Conjunto de Instruções e Arquitectura

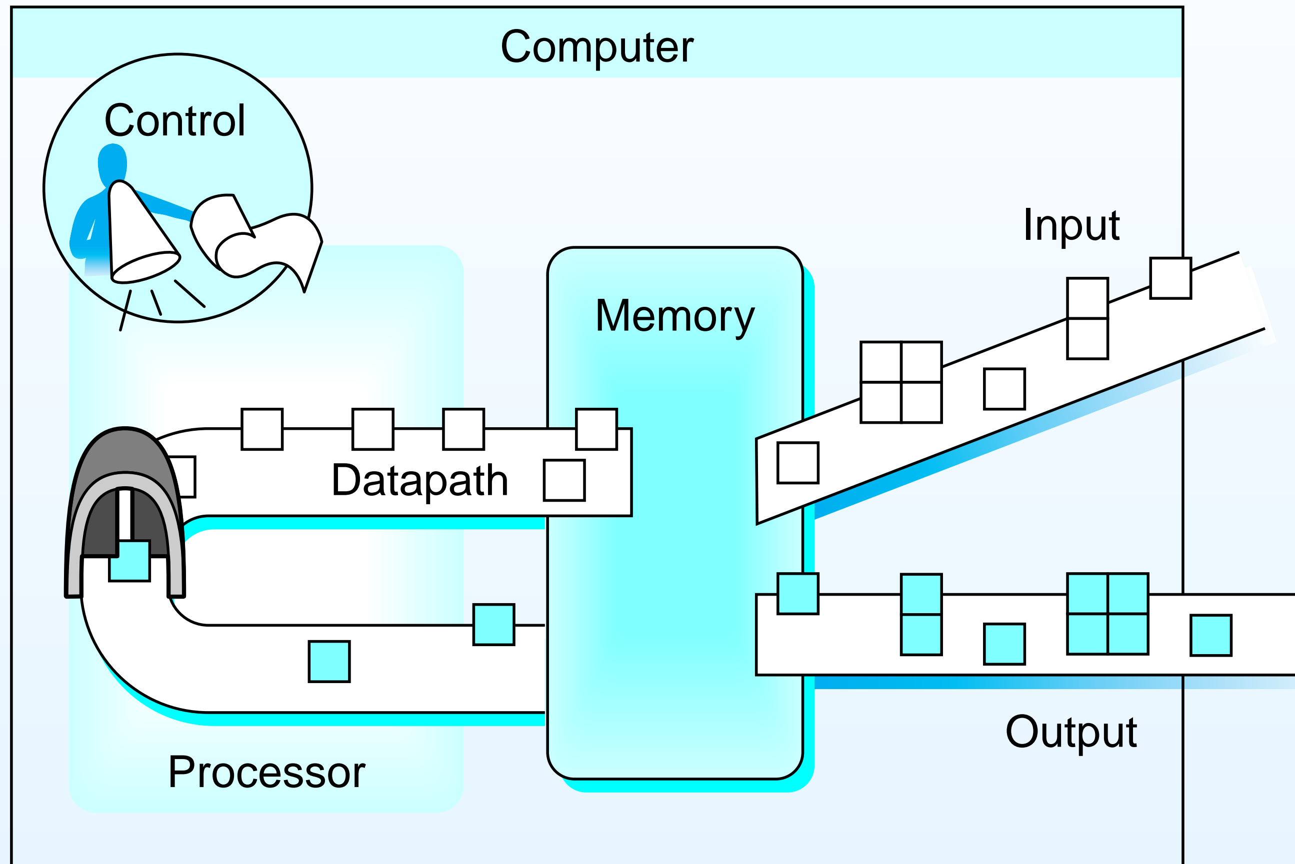
Luís Nogueira

luis@dei.isep.ipp.pt

Departamento Engenharia Informática

Instituto Superior de Engenharia do Porto

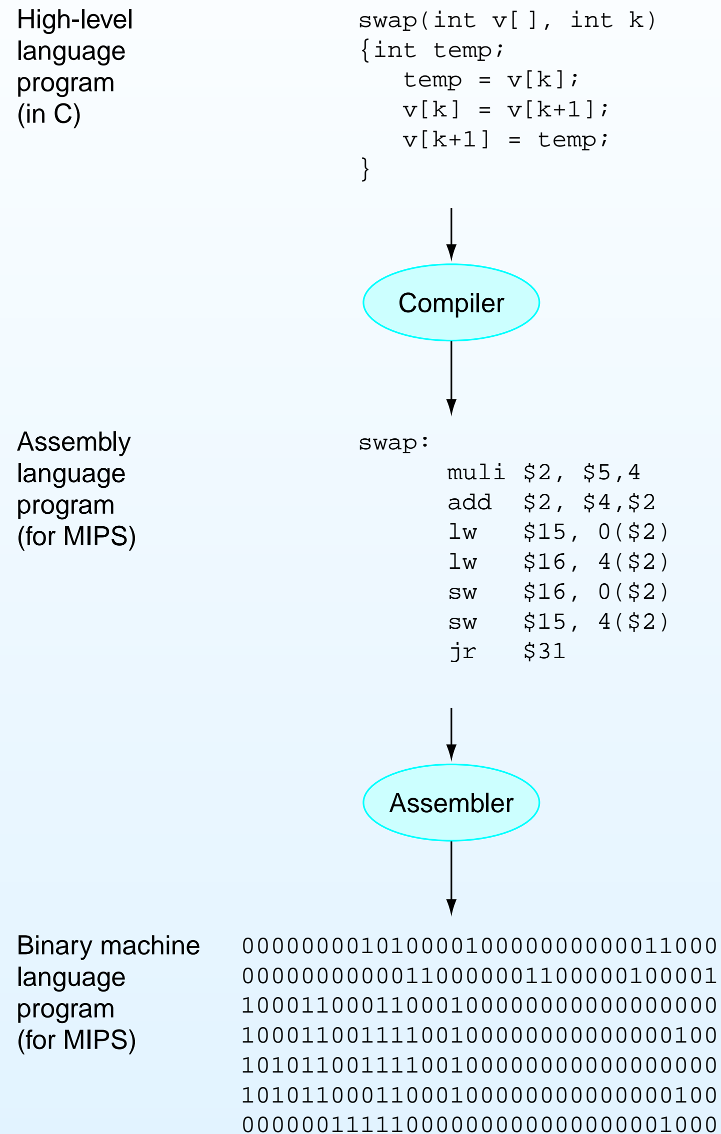
Organização de um computador



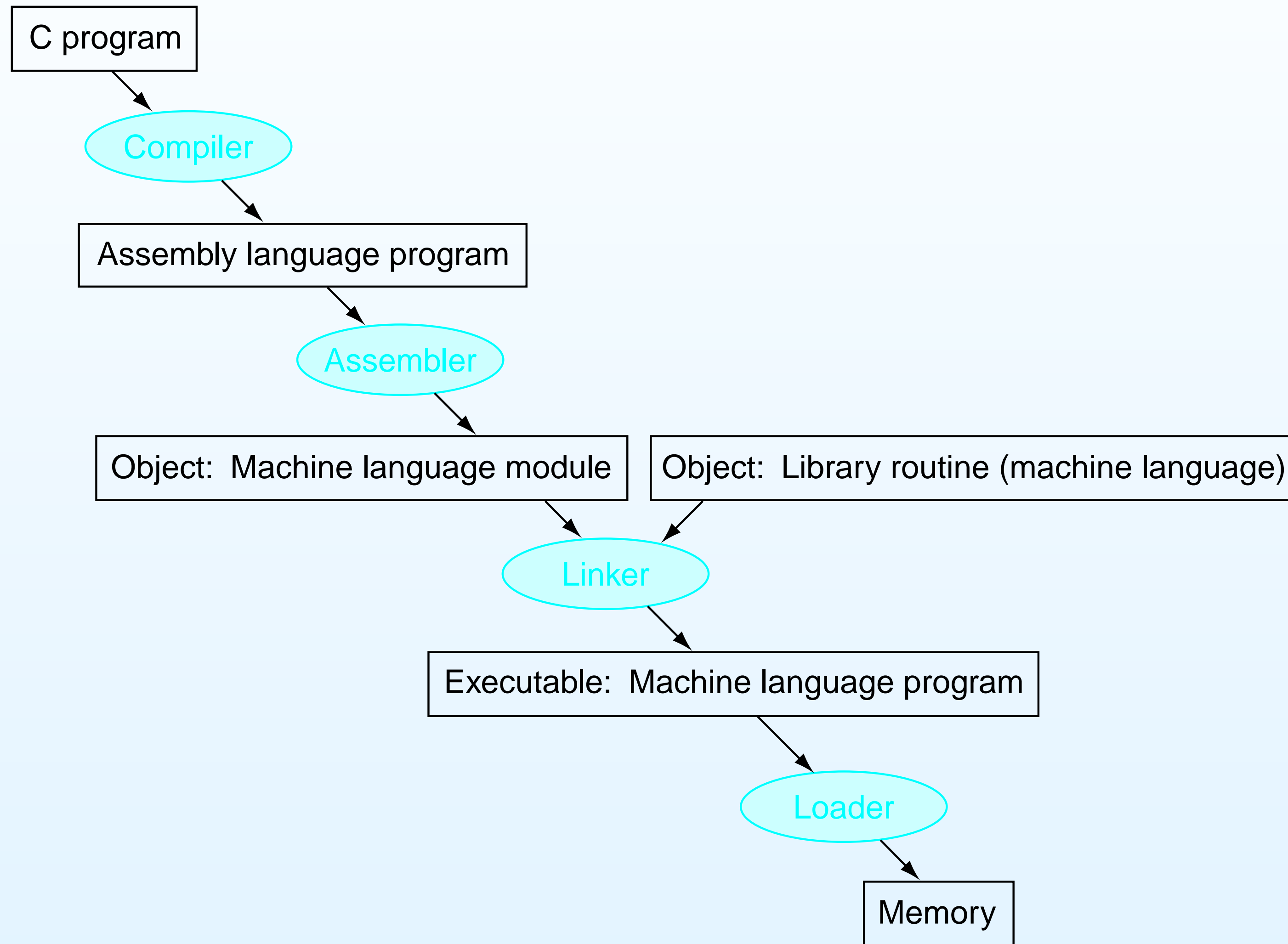
Como controlar o hardware?

- É necessário falar a sua linguagem
  - Instruções são o vocabulário do CPU
- Instruction Set Architecture (ISA)
  - Interface entre hardware e software
  - Exemplos: MIPS, SPARC, PA-RISC, ix86, Alpha
- Mas é frequente programar-se a mais alto nível
- Tradução linguagem alto nível $\longrightarrow$ linguagem do hardware
  - Compilador `programa linguagem X` $\longrightarrow$ `programa assembly`
  - Assemblador `programa assembly` $\longrightarrow$ `binário ISA`

Linguagem alto nível ⟶ linguagem do hardware



Traduzir e iniciar um programa



Instruction Set Architecture (ISA)

- Como desenhar um processador?
  - Datapath
  - Lógica de controlo
  - Organização da memória
  - Circuito de relógio
  - . . .
- Duas abordagens fundamentais
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)

Complex Instruction Set Computer (CISC)

- Objectivo
  - Reduzir fosso entre linguagens de programação de alto nível e hardware
  - Fornecendo instruções e modos de endereçamento complexos
- Cada instrução pode indicar conjunto de sub-operações
- Dificulta desenho do processador
  - Descodificação complexa de instruções
  - Maior número de circuitos
- Exemplos
  - VAX, Motorola 68000, Intel ix86

Reduced Instruction Set Computer (RISC)

- Algumas características dos processadores CISC não estavam a ser usadas pelos programas
  - Complexidade desnecessária do hardware
- Aumento do fosso entre velocidade do CPU e memória
  - Necessário diminuir nº de acessos à memória
- Filosofia RISC
  - Simplificar desenho do processador
  - Colocar a complexidade nos compiladores

Reduced Instruction Set Computer (RISC)

- Menor nº de instruções
- Instruções simples
- Hardware mais simples e rápido
  - Ciclo de relógio é limitado pela instrução mais lenta
- Elevado número de registos
  - Menos acessos à memória
- Arquitectura load/store
  - Só duas instruções acedem à memória
- Exemplos
  - ARM, DEC Alpha, SPARC, MIPS, PowerPC

MIPS

- Microprocessor without Interlocked Pipeline Stages
- Arquitectura RISC desenvolvida pela *MIPS Computer Systems*
- Bastante aceitação em sistemas embebidos, dispositivos Windows CE, routers Cisco e workstations SGI
  - 1 em cada 3 chips RISC são baseados em MIPS
  - Nintendo 64, PlayStation 1 e 2
- Inicialmente a 32 bits, actualmente arquitectura de 64 bits
- Conjunto de instruções bastante simples

Instruções Aritméticas e Lógicas

Aritméticas	
`add $1,$2,$3`	`$1= $2 + $3`
`sub $1,$2,$3`	`$1= $2 - $3`

Lógicas	
`and $1,$2,$3`	`$1= $2 & $3`
`or $1,$2,$3`	`$1= $2 | $3`
`slt $1,$2,$3`	`$1= ($2 < $3) ? 1:0`

- Todos os operandos em registos
- Todas as operações têm 3 argumentos
  - Simplicidade no hardware

> 1º Princípio de Desenho de Hardware
> Simplicidade favorece a regularidade

Instruções Aritméticas e Lógicas

C	Compilador	MIPS
`a = b + c;`	a→\$1, b→\$2	`add $1,$2,$3`
`d = a - e;`	c→\$3, d→\$4	`sub $4,$1,$5`
	e→\$5	

- Compilador atribui um registo a cada variável

C	Compilador	MIPS
`f=(g+h)-(i+j);`	g→\$1, h→\$2	`add $3,$1,$2`
	i→\$4, j→\$5	`add $6,$4,$5`
	\$3→temporário	`sub $7,$3,$6`
	\$6→temporário	
	f→\$7	

Número limitado de registos

- MIPS tem 32 registos

2º Princípio de Desenho de Hardware
Mais pequeno é mais rápido

- Muitos registos aumentam o ciclo de relógio
  - Sinais eléctricos demoram mais se tiverem que percorrer maior distância
  - Balancear necessidade de mais registos com velocidade do ciclo de relógio
- Como representar estruturas complexas?
  - Acedendo à memória

Instruções de acesso à memória

- Permitem leitura/escrita de dados da/na memória
- Memória é um array uni-dimensional
  - Endereço inicial + deslocamento
- Ler dados da memória

```
lw $1,X($2)
```

  - Lê conteúdo da memória em `$2+X` para `$1`
- Escrever dados na memória

```
sw $1,X($2)
```

  - Guarda na memória em `$2+X` conteúdo de `$1`

Instruções de acesso à memória

C	Compilador	MIPS
`a[30] = b + a[30]`	`$2`→`endereço a[]`	`lw $9,120($2)`
	`$9`→`temporário`	`add $9,$3,$9`
	`$3`→`b`	`sw $9,120($2)`

- Compilador associa vectores a endereços de memória
  - Endereço inicial usado nas instruções lw/sw
- Compilador mantém variáveis mais usadas nos registos
  - Registos mais rápidos que memória
  - Instruções operam sobre registos

Instruções imediatas

- Constantes são frequentes nos programas
- Operações aritméticas melhoram performance se as puderem aceder directamente

```
addi $1,$2,5
```
$1 = $2 + 5

- alternativa. . .

```
lw $3,AddrConstant5($6)
add $1,$2,$3
```
$3 = 5

$1 = $2 + $3

3º Princípio de Desenho de Hardware
Optimizar casos frequentes

Instruções de salto

- Permitem alterar fluxo do programa
  - Alteram normal incremento do Program Counter (PC)
- Salto incondicional absoluto

  `j L` PC = L

- Salto incondicional por registo

  `jr $1` PC = $1

- Salto condicional

  `beq $1,$2,L` PC = ($1 == $2) ? PC + 4 + L : PC + 4

  `bne $1,$2,L` PC = ($1 != $2) ? PC + 4 + L : PC + 4

Instruções de salto

C	Compilador	MIPS
`if(i != j)`	i→$1, j→$2	`    beq $1,$2,L`
`  f = g + h;`	f→$3, g→$4	`    add $3,$4,$5`
`f = f - i;`	h→$5	`L:sub $3,$3,$1`

C	Compilador	MIPS
`if(i == j)`	i→$1, j→$2	`      bne $1,$2,ELSE`
`  f = g + h;`	f→$3, g→$4	`      add $3,$4,$5`
`else`	h→$5	`      j EXIT`
`  f = f - i;`		`ELSE: sub $3,$3,$1`
		`EXIT:`

Instruções de salto

C	Compilador	MIPS
`while(i != k){`	`i`$\longrightarrow$`$1`	`LOOP:beq $1,$3,EXIT`
`j = j + i;`	`j`$\longrightarrow$`$2`	`add $2,$2,$1`
`i = i + 1;`	`k`$\longrightarrow$`$3`	`addi $1,$1,1`
`}`		`j LOOP`
		`EXIT:`

C	Compilador	MIPS
`while(a[i]==k){`	`i`$\longrightarrow$`$3,k`$\longrightarrow$`$5`	`LOOP:sll $1,$3,2`
`i = i + 1;`	`$6`$\longrightarrow$`addr a[]`	`add $1,$1,$6`
`}`	`$1`$\longrightarrow$`temporário`	`lw $7,0($1)`
	`$7`$\longrightarrow$`temporário`	`bne $7,$5,EXIT`
		`addi $3,$3,1`
		`j LOOP`
		`EXIT:`

Instruções de chamada a procedimentos

- Executar rotinas em diferentes pontos do programa
- Informação mantida dinâmicamente na pilha
  - Retorno da chamada (registo `$31`)
  - Pilha apontada pelo registo `$30`
- Quem guarda os registos?
  - Caller saves
    - Antes da chamada: registos $\longrightarrow$ pilha
    - Após retorno: pilha $\longrightarrow$ registos
  - Calle saves
    - Início da rotina: registos $\longrightarrow$ pilha
    - Antes de retornar: pilha $\longrightarrow$ registos
- Salto para procedimento

  `jal L` $31 = PC; PC = L

Instruções de chamada a procedimentos

```
A:   ...
     ...
     jal B                 $31→ retorno A
     ...
B:   ...
     sw $31,0($30)         $31→pilha
     addi $30,$30,4        incrementar apontador pilha
     jal C                 $31→ retorno B
     lw $31,0($30)         pilha→$31
     ...
     jr $31                retorno A
C:   ...
     ...
     jr $31                retorno B
```

Modos de endereçamento

- Por registo
  - Todos os operandos são registos

    `add $1,$2,$3` $1 = $2 + $3

- Relativo ao PC
  - Últimos 16 bits adicionados ao PC

  `beq $1,$2,L` PC = ($1 == $2) ? PC + 4 + L : PC + 4

- Por descolamento
  - Registo (endereço base) + deslocamento 16 bits

  `lw $1,A($2)` $1 = conteúdo(A + 2)

Modos de endereçamento

- Modo imediato (ou constante)
  - Um operando constante

    `addi $1,$2,5` $1 = $2 + 5
- Absoluto ao PC
  - Últimos 26 bits atribuídos ao PC `j L` PC = L

Representar instruções no CPU

- Necessário traduzir instruções
  - Série de sinais eléctricos 0 ou 1
- Instrução dividida em diversos campos
  - Organização dos campos define formato da instrução
- Todas as instruções com 32 bits de tamanho
  - Facilita descodificação
  - Favorece regularidade

Tradução de MIPS em linguagem máquina

- Formato R (*register format*)
  - `opcode` código da instrução
  - `rs,rt,rd` registos
  - `shamt` shift amount
  - `funct` variante da instrução

opcode	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Exemplo: `add $1,$2,$3`

000000	00010	00011	00001	00000	100000

Tradução de MIPS em linguagem máquina

- Necessidade de operandos maiores que 2^5 (ex: `lw/sw`)
- Conflito: Mesmo tamanho vs Mesmo formato

4º Princípio de Desenho de Hardware
Um bom desenho exige bons compromissos

- Compromisso
  - Todas as instruções do mesmo tamanho
  - Diferentes formatos nos operandos
  - Maximizar equivalência dos formatos

Tradução de MIPS em linguagem máquina

- Formato I (*instruction format*)
  - `opcode` código da instrução
  - `rs,rt` registos
  - `addr` constante ou endereço

opcode	rs	rt	addr
6 bits	5 bits	5 bits	16 bits

- Formatos I e R similares
  - 3 primeiros campos são iguais
  - `addr` mesmo tamanho que `rd+shamt+funct`
- Formatos são distinguidos pelo valor do primeiro campo

Tradução de MIPS em linguagem máquina

C	Compilador	MIPS
`a[300] = b + a[300]`	`$2`→`endereço a[]`	`lw $9,1200($2)`
	`$9`→`temporário`	`add $9,$3,$9`
	`$3`→`b`	`sw $9,1200($2)`

100011	01001	00010	0000 0100 1011 0000		
000000	00010	00011	00001	00000	100000
101011	01001	00010	0000 0100 1011 0000		

- lw/sw apenas diferem num bit
- Semelhança entre instruções facilita desenho do hardware

Tradução de MIPS em linguagem máquina

C	Compilador	MIPS
`a[300] = b + a[300]`	`$2`→`endereço a[]`	`lw $9,1200($2)`
	`$9`→`temporário`	`add $9,$3,$9`
	`$3`→`b`	`sw $9,1200($2)`

100011	01001	00010	0000 0100 1011 0000		
000000	00010	00011	00001	00000	100000
101011	01001	00010	0000 0100 1011 0000		

- lw/sw apenas diferem num bit
- Semelhança entre instruções facilita desenho do hardware

Tradução de MIPS em linguagem máquina

- Formato J (*jump format*)
  - `opcode` código da instrução
  - `addr` endereço

opcode	addr
6 bits	26 bits

- Exemplo: `j 1200`

000010	0000 0100 1011 0000

Intel IA-32

- Intel lança processador 16 bits dois anos antes do Motorola 68000
  - 8086 escolhido para o IBM PC
  - Evolução condicionada pela retro-compatibilidade
- Objectivo da arquitectura
  - Reduzir nº de instruções num programa
  - Origina instruções complexas
- Instruções aritméticas, lógicas e acesso a memória
  - Diversas combinações nos tipos de operandos
  - Um dos operandos pode estar em memória
  - Destino pode ser a memória

Intel IA-32

- Diversos modos de endereçamento
  - Complexos (Base + Scale Index)
  - Restrições nos registos usados nos diversos modos
- Formato das instruções complexo
  - opcode indica se operando de 8 ou 32 bits
  - Instruções com diferentes tamanhos

Intel IA-32

- **1978** Intel 8086, 16 bits, registos de uso dedicado
- **1982** Intel 80286, 24 bits, adiciona instruções e complexo mapeamento de memória
- **1985** Intel 80386, 32 bits, novos modos de endereçamento e instruções
- **1989-95** 80486, Pentium e Pentium Pro, maior performance, adicionadas 4 instruções
- **1997** MMX para Pentium e Pentium Pro, 57 novas instruções

Intel IA-32

- **1999** SSE com 70 instruções, novos registos de 128 bits
- **2001** SSE2 com 144 instruções para operações de virgula flutuante em paralelo
- **2003** AMD64, 64 bits, modo retro-compatibilidade
- **2004** Intel EM64T, adicionadas instruções 128 bits e SSE3 com 13 instruções

Intel IA-32

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

Resumo

- Selecção do conjunto de instruções
  - Impacto no desenho do hardware
- Balanço delicado
  - Número de instruções fornecidas
  - Formato das instruções
  - Número de ciclos de relógio por instrução
  - Velocidade do ciclo de relógio

Quatro princípios de desenho do hardware

- Simplicidade favorece a regularidade
- Mais pequeno é mais rápido
- Optimizar casos frequentes
- Um bom desenho exige bons compromissos