Curso: Engenharia da Computação
ARQUITETURA DE COMPUTADORES
Prof. MSc. Luís Caldas - 2017

CURSO: ENGENHARIA DA COMPUTAÇÃO
SÉRIE: 6º SEMESTRE
DISCIPLINA: Arquitetura de Computadores
CARGA HORÁRIA SEMANAL: 02 Hora/aula
CARGA HORÁRIA SEMESTRAL: 40 horas

I – EMENTA

Conceituação de arquitetura de computadores através de um breve histórico, tratamento de números em ponto flutuante, normalização, operações em ponto flutuante, algoritmos de alto desempenho, conjunto de instruções de uma máquina RISC, tipos de barramento, modos de endereçamento, montadores, microprogramação, pilhas e sub-rotinas, hierarquia de memória e virtual e dispositivos de entrada e saída.

II - OBJETIVOS GERAIS

Conceituar e analisar os elementos que compõem os diversos níveis da arquitetura de computadores modernos, abrangendo desde o nível de lógica digital até o de programação de aplicativos; Oferecer um modelo de referência, que deve permitir não só uma melhor compreensão das estruturas internas de sistemas computacionais, como também a organização dos conhecimentos adquiridos pelo aluno ao longo do curso, pela observação de como as diferentes disciplinas se inter-relacionam no contexto da arquitetura de computadores.

III - OBJETIVOS ESPECÍFICOS

Apresentar uma visão de como a configuração dos elementos pertencentes a um determinado nível interferem naquele e nos demais níveis. Introduzir o conceito de máquina virtual. Relacionar conceitos teóricos a exemplos práticos e a sistemas de mercado. Introduzir a programação em linguagem *assembly*. Apresentar tendências de evolução e arquiteturas avançadas de computadores. Mostrar uma visão geral das arquiteturas de computadores. Fornecer conceitos de processadores, memória, unidades de entrada/saída e periféricos. Fornecer conceitos de arquiteturas avançadas RISC, CISC e paralelas.

IV - CONTEÚDO PROGRAMÁTICO

1. Breve histórico do desenvolvimento dos computadores. Gerações de computadores.
2. Representação dos dados. Números em notação fixa e em ponto-flutuante. Norma IEEE 754.
3. Aritmética em ponto fixo, unidade de adição e subtração. Operações de multiplicação e divisão.
4. Caracterização de processadores RISC e CISC. Aspectos de desempenho.
   Tipos de arquiteturas e endereçamento.
5. Arquitetura do conjunto de instruções. Modos de endereçamento.
   Conjunto de instruções. O formato dos dados ARC.
6. Exemplos de programas em linguagem de montagem.
7. Pilhas (LIFO/FIFO) e Sub-rotinas.
8. Entrada e Saída em Linguagem de montagem.
9. Microarquitetura básica. Seção de controle. Desenvolvimento de um microprograma. Nanoprogramação.
10. Estudo de caso com VHDL. Estudo de casos com microprocessadores da família Intel e Motorola.
11. A hierarquia de memória. Estudo de memória Cache, RAM, ROM. Memória virtual.
12. Armazenamento de massa. Tipos de dispositivos físicos. Tecnologia RAID.

V - ESTRATÉGIA DE TRABALHO

A disciplina será desenvolvida com aulas expositivas, sendo incentivada a participação dos alunos nos questionamentos e discussões apresentadas, acompanhadas de metodologias que privilegiam a integração entre teoria e prática, entre elas: estudos de casos, elaboração de trabalhos práticos e produção de textos.

VI – AVALIAÇÃO

A avaliação será realizada por intermédio de provas regimentais e atividades desenvolvidas em sala de aula, conforme solicitação do professor da disciplina, tendo como referência as metodologias adotadas de integração entre teoria e prática.

VII – BIBLIOGRAFIA

Básica

MURDOCCA, M. J.; Heuring, V. P. **Introdução a Arquitetura de Computadores.** Campus, 2001.

STALLINGS, W. **Arquitetura e organização de computadores**. Prentice Hall, 2003.

TANENBAUM, A. S. **Organização estruturada de computadores**. Prentice Hall Brasil, 2007.

Complementar

HAYES, J. **Computer architecture and organization**. 2ª ed. McGraw Hill Publishing Company, 1988.

DAVID A. PATTERSON. **Organização e Projeto de Computadores. Interface harware/software**. 4ª ED. ISBN-13: 978-0123747501

FRANÇA, P.; ZELENOVSKY, R.; MENDONÇA, A. **Hardware**: **programação virtual de I/O e interrupção**. MZ, 2001.

OLIVEIRA, J F de; MANZANO, J A N G. **Algoritmos**: **lógica para desenvolvimento de programação de computadores**. Érica, 2009.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos da programação de computadores**: **Algoritmos, Pascal e C/C++.** São Paulo: Prentice Hall, 2002.

HISTÓRIA DO DESENVOLVIMENTO DOS COMPUTADORES

1. Introdução

Os dispositivos mecânicos serviram desde o século XVI para controlar operações complexas e eles têm sido utilizados para aplicações, tais como:

- Cilindros mecânicos para caixas de músicas;
- Máquinas de calcular, calculadora de Pacal "Pascalene",

A partir do século XIX, conceitos de cálculo mecânico e controle mecânico foram introduzidos na máquina de Charles Babbage. Conhecido como avô do computador e não pai do computador em virtude de não ter construído uma versão prática. O computador de Babbage podia calcular tabelas simplesmente girando engrenagens. A máquina tinha a capacidade de ler dados de entrada, armazenar esses dados, fazer cálculos, gerar dados e controlar a operação da máquina. Babbage criou um protótipo de sua máquina e o governo apoiou um projeto para construir uma máquina analítica, mas nunca foi construída devido as tolerâncias mecânicas exigidas pelo projeto. Existe no museu da ciência em Londres uma versão da máquina analítica construída em 1991.

Na segunda guerra mundial uma máquina conhecida como ENIGMA usava um código criptográfico para construir e transmitir suas mensagens para os submarinos alemães. O código era alterado diariamente e a sua decodificação manual se tornava impossível nesse período de tempo de 24 horas. Embora todos os esforços foram aplicados usando cientistas e peritos em codificação criptográfica não se consegui resultado positivo e muitas vezes esse apoio do governo foi questionado. A equipe liderada por Alan Turing enfrentou esse desafio por longo tempo e Turing chegou a conclusão que somente uma máquina poderia realizar essa decodificação em tempo curto. E assim ele criou a máquina de Turing e muito mistério que envolve o projeto se tornou bastante obscuro.

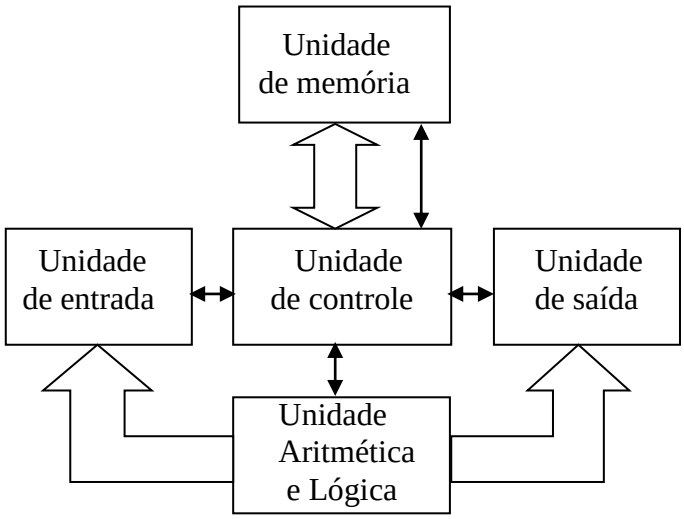
No mesmo período do trabalho de Turing, Eckert e Mauchy projetaram uma máquina para calcular a trajetória balística para o exército dos Estados Unidos. O resultado foi o Integrador e Computador Numérico Eletrônico (ENIAC). Usava 18.000 tubos de vácuo na parte computacional da máquina. Não existia o conceito de programa armazenado e não exista a memória principal.

2. Modelo da máquina de Von Neumann

Neumann trabalhou num projeto de um computador chamado de EDVAC. O modelo consiste em cinco componentes principais, como a seguir. As partes básicas desse computador foi uma base introdutória para os computadores atuais e são elas:

1. Unidade de controle;
2. Unidade de memória;
3. Unidade lógica e aritmética;
4. Unidade de entrada;
5. Unidade de saída.

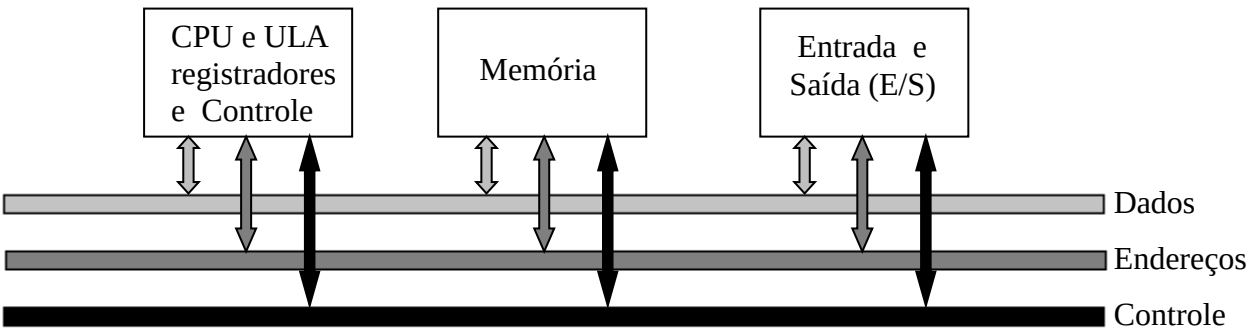
As conexões entre as cinco unidades mostram uma arquitetura de um computador e permitiram o desenvolvimento de compiladores e sistemas operacionais de grande versatilidade, pois com o programa armazenado foi possível manipular como se fosse um dado. Os programas antes eram armazenados em dispositivos cartões perfurados etc...



A figura 1.1 mostra a máquina de Von Neumann, onde seta grande é o caminho dos dados e seta fina o caminho dos sinais de controle da unidade de controle sobre as unidades.

3. Modelo de Barramento do sistema

Embora o modelo de Von Neumann esteja presente em computadores modernos esse modelo foi modernizado para o modelo de barramento de sistema. Com a evolução da tecnologia de circuitos integrados a saída "Three state" ou terceiro estado permitiu a interligação das saídas de dados em um barramento compartilhado sem provocar "curto-circuito", pois a unidade de controle gerencia o barramento de dados evitando conflito entre as saídas. O **barramento** permite particionar o sistema computacional em três unidades funcionais a saber: CPU, memória e entrada/saída (E/S). O modelo de barramento combina a ULA e a U.C em uma unidade funcional chamada de unidade central de processamento CPU. As unidades de entrada e de saída são combinadas em unidade de entrada/saída E/S. O barramento de dados transporta a informação que está sendo transmitida e da mesma forma outros barramentos foram criados como o barramento de endereços que identifica para onde os dados devem ser enviados e o barramento de controle que permite a transmissão e recepção dos dados pelas unidades de memória de entrada/saída controlando a direção dos dados gerenciando os barramentos. A figura a seguir mostra o modelo de barramento do sistema computacional.



Barramentos do Sistema

3. DESCRIÇÃO DO BARRAMENTO DO SISTEMA

3.1 Barramento de dados.

Os barramentos são realizados por uma coleção de fios que são agrupados por função. O barramento de dados de oito bits tem oito fios individuais, cada um dos quais transporta um bit de dados nas duas direções ou para leitura ou escrita. Para um barramento compartilhado com todas as unidades que se comunicam entre si, então, elas devem ter endereços que as identificam. Quando o sistema de entrada/saída é mapeado na memória, então todos os endereços são endereços de memória. Quando o sistema de entrada/saída é isolado, então o endereço separado do endereço de memória deve ser identificado. Nesse caso, o barramento de controle auxilia na identificação gerando sinal de controle E/S ou de Mem quando o endereço for para a unidade de E/S ou para a unidade de memória.

3.2 Barramento dos endereços

O barramento de endereços identifica a localização dos dados na memória. O endereço de memória é a localização onde os dados serão armazenados ou lidos. A operação é controlada pela CPU a qual informa a direção dos dados e a operação de leitura é feita pela CPU lendo os dados na memória e a operação de escrita, a CPU transmite os dados para o endereço de memória de onde serão armazenados.

3.3 Barramento de controle

O barramento de controle identifica às unidades se os dados serão lidos ou escritos pela CPU ou entre as unidades. Esses sinais de controle serão gerados através de uma palavra de controle pela CPU. Cada instrução de programa que está sendo executada pela CPU, os sinais de controle microprogramados na CPU são gerados para o controle do fluxo de dados.

4. Nível das máquinas

O nível mais alto no computador é o nível do usuário e o nível mais baixo o nível dos transistores. Cada um dos níveis representa uma abstração do computador. Existem muitos níveis entre o nível do usuário e o nível dos transistores e o interessante é o grau de separação entre esses níveis e a independência entre eles. Um usuário que executa em editor de textos em um computador não necessita saber da programação e da mesma forma que um programador não precisa saber das interligações das portas lógicas.

5. Compatibilidade “para cima”

A invenção do transistor levou ao desenvolvimento do hardware e com isso a um problema de portabilidade do software desenvolvido em uma máquina antiga e que possa rodar em uma máquina mais nova. As máquinas IBM rodavam programas tanto nas máquinas mais poderosas como nas máquinas menos poderosas e o conceito de “família de computadores” de sua série. Permitiu que os programadores não mais necessitavam reescrever o software.

6. Os níveis

São sete níveis no computador, do nível do usuário ao nível dos transistores. Quando se caminha do nível superior para o nível inferior os níveis ficam menos abstratos e mais da estrutura interna do computador pode ser vista. A seguir apresentamos os sete níveis do computador.

Alto Nível	Nível do Usuário: Programas Aplicativos
	Linguagem de Alto nível
	Linguagem de montagem/Código de máquina
	Controle Microprogramado/Fixo
	Unidades funcionais (memória, ULA, etc...)
	Portas Lógicas
Baixo Nível	Transistores e fios

Níveis da máquina na hierarquia do computador

1. Transistores e fios e Portas Lógicas – Esse é o nível mais baixo do computador e por ele é responsável pelo hardware. As portas lógicas estão inseridas nos circuitos integrados. As tensões e correntes presentes nesse nível e dependendo da tecnologia empregada na confecção dos circuitos está ligada ao desempenho da máquina e da potência dissipada.

2. Unidades funcionais – As transferências entre registradores RTL e outras operações implementadas pela unidade de controle movem dados para dentro das unidades funcionais, as quais calculam alguma função que é importante na operação do computador. As unidades funcionais incluem registradores internos da CPU, da ULA e da memória principal

3. Controle Microprogramado/Fixo – A unidade de controle é responsável na geração dos sinais de controle do fluxo de dados, da tomada de decisão. Controlada por um sinal de relógio e a cada ciclo de clock uma operação interna é realizada. Uma transferência entre registradores, o conteúdo interno do registrador é transferido para um segundo registrador e esse movimento dos dados é controlado pela unidade de controle, a qual gera um sinal de controle de leitura dos dados no registrador fonte e ao mesmo tempo a unidade de controle gera concomitantemente um sinal de escrita para o gerador destino. Na borda seguinte do relógio essa operação é efetivada. A unidade de controle pode ser fixa ou microprogramada e dependendo da máquina existem vantagens no uso dessas unidades de controle. A unidade de controle fixa é conhecida como hardwired, vantagens como velocidade e tamanho, mas difícil de implementação. A unidade de controle microprogramada tem internamente um pequeno programa escrito em linguagem de baixíssimo nível e implementada em hardware. A função dessa linguagem é interpretar instruções de linguagem de máquina (códigos de operações “opcode”). O microprograma é chamado de firmware, poraquê parte é hardware e parte é software e é executado por um microcontrolador o qual executa as microinstruções.

4. Linguagem de montagem/Fixo – A linguagem de mais baixo nível que um computador executa um programa é chamada de linguagem de máquina (opcode). As instruções são identificadas pelo cordão de bits “1” e “0” formando um conjunto de 8, 16 ou mais bits. Embora seja a linguagem que o computador entende é mais difícil de verificar erros bem como de programar. Os montadores foram criados para facilitar a vida do programador, pois usando a linguagem de montagem, o programa torna

muito mais fácil de "debug" conferência e correção de erros e a função do montador é transformar o programa fonte criado pelo programador em programa objeto (linguagem de máquina).

5. Linguagem de alto nível – A tarefa de programação fica muito mais fácil quando se utiliza uma linguagem de alto nível. Como o computador entende somente linguagem da máquina é necessário a conversão dessa linguagem de alto nível para máquina. Os interpretadores e compiladores são utilizados para essa finalidade.

6. Nível do usuário – O nível mais alto na hierarquia, o usuário utiliza os aplicativos entre outros, tais como, banco de dados, planilhas eletrônicas, editores de textos entre outros. Esses aplicativos são escritos normalmente em linguagens de alto nível e compiladas para linguagem de máquina para rodar no computador

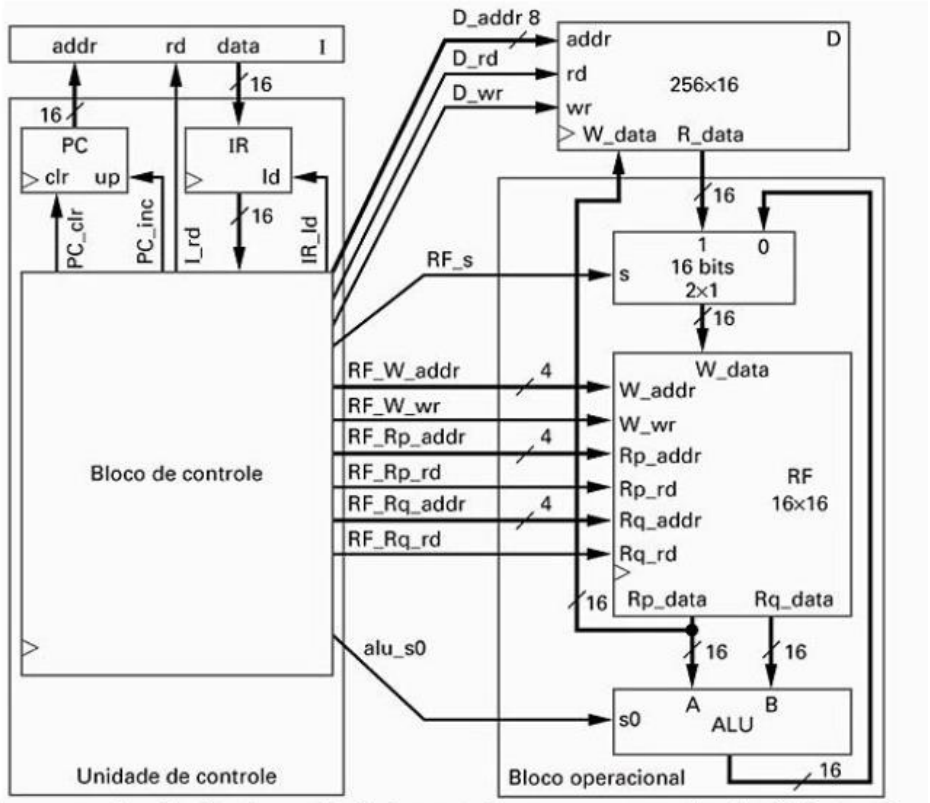
7. Máquinas paralelas - Podem ser divididas em três categorias, baseando-se no número de fluxos de instruções e de dados que elas têm:

1. SISD - Single Instruction, Single Data Fluxo único de instruções e de dados.
2. SIMD - Single Instruction, Multiple Data Fluxo único de instruções e múltiplo de dados.
3. MIMD - Multiple Instruction, Multiple Data Fluxo múltiplo de instruções e de dados.

A máquina tradicional de von Neumman é SISD. Ela tem apenas um fluxo de instruções (i. é, um programa), executado por uma única CPU, e uma memória conectando seus dados. A primeira instrução é buscada da memória e então executada. A seguir, a Segunda instrução é buscada e executada. Máquinas SIMD, ao contrário, operam uns múltiplos conjuntos de dados em paralelo. Uma aplicação típica para uma máquina SIMD é a previsão do tempo. Imagine o cálculo da temperatura média diária a partir de 24 médias horárias para muitos locais. Para cada local, exatamente o mesmo cálculo precisa ser feito, porém com dados diferentes. A terceira categoria de Flynn é a MIMD, na qual CPUs diferentes executam programas diferentes, às vezes compartilhando alguma memória em comum. Por exemplo, no sistema de reserva de passagens aéreas, reservas simultâneas múltiplas não 16 prosseguem em paralelo, instrução por instrução, e assim temos fluxo múltiplo de instrução e fluxo múltiplo de dados. Outros sistemas multiprocessadores usam não apenas um barramento, mas vários para reduzir a carga. Outros usam ainda uma técnica chamada cache, que consiste em manter as palavras de memória freqüentemente referidas dentro de cada processador.

PROCESSADOR DE USO GERAL

1. Introdução: Um sistema digital completo é constituído de uma unidade controle e o bloco operacional. Na figura a seguir alguns elementos externos necessários para completar a arquitetura do computador como a memória de programa e a memória de dados. A memória de programa é responsável pelo programa a ser lido e executado pela unidade de controle e a memória de dados é responsável para o armazenamento dos dados a serem lidos e armazenados no bloco operacional. Essa arquitetura é de largura de 16 bits e a capacidade da memória de programa é de 65k x 16 bits e da memória de dados de 256 x 16bits. A figura a seguir mostra um modelo de computador baseado na máquina de Von Neumann.



Bloco operacional refinado e unidade de controle para o processador de três instruções.

Descrição do hardware

PC = 16 bits sendo de 0000 a FFFF (0 a 65535)
IR = 16 bits opcode = 4 bits de 0 a F, reg. ra = 4 bits, rfb = 4bits e rc = 4bits e d = 8 bits, onde ra se refere ao registrador destino e rb e rc são registradores fontes de operandos e d = endereço de memória de dados.

Exemplo: 0010 0000 0001 0010 – A instrução 0010 é uma instrução de soma entre registradores fontes e (opcode = 0010), os registradores fonte rb = 0001 e rc = 0001 e destino ra = 0000.

Uma descrição breve da operação desse computador.

Um programa escrito em linguagem de máquina é inserido na memória de programa do computador e será acessado pelo registrador de endereço de programa chamado de contador de programa PC.

Ciclo de busca

1. PC = retém o endereço da instrução a ser buscada pela unidade de controle;
2. PC = incrementa o contador (PC +1) apontando para a próxima instrução a ser buscada;
3. IR = contém a instrução a ser decodificada pela unidade de controle;

Ciclo de Execução

O bloco de controle gera os sinais de controle da U.C para o fluxo de dados de acordo com o código de operação da instrução (opcode).

Exemplo: A instrução ADD Ra[0], Rb[1], Rc[2] manda somar o conteúdo do registrador 1 ao conteúdo do registrador 2 e armazenar o resultado no endereço do registrador 0.
Rf[0] = Rf[1] + Rf[2]

Os sinais de controle gerados pela U.C. para a realização dessa operação de soma ficam:

Rf_W_addr = 0000
Rf_W_wr = ativo
Rf_Rp_addr = 0001
Rf_Rp_rd = ativo
Rf_Rq_addr = 0010
Rf_Rq_rd = ativo
S0 = soma
Rf_s = 0

Os sinais de controle são chamados de microinstruções e são próprias para cada uma das instruções do computador. As operações são controladas pelo sinal de relógio e a cada ciclo de relógio é um ciclo de máquina. A instrução completa consome um total de 3 ciclos de máquina e esse é o ciclo de instrução. A microinstrução de controle PC = PC +1 é executada quando a instrução está sendo depositada no registrador de instrução.

Modelo de Von Neumann

Em 1946, Von Neumann e sua equipe iniciaram o projeto de um computador de programa armazenado. O projeto foi elaborado em Princeton no Instituto de Estudos avançados IAS. Essa máquina usava como memória principal tubo de raios catódicos de acesso randômico o que permitia o acesso a uma palavra inteira em uma única operação. A instrução continha um endereço de memória e tinha o seguinte formato.

OP A

ARQUITETURA DE MÁQUINAS DE 4, 3, 2, 1, 0 ENDEREÇOS

As arquiteturas podem operar basicamente com endereços de memória e esses influenciam nas instruções e na própria arquitetura. A máquina IAS é uma máquina de um endereço e outras podem ter até 4 endereços de memória, embora a capacidade sejam as mesmas. Para resumir as diferenças entre

as arquiteturas de diferentes endereços de memória é colocado uma programação de uma equação aritmética.

A = (B + C)*D + E – F)/G*H,

onde as letras A,...H denotam endereços de memória e as arquiteturas possuem as 4 operações aritméticas. A instrução ADD = Soma, SUB = Subtração, MUL = Multiplicação e DIV = divisão.

a. Arquitetura de 4 endereços

Uma máquina de 4 endereços utiliza OP como a operação a ser realizada entre os operandos fontes e o operando resultado. O destino o endereço da próxima instrução vem indicado na instrução.

A instrução OP E1,E2,E3,E4

OP = Opcode – Operação a ser realizada e E1 e E2 indicam a localização dos dois operandos fontes e E3 sinaliza a localização do operando destino (onde o resultado deve ser armazenado) e E4 aponta o endereço onde está localizada a próxima instrução a ser executada.

Endereço	**Mnemonica**	**Comentário**
e1	ADD B,C,A,e2	Soma B com C, resultado em A, vai para e2
e2	MUL A,D,A,e3	Multiplica A por D, resultado em A, vai para e3
e3	ADD A,E,A,e4	Soma A com E, resultado em A; vai para e4
e4	SUB A,F,A,e5	Soma A com F, resultado em A; vai para e5
e5	DIV A,G,A,e6	Divide A por G, resultado em A; vai para e6
e6	DIV A,H,A,e7	Divide A por H, resultado em A; vai para e7
e7	HLT	Fim de programa

Essa máquina de quatro endereços ocupa muito espaço de memória, uma vez que são quatro endereços. Como o programa é montado com endereço sequencial, então a máquina de três endereços poderá substitui-la.

b. Arquitetura de 3 endereços

Uma máquina de 3 endereços utiliza OP como a operação a ser realizada entre os operandos fontes e o resultado, destino vem indicado, não sendo necessário carregar o endereço da próxima instrução.

A instrução OP E1,E2,E3

OP = Opcode – Operação a ser realizada e E1 e E2 indicam a localização dos 2 operandos fontes e E3 sinaliza a localização do operando destino (onde o resultado deve ser armazenado). No lugar de E4 que aponta o endereço onde está localizada a próxima instrução a ser executada foi criado um registrador de endereço chamado de PC contador de instrução, o qual é incrementado automaticamente na busca da instrução. O programa fica:

Endereço	**Mnemonica**	**Comentário**
e1	ADD B,C,A	Soma B ao C e o resultado em A = (B+C).
e1+1	MUL A,D,A	Multiplica A por D e o resultado em A = (B+C)*D.
e1+2	ADD A,E,A	Soma A ao E e o resultado em A = (B+C)*D + E.
e1+3	SUB A,F,A	Subtrai A de F e o resultado em A = (B+C)*D +E – F.

e1+4	DIV A,G,A	Divide A por G e o resultado em A = (B+C)*D + E – F/G
e1+5	DIV A,H,A	Divide A por H e o resultado em A = (B+C)*D +E – F/G*H.
e1+6	HLT	Pára o programa.

Com a redução da máquina de 4 para 3 endereços houve uma diminuição no tamanho da instrução e consequentemente uma redução no tamanho da memória. Dessa forma perde-se em flexibilidade, pois os programas são contíguos, ou seja, instrução a instrução consecutivamente e assim instruções de saltos ou de desvios ou mesmo de decisões ficam limitados com essa arquitetura. Como vantagem além do tamanho reduzido de memória em relação à máquina de 4 endereços é a obrigatoriedade de se fazer programa sequenciais, mais fáceis de elaborar e mais fáceis de serem corrigidos.
Embora a limitação da arquitetura da máquina de 3 endereços, ainda se consome muita memória e, portanto, foi criada a máquina de 2 endereços.

c. Arquitetura de 2 endereços

Uma máquina de 2 endereços utiliza OP como a operação a ser realizada entre os operandos fontes sendo um deles o acumulador e também o resultado.

A instrução OP E1,E2

OP = Opcode – Operação a ser realizada e E1 e E2 indicam a localização dos 2 operandos fontes e não é necessário indicar a localização do operando destino (onde o resultado deve ser armazenado), pois as operações são feitas sempre no registrador A.

Endereço	**Mnemonica**	**Comentário**
e1	MOV A,B	Transfere B para o acumulador A = B.
e1+1	ADD B,C	Soma B ao C e o resultado em A = (B+C).
e1+2	MUL A,D	Multiplica A por D e o resultado em A = (B+C)*D.
e1+3	ADD A,E	Soma A ao E e o resultado em A = (B+C)*D + E.
e1+4	SUB A,F	Subtrai A de F e o resultado em A = (B+C)*D + E – F.
e1+5	DIV A,G	Divide A por G e o resultado em A = (B+C)*D + E – F/G.
e1+6	DIV A,H	Divide A por G e o resultado em A = (B+C)*D + E – F/G*H.
e1+7	HLT	Pára o programa.

A diferença entre o programa da máquina de 3 endereços com a máquina de 2 endereços é que o número de operandos na instrução diminuiu e por exemplo uma instrução ADD A,B,A na máquina de 3 endereços foi acrescentada uma instrução de movimentação de dados como MOV A,B e assim aumentou uma instrução nessa operação. Como a instrução diminuiu de tamanho, a largura da memória também diminuiu e aumentou as instruções, mas no computo geral há uma diminuição do número de bits.

d. Arquitetura de um endereço

Uma máquina de 1 endereço utiliza OP como a operação a ser realizada entre os operandos fontes e operação de transferência para o acumulador também o resultado.

A instrução OP E1

O Opcode seguido de somente 1 operando E1. É óbvio que o outro operando fonte é implícito. Como nas outras máquinas o registrador acumulador conhecido AC ou ACC é um operando fonte e destino

ao mesmo tempo. Para a realização do programa as operações de manipulação foram necessárias sendo a transferência de dados entre memória e acumulador tipo LDA (Load AC) e a transferência entre o acumulador e a memória STA. O programa deverá ser alterado para executar as operações.

Endereço	**Mnemonica**	**Comentário**
e1	LDA B	Transfere o operando B para o registrador A.
e1+1	ADD C	Soma o acumulador com C e o resultado em A = (B+C).
e1+2	MUL D	Multiplica o acumulador com D e o resultado em A = (B+C)*D.
e1+3	ADD E	Soma o acumulador com E e o resultado em A = (B+C)*D + E.
e1+4	SUB F	Subtrai o acumulador com F e o resultado em A = (B+C)*D + E - F.
e1+5	DIV G	Divide o acumulador com G e o resultado em A = (B+C)*D + E – F/G.
e1+6	DIV H	Divide o acumulador com H e o resultado em A = (B+C*D +E – F/G*H.
e1+7	STA A	Armazena o acumulador no endereço de A.
e1+8	HLT	Pára o programa.

e. Arquitetura de zero endereço

Na máquina zero endereço, a instrução deve conter o operando e as operações aritméticas são realizadas na memória. Para funcionar utiliza-se uma parte da memória chamada de pilha. A\ pilha opera no modo LIFO (last-in first-out), ou seja, o último que entra é o primeiro a sair e os dados ou operandos são empilhados em cada endereço de memória. Por exemplo entrando na pilha os operandos A,B,C,D e E nessa sequência e ordem a partir do endereço n, após a entradas dos operandos a ordem na pilha será o seguinte. Endereço n-4 = A, n – 3 = B, n – 2 = C, n – 1 = D e n = E. As instruções que realizam essa operação são elas PUSH E1 e POP A. A instrução PUSH B empurra para o topo da pilha, o conteúdo de B e a instrução POP A remove o operando do topo da pilha para o registrador A.

A instrução: OP

Endereço	**Mnemonico**	**Comentário**
e1	PUSH H	Coloca o operando H no topo (atual) da pilha.
e1+1	PUSH G	Coloca o operando G no topo da pilha.
e1+2	PUSH F	Coloca o operando F no topo da pilha.
e1+3	PUSH E	Coloca o operando E no topo da pilha.
e1+4	PUSH D	Coloca o operando D no topo da pilha.
e1+5	PUSH C	Coloca o operando C no topo da pilha.
e1+6	PUSH B	Coloca o operando B no topo da pilha.
e1+7	ADD	Topo da pilha recebe Soma (B + C) e B e C são retirados da pilha.
e1+8	MUL	Topo da pilha recebe o resultado do produto (B + C)*D.
e1+9	ADD	Topo da pilha recebe o resultado do produto (B + C)*D + E.
e1+10	SUB	Topo da pilha recebe o resultado do produto (B + C)*D + E – F.
e1+11	DIV	Topo da pilha recebe o resultado do produto (B + C)*D + E – F/G.
e1+12	DIV	Topo da pilha recebe o resultado do produto (B + C)*D + E – F/G*H.
e1+13	POP A	Topo da pilha é armazenado em A.
e1+14	HLT	Pára o programa.

SISTEMAS DE NUMERAÇÃO E REPRESENTAÇÃO DOS NÚMEROS

Sistemas de Numeração e Somadores Binários

I – Base Numérica

Um número em uma base qualquer pode ser representado da forma:

$N = A_{n-1}.B^{n-1} + A_{n-2}.B^{n-2} + ...+ A_1.B^1 + A_0.B^0$, onde $A < B$ e $B > 0$, sendo A_i e B^i os coeficientes
da base e B a base do número.

Dessa forma pode-se transformar qualquer número em qualquer base para a base 10.

Exemplo: $N = (431)_5$.

$N = 4 . 5^2 + 3 . 5^1 + 1 . 5^0 = 100 + 15 + 1 = (116)_{10}$.

Base 10	Base 2	Base 3	Base 4	Base 5	Base 8	Base 16	Base 12
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	10	2	2	2	2	2	2
3	11	10	3	3	3	3	3
4	100	11	10	4	4	4	4
5	101	12	11	10	5	5	5
6	110	20	12	11	6	6	6
7	111	21	13	12	7	7	7
8	1000	22	20	13	10	8	8
9	1001	100	21	14	11	9	9
10	1010	101	22	20	12	A	A
11	1011	102	23	21	13	B	B
12	1100	110	30	22	14	C	10
13	1101	111	31	23	15	D	11
14	1110	112	32	24	16	E	12
15	1111	120	33	30	17	F	13
16	10000	121	100	31	20	10	14
17	10001	122	101	32	21	11	15
18	10010	200	102	33	22	12	16
19	10011	201	103	34	23	13	17
20	10100	202	110	40	24	14	18
21	10101	210	111	41	25	15	19
22	10110	211	112	42	26	16	1A
23	10111	212	113	43	27	17	1B
24	11000	220	120	44	30	18	20

II - Outras Bases.

Sistema de Numeração	Base
Horas em Min e Seg.	60
Dia em Horas	24
Mês em Dias	30
Ano em Mêses	12
Estações do ano	4
Algarismos Romanos	10

Sistema de Numeração	Base
Círculo em gráus	360
Século	100
Milênio	1000
Semestre	6
Semana em dia	7
Algarismo arábico	10

Exercício: Transformar os números a seguir para a base 10.

1) N = $(101111)_2$ = N = 47_{10}

2) N = $(123)_4$ = N = 27_{10}

3) N = $(702)_8$ = N = 514_{10}

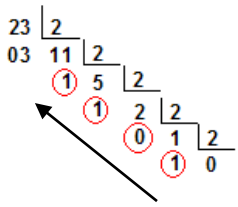
4) N = $(A12)_{16}$ = N = 2578_{10}

III - Transformação de números de uma base qualquer para outra base.

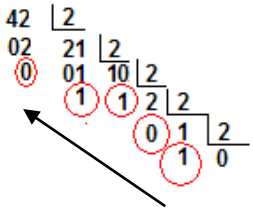
Regra: Divide-se o número a ser transformado pela base desejada até o quociente ser 0.

Exemplo: Transformar o número:

1) N = $(23)_{10}$ para a base 2. N = $(10111)_2$



2) N = $(420)_5$ para a base 2. N = $(10110)_2$.



Observação: Pode-se transformar o número inicialmente para a base 10 e em seguida converter para a base desejada.

Exercícios: Transformar os números.

1) N = $(32)_{10}$ para a base 4.

N = 202_4

2) N = $(101111)_2$ para a base 5.

N = 142_5

3) N = $(A10)_{16}$ para a base 8.

N = 101 000 010 000 = 5020_8

4) N = $(420)_5$ para a base 3.

N = 112_3

IV - Transformação entre bases múltiplas de 2^n.

Na transformação entre bases múltiplas de 2^n, a conversão é realizada no número a ser transformado codificado em binário e iniciando-se pelo dígito menos significativo em direção ao dígito mais significativo (direita para a esquerda do número), separando de n em n casas, onde n é o número de bits da base.

Base 2 => n =
Base 4 => n =

Base 8 => n =
Base 16 => n =
Exemplo: N = $(32)_4$ para a base 8.

Após a codificação em binário de cada coeficiente 3 = $(11)_2$ e 2 = $(10)_2$, para base 8, n = 3 temos:

N = 1110 = N = 1 | 110 | = $(16)_8$.

Exercícios: Transformar o número $(FA3)_{16}$, para as bases:

a) Base 2 = $111\ 110\ 100\ 011_2$
b) Base 4 = 332203_4
c) Base 8 = 7643_8

V – Codificação binária para números na base 10.

a) BCD – Codificação binária Decimal – 8421

2^3	2^2	2^1	2^0	-
8	**4**	**2**	**1**	**N.o**
0	**0**	**0**	**0**	**0**
0	**0**	**0**	**1**	**1**
0	**0**	**1**	**0**	**2**
0	**0**	**1**	**1**	**3**
0	**1**	**0**	**0**	**4**
0	**1**	**0**	**1**	**5**
0	**1**	**1**	**0**	**6**
0	**1**	**1**	**1**	**7**
1	**0**	**0**	**0**	**8**
1	**0**	**0**	**1**	**9**

b) BCD - Codificação binária Decimal – 5211

5	2	1	1	N.o
0	**0**	**0**	**0**	**0**
0	**0**	**0**	**1**	**1**
0	**0**	**1**	**1**	**2**
0	**1**	**0**	**1**	**3**
0	**1**	**1**	**1**	**4**
1	**0**	**0**	**0**	**5**
1	**0**	**0**	**1**	**6**
1	**0**	**1**	**1**	**7**
1	**1**	**0**	**1**	**8**
1	**1**	**1**	**1**	**9**

Exercícios: Realizar a operação de subtração entre os números:

a) $N_1 = (1011110011)_2$ e $N_2 = (10010100010)_2$

a) $S = N_1 + N_2$

Exercícios de subtração em bases diversas

$N_1 = (19)_{10}$ $N_2 = (14)_{10}$ $N_3 = (23)_4$ $N_4 = (15)_4$ $N_5 = (6714)_8$ $N_6 = (621)_{16}$ $N_7 = (1001111)_2$ $N_8 = (221)_4$

b) $S = N_1 + N_2$.
c) $S = N_3 + N_4$
d) $S = N_5 + N_6$ (resultado na base 8).
e) $S = N_7 + N_8$ (resultado na base 2).

CODIFICAÇÃO DE NÚMEROS FRACIONÁRIOS DE UMA BASE QUALQUER

Os números inteiros vimos como transformar e representar em uma determinada base, agora quando os números são fracionários como são representados.
Para os números fracionários a representação em uma determinada base é idêntica à representada na base 10. Coloca-se uma virgula à direita do número inteiro e na parte fracionária colocamos os coeficientes do número. Por exemplo na base decimal o primeiro coeficiente ã direita da virgula, portanto a parte fracionária do número tem peso igual a 1/10 ou 0,1 e assim por diante o segundo 1/100, o terceiro 1/1000. Ou seja a notação para a parte fracionária ficará :

N = ,$1/B^1 + 1/B^2 + 1/B^3 + 1/B^4$ ou representado por $B^{-1} + B^{-2} + B^{-3} + B^{-4} + ... + B^{-n}$.

Exemplo: O número $(21221,012)_3$, a parte inteira é 21221 e a parte fracionária do número é 012.

Exemplo: O número $(10110,011)_2$, onde a parte inteira do número é 10110 e a parte fracionária do número é 011.

Para converter o número na base 10 aplicamos para a parte inteira a expressão de transformação conhecida e da mesma forma para a parte fracionária, não esquecendo que na parte fracionária a base é elevada a potência negativa.

Exemplo : Para o número $(1001),10101)_2$ será convertido como :

$N = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5}$

N = 9,65625

Observar que a conversão do número fracionário para a base 10 será sempre uma aproximação sucessiva do número, pois um número pode ser representado uma dizima periódica, como ,101 101 101....101.

Exemplo: Converter o número $(321,221)_4$ para a base 10.

$N = 3 \times 4^2 + 2 \times 4^1 + 1 \times 4^0 + 2 \times 4^{-1} + 2 \times 4^{-2} + 1 \times 4^{-3} = 57,640625$

Exemplo: Converter o número $(753,642)_8$ para a base 10.
$N = 7 \times 8^2 + 5 \times 8^1 + 3 \times 8^0 + 6 \times 8^{-1} + 4 \times 8^{-2} + 2 \times 8^{-3} = 491,81640625$

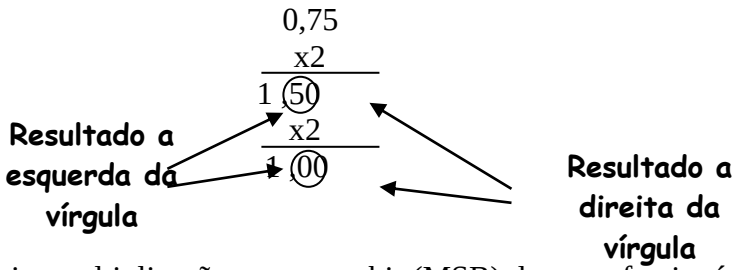
CONVERSÃO DE UM NÚMERO FRACIONÁRIO NA BASE 10 PARA UMA BASE QUALQUER

A conversão de um número na base 10 para uma outra base é realizado para números inteiros como uma divisão pela base a ser convertida. No caso de números fracionários o processo deve ser o mesmo divisão sucessiva pela base, porém como a base é 1/B, a divisão por 1/B se torna uma multiplicação. O exemplo a seguir mostra uma conversão do número $N = (6,75)_{10}$ para a base 2.

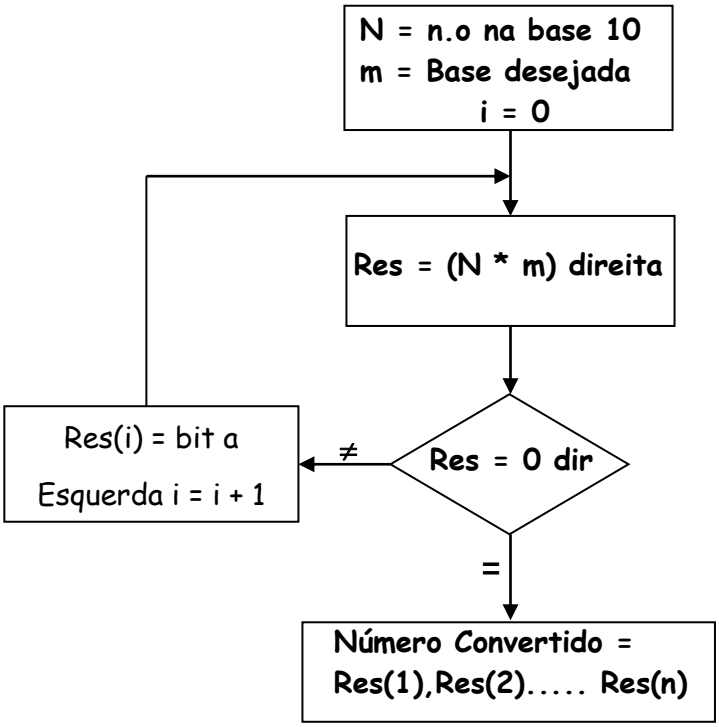
A parte inteira do número N = 6, convertida para binário será : 110
A parte fracionária dp número N = ,75 será convertida como :

1) Multiplica-se a parte fracionária pela base que se quer converter.

2) O resultado da multiplicação a esquerda da vírgula item 3 ou 4;
3) Se for menor do que a base é 0 e
4) Se for maior do que a base é 1 e
5) Continuar a multiplicação com o número a dreita da vírgula só considerando a parte fracionária até que o resultado a direita da vírgula seja igual a 0 se não
6) Volte ao item 2.



A primeira multiplicação gerou um bit (MSB) da parte fracionária o bit =1 e o resultado a direita após a virgula = 50. Continuando a multiplicação será gerado o segundo bit = 1 e o resultado a direita é igual a zero, o que encerra a conversão. Assim o número convertido será: $(110,11)_2$.



Exemplo: Converter o número $(1,2)_{10}$ para a base 3.

A parte inteira é 1 convertida para a base 3, são necessários 2 bits para representação, assim = 01. Para a parte fracionária ,2 as multiplicações sucessivas convertem o número obedecendo ao procedimento.

,2	,6	,8	,4
x3	x3	x3	x3
0,6	1,8	2,4	1,2

O número convertido N = 1,121121...121... uma dízima periódica.

Exemplo: Converter o número $(7,643)_{10}$ para a base 8.

A parte inteira é 7 convertida para a base 8, são necessários 3 bits para representação, assim = 111. Para a parte fracionária ,643 as multiplicações sucessivas convertem o número obedecendo ao procedimento.

```
    ,7     ,6      ,8     ,4      ,3
    x8     x8      x8     x8      x8
  _____  _____  _____  _____  _____
   5,6    4,8     6,4    3,2     2,4
```

O número convertido N = 7,546323232...32... uma dízima.

CONVERSÃO ENTRE BASES MÚLTIPLAS DE POTÊNCIA DE 2 PARA NÚMEROS FRACIONÁRIOS

A conversão entre bases múltiplas de potência de 2 para os números fracionários, como na parte inteira, não necessita de multiplicação pois a conversão é direta.

Exemplo: Converter o número na base 4 $(2,321)_4$ para a base 2.
Inicialmente codifique os coeficientes em binário como a seguir. Como na base 4 são 02 bits para cada dígito, a representação fica :

```
 2   3   2   1→ base 4
10, 11  10  01
```

A conversão é direta, resultando o número na base 2 $(10,111001)_2$.

Exemplo: Converter o número na base 4 $(2,321)_4$ para a base 8.

```
 2,  3    2    1→ base 4
10, 11  10  01, para a base 8 são 03 bits necessários para a representação, assim o número codificado,
```

fica :

```
 2    7    1 → base 8
010,  111  001
```

O número na base 8, será 2,71

Representação do algoritmo de conversão por uma tabela de fluxo.

O modelo de descrição do sistema para conversão de números na base decimal para a base dois é feito através de uma tabela de fluxo apresentada a seguir.
A tabela possui entrada dos números de 0 a 9 portanto 10 colunas. A conversão é para a base 2 portanto 2 estados internos, assim a tabela é de 10 x 2.

Dec.	0	1	2	3	4	5	6	7	8	9
Est. 0	0,0	1,0	0,1	1,1	0,2	1,2	0,3	1,3	0,4	1,4
1	0,5	1,5	0,6	1,6	0,7	1,7	0,8	1,8	0,9	1,9

c) Para mostrar o funcionamento da tabela acima, vamos simular o número $(151)_{10}$.

Entrada	Estado	Saída
1	1	0
5	1	7
1	1	5

Realimentando o quociente $(75)_{10}$ para a entrada

Entrada	Estado	Saída
7	1	3
5	1	7

Realimentando o quociente $(37)_{10}$ para a entrada

Entrada	Estado	Saída
3	1	1
7	1	8

Realimentando o quociente $(18)_{10}$ para a entrada

Entrada	Estado	Saída
1	1	0
8	0	9

Realimentando o quociente $(9)_{10}$ para a entrada

Entrada	Estado	Saída
9	1	4

Realimentando o quociente $(4)_{10}$ para a entrada

Entrada	Estado	Saída
4	0	2

Realimentando o quociente $(2)_{10}$ para a entrada

Entrada	Estado	Saída
2	0	1

Realimentando o quociente $(1)_{10}$ para a entrada

Entrada	Estado	Saída
1	1	0

O resultado é lido no estado assim o número convertido é $(10010111)_2$
Conversão de números inteiros na base 10 para a base 16.

base 10 para a base 16.

Hex.	0	1	2	3	4	5	6	7	8	9
Est. 0	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0
1	A,0	B,0	C,0	D,0	E,0	F,0	0,1	1,1	2,1	3,1
2	4,1	5,1	6,1	7,1	8,1	9,1	A,1	B,1	C,1	D,1
3	E,1	F,1	0,2	1.2	2,2	3,2	4,2	5,2	6,2	7,2
4	8,2	9,2	A,2	B,2	C,2	D,2	E,2	F,2	0,3	1.3

5	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	A,3	B.3
6	C,3	D,3	E,3	F,3	0,4	1,4	2,4	3,4	4,4	5,4
7	6,4	7,4	8,4	9,4	A,4	B.4	C,4	D,4	E,4	F,4
8	0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5
9	A,5	B,5	C,5	D,5	E,5	F,5	0,6	1,6	2,6	3,6
A	4,6	5,6	6,6	7.6	8.6	9,6	A,6	B,6	C,6	D,6
B	E,6	F,6	0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7
C	8,7	9,7	A,7	B,7	C,7	D,7	E,7	F,7	0,8	1,8
D	2,8	3,8	4,8	5,8	6,8	7,8	8,8	9,8	A,8	B,8
E	C,8	D,8	E,8	F,8	0,9	1,9	2,9	3,9	4,9	5,9
F	6,9	7,9	8.9	9,9	A,9	B,9	C,9	D,9	E,9	F,9

Conversão da base 16 para a base 10.

Dec.	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Est.0	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0	0,1	1,1	2,1	3,1	4,1	5,1
1	6,1	7,1	8,1	9,1	0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2	0,3	1,3
2	2,3	3,3	4,3	5,3	6,3	7,3	8,3	9,3	0,4	1,4	2,4	3,4	4,4	5,4	6,4	7,4
3	8,4	9,4	0,5	1,5	2,5	3,5	4,5	5,5	6,5	7,5	8,5	9,5	0,6	1,6	2,6	3,6
4	4,6	5,6	6,6	7,6	8,6	9,6	0,7	1,7	2,7	3,7	4,7	5,7	6,7	7,7	8,7	9,7
5	0,8	1,8	2,8	3,8	4,8	5,8	6,8	7,8	8,8	9,8	0,9	1,9	2,9	3,9	4,9	5,9
6	6,9	7,9	8,9	9,9	0,A	1,A	2,A	3,A	4,A	5,A	6,A	7,A	8,A	9,A	0,B	1,B
7	2,B	3,B	4,B	5,B	6,B	7,B	8,B	9,B	0,C	1,C	2,C	3,C	4,C	5,C	6,C	7,C
8	8,C	9,C	0,D	1,D	2,D	3,D	4,D	5,D	6,D	7,D	8,D	9,D	0,E	1,E	2,E	3,E
9	4,E	5.E	6,E	7,E	8,E	9,E	0,F	1,F	2,F	3,F	4,F	5,F	6,F	7,F	8,F	9,F

c) Conversão de números decimais fracionários para binário.

Dec.	0	1	2	3	4	5	6	7	8	9
Est. 0	0,0	0,2	0,4	0,6	0,8	1,0	1,2	1,4	1,6	1,8
1	0,1	0,3	0,5	0,7	0,9	1,1	1,3	1,5	1,7	1,9

Como exemplo, converter o número decimal fracionário : $(,5275)_{10}$

d) Simulação do número.

0,5275	0,550	0,100	0,200	0,400	0,800
x2	x2	x2	x2	x2	x2
1,0550	1,100	0,200	0,400	0,800	1,600

Entrada	Estado	Saída
5	1	0
7	1	5
2	0	5
5	1	0

Realimentando o quociente $(550)_{10}$ para a entrada

Entrada	Estado	Saída
5	1	0
5	1	0
0	1	1

Realimentando o quociente $(1)_{10}$ para a entrada

Entrada	Estado	Saída
1	0	2

Realimentando o quociente $(2)_{10}$ para a entrada

Entrada	Estado	Saída
2	0	4

Realimentando o quociente $(4)_{10}$ para a entrada

Entrada	Estado	Saída
4	0	8

Realimentando o quociente $(8)_{10}$ para a entrada

Entrada	Estado	Saída
8	1	6

Realimentando o quociente $(6)_{10}$ para a entrada

Entrada	Estado	Saída
6	1	2 (Dízima)

O número convertido decimal para a base 2 fracionário será : $(,110001)_2$.

Outras conversões de base para números fracionários podem ser realizados conforme modelo descrito anteriormente.

REPRESENTAÇÃO DOS NÚMEROS ASSINALADOS

1. Representação em sinal e amplitude

Esta representação usa um dígito (normalmente o bit/dígito mais significativo) para indicar o sinal do número onde o símbolo (+) indica que o sinal do número é positivo e o (-) indica negativo.

Faixa de representação do número de n dígitos $[-(B^{n-1} – 1), +(B^{n-1} – 1)]$ ou $[-(B^{n-1} – 1), (B^{n-1} – 1)]$.

Para representar um número N = 4 bits na base 2, a faixa de representação será: -7 -6 -6 -4 -3 -2 -1 -0 +0 +1 + 2 + 3 + 4 + 5 + 6 +7.

Para um número N = 2 digitos na base 10, a faixa de representação será: -9 -8 .....-0 +0 .....+9.

Cálculo do valor de um número – Um número em sinal e amplitude, independente da base utilizada é formado por 2 parcelas como a seguir: Sendo a um número então a parcela da esquerda representa o sinal S(a) e a parcela da direita M (a) a amplitude.

a = S(a)M(a), onde S(a) é igual + ou – (0 ou 1) e $M(a) = \Sigma_{n-2} A_i B^i$

Obs.: Devido do duplo zero essa representação é pouco utilizada na aritmética dos computadores e outras representações são utilizadas para os números. As operações de soma e subtração deve verificar o sinal dos operandos, o que torna muito complicado essas operações manipuladas pelo computador;

Exemplo: Soma de 2 números assinalados (-7) + (+7)

```
  1111                    0111
  0111 -                  0111 -
 ------                  ------
  1000 -> resultado (-0)  0000 -> resultado (+0)
```

2. Representação dos números em complemento B-1.

Essa forma de representação dos números é utilizada, pois as operações de soma e subtração são realizadas sem a preocupação com os sinais dos operandos. Os números positivos são representados normalmente e os números negativos representados em complemento.

Como obter o número N em complemento B-1.
O número N em complemento B-1 é obtido subtraindo esse número N da maior quantidade representativa, ou seja, $N' = B^{n-1} - N$.

Exemplo: O complemento do número N = 5, na base 10.

$N' = (10 - 1) - 5 = 4$

Exemplo: O complemento de N = 0101, na base 2.

$N = (16 - 1) - 5 = 1010$

Exemplo: Para um número de 3bits: +3, +2, +1, +0, -0, -1, -2, -3.

3. Representação dos números em complementos B-2

Essa forma de representação dos números é muito utilizada, pois -0 não tem significado.
Como obter o número em complemento B-2.

O número negativo aparece em complemento de dois e é obtido fazendo o complemento B-1 e somar uma unidade.

A faixa de valores em complemento de dois para 4 bits, será: $-2^n \leq N \leq 2^n - 1$

N = Um número de n igual a três bits na base dois, sendo 1bit (MSB) para o sinal e 2 bits para a amplitude.

-4, -3, -2, -1, 0, +1, +2, +3

4. Representação em excesso 4

Essa forma de representação em excesso 4 é utilizada no representação do expoente do n úmero em ponto flutuante.

Como obter o número em excesso 4

Somando ao número sem sinal o valor 4. Exemplo: N = 3 em excesso 4 será 7.

Representação dos números nas várias representações para números inteiros de 3 bits.

Decimal	Sem sinal	Sinal e Amplitude	Complemento de um	Complemento de dois	Excesso quatro
7	111	-	-	-	-
6	110	-	-	-	-
5	101	-	-	-	-
4	100	-	-	-	-
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
+0	000	000	000	000	100
-0	-	100	111	000	100
-1	-	101	110	111	001
-2	-	110	101	110	010
-3	-	111	100	101	001
-4	-	-	-	100	000

REPRESENTAÇÃO DOS NÚMEROS EM PONTO FLUTUANTE

Os números reais representando os dados numéricos podem ser fracionários (tais como 14,212). Como é representar a parte fracionária do número, representada (após a vírgula que é a separação da parte inteira), de forma que permita no computador o seu armazenamento e também ocupe um pequeno volume de memória?

NÚMEROS REAIS

Os números reais são números que possuem parte inteira e fracionária (por exemplo, 14,212). O formato de representação desses números da parte inteira, vírgula (ou ponto) e da parte fracionária, é apresentado a seguir:
Por exemplo: 123, 012, sendo 123 a parte inteira e 012 a parte fracionária do número.

Nem toda a notação que serve para representar os números é adequada para um processamento no computador. É necessária uma representação dos números, de tal forma que os números possam ser expressos sem uma perda significativa de precisão e de intervalo numérico.

REPRESENTAÇÃO EM PONTO FLUTUANTE

Consideremos o número 14,212 do exemplo. Este número pode ser também expresso como 14,212 x

10^0 e também ser expresso como 14212 x 10^{-3} ou ainda 0,14212 x 10^2. Na realidade, qualquer número, inteiro ou fracionário, pode ser expresso neste formato **número x base** $^{\textbf{expoente}}$, no qual se variam dois parâmetros: a posição da vírgula (que delimita a parte fracionária) e a potência à qual se eleva na base. Essa representação é denominada **representação em ponto flutuante**, pois o ponto varia sua posição, modificando, em consequência, o valor representado.

REPRESENTAÇÃO NORMALIZADA

Na representação normalizada, o número é construído movendo a vírgula para a direita ou para a esquerda de tal forma que o número seja menor do que 1, o mais próximo possível de 1, multiplicado pela potência da base de forma a manter o valor próprio do número. Em geral, isso significa que o primeiro dígito significativo seguirá imediatamente ao ponto (ou vírgula).

Por exemplo:
$14{,}212_{10}$ --> normalizando ==> 0,14212 x 10^2
$0{,}0003456_{10}$ --> normalizando ==> 0,3456 x 10^{-3}
$0{,}00001100_2$ --> normalizando ==> 0,1100 x 2^{-4}
De forma genérica, podemos representar a forma normalizada:
± número x base $^{\pm\text{expoente}}$
A parte do número representado dessa forma normalizada (os algarismos significativos), damos o nome de mantissa.e portanto podemos representar:

± 0,M x B ± e

onde ***M*** é a mantissa, ***B*** é a base e ***e*** é o expoente.

REPRESENTAÇÃO DE NÚMEROS REAIS NO COMPUTADOR

Uma forma comum de representação de números reais no computador pode ser expressa como segue:

1 BIT	X BITS	Y BITS
SINAL DO NÚMERO	EXPOENTE DO NÚMERO	MANTISSA

Em virtude de os computadores operarem na base 2 pois os valores são “0 e 1”, não será necessário na representação dos números reais, especificar a base pois se subentende que a base é **implícita**.

O número total ***M*** de bits para a representação, depende do tipo de computador, sendo distribuídos em : ***1*** bit para o sinal do número positivo ou negativo ***1*** bit para o sinal do expoente, ***y bits*** para a mantissa e ***x bits*** para o expoente.
O número de bits do expoente, define a intervalo numérico que o computador pode representar, assim quanto maior o número de bits, maior será o intervalo de representação do número. Para a mantissa quanto maior o número de bits para a mantissa, maior será a precisão na representação do número. Uma vez que no processo de conversão de decimal para binário, a conversão é realizada na parte fracionária do número através de uma aproximação e assim quanto maior o número de bits mais próximo ao valor real do número. Porém, reduzindo-se a mantissa, perde-se em precisão pois estaremos truncando o número (o erro de truncamento é quando limitamos o número em um certo número de algarismo, ou seja, cortando os últimos algarismos significativos que não podem ser representados).

Considerando-se a representação acima, na base implícita 2:

maior expoente possível ***e*** = $2^x - 1$, para x bits, tem-se $0 \leq e \leq 2^x - 1$
maior mantissa possível = $2^y - 1$, para y bits, tem-se $0 \leq e \leq 2^y - 1$

a) Intervalo de representação dos números positivos

Para N é representado de $0{,}100.000 \times 2^{-e} \leq N \leq + 0{,}111....111 \times 2^e$

Sendo:
maior número real : $+(0.111...1 \times 2^e)$ sendo $e = 2^x - 1$
menor real positivo: $+(0.100...0 \times 2^{-e}$ sendo $e = 2^x - 1$

b) Intervalo de representação dos números negativos
Para N é representado de $- 0{,}100.000 \times 2^{-e} \leq N \leq - 0{,}111....111 \times 2^e$, sendo:
maior real negativo: $-(0.100...0 \times 2^{-e}$ sendo $e = 2^x - 1$
menor número real: $-(0.111...1 \times 2^e)$ sendo $e = 2^x - 1$

Overflow ocorre quando o valor absoluto do dado a ser representado excede a capacidade de representação, porque o número de bits do expoente (neste caso, positivo) é insuficiente para representar o dado.
Um outro problema ocorre na região de números próximos de zero, que tem o maior expoente negativo possível.

Underflow ocorre quando o valor absoluto do dado a ser representado é tão pequeno que fica menor que o menor valor absoluto representável. Quando os números são muito próximos a zero sendo o expoente negativo, não há representação desses números e ocorre uma descontinuidade na representação.

Observação: Não se deve confundir o arredondamento para a normalização do número a ser representado ocorrendo imprecisão com underflow o qual ocore quando os dados na faixa do **underflow** não podem ser representados em virtude do estouro do expoente.

CARACTERÍSTICA - A característica é o expoente, representado na forma de **excesso de *n***, ou seja,
CARACTERÍSTICA = EXPOENTE + EXCESSO
A representação substituindo expoente por característica implica que todas as características serão positivas, de forma que é possível eliminar a representação do sinal do expoente.

REPRESENTAÇÃO DO EXPOENTE EM CARACTERÍSTICA

Com 7 bits:	CARACTERÍSTICA	0	1	2	64	126	127	
	EXPOENTE $M-1$	-64	-63	-62	0.....	+62	+63	$M-1$
Com M bits:	CARACTERÍSTICA	0	1	2	.....		2^M-1	
	EXPOENTE	-2^{M-1}	.....		.....		$2^{M-1}-1$	

Se CARACTERÍSTICA = EXPOENTE + EXCESSO,

sendo M o número de bits para a representação da característica, temos:

$0 = - 2^{M-1} + EXCESSO \Rightarrow EXCESSO = + 2^{M-1}$

EXEMPLO DE REPRESENTAÇÃO EM PONTO FLUTUANTE

Representação no IBM /370
A família IBM /370 representa os dados em ponto flutuante com base implícita = 16, no seguinte formato:

REPRESENTAÇÃO NO IBM /370

1 bit	7 bits	24 bits	base = 16
SN	CARACTERÍSTICA	MANTISSA	

Sendo:
SN = sinal do dado
CARACTERÍSTICA = o expoente, representado na forma de excesso de n, ou seja,
CARACTERÍSTICA = EXPOENTE + EXCESSO

No caso da IBM, o excesso é de 64_{10}, portanto: CARACTERÍSTICA = EXPOENTE + 64_{10}
Exemplificando: expoente = 8_{10}, logo característica = $8_{10} + 64_{10} = 72_{10}$
Assim, uma característica entre 0 e 63_{10} significa que o expoente é negativo, enquanto uma característica entre 65 e 127 significa que o expoente é positivo (característica igual a 64_{10} significa expoente igual a 0)..

Exemplo: Representar $(25,5)_{10}$

Como a base implícita é 16, vamos converter para hexadecimal: 25 / 16 = 1, resto 9 logo: $(25)_{10} = (19)_{16}$

Parte fracionária: 0.5 x 16 = 8,0

Logo: $(25,5)_{10} = (19,8)_{16} \times 16^0$

Normalizando: $(19,8)_{16} \times 16^0 = (0,198)_{16} \times 16^2$. Em binário com 24 bits, a mantissa normalizada será:

$0,198_{16}$ = 0001.1001.1000.0000.0000.0000

Obs.: Como o número $0,198_{16}$ será representado em 24 bits, os bits não representativos (à direita) serão preenchidos com zeros.

Como o expoente é 2, a característica será: $2_{10} + 64_{10} = 66_{10}$. Em binário com 7 bits, será: 100.0010
Portanto, a representação será:

REPRESENTAÇÃO NO IBM /370 - EXEMPLO

SN	CARACT.	MANTISSA
0	1000010	00011001100000000000000
	64 + 2	1 . 9 . 8 .

Padrão IEEE 754

Existem muitas maneiras de se representar números em ponto flutuante e para garantir a portabilidade a norma foi escrita para números binários.

sinal 1 bit	Expoente 8bits	Mantissa 23 bits

Existem 2 formatos relativos a precisão: simples para 32 bits e dupla para 64 bits. A seguir são mostrados os formatos:

32bits – Simples precisão

sinal 1 bit	Expoente 8bits	Mantissa 23 bits

64bits – Dupla precisão

sinal 1 bit	Expoente 11bits	Mantissa 52 bits

Obs: O menor expoente representável é -126. A fração representa a magnitude do numero em vez da mantissa.
NaN – Não é um número, exemplo raiz de (-1) = NaN.
Exemplo: Para os valores a seguir encontrar o padrão de bits simples precisão

Valor	Sinal	Expoente	Fração
$+1{,}101 \times 2^{5}$	0	1000 0100	101 0000 0000 0000 0000 0000
$-1{,}01011 \times 2^{-126}$	1	0000 0001	010 1100 0000 0000 0000 0000
$+1{,}0 \times 2^{127}$	0	1111 1110	000 0000 0000 0000 0000 0000
+0	0	0000 0000	000 0000 0000 0000 0000 0000
-0	1	0000 0000	000 0000 0000 0000 0000 0000
∞	0	1111 1111	000 0000 0000 0000 0000 0000
NaN	0	1111 1111	101 0000 0000 0000 0000 0000
$+2^{-128}$	0	0000 0000	010 0000 0000 0000 0000 0000

ARITMÉTICA EM COMPUTADORES

As quatro operações aritméticas: soma, subtração, multiplicação e divisão serão apresentadas considerando a representação dos números. Serão estudados alguns desses subsistemas digitais para a realização dessas operações, bem como algoritmos para a realização dessas operações. Será apresentado um somador de alto desempenho.

Aritmética de números inteiros não assinalados.

1. Soma de 2 números A = 0111 e B = 0100

1
0111
0100 +
1011

2. Subtração de 2 números A = 1100 e B = 0101

0 ^{1}0 1 ^{1}0
0 1 0 1 -
0 1 1 1

Aritmética de números assinalados de 4 bits em complemento de dois

1. Soma de 2 números A = + 3 e B = + 2

A = 0011 e B = 0010

0011
0010 +
0101

O resultado não ocorreu overflow OV = = $c_3 \oplus c_4$, onde $c_3 = 0$ e $c_4 = 0$ e OV = 0

2. Soma de 2 números A = + 6 e B = + 4

0110
0100 +
1010
O resultado ocorreu overflow OV = $c_3 \oplus c_4$, onde $c_3 = 1$ e $c_4 = 0$ e OV = 1

3. Subtração de 2 números A = +7 e B = + 4
0111
0100
0011

4. Soma de 2 números A = -3 e B = - 2

Em complemento de 2 A = 1101 e B = 1110

1101
1110+
1011
O resultado está em complemento de 2 S = -5 e OV = 0

Despreza-se c_4

5. Soma de 2 números A = -6 e B = - 4

1010
1100+
0110
O resultado está em complemento de 2 S = -10 e OV = 1 ($c_3 = 0$ e $c_4 = 1$).

6. Subtração de 2 números A = -7 e B = - 4

1001
1100-
1101
O resultado está em complemento de 2 S = - 3 OV = 0

7. Subtração de 2 números A = +5 e B = -7

0101
1001-
1100
O resultado está em complemento de 2 S = +12 e OV = 1

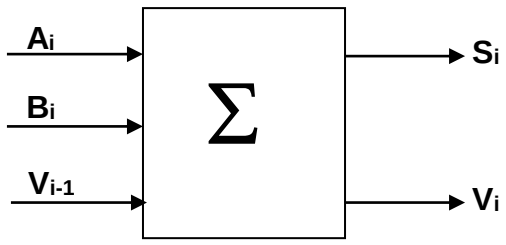
Exercícios:

Computar para 4 bits soma e subtração usando representação em complemento de dois.

1. A = + 5 e B = +10 base 2
2. A = - 5 e B = -10 base 2
3. A = + 7 e B = -7
4. A = - 7 e B = +7

Tipos de somadores

Somador completo



c) Tabela da Verdade

A	B	C_0	S	C_1
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	1	1

d) Mapas de Karnaughs

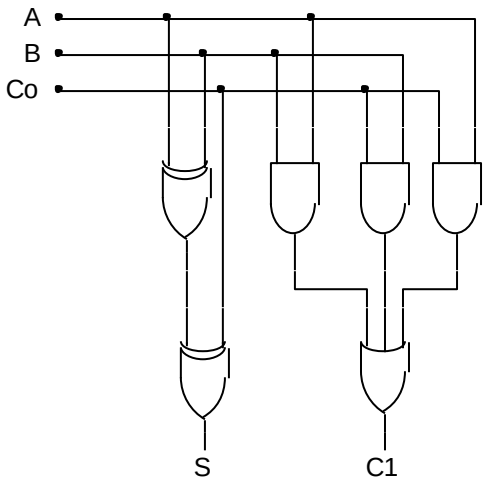
AB	00	01	11	10
C_0 0	0	1	0	1
1	1	0	1	0

$S = A \oplus B \oplus C_0$

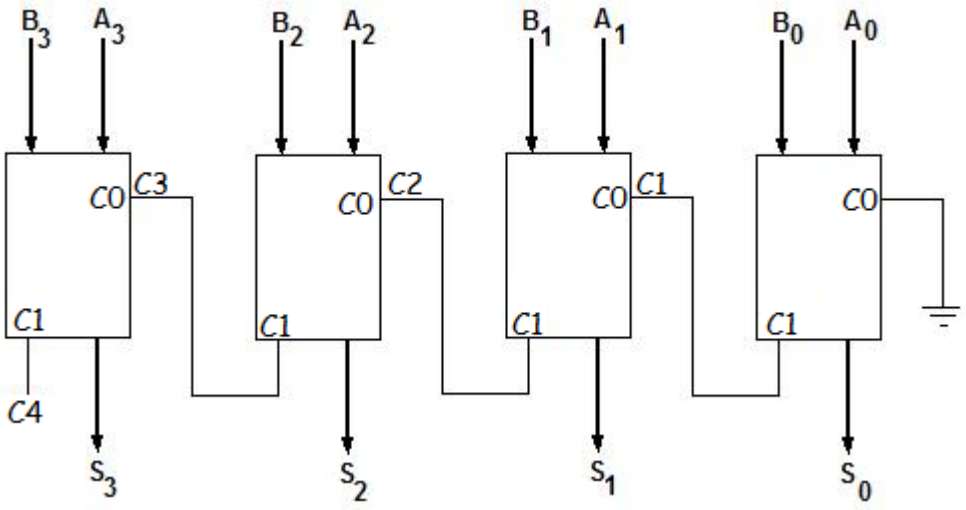
AB	00	01	11	10
C_0 0	0	0	1	0
1	0	1	1	1

$C_1 = AB + C_0(A + B)$

Circuito aritmético das funções booleanas do somador completo



Somadores para 4 bits, onde A = $A_3A_2A_1A_0$ e B = $B_3B_2B_1B_0$ e C_0 = 0.



Somadores mais rápidos.

Um somador completo precisa de uma lógica a qual consome um tempo de propagação do Vai Um, além desse bit se propagado para outras unidades MSB, para que o resultado da soma fique correto.

Caminho crítico – é o caminho mais longo do circuito.

Um somador completo é uma lógica de dois níveis portanto, dois atrasos das portas. Se considerar um somador de quatro bits com propagação de Vai Um, o número total de atrasos será:

Atraso final = n bits do somador x atraso do somador completo =

Para um somador de trinta e dois bits, o atraso final =

Somador com lógica de dois níveis.

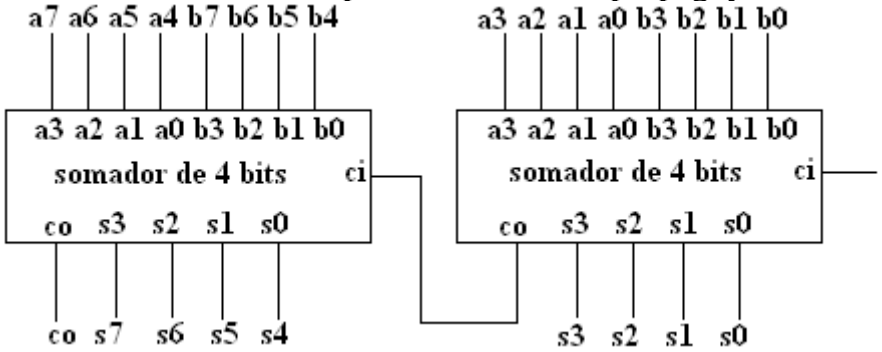
O atraso dessa unidade será de duas portas (dois atrasos).

Projetar um meio somador com lógica de dois níveis para 2 números a e b de 2 bits cada. Sendo a = a_1a_0 e b = b_1b_0 e s a saída s = s_1s_0 e c_0 saída do bit de transporte.

a	a_1	a_0	b	b_1	b_0	c_{in}	s	s_1	s_0	c_0
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	0	1	0
0	0	0	1	0	1	0	1	0	1	0
0	0	0	1	0	1	1	2	1	0	0
0	0	0	2	1	0	0	2	1	0	0
0	0	0	2	1	0	1	3	1	1	0
0	0	0	3	1	1	0	3	1	1	0
0	0	0	3	1	1	1	10	0	0	1
1	0	1	0	0	0	0	1	0	1	0
1	0	1	0	0	0	1	2	1	0	0
1	0	1	1	0	1	0	2	1	0	0
1	0	1	1	0	1	1	3	1	1	0
1	0	1	2	1	0	0	3	1	1	0
1	0	1	2	1	0	1	10	0	0	1
1	0	1	3	1	1	0	10	0	0	1
1	0	1	3	1	1	1	11	0	1	1

a	a_1	a_0	b	b_1	b_0	c_{in}	s	s_1	s_0	c_0
2	1	0	0	0	0	0	2	1	0	0
2	1	0	0	0	0	1	3	1	1	0
2	1	0	1	0	1	0	3	1	1	0
2	1	0	1	0	1	1	10	0	0	1
2	1	0	2	1	0	0	10	0	0	1
2	1	0	2	1	0	1	11	0	1	1
2	1	0	3	1	1	0	11	0	1	1
2	1	0	3	1	1	1	12	1	0	1
3	1	1	0	0	0	0	3	1	1	0
3	1	1	0	0	0	1	10	0	0	1
3	1	1	1	0	1	0	10	0	0	1
3	1	1	1	0	1	1	11	0	1	1
3	1	1	2	1	0	0	11	0	1	1
3	1	1	2	1	0	1	12	1	0	1
3	1	1	3	1	1	0	12	1	0	1
3	1	1	3	1	1	1	13	1	1	1

Somador de 32 bits usando somadores rápidos de 4 bits com propagação do Vai Um.



A unidade somadora de 4 bits menos significativa com 2 níveis de atraso, para a soma e o transporte Vai Um. Para o somador MSB o atraso do transporte será:
Atraso Somador LSB + atraso do Somador MSB =

Um somador de 8 bits consome então atrasos e para Somador 32 bits = atrasos.

SOMADOR DE ALTO DESEMPENHO

Um esquema eficiente de antecipação do bit de transporte;
Conceito: Considere um somador completo, as equações de S e C serão:

a_i	b_i	c_0	s	c_1
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 equação de S na forma canônica será:
$S = a_i'b_i'c_i + a_i'b_ic_i' + a_ib_i'c_i' + a_ib_ic_i$

A equação do Vai Um na forma canônica será:
$C_1 = a_i'b_ic_i + a_ib_i'c_i + a_ib_ic_i' + a_ib_ic_i$

Para somador de quatro bits podem-se escrever as equações dos "Vai Um".

c1 =

c2 =

c3 =

c4 =

Dedução da lógica de antecipação dos bits de transportes.

Chamando-se de $G_i = a_ib_i$ e $P_i = (a_i \oplus b_i)$.

G_0 = P_0 =
G_1 = P_1 =
G_2 = P_2 =
G_3 = P_3 =

Manipulando-se as equações dos transportes, temos:

C_1 =

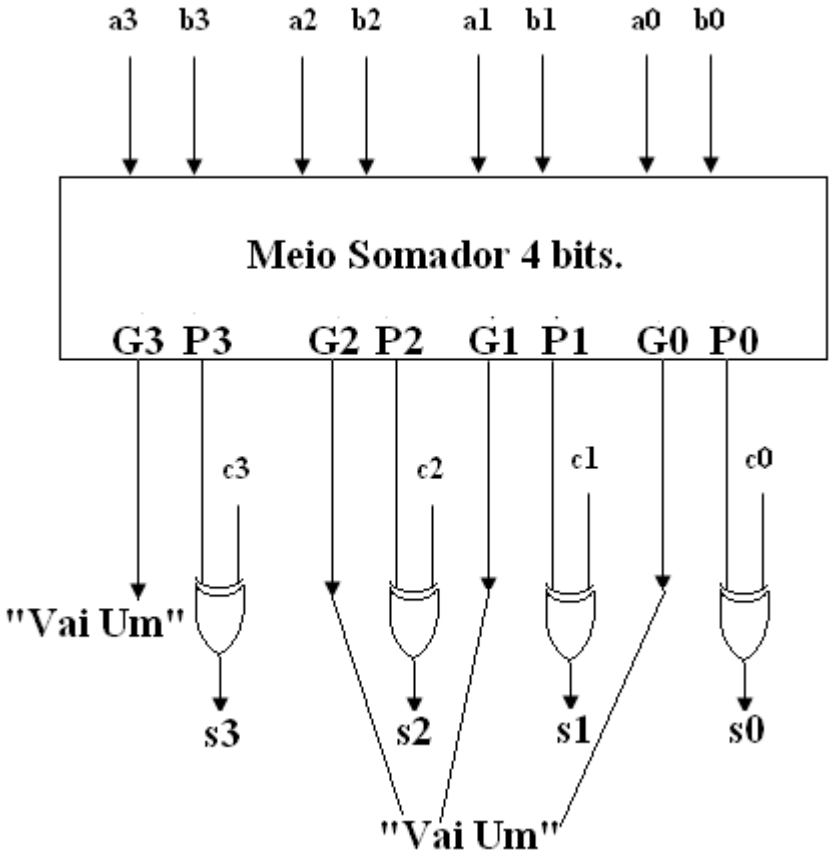
C_2 =

C_2 =

C_3 =
C_3 =

C_4 =

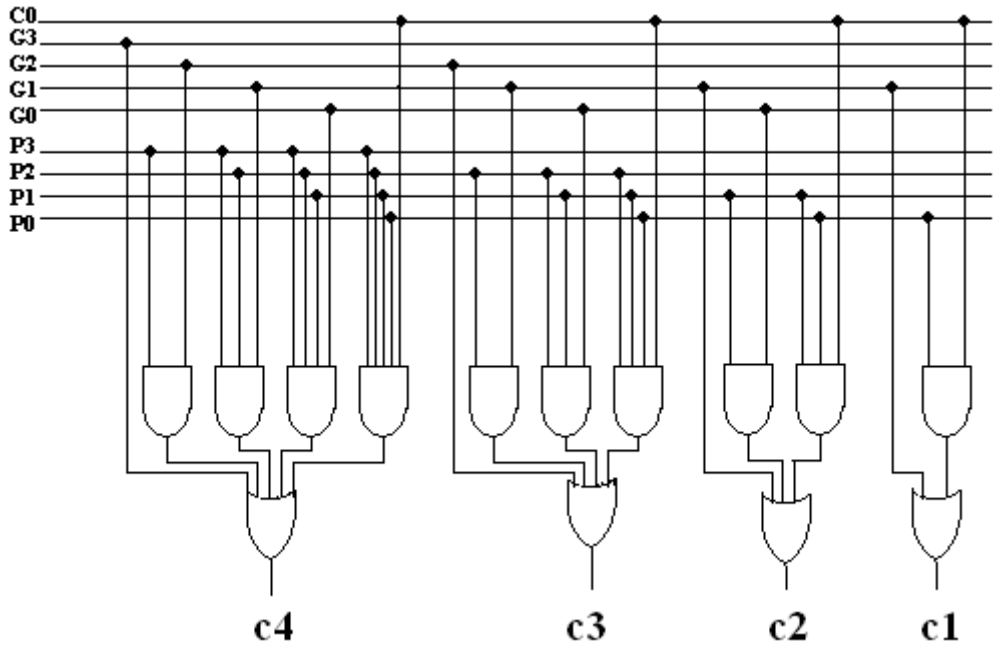
O circuito para geração de G_i e P_i é feito usando meio somador.



Número de Portas.

$G_i = a_i . b_i$ = 1 porta.
$P_i = a_i \oplus b_i$ = 2 portas.
$S_i = P_i \oplus c_i$ = 1 porta.
Total = 4 portas.

Circuito lógico para geração do bit de transporte c4.



Número de Portas

$$= \sum_{i=1}^{N} i + 1$$

Onde N é o n.o bits.

Cálculo do número de portas do somador com a lógica de antecipação do Vai Um.

Meio Somador = 2 portas (E e XOR)
Vai Um = 1 porta (XOR)

Total de portas do somador = 4 x 3 = 12 portas.
Lógica de Antecipação = 14 portas.
Total = 26 portas.

Cálculo do atraso do somador com a lógica de antecipação do Vai Um.

Meio Somador = 1 porta
Lógica de Antecipação = 2 portas
Somador = 1 porta.
Total = 4 portas.

Exemplo: Calcular o total de portas para um Somador com antecipação do Vai Um para N bits e calcular o atraso desse somador.

Meio Somador = N x 3 portas.

Lógica de antecipação - $\text{num.portas} = \sum_{i=1}^{N} i + 1$ portas

Total = N x 3 + $\sum_{i=1}^{N} i + 1$ portas.

O atraso do transporte será igual a 4 (independente do número de bits do Somador).

Exemplo: Calcular o total de portas para um Somador com antecipação do Vai Um para 8 bits e calcular o atraso desse somador.

Exemplo: Calcular o total de portas para um Somador com antecipação do Vai Um para 16 bits e calcular o atraso desse somador.

Exemplo: Calcular o total de portas para um Somador com antecipação do Vai Um para 32 bits e calcular o atraso desse somador.

Aritmética de números reais assinalados em ponto fixo.

Exemplo: A = +0,75 e B = - 0,625 representação em complemento de 2.

A = 0000.1100 e B = 0000.1010 (amplitude) em complemento de 2 B = 1111.0110
0000.1100
1111.0110+
0000.0010

O resultado é S = +0,125

Aritmética de números em ponto flutuante

Números em ponto flutuante de 32 bits, sendo 1 bit para o sinal, 8 bits para o expoente e 23 bits para a mantissa.

Exemplo: N = $(228)_{10}$

N = $(11100100)_2 = 0.111001 \times 2^8$

S	Expoente	mantissa
0	00001000	111 0010 0000 0000 0000 0000

Usando para o expoente Excesso – n, nesse caso para números de 32 bits usamos excesso 128. A faixa vai de -128 a + 127, então fica

S	Expoente	mantissa
0	10000111	111 0010 0000 0000 0000 0000

1. Somar 2 números em ponto flutuante como A = 4,625 e B = 2,75 para 10 bits, sendo 1 bit para sinal. 3 bits para o expoente Excesso 4 e 6bits para a mantissa.

a. Normalizando os números

A = 100,101000 e B = 010,110000 A = 0,100 101 x 2^3 e B = 010 110 x 2^3

0	111	100101

+

0	111	010110

Os expoentes são os mesmos então soma-se as mantissas de A e B

100101
010110+
111011

O resultado é + 0,111011 x 2^3, então: 7,375.

2. Somar 2 números A = +12,5 e B = +0,25 usar excesso 8

Normalizando-se os números

A = 0,125 x 10^2 e B = 0,0025 x 10^2 e B = 0,0001 x 2^2

Como os expoentes são os mesmos podemos somar diretamente as mantissas.

0,1250
0,0025+
0,1275

0 resultado é S = 0,1275 x 10^2 = 12,75

A = 12,5 = 1100,1 e B = 0000,0100 normalizando A = 0,11001 x 2^4 e B = 0,000001 x 2^4

0	1100	1100 1000
0	1100	0000 0100

Somando-se as mantissas, temos:

S = .11001100 x 2^4 = 12,75.

3. Somar 2 números A = 14,75 e B = - 12,25

Para o exercício é a realização de A – B sendo A e B em complemento de 2, temos:

A = 0,1475 x 10^2 e B = - 0,1275 x 10^2.

Transformando os números em binário, temos:

A = 1110,11 e B = 1100,01

Normalizando-se os números temos:

A = 0,111011 x 2^4 e B = -0,110001 x 2^4

B em complemento de 2

B = 001111 em complemento de 2

0	1100	111011
1	1100	001111

Somando-se os números, temos:

S = 001010 x 2^4 = 2,50

Erros na representação em ponto flutuante

É inevitável que a representação em ponto flutuante introduz erro quando o número a ser representado é maior do que a distância entre 2 valores adjacentes.
Por exemplo: O número N = 7,51 com incremento e decremento de 0,01 segue 7,52, 7,53, 7,54 .... se o número 7,545 deve ser aproximado para 7,55 e 7,542 para 7,54.

Multiplicação entre números assinalados e não assinalados

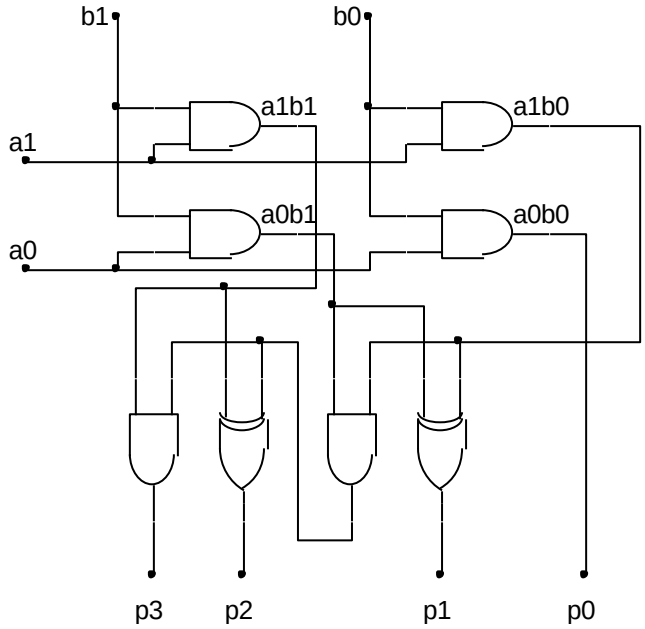
A multiplicação é uma operação aritmética considerada como operação básica e pode ser realizada de diversas formas e conforme o desempenho pode-se ter diferentes tipos de algoritmos.

1. Multiplicação básica – O algorítmo da multiplicação básica é realizado como operações de soma e deslocamentos de acordo com os bits do multiplicador. Um registrador produto de 2n bits é somado ao multiplicando se o bit do multiplicador igual a 1, caso contrário não se faz nada. Um deslocamento a direita do registrador produto e assim até todos os bits do multiplicador forem multiplicados.
A multiplicação a seguir mostra a operação de multiplicação entre 2 números de 2 bits cada.

A = a_1a_0 e B = b_1b_0.
Vamos a multiplicação dos números A e B.

$$\begin{array}{rl} & a_1\ a_0 \\ & b_1b_0 \\ \hline 0 & pp_1\ pp_0 \\ pp_3\ pp_2 & 0 \\ \hline p_3\ p_2 & p_1\ p_0 \end{array} \qquad \begin{array}{l} pp_0 = a_0b_0 \text{ e } pp_1 = a_1b_0\ pp_2 = a_0b_1 \text{ e } pp_3 = a_1b_1,\ c_0 = 0. \\ p_0 = pp_0 = a_0b_0 \text{ e } c_1 = 0. \\ p_1 = pp_1 \oplus pp_2 = a_1b_0 \oplus a_0b_1 \text{ e } c_2 = pp_1.pp_2 = a_1b_0.a_0b_1 \\ p_2 = pp_3 \oplus c_2 = a_1b_1 \oplus a_1b_0.a_0b_1 \text{ e } c_3 = a_1b_1.a_1b_0.a_0b_1 \\ p_3 = c_3 = a_1b_1.a_1b_0.a_0b_1 \end{array}$$

O circuito descrito pelas equações booleanas é apresentado a seguir.



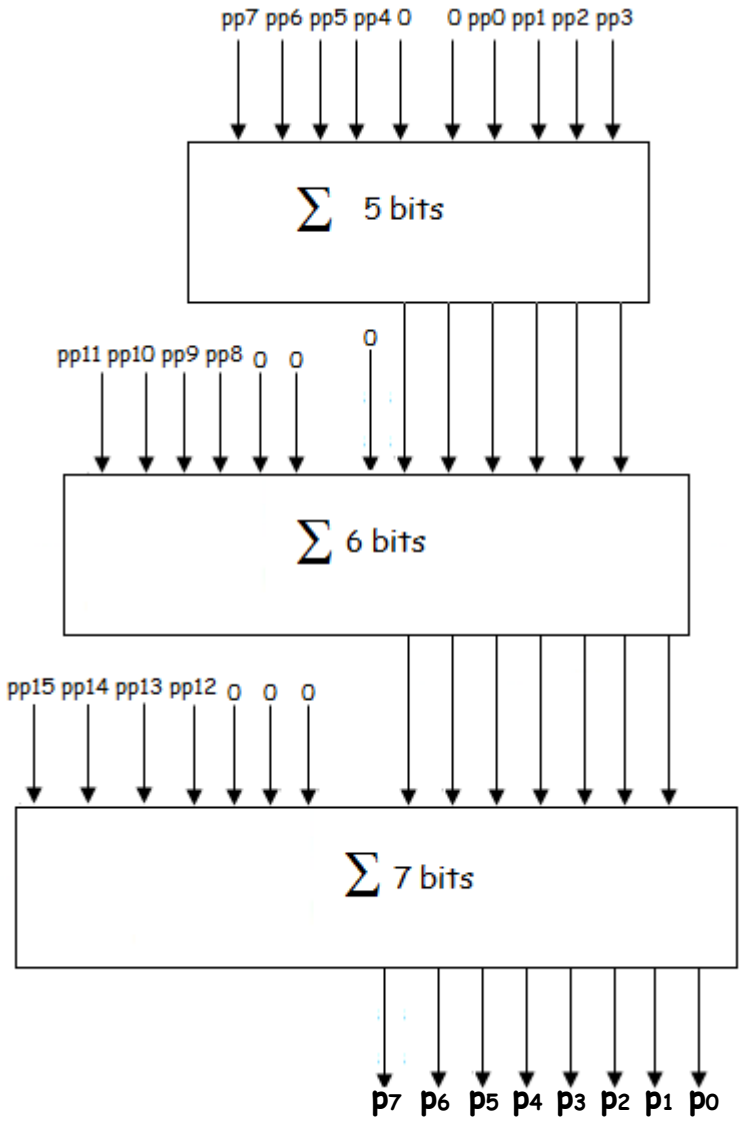
Multiplicar 2 números A e B positivos de 4 bits.

A = $(13)_{10}$ = $(1101)_2$ e B = $(11)_{10}$ = $(1011)_2$

```
   1101
   1011
  01101
  11010+
 0100111
 1101000
10001111 = (143)10
```

A implementação do somador de 4 bits é mostrado a seguir.

Os 16 produtos parciais onde a = $a_3a_2a_1a_0$ e b = $b_3b_2b_1b_0$, os produtos são:
$pp_0 = a_0b_0$, $pp_1 = a_0b_1$, $pp_2 = a_0b_2$, $pp_3 = a_0b_3$, $pp_4 = a_1b_0$, $pp_5 = a_1b_1$, $pp_6 = a_1b_2$, $pp_7 = a_1b_3$, $pp_8 = a_2b_0$, $pp_9 = a_2b_1$, $pp_{10} = a_2b_2$, $pp_{11} = a_2b_3$, $pp_{12} = a_3b_0$, $pp_{13} = a_3b_1$, $pp_{14} = a_3b_2$ e $pp_{15} = a_3b_3$.



2. Algorítmo de Booth

O algoritmo de Booth trata números positivos e negativos de forma uniforme. O algoritmo é baseado no fato de que sequências de zeros ou de uns no multiplicador dispensam as adições, necessitando somente de deslocamentos. As adições e subtrações são usadas somente nos limites das sequências, que são as posições onde as sequências mudam de 0 para 1 ou de 1 para 0.

Procedimento:

1. Inicialização: Para n bits no multiplicando e multiplicador colocar 0's na metade superior e o multiplicador na metade inferior do registrador produto.
2. Colocar um bit 0 no LSB do registrador produto;
3. Da direita para a esquerda do registrador produto observar os bits adjacentes BA

1. Se BA = 00 -> não faça nada;
2. Se BA = 10 -> subtraia o multiplicando ao registrador produto;
   (Início da seqüência de 1's no registrador produto)
3. Se BA = 11 -> não faça nada;

4. Se BA = 01 -> some o multiplicando ao registrador produto;

4. Deslocar o produto um bit a direita;
5. O resultado deve excluir o bit extra.

Exemplo: Encontrar o resultado da multiplicação entre os números A = $(+2)_{10}$ e B = $(-3)_{10}$.

DIVISÃO:

Computar a divisão de A = $(23,625)_{10}$ = $(10111,101)_2$ e B = $(4,50)_{10}$ = $(100,1)_2$

Normalizando os números

$0,23625 \times 10^2 = 10111,101 = 0,10111101 \times 2^5$
$0,45 \times 10^1 = 100,1 = 0,1001 \times 2^3$

M = 0,10111101 ÷ 0,1001 = 0,10101 x 2^1
E = (p1 – p2 + pm) = 101 – 11 + 1 + 1000 = 1011

0	1011	10101

Resultado: 5,25

5. RISC x CISC

O projeto do Conjunto de Instruções inicia com a escolha de uma entre duas abordagens, a abordagem RISC e a CISC. O termo RISC é a abreviação de **Reduced Instruction Set Computer**, ou Computador de Conjunto de Instruções Reduzido e CISC vem de **Complex Instruction Set Computer**, ou Computador de Conjunto de Instruções Complexo. Um computador RISC parte do pressuposto de que um conjunto simples de instruções vai resultar numa Unidade de Controle *simples, barata e rápida*. Já os computadores CISC visam criar arquiteturas complexas o bastante a ponto de *facilitar a construção dos compiladores, assim, programas complexos são compilados em programas de máquina mais curtos*. Com programas mais curtos, *os computadores CISC precisariam acessar menos a memória para buscar instruções e seriam mais rápidos*.

A Tabela 5.1, "Arquiteturas RISC x CISC" resume as principais características dos computadores RISC em comparação com os CISC. Os processadores RISC geralmente adotam arquiteturas mais simples e que acessam menos a memória, em favor do acesso aos registradores. A arquitetura Registrador-Registrador é mais adotada, enquanto que os computadores CISC utilizam arquiteturas Registrador-Memória.

Tabela 5.1. Arquiteturas RISC x CISC

Características	**RISC**	**CISC**
Arquitetura	Registrador-Registrador	Registrador-Memória
Tipos de Dados	Pouca variedade	Muito variada
Formato das Instruções	Instruções poucos endereços	Instruções com muitos endereços
Modo de Endereçamento	Pouca variedade	Muita variedade
Estágios de Pipeline	Entre 4 e 10	Entre 20 e 30
Acesso aos dados	Via registradores	Via memória

Como as arquiteturas RISC visam Unidades de Controle mais simples, rápidas e baratas, elas geralmente optam por instruções mais simples possível, com pouca variedade e com poucos endereços. A pouca variedade dos tipos de instrução e dos modos de endereçamento, além de demandar uma Unidade de Controle mais simples, também traz outro importante benefício, que é a *previsibilidade*. Como as intrusões variam pouco de uma para outra, é mais fácil para a Unidade de Controle prever quantos ciclos serão necessários para executa-las. Esta previsibilidade traz benefícios diretos para o ganho de desempenho com o *Pipeline*. Ao saber quantos ciclos serão necessários para executar um estágio de uma instrução, a Unidade de Controle saberá exatamente quando será possível iniciar o estágio de uma próxima instrução.

Já as arquiteturas CISC investem em Unidades de Controle poderosas e capazes de executar tarefas complexas como a Execução Fora de Ordem e a Execução Superescalar. Na execução Fora de Ordem, a Unidade de Controle analisa uma sequência de instruções ao mesmo tempo. Muitas vezes há dependências entre uma instrução e a seguinte, impossibilitando que elas sejam executadas em Pipeline. Assim, a Unidade de Controle busca outras instruções para serem executadas que não são as próximas da sequência e que não sejam dependentes das instruções atualmente executadas. Isso faz com que um programa não seja executado na mesma ordem em que foi compilado. A Execução Superescalar é a organização do processador em diversas unidades de execução, como Unidades de Pontos Flutuante e Unidades de Inteiros. Essas unidades trabalham simultaneamente. Enquanto uma

instrução é executada por uma das unidades de inteiros, outra pode ser executada por uma das unidades de Pontos Flutuantes. Com a execução Fora de Ordem junto com a Superescalar, instruções que não estão na sequência definida podem ser executadas para evitar que as unidades de execução fiquem ociosas.

É importante ressaltar que a execução fora de ordem não afeta o resultado da aplicação pois foram projetadas para respeitar as dependências entre os resultados das operações.

Estas características de complexidade tornam os estágios de Pipeline dos processadores CISC mais longos, em torno de 20 a 30 estágios. Isto porque estas abordagens de aceleração de execução devem ser adicionadas no processo de execução. Já os processadores RISC trabalham com estágios mais curtos, em torno de 4 a 10 estágios.

Os processadores CISC também utilizam mais memória principal e Cache, enquanto que os processadores RISC utilizam mais registradores. Isso porque os processadores CISC trabalham com um maior volume de instruções e dados simultaneamente. Esses dados não poderiam ser armazenados em registradores, devido à sua elevada quantidade e são, geralmente, armazenados em memória Cache. Enquanto que os processadores RISC trabalham com menos instruções e dados por vez, o que possibilita a utilização predominante de registradores.

5.2 Afinal, qual a melhor abordagem?

Sempre que este assunto é apresentado aos alunos, surge a pergunta crucial sobre qual é a melhor abordagem, a RISC ou a CISC? Esta é uma pergunta difícil e sem resposta definitiva. A melhor resposta que acho é de que depende do uso que se quer fazer do processador.

Processadores RISC geralmente resultam em projetos menores, mais baratos e que consumem menos energia. Isso torna-os muito interessante para dispositivos móveis e computadores portáteis mais simples. Já os processadores CISC trabalham com clock muito elevado, são mais caros e mais poderosos no que diz respeito a desempenho. Entretanto, eles são maiores e consomem mais energia, o que os torna mais indicados para computadores de mesa e notebooks mais poderosos, além de servidores e computadores profissionais.

Os processadores CISC iniciaram com processadores mais simples e depois foram incorporando mais funcionalidades. Os fabricantes, como a Intel e a AMD, precisavam sempre criar novos projetos mas mantendo a compatibilidade com as gerações anteriores. Ou seja, o Conjunto de Instruções executado pelo 486 precisa também ser executado pelo Pentium para os programas continuassem compatíveis. O Pentium IV precisou se manter compatível ao Pentium e o Duo Core é compatível com o Pentium IV. Isso tornou o projeto dos processadores da Intel e AMD muito complexos, mas não pouco eficientes. Os computadores líderes mundiais em competições de desempenho computacional utilizam processadores CISC.

Já o foco dos processadores RISC está na simplicidade e previsibilidade. Além do benefício da previsibilidade do tempo de execução ao Pipeline, ele também é muito interessante para aplicações industriais. Algumas dessas aplicações são chamadas de Aplicações de Tempo Real. Essas aplicações possuem como seu requisito principal o tempo para realizar as tarefas. Assim, o Sistema Operacional precisa saber com quantos milissegundos um programa será executado. Isso só é possível com processadores RISC, com poucos estágios de Pipeline, poucos tipos de instrução, execução em ordem

etc. Mesmo que os processadores RISC sejam mais lentos do que os CISC, eles são mais utilizados nessas aplicações críticas e de tempo real, como aplicações industriais, de automação e robótica.

Resumo: **Arquitetura RISC**

A arquitetura RISC é constituída por um pequeno conjunto de instruções simples que são executadas diretamente pelo hardware, onde não há a intervenção de um interpretador (microcódigo), o que significa que as instruções são executadas em apenas uma microinstrução (de uma única forma e seguindo um mesmo padrão). As máquinas RISC só se tornaram viáveis devido aos avanços de software otimizado para essa arquitetura, através da utilização de compiladores otimizados e que compensem a simplicidade dessa arquitetura. Existe um conjunto de características que permite uma definição de arquitetura básica RISC, são elas:

- Utilização de apenas uma instrução por ciclo do datapath (ULA, registradores e os barramentos que fazem sua conexão);
- O processo de carregar/armazenar, ou seja, as referências à memória são feitas por instruções especiais de load/store;
- Inexistência de microcódigo, fazendo com que a complexidade esteja no compilador;
- Instruções de formato fixo;
- Conjunto reduzido de instruções, facilitando a organização da UC de modo que esta tenha uma interpretação simples e rápida;
- Utilização de pipeline (é uma técnica de dividir a execução de uma instrução em fases ou estágios, abrindo espaço para execução simultânea de múltiplas instruções);
- Utilização de múltiplos conjuntos de registradores.

Processadores RISC

Ao contrário dos complexos CISC, os processadores RISC são capazes de executar apenas poucas instruções simples, e justamente por isso que os chips baseados nesta arquitetura são mais simples e muito mais baratos. Uma outra vantagem dos processadores que utilizam essa arquitetura é o fato de terem um menor número de circuitos internos, permite que se trabalhe com clocks mais altos. Um bom exemplo são os processadores Alpha, que em 97 já operavam a 600 MHz.

Arquitetura CISC

CISC ou Complex Instruction Set Computer, é uma arquitetura de processadores capaz de executar centenas de instruções complexas diferentes o que a torna extremamente versátil. Exemplos de processadores que utilizam essa arquitetura são os 386 e os 486 da Intel. Os processadores baseados na computação de conjunto de instruções complexas contêm uma micro-programação, ou seja, um conjunto de códigos de instruções que são gravados no processador, permitindo-lhe receber as instruções dos programas e executá-las, utilizando as instruções contidas na sua micro-programação. Seria como quebrar estas instruções, já em baixo nível, em diversas instruções mais próximas do hardware (as instruções contidas no microcódigo do processador).

Tecnologia CISC

Como característica marcante esta arquitetura contém um conjunto grande de instruções, a maioria deles em um elevado grau de complexidade. Algumas características dessa arquitetura são:

- Controle microprogramado;
- Modos registrador-registrador, registrador-memória, e memória-registrador;
- Múltiplos modos de endereçamento à memória, incluindo indexação (vetores);
- Instruções de largura (tamanho) variável, conforme modo de endereçamento utilizado;
- Instruções requerem múltiplos ciclos de máquina para execução, variando também com o modo de endereçamento;
- Poucos registradores;
- Registradores especializados.

O fato é que cada arquitetura só será melhor dependendo do objetivo final a ser alcançado, processadores do tipo RISC se saem melhor quando o assunto é servidores, smartphones e supercomputadores. Enquanto processadores do tipo CISC geralmente são usados em computadores de uso cotidiano.

Atualidades

Atualmente não se pode afirmar com 100% de certeza que um processador utiliza apenas a arquitetura CISC ou RISC, pois os modelos atuais de processadores abrigam as características de ambas as arquiteturas. Processadores ARM usados em celulares são um com exemplo de uso da arquitetura RISC, outro exemplo de uso dessa arquitetura é em consoles como o Nintendo 64 e o Playstation.

Com o passar dos anos, tanto a Intel quanto a AMD perceberam que usar alguns conceitos da arquitetura RISC em seus processadores poderia ajudá-las a criar processadores mais rápidos. Porém, ao mesmo tempo, existia a necessidade de continuar criando processadores compatíveis com os antigos. A ideia então passou a ser construir chips híbridos, que fossem capazes de executar as instruções x86, sendo compatíveis com todos os programas, mas ao mesmo tempo comportando-se internamente como chips RISC, quebrando estas instruções complexas em instruções simples, que podem ser processadas por seu núcleo RISC.

Análise quantitativa de desempenho

Quando estimamos o desempenho da máquina, a medida mais utilizada é o tempo de execução T. Quando uma melhoria no desempenho é apresentada, o efeito da melhoria é expresso em termos de speedup S. O seu cáculo é a razão entre o tempo de execução sem a melhoria T_{w0} e o tempo de execução com a melhoria T_w.

$$S = T_{w0} / T_w.$$

Por exemplo se somarmos um módulo cache de 1MB a um sistema de computador e obter uma redução no tempo de execução de 12 para 8 segundos, podemos dizer que houve uma melhoria de desempenho de 50%, pois a relação entre 12 e 8 é de 1,5 ou seja um aumento de 50% e em termos de porcentagem fica:

$S = (T_{w0} – T_w)/ T_w$.

Podemos derivar uma equação mais detalhada para estimar T se dispor das informações sobre o período do clock da máquina, t, o número de ciclos de clock por instrução CPI (clocks por instrução) e uma contagem do número de instruções executadas pelo programa durante a execução, IC (contador de instruções). O tempo total de execução é dado por:

$T = IC \times CPI \times t$

Os valores de CPI e IC podem ser expressos com o uma média sobre o conjunto de instruções e o número total, respectivamente. A equação final do desempenho será:

$S = 100 \times (IC_{w0} \times CPI_{w0} \times t_{w0} – IC_w \times CPI_w \times t_w) / IC_w \times CPI_w \times t_w$

Exemplo: Calcular o speedup obtido substituindo-se uma CPU com CPI médio de 5 por outra CPI médio de 3,5, com o período de clock aumentado de 100ns para 120ns.

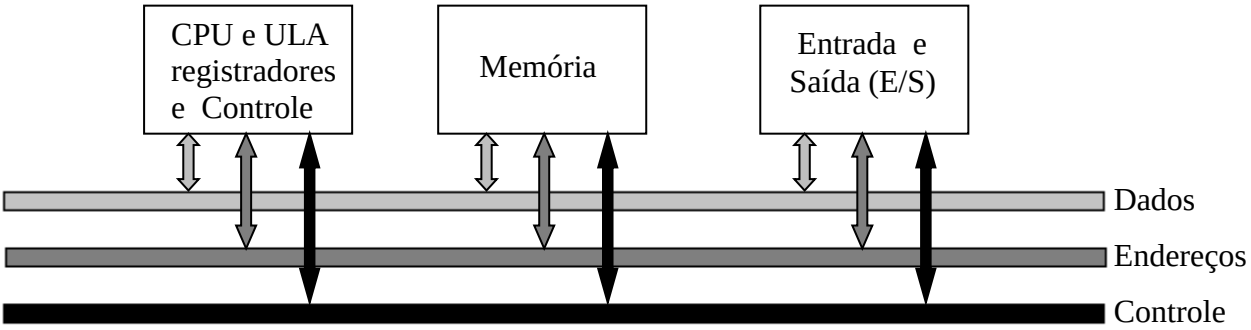
$S = 100 \times (5 \times 100 – 3,5 \times 120)/ 3,5 \times 120 = 19\%$

Arquitetura do conjunto de instruções

O estudo da arquitetura de computadores passa pelo conhecimento das diversas arquiteturas CISC utilizada nos processadores Intel até o Pentium e RISC a partir dos anos 80 e presentes em processadores ARM e outros. Uma linguagem comum a todos os processadores é chamada de linguagem de máquina que é a linguagem entendida pelo computador. Como a sua construção é códigos formados por 0s e 1s, os programadores utilizam uma linguagem chamada de assembly ou de montagem, a qual usa nomes ao invés de códigos para as instruções tipo ADD, uma instrução de soma e que convertida para a linguagem de máquina é 10110010. Para que possamos descrever a natureza da linguagem de montagem de programação em linguagem de montagem, escolhemos como arquitetura comercial SPARC comum a computadores SUN (Stanford University Network).

Componentes da arquitetura do conjunto de instruções

O ISA (Instruction Set Architecture) de um computador apresenta ao programador de linguagem de montagem uma visão da máquina que inclui todo o hardware acessível ao programador e instruções que manipulam dados dentro desse hardware.



Barramentos do Sistema

Funcionamento

O usuário escreve o programa em linguagem de alto nível, a qual um compilador traduz para a linguagem de montagem. O montador então traduz a linguagem de montagem para a linguagem de máquina, ou seja, código de máquina.

Barramento do sistema

A partir daí o processo de execução passa pela instrução sendo levada até a CPU no ciclo de busca da instrução pela CPU, a qual decodifica a instrução e executa. A instrução é guardada na memória de programa e endereços sequenciais, onde o PC, contador de instrução, vai buscar para a CPU executar. A comunicação entre a memória e a CPU é feita através do barramento de dados, de endereços e de controle.

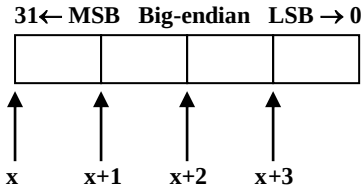
Memória

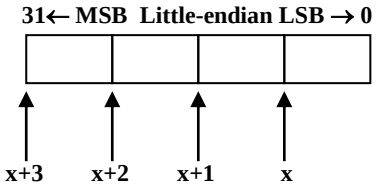
A memória do computador consiste na construção de uma série de registradores numerados consecutivamente (endereçados), onde cada um dos quais armazena um byte igual a 8 bits. A seguir a terminologia utilizada para os tipos de dados no computador.

Bit = Valor 0 ou 1.
Nibble = 4 bits
Byte = 8 bits;
Palavra de 16 bits (meia palavra)
Palavra de 32 bits;
Palavra de 64 bits (dupla palavra)
Palavra de 128 bits (quádrupla palavra)

Uma ins1trução ocupa no mínimo na memória um byte, embora instruções possam ocupar mais bytes e são armazenas numa sequência de bytes. A maioria das máquinas têm instruções que podem acessar bytes, meia palavra, palavra e palavras duplas. Quando se trata de múltiplos dados armazenados na memória, o formato pode ser: byte mais significativo no menor endereço, chamado de big-endian, ou o

byte menos significativo armazenado no menor endereço little-endian. O termo endian vem da questão se "ovos devem ser quebrados no lado maior ou no lado menor" no livro "Viagens de Gulliver".

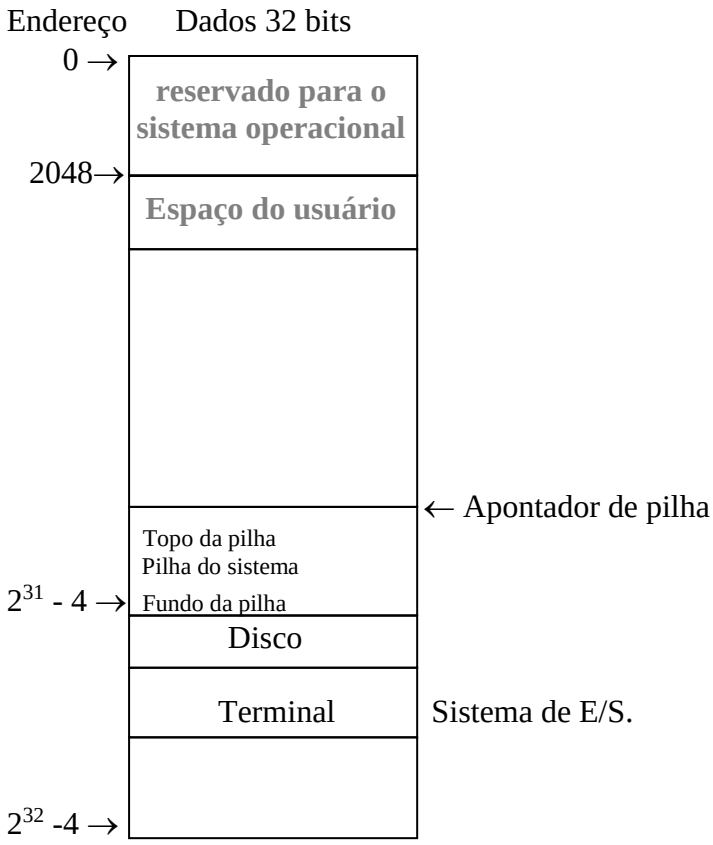




Cada um têm 8 bits.

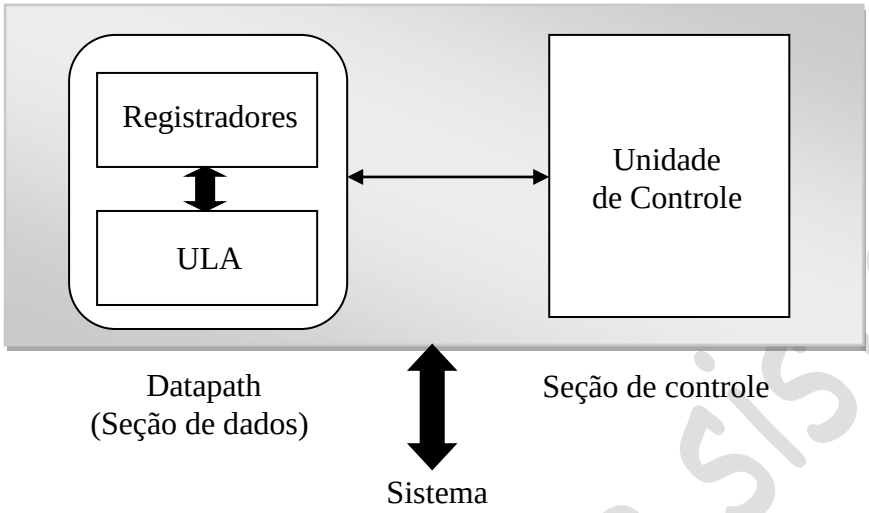
Organização da memória

As memórias podem ser organizadas em endereços consecutivos e lineares e cada um corresponde a uma palavra armazenada (a palavra é composta de 4 bytes nesse caso). Uma memória de 32 bits de endereçamento tem o endereço 0 como o menor endereço e 2^{32} -1 como o maior e último endereço. O mapa de memória é dividido em seções distintas pelo sistema operacional, assim: memória de dados, memória de entrada/saída, pilha e programa do usuário, por exemplo. A seguir apresentamos um mapa de memória com sistema de entrada e saída mapeado nela.

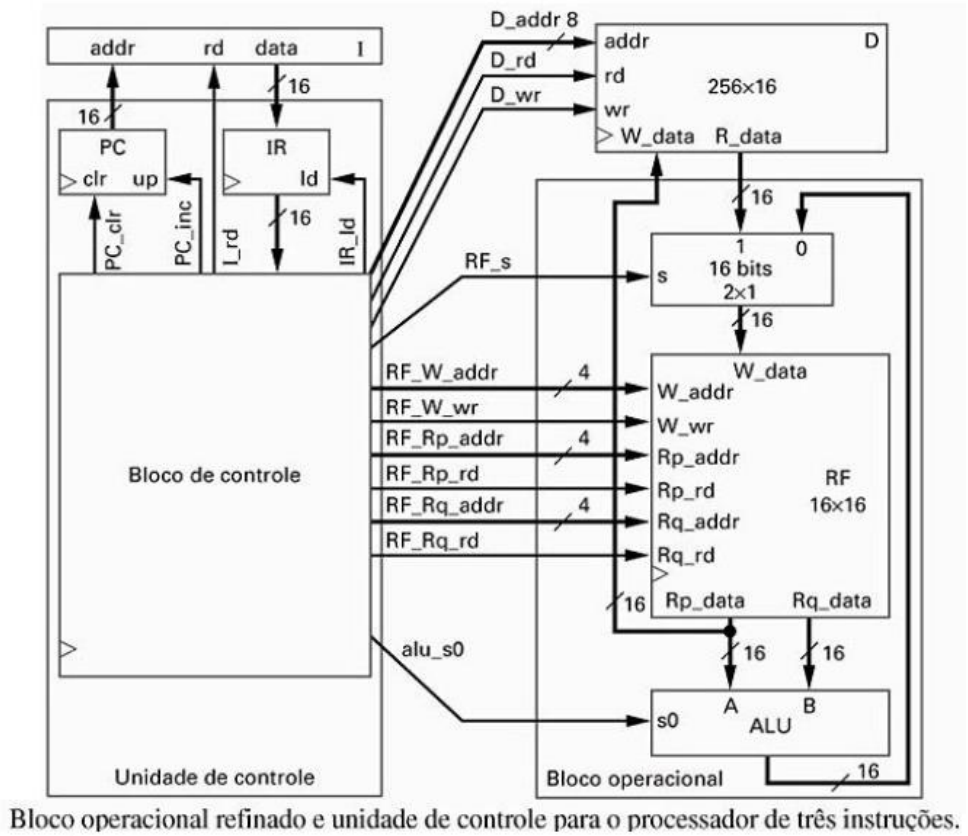


6. ARC- Um computador com arquitetura RISC

A CPU (unidade central de processamento), consiste numa seção de dados, que contém registradores, uma unidade lógica e aritmética e uma seção de controle que interpreta as instruções, efetua transferências entre registradores, como ilustrado na figura a seguir.



A unidade de controle de um computador é responsável pela execução das instruções do programa armazenadas na memória principal. A figura a seguir mostra uma CPU completa e sua interface com a memória.



Bloco operacional refinado e unidade de controle para o processador de três instruções.

Da figura olhando a seção de controle dois registradores são muito importantes na busca da instrução. São eles o PC (contador de instrução) e o IR (registrador de instrução).

Contador de programa (PC) – É responsável por conter o endereço da instrução a ser executada pela CPU. A instrução apontada pelo PC é lida da memória e armazenada no IR onde é decodificada. Esse ciclo é chamado de ciclo de busca da instrução.

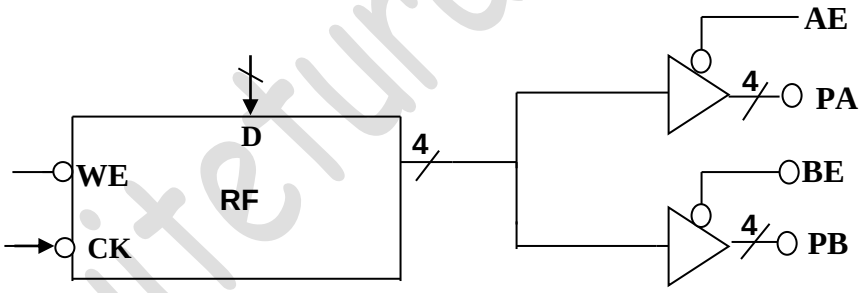
Registrador de instrução (IR) – É responsável pelo armazenamento temporário da instrução enquanto a unidade de controle gera os sinais de controle para o "datapath" executar a instrução.

Ciclo de instrução

1. Leitura da próxima instrução a ser executada;
2. Decodificar o código de operação da instrução;
3. Leitura/escrita na memória do operando da instrução se houver;
4. Executar a instrução e armazenar os resultados;
5. Repetir.

DATAPATH – Caminho ou fluxo de dados – O fluxo de dados é uma coleção de registradores, unidade lógica e aritmética, multiplexador, comparador, multiplicador, e dutos de comunicações que junto com a CPU é responsável pela manipulação de dados na execução da instrução. Cada sub-sistema do datapath é responsável por uma função e por exemplo o registrador de arquivo (Register File) é um sub-sistema descrito a seguir. Pode ter n registradores internos de largura de m bits.

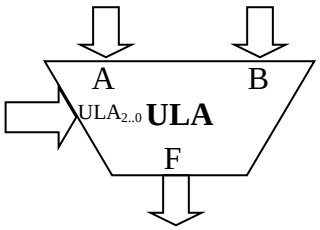
Bloco do Registrador de Arquivos - RF



PA = Porto A (saída)
PB = Porto B (saída)
WE = Habilita reg.
D = Entrada de dados
AE = Habilita porto A
BE = Habilita porto B

Unidade Lógica e aritmética (ULA) – É responsável pelas operações lógicas e aritméticas e possui duas entradas para os operando A e B e uma saída do resultado da operação realizada. Possui uma entrada funcional de 3 a 4 bits, as quais selecionam a função realizada pela ULA. A seguir é apresentada a tabela funcional da ULA de 3 bits. A ULA pode ter largura de n bits e algumas saídas como bit "Vai um" (carryout) da operação, bit de estouro (overflow). A seguir a tabela funcional da ULA.

ULA_2	ULA_1	ULA_0	Funcional
0	0	0	Soma A + B
0	0	1	Subtração A – B
0	1	0	A + 1
0	1	1	A - 1
1	0	0	A and B
1	0	1	A or B
1	1	0	A xor B
1	1	1	A



Conjunto de Instruções – O conjunto de instruções de um processador é o número de instruções as quais esse processador pode executar. É particular para cada processador e, portanto, cada processador possui o seu conjunto próprio de instruções as quais ele pode executar. As instruções podem carregar os operandos e tipos de operandos e os resultados gerados podem ser armazenados internamente ou na memória ou enviados para o sistema de entrada e saída de dados. Os programas escritos para um processador podem ser compilados (convertidos para a linguagem de baixo nível) e executada pela CPU. Os compiladores são personalizados para cada processador.

Compilador – É um programa de computador que transforma programas escritos em linguagem de alto nível, tais como, C, Pascal, Fortran entre outras, em linguagem de baixo nível ou linguagem de máquina.

Front- End – Compiladores para a mesma linguagem de alto nível têm esse nome e reconhece comandos na linguagem de alto nível.

Back – End – É responsável por gerar o código máquina para um processador específico. Cada processador terá o seu "back-end" e será diferente para cada processador alvo. Diferentes compiladores gerarão diferentes códigos de máquinas para o mesmo processador.

SPARC – Arquitetura baseada no processador comercial Scalable Processor Architecture (SPARC) que foi desenvolvido pela SUN Microsystems nos anos 80. As características do SPARC são:

- Arquitetura de natureza "aberta";
- Arquitetura RISC;
- Iniciou com 32 bits e migrou para 64 bits;
- Instruções orientada para registrador;
- Uma arquitetura muito similar a MIPS;
- Escalável permite implementações em escala de processadores embarcados para grandes servidores, todos compartilhando o mesmo núcleo e conjunto de instruções.

O ARC tem a maioria das características mais complexas desse processador baseado.

MEMÓRIA ARC - É uma memória de 32 bits com memória endereçada por byte e pode manipular dados de 32 bits, mas todos os dados são armazenados na memória como bytes e o endereço de uma palavra de 32 bits é o endereço do byte que tem o menor endereço. O ARC é uma arquitetura big-endian, o bye MSB é armazenado no menor endereço. Por exemplo, na arquitetura ARC, o maior endereço possível é $2^{32} – 1$, assim o byte MSB de uma palavra armazenada na última locação de

memória será no endereço $2^{32} – 4$ e o byte LSB estará no endereço de memória $2^{32} – 1$. O mapa de memória ARC é apresentado a seguir.

Mapa de memória ARC

Dispositivos de Entrada/Saída Espaço de $2^{32} – 1$ a 2^{31}
PILHA • Inicia no endereço $2^{31} – 4$ • Cresce para baixo;
Capacidade de 2^{11} Espaço reservado para o programa do usuário, podendo crescer até a pilha.
Capacidade de 2^{11} reservado para o sistema operacional

O conjunto de instruções ARC – O ARC têm 32 registradores de 32 bits de uso geral, um PC e um IR. Um registrador de status da ULA (PSR), o qual contém informações (bits flags) os quais informam o resultado da operação aritmética realizada, tais como, zero, overflow, carry e outras. Todas as instruções têm o tamanho de 32 bits. As operações com a memória são feitas usando as instruções Load e Store. As operações aritméticas são realizadas entre registradores e o resultado é armazenado em registrador. São aproximadamente 200 instruções para o SPARC e o ARC se baseou com um sub-conjunto de 15 instruções. Cada instrução é representada por um mnemônico, que é o nome que representa a instrução.

Mnemônico	Significado
ld	Leia um registrador da memória
st	Grave um registrador na memória
sethi	Leia os 22 bits MSB do registrador
andcc	AND bit a bit
orcc	OR bit a bit
orncc	NOR bit a bit
srl	Deslocamento à direita
andcc	Soma
call	Chama a subrotina
jmpl	Desvie e ligue (jump and link, retorno de chamada de subrotina
be	Desvie se igual
bneg	Desvie se negativo
bcs	Desvie se excedente

bvs	Desvie se overflow
ba	Desvio incondicional

Esse sub-conjunto com 15 instruções de ISA ARC podem ser classificadas como:

1. Instruções de manipulações de dados – São instruções que transferem dados entre a memória e registrador ARC e são elas ld e st.
A instrução sethi – Set os 22 bits MSB do registrador com uma constante de 22 bits contida dentro da instrução.

2. Instruções lógica e aritmética – São instruções as quais efetuam uma operação AND, OR ou NOR bit a bit em seus operandos. Um dos operandos deve ser um registrador e o resultado é armazenado em um registrador. A instrução srl lógica de deslocamento à direita

3. Instruções de controle – São instruções como call e jmpl que são usadas para chamar e retorna de uma subrotina. As instruções de desvio do programa como be, bneg, bcs, bvs e ba.

Formato da linguagem de montagem ARC – A sintaxe da linguagem de montagem SPARC é mostrado com a formatação a seguir. Consiste de 4 campos sendo: um campo opcional de rótulo (label), um campo de código de operação, um ou mais campos especificando os operandos origem e destino de existirem e um campo opcional para comentário.

Rótulo – Consiste em uma combinação de caracteres alfabéticos ou numéricos, tais como: underscore (_), cifrão ($), ponto (.), contanto que não seja um dígito. A linguagem é sensitiva maiúsculo/minúsculo e tem "formato livre", mas a ordem deve ser mantida.

Rótulo	**Mnemônico**	**Operando origem**	**Operando destino**	**Comentário**
lab_1	addcc	%r1,%r2	%r3	!Código de montagem exemplo

Identificação dos 32 registradores de 32 bits de largura.

Registrador 00	% r0	[= 0]
Registrador 01	% r1	
Registrador 02	% r2	
Registrador 03	% r3	
Registrador 04	% r4	
Registrador 05	% r5	
Registrador 06	% r6	
Registrador 07	% r7	
Registrador 08	% r8	
Registrador 09	% r9	
Registrador 10	% r10	

Registrador 11	% r11	
Registrador 12	% r12	
Registrador 13	% r13	
Registrador 14	% r14	% sp
Registrador 15	% r15	% ligação
Registrador 16	% r16	
Registrador 17	% r17	
Registrador 18	% r18	
Registrador 19	% r19	
Registrador 20	% r20	
Registrador 21	% r21	

Registrador 22	% r22
Registrador 23	% r23
Registrador 24	% r24
Registrador 25	% r25
Registrador 26	% r26
Registrador 27	% r27
Registrador 28	% r28
Registrador 29	% r28
Registrador 30	% r30
Registrador 31	% r31

PSR	%psr

PC	%pc

Esses registradores PSR e PC de 32 bits de largura.

A instrução addcc %r1, 12, %r3 soma o conteúdo do registrador 1 com $(12)_{10}$ e o resultado será armazenado no registrador r3. Os números sempre na base decimal, exceção quando precedido por "0x" ou terminado em H (hexadecimal).
Comentário inicia com ponto de exclamação!

Formatos das instruções ARC – Define vários campos de bits de uma instrução**.** A arquitetura ARC têm cinco formatos de instruções. **SETHI**, **desvio**, **CALL** e **memória**. Cada instrução tem o seu mnemônico e a seguir apresentamos o formato da instrução.

1. Formato SETHI

Op.			
31 – 30	29 – 25	24 – 22	21 – 00
(2 bits)	(5 bits)	(3 bits)	(22 bits)
00	rd	op2	imm22

2. Formato de desvio

Op.				
31 – 30	29	28 – 25	24 – 22	21 – 00
(2 bits)	(1 bit)	(5 bits)	(3 bits)	(22 bits)
00	0	cond	op2	disp22

3. Formato de chamada

Op.	
31 – 30	29 – 00
(2 bits)	(30 bits)
01	disp30

4. Formatos aritméticos

Op.				i	
31 – 30	29 – 25	24 – 19	18 – 14	13 – 05	04 – 00
(2 bits)	(5 bits)	(6 bits)	(5 bits)	(9 bits)	(5 bits)
10	rd	op3	rs1	0..... 0	rs2

Op.				i	
31 – 30	29 – 25	24 – 19	18 – 14	13	12 – 00
(2 bits)	(5 bits)	(6 bits)	(5 bits)	(1 bit)	(13 bits)
10	rd	op3	rs1	1	simm13

Formato de memória

Op.					
31 – 30	29 – 25	24 – 19	18 – 14	13 – 05	04 – 00
(2 bits)	(5 bits)	(6 bits)	(5 bits)	(9 bits)	(5 bits)
11	rd	op3	rs1	0.....0	rs2

Op.				i	
31 – 30	29 – 25	24 – 19	18 – 14	13	12 – 00
(2 bits)	(5 bits)	(6 bits)	(5 bits)	(1 bit)	(13 bits)
10	rd	op3	rs1	1	simm13

Resumo de formato das instruções

op	Formato
00	SETHI/Branch
01	CALL
10	Aritmética
11	Memória

op2	Instrução
010	branch
100	sethi

op3	Op = 11
000000	ld
000100	st

Op3	Op = 10
010000	addcc
010001	andcc
010010	orcc
010110	orncc
100110	srl
111000	jmpl

Cond	Desvio
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba

Formato do PSR do ARC

31 – 24	23 – 20	19 – 00
-	n z v c	-

Opcode – São os 2 bits mais significativos do formato da instrução.

Imm22 e disp22 – São usados como operandos de 22 bits, como constante para o formato SETHI (imm2) ou para calcular o tamanho do desvio para o endereço (disp22).

Formatos dos dados ARC

O ARC implementa 12 formatos de dados, conforme ilustrado a seguir e os dados são agrupados em 3 tipos inteiro com sinal, inteiro sem sinal e ponto flutuante. Dentro desses tipos, as larguras de dados permitidas são:

a. meia palavra ou byte (8bits);
b. palavra (16bits);
c. palavra simples (32bits);
d. palavra marcada (32bits), sendo 2 bits para a marca e 30 bits o valor;
e. palavra dupla (64bits);
f. palavra quádrupla (128 bits).

Obs.: O ARC não diferencia o número sem sinal ou com sinal, pois ambos são inteiros em complemento de dois.

Os formatos de ponto flutuante seguem o padrão IEEE 754 – 1985.

Descrição das instruções ARC

A descrição das 15 instruções listadas a seguir detalha o código objeto (opcode) de cada uma instrução, onde a referência *conteúdo* de uma posição de memória (ld e st) é indicada por colchetes.

Instrução: ld

Descrição: Carrega um registrador de um conteúdo alocada na memória. O endereço de memória deve estar referenciado com a instrução e deve ser divisível por 4. O endereço é calculado de acordo com o formato da instrução. Quando o bit 13 do formato igual:
Bit 13 = 0 – O endereço é imediato simm13, porém se bit campo (13) do dado for igual a 1 (pode ser usado até 32bits).

Exemplo: Transferir o conteúdo da posição de memória 2064 para o registrador 1.

```
                Op   rd      op3     rs1   i     simm13
ld [x], %r1 – 11 00001 000000 00000 1 0100000010000.
```

Instrução: st

Descrição: Armazena na posição de memória o conteúdo de um registrador. O endereço de memória deve estar referenciado com a instrução e deve ser divisível por 4. O endereço é calculado de acordo com o formato da instrução. Quando o bit 13 do formato igual:
Bit 13 = 0 – O endereço é imediato simm13, porém se bit campo (13) do dado for igual a 1 (pode ser usado até 32bits).

Exemplo: Transferir o conteúdo do registrador 1 para a memória 2064.

```
                Op   rd      op3     rs1   i     simm13
st %r1, [x] – 11 00001 000100 00000 1 0100000010000.
```

Instrução: sedthi

Posiciona os 22 bits mais significativos e zera os 10 bits menos significativos de um registrador. Se o registrador especificado for r0, a instrução se comporta como nop (no operation).

Exemplo: Sethi 0x304F15 no registrador r1.

sethi %r1,0x304F15 – 00 00001 100 11 0000 0100 1111 0001 0101

Instrução: andcc

AND bit a bit dos operandos origem para o operando destino. O resultado são bits set e reset de acordo com a função booleana.

Exemplo: andcc %r1,%r2,%r3 – Operação %r3 = %r1 AND %r2.

10 00011 010001 00001 0 00000000 00010

Instrução: orcc

OR bit a bit dos operandos origem para o operando destino. O resultado são bits set e reset de acordo com a função booleana.

Exemplo: orcc %r1,1,%r1 – Operação %r1 = %r1 OR 1.

10 00001 010010 00001 1 00000000 00001
Instrução: orncc

NOR bit a bit dos operandos origem para o operando destino. O resultado são bits set e reset de acordo com a função booleana.

Exemplo: orncc %r1,%r0,%r1 – Operação %r3 = %r1 NOR %r2.

10 00001 010110 00001 0 00000000 00000

Instrução: srl

Deslocar um bit a direita de zero a 31 bits. Os bits vagos são preenchidos com zeros.

Exemplo: srl %r1,3,%r2 – Operação %r2 = %r1 deslocar %r1 à direita por 3bits e armazenar em %r2.

10 00010 100110 00001 1 00000000 00011

Instrução: addcc

Somar os operandos origem e o resultado no operando de destino usando aritmética em complemento de dois.

Exemplo: addcc 5 a %r1 – Operação %r1 = %r1 addcc 5.

10 00001 010000 00001 1 00000000 00101

Instrução: call

Chamar uma sub-rotina e armazenar o endereço da instrução atual (onde a chamada ocorreu) em %r15 efetuando uma operação de “chamar e ligar”. O campo disp30 conterá a distância do endereço da instrução call. O endereço da próxima instrução a ser executada é calculado somando-se 4 x disp30

(que desloca disp30 para os 30 bits mais significativos de endereço de 32 bits) para endereço da instrução atual. O disp30 pode ser negativo.

Exemplo: call sub_r – Operação chamar uma sub-rotina que inicia em sub_r. O sub_r está a 25 palavras (100bytes) à frente da instrução call na memória. Cada palavra ocupa espaço de 4 bytes na memória.

01 00000000000000000000000011001

Instrução: jmpl

Deasviar e ligar (retornar de uma sub-rotina). Desviar para um novo endereço e armazenar o endereço da instrução corrente (onde a instrução jmpl estiver localizada) no registrador destino.

Exemplo: jmpl %r15 + 4, %r0 – Retornar de uma sub-rotina. O valor do PC para a instrução de chamada tinha previamente armazenado em %r15 e, portanto, o endereço de retorno deve ser calculado para a instrução que segue a chamada, em %r15 +4. O endereço atual é descartado em %r0.

10 00000 111000 01111 1 000000000100

Instrução: be

Se o código z da condição z for 1, então, desvie para o endereço calculado somando-se 4 x disp22 no formato de desvio ao endereço da instrução atual. Se o código de condição z for zero, então o controle é transferido para a instrução que segue be.

Exemplo: be label – O label está a 5 palavras da instrução (20 bytes) além da instrução be na memória.

00 0 0001 010 0000000000000000000101

Instrução: bneg

Se o código n da condição n for 1, então, desvie para o endereço calculado somando-se 4 x disp22 no formato de instruções de desvio ao endereço da instrução atual. Se o código de condição n for zero, então o controle é transferido para a instrução que segue bneg.

Exemplo: bneg label – O label está a 5 palavras da instrução (20 bytes) além da instrução bneg na memória.

00 0 0110 010 0000000000000000000101

Instrução: bcs

Se o código c da condição n for 1, então, desvie para o endereço calculado somando-se 4 x disp22 no formato de instruções de desvio ao endereço da instrução atual. Se o código de condição c for zero, então o controle é transferido para a instrução que segue bcs.

Exemplo: bcs label – O label está a 5 palavras da instrução (20 bytes) além da instrução bcs na memória.

00 0 0101 010 00000000000000000101

Instrução: bvs

Se o código v da condição v for 1, então, desvie para o endereço calculado somando-se 4 x disp22 no formato de instruções de desvio ao endereço da instrução atual. Se o código de condição v for zero, então o controle é transferido para a instrução que segue bvs.

Exemplo: bvs label – O label está a 5 palavras da instrução (20 bytes) além da instrução bvs na memória.

00 0 0111 010 00000000000000000101

Instrução: ba

Desvie para o endereço calculado somando-se 4 x disp22 no formato de instruções de desvio ao endereço da instrução atual.

Exemplo: ba label – O label está a 5 palavras da instrução (20 bytes) aquém da instrução ba na memória.

00 0 1000 010 11111111111111111011

PSEUDO – OPS

Além das instruções que são implementadas na arquitetura ARC existem também as pseudo-operações que não são opcodes, mas instruções para o montador efetuar alguma ação durante a montagem. A lista das pseudo-operações é apresentada a seguir.

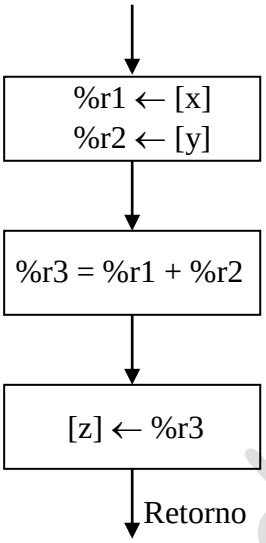
Psedudo-operação	Uso	Significado
.equ	X .equ #10	Considere o símbolo X como $(10)_{16}$
.begin	.begin	Início da montagem
.end	.end	Término da montagem
.org	.org 2048	Modifique o contador de posições para 2048
.dwb	.dwb 25	Reserve um bloco de 25 palavras
.global	.global Y	Y é usado em outro módulo
.extedrn	.extern Z	Z é definido em outro módulo
.macro	.macro M a,b,...	Define macro M com parâmetros formais a, b,...
.endmacro	.endmacro	Fim da definição de macro
.if	.if <cond>	Monte se <cond> for verdadeira
.endif	. endif	Fim da construção .if

Exemplos de programas de montagem

Programa: Somar dois inteiros

Os operandos são 15 e 9 e armazenados nas variáveis x e y e a variável z é o resultado da soma.

z = x + y. Onde x, y e z posições de memória e os registradores r1, r2 e r3. O programa é montado no endereço 2048.

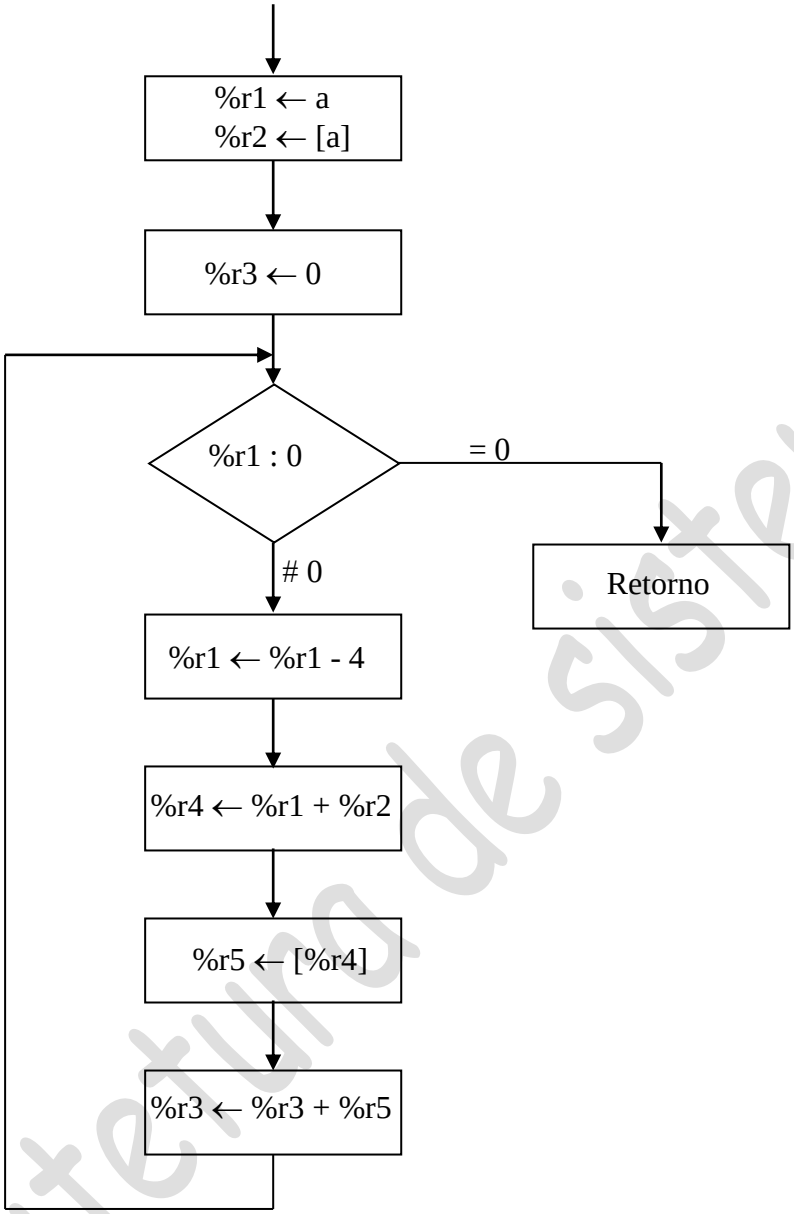


! Este programa soma dois números inteiros

```
        .begin
        .org 2048
Progl:  ld      [x],    %r1         !leia x em %r1
        ld      [y],    %r2         !leia y em %r2
        addcc   %r1,    %r2,  %r3   !%r3 = %r1 + %r2
        st      %r3,    [z]         !armazena %r3 em z
        jmpl    %r15 + 4      ,%r0  !retorne
x:      15
y:      9
z:      0
        .end
```

Programa: Soma de um vetor de inteiros

Definições:
%r1 = Tamanho do vetor a
%r2 = Endereço inicial de a
%r3 = A soma parcial
%r4 = Apontador para o vetor a
%r5 = Armazena um elemento de a




```
! Este programa soma N números inteiros

!definições dos registradores: %r1 – Tamanho do vetor a
!                              %r2 – Endereço inicial de a (base)
!                              %r3 – Soma parcial
!                              %r4 – Apontador para o vetor a
!                              %r5 – Armazena o valor de a

                .begin                  !Inicia a montagem
                .org 2048
inicio          .equ 3000
                ld [n],         %r1             ! %r1 ← tamanho do vetor a
                ld [end],       %r2             ! endereço de a
                andcc %r3,      %r0,    %r3     ! %r3 ← 0
```

```
loop:   andcc %r1,    %r1,   %r0    ! Teste o restante dos elementos
        be                          ! teste se fim, n = 0

fim:    jmpl %r15 + 4,        %r0   ! Retorno para o programa principal

n:      20                          ! 5 números (20 bytes) em a
end:    inicio

        .org inicio                 !inicio do vetor a

a:      25
        -10
        33
        -5
        7
        .end                        !fim da montagem
```

7. Modos de endereçamento ARC- Um computador com arquitetura RISC

Existem 7 tipos de endereçamento na arquitetura ARC para o acesso aos dados, embora cada modo tenha seu uso particular de acordo com a necessidade

Endereçamento imediato – Permite uma referência a uma constante que é conhecida durante a montagem.

Endereçamento direto – É usado para acessar dados cujo endereço é conhecido durante a montagem.

Endereçamento indireto – É usado para acessar uma variável que aponta para os dados cujo endereço é conhecido durante a montagem. Raramente utilizada por ser uma instrução complexa.

Endereçamento indireto por registrador – É usado quando o endereço de um operando não é conhecido até a montagem. As instruções push e pop que também incrementam e decrementam registrador respectivamente.

Endereçamento indexado por registrador, **baseado em registrador e indexado baseado em registrador** – São usados para acessar componentes de vetores e componentes abaixo da pilha, em uma estrutura de dados conhecida como stack frane ou quadro de pilha.

PILHAS E LIGAÇÃO DE SUB-ROTINAS

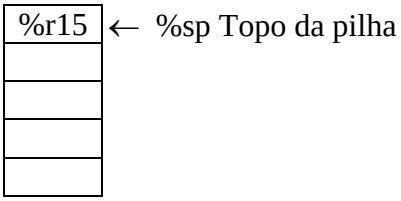
Uma sub-rotina é um programa que é chamado pelo programa principal e é criada sempre que o programa principal necessitar dessa sequência de instruções várias vezes. Esse programa passa parâmetros ao programa principal quando é executada e é chamado de ligação de sub-rotina.

Uma das convenções de chamada simplesmente coloca os argumentos em registradores. Se o número de argumentos ultrapassa o número de registradores, uma segunda convenção deve ser utilizada. O exemplo a seguir mostra a chamada de ligação de sub-rotina.

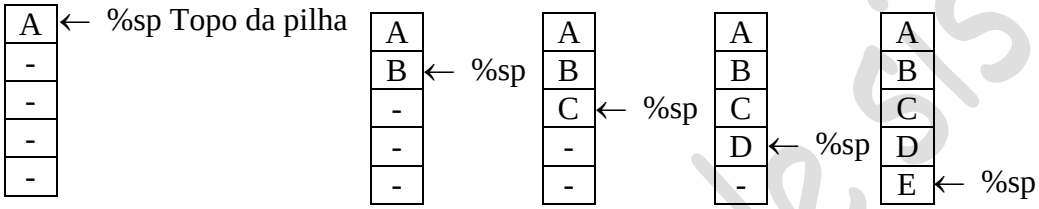
```
! Rotina principal                    !Rotina chamada

        .                             !%r3 ← %r1 + %r2
        .
        .
        ld      [x],    %r1           add_1: addcc %r1, %r2, %r3
        ld      [y],    %r2           jmpl    %r15 +4, %r0
        call    add_1
        st      %r3,    [z]
        .
        .
        .
x:      53
y:      10
z:      0
```

LIFO – (*last in first out)* – A operação da pilha é acessada por instruções push e pop, as quais colocam na pilha (stack) conteúdos de registradores e removem esses conteúdos. O topo da pilha recebe os conteúdos e é apontada por um registrador conhecido como apontador de pilha (stack pointer). A operação LIFO, onde o último dado que entra é o primeiro a sair da pilha de memória. O funcionamento do apontador de pilha descreve a operação com detalhes conforme a seguir.

Pilha



Exemplo: Colocar 5 dados na sequencia A,B,C,D e E,,

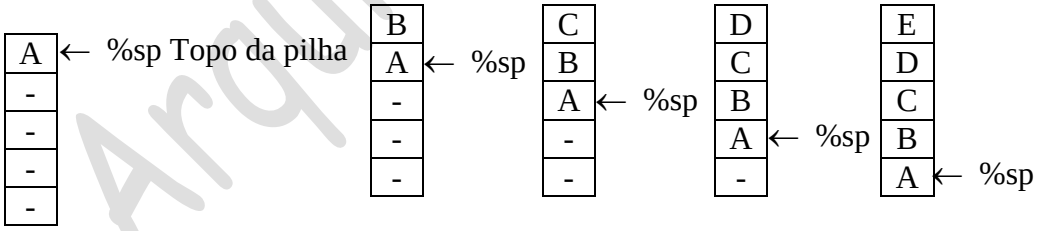


FIFO – (***first-in first-out***) – A operação da pilha é acessada por instruções push e pop, as quais colocam na pilha (stack) conteúdos de registradores e removem esses conteúdos. O topo da pilha recebe os conteúdos e é apontada por um registrador conhecido como apontador de pilha (stack pointer). A operação FIFO, onde o primeiro dado que entra é o primeiro a sair da pilha de memória. O funcionamento do apontador de pilha descreve a operação com detalhes conforme a seguir.

Pilha - Funcionamento

Funciona como no registrador de deslocamento como é apresentado a seguir.

Exemplo: Colocar 5 dados na sequencia A,B,C,D e E,,



Exemplos de programação: Elaborar um programa que soma 5 números da memória nos endereços 2000_H e armazena o valor em 2010_H.

Exemplos de programação: Elaborar um programa que zera 5 posições da memória iniciando no endereço 2000_H. Exemplos de programação: Elaborar um programa que realiza a operação lógica AND entre os números x,y e z alocados na memória na posição 2000_H.

8. Microarquitetura básica - ARC

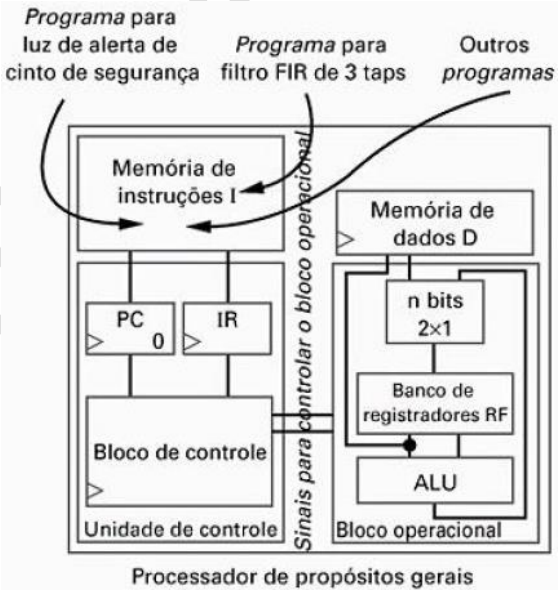
Introdução: A máquina de uso geral ou de propósito geral tem que a tarefa fica armazenada em memória e não no circuito digital. A diferença básica entre uma máquina de uso específico e uma máquina de uso geral está na unidade de controle. Uma máquina de uso específico onde o fluxo de dados é projetado de acordo com a aplicação e as microoperações são geradas pela unidade de controle de acordo com a operação a ser realizada e, portanto, a máquina se torna específica para a aplicação. Para a máquina de uso geral o fluxo de dados e a unidade de controle não alteram com a aplicação e não é preciso projetar a unidade de controle. A diferença básica é que na máquina de uso geral a solução de uma aplicação é feita por instrução a instrução até que a máquina complete todo o programa da aplicação. O usuário transforma o quadro de instrução a qual ele monta para a máquina de uso específico em linhas de programa com instruções que serão executadas passo a passo pela unidade de controle no fluxo de dados.

Arquitetura da máquina de uso geral

Uma máquina de uso geral é conhecida como processador cuja arquitetura interna é composta por uma unidade de controle e fluxo de dados. As instruções são colocadas na memória de instrução, onde o controlador realiza a busca para o controlador decodificar e executar. A seguir apresentamos o processador de uso geral.

ARQUITETURA BÁSICA:

1. Unidade de controle;
2. Fluxo de dados



Bloco Operacional básico

1. Carga de dados:
2. Transformação de dados:
3. Armazenamento dos dados:

Memória de dados:

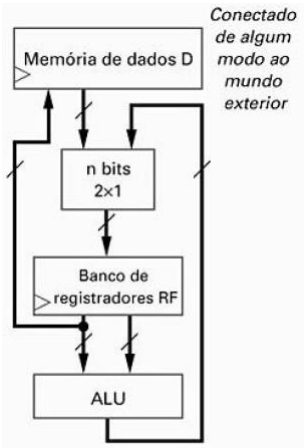


Figura 8.2 Bloco operacional básico de um processador programável.

Operações do bloco operacional –

1. Operação de carga:

2. Operação da ULA:

3. Operação de armazenamento:

Arquitetura de carga e armazenamento

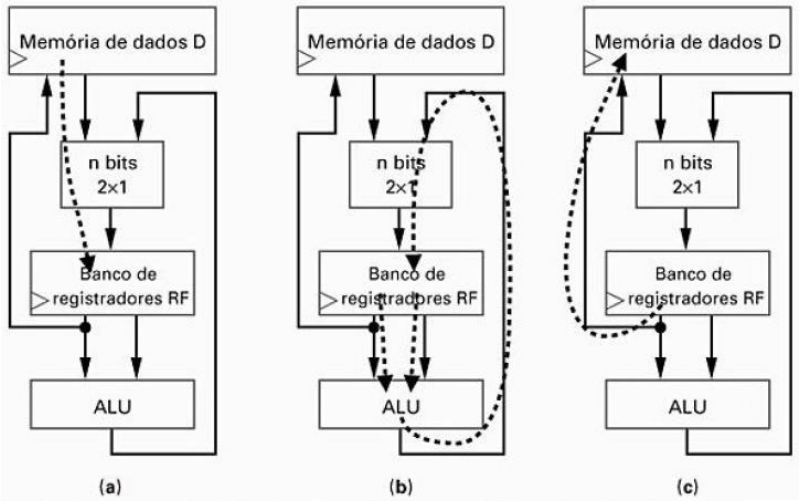


Figura 8.3 Operações básicas do bloco operacional: (a) carga (leitura), (b) operações (transformações) de ALU e (c) armazenamento (escrita).

Quais operações no bloco operacional que consomem somente 01 ciclo de relógio?

1. Copiar dados de uma posição de memória, colocando-os em uma posição de um banco de registradores.
2. Ler dados de duas posições de uma memória de dados, colocando-os em duas posições de um banco de registradores.
3. Somar dados de duas posições de uma memória de dados e armazenar o resultado em uma posição de um banco de registradores.
4. Copiar dados de uma posição de um banco de registradores, colocando-os em outra posição do banco de registradores.
5. Subtrair os dados, que em uma posição de um banco de registradores, de uma posição de uma memória de dados. O resultado será armazenado em uma posição do banco de registradores.

Unidade de controle básica:
Exemplo: Realizar a operação a seguir: D[9] = D[0] + D[1], no bloco operacional indicando os sinais a serem ativos nas operações realizadas. Os dados se encontram armazenados na memória de dados em determinada posição ou endereço de memória e o registrador de arquivos contém pelo menos 3 registradores internos.

Instruções executadas no bloco operacional

1.
2.
3.
4.

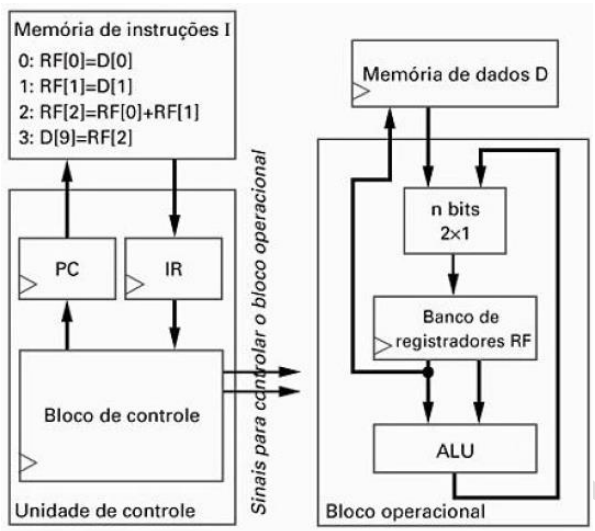


Figura 8.4 A unidade de controle de um processador programável.

ARQUITETURA DA UNIDADE DE CONTROLE

Conforme o diagrama a seguir a U.C. é responsável pela busca da instrução e a geração dos sinais de controle do fluxo de dados. São dois ciclos que a U.C. executa.
1) Ciclo de Busca;
2) Ciclo de Execução.

1.a) CICLO DE BUSCA – O ciclo de busca de uma instrução alocada na memória externa é realizado através do registrador contador de instruções denominado de PC. O conteúdo do PC é o endereço da instrução. O ciclo de busca é dividido em 03 microoperações a saber:

1) O PC endereça a instrução na memória ***Drive do endereço = (PC)***
2) O PC é incrementado para a próxima busca ***PC= PC +1***
3) A instrução é carregada no registrador de instrução ***RI = Instrução***

1.b) CICLO DE EXECUÇÃO – O ciclo de execução da instrução, segue arquitetura RISC (Reduzido Conjunto de Instruções), onde cada execução é realizada por uma única microoperação. Cada instrução se torna desta forma uma microoperação.

A seqüência de operações realizadas pela unidade de controle para executar o ciclo de busca da primeira instrução requere 3 ciclos de relógio a saber:

1. Busca:
2. Decodificação:
3. Execução:

Estágios para o processamento do ciclo de busca e execução da primeira instrução

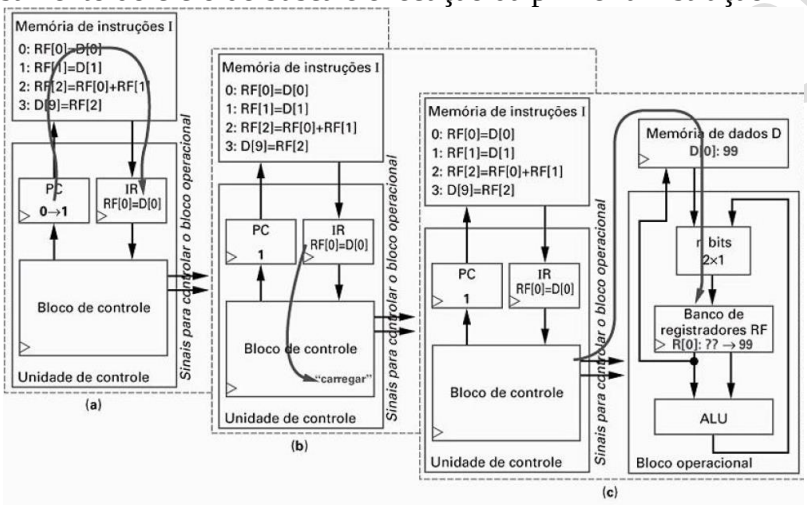


Figura 8.5 Três estágios do processamento de *uma* instrução (a) busca, (b) decodificação e (c) execução.

Total de ciclos de relógio para a execução do programa.

Representação por uma F.S.M. do ciclo de busca e execução.

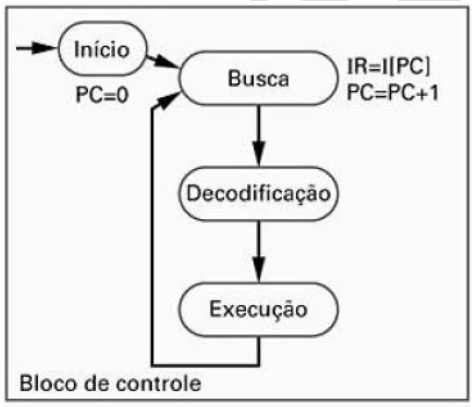


Figura 8.6 Estados básicos do bloco de controle.

Operação da unidade de controle:

1. PC = 0;
2. IR = I[PC]; PC = PC + 1;
3. RF[0] = D[0].

Ciclos de relógio = 03

Exercício: Calcular o número de ciclos de relógio para a execução do programa a seguir o qual realiza a operação: D[3] = D[2] + D[1] + D[0].

Exercício: Calcular o número de ciclos de relógio para a execução do programa a seguir o qual realiza a operação: D[3] = D[2] + D[1] + D[0].

1. Montagem da instrução

instrução	opcode	ra	rb	rc	d	Operação
Carga	0000	0000	0000	0000	0000 0000	RF[0] = D[0]
Carga	0000	0001	0000	0001	0000 0001	RF[1] = D[1]
Carga	0000	0010	0000	0010	0000 0010	RF[2] = D[2]
Soma	0010	0000	0000	0001	0000 0001	RF[0] = RF[0] + RF[1]
Soma	0010	0000	0000	0010	0000 0010	RF[0] = RF[0] + RF[2]
Armazena	0001	000	0000	0011	0000 0011	D[3] = RF[0]

Sinais gerados pela U.C. na execução do programa.

Instrução	Out_D_rd	Out_D_wr	Out_RF_s	Out_RF_w_wr	Out_RF_Rp_rd	RF_Rp_addr	Out_RF_Rq_rd	RF_Rq_addr	RF_w_addr	D_addr	ALU
Carga	1	0	1	1	0	xxxx	0	xxxx	ra	d	000
Soma	0	0	0	1	0	rb	0	rc	ra	d	100
Armazena	0	1	x	0	1	ra	0	xxxx	-	d	000

UNIDADE DE CONTROLE – (ARQUITETURA DA UNIDADE DE CONTROLE)

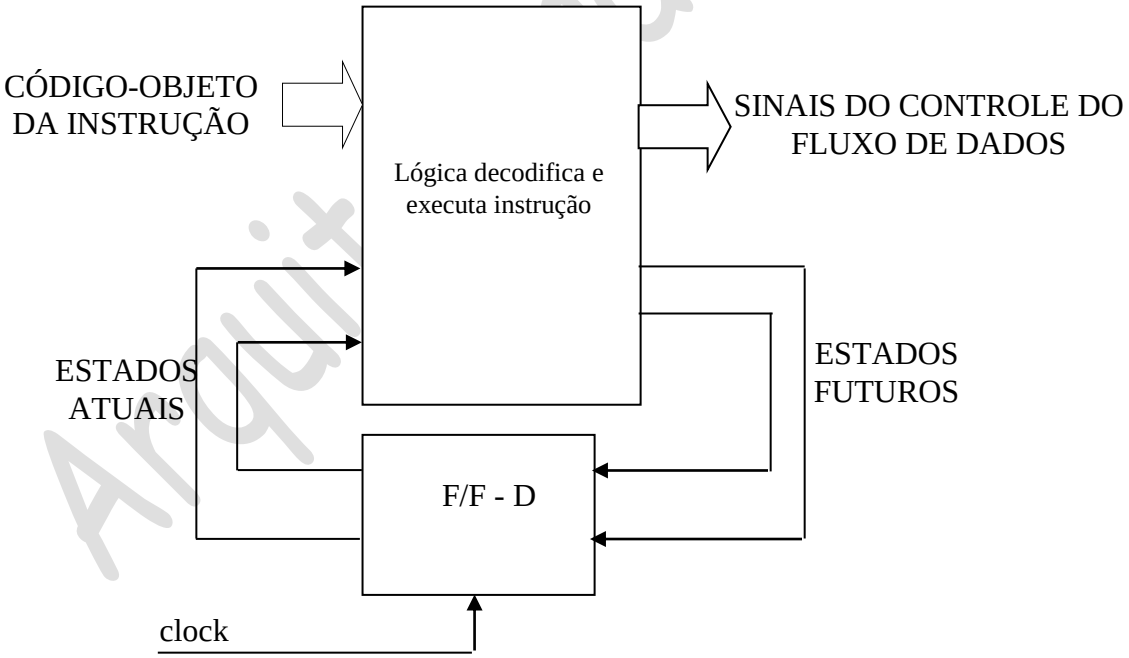
A Unidade de controle é responsável pela busca da instrução e a geração dos sinais de controle do bloco operacional de dados conhecido como fluxo de dados. São dois ciclos que a U.C. executa, a saber:

1) Ciclo de Busca;
2) Ciclo de Execução.

1.a) CICLO DE BUSCA – O ciclo de busca de uma instrução alocada na memória externa é realizado através do registrador contador de instruções denominado de PC. O conteúdo do PC é o endereço da instrução. O ciclo de busca é dividido em 02 microoperações, a saber:

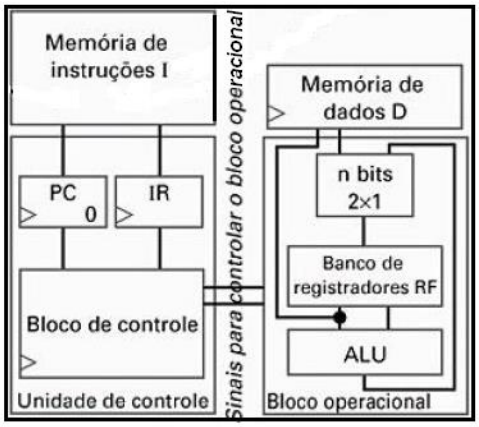
1) O PC endereça a instrução na memória ***Drive do endereço = (PC)***
2) O PC é incrementado para a próxima busca ***PC= PC +1*** e a instrução é carregada no registrador de instrução ***RI =[PC]" Instrução"***.

1.b) CICLO DE EXECUÇÃO – O ciclo de execução da instrução, segue arquitetura RISC (Reduzido Conjunto de Instruções) onde cada execução é realizada por uma única microoperação. Cada instrução se torna desta forma uma microoperação.
As microoperações das instruções são **armazenadas** em uma tabela de dados e opera como uma máquina de estados, cuja entrada externa é identificada pelo código objeto da instrução, o estado atual da máquina e gera o estado futuro e as saídas que são os sinais necessários para o fluxo de dados realizar a execução da instrução em andamento. O diagrama de blocos a seguir mostra um esquema de representação por máquina de estados da unidade de controle.



Arquitetura em bloco da unidade de controle e fluxo de dados.

A seguir é apresentada a arquitetura básica da unidade de controle e o fluxo de dados.

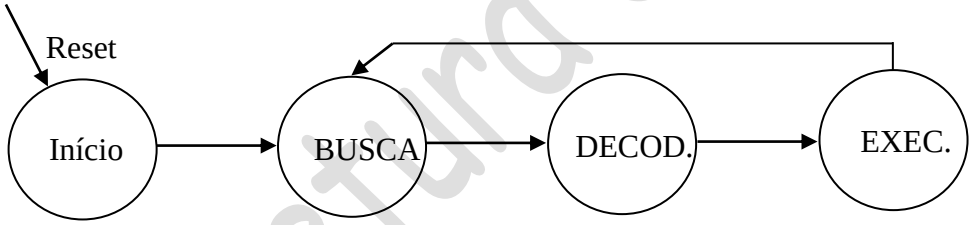


Processador de propósitos gerais

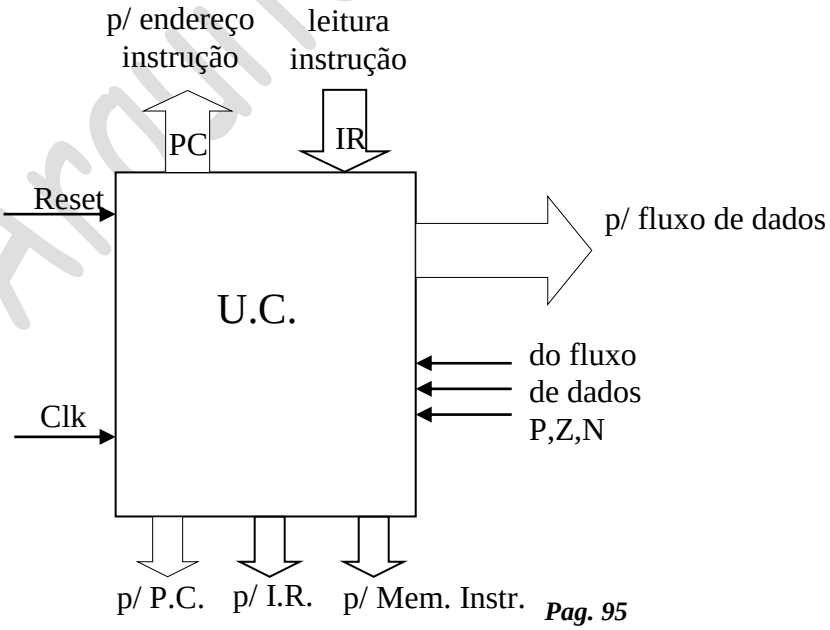
Projeto do processador de uso geral utilizando a arquitetura básica unidade de controle e do fluxo de dados.

Diagrama de estados do controlador.

A seguir é apresentado o diagrama de estados do ciclo de busca da máquina, a qual consiste na busca da instrução, na decodificação da instrução e na execução da instrução.



Projeto da U.C.

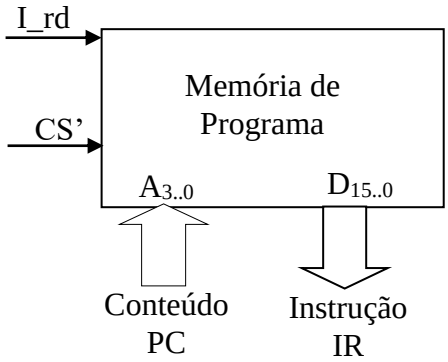


Sinais recebidos e gerados na U.C.

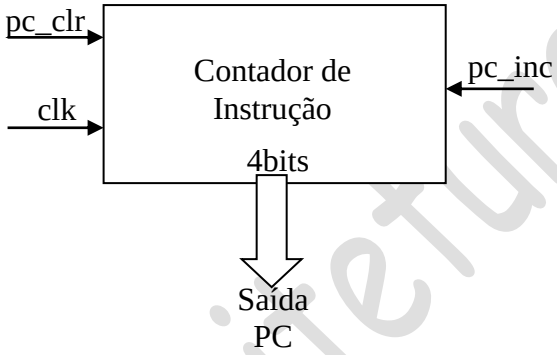
a) Recebidos – P,Z,N, clk, reset, data_Mem (dados da memória de programa).

b) Gerados – $I_{E1.0}$, W_E, $W_{A1..0}$, R_{AE}, $R_{AA1..0}$, $R_{BA1..0}$, $ULA_{2..0}$, $SH_{1..0}$, OE, data PC, data IR, R/W', CS', LD_PC, LD_IR.

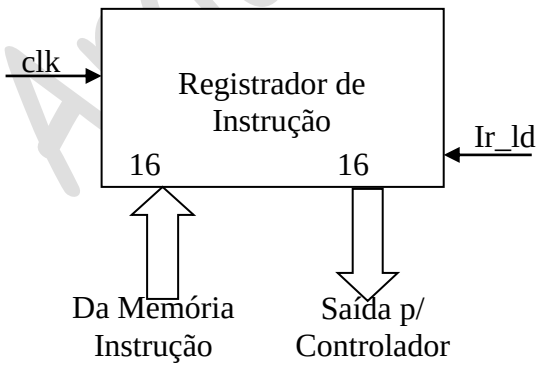
Memória de instruções – Uma memória de 16 x 16 do tipo RAM, sendo 4 bits de endereço por 16 bits de conteúdo. O controle da memória é feito por 2 sinais, sendo um de leitura e escrita e o outro de seleção do dispositivo.



Contador de instrução – É um dispositivo contador síncrono de 4 bits com clock e reset e carga paralela síncrona.

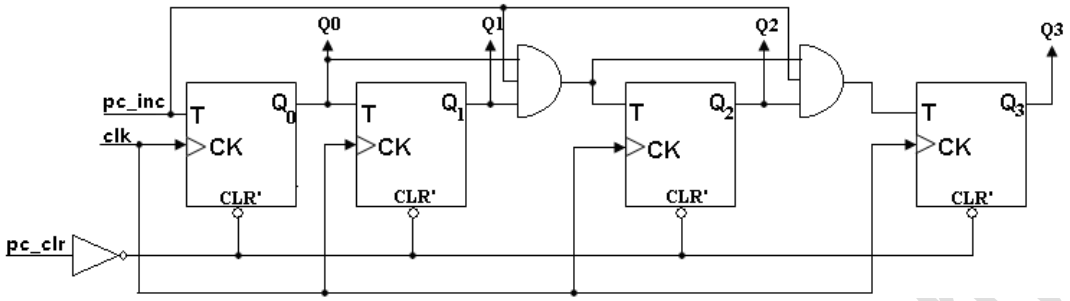


Registrador de instrução – É um dispositivo tipo latch com 8 bits com carga paralela o qual recebe uma instrução da memória de instruções para ser decodificada e executada a seguir.

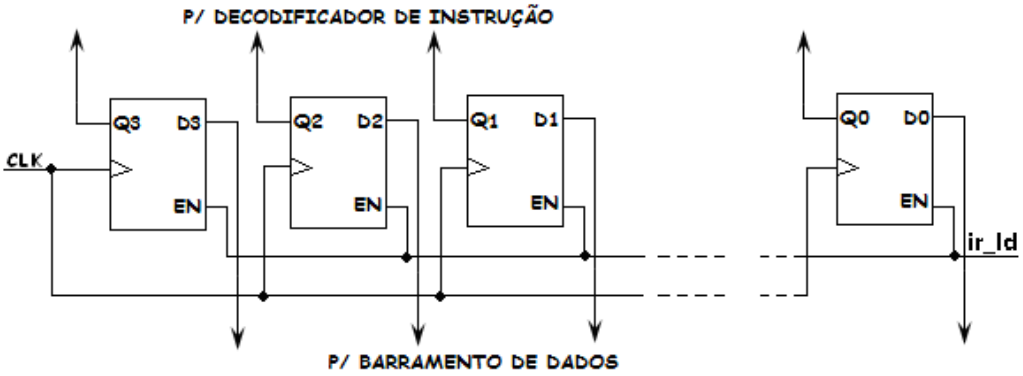


Implementação dos circuitos digitais da unidade de controle.

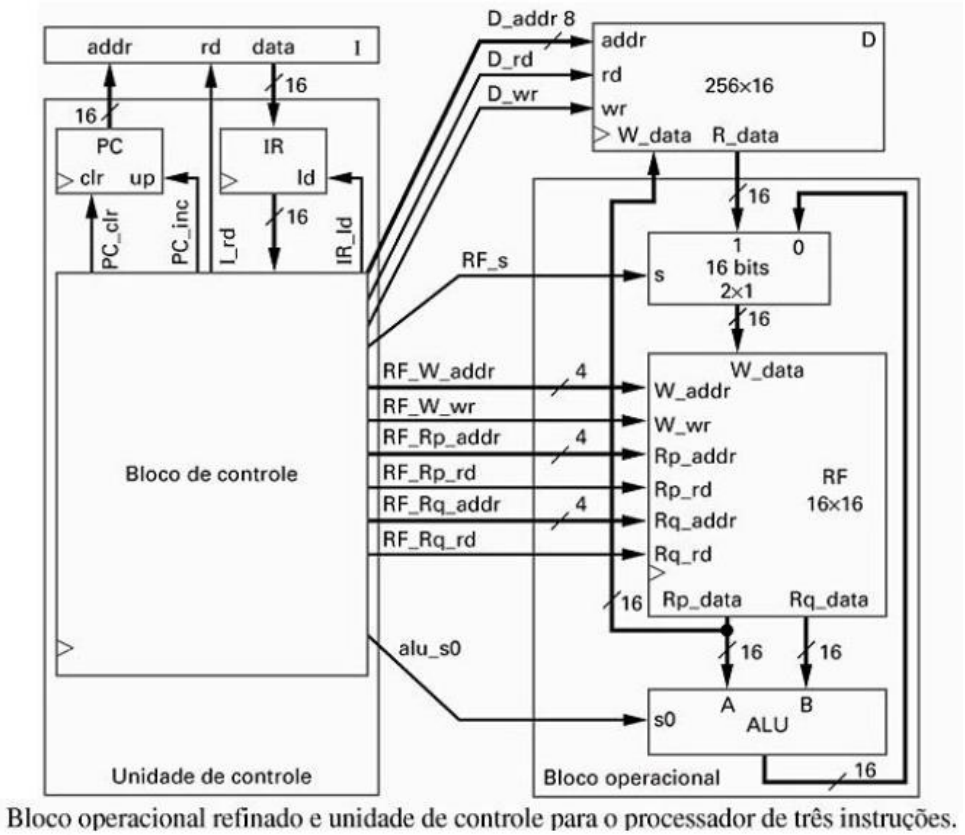
a. Contador de instrução – Contador em de 4 bits com 4 flip-flops do tipo T.



b. Registrador de instrução – Circuito latch de 16 bits de largura.



Bloco operacional e unidade de controle do processador



Bloco operacional refinado e unidade de controle para o processador de três instruções.

Quadro de instruções para o conjunto de 03 instruções. Sendo ra = RF_waddr, rb = RF_Rp_addr, rc = RF_Rq_addr e d = $data_{7..0}$.

item	instrução	pc_clr	pc_inc	ir_ld	i_rd	D_wr	D_rd	RF_s	RF_wr	RF_waddr	RF_Rp_rd	RF_Rp_addr	RF_Rq_rd	RF_Rq_addr	$Alu_{2..0}$
1	Inicio														
2	Busca														
3	Decodificação														
4	Carregar														
5	Somar														
6	Armazenar														

Exercício: Calcular o número de ciclos de relógio para a execução do programa a seguir o qual realiza a operação: D[3] = D[2] + D[1] + D[0].

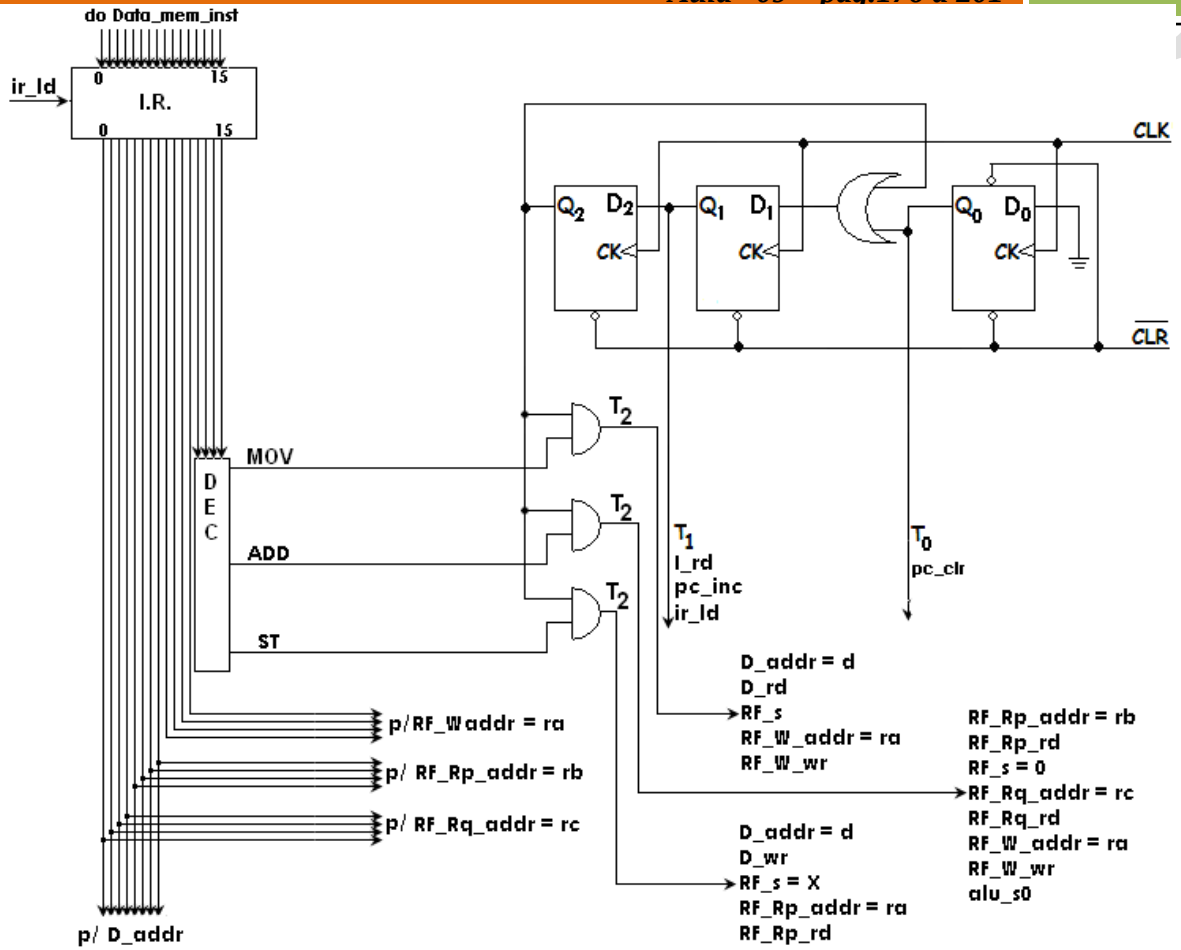
1. Montagem da instrução

instrução	opcode	ra	rb	rc	d	Operação
Carga	0000	0000	0000	0000	0000 0000	RF[0] = D[0]
Carga	0000	0001	0000	0001	0000 0001	RF[1] = D[1]
Carga	0000	0010	0000	0010	0000 0010	RF[2] = D[2]
Soma	0010	0000	0000	0001	0000 0001	RF[0] = RF[0] + RF[1]
Soma	0010	0000	0000	0010	0000 0010	RF[0] = RF[0] + RF[2]
Armazena	0001	000	0000	0011	0000 0011	D[3] = RF[0]

Sinais gerados pela U.C. na execução do programa.

Instrução	Out_D_rd	Out_D_wr	Out_RF_s	Out_RF_w_wr	Out_RF_Rp_rd	RF_Rp_addr	Out_RF_Rq_rd	RF_Rq_addr	RF_w_addr	D_addr	ALU
Carga	1	0	1	1	0	xxxx	0	xxxx	ra	d	000
Soma	0	0	0	1	0	rb	0	rc	ra	d	100
Armazena	0	1	x	0	1	ra	0	xxxx	-	d	000

c. Decodificador de instrução – Abaixo segue o decodificador de instruções com três estados T_0, T_1 e T_2 para três instruções Carregar, Somar e Armazenar.



Unidade de Controle bloco de decodificação da instrução.

DESCRIÇÕES DAS 03 INSTRUÇÕES – A seguir é apresentada uma descrição e formatação das instruções.

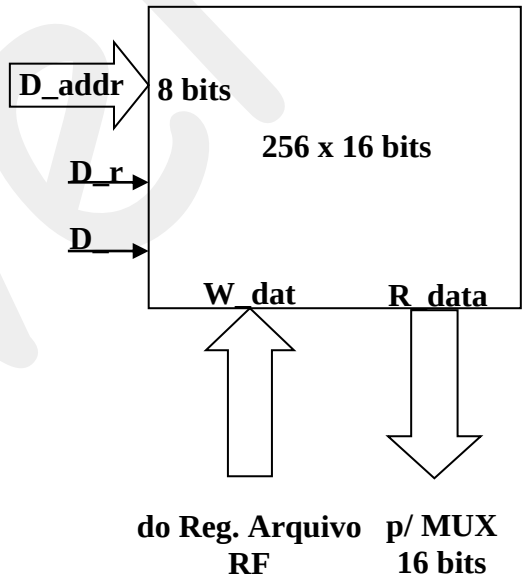
1. Formatação da instrução

Bits	Tipo	Comentário
15...12	Opcde	0000 – Carregar 0001 – Armazenar 0010 – Somar
11...8	ra	0000 – Reg.0 1111 – Reg. 15
7...4	rb	0000 – Reg.0 1111 – Reg. 15
3...0	rc	0000 – Reg.0 1111 – Reg. 15
7...0	d	Posição 0 a 255

2. Instrução Carregar – 0000 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$ – Onde é definido por:

0000 – Opcode da instrução carregar.
$r_3r_2r_1r_0$ – Destino dos dados. $d_7d_6d_5d_4d_3d_2d_1d_0$ – Fonte dos dados.

r_3	r_2	r_1	r_0	Registrador destino
0	0	0	0	Reg. 0
0	0	0	1	Reg.1
0	0	1	0	Reg.2
0	0	1	1	Reg. 3
0	1	0	0	Reg. 4
0	1	0	1	Reg.5
0	1	1	0	Reg.6
0	1	1	1	Reg.7
1	0	0	0	Reg.8
1	0	0	1	Reg. 9
1	0	1	0	Reg. 10
1	0	1	1	Reg. 11
1	1	0	0	Reg.12
1	1	0	1	Reg.13
1	1	1	0	Reg.14
1	1	1	1	Reg.15



3. Instrução Armazenar - 0001 $r_3r_2r_1r_0$ $d_7d_6d_5d_4d_3d_2d_1d_0$ – Onde é definido por:

0001 – Opcode da instrução armazenar.
$r_3r_2r_1r_0$ – Fonte dos dados. $d_7d_6d_5d_4d_3d_2d_1d_0$ – Destino dos dados.

Conforme tabela acima os dados são transferidos do registrador fonte dado por $\mathbf{r_3r_2r_1r_0}$ para a posição de memória dada por $\mathbf{d_7d_6d_5d_4d_3d_2d_1d_0}$.

4. Instrução Somar – 0010 $\mathbf{ra_3ra_2ra_1ra_0}$ $\mathbf{rb_3rb_2rb_1rb_0}$ $\mathbf{rc_3rc_2rc_1rc_0}$. Onde é definido por:

0010 – Opcode da instrução somar.
$\mathbf{ra_3ra_2ra_1ra_0}$ – Destino do conteúdo executado pela instrução.
$\mathbf{rb_3rb_2rb_1rb_0}$ – Fonte dos dados.
$\mathbf{rc_3rc_2rc_1rc_0}$.- Fonte dos dados.

EXEMPLO: Programar o processador para a execução de cálculo da expressão a seguir:
Somar o conteúdo dos endereços de memória a seguir: D[5] = D[5] + D[6] + D[7] e armazenar no endereço 5 de memória. Pede-se o programa realizado na memória de instruções do processador.

0: 0000 0000 00000101 // Carrega o registrador 0 com o conteúdo do endereço 5: RF[0] = D[5].
1: 0000 0001 00000110 // Carrega o registrador 1com o conteúdo do endereço 6: RF[1] = D[6].
2: 0000 0010 00000111 // Carrega o registrador 2 com o conteúdo do endereço 7: RF[2] = D[7].
3: 0010 0000 00000001 // Somar os conteúdos dos registradores 0 e 1 e armazenar no reg.0
RF[0] = RF[0] + RF[1];
4: 0010 000000000010 // Somar os conteúdos dos registrasdores 0 e 2 e armazenar no reg.0
RF[0] = RF[0] + RF[2];
5: 0001 0000 00000101 // Armazenar o conteúdo do reg.0 no endereço de memória 5:
D[5] = RF[0].

UNIDADE DE CONTROLE DO PROCESSADOR DE 03 INSTRUÇÕES

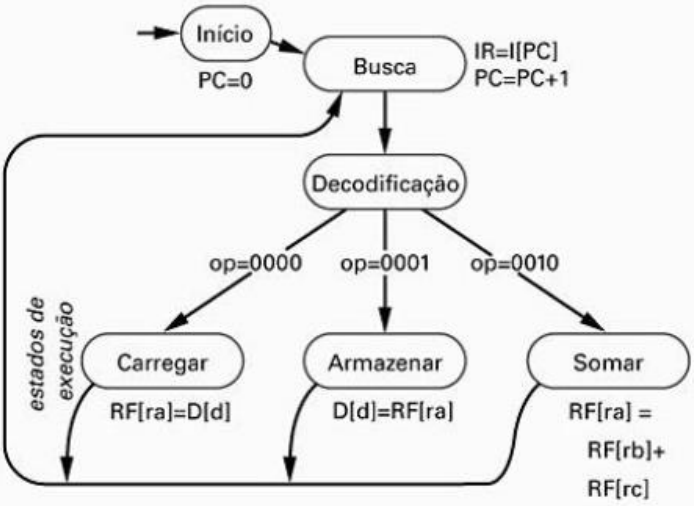


Figura 8.9 Descrição de um processador programável de três instruções por meio de uma máquina de estados de alto nível.

EXEMPLO: DADOS O PROGRAMA DE EXECUÇÃO A SEGUIR:
Linha 0 – RF[0] = D[0]
Linha 1 – RF[1] = D[1]
Linha 2 – RF[2] = RF[0] + RF[1]
Linha 3 – D[9] = RF[2]

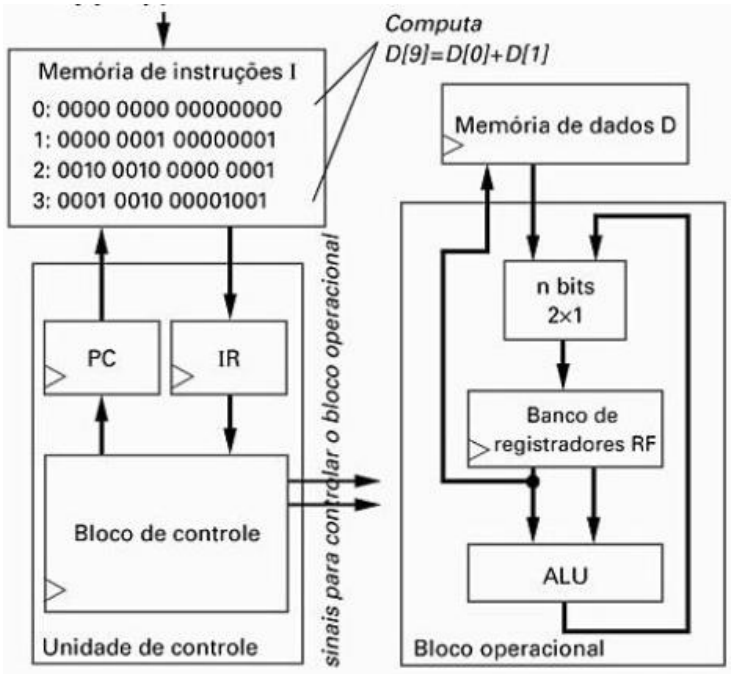


Figura 8.7 Um programa que computa *D[9]=D[0]+D[1]*, usando um dado conjunto de instruções. Inserimos espaços em branco entre os bits da memória de instruções apenas por legibilidade – esses espaços não existem na memória.

F.S.M para o bloco de controle do processador para as 03 instruções.

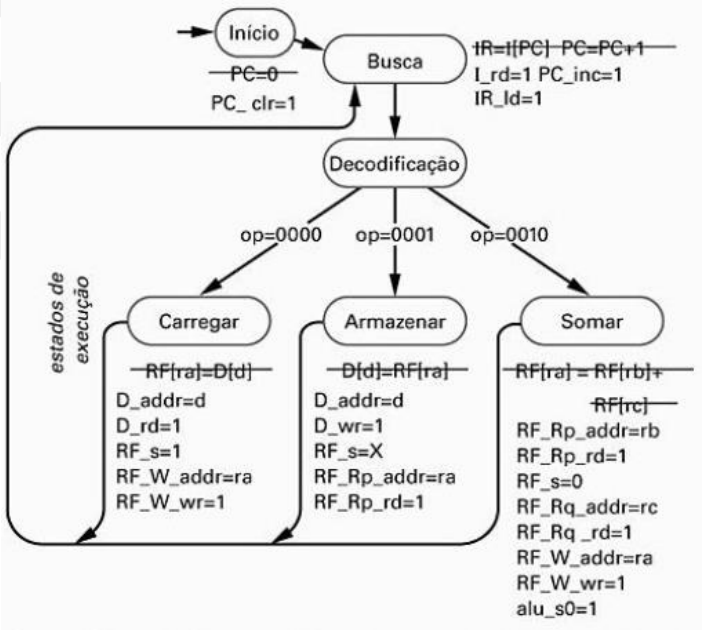
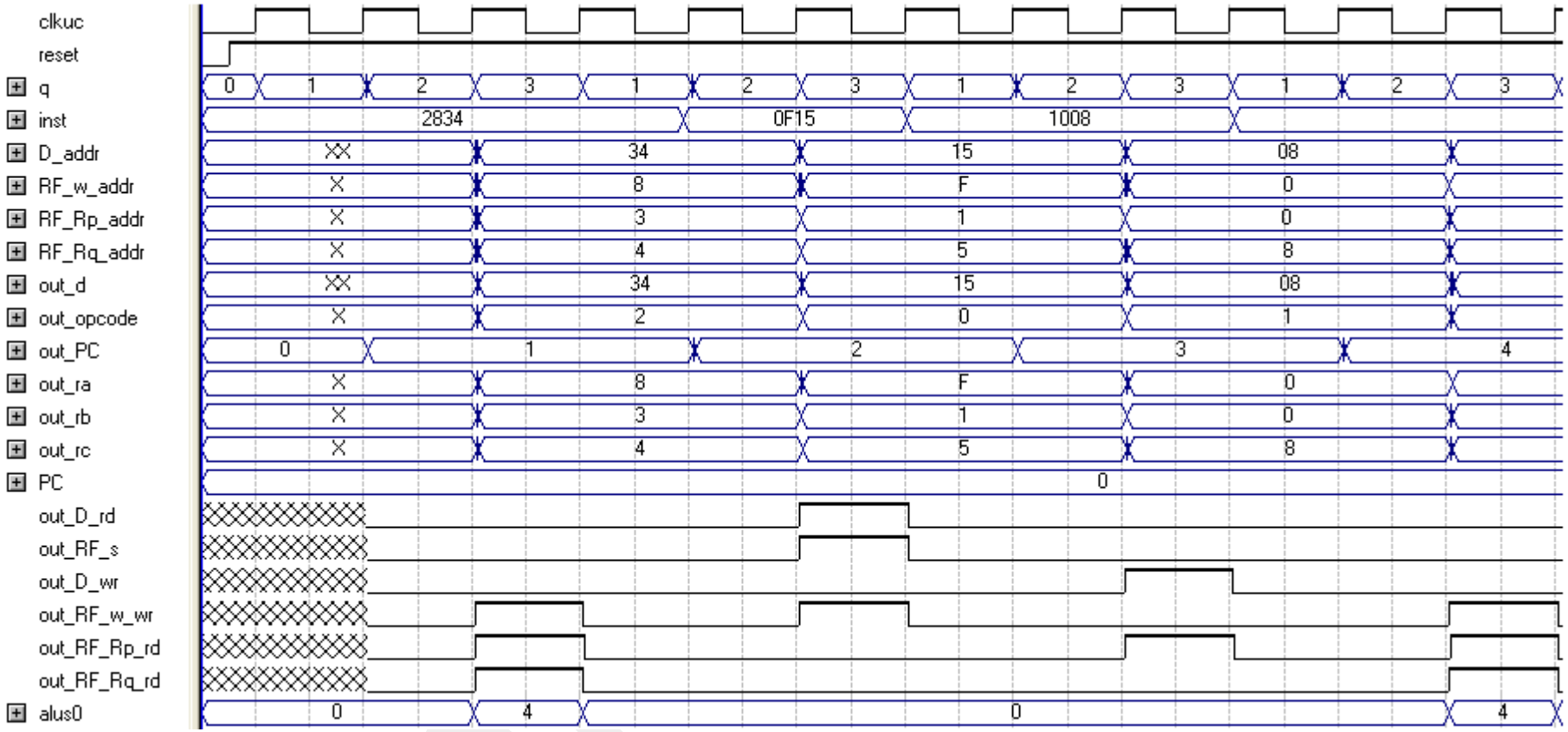
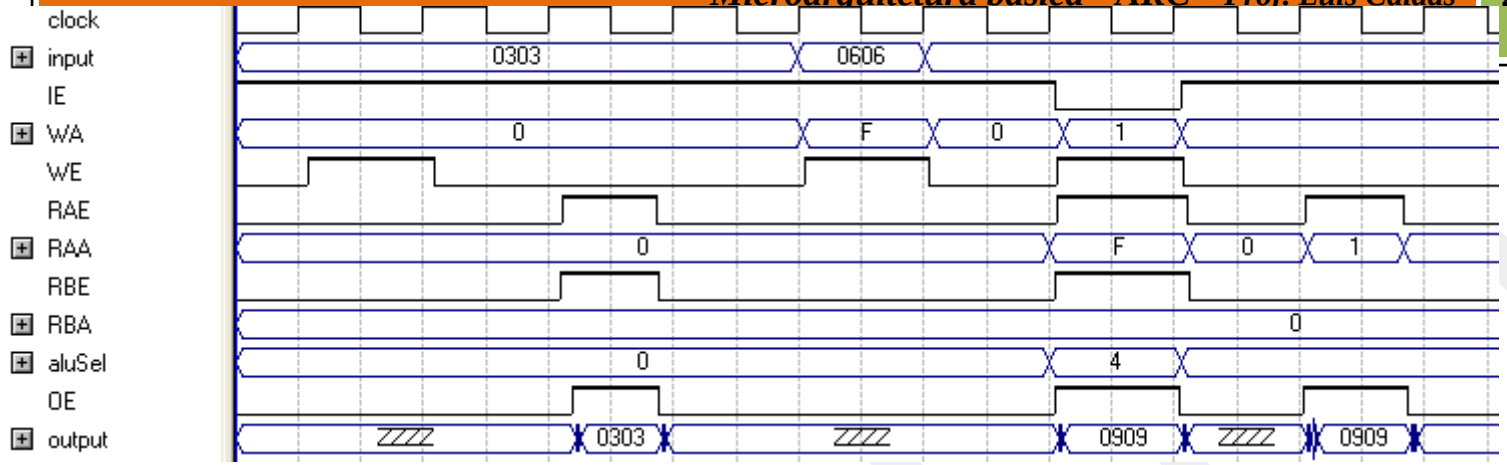


Figura 8.11 FSM para o bloco de controle do processador de três instruções.

Formas de ondas para 03 instruções: Carregar, Armazenar e Somar e a instrução de 16bits sendo bits: opcode(15..12), ra(11..8), rb(7..4), rc(3..0), d(7..0).
Opcode das instruções: 0000 – Carregar, 0001 – Armazenar e 0010 – Somar. Exemplo: Instrução 2834 => opcode 2, ra = 8 e Rb = 3H e rc = 4H, alus0 = 4H.
A instrução pede: RF(8) ← RF(3) + RF(4) e instrução 1005 pede: D(05) = RF(0) e instrução 0F15 pede: RF(F) = D(15).
0F15 => opcode 0,



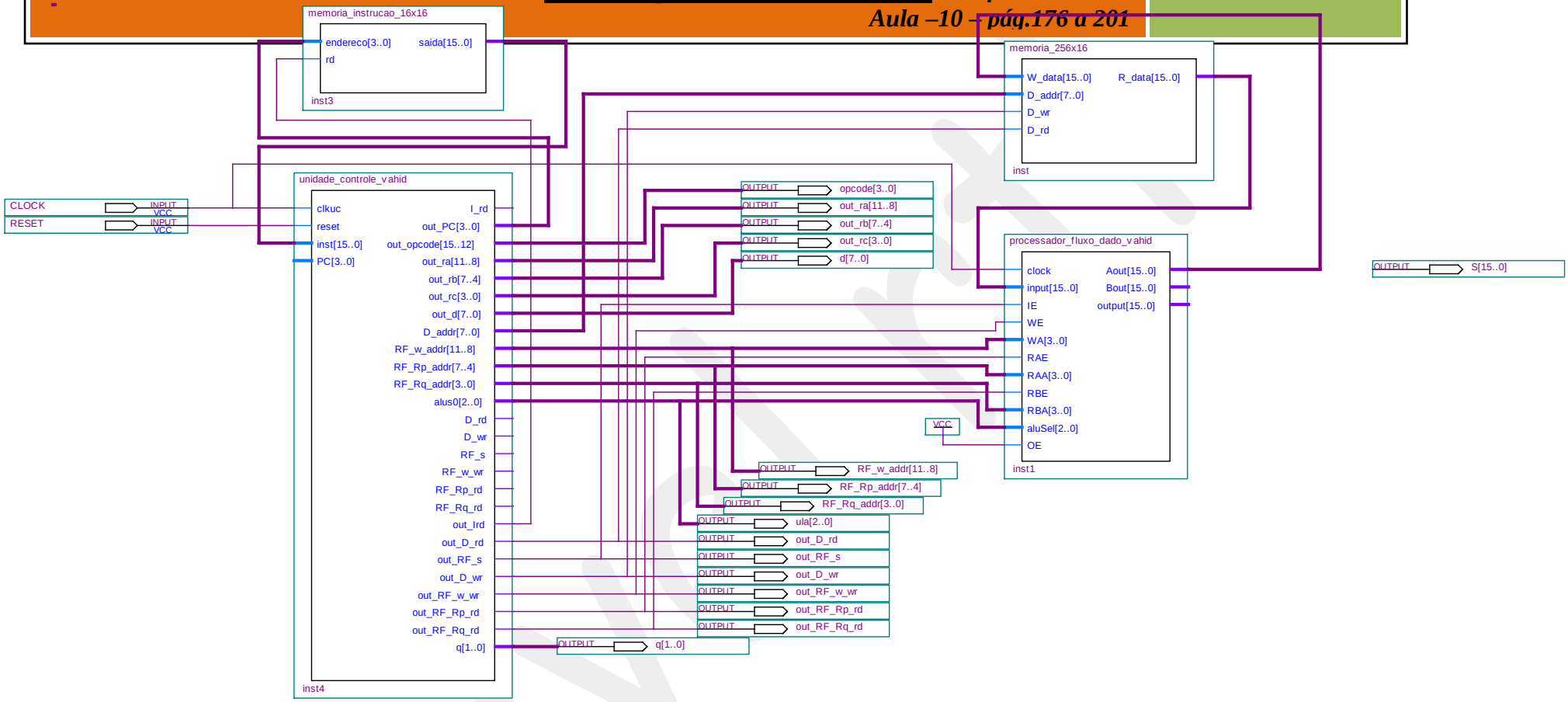
Formas de ondas para o fluxo de dados para duas instruções de carga e uma instrução de soma e instrução de armazenar. A primeira instrução carregar: RF[0} = 0303 e a segunda instrução carregar: RF[15] = 0606. A instrução somar e armazenar: RF[1] = RF[0] + RF[15].
Obs.: Foi introduzido no fluxo de dados uma saída para a simulação das instruções e conferência dos resultados.



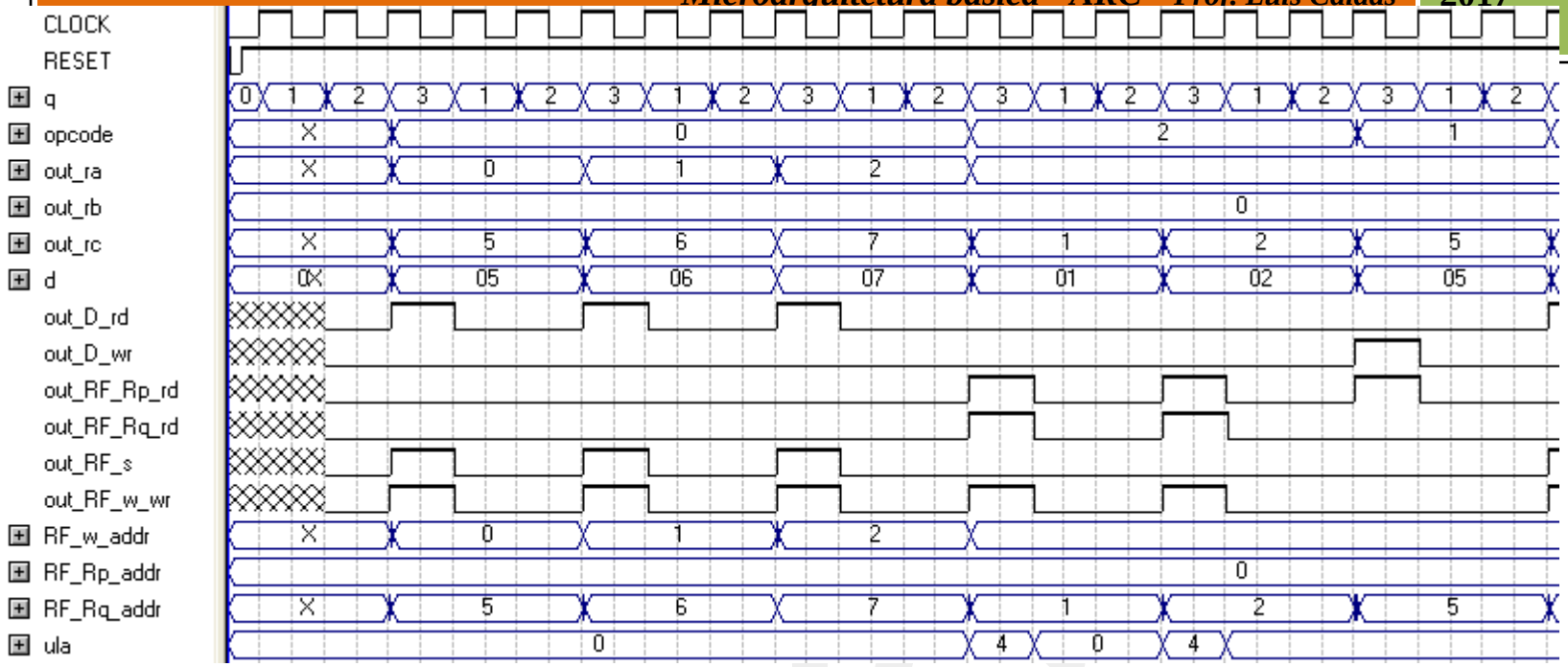
Programa a ser executado no processador gera os sinais para o fluxo de dados de acordo com a tabela a seguir.

Linha	instrução	Opcode	Mnem.	D_addr	D_rd	D_wr	RF_s	RF_w_addr	RF_w_wr	RF_Rp_rd	RF_Rp_addr	RF_Rq_rd	RF_Rq_addr	Ula
0	0005	0	RF[0] = D[5]	05	1	0	1	0000	1	0	0000	0	0000	000
1	0106	0	RF[1] = D[6]	06	1	0	1	0001	1	0	0000	0	0000	000
2	0207	0	RF[2] = D[7]	07	1	0	1	0002	1	0	0000	0	0000	000
3	2001	2	RF[0] = RF[0] + RF[1],	xx	0	0	0	0000	1	1	0000	1	0001	100
4	2002	2	RF[0] = RF[0] + RF[2],	xx	0	0	0	0000	1	1	0000	1	0010	100
5	: 1005	1	D[5] = RF[0].	05	0	1	x	0000	0	1	0000	0	0000	000

Circuito do processador para 03 instruções com blocos de controlador unidade de controle, bloco memória de instruções, bloco fluxo de dados e bloco de memória de dados.



O programa da fig. 8.8 é mostrado nas formas de ondas abaixo, onde foi rodado no processador Vahid.


CLOCK
RESET
q
opcode
out_ra
out_rb
out_rc
d
out_D_rd
out_D_wr
out_RF_Rp_rd
out_RF_Rq_rd
out_RF_s
out_RF_w_wr
RF_w_addr
RF_Rp_addr
RF_Rq_addr
ula

PROCESSADORES PROGRAMÁVEIS PARA 32 INSTRUÇÕES

O projeto de 32 instruções é uma extensão máxima do projeto do processador programável de 03 instruções. A arquitetura básica do processador de 03 instruções é modificada para que programas que contenham blocos de decisões fossem introduzidas e com isso a implementação de algorítmos mais complexos. A introdução do bloco "detetor de zero" vai permitir a resposta da ULA com operações cujo resultado produz um valor zero na saída, bem como o bloco comparador de amplitude que gera 03 saídas "maior, igual e menor" e um segundo bloco comparador o qual produz 02 saídas maior ou igual e menor ou igual. Todos esses indicadores servirão para produzirem um "status" da operação realizada na ULA ou nos testes de comparações. A tabela a seguir define todos os indicadores "flags" e a sua lógica ativa.

Indicador	Flag	Comentário
Zero	Z	Z = "1" qdo a operação realizada na ULA produz saída igual a zero.
Maior	M	M = "1" qdo a operação de comparação realizada entre 2 operandos do registrador de arquivos produzir a saída maior a "1".
Menor	N	N = "1" qdo a operação de comparação realizada entre 2 operandos do registrador de arquivos produzir a saída menor a "1".
Igual	I	I = "1" qdo a operação de comparação realizada entre 2 operandos do registrador de arquivos produzir a saída igual a "1".
Maior ou igual	MEQ	MEQ = "1" qdo a operação de comparação realizada entre 2 operandos do registrador de arquivos produzir a saída maior ou igual a "1".
Menor ou igual	NEQ	NEQ = "1" qdo a operação de comparação realizada entre 2 operandos do registrador de arquivos produzir a saída menor ou igual a "1".
Bit_DO	B0	B0 = "1" qdo o bit LSB do deslocador é igual a "1".
Bit_D7	B7	B7 = "1" qdo o bit MSB do deslocador é igual a "1".
Carry	C	C = "1" qdo a operação realizada na ULA estoura a sua capacidade máxima.
Overflow	OV	OV = "1" qdo a operação entre 2 operandos assinalados realizada na ULA produz um transbordamento na ULA.

Os indicadores informam a unidade de controle sobre o "status" da operação na execução da instrução. O programa feito pelo usuário deve utilizar essa informação para a tomada de decisão. A instrução de Salto foi expandida para 12 novas instruções e 4 bits seguidos dos 4 bits fixos do "opcode" servirão para a criação dessas instruções de saltos a seguir.

Expansão	Mnem.	Operação	Comentário
0000	JZ	Salto se zero	Se a operação realizada na ULA produz zero;
0001	JNZ	Salto se não zero	Se a operação realizada na ULA produz um resultado diferente de zero;
0010	-	-	-
0011	JNEQ	Salto se menor ou igual	Salto se a operação de comparação entre 2 números resulta em menor ou igual;
0100	-	-	-
0101	JMEQ	Salto se maior ou igual	Salto se a operação de comparação entre 2 números resulta em maior ou igual;
0110	-	-	-
0111	-	-	-
1000	JB0	Salto de bit_D0	Salto se bit D0 do deslocador é igual a "1";

1001	JNB0	Salto se bit_NDO	Salto se bit D0 do deslocador é igual a "0";
1010	JB7	Salto se bit_D7	Salto se bit D7 do deslocador é igual a "1";
1011	JNB7	Salto se bit_ND7	Salto se bit D7 do deslocador é igual a "0";
1100	JM	Salto se maior	Salto se a operação de comparação entre 2 números resulta em maior;
1101	JI	Salto se igual	Salto se a operação de comparação entre 2 números resulta em igual;
1110	JN	Salto se menor	Salto se a operação de comparação entre 2 números resulta em menor;
1111	JMP	Salto incondicional	Salto incondicionalmente

O conjunto de instruções com 32 instruções é apresentado a seguir, bem como a tabela com os "opcode" e comentários.

Opcode	Mnem	Operação	Comentário
0000	LD(ra),(d)	RF[ra] ← (d)	Carrega o conteúdo endereçado em d para o RF[ra];
0001	ST(d),(ra)	(d) ← RF[ra]	Armazena o conteúdo de RF[ra] no endereço (d);
0010	ADD(ra),(rb),(rc)	RF[ra] = RF[rb] + RF[rc]	Soma os conteúdos dados por RF[rb,rc] e armazena em RF[ra];
0011	MVI(ra) #c	RF[ra] ← w_data	Transfere o conteúdo imediato em RF[ra];
0100	SUB(ra),(Rb),(rc)	RF[ra] = RF[rb] - RF[rc]	Subtrai os conteúdos dados por RF[rb,rc] e armazena em RF[ra];
0101	JX, offset	X = 0000 a 1111	Salta para o endereço dado em offset;
0110	OUT(ra)	Saída ← RF[ra]	Transfere para a saída o conteúdo de RF[ra];
0111	IN(ra)	RF[ra] ← input_ext	Transfere da entrada o conteúdo para RF[ra];
1000	CMP(ra),(rb)	RF[ra] : RF[Rb]	Compara os conteúdos de RF[ra] com RF[rb];
1001	XOR(ra),(Rb),(rc)	RF[ra] = RF[rb] ⊕ RF[rc]	XOR os conteúdos dados por RF[rb,rc] e armazena em RF[ra];
1010	AND(ra),(Rb),(rc)	RF[ra] = RF[rb].RF[rc]	AND os conteúdos dados por RF[rb,rc] e armazena em RF[ra];
1011	OR(ra),(Rb),(rc)	RF[ra] = RF[rb] + RF[rc]	OR os conteúdos dados por RF[rb,rc] e armazena em RF[ra];
1100	INC(ra)	RF[ra] = RF[ra] - 1	Incrementa o conteúdo de RF[ra];
1101	DEC(ra)	RF[ra] = RF[ra] – 1	Decrementa o conteúdo de RF[ra];
1110	SHX	RF[ra] ← bit	Desloca o conteúdo RF[ra] bit;
1111	HLT	(PC) = (PC)	Pára o processamento.

Assim como foi projetado para a instrução SALTO é adotado o mesmo procedimento para as instruções de deslocamentos, as quais podem ser do tipo: deslocamento para esquerda, para a direita, circular a esquerda e circular a direita. A tabela a seguir apresenta a expansão da instrução SHX para 04 instruções a seguir.

Expansão	Mnem.	Lógica	Comentário

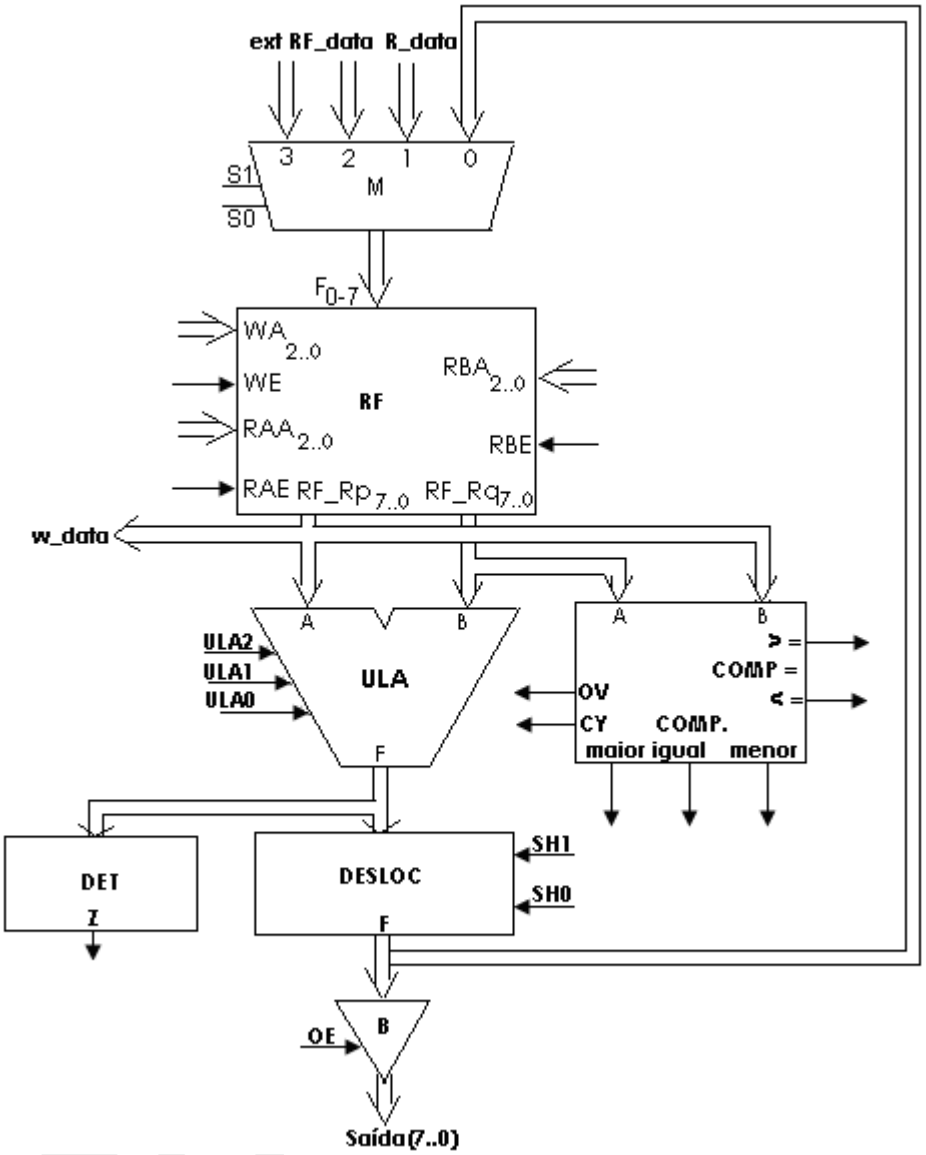
0000	SHL	RF[ra] ← 0	Deslocamento a esquerda do conteúdo do deslocador um bit e introdução do bit D0 igual a zero.
0001	SHR	RF[ra] ← 0	Deslocamento a direita do conteúdo do deslocador um bit e introdução do bit D7 igual a zero.
0010	ROL	RF[ra] ← D7	Deslocamento circular a esquerda do conteúdo do deslocador um bit e introdução do bit D7.
0011	ROR	RF[ra] ← D0	Deslocamento circular a direita do conteúdo do deslocador um bit e introdução do bit D0.

Projeto do fluxo de dados para o processador de 32 instruções.

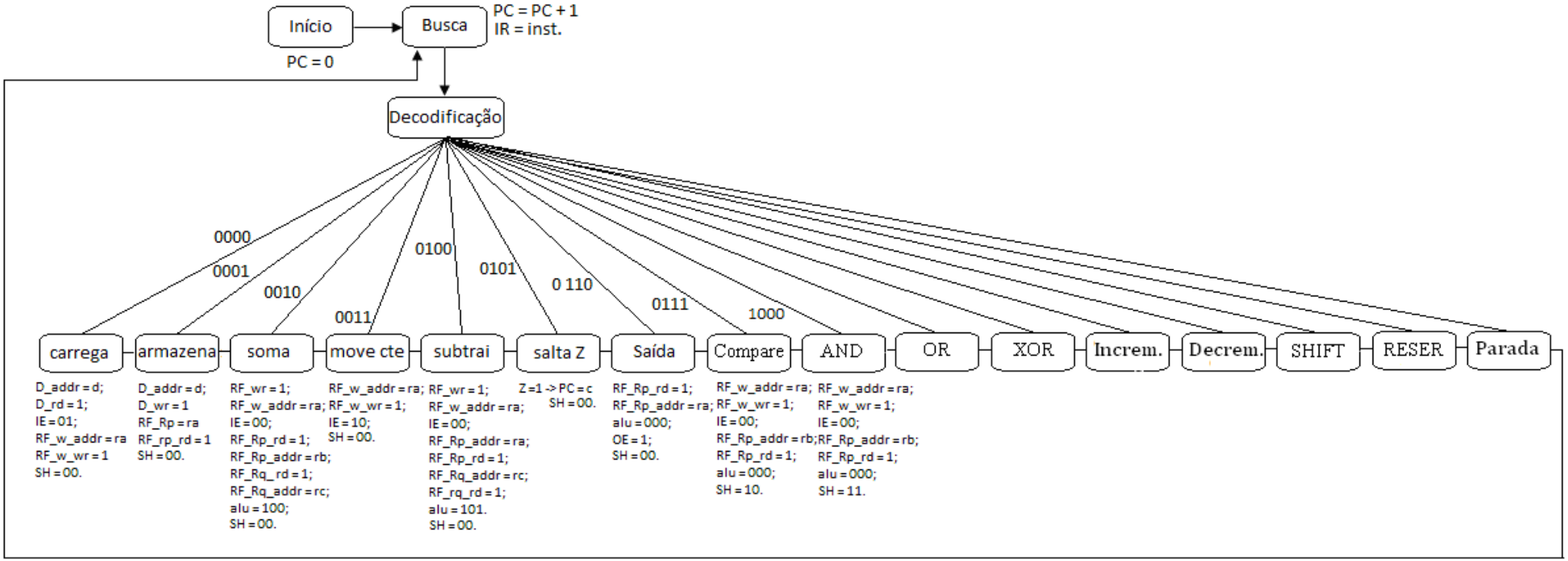
O projeto do fluxo de dados é uma melhoria na arquitetura do processador de 03 instruções apresentado nesse capítulo. Para que fossem expandido o conjunto de instruções, a introdução dos indicadores para a U.C. é fundamental para a tomada de decisão. Os blocos a seguir fazem parte do fluxo de dados para 32 instruções. A saber:

1. Bloco MUX;
2. Bloco RF;
3. Bloco comparador;
4. Bloco detetor de zero;
5. Bloco comparador com igualdade;
6. Bloco ULA;
7. Bloco deslocador;
8. Bloco saída.

Arquitetura do processador de 32 instruções


ext RF_data R_data
3
2
1
0
M
S1
S0
F0-7
WA2..0
WE
RF
RBA2..0
RAA2..0
RBE
RAE RF_Rp7..0
RF_Rq7..0
w_data
A
B
ULA2
ULA1
ULA0
ULA
F
A
B
> =
COMP =
< =
OV
CY
COMP.
maior igual menor
DET
Z
DESLOC
SH1
SH0
F
OE
B
Saída(7..0)

O diagrama de estados a seguir mostra a busca da instrução, decodificação da instrução e execução.



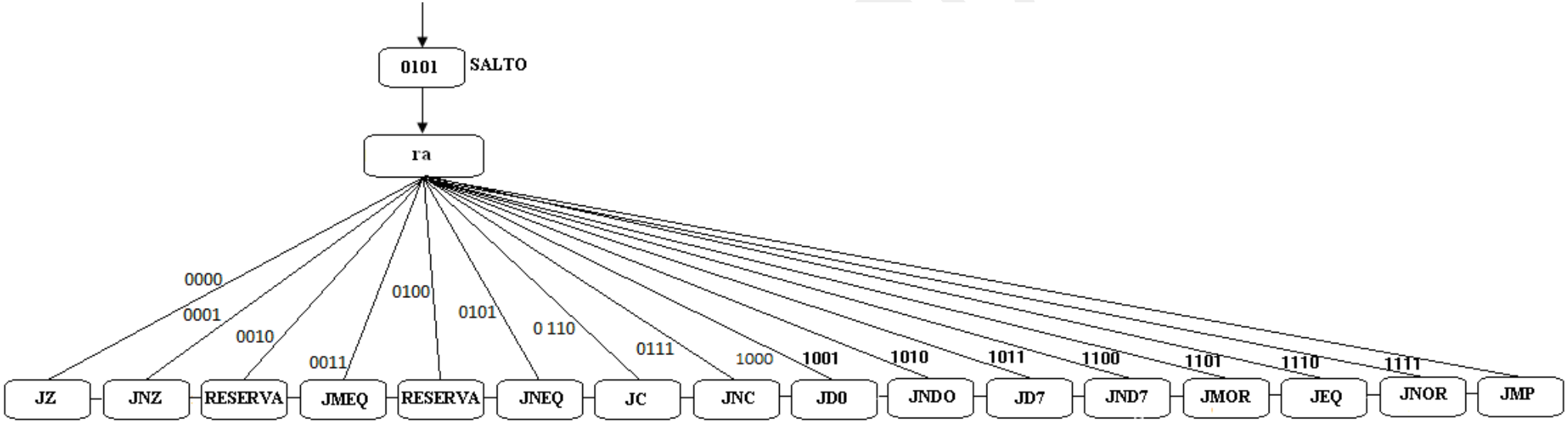
O quadro das palavras de controle geradas pela u.c. para a execução da instrução.

Instrução	Opcode	D_addr	D_rd	D_wr	RF_s	RF_w_addr	RF_w_wr	RF_Rp_rd	RF_Rp_addr	RF_Rq_rd	RF_Rq_addr	ALU	SH	OE
Carga	0	d	1	0	1	ra	1	0	xxxx	0	xxxx	xxx	xx	0
Armazena	1	d	0	1	x	xxxx	0	1	ra	0	xxxx	xxx	xx	0
Soma	2	x..x	0	0	0	ra	1	1	rb	1	rc	100	00	0
Transfer	3	x..x	0	0	2	ra	1	0	xxxx	0	xxxx	Xxx	xx	0
Subtrai	4	x..x	0	0	0	ra	1	1	rb	1	rc	101	00	0
Salto	5	x..x	0	0	0	xxxx	0	0	xxxx	0	xxxx	xxx	xx	x
Saída	6	x..x	0	0	0	xxxx	0	1	ra	0	xxxx	000	00	1
Entrada	7	x..x	0	0	3	ra	1	0	xxxx	0	xxxx	xxx	xx	0
Compare	8	x..x	0	0	0	xxxx	0	1	ra	1	Rb	xxx	xx	0

Instrução	Opcode	D_addr	D_rd	D_wr	RF_s	RF_w_addr	RF_w_wr	RF_Rp_rd	RF_Rp_addr	RF_Rq_rd	RF_Rq_addr	ALU	SH	OE

XOR	9	0	0	0	0	ra	1	1	rb	1	rc	011	00	0
AND	A	0	0	0	0	ra	1	1	rb	1	rc	001	00	0
OR	B	0	0	0	0	ra	1	1	rb	1	rc	010	00	0
INC	C	0	0	0	0	ra	1	1	ra	0	xxxx	111	00	0
DEC	D	0	0	0	0	ra	1	1	ra	0	xxxx	110	00	0
SHIFT	E	-	-	-	-	-	-	-	-	-	-	-	-	-
STOP	F	0	0	0	0	xxxx	0	0	xxxx	0	xxxx	xxx	xx	x

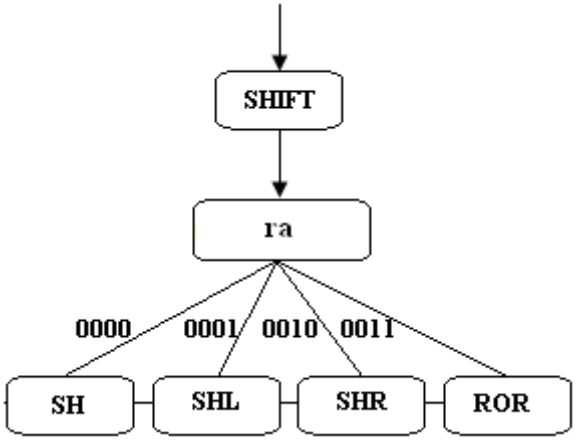
As instruções "salto" e "shift" vão ser tratadas separadamente, pois fazem parte da expansão das instruções. A instrução salto se expande em 12 instruções a saber. O diagrama de estados a seguir mostra as instruções expandidas.



Quadro das palavras de controle geradas pela u.c. nas instruções de saltos.
Na execução da instrução de salto tanto condicional como incondicional os bits da instrução $d_7...d_0$ são carregados no PC se satisfeita a condição ou se a instrução for incondicional.

Instrução	JZ	JNZ	JMEQ	JNEQ	JC	JNC	JD0	JND0	JD7	JND7	JMOR	JEQ	JNOR	JMP
ra	0	1	3	5	6	7	8	9	A	B	C	D	E	F
PC	d	d	d	d	D	d	d	d	d	d	d	d	d	d

A instrução de deslocamento "shift" também será expandida como a seguir:



Quadro de palavras de controle gerados pela u.c. na execução da instrução de deslocamento "shift".

Instrução	ra	D_addr	D_rd	D_wr	RF_s	RF_w_addr	RF_w_wr	RF_Rp_rd	RF_Rp_addr	RF_Rq_rd	RF_Rq_addr	ALU	SH	OE
SH	0	0	0	0	0	rb	1	1	rc	0	xxxx	000	00	0
SHL	1	0	0	0	0	rb	1	1	rc	0	xxxx	000	01	0
SHR	2	0	0	0	0	rb	1	1	rc	0	xxxx	001	10	0
ROR	3	0	0	0	0	rb	1	1	rc	0	xxxx	111	11	0

Formato da instrução – A instrução tem 16bits subdivididos de 4 em 4 para informação de fonte e destino. Quando se utiliza a memória de dados são necessários 8bits para o endereçamento e a união das fontes rb e rc forma o d.

Opcode Instrução	Destino ra	Fonte rb	Fonte rc
$d_{15}..d_0$	$d_{11}...d_8$	$d_7,,d_4$	$d_3..d_0$

d(8bits) = rb(4bits) e rc(4bits).

Descrição de cada instrução.

1. LD(ra),(d) - Carrega da memória de dados o conteúdo endereçado pelo parâmetro d e transfere para o registrador de arquivo endereçado pelo endereço dado por ra.

Instrução	Mnem	opcode	Operação
Carrega	LD(ra) ← D(d)	0000	RF[ra] ← D(d)

2. ST(d),(ra) - Carrega na memória de dados o conteúdo endereçado pelo parâmetro d e transfere do registrador de arquivo endereçado pelo endereço dado por ra.

Instrução	Mnem	opcode	Operação
Armazena	ST(d) ← (ra)	0001	D(d) ← RF[ra]

3. ADD(ra),(rb),(rc) – Os conteúdos no registrador de arquivo dado pelos registradores b e c são somados na ULA e o resultado deve ser armazenado no próprio registrador de arquivo no endereço dado pelo registrador ra.

Instrução	Mnem	opcode	Operação
Carrega	ADD(ra),(rb),(rc)	0010	RF[ra] ← RF[rb] + RF[rc]

4. MVI(ra), #C - Carrega imediatamente o conteúdo c no registrador de arquivo no endereço fornecido pelo registrador ra.

Instrução	Mnem	opcode	Operação
Mover dado	MVI(ra) ← c	0011	RF[ra] ← c

5. SUB(ra),(rb),(rc) - Os conteúdos no registrador de arquivo dado pelos registradores b e c são subtraídos na ULA e o resultado deve ser armazenado no próprio registrador de arquivo no endereço dado pelo registrador ra.

Instrução	Mnem	opcode	Operação
Subtrai	SUB(ra),(rb),(rc)	0100	RF[ra] ← RF[Rb] – RF[rc]

6. JZ,(d) – Salta para o endereço dado em d se o resultado na saída da ULA for igual a zero.

Instrução	Mnem	opcode	Ra	Operação
Salto se zero	JZ,(d)	0101	0000	PC ← (d)

7. JNZ,(d) - Salta para o endereço dado em d se o resultado na saída da ULA for diferente de zero.

Instrução	Mnem	opcode	ra	Operação
Salto se não zero	JNZ,(d)	0101	0001	PC ← (d)

8. JNEQ,(d) - Salta para o endereço dado em d se a comparação entre conteúdos A e B for menor ou igual.

Instrução	Mnem	opcode	ra	Operação
Salto menor ou igual	JNEQ,(d)	0101	0011	PC ← (d)

9. JNEQ,(d) - Salta para o endereço dado em d se a comparação entre conteúdos A e B for maior ou igual.

Instrução	Mnem	opcode	ra	Operação
Salto maior ou igual	JMEQ,(d)	0101	0101	PC ← (d)

10. JC,(d) - Salta para o endereço dado em d se a a operação realizada na ULA gerou um bit vai um "carry" .

Instrução	Mnem	opcode	Ra	Operação
Salto se carry	JC,(d)	0101	0110	PC ← (d)

11. JNC,(d) - Salta para o endereço dado em d se a operação realizada na ULA não gerou um bit vai um "carry".

Instrução	Mnem	Opcode	ra	Operação
Salto se não carry	JNC,(d)	0101	0111	PC ← (d)

12. JD0,(d) - Salta para o endereço dado em d se o bit D0 do conteúdo no registrador de arquivo dado pelo registrador ra e deslocado para o deslocador é igual a "1'.

Instrução	Mnem	opcode	ra	Operação
Salto se bit D0	JD0(ra) ← (d)	0101	1000	PC ← (d)

13. JND0,(d) - Salta para o endereço dado em d se o bit D0 do conteúdo no registrador de arquivo dado pelo registrador ra e deslocado para o deslocador é igual a "0'.

Instrução	Mnem	opcode	ra	Operação
Salto se não bit D0	JND0(ra) ← (d)	0101	1001	PC ← (d)

14. JD7,(d) - Salta para o endereço dado em d se o bit D7 do conteúdo no registrador de arquivo dado pelo registrador ra e deslocado para o deslocador é igual a "1'.

Instrução	Mnem	opcode	ra	Operação
Salto se bit D7	JD7(ra) ← (d)	0101	1010	PC ← (d)

15. JND7,(d) - Salta para o endereço dado em d se o bit D0 do conteúdo no registrador de arquivo dado pelo registrador ra e deslocado para o deslocador é igual a "0'.

Instrução	Mnem	opcode	ra	Operação
Salto se não bit D7	JND7(ra) ← (d)	0101	1011	PC ← (d)

16. JMOR,(d) - Salta para o endereço dado em d se a comparação entre os conteúdos nos registradores de arquivo dados pelos endereços dos registradores ra e rb, for maiorl.

Instrução	Mnem	Opcode	ra	Operação
Salta se maior	JMOR(ra) ← (d)	0101	1100	PC ← (d)

17. JEQ,(d) - Salta para o endereço dado em d se a comparação entre os conteúdos nos registradores de arquivo dados pelos endereços dos registradores ra e rb, for igual.

Instrução	Mnem	Opcode	ra	Operação
Salto se igual	JEQ(ra) ← (d)	0101	1101	PC ← (d)

18. JNOR,(d) - Salta para o endereço dado em d se a comparação entre os conteúdos nos registradores de arquivo dados pelos endereços dos registradores ra e rb, for menor.

Instrução	Mnem	opcode	ra	Operação
Salto se menor	JNOR(ra) ← (d)	0101	1110	PC ← (d)

19. JMP,(d) - Salta para o endereço dado em d.

Instrução	Mnem	opcode	ra	Operação
Salto incondicional	JMP(ra) ← (d)	0101	1111	PC ← (d)

20. OUT(ra) – Transfere para a saída de dados o conteúdo do registrador de arquivo endereçado pelo registrador ra.

Instrução	Mnem	opcode	Operação
Saída	OUT(ra)	0110	OUT ← [ra]

21. IN(ra),ext - Carrega no registrador de arquivo no endereço dado pelo registrador ra o conteúdo externo.

Instrução	Mnem	opcode	Operação
Entrada	IN(ra) ← ext	0111	RF[ra] ← ext

22. CMP(ra),(rb) – Compara os conteúdos endereçados pelos registradores ra e rb no registrador de arquivo.

Instrução	Mnem	Opcode	Operação
Compare	CMP(ra),(rb)	1000	[ra] : [rb]

23. XOR(ra),(rb),(rc) - Os conteúdos no registrador de arquivo dado pelos registradores b e c são realizadas a operação exclusivo-ou na ULA e o resultado deve ser armazenado no próprio registrador de arquivo no endereço dado pelo registrador ra.

Instrução	Mnem	opcode	Operação
XOR	XOR(ra),(rb),(rc)	1001	RF[ra] ← RF[rb] ⊕ RF[rc]

24. AND(ra),(rb),(rc) - Os conteúdos no registrador de arquivo dado pelos registradores b e c são realizadas a operação "E" na ULA e o resultado deve ser armazenado no próprio registrador de arquivo no endereço dado pelo registrador ra.

Instrução	Mnem	opcode	Operação

E	AND(ra),(rb),(rc)	1010	RF[ra] ← RF[rb] ∧ RF[rc]

25. OR(ra),(rb),(rc) - Os conteúdos no registrador de arquivo dado pelos registradores b e c são realizadas a operação "OU" na ULA e o resultado deve ser armazenado no próprio registrador de arquivo no endereço dado pelo registrador ra.

Instrução	Mnem	Opcode	Operação
OU	OU(ra),(rb),(rc)	1011	RF[ra] ← RF[rb] ∨ RF[rc]

26. INC(ra) – Incrementa o conteúdo endereçado do registrador de arquivo endereçado pelo endereço dado por ra.

Instrução	Mnem	opcode	Operação
Incrementa	INC(ra)	1100	RF[ra] ← RF[ra] + 1

27. DEC(ra) - Decrementa o conteúdo endereçado do registrador de arquivo endereçado pelo endereço dado por ra.

Instrução	Mnem	opcode	Operação
Decrementa	DEC(ra)	1101	RF[ra] ← RF[ra] - 1

28. SH(rb),(rc) - Transfere o conteúdo do registrador de arquivo fornecido pelo registrador Rb e escreve o conteúdo no endereço fornecido pelo registrador rc no registrador de arquivo.

Instrução	Mnem	opcode	ra	Operação
Transfere	SH(ra),(rb)	1110	0000	RF[ra] ← (rc)

29. SHL(ra) – Desloca um bit a esquerda o conteúdo endereçado pelo registrador ra e preenche o bit com zero e armazena o conteúdo no endereço dado por ra.

Instrução	Mnem	opcode	ra	Operação
Desloca esquerda	SHL(ra) ← (d)	1110	0001	RF[ra] ← (d)

30. SHR(ra) - Desloca um bit a direita o conteúdo endereçado pelo registrador ra e preenche o bit com zero e armazena o conteúdo no endereço dado por ra.

Instrução	Mnem	opcode	ra	Operação
Desloca direita	SHR(ra) ← (d)	1110	0010	RF[ra] ← (d)

31. ROR(ra) – Gira um bit a direita o conteúdo endereçado pelo registrador ra e preenche o bit com bit deslocado e armazena o conteúdo no endereço dado por ra.

Instrução	Mnem	opcode	ra	Operação
Gira a direita	ROR(ra) ← (d)	1110	0011	RF[ra] ← (d7)

32. HLT - Carrega da memória de dados o conteúdo endereçado pelo parâmetro d e transfere para o registrador de arquivo endereçado pelo endereço dado por ra.

Instrução	Mnem	opcode	Operação
Parada	HLT	1111	PC ← PC

O próximo passo após concluído o projeto é a simulação e para isso precisamos criar vários programas envolvendo todo o conjunto de instruções e situações de decisões. Usaremos o quartus II para essa finalidade.

Simulação – Vamos iniciar criando um programa com instruções.

Programa 1 – Manipulação de dados

Opcode	Mnemonica	Operação
0: 3005	MVI[0],05	RF[0] = #05;
1: 3106	MVI[1],06	RF[1] = #06;
2: 3207	MVI[2],07	RF[2] = #07;
3: 2001	ADD[0],[0],[1]	RF[0] = RF[0] + RF[1];
4: 2002	ADD[0],[0],[2]	RF[0] = RF[0] + RF[2]
5: 6000	OUT[0]	OUT = RF[0].

Programa 2 – Manipulação aritmética de dados

Opcode	Mnemônica	Operação
6: 1000	ST[0],[00]	D[0] = RF[0];
7: 0300	LD[3],[00]	RF[3] = D[0];
8: 4421	SUB[4],[2],[1]	RF[4] = RF[2] – RF[1];
9: 9504	XOR[5],[0],[4]	RF[5] = RF[0] $\oplus$ RF[4];
A: 6500	OUT = RF[5]	Saída = RF[5].

Programa 3 – Manipulação lógica de dados

Opcode	Mnemônica	Operação
1: 3600	MVI[6],00	RF[6] = #00;
2: A665	AND[6],[6],[5]	RF[6] = RF[6] $\wedge$ RF[5];
3: 37F0	MVI[7],F0	RF[7] = #F0;
4: 6700	OUT = RF[7]	Saída = RF[7].

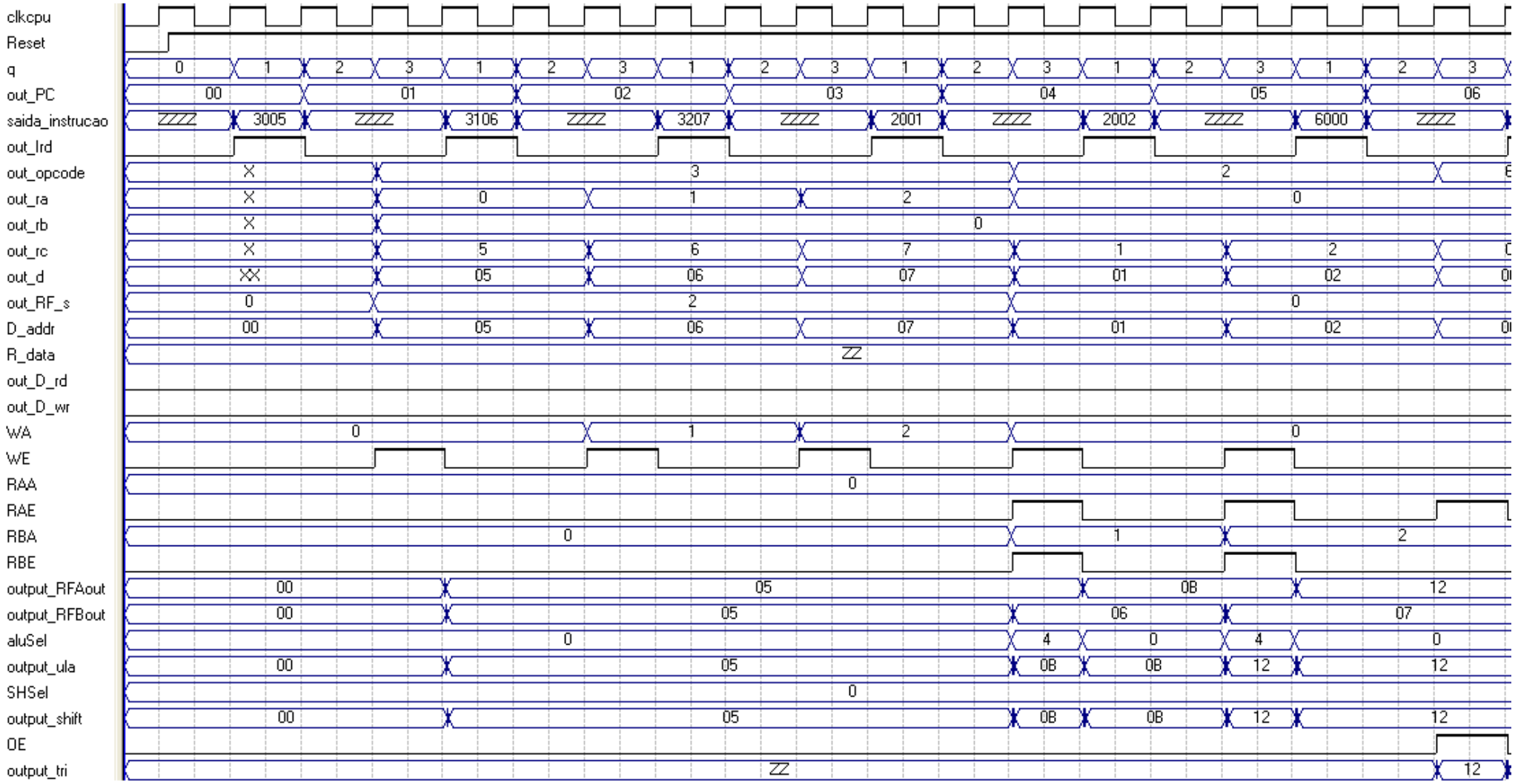
Programa 4 – Multiplicação de 02 números

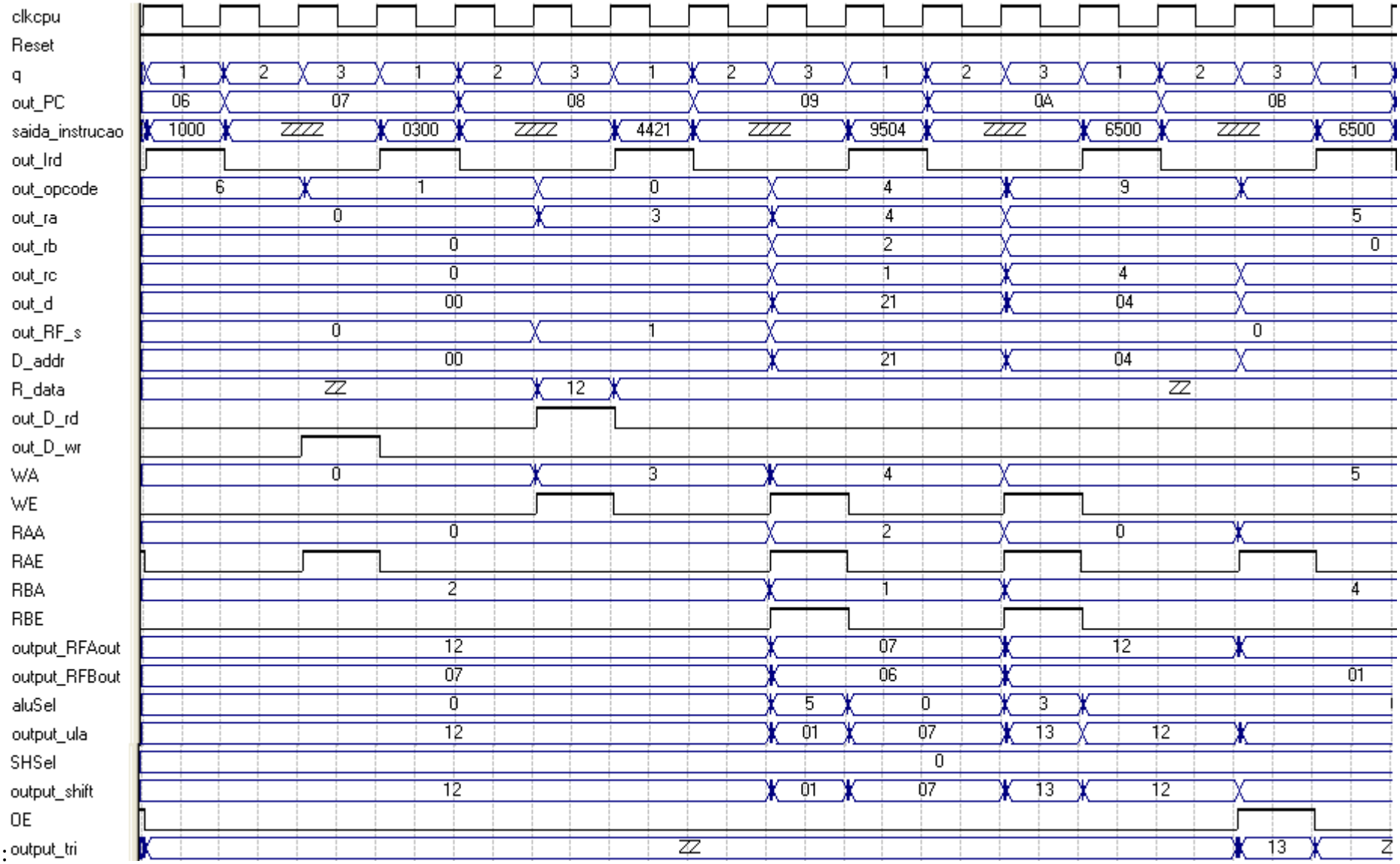
Opcode	Mnemônica	Operação
1: 4222	SUB[2],[2],[2]	RF[2] = RF[2] –RF[2];-- Zera Resultado
2: 300A	MVI[0],0A	RF[0] = #0A;-- Multiplicador
3: 3170	MVI[1],70	RF[1] = #70;--Multiplicando
4: 3304	MVI[3],04	RF[3] = #04;--Contador
5: E000	SH[0],[0]	RF[0] = RF[0];--Teste bit d0
6: 5908	JND0,[08]	PC = #d;--Salta DO = 1 para d;
7: 2221	ADD[2],[2],[1]	RF[2] = RF[2] + RF[1];--Resultado + multiplicando
8: E220	SHR[2]	C $\rightarrow$ RF[2];--Desloca bit a direita
9: E200	SHR[0]	0 $\rightarrow$ RF[0],--Desloca multiplicador
A: D300	DEC[3]	RF[3] = RF[3] – 1;Atualiza contador
B: 5105	JNZ,[05]	PC = #d;--Salta Z = 0 para d
C: 6200	OUT[2]	Saída = RF[2];--Apresenta resultado
D: F000	HLT	PC = PC;--Parada

Programa 5 – Determinação de número por aproximação sucessivas

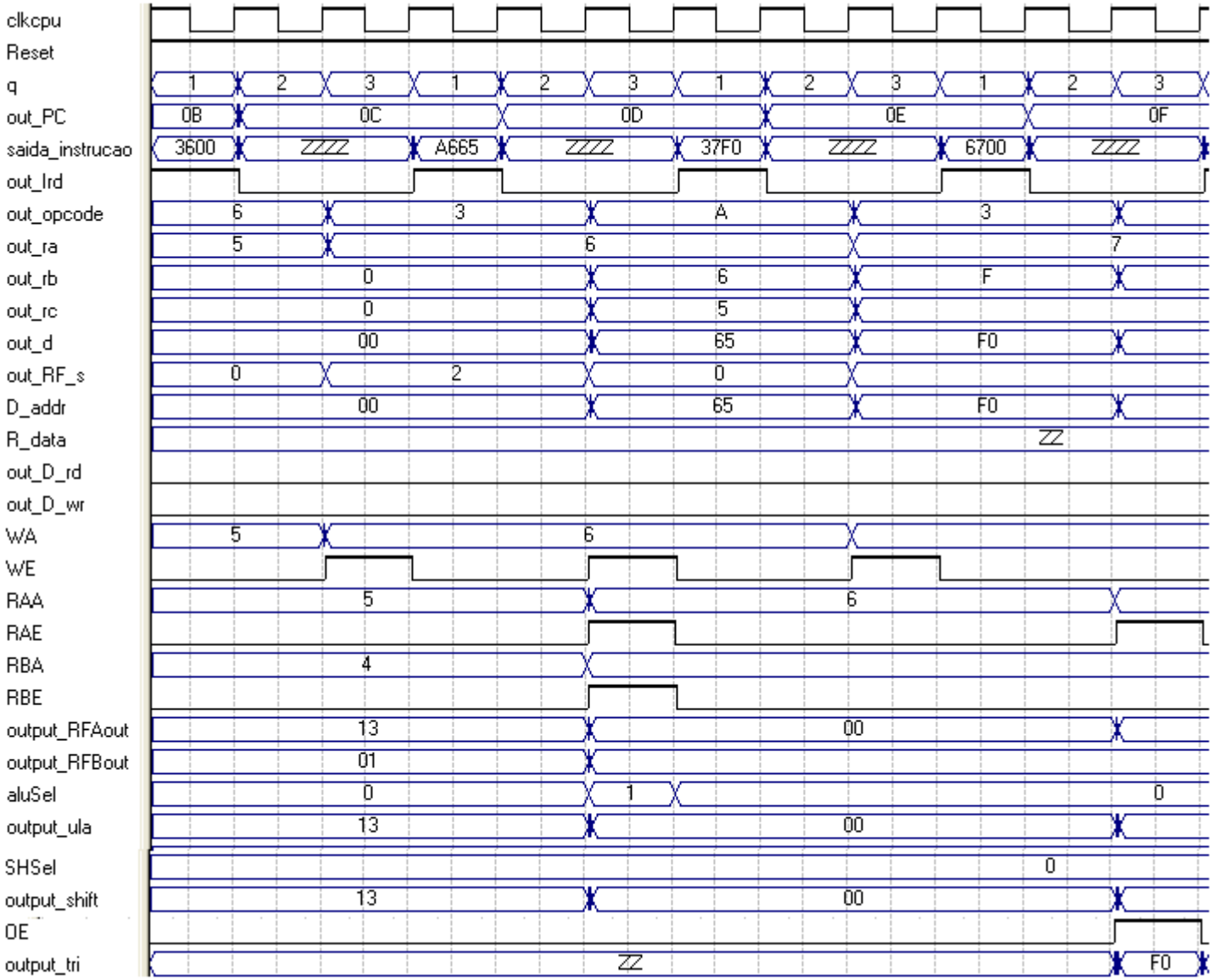
Opcode	Mnemônica	Operação
1: 301D	MVI[0],1D	RF[0] = #1D;--Número a ser determinado
2. 3180	MVI[1],80	RF[1] = #80;--Valor da primeira tentativa
3: 3208	MVI[2],08	RF[2] = #08;--Contador de tentativas igual a 8
4: 4333	SUB[3],[3],[3]	RF[3] = RF[3] – RF[3];--Zera resultado
5: 2313	ADD[3],[1],[3]	RF[3] = RF[1] + RF[3];--Primeira tentativa
6: 8300	CMP[3],[0]	RF[3] ≤ RF[0];--Comparação
7: 5309	JNEQ,[09]	True PC = 09;--Salta para endereço 09
8: 4331	SUB[3],[3],[1]	RF[3] = RF[3] – RF[1];--Ajusta valor
9: E210	SHR[1]	RF[1] ÷ 2;--Desloca a direita
A: D200	DEC[2]	RF[2] = RF[2] – 1;--Decrementa contador
B: 5105	JNZ,[05]	True PC = 05;--Salta para endereço soma
C: 6300	OUT[3]	Saída = RF[3];--Apresenta
D; F000	HLT	Pára;--Parada do programa

As formas de ondas a seguir mostram a evolução dos sinais gerados e a solução é apresentada na saída igual a 12;



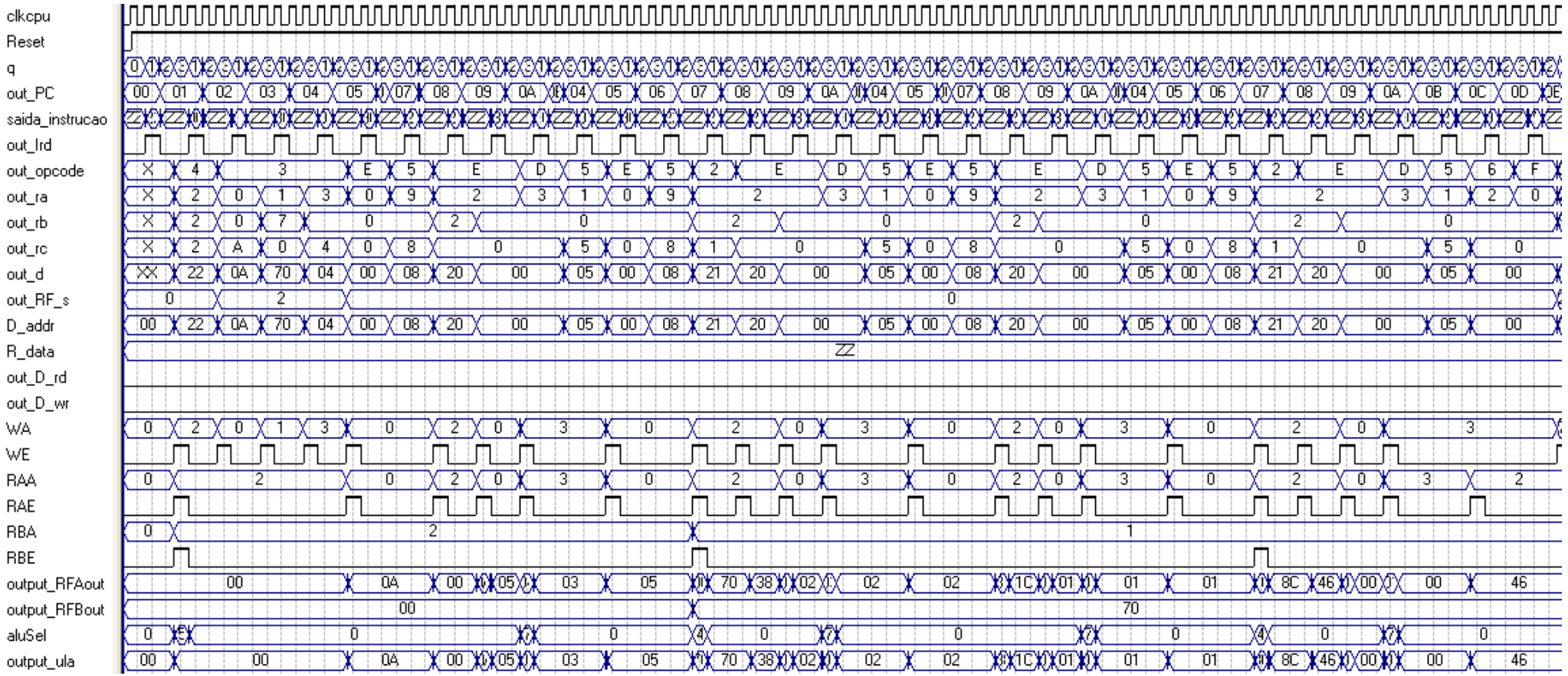

clkcpu
Reset
q
out_PC
saida_instrucao
out_Ird
out_opcode
out_ra
out_rb
out_rc
out_d
out_RF_s
D_addr
R_data
out_D_rd
out_D_wr
WA
WE
RAA
RAE
RBA
RBE
output_RFAout
output_RFBout
aluSel
output_ula
SHSel
output_shift
OE
output_tri

Programa 2:

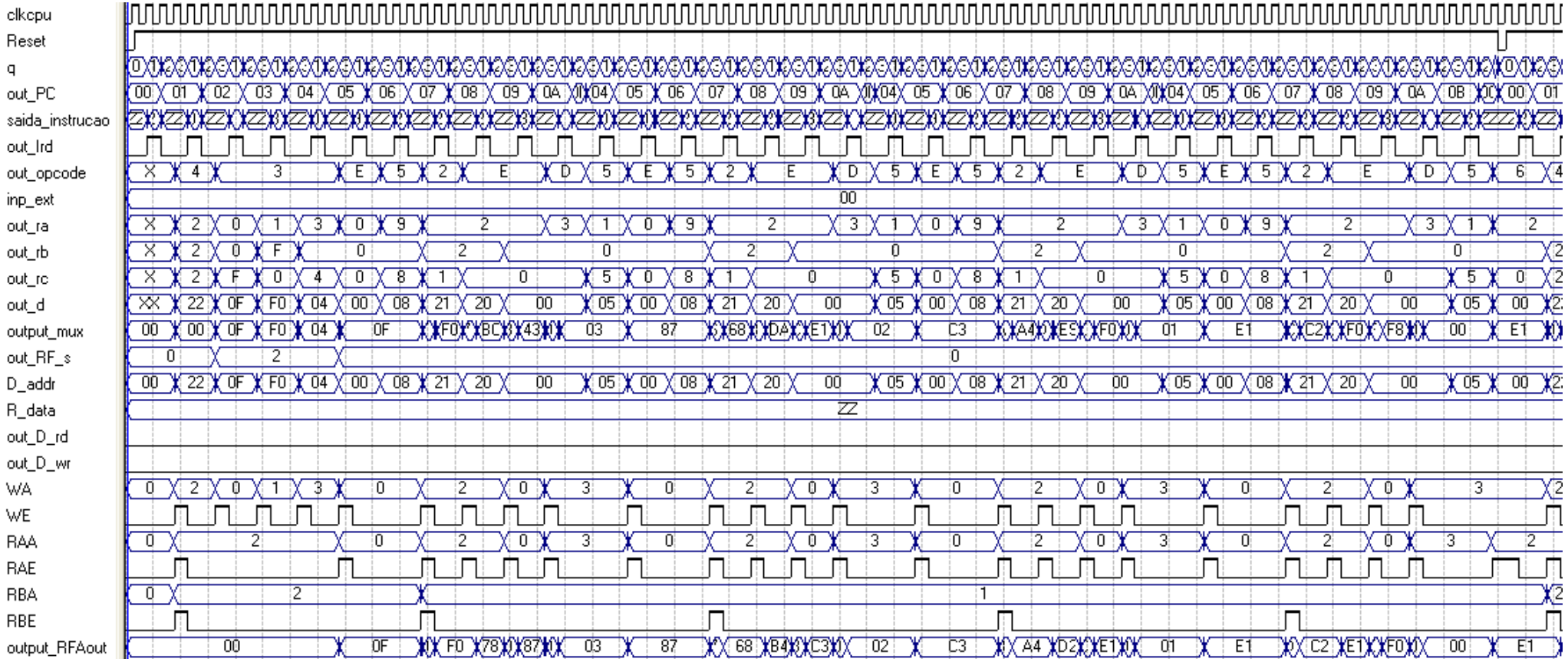

clkcpu
Reset
q
out_PC
saida_instrucao
out_Ird
out_opcode
out_ra
out_rb
out_rc
out_d
out_RF_s
D_addr
R_data
out_D_rd
out_D_wr
WA
WE
RAA
RAE
RBA
RBE
output_RFAout
output_RFBout
aluSel
output_ula
SHSel
output_shift
OE
output_tri

Programa 3:

Programa 4: Multiplicação entre 2 números 10 e 7 = 70 ou 46_H.

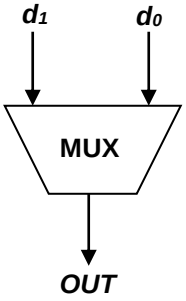


Programa 5: Multiplicação de 8 bits 15 x 15 = 225 ou E1.



Introdução: Projetos de sistemas digitais podem ser realizados em níveis de abstração. O diagrama a seguir mostra os níveis de abstração no projeto e o nível mais baixo nesta hierarquia mostra o ***nível do transistor*** onde os transistores como componentes discretos são conectados juntos para criar o circuito. O nível seguinte ao nível de abstração é o nível das portas lógicas e daí são estas conectadas para criar os circuitos combinatórios, como somadores, multiplexadores, decodificadores e outros e os circuitos seqüênciais como os Flip-Flops. Neste nível as equações são descritas por exepressões booleans e definidas pela sua tabela da verdade. Neste nível é possível a craição de grandes circuitos e até parte de um microprocessador. A metodologia de projeto diz que é muito mais fácil solucionar problemas hierarquicamente do que o problema interiro como um todo. Os blocos combinacionais e seqüenciais são utilizados no nível de fluxo de dados e unidade de controle de um microprocessador. Neste nível são possíveis a transferência dos registros internos e a sua movimentação no fluxo de dados a fim de solucionar o problema. O fluxo de dados contém componentes como ULA, registrador de arquivos, multiplexadores, registradores e outros para realizar uma ou mais operações conforme dita a instrução. Finalmente o nível mais alto hierarquico é o nível comportamental, onde descreve-se o circuito pelo seu comportamento lógico através de uma linguagem de descrição de hardware denominada de VHDL (Very Hardware Description Language).

Nível Comportamental: Um exemplo de descrição no nível comportamental é um circuito Multipleador para duas entradas ***d_0*** e ***d_1*** e uma saída ***out_bit***. A sintaxe correta para descrição da linguagem do hardware será:




```
ENTITY Multiplexer IS PORT (
d0,d1,s : IN BIT;
y : OUT BIT);
END multiplexer;

Architecture Behavioral OF Multiplexer IS
 Begin
Process (s,d0,d1)
Begin
y <= d0 WHEN s = '0' ELSE d1;
END Process;
END Behavioral;
```

INTRODUÇÃO AO VHDL

O VHDL é uma linguagem de descrição de hardware o qual serve para a modelagem de sistemas digitais. Igualmente outras linguagens utilizadas por computador, uma vez descrito o programa, um compilador trasnforma a linguagem fonte em linguagem objeto ou código de máquina. Na linguagem VHDL o compilador é um sintetizador que transforma o código fonte VHDL para uma descrição atual do hardware o qual implementa o código. Da descrição é gerada a netlist, dispositivo físico que realiza o código fonte e uma simulação funcional e temporal pode ser realizada para a correção de problemas no circuito.

VHDL PARA PORTA LÓGICA NAND DE 02 ENTRADAS

A implementação da linguagem VHDL para a porta NAND de 02 entradas serve também como template básico para todos os códigos VHDL.
As linhas iniciais com 02 hyfens são comentários. As declarações **Library** e **Use** especificam que a biblioteca **IEEE** é necessária e que todos os componentes da biblioteca podem ser utilizados. Estas 02 declarações são eqüivalentes para a linha preprocessador "#include" em C++.
Cada componente definedo em VHDL, se é uma simples porta NAND ou um complexo microprocessador, tem 02 partes: primeira entidade e a segunda a arquitetura. A entidade é similar a uma declaração de função em C++ e serve como a interface entre o componente e o lado externo. Cada entidade deve ter um único nome; no examplo, o nome NAND2gate é usedo. A entidade contém uma lista, o qual, como uma lista de parametros, especifica o dado a ser passado dentro e fora do componente. No examplo, existem 02 sinais de entrada chamado x e y do tipo std_logic e um sinal de saída chamado de F do mesmo tipo. O tipo std_logic é como um bit mas contém valores adicionais além de apenas 0 e 1. A arquitetura é uma definição de compoenente e contém o codigo que realiza a operação do componente. Para cada arquitetura se necessita especificar seu nome e qual entidade ela é; no examplo, o nome é Dataflow e ela é para a entidade NAND2 gate. É possível para uma entidade ter mais do que uma arquitetura uma vez uma entidade pode ser implementada em mais do que um modo. Dentro do corpo da arquitetura, pode-se ter uma ou mais declarações concorrentes. Diferentes declarações em C++ onde elas são executadas em ordem sequencial, declaração concorrente no corpo da arquitetura são executadas em paralelo. Assim, a ordenação desta declaração é irrelevante. O símbolo "<=" é usedo na declaração designação do sinal. Apenas como uma declaração regular de designação, a expressão do lado direitp é avaliada primeiro e o resultado é designado para o sinal do lado esquerdo. O operador nand é um operador pronto.

Princípios de\Projeto Lógico Digital

A seguir uma lista de operadores lógicos é apresentada.

A.1.5 Operadores dos Dados

O VHDL listado a seguir são operadores prontos

EXEMPLOS DE OPERADORES LÓGICOS

Operadores Lógicos	Operação	Exemplo
AND	AND	a AND b
OR	OR	a OR b
NOT	NOT	NOT a
NAND	NAND	a NAND b
NOR	NOR	a NOR b
XOR	XOR	a XOR b
XNOR	XNOR	a XNOR b
Operadores Aritméticos		
+	Adição	a + b
-	Subtração	a - b
*	Multiplicação	a * b
/	Divisão	a / b
MOD	Módulo	a MOD b

REM	Resto	a REM b
**	Exponenciação	a EXP b
&	Concatenação	‘a' & ‘b’
ABS	Absoluto	
Operadores Relacionais		
=	Igual	
/=	Diferente	
<	Menor	
<=	Menor Iugal	
>	Maior	
>=	Maior Igual	
Operadores Deslocamento		
sll	Deslocamento lógico a esquerda	
srl	Deslocamento lógico a direita	
sla	Deslocamento aritmético a esquerda	
sra	Deslocamento lógico a direita	
rol	Giro a esquerda	
Ror	Giro a Direita	

1.6 ENTIDADE

Uma declaração **ENTIDADE** declara a interface externa ou usuário do módulo similar à declaração de uma função. Ela especifica o nome da entidade e sua interface. A interface consiste dos sinais a serem passados dentro ou fora da entidade.
Sintaxe:

ENTITY nome-da-entidade **IS**
PORT (listagem-dos-nomes-dos -portos-e-tipos);
END nome-da-entidade;

Exemplo:

```
ENTITY Siren IS PORT (
M: IN BIT;
D: IN BIT;
V: IN BIT;
S: OUT BIT);
END Siren;
```

1.7 ARQUITETURA

O corpo ARQUITETURA define a implementação atual da funcionalidade da entidade. Ela é similar a definição ou implementação da função. A sintaxe para a arquitetura varia e depende sobre o modelo (dataflow, behavioral, ou structural) a ser utilizado.

Sintaxe para o modelo dataflow:

ARCHITECTURE nome-arquitetura **OF** nome-entidade **IS**
signal-declarations;

BEGIN
concurrent-statements;
END nome-arquitetura;

As declarações concorrentes são executadas concorrentemente.

Exemplo:

ARCHITECTURE Siren_Dataflow OF Siren IS
SIGNAL term_1: BIT;

BEGIN
term_1 <= D OR V;
S <= term_1 AND M;
END Siren_Dataflow;

Sintaxe para o modelo behavioral:

ARCHITECTURE nome-arquitetura **OF** nome-entidade **IS**
declarações dos sinais;
definições das funções;
definições dos procedimentos;

BEGIN

Blocos-processo;
Declarações-concorrentes;
END nome-arquitetura;

Declarações dentro do bloco-processo são executadas sequencialmente. Entretanto, o proprio bloco-processo é uma declaração concorrente.

Exemplo:

ARCHITECTURE *Siren_Behavioral* ***OF*** *Siren* ***IS***
SIGNAL *term_1: BIT;*
BEGIN
PROCESS *(D, V, M)*
BEGIN
term_1 <= D OR V;
S <= term_1 AND M;
END PROCESS;
END *Siren_Behavioral;*

Sintaxe para o modelo estrutural

ARCHITECTURE nome-arquitetura **OF** nome-entidade **IS**
declarações-componentes;
declarações-sinais;

BEGIN
nome-instance : declaração-PORT MAP;
declaração-concorrente;
END nome-arquitetura;

Para cada declaração usada de componente, deve exisitir uma entidade correspondente e arquitetura para aquele componente.
A declaração PORT MAP são declarações concorrentes.

Exemplo:

ARCHITECTURE Siren_Structural OF Siren IS
COMPONENT myOR PORT (
in1, in2: IN BIT;
out1: OUT BIT);
END COMPONENT;
SIGNAL term1: BIT;
BEGIN
U0: myOR PORT MAP (D, V, term1);
S <= term1 AND M;
END Siren_Structural;

1.8 PACKAGE

Um **PACKAGE** proporciona um mecanismo para agrupar juntas e dividir declarações que são utilizadas por várias unidades entidades.

Um package próprio inclui uma declaração e, opcionalmente, um corpo. A declaração package e corpo são usualmente armazenadas juntas num arquivo separado do resto das unidades de projeto. O nome do arquivo dado por este deve ser o mesmo nome do arquivo package. A fim de completar o projeto para sintetizar corretamente usando MAX+PLUS II, deve-se primeiro sintetizar o package comoa uma unidade separada. Depois pode-se sintetizar a unidade que utiliza o package.

Declaração PACKAGE e BODY

A declaração **PACKAGE** contém declarações que podem ser divididas entre diferentes unidades entidades. Ela proporciona a interface, que são, itens que são visíveis para as outras unidades entidades. O opcional **PACKAGE BODY** contém as implementações das funções e procedimentos que são declarados na declaração PACKAGE.

Sintaxe para a declaração **PACKAGE**:

PACKAGE package-name IS
type-declarations;
subtype-declarations;

signal-declarations;
variable-declarations;
constant-declarations;
component-declarations;
function-declarations;
procedure-declarations;
END package-name;

Sintaxe para **PACKAGE body**:

PACKAGE BODY nome-package IS
definições-funções; -- para funções declaradas na declaração package
definições-procedimentos; -- para procedimentos declarados na declaração package
END package-name;

Exemplo:

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
PACKAGE my_package IS
SUBTYPE bit4 IS std_logic_vector(3 DOWNTO 0);
FUNCTION Shiftright (input: IN bit4) RETURN bit4; -- declare a function
SIGNAL mysignal: bit4; -- a global signal
END my_package;

PACKAGE BODY my_package IS
-- implementation of the Shiftright function
FUNCTION Shiftright (input: IN bit4) RETURN bit4 IS
BEGIN
RETURN '0' & input(3 DOWNTO 1);
END shiftright;
END my_package;

Usando um PACKAGE

To use a package, you simply include a LIBRARY and USE statement for that package. Before synthesizing the
module that uses the package, you need to first synthesize the package by itself as a top-level entity.

Syntax:

LIBRARY WORK;
USE WORK.package-name.ALL;

Exemplo:

LIBRARY work;
USE work.my_package.ALL;
ENTITY test_package IS PORT (
x: IN bit4;

```
z: OUT bit4);
END test_package;
ARCHITECTURE Behavioral OF test_package IS
BEGIN
mysignal <= x;
z <= Shiftright(mysignal);
END Behavioral;
```

2. Dataflow Model Concurrent Statements

Concurrent statements used in the dataflow model are executed concurrently. Hence, the ordering of these
statements does not affect the resulting output.

2.1 Concurrent Signal Assignment

Assigns a value or the result of evaluating an expression to a signal. This statement is executed whenever a
signal in its expression changes value. However, the actual assignment of the value to the signal takes place after a
certain delay and not instantaneously as for variable assignments.

Syntax:
signal <= expression;

Exemplo:

```
y <= '1';
z <= y AND (NOT x);
```

2.2 Conditional Signal Assignment

Selects one of several different values to assign to a signal based on different conditions. This statement is
executed whenever a signal in any one of the value or condition changes.
Syntax:
signal <= value1 WHEN condition ELSE
value2 WHEN condition ELSE
…
value3;

Exemplo:

```
z <= in0 WHEN sel = "00" ELSE
in1 WHEN sel = "01" ELSE
in2 WHEN sel = "10" ELSE
in3;
```

2.3 Selected Signal Assignment

Selects one of several different values to assign to a signal based on the value of a select expression. This
statement is executed whenever a signal in the expression or any one of the value changes.
Syntax:
WITH expression SELECT
signal <= value1 WHEN choice1,
value2 WHEN choice2 | choice3,
…
value4 WHEN choice4;

All possible choices for the expression must be given. The keyword OTHERS can be used to denote all remaining
choices.

Exemplo:

WITH sel SELECT
z <= in0 WHEN "00",
in1 WHEN "01",
in2 WHEN "10",
in3 WHEN OTHERS;

Dataflow Model Example

-- outputs a 1 if the 4-bit input is a prime number, 0 otherwise
ENTITY Prime IS PORT (
number: IN BIT_VECTOR(3 DOWNTO 0);
yes: OUT BIT);
END Prime;
ARCHITECTURE Prime_Dataflow OF Prime IS
BEGIN
WITH number SELECT
yes <= '1' WHEN "0001" | "0010",
'1' WHEN "0011" | "0101" | "0111" | "1011" | "1101",
'0' WHEN OTHERS;
END Prime_Dataflow;

3. Behavioral Model Sequential Statements

The behavioral model allows statements to be executed sequentially just like in a regular computer program.
Sequential statements include many of the standard constructs such as variable assignments, if-then-else, and loops.

3.1 PROCESS

The PROCESS block contains statements that are executed sequentially. However, the PROCESS statement itself is

a concurrent statement. Multiple process blocks in an architecture will be executed simultaneously. These process
blocks can be combined together with other concurrent statements.

Syntax:
process-name: PROCESS (sensitivity-list)
variable-declarations;
BEGIN
sequential-statements;
END PROCESS process-name;

A lista-sensibilidade é uma lista de sinais separada por vírgulas no qual o processo é sensível para. Em outraspalavras, sempre que um sinal troca o valor na lista, o processo será executado, isto é, todas as delcarações listadas na ordem sequencial. Depois da última declaração tem sido executada, o processo será suspenso até a próxima vez que um sinal na lista sensibilidade altera o valor antes dele é novamente executado.

Exemplo:

PROCESS (D, V, M)
BEGIN
term_1 <= D OR V;
S <= term_1 AND M;
END PROCESS;

3.2 Sequential Signal Assignment

Designa o valor para um sinal. Esta declaração é apenas como sua concorrente counterpart exceto que ela é executada sequencialmente, isto é, somente quando a execução alcança ela.

Sintaxe:

signal <= expression;

Exemplo:

y <= '1';
z <= y AND (NOT x);

3.3 Variable Assignment

Designa um valor ou o resultado da avaliação de uma expressão para o valor da variável. Sempre designada a variável instataneamente sempre que esta declaração é executada.
Variáveis são somente delcaradas dentro do bloco processo.

Sintaxe:
signal:= expression;

Exemplo:

```
y := '1';
yn := NOT y;
```

3.4 WAIT

When a process has a sensitivity list, the process always suspends after executing the last statement. An alternative to using a sensitivity list to suspend a process is to use a WAIT statement, which must also be the first statement in a process1.
Syntax2:
WAIT UNTIL condition;

Exemplo:

```
-- suspend until a rising clock edge
WAIT UNTIL clock'EVENT AND clock = '1';
```

3.5 IF THEN ELSE

Syntax:
IF condition THEN
sequential-statements1;
ELSE
sequential-statements2;
END IF;
IF condition1 THEN
sequential-statements1;
ELSIF condition2 THEN
sequential-statements2;
…
ELSE
sequential-statements3;
1 This is only a MAX+PLUS II restriction.
2 There are three different formats of the WAIT statement, however,

MAX+PLUS II only supports one.
END IF;

Exemplo:

```
IF count /= 10 THEN -- not equal
count := count + 1;
ELSE
count := 0;
END IF;
```

3.6 CASE

Syntax:

CASE expression IS
WHEN choices => sequential-statements;
WHEN choices => sequential-statements;
…
WHEN OTHERS => sequential-statements;
END CASE;

Exemplo:

CASE sel IS
WHEN "00" => z <= in0;
WHEN "01" => z <= in1;
WHEN "10" => z <= in2;
WHEN OTHERS => z <= in3;
END CASE;

3.7 NULL

The NULL statement does not perform any actions.
Syntax:
NULL;

3.8 FOR

Syntax:
FOR identifier IN start [TO | DOWNTO] stop LOOP
sequential-statements;
END LOOP;

Loop statements must have locally static bounds3. The identifier is implicitly declared, so no explicit declaration
of the variable is needed.

Example:

sum := 0;
FOR count IN 1 TO 10 LOOP
sum := sum + count;
3 This is only a MAX+PLUS II restriction.
END LOOP;

3.9 WHILE4

Syntax:
WHILE condition LOOP
sequential-statements;

END LOOP;

3.10 LOOP4

Syntax:
LOOP
sequential-statements;
EXIT WHEN condition;
END LOOP;

3.11 EXIT4

The EXIT statement can only be used inside a loop. It causes execution to jump out of the innermost loop and is
usually used in conjunction with the LOOP statement.
Syntax:
EXIT WHEN condition;

3.12 NEXT

The NEXT statement can only be used inside a loop. It causes execution to skip to the end of the current iteration
and continue with the beginning of the next iteration. It is usually used in conjunction with the FOR statement.

Syntax:
NEXT WHEN condition;
Example:

sum := 0;
FOR count IN 1 TO 10 LOOP
NEXT WHEN count = 3;
sum := sum + count;
END LOOP;

3.13 FUNÇÃO

Syntax for function declaration:
4 Not supported by MAX+PLUS II.

FUNCTION function-name (parameter-list) RETURN return-type;
Syntax for function definition:
FUNCTION function-name (parameter-list) RETURN return-type IS
BEGIN
sequential-statements;
END function-name;
Syntax for function call:
function-name (actuals);
Parameters in the parameter-list can be either signals or variables of mode IN only.

Exemplo:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY test_function IS PORT (
x: IN std_logic_vector(3 DOWNTO 0);
z: OUT std_logic_vector(3 DOWNTO 0));
END test_function;

ARCHITECTURE Behavioral OF test_function IS
SUBTYPE bit4 IS std_logic_vector(3 DOWNTO 0);
FUNCTION Shiftright (input: IN bit4) RETURN bit4 IS
BEGIN
RETURN '0' & input(3 DOWNTO 1);
END shiftright;

SIGNAL mysignal: bit4;
BEGIN
        PROCESS
                BEGIN
                        mysignal <= x;
                        z <= Shiftright(mysignal);
        END PROCESS;
END Behavioral;
```

3.14 PROCEDURE

Syntax for procedure declaration:
PROCEDURE procedure -name (parameter-list);
Syntax for procedure definition:
PROCEDURE procedure-name (parameter-list) IS
BEGIN
sequential-statements;
END procedure-name;
Syntax for procedure call:
procedure -name (actuals);
Parameters in the parameter-list are variables of modes IN, OUT, or INOUT.

Example:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY test_procedure IS PORT (
x: IN std_logic_vector(3 DOWNTO 0);
z: OUT std_logic_vector(3 DOWNTO 0));
END test_procedure;

ARCHITECTURE Behavioral OF test_procedure IS
SUBTYPE bit4 IS std_logic_vector(3 DOWNTO 0);
PROCEDURE Shiftright (input: IN bit4; output: OUT bit4) IS
```

```
BEGIN
        output := '0' & input(3 DOWNTO 1);
END shiftright;
        BEGIN
                PROCESS
                VARIABLE mysignal: bit4;
                BEGIN
                        Shiftright(x, mysignal);
                        z <= mysignal;
                END PROCESS;
END Behavioral;
```

3.15 Behavioral Model Example

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY bcd IS PORT (
I: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
Segs: OUT std_logic_vector (1 TO 7));
END bcd;

ARCHITECTURE Behavioral OF bcd IS
BEGIN
        PROCESS(I)
                BEGIN
                CASE I IS
                                WHEN "0000" => Segs <= "1111110";
                                WHEN "0001" => Segs <= "0110000";
                                WHEN "0010" => Segs <= "1101101";
                                WHEN "0011" => Segs <= "1111001";
                                WHEN "0100" => Segs <= "0110011";
                                WHEN "0101" => Segs <= "1011011";
                                WHEN "0110" => Segs <= "1011111";
                                WHEN "0111" => Segs <= "1110000";
                                WHEN "1000" => Segs <= "1111111";
                                WHEN "1001" => Segs <= "1110011";
                                WHEN OTHERS => Segs <= "0000000";
                END CASE;
        END PROCESS;
END Behavioral;
```

A.4 Structural Model Statements

The structural model allows the manual connection of several components together using signals. All components used must first be defined with their respective ENTITY and ARCHITECTURE sections, which can be in the same file or they can be in separate files.
In the topmost module, each component used in the netlist is first declared using the COMPONENT statement. The declared components are then instantiated with the actual components in the circuit using the PORT MAP statement.
SIGNALs are then used to connect the components together according to the netlist.

4.1 COMPONENT Declaration

Declares the name and the interface of a component that is used in the circuit description. For each component declaration used, there must be a corresponding entity and architecture for that component. The declaration name and the interface must match exactly the name and interface that is specified in the entity section for that component.

Syntax:
COMPONENT component-name IS
PORT (list-of-port-names-and-types);
END COMPONENT;
Example:
COMPONENT half_adder IS PORT (
xi, yi, cin: IN BIT;
cout, si: OUT BIT);
END COMPONENT;

4.2 PORT MAP

The PORT MAP statement instantiates a declared component with an actual component in the circuit by specifying how the connections to this instance of the component are to be made.
Syntax:
label: component-name PORT MAP (association-list);
The association-list can be specified using either the *positional* or *named* method.
Example (positional association):
SIGNAL x0, x1, y0, y1, c0, c1, c2, s0, s1: BIT;
U1: half_adder PORT MAP (x0, y0, c0, c1, s0);
U2: half_adder PORT MAP (x1, y1, c1, c2, s1);
Example (named association):
SIGNAL x0, x1, y0, y1, c0, c1, c2, s0, s1: BIT;
U1: half_adder PORT MAP (cout=>c1, si=>s0, cin=>c0, xi=>x0, yi=>y0);
U2: half_adder PORT MAP (cin=>c1, xi=>x1, yi=>y1, cout=>c2, si=>s1);

4.3 OPEN

The OPEN keyword is used in the PORT MAP association-list to signify that that particular port is not connected or used.
Example:
U1: half_adder PORT MAP (x0, y0, c0, OPEN, s0);

4.4 GENERATE

The GENERATE statement works like a macro expansion. It provides a simple way to duplicate similar components.
Syntax:
label: FOR identifier IN start [TO | DOWNTO] stop GENERATE port-map-statements;
END GENERATE label;

Exemplo:

```
-- using a FOR-GENERATE statement to generate four instances of the full adder
-- component for a 4-bit adder
ENTITY Adder4 IS PORT (
        Cin: IN BIT;
        A, B: IN BIT_VECTOR(3 DOWNTO 0);
        Cout: OUT BIT;
        SUM: OUT BIT_VECTOR(3 DOWNTO 0));
END Adder4;

ARCHITECTURE Structural OF Adder4 IS
        COMPONENT FA PORT (
        ci, xi, yi: IN BIT;
        co, si: OUT BIT);
END COMPONENT;

        SIGNAL Carryv: BIT_VECTOR(4 DOWNTO 0);
                BEGIN
                Carryv(0) <= Cin;
                Adder: FOR k IN 3 DOWNTO 0 GENERATE
                FullAdder: FA PORT MAP (Carryv(k), A(k), B(k), Carryv(k+1), SUM(k));
                END GENERATE Adder;
                Cout <= Carryv(4);
END Structural;
```

4.5 Structural Model Example

This example is based on the following circuit:
D
M
V S

```
-- declare and define the 2-input OR gate
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
        ENTITY myOR IS PORT (
        in1, in2: IN STD_LOGIC;
        out1: OUT STD_LOGIC);
END myOR;

ARCHITECTURE OR_Dataflow OF myOR IS
        BEGIN
        out1 <= in1 OR in2;
END OR_Dataflow;
-- topmost module for the siren
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
        ENTITY Siren IS PORT (
        M: IN STD_LOGIC;
```

```
        D: IN STD_LOGIC;
        V: IN STD_LOGIC;
        S: OUT STD_LOGIC);
END Siren;

ARCHITECTURE Siren_Structural OF Siren IS
-- declaration of the needed OR gate
        COMPONENT myOR PORT (
        in1, in2: IN STD_LOGIC;
        out1: OUT STD_LOGIC);
END COMPONENT;
-- signal for connecting the output of the OR gate
-- with the input to the AND gate
        SIGNAL term1: STD_LOGIC;
                BEGIN
                        U0: myOR PORT MAP (D, V, term1);
                        S <= term1 AND M;
-- note how we can have both PORT MAP and signal assignment statements
        END Siren_Structural;
```

5. Conversion Routines

5.1 CONV_INTEGER()

Converts a std_logic_vector type to an integer;
Requires:

LIBRARY ieee;
USE ieee.std_logic_unsigned.ALL;
Syntax:
CONV_INTEGER(std_logic_vector)

Exemplo:

```
LIBRARY ieee;
USE ieee.std_logic_unsigned.ALL;
SIGNAL four_bit: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL n: INTEGER;
n := CONV_INTEGER(four_bit);
```

5.2 CONV_STD_LOGIC_VECTOR(,)

Converts an integer type to a std_logic_vector type.

Requires:
LIBRARY ieee;
USE ieee.std_logic_arith.ALL;

Sintaxe:

CONV_STD_LOGIC_VECTOR (integer, number_of_bits)

Exemplo:

```
LIBRARY ieee;
USE ieee.std_logic_arith.ALL;
SIGNAL four_bit: std_logic_vector(3 DOWNTO 0);
SIGNAL n: INTEGER;
four_bit := CONV_STD_LOGIC_VECTOR(n, 4);
```

PORTA AND DE 02 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY AND2gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END AND2gate;

ARCHITECTURE Dataflow OF AND2gate IS

        BEGIN
        f <= x AND y;
END Dataflow;
```

PORTA OR DE 02 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY OR2gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END OR2gate;

ARCHITECTURE Dataflow OF OR2gate IS
        BEGIN
        f <= x OR y;
END Dataflow;
```

PORTA NOT

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY NAND2gate IS PORT (
        x: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END NOTgate;

ARCHITECTURE Dataflow OF NOTgate IS
        BEGIN
        f <=  NOT x;
END Dataflow;
```

PORTA NAND DE 02 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY NAND2gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END NAND2gate;

ARCHITECTURE Dataflow OF NAND2gate IS
        BEGIN
        f <= x NAND y;
END Dataflow;
```

PORTA NOR DE 02 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY NOR2gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END NOR2gate;

ARCHITECTURE Dataflow OF NAND2gate IS
        BEGIN
        f <= x NOR y;
END Dataflow;
```

PORTA XOR DE 02 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY XOR2gate IS PORT (
```

```
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END XOR2gate;

ARCHITECTURE Dataflow OF XOR2gate IS
        BEGIN
        f <= x XOR y;
END Dataflow;
```

PORTA XNOR DE 02 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY XNOR2gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END XNOR2gate;

ARCHITECTURE Dataflow OF XNOR2gate IS
        BEGIN
        f <= x XNOR y;
END Dataflow;
```

PORTA NOR DE 03 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY NOR3gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        z: IN STD_LOGIC;
        f: OUT STD_LOGIC);
END NOR3gate;

ARCHITECTURE Dataflow OF NOR3gate IS
        SIGNAL xory, xoryorz : STD_LOGIC;
                BEGIN
                xory <= x OR y;
                xoryorz <= xory OR z;
                f <= NOT xoryorz;
END Dataflow;
```

PORTA NAND DE 03 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY NAND3gate IS PORT (
                x: IN STD_LOGIC;
                y: IN STD_LOGIC;
                z: IN STD_LOGIC;
                f: OUT STD_LOGIC);
        END NAND3gate;

ARCHITECTURE Dataflow OF NAND3gate IS
        SIGNAL xandy,xandyandz:STD_LOGIC;
                BEGIN
                xandy <= x AND y;
                xandyandz <= xandy AND z;
                f <= NOT xandyandz;
END Dataflow;
```

PORTA XOR DE 03 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY XOR3gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        z: IN STD_LOGIC;
        f: OUT STD_LOGIC);
        END XOR3gate;

ARCHITECTURE Dataflow OF XOR3gate IS
        SIGNAL xxory,xxoryxorz:STD_LOGIC;
                BEGIN
                xxory <= x XOR y;
                xxoryxorz <= xxory XOR z;
                f <= xxoryxorz;
END Dataflow;
```

PORTA XNOR DE 03 ENTRADAS

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY XNOR3gate IS PORT (
        x: IN STD_LOGIC;
        y: IN STD_LOGIC;
        z: IN STD_LOGIC;
        f: OUT STD_LOGIC);
        END XNOR3gate;

ARCHITECTURE Dataflow OF XNOR3gate IS
```

```
        SIGNAL xxory,xxoryxorz:STD_LOGIC;
                BEGIN
                xxory <= x XOR y;
                xxoryxorz <= xxory XOR z;
                f <= NOT xxoryxorz;
END Dataflow;
```

As funções booleanas podem ser programadas como a seguir :

S = (A B' C) + (A B C') + (A B C).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
        ENTITY Siren IS PORT (
        A: IN STD_LOGIC;
        B: IN STD_LOGIC;
        C: IN STD_LOGIC;
        S: OUT STD_LOGIC);
        END Siren;

ARCHITECTURE Dataflow OF Siren IS
        SIGNAL term_1, term_2, term_3: STD_LOGIC;
                BEGIN
                term_1 <= A AND (NOT B) AND C;
                term_2 <= A AND B AND (NOT C);
                term_3 <= A AND B AND C;
                S <= term_1 OR term_2 OR term_3;
END Dataflow;
```

1. AULA_1

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_E_02_ENTRADAS IS
        Port(a, b : IN BIT;
                c: OUT BIT);
END PORTA_E_02_ENTRADAS;

ARCHITECTURE Structural OF PORTA_E_02_ENTRADAS IS
COMPONENT AND2 PORT(x, y: IN BIT;
Z : OUT BIT);
END COMPONENT;

SIGNAL X1 : BIT;
BEGIN
        G1 : AND2 port map(a,b,X1);
END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_E_02_ENTRADAS IS
        Port(a, b : IN BIT;
                c: OUT BIT);
END PORTA_E_02_ENTRADAS;

ARCHITECTURE Structural OF PORTA_E_02_ENTRADAS IS
BEGIN
        C <= a AND b;
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_NAO IS
        Port(a : IN BIT;
                b: OUT BIT);
END PORTA_NAO;

ARCHITECTURE Structural OF PORTA_NAO IS
BEGIN
        b <= NOT a;
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_OU_02_ENTRADAS IS
        Port(a, b : IN BIT;
                c: OUT BIT);
END PORTA_OU_02_ENTRADAS;

ARCHITECTURE Structural OF PORTA_OU_02_ENTRADAS IS
BEGIN
        C <= a OR b;
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_XOR IS
        Port(a, b : IN BIT;
                x: OUT BIT);
END PORTA_XOR;


ARCHITECTURE Structural OF PORTA_XOR IS
BEGIN
        x <=  (a AND (NOT b)) OR ((NOT a) AND b);
END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_XNOR IS
        Port(a, b : IN BIT;
                x: OUT BIT);
END PORTA_XNOR;

ARCHITECTURE Structural OF PORTA_XNOR IS
BEGIN
        x <=  (a AND b) OR ((NOT a) AND (NOT b));
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_NE_02_ENTRADAS IS
        Port(a, b : IN BIT;
                x: OUT BIT);
END PORTA_NE_02_ENTRADAS;

ARCHITECTURE Structural OF PORTA_NE_02_ENTRADAS IS
BEGIN
        x <= NOT (a AND b);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_NE_03_ENTRADAS IS
        Port(a, b,c : IN BIT;
                x: OUT BIT);
END PORTA_NE_03_ENTRADAS;

ARCHITECTURE Structural OF PORTA_NE_03_ENTRADAS IS
BEGIN
        x <= NOT (a AND b AND c);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PORTA_NOU_02_ENTRADAS IS
        Port(a, b : IN BIT;
                c: OUT BIT);
END PORTA_NOU_02_ENTRADAS;

ARCHITECTURE Structural OF PORTA_NOU_02_ENTRADAS IS
BEGIN
        C <= NOT (a OR b);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY PORTA_NOU_03_ENTRADAS IS
        Port(a, b,c : IN BIT;
                x: OUT BIT);
END PORTA_NOU_03_ENTRADAS;

ARCHITECTURE Structural OF PORTA_NOU_03_ENTRADAS IS
BEGIN
        x <= NOT (a OR b OR c);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PARIDADE_PAR_03_VAR IS
        Port(a, b, c : IN BIT;
                x: OUT BIT);
END PARIDADE_PAR_03_VAR;

ARCHITECTURE Structural OF PARIDADE_PAR_03_VAR IS
BEGIN
        x <=  a XOR b XOR c;
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY PARIDADE_IMPAR_03_VAR IS
        Port(a, b, c : IN BIT;
                x: OUT BIT);
END PARIDADE_IMPAR_03_VAR;

ARCHITECTURE Structural OF PARIDADE_IMPAR_03_VAR IS
BEGIN
        x <=  NOT (a XOR b XOR c);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY funcao_booleana IS PORT(
A, B, C: IN BIT;
F: OUT BIT);
END funcao_booleana;

ENTITY inv IS
        Port(i : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <= NOT i ;
END Structural;
```

```
ENTITY myand2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myand2;
ARCHITECTURE Structural OF myand2 IS
BEGIN
        o <=  i1 AND i2;
END Structural;

ENTITY myor2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myor2;
ARCHITECTURE Structural OF myor2 IS
BEGIN
        o <=  i1 OR i2;
END Structural;

ARCHITECTURE Structural OF funcao_booleana IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myand2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myor2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL g,h,i,j,k,l,m,n,o,p,q : BIT;
BEGIN
U1: inv port map(A,g);
U2: inv port map(B,h);
U3: inv port map(C,i);
U4: myand2 port map(A, h, j);
U5: myand2 port map(b, g, k);
U6: myand2 port map(i, h, l);
U7: myand2 port map(A, B, m);
U8: myand2 port map(j, C, n);
U9: myand2 port map(m, C, o);
U10: myor2 port map(n, k, p);
U11: myor2 port map(l, o, q);
U12: myor2 port map(p, q, F);
END Structural;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY funcao_booleana_conjuntiva IS PORT(
A, B, C: IN BIT;
F: OUT BIT);
END funcao_booleana_conjuntiva;

ENTITY inv IS
        Port(i : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <= NOT i ;
END Structural;

ENTITY myand3 IS
        Port(i1, i2, i3 : IN BIT;
                o: OUT BIT);
END myand3;
ARCHITECTURE Structural OF myand3 IS
BEGIN
        o <=  i1 AND i2 AND i3;
END Structural;

ENTITY myor2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myor2;
ARCHITECTURE Structural OF myor2 IS
BEGIN
        o <=  i1 OR i2;
END Structural;

ARCHITECTURE Structural OF funcao_booleana_conjuntiva IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myor2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myand3 PORT(
i1, i2, i3: IN BIT;
o: OUT BIT);
END COMPONENT;
```

```
SIGNAL g,h,i,j,k,l,m,n : BIT;
BEGIN
U1: inv port map(A,g);
U2: inv port map(B,h);
U3: inv port map(C,i);
U4: myor2 port map(A, h, j);
U5: myor2 port map(h, g, k);
U6: myor2 port map(B, g, l);
U7: myor2 port map(j,C, m);
U8: myor2 port map(l, i, n);
U9: myand3 port map(m, k, n, F);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY funcao_booleana_conjuntiva_dataflow IS PORT(
A, B, C: IN BIT;
F: OUT BIT);
END funcao_booleana_conjuntiva_dataflow;

ARCHITECTURE DATAFLOW OF funcao_booleana_conjuntiva_dataflow  IS
        BEGIN
                F <= (A OR (NOT B) OR C) AND ((NOT A) OR (NOT B)) AND ((NOT A) OR B OR
(NOT C));
END DATAFLOW;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY funcao_booleana_conjuntiva_dataflow_1 IS PORT(
A, B, C: IN BIT;
F: OUT BIT);
END funcao_booleana_conjuntiva_dataflow_1;

ARCHITECTURE DATAFLOW OF funcao_booleana_conjuntiva_dataflow_1 IS
        BEGIN
                F <= NOT(A OR B OR C) OR NOT(NOT A OR B) OR NOT(A OR NOT B OR C) OR
NOT(NOT B OR NOT C);
END DATAFLOW;

-- A 4-to-1 8-bit multiplexer
LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY Multiplex_4_x_1 IS
PORT(S: IN std_logic_vector(1 downto 0); -- select lines
D0, D1, D2, D3: IN std_logic_vector(7 downto 0); -- data bus input
Y: OUT std_logic_vector(7 downto 0)); -- data bus output
END Multiplex_4_x_1;
```

```
-- using a process statement

ARCHITECTURE Behavioral OF Multiplex_4_x_1 IS
BEGIN
PROCESS (S,D0,D1,D2,D3)
BEGIN
CASE S IS
WHEN "00" => Y <= D0;
WHEN "01" => Y <= D1;
WHEN "10" => Y <= D2;
WHEN "11" => Y <= D3;
WHEN OTHERS => Y <= (OTHERS => 'U'); -- 8-bit vector of U
END CASE;
END PROCESS;
END Behavioral;

-- Um 8 para 1 8-bit multiplexador
LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY Multiplex_8_x_1 IS
PORT(
 s2,s1,s0 : IN BIT;  --linhas de seleção
 d0,d1,d2,d3,d4,d5,d6,d7 :IN BIT;  -- canais de entrada
Y: OUT BIT); -- saída
END Multiplex_8_x_1;

ARCHITECTURE Dataflow OF Multiplex_8_x_1 IS
        SIGNAL S : bit_vector(2 downto 0);
BEGIN
        S <= (S2 & S1 & S0);
        WITH S SELECT
        Y <=
                d0 WHEN "000",
                d1 WHEN "001",
                d2 WHEN "010",
                d3 WHEN "011",
                d4 WHEN "100",
                d5 WHEN "101",
                d6 WHEN "110",
                d7 WHEN "111";

END Dataflow;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;
ENTITY Multiplex_8_x_1_dataflow IS
PORT(S : IN std_logic_vector(2 downto 0); -- select lines
d0,d1,d2,d3,d4,d5,d6,d7: IN std_logic_vector(0 downto 0);  -- data bus input
Y: OUT std_logic_vector(0 downto 0)); -- data bus output
```

```
END Multiplex_8_x_1_dataflow;

-- using a process statement

ARCHITECTURE Dataflow OF Multiplex_8_x_1_dataflow IS
BEGIN

        WITH S SELECT
        Y <=
                d0 WHEN "000",
                d1 WHEN "001",
                d2 WHEN "010",
                d3 WHEN "011",
                d4 WHEN "100",
                d5 WHEN "101",
                d6 WHEN "110",
                d7 WHEN "111",
                (OTHERS => 'U') WHEN OTHERS; -- 8-bit vector of U

END Dataflow;

-- MULTIPLEX DE 8 CANAIS
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY mux_8_x_1_estrutural IS PORT(
s2, s1, s0: IN BIT; -- variaveis de seleção
d0,d1,d2,d3,d4,d5,d6,d7 : IN BIT; -- canais de entrada
E : IN BIT; -- enable
F: OUT BIT); -- saída booleana
END mux_8_x_1_estrutural;

ENTITY inv IS
        Port(i : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <= NOT i ;
END Structural;

ENTITY myand5 IS
        Port(i1, i2, i3, i4, i5 : IN BIT;
                o: OUT BIT);
END myand5;
ARCHITECTURE Structural OF myand5 IS
BEGIN
        o <=  i1 AND i2 AND i3 AND i4 AND i5;
END Structural;

ENTITY myor8 IS
```

```
        Port(i1, i2, i3, i4, i5, i6, i7,i8 : IN BIT;
                o: OUT BIT);
END myor8;
ARCHITECTURE Structural OF myor8 IS
BEGIN
        o <=  i1 OR i2 OR i3 OR i4 OR i5 OR i6 OR i7 OR i8;
END Structural;

ARCHITECTURE Structural OF mux_8_x_1_estrutural IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myand5 PORT(
i1, i2, i3, i4, i5: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myor8 PORT(
i1, i2, i3, i4, i5, i6, i7, i8: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL g,h,i,j,k,l,m,n,o,p,q : BIT;
BEGIN
U1: inv port map(s2,g);
U2: inv port map(s1,h);
U3: inv port map(s0,i);
U4: myand5 port map(g, h, i, E, d0, j);
U5: myand5 port map(g, h, s0, E, d1, k);
U6: myand5 port map(g, s1, i, E, d2, l);
U7: myand5 port map(g, s1, s0, E, d3, m);
U8: myand5 port map(s2, h, i, E, d4, n);
U9: myand5 port map(s2, h, s0, E, d5, o);
U10: myand5 port map(s2, s1, i, E, d6, p);
U11: myand5 port map(s2, s1, s0, E, d7, q);
U12: myor8 port map(j,k,l,m,n,o,p,q,F);
END Structural;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY demultiplex_3_x_8 IS
PORT(E : IN BIT;
 s2,s1,s0 : IN BIT;  --linhas de seleção
 O0,O1,O2,O3,O4,O5,O6,O7 : OUT BIT); -- saída
END demultiplex_3_x_8;

ARCHITECTURE Dataflow OF demultiplex_3_x_8 IS
```

```
    BEGIN
            O0 <= ((not s2) and (not s1)and (not s0))and E;
            O1 <= ((not s2) and (not s1) and s0)and E;
            O2 <= ((not s2) and (s1) and (not s0))and E;
            O3 <= ((not s2) and s1 and s0)and E;
            O4 <= (s2 and (not s1) and (not s0))and E;
            O5 <= (s2 and (not s1) and s0)and E;
            O6 <= (s2 and s1 and (not s0))and E;
            O7 <= (s2 and s1 and s0)and E;
    END Dataflow;

-- DEMULTIPLEX DE 3 POR 8 SAIDAS LÓGICA NEGATIVA
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY demux_3_x_8_neg_estrutural IS PORT(
s2, s1, s0: IN BIT; -- variaveis de seleção
NE : IN BIT; -- canal de entrada
o0,o1,o2,o3,o4,o5,o6,o7 : OUT BIT); -- saídas
END demux_3_x_8_neg_estrutural;

ENTITY inv IS
    Port(i : IN BIT;
        o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
    o <= NOT i ;
END Structural;

ENTITY mynand4 IS
    Port(i1, i2, i3, i4 : IN BIT;
        o: OUT BIT);
END mynand4;
ARCHITECTURE Structural OF mynand4 IS
BEGIN
    o <=  NOT(i1 AND i2 AND i3 AND i4);
END Structural;

ARCHITECTURE Structural OF demux_3_x_8_neg_estrutural IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT mynand4 PORT(
i1, i2, i3, i4: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL g,h,i,j: BIT;
```

```
BEGIN
U1: inv port map(s2,g);
U2: inv port map(s1,h);
U3: inv port map(s0,i);
U4: inv port map(NE,j);
U5: mynand4 port map(g, h, i,j,o0);
U6: mynand4 port map(g, h, s0,j,o1);
U7: mynand4 port map(g, s1,i,j,o2);
U8: mynand4 port map(g, s1,s0,j,o3);
U9: mynand4 port map(s2, h,i,j,o4);
U10: mynand4 port map(s2, h,s0,j,o5);
U11: mynand4 port map(s2,s1,i,j,o6);
U12: mynand4 port map(s2,s1,s0,j,o7);
END Structural;

-- DEMULTIPLEX DE 3 POR 8 SAIDAS LÓGICA POSITIVA
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY demux_3_x_8_pos_estrutural IS PORT(
s2, s1, s0: IN BIT; -- variaveis de seleção
E : IN BIT; -- canal de entrada
o0,o1,o2,o3,o4,o5,o6,o7 : OUT BIT); -- saídas
END demux_3_x_8_pos_estrutural;

ENTITY inv IS
        Port(i : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <= NOT i ;
END Structural;

ENTITY myand4 IS
        Port(i1, i2, i3, i4 : IN BIT;
                o: OUT BIT);
END myand4;
ARCHITECTURE Structural OF myand4 IS
BEGIN
        o <=  i1 AND i2 AND i3 AND i4;
END Structural;

ARCHITECTURE Structural OF demux_3_x_8_pos_estrutural IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myand4 PORT(
i1, i2, i3, i4: IN BIT;
```

```
o: OUT BIT);
END COMPONENT;

SIGNAL g,h,i: BIT;
BEGIN
U1: inv port map(s2,g);
U2: inv port map(s1,h);
U3: inv port map(s0,i);
U4: myand4 port map(g, h, i,E,o0);
U5: myand4 port map(g, h, s0,E,o1);
U6: myand4 port map(g, s1,i,E,o2);
U7: myand4 port map(g, s1,s0,E,o3);
U8: myand4 port map(s2, h,i,E,o4);
U9: myand4 port map(s2, h,s0,E,o5);
U10: myand4 port map(s2,s1,i,E,o6);
U11: myand4 port map(s2,s1,s0,E,o7);
END Structural;

-- SOMADOR COMPLETO DE 1 BIT - DATAFLOW
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY somador_completo_1bit_dataflow IS PORT (
ci, xi, yi: IN BIT; -- xi e yi são bits de entrada e ci vem um
co, si: OUT BIT); -- si soma e co vai um
END somador_completo_1bit_dataflow;

ARCHITECTURE Dataflow OF somador_completo_1bit_dataflow IS
BEGIN
co <= (xi AND yi) OR (ci AND (xi XOR yi));
si <= xi XOR yi XOR ci;
END Dataflow;

-- SOMADOR COMPLETO DE 1 BIT - MODO ESTRUTURAL
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY somador_completo_1bit_estrutural IS PORT(
x0, y0, ci: IN BIT; -- x0 e y0 = bit de entrada e ci = vem um
co,s0 : OUT BIT); -- s0 = saída e co = vai um
END somador_completo_1bit_estrutural;

ENTITY myand2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myand2;
ARCHITECTURE Structural OF myand2 IS
BEGIN
        o <=  i1 AND i2;
END Structural;

ENTITY myor2 IS
```

```
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myor2;
ARCHITECTURE Structural OF myor2 IS
BEGIN
        o <=  i1 OR i2;
END Structural;

ENTITY myxor2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myxor2;
ARCHITECTURE Structural OF myxor2 IS
BEGIN
        o <=  i1 XOR i2;
END Structural;

ARCHITECTURE Structural OF somador_completo_1bit_estrutural IS
COMPONENT myand2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myor2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myxor2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL a,b,c: BIT;

BEGIN
U1: myxor2 port map(x0,y0,a);
U2: myand2 port map(x0,y0,b);
U3: myxor2 port map(a,ci,s0);
U4: myand2 port map(a,ci,c);
U5: myor2 port map(b,c,co);
END Structural;


-- SUBTRATOR COMPLETO DE 1 BIT - DATAFLOW
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY subtrator_completo_1bit_dataflow IS PORT (
ci, xi, yi: IN BIT; -- xi e yi são bits de entrada e ci vem um
co, si: OUT BIT); -- si soma e co vai um
```

```
END subtrator_completo_1bit_dataflow;

ARCHITECTURE Dataflow OF subtrator_completo_1bit_dataflow IS
BEGIN
co <= ((NOT xi) AND (ci OR yi)) OR (yi AND ci);
si <= xi XOR yi XOR ci;
END Dataflow;

-- SUBTRATOR COMPLETO DE 1 BIT - MODO ESTRUTURAL
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY subtrator_completo_1bit_estrutural IS PORT(
x0, y0, ci: IN BIT; -- x0 e y0 = bit de entrada e ci = vem um
co,s0 : OUT BIT); -- s0 = saída e co = vai um
END subtrator_completo_1bit_estrutural;

ENTITY myand2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myand2;
ARCHITECTURE Structural OF myand2 IS
BEGIN
        o <=  i1 AND i2;
END Structural;

ENTITY myor2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myor2;
ARCHITECTURE Structural OF myor2 IS
BEGIN
        o <=  i1 OR i2;
END Structural;

ENTITY myxor2 IS
        Port(i1, i2 : IN BIT;
                o: OUT BIT);
END myxor2;
ARCHITECTURE Structural OF myxor2 IS
BEGIN
        o <=  i1 XOR i2;
END Structural;

ENTITY inv IS
        Port(i1 : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <=  NOT i1;
```

```
END Structural;

ARCHITECTURE Structural OF subtrator_completo_1bit_estrutural IS
COMPONENT myand2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myor2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myxor2 PORT(
i1, i2: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT inv PORT(
i1: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL a,b,c,d,e: BIT;

BEGIN
U1: myxor2 port map(x0,y0,a);
U2: myand2 port map(ci,y0,b);
U3: myxor2 port map(a,ci,s0);
U4: myor2 port map(y0,ci,c);
U5: inv port map(x0,d);
U6: myand2 port map(d,c,e);
U7: myor2 port map(b,e,co);
END Structural;

LIBRARY ieee;
USE ieee.std_logic_1164.all;
-- O seguinte pacote é necessário para os sinais STD_LOGIC_VECTOR
-- A e B podem ser usados como operações aritméticas de numeros não assinalados.
USE ieee.std_logic_unsigned.all;
ENTITY ula_4bit_dataflow IS PORT (
S: IN std_logic_vector(2 downto 0); -- seleção das operações
A, B: IN std_logic_vector(3 downto 0); -- operandos de entrada
F: OUT std_logic_vector(3 downto 0)); -- saída
END ula_4bit_dataflow;
ARCHITECTURE Dataflow OF ula_4bit_dataflow IS
BEGIN
PROCESS(S, A, B)
BEGIN
CASE S IS
```

```
WHEN "000" => -- passagem de A através
F <= A;
WHEN "001" => -- AND
F <= A AND B;
WHEN "010" => -- OR
F <= A OR B;
WHEN "011" => -- NOT A
F <= NOT A;
WHEN "100" => -- soma
F <= A + B;
WHEN "101" => -- subtrai
F <= A - B;
WHEN "110" => -- incremento
F <= A + 1;
WHEN OTHERS => -- decremento
F <= A - 1;
END CASE;
END PROCESS;
END Dataflow;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY decodificador_3_x_8_neg_dataflow IS
PORT(NE : IN BIT; -- enable
 s2,s1,s0 : IN BIT;  --linhas de seleção
 O0,O1,O2,O3,O4,O5,O6,O7 : OUT BIT); -- saída
END decodificador_3_x_8_neg_dataflow;

ARCHITECTURE Dataflow OF decodificador_3_x_8_neg_dataflow IS
        BEGIN
                O0 <= NOT((not s2) and (not s1)and (not s0))and (NOT NE);
                O1 <= NOT((not s2) and (not s1) and s0)and (NOT NE);
                O2 <= NOT((not s2) and (s1) and (not s0))and (NOT NE);
                O3 <= NOT((not s2) and s1 and s0)and (NOT NE);
                O4 <= NOT(s2 and (not s1) and (not s0))and (NOT NE);
                O5 <= NOT(s2 and (not s1) and s0)and (NOT NE);
                O6 <= NOT(s2 and s1 and (not s0))and (NOT NE);
                O7 <= NOT(s2 and s1 and s0)and (NOT NE);
        END Dataflow;

-- DECODIFICADOR DE 3 POR 8 SAIDAS LÓGICA NEGATIVA
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY decodificador_3_x_8_neg_estrutural IS PORT(
s2, s1, s0: IN BIT; -- variaveis de seleção
NE : IN BIT; -- enable
o0,o1,o2,o3,o4,o5,o6,o7 : OUT BIT); -- saídas
END decodificador_3_x_8_neg_estrutural;
```

```
ENTITY inv IS
        Port(i : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <= NOT i ;
END Structural;

ENTITY mynand4 IS
        Port(i1, i2, i3, i4 : IN BIT;
                o: OUT BIT);
END mynand4;
ARCHITECTURE Structural OF mynand4 IS
BEGIN
        o <=  NOT(i1 AND i2 AND i3 AND i4);
END Structural;

ARCHITECTURE Structural OF decodificador_3_x_8_neg_estrutural IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT mynand4 PORT(
i1, i2, i3, i4: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL g,h,i,j: BIT;
BEGIN
U1: inv port map(s2,g);
U2: inv port map(s1,h);
U3: inv port map(s0,i);
U4: inv port map(NE,j);
U5: mynand4 port map(g, h, i,j,o0);
U6: mynand4 port map(g, h, s0,j,o1);
U7: mynand4 port map(g, s1,i,j,o2);
U8: mynand4 port map(g, s1,s0,j,o3);
U9: mynand4 port map(s2, h,i,j,o4);
U10: mynand4 port map(s2, h,s0,j,o5);
U11: mynand4 port map(s2,s1,i,j,o6);
U12: mynand4 port map(s2,s1,s0,j,o7);
END Structural;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;

ENTITY decodificador_3_x_8_pos_dataflow IS
PORT(E : IN BIT;
```

```
 s2,s1,s0 : IN BIT;  --linhas de seleção
 O0,O1,O2,O3,O4,O5,O6,O7 : OUT BIT); -- saída
END decodificador_3_x_8_pos_dataflow;

ARCHITECTURE Dataflow OF decodificador_3_x_8_pos_dataflow IS
        BEGIN
                        O0 <= ((not s2) and (not s1)and (not s0))and E;
                        O1 <= ((not s2) and (not s1) and s0)and E;
                        O2 <= ((not s2) and (s1) and (not s0))and E;
                        O3 <= ((not s2) and s1 and s0)and E;
                        O4 <= (s2 and (not s1) and (not s0))and E;
                        O5 <= (s2 and (not s1) and s0)and E;
                        O6 <= (s2 and s1 and (not s0))and E;
                        O7 <= (s2 and s1 and s0)and E;
        END Dataflow;

-- DECODIFICADOR DE 3 POR 8 SAIDAS LÓGICA POSITIVA
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY decodificador_3_x_8_pos_estrutural IS PORT(
s2, s1, s0: IN BIT; -- variaveis de seleção
E : IN BIT; -- enable
o0,o1,o2,o3,o4,o5,o6,o7 : OUT BIT); -- saídas
END decodificador_3_x_8_pos_estrutural;

ENTITY inv IS
        Port(i : IN BIT;
                o: OUT BIT);
END inv;
ARCHITECTURE Structural OF inv IS
BEGIN
        o <= NOT i ;
END Structural;

ENTITY myand4 IS
        Port(i1, i2, i3, i4 : IN BIT;
                o: OUT BIT);
END myand4;
ARCHITECTURE Structural OF myand4 IS
BEGIN
        o <=  i1 AND i2 AND i3 AND i4;
END Structural;

ARCHITECTURE Structural OF decodificador_3_x_8_pos_estrutural IS
COMPONENT inv PORT(
i: IN BIT;
o: OUT BIT);
END COMPONENT;

COMPONENT myand4 PORT(
```

```
i1, i2, i3, i4: IN BIT;
o: OUT BIT);
END COMPONENT;

SIGNAL g,h,i: BIT;
BEGIN
U1: inv port map(s2,g);
U2: inv port map(s1,h);
U3: inv port map(s0,i);
U4: myand4 port map(g, h, i,E,o0);
U5: myand4 port map(g, h, s0,E,o1);
U6: myand4 port map(g, s1,i,E,o2);
U7: myand4 port map(g, s1,s0,E,o3);
U8: myand4 port map(s2, h,i,E,o4);
U9: myand4 port map(s2, h,s0,E,o5);
U10: myand4 port map(s2,s1,i,E,o6);
U11: myand4 port map(s2,s1,s0,E,o7);
END Structural;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;
ENTITY codificador_3_x_8_neg_dataflow IS
PORT(t1,t2,t3,t4,t5,t6,t7 : IN BIT;  --linhas de entradas
 NO0,NO1,NO2 : OUT BIT); -- saída
END codificador_3_x_8_neg_dataflow;

ARCHITECTURE Dataflow OF codificador_3_x_8_neg_dataflow IS
        BEGIN
                        NO0 <= NOT(t1 or t3 or t5 or t7);
                        NO1 <= NOT(t2 or t3 or t6 or t7);
                        NO2 <= NOT(t4 or t5 or t6 or t7);
        END Dataflow;

-- CODIFICADOR DE 3 POR 8 SAIDA OCTAL
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY codificador_3_x_8_neg_estrutural IS PORT(
t1, t2, t3, t4, t5, t6, t7 : IN BIT; -- variaveis de seleção
No0,No1,No2 : OUT BIT); -- saídas
END codificador_3_x_8_neg_estrutural;

ENTITY mynor4 IS
        Port(i1, i2, i3, i4 : IN BIT;
                o: OUT BIT);
END mynor4;
ARCHITECTURE Structural OF mynor4 IS
BEGIN
        o <=  NOT (i1 OR i2 OR i3 OR i4);
END Structural;
```

```
ARCHITECTURE Structural OF codificador_3_x_8_neg_estrutural IS

COMPONENT mynor4 PORT(
i1, i2, i3, i4: IN BIT;
o: OUT BIT);
END COMPONENT;

BEGIN
U1: mynor4 port map(t1,t3,t5,t7,No0);
U2: mynor4 port map(t2,t3,t6,t7,No1);
U3: mynor4 port map(t4,t5,t6,t7,No2);
END Structural;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;
ENTITY codificador_3_x_8_pos_dataflow IS
PORT(t1,t2,t3,t4,t5,t6,t7 : IN BIT;  --linhas de entradas
 O0,O1,O2 : OUT BIT); -- saída
END codificador_3_x_8_pos_dataflow;

ARCHITECTURE Dataflow OF codificador_3_x_8_pos_dataflow IS
        BEGIN
                        O0 <= t1 or t3 or t5 or t7;
                        O1 <= t2 or t3 or t6 or t7;
                        O2 <= t4 or t5 or t6 or t7;
        END Dataflow;

-- CODIFICADOR DE 3 POR 8 SAIDA OCTAL
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY codificador_3_x_8_pos_estrutural IS PORT(
t1, t2, t3, t4, t5, t6, t7 : IN BIT; -- variaveis de seleção
o0,o1,o2 : OUT BIT); -- saídas
END codificador_3_x_8_pos_estrutural;

ENTITY myor4 IS
        Port(i1, i2, i3, i4 : IN BIT;
                o: OUT BIT);
END myor4;
ARCHITECTURE Structural OF myor4 IS
BEGIN
        o <=  i1 OR i2 OR i3 OR i4;
END Structural;

ARCHITECTURE Structural OF codificador_3_x_8_pos_estrutural IS

COMPONENT myor4 PORT(
i1, i2, i3, i4: IN BIT;
o: OUT BIT);
END COMPONENT;
```

```
BEGIN
U1: myor4 port map(t1,t3,t5,t7,o0);
U2: myor4 port map(t2,t3,t6,t7,o1);
U3: myor4 port map(t4,t5,t6,t7,o2);
END Structural;

LIBRARY ieee;
USE IEEE.std_logic_1164.all;
ENTITY codificador_3_x_8_prioridade_dataflow IS
PORT(t1,t2,t3,t4,t5,t6,t7 : IN BIT;  --linhas de entradas
 O0,O1,O2 : OUT BIT); -- saída
END codificador_3_x_8_prioridade_dataflow;

ARCHITECTURE Dataflow OF codificador_3_x_8_prioridade_dataflow IS
        BEGIN
        O0 <= (t1 AND NOT T2 AND NOT T3 AND NOT T4 AND NOT T5 AND NOT T6 AND NOT
        T7) OR (t3 AND NOT T4 AND NOT T5 AND NOT T6 AND NOT T7) OR (t5 AND NOT T6
        AND NOT T7) OR t7;
        O1 <= (t2 AND NOT T3 AND NOT T4 AND NOT T5 AND NOT T6 AND NOT T7) OR (t3
        AND NOT T4 AND NOT T5 AND NOT T6 AND NOT T7) OR (t6 AND NOT T7) OR t7;
        O2 <= (t4 AND NOT T5 AND NOT T6 AND NOT T7) OR (t5 AND NOT T6 AND NOT T7) OR
        (t6 AND NOT T7) OR t7;
        END Dataflow;
```

Tipos de arquitetura de processadores (Overview)

Mostramos neste tópico alguns conceitos importantes sobre o funcionamento interno dos processadores. Tomaremos como exemplo os processadores Intel, e com eles você entenderá conceitos como execução especulativa, pipeline, previsão de desvio, paralelismo, micro-operações, linguagem assembly, memória virtual, paginação e outros termos complexos.

Registradores internos do processador

Para entender como um processador executa programas, precisamos conhecer a sua arquitetura interna, do ponto de vista de software. Dentro de um processador existem vários circuitos chamados de registradores. Os registradores funcionam como posições de memória, porém o seu acesso é extremamente rápido, muito mais veloz que o da cache L1. O número de bits dos registradores depende do processador.

- Processadores de 8 bits usam registradores de 8 bits;
- Processadores de 16 bits usam registradores de 16 bits;
- Processadores de 32 bits usam registradores de 32 bits;
- Processadores de 64 bits usam registradores de 64 bits.

A figura 1 mostra os registradores internos dos processadores 8086, 8088 e 80286, todos de 16 bits. Todos os processadores têm uma linguagem baseada em códigos numéricos na memória. a. Cada código significa uma instrução. Por exemplo, podemos ter uma instrução para somar o valor de AX com o valor de BX e guardar o resultado em AX. As instruções do processador que encontramos na memória são o que chamamos de linguagem de máquina. Nenhum programador consegue criar programas complexos usando a linguagem de máquina, pois ela é formada por códigos numéricos. É verdade que alguns programadores conseguem fazer isso, mas não para programas muito longos, pois tornam-se difíceis de entender e de gerenciar. Ao invés disso, são utilizados códigos representados por siglas. As siglas são os nomes das instruções, e os operandos dessas instruções são os registradores, valores existentes na memória e valores constantes.

FIGURA 8.1
Registradores internos do processador 8086.

Por exemplo, a instrução que acabamos de citar, que soma o valor dos registradores AX e BX e guarda o resultado em AX, é representada por:

```
ADD AX,BX
```

Esta instrução é representada na memória pelo seguinte código de máquina:

```
01 D8
```

Linguagem de montagem – A linguagem de montagem "assembly" é usada para escrever programas que têm contato direto com o hardware, como o BIOS e drivers. O "assembly" também é chamado linguagem de baixo nível, pois interage intimamente com o hardware. Programas que não necessitam deste contato direto com o hardware não precisam ser escritos em "assembly", e são em geral escritos em linguagens como C, Pascal, Delphi, Basic e diversas outras. Essas são chamadas linguagens de alto nível. Nas linguagens de alto nível, não nos preocupamos com os registradores do processador, nem com a sua arquitetura interna. Os programas pensam apenas em dados, matrizes, arquivos, telas, etc. **ontagem (assembly)**

A linguagem "assembly" é usada para escrever programas que têm contato direto com o hardware, como o BIOS e drivers. O "assembly" também é chamado linguagem de baixo nível, pois interage intimamente com o hardware. Programas que não necessitam deste contato direto com o hardware não precisam ser escritos em "assembly", e são em geral escritos em linguagens como C, Pascal, Delphi, Basic e diversas outras. Essas são chamadas linguagens de alto nível. Nas linguagens de alto nível, não nos preocupamos com os registradores do processador, nem com a sua arquitetura interna. Os programas pensam apenas em dados, matrizes, arquivos, telas, etc.

Em cada linha deste programa temos na parte esquerda, os endereços, formados por duas partes (segmento e offset). A seguir temos uma instrução em código de máquina, e finalmente a instrução em "assembly".

Endereço	Código Assembly	Mnemonica
1B8D	0100 01D8	ADD AX,BX

Quando estamos programando em linguagem "assembly", escrevemos apenas os nomes das instruções. Depois de escrever o programa, usando um editor de textos comum, usamos um programa chamado compilador de linguagem "assembly", ou simplesmente, montador "Assembler". O que este programa faz é ler o arquivo com as instruções (arquivo fonte) e gerar um arquivo contendo apenas os códigos das instruções, em linguagem de máquina (arquivo objeto). O arquivo objeto passa ainda por um processo chamado link edição, e finalmente se transforma em um programa, que pode ser executado pelo processador. O Assembler também gera um arquivo de impressão, contendo os endereços, códigos e instruções em "assembly", como no trecho de listagem que mostramos acima. O programador pode utilizar esta listagem para depurar o programa, ou seja, testar o seu funcionamento. Os códigos hexadecimais que representam as instruções do processador são chamados de opcodes. As siglas que representam essas instruções são chamadas de mnemônicos.

Arquitetura do primeiro microprocessador Intel 8080

Constituído de 6 registradores de uso geral de 8 bits, um acumulador de 8 bits, registrador de status de 8 bits da ULA, apontador de pilha de 16 bits e um contador de instrução de 16 bits.

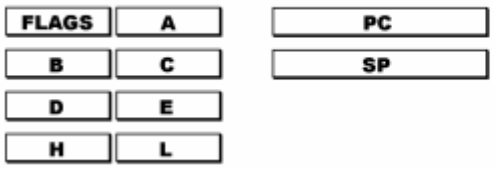


FIGURA 8.2
Registradores internos do 8080.

Códigos das instruções do 8080

Apresentamos a seguir uma tabela com os códigos de todas as instruções do 8080. Não que você vá programar 8080, mas para que você tenha uma idéia da relação entre as instruções e os seus códigos. Na tabela que se segue, temos as seguintes convenções:

- D8 representa um dado constante de 8 bits;
- D16 representa um dado constante de 16 bits;
- Addr representa um endereço de 16 bits.

A seguir apresentamos algumas instruções do conjunto de instruções (Set of instructions) do microprocessador 8080.

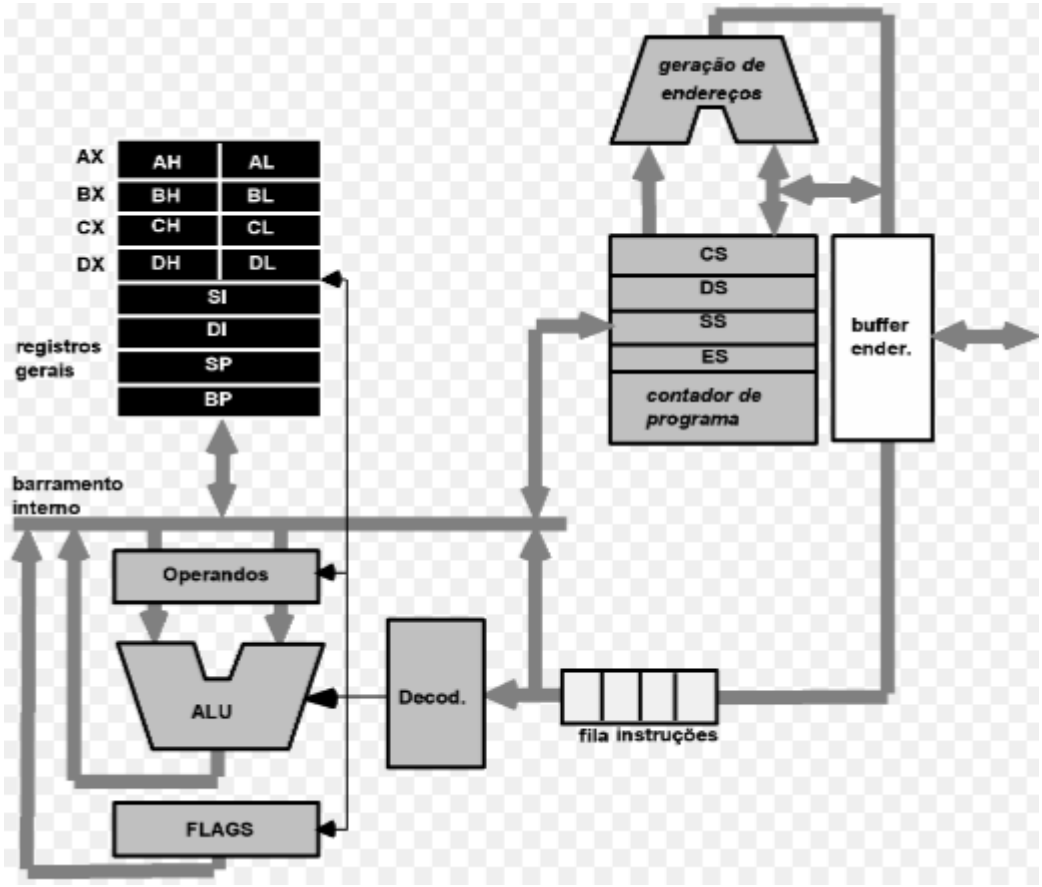
Op Code	Mnemonic	Op Code	Mnemonic	Op Code	Mnemonic	Op Code	Mnemonic	Op Code	Mnemonic	Op Code	Mnemonic
00	NOP	2B	DCX H	56	MOV D,M	81	ADD C	AC	XRA H	D7	RST 2
01	LXI B,D16	2C	INR L	57	MOV D,A	82	ADD D	AD	XRA L	D8	RC
02	STAX B	2D	DCR L	58	MOV E,B	83	ADD E	AE	XRA M	D9	-
03	INX B	2E	MVI L,D8	59	MOV E,C	84	ADD H	AF	XRA A	DA	JC ADDR
04	INR B	2F	CMA	5A	MOV E,D	85	ADD L	B0	ORA B	DB	IN D8
05	DCR B	30	-	5B	MOV E,E	86	ADD M	B1	ORA C	DC	CC ADDR
06	MVI B,D8	31	LXI SP,d16	5C	MOV E,H	87	ADD A	B2	ORA D	DD	-

Arquitetura do 8086

Arquitetura utilizada na era dos PCs, com o "assembly" do processador 8086. Os seus registradores internos são de 16 bits, mas foram inspirados nos registradores do 8080. Na figura 7, os registradores indicados em branco são "herdados" do 8080, enquanto os indicados em cinza são novos, próprios do 8086.

FIGURA 8.7
Registradores internos do 8086.

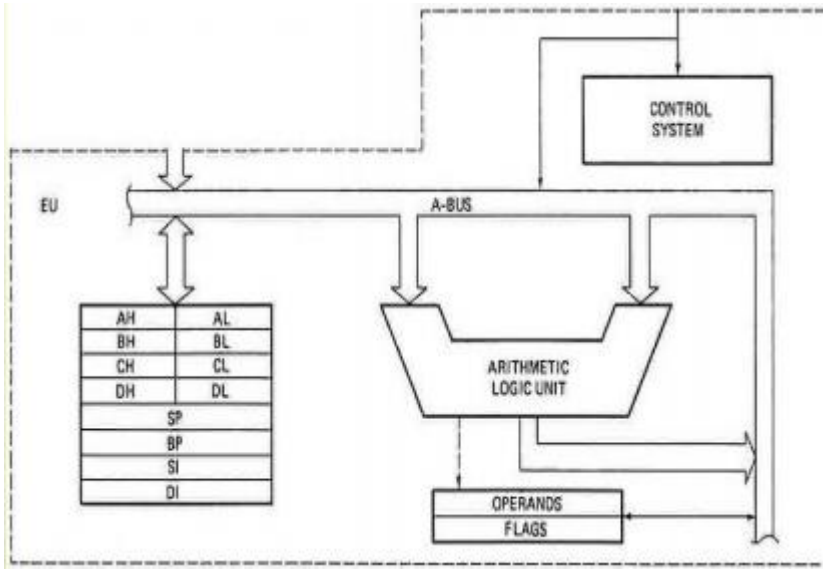
Depois de converter os antigos programas "assembly" de 8080 para 8086, os produtores de software passaram a criar programas novos já usando os recursos mais avançados do 8086, resultando em programas mais eficientes. Programas em linguagem de alto nível (C, Pascal, etc.) podiam ser convertidos com mais facilidade, já que eram desvinculados do "assembly". A seguir apresentamos uma arquitetura pipeline do 8086 localizada na fila de instruções "queue".



A arquitetura do 8086 possui 2 unidades chamadas de BIU e EU. A BIU é a unidade responsável pela

- Busca da instrução;
- Leitura e escrita de operandos na memória;
- Cálculo dos endereços dos operandos.

Os bytes das instruções são transferidas para o sistema byte queue (sequenciamento das instruções a serem executadas pela unidade de execução EU). A unidade EU é responsável pela execução das instruções fornecidas pelo queue. A BIU contém o "queue", registrador de segmento, apontador de instrução (IP), endereço somador e contém circuito de controle, decodificador de instrução, ALU, registradores apontadores e de índice, registrador Flag. A seguir a parte da arquitetura do 8086 referente a unidade EU de execução.



Arquitetura do 80286

O 80286 também é um processador de 16 bits. Possui os mesmos registradores internos existentes no 8086. Entretanto possui algumas novas instruções, bem como um novo modo de endereçamento capaz de operar com 16 MB de memória, o que era uma quantidade espantosa para a época do seu lançamento (1982), quando a maioria dos computadores tinha 64 kB de memória. O 80286 podia operar em duas modalidades. O chamado modo real (8086 real address mode) permite endereçar até 1 MB de memória. Nesse caso o processador comporta-se como um 8086, apenas acrescido de algumas novas instruções. Para uso em sistemas operacionais mais avançados, o 80286 podia operar no modo protegido (protected virtual address mode). Neste modo, o processador pode operar com 16 MB de memória física e até 1 GB de memória virtual por tarefa. Multitarefa O 80286 foi criado visando facilitar a multiprogramação ou multitarefa, na qual vários programas podem ser executados "simultaneamente". O que ocorre é uma divisão do tempo entre os vários processos que estão sendo executados. Uma forma simples de dividir o tempo é alocar períodos iguais (10 milésismos de segundo, por exemplo), e distribuir esses períodos entre os processos. Quando um processo começa a ser executado, será interrompido 10 ms depois, e o sistema operacional deve fazer com que o processador dê atenção ao processo seguinte. Desta forma usando um esquema de "rodízio", todos os processos são executados ao mesmo tempo, porém em cada instante um só está efetivamente em execução, e os demais estão aguardando. O período no qual o processador está dedicado a um processo é chamado time slice.

Modo protegido

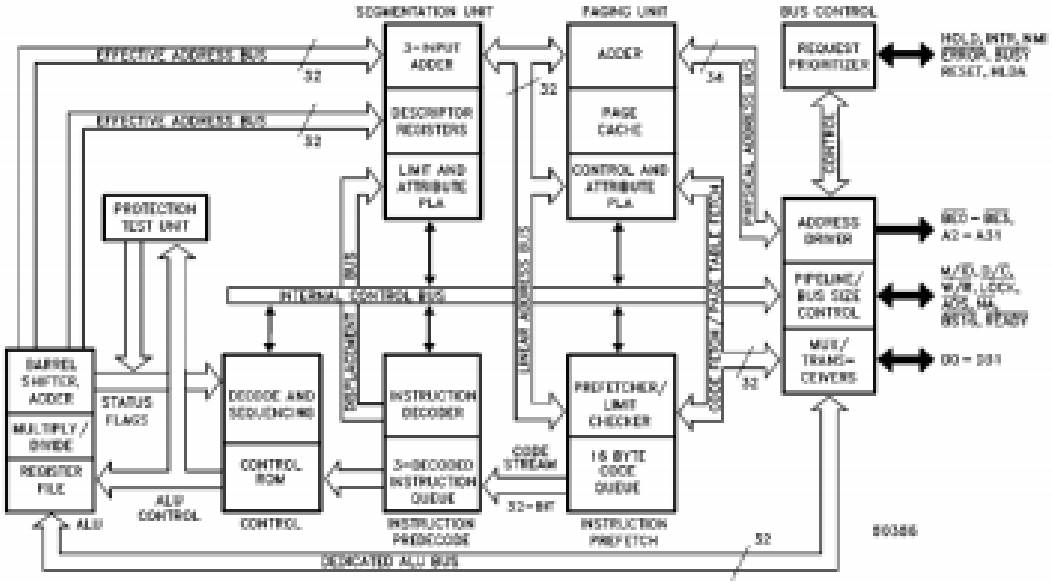
O 80286 passa a ter novos recursos para gerenciamento de tarefas e endereçamento de memória quando opera no modo protegido. Lembre que no 8086, cada segmento tinha 64 kB, e era definido pelos registradores de segmento (CS, DS, ES e SS). Todos os segmentos eram contidos dentro da memória física de 1 MB. No 286 operando em modo protegido, os segmentos também têm 64 kB, e são definidos por um registrador de segmento (CS, DS, ES e SS) e um offset. A diferença está na formação desses endereços. Consideremos por exemplo o endereço F000:1000 no modo real. Conforme mostramos neste capítulo, o endereço absoluto correspondente é F1000. É obtido acrescentando um zero hexadecimal (ou 4 zeros binários) à direita do segmento e somando o resultado com o offset. O resultado terá 20 bits, permitindo endereçar até 1 MB. Em suma uma expansão na linha de endereços e consequentemente acesso a uma memória de tamanho expandido. As tarefas (tasks) no 286 recebem um identificador de privilégio que varia de 0 a 3. O privilégio 0 é dado ao

núcleo do sistema operacional. É o único nível que permite gerenciar parâmetros das demais tarefas, tendo acesso a todas as instruções de gerenciamento de memória e de tarefas. Os níveis de privilégio 1 e 2 são usados pelo sistema operacional, e o nível 3 é dado às aplicações. Isso impede que um programa de um usuário possa com o gerenciamento de memória e de tarefas. Note que esses recursos só estão disponíveis no modo protegido.

O modo real é bastante limitado. Lembra muito a operação dos processadores de 8 bits. Já o modo protegido tem características de computadores mais poderosos. Recursos antes encontrados apenas em computadores de grande porte passariam a fazer parte dos microcomputadores. Era tido como óbvia a criação de novos sistemas operacionais mais avançados, operando em modo protegido.

Arquitetura do 80386

É muito importante conhecer bem o 386, pois todos os processadores posteriores, do 486 ao Pentium 4, utilizam métodos semelhantes de gerenciamento de memória e multitarefa, bem como possuem conjuntos de instruções similares, com apenas algumas poucas diferenças. A figura mostra o diagrama interno do 386. Além de vários elementos encontrados em outros processadores, destacam-se as unidades de gerenciamento de memória (segmentation unit e paging unit) e a unidade de proteção (protection test unit), necessária ao funcionamento da multitarefa.



*** 100%
FIGURA
8.14

Diagrama interno do processador 80386.

Modo virtual 8086

Este é um novo modo de operação introduzido no 386, compatível com o modo real e com o modo protegido simultaneamente. Não é na verdade um novo modo de operação, pois faz parte do modo protegido. Apenas as tarefas designadas a operar neste modo têm atributos específicos que alteram a formação dos endereços e o modo como os registradores de segmento são tratados. Programas escritos para o modo real (exemplo: MS-DOS e seus aplicativos) não funcionam no modo protegido. Basta lembrar que o sistema de endereçamento é completamente diferente.
Durante o projeto do 80386, o IBM PC e o MS-DOS haviam assumido fortes posições no mercado, portanto a Intel teve a preocupação de tornar o seu novo processador compatível com este sistema. O 80386 não apenas permite comutar entre o modo real e o modo virtual (tornando possível usar a memória estendida no ambiente DOS, bem como usar programas de modo protegido e ainda assim ter

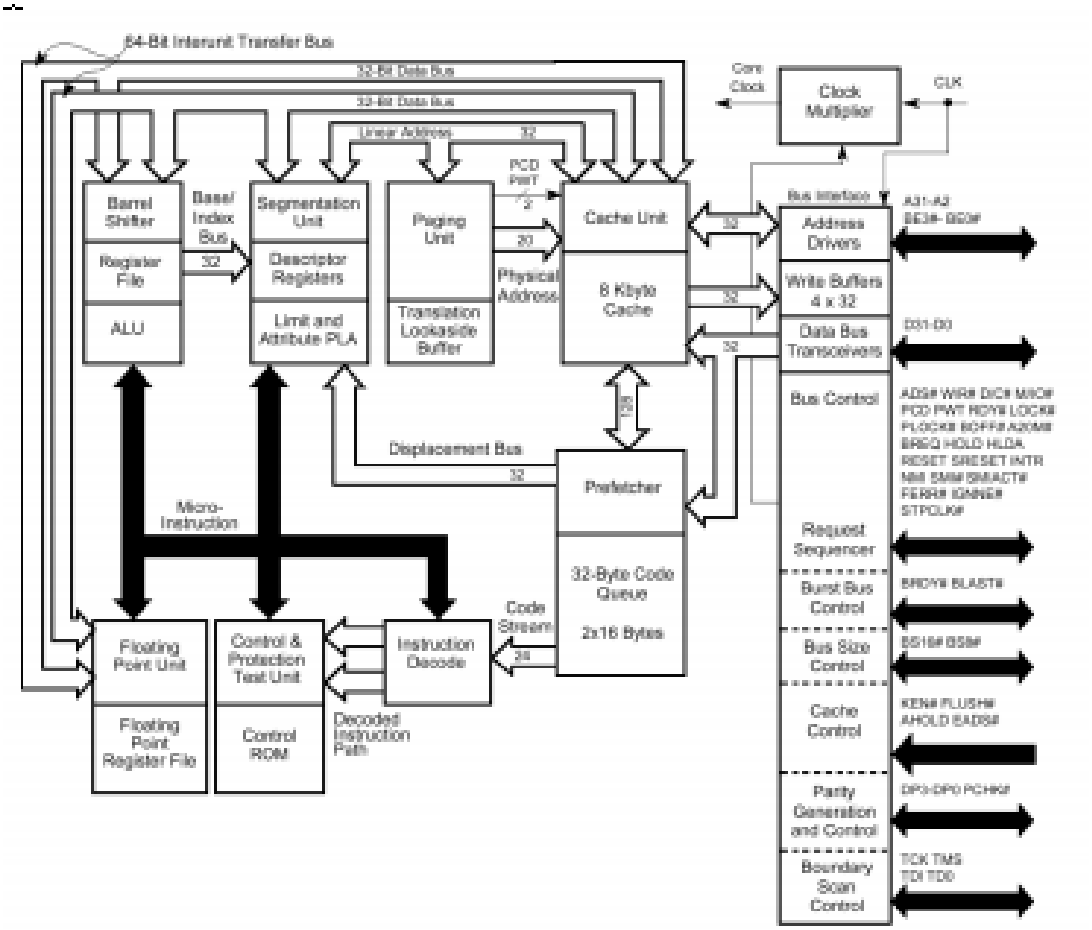
acesso às funções do DOS, que operam em modo real), mas também permite que tarefas no modo protegido possam operar no modo virtual 8086.

Multitarefa

Um computador pode executar vários programas ao mesmo tempo, compartilhando seu tempo entre todos eles. Este tipo de ambiente é chamado de multitarefa (multitask). Assim como o 286, o 80386 tem no seu modo protegido, todos os recursos para operar em um ambiente multitarefa.

Arquitetura do 80486

Explicando em poucas palavras, um processador 80486 é similar a um 80386, acrescido de um coprocessador matemático 80387, mais 8 kB de cache L1 integrada. Existem, entretanto, outras diferenças na arquitetura interna, sendo a principal delas, o aumento do número de estágios pipeline. A figura 20 mostra o diagrama interno do 486.



*** 75%
FIGURA
8.20

Diagrama interno do 486.

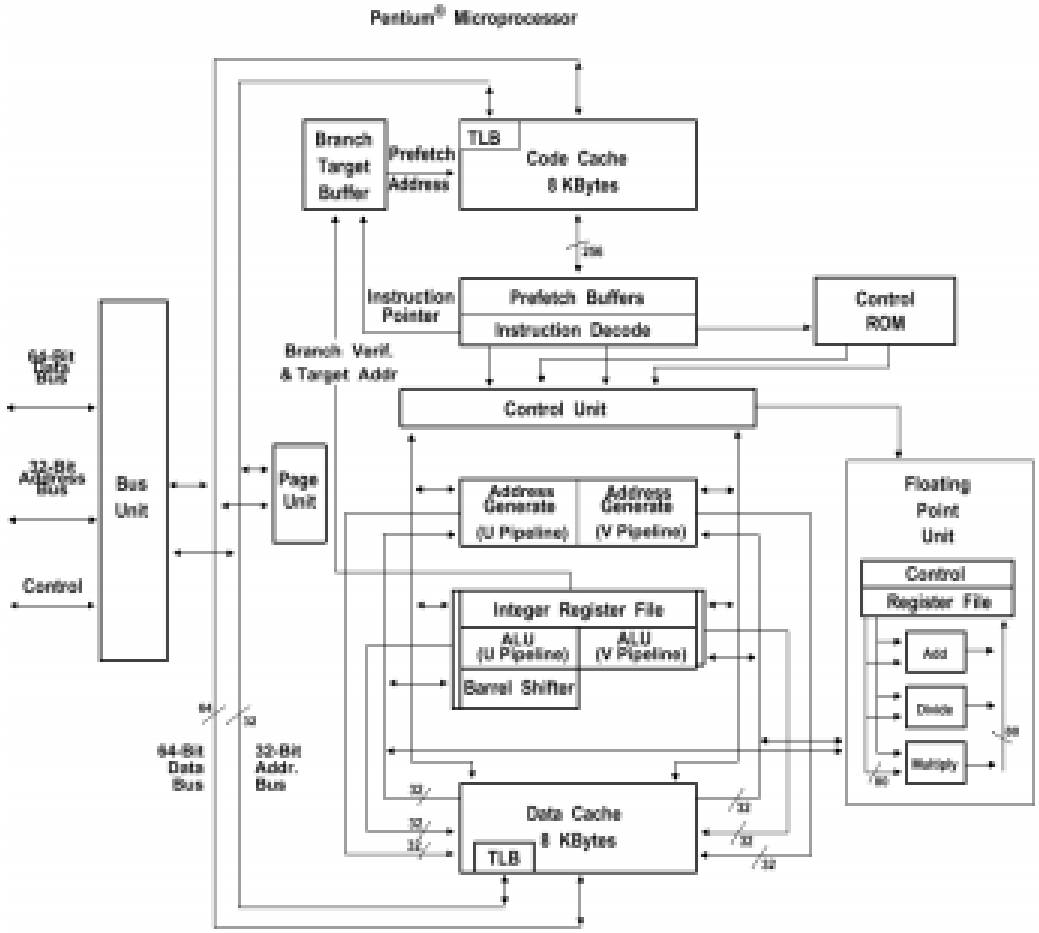
Entre as principais diferenças em relação ao 386, notamos a cache de 8 kB, ligada à unidade de pré-busca por um caminho de 128 bits. A fila de instruções teve o tamanho dobrado para 32 bytes. As transferências internas de dados são agora feitas por dois caminhos de 32 bits. O pipeline do 486 realmente traz um grande aumento de desempenho. Apesar de terem arquiteturas muito parecidas (sendo o pipeline mais avançado, a principal diferença), um 486 é duas vezes mais rápido que um 386 de mesmo clock.

Arquitetura do Pentium

Além do aumento de clock, o uso de arquitetura pipeline foi utilizada nos processadores 386 e 486 para aumentar o desempenho. O Pentium também tem suas operações de decodificação e execução realizadas por 5 estágios, tal qual o 486. A grande evolução é a introdução da arquitetura superescalar, através da qual podem ser executadas duas instruções ao mesmo tempo. Podemos ver na figura 22 o diagrama do Pentium, no qual encontramos os módulos U-Pipeline e V-Pipeline. Esses dois módulos operam de forma simultânea, e graças a eles é possível executar duas instruções ao mesmo tempo. Outro melhoramento é a adoção do barramento de dados com 64 bits, com a qual é possível reduzir os efeitos da lentidão da memória RAM. O Pentium tem ainda uma cache L1 maior, dividida em duas seções independentes, sendo uma para código e outra para dados. A unidade de ponto flutuante foi reprojetada, e é muito mais rápida que a do 486.

** 75%
FIGURA
8.22

Diagrama interno do Pentium.



Arquitetura superescalar

Uma arquitetura superescalar é aquela na qual múltiplas instruções podem ser executadas simultaneamente. Podemos ver na figura 23 como os pipelines U e V do Pentium executam instruções. A execução simultânea é possível desde que se tratem de instruções independentes, ou seja, que a execução da segunda operação não dependa do resultado da primeira. Observe que no instante t1, as instruções I1 e I2 são finalizadas, e em cada um dos instantes seguintes, duas novas instruções são finalizadas. Desta forma o Pentium pode teoricamente executar duas instruções a cada ciclo. Nem sempre é possível executar instruções em paralelo. Os programas são formados por instruções seqüenciais ou seja, um programa é uma seqüência de instruções. O paralelismo funciona entretanto

em boa parte dos casos, pois mesmo com instruções seqüencias, muitas são independentes. Veja por exemplo o seguinte trecho de programa: MOV SI, 1000 MOV DI, 2000 MOV CX,100 MOVER: MOV AL,[SI] INC SI MOV [DI],AL INC DI DEC CX JNZ MOVER As instruções são alimentadas nas duas pipelines de forma alternada, sendo uma instrução para U e outra para V. A ordem de alimentação é mostrada abaixo.

U-Pipeline	V-Pipeline
MOV SI, 1000	MOV DI, 2000
MOV CX,1000	MOV AL,[SI]
INC SI	MOV [DI],AL
INC DI	DEC CX
JNZ MOVER	MOV AL,[SI]

Microarquitetura P6

Esta arquitetura foi usada a partir de 1995, com o lançamento do Pentium Pro, e sofreu algumas modificações posteriores, dando origem ao Pentium II, Celeron e Pentium III, bem como suas versões Xeon. Uma das principais características desta nova arquitetura é a introdução de um "velho" conceito já usado há alguns anos em computadores mais poderosos: a tecnologia RISC – Reduced Instruction Set Computer.

CISC x RISC

Os processadores Intel, até o Pentium inclusive, são considerados máquinas CISC (Complex Instruction Set Computer). Significa que seu conjunto de instruções é bastante complexo, ou seja, tem muitas instruções que são usadas com pouca freqüência. Um programador de linguagem Assembly usa com muita freqüência instruções MOV, CALL, RET, JMP, ADD e outras, mas raramente usa instruções como XLAT e todas as opções de shifts e rotates A instrução XLAT, por exemplo, poderia ser substituída pela seqüência ADD BX,AX / MOV AL,[BX]. Teoricamente o uso de uma única instrução teria uma execução mais rápida que se fossem usadas duas ou mais instruções, mas na prática não é o que ocorre. Um processador com um conjunto de instruções muito complexo perde muito tempo para decodificar uma instrução. Além disso, um maior número de circuitos internos seria necessário para a implementação de todas essas instruções. Muitos desses circuitos são pouco utilizados a acabam apenas ocupando espaço e consumindo energia. Maior número de transistores resulta em maior aquecimento, o que impõe uma limitação no clock máximo que o processador pode utilizar. Por outro lado, a arquitetura RISC (Reduced Instruction Set Computer – computador com conjunto de instruções reduzido) utiliza instruções mais simples, porém resulta em várias vantagens. Instruções simples podem ser executadas em um menor número de ciclos. Com menos instruções, a decodificação de instruções é mais rápida e os circuitos do decodificador passam a ocupar menos espaço. Com menos circuitos, torna-se menor o aquecimento, e clocks mais elevados podem ser usados. Todas essas vantagens compensam o fato de um processador RISC não ter instruções "poderosas" como XLAT, por exemplo. Colocando tudo na balança, um processador RISC consegue ser mais veloz que um CISC de mesma tecnologia de fabricação. Existe um caso clássico em um processador CISC usado em um computador de grande porte produzido pela Digital (VAX 8600) nos anos 80. Ele tinha uma instrução BOUND, usada para checar se um índice está dentro dos limites permitidos por um array (a instrução BOUND do 80826 faz um trabalho similar). Se fossem usadas instruções mais simples de comparação do próprio processador do VAX 8600, a execução seria mais rápida que com o uso da sua própria instrução BOUND. Isso dá uma idéia de como instruções complexas tendem a reduzir a eficiência do computador. A única vantagem aparente das instruções

complexas é a economia de memória. Entretanto isso já deixou de ser vantagem há muitos anos, devido à queda de preços das memórias.

Há muitos anos se fala em abandonar completamente o velho conjunto de instruções CISC da família x86 e adotar uma nova arquitetura, com um conjunto de instruções novo. Isto traria entretanto um grande problema, que é a incompatibilidade com os softwares já existentes, mas um dia será feita esta transição (é o que a Intel espera fazer com a chegada do processador Itanium). Enquanto isso, os fabricantes de processadores adotaram um novo método de construção dos seus chips. Utilizam internamente um núcleo RISC, mas externamente comportam-se como máquinas CISC. Esses processadores aceitam as instruções x86 e as convertem para instruções RISC no seu interior para que sejam executadas pelo seu núcleo. Este processo mostrou-se mais eficiente que tentar produzir novos processadores CISC. O Pentium MMX foi o último processador totalmente CISC. Todos os novos processadores utilizam um núcleo RISC e tradutores internos de CISC para RISC.

Micro-ops

O núcleo dos processadores de arquitetura P6 é RISC. Sua unidade de busca e decodificação converte as instruções CISC (x86) obtidas da memória em instruções RISC. A Intel prefere chamar essas instruções RISC de microops.

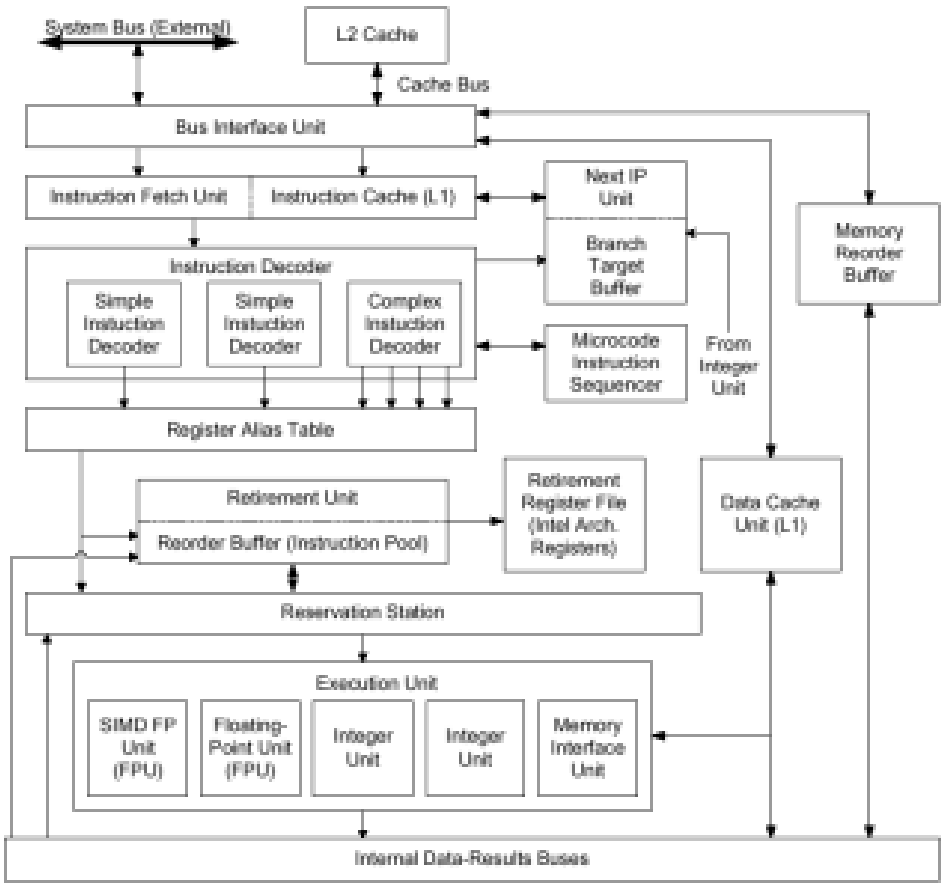


FIGURA 8.25

Diagrama interno de processadores P6.

Arquitetura do Pentium 4

O Pentium 4 e o Intel Xeon são os primeiros processadores a utilizarem a nova arquitetura Netburst da Intel. Apesar de ter muitos pontos em comum com a arquitetura P6, a Netburst é um projeto totalmente novo.

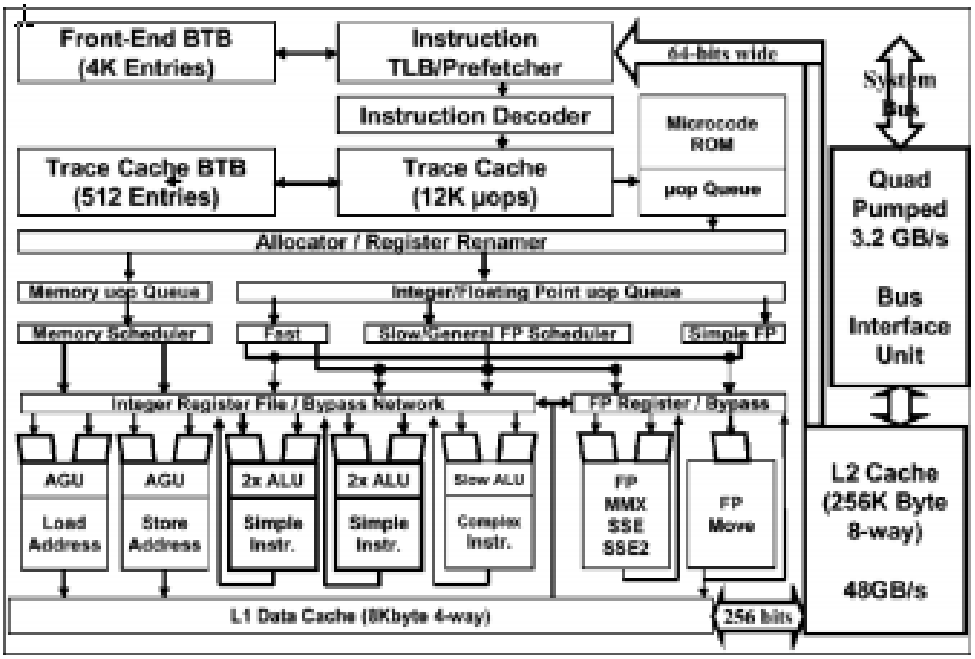


FIGURA 8.26
Diagrama interno do Pentium 4.

Um ponto notável desta arquitetura é a nova cache L1, chamada de Execution Trace Cache. As caches L1 de processadores anteriores armazenam instruções x86 (IA-32). As unidades de pré-busca e decodificação lidam com instruções provenientes da cache L1 e as introduzem nas pipelines ou no pool de instruções para que sejam executadas. Nos processadores de arquitetura P6, as instruções provenientes dos decodificadores são micro-ops (instruções RISC). Nas máquinas Netburst como o Pentium 4, as unidades de pré-busca e decodificação obtém as instruções diretamente da cache L2, e não da cache L1. Instruções já decodificadas e convertidas em micro-ops são transferidas para a cache L1, de onde as unidades de execução obtém as micro-ops a serem executadas. A vantagem de operar desta forma é que nos programas sempre temos instruções que são executadas repetidas vezes. Não é necessário decodificar novamente instruções que foram executadas há pouco tempo, pois sua forma já convertida em micro-ops ainda estará na cache L1. Portanto o trabalho de decodificação é feito uma só vez, e é aproveitado novamente quando uma mesma instrução é executada outras vezes.

Máquinas Netburst têm um pool de instruções capaz de manter 126 instruções em andamento. Nas máquinas P6 eram apenas 40 instruções. Isto permite executar um número maior de instruções antecipadas, ou seja, fora de ordem. A unidade de execução pode executar até 6 instruções por ciclo de clock, resultando em alto grau de paralelismo.

Máquinas Netburst são classificadas como hyperpipelined. Operando com pipeline de 20 estágios (máquinas P6 operavam com 10 estágios). Usar estágios menores significa que cada um dos estágios pode ser mais simples, com um menor número de portas lógicas. Com menos portas lógicas ligadas em série, é menor o retardo de propagação entre essas portas, e desta forma o ciclo de operação pode ser menor, ou seja, a freqüência de operação pode ser maior. Máquinas Netburst podem, portanto, operar

com maiores freqüências de operação que as máquinas P6 de mesma tecnologia. Usando a mesma tecnologia de produção do Pentium III (0,18μ), o Pentium 4 é capaz de atingir clocks duas vezes maiores.
O termo **IA-32** é mais novo, e significa Arquitetura Intel de 32 bits. É derivada do processador 80386. Todos os processadores usados nos PCs modernos são derivados desta arquitetura. Em outras palavras, podemos considerar o Pentium 4 e o Athlon como versões super velozes do 386, acrescidos de alguns recursos, porém são todos baseados em conjuntos de instruções compatíveis com o do 386. Os termos x86 e IA-32 são usados como sinônimos, sendo que a AMD prefere usar o termo x86, enquanto a Intel prefere usar IA-32. Tanto a Intel como a AMD estão entrando na era dos 64 bits, cada uma com sua própria arquitetura: Intel: IA-64 AMD: AMD x86-64 Essas duas arquiteturas têm características distintas. Ambas são de 64 bits, ou seja, utilizam registradores, valores e endereços de 64 bits, apesar de poderem também manipular valores de 32, 16 e 8 bits.

Intel IA-64

A arquitetura IA-64 é incompatível com a IA-32. Isto significa que os programas que usamos nos PCs atuais não funcionarão nos PCs baseados na arquitetura IA-64. Para facilitar a transição entre as arquiteturas IA-32 e IA- 64, o processador Intel Itanium (o primeiro a ser produzido com a IA-64) utiliza um tradutor interno de instruções IA-32 para IA-64. Desta forma poderá utilizar os programas atuais, porém com desempenho reduzido.

Processador Intel Itanium

A figura 27 mostra um processador Intel Itanium, o primeiro baseado na arquitetura IA-64. No início do seu desenvolvimento era conhecido pelo seu nome-código, Merced. É produzido na forma de um cartucho chamado PAC418, o mesmo nome do soquete usado nas placas de CPU para este processador. No interior do cartucho encontramos o processador propriamente dito e os chips SRAM que formam a cache L3. As primeiras versões do Itanium têm 2 MB ou 4 MB de cache L3. As caches L1 e L2 são integradas ao núcleo do processador, e as placas de CPU ainda poderão usar uma cache L4 opcional. O Itanium possui 15 unidades de execução e 256 registradores internos de 64 bits, sendo 128 para números inteiros e 128 para números de ponto flutuante.

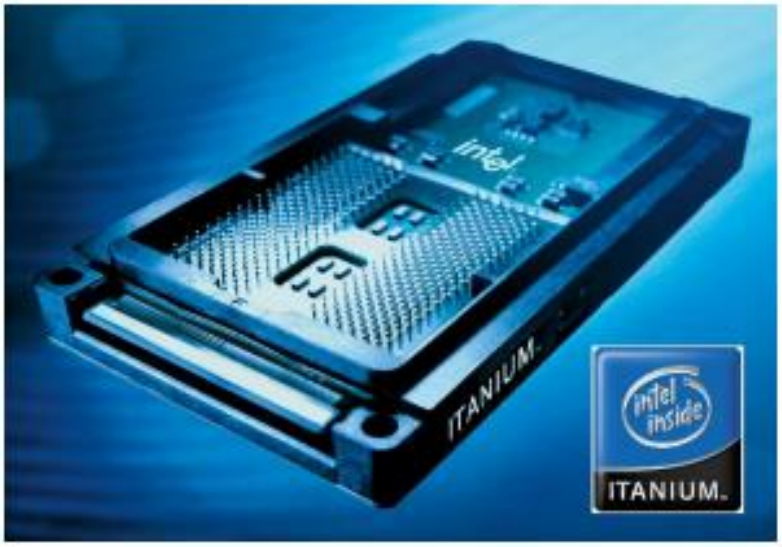


FIGURA 8.27
Processador Intel Itanium.

Barramentos de dados e endereços

O barramento de dados do Itanium opera com 133 MHz e DDR, produzindo o mesmo resultado que o de um clock de 266 MHz. Opera com 64 bits, portanto apresenta uma taxa de transferência máxima teórica de 2133 MB/s. Até 4 processadores podem ser ligados em conjunto na mesma placa. Seu barramento de endereços tem 44 bits, permitindo acessar diretamente uma memória física de até 16 Terabytes (17.592.186.044.416 bytes). Ao acessar memória virtual, opera com endereços de 54 bits, podendo endereçar até 16 Petabytes (18.014.398.509.481.984 bytes).

Futuros processadores Intel de 64 bits

Ainda são bastante escassas as informações sobre os processadores IA-64 posteriores ao Itanium. A própria Intel não divulga publicamente tais informações, exceto em conferências. Estão previstos novos processadores que por enquanto têm nomes códigos de McKinley, Madison e Deerfield. O McKinley (possivelmente será chamado de Itanium II) deverá usar a tecnologia de 0,13µ e sua cache L3 será integrada ao núcleo. Irá operar com clocks superiores a 1000 MHz. O Madison terá uma cache L3 maior (possivelmente 8 MB) e um barramento externo mais veloz. O Deerfield deverá ser uma versão de baixo custo do Madison, com cache L3 de apenas 1 MB, voltado para o mercado de PCs de baixo custo. Será uma espécie de "Celeron de Itanium".

MEMÓRIAS NÃO VOLÁTEIS

Introdução: As memórias não voláteis é uma classe de memórias que preservam o conteúdo mesmo quando a energia no dispositivo é desligada. As memórias não voláteis de uma grande aplicação como: tabelas de consultas, circuitos decodificadores, gerador de caractere e outros. As memórias não voláteis podem ser classificadas em:

- Memória apenas de leitura ROM;
- Memórias apenas de leitura com máscara MROM;
- Memória programável apenas de leitura PROM;
- Memória programável e por luz ultravioleta apagável de apenas de leitura EPROM;
- Memória programável e eletricamente apagável de apenas de leitura EEPROM;
- Memória programável e eletricamente apagável de apenas de leitura Flash;
- Memória programável e eletricamente apagável de apenas de leitura PEN;
- Outras.

Tecnologia e Estrutura interna: Para entender o dispositivo de memória não volátil, bem como a sua evolução para outros dispositivos mais modernos, inicia-se pela estrutura interna da memória ROM, assim como foi feita no capítulo referente às memórias voláteis. O dispositivo primitivo que permitiu através a evolução para os outros dispositivos foi a ROM considerada como um dispositivo de lógica programável da lógica combinacional, pois poderia ser implementada com portas lógicas organizadas matricialmente com duplo encadeamento. Vamos supor uma estrutura combinacional de memória ROM de 2 x 2bits, cujo conteúdo é igual a: $a_0b_0 = 1$, $a_0b_1 = 0$, $a_1b_0 = 1$ e $a_1b_1 = 0$, onde a_0b_0 será endereço zero, a_0b_1 o endereço um, a_1b_0 endereço dois e a_1b_1 endereço três.

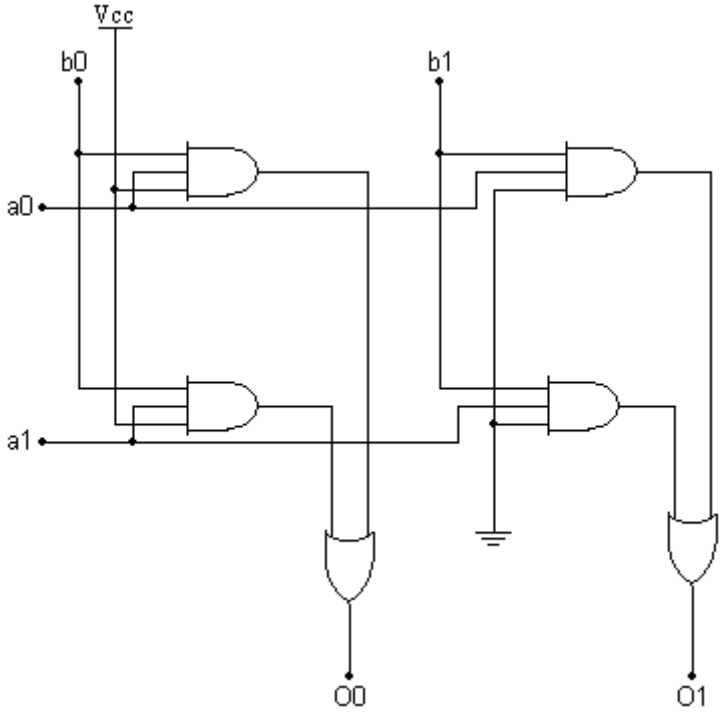


Figura: ROM – combinacional de 2 x 2bits.

A memória ROM foi utilizada em inúmeras aplicações e uma vez que vinham programada de fábrica, somente aplicações em grande quantidade era possível o seu uso. A necessidade do mercado para usar o dispositivo em projetos os quais precisavam de quantidades pequenas acelerou os fabricantes de dispositivos a criar uma nova memória e justamente para atender aquele usuário que tivesse um uso

específico e único no seu projeto. A memória poderia ser programada pelo usuário de acordo com a sua necessidade PROM (programável memória apenas de leitura), mas não permitia o seu apagamento. A dificuldade a qual limitou o seu uso foi com relação à atualização de circuitos, pois a memória não aceitava uma reprogramação. A seguir é mostrada a estrutura interna de uma PROM de 2 x 4bits.

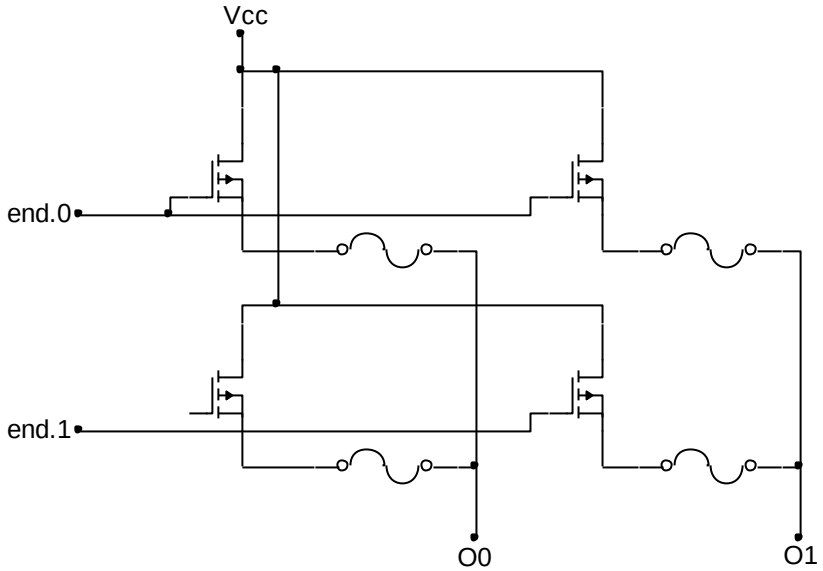


Figura: PROM de 2 x 2bits.

A memória PROM não pode ser apagada, a nova necessidade era uma memória PROM que pudesse ser apagada. A tecnologia agora seria totalmente diferente da tecnologia empregada nas memórias ROM e PROM para a nova memória EPROM (programável e apagável memória apenas de leitura). O novo recurso de apagamento seria efetuado pela aplicação de uma luz utravioleta, com determinada intensidade e por um período de tempo. A memória é totalmente apagada e não pode ser apagada setorialmente. Os dispositivos EPROM utilizam um transistor especial de porta flutuante, a qual em síntese aprisiona os elétrons na porta. Um transistor o qual contém elétrons aprisionados está no estado de não condução e está programado com o bit zero e no caso contrário está programado com o bit um. Quando uma memória é virgem ela não contém elétrons aprisionados e dessa forma o seu conteúdo é todas as células ou bits em um. A programação de um zero é feita aplicando-se uma tensão mais alta, dá ordem de 21 Volts aos transistores localizados nas células onde o usuário deseja colocar zeros. A alta tensão faz com que os elétrons abram um caminho ou túnel (Fowler-Nordheim tunneling) entre uma parte isolante até a porta flutuante. Quando a tensão é retirada, não são possíveis os elétrons a criar um túnel de volta e daí eles ficam aprisionados na porta flutuante. O caminho de volta dos elétrons é conseguido com a aplicação de uma luz ultravioleta aplicada na janela localizada no topo do dispositivo, conforme mostrado a seguir.

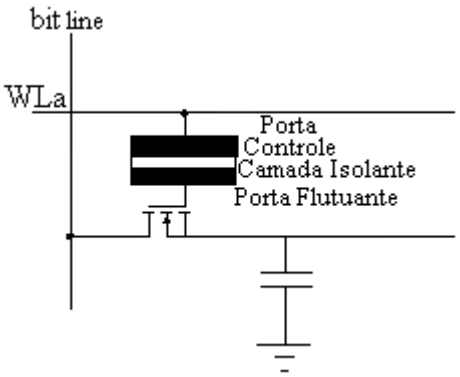


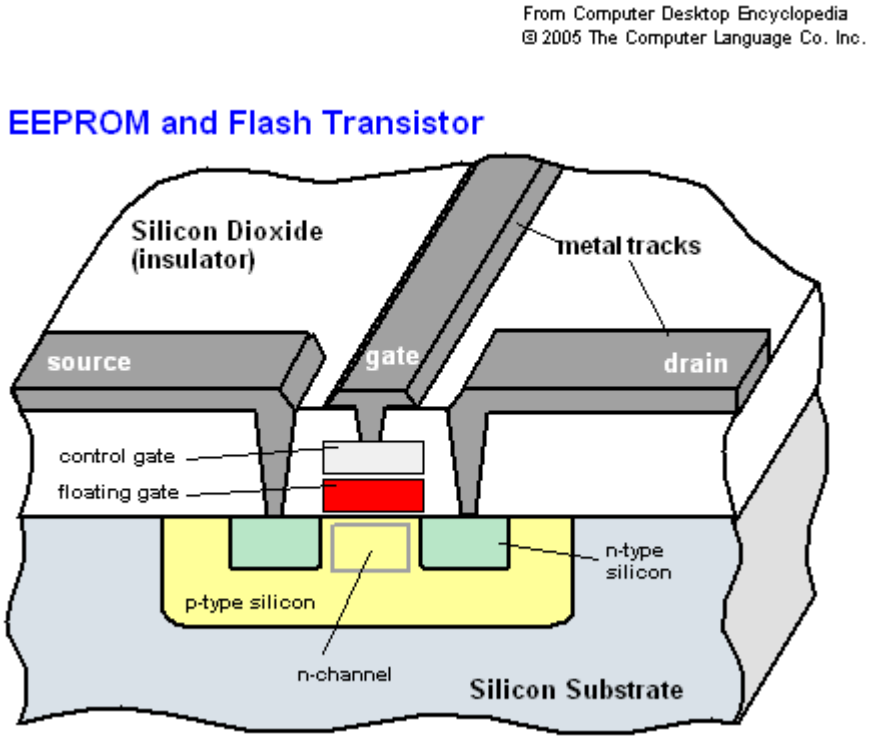
Figura: Célula de transistor de porta flutuante.

A luz ultravioleta faz com que os elétrons aprisionados sejam descarregados para a terra e a memória retorna com todas as células em nível lógico um, enfim o apagamento foi total. A figura a seguir mostra uma célula a transistor porta flutuante.

Uma vantagem do uso da EPROM (apagável e programável memória apenas de leitura) era apagar e atualizar versões de circuitos, simplesmente retirando a memória do circuito e aplicando uma nova memória. Diferente da PROM que a atualização por uma nova memória inutilizava definitivamente o dispositivo. Porém uma grande dificuldade de retirar a memória do circuito para atualização obrigava a colocação de soquetes apropriados para o acesso rápido. Se essa operação fosse feita inúmeras vezes ocorria certamente problemas mau-contato e a solução final era ou trocar o soquete do circuito ou soldar definitivamente a memória. Nenhuma das duas soluções foi bem aceita pelo usuário que pedia a reprogramação ou atualização da memória fosse feita dentro do circuito ou o circuito de programação e de apagamento fizesse parte do circuito de acesso e operação da memória.

Figura: Um chip de EPROM.

Nasce dessa idéia a memória EEPROM (eletricamente apagável e programável memória apenas de leitura). A tecnologia da memória novamente é um arranjo de transistores de portas flutuantes e a memória permite ser apagada completamente pelo circuito de apagamento no próprio circuito e com tempo de gravação dos dados razoável. A estrutura a seguir mostra a memória EEPROM.



A grande aplicação das EEPROM foi a sua utilização como memória temporária para equipamentos que tinham diversas ordens de operações e informações que mudavam muito pouco, pois o processo de gravação e apagamento dos dados é limitado por palavra de memória um processo muito lento embora

as memórias EEPROM modernas possam ter um processo multibytes. Essa operação trouxe para o usuário alguns transtornos que devem ser superados na próxima memória a ser criada. Daí nasceu a memória chamada Flash. Com características de uma EEPROM a memória Flash teve que superar vários obstáculos como preço e tempo de acesso (velocidade de gravação e apagamento mais rápido dos dados).

APLICAÇÕES: A aplicação da memória apenas de leitura está na geração booleana, tabela de dados e outras. Para dar exemplos da potencialidade das memórias não voláteis resolveremos alguns exercícios.

Exercício: Gerar as funções F_1, F_2, F_3 e F_4, de acordo com as expressões booleanas a seguir.

$F_1 = AC + AB + BC.$

$F_2 = A'C + AB' + BC'.$

$F_3 = ABC + A'B'C + AC'.$

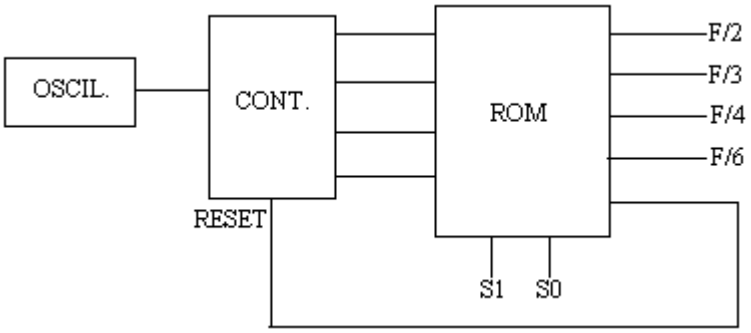
$F_4 = A'C' + A'B' + B'C'.$

a) Tabela da verdade
b) Mapa de endereço e conteúdo da memória ROM.

Exercício: Gerar y = 3x + 7, sendo x um número em BCD-8421 e y um número binário. Pede-se:
a) A tabela da verdade.
b) O mapa de endereço e conteúdo da ROM.

Exercício: Construir um conversor BCD-8421 para BCD-5211. Pede-se:
a) A tabela da verdade.
b) O mapa de endereço e conteúdo da ROM.

Exercício: Construir um gerador de formas de ondas f_1 = f/2, f/3, f/4. f/6, de acordo com as variáveis de seleções S_1S_0, onde 00 – f/2, 01 – f/3, 10 –f/4 e 11 – f/6. Pede-se:
a) Tabela da verdade (mapa da ROM), endereço e conteúdo.



Exercício: Construir um gerador de forma de onda senoidal v(t) = 3 sen(2Π1.000t). Gerar 12 pontos com ROM e um DAC de 5 bits. Pede-se:
a) O mapa de endereços e conteúdo da rom.
b) A forma de onda na saída do amplificador.

MEMÓRIAS VOLÁTEIS

Introdução: O mercado consome um volume muito grande de memória não volátil. As memórias são empregadas em todo o tipo de equipamento de áudio, telefonia, televisão em computação nos lap-tops, computadores de mesa enfim é um dispositivo imprescindível em qualquer arquitetura de sistemas desde a mais simples até a mais complexa e vai depender somente da quantidade de bits que podem ser armazenados e da sua velocidade de acesso aos dados. Nesse capítulo serão descritos os tipos de memórias voláteis e estão incluídas as memórias estáticas SRAM dos tipos (regular, DDR e QDR). Também serão descritas as memórias dinâmicas DRAM, as memórias síncronas SDRAM dos tipos (regular, DDR, DDR2 e DDR3) e a memória do tipo conteúdo endereçável CAM.

TERMINOLOGIA

Para que possamos descrever sobre as memórias, os seus tipos, suas aplicações e associações uma terminologia se faz necessária antes de promover o estudo delas.

Bit – É a menor quantidade de informação. Pode representar a informação com ‘0’ ou ‘1’.

Byte – É um cordão com oito bits. Pode representar uma instrução, ou um dado, ou um número.

Palavra – É um grupo de bits que são processados juntos pelo sistema. O tamanho da palavra ou comprimento da palavra depende da característica do processador e pode variar de 8 a 64bits nos computadores modernos.

Dataword – É o tamanho em bits do dispositivo ou o a largura máxima de bits que podem ser alocados no dispositivo.

Capacidade – É a quantidade total de bits que podem ser armazenados na memória. Pode ser também a quantidade de bytes.

Densidade – É a medida da quantidade de transistores que são inseridos no mesmo espaço no chip. Aumentar o número de transistores no chip é torná-lo mais denso.

Célula de armazenamento – É a célula que tem a capacidade de reter um bit de informação. Pode ser um capacitor ou um flip-flop dependendo do tipo de memória. Em ambos os casos é necessária manter ativa uma fonte de energia para a célula operar.

Memória – É um dispositivo que retém os dados internamente ou em células de armazenamento ou em capacitores ou em dispositivos de porta flutuante.

Endereço – É a localização exata de uma célula de armazenamento. É representada em binário por um barramento de endereços.

Conteúdo – É a informação retida na memória. Pode ter de um a oito bits de acordo com o arranjo da memória e pode ser lido ou escrito da memória.

Volátil – É a característica dos dispositivos que necessitam constantemente da energia vinda da fonte de alimentação para a retenção dos dados.

Não Volátil – É a característica dos dispositivos que não necessitam da energia vinda da fonte de alimentação para a retenção dos dados.

RAM – É o termo usado para os dispositivos de memória cujo acesso pode ser aleatório. Vem do inglês (random acess memory) e podem ser de vários tipos. Normalmente é caracterizada pela sua capacidade de armazenamento e é do tipo volátil.

ROM – É o termo usado para os dispositivos de memória cujo acesso é apenas de leitura dos dados retidos. Vem do inglês (read only memory) e podem ser de vários tipos. Também como as memórias RAM é caracterizada pela capacidade de dados armazenados e é do tipo não volátil.

Tempo de acesso – É o tempo requerido pelo dispositivo para localizar o conteúdo específico de um endereço de memória. É um parâmetro muito importante para a memória, pois define a velocidade do dispositivo e a sua compatibilidade em tempo com outros dispositivos.

Ciclo de leitura – É o processo necessário para a realização da leitura do conteúdo da memória. Nesse processo participa sinais do hardware, como sinal de leitura (read), de habilitação (chip enable) combinada com as linhas de endereço e as linhas de dados.

Ciclo de escrita – É o processo necessário para a realização da escrita do conteúdo da memória. Nesse processo participa sinais do hardware, como sinal de escrita (write), de habilitação (chip enable) combinada com as linhas de endereços e as linhas de dados.

Ciclo de Refrescamento – É o processo precisa refrescar os dados periodicamente na memória sob o risco de perda deles. É utilizado somente na memória dinâmica, a qual retém o conteúdo em capacitor.

Buffer – É um dispositivo de memória temporária utilizada para reter os dados enquanto estão sendo transferidos de uma unidade para outra, principalmente quando existe uma diferença entre a relação entre os dados recebidos e processados.

BUFFER PRÉ-BUSCA – É um buffer de dados empregados nos dispositivos de memórias modernas do tipo DRAM os quais armazenam temporariamente os dados localizados na linha de endereços físicos da memória.

CACHE – É um tipo de buffer que armazena instruções e dados para o processamento futuro a fim de processar mais rapidamente;

Memória de massa – É um tipo de memória que tem a capacidade de armazenar um volume muito grande de informação. São as memórias conhecidas como de disco magnético, fitas magnéticas e outros.

MEMÓRIA FLASH – É um tipo de memória E^2PROM não volátil cujo acesso aos dados é feito em alta velocidade. É um dispositivo que pode ser utilizado em milhares de operações e tem grande capacidade de armazenagem de dados.

PEN DRIVE – É um tipo de memória não volátil de acesso rápido e de grande densidade de informação. É um tipo de memória conhecida como E^2PROM do tipo flash.

Baud-rate – É a taxa de transmissão/ recepção de dados. É expressão em bit/s e pode ser de de 1200, 2400, 4800, 9600 e 38400 bit/s.

Bandwidth – É a largura de faixa de um barramento de dados. É expressa em freqüência (MHz) e define a capacidade do barramento de se comunicar em velocidade com a unidade central de processamento.

SIMM – Módulo de memória de via única (Single in-line memory module) montado sobre um conector de 72 conexões.

DIMM – Módulo de memória de via dupla (Dual in-line memory module) montado sobre um conector de inúmeros pinos alguns com 168 pinos e 240 pinos.

SDRAM – É uma memória síncrona de alto desempenho da família DRAM.

DDR – É uma memória síncrona do tipo SDRAM com dupla taxa de dados

DDR2 – É uma memória síncrona do tipo SDRAM com dupla taxa de dados para a memória e outra para o barramento de dados.

DDR3 - É uma memória síncrona do tipo SDRAM com dupla taxa de dados para a memória e outra para o barramento de dados e com busca antecipada de 8 bits e freqüência do clock de entrada e saída é quatro vezes a freqüência do clock da memória.

ORGANIZAÇÃO DAS MEMÓRIAS

A memória denominada de RAM (random acess memory) ou memória de acesso aleatório pode ser do tipo volátil e não volátil. De acordo com a sua capacidade de armazenamento de dados as memórias podem ser organizadas em sua estrutura interna.

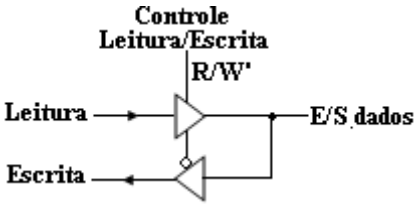
Organização Interna – As memórias podem ser organizadas de forma matricial como veremos adiante mas para o aluno entender como elas foram estruturadas internamente começaremos definindo uma célula de armazenamento, que conforme a terminologia pode ser um capacitor ou um flip-flop.

Entrada/Saída de dados – As memórias possuem pinos bidirecionais destinados para a entrada e saída de dados. O sentido dos dados será controlado pelo sinal de leitura e escrita (R/W') o qual atua sobre na saída de cada um dos buffers de entrada e saída habilitando somente um deles por operação. O buffer não ativo tem a sua saída colocada no terceiro estado. A tabela da verdade a seguir mostra a operação do controle e figura a seguir mostra como é feito essa separação das linhas de dados de entrada e saída através dos buffers terceiro estado.

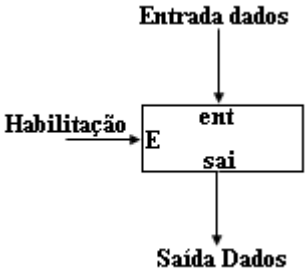
Tabela da verdade

R/W'	Operação
0	Escrita
1	Leitura

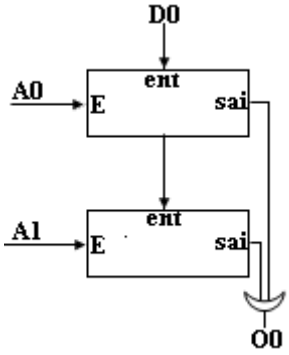
Circuito Entrada/Saída



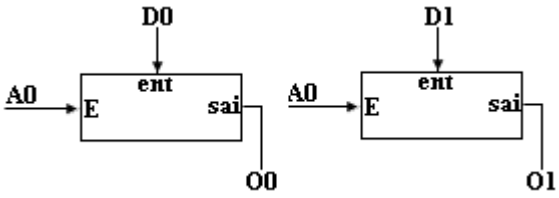
1. Célula de 1 x 1 bit



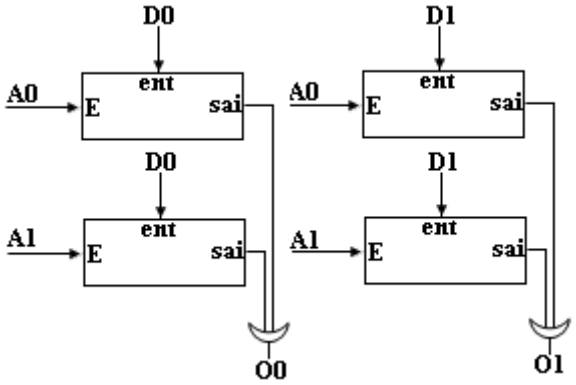
2. Células de 2 x 1 bits.



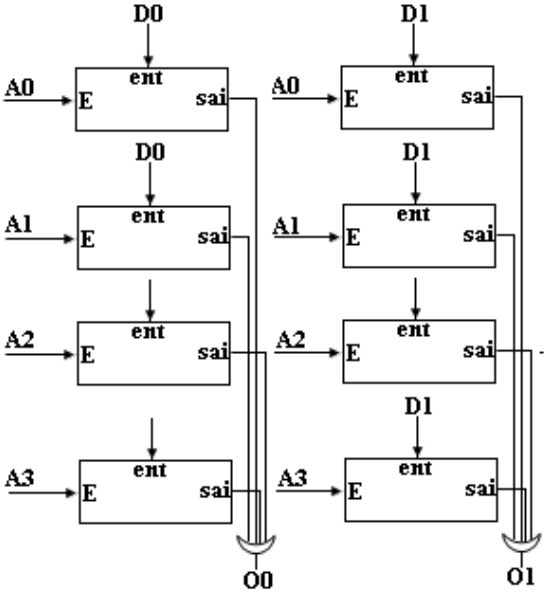
3. Células de 1 x 2 bits.



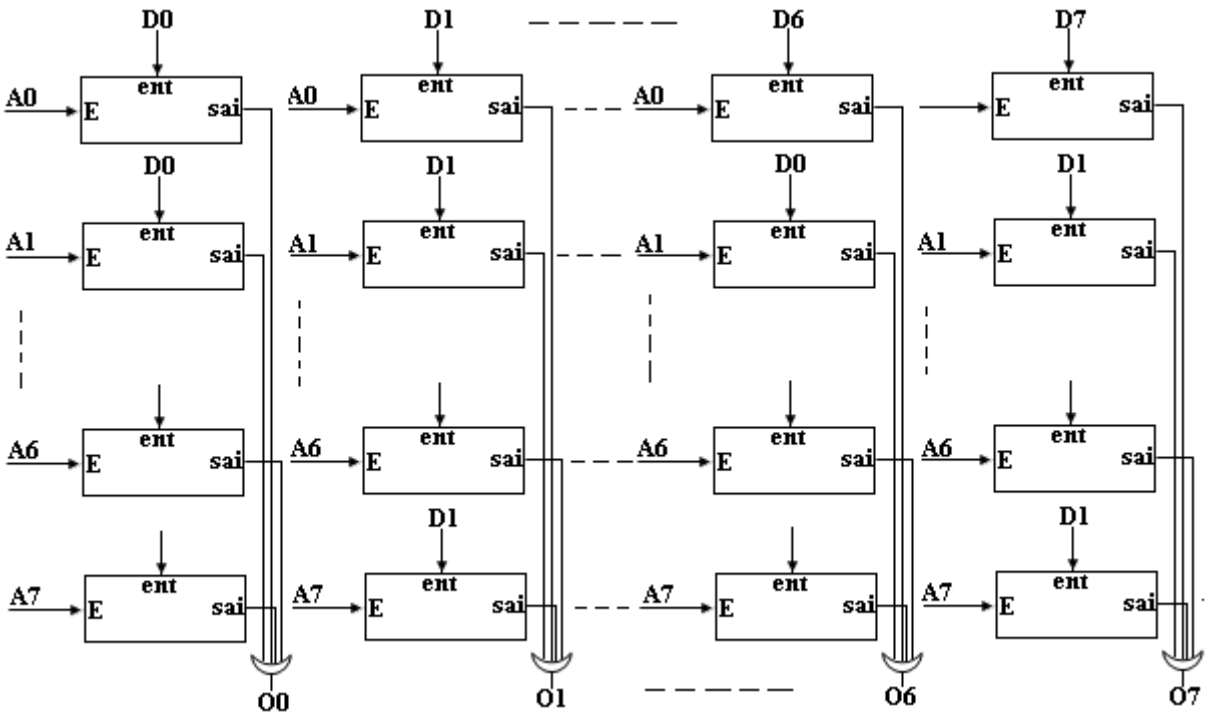
4. Células de 2 x 4 bits



5. Células de 4 x 2 bits



6. Células de 8 x 8 bits.



As linhas de endereçamento como na memória de 8 x 8, irão crescer com aumento da capacidade da memória e por exemplo uma memória de 1K x 8 já não pode usar o mesmo sistema de acesso que a memória 8 x 8, pois necessitaria de 1024 linhas de acesso (endereçamento linear). Afim de reduzir o número de linhas do endereçamento a solução inicial foi gerar as linhas de acesso através de um dispositivo lógico capaz de decodificar as linhas codificadas em binário na entrada. A codificação reduz o número de linhas de acesso igual a 2^n, onde n é igual ao número de linhas de entrada. A figura a seguir mostra o decodificador de dez linhas de endereços de A0 a A9, com uma entrada de controle CS' que permite ou não o acesso à memória.

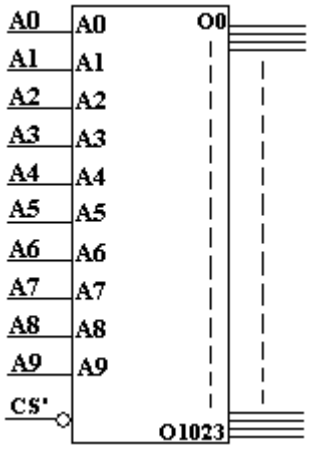


Figura: Decodificador 1K linhas

O esquema apresentado permite o acesso a memórias de pequena capacidade de armazenamento, pois criar decodificadores maiores é um pouco mais complexo e mais caro. O esquema utilizado para

memórias acima dessa capacidade, um esquema matricial funciona melhor mais complexo na estruturação. O exemplo a seguir mostra um endereçamento de um mega igual a 2^{20}, com vinte linhas de endereços.

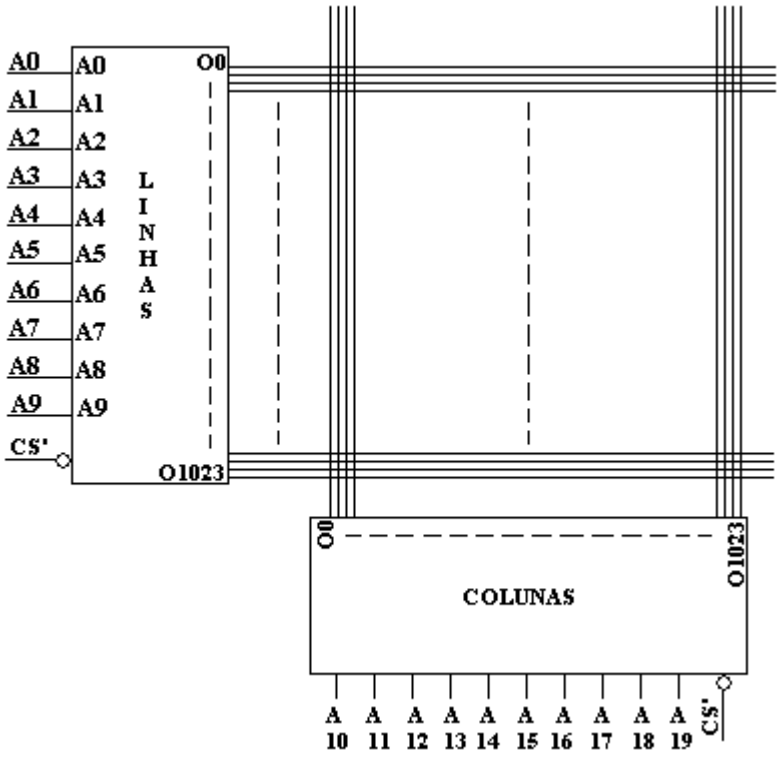


Figura: Decodificador de 1M de endereçamento esquema matricial.

A alternativa de endereçar uma memória com um arranjo matricial e quadrada, isto é, o número de linhas do decodificador igual ao número de colunas evita problemas de atrasos quando o caso não for uma matriz quadrada. Uma terminologia é utilizada para as linhas de endereços que selecionam as linhas da matriz serão doravante chamadas de *word line* enquanto as linhas de endereços que selecionam as colunas da matriz serão doravante chamadas de *bit line*.
Por exemplo, é normal reduzir a excursão da tensão sobre as linhas *bit lines* para uma tensão muito menor do que a tensão de fonte de alimentação V_{DD}. A conseqüência é a redução do tempo de propagação e o consumo de energia. O cuidado que se deve ter é com relação à margem de ruído com ruído do tipo "cross-talk" uma interferência causada pela indução em linhas próximas às linhas de sinais, além de outras perturbações. Para interfacear com o mundo externo requer uma amplificação do sinal de excursão interna pelo amplificador chamado de *amplificador sense*. O funcionamento do amplificador sense é discutido na seção amplificador sensor deste capítulo. Quando se não se estabelece limites, a célula de memória pode ser reduzida de 1 a 6 transistores, como a célula 6T que veremos adiante.

TAMANHO DA MEMÓRIA

A arquitetura acima funciona muito bem quando a capacidade da memória vai até 256K, mas para arquiteturas maiores as memórias sofrem um grande problema de degradação na velocidade como o comprimento, capacitância e resistência da *word e bit line* os quais tornam excessivamente grande. A figura a seguir apresenta um particionamento em blocos pequenos para o acesso em grandes memórias. A memória então é particionada em pequenos blocos P e idênticos. Uma palavra é selecionada pelos endereços das linhas e colunas de endereçamentos e são comuns a todos os blocos. Para o endereço do

bloc P a arquitetura usa um bloco de endereço extra e uma palavra de endereço é requerida para a seleção do bloco P para ler ou escrever. A abordagem tem duas vantagens a saber:

- Os comprimentos das linhas *Word e bit lines* são restritos aos blocos e são mantidos dentro de limites resultando acessos mais rápidos;
- O endereço do bloco permite o acesso ao bloco selecionado e os outros blocos permanecem não ativos resultando em economia de energia, pois os decodificadores dos sensores e das linhas e colunas estão desligados.

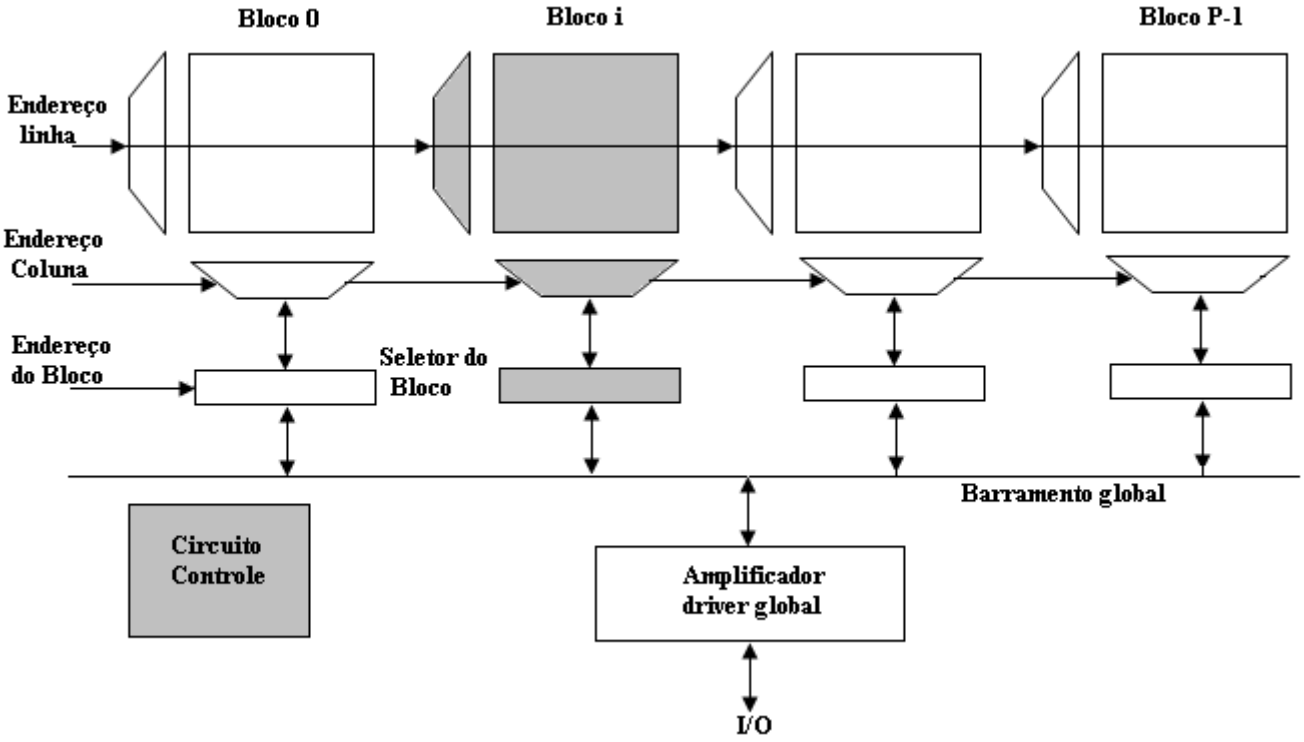


Figura: Arquitetura de memória com bloco de partição no tempo.

NÚCLEO DE MEMÓRIA

Nesta seção o foco é o projeto do núcleo da memória e sua célula de composição usando a tecnologia CMOS para o tipo de memória. A maior preocupação dos projetistas é quanto ao tamanho da célula de armazenagem tão pequena quanto possível. Esta diminuição na célula não deve afetar outras características não menos importantes como velocidade e realizabilidade. Na seção SRAM circuito da SRAM é apresentada a célula seis-T.

TIPOS DE MEMÓRIAS

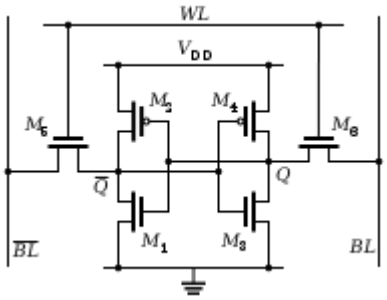
Como falamos anteriormente, as memórias podem ser classificadas quanto aos tipos voláteis e não voláteis, podem ser síncronas ou assíncronas, estáticas ou dinâmicas e ainda podem ser do tipo conteúdo endereçável conhecida como memória associativa. Dentro da classificação de volátil estão as memórias a seguir:

- SRAM (RAM estática);
- SRAM DDR (dupla taxa de dados) e QDR (quádrupla taxa de dados);
- DRAM (RAM dinâmica);
- SDRAM (RAM síncrona);
- SDRAM DDR/DDR2/DDR3 (dupla taxa de dados);
- CAM (memória de conteúdo endereçável).

SRAM (Static random acess memory)

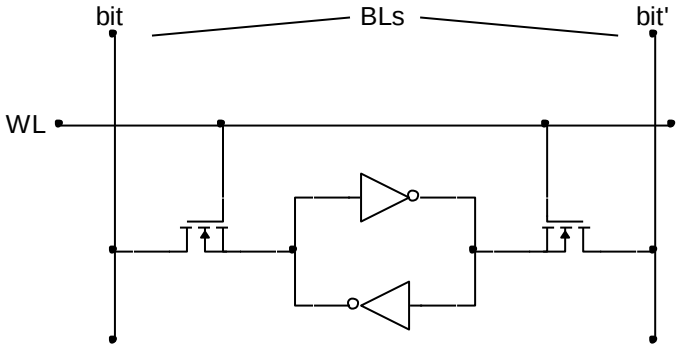
A SRAM (memória de acesso aleatória e estática) é a memória mais tradicional e a primeira a ser implementada. Possui uma célula de armazenamento composta de circuitos que retêm informações do tipo flip-flops e não necessita de qualquer mecanismo de retenção de dados. É volátil, pois de pende da fonte de energia para a célula guardar o dado e a sua construção pode ser mostrada a seguir. É usada na construção de memórias cachê do computador, em virtude da sua alta velocidade.

Circuito da SRAM



Uma célula SRAM com seis-transistores CMOS.

Representação em bloco

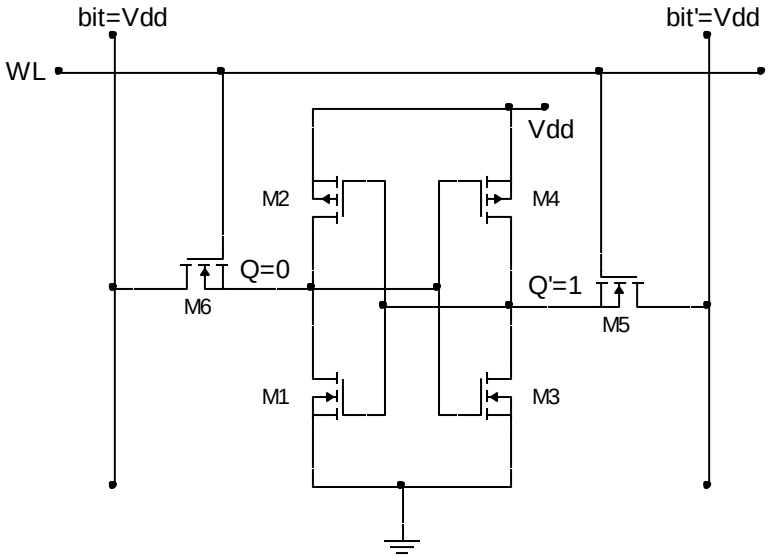


Representação da célula SRAM – 6T

Cada bit de uma célula SRAM é armazenado nos quatro transistores que forma o circuito biestável com dois inversores cruzadamente acoplados. Esse biestável como célula de armazenamento possui dois estados estáveis os quais são usados para operar em zero e um. Para o acesso ao biestável a célula de armazenagem possui dois transistores adicionais os quais permitem o acesso quando a operação é de leitura ou de escrita. Uma célula típica de armazenagem SRAM usa seis MOSFET para cada bit de memória para armazenagem. Existem células que usam menos do que seis transistores como: 3T[5][6] ou célula de 1T usada na memória DRAM.

OPERAÇÃO

A operação da célula 6T se resume em leitura e escrita. Para a leitura do bit armazenado na célula, uma maneira consiste em aplicar nas linhas BLs (bit line) a tensão de fonte V_{DD} e em seguida a esta carga de tensão deixar as linhas em flutuação, para em seguida ativar a linha WL (word line). Vamos para efeito de entendimento considerar que o conteúdo da SRAM seja Q = 1 e Q' = 0 (Saídas dos MOSFETs dos inversores). Para essa condição então os transistores da figura a seguir se encontram M1 no estado de condução e M3 no estado de corte e os transistores M2 no estado de corte e M4 no estado de condução. Quando WL é ativo, então os transistores M5 e M6 são ligados e as linhas BLs (bit e bit') pré-carregadas com V_{DD} são conectadas aos transistores M1 e M3. A linha BL ligada ao bit' mantém a tensão, pois o transistor M3 está cortado, mas a linha BL ligada ao bit será diminuída porque o transistor M1 está conduzindo, resultando linha bit = 0 e linha bit' = 1 na saída. Deve-se projetar a resistência de M6 maior do que dos transistores M1 para prevenir que a tensão aplicada ao transistor saturado não exceda a tensão de limiar do transistor M3 levando a mudança de estado. Isso quando ocorre é considerado malfuncionamento da célula chamada de *read upset*.



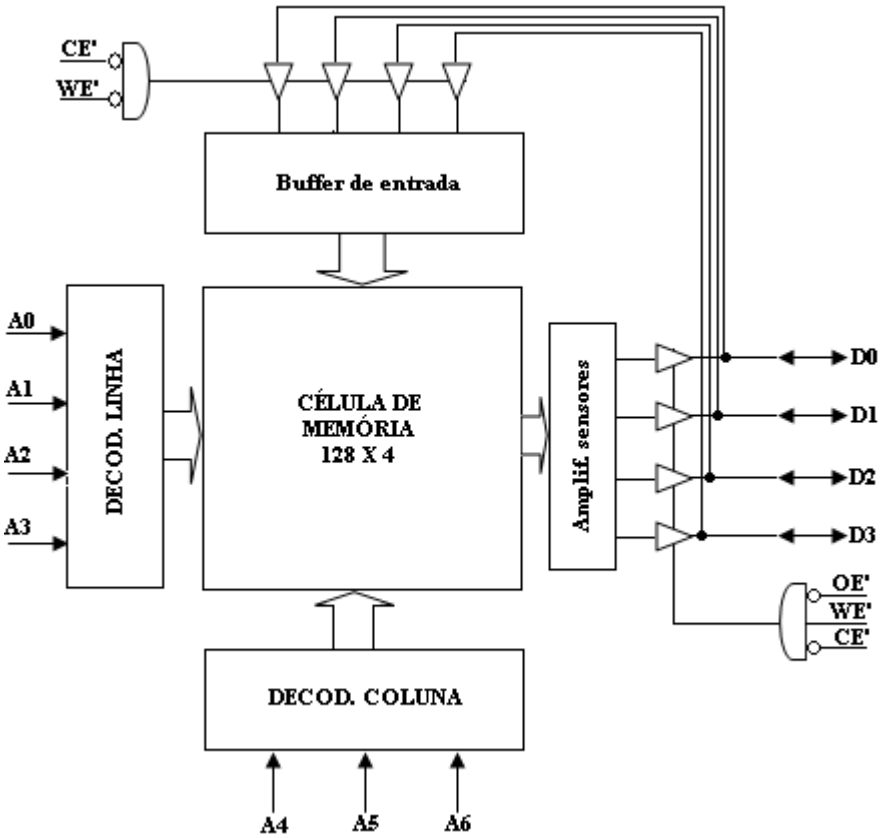
Operação de leitura na célula SRAM – 6T.

Teoricamente a célula funciona bem, mas uma preocupação na leitura da memória é a grande capacitância parasitária encontradas nas linhas BLs. Quando M6 entra no estado de condução e é ligado a linha BL é conectada diretamente à saída do transistor M1 que também está em condução e esta conexão é um nó intermediário com as portas dos transistores M3 e M4, nos quais recebem a linha BL bit igual a V_{DD} e momentaneamente tende a aumentar de tensão. A diferença de tensão então pode provocar a transição do inversor M3 e M4 e, portanto, inversão dos bits armazenados. Depende da resistência do canal dos transistores M1 e M6 e o aumento de tensão não pode ultrapassar a tensão de limiar (threshold) dos transistores M3 e M4.

Quando o ciclo de leitura inicia as linhas BLs ligadas aos inversores as quais são acionadas pelos níveis zero e um na célula SRAM. Essa condição melhora a operação da SRAM comparada com as DRAMs, a qual a linha BLs é ligada ao capacitor de armazenagem. Nesse caso há uma divisão na carga provocando uma excursão da tensão subida e descida. Essa simetria estrutural da SRAM permite um diferencial o qual faz com que pequenas excursões de tensões são facilmente detectáveis.
O tamanho da SRAM com *m* linhas de endereços e *n* linhas de dados é 2^m palavras, ou $2^m \times n$ bits.

ARQUITETURA DAS MEMÓRIAS SRAM

Uma arquitetura típica para um chip SRAM é mostrada a seguir com arranjo matricial de 128 x 8bits. A tabela da verdade mostra a operação da memória. A memória possui sete linhas de endereçamento de A_0 a A_6 com 4 bits para a linha de endereço e 3 bits para a coluna de endereços. O barramento de dados é de 4 bits.



WE'	CE'	OE'	Operação
0	0	x	Escrita
1	0	0	Leitura
x	1	x	Ociosa
x	x	1	Ociosa

Figura: Memória de 128 x 4bits tipo SRAM.

OPERAÇÃO SRAM

Uma célula SRAM tem três estados diferentes: standby onde o circuito é ocioso, leitura quando o dado é requisitado para leitura e escrita quando o conteúdo da SRAM é atualizado.

Standby

Quando não há cesso à célula SRAM, então os transistors M5 and M6 da célula 6T são desconectados das linhas BLs.

LEITURA

A operação de leitura da célula de armazenamento 6T tem o seguinte procedimento. Vamos considerar que a memória armazenou zero na memória e a saída Q = 0. O ciclo começa com a pré-carga de ambas as linhas BLS bit e bit' para a tensão de nível lógico um $V_{DD.}$ Então quando a linha WL é ativa, o acesso aos transistores é habilitado. O próximo passo ocorre quando os valores armazenados em Q e Q' são transferidos para as linhas BLs bit e bit' com o valor pré-carregado e descarregando BL através de M1 e M6 para a lógica zero. Do outro lado da linha BL, os transistores M4 e M5 mantêm a tensão em V_{DD}, o estado lógico um. Se o conteúdo da memória fosse invertido Q = 1, o contrário ocorreria e a linha bit iria para nível lógico um e a linha bit' iria para nível lógico zero. As linhas bit e bit' terão uma pequena diferença entre elas e a diferença aciona um amplificador, o qual sente quais das linhas têm mais alta tensão e assim identificará se foi armazenado um ou zero na memória. A alta sensibilidade do amplificador torna a operação de leitura da SRAM mais rápida.

Ciclo de Leitura

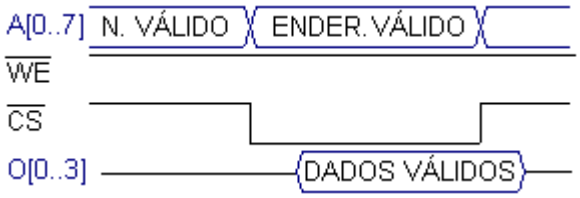


Figura: Ciclo de leitura.

ESCRITA

A operação de escrita da célula de armazenamento 6T tem o seguinte procedimento. Vamos considerar que foi aplicada à memória valor zero ou um nas linhas BLs. Se a operação de escrita é bit zero aplicada na linha bit = 0, isto é, colocando bit = 1 e bit' = 0. Este é similar a aplicação de um pulso de reset para um latch RS, O qual provoca a troca de estado do flip-flop para um. Um nível lógico um é escrito pela inversão dos valores das linhas BLs. Quando a linha WL é ativa e o valor é armazenado no latch. A única precaução são os tamanhos dos transistores na célula SRAM é necessário para garantir a operação.

Ciclo de Escrita

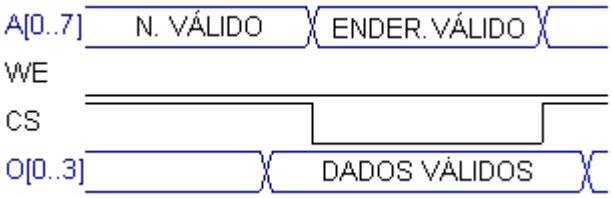


Figura: Ciclo de escrita.

COMPORTAMENTO DO BARRAMENTO

Uma memória RAM com um tempo de acesso de 70ns, os dados estarão válidos no barramento de dados dentro de 70ns após o tempo que as linhas de endereços são válidas. Os dados serão mantidos por um tempo de manutenção de (5-10ns). Tempos de subida e descida também influenciam em aproximadamente 5ns.

AMPLIFICADOR SENSOR

A finalidade do amplificador sensor é acelerar o acesso à memória SRAM e com isso um aumento na velocidade da memória SRAM. O amplificador sensor deve ser instalado entre as linhas BLs da célula de armazenagem 6T. É também inserido entre as BLs um circuito equalizador com um transistor pMOS. O equalizador instalado entre as linhas BLs tem a finalidade de equalizar a mesma tensão pré-carregada nas BLs quando a célula 6T está realizando uma operação de leitura. Nessa operação as linhas BLs são pré-carregadas com V_{DD} e assim conforme a figura a seguir, os transistores pMOS são ativos pelo sinal do equalizador para elevação e equalização das tensões nas BLs. Depois de ocorrer a pré-carga as linhas BLs são deixadas em flutuação e isso ocorre quando o sinal de equalização é retirado. O próximo passo é ativar a linha WL e a célula 6T em uma das linhas BLs a tensão diminuirá (Q ou Q' = 0).

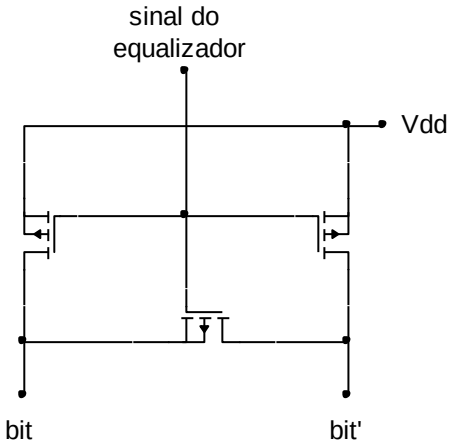


Figura: Equalizador da célula.

A diferença de potencial entre as linhas bit e bit' for igual a aproximadamente 0,5V, o sinal do sensor é ativo e daí os inversores biestáveis são acionados. O lado que tiver a tensão na linha BL mais alta conseqüentemente aciona a porta do inversor oposto cujo transistor é nMOS e a que tiver a tensão mais baixa aciona a porta do inversor oposto cujo transistor é pMOS. Dessa forma há uma rapidez em se atingir a tensão de nível lógico um e a tensão de nível lógico zero.

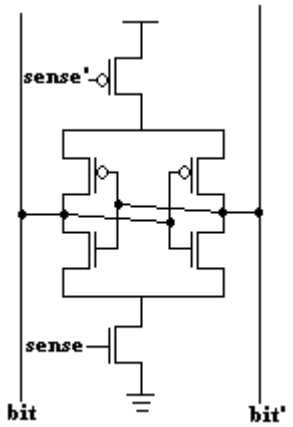


Figura: Leitura da célula.

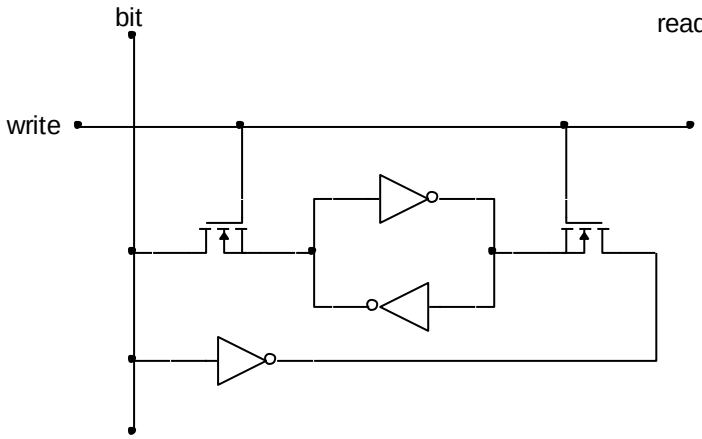
SRAMS TIPO DDR E QDR

As memórias convencionais até então eram assíncronas, diferente conceitualmente das memórias SRAM modernas que são síncronas, portanto todas as entradas e saídas são registradas e todas as operações são controladas diretamente pelo relógio (clock) do sistema. A operação da memória DDR (taxa de dados dupla), que consiste em processar os dados (isto é, ler ou escrever) em ambas as transições do clock.

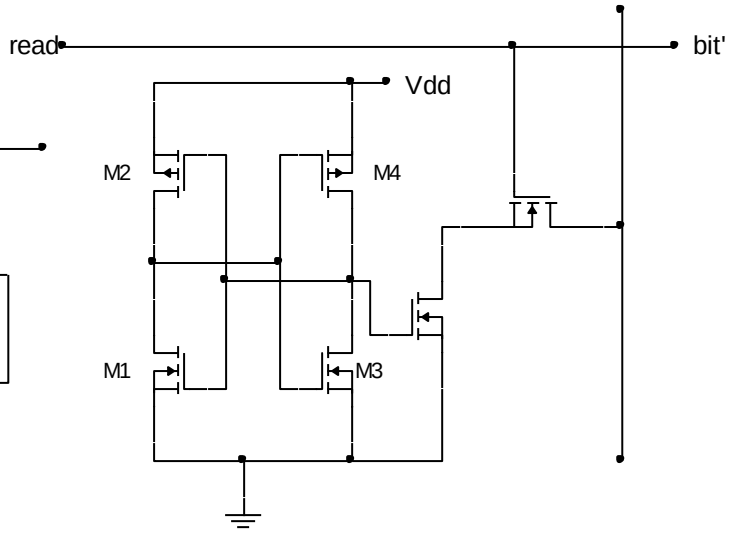
PRINCÍPIO DE OPERAÇÃO DAS MEMÓRIAS SRAMs TIPOS DDR E QDR

As memórias DDR (taxa de dados dupla) e QDR (taxa de dados quádrupla) ambas podem funcionar no modo DDR, com a individualização dos barramentos de dados, sendo um barramento para a entrada de dados (escrita dos dados) e o outro barramento para a saída dos dados (leitura dos dados). O funcionamento do barramento individualizado se baseia na introdução de células com duas portas como visto na célula 6T. As figuras a seguir mostram a célula 6T numa operação individual de escrita e individual de leitura. As duas operações podem ser reunidas em dois barramentos separados criando a célula de duas portas.

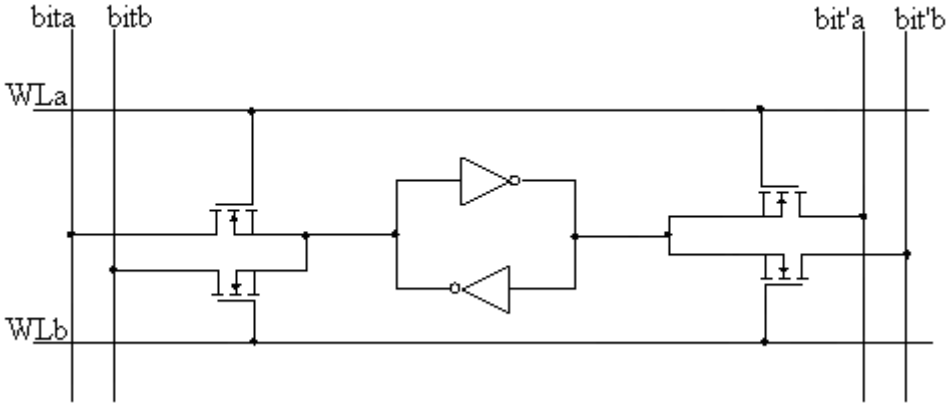
a) Operação de escrita.



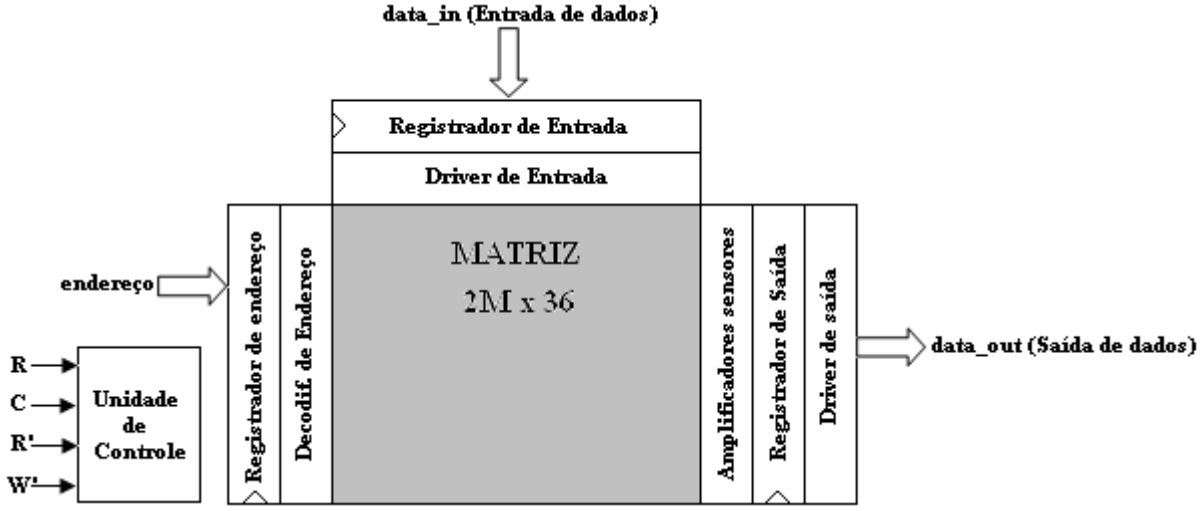
b) Operação de leitura.



c) Célula completa de porta dupla.



A QDRT é uma (Quad Data RateT) o nome que descreve a funcionalidade da arquitetura a qual permite dois portos rodar independentemente em dupla taxa de dados, a qual resulta em quatro itens por ciclo de clock ou quádrupla taxa de dados. A QDR SRAMs é o alvo da próxima geração de chaves e roteadores que operam nas taxas de dados acima de 200MHz. As novas SRAMs são idealmente aceitas para aplicações largura de faixa alta onde elas servem como a memória principal para tabelas de consultas e outros. A seguir é apresentado um diagrama simplificado de uma SRAM QDR mostrando-se os dois barramentos de dados (data_in e data_out), mais o barramento de endereço, todos com registradores. O diagrama também mostra dois clocks, denominados K (para a escrita) e C (para a leitura). Os sinais R' e W' são respectivamente sinais de controle de leitura e escrita e a capacidade de memória é de 72Mbits, distribuídos em 2M linhas, cada uma com uma palavra de 36bits. O funcionamento das SRAMs QDR é baseado em rajadas síncronas de dados em pipeline (synchronous pipelined bursts).



CARACTERÍSTICAS

- 72Mbits;
- Organização dos bits em linhas e colunas;
- Freqüência máxima de operação 400MHz;
- Taxa de dados 800Mbps de entrada + 800Mbps de saída por linha;
- Comprimento do bloco (rajada) de dados: 1,2,4 ou 8bits;
- Tensão de alimentação de 1,8V;
- Tipo de I/O: HSTL-18.

As memórias SRAM são síncronas e podem operar no modo rajada (burst), versão pipeline e no modo (flow-through) (fluxo através). A diferença é que pode realizar a transição imediatamente entre um ciclo de leitura e um ciclo de escrita, sem a necessidade de pausas (latência ou turnaround).
Os portos duplos flow-through permitem o acesso aos dados sem latência. Em outras palavras, o dado de uma leitura é retornado no mesmo ciclo de clock. This is advantageous in applications where access time to a single piece of data is critical. A leitura na memória e o retornar o valor no mesmo ciclo resulta numa diminuição na freqüência de operação e, contudo, uma diminuição na largura de faixa. O pipeline porto duplo aumenta a largura de faixa do dispositivo pelo particionamento da operação de leitura em dois passos. O arranjo de memória é acessado durante o primeiro ciclo de relógio. O dado lido é registrado e enviado à saída no segundo ciclo. Como resultado, os dispositivos pipeline têm um ciclo de latência para ler o dado. Entretanto, particionando o acesso em dois passos o ciclo de relógio pode ser mais curto e por isso a largura de faixa do dispositivo é incrementada. Não existe diferenças na operação de escrita entre os dispositivos flow-through e pipeline. Nos dispositivos futuros, os estágios adicionais pipelines podem ser adicionados. Neste caso, a latência para a leitura aumentará para mais de três ciclos, mas a vantagem do aumento da largura de faixa da memória. Todos os demais tipos de SRAM síncrona têm a limitação de não poderem passar imediatamente de uma leitura para escrita, ou vice-versa. A razão disso é que o sistema de endereçamento interno da memória tem diferenças, nas leituras e nas escritas. É necessário um tempo para a memória desativar internamente o endereçamento da leitura e ativar o endereçamento da escrita, e vice-versa. As memórias com as iniciais ZBT (Zero Bus Turnaround) ou NoBL (No Bus Latency) ou Network SRAM, onde o nome varia conforme o fabricante têm seus circuitos internos de endereçamento organizado de forma que o mesmo endereçamento usado para a leitura é usado também para a escrita, portanto não tem necessidade esperar pela desabilitação de um circuito e a habilitação de outro quando são feitas inversões entre operações de leitura e escrita.

MEMÓRIA PORTO DUPLO SRAM

Este é um tipo especial de memória que pode ser acessada simultaneamente por dois barramentos independentes. Na estrutura destas memórias, existem dois conjuntos de sinais independentes, com barramentos de dados, endereços e controle. Até os circuitos internos são simétricos e independentes. Ambos acessam uma única matriz de células de memória. Existem muitos casos em que são usadas memórias comuns e existe mais de um circuito que faz acessos. Apenas quando ocorre colisão, um circuito terá que esperar pelo acesso. Existem inúmeras aplicações para as memórias porto duplo. Um exemplo é a cache externa em placas com múltiplos processadores (fala-se aqui de máquinas mãos sofisticadas). Placas de vídeo de alto desempenho também podem fazer uso deste tipo de memória. Ao mesmo tempo em que a memória de vídeo está sendo lida e transferida para o monitor, o chip gráfico pode fazer seus acessos a esta mesma memória. Placas digitalizadoras de vídeo de alto desempenho também podem usar o mesmo recurso.

Tempo de Acesso

O tempo de acesso da memória é da ordem de 70ns. Não é usada a terminologia para o "tempo de acesso" no caso das memórias síncronas que ao invés é especificado o clock (ou o período de duração do ciclo) e a latência. Nos chips os sufixos indicam o clock ou o período dele dependendo do fabricante.

Ciclo	Clock
20 ns	50 MHz
15 ns	66 MHz
13,3 ns	75 MHz
12 ns	83 MHz
10 ns	100 MHz
08 ns	125 MHz
7,5 ns	133 MHz

Ciclo	Clock
7 ns	143 MHz
6 ns	166 MHz
5 ns	200 MHz
4 ns	250 MHz
3,3 ns	300 MHz
3 ns	333 MHz
2,5 ns	400 MHz

Uma leitura pode consumir dois ou três ciclos de 10ns para dispor dos dados no barramento de dados (latência), num total de 20 ou 30ns. A partir da leitura do primeiro dado, os três dados seguintes serão entregues a cada 10ns, desde que a memória esteja operando em modo rajada (burst).

Estados de Espera

O estado de espera é um recurso usado nos microprocessadores para compatibilizar a velocidade da CPU (mais rápida) com dispositivos mais lentos. Um sinal de pronto para a CPU libera a leitura, pois enquanto o sinal não chega o processador fica no estado de espera. Após o recebimento do sinal a CPU é liberada para ler ou escrever. Um acesso à memória, o processador espera normalmente dois ciclos. O ciclo tem duração de acordo com o clock externo do processador. Por exemplo, com clock externo de 100 MHz, a duração do ciclo é de 10 ns. Existe uma operação normal de leitura, com duração de dois ciclos e durante o primeiro ciclo o processador deve entregar o endereço ao barramento, juntamente com outros sinais de controle. No final do ciclo seguinte, o processador testa o sinal da sua entrada de controle "pronto". Se

estiver em nível zero, significa que o ciclo pode ser finalizado, e que o dado estará disponível no seu barramento de dados. O processador pode aguardar mais dois, três ou quantos estados de espera adicionais forem necessários, até que o circuito controlador da memória ative o sinal de pronto que finaliza o ciclo.

DRAM (MEMÓRIA DINÂMICA DE ACESSO ALEATÓRIO)

Como as SRAMs, como DRAMs (memórias dinâmicas de acesso aleatório) são memórias voláteis. Uma dificuldade nas memórias dinâmicas os dados são armazenados em capacitores e, portanto, necessitam de uma atualização dos dados através de um ciclo de refrescamento periodicamente de dois a cinco milisegundos. As características das memórias DRAMs são:

- Com células de armazenagem de pequeno tamanho permite a construção de memórias mais densas e de grande capacidade de armazenamento;
- As DRAMs são mais lentas que as memórias SRAMs;
- As DRAMs são mais baratas que as SRAMs;
- As DRAMs necessitam ciclo de refrescamento dos dados.

CIRCUITO DRAM

Uma célula DRAM com um transistor e um capacitor 1T-1C, conforme é mostrado o arranjo de 2 x 2 na figura em a) a seguir. O capacitor é construído verticalmente (trench capacitor) ou com múltiplas camadas empilhadas (stacked capacitor). A célula 1T-1C usando o capacitor é mostrado na figura a seguir, em b).

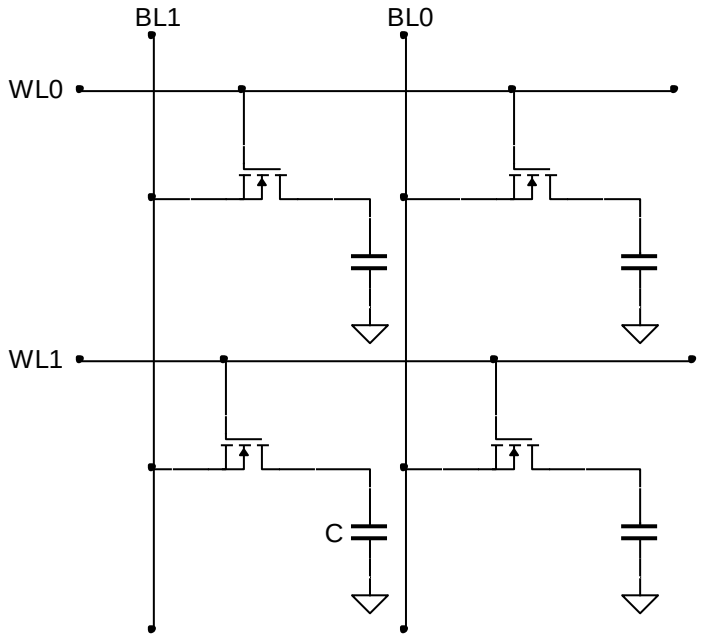


Figura: a) Arranjo DRAM de 2 x 2 com célula DRAM 1T-1C. b) Célula trench capacitor.

Na célula de armazenagem, o nó de armazenagem é uma depressão entalhada no substrato. No entalhe do silício uma depressão profunda é formada e um filme dielétrico entre as placas do capacitor.

PROCEDIMENTO DE LEITURA

ARQUITETURA DO DISPOSITIVO DRAM

A seguir é apresentada a arquitetura de uma memória DRAM de 256 x 256 x 4bits. As memórias DRAMs são de grande capacidade e usa a multiplexagem para o endereçamento. Esse procedimento reduz o número de pinos do dispositivo, mas obriga a introduzir um circuito de memorização para os endereços linhas-colunas (WLs e BLs) antes dos decodificadores. Dois sinais de controles devem ser criados para a seleção das linhas e colunas de endereços respectivamente RAS (row address strobe) e CAS (column addres strobe). Os sinais RAS e CAS são controlados externamente assim como os sinais CE' e OE'. A seguir é apresentada a arquitetura da DRAM de 64K x 4bits.

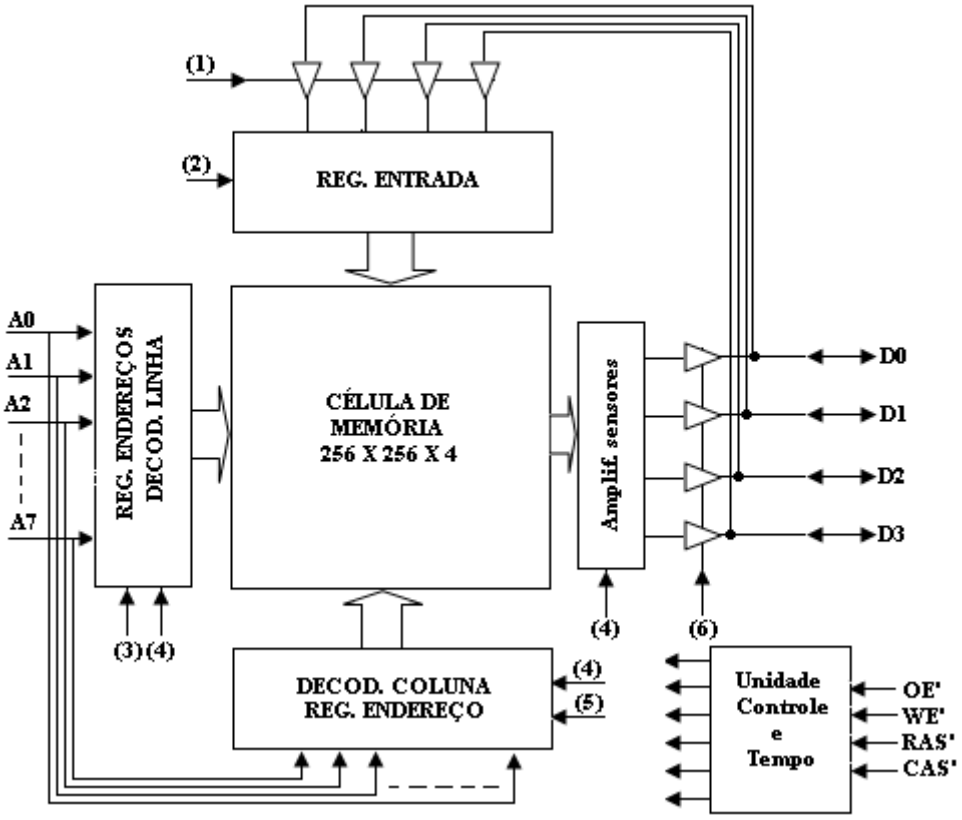


Figura: Arquitetura da DRAM de 256 x 4bits.

A tabela da verdade a seguir mostra a lógica dos sinais e a operação da memória.

RAS'	CAS'	WE'	OE'	I/O_0 a I/O_3	Operação
1	x	0	1	Hi-Z	STANDBY
0	0	1	0	D_{OUT}	Leitura
0	0	0	x	D_{IN}	Ciclo Recente Escrita
0	0	0	1	D_{IN}	Ciclo atrasado Escrita
0	0	$1 \rightarrow 0$	$0 \rightarrow 1$	D_{IN}/D_{OUT}	Ciclo modifica Leitura Escrita
0	1	x	x	Hi-Z	Ciclo de refrescamento somente RAS
$1 \rightarrow 0$	0	1	x	Hi-Z	CAS antes de RAS ciclo de refrescamento ou Ciclo de Auto-refrescamento
0	0	1	1	Hi-Z	Ciclo de Leitura Saída desabilitada

MULTIPLEXAGEM NA ENTRADA DO ENDEREÇO DA DRAM.

As memórias DRAMs são dispositivos com altíssima densidade de bits. Pode-se citar memórias DRAMs com capacidade de armazenagem de 128Mbits, 512Mbits e 1Gbits. O número de linhas de entrada de endereços é grande então o acesso à memória é feita multiplexando os endereços da matriz. Essa é uma das características das DRAMs e a multiplexagem dos endereços de entrada reduz a pinagem dos endereços pela metade. Para o controle dos endereços, dois sinais são utilizados denominados de RAS e CAS. A entrada de endereços da memória é feita por pinagem A demultiplexagem dos endereços é feita por dois latches controlados pelos sinais RAS e CAS. A seguir é apresentada uma forma de entrada nos pinos de endereços de entrada de uma memória DRAM.

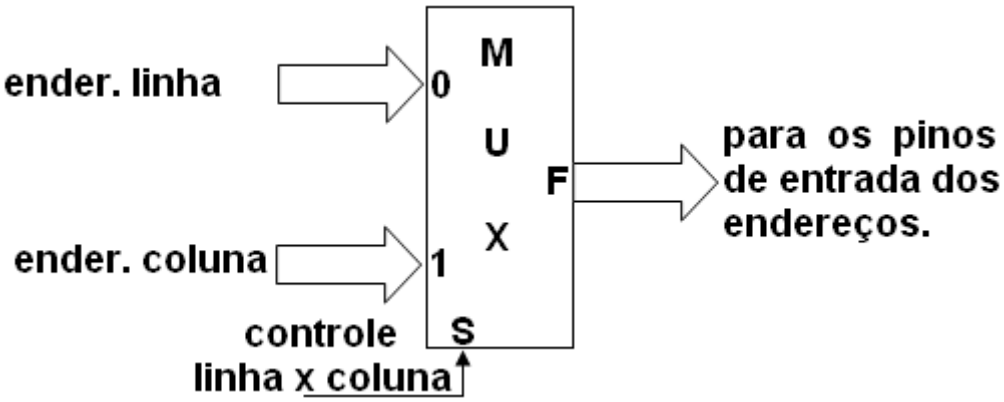
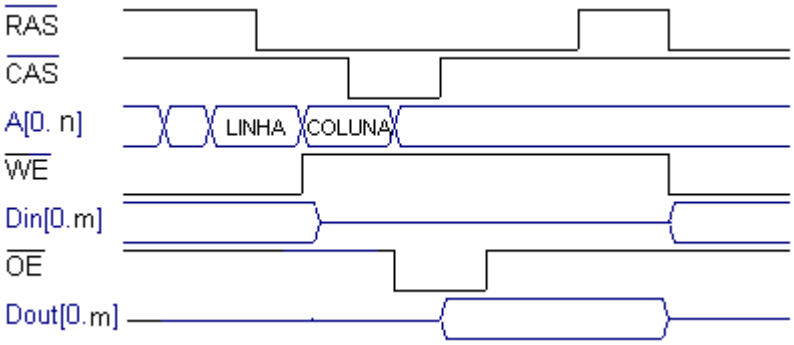
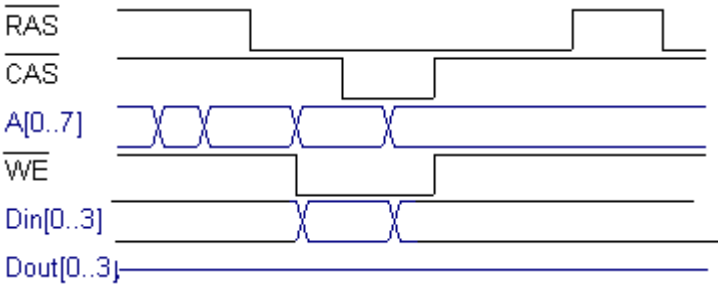


Figura: Multiplexagem das entradas de endereços linhas e colunas.

Ciclo de Leitura



Ciclo de Escrita Recente



CICLO DE RESFRESCAMENTO DA DRAM

As memórias DRAMs necessitam periodicamente que seus dados armazenados sejam refrescados nas suas células de armazenagem. Os capacitores têm fuga e se a carga for resposta dentro de um limite de tempo perde a informação. Daí de tempos em tempos prioritariamente o processador interrompe qualquer tarefa para atender ao ciclo de refrescamento. Uma noção de como é refrescado a informação numa DRAM é mostrado a seguir. Como o arranjo no endereçamento é matricial linha x coluna, o refrescamento é feito percorrendo todos os estados do decodificador de linhas. O ciclo de refrescamento é de 2 a 4ms e nesse intervalo todas as linhas devem ser ativadas. A figura a seguir mostra um esquema de refrescamento realizado por chaves de entrada e saída.

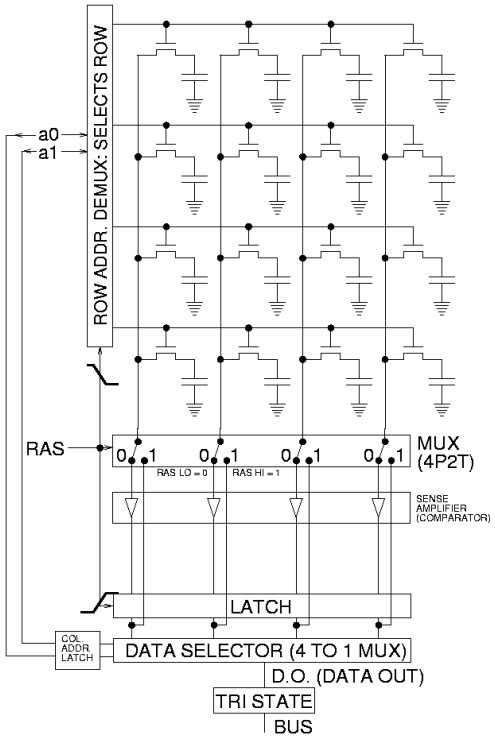


Figura: Célula de representação de um circuito que realiza o refrescamento dos dados.

Conforme é visto na figura o circuito deve refrescar os dados nos capacitores e ele executa essa ação varrendo todas as linhas do decodificador de endereços da memória. As chaves S1, S2 e S3 são transistores MOS operando como uma chave (corte e saturação). Quando o processo é uma escrita as chaves S1, S2 estão fechadas e a chave S3 aberta. Quando a operação é leitura do dado, um amplificador realiza a leitura do valor do capacitor até 50% da carga é nível um e abaixo de 50% da carga é zero. A chave S1 aberta e as chaves S2 e S3 fechadas, um auto-refrescamento é realizado na operação de leitura. Para o refrescamento total de todas as células de armazenagem, basta enviar à memória o comando de leitura e varrer todos os endereços da memória através somente do decodificador de linhas, que a memória é refrescada. A seguir apresenta-se um circuito de representação da memória DRAM.

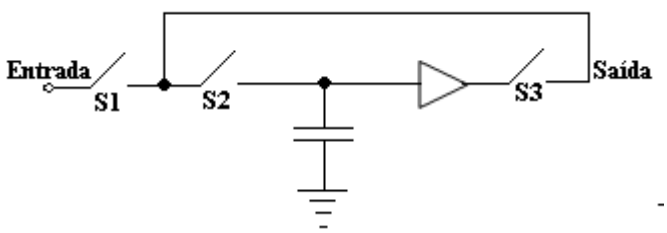


Figura: Representação do funcionamento do circuito de refrescamento das memórias DRAMs.

EXPANSÃO DO BANCO DE MEMÓRIA E CRIAÇÃO DOS MÓDULOS DE MEMÓRIAS.

Uma expansão de memória é sempre necessária para aumentar a capacidade de armazenagem da memória. A expansão pode ser no comprimento da memória com o aumento do número de endereços da memória ou pode ser na largura, com o aumento no tamanho dos bits de saída. Deve-se preservar na associação os sinais de controle da memória como CS' (chip select) ou o seletor da pastilha. Para a associação de memórias deve-se criar uma metodologia que leva a associação perfeita. A seguir, seguem os passos necessários para o projeto da associação de memórias.

1. Cálculo do número de memórias em função do tamanho do banco de memória desejado;
2. Um esquema de representação em bloco do banco de memória desejado com todos os sinais;
3. Determinação se o aumento da capacidade final é na largura ou nio comprimento em relação ao chip de partida;

Os três passos se seguidos vai levar à associação perfeita das memórias. De acordo com o primeiro passo o número de memórias necessárias para a formação do banco de memória é assim calculado:

Número de memórias = total do banco de memória/chip de partida.

Após o primeiro passo, partimos para a determinação do tipo de aumento na associação e se o total de endereços do banco é igual ao total de endereço do chip. Se for igual o aumento será na largura e se for diferente o aumento será no comprimento, embora é possível ter crescimento nas duas formas. Para estudo considera-se o aumento inicial somente pela largura da memória associada e para ser prático, vamos a um exemplo.

Exemplo: Criar um banco de memória de 16 x 8bits usando somente chips de memória de 16 x 4. Pede-se:

a) Número de memórias necessárias.
b) Desenho da configuração das memórias para a formação do banco de memórias.

MEMÓRIAS DRAM ATUAIS

Atualmente os dispositivos cresceram na capacidade de armazenamento como é visto no dispositivo da série HM5165405F da ELPIDA. O dispositivo é de 64M EDO DRAM (16Mword x 4-bit), sendo uma memória de alta capacidade de armazenagem baixo consumo e tecnologia CMOS e oferece um EDO (Extended Data Out) Page Mode como modo de acesso de alta velocidade. O chip tem encapsulamento do tipo SOJ ou padrão tipo TSOPII.

Características

- Fonte única de +3,3V;
- Tempo de acesso 50 a 60ns;
- Dissipação em potência: Ativa 468/396mW (max) e Standby 1,8mW;
- Capacidade EDO modo página;
- Ciclo de refrescamento – RAS' (somente refrescamento) ciclo 8192 em 64ms.

DESCRIÇÃO DE PINOS

A_0 a A_{11} – Endereço de entrada para: Linha/refrescamento – A_0 a A_{11} e
Coluna – A_0 a A_{11}.

I/O_{0-3} – Dados Entrada e saída;
RAS' – Habilitação do endereço da linha.
CAS' – Habilitação do endereço da coluna.
WE' – Sinal de escrita e leitura de dados;
OE' – Sinal de habilitação da saída de dados.

A seguir é apresentado o diagrama de bloco da arquitetura do chip HM5165405F.

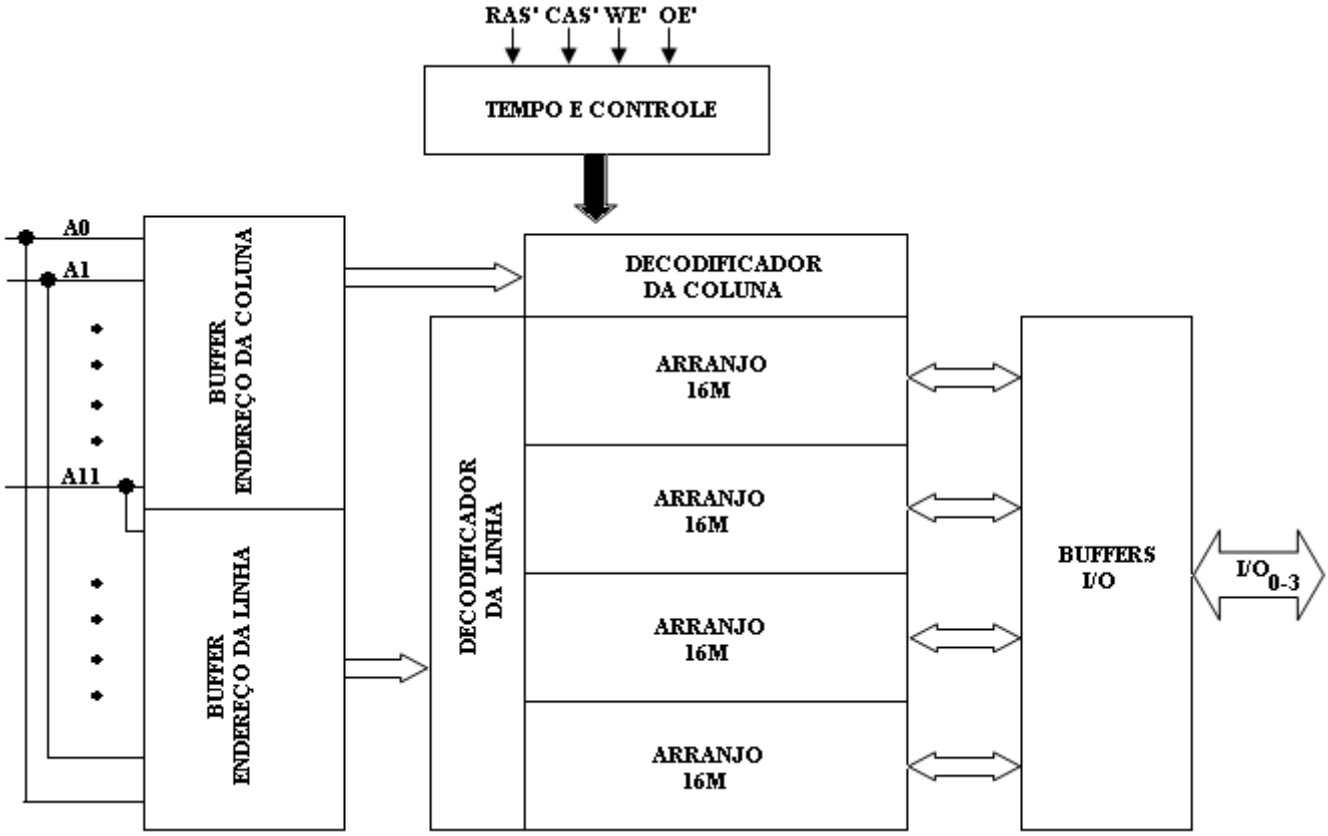


Figura: Arquitetura da DRAM de 64K x 4 EDO modo paginação.

A seguir a memória é descrita pela tabela da verdade com os sinais de controle.

RAS'	*CAS'*	*WE'*	*OE'*	*I/O$_{0-3}$*	**Operação**
H	x	x	x	Hi-Z	Standby
L	L	H	L	D_{OUT}	Ciclo de Leitura
L	L	L	x	D_{IN}	Ciclo de Escrita
L	L	L	H	D_{IN}	Ciclo atrasado de Escrita
L	L	H → L	L → H	D_{OUT}/D_{IN}	Modificado ciclo de Leitura/Escrita
L	H	x	x	Hi-Z	Ciclo de refrescamento somente RAS
H → L	L	H	x	Hi-Z	Ciclo de refrescamento CAS antes do RAS
					Ciclo de auto-refrescamento versão L
L	L	H	H	Hi-Z	Ciclo de Leitura (Saída desabilitada)

O histórico de evolução das memórias passa pela necessidade cada vez mais de aumento da capacidade de armazenagem e da velocidade compatível com os modernos computadores. As memórias síncronas ganharam bastante força, pois são memórias de alto desempenho e de grande capacidade. As arquiteturas das memórias SDRAM e outras como DDRAM e outras DDRAM2 e DDRAM3.

SDRAM – SIMM

O acrônimo SIMM (Single in-line memory module) é um módulo de memória com via único com pelo menos 72 pinos de entrada e saída. Esse módulo foi muito utilizado nos computadores de mesa, mas foram base para novas versões de memórias DRAM. Para entender melhor como as memórias DRAMs montadas em via única SIMM, a seguir apresenta-se a arquitetura e distribuição da pinagem de uma memória SDRAM de 128M byte com arranjo de memória 16M x 32bits.

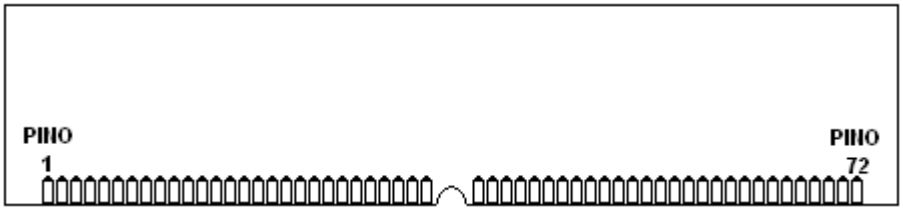


Figura: Módulo SIMM – 72 pinos.

SDRAM - DIMM

O acrônimo DIMM (Dual in-line memory module) é um módulo de memória com via dupla com pelo menos 168 pinos de entrada e saída. Esse módulo foi muito utilizado nos computadores de mesa, mas foram base para novas versões de memórias DRAM. Para entender melhor como as memórias DRAMs montadas em via dupla DIMM, a seguir apresenta-se a arquitetura e distribuição da pinagem de uma memória SDRAM de 512M byte com arranjo de memória 16M x 72bits.

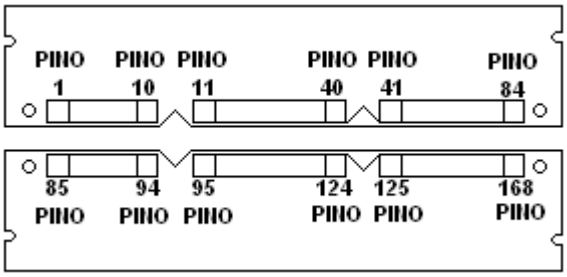


Figura: Modulo DIMM – 168 pinos.

OPERAÇÃO DA MEMÓRIA SÍNCRONA – SDRAM

As memórias síncronas são memórias cuja maior aplicação é para máquinas que usam baterias, tais como PDAs, telefones celulares 2.5G e 3G com acesso a Internet e capacidade multimídia, mini-notebooks, handheld PCs. Por exemplo, a memória uma memória SDRAM da Hynix 128M Mobile SDRAM cuja capacidade de armazenagem é de 134.217.728bit CMOS Mobile Synchronous DRAM(Mobile SDR) é muito requerida nas aplicações de computadores como a memória principal, pois possui grande densidade de memória e uma alta largura de faixa e é organizada como um banco de 2.097.152 x 16.
Diferente das DRAMs cuja interface é assíncrona, as memórias SDRAMs possuem uma interface síncrona com o relógio. A SDRAM é um tipo de memória DRAM a qual opera em sincronismo com o sinal de entrada de relógio (clock) e "latch" cada sinal de controle na borda de subida de relógio (CLK) e os dados de entrada/saída estão em sincronismo com a entrada do relógio. As linhas de endereços são multiplexadas com as linhas dos dados de entrada por um barramento multiplexado de 16 Entrada/Saídas.

PIPELINE

A estrutura interna da SDRAM apresenta uma unidade de controle seqüencial operando com instruções em "pipeline". O pipeline permite que várias instruções sejam executadas concorrentemente enquanto operações estão sendo executadas. Por exemplo, é possível durante um ciclo pipeline de escrita que se busque e se execute novas instruções, antes que o dado seja escrito na memória e essas operações ocorrem simultaneamente, ou seja na forma concorrente. Num ciclo pipeline de leitura um número de fixo de pulsos de relógio são necessários antes que o dado seja disponível na saída e durante essa espera novos ciclos ou outras instruções são executados. Isso define o chamado tempo de latência da memória e este é um parâmetro importante na escolha da SDRAM.

CICLO DE PRÉ-BUSCA

Um buffer de pré-busca é um tipo buffer de dados que é utilizado nos computadores modernos e especificamente no chip de memória e que é responsável pelo acesso real e rápido á múltiplas palavras localizadas nas linhas comuns dos endereços físicos da memória.
A característica do buffer pré-busca apresenta vantagem, em relação ao acesso à memória DRAM, pois as operações realizadas na DRAM são: pré-carga das linhas bit-line, acesso às linhas e acesso às colunas.
O acesso físico às linhas de endereços é o centro da operação de leitura na memória e os sinais na DRAM são longos e lentos, entretanto quando há uma leitura nas linhas de endereços, o acesso às colunas subseqüentes à mesma linha de endereço será mais rápido, uma vez que os amplificadores sensores também atuam como latches.

Exemplo

Uma memória SDRAM – DDR3, cuja largura da fila é de 2.048 bits e onde internamente os 2,048 bits são lidos em 2,048 amplificadores sensores separados durante a fase de acesso à linha comum de endereço. Um acesso à linha de endereço consome um tempo de 50 ns e depende da velocidade da DRAM, considerando que o acesso às colunas de endereços com a linha aberta de endereço é menor do que 10 ns.
As arquiteturas tradicionais DRAM têm suportado um longo acesso rápido às colunas de endereços para os bits sobre uma fila aberta de endereços.
Para um chip cuja memória é de largura igual a 8 bits com uma linha de endereços igual a 2.048 bits de largura, o accesso a qualquer das 256 datawords (2048/8) sobre a linha de endereços pode ser muito rápida.
A desvantagem do método velho de acesso rápido às colunas de endereços é que um novo endereço da coluna tem que ser enviado para cada adicional dataword sobre a linha comum de endereço.
O barramento de endereço tem que operar na mesma freqüência do barramento dos dados. Um buffer de pré-busca simplifica este processo permitindo que um único acesso ao endereço resulte em múltiplas datawords.
Na arquitetura pré-busca, quando ocorre um acesso à memória à uma linha de endereço o buffer se apropria do conjunto de datawords adjacentes sobre a linha de endereço e faz a leitura deles de forma de rajada numa seqüência rápida sobre os pinos de entrada e saída, sem a necessidade de requisitar as colunas individuais de endereços. O dispositivo entende que a CPU deseja as datawords adjacentes na memória as quais na prática é muito freqüentemente o caso. Por exemplo, quando uma CPU de 64 bits acessa um chip de DRAM e de largura igual a 16 bits, então necessita das quatro datawords adjacentes de 16 bits para o total de 64 bits. Um buffer pré-busca 4n ("n" se refere à largura do chip de memória e é multiplicada pela profundidade da rajada "4" para dar o tamanho em bits da seqüência total da rajada. Um buffer pré-busca 8n sobre a largura de 8 bits da DRAM terá uma transferência de 64 bits.

Largura de faixa

A profundidade do buffer pré-busca é a razão entre a freqüência da memória e a freqüência de entrada/saída. Na arquitetura do pré-busca 8n, como a DDR3, a entrada e saída opera 8 vezes mais rápida do que o núcleo da memória (cada acesso à memória resulte numa rajada de 8 datawords sobre a entrada/saída. Se o núcleo da memória é de 200 MHz é combinado com a entrada/saída operará oito vezes mais rápida (1600 megabits/s). Se a memória tem 16 entradas e saídas, a largura de faixa total da leitura será igual a 200 MHz x 8 datawords/accesso x 16 entradas/saídas = 25.6 gigabits/s (Gbps), ou 3.2 gigabytes/s (GBps). Os módulos com múltiplos chips DRAM podem providenciar correspondentemente mais altas larguras de faixas. Cada geração de SDRAM tem diferentes tamanhos de buffers pré-busca:

- DDR SDRAM é um buffer de pre-busca com tamanho igual a 2n (duas datawords por à memória);
- DDR2 SDRAM é um buffer pré-busca com tamanho igual a 4n;
- DDR3 SDRAM é um buffer pré-busca com tamanho igual a 8n (oito datawords por acesso à memory).

Incremento da Largura de faixa

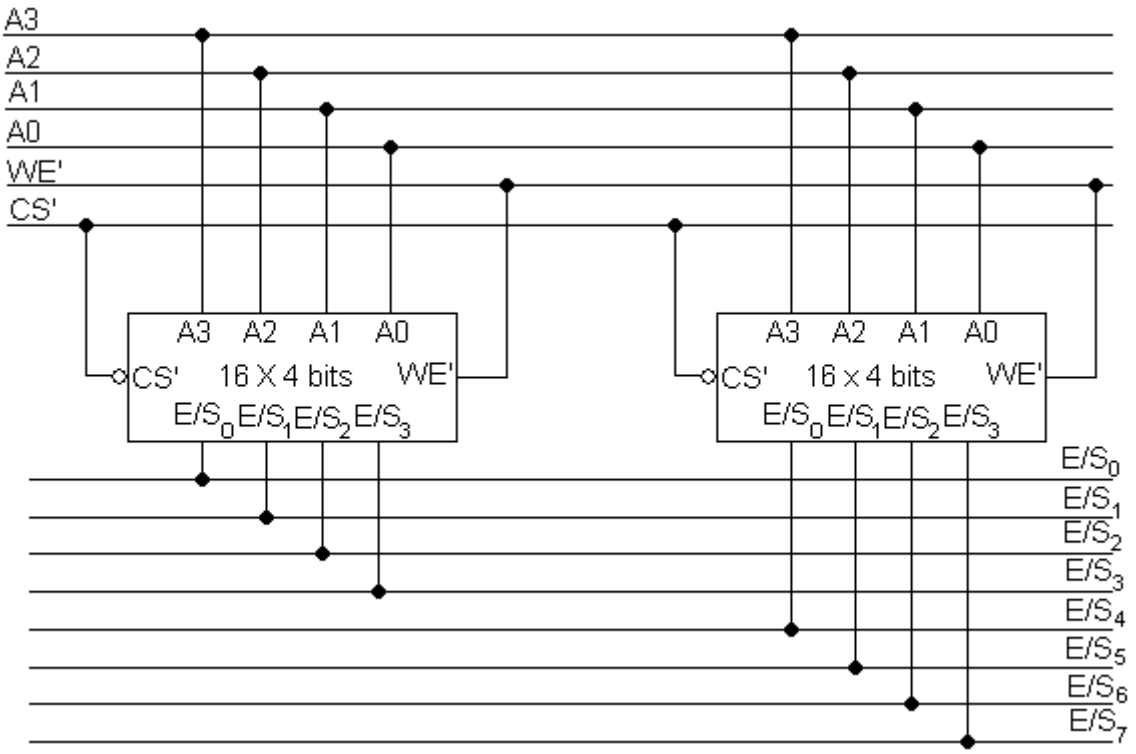
A velocidade da memória não tem historicamente incrementado melhoramentos com inline com a CPU. Para o incremento da largura de faixa, o módulo de memória com buffer de pré-busca lê simultaneamente os dados de múltiplos chips de memórias. A largura de faixa para o acesso seqüencial é melhorado usando buffers de pré-busca, mas acesso aleatório não é alterado.

ASSOCIAÇÃO DE MEMÓRIAS

A necessidade de aumentar a capacidade de memória em relação à capacidade do dispositivo de memória leva à associação de dispositivos de memórias. Neste capítulo far-se-á de duas formas, sendo a primeira uma associação usando somente um tipo de dispositivo e a segunda usando vários tipos de dispositivos com capacidades diferentes. A associação pode ser feita de três maneiras, a saber:
Aumento no comprimento da memória com aumento do número de endereços da memória;
Aumento na largura da memória com aumento no tamanho da palavra da memória;
Aumento no comprimento e na largura da memória com crescimento em ambos os endereços e os conteúdos da memória.

Aumento da largura da memória

O aumento na largura da memória é o incremento no barramento dos dados de n para m bits. A manutenção do barramento dos endereços.



Memória 1 – CS' = 0

F				
E				
D				
C				
B				
A				
9				
8				
7				
6				
5				
4				
3				
2				
1				
0				
End.	**Bit0**	**Bit 1**	**Bit 2**	**Bit 3**

MEMÓRIA 2 – CS' = 0

F				
E				
D				
C				
B				
A				
9				
8				
7				
6				
5				
4				
3				
2				
1				
0				
End.	**Bit 4**	**Bit 5**	**Bit 6**	**Bit 7**

Aumento do comprimento da memória

O aumento no comprimento da memória se resume no crescimento do número de linhas de endereçamento. Por exemplo, para a facilidade de análise vamos trabalhar com um único dispositivo de capacidade igual a 16 x 4bits e o objetivo é montar um banco de memória, associando esse dispositivo, cuja capacidade é de 32 x 4bits. Para ilustração o bloco abaixo é o resultado final da associação.

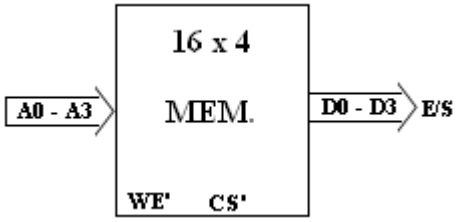


Figura: Chip de memória de 16 x 4bits.

Para a montagem do banco pode-se calcular o número de dispositivos necessários da forma:
Número de dispositivos = total do banco/capacidade do dispositivo.
Para o exemplo, terá
Número de dispositivos = 32 x 4/16 x 4 = 2 memórias.

Para o endereçamento de 32 linhas há a necessidade de cinco bits de A0 a A4. As linhas denominadas comuns são as linhas de endereços as quais são iguais nos dois dispositivos. Como cada dispositivo é o mesmo, então as linhas comuns são as mesmas linhas de endereços do chip, ou seja, A0 a A3. A linha A4 será a linha de seleção. Isso pode ser usado como regra geral em todos os casos. Como os endereços das duas memórias são linhas comuns, então num endereçamento real, como as memórias sabem se o usuário quer acessar uma ou outra? A resposta é simples a linha A4 será a linha de seleção e vai selecionar qual das duas memórias deve operar, quando A4 = 0 estamos buscando o acesso aos endereços de 0 a 15 e quando A4 = 1 estamos buscando o acesso para aos endereços de 16 a 31. Cada dispositivo de memória tem uma entrada de habilitação chamada de seletor do dispositivo (chip select) que ativo permite o acesso à memória e não ativo desliga completamente a saída da memória (terceiro estado). A utilização desse

recurso permite A4 selecionar uma ou outra das memórias, mas a utilização dessa entrada usando somente a entrada A4 ocupa esta entrada e, portanto, deve-se criar uma nova entrada CS para o banco, conforme esquema de representação. A seguir implementa-se a lógica do CS para ativação de uma ou outra memória por A4. Sendo as variáveis de entrada CS(do banco) e A4 linha de endereço e as variáveis de saídas CS1 da memória de 0 a 15 e CS2 da memória de 16 a 31. As linhas CS ativam com nível lógico zero e a memória está pronta para o acesso para nível lógico 1 a memória está desligada.

CS’	A4	CS1’	CS2’
0	0	0	1
0	1	1	0
1	x	1	1

As expressões booleanas serão:
CS1’ = (/CS’. A4’)’ = (CS.A4’)’ = CS’ + A4
CS2’ = (/CS’.A4)’= (CS.A4)’ = CS’ + A4’
Podemos representar as entradas CS’s com um símbolo de inversão e chamar essas entradas de CS1’ e CS2’.

As implementações da lógica dos CS’s serão com duas portas OU de duas entradas e um inversor para a linha de endereço A4. O circuito da associação fica:

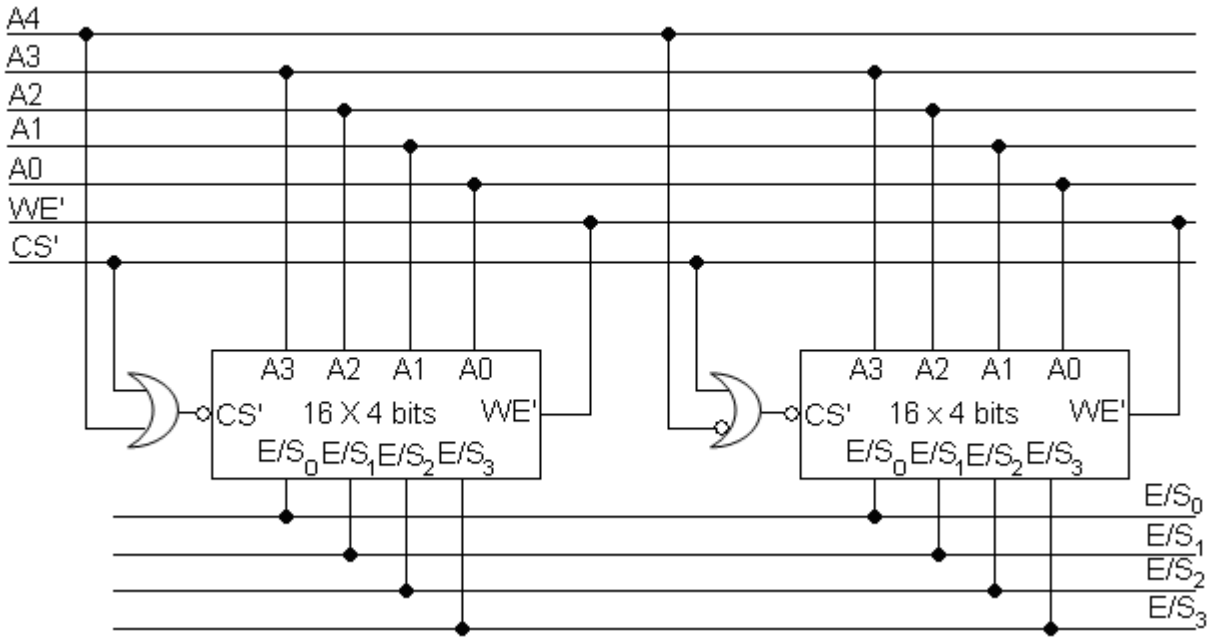


Figura: Banco de memória de 32 x 4 usando chips de memória de 16 x 4.

Memória 1 A4 = 0 – CS1' = 0

F				
E				
D				
C				
B				
A				
9				
8				
7				
6				
5				
4				
3				
2				
1				
0				
End.	**Bit0**	**Bit 1**	**Bit 2**	**Bit 3**

Memória 2 A4 = 1 – CS2' = 0

1F				
1E				
1D				
1C				
1B				
1A				
19				
18				
17				
16				
15				
14				
13				
12				
11				
10				
End.	**Bit0**	**Bit 1**	**Bit 2**	**Bit 3**

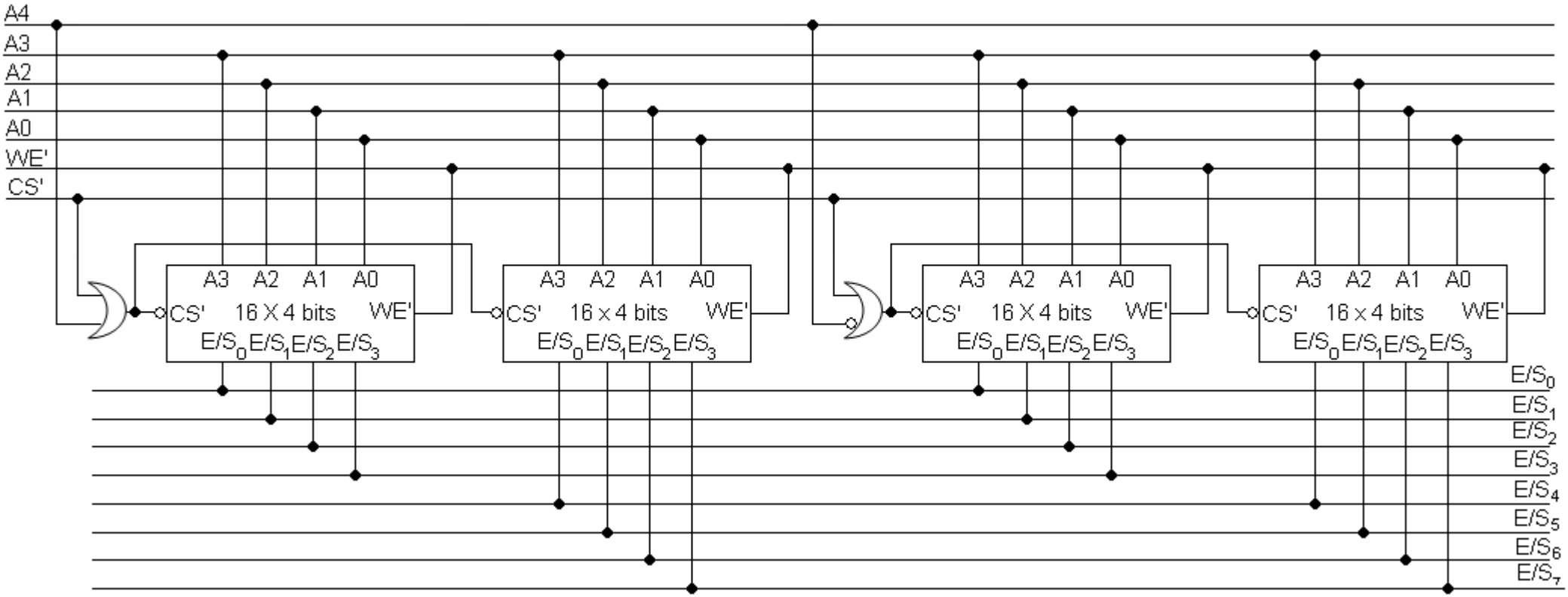


Figura: Banco de memória de 32 x 8 bits usando chips de memórias de 16 x 8 bits.

Exemplo: Construir um banco de memória de 8K x 8 bits, usando memória de 1k x 8 bits. Para o circuito de seleção usar um decodificador comercial 74138, conforme descrito pela tabela da verdade a seguir, Realizar a configuração do banco.

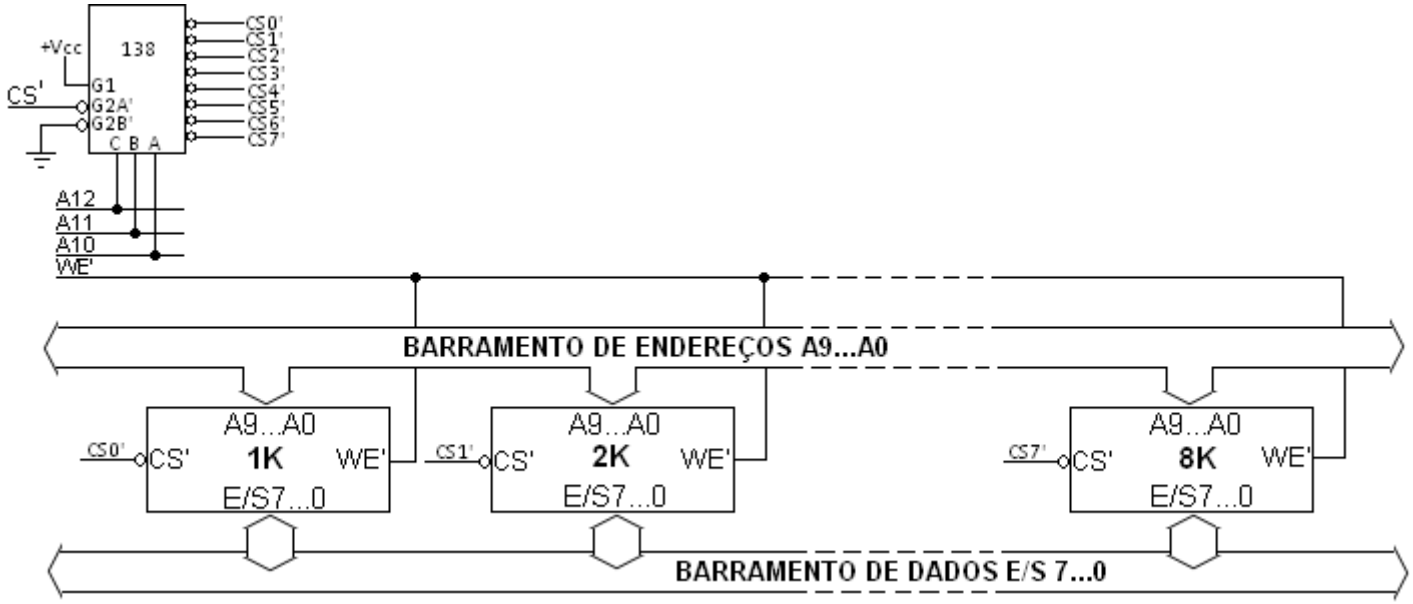


Figura: Banco de memória de 8K x 8 bits usando chips de memória de 1K x 8 bits.

Exemplo: Construir um banco de memória conforme mapa de memória a seguir.

Endereço Inicial, Mapa de 0 a 15K	memória
11264D - 2C00H	4K
10240D - 2800H	1K
7168D - 1C00H	3K
3072D - 0C00H	4K
1024D - 0400H	2K
0D - 0000H	1K

a) Configuração do banco de memória usando PLD - Decodificador.
b) Idem anterior usando a PLD – ROM.
c) Idem anterior usando a PLD – PAL.

a) Decodificador. 4 x 16 saídas.

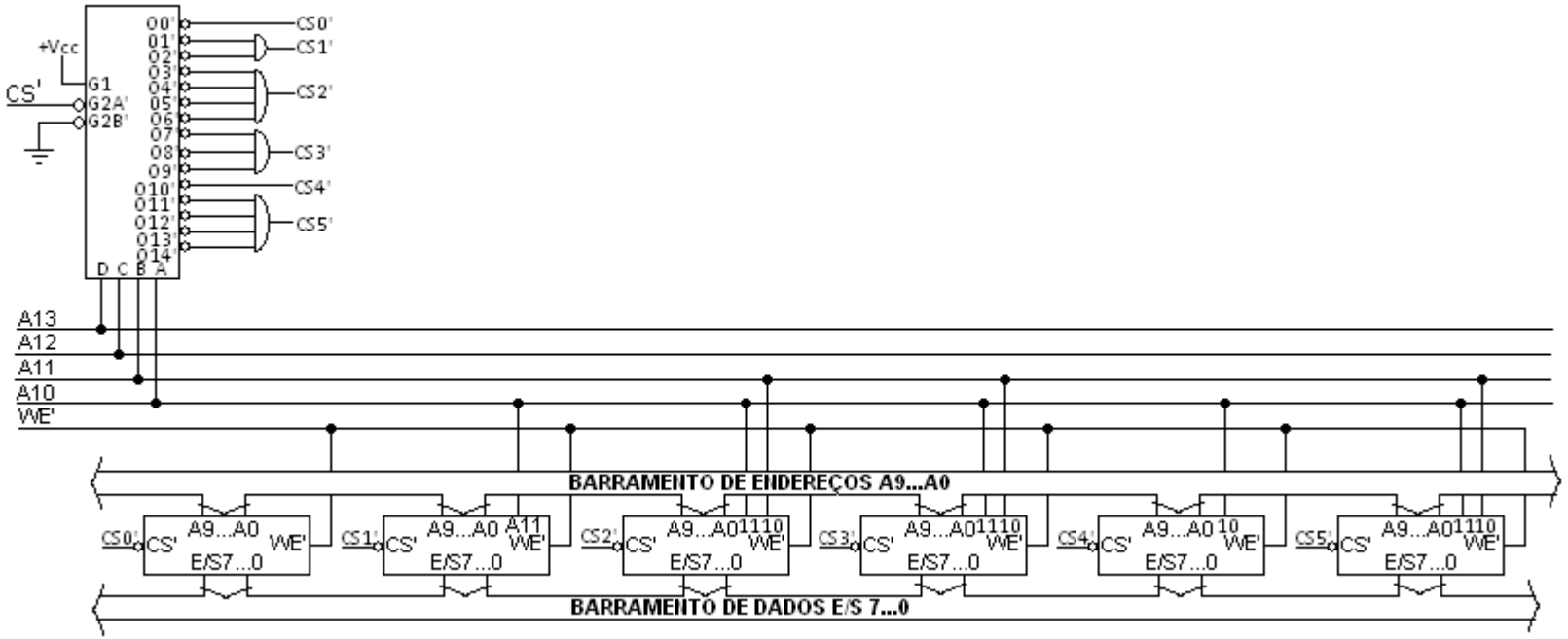


Figura: Banco de 15K x 8 bits usando chips de diversos tamanhos diferentes com decodificador.

b) ROM.

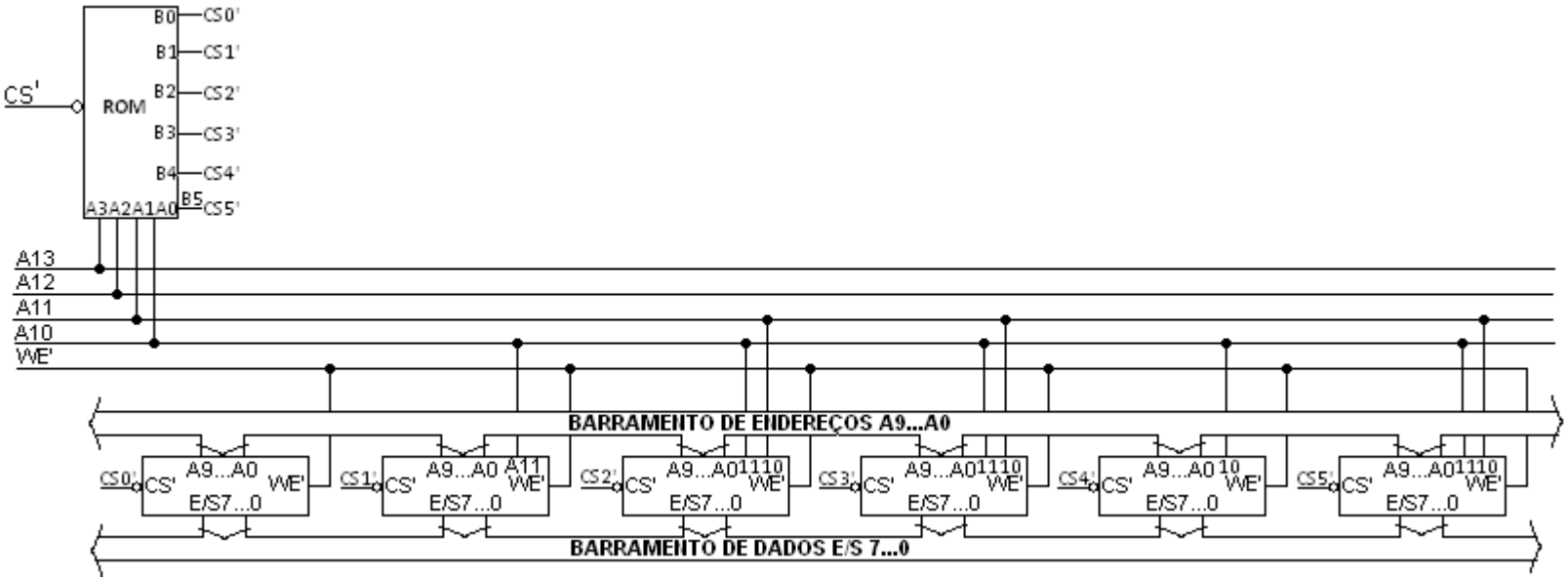


Figura: Banco de 15K x 8 bits usando chips de diversos tamanhos e implementação com ROM.

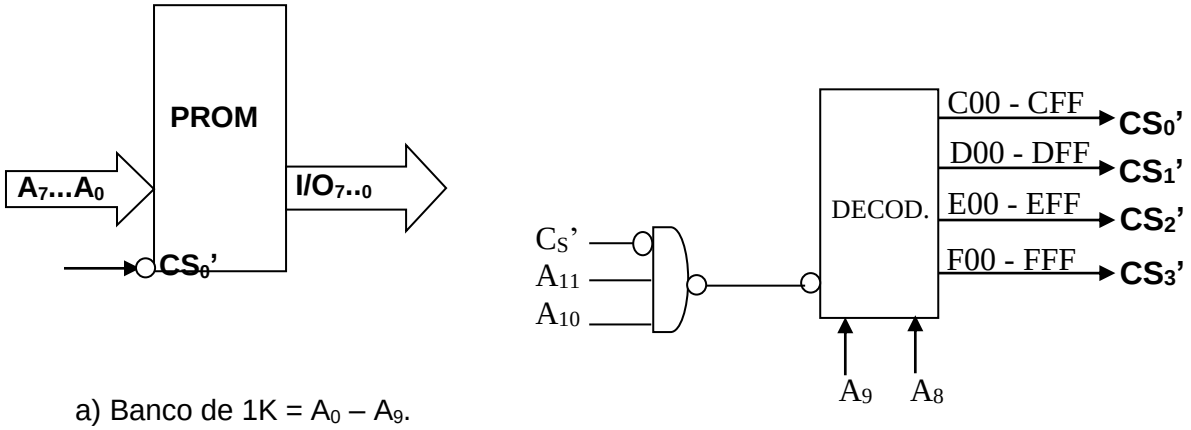
Mapa da ROM.

CS'	A13	A12	A11	A10	CS0'	CS1'	CS2'	CS3'	CS4'	CS5'
1	X	X	X	X	1	1	1	1	1	1
0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	0	1	1	1	1
0	0	0	1	0	1	0	1	1	1	1
0	0	0	1	1	1	1	0	1	1	1
0	0	1	0	0	1	1	0	1	1	1
0	0	1	0	1	1	1	0	1	1	1
0	0	1	1	0	1	1	0	1	1	1
0	0	1	1	1	1	1	1	0	1	1
0	1	0	0	0	1	1	1	0	1	1
0	1	0	0	1	1	1	1	0	1	1
0	1	0	1	0	1	1	1	1	0	1
0	1	0	1	1	1	1	1	1	1	0
0	1	1	0	0	1	1	1	1	1	0
0	1	1	0	1	1	1	1	1	1	0
0	1	1	1	0	1	1	1	1	1	0

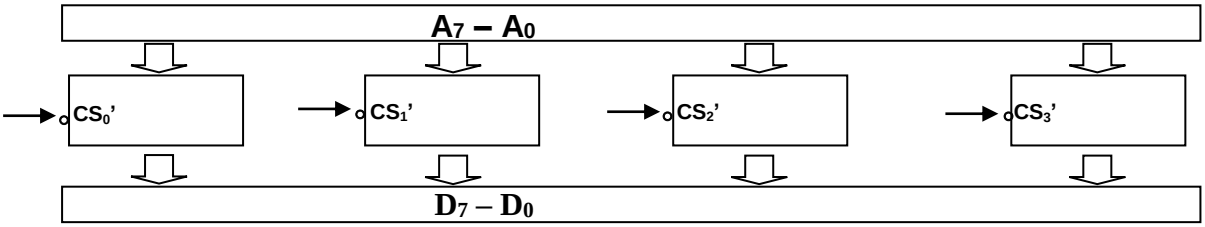
Capacidade da ROM = 15 x 6 bits.

c) PAL – As equações SÃO:
CS0’ = [(A13’.A12’.A11’.A10’).CS]’
CS1’ = [A13’A12’(A11.A10’ + A11’.A10)CS]’
CS2’ = [A13’.A12’.A11.A10 + A13’A12(A11’ + A11A10’)CS]’
CS3’ = [(A13’.A12.A11.A10 + A13.A12.A11’)CS]’
CS4’ = [(A13.A12’A11.A10’)CS]’
CS5’ = [(A13.A12’.A11.A10 + A13.A12)CS]’

Exercício: Conforme a tabela a seguir, construir:
a) Um banco de 1K memórias PROM associadas.
b) Implementação do sistema de seleção usando ROM como decodificador.
c) Idem item b) implementação com PAL.
Endereço inicial do banco é C00H.



a) Banco de 1K = $A_0 – A_9$.



b) ROM

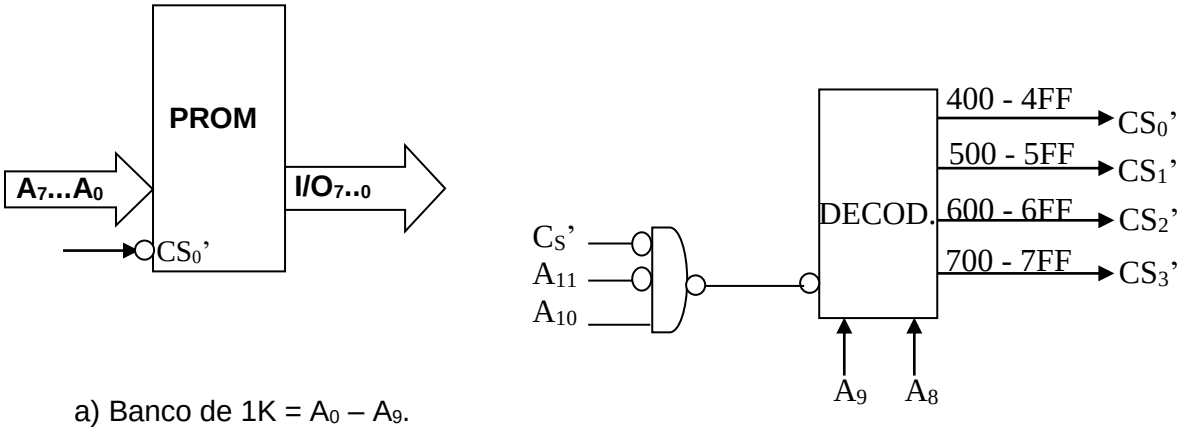
A9	A8	CS0'	CS1'	CS2'	CS3'	-	-
A1	A0	B0	B1	B2	B3	End.	Cont.
0	0	0	1	1	1	0	7
0	1	1	0	1	1	1	B
1	0	1	1	0	1	2	D

C = 4 x 4.

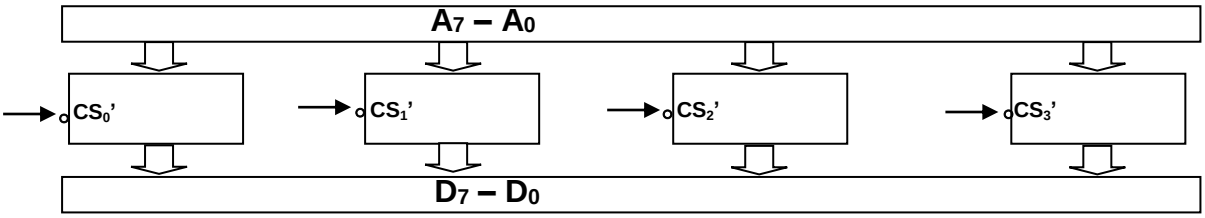
c) PAL

$CS_0' = [CS'(A_{11}A_{10}A_9'A_8')]'$
$CS_1' = [CS'(A_{11}A_{10}A_9'A_8)]'$
$CS_2' = [CS'(A_{11}A_{10}A_9A_8')]'$
$CS_3' = [CS'(A_{11}A_{10}A_9A_8)]'$

Exercício: Conforme a tabela a seguir, construir:
a) Um banco de 1K memórias PROM associadas.
b) Implementação do sistema de seleção usando ROM como decodificador.
c) Idem item b) implementação com PAL.
Endereço inicial do banco é 400H.



a) Banco de 1K = $A_0 – A_9$.



b) ROM

A_9	A_8	CS_0'	CS_1'	CS_2'	CS_3'	-	-
A_1	A_0	B_0	B_1	B_2	B_3	End.	Cont.
0	0	0	1	1	1	0	7
0	1	1	0	1	1	1	B
1	0	1	1	0	1	2	D
1	1	1	1	1	0	3	E

C = 4 X 4.

c) PAL

$CS_0' = [CS'(A_{11}'A_{10}A_9'A_8')]'$
$CS_1' = [CS'(A_{11}'A_{10}A_9'A_8)]'$
$CS_2' = [CS'(A_{11}'A_{10}A_9A_8')]'$
$CS_3' = [CS'(A_{11}'A_{10}A_9A_8)]'$

Exercício: De acordo com o mapa a seguir um banco de memória de capacidade 16K x 8. Pede-se:
a) O projeto do decodificador realizado com memória ROM. Tabela de endereços e conteúdos, sabendo-se que cada dispositivo dispõe de um seletor de chip CS'i, onde i = 0 a 3.
b) As equações booleanas para geração com PAL.

4K
4K
6K
2K

2000

Exercício: De acordo com o mapa a seguir um banco de memória de capacidade 16K x 8. Pede-se:

a) O projeto do decodificador realizado com memória ROM. Tabela de endereços e conteúdos, sabendo-se que cada dispositivo dispõe de um seletor de chip CS'i, onde i = 0 a 3.
b) As equações booleanas para geração com PAL.

2K
4K
6K
4K

2000

Exercício: Determinar para a faixa de endereços apresentada na tabela a seguir. Sabendo-se que o endereço inicial é igual a (0400)16, e o banco inicia pela memória de 4K inferior. Pede-se:
a) Indicar a faixa de endereços de cada CI no quadro abaixo.
b) A equação de cada seletor de pastilha (CS = ativo com zero) cada CI.
c) A equação do seletor do banco de memória (CS = ativo com zero). O decodificador de seleção é uma ROM com seleção de pastilha (CS = ativa com zero).

CI	Faixa de Endereço em Hex
4K	2000 – 2FFF
1K	1C00 – 1FFF
1K	1800 – 1BFF
1K	1400 – 17FF
4K	0400 – 13FF

As linhas $A_{15} = A_{14} = 0$ e $A_0 – A_9$ = Linhas comuns.

A_{13}	A_{12}	A_{11}	A_{10}	CS_1	CS_2	CS_3	CS_4	CS_5
0	0	0	0	1	1	1	1	1
0	0	0	1	0	1	1	1	1
0	0	1	0	0	1	1	1	1
0	0	1	1	0	1	1	1	1
0	1	0	0	0	1	1	1	1
0	1	0	1	1	0	1	1	1
0	1	1	0	1	1	0	1	1
0	1	1	1	1	1	1	0	1
1	0	0	0	1	1	1	1	0
1	0	0	1	1	1	1	1	0
1	0	1	0	1	1	1	1	0
1	0	1	1	1	1	1	1	0
1	1	0	0	1	1	1	1	1
1	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1

b) As equações de cada seletor, será:
$CS_1' = [A_{13}'A_{12}'(A_{11} + A_{10}) + A_{13}'A_{12}A_{11}'A_{10}']'$
$CS_2' = [A_{13}'A_{12}A_{11}'A_{10}]'$
$CS_3' = [A_{13}'A_{12}A_{11}A_{10}']'$
$CS_4' = [A_{13}'A_{12}A_{11}A_{10}]'$
$CS_5' = [A_{13}A_{12}']'$

$A_{13}A_{12}$	00	01	11	10
$A_{11}A_{10}$ 00	1	0	1	0
01	0	0	1	0
11	0	0	1	0
10	0	0	1	0

c) $CS'_{ROM} = CS'(A_{13}A_{12} + A_{13}'A_{12}'A_{11}'A_{10}')$

Exercício: Determinar para a faixa de endereços apresentada na tabela a seguir. Sabendo-se que o endereço inicial é igual a (0400)16, e o banco inicia pela memória de 4K inferior. Pede-se:
a) Indicar a faixa de endereços de cada CI no quadro abaixo.
b) A equação booleana de cada seletor de pastilha (CS = ativo com zero) cada CI.
c) A equação booleana do seletor do banco de memória (CS = ativo com zero). O decodificador de seleção é uma ROM com seleção de pastilha (CS = ativa com zero).

CI	Faixa de Endereço em Hex
1K	2C00 – 2FFF
1K	2800 – 2BFF
1K	2400 – 27FF
4K	1400 – 23FF
4K	0400 – 13FF

As linhas $A_{15} = A_{14} = 0$ e $A_0 – A_9$ = Linhas comuns.

A_{13}	A_{12}	A_{11}	A_{10}	CS_1	CS_2	CS_3	CS_4	CS_5
0	0	0	0	1	1	1	1	1
0	0	0	1	0	1	1	1	1
0	0	1	0	0	1	1	1	1
0	0	1	1	0	1	1	1	1
0	1	0	0	0	1	1	1	1
0	1	0	1	1	0	1	1	1
0	1	1	0	1	0	1	1	1
0	1	1	1	1	0	1	1	1
1	0	0	0	1	0	1	1	1
1	0	0	1	1	1	0	1	1
1	0	1	0	1	1	1	0	1
1	0	1	1	1	1	1	1	0
1	1	0	0	1	1	1	1	1
1	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1

b) As equações de cada seletor, será:
$CS_1' = [A_{13}'A_{12}'(A_{11} + A_{10}) + A_{13}'A_{12}A_{11}'A_{10}']'$
$CS_2' = [A_{13}'A_{12}(A_{11} + A_{10}) + A_{13}A_{12}'A_{11}'A_{10}']'$
$CS_3' = [A_{13}A_{12}'A_{11}'A_{10}]'$
$CS_4' = [A_{13}A_{12}'A_{11}A_{10}']'$
$CS_5' = [A_{13}A_{12}'A_{11}A_{10}]'$

c) $CS'_{ROM} = [CS' + (A_{13}A_{12} + A_{13}'A_{12}'A_{11}'A_{10}')]$

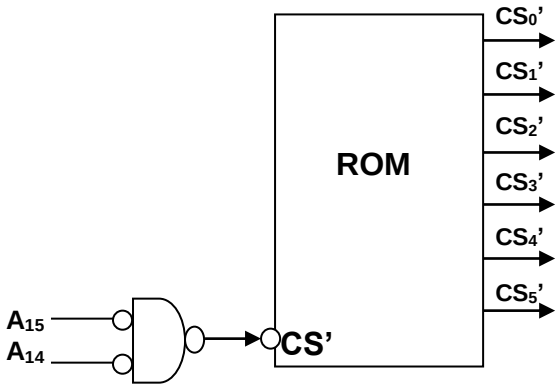
$CS'A_{13}A_{12}$	000	001	011	010	110	111	101	100
$A_{11}A_{10}$ 00	1	0	1	0	1	1	1	1
01	0	0	1	0	1	1	1	1
11	0	0	1	0	1	1	1	1
10	0	0	1	0	1	1	1	1

Exercício: Deseja-se construir um banco de memória cuja faixa é dada pelas equações booleanas das linhas de endereçamento, descritas abaixo. Pede-se:
a) O mapa da memória indicando a capacidade de cada chip de memória e sua disposição no banco de memória, indicando em hexadecimal endereço inicial e final de cada memória.
b) Implementação com ROM decodificadora, mostrando o mapa da ROM e sua capacidade.
c) Esquema da configuração do banco de memória, sabendo-se que as larguras dos chips são de 8 bits.

$\overline{CS_0} = (A_{15}' . A_{14}' . A_{13}' . A_{10}')'$;
$\overline{CS_1} = (A_{15}' . A_{14}' . A_{13}' . A_{12} . A_{11} . A_{10})'$;
$\overline{CS_2} = (A_{15}' . A_{14}' . A_{13} . A_{12}' . A_{11}')'$;
$\overline{CS_3} = (A_{15}' . A_{14}' . A_{13} . A_{12}' . A_{11})'$;
$\overline{CS_4} = (A_{15}' . A_{14}' . A_{13} . A_{12} . A_{10}')'$;
$CS_5 = (A_{15}' . A_{14}' . A_{13} . A_{12} . A_{11} . A_{10})'$;

Faixa de Endereços	Capacidade
CS_5' = 3C00 – 3FFF	1K
CS_4' = 3000 – 3BFF	2K*
CS_3' = 2800 – 2FFF	2K
CS_2' = 2000 – 27FF	2K
CS_1' = 1C00 – 1FFF	1K
CS_0' = 0000 – 1BFF	4K*

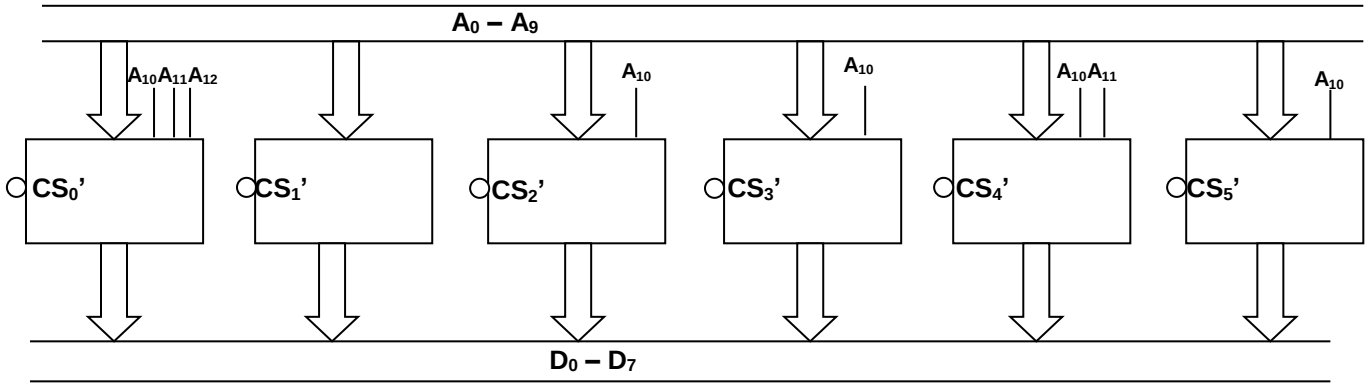
b) Mapa da ROM decodificadora.



A_{13}	A_{12}	A_{11}	A_{10}	CS_0'	CS_1'	CS_2'	CS_3'	CS_4'	CS_5'
0	0	0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1
0	0	1	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1
0	1	1	0	0	1	1	1	1	1
0	1	1	1	1	0	1	1	1	1
1	0	0	0	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1
1	0	1	0	1	1	1	0	1	1
1	0	1	1	1	1	1	0	1	1
1	1	0	0	1	1	1	1	0	1
1	1	0	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	0

Capacidade = 15 x 6, como 4 linhas não são utilizadas, daí C = 11 x 6.

*** Obs.:** Locações: 0400 – 07FF, 0C00 – 0FFF, 1C00 – 1FFF = 3K e 3400 – 37FF não estão acessíveis pela tabela da verdade, pois não são ativas na faixa de endereços, embora estejam dentro da faixa inicial e final.



Exercício: Uma associação de memória, construir um sistema de decodificação para um banco de memória é descrito conforme a seqüência a seguir. Pede-se:

a) Implementar o sistema completo de decodificação utilizando ROM, apresentando o mapa da ROM e identificando as entradas e saídas da ROM e com endereço e conteúdo e a sua capacidade mínima.
b) As equações do decodificador para o uso de PAL.

a) C = 16 X 4bits.

4 – 1,0K x 8
3 – 3,5K x 8
2 – 1,5K x 8
1 – 2,0K x 8

A_3	**A_2**	**A_1**	**A_0**	**B3**	**B2**	**B1**	**B0**	**-**	**-**
A12	**A11**	**A10**	**A9**	**CS_3**	**CS_2**	**CS_1**	**CS_0**	**End.**	**Cont**
0	0	0	0	1	1	1	0	**0**	**E**
0	0	0	1				0	**1**	**E**
0	0	1	0				0	**2**	**E**
0	0	1	1				0	**3**	**E**
0	1	0	0	1	1	0	1	**4**	**D**
0	1	0	1			0		**5**	**D**
0	1	1	0			0		**6**	**D**
0	1	1	1	1	0	1	1	**7**	**B**
1	0	0	0		0			**8**	**B**
1	0	0	1		0			**9**	**B**
1	0	1	0		0			**A**	**B**
1	0	1	1		0			**B**	**B**
1	1	0	0		0			**C**	**B**
1	1	0	1		0			**D**	**B**
1	1	1	0	0	1	1	1	**E**	**7**
1	1	1	1	0				**F**	**7**

b) $CS_0' = (A_{12}'A_{11}')'$
$CS_1' = (A_{12}'A_{11}A_{10}' + A_{12}'A_{11}A_9')'$
$CS_2' = (A_{12}A_{11}' + A_{12}A_{10}' + A_{12}'A_{11}A_{10}A_9)'$
$CS_3' = (A_{12}A_{11}A_{10})'$

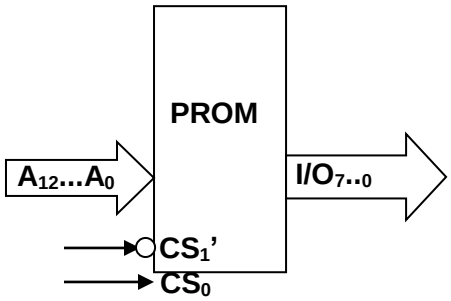
Exercício: Uma associação de memória, construir um sistema de decodificação para um banco de memória é descrito conforme a seqüência a seguir. Pede-se:
a) Implementar o sistema completo de decodificação utilizando ROM, apresentando o mapa da ROM e identificando as entradas e saídas da ROM e com endereço e conteúdo e a sua capacidade mínima.

b) As equações do decodificador para o uso de PAL.

4 – 2,0K x 8	A_3	A_2	A_1	A_0	B3	B2	B1	B0	-	-
3 – 1,5K x 8	A12	A11	A10	A9	CS_3	CS_2	CS_1	CS_0	End.	Cont
2 – 3,5K x 8	0	0	0	0	1	1	1	0	0	E
1 – 1,0K x 8	0	0	0	1				0	1	E
	0	0	1	0			0		2	D
	0	0	1	1			0		3	D
	0	1	0	0			0		4	D
	0	1	0	1			0		5	D
	0	1	1	0			0		6	D
	0	1	1	1			0		7	D
	1	0	0	0			0		8	D
	1	0	0	1		0			9	B
	1	0	1	0		0			A	B
	1	0	1	1		0			B	B
	1	1	0	0	0				C	7
	1	1	0	1	0				D	7
	1	1	1	0	0				E	7
	1	1	1	1	0				F	7

b) $CS_3' = (A_{12}A_{11})'$
$CS_2' = (A_{12}A_{11}' A_{10} + A_{12}A_{11}' A_9)'$
$CS_1' = (A_{12}' A_{11} + A_{12}'A_9 + A_{12}A_{11}'A_{10}'A_9')'$
$CS_0' = (A_{12}' A_{11}' A_{10}')'$

Exercício: Implementar um banco de memórias de 32K de PROM, usando lógica mínima de seleção (somente inversor) das memórias e partindo da memória a seguir. Pede-se:
a) Apresentar uma tabela da verdade da seleção das memórias.
a) Configuração das memórias.

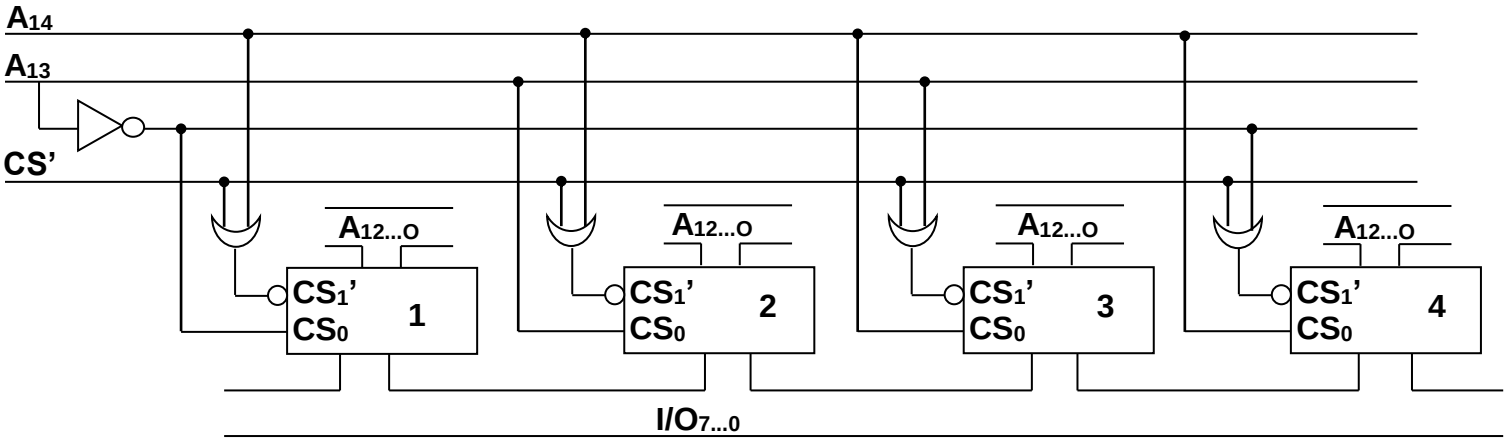


a) A tabela de endereços

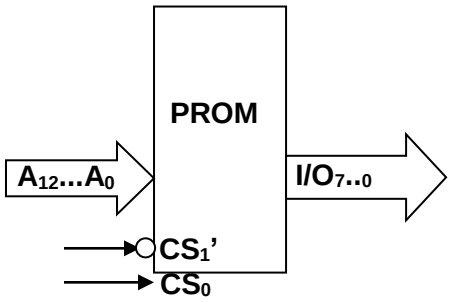
Memória	A_{14}	A_{13}	CS_1'	CS_0
1	0	0	A_{14}	A_{13}'
2	0	1	A_{14}	A_{13}
3	1	0	A_{13}	A_{14}
4	1	1	A_{13}'	A_{14}

Obs.: Existem outras possibilidades.

b) Configuração das memórias.



Exercício: Implementar um banco de memórias de 32K de PROM, usando lógica mínima de seleção (somente inversor) das memórias e partindo da memória a seguir. Pede-se:
a) Apresentar uma tabela da verdade da seleção das memórias.
a) Configuração das memórias.

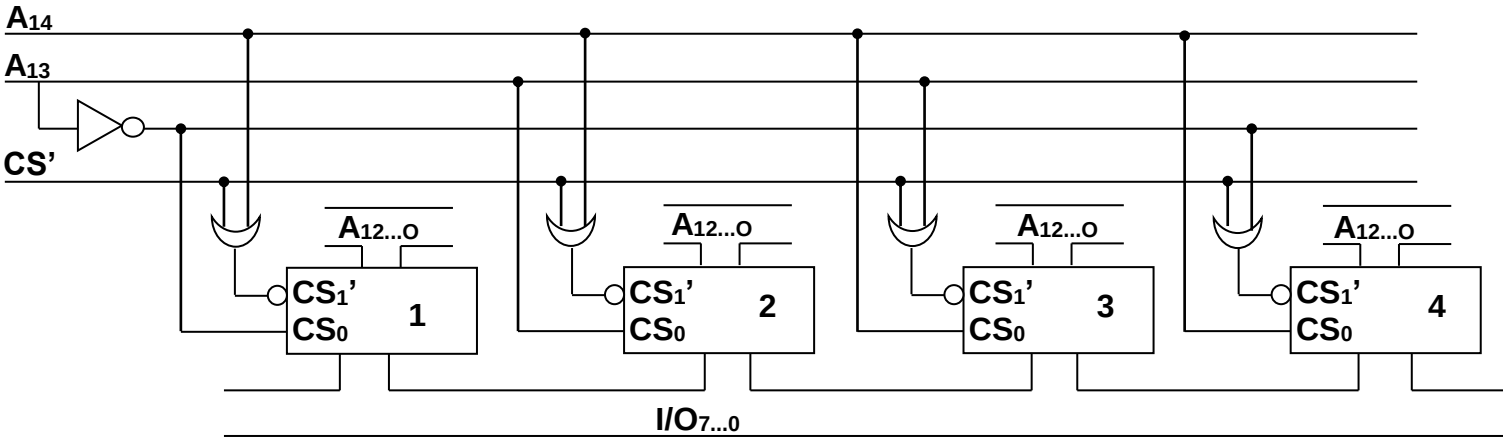


a) A tabela de endereços

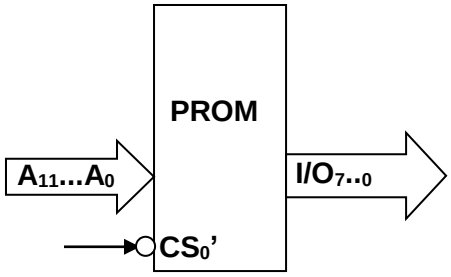
Memória	A_{14}	A_{13}	CS_1'	CS_0
1	0	0	A_{14}	A_{13}'
2	0	1	A_{14}	A_{13}
3	1	0	A_{13}	A_{14}
4	1	1	A_{13}'	A_{14}

Obs.: Existem outras possibilidades.

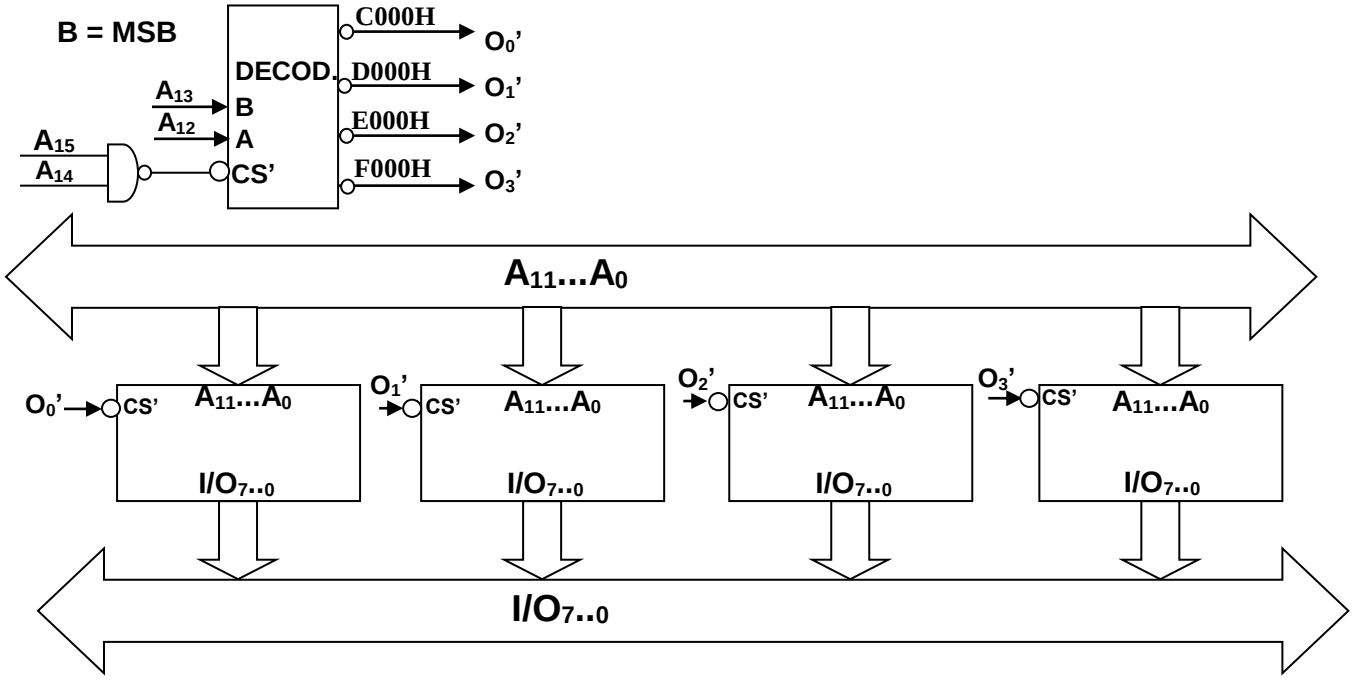
b) Configuração das memórias.



Exercício: Implementar um banco de memórias de 16K de PROM, usando decodificador de endereços, saída lógica negativa, para a seleção das memórias e usando a memória a seguir. O endereço inicial das EPROMs ocorre em C000H. Pede-se:
a) Apresentar uma tabela da verdade da seleção das memórias.
b) Configuração das memórias.
c) Usando o sistema de decodificação por PAL.



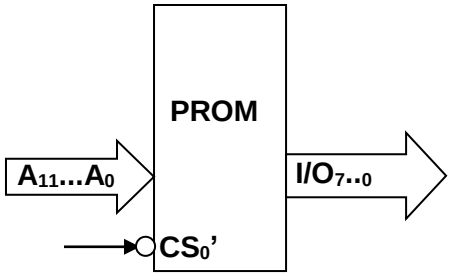
A_{13}	A_{12}	MEM
0	0	1
0	1	2
1	0	3
1	1	4



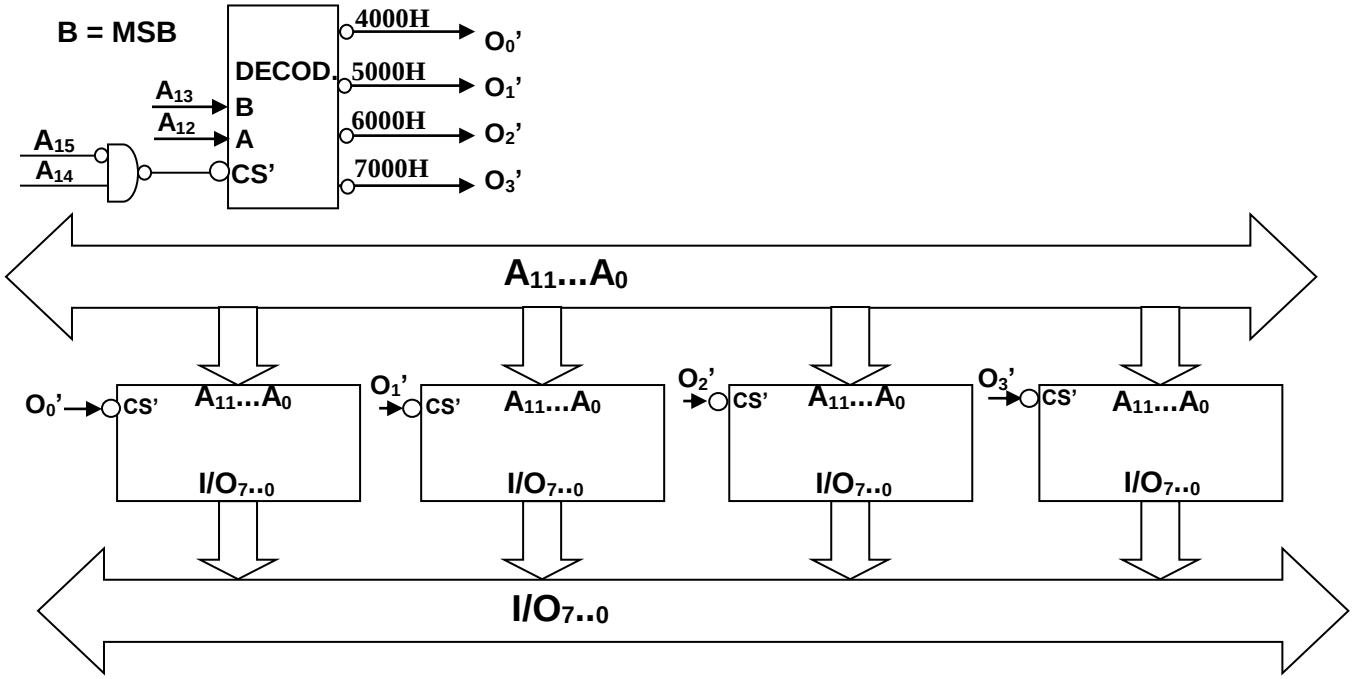
c) Implementação com PAL, será :

$O_0' = (A_{15} \cdot A_{14} \cdot /A_{13} \cdot /A_{12})'$.
$O_1' = (A_{15} \cdot A_{14} \cdot /A_{13} \cdot A_{12})'$.
$O_2' = (A_{15} \cdot A_{14} \cdot A_{13} \cdot /A_{12})'$.
$O_3' = (A_{15} \cdot A_{14} \cdot A_{13} \cdot A_{12})'$.

Exercício: Implementar um banco de memórias de 16K de PROM, usando decodificador de endereços saída lógica negativa, para seleção das memórias e partindo da memória a seguir. O endereço inicial das EPROMs ocorre em 4000H. Pede-se:
a) Apresentar uma tabela da verdade da seleção das memórias.
b) Configuração das memórias.
c) Usando o sistema de decodificação por PAL.



A_{13}	A_{12}	MEM
0	0	1
0	1	2
1	0	3
1	1	4



c) Implementação com PAL, será :

$O_0' = (A_{15}' \,.\, A_{14} \,./A_{13} \,./A_{12})'$.
$O_1' = (A_{15}' \,.\, A_{14} \,./A_{13} \,.\, A_{12})'$.
$O_2' = (A_{15}' \,.\, A_{14} \,.\, A_{13} \,./A_{12})'$.
$O_3' = (A_{15}' \,.\, A_{14} \,.\, A_{13} \,.\, A_{12})'$.

Exercício: Desenhe o diagrama completo para uma memória de 256K x 8 que usa chips RAM com as seguintes especificações: capacidade de 64K x 4, linhas comuns de entrada / saídas e duas entradas para a seleção do chip ativas em NL0. Utilizar no circuito de seleção lógica de decodificação a inversores.

a) O número de memórias.

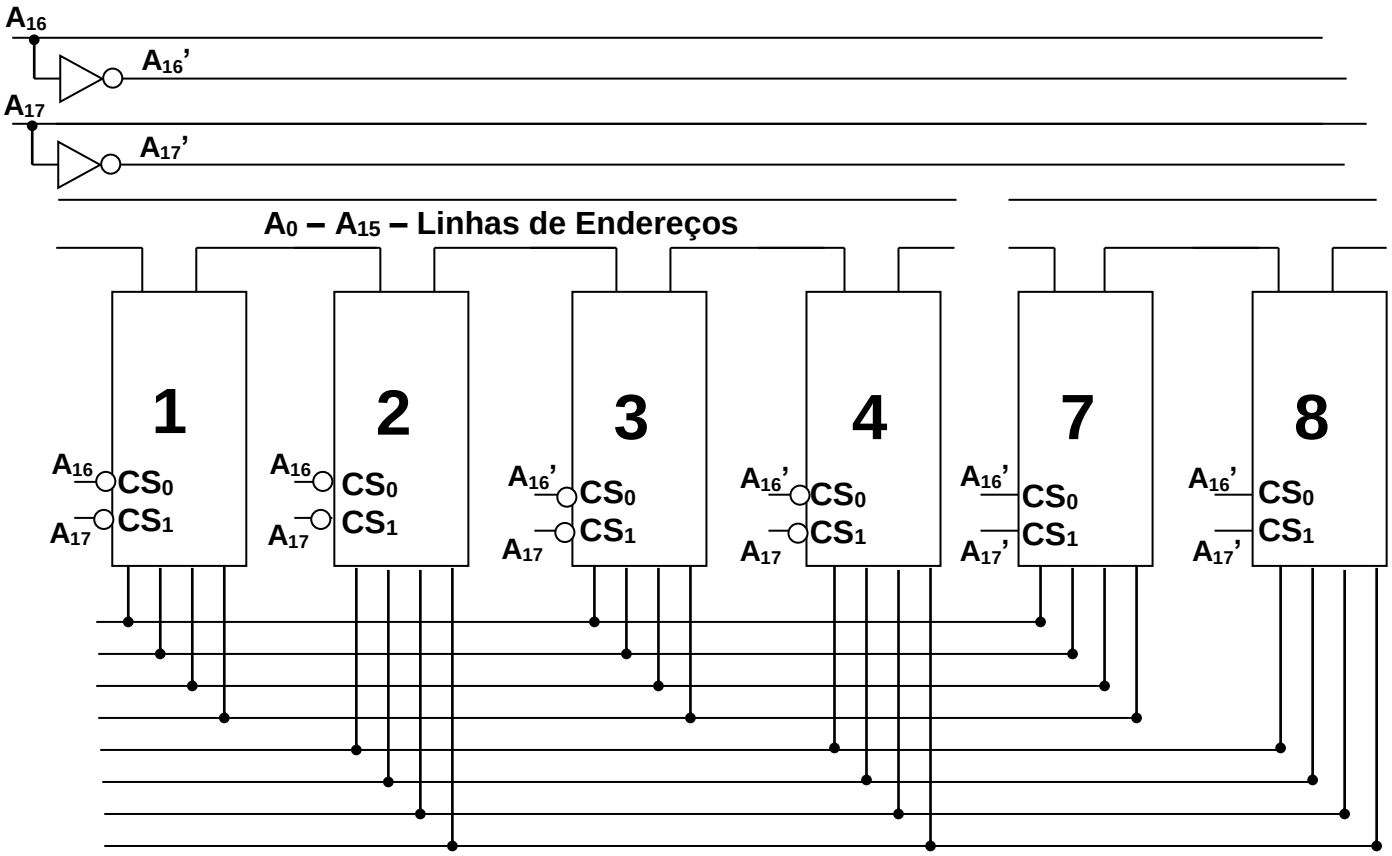
b) O circuito de seleção com os inversores.

c) A configuração do banco das memórias com interligações entrada/saída, seleção e comandos de leitura e escrita.

Solução:

a) Número de memórias = total do banco / chip de partida = 256k x 8 / 64K x 4 = 08 memórias.

b) e c) Conforme abaixo.



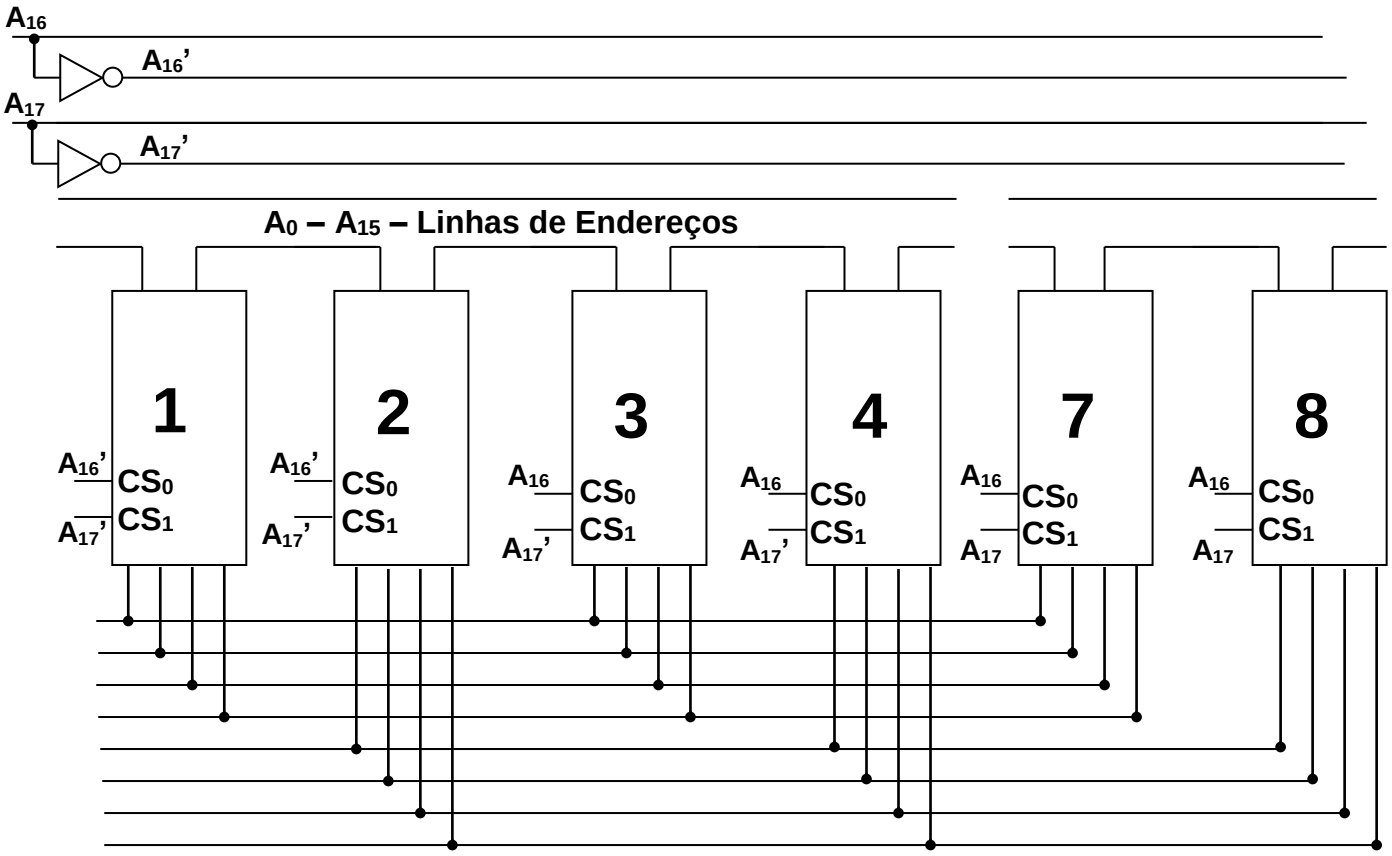
A_{17}	A_{16}	CS_0	CS_1	Memórias
0	0	A_{17}	A_{16}	1 e 2
0	1	A_{17}	A_{16}'	3 e 4
1	0	A_{17}'	A_{16}	5 e 6
1	1	A_{17}'	A_{16}'	7 e 8

Exercício: Desenhe o diagrama completo para uma memória de 256K x 8 que usa chips RAM com as seguintes especificações: capacidade de 64K x 4, linhas comuns de entrada / saídas e duas entradas para a seleção do chip ativas em NL1. Utilizar no circuito de seleção lógica de decodificação a inversores.

a) O número de memórias.
b) O circuito de seleção com os inversores.
c) A configuração do banco das memórias com interligações entrada/saída, seleção e comandos de leitura e escrita.

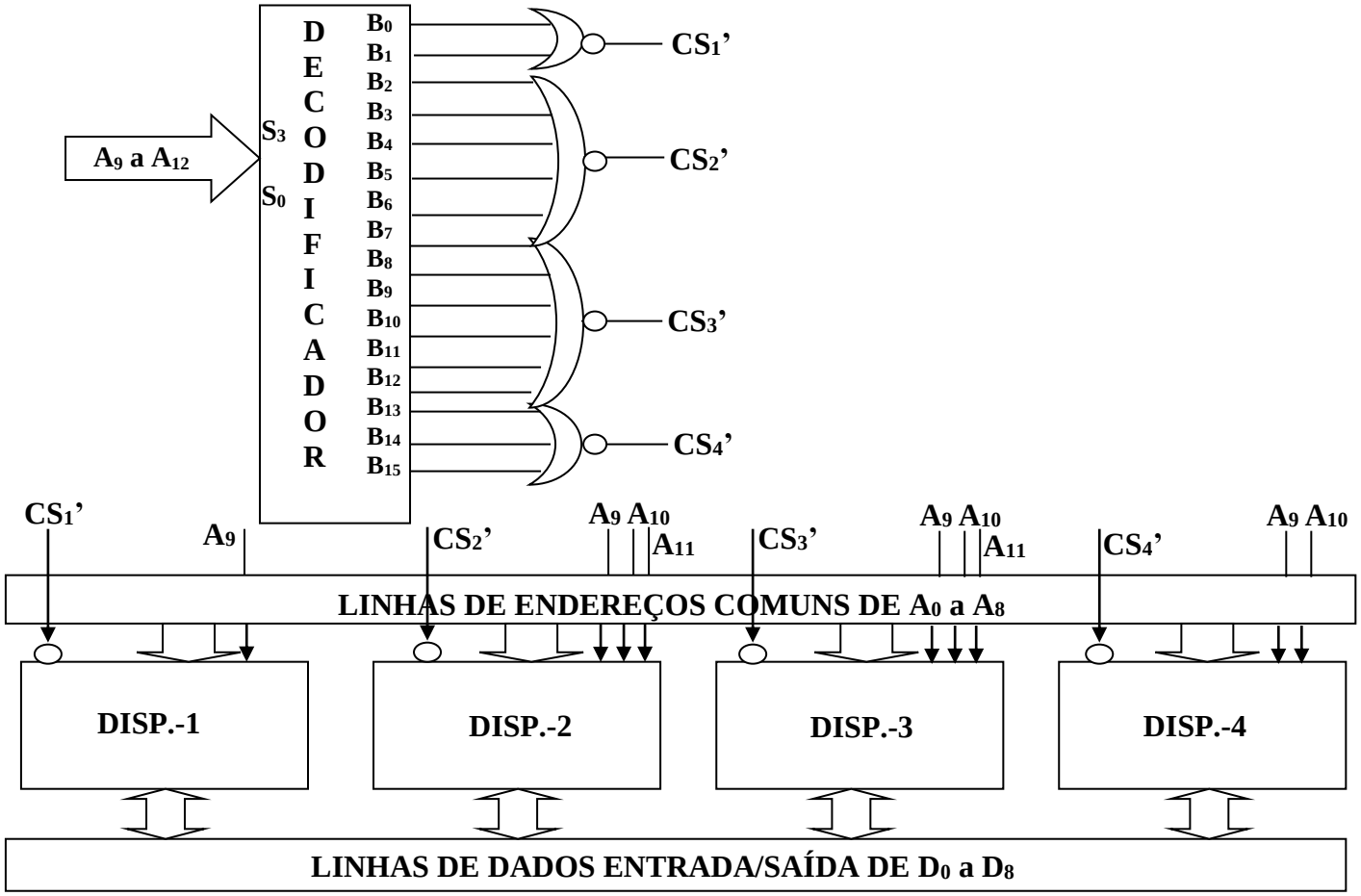
Solução:

a) Número de memórias = total do banco / chip de partida = 256k x 8 / 64K x 4 = 08 memórias.
b) e c) Conforme abaixo.



A_{17}	A_{16}	CS_0	CS_1	Memórias
0	0	A_{17}'	A_{16}'	1 e 2
0	1	A_{17}'	A_{16}	3 e 4
1	0	A_{17}	A_{16}'	5 e 6
1	1	A_{17}	A_{16}	7 e 8

Exercício: Para a associação de memória mostrada a seguir, pede-se:
a) Capacidade total do banco de memória.
b) Faixa de endereços de cada dispositivo de memória.
c) O mapa da memória ROM usada no lugar do decodificador de endereços e a capacidade da ROM.



a) Capacidade é de 8K x 8.

b) A faixa de endereços

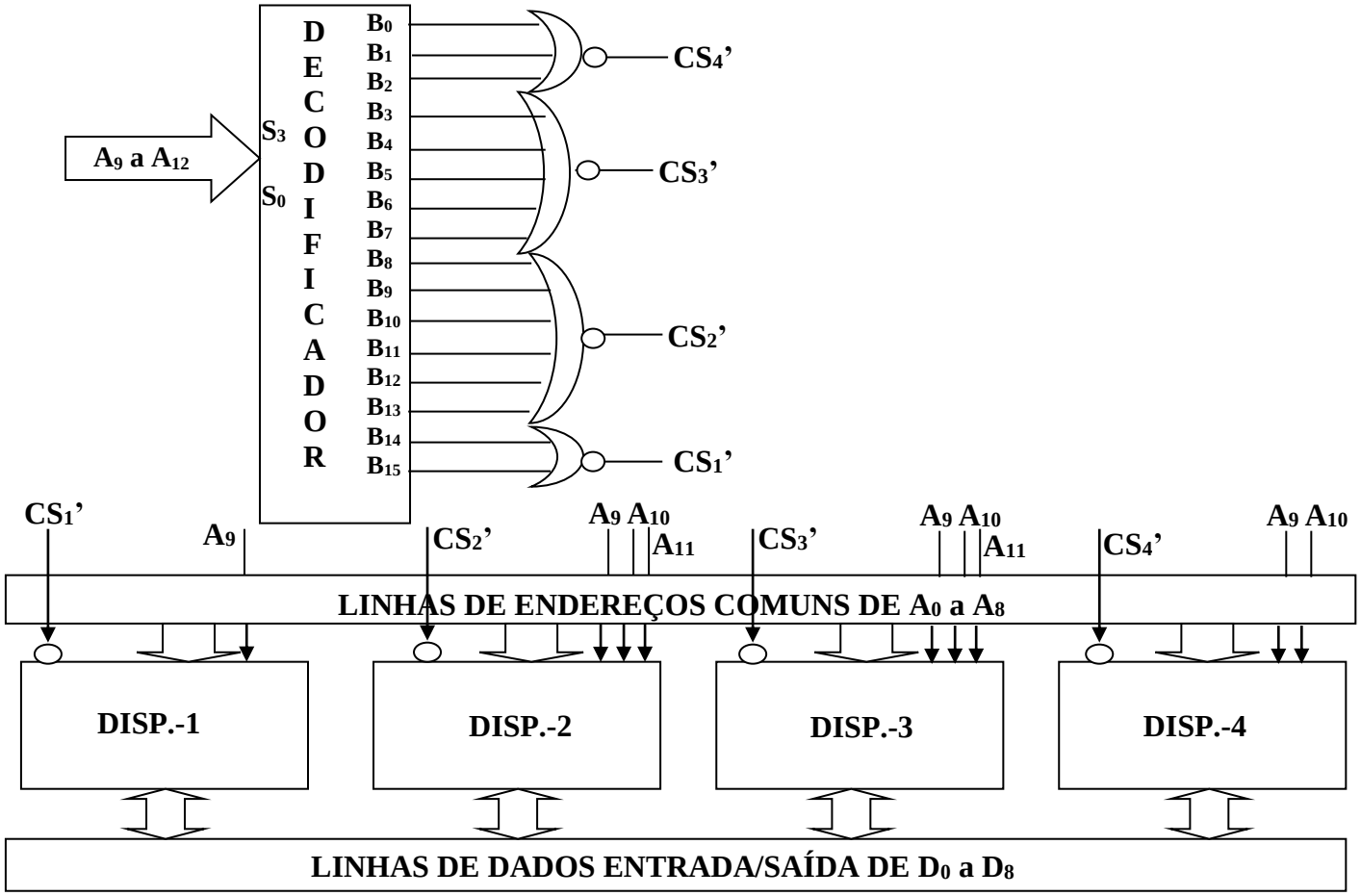
Dispostivos	FAIXA DE ENDEREÇOS EM HEXADECIMAL
Dispostivo 1	0000 a 03FF – 0 a 1023
Dispostivo 2	0400 a 0DFF – 1024 a 3583
Dispostivo 3	0E00 a 19FF – 3584 a 6655
Dispostivo 4	1A00 a 1FFF – 6656 a 8191

c) Mapa da ROM

A_{12}	A_{11}	A_{10}	A_9	CS_4'	CS_3'	CS_2'	CS_1'
0	0	0	0	1	1	1	0
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	1
0	1	0	1	1	1	0	1
0	1	1	0	1	1	0	1
0	1	1	1	1	0	1	1
1	0	0	0	1	0	1	1
1	0	0	1	1	0	1	1
1	0	1	0	1	0	1	1
1	0	1	1	1	0	1	1
1	1	0	0	1	0	1	1
1	1	0	1	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1

A capacidade da ROM é C = 16 x 4

Exercício: Para a associação de memória mostrada a seguir, pede-se:
a) Capacidade total do banco de memória.
b) Faixa de endereços de cada dispositivo de memória.
c) O mapa da memória ROM usada no lugar do decodificador de endereços e a capacidade da ROM.



a) C = 8192 X 9 bits = 8K x 9.

b) A faixa de endereços em hexadecimal

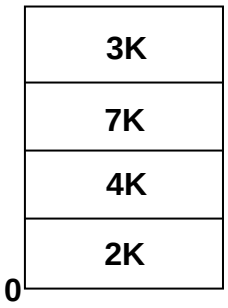
Dispostivos	FAIXA DE ENDEREÇOS EM HEXADECIMAL
Dispostivo 1	1C00 – 1FFF
Dispostivo 2	1000 – 1BFF
Dispostivo 3	0600 – 0FFF
Dispostivo 4	0000 – 05FF

c) Mapa da ROM

A_{12}	A_{11}	A_{10}	A_9	CS_4'	CS_3'	CS_2'	CS_1'
0	0	0	0	0	1	1	1
0	0	0	1	0	1	1	1
0	0	1	0	0	1	1	1
0	0	1	1	1	0	1	1
0	1	0	0	1	0	1	1
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1
0	1	1	1	1	0	1	1
1	0	0	0	1	1	0	1
1	0	0	1	1	1	0	1
1	0	1	0	1	1	0	1
1	0	1	1	1	1	0	1
1	1	0	0	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	0
1	1	1	1	1	1	1	0

Exercício: De acordo com o mapa a seguir, um banco de memória de capacidade 16K x 8. Pede-se:

a) O projeto do decodificador realizado com memória ROM. Tabela de endereços e conteúdos, sabendo-se que cada dispositivo dispõe de um seletor de chip CS'_i , onde i = 0 a 3.



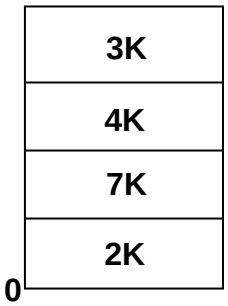
A_3,A_2,A_1,A_0 = Endereços da ROM.
Como não existem multiplos comuns na capacidade da ROM, escolhe-se um múltiplo comum a todas memórias igual a 1K.
Daí cada dispositivo terá uma multiplicidade de 1K.
A tabela da ROM decodificada, será:

A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0
0	0	0	0	1	1	1	0
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	1
0	1	0	1	1	1	0	1
0	1	1	0	1	0	1	1
0	1	1	1	1	0	1	1
1	0	0	0	1	0	1	1
1	0	0	1	1	0	1	1
1	0	1	0	1	0	1	1
1	0	1	1	1	0	1	1
1	1	0	0	1	0	1	1
1	1	0	1	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1

a) Considerando comum 1K, temos A_0 a A_9 comum a todas as memórias. Onde o banco de memória é:
16K = A_0 – A_{13}, onde A_{10} = A_0, A_{11} = A_1, A_{12} = A_2, A_{13} = A_3.
B_3 = CS'_3 e B_2 = CS'_2 e B_1 = CS'_1 e B_0 = CS'_0.
3K = CS'_3; 7K = CS'_2; 4K = CS'_1 e 2K = CS'_0.

Exercício: De acordo com o mapa a seguir, um banco de memória de capacidade 16K x 8. Pede-se:

a) O projeto do decodificador realizado com memória ROM. Tabela de endereços e conteúdos, sabendo-se que cada dispositivo dispõe de um seletor de chip CS'_i , onde i = 0 a 3.



a) Considerando comum 1K, temos A_0 a A_9 comum a todas as memórias. Onde o banco de memória é :
16K = $A_0 – A_{13}$, onde $A_{10} = A_0$, $A_{11} = A_1$, $A_{12} = A_2$, $A_{13} = A_3$.
$B_3 = CS'_3$ e $B_2 = CS'_2$ e $B_1 = CS'_1$ e $B_0 = CS'_0$.
3K = CS'_3; 4K = CS'_2; 7K = CS'_1 e 2K = CS'_0.
A tabela da ROM decodificadora, será :

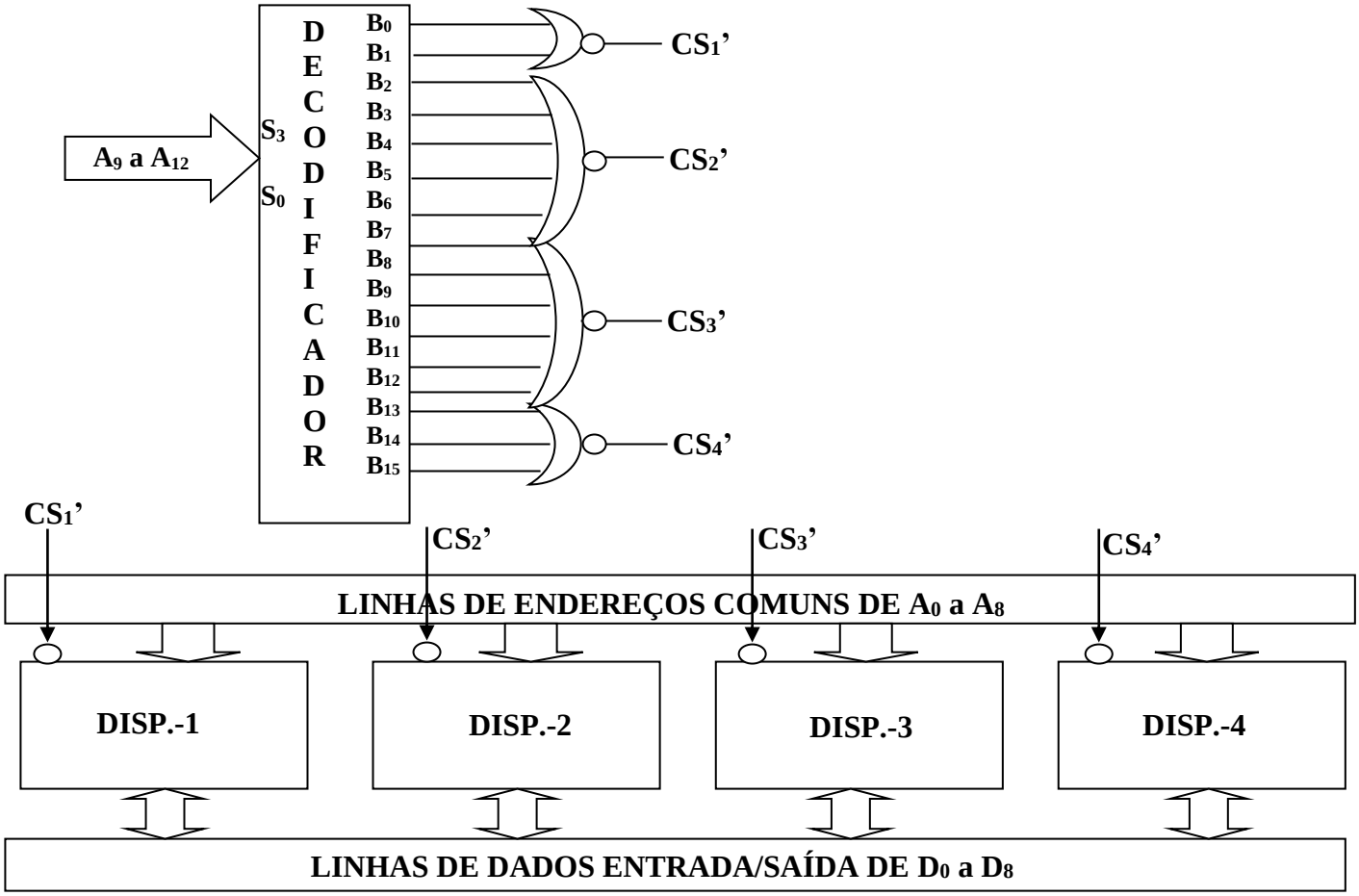
A_3	A_2	A_1	A_0	B_3	B_2	B_1	B_0
0	0	0	0	1	1	1	0
0	0	0	1	1	1	1	0
0	0	1	0	1	1	0	1
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	1
0	1	0	1	1	1	0	1
0	1	1	0	1	1	0	1
0	1	1	1	1	1	0	1
1	0	0	0	1	1	0	1
1	0	0	1	1	0	1	1
1	0	1	0	1	0	1	1
1	0	1	1	1	0	1	1
1	1	0	0	1	0	1	1
1	1	0	1	0	1	1	1
1	1	1	0	0	1	1	1
1	1	1	1	0	1	1	1

Exercício: Construir um banco de memória, conforme distribuição abaixo, utilizando-se na seleção dos dispositivos, um decodificador comum de 4 variáveis de seleção S_3,S_2,S_1,S_0, saída lógica positiva. Pede-se :
a) Faixa de endereços de operação de cada dispositivo.
b) O diagrama de ligações do decodificador e os dispositivos.

Dispositivo 4 - 1,5K
Dispositivo 3 – 3,0K
Dispositivo 2 – 2,5K
Dispositivo 1 – 1,0K

a) A faixa de operação dos dispositivos.

Dispostivos	FAIXA DE ENDEREÇOS EM HEX. E DEC
Dispostivo 1	**0000 a 03FF – 0 a 1023**
Dispostivo 2	**0400 a 0DFF – 1024 a 3583**
Dispostivo 3	**0E00 a 19FF – 3584 a 6655**
Dispostivo 4	**1A00 a 1FFF – 6656 a 8191**


A9 a A12
S3
S0
DECODIFICADOR
B0
B1
B2
B3
B4
B5
B6
B7
B8
B9
B10
B11
B12
B13
B14
B15
CS1'
CS2'
CS3'
CS4'
CS1'
CS2'
CS3'
CS4'
LINHAS DE ENDEREÇOS COMUNS DE A0 a A8
DISP.-1
DISP.-2
DISP.-3
DISP.-4
LINHAS DE DADOS ENTRADA/SAÍDA DE D0 a D8

Exercício: Construir um banco de memória, conforme distribuição abaixo, utilizando-se na seleção dos dispositivos, um decodificador comum de 4 variáveis de seleção S_3,S_2,S_1,S_0, saída lógica positiva. Pede-se:

a) Faixa de endereços de operação de cada dispositivo.

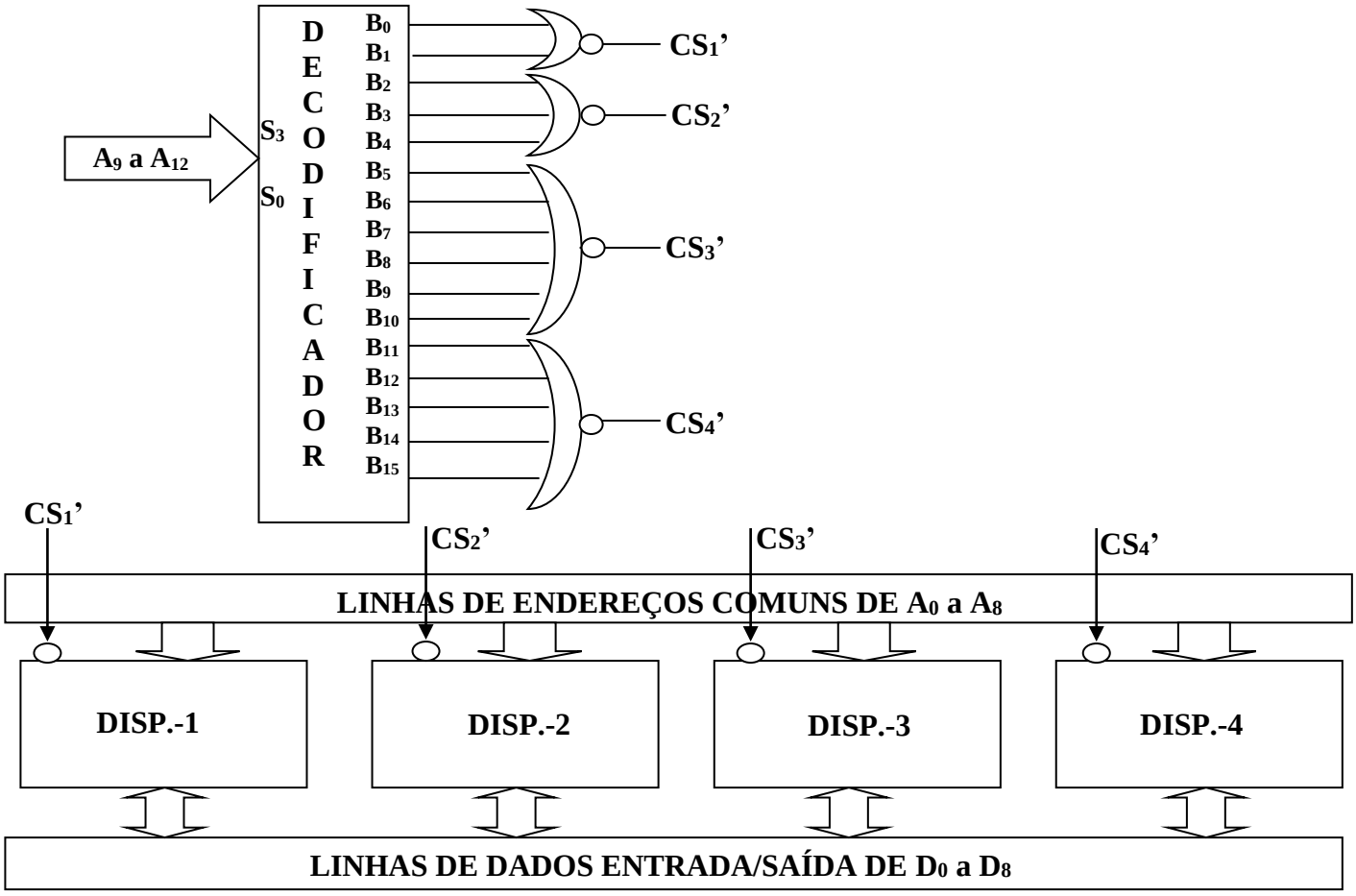
b) O diagrama de ligações do decodificador e os dispositivos.

Dispositivo 4 - 2,5K
Dispositivo 3 – 3,0K
Dispositivo 2 – 1,5K
Dispositivo 1 – 1,0K

0

a) A faixa de operação dos dispositivos.

Dispostivos	FAIXA DE ENDEREÇOS EM HEX. E DEC
Dispostivo 1	0000 a 03FF – 0 a 1023
Dispostivo 2	0400 a 09FF – 1024 a 2559
Dispostivo 3	0A00 a 15FF – 2560 a 5631
Dispostivo 4	1600 a 1FFF – 5632 a 8191



Exercício: Um sistema de decodificação para um banco de memória é descrito conforme a seqüência a seguir. Pede-se:
a) Utilizando um decodificador tradicional de 4 entradas A,B,C e D(MSB) e com 16 saídas, sendo as suas saídas por lógica negativa (Ativa com zero) e circuito lógico adicional se necessário.
b) Implementar o sistema completo de decodificação utilizando ROM, apresentando o mapa da ROM e identificando as entradas e saídas da ROM e a sua capacidade.

4095

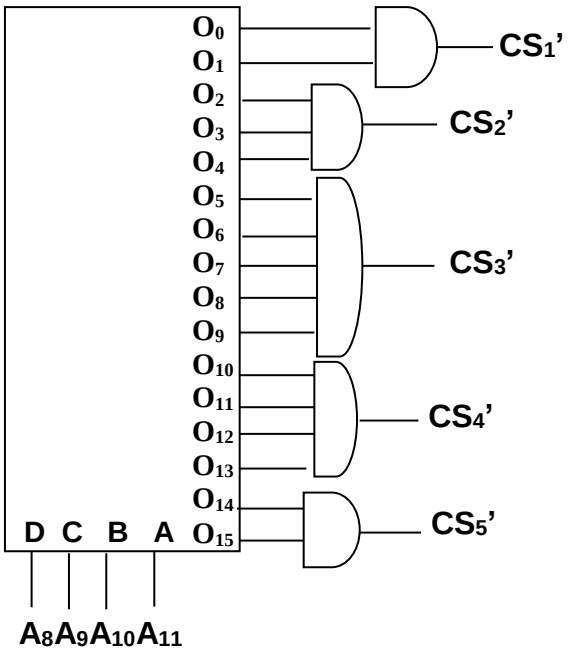
5 – 512 x 8
4 – 1024 x 8
3 – 1280 x 8
2 – 768 x 8
1 – 512 x 8

0

Como o decodificador tem 16 linhas de saídas, então cada linha deve selecionar:

$$\text{Por linha} = \frac{4096}{16} = 256$$

a) O sistema de decodificação será: Saída O_i' e de A_0 a A_7 comum a todas memórias.



b) Mapa da ROM = 16 x 5 bits

A_{11}	A_{10}	A_9	A_8	CS_5	CS_4	CS_3	CS_2	CS_1
A_3	A_2	A_1	A_0	B_4	B_3	B_2	B_1	B_0
0	0	0	0	1	1	1	1	0
0	0	0	1	1	1	1	1	0
0	0	1	0	1	1	1	0	1
0	0	1	1	1	1	1	0	1
0	1	0	0	1	1	1	0	1
0	1	0	1	1	1	0	1	1
0	1	1	0	1	1	0	1	1
0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	0	1	1
1	0	0	1	1	1	0	1	1
1	0	1	0	1	0	1	1	1
1	0	1	1	1	0	1	1	1
1	1	0	0	1	0	1	1	1
1	1	0	1	1	0	1	1	1
1	1	1	0	0	1	1	1	1
1	1	1	1	0	1	1	1	1

EXERCÍCIOS DE MEMÓRIAS

1. Construir um sistema de decodificação para a seguinte distribuição. Pede-se:
a) O projeto do decodificador implementado com ROM como decodificador.
b) O projeto utilizando decodificadores de 64 x 9 com saídas lógica negativa.

Dado o mapa de distribuição da memória.

42K	Dispositivo 9 - 1K
	Dispositivo 8 - 2K
	Dispositivo 7 - 4K
	Dispositivo 6 - 4K
	Dispositivo 5 - 8K
	Dispositivo 4 –16K
	Dispositivo 3 - 4K
	Dispositivo 2 - 2K
0	Dispositivo 1 - 1K

Solução:

a) Com ROM

Vamos considerar para o projeto do decodificador sempre o menor módulo de memória ou a menor capacidade em endereços da memória.

A menor capacidade de memória é de 1K.

- Número de linhas iguais a 1K na ROM será igual a capacidade ocupada pelos dispositivos de 1 a 9 ou seja 42 linhas ou 42 endereços da ROM;
- As linhas de endereços comuns aos 9 dispositivos são 10 de A_0 a A_9;
- Número de bits que endereçam as 42 linhas é igual a 6;
- As linhas de endereços da ROM vão de A_{10} a A_{15};
- Número de saídas da ROM é igual ao total de dispositivos B_0 a B_8.

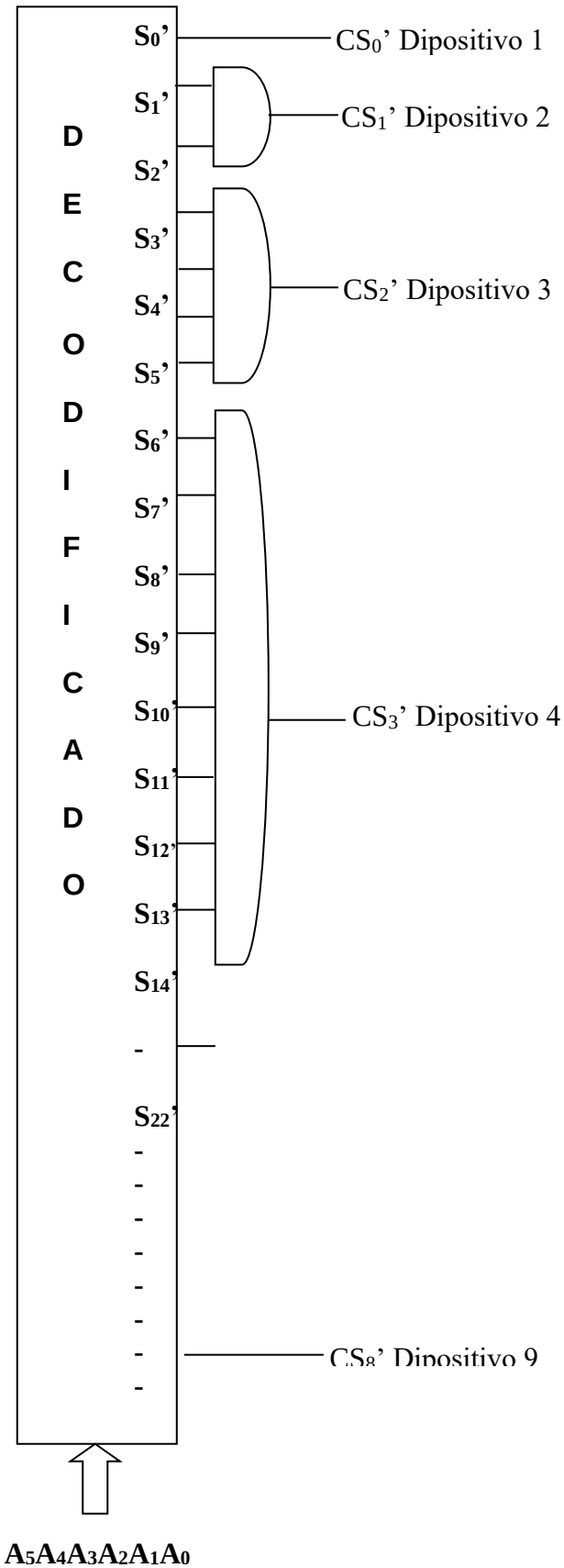
TABELA DA MEMÓRIA ROM

ENDEREÇOS							CONTEÚDO							
A_5	A_4	A_3	A_2	A_1	A_0	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	1	1	0	1	1	1	1	1	1	1
0	0	0	0	1	0	1	0	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	0	1	1	1	1	1	1
0	0	0	1	0	0	1	1	0	1	1	1	1	1	1
0	0	0	1	0	1	1	1	0	1	1	1	1	1	1
0	0	0	1	1	0	1	1	0	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	0	1	1	1	1	1
0	0	1	0	0	0	1	1	1	0	1	1	1	1	1
0	0	1	0	0	1	1	1	1	0	1	1	1	1	1
0	0	1	0	1	0	1	1	1	0	1	1	1	1	1
0	0	1	0	1	1	1	1	1	0	1	1	1	1	1
0	0	1	1	0	0	1	1	1	0	1	1	1	1	1
0	0	1	1	0	1	1	1	1	0	1	1	1	1	1
0	0	1	1	1	0	1	1	1	0	1	1	1	1	1
0	0	1	1	1	1	1	1	1	0	1	1	1	1	1
0	1	0	0	0	0	1	1	1	0	1	1	1	1	1
0	1	0	0	0	1	1	1	1	0	1	1	1	1	1
0	1	0	0	1	0	1	1	1	0	1	1	1	1	1
0	1	0	0	1	1	1	1	1	0	1	1	1	1	1
0	1	0	1	0	0	1	1	1	0	1	1	1	1	1
0	1	0	1	0	1	1	1	1	0	1	1	1	1	1
0	1	0	1	1	0	1	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	1	1	0	1	1	1	1
0	1	1	0	0	0	1	1	1	1	0	1	1	1	1
0	1	1	0	0	1	1	1	1	1	0	1	1	1	1
0	1	1	0	1	0	1	1	1	1	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1	1	1
0	1	1	1	0	0	1	1	1	1	0	1	1	1	1
0	1	1	1	0	1	1	1	1	1	0	1	1	1	1
0	1	1	1	1	0	1	1	1	1	0	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	0	1	1	1
1	0	0	0	0	0	1	1	1	1	1	0	1	1	1
1	0	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1	1
1	0	0	0	1	1	1	1	1	1	1	1	0	1	1
1	0	0	1	0	0	1	1	1	1	1	1	0	1	1
1	0	0	1	0	1	1	1	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1	1	1	1	1	0	1	1
1	0	0	1	1	1	1	1	1	1	1	1	1	0	1
1	0	1	0	0	0	1	1	1	1	1	1	1	0	1
1	0	1	0	0	1	1	1	1	1	1	1	1	1	0

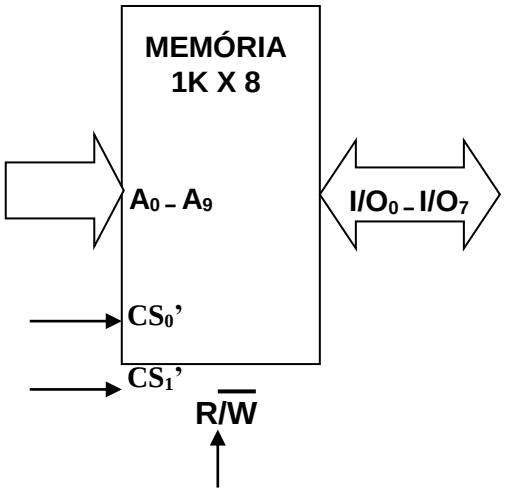
As equações booleanas de cada saída da memória ROM são:

$B_0 = A_0'A_1' \ldots\ldots A_5'$
$B_1 = A_0A_1'A_2'\ldots.A_5' + A_0'A_1A_2'\ldots.A_5'$
$B_2 = A_0A_1A_2'\ldots.A_5' + A_0'A_1'A_2A_3'\ldots A_5' + A_0A_1'A_2\,A_3'\ldots A_5' + A_0'A_1A_2A_3'\ldots A_5'$
B_3 = 16 produtosbooleanos
B_4 = 8 produtos booleanos
B_5 = 4 produtos booleanos
B_6 = 4 produtos booleanos
B_7 = 2 produtos booleanos
$B_8 = A_0A_1'A_2'A_3A_4'A_5$

b) COM DECODIFICADOR



2. Para o chip de memória a seguir construir um banco de memória 4K x 8, utilizando a forma mais econômica para a lógica de seleção dos chips.



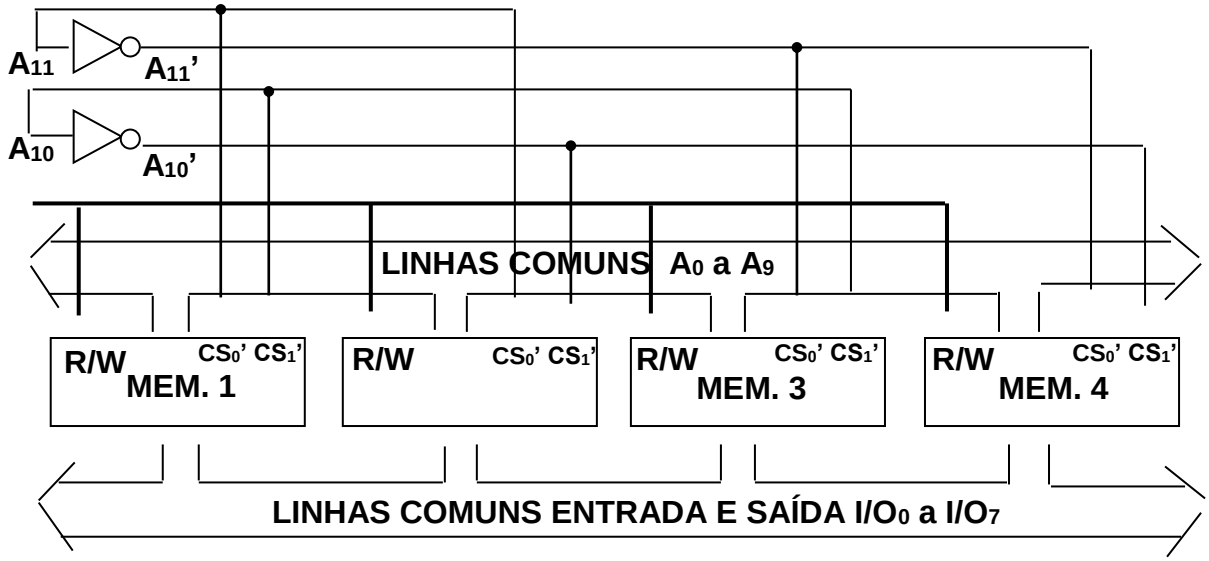
Solução :

Para 4K a capacidade de endereço aumentou, daí são necessárias para acesso à memória 12 linhas de A_0 a A_{11}. As linhas A_0 até A_9 são comuns aos dispositivos de memória.

$$\text{Número de memórias} = \frac{\text{Capacidade do banco}}{\text{Capacidade do dispostivo}} = \frac{4K}{1K} = 4 \text{ dispositivos}$$

END.		CHIP 1		CHIP 2		CHIP 3		CHIP 4	
A_{11}	A_{10}	CS_0'	CS_1'	CS_0'	CS_1'	CS_0'	CS_1'	CS_0'	CS_1'
0	0	A_{11}	A_{10}						
0	1			A_{11}	A_{10}'				
1	0					A_{11}'	A_{10}		
1	1							A_{11}'	A_{10}'

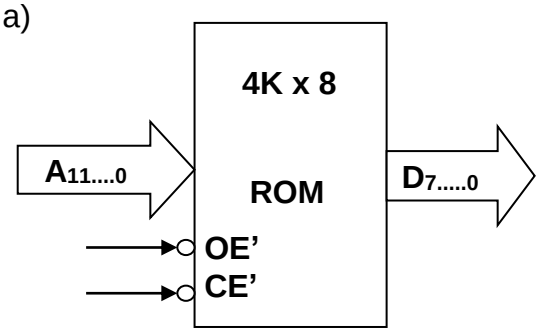
Utilizando-se de 02 inversores, gera-se A_{10}' e A_{11}'.



3. Deseja-se expandir a capacidade da memória ROM para 32K x 8bits. A memória ROM de estoque é de capacidade de 4K x 8. No mapa de memória o endereço inicial das memórias ROM é 8000H. Pede-se:
a) O sistema de decodificação para a faixa indicada no enunciado.
b) Implementação do sistema usando PAL.
c) Implementação do sistema usando ROM.

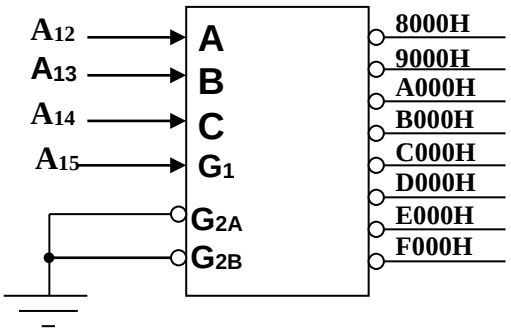
Solução: A memória ROM de 4K x 8 é mostrada a seguir.

a)



Endereço	ROM
8000H	8FFFH
9000H	9FFFH
A000H	AFFFH
B000H	BFFFH
C000H	CFFFH
D000H	DFFFH
E000H	EFFFH
F000H	FFFFH

O sistema de decodificação será:



b) Com PAL.

8000H' = A_{15} . A_{14}' . A_{13}' . A_{12}'.
9000H' = A_{15} . A_{14}' . A_{13}' . A_{12}.
A000H' = A_{15} . A_{14}' . A_{13} . A_{12}'.
B000H' = A_{15} . A_{14}' . A_{13} . A_{12}.
C000H' = A_{15} . A_{14} . A_{13}'. A_{12}'.
D000H' = A_{15} . A_{14} . A_{13}'. A_{12}.
E000H' = A_{15} . A_{14} . A_{13} . A_{12}'.
F000H' = A_{15} . A_{14} . A_{13} . A_{12}.

c) Com ROM.
O mapa da ROM :

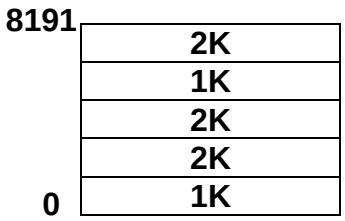
A_2	A_1	A_0	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	0	0								0
0	0	1							0	
0	1	0						0		
0	1	1					0			
1	0	0				0				
1	0	1			0					
1	1	0		0						
1	1	1	0							

A_2 = A_{14}
A_1 = A_{13}
A_0 = A_{12}
B_7 = 8000H'
B_6 = 9000H'
B_5 = A000H'
B_4 = B000H'
B_3 = C000H'
B_2 = D000H'
B_1 = E000H'
B_0 = F000H'

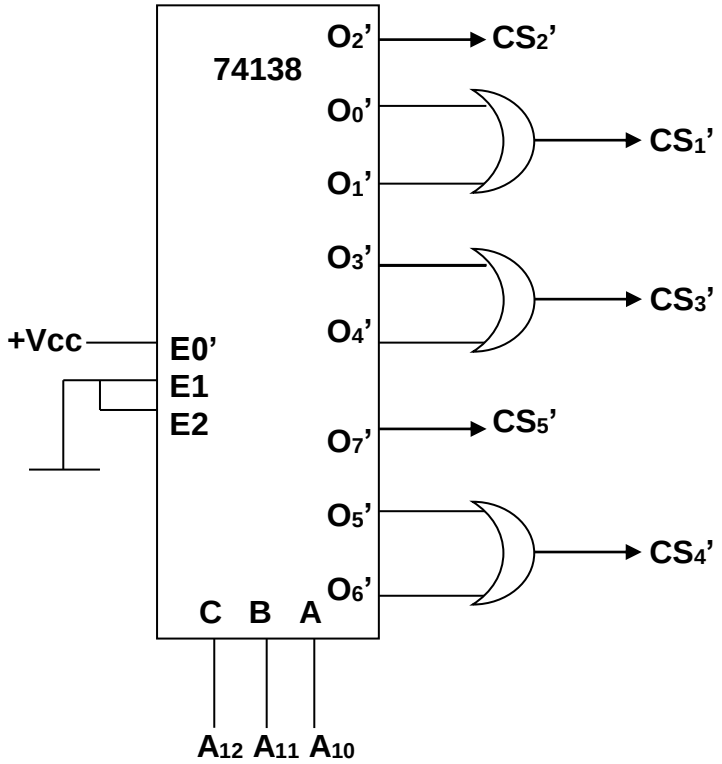
Ligar CS' = A_{15}

1. Construir um circuito de decodificação para um banco de memória conforme disposição a seguir. Pede-se:

a) Utilizando o CI- decodificador 74138 mais uma lógica adicional, esboçar o circuito de decodificação.
b) Implementar o circuito de decodificação usando memória ROM como decodificador.



4) Construção de um decodificador para o banco de 8K. Considerar como se fossem 8 dispositivos de 1K pois 1K é a memória de menor capacidade.

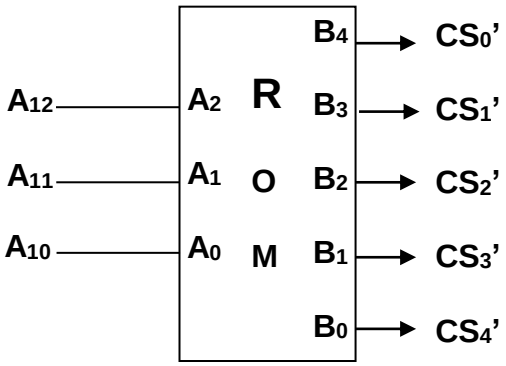


As linhas de endereços para 8K : 13 Linhas $A_0 – A_{12}$
As linhas para 1K são : 10 linhas – $A_0 – A_9$
Sobraram A_{10}, A_{11} e A_{12}

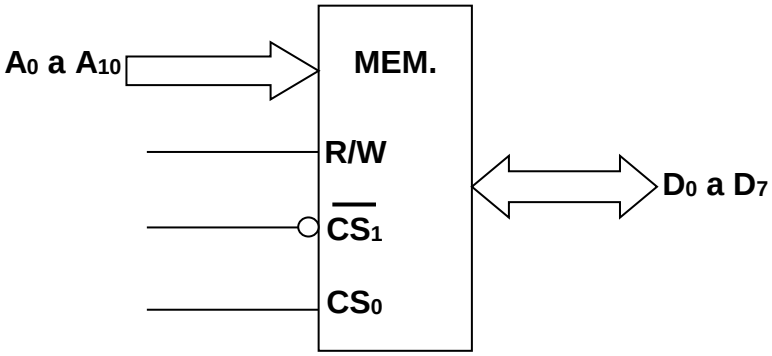
End. Inicial	Endereço Final	Capacidade Mem.	Dispositivo de Mem.	Linha de saída dec.
0000	07FF	2K	1	O_0' e O_1'
0800	0BFF	1K	2	O_2'
0C00	0FFF	2K	3	O_3' e O_4'
1000	17FF	2K	4	O_5' e O_6'
1800	1BFF	1K	5	O_7'

a) Com ROM

A_{12}	A_{11}	A_{10}	CS_0'	CS_1'	CS_2'	CS_3'	CS_4'
A_2	A_1	A_0	B_4	B_3	B_2	B_1	B_0
0	0	0	0	1	1	1	1
0	0	1	0	1	1	1	1
0	1	0	1	0	1	1	1
0	1	1	1	1	0	1	1
1	0	0	1	1	0	1	1
1	0	1	1	1	1	0	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	0



Construir um banco de memória de 8K x 8, partindo de um chip de 2K x 8. Construir o sistema de decodificação usando somente inversores.



Associação de memória

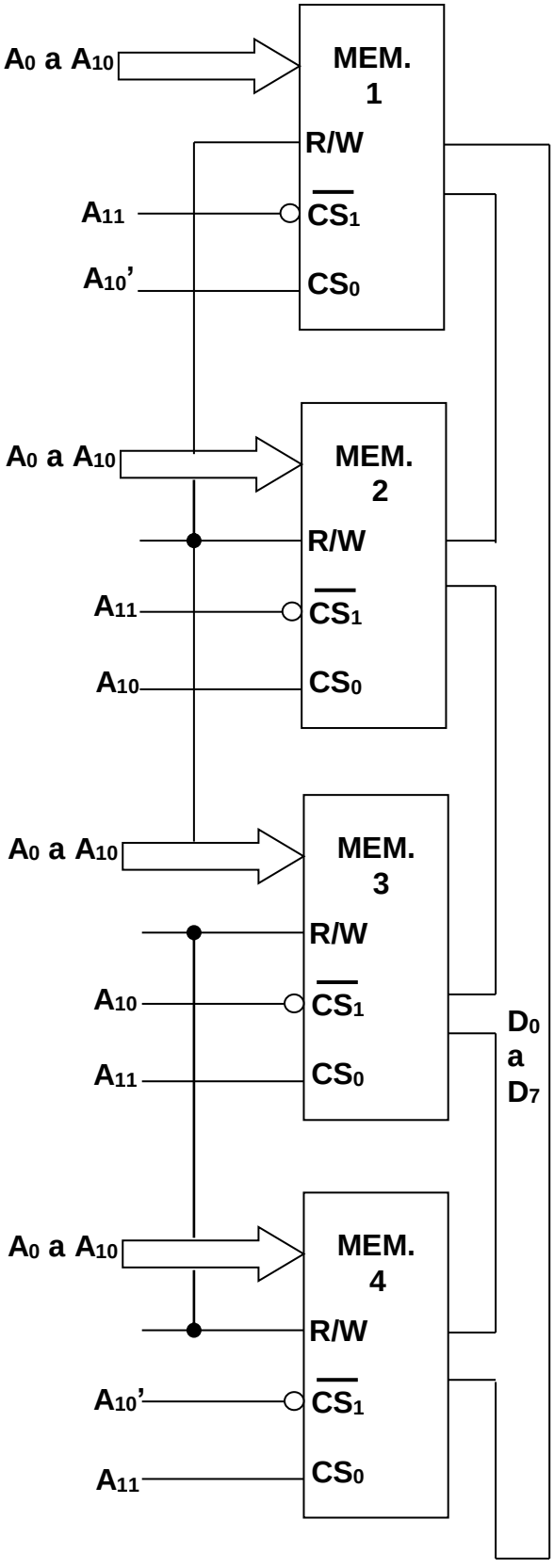


Tabela de endereços da memória de cada memória.

A_{11} A_{10}	Faixa de Endereço	Dispositivo
0 0	0000 a 07FF	1
0 1	0800 a 0FFF	2
1 0	1000 a 17FF	3
1 1	1800 a 1FFF	4

MEMÓRIA ENDEREÇÁVEL PELO CONTEÚDO - CAM

A memória conhecida como CAM é uma memória do tipo associativa, onde o conteúdo desejado é pesquisado na memória inteira se ele está armazenado em qualquer locação. Essa característica da CAM é mais rápida a busca do que a memória RAM.

Memória CACHE

Na área da computação, *cache* é um dispositivo de acesso rápido, interno a um sistema (processador), que serve de intermediário entre um operador de um processo e o dispositivo de armazenamento ao qual esse operador concorda. A vantagem principal na utilização de um cache consiste em evitar o acesso ao dispositivo de armazenamento - que pode ser demorado -, armazenando os dados em meios de acesso mais rápidos. O uso de memórias cache visa obter uma velocidade de acesso a memória próxima da velocidade de memórias mais rápidas, e ao mesmo tempo disponibilizar no sistema uma memória de grande capacidade, a um custo similar de memórias de semicondutores mais baratas.

Com os avanços tecnológicos, vários tipos de cache foram desenvolvidos. Atualmente há cache em processadores, discos rígidos, sistemas, servidores, nas placas-mãe, clusters de bancos de dados, entre outros. Qualquer dispositivo que requeira do usuário uma solicitação/requisição a algum outro recurso, seja de rede ou local, interno ou externo a essa rede, pode requerer ou possuir de fábrica o recurso de cache.

Por ser mais caro, o recurso mais rápido não pode ser usado para armazenar todas as informações. Sendo assim, usa-se o cache para armazenar apenas as informações mais frequentemente usadas. Nas unidades de disco também conhecidas como disco rígido ou *Hard Drive* (HD), também existem chips de cache nas placas eletrônicas que os acompanham. Como exemplo, a unidade Samsung de 160 GB tem 8 MBytes de cache.

Cache de disco

O cache de disco representa uma pequena quantidade de memória incluída na placa lógica do HD. Tem como principal função armazenar as últimas trilhas lidas pelo HD. Esse tipo de cache evita que a cabeça de leitura e gravação passe várias vezes pela mesma trilha, pois como os dados estão no cache, a placa lógica pode processar a verificação de integridade a partir dali, acelerando o desempenho do HD, já que o mesmo só requisita a leitura do próximo setor assim que o último setor lido seja verificado. A seguir é apresentado um diagrama de uma memória cache da CPU.

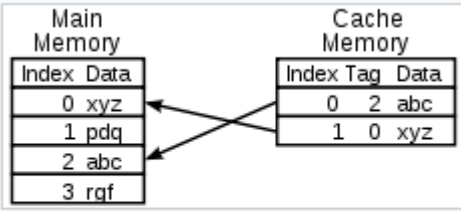


Diagrama de uma memória de cache da CPU.

Operação

Um cache é um bloco de memória para o armazenamento temporário de dados que possuem uma grande probabilidade de serem utilizados novamente.

Uma definição mais simples de cache poderia ser: uma área de armazenamento temporária onde os dados frequentemente acedidos são armazenados para acesso rápido.

Um cache é feito de uma fila de elementos. Cada elemento tem um dado que é a cópia exata do dado presente em algum outro local (original). Cada elemento tem uma etiqueta que especifica a identidade do dado no local de armazenamento original, que foi copiado.

Quando o cliente do cache (CPU, navegador etc.) deseja aceder a um dado que acredita estar no local de armazenamento, primeiramente ele verifica o cache. Se uma entrada for encontrada com uma etiqueta correspondente ao dado desejado, o elemento do cache é então utilizado ao invés do dado original. Essa situação é conhecida como *cache hit* (acerto do cache). Como exemplo, um navegador poderia verificar o seu cache local no disco para ver se tem uma cópia local dos conteúdos de uma página Web numa URL particular. Nesse exemplo, a URL é a etiqueta e o conteúdo da página é o dado desejado. A percentagem de acessos que resultam em *cache hits* é conhecida como a taxa de acerto (*hit rate* ou *hit ratio*) do cache.

Uma situação alternativa, que ocorre quando o cache é consultado e não contém um dado com a etiqueta desejada, é conhecida como *cache miss* (erro do cache). O dado então é copiado do local original de armazenamento e inserido no cache, ficando pronto para o próximo acesso.

Se o cache possuir capacidade de armazenamento limitada (algo comum de acontecer devido ao seu custo), e não houver mais espaço para armazenar o novo dado, algum outro elemento deve ser retirado dela para que liberte espaço para o novo elemento. A forma (heurística) utilizada para seleccionar o elemento a ser retirado é conhecido como política de troca (*replacement policy*). Uma política de troca muito popular é a LRU (*least recently used*), que significa algo como "elemento recentemente menos usado".

Quando um dado é escrito no cache, ele deve ser gravado no local de armazenamento em algum momento. O momento da escrita é controlado pela política de escrita (*write policy*). Existem diferentes políticas. A política de *write-through* (algo como "escrita através") funciona da seguinte forma: a cada vez que um elemento é colocado no cache, ele também é gravado no local de armazenamento original. Alternativamente, pode ser utilizada a política de *write-back* (escrever de volta), onde as escritas não são diretamente espelhadas no armazenamento. Ao invés, o mecanismo de cache identifica quais de seus elementos foram sobrepostos (marcados como sujos) e somente essas posições são colocadas de volta nos locais de armazenamento quando o elemento for retirado do cache. Por essa razão, quando ocorre um *cache miss* (erro de acesso ao cache pelo fato de um elemento não existir nele) em um cache

com a política *write-back*, são necessários dois acessos à memória: um para recuperar o dado necessário e outro para gravar o dado que foi modificado no cache.

O mecanismo de *write-back* pode ser acionado por outras políticas também. O cliente pode primeiro realizar diversas mudanças nos dados do cache e depois solicitar ao cache para gravar os dados no dispositivo de uma única vez.

Os dados disponíveis nos locais de armazenamento original podem ser modificados por outras entidades diferentes, além do próprio cache. Nesse caso, a cópia existente no cache pode se tornar inválida. Da mesma forma, quando um cliente atualiza os dados no cache, as cópias do dado que estejam presentes em outros caches se tornarão inválidas. Protocolos de comunicação entre gerentes de cache são responsáveis por manter os dados consistentes e são conhecidos por protocolos de coerência.

Princípio da localidade de referência

É a tendência de o processador ao longo de uma execução referenciar instruções e dados da memória principal localizados em endereços próximos. Tal tendência é justificada devido as estruturas de repetição e as estruturas de dados, vetores e tabelas utilizarem a memória de forma subsequente (um dado após o outro). Assim a aplicabilidade do cache internamente ao processador fazendo o intermédio entre a memória principal e o processador de forma a adiantar as informações da memória principal para o processador.

Tipos de memória cache

Os tipos de memória cache mais conhecidos são: mapeamento direto, totalmente associativa e associativa por conjunto (N-way). Mapeamento direto : cada bloco da memória principal é mapeado para uma linha do cache. Mapeamento associativo : um bloco da memória principal pode ser carregado para qualquer linha do cache. Mapeamento associativo por conjunto: meio termo direto e o associativo.

Ausência de conteúdo no cache - CACHE MISS

Quando o processador necessita de um dado, e este não está presente no cache, ele terá de realizar a busca diretamente na memória RAM, utilizando wait states e reduzindo o desempenho do computador. Como provavelmente será requisitado novamente (localidade temporal) o dado que foi buscado na RAM é copiado no cache.

Cache em níveis

Com a evolução na velocidade dos dispositivos, em particular nos processadores, o cache foi dividido em níveis, já que a demanda de velocidade a memória é tão grande que são necessários caches grandes com velocidades altíssimas de transferência e baixas latências. Sendo muito difícil e caro construir memórias caches com essas características, elas são construídas em níveis que se diferem na relação tamanho X desempenho.

Cache L1

Uma pequena porção de memória estática presente dentro do processador. Em alguns tipos de processador, como o Pentium 2, o L1 é dividido em dois níveis: dados e instruções (que "dizem" o que fazer com os dados). O primeiro processador da Intel a ter o cache L1 foi o i486 com 8KB. Geralmente tem entre 16KB e 128KB; hoje já encontramos processadores com até 16MB de cache.

Cache L2

Possuindo o Cache L1 um tamanho reduzido e não apresentando uma solução ideal, foi desenvolvido o cache L2, que contém muito mais memória que o cache L1. Ela é mais um caminho para que a informação requisitada não tenha que ser procurada na lenta memória principal. Alguns processadores

colocam esse cache fora do processador, por questões econômicas, pois um cache grande implica num custo grande, mas há exceções, como no Pentium II, por exemplo, cujas caches L1 e L2 estão no mesmo cartucho, que está o processador. A memória cache L2 é, sobretudo, um dos elementos essenciais para um bom rendimento do processador mesmo que tenha um clock baixo. Um exemplo prático é o caso do Intel Itanium 9152M (para servidores) que tem apenas 1.6 GHz de clock interno e ganha de longe do atual Intel Extreme, pelo fato de possuir uma memória cache de 24MB. Quanto mais alto é o clock do processador, mais este aquece e mais instável se torna. Os processadores Intel Celeron têm um fraco desempenho por possuir menos memória cache L2. Um Pentium M 730 de 1.6 GHz de clock interno, 533 MHz FSB e 2 MB de cache L2, tem rendimento semelhante a um Intel Pentium 4 2.4 GHz, aquece muito menos e torna-se muito mais estável e bem mais rentável do que o Intel Celeron M 440 de 1.86 GHz de clock interno, 533 MHz FSB e 1 MB de cache L2.

Cache L3

Terceiro nível de cache de memória. Inicialmente utilizado pelo AMD K6-III (por apresentar o cache L2 integrado ao seu núcleo) utilizava o cache externo presente na placa-mãe como uma memória de cache adicional. Ainda é um tipo de cache raro devido a complexidade dos processadores atuais, com suas áreas chegando a milhões de transístores por micrómetros ou nanómetros de área. Ela será muito útil, é possível a necessidade futura de níveis ainda mais elevados de cache, como L4 e assim por diante.

Caches inclusivos e exclusivos

Caches Multi-level introduzem novos aspectos na sua implementação. Por exemplo, em alguns processadores, todos os dados no cache L1 devem também estar em algum lugar no cache L2. Estes caches são estritamente chamados de inclusivos. Outros processadores (como o AMD Athlon) têm caches exclusivos - os dados podem estar no cache L1 ou L2, nunca em ambos. Ainda outros processadores (como o Pentium II, III, e 4 de Intel), não requerem que os dados no cache L1 residam também no cache L2, embora possam frequentemente fazê-lo. Não há nenhum nome universal aceitado para esta política intermediária, embora o termo inclusivo seja usado.

A vantagem de caches exclusivos é que são capazes de armazenarem mais dados. Esta vantagem é maior quando o cache L1 exclusivo é de tamanho próximo ao cache L2, e diminui se o cache L2 for bastante acoplado ao núcleo, as vezes maior do que o cache L1. Quando o L1 falha e o L2 acerta o acesso, a linha correta do cache L2 é trocada com uma linha no L1. Esta troca é um problema, uma vez que a quantidade de tempo para tal troca ser realizada é relativamente alta.

Uma das vantagens de caches estritamente inclusivos é que quando os dispositivos externos ou outros processadores em um sistema multiprocessado desejam remover uma linha do cache do processador, necessitam somente mandar o processador verificar o cache L2. Nas hierarquias de cache exclusiva, o cache L1 deve ser verificado também.

Uma outra vantagem de caches inclusivos é que um cache maior pode usar linhas maiores do cache, que reduz o tamanho das Tags do cache L2. (Os caches exclusivos requerem ambos os caches teres linhas do mesmo tamanho, de modo que as linhas do cache possam ser trocadas em uma falha no L1 e um acerto no L2).

Tamanho do cache

Quando é feita a implementação da memória cache, alguns aspectos são analisados em relação a seu tamanho:

- relação acerto/falha;
- tempo de acesso a memória principal sera mais lenta
- o custo médio, por bit, da memória principal, do cache L1 e L2:

- o tempo de acesso do cache L1 ou L2;
- a natureza do programa a ser executado no momento é inferior

Bbliografia:

MURDOCCA, M. J.; Heuring, V. P. **Introdução a Arquitetura de Computadores.** Campus, 2001.

STALLINGS, W. **Arquitetura e organização de computadores**. Prentice Hall, 2003.

TANENBAUM, A. S. **Organização estruturada de computadores**. Prentice Hall Brasil, 2007.

Complementar

HAYES, J. **Computer architecture and organization**. 2ª ed. McGraw Hill Publishing Company, 1988.

DAVID A. PATTERSON. **Organização e Projeto de Computadores. Interface harware/software**. 4ª ED. ISBN-13: 978-0123747501

FRANÇA, P.; ZELENOVSKY, R.; MENDONÇA, A. **Hardware**: **programação virtual de I/O e interrupção**. MZ, 2001.

OLIVEIRA, J F de; MANZANO, J A N G. **Algoritmos**: **lógica para desenvolvimento de programação de computadores**. Érica, 2009.

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos da programação de computadores**: **Algoritmos, Pascal e C/C++.** São Paulo: Prentice Hall, 2002.

CALDAS, L **– Roteriros de aulas:**
Fluxo de dados, memórias voláteis e mão voláteis, conversores de bases numéricas.

VAHID, F **– Sistemas digitais – Projeto, otimização HDL** – Editora John Wiley and Sons 2007.