**INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA**

Licenciatura em Engenharia Informática e de Computadores

Logo

Description automatically generated

Arquitetura de Computadores

**Relatório Trabalho 2**

Trabalho realizado por:

|  |  |
| --- | --- |
| Nome: André Monteiro | Nº 43842 |
| Nome: Constança Costa | Nº 50541 |
| Turma: LEIC24D  Docente: Rui Duarte |  |

2022 / 2023 Semestre Verão

15 de abril de 2023

**Índice**

1. **Introdução**
2. **Desenvolvimento do trabalho**
   1. Análise da microarquitectura
   2. Codificação das instruções
   3. Projeto do descodificador de instruções
   4. Codificação de programas em linguagem máquina
3. **Conclusão**

# **Introdução**

Este relatório remete para o acompanhamento da construção de uma computer network proposta pelo professor da cadeira. Neste momento será descrito a execução da 1º fase deste projeto, que se foca na criação de um webserver.

Temos como objetivo analisar o funcionamento de uma conexão HTTP com base no protocolo TCP, protocolo este que é usado para realizar uma conexão estável e sem perda de informação entre um webserver, disponibilizado pelo enunciado e o cliente (nós), cada um operando em locais distintos.

Numa conexão com base no protocolo TCP inicia-se com o cliente enviando uma mensagem TCP para o servidor a pedir que se inicie uma sessão HTTP, caso o servidor esteja à escuta este deverá responder com uma mensagem a aceitar a conexão, a este procedimento chama-se **Handshaking.** Agora o cliente poderá comunicar com o servidor, no entanto qualquer pedido que este queira transmitir para o servidor terá de passar pela fase de **Encapsulamento,** ou seja, a mensagem que vem da camada de Aplicação terá de percorrer as camadas de Transporte, Rede, Ligação e Física, nesta ordem, e irá ganhar um cabeçalho quando passa por elas. Ao percorrer a última camada, do lado do servidor irá ocorrer a fase de **Desencapsulamento** que consiste no procedimento inverso do encapsulamento. A mensagem chega ao servidor e este irá responder de acordo com o que o cliente quer. Quando o cliente não tiver mais pedidos o servidor fecha a sessão.

Implementou-se e testou-se um web client desenvolvido pelo grupo através de programação em Kotlin e um server local através de XAMPP (instalado no início da realização deste trabalho), sendo tudo possível devido ao uso da aplicação WireShark.

# **Desenvolvimento do trabalho**

***2.1.*** Análise da microarquitectura

1)

A afirmação é verdadeira, pois como indicado no enunciado o processador é de ciclo único e como se pode observar na imagem a memória para dados e programa estão separadas.

Diagram, schematic

Description automatically generated

2)

-O bloco EXT serve para a extensão(aumentar) do número de bits, ou seja, 6-8

-Sinal SE indica o valor que os bits estendidos será 0 ou 1

-bne e mov, sendo um processador de 8 bits e o opcode ocupando 3, não há como ter espaço suficiente no caso do bne pois a label é 6 bits,+-32 endereços. Pelo que para codificar a instrução o valor da label é calculado pelo label-PC. O mov apresenta a mesma situação, opcode 3, rd 2/3 não há espaço para o valor imediato de 3 bits

3)

Vantagens:

bne

- condicional por isso só executa caso a condição se verifique

-tendo em conta a fig 1 podemos usar labels não estando restritos a colocar o valor da label num registo antes de executar o salto

b

- mais rápido

Desvantagens:

bne

-demora mais ciclos, devido à subtração, o incremento do PC e a verificação do valor da flag Z (0)

b

-executa sempre

***2.2*** Codificação das instruturas

1)

Graphical user interface, text, application

Description automatically generated



2)

|  |  |
| --- | --- |
| instrução | opcode |
| b | 000 |
| bne | 001 |
| ldr | 010 |
| str | 011 |
| add | 100 |
| cmp | 101 |
| push | 110 |
| mov | 111 |

***2.3*** Projeto do descodificador de instruções

Na fig.1 captámos um 503 GET que indica que o servidor ainda não está disponível para responder ao cliente.

Enquanto, na fig.2 captámos um 200OK que indica que o pedido foi bem sucedido, neste caso o cliente queria fazer *fetch* de algo do servidor visto que esta é a resposta de um pedido GET.

***2.4*** Projeto Codificação de programas em linguagem máquina

Estes foram os Headers que apanhámos com mais frequência:

**GET**: Componente do pacote de rede enviado de um browser ou cliente para o servidor para pedir uma certa página ou data no webserver;

**HOST:** Especifica que website ou aplicação web processou o http request;

**Connection (Keep-Alive):** Permite que a conexão permaneça aberta no caso de se estabelecerem mais solicitações/ respostas HTTP;

# Conclusão

Com a realização deste trabalho pudemos observar as operações entre servidor e cliente. Neste momento somos capazes de entender o código de resposta retornado pela conexão servidor ao cliente. Pudemos apreender que existem vários cabeçalhos, os principais são os cabeçalhos de resposta (enviados pelo servidor) e os cabeçalhos de requisição (enviados pelo cliente).

# Anexo

import java.io.BufferedReader

import java.io.InputStreamReader

import java.io.PrintWriter

import java.net.Socket

fun main (){

val read = readln()

// lê e guarda o input do utilizador

val socket = Socket("localhost", 80)

//estabelece a conexão entre o host e o servidor com o respetivo port

val autoflush = true

// quando verdadeiro permite que o PrintWriter escreva no canal de Output

val out = PrintWriter( socket.getOutputStream(),

// cria um canal de Output para enviar informação para o servidor

autoflush)

// envia informação através do canal de comunicação

val `in` = BufferedReader(

// acumula a informação enviada pelo servidor para ser depois apresentada ao utilizador

InputStreamReader(

// recebe a informação enviada pelo servidor

socket.getInputStream())

// cria um canal de Input para receber a informação enviada pelo servidor

)

// envia um http request para o web server

out.println("HEAD https://$read/dashboard/ HTTP/1.1")

// escreve no Output stream a mensagem de pedido get

out.println("Host: localhost")

out.println("Connection: Close")

// fecha a conexão com o servidor

out.println()

// lê a resposta so servidor

// e cria a mensagem que vai ser escrita para o utilizador

var loop = true

val sb = StringBuilder(8096)

while (loop) {

if (`in`.ready()) {

var i = 0

while (i != -1) {

i = `in`.read()

sb.append(i.toChar())

}

loop = false

}

}

System.out.println(sb.toString())

//escreve a mensagem para o utilizador

socket.close()

}

Text

Description automatically generated

**Fig.5** output do Código do cliente