CISC (Complex Instruction Set Computer) - Microprogramação

MAC0344 - Arquitetura de Computadores
Prof. Siang Wun Song
Slides usados: https://www.ime.usp.br/∼song/mac412/oc-cisc.pdf

Baseado no livro de Tanenbaum - Structured Computer Organization

Arquitetura CISC e Microprogramação

- Hoje e nas próximas aulas veremos a arquitetura CISC e microprogramação.
- A arquitetura CISC usando microprogramação foi proposta em 1951 e utilizada até hoje.
- No final destas aulas, será passada a lista de exercícios no. 5, a última lista.

Como surgiu a microprogramação

Maurice Wilkes foi um cientista da computação britânico que construiu o computador EDSAC, sucessor do ENIAC. EDSAC foi um dos primeiros computadores a armazenar o programa na memória do computador. Em 1951 inventou a microprogramação que revolucionou o projeto de processadores. Wilkes foi professor emérito da Universidade de Cambridge. Em 1967 recebeu o Turing Award, criado em 1966. Wilkes foi portanto o segundo recipiente deste prêmio.

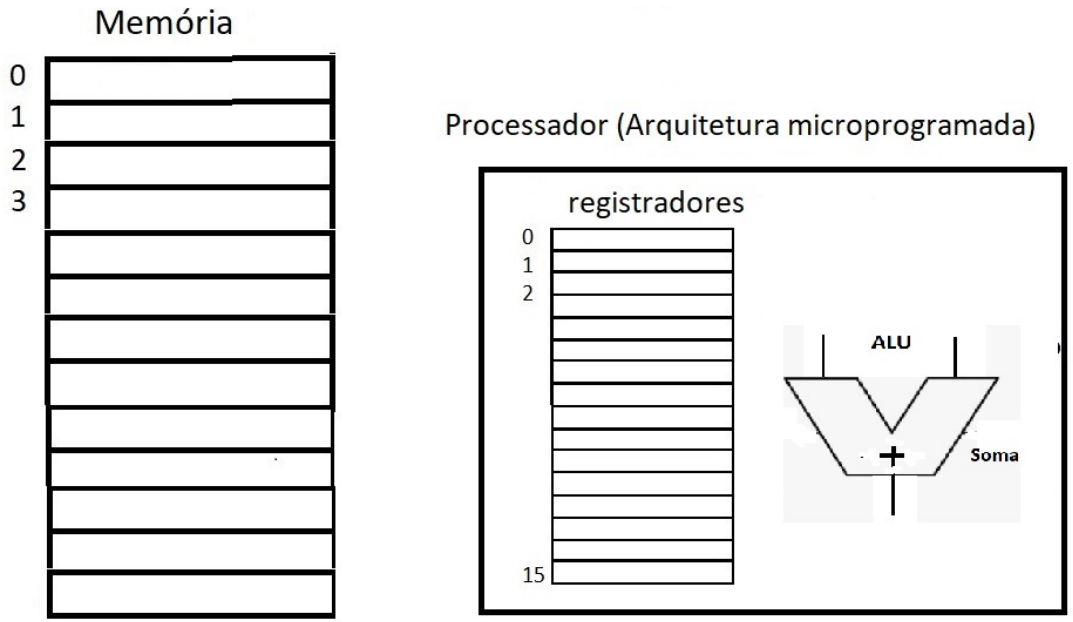
Como surgiu a microprogramação

- Os primeiros computadores tinham poucas instruções, todas implementadas em hardware.
- Maurice Wilkes (1951) introduziu a microprogramação, que permite
  - um conjunto grande de instruções de máquina (no nível convencional, i.e. conforme constam no manual de referência) usando, no entanto, um hardware simples capaz de executar apenas as chamadas microinstruções. A execução de uma instrução de máquina envolve, na verdade, a execução de muitas microinstruções.

Exemplos de máquinas CISC

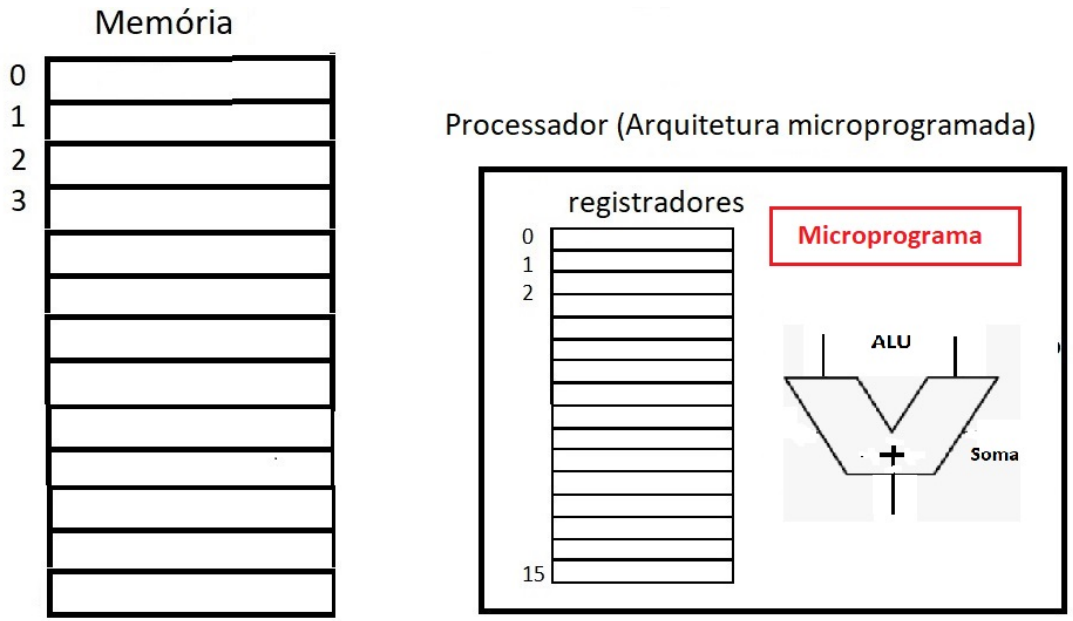
- O compilador de uma linguagem de alto nível gera código em instruções de máquina.
- Durante a sua execução, cada instrução de máquina é interpretada e envolve a execução de dezenas ou centenas de microinstruções.
- Os computadores que usam microprogramação são ditos da família CISC - Complex Instruction Set Computer.
- Exemplos: IBM 360, DEC VAX, Motorola 68030, família Intel como 8088, 80386, Pentium etc.

Uma explicação simples de CISC e microprogramação



- Para vocês sentirem um “sabor” do conceito de CISC e microprogramação, vejamos um exemplo simples:
- Suponha uma máquina cujo processador sabe fazer soma.
- No entanto, no manual de referências, consta uma instrução em linguagem de máquina chamada MULT x, y, z. Essa instrução multiplica o valor contido no endereço x da memória pelo valor no endereço y e coloca o produto no endereço z da memória.

Uma explicação simples de CISC e microprogramação



- No processador, quando se descobre que se trata de uma instrução MULT, então aciona-se a execução de um microprograma dentro do processador que soma *y* vezes o valor de *x* para produzir o produto que é armazenado em *z*.
- Assim, é possível ter um processador simples e um grande conjunto de instruções de máquina que, na verdade, envolve a execução de trechos de microinstruções dentro de um microprograma. O microprograma é previamente desenvolvido pelo fabricante e armazenado em uma ROM do processador.

Máquina MAC e arquitetura do processador MIC

- Nesta disciplina, vamos adotar um método de Tanenbaum que usa uma máquina fictícia, para fins didáticos, chamada MAC para ilustrar o conceito de microprogramação.
- MAC tem um conjunto de instruções de máquina. A execução de cada instrução é realizada no processador com arquitetura MIC (microprogramada).
- O microprograma de um processador real é bem mais complexo. Mas aqui vocês vão ter uma boa idéia dessa técnica.

A máquina MAC e a arquitetura de processador MIC

A máquina MAC apresenta as características:

- Memória com 4096 palavras de 16 bits (endereço 12 bits)
- Processador com 16 registradores, incluindo:
  PC (program counter)
  AC (acumulador)
  SP (stack pointer)
- Instrução de máquina de 16 bits.

Conjunto de instruções da máquina MAC

0000xxxxxxxxxxx LODD ac:=m[x]
0001xxxxxxxxxxx STOD m[x]:=ac
0010xxxxxxxxxxx ADDD ac:=ac+m[x]
0011xxxxxxxxxxx SUBD ac:=ac-m[x]
0100xxxxxxxxxxx JPOS if ac >= 0 then pc:=x
0101xxxxxxxxxxx JZZR if ac=0 tehn pc:=x
0110xxxxxxxxxxx JUMP pc:=x
0111xxxxxxxxxxx LOCO ac:x (0 <= x <= 4095)
1000xxxxxxxxxxx LODL ac:=m[sp+x]
1001xxxxxxxxxxx STOL m[x+sp]:=ac
1010xxxxxxxxxxx ADDL ac:=ac+m[sp+x]
1011xxxxxxxxxxx SUBL ac:=ac-m[sp+x]
1100xxxxxxxxxxx JNEG if ac<0 then pc:=x
1101xxxxxxxxxxx JNZE if ac not= 0 then pc:=x
1110xxxxxxxxxxx CALL sp:=sp-1; m[sp]:=pc; pc:=x
1111000000000000 PSHI sp:=sp-1; m[sp]:=m[ac]
1111001000000000 POPI m[ac]:=m[sp]; sp:=sp+1
1111010000000000 PUSH sp:=sp-1; m[sp]:=ac
1111011000000000 POP ac:=m[sp]; sp:=sp+1
1111100000000000 RETN pc:=m[sp]; sp:=sp+1
1111101000000000 SWAP tmp:=ac; ac:sp; sp:tmp
11111100yyyyyyyy INSP sp:=sp+y (0 <= y <= 255)

11111110yyyyyyyy DESP sp:=sp-y (0 <= y <= 255)

Não precisam decorar isso :-)

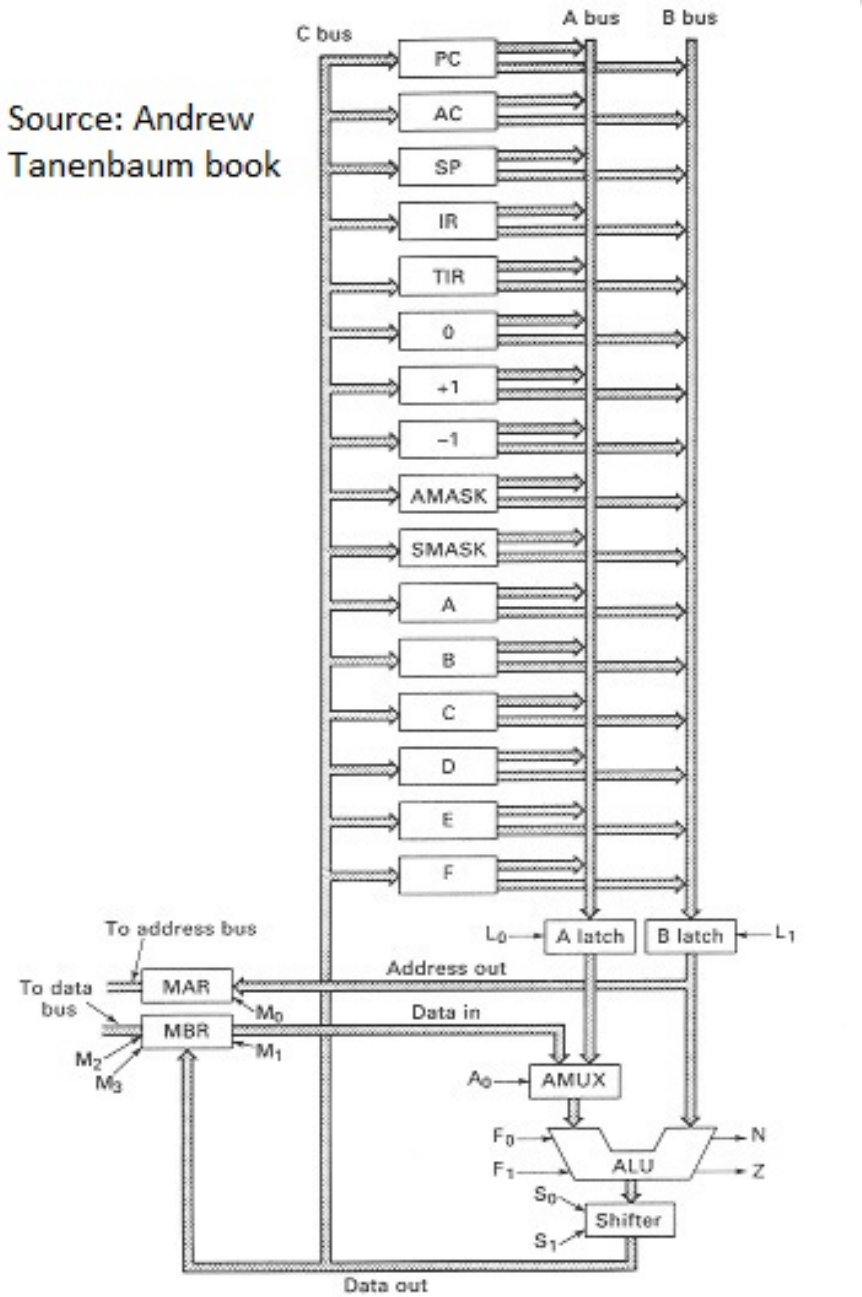
Apenas notem que, numa máquina real, este conjunto pode ser bem grande e complexo.

A arquitetura MIC

A arquitetura do processador (chamada MIC) é simples e não implementa as instruções de máquina diretamente.
Apresenta os seguintes componentes:

- Uma ALU capaz de fazer apenas 4 operações simples
- Um *shifter* para deslocar 1 bit para direita ou para esquerda
- 16 registradores
- 2 registradores denominados *latches*
- Um multiplexador MUX de duas entradas
- Três decodificadores 4-para-16
- Registradores MAR e MBR servindo de interface com a memória
- Um relógio de 4 fases

O datapath do processador

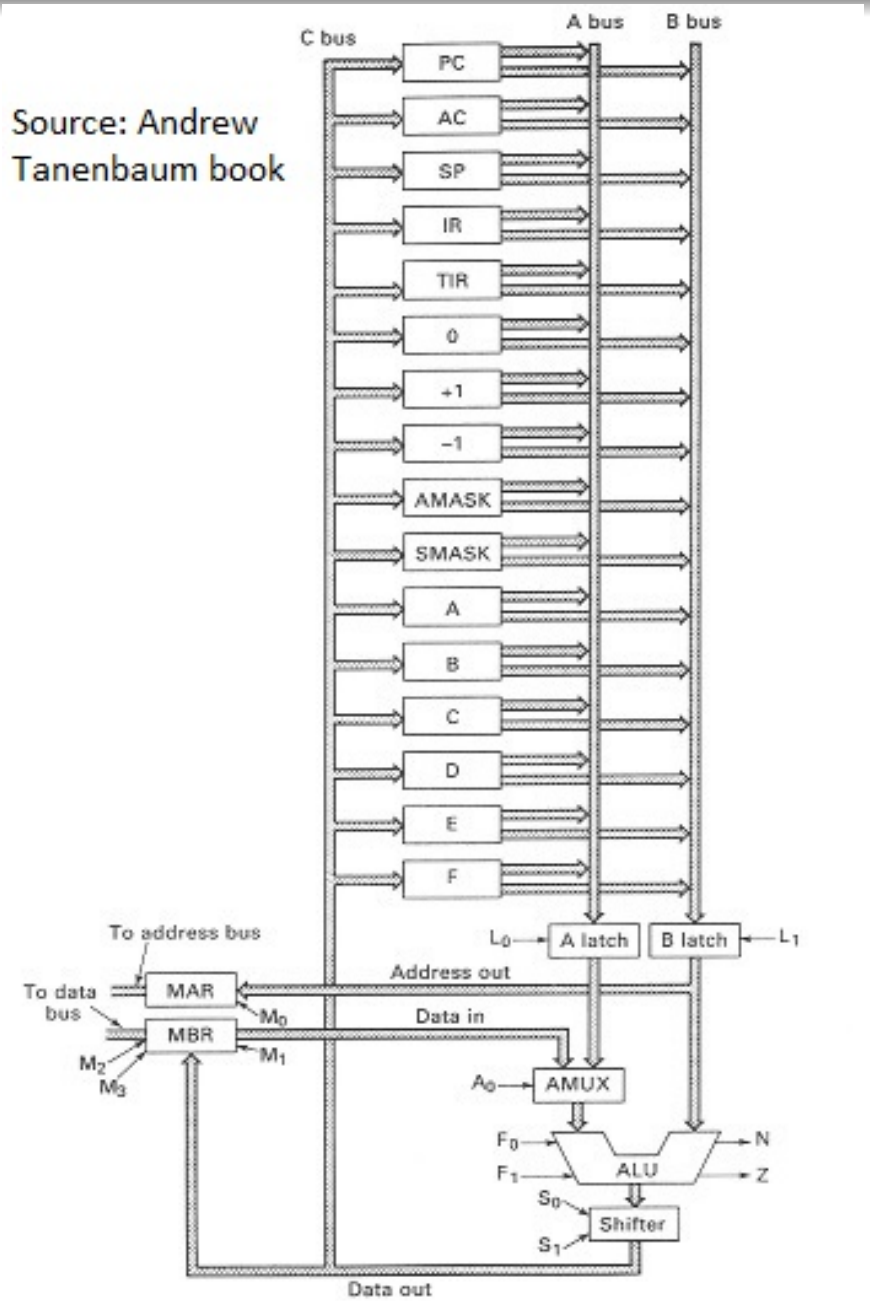


Datapath é a coleção de barramentos, registradores e unidades funcionais como ALU que realizam operações de processamento de dados.

Datapath, juntamente com a unidade de controle, compõem a unidade de processamento central CPU.

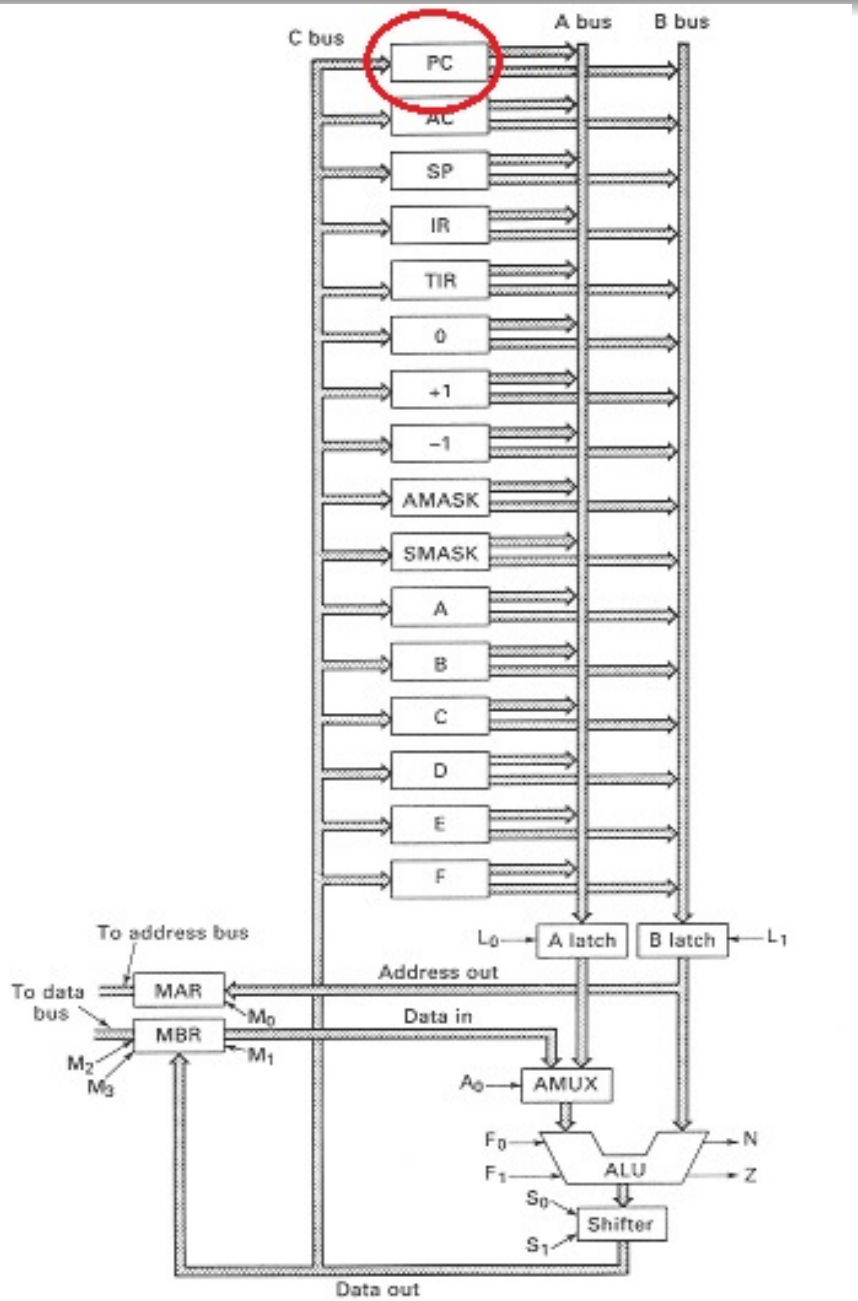
À esquerda: Visão geral do processador apresentando os componentes principais e os barramentos (ou *buses*): A, B, C.

Sinais de controle



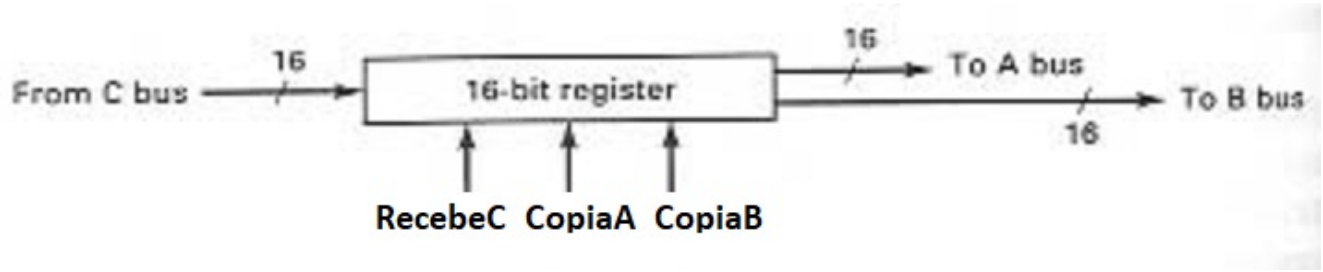
- Em cada ciclo, os componentes da arquitetura MIC são controlados por sinais de controle para executar as ações necessárias.
- Uma microinstrução basicamente é o conjunto desses sinais de controle.
- Começamos a explicar isso.

Cada componente e os sinais de controle



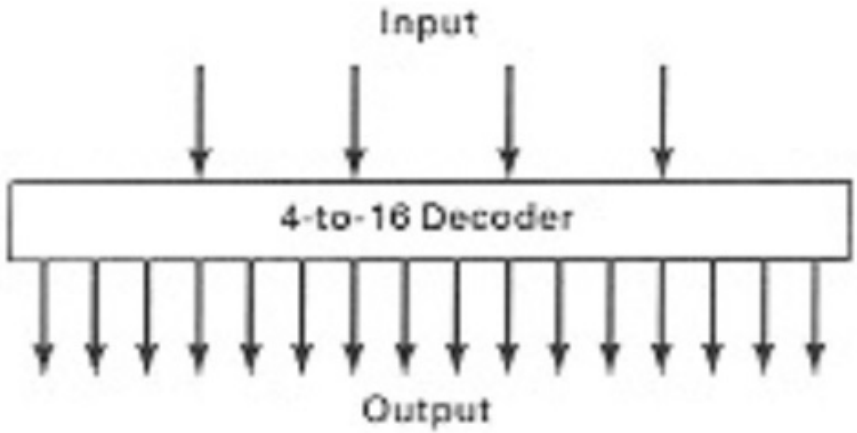
- Vejamos cada componente e seus sinais de controle.
- Primeiro vamos ver Registrador.
- São 16 Registradores, com nomes como PC, AC, SP, IR, etc..

Sinais de controle de cada registrador



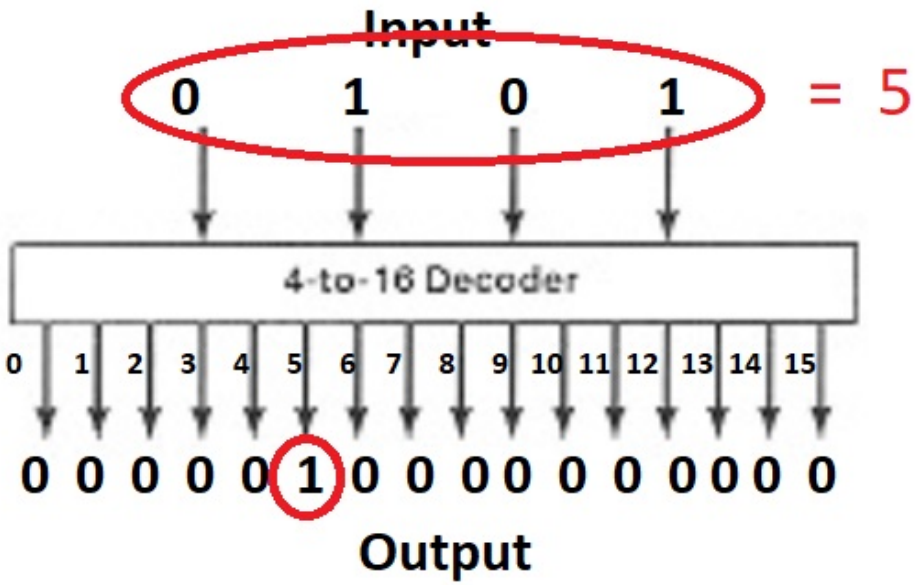
- São 16 registradores no processador.
- Cada um é controlado por 3 sinais de controle:
  - *RecebeC* $= 1$: valor do bus C é colocado dentro do registrador, apagando o valor anterior; caso contrário o valor do registrador não muda.
  - *CopiaA* $= 1$: valor do registrador é copiado no barramento A; caso contrário não copia.
  - *CopiaB* $= 1$: valor do registrador é copiado no barramento B; caso contrário não copia.
- Como são 16 registradores, temos um total de $16 \times 3 = 48$ sinais de controle.

Decodificador 4-para-16



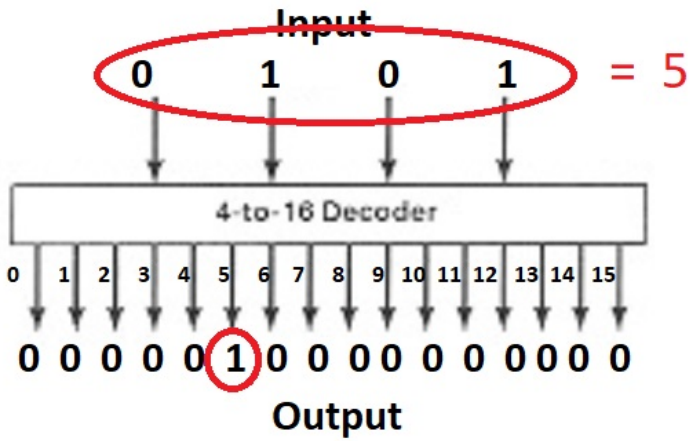
- O decodificador é uma lógica combinatória que vamos recordar a seguir.
- O decodificador não precisa de sinais de controle.
- Três decodificadores 4-para-16 são necessários, como veremos.

Decodificador 4-para-16



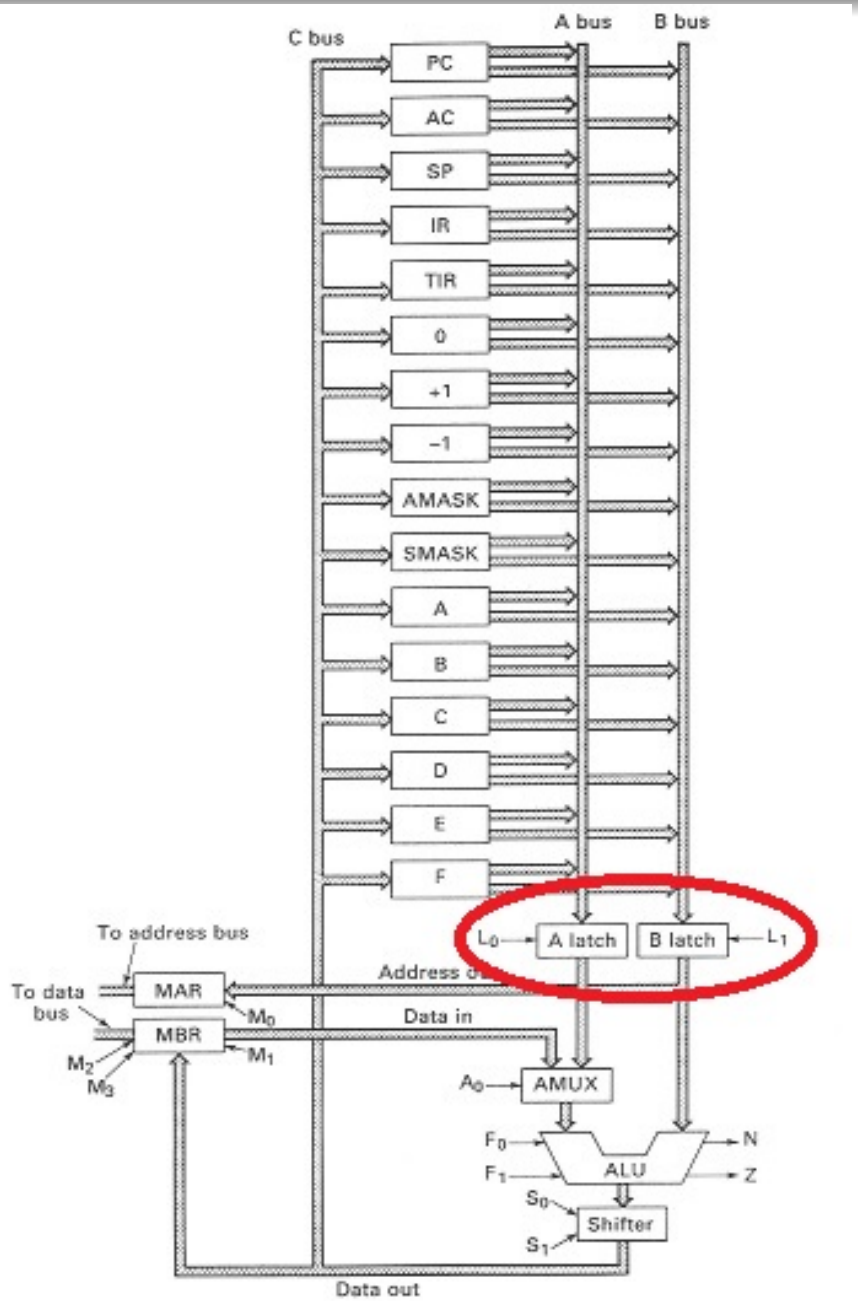
- O decodifica recebe uma entrada (e.g. 0101 ou 5 em decimal).
- A saída 5 vale 1, todas as demais saídas valem 0.
- Com o uso de um decodificador 4-para-16, podemos gerar 16 bits a partir de uma entrada de 4 bits.

Para que serve o decodificador 4-para-16



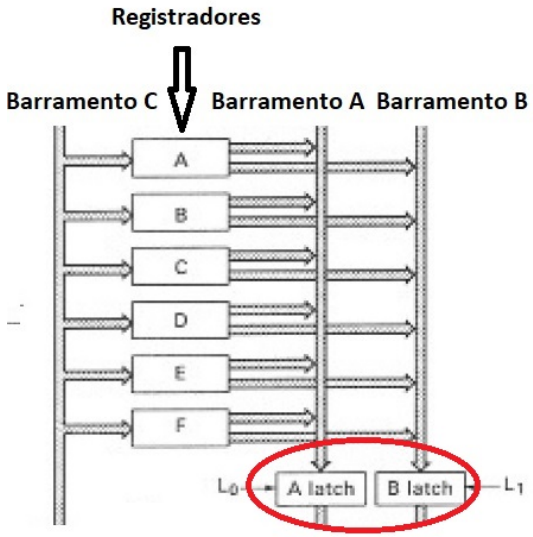
- 16 sinais são necessários para indicar o registrador que copia seu valor no Barramento A. Um deles vale 1 e os demais 0.
- Assim, ao invés de usar 16 sinais, podemos usar um decodificador 4-para-16 e gastar apenas 4 bits para indicar qual registrador deve copiar seu valor no Barramento A. No exemplo acima, o registrador 5. O decodificador gera os 16 bits de controle necessários.
- O mesmo vale para Barramento B. Usamos mais um decodificador.
- E também para Barramento C, usamos mais um decodificador para indicar o registrador deve receber o valor do Barramento C.

Cada componente e os sinais de controle



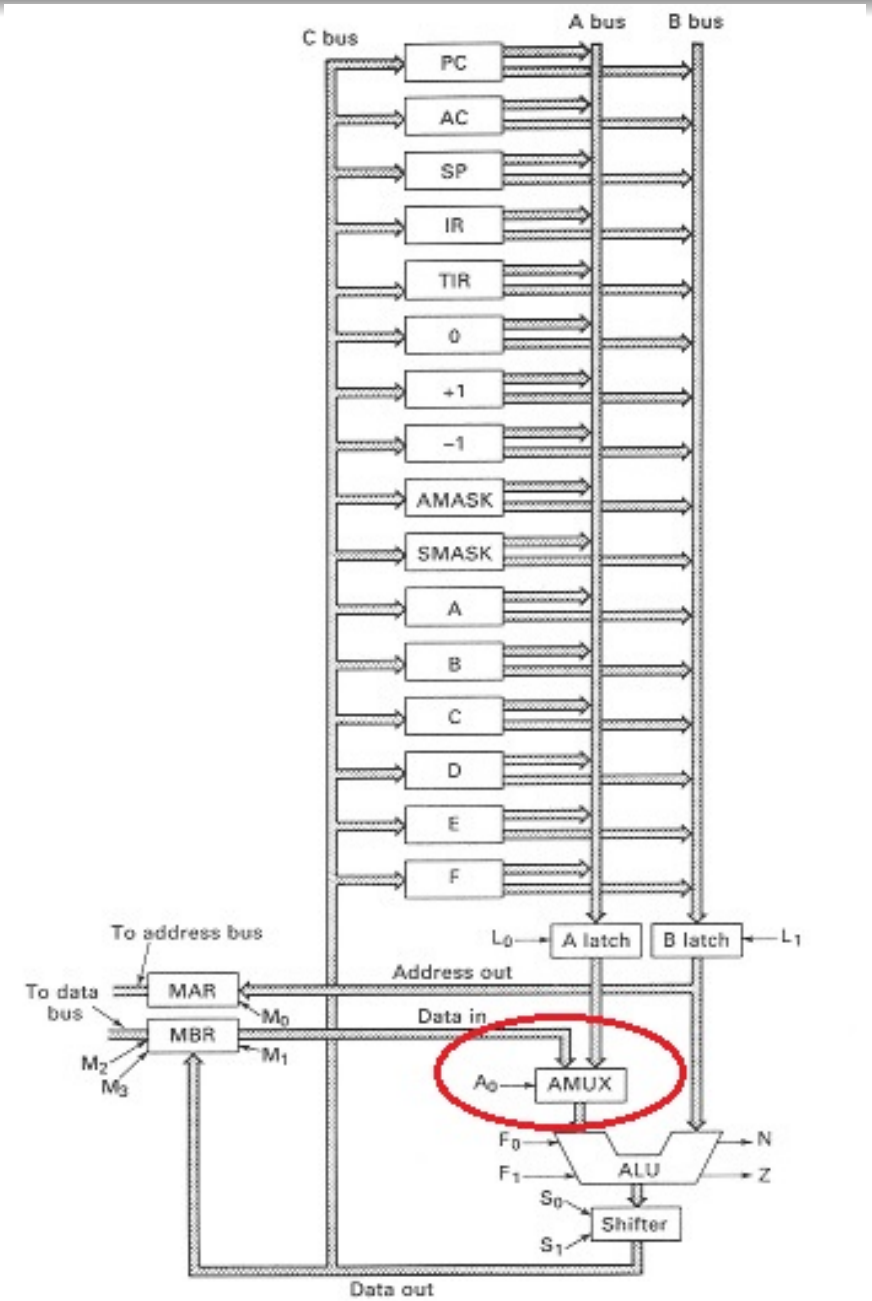
- Vejamos cada componente e seus sinais de controle.
- A Latch e B Latch.

Sinais de controle do A Latch e B Latch



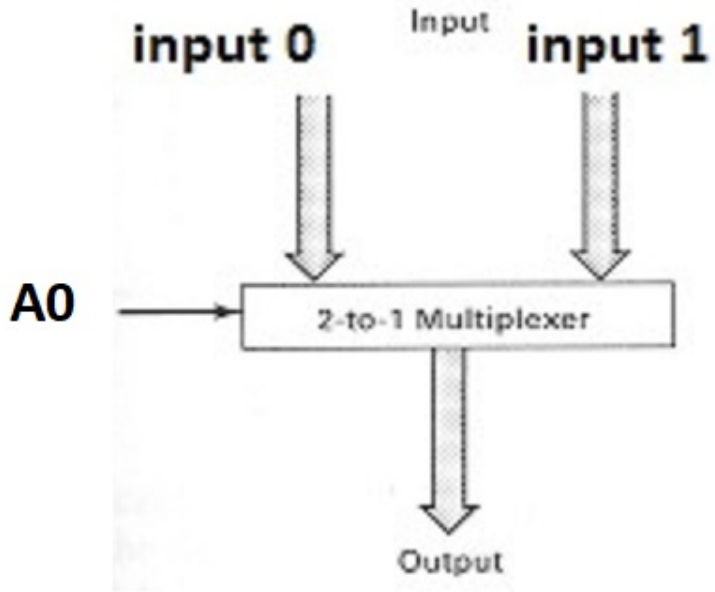
- A Latch é um registrador controlado por L_0. Quando o sinal $L_0 = 1$, ele captura o valor que está no barramento A.
- B Latch é um registrador controlado por L_1. Quando $L_1 = 1$, ele captura o valor que está no barramento B.
- Um valor de um registrador copiado ao barramento A precisa esperar L_0 valer 1 para poder prosseguir.
- Um valor de um registrador copiado ao barramento B precisa esperar L_1 valer 1 para poder prosseguir.

Cada componente e os sinais de controle



- Vejamos cada componente e seus sinais de controle.
- Multiplexador MUX de 2 para 1.

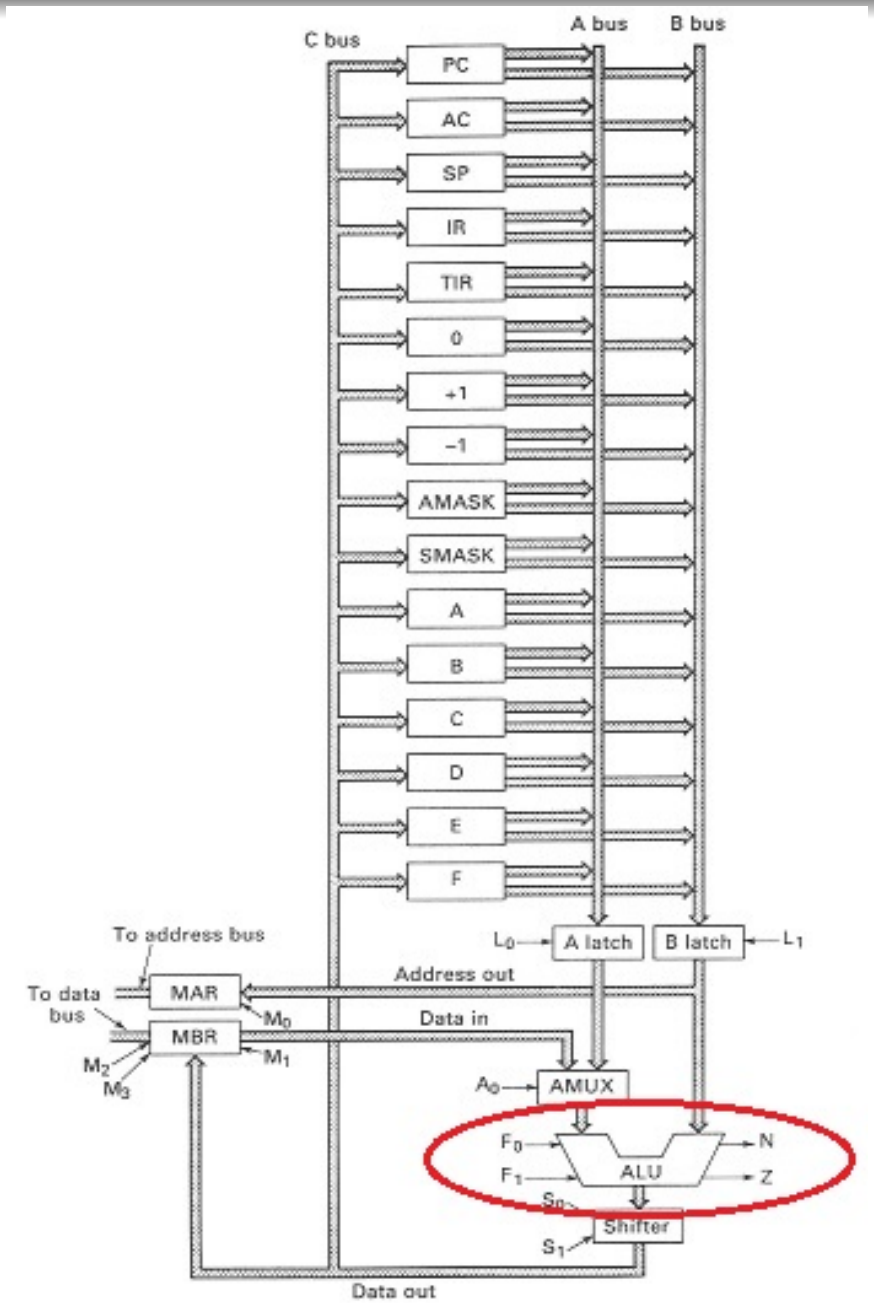
Sinais de controle do multiplexador MUX 2 para 1



O multiplexador (também chamado seletor) MUX é controlado por um sinal $A0$:

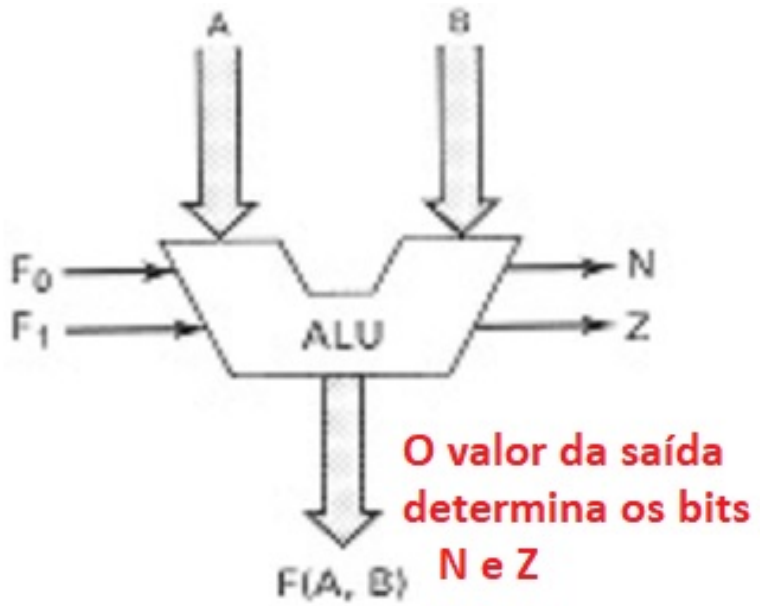
- $A0 = 0$: a saída do MUX seleciona o valor do input 0.
- $A0 = 1$: a saída do MUX seleciona o valor do input 1.

Cada componente e os sinais de controle



- Vejamos cada componente e seus sinais de controle.
- Unidade Aritmético-Lógica ALU.

Sinais de controle da ALU



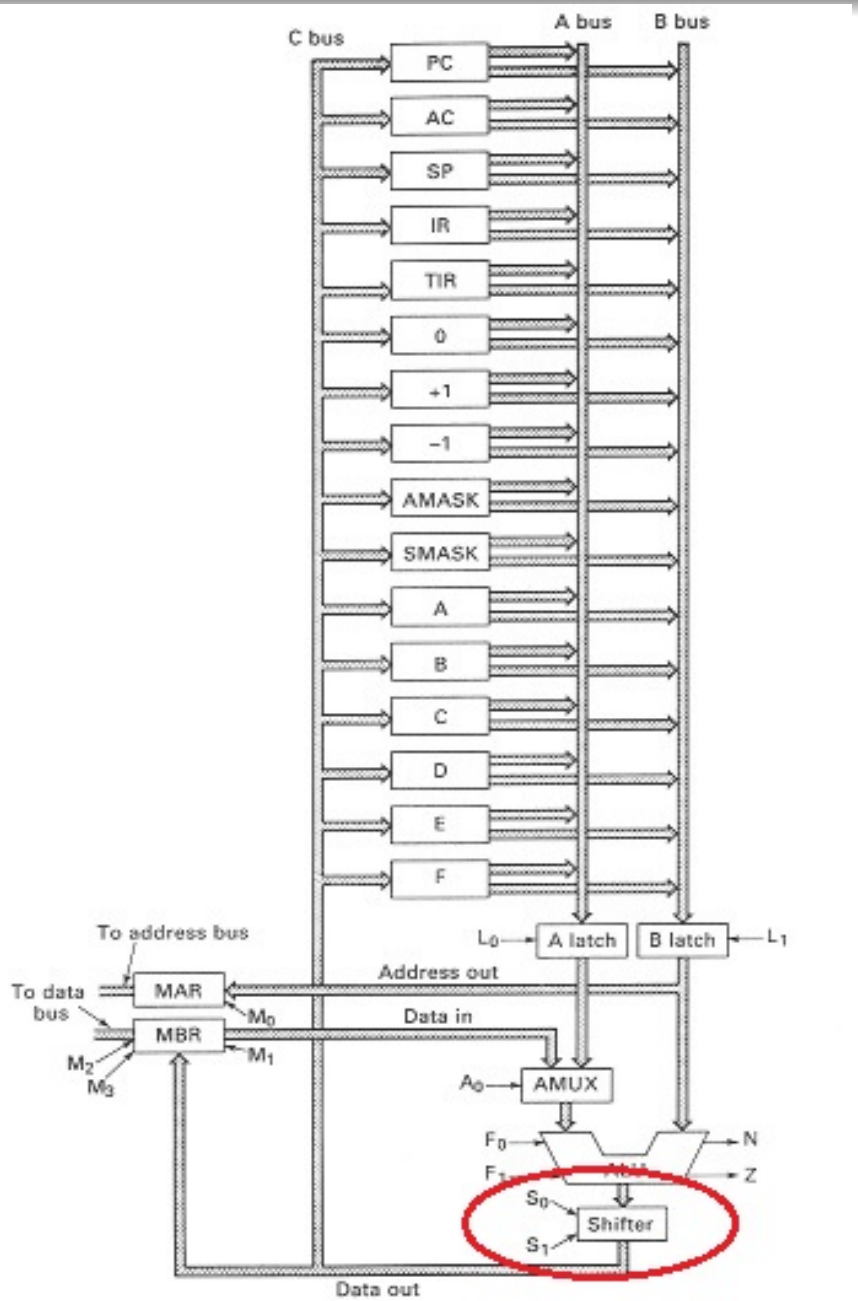
A unidade aritmético-lógica ALU sabe fazer 4 operações. Ela é controlada por dois sinais F_0 e F_1:

- $F_0F_1 = 00$: saída igual a A + B
- $F_0F_1 = 01$: saída igual a A and B
- $F_0F_1 = 10$: saída igual a A
- $F_0F_1 = 11$: saída igual ao complemento de A

ALU produz ainda duas saídas booleanas N e Z:

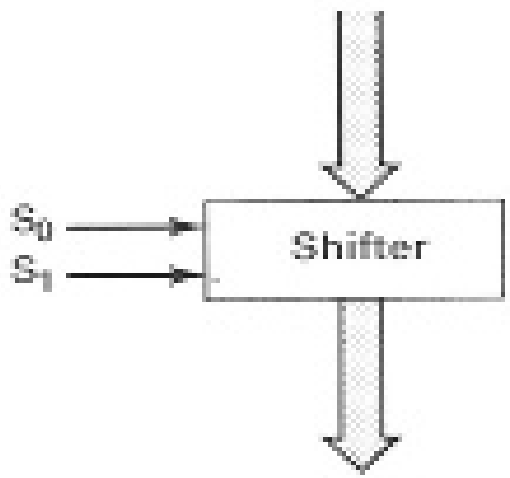
- $N = 1$ quando a saída é negativa.
- $Z = 1$ quando a saída é zero.

Cada componente e os sinais de controle



- Vejamos cada componente e seus sinais de controle.
- *Shifter* ou deslocador de bits.

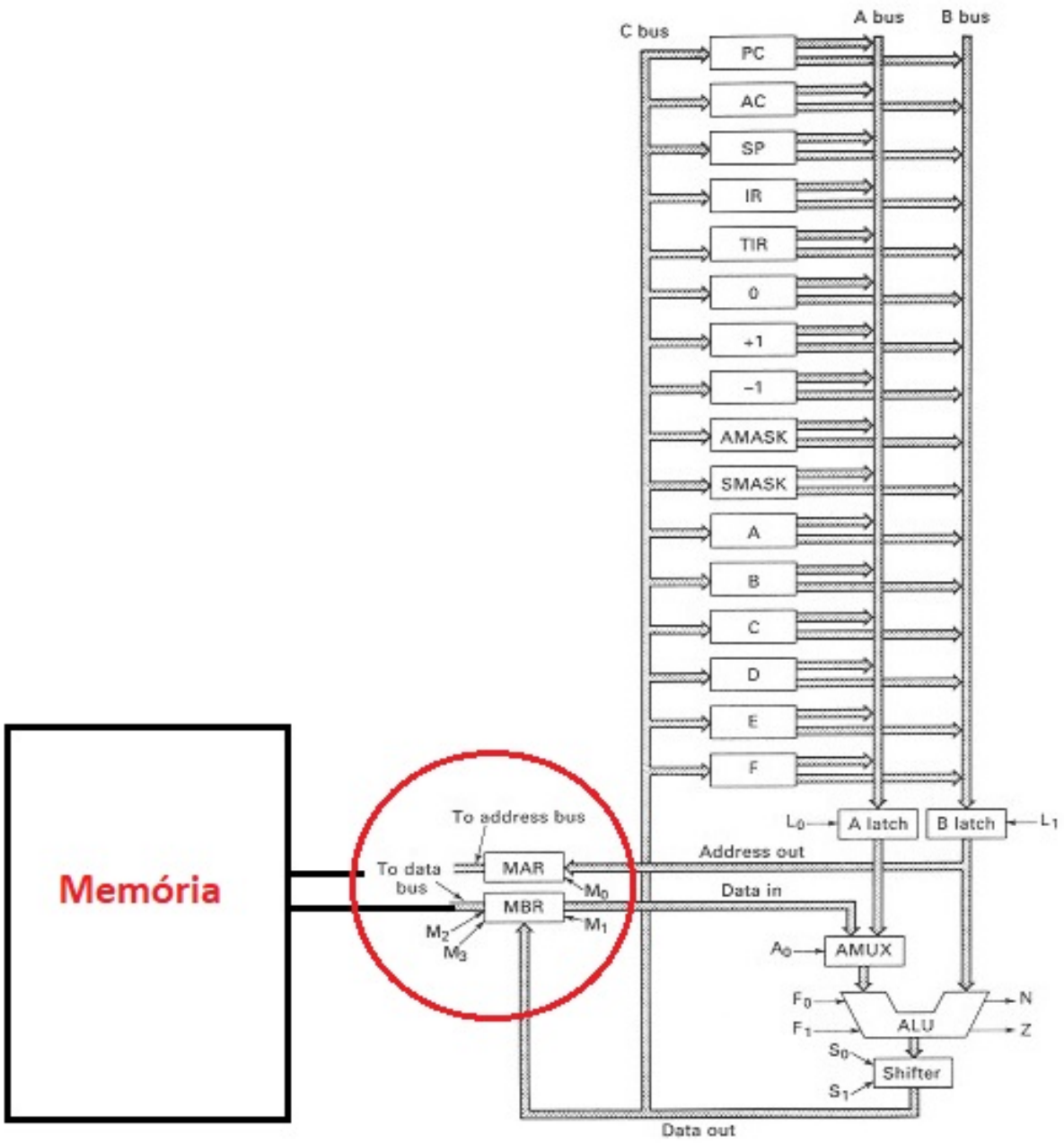
Sinais de controle do Shifter



O *Shifter* é um deslocador de bits.
Ele é controlado por dois sinais S_0 e S_1:

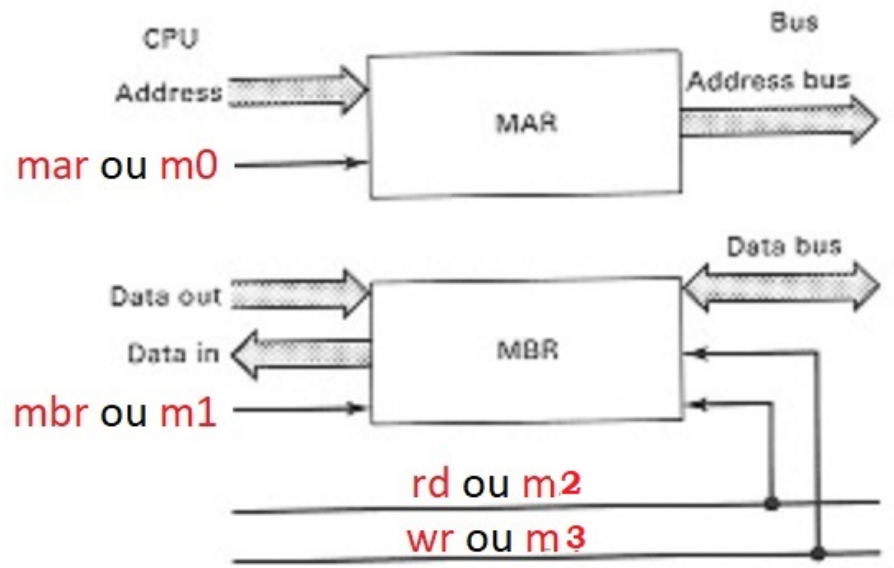
- $S_0 S_1 = 00$: saída igual à entrada (nada muda)
- $S_0 S_1 = 01$: desloca entrada de um bit para direita
- $S_0 S_1 = 10$: desloca entrada de um bit para esquerda
- $S_0 S_1 = 11$: não usada

Cada componente e os sinais de controle



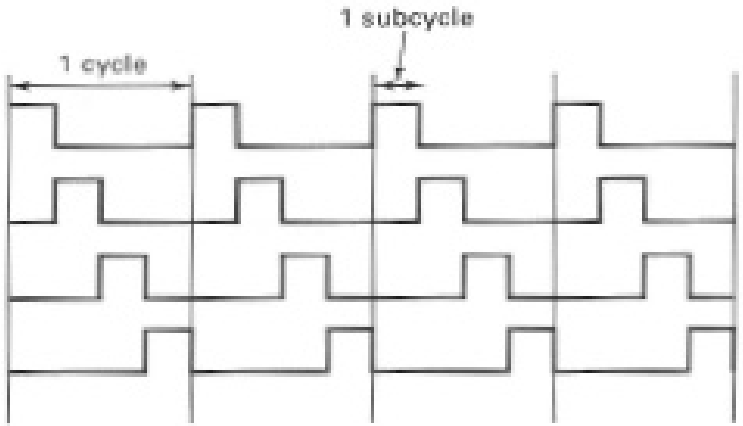
- Vejamos cada componente e seus sinais de controle.
- Registradores MAR e MBR (interface com a memória).

Sinais de controle de MAR e MBR



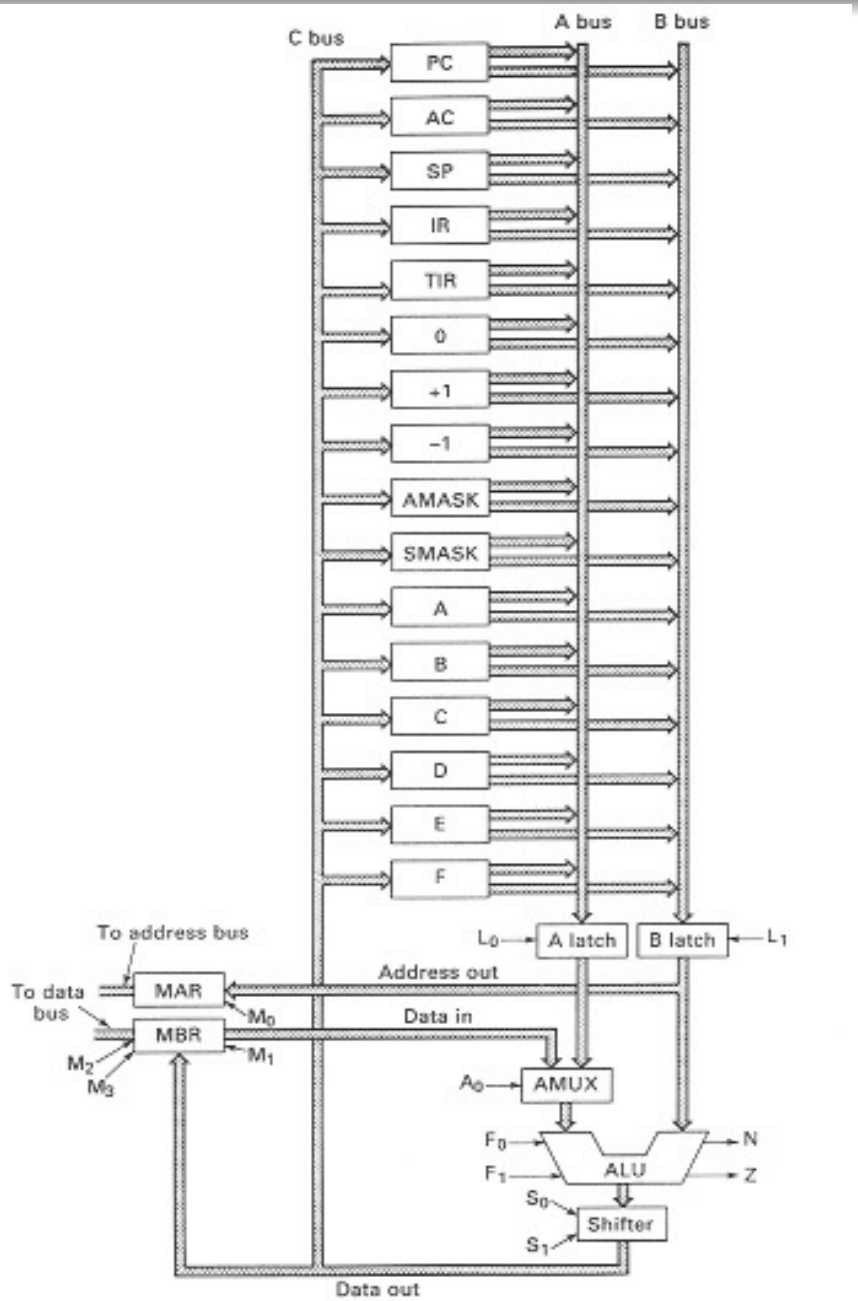
- MAR - Memory Address Register:
  - Quando *mar* ou *m0* = 1, um endereço é colocado dentro de MAR.
- MBR - Memory Buffer Register: 3 sinais de controle são usados.
  - Quando *mbr* ou *m1* = 1, um dado é colocado dentro de MBR.
  - *rd* ou *m2* =1 indica leitura
  - *wr* ou *m3* = 1 indica escrita.

Relógio de 4 fases



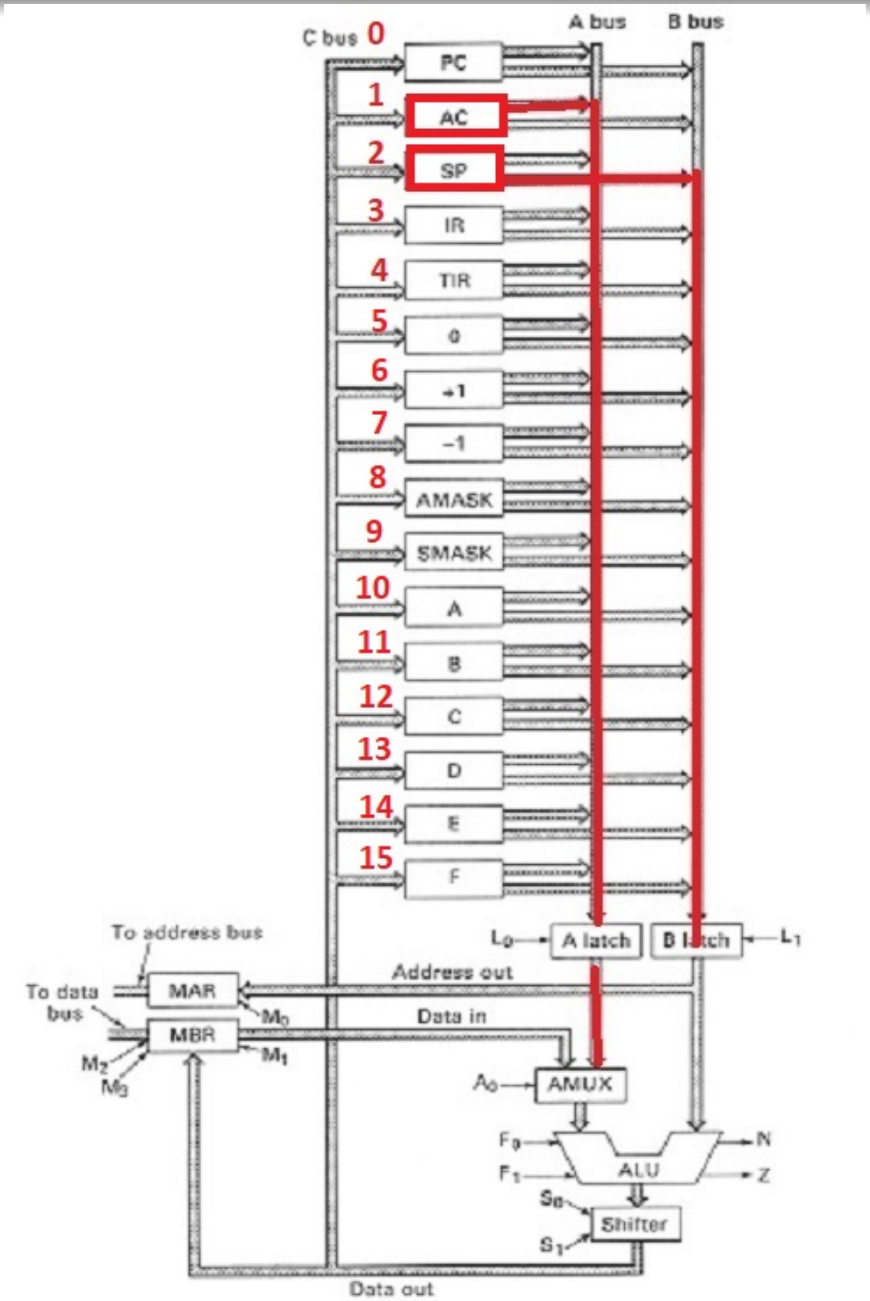
- O ciclo do relógio é dividido em 4 subciclos.
- Cada subciclo apresenta valor alto em um quarto do ciclo.
- Em cada instante, apenas um dos 4 subciclos possui valor alto.
- É usado para controlar e disciplinar o andamento na execução de uma microinstruções, conforme será visto.

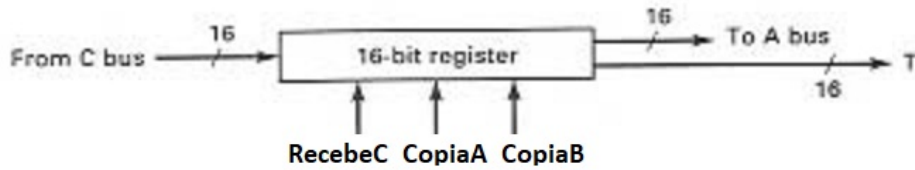
Sinais de controle



- Sinais de controle são usados para determinar o que é feito no processador durante um ciclo.
- Exemplo: Queremos escrever o valor do registrador AC na memória de endereço SP.
- Vejamos como podemos fazer isso, usando sinais de controle adequados.

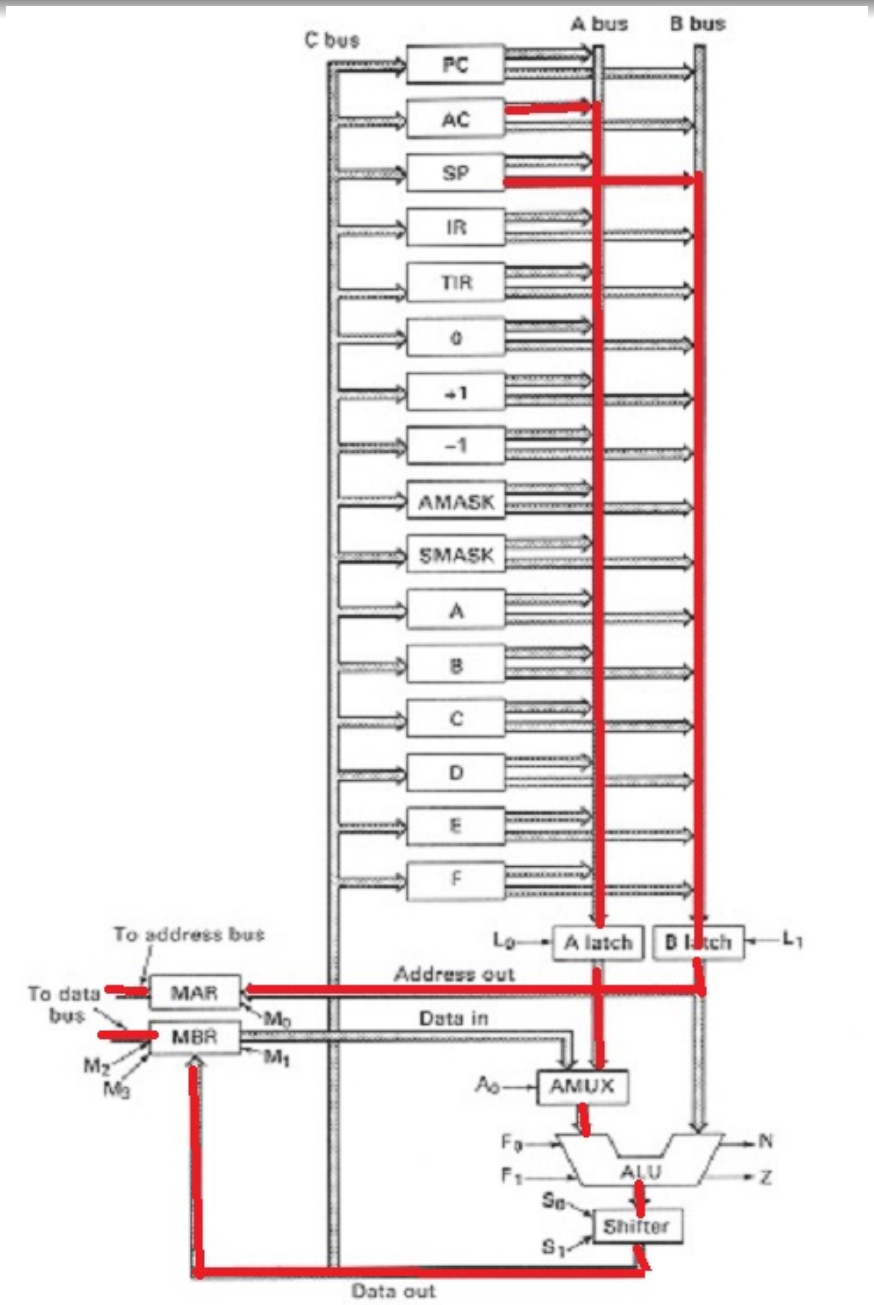
Sinais de controle - selecionam quais registradores

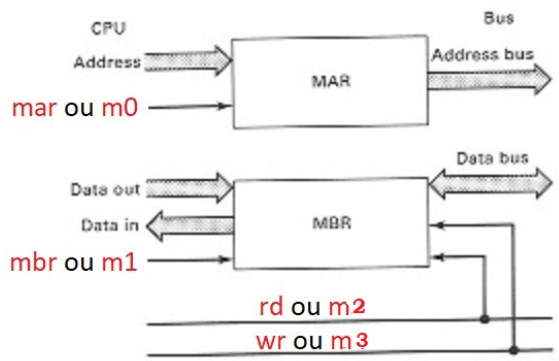




- Lembrem-se dos sinais de controle dos registradores?
- Eles indicam qual registrador deve copiar o seu valor no Barramento A, e qual no Barramento B, e qual registrador deve receber o valor do Barramento C.
  - Fazemos *CopiaA* = 1 do registrador 1 (AC) para copiar o seu valor no barramento A
  - Fazemos *CopiaB* = 1 do registrador 2 (SP) para copiar seu valor no barramento B.
  - Fazemos $L_0 = L_1 = 1$ do A latch e B latch

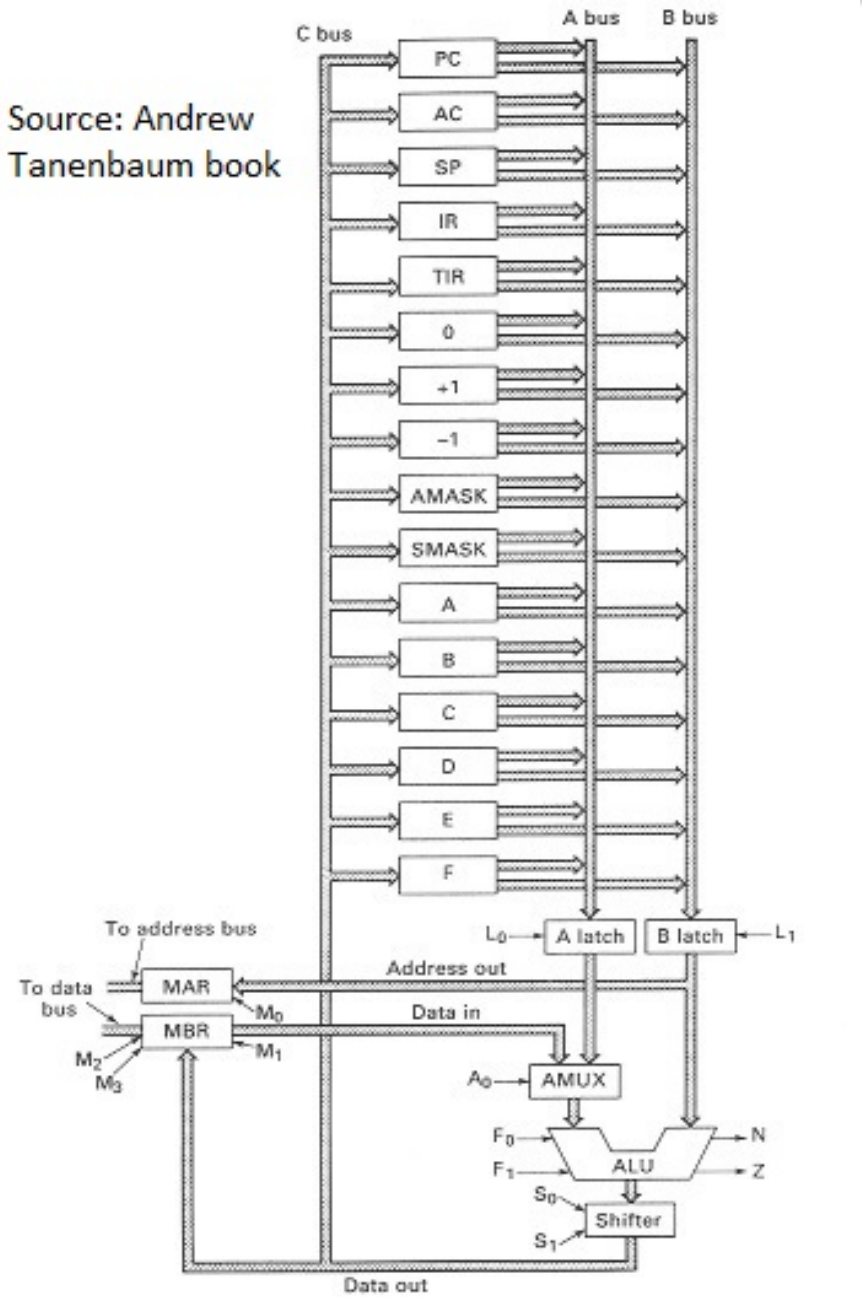
Sinais de controle - MUX, ALU, Shifter, MAR e MBR





- Fazemos $A0 = 1$ do MUX para selecionar a entrada da direita
- $F_0F_1 = 10$ da ALU para repetir o valor de A. (A é o ramo esquerdo que entra na ALU).
- $S_0S_1 = 00$ do shifter para não mudar nada
- $mbr = 1$ do MBR para receber o valor AC.
- $mar = 1$ do MAR para receber o valor SP.
- $wr = 1$ para indicar escrita na memória.

A arquitetura MIC

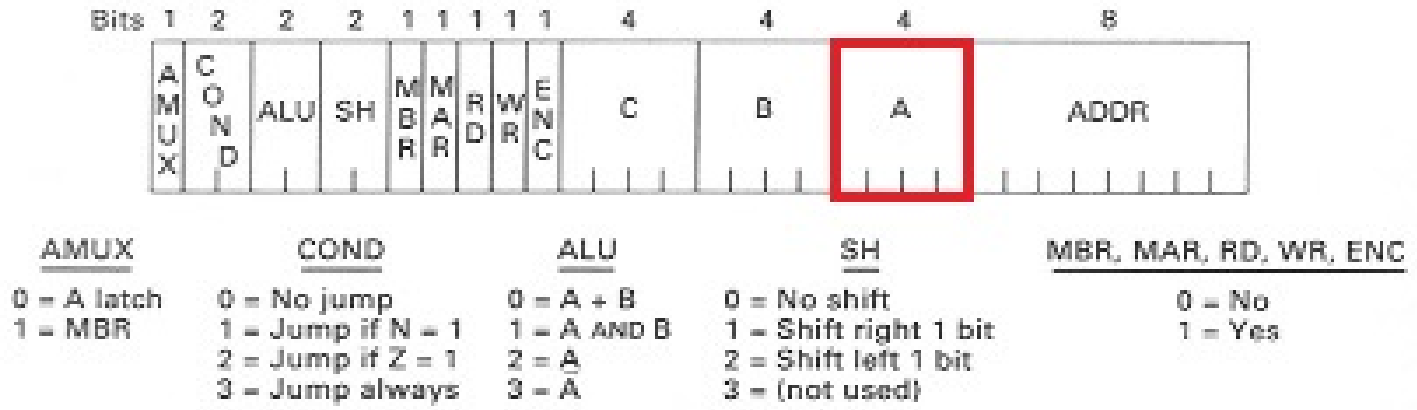


- Vimos todos os sinais de controle.
- Agora vamos ver o formato de uma microinstrução.

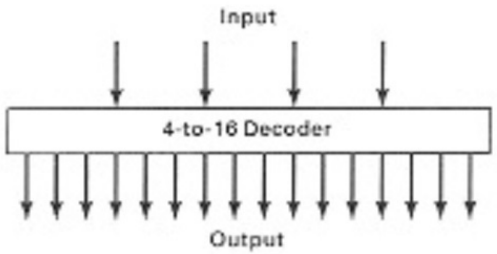
Formato da microinstrução

- Uma microinstrução da arquitetura MIC é o conjunto de sinais de controle para o datapath durante um ciclo.
- Um formato possível de uma microinstrução é usar 60 bits:
  - 16 sinais p/ controlar a cópia de um valor no barramento A
  - 16 sinais p/ controlar a cópia de um valor no barramento B
  - 16 sinais p/ controlar a carga de um registrador pelo 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 (*mar*, *mbr*, *rd* e *wr*), também denominados (M_0, M_1, M_2, M_3) nas figuras.
  - 1 sinal para controlar o MUX (A0)
  - 1 sinal ENC (enable C) para indicar se o resultado calculado deve ser carregado do Barramento C para algum registrador. Esse sinal será melhor explicado adiante.

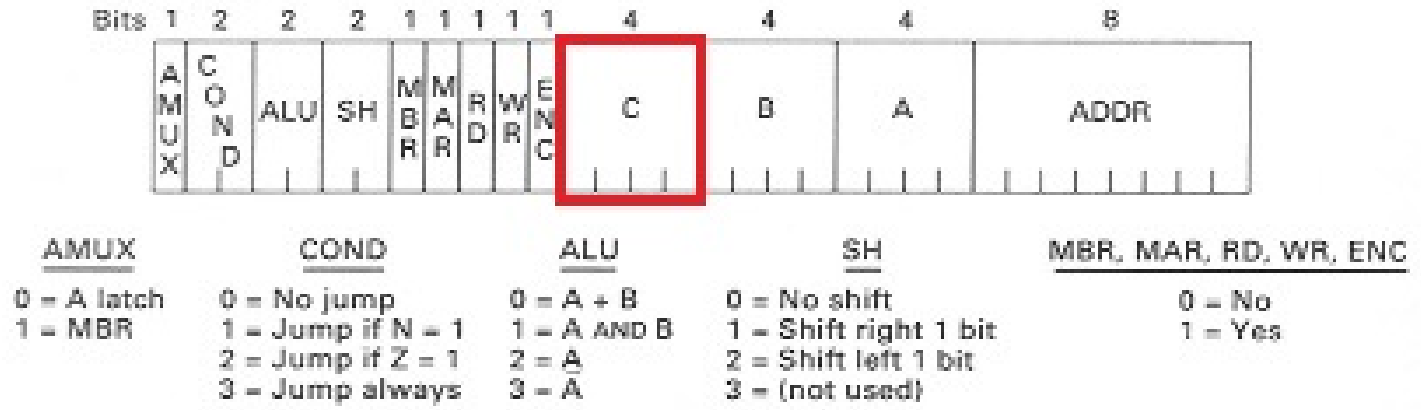
Formato da microinstrução



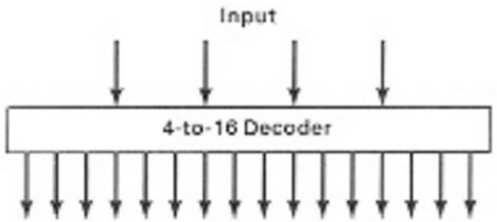
- Ao invés de gastar 16 bits para controlar a cópia de um valor de um registrador no barramento A, podemos usar um decodificador gastando apenas 4 bits para gerar os 16 valores. O campo A acima especifica os 4 bits para o decodificador.
- O mesmo vale para a carga no barramento B. Economizamos 24 bits.



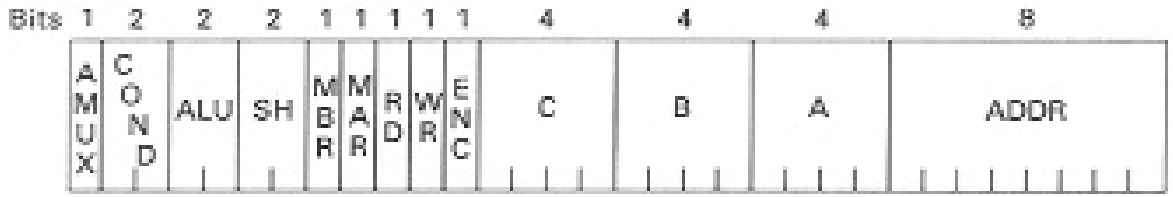
Formato da microinstrução



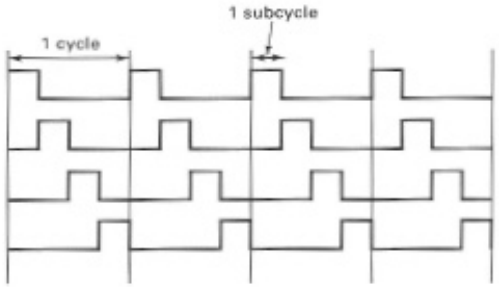
- Para gerar os 16 bits para indicar se um registrador deve receber o valor do barramento C, usamos um outro decodificador. Mas isso cria o seguinte problema.
- Um os 16 bits gerados pelo decodificador sempre vale 1 e isso forçaria um dos 16 registradores a receber valor do barramento C. Para evitar isso, fazemos o sinal ENC (enable C) = 0. Um registrador só recebe valor do barramento C se ENC = 1.



Formato da microinstrução

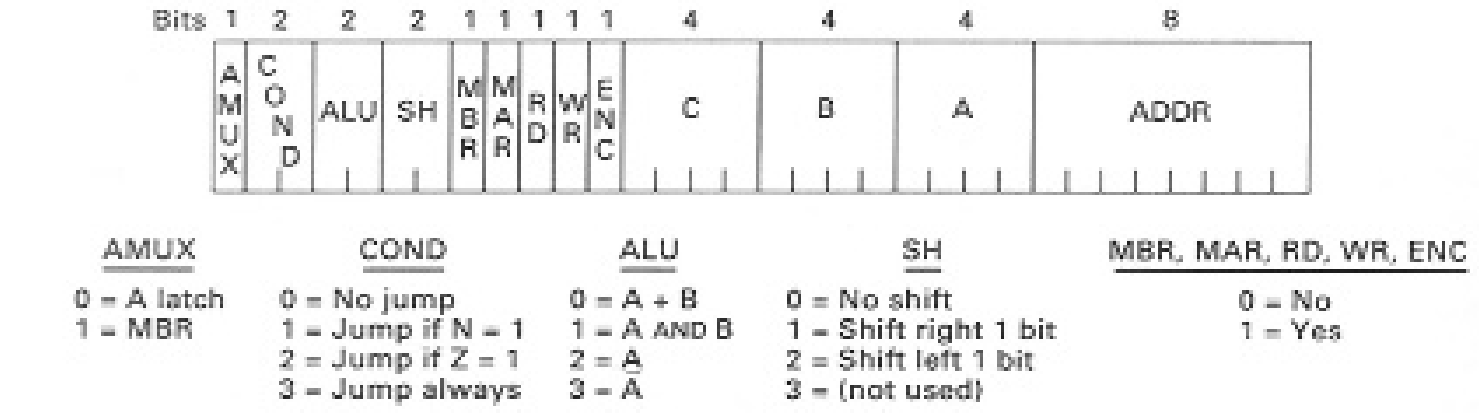


AMUX	COND	ALU	SH	MBR, MAR, RD, WR, ENC
0 = A latch 1 = MBR	0 = No jump 1 = Jump if N = 1 2 = Jump if Z = 1 3 = Jump always	0 = A + B 1 = A AND B 2 = A 3 = A	0 = No shift 1 = Shift right 1 bit 2 = Shift left 1 bit 3 = (not used)	0 = No 1 = Yes



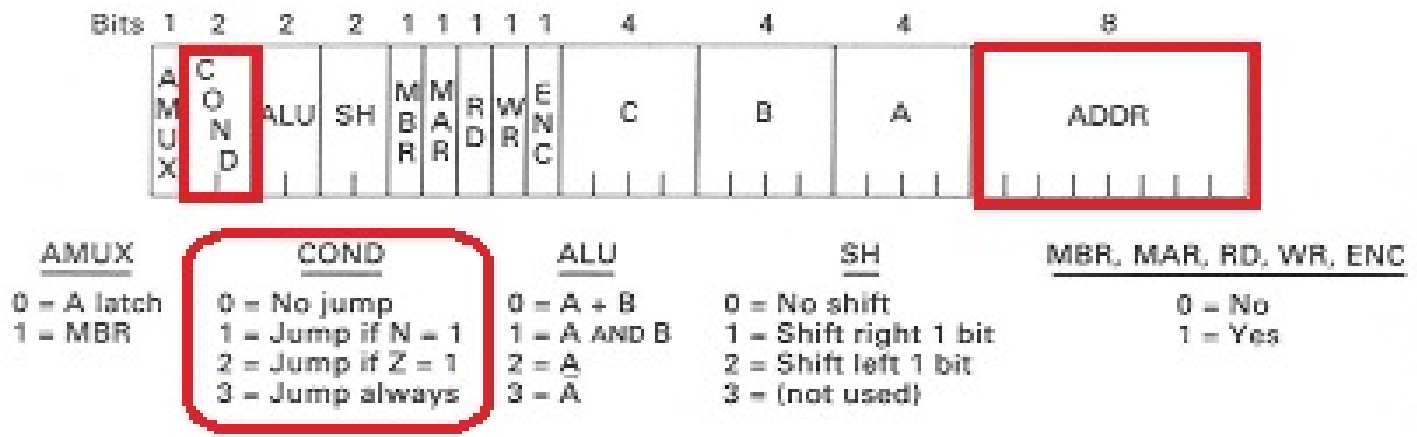
- Os 2 bits para controlar *A latch* e *B latch* devem valer 1 no segundo subciclo (num ciclo composto de 4 subciclos).
- Assim, podem ser substituídos pela fase 2 do relógio de 4 fases. Economizamos mais 2 bits.

Formato da microinstrução



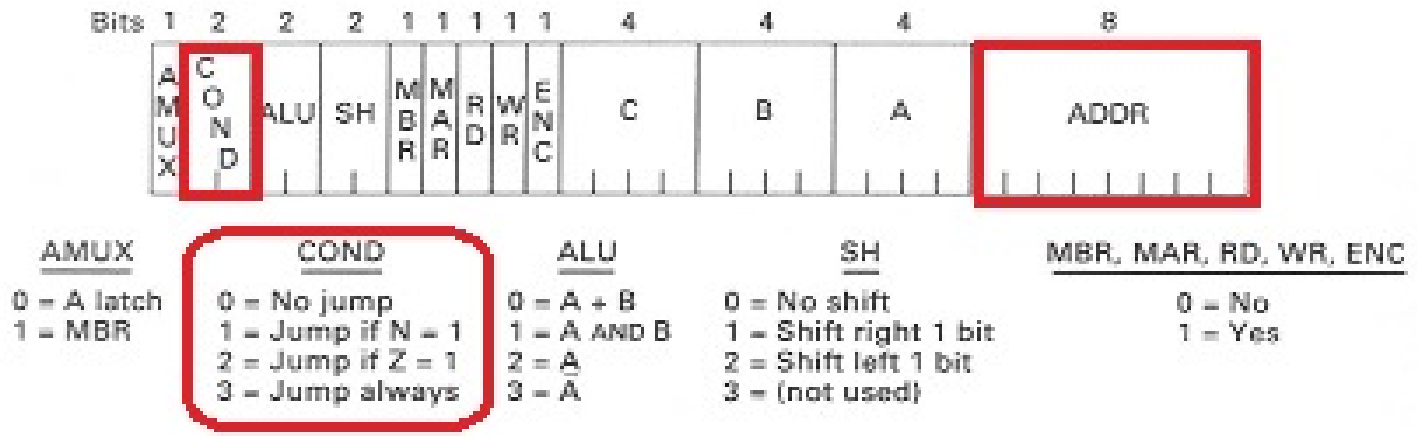
- Linguagens de alto nível possuem comandos como “if ... then ... else ...” ou “for i = 0 to 999 do ...”. A boa prática de programação estruturada evita usar desvios.
- O código gerado por um compilador é em instruções de máquina, de baixo nível. Desvios aí são inevitáveis e o código de baixo nível usa desvios com frequência. Por exemplo, o código gerado para “for i = 0 to 999 do ...” precisa usar um desvio para voltar ao início do laço. O código para “if ... then ... else ...” também vai ter desvios.

Formato da microinstrução



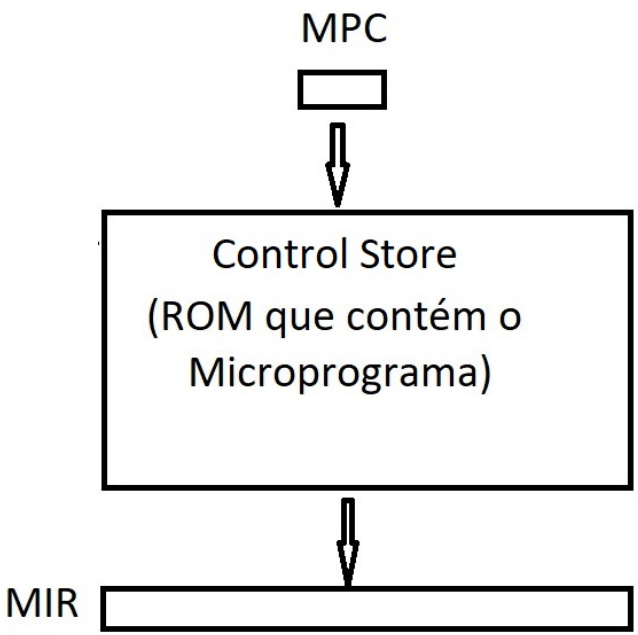
- Isso também acontece com microinstruções, de nível mais baixo ainda. No microprograma, há muitos desvios. Portanto no formato de microinstrução, já há um campo ADDR preparado para um endereço de desvio, se houver.
- COND e ADDR (usados para desvio condicional ou incondicional): controlar qual a próxima microinstrução a ser executada.
  - ADDR: endereço da próxima microinstrução
  - COND: condição para que a próxima microinstrução seja aquela dada por ADDR

Formato da microinstrução



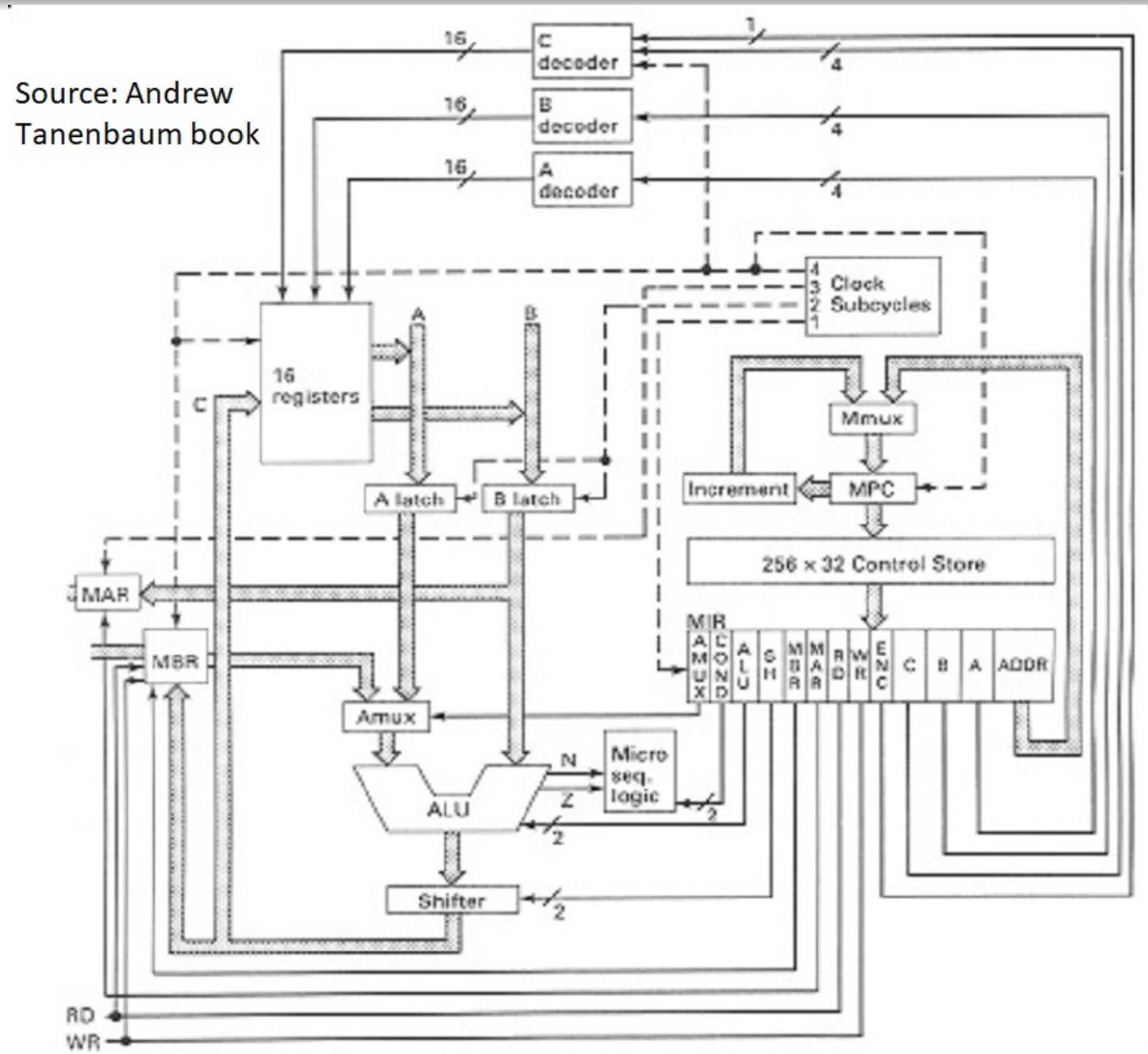
- Normalmente, após a execução de uma microinstrução, é executada a próxima, na sequência.
- Mas pode haver um desvio, dependendo do campo COND, usado em conjunto com ADDR.
  - Se COND = 0, então não há desvio.
  - Se COND = 1, então desvia para ADDR se N = 1.
  - Se COND = 2, então desvia para ADDR se Z = 1.
  - Se COND = 3, então desvia para ADDR, incondicionalmente.
- Uma microinstrução tem portanto 32 bits.

Microprograma

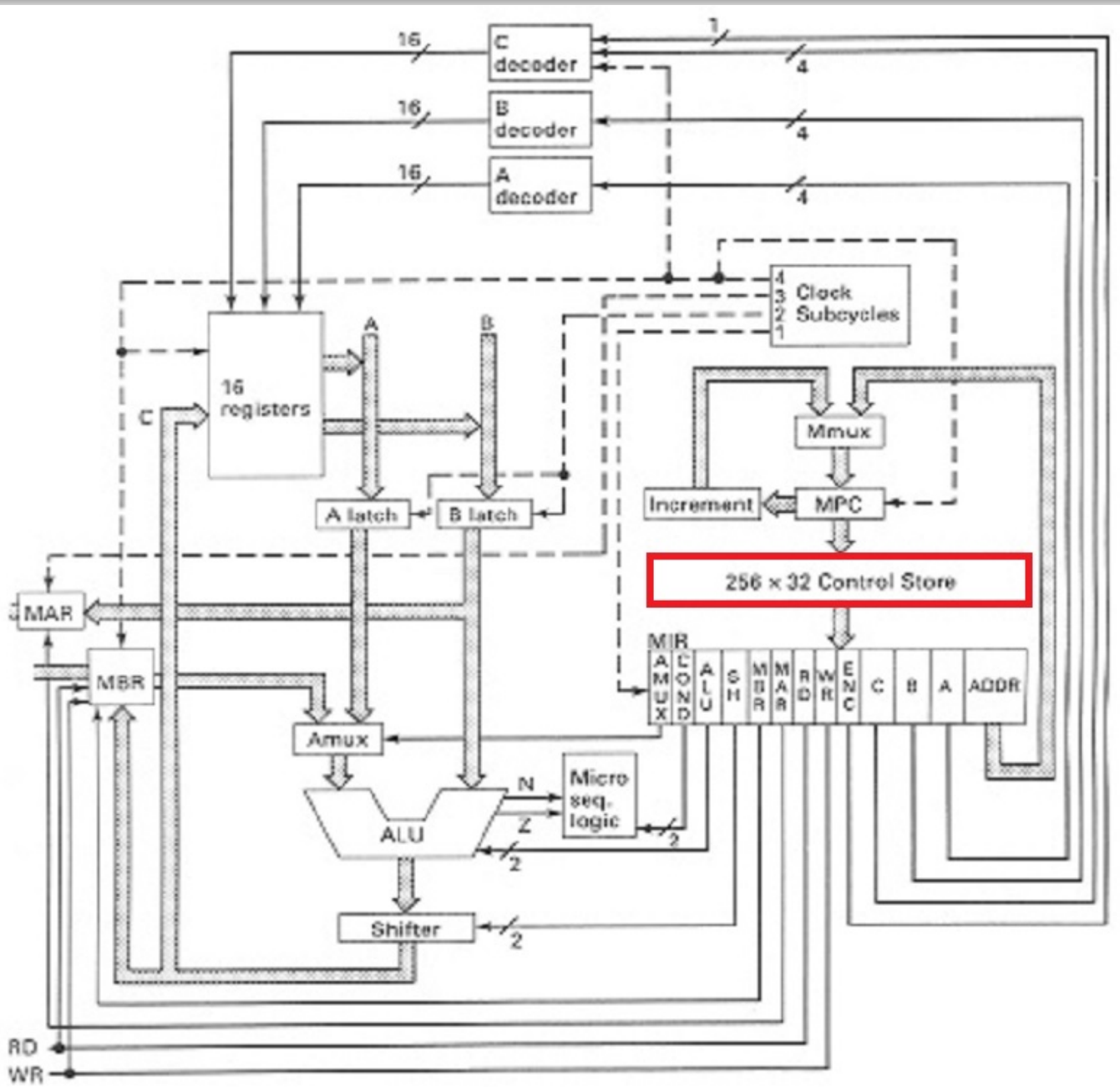


- O microprograma é um conjunto de microinstruções.
- Ele é armazenado numa memória ROM no processador, chamada *control store*.
- Há um MPC (*micro program counter*) que aponta para a próxima microinstrução a executar dentro do control store.
- Há um registrador chamado MIR (*micro instruction register*) que contém a microinstrução em execução.

Arquitetura MIC completa com control store

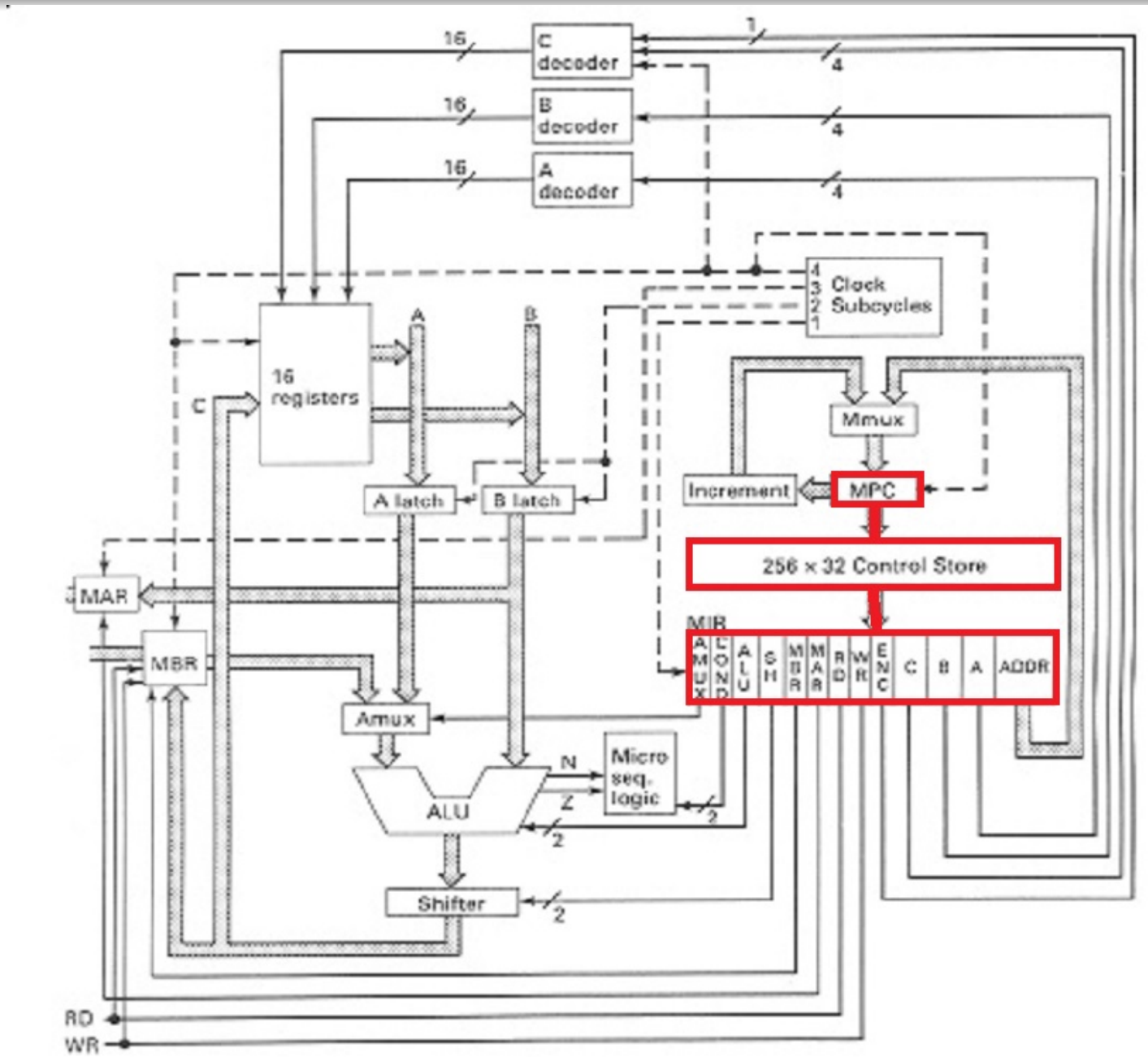


Arquitetura MIC completa com control store



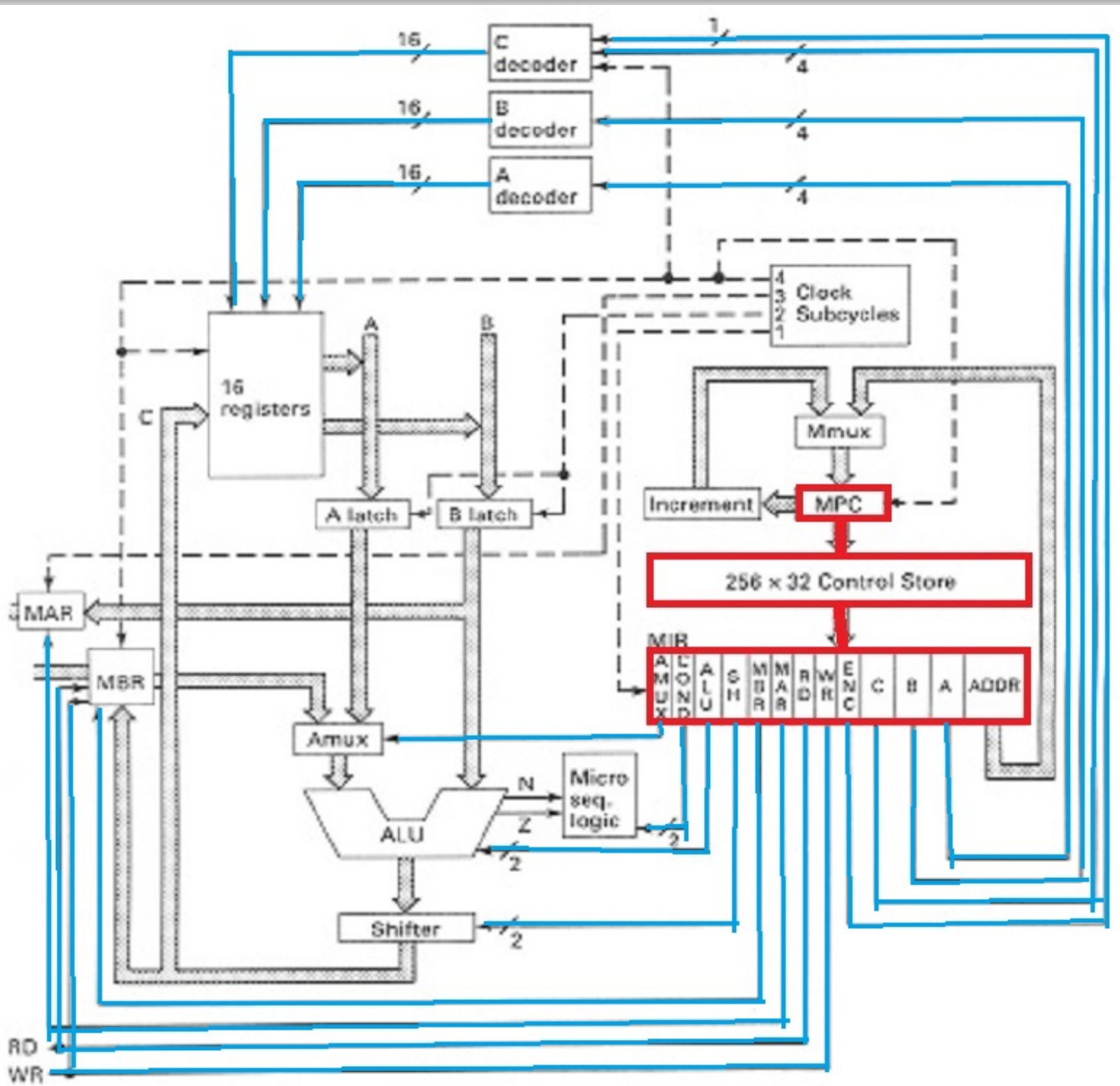
- O control store contém o micropro-grama.

Arquitetura MIC completa com control store



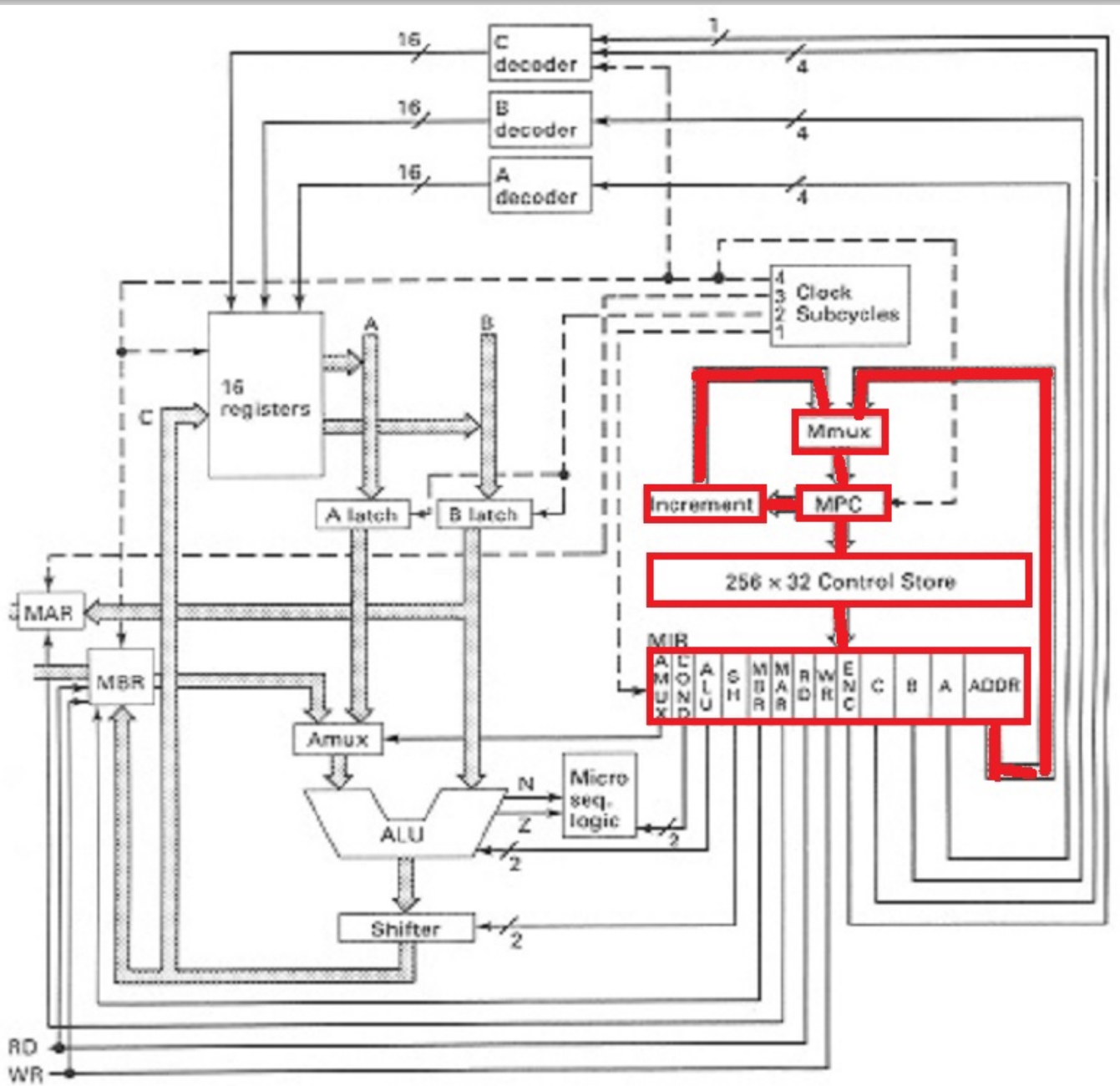
- MPC aponta para a próxima microinstrução a executar.
- A microinstrução endereçada por MPC é trazida para MIR para executar em um ciclo.

Arquitetura MIC completa com control store



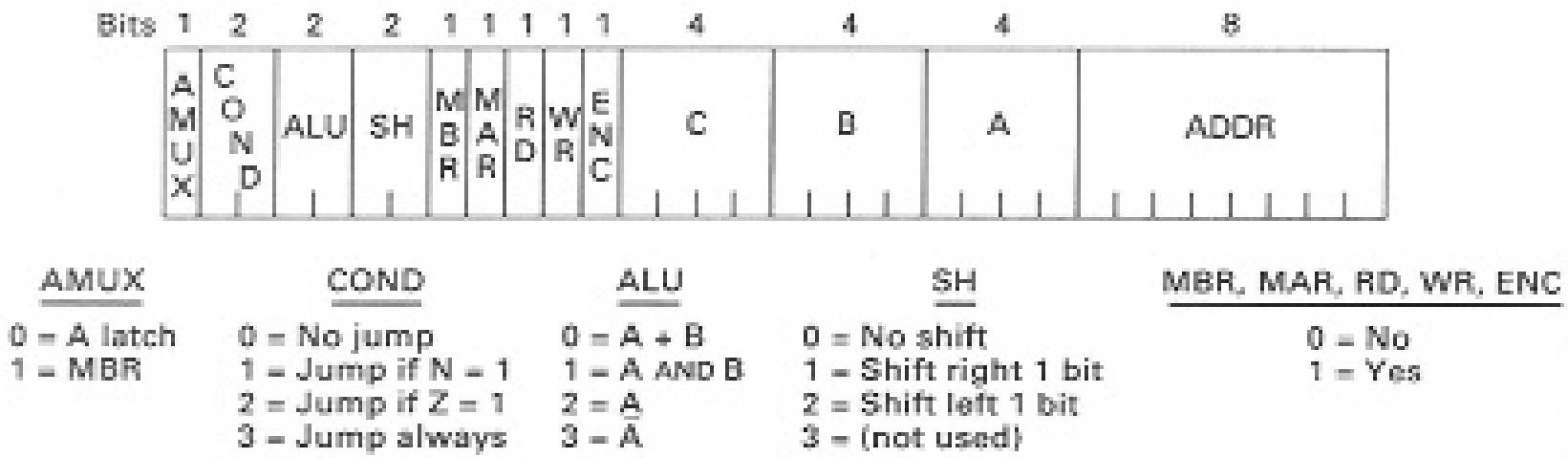
- MPC aponta para a próxima microinstrução a executar.
- A microinstrução endereçada por MPC é trazida para MIR para executar em um ciclo.

Arquitetura MIC completa com control store



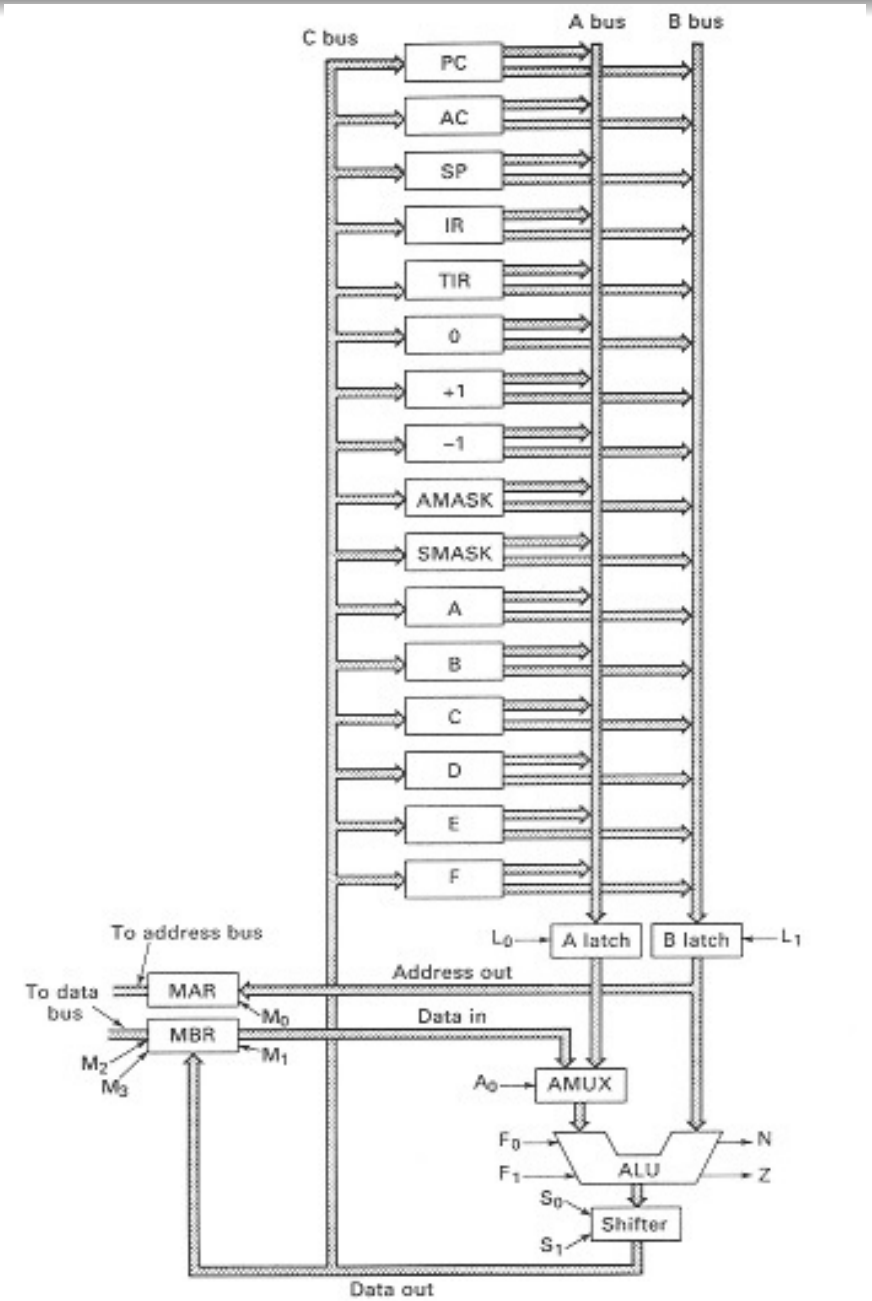
- Mmux seleciona MPC + 1 ou ADDR para colocar em MPC. Isso depende de COND e os sinais N e Z da ALU.

Dificuldade de escrever microinstruções



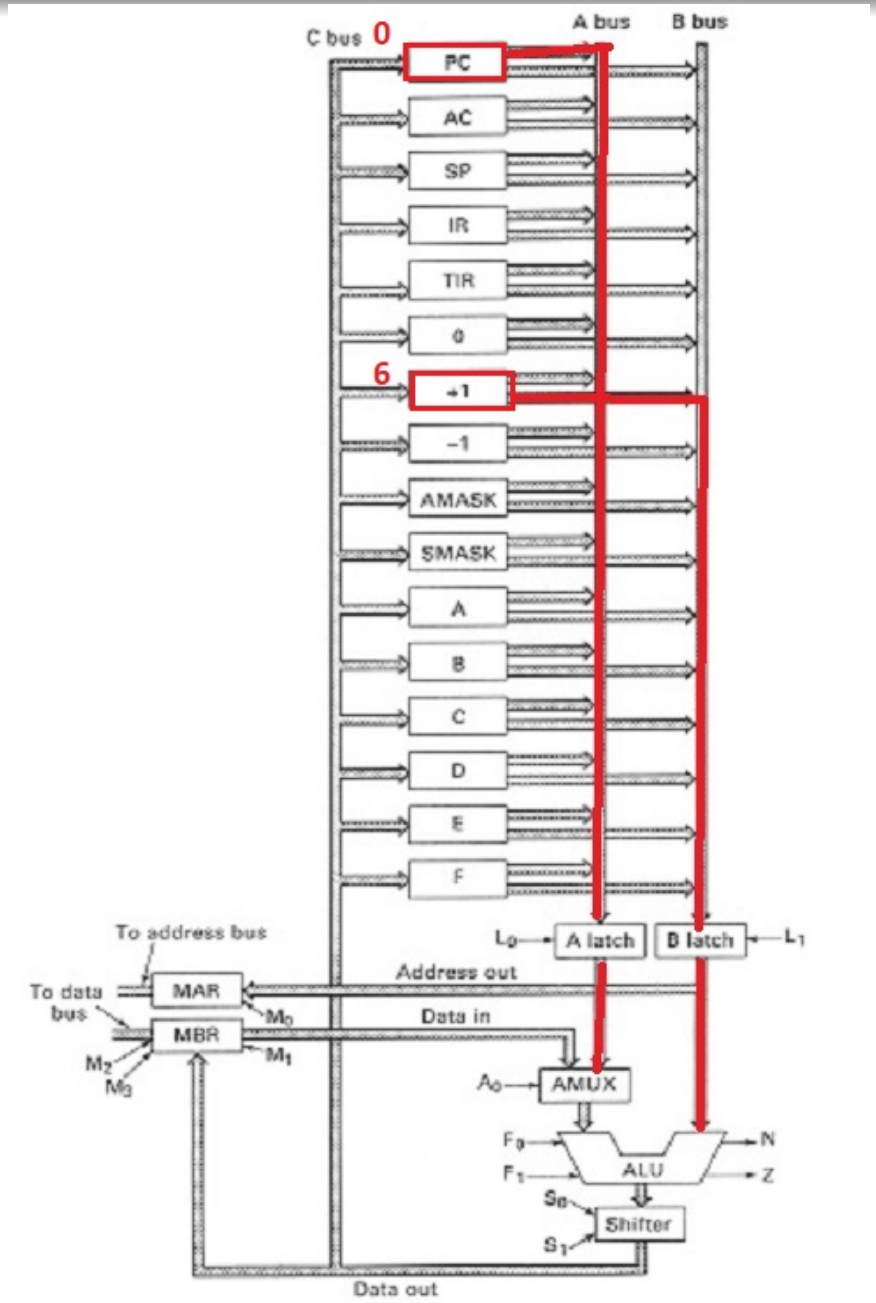
- Cada microinstrução consta de 32 bits (conforme já visto).
- Esses 32 bits determinam o que deve acontecer num ciclo (4 subciclos).
- Escrever cada microinstrução é uma tarefa árdua (pois lida com o nível muito baixo - zeros e uns).

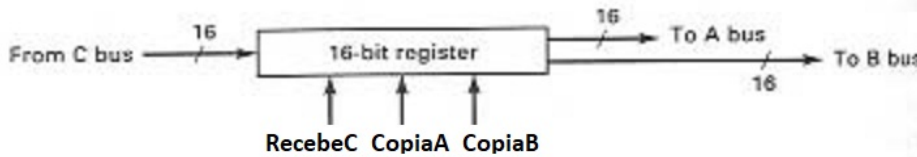
Dificuldade de escrever microinstruções



- Vejamos um exemplo para mostrar a dificuldade de escrever microinstruções.
- Exemplo: Queremos somar 1 com o valor de PC e colocando o resultado da soma de volta em PC.
- Vejamos como é a microinstrução necessária para este exemplo.

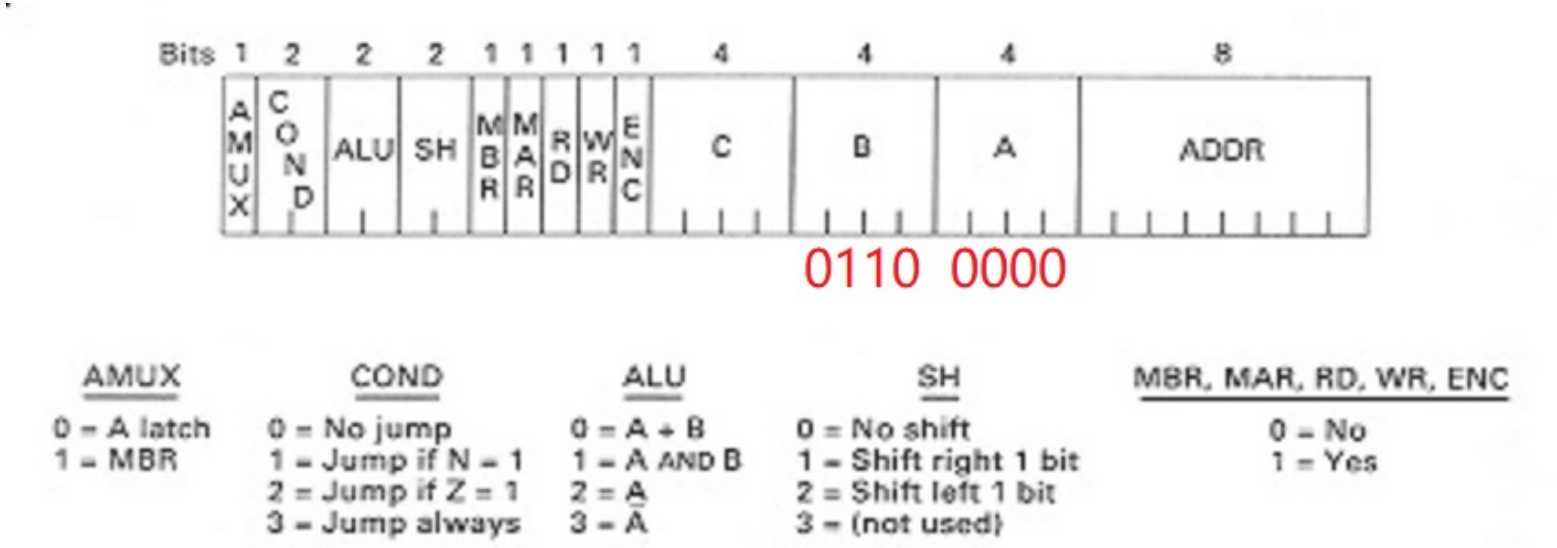
Sinais de controle - selecionam quais registradores





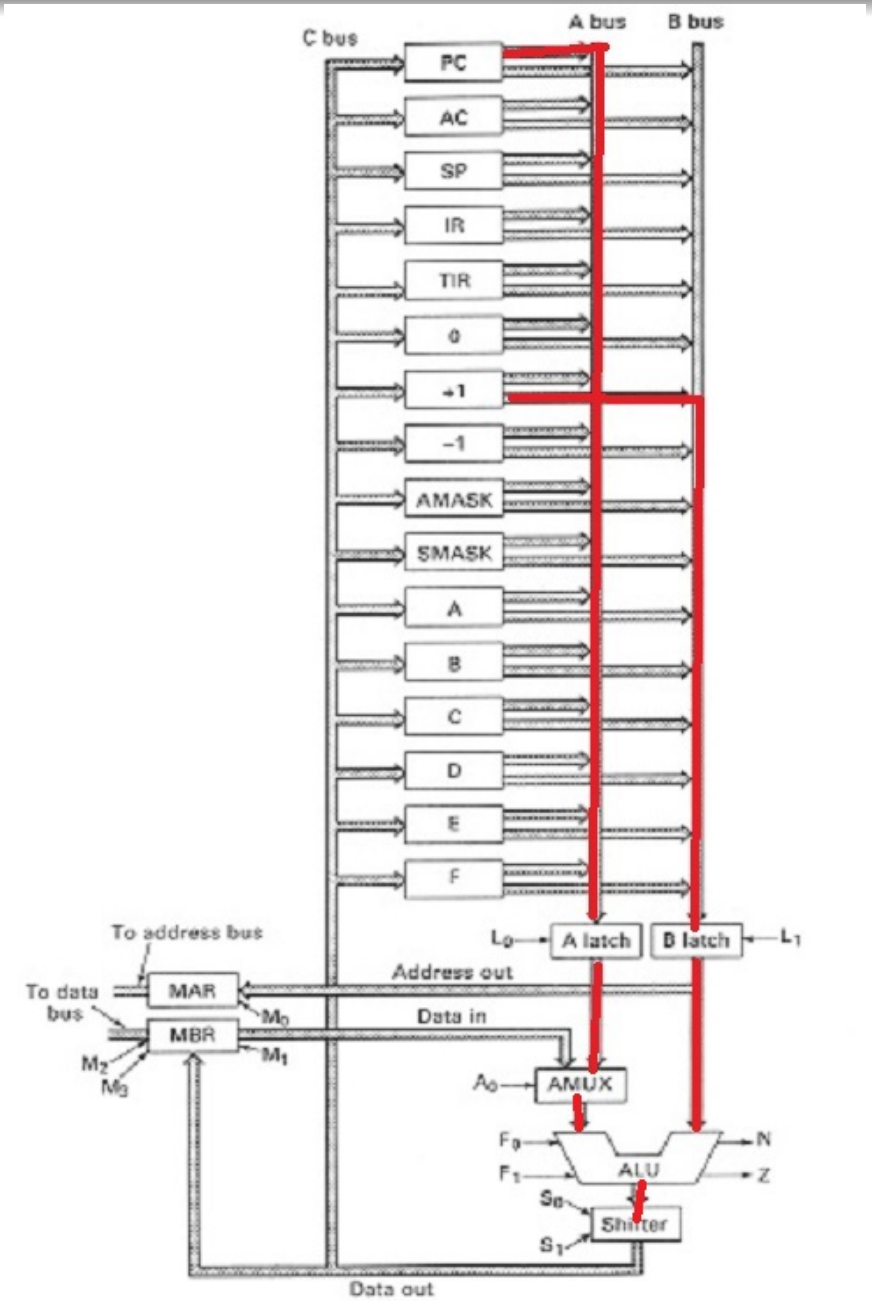
- Queremos *CopiaA* = 1 do registrador 0 (PC) para copiar o seu valor no barramento A. Para isso o decodificador A tem como entrada 0000.
- Queremos *CopiaB* = 1 do registrador 6 (contendo 1) para copiar 1 no barramento B. Para isso o decodificador B tem como entrada 0110.

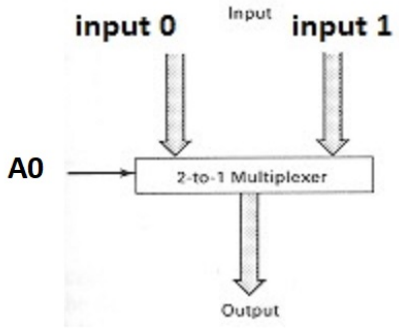
Sinais de controle - selecionam quais registradores

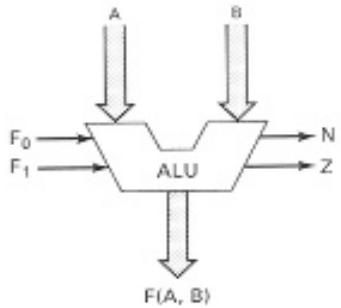


- Fazemos $A = 0000$: o registrador 0 (PC) copia o seu valor no barramento A
- Fazemos $B = 0110$: o registrador 6 (contendo 1) copia o seu valor no barramento B.

Sinais de controle - MUX e ALU

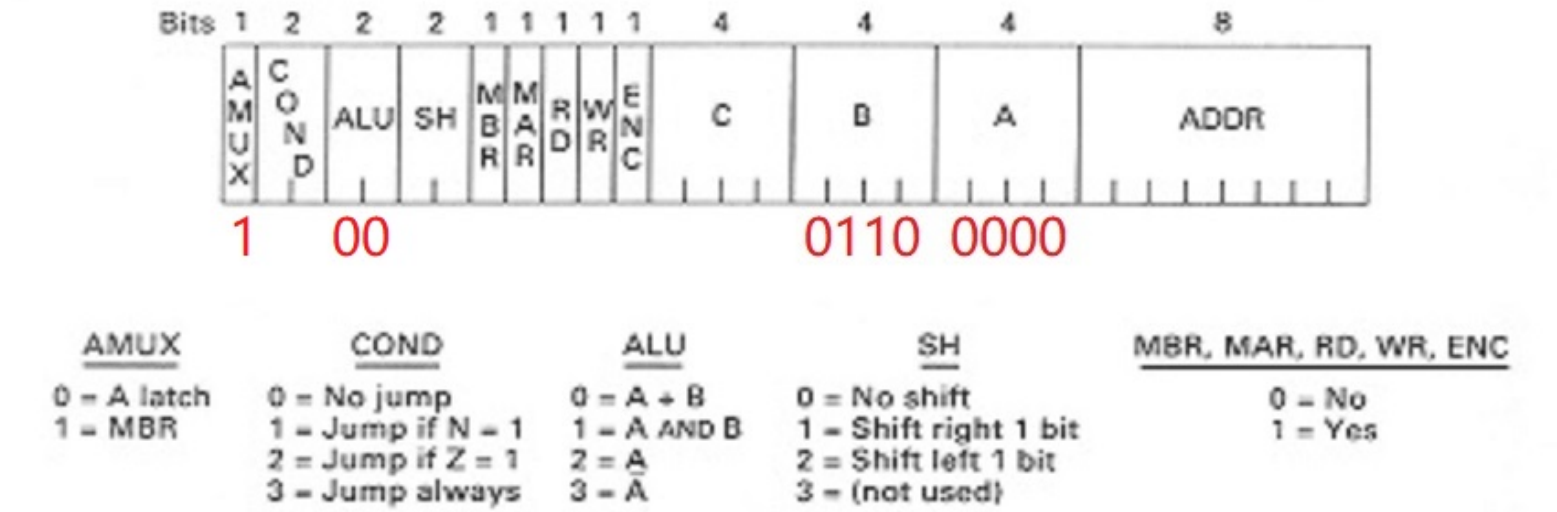






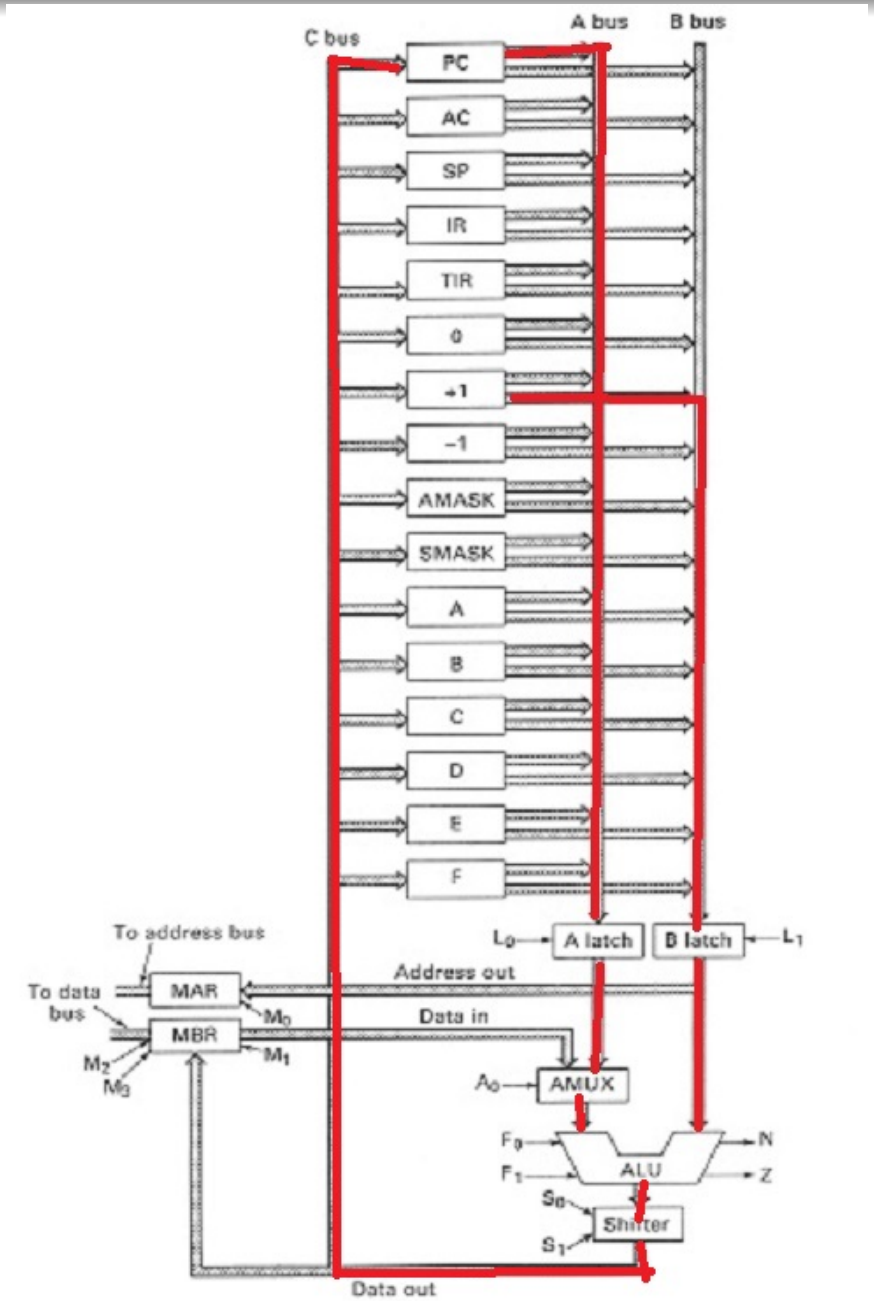
- Fazemos $A0$ ou $AMUX = 1$ do MUX para selecionar a entrada da direita.
- Fazemos $F_0F_1 = 00$ da ALU para fazer soma.

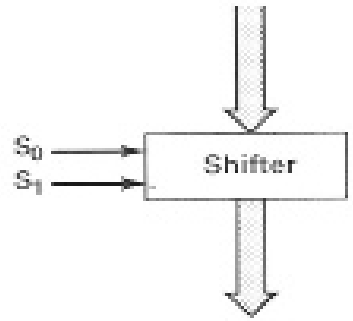
Sinais de controle - MUX e ALU

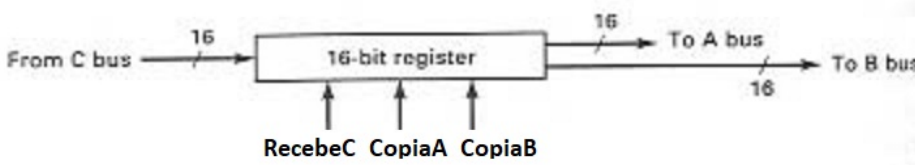


- $AMUX = 1$: o MUX seleciona a entrada da direita
- $F_0 F_1 = 00$: a ALU faz a soma

Sinais de controle - Shifter e Recebe C em PC

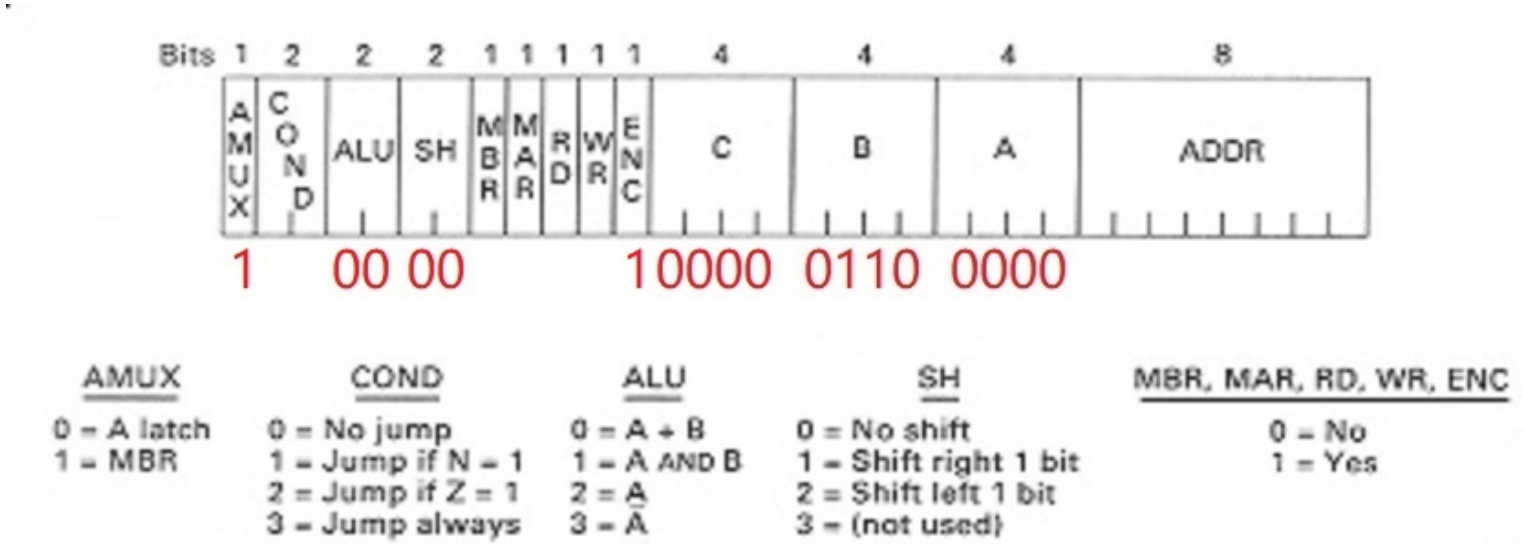






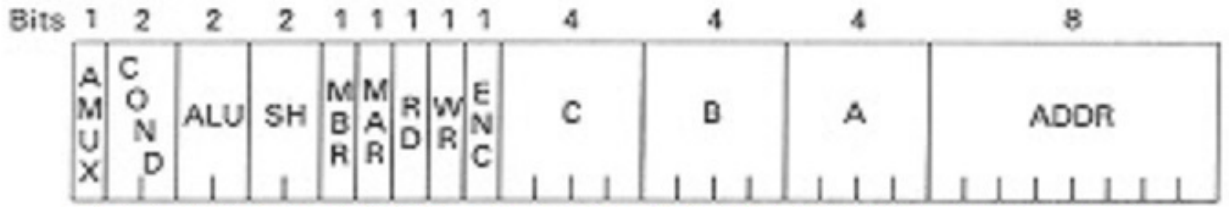
- Fazemos $S_0 S_1 = 00$ do Shifter para não deslocar nada
- Fazemos decodificador $C = 0000$ para o registrador 0 (PC) receber o valor do barramento C.
- Fazemos $EnableC = 1$ para habilitar um registrador a receber o valor do barramento C.

Sinais de controle - Shifter e Recebe C em PC



- $S_0 S_1 = 00$: o Shifter não desloca nada
- $C = 0000$: o registrador 0 (PC) recebe o valor do barramento C.
- $EnableC = 1$: habilita registrador para receber um valor do barramento C.

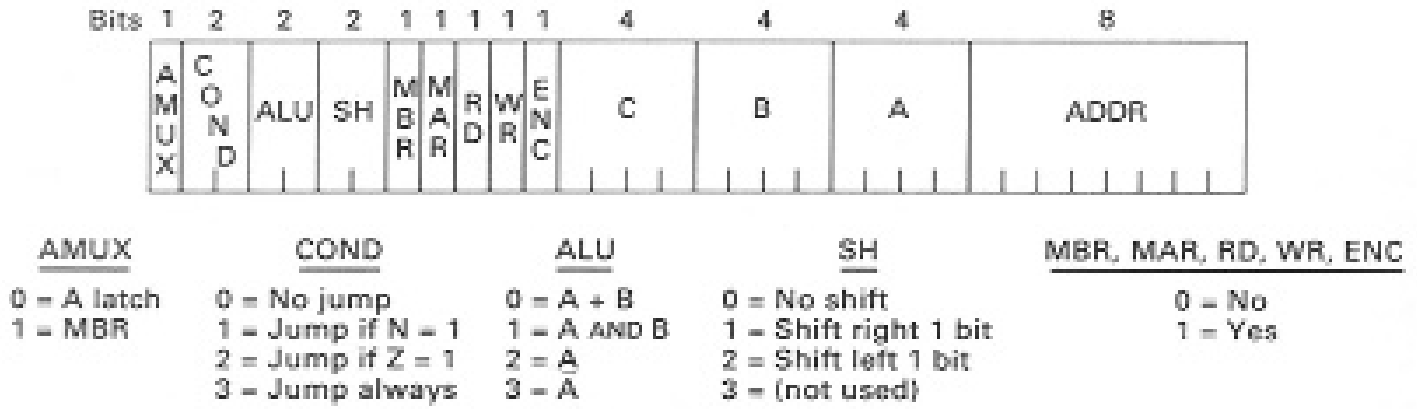
Dificuldade de escrever microinstruções



AMUX	COND	ALU	SH	MBR, MAR, RD, WR, ENC
0 = A latch 1 = MBR	0 = No jump 1 = Jump if N = 1 2 = Jump if Z = 1 3 = Jump always	0 = A + B 1 = A AND B 2 = A 3 = A	0 = No shift 1 = Shift right 1 bit 2 = Shift left 1 bit 3 = (not used)	0 = No 1 = Yes

- Precisamos ainda preencher todos os demais bits da microinstrução.
- Acima a microinstrução completa para fazer somar 1 no PC. isso.

Micro-assembler



- Micro-assembler (ou micro-montador) facilita a escrita de microinstruções por permitir mnemônicos e símbolos parecidos com um programa de alto nível.
- Na verdade o micro-assembler ainda é de baixo nível, no sentido de cada microinstrução em micro-assembler deve corresponder a uma microinstrução de 32 bits.

Exemplo de microinstruções em micro-assembler

Voltemos ao exemplo anterior: somar 1 ao registrador PC.

- Uma microinstrução em micro-assembler pode ser assim:
  pc := pc + 1
- Ela corresponde a uma microinstrução de 32 bits, onde
  A = 0 (Registrador 0 é PC)
  B = 6 (Registrador 6 contém o número 1)
  C = 0 (Registrador 0 é PC)
  ALU = 0 (0 corresponde à operação soma na ALU)
  Shifter = 0 (0 significa não desloca nada)
  ENC = 1 (indica que o resultado da ALU deve voltar a um registrador)
- Fica mais fácil escrever pc := pc + 1 do que 10000000000100000110000000000000.
- O micro-assembler transforma pc := pc + 1 em 10000000000100000110000000000000.

Exemplo de microinstruções em micro-assembler

Voltemos ao exemplo anterior: somar 1 ao registrador PC.

- Uma microinstrução em micro-assembler pode ser assim:
  pc := pc + 1
- Ela corresponde a uma microinstrução de 32 bits, onde
  A = 0 (Registrador 0 é PC)
  B = 6 (Registrador 6 contém o número 1)
  C = 0 (Registrador 0 é PC)
  ALU = 0 (0 corresponde à operação soma na ALU)
  Shifter = 0 (0 significa não desloca nada)
  ENC = 1 (indica que o resultado da ALU deve voltar a um registrador)
- Fica mais fácil escrever pc := pc + 1 do que 10000000000100000110000000000000.
- O micro-assembler transforma pc := pc + 1 em 10000000000100000110000000000000.

As 4 operações da ALU

Para especificar os 2 bits ALU que especifica uma das 4 operações da ALU, usamos +, band ou inv. Exemplos:

- pc: := pc + 1
- ac := band (ac, tir)
- tir := inv(tir)

ALU só sabe realizar 4 operações. Não se pode escrever operações que a ALU não sabe fazer: por exemplo

- pc: := pc / ac (Não pode)
- ac := sp * sp (Não pode)

Operações de deslocamento (shifter)

Para especificar os bits SH que controlam o “shifter”, escrevemos
rshift (desloca um bit para direita) e
lshift (desloca um bit para esquerda).

- ac := rshift(ir)
- tir:=lshift(tir+tir)

Esta última microinstrução coloca tir nos barramentos A e B, realiza a adição, desloca o resultado de 1 bit para esquerda, e finalmente guarda o resultado de volta a tir.

Pergunta: qual o efeito dessa microinstrução? o valor de tir é multiplicado por que valor?

Operações de deslocamento (shifter)

Para especificar os bits SH que controlam o “shifter”, escrevemos
rshift (desloca um bit para direita) e
lshift (desloca um bit para esquerda).

- ac := rshift(ir)
- tir:=lshift(tir+tir)

Esta última microinstrução coloca tir nos barramentos A e B, realiza a adição, desloca o resultado de 1 bit para esquerda, e finalmente guarda o resultado de volta a tir.

Pergunta: qual o efeito dessa microinstrução? o valor de tir é multiplicado por que valor?

Operações de deslocamento (shifter)

Para especificar os bits SH que controlam o “shifter”, escrevemos
rshift (desloca um bit para direita) e
lshift (desloca um bit para esquerda).

- ac := rshift(ir)
- tir:=lshift(tir+tir)

Esta última microinstrução coloca tir nos barramentos A e B, realiza a adição, desloca o resultado de 1 bit para esquerda, e finalmente guarda o resultado de volta a tir.

Pergunta: qual o efeito dessa microinstrução? o valor de tir é multiplicado por que valor?

Desvio incondicional e condicional if

Desvios utilizam os campos COND e ADDR. Lembrem-se de que o desvio é para uma microinstrução dentro do microprograma.

Desvio incondicional usa o comando goto. Exemplo:

- goto 12

  O micro-assembler gera uma microinstrução onde

  COND = 3 (desvia sempre) e ADDR = 12.

Desvios condicionais podem testar N ou Z (saída da ALU negativa ou zero). Exemplo:

- ac:= ac + 1; if z then goto 45

Note que tudo isso acima pode ser gerado por uma microinstrução de 32 bits.
Isso é essencial, pois cada microinstruçao em micro-assembler deve corresponder a apenas uma microinstrução.
Assim sendo, nada de instrução para micro-assembler do tipo
ac := 7 * ac + sqrt (pc) - cos (tir); if ac < arcsin(sp) then goto 45

Desvio incondicional e condicional if

Desvios utilizam os campos COND e ADDR. Lembrem-se de que o desvio é para uma microinstrução dentro do microprograma.

Desvio incondicional usa o comando goto. Exemplo:

- goto 12

  O micro-assembler gera uma microinstrução onde

  COND = 3 (desvia sempre) e ADDR = 12.

Desvios condicionais podem testar N ou Z (saída da ALU negativa ou zero). Exemplo:

- ac:= ac + 1; if z then goto 45

Note que tudo isso acima pode ser gerado por uma microinstrução de 32 bits.

Isso é essencial, pois cada microinstruçao em micro-assembler deve corresponder a apenas uma microinstrução.
Assim sendo, nada de instrução para micro-assembler do tipo
ac := 7 * ac + sqrt (pc) - cos (tir); if ac < arcsin(sp) then goto 45

Desvio incondicional e condicional if

Desvios utilizam os campos COND e ADDR. Lembrem-se de que o desvio é para uma microinstrução dentro do microprograma.

Desvio incondicional usa o comando goto. Exemplo:

- goto 12

  O micro-assembler gera uma microinstrução onde

  COND = 3 (desvia sempre) e ADDR = 12.

Desvios condicionais podem testar N ou Z (saída da ALU negativa ou zero). Exemplo:

- ac:= ac + 1; if z then goto 45

Note que tudo isso acima pode ser gerado por uma microinstrução de 32 bits.
Isso é essencial, pois cada microinstruçao em micro-assembler deve corresponder a apenas uma microinstrução.
Assim sendo, nada de instrução para micro-assembler do tipo
ac := 7 * ac + sqrt (pc) - cos (tir); if ac < arcsin(sp) then goto 45

Desvio incondicional e condicional if

Desvios utilizam os campos COND e ADDR. Lembrem-se de que o desvio é para uma microinstrução dentro do microprograma.

Desvio incondicional usa o comando goto. Exemplo:

- goto 12

  O micro-assembler gera uma microinstrução onde

  COND = 3 (desvia sempre) e ADDR = 12.

Desvios condicionais podem testar N ou Z (saída da ALU negativa ou zero). Exemplo:

- ac:= ac + 1; if z then goto 45

Note que tudo isso acima pode ser gerado por uma microinstrução de 32 bits.
Isso é essencial, pois cada microinstruçao em micro-assembler deve corresponder a apenas uma microinstrução.
Assim sendo, nada de instrução para micro-assembler do tipo
ac := 7 * ac + sqrt (pc) - cos (tir); if ac < arcsin(sp) then goto 45

Desvio condicional

Quero escrever uma microinstrução que faz o seguinte.

Se ac for zero, então desvia para 47.

Como fazemos isso?

Para podermos usar if z then goto 47, o valor de ac deve aparecer na saída da ALU. Para isso, podemos escrever:

- alu:=ac; if z then goto 47

Isso faz o ac passar pela ALU, apenas para podermos usar o teste if z.

Outra solução:

- ac:=ac+0; if z then goto 47

Desvio condicional

Quero escrever uma microinstrução que faz o seguinte.

Se ac for zero, então desvia para 47.

Como fazemos isso?

Para podermos usar if z then goto 47, o valor de ac deve aparecer na saída da ALU. Para isso, podemos escrever:

- alu:=ac; if z then goto 47

Isso faz o ac passar pela ALU, apenas para podermos usar o teste if z.

Outra solução:

- ac:=ac+0; if z then goto 47

Desvio condicional

Quero escrever uma microinstrução que faz o seguinte.

Se ac for zero, então desvia para 47.

Como fazemos isso?

Para podermos usar if z then goto 47, o valor de ac deve aparecer na saída da ALU. Para isso, podemos escrever:

- alu:=ac; if z then goto 47

Isso faz o ac passar pela ALU, apenas para podermos usar o teste if z.

Outra solução:

- ac:=ac+0; if z then goto 47

Desvio condicional

Quero escrever uma microinstrução que faz o seguinte.

Se ac for zero, então desvia para 47.

Como fazemos isso?

Para podermos usar if z then goto 47, o valor de ac deve aparecer na saída da ALU. Para isso, podemos escrever:

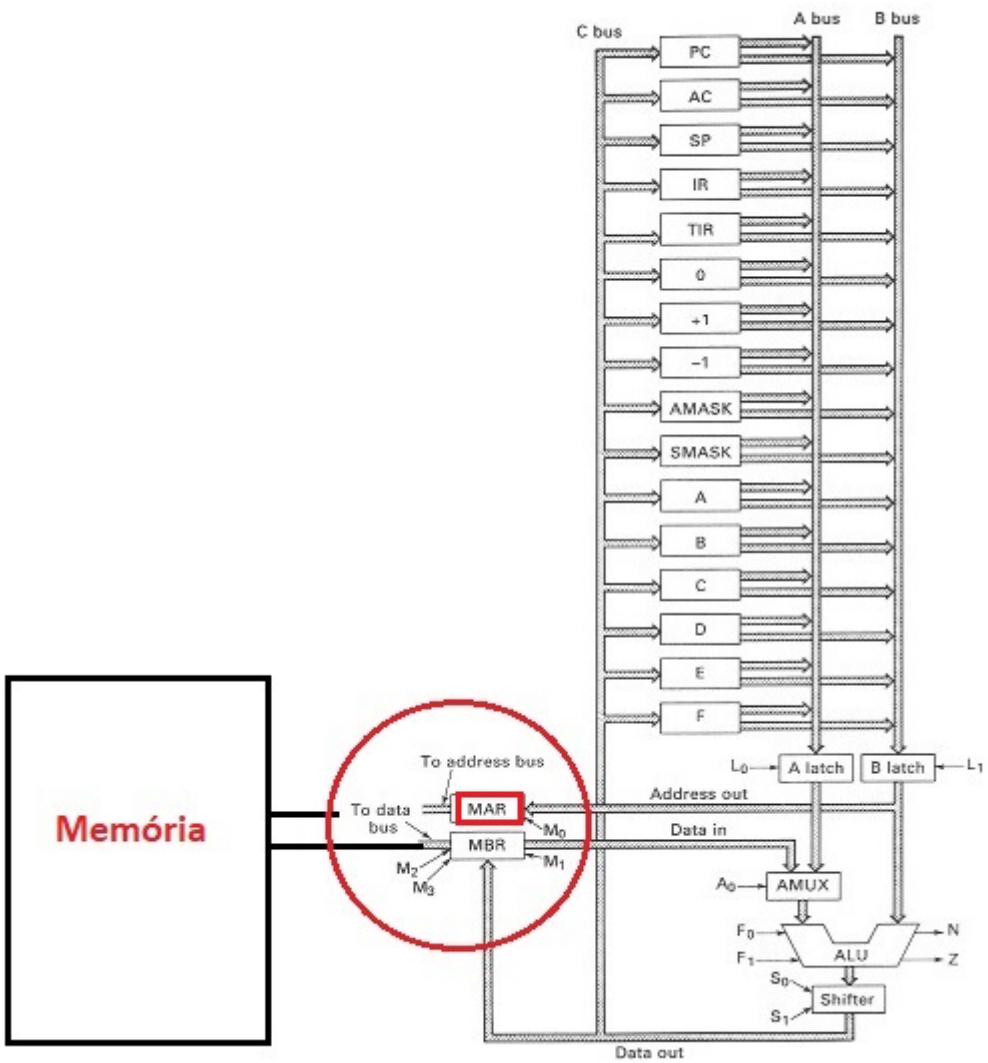
- alu:=ac; if z then goto 47

Isso faz o ac passar pela ALU, apenas para podermos usar o teste if z.

Outra solução:

- ac:=ac+0; if z then goto 47

Leitura da memória



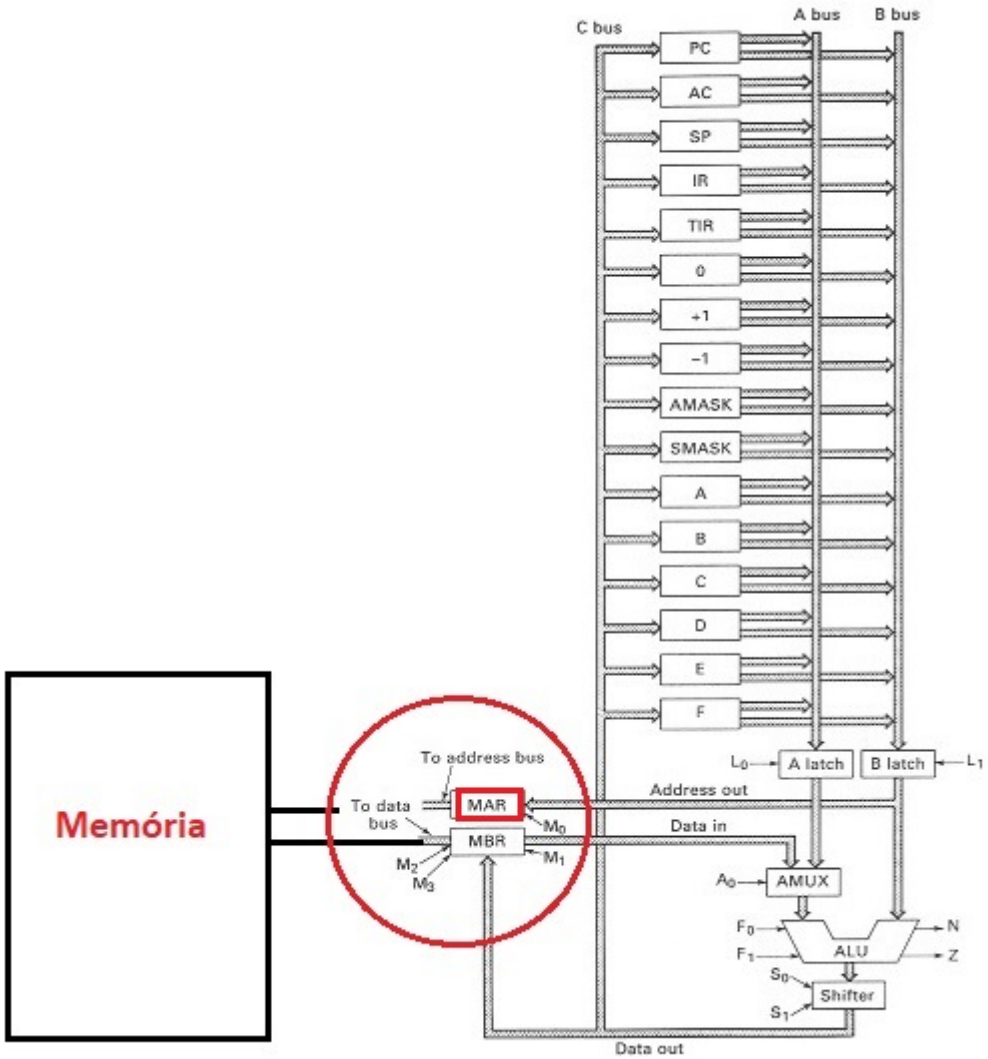
Leitura da memória leva dois ciclos, em que o bit rd deve estar ligado para a memória saber que é leitura.

No primeiro ciclo mar deve receber o endereço a ser lido e o bit rd ligado. No segundo ciclo, o bit rd deve continuar ligado. Exemplo:

- ciclo 1: mar:=sp; rd
- ciclo 2: rd

O dado lido fica disponível no mbr no terceiro ciclo.

Leitura da memória



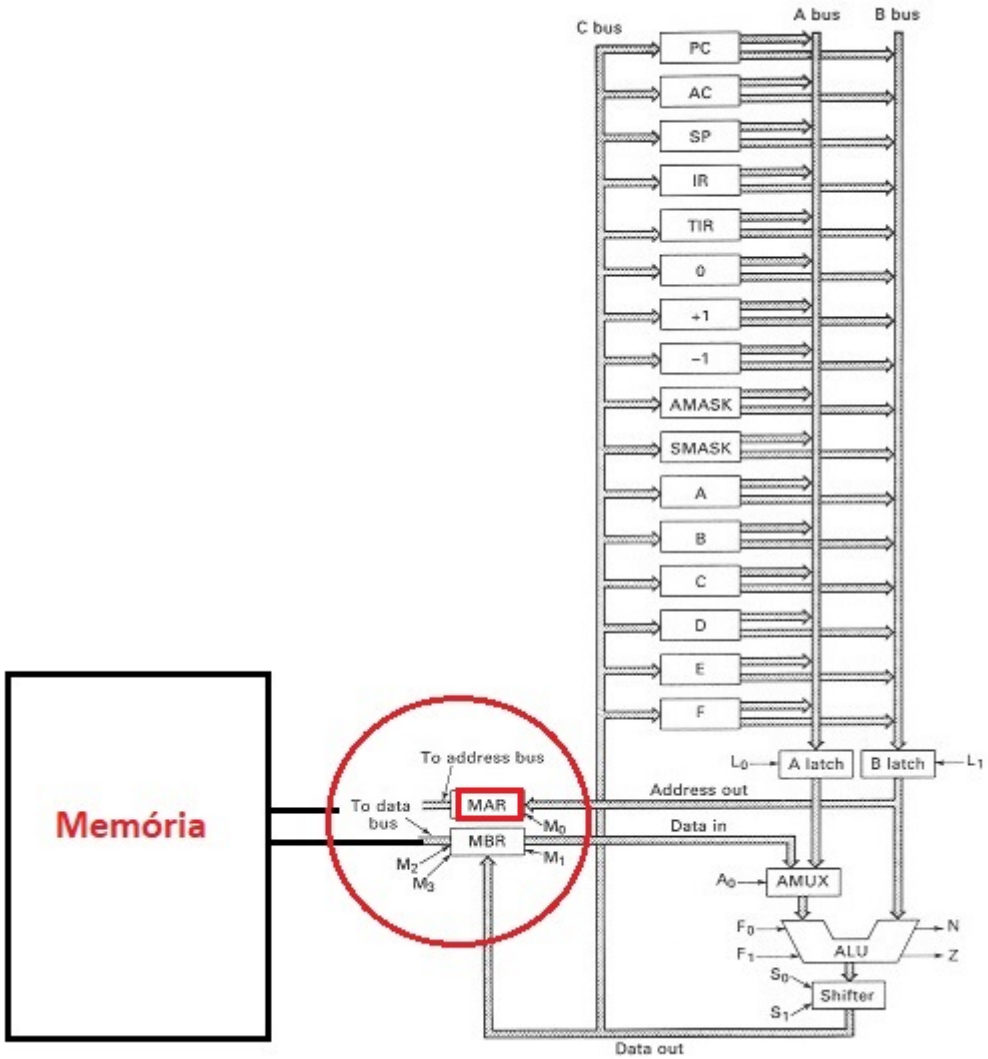
- ciclo 1: mar:=sp; rd
- ciclo 2: rd

Note o desperdício no ciclo 2: uma microinstrução tem 32 bits, dos quais apenas um bit rd está usado.

Então um bom microprogramador tentaria aproveitar melhor essa microinstrução, procurando incluir algo que pode ser feito no mesmo ciclo e assim usa melhor a microinstrução. Por exemplo:

- ciclo 1: mar:=sp; rd
- ciclo 2: ac:=ac+1; rd

Leitura da memória



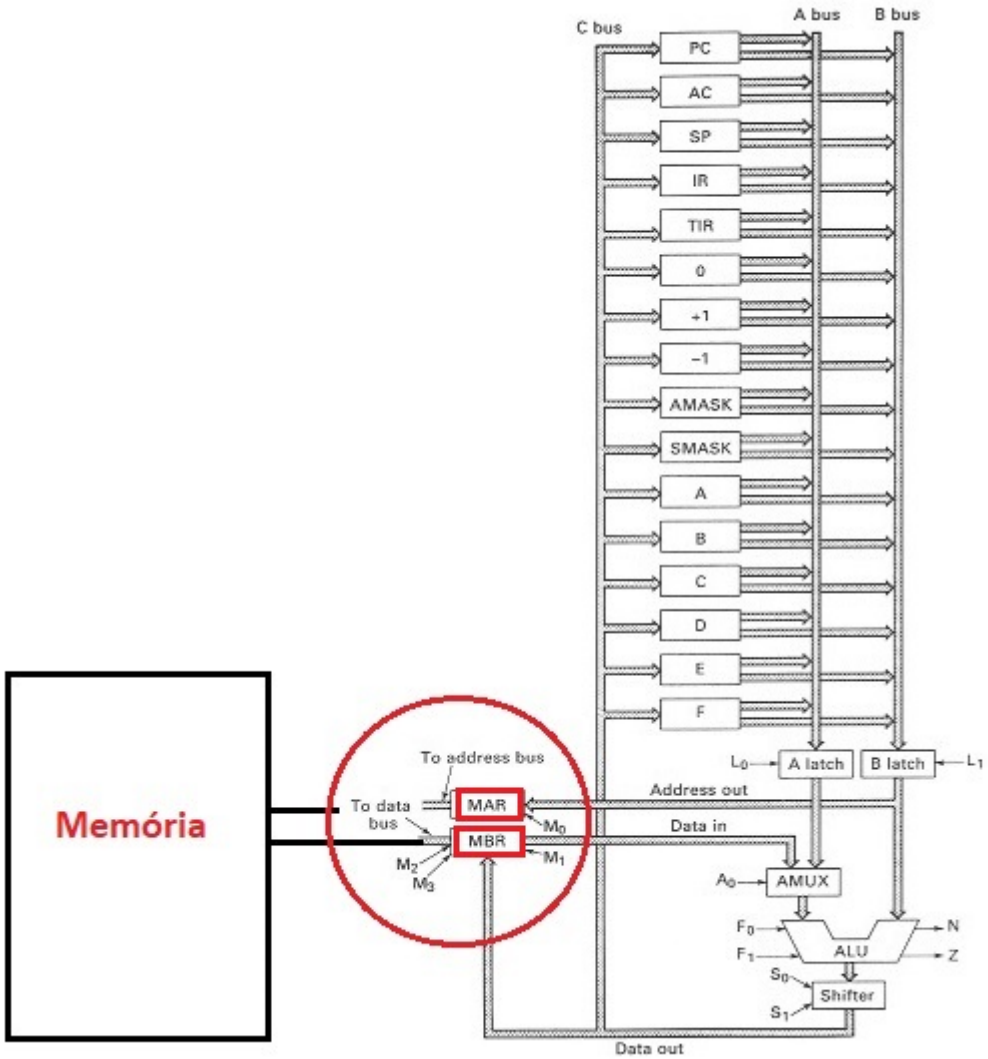
- ciclo 1: mar:=sp; rd
- ciclo 2: rd

Note o desperdício no ciclo 2: uma microinstrução tem 32 bits, dos quais apenas um bit rd está usado.

Então um bom microprogramador tentaria aproveitar melhor essa microinstrução, procurando incluir algo que pode ser feito no mesmo ciclo e assim usa melhor a microinstrução. Por exemplo:

- ciclo 1: mar:=sp; rd
- ciclo 2: ac:=ac+1; rd

Escrita na memória



Escrita na memória também leva dois ciclos, em que o bit wr deve estar ligado para a memória saber que é escrita.

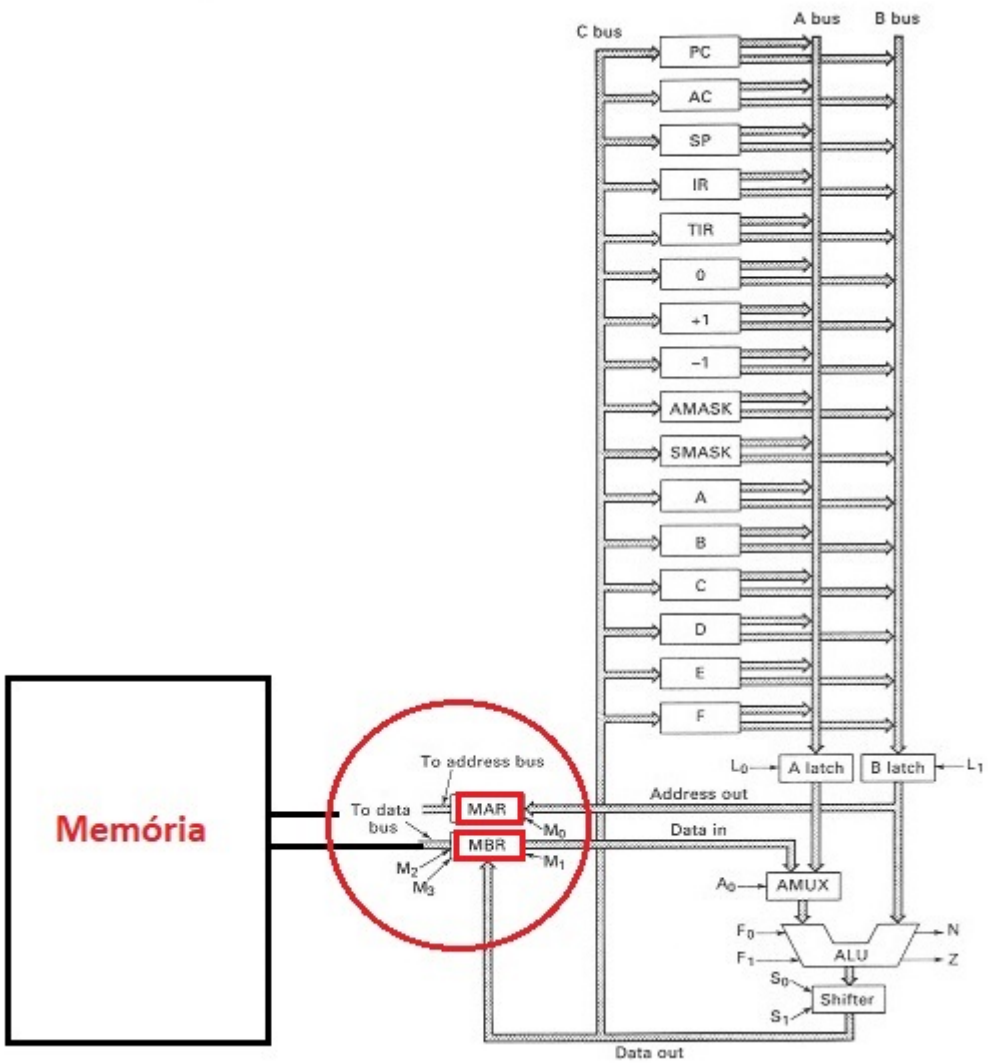
No primeiro ciclo mbr deve conter o dado a ser escrito, mar deve conter o endereço em que será escrito o dado e o bit wr deve estar ligado.

No segundo ciclo o bit wr deve continuar ligado.

Exemplo:

- ciclo 1: mar:=sp; mbr:=ac; wr
- ciclo 2: wr

Escrita na memória

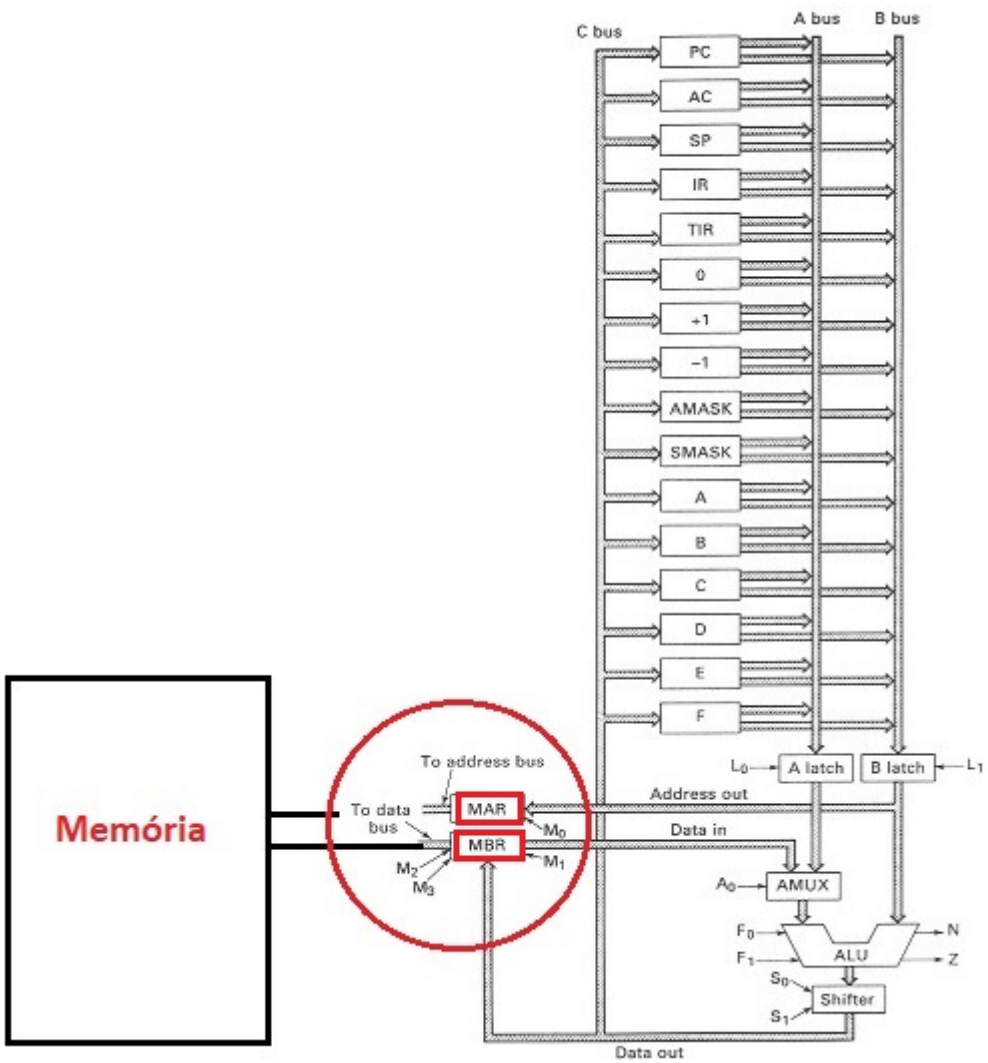


- ciclo 1: mar:=sp; mbr:=ac; wr
- ciclo 2: wr

Note novamente o desperdício no ciclo 2: dos 32 bits da microinstrução usamos apenas um bit wr. Podemos por exemplo incluir mais coisas no segundo ciclo.

- ciclo 1: mar:=sp; mbr:=ac; wr
- ciclo 2: ac:=ac+1; if z goto 7; wr

Escrita na memória



- ciclo 1: mar:=sp; mbr:=ac; wr
- ciclo 2: wr

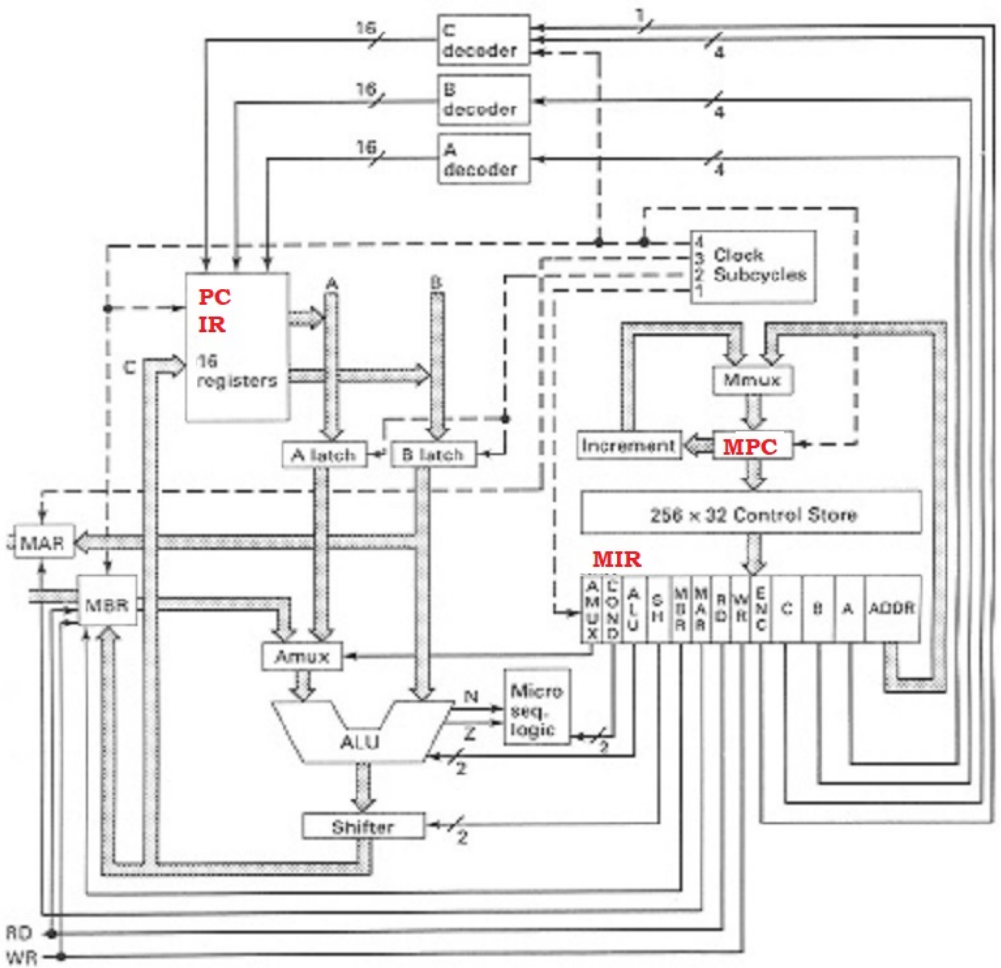
Note novamente o desperdício no ciclo 2: dos 32 bits da microinstrução usamos apenas um bit wr. Podemos por exemplo incluir mais coisas no segundo ciclo.

- ciclo 1: mar:=sp; mbr:=ac; wr
- ciclo 2: ac:=ac+1; if z goto 7; wr

Como escrever microprograma eficiente

- Uma microinstrução tem 32 bits que comandam a ação do processador durante um ciclo.
- É importante explorarmos, se possível, todos esses bits disponíveis numa mesma microinstrução, ao invés de dividir algo que poderia ser feito em um ciclo para serem feitos em dois ciclos ou mais.
- Leitura (rd) leva dois ciclos. Assim rd deve aparecer em duas microinstruções seguidas. O mesmo para escrita (wr). Então é importante aproveitar a microinstrução e não escrever uma microinstrução, se possível, apenas com rd ou apenas com wr.
- Veremos um exemplo a seguir de como escrever um microprograma eficiente.

PC e MPC, IR e MIR, como fazer desvios



Atenção para o seguinte:

- Temos a memória que é fora do processador e a memória ROM com o microprograma no processador.
- O PC aponta para a próxima instrução de máquina dentro da memória a executar e o IR contém a instrução de máquina em execução.
- O MPC aponta para a próxima microinstrução dentro do microprograma a executar e o MIR contém a microinstrução em execução.
- Desvio na memória é feito mudando o valor de PC. Desvio no microprograma é feito com COND e ADDR, exemplo: if z goto 20. .

Exercício em classe

- Suponha a criação de uma nova instrução de máquina chamada NOVA que faz o seguinte:

  Escreve o valor 0 na memória endereçada por SP
  Faz AC ficar igual a 4 vezes o valor de SP
  Soma TIR em AC e se o valor da soma ficar negativa
  então faz AC igual a 0, senão faz AC igual a 1
  No final o controle deve voltar a posição 0 do micropro-
  grama.

- Suponha que a instrução NOVA já está lida e encontra-se no IR. Suponha ainda que já foi feita a decodificação e sabe-se que se trata da instrução NOVA.
- Vamos escrever, em micro-assembler, o trecho das microinstruções que implementa NOVA. Suponha que o início desse trecho é na linha 101.

Solução: microprograma para NOVA

Escreve o valor 0 na memória endereçada por SP
Faz AC ficar igual a 4 vezes o valor de SP
Soma TIR em AC e se o valor da soma ficar negativa
então faz AC igual a 0, senão faz AC igual a 1
No final o controle deve voltar a posição 0 do microprograma.

Ineficiente:

```
101: mar:=sp;
102: mbr:=0; wr
103: wr
104: ac:=(sp+sp)
105: ac:=ac+ac
106: ac:=ac+tir; if n goto 109
107: ac:=1
108: goto 0
109: ac:=0
110: goto 0
```

Eficiente:

```
101: mar:=sp; mbr:=0; wr
102: ac:=lshift(sp+sp); wr
103: ac:=ac+tir; if n goto 105
104: ac:=1; goto 0
105: ac:=0; goto 0
```

Solução: microprograma para NOVA

Escreve o valor 0 na memória endereçada por SP
Faz AC ficar igual a 4 vezes o valor de SP
Soma TIR em AC e se o valor da soma ficar negativa
então faz AC igual a 0, senão faz AC igual a 1
No final o controle deve voltar a posição 0 do microprograma.

Ineficiente:

```
101: mar:=sp;
102: mbr:=0; wr
103: wr
104: ac:=(sp+sp)
105: ac:=ac+ac
106: ac:=ac+tir; if n goto 109
107: ac:=1
108: goto 0
109: ac:=0
110: goto 0
```

Eficiente:

```
101: mar:=sp; mbr:=0; wr
102: ac:=lshift(sp+sp); wr
103: ac:=ac+tir; if n goto 105
104: ac:=1; goto 0
105: ac:=0; goto 0
```

Lista de Exercícios 5

- Fazer e entregar por email a Lista de Exercícios 5.
- Há prazo para entrega. Recomendo não demorar muito. Bom fazer logo com a matéria fresquinha na cabeça.

Outro exercício em classe

Exercício em classe.

- Considere a micro-arquitetura MIC vista em classe que implementa as instruções de máquina. Suponha uma nova instrução de máquina chamada NEW que faz o seguinte:

  Escreve o valor de AC na memória de endereço dado por SP. Multiplica TIR por 4 e coloca o resultado em TIR,
  Soma a TIR o valor de AC. Se TIR ficar zero então desvia para (isto é, faz PC igual a) a SP senão desvia para a posição 0 da memória. Retorne à posição 0 do microcódigo

- Suponha que a instrução NEW já está lida e encontra-se no IR. Suponha ainda que já foi feita a decodificação e sabe-se que se trata da instrução NEW. Escreva, em micro-assembler, o trecho das micro-instruções que correspondem à execução de NEW. Suponha que o início desse trecho é na linha 61. Use o menor número possível de micro-instruções, caso contrário pontos serão descontados.

Solução no próximo slide.

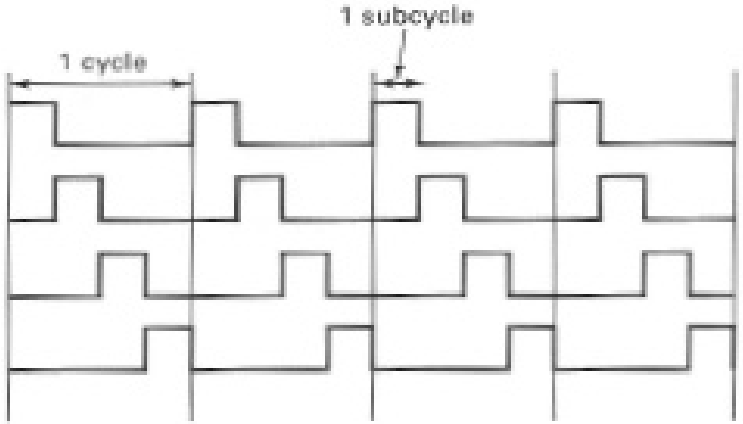
Exemplo de como escrever um microprograma eficiente

Escreve o valor de AC na memória de endereço dado por SP. Multiplica TIR por 4 e coloca o resultado em TIR,
Soma a TIR o valor de AC. Se TIR ficar zero então desvia para (isto é, faz PC igual a) a SP senão desvia para a posição 0 da memória. Retorne à posição 0 do microcódigo

Solução:

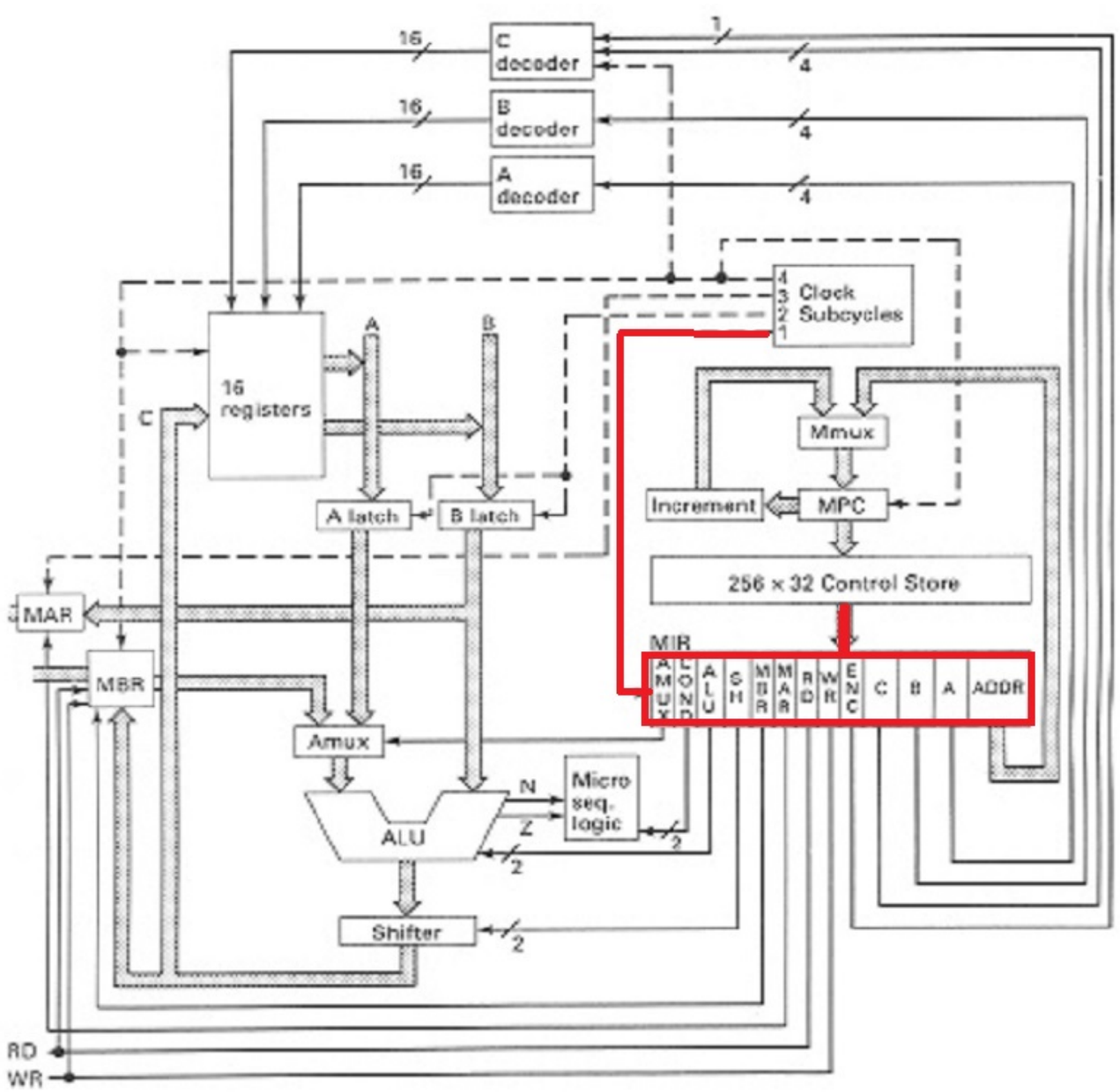
```
61   mar:=sp; mbr:=ac; wr
62   tir:=lshift(tir+tir); wr
63   tir:=tir+ac; if z goto 65
64   pc:=0; goto 0
65   pc:=sp; goto 0
```

4 subciclos na execução da microinstrução



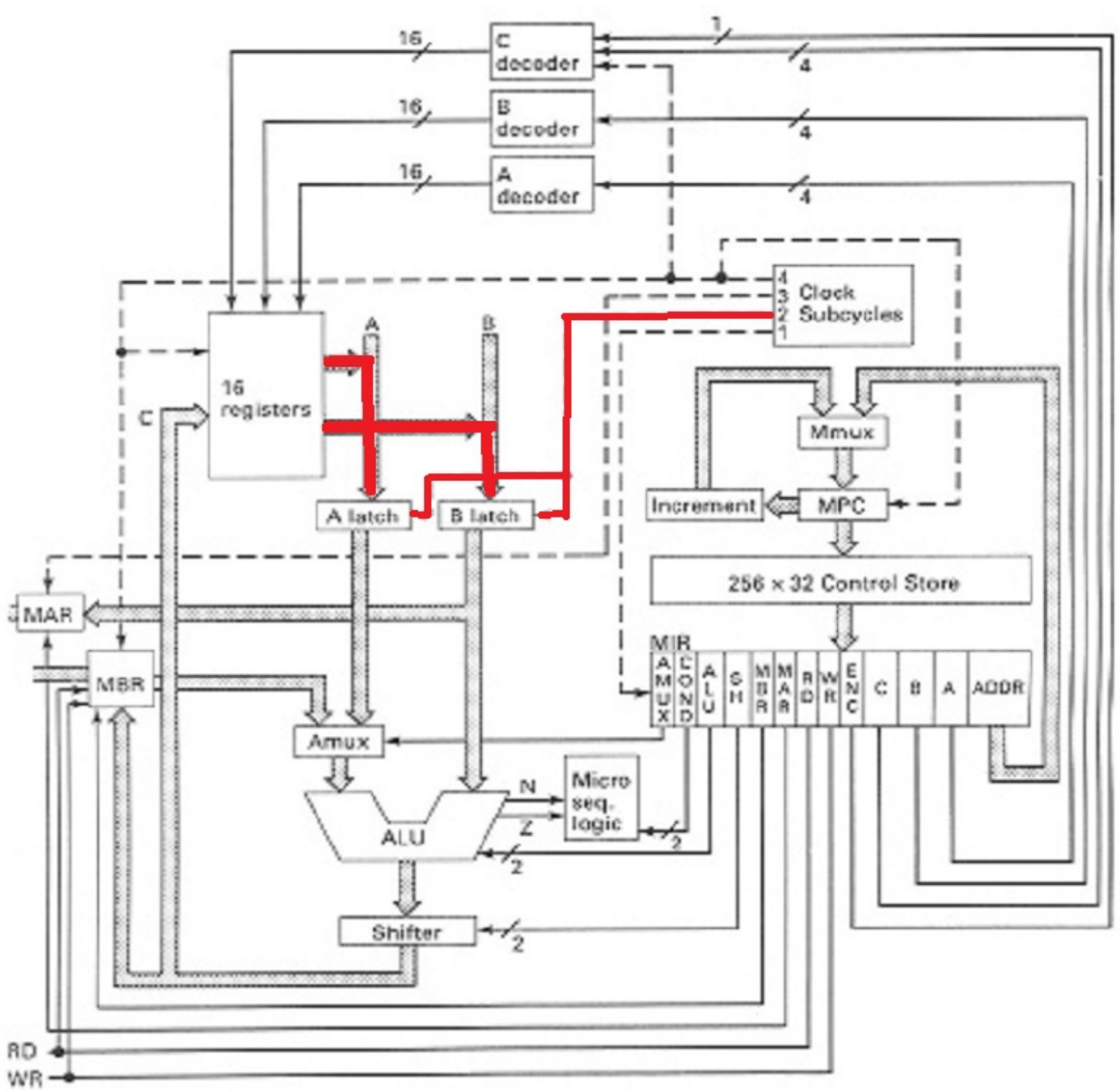
- O relógio de 4 fases fornece 4 subciclos.
- Subciclo 1: carrega a próxima microinstrução a ser executada num registrador chamado MIR (*micro instruction register*)
- Subciclo 2: coloca valores dos registradores nos barramentos A e B, carregando os *A latch* e *B latch*.
- Subciclo 3: dá o tempo necessário para a ALU e shifter produzirem seu resultado, carregando-o no MAR se for o caso.
- Subciclo 4: armazena o resultado no registrador ou no MBR. Prepara MPC para obter a próxima microinstrução a executar.

4 subciclos na execução da microinstrução



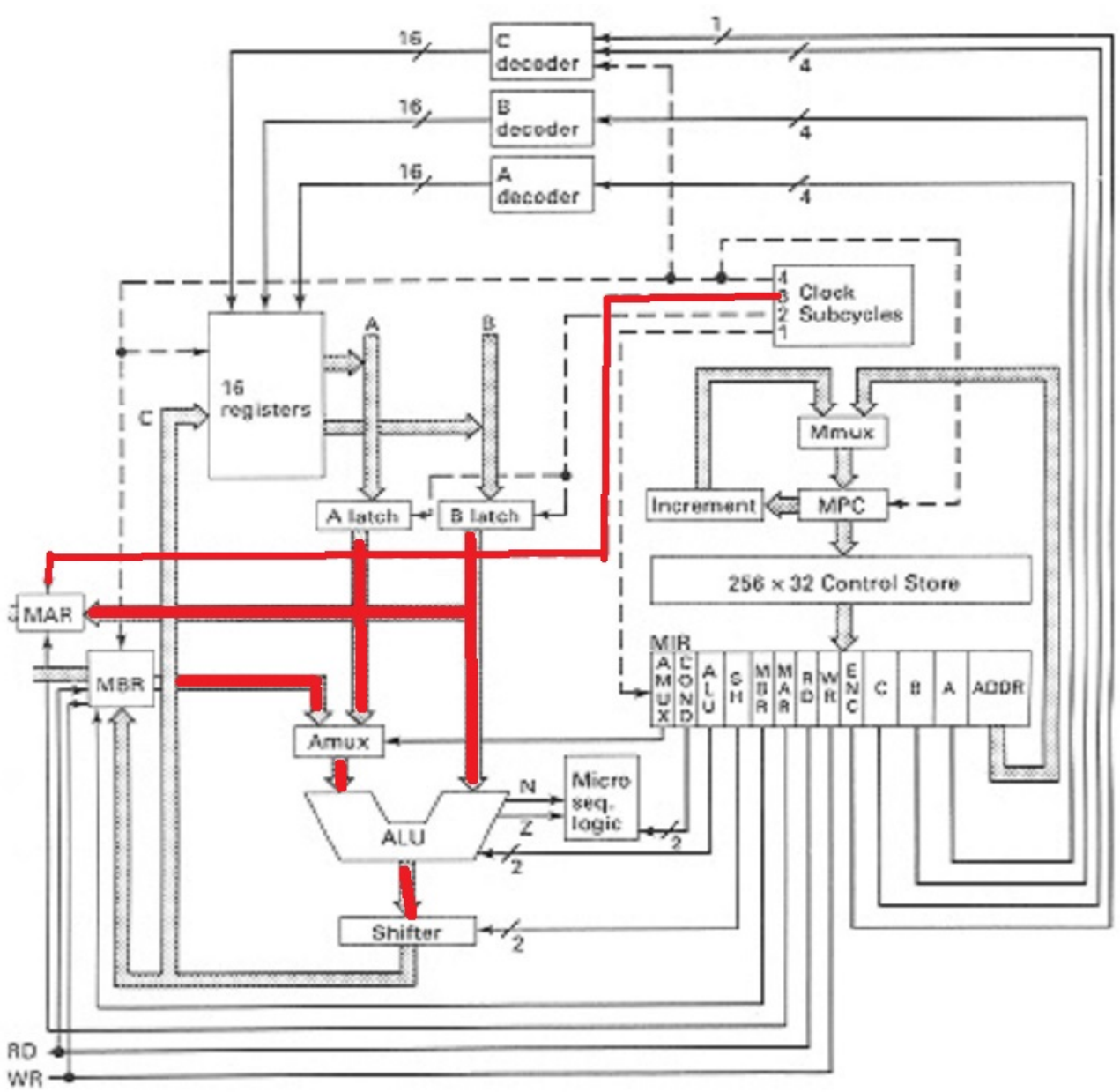
- Subciclo 1: carrega a próxima microinstrução a ser executada num registrador chamado MIR (*micro instruction register*)

4 subciclos na execução da microinstrução



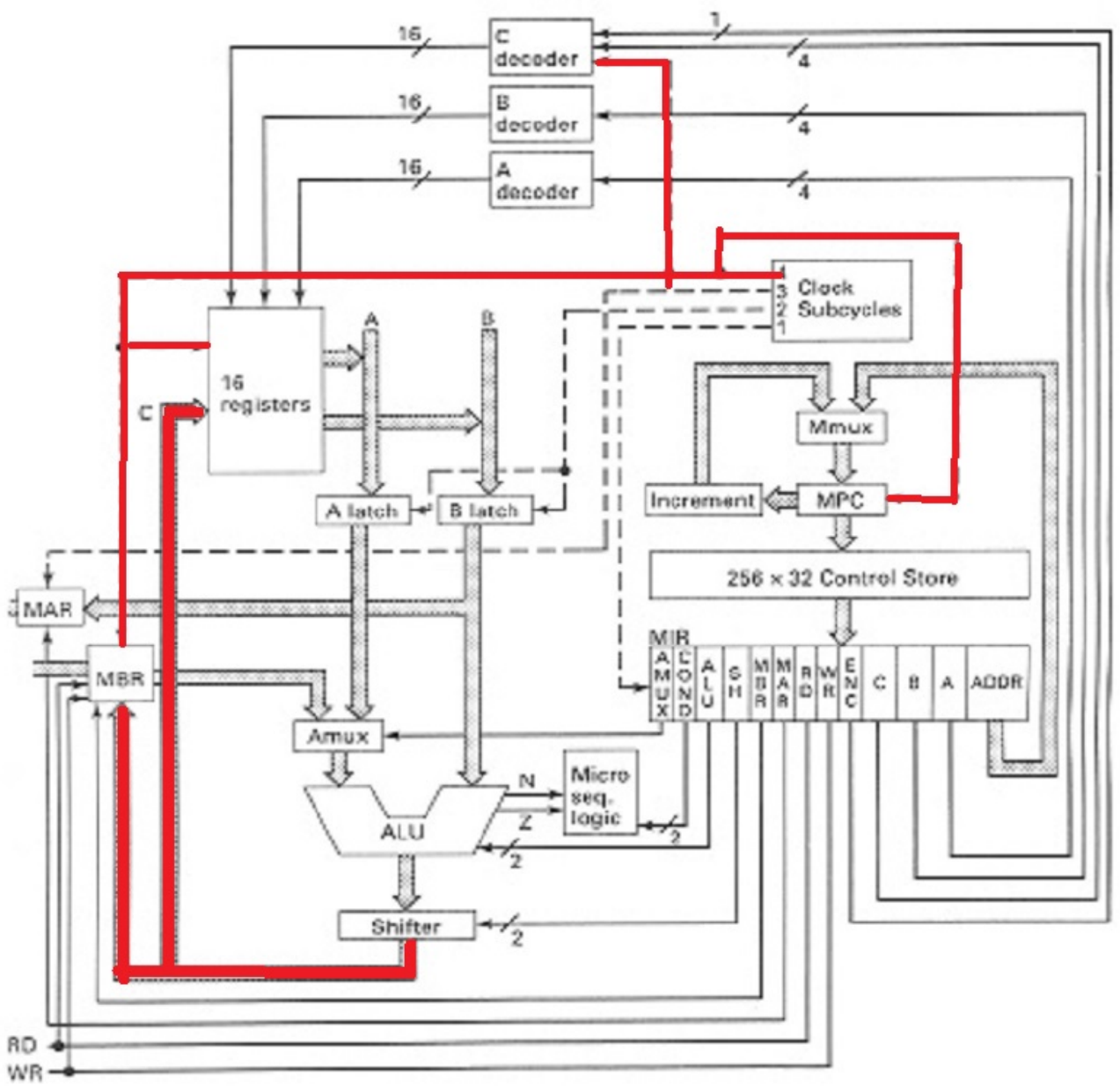
- Subciclo 2: coloca valores dos registradores nos barramentos A e B, carregando os *A latch* e *B latch*.

4 subciclos na execução da microinstrução



- Subciclo 3: dá o tempo necessário para a ALU e shifter produzirem seu resultado, carregando-o no MAR se for o caso.

4 subciclos na execução da microinstrução



- Subciclo 4: armazena o resultado no registrador ou no MBR. Prepara MPC para obter a próxima microinstrução a executar.

O microprograma completo na MIC

```
0: mar := pc ; rd ;                                  {main loop}
1: pc := pc + 1; rd ;                                {increment pc}
2: ir := mbr ; if n then goto 28;                    {save, decode mbr}
3: tir := lshift (ir + ir ); if n then goto 19;      {000x or 001x?}
4: tir := lshift (tir ); if n then goto 11;          {0000 or 0001?}
5: alu := tir ; if n then goto 9;                    {0000 = LODD}
6: mar := ir ; rd ;
7: rd ;
8: ac := mbr ; goto 0;
9: mar := ir ; mbr := ac ; wr ;                      {0001 = STOD}
10: wr ; goto 0;
11: alu := tir ; if n then goto 15;                  {0010 or 0011?}
12: mar := ir ; rd ;                                 {0010 = ADDD}
13: rd ;
14: ac := mbr + ac ; goto 0;
15: mar := ir ; rd ;                                 {0011 = SUBD}
16: ac := ac + 1; rd ;                               {Note: x − y = x + 1 + not y}
17: a := inv (mbr );
18: ac := ac + a ; goto 0;
19: tir := lshift (tir ); if n then goto 25;         {010x or 011x?}
20: alu := tir ; if n then goto 23;                  {0100 or 0101?}
21: alu := ac ; if n then goto 0;                    {0100 = JPOS}
22: pc := band (ir , amask ); goto 0;                {perform the jump}
23: alu := ac ; if z then goto 22;                   {0101 = JZER}
24: goto 0;                                          {jump failed}
25: alu := tir ; if n then goto 27;                  {0110 or 0111?}
26: pc := band (ir , amask ); goto 0;                {0110 = JUMP}
27: ac := band (ir , amask ); goto 0;                {0111 = LOCO}
28: tir := lshift (ir + ir ); if n then goto 40;     {10xx or 11xx?}
29: tir := lshift (tir ); if n then goto 35;         {100x or 101x?}
30: alu := tir ; if n then goto 33;                  {1000 or 1001?}
31: a := ir + sp ;                                   {1000 = LODL}
32: mar := a ; rd ; goto 7;
33: a := ir + sp ;                                   {1001 = STOL}
34: mar := a ; mbr := ac ; wr ; goto 10;
35: alu := tir ; if n then goto 38;                  {1010 or 1011?}
36: a := ir + sp ;                                   {1010 = ADDL}
37: mar := a ; rd ; goto 13;
38: a := ir + sp ;                                   {1011 = SUBL}
39: mar := a ; rd ; goto 16;
```

Ao lado o microprograma completo armazenado no Control Store.

O microprograma completo na MIC - continuação

```
40: tir := lshift(tir); if n then goto 46;          {110x or 111x?}
41: alu := tir; if n then goto 44;                  {1100 or 1101?}

42: alu := ac; if n then goto 22;                   {1100 = JNEG}
43: goto 0;

44: alu := ac; if z then goto 0;                    {1101 = JNZE}
45: pc := band(ir, amask); goto 0;

46: tir := lshift(tir); if n then goto 50;

47: sp := sp + (-1);                                {1110 = CALL}
48: mar := sp; mbr := pc; wr;
49: pc := band(ir, amask); wr; goto 0;

50: tir := lshift(tir); if n then goto 65;          {1111, examine addr}
51: tir := lshift(tir); if n then goto 59;
52: alu := tir; if n then goto 56;

53: mar := ac; rd;                                  {1111000 = PSHI}
54: sp := sp + (-1); rd;
55: mar := sp; wr; goto 10;

56: mar := sp; sp := sp + 1; rd;                    {1111001 = POPI}
57: rd;
58: mar := ac; wr; goto 10;

59: alu := tir; if n then goto 62;

60: sp := sp + (-1);                                {1111010 = PUSH}
61: mar := sp; mbr := ac; wr; goto 10;

62: mar := sp; sp := sp + 1; rd;                    {1111011 = POP}
63: rd;
64: ac := mbr; goto 0;

65: tir := lshift(tir); if n then goto 73;
66: alu := tir; if n then goto 70;

67: mar := sp; sp := sp + 1; rd;                    {1111100 = RETN}
68: rd;
69: pc := mbr; goto 0;

70: a := ac;                                        {1111101 = SWAP}
71: ac := sp;
72: sp := a; goto 0;

73: alu := tir; if n then goto 76;

74: a := band(ir, smask);                           {1111110 = INSP}
75: sp := sp + a; goto 0;

76: a := band(ir, smask);                           {1111111 = DESP}
77: a := inv(a);
78: a := a + 1; goto 75;
```

O microprograma completo tem apenas 79 microinstruções.

Conjunto de instruções da máquina MAC

0000xxxxxxxxxxx LODD ac:=m[x]
0001xxxxxxxxxxx STOD m[x]:=ac
0010xxxxxxxxxxx ADDD ac:=ac+m[x]
0011xxxxxxxxxxx SUBD ac:=ac-m[x]
0100xxxxxxxxxxx JPOS if ac >= 0 then pc:=x
0101xxxxxxxxxxx JZZR if ac=0 tehn pc:=x
0110xxxxxxxxxxx JUMP pc:=x
0111xxxxxxxxxxx LOCO ac:x (0 <= x <= 4095)
1000xxxxxxxxxxx LODL ac:=m[sp+x]
1001xxxxxxxxxxx STOL m[x+sp]:=ac
1010xxxxxxxxxxx ADDL ac:=ac+m[sp+x]
1011xxxxxxxxxxx SUBL ac:=ac-m[sp+x]
1100xxxxxxxxxxx JNEG if ac<0 then pc:=x
1101xxxxxxxxxxx JNZE if ac not= 0 then pc:=x
1110xxxxxxxxxxx CALL sp:=sp-1; m[sp]:=pc; pc:=x
1111000000000000 PSHI sp:=sp-1; m[sp]:=m[ac]
1111001000000000 POPI m[ac]:=m[sp]; sp:=sp+1
1111010000000000 PUSH sp:=sp-1; m[sp]:=ac
1111011000000000 POP ac:=m[sp]; sp:=sp+1
1111100000000000 RETN pc:=m[sp]; sp:=sp+1
1111101000000000 SWAP tmp:=ac; ac:sp; sp:tmp
11111100yyyyyyyy INSP sp:=sp+y (0 <= y <= 255)

11111110yyyyyyyy DESP sp:=sp-y (0 <= y <= 255)

Retomamos o conjunto de instruções de máquina.

Para cada instrução lida, o microprograma irá determinar qual das instruções ao lado é para executar.

Para isso, cada bit do código de operação é examinado.

Conjunto de instruções da máquina MAC

0000xxxxxxxxxxx LODD ac:=m[x]
0001xxxxxxxxxxx STOD m[x]:=ac
0010xxxxxxxxxxx ADDD ac:=ac+m[x]
0011xxxxxxxxxxx SUBD ac:=ac-m[x]
0100xxxxxxxxxxx JPOS if ac >= 0 then pc:=x
0101xxxxxxxxxxx JZZR if ac=0 tehn pc:=x
0110xxxxxxxxxxx JUMP pc:=x
0111xxxxxxxxxxx LOCO ac:x (0 <= x <= 4095)
1000xxxxxxxxxxx LODL ac:=m[sp+x]
1001xxxxxxxxxxx STOL m[x+sp]:=ac
1010xxxxxxxxxxx ADDL ac:=ac+m[sp+x]
1011xxxxxxxxxxx SUBL ac:=ac-m[sp+x]
1100xxxxxxxxxxx JNEG if ac<0 then pc:=x
1101xxxxxxxxxxx JNZE if ac not= 0 then pc:=x
1110xxxxxxxxxxx CALL sp:=sp-1; m[sp]:=pc; pc:=x
1111000000000000 PSHI sp:=sp-1; m[sp]:=m[ac]
1111001000000000 POPI m[ac]:=m[sp]; sp:=sp+1
1111010000000000 PUSH sp:=sp-1; m[sp]:=ac
1111011000000000 POP ac:=m[sp]; sp:=sp+1
1111100000000000 RETN pc:=m[sp]; sp:=sp+1
1111101000000000 SWAP tmp:=ac; ac:sp; sp:tmp
11111100yyyyyyyy INSP sp:=sp+y (0 <= y <= 255)

11111110yyyyyyyy DESP sp:=sp-y (0 <= y <= 255)

Exemplo: Suponha que PC aponta para a memória onde está a instrução LODD (em vermelho).

O código de operação 0000 é seguido por 12 bits especificando um endereço x. A instrução LODD carrega o valor da memória de endereço x no registrador AC.

Vamos ver como o microprograma lê uma instrução, descobre qual é, e executa as ações.

O microprograma completo na MIC

```
00: mar:=pc; rd
01: pc:=pc+1; rd
02: ir:=mbr; if n then goto 28
03: tir:=lshift(ir+ir); if n then goto 19
04: tir:=lshift(tir); if n then goto 11
05: alu:=tir; if n then goto 9
06: mar:=ir; rd
07: rd
08: ac:=mbr; goto 0
09: mar:=ir; mbr:=ac; wr
10: wr; goto 0
11: alu:=tir; if n then goto 15
12: mar:ir; rd
13: rd
...
```

Exemplo: Suponha que PC aponta para a memória onde está a instrução

0000xxxxxxxxxxx LODD ac:=m[x]

O microprograma começa lendo a instrução de máquina apontada por PC, decodifica para saber qual é a instrução, e executa.

Detalhamos a seguir, passo a passo.

O microprograma completo na MIC

```
00: mar:=pc; rd
01: pc:=pc+1; rd
02: ir:=mbr; if n then goto 28
03: tir:=lshift(ir+ir); if n then goto 19
04: tir:=lshift(tir); if n then goto 11
05: alu:=tir; if n then goto 9
06: mar:=ir; rd
07: rd
08: ac:=mbr; goto 0
09: mar:=ir; mbr:=ac; wr
10: wr; goto 0
11: alu:=tir; if n then goto 15
12: mar:ir; rd
13: rd
...
```

Ler da memória a próxima instrução a executar:

- PC é colocado em MAR
- Faz-se a leitura (ligando rd duas vezes)
- PC fica somado de 1. (Assim fica preparado para a próxima instrução de máquina.)

O microprograma completo na MIC

```
00: mar:=pc; rd
01: pc:=pc+1; rd
02: ir:=mbr; if n then goto 28
03: tir:=lshift(ir+ir); if n then goto 19
04: tir:=lshift(tir); if n then goto 11
05: alu:=tir; if n then goto 9
06: mar:=ir; rd
07: rd
08: ac:=mbr; goto 0
09: mar:=ir; mbr:=ac; wr
10: wr; goto 0
11: alu:=tir; if n then goto 15
12: mar:ir; rd
13: rd
...
```

Examina o primeiro bit da instrução lida:

- A instrução lida em MBR é transferida para IR
- Ao fazer isso, a instrução lida passou por ALU
- O teste "if n then goto 28" testa se o primeiro bit de IR é 1 (número negativo começa com 1 senão começa com 0)
- Se não começa com 1 então prossegue em frente. (É o nosso caso.)

O microprograma completo na MIC

```
00: mar:=pc; rd
01: pc:=pc+1; rd
02: ir:=mbr; if n then goto 28
03: tir:=lshift(ir+ir); if n then goto 19
04: tir:=lshift(tir); if n then goto 11
05: alu:=tir; if n then goto 9
06: mar:=ir; rd
07: rd
08: ac:=mbr; goto 0
09: mar:=ir; mbr:=ac; wr
10: wr; goto 0
11: alu:=tir; if n then goto 15
12: mar:ir; rd
13: rd
...
```

- Ao somar "ir + ir" a saída da ALU é um número que deslocou o valor de ir um bit para esquerda, portanto começa com o segundo bit de ir.
- O teste "if n then goto 19" testa se o segundo bit da instrução é 1 (número negativo começa com 1 senão começa com 0)
- Se não começa com 1 então prossegue em frente. (É o nosso caso.)
- Ao mesmo tempo, veja que tir já contém a instrução deslocada de 2 bits para esquerda.

O microprograma completo na MIC

```
00: mar:=pc; rd
01: pc:=pc+1; rd
02: ir:=mbr; if n then goto 28
03: tir:=lshift(ir+ir); if n then goto 19
04: tir:=lshift(tir); if n then goto 11
05: alu:=tir; if n then goto 9
06: mar:=ir; rd
07: rd
08: ac:=mbr; goto 0
09: mar:=ir; mbr:=ac; wr
10: wr; goto 0
11: alu:=tir; if n then goto 15
12: mar:ir; rd
13: rd
...
```

- Sendo os 2 primeiros bits 0, as microinstruções em vermelho testam o 3.o bit e o 4.o bit.
- Se o 3.o ou o 4.o bit for 1, o desvio é para goto 11 ou goto 9, resp.
- Se continuar na microinstrução 06, significa que os 4 primeiros bits da instrução valem 0000, ou seja, a instrução é LODD (ac = m [x])

O microprograma completo na MIC

```
00: mar:=pc; rd
01: pc:=pc+1; rd
02: ir:=mbr; if n then goto 28
03: tir:=lshift(ir+ir); if n then goto 19
04: tir:=lshift(tir); if n then goto 11
05: alu:=tir; if n then goto 9
06: mar:=ir; rd
07: rd
08: ac:=mbr; goto 0
09: mar:=ir; mbr:=ac; wr
10: wr; goto 0
11: alu:=tir; if n then goto 15
12: mar:ir; rd
13: rd
...
```

- Vamos executar ac = m [x]
- IR contém 0000xxxxxxxxxxx, portanto contém o endereço x (pois os 4 primeiros bits são todos 0, valem portanto os últimos 12 bits).
- Basta colocar IR em MAR e ler a memória (ligar rd duas vezes)
- O valor lido em MBR é transferido para AC.
- Retorna ao endereço 00 do microprograma para executar a próxima microinstrução

Microinstruções horizontais e verticais

Uma microinstrução especifica os sinais de controle necessários para controlar a microarquitetura.

- Microinstrução horizontal:
  - Todos os sinais necessários estão colocados na mesma microinstrução, sem nenhuma codificação.
  - 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.

Conclusão

O que pode se concluir de microprogramação?

- Ela é bem chata:-)
- Mas, falando sério, a microprogramação é uma técnica poderosa que permite implementar instruções complexas de um repertório grande de instruções em um hardware simples.

Conclusão

O que pode se concluir de microprogramação?

- Ela é bem chata:-)
- Mas, falando sério, a microprogramação é uma técnica poderosa que permite implementar instruções complexas de um repertório grande de instruções em um hardware simples.

Conclusão

O que pode se concluir de microprogramação?

- Ela é bem chata:-)
- Mas, falando sério, a microprogramação é uma técnica poderosa que permite implementar instruções complexas de um repertório grande de instruções em um hardware simples.