Instruções - Parte 2

Arquitetura de Computadores

Emilio Francesquini
e.francesquini@ufabc.edu.br

2020.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Arquitetura de Computadores** na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- O conteúdo destes slides foi baseado no conteúdo do livro *Computer Organization And Design: The Hardware/Software Interface*, 5th Edition.

Instruções - A linguagem do computador - Parte 2

Constantes de 32-bits

UFABC

- A maior parte das constantes é pequena
  - 16 bits para armazenar o imediato é normalmente suficiente
- Para quando tivermos uma constante maior, de 32 bits

```
1  lui rt, constant
```

  - Copia 16 bits da constante para a parte esquerda de rt
  - Limpa 16 bits à direita de rt (preenche com zeros)

```
1  lui $s0, 61
2  ori $s0, $s0, 2304
```

Endereçamento de branches

UFABC

- Instruções de branches especificam
  - Opcode, 2 registradores, endereço de destino
- A maior parte dos branches são próximos
  - Tanto para frente quanto para trás

op	rs	rt	constante ou endereço
6 bits	5 bits	5 bits	16 bits

- Endereço relativo ao PC
  - Endereço destino = PC + offset x 4
  - PC já é aumentando de 4 por esta linha

Endereçamento para saltos

UFABC

- Saltos (`j` e `jal`) podem ter como destino qualquer local dentro do segmento de texto
  - Codifica o endereço completo na instrução

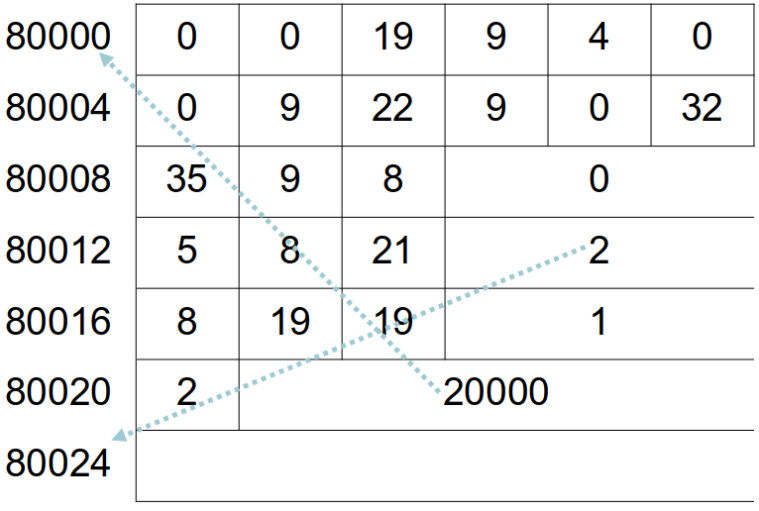
op	constante ou endereço
6 bits	26 bits

- Endereçamento de saltos (Pseudo)diretos
  - Endereço destino = $PC_{31...28}$: (endereço * 4)

Exemplo de destinos

- Código do exemplo anterior
- Assuma que **Loop** esteja na posição 80000

```
1  Loop: sll $t1, $s3, 2
2        add $t1, $t1, $s6
3        lw  $t0, 0($t1)
4        bne $t0, $s5, Exit
5        addi $s3, $s3, 1
6        j Loop
7  Exit: ...
```



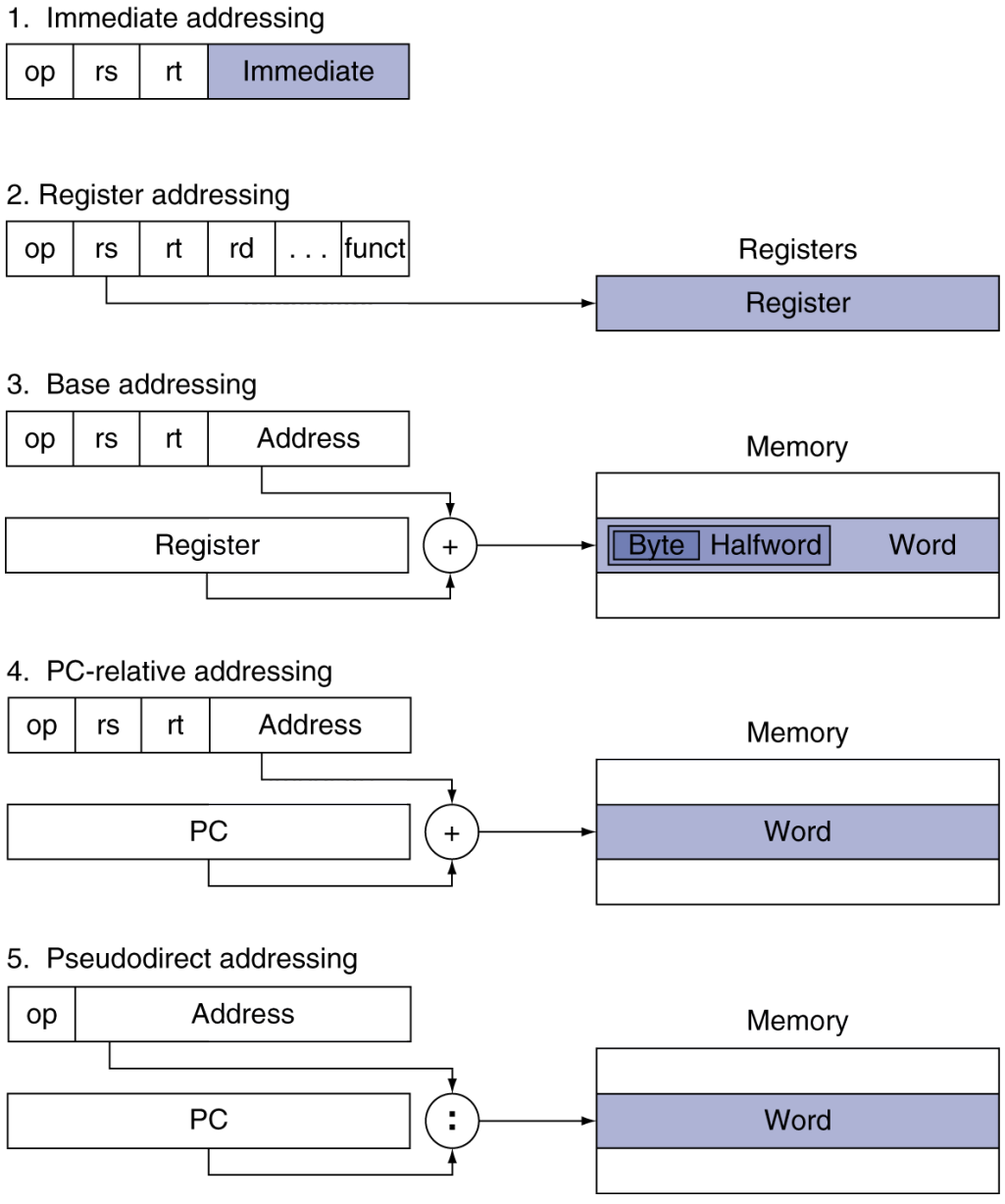
Branchings distantes

UFABC

- Se o branch estiver longe demais para codificar com 16 bits, o assembler reescreve o código
- Exemplo:

```
1      beq $s0,$s1, L1
2              ↓
3      bne $s0,$s1, L2
4      j L1
5  L2:          …
```

Resumo dos modos de endereçamento



UFABC

- Dois processos que compartilham a mesma área de memória
  - P1 escreve e em seguida P2 lê
  - Condição de corrida se P1 e P2 não sincronizarem
    - Resultado depende da ordem em que ocorrem os acessos
- É necessário suporte de hardware
  - Operações atômicas de escrita/leitura
  - Garante que nenhum acesso é feito na posição de memória entre a leitura e a escrita
- Poderia ser uma simples instrução
  - Por exemplo, swap atômico entre registrador e memória
  - Ou um ”par atômico”de instruções

Sincronização no MIPS

- `ll` *load linked*

```
1  ll rt, offset(rs)
```

- `sc` *store conditional*

```
1  sc rt, offset(rs)
```

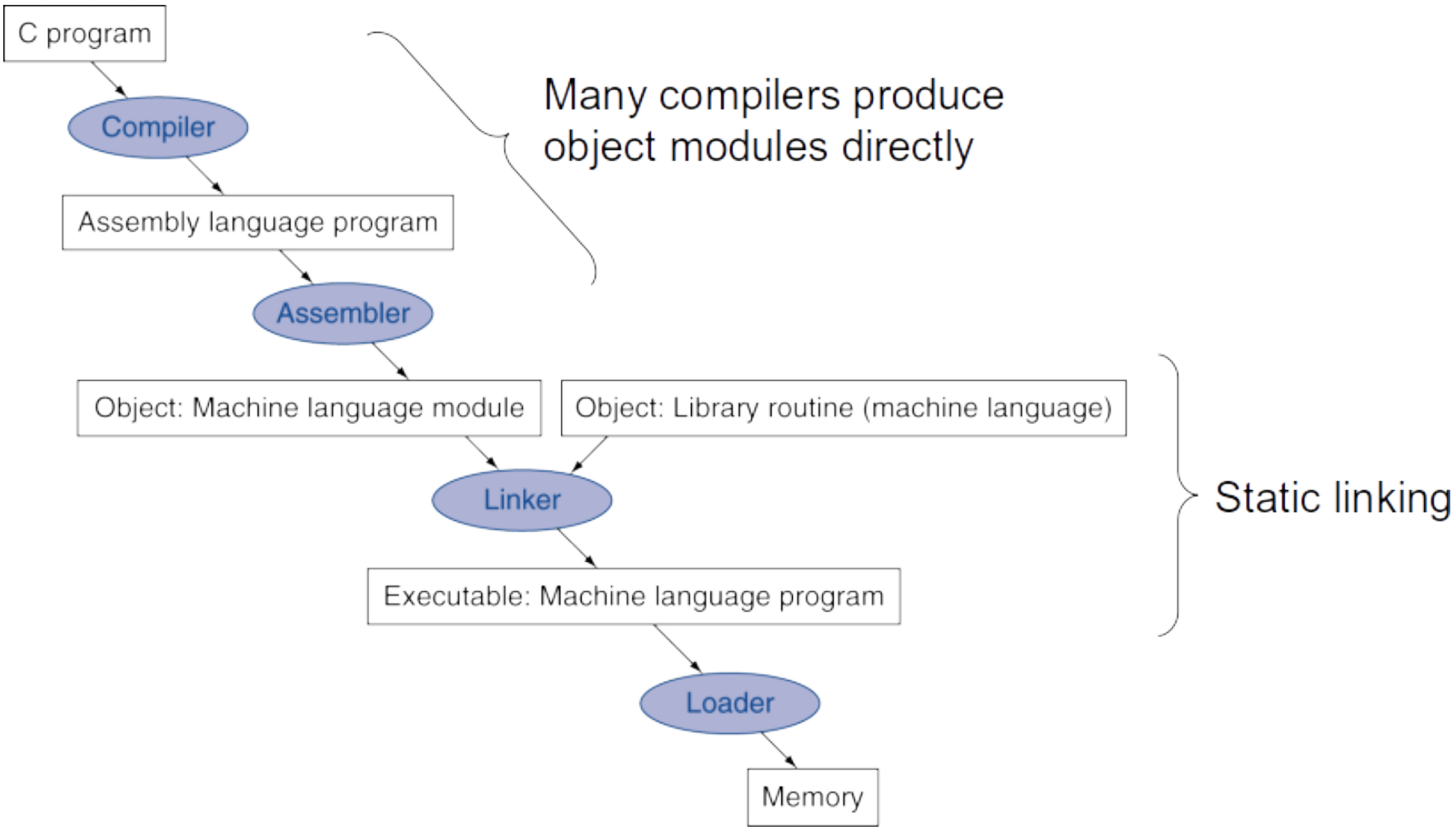
  - Apenas executa com sucesso caso a localização não tenha sido modificada desde o último `ll`
    - Devolve 1 no `rt`
  - Falha caso a posição tenha sido modificada
    - Devolve 0 no `rt`

UFABC

- Exemplo: troca atômica (test & set)

```
1  Try: add $t0, $zero, $s4 # Copia o endereço a ser
   ↪  trocado
2       ll  $t1, 0($s1)     # load linked
3       sc  $t0, 0($s1)     # store conditional
4       beq $t0, $zero, Try # branch se o store falhou
5       add $s4, $zero, $t1 # store do valor em $s4
```

Tradução e inicialização



Pseudo-instruções

UFABC

- O assembler pode fornecer algumas pseudo-instruções
- Normalmente há uma correspondência 1 para 1, nas pseudo-instruções a imaginação do assembler é o limite!

```
1  move $t0, $t1
2  blt $t0, $t1, L
```

```
1  add $t0, $zero, $t1
2  slt $at, $t0, $t1
3  bne $at, $zero, L
```

- `$at` (registrador 1): reservada para o assembler usar como variável temporária

Produzindo um Object Module

UFABC

- Assembler (ou o compilador) traduz o programa para instruções de máquina
- Fornece as informações para construir um programa completo a partir das peças
  - Cabeçalho: descreve o conteúdo do object module
  - Segmento de texto: instruções traduzidas
  - Segmento de dados estáticos: espaço de dados alocados que existem durante a execução completa do programa
  - Relocation info: usado para conteúdos que dependem da posição absoluta do programa (após carregado)
  - Symbol table: definições globais e referências externas
  - Debug info: informações para ligar endereços do objeto com o código fonte

Efetuando o linking de object modules

UFABC

- Produzem uma imagem executável
  1. Mescla segmentos
  2. Ajusta labels (calcula os seus endereços)
  3. Ajusta elementos que são dependentes de localização e referências externas
- Poder-se-ia deixar a correção dos elementos que dependem de localização para ser feito pelo *loader*
  - Contudo, o uso de memória virtual permite que adiantemos o serviço
    - Programa pode ser sempre carregado no mesmo endereço (virtual)

Carregando um programa

UFABC

- Carrega a imagem do programa do disco para a memória
  1. Lê header para determinar os tamanhos dos segmentos
  2. Cria espaço de memória virtual
  3. Copia texto e dados (pré) inicializados para a memória
     - Ou ajusta a tabela de páginas para que quando ocorra uma falha eles possam ser carregados
  4. Ajusta argumentos na pilha
  5. Inicializa registradores (incluindo `$sp`, `$fp`, `$gp`)
  6. Salta para a rotina de inicialização
     - Copia os argumentos para `$a0`, `$a1`, ...
     - Salta para `main`
     - Quando `main` retorna, chama *syscall* para indicar saída

Linking dinâmico

UFABC

- Apenas fazer o procedimento de link/load quando a biblioteca é chamada
  - É preciso que o código do procedimento seja reposicionável
  - Evita o inchamento da imagem causada pelo linking estático de todas as bibliotecas utilizadas (e suas dependências também!)
  - Automaticamente seleciona versões mais novas das bibliotecas

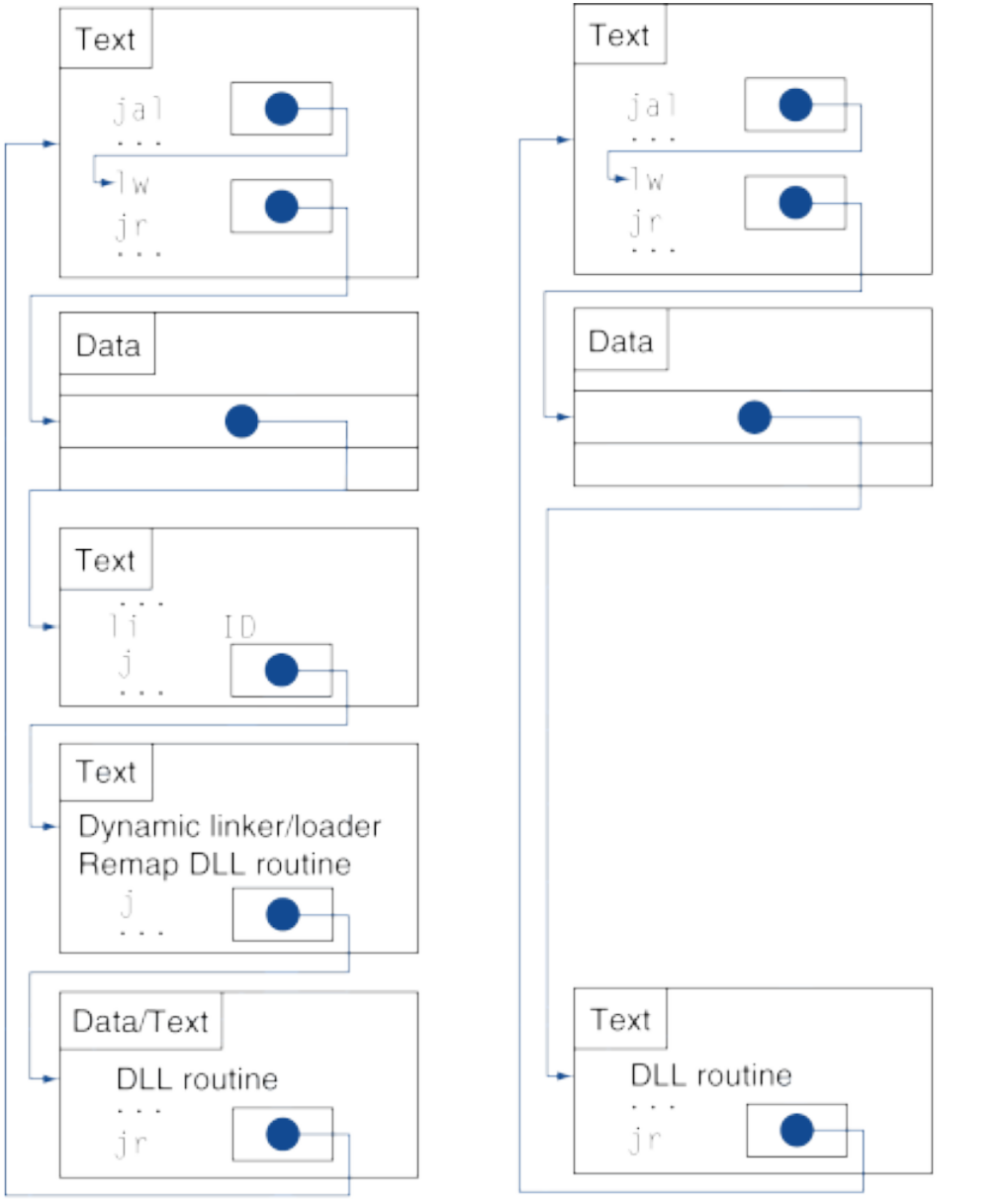
Lazy Linkage

Indirection table

Stub: Loads routine ID,
Jump to linker/loader

Linker/loader code

Dynamically
mapped code



a. First call to DLL routine

b. Subsequent calls to DLL routine

Exemplo: Bubble Sort - swap

UFABC

```
1  void swap(int v[], int k) {
2      int temp;
3      temp = v[k];
4      v[k] = v[k+1];
5      v[k+1] = temp;
6  }
```

- v em $a0
- k em $a1
- temp em $t0

Exemplo: Bubble Sort - swap

UFABC

```
1  swap: sll $t1, $a1, 2    # $t1 = k * 4
2        add $t1, $a0, $t1  # $t1 = v+(k*4)
3                           # (endereço de v[k])
4        lw $t0, 0($t1)     # $t0 (temp) = v[k]
5        lw $t2, 4($t1)     # $t2 = v[k+1]
6        sw $t2, 0($t1)     # v[k] = $t2 (v[k+1])
7        sw $t0, 4($t1)     # v[k+1] = $t0 (temp)
8        jr $ra             # retorna
```

Exemplo: Bubble Sort - `bsort`

UFABC

```
1  void sort (int v[], int n) {
2    int i, j;
3    for (i = 0; i < n; i += 1) {
4      for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
5          swap(v,j);
6      }
7    }
8  }
```

- Função não folha (chama swap)
- v em $a0
- k in $a1
- i in $s0
- j in $s1

Exemplo: Bubble Sort - **bsort** - Corpo

UFABC

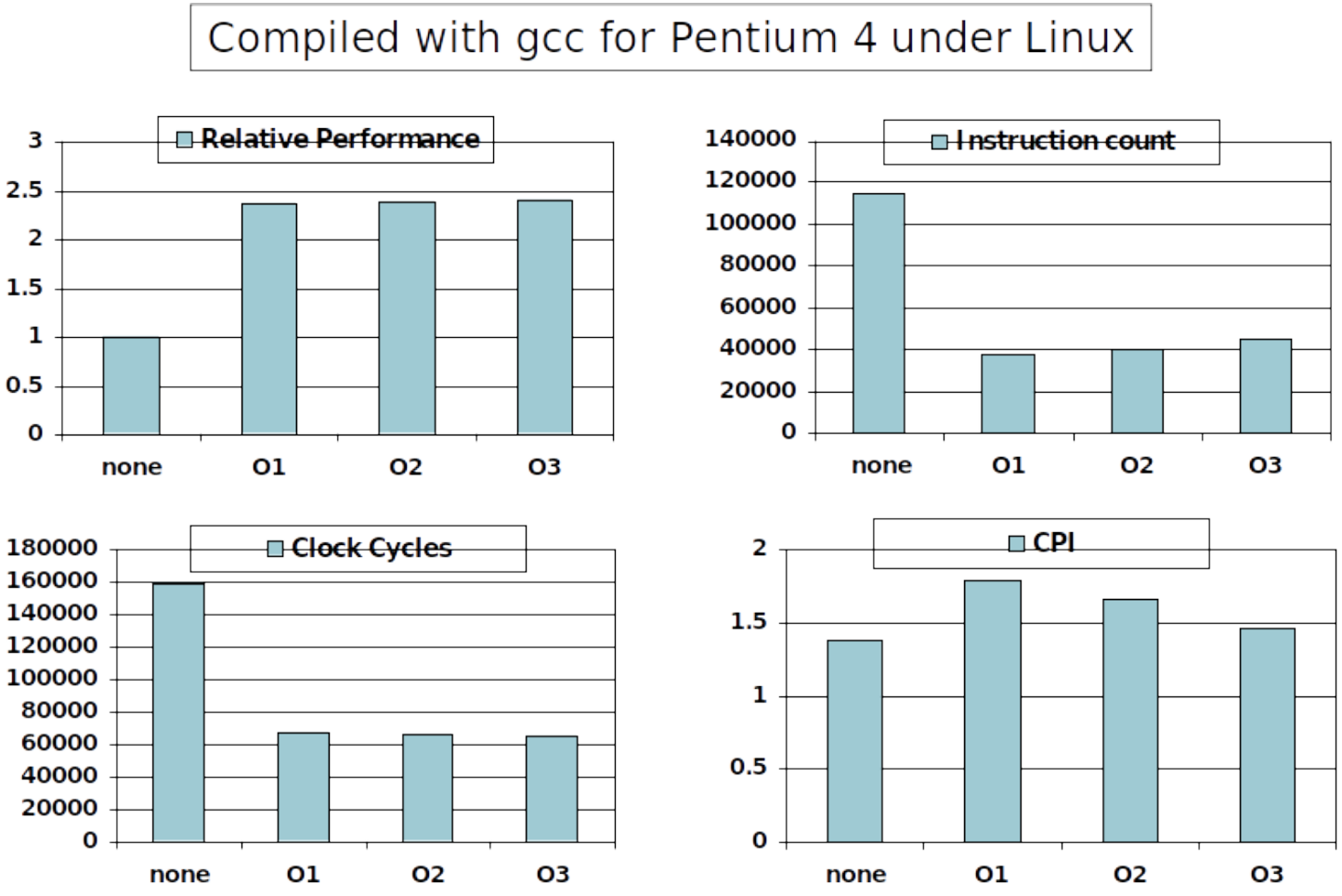
```
1           # Move parâmetros
2           move $s2, $a0           # salva $a0 em $s2
3           move $s3, $a1           # salva $a1 em $s3
4           # Laço externo
5           move $s0, $zero         # i = 0
6   for1tst: slt  $t0, $s0, $s3     # $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
7           # Laço interno
8           beq  $t0, $zero, exit1  # vai para exit1 se $s0 ≥ $s3 (i ≥ n)
9           addi $s1, $s0, -1       # j = i - 1
10  for2tst: slti $t0, $s1, 0       # $t0 = 1 if $s1 < 0 (j < 0)
11          bne  $t0, $zero, exit2  # vai p/ exit2 se $s1 < 0 (j < 0)
12          sll  $t1, $s1, 2        # $t1 = j * 4
13          add  $t2, $s2, $t1      # $t2 = v + (j * 4)
14          lw   $t3, 0($t2)        # $t3 = v[j]
15          lw   $t4, 4($t2)        # $t4 = v[j + 1]
16          slt  $t0, $t4, $t3      # $t0 = 0 if $t4 ≥ $t3
17          beq  $t0, $zero, exit2  # vai p/ exit2 se $t4 ≥ $t3
18          #Passagem de parâmetros e chamada à swap
19          move $a0, $s2           # 1o par de swap é v (era $a0)
20          move $a1, $s1           # 2o par de swap é j
21          jal  swap               # chama swap
22          # Laço interno
23          addi $s1, $s1, -1       # j -= 1
24          j    for2tst            # salta p/ teste do laço interno
25          # Laço externo
26  exit2:  addi $s0, $s0, 1        # i += 1
27          j    for1tst            # salta p/ teste do laço externo
```

Exemplo: Bubble Sort - **bsort** - Casca

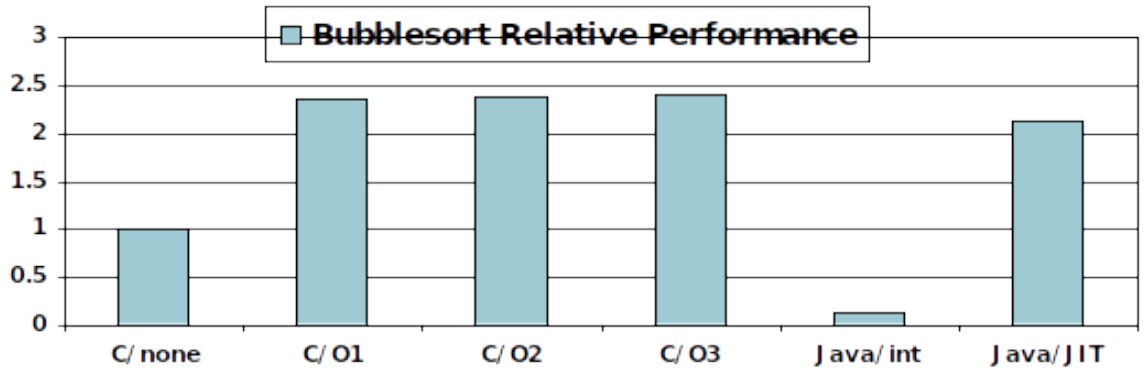
UFABC

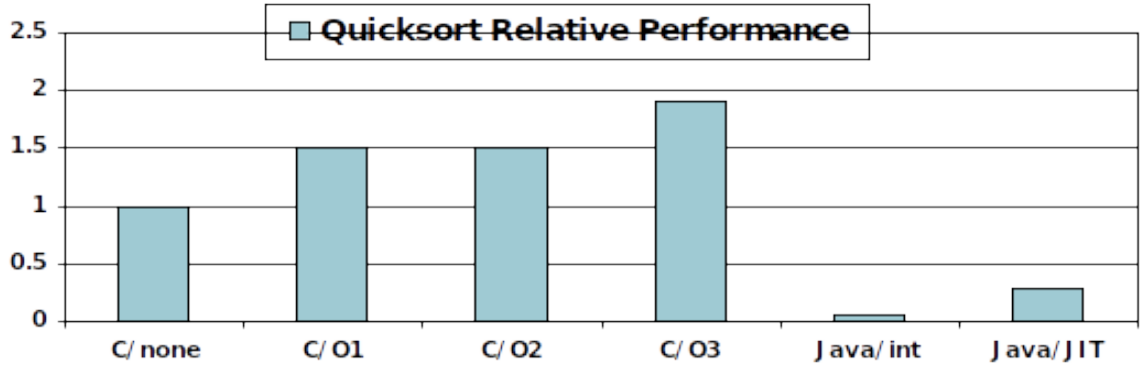
```
1  bsort: addi $sp,$sp, -20        # Reserva espaço na pilha
   ↪  para 5 registradores
2         sw $ra, 16($sp)          # salva $ra na pilha
3         sw $s3,12($sp)           # salva $s3 na pilha
4         sw $s2, 8($sp)           # salva $s2 na pilha
5         sw $s1, 4($sp)           # salva $s1 na pilha
6         sw $s0, 0($sp)           # salva $s0 na pilha
7         …
8         …                        # corpo do bsort
9         …
10        exit1: lw $s0, 0($sp)    # restaura $s0 da pilha
11        lw $s1, 4($sp)           # restaura $s1 da pilha
12        lw $s2, 8($sp)           # restaura $s2 da pilha
13        lw $s3,12($sp)           # restaura $s3 da pilha
14        lw $ra,16($sp)           # restaura $ra da pilha
15        addi $sp,$sp, 20         # restaura stack pointer
16        jr $ra                   # retorna
```

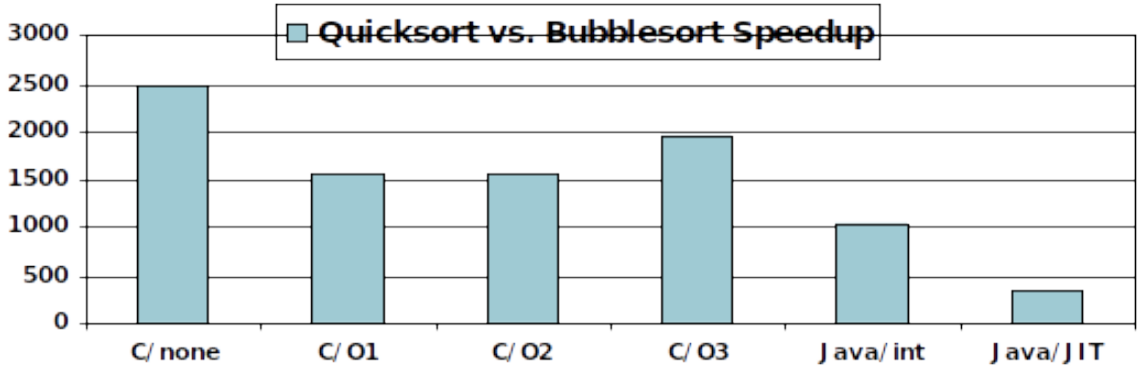
Efeito das otimizações do compilador



Efeito da linguagem e do algoritmo







Lições aprendidas

UFABC

- Número de instruções e CPI não são bons indicadores de desempenho se vistos isoladamente
- Otimizações do compilador são sensíveis ao algoritmo
- Java/JIT é bem mais rápido do que código interpretado
  - Comparável à C em alguns casos
- Nada conserta um algoritmo ruim!

UFABC

- Indexação de arrays envolve
  - Multiplicação do índice pelo tamanho do elemento
  - Adição à base da array
- Ponteiros apontam diretamente para a posição de memória desejada
  - Podem evitar a complexidade envolvida com idexação

Exemplo: Limpando uma Array

UFABC

```
1  clear1(int array[], int size) {
2    int i;
3    for (i = 0; i < size; i += 1)
4      array[i] = 0;
5  }
```

```
1    move $t0,$zero    # i = 0
2  loop1:
3    sll $t1,$t0,2     # $t1 = i * 4
4    add $t2,$a0,$t1   # $t2 =
5                      #   &array[i]
6    sw $zero, 0($t2) # array[i] = 0
7    addi $t0,$t0,1    # i = i + 1
8    slt $t3,$t0,$a1   # $t3 =
9                      #   (i < size)
10   bne $t3,$zero,loop1 # if (…)
11                       # goto loop1
```

```
1  clear2(int *array, int size) {
2    int *p;
3    for (p = &array[0]; p < &array[size]; p = p + 1)
4      *p = 0;
5  }
```

```
1    move $t0,$a0     # p = & array[0]
2    sll $t1,$a1,2    # $t1 = size * 4
3    add $t2,$a0,$t1 # $t2 =
4                    #   &array[size]
5  loop2:
6    sw $zero,0($t0) # Memory[p] = 0
7    addi $t0,$t0,4  # p = p + 4
8    slt $t3,$t0,$t2 # $t3 =
9                    #(p<&array[size])
10   bne $t3,$zero,loop2 # if (…)
11                       # goto loop2
```

Comparando Array *vs.* Ponteiros

UFABC

- Multiplicação acaba virando um shift
- A versão com array necessita que o shift esteja dentro do laço
  - Faz parte do cálculo do índice (ao incrementar o i)
  - ... *vs.* incrementar um ponteiro
- O compilador é capaz de otimizar o código e atingir o mesmo efeito que usando ponteiros diretamente
  - Elimina variáveis
  - O programador faz bem em se aproveitar para tornar o programa mais seguro e simples

Semelhanças entre ARM e MIPS

UFABC

- ARM é o processador embarcado mais popular
- Tem um conjunto de instruções parecido com MIPS

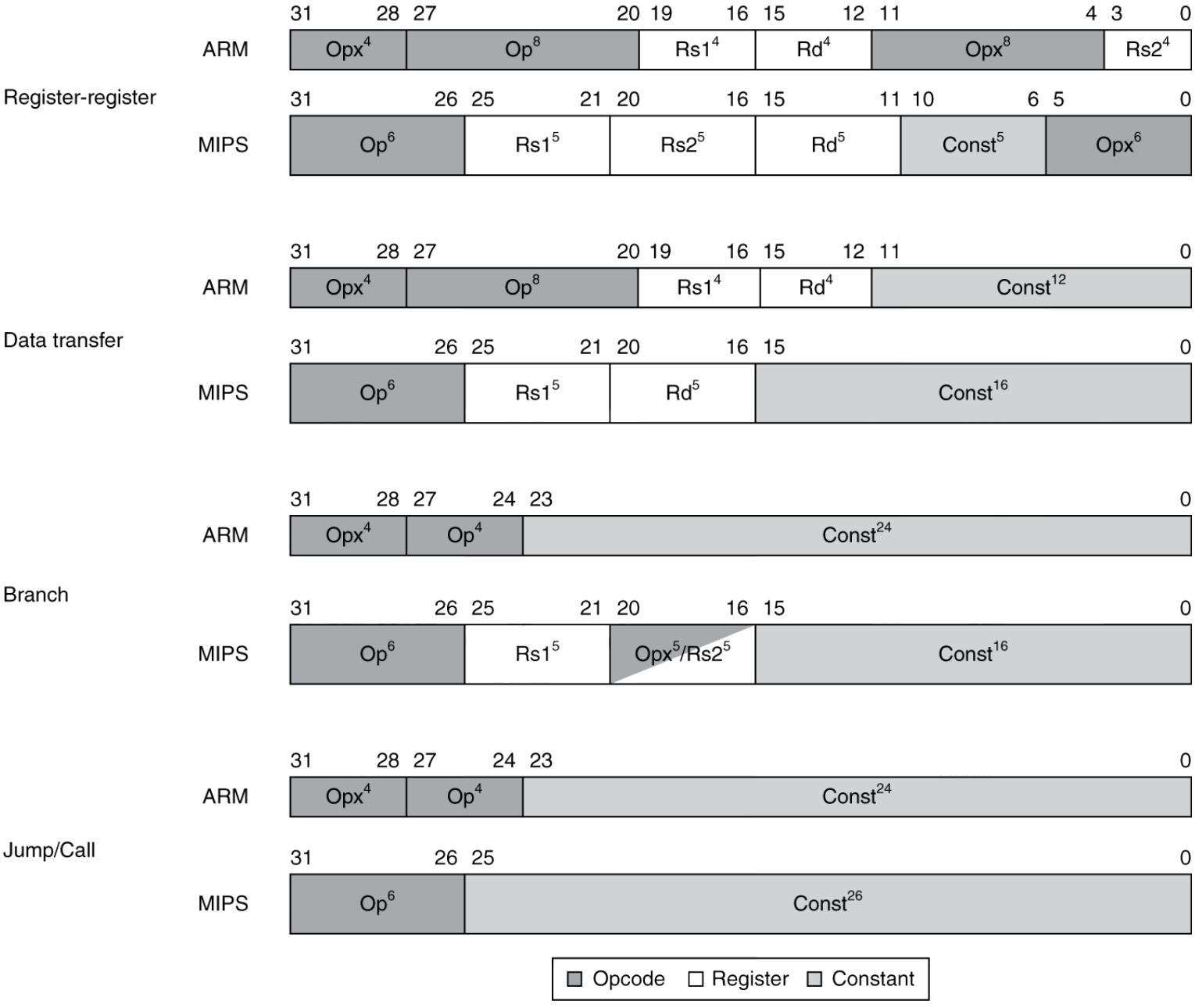
	ARM	MIPS
Ano	1985	1985
Tam. instrução	32 bits	32 bits
Espaço de End.	32 bits (flat)	32 bits (flat)
Alinhamento	Alinhado	Alinhado
Modos de End.	9	3
Registradores	15 x 32 bits	31 x 32 bits
E/S	Mapeado em memória	Mapeado em memória

Compare and Branch no ARM

UFABC

- Usa códigos de condição para resultados de operações lógico/aritméticas
  - Negativo, zero, carry, overflow
  - Compara instruções aos bits de condição sem manter o resultado
- Cada instrução pode ser condicional
  - 4 primeiros bits da instrução: condições para execução
  - Consegue evitar branches em uma única instrução

ARM - Formato das instruções



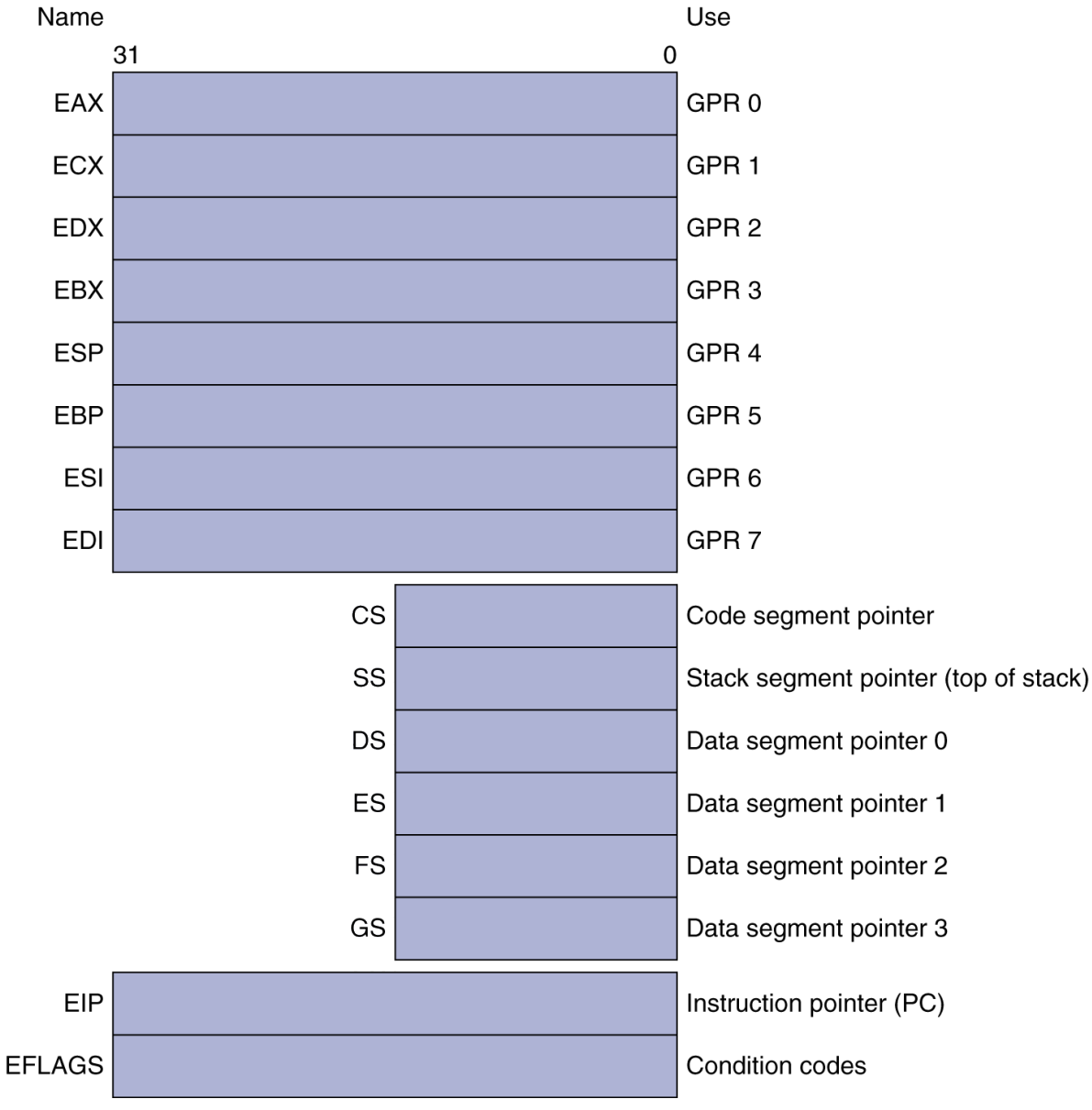
- Evolução mantendo a compatibilidade
  - 8080 (1974): microprocessador 8 bits
    - Acumulador, + 3 pares índice registrador
  - 8086 (1978): extensão 16 bits no 8080
    - Complex instruction set (CISC)
  - 8087 (1980): coprocessador de ponto flutuante
    - Adicionava suporte a instruções de ponto flutuante e pilha de registradores
  - 80286 (1982): endereços de 24 bits, MMU
    - Memória segmentada e protegida
  - 80386 (1985): extensão 32 bits (conhecida hoje como IA-32)
    - Modos adicionais de endereçamento e operação
    - Memória paginada e segmentos

UFABC

- Continuou evoluindo...
  - i486 (1989): pipelined, on-chip caches e FPU
    - Competidores com compatíveis: AMD, Cyrix, ...
  - Pentium (1993): superescalar, datapath de 64 bits
    - Versões seguintes adicionaram a extensão MMX (Multi-Media eXtension)
    - Teve o infame bug FDIV
  - Pentium Pro (1995), Pentium II (1997)
    - Nova microarquitetura (veja Colwell, The Pentium Chronicles)
  - Pentium III (1999)
    - Adicionou SSE (Streaming SIMD Extensions) e novos registradores associados
  - Pentium 4 (2001)
    - Nova microarquitetura
    - Adicionou SSE2

- Mais um pouquinho...
  - AMD64 (2003): estendeu a arquitetura para 64 bits
  - EM64T – Extended Memory 64 Technology (2004)
    - A Intel adota o AMD64 (com alguns refinamentos)
    - Adiciona SSE3
  - Intel Core (2006)
    - Adiciona SSE4, suporte a máquinas virtuais
  - AMD64 (2007): SSE5
    - Intel decidiu não seguir a AMD e acabou lançando..
  - Advanced Vector Extension (2008)
    - Registradores SSE mais longos e mais instruções
  - ...
- Se a Intel não estendesse a compatibilidade, seus competidores o fariam!
  - Elegância técnica ≠ sucesso no mercado

Registradores Intel x86 (Básicos)



Intel x86 - Modos de endereçamento básicos

UFABC

- Dois operandos por instrução

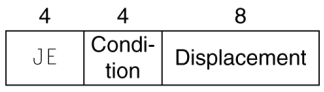
Origem/Dest	Origem
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

- Modos de endereçamento de memória
  - Endereço no registrador
  - Endereço = R_{Base} + Deslocamento
  - Endereço = $R_{Base} + 2^{escala} \times R_{índice}$ (escala = 0, 1, 2 ou 3)
  - Endereço = $R_{Base} + 2^{escala} \times R_{índice}$ + Deslocamento

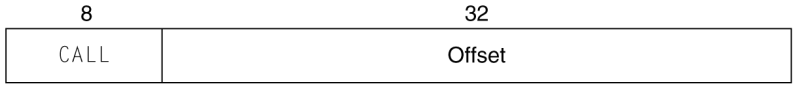
Formato das instruções Intel x86

UFABC

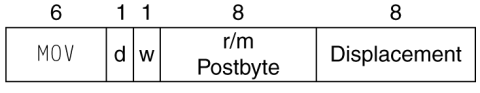
a. JE EIP + displacement



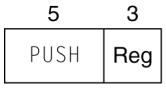
b. CALL



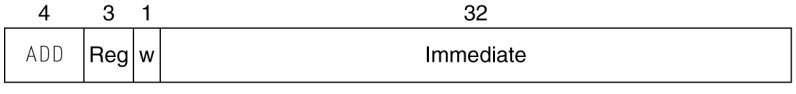
c. MOV EBX, [EDI + 45]



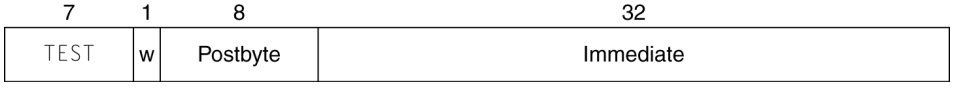
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



- Formato de comprimento variável
  - Bytes sufixos indicam modo de endereçamento
  - Bytes prefixos modificam a operação
    - Comprimento do operando, repetição, travas, ...

UFABC

- Conjunto de instruções complexo dificulta a implementação
  - O hardware traduz instruções complexas para microoperações mais simples
    - Instruções simples 1-1
    - Instruções complexas 1-muitas
    - Mecanismo de execução parecido com RISC
    - A fatia de mercado da Intel possibilita que isso seja viável economicamente
  - Desempenho comparável com RISC
    - Compiladores acabam evitando instruções complexas

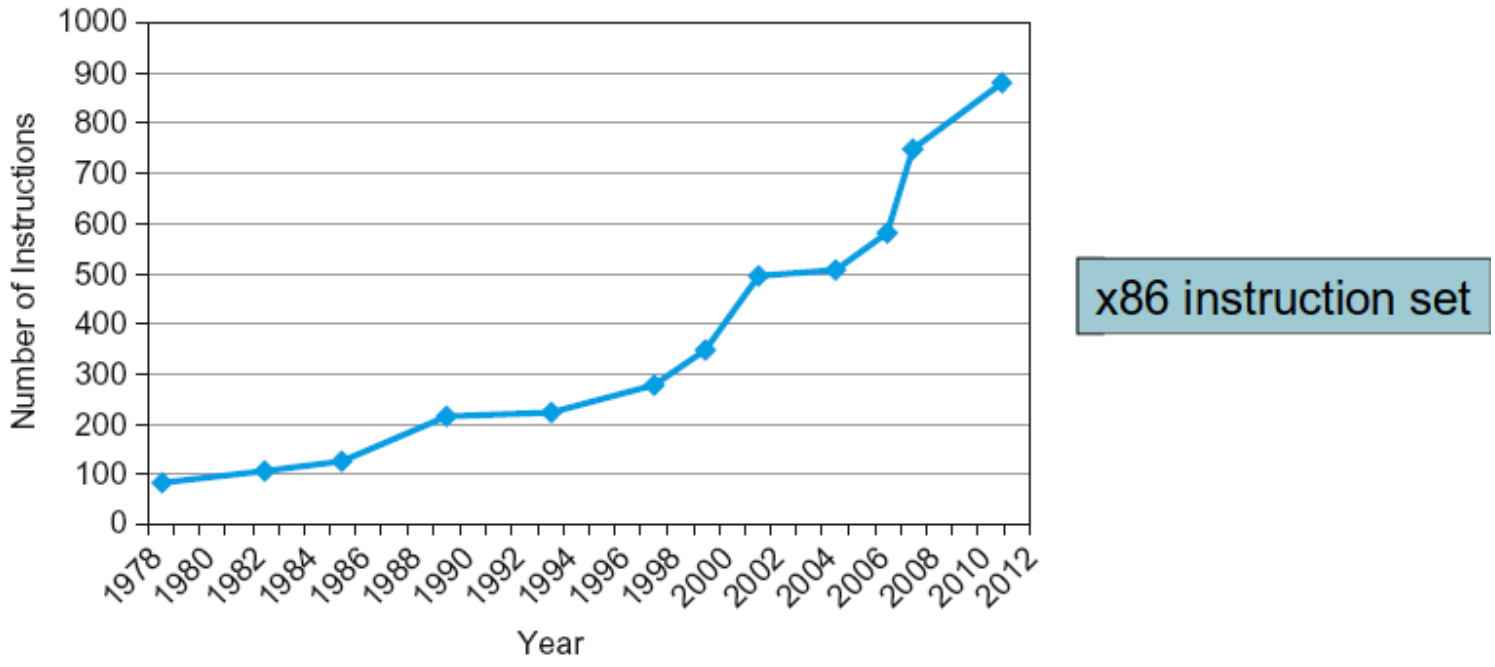
- Quando lançou a versão 64 bits, ARM fez uma reforma completa
- ARMv8 é muito parecido com MIPS
- Mudanças de ARMv7:
  - Não há campos de execução condicional
  - Campo de imediato é uma constante de 12 bits
  - Descartaram operações de load/store múltiplas
  - PC não é mais um GPR
  - 32 GPRs
  - Modos de endereçamento funcionam para todos os tamanhos de palavra
  - Divide instruction
  - Instruções **beq** e **bne**

Falácias

- Instrução poderosa $\rightarrow$ melhor desempenho
  - Precisa de menos instruções
  - Mas instruções complexas são difíceis de implementar
    - Podem atrasar outras instruções, incluindo as simples
  - Compiladores são bons em criar código rápido usando instruções simples
- Use código assembly para códigos mais rápidos
  - Mas compiladores modernos são melhores para lidar com processadores modernos
  - Mais linhas de código $\rightarrow$ mais erros e menos produtividade

UFABC

- Compatibilidade reversa → conjunto de instruções não muda
  - mas ele acumula cada vez mais instruções!



Armadilhas

UFABC

- Palavras sequenciais não tem endereços sequenciais
  - Incrementam de 4 em 4, e não de 1 em 1
    - Cada palavra tem 4 bytes
- Manter um ponteiro para uma variável automática depois que um procedimento retorna
  - Por exemplo, passar um ponteiro de volta como um argumento
  - O ponteiro se torna inválido quando a desempilhamos a pilha

Conclusões

UFABC

- Princípios de design
  - Simplicidade favorece a regularidade
  - Menor é mais rápido
  - Faça o caso mais comum ser rápido
  - Um bom design exige alguns bons compromissos
- Camadas de software/hardware
  - Compilador, assembler, hardware
- MIPS é um exemplo típico de ISAs RISC
  - Em contraste com Intel x86

UFABC

- Medida de instruções MIPS em benchmarks
  - Considere fazer o caso comum rápido
  - Considere os compromissos

Classe	Exemplos MIPS	SPEC2006 Int	SPEC2006 FP
Aritmética	`add`, `sub`, `addi`	16%	48%
Tranf. Dados	`lw`, `sw`, `lb`, `lbu`, `lh`, `lhu`, `sb`, `lui`	35%	36%
Lógicas	`and`, `or`, `nor`, `andi` `ori`, `sll`, `srl`	12%	4%
Branchs Cond.	`beq`, `bne`, `slt`, `slti`, `sltiu`	34%	8%
Saltos	`j`, `jr`, `jal`	2%	0%