Organização de Computadores

Siang W. Song

(Material baseado parcialmente no livro de A. S. Tanenbaum – *Structured Computer Organization*, third edition, Prentice-Hall, 1990.)

1 Organização de um computador

- Processador ou CPU
- Memória
- Dispositivos de entrada/saída (E/S)

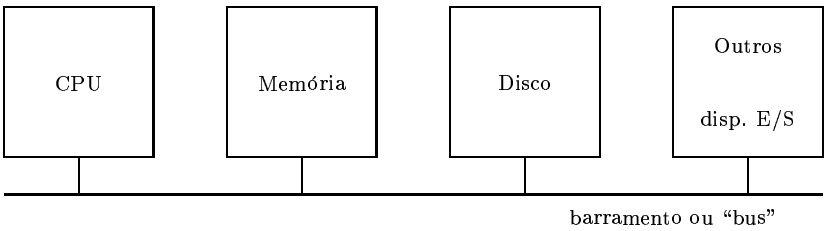


Figura 1: Organização de um computador

1.1 Processador ou CPU

Tem a função de

- buscar instruções da memória
- examinar ou decodificar a instrução
- executar a instrução

São três as partes principais da CPU:

- Unidade de controle: busca a instrução da memória e decodifica-a.
- ALU (ou unidade aritmética e lógica): realiza operações aritméticas e booleanas.
- Registradores: memória rápida para guardar informações de controle, resultados intermediários.

Alguns registradores importantes incluem: PC ("Program Counter") - registrador que contém o endereço de memória da próxima instrução a ser executada. IR ("Instruction Register") - registrador que contém a instrução em execução. PS ou PSW ("program status word") - registrador que contém uma série de informações de controle, dando o estado do processador (o conteúdo varia de máquina para máquina).

Ciclo "busca-decodifica-executa" ("fetch-decode-execute") A CPU realiza repetidamente o seguinte ciclo chamado "fetch-decode-execute":

1. Busca a instrução (apontada por PC) da memória e carrega-a no IR.
2. Muda o PC para apontar para a próxima instrução da memória.
3. Decodifica a instrução, determinando o seu tipo, operandos, etc.
4. Se a instrução usa operandos (dados) da memória, determina os seus endereços.
5. Busca os dados de memória e carrega-os nos registradores.
6. Executa a instrução.
7. Armazena resultados (em registradores ou memória).

Responda: o que o passo 6 deve fazer se a instrução é do tipo goto endereço?

Conjunto de instruções ou "instruction set" Denomina-se "instruction set" de uma máquina o conjunto de instruções disponíveis ao programador no nível de máquina convencional. Tipicamente o conjunto possui de 20 a 300 instruções. Muitas vezes essas instruções são implementadas como microprogramas (a ser vistos mais tarde).

Em particular, as chamadas arquiteturas RISC ("reduced instrucion set computer") possuem um pequeno conjunto de instruções simples e rápidas. Veremos também tais arquiteturas mais tarde.

Organização da CPU "Data path" de uma CPU típica é mostrado na Figura 2 e inclui a ALU (Unidade aritmética-lógica), os registradores e as interconexões.

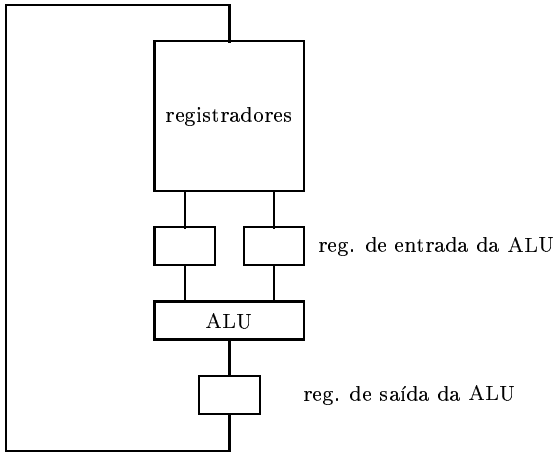


Figura 2: "Data path" de uma CPU

Vários tipos de instruções são possíveis:

- registrador-memória
- registrador-registrador
- memória-memória

Execução paralela de instruções Há diversos motivos para se desejar ter máquinas paralelas de diversas CPU's ou mesmo CPU com várias ALU's.

Limite físico: velocidade da luz no vácuo (30 cm/ns) ou velocidade de um sinal elétrico no cobre (20 cm/ns). Assim, para ter instruções de 1 ns, por exemplo, a distância dos sinais percorridos, entre a CPU e a memória deve ser menor que 20 cm.

Outro problema é a dissipação do calor. Circuitos de alta velocidade geram grande quantidade de calor. Tanto menor é o volume dos circuitos, mais difícil é a dissipação do calor gerado. Frequentemente os processadores são mergulhados em líquido (água, freon, ou nitrogênio líquido).

Podemos, por outro lado, obter maiores computações com arquiteturas com várias CPU's, ou CPU com várias ALU's.

Segundo Flynn(72), há várias categorias de máquinas paralelas:

- SISD: Single Instruction Single Data
- SIMD: Single Instruction Multiple Data
- MIMD: Multiple Instruction Multiple Data

A máquina tradicional de von Neumann é SISD. Mesmo nesse modelo sequencial, algum grau de paralelismo é possível.

Por exemplo, o CDC 6600 possui ALU com 6 unidades funcionais (Figura 3).

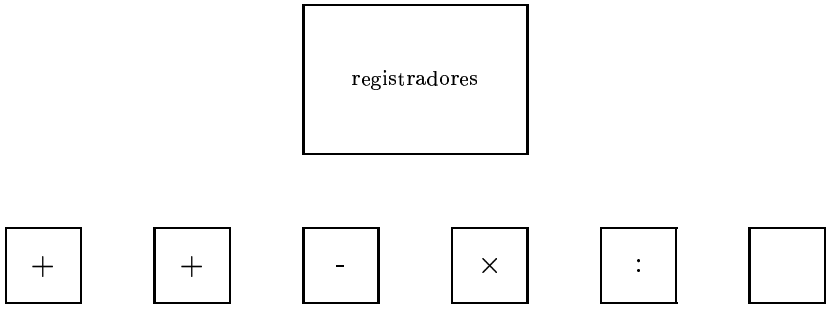


Figura 3: ALU do CDC 6600

Uma variante dessa idéia é a máquina "pipeline". A execução de uma instrução é desdobrada em vários estágios, cada um executado numa unidade independente (como numa linha de montagem). Mostramos um exemplo de 5 estágios:

Se cada unidade leva n ns, uma instrução leva $5n$ ns para ser executada. Porém, se todas as unidades podem ser mantidas sempre ocupadas, então é possível completar uma instrução em cada n ns.

A máquina pipeline é ainda do tipo SISD, pois há um só programa e um conjunto de entrada de dados.

```
   P1         P2         P3         P4         P5

busca      analisa      calcula    busca      executa
instr.     instr.       enderecos  dados      instr.

---------------------------------------------------------->
                                                      tempo
instr.
    1
    2          1
    3          2          1
    4          3          2          1
    5          4          3          2          1
```

A ALU pode ser do tipo pipeline e temos os chamado processador vetorial. A ALU tem múltiplas unidades cada uma realizando um estágio de uma operação aritmética (Figura 4).

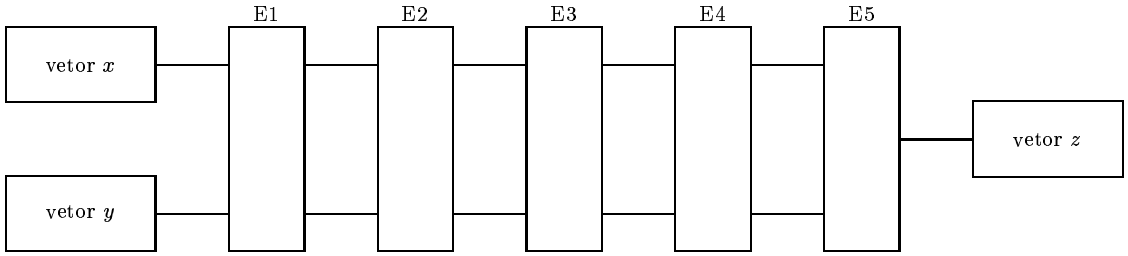


Figura 4: Vários estágios de um processador vetorial

Tal tipo de máquina é adequada para realizar computações envolvendo vetores. Muitos supercomputadores possuem a processadores vetoriais, com instruções vetoriais do tipo

$z = x + y$, onde x, y e z sao vetores.

Nesse caso, o compilador desempenha um importante papel de descobrir em "loops" de programas quais as instruções ou conjunto de instruções que podem ser executados com instruções vetoriais da máquina.

1.2 Memória

Usando um endereço de m bits, pode-se endereçar 2^m posições de memória ou célula de memória. Célula é a menor unidade endereçável de uma máquina. O tamanho de uma célula varia de cada máquina. Por exemplo,

IBM PC	célula de 8 bits
DEC PDP-8	célula de 12 bits
IBM 1130	célula de 16 bits
CDC Cyber	célula de 60 bits

Computadores modernos adotam células de 8 bits (1 byte).

Palavra (word) é um agrupamento de bytes formando uma unidade para a maioria das operações aritméticas.

Códigos de deteção e correção de erros Para uma referência bibliográfica sobre este assunto, ver

- V. Pless, *Introduction to the theory of error-correcting codes*, John Wiley and Sons, 1982.

Como prevenção contra erros ocasionais de memória, devidos a problemas de voltagem nas linhas, usam-se códigos de deteção ou correção de erros. Bits adicionais são acrescentados a cada palavra de memória visando a deteção ou correção de erros. As seguintes técnincas podem também ser usadas para transmissão de dados.

Código de deteção Um bit paridade é acrescentado a cada palavra (Figura 5).

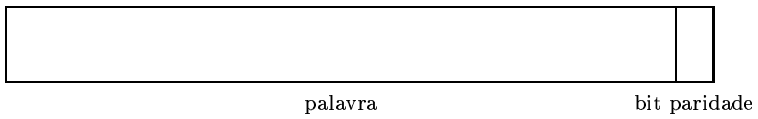


Figura 5: Um bit paridade no fim de cada palavra

O bit paridade é escolhido de tal modo que o número de 1's do código resultante (palavra+paridade) é par (ou ímpar).

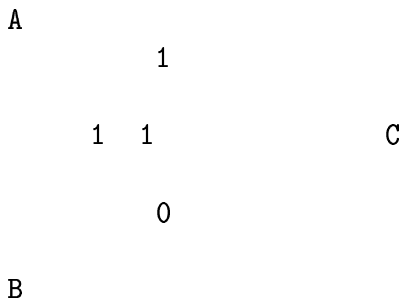
Esse código detecta erros de 1 só bit no código.

Seja o código	00110 (último bit é paridade)
se lido como	01110 (erro de 1 bit: erro detectado)
se lido como	00111 (erro de 1 bit: erro detectado)
se lido como	01111 (erro de 2 bits: nao detectado)

Código de correção - código de Hamming Num código de deteção, como paridade, o erro de 1 bit é detectado. Mas não se sabe onde está o erro, ou seja, qual o bit errado. O dado precisa ser lido (ou retransmitido, no caso de comunicação de dados) de novo.

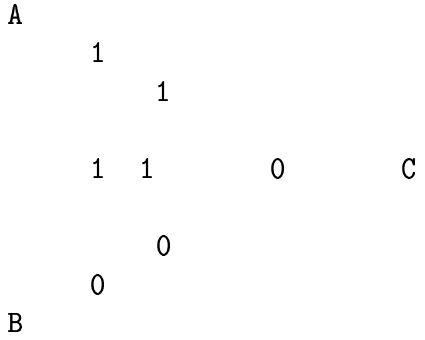
Num código de correção, sabe-se onde está o erro, de modo que é possível corrigi-lo.

Vejamos o código de Hamming para dados de 4 bits. Vamos acrescentar nesse caso mais 3 bits adicionais. Primeiro um exemplo.

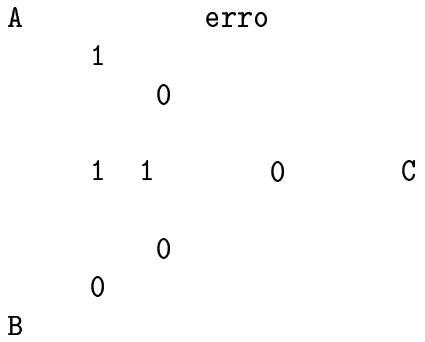


Seja o dado 1101. Para obter os 3 bits extras, vamos inicialmente encarar 1101 como os bits nas regiões de interseção AB, AC, BC, ABC, onde A, B, C são diagramas de Venn.

Agora vamos acrescentar um bit de paridade em cada uma das 3 regiões vazias acima para dar paridade par em A, B, e C.



Erro de 1 bit (qualquer um dos 7 bits) pode ser localizado.



Tal erro pode ser detectado de modo simples, como se segue:

- região A: paridade errada
- região B: paridade OK
- região C: paridade errada

- Conclusão: regiao AC errada. Logo o bit daquela região devia ser 1.

Agora vamos rever o método:

Sejam os n (igual a 4) bits do dado original

$$m_1 m_2 m_3 m_4 = 1101$$

Vamos acrescentar k (igual a 3) bits extras e produzimos o seguinte código de $n + k$ (igual a 7) bits

$$\begin{array}{ll} x_1 = m_1 \oplus m_2 \oplus m_4 & 1 \\ x_2 = m_1 \oplus m_3 \oplus m_4 & 0 \\ x_3 = m_1 & 1 \\ x_4 = m_2 \oplus m_3 \oplus m_4 & 0 \\ x_5 = m_2 & 1 \\ x_6 = m_3 & 0 \\ x_7 = m_4 & 1 \end{array}$$

Ao ler a memória desses 7 bits, suponhamos o seguinte resultado de leitura (com erro de 1 bit):

$$\begin{array}{lcl} y_1 & = & 1 \\ y_2 & = & 0 \\ y_3 & = & 1 \\ y_4 & = & 0 \\ y_5 & = & 0 \text{ (erro: devia ser 1)} \\ y_6 & = & 0 \\ y_7 & = & 1 \end{array}$$

Calculamos os valores $k_3 k_2 k_1$:

$$\begin{array}{ll} k_3 = y_4 \oplus y_5 \oplus y_6 \oplus y_7 & 1 \\ k_2 = y_2 \oplus y_3 \oplus y_6 \oplus y_7 & 0 \\ k_1 = y_1 \oplus y_3 \oplus y_5 \oplus y_7 & 1 \end{array}$$

Temos $k_3 k_2 k_1 = 101$.

Se $k_3 k_2 k_1 = 000$ então não há erro. Caso contrário, $k_3 k_2 k_1$ fornecerá o valor binário de j do bit errado y_j.

No nosso caso, $k_3 k_2 k_1 = 101$ indica que y_5 está errado. Logo, y_5 deve ser 1 ao invés de 0.

A tabela baixo mostra o número de bits adicionais para vários tamanhos de palavras:

Palavra n bits	Extras k bits	Total	"overhead"
8	4	12	50 %
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4

1.3 Memória secundária

Fita magnética Ver Figura 6.

Figura 6: Organização dos blocos de uma fita

Densidade típica de gravação: 1600 BPI (bytes por polegada)
Outras densidades: 800 e 6250 BPI.

Disco magnético "seek time" = tempo de posicionamento da cabeça de leitura/gravação em cima da trilha desejada. O melhor caso (posicionar na trilha vizinha) é da ordem de 3 ms. O pior caso (ir da trilha mais interna a mais externa ou vice versa) é da ordem de 20 a 100 ms.

"latência rotacional" = uma vez posicionada a cabeça na trilha certa, o tempo de espera até que o dado desejado passe por baixo da cabeça. Supondo 3600 rotações por minuto, a máxima latência rotacional é 16,67 ms.

Disco rígido Varia de dezenas a centenas de Mbytes. O mais comum é o disco selado do tipo Winchester. Também existem disco do tipo "disk card". Existem ainda unidades para discos rígidos removíveis (Bernoulli box, Mega Drive, etc.)

Disco ótico Além da grande capacidade de armazenamento, o disco ótico tem a vantagem de ter uma vida útil da ordem de dezenas de anos.

- CD ROM (compact disc - read only memory): por causa de erros de gravaçaõ que são frequentes, usa-se um código de correção (Reed-Solomon, parecido com o de Hamming). A capacidade útil típica é de 500 Mbytes por disco.

333000 setores de 2 Kbytes cada, organizados em uma única trilha espiral. Um complexo controle de servo-mecanismo permite a leitura à velocidade constante.

Leitura de 75 setores/seg, dando 150 Kbytes/seg. O tempo médio de posicionamento é de cerca de 500ms (alto).

- CD WORM (Write once read many): pode ser gravado apenas uma vez pelo usuário.
- CD regrável: podem ser tanto lidos como gravados pelo usuário. Há duas tecnologias: tecnologia magneto-ótico e tecnologia de mudança de fase.

1.4 Dispositivos de E/S

Computadores de grande porte utilizam um ou mais processadores de E/S, chamados canais de dados ("data channels"), conforme Figura 7.

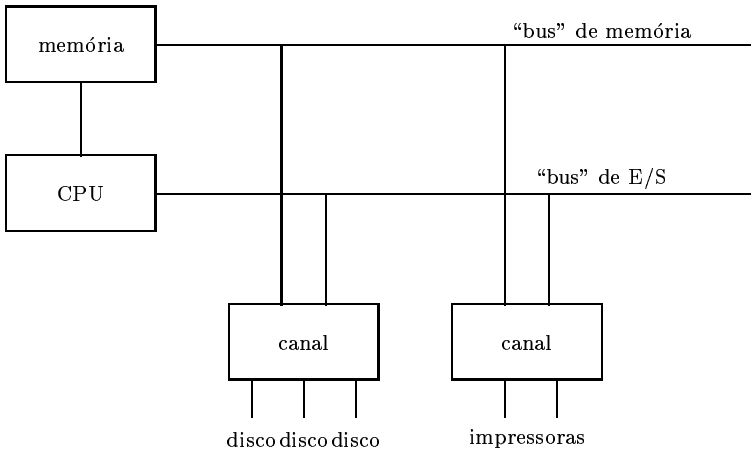


Figura 7: Processadores de E/S ou canais

Quando a CPU deseja realizar E/S, ela carrega um programa especial em um dos canais que o executa. A CPU pode então realizar outras tarefas. Quando o canal terminar sua tarefa, ele envia um sinal especial, chamado interrupção, à CPU.

Em computadores grandes, é comum o uso de vários barramentos ou "bus":

- "bus" de memória: através deste barramento canais lêem e escrevem dados da memória.
- "bus" de E/S: serve para transmitir comando da CPU aos canais e sinais de interrupção enviados pelos canais.

Já os computadores pessoais possuem estruturas de E/S mais simples. A estrutura básica é uma placa colocada no fundo chamada "motherboard", contendo a CPU, alguma memória, e alguns "chips" de suporte. O barramento fica na borda da placa onde podem ser acopladas placas de controle de E/S, placas de memória, etc.

A estrutura lógica é mostrada na Figura 8.

A função de um controlador de E/S é controlar o dispositivo de E/S, além de manipular acessos ao "bus". Quando um programa precisa de dados do disco, por exemplo, ele envia

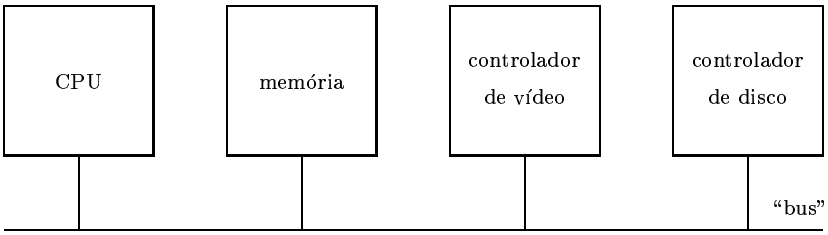


Figura 8: Componentes ligados ao barramento ou "bus"

um comando ao controlador de disco, que emite comandos como "seek" à unidade de disco. Também é tarefa do controlador receber os bits lidos do disco, montá-los em palavras, gravando-as na memória. Quando o controlador de E/S lê ou escreve blocos de memória sem intervençaõ da CPU, dizemos que ele realiza DMA ou seja, acesso direto a memória ("Direct Memory Access").

Em computadores menores, como nos pessoais, somente dispomos de um "bus". O mesmo "bus" é usado pelos controladores de E/S e pela CPU para buscar intruções/dados da memória.

Para disciplinar o uso do "bus", uma pastilha "bus arbiter" (árbitro do barramento) decide quem tem posse do "bus" (para utilizar os ciclos do "bus"). Em geral dispositivos de E/S têm prioridade sobre CPU no uso dos ciclos do "bus". Quando não há E/S, a CPU tem todos os ciclos de "bus" para ela. Um dispositivo de E/S, quando precisar, pode requerer e receber o uso do "bus". Esse processo é conhecido pelo nome de "cycle stealing" (roubo do ciclo).

1.5 Notação PMS

A notação PMS é usada para descrever um computador mostrando a interconexão dos seus componentes como processador, memória, chaves, controladores e unidades periféricas. O nome vem dos componentes "Processor-Memory-Switch". Utilizando um conjunto de componentes primitivos, ela ilustra como os componentes são interlidados para formar o sistema de computação. E importante conhecer a notação PMS que é comum na literatura para descrição de computadores.

São 6 os componentes primitivos:

- M ("Memory"): um componente do tipo M é capaz de armazenar informações para uso subsequente por outros componentes. Ele é passivo, no sentido de não alterar a informação recebida para armazenar.
- L ("Link"): um componente do tipo L transmite informações entre outros componentes. Como M, um componente L não é capaz de alterar informações.
- S ("Switch"): um componente do tipo S constrói links entre outros componentes. Um componente S tem um conjunto associado de L's que ele habilita ou inibe para fazer as conexões desejadas.

- D ("Data"): um componente do tipo D realiza operações sobre dados, criando e alterando informações. D é o único componente que modifica informação. Uma unidade aritmética (ALU) é do tipo D.
- K ("Kontrol"!): um componente do tipo K controla e ativa a operação de outros componentes. Todos os componentes não do tipo K são intrinsecamente passivos e requerem um K para serem ativados.
- T ("Transducer"): um componente do tipo T acopla o computador ao mundo externo, traduzindo toques de teclas em sinais digitais, etc.

Um computador convencional tem a estrutura básica da Figura 9, usando a notação PMS.

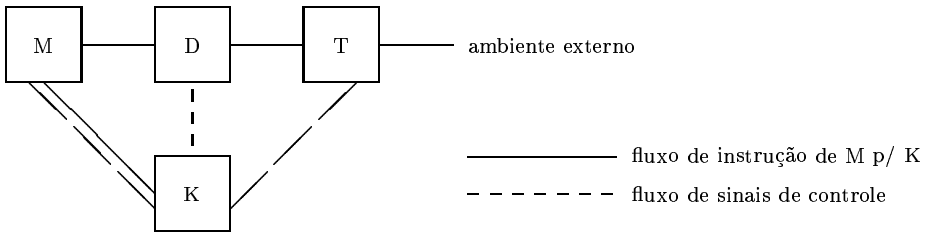


Figura 9: Computador convencional usando PMS

- A linha sólida indica fluxo de dados.
- A linha tracejada indica tranferência de informações de controle.

O par da Figura 10 é comumente conhecido pelo nome de processador central. Em geral é útil definir este par como um outro componente PMS (não primitivo) chamado P.

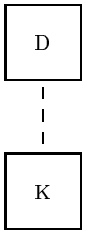


Figura 10: Processador central

Componente P: um processador capaz de interpretar uma sequência de instruções e executar as ações correspondentes.

O nosso sistema de computação pode agora ser reduzido, omitindo ainda as linhas de controle, para o que está mostrado na Figura 11.

Podemos ainda definir um outro componente não primitivo C (computador), que engloba o acima. Pode parecer não muito interessante, mas esse componente pode ser usado para descrever sistemas multicomputadores.

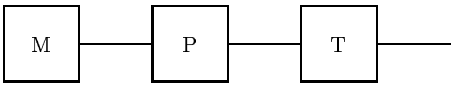


Figura 11: Computador usando componente P

Uma maneira usual de descrever os detalhes ou atributos de um componente genérico U é

$U(a_1 : v_1; a_2 : v_2; ....)$

onde a_i é o atributo e v_i o seu valor correspondente.

Por exemplo,

P (função: controle; nome: UNIVAC; tempo adição: 2 micro-seg)

M (t.cycle: 100ns; capacidade: 16 Kpalavras; palavra: 36 bits)

Algumas abreviaturas comuns usadas na literatura são derivadas como se segue.

Processador central:

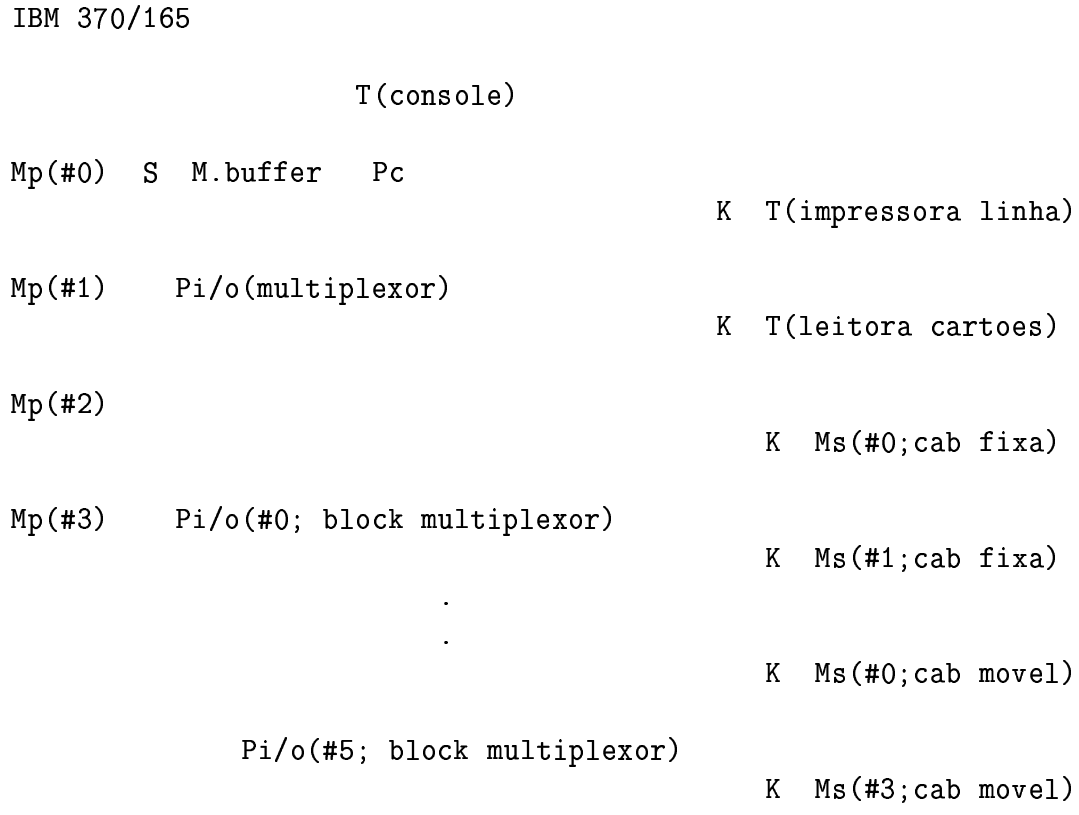
P (função: central) P (central) P.central P.c Pc

Memória primária:

M (função: primária) M (primária) M.primária M.p M_p

A título de ilustração da notação PMS, vamos dar alguns exemplos de computadores usando essa notação.

```
DEC PDP-11/40

 S (Unibus; data path: 16 bits; address path: 18 bits)

Pc   Mp(#0)  Mp(#1)   Mp(#7)                    K   T(terminal)

T.console                                      K  Ms(#0;DECtape)

                                                  Ms(#1;DECtape)

                                               K  Ms(#0;disco)

                                                  Ms(#1;disco)

                                                         .
                                                         .
                                                         .

                                                  Ms(#7;disco)

Pc(16 bits/palavra; 8 registradores; ...)

Mp(850 ns/palavra; capacidade: 8 Kpalavras; tecnologia: nucleo magnetico)

Ms(DECtape; 200 microseg/palavra; 147.968 palavras/rolo)

Ms(disco cabeca fixa; 33 ms/revolucao; capacidade: 262.244 palavras)
```



Pc(data path: 64 bits; ponto flutuante 32, 64 e 128 bits;
refrigeracao: agua)

Mp(256 Kbytes; t.cycle: 2 micro seg; 8 bytes/palavra)

M.buffer(16 Kbytes; t.cycle: 80 ns; 8 bytes/palavra)

Pi/o(block multiplexor; transferencia: 3 Mbytes/s)

Ms(disco cabeca fixa; capacidade: 5 Mbytes; rotacao: 10 ms;
transferencia: 1.5 Mbytes/s)

Ms(disco cabeca movel; capacidade: 2 disk packs de 100 Mbytes/pack;
transferencia: 806 Kbytes/s)

2 Nível de máquina convencional

2.1 Espaço de endereçamento

Um endereço absoluto de m bits pode endereçar 2^m posições de memória. Há várias maneiras de obter tal endereço absoluto.

Contido na própria instrução A instrução contém espaço suficiente para incluir todos os m bits necessários para especificar o endereço.

Exemplo - IBM 7094 (Figura 12).

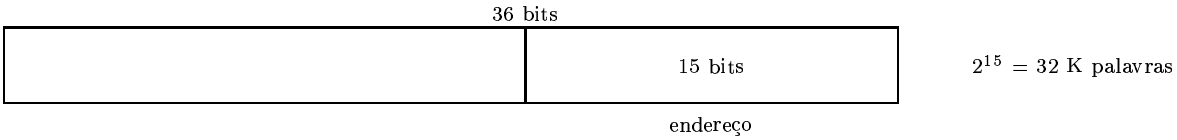


Figura 12: Endereço na própria instrução - IBM 7094

Outro exemplo - DEC PDP-11 (Figura 13).

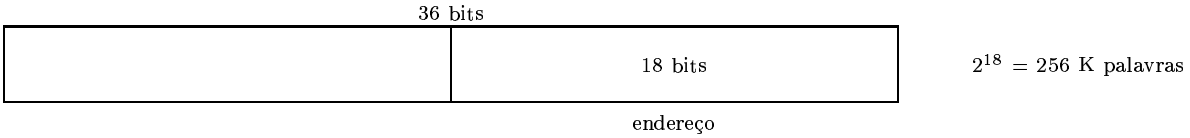


Figura 13: Endereço na própria instrução - PDP 11

Esse esquema simples apresenta alguns problemas:

- requer instruções de muitos bits
- espaço de endereçamento limitado

Uso de registradores base O endereço é obtido somando-se o conteúdo de um registrador base e um deslocamento ("offset").

Exemplo - IBM 360 e 370:
Um endereço de 24 bits é obtido com um registrador base e um deslocamento ou "offset" (Figuras 14 e 15).

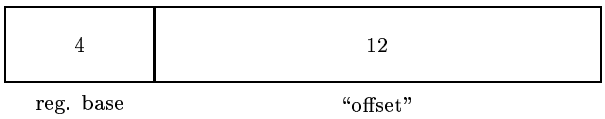


Figura 14: Registrador base e desclocamento

Uso de registradores de segmento Registradores de segmento apontam para determinados segmentos (programa, dados, pilha, etc.) e necessita-se apenas um deslocamento ("offset") em relação ao início do segmento.

Exemplo - Intel 8088 e 8086 (Ver Figura 16):
Apresenta 4 registradores de segmento chamados

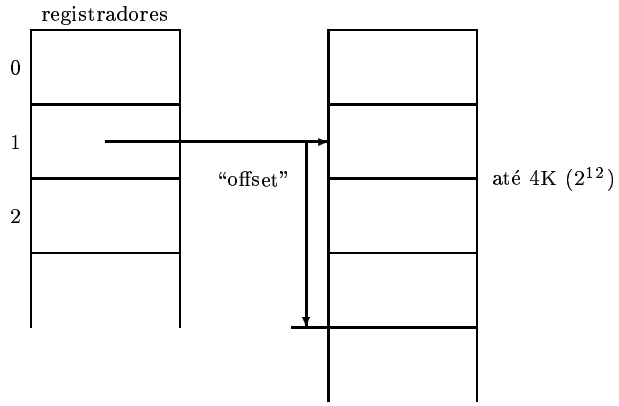


Figura 15: Endereçamento usando registrador base

- CS code segment (programa)
- DS data segment (dados)
- SS stack segment (stack ou pilha)
- ES extra segment (outros dados)

O Intel 8088 ou 8086 usa endereço absoluto de 20 bits, dando 2^{20} ou 1Mbytes endereçáveis.

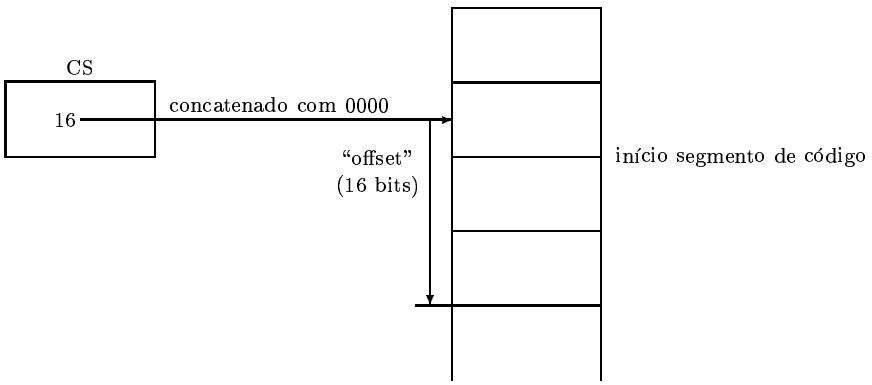


Figura 16: Endereçamento com registrador de segmento

Num dado instante, com os dados valores dos 4 registradores de segmento, pode-se endereçar um total de 256 Kbytes de memória (4 segmento de 64K cada). Mudando-se os valores dos registradores de segmento, outras posições (até 1M) podem ser endereçadas.

Uso de registradores de banco Nesse esquema, o endereço é o conteúdo do registrador de banco *concatenado* com o campo de offset.

Exemplo - O multiprocessador C.mmp (constituído de 16 processadores PDP-11) utiliza o esquema de endereçamento da Figura 17.

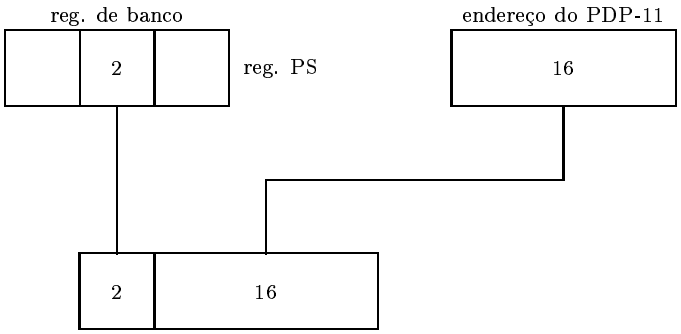


Figura 17: Endereçamento com registrador de banco

Outro exemplo - PDP-8: usam-se os primeiros 5 bits do PC (Program counter) que são concatenados a outros 7 bits para formar um endereço de 12 bits.

Uso de modos de endereçamento Esse esquema foi introduzido pelo PDP-11, cuja instrução de apenas 16 bits pode gerar 2 endereços de operandos, como mostra a Figura 18.

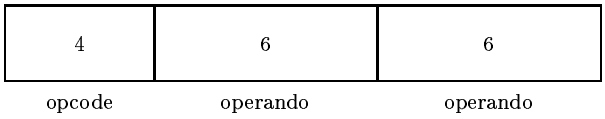


Figura 18: Formato de uma instrução

Cada operando é constituído de 3 campos, conforme a Figura 19.

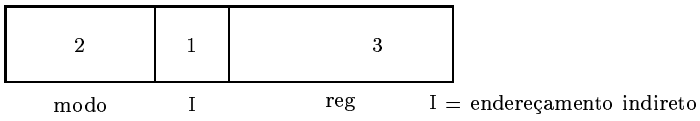


Figura 19: Os campos de um operando

- modo 00 - registrador: reg contém o operando
- modo 01 - auto-incr: reg é incrementado após uso como endereço do operando
- modo 10 - auto-decr: reg é decrementado e usado como endereço do operando

- modo 11 - indexado: conteúdo da palavra seguinte à instrução é somado com conteúdo do registrador reg para dar o endereço do operando

Intel e Motorola usam esquemas analógos, porém mais complexos.

Uso de memória virtual Esse assunto pertence a área de Sistemas Operacionais onde deverá ser tratado com mais detalhes.

2.2 Formato de instruções

Os elementos essenciais de uma instrução são

- opcode: código de operação
- endereços de operandos

Pode haver vários tipos de instruções, conforme ilustrados na Figura 20.

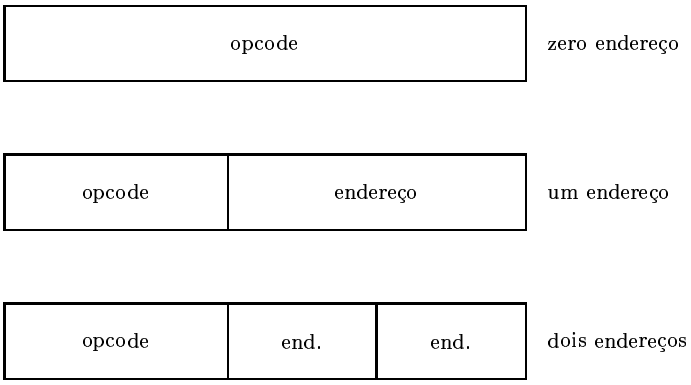


Figura 20: Tipos de instruções

Alguns critérios para a escolha de formato de instruções:

1. Instruções curtas são preferíveis a instruções longas.
2. Razão de transferência de instruções da memória para o processador
   - velocidade de transferência da memória: t bits/s
   - instrução de r bits
   - então t/r instruções por segundo podem ser transferidas (e executadas, se a CPU for suficientemente rápida)
3. Tamanho de instrução deve ser tal que ou ela cabe em um número inteiro de palavras ou um número inteiro de instruções cabem numa palavra.

Figura 21: Exemplo de uma instrução

"Opcodes que expandem" Consideremos um exemplo de uma máquina cujas instruções são de 16 bits e endereços de 4 bits, como mostra a Figura 21.

Cada endereço de 4 bits especificam, por exemplo, um dos 16 registradores da máquina. Com o formato acima podemos ter 16 instruções de 3 endereços.

Suponhamos, porém, que precisamos de

- 15 instruções de 3 endereços
- 14 instruções de 2 endereços
- 31 instruções de 1 endereço
- 16 instruções de 0 endereço

Então podemos ter as 15 instruções de 3 endereços assim:

```
opcode  0000  xxxx  yyyy  zzzz
        0001  xxxx  yyyy  zzzz
              .
              .
        1110  xxxx  yyyy  zzzz
```

Quando os primeiros 4 bits valem 1111, usamos os seguintes 4 bits para especificar 14 instruções de 2 endereços:

```
opcode  1111  0000  yyyy  zzzz
        1111  0001  yyyy  zzzz
        1111  0010  yyyy  zzzz
              .
              .
        1111  1101  yyyy  zzzz
```

Quando os primeiros 8 bits valem 1111 1110 ou 1111 1111, usamos os próximos 4 bits para especificar 31 instruções de 1 endereço:

```
opcode  1111  1110  0000  zzzz
        1111  1110  0001  zzzz
              .
              .
        1111  1110  1111  zzzz
        1111  1111  0000  zzzz
        1111  1111  0001  zzzz
              .
              .
        1111  1111  1110  zzzz
```

Finalmente quando todos os primeiros 12 bits valem 1, usamos os últimos 4 bits para especificar 16 instruções de 0 endereço:

```
 opcode  1111  1111  1111  0000
         1111  1111  1111  0001
               .
               .
         1111  1111  1111  1111
```

Na prática, opcode não são expandidos como no exemplo, da maneira tão regular e limpa, como veremos abaixo.

Formato de instrução no Motorola 68030 Teve uma influência grande do formato de instruções do PDP-11, porém muito mais complexo, devido a maior número de instruções. Além disso, o Motorola 68030 deve ainda considerar 3 tipos de operandos:

- byte 8 bits
- word 16 bits
- long 32 bits

A Figura 22 mostra as primeiras palavras de 18 formatos diferentes:

```
 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00

| op  | size|    operand      |    operand      |  MOVE
|  opcode   |  reg   |  mod   |    operand      |  ADD, AND
|  opcode   |  reg   |  op    |    operand      |  CHK, DIVS
|  opcode   |  reg   |  mod   |   op   |  reg   |  MOVEP
|  opcode   |  reg   |op| size|   op   |  reg   |  ASL, ROL
|  opcode   |  reg   |    opcode       |  reg   |  ABCD
|  opcode   |  reg   |op|          data         |  MOVEQ
|  opcode   | count  |op| size|   op   |  reg   |  ASL, ROL
|  opcode   | data   |op| size|    operand      |  ADDQ
|  opcode   | condition |  op |    operand      |  Scc
|  opcode   | condition |     displacement      |  Bcc
|  opcode   | condition |   opcode      | reg   |  DBcc
|  opcode               | size|    operand      |  ADD1
|  opcode                  |sz|    operand      |  MOVEM
|  opcode                     |    operand      |  JMP
|  opcode                        |   vector     |  TRAP
|  opcode                           |  reg   |  EXT, SWAP
|  opcode                                    |  NOP
```

Figura 22: Formato de instrução no Motorola 68030

Formato de instrução no Intel 80386 O formato de instrução no 386 é também muito complicado, sendo a instrução mais curta de 1 byte e a mais comprida até 17 bytes.

Há pouca estrutura ou regularidade no formato, exceto talvez nos 2 últimos bits do opcode:

- O bit da direita do opcode indica se o endereço de memória, se usado, é fonte ou destino da operação.
- O penúltimo bit da direita do opcode indica se o operando é tipo byte ou word.

O formato geral está mostrado na Figura 23. A Figura 24 mostra os campos opcode e mode. SIB é presente apenas no Motorola 68030, para dar um byte adicional ao modo.

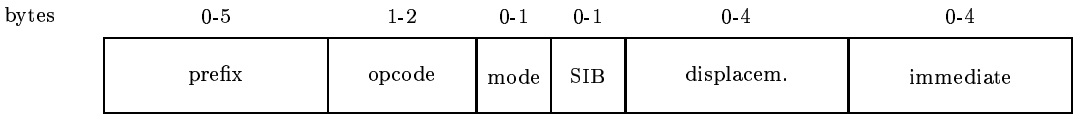


Figura 23: Formato de instrução no Intel 80386

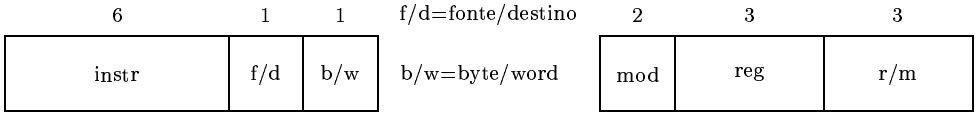


Figura 24: Campos opcode e mode

3 ISPS - Instruction Set Processor Specification

ISP ou ISPS (Bell e Newell) é uma linguagem de descrição formal de hardware do nível convencional de uma máquina. A notação permite especificar

- o estado do processador Pc
- o estado da memória Mp
- computação do endereço efetivo
- ciclo de interpretação de instruções
- computação de cada instrução

Linguagens de descrição formal de hardware é essencial para a formalização do processo de projeto de computadores digitais. ISPS tem sido usado como uma ferramenta de projeto, possibilitando a simulação e síntese de hardware, avaliação de arquitetura, análise e diagnóstico de falhas, geração de compiladres, etc (Figura 25).

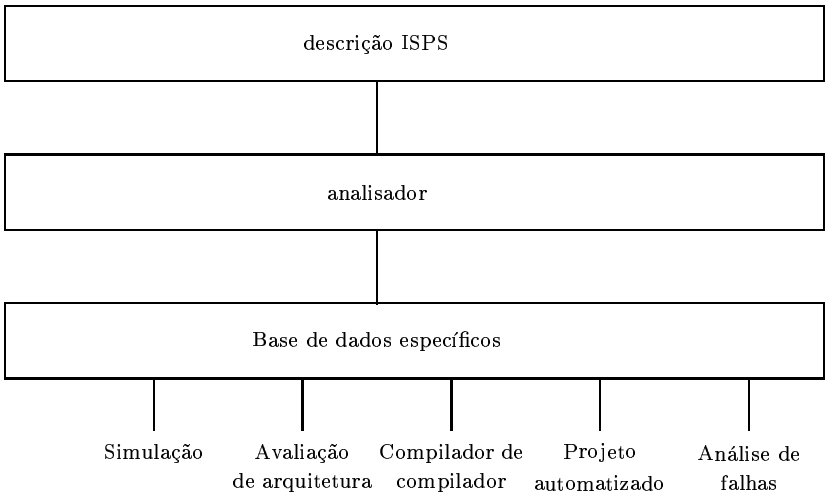


Figura 25: As aplicações de ISPS

3.1 Estado do processador - Pc

O estado do processador é o conjunto de registradores e flags (registradores de 1 bit) que guardam informações após a execução de uma instrução, informações essas disponíveis para a execução da próxima instrução. Como já vimos, eles incluem:

- PC ou "program counter"
- apontador a pilha
- códigos de condição (resultados de comparação)

- prioridade de processo
- valores intermediários de computações

3.2 Estado da memória - Mp

O estado da memória é o conjunto de palavras que constituem a memória principal.

Um exemplo do estado de processador (do IBM 370) é mostrado na Figura 26. O estado de memória é

$Mp[0{:}2^{24}] < 0:7 >$ ou seja 16 Mbytes.

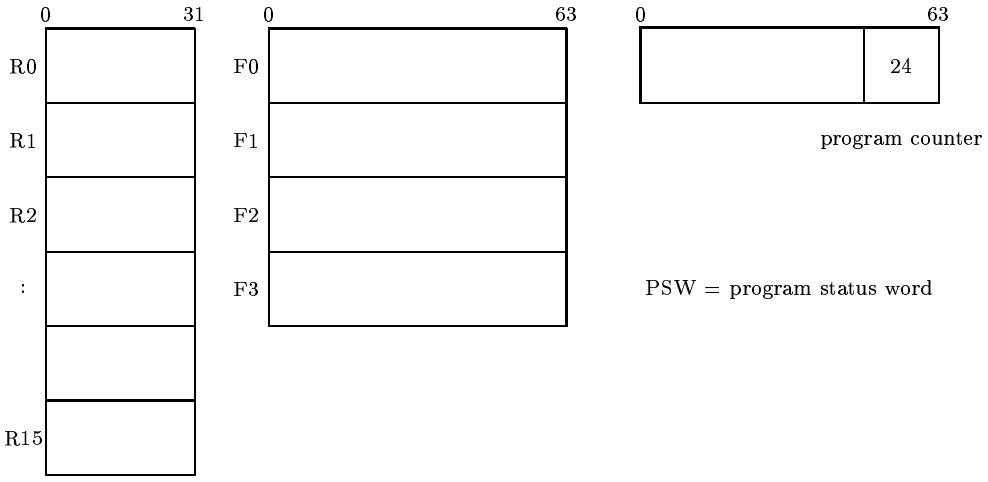


Figura 26: Estado de processador do IBM 370

O estado de processador do Motorola 68030 é mostrado na Figura 27.

3.3 ISPS - uma introdução

O que vem a seguir visa introduzir a notação ISPS, sem incluir todos os detalhes. Usaremos um exemplo de um minicomputador simples (PDP-8), para fins de ilustração.

3.4 Estado da memória

```
Mp\Primary.Memory[0:255]<0:11>
```

A memória tem um nome, Mp, e um "apelido" ou uma espécie de comentário, Primary.Memory. Para o ISPS, é considerado apenas o primeiro nome Mp.

Uma convenção geralmente seguida no ISPS é usar nomes que começcam com letra maiúscula para designar componentes concretos (físicos).

A memória primária é um array de palavras, de 0 a 255, especificado pelos colchetes. Cada palavra é de 12 bits, numerados de 0 a 11, da esquerda para a direita, especificada pelos colchetes angulares.

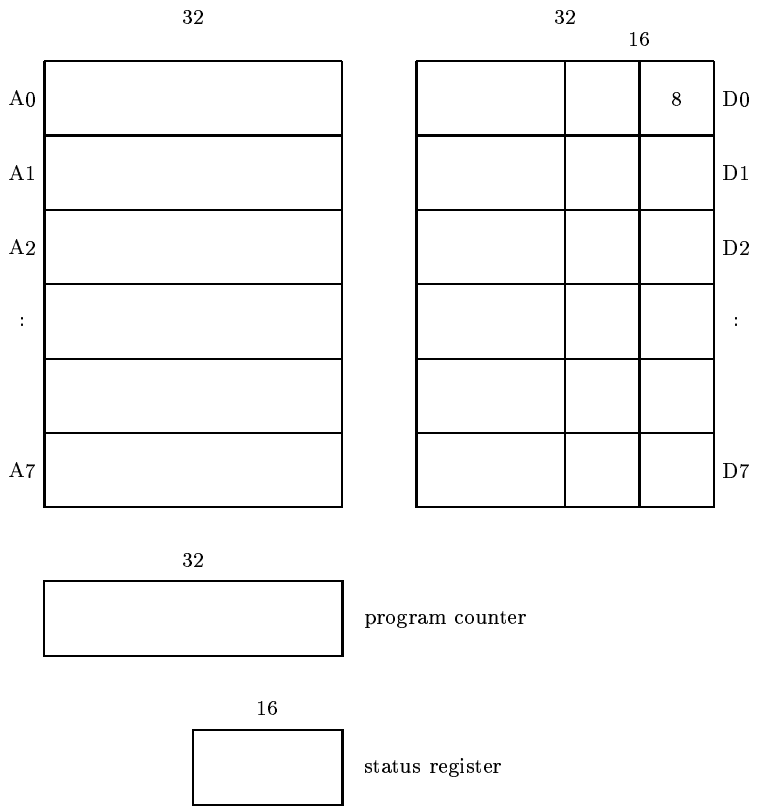


Figura 27: Estado de processador do Motorola 68030

3.5 Estado do processador

Aqui descrevemos os registradores que fazem parte do processador.

```
Acc\Accumulator<0:11>,
PC\Program.Counter<0:7>,
IR\Instruction.Register<0:11>,
    Op\Operation.Code<0:2>:=IR<0:2>,
    Ibit\Indirect.Bit<>:=IR<3>,
    Adr\Address<0:7>:=IR<4:11>,
Interrupt.Enable<>,
Interrupt.Request<>
```

O acumulador Acc é um registrador de 12 bits.

O apontador de instruçãoes PC contém 8 bits, para endereçar uma memória de 256 palavras.

O registrador de instrução IR contém a instrução corrente em execução, sendo usado no ciclo de interpretação de instrução, a ser visto. O IR por sua vez pode ser encarado como formado de Op, Ibit e Adr, significando respectivamente código de operação, bit indireto, e endereço. A notação Adr$< 0 : 7 >$:=IR$< 4 : 11 >$ significa que os bits 0 a 7 de Adr

correspondem aos bits 4 a 11 do IR. Op, Ibit e Adr não são registradores novos; apenas representam nomes alternativos dos vários campos do IR, dando o formato da instrução.

Temos ainda dois registradores de 1 bit cada: Interrupt.Enable vale 1 se o processador está permitindo ser interrompido, e 0 caso contrário. Interrupt.Request vale 1 se há algum pedido de interrupção, e 0 caso contrário.

Cálculo do endereço efetivo O endereço efetivo de um dado ou uma instrução é obtido por uma regra de interpretação expresso como um algoritmo. A memória da máquina simples é de 256 palavras; portanto endereços são de 8 bits.

```
Z\Effective.Address<0:7>
```

Z é o registrador que guarda o resultado do cálculo do endereço efetivo. O algoritmo para obtenção do endereço efetivo é especificado assim:

```
Z\Effective.Address<0:7>:=
    Begin
    If Ibit Eql 0 => Z <- Adr;
    If Ibit Eql 1 => Z <- Mp[Adr]<4:11>
    End
```

Z possui uma estrutura ($< 0 : 7 >$) e um corpo do procedimento. O corpo do procedimento, compreendido entre Begin e End, consiste de duas linhas, separadas por ponto-e-vírgula (;). O separador ponto-e-vírgula é usado para indicar comandos que podem ser executados em paralelo. No ISPS, o paralelismo ou concorrência é o padrão, quando comandos devem ser executados em sequência, tal deve ser especificado explicitamente, pela palavra Next, a ser visto mais tarde.

A linha

```
If Ibit Eql 0 => Z <- Adr
```

indica a execução condicional do comando seguido por $=>$. No caso, o comando é executado somente se Ibit for igual a 0. Vemos assim que $=>$ é uma espécie de "then".

O operador de atribuição é representado em ISPS por $<-$ (flechinha para esquerda). Quando o lado esquerda tem menos bits que o lado direito, então o valor do lado direito é truncado, ignorando os bits esquerdos a mais.

O corpo de Z pode também ser expresso de um outro modo, como seguinte:

Aqui "Decode Ibit $=>$" é usado para indicar um grupo de ações mutuamente exclusivas, dependendo do valor de Ibit. As ações correspondente são expressas dentro do Begin End interno, conforme o valor de Ibit. Vemos que Decode é parecido com o comando "case".

```
Begin
Decode Ibit =>
    Begin
    0:= Z<-Adr,
    1:= Z<-Mp[Adr]
    End
 End
```

Interpretação da instrução Temos alocado 3 bits para o código de operação. Podemos ter 8 instruções, conforme o valor de Op.

Código de operação 0:

```
If Op Eql 0 => Acc <- Acc And Mp[Z()];
```

Aqui Z() é usado para indicar uma chamada do algoritmo de cálculo de endereço efetivo Z.

Código de operação 1:

```
If Op Eql 1 => Acc <- Acc + Mp[Z()];
```

Código de operação 2:

```
If Op Eql 2 =>
    Begin
    Mp[Z] <- Mp[Z()] + 1 Next
    If Mp[Z] Eql 0 => PC <- PC+1
    End
```

A operação 2 corresponde a duas linhas, executadas em sequência, indicadas pela palavra Next. O comando que precede Next deve ser completado antes da execução do comando seguinte. Na linha

```
    Mp[Z] <- Mp[Z()] + 1 Next
```

notem que o cálculo de endereço é feito por Z(), apenas uma vez. Mp[Z], onde Z é usado **sem** parêntesis, significa apenas usar o endereço calculado, sem ativar o algoritmo de cálculo de novo. Isso é particularmente importante quando o algoritmo Z introduz efeitos colaterais, como no exemplo

```
Z\Effective.Address<0:7>:=
    Begin
    Decode Ibit =>
        Begin
        0:= Z <- Adr,
        1:= Z <- Mp[Adr] <- Mp[Adr]+1
        End
    End
```

Com esse algoritmo Z, teríamos um resultado diferente se tivéssemos escrito

```
Mp[Z()] <- Mp[Z()] + 1 Next
```

A instrução correspondente ao código 2 acrescenta 1 a uma palavra da memória e testa o resultado da adição, pulando a próxima instrução se a adição der resultado zero.

Código de operação 3:

```
If Op Eql 3 => Mp[Z()]@Acc <- Acc@#0000;
```

O operador @ é usado para indicar a concatenação de registradores ou cadeia de bits. #0000 indica uma cadeia de 4 dígitos octais. Alternativamente podemos escrever '0000000000000000 (12 dígitos binários) ou "000 (3 dígitos hexadecimais). Resumindo:

- # indica dígito octal
- ' indica dígito binário
- " indica dígito hexadecimal

O resultado dessa instrução é armazenar 24 bits (12 do acumulador mais 12 bits 0) no lado esquerdo - 12 bits da memória mais o acumulador. Poderíamos ter usado duas linhas:

```
Begin
Mp[Z()] <- Acc Next
Acc <- 0
End
```

Código de operação 4:

```
If Op Eql 4 => Stop()
```

Stop é uma instrução predeclarada do ISPS para congelar o estado da máquina permanentemente.

Código de operação 5:

```
If Op Eql 5 => PC <- Z();
```

Essa instrução altera a sequência normal de execução de instruções, alterando o valor do PC.

Códigos de operação 6 e 7:

Vamos por enquanto definir

```
If Op Eql 6 => Stop();
If Op Eql 7 => Stop()
```

3.6 Ciclo de interpretação de instrução

Estamos agora pronto para descrever o ciclo de interpretação de instrução. Vamos primeiro agrupar o que foi visto em termos de interpretação de cada instrução em um procedimento IExec, como abaixo:

```
IExec\Instruction.Execution:=
    Begin
    If Op Eql 0 => Acc <- Acc And Mp[Z()];
    If Op Eql 1 => Acc <- Acc + Mp[Z()];
    If Op Eql 2 =>
        Begin
        Mp[Z] <- Mp[Z()] + 1 Next
        If Mp[Z] Eql 0 => PC <- PC+1
        End;
    If Op Eql 3 => Mp[Z()]@Acc <- Acc@#0000;
    If Op Eql 4 => Stop();
    If Op Eql 5 => PC <- Z();
    If Op Eql 6 => Stop();
    If Op Eql 7 => Stop()
    End
```

O ciclo de interpretação de instruções pode agora ser descrito como

```
ICycle\Interpretation.Cycle:=
    Begin
    Repeat
        Begin
        IR <- Mp[PC] Next
        PC <- PC + 1 Next
        IExec() Next
        If Interrupt.Enable And Interrupt.Request =>
            Begin
            Mp[0] <- PC Next
            PC <- 1
            End
        End
    End
```

Em cada ciclo de interpretação de instrução, uma instrução é buscada da posição PC da memória e carregada em IR. O valor de PC é atualizada. A instrução é então executada pela chamada IExec(). Se os bits Interrupt.Enable e Interrupt.Request estão ambos ligados, então significa que há um pedido de interrupção e o processador pode atender a tal pedido. Nesse caso, o valor do PC é guardado em Mp[0] e o PC passa a conter 1. A instrução da palavra 1 da memória será então executada, para tratar a interrupção.

E interessante verificar o que deve conter a palavra 1 da memória, onde inicia a rotina de tratamento de interrupção. A rotina de interrupção deve salvar primeiro as informações

relevantes do processador (também chamado o contexto da máquina) e somente depois disso pode ir tratar a interrupção em questão. Note-se que antes de salvar o contexto, não podemos permitir nova interrupção. Assim, devemos colocar na palavra 1 uma instrução que inibe novas interrupções. Podemos usar o código de operação 6 para isso:

```
If Op Eql 6 => Interrupt.Enable <- 0;
```

Quando terminamos o tratamento da interrupção, devemos colocar em PC o valor guardado, para retornar ao ponto suspenso antes de acontecer a interrupção. Antes disso, porém, devemos permitir interrupções de novo, fazendo Interrupt.Enable igual a 1. Para isso, vamos usar o código de operação 7:

```
If Op Eql 7 =>
    Begin
    Interrupt.Enable <- 1 Next
    Restart ICycle
    End
```

O Restart significa ir para o começo do ciclo de interpretação de instrução. Isso é fundamental para que após a execução da instrução 7 (habilitando a interrupção), não se testem os bits Interrupt.Enable (que acabou de ficar 1) e Interrupt.Request (que pode ser 1, significando uma interrupção pendente). Por que? Porque antes de atender qualquer interrupção, queremos garantir a execução de mais uma instrução, que restaura o valor do PC guardado.

Podemos reescrever IExec usando Decode Op, incluindo as instruções 6 e 7, e acrescentano mnemônicos para as instruções.

```
IExec\Instruction.Execution:=
    Begin
    Decode Op =>
        Begin
        0\AND:= Acc <- Acc And Mp[Z()],
        1\TAD:= Acc <- Acc + Mp[Z()],  !Two's complement add
        2\ISZ:=              !Increment and skip if zero
            Begin
            Mp[Z] <- Mp[Z()] + 1 Next
            If Mp[Z] Eql 0 => PC <- PC+1
            End,
        3\DCA:= Mp[Z()]@Acc <- Acc@#0000,  !Deposit and clear Acc
        4\HLT:= Stop(),                    !Halt
        5\JMP:= PC <- Z(),                 !Jump
        6\IOF:= Interrupt.Enable <- 0,     !Interrupt OFF
        7\ION:=                            !Interrupt ON
            Begin
            Interrupt.Enable <- 1 Next
            Restart ICycle
```

```
            End
        End
    End
```

3.7 Divisão em seções

Em ISPS, as descrições são divididas em várias seções da forma:

** Seção 1 **
descrição
** Seção 2 **
descrição

O nosso exemplo do minicomputador simples fica então:

```
mini:=
    Begin

    ** Memory.State **

    Mp\Primary.Memory[0:255]<0:11>

    ** Processor.State **

    Acc\Accumulator<0:11>,
    PC\Program.Counter<0:7>,
    Interrupt.Enable<>,
    Interrupt.Request<>

    ** Instruction.Format **

    IR\Instruction.Register<0:11>,
        Op\Operation.Code<0:2>:=IR<0:2>,
        Ibit\Indirect.Bit<>:=IR<3>,
        Adr\Address<0:7>:=IR<4:11>

    ** Effective.Address.Calculation **

    Z\Effective.Address<0:7>:=
        Begin
        Decode Ibit =>
            Begin
            0:= Z<-Adr,
            1:= Z<-Mp[Adr]
            End
        End
```

```
** Instruction.Interpretation **
IExec\Instruction.Execution:=
  Begin
  Decode Op =>
      Begin
      0\AND:= Acc <- Acc And Mp[Z()],
      1\TAD:= Acc <- Acc + Mp[Z()],  !Two's complement add
      2\ISZ:=              !Increment and skip if zero
          Begin
          Mp[Z] <- Mp[Z()] + 1 Next
          If Mp[Z] Eql 0 => PC <- PC+1
          End,
      3\DCA:= Mp[Z()]@Acc <- Acc@#0000,  !Deposit and clear Acc
      4\HLT:= Stop(),                    !Halt
      5\JMP:= PC <- Z(),                 !Jump
      6\IOF:= Interrupt.Enable <- 0,     !Interrupt OFF
      7\ION:=                            !Interrupt ON
          Begin
          Interrupt.Enable <- 1 Next
          Restart ICycle
          End
      End
  End

ICycle\Interpretation.Cycle:=
    Begin
    Repeat
        Begin
        IR <- Mp[PC] Next
        PC <- PC + 1 Next
        IExec() Next
        If Interrupt.Enable And Interrupt.Request =>
            Begin
            Mp[0] <- PC Next
            PC <- 1
            End
        End
    End

End
```

4 Microprocessador numa pastilha

A interface de um microprocessador numa pastilha com o resto do sistema é bem definida, através de pinos de entrada e saída da pastilha. Em geral o número de pinos de uma pastilha varia entre 40 a um pouco menos de 150. Existem três tipos de pinos:

- pinos para endereços
- pinos para dados
- pinos para controle

A pastilha do processador Intel 386 possui 132 pinos (14x14-8x8 = 132) dispostos como mostra a seguinte figura.

Figura 28: 132 pinos do processador Intel 386

Os pinos da pastilha do processador são ligados a pinos correspondentes de outras pastilhas (como memória e dispositivos de entrada/saída) através de linhas paralelas formando um barramento ou "bus". Uma pastilha de um microprocessador típico apresenta tipicamente os pinos da Figura 29.

Por exemplo, para ler um dado da memória, o processador coloca o endereço nos pinos do endereço, e valida (ou "faz valer") uma linha de controle para informar a memória a ler uma palavra de memória. A memória responde colocando a palavra lida nos pinos de dados. (Ver Figura 30.)

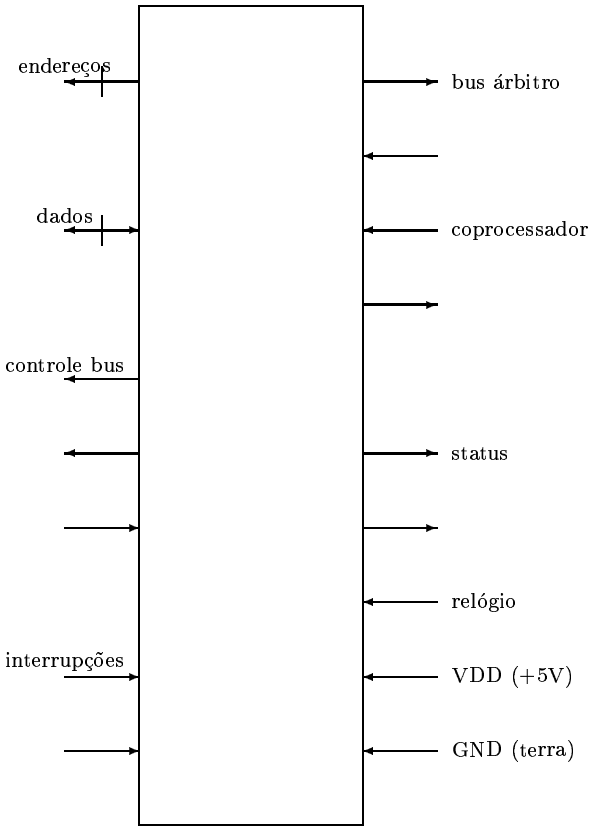


Figura 29: Os pinos de um microprocessador

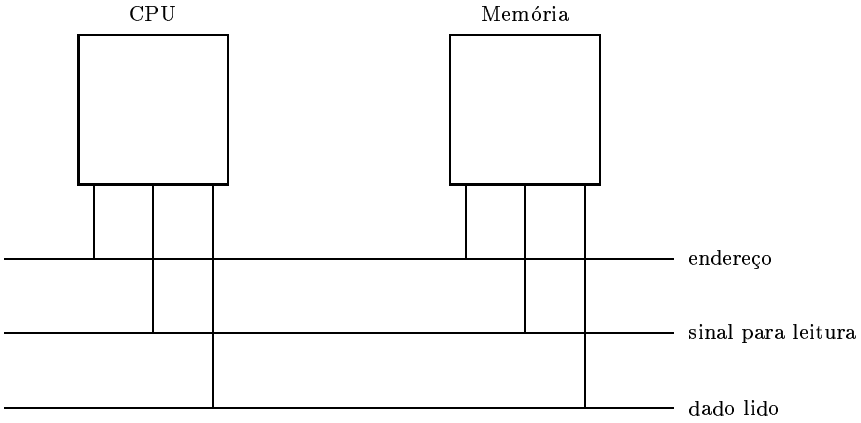


Figura 30: Leitura da memória

Terminologia Um sinal é dito validado (em inglês "asserted") se ele está preparado para causar uma certa ação.

Para alguns pinos, validar um sinal significa fazer o sinal igual a 1; para outros pinos, validar um sinal significa fazê-lo igual a 0. Neste último caso, colocamos uma barra em cima do sinal. Assim, LEIA (com barra em cima) significa a ação de leitura é ligada com o valor 0.

Os dispositivos conectados a um barramento podem ser:

- ativos: tomam iniciativa para começar uma transferência no barramento. Nesse caso o dispositivo é dito Mestre.
- passivos: ficam esperando a solicitação de algum outro dispositivo para fazer a transferência. O dispositivo passivo é conhecido como Escravo.

Exemplo

1. A CPU (mestre) manda o controlador de disco (escravo) ler um bloco de disco.
2. O controlador de disco (mestre), tendo lido um bloco, comanda a memória (escravo) para receber os dados lidos, através de DMA.

Pelos exemplos acima, vê-se que um mesmo dispositivo pode as vezes funcionar como mestre, outras vezes como escravo. A memória, entretanto, nunca pode funcionar como mestre.

Em geral, um sinal saído de uma pastilha precisa ser amplificado antes de ser colocado no barramento. A amplificação é feita por chamados "bus drivers" (não tem nada a ver com a CMTC). Por outro lado, o recebimento de sinais do barramento é feito por chamados "bus receivers". Um dispositivo capaz tanto de enviar como receber do barramento é conhecido de "bus transceiver" (Figura 31).

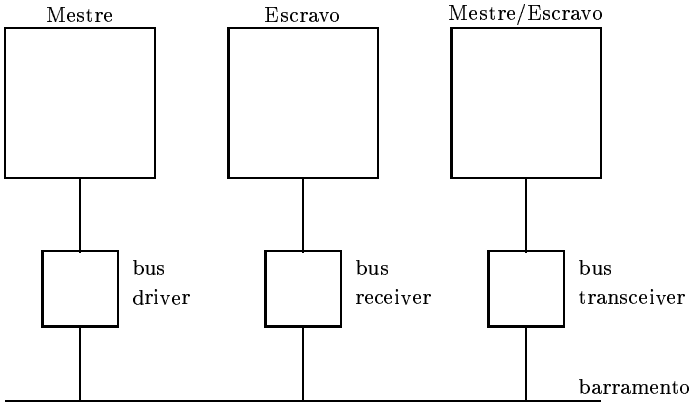


Figura 31: Transmissão de sinais no barramento

Em geral vários dispositivos estão conectados a um barramento; apenas alguns dos quais estão de fato utilizando o mesmo. Para não interferir no barramento quando um

dispositivo não o está usando, as interfaces dos dispositivos ao barramento são em geral do tipo "tri-state" (três estados: 0, 1 ou em aberto). Os dispositivos que não estão usando o barramento ficam no estado em aberto.

4.1 Barramento ou "bus"

Um barramento liga a CPU a memória e outros dispositivos. A largura de um barramento é tipicamente de 50 a 100 linhas paralelas. Além desse barramento central do sistema, pode também haver barramentos locais, como ilustra a Figura 32.

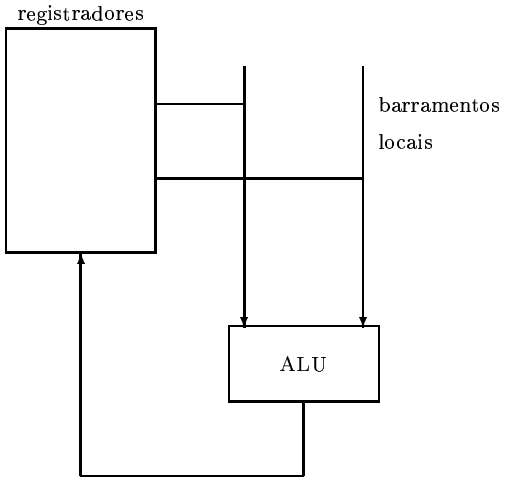


Figura 32: Barramento ou "bus"

As regras que disciplinam o uso do barramento formam o protocolo do barramento. Infelizmente, não há um único padrão para tais regras. Alguns protocolos mais conhecidos são:

- VME (Motorola 680x0)
- Multibus e EISA (Intel 386)
- Micro-channel (IBM PS/2)

Barramento síncrono Um barramento síncrono tem seu funcionamento controlado por um sinal de relógio gerado por um oscilador. As frequências variam, desde 5 MHz a 50 MHz. Uma frequência de 1 MHz corresponde a um ciclo de 1 micro-segundo, 4 MHz (250 ns), 20 MHz (20 ns) e assim por diante. Todas as atividades do barramento ocorrem em um número inteiro de ciclos do barramento.

Como exemplo, observe-se a Figura 33, que usa uma frequência de 4 MHz (ou ciclo de 250 ns). Um ciclo de leitura é consitituído de 3 ciclos do barramento.

Suponha que a CPU deseja ler uma palavra de memória. No primeiro ciclo, o endereço é colocado pela CPU nas linhas de endereço do barramento. Depois as linhas $MREQ$ (com barra em cima) e RD (com barra em cima) são ligadas (com 0), significando respectivamento pedido de acesso de memória e leitura. Nada acontece no segundo ciclo, para dar tempo para a memória decodificar o endereço e colocar a palavra lida, no terceiro ciclo, no

Figura 33: Ciclos de leitura

barramento de dados. Na descida do sinal de relógio do terceiro ciclo, a CPU lê os dados do barramento de dados, guardando-os em um registrador. Em seguinda, a CPU coloca 1 nos sinais $MREQ$ e RD, negando-os.

Figura 34: Alguns parâmetros

Figura 35: Arbitro do barramento

"Bus arbiter" A concessão do barramento, quando vários dispositivos o solicitam, é determinada pelo "árbitro". Os dispositivos são ligados como na figura e a concessão ou autorização para uso do barramento é enviada para o dispositivo 1. Este, se não pediu uso do barramento, passa o sinal de concessão adiante, para o dispositivo 2, e assim sucessivamente até chegar a um dispositivo que requereu o uso. Este dispositivo, ao receber o sinal de concessão, deixa de encaminhar o sinal adiante e fica dono do barramento. Este processo é conhecido pelo nome de "daisy chaining".

5 Microprogramação

A fronteira entre hardware e software não sempre é bem definida e muda constantemente. Máquinas mais antigas tinham as instruções aritméticas e booleanas diretamente implementadas em hardware. Máquinas mais modernas, muitas vezes, tem suas instruções do nível convencional realizadas passo a passo por um interpretador executando um microprograma. Exemplos de máquinas microprogramadas incluem as famílias Intel e Motorola. Uma classe de máquinas modernas, chamadas RISC, entretanto, não são microprogramadas e serão objeto de estudo mais adiante. Aqui vamos descrever o mecanismo de microprogramação.

O nível de microprogramação supõe a existência de um repertório primitivo de operações executadas diretamente por circuitos, que podem envolver registradores, barramentos, ALU, etc. Para a manipulação de tais componentes são usados sinais de controle, cada qual com a finalidade bem definida. Daremos a seguir alguns sinais de controle fundamentais, bem como alguns outros sinais importantes.

5.1 Sinais de controle para conexão ao barramento

A Figura 36 ilustra a conexão de um registrador de 8 bits entre um barramento de entrada e um barramento de saída.

O registrador é composto de 8 flip-flops do tipo D ligados a um barramento de saída via um buffer (amplificador) não-inversor. O registrador possui 2 sinais de controle: CK, que é usado para carregar o registrador com o valor do barramento de entrada; OE (output enable), usado para carregar o valor do registrador no barramento de saída. Normalmente, CK e OE estão no seu estado quiescente (isto é, que não causa ação). Eles são ligados quando a ação correspondente é desejada. (Figura 36.)

A Figura 37 representa um registrador de 16 bits ligado a um barramento C de entrada e 2 barramentos A e B, de saída. O sinal CK controla a carga do registrador e os sinais OE_1 e OE_2 controlam a carga do valor do registrador no barramento de saída A ou B, respectivamente.

Sinal de controle para multiplexador O sinal de controle de um multiplexador seleciona um dos valores da entrada para aparecer na saída do MUX.

O decodificador de n linhas de entrada produz 2^n linhas de saída, apenas uma das quais valendo 1, conforme o valor binário da entrada. (Figura 38.)

Sinais de controle da ALU e "shifter" Os sinais de controle, no exemplo, F_0 e F_1, determinam qual operação deve ser realizada pela ALU. No nosso exemplo, a ALU é capaz de realizar uma das 4 operações sobre suas entradas A e B (Figura 39):

1. $A + B$
2. A AND B
3. A (simplesmente copiando A da entrada)

Figura 36: Flips-flops

Figura 37: Um registrador e seus bits de controle

Figura 38: Um multiplexador e seus bits de controle

Figura 39: ALU e "shifter" e seus bits de controle

4. A' (isto é, NÃO A)

A ALU pode produzir sinais de controle de saída: o sinal N indica que o valor de saída da ALU é negativo; o sinal Z indica que o valor de saída da ALU é zero.

A unidade chamada "shifter" realiza o deslocamento da entrada de 0 ou 1 bit para esquerda ou direita. Os sinais de controle S_0 e S_1 indicam o tipo de deslocamento desejado.

Sinal de relógio O relógio é um dispositivo que emite sequências periódicas de pulsos que definem ciclos da máquina. Em cada ciclo de máquina, ocorre alguma atividade, como a execução de uma microinstrução. E comum dividir um ciclo em subciclos, que determinam a ordem de execução de partes de uma microinstrução numa ordem bem definida. Por exemplo, as entradas à ALU devem estar disponíveis e estáveis antes de guardar o resultado da ALU num registrador. Usaremos um relógio de 4 subciclos em cada ciclo, conforme indicada a Figura 40.

Figura 40: Sinal de relógio

Sinal de controle de memória Usaremos um registrador MAR (memory address register) que contém o endereço de memória a ser acessada, e um registrador MBR (memory buffer register) que contém o dado lido ou a ser gravado. (Figura 41.)

O sinal de controle do MAR controla a carga do MAR por um endereço vindo da CPU. O sinal de controle do MBR causa a carga do dado do MBR à CPU pelo barramento "data in" da figura. Os sinais RD e WR indicam leitura ou gravação.

5.2 Maneiras de construir um processador

Um processador pode ser construído de várias maneiras. Uma delas é usar pastilhas MSI, cada uma contendo um componente como ALU, registradores, shifter, conforme parte (a)

Figura 41: Registradores para acesso de memória

Figura 42: Construção de um processador

da Figura 42. Uma desvantagem é o grande número de pastilhas envolvidas, que podem ocupar várias placas.

Uma outra maneira é usar pastilhas chamadas "bit-slice". Cada pastilha "bit-slice" tem ALU e registradores de poucos bits (por exemplo 2 bits). Assim usando 4 pastilhas "bit-slice" de 2 bits podemos implementar um processador de 8 bits, conforme a parte (b) da Figura 42.

Uma terceira maneira é colocar todo o processador numa só pastilha, que demanda um elevado custo de projeto que em geral é complicado e demorado.

5.3 Um exemplo de microarquitetura

Na Figura 43 mostramos o "data path" (parte da CPU contendo a ALU e suas entradas e saídas) de uma microarquitetura.

A microarquitetura contém 16 registradores, denominados PC, AC, SP, etc., conforme mostra na figura. Esses registradores formam a chamada "memória rascunho", somente acessível no nível de microprogramação. Os registradores denominados 0, +1 e -1 são usados para conter as constantes indicadas. A saída de cada registrador pode ir a um ou ambos os barramentos A e B. Cada registrador da memória rascunho pode ser carregado através do barramento C.

Os barramentos A e B alimentam uma ALU de 16 bits que pode realizar 4 funções:

1. $A + B$
2. A AND B
3. A
4. NOT A

A função da ALU a ser realizada é especificada por 2 sinais de controle F_0 e F_1. A ALU gera 2 sinais de controle de saída: N (indicando se a saída da ALU é negativa) e Z (indicando se a saída da ALU é zero).

A ALU entra num "shifter", capaz de deslocar a entrada de 0 ou 1 bit para esquerda ou direita, conforme os sinais de controle S_0 e S_1. É possível realizar, por exemplo, um deslocamento para esquerda de 2 bits de um registrador R da memória rascunho, computando R+R na ALU (equivale a deslocar de 1 bit para esquerda) e depois deslocar mais 1 bit no shifter.

Os barramentos A e B alimentam a ALU através de dois registradores de entrada, denominados "A latch" e "B latch". A carga desses dois registradores é controlada pelos sinais de controle L_0 e L_1.

Para comunicação com a memória, temos os registradores MAR e MBR. O MAR pode carregado do B latch, em parelelo com uma operação da ALU. O sinal M_0 controla a carga do MAR. A saída do shifter pode entrar na memória rascunho, ou entrar no MBR, ou ambos. A carga de um valor do shifter para MBR é controlada por M_1. Os sinais M_2 e M_3 indicam leitura e gravação da memória (também usaremos os nomes RD

Figura 43: “Data path” da micro-arquitetura

e WR, respectivamente). No caso de leitura, o valor, lido para MBR, também pode ir ao lado esquerdo da ALU através do multiplexador AMUX. O sinal Ao controla se a entrada esquerda da ALU vem do A latch ou do MBR. A microarquitetura apresentada é análoga a de várias pastilhas "bit-slice" do mercado.

5.4 Microinstrução

Para controlar o "data path" da microarquitetura do exemplo precisamos de 60 sinais:

- 16 sinais para controlar a carga do barramento A pela memória rascunho
- 16 sinais para controlar a carga do barramento B pela memória rascunho
- 16 sinais para controlar a carga da memória rascunho através do barramento C
- 2 sinais para controlar "A latch" e "B latch" (L_0 e L_1)
- 2 sinais para controlar a função da ALU (F_0 e F_1)
- 2 sinais para controlar o shifter (S_0 e S_1)
- 4 sinais para controlar o MAR e MBR (M_0, M_1, M_2, M_3 - também denominados MAR, MBR, RD e WR)
- 1 sinal para controlar o AMUX (A0)
- 1 sinal ENC (enable C)

Este último, ENC, é útil para indicar se o resultado calculado deve ou não ser carregado de volta para a memória rascunho.

Dados os 60 sinais, podemos realizar um ciclo do "data path". Um ciclo consiste em colocar valores dos registradores da memória rascunho nos barramentos A e B, carga nos "A latch" e "B latch", realizaçaõ da operação pela ALU e depois shifter, e finalmente armazenamento do resultado de volta à memória rascunho ou MBR. Além disso, o MAR pode também ser carregado e um ciclo de acesso de memória iniciado.

Em vez de usar 16 bits de controle para controlar o barramento A, podemos codificar com 4 bits cada uma das combinações e usar um decodificador para produzir os 16 sinais necessários. O mesmo pode ser feito em relação ao barramento B.

No caso do barramento C, em princípio, podemos querer carregar múltiplos registradores da memória rascunho (isto é, carregar o resultado obtido em vários registradores). Limitando, entretanto, a carga de no máximo um registrador, podemos também codificar o registrador desejado com 4 bits e usar um decodificador, como no caso dos barramentos A e B. Economizamos assim $3 \times 12 = 36$ bits. Além disso, L_0 e L_1 serão fornecidos pelo sinal do relógio, como veremos, e dispensamos esses dois bits também. Reduzimos assim os 60 bits iniciais para 22 bits.

Uma microinstrução terá os 22 bits mais dois campos adicionais, denominados $COND$ e ADDR (a serem explicados adiante), dando um total de 32 bits, conforme a Figura 44.

Figura 44: Formato da micro-instrução

5.5 "Timing" da microinstrução em subciclos

Usaremos 4 subciclos, com os seguintes eventos chaves em cada um:

- Subciclo 1: carregar a próxima microinstrução a ser executada num registrador chamado MIR (micro instruction register)
- Subciclo 2: colocar valores dos registradores nos barramentos A e B, carregando os "A latch" e "B latch".
- Subciclo 3: com as entradas da ALU estáveis, o terceiro subciclo dá o tempo necessário para a ALU e shifter produzirem seu resultado estável, carregando-o no MAR se for o caso.
- Subciclo 4: armazenar o resultado na memória rascunho ou no MBR.

A Figura 45 dá o digrama detalhado da microarquitetura do nosso exemplo.

5.6 Os 4 subciclos para uma microinstrução

Um bloco importante no diagrama é a memória de controle ou "control store", uma memória rápida para armazenar microinstruções. No exemplo, temos um "control store" de capacidade para 256 instruções ou 256 x 32 = 8192 bits. Como outras memórias, o "control store" também precisa de seu MAR e MBR correspondentes, no caso denominados MPC (micro program counter) e MIR (micro instruction register), respectivamente. O "control store" continuamente tenta copiar a microinstrução endereçada por MPC no MIR. O MIR, entretanto, só recebe uma nova microinstrução no subciclo 1, como indicado pelas linhas tracejadas entrando no MIR. Nos outros 3 subciclos, o MIR não é afetado, não importando o valor de MPC.

No subciclo 2, o MIR está estável e os vários campos da microinstrução começam a controlar o "data path". Em particular, os campos A e B (de 4 bits cada) são decodificados para produzir dois conjuntos de 16 sinais necessários para controlar a carga dos

Figura 45: Os quatro ciclos da micro-arquitetura

registradores da memória rascunho nos barramentos A e B. Os "A latch" e "B latch" são carregados neste subciclo, produzindo as entradas para a ALU, e mantendo-se estáveis pelo resto do ciclo da microinstução (isto é, até o fim do quarto subciclo). Ainda no segundo subciclo, o valor do MPC sofre um incremento de 1, em preparo para a carga da próxima microinstrução.

No subciclo 3, a ALU e o "shifter" são dados tempo suficiente para produzirem resultados válidos. O campo AMUX da microinstrução determina a entrada no lado esquerdo da ALU; a do lado direito é sempre o "B latch". Ainda neste subciclo, o MAR é carregado do barramento, se for o caso.

Durante o subciclo 4, o valor do barramento C é armazenado de volta a memória rascunho, ou no MBR, conforme o caso. Um registrador especificado pelo campo C só é carregado com o valor presente no barramento C se $ENC = 1$ e o subciclo é 4.

5.7 "Sequenciamento das microinstruções"

As microinstrução são executadas uma a uma, em sequência, ou podem ser escolhidas pelo um desvio condicional. Para essa última finalidade, temos os campos COND e ADDR na microinstrução. No subciclo 4, quando a saída da ALU é estável, os sinais de saída N e Z entram no "micro seq. logic" juntamente com os 2 bits do campo COND. Esses 2 bits têm o seguinte significado:

- 0 = não há desvio; próxima microinstrução é MPC+1
- 1 = desvia para ADDR se $N = 1$
- 2 = desvia para ADDR se $Z = 1$
- 3 = desvia para ADDR incondicionalmente

Assim qualquer microinstrução potencialmente pode conter um desvio. Isso é devido a relativa frequência de haver desvios num microprograma.

Para tornar o nosso exemplo mais realístico, vamos supor que um ciclo de leitura de memória principal leva mais tempo que uma microinstrução. Se uma microinstução inicia uma leitura de memória principal, ao fazer $RD = 1$, vamos exigir que RD continua 1 na próxima microinstrução. A palavra lida só será disponível no MBR na microinstrução seguinte ainda.

5.8 Arquitetura MAC-1

Consideremos uma máquina simples MAC-1, com 4096 palavras de 16 bits, CPU com 3 registradores visíveis ao programador do nível convencional:

- PC (program counter)
- SP (stack pointer)
- AC (accumulator)

As instruções do nível convencional têm 12 bits cada e estão mostradas na Figura 46.

Figura 46: O conjunto de instruções de MAC

5.9 O Microprograma

A microarquitetura do exemplo é comandada por um microprograma para implementar as instruções do nível convencional de MAC-1. Chamamos tal microprograma de MIC-1. Cada microinstrução de MIC-1 tem 32 bits. A não ser que o microprogramador (quem escreve o microprograma) seja um masoquista, o seu trabalho pode ser facilitada pelo uso de um micro-montador ou micro-assembler. O micro-assembler converte cada instrução da linguagem do micro-assembler a uma microinstrução. Assim o microprogramado final, resultado da micro-montagem, é guardado no "control store" da microarquitetura.

Uma instrução em micro-assembler, por exemplo, para somar AC e A e guardar o resultado em AC, poderia ser definida como abaixo, o que, entretanto, não é ainda muito agradável para o microprogramador.

$$ENC{=}1\ C{=}1\ B{=}1\ A{=}10\ ALU{=}0$$

Vamos usar uma linguagem de micro-assembler mais parecida com linguagem de alto nível. Cada instrução, entretanto, ainda corresponde a uma microinstrução. Assim, a instrução exemplificada antes seria

ac:=ac+a

O registrador desejado pode aparecer no lado esquerdo do sinal de atribuição ":=". A função soma ($ALU = 0$) é indicada pelo sinal "+". As demais funções da ALU (AND, A, NOT A), correspondentes a ALU=1, 2 e 3, são indicadas por

... := band (... , ...)
... := a
... := inv(a)

Para especificar os bits SH que controlam o "shifter", usamos as funções lshift e rshift, como

tir:=lshift(tir+tir)

Essa instrução coloca tir no A e B, realiza a adição, desloca o resultado de 1 bit para esquerda, e finalmente guarda o resultado de volta a tir.

Desvio incondicional usa o comando goto; desvios condicionais podem testar n ou z, correspondentes aos sinais de controle N e Z gerados pela ALU. Assim, podemos ter

if n then goto 27

Atribuição e desvio podem ocupar a mesma instrução, como por exemplo,

ac:=ac+a; if z then goto 45

Temos uma pequena dificuldade, porém, se queremos testar um registrador sem desejando fazer nenhuma atribuição. Por exemplo, testar tir e desviar para 27 se negativo. Como vamos especificar qual registrador queremos testar. Para isso, introduz-se a pseudo-variável alu à qual pode-se atribuir um valor apenas para indicar o conteúdo da ALU. A instrução

alu:=tir; if n then goto 27

significa passar tir pela ALU, o que vai gerar os sinais de N e Z. O resultado da ALU não é armazenado de volta à memória rascunho. Assim ENC é sempre 0 se alu é usado.

A Figura 47 ilustra alguns exemplos de instruções de micro-assembler e suas microinstruções correspondentes.

Figura 47: Algumas instruções do micro-assembler

Podemos agora colocar todos os pedaços juntos. As Figuras 48 e 49 mostram o microprograma MIC-1. Ele surpreendentemente tem apenas 79 linhas. O microprograma ocupa portanto 79 palavras do "control store" da nossa microarquitetura.

Os nomes dos registradores da memória rascunho agora ficam óbvios. PC, AC, SP são usados para guardar os 3 registradores de MAC-1. IR é o registrador de instrução ("macroinstrução" do nível convencional) em execução. TIR é uma cópia de IR, usada para decodificar o código de operação da instrução. AMASK (address mask) é uma máscara de endereço, 007777 (octal), usada para separar os bits do endereço. SMASK (stack mask) é a máscara de pilha, 000377 (octal), usada para isolar um offset de 8 bits, usado para acessar pilhas. Os outros 6 registradores, A, B, ..., até F, não têm funções específicas e podem ser usados pelo microprogromador como ele bem entender.

O microprograma tem um "loop" principal que busca, decodifica, e executa instuções (macroinstruções, do nível convencional) da memória principal. O loop principal começa na linha 0, onde é buscada a macroinstrução de endereço apontado por PC. Enquanto aguarda a chegada da instrução lida, o microprograma incrementa PC de 1 e continua validando a linha RD. Na linha 2, a instrução lida chega e é guardada em IR; simultaneamente, os bit

Figura 48: O microprograma MIC-1

Figura 49: O microprograma MIC-1 - continuação

de ordem superior (bit 15) é testado. Se tal bit é 1, o processo de decodificação prossegue na linha 28; caso contrário, continua a linha 3.

Suponhamos que a macroinstrução é LODD, na linha 3 é testado o bit 14 e TIR é carregado com a instrução original IR deslocada de 2 bits para esquerda. Tal deslocamento de 2 bits é produzido pela soma de TIR+TIR e pelo "shifter". A linha 3 testa o bit 14 da instrução pois os sinais N e Z referem-se a saída da ALU (IR+IR ou IR deslocada de 1 bit) e não do "shifter".

Todas as instruções que começam com 00 (bits 15 e 14) chegam a linha 4 para ter seu bit 13 testado. As que começam com 000 continuam para linha 5 e as com 001 vão a linha 11. A linha 5 testa tir (bit 12 da instrução) mas não armazena nenhum resultado de volta à memória rascunho (uso de alu e ENC=0). Dependendo deste teste, temos LODD (linha 6) ou STOD (linha 9). Para LODD, os 12 bits inferiores de IR especificam o endereço da palavra a ser carregada no acumulador. Como o código de operação LODD é 0000, não há necessidade de separar os 12 bits de endereço, todo o registrador IR é enviado para MAR, para iniciar uma leitura. A palavra lida MBR é guardada no AC na linha 8 e volta-se a linha 0. Uma maneira mais geral para separar os 12 bits inferiores contendo o endereço numa instrução é realizar o AND lógico da instrução com a máscara de endereço (0000111111111111 ou 0777 octal), isto é,

band(ir, amask).

As outras instruções STOD, ADDD e SUBD são tratadas de maneira análoga. O único ponto que merece algum destaque é como realizar a subração. Com complemento de dois, temos o seguinte fato:

x - y = x + (-y) = x + (y' + 1) = x + 1 + not(y)

5.10 Microinstruções horizontais e verticais

A finalidade da microinstrução é especificar os sinais de controle necessários para controlar a microarquitetura. Todos os sinais podem estar presentes na microinstrução, da maneira que eles são usados, ou os mesmos podem estar codificados. Além disso, os sinais para um ciclo da microinstrução podem estar todos numa mesma microinstrução, ou contidos em várias microinstruções.

Microinstrução horizontal: todos os sinais necessários estão colocados na mesma microinstrução, sem nenhuma codificação. A microinstrução contém um grande número de campos não codificados. Como consequência, o "control store" contém um pequeno número de microinstruções compridas formadas com muitos campos, daí o nome horizontal.

Microinstrução vertical: a microinstrução contém poucos campos, altamente codificados. Mais de uma microinstrução podem ser necessárias para especificar todos os sinais necessários. O "control store" contém em geral um grande número de microinstruções curtas, daí o nome vertical.

5.11 Um exemplo de microinstrução vertical

Ao invés da microarquitetura MIC-1, podemos ter uma outra, MIC-2, cujo formato de microinstrução usa apenas 12 bits (Figura 50):

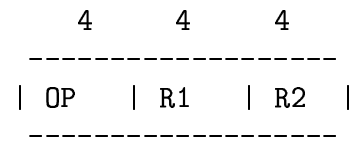


Figura 50: Microinstrução vertical

OP diz o que a microinstrução faz. R1 e R2 codificam 2 registradores. Em instruções de desvio, R1 e R2 são combinados para dar um endereço de 8 bits R.

Por exemplo,
ADD SP, AC
significa que o valor de AC é somado a SP.

Os 16 OP são:

0000	ADD	Addition	r1:=r1 + r2
0001	AND	Boolean And	r1:=r1 AND r2
0010	MOVE	Move register	r1:=r2
0011	COMPL	Complement	r1:=inv(r2)
0100	LSHIFT	Left shift	r1:=lshift(r2)
0101	RSHIFT	Right shift	r1:=rshift(r2)
0110	GETMBR	Store MBR in reg	r1:=mbr
0111	TEST	Test register	if r2<0 then n:=true;
			if r2=0 then z:=true
1000	BEGRD	Begin read	mar:=r1; rd
1001	BEGWR	Begin write	mar:=r1; mbr:=r2; wr
1010	CONRD	Continue read	rd
1011	CONWR	Continue write	wr
1100			Not used
1101	NJUMP	Jump if N=1	if n then goto r
1110	ZJUMP	Jump if Z=1	if z then goto r
1111	UJUMP	Unconditional jump	goto r

Dependendo do OP, sinais apropriados de controle devem ser produzidos. A tarefa de decodificar OP para produzir esses sinais é em geral feita por meio de PLAs ou ROMs.

5.12 A microarquitetura Intel 8088

As microarquiteturas da família Intel são semelhantes, sendo a mais simples a do 8088. A microarquitetura do 8088 consta de um "data path" mostrado na Figura 51.

Figura 51: “Data path” do Intel 8088

As microinstruções são do tipo vertical, com vários campos especificando funções gerais, ao invés de bits de controle individuais.

O "data path" possui duas partes: uma superior e uma inferior. A parte inferior do "data path" é semelhante ao "data path" da maioria dos computadores, com registradores (16 bits cada) e ALU. Os registradores TMPA, TMPC e TMPB são carregados antes de alimentar a ALU. O barramento marcado "cross" tem a capacidade de acessar campos de 8 bits dos registradores.

A parte superior do "data path" realiza o cálculo de endereços. Lembre-se de que no 8088, os 16 bits de um registrador de segmento são concatenados com 0000 para formar 20 bits e depois somados ao valor de um deslocamento (ou offset) de 16 bits. Tal cálculo é efetuado na parte superior, da seguinte maneira (Figura 52):

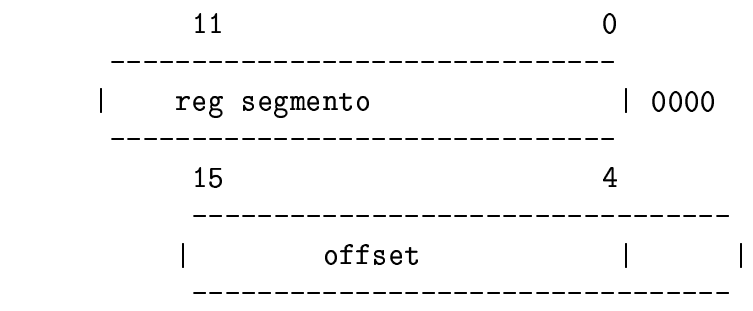


Figura 52: Cálculo do endereço

Os 4 bits inferiores do offset vão direto para o barramento de endereço. Os bits 4 a 15 são somados com os bits 0 a 11 do registrador de segmento no somador ("adder") da parte superior, dando os bits superiores do barramento de endereço.

A parte de controle da microarquitetura do 8088 é mostrada na Figura 53.

A parte de controle armazena o microprograma que dirige o "data path". Quando uma instrução do nível convencional é executada, o seu Opcode é carregado no IR da figura. A unidade "group decode" da figura extrai informações do IR e prepara vários sinais de controle para a ALU: M e N para o cálculo de endereços dos operandos, X para determinar a função da ALU.

As microinstruções são de 21 bits cada. O microprograma consta de 504 microinstruções. Uma PLA converte o Opcode contido no IR em um endereço do microcódigo que trata aquela instrução. A decodificação é feita portanto em hardware, ao contrário do nosso exemplo anterior, em que a decodificação foi feita bit a bit por software.

Figura 53: Parte de controle do 8088

6 Máquinas RISC

RISC significa "Reduced Instruction Set Computer". A família RISC contrasta-se com as chamadas máquinas CISC - "Complex Instruction Set Computer". Exemplos de máquinas CISC incluem computadores de grande porte como o IBM 360, e máquinas como DEC VAX, Intel 386 e Motorola 68030.

Proponentes da filosofia RISC pregam uma completa revira-volta sobre a maneira de pensar sobre arquiteturas de computadores. O argumento principal alegado é que computadores estão ficando demasiadamente complicados. Essa opinião gerou bastante controvérsia e discussão, que iremos examinar nesta seção.

6.1 Evolução da arquitetura de computadores

As primeiras máquinas eram extremamente simples, com relativamente poucas instruções e poucos modos de endereçamento.

Tudo começou a mudar com o conceito de microprogramação (devido a Wilkes) e a introdução do IBM 360, que é microprogramado, possibilitando um conjunto complexo de instruções do nível convencional, conhecidas como instruções de máquina. Dentro de poucos anos, máquinas como VAX tipicamente tem mais de 200 instruções e mais de uma dúzia de modos de endereçamento, todos implementados por um microprograma rodando num hardware simples.

Um outro fator que encorajou a evolução das máquinas CISC é a baixa velocidade de acesso da memória principal RAM, em relação a velocidade de um ROM dentro da CPU. Para ilustrar isso, pense em aplicações COBOL que necessitam de aritmética decimal. Como todas as máquinas internamente são binárias, a aritmética decimal teria que ser simulada. Há duas maneiras possíveis. A primeira é exigir que o programa COBOL chame rotinas de biblioteca armazenadas na memória principal. A segunda é colocar tais rotinas no microprograma (ROM dentro da CPU) e acrescentar instruções do tipo ADD DECIMAL ao repertório de instruções. Devido a baixa velocidade de acesso de memória RAM, a tentação é colocar maior complexidade no microprograma.

A situação começou a mudar com a introdução de memória RAM rápidas, que não são mais cerca de 10 vezes mais lentas que a ROM, como antigamente. A manutenção do microprograma também tem gerado grandes dores de cabeça: consertar um "bug" num microprograma gravado em ROM já entregue ao mercado significa visitar todas as instalações de clientes para trocar as ROMs defeituosas.

Um outro acontecimento marcante foi a descoberta de que realmente poucos tipos comandos são usados na maioria dos programas examinados em vários experimentos realizados. Knuth, Wortman, Tanenbaum e Patterson mediram programas escritos em várias linguagens e constataram os seguintes comandos mais usados:

Comando	Fortran	C	Pascal
atribuicao :=	51%	38%	45%
if	10	43	29
call	5	12	15
loop	9	3	5
goto	9	3	0
outros	16	1	6

Em média, atribuição ":=", "if" e "call" constituem 85% dos comandos usados. Mais interessante ainda é a constatação de que 80% das atribuições é do tipo

variável := um só termo (constante ou variável)

como mostra a figura seguinte.

Numero de termos	%
1	80
2	15
3	3
4	2
5 ou mais	0

A distribuição de variáveis locais escalares em procedimentos também é interessante: 22 % não tem nenhuma variável local e 80 % tem 4 ou menos variáveis locais, como mostra a figura seguinte.

Numero de variaveis locais	%
0	22
1	17
2	20
3	14
4	7
5 ou mais	20

Quanto ao número de parâmetros de procedimentos, 84 % usam 3 parâmetros ou menos, conforme a tabela abaixo:

Numero parametros	%
0	41
1	19
2	15
3	9
4	7
5 ou mais	9

A conclusão é simples: na prática os programas realmente escritos consistem de comandos de atribuição (com poucos termos), if's e chamadas de procedimentos com poucos parâmetros e variáveis locais. Essa constatação põe em dúvida a introdução de maior complexidade no microcódigo. Um microprograma complexo significa maior tempo para

decodificar e executar uma instrução, muitas das quais raramente são usadas. O número grande de modos de endereçamento significa que a análise de endereço tem que ser feita por um microprocedimento. Assim, cada instrução de dois operandos, por exemplo, teria que chamar tal microprocedimento duas vezes.

A consequência de tudo isso é a constatação de que o computador pode ser mais rápido se jogarmos fora todo o interpretador (microprograma) duma vez por todas.

Uma máquina RISC é, portanto, essencialmente um computador com um pequeno repertório de instruções parecidas com microinstruções verticais. Programas de usuários são compilados em sequencias dessas instruções que são armazenadas na memória principal RAM, buscadas e diretamente executadas por hardware, sem nunhuma interpretação.

A primeira máquina RISC foi o minicomputador 801, construído em 1975 por IBM. Em 1980, uma equipe liderada por dois professores de Berkeley, Patterson e Séquin, projetou a pastilha RISC I e depois RISC II. Em 1981, Hennessy, de Stanford, projetou a pastilha MIPS. A Figura 54 faz uma comparação dessas 3 máquinas RISC com 3 máquinas CISC.

Figura 54: Máquinas CISC com 3 primeiras RISC

Os três projetos RISC acima mencionados levaram a produtos comerciais: o 801 é o antecessor do IBM PC/RT (Risc Technology); o RISC I inspirou a arquitetura SPARC da Sun Microsystems; a pastilha MIPS levou a formação da empresa MIPS Computer Systems que produz pastilhas RISC da DEC e de alguns outros fabricantes.

6.2 Pricípios de projeto de máquinas RISC

O projeto de uma máquina RISC seguen alguns princípios que podem ser resumidos abaixo:

1. Analizar aplicações que a máquina pretende tratar para localizar operações chave.

2. Projetar um "data path" que é o mais eficiente possível para essas

3. operações chave.

4. Projetar instruções que realizam as operações chave usando o "data path".

5. Acrescentar mais instruções, desde que não tornem a máquina mais lenta.

A **regra de ouro** é a seguinte:

Sacrificar tudo para reduzir o ciclo do "data path".

A meta é poder executar uma instrução por ciclo do "data path".

6.3 Endereçamento apenas por registrador

Uma das consequências da regra de ouro é na redução drática dos modos de endereçamento. Ao contrário das máquinas CISC, que possuem muitos modos de endreçamento, as máquinas RISC apresentam apenas o endereçamento por registrador. Assim, não envolvendo acessos à memória, as instruções são executadas uma por ciclo.

A pergunta natural que se faz é como carregar e descarregar registradores. A solução é permitir apenas duas instruções, LOAD e STORE, que acessam a memória. Tais instruções em geral não podem ser completadas em um ciclo. Aumentar o ciclo por causa de LOAD e STORE contraria o item número 4 acima. A solução é o uso de *pipelining* como veremos a seguir.

6.4 Uso de pipelining

A regra de ouro vai ser relaxada ligeiramente: ao invés de exigir a execução de uma instrução por ciclo, será exigido o início da execução de uma instrução em cada ciclo.

Vamos supor que a execução de uma instrução envolve 2 ou 3 etapas, cada etapa levando um ciclo. As etapas são:

1. busca da instrução da memória

2. execução

3. no caso de LOAD e STORE, acesso a memória

As etapas são executadas por cicuitos separados de hardware, como em uma linha de montagem ("pipelining"). A Figura 55 mostra a idéia.

O hardware permite a execuçaõ simultânea das 3 etapas: busca, execução, e acesso a memória. Notem que as 3 etapas não se referem a uma mesma instrução. (O mesmo acontece com uma linha de montagem de automóveis: os vários estágios em executando simultânea tratam de vários carros.) Veja a figura do exemplo. No ciclo 1, a instrução 1 é buscada. No ciclo 2, a instrução 2 é buscada e a instrução 1 executada. No ciclo 3, a instrução 3 (marcada L para significar LOAD) é buscada e a instrução 2 executada. No ciclo 4, a instrução L é iniciada e, como envolve acesso à memória, deve levar mais um ciclo para ser completada. No ciclo 5, algo interessante acontece (marcado pelo círculo). A instrução 4 é executada, embora a instrução L não tenha sido completada ainda. Isso

Figura 55: Load retardado

é possível desde que a instrução 4 não utilize o registrador que está sendo carregado pela instrução L.

É trabalho do compilador descobrir alguma instrução (a instrução 4 no exemplo) para poder preender o "buraco" deixado por instruções LOAD e STORE.

6.5 Problema de instruções de desvio

A execução de instruções por pipelining tem o velho problema do desvio: precisa começar a busca de uma nova sequência de instruções. A solução que se adota, para ter algo a fazer enquanto se busca a nova sequência de instruções por cause de um desvio é a seguinte: colocar após a instrução de desvio uma instrução que logicamente deveria ser executada *antes* do desvio, e executar sempre a instrução seguinte a uma instrução de desvio.

6.6 Trabalho do Compilador

Temos assim LOAD/STORE e desvios *retardados*. E trabalho do compilador descobrir instruções que podem ser colocadas nos "buracos" deixados por tais instruções. De modo geral, para as máquinas RISC, uma parte da complexidade é deixada para o compilador.

6.7 Acelerar a chamada de procedimentos

Uma maneira de usar o espaço que antes era ocupado pelo microprograma e os circuitos correspondentes é colocar mais registradores. Não é incomum ter mais de 500 registradores numa CPU do tipo RISC.

Em particular o RISC I usa os registradores adicionais para acelerar a chamada de procedimentos, uma das operações mais usadas em programas. O conceito usado é o de *janelas superpostas de registradores*, explicado a seguir.

Cada programa vê um conjuntos de 32 registradores de 32 bits cada especialmente usados para chamada de procedimentos: 8 para variáveis globais, 8 para parâmetros de entrada, 8 para variáveis locais do procedimento e 8 para parâmetros de saída que são transmitidos como entrada a outros procedimentos. Os registradores com variáveis globais devem ser os mesmos para todos os procedimentos; os outros devem ser mudados para cada chamada.

A Figura 56 mostra em (a) a situação da primeira chamada de um procedimento. O apontador CWP ("current window pointer") aponta para o início dos registradores contendo os parâmetors de entrada. O procedimento prepara os parâmetros de saída como mostra a Figura 56 (a). Esse procedimento pode chamar outro procedimento, e os parâmetros de entrada do novo procedimento são exatamente os de saída do primeiro. Assim basta superpor os 8 registradores correspondentes dos dois procedimentos, como mostra a Figura 56 (b).

Figura 56: Janelas de registradores superpostas

Desse jeito, a chamada de procedimentos podem em geral ser executadas sem utilizar a pilha na memória. Isso somente seria necessário quando por exemplo o número de parâmetros exceder o limites dos 8 registradores, o que deve acontecer raramente.

6.8 Desempenho das máquinas RISC

A Figura 57 mostra o desempenho relativo de algumas máquinas conhecidas.

A Figura 58 mostra o desempenho expresso em drhystones por segundo (ou seja, o número de operações inteiras por segundo). Como aproximação grosseira, cada 2000 drhystones equivalem a 1 MIPS.

Figura 57: Comparação do RISC-I com várias máquinas

Figura 58: Comparação entres máquinas RISC e CISC

7 Arquiteturas não convencionais

Classificação de computadores de alto desempenho

- Pipeline ou computadores vetoriais
- SIMD "Single instruction multiple data"
- MIMD "Multiple instruction multiple data"
- Dataflow

Exemplos de computadores de alto desempenho