Universidade do Minho

Módulo 1 - Consolidação

Ficha 1
Execução de Instruções

1. Introdução

Pretende-se com esta sessão teórico-prática que os alunos consolidem o conhecimento previamente adquirido sobre o ciclo de execução de instruções de um computador. Recorre-se ao Y86 como ferramenta de trabalho.

2. Execução de instruções no Y86

O modelo de execução de uma máquina sequencial como o Y86 é extremamente simples. As instruções são executadas uma a uma pela ordem com que aparecem escritas no programa. Alterações a esta ordem têm que ser explícitas, isto é, tem que existir uma instrução que explicitamente altera a ordem de execução; chamamos a estas instruções saltos. No Y86 existem 6 tipos de saltos: saltos incondicionais (`jmp`), saltos condicionais (`jle`, `jl`, `je`, `jne`, `jge`, `jg`), invocação de funções (`call`) e retorno de funções (`ret`).

O processador executa repetidamente um ciclo que pode ser visto como constituído por 5 fases (ver figura na próxima página):

extracção da instrução – A instrução armazenada na posição de memória cujo endereço está no registo Program Counter (PC) é lida da memória para um registo auxiliar designado por Instruction Register (IR); O endereço da instrução imediatamente a seguir à actual é calculado somando o comprimento da instrução ao valor do PC; no Y86 este endereço da próxima instrução é designado por `valP`;

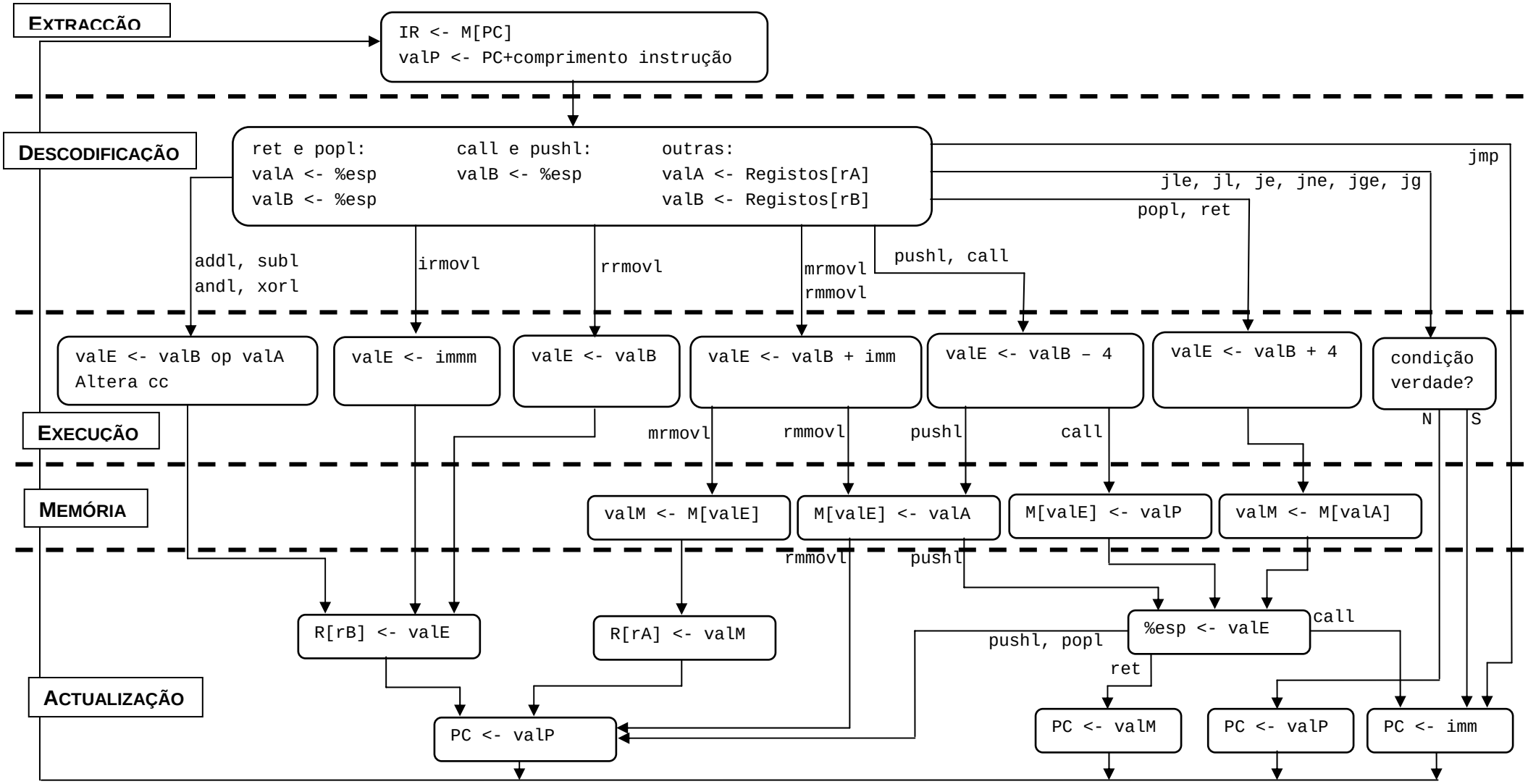
descodificação da instrução e leitura dos operandos – A unidade de controlo descodifica a instrução que se encontra no IR e, simultaneamente, são lidos do banco de registos os operandos desta instrução. No Y86 todos os operandos são valores imediatos ou valores que se encontram armazenados nos registos, podendo, portanto, ser disponibilizados nesta fase; os registos a utilizar são indicados nos campos `rA` e `rB` da instrução; os respectivos valores são colocados pelo banco de registos nos sinais `valA` e `valB` do processador; o sinal correspondente ao valor imediato é referido como `imm`;

execução – é realizada na Unidade Lógica-Aritmética (vulgarmente designada por ALU – Arithmetic and Logic Unit) a operação correspondente à instrução em causa, alterando os códigos de condição. Esta operação pode ser indicada explicitamente pela instrução (caso de um addl, por exemplo) ou pode ser necessária implicitamente (caso de um acesso à memória onde o conteúdo do registo base deve ser somado ao deslocamento para calcular o endereço a ler ou escrever). O resultado é designado por `valE`. As instruções de salto condicional consultam os códigos de condição (cc, designados por *flags* na terminologia inglesa) para determinar se o salto é tomado;

memória – se a instrução implica um acesso à memória (`mrmovl`, `rmmovl`) este é realizado nesta fase. De notar que o endereço da posição de memória a ler ou escrever foi calculado na fase anterior; o valor lido é designado por `valM`.

actualização –o resultado da execução da instrução é escrito no registo correspondente. Adicionalmente, o PC é actualizado para apontar para a próxima instrução; esta pode ser a instrução consecutiva (`valP`), ou, no caso de saltos tomados, a instrução indicada como destino do salto.

De realçar que esta subdivisão e ordenamento das fases de execução de instruções é essencialmente conceptual, ajudando a perceber como é que uma instrução deve ser executada. Na realidade, o número e ordem das fases podem ser diferentes.


EXTRACÇÃO
IR <- M[PC]
valP <- PC+comprimento instrução
DESCODIFICAÇÃO
ret e popl:
valA <- %esp
valB <- %esp
call e pushl:
valB <- %esp
outras:
valA <- Registos[rA]
valB <- Registos[rB]
jmp
jle, jl, je, jne, jge, jg
popl, ret
addl, subl
andl, xorl
irmovl
rrmovl
mrmovl
rmmovl
pushl, call
valE <- valB op valA
Altera cc
valE <- immm
valE <- valB
valE <- valB + imm
valE <- valB – 4
valE <- valB + 4
condição
verdade?
N
S
EXECUÇÃO
mrmovl
rmmovl
pushl
call
MEMÓRIA
valM <- M[valE]
M[valE] <- valA
M[valE] <- valP
valM <- M[valA]
rmmovl
pushl
R[rB] <- valE
R[rA] <- valM
%esp <- valE
call
pushl, popl
ret
ACTUALIZAÇÃO
PC <- valP
PC <- valM
PC <- valP
PC <- imm

Considere o programa em *assembly* do Y86 que se apresenta na próxima tabela. Este encontra-se disponível na página de apoio às sessões TP da disciplina. Grave-o com o nome `prog.ys` (Nota: os ficheiros com código *assembly* do Y86 devem ter a extensão `ys`).

prog.ys

```
      .pos 0  # a execução começa no endereço 0
      jmp main
      .pos 0x010

      .align 4
v:    .long 10
      .long 20

main:
      irmovl v, %ebx
      mrmovl 4(%ebx), %eax
      irmovl $100, %ecx
      addl %ecx, %eax
      rmmovl %eax, 0(%ebx)
      halt
```

Questão 1 – Usando a descrição da arquitectura do Y86 disponível para esta ficha (ver página) apresente a ocupação de memória para este código e respectivos dados. Os resultados devem ser apresentados em hexadecimal. Lembre-se de que o Y86, à semelhança do IA32, é *little endian* – quer isto dizer que constantes com vários *bytes* são representadas com o *byte* menos significativo no endereço mais baixo.

EXEMPLO: representação da primeira instrução `jmp main`

Esta instrução começa no endereço 0x000 (porquê?) e ocupa 5 *bytes* (porquê?).
O seu formato é:

opcode (4 bits)	function (4 bits)	Endereço destino (32 bits)

O opcode é 7, function=0 e o endreço de destino é 0x018 (porquê?).
Em binário temos: 0111 0000 0001 1000 0000 0000 0000 0000 0000
Em hexadecimal: 0x000: 0x7018000000, sendo que 0x000: representa o endereço.

Após repetir o exemplo acima para todas as instruções e dados pode verificar os seus resultados usando o *assembler* do Y86:

```
yas prog.ys
```

Este gera o código binário para um ficheiro designado `prog.yo`

Os ficheiros com código binário do Y86 têm a particularidade de estarem em formato de texto (!!!).
Neste ficheiro para cada linha a coluna da esquerda apresenta o endereço, seguida do conteúdo da respectiva posição de memória em hexadecimal (isto pode ser código ou dados) e finalmente, sob a forma de comentário, a linha do código em *assembly* que a originou.

Questão 2 – Olhando para o código apresentado é fácil perceber que v é um vector com 2 elementos, cada elemento com 4 *bytes*. Qual o endereço base de v? Qual o endereço de v[0]? E de v[1]?

Questão 3 – Para cada instrução identifique os registos e/ou posições de memória lidos/escritos para a executar. Indique o estado antes da execução da instrução, os vários passos durante a sua execução e o estado após a execução. Para indicar o estado apresente apenas os registos, posições de memória e códigos de condição lidos ou escritos por essa instrução.

EXEMPLO: `jmp main`

São lidos os 5 *bytes* a partir do endereço 0x000 e é lido e escrito o PC.

jmp main		
Antes:	Durante:	Após:
PC = 0x000 Mem[0x000] = 0x7018000000	Extracção: IR= Mem[0x000] =0x7018000000 valP = 0x005 Actualização: PC = imm = 0x018	PC = 0x018

Questão 4 – Descreva a um nível mais alto, por exemplo em C, o que faz este programa.