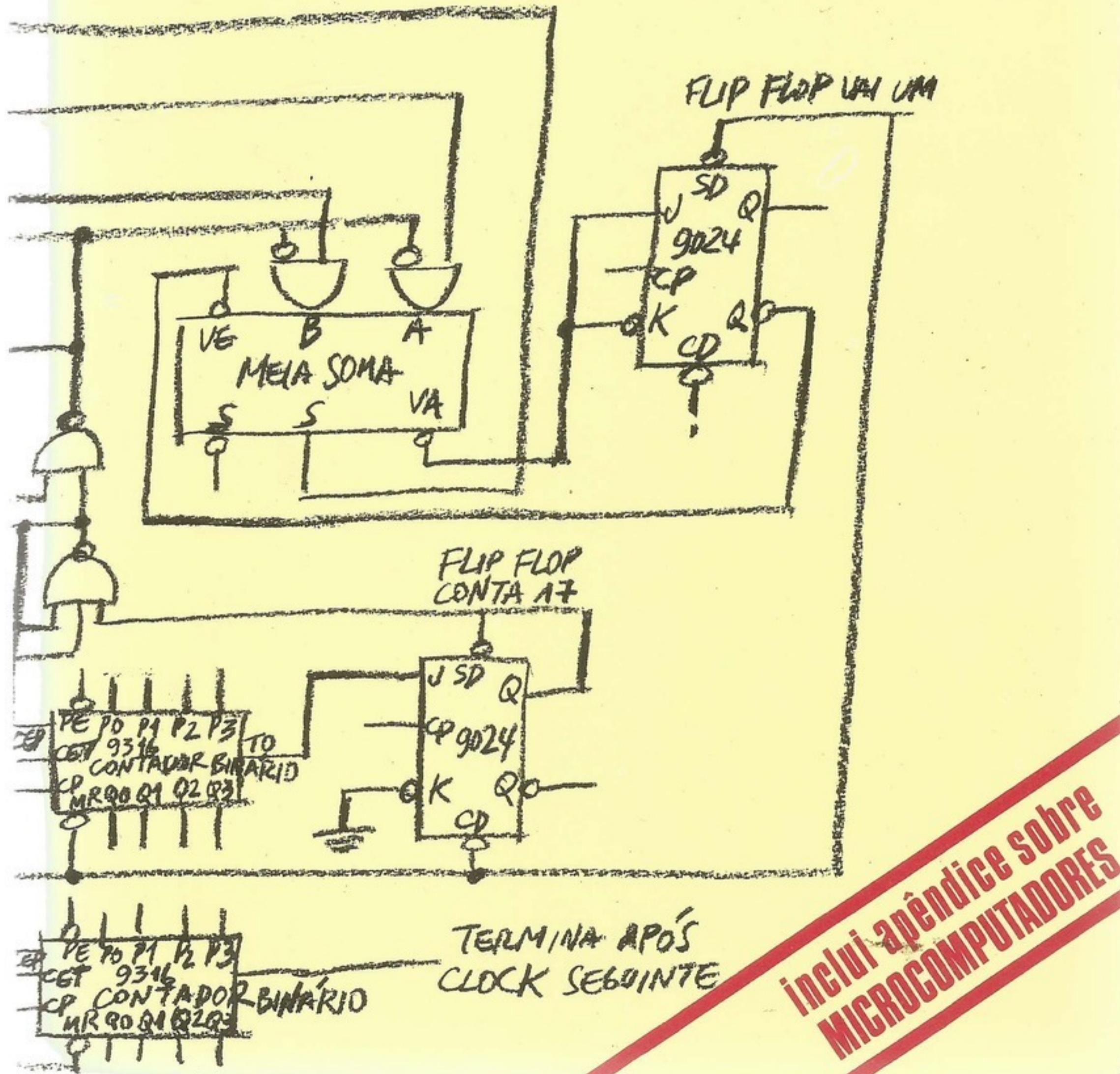


Glen George Langdon Jr.
Edson Fregni

PROJETO DE COMPUTADORES DIGITAIS

2.ª edição



PROJETO DE COMPUTADORES
DIGITAIS

FICHA CATALOGRÁFICA

Langdon, Glen George, 1937—
L263p Projeto de computadores digitais [por] Glen George
Langdon Júnior [e] Edson Fregni. São Paulo, Edgard
Blücher, 1974. 1977 2.^a edição.
ilust.
Bibliografia.
1. Computadores eletrônicos digitais — Projeto e cons-
trução I. Fregni, Edson, 1947— II. Título.

74-0646

17. CDD-621.381958
18. -621.3819582

Índices para catálogo sistemático:

1. Computadores digitais : Projeto : Engenharia eletrônica 621.381958 (17.)
621.3819582 (18.)
2. Projeto de computadores digitais : Engenharia eletrônica 621.381958 (17.)
621.3819582 (18.)

GLEN GEORGE LANGDON JUNIOR

*Professor visitante na Escola Politécnica da Universidade de
São Paulo. Pesquisador da IBM em San Jose, California, EUA*

EDSON FREGNI

*Mestre em Engenharia Elétrica; Professor da
Escola Politécnica da Universidade de São Paulo
Diretor técnico de SCOPUS TECNOLOGIA, São Paulo*

PROJETO DE COMPUTADORES DIGITAIS

2.ª edição



EDITORAS EDGARD BLÜCHER LTDA.

CONT

©1974 Editora Edgard Blücher Ltda.

1.^a Reimpressão da 2.^a edição 1977
2.^a Reimpressão da 2.^a edição 1979

*É proibida a reprodução total ou parcial
por quaisquer meios
sem autorização escrita da editora*

EDITORIA EDGARD BLÜCHER LTDA.

01000 CAIXA POSTAL 5450
END. TELEGRAFICO: BLUCHERLIVRO
SÃO PAULO — SP — BRASIL

Impresso no Brasil Printed in Brazil

CONTEÚDO

<i>Prefácio</i>	XV
Prefácio dos autores	XVI
1 <i>Introdução e conhecimentos básicos</i>	1
1.1 Organização de sistemas de computação	1
1.2 Álgebra booleana	5
a. Postulados e definições	5
b. Blocos lógicos	6
c. Teorema de álgebra booleana	7
1.3 Circuitos digitais	9
a. Introdução	9
b. Descrição de um circuito combinatório	10
c. Análise de circuitos combinatórios	12
d. Síntese de circuitos combinatórios	13
e. Determinação da soma mínima	15
f. Circuitos seqüenciais	19
g. Descrição de um circuito seqüencial	20
1.4 Tecnologia: transistores e diodos	23
a. Circuitos lógicos com diodos	23
b. O transistor como chave	24
c. Transistores em circuitos lógicos	26
1.5 Tecnologia: circuitos integrados digitais	28
a. Introdução	28
b. Família <i>RTL</i> (<i>resistor transistor logic</i>)	28
c. Família <i>DTL</i> (<i>diode transistor logic</i>)	29
d. Família <i>TTL</i> (<i>transistor transistor logic</i>)	29
e. Família <i>HTL</i> (<i>high threshold logic</i>)	30
f. Família <i>ECL</i> (<i>emitter coupled logic</i>)	30
g. Circuitos <i>MOS-FET</i>	31
<i>Exercícios</i>	33
<i>Bibliografia</i>	34
2 <i>Códigos, números e aritmética</i>	36
2.1 Códigos	36
2.2 Números	37
a. Sistemas	37
b. Números inteiros e fracionários	39

3.6	Fluxo de sinal
a.	Transistor
b.	Transistor
c.	Transistor
d.	Fluxo de sinal
	Exercícios
3.7	O timing de um circuito
a.	Exemplos
b.	A descrição
	Exercícios
3.8	Circuitos es
a.	Oscilador
b.	"Limpador"
c.	Entrada
d.	Driver
e.	Forneced
f.	Outros
g.	Circuito
3.9	Sumário
Bibliografia	
4	Memória e armazenamento
4.1	Introdução
4.2	O hardware
4.3	Memória programável
4.4	Memória de massa
4.5	Memória unidirecional
a.	Bipolar
b.	Memória
4.6	Sistemas de armazenamento
4.7	O sistema de armazenamento
a.	Motivação
b.	O sistema
c.	Localização
d.	A performance
e.	A memória
f.	A administração
g.	Assumptions
4.8	Armazenamento
a.	Componentes
b.	A técnica
c.	O sistema
d.	Exemplos
4.9	Memória virtual
a.	Motivação
b.	Transição
c.	O mecanismo
d.	O mecanismo
e.	Tabelas
f.	Tabelas
g.	Implementação
h.	Experiência

c.	Números negativos	39
d.	Números binários em complemento de um	39
e.	Números em complemento de dois	40
f.	Ponto fixo e ponto flutuante	41
2.3	Aritmética binária	42
a.	Números positivos	42
b.	A técnica da complementação	43
c.	Aritmética usando complemento de um do subtraendo	44
d.	Aritmética usando complemento de dois	47
e.	Aritmética com números em sinal e amplitude	48
f.	Variações	48
2.4	Aritmética decimal	50
a.	Representação dos negativos	51
b.	Exemplos de aritmética decimal	51
c.	Aritmética em <i>BCD</i> usando representação sinal e amplitude	51
2.5	Conversão entre números binários e decimais	53
a.	Conversão binária a decimal	55
b.	Conversão decimal a binária	56
2.6	Sumário	58
<i>Exercícios</i>		59
<i>Bibliografia</i>		59
		60
3	<i>Elementos do projeto lógico</i>	61
3.1	Introdução	61
3.2	Circuitos combinatórios	61
a.	Decodificadores	61
b.	Circuitos de paridade	62
c.	Circuitos de prioridade	63
	Exercícios	64
3.3	<i>Flip-flop</i>	64
a.	<i>Flip-flop RS</i>	66
b.	<i>Flip-flop</i> tipo <i>PH</i>	66
c.	<i>Flip-flop</i> tipo <i>D</i> , sensível à borda	67
d.	<i>Flip-flop</i> tipo <i>JK</i>	68
e.	<i>Flip-flop</i> master-slave	71
f.	Outros tipos de <i>flip-flops</i>	73
	Exercícios	74
3.4	Registradores	75
a.	Registrador deslocador	78
b.	Registrador-contador	79
c.	Contadores binários	85
d.	Contadores não-binários	85
	Exercícios	90
3.5	Unidades aritméticas	93
a.	Somadores	98
b.	Subtratores	98
c.	Complementação	107
d.	Funções lógicas	109
e.	Unidades aritméticas	110
f.	Exemplo detalhado de uma unidade aritmética	111
	Exercícios	114

39	3.6 Fluxo de sinais	115
39	a. Transferência de dados para registradores	115
40	b. Transferência em paralelo	118
41	c. Transferências por vias	121
42	d. Fluxo de dados	122
42	Exercícios	123
43	3.7 O <i>timing</i> e o ciclo da máquina	124
44	a. Exemplo	124
47	b. A determinação do ciclo da máquina	125
48	Exercício	126
50	3.8 Circuitos especiais	126
51	a. Osciladores	127
51	b. "Limpa ao ligar"	129
51	c. Entradas do sistema	129
53	d. <i>Driver</i>	129
55	e. Fontes de alimentação	130
56	f. Outros circuitos	131
58	g. Circuitos de interface com contatos mecânicos	132
59	3.9 Sumário	136
59	<i>Bibliografia</i>	137
60	4 Memória e armazenamento	138
61	4.1 Introdução	138
61	4.2 O <i>bandwidth</i> e suas implicações	138
61	4.3 Memória primária	140
62	4.4 Memória de núcleo de ferrite	141
63	4.5 Memória monolítica	147
64	a. Bipolar	148
64	b. Memória <i>MOS</i>	149
64	4.6 Sistemas de memória primária	152
66	4.7 O sistema <i>cache</i>	154
66	a. Motivação	154
67	b. O sistema <i>cache-backing</i>	154
68	c. Localidade de referência	154
71	d. A <i>performance</i> do esquema	155
73	e. A viabilidade do <i>cache</i>	156
74	f. A administração do <i>cache</i>	157
75	g. Assuntos relacionados ao <i>cache</i>	158
78	4.8 Armazenamento secundário	159
79	a. Composição de arquivos	159
85	b. A técnica de <i>blocking</i>	160
85	c. O sistema <i>IOCS</i>	160
90	d. Exemplo	161
93	4.9 Memória virtual	162
98	a. Motivação	162
98	b. Transcodificação dinâmica de endereço	162
107	c. O mecanismo de transcodificação	163
109	d. O tamanho dos blocos ou páginas	164
110	e. Tabelas de paginação de um nível	164
110	f. Tabelas de paginação de dois níveis	165
111	g. Implementação de relocação dinâmica no <i>S/360</i> modelo 67	167
114	h. Experiência com memória virtual	167

4.10	Sumário	167
	<i>Exercícios</i>	168
	<i>Bibliografia</i>	168
5	<i>Exemplo de um minicomputador</i>	170
5.1	Introdução	170
5.2	Esboço da arquitetura	170
5.3	Instruções principais	172
	a. Instruções longas.....	172
	a.1. Grupo principal	172
	a.2. Grupo de entrada/saída	173
	a.3. Grupo das imediatas	173
	b. Instruções curtas	175
	b.1. Microgrupo 1 (<i>MICG1</i>)	176
	b.2. Microgrupo 2a (<i>MICG2a</i>)	176
	b.3. Microgrupo 2b (<i>MICG2b</i>)	177
5.4	Fluxo de dados	178
	a. Análise dos circuitos	179
	a.1. Memória.....	181
	a.2. Registradores	181
	a.3. Unidade aritmética	185
	a.4. Portas de seleção	185
	a.5. Ciclo da máquina	188
	b. Implementação dos circuitos	189
	b.1. <i>FR1-1 e FR2-2</i>	190
	b.2. <i>FS1-3 e FS2-4</i>	190
	b.3. <i>FAC-5</i>	190
	b.4. <i>FM1-6</i>	191
	b.5. <i>FCM-7</i>	191
	b.6. <i>FIN-8</i>	191
5.5	Fases e microoperações	192
	a. Fases	193
	b. Microoperações	193
	c. Exemplo	194
5.6	Unidade de controle	194
	a. Considerações gerais	195
	b. Exemplo	195
	c. Decodificador	199
	d. Controle de estado	199
5.7	Interrupção	199
5.8	Modos de operação especiais	204
	a. Grupo 1	205
	b. Grupo 2	205
5.9	Sumário	206
5.10	Pranchas	207
	<i>Exercícios</i>	209
		225
6	<i>Arquitetura da UCP</i>	226
6.1	Introdução	226
6.2	Formatos de informação	226
	a. Tamanho de palavra	226

- b. Relações entre instruções
- c. Formato de instruções
- d. Operações de memória
- e. Relações entre operações
- f. Algoritmos de implementação
- g. Comunicação entre módulos
- 6.3 Especificações de arquitetura
 - a. Imediatas
 - b. Diretas
 - c. Indiretas
 - d. Indiretas de base
 - e. Relacionadas
 - f. Alternativas
 - g. Combinatórias
- 6.4 A posibilidade de implementação
 - a. Instruções
 - b. Instruções
 - c. Descrições
 - d. Descrições
 - e. Superestruturas
 - f. Outras
- 6.5 O PDP-8
 - a. Introdução
 - b. As instruções
 - c. Estudo de caso
 - d. Considerações finais
- 6.6 Arquitetura do PDP-8
 - a. Memória
 - b. As dimensões
 - c. Unidades de processamento
 - d. Comunicação
- 6.7 A arquitetura do PDP-8
 - a. Processador
 - b. Formato de instruções
 - c. Formato de operação
- 6.8 Feições da arquitetura
 - a. Introdução
 - b. Princípios
 - c. Relacionamento com o PDP-8
 - d. Considerações finais
 - e. Comunicação
 - f. O estudo de caso
 - g. A implementação
- 6.9 Sumário
 - Exercícios*
 - Bibliografia*
 - Leituras complementares*
- 7. Entrada/saída
 - 7.1 Introdução
 - 7.2 Dispositivos de saída
 - a. Funções
 - b. A comunicação
 - c. A ligação
 - d. Os sistemas
 - e. O monitor

167	b. Relacionamento operando/instrução	226
168	c. Formato dos operandos	227
168	d. O formato das instruções	228
170	6.3 Especificação do operando	230
170	a. Imediato ou implicado	230
170	b. Direto	231
170	c. Indireto	231
172	d. Indexado	231
172	e. Relativo	233
172	f. Abreviado	233
173	g. Combinações	233
175	6.4 A potência do conjunto de instruções	233
176	a. Instruções aritméticas	234
176	b. Instruções referentes à memória	235
177	c. Desvios	235
178	d. Deslocamento e operações booleanas	236
179	e. Supervisão e entrada/saída	237
181	f. Outras instruções	237
181	6.5 O PDP-8	237
181	a. Introdução	237
185	b. As instruções	238
185	c. Estados principais e interrupção	241
188	d. Comentário	242
189	6.6 Arquitetura da terceira geração e os sistemas IBM S/360 e IBM S/370	242
190	a. Motivação	242
190	b. As direções da arquitetura	242
191	6.7 A arquitetura do IBM S/360	243
191	a. Ponto de vista do programador	244
191	b. Formato dos dados	244
192	c. Formatos da instrução e do endereçamento	245
193	6.8 Feições do S/360 e S/370 para multiprogramação	248
193	a. Introdução e o sistema <i>Stretch</i>	248
194	b. Proteção da memória	248
194	c. Relocação e transcodificação dinâmica do endereço	249
195	d. O PSW e o mecanismo de interrupção	249
195	e. Comentário sobre o PSW	250
199	f. O estado “supervisor”	251
199	g. A instrução “ <i>test and set</i> ” (TS)	251
199	6.9 Sumário	251
204	<i>Exercícios</i>	252
205	<i>Bibliografia</i>	252
205	<i>Leituras complementares</i>	253
206		
207	7. Entrada/saída	254
209		
225	7.1 Introdução	254
225	7.2 Dispositivos e suas interfaces	255
226	a. Funções dos periféricos	255
226	b. A essência do controle dos periféricos	255
226	c. A ligação entre a UCP e o dispositivo	255
226	d. Os sinais da interface	257
226	e. O <i>timing</i> da via de E/S	258

e.1. Caso síncrono	258
e.2. Caso assíncrono	259
f. Características tecnológicas da via	260
7.3 A interface do programa e o dispositivo	260
a. Programado	261
b. Acesso direto à memória (<i>DMA</i>)	262
c. Canal	263
d. Processador periférico (<i>CDC 6600</i>)	263
7.4 Interrupção	263
a. Classes e tratamento geral de interrupção	263
b. Interrupção de nível simples	264
c. Interrupção de níveis múltiplos	265
d. <i>Hardware</i> generalizado para cada nível	265
e. Técnicas para efetuar a interrupção	267
f. Técnicas para descobrir a causa da interrupção	269
g. Interrupção para casos de falta de energia	269
7.5 Interrupção no <i>HP 2116B</i>	270
a. Assuntos gerais	270
b. Os níveis e a seqüência da interrupção	271
c. A rede de prioridade	271
d. Os <i>flip-flops control</i> e <i>flag</i>	271
e. O projeto da parte da interrupção do cartão de interface	272
f. O subsistema de interrupção da <i>UCP</i>	274
g. Carta de <i>timing</i> da interrupção	275
7.6 A arquitetura de E/S no <i>S/360</i>	277
a. Assuntos gerais	277
b. Os compromissos da implementação	277
c. O formato e os tipos de comando	278
d. A seqüência típica das operações	279
e. Unidades de controle	280
f. As características dos canais	280
g. A interface padronizada	281
7.7 O painel	282
a. Função	282
b. Contatos mecânicos	283
c. A leitura e a escrita da memória principal	283
d. A carga do programa inicial (<i>IPL</i>)	283
e. Controles “roda”, “partida” e “pare”	283
7.8 Sumário e conclusões	283
<i>Exercícios</i>	284
<i>Bibliografia</i>	284
8 Unidades de controle	285
8.1 Conceitos gerais	285
a. Possíveis classificações das unidades de controle	287
8.2 Controle fixo	288
a. Registrador do código da instrução	288
b. Registrador de endereço de instrução	288
c. Decodificador	289
d. Gerador de sinais de controle	289
e. Relógio central	289
f. Controle do modo de operação	289

8.3 Unidade de

- a. Memória
- b. Paralelo
- c. Codificação
- d. Resumo

8.4 Comparação

- a. Facilidade
- b. Flexibilidade
- c. Manutenção
- d. Velocidade
- e. Economia

*Bibliografia**Apêndice**Índice*

.....	258	8.3 Unidade de controle microprogramada	290
.....	259	a. Monofásicos e polifásicos	291
.....	260	b. Paralelo e série	292
.....	260	c. Codificado e não-codificado	292
.....	261	d. Resumo	292
.....	262	8.4 Comparação entre as duas técnicas de controle	293
.....	263	a. Facilidade de projeto	293
.....	263	b. Flexibilidade	293
.....	263	c. Manutenção	293
.....	263	d. Velocidade	293
.....	264	e. Emulação	293
.....	265	<i>Bibliografia</i>	294
.....	265	<i>Apêndice Os microcomputadores</i>	295
.....	267	<i>Índice</i>	353
.....	269		
.....	269		
.....	270		
.....	270		
.....	271		
.....	271		
.....	271		
.....	272		
.....	274		
.....	275		
.....	277		
.....	277		
.....	277		
.....	278		
.....	278		
.....	279		
.....	280		
.....	280		
.....	281		
.....	282		
.....	282		
.....	283		
.....	283		
.....	283		
.....	283		
.....	284		
.....	284		
.....	284		
.....	285		
.....	285		
.....	287		
.....	288		
.....	288		
.....	288		
.....	289		
.....	289		
.....	289		
.....	289		

PREFÁCIO

As atividades relacionadas com projeto, montagem e produção de computadores, iniciadas há alguns anos no Brasil, estão sendo intensificadas rapidamente. Diversas iniciativas importantes nessa nova área foram tomadas, ou serão tomadas a curto prazo, tanto em universidades como na indústria, em grandes empresas estatais e outros órgãos do Governo. Como consequência, o país defronta-se com a necessidade de um número crescente de especialistas em sistemas digitais, e começa a formá-los. A publicação deste livro, ajuda significativa para a solução desse problema, é, portanto, muito oportuna.

É também auspicioso que essa tarefa tenha sido encetada sob a responsabilidade de profissionais altamente credenciados como o doutor Glen G. Langdon Jr. e o engenheiro Edson Fregni.

O doutor Langdon obteve com brilhantismo o título de doutor na University of Syracuse, em 1967, após seu mestrado na University of Pittsburgh e sua graduação na Washington University. Toda sua atividade acadêmica, suas inúmeras publicações e sua intensa atividade profissional têm sido concentradas na mesma área enfocada neste livro. Em várias oportunidades, o doutor Langdon ministrou cursos de pós-graduação na Escola Politécnica da Universidade de São Paulo, onde tem contribuído, com rara dedicação, para o desenvolvimento do Laboratório de Sistemas Digitais do Departamento de Engenharia de Eletricidade.

O engenheiro Fregni graduou-se pela Escola Politécnica da Universidade de São Paulo, em 1972, onde obteve o grau de Mestre em Engenharia. Freqüentou vários cursos e desenvolveu atividades de pesquisa no Digital Systems Laboratory da Stanford University, e está engajado num programa de doutoramento. Pertence ao corpo docente da Escola Politécnica, e participa das atividades do Laboratório de Sistemas Digitais. Os trabalhos realizados pelo engenheiro Fregni, neste seu início de vida profissional, prenunciam uma brilhante carreira.

Nessas condições, foi com prazer que recebi a incumbência de escrever esta página introdutória, seguro de que o primeiro livro do doutor Langdon e do engenheiro Fregni será de grande valia para os interessados nos fascinantes problemas dos computadores e de todos os sistemas digitais.

Antonio Hélio Guerra Vieira

PREFÁCIO DOS AUTORES

A primeira versão deste texto surgiu como “notas de aulas” do curso de pós-graduação “Projeto de sistemas digitais”, ministrado pelo professor Langdon, na Escola Politécnica da Universidade de São Paulo (EPUSP), no primeiro semestre de 1971. Tal curso era dirigido principalmente aos engenheiros e professores do Laboratório de Sistemas Digitais (LSD) da EPUSP com planos para projetarem um minicomputador. No segundo semestre de 1971, o professor Langdon convidou o engenheiro Fregni, que assistira ao curso, para, juntos, reescreverem aquelas notas de aula. Surgiu então uma segunda versão, chamada “Estruturas de computadores”, em sete capítulos (os primeiros sete capítulos deste livro). Ambos os autores, bem como outros professores da EPUSP, tiveram oportunidade de usar tal texto em vários cursos, tanto de graduação, como de pós-graduação. Muito os autores devem aos estudantes e professores da EPUSP, principalmente aos elementos do LSD, pelos erros encontrados e modificações sugeridas.

Acredita-se ser desnecessário justificar a existência de tal livro no Brasil. Aqui ele não é “mais um”, pois pouca coisa se tem publicado em português nessa área. Espera-se que tal trabalho venha a incentivar o início de outras publicações correlatas. Na época da organização do seu primeiro curso na EPUSP, o professor Langdon não encontrou livro algum no idioma inglês que se adaptasse às necessidades, ou seja, que, além de detalhar o funcionamento das unidades básicas de um computador mostrasse como elas se interligam e integravam. Espera-se que o presente trabalho tenha conseguido reunir essas duas qualidades. Por isso, acreditam os autores que este livro seja válido e útil, não apenas pelo fato de ter sido escrito em português.

Este livro é dirigido principalmente aos estudantes de engenharia de computação ou ciência de computação, nos últimos anos de graduação ou em nível de pós-graduação. Conhecimentos de álgebra booleana e teoria da comutação (*switching theory*) são pré-requisitos, apesar de uma breve introdução a esses assuntos ser dada no Capítulo 1. Além disso, procurou-se redigir este texto de forma tal a facilitar o engenheiro autodidata que queira se atualizar no assunto, mas que não disponha de tempo livre para freqüentar algum curso de pós-graduação. Os autores testaram este manuscrito em um curso organizado por eles e pelo engenheiro Victor Penna da Rocha, da IBM do Brasil, e ministrado pelos professores do LSD aos engenheiros e técnicos da fábrica da IBM do Brasil, em Campinas (SP). Os estudantes, apesar de não possuírem muita experiência no assunto, estavam se preparando para a produção e ensaio da unidade central de processamento de um computador de porte médio (IBM S/370, modelo 145) e unidades de controle de fitas magnéticas. Constatou-se que este texto se adapta bem às circunstâncias onde o espírito totalmente acadêmico é posto em segundo plano, destacando-se os detalhes práticos do assunto.

Os primeiros capítulos apresentam as unidades básicas de um sistema isoladamente e os últimos analisam o problema de como interligar essas unidades, e como controlar os sinais no tempo (*timing*).

O Capítulo 1 oferece um resumo do que se entende serem os pré-requisitos para os capítulos seguintes. Esse capítulo é muito útil para o professor, nas primeiras aulas do curso, quando normalmente se faz uma revisão rápida dos pré-requisitos. Um problema básico nessa área é o da representação dos números internamente à máquina. Sabe-se que os computadores digitais eletrônicos usam grandezas elétricas, tais como tensões ou correntes, para representar as constantes e variáveis do problema que está sendo resolvido.

O Capítulo 2 é dedicado ao estudo e comparação das inúmeras maneiras de se representarem tais números e operar com tais representações. Um sistema eletrônico tão sofisticado quanto um computador digital seria tremendamente complicado para se entender e projetar se o problema fosse tratado ao nível dos transistores. Por isso, o que se faz é definir algumas unidades básicas, tais como registradores, unidades aritméticas etc., que são interligadas para formar o sistema completo. Tais unidades, que, por sua vez, são subdivididas em partes menores, os blocos lógicos (Capítulo 1), são estudadas no Capítulo 3.

O Capítulo 4 é reservado para o estudo das memórias. Dedica-se um capítulo a esse assunto, dada a importância dessa unidade também pelo fato de estar em franco desenvolvimento, acompanhando a evolução da tecnologia. Nesse capítulo discutem-se também alguns aspectos de como a organização do sistema pode melhorar a eficiência das unidades. Tome-se, por exemplo, a organização tipo "memória virtual", onde se faz uma memória grande e lenta, associada a uma pequena, mas veloz, comportar-se como uma memória grande e de alta velocidade.

No Capítulo 5, existe uma descrição detalhada de um minicomputador. A existência de tal material é justificada pela necessidade de um exemplo que ajude o estudante, a essa altura, a entender como as unidades, estudadas isoladamente, funcionam em um todo. Tal computador é o mesmo que foi projetado e montado após o primeiro curso do professor Langdon, mais tarde chamado de "Patinho Feio" pelos membros da equipe que o projetou. Pelo que se tem conhecimento, esse foi o primeiro computador digital brasileiro.

No Capítulo 6, estudam-se as diferentes maneiras de organizar as unidades básicas formando um sistema completo ("arquitetura"). Analisa-se nesse capítulo como tal organização evoluiu de geração para geração. Apresentam-se também alguns exemplos reais, tais como *PDP-8*, *IBM S/360* e *Burroughs 5000*.

O Capítulo 7 é dedicado ao estudo dos processos de entrada e saída. Também aí se manteve o que os autores procuraram fazer em todo o livro: apresentar inúmeras diferentes soluções para o mesmo problema, e sempre que possível citar um exemplo real de onde e por que se adotou tal solução. No final desse capítulo estuda-se o sistema de entrada e saída dos computadores *HP2116B* e *IBM S/360*.

O Capítulo 8 oferece um breve estudo das unidades de controle. Dá-se ênfase às diferenças entre as unidades de controle cujo funcionamento é definido por circuitos (*hardwired*) e as microprogramadas. Ao longo de todo o livro procurou-se, sempre que possível, apresentar alguns exercícios ao leitor. Recomendamos aos professores que adotarem este texto em algum curso que o usem como tarefa para os estudantes, pois tais exercícios complementam o material apresentado, e são peças importantes para o aprendizado.

Um problema enfrentado pelos autores durante a redação deste trabalho foi a necessidade de se escrever em *português* sobre um assunto aprendido pela maioria em inglês. É fato sabido que os técnicos evitam traduzir os nomes dos elementos ou conceitos introduzidos pela literatura em inglês. Isso é devido, principalmente, ao medo de não se fazerem entender, caso se aventurem a traduzi-los. Este trabalho, que se propõe ser um livro-texto, tem a grande responsabilidade de introduzir muitas pessoas a essa área, e se os autores não fossem cuidadosos e, por que não dizer, conservadores, correr-se-ia o risco de dificultar o prosseguimento do estudo dessas pessoas através das referências fornecidas, que, na sua grande maioria, são escritas no idioma inglês. Entretanto os termos que são normalmente traduzidos, e cuja tradução é única e fácil de se voltar ao inglês, foram aqui traduzidos sem

referência ao nome original^[1]; outros termos cuja tradução não é ainda geralmente usada, mas que já começa a se popularizar, foram traduzidos com o correspondente termo em inglês fornecido entre parênteses^[2]. Foram escritas em inglês, sem se tentar uma tradução, as palavras que ou são usadas em inglês pela maioria^[3] ou cujos conceitos são muito novos para que se possa determinar qual a tradução adotada^[4]. Deve-se esclarecer, entretanto, que o fato de serem usados muitos termos em inglês não implica que os autores concordem com a "anglicanização" do nosso idioma. A uniformização de tais termos deveria ser tarefa de alguma revista técnica, que, infelizmente, inexiste em nosso idioma.

Para terminar, os autores gostariam de agradecer à Editora Edgard Blücher Ltda. e à Editora da USP, pelo apoio nesta tão pouco lucrativa tarefa; ao professor A. H. Guerra Vieira, pelas críticas e sugestões ao nosso trabalho bem como pelo apoio fundamental para que nossos planos em relação a este livro se tornassem realidade; a toda equipe do LSD, pelas críticas e sugestões, da qual gostaríamos de citar em particular o engenheiro Célio Yoshiyuki Ikeda, pelas intermináveis discussões; ao engenheiro Walter Rodrigues Ferreira Filho, pelo trabalho de coordenação durante a revisão final do texto, enquanto os autores se encontravam fora do país. Finalmente, à IBM (IBM do Brasil, IBM World Trade e IBM Endicott N.Y.) pelo apoio dado — sem o qual este livro não existiria — quando da visita do professor Langdon à EPUSP.

São Paulo, fevereiro de 1974

Edson Fregni
Glen George Langdon, Júnior

^[1]Núcleo magnético, instrução, memória etc.

^[2]"Vai-um antecipado" (*carry lookahead*), via (*bus*)

^[3]Hardware, flip-flop, fan-out etc.

^[4]Cache, schedules etc.

capítulo

INTROD

Os autores ficariam muito gratos se sugestões e eventuais falhas fossem comunicadas para

Professor Edson Fregni
Laboratório de Sistemas Digitais
Escola Politécnica da USP – DEE
Cidade Universitária
05508 – São Paulo – SP

1.1 ORGANIZAÇÃO

Em 1950, os primeiros computadores eram capazes de calcular (em dade aritmética) com precisão de 20 dígitos e eram controlados por uma máquina de comando. Por falta de tempo, os resultados eram suficientes para cobrir quase todo o espaço em conta a tempo. As válvulas a vácuo eram usadas para controlar os computadores, que eram acionados com relés pela eletrônica. O resultado do cálculo de todos os resultados era feita por circuitos que eram facilmente mudados para armazenar resultados pelo relatório.

É desse período que foram implementados os primeiros computadores.

Atualmente, os computadores consistem em um processador, somador, registradores e memória. O conjunto formado por processador, somador, registradores e memória é conhecido como unidade central de processamento (UCP).

O programador pode executar conjuntos de instruções de operação de 8 bits. Cada instrução contém um código de instruções que indica qual instrução deve ser executada. As instruções contêm endereços de memória que estão sendo manipuladas.

A separação entre instruções e endereços de memória não permite a execução de instruções de maneira

capítulo 1

INTRODUÇÃO E CONHECIMENTOS BÁSICOS

1.1 ORGANIZAÇÃO DE SISTEMAS DE COMPUTAÇÃO

Em 1833, o matemático inglês Charles Babbage apresentou os planos de uma máquina de calcular (engenho analítico)⁽¹⁾ provida de memória (1 000 números de 50 dígitos), unidade aritmética (que somava dois números de 50 dígitos em 1 s e multiplicava dois números de 20 dígitos em 3 min⁽²⁾) e instruções. As instruções foram baseadas no “Jacquard loom”, uma máquina para controlar a tecelagem com fios coloridos, aperfeiçoados por Babbage pela inclusão de uma instrução do tipo desvio condicional. Essa máquina não foi realizada por falta de tecnologia, já que os dispositivos mecânicos disponíveis no século XIX não eram suficientes para implementar a organização projetada. Babbage foi o primeiro a descobrir que, no projeto de organização de uma máquina de computação, é necessário levar-se em conta a tecnologia (*hardware*) existente. Somente após o desenvolvimento dos relés e válvulas a vácuo que as idéias de Babbage foram redescobertas e realizadas. O primeiro computador, o Harvard Mark I, projetado por Aiken (de Harvard) em 1939 e implementado com relés pela IBM em 1944⁽³⁾, utilizou os princípios de Babbage. O primeiro computador eletrônico foi o Eniac, cujo projeto começou em 1943 (com a aplicação específica para o cálculo de tabelas de balística), ficando operacional em 1946. No Eniac a programação era feita por chaves e pela colocação de vários fios em soquetes (*plugboard wiring*). Não era fácil mudar o programa e, através dessa experiência, evoluiu-se para o conceito de *programa armazenado* (*stored program concept*), cuja idéia foi apresentada pela primeira vez em 1946 pelo relatório de Burks, Goldstine e von Neumann⁽⁴⁾.

É desse relatório que conhecemos a clássica “máquina tipo von Neumann”. As idéias foram implantadas no computador *IAS*, cuja organização aparece na Fig. 1-1.

Atualmente, a unidade aritmética é conhecida como *fluxo de dados*, constando de um somador, registradores centrais e as vias (“busses”) de interligação entre essas unidades. O conjunto formado pelos fluxo de dados, unidade de controle e memória é chamado de *unidade central de processamento (UCP)*. A unidade de controle recebe as instruções codificadas da memória. A unidade aritmética e a entrada/saída recebem e fornecem dados para a memória. Os dados são codificados em binário de 40 bits, lidos da memória em paralelo.

O programa consta de instruções, que são codificadas e armazenadas na memória, em conjunto com os dados. Cada instrução é dividida em duas partes: um campo de código de operação de 8 bits e um campo de endereço de 12 bits [Fig. 1-1(b)]. Essa máquina dispõe de instruções para modificar ou carregar o campo de endereço de outras instruções. Assim, é possível o uso da técnica de programação de *loop*. Os dois principais aperfeiçoamentos do computador *IAS* sobre o esquema de Babbage foram, então, (1) o armazenamento das instruções na memória, juntamente com os dados e (2) a possibilidade de se operarem as instruções como se fossem dados. É interessante notar que atualmente essas duas técnicas estão sendo criticadas.

A separação física das instruções dos dados aumenta a confiabilidade e a técnica de não permitir a modificações das instruções, facilita a descoberta e eliminação de erros

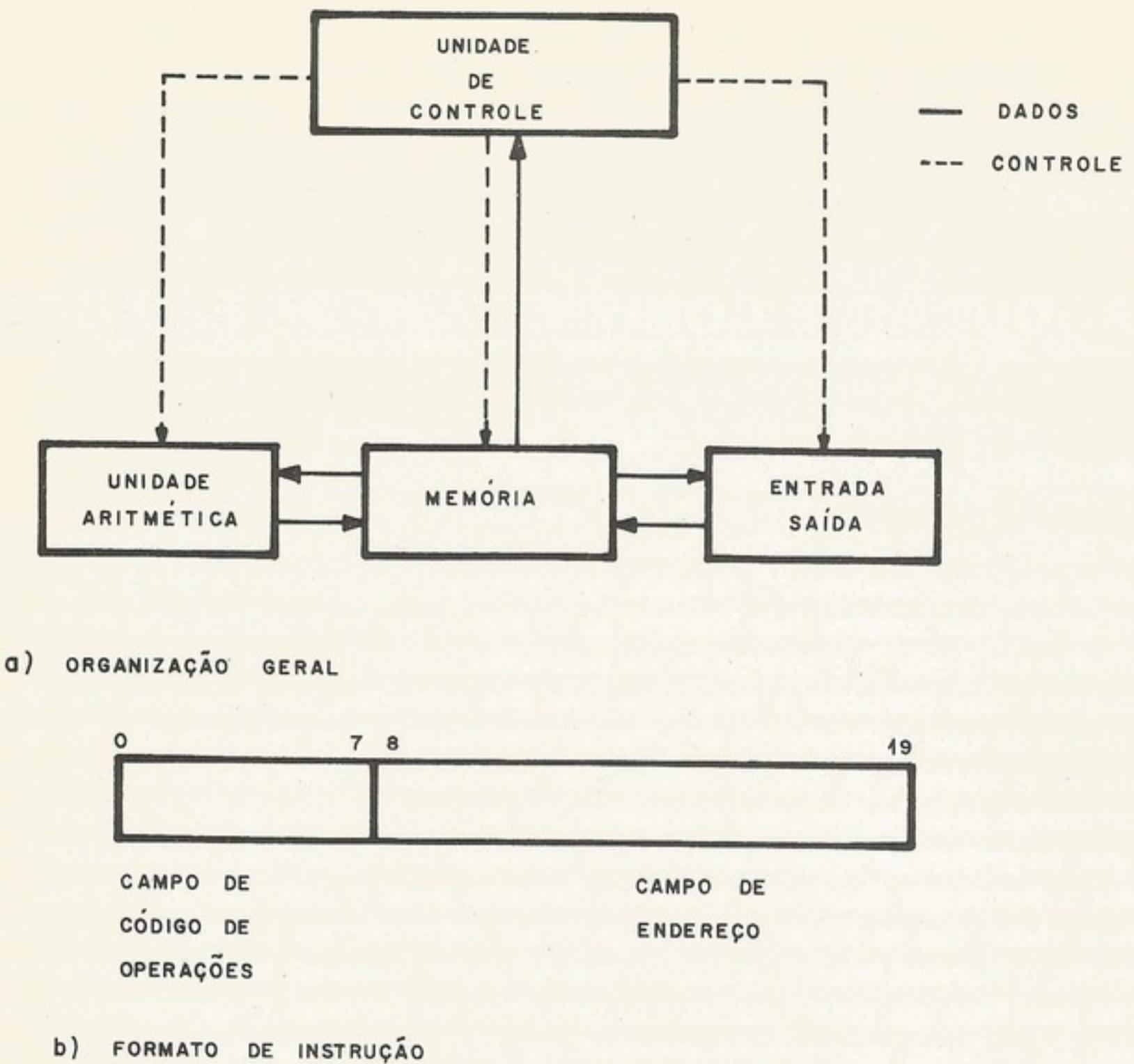


Figura 1-1. O computador IAS

(debugging) e a modificação e atualização dos programas. (Além do mais, existem dificuldades de simulação e emulação de programas que modificam a si mesmos).

No projeto do computador *IAS*, considerou-se a possibilidade de implementação das instruções em ponto flutuante, o que foi rejeitado por motivos econômicos. Para compensar, foram implantadas instruções de deslocamento que facilitaram a programação das sub-rotinas de ponto flutuante.

Um programa de instruções, ou simplesmente um *programa*, é o meio pelo qual o computador faz o processamento de dados. A unidade central só pode retirar as instruções da memória e executá-las exatamente como foi programado. Um programa mal feito é capaz de cometer muitos erros. (Muitos caracterizam o computador como o idiota mais veloz do mundo.)

As características do computador do ponto de vista do programador, são enquadradas dentro do que chamamos de *arquitetura* do sistema de computação. O conjunto de instruções, a organização e o endereçamento da memória, a interface e o método de entrada/saída, o painel (*console*) e o sistema de interrupção, são todos aspectos de arquitetura. A aplicação da tecnologia na implementação da arquitetura em *hardware* é o *projeto lógico* do sistema. O engenheiro, na implementação do sistema, tem de estudar a relação entre vários métodos, avaliando o custo e a *performance* de cada um (ou seja, estudar os compromissos ou *trade-offs*).

Um compromisso (*trade-off*) muito comum em sistemas de computação é o de “espaço-tempo”, onde o programador pode tornar seu programa mais veloz, ao custo de uma quantidade maior de memória principal. O mesmo compromisso ocorre em *hardware*, onde a *UCP*, por exemplo, pode processar instruções com mais velocidade ao custo de mais circuitos.

Desde a época do desenvolvimento do primeiro computador, pelo transistores dispostos em tecnologia planar (usando endereçamento especializado para memória). Também evoluíram os controladores fazem a execução. Para se indicar a geração, “segunda

Na primeira geração, a memória de líquido cristal eletrostático. Na segunda geração, a memória primária usava memórias simples, com poucas células.

Na terceira geração, os circuitos lógicos. Transistores de silício. A memória primária usava endereçamento eficiente. A memória secundária desenvolveu-se com o surgimento do disco rígido da instrução. Os computadores usam entradas/saídas de interrupção. O processador é feito com circuitos integrados, compenhando-se as diferenças entre a ajuda o operador. O processador deve ser feito a menor custo. Os computadores usam números decimais para velocidade, números binários para precisão.

No começo da era dos circuitos integrados, muitos fabricantes usavam uma só arquitetura para diferentes tipos de circuitos integrados, formando uma “família” de chips com diferentes performances (e custos). O sistema compreende o processador e que cuida do usuário. Software de gerenciamento apoiava os programadores visores ou monitores. A terceira geração incluiu um processador centralizado, o sistema operacional, a UCP, a memória, o diagrama de um computador das gerações anteriores.

A ideia de “hardware” é tinta escura em fundo branco, e arquitetura típica de computador.

Desde a época dos primeiros computadores, houve desenvolvimentos fantásticos. O desenvolvimento mais acentuado foi na tecnologia, onde a válvula a vácuo foi substituída pelo transistor discreto e este pelo circuito integrado. Acompanhando o desenvolvimento em tecnologia, a arquitetura dos sistemas tem recebido melhoramentos no endereçamento (usando endereços relativos, indiretos ou indexados), circuitos para ponto flutuante, canais especializados para entrada/saída independente e esquemas de *interrupção* de programas. Também evoluíram sistemas de programação chamados *software*, onde os programas monitores fazem a escalação dos serviços e regulam o fluxo de trabalho através do sistema. Para se indicarem as etapas desses desenvolvimentos, é comum o uso dos termos “primeira geração”, “segunda geração” e “terceira geração”.

Na *primeira geração* (1946-1956), a tecnologia era caracterizada pela válvula a vácuo, memória de linhas de atraso (tipicamente de mercúrio) ou de tubo de raios catódicos (*CRT*) eletrostático. No fim da primeira geração, foi introduzida a memória de núcleo de ferrite. A memória principal era muito pequena (da ordem de 1 024 a 4 096 palavras). Alguns sistemas usavam tambores magnéticos para memória principal. A entrada/saída era muito simples, com pouca concorrência com o processamento.

Na *segunda geração* (1956-1963), foi introduzido o transistor na tecnologia dos circuitos lógicos. Transistores, resistores e capacitores eram montados em placas de circuito impresso. A memória principal era de núcleo de ferrite (como na terceira geração). Para calcular o endereço efetivo das instruções, eram usados registradores de índice. Também nessa época, desenvolveu-se o conceito de endereçamento indireto, onde o conteúdo do campo de endereço da instrução é o endereço do operando (em vez do próprio operando). Muitos sistemas usam entrada/saída (*E/S*) com processamento simultâneo auxiliado por um esquema de *interrupção*. O *canal* controla a *E/S* independentemente da *UCP*. O armazenamento secundário é feito com tambores e fita magnética. O processamento é feito em “lotes”, desempenhando-se os serviços (*job* ou tarefa) seqüencialmente, com um programa monitor que ajuda o operador a evitar tempo ocioso da unidade central. Esse programa determina quando deve ser feita a montagem de fitas magnéticas, e também faz a contabilidade dos serviços. Os computadores dessa geração dividiam-se em duas categorias: “comercial” (mais lento, números decimais e nada para facilitar operações em ponto flutuante); e “científico” (mais veloz, números binários, com instruções que facilitavam ponto flutuante).

No começo da *terceira geração* (1964), a tecnologia era a de transistores montados em circuitos integrado, com uma “integração” de um ou dois circuitos lógicos por pastilha. Muitos fabricantes resolveram reunir aspectos do processamento comercial e científico numa só arquitetura. Além disso, essa mesma arquitetura (conjunto de instruções, endereçamento, formato de dados, esquema de *interrupção* e entrada/saída) foi implementada numa “família” de vários modelos [modelos de baixa *performance* (e custo) até modelos de alta *performance* (e custo)]. Dos sistemas de programação, evoluiu o *sistema operacional*, um sistema complexo de rotinas que oferece muitas opções e serviços aos usuários do sistema, e que cuida do andamento dos programas dos usuários. Esse sistema é *software* bem sofisticado. *Software* é uma palavra genérica usada para designar os programas do sistema que apóiam os programas de aplicação do usuário, tais como compiladores, montadores, supervisores ou monitores, editores, sub-rotinas de entrada/saída, e programas utilitários. Com a terceira geração, introduziu-se a técnica de *multiprogramação*, onde pode haver mais de um programa na memória principal ao mesmo tempo. Multiprogramação entrelaça o processamento de programas diferentes para melhor aproveitar os principais recursos do sistema, a *UCP*, a memória principal e o armazenamento em fita ou disco magnético. Um diagrama de um sistema de terceira geração aparece na Fig. 1-2 e as características principais das gerações são indicadas na Tab. 1-1.

A idéia de “geração” vem sendo amplamente criticada, pois a noção de gerações distintas esbarra em muitas exceções. O *CDC 6600*, um sistema de computador de grande porte e arquitetura típica da terceira geração, foi implementado em transistores discretos, o que

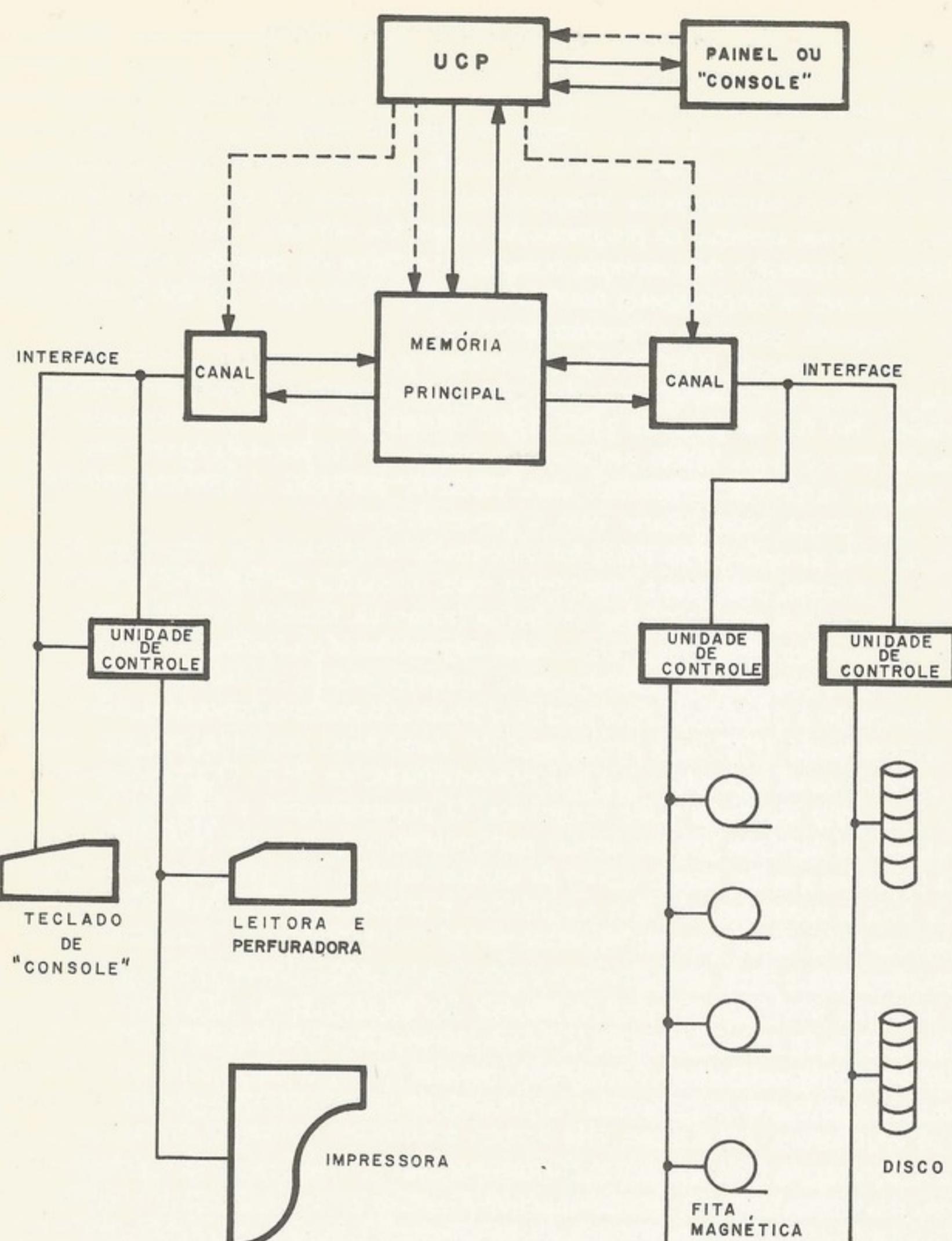


Figura 1-2. Um computador de terceira geração

é uma característica da segunda geração. E, ainda, muitos computadores implementados em circuitos integrados em grande escala, os minicomputadores, possuem algumas características da primeira geração.

Na primeira geração, os interesses eram mais concentrados na unidade central e as técnicas de projeto lógico tinham grande influência na organização do sistema. Naquela época o *hardware* era bem mais caro que a programação. Os engenheiros pensavam em termos de US\$10 por válvula ou *gate* lógico. Hoje em dia, o *software* e a programação são mais caros. Então aqueles que falam na arquitetura da quarta geração tendo em vista os novos compromissos pensam mais nas possibilidades de se aumentar o *hardware* para facilitar a programação do *software*. Também, em vez de se considerar o sistema como uma simples máquina de fazer somas e multiplicações, está surgindo a idéia da “informação” e seu fluxo entre os arquivos de armazenamento, a memória principal, e os registradores

Tabela 1-1. Características das gerações

Geração	Tecnologia	Processamento	Operação
Primeira	Válvulas a vácuo	Escalação com hora marcada	Por programador-operador
Segunda	Transistores discretos	Em lotes	Programa monitor simples para ajudar o operador
Terceira	Circuitos integrados	Multiprogramação	O operador segue as exigências de um sistema operacional complexo

centrais da UCP conforme a atividade. Hoje há grande interesse na organização de dados em arquivos (*file organization*) para tornar mais eficiente o acesso às informações. Os sistemas de computação já estão sendo caracterizados, em grande contraste com os primeiros sistemas usados para fazer cálculos rotineiros para tabelas matemáticas, como sistemas de informação.

1.2 ÁLGEBRA BOOLEANA

A álgebra booleana foi desenvolvida pelo matemático George Boole (1815-1864) para o estudo da lógica. Ela foi apresentada em 1854 no seu *An investigation of the laws of thought*⁽⁵⁾ e, em 1938, Claude E. Shannon⁽⁶⁾ adaptou a álgebra booleana para o projeto de redes telefônicas.

a. Postulados e definições

A álgebra booleana é definida sobre um conjunto com dois elementos. Esses elementos recebem, comumente, os nomes *verdadeiro* e *falso*, ou *alto* e *baixo*, ou “0” e “1”. O leitor deve estar alerta para o fato de não existir significado numérico algum nesses elementos. No campo dos sistemas digitais, esses dois valores são dois níveis de tensão pré-fixados nos quais associamos os nomes “0” e “1”. Uma variável que só pode assumir os valores “0” e “1” é chamada de variável booleana.

O conjunto $V = \{0,1\}$ é definido pelos seguintes postulados: seja x uma variável booleana; então

$$\begin{aligned} (P.1) \quad & \text{se } x \neq 1, \text{ então } x = 0; \\ (P.2) \quad & \text{se } x \neq 0, \text{ então } x = 1. \end{aligned}$$

O complemento de uma variável booleana x (que pode ser representado por um dos símbolos \bar{x} , x' ou $-x$) é definido pelos seguintes postulados:

$$\begin{aligned} (P.3) \quad & \text{se } x = 0, \text{ então } x' = 1; \\ (P.4) \quad & \text{se } x = 1, \text{ então } x' = 0. \end{aligned}$$

Existem duas operações básicas na álgebra booleana, que são chamadas de *AND* (*E*) e *OR* (*OU*):

Operação AND

Usualmente representada pelo símbolo \cdot (ou \wedge), é definida pelos seguintes postulados:

$$\begin{aligned} (P.5) \quad & 0 \cdot 0 = 0; \\ (P.6) \quad & 0 \cdot 1 = 0; \\ (P.7) \quad & 1 \cdot 0 = 0; \\ (P.8) \quad & 1 \cdot 1 = 1. \end{aligned}$$

Operação OR

Usualmente representada pelo símbolo + (ou \vee), é definida pelos seguintes postulados:

$$(P.9) \quad 0 + 0 = 0;$$

$$(P.10) \quad 0 + 1 = 1;$$

$$(P.11) \quad 1 + 0 = 1;$$

$$(P.12) \quad 1 + 1 = 1.$$

Uma outra operação booleana (ou binária) também importante é a *exclusive OR* (OU exclusivo), abreviadamente *XOR* (OU X). Ela é usualmente representada pelo símbolo \oplus (ou $\vee\!\vee$), e pode ser definida pelos seguintes postulados:

$$(P.13) \quad 0 \oplus 0 = 0;$$

$$(P.14) \quad 0 \oplus 1 = 1;$$

$$(P.15) \quad 1 \oplus 0 = 1;$$

$$(P.16) \quad 1 \oplus 1 = 0.$$

É fácil demonstrar que as três operações definidas são comutativas e associativas. Sejam A , B e C três variáveis booleanas. Então,

operação AND (passaremos a omitir o ponto nas expressões),

$$\begin{aligned} A B &= B A, \\ (A B) C &= A (B C) = A B C; \end{aligned}$$

operação OR,

$$\begin{aligned} A + B &= B + A; \\ (A + B) + C &= A + (B + C) = A + B + C; \end{aligned}$$

operação XOR,

$$\begin{aligned} A \oplus B &= B \oplus A, \\ (A \oplus B) \oplus C &= A \oplus (B \oplus C) = A \oplus B \oplus C. \end{aligned}$$

b. Blocos lógicos

O nosso estudo de álgebra booleana objetiva exclusivamente sua aplicação aos sistemas digitais. Vamos então ilustrar esse estudo com comentários sobre os elementos lógicos digitais e, para que isso seja possível, vamos abrir um parêntese no nosso estudo para dar algumas definições necessárias.

Existem unidades funcionais básicas, que chamaremos de blocos lógicos, com os quais sintetizamos os circuitos digitais através de combinações. Neste mesmo capítulo faremos um breve estudo desses circuitos digitais. Apresentaremos 4 blocos lógicos básicos.

Inversor ou NOT

É o bloco que realiza a complementação (ou inversão) de uma variável booleana. A Fig. 1-3 mostra os vários símbolos desse bloco.

AND (E)

Tem várias entradas, e a saída é o resultado da operação AND das entradas (Fig. 1-4). Seu funcionamento pode ser caracterizado pela seguinte frase: "a saída do bloco AND será igual a '1' se, e apenas se, todas as entradas forem iguais a '1'".

OR (OU)

Tem várias entradas, e a saída é o resultado da operação OR das entradas (Fig. 1-5). Seu funcionamento pode ser caracterizado pela seguinte frase: "a saída do bloco OR será igual a '0' se, e apenas se, todas as entradas forem iguais a '0'".

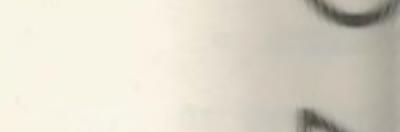
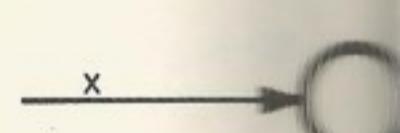
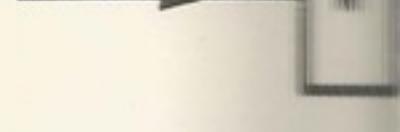
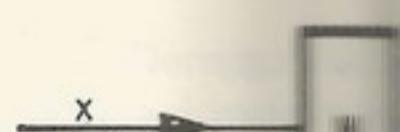
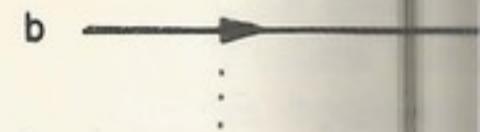
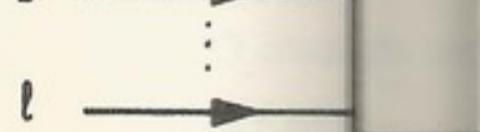
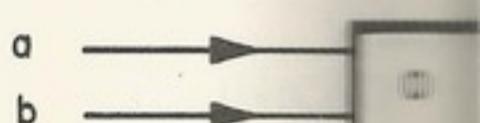
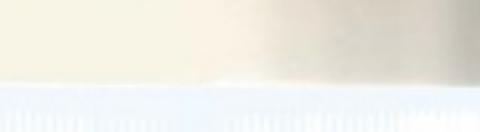
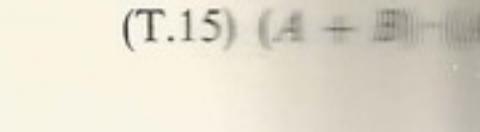
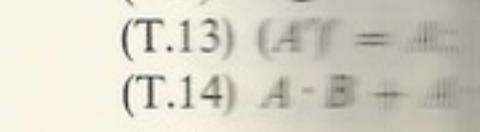
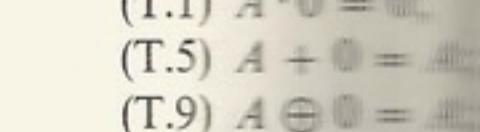
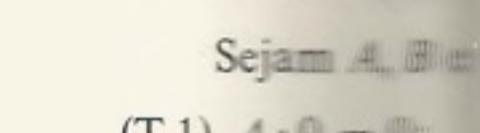
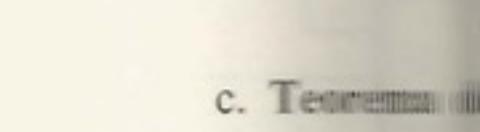
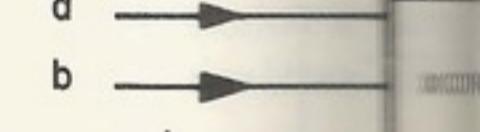
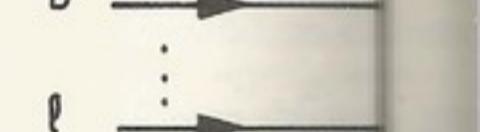
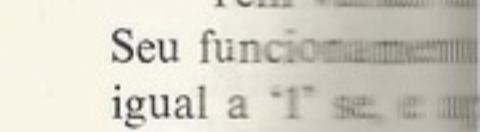


Figura 1-3. Símbolos lógico NOT



XOR (OU EXCLUSIVO)

Tem várias entradas, seu funcionamento é igual a '1' se, e apenas se,



c. Teoremas de De Morgan

Sejam A , B e C variáveis booleanas.

$$(T.1) \quad A \cdot 0 = 0;$$

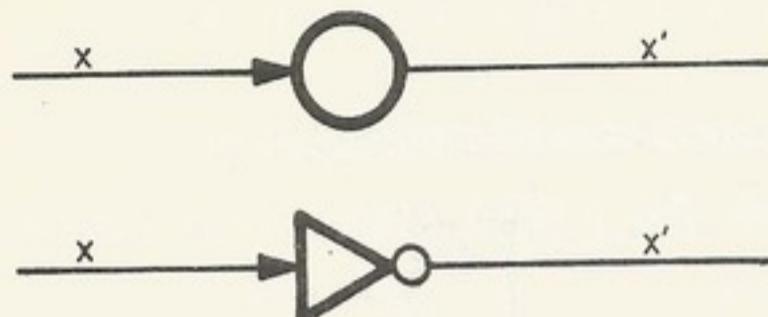
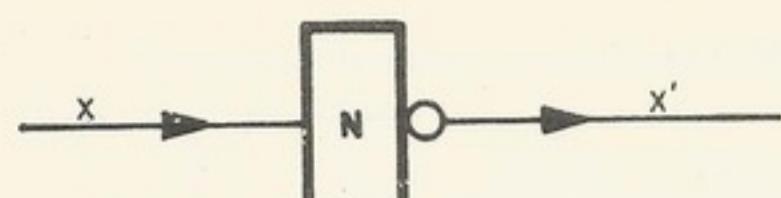
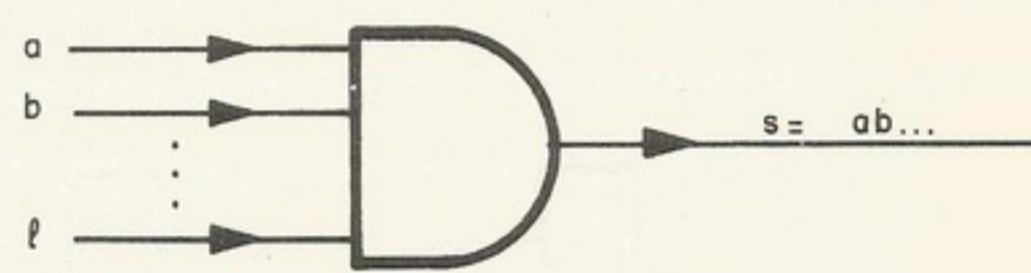
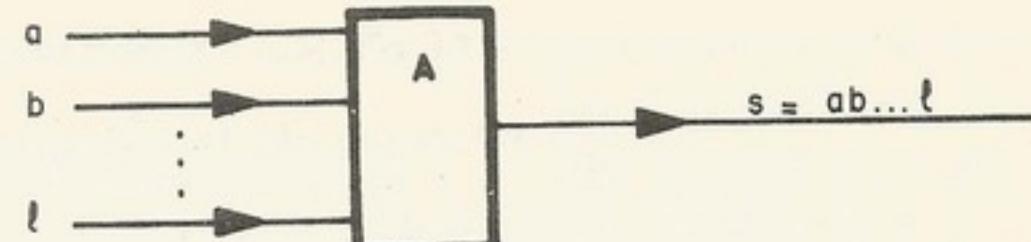
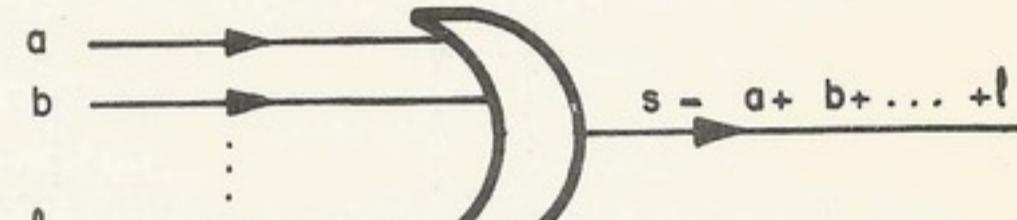
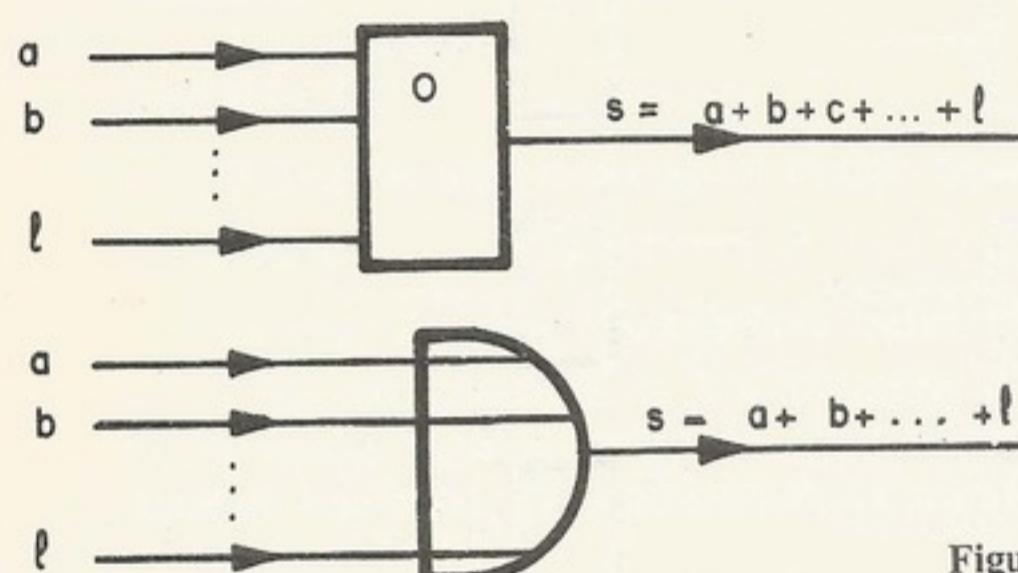
$$(T.5) \quad A + 0 = A;$$

$$(T.9) \quad A \oplus 0 = A;$$

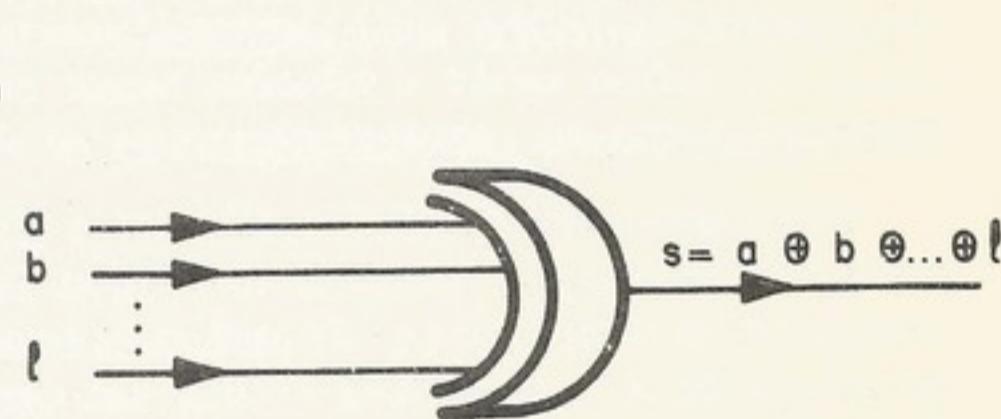
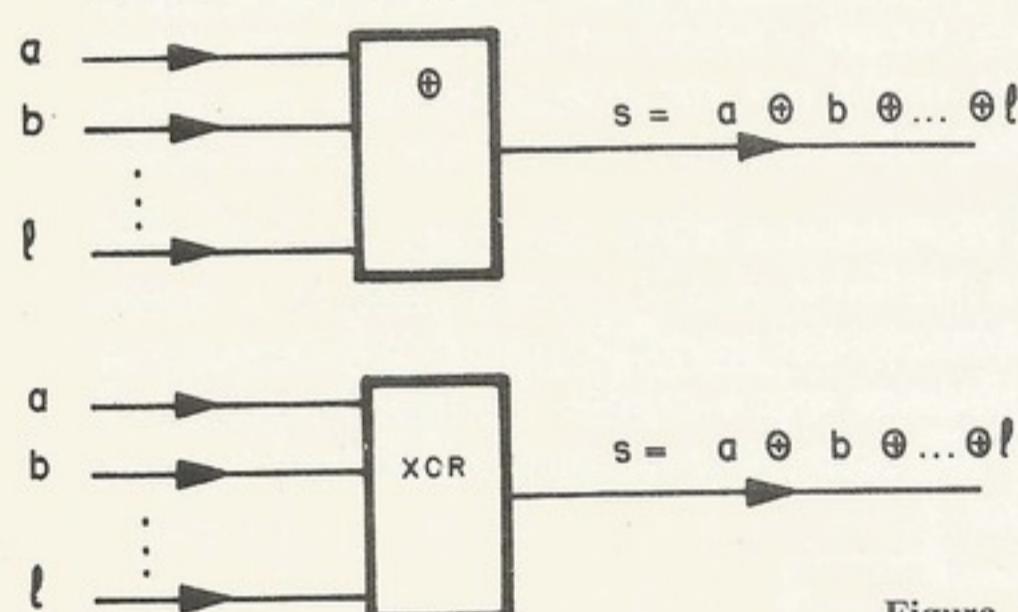
$$(T.13) \quad (A')' = A;$$

$$(T.14) \quad A \cdot B + A \cdot C = A \cdot (B + C);$$

$$(T.15) \quad (A + B)' = A' \cdot B';$$

Figura 1-3. Símbolos mais comuns do bloco lógico *NOT*Figura 1-4. Símbolos mais comuns do bloco lógico *AND*Figura 1-5. Símbolos mais comuns do bloco lógico *OR****XOR (OUX)***

Tem várias entradas e a saída é o resultado da operação *XOR* das entradas (Fig. 1-6). Seu funcionamento pode ser caracterizado pela seguinte frase: "a saída do bloco *XOR* será igual a '1' se, e apenas se, existir um número *ímpar* de entradas iguais a '1'".

Figura 1-6. Símbolos mais comuns do bloco lógico *XOR***c. Teorema de álgebra booleana**

Sejam A , B e C três variáveis booleanas. É possível demonstrar as seguintes identidades:

- | | | | |
|--|----------------------------|---------------------------|----------------------------|
| (T.1) $A \cdot 0 = 0$; | (T.2) $A \cdot 1 = A$; | (T.3) $A \cdot A = A$; | (T.4) $A \cdot A' = 0$; |
| (T.5) $A + 0 = A$; | (T.6) $A + 1 = 1$; | (T.7) $A + A = A$; | (T.8) $A + A' = 1$; |
| (T.9) $A \oplus 0 = A$; | (T.10) $A \oplus 1 = A'$; | (T.11) $A \oplus A = 0$; | (T.12) $A \oplus A' = 1$; |
| (T.13) $(A')' = A$; | | | |
| (T.14) $A \cdot B + A \cdot C = A \cdot (B + C)$ | | | |
| (T.15) $(A + B) \cdot (A + C) = A + B \cdot C$ | | | |
- } (fatoração);

- (T.16) $A \cdot B + A \cdot B' = A$; (T.17) $A + A \cdot B = A$; (T.18) $A + A' \cdot B = A + B$;
 (T.19) $(A \cdot B)' = A' + B'$
 (T.20) $(A + B)' = A' \cdot B'$ (teoremas de De Morgan);
 (T.21) $A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$;
 (T.22) $A \oplus B = A \cdot B' + A' \cdot B$;
 (T.23) $(A \oplus B)' = A' \oplus B = A \oplus B' = A' \cdot B' + A \cdot B$.

Na Fig. 1-7 apresentamos o significado físico de algumas dessas identidades.

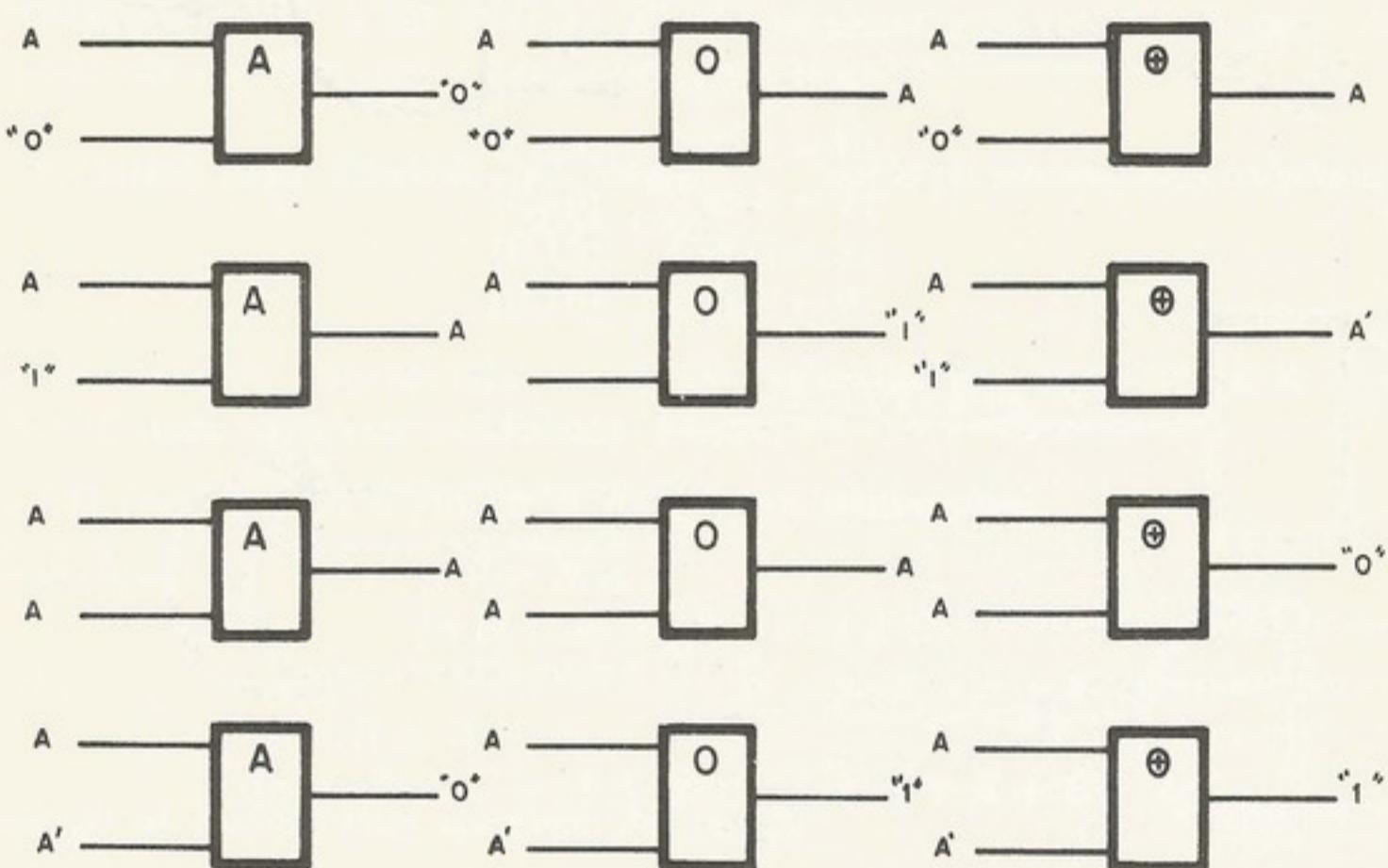


Figura 1-7. Significado físico das doze primeiras identidades

A demonstração dos teoremas pode ser feita pela sua verificação para todos os valores das variáveis, já que o conjunto universo tem apenas dois elementos. Algumas identidades podem ser demonstradas por dedução através da manipulação das expressões utilizando-se os teoremas anteriores e os postulados.

EXEMPLO. Prova por verificação

Seja a identidade (T.19), $(A \cdot B)' = A' + B'$. Montamos a tabela que segue, consideramos todas as combinações possíveis de valores das variáveis A e B (colunas 1 e 2), a seguir calculamos os valores de $(A \cdot B)'$ para cada combinação das entradas (coluna 3) e, finalmente, os valores de $A' + B'$ (coluna 4). Por fim verificamos que a coluna 3 é igual à coluna 4. Portanto a igualdade (T.19) é verdadeira para todos os valores possíveis das variáveis, o que mostra ser essa igualdade uma identidade.

1	2	3	4
A	B	$(A \cdot B)'$	$A' + B'$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

EXEMPLO. Prova por dedução

Seja a identidade (T.16), $A \cdot B + A \cdot B' = A$. Basta evoluirmos, conforme segue:

$$(T.14) \quad (T.8) \quad (T.2)$$

$$A \cdot B + A \cdot B' = A \cdot (B + B') = A \cdot 1 = A.$$

Demonstrações por manipulação da expressão, algumas vezes, são árduas e artificiosas.
Seja a identidade

$$(T.21) A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C,$$

$$(T.2) \quad (T.8)$$

$$A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C + B \cdot C \cdot 1 = \\ (T.14)$$

$$= A \cdot B + A' \cdot C + B \cdot C \cdot (A + A') = A \cdot B + A' \cdot C + B \cdot C \cdot A + B \cdot C \cdot A'.$$

Como as operações *AND* e *OR* são associativas e comutativas, podemos escrever

$$(T.15)$$

$$A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A \cdot B \cdot C + A' \cdot C + A' \cdot C \cdot B = A \cdot B + A' \cdot C.$$

Wood⁽⁷⁾ apresenta o conceito de dualidade, que é muito útil no estudo da álgebra booleana.

DEFINIÇÃO

O dual de uma expressão é obtido trocando-se as operações *AND* por *OR* e vice-versa.

TEOREMA

O dual de uma identidade é outra identidade.

Com esse conceito, se, por exemplo, demonstramos o teorema (T.19) a expressão dual (T.20) estará também provada. Para o leitor interessado num estudo mais detalhado da álgebra booleana, recomendamos as referências⁽⁷⁾ e⁽⁸⁾.

1.3 CIRCUITOS DIGITAIS

a. Introdução

A grosso modo, chamaremos de circuitos digitais (ou circuitos lógicos ou, ainda, circuitos de comutação) a combinação de blocos lógicos interligados. Na Fig. 1-8(a) apresentamos um exemplo de um circuito digital. Esquematicamente [Fig. 1-8(b)] podemos representar um circuito digital por um bloco com várias entradas e várias saídas. As saídas dependem das entradas e cada linha delas pode ter apenas dois valores, que representaremos por "0" e "1".

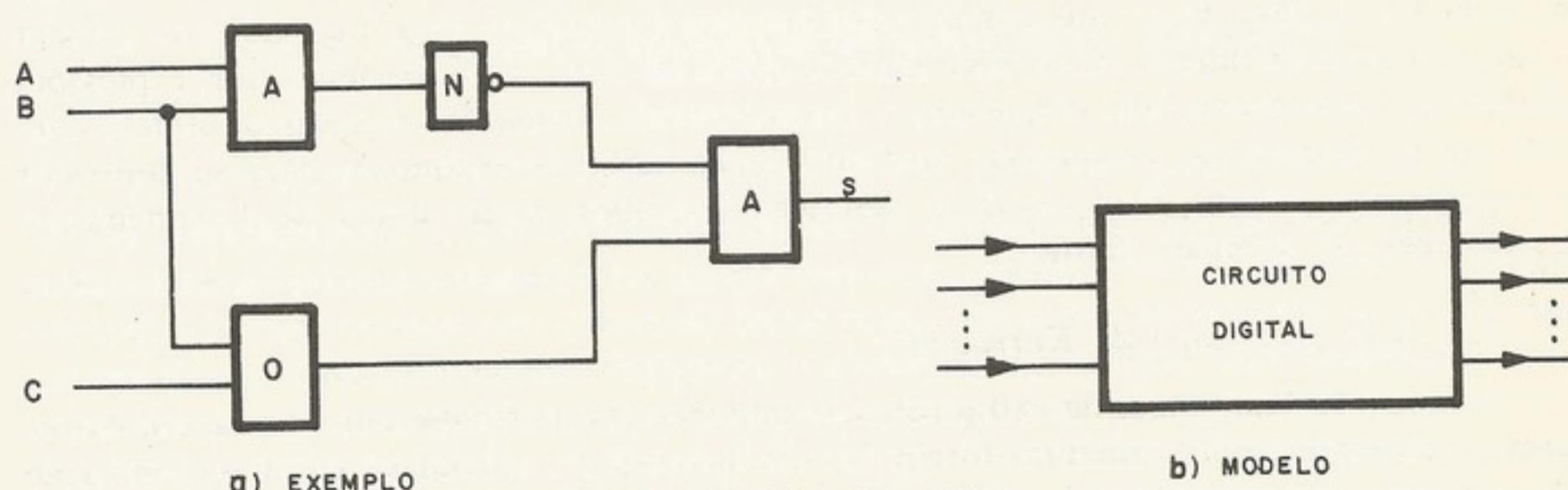


Figura 1-8. Circuitos digitais

Os circuitos lógicos são divididos em duas classes, *combinatórios* e *seqüenciais*. Circuitos combinatórios (ou combinacionais) são aqueles cuja saída depende apenas dos valores atuais das entradas. Circuitos seqüenciais são aqueles cuja saída depende dos valores atuais e dos valores anteriores da entrada. Têm uma espécie de memória interna.

b. Descrição de um circuito combinatório

Vamos estudar aqui apenas o caso de circuitos de apenas uma saída. O leitor pode generalizar para o caso de n saídas, sem muitos problemas.

Consideremos um circuito combinatório com 3 entradas (Fig. 1-9). Já que um circuito combinatório se caracteriza por ter a saída dependendo apenas das entradas atuais, podemos dizer que a saída (s) é dada por $s = F(a, b, c)$. Vamos estudar aqui as várias maneiras de se descrever a função $F(a, b, c)$, que define um circuito combinatório.

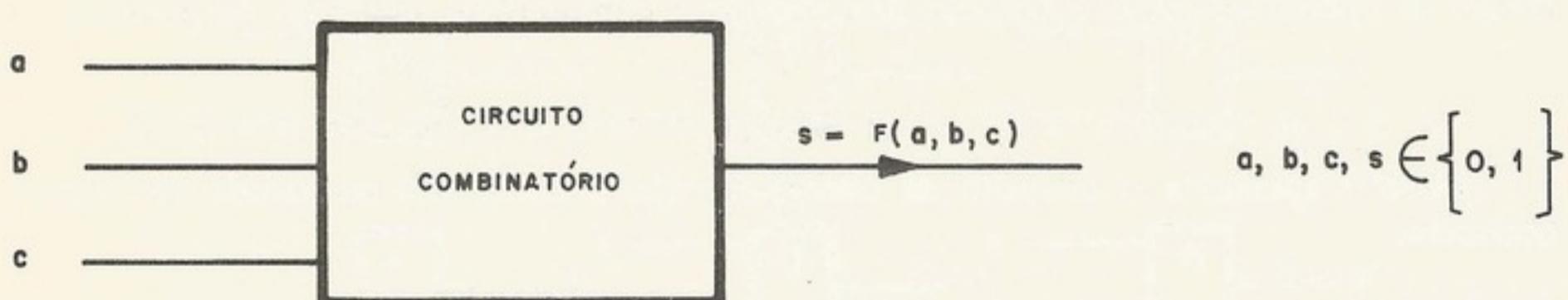


Figura 1-9. Esquema de um circuito combinatório

Apresentaremos quatro maneiras diferentes de se descrever um circuito combinatório, além de seu diagrama lógico (desenho detalhado do circuito mostrando os blocos e as interligações): tabela de combinações, expressão booleana, mapa de Karnaugh e transformada numérica. Conforme o objetivo da descrição, cada maneira tem suas vantagens, seja na documentação, seja nas várias formas de síntese ou, ainda, para ressaltar alguma particularidade importante dos circuitos.

Descrição pela tabela de combinações (tabela da verdade)

A função $s = F(a, b, c)$ pode ser dada por uma tabela, conforme se vê na Fig. 1-10(b). Nessa tabela, fazemos uma primeira coluna com todas as combinações dos valores possíveis das entradas. E, para cada combinação, temos a segunda coluna com o valor correspondente de saída (s). Essa descrição para funções com um grande número de variáveis tem o inconveniente de apresentar um excessivo número de linhas.

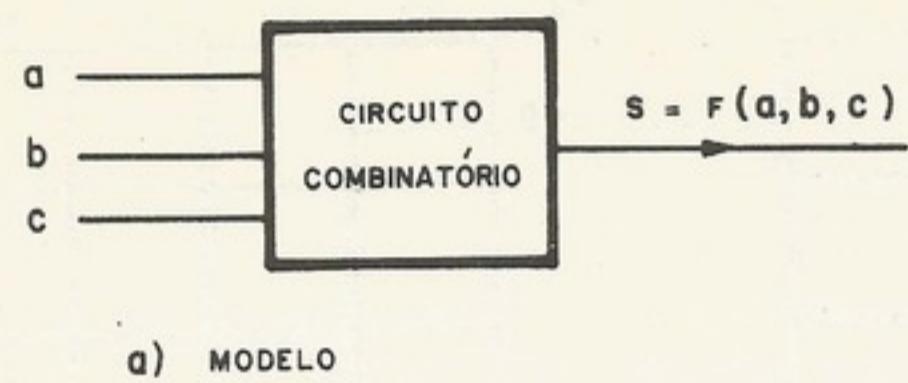
Descrição por uma expressão booleana

A função $s = F(a, b, c)$ pode ser apresentada por uma expressão booleana, conforme se vê na Fig. 1-10(c). Para cada combinação possível dos valores de a, b e c pode-se calcular o valor da correspondente saída. Essa maneira não é unívoca, isto é, existem várias expressões diferentes, equivalentes, que descrevem o mesmo circuito. Uma das técnicas de síntese consiste em se determinar a expressão booleana equivalente mais simples, para se conseguir o circuito mais barato. Essa forma de descrição é também muito usada na documentação e simulação de sistemas digitais.

Descrição pelo mapa de Karnaugh⁽⁹⁾

O mapa de Karnaugh, de extraordinária importância na síntese em dois níveis, é uma tabela com tantas células quantas forem as combinações das entradas. A cada combinação das entradas, corresponde uma célula, dentro da qual colocamos o correspondente valor da saída.

Na Fig. 1-11, apresentamos as tabelas para circuitos com duas, três e quatro entradas, para um maior número de entradas, a descrição pelo mapa perde as suas vantagens. A forma de montagem do mapa é feita de tal maneira que as células vizinhas correspondem a duas combinações de valores das entradas, com apenas uma das entradas tendo seu valor alterado. Na Fig. 1-10(d) há um exemplo do mapa na descrição de um circuito combinatório.



a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$$s = f(a, b, c) = \bar{a}c + ab$$

c) EXPRESSÃO BOLEANA

b) TABELA DE COMBINAÇÕES

bc \ a	0	1
00	0	0
01	1	0
11	1	1
10	0	1

d) MAPA DE KARNAUGH

$$TN(s) = [1100 \quad 1010]_2$$

e) TRANSFORMADA NUMÉRICA

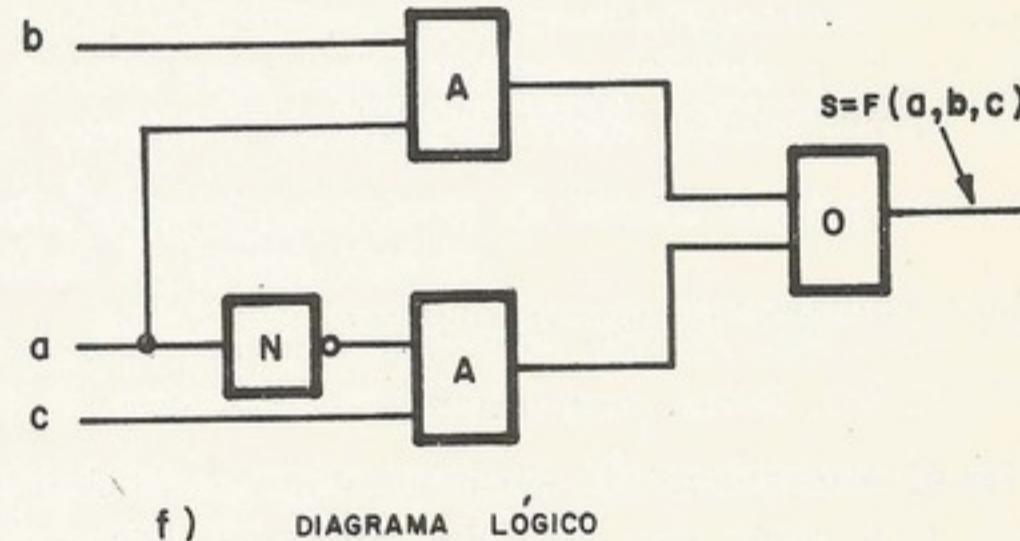


Figura 1-10. Formas de descrição de um mesmo circuito combinatório

Descrição pela transformada numérica⁽¹⁰⁾

Se considerarmos a tabela de combinação e supusermos que as combinações das entradas sejam sempre ordenadas na ordem crescente, apenas a coluna de saída descreverá o circuito. Se escrevermos essa coluna de baixo para cima como um número na base 2, esse número (transformada numérica) descreverá o circuito [Fig. 1-10(c)]. Essa descrição é recomendada quando se pretende analisar o circuito por meio de computadores.

Vamos estudar o seguinte problema: dado um circuito definido por uma certa descrição, determinar uma outra descrição.

As interconversões entre as descrições por tabelas da verdade, mapa de Karnaugh e transformada numérica são imediatas. Para convertermos a descrição através de expressão booleana para o mapa de Karnaugh ou tabela da verdade, basta calcularmos o valor da expressão para cada combinação dos valores das entradas.

A conversão mais delicada é a da tabela de combinações para a expressão booleana. É o que veremos a seguir.

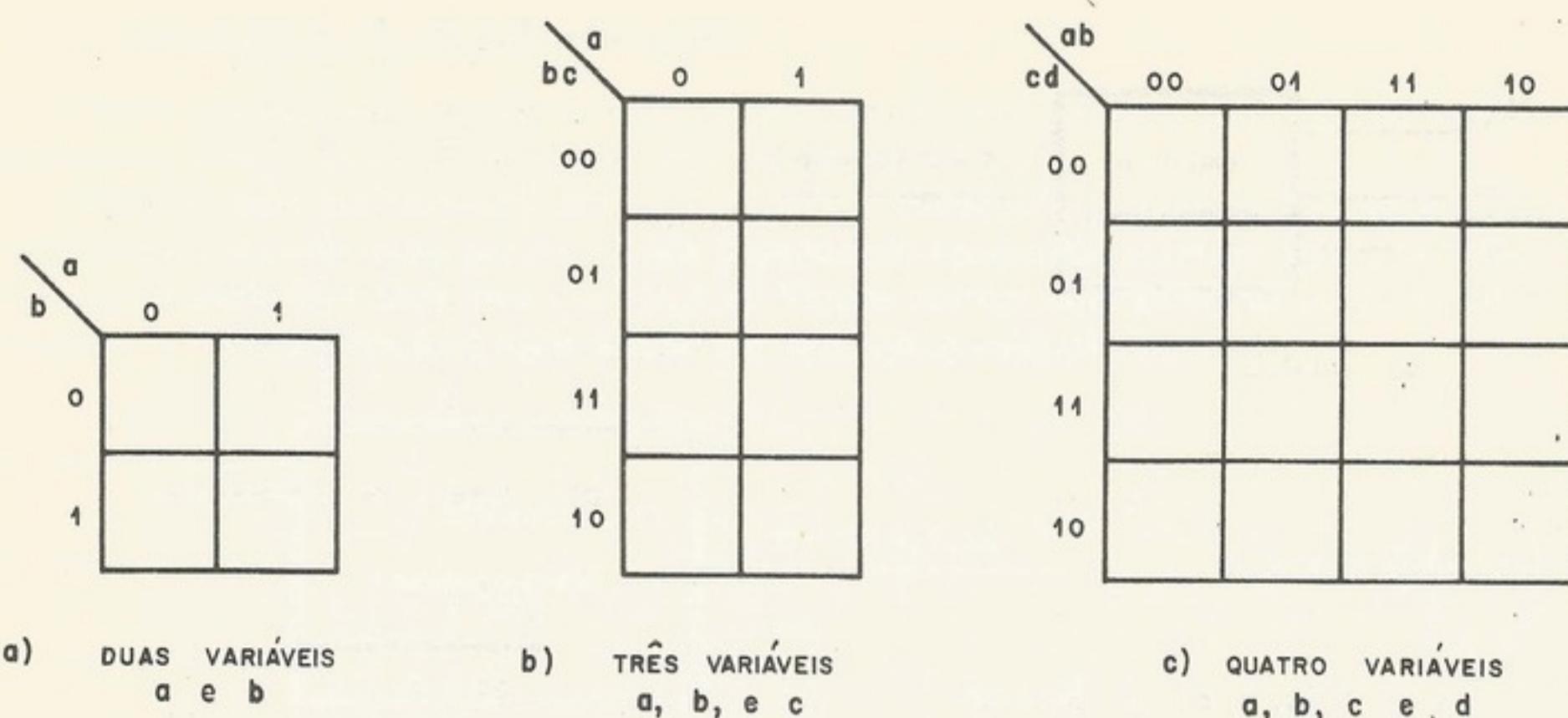


Figura 1-11. Mapas de Karnaugh

DEFINIÇÃO. Produto fundamental

Dada uma tabela da verdade, a cada combinação de valores das entradas, associamos um produto (*AND*) das variáveis de entrada, complementadas se o correspondente valor for “0” e não-complementadas se for “1”. Esse produto é chamado de produto fundamental.

EXEMPLOS

$$\begin{array}{l} \text{entrada} \quad | \begin{array}{ccc} a & b & c \\ 0 & 1 & 1 \end{array} \\ \text{produto fundamental correspondente, } a' \cdot b \cdot c; \end{array}$$

$$\begin{array}{l} \text{entrada} \quad | \begin{array}{cccc} a & b & c & d \\ 1 & 0 & 0 & 1 \end{array} \\ \text{produto fundamental correspondente, } a \cdot b' \cdot c' \cdot d. \end{array}$$

DEFINIÇÃO. Soma canônica⁽⁹⁾

Dada uma tabela de verdade, a descrição equivalente por uma expressão booleana é dada pela soma dos produtos fundamentais correspondentes a cada combinação das entradas cuja saída seja “1”.

EXEMPLO

Seja o circuito dado pela seguinte tabela da verdade:

a	b	c	s
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

A descrição pela expressão booleana é dada pela soma canônica, que é igual à soma de cinco produtos fundamentais:

$$s = a' \cdot b' \cdot c + a' \cdot b \cdot c' + a \cdot b' \cdot c + a \cdot b \cdot c' + a \cdot b \cdot c.$$

[Observação. Existe um método⁽⁹⁾ dual que nos leva a um produto de somas, cada soma correspondendo a uma linha com saída zero na tabela da verdade.]

c. Análise de circuitos combinatórios

Dado um circuito combinatório através de seu diagrama lógico, analisá-lo é encontrar a sua descrição por uma expressão booleana.

Isso é feito escrevendo as exp do circuito. Veja

d. Síntese de

O problema difícil. Como o a e⁽⁹⁾ da Babilônia a tecnologia, bus

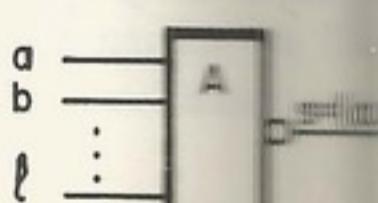
Custo (1). Embarato aquele que

Custo (2). Em aquele que tiver

É claro que classe. Um bloco Antes de que são os minis tecnologia atua

NAND

É um bloco inversor na saída



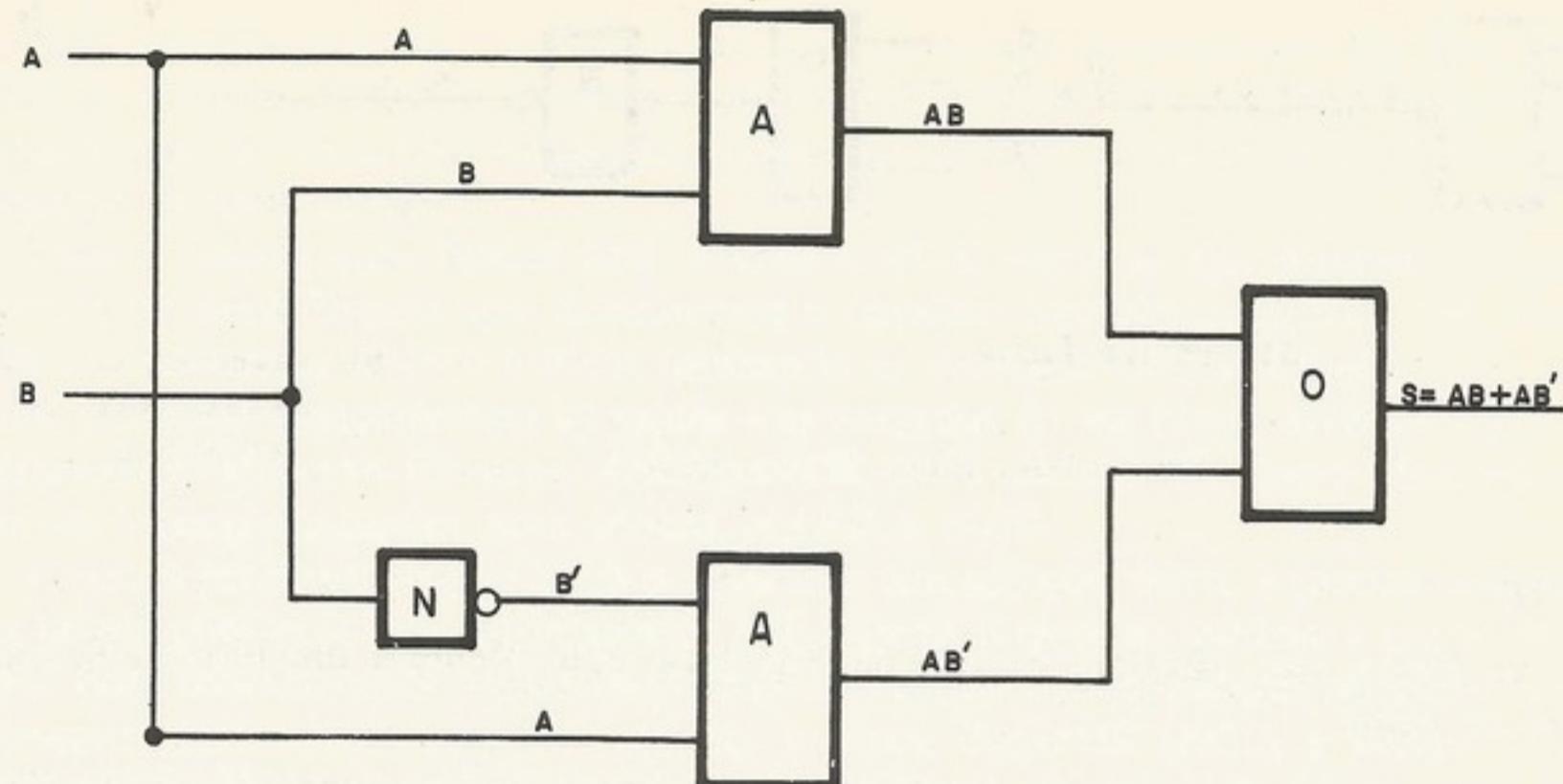


Figura 1-12. Exemplo de análise de um circuito combinatório

Isso é feito de um modo simples e metódico: basta partirmos das entradas e irmos escrevendo as expressões das saídas dos blocos lógicos até chegarmos à expressão de saída do circuito. Veja um exemplo na Fig. 1-12.

d. Síntese de circuitos combinatórios

O problema da síntese de circuitos combinatórios é um problema muito delicado e difícil. Como o abordaremos com superficialidade, recomendamos aos leitores os itens⁽⁷⁾ e⁽⁹⁾ da Bibliografia. Esse problema envolve uma variável muito delicada e imprevisível, a *tecnologia*, buscando um ótimo que envolve o conceito de *custo*. Vamos definir custo.

Custo (1). Entre dois circuitos com diferentes números de blocos lógicos, será mais barato aquele que tiver o menor número de blocos lógicos.

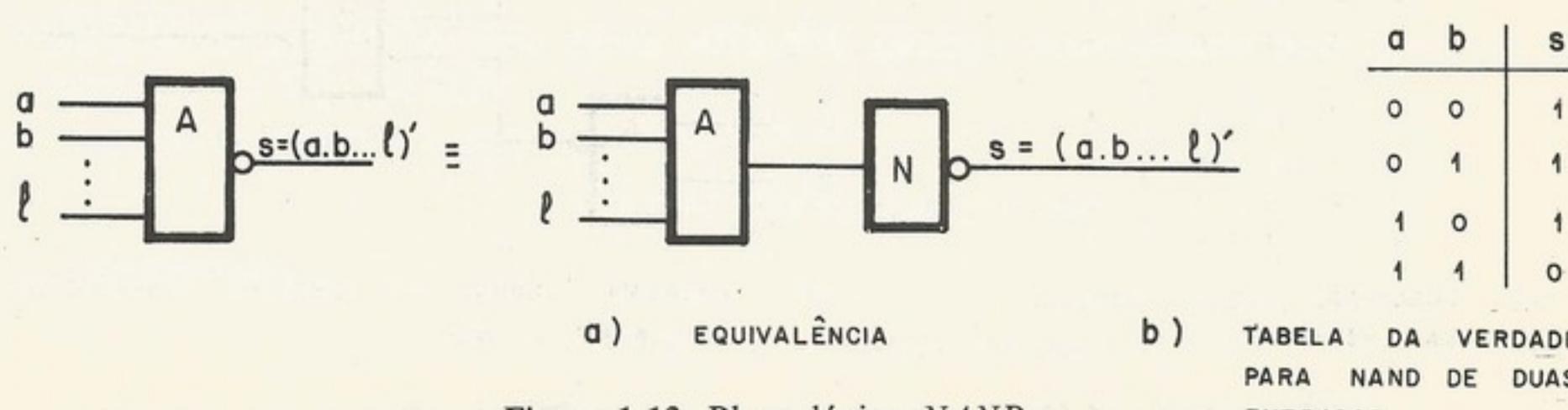
Custo (2). Entre dois circuitos de mesmo número de blocos lógicos, será mais barato aquele que tiver menor número total de entradas dos blocos.

É claro que esse conceito de custo exige a comparação de blocos lógicos de mesma classe. Um bloco *XOR* custa mais caro que um bloco *AND*.

Antes de tratarmos do problema da síntese, vamos definir dois novos blocos lógicos, que são os mais usados em circuitos lógicos, dadas as suas vantagens de custo e atraso na tecnologia atual: os blocos *NAND* e *NOR*.

NAND

É um bloco, cujo símbolo é visto na Fig. 1-13, equivalente a um bloco *AND* com um inversor na saída.

Figura 1-13. Bloco lógico *NAND*

a) EQUIVALÊNCIA
b) TABELA DA VERDADE
PARA NAND DE DUAS ENTRADAS

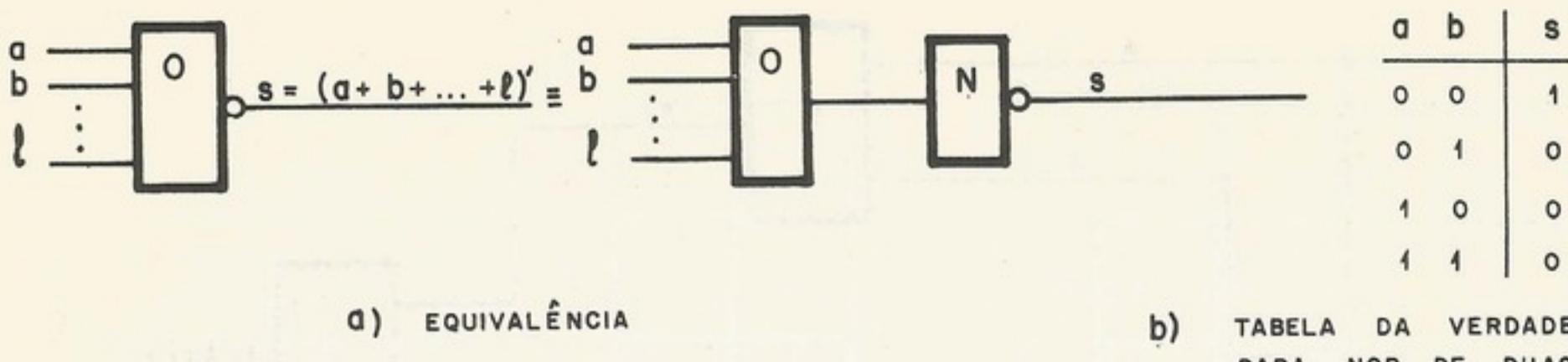


Figura 1-14. Bloco lógico NOR

NOR

É um bloco, cujo símbolo é visto na Fig. 1-14, equivalente a um bloco *OR* com um inversor na saída.

Uma técnica de síntese muito usada, pela sua simplicidade é aquela já citada: parte-se de sua expressão booleana e, utilizando-se os teoremas, busca-se uma expressão mais simples, que é sintetizada pelo processo inverso ao de análise.

EXEMPLO

Sintetizar o circuito dado pela tabela da verdade da Fig. 1-15(a).

Numa primeira etapa, escrevemos a sua expressão booleana:

$$s = abcd + abcd' + abc'd + abc'd' + ab'cd + ab'cd'.$$

A seguir, tentamos obter uma expressão mais simples cuja síntese deverá ser mais barata. Então, simplificando

$$\begin{aligned} s &= abc + abc' + ab'c, \\ s &= ab + ab'c, \\ s &= a(b + b'c), \\ s &= a(b + c). \end{aligned}$$

O circuito sintetizado com a expressão mais simples é visto na Fig. 1-15(b).

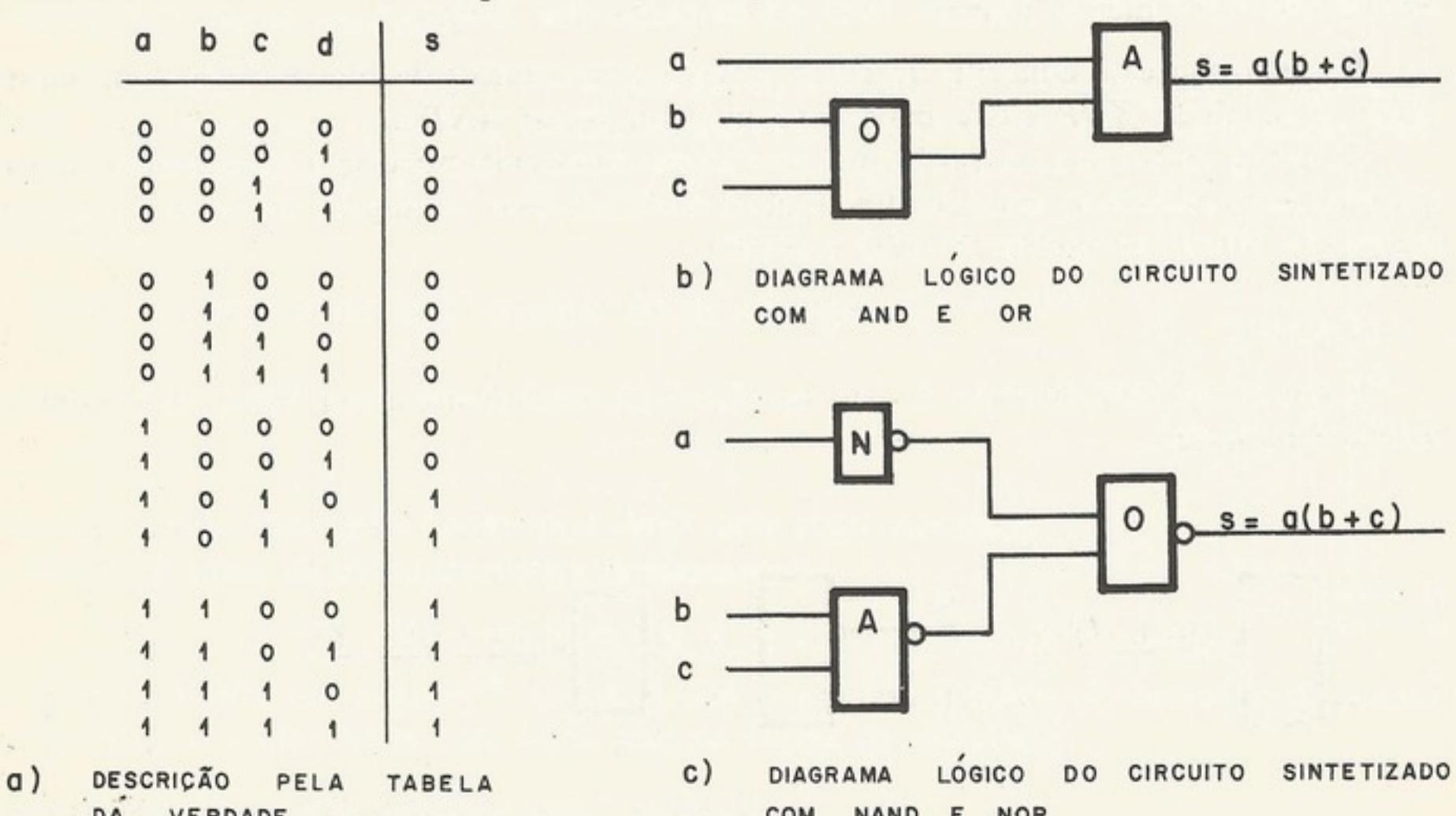


Figura 1-15. Circuito detetor de código BCD inválido

introdução e ...

Esse método é ...
mente ao mínimo ...
caminho seguido ...
realmente, o ma ...

Para a ...
diferença no f ...
a transforma ...
remas, podemos ...
invertamos a ...
cuito do exempi ...

Como o ...
uma tabela p ...
crição pela tra ...
economicamente ...
uma tabela am ...

Um caso pa ...
NAND-NAND, ...
circuito combin ...
mentos ativos ...
reside na veloci ...
circuito combin ...
DOT-OR (veja ...

Como o ...
expressão num ...
barata, que ch ...
para encontrar ...

e. Determina ...
Determinação d ...

Dado o ...
soma canônica ...

e, eventualmen ...

simplificamos a ...
Aqui contém ...
irreduzível, num ...
ga-se a soma c ...

EXEMPLO 1

Projetar o ...
binações da Fig ...

Vamos co ...

Simplificando ...

O circuito minim ...

a	b	s
0	0	1
0	1	0
1	0	0
1	1	0

VERDADE
MÍNOR DE DUAS
TRUTHS

OR com um

parte-se
mais sim-

mais barata.

$s = a(b + c)$

SINTETIZADO

$s = a(b + c)$

SINTETIZADO

Esse método, que exige muita arte do projetista, é um processo que não leva necessariamente ao mínimo, podendo-se obter expressões irredutíveis de alto custo, dependendo do caminho seguido na simplificação. O problema, portanto, é saber se o circuito obtido é, realmente, o mais barato.

Para a síntese com blocos tipo *NAND* e *NOR*, segue-se um processo idêntico, com diferença no final. Depois que desenhamos o circuito com blocos *AND* e *OR*, começamos a transformá-lo em *NAND* e *NOR* utilizando os teoremas de De Morgan. Com esses teoremas, podemos complementar as saídas e as entradas de um bloco *AND* ou *OR* desde que invertamos a operação (troca-se *AND* por *OR*, e vice-versa). Na Fig. 1-15(c) vemos o circuito do exemplo anterior com blocos *NAND* e *NOR*.

Como o problema com *NAND* e *NOR* não tem solução viável, é prática comum usar-se uma tabela publicada por Hellerman⁽¹¹⁾ que fornece o circuito mínimo a partir de sua descrição pela transformada numérica. Com a evolução da tecnologia, obteve-se um bloco, economicamente viável com duas saídas, o bloco *OR/NOR* (*ECL*), e Baugh⁽¹²⁾ publicou uma tabela análoga para circuitos sintetizados com esse tipo de bloco.

Um caso particular de síntese que tem solução é a síntese em dois níveis *AND-OR* ou *NAND-NAND*. Esse problema foi estudado principalmente na segunda geração, pois o circuito combinatório em dois níveis *AND-OR* [Fig. 1-16(a)] podia ser realizado sem elementos ativos [Fig. 1-16(b)], o que era um grande fator de economia. Sua importância atual reside na velocidade; com circuito *NAND-NAND* [Fig. 1-16(c)] implementa-se qualquer circuito combinatório com dois níveis de atraso e, quando implementado com a técnica *DOT-OR* (veja à frente), consegue-se um atraso de apenas um nível.

Como o leitor já deve ter observado, o circuito em dois níveis, se analisado, terá como expressão uma soma de produtos. Então o problema é obter uma soma de produtos mais barata, que chamaremos de *soma mínima*. Vamos tratar das duas maneiras mais usadas para encontrar-se a soma mínima: pela simplificação booleana e pelo mapa de Karnaugh.

e. Determinação da soma mínima

Determinação da soma mínima por simplificação booleana

Dado o circuito combinatório por qualquer uma das descrições, escreveremos a sua soma canônica. A seguir, utilizando os teoremas

$$(T.16) \quad A \cdot B + A \cdot B' = A,$$

$$(T.21) \quad A \cdot B + A' \cdot C + B \cdot C = A \cdot B + A' \cdot C$$

e, eventualmente, o teorema

$$(T.18) \quad A + A' \cdot B = A + B$$

simplificamos a soma canônica até chegarmos à soma mínima.

Aqui continua o mesmo problema anterior: o projetista, após chegar a uma soma irredutível, nunca sabe se é a mínima. Seguindo caminhos diferentes na simplificação, chega-se a somas irredutíveis diferentes.

EXEMPLO 1

Projetar o circuito, em dois níveis *AND-OR*, mínimo, caracterizado pela tabela de combinações da Fig. 1-17(a).

Vamos escrever a soma canônica

$$s = a'b'c' + a'b'c + ab'c + abc.$$

Simplificando,

$$s = a'b' + ac \quad (\text{soma mínima}).$$

O circuito mínimo, em dois níveis *AND-OR*, pode ser visto na Fig. 1-17(b).

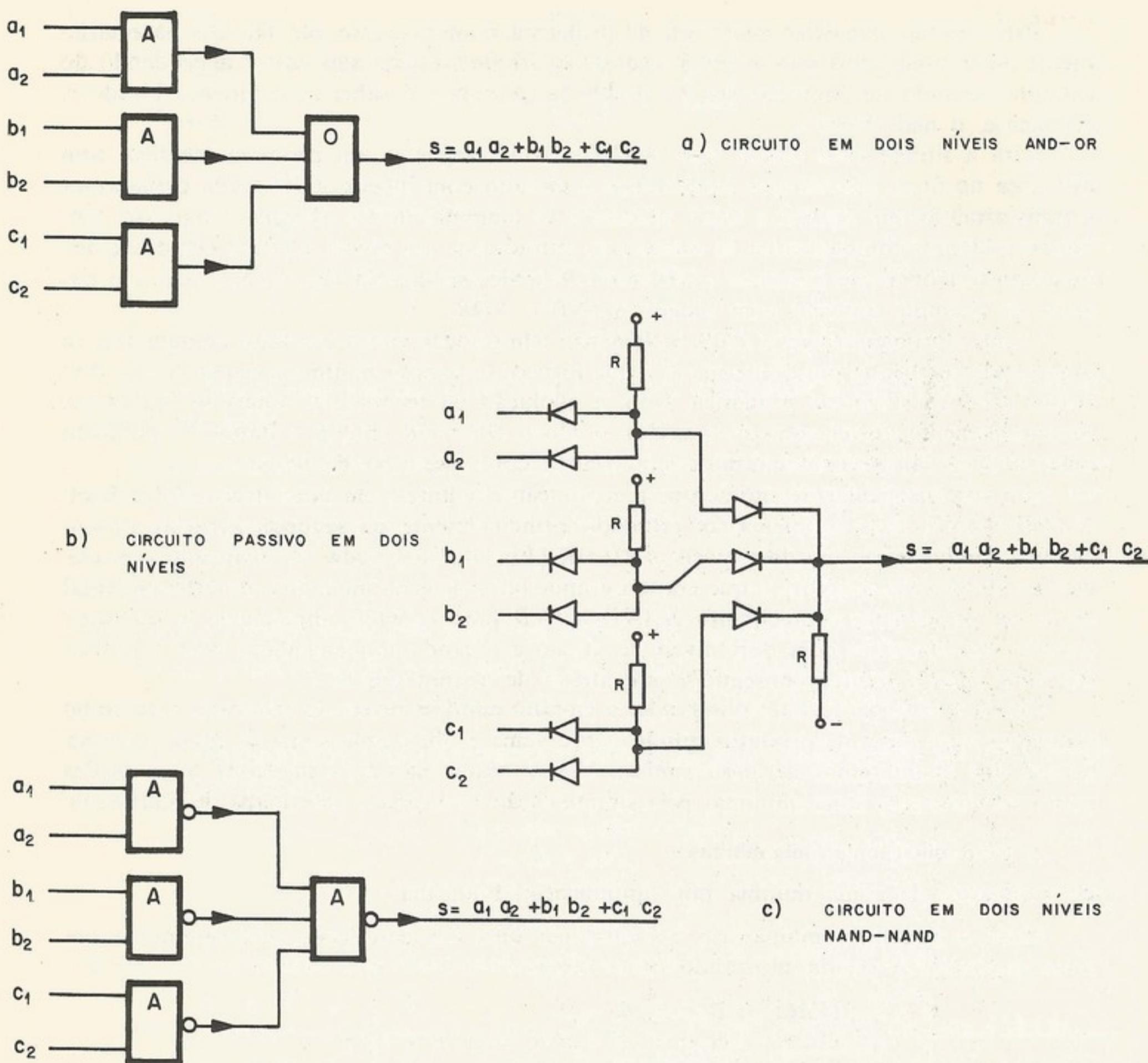


Figura 1-16. Várias implementações do mesmo circuito em dois níveis

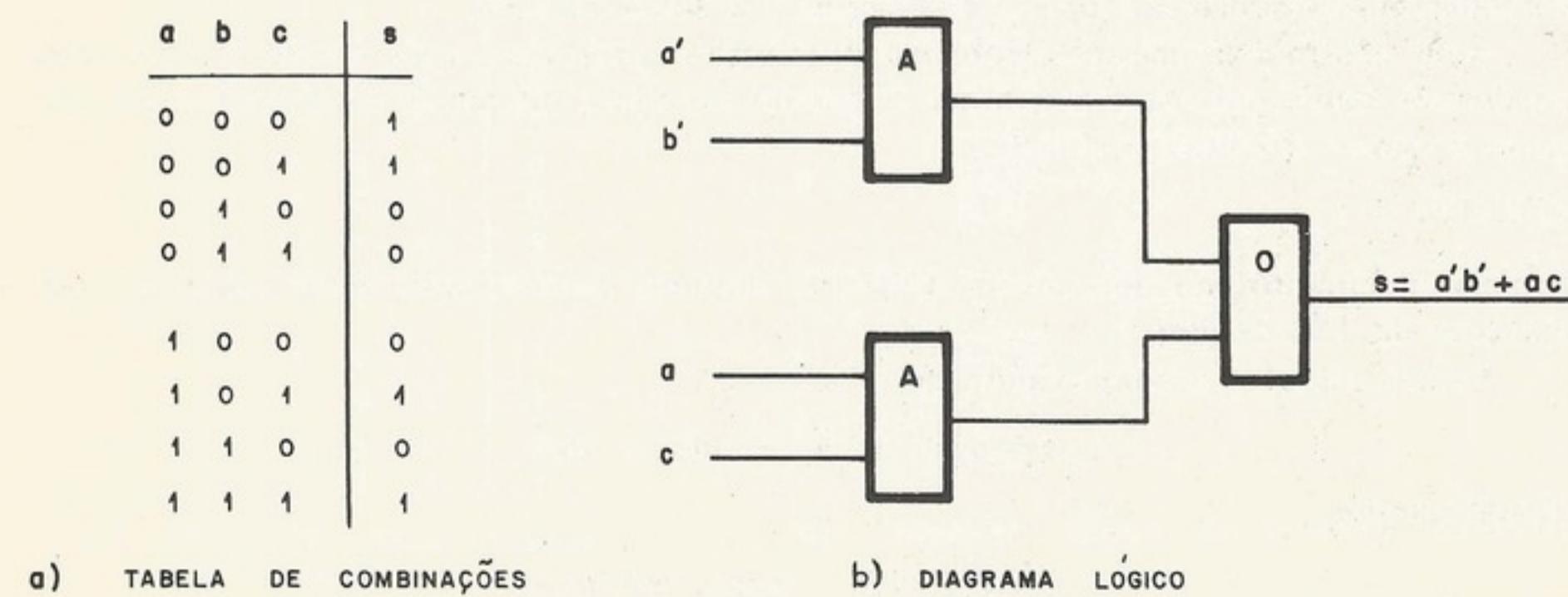


Figura 1-17. Circuito combinatório do Exemplo 1

introdução e

EXEMPLO 2

Vamos ilustrar diferentes.

Projetar o

Primeira

s = 0

s = 1

Segunda

O circuito

Determinação da

Vamos examinar as definições que

Olhando para os correspondentes a "1" pelo mapa

EXEMPLO 2

Vamos ilustrar o problema de caminhos diferentes levarem a somas irreduzíveis diferentes.

Projetar o circuito mínimo em dois níveis *NAND-NAND*, dado pela expressão booleana

$$s = a'b' + b'c' + ac + ab + bc'$$

Primeira maneira. Simplificando,

$$s = a'b' + \underbrace{b'c + ac + ab}_{(b'c + ab)} + bc'$$

$s = a'b' + b'c + ab + bc'$ (a soma irreduzível não é mínima).

Segunda maneira. Simplificando por outro caminho,

$$s = \underbrace{a'b' + b'c + ac}_{(a'b' + ac)} + ab + bc'$$

$$s = a'b' + \underbrace{ac + ab + bc'}_{(ac + bc')}$$

$s = a'b' + ac + bc'$ (a soma irreduzível é mínima).

O circuito mínimo é mostrado na Fig. 1-18.

Determinação da soma mínima pelo mapa de Karnaugh

Vamos escrever a soma mínima da função dada pelo mapa da Fig. 1-19, para ilustrar as definições que seguem. A soma canônica será

$$s = \underbrace{a'b'c'd' + a'b'c'd}_{(T.16) - a'b'c'} + \underbrace{abc'd + abcd}_{(T.16) - abd} + \underbrace{ab'c'd + ab'cd}_{(T.16) - ab'd}$$

$$s = a'b'c' + \underbrace{abd + ab'd}_{(T.16) - ad},$$

$$s = a'b'c' + ad.$$

Olhando para a expressão da soma mínima, vemos que os produtos fundamentais correspondentes a "1" vizinhos se agrupam pelo (T.16). A técnica de determinação da soma mínima pelo mapa irá explorar essa característica.

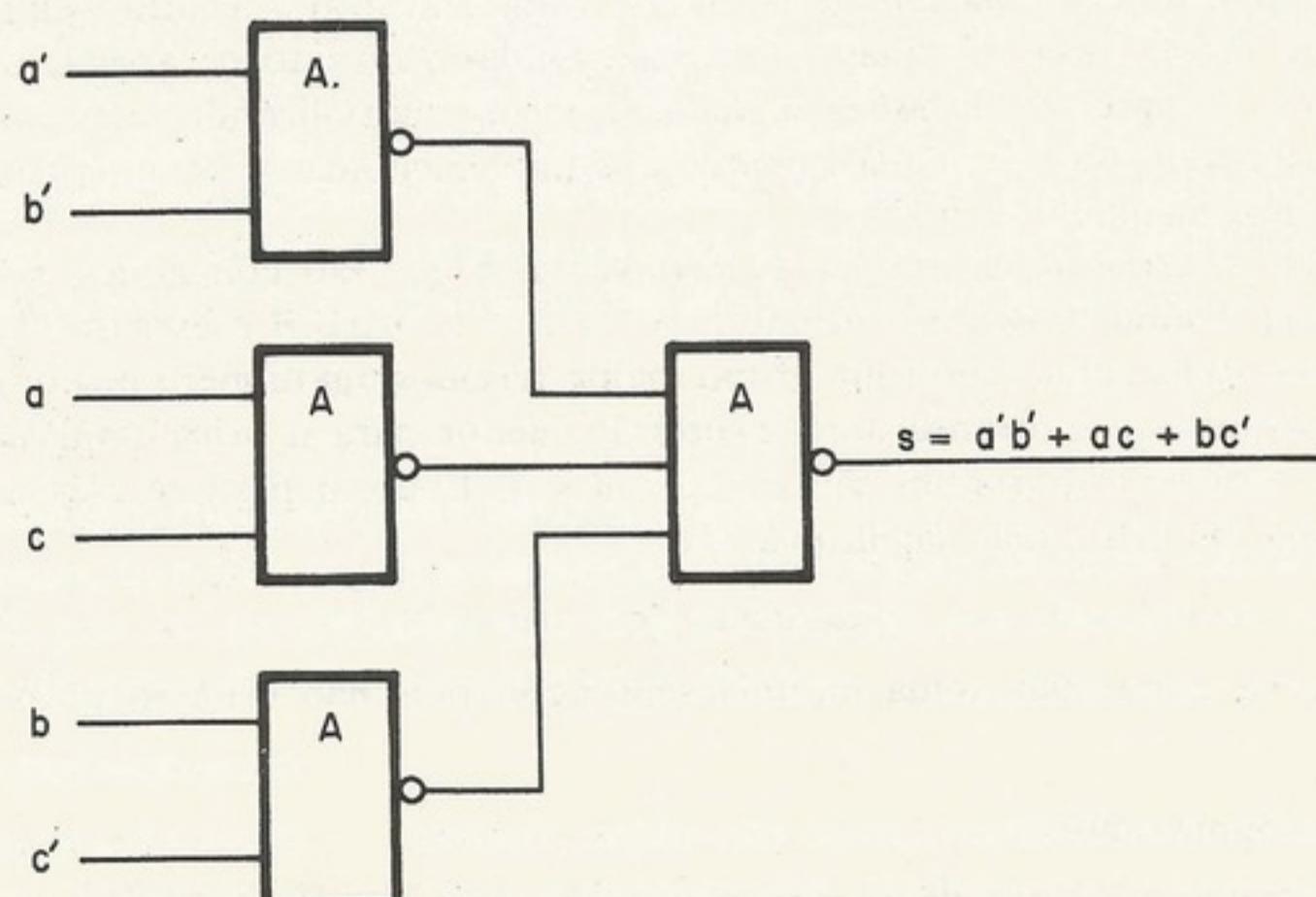


Figura 1-18. Circuito mínimo em dois níveis *NAND-NAND* do Exemplo 2

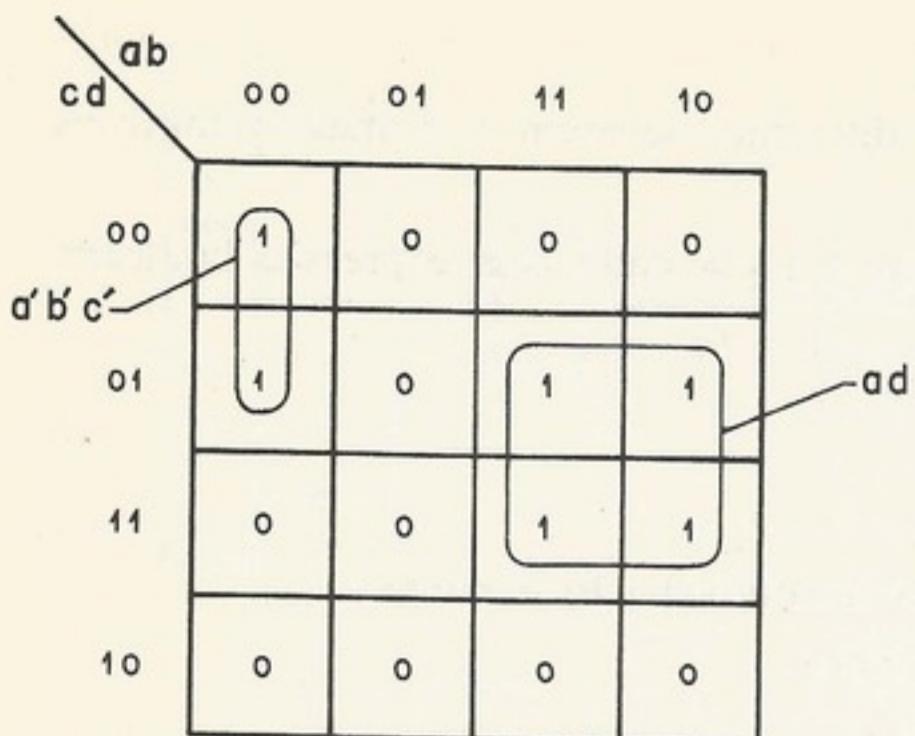


Figura 1-19. Prime implicant no mapa de Karnaugh

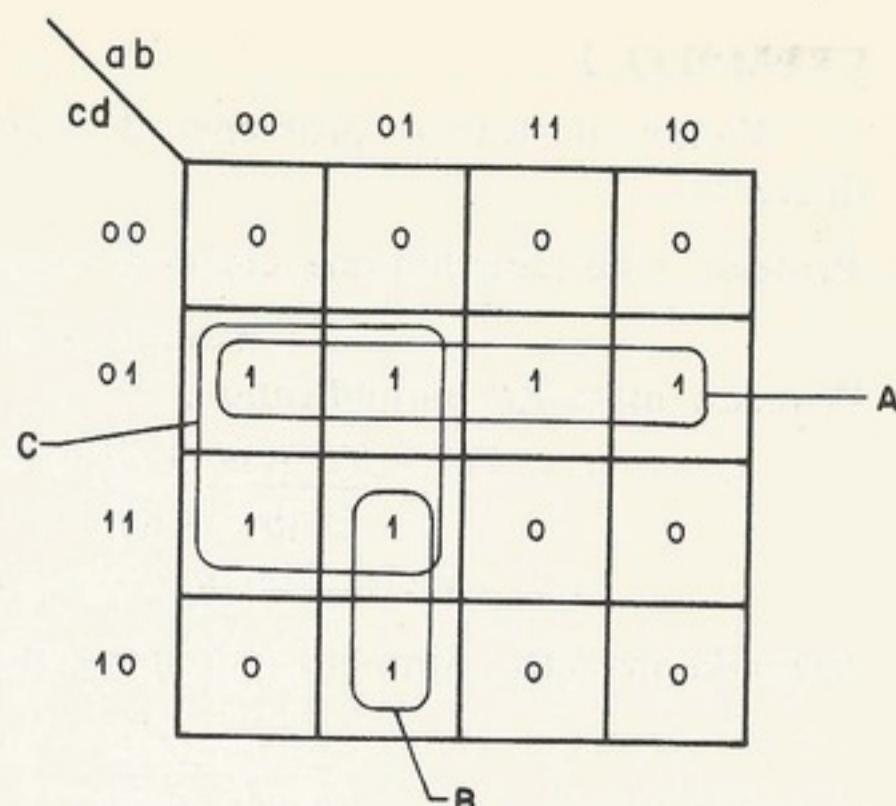


Figura 1-20. Mapa de Karnaugh do Exemplo 3

DEFINIÇÃO. Prime implicant

Dado um circuito combinatório descrito pelo mapa de Karnaugh, agrupamos as casas com "1" nos maiores grupos retangulares de tamanho 2ⁿ. A cada grupo desses associamos um produto das variáveis (complementadas ou não) que pode ser obtido de soma canônica de todos os "1" do grupo, simplificados pelo (T.16). Esse produto é chamado de *prime implicant* (*p.i.*) da função. No exemplo anterior temos dois *prime implicants* (observe que está assinalado na Fig. 1-19):

$$a'b'c' \text{ e } ad$$

EXEMPLO 3

No mapa de Karnaugh da Fig. 1-20, indicamos três grupos retangulares com 2ⁿ células preenchidas com "1". Portanto essa função tem três *prime implicants*, que escreveremos a seguir.

$$p.i. A = a'b'c'd + a'bc'd + abc'd + ab'c'd = a'c'd + ac'd; \text{ portanto } p.i. A = c'd.$$

Na determinação dos *prime implicants* *B* e *C*, vamos mostrar a técnica usual, escrevendo diretamente do mapa; casas vizinhas são caracterizadas pelo fato de apenas uma variável mudar. Portanto a expressão do *prime implicant* é obtida multiplicando-se as variáveis constantes para todas as casas circuladas. (Variável complementada se o valor da entrada for "0", e não-complementada se for "1").

p.i. B. Para as casas desse grupo, as variáveis *a*, *b* e *c* são constantes, pois apenas a variável *d* muda. Como nesse caso *a* = 0, *b* = 1 e *c* = 1, o *p.i. B* é escrito *a'bc*.

p.i. C. Nesse caso, como temos um grupo maior, teremos um número menor de variáveis de entrada constantes e, portanto, uma expressão menor para o *prime implicant*.

Então, as variáveis constantes são *a* = 0 e *d* = 1. Logo, o *p.i. C* é escrito *a'd*. Concluindo, os três *prime implicants* são

$$c'd, a'bc \text{ e } a'd.$$

Vamos agora conceituar soma mínima sem rigor, pois não é nosso propósito entrar em detalhes.

DEFINIÇÃO. Soma mínima

A soma mínima é a soma do menor número de *prime implicants* que cobre completamente a função. Admitimos como intuitivo o conceito de "cobrir". Então a técnica de deter-

minação da soma mínima consiste em marcar todos os *prime-implicants* e, a seguir, escolher o conjunto mínimo. É claro que se começa separando os essenciais (aqueles que cobrem *sozinhos* uma casa com "1") para, depois, escolherem-se os não-essenciais, levando-se em conta o conceito de custo.

EXEMPLO 4

"Implementar em dois níveis o circuito dado pelo mapa de Karnaugh da Fig. 1-21(a)."

Existem três *prime implicants* A, B e C que estão marcados no mapa, dos quais A e C são essenciais, pois cobrem uma casa (marcada com *) *sozinhos*. Como os dois são suficientes para cobrir toda a função, a soma mínima é dada pela soma deles:

$$s = p.i. A + p.i. C = ab' + a'c$$

O circuito mínimo pode ser visto na Fig. 1-21(b).

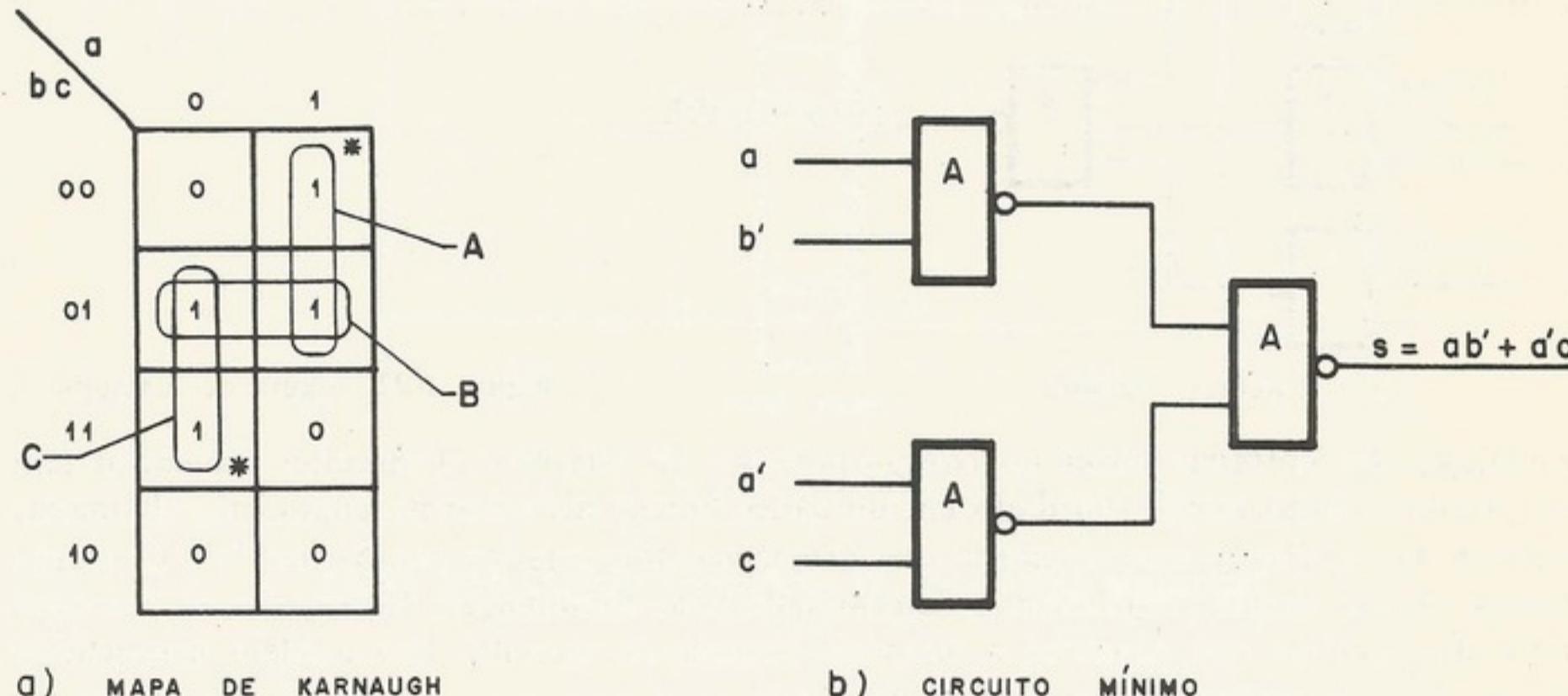


Figura 1-21. Figura do Exemplo 4

EXEMPLO 5

Implementar em dois níveis o circuito dado pelo mapa de Karnaugh da Fig. 1-22(a).

Marcados os *prime implicants*, vemos que apenas os A e E são essenciais, ou seja, necessariamente estarão presentes na soma mínima. Separando-os, para facilitar a escolha dos não-essenciais, obtemos o mapa visto na Fig. 1-22(b) (na escolha dos não-essenciais, deve-se levar em conta que um *prime implicant* envolvendo uma quantidade maior de "1" é mais barato, pois sua expressão é mais simples). Vemos na Fig. 1-22(b) que falta apenas cobrirmos dois "1" e a escolha mais barata é o *p.i. C*, pois ele cobre as duas casas que faltam. Então a soma mínima é dada pela soma dos *prime implicants* essenciais (A e E) e o *p.i. C*,

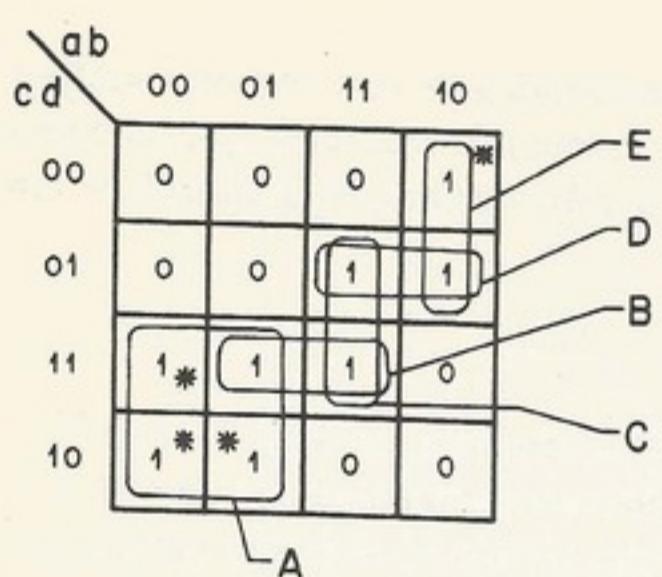
$$\begin{aligned} s &= p.i. A + p.i. E + p.i. C, \\ s &= a'c + ab'c' + abd \quad (\text{soma mínima}). \end{aligned}$$

O circuito mínimo pode ser visto na Fig. 1-22(c).

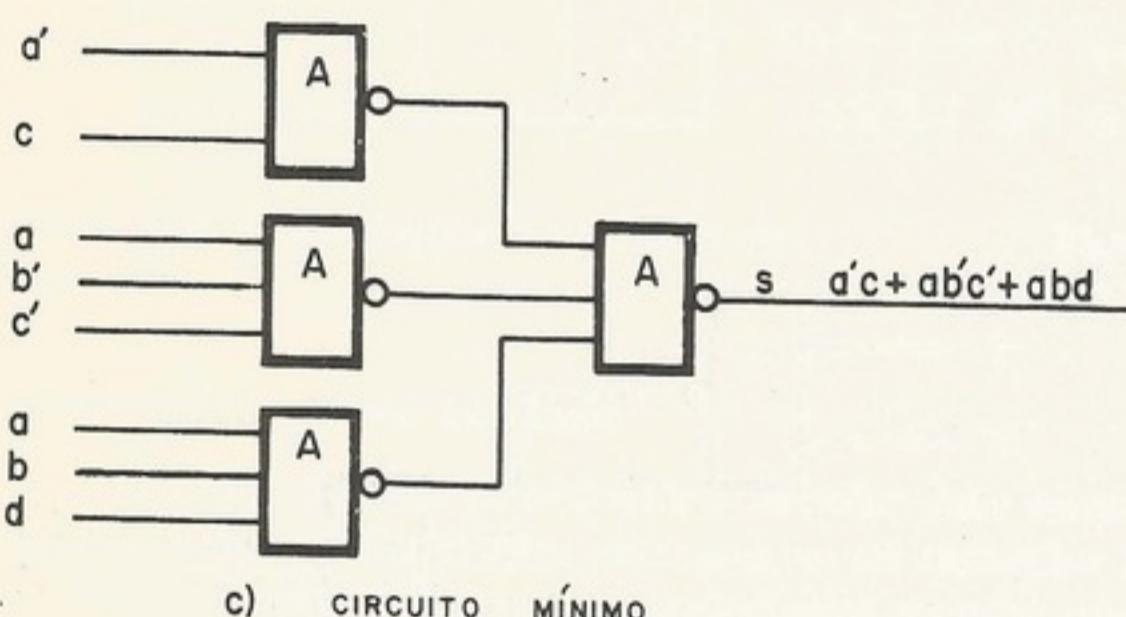
Acreditamos que essa segunda técnica de determinação da soma mínima seja a mais recomendada, por dar ao projetista uma boa visualização das alternativas, permitindo-lhe assegurar-se de que a escolha é realmente a melhor.

f. Circuitos seqüenciais

Se o circuito lógico tiver memória, então não bastará saber-se o estado das entradas atuais para se determinarem as saídas; será necessário, também, conhecer-se o *estado interno* do circuito. Circuitos desse tipo são chamados *circuitos seqüenciais*, pois a seqüência das

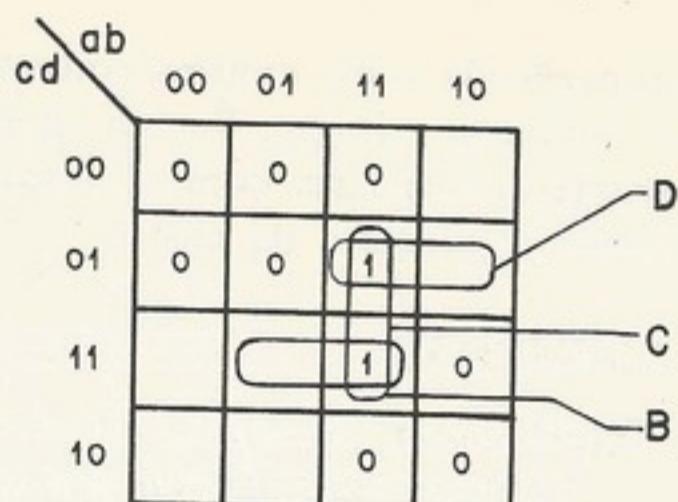


a) MAPA DE KARNAUGH



c) CIRCUITO MÍNIMO

projeto de computadores digitais



b) MAPA DE KARNAUGH SEM OS "PRIME IMPLICANTS" ESSENCIAIS

mudanças das entradas influem no comportamento do circuito. O circuito seqüencial tem um estado interno que é guardado em unidades funcionais de armazenamento, chamadas *flip-flop*. O *flip-flop* é uma memória que armazena dois estados, ou “0” ou “1”. O estado interno de um circuito seqüencial é determinado pela combinação dos estados de um conjunto de *flip-flops* que constitui a memória interna do circuito. Um modelo matemático de um circuito seqüencial aparece na Fig. 1-23.

É importante observar que o circuito seqüencial tem realimentação, isto é, parte das saídas são reinjetadas na entrada. O circuito combinatório da Fig. 1-23 tem dois conjuntos de saídas, o conjunto (Z_1, Z_2, \dots, Z_m) , que são as saídas do circuito seqüencial, e o conjunto (y_1, y_2, \dots, y_p) , que determina o estado seguinte do sistema (Y_1, Y_2, \dots, Y_p) . Então podemos dizer, em outras palavras, que o circuito seqüencial é caracterizado por uma memória que guarda o estado do sistema, e por um circuito combinatório cujas entradas são as entradas primárias do circuito seqüencial e o estado interno corrente. O circuito seqüencial fornece como saídas a saída do circuito combinatório e o estado seguinte do sistema. O conceito “seguinte” nesse caso, está relacionado com os atrasos intrínsecos dos blocos lógicos, mas o leitor, para melhor entender o funcionamento dos circuitos seqüenciais, pode imaginar como um tempo finito e determinado. Portanto vemos que, para conhecermos a saída de um circuito seqüencial, precisamos conhecer o valor das entradas e o estado interno, e não simplesmente as entradas, como era o caso dos circuitos combinatórios.

Existe uma classe de circuitos seqüenciais chamada “varredura finita” cuja saída poderá ser prevista, mesmo desconhecendo-se o valor do estado interno, se soubermos os n últimos valores das entradas. Podemos dizer, então, que esses n últimos valores das entradas determinam o estado atual.

g. Descrição de um circuito seqüencial

Como já vimos, um circuito seqüencial ficará definido se soubermos, para cada combinação dos valores do estado interno e da entrada, qual será a saída e o estado seguinte. As duas descrições que veremos a seguir têm essa característica. Elas são uma espécie de

tabela onde, dadas as entradas e o estado seguem, exprimem o resultado.

Descrição por tabelas.

O diagrama de bloco, fazemos com que o resultado seja determinado pelo valor das entradas.

A correspondência entre as entradas e o resultado é dada por tabelas. Dado um certo número de entradas a, b, c e, sabendo que elas determinam o resultado s , podemos obter a correspondência.

Por exemplo, se temos $(X_1 = 0, X_2 = 1, X_3 = 0)$ e a entrada “01” é a mesma que “10”, é o mesmo e a saída é a mesma. Seja, a nova saída de a com $b = 1$ é uma parte da saída. O estado seguinte do circuito se manteve.

Observe, portanto,

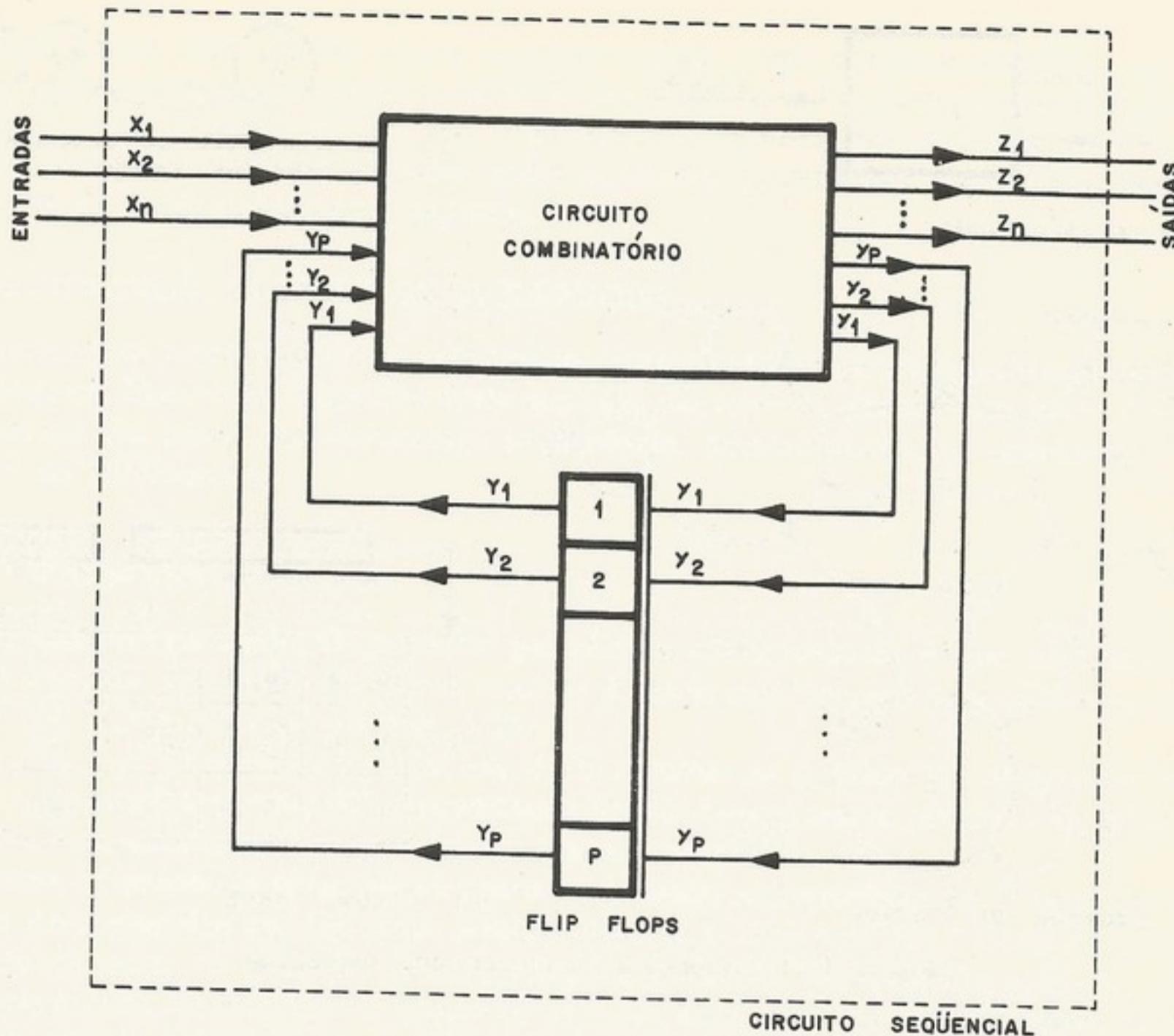


Figura 1-23. Modelo matemático de um circuito seqüencial

tabela onde, dado um certo estado e o valor da entrada, são fornecidos os valores da saída e do estado seguinte. Em síntese, podemos dizer que as descrições dos circuitos seqüenciais exprimem o funcionamento do circuito combinatório que faz parte dele.

Descrição pelo diagrama de estado

O diagrama de estado é um gráfico com tantos nós quantos forem os estados e, a cada nó, fazemos corresponder um estado. Ligando-se os nós, existem ramos que são determinados pelo valor das entradas. Esses ramos mostram o estado seguinte do circuito com aquela determinada entrada e, sobre o ramo, marcamos o correspondente valor da saída.

A correspondente descrição pelo diagrama de estado pode ser vista na Fig. 1-24(b). Dado um certo estado interno, situamos um dos três nós possíveis (existem três estados, *a*, *b*, e *c*) e, sabendo qual a entrada, localizamos um dos possíveis ramos que saem do nó (pelo primeiro número dos escritos no ramo). O final do ramo vai para o estado seguinte e a correspondente saída está indicada no segundo número escrito no ramo (depois de vírgula).

Por exemplo, vamos supor que estejamos no estado *a* e que a entrada seja “01” ($X_1 = 0$, $X_2 = 1$); como vemos na Fig. 1-24(b) (em negrito), o ramo que sai do nó *a* com a entrada “01” volta para *a* e a saída é “1”. Portanto, com essa entrada, o estado seguinte é o mesmo e a saída permanece igual a “1”. Admitamos agora que X_1 mude para “1”, ou seja, a nova entrada passe a ser “11” (maior que a anterior). Então identificamos o ramo que sai de *a* com entrada “11”. Ele indica que o estado seguinte será *c* (veja, na Fig. 1-24(c), que é uma parte da Fig. 1-24(b), ou seja, se estivermos no estado *a* e a entrada passa a ser “11” o estado seguinte será *c* e a saída será “1”). Depois disso, se mantivermos a entrada “11”, o circuito se manterá no estado *c* com a saída “1”.

Observe, portanto, que o diagrama de estado descreve completamente o circuito.

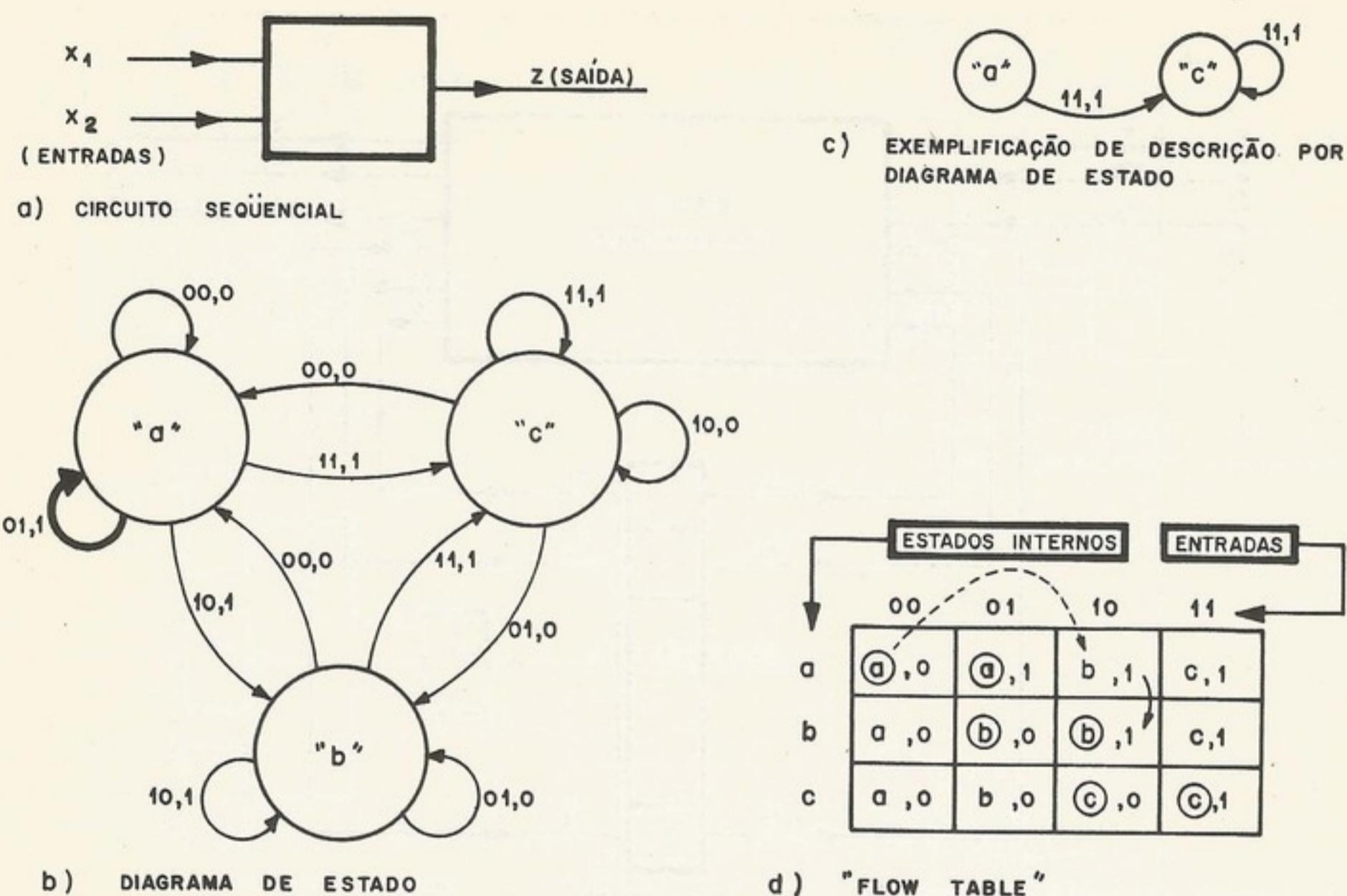


Figura 1-24. Descrições de um circuito seqüencial

Descrição pela "flow table"

O comportamento de um circuito seqüencial pode ser descrito por uma *flow table* (tabela de estados), uma tabela na qual as colunas são os valores das entradas e as linhas são os estados internos. Cada lugar da tabela corresponde à combinação dos dois componentes, a entrada e o estado interno. A essa combinação, daremos o nome de estado total do circuito seqüencial. A tabela é preenchida colocando-se o estado seguinte e a saída no lugar correspondente a cada estado total da tabela. Cada lugar na tabela, então, tem dois componentes, o estado interno seguinte e o valor da saída.

A *flow table* do circuito da Fig. 1-24(a) aparece na Fig. 1-24(d). O primeiro componente na tabela é o estado interno (a , b e c) e o segundo componente é a saída, ou "0" ou "1". O circuito do exemplo tem três estados internos. Então a *flow table* tem três linhas. Começando na primeira linha e primeira coluna, o estado total é $(00, a)$ significando que o valor da entrada é "00" (isto é, $X_1 = 0$, $X_2 = 0$) e o estado interno é " a ". A saída é "0". Supondo-se que a entrada mude de "00" para "10" (isto é, o sinal X_1 mude para "1"), o estado total será $(10, a)$ e notamos, na tabela da Fig. 1-24(d), que o estado seguinte será b e não a . Quando o estado interno seguinte não for igual ao atual, o estado total será chamado de *transitório* ou *instável*. Isso porque o circuito irá mudar de estado interno. Quando o estado interno mudar de a para b , estaremos no estado total $(10, b)$. O próximo estado interno para esse estado total é b , igual ao atual. Nesse caso, o estado total é chamado de *estável*. É costume, na *flow table*, indicar os estados estáveis por uma bolinha no próximo estado interno.

O comportamento do circuito, para esse exemplo está indicado na Fig. 1-24(d) pelas setas tracejadas. As setas indicam o movimento do chamado *ponto de operação (operating point)* do circuito; começando em $(00, a)$ com saída "0" ele muda para $(10, b)$, que é estável. A saída do circuito, que estava no estado "0", agora está no estado "1". Em termos gerais, uma mudança de entradas é um movimento horizontal do ponto de operação na *flow table*. Esse movimento horizontal provoca um movimento vertical quando o ponto de operação cai num estado instável.

introdução e ...

Nesse tópico, ...
table ou pelo diagrama de circuito, nem circuitos sequenciais (7), (9) e (23) da Bi...

1.4 TECNOLOGIA

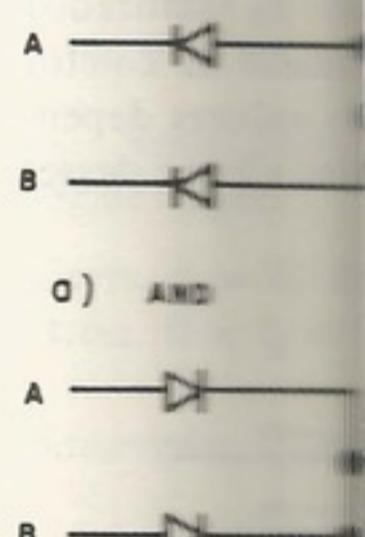
Estudaremos blocos lógicos. ...
dos assuntos. As (15) e (16) da Bi...

a. Circuitos

Um dispositivo usado, é o diodo AND e OR, com ...

Na Fig. 1-22, a alta (mais possivelmente) verificar o que ...

Sempre há possibilidade de circuitos AND se tornarem indefinidamente instáveis. ...



b) ...

Nesse último item não foi nossa intenção indicar *como* chegar à descrição pela *flow table* ou pelo diagrama de estados, a partir da descrição verbal do comportamento desejado do circuito, nem a partir do circuito. Não tratamos também do problema da síntese de circuitos seqüenciais. Maiores detalhes sobre o assunto poderão ser encontrados nos itens ⁽⁷⁾, ⁽⁹⁾ e ⁽²³⁾ da Bibliografia deste capítulo.

1.4 TECNOLOGIA: TRANSISTORES E DÍODOS

Estudaremos nesta seção e na próxima as técnicas usadas na implementação física dos blocos lógicos. Mantendo o espírito deste capítulo, ou seja dar apenas uma breve descrição dos assuntos. Ao leitor que desconheça o problema, recomendamos as referências^{(13), (14), (15) e (16)} da Bibliografia.

a. Circuitos lógicos com díodos

Um dispositivo usado na primeira geração de computadores, e que continua sendo usado, é o díodo. Junto com resistores, o díodo é usado para implementar os blocos lógicos *AND* e *OR*, como indicado na Fig. 1-25.

Na Fig. 1-25, o circuito (a) será um *AND* somente se o valor “1” representar tensão alta (mais positiva) e o valor “0” representar a tensão baixa (menos positiva). O leitor pode verificar o que aconteceria se os códigos fossem invertidos.

Sempre há perdas de níveis em circuitos como esses, sem amplificação. É comum termos circuitos *AND* seguidos por *OR* como indicado na Fig. 1-25(c), mas não é viável encadearem-se indefinidamente os *gates AND* e *OR* sem amplificação. Na primeira geração, os circuitos com díodos, sintetizando os blocos *AND* e *OR*, eram seguidos de um inversor montado com uma válvula a vácuo para amplificar os sinais. Atualmente, a função de amplificação e inversão é feita com transistores.

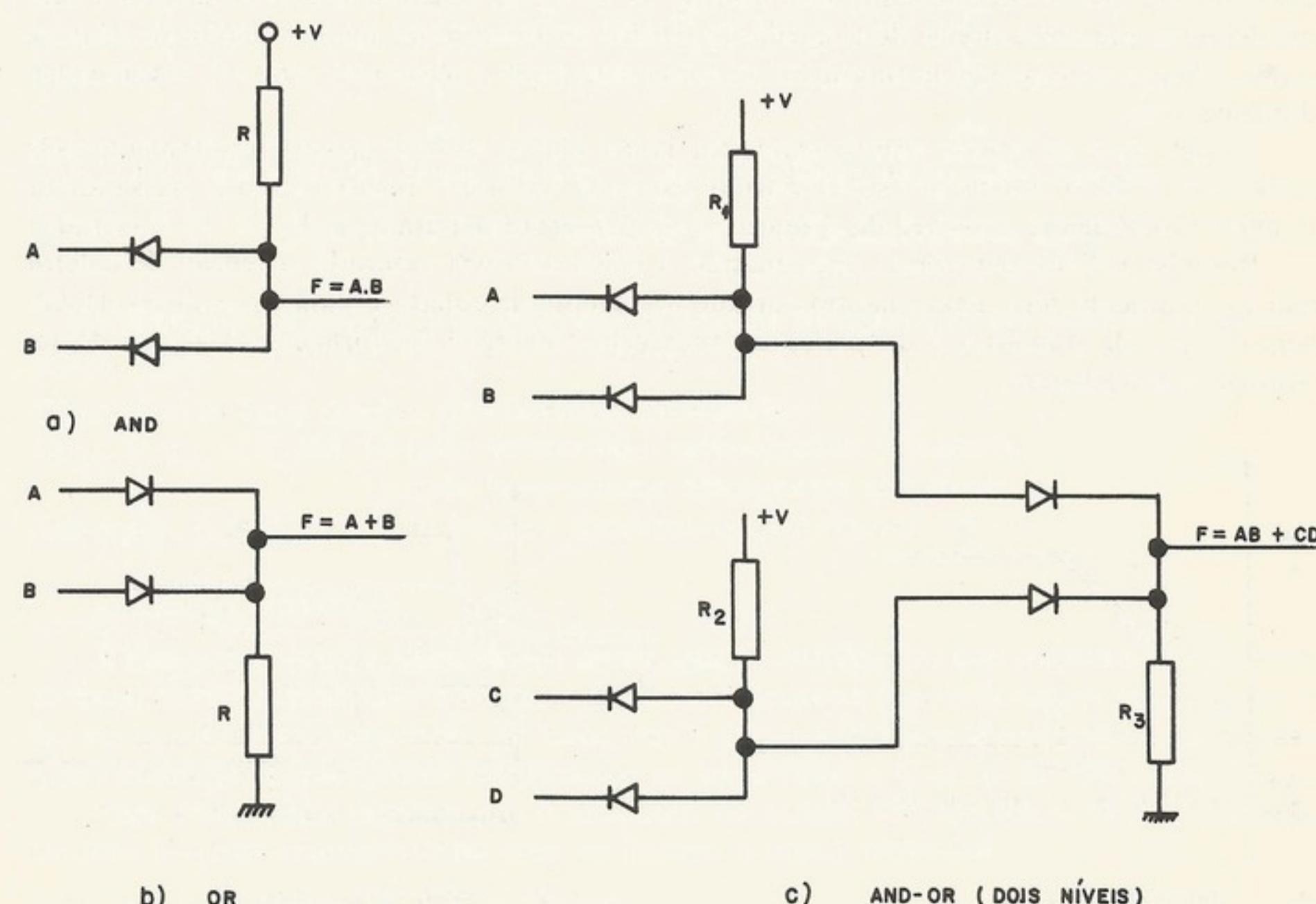


Figura 1-25. Circuitos lógicos com díodos

b. O transistor como chave

Estudo do regime

O transistor pode ser usado como chave, conforme visto na Fig. 1-26. Quando injetamos uma corrente, na base do transistor, suficientemente grande, a chave fecha, isto é, o ponto C é ligado para a terra. Quando a corrente de base é nula, a chave fica aberta, ou seja, o ponto C desligado da terra.

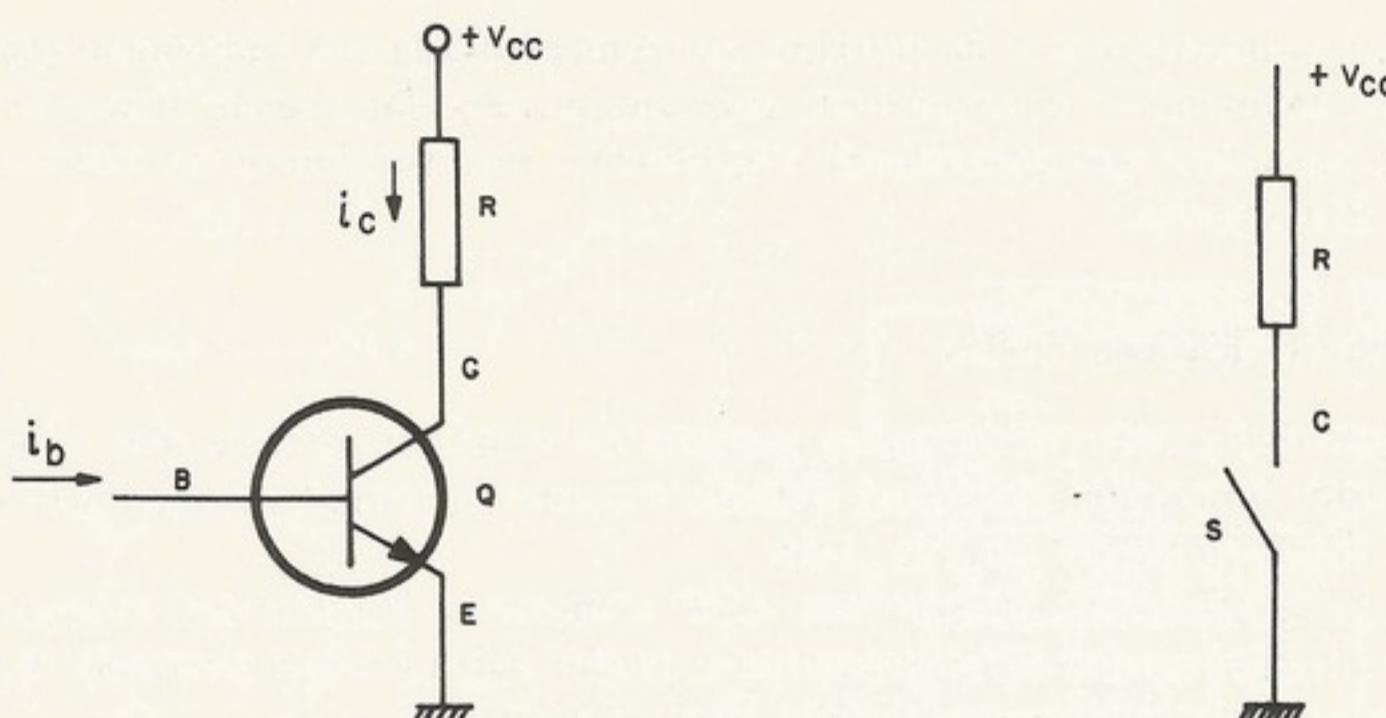


Figura 1-26. O transistor como chave

O transistor tem seu funcionamento caracterizado por três regiões (Fig. 1-27). Quando a corrente de base é nula, o transistor fica na *região de corte*, isto é, fica aberto. À medida que vamos aumentando a corrente da base (i_b), a corrente do coletor ($i_c = \beta i_b$) vai aumentando, caracterizando a *região ativa*, e o potencial do ponto C vai caindo, aproximando-se da terra. Até que se chega a um ponto onde $i_c \cdot R = V_{cc}$, ou seja, a corrente do coletor é suficientemente grande a ponto de a queda de tensão no resistor ser igual à tensão de alimentação. Chegou-se à *região de saturação* do transistor, onde a tensão no ponto C é aproximadamente nula.

Então, para se operar com o transistor como chave, deve-se passar rapidamente pela região ativa. Se o transistor estiver cortado, deve-se saturá-lo imediatamente, passando-se o mais rapidamente possível da situação $i_b = 0$ (cortado) para $i_b \geq V_{cc}/(R \cdot \beta)$ (saturado).

Na Fig. 1-27(b), percebe-se que, quando o transistor está saturado, a tensão de coletor não é exatamente zero, nem quando cortado a corrente de coletor é nula. Os valores dependem do tipo de transistor e do ponto de operação. Na Fig. 1-28, fornecemos alguns desses valores característicos.

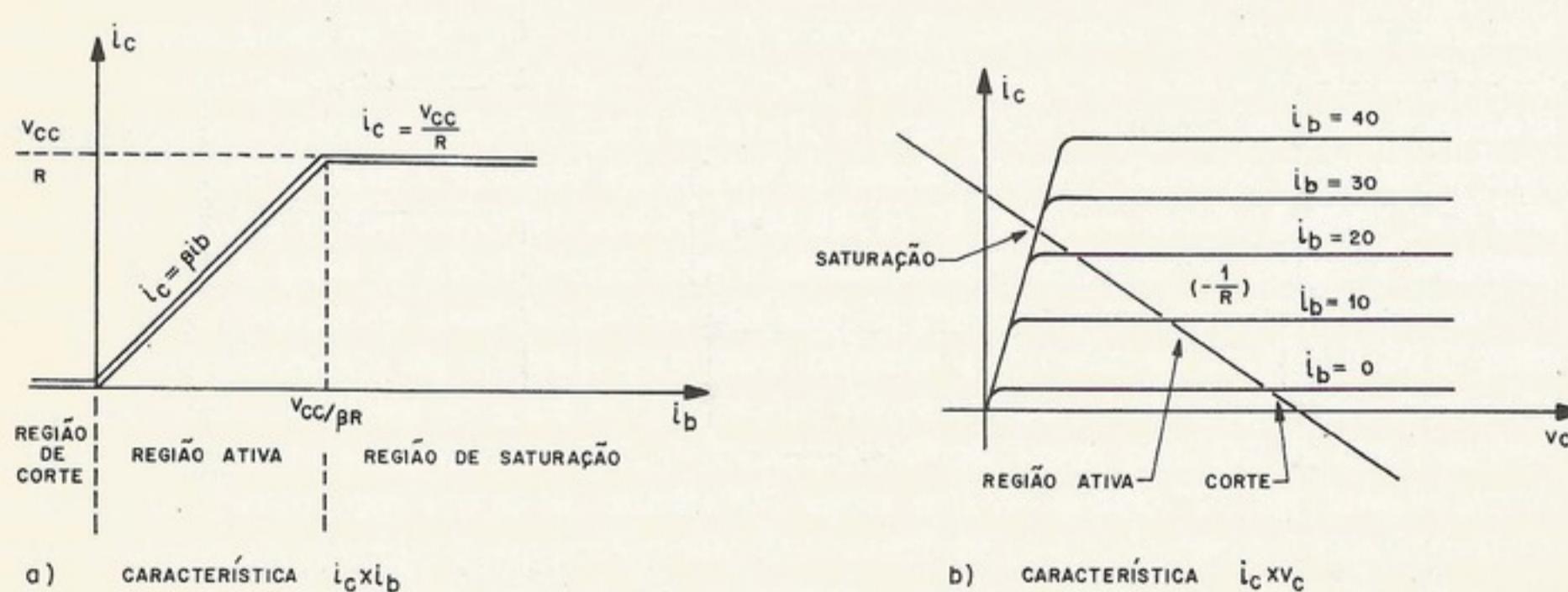


Figura 1-27. Curvas características do transistor como chave

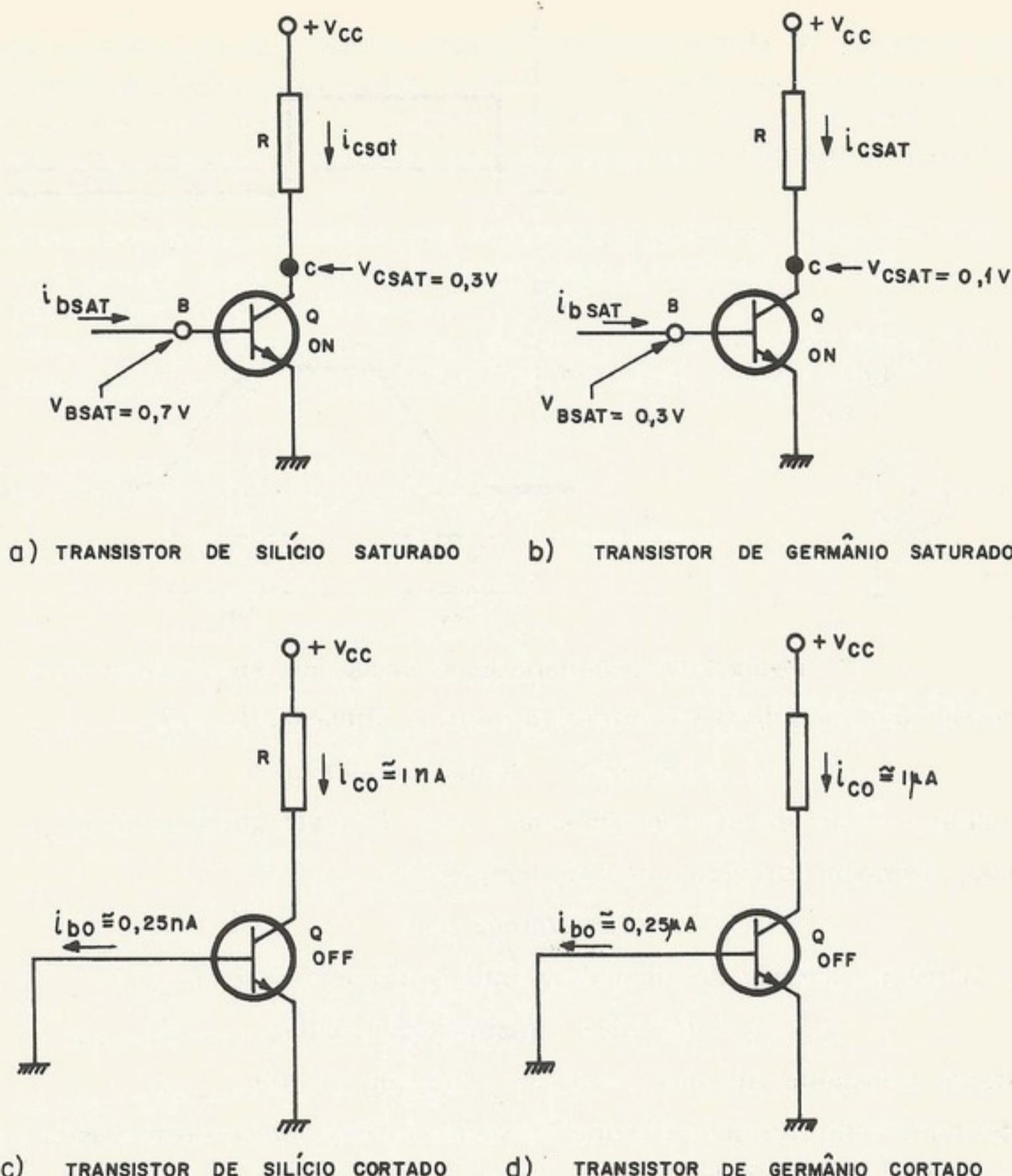


Figura 1-28. Valores característicos dos transistores como chave

Resumindo, podemos dizer que as três regiões de operação são caracterizadas conforme segue.

Corte. $i_b = 0$, duas junções reversamente polarizadas.

Ativa. $0 < i_b < V_{cc}/\beta R$ {junção coletor-base reversamente polarizada,
junção base-emissor diretamente polarizada.

Saturação. $i_b \geq V_{cc}/\beta R$, duas junções diretamente polarizadas.

Estudo do transitório

É importante para o leitor ter em mente que, no instante em que comutamos a corrente de base, nada acontece com a corrente de coletor. Somente depois de algum tempo é que se faz sentir no coletor a variação na base. Isso é provocado pelas capacitâncias das junções e por acúmulo de portadores minoritários na base. (Veja o Cap. 20 da referência⁽¹⁴⁾.)

Para ilustrarmos rapidamente o problema do atraso, apresentamos, na Fig. 1-29, um circuito inversor transistorizado com os tempos a seguir indicados.

Turn-on time. Formado por dois tempos,

$$t_d = \text{delay time},$$

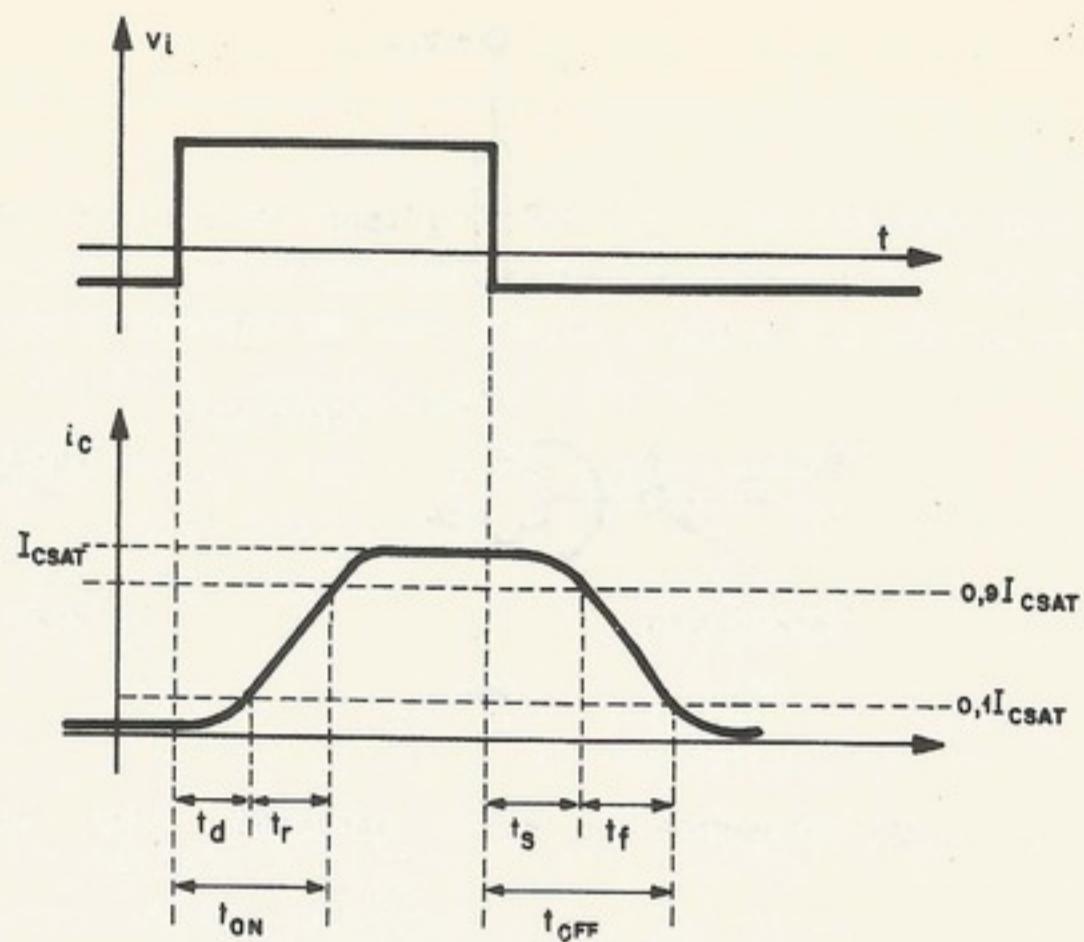
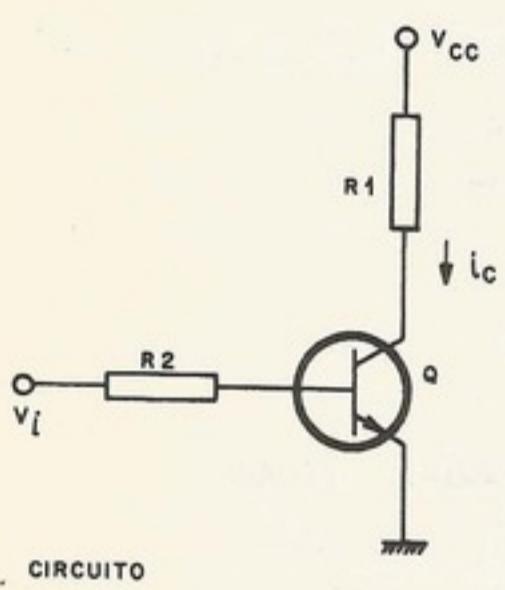


Figura 1-29. Transitório num circuito inversor

que é contado desde a subida da entrada até que i_c atinja $0,1 I_{c\text{sat}}$; e

$t_r = \text{rise time}$ (tempo de subida),

que é contado a partir do instante em que $i_c = 0,1 I_{c\text{sat}}$ até que $i_c = 0,9 I_{c\text{sat}}$.

Turn-off time. Também formado por dois tempos,

$t_s = \text{storage time}$,

contado a partir da descida do sinal de entrada até que $i_c = 0,9 I_{c\text{sat}}$; e

$t_f = \text{fall time}$ (tempo de descida),

contado desde o instante em que $i_c = 0,9 I_{c\text{sat}}$ até que $i_c = 0,1 I_{c\text{sat}}$.

É importante notar que, mais à frente, usaremos a palavra *delay* (atraso) para os tempos *turn-on* ou *turn-off* indistintamente.

c. Transistores em circuitos lógicos

Os circuitos lógicos usualmente implementados com transistores são os *NAND* e *NOR*. A técnica adotada é a da associação de um circuito inverter [Fig. 1-29(a)] na saída de um *AND* ou de um *OR* passivos (veja a Fig. 1-30). O funcionamento do inverter é entendido sem

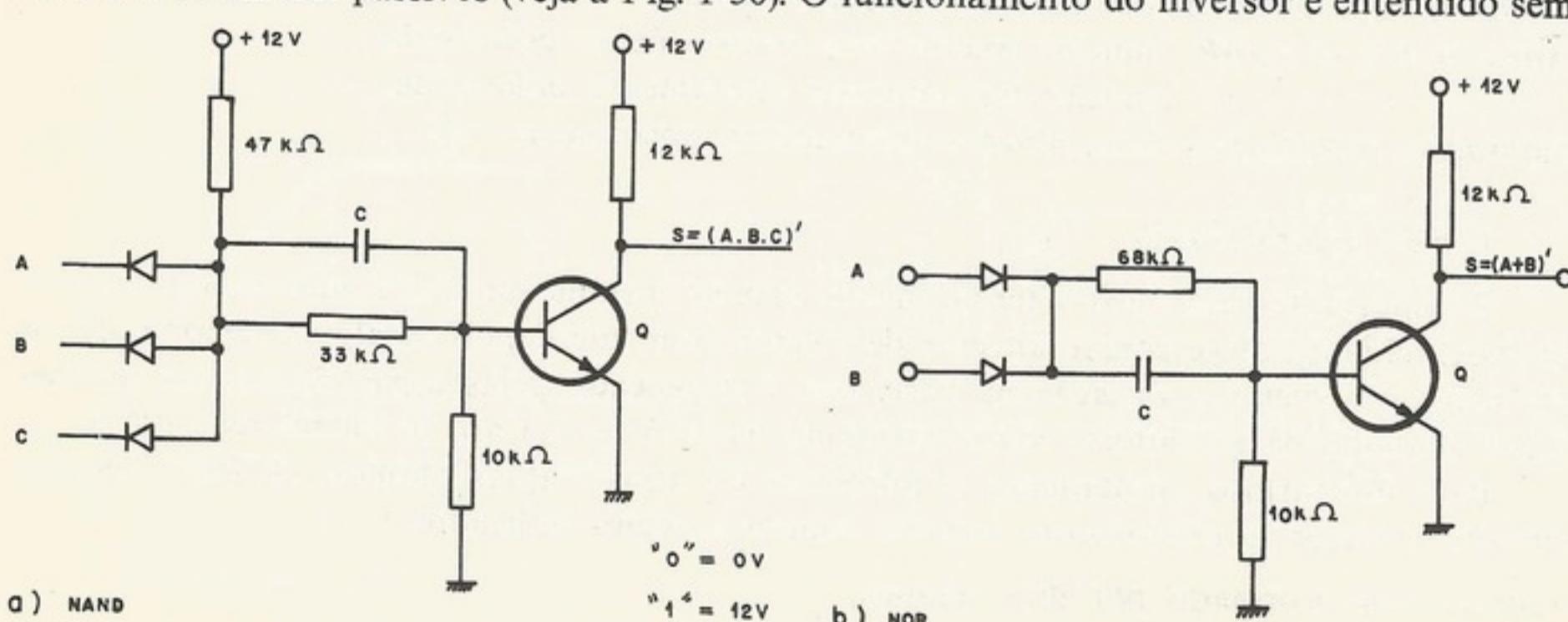
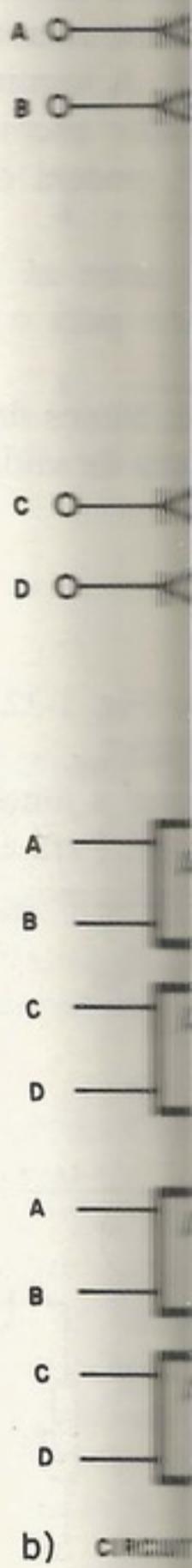


Figura 1-30. Circuitos lógicos transistorizados

problemas se pensarmos que a tensão de saída (Fig. 1-29) é alta quando a saída é menor que a tensão de saída de um inverter com uma única saída. Isso ocorre porque a saída de um inverter com uma única saída é menor que a tensão de saída de um inverter com duas saídas.

Os capacitores de turn-on e turn-off de um *NAND* passa a serem maiores.

No nível de tempo nulo e atraso zero.

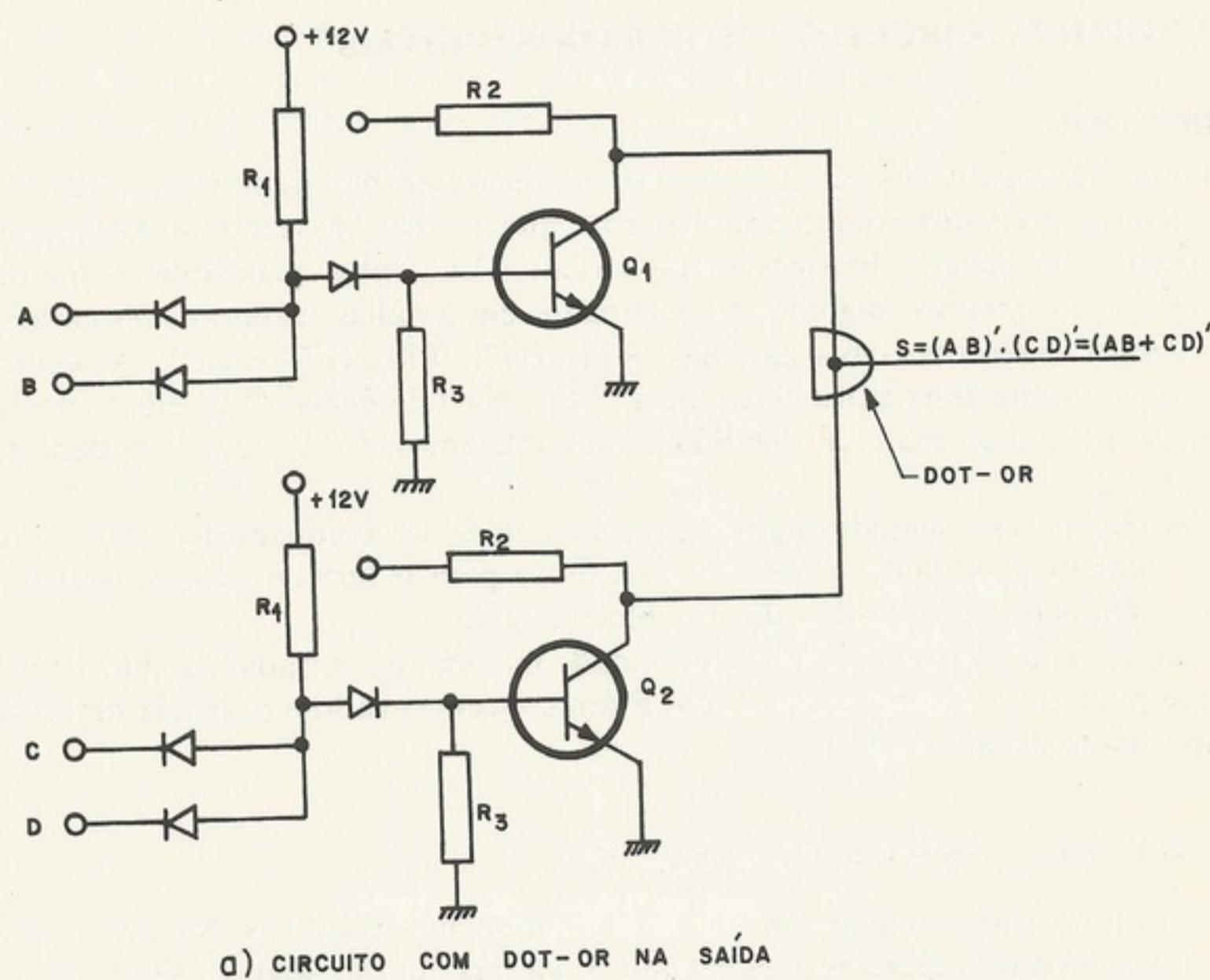


b) CIRCUITO AD

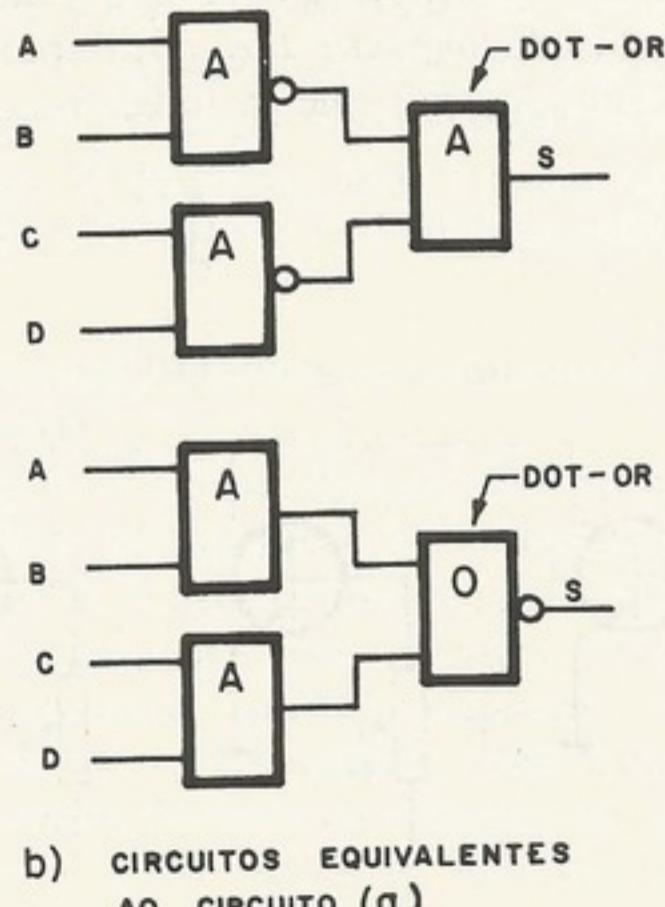
problemas se pensarmos da seguinte maneira: quando a tensão na entrada (V_i no circuito da Fig. 1-29) é alta, existe uma grande corrente na base do transistor que o satura, o que fará com que sua saída (tensão no coletor) caia para zero. Por outro lado, se a tensão de entrada for zero, a corrente na base será nula, o que cortará o transistor e, portanto, a tensão no coletor do transistor será alta.

Os capacitores foram colocados para melhorar a velocidade, ou seja, diminuir os tempos de *turn-on* e *turn-off*. Aqui também, se invertermos a convenção de "1" e "0", o circuito *NAND* passa a ser *NOR* e vice-versa.

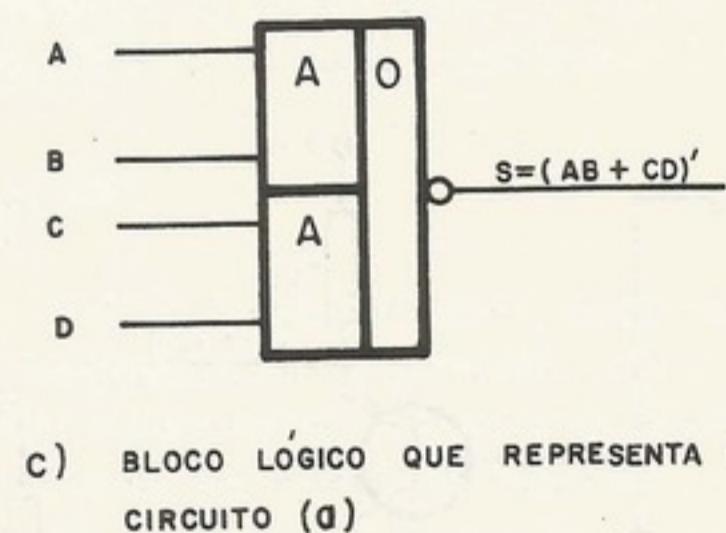
No nível de transistores e resistores, pode-se conceber um novo bloco lógico de custo nulo e atraso zero, o *DOT-OR* ou *Wired-OR*, ou, ainda, *Implied-AND*⁽¹⁷⁾.



a) CIRCUITO COM DOT-OR NA SAÍDA



b) CIRCUITOS EQUIVALENTES AO CIRCUITO (a)



c) BLOCO LÓGICO QUE REPRESENTA O CIRCUITO (a)

Figura 1-31. O DOT-OR

DOT-OR

Se tomarmos dois *NAND* transistorizados e ligarmos suas saídas em curto, tal como na Fig. 1-31, teremos, com esse ponto de ligação, realizado uma função *AND*, pois esse ponto só estará no nível “1” se os dois *NAND* tiverem suas saídas iguais a “1”. Um dos *NAND*, tendo sua saída igual a “0”, é suficiente para o ponto cair a zero, pois o seu transistor de saída saturado drena a corrente dos dois resistores, “derrubando” o ponto para a terra. Pode-se justificar o nome *DOT-OR* a esse ponto que realiza a operação *AND*, mostrando o circuito equivalente da Fig. 1-31(b), onde, pelo teorema de De Morgan, esse *AND* se transforma em um *OR*.

1.5 TECNOLOGIA: CIRCUITOS INTEGRADOS DIGITAIS

a. Introdução

Todos os blocos lógicos estudados existem sob forma de circuito integrado e o projetista não mais se preocupa com o circuito propriamente dito, passando a considerar apenas as interligações dos blocos. Existem vários circuitos diferentes, realizando o mesmo bloco lógico. Por isso os circuitos digitais são agrupados em famílias segundo suas características elétricas. Vamos apresentar cinco famílias, *RTL*, *DTL*, *TTL*, *HTL* e *ECL*. A seguir, vamos falar um pouco sobre uma nova tecnologia, *FET*. Na referência⁽¹⁸⁾ o leitor encontra esse assunto com um pouco mais de detalhes e, nas referências^{(19), (20) e (21)}, poderá estudá-lo detalhadamente.

Vamos definir em seguida alguns conceitos úteis na comparação entre as famílias. *Delay (atraso)*. Usaremos, genericamente, tanto para o *turn-on* quanto para o *turn-off*.

Fan-in. Número de entradas de um bloco lógico.

Fan-out. *Fan-out* de um bloco é a quantidade máxima de entradas, de blocos da mesma família, que podem ser ligadas à sua saída. É um conceito ligado à potência de saída do circuito (capacidade de *driving*).

b. Família *RTL* (“resistor transistor logic”)

São circuitos usados principalmente com componentes discretos. Na Fig. 1-32, o leitor encontra dois circuitos diferentes realizando a mesma função lógica, *NOR*.

O resistor em série com a base torna esse circuito lento, já que, com a junção base-emissor, forma um circuito *RC*, aumentando os tempos de *turn-on* e *turn-off*. Para aumentar a velocidade, poderíamos pensar em adotar a mesma solução que já indicamos, ou seja,

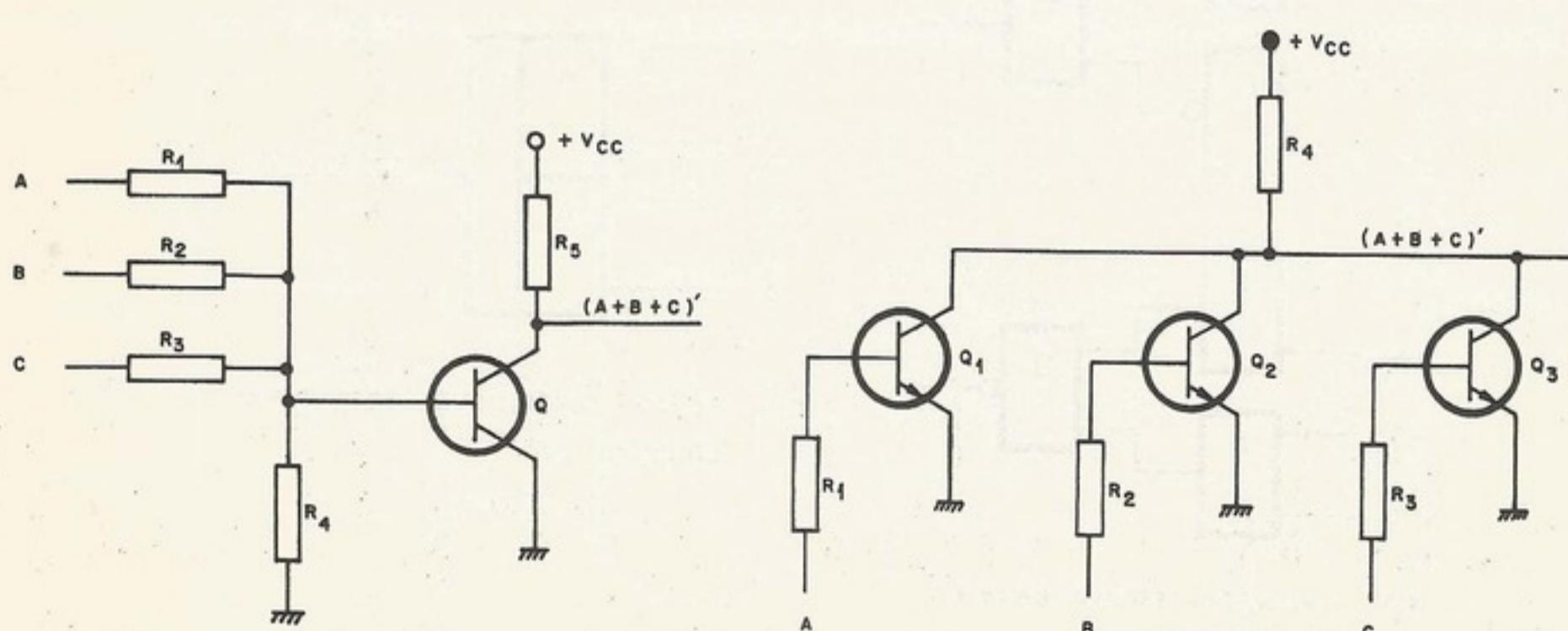


Figura 1-32. Circuitos *RTL*

em custo, tal como TTL, pois esse ponto é um dos *NAND*, comum a todos os circuitos integrados.

considerar apenas o mesmo bloco das características. A seguir, vamos encontrar esse moderno estudo.

as famílias. para o turn-off. micos da mesma saída do cir-

Fig. 1-32, o leitor

a junção base. Para aumentar isso, ou seja,

usar capacitores em paralelo com o resistor de base (família *RCTL*). Essa solução, que é adotada em componentes discretos, fica impraticável em circuitos integrados dado o grande volume exigido pelo capacitor.

Essa família ganha em custo de todas as outras, perdendo em *fan-out*. Dado o baixo custo e pequena quantidade de componentes, essa tecnologia tem sido usada em memórias monolíticas bipolares.

c. Família DTL ("diode transistor logic")

Tem a estrutura do tipo da que é vista na Fig. 1-33. Com a saída com resistores em *pull-up*, a capacidade de *drive* é muito maior no nível "0" e, por isso, para aumentar o *fan-out*, aumentou-se a impedância de entrada no nível "1" colocando-se os diodos vistos na figura. O diodo em série com a base serve para aumentar a imunidade ao ruído, e o resistor que liga a base à terra é importante para diminuir o tempo de *turn-off*.

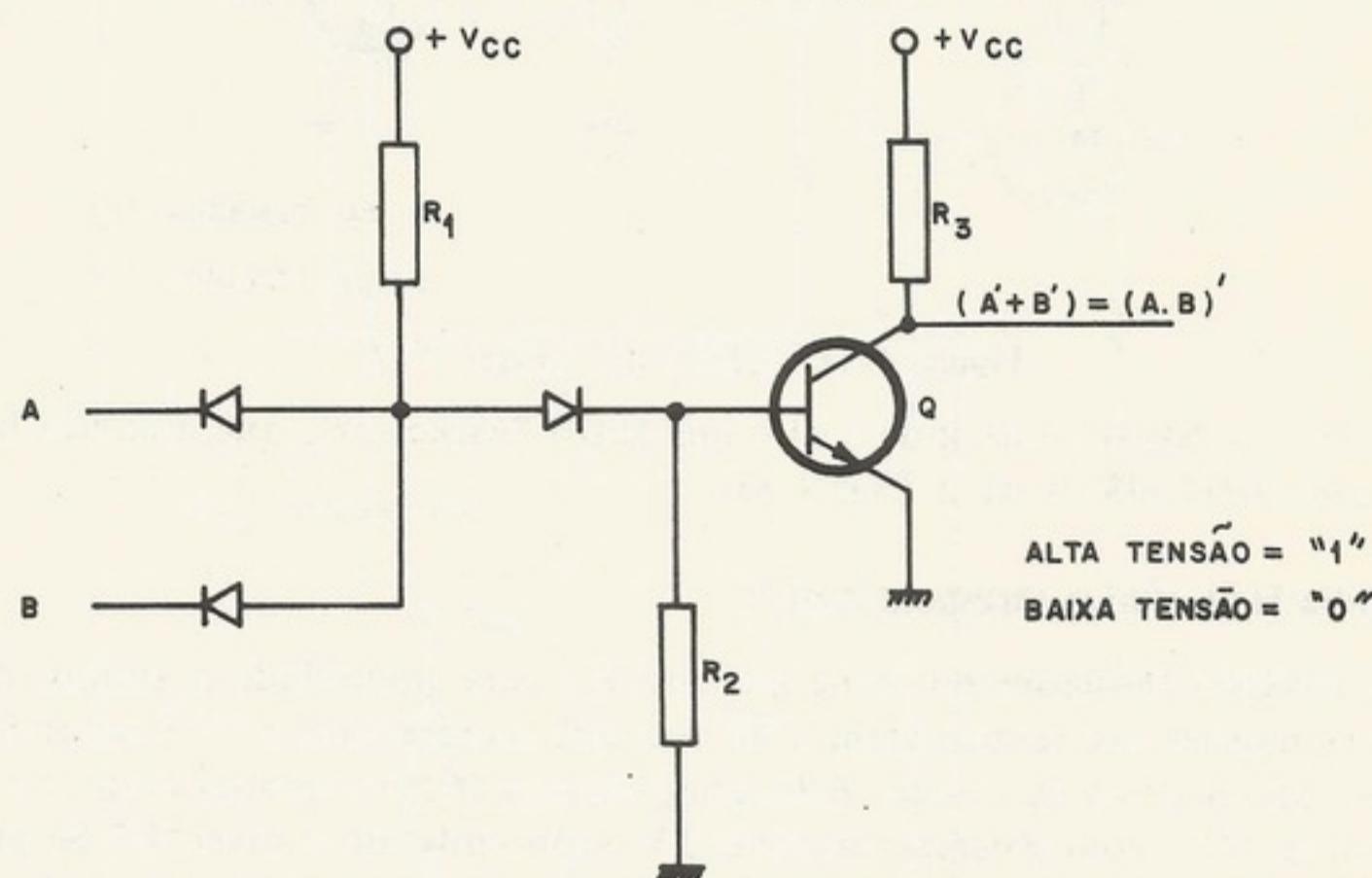


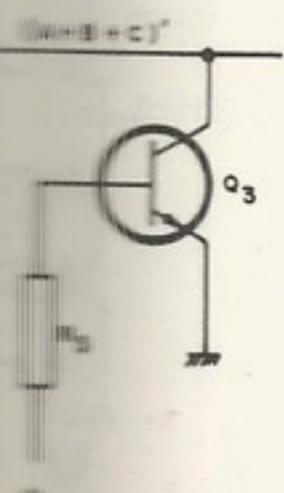
Figura 1-33. *NAND* da família *DTL*

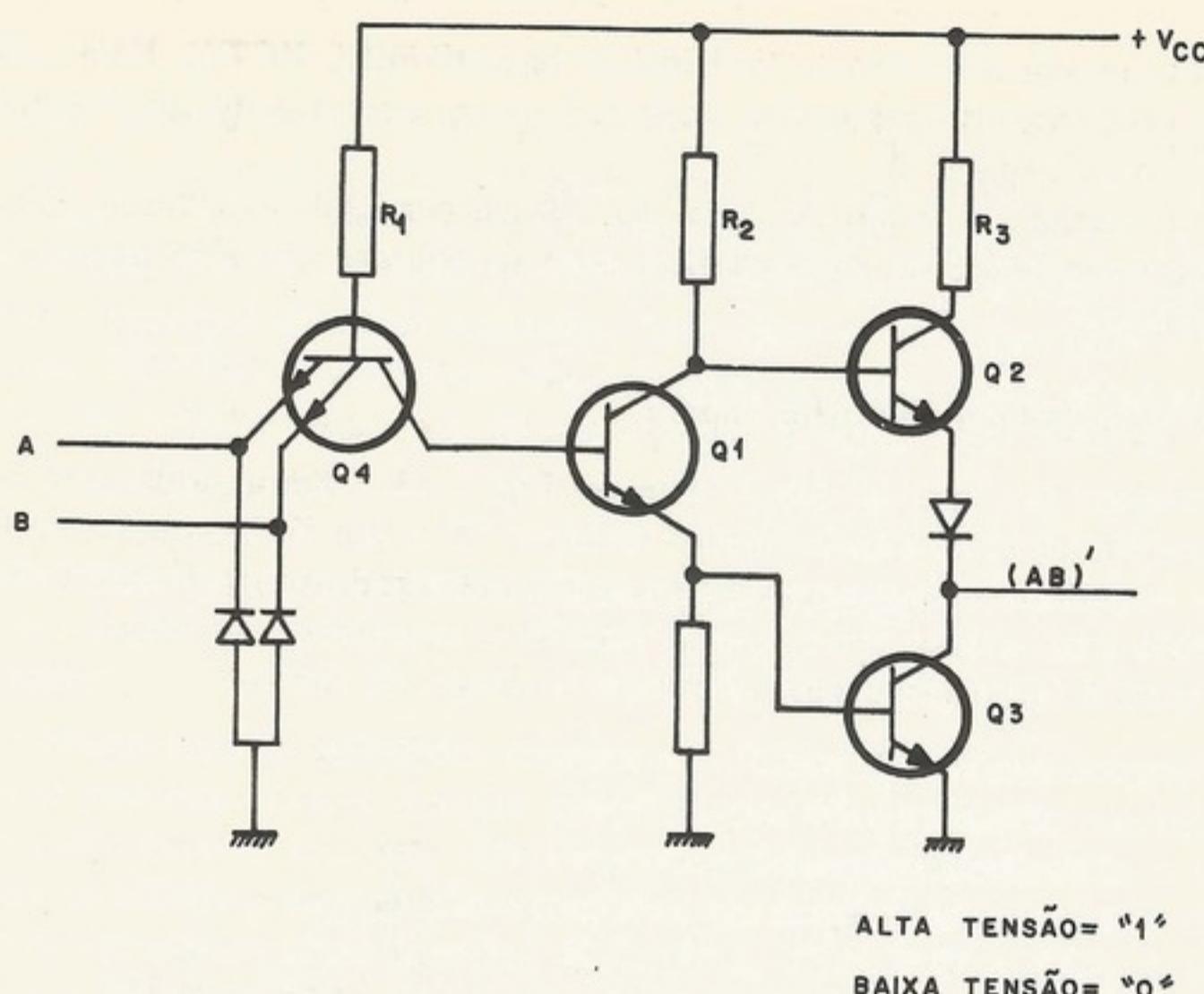
O problema da imunidade ao ruído é o que segue. Se não existisse o diodo em série com a base, o potencial na saída do *AND* quando igual a "0" (0,6 V, se silício), refletir-se-ia na base, e o transistor ficaria no limite entre o corte e a saturação, e qualquer ruído na entrada o levaria à região ativa; dependendo do ganho, poderia saturá-lo, mudando a sua saída. O diodo em série é como uma barreira de tensão (0,6 V, se silício) impedindo o transistor de conduzir se o sinal na saída do *AND* for de baixa amplitude (1,2 V, no caso).

d. Família TTL ("transistor transistor logic")⁽²²⁾

A Fig. 1-34 mostra um circuito apenas viável na forma integrada, dada a necessidade do transistor multiemissor na entrada. Seu funcionamento é parecido com o *DTL*, com a vantagem do transistor multiemissor aumentar o *turn-off* do transistor Q_1 . Para aumentar o *fan-out*, geralmente, esses circuitos são providos de um estágio de saída de potência, tipo *push-pull*, chamado aqui de *totem-pole*. Além do custo, têm a desvantagem de criar uma situação, durante a comutação, onde os transistores Q_2 e Q_3 ficam conduzindo, o que drena grande corrente da fonte de alimentação, gerando ruídos (*glitches*), muitas vezes indesejáveis.

Além de gerar ruídos na rede de alimentação, essa saída de potência não permite a realização da função *DOT-OR*, que, pelo seu custo e atraso nulos, é muito usada. Para resolver o problema, os projetistas de circuitos digitais criaram um bloco chamado "expan-



Figura 1-34. *NAND* da família *TTL*

sível" (dá acesso a dois pontos internos) e um outro "expansor", que, quando ligados, realizam a função *DOT-OR* (veja a Fig. 1-35).

e. Família HTL ("high threshold logic")

A característica fundamental desse circuito é a alta imunidade a ruídos, dado que o limiar entre as tensões que representam o "0" e as que representam o "1" é relativamente alto. Por exemplo, um circuito da família *HTL* pode funcionar com tensões entre 3 e 5 V, representando o nível "1" e com tensões entre 0 e 2 V, representando o nível "0". Se procurarmos operar o sistema com sinais do 0 V para o nível "0" e 5 V para nível "1", ruídos de até 2 V não atuarão no circuito.

Existem várias maneiras de se implementarem circuitos dessa família. A mais simples é vista na Fig. 1-36. O diodo zener é responsável pelo alto limiar.

f. Família ECL ("emitter coupled logic")

É uma família totalmente diferente das outras, já que, para diminuir o atraso, os transistores não operam saturados (técnica *current-switch*), o que descreveremos tomando um circuito típico como exemplo (Fig. 1-37).

O circuito funciona da maneira que segue. Pelo resistor de $1,24\text{ k}\Omega$ passa uma corrente mais ou menos constante; se nenhum dos transistores, Q_1 ou Q_2 , está conduzindo, então o transistor de referência, Q_3 , conduz, mas não satura. Com Q_3 conduzindo, o transistor Q_4 está cortado e o transistor Q_5 conduz. Com Q_1 ou Q_2 conduzindo, o transistor Q_4 conduz e o transistor Q_5 corta. A saída, sendo do tipo seguidor em emissor (*emitter follower*) dispõe-se de alta capacidade de *drive*.

Uma vantagem do circuito são as duas saídas de polaridade invertida, isto é, um mesmo bloco tem duas saídas, *OR* e *NOR*. O circuito é relativamente veloz porque os transistores não operam saturados, necessitando, porém, de bastante potência. Sua imunidade ao ruído é razoável, pois tem um limiar definido pela tensão na base de Q_3 . É, porém, incompatível com as famílias *TTL* e *DTL*, isto é, não pode ser ligado junto com blocos dessas famílias, por problemas de acoplamento de impedâncias e níveis.

g. Circuitos integrados

O transistores de efeito de campo (Tec) é a tecnologia *MOS* (*metal-oxigênio-semicondutor*). Assemelha à *PNP* e *NPN*, mas são portadores de carga. São portadores de carga. Pelo potencial de referência, é chamado de *VDD*.

A Fig. 1-38 mostra o princípio de funcionamento de circuitos integrados. Colocando os transistores em série obtém-se uma saída com maior potência de saída. Devido ao efeito de carga.

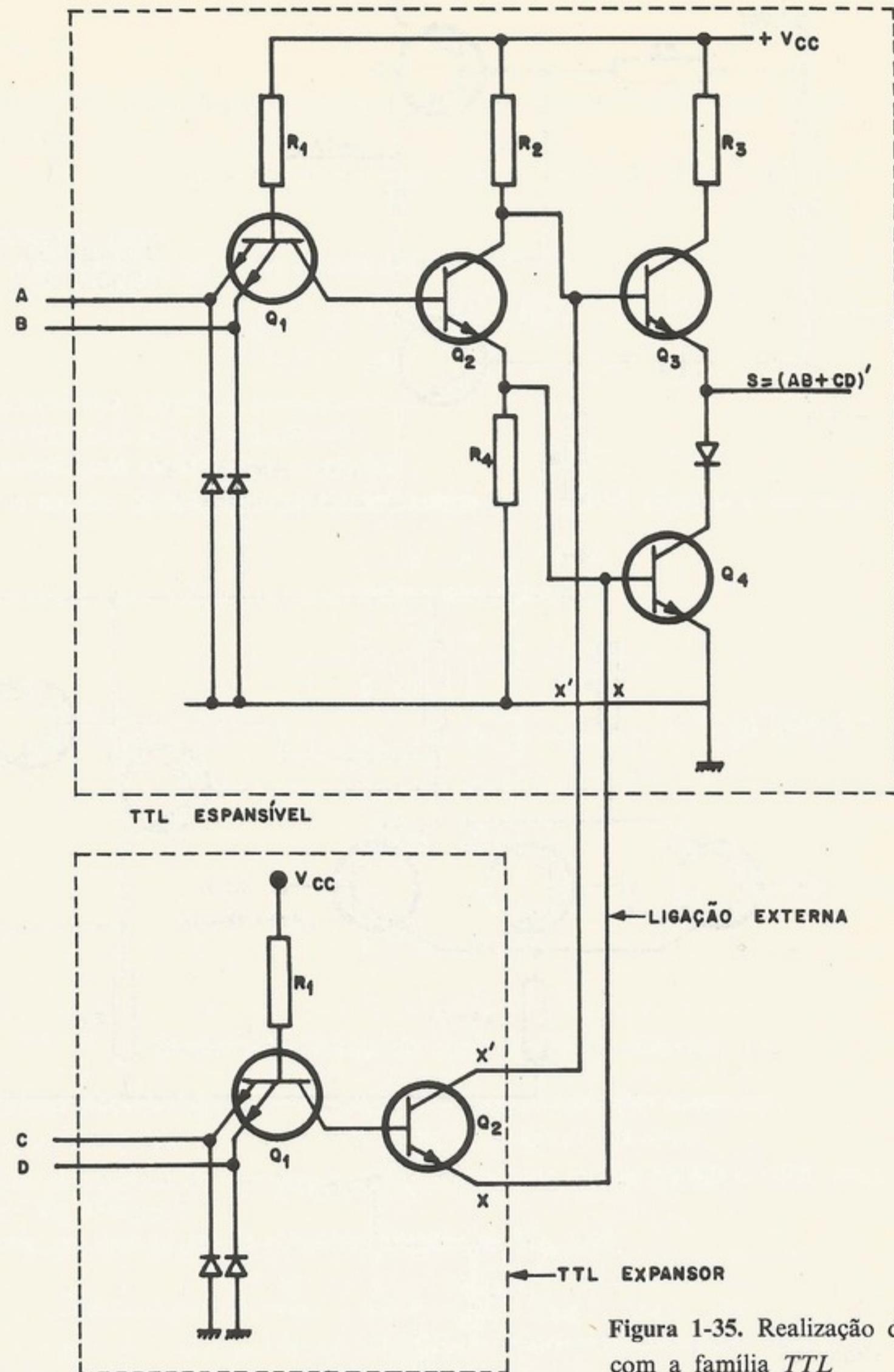
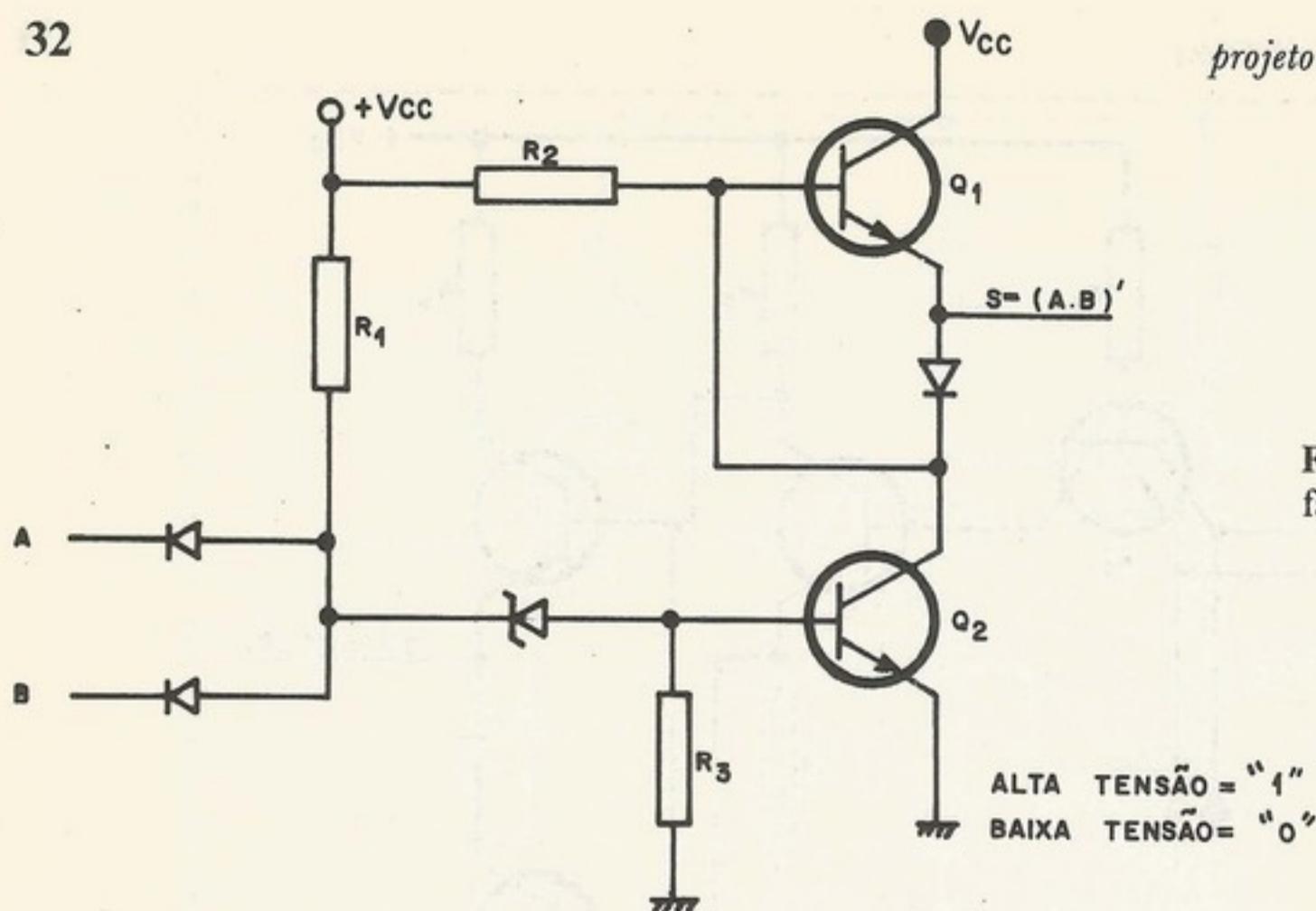
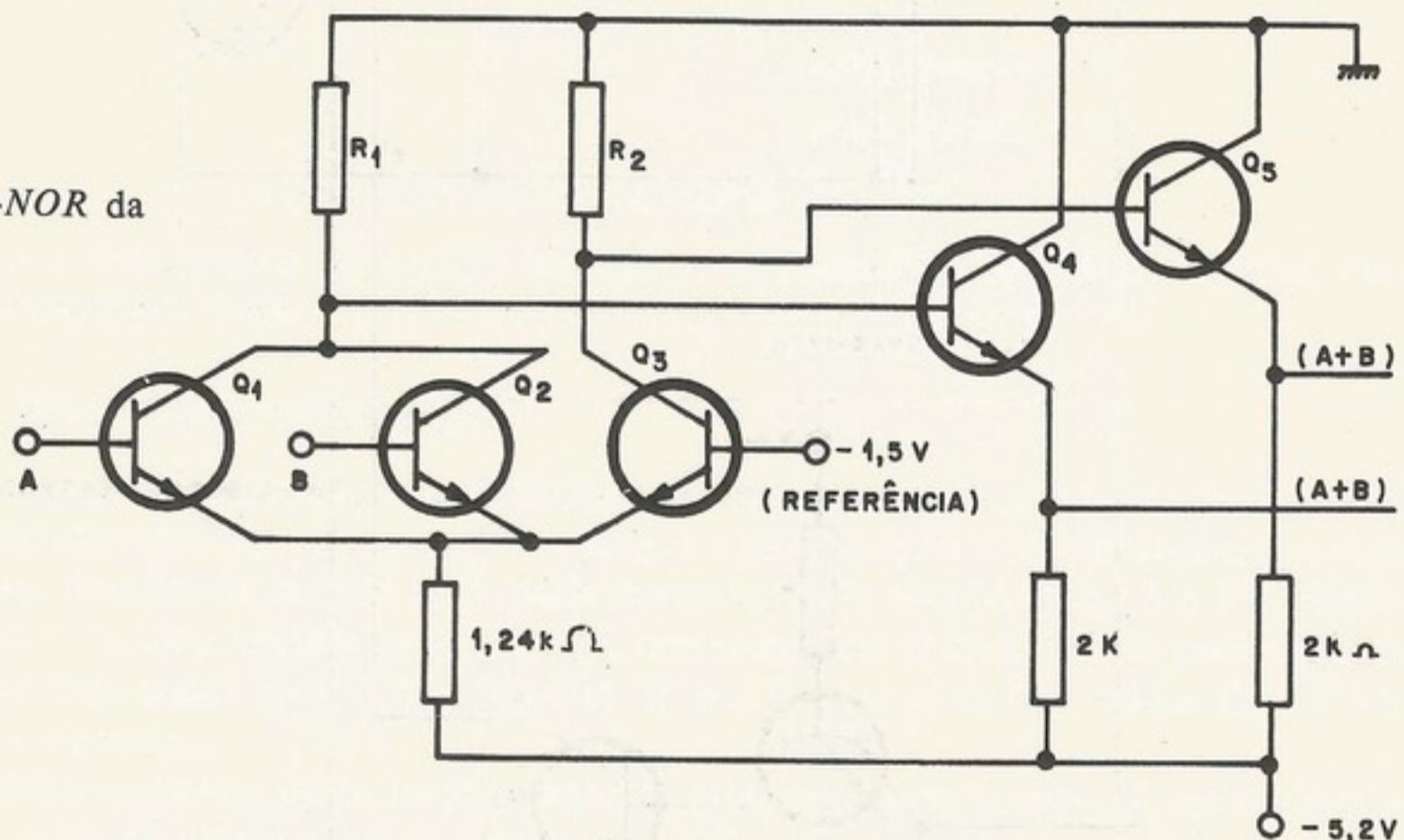


Figura 1-35. Realização de *DOT-OR* com a família TTL

g. Circuitos MOS-FET

O transistor de efeito de campo *FET* (*field-effect transistor*) é implementado na tecnologia *MOS* (*metal-oxide semiconductor*). O FET é o dispositivo semicondutor que mais se assemelha à válvula a vácuo. É caracterizado como monopolar porque apenas os elétrons são portadores de carga. O fluxo de elétrons entre os elétrodos *source* e *drain* é controlado pelo potencial no elétrodo *gate*. (Não confunda o elétrodo *gate* com o circuito lógico que também é chamado *gate*.)

A Fig. 1-38 mostra a aplicação de circuitos com *FET* como circuitos lógicos combinatórios. Colocando os dispositivos *FET* em paralelo formam-se *gates* tipo *NOR* e colocando-os em série obtém-se *gates* tipo *NAND*. Na figura, o *FET* ligado à fonte funciona como o resistor de carga.

Figura 1-36. *NAND* da família *HTL*Figura 1-37. *OR-NOR* da família *ECL*

introdução e ...

EXERCÍCIOS

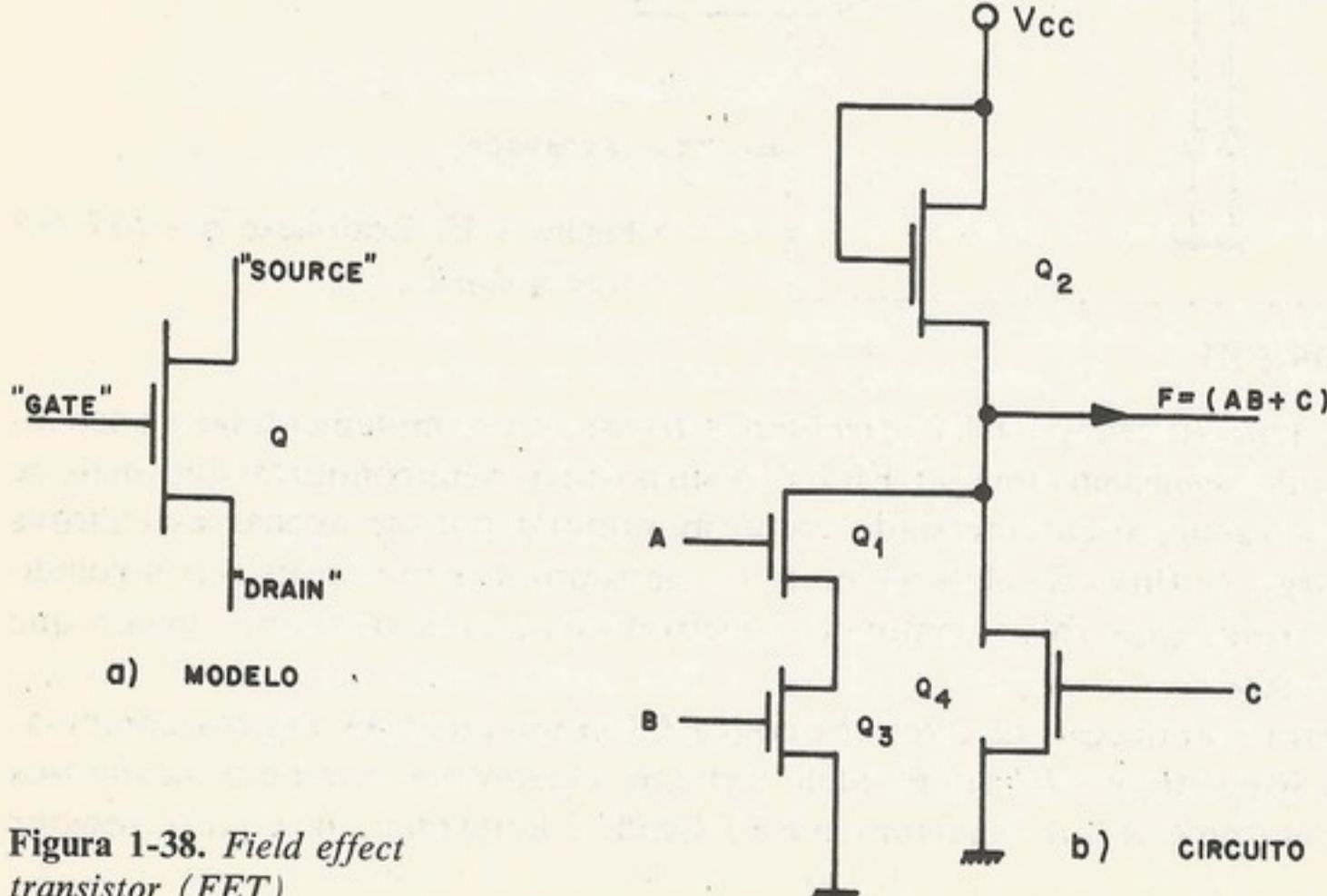
- 1-1. Demonstre que
1-2. A igualdade

é uma identidade:
1-3. Dada a expressão

determine:

- (a) sua descrição
(b) sua descrição
(c) sua descrição
1-4. Dado o circuito

- 1-5. Sintetize os circuitos:
(a) $s = a'(bc + d'')$
1-6. Sintetize os circuitos usando os teoremas de De Morgan.
1-7. Sintetize, em

Figura 1-38. *Field effect transistor (FET)*

EXERCÍCIOS

- 1-1. Demonstre por manipulação da expressão booleana os teoremas (T. 17) e (T. 18).
 1-2. A igualdade

$$(A + B) \cdot (A' + C) \cdot (B + C) = (A + B) \cdot (A' + C)$$

é uma identidade? Prove.

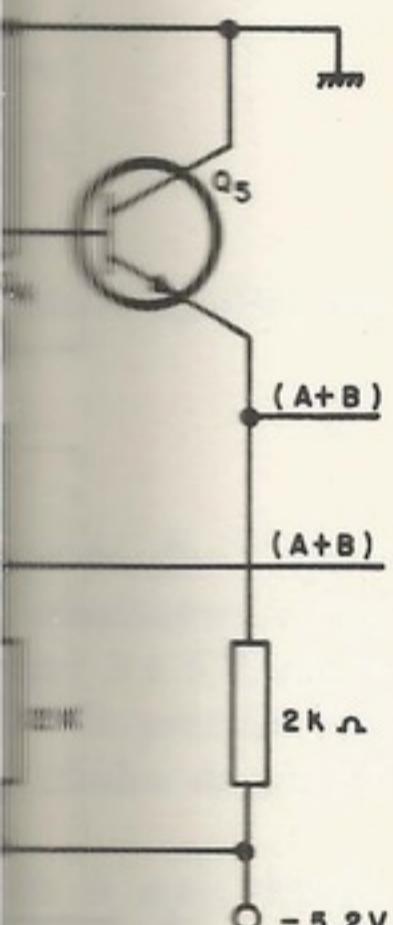
- 1-3. Dada a descrição de um circuito combinatório,

$$s = ab + a'b' + acb'$$

determine:

- (a) sua descrição pela tabela de combinações;
 (b) sua descrição pelo mapa de Karnaugh;
 (c) sua descrição pela transformada numérica.

- 1-4. Dado o circuito descrito pela tabela de verdade da Fig. 1-39, obtenha a sua soma canônica.



a	b	c	s
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Figura 1-39. Tabela de combinações do Exercício 1-4

- 1-5. Sintetize os circuitos mínimos dados pelas seguintes expressões booleanas:
 (a) $s = a'(bc + b'') + a'b$; (b) $s = x(y' + z) + z'(x + y)$; (c) $s = ab + a'b'$.
 1-6. Sintetize os circuitos do exercício anterior apenas com apenas *NAND* e *NOR*. [Sugestão. Use os teoremas de De Morgan.]
 1-7. Sintetize, em dois níveis, os circuitos descritos na Fig. 1-40.

x	y	z	s
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

a) CIRCUITO a

		ab	00	01	11	10
		cd	00	01	11	10
00	0	0	0	0	1	0
01	1	0	1	0	1	1
11	0	1	0	1	1	0
10	1	0	0	0	1	0

b) CIRCUITO b

Figura 1-40. Descrições dos circuitos do Exercício 1-7

1-8. Sabendo que o β do transistor do circuito da Fig. 1-41 é igual a 10, calcule o valor de R_1 .

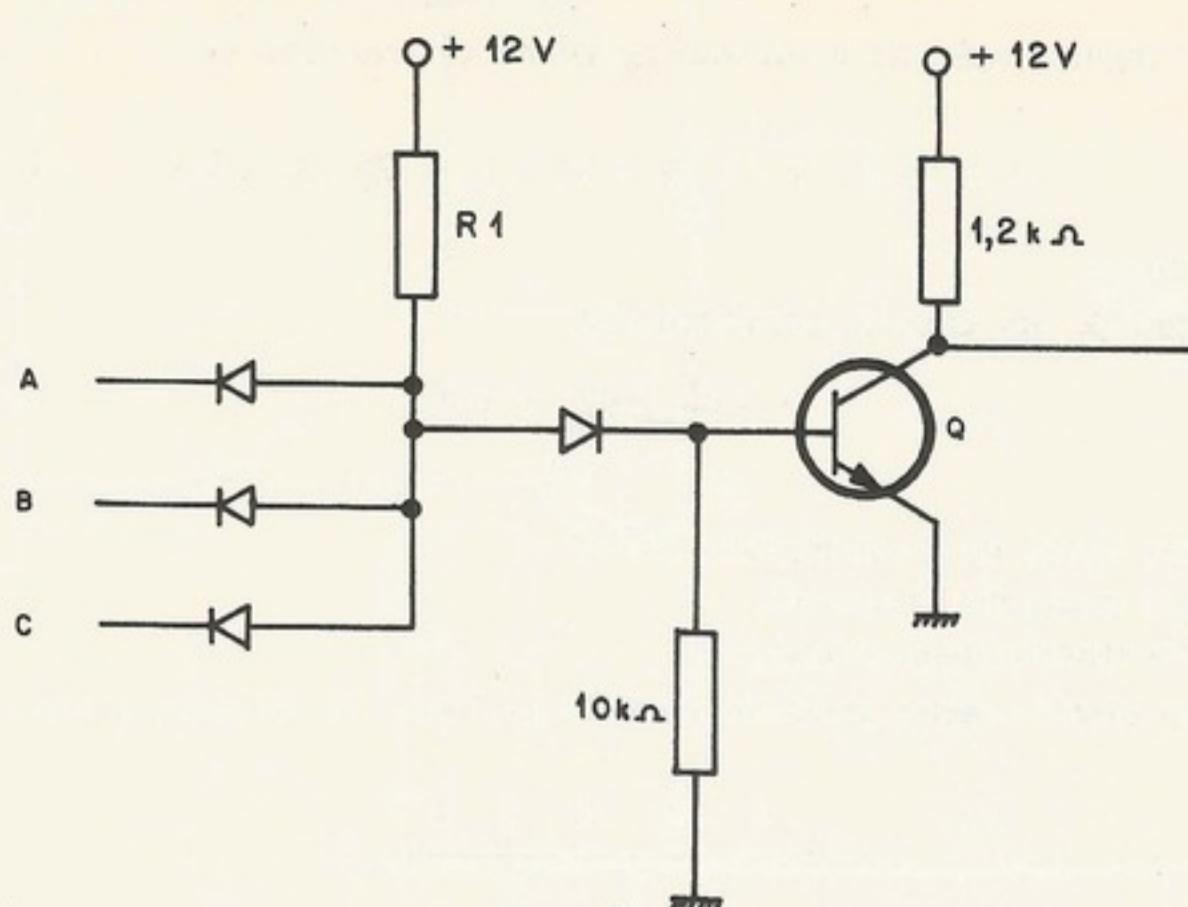


Figura 1-41. Circuito do Exercício 1-8

1-9. Projete dois circuitos que acoplem as famílias TTL com ECL, isto é, um que acopla a saída do TTL com a entrada do ECL, e outro que acople a saída do ECL com a entrada do TTL.

1-10. Faça um circuito apenas com FET que sintetize a função

$$s = A(C + D) + B.$$

BIBLIOGRAFIA

- (1) Nievergelt, J., "Computers and Computing – Past, Present, Future", *IEEE Spectrum*, janeiro de 1968, pp. 57-61
- (2) Laver, F. J. M., *Introducing Computers*, Her Majesty's Stationery Office, Londres, 1965
- (3) Aiken, H., "Proposed Automatic Calculating Machine", *IEEE Spectrum*, agosto de 1964, pp. 62-69
- (4) Burts, A. W. et al., *Preliminary Discussion of the Logical Design of an Electronic Computing Instrument*, junho de 1946. (Republicado na Datamation em setembro/outubro de 1962)
- (5) Boole, G., *An Investigation of the Laws of Thought*..., Londres, 1854. (Republicado pela Dover Publications, New York, 1951)
- (6) Shannon, C. E., "A Symbolic Analysis of Relay and Switching Circuits", *Electrical Engineering*, Trans. Suppl. 57, pp. 713-723, 1938
- (7) Wood, P. E., *Switching Theory*, Lincoln Laboratory Publications, McGraw-Hill, 1968
- (8) Miller, R. E., *Switching Theory*, John Wiley and Sons, N.Y. 1965
- (9) McCluskey, E. J., *Introduction to the Theory of Switching Circuits*, McGraw-Hill, 1965
- (10) Del Picchia, W., "A transformada numérica e sua aplicação à simplificação de funções e à resolução de equações booleanas", Tese do Departamento de Engenharia de Eletricidade da EPUSP, 1971
- (11) Hellerman, L., "A Catalog of Three Variable Or-Invert and And-Invert Logical Circuits", *IEEE Transactions on Electronic Computers*, junho de 1963, pp. 198-223
- (12) Baugh, G. R., et al., "Optimal Networks of NOR-OR Gates for functions of Three Variables", *IEEE Transactions on Computer*, fevereiro de 1972, pp. 153-160
- (13) Sifferlen, T. P., *Digital Electronics With Engineering Applications*, Prentice Hall, N.J., 1970
- (14) Millman, J., e Taub, H., *Pulse Digital and Switching Waveforms*, McGraw-Hill, 1965
- (15) Harris, J. N., et al, *Digital Transistor Circuits – Semiconductor Electronics Education Committee*, SEEC – Vol. 6, John Wiley, N.Y., 1966
- (16) Zuffo, J. A., *Subsistemas Digitais e Circuitos de Pulso*, EPUSP, 1972
- (17) Weger, Ph., "Wired-or With the FC Family of Integrated Circuits", Philips, Application Information, 847, Holanda, 1969
- (18) Comenzind, H. R., "Digital Integrated Circuit Design Techniques", *Computer Design*, novembro de 1968, pp. 52-62

- (19) Garret, L. S., "Influence of Logic on Computer Design", *IEEE Spectrum*, januário de 1968
- (20) Garret, L. S., "Influence of Logic on Computer Design", *IEEE Spectrum*, novembro de 1968
- (21) Garret, L. S., "Influence of Logic on Computer Design", *IEEE Spectrum*, dezembro de 1968
- (22) Beeson, R. H., *Designing Solid-State Circuits*, McGraw-Hill, 1968
- (23) Unger, S. H., *Application of Solid-State Components to Computer Design*, McGraw-Hill, 1968

- (19) Garret, L. S., "Integrated circuit Digital Logic Families — Part I, RTL and TTL devices", *IEEE Spectrum*, outubro de 1970
- (20) Garret, L. S., "Integrated Circuito Digital Logic Families — Part II, TTL devices". *IEEE Spectrum*, novembro de 1970, pp. 63-72
- (21) Garret, L. S., "Integrated Circuit Digital Logic Families — Part III, ECL and MOS devices", *IEEE Spectrum*, dezembro de 1970, pp. 30-42
- (22) Beeson, R. H., Ruegg, H. W., "New Forms of All-Transistor Logic", *Digest*, 1962, International Solid-State Circuits Conference, IEEE, pp. 10-11, 104
- (23) Unger, S. H., *Asynchronous Sequential Switching Circuits*, John Wiley Interscience, N.Y., 1969

capítulo 2

CÓDIGOS, NÚMEROS E ARITMÉTICA

2.1 CÓDIGOS

Para se processarem informações por um computador, é necessário representar as informações numa forma que ele reconheça. A unidade mais básica de informação é o dígito binário, ou *bit*. O *bit* pode ter valor “0” ou valor “1”. Um conjunto de 7 a 9 *bits* (dependendo do veículo, meio ou código) é chamado um carácter. Esses códigos têm 1 *bit* de paridade; então 6 a 8 *bits* para representar conjuntos de 64 a 256 caracteres alfa-numéricos. Às vezes, carácter é usado como subdivisão de uma *palavra* de 16 a 64 *bits*. A troca de informações entre a máquina e o homem geralmente é feita em caracteres. Um código de 7 bits dispõe de 127 ou 128 caracteres distintos.

Um *código* é um conjunto de regras (ou tabela de combinações) pelo qual informações ou dados (por exemplo, números ou letras) podem ser convertidos a uma representação do código e vice-versa. Um dos códigos mais importantes é o código de cartões perfurados, chamado Hollerith, o nome do inventor. Esse código é de 12 *bits*, e tem representações para a maioria dos símbolos encontrados numa máquina de escrever.

O processo de codificação acontece quando uma tecla de uma perfuradora é batida, sendo codificada mecanicamente num código de 7 *bits*. Uma espécie de decodificação é a que o leitor está fazendo neste instante. Os conjuntos de letras que compõem estas palavras estão sendo reconhecidos, trazendo da memória o sentido da palavra.

Quando um carácter codificado é enviado de uma unidade a outra, por exemplo do computador à impressora, é costume enviar junto um *bit* a mais, chamado *bit* de “paridade”. Num sistema com paridade “ímpar”, para cada carácter junto com o *bit* de paridade, deve haver um número ímpar de *bits* no estado “1”. Por exemplo, a combinação 0010110 tem três *bits* no estado “1”, então o conjunto é “válido” num sistema de paridade ímpar. Um código padronizado para o intercâmbio de informação é chamado *ASCII* (código-padrão americano para intercâmbio de informação). Existem códigos *ASCII* para 7 *bits* de informação com 1 *bit* de paridade e, também, versões de 8 *bits* mais um de paridade para fitas magnéticas. As máquinas dos sistemas 360 e 370 da IBM usam um código de 8 *bits* de informação mais um *bit* de paridade chamado *EBCDIC* (*extended binary coded decimal interchange code*). Um outro código comum, usado em fita perfurada, é o código Baudot, de 5 *bits*.

Existe uma classe de códigos chamados *códigos de distância unitária* (*unit distance codes*). Esses códigos são úteis para se codificar uma posição mecânica ou angular, como um eixo. A distância Hamming entre dois caracteres de um código é o número de posições de *bit* em que os dois caracteres diferem. Por exemplo, caracteres 000111 e 001111 diferem em uma posição só, enquanto que caracteres 000111 e 111000 diferem em todas as seis posições. A idéia básica de códigos de distâncias unitárias é ser código ordenado (existe uma ordem 1, 2, 3, 4, etc.) de modo tal que caracteres vizinhos ($n, n + 1$) difiram em uma só posição.

0
1
a) 1 BIT

NOVA COLUNA

0	0000
0	0001
0	0010
0	0011
1	0100
1	0101
1	0110
1	0111
0	1000
0	1001
0	1010
0	1011
1	1100
1	1101
1	1110
1	1111

c) código

O código mais p...
é, também, comum...
é gerado de um...

Para o código...
do último caracte...
esquerda da qual...

O código de...
o último caracte...
tanto, para com...

Uma aplicac...
de informação é p...
centar bits às pa...
no veículo, meio...
(um bit só da pa...
uma unidade. Ele...
o carácter em qu...
de correção de er...
mapa de Karnaugh...
como tais códigos...

Para aplicac...

2.2 NÚMEROS

a. Sistemas

Um número...
é o valor associ...
dos pesos comuns...
valor de cada pa...

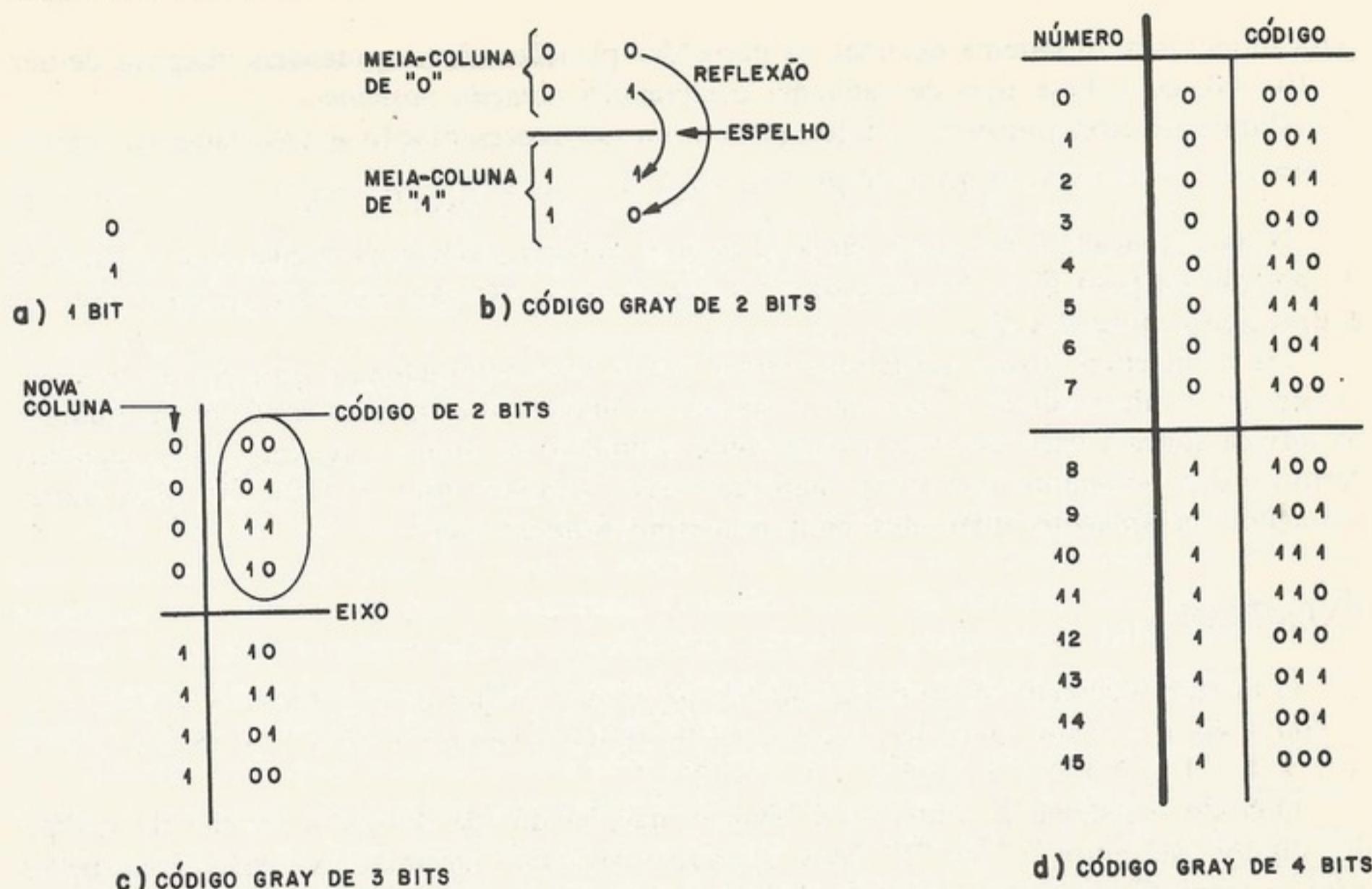


Figura 2-1. A geração de códigos de Gray

O código mais popular desse tipo é o código de Gray (*Gray code*, inventado por Gray), que é, também, conhecido como o código binário refletido (*reflected binary code*). Esse código é gerado de um modo bastante simples, como mostrado na Fig. 2-1.

Para o código de Gray de dois bits, a coluna "0,1" é refletida sobre um eixo embaixo do último carácter, fazendo a coluna 0,1,1,0. A seguir, aumenta-se uma nova coluna, na esquerda da qual, a metade superior consta de "0" e a metade inferior consta de "1".

O código de Gray também tem como propriedade o fato de a distância Hamming entre o último carácter na ordem e o primeiro ser uma unidade só. O código de Gray é útil, portanto, para contadores em que a primeira conta segue a última (veja Cohn e Even⁽¹⁾).

Uma aplicação de códigos em que existem mais combinações de bits do que caracteres de informação é para deteção e/ou correção de erros em transmissão⁽²⁾⁽³⁾. A idéia é acrescentar bits às palavras, aproveitando-se essa redundância para diminuir o efeito de erros no veículo, meio de armazenamento ou transmissão. Códigos de correção de erros simples (um bit só da palavra errada) têm a seguinte propriedade: cada palavra do código que dista uma unidade Hamming do carácter codificador corretamente está no conjunto que implica o carácter em questão. Por exemplo, se o código 101010 significa o dígito "1" num código de correção de erro simples, então 101011 é também interpretado como "1". Em termos do mapa de Karnaugh, os "vizinhos" das combinações corretas também são interpretados como tais combinações.

Para aplicações de códigos em sistemas digitais, recomendamos a referência⁽⁴⁾.

2.2 NÚMEROS

a. Sistemas

Um número binário é uma cadeia de bits em que cada posição tem um "peso". O peso é o valor associado com o bit quando no estado "1". O valor do número é o valor da soma dos pesos correspondentes às posições dos bits no estado "1". Para números binários, o valor de cada posição é 2 (a base do sistema) levada a uma potência inteira. Analogamente,

para números num sistema decimal, os pesos das posições são as potências integrais de dez (1, 10, 100, etc.). Esse tipo de esquema é chamado *notação posicional*.

Para números positivos e inteiros, o valor da representação é calculado na forma:

$$N = A_0 B^n + A_1 B^{n-1} + \cdots + A_{n-2} B^2 + A_{n-1} B + A_n. \quad (2-1)$$

Nessa equação, B é a base, sendo 2 para o sistema binário. O dígito A_i é coeficiente da posição de peso B^{n-i} . No caso de base dez, ou seja, números decimais, o coeficiente A_i é um dígito entre 0 e 9.

Os coeficientes A_i são agrupados numa cadeia para representar o número em notação posicional; conhecendo-se a base, não é necessário indicar os elementos B^i . Convém, quando há dúvida sobre a base, escrever a base como subíndice do número. Assim, $(110)_{10}$ significa "cento e dez" enquanto que $(110)_2$ significa "seis". Também $(6)_{10}$ e $(110)_2$ são duas representações ou *numerais* diferentes para o mesmo número (seis).

EXEMPLOS

O número decimal 946 [representado $(946)_{10}$] tem o valor de $9 \times 100 + 4 \times 10 + 6 = 900 + 40 + 6 = 946$. O número binário $(010110)_2$ tem o valor $0 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 16 + 4 + 2 = (22)_{10}$.

Quando o sistema de números é binário, não há dúvida de que a seleção do código dos dígitos está entre "0" ou "1". Mas, quando o sistema é decimal, existem várias opções para se representarem os dígitos 0 a 9. Isso porque os dispositivos de sistema digitais são binários e não decimais; por exemplo, um transistor conduzindo corrente (saturado) ou não-conduzindo (cortado). Então precisamos representar os dígitos decimais como combinações de sinais binários. O código mais óbvio para essa tarefa é a própria representação do dígito em binário. Essa representação é chamada *BCD* (*binary coded decimal*), que significa "decimal, codificado em binário".

Uma outra representação que foi popular na primeira e segunda gerações, é o código *BCD* acrescentado de três. Assim, três em binário $(0011)_2$ representa o dígito decimal zero, e doze em binário $(1100)_2$ representa o dígito decimal nove. Esse código é chamado *XS-3* (*excess 3*), que significa "três em excesso". Esses códigos são mostrados na Tab. 2-1.

Para simplificar a impressão de números binários, é costume aproveitar-se o dígito *octal* (3 bits) ou *hexadecimal* (4 bits). No sistema octal, os bits "000" a "111" são transcodificados para "0" a "7". No sistema hexadecimal, os bits "0000" a "1001" são transcodificados "0" a "9" e os bits "1010" a "1111" são transcodificados "A" a "F".

Tabela 2-1. Códigos decimais

Dígito decimal	Código BCD ^[*]	Código XS-3
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

[*] Combinações "ilegais" em BCD: 1010, 1011, 1100, 1101, 1110 e 1111

b. Números inteiros

A representação de frações usando-se p-tada como 1×10^{-n} da posição depende da representação depende da 017890 é dezenas e o quarto dígi

c. Números negativos

Para indicar que lado esquerdo da ampla negativos é chamado o sinal é vizinho da

Para números binários, com "0" indica

Existem mais dígitos. A grandeza de (1) complemento de base diminuída teóricas para o émico foram adotadas, comunica do computador.

Vamos super à nossa disposição. Em com o bit do sinal mais 2^{n-1} ou menos negativos, quando o tadas em binário. A é "1".

d. Números binários

Sendo $(+N)$ um em complemento de inclusive o bit de s

EXEMPLO. $n = 4$

A operação obviamente Sendo N número m

EXEMPLO. $n = 4$

Uma desvantagem da para o número 0, o +

EXEMPLO

Se $n = 4$ (4 bits), ent

b. Números inteiros e fracionários

A representação de números inteiros positivos pode ser estendida para representação de frações usando-se potências negativas da base. Por exemplo, a fração $(0,175)$ é representada como $1 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$. Assim para os números inteiros, o peso da posição depende da sua localização, ou seja, o valor de um número nesse tipo de representação depende da localização do ponto da base. Por exemplo, o valor da cadeia de dígitos 017890 é dezessete e oitenta e nove centésimos quando o ponto decimal está entre o terceiro e o quarto dígito da palavra.

c. Números negativos

Para indicar que um número é negativo, normalmente escrevemos o símbolo – ao lado esquerdo da amplitude do número, como -39 . Esse tipo de representação de números negativos é chamado de *sinal e amplitude*. Quando usado em sistemas decimais, às vezes o sinal é vizinho do dígito menos significativo, como no computador IBM 1401.

Para números binários, o sinal é normalmente o *bit* na posição mais significativa da palavra, com “0” indicando + e “1” indicando –.

Existem mais duas representações para sistemas de números encontrados em computadores. A grandeza de número será representada diferentemente para números negativos: (1) complemento de base (*complemento de dois* para o sistema binário) e (2) complemento de base diminuída (*complemento de um* para o sistema binário). Não tente procurar razões teóricas para o êxito dessas duas representações em sistemas digitais, pois as representações foram adotadas, como veremos, para simplificar e economizar circuitos da unidade aritmética do computador.

Vamos supor que a palavra binária disponha de n bits. Então temos 2^n combinações à nossa disposição. Em termos grosseiros, metade das combinações será negativa (a metade com o bit do sinal no estado “1”), então, com n bits, podemos representar números entre mais 2^{n-1} ou menos 2^{n-1} . No sistema binário para todas as três representações de números negativos, quando o bit de sinal é “0” as grandezas dos números de 0 a $2^{n-1} - 1$ são representadas em binário. A única diferença entre as representações acontece quando o bit de sinal é “1”.

d. Números binários em complemento de um

Sendo $(+N)$ um número binário de n bits entre 0 e $2^{n-1} - 1$, a representação de $(-N)$ em complemento de um é $[(+N)_2]'$ onde $[]'$ indica que cada bit de $(+N)$ é complementado, inclusive o bit de sinal.

EXEMPLO. $n = 4$

$$N = (+4)_{10} = (0100)_2, -N = (1011)_2.$$

A operação obviamente é a mesma na passagem de um número negativo para positivo. Sendo N número negativo em complemento de 1, a representação de $-N$ é $[(N)_2]'$.

EXEMPLO. $n = 4$

$$N = (-5)_{10} = (1010)_2, -N = (0101)_2 = (+5)_{10}.$$

Uma desvantagem da representação complemento de um é que existem duas representações para o número 0, o +0 e o -0. O problema é chamado de *ambigüidade de zero*.

EXEMPLO

Se $n = 4$ (4 bits), então zero = 0000. Fazendo o complemento de um de zero, obtemos 1111.

Alguns sistemas evitam o chamado “problema de menos zero” por hardware. A deteção da combinação -0 como resultado de qualquer operação aritmética provoca uma conversão imediata para $+0$.

Outros sistemas evitam -0 pela técnica de complemento da base (complemento de dois em nosso caso), que é bastante semelhante com complemento de um, gozando das mesmas vantagens (e até mais) em hardware, mas evitando a combinação -0 .

e. Números em complemento de dois

Sendo N uma palavra de n bits, representando um número entre 0 e $2^{n-1} - 1$, $-N$ na representação de complemento de dois é dado por $[(N)_2]' + (1)_2$. O complemento de dois de um número binário é, então, o complemento de um do número acrescentado de um.

EXEMPLO. $n = 4$

$$\begin{aligned} N &= (+4)_{10} = (0100)_2, \\ -N &= (1011)_2 + (1)_2 = (1100)_2 = (-4)_{10}. \end{aligned}$$

Observa-se que aqui também, complemento de dois do complemento de dois de um número é o próprio número. Em resumo, usando-se n bits, a soma binária (não se tratando o bit de sinal como exceção) de N e o seu complemento de um são $11\dots1$ (n bits) = $2^n - 1$.

EXEMPLO. $n = 4$

$$\begin{aligned} N &= (+4)_{10} = 0100, \\ N + (-N) &= 0100 + 1011 = 1111. \end{aligned}$$

Usando-se n bits, a soma de N e o complemento de dois N dá $100\dots0$ ($n+1$ bits) = 2^n .

EXEMPLO. $n = 4$

$$\begin{aligned} N &= (+4)_{10} = 0100, \\ N + (-N) &= 0100 + 1100 = 10000. \end{aligned}$$

Uma comparação das três representações de números negativos para $n = 4$ aparece na Tab. 2-2.

Tabela 2-2. Números binários negativos

N	Sinal e amplitude	Complemento de um	Complemento de dois
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0010
+0	0000	0000	0001
-0	1000	1111	não existe
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	não existe	não existe	1000

f. Ponto fixo e ponto flutuante
Em hardware, de bits. Muitas vezes é mais fácil processar palavras de ponto fixo, mas significativa operação de ponto flutuante é realizada utilizandomos multiplicador. Se a palavra é de ponto fixo, fornece um resultado final sempre que representa uma palavra com 16 bits usados para $\pm 32\,768$, ou seja, a escala de ponto flutuante é de 10^{-15} . Um número é representado por

Na equação $a \times b$ é a base e e é o expoente. A precisão em algoritmos de ponto fixo é normalmente deslocamentos de casa decimal.

EXEMPLO. Número

EXEMPLO. Número

O transbordamento é maior que a faixa quando o resultado é maior que o resultado é menor.

No sistema IBM, os números têm 32 bits em vez de 64.

A mantissa é normalizada para que a parte inteira da mantissa para ponto fixo seja sempre 16, como deve ser.

f. Ponto fixo e ponto flutuante

Em hardware, as unidades aritméticas trabalham com palavras de um certo número de bits. Muitas vezes, o comprimento da palavra é igual ao da memória principal. É mais fácil processar palavras em que o ponto de base não aparece explicitamente. Para a representação de *ponto fixo*, o ponto de base fictício fica num lugar fixo, ou à esquerda da posição mais significativa (fração) ou à direita da posição menos significativa (inteiro). Para somar ou subtrair em ponto fixo, a localização do ponto de base não faz diferença, porém, quando utilizamos multiplicação e divisão, o resultado depende da localização do ponto de base. Se a palavra é de 16 bits, a operação de multiplicação de dois operandos (de 16 bits cada) fornece um resultado bruto de 32 bits. Se os operandos forem inteiros, o resultado final será a palavra de 16 bits menos significativa. Se a palavra de 16 bits mais significativa não for zero, terá havido transbordamento (*overflow*). No caso dos operandos serem frações, o resultado final será a palavra de 16 bits mais significativas; a palavra menos significativa representa uma perda de precisão através de truncamento. A dificuldade na utilização do ponto fixo é a faixa (*range*) limitada de números que podem ser representados. Por exemplo, com 16 bits usando um bit para o sinal, só é possível representar números entre $\pm 2^{15}$ (ou $\pm 32\,768$), ou frações entre $\pm 2^{-15}$. O problema é o da mudança de escala, e a representação de ponto flutuante é utilizada para superar esse problema.

Um número N em ponto flutuante tem o valor

$$N = m \times b^e \quad (2-2)$$

Na equação anterior, m representa a *mantissa*, que pode ser ou inteira ou fracionária, b é a *base* e e é o *expoente*. Observar que ambos, mantissa e expoente, possuem sinal. A precisão em algarismos significativos é a precisão da mantissa. A faixa depende da base e do expoente. Se a mantissa é fracionária e o dígito menos significativo é não-zero, a representação é *normalizada*. Números não-normalizados podem ser normalizados através de deslocamentos à esquerda da mantissa e uma diminuição do expoente.

EXEMPLO. Números normalizados

$$\begin{aligned} 2,1416 &= 0,21416 \times 10^{+1}, \\ 1\,024 &= 0,1024 \times 10^4, \\ 0,000985 &= 0,985 \times 10^{-3}. \end{aligned}$$

EXEMPLO. Números não-normalizados com normalização

$$\begin{aligned} 0,00123 \times 10^6 &= 0,123 \times 10^4, \\ 0,0370 \times 10^0 &= 0,370 \times 10^{-1}. \end{aligned}$$

O transbordamento do expoente acontece quando o resultado de uma operação é maior que a faixa permitida. Acontece também uma situação, chamada *underflow*, em que o resultado é não-zero, mas menor do que pode ser representado.

No sistema IBM S/360, o formato para números em ponto flutuante “curto” (de 32 bits em vez de 64), é visto na Fig. 2-2.

A mantissa é considerada um número de 6 dígitos em base 16. Então os deslocamentos da mantissa para fins de normalização são de 4 bits por vez. A base do expoente é também 16, como deve ser. O expoente dispõe de 7 bits, que pode representar expoentes entre 0 e

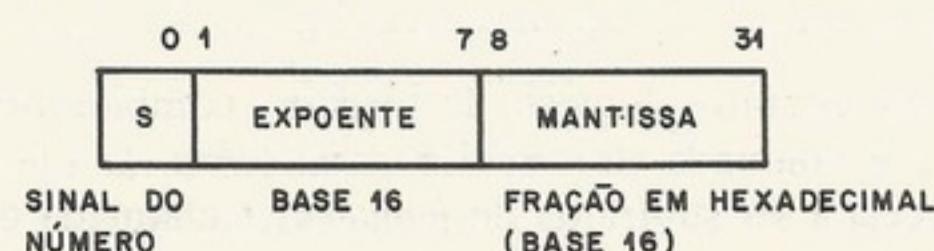


Figura 2-2. O formato de ponto flutuante curto no S/360

127. Contudo o expoente poderá também ser negativo. Para esse fim, o expoente $(0000000)_2$ representa -64 (10000000_2) e $(1111111)_2$ como $+63$. A faixa do expoente binário é, então, deslocada -64 . A faixa de variação da grandeza dos números normalizados em ponto flutuante curto do IBM S/360 vai de 16^{-65} a $(1 - 16^{-6}) \times 16^{63}$. Resultados não-nulos menores que 16^{-65} sofrem *underflow* do expoente.

Em muitas unidades centrais de sistemas não sofisticados, o conjunto de instruções só dispõe de adição e subtração em ponto fixo. Cabe ao programador, ou às sub-rotinas especiais, a implementação de multiplicação, divisão e aritmética em ponto flutuante.

2.3 ARITMÉTICA BINÁRIA

a. Números positivos

A soma de $(12)_{10}$ e $(26)_{10}$ é $(38)_{10}$, mostrada em binário na Fig. 2-3.

	32	16	8	4	2	1	PESO DA POSIÇÃO
	1	1					TRANSPORTE
12 =	0	0	1	1	0	0	
26 =	0	1	1	0	1	0	
38 =	1	0	0	1	1	0	

Figura 2-3. A soma de $(12)_{10}$ e $(26)_{10}$

Examinando cada posição, começando com a posição menos significativa (de peso 1), veremos que as parcelas “0” e “0” dão bit de soma igual a “0” e não têm transporte. As duas posições seguintes (de peso 2 e 4) indicam que as parcelas “0” e “1” resultam um bit de soma de “1”, mas não têm transporte. A próxima posição (peso 8) mostra que as parcelas “1” e “1” resultam um bit de soma de “0”, mas agora o transporte é “1”. O transporte dessa posição é chamado de “vai um” (*carry-out*) dessa posição. Esse transporte influí na soma da posição vizinha onde é chamado de “vem um” (*carry-in*). Agora, com as parcelas de “0” e “1” mais o “vem um”, o bit de soma dessa posição (peso 16) é “0”, mas o transporte é propagado à próxima posição (peso 32). As regras de adição de números positivos em binário para uma posição determinada, seguem a tabela da verdade da Tab. 2-3.

Tabela 2-3. Tabela da verdade, adição binária

Primeira parcela	Segunda parcela	“Vem um”	Soma	“Vai um”
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

A implementação em circuitos lógicos do circuito combinatório de três entradas e duas saídas da Tab. 2-3 é chamada de circuito *soma completa* (*full adder*).

Na subtração, a parcela a ser subtraída do minuendo é chamada de *subtraendo*. O transporte é chamado de *emprestimo*, mas continuaremos a usar “vai um” e “vem um”. A tabela de verdade para subtração é mostrada na Tab. 2-4.

Minuendo
0
0
0
0
1
1
1
1

Observamos que o resultado é completo. A diferença é a seguir.

Sabemos que com números de 6 dígitos, as Tabs. 2-3 e 2-4 servem para negativos precisamente.

b. A técnica da subtração

No tratamento da subtração usaremos a operação de complementação. Por exemplo, se o subtraendo é $(B)_{10}$, o complemento é $(-B)_{10}$.

A aritmética binária é feita da mesma forma.

Nessa equação, “[...],” e 2.2(e)] o número é negativo de A (número menor).

É importante que o bit significativo da subtração seja sempre 1.

EXEMPLO. $x = 10101010$

Podemos mostrar que

Tabela 2-4. Tabela da verdade, subtração binária

Minuendo	Subtraendo	"Vem um"	Diferença	"Vai um"
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Observamos que a função diferença é o *exclusive OR* das três entradas, como no somador completo. A diferença entre um minuendo positivo menor que um subtraendo é mostrada a seguir.

$$\begin{array}{r}
 & & 1 & 1 & & 1 & \\
 & & \text{12} = & 0 & 0 & 1 & 1 & 0 & 0 \\
 & & 26 = & 0 & 1 & 1 & 0 & 1 & 0 \\
 & & & \hline
 & & 1 & 1 & 0 & 0 & 1 & 0
 \end{array}
 \quad \text{← EMPRÉSTIMO}$$

Figura 2-4. A diferença de $(12)_{10}$ e $(26)_{10}$

Sabemos que a diferença é $(-14)_{10}$ e notamos que $(-14)_{10}$ em complemento de dois com números de 6 bits é $(110010)_2$, como aparece na Fig. 2-4. Isso significa que as Tabs. 2-3 e 2-4 servem para números positivos com resultados positivos e, para tratar com números negativos precisamos derivar as respectivas regras de operação.

b. A técnica da complementação

No tratamento de adição e subtração com números positivos e negativos, é costume usar-se a operação de adição (através de um somador) junto com a operação de complementação. Por exemplo, para subtrair B de um número A , basta somar A a B com o sinal trocado ($-B$):

$$A - B = A + (-B). \quad (2-3)$$

A aritmética binária de n bits é baseada na equação

$$A + [A]' = 2^n - 1 = \underbrace{11\dots1}_n \quad (2-4)$$

Nessa equação, "11...1" significa uma cadeia de n 1 e $[A]'$ indica [como nas Secs. 2.2(d) e 2.2(e)] o número binário A com todos os bits complementados. Observar que $[A]'$ é o negativo de A (ou seja, $-A$) em complemento de um.

É importante observar que o bit de sinal também entra nos cálculos como o bit mais significativo da palavra.

EXEMPLO. $n = 6$

$$\begin{aligned}
 A &= 000110 \\
 [A]' &= 111001 \\
 A + [A]' &= \underline{111111} = 2^6 - 1.
 \end{aligned}$$

Podemos mostrar um corolário,

$$[A]' = 2^n - 1 - A. \quad (2-5)$$

c. Aritmética usando complemento de um do subtraendo

O corolário (equação anterior) é utilizado para se fazer economicamente subtração com equipamento para somar que possua a capacidade de complementar o subtraendo bit a bit (complemento de um).

Suponha que números A e B são positivos, de 6 bits cada [1 bit (o mais significativo) de sinal e 5 bits de grandeza] e desejamos obter a diferença $A - B$ (que chamaremos de R) usando a operação de adição. Aplicando as Eqs. (2-3) e (2-5),

$$R = A + [B]' = A + 2^n - 1 - B. \quad (2-6)$$

Notamos que $A + [B]'$ difere com $A - B$ pela quantia $(2^n - 1) = 11\dots1$. Se a diferença for positiva, o resultado de $A + [B]'$ virá acompanhado por um “vai um transbordo” que tem peso 2^n ou, nesse caso, $(64)_{10}$, e então os 6 bits de R representam $A - B - 1$. O “vai um transbordo” acontece porque o bit de sinal de A é “0” (pois A é positivo) e o bit de sinal de $[B]'$ é “1” (pois B é positivo e o bit de sinal foi complementado). Então, se o sinal do resultado é “0” através de uma soma, só poderia ter ocorrido um “vem um” na posição do sinal. Assim sendo, ocorreu um “vai um transbordo” (veja a Fig. 2-5). Para se corrigir o resultado, é necessário somar mais um na posição menos significativa (veja a Fig. 2-6).

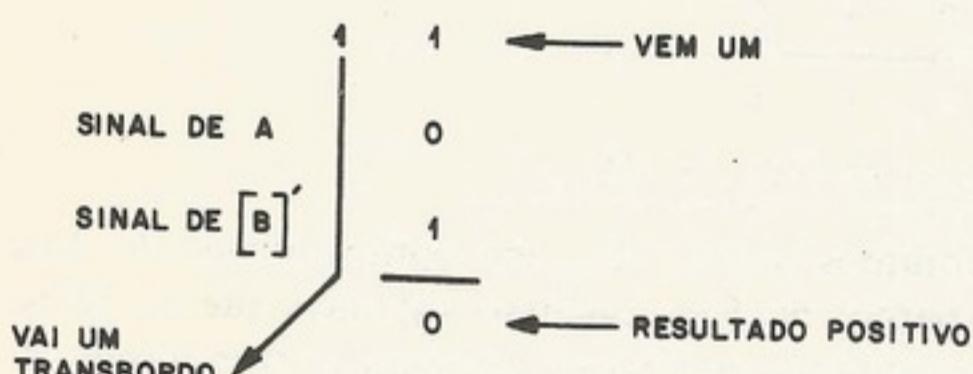


Figura 2-5. “Vai um transbordo” durante subtração usando somador e complementação

Figura 2-6. Exemplo de subtração em complemento de um

$$\begin{array}{r}
 A = 0\ 0\ 1\ 1\ 0\ 1 \quad (13)_{10} \\
 B = 0\ 0\ 1\ 0\ 1\ 1 \quad (11)_{10} \\
 \text{COMPLEMENTAÇÃO: } [B]' = 1\ 1\ 0\ 1\ 0\ 0 \\
 A + [B]' = 0\ 0\ 1\ 1\ 0\ 1 \\
 \hline
 & 1\ 1\ 0\ 1\ 0\ 0 \\
 & \hline
 & 0\ 0\ 0\ 0\ 0\ 1 \\
 + & \hline
 & 1 \quad \text{MAIS "1"} \\
 \text{RESULTADO CERTO} \longrightarrow & 0\ 0\ 0\ 0\ 1\ 0 \quad = (2)_{10}
 \end{array}$$

Vamos supor agora que o resultado seja negativo. Se B for maior que A , então $B - A$ será um número positivo e o resultado $R = A - B$ deve ser o complemento de um de “ $B - A$ ”; ou seja, deve ser $[B - A]'$. Segundo o Corolário (2-5), obtemos a Eq. (2-7), que é igual à Eq. (2-6).

$$R = [B - A]' = 2^n - 1 - B + A = A + 2^n - 1 - B. \quad (2-7)$$

Então, quando o resultado é negativo, não há “vai um transbordo” e o resultado não necessita correção (veja a Fig. 2-7).

Resumindo, durante subtração, quando não há “vai um transbordo”, o resultado está certo; caso contrário, é necessário somar-se um ao resultado. Esse processo, na implementação, é chamado de realimentação ou *retorno de transporte* (*end-around carry*). Assim, o sinal de “vai um transbordo” sempre alimenta o “vem um” da posição menos significativa do somador. Quando o “vai um transbordo” é “0”, nada acontece e, quando o “vai um transbordo” é “1”, o resultado é corrigido.

EXEMPLOS

Foi mostrado que o circuito de subtração é similar ao de adição, com exceção do processo de adição de sinal.

EXEMPLO. Resultado de subtração

Neste exemplo, obtemos o resultado de subtração de dois números positivos.

A situação é descrita na Figura 2-6.

EXEMPLO. Resultado de subtração

Neste exemplo, obtemos o resultado de subtração de dois números positivos.

Observe que os resultados obtidos são os mesmos que os resultados obtidos para subtração de números positivos em complemento de dois.

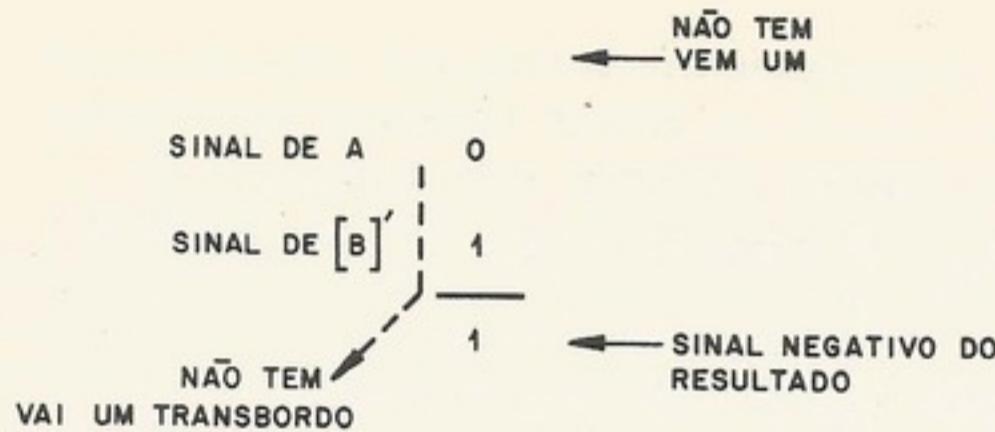


Figura 2-7. Subtração de dois positivos sem “vai um transbordo”

(2-6)

EXEMPLOS

$$\begin{array}{r}
 1111 \\
 A = 001101 = (13)_{10}, \quad B = 010000 = (16)_{10} \\
 [B]' = 101111 \\
 \hline
 111100 \\
 \text{returno: 0} \\
 \hline
 111100 = (-3)_{10}
 \end{array}$$

$$\begin{array}{r}
 1111 \\
 A = 001101 = (13)_{10}, \quad B = 001010 = (10)_{10} \\
 [B]' = 110101 \\
 \hline
 000010 \\
 \text{returno: 1} \\
 \hline
 000011 = (+3)_{10}.
 \end{array}$$

Foi mostrado que o somador binário pode servir para subtração. Tendo-se em vista que o circuito do somador foi projetado para adicionar números positivos, será que esse processo de adição binária serve para somar um número positivo com um número negativo?

EXEMPLO. Resultado negativo

$$\begin{array}{r}
 1 \\
 A = 000111 \quad (7)_{10} \\
 B = 110100 \quad (-11)_{10} \\
 \hline
 111011 \quad (-4)_{10}.
 \end{array}$$

Neste exemplo, observar que o processo não deu “vai um transbordo” e o resultado é válido. A situação é descrita pela Eq. (2-7).

EXEMPLO. Resultado positivo

$$\begin{array}{r}
 1111 \\
 A = 001111 \quad (15)_{10} \\
 B = 110100 \quad (-11)_{10} \\
 \hline
 000011 \quad (3)_{10} \quad \text{resultado intermediário} \\
 \text{returno: 1} \\
 \hline
 000100 \quad (4)_{10} \quad \text{resultado certo.}
 \end{array}$$

Neste exemplo, houve “vai um transbordo”, corrigido pelo retorno do transporte.

Observe que esse processo funciona tanto para adição quanto para a subtração de números positivos ou números negativos em representação complemento de um.

EXEMPLO

$$\begin{array}{r}
 (-9)_{10} - (3)_{10} \\
 1111 \\
 A = 110110 = (-9)_{10} \quad B = 000011 = (3)_{10} \\
 [B]' = \underline{111100} \\
 110010 \\
 \text{returno: 1} \\
 \hline
 110011 = (-12)_{10}.
 \end{array}$$

Acontece que, com esse esquema, subtraindo-se um número de si próprio, resulta -0 .

EXEMPLO

$$\begin{array}{r}
 (-9)_{10} - (-9)_{10} \\
 A = 110110 \\
 [A]' = 001001 \\
 \hline
 111111 \\
 \text{returno: 0} \\
 \hline
 111111 = (-0).
 \end{array}$$

Existe ainda um problema. Somando-se dois números de sinais iguais ou tomando-se a diferença de dois operandos com sinais diferentes, existe a possibilidade de a grandeza do resultado ser maior que $2^n - 1$ [ou menor que $-(2^n - 1)$]. Essa situação é chamada de estouro ou transbordamento (*overflow*). Existe uma regra simples para se descobrir um estouro: é quando o “vem um” que entra na posição do sinal não é igual ao “vai um transbordo”.

EXEMPLO. A e B positivos

$$\begin{array}{r}
 1 \\
 A = 010000 = + (16)_{10} \\
 B = + 010000 = (16)_{10} \\
 \hline
 100000 = (-31)_{10} \text{ (errado).}
 \end{array}$$

Neste exemplo, o “vem um” é igual a 1 e o “vai um transbordo” é igual a “0”. Sendo A e B negativos, o estouro acontece quando o “vem um” é igual a “0” e o “vai um transbordo” é igual a “1”, como no exemplo seguinte.

EXEMPLO. A e B negativos

$$\begin{array}{r}
 1 \\
 A = 100000 = (-31)_{10} \\
 B = + 110111 = (-8)_{10} \\
 \hline
 010111 \\
 1 \\
 \hline
 011000 = (+24)_{10} \text{ (errado).}
 \end{array}$$

Infelizmente, a regra não funciona em um caso: quando uma parcela é -0 e a outra é $-2^{n-1} + 1$. O exemplo com $n = 6$ segue.

EXEMPLO. Problema com -0

$$\begin{array}{r}
 1 \\
 100000 = (-31)_{10} \\
 111111 = (-0)_{10} \\
 \hline
 011111 \\
 1 \\
 \hline
 100000
 \end{array}$$

Neste exemplo, normalmente o fluxograma é mostrado na forma

Figura

d. Aritmética

Nesta seção, simplesmente o bit.

Os processos dois são quase implemento que é

Neste exemplo, deu sinal de transbordamento (*overflow*) apesar de certo. Esse problema é normalmente resolvido através de circuitos especiais para evitar o -0 .

O fluxograma para aritmética (adição e subtração) binária em complemento de um é mostrado na Fig. 2-8.

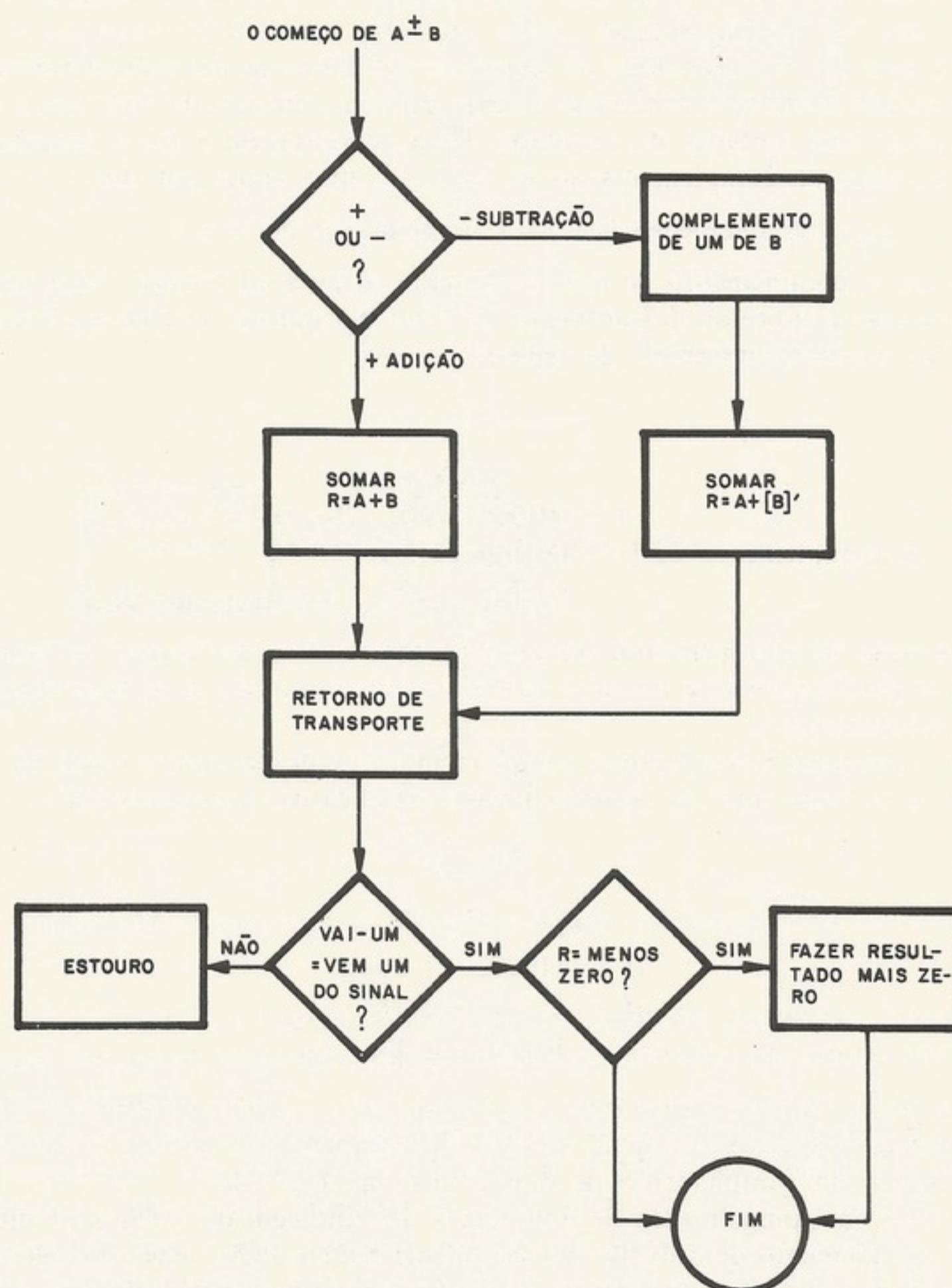


Figura 2-8. Fluxograma de aritmética binária em complemento de um

d. Aritmética usando complemento de dois

Nesta seção, como na anterior, queremos tratar a aritmética sob o aspecto de utilizar-se simplesmente o processo de adição binária e o processo de complementar um número *bit a bit*.

Os processos aritméticos com números negativos representados em complemento de dois são quase iguais aos do complemento de um. Uma diferença existe na equação do complemento que é alterada para

$$[A]' = 2^n - A. \quad (2-8)$$

Agora, para subtrair fazendo a soma com o minuendo e o complemento do subtraendo, obtemos

$$R = A + [B]' = A + (2^n - B) = A - B + 2^n. \quad (2-9)$$

Se $A - B$ for positivo, então 2^n mais uma quantidade 2^n irá estourar a capacidade de n bits, que significa um "vai um transbordo". Caso contrário, se $A - B$ for negativo, o resultado é $2^n - (B - A)$, que é a representação da grandeza da diferença em complemento de dois, ou seja, o resultado correto. A aritmética binária em complemento de dois, tem a vantagem de não necessitar uma correção do resultado. Uma desvantagem é que o complemento de dois necessita, além da complementação de bit a bit, uma soma por um,

$$(\text{complemento de } 2) = (\text{complemento de } 1) + 1. \quad (2-10)$$

Na prática, o complemento de dois é efetuado através da técnica "vem um quente" (*hot carry*), durante o processo de subtração. O "vem um quente" (*VUQ*) é o "vem um" da posição do bit menos significativo da palavra.

EXEMPLO

$$\begin{array}{r} & 1 \leftarrow VUQ \\ A = 001000 & = (8)_{10} \quad B = 001011 = (11)_{10} \\ B \text{ (complemento de 1)} = 110100 \\ \hline 111101 & = (-3)_{10} \text{ (complemento de 2).} \end{array}$$

A combinação de complementar bit a bit o subtraendo e de "forçar" o "vem um" da posição menos significativa (que possui o peso de um) tem o efeito de se fazer complemento de dois do subtraendo.

Quando o processo é de adição, a complementação é desnecessária e não há "vem um quente", isto é, o "vem um" da posição menos significativa permanece "0".

EXEMPLO

$$\begin{array}{r} 11 \\ A = 110000 = (-16)_{10} \\ B = 111111 = (-1)_{10} \\ \hline 101111 = (-17) \end{array}$$

No exemplo, o "vai um transbordo" é ignorado, pois é igual ao "vem um" da posição do sinal. A deteção de estouro é igual ao caso do complemento de um. O fluxograma de aritmética binária em complemento de dois é visto na Fig. 2-9.

A aritmética em complemento de dois tem a desvantagem de o valor de um número negativo em complemento de dois não ser facilmente reconhecido; mas a aritmética dispõe de certas vantagens sobre o complemento de um. Por exemplo, o problema de -0 não existe. O -0 pode ser inconveniente para instruções que testam o sinal porque o resultado será ambíguo se o número for zero. Além do mais, o -0 cria problemas quando somado com $(-2^{n-1} + 1)$. O processo do retorno do transporte é inconveniente quando a adição é feita seriamente em etapas; por exemplo, quando a palavra tem 16 bits e o somador é de 4 bits. Tendo retorno de transporte, o processo volta ao começo.

e. Aritmética com números em sinal e amplitude

A aritmética com números em sinal e amplitude pode aproveitar-se as técnicas de complemento de um ou de complemento de dois para subtração. Primeiro é importante observar que o processo de obtenção da diferença das grandezas ocorre quando se estão adicionando números de sinais diferentes ou subtraindo-se números de sinais iguais. Por

isso, antes da adição, as grandezas entram no complemento de um.

O estouro só ocorre quando se "vai um transbordo". Mas, quando se subtrai um número maior, o que se obtém é o complemento de dois no exemplo visto na aritmética.

EXEMPLO. Subtração

O processo é

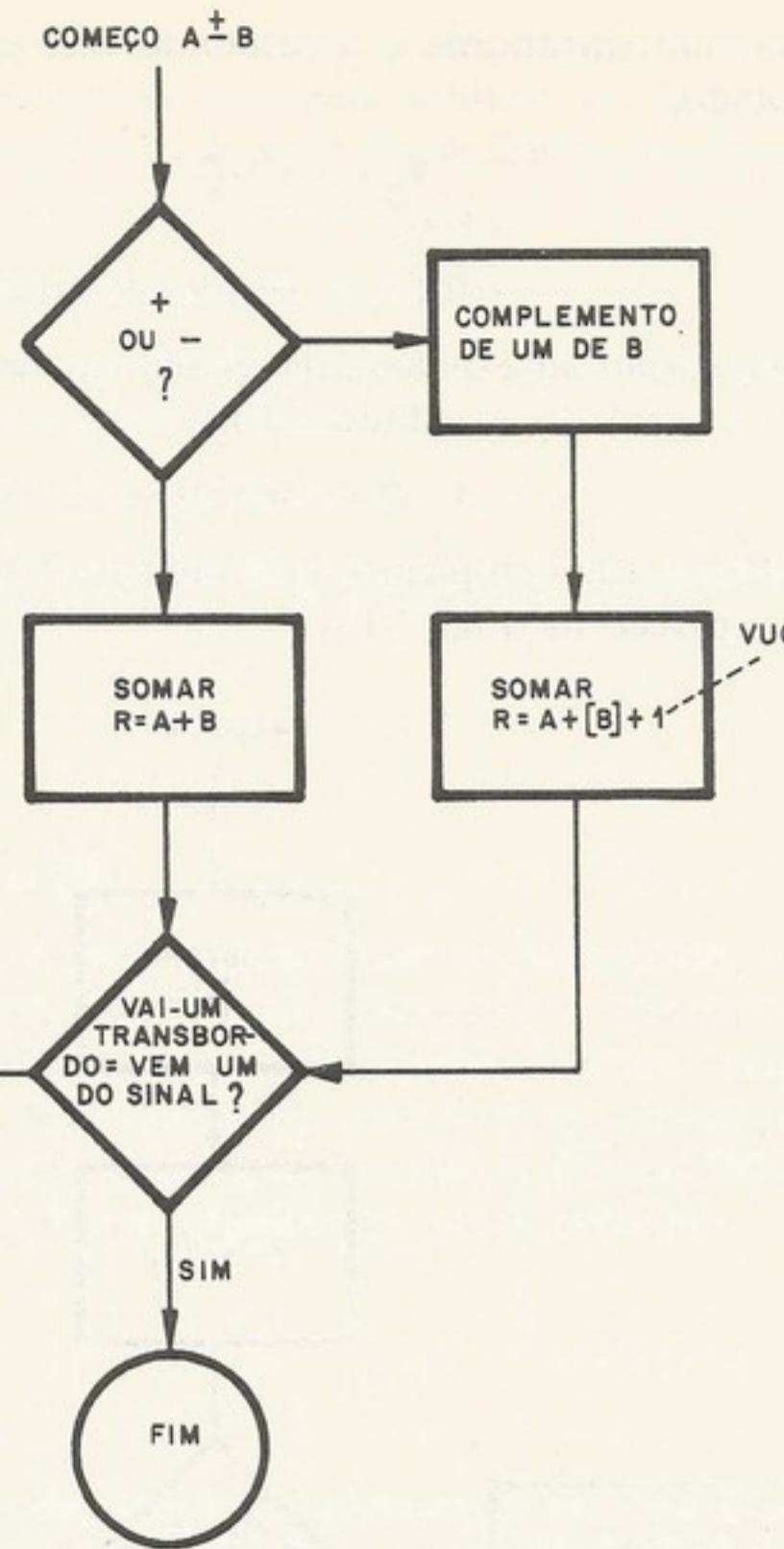


Figura 2-9. Fluxograma de aritmética binária em complemento de dois

isso, antes da aritmética, é necessário examinar os sinais das grandezas. Portanto somente as grandezas entram na aritmética (em contraste com sistemas de complemento de um ou complemento de dois).

O estouro só pode acontecer na obtenção da soma das grandezas, e é indicado pelo "vai um transbordo". Na obtenção da diferença entre grandezas, é impossível haver estouro. Mas, quando se subtrai uma grandeza da outra, pode ocorrer a necessidade do *recomplementar*, o que se indica pela ausência do "vai um transbordo". Usaremos complemento de dois no exemplo a seguir, com $n = 6$, onde o bit mais significativo é o sinal e não entra na aritmética.

EXEMPLO. Soma A e B , números em sinal e amplitude

$$A = 000011 = (3)_{10}, \quad B = 101101 = (-13)_{10}$$

O processo é realmente de subtrair, sendo B negativo,

$$\begin{array}{r} 11\textcircled{1} \leftarrow VUQ \\ A = 00011 \\ [B]' = \overline{10010} \\ \hline 10110 \end{array}$$

Faltando “vai um transbordo”, o resultado está em complemento de dois e tem de ser recomplementado,

$$\begin{array}{r} 1\bar{1} \leftarrow VUQ \\ 01001 \\ \hline 01010 = \text{amplitude } (10)_{10}. \end{array}$$

A recomplementação só é necessária quando o sinal da diferença difere do sinal do minuendo; então o sinal do resultado é “1”;

$$A - B = 101010 \leftarrow (-10)_{10}.$$

Também podemos usar complemento de um para subtrair números em sinal e amplitude. Esse fluxograma aparece na Fig. 2-10.

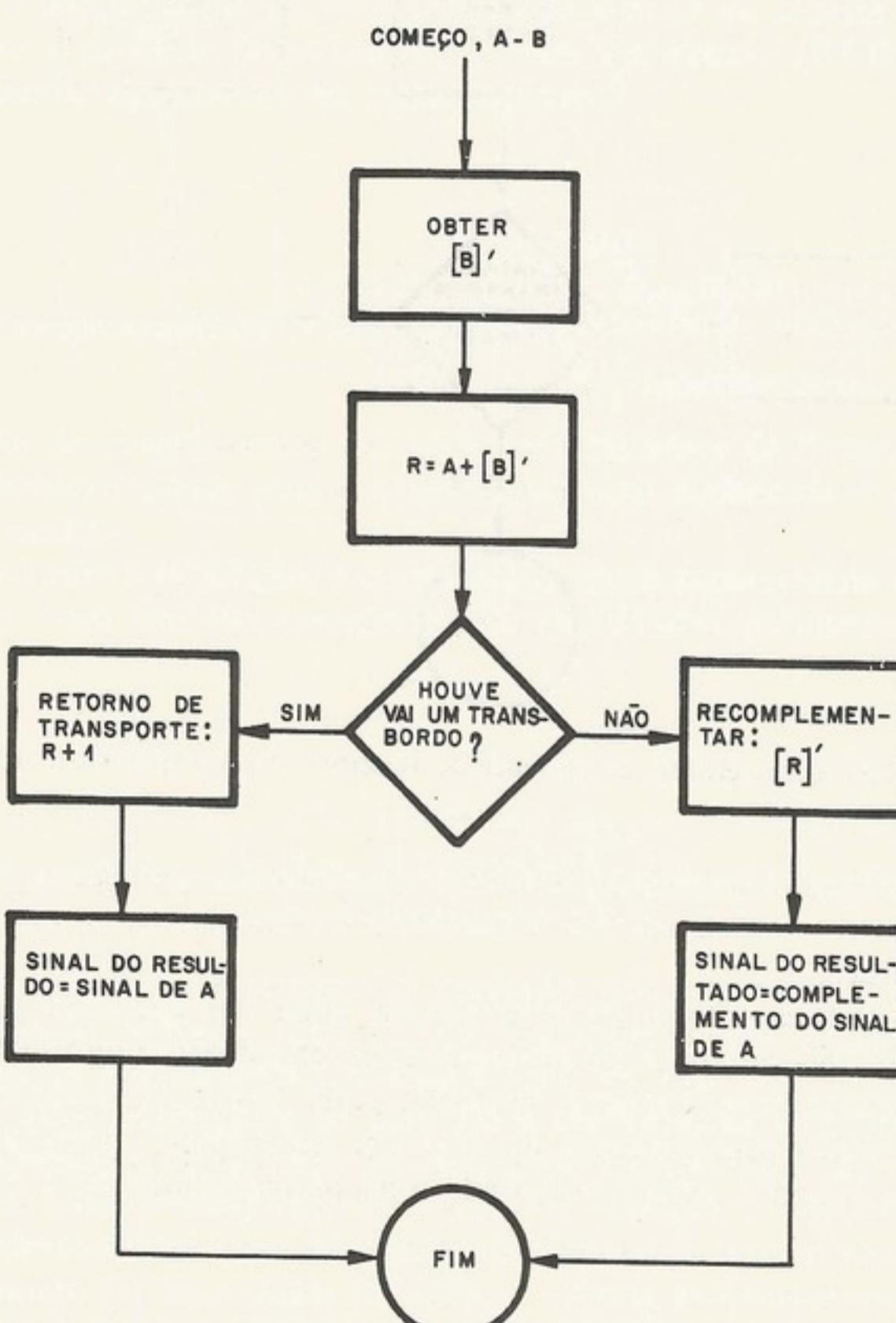


Figura 2-10. Subtração com números binários em sinal e amplitude usando-se complemento de um

f. Variações

Até agora, o nosso interesse foi aproveitar a adição binária com a técnica de complementar o subtraendo, para realizar subtrações.

Descobrimos que, com o processo de adição associado à complementação de um operando bit a bit, obtemos a generalidade de somar ou subtrair números positivos ou negativos. Obteremos essa mesma generalidade se usarmos a técnica de complemento do

minuendo⁽⁵⁾. Essa técnica pode ser complementada bit a bit A antes de ser desejado; então compõe

Não examinaremos vários exemplos, e

EXEMPLO

Também existe a bit. Para somar com subtraendo. Dependendo ou “retorno de empurrar” basta dizer que compõe com UCP projetada

2.4 ARITMÉTICA II

a. Representação

A aritmética com Existem as representações O complemento de um dígito, isto é, subtraendo dígito “6” é “3”, do dígi do código XS-3 (nega a bit do código puro) é “1001”. Essa propriedade goza dessa vantagem complementar um op

O complemento de um isso porque para um nove vale $(10^8 - 1 - 1)$ sistemas decimais, n

EXEMPLO

b. Exemplos de

Como em sistemas problema de -0 e da

minuendo⁽⁵⁾. Essa técnica é usada para decrementar por um a conta de um contador que pode ser complementado. Suponhamos que seja para subtrair $A - B$. Se complementarmos bit a bit A antes de somar, obtemos $R = (-A + B)$. O resultado parece ser o negativo do desejado; então complementarmos bit a bit o resultado,

$$-R = -(-A + B) = A - B. \quad (2-11)$$

Não examinaremos esse assunto em profundidade, mas o leitor pode experimentar com vários exemplos, e verificar as características da técnica.

EXEMPLO

$$\begin{array}{r} A = (001110)_2 = (14)_{10}, \quad B = (000101)_2 = (5)_{10} \\ \hline -A = 110001 \\ B = 000101 \\ \hline R = 110110 \\ -R = 001001 = (9)_2. \end{array}$$

Também existe generalidade se podemos subtrair e, juntamente, complementar bit a bit. Para somar com um subtrator, basta só complementar o operando que serve como subtraendo. Dependendo da representação de negativos, ou ocorrerá “emprestimo quente” ou “retorno de empréstimo”. Essa técnica também não será examinada em profundidade, basta dizer que computadores de médio porte (por exemplo, o Prodac 580 da Westinghouse com UCP projetada pela Univac) foram construídos neste princípio.

2.4 ARITMÉTICA DECIMAL

a. Representação dos negativos

A aritmética com números em decimal é completamente análoga à aritmética binária. Existem as representações sinal e amplitude, complemento de nove, e complemento de dez. O complemento de nove, análogo ao complemento de um, é feito complementando-se cada dígito, isto é, subtraindo-se cada dígito de nove. Por exemplo: o complemento de nove do dígito “6” é “3”, do dígito “0” é “9”, e do dígito “2” é “7”. Agora podemos ver uma vantagem do código XS-3 (veja a Tab. 2-1): o complemento de nove do dígito é o complemento bit a bit do código para o dígito. Por exemplo: o código de “3” é “0110” e o complemento “6” é “1001”. Essa propriedade do código é chamada *autocomplementação*. O código BCD não goza dessa vantagem, mas, como veremos, podemos também aproveitar a capacidade de complementar um operando em BCD bit a bit.

O complemento de dez de um número decimal é o complemento de nove mais um; isso porque para um número A de n dígitos (excluindo-se o sinal) o seu complemento de nove vale $(10^n - 1 - A)$ e seu complemento de dez vale $(10^n - A)$. O sinal, muitas vezes, em sistemas decimais, necessita um dígito para si.

EXEMPLO

$$\begin{aligned} A &= +103 \\ -A \text{ (complemento de 9)} &= -896 \\ -A \text{ (complemento de 10)} &= -897. \end{aligned}$$

b. Exemplos de aritmética decimal

Como em sistemas binários, a aritmética decimal em complemento de nove tem o problema de -0 e do “retorno do transporte” quando se utiliza o somador para subtrair.

Sistemas em complemento de dez, para subtração, podem aproveitar a técnica de fazer o complemento de nove para o subtraendo e forçar o "vem um quente" na posição menos significativa. Sistemas decimais usando a representação sinal e amplitude podem subtrair usando o complemento de dez ou complemento de nove do subtraendo, como se quiser, mas, de qualquer maneira, se o resultado for negativo, será necessário *recomplementá-lo*.

EXEMPLO. Complemento de nove

$$A = +0367, \quad B = -9831 = (-0168)_{10}$$

$$\begin{array}{r} A + B \\ \hline 1 \\ +0367 \\ -9831 \\ \hline +0198 \\ 1 \\ \hline +0199 \end{array} \quad (367 - 168 = 199).$$

O sinal pode ser tratado como $+ = 0$ e $- = 1$.

$$B - A$$

$$\begin{array}{r} 111 \\ -9831 \\ -9632 \\ \hline -9463 \\ 1 \\ \hline -9464 \end{array} \quad (\text{complemento de } 9 \text{ de } A)$$

$$(-168 - 367 = -535).$$

EXEMPLO. Complemento de dez

$$A = +0367, \quad B = -9832 = (-0168)_{10}$$

$$\begin{array}{r} A + B \\ \hline 111 \\ +0367 \\ -9832 \\ \hline +0199 \end{array}$$

$$B - A$$

$$\begin{array}{r} 111 \quad \textcircled{1} \leftarrow \text{"vem um quente" (VUQ)} \\ -9832 \\ -9632 \\ \hline -9465 \end{array} \quad (\text{complemento de } 9 \text{ de } A)$$

$$(-168 - 367 = -535).$$

EXEMPLO. Sinal e amplitude

$$A = +0367, \quad B = -0168$$

$A + B$: a operação é de subtração; então

$$\begin{array}{r} 11 \quad \textcircled{1} \leftarrow \text{"vem um quente" (VUQ)} \\ A = 0367 \\ \text{complemento de } 9 \text{ de } B = \frac{9831}{+0199}. \end{array}$$

Aqui, o "vem um" na posição do sinal indica que o sinal do resultado é o sinal do minuendo.

$$B + A$$

$$\begin{array}{r} 11\textcircled{1} \leftarrow \text{"vem um quente"} \\ 0168 \\ 9632 \\ \hline 9801 \end{array}$$

códigos, mámen-

A ausência di-

c. Arithme-

Como em

dos operandos,

das amplitudes

possibilidade di-

O sistema

"ilegais". Quan-

casos:

O caso I m-

quando o resul-

e 19. O caso II m-

tado. No caso

EXEMPLO

$A + B$

Primeira etapa

Para o caso

para o caso I.

Segunda etapa

resultado certo.

Uma despu-

"vai um" com o

EXEMPLO

Primeira etapa

A ausência do sinal indica que o sinal do resultado não é o sinal do minuendo, significando que o resultado tem de ser recomplementado:

$$\begin{array}{r} \text{complemento de 9: } \quad 0198 \\ + 0199 \\ \hline \text{resultado certo.} \end{array}$$

① ← “vem um quente”

c. Aritmética em BCD usando representação sinal e amplitude

Como em números binários para sinal e amplitude, é necessário examinar os sinais dos operandos, antes de fazer aritmética, para descobrir se a operação é basicamente a soma das amplitudes ou a diferença delas. No caso de somar amplitudes como sempre, tem a possibilidade de estouro de capacidade do resultado.

O sistema *BCD* usa 10 das combinações disponíveis com 4 bits, sendo as outras seis “ilegais”. Quando somamos dois dígitos *BCD* em binário, o resultado, pode estar em três casos:

- caso 1, dígito legal sem “vai um”;
- caso 2, dígito ilegal sem “vai um”;
- caso 3, dígito legal com “vai um”.

O caso 1 acontece quando o dígito do resultado é nove ou menos. O caso 2 acontece quando o resultado fica entre 10 e 15, e o caso 3 acontece quando o resultado cai entre 16 e 19. O caso 1 não precisa de correção, mas para os casos 2 e 3 precisamos corrigir o resultado. No caso 2, também precisamos propagar o “vai um”.

EXEMPLO

$$A = 0832, \quad B = 0983$$

A + B

Primeira etapa

$$\begin{array}{r} & & 1 \\ A = 0000 & 1000 & 0011 & 0010 \\ B = 0000 & 1001 & 1000 & 0011 \\ \hline 0001 & 0001 & 1011 & 0101 \\ \text{caso 3} & \text{caso 2} & \text{caso 1} \end{array}$$

Para o caso 2, precisamos somar 6, para corrigir o dígito e propagar o “vai um”; e, para o caso 3, só precisamos somar 6, pois o “vai um” já foi propagado.

Segunda etapa

$$\begin{array}{r} 1111 \quad 11 \\ 0001 \quad 0001 \quad 1011 \quad 0101 \\ 0110 \quad 0110 \\ \hline 0001 \quad 1000 \quad 0001 \quad 0101 \end{array}$$

resultado certo, 1815.

Uma desvantagem da correção do resultado dessa maneira é que o tratamento do “vai um” com o caso 2 pode criar correções não previstas na primeira etapa.

EXEMPLO

$$A = 0372, \quad B = 0633$$

A + B

Primeira etapa

$$\begin{array}{r} 11 \quad 111 \quad 1 \\ 0000 \quad 0011 \quad 0111 \quad 0010 \\ 0000 \quad 0110 \quad 0011 \quad 0011 \\ \hline 0000 \quad 1001 \quad 1010 \quad 0101 \\ \text{caso 2} \end{array}$$

Segunda etapa

$$\begin{array}{r}
 & 11 & 11 \\
 0000 & 1001 & 1010 & 0101 \\
 & & 0110 \\
 \hline
 0000 & 1010 & 0000 & 0101 \\
 \text{caso 2}
 \end{array}$$

Para evitar esse tratamento especial, existe um outro algoritmo (veja Hellerman⁽¹¹⁾), o da soma de 6 em todos os dígitos de um dos operandos antes da soma das duas parcelas. Assim, só existem dois casos, distinguidos pelo “vai um” do dígito:

- caso 1: o resultado não deu “vai um” e então caiu entre 6 e 15 – tem de subtrair 6;
 caso 2: o resultado deu “vai um”; então o “vai um” já foi propagado e o dígito está correto entre 0 e 9.

EXEMPLO

Primeira etapa

$$A = 0372, \quad B = 0633$$

$$\begin{array}{r}
 & 11 & 11 & 11 \\
 A = 0000 & 0011 & 0111 & 0010 \\
 \text{soma de 6} & 0110 & 0110 & 0110 \\
 \hline
 0110 & 1001 & 1101 & 1000
 \end{array}$$

Se a primeira etapa der “vai um” entre dígitos, o dígito original do operando terá sido ilegal.

Segunda etapa

$$\begin{array}{r}
 & 1 & 1111 & 111 \\
 A + 6 = 0110 & 1001 & 1101 & 1000 \\
 B = 0000 & 0110 & 0011 & 0011 \\
 \hline
 0111 & 0000 & 0000 & 1011 \\
 \text{caso 1} & \text{caso 2} & \text{caso 2} & \text{caso 1}
 \end{array}$$

Terceira etapa (em vez de subtrair “0110”, somamos “1010” e ignoramos o “vai um”).

$$\begin{array}{r}
 & 11 & & 1 \\
 & 0111 & 0000 & 0000 & 1011 \\
 & 1010 & & & 1010 \\
 \hline
 0001 & 0000 & 0000 & 0101
 \end{array}$$

resultado certo, 1005.

Para obter a diferença entre duas amplitudes decimais em *BCD*, podemos usar a técnica de complementar o dígito *BCD* bit a bit. Se o dígito do minuendo for maior que o dígito do subtraendo, a diferença em binário cairá entre 0 e 9 e não haverá problema de correção. Como no caso binário, o “vai um” transbordo indica que não precisa recomplementar.

EXEMPLO. Complemento de dez

$$A = 9865, \quad B = 3112$$

$$A - B$$

$$\begin{array}{r}
 & 1 & 11 & & 1 & 11 & 1 & 1 & 1 & 1 & 1 \\
 & A = & 1001 & & 1000 & 0110 & 0101 & & & & \\
 [B]' = & 1100 & 1110 & 1110 & 1101 & & & & & & \\
 & & 0110 & 0111 & 0101 & 0011 & & & & & \\
 & & 6 & 7 & 5 & 3 & & & & &
 \end{array}
 \begin{array}{l}
 \text{1} \text{ } \text{1} \leftarrow \text{“vem um quente”}
 \end{array}$$

O resultado 6753 está certo e não precisa de correção. Em cada posição, se o dígito do subtraendo for menor, a soma do dígito do minuendo e o complemento do dígito do subtraendo

códigos, nã... sempre dá “vai um que implementa...

EXEMPLO

Primeira etapa

Segunda etapa

resultado certo.
Nos casos em que precisamos subtrair

EXEMPLO

O resultado é certo.
precisamos subtrair

A Fig. 2-11 mostra os resultados em binário. Nós vimos que para subtrair números em símbolos devemos subtrair os dígitos de direita para esquerda, mantendo os resultados de soma e subtração.

2.5 CONVERSÃO DE CODIGOS

Em alguns sistemas de computação para fazer conversões entre diferentes tipos de dados, supomos que os dados estão em código BCD.

sempre dá “vai um” do dígito. Caso contrário, o dígito necessita a correção de subtrair “6”, que implementamos no exemplo pela técnica de somar dez $(1010)_2$ e ignorar o “vai um”.

EXEMPLO

$$A = 9865, \quad B = 3972$$

Primeira etapa

$$\begin{array}{r} & 1 & & 1 & 1 \\ A = & 1001 & 1000 & 0110 & 0101 \\ [B]' = & 1100 & 0110 & 1000 & 1101 \\ & 0101 & 1110 & 1111 & 0011 \end{array} \quad \text{①} \leftarrow \text{“vem um quente” (VUQ)}$$

Segunda etapa

$$\begin{array}{r} 1010 & 1010 \\ \hline 0101 & 1000 & 1001 & 0011 \end{array}$$

resultado certo, 5893

Nos casos em que não há “vai um transbordo”, o resultado está na forma complementar, e precisamos recomplementar.

EXEMPLO

$$\begin{array}{r} A = 3972, \quad B = 9865 \\ & 1111 & 1111 & 111 & 1 \\ A = & 0011 & 1001 & 0111 & 0010 \\ [B]' = & 0110 & 0111 & 1001 & 1010 \\ & 1010 & 0001 & 0000 & 1101 \\ & 1010 & & & 1010 \\ R = & 0100 & 0001 & 0000 & 0111 \end{array} \quad \text{①} \leftarrow \text{VUQ}$$

O resultado 4107 é o complemento de dez do resultado certo, 5893. Para recomplementar, precisamos subtrair 4107 de zero:

$$\begin{array}{r} & & & \text{①} \leftarrow \text{VUQ} \\ & 0000 & 0000 & 0000 & 0000 \\ [R]' = & 1011 & 1110 & 1111 & 1000 \\ & 1011 & 1110 & 1111 & 1001 \\ & 1010 & 1010 & 1010 & 1010 \\ & 0101 & 1000 & 1001 & 0011 \\ & 5 & 8 & 9 & 3 \end{array}$$

A Fig. 2-11 representa um resumo das técnicas de fazer-se aritmética *BCD* com os números em sinal e amplitude. Para demais informações sobre números e aritmética, recomendamos os textos das referências⁽⁶⁾ e⁽⁷⁾.

2.5 CONVERSÃO ENTRE NÚMEROS BINÁRIOS E DECIMAS

Em alguns sistemas, como o IBM S/360, existem instruções específicas em hardware para fazer conversões; mas a maioria dos sistemas usa sub-rotinas para esse fim. Aqui, supomos que os números sejam inteiros e que a representação dos dígitos decimais esteja em código *BCD*.

EXEMPLO. Conversão binária para decimal

a conversão:

Se não houver
(veja Couleur¹⁰). O
decimal pode ser

1. Começar com
2. Se o seguir
3. Se o seguir
4. Continuar

EXEMPLO

Começando com o bit é “1”. O terceiro bit é “1”; então temos 1. O sexto bit é “0” (dobrando, + 1). O processo é justificável.

$$A_0 2^{n-1} + \dots$$

Um somador de sinal e amplitude pode ser usado para apenas somar o resultado das parcelas. Durante a adição de número binário é implementado alternadamente de 4 bits cada. Cada combinação ilegal é de seis (0110_2). O resultado a ser convertido em um bit na posição do vizinho mais significativo até tratar o último.

No exemplo anterior, examinar e corrigir as sequências (veja Seções F8, F4, F2 e F1) entre 0000 e 1111 e o “vai um”. Essa figura da Fig. 1-24 no sentido melhor, pulso de sincronização interna do

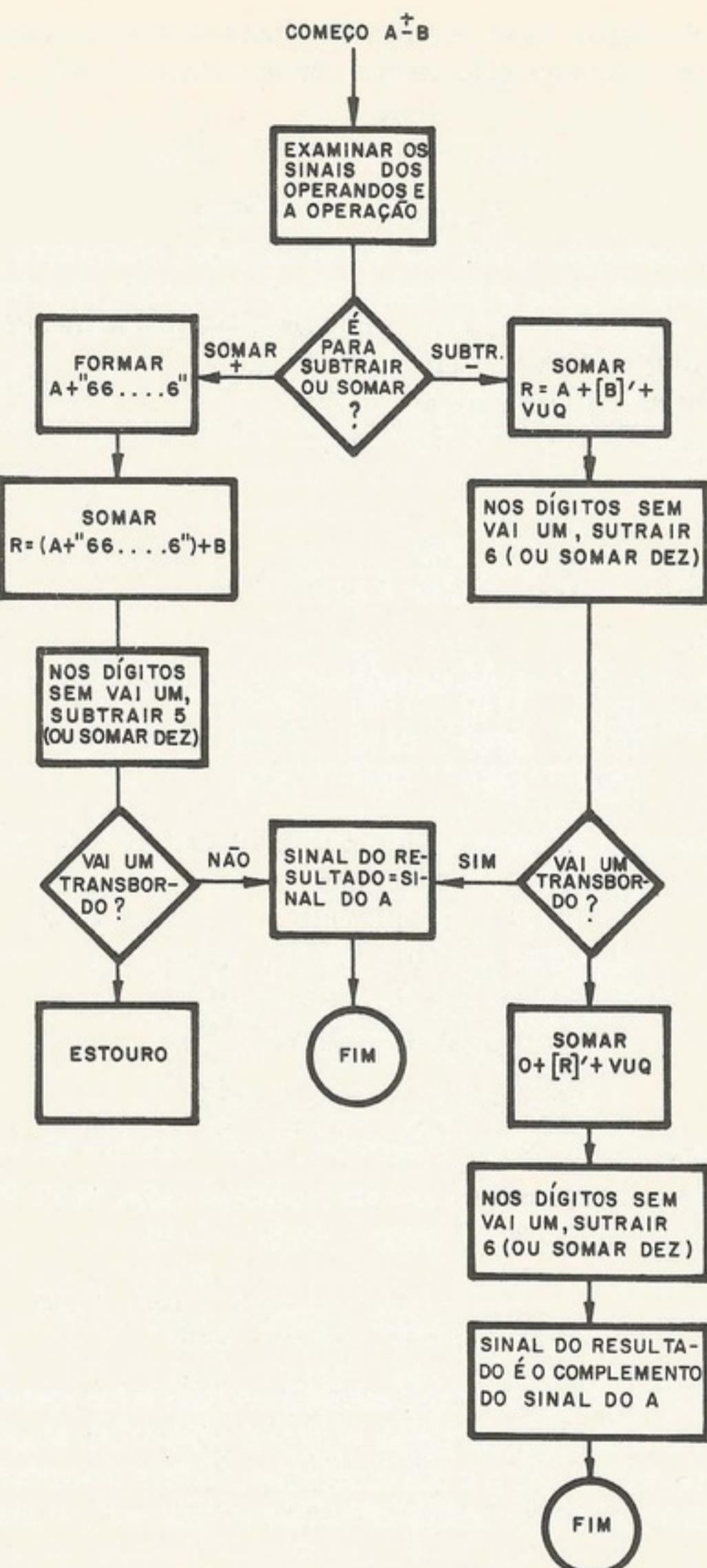


Figura 2-11. Aritmética BCD em sinal e amplitude

a. Conversão binária a decimal

Se o sistema tem meios de fazer divisão binária, a conversão pode ser efetuada através de divisões sucessivas por dez (1010_2). O resto da primeira divisão é o dígito decimal menos significativo. O quociente da primeira iteração é o dividendo da segunda, e o resto da segunda iteração é o dígito decimal segundo menos significativo. O processo continua reiterativamente até que o quociente fica zero. Esse último resto é o dígito decimal mais significativo do número convertido.

EXEMPLO. Converter $(11011011)_2$

$$\begin{array}{lll} 1) & 11011011 & 1010 = 10101 \\ 2) & 10101 & 1010 = 0010 \\ 3) & 0010 & 1010 = 0 \end{array} \quad \begin{array}{l} \text{quociente com } 1001 \text{ resto,} \\ \text{quociente com } 0001 \text{ resto,} \\ \text{quociente com } 0010 \text{ resto;} \end{array}$$

$$\begin{array}{lll} \text{a conversão:} & 0010 & 00001 & 1001 \\ & 2 & 1 & 9 \end{array}$$

Se não houver meios de divisão, existe o algoritmo que segue, chamado *double-dabble* (veja Couleur⁽⁹⁾). Dado um número binário, positivo e inteiro de n bits, a conversão a decimal pode ser efetuada da maneira que segue.

1. Começar com o bit mais significativo.
2. Se o seguinte bit é 0, dobrar o resultado intermediário.
3. Se o seguinte bit é “1”, dobrar o resultado intermediário, a acrescentar um.
4. Continuar o processo para todos os bits até o fim; o resultado é em decimal.

EXEMPLO

$$A = (11011011)_2 = (219)_{10}.$$

Começando com 1, dobramos e acrescentamos um para obter 3, porque o segundo bit é “1”. O terceiro bit é “0” e, então, simplesmente dobramos 3 para obter 6. O quarto bit é “1”; então temos 13 (dobrando, + 1). O quinto bit é “1” e agora temos 27 (dobrando, + 1). O sexto bit é “0”; então só dobrarmos para obter 54. O sétimo bit é “1”; então temos 109 (dobrando, + 1). O último bit é “1”; então, dobrando, mais 1, dá o resultado certo 219. O processo é justificado pela seguinte identidade:

$$A_0 2^{n-1} + A_1 2^{n-2} + \cdots + A_{n-1} 2^0 = (\cdots ((A_0)2 + A_1)2 + \cdots A_{n-2})2 + A_{n-1}.$$

Um somador decimal (ou um somador binário com meios de fazer correção para *BCD*) pode ser usado para implementar o algoritmo. Para dobrar o resultado intermediário, basta apenas somar o resultado a si mesmo, isto é, o resultado intermediário serve como ambas as parcelas. Durante esse processo, o “vem um quente” ficará “1” só se o seguinte bit do número binário for “1”; caso contrário, ficará “0”. O mesmo algoritmo também pode ser implementado através de um registrador especial de deslocamento que é dividido em dígitos de 4 bits cada. Cada seção de 4 bits tem meios de descobrir se o dígito guardado é uma combinação ilegal (veja a Tab. 2-1). Cada seção também tem meios de somar a parcela de seis $(0110)_2$. O registrador começa limpo. A cada iteração, um novo bit do número binário a ser convertido entra na posição menos significativa do registrador. Depois de cada deslocamento, se um dígito decimal de 4 bits apresentar uma combinação *BCD* ilegal ou se um bit na posição de peso “8” desse dígito passar para a posição de peso “1” do dígito decimal vizinho mais significativo, então a parcela de $(0110)_2$ será somada a esse dígito. Continuar até tratar o último bit (o bit menos significativo) do número binário.

No exemplo a seguir, cada iteração consta de dois passos: primeiro, deslocar; segundo, examinar e corrigir se for necessário. Rhyne⁽¹⁰⁾ estudou o problema em termos de circuitos seqüenciais (veja Sec. 1.2 e Fig. 1-11). Supomos que cada dígito conste de quatro *flip-flops*, $F8$, $F4$, $F2$ e $F1$ (conforme o peso da posição). Em termos do estado interno (o dígito atual, entre 0000 e 1001) e o “vem um” do vizinho, podemos predizer o próximo estado interno e o “vai um”. Essa informação é mostrada na *flow table* da Tab. 2-5. A tabela difere daquela da Fig. 1-24 no sentido que esse circuito seqüencial é sincronizado através de um sinal, ou melhor, pulso de controle não mostrado aqui explicitamente. O circuito faz uma só transição interna do estado para cada pulso de controle.

Tabela 2-5. Circuito seqüencial para realizar o algoritmo *double dabble* (Rhyme)

Estado interno				“Vem um”	
F8	F4	F2	F1	0	1
0	0	0	0	0000,0	0001,0
0	0	0	1	0010,0	0010,0
0	0	1	0	0100,0	0101,0
0	0	1	1	0110,0	0111,0
0	1	0	0	1000,0	1001,0
0	1	0	1	0000,1	0001,1
0	1	1	0	0010,1	0011,1
0	1	1	1	0100,1	0101,1
1	0	0	0	0110,1	0111,1
1	0	0	1	1000,1	1001,1

EXEMPLO. *Double dabble*

$A = (11011011)_2 = (219)_{10}$	8421	8421	8421	A
Começo	0000	0000	0000	11011011
Deslocar			1	1011011.
Deslocar			11	011011..
Deslocar			110	11011...
Deslocar			1101	1011....
Corrigir (+ 6)	1	0011		
Deslocar	10	0111	011.....	
Deslocar	100	1110	11.....	
Corrigir (+ 6)	101	0100		
Deslocar	1010	1001	1.....	
Corrigir (+ 6)	1	0000	1001	
Deslocar	10	0001	0011
Corrigir (+ 6)	10	0001	1001	← conversão
Resultado	2	1	9	

Observar a linha do estado interno “0100”, que significa $(4)_{10}$. Se o “vem um” é “0”, nessa iteração, simplesmente dobramos 4 para obter $(8)_{10}$, que é $(1000)_2$; e, se o “vem um” é “1”, “dobro + 1” é $(9)_{10}$, que é $(1001)_2$. Em cada caso, o “vai um” é “0”. No caso do estado interno “0110”, que significa $(6)_{10}$, o dobro é $(12)_{10}$, que já cria o “vai um” de “1” ao dígito vizinho, e o próximo estado interno é $(2)_{10}$ se o “vem um” é “0”, ou $(3)_{10}$ se o “vem um” é “1”.

b. Conversão decimal a binária

Rhyne⁽¹⁰⁾ também mostra que um processo iterativo baseado em deslocamento e correção pode ser realizado aplicando a técnica de *flow table*. Aqui, o processo começa com o bit menos significativo do número decimal e o sentido do deslocamento é para a direita.

Também existe um algoritmo para a conversão de *BCD* a binário, que só usa deslocamento e adição binária. O algoritmo aproveita o fato que para multiplicar um número binário por dez [multiplicador de $(1010)_2$], basta somar o multiplicando deslocado à esquerda três posições ao multiplicando deslocado à esquerda uma posição.

A conversão *BCD* para binário pode começar com o bit mais significativo. Para multiplicar o multiplicando por dez, basta somar com o multiplicador deslocado à esquerda três posições. Caso o resultado seja maior que 9, é necessário subtraí-lo de 10.

EXEMPLO. Transformação de *BCD* para binário

2.6 SUMÁRIO

Estudamos a representação de números inteiros em binário e ampliámos a aritmética que usamos. Os exemplos e fluxogramas não indicam estímulos de entrada binária, foram assumidos.

Um assunto que não foi efetuado através de exercícios é a conversão de decimal para binário, feita através de adição, subtração e multiplicação.

EXERCÍCIOS

- 2-1. É possível representar os números inteiros de -128 a 127 em 8 bits? (veja a referência)
- 2-2. (a) Qual é o resultado da operação $(-47)_{10} + (-12)_{10}$? (b) Qual é o resultado da operação $(-47)_{10} \times (-12)_{10}$?
- 2-3. Representar o resultado da operação $(-47)_{10} \times (-12)_{10}$ em binário.
- 2-4. O que é $(-47)_{10} \times (-12)_{10}$ em binário?
- 2-5. Como passar de decimal para binário?
- 2-6. Mostre como converter $(-47)_{10}$ para binário.
- 2-7. Repetir o exemplo da conversão decimal para binário.

EXEMPLO. Multiplicar $M = (000110)_2 = (6)_{10}$ por dez

$$\begin{aligned} M \times 2^3 &= 110000 \quad (M \text{ deslocado por } 3) \\ M \times 2^1 &= 001100 \quad (M \text{ deslocado por } 1) \\ \text{produto} &= \underline{111100} = (60)_{10}. \end{aligned}$$

A conversão *BCD* a binário é feita da seguinte maneira:

começar com o dígito *BCD* mais significativo;
multiplicar o resultado intermediário por dez $(1010)_2$;
somar com o próximo dígito *BCD*;
se, no passo anterior, o dígito *BCD* não foi o dígito menos significativo, voltar para segundo passo. Caso contrário, o resultado é a conversão.

EXEMPLO. Transcodificar $(219)_{10}$ em *BCD* (0010 0001 1001) para binário

primeiro resultado intermediário	0	0000	0010	$(2)_{10}$
multiplicar	0	0001	0000	
	0	0000	0100	
	0	0001	0100	
somar o próximo			0001	$(1)_{10}$
segundo resultado intermediário	0	0001	0101	
multiplicar	0	1010	1000	
	0	0010	1010	
	0	1101	0010	
somar o próximo			1001	$(9)_{10}$
resultado $(219)_{10}$	0	1101	1011	

2.6 SUMÁRIO

Estudamos alguns dos códigos mais comuns em sistemas digitais, bem como a representação de números. Em geral, existem três representações para os números negativos: sinal e amplitude, complemento da base e o complemento da base diminuída. Sistemas de aritmética que usam o processo de adição e complementação foram demonstrados com exemplos e fluxogramas. Dependendo da operação, o “vai um transbordo” normalmente não indica estouro de capacidade. Da mesma forma, conversões binária a decimal, e decimal a binária, foram estudadas com vários métodos ilustrados.

Um assunto não tratado foi a multiplicação e a divisão. A multiplicação é normalmente efetuada através de adições e deslocamentos. A divisão, que é muito mais complicada, consta de adição, subtração e deslocamentos. Esses assuntos são amplamente tratados em Flores⁽⁶⁾.

EXERCÍCIOS

- 2-1. É possível representar os inteiros com base negativa? Explique usando a base (-2) como exemplo (veja a referência⁽⁵⁾).
- 2-2. (a) Qual o valor decimal de $(01101101)_2$? (b) Qual a representação binária de 654?
- 2-3. Representar o número 12,1 em binário de dez bits, 5 bits inteiros e 5 bits fracionais.
- 2-4. O que é $(-47)_{10}$, usando representações binárias de 8 bits de:
 - (a) sinal e amplitude; (b) complemento de um; (c) complemento de dois.
- 2-5. Como parece o número 32000 em ponto flutuante “curto” normalizado do S/360? (veja a Fig. 2-2.)
- 2-6. Mostre como somar em complemento de um, $n = 6$ bits, as seguintes parcelas:
 - (a) $A = (-27)_{10}$, $B = (-7)_{10}$; (b) $A = (+27)_{10}$, $B = (+8)_{10}$; (c) $A = (+1)_{10}$, $B = (+5)_{10}$.
- 2-7. Repetir o exercício 2-6, com as parcelas codificadas em complemento de dois (6 bits).

2-8. Usando a técnica de subtrair através de complementação do subtraendo, mostrar como obter as seguintes diferenças $A - B$, em binário complemento de um, 6 bits:

- (a) $A = (+8)_{10}$, $B = (+7)_{10}$; (b) $A = (-16)_{10}$, $B = (+16)_{10}$; (c) $A = (+15)_{10}$, $B = (+24)_{10}$.

2-9. Repetir o Exercício 2-8 em complemento de dois de 6 bits.

2-10. Repetir o Exercício 2-8 em sinal e amplitude, (1 bit de sinal, 5 de amplitude).

2-11. Vamos supor que tenhamos um sistema decimal de 4 dígitos, mais um bit de sinal, que usa a representação de negativos em complemento de nove. Mostrar como fazer:

- (a) $A + B$, $A = +0136$, $B = -7654$;
 (b) $A + B$, $A = -9998$, $B = +7777$; [Observação. -9998 é $(-1)_{10}$.]
 (c) $A - B$, $A = -0010$, $B = +0108$. [Observação. -0010 é $(-9989)_{10}$.]

2-12. Transcodificar os números negativos (em complemento de nove) do Exercício 2-11 para a representação sinal e amplitude e repetir, mostrando os resultados em sinal e amplitude.

2-13. Repetir o Exercício 2-12, mas usando o código BCD com adição binária, complementação bit a bit e a técnica de correção de ± 6 .

2-14. Transformar a conversão binária a BCD dos seguintes números binários:

- (a) 001101; (b) 00100110111; (c) 01010101.

2-15. Efetuar a conversão BCD a binário dos seguintes números BCD:

- (a) 0010 0110 1001; (b) 0110 1000 0011; (c) 0100 0001 0111.

BIBLIOGRAFIA

- (1) M. Cohn e S. Even, "A Gray Code Counter", *IEEE Trans. Computers*, Vol. C-18, pp. 662-664 (julho de 1969)
- (2) R. W. Hamming, "Error Detecting and Correcting Codes", *Bell Systems Technical Journal*, Vol. 29, n.º 2, pp. 147-160 (abril de 1950)
- (3) W. W. Peterson, *Error-Correcting Codes*, The M.I.T. Press e John Wiley and Sons. Inc., New York, 1961
- (4) *IBM Journal of Research and Development*, Vol. 14, n.º 4 (julho de 1970). Exemplar especial em teoria e aplicações de codificação
- (5) Glen G. Langdon, Jr., "Subtraction by Minuend Complementation", *IEEE Trans. Computers*, Vol. C-18, n.º 1, pp. 74-75 (janeiro de 1969)
- (6) Ivan Flores, *The Logic of Computer Arithmetic*, Prentice-Hall, Englewood Cliffs, N.J., 1963
- (7) Y. Chu, *Digital Computer Design Fundamentals*, McGraw-Hill Book Co., New York, 1962
- (8) Peter Calingaert, *Princípios de computação*, Parte II. Ao Livro Técnico, Rio de Janeiro, 1969
- (9) J. F. Couleur, "BIDEC - a binary-to-decimal or decimal-to-binary converter", *IRE Trans. Elec. Computers*, Vol. EC-7, pp. 313-316 (dezembro de 1958)
- (10) V. T. Rhyne, "Serial Binary-to-Decimal and Decimal-to-Binary Conversion", *IEEE Trans. Computers*, Vol. C-19, n.º 9, pp. 808-812 (setembro de 1970)
- (11) H. Hellerman, *Digital System Design Principles*, McGraw-Hill, 1967

capítulo 3

ELEMENTOS

3.1 INTRODUÇÃO

O engenheiro eletrônico e também a tecnologia depende de seu know-how em componentes chamados circuitos integrados. Com o desenho preciso pensar em circuitos integrados são do tipo somente lógica, problema de ligação entre as ligações representadas por uma rede de componentes que o transistores.

Os circuitos integrados os circuitos de dados geralmente comunicam entre os tipos de circuitos.

Os circuitos integrados que regulam o trabalho da unidade de controle do fluxo de dados e estudada no Capítulo 3.

O projeto de circuitos que regula o trabalho da unidade de controle do fluxo de dados e estudada no Capítulo 3.

3.2 CIRCUITOS

Uma rápida visão para mostrar que os circuitos de seleção, decodificação e controle, etc.

capítulo 5

EXEMPLO DE UM MINICOMPUTADOR

5.1 INTRODUÇÃO

Para melhor explicar como se projetam sistemas digitais, neste capítulo usaremos como veículo didático um minicomputador cuja arquitetura foi definida, no primeiro semestre de 1971, pelos estudantes de pós-graduação do curso PEL-727, "Projeto de sistemas digitais", da Escola Politécnica da USP, projetado e implementado no Laboratório de Sistemas Digitais do Departamento de Engenharia de Eletricidade. Este capítulo também tem por finalidade dar uma introdução aos vários assuntos de arquitetura, tais como o formato e os campos da instrução, o formato dos dados e o endereçamento dos operandos pelas instruções. Será examinado em detalhes o projeto da arquitetura geral, o conjunto de instruções, o fluxo de dados, a unidade aritmética, o ciclo da máquina, as fases, a unidade de controle, o esquema de interrupção e os controles manuais.

5.2 ESBOÇO DA ARQUITETURA

Um vínculo do projeto foi a memória principal, o modelo FI-21 da Philips, já disponível. A memória tem ciclo de 1,6 µs, largura de 8 bits, e capacidade em quantidades de 1K (1 024) palavras. Assim, o tamanho da palavra ficou 8 bits ou seja, 1 byte. Para muitos fins, o tamanho de 1 byte é razoável. Como 8 bits já é um carácter de código *ASCII*, um processador de 8 bits serve para receber, guardar e retransmitir mensagens entre terminais do teclado impressora ou entre computadores. Chama-se essa aplicação de chaveamento de mensagens (*message switching*). O tamanho de 8 bits também serve para concentrador de dados (*data concentrator*). O concentrador regula o fluxo de mensagens em tempo real entre um computador e um conjunto de terminais, usando o computador em tempo partilhado (*time-sharing*). Uma outra aplicação que usa 8 bits é o teclado à fita (*key-to-tape*), em que informações para um computador são montadas diretamente de um teclado a uma fita magnética.

A palavra, então, ficou de 8 bits. Para fins aritméticos, foi definida a representação de negativos como a complementação de dois. O formato da instrução é do tipo *endereço simples*, isto é, a instrução só tem um campo de endereço. Quando a operação (definida pelo código da operação na instrução) precisa de dois operandos, como soma, por exemplo, o outro operando é um registrador central chamado *acumulador*. O acumulador é de 8 bits, com mais um *flip-flop* chamado *V* para guardar o último "vai um transbordo".

Com 8 bits, só podemos representar números de 0 a 255 ou, no caso de representação de negativos, de -128 a +127. Para facilitar o aumento da precisão, a palavra "1" da memória foi designada extensão do acumulador, e existe uma instrução para trocar o acumulador e a extensão. Assim, sub-rotinas podem ser programadas para executar aritmética em dupla precisão de 16 bits.

No início do projeto, tentou-se utilizar no máximo 1K (1 024) bytes da memória principal, que só necessita de 10 bits de endereço, mas ficou evidente que era viável prover um

campo de endereço de 11 bits, com 1 bit para o sinal de sinal de endereço.

A técnica de endereçamento é baseada no endereço de campo de endereço de 11 bits. Com 11 bits de endereço, é possível endereçar 2¹¹ = 2 048 bytes. O endereço de campo de endereço é dividido em dois campos: o campo de endereço de 10 bits, que indica o byte dentro da palavra, e o campo de endereço de 1 bit, que indica se o byte é o byte de endereço ou o byte de dados. O campo de endereço de 10 bits é dividido em dois campos: o campo de endereço de 9 bits, que indica o byte dentro da palavra, e o campo de endereço de 1 bit, que indica se o byte é o byte de endereço ou o byte de dados.

Na Fig. 5-1, o fluxo de dados é mostrado. Os dados são divididos em bytes, que são armazenados em registradores temporários. Um concentrador de dados é usado para conectar os registradores temporários a um byte de saída.

campo de endereço de 12 bits, já que fora resolvido usarem-se instruções longas de duas palavras. Com 16 bits disponíveis, quatro foram usados como código de instrução e doze de campo de endereço e, assim, a capacidade da memória principal passou para 4K bytes.

A técnica de indexação é, basicamente, a formação do endereço efetivo (o endereço real do operando) como a soma do campo de endereço e o conteúdo de um registrador indexador. Modificando o conteúdo do indexador, favorece a técnica de laços (*loops*) no programa. Um outro tipo de endereçamento chama-se *indireto*; o endereço efetivo do operando é o conteúdo da localização, indicado pelo campo de endereço da instrução. Em outras palavras, o endereço que vem junto com a instrução indica o *ponteiro* para o operando. Nessa arquitetura, o endereçamento indireto é complicado, devido ao fato de o conteúdo da palavra ser de 8 bits, enquanto o endereço precisa ser de 12 bits. Selecionou-se um sistema de endereçamento indexado. O esquema é restrito, no sentido de ser o índice de apenas 8 bits. Assim, a técnica do uso do indexador como ponteiro para qualquer palavra na memória não funciona porque, com 8 bits, só podemos indicar uma de 256 posições. Para economizar em registradores centrais, o registrador indexador ficou na posição "0" da memória. Assim, as instruções para carregar ou retirar palavras da memória também são usadas para carregar o indexador.

Na Fig. 5-1, o leitor encontra um diagrama simplificado da estrutura interna, ou seja, o fluxo de dados da UCP, ressaltando apenas os caminhos e paradas importantes. Dois registradores trabalham com a memória principal, o registrador de dados (RD) e o registrador de endereço (RE). O registrador da instrução (RI) guarda as instruções curtas de apenas um byte, ou o primeiro byte das instruções longas, isso para controlar as suas exe-

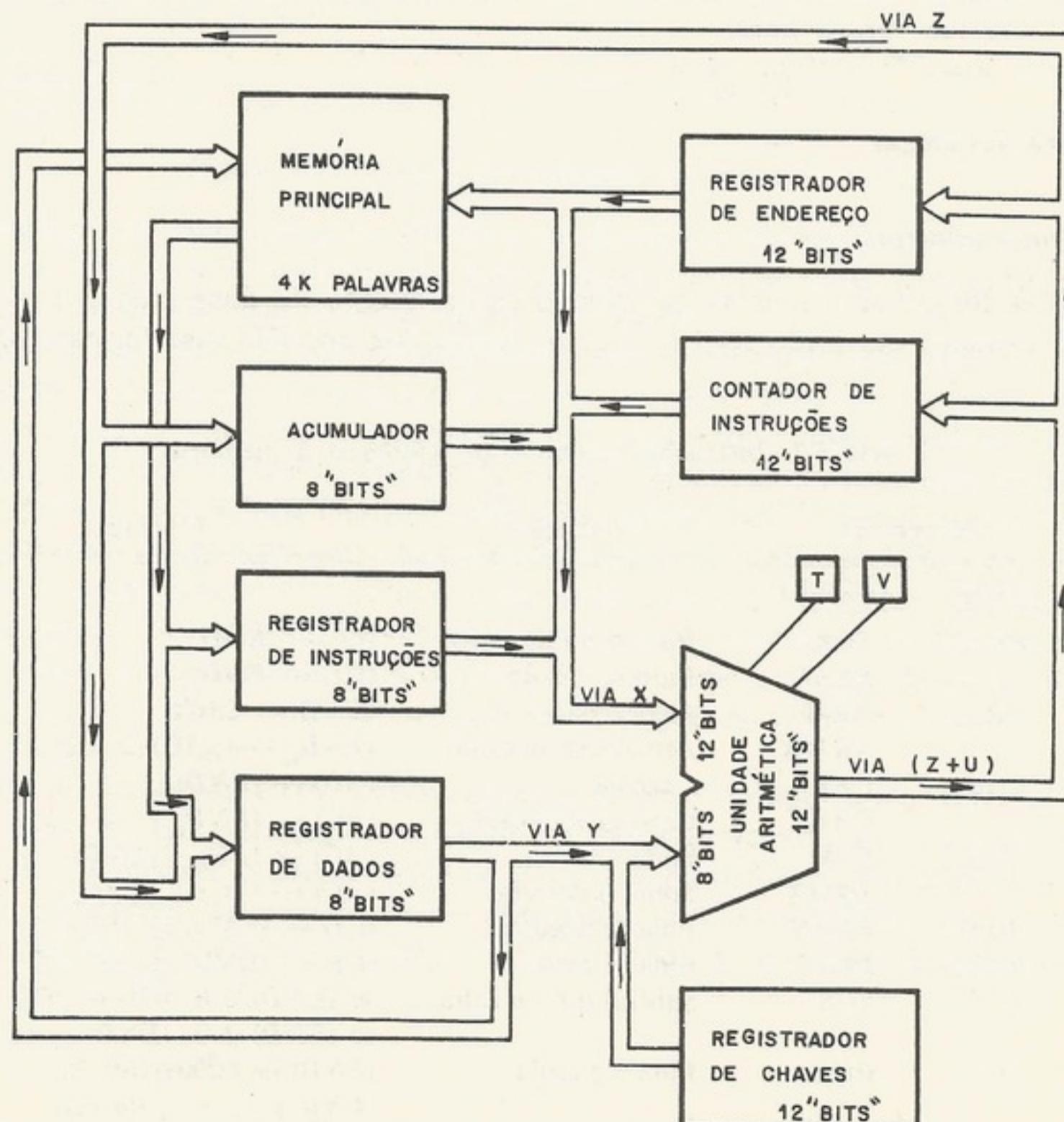


Figura 5-1. Fluxo de dados simplificado

ções, pois é no primeiro *byte* que está o código de operação. O conteúdo do contador de instrução (*CI*) é o ponteiro que mostra a localização, na memória, da próxima instrução. O acumulador (*AC*) é o registrador cujo conteúdo é controlado pelo programador através das instruções.

5.3 INSTRUÇÕES PRINCIPAIS

Utilizando uma memória com palavra de 8 bits, ficou definida a largura das vias e circuitos tratadores de dados (somador, portas de seleção, registradores etc.). Escolheu-se a representação dos números como o código complemento de dois, dadas as suas vantagens na realização das operações aritméticas. Fizeram-se as vias de endereçamento dos 12 bits necessários para o endereçamento direto das 4K palavras da memória.

As instruções com referência à memória são do tipo endereço simples, com o segundo operando operando no acumulador (registrador de 8 bits). Essas instruções são longas, ou seja, utilizam duas palavras (16 bits) já que se necessita de 12 bits para endereçar-se o primeiro operando. A maioria das instruções sem referência à memória é de largura de uma palavra (8 bits), pois não se precisa de campo de endereço.

Durante a fase de projeto lógico utilizou-se um conjunto de siglas mnemônicas para especificar as instruções. Terminando o projeto, tendo em vista a preparação dos programas ao *assembler*, mudaram-se as siglas mnemônicas para outros códigos melhores. Por isso, ver-se-á, na descrição a seguir, para cada instrução, duas siglas mnemônicas: a da primeira coluna (inicial), que aparecerá nos gráficos dos circuitos é a sigla usada na fase inicial do projeto; e a segunda coluna (definitivo) é a sigla definida para utilização pelo programador da máquina, na sua versão final.

a. Instruções longas

a.1. Grupo principal

Instruções longas são aquelas que têm o comprimento de duas palavras. A maioria delas tem o formato mostrado na Fig. 5-2. As instruções que têm esse formato são vistas na Tab. 5-1.

Tabela 5-1. Instruções longas com referência à memória

Código	Mnemônico		Descrição	Operação
	Inicial	Definitivo		
0000	PLI	PLA	Pulo incondicional	$(CI) \leftarrow END$
0001		PLAX	Pulo indexado	$(CI) \leftarrow END_{ef}$
0010	ARA	ARM	Armazena	$(END) \leftarrow (AC)$
0011		ARMX	Armazena indexado	$(END_{ef}) \leftarrow (AC)$
0100	CAC	CAR	Carrega	$(AC) \leftarrow (END)$
0101		CARX	Carrega indexado	$(AC) \leftarrow (END_{ef})$
0110	SOM	SOM	Soma	$(AC) \leftarrow (AC) + (END)$
0111		SOMX	Soma indexada	$(AC) \leftarrow (AC) + (END_{ef})$
1010	PAN	PLAN	Pula se negativo	$(CI) \leftarrow (END) \text{ se } AC < 0$
1011	PAZ	PLAZ	Pula se zero	$(CI) \leftarrow (END) \text{ se } AC = 0$
1110	SUS	SUS	Subtrai um ou salta	se $(END) = 0$: $(CI) \leftarrow (CI) + 2$ se $(END) \neq 0$: $(END) \leftarrow (END) - 1$
1111	PUG	PUG	Pula e guarda	$(END) \leftarrow 0000(CI(0-3))$ $(END + 1) \leftarrow (CI(4-11))$ $(CI) \leftarrow (END) + 2$

Figura 5-2. Formato de instruções longas

As instruções longas podem ser imediatas ou não. Com elas pode ser manipulada a "memória de operação", dependendo de qual "subrotina" (*SUS*) é invocada. O contador de endereço de memória é incrementado quando "salta" (*SUS*) e decrementado quando "vai para a memória" (*END*). O leitor já deve ter percebido que a maioria das instruções longas é de tipo "multiplicação".

A instrução de multiplicação é dividida em rotinas. Isso permite que o software utilize a mesma rotina para diferentes operações. O leitor já deve ter percebido que a maioria das instruções longas é de tipo "multiplicação".

Com a instrução de multiplicação, o conteúdo do CI é dividido entre a palavra da subrotina e a palavra da subrotina. O leitor já deve ter percebido que a maioria das instruções longas é de tipo "multiplicação".

a.2. Grupo de instruções de armazenamento

Ainda com o mesmo propósito, a instrução de multiplicação é dividida em rotinas.

Convém dizer que a maioria das instruções longas é de tipo "multiplicação".

exemplo de um minicomputador

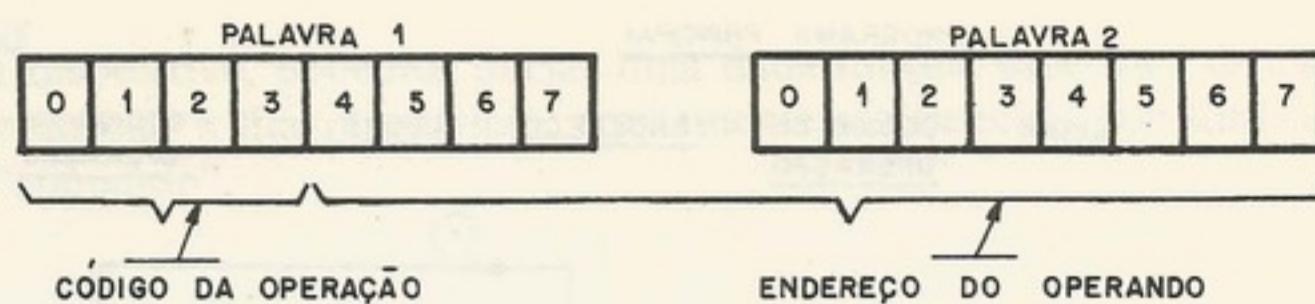


Figura 5-2. Formato das instruções longas

As instruções “pulo incondicional”, “armazena”, “carrega” e “soma”, são as únicas que podem ser indexadas. Nessas instruções, o bit 3 do código da operação indica se é indexado ou não. Com essas quatro, julga-se que tabelas de dados ou cadeias de caracteres podem ser manipuladas sem grande dificuldade. O “pulo incondicional” indexado tem aplicação na implementação de desvios de tipo “multi-rumo” (*many-way-branch*). Os “pulos condicionais”, dependendo do acumulador, são usados para testá-lo. A instrução “subtrai um ou salta” (*SUS*) é interessante, pois permite que qualquer palavra da memória possa servir como contador: o conteúdo do endereço efetivo, se não for zero, é decrementado e o computador executa a próxima instrução, que provavelmente é um pulo incondicional para o começo de um laço; quando o contador se anula, a próxima instrução é saltada, e o programa não volta ao começo do laço (veja a Fig. 5-3). Neste exemplo, “conta” (uma certa posição na memória) é carregada com “9”, então o laço será percorrido dez vezes até que termine (quando “conta” = 0) e saia do laço.

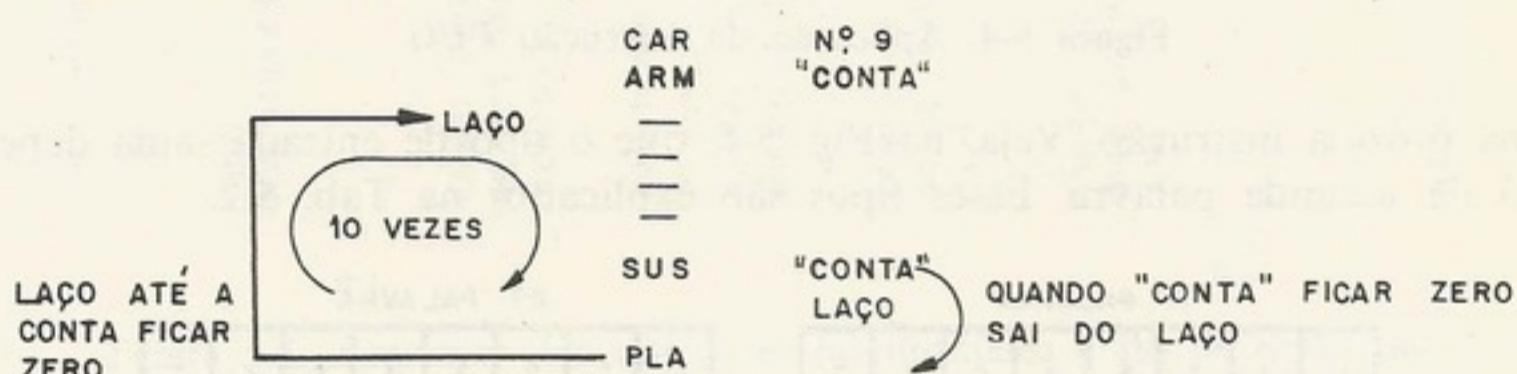


Figura 5-3. Exemplo da aplicação da instrução SUS

A instrução de “pula e guarda” (*PUG*) é um desvio que permite o uso de técnica de sub-rotinas. Isso porque o *PUG* deixa uma indicação do ponto de saída, para permitir regresso a ele. Vamos dizer algumas palavras sobre essa técnica.

O leitor já deve ter notado que a arquitetura desse minicomputador não tem a instrução “multiplicação”. Portanto, se quisermos multiplicar dois números, devemos fazê-lo por software isto é, fazer um trecho de programa que, utilizando somas e deslocamentos sucessivos, multiplique dois números. Admita agora que tenhamos um programa principal onde, em vários pontos, faça-se uma multiplicação. Uma solução para o problema seria inserirmos vários programas de multiplicação no programa principal. Outra solução seria termos uma só rotina de multiplicação (sub-rotina) e, em cada ponto do programa principal onde se queira multiplicar, desviamos o processamento para essa sub-rotina. No final da sub-rotina, voltamos com o processamento para o ponto do programa principal de onde saímos. Aqui surge um problema: como voltar ao ponto de saída, já que são vários os possíveis? Ai torna-se útil a instrução de *PUG*.

Com a instrução de *PUG*, endereçamos o começo da sub-rotina, onde é, automaticamente, montado um *PLA* para a volta ao programa principal (esse ponto de volta era o conteúdo do *CI* durante a fase de *fetch* de *PUG*) e o processamento é indicado na terceira palavra da sub-rotina. Terminada a sub-rotina existe um desvio (*PLA*) para o seu começo, onde será encontrado outro *PLA* para a volta ao programa principal. A aplicação do *PUG* está ilustrada na Fig. 5-4.

a.2. Grupo entrada/saída

Ainda como instruções longas, temos as instruções de entrada/saída num novo formato. Convém dizermos que esse sistema admite até dezesseis dispositivos de entrada/saída ende-

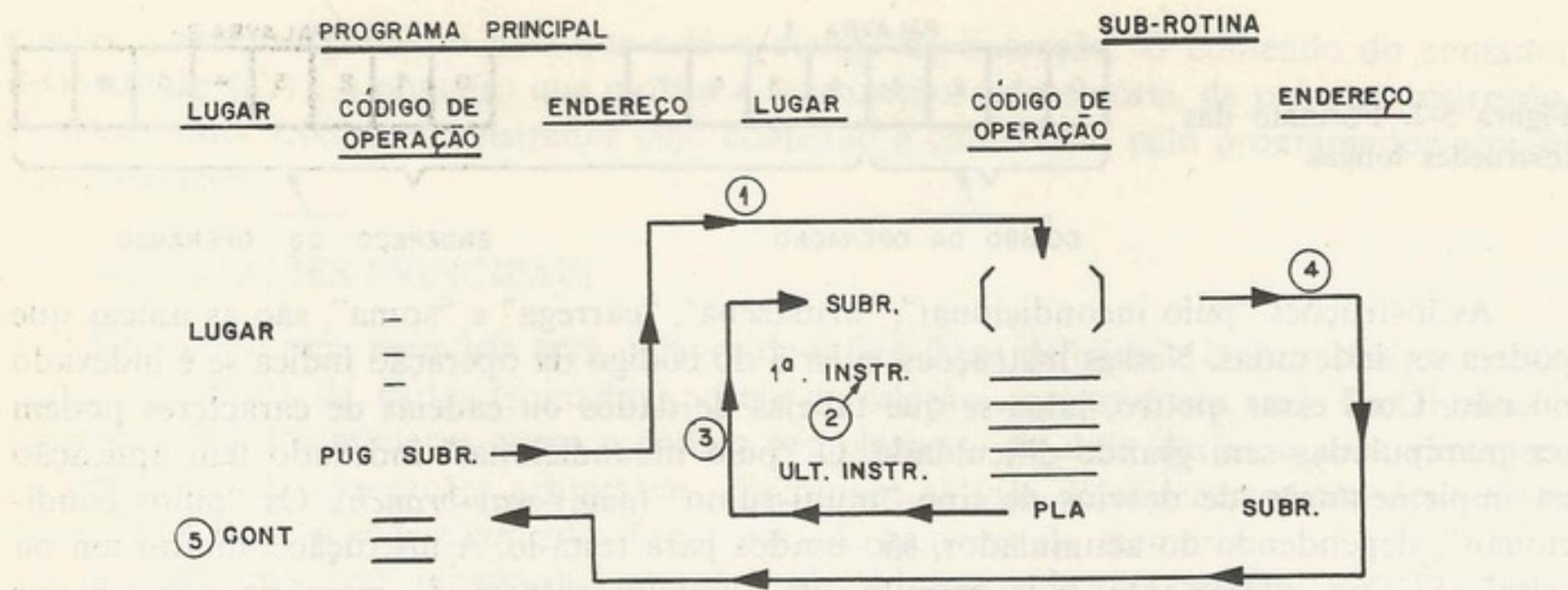
FNC.
um deles (p
e codificada

a.3. Grm

São aqu
de código d
grupo. Comit
lador". Apa
dificador. E
não é imedi
instruções d

0 1 2

1101 1101 1101



1. PUG: o CI que indica CONT está guardado no início da sub-rotina
2. O computador executa a primeira instrução da sub-rotina
3. Depois de executar a última instrução da sub-rotina, o computador executa um pulo para o lugar (SUBR) onde foi guardado o CI
4. No lugar SUBR o computador encontra um pulo incondicional para o lugar CONT
5. O computador executa a próxima instrução do programa principal

Figura 5-4. Aplicação da instrução PUG

reçáveis na própria instrução. Veja, na Fig. 5-5, que o tipo de entrada/saída depende dos bits 0 a 3 da segunda palavra. Esses tipos são explicados na Tab. 5-2.

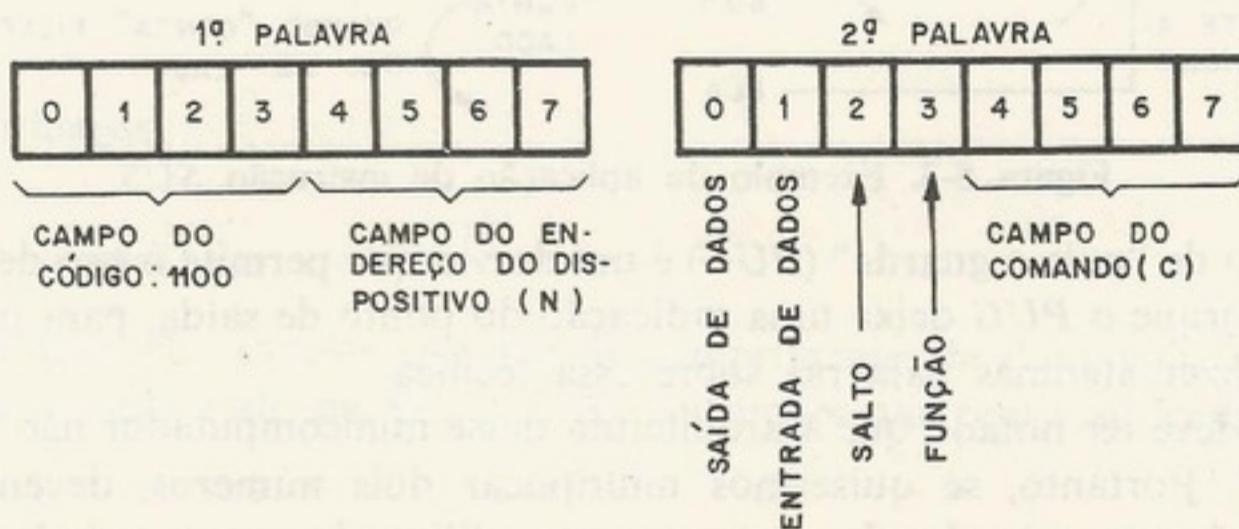


Figura 5-5. Formato das instruções longas do grupo entrada-saída

Tabela 5-2. Instruções do grupo entrada e saída

Mnemônico		Descrição
Inicial	Definitivo	
SAEDS	SAI, n, c	Saída de dados, comando c para dispositivo n
SAEDE	ENT, n, c	Entrada de dados, comando c do dispositivo n
SAESA	SAL, n, c	Salto, tipo c para o dispositivo n
SAEFU	FNC, n, c	Função tipo c para o dispositivo n

As técnicas de entrada/saída serão ressaltadas em detalhes no Cap. 7, mas acreditamos que aqui seja necessária uma breve descrição das duas últimas instruções.

SAL, n, c. Para cada dispositivo, dos dezesseis possíveis, podemos testar alguma condição (por exemplo, se o dispositivo está ocupado) previamente estabelecida por hardware e codificada no campo "comando" (portanto até dezesseis condições por dispositivo) e, no caso dessa condição testada ser satisfeita saltamos a instrução seguinte.

Código da
imediata

1000
0100
0010
1010

Código da
deslocamento

1000
0000
0100
0000
0000
0000
0000
0000

FNC, n. c. Para cada dispositivo, podemos iniciar uma dada função, específica de cada um deles (por exemplo, reenrolar a fita magnética), previamente estabelecida por hardware e codificada no campo “comando”.

a.3. Grupo das imediatas

São aquelas em cujo campo de endereço existe o próprio operando. Com um campo de código das imediatas de 4 bits, poder-se-iam implementar até dezesseis instruções nesse grupo. Contudo bastaram quatro delas: “soma”, *NAND*, *exclusive OR* e “carrega acumulador”. Aproveitou-se então para escolherem-se os códigos de modo a simplificar o decodificador. E, ainda, colocou-se, junto com essas instruções, a instrução de deslocamento que não é imediata. Veja, na Fig. 5-6(a), o formato das instruções imediatas e, na 5-6(b), o das instruções de deslocamento.

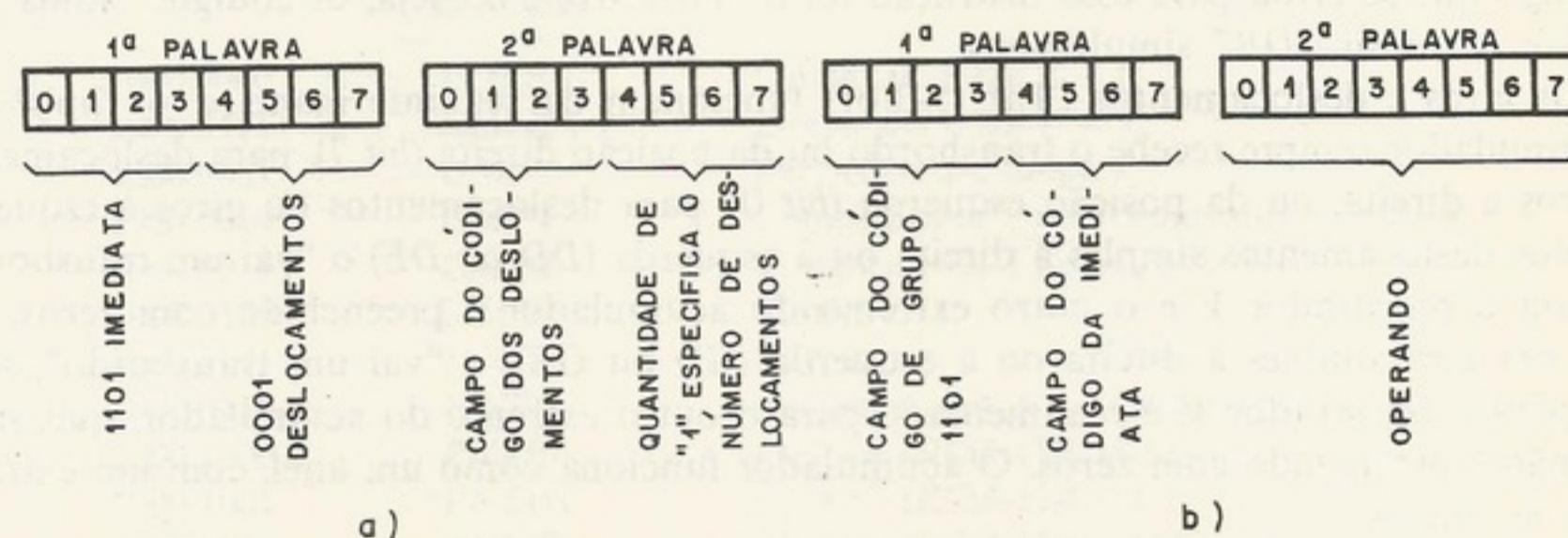


Figura 5-6. Formatos das instruções (a) imediatas e (b) de deslocamentos

Tabela 5-3.(a) Instruções imediatas

Código da imediata	Mnemônico	Inicial	Definitivo	Descrição	Operação
1000	IMEADD	IMEADD	SOMI	Soma imediata	$(AC) \leftarrow (AC) + \text{Operando}$
0100	IMENAND	IMENAND	NAND	<i>NAND</i> imediato	$(AC) \leftarrow ((AC) \wedge \text{Operando})'$
0010	IMEXOR	IMEXOR	XOR	<i>Exclusive-OR</i> imediato	$(AC) \leftarrow (AC) \oplus \text{Operando}$
1010	IMECAC	IMECAC	CARI	Carrega imediato	$(AC) \leftarrow \text{Operando}$

Tabela 5-3.(b) Instruções de deslocamento

Código do deslocamento	Mnemônico	Inicial	Definitivo	Descrição
1000	SR + DUPSI	SR + DUPSI	DDS	Desloca à direita com duplicação do sinal
0000	SR	SR	DD	Desloca à direita
0100	SL	SL	DE	Desloca à esquerda
0001	SR C/T	SR C/T	DDV	Desloca à direita com V
0101	SL C/T	SL C/T	DEV	Desloca à esquerda com V
0010	SR C/G	SR C/G	GD	Giro à direita
0110	SL C/G	SL C/G	GE	Giro à esquerda
0011	SR C/TG	SR C/TG	GDV	Giro à direita com V
0111	SL C/TG	SL C/TG	GEV	Giro à esquerda com V

Nessa arquitetura, existe uma certa economia no uso das instruções imediatas, pois o endereço de um *byte* precisa mais *bits* que o próprio *byte*. Então, para instruções em que um operando é uma constante (por exemplo, para somar de 1 ao acumulador), é mais econômico que o operando seja o próprio campo de endereço da instrução. Julgou-se também que, nas ocasiões em que se usam operações booleanas, fazemo-lo, freqüentemente, com constantes. A operação *NAND* é útil, pois, com ela, conseguimos executar as operações de *AND* e *OR*, com o auxílio de complementação. Suponha que queiramos fazer *AND* do acumulador com "11110000". Basta fazermos o *NAND* imediato com "11110000" e depois complementar o acumulador. Agora suponha que queiramos fazer *OR* do acumulador com "00001111". Podemos primeiro complementar o acumulador e depois fazer *NAND* com "11110000" (o complemento *bit a bit* dá "00001111").

Depois de definida a arquitetura, durante o projeto lógico desse computador, percebeu-se que era muito simples e barato criar-se a instrução de "carrega imediato" (*CARI*). O código que se criou para essa instrução foi o "11011010", ou seja, os códigos "soma imediata" e "exclusive *OR*" simultâneos.

Os giros e deslocamentos [Tab. 5-3(b)] funcionam da seguinte maneira: o *flip-flop V* do acumulador sempre recebe o transbordo ou da posição direita (*bit 7*), para deslocamentos ou giros à direita, ou da posição esquerda (*bit 0*), para deslocamentos ou giros à esquerda.

Nos deslocamentos simples à direita ou à esquerda (*DD* ou *DE*) o "vai um transbordo" vai para o registrador *V* e o outro extremo do acumulador é preenchido com zeros.

Com giro simples à direita ou à esquerda (*GD* ou *GE*), o "vai um transbordo", além de ir para o registrador *V*, é realimentado para o outro extremo do acumulador, que, nesse caso, não é preenchido com zeros. O acumulador funciona como um anel, com um extremo ligado ao outro.

As instruções *DDV* e *GDV* são iguais. O mesmo se diz dos *DEV* e *GEV*. Nessas instruções, o acumulador se fecha, também como um anel, tendo o registrador *V* intercalado entre os extremos dele. E o deslocamento se processa em giro, à esquerda ou à direita.

Na instrução particular *DDS* (desloca à direita com duplicação de sinal), o *bit "0"* (sinal do acumulador) não muda e ocorre um deslocamento à direita. Resulta então uma duplicação desse sinal para dentro do acumulador, conforme os deslocamentos vão ocorrendo.

b. Instruções curtas

As instruções curtas são instruções de apenas uma palavra. O computador distingue as instruções curtas das longas pelos três primeiros *bits*, que, nas curtas, devem ser 100. Qualquer instrução cujos três primeiros *bits* da primeira palavra não formem o número 100 é uma instrução longa. Nessa arquitetura, a instrução curta recebeu o nome de *microinstrução*. É importante ressaltar que esse nome nada tem a ver com microprogramas e foi dado apenas porque é uma instrução *curta*. Essas instruções curtas são divididas em dois microgrupos, conforme segue.

b.1. Microgrupo 1 (*MICG1*)

As microinstruções desse grupo controlam um ciclo em que o acumulador passa pela entrada *X* do somador e o resultado é devolvido ao acumulador. O formato da instrução é visto na Fig. 5-7, e a descrição é vista na Tab. 5-4.

Os *bits* 4 a 7 dessa microinstrução agem da seguinte maneira: o *bit 4* deixa passar o estado das chaves do painel para a entrada *Y* do somador. Se o acumulador não passa pela entrada *X* do somador (*bits* 5, 6 e 7 da microinstrução no estado "0"), a condição das chaves do painel é colocada no acumulador. Esse é um modo pelo qual o operador pode se comunicar com o programa. O *bit 5*, no estado "1", permite a passagem do acumulador para a entrada *X* do somador e o *bit 6*, no estado "1", permite a passagem do acumulador complementado pela entrada *X* do somador. Com os *bits* 5 e 6 dessa instrução simultaneamente

no estado "1",
("vem um que
Para incremen
5 = 1) com
basta passar
Seri um

b.2. Mic

Essas mi
e, se for venu
O forma

Figura 5-4. O
instruções, gru

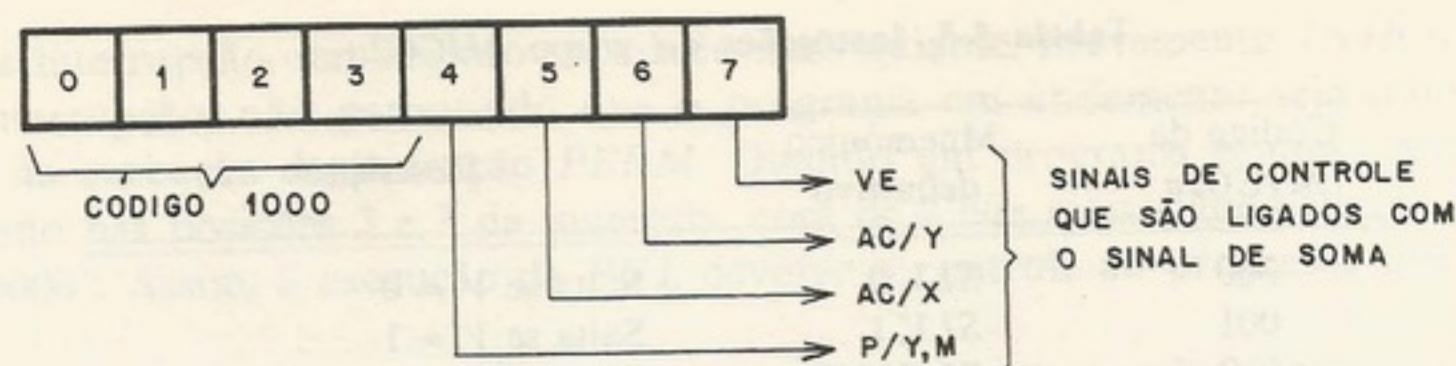
exemplo de um minicomputador

Figura 5-7. Formato da instrução curta, grupo 1

Tabela 5-4. Relação das microinstruções do grupo 1

Instrução	Mnemônico	Descrição
	Definitivo	
1000 0000	LIMP, 0	Limpa AC e faz $V = 0$
1000 0111	LIMP, 1	Limpa AC e faz $V = 1$
1000 0001	UM	Faz $AC = 1$ e limpa V
1000 0010	CMP1	Complementa 1 no acumulador e limpa V
1000 0011	CMP2	Complementa 2 no acumulador e limpa V
1000 0100	LIMV	Limpa V
1000 0101	INC	Incrementa o acumulador
1000 0110	UNEG	Faz acumulador $= -1$ e limpa T
1000 1000	PNL(0)	$(AC) \leftarrow (RC(4-11)), (V) \leftarrow 0$
1000 1001	PNL(1)	$(AC) \leftarrow (RC(4-11)) + 1$
1000 1010	PNL(2)	$(AC) \leftarrow (RC(4-11)) - (AC) - 1$
1000 1011	PNL(3)	$(AC) \leftarrow (RC(4-11)) - (AC)$
1000 1100	PNL(4)	$(AC) \leftarrow (RC(4-11)) + (AC)$
1000 1101	PNL(5)	$(AC) \leftarrow (RC(4-11)) + (AC) + 1$
1000 1110	PNL(6)	$(AC) \leftarrow (RC(4-11)) - 1$
1000 1111	PNL(7)	$(AC) \leftarrow (RC(4-11)), (V) \leftarrow 1$

no estado “1”, a entrada X do somador fica “1111 1111”. O bit 7 controla a entrada VUQ (“vem um quente”) do acumulador. Com o bit 7 no estado “1”, a quantidade “1” é somada. Para incrementar 1 ao acumulador, então basta passar o acumulador pela entrada X (bit 5 = 1) com VUQ (bit 7 = 1) do somador. Para fazer o complemento de 2 do acumulador, basta passar o complemento do acumulador com VUQ .

Será um bom exercício analisar todas essas combinações e testar o resultado.

b.2. Microgrupo 2a (MICG2a)

Essas microinstruções são usadas como salto, ou seja, é testada uma dada condição e, se for verdadeira, salta sobre a próxima instrução, supondo que ela seja longa.

O formato dessas instruções é visto na Fig. 5-8 e a descrição é mostrada na Tab. 5-5.

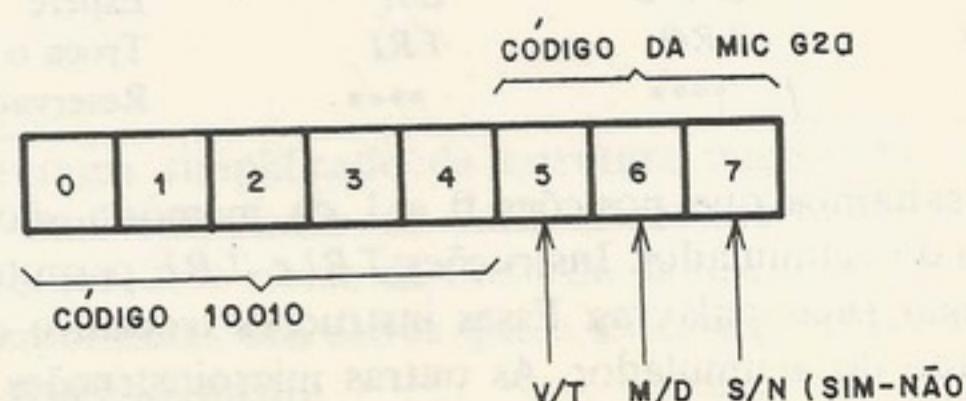


Figura 5-8. O formato das microinstruções, grupo 2

Tabela 5-5. Instruções do grupo *MICG2a*

Código da <i>MICG2a</i>	Mnemônico definitivo	Descrição
000	<i>SLV, 0</i>	Salta se $V = 0$
001	<i>SLV, 1</i>	Salta se $V = 1$
010	<i>SLVM, 0</i>	Se $V = 0$, salta e faz $V = 1$
011	<i>SLVM, 1</i>	Se $V = 1$, salta e faz $V = 0$
100	<i>SLT, 0</i>	Salta se $T = 0$
101	<i>SLT, 1</i>	Salta se $T = 1$
110	<i>SLTM, 0</i>	Se $T = 0$, salta e faz $T = 1$
111	<i>SLTM, 1</i>	Se $T = 1$, salta e faz $T = 0$

As microinstruções do grupo 2a são usadas para testar o estado dos *flip-flops V* ("vai-um" do somador e o último bit de derrame dos deslocamentos ou giros) e *T* (transbordo-estouro das instruções aritméticas). O bit 5 seleciona qual *flip-flop* será testado pela microinstrução, ou *V* ou *T*. O bit 6, quando no sentido "muda", provê a capacidade de disparar ou limpar os *flip-flops*. Esse bit, no sentido "muda", só realizará a mudança (de "1" para "0" ou "0" para "1") quando o computador saltar; caso não salte, ele deixará o *flip-flop* como está. Assim, o mnemônico *SLTM 1* sempre deixa o *flip-flop T* limpo; se *T* for "0", o computador não saltará e deixará *T* limpo; caso contrário, se *T* for "1" o computador saltará e mudará *T* para "0". Os saltos são executados (incluído o ciclo *I*) em um ciclo da máquina apenas. Essas instruções, seguidas de um pulo incondicional, equivalem a *pulos condicionais*, testando os *flip-flops V* e *T*.

b.3. Microgrupo 2b (*MICG2b*)

O grupo 2b consta dos comandos especiais. O formato é visto na Fig. 5-9 e a descrição dos subcódigos é mostrada na Tab. 5-6.

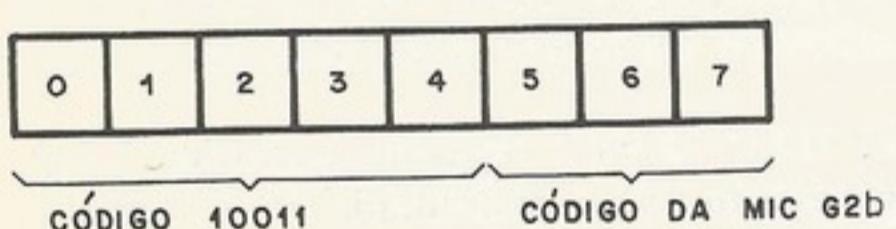


Figura 5-9. Formato das microinstruções grupo 2b

Tabela 5-6. Instruções do grupo *MICG2b*

Código da <i>MICG2b</i>	Inicial	Definitivo	Descrição
000	<i>PUL</i>	<i>PUL</i>	Pula para a posição 2 e limpa interrupção
001	<i>TRE</i>	<i>TRE</i>	Troca o acumulador com extensão
010	<i>INI</i>	<i>INIB</i>	Inibe interrupção
011	<i>PER</i>	<i>PERM</i>	Permite interrupção
100	<i>PAR-D</i>	<i>PARE</i>	Pare
101	<i>ESP-D</i>	<i>ESP</i>	Espere
110	<i>TRO</i>	<i>TRI</i>	Troca o acumulador com registrador de índice
111	****	****	Reservado para uso futuro

Ressaltamos que posições 0 e 1 da memória são, respectivamente, o indexador e a extensão do acumulador. Instruções *TRI* e *TRE* permitem os meios econômicos para retirar ou carregar essas palavras. Essas instruções trocam o conteúdo da respectiva palavra com o conteúdo do acumulador. As outras microinstruções de grupo 2b são vinculadas com o

exemplo de um m...

esquema de interrupção. O tema de interrupção é abordado mais tarde, mas é importante mencionar que a interrupção é realizada quando o bit 6 da microinstrução é alterado de 0 para 1. Isso ocorre quando a interrupção é acionada ou quando a microinstrução é executada. A interrupção é cancelada quando o bit 6 é alterado de 1 para 0.

A instrução *SUS* (STOP) é usada para parar o computador. Ela é executada quando o botão de parada é pressionado. A instrução *ARM* (ARMAMENTO) é usada para armazenar a configuração de interrupção. A instrução *ARMX* (ARMAMENTO DE EXCEÇÃO) é usada para armazenar a configuração de exceção. A instrução *CAR* (CARREGAMENTO) é usada para carregar o conteúdo de uma palavra de memória para o acumulador. A instrução *CARX* (CARREGAMENTO DE EXCEÇÃO) é usada para carregar o conteúdo de uma palavra de memória para o registrador de índice. A instrução *CARI* (CARREGAMENTO DE INÍCIO) é usada para carregar o conteúdo de uma palavra de memória para o indexador. A instrução *PUG* (PULGADA) é usada para pular para a posição 2. A instrução *SUS* (STOP) é usada para parar o computador. A instrução *NAND* (NAND) é usada para realizar a operação lógica AND complementada. A instrução *XOR* (XOR) é usada para realizar a operação lógica XOR. A instrução *PLA* (PLA) é usada para realizar a operação lógica PLA. A instrução *PLAX* (PLA X) é usada para realizar a operação lógica PLA X. A instrução *PLAN* (PLAN) é usada para realizar a operação lógica PLAN. A instrução *PLAZ* (PLAZ) é usada para realizar a operação lógica PLAZ. A instrução *TRE* (TRE) é usada para trocar o acumulador com a extensão. A instrução *TRI* (TRI) é usada para trocar o acumulador com o registrador de índice. A instrução *PUL* (PUL) é usada para pular para a posição 2. A instrução *PARE* (PARE) é usada para parar o computador. A instrução *ESP* (ESP) é usada para esperar. A instrução *INIB* (INIB) é usada para inibir a interrupção. A instrução *PERM* (PERM) é usada para permitir a interrupção. A instrução *DDS* (DDS) é usada para desabilitar o deslocamento. A instrução *DD* (DD) é usada para desabilitar o deslocamento. A instrução *DE* (DE) é usada para desabilitar o deslocamento. A instrução *DET* (DET) é usada para desabilitar o deslocamento.

5.4 FLUXO DE DADOS

Na Fig. 5-10, vemos o fluxo de dados entre o processador e a memória. O processador tem uma interface central de memória que pode ser dividida em dois caminhos: um para o processador e outro para a memória. O processador também tem uma interface de comunicação com o sistema de gerenciamento de memória (SMM). O SMM é responsável por gerenciar a memória e fornecer informações ao processador sobre a localização de dados. O processador também tem uma interface de comunicação com o sistema de gerenciamento de interrupções (SGI). O SGI é responsável por gerenciar as interrupções e fornecer informações ao processador sobre a origem de uma interrupção. O processador também tem uma interface de comunicação com o sistema de gerenciamento de energia (SGE). O SGE é responsável por gerenciar a energia e fornecer informações ao processador sobre a disponibilidade de energia. O processador também tem uma interface de comunicação com o sistema de gerenciamento de segurança (SGS). O SGS é responsável por gerenciar a segurança e fornecer informações ao processador sobre a integridade dos dados.

esquema de interrupção, um assunto abordado mais adiante. Brevemente, *INIB* inibe o sistema de interrupção, não permitindo que o programa em andamento seja interrompido, até depois da execução da instrução *PERM*. Quando um programa é interrompido, o *CI* fica guardado nas posições 2 e 3 da memória, com os 4 bits mais significativos da palavra 2 sendo "0000". Assim, a execução de *PUL* devolve o controle ao programa que foi interrompido.

A instrução *ESP* coloca o computador num estado de "espera" até que seja acionado o botão de partida do painel ou que uma interrupção aconteça. A instrução *PARE* é igual à *ESP*, com exceção de que *PARE* não aceita interrupção, o computador só continua com intervenção pelo painel.

O leitor encontra na Tab. 5-7 uma relação das principais instruções desse minicomputador.

Tabela 5-7. Instruções principais

Mnemônico	Descrição	Mnemônico	Descrição
<i>SOM</i>	Soma	<i>GD</i>	Giro à direita
<i>SOMX</i>	Soma indexado	<i>GE</i>	Giro à esquerda
<i>SOMI</i>	Soma imediato	<i>GDT</i>	Giro à direita com <i>T</i>
<i>ARM</i>	Armazena	<i>GET</i>	Giro à esquerda com <i>T</i>
<i>ARMX</i>	Armazena indexado	<i>SLT</i>	Salta se <i>T</i> igual operando
<i>CAR</i>	Carrega	<i>SLO</i>	Salta se <i>OVF</i> igual operando
<i>CARX</i>	Carrega indexado	<i>SLTM</i>	Salta se <i>T</i> = operando e muda <i>T</i>
<i>CARI</i>	Carrega imediato	<i>SLOM</i>	Salta se <i>OVF</i> = operando/muda <i>OVF</i>
<i>PUG</i>	Pula e guarda	<i>LIMP</i>	Limpa <i>AC</i> e faz <i>T</i> = operando
<i>SUS</i>	Subtrai um ou salta	<i>CMP1</i>	Comp. de 1 o <i>AC</i> e limpa <i>T</i>
<i>NAND</i>	<i>NAND</i>	<i>CMP2</i>	Complementa de 2 o <i>AC</i>
<i>XOR</i>	<i>Exclusive-OR</i>	<i>INC</i>	Incrementa <i>AC</i>
<i>PLA</i>	Pulo incondicional	<i>UM</i>	Faz <i>AC</i> = 1 e limpa <i>T</i>
<i>PLAX</i>	Pulo indexado	<i>UNEG</i>	Faz <i>AC</i> = -1 e limpa <i>T</i>
<i>PLAN</i>	Pula se negativo	<i>LIMT</i>	Limpa <i>T</i>
<i>PLAZ</i>	Pula se zero	<i>PNL(0)</i>	$(AC) \leftarrow (RC(4-11)), (T) \leftarrow 0$
<i>TRE</i>	Troca com extensão	<i>PNL(1)</i>	$(AC) \leftarrow (RC(4-11)) + 1$
<i>TRI</i>	Troca com índice	<i>PNL(2)</i>	$(AC) \leftarrow (RC(4-11)) - (AC) - 1$
<i>PUL</i>	Pula e limpa	<i>PNL(3)</i>	$(AC) \leftarrow (RC(4-11)) - (AC)$
<i>PARE</i>	Pare	<i>PNL(4)</i>	$(AC) \leftarrow (RC(4-11)) + (AC)$
<i>ESP</i>	Espere	<i>PNL(5)</i>	$(AC) \leftarrow (RC(4-11)) + (AC) + 1$
<i>INIB</i>	Inibe interrupção	<i>PNL(6)</i>	$(AC) \leftarrow (RC(4-11)) - 1$
<i>PERM</i>	Permite interrupção	<i>PNL(7)</i>	$(AC) \leftarrow (RC(4-11)), (T) \leftarrow 1$
<i>DDS</i>	Desloca à direita c/dupl. sinal	<i>SAI, n, c</i>	Saída de dados p/dispositivo <i>n</i>
<i>DD</i>	Desloca à direita	<i>ENT, n, c</i>	Entr. de dados p/dispositivo <i>n</i>
<i>DE</i>	Desloca à esquerda	<i>SAL, n, c</i>	Salto tipo I p/dispositivo <i>n</i>
<i>DDT</i>	Desloca à direita com <i>T</i>	<i>FNC, n, c</i>	Função tipo I p/dispositivo <i>n</i>
<i>DET</i>	Desloca à esquerda com <i>T</i>		

5.4 FLUXO DE DADOS

Na Fig. 5-1, encontra-se um diagrama simplificado da estrutura interna da unidade central de processamento (*UCP*) do minicomputador do LSD (Laboratório de Sistemas Digitais). Esse diagrama, normalmente, recebe o nome de fluxo de dados, pois ele ressalta apenas os caminhos e paradas mais importantes dos dados que o sistema processa. Na Fig. 5-10, vemos o fluxo de dados mais detalhadamente.

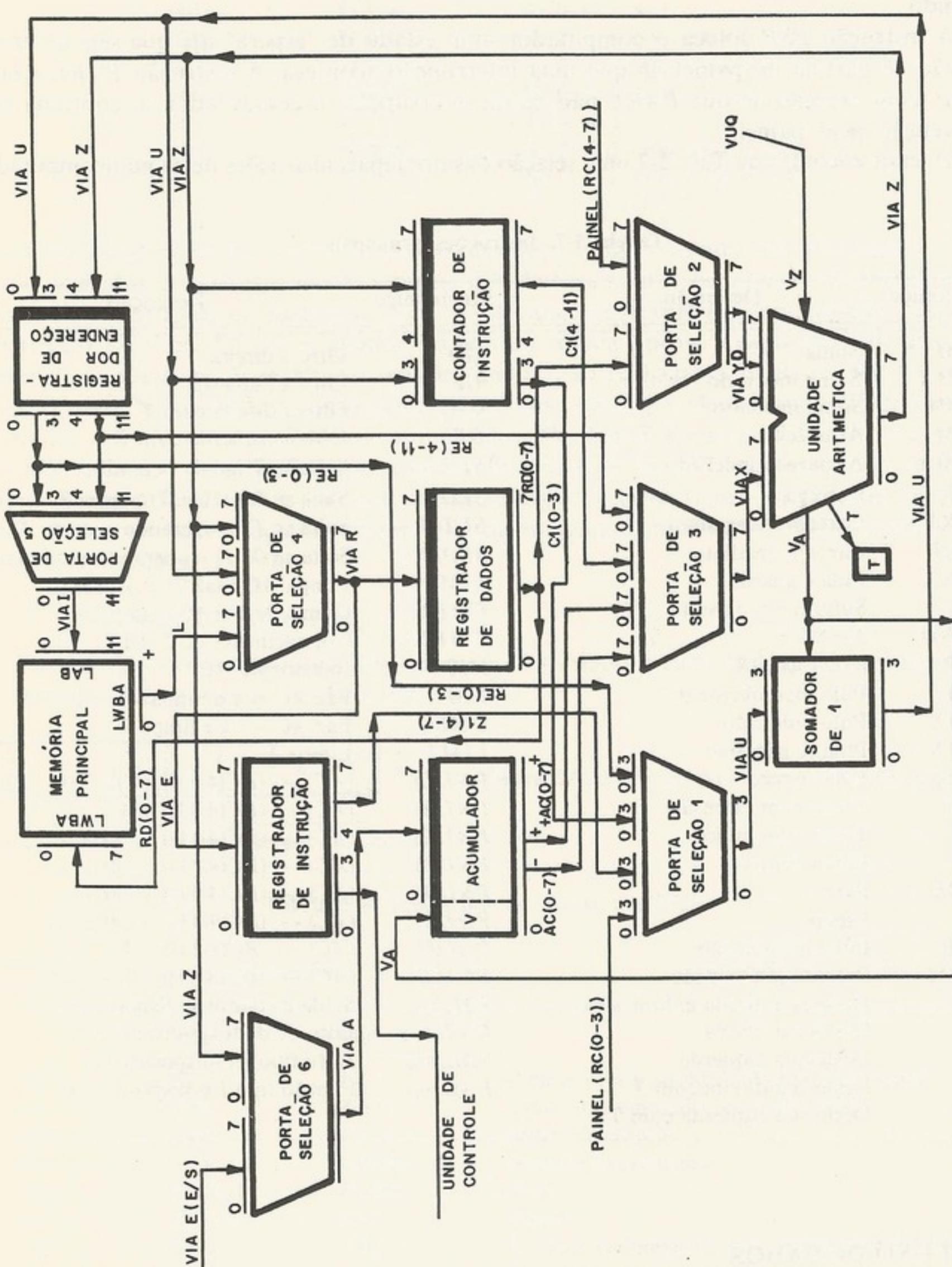


Figura 5-10. Fluxo de dados mais detalhado

a. Análise de circuitos

a.1. Memória

A memória principal, organizada em palavras de 8 bits num total de 4096 palavras, é uma memória de núcleos de ferrite, numa montagem de três fios por núcleo (FI 21, Philips). Dos sete modos de funcionamento, ressaltar-se-ão os três mais usados.

Ciclo completo

“Ler e restaurar” (*M*4): duração total, 1,6 μ s; tempo de acesso, 0,4 μ s.

Ciclo rachado

“Ler” (*M*1): duração total, 0,6 μ s; tempo de acesso, 0,4 μ s.

“Escrever” (*M*3): duração total, 1,0 μ s.

Sob o ponto de vista da unidade de controle, é preciso lembrar que, durante todo o ciclo, o endereço deve ficar constante. Existem inúmeros sinais elétricos necessários para controlar a memória e, por isso, projetou-se um circuito de interface de tal forma que, quando a unidade de controle precisar ler ou escrever algo na memória, ela envia apenas os sinais *M*4 ou *M*1 ou *M*3 que esse circuito gera os controles necessários.

As posições 0 e 1 da memória são reservadas como índice e extensão do acumulador, respectivamente.

Na Fig. 5-10, distinguem-se os seguintes blocos funcionais: memória, registradores, unidade aritmética e portas de seleção.

a.2. Registradores

Os cinco registradores do fluxo de dados podem ser agrupados em duas classes: os registradores de dados (*RD*, *AC*, *RI*), de 8 bits, com a finalidade de armazenar uma palavra de memória; e os registradores de endereço (*RE* e *CI*), com o objetivo de guardar endereços. Todos os registradores foram implementados com *flip-flops* tipo *D* sensíveis à borda. Segue-se uma breve descrição desses registradores.

Registrador de endereço (*RE*)

É o registrador que tem a função de segurar o endereço de memória e mantê-lo fixo durante todo o ciclo. Sua entrada são vias *Z* e *U*, saídas da unidade aritmética, e sua saída alimenta a memória através da porta de seleção 5. Esse registrador tem uma linha de controle (*set-reset*) para forçar o endereço 2, utilizado quando chega um pedido de interrupção. A Fig. 5-11 mostra alguns detalhes desse circuito.

Contador de instruções (*CI*)

É outro registrador de 12 bits com a entrada ligada à saída da unidade aritmética. Sua função é armazenar o endereço da instrução seguinte (note-se que, nessa arquitetura, o registrador faz parte do fluxo de dados e não da unidade de controle). Para se incrementar um ao endereço, utiliza-se a unidade aritmética, fazendo-se a soma com o *VUQ* (“vai um quente” ou “vem um”). A saída do *CI* vai direto à entrada da unidade aritmética.

Acumulador (*AC*)

É um registrador deslocador de 8 bits, com a opção de colocar o *flip-flop V* (“vai um”) em série com os 8 bits, como se fosse um nono bit do acumulador (Fig. 5-12). Esse registrador é, talvez, o núcleo da arquitetura do computador; todas as operações aritméticas e lógicas são feitas tendo o conteúdo do acumulador como um dos operandos, sendo o resultado armazenado nele: todos os desvios condicionais são feitos a partir de testes em seu conteúdo; a entrada e a saída de dados são feitas através dele.

Figura 5-10. Fluxo de dados mais detalhado



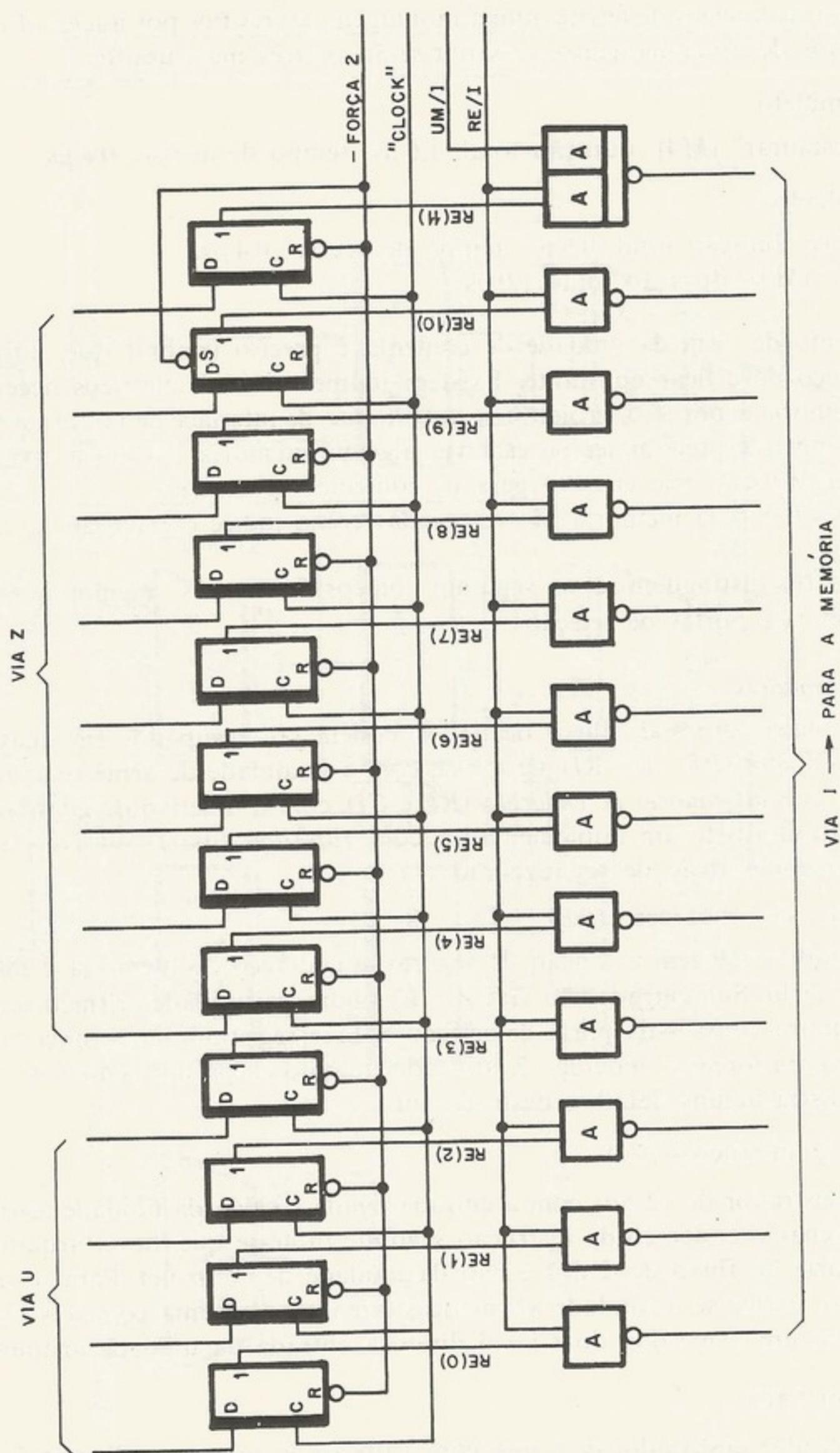


Figura 5-11. Registrador de endereço e porta de seleção 5

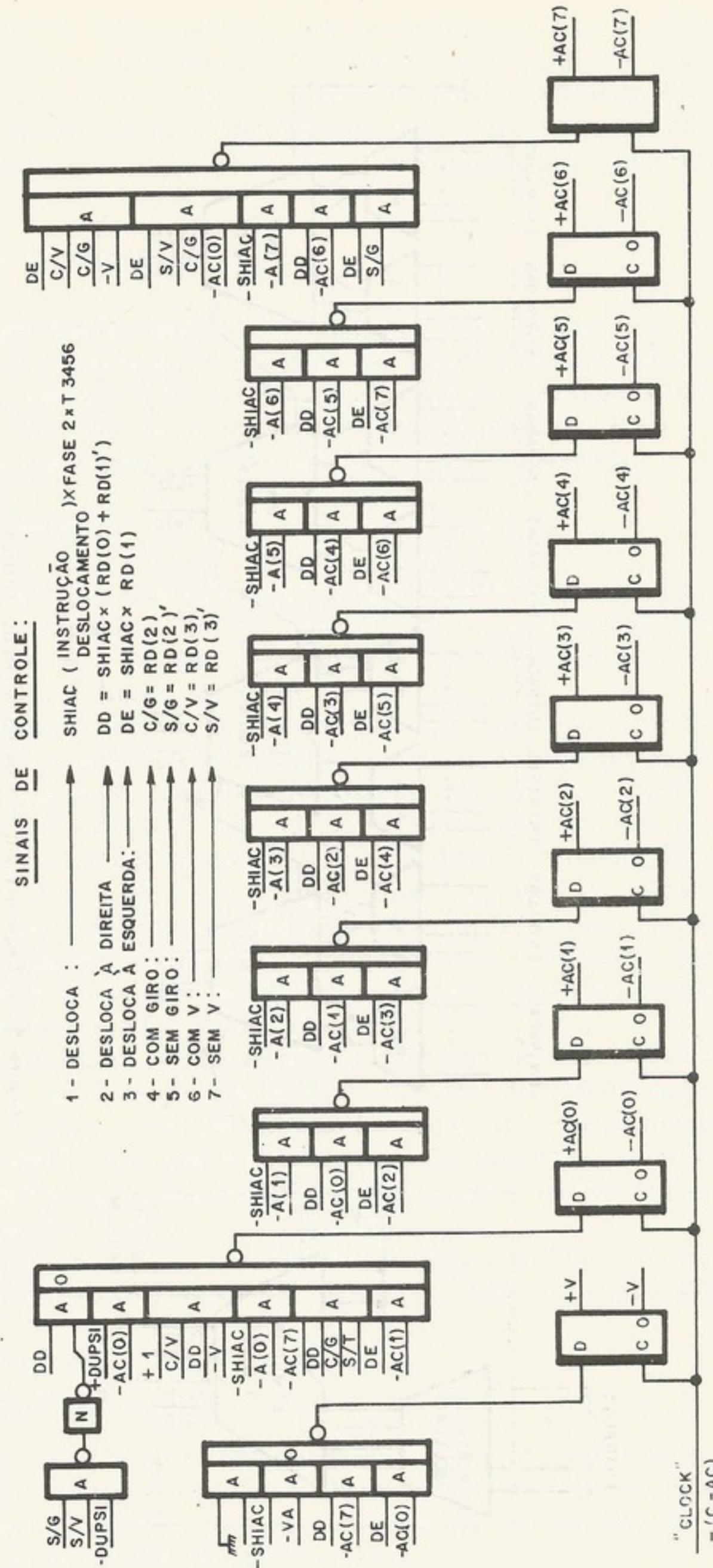


Figura 5-12. Circuito do acumulador

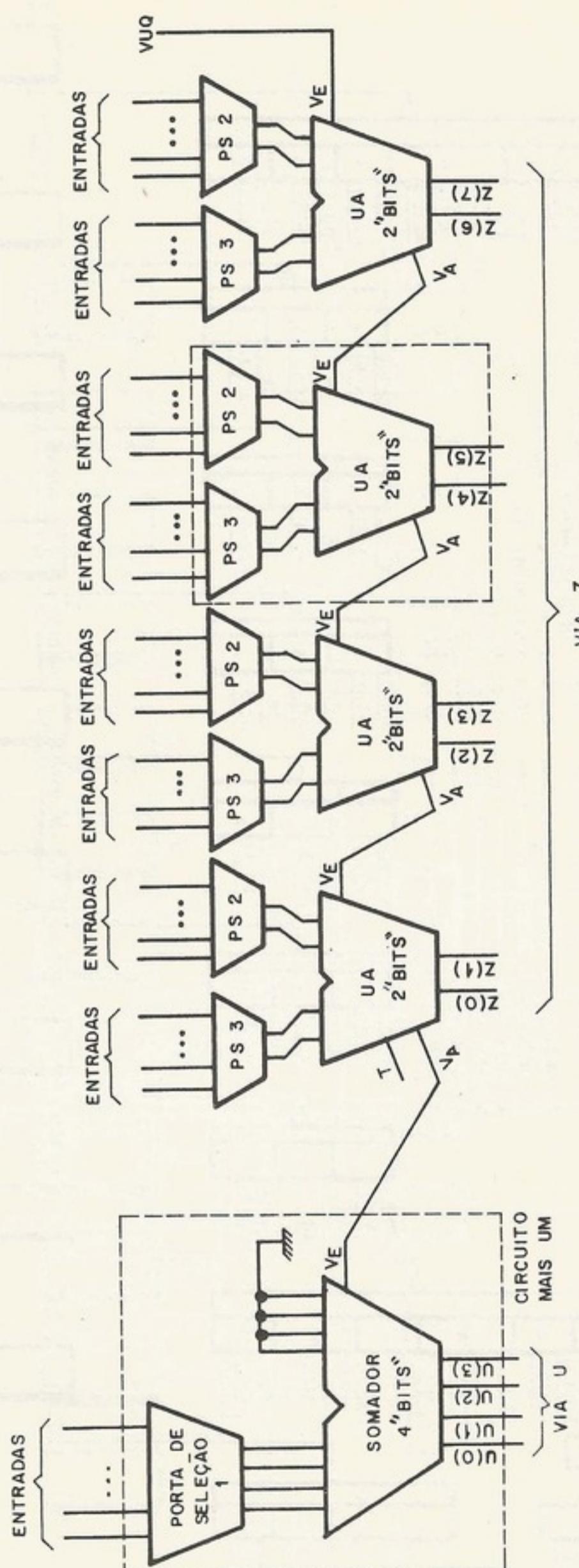


Figura 5-13. Esquema em blocos da unidade aritmética

Registrando

É um registrador que armazena os resultados das operações aritméticas realizadas na memória principal (RAM) durante a execução de instruções aritméticas e lógicas.

Registrando

Como a unidade aritmética opera sobre 4 bits, o RI é usado para indicar o resultado da operação (endereço de memória de 8 bits provendo o endereço necessário durante a leitura).

a.3. Unidade aritmética

A função da unidade aritmética é especificada por 8 funções. Essas funções são:

A unidade aritmética soma o resultado de 4 bits em paralelo. No caso de se tratar dos 4 bits mais significativos, é “desequilibrado”, ou seja, foi projetado para somar os 4 bits mais significativos da soma de um “dígito” mais significativo.

Para se obter esse resultado, o somador (Fig. 5-13) divide a parcela em 4 bits em 2 parcelas e, com isso, forçamos um bit de trocado. As operações de *NAND* e *exclusive OR*

Na Fig. 5-13, os 4 blocos em destaque

a.4. Porta de seleção

Na Fig. 5-13, a porta de seleção sempre que entra em operação, é sempre de um circuito que tem como saída a saída de menor peso.

Esse circuito é chamado de porta de seleção (*S*) é nula. Quando *E1* é zero, a saída é zero. Com *E2* é zero, a saída é zero.

Registrador de dados da memória (RD)

É um registrador de 8 bits que serve com um *buffer* entre a *UCP* e a memória. Dados lidos na memória são armazenados nele. Também os dados para gravar na memória ficam nele durante o ciclo de escrita. Por esse motivo, o segundo operando das instruções aritméticas e lógicas é armazenado nele (veja, sua saída está ligada à entrada *Y* da unidade aritmética), já que o primeiro, conforme já foi dito, é o conteúdo do acumulador.

Registrador de instruções (RI)

Como a maioria das instruções desse computador são longas, isto é, de duas palavras, o *RI* é usado para armazenar a primeira palavra da instrução enquanto que a segunda palavra (endereço do operando) fica no *RD* até a leitura do operando. É um registrador de 8 bits provido de uma linha de *set* que permite forçar a instrução *PUG* (código 1111), necessária durante a interrupção, por um dispositivo de entrada/saída.

a.3. Unidade aritmética

A função básica da unidade aritmética é a realização das operações aritméticas e lógicas especificadas pelas instruções. São elas soma, *NAND* e *exclusive OR*, com operandos de 8 bits. A Fig. 5-10 mostra um bloco, denominado “unidade aritmética”, que realiza essas funções. Esse bloco tem largura de 8 bits.

A unidade aritmética, porém, precisa executar uma outra função: operar em endereços, somando o índice ao endereço ou somando um ao contador de instruções. Por isso, inclui-se em paralelo com a unidade aritmética um circuito chamado “mais um” (*one-upper*), que trata dos 4 bits mais significativos dos endereços. Com essa providência, consegue-se somar um número de 12 bits (endereço) com um de 8 bits (índice). Poderíamos dizer que o somador é “desequilibrado”, ou seja, ele soma uma parcela de 8 bits com outra de doze. Por isso, ele foi projetado de modo a fazer a soma dos 8 bits da primeira palavra com os oito menos significativos da segunda, num somador com propagação de “vai um” convencional e o “vai um” dessa soma de 8 bits é introduzido a um circuito que apenas soma esse sinal aos 4 bits mais significativos da segunda palavra (Fig. 5-13).

Para se fazer subtração, usamos as portas de seleção que estão incluídas na entrada do somador (Fig. 5-10, em blocos, e Fig. 5-14, em detalhes) para complementarmos uma das parcelas e, como o código usado pelo sistema é a codificação binária complemento de dois, forçamos um na entrada “vem um” e obtemos como resultado final essa parcela com o sinal trocado. As operações lógicas são realizadas pelos próprios circuitos do somador. São elas *NAND* e *exclusive OR*.

Na Fig. 5-13 vê-se, em blocos, a unidade aritmética e, na Fig. 5-14, encontra-se um dos blocos em detalhes. O cálculo de transbordamento é feito como explicado no Cap. 3.

a.4. Portas de seleção

Na Fig. 5-15, encontra-se o esquema de uma porta de seleção. Esse circuito é usado sempre que existem duas (ou mais) palavras, distintas, possíveis de serem colocadas na entrada de um circuito qualquer. Essas palavras são colocadas nas entradas da porta de seleção cuja saída é ligada na entrada do circuito citado.

Esse circuito tem o seguinte comportamento: com *E1/S* e *E2/S* no nível “zero”, a saída (*S*) é nula. Quando *E1/S* fica igual a “um”, a via de entrada *E1* é colocada na via de saída (*S*). Com *E2* é análogo. Nessa arquitetura, existem seis portas de seleção.

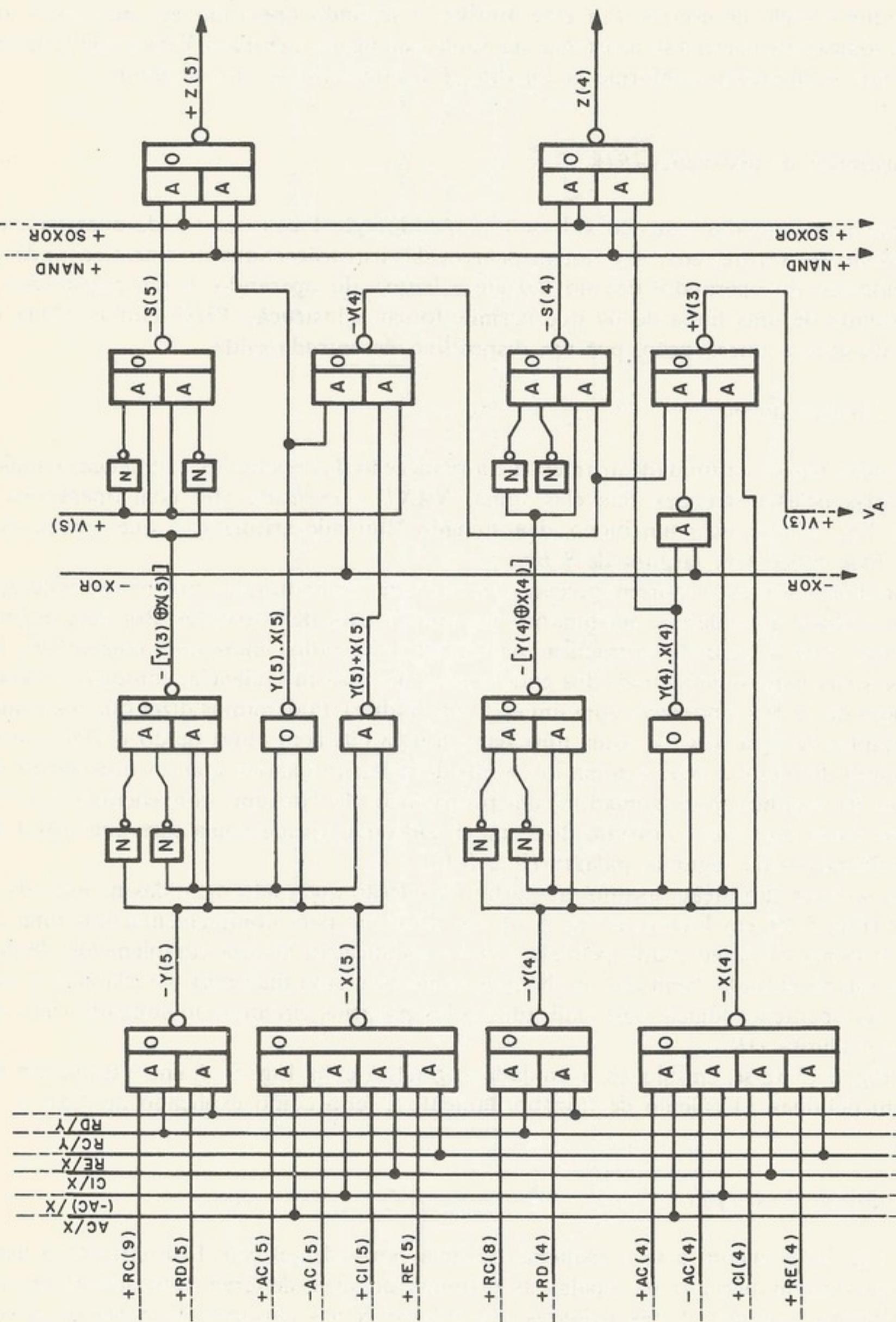
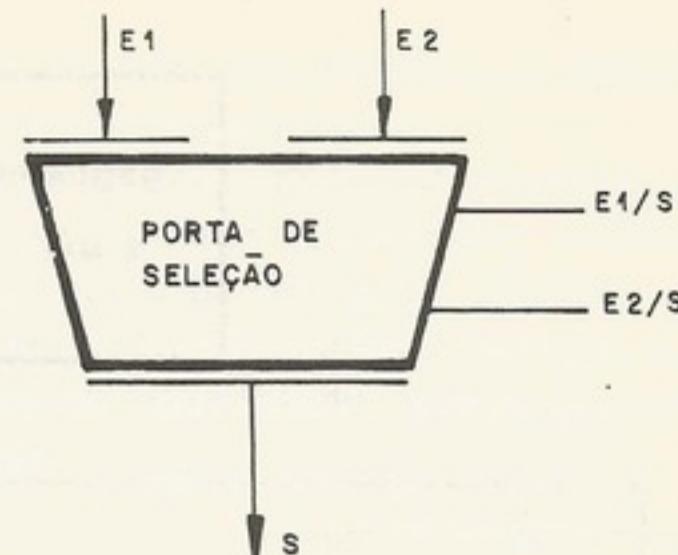


Figura 5-14. Unidade aritmética, detalhes do bloco indicado na Fig. 5-13

Figura 5-15. ...

Figura 5-15. Esquema de uma porta de seleção



Porta de seleção 1

Controla a entrada do circuito “mais um”. Permite que se coloquem na via M as seguintes palavras:

- $RI(4-7)$, para endereçar o operando;
- $CI(0-3)$, para somar um ao CI ou transportar o CI para RE ;
- $RE(0-3)$, para somar o índice de registro ao endereço;
- $RC(0-3)$, para endereçar pelo painel.

Porta de seleção 2

Controla a entrada Y do somador. Seleciona entre duas palavras:

RD , nas operações aritméticas;

$RC(4-11)$, para introduzir dados através das chaves do painel, ou junto com a porta de seleção 1 para endereçar pelo painel.

Porta de seleção 3

Controla a entrada X do somador. Na maioria das vezes, funciona junto com a porta de seleção 1 ou 2. Seleciona uma das quatro seguintes entradas:

- AC , para operar com o acumulador;
- \overline{AC} , para complementar o acumulador;
- $RE(4-11)$, para somar o índice ao endereço;
- $CI(4-11)$, para somar um ao CI , ou transportar CI para RE .

Porta de seleção 4

Controla a entrada de RD , selecionando uma das seguintes três entradas:

Via Z , para operandos gravados na memória;

$Lwbo$, para palavras lidas na memória;

Via U , utilizada pela instrução de PUG .

Porta de seleção 5

Controla o endereçamento da memória. Possibilita as seguintes três situações:

saída zero, para endereçar o índice de registro;

saída um, para endereçar a extensão do acumulador;

saída RE , nos endereçamentos normais, onde a memória é endereçada pelo RE .

Porta de seleção 6

Controla a entrada do acumulador, selecionando entre via Z (funcionamento normal) a via E (caso de entrada de dados).

Figura 5-14. Unidade aritmética, detalhes do bloco indicado na Fig. 5-13

exemplo de um

de tempo menor

múltiplo dessas un

O somador é
é feito partindo-se
para um registrador
tanto define-se com
500 ns. Como ver
dependendo da

b. Implementação

Dividiram-se
da memória. Na
xima de pastilhas
da placa etc. O cri
atingir o máximo di
tante foi tentar a
do layout e ate

FR1-1, regis

FR2-2, regis

FS1-3, somad

FS2-4, somad

FAC-5, accu

FM1-6, "mem

FCM-7, "mem

FIN-8, simula

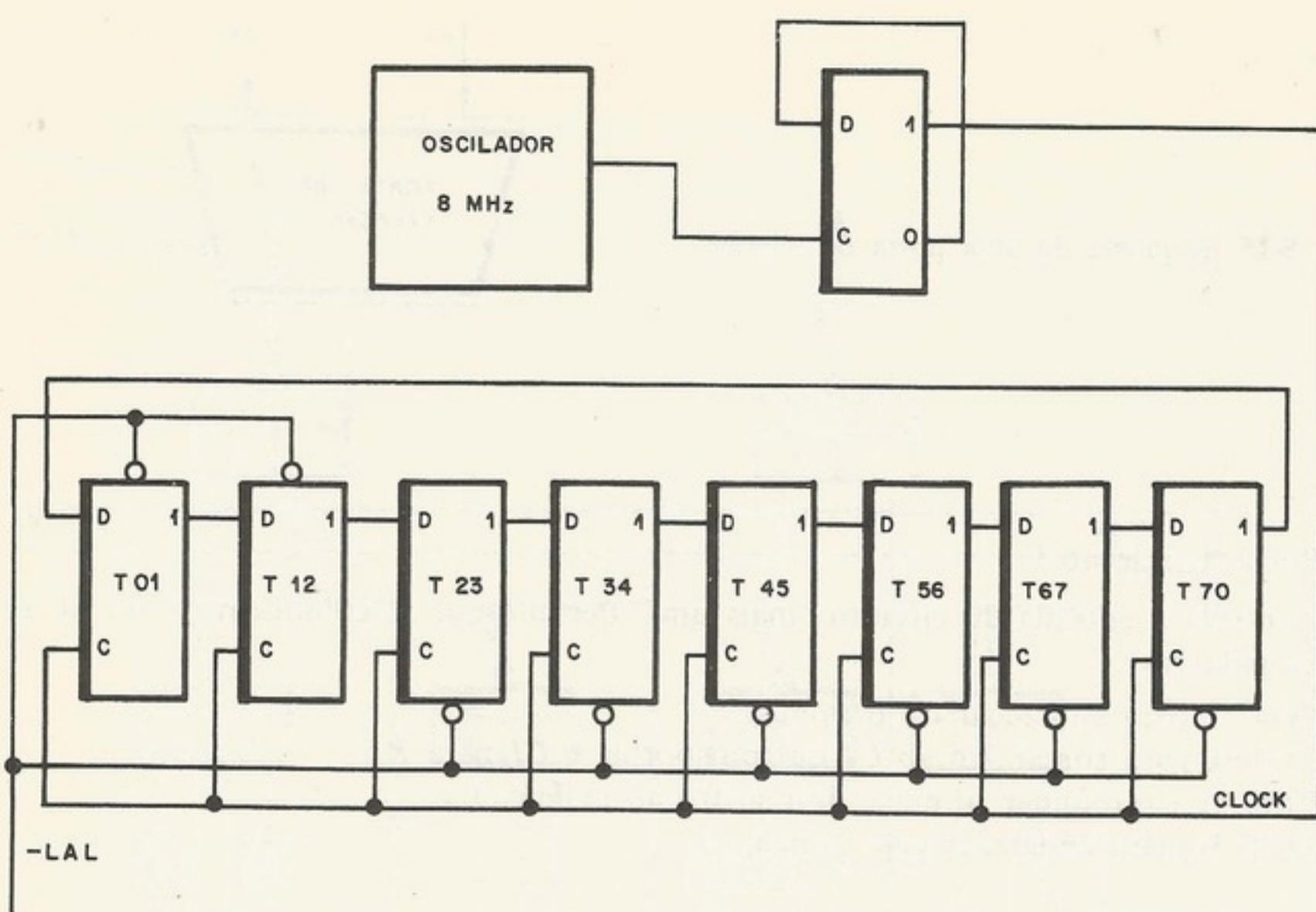


Figura 5-16. Esquema simplificado do relógio central

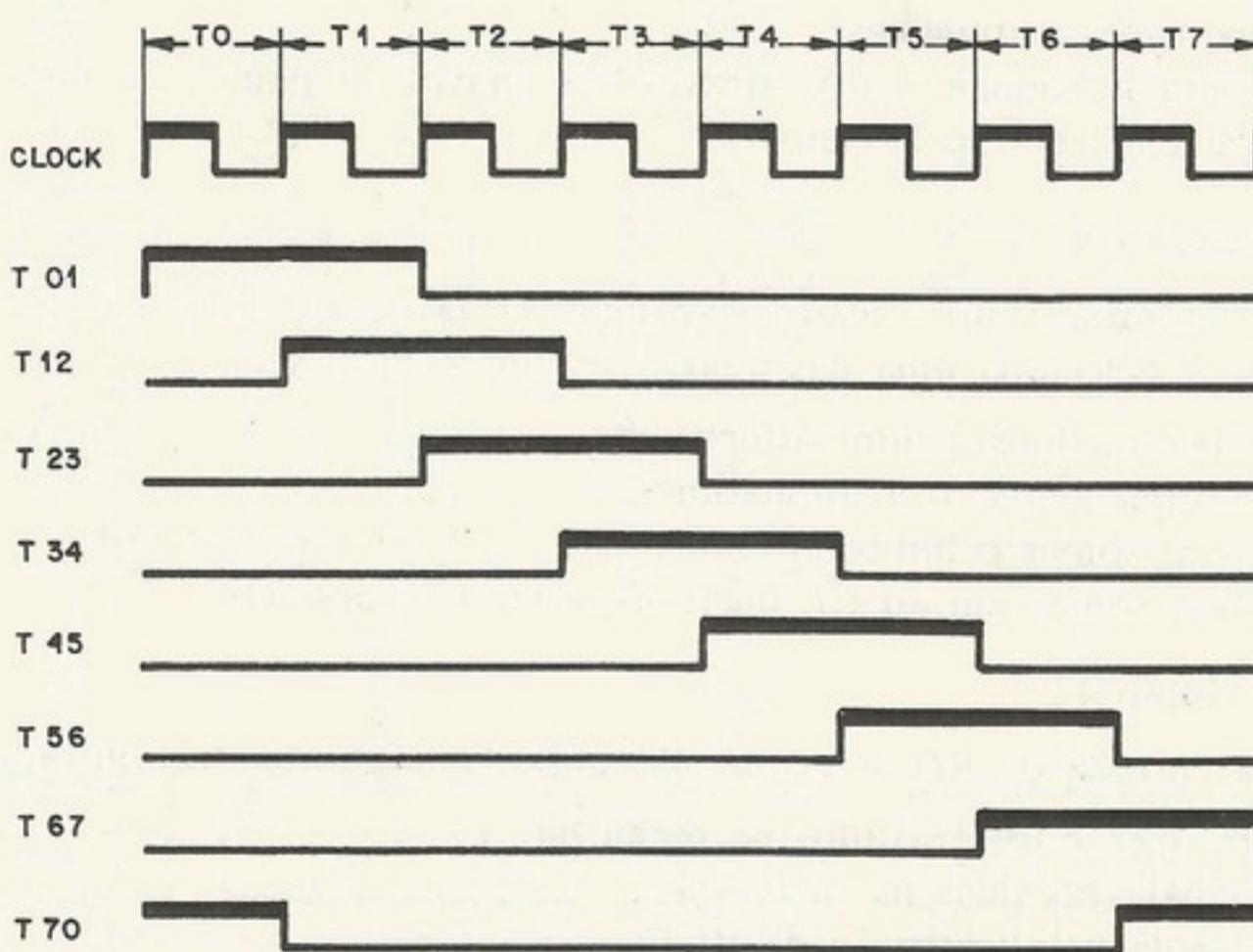


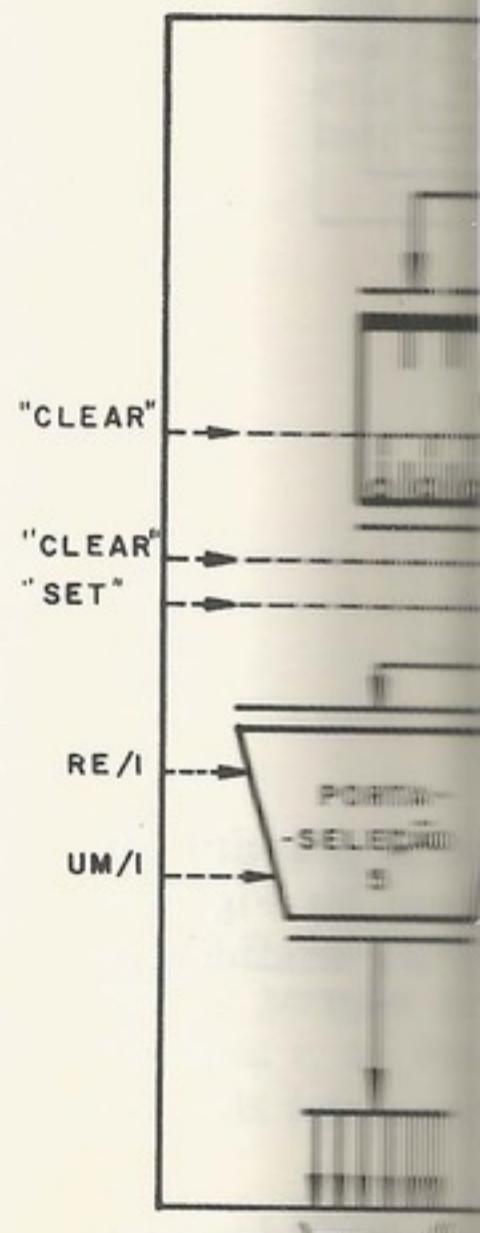
Figura 5-17. Gráfico de timing dos sinais do relógio central

a.5. Ciclo da máquina

Todos os pulsos elétricos usados pela unidade de controle para gerar sinais de *clock* ou sinais de comando de portas provêm de uma pequena unidade funcional chamada *relógio central*.

O relógio central do microcomputador do LSD está mostrado na Fig. 5-16. Basicamente, ele é constituído de um oscilador a cristal e de um *overlaped ring counter*, isto é, um deslocador em anel que funciona girando dois "1", consecutivos, conforme se vê na Fig. 5-17.

Os sinais gerados pelo relógio central são periódicos e, a esse período ($2 \mu s$), damos o nome de ciclo da memória. As oito partes do ciclo da memória (T0 a T7) são as unidades



de tempo menores possíveis do sistema, ou seja, dois eventos não-simultâneos distam um múltiplo dessas unidades, com raras exceções, que mostraremos adiante.

O somador é o caminho mais importante do fluxo de dados. Um *loop* do fluxo de dados é feito partindo-se de um registrador, passando-se pela unidade aritmética e voltando-se para um registrador. Esse caminho tem um atraso da ordem de 400 ns, no pior caso. Portanto define-se como ciclo da máquina um período igual a duas unidades de tempo, ou seja 500 ns. Como veremos adiante, os ciclos de máquina são distribuídos convenientemente, dependendo da instrução dentro do ciclo de memória.

b. Implementação dos circuitos

Dividiram-se os circuitos do fluxo de dados em 8 placas de circuito impresso, além da memória. Na partição, levaram-se em conta inúmeros vínculos, como quantidade máxima de pastilhas de circuitos impressos por placa, número máximo de entradas e saídas da placa etc. O critério de partição foi a economia de pinos nos conectores, ou seja, tentou-se atingir o máximo de circuitos com um mínimo de entrada e saída da placa. Outro fator importante foi tentar, à medida do possível, fazer placas iguais, para se economizar na preparação do *layout* e arte final do circuito impresso. A partição resultou nas seguintes placas:

- FR1-1*, registradores, bits pares;
- FR2-2*, registradores, bits ímpares;
- FS1-3*, somador, bits menos significativos;
- FS2-4*, somador, bits mais significativos;
- FAC-5*, acumulador;
- FM1-6*, "mais um" e porta de seleção 6;
- FCM-7*, interface com memória;
- FIN-8*, sinais para cartões de interface.

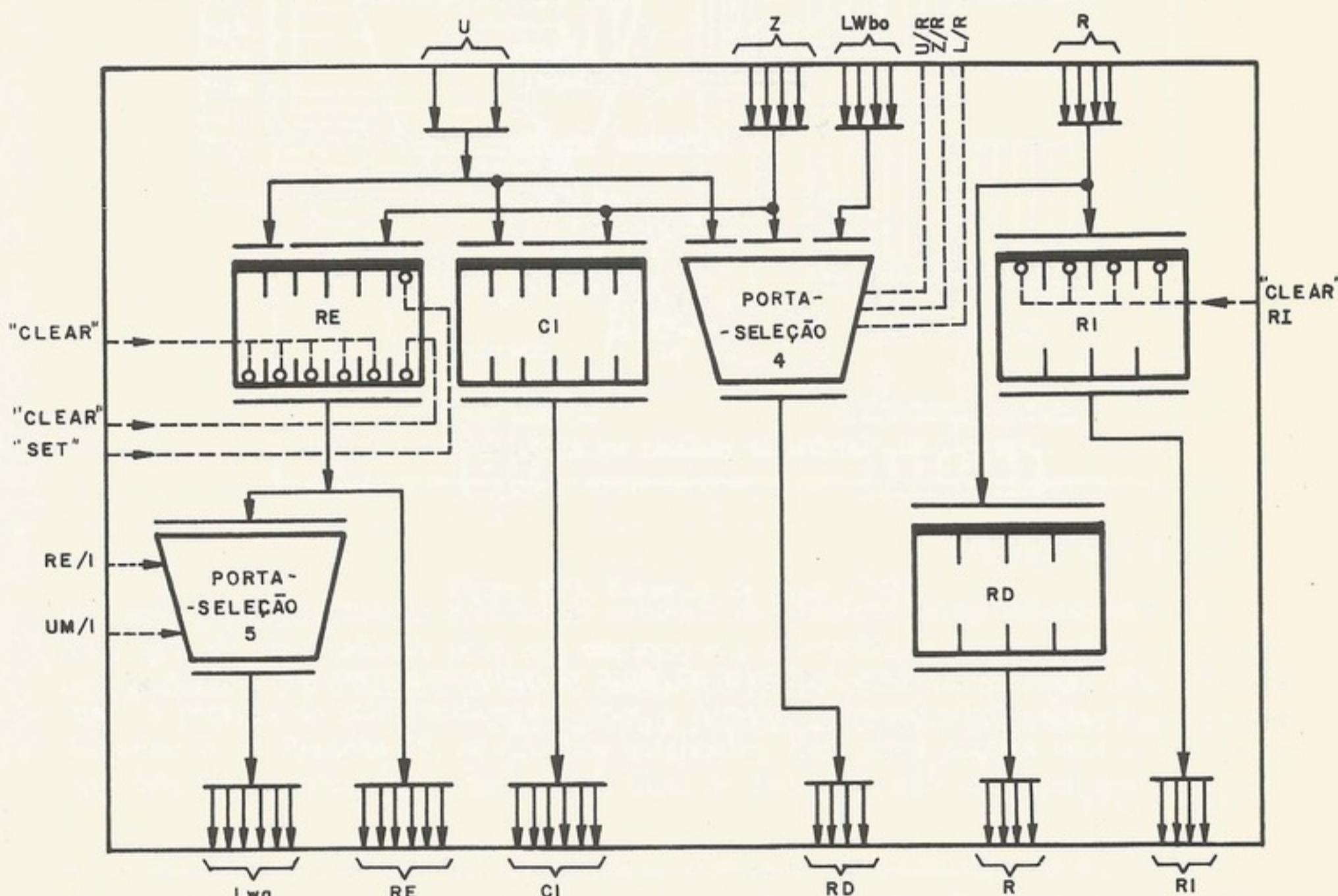


Figura 5-18. Cartão dos registradores (*FR1-1, FR2-2*)

gerar sinais de clock
chamada relógio

Fig. 5-16. Basicamente,
é um deslocador
na Fig. 5-17.
período (2 µs), damos
(II)) são as unidades

São esses os oito cartões que compõem o fluxo de dados. Segue-se uma breve descrição de cada um deles. Nos esquemas apresentados a seguir, as linhas são de dados e as linhas pontilhadas de controle.

b.1. FR1-1 e FR2-2

São duas placas de circuito impresso, iguais. A primeira com os bits das posições pares e a segunda das posições ímpares. Na Fig. 5-18 encontra-se um esquema do circuito dessa placa. Observar que ela inclui os registradores CI , RE — com sua linha de controle, (*set-reset*) para forçar 2 —, RI com outra linha de controle (*set*) para forçar o PUG , e RD . Há, ainda, nessa placa, as portas de seleção 4 e 5. Na Prancha I, encontra-se o circuito em detalhes.

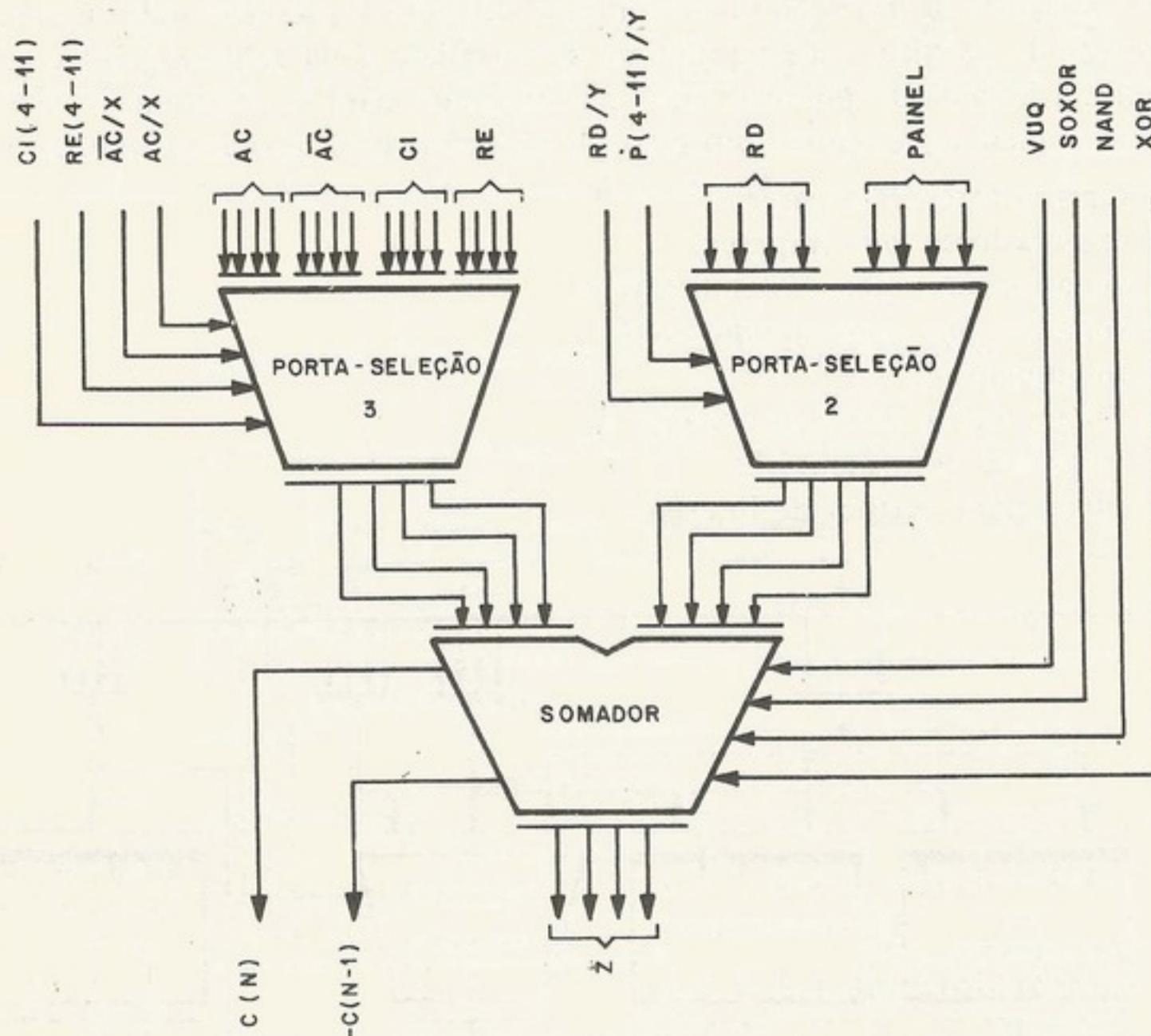


Figura 5-19. Cartões de unidade aritmética (FS1-3 e FS2-4)

b.2. FS1-3 e FS2-4

São dois cartões iguais contendo os circuitos da unidade aritmética. O primeiro corresponde aos bits menos significativos e o segundo aos mais significativos. Na Fig. 5-19, encontra-se o esquema em bloco de seu conteúdo: portas de seleção 2 e 3 e unidade aritmética. Na Prancha II, vêem-se os detalhes desse circuito.

b.3. FAC-5

O *FAC-5* é um registrador deslocador, o acumulador. Colocou-se o acumulador numa só placa, junto com os *flip-flops* V e T , já que sua estrutura não é uniforme a ponto de ser “quebrada em duas” iguais. A Fig. 5-20 mostra a placa em blocos e a Prancha III em detalhes.

-C-DIFF
-CLEAR-HIGH
-SET-HIGH

b.4. FMD
É a placa um”, e gerenciando esquema e, na que nela fizeram

b.5. FCM
É a placa corretamente

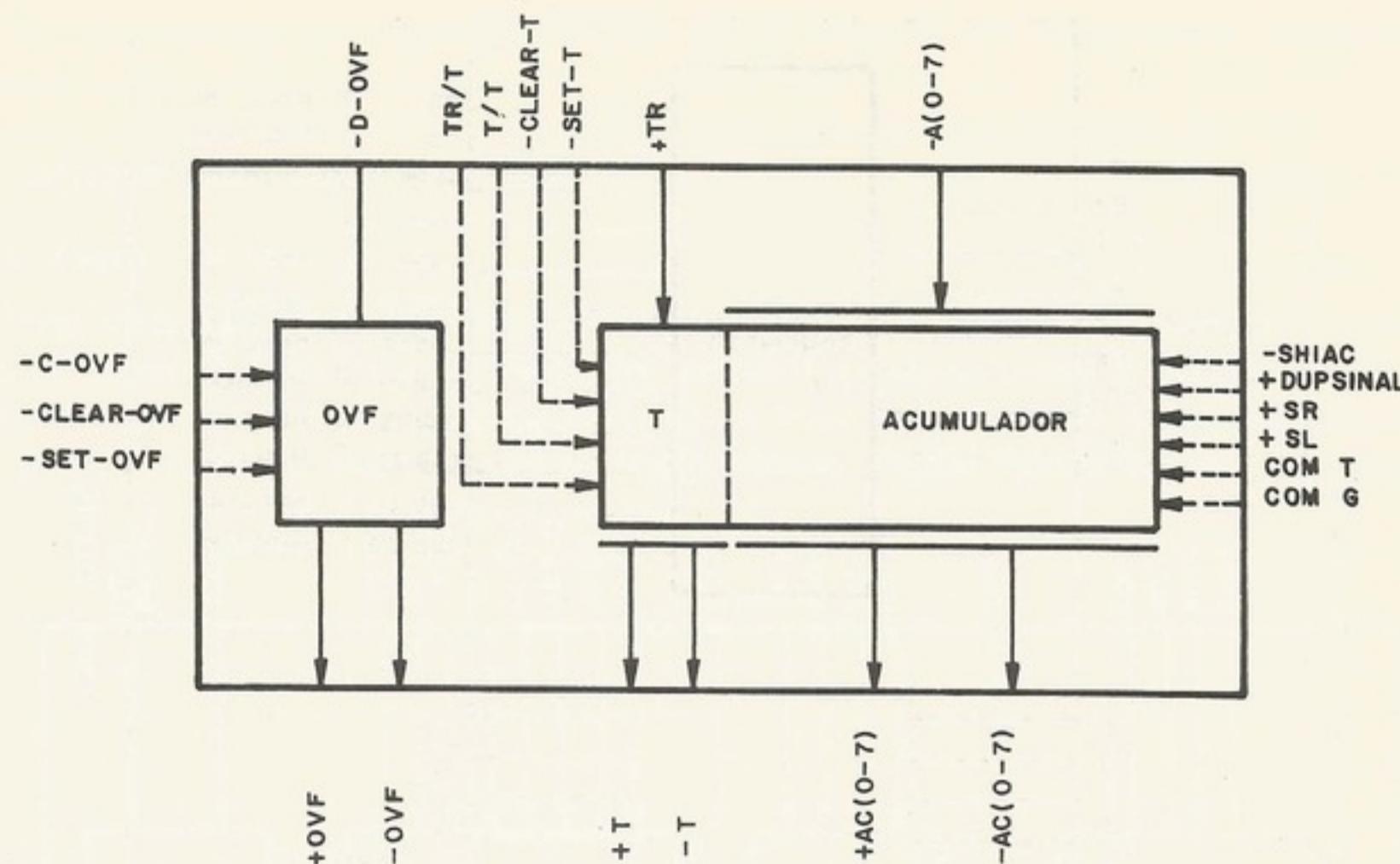
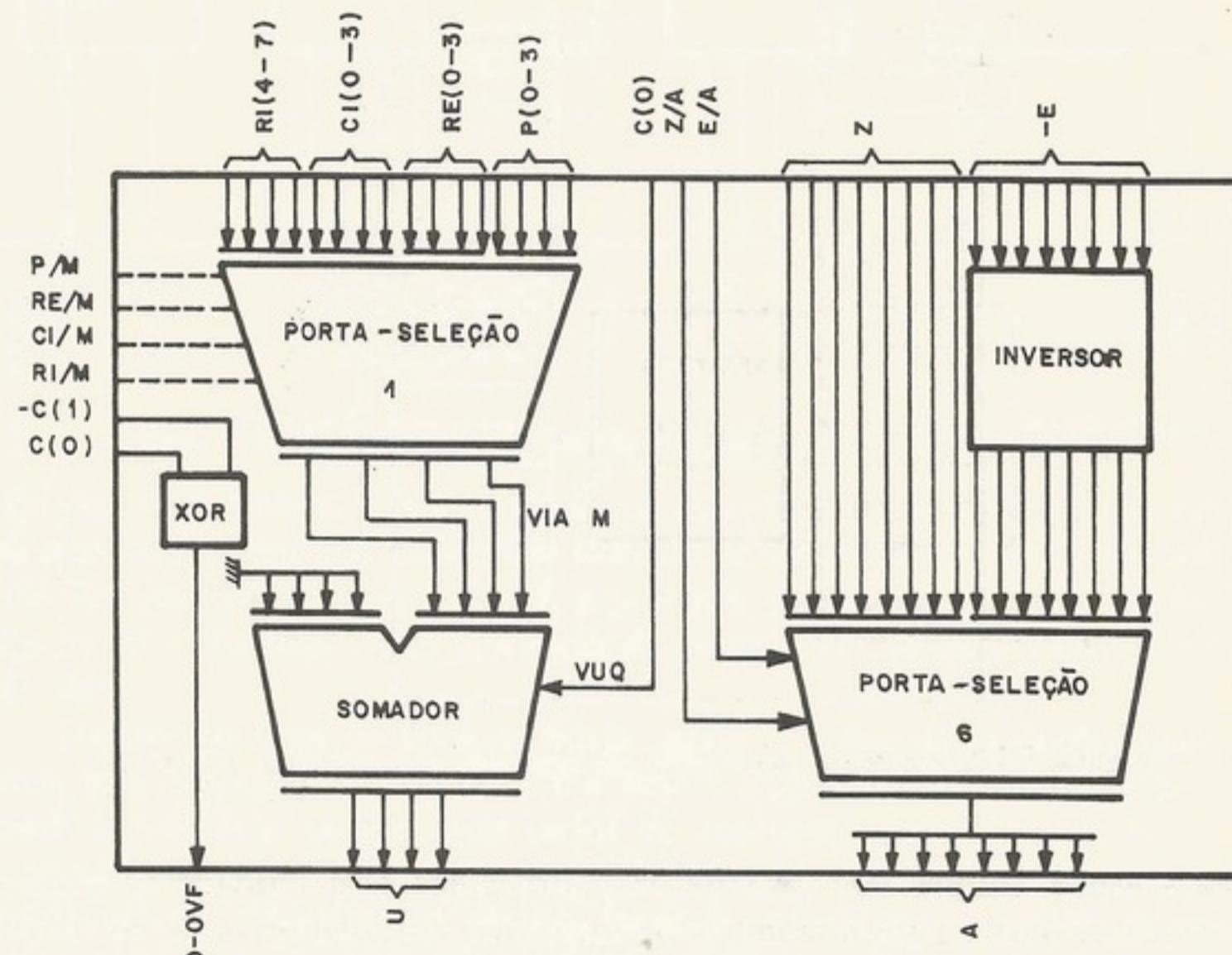


Figura 5-20. Esquema da placa do acumulador (FAC-5)



b.4. FM1-6

É a placa que tem os circuitos da porta de seleção 6, porta de seleção 1, circuito "mais um", e geração do transbordamento para o flip-flop *T* (*overflow*). Na Fig. 5-21 vê-se um esquema e, na Prancha IV, os circuitos detalhados. Essa placa contém uma miscelânea já que nela foram colocados os pedaços do fluxo de dados que não couberam em outro lugar.

b.5. FCM-7

É a placa que gera os controles da memória. Suas funções são as seguintes: endereçar corretamente os módulos da memória (quatro); coordenar proteção das 128 últimas po-

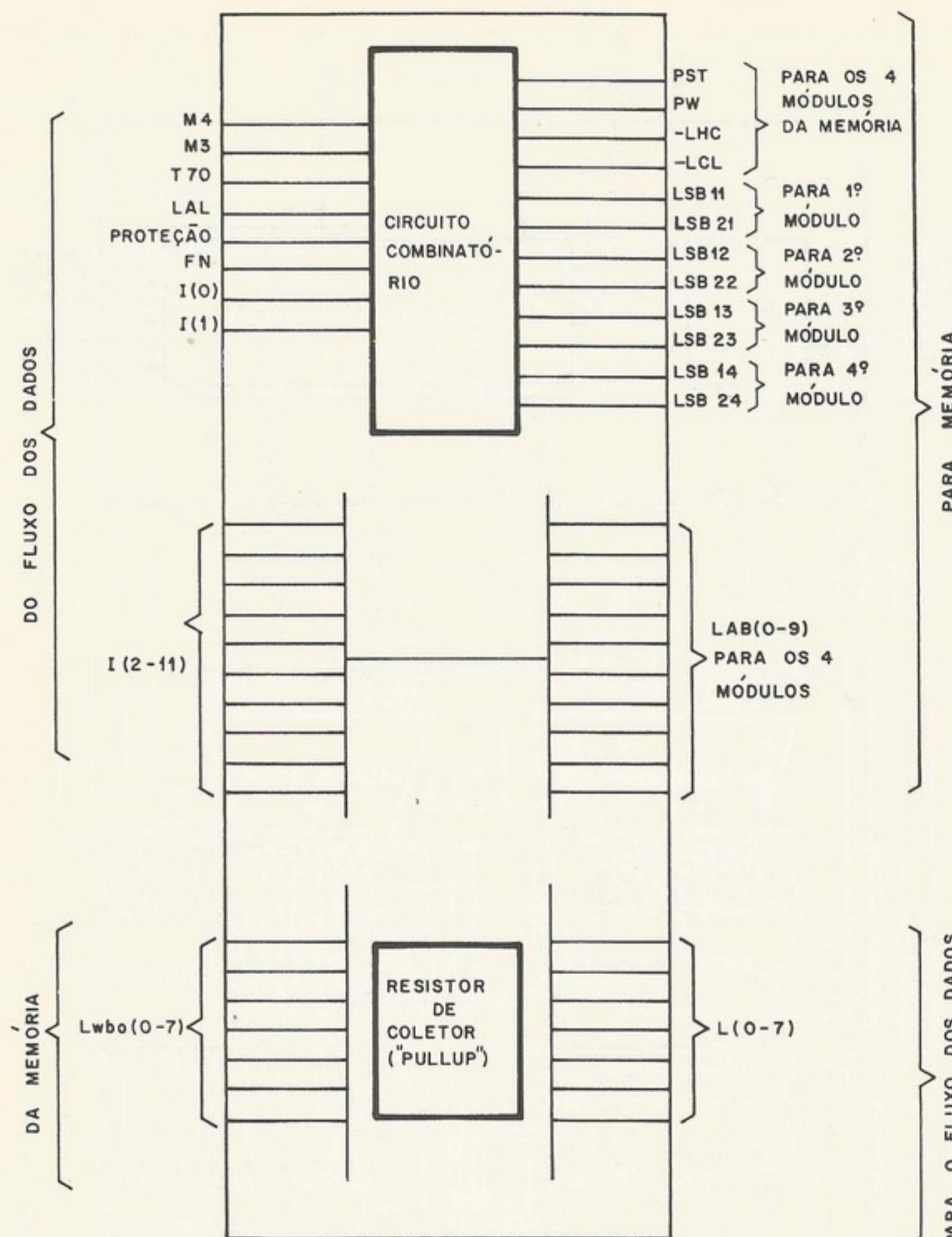


Figura 5-22. Esquema do cartão de controle de memória (FCM-7)

sições; com os sinais simples M_1 , M_3 ou M_4 , fornecidos pela unidade de controle, gerar todos os sinais necessários para o controle da memória; e, ainda, preparar os sinais de saída da memória (agrupando as saídas dos módulos num só ponto). A Fig. 5-21 mostra o esquema e a Prancha V os detalhes.

b.6. FIN-8

Para enviar os sinais para os cartões de interface, deve-se fazer com que as vias passem por portas que se abrem apenas quando é uma instrução de entrada e saída. Isso evita que existam sinais nas vias tal quando não é necessário, já que esses sinais geram ruídos. Essa é a função da placa FIN-8, além de decodificar o endereço do dispositivo, controlar a interrupção pelo canal zero (pelo painel ou por falta de força), além de aumentar a potência dos sinais de saída. Na Fig. 5-23 existe um esquema desse cartão e, na Prancha VI os circuitos em detalhes.

exemplo de um m...

Os outros ca...
só bloco o fluxo

5.5 FASES E TI

a. Fases

Uma instru...
ciclos de memóri...
são as fases 1 e
ções, por exem...



é, apenas a funç...
endereçar a ins...
dois ao CI. Em...
e colocada no ...
locada no ...

Dep...
especificada pa...
dois acessos a ...

No caso di...
cessidade da ...
sião 0) e sum...
Resumindo,

Durante a ...
da unidade de ...
para a fase 2, d...

exemplo de um minicomputador

Os oito cartões são interligados pelo painel traseiro do computador, formando um só bloco: o fluxo de dados.

5.5 FASES E MICROOPERAÇÕES

a. Fases

Uma instrução, para ser executada, passa por várias etapas. É necessário um ou dois ciclos de memória para ler a instrução da memória e colocá-la nos registradores *RI* e *RD*; são as fases 1 e 2 normalmente conhecidas como *fases de fetch* ou de busca. As microinstruções, por terem apenas uma palavra de comprimento, exigem apenas um ciclo de *fetch*, isto

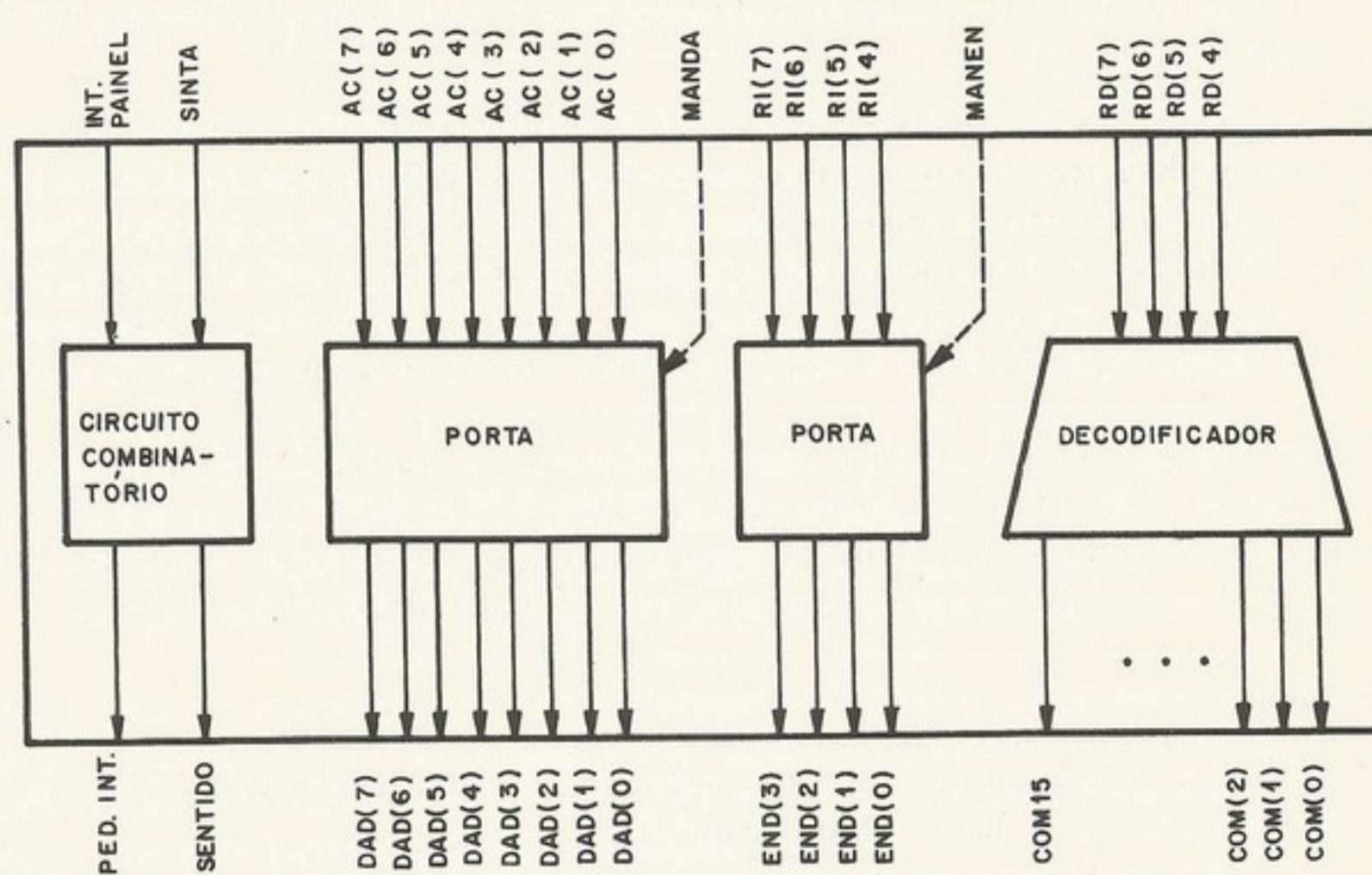


Figura 5-23. Esquema do cartão dos sinais de interface (FIN-8)

é, apenas a *fase 1*. É durante a fase de *fetch* que somamos um ao contador de instrução, para endereçar a instrução seguinte. Instruções longas têm duas fases de *fetch*; portanto somamos dois ao *CI*. Em resumo, durante a fase 1, a primeira palavra de instrução é lida da memória e colocada no *RI* e, durante a fase 2, a segunda palavra de instrução é lida da memória e colocada no *RD*.

Depois das fases de *fetch* ou busca, passa-se à fase de execução, onde a operação especificada pela ilustração é executada; é a *fase 3*. Algumas instruções, por necessitarem de dois acessos à memória, exigem dois ciclos de execução; *fase 3* e *fase 4*.

No caso de instruções indexadas, antes de entrarem nas fases de execução, existe a necessidade do cálculo do endereço efetivo, lendo o registrador de índice da memória (posição 0) e somando-o no endereço especificado. Existe uma fase especial para isso, a *fase 5*. Resumindo,

- fase 1} fases de *fetch* ou busca;
- fase 2}
- fase 3} fases de execução;
- fase 4}
- fase 5} fase de cálculo de endereço efetivo.

Durante a execução de um programa, a *UCP* vai pulando de fase em fase, sob o comando da unidade de controle. Por exemplo, da fase 1 de uma instrução de “soma indexada”, evolui para a fase 2, daí para a fase 5 e volta para a fase 3. Para isso, existem cinco sinais elétricos,

mutuamente exclusivos, que são gerados durante uma fase para indicar qual a fase seguinte. São eles CF_1, CF_2, CF_3, CF_4 e CF_5 (condições para a fase 1, condições para a fase 2 etc.).

Quando existe uma instrução "pare" ou "espere" são bloqueadas todas as fases e o sistema fica sem fazer nada.

b. Microoperações

Cada instrução, em cada fase, é composta de inúmeros passos, seqüenciais no tempo, chamados de microoperações. Por exemplo, são microoperações:

incrementar um no contador de instruções, $CI \leftarrow CI + 1$;
transferir o conteúdo do contador de instruções para o registrador de endereço, $RE \leftarrow CI$;
pular para a fase 3, $F3 \leftarrow 1$.

Cada instrução, desde sua fase 1 até sua última fase, é executada, como já dissemos, pela combinação de microoperações. Por exemplo, a instrução que já analisamos, "soma indexada", tem quatro ciclos de memória, ou seja, fase 1, fase 2, fase 5 e fase 3. Cada fase dessas é composta de várias microoperações, que se sucedem ao tempo. Para a documentação da máquina, é importante que se conheçam, para cada instrução, suas fases e microoperações. Por isso, nas tabelas apresentadas a seguir (quatro), observam-se as cartas das microoperações.

Observar nas tabelas que algumas instruções estão agrupadas em algumas fases, visto que, então, elas se comportam da mesma maneira. Isso ajuda muito no projeto de unidade de controle que irá gerar sinais elétricos para a execução dessas microoperações. Nessas tabelas, o eixo horizontal é a variável tempo, nas unidades da máquina ($0,25 \mu s$).

c. Exemplo

Para ilustrar as tabelas, vejamos como o sistema executa a instrução de "soma indexada", já tão comentada. A fase 1 dessa instrução é

T0	T1	T2	T3	T4	T5	T6	T7
Memória: ler e restaurar							
	Lwbo $RI \leftarrow$				$RE \leftarrow CI$	$F2 \leftarrow 1$	
	$CI \leftarrow CI + 1$						

Inicia-se o ciclo lendo-se a memória. Nessa fase, a primeira palavra da instrução é lida na memória e, no final do tempo $T1$, é colocada em RI . Enquanto isso, incrementa-se um no contador de instrução. No final desse ciclo ($T5$ e $T6 = T56$), colocamos o conteúdo de CI em RE para endereçar a segunda metade da instrução e, em $T7$, dispararmos a fase 2, que será o ciclo de memória seguinte. É importante notar que, até $T2$, o sistema desconhecia a particular instrução que iria ser executada (no nosso exemplo, $SOMX$) e, por isso, *todas* as instruções são iguais na fase 1 e nos tempos $T0$, $T1$ e $T2$.

A fase 2 de $SOMX$ será

T0	T1	T2	T3	T4	T5	T6	T7
Memória: ler e restaurar							
					$RE \leftarrow RI(4-7)$ e RD		$F5 \leftarrow 1$
	RD $Lwbo$						
	$CI \leftarrow CI + 1$						

Nesse ciclo, lemos a segunda palavra da memória e colocamo-la em *RD*, enquanto somamos um ao *CI* (para endereçar a próxima instrução). No final do ciclo, montamos o endereço [*RI*(4-7) e *RD*] no registrador (*RE*), que serve para endereçar o operando, caso não seja indexado, ou para somar com o registrador de índice, caso seja indexado. Liga-se então a fase 5, já que é indexada.

A fase 5, que é a mesma para as quatro instruções indexadas é

<i>T0</i>	<i>T1</i>	<i>T2</i>	<i>T3</i>	<i>T4</i>	<i>T5</i>	<i>T6</i>	<i>T7</i>
Memória: ler e restaurar							
	<i>RD</i> <i>Lwbo</i>				<i>RE</i> \leftarrow <i>RE</i> + <i>RD</i>		<i>F3</i> \leftarrow 1
<i>0</i> \rightarrow <i>Lwa</i>							

Iniciamos lendo a memória e forçando o endereço “0” pela porta de seleção 5. No fim de *T1*, temos o índice em *RD*, que é somado com *RE*, sendo a soma guardada no próprio *RE* no final do ciclo. Portanto o endereço efetivo do operando já está em *RE*. A seguir, em *T7*, forçamos a fase 3.

A fase 3, de execução é

<i>T0</i>	<i>T1</i>	<i>T2</i>	<i>T3</i>	<i>T4</i>	<i>T5</i>	<i>T6</i>	<i>T7</i>
Memória: ler e restaurar							
	<i>RD</i> <i>Lwbo</i>			<i>AC</i> \leftarrow <i>AC</i> + <i>RD</i>	<i>RE</i> \leftarrow <i>CI</i>		<i>F1</i> \leftarrow 1

Aqui, lemos o operando que fora endereçado no final do ciclo anterior. Em *T34*, esse operando é somado com o acumulador e o resultado guardado no acumulador. No final do ciclo, colocamos o conteúdo de *CI* em *RE* que corresponde a endereçar a próxima instrução, já que esta terminou. Então forçamos a fase 1.

5.6 UNIDADE DE CONTROLE

a. Considerações gerais

A unidade de controle é o centro motriz do sistema. É ela que gera todos os sinais elétricos no momento e no lugar certo para a realização das microoperações que, combinadas, resultam nas instruções.

Essa unidade tem como informação os sinais de tempo do relógio central, as fases e a instrução corrente. É, basicamente, um circuito combinatório, montado com portas da família TTL 400 de circuitos integrados, que gera os sessenta sinais que abrem portas, disparam flip-flops, dão clock em registradores, forçam um na entrada *VE* da unidade aritmética etc. A combinação conveniente desses sinais elétricos produz uma microoperação que é parte de uma instrução.

Por exemplo, os sinais

- CI(0-3)/M* (abre a porta de seleção 1 para o *CI(0-3)*),
- CI(4-11)/X* (abre a porta de seleção 3 para o *CI(4-11)*),
- VU* (força 1 na entrada *VE* do somador),
- SOXOR* (estrutura de soma na unidade aritmética),

seguidos, depois de 500 μ s do *clock*, em *CI*, perfazem microoperação:

$$CI \leftarrow CI + 1$$

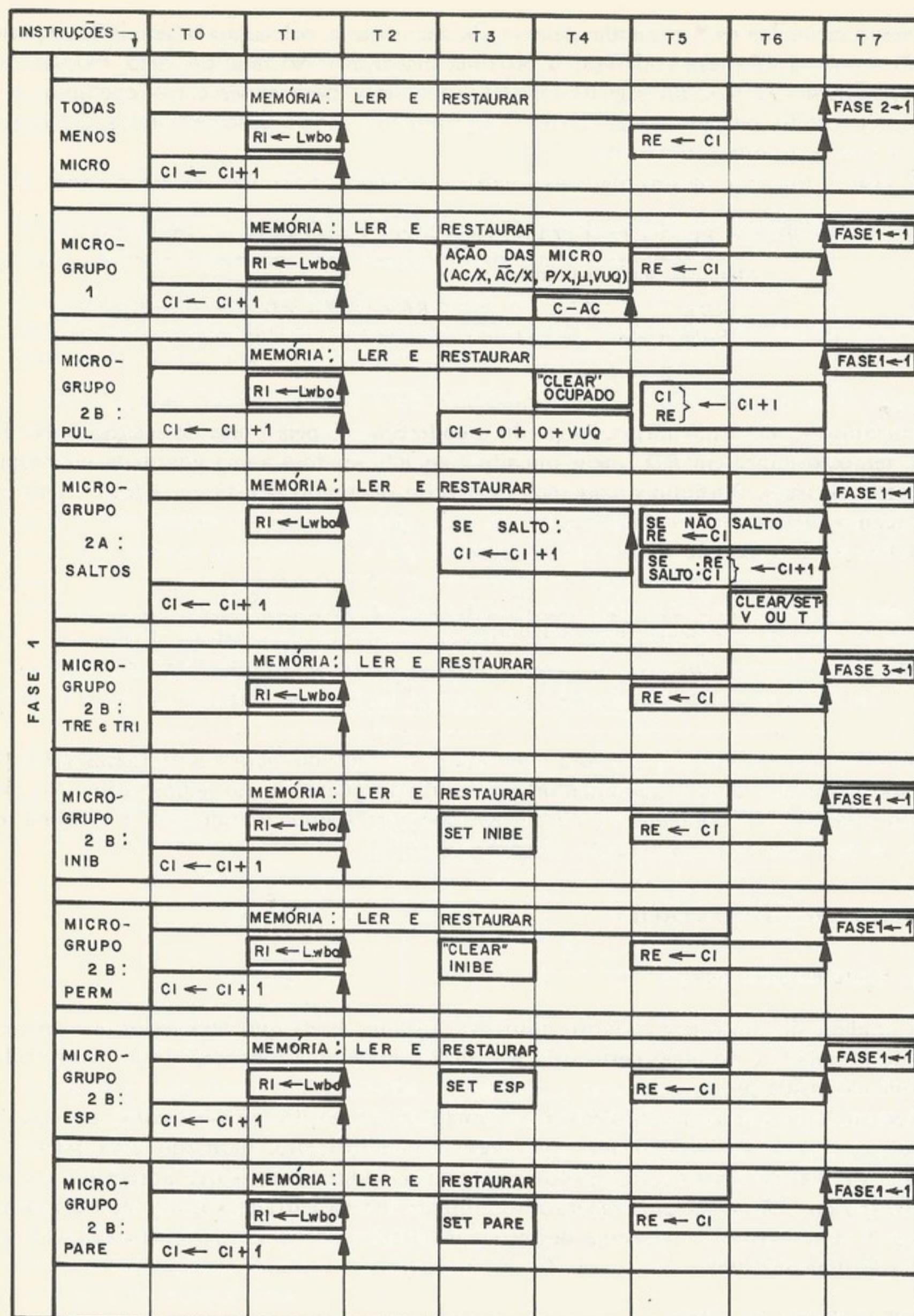


Figura 5-24. Carta de microoperações da fase 1

Portanto o projeto de unidade de controle é feito tendo-se por base as cartas de microoperações. Temos as condições e os tempos em que devem ocorrer as microoperações. Portanto tiramos das cartas as condições e tempos nos quais os sinais elétricos de controle devem ser "1" ou "0". O problema seguinte será o projeto lógico do circuito que gera esses sinais.

INSTRUÇÃO →	T0	T1	T2	T3	T4	T5	T6	T7
CAR ARM (COM OU SEM ÍNDICE)		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1				RE ← RI(4-7) e RD		SEM ÍNDICE FASE 3-1 COM ÍNDICE FASE 5-1
SUS PUG		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1				RE ← RI(4-7) e RD		FASE 3-1
PLA PLAX		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1				RE ← RI(4-7) e RD CI ← RI(4-7) e RD		SEM ÍNDICE FASE 1-1 COM ÍNDICE FASE 5-1
PLAN		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1				SE AC = 0: RE ← RI(4-7) e RD SE AC ≠ 0: RE ← CI		FASE 1-1
PLAZ		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1				SE AC = 0: RE ← RI(4-7) e RD SE AC ≠ RE ← CI		FASE 1-1
NAND XOR		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1		AC ← AC { SOMA NAND } RD		RE ← CI		FASE 1-1
DESLOCA- MENTOS		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1	SE RD(4)=1 C - AC	SE RD(5)=1 C - AC	SE RD(6)=1 C - AC	SE RD(7)=1 C - AC		FASE 1-1
SAL		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1	BUS Z+U ← CI + 1 SE SENTIDO: SET S SINTA = 1	SE S=1: RE ← CI + 1 SE S=1: C - CI SE S=D: RE ← CI				FASE 1-1
FNC		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1				RE ← CI		FASE 1-1
SAI		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1		DADOS SAÍDA		RE ← CI		FASE 1-1
ENT		MEMÓRIA : LER E RESTAURAR RD ← Lwbo CI ← CI + 1		AC ← E		RE ← CI		FASE 1-1

Figura 5-25. Carta de microoperações da fase 2

INSTRUÇÕES	T0	T1	T2	T3	T4	T5	T6	T7
FASE 3	CAR CARX	MEMÓRIA: LER E RESTAURAR	RD ← Lwbo	AC ← RD	RE ← CI		FASE1 ← 1	
	ARM ARMX	MEMÓRIA: LER SOMENTE	RD ← AC		RE ← CI		FASE 1 ← 1	
	SOM SOMX	MEMÓRIA: LER E RESTAURAR	RD ← Lwbo	AC ← AC + RD	RE ← CI		FASE 1 ← 1	
	PUG	MEMÓRIA: LER SOMENTE	RD ← CI(0-3)		RE ← RE + 1		FASE 4 ← 1	
	TRE	MEMÓRIA: LER SOMENTE	O/Lwq				FASE 4 ← 1	
	TRI	MEMÓRIA: LER SOMENTE	O/Lwq				FASE 4 ← 1	
	SUS	MEMÓRIA: LER SOMENTE	RD ← Lwbo	RE ← RD			FASE 1 ← 1	
FASE 4	TRE	MEMÓRIA: ESCREVER						
	TRI	MEMÓRIA: ESCREVER						
	PUG	MEMÓRIA: ESCREVER						
FASE 5	CARX ARMX SOMX PLAX	MEMÓRIA: LER E ESCREVER					SE PLI: FASE1 ← 1	
		O/Lwq					SE NÃO PLI: FASE 4 ← 1	
		RD ← Lwbo			RE ← RE + RD			
					SE PLI 1 C ← RE + RD			

Figura 5-26. Carta de microoperações das fases 3, 4 e 5

b. Exemplo

Como exemplo bastante simples, veja que o conteúdo da inspeção na carta de condições:

instrução de LER
instrução de T
instrução de S
instruções de I
instruções de R
T34;

instrução de S

Na Fig. 5-27, os combinatórios que em cinco placas de

**c. Decodificação**

Para a gerar corrente. Para isso bits do RI, que em grande parte, e a decodificação a esses grupos de RI, no caso de servir que têm a camutação.

d. Controle de

Durante o fun de máquina. Chama busca ou ciclos II.

b. Exemplo

Como exemplo do circuito da unidade de controle, vejamos um dos sinais de geração bastante simples, o sinal *AC/X*, ou seja, aquele que age na porta de seleção 3, fazendo com que o conteúdo do acumulador seja colocado na entrada *X* do somador. Por uma simples inspeção na carta de microoperações, vemos que esse sinal elétrico deve existir nas seguintes condições:

- instrução de *ARM* ou *ARMX*, na fase 3, em *T12*;
- instrução de *TRE* ou *TRI*, na fase 4, em *T12*;
- instrução de *SUS*, com $RD \neq 0$, na fase 3, em *T23*;
- instruções do grupo *MICG1*, com $RI(5) = 1$, na fase 1, em *T34*;
- instruções de *SOMI* ou *NAND* ou *XOR*, e não *SOMI* e *XOR* juntos na fase 2, no tempo *T34*;
- instrução de *SOM* ou *SOMX*, em *T34*, da fase 3.

Na Fig. 5-27, encontra-se o circuito combinatório que gera o sinal *AC/X**. Os circuitos combinatórios que geram os sinais de controle do fluxo de dados foram implementados em cinco placas de circuitos. As Pranchas de XII até XVI mostram os detalhes dos circuitos.

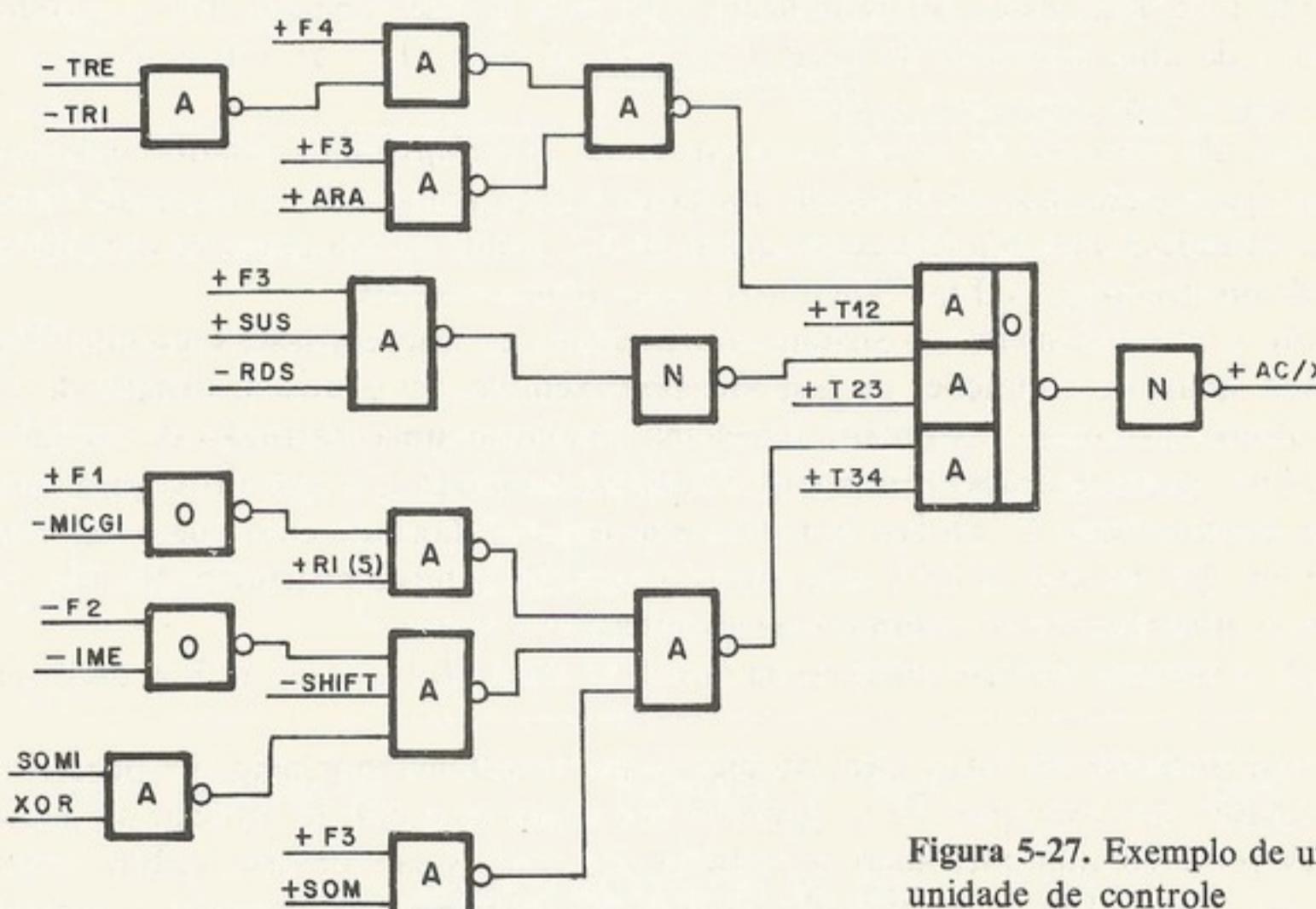


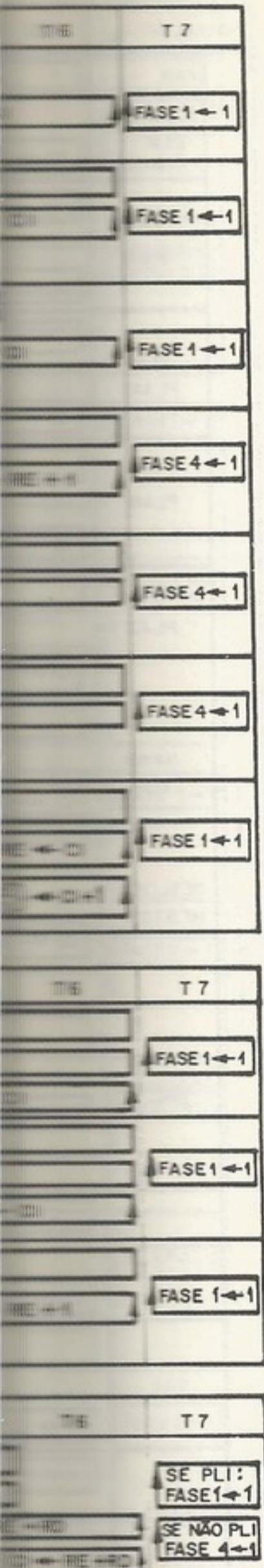
Figura 5-27. Exemplo de um sinal da unidade de controle

c. Decodificador

Para a geração correta dos sinais de controle, é importante que se saiba a instrução corrente. Para isso, existe, na unidade de controle, um decodificador, cuja entrada são os bits do *RI*, que armazena a primeira palavra de instrução. Como muitas instruções têm, em grande parte, as mesmas microoperações, estas agrupam-se em códigos convenientes e a decodificação em árvore (veja o Cap. 3) é eficiente, pois gera sinais de decodificação comum a esses grupos de instruções. Por exemplo, a decodificação do bit 0 (mais significativo) de *RI*, no caso de ser igual a 0, informa-nos que é uma das instruções *PLA*, *SOM*, *CAR* ou *ARM*, que têm a característica comum de serem indexáveis.

d. Controle de estado

Durante o funcionamento normal, podem-se distinguir cinco tipos diferentes de ciclo de máquina. Chamam-se de fase 1, fase 2, fase 3, fase 4 e fase 5. As fases 1 e 2 são ciclos de busca ou ciclos *I*. Na fase 1, a primeira palavra da instrução é lida na memória e, na fase 2,



a segunda palavra da instrução. A fase 3 é o ciclo de execução, onde é lido ou gravado o operando na memória e, como algumas instruções exigem dois acessos à memória para serem executadas, existe a fase 4. Instruções indexadas exigem um ciclo de máquina diferente, para ler o índice na posição zero da memória e calcular o endereço efetivo; é a fase 5.

Note-se que o simples controle das fases resolve o problema da seqüencialização de programa. Se, em toda a fase 1, é lida na memória a primeira palavra de uma instrução, endereçada pelo contador de instruções, e se, em toda a fase de busca, atualizar-se o contador de instruções, bastará, após se executar cada instrução, encaminhar-se, automaticamente, para a fase 1, para que a seqüencialização do programa seja garantida.

Para o controle das fases, o "gerador de sinais de controle" fornece cinco sinais, mutuamente exclusivos,

- CF1*, condição para fase 1;
- CF2*, condição para fase 2;
- CF3*, condição para fase 3;
- CF4*, condição para fase 4;
- CF5*, condição para fase 5;

Durante uma fase, esses sinais indicam qual será a fase seguinte. Por exemplo, se se está na fase 2 de uma instrução de *CAR*, *CF3* está no nível "1" e os outros no nível "zero", indicando que a fase seguinte é a fase 3.

Na placa de controle de estado, existem cinco *flip-flops* cujos estados indicam fases, um *flip-flop* para cada fase. O controle das fases é feito colocando-se os sinais "condições de fase" na entrada *D* dos *flip-flops* e no início de *T7*, dando-se um pulso na entrada de controle (*clock*) do *flip-flop*. A Fig. 5-28 ilustra essa idéia.

Durante o funcionamento, o sistema vai pulando de fase em fase, seguindo o que lhe dita os cinco sinais de condições de fase. Se, por exemplo, após uma instrução de *PUG* (a fase 4 é a última fase dessa instrução), o sistema encontrar uma instrução de "soma", ocorrerá o seguinte: durante a fase 1, o sinal *CF2* estará no nível 1, o que provocará, em *T7*, o disparo do *flip-flop* "fase 2"; na fase 2, *CF3* é que será "1", fazendo com que o *flip-flop* "fase 3" se ligue em *T7*; na fase 3, *CF1* será ativado para ler a nova instrução. Na Fig. 5-8 tem um diagrama que ilustra, no tempo, essa evolução.

Ao ligar o sistema, deve-se iniciá-lo na fase 1. O sinal *-LAL*, visto na Fig. 5-28, tem essa função.

Com essa filosofia de fases, fica simples parar o sistema no meio do processamento (instrução "pare", por exemplo). Se o "gerador dos sinais de controle" utilizar *sempre* o sinal de fase para produzir qualquer sinal, a inibição dos cinco sinais de fase, inibirá todo o sistema. O sinal "roda" (Fig. 5-28), quando igual a "0", inibe as fases parando o computador.

É controlando o sinal "roda" que se operam os modos especiais de funcionamento, como o ciclo único e a instrução única. Esse sinal é composto por três outros (Fig. 5-29), "roda 1", "roda 2" e "roda 3".

"Roda 1"

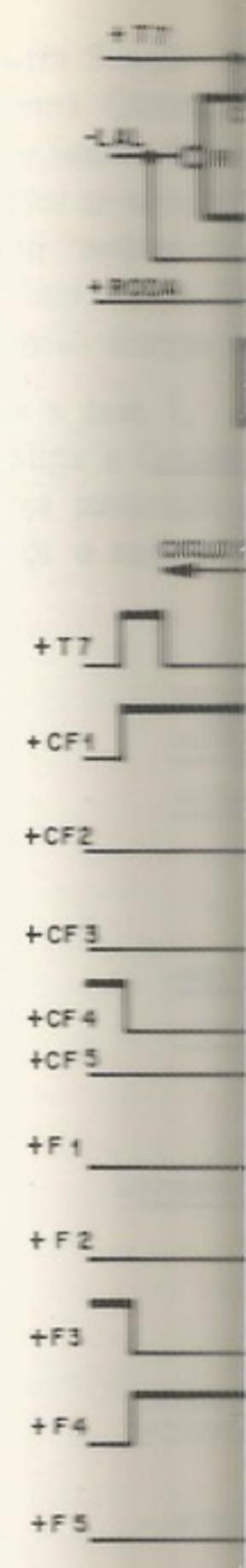
Usado durante o modo normal de funcionamento. Fica sempre no nível "1" até que se pare o sistema. As condições para ligá-lo e desligá-lo são as que seguem.

Ligar. Quando se aciona o botão de partida ou quando se está no estado de espera e chega um pedido de interrupção.

Desligar. Sempre antes de uma fase 1, nas seguintes circunstâncias: instrução de "pare", instrução ou acionamento do "espere" ou tirando-se do modo normal de funcionamento.

"Roda 2"

Usado no modo especial "instrução única" (IS). É ligado quando se está nesse modo de funcionamento e se pressiona o interruptor de partida. É desligado sempre que se vai entrar numa fase 1.



"Roda 3"

Utilizado no ...
um ciclo. Utiliz...
Uma tarefa im...

o operador e o sis...
acaba, no final, com
o flip-flop correspon...
paração" (presente) u...
o relógio central. I...

Um pouco m...
ele deve existir p...
sinal é usado em ...
-flops, sendo o pri...
"partida".

lido ou gravado o operador na memória para serem utilizadas diferentes, para o qual é a fase 5. Na sequencialização de uma instrução, endereçando-se o contador automaticamente, gera-se cinco sinais, mu-

ta. Por exemplo, se se

estados indicam fases,

segundo o que lhe

Fig. 5-28, tem essa

do processamento

nível "1" até que se

no estado de espera e

instrução de "pare",

esse modo

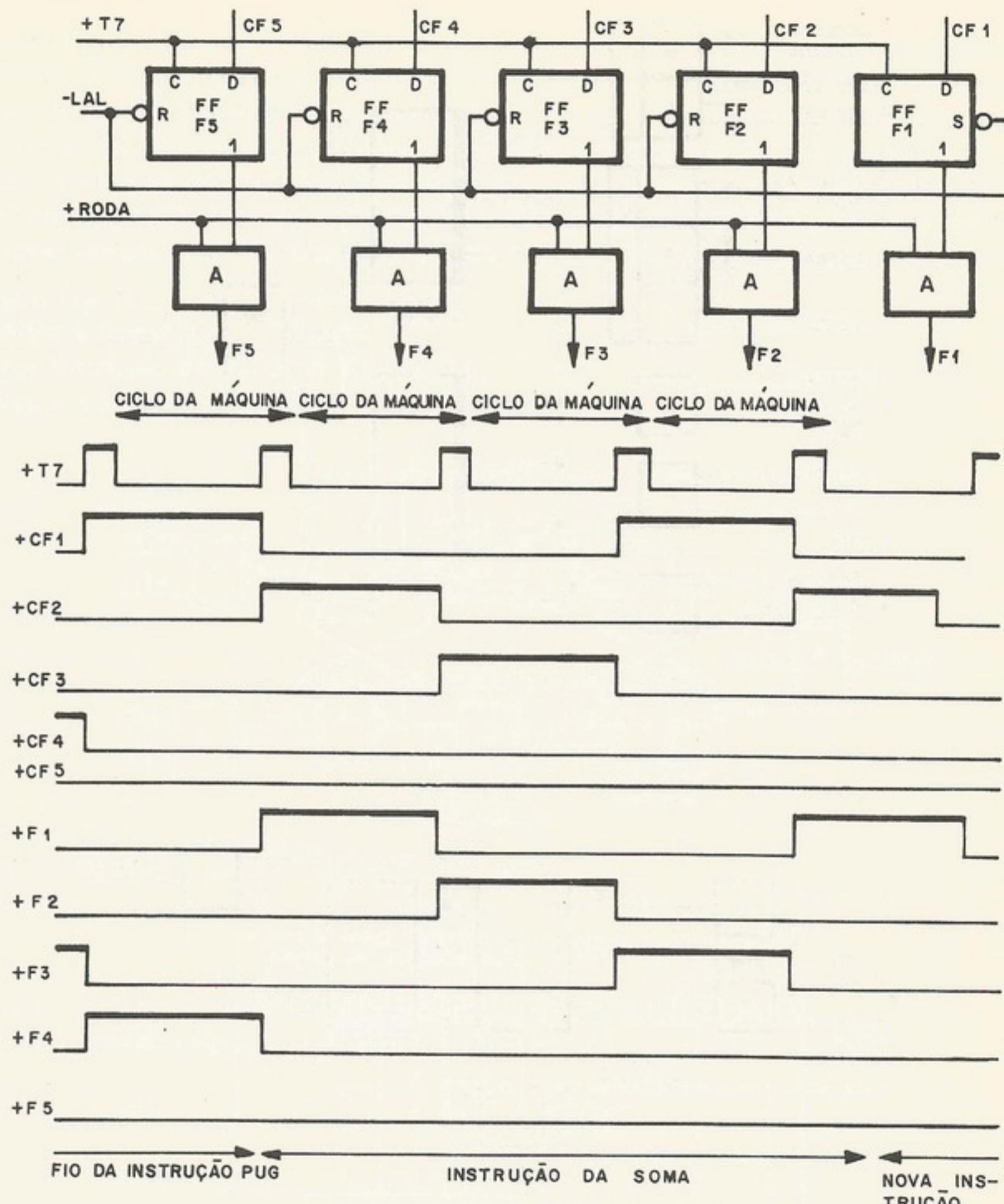


Figura 5-28. Controle das fases

"Roda 3"

Utilizado no modo especial "ciclo único" (CS). É um sinal que fica no nível 1 por apenas um ciclo. Utiliza-se o próprio sinal de partida, que dura um ciclo, para gerá-lo.

Uma tarefa importante do circuito "controle de estado" é a provisão da interface entre o operador e o sistema. Já se comentou que o operador, ao selecionar o modo de operação, acaba, no final, controlando o sinal "roda". O acionamento do interruptor "espere" irá ligar o flip-flop correspondente, que, por sua vez, desligará o sinal "roda 1". O interruptor "preparação" (preset) age diretamente no circuito de geração do LAL (limpa ao ligar), parando o relógio central, forçando a fase 1, desligando interrupção etc.

Um pouco mais delicada é a geração do sinal "partida", já que, a cada pressão da chave ele deve existir por apenas um ciclo, começando em T0 e acabando em T7. Isso porque esse sinal é usado em inúmeros pontos onde o sincronismo é vital. Um circuito com três flip-flops, sendo o primeiro eliminador de ruído (bounce) da chave, gera esse sinal, chamado "partida".

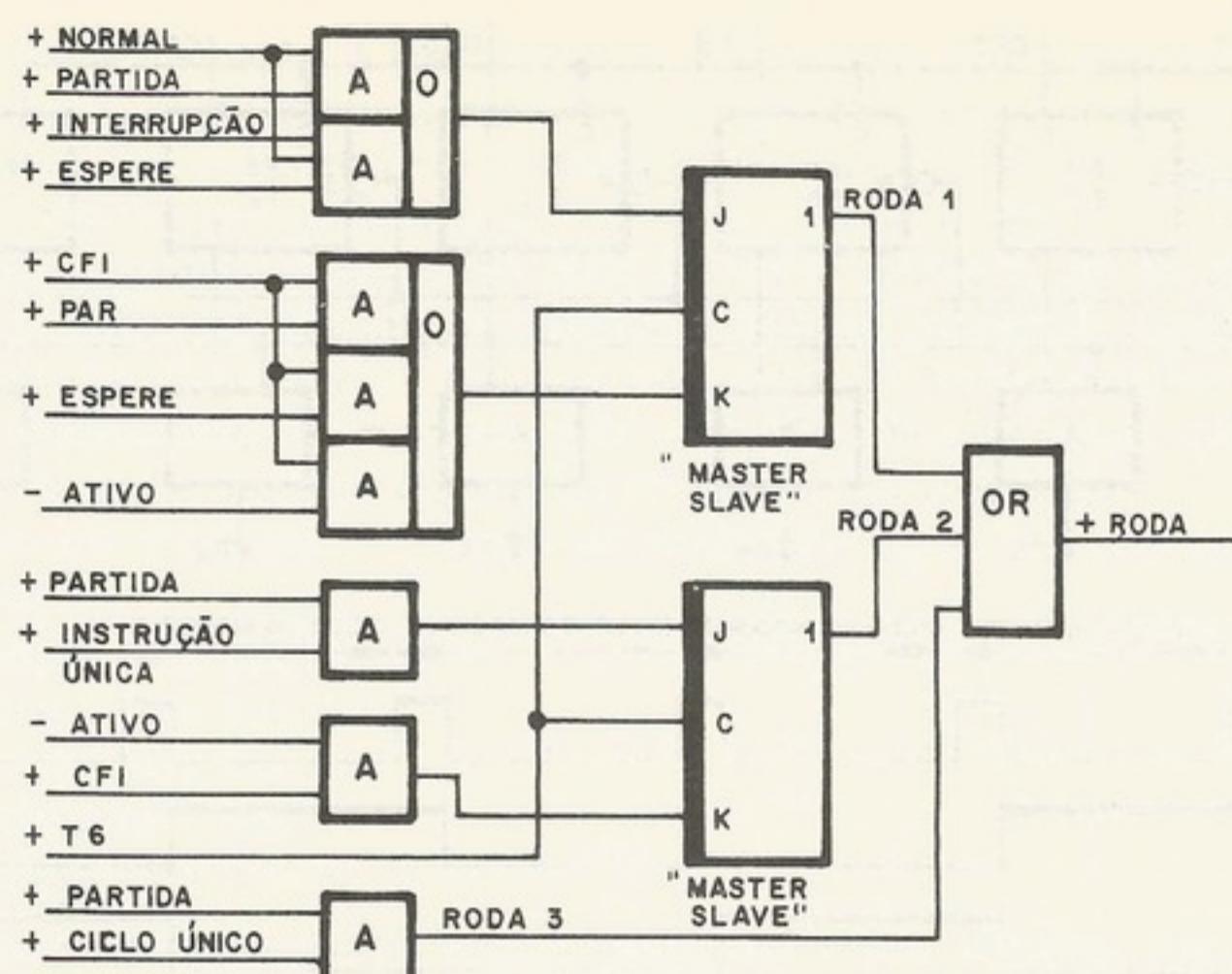


Figura 5-29. Geração do sinal "roda"

Outra função (último item). Quando a fase 1 se inicia, o circuito faz as seguintes operações (ver se não é interessante): chamado "ordem de interrupção" no tempo de tempo de tempo. Nesse meio tempo,

liga a fase 1, desliga a fase 2, força instrução, força o endereço

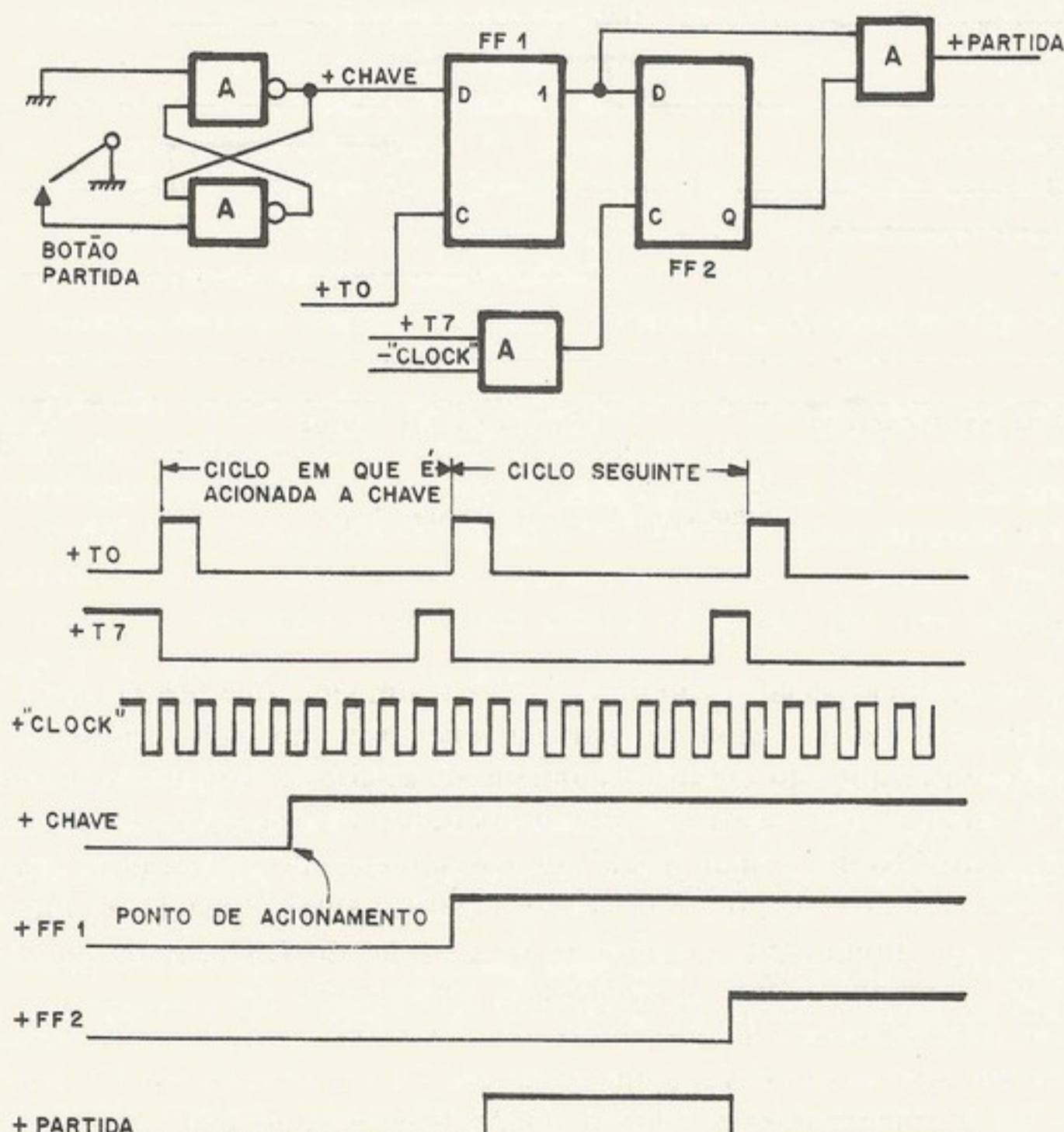
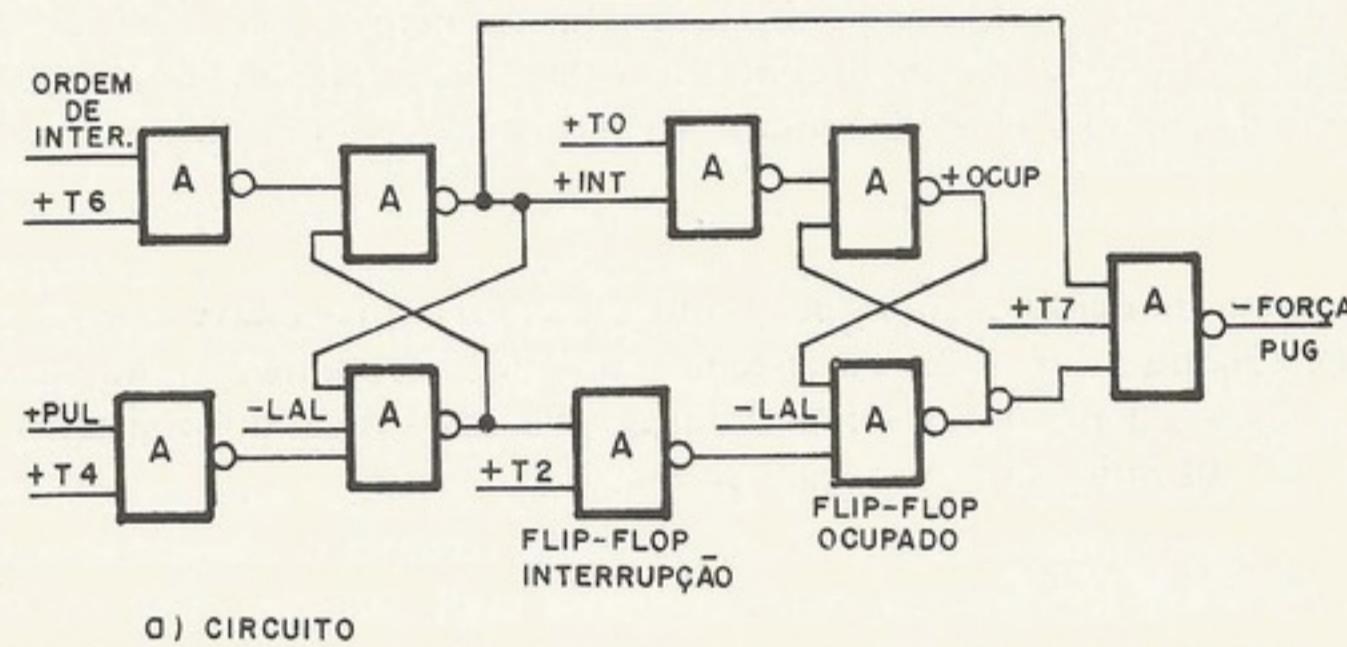


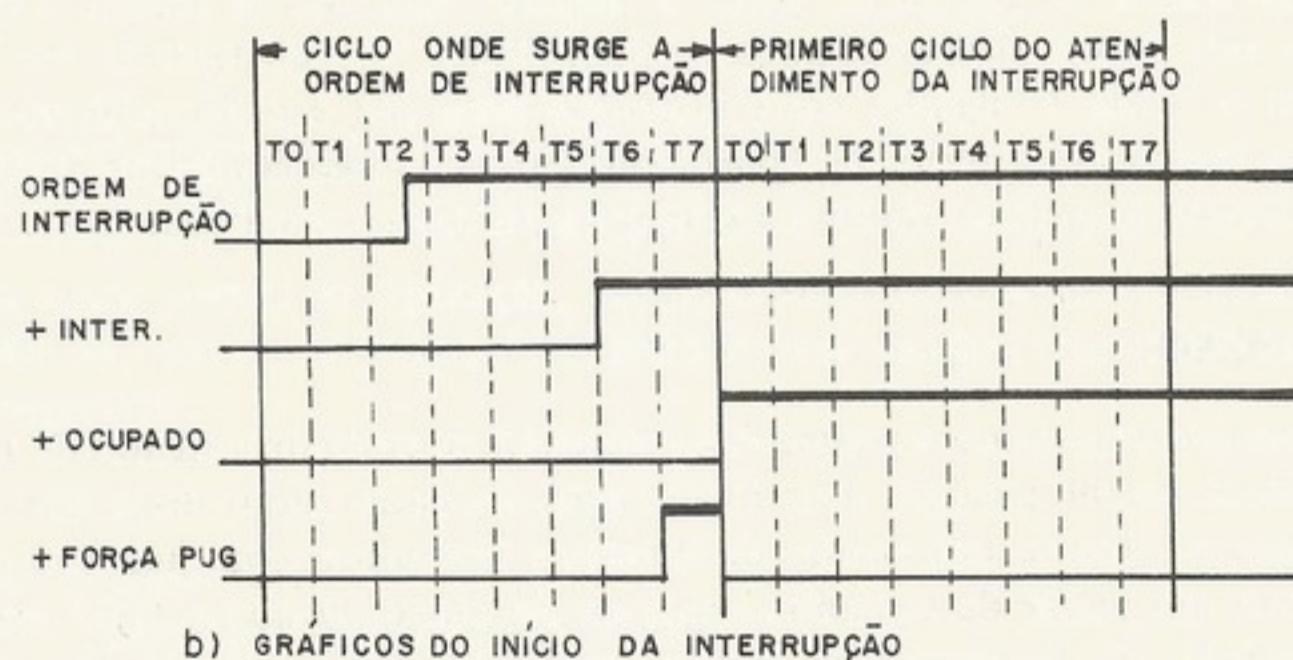
Figura 5-30. Geração do sinal "partida"

Outra função do circuito "controle de estado" é controlar a interrupção (veja o próximo item). Quando chega um pedido de interrupção, ele é atendido antes que a próxima fase 1 se inicie. O pedido de interrupção passa por algumas portas que testam certas condições (ver se não está ocupado, se não está inibido, se é fase 1 etc.) e se transforma num sinal chamado "ordem de interrupção". Esse sinal (Fig. 5-31) dispara um flip-flop chamado "interrupção" no tempo T_6 . No segmento de tempo T_0 seguinte, é disparado o flip-flop "ocupado". Nesse meio tempo (em T_7), é gerado o sinal "força PUG", que realiza o seguinte:

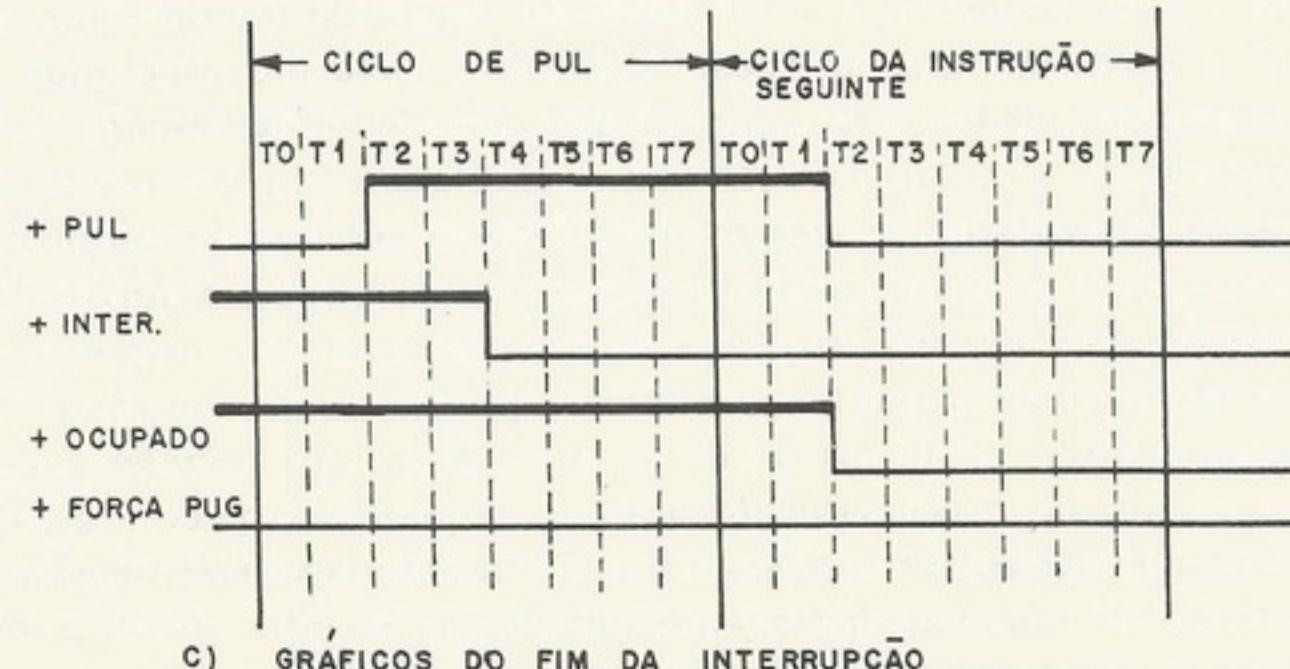
- liga a fase 3,
- desliga a fase 1,
- força instrução de PUG no RI (set RI),
- força o endereço 2 no RE.



a) CIRCUITO



b) GRAFICOS DO INÍCIO DA INTERRUPÇÃO



c) GRAFICOS DO FIM DA INTERRUPÇÃO

Figura 5-31. Mecanismo de interrupção

Isso tudo equivale a iniciar uma instrução de *PUG* para a posição 2 da memória. Os *flip-flops INT* e "ocupado" ficam ligados durante todo o tempo de atendimento da interrupção, e apenas uma instrução de *PUL* (ou acionamento da chave "preparação") os desliga. Para que não se perca o endereço de volta ao programa principal, que, durante o atendimento da interrupção, ficou armazenado na posição 2 da memória, o *flip-flop* "ocupado" é desligado no ciclo seguinte ao ciclo de *PUL*, aproveitando-se o fato de que, enquanto o *flip-flop* ocupado se mantiver no nível "1", nenhum outro pedido de interrupção será atendido.

Na Prancha XI, existem os circuitos da placa de controle de estado. Nessa prancha encontram-se alguns outros detalhes que merecem algumas palavras.

Gerador do LAL

É um circuito de componentes discretos. Basicamente é um *SCR* que, ao ligar o sistema, encontra-se cortado, fazendo com que *LAL* = "1". Ao se acionar, pela primeira vez, o interruptor de partida, o *SCR* conduz, levando o *LAL* para "0".

Sinal FN

É um sinal, chamado *fase normal*, controlado por uma chave que se encontra dentro da máquina. Quando *FN* = "1", o sistema funciona normalmente. Ao se acionar a chave fazendo *FN* = "0", deixa de existir a evolução das fases, ficando o sistema rodando preso a uma fase. É utilizado na depuração do sistema.

Sinal "partida atrasada"

É um sinal que inicia em *T6* de um sinal "partida" e acaba em *T5* do ciclo seguinte. Ele é usado para limpar os *flip-flops* "pare" e "espere". Quando se aciona o interruptor de partida para sair de um desses estados, o sinal "roda 1" é energizado no final do ciclo de partida. Com o "roda 1" ligado, nova instrução será lida e o decodificador que estava dando o sinal da instrução anterior de "pare" ou "espere" passa a indicar a nova instrução. Então, pode-se limpar (no ciclo seguinte ao sinal de partida) o estado de "pare" ou "espere".

5.7 INTERRUPÇÃO

Suponha que queiramos gravar num disco magnético, que esteja ligado ao nosso computador, vários blocos de informação. Um bloco, uma unidade que o disco grava, é formado de inúmeras palavras. Existe um cartão de interface fazendo a comunicação entre o disco e a *UCP*, que tem um *buffer* de tamanho igual ao do bloco. O que a *UCP* faz é preencher o *buffer* da interface. (O programador indica isso com uma instrução do tipo *SAI, n, c.*) Quando termina o preenchimento, a *UCP* envia ao cartão de interface um sinal de "pode gravar no disco" a partir de uma instrução tipo *FNC, n, c* e fica esperando o término da gravação para preencher o *buffer* com o novo bloco. Esse tempo de espera da *UCP* é ocioso e poderia ser aproveitado para fazer outras coisas. É aí que entra o conceito de *interrupção*.

A idéia geral de interrupção é a seguinte: a *UCP* está executando um programa normal, quando chega um pedido de interrupção do dispositivo doze (por exemplo). Então ela pára o que estava fazendo, executa um programinha especial para esse dispositivo, ao término do qual volta ao ponto onde parara e continua o processamento normal.

Vejamos agora o nosso problema do disco. A *UCP* carrega o *buffer* com um bloco de palavras e volta, no fim, ao processamento normal, enquanto o bloco é gravado no disco. Quando o disco termina de gravar o bloco, o cartão de interface *interrompe* a *UCP*, que irá parar o que estava fazendo para encher novamente o *buffer* e voltar ao seu trabalho anterior.

Evidentemente a importância desse conceito é enorme, e apenas demos uma idéia intuitiva e bastante vaga do que seja. Podemos imaginar que o sistema seja ligado a máquinas

e aparelhos de ...
deles, é interrupção
processamento num

No ...
seguir.

Como já sabem...
índice e extensão
a rotina tratad...
terminar a execu...
çada pela unidade
de set (1111 = P...
o endereço 2.

Todas essas...
2 assim que chega...
é colocado um "pu...
cessamento é des...
tratadora da inter...
de *PUL* que equi...

Quando o ...
de uma instrução...
provoca uma int...
rupção, apenas m...
geral. Por isso, é m...
sinal "ocupado" m...
um *T4*, só ocorre...

5.8 MODOS DE

Além do modo...
uma instrução de...
que são os segu...
modo 1 ...
modo 2 ...

grupo 1 ...

grupo 2 ...

Existem seis chaves...
mutuamente exclus...
modo 1 ...
modo 2 ...

a. Grupo 1 ...

"Ciclo único"

É um modo de...
tida, evoluí apena...
operador do sistema...
cipalmente, ...

e aparelhos de medições e que, quando houver alguma irregularidade com qualquer um deles, é interrompido para resolver o “galho”. *Enquanto isso não acontecer, ele atende ao processamento normal.*

No minicomputador do LSD, a interrupção funciona da maneira que expomos a seguir.

Como já sabemos, as posições 0 e 1 da memória são reservadas para o registrador de índice e extensão do acumulador. Nas posições 4 e 5, existe uma instrução de desvio para a rotina tratadora da interrupção. Quando chega um pedido de interrupção, a UCP espera terminar a execução da instrução corrente e, em vez de ir para a fase 1 da seguinte, ela é forçada pela unidade de controle a ir para a fase 3 (execução) enquanto o RI recebe um pulso de set (1111 = PUG) e o RE recebe também um pulso na sua linha de set-reset, sendo forçado o endereço 2.

Todas essas coisas equivalem à introdução de uma instrução de PUG para a posição 2 assim que chega um pedido de interrupção. Com essa instrução de PUG nas posições 2 e 3, é colocado um “pulo incondicional” (código 0000) para o CI do programa normal e o processamento é desviado para as posições 4 e 5, que têm um “pulo incondicional” para a rotina tratadora da interrupção. No final do atendimento da interrupção, existe uma instrução de PUL que equivale a um “pulo incondicional” para a posição 2.

Quando o sistema atende a um pedido de interrupção ele só atenderá ao seguinte depois de uma instrução após a de PUL, para não perder o endereço do programa normal. Isso provoca uma situação incomum, pois a instrução de PUL deverá agir, limpando a interrupção, apenas na instrução seguinte, que, no caso, é um PLA, que é uma instrução de uso geral. Por isso, é necessário o circuito visto na Fig. 5-31, que mostra como o PUL limpa o sinal “ocupado” no ciclo seguinte. Para atendê-lo, basta lembrarmos que o tempo T2, após um T4, só ocorre no ciclo seguinte.

5.8 MODOS DE OPERAÇÃO ESPECIAIS

Além do modo *normal* de operação, aquele onde o sistema fica operando até encontrar uma instrução de “pare” ou “espere”, existem outros modos, que chamamos de especiais, que são os seguintes:

grupo 1 “ciclo único”,
 “instrução única”;

grupo 2 “carrega endereço”,
 “carrega posição”,
 “mostra posição”.

Existem seis chaves no painel para selecionarmos um dos modos de operação, os quais são mutuamente exclusivos.

a. Grupo 1

“Ciclo único”

É um modo de funcionamento onde o sistema, a cada acionamento do botão de partida, evolui apenas um ciclo. Isso permite que, de ciclo em ciclo, isto é, de fase em fase, o operador do sistema acompanhe a evolução de seu programa. Seus objetivos são, principalmente, didáticos e de diagnose de falhas do sistema.

"Instrução única"

É um modo de funcionamento onde o sistema, a cada acionamento do botão de partida, executa uma instrução e pára na fase 1 da próxima. Serve para o operador acompanhar o seu programa de instrução a instrução. Seu objetivo é o *debug* de programas.

A implementação física desses modos de funcionamento foi feita controlando os sinais de fase. Existe um sinal elétrico chamado "roda", que controla as fases. No modo normal de funcionamento, esse sinal fica permanentemente no nível 1, permitindo a franca evolução do programa. No modo especial "ciclo único", a cada pressionamento do interruptor de partida, o sinal "roda" eleva-se ao nível 1 até o fim de T_7 , quando volta a zero e, assim, a cada partida que se dá, teremos a liberação do sinal de fase por um ciclo, fazendo com que exista o processamento normal por uma fase apenas. No outro modo especial de funcionamento ("instrução única"), o comportamento do sistema é análogo com a única diferença de que o sistema pára de operar a cada vez que vai entrar na fase 1. Portanto o sistema, a cada acionamento do botão de partida, executa *uma* instrução completa e pára. O leitor encontra na Fig. 5-36 um esquema lógico do circuito de controle do modo de operação.

b. Grupo 2

Os modos especiais de funcionamento desse grupo têm a característica particular de fazer o sistema funcionar *sem fases*. O sinal do interruptor de partida tem a forma vista na Fig. 5-20.

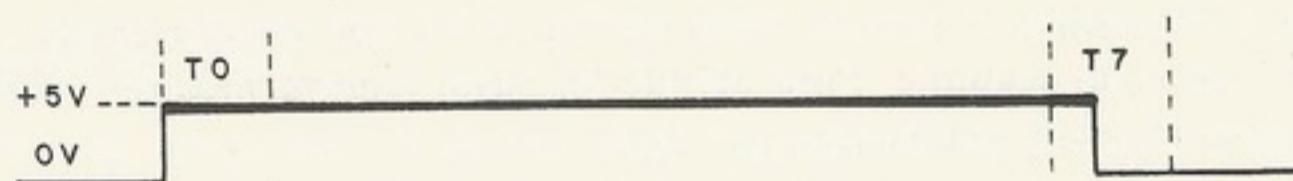


Figura 5-32. Sinal de partida

O sistema, com o modo especial de funcionamento desse grupo opera usando o sinal de partida ao invés do de fase, tendo uma seqüência de microoperações específicas.

Carrega endereço

Com o acionamento do interruptor de partida, o conteúdo do registrador de chaves do painel é transferido para o registrador de endereço da memória e o contador de instrução (Fig. 5-33).

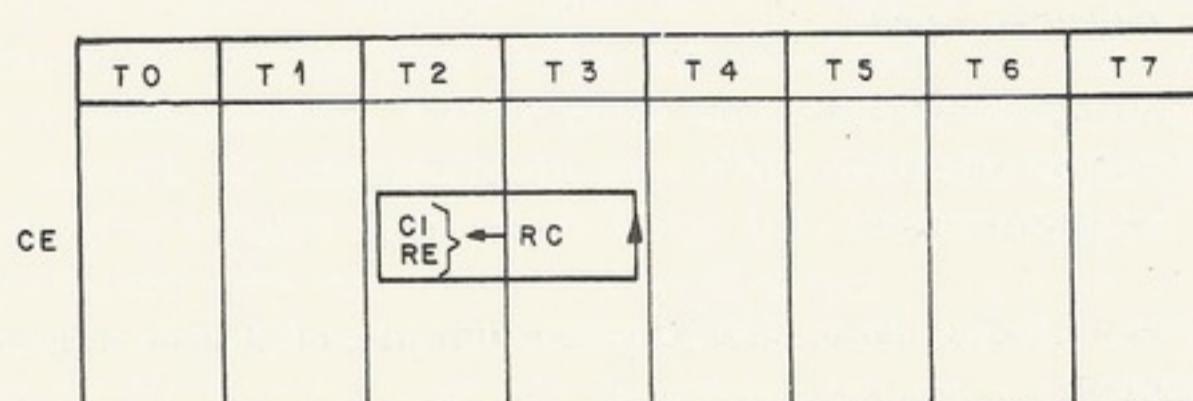
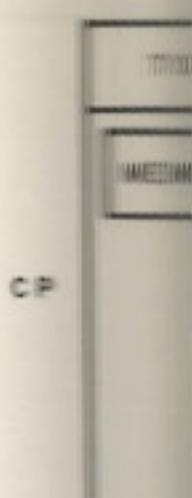


Figura 5-33. Distribuição das microoperações no modo "carrega endereço"

Carrega posição

Ao se acionar o interruptor de partida, o conteúdo dos 8 bits menos significativos do registrador de chaves do painel é armazenado na posição de memória cujo endereço está no *RE* (Fig. 5-34).



Mostra p

Com o acionamento do interruptor de partida, o conteúdo do registrador de chaves do painel é transferido para o registrador de endereço da memória e o contador de instrução (Fig. 5-33).

Como você está na inicialização do sistema, o programa (carga) é carregado para a memória, que permite a execução.

5.9 SUMÁRIO

Após termos estudado um sistema digital, onde procuramos entendermos e compreendermos a estrutura de um sistema digital. Neste ponto, podemos e vamos ver como o sistema digital é construído. Dissemos e, sobretudo, que é a arquitetura.

A partir de "arquitetura",

do botão de partida para o operador acompanhar os programas.

enviando os sinais de comando. No modo normal de funcionamento, a fraca evolução do interruptor de partida é para zero e, assim, a unidade fazendo com que especial de funcionamento a única diferença entre o sistema, a unidade e pára. O leitor muda de operação.

mística particular de em a forma vista na

usando o sinal específico.

operador de chaves operador de instrução

significativos do cujo endereço está

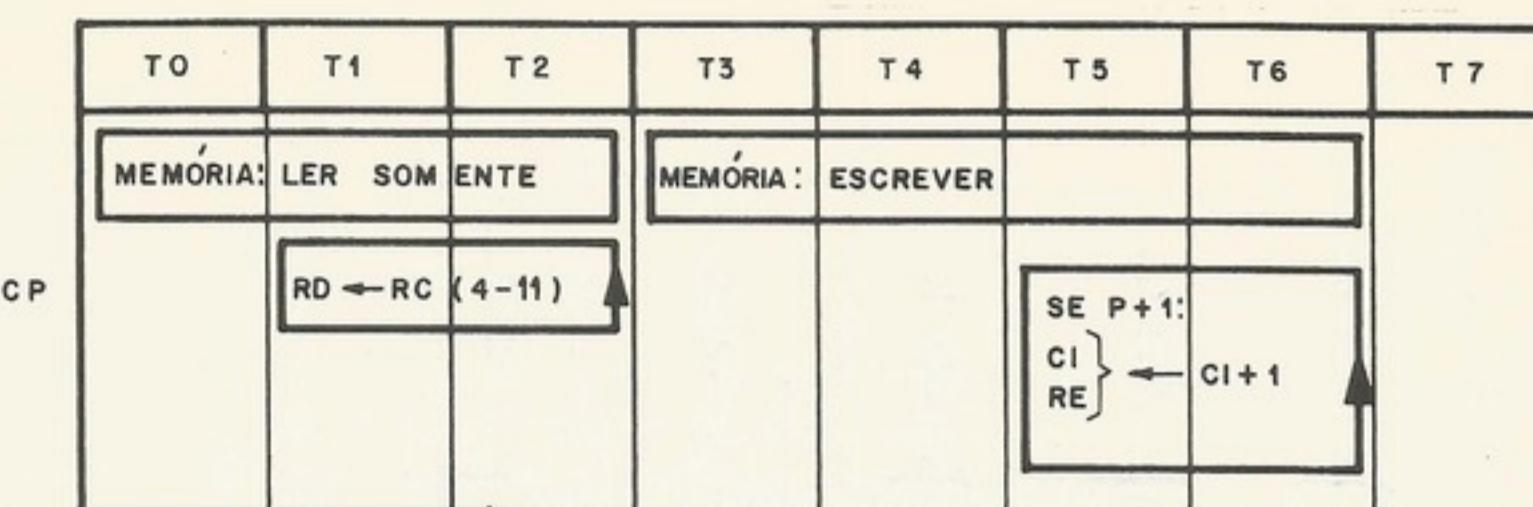


Figura 5-34. Seqüência das microoperações no modo "carrega posição"

Mostra posição

Com o acionamento do interruptor de partida, o conteúdo da posição de memória cujo endereço está no RE é transferido para o RD, permitindo ao operador a visualização do conteúdo dessa posição da memória (Fig. 5-35).

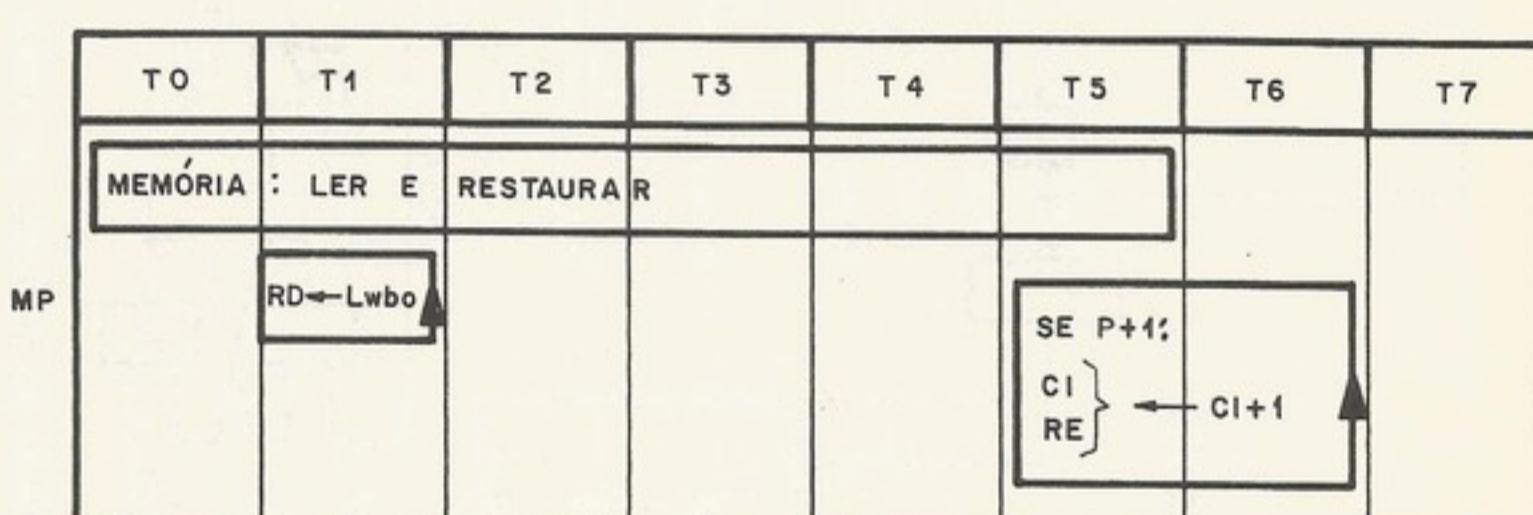


Figura 5-35. Seqüência das microoperações no modo "mostra posição"

Como você já deve ter notado, a importância dos modos de funcionamento do grupo 2 está na inicialização do sistema e *debug* de programas. Se quisermos carregar o primeiro programa (carga fria), deveremos fazê-lo pelas chaves do painel. Ao ligar o sistema, deveremos endereçar o "programa carregador" pelas chaves. É possível a análise de um programa na memória, ou mesmo sua modificação, pelo próprio painel. Enfim, é esse grupo que permite a interação do operador no sistema, através das chaves.

5.9 SUMÁRIO

Após termos estudado, nos capítulos anteriores, os elementos e conceitos que formam um sistema digital, vimos, neste capítulo, uma breve descrição de um minicomputador onde procuramos mostrar, em síntese, um exemplo da reunião daqueles conceitos.

Neste ponto, acreditamos que você já tenha adquirido uma visão detalhada dos elementos e uma visão geral do conjunto, entendendo como funciona e como se projeta um sistema digital. Evidentemente, muitas dúvidas existem, pois, sobre muitos detalhes, nada dissemos e, sobre muitos conceitos, apenas demos uma idéia vaga.

A partir de agora, nos próximos capítulos, veremos, com mais detalhes, os conceitos de "arquitetura" e "entrada/saída".

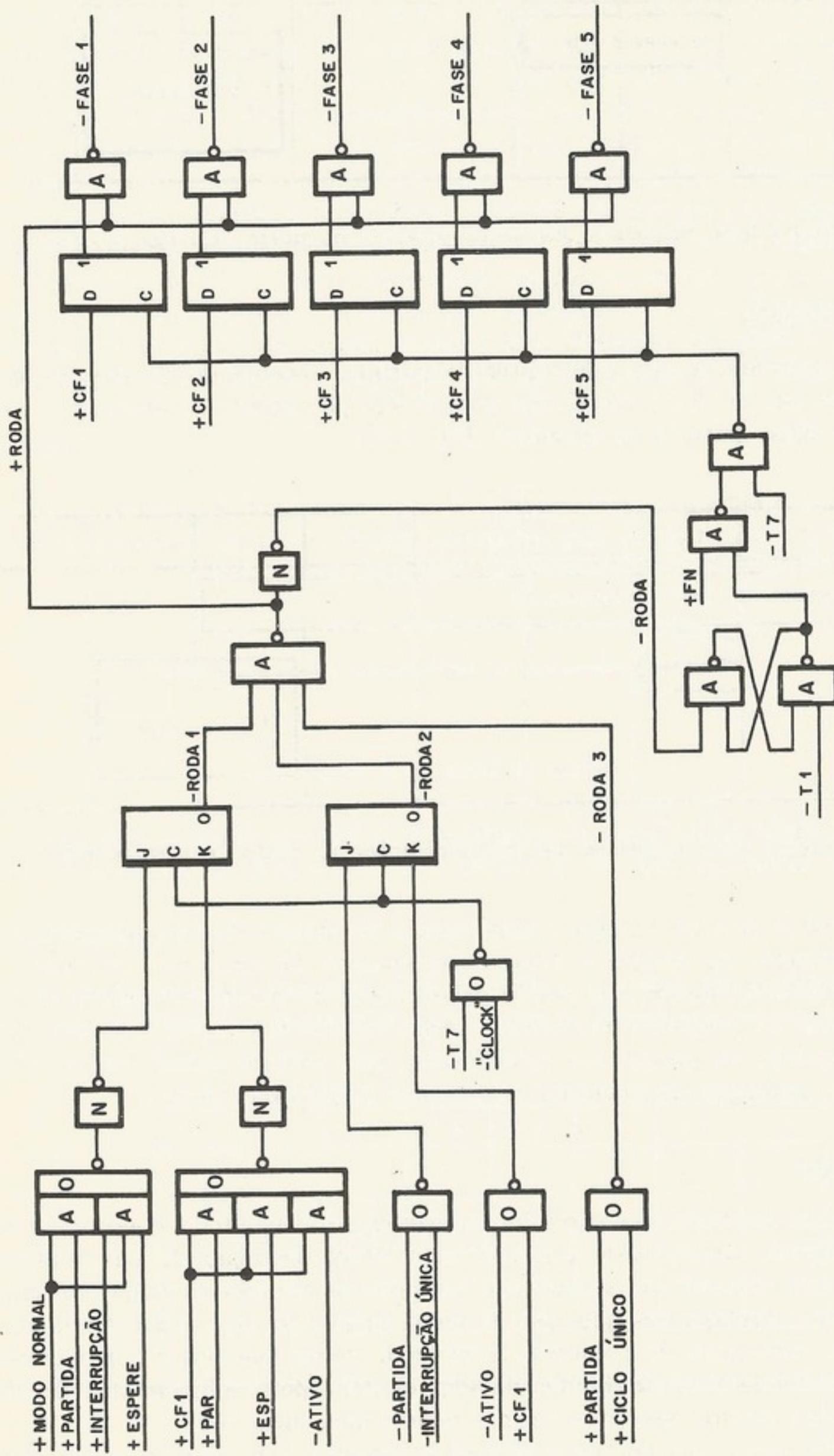


Figura 5-36. Esquema simplificado do controle do modo de operação

5.10 PRANCHAS

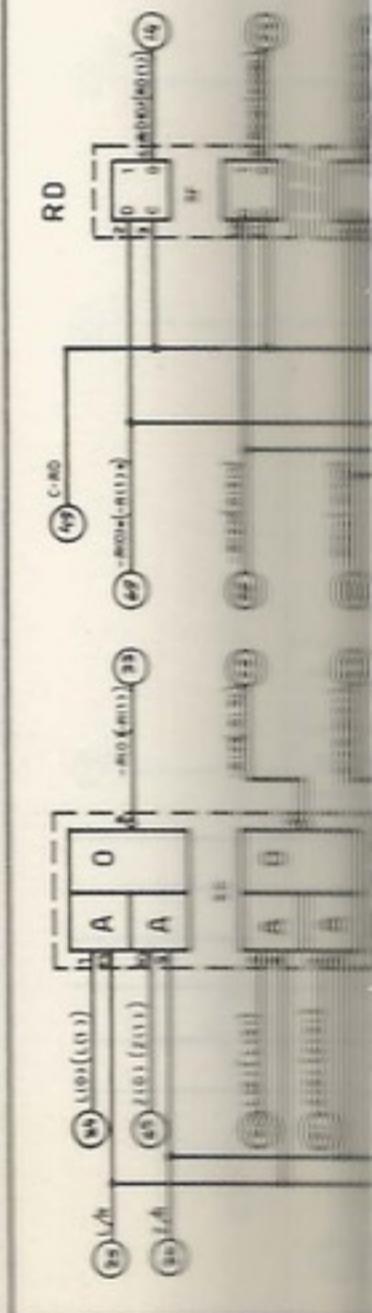
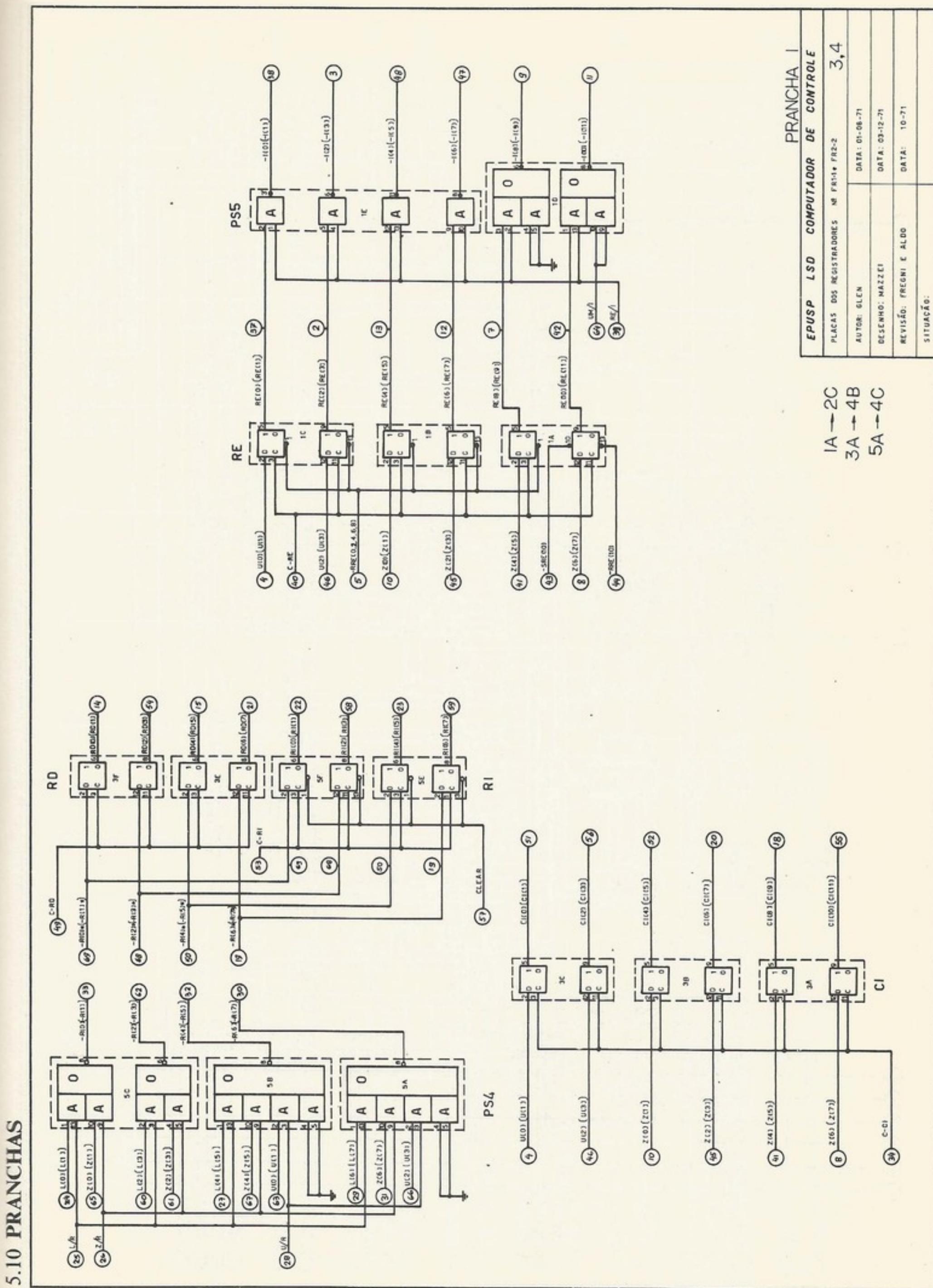
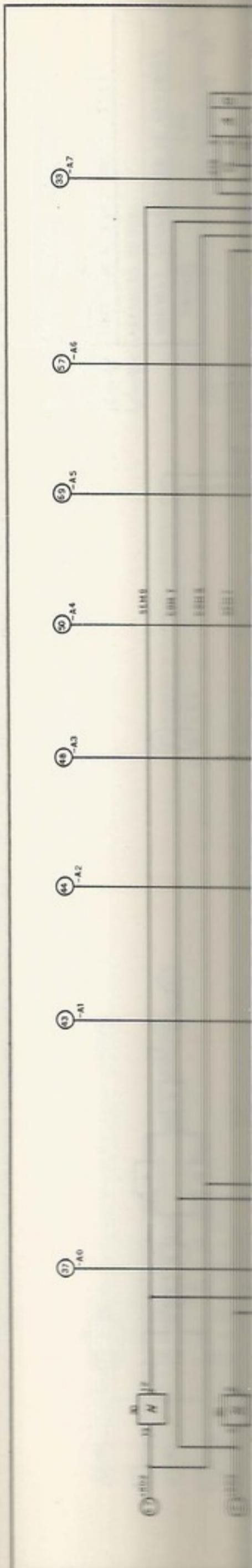
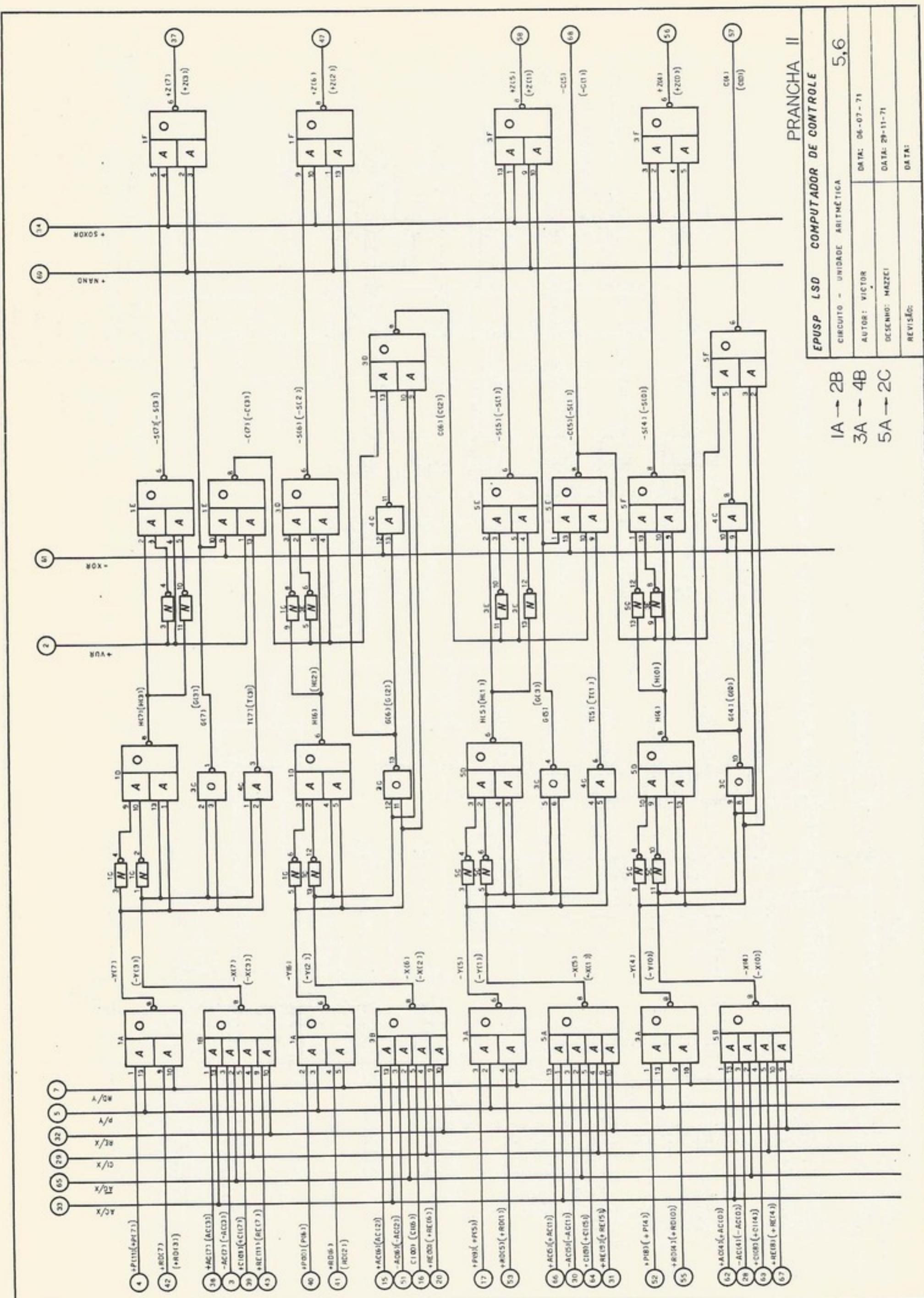
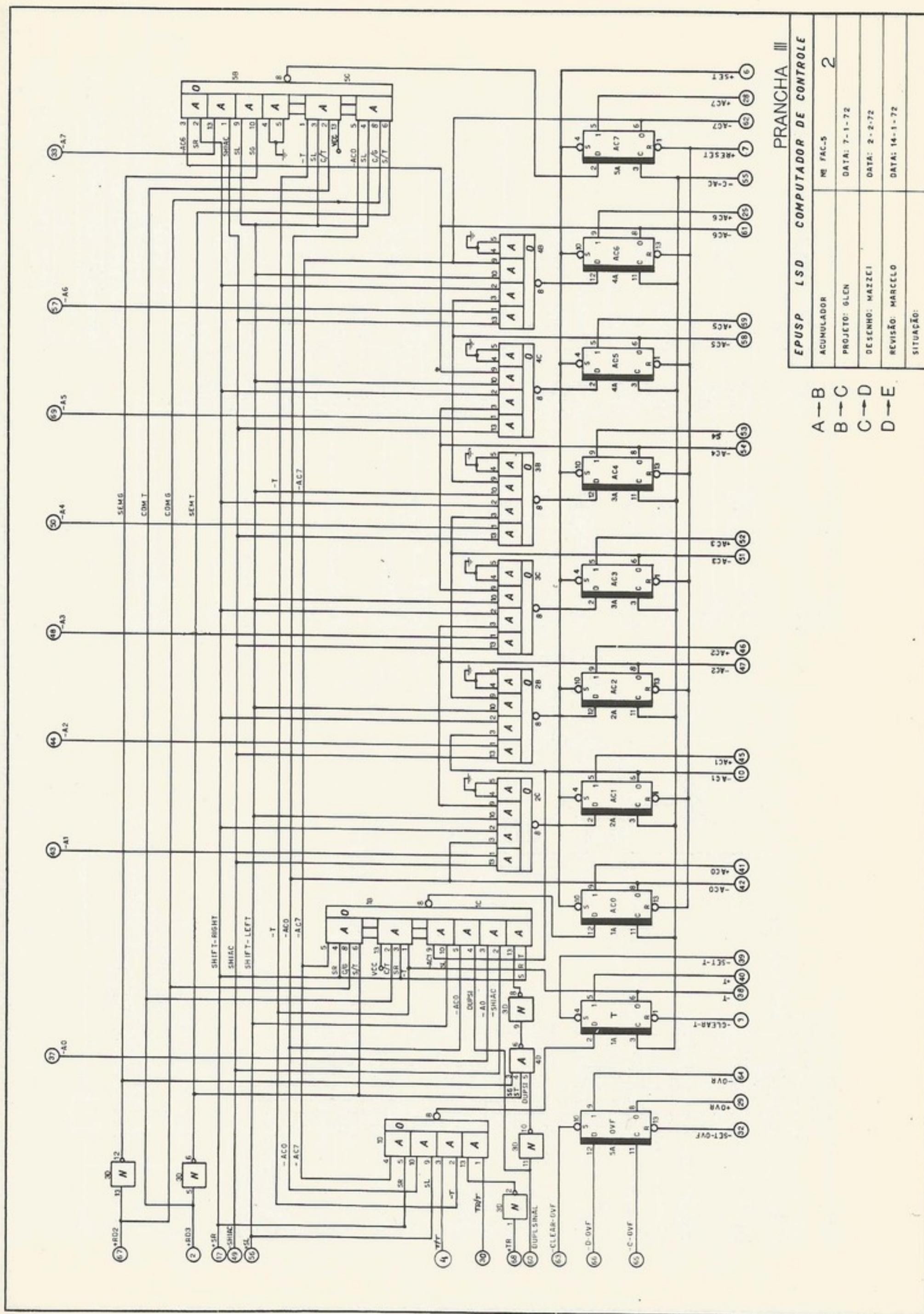


Figura 5-36. Esquema simplificado do controle do modo de operação





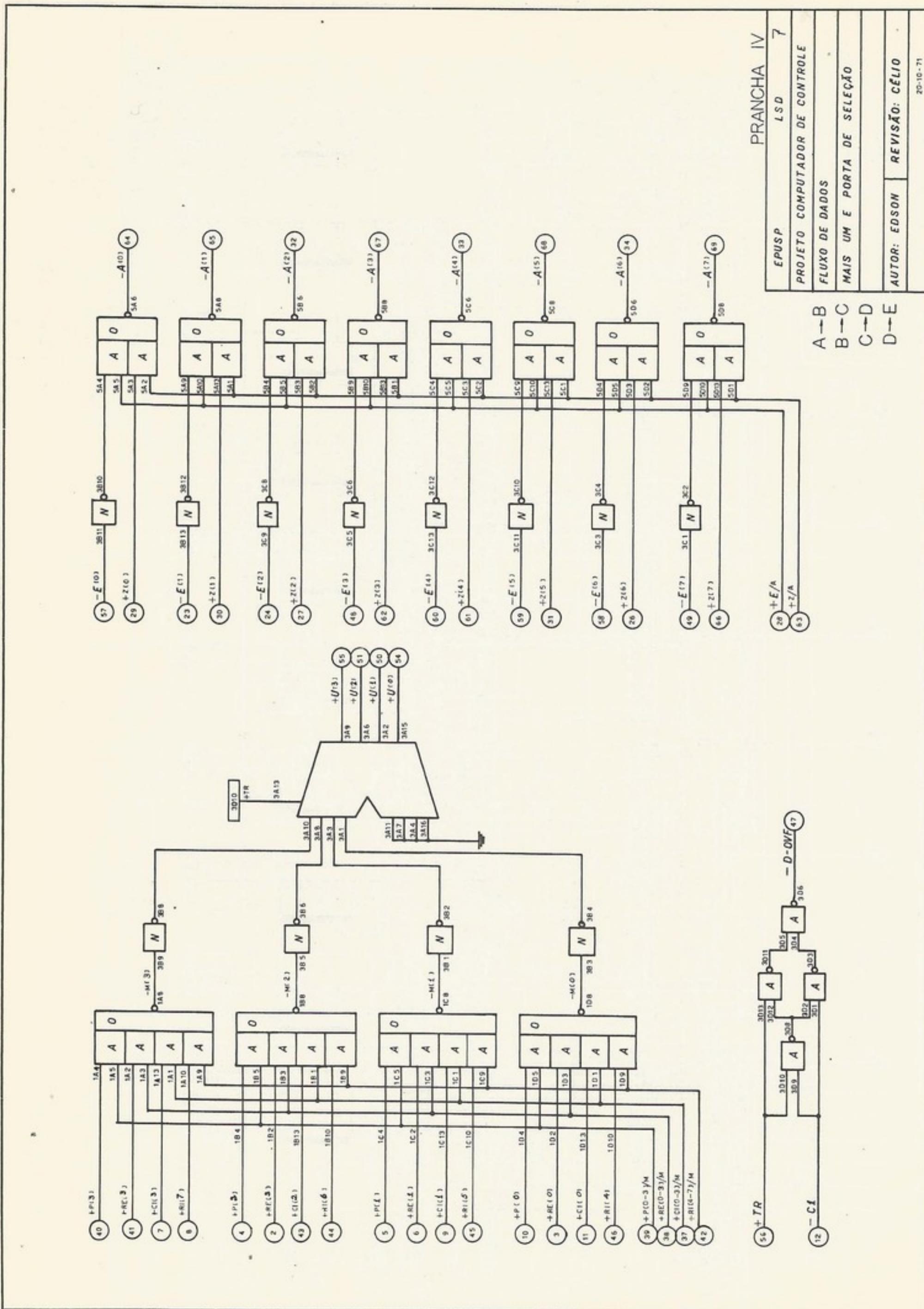
PRANCHAS	
IA → 2B	EPUSEP LSD COMPUTADOR DE CONTROLE
3A → 4B	UNIDADE ARitmética
5A → 2C	AUTEN: VICTOR DATA: 06-07-71
	DESENHO: MAZZEI DATA: 29-11-71
	REVISÃO: DATA:



PRANCHAS III

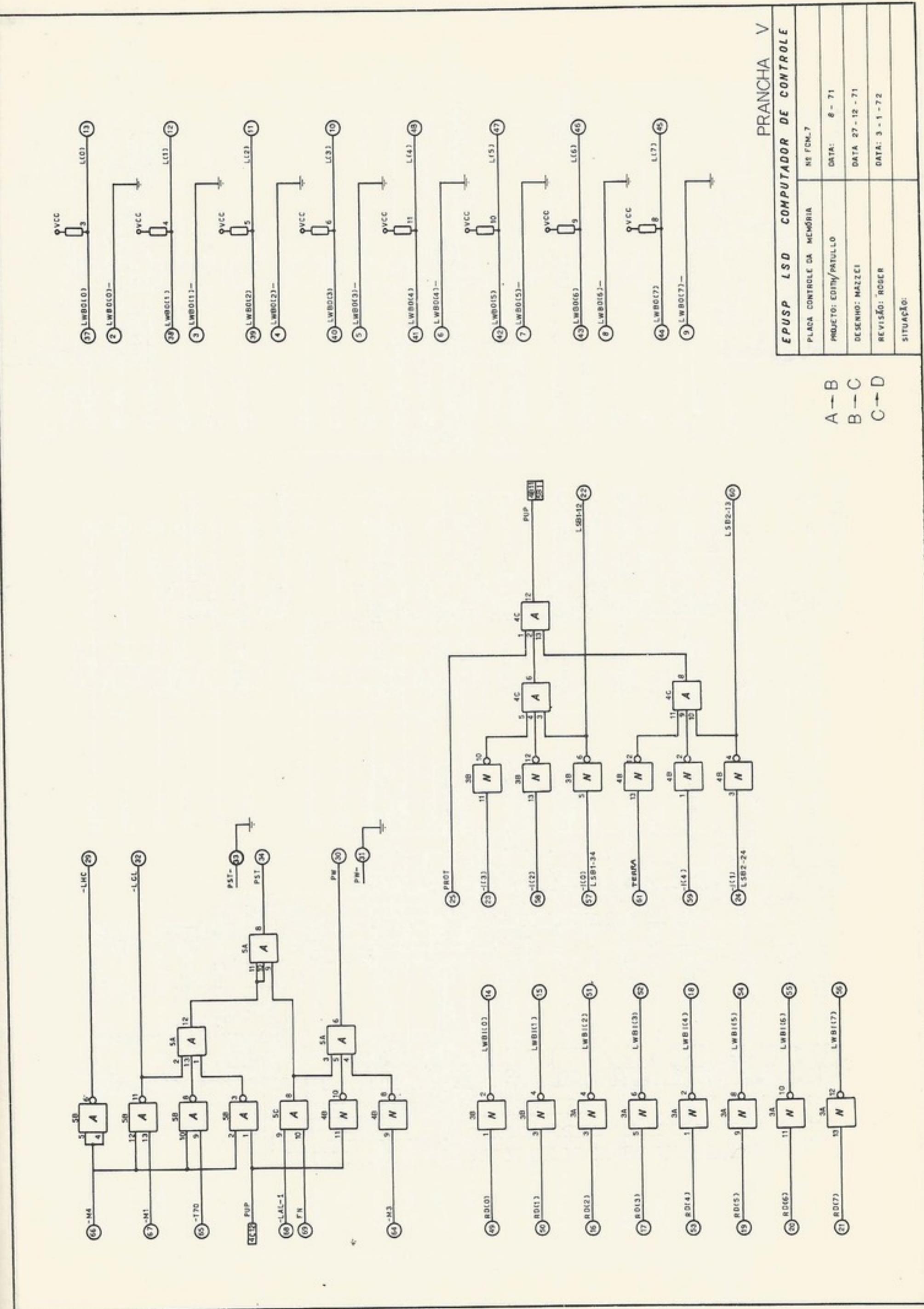
EPUSEP LSD COMPUTADOR DE CONTROLE	
A → B	ACUMULADOR
B → C	PROJETO: GLEN NO: FAC-5
C → D	DATA: 7-1-72
D → E	DESENHO: MAZZEI DATA: 2-2-72
	REVISÃO: MARCELO DATA: 14-1-72
	SITUAÇÃO:

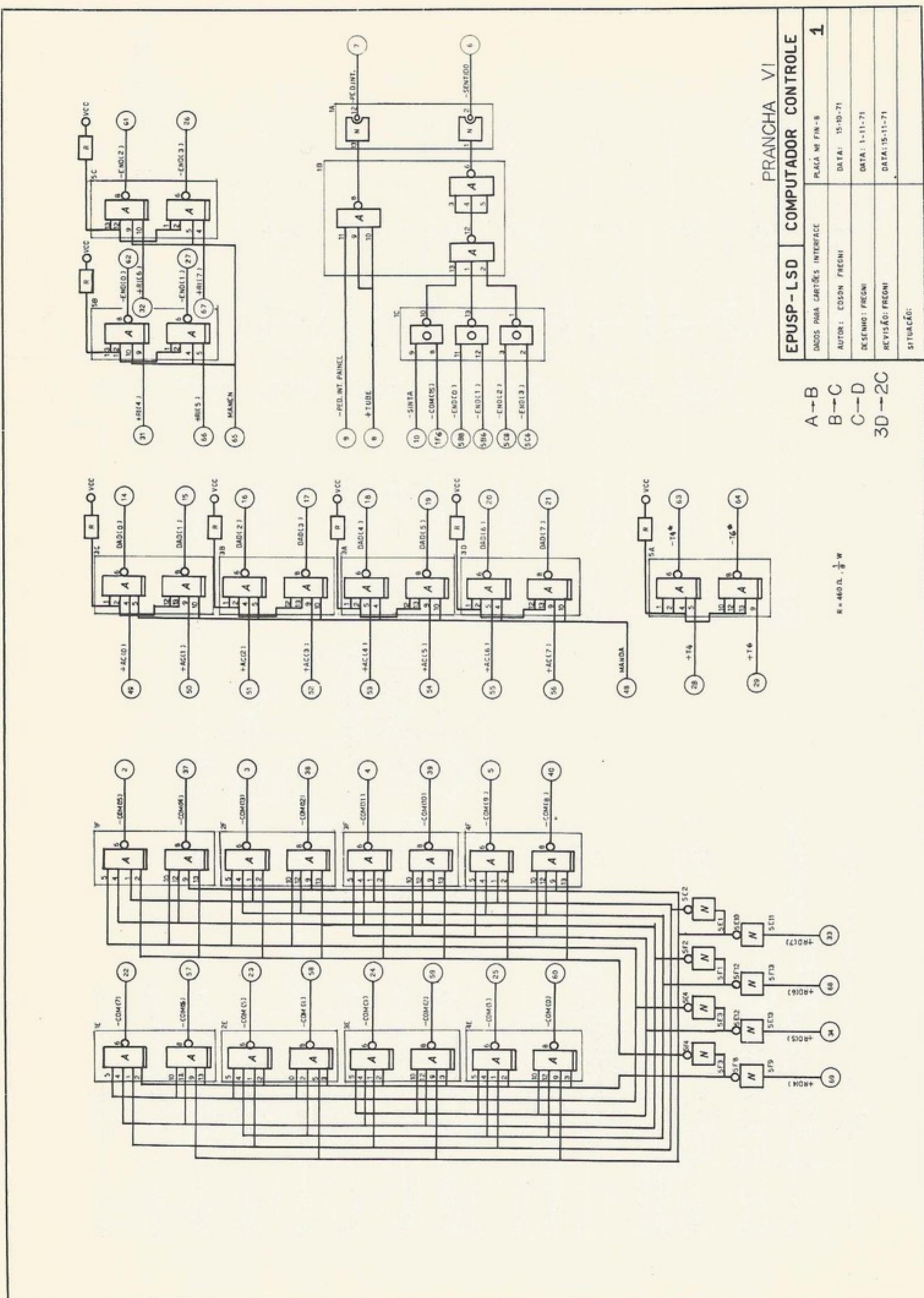
exemplo de

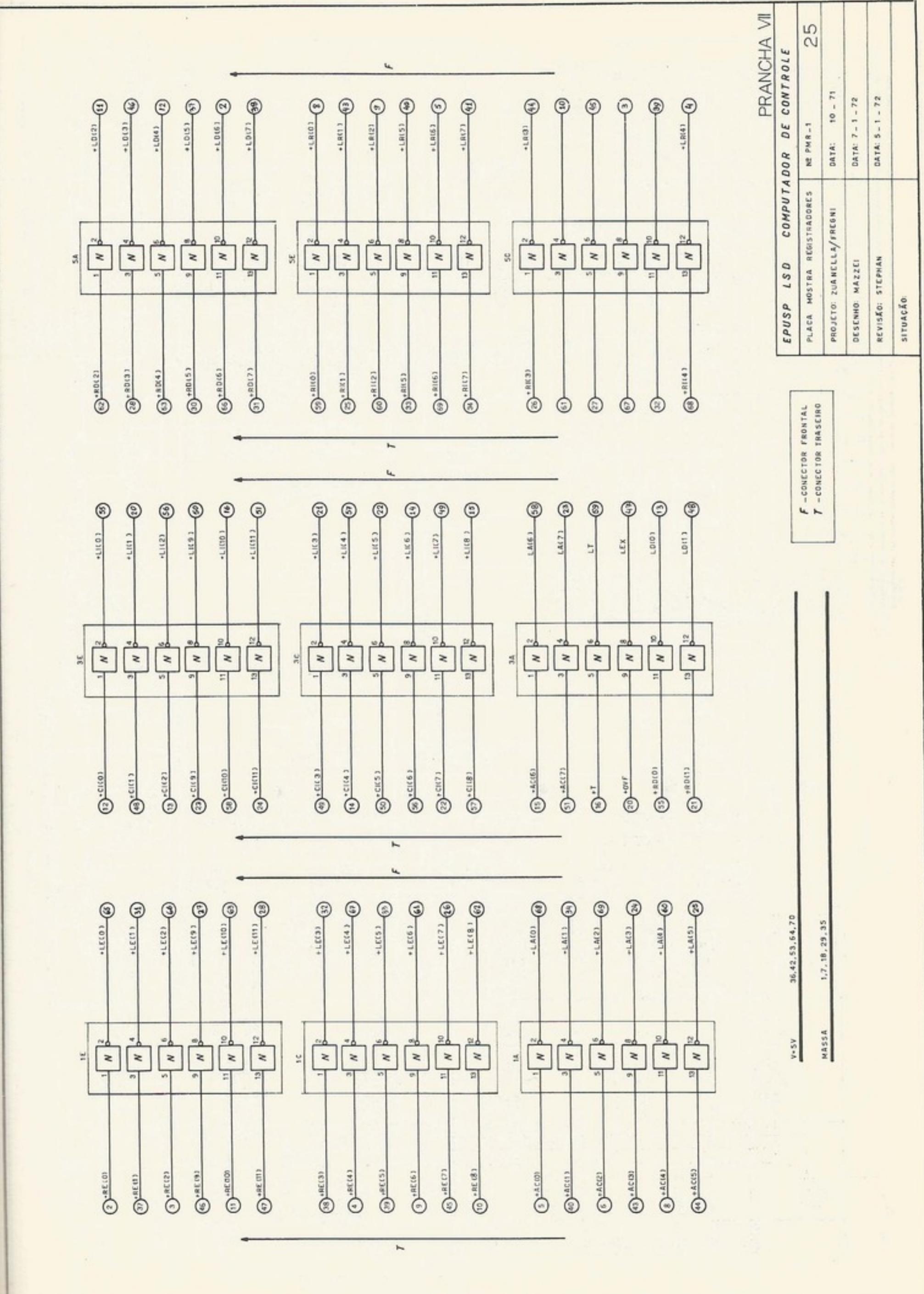
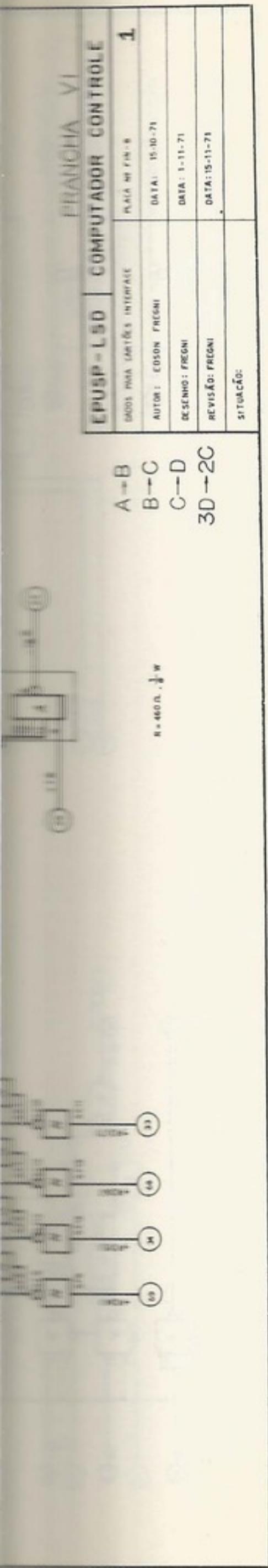


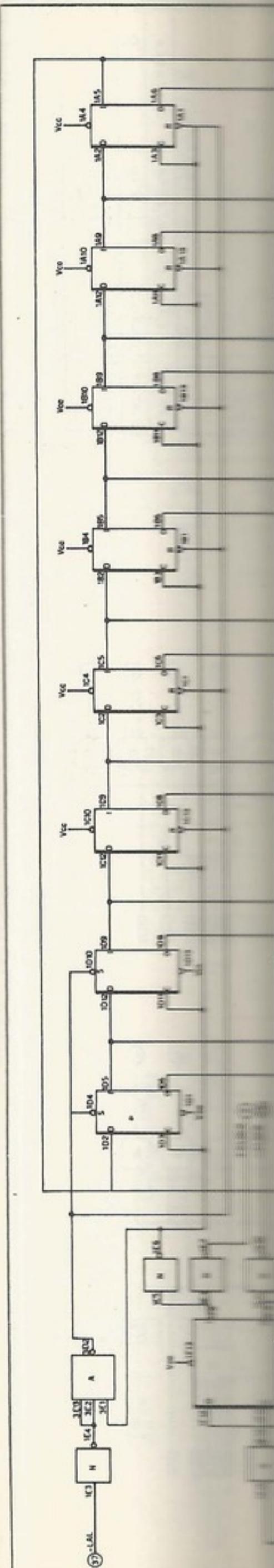
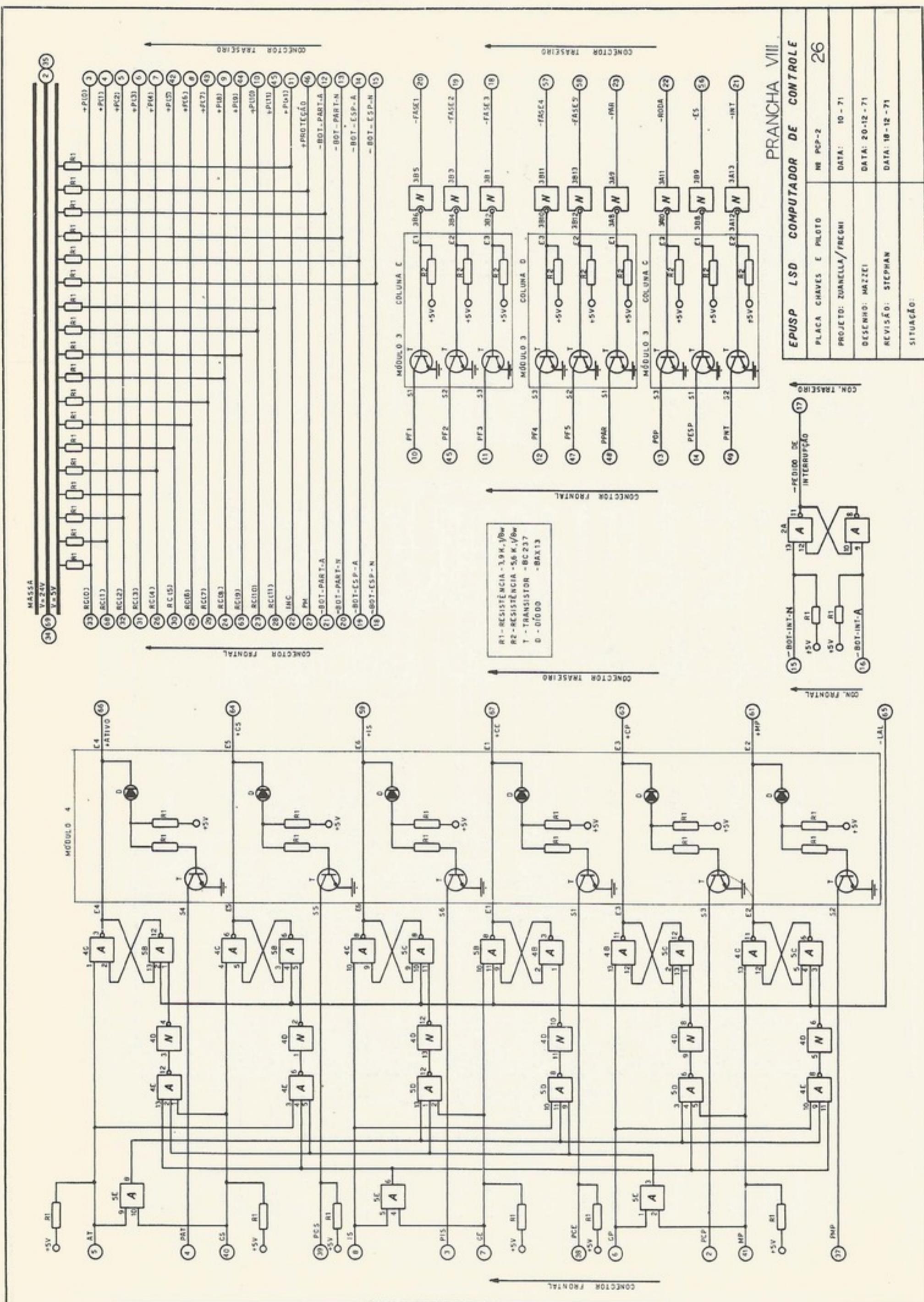
exemplo de um minicomputador

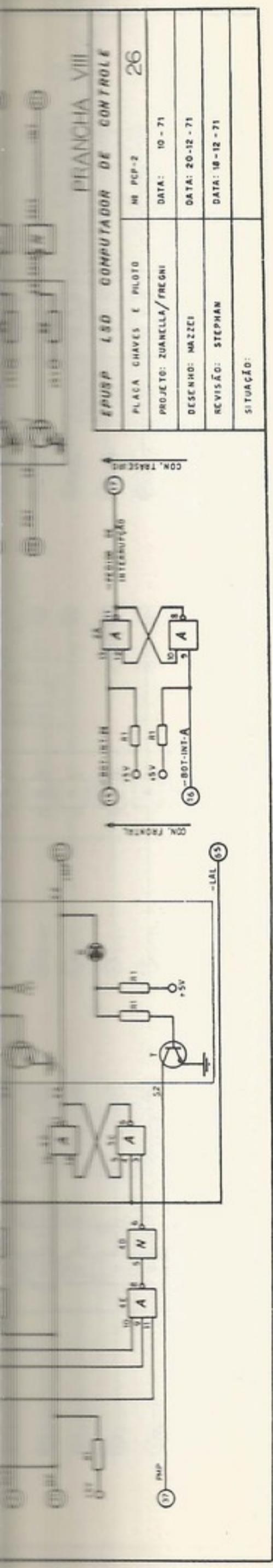
213



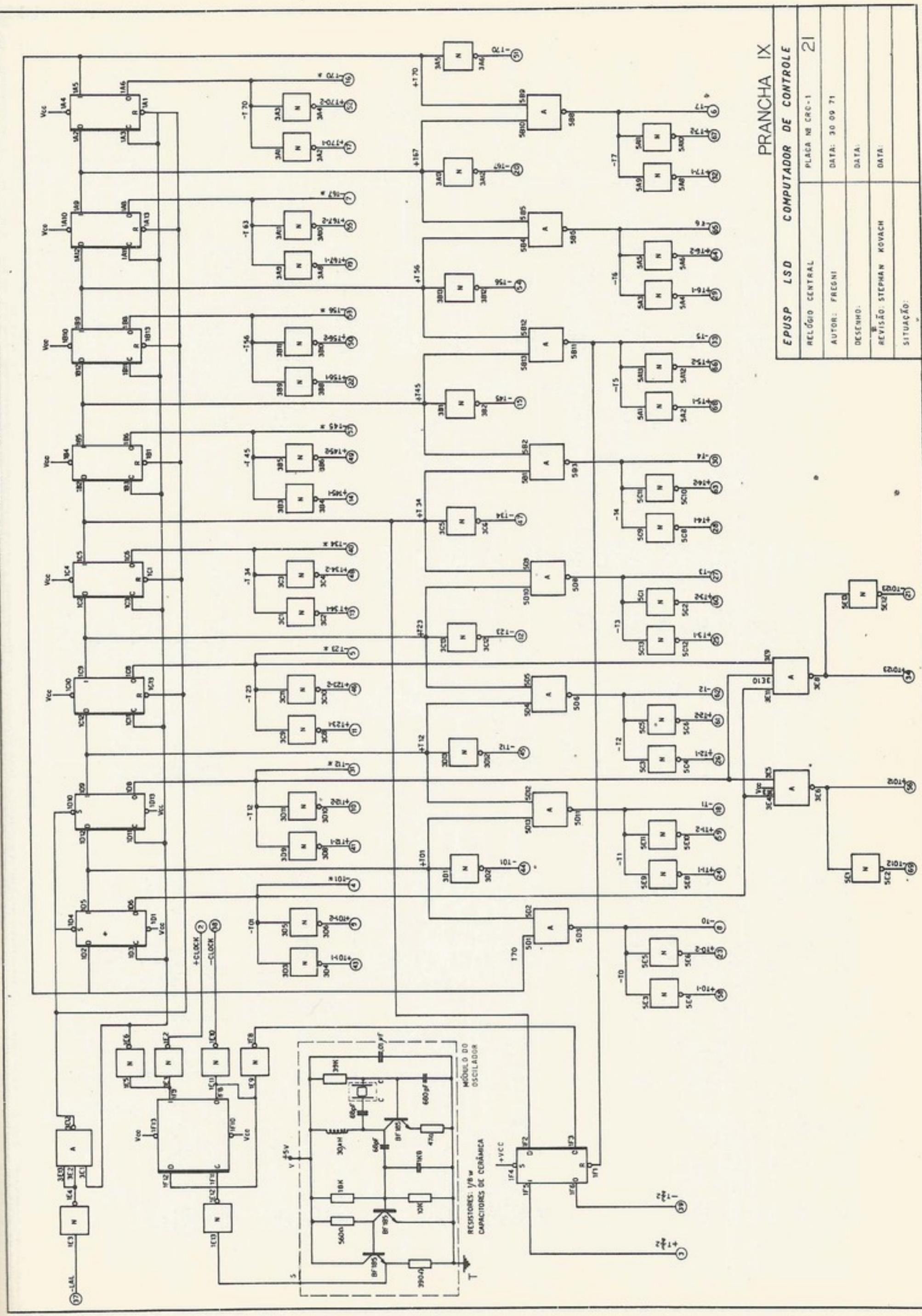


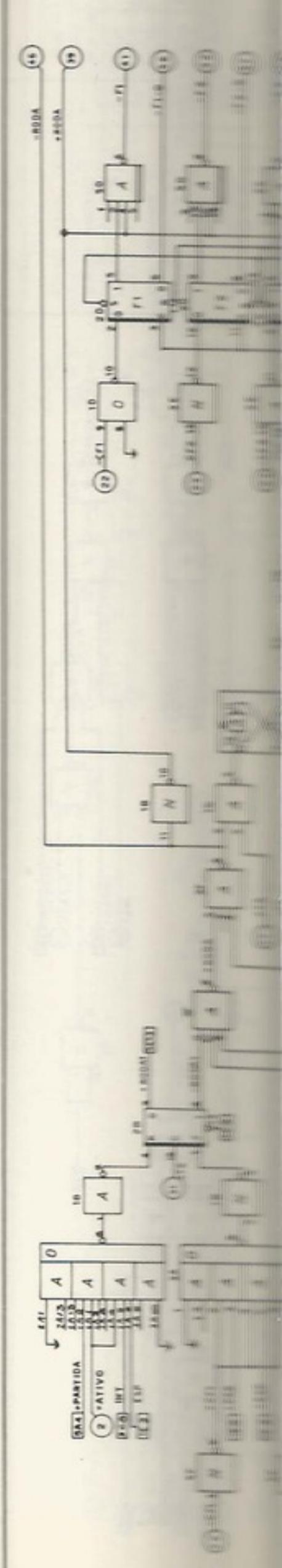
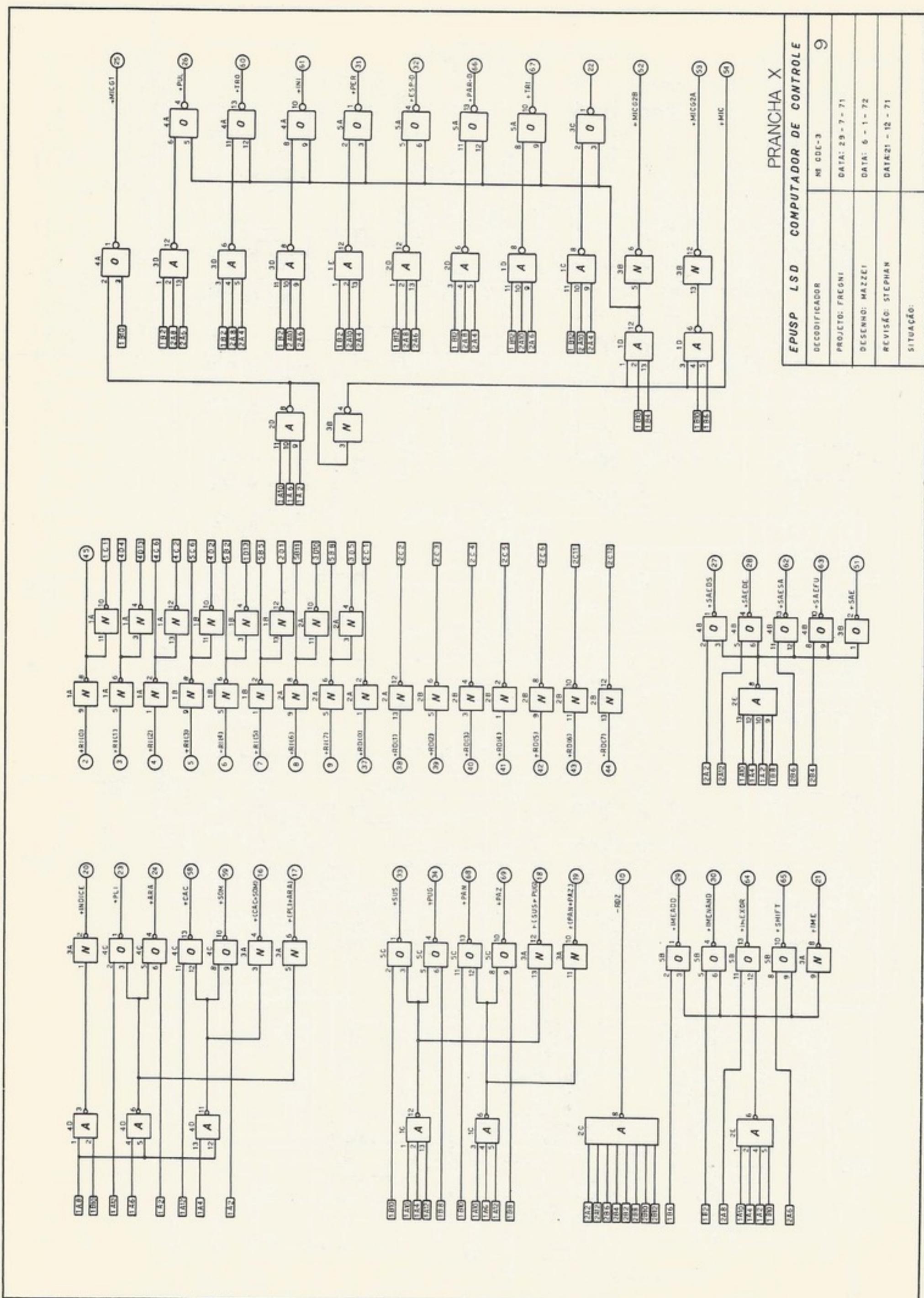


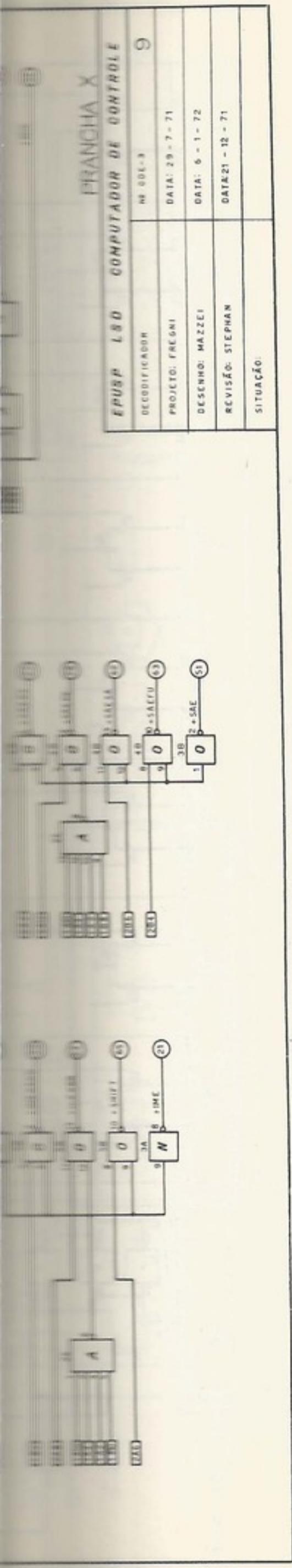




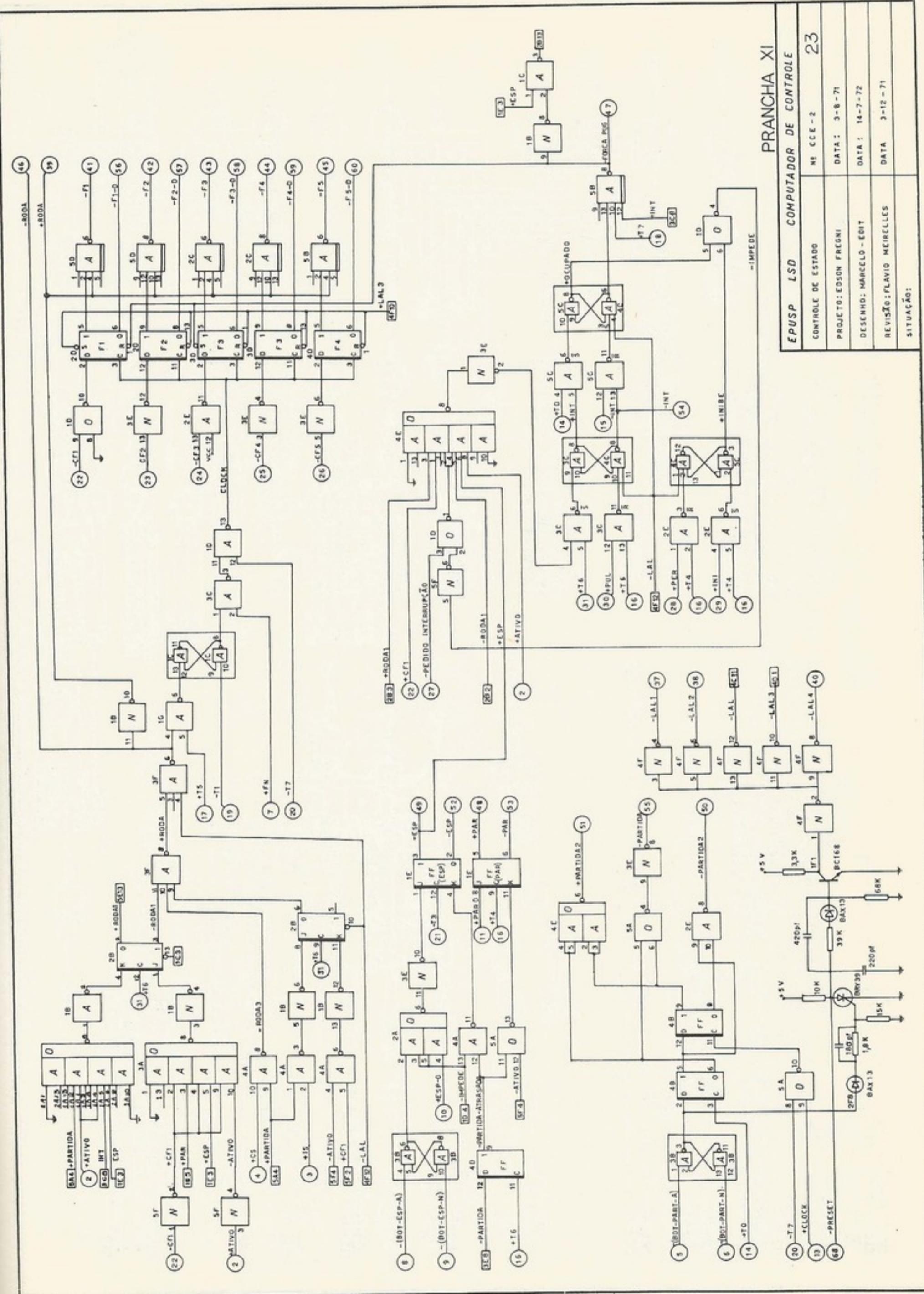
exemplo de um minicomputador

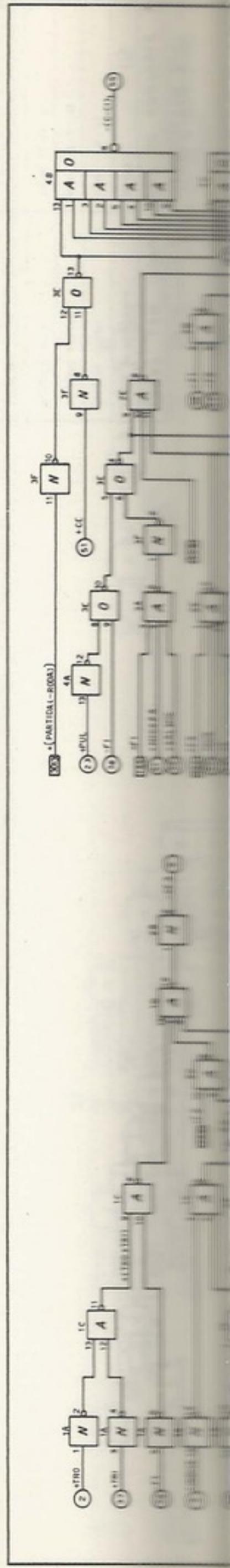
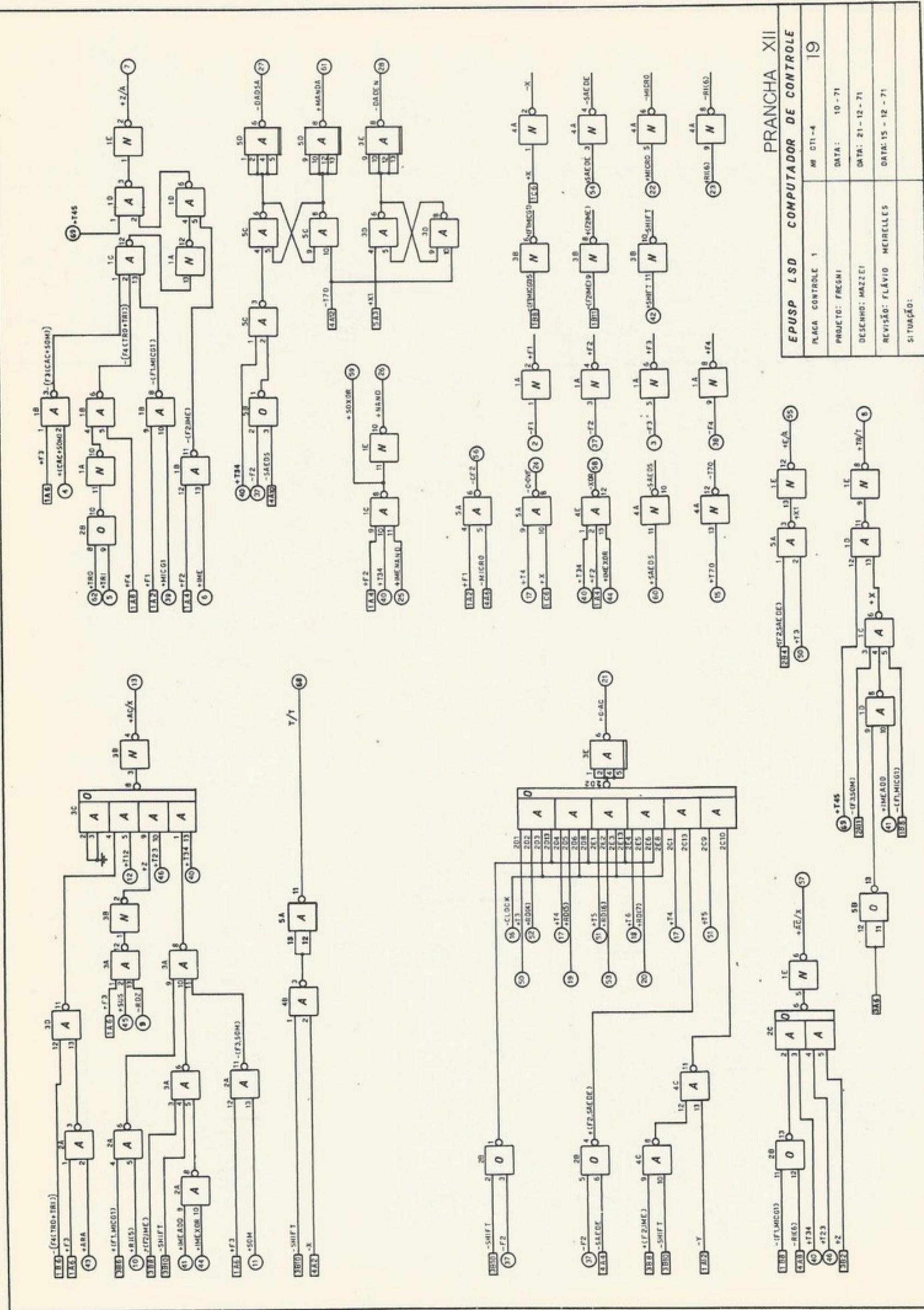


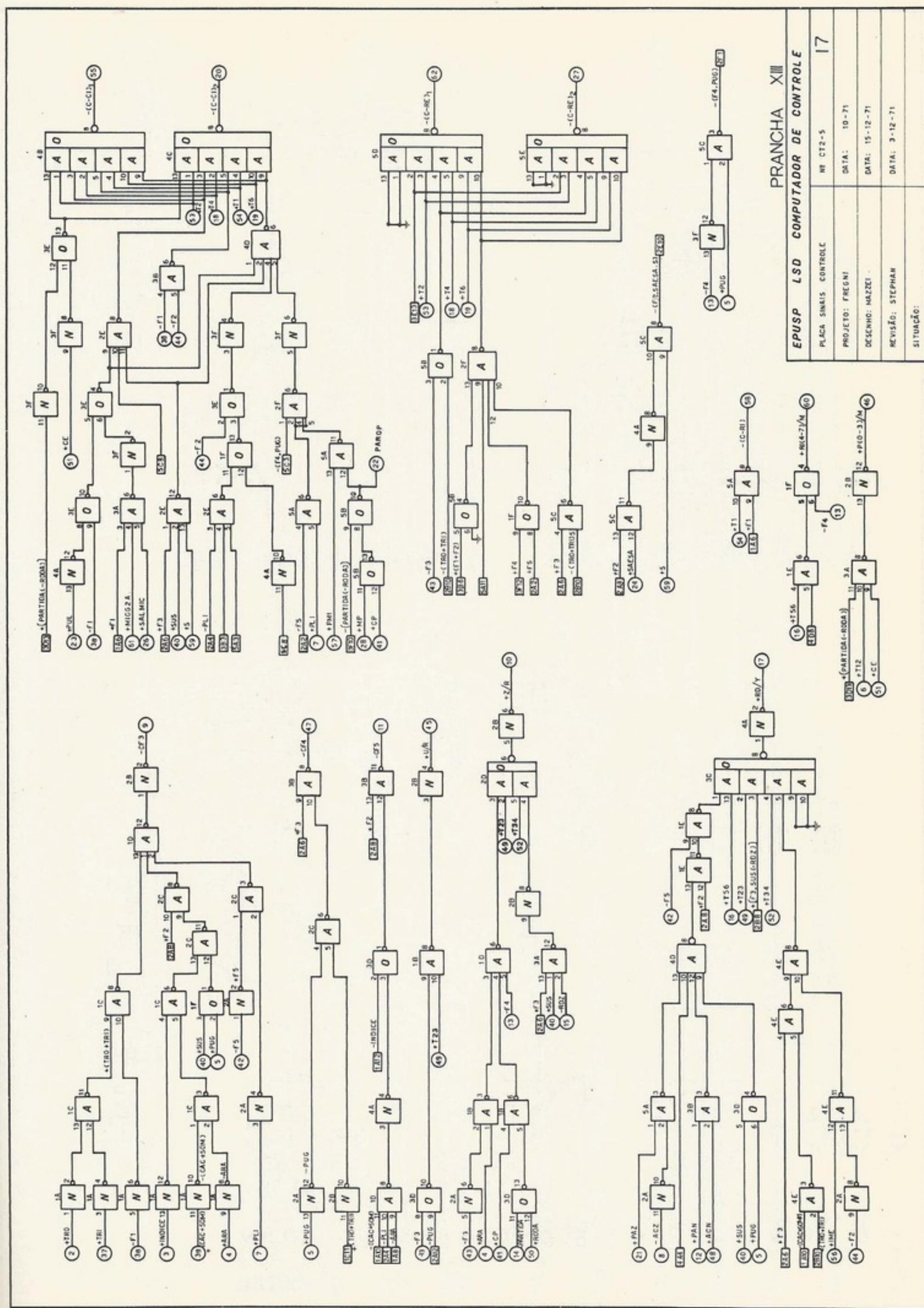
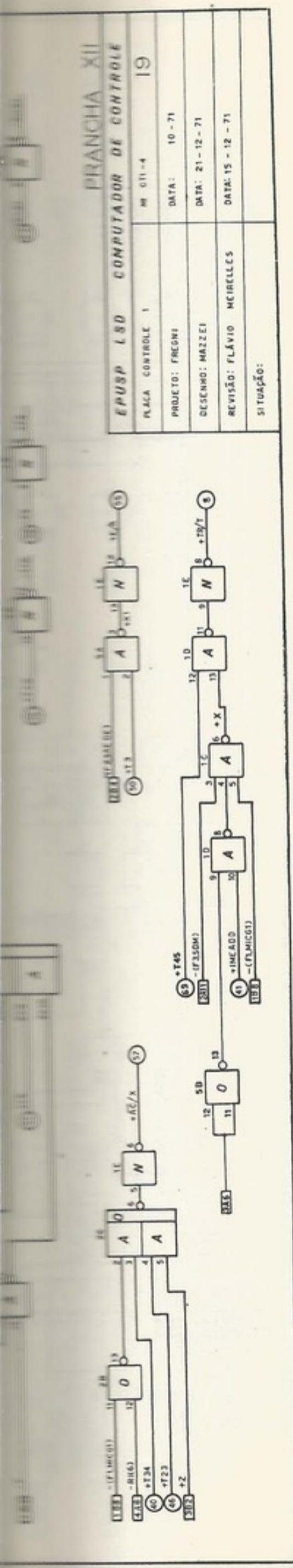


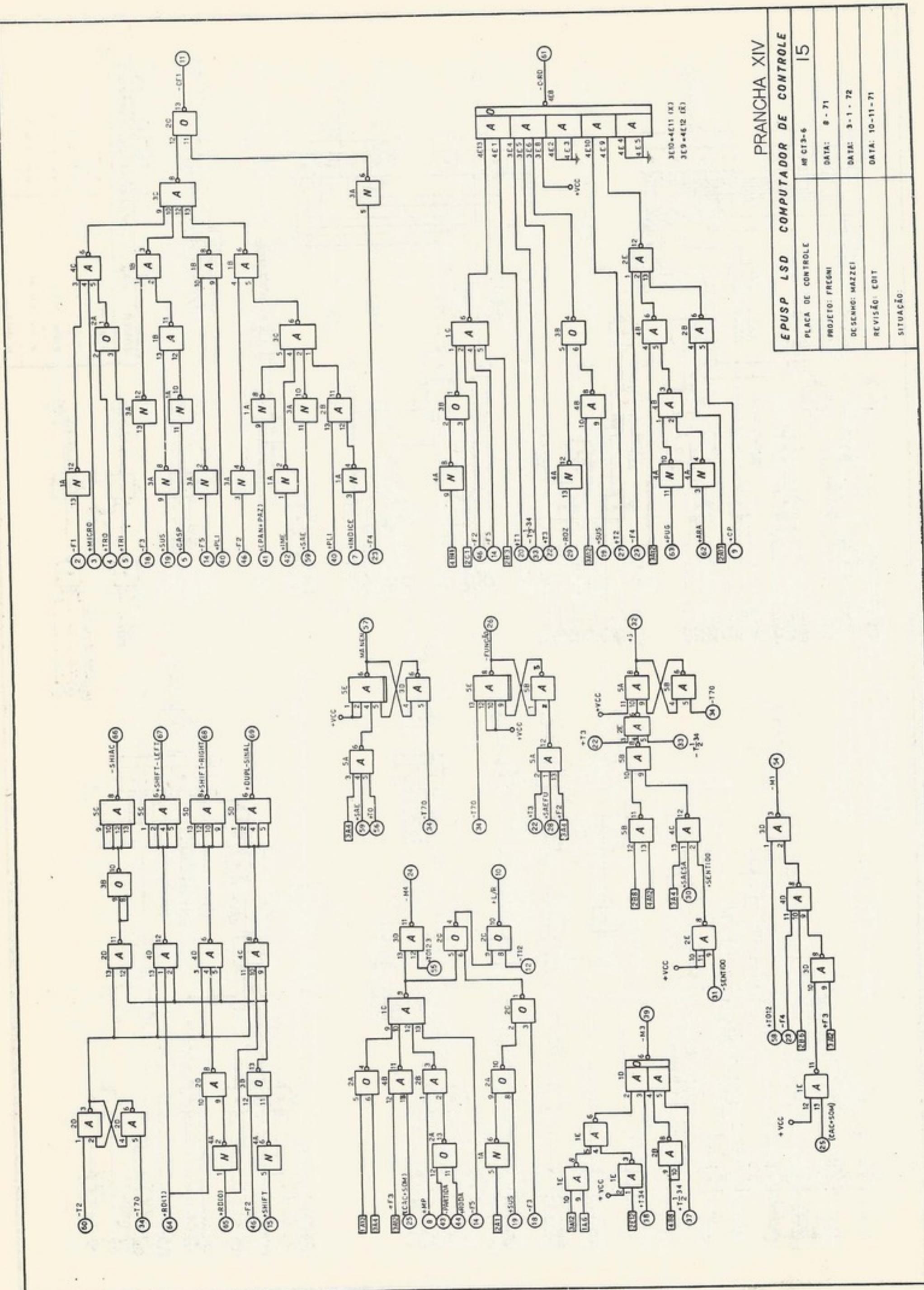


exemplo de um minicomputador

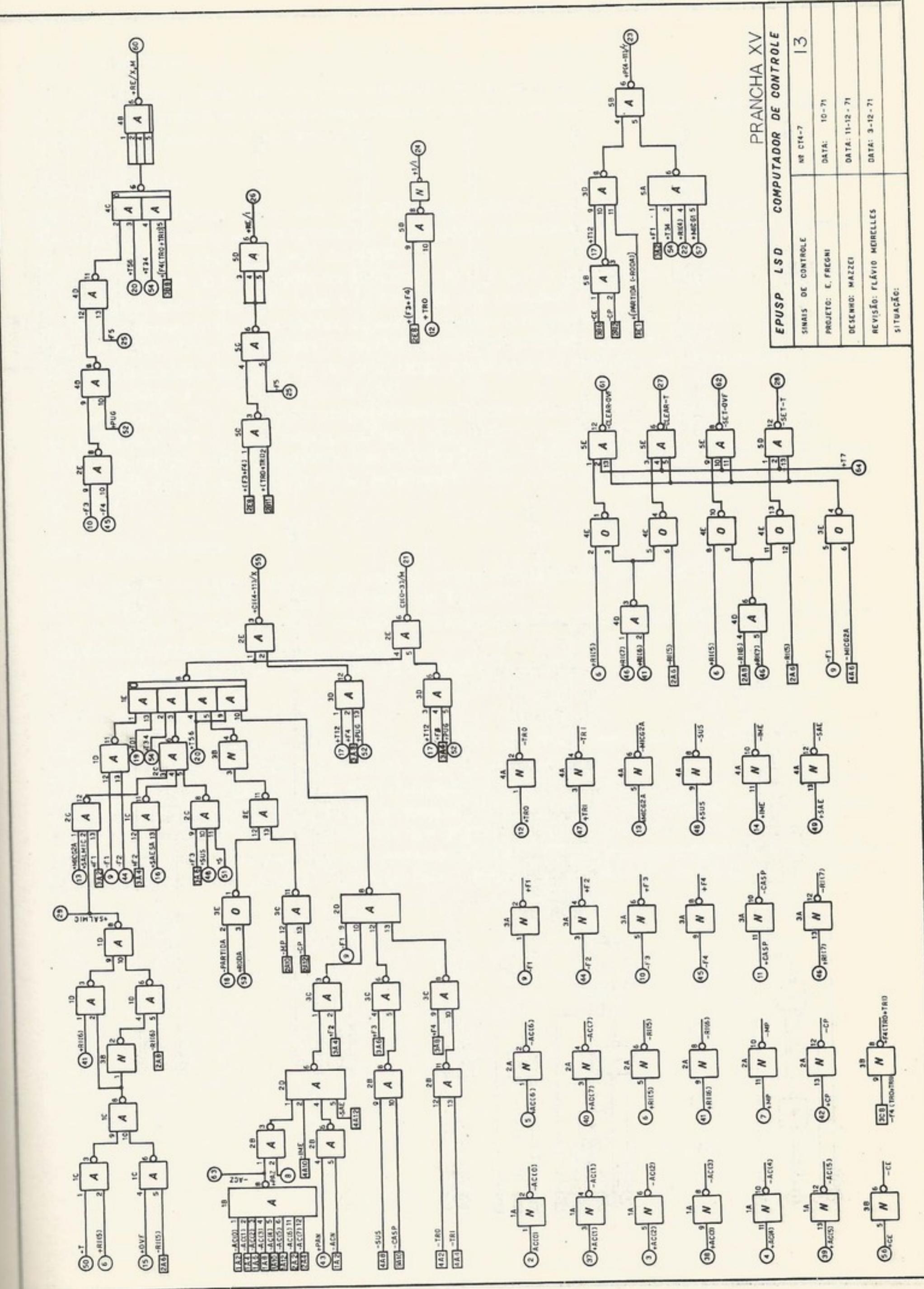


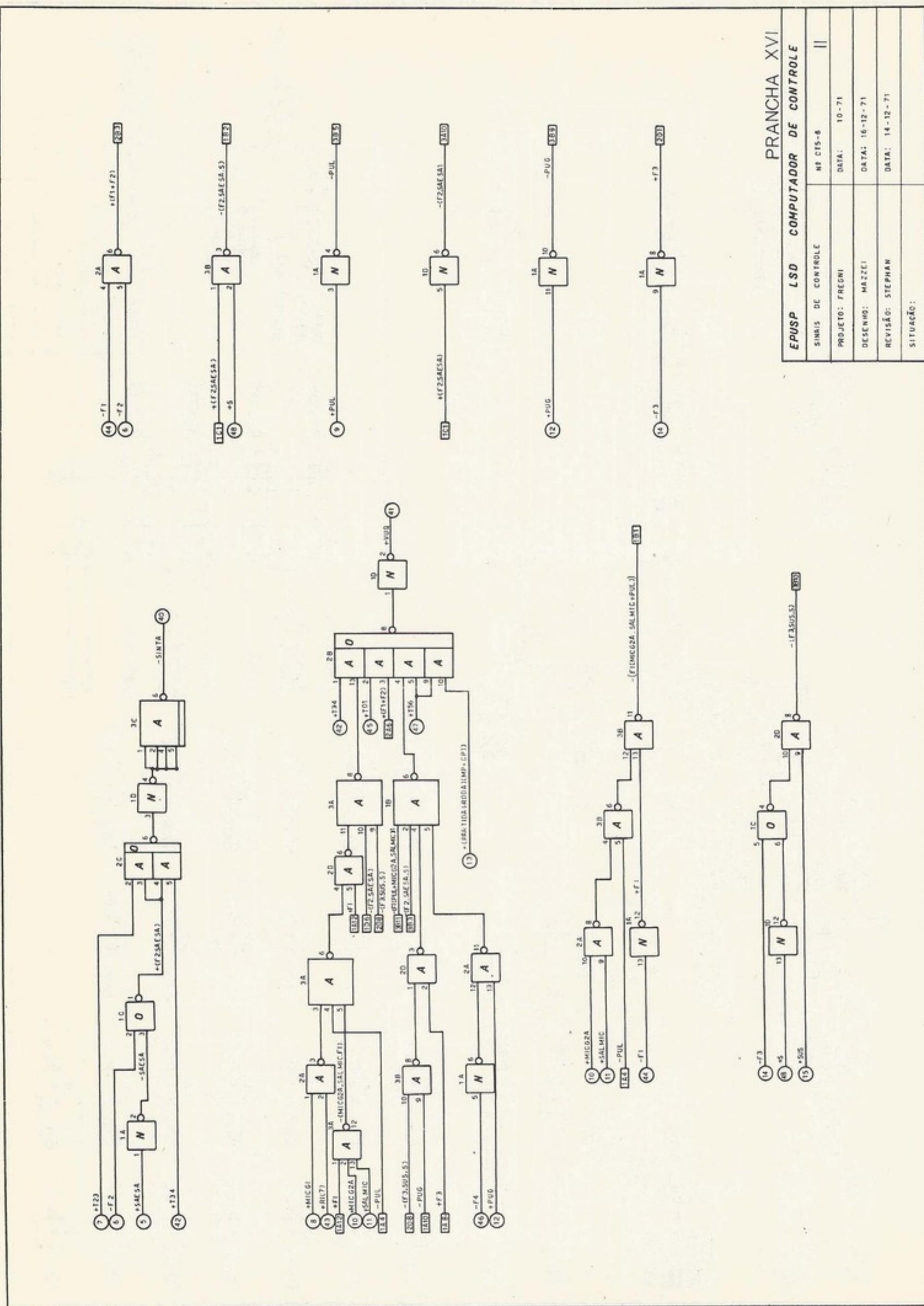






exemplo de um minicomputador





exemplos de ...

EXERCÍCIOSSobre a arqu...
5-1. Projete ...

5-2. O que se ...

Quais as mu...

Sobre as impre...

5-3. O que se ...

5-4. O que se ...

de PUG para ...

5-5. Se o com...
da instrução ...5-6. Qual o co...
seria 100000000...5-7. Qual a o...
de que o com...
5-8. Compre...
5-9. E como ...
5-10. Faga u...

A e B, semelh...

105 e 116.

Sobre o filtra...

5-11. Faga u...

5-12. Projet...

5-13. Modific...

decodificação ...

5-14. Faga u...

Sobre microc...

5-15. Faga u...

(a) PLA. (b)

5-16. Compre...

"endereços", ...

5-17. Quais s...

5-18. Adminis...

as formas de ...

PRANCHA XVI	
EPUSP	LSD COMPUTADOR DE CONTROLE
SINALS DE CONTROLE	Nº CTS-6 11
PROJETO: FREINI	DATA: 10-71
DESENHO: MAZZEI	DATA: 10-12-71
REVISÃO: STEPHAN	DATA: 14-12-71
SITUAÇÃO:	

EXERCÍCIOS

Sobre a arquitetura

- 5-1. Projete uma arquitetura para uma memória de 1K palavras de 8 bits.
 5-2. O que se perderia se usássemos um somador de 4 bits com velocidade igual ao dobro do de 8 bits? Quais as modificações na arquitetura?

Sobre as instruções

- 5-3. O que aconteceria com a instrução *SUS* com endereço zero? Qual sua utilidade?
 5-4. O que aconteceria se o programador, por engano, incluisse no meio de um programa a instrução de *PUG* para o próprio endereço da *PUG*?
 5-5. Se o conteúdo do acumulador for 11001011 e $V = 1$, qual o conteúdo do acumulador e de V depois da instrução “giro à esquerda com V ” (*GEV*) de três posições?
 5-6. Qual o conteúdo do acumulador, depois de executar a instrução *NAND* com 00001111 (a instrução seria 1101 0100 0000 1111), sabendo que, inicialmente, seu conteúdo era 1101 1001?
 5-7. Qual a maneira mais rápida de multiplicar por *menos um* conteúdo do acumulador? Lembre-se de que o código é o complemento de dois.
 5-8. Como se faz $T = 0$ (“limpa transbordamento”), sem alterar-se o acumulador?
 5-9. E como se faz $V = 1$ (“dispara vai um”) sem alterar o acumulador?
 5-10. Faça um programa que guarde, nas posições $(101)_{10}$ e $(102)_{10}$, o resultado da soma dos números A e B , sendo ambos em dupla precisão armazenados em: A , nas posições 103 e 104, e B , nas posições 105 e 106.

Sobre o fluxo de dado

- 5-11. Faça um esquema completo da unidade aritmética, incluindo as saídas V_A e T .
 5-12. Projete o circuito de “mais um” visto na Fig. 5-14.
 5-13. Modifique o relógio central, mudando o *overlap ring counter* para um contador binário com um decodificador para gerar os sinais corretos.
 5-14. Faça um esquema do registrador de instrução.

Sobre microoperações e controle

- 5-15. Faça um esquema das microoperações, fase por fase, explicando cada uma delas, das instruções: (a) *PLA*, (b) *ARM*, (c) *SOMI*, (d) *SUS*, (e) *PUG*, (f) *TRE*, (g) *TRI*.
 5-16. Como seria a seqüência das microoperações de uma instrução que tivesse o formato *SUB* e “endereço”, que equivaleria a subtrair do acumulador o operando endereçado no campo “endereço”?
 5-17. Quais os sinais elétricos de controle que produzem a micro-instrução $RD \leftarrow AC + RD$?
 5-18. Admita que um programador se enganasse e fizesse um *PLA* para seu próprio endereço. Desenhe as formas de onda dos seguintes sinais: (a) fase 1, (b) memória “modo 4”, (c) *VUQ*, (d) *RI(4-7)/M*.