.0	1.0+0.0i
0.0	1.0+0.0i
0.0	1.0+0.0i

x_3

x[n]	X[k]
0.0	-5.9604645E-8+0.0i
0.7071	5.9604645E-8+3.9999797i
1.0	-6.668487E-7-7.1525574E-7i
0.7071	-1.4305115E-6-1.8388033E-5i
0.0	0.0+8.429289E-7i
-0.7071	9.23872E-7+1.9133091E-5i
-1.0	1.4437442E-6-9.536743E-7i
-0.7071	3.784895E-6-3.999981i

x_2

x[n]	X[k]
1.0	-5.9604645E-8+0.0i
0.7071	3.999981+3.2782555E-7i
0.0	9.760695E-7+0.0i
-0.7071	1.963973E-5-7.4505806E-7i
-1.0	0.0-5.057573E-7i
-0.7071	1.8894672E-5+2.9802322E-8i
0.0	-8.34465E-7+0.0i
0.7071	3.9999797+2.592802E-6i

x_4

x[n]	X[k]
1.0	4.0+0.0i
1.0	0.9999994+2.4142132i
1.0	-1.02313E-6-2.3841858E-7i
1.0	0.9999994+0.4142136i
0.0	0.0+2.3841858E-7i
0.0	0.999999-0.4142139i
0.0	9.851078E-9+4.7683716E-7i
0.0	1.0000013-2.4142137i

Novamente, percebe-se um pequeno erro nos resultados: alguns números são da ordem

de 10^{-7}, outros, extremamente próximos de 1 ou 4, que eram os valores esperados. Justificamos essas imprecisões da mesma forma que as da questão anterior: erros na aproximação por polinômios e na precisão de máquina.

2.3.5 Tarefa V

Enunciado

Para os sinais $x[n]$ abaixo

a) $N = 8$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0\}$

b) $N = 12$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

c) $N = 16$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

d) $N = 20$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

e) $N = 24$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

f) $N = 28$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

g) $N = 32$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

h) $N = 36$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

i) $N = 40$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

j) $N = 44$, $x[n] = \{1.0,\ 1.0,\ 1.0,\ 0.0,\ 0.0,\ 0.0,\ 0.0,\ \ldots,\ 0.0\}$

onde ... são todos valores 0.0. Para cada item, meça o tempo de execução do procedimento DFT e calcule a frequência do processador RISC-V Uniciclo simulado pelo RARS. Finalmente, faça um gráfico $t_{exec} \times N$ em escala. Quais conclusões podemos tirar dessa análise?

Utilizando os registradores de status e controle do RARS, medimos o tempo de execução (t_{exec}) e a quantidade de instruções (I) do programa para cada caso. A frequência foi calculada pela fórmula

$$f_{clock} = \frac{I \cdot CPI}{t_{exec}} \overset{CPI=1}{=} \frac{I}{t_{exec}}$$

A seguir, os resultados:

a) $N = 8 \rightarrow t_{exec} = 48ms$, $I = 15.606$, $f = 325kHz$

b) $N = 12 \rightarrow t_{exec} = 84ms$, $I = 34.999$, $f = 416kHz$

c) $N = 16 \rightarrow t_{exec} = 144ms$, $I = 62.137$, $f = 431kHz$

d) $N = 20 \rightarrow t_{exec} = 235ms$, $I = 96.981$, $f = 412kHz$

e) $N = 24 \rightarrow t_{exec} = 343ms$, $I = 139.615$, $f = 407kHz$

f) $N = 28 \rightarrow t_{exec} = 483ms$, $I = 189.955$, $f = 393kHz$

g) $N = 32 \rightarrow t_{exec} = 595ms,\ I = 248.028,\ f = 416kHz$

h) $N = 36 \rightarrow t_{exec} = 769ms,\ I = 313.777,\ f = 408kHz$

i) $N = 40 \rightarrow t_{exec} = 949ms,\ I = 387.373,\ f = 408kHz$

j) $N = 44 \rightarrow t_{exec} = 1153ms,\ I = 468.639,\ f = 406kHz$

Com esses dados, podemos fazer o gráfico solicitado. Novamente, utilizamos o gnuplot e plotamos o seguinte gráfico:

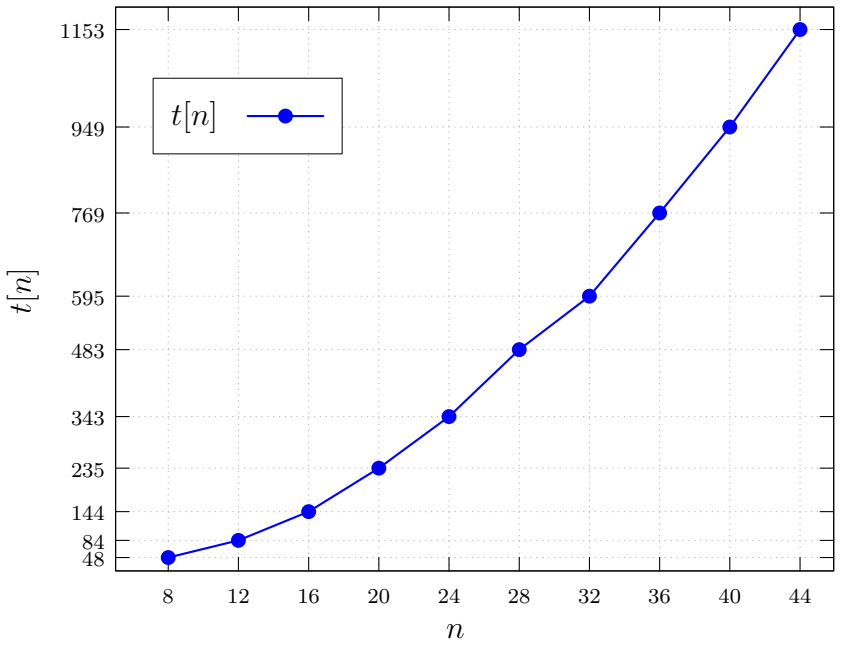

Figura 3: Tempo de execução em função do tamanho da entrada

Note que o gráfico parece representar uma função quadrática. De fato isso é coerente: fazendo a análise assintótica do algoritmo implementado, ele é composto por duas estruturas de repetição aninhadas que dependem do tamanho da entrada (N), ou seja, o algoritmo é $\Theta(N^2)$. A chamada a `SINCOS` dentro desses loops pode causar confusão, pois há estrutura de repetição nessa função, mas note: a quantidade de iterações dessa estrutura é fixa, são sempre vinte e uma iterações.

3 Conclusão

Nesse laboratório, tivemos intenso contato com o RARS e suas ferramentas, desenvolvemos códigos complexos em Assembly e entendemos melhor a compilação de linguagens de alto nível. Além disso, fizemos análises de desempenho e todas tiveram resultados coerentes.

A experiência obtida neste laboratório, bem como os resultados alcançados e a pesquisa realizada, conduzem à reflexão sobre aspectos cruciais da programação, tais como: a necessidade de compiladores eficientes para que a abstração proporcionada por linguagens de alto nível não sejam demasiado caras; o custo associado ao uso de números de

ponto flutuante — que exigem mais instruções para operações aparentemente simples; e a importância de conhecer o *hardware* utilizado para compreender adequadamente o tempo de execução e o desempenho de programas.

Todos os códigos desenvolvidos ou gerados (via compilação cruzada) para a realização desse laboratório foram enviados compactados em um arquivo `.zip`. Para acessar o vídeo solicitado, clique aqui.