Fundamentos de Arquitetura de Computadores

Tiago Alves

Faculdade UnB Gama
Universidade de Brasília

Simuladores SPIM/MARS: chamadas de sistema

Implementa o montador, pseudo-instruções, simula um sistema operacional com funções de Entrada/Saída em console próprio.

Ex.: Aplicação que escreve na tela: `the answer = 5`

```
    .data
str:
    .asciiz "the answer = "
    .text
    li      $v0, 4      # código de chamada ao sistema para print_str
    la      $a0, str    # endereço da string a imprimir
    syscall             # imprime a string

    li      $v0, 1      # código de chamada ao sistema para print_int
    li      $a0, 5      # inteiro a imprimir
    syscall             # imprime
```

Simuladores SPIM/MARS: chamadas de sistema

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

Exemplo: clear (ponteiro vs. array)

Objetivo: Zerar os componentes do array de tamanho size

```
void clear1(int array[], int size)
{
  int i;
  for(i=0;i<size;i++)
    array[i]=0;
}
```

```
void clear2(int *array, int size)
{
  int *p;
  for(p=&array[0];p<&array[size];p++)
    *p=0;
}
```

Qual o mais eficiente?

Exemplo: clear (ponteiro vs. array)

Objetivo: Zerar os componentes do array de tamanho size

```
void clear1(int array[], int size)
{
  int i;
  for(i=0;i<size;i++)
    array[i]=0;
}
```

```
clear1: move $t0,$zero  # i = 0
Loop1: sll $t1,$t0,2    # $t1 = 4*$t0
  add $t2,$a0,$t1       # $t2 = &array[i]
  sw $zero,0($t2)       # array[i] = 0
  addi $t0,$t0,1        # i = i+1
  slt $t3,$t0,$a1       # $t3 = (i<size)
  bne $t3,$zero,Loop1   # if (), go to Loop1
  jr $ra                # bye!
```

```
void clear2(int *array, int size)
{
  int *p;
  for(p=&array[0];p<&array[size];p++)
    *p=0;
}
```

```
clear2: move $t0,$a1      # $t0 = size
        move $t5,$a0      # $t5 = &array[0]
Loop2:  sw $zero,0($t5)   # array[i] = 0
  addi $t5,$t5,4          # $t5 = $t5+4 = &array[i+1]
  addi $t0,$t0,-1         # $t0 = $t0-1
  bne $t0,$zero,Loop2     # if (), go to Loop2
  jr $ra                  # bye!
```

Qual o mais eficiente?

UnB Gama

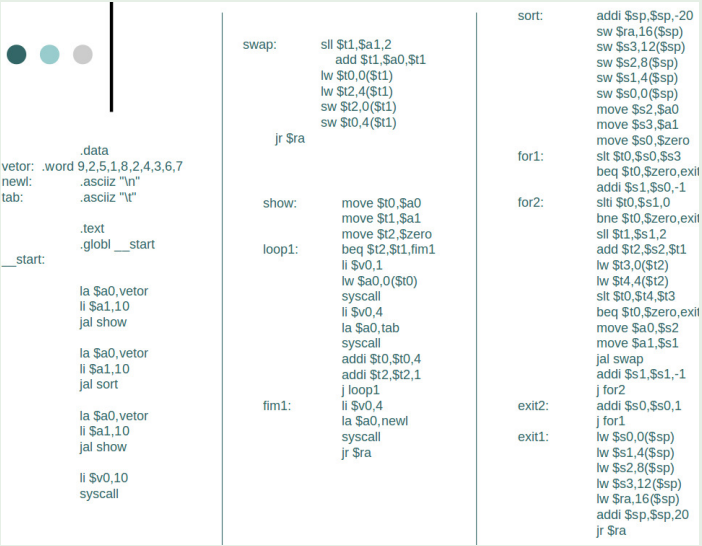
Exemplo: SORT

Compile para Assembly MIPS o seguinte programa C

```
#include <stdio.h>

void show(int v[], int n)
{
   int i;
   for(i=0;i<n;i++)
         printf("%d\t",v[i]);
   printf("\n");
}

void swap(int v[], int k)
{
   int temp;
   temp=v[k];
   v[k]=v[k+1];
   v[k+1]=temp;
}
```

```
void sort(int v[], int n)
{
   int i,j;
    for(i=0;i<n;i++)
        for(j=i-1;j>=0 && v[j]>v[j+1];j--)
            swap(v,j);
}

void main()
{
   int v[10]={9,2,5,1,8,2,4,3,6,7};
   int n=10;

   show(v,n);
   sort(v,n);
   show(v,n);
}
```

Exemplo: SORT



```
            .data
vetor:  .word 9,2,5,1,8,2,4,3,6,7
newl:       .asciiz "\n"
tab:        .asciiz "\t"

            .text
            .globl __start
__start:

            la $a0,vetor
            li $a1,10
            jal show

            la $a0,vetor
            li $a1,10
            jal sort

            la $a0,vetor
            li $a1,10
            jal show

            li $v0,10
            syscall
```

```
swap:   sll $t1,$a1,2
        add $t1,$a0,$t1
        lw $t0,0($t1)
        lw $t2,4($t1)
        sw $t2,0($t1)
        sw $t0,4($t1)
    jr $ra


show:   move $t0,$a0
        move $t1,$a1
        move $t2,$zero
loop1:  beq $t2,$t1,fim1
        li $v0,1
        lw $a0,0($t0)
        syscall
        li $v0,4
        la $a0,tab
        syscall
        addi $t0,$t0,4
        addi $t2,$t2,1
        j loop1
fim1:   li $v0,4
        la $a0,newl
        syscall
        jr $ra
```

```
sort:   addi $sp,$sp,-20
        sw $ra,16($sp)
        sw $s3,12($sp)
        sw $s2,8($sp)
        sw $s1,4($sp)
        sw $s0,0($sp)
        move $s2,$a0
        move $s3,$a1
        move $s0,$zero
for1:   slt $t0,$s0,$s3
        beq $t0,$zero,exit
        addi $s1,$s0,-1
for2:   slti $t0,$s1,0
        bne $t0,$zero,exit
        sll $t1,$s1,2
        add $t2,$s2,$t1
        lw $t3,0($t2)
        lw $t4,4($t2)
        slt $t0,$t4,$t3
        beq $t0,$zero,exit
        move $a0,$s2
        move $a1,$s1
        jal swap
        addi $s1,$s1,-1
        j for2
exit2:  addi $s0,$s0,1
        j for1
exit1:  lw $s0,0($sp)
        lw $s1,4($sp)
        lw $s2,8($sp)
        lw $s3,12($sp)
        lw $ra,16($sp)
        addi $sp,$sp,20
        jr $ra
```

Arquiteturas alternativas

Alternativa de projeto:

- forneça operações mais poderosas;
- o objetivo é reduzir o número de instruções executadas;
- o risco é um tempo de ciclo mais lento e/ou uma CPI mais alta.

Vejamos o IA-32!

IA-32

Linha temporal:

- 1978: O Intel 8086 é anunciado (arquitetura de 16 bits);
- 1980: O co-processador de ponto flutuante Intel 8087 é acrescentado;
- 1982: O 80286 aumenta o espaço de endereçamento para 24 bits; disponibiliza mais instruções;
- 1985: O 80386 estende para 32 bits; novos modos de endereçamento;
- 1989-1995: O 80486, Pentium e Pentium Pro acrescentam algumas instruções (especialmente projetadas para um maior desempenho);
- 1997: 57 novas instruções MMX (SIMD) são acrescentadas; Pentium II;
- 1999: O Pentium III acrescenta outras 70 instruções (SSE — Streaming SIMD Extensions): reação aos concorrentes (ADM 3DNow).
- 2001: Outras 144 instruções (SSE2)

IA-32

Linha temporal:

- 2003: A AMD estende a arquitetura para aumentar o espaço de endereço para 64 bits; estende todos os registradores para 64 bits, além de outras mudanças (AMD64)
- 2004: A Intel se rende e abraça o AMD64 (o chama EM64T) e inclui mais extensões de mídia Essa história ilustra o impacto das "algemas douradas" da compatibilidade "adicionando novos recursos da mesma forma que se coloca roupas em uma sacola", uma arquitetura "difícil de explicar e impossível de amar".
- 2006: SSE4 adiciona 54 instruções! Suporte especial a máquinas virtuais.
- 2007: AMD anuncia 170 instruções do conjunto SSE5
- 2011: Intel anuncia Advanced Vector Extension: registradores SSE evoluem de 128 bits para 256 bits, redefinição de 250 instruções e adição de 128 instruções.

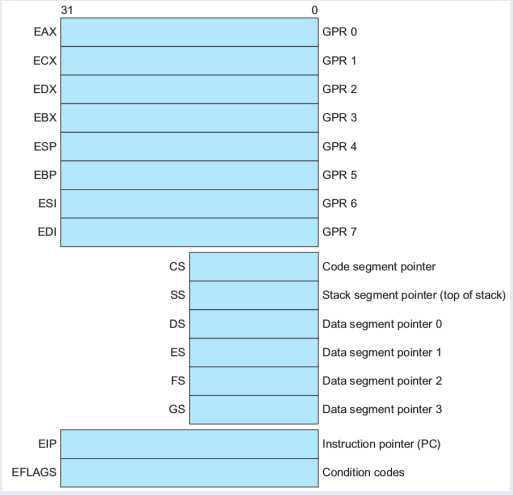
Visão geral do IA-32

Complexidade:

- instruções de 1 a 17 bytes (136 bits) de tamanho
- um operando precisa agir como origem e destino
- um operando pode vir da memória
- modos de endereçamento complexos, por exemplo, "índice base ou escalado com deslocamento de 8 ou 32 bits"

Registradores e endereçamento de dados do IA-32

Registradores no subconjunto de 32 bits que surgiram com o 80386



Restrições de registrador do IA-32

Não são viáveis operações memória-memória.

Imediatos podem ser de 8, 16 ou 32 bits. Qualquer um dos registradores (exceto EIP e EFLAGS) se enquadram nas combinações.

Source/destination operand type	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

Restrições de registrador do IA-32

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`lw $s0,100($s1) # <= 16-bit # displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0) # <=16-bit # displacement`

Instruções típicas do IA-32

Quatro tipos principais de instruções de inteiro:

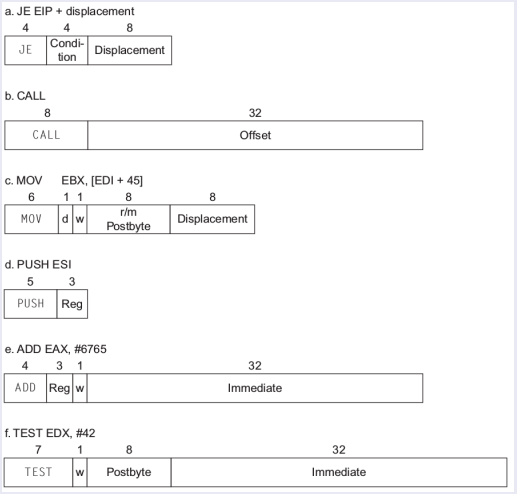
- Movimento de dados, incluindo move, push, pop
- Aritmética e lógica (registrador de destino ou memória)
- Fluxo de controle (uso de códigos de condição/flags)
- Instruções de string, incluindo movimento e comparação de strings.

Instruction	Function
je name	if equal(condition code) {EIP=name}; EIP-128 <= name < EIP+128
jmp name	EIP=name
call name	SP=SP-4; M[SP]=EIP+5; EIP=name;
movw EBX,[EDI+45]	EBX=M[EDI+45]
push ESI	SP=SP-4; M[SP]=ESI
pop EDI	EDI=M[SP]; SP=SP+4
add EAX,#6765	EAX= EAX+6765
test EDX,#42	Set condition code (flags) with EDX and 42
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

call salva EIP da próxima instrução na pilha/stack. EIP é o PC da Intel.

Formatos de instruções do IA-32

Formatos típicos:



Restrições de registrador do IA-32

Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for`JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX $\neq 0$
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory

Restrições de registrador do IA-32

Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

Resumo

A complexidade da instrução é apenas uma variável

- instrução mais simples versus CPI mais alta / velocidade de clock mais baixa

Princípios de projeto:

- simplicidade favorece a regularidade
- menor é melhor
- bom projeto exige comprometimento
- agilizar o caso comum

Arquitetura do conjunto de instruções: uma abstração muito importante!