Arquitetura de Computadores 2023/24

TPC 2

Entrega: 9h59m de 16 de abril de 2024

Este trabalho de casa consiste num exercício de programação **a ser realizado em grupo de no máximo dois alunos**. Pode esclarecer dúvidas gerais com colegas, mas a solução e a escrita do código deve ser estritamente realizada pelos membros do grupo. Todas as resoluções serão comparadas de forma automática e os casos de plágio serão punidos de acordo com os regulamentos em vigor. **Caso use ferramentas como CoPilot ou o ChatGPT, deve incluir no código fonte um comentário a relatar esse uso.**

Exercício

Neste exercício deve completar o código de um simulador (escrito na linguagem C) de uma arquitetura composta por um CPU muito simples e uma memória. Este inclui, já implementada, uma consola onde é possível um utilizador dar comandos para ler e escrever na memória central, incluindo ler para memória os valores representados num ficheiro de texto. Pode ainda mandar executar as instruções presentes na memória. Os comandos implementados são os seguintes:

> `poke` *EEE X* – escreve no endereço *EEE* o valor *X*.
>
> `peek` *EEE* – ler o valor no endereço de memória *EEE*. O valor é afixado em hexadecimal.
>
> `dump` *X* – mostra o conteúdo do acumulador, flags e de *X* palavras da memória a partir do endereço 0.
>
> `load` *FILE* – lê o ficheiro de nome *FILE*, interpretando cada linha de texto como um valor que é colocado sequencialmente na memória de programa.
>
> `run` – executa as instruções a partir do endereço 0 de memória.

Os valores *X* e *EEE* podem ser dados em base 10 ou em base 16 se iniciados por `0x`.

O código a completar está no ficheiro `dorun.c` e deve implementar o ciclo de *fetch*, *decode* e *execute* para simular a execução das instruções na memória. A execução deve decorrer sempre a partir do endereço 0 de memória, até à execução da instrução HALT (ver secção seguinte).

Instruções máquina do processador:

O processador usa palavras de 8 bits e tem 1 registo de 8 bits que é o acumulador. Tem duas memórias com 256 posições:

- Memória de programa com 256 posições de 16 bits.
- Memória de dados com 256 posições de 8 bits.

Existem os registos:

- *Program Counter* (PC) com 8 bits que contém o endereço da instrução a executar.
- *Instruction Register* com 16 bits que contém a instrução em execução.

Existem 3 flags associadas à unidade artitmética e lógica:

- ZERO contém o valor 1 quando a última operação aritmética deu resultado zero e 0 no caso contrário.
- TRANSPORTE(CARRY) contém 1 quando, na última operação aritmética realizada, houve transporte no bit mais significativo e 0 no caso contrário.
- OVERFLOW contém 1 quando, na última operação aritmética realizada, houve overflow positivo ou negativo e 0 no caso contrário

As instruções têm tamanho fixo de 16 bits distribuídos como indicado a seguir.

15 — 12	11 — 8	7 — 0
Código da instrução (4 bits)	Operação da ALU (4bits)	Endereço ou constante (8 bits)

Bits 15-12: *Código da instrução*: especifica a instrução a executar e como devem ser interpretados os restantes bits.

Bits 11-8: *Especifica a operação a executar pela ALU*. Apenas são utilizadas as sequências 0000 (ADD) e 0001 (SUB).

Bits 7-0: Constante com 8 bits: pode ser um endereço (inteiro sem sinal) ou um valor (inteiro com sinal)

Em algumas instruções há bits que não são considerados. As instruções suportadas e o respetivo código máquina em representação hexadecimal, são descritas a seguir. Na coluna *Instrução* os valores apresentados são sequências de dígitos em binário. São usadas as convenções:

Símbolo	Significado
x	1 bit a ignorar
aaaa	4 bits que indicam a operação a efetuar pela ALU
eeeeeeee	8 bits que especificam um endereço (inteiro sem sinal 0 a 255)
vvvvvvvv	8 bits que especificam uma constante inteiro com sinal de 8 bits (-128 a + 127)

Instrução	**Mnemónica**	**Descrição**
0000 xxxx xxxxxxxx	noop	Não faz nada
0001 0000 vvvvvvvv 0001 0001 vvvvvvvv	addi valor subi valor	Acumulador ← Acumulador + vvvvvvvv Acumulador ← Acumulador - vvvvvvvv
0010 0000 eeeeeeee 0010 0001 eeeeeeee	add endereço sub endereço	Acumulador ← Acumulador + MemóriaDados[eeeeeeee] Acumulador ← Acumulador - MemóriaDados[eeeeeeee]
0011 xxxx xxxxxxxx	clac	Acumulador ← 0; põe a flag ZERO a 1
0100 xxxx eeeeeeee	stor endereço	MemóriaDados[eeeeeeee] = Acumulador
0101 xxxx eeeeeeee	beqz endereço	if (acumulador == 0) pc ← eeeeeeee
1111 xxxx xxxxxxxx	halt	Fim do programa e da simulação

As instruções ADD, SUB, ADDI, SUBI e CLAC devem posicionar corretamente as flags.

Exemplo de um programa para multiplicar dois inteiros sem sinal (como exemplo, calcular 2x3). Os bits irrelevantes estão colocados a 0. Note-se que no ficheiro a carregar no simulador apenas pode estar a coluna do meio, sem o cabeçalho.

Código

```
Address     code            mnemónica
0x00:       0x3000 clac     // ac ← 0
0x01:       0x4002 stor 0x02    // result ←0
0x02:       0x4003 stor 0x03    // counter ←0
0x03:       0x3000 clac     // ac ← 0
0x04:       0x2001 add 0x01     // ac ← multiplier
0x05:       0x2103 sub 0x03     // ac ← multiplier - counter
0x06:       0x5011 beqz 0x11    // se counter == multiplier termina
0x07:       0x3000 clac
0x08:       0x2002 add 0x02     //  ac ← result
0x09:       0x2000 add 0x00     //  ac ← result + multiplicand
0x0A:       0x4002 stor 0x02    // result ← ac
0x0B:       0x3000 clac
0x0C:       0x2003 add 0x03     // ac ← counter
0x0D:       0x1001 addi 0x01    // ac ← ac + 1
0x0E:       0x4003 stor 0x03    // counter ← counter + 1
```

```
0x0F:       0x3000 clac
0x10:       0x5003 beqz 0x03    // jump to instruction at program memory address 0x03
0x11:       0xf000 halt
```

Dados

```
0x00:  2     // multiplicand initialized with poke
0x01:  3     // multiplier initialized with poke
0x02:  ?     // result, non-initialized, read with peek in the end
0x03:  ?     // counter, non-initialized
```

O ficheiro `p.code` fornecido contém o código e dados deste programa em notação hexadecimal, ocupando uma palavra de memória por linha. Correndo o simulador, podemos carregar o programa fazendo `load p.code` e executar com `run`. Podemos consultar o resultado vendo o que fica na variável de endereço 0x02 com `peek`. Podemos alterar as variáveis correspondentes ao multiplicando e multiplicador usando o comando `poke`. Exemplo duma sessão para calcular 4x2:

```
cmd> load p.code

cmd> poke 0x00 4
cmd> poke 0x01 2
cmd> run

HALT instruction executed
cmd> peek 0x02
8
cmd>
```

Entrega

A entrega faz-se de forma a anunciar via CLIP.

A entrega deve ser feita dentro do prazo, submetendo apenas o seu ficheiro *dorun.c.* O cumprimento da especificação pelo código entregue não é o único critério para definir a nota. Outros critérios, como a qualidade do código e completude da solução, também são tidos em conta.

O programa será compilado com o seguinte comando:

```
gcc -Wall -std=c99 -o sim sim.c dorun.c
```