Laboratório de Organização e Arquitetura de Computadores

Relatório referente ao experimento nº 1

Eric do Vale de Castro - 15/0124236
Guilherme de Castro Ribeiro - 14/0142151
Rodrigo Matias Xavier - 15/0147431
João Pedro Ramos do Amaral - 14/0056394
Victor Neves Martorelli - 16/0085012

20 de Maio de 2019

1 Objetivos

Familiarização com o ambiente de simulação RARS e o compilador GCC para o Assembly RISC-V. Ao final, pretende-se estar apto a construir algoritmos e analisar o desempenho dos códigos em linguagem Assembly RISC-V.

2 Introdução

Arquitetura de computador refere-se aos atributos de um sistema visíveis a um programador ou, em outras palavras, aqueles atributos que possuem um impacto direto sobre a execução lógica de um programa.

Alguns exemplos de atributos arquiteturais incluem o conjunto de instruções, o número de bits usados para representar diversos tipos de dados (números, caracteres), mecanismos de E/S e técnicas para endereçamento de memória. Uma questão de projeto arquitetural exemplifica-se na decisão se um computador terá uma instrução de multiplicação.

Quanto a performance, Patterson [2] ainda explicita que em um programa, um conjunto de combinações podem interferir nela, dentre elas há um grande destaque ao conjunto de instruções definido.

Nesse contexto, a arquitetura RISC, conjunto reduzido de instruções, é um grande avanço da tendência histórica na arquitetura de processadores [2]. Uma análise da arquitetura RISC traz à tona questões importantes da organização e da arquitetura de computadores. Embora os sistemas RISC tenham sido definidos e projetados de muitas maneiras diferentes e por grupos diferentes, o autor destaca três elementos principais compartilhados pela maioria dos projetos:

1. Um conjunto de instruções limitado com um formato fixo;
2. Um número grande de registradores ou o uso do compilador que otimize o uso de registradores;
3. Uma ênfase na otimização do pipeline de instruções.

Os projetistas de arquitetura de processadores alegam que um conjunto de instruções é a principal interface em um computador. Isso se deve ao fato de que recai sobre ele o papel de de interface entre o hardware e o software. Para os desenvolvedores de ISAs de licenças livres, se um bom conjunto de instruções for bem elaborado, disponível para uso de todos, isso pode fazer com que os preços de softwares sejam reduzidos, já que deve haver maior reusabilidade de código[2].

Outro ponto interessante é que ter uma arquitetura de acesso livre pode impulsionar maior competição entre os fabricantes de hardware, que poderiam utilizar mais recursos para o desenvolvimento de hardware, gastando menos com suporte para software.

2.1 RISC-V

Segundo o site oficial de desenvolvimento da arquitetura RISC-V, ela é uma ISA aberta, gratuita que tem por objetivo permitir inovação de processadores por meio da colaboração de desenvolvedores independentes. Foi criada no meio acadêmico e entrega flexibilidade de software, liberdade de hardware na arquitetura, o que pretende ser base para o desenvolvimento futuro nesta linha de pesquisa [3] [2].

Pautada na filosofia do conjunto de instruções reduzidas, a RISC-V tem como precursora mais notória o MIPS (acrônimo para Microprocessor without Interlocked Pipeline Stages) adquirido recentemente por uma empresa privada.

E como argumento para a escolha do estudo desta arquitetura, Patterson [2], defende que por não ser proprietário, existem múltiplos simuladores abertos, debuggers, e várias implementações em linguagens descritivas hardware com livre acesso. Ele ainda projeta que, em breve, haverá plataformas de baixo-custo com suporte aos programas RISC-V.

2.2 RARS: RISC-V Assembler and Runtime Simulator

O RARS, o RISC-V Assembler, Simulator e Runtime, monta e simula a execução de programas de linguagem de montagem RISC-V. Seu principal objetivo é ser um ambiente de desenvolvimento efetivo para pessoas iniciando o desenvolvimento com o RISC-V [2]. Desenvolvido em Java, o simulador pode ser usado da linha de comando ou pelo ambiente de desenvolvimento integrado. Nas distribuições mais recentes, ele suporta RV32IMFN (ISA 32 bits, com multiplicação inteiros, ponto flutuante e interrupções a nível de usuário).

A IDE oferece suporte de edição e assembling, mas o ponto forte do simulador é o suporte para tratamento de erros interativo. O programador pode facilmente configurar pontos de parada de execução ou executar em passos, para frente ou para trás, enquanto visualiza e edita registradores diretamente no conteúdo da posição de memória [2].

2.3 Diretivas de Montagem

Todas as diretivas do montador possuem nomes começados com ponto (". "). O restante do nome é composto normalmente por letras minúsculas. Serão abordadas algumas das possíveis diretivas e explicado seu funcionamento, na ordem que aparecem no arquivo base do anexo I, sort.s.

- .file
- .option
- .data
- .align
- .type
- .size
- .word
- .section
- .rodata
- .string
- .text
- .global
- .type
- .size
- .-show
- .ident

2.4 Compilador RISC-V GCC: riscv64-unknown-elf-gcc

O prefixo 64 ou 32 anexado à arquitetura não significa que o conjunto de ferramentas seja executado apenas em plataformas de 64 ou 32 bits assim como não significa que o compilador produz binários RISC-V de 64 bits ou 32 bits. Na verdade, os compiladores produzem binários de 32 e 64 bits, baseados em parâmetros -march e -mabi. A única diferença são as configurações padrões, quando o compilador é invocado sem o -march e -mabi explicitamente definidos na linha de comando (GNU MCU Eclipse, 2018). A parte desconhecida do comando se refere ao sistema operacional, no sentido de que a escolha é indiferente ao binário resultante.

2.4.1 Argumento -march

O projeto RISC-V não é uma arquitetura única, mas uma família de arquiteturas, com componentes opcionais, identificados por letras[9]. As strings da ISA RISC-V começam com RV32I, RV32E, RV64I ou RV128I indicando o tamanho do espaço de endereço suportado em bits para o inteiro base da ISA .

- RV32I: ISA de armazenamento e carregamento da memória com registradores de números de propósito geral de 32 bits.
- RV32E: Versão nativa da RV32I com apenas 16 registradores inteiros.
- RV64I: Versão de 64 bits da RV32I, na qual os registradores de propósito geral são de 64 bits.

Além dessas ISAs básicas, outras extensões são especificadas. O conjunto de ferramentas é constituído por:

- M: Multiplicação e Divisão de Inteiros
- A: Atomicidade
- F: Ponto Flutuante de Precisão Simples
- D: Ponto Flutuante de Precisão Dupla
- C: Instruções compactas de 16 bits
- G: “Geral”, um atalho para o IMAFD

As strings da ISA do RISC-V são definidas concatenando as extensões suportadas a ISA base na ordem pontuada acima. Os usuários podem controlar o conjunto de instruções que o GCC usa ao gerar o código do assembly passando a string da ISA em letra minúscula para a opção -march do GCC[2].

2.4.2 Argumento -mabi

Para controlar as instruções disponíveis para o GCC durante a geração de código (a qual define o conjunto de implementações nas quais o código gerado será executado), os usuários podem selecionar várias ABIs para destino (que define a convenção de chamada e o layout de objetos na memória). Objetos e bibliotecas só podem ser vinculados se seguirem a mesma ABI. O manual de instruções do RISC-V (2017) define duas ABIs inteiras e três ABIs de ponto flutuante, que juntas são tratadas como uma única cadeia ABI. As ABIs inteiras seguem o esquema de nomenclatura padrão:

- ilp32: int, long e ponteiros são todos de 32 bits. Long long é um tipo de 64 bits, char é de 8 bits e short é de 16 bits.
- lp64: long e ponteiros são de 64 bits, enquanto int é um tipo de 32 bits. Os outros tipos permanecem os mesmos que o ilp32.

As ABIs de ponto flutuante são uma adição específica do RISC-V:

- “” (String vazia): Nenhum argumento de ponto flutuante é passado nos registradores.
- f: argumentos de ponto flutuante de 32 bits são passados para os registradores. Esta ABI requer a extensão F, pois sem ela não há registradores de ponto flutuante.
- d: argumentos de ponto flutuante de 64 bits são passados nos registradores. Esta ABI requer a extensão D.

2.4.3 Tempo de Execução

Para efeitos de esclarecimento, ao longo de todo experimento, será usado o Tempo como medida. Mais especificamente o tempo de execução de um programa, já que não é foco de estudo atrasos relacionados a computação de sinais de I/O ou tarefas do sistema operacional. Sobre o tempo como medida, PATTERSON (2015, P. 38) afirma que "a única medida completa e confiável é o Tempo". Ele também define as seguintes relações para o cálculo do tempo de execução:

$$T_{exec} = I \times CPI \times T \tag{1}$$

Onde T_{exec} é o tempo de execução, I a quantidade de instruções, CPI a quantidade média de ciclos de clock por instrução e T o inverso da frequência de clock.

3 Descrição Experimental

1. Materiais
   - Sistema Operacional com suporte ao Java Runtime 1.8 do Java SE Java Runtime Environment (JRE);
   - Compilador riscv64-unknown-elf-gcc
   - RARS - RISC-V Assembler and Runtime Simulator, Release 1.0;
2. Metodologia
   - O presente relatório tem sua rotina central no estudo dirigido fornecido pela plataforma online de aprendizado da Universidade de Brasília da disciplina Organização e Arquitetura de Computadores[7], ministrada pelo professor Ph.D M. V. Lamar.[8]

4 Questão 1

4.1 Questão 1.1 - Memórias

4.1.1 Memória de Instruções

Endereço do início da memória de instruções: 0x00400000 Endereço do fim da memória de instruções: 0x0040011b Tamanho hexadecimal da memória de instruções:

$$0x00400000 - 0x0040011b + 1 = 0x11c \tag{2}$$

. Convertendo para decimal:

$$16^2 + 16^1 + 12.16^0 = 284bytes \tag{3}$$

4.1.2 Memória de Dados

Endereço do início da memória de dados: 0x10010000 Endereço do fim da memória de dados: 0x1001002b Tamanho hexadecimal da memória de dados:

$$0x10010000 - 0x1001002b + 1 = 0x2c \tag{4}$$

Convertendo para decimal:

$$2.16^1 + 12.16^0 = 44bytes \tag{5}$$

Tamanho total do arquivo:

$$MEMORIADEDADOS + MEMORIADEINSTRUCOES = 284 + 44 = 328bytes \tag{6}$$

4.2 Questão 1.2

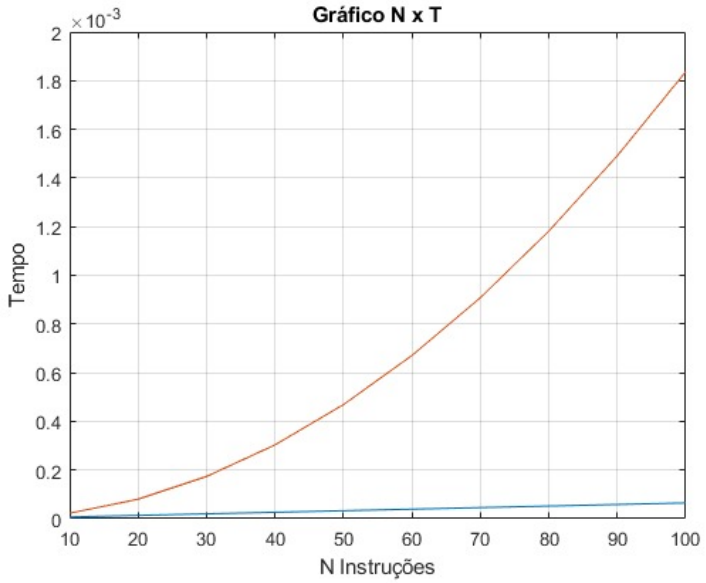

Figura 1: Gráfico - N elementos por tempo de execução.

4.3 Questão 1.4

Através da linha de comando Powershell/Bash, acesse o diretório "laboratory1"e insira o seguinte comando:

$$./Q1/4/bmp2isc.exe/Q1/4/eagles$$

Será gerado um arquivo eagles.s no diretório "laboratory1/Q1/4/". Agora é necessário editar o arquivo "eagle.s"e alterar a primeira linha. Substitua:

$$/Q1/4/eagles : .word\ 320, 240$$

Por

$$eagles : .word\ 320, 240$$

A seguir basta abrir o Rars (v12.3), abrir o arquivo bitmap.s, conectar o display bitmap e rodar a aplicação. Segue a imagem utilizada:

Figura 2

5 Questão 2

5.1 Questão 2.2

Para este item, tivemos que implementar as funções printf e putchar. Para isso, utilizamos de chamadas do sistema "ecall", chamando os serviços PrintInt e PrintString. No caso do arquivo sortc2, também tivemos que implementar a função memcpy, utilizando instruções de acesso a memória SW e LW, que insere numa pilha os valores do vetor a ser ordenado. Em alguns casos, tivemos que adicionar um jump para o registrador ra, onde alteramos o valor do registrador PC para o endereço de retorno da chamada da função, para que o procedimento retornasse a função main.

5.2 Questão 2.3

Compile os programas sortc.c e sortc2.c e, com a ajuda do Rars, monte uma tabela comparativa com o número total de instruções executadas e o tamanho em bytes dos códigos em linguagem de máquina gerados para cada diretiva de otimização da compilação (-O0, -O1, -O2, -O3, -Os). Compare ainda com os resultados obtidos no item 1.1) do programa sort.s que foi implementado diretamente em Assembly. Analise os resultados obtidos.

Sortc.c		
Diretiva	Número de instruções	Tamanho do código
-O0	1825	2,33 KB
-O1	772	1,77 KB
-O2	18	1,39 KB
-O3	553	1,52 KB
-Os	848	1,68 KB

Sortc2.c		
Diretiva	Número de instruções	Tamanho do código
-O0	1910	2,73 KB
-O1	809	2,09 KB
-O2	616	1,91 KB
-O3	597	1,88 KB
-Os	908	1,98 KB

Para este item, tivemos que implementar as funções printf e putchar. Para isso, utilizamos de chamadas do sistema "ecall", chamando os serviços PrintInt e PrintString. No caso do arquivo sortc2, também tivemos que implementar a função memcpy, utilizando instruções de acesso a memória SW e LW, que insere numa pilha os valores do vetor a ser ordenado. Em alguns casos, tivemos que adicionar um jump para o registrador ra, onde alteramos o valor do registrador PC para o endereço de retorno da chamada da função, para que o procedimento retornasse a função main.
É possível perceber que algumas diretivas foram mais eficientes que o código feito em assembly(Q1.1) e que a diretiva o qual apresentou o menor número de instruçoes foi a -O3. é necessário resaltar támbem que houve um erro na diretiva -O2 no sortc, por isto que na tabela ela se encontra somente com 18 instruções.

6 Entregador de Pizzas

Para a realização da questão 3, primeiramente, foi desenvolvido o procedimento SORTEIO que recebe a quantidade de clientes como números inteiro (N) e um ponteiro (C) de inicio de um vetor que mostra o endereço, escolhido de forma aleatória, de cada cliente(x,y). Para o seu desenvolvimento foi realizado o um método *ecall*, já apresentada pela própria documentação, onde adicionando o valor *li a7, 42 # Adiciona o valor 42 ao a7*, e adicionando um limite, recebemos de volta um valor aleatório até a demarcação já pré estabelecida. Assim, obetemos os seguintes resultados:

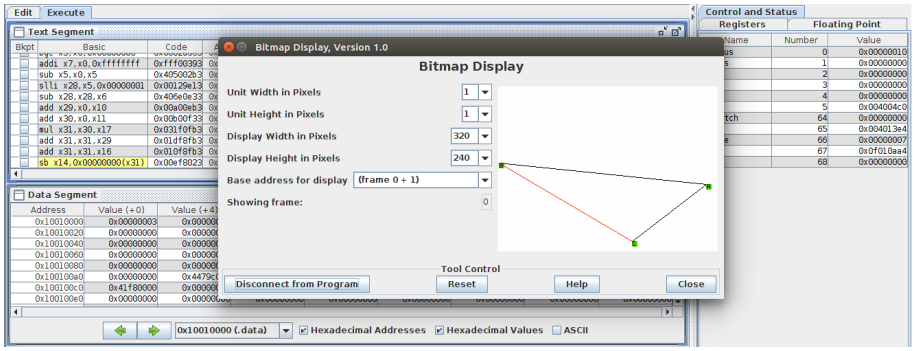

Figura 3: Primeiro caso aleatório.

Assim, podemos notar, na figura 1, em primeira execução do código os pontos sendo bem espalhado na tela *Bitmap*. Mas, tendo a seguinte figura, temos:

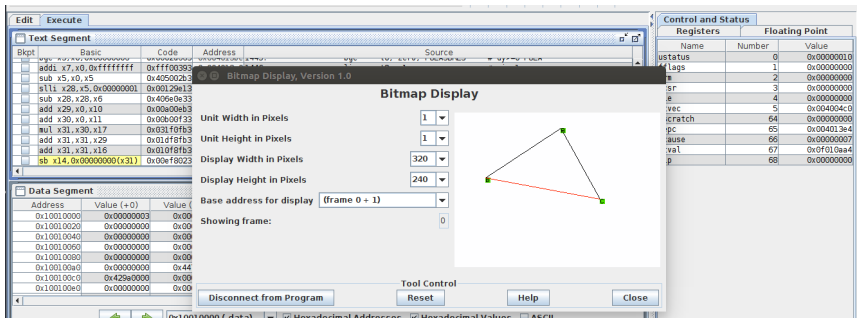

Figura 4: Segundo caso.

Com isso, pela figura 2 podemos perceber a diferença entre as casas dos clientes. Por isso, a implementação do procedimento realizado com sucesso.

Logo, para o próximo procedimento DESENHA era necessário pintar a casa do cliente. Todos os desenvolvimentos para preencher os pixels do *Bitmap* foi ultilizado, ja fornecido pelo Professor Marcus Lamar onde limpa toda a tela com a cor branca(de código 255 em RGB), pegando os endereço (x e y) de cada cliente e pintando o fundo de verde e a letra que ele representa de branco. Para isso, o código era necessário ter um hexadecimal de representação: *hex: 0x0000bbff, b é valor*

dos pixel para o fundo e f para o de frente. Notasse nas figuras 1 e 2 que o fundo é verde (bb = 32) e a frente(ff = 00) é preto.
O procedimento 3 denominado ROTAS era necessário desenhar linhas de uma casa a outra. Retirando um método *ecall*, fornecido pelo Professor, que desenhar linhas usando o endereço x e y tanto do ponto inicial até o ponto final. Contudo, pegando sempre a distância de uma casa a outra e guardando numa matriz D e para isso usamos a seguinte equação:

$$d_{i,j} = \sqrt{(C_i(x) - C_j(x))^2 + (C_i(y) - C_j(y))^2} \tag{7}$$

Todo o procedimento foi devidamente guardado no na matriz, porém foi desenvolvida uma nova matriz(ORDEM) guardando os mesmos valores para ser ultilizada no procedimento seguinte.

Para o quarto e último procedimento ORDENA, era necessário ordenar a matriz D de distâncias pegos anteriormente e pegar o menor caminho entre eles e desenha essa rota de vermelho.

No processo de seu desenvolvimento usamos o processo de ordenação no na matriz ORDEM para em seguida compara-la com a matriz D. Com isso, era possivel se obter a rota que vai de casa para qual casa.

Com o decorrer do código tive um erro que não conseguimos identificar no desenvolvimento, e atualmente desenhamos somente a primeira rota menor. Como podemos ver nas figuras seguintes:

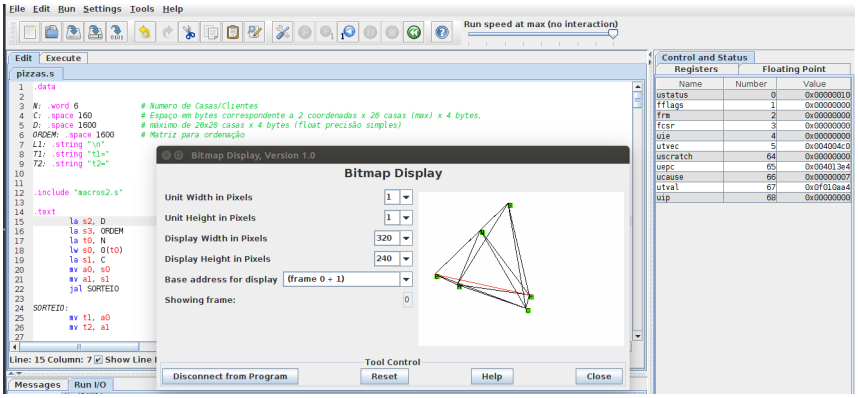

Figura 5: Mapa para 6 clientes

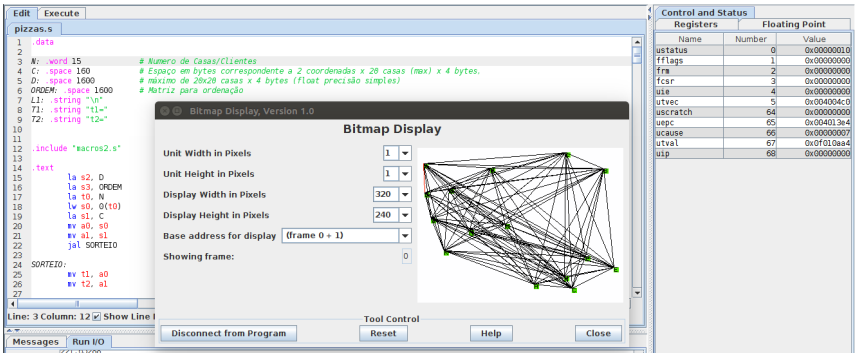

Figura 6: Mapa para 15 clientes

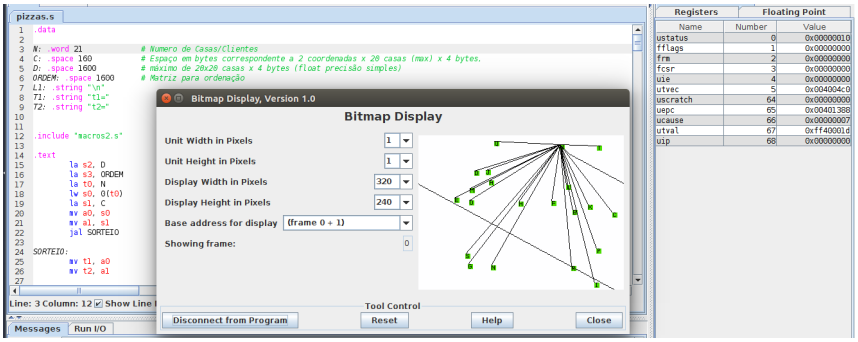

Figura 7: Mapa para 21 clientes

O exercício era como um todo ser realizado para no máximo 20 clientes. Devido nos reservarmos espaço de memória para no maximo 20 clientes, a figura 5 ilustra muito bem o que o ocorre para quando ultrapassamos esse limite.

Para o número de instruções para o código a sua crescente linearidade é perceptivel com o decorrer do numero de clientes aumenta. apesar do fato de ter vários laços de repetição o que tornaria mais complexo o código, mas as imagens a seguir ilustram exatamente a quantidade de instruções.

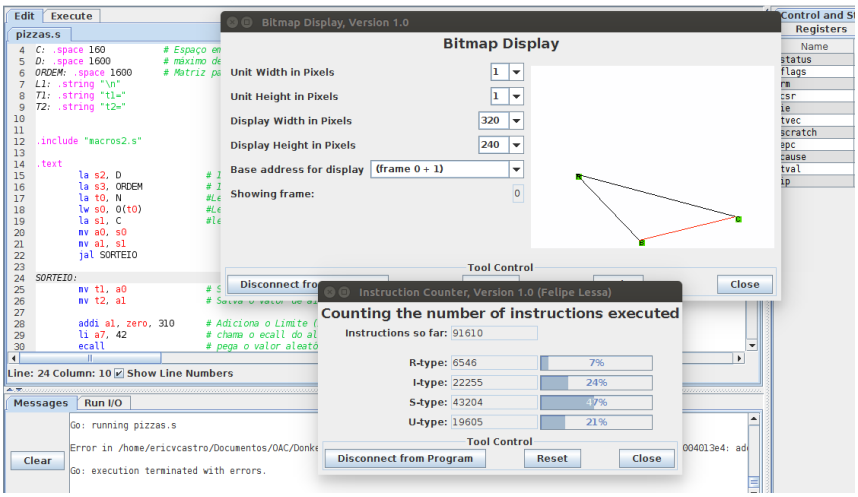

Figura 8: Para 3 clientes temos 91.610 instruções

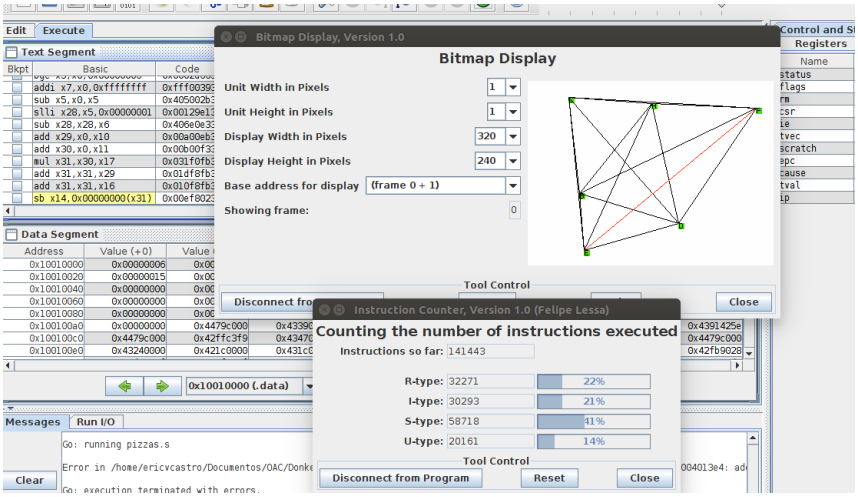

Figura 9: Para 6 clientes temos 141.443 instruções

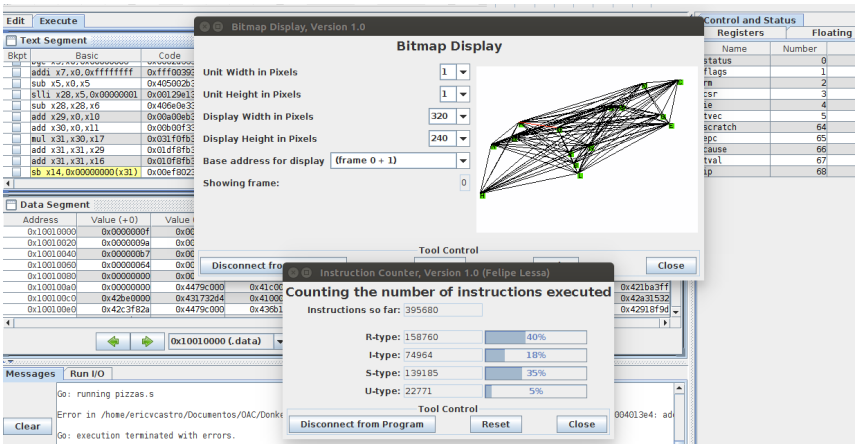

Figura 10: Para 15 clientes temos 395.680 instruções

A linearidade da instrução com relação ao numero de clientes deriva ao fato de que como o código não está completo, a sua complexidade cai drasticamente, pois em teoria, por trabalhar com ordenação e muitos laços de repetições, era para apresentar crescente exponencial.

7 Conclusão

Ao fim do laboratório, pode-se concluir que grande parte dos procedimentos foram implementados como requisitado e alcançaram os resultados esperados, baseando-se na teoria.

Considerando o que foi proposto para o laboratório nº 1, que foi trabalhar com ambiente de simulação RARS e o compilador GCC para o Assembly RISC-V, implementando as etapa de analise de código e trabalhando com o problema do Entregador de Pizza, conferimos a relação e dificuldade em idealizar e implementar um projeto computacional em um linguagem de baixo nivel.

Ao fim do experimento, foi realizado a gravação de um vídeo [1], como requisitado, para validação dos resultados obtidos.

Referências

[1] UnB - OAC Turma A - 2019/1 – Grupo 3 - Laboratório 1. https://youtu.be/ZrfdWhnLPcg. Accessed: 2019-05-08.

[2] D. A. Patterson. *Computer Organization and Design – The Hardware/Software Interface, RISC-V edition*. Morgan Kaufmann, 2017.

[3] D. A. Patterson and A. Waterman. *Guia Prático RISCV*. Strawberry Canyon, 2019.