

J. ASSUMPTÃO, C. KLÜSER, V.
HERRERA, J. CARMO E M. GAZZIRO

COMPUTADORES E VIDEOGAMES



ABORDAGEM PRÁTICA DAS
ARQUITETURAS CLÁSSICAS



A851c

Assumpção Junior, Jecel Mattos de
Computadores e videogames : uma abordagem prática das
arquiteturas clássicas / Jecel Mattos de Assumpção Junior;
Chandler Klüser; Victoria Alejandra Salazar Herrera;
João Paulo Pereira do Carmo; Mario Alexandre Gazziro;
Capa de Vinícius de Souza Caffeu.-

[221]p.

ISBN 978-65-571-9060-9

1. Engenharia de computadores. 2. Ciência da computação.
3. Videogames. I. Título.

CDD: 621.39

Prefácio

Em um mundo onde a tecnologia define as fronteiras do possível, a arte de compreender a arquitetura de computadores emerge não apenas como um campo de estudo, mas como a essência da criação no universo dos videogames. "Computadores e Videogames" é uma jornada que transcende a simples funcionalidade de software e hardware, adentrando no mundo onde a eficiência e a inovação se entrelaçam para dar vida a experiências imersivas.

Tal qual um artesão que, com suas mãos, transforma a madeira bruta em objetos de beleza e propósito, o desenvolvedor de videogames utiliza seu profundo entendimento da arquitetura de computadores para esculpir experiências que cativam e emocionam. Este livro é um convite para explorar os fundamentos que possibilitam tal magia: desde os circuitos digitais que são o coração pulsante dos computadores até a geração de vídeos, que compõem a alma de um jogo.

Ao adentrarmos neste mundo, percebemos que o domínio sobre arquitetura de computadores é crucial. Ela não é apenas uma disciplina técnica, mas o alicerce sobre o qual as experiências de jogo são construídas.

"Computadores e Videogames" é, portanto, mais do que um livro; é um manifesto sobre a importância de entender a fundo a arquitetura de computadores para quem deseja não apenas jogar, mas criar jogos que deixarão sua marca no mundo. À medida que navegamos por seus capítulos, somos convidados a nos tornarmos artesãos da era digital, cada um com a habilidade de criar mundos que, até então, residiam apenas na imaginação.

Seja bem-vindo a este fascinante encontro de caminhos, onde a paixão por computadores e videogames se encontra com o conhecimento técnico e a criatividade, desbloqueando um universo de possibilidades.

Rafael Sene - Technical Program Manager, RISC-V, The Linux Foundation

Sumário

Introdução	7
------------------	---

I

Parte I - Videogames de PRIMEIRA geração

Videogames de Primeira Geração

17

Circuitos Digitais	18
Do que são feitos os computadores	19
Circuitos Combinacionais	24
Multiplicador	40
Circuitos Sequenciais	46
Latches e Flip-flops	49
Memórias de Acesso Aleatório - RAM	51
Contadores	53
Máquinas de Estados Finitos	59
Multiplicador Sequencial	61
Experimentos com vídeo	72
Jogo Pong criado no simulador Digital	80
Linguagens HDL	83
Jogo Pong em Verilog	99

II

Parte II - Videogames de SEGUNDA geração

Processador AP9	103
Jogo Tetris em Assembler	163

III

APÊNDICE A - Jogo Pong codificado em Verilog

IV**APÊNDICE B - Tetris codificado em Assembler**



Introdução

Os videogames, ou jogos eletrônicos, são hoje uma grande parte da indústria de entretenimento, com um faturamento três vezes maior que o do cinema mundial e sete vezes o do setor de música. Para muitos de nós é como gastamos boa parte (senão a maior parte) das nossas horas de diversão.

Mas é possível entender como são feitos os videogames, em todos os níveis, com detalhes suficientes para podermos criar nossas próprias versões? Este livro é uma resposta positiva a esta pergunta. E como os videogames são um tipo especializado de computadores (tele-

crédito de todas as imagens desse capítulo: pixabay.com

phones celulares modernos são outro tipo de computadores, por exemplo) a maior parte do conhecimento obtido no estudo dos games serve para computadores em geral também.

Um videogame moderno contém muitos bilhões de componentes. Como é possível entender algo tão complicado? Vamos adotar duas estratégias para lidar com isso. A primeira estratégia é uma recapitulação histórica. A história dos videogames domésticos é dividida em gerações, como mostram as tabelas desse capítulo. Note que os anos em que estes consoles (nome do aparelho do videogame que é ligado à televisão) chegaram ao Brasil raramente são os mostrados na tabela, em função das restrições à importação até os anos 1990. E os nomes são os usados no mercado norte-americano: o NES (Nintendo Entertainment System) era Famicom no Japão enquanto o Sega Genesis era Mega Drive em outros países.

Geração	Datas	Exemplos
Primeira	1972-1980	Odyssey, Pong, Telstar
Segunda	1976-1992	Channel F, Atari 2600, Odyssey 2, Intellivision, ColecoVision
Terceira	1983-1992	NES (EUA)/Famicom (Japão), Master System, Atari 7800
Quarta	1987-2004	TurboGrafx-16, Genesis/Mega Drive, Neo Geo, Super NES
Quinta	1994-2006	Saturn, PlayStation, Nintendo 64
Sexta	1998-2013	Dreamcast, PlayStation 2, GameCube, Xbox
Sétima	2005-2017	Xbox 360, PlayStation 3, Wii
Oitava	2012-hoje	PlayStation 4, Xbox One
Nona	2020-hoje	Xbox séries X e S, PlayStation 5

Tabela 1: Resumo das Gerações dos Videogames Domésticos



Tabela 2: Primeira Geração: Odyssey, Pong, Telstar (sem suporte a cartuchos, jogos internos)



Tabela 3: Segunda Geração: Channel F, Atari 2600, Odyssey 2, Intellivision, ColecoVision



Tabela 4: Terceira Geração: NES (EUA)/Famicom (Japão), Master System, Atari 7800

Os videogames de cada geração foram mais complexos que os da geração anterior, de modo que, se seguirmos a evolução histórica, poderemos entender completamente um exemplo mais limitado antes de estudar os mais avançados. Os projetos mostrados neste livro são no estilo dos videogames históricos da primeira até mais ou menos a segunda geração, mas sem serem uma reprodução dos mesmos. As demais gerações são apresentadas nas tabelas a seguir.



Tabela 5: Quarta Geração: TurboGrafx-16, Genesis/Mega Drive, Neo Geo, Super NES



Tabela 6: Quinta Geração: Saturn (primeiro a usar CD-ROM), PlayStation, Nintendo 64



Tabela 7: Sexta Geração: Dreamcast, PlayStation 2, GameCube, Xbox



Tabela 8: Sétima Geração: Xbox 360, PlayStation 3, Wii



Tabela 9: Oitava Geração: PlayStation 4, Xbox One



Tabela 10: Nona Geração: Xbox séries X e S, PlayStation 5

A segunda estratégia é o uso do que chamamos de “níveis de abstração”. Uma cidade, por exemplo, é feita de milhões de tijolos. Mas “milhões” não é algo que podemos realmente compreender. Uma visão alternativa, mais abstrata, da cidade é que ela é constituída por uma dezena de bairros. Dez é um número mais humano. Conseguimos verdadeiramente entender a

cidade assim. Em outro momento, podemos esquecer a cidade e nos concentrarmos num dos bairros. Vemos que ele é feito de algumas praças, ruas, avenidas e quarteirões. Se focarmos num determinado quarteirão, veremos alguns prédios e casas.

Continuando o processo, podemos ignorar o quarteirão e observar que uma certa casa é feita de telhado, fundação e por volta de 8 paredes em média. Uma parede pode ter até milhares de tijolos, mas é um padrão repetitivo e se entendermos como é feito um metro de parede, também teremos entendido como é feita a parede completa, com exceção de algumas condições de contorno. Estes são os lugares onde o padrão repetitivo é interrompido, como nas portas, janelas ou quinas.

Da mesma forma que uma cidade com milhões de tijolos pode ser estudada completamente focando em um nível de abstração por vez, um videogame ou computador de bilhões de transistores pode ser entendido seguindo um caminho equivalente. Duas direções são igualmente válidas para esta jornada. Podemos começar com os tijolos, depois as paredes e assim por diante até chegar na cidade. Esta estratégia é conhecida como “de baixo para cima”. O caminho inverso é “do alto para baixo”.

A vantagem de irmos de baixo para cima é que em cada etapa usamos componentes já estudados para compor o nível seguinte. Sempre sabemos o “como” do que está sendo feito. Mas falta a visão dos passos seguintes, o “porquê” do que está sendo aprendido. Já na estratégia de alto para baixo o “porquê” é sempre bem claro, pois a primeira coisa estudada é o resultado final. O “como” é que fica para mais tarde, por isso esta alternativa também é conhecida como “revelação sucessiva”.

Neste livro estudaremos as abstrações de baixo para cima com a ideia de que saber que o “porquê” final é o videogame seja motivação suficiente para o aprendizado das abstrações intermediárias. Além disso, na estratégia de seguir a evolução histórica dos videogames, veremos que os níveis mais altos de abstração foram sendo introduzidos ao longo do tempo e só precisamos dos níveis mais baixos a intermediários para os jogos de primeira geração, por exemplo.

A Figura 1 acima é um exemplo de representação abstrata de um videogame. O uso de retângulos acompanhados de textos indicativos é bem comum para representar componentes de um sistema. Note que os dois retângulos mais à esquerda estão sem nenhum texto indicando sua função, mas sua aparência é semelhante à dos controles do videogame. O uso de formatos

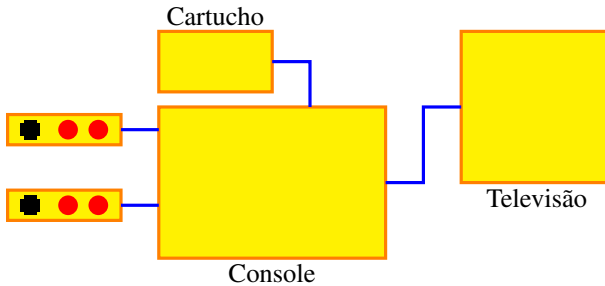


Figura 1: Videogame abstrato

especiais para certos componentes facilita o reconhecimento dos mesmos.

Uma convenção bastante popular é mostrar os sinais entrando do lado esquerdo dos componentes e as saídas pelo lado direito. Na Figura 1 a ligação entre o cartucho e o aparelho de videogame em si (o console) não segue esta convenção, pois existem sinais que vão do cartucho para o console, enquanto outros vão do console para o cartucho. Mesmo sinais mais simples podem fugir da convenção se o resultado for um desenho mais confuso ou desajeitado.

Em princípio, o sistema da Figura 1 não pode funcionar, pois nem a televisão e nem o console estão ligados à rede elétrica. Uma possibilidade é que os dois aparelhos funcionam com pilhas, mas o que realmente está acontecendo é que representações abstratas simplesmente omitem todos os detalhes que não sejam necessários para a missão da figura. Neste caso, o objetivo é mostrar que componentes são necessários para usar o videogame e o que vai ligado aonde. O fato de que os dois principais componentes precisam ter fios ligados à tomada na parede fica subentendido. Mas mesmo que isso fosse explicitamente desenhado, as tomadas seriam então exibidas como se estivessem flutuando? Nesse caso estaria mais uma vez faltando detalhes na representação. As tomadas fazem parte da rede elétrica da casa. Mas, e se fosse desenhada esta rede, como ficam os fios que vão para a rua? Teríamos que incluir a rede do bairro, da cidade e do país. Ou podemos nem desenharmos os fios de força dos aparelhos, supondo que os leitores saibam que eles estão lá. Este vai ser o caso para praticamente todas as figuras deste livro.



Parte I Videogames de PRIMEIRA geração

Videogames de Primeira Geração

Circuitos Digitais 18

Do que são feitos os computadores
Circuitos Combinacionais
Circuitos Sequenciais
Experimentos com vídeo

Jogo Pong criado no simulador Digital 80

Linguagens HDL 83

Jogo Pong em Verilog

Videogames de Primeira Geração

Geralmente considerado como sendo o primeiro videogame, o “Tenis for Two” foi construído em 1958 por William Higinbotham como uma demonstração para o público organizada pelo seu laboratório. Ele usou o computador analógico Modelo 30 da Donner junto com controladores e circuitos que ele criou para gerar formas de onda que podiam ser observadas na tela de um osciloscópio. Um risco horizontal representava uma quadra de tenis vista de lado e um pequeno risco vertical era a rede. Um ponto brilhante era a bola que se movia em trajetórias parabólicas que eram refletidas no chão. Arrasto aerodinâmico e outros efeitos físicos eram simulados. O objetivo era passar por cima da rede.

O primeiro grande impacto na área de videogames veio com o *Spacewar!* programado em 1962 no computador PDP-1 no MIT por Steve Russell e alguns outros estudantes. A tela mostrava duas espaçonaves que se moviam de maneira realista (para reduzir a velocidade você tinha que virar a nave ao contrário e acionar o foguete, por exemplo). As naves podiam atirar uma na outra e precisavam lidar com obstáculos como uma estrela cuja gravidade atraía as naves. O jogo foi depois traduzido para outros computadores e se tornou bem popular.

Em 1966 o Ralph Baer começou o desenvolvimento de um videogame caseiro que usaria o aparelho de TV que as pessoas já tinham para exibir suas imagens. O circuito digital era capaz de mostrar alguns pequenos quadrados na tela permitindo um pequeno número de jogos, todos mais ou menos parecidos. A Magnavox acabou licenciando a invenção e em 1972 lançou com o nome Odyssey. O produto chegou a ser comercializado no Brasil mas poucas unidades foram vendidas de modo que quando a Philips lançou um videogame mais avançado que se chamava Odyssey 2 no exterior, ela não achou necessário usar o 2 no país.

Em 1971 o Nolan Bushnell e Ted Dabney lançaram o *Computer Space* para tentar tornar disponível comercialmente a experiência do *Spacewar!*. Os usuários podiam jogar colocando moedas na máquina. Como um computador ficaria muito caro, eles projetaram um hardware específico para o jogo. Eles não tiveram o sucesso desejado e no ano seguinte criaram a empresa Atari.

Com um contrato para projetar um jogo de corridas para a Bally, a Atari contratou o Allan Alcorn mas, o Bushnell estava preocupado que o projeto seria complicado demais para o primeiro videogame do Alcorn. Por isso falou para ele começar por um jogo com uma bola e duas raquetes como o Tenis do Odyssey, com uma quadra de tenis ou mesa de ping-pong

vista de cima. Ele deu a entender que existia um cliente interessado nisso para motivá-lo, mas quando ficou pronto os diretores ficaram impressionados e resolveram que a Atari mesma iria comercializar. O lançamento do Pong no fim de 1972 foi um grande sucesso.

Os videogames de primeira geração, então, eram circuitos digitais especialmente projetados para cada jogo. Um exemplo famoso foi a criação de uma versão do Pong para um só jogador chamada de *Breakout*. O Bushnell ofereceu um bonus para seu funcionário Steve Jobs para o desenvolvimento do jogo com um valor crescente para cada chip a menos que 50 (os jogos da Atari usavam tipicamente 120 a 150 chips). O Jobs buscou ajuda de seu amigo Steve Wozniak que era funcionário da HP. O Wozniak conseguiu fazer funcionar uma versão com apenas 44 chips mas o Jobs repassou para ele apenas \$350 escondendo o valor total do bonus. No fim a Atari achou complicado de mais mexer no projeto dos dois Steve e acabou comercializando outra versão com 100 chips.

O Wozniak ficou curioso sobre a possibilidade de criar o Breakout como um programa na linguagem BASIC em um computador de baixo custo. Ele evoluiu o computador Apple que ele havia criado para ter os recursos necessários para tal jogo. O Apple II permitiu o lançamento da empresa Apple com muito sucesso, sendo superior aos grandes concorrentes TRS-80 da Radio Shack e o Pet da Commodore em termos de jogos.



Circuitos Digitais

Do que são feitos os computadores

Originalmente “computador” era o nome de uma profissão. Era uma pessoa que produzia resultados numéricos para problemas de cientistas, militares ou empresários. As máquinas construídas para fazer a mesma função nos anos 1940 acabaram herdando o nome, que acabou sendo preferido a alternativas como “cérebros eletrônicos” ou “calculadoras automáticas”.

Um tipo de computador é conhecido como “analógico” pois valores dentro da máquina são

crédito da imagem: wallpapersafari.com / Green Motherboard

análogos aos valores do problema que está sendo calculado. Para uma simulação ecológica, por exemplo, a tensão num ponto do circuito pode representar a população de coelhos numa floresta, enquanto a tensão em outro ponto representa o número de lobos.

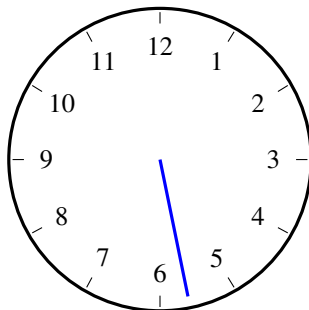


Figura 2: Relógio analógico

A Figura 2 mostra um exemplo de um computador analógico mecânico. O ângulo do ponteiro azul é análogo ao tempo e, se o ponteiro der duas voltas por dia, esta analogia vai ser de 1 para 1. Selecionando o valor inicial correto, este aparelho vai mostrar a hora atual. No exemplo, o ponteiro está num ângulo de 281,40833 graus, que corresponde a 5 horas, 37 minutos e 11 segundos. Infelizmente, não conseguimos medir o ângulo do ponteiro com esta precisão e, mesmo que conseguíssemos, as engrenagens não são perfeitas e o ponteiro tem uma certa folga. Se o ponteiro estiver apenas um grau para frente ou para trás de onde deveria estar, isso representaria um erro de dois minutos em relação à hora atual.

Na Figura 3 vemos uma possível solução. Adicionamos um segundo relógio cujo ponteiro dá uma volta a cada hora. Fica bem mais fácil ver que atualmente é 5 horas e 37 minutos. A legenda diz que este é um relógio digital, mas na verdade o da esquerda é digital enquanto o da direita é analógico, então seria melhor falar em relógio híbrido. Mas, como se deu esta mudança se o relógio da esquerda é exatamente o mesmo que o da Figura 2? A diferença é muito sutil, mas importante. Na primeira figura, ângulos de 281 graus e 283 graus representavam horas diferentes, enquanto na segunda figura todos os ângulos entre 270 e 300 graus representam 5

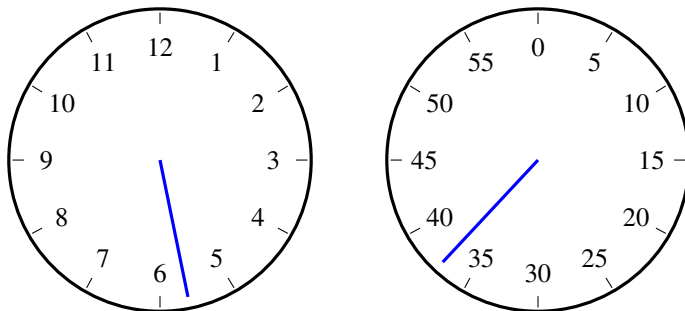


Figura 3: Relógio digital (híbrido)

horas e é o relógio da direita que indica os minutos.

Uma alteração nas engrenagens pode deixar isso ainda mais claro. É possível fazer o ponteiro das horas ficar parado bem em cima da hora atual e dar um salto de 30 graus cada vez que o ponteiro dos minutos passar pelo 0. Isso não apenas facilita ainda mais a leitura, mas também faz folgas mecânicas de menos de 15 graus deixarem de causar erros.

O problema desta solução é que estamos pagando por dois relógios para fazer a mesma coisa que antes um só resolvia. Podemos continuar nesta direção acrescentando um terceiro relógio, também com marcações de 0 a 59, cujo ponteiro dá uma volta por minuto. Nesse caso fica fácil saber que são 5 horas, 37 minutos e 11 segundos. Mas o custo agora é o triplo da primeira solução. Nada impede o uso de um quarto relógio com marcações de 0 a 9 com ponteiro que dá uma volta por segundo. Ou um quinto relógio (também de 0 a 9) que dá 10 voltas por segundo. Também é possível ir na outra direção, com um sexto relógio com marcações de “dia” e “noite” e com ponteiro que dá uma volta por dia. Já um sétimo relógio com os nomes dos dias da semana e com ponteiro dando uma volta a cada 7 dias seria um pouco de exagero, mas completamente viável.

Normalmente, os vários relógios partilham um único mostrador, mas separando-os lado a lado na figura ilustra melhor o custo de se ter mais dígitos.

Um detalhe é que “4” não é o número quatro, mas sim uma representação deste número no

sistema hindu-arábico. Uma outra representação possível seria “kkkk”. Isto também não é o número quatro, mas quatro é o número de letras “k” no texto. Uma característica do sistema hindu-arábico é que ele é um sistema posicional - as mesmas figuras representam números diferentes quando aparecem em posições diferentes. Já nos números romanos, “X” é usado para indicar dez, enquanto “C” é usado para indicar cem. Um sistema assim pode existir sem o zero (mas não sem alguns problemas) enquanto que nos sistemas posicionais o zero é fundamental. Se imaginarmos um conjunto de 3 relógios representando horas, minutos e segundos, poderemos distinguir os dois últimos pela velocidade dos seus ponteiros. Mas, em uma foto destes relógios, iríamos depender da posição deles para saber qual é qual.

Uma diferença sutil é que os computadores analógicos manipulam números diretamente enquanto os computadores digitais manipulam representações destes números na forma de vários dígitos. Quantos valores pode assumir um dígito? No exemplo do relógio falamos em 12, 60, 10, 2 e 7. São todas opções válidas, mas note que quanto menos valores por dígito, mais dígitos precisamos ter para representar o mesmo número. Isso significa mais cópias do mecanismo ou circuito, mas se cada um destes puder ser mais simples pode até compensar. Circuitos onde só existem dois valores diferentes, por exemplo, são bem mais simples que as alternativas. E observando o relógio manhã/tarde vemos que ele é o mais robusto de todos pois só uma folga mecânica de mais de 90 graus faz ele dar uma leitura errada.

Quaisquer que sejam os componentes de um computador, eles devem ter pelo menos estas características:

1. entradas e saídas com a mesma natureza: um bloco que receba sons como entrada e emita luz como saída, por exemplo, não pode ser ligado a outros blocos iguais para formar sistemas maiores.
2. saídas com mesma intensidade das entradas: se pressão hidráulica for usada para indicar valores, por exemplo, e se a pressão na saída for mais fraca do que a que está entrando, não será possível ligar mais do que uns poucos componentes até o resultado ser fraco demais para ser útil.
3. saídas com menos ruído que as entradas: nos sistemas analógicos o nível de ruído normalmente aumenta a cada operação. Isso limita o tamanho dos sistemas que podem ser construídos. Nos sistemas digitais é possível gerar saídas com menos ruído que as entradas (como no caso dos ponteiros de relógio que dão um salto) eliminando qualquer

Nível	Exemplos de Ferramentas	
Arquitetura	QEMU	MAME
Micro-arquitetura	SPIM	SimpleScalar
Transferência de Registradores	Verilator	ModelSim
Portas Lógicas	Digital	TkGate
Chaves	IRSIM	MOSSIM
Circuitos Analógicos	Spice	Xyce
Dispositivos	TCAD	DEVSIM
Física	Elmer	Matlab

Tabela 11: CAD para diferentes níveis de abstração

limite para o tamanho do sistema.

No resto do livro descreveremos computadores digitais eletrônicos que manipulam números representados na base 2 (números binários) usando blocos básicos que atendem às características acima.

Existem programas de computador que podem ajudar no desenvolvimento dos computadores e dos videogames. As ferramentas de CAD (“Computer Aided Design”) permitem a criação de desenhos que sejam uma descrição detalhada do circuito a ser construído, enquanto que simuladores permitem que seu funcionamento seja verificado antes mesmo da sua construção.

Para cada nível de abstração existem diferentes programas que são os mais indicados. A Tabela 11 mostra dois exemplos para cada nível, mas na verdade várias destas ferramentas podem ser usadas em mais de um nível. O simulador Digital¹ do Helmut Neemann, por exemplo, é muito bom para projetos no nível de portas lógicas, como veremos no resto deste capítulo. Mas também serve muito bem para projetos ao nível de transferência de registradores.

A Figura 4 mostra que o Digital também pode ser usado para o nível de chaves apesar deste não ser seu objetivo principal. Na parte de cima do circuito vemos 3 chaves manuais ligando a fonte de energia a um LED. Durante a simulação, quando a combinação correta de chaves é pressionada, o LED acende. O problema deste circuito é que viola a primeira regra

¹<https://github.com/hneemann/Digital>

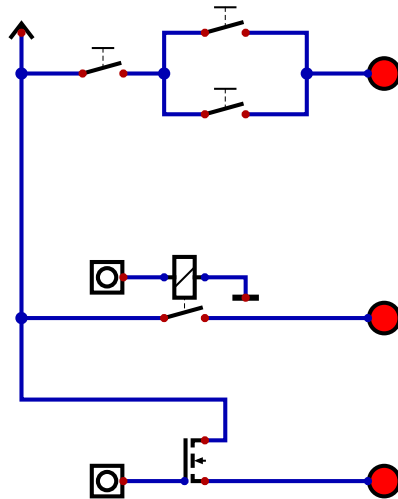


Figura 4: Exemplos do nível de chaves

da lista acima. A entrada é o dedo de uma pessoa pressionando a chave e a saída é luz ou um sinal elétrico. No circuito do meio, temos um relê que é exatamente igual à chave mas substitue o dedo por um solenoide que é acionado por uma corrente elétrica. Agora a entrada e saída têm a mesma natureza como em alguns computadores, incluindo o Mark I de Harvard de 1944, que é de onde vem o termo “arquitetura de Harvard” para computadores que usam memórias separadas para dados e instruções.

O circuito de baixo na Figura 4 é exatamente igual ao do meio, mas com o relê substituído por um transistor MOSFET tipo N.

O Digital é um simulador interativo onde, durante a simulação, os valores das entradas podem ser alteradas e chaves e botões podem ser pressionados. O valor de cada sinal é mostrado com mudanças de cor nos fios e dispositivos de saída como LEDs, o terminal ou até monitor VGA mostram seus resultados.

Circuitos Combinacionais

Todos os blocos têm um certo número de entradas e de saídas, onde esses sinais só podem assumir valores correspondentes a 0 ou 1, já que adotamos dígitos binários (“bit” de “binary digit”). Nos circuitos combinacionais, os valores das saídas dependem apenas das diferentes combinações nas entradas.

Portas Lógicas

Para circuitos com duas entradas (e uma saída) existem apenas 4 combinações possíveis para essas entradas: 0 e 0, 0 e 1, 1 e 0 e 1 e 1. Já que os diferentes circuitos podem ter ou 0 ou 1 na saída para cada combinação, isso significa que existem apenas 2^4 circuitos combinacionais possíveis com duas entradas. 16 é um número suficientemente pequeno para que possamos mostrá-los todos de uma só vez.

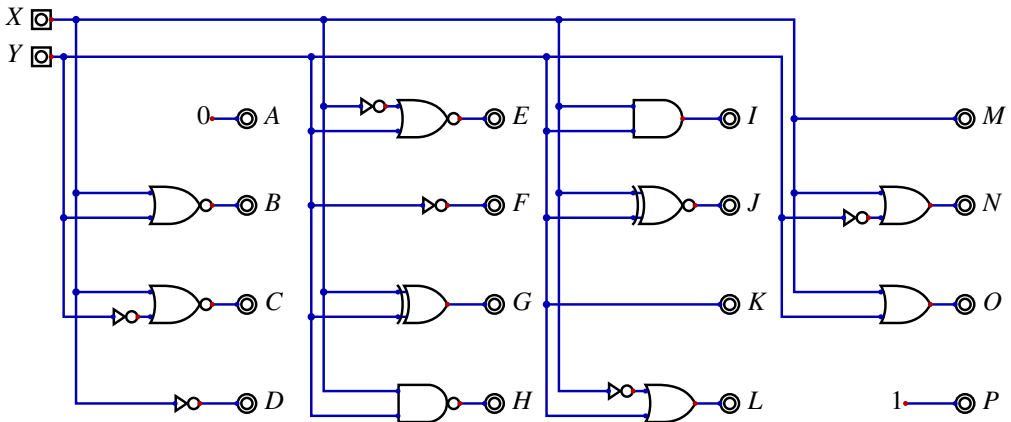


Figura 5: Todos os circuitos combinacionais de duas entradas

Usando a função “Análises” do simulador Digital, podemos verificar que estes 16 circuitos realmente são todos os possíveis circuitos combinacionais de duas entradas observando que as

colunas A a P são os números binários correspondentes a 0 até 15 sem pular nenhum e sem repetições.



Tabela																	
Arquivo Novo Editar Criar K-Map																	
X	Y	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1

Figura 6: Tabela verdade dos circuitos de duas entradas

Observe que A e P não têm, na verdade nenhuma entrada, enquanto D, F, K e M só usam realmente uma entrada. Mas seria bem simples implementar todas as 6 funções usando duas entradas. Um circuito que faça $\text{AND}(X, \text{NOT}(X))$ teria sempre 0 na saída independente de X, por exemplo, e por isso seria uma maneira de implementar a função A.

Essa notação textual para descrever circuitos é apenas uma das que você poderá encontrar por aí. Tanto é que, no simulador Digital, no menu “Editar”, no comando “Configurações”, existe a opção de escolher uma notação diferente.

A primeira notação oferecida no menu é a sintaxe usada pela linguagem de programação C. A segunda é bem parecida com a que usamos acima para o circuito que gera sempre zero. As demais notações foram derivadas de áreas da matemática que têm certa sobreposição.

Introduzido pela primeira vez em 1847 por George Boole, e ampliado por ele em 1854, o que hoje é chamado de Álgebra Booleana funciona de maneira muito semelhante à álgebra normal, mas suas variáveis só podem adotar os valores 1 e 0. Isso torna a adição ligeiramente diferente do normal e o resultado 1 mais 1 é 1 em vez de 2. Podemos usar “+” para indicar somas e nada ou “ $\times$ ” para indicar multiplicação. A função inversão pode ser representada por um prefixo “ $\sim$ ”, “!”, “ $\neg$ ” ou por um traço sobre a expressão a ser invertida.

Para o caso especial, onde apenas o conjunto universal e o conjunto vazio são usados, a Teoria dos Conjuntos (formalizada por Georg Cantor em 1874) nos dá os mesmos resultados que a Álgebra Booleana. Em vez de um produto podemos usar a operação de interseção ($\cap$) e

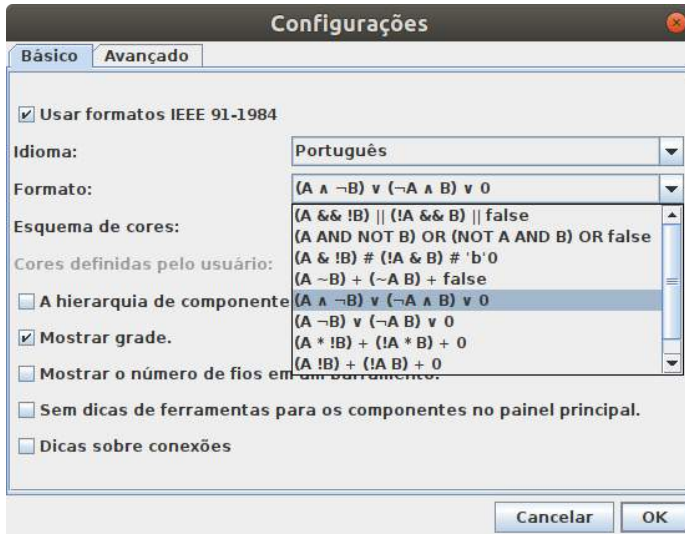


Figura 7: Notações alternativas oferecidas pelo simulador Digital

em vez de uma soma podemos usar a operação de união ($\cup$). O complemento é a diferença entre o conjunto universal e alguma variável.

Tradicionalmente, a lógica tem feito parte da filosofia e da retórica, embora o desenvolvimento da lógica de predicados pelos estóicos (século III aC) tenha preparado o cenário para a evolução do século XIX em um ramo da matemática. Em analogia aos operadores de conjunto, podemos usar “ $\wedge$ ” para conjunção (AND) e “ $\vee$ ” para disjunção (OR). Esses símbolos também são frequentemente usados em Álgebra Booleana como alternativas aos já listados acima.

Embora os blocos de construção básicos sejam chamados de “portas lógicas” porque suas operações podem ser descritas, entre outras, pela lógica de matemática, isso não significa que os computadores construídos a partir de tais blocos possam ser chamados de “lógicos” no sentido popular da palavra. A lógica matemática pode ser implementada em computadores

área	elementos equivalentes				
Álgebra Booleana	1	0	inversão	soma	produto
Lógica de predicados	verdade	falso	não	ou	e
Teoria dos conjuntos	conjunto universal	conjunto vazio	complemento	união	intersecção
Circuitos de chaveamento	5V	0V	normalmente fechado	paralelo	série

Tabela 12: Áreas equivalentes

através de linguagens de programação como Prolog, mas você ainda precisa de enormes bancos de dados de fatos de “senso comum” (como tentado no projeto Cyc) para que os computadores operem de uma maneira que a maioria das pessoas chamaria de lógica. Os atuais Grandes Modelos de Linguagem, treinados no conteúdo da *World Wide Web*, são uma boa alternativa para alcançar essa funcionalidade.

A tese de mestrado de Claude Shannon de 1937 “Uma análise simbólica de relés e circuitos de comutação” provou que a álgebra booleana poderia ser usada para descrever circuitos de comutação feitos de relés. Veja a seguinte ilustração da Figura 8:

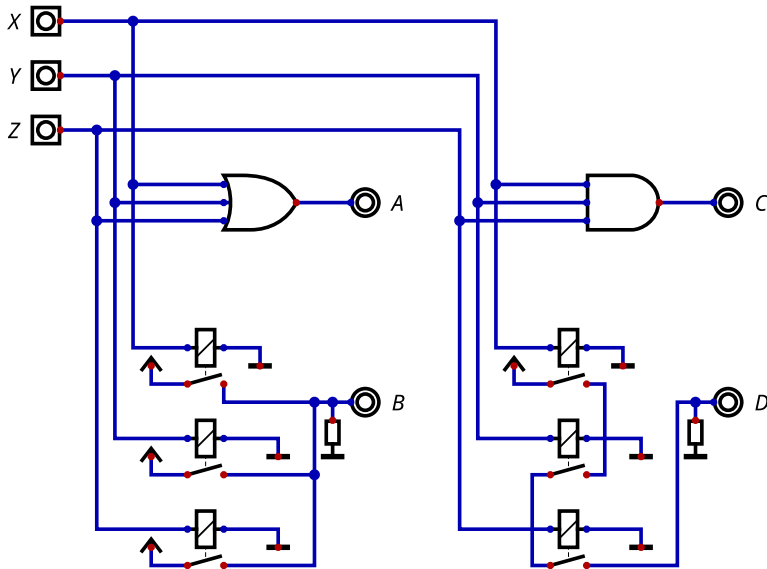
Usando o software Digital para gerar uma tabela verdade para os quatro circuitos mostra que A e B implementam a mesma função, assim como C e D.

Isso significa que conectar relés em paralelo é equivalente à porta OR e conectá-los em série nos dá o mesmo resultado do que a porta AND. Isso pode ser visto nas equações geradas a partir da opção análise do Digital.

Pensando em termos de chaves, é óbvio que as portas OR e AND podem ser facilmente estendidas para qualquer número de entradas e não apenas 2. A porta NÃO (que pode ser implementada com um relé normalmente fechado) sempre possui uma única entrada.

Ao gerar as equações para um circuito, o Digital sempre as expressará em termos dos operadores NOT, AND e OR. No circuito onde vimos todas as 16 portas de duas entradas, vemos também as portas XOR (OR exclusivo) e XNOR (equivalência) e estas são operações básicas na lógica matemática. Então, quais portas são as mais fundamentais e quais podem ser criadas como circuitos compostos usando outras portas?

Figura 8: Exemplo de uso de circuitos de comutação



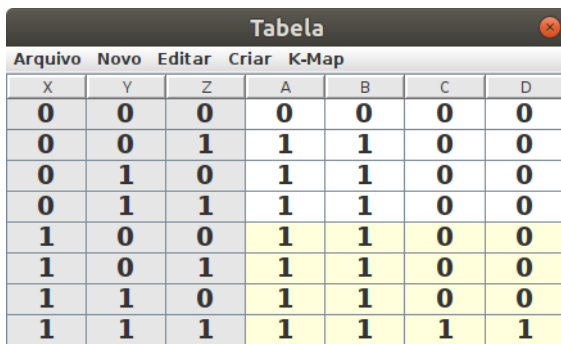
A primeira coluna da Figura 11 apenas repete as duas portas lógicas básicas que vimos na lista completa de todos os 16 circuitos possíveis. A segunda coluna da mesma figura implementa as portas no punho usando apenas portas NOT, AND e OR. Já a terceira coluna faz isso de novo, mas usando apenas portas NAND de duas entradas, enquanto a quarta coluna usa apenas portas NOR de duas entradas.

A tabela verdade gerada pelo Digital mostra que os circuitos em cada coluna são de fato equivalentes.

O curso “NAND to Tetris”² tem a frase “Deus nos deu a NAND e podemos construir toda a lógica a partir dela” justamente por ser possível construir um computador inteiro usando

²<https://www.nand2tetris.org/>

Figura 9: Exemplo de tabela verdade



X	Y	Z	A	B	C	D
0	0	0	0	0	0	0
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	1	0	0
1	0	0	1	1	0	0
1	0	1	1	1	0	0
1	1	0	1	1	0	0
1	1	1	1	1	1	1

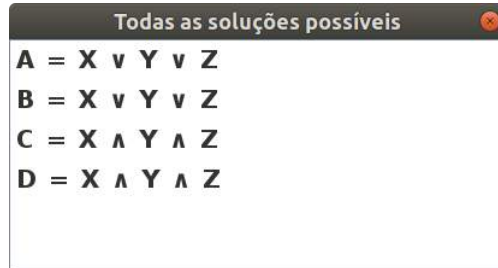
apenas esta porta. Mas por que NANDs e não NORs? Na verdade, o primeiro produto a usar circuitos integrados foi o Apollo Guidance Computer (AGC) ³, que levou o homem à Lua em 1969, e foi construído usando 4100 circuitos integrados, cada um dos quais tinha duas portas NOR de 3 entradas. Nos circuitos CMOS (*Complementary Metal Oxide Semiconductor*), que é a tecnologia mais usada nos chips atuais, as portas NAND são preferidas.

Os circuitos “o3” e “a4” da Figura 11 são uma demonstração das Leis de De Morgan. George Boole popularizou a ideia de que elas foram descobertas por Augustus De Morgan, mas na verdade elas já haviam sido observadas por William de Ockham séculos antes e até por Aristóteles. Se tanto as entradas quanto a saída de uma porta AND forem invertidas, ela passa a funcionar como um OR, e se forem invertidas as entradas e saídas de uma porta OR ela funciona como um AND. Isso pode ser usado para simplificar circuitos em alguns casos.

O circuito “x2” da Figura 11 mostra uma estrutura muito importante que chamamos de “soma de produtos”. A saída é mostrada na coluna “xor1” na Figura 12 (que é exatamente igual à outras 3 colunas xor) e o detalhe interessante é que o AND de cima do circuito corresponde ao 1 mais de cima da tabela enquanto o AND de baixo é responsável pelo 1 de baixo. Podemos implementar um circuito para qualquer tabela verdade assim, com um AND para cada 1 da

³https://en.wikipedia.org/wiki/Apollo_Guidance_Computer

Figura 10: Equações de relé



Todas as soluções possíveis

$$A = X \vee Y \vee Z$$

$$B = X \vee Y \vee Z$$

$$C = X \wedge Y \wedge Z$$

$$D = X \wedge Y \wedge Z$$

tabela (com algumas entradas possivelmente invertidas) e a saída de todos os AND indo para um OR que gera o resultado final. Como os AND e o OU podem ter qualquer número de entradas, isso funciona para tabelas verdade de qualquer tamanho.

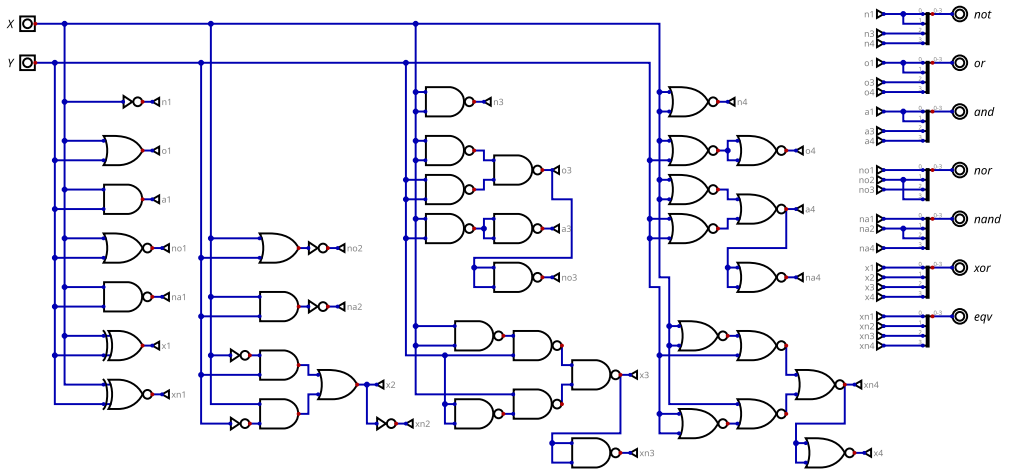
Apesar de não ser prático enumerar todas os circuitos combinacionais de 3 entradas (256), 4 entradas (65536) ou mais, podemos ter certeza que conseguiremos implementar qualquer um deles dado sua tabela verdade usando a soma de produtos.

Um circuito combinacional interessante com 3 entradas é o circuito de votação, também conhecido como “threshold logic” em inglês. Sua saída é o valor que a maioria de suas entradas tiverem. Para um número par de entradas haveria a possibilidade de empate, mas para 3, 5, 7 ou mais entradas a saída é sempre bem definida.

No Digital podemos criar um circuito novo e no menu “Análises” opção “Sintetizar” ele mostra uma tabela verdade com 3 entradas e a saída sempre zero. Podemos mudar os zeros em todas as linhas onde duas ou mais entradas são um e ver a equação do circuito de votação na forma de soma de produtos.

A Figura 13 mostra o resultado, mas a equação correspondente só tem 3 produtos com 2 elementos cada enquanto a tabela verdade tem 4 uns no resultado. O circuito precisaria, em princípio, e quatro portas NAND de 3 entradas cada uma. E uma porta OU de 4 entradas. Mas a Álgebra Booleana oferece uma simplificação. Se temos a soma $ABC + AB!C$ podemos separar o elemento comum e escrever isso como $AB(C + !C)$. A soma de um sinal com seu inverso é sempre 1 e o produto de qualquer coisa com 1 é a própria coisa. Por isso podemos

Figura 11: Portas lógicas básicas



usar um AND de 2 entradas no lugar de dois ANDs de 3 entradas.

Mas se formos simplificando a tabela desta forma não vamos chegar em uma equação tão reduzida quanto a mostrada na figura. Para isso usamos uma ferramenta chamada de Mapa de Karnaugh como mostra a Figura 14. Neste mapa a tabela verdade é reorganizada com a saída na forma de um quadrado ou retângulo, dependendo do número de entradas. Cada saída é posicionada de modo que seus vizinhos diferem em apenas uma entrada. Em seguida marcamos grupos de 1s vizinhos. Procuramos marcar todo o mapa até que não haja nenhum 1 sem fazer parte de um grupo (mesmo que seja um grupo só com ele) e sempre tentamos criar os maiores grupos possíveis. A primeira e última linha são consideradas vizinhas assim com a primeira e última coluna, de modo que grupos podem sair por um lado e continuar no outro (como o túnel no jogo do Pac-Man).

A grande vantagem do Mapa de Karnaugh é que mostra quando um 1 pode ser usado por

Figura 12: Tabela verdade para portas lógicas básicas

Tabela																													
Arquivo	Movo	Editar	Criar	K-Map																									
x	y	not3	not2	not1	not0	or3	or2	or1	or0	and3	and2	and1	and0	nor3	nor2	nor1	nor0	rand3	rand2	rand1	rand0	xor3	xor2	xor1	xor0	eqv3	eqv2	eqv1	eqv0
0	0	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	0	1	1	0	0	0	0	0	1	1	0	1	1	1	1	1	1	1	1	0	0	1	1
1	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0
1	1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

Figura 13: Tabela verdade do circuito de votação

*Tabela			
A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$Y = (A \wedge C) \vee (A \wedge B) \vee (B \wedge C)$

mais de um grupo. Isso reduz o número de portas AND necessárias e também reduz o número de entradas de cada AND. No caso a saída correspondente à entrada 111 pode ser combinada com cada uma das outras saídas um e precisamos de um AND a menos.

Na ferramenta tabela verdade podemos usar o menu “Criar” com a opção “Circuito” para obter a Figura 15. Observando a tabela verdade podemos ver que se a entrada A for ligada permanentemente em 0 o circuito funcionará como AND(B,C) e se A for sempre 1 o circuito se transforma em OR(B,C). Isso mostra que para ser universal o circuito de votação só falta poder implementar o NOT.

Nem todas as democracias são perfeitas e alguns sinais podem ter votos que valem mais que os outros. Uma maneira de se implementar isso é ligando o mesmo sinal em mais de uma entrada. Num circuito de votação de 7 entradas, por exemplo, A poderia ser ligado em 4 delas, B em duas e C e D em uma cada um. Ai A teria um peso de 0,57 (57% dos votos), B um peso de 0,29 e os outros dois 0,14 cada um. Com certas tecnologias é possível ter o mesmo efeito

Figura 14: Mapa de Karnaugh do circuito de votação



de maneira muito eficiente e inclusive permitir pesos negativos, o que resolve o problema de como implementar o NOT.

É este tipo de circuito que as redes neurais artificiais implementam, inspiradas nos neurônios naturais que são como a natureza implementa computação. Como acabamos de ver isso é uma solução universal capaz de fazer qualquer coisa que as portas lógicas fazem. Mas no resto do livro apenas usaremos as portas lógicas e deixaremos redes neurais e circuitos de votação de lado.

Somadores

Computadores digitais manipulam representações de números na forma de dígitos. Em representações posicionais cada dígito tem um peso diferente baseado na sua posição. Um número decimal como 729, por exemplo, tem o mesmo valor que $700 + 20 + 9$ que é igual a $7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$.

No menu “Componentes” do simulador Digital podemos encontrar um valor constante em “Conexões”. As opções avançadas para este tipo de componente podem ser vistas na Figura 16, incluindo o formato do número. Acabamos de descrever o “decimal” e por enquanto vamos ignorar “decimal com sinal”, “ASCII” e “Ponto fixo”. O que muda nas outras opções é a base

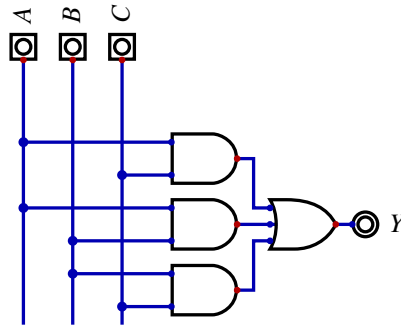


Figura 15: Circuito de votação

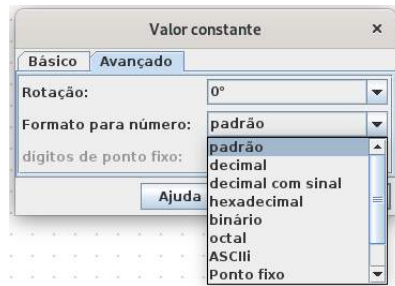
de cada representação numérica. Na fórmula do 729 a base é 10, no caso do hexadecimal a base é 16, para números binários a base é 2 e em octal a base é 8.

O número representado por 1001 em binário é $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ que é nove. A base 10 usa dígitos de 0 a 9, a base 8 de 0 a 7 e a base 2 apenas 0 e 1. Como a base 16 precisa de dígitos além do 9 foram criadas várias convenções mas a mais popular é usar A, B, C, D, E e F para indicar 10, 11, 12, 13, 14 e 15 respectivamente.

Um problema é que o número binário 101 (cinco), o octal 101 (dez), o decimal 101 (cento e um) e o hexadecimal 101 (duzentos e cinquenta e sete) parecem exatamente iguais. Na linguagem C o octal seria escrito como 0101 e o hexadecimal como 0x101. O C não tem representação binária. Na linguagem de descrição de hardware Verilog estes números seriam escritos como 'b101, 'o101, 'd101 e 'h101 respectivamente.

A vantagem da representação decimal é que as pessoas estão acostumadas com ela. A vantagem da representação binária é que com apenas dois dígitos fica fácil usar as portas lógicas que já vimos, bastando repetir os circuitos para cada dígito. Um problema dos números binários é que precisam ser traduzidos de e para decimais para serem usados pelas pessoas. Isso não é uma tarefa muito complicada mas acrescenta etapas em todos os programas. Outro problema dos binários é que eles possuem muitos dígitos e estes não tem muita variação de modo que é muito fácil as pessoas lerem errado. As representações octal e hexadecimal

Figura 16: Representações de números



são triviais de se converter de e para binário (basta agrupar de 3 em 3 ou de 4 em 4 bits respectivamente) e representam os mesmos números com bem menos dígitos. O uso do octal já foi mais popular no passado quando existiam máquinas de 18 bits ou de 60 bits para seus números, mas o hexadecimal domina atualmente com as máquinas são quase que exclusivamente de 8, 16, 32 ou 64 bits.

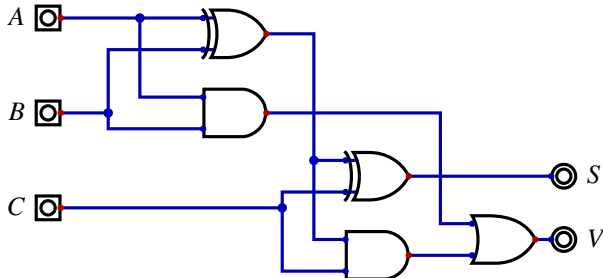


Figura 17: Somador

Enquanto $1 + 1$ é um na Álgebra Booleana, é dois na aritmética. Na representação binária isso é 10. Se usarmos sempre dois dígitos para a resposta teremos 00 para $0 + 0$ e 01 para

$0 + 1$ ou $1 + 0$. O dígito da direita é um XOR das entradas enquanto o da esquerda é um AND das entradas, duas portas que já vimos antes. Vamos chamar o dígito da direita de “soma” e o da esquerda de “vai um”. Para somar números com múltiplos dígitos precisamos levar em conta os vai um gerados pelas somas dos dígitos menos significativos. A Figura 17 mostra que combinado dois pares de portas XOR e AND (chamamos cada par de “meio somador” já que dois deles formam um somador completo de um bit) e mais uma porta OR para combinar os dois vai um parciais num só resultado.

Este jeito de somar dois bits e um vai um é bem didático, mas entre os sinais de entrada e a saída V tem um caminho que tem que passar por 3 portas seguidas. Podemos usar a tabela verdade e mapas de Karnaugh para converter este circuito para a forma de soma de produtos onde só existem dois níveis de portas lógicas entre todas as entradas e as saídas (mas três níveis se precisarmos inverter algumas entradas).

Para somar números de 4 bits podemos repetir este circuito 4 vezes e ligar o V de cada circuito no C do que lida com bits logo à esquerda. O V do circuito mais da esquerda fica sendo o vai um geral do bloco ou o quinto bit da soma. O circuito do bit mais à direita poderia ser um meio somador, mas por uniformidade vamos deixar todos os circuitos iguais e considerar a entrada C deste como entrada vai um (“vem um”?) geral do bloco. Em inglês este tipo muito simples de somador é chamado de *ripple carry adder* pois quando os dados são apresentados nas entradas os sinais vai um se propagam da direita para a esquerda como se fossem uma cascata. Isso torna um somador de 16 bits duas vezes mais lento que um de 8 bits, e um de 64 bits oito vezes mais lento que o de 8 bits. Existem maneiras de melhorar isso, mas este assunto ocupa livros inteiros e não será mais abordado aqui.

A operação de subtração tem duas complicações em relação à soma: a ordem dos operandos faz diferença e os resultados podem ser números negativos. A tabela 13 mostra algumas alternativas para o uso de 3 bits para representar números negativos. Até agora estivemos supondo que os números binários são sempre positivos e, neste caso, os números binários 000 a 111 representam os valores zero a sete como na segunda coluna.

A representação sinal-magnitude usa o bit mais da esquerda para indicar se o número é negativo e os dois outros bits indicam o valor positivo (de zero a três no caso de dois bits). Um problema óbvio é que existem dois zeros diferentes. Isso atrapalha se quisermos comparar se dois números tem o mesmo valor. A vantagem é que se parece com o sistema usado para

binário	positivo	sinal magnitude	complemento de um	complemento de dois
000	0	+0	+0	+0
001	1	+1	+1	+1
010	2	+2	+2	+2
011	3	+3	+3	+3
100	4	-0	-3	-4
101	5	-1	-2	-3
110	6	-2	-1	-2
111	7	-3	-0	-1

Tabela 13: Números negativos

representações decimais. Mas o circuito para lidar com isso pode ficar complexo, tendo que testar o sinal toda hora e fazer coisas diferentes dependendo do resultado. Este sistema é usado para a mantissa de números de ponto flutuante no padrão IEEE 754.

Na representação de complemento de um o negativo de um número é obtido simplesmente invertendo todos os bits. O bit mais à esquerda continua indicando o sinal e ainda temos duas representações para zero. Computadores que usavam complemento de um incluíram o UNIVAC 1101, CDC 160, CDC 6600, LINC, PDP-1 e UNIVAC 1107. Ao longo dos anos 1960 este sistema acabou completamente superado pelo complemento de dois. Para se obter o negativo de um número no sistema complemento de dois, além de se invertermos todos os bits somamos um ao resultado. Como mostra a tabela 13, o complemento de dois só tem uma representação para o zero mas tem mais números negativos que positivos. A grande vantagem desta representação é a simplificação dos circuitos pois os números podem simplesmente serem somados como se fosse da coluna dos positivos e o resultado fica certo. Somando os binários 001 e 101 deve dar 110 no caso dos positivos, pois seria $1 + 5 = 6$. Os mesmos bits na coluna de complemento de dois seria $(+1) + (-3) = (-2)$ que também está correto.

A unidade aritmética de 4 bits da Figura 18 pode somar dois números ou subtrair B de A gerando o complemento de dois de B e somando com A. Quando o sinal de controle sub é 0, os bits de B são usados diretamente. Se sub é 1 então todos os bits de B são invertidos, o que

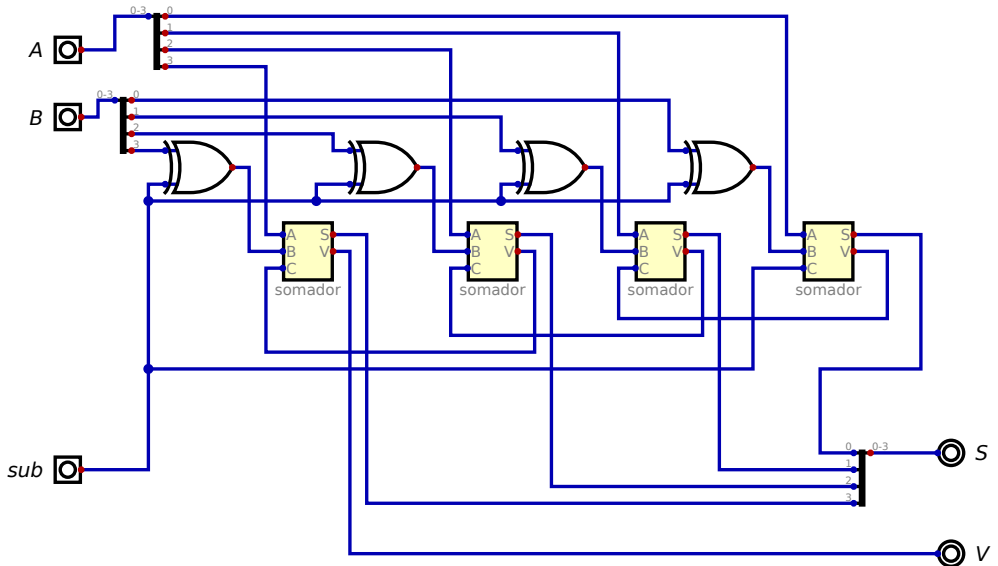


Figura 18: Unidade Aritmética de 4 bits

seria o complemento de um. Mas sub também serve de vai um de entrada do bit mais à direita de modo que mais um é somado quando sub é 1 gerando o complemento de dois de B.

A figura é um pouco mais confusa do que poderia ser pois a convenção de ter entradas à esquerda e saídas à direita conflita com a convenção dos bits menos significativos ficarem à direita. Por isso cada saída V precisa ser ligada à entrada C do bloco à sua esquerda dando uma volta desajeitada.

Multiplexadores

Em todos os circuitos vistos até agora os sinais podem estar ligados a qualquer número de entradas mas a apenas uma única saída. Se fossem ligadas duas saídas no mesmo sinal haveria

um conflito sempre que uma quisesse gerar um 0 e a outra um 1.

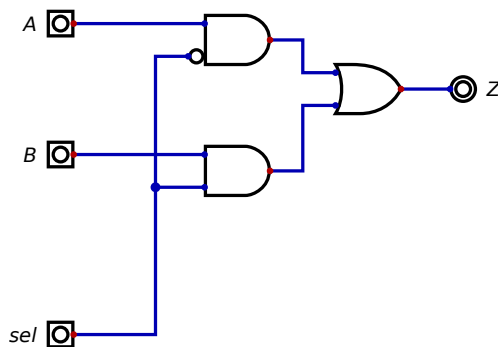


Figura 19: Multiplexador de 2 sinais

Muitas vezes é necessário que uma entrada seja ligada a uma saída em determinado momento e a uma saída diferente em outro momento. Algum sinal de seleção precisa indicar qual das duas saídas deve ser usada. O circuito da Figura 19 faz exatamente isso. O sinal Z repete o que está em A quando a seleção é 0 e repassa B quando a seleção é 1.

Existem outras alternativas para combinar sinais, como os barramentos de alta impedância ou os sinais de “coletor aberto” (também conhecidos como “wired and”), mas focaremos exclusivamente nos multiplexadores nos projetos deste livro. Note que o multiplexador tem a forma soma de produtos e já mencionamos que este tipo de circuito pode ser usado com qualquer número de entradas. Podemos, então, criar multiplexadores com mais entradas desde que o sinal de seleção tenha um número suficiente de bits para indicar todos os sinais de entrada. A Figura 20, por exemplo, usa um sinal de seleção de 3 bits e consegue multiplexar oito entradas diferentes.

Multiplicador

O produto da Álgebra Booleano é exatamente o mesmo da aritmética no caso de números de apenas um bit: a porta AND. Para multiplicarmos um número decimal N por 729, por

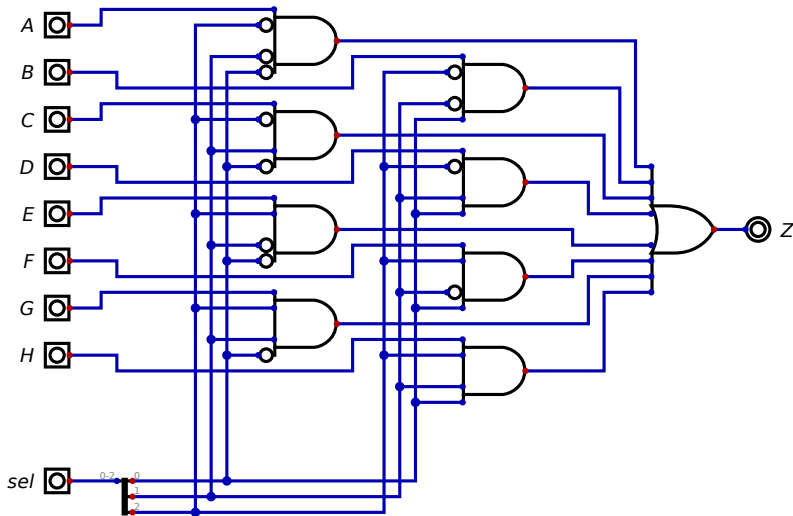


Figura 20: Multiplexador de 8 sinais

exemplo, podemos multiplicar N por cada um dos dígitos e somar os resultados dando $N \times 7 \times 10^2 + N \times 2 \times 10^1 + N \times 9 \times 10^0$. Se trocarmos a ordem dos termos em cada produto podemos ter $N \times 10^k$ com k variando de 2 a 0. Isso é apenas o próprio N deslocado k dígitos à esquerda e com os dígitos extras preenchidos com zeros. Dai precisamos multiplicar isso pelo dígito correspondente. Isso é um pouco complicado para dígitos decimais, mas para dígitos binários é apenas a operação AND como já dissemos.

A Figura 21 implementa diretamente esta idéia para o caso de duas entradas de 4 bits cada uma. Cada fileira de portas AND é o produto de um bit da entrada B por todos os bits da entrada A deslocada à esquerda. Como os bits novos à direita são sempre zero e a soma com zero não muda o valor, não são necessários somadores para os bits novos. Mesmo com esta simplificação o número de somadores é proporcional ao quadrado do número de bits dos operandos. Dá para ver que um multiplicador para números de 16, 32 ou 64 bits vai ser

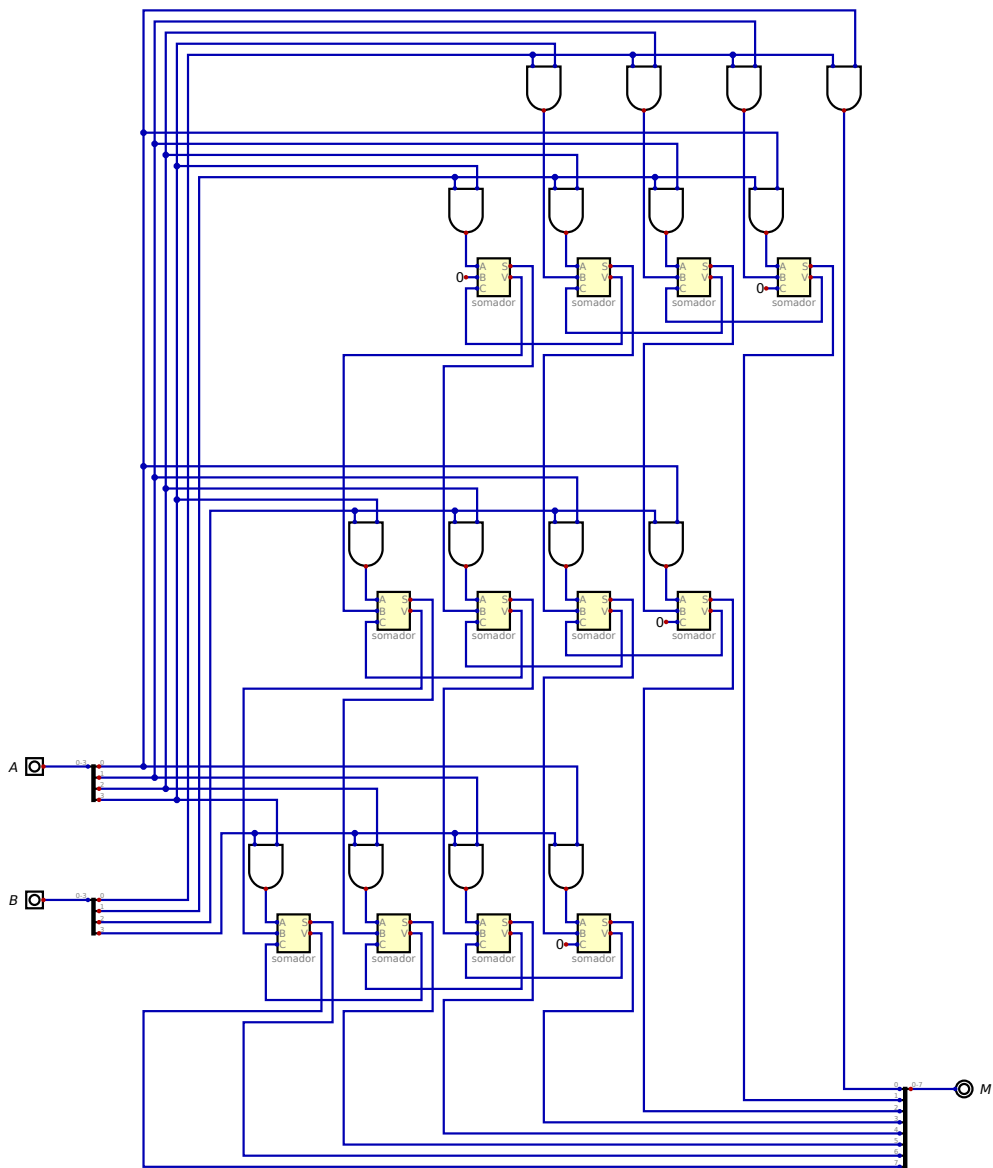


Figura 21: Multiplicador de 4 por 4 bits

enorme. Também vai ser muito lento pois os vai um precisam cascatear da direita para a esquerda em cada nível e também de um nível para o nível logo abaixo.

Uma coisa que simplifica este circuito é que enquanto o deslocamento à esquerda por uma variável usa muitas portas lógicas (dá para fazer com um multiplexador por bit, por exemplo), o deslocamento por um valor constante é só ligar fios da maneira correta sem nenhuma eletrônica.

O circuito da figura nem leva em conta números negativos. Para obtermos os resultados corretos onde um ou os dois operandos são negativos seria necessário umas modificações que não iremos mostrar aqui. Enquanto a soma de dois números gera resultados com um bit à mais que os operandos, o resultado da multiplicação tem um número de bits que é a soma dos números de bits dos operandos (o dobro se os operandos forem do mesmo tamanho). Alguns processadores conseguem usar um par de registradores para guardarem o resultado completo e alguns outros tem instruções separadas com MulHigh e MulLow para selecionar qual metade do resultado deve ir para o registrador de destino.

Um possível circuito de divisão seria essencialmente o oposto deste. O operando A seria usado como um valor inicial e em cada nível, do circuito o operando B seria comparado com isso e, se for menor, seria subtraído. Em cada nível do circuito B seria deslocado um bit para a direita. O circuito para comparar dois números binários para ver se um é menor que outro é a maior complicação do divisor. Enquanto o multiplicador tinha somadores e simples ANDs para cada combinação de bits, o divisor tem um comparador de magnitude, um subtrator e um AND. Os resultados dos comparadores de cada nível do circuito vão formando os bits do resultado da divisão. O resultado pode ser um valor fracionário com o dobro de bits dos operandos (se estes forem do mesmo tamanho) ou um par de valores inteiros (resultado e resto) com o mesmo número de bits dos operandos. Neste segundo caso muitos processadores usam instruções separadas, como DIV e REM, para definir o que deve ser guardado no registrador de destino.

Memórias só de leitura - ROM

Um circuito que seja o oposto do da Figura 20 é chamado de “decodificador”. Ele tem o mesmo sinal de seleção de 3 bits, mas apenas uma entrada e oito saídas. O valor da entrada aparece na saída selecionada enquanto as demais saídas ficam todas em zero.

Os circuitos de memória usam um decodificador para converter um número binário representando o endereço em linhas individuais de seleção de palavras. A memória mais simples é a ROM (memória de apenas leitura). Na parte da esquerda da Figura 22 dá para ver um decodificador com sinal de seleção de dois bits que envia sua outra entrada para uma das quatro linhas horizontais.

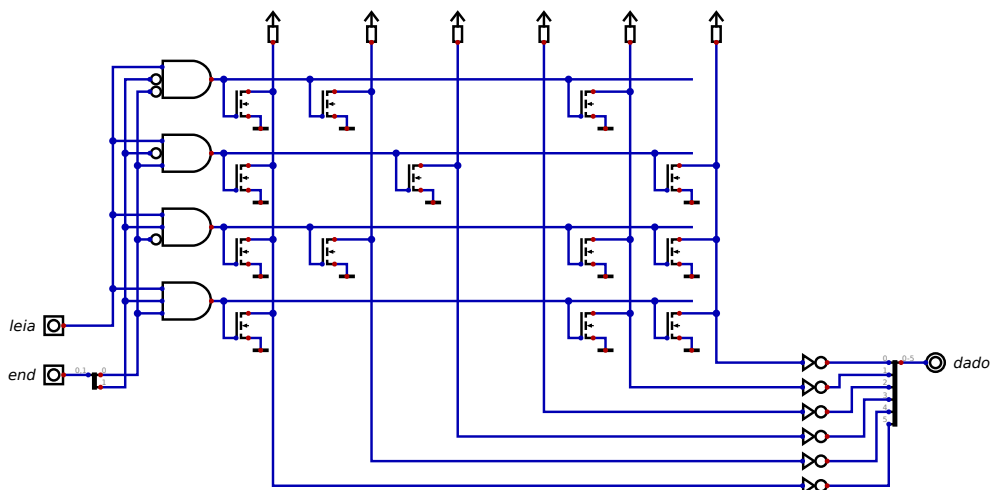


Figura 22: ROM com 4 palavras de 6 bits cada uma

Os resistores ligados à alimentação combinados com os transistores e os inversores formam portas OR, uma para cada coluna. Este circuito, então, está na forma de soma de produtos que já vimos algumas vezes. Gerando a tabela verdade (Figura 23) vemos a correspondência direta entre valores 1 nas saídas e a presença de transistores ligando as linhas às colunas. Se o sinal *leia* não estiver ativo a saída mostra só zeros. O conteúdo de uma ROM é definido durante os seu projeto e não pode ser alterado depois. Os circuitos integrados são fabricados por uma série de etapas que envolvem um processo chamado “fotolitografia” onde uma máscara (imagem em uma superfície transparente) é projetada no chip que está coberto por um material

sensível à luz. Os fabricantes de ROM costumam incluir todos os transistores possíveis no projeto de modo que todos os clientes possam usar as mesmas máscaras para reduzir o custo, e apenas uma máscara que liga os transistores à linhas de seleção de palavra precisam ser diferentes de um cliente para o outro. Por isso é possível encontrar o termo “mask ROM” em textos mais antigos.

O conteúdo da ROM da Figura 22 são as letras “R”, “I”, “S” e “C” no código DEC SIXBIT (que é o código ASCII menos 32 e truncado para 6 bits).

Mesmo uma única máscara por cliente é um custo bastante elevado e por isso os fabricantes criaram uma ROM com todos os transistores presentes onde uma tensão bem mais alta que a normal poderia queimar os transistores não desejados usando um aparelho “programador” do próprio cliente ao invés do conteúdo ser definido durante a fabricação. Infelizmente estas PROMs (de *Programmable ROM*) não podem ter seus transistores restaurados depois de queimados. Qualquer alteração implica em jogar fora a PROM antiga e comprar uma nova.

Isso foi resolvido com a criação de um circuito que usa cargas elétricas para ligar ou desligar os transistores das linhas de seleção de palavra. A programação continua sendo com pulsos de tensão bem mais altos que o normal, mas se tornou possível voltar ao estado inicial expondo o chip à luz ultravioleta por um certo tempo. Para isso estas EPROMs (*Erasable Programmable Read Only Memory*) vinham encapsuladas com uma pequena janela de quartzo de modo que a luz possa alcançar os transistores. Como a luz normal (especialmente a do Sol) tem um pouco de ultravioleta, criou-se o costume de cobrir a janela de quartzo com uma etiqueta colante para o uso normal.

Mais tarde foram criadas as EEPROM (*Electrically Erasable Programmable ROM*) que usavam outros tipos de pulsos para apagar os bits ao invés da luz. A maior evolução veio com as memória Flash que são as mais usadas atualmente e que também podem ser gravadas e apagadas eletricamente. As EEPROMs ainda são usadas em aplicações que precisam de apenas uns poucos bytes onde uma Flash seria um exagero.

A PROM é um circuito de soma de produtos onde a soma é definida pelo cliente. Existiria alguma vantagem em ter a soma fixa mas permitir que os produtos sejam alterados? Este tipo de circuito foi lançado no fim dos anos 1970 com o nome de PAL (*Programmable Array Logic*) e foi bastante usado até o início dos anos 1990. As PALs precisavam ser queimadas como as PROMs, mas depois surgiram versões que podiam ser apagadas eletricamente. Quando vários

Figura 23: Tabela verdade da ROM

*Tabela									x
Arquivo	Novo	Editar	Criar	K-Map					
leia	end1	end0	dado5	dado4	dado3	dado2	dado1	dado0	
0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	1	0	
1	0	1	1	0	1	0	0	1	
1	1	0	1	1	0	0	1	1	
1	1	1	1	0	0	0	1	1	

deste tipo de circuito eram combinados em um só chip com uma ligação configurável entre eles o resultado foi chamado de CPLD (*Complex Programmable Logic Device*) e estes ainda são usados. Infelizmente as empresas nem sempre são consistentes no uso do nome CPLD e isso cria um pouco de confusão.

Circuitos Sequenciais

Enquanto os circuitos combinacionais dependem exclusivamente dos valores atuais das entradas, os circuitos sequenciais dependem também dos valores passados das entradas. Podemos continuar usando as tabelas verdade mas como elas só mostram os valores dos sinais para determinado instante iremos depender mais das formas de onda daqui em diante. Estas formas de onda são gráficos com os diferentes sinais separados no eixo Y e o tempo no eixo X avançando da esquerda para a direita. Cada sinal ocupa uma faixa vertical sendo a parte mais baixa da faixa correspondente ao valor 0 e a parte mais alta ao 1. Isso no caso de sinais de um único bit. Para sinais com mais bits a faixa pode ser dividida de maneira diferente.

Uma vantagem das formas de onda é que se parecem com o que é mostrado por instrumentos de medição como osciloscópios e analisadores lógicos. Isso facilita a comparação de simulações com os circuitos no mundo real.

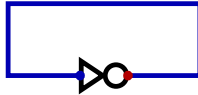


Figura 24: Circuito paradoxal

O circuito ultra simples da Figura 24 é um paradoxo. Se o sinal de entrada for 0, então a saída deveria ser 1 por causa do inversor. Mas a saída e a entrada são o mesmo sinal! E se a entrada é 1 então a saída deveria ser 0 que também é uma contradição. Se pedirmos para o Digital simular isso ele se recusa. Ele sugere a simulação passo a passo mas isso também dá erro. Este paradoxo se parece com os argumentos que o Capitão Kirk do seriado original “Jornada nas Estrelas” usava para explodir computadores vilões. Será que se for construído este circuito vai explodir?

A mensagem de erro do Digital diz “Oscilação aparente”. Precisamos levar em conta que o inversor não é infinitamente rápido. Depois que a entrada muda leva um tempo para a saída mostrar o inverso. Se a entrada for para 0, só um tempo depois a saída irá para 1 levando a entrada também para 1 e vai levar mais outro tempo para a saída agora ir para 0 e assim infinitamente. Esta alternância muito rápida entre 1 e 0 é a oscilação que o Digital não quer simular. Mas se for construído o circuito veremos o que chamamos de “onda quadrada” de alta frequência em um osciloscópio (que, como o nome diz, foi feito justamente para observar oscilações). Se construirmos vários destes circuitos, cada um vai oscilar numa frequência um pouco diferente. E mesmo num só circuito a frequência muda dependendo da temperatura e da tensão de alimentação.

No mundo real, se a entrada do inversor está indo de 0 para 1 o sinal passa por todas as tensões intermediárias. Nenhuma transição pode ser instantânea. Ao mesmo tempo, a saída estaria indo de 1 para 0 e também passando por todas as tensões intermediárias. Isso significa que existe uma tensão na entrada que gera exatamente a mesma tensão na saída. Dado o atraso do inversor a chance de sistema parar nesta situação é absurdamente pequena. Mas não é zero. Se isso ocorrer dizemos que o sistema está “meta estável”. Mesmo que isso ocorra, qualquer ruído no sistema vai forçar o circuito a voltar a oscilar.

Será que podemos obter metade da frequência com dois inversores onde a saída do segundo

vira a entrada do primeiro? A idéia básica é boa, mas se a entrada do primeiro inversor for 0, sua saída será 1 e a saída do segundo inversor será 0. Estes valores continuarão assim para sempre. O circuito não oscila. Por outro lado, se a entrada do primeiro inversor for 1 a sua saída será 0 e o segundo inversor soltará 1. Isso também continuará assim eternamente. Este circuito é conhecido como “bi-estável” pois existem duas situações diferentes nas quais ele fica parado.

Para termos oscilações precisamos de um número ímpar de inversores. Quanto mais inversores menor será a frequência de oscilação. Este circuito é conhecido como oscilador em anel e é muito usado para testar as características dos produtos de uma fábrica de chips. A variação de sua frequência pode ser usada para medir temperaturas se forem tomados certos cuidados.

O anel com um número par de inversores é bi-estável. Mas se um anel com dois inversores tiver um cristal de quartzo ligado entre eles existe uma única frequência na qual ele pode oscilar. Esta frequência depende das características mecânicas e elétricas do cristal muito mais do que das características dos inversores. Num computador ou videogame circuitos assim são os únicos que são feitos para oscilar. Em todo o resto do sistema oscilações serão consideradas erros de projeto.

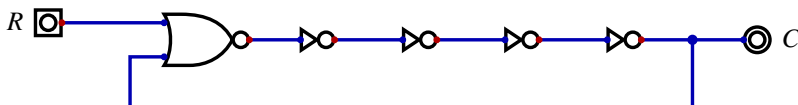


Figura 25: Oscilador em anel com inicialização

Além da oscilação, uma coisa que complica a simulação de circuitos como o da Figura 24 é que no instante inicial o valor do sinal é desconhecido. E o inverso de um valor desconhecido também é desconhecido e a simulação não consegue sair disso. No circuito real tanto faz se o sinal começa em 0 ou em 1 ele oscila de qualquer jeito. O que muda é em que instantes o sinal é 0 ou é 1 (a fase da onda quadrada) mas a maioria dos projetos não dependem disso.

A Figura 25 mostra a solução para isso - um sinal de inicialização. Trocando o primeiro NOT por um NOR é possível forçar C a ficar parado em 0 colocando 1 em R. Assim que R for para 0 o circuito começa a oscilar outra vez e teremos controlado a fase da onda quadrada em

C.

Latches e Flip-flops

Se fizermos uma variação da Figura 25 com apenas um NOR e um NOT teremos um circuito bi-estável onde podemos controlar o estado inicial. Infelizmente este será também o estado final. Será como um circuito lógico feito de dominós onde depois que eles caem eles não levantam mais de modo que não servem para mais de um cálculo. Se trocarmos também o segundo NOT por um NOR teremos um segundo sinal capaz de forçar o circuito para o outro estado. O circuito da Figura 26 é conhecido como “flip flop” básico pois pode ser empurrado para qualquer um dos dois estados e ele fica lá indefinidamente até um outro sinal de controle empurrá-lo para o outro estado. Acionar o mesmo sinal de controle várias vezes seguidas não tem nenhum efeito.

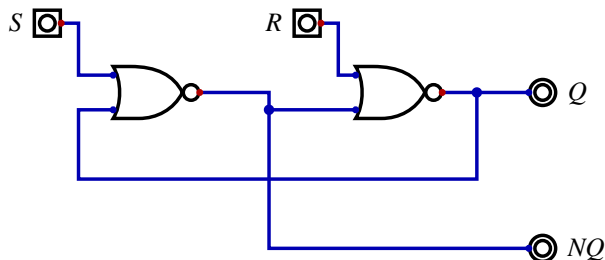


Figura 26: Flip Flop básico

Este é um circuito sequencial fundamental pois consegue “lembrar” que sinais de controle recebeu no passado. É uma memória de um bit. Também é conhecido como flip flop RS em função dos sinais de controle R (Reset - leva Q a 0) e S (Set - leva Q a 1). A saída invertida NQ (não Q) é gerada de graça e podemos aproveitá-la para eliminar inversores em outras partes do circuito como nas entradas de somas de produtos.

Uma maneira alternativa de se lembrar um bit é o circuito da Figura 27, chamado “latch” em inglês e que poderia ser traduzido por “engate”. Ele usa o multiplexador de 2 para 1 que já vimos na Figura 19. Quando a entrada C (Clock) é 1 a saída Q reflete a entrada D (Data).

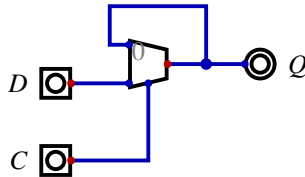


Figura 27: Latch

Assim que C vai para 0 o último valor de Q fica circulando até C voltar para 1. Chamamos a transição de um sinal de um nível para outro de “borda”, sendo a borda de subida a passagem de 0 para 1 e a borda de descida a transição de 1 para 0. Dizemos que Q amostrou D na borda de descida de C. Como Q mostra todas as alterações em D enquanto C for 1, dizemos que este é um latch transparente. Uma maneira de eliminarmos isso é ligar dois latches em seguida com o C de um em certo sinal e o C do outro no inverso do mesmo sinal. Agora quando um está transparente o outro não está. Q indicará o valor de D apenas no instante da borda de subida do sinal C.

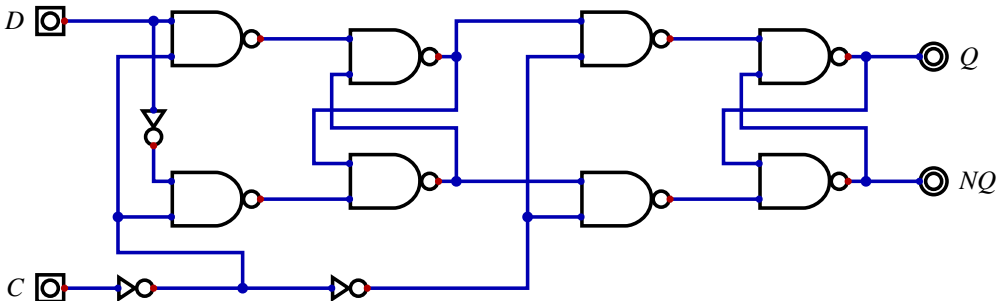


Figura 28: Flip Flop sensível à borda de subida do relógio

O mesmo truque dos dois latches se aplica aos flip flops, como mostra a Figura 28. O fato de estarmos usando portas NAND no lugar dos NOR da versão anterior não importa segundo

De Morgan (desde que todas as inversões sejam feitas de maneira coerente). Agora a saída Q fica sempre com o mesmo valor até ocorrer uma borda de subida em C , quando Q passar a ter o valor que D tinha naquele instante.

Existem vários outros tipos de flip flop, mas estes já são suficientes para dar uma idéia de como bits podem ser guardados dentro de um computador.

Memórias de Acesso Aleatório - RAM

Até agora só falamos em guardar um único bit. Será que não seria possível combinar a ROM com o flip flop para criar um circuito capaz de lembrar N palavras com W bits cada uma? Poderíamos usar um endereço para selecionar uma palavra para ser alterada ou lida.

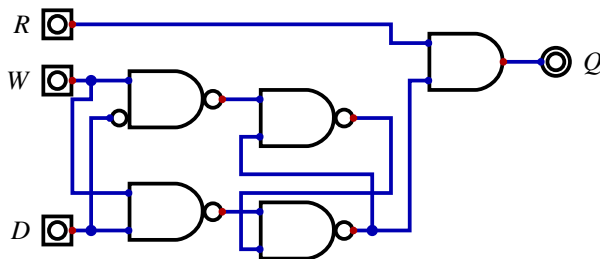


Figura 29: Flip Flop modificado para ser bit de RAM

A Figura 29 mostra uma versão do flip flop que é facilmente integrado com outros em blocos maiores. A saída é sempre 0 até que o sinal R seja acionado e quando W é 1 o valor de D fica registrado no flip flop. Como mostra a Figura 30 os sinais R e W são os mesmos para todos os bits de uma mesma palavra. O decodificador seleciona uma linha baseada no endereço de dois bits fornecido e quando for acionada a entrada escrita, todos os W da linha são ativados. Da mesma forma ao ser acionada a entrada leitura, todos os R da linha são acionados.

Todas as saídas Q dos flip flops controlam um transistor para colocar seu valor ou não na sua coluna. Cada coluna gera um bit na saída de dados. As linhas que não forem selecionadas

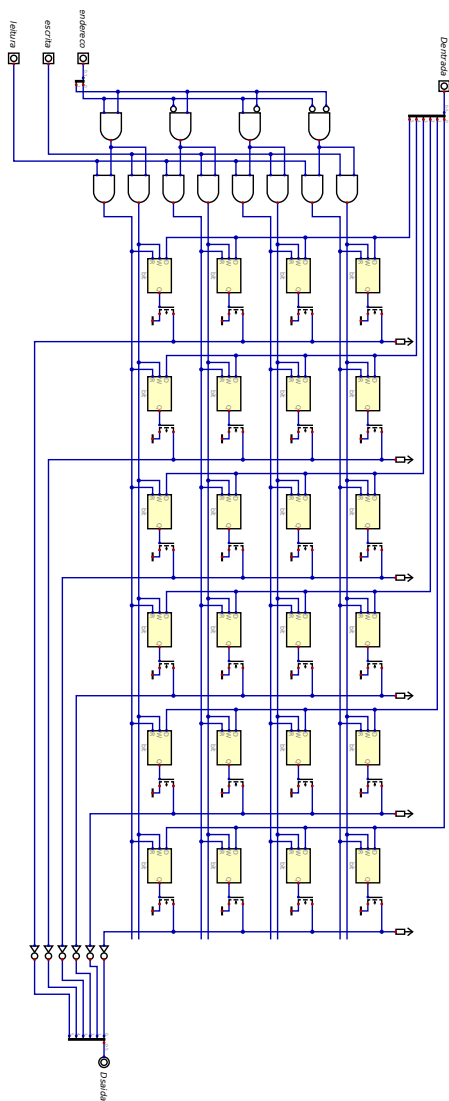


Figura 30: RAM de 4 palavras de 6 bits cada

não ativam seus transistores e, se o sinal leitura não estiver acionado então nenhum flip flop fornece dados e a saída vai mostrar 0.

As entradas D de cada flip flop da mesma linha vão diretamente para um dos bits da entrada Dentrada. Apesar de todas as linhas receberem o mesmo dado, apenas quando o sinal escrita for acionado os flip flops da linha indicada pelo endereço guardarão o dado recebido.

No passado foram usadas várias tecnologias diferentes para a construção de memórias pois os transistores (e as válvulas antes deles) eram caros demais para se incluir mais que uns poucos flip flops no projeto. Várias destas tecnologias (tanques de mercúrio, tambores magnéticos, fitas de todos os tipos) tinham o inconveniente que os dados só podiam ser lidos na mesma ordem em que foram gravados. Estas são as memórias de acesso sequencial. Já as palavras da memória da Figura 30 podem ser gravadas e lidas em qualquer ordem pois os bits da entrada endereço podem receber qualquer valor. Por isso é uma memória de acesso aleatório (Random Access Memory).

Cada flip flop da Figura 29 precisa de uns 24 transistores mais o transistor extra que liga sua saída à coluna. Com bem mais cuidado dá para fazer a mesma função com apenas 6 transistores e a economia é fundamental se um projeto tem milhões ou bilhões de bits de memória. Este tipo de memória é chamada de estática (SRAM) pois uma vez armazenado o bit ele permanecerá lá enquanto o circuito receber alimentação. Se a força for desligada todos os dados serão perdidos. Existe uma tecnologia alternativa onde são usados apenas um transistor e um capacitor por bit. Infelizmente estas memórias dinâmicas (DRAM) perdem o dado depois de uns poucos segundos. A solução é ficar lendo todos os dados e gravando de volta pois cada escrita devolve a carga elétrica ao seu nível máximo. Outra limitação é que estas DRAMs são fabricadas em linhas especializadas e bem diferentes das usadas para se fabricar os demais componentes do sistema. Isso torna difícil integrar DRAMs nos mesmos chips que os processadores. Por isso a maioria dos sistemas tem pequenas SRAM nos chips dos processadores e uma memória principal externa e bem maior feita de DRAMs.

Contadores

Os contadores são uns dos mais importantes circuitos sequenciais. Eles podem indicar quanto tempo falta para algo acontecer, quantos elementos já foram recebidos numa entrada, qual é a posição de um periférico e muitas outras coisas.

Felizmente para um circuito tão útil a sua implementação é relativamente simples. Um registrador armazena a contagem atual. Chamamos de registrador o circuito que usam um flip flop para cada bit de um mesmo dado. Ligando a saída do registrador a uma das entradas de um somador e o valor contante 1 à outra entrada, basta conectar a saída do somador à entrada do registrador. Cada vez que chegar um novo pulso do relógio o valor no registrador será aumentado por 1. Seria possível também fazer o contador reduzir o valor por 1 a cada relógio.

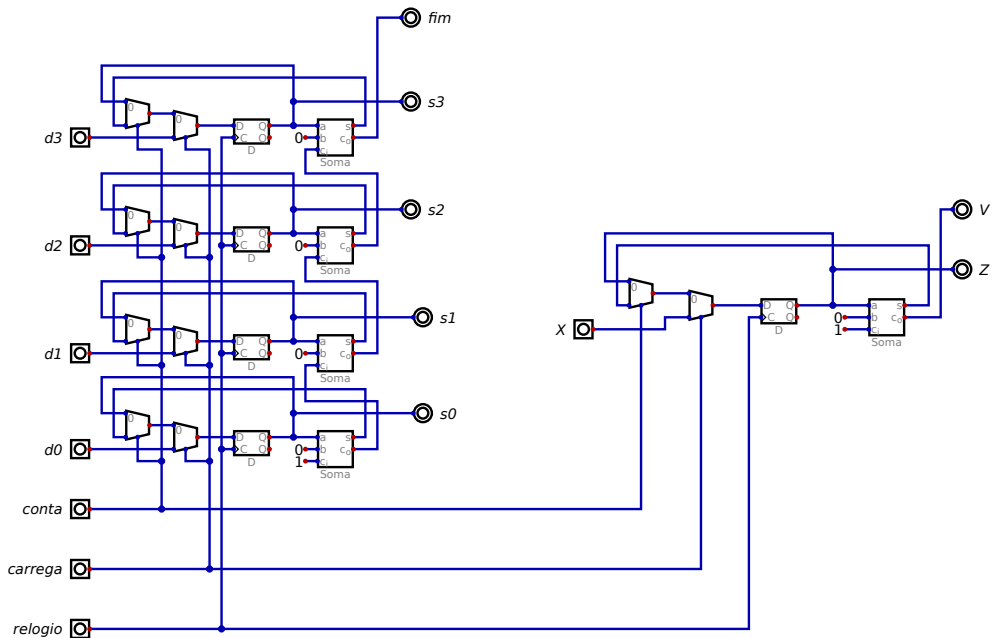


Figura 31: Um contador de 4 bits e outro de 8 bits

A Figura 31 mostra um exemplo de um contador de 4 bits na parte da esquerda. Além dos flip flops (registradores) e somador já mencionados, foram usados ainda dois multiplexadores

para permitir novos sinais de controle. O selecionado pelo sinal **conta** pode ligar ou a saída do somador como dissemos ou a saída do próprio registrador à entrada do registrador. No primeiro caso o valor será incrementado a cada pulso do **relógio** mas no segundo caso o valor ficará parado, como se o **relógio** estivesse sendo ignorado.

O segundo multiplexador é controlado pelo sinal **carrega** e permite que um valor definido externamente seja usado para inicializar o contador. Como este multiplexador vem depois do outro, o sinal **carrega** funciona independentemente do sinal **conta**. Tanto o **carrega** quanto o **conta** só fazem alguma coisa na próxima borda de subida do **relógio**. Dizemos que estes sinais são “síncronos”. É possível projetar contadores com alguns sinais de controle assíncronos que atuam independentemente do **relógio**.

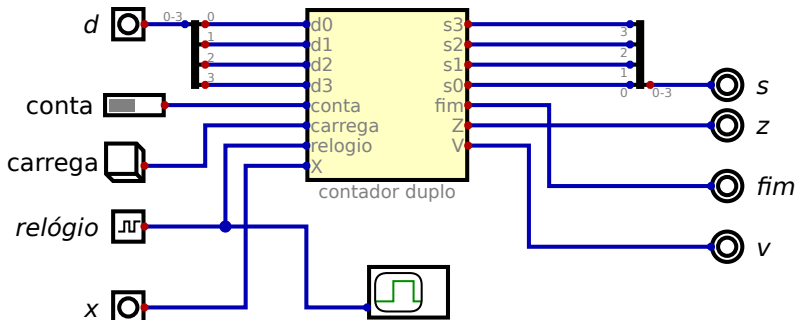


Figura 32: Circuito de teste dos dois contadores

Enquanto o circuito da esquerda da Figura 31 tem quatro cópias do que foi descrito acima (como sempre para circuitos digitais - um para cada bit dos números), o da direita parece ser de apenas um bit. Mas a legenda da figura diz que é um contador de 8 bits. O Digital implementa componentes parametrizados. Uma porta AND, por exemplo tem um parâmetro que indica o número de entradas. Isso é normalmente 2 mas pode ser alterado pelo usuário permitindo mais entradas. O próprio desenho da porta AND muda um pouco conforme o número de entradas. Mas ela tem um segundo parâmetro, que é o número de bits. Normalmente isso é 1 e todos os circuitos que vimos até agora não alteram isso. Se este parâmetro for mudado a aparência

continua exatamente igual. Mas agora o sistema funciona como se houvessem múltiplas cópias do AND e as entradas e saídas só podem ser ligadas a outros circuitos com o mesmo número de bits. Infelizmente o Digital não altera a aparência dos sinais quando eles representam mais de um bit (estes sinais conjuntos são muitas vezes chamados de “barramentos” ou “dutos”). Outras ferramentas de desenho de esquemáticos usam coisas como cores diferentes, linhas mais espessas ou pequenas marcas com um número indicando quantos sinais são representados pela linha. No Digital não dá para saber simplesmente olhando o desenho. Na ferramenta dá pedir informações sobre os componentes e as portas de entrada e saída para saber quantos bits cada um representa. E qualquer tentativa de simular um circuito onde elementos com números de bits diferentes estão ligados gera uma mensagem de erro.

Fora um contador ser de 4 bits e o outro de 8, eles deveriam ser iguais. Especialmente já que estão ligados no mesmo relógio e sinais de controle. Para confirmar isso foi criado o circuito de teste mostrado na Figura 32. Os sinais d0 a d3 são combinados em uma única entrada d para facilitar a comparação com a entrada x. Da mesma forma s0 a s3 são combinados para podermos estudar s lado a lado com z. O sinal carrega está ligado num componente que simula um botão de contato momentâneo. O sinal conta está ligado num “dip switch”, uma chave que pode ser ligada ou desligada e fica nesta posição. O sinal relógio está ligado em um componente especial que gera uma onda quadrada. Um parâmetro deste componente é a frequência desta onda e se o Digital deve tentar simular isso em tempo real. O normal desta opção é não, mas neste teste foi configurado para sim com uma onda quadrada de 1 Hz (um ciclo por segundo). Isso torna o circuito lento o suficiente para que o botão possa ser pressionado no momento desejado.

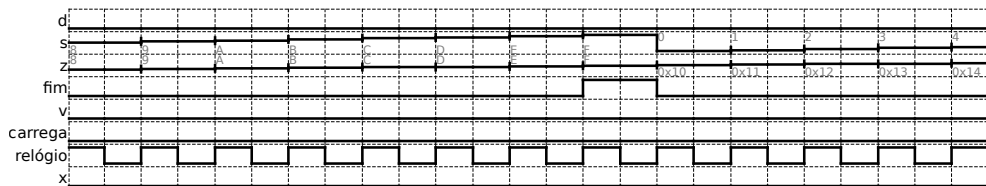


Figura 33: Formas de onda dos dois contadores

Um outro componente novo é o osciloscópio que está ligado no relógio e durante a simulação abre uma nova janela onde são mostradas as formas de onda capturadas. A Figura 33 é um exemplo destas formas de onda. Dá para ver bem a onda quadrada do relógio. A maioria dos sinais alternam entre 0 e 1 mas s e z tem valores indicados por números hexadecimais (isso é configurável. A notação hexadecimal do Digital é um pouco inconsistente: se o número tem apenas dígitos 9 ou menores então um “0x” é colocado na frente para que o número não seja confundido com decimal. Mas se algum dígito é uma letra o número é mostrado sem o 0x). O nível do sinal é mostrado relativo ao maior valor encontrado na simulação, de modo que a saída dos contadores parece uma rampa. As entradas d e x também tem múltiplos bits mas nesta simulação os dois tem valor 0.

O detalhe mais interessante da simulação é que quando s chega ao seu valor máximo (F em hexadecimal, que é 15 em decimal) o sinal fim é acionado. No próximo ciclo de relógio o fim termina ao mesmo tempo que s volta para 0. Já para z o valor 0F não é o máximo e no ciclo seguinte ele vai para 11 em hexadecimal (16 decimal). Se a simulação for até z atingir FF o sinal v será acionado exatamente como foi o sinal fim e no ciclo seguinte z também irá para 0. Com um relógio de 1 Hz levaria mais de quatro minutos para o z chegar ao seu valor máximo, mas dá para apressar isso colocando um valor bem grande em x e acionando o botão carrega.

Na Figura 34 temos dois exemplos de como combinador dois contadores de 4 bits para formar um contador de 8 bits. No da esquerda o bit mais significativo do contador de baixo é invertido e isso é usado como relógio do contador de cima. Quando o contador de baixo passa de F para 0 o contador de cima avança 1.

No circuito da direita o sinal de fim do contador de baixo é usado para liberar o contador de cima para avançar um. Olhando as formas de onda da Figura 33 parece que o sinal fim termina ao mesmo tempo que o relógio sobe, mas na prática o fim ainda é 1 na subida do relógio e só baixa depois e aí o contador de cima já avançou um. Os sinais fim (“ovf” na figura) estão ligados aos pontos decimais dos mostradores de 7 segmentos para facilitar sua visualização durante a simulação.

Na tabela 11 mencionamos o nível de abstração de transferência de registradores (em inglês “*Register Transfer Level*” - RTL) e é comum falar que linguagens de descrição de hardware como Verilog e VHDL são RTL. Mas além de ser um nível o RTL é um estilo de projeto de hardware e estas linguagens podem ser usadas para outros estilos diferentes.

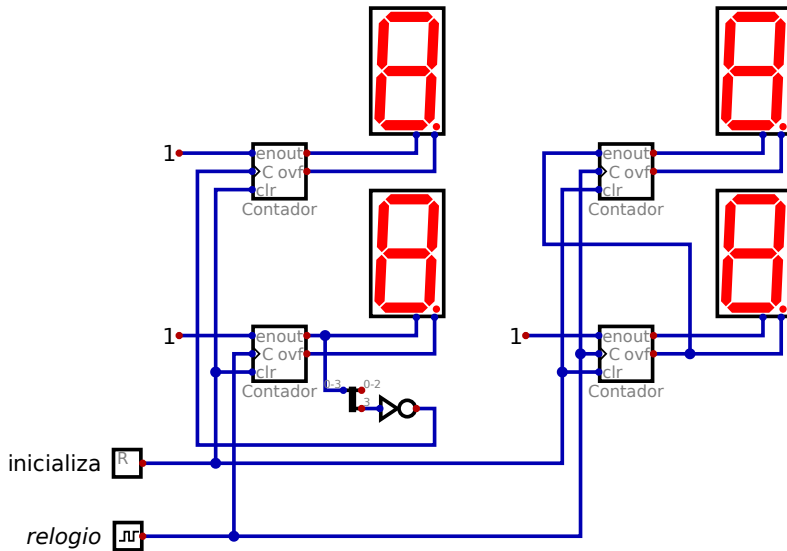


Figura 34: Dois estilos de expandir contadores

A Figura 34 ilustra bem isso, pois o circuito da direita usa o estilo RTL, porém o da esquerda não. Em um projeto RTL todos os registradores devem receber o mesmo sinal de relógio e se algum registrador não deve receber novos dados em determinado momento, então um sinal de habilitação não deve ser acionado, porém o relógio continua chegando. Isso pode ser visto no contador de cima do circuito da direita, enquanto o seu equivalente à esquerda está recebendo um relógio diferente do resto do sistema. Enquanto os dois dígitos da direita mudam ao mesmo tempo, na esquerda, o dígito de cima tem um pequeno atraso em relação ao de baixo. Neste caso a diferença nem pode ser percebida, mas se o estilo da direita for usado em um sistema suficientemente grande é quase certo que coisas vão acontecer que não parecem fazer sentido.

Ainda na Figura 34 o circuito da direita (RTL) parece ser mais simples, mas isso é em

parte em função dos pontos fortes e fracos do Digital. Trazer um sinal do meio de um grupo de sinais ocupa um certo espaço no esquemático, por exemplo. O RTL impõe restrições e é necessário um projeto mais cuidadoso para não se introduzir novos relógios. Mas o resultado é um sistema mais estável.

Existem muitas variedades de contadores além das opções já mencionadas aqui. Um sinal de controle pode decidir se o contador incrementa ou decrementa seu valor, por exemplo. Um contador de 4 bits que só vai até 9 pode ser útil em aplicações que querem mostrar resultados em números decimais.

Máquinas de Estados Finitos

Entre os vários modelos computacionais as máquinas de estados finitos (“*Finite State Machines*” ou FSM em inglês) estão entre as mais simples. Mas na teoria elas podem fazer tudo que qualquer outro computador pode fazer. Elas podem ser implementadas com um registrador para indicar o estado atual e um circuito combinacional que calcula o próximo estado baseado nas entradas e no estado atual e que também gera as saídas. Quando chega a próxima borda do relógio a máquina “salta” para o estado previamente calculado.

Existem várias notações diferentes para se descrever as FSM. Uma possibilidade é uma tabela com uma linha para cada estado e uma coluna para cada combinação possível de entradas. A alternativa implementada pelo Digital é um grafo onde círculos representam os estados e setas mostram as possíveis transições para o próximo estado. Dentro do círculo pode ser mostrado o nome do estado, o valor que o registrador usa para indicar este estado e os valores das saídas. Nas setas podem ser indicados os valores das entradas para que esta transição seja a escolhida (podem existir múltiplas setas saindo de um estado) e também os valores das saídas.

Quando as saídas dependem apenas do estado atual dizemos que a FSM é uma “máquina de Moore” (definida por Edward F. Moore em 1956). Se as saídas estão indicadas nas setas então dependem não apenas do estado atual mas também das entradas atuais e neste caso a chamamos de “máquina de Mealy” (definida por George H. Mealy em 1955). Cada alternativa tem suas vantagens. As máquinas de Mealy respondem mais rápido às mudanças nas entradas mas as de Moore são mais fáceis de se projetar.

Usado o editor de FSM do Digital podemos criar uma computação que determina se

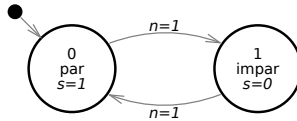


Figura 35: FSM para determinar par ou ímpar

uma entrada foi acionada um número para ou ímpar de vezes, como na Figura 35. Não são mostradas setas onde $n = 0$ e o sistema assume que o estado deve continuar o mesmo neste caso.

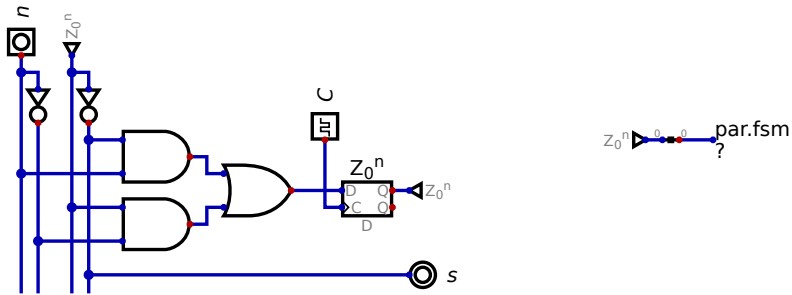


Figura 36: Circuito da FSM para determinar par ou ímpar

O Digital pode mostrar a tabela verdade da FSM e automaticamente gerar um circuito que a implementa. O circuito da Figura 36 é apenas uma das variações que o Digital oferece. Neste caso obtivemos essencialmente um contador de um bit. Entre os exemplos de FSM que o Digital pode gerar estão justamente os contadores. Geralmente eles são um bom ponto de partida e depois podem ser alterados para atender necessidades que os contadores normais não resolvem.

Enquanto teoricamente as FSM podem fazer qualquer computação, na prática elas tem os seus pontos fracos. Para a manipulação de números é necessário um número absurdo de estados e para guardar dados é o mesmo problema. A solução é combinar a FSM com um

circuito externo controlado por ela que faça contas ou tenha uma memória como as que já vimos. A famosa máquina apresentada por Alan Turing no seu artigo de 1936, por exemplo, é uma FSM que controla um leitor de fita. O leitor pode avançar ou retroceder uma posição na fita e pode gravar um símbolo na posição atual. A entrada da FSM é o símbolo lido da fita (que é infinita nas duas direções).

Multiplicador Sequencial

Um bom exemplo de como aproveitar as FSMs é um circuito que multiplique dois números. Já vimos uma solução na Figura 21 mas o circuito é enorme e lento. Podemos ter um circuito bem menor se dividirmos a tarefa em etapas a serem feitas umas depois das outras em ciclos de relógio diferentes.

O circuito da Figura 37 segue o estilo RTL onde todos os registradores estão ligados a um único sinal de relógio. Para reforçar esta idéia, os registradores foram desenhados alinhados verticalmente à direita da figura. O eixo horizontal da figura representa a passagem do tempo da esquerda para a direita. Em ilustrações de circuitos RTL existe a convenção de se dividir os registradores ao meio e mostrar à direita as entradas dos registradores e bem à esquerda as saídas dos mesmos registradores. O Digital não permite isso, então mostramos os registradores completos à direita e usamos os pequenos triângulos (“tuneis” no Digital) para fazer as saídas reaparecerem à esquerda. O meio da figura representa como os sinais são manipulados pela lógica combinacional e está marcada como “ciclo N”. As saídas dos registradores aparecem na região indicada como “ciclo N+1” pois terão validade depois da próxima subida do sinal de relógio. Já a região à esquerda de onde reaparecem as saídas foi designada “ciclo N-1” para indicar quando os sinais que a lógica combinacional está recebendo foram gerados. Se $N=7$, por exemplo, os sinais A, Q, M e assim por diante foram aqueles que foram gerados pela lógica combinacional no ciclo 6. E os sinais sendo gerados agora servirão de entrada para a lógica combinacional ao longo do ciclo 8.

Os dados neste circuito são de 16 bits de largura. Por isso o somador também é de 16 bits, assim como os registradores A, Q e M. O contador é de 16 bits e evita que a FSM de controle precise de estados diferentes para saber quais bits estão sendo processados. Quando o sinal fim é acionado sabemos que estamos processando o último bit. Se quisermos um multiplicador de números de 32 bits, basta aumentar o contador para 5 bits e os registradores, somador e outros

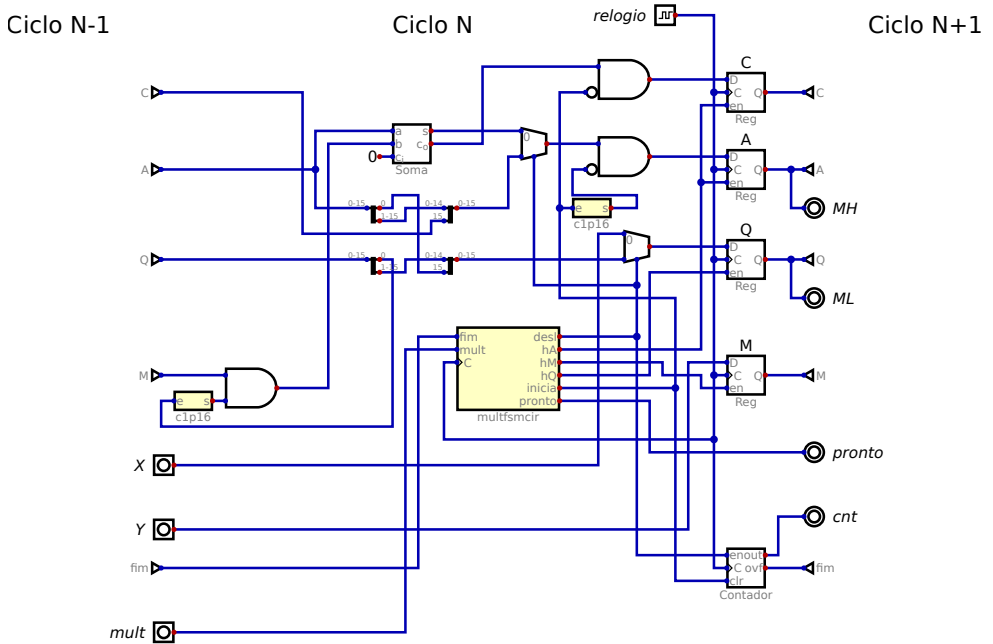


Figura 37: Multiplicador sequencial de 16 por 16 bits

blocos para 32 bits. Isso é o dobro do circuito enquanto um multiplicador combinacional precisaria de um circuito quatro vezes maior. O resultado vai levar o dobro do tempo para ficar pronto. Um registrador de um bit (C) guarda o vai um do somador.

A FSM precisa habilitar os registradores quando devem receber novos valores e usar multiplexadores para escolher de onde devem vir estes dados. Portas AND são usados para forçar C e A a carregarem inicialmente um valor 0 e outra porta AND pode forçar uma das entradas do somador a ser 0 ao invés de M. No caso do A e M estes ANDs trabalham com dados de 16 bits mas o sinal de controle é de apenas 1 bit. Foi criado um pequeno bloco

chamado `clp16` (converte 1 bit para 16 bits) que é trivial e só contém fios. Seu uso, aqui, é só para ocupar menos espaço no desenho.

Logo acima da FSM de controle tem uma fiação meio confusa. Ela envia C mais os 15 bits de cima de A para A, e o bit de baixo de A mais os 15 bits de cima de Q para Q. Chamamos isso de “deslocamento para a direita”. Já que o bit de baixo de Q foi separado para este deslocamento, aproveitamos para usá-lo para indicar se M deve ou não ser somado a A. Esta é a multiplicação de 16 bits por 1 bit.

Normalmente precisaríamos de quatro registradores de 16 bits, mas ao observarmos que um dos operandos perde um bit a cada vez que o resultado aumenta em um bit podemos fazer a parte baixa do resultado partilhar o mesmo registrador Q ocupado inicialmente pelo operando X.

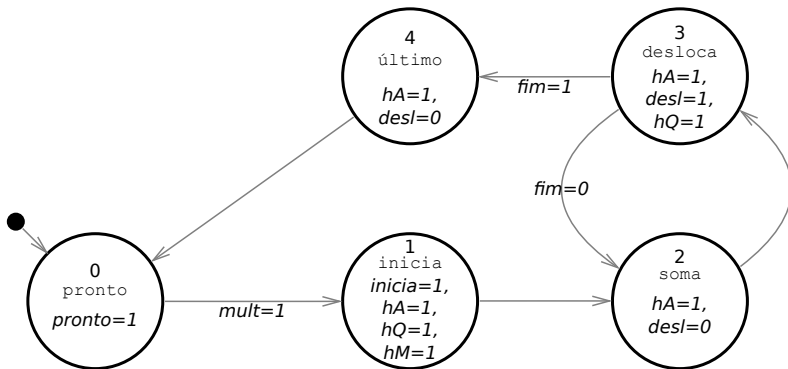


Figura 38: FSM que controla o multiplicador sequencial

Sabendo que sinais a lógica de controle precisa receber e que sinais ela precisa gerar para a lógica combinacional e habilitação dos registradores podemos usar o editor de FSM para criar a Figura 38. No estado inicial indicamos que o resultado está pronto e não fazemos mais nada. Quando o usuário colocar valores a serem multiplicados em X e Y e acionar o sinal `mult`, passamos para o estado `inicia` e deixamos de indicar pronto já que agora estamos ocupados.

Inicia zera C, A e o contador e guarda X e Y em Q e M respectivamente. Com sua tarefa

concluída a FSM pula para soma no próximo ciclo de relógio. Em soma fazemos a operação $A = A + (M \times Q_0)$ e pulamos para desloca. Em desloca A e Q são deslocados um bit para a direita e o contador avança. Normalmente a FSM volta para soma para cuidar do próximo bit, mas se fim indica que já deslocamos todos os bits pulamos para ultimo. Ultimo é exatamente igual a soma, mas depois pula para pronto para indicar que o resultado pode ser lido. O resultado é de 32 bits com A tendo os 16 bits de cima e Q os 16 de baixo.

O Digital gerou automaticamente um circuito para a FSM de controle do multiplicador (Figura 39). As figuras mostradas são o resultado final do projeto depois de corrigido os vários problemas nas versões iniciais. Os erros mais comuns são coisas acontecerem cedo demais ou tarde demais. A versão inicial da FSM não tinha o estado ultimo, por exemplo. Tentava perceber fim diretamente em soma. Mas fim só aparece quando o contador está habilitado e isso ocorre em desloca e não em soma.

Outro erro comum e relacionado com este é contagem errada. Isso é como o erro de programação onde valores são um a mais ou um a menos do que deveriam ser. Dependendo de quando o contador é inicializado, por exemplo, ele pode já ir para 1 na operação do primeiro bit e aí chegaria a 15 quando apenas o penúltimo bit está sendo operado. O oposto também ocorre muitas vezes.

Esta organização de um sistema em um fluxo de dados (“*datapath*” em inglês) e uma unidade de controle será usada nos processadores apresentados neste livro.

FPGAs

As CPLDs, mencionadas anteriormente, foram introduzidas pela Altera em 1985. Logo depois a Xilinx lançou uma alternativa chamada de FPGA (*Field Programmable Gate Array*). O nome é uma referência aos *Gate Arrays* (nome mais usado nos Estados Unidos, enquanto na Inglaterra *Uncommitted Logic Array* era mais popular). Este tipo de circuito tinha uma matriz de portas lógicas (NANDs, por exemplo) permitindo que todos os clientes usassem as mesmas máscaras para uma redução de custos. Apenas a máscara dos fios ligando as portas entre si eram específicas de cada projeto. É uma máscara diferente da usada nas ROMs mas o objetivo era o mesmo.

No caso das FPGAs a ligação entre os blocos precisa ser feita com circuitos reconfiguráveis e não por fios definidos por uma máscara. Isso torna desejável uma redução destas ligações.

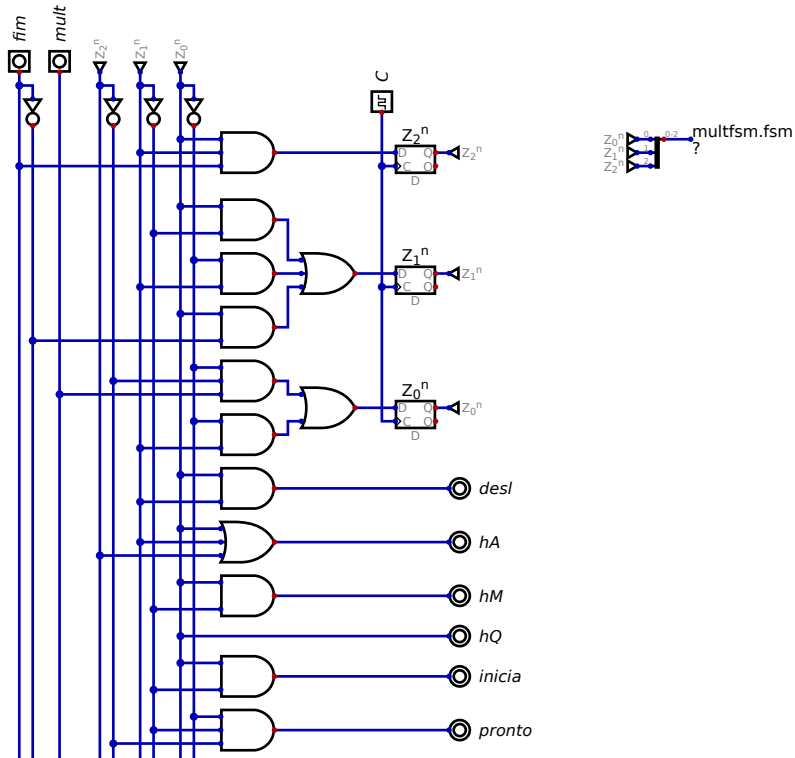


Figura 39: Circuito gerado da FSM de controle do multiplicador

Até seria possível criar uma FPGA em que cada bloco é uma porta NAND de duas entradas, mas isso exigiria muitas ligações. Por isso seria melhor ter um bloco básico maior e termos menos blocos.

Uma ROM pode implementar qualquer função lógica e uma RAM também, com a van-

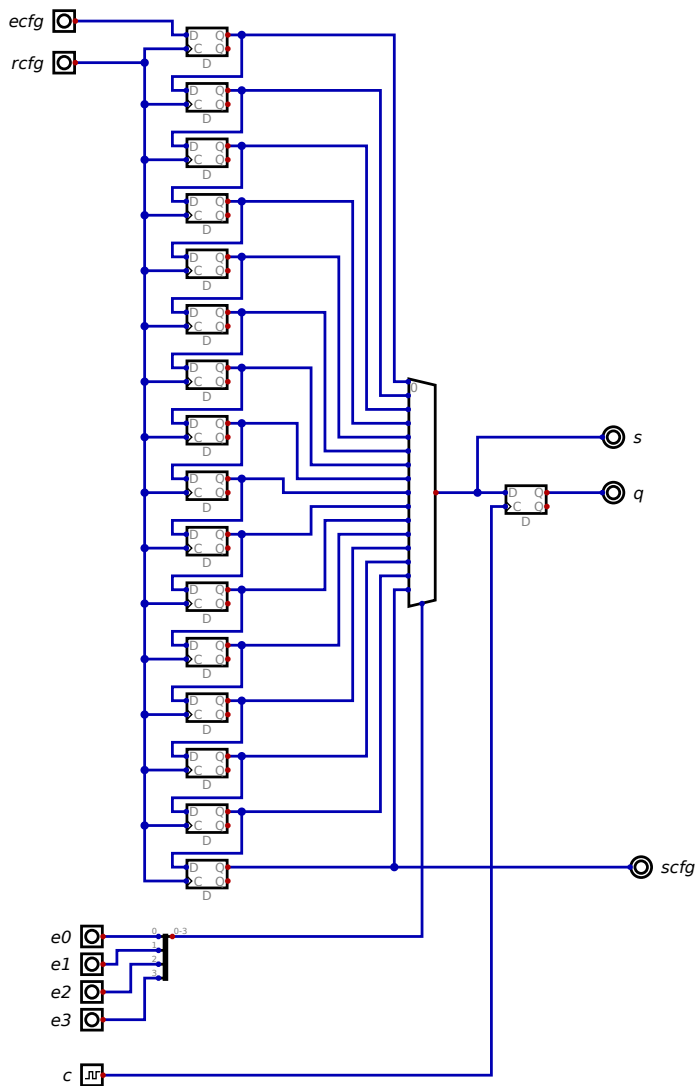


Figura 40: Circuito básico da FPGA (lookup table)

tagem de poder ter seu conteúdo alterado. A Figura 40 mostra uma RAM com 16 palavras de 1 bit cada uma. Com os valores corretos nos bits, este circuito implementa qualquer função lógica de até 4 entradas. Este circuito tem duas diferenças em relação à RAM da Figura 30 (além de ser 16x1 ao invés de 4x6). No lugar de um decodificador gerando sinais de seleção de palavras este circuito usa um multiplexador para levar o bit desejado à saída. A segunda diferença é que o endereço não é usado na gravação dos dados, mas estes são inseridos serialmente via 16 pulsos seguidos do sinal *rcfg*. Por estas diferenças normalmente se fala de *Lookup Table* (LUT) ao invés de RAM. Também é comum colocar o número de entradas, falando em LUT4 (16 bits), LUT3 (8 bits) ou até em LUT6 (64 bits).

A primeira FPGA tinha 64 LUT4 e até hoje LUT4 é o mais usado. Apenas algumas FPGAs mais caras usam tamanhos diferentes. Além dos 16 flip flops que guardam os bits que definem a função do bloco, um flip flop extra facilita a criação de circuitos sequenciais nas FPGAs. Na FPGA exemplo que estamos criando aqui, todos os flip flops extras tem o seu sinal de relógio ligados a um único pino. Uma FPGA real oferece mais opções de onde buscar o relógio, além de opções de inicialização do flip flop. Inicialmente as FPGAs usavam apenas LUTs, mas atualmente é popular ter alguns blocos para funções dedicadas como memórias, multiplicadores e até processadores completos.

Precisamos ligar as entradas das LUTs às saídas de outras LUTs ou aos pinos de entrada. A Figura 41 mostra como fazer isso com os mesmos blocos da LUT arranjados de maneira diferente. Dois multiplexadores permitem ligar as saídas a qualquer uma de oito entradas conforme indicado por 3 bits de configuração (6 bits no total para os dois multiplexadores). O conteúdo dos flip flops é inserido pelos mesmos dois sinais (*rcfg* e *ecfg*) que na LUT. Dá para ligar estes blocos e as LUT entre si via estas pinos para que um par de entradas no chip possam carregar todos os bits serialmente qualquer que seja o número de bits.

O circuito da Figura 42 combina quatro cópias do circuito anterior num bloco que tem duas saídas e duas entradas em cada um dos seus quatro lados. Tem também duas entradas extras para as saídas da LUT vizinha. E liga os pinos de configuração dos quatro blocos no sentido norte, oeste, sul e leste. Como cada bloco precisa de 6 bits para definir sua função, este circuito é configurado com 24 bits.

Cada bloco pode colocar em suas saídas os sinais da LUT com ou sem passar pelo flip flop. Ou pode colocar qualquer uma das duas entradas de qualquer um dos outros lados. A

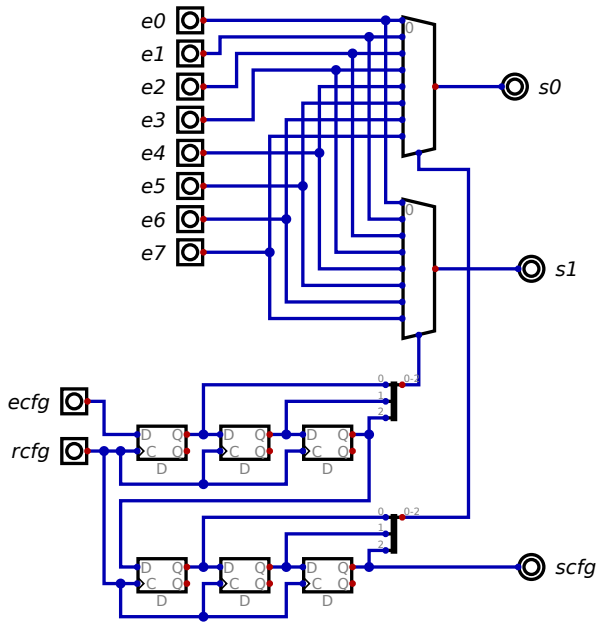


Figura 41: Circuito de roteamento em uma direção

Figura 42 mostra a ligação dos sinais de configuração e dos sinais vindos da LUT mas usa os túneis para evitar um cruzamento ilegível de sinais do meio da figura. Foi adotada a convenção que depois dos sinais da LUT vem os sinais do lado no sentido horário, depois do lado oposto e finalmente do lado no sentido anti horário.

A FPGA em si é uma matriz de pares de LUT e blocos de roteamento. A Figura 43 mostra um exemplo mínimo de 4 elementos organizados em duas linhas e duas colunas. As FPGAs reais tem blocos especiais para os pinos de entrada e saída mas neste exemplo os pinos foram diretamente ligados aos blocos de roteamento. Não haveria nenhuma dificuldade na criação de um circuito com 64 elementos (8x8) como a primeira FPGA e tamanhos ainda maiores

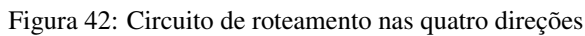


Figura 42: Circuito de roteamento nas quatro direções

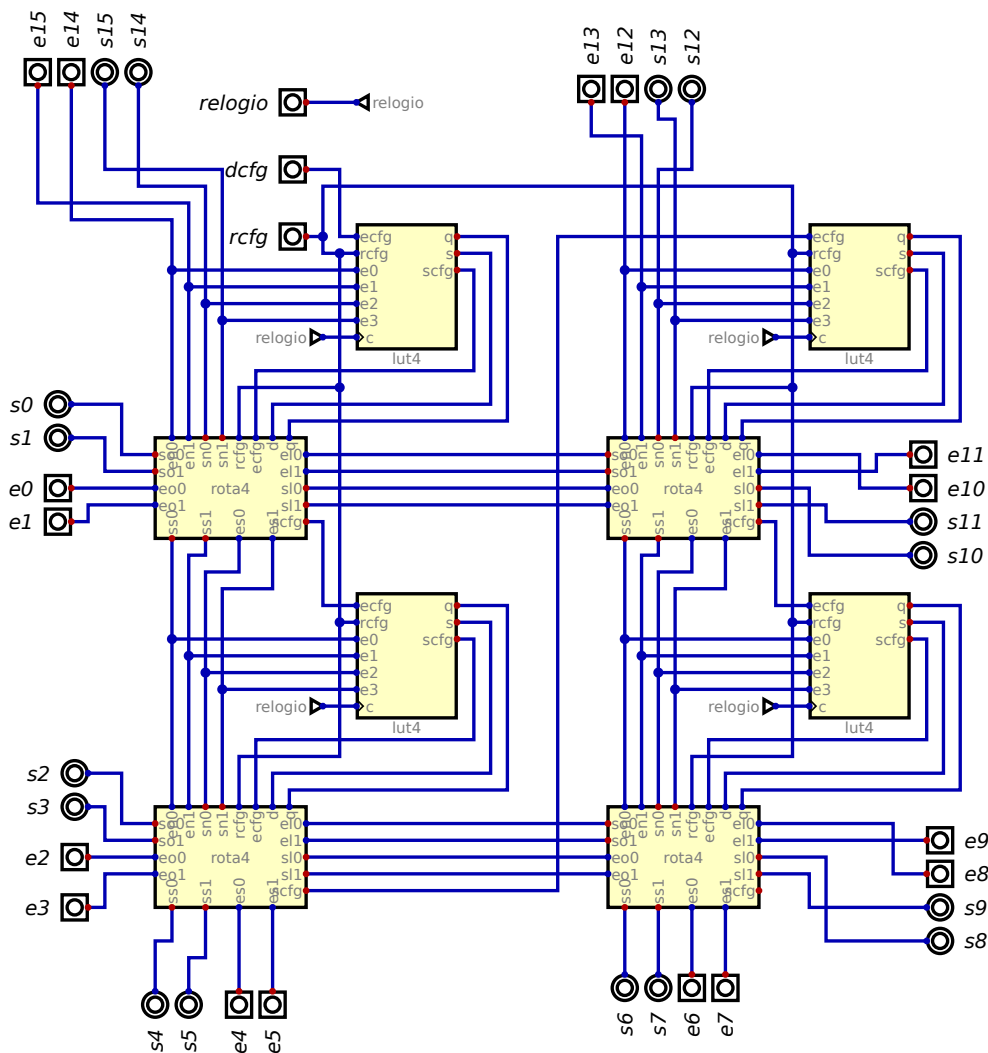


Figura 43: FPGA com 4 elementos (2x2)

podem ser criados em minutos. O tamanho reduzido foi escolhido apenas para ficar uma figura razoavelmente detalhada no livro.

Mesmo este tamanho pequeno precisa de 40 bits para cada par (16 para a LUT4 e 24 para o bloco de roteamento), o que dá 160 bits no total. É possível simular a FPGA e, pressionando as entradas `dcfg` e `rcfg` configurar o circuito e depois usá-lo. O `rcfg` deve ser pressionado duas vezes para cada bit (indo para 1 e depois voltando para 0). Seria melhor criar um circuito externo que leia os bits de uma ROM e configure a FPGA automaticamente. É assim que elas funcionam na realidade, geralmente incluindo o circuito que lê a ROM (ou Flash, atualmente). Toda vez que a FPGA é desligada ela perde todos os bits da configuração. Por isso este processo de carregar bits precisa ocorrer cada vez que a FPGA é ligada.

Se inserir manualmente os bits na FPGA é absurdamente tedioso, determinar qual valor deve ser cada bit para fazer a função que desejamos seria muito pior. Atualmente temos ferramentas (inclusive software livre) que fazem esta tarefa. Podemos começar com o próprio Digital e desenho o circuito que queremos e, depois que a simulação mostrar que está correto, podemos exportar como um arquivo texto na linguagem de descrição de hardware Verilog ou VHDL.

O Verilog pode ser transformado pelo programa Yosys, por exemplo, para vários formatos diferentes. Ele pode gerar um circuito equivalente mas feito exclusivamente de LUT4 ligadas entre si. Isso nos dá o número de LUT4 necessárias e os bits que precisamos para configurar cada uma.

Este resultado pode servir de entrada para um programa como VPR que, para poder cumprir sua tarefa, precisa de uma descrição detalhada da nossa FPGA. Ele associa cada LUT4 do circuito do Yosys (LUT lógica) a uma LUT4 real em uma posição definida dentro da FPGA (LUT física). Este posicionamento dos recursos (*placement*) visa tornar viável a etapa seguinte, que é o roteamento. O número de fios horizontais e verticais em cada parte da FPGA é limitado. No nosso exemplo se um projeto exigir cinco ou mais sinais em um trecho horizontal o VPR não vai conseguir configurar os blocos roteadores. Talvez movendo uma parte do projeto para uma LUT diferente reduza para quatro sinais neste trecho.

O último passo é converter a configuração gerada para um arquivo com os bits nos lugares certos. No nosso caso são quatro grupos de 40 bits, com cada grupo começando com 24 bits de roteamento seguidos de 16 bits para a LUT.

As FPGAs comerciais usam ferramentas próprias que escondem a maior parte das complicações. Em alguns casos são cobradas anuidades bastante elevadas, mas escolhendo produtos com FPGAs menores geralmente existem versões grátis (mas não livres) dos programas necessários.

Experimentos com vídeo

Pelo nome dá para perceber que “vídeo” é uma parte muito importante dos videogames. A idéia de que um aparelho de televisão poderia exibir imagens que não estavam sendo filmadas em algum outro lugar era chocante quando o Odyssey foi lançado. O truque era ver que sinal uma câmera geraria se estivesse filmando a imagem desejada e criar um circuito que gere exatamente o mesmo sinal.

Uma imagem na tela é uma matriz bidimensional de pontos coloridos, assim como o texto nesta página é uma matriz bidimensional de letras. Como poderíamos reproduzir a página com apenas um canal de voz com outra pessoa? Poderíamos ler o texto na ordem normal, talvez soletrando para evitar erros. Mas para realmente ficar igual precisaríamos falar coisas como “fim da linha” e “nova página”. Estes seriam os comandos de sincronização. Com eles a página começa no lugar certo e cada linha começa da forma correta.

O sinal de televisão precisava enviar todas as informações no mesmo canal: cor, brilho sincronismo horizontal e sincronismo vertical. Se a fonte das imagens está ligada à TV por um cabo dá para simplificar bastante dedicando um fio para cada informação. No padrão de monitores VGA temos 3 sinais de cor (vermelho, verde e azul) e fios para o sincronismo horizontal e vertical, por exemplo. Tem mais alguns sinais, mas não vamos estudá-los aqui.

O Digital tem um componente que compara dois números, dizendo se o primeiro é menor, igual ou maior que o segundo. Mas isso é uma complicação desnecessária se apenas queremos saber se são iguais. O circuito da Figura 44 indica se dois números de 12 bits são iguais. Um XOR de cada par de bits só deve ter zeros, o que pode ser verificado com um NOR.

O tempo de uma linha de vídeo pode ser dividido em quatro regiões:

1. a área ativa contém os pontos que devem aparecer na tela
2. o *front porch* é o tempo entre o fim do vídeo e o sinal de sincronização
3. o pulso de sincronização
4. o *back porch* é o tempo entre o fim do sinal de sincronização e o reinício do vídeo

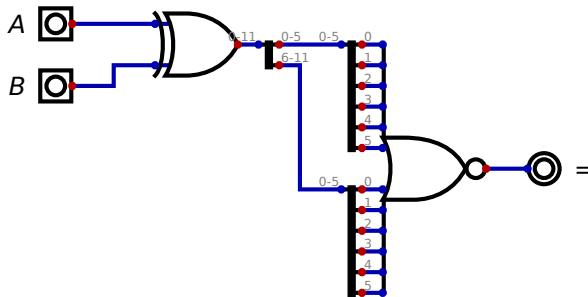


Figura 44: Comparador de igualdade de números inteiros

Da mesma maneira uma tela é dividida em regiões equivalentes, mas contando linhas ao invés de pontos. O contador estranho da Figura 45 não compara apenas com o fim da contagem (sinal fimcont) que volta o contador para zero, mas também observa 3 outros pontos de interesse (sinais **upix**, **inisinc** e **fimsinc**). A saída mostra indica a região ativa enquanto o pulso de sincronismo é gerado diretamente na saída sinc.

Um circuito simples de teste verifica o funcionamento do gerador de sincronismo como mostrado na Figura 46. São usados valores bem baixos para os parâmetros para que os sinais gerados repitam logo. O sinal fim foi problemático e garantir a largura exata do pulso de sincronismo foi complicado. O Digital tem um componente que simula um monitor VGA. Se ele recebe os sinais corretos ele mostra a imagem resultante em outra janela. Um sinal que ele precisa que o monitor verdadeira não tem é o relógio de pixel. A simulação não ocorre em tempo real. No lugar de 60 imagens por segundo, a janela é atualizada a cada poucos segundos. O Digital considera os sinais de sincronismo em relação ao relógio de pixel. E ele é muito mais exigente que um monitor real. Se o pulso de sincronismo estiver com um pixel de erro na largura uma mensagem de erro é mostrada e a imagem não aparece. Num monitor ou TV de verdade provavelmente o circuito funcionaria e daria para ver algo na tela.

A Figura 47 é nossa primeira tentativa de gerar uma imagem na tela. O monitor VGA simulado tem vários modos que ele implementa, mas o mais básico é com imagem de 640 pontos por linha, 480 linhas visíveis e taxa de varredura de 60 Hz. O bloco temporizador de

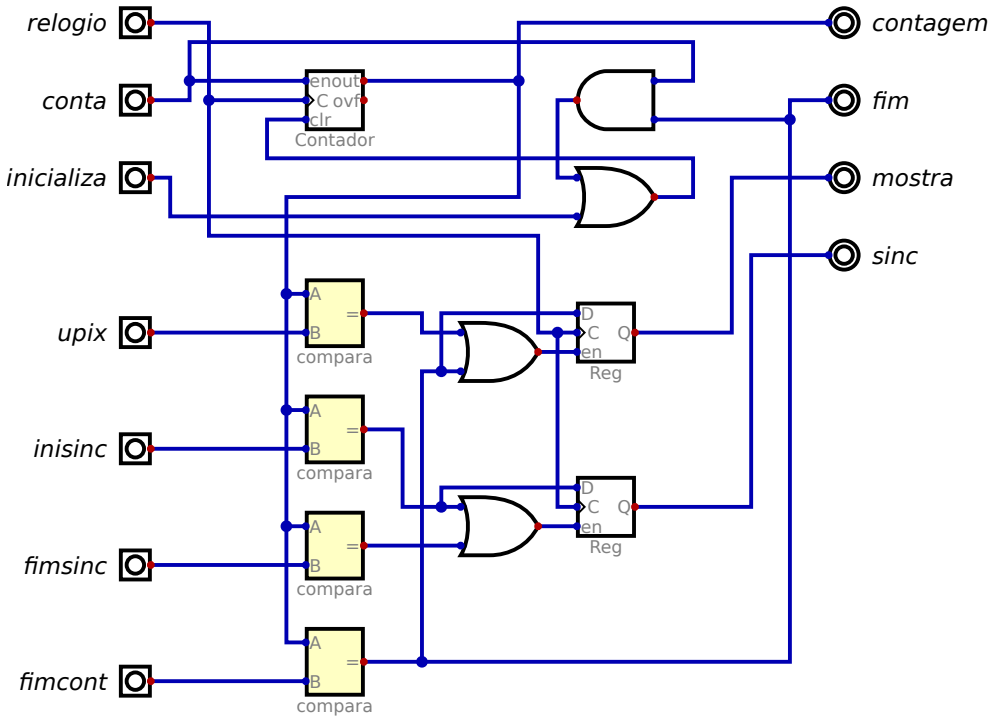


Figura 45: Gera sinais de sincronismo e de habilitação

baixo gera os sinais correspondentes à linha e suas entradas estão em pixels. Se o primeiro pixel é 0, então o último é 639 e não 640. O temporizador de cima gera os sinais verticais e suas entradas contam linhas, daí **upix** ser 479 (o nome do sinal não está muito certo neste caso).

Os primeiros monitores VGA usavam a polaridade dos sinais de sincronismo para indicar qual modo usar. Os monitores atuais são mais flexíveis, mas é melhor seguir a convenção.

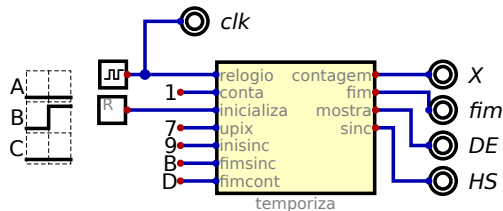


Figura 46: Teste do gerador de sincronismo

O modo 640x480 é indicado com os dois sincronismos de polaridade negativa e por isso aparecem as duas portas NOT.

Um multiplexador é usado para forçar os sinais de cores para 0 fora da região ativa. A região ativa é a combinação das regiões ativas horizontal e vertical, daí a porta AND combinando os dois sinais mostra.

O que aparece na tela do monitor VGA pode ser visto na Figura 48. Juntos, os sinais de vermelho, verde e azul (R, G e B) precisam de 24 bits. Como os contadores de pixel (X) e de linha (Y) tem 12 bits cada simplesmente juntamos seus valores. Só que X vai até 799 e Y apenas até 524. Por isso a imagem mostra apenas uma pequena fração das cores possível. O objetivo deste teste é ver se alguma imagem poderia ser mostrada e nisso ele foi um sucesso.

O problema do teste de cores é que se houverem pixels ou linhas a mais ou a menos não fará uma diferença notável na imagem. O circuito da Figura 49 é quase igual ao anterior mudando apenas como são gerados os sinais R, G e B (a parte de forçar para 0 fora da área ativa continua igual). Os 4 bits de baixo de X e de Y são comparados com tudo 0 e com tudo 1 e branco é mostrado neste caso.

Isso gera um padrão quadriculado que pode ser visto na Figura 50. As linhas do meio tem 2 pixels de largura ou altura, mas as da borda externa tem apenas 1 pixel. Com isso dá para ver que a imagem é exata sem erros de alinhamento. Num monitor antigo de tubo de raios catódicos este padrão também ajudaria a encontrar problemas nos ajustes do monitor. Mas com telas de cristal líquido esta parte, pelo menos, não causa problemas e o meio da imagem deve ser perfeita.

640x480 60Hz, 25.175MHz, 640, 16, 96, 48, 480, 11, 2, 31

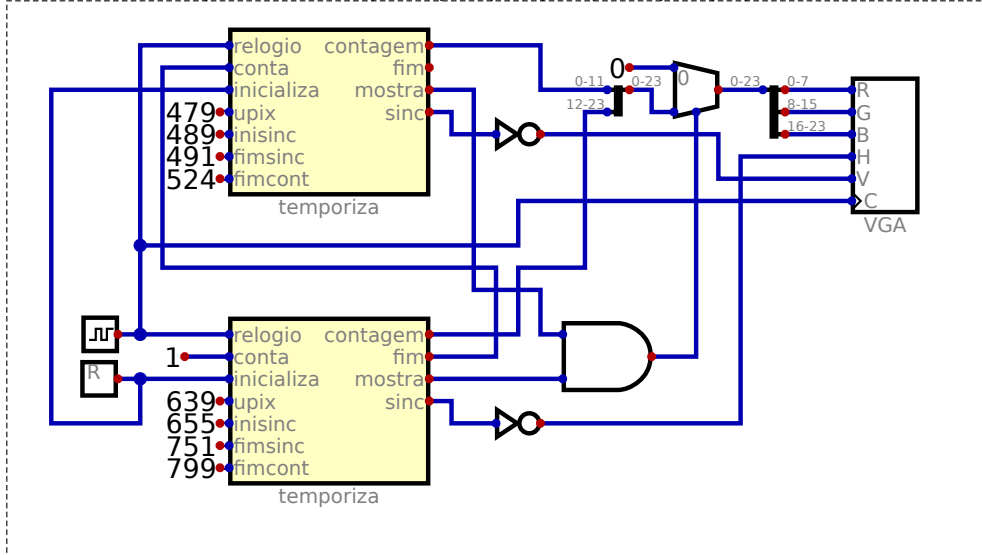
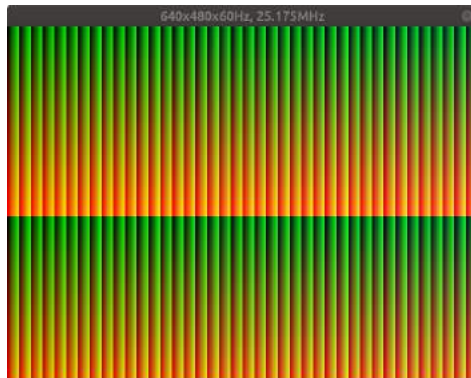


Figura 47: Gera um padrão de cores na tela VGA

Figura 48: Tela do teste de cores



640x480 60Hz, 25.175MHz, 640, 16, 96, 48, 480, 11, 2, 31

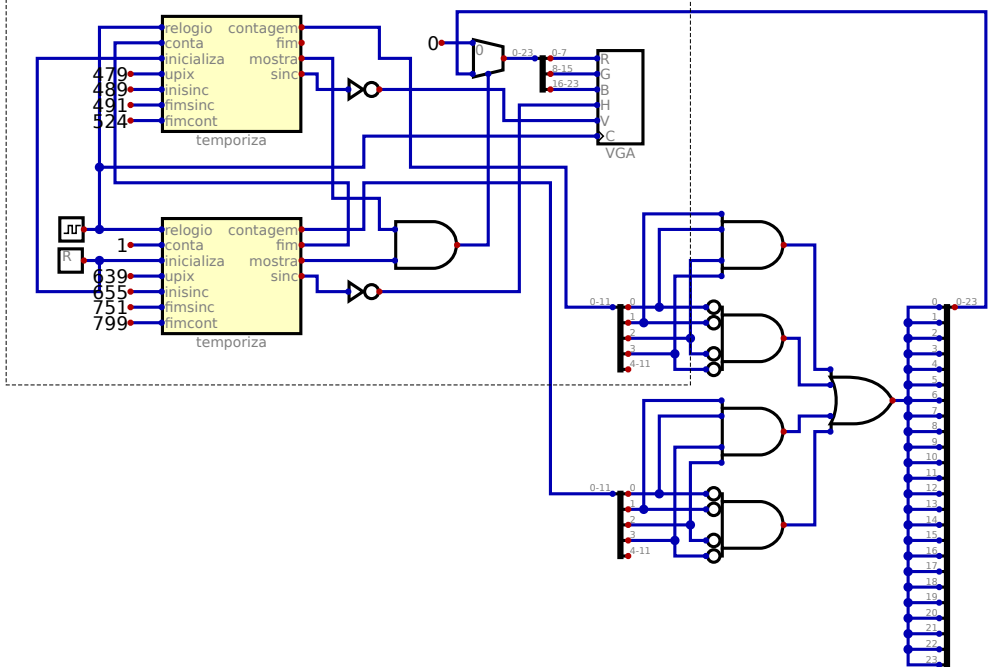


Figura 49: Gera um padrão quadriculado na tela VGA

Figura 50: Tela do teste padrão quadriculado



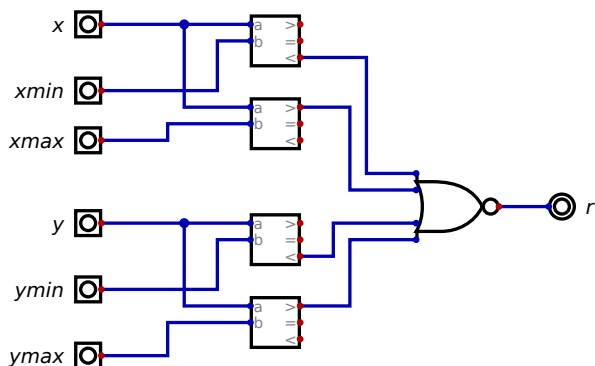


Figura 51: Indica se um ponto está dentro de um retângulo

O Odyssey da Magnavox podia mostrar alguns quadrados na tela. Queremos generalizar para retângulos de qualquer tamanho. O circuito da Figura 51 compara as coordenadas de um ponto e sinaliza se o ponto faz parte do retângulo ou não.

O circuito da Figura 52 usa dois blocos de teste de retângulo para saber quais pontos estão em certas regiões da tela. Um vai do ponto 100,100 ao ponto 300,300 enquanto o segundo vai de 200,150 a 500,400.

A mudança mais interessante é como são geradas as cores. O circuito codificador com prioridade é o oposto do decodificador. No decodificador um número binário escolhe uma entre muitas saídas. No codificador um sinal em uma das muitas entradas é convertido em um número binário. Enquanto no decodificador o número corresponde a exatamente uma saída, no codificador nada impede que mais de uma entrada seja acionada ao mesmo tempo. No codificador com prioridade é definida uma ordem para as entradas e a saída será o número da entrada sinalizada com a maior prioridade.

No circuito de teste de retângulos a entrada de menor prioridade está sempre ativa. Ele corresponde à cor do fundo. O sinal de maior prioridade é acionado fora da área ativa e os dois retângulos estão no meio, com o 100,100 a 300,300 tendo menor prioridade. O número gerado pelo codificador com prioridade é usado pelo multiplexador para escolher um valor de

640x480 60Hz, 25.175MHz, 640, 16, 96, 48, 480, 11, 2, 31

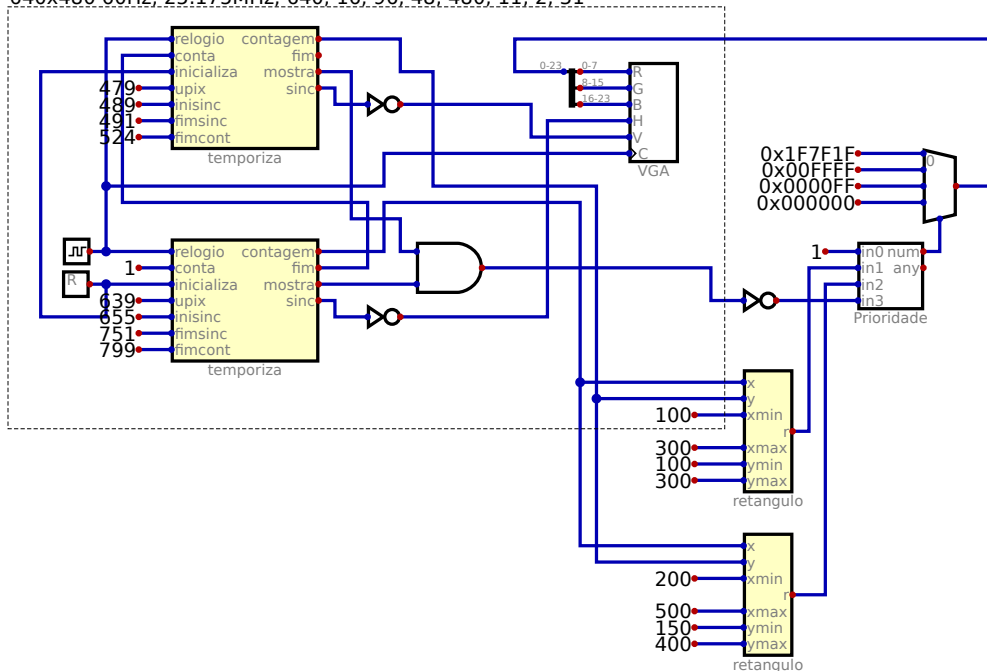


Figura 52: Gera dois retângulos na tela VGA

Figura 53: Tela do teste de retângulos



24 bits para ser usado como cor. A cor da maior prioridade (vídeo não ativo) é 0.

A tela do teste de retângulos (Figura 53) mostra que nos pontos onde os dois retângulos estão sobrepostos é a cor do retângulo da entrada 2 do codificador que aparece. Temos, assim, uma maneira de fazer uns objetos aparecerem na frente de outros na tela.

Podendo gerar retângulos coloridos na tela já é possível criar jogos parecidos com os típicos da primeira geração.

Usando uma ROM com o formato dos caracteres, é possível mostrar texto na tela. O circuito da figura 54 demonstra alguns dos problemas que precisam ser resolvidos no caso dos textos. A saída da ROM com os formatos dos caracteres é paralela, mas estes bits precisam ser enviados para a tela um de cada vez. O circuito usa um registrador que ou pode receber o dado ou a própria saída do registrador mas deslocada de um bit. Isso também já foi usado no multiplicador sequencial.

O texto gerado pode ser visto na Figura 55 e é um padrão repetitivo pois alguns bits dos contadores X e Y são usados para escolher o caracter. Os bits inferiores do contador Y escolhem a linha dentro da imagem do caracter. Numa aplicação prática os contadores X e Y selecionariam um caracter em uma RAM e a saída disso seria a entrada da ROM. Mudanças no conteúdo da RAM alterariam o texto exibido na tela.

640x480 60Hz, 25.175MHz, 640, 16, 96, 48, 480, 11, 2, 31

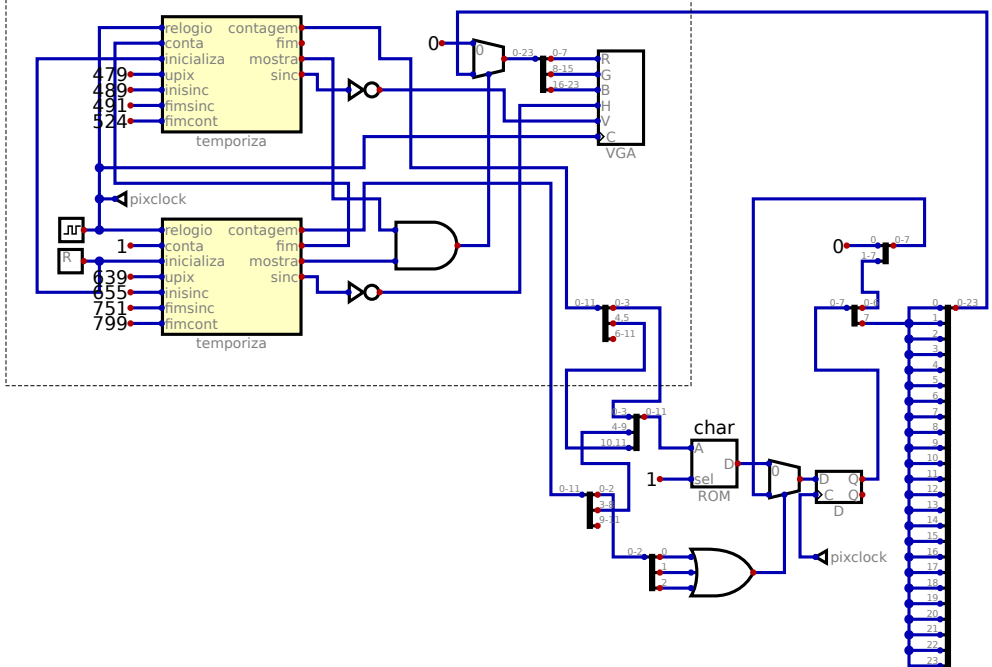
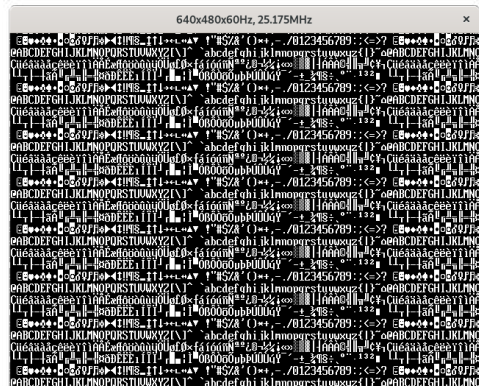


Figura 54: Gera um texto na VGA

Figura 55: Tela do teste de texto

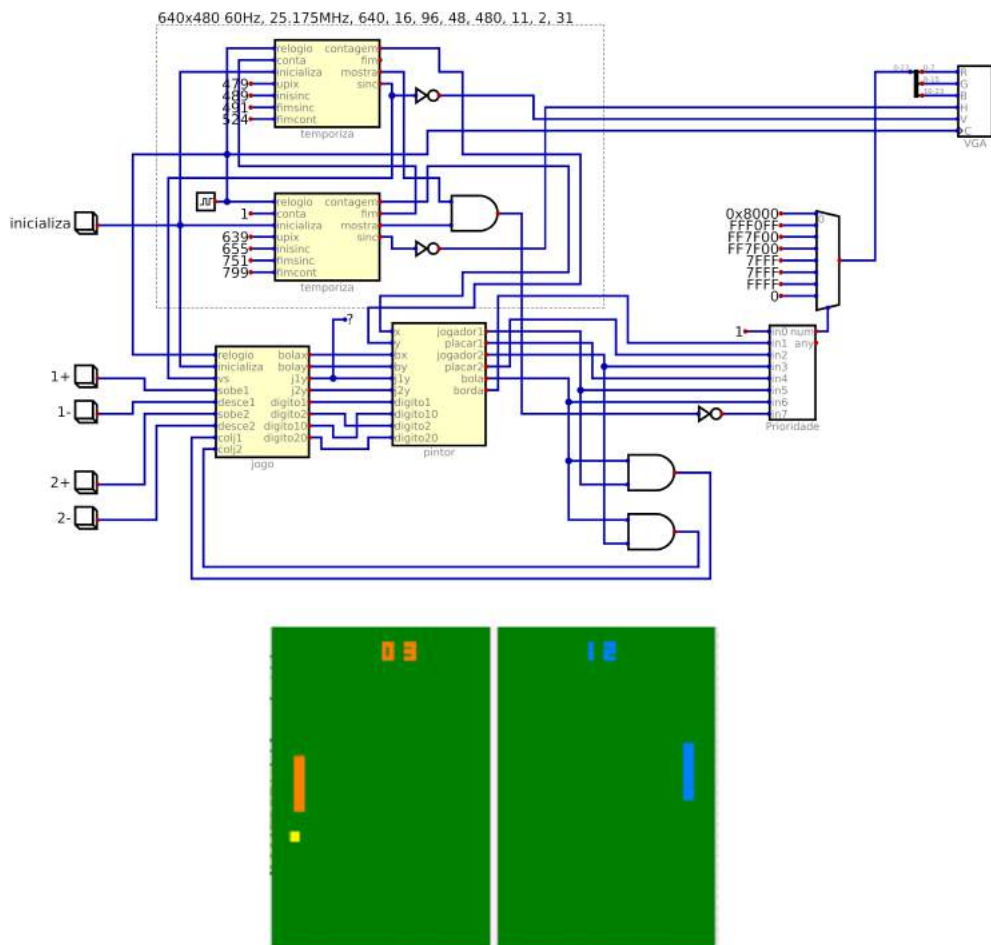




Jogo Pong criado no simulador Digital

Foi elaborado um jogo de primeira geração conhecido como Pong, no qual cada jogador deve comandar uma raquete virtual em um jogo eletrônico simulando uma partida de tênis de mesa, exibido nas Figuras 56, usando apenas as primitivas de componentes digitais do simulador Digital.

Figura 56: Jogo Pong criado no simulador Digital



Linguagens HDL

Uma linguagem de descrição de hardware (HDL do inglês Hardware Description Language) é uma linguagem de programação que modela o hardware utilizando um design baseado na descrição textual de circuitos eletrônicos, principalmente de lógica digital. Quando utiliza-

crédito da imagem: pixabay.com

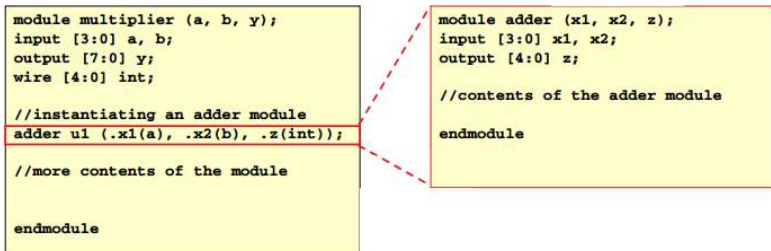
mos HDL's, não estamos programando como em C ou Python, e sim descrevendo um hardware. Dessa forma, diferente das linguagens tradicionais de programação que funcionam de maneira sequencial, ou seja, cada linha é lida pelo compilador, uma após a outra, em HDL os circuitos ali descritos funcionam como uma implementação em protoboard (placa de prototipação usada em eletrônica, na qual os circuitos são conectados e interligados com fios), por exemplo. As partes funcionam simultaneamente, recebendo valores de entrada e apresentando valores de saída.

Atualmente, as duas linguagens mais utilizadas para isso são: Verilog e VHDL. Nesse livro vamos utilizar apenas a linguagem Verilog.

Estrutura de código em Verilog

Um projeto em Verilog é dividido em blocos funcionais chamados de módulos. Os módulos são análogos às funções ou procedimentos em linguagens de alto nível e possuem papéis específicos, como por exemplo somar ou manipular dados em uma estrutura de fila. Eles também podem ser portas de entrada, saída ou mesmo uma porta bidirecional, além de comunicar-se com outros módulos dentro de um mesmo projeto, já que a linguagem é hierárquica, ou seja, um módulo pode conter outros módulos, conforme mostra o exemplo a seguir.

Figura 57: Exemplo de módulos chamando outros módulos em Verilog



Operadores em Verilog

Assim como em linguagens de software convencionais, o Verilog possui seu próprio conjunto de operadores, ilustrados na Tabela 14.

Tipo	Símbolo	Exemplo	Comentário
Aritmética	- * / %	c = a + b;	Tome cuidado ao utilizar * / ou % na síntese de lógicas mais complicadas
Bit wise	~& ^~^	c = a b;	Realiza a operação lógica de bits nos operandos
Logical	! &&	if (flag1 && flag2)	Resultado produz 1'b0 or 1'b1
Redução	& ^~& ~ ~^	c = &a;	Realiza uma operação lógica em todos os bits de um único operando
Shift	<<>>	c = a <<2;	Desloca os bits de um operando. Sempre que for possível use o shift para multiplicar ou dividir.
Relacional	<<=> >= == !=	c = a <= b;	O resultado é 1'b0 ou 1'b1. Não confunda com o símbolo de non-blocking assignemnt
Condicional	?:	assign mux_out = sel? a : b;	Atalho para if-else
Concatenação	{ }	c = { 1'b1, 2'b00, a }	Concatena bits (junta lado a lado num barramento)
Replicação	{ { } }	c = {3{2'b10}}; //c é agora 6'b101010	Réplica um grupo de bits

Tabela 14: Conjunto de operadores da linguagem Verilog

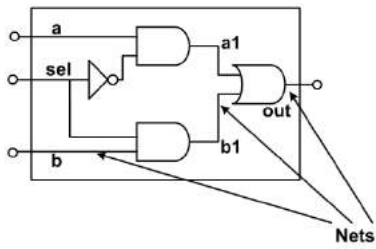
Em linguagens de alto nível como C ou Java possuem tipos de dados como int, char, float, entre outros. Para Verilog, temos três tipos de dados:

- **Nets:** são conexões físicas entre os blocos;
- **Registers:** são elementos de armazenamento sequencial, ou seja, armazenam os valores recebidos até que recebem um valor diferente;
- **Nets:** são restrições personalizadas para que, através de constantes cria-se a parametrização dos módulos criados (como por exemplo, para definição de delay, tamanho de variáveis e enumeração de tipos literais).

Nets

O dado tipo Net faz a conexão entre blocos e módulos. Este tipo de dado é continuamente impulsionado pelos dispositivos que os controlam. O Verilog propaga automaticamente um novo valor em um Net quando os drivers mudam de valor (conforme visto na Figura 58). Na Tabela 15, estão listados todos os tipos de Net disponíveis, sendo o tipo wire o mais utilizado.

Figura 58: Mudança de valor dos drivers



Tipos de Net	Funcionalidade	Exemplo	Comentário
wire, tri	Interconexão padrão para fios	c = a + b;	Tome cuidado ao utilizar * / ou % na síntese de lógicas mais complicadas
supply1, supply0	Para power rails/ ground rails	c = a b;	Realiza a operação lógica de bits nos operandos
wor, trior	Para múltiplos drivers que são Wire-ORed	if (flag1 && flag2)	Resultado produz 1'b0 or 1'b1
wand, triand	Para múltiplos drivers que são Wire-ANDed	c = &a;	Realiza uma operação lógica em todos os bits de um único operando
trireg	Para nets com armazenamento capacitivo	c = a <<2;	Desloca os bits de um operando. Sempre que for possível use o shift para multiplicar ou dividir.
tri1, tri0	Para nets que puxam para cima/baixo quando não orientadas	c = a <= b;	O resultado é 1'b0 ou 1'b1. Não confunda com o símbolo de non-blocking assignemnt

Tabela 15: Tipos de Nets e suas funcionalidades

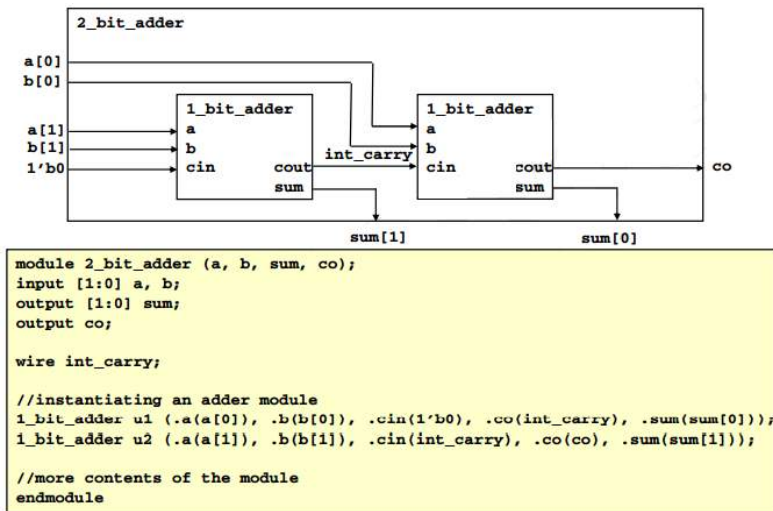
Wires

Sobre os wires, eles podem ser utilizados para:

- Conectar módulos instanciados;
- Realizar combinações lógicas em tempo de execução.

Quando um tipo de dado de entrada ou saída é declarado em um módulo, eles também são implicitamente um wire, conforme o exemplo da Figura 59.

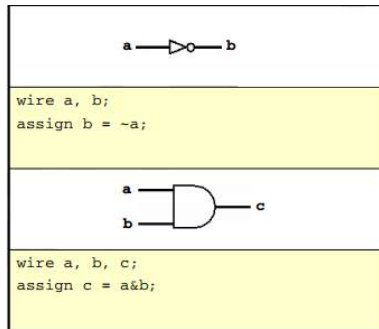
Figura 59: Exemplo de soma de dois bits



A declaração de atribuição e o tipo de dados wire são usados em conjunto para criar combinações lógicas em tempo de execução, conforme o exemplo da Figura 60.

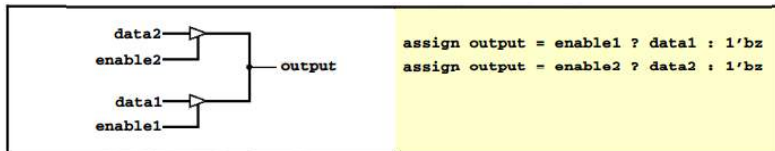
Outra combinação são wires e 'tri-state buffers'. 'Tri-state buffers' são dispositivos que guiam o valor de saída para o valor de sua entrada, ou deixam a saída em um estado de alta impedância. Quando um Net se encontra no estado de alta impedância, pode ser conduzido por

Figura 60: Exemplo de uso de wire e assign



qualquer buffer em tri-state que esteja conectado. Cuidados devem ser tomados para garantir que apenas um buffer conduza o wire de cada vez, enquanto os outros permaneçam em alta impedância, conforme o exemplo da Figura 61.

Figura 61: Exemplo de uso de wire e 'tri-state buffer'



Registradores

Os dados do tipo registers (registradores) armazenam valores em um bloco. Os registers possuem diversos tipos, conforme quadro abaixo. Apenas como informação, não confunda um reg com um registrador real em hardware; um reg pode ou não sintetizar em um registrador.

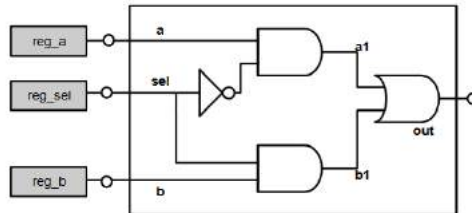
Registers são usados extensivamente em modelagem comportamental e na aplicação de

Tipos de Registradores	Funcionalidade
reg	Variável inteira não assinada de tamanho variado
integer	Variável inteira de tamanho variado
time	Variável inteira não assinada, 64bits
real	Variável de ponto flutuante com dupla precisão
realtime	Mesmo que real
parameter	Constantes de qualquer tipo

Tabela 16: Tipos de Registradores e suas funcionalidades

estímulos. Os valores são atribuídos usando construções comportamentais. Um exemplo é apresentado na Figura 62.

Figura 62: Exemplo de entrada 'registrada', que mantém ou sincroniza com um sinal de relógio os dados da entrada



Mas como podemos usar regs?

- Podemos atribuí-los dentro de blocos processuais;
- Fazer declarações processuais para registrar apenas os tipos;
- Atribuir a partir de nets internas ou fora de blocos processuais;
- Podem ser transformados em lógica combinacional ou sequencial.

Parâmetros

Os parameters (parâmetros) são constantes específicas do módulo, assim como constantes são utilizadas em linguagens de programação convencionais. Os parâmetros predefinidos podem ser substituídos cada vez que você instanciar um módulo.

Eles são geralmente usados para definir larguras, nomes dos estados e atrasos, conforme apresentado na Figura 63.

Figura 63: Exemplo de uso de parameter

```
module adder (a, b, y);  
  parameter WIDTH = 4;  
  input [WIDTH-1:0] a, b;  
  output [WIDTH:0] y;  
  
  //more contents of the module  
  ...  
endmodule
```

Blocos

Por definição, blocos processuais são seções que contêm declarações que são executadas linha por linha, como um software convencional. Vários blocos processuais podem interagir simultaneamente.

Utilizamos dois tipos de blocos: o **bloco always** e o **bloco initial**.

O **bloco always** executa sempre que um evento dentro da lista de eventos é acionado, além disso pode construir combinações lógicas ou sequenciais.

O **bloco initial** é usado apenas para simulações e não é sintetizável. O bloco initial é executado no começo de cada simulação e é utilizado para aplicar estímulos no circuito que está sob teste. Uma observação importante é que todo sinal alternado no bloco initial deve ser do tipo register.

Por sua vez, existem dois tipos de declarações em blocos: **blocking** e **non-blocking**.

Declarações **blocking** são executadas uma após a outra, semelhante as linguagens de

software convencionais, ou seja, novas execuções são bloqueadas até que a execução atual esteja completa. Este tipo de declaração é utilizado para criar lógicas combinacionais.

Lógicas combinacionais podem ser criadas de duas formas (Figura 64):

- Lógicas simples através de atribuição;
- Lógicas complexas, usando a construção `always @(*)`;

Figura 64: Exemplos de lógicas combinacionais simples e complexas

Simple combinational logic
<pre>wire a, b, c; assign c = a & b;</pre>
Complex Logic (State machine)
<pre>reg next_state, current_state; always @ (*) begin next_state = current_state; if(current_state == STATE_INIT) next_state = STATE_1; else next_state = STATE_INIT; end</pre>

Já o **non-blocking** é usado para especificar comportamentalmente registradores de hardware ou estágios de pipeline, e para criar lógicas sequenciais. Neste tipo de declaração, as atribuições não ocorrem imediatamente pois são programadas por um gatilho (geralmente na variação do clock).

Lista de eventos

Cada bloco vem sempre com uma lista de eventos (por vezes referido como a lista de sensibilidade). Instruções dentro do bloco `always` são executadas apenas quando um evento dentro da lista ocorre. Um evento é qualquer transição dos sinais especificados.

Você pode especificar a transição com **posedge** (borda de transição de subida) ou **negedge** (borda de transição de descida).

O operador ***** adiciona todos os sinais do bloco dentro de uma lista de eventos, conforme visto na Figura 65.

Figura 65: Exemplo lista de eventos

Event list using the or operator
<pre>always @ (a or b or sel) begin if (sel == 1) op = a; else op = b; end</pre>
Event list using the comma operator
<pre>always @ (a or b or sel) begin if (sel == 1) op = a; else op = b; end</pre>
Event list using the * wildcard
<pre>always @ (*) begin if (sel == 1) op = a; else op = b; end</pre>

Escrevendo um código de apoio não-sintetizável

Sabemos que atrasos e o bloco inicial não são sintetizáveis (que não serão traduzidos em circuitos, auxiliando apenas na depuração da saída de monitoramento do software que faz a

síntese). A linguagem Verilog contém muitas construções que a síntese ignora. A Tabela 17 apresenta uma lista dessas construções.

Construção em Verilog	Descrição
<code>\$display</code>	Imprime sentenças para simulações
<code>\$signed</code>	Usado em <code>\$display</code> para imprimir variáveis em um formato declarado
<code>\$unsigned</code>	Usado em <code>\$display</code> para imprimir variáveis em um formato não declarado
<code>real</code>	Tipo de dado para armazenar números decimais
<code>initial block</code>	Usado somente em simulações
<code>gate delays</code>	Usado somente em simulações
<code>always @</code>	Quando usado dentro de blocos <code>initial</code> não é sintetizável
<code>fork, join</code>	A construção <code>fork ... join</code> é usada em blocos <code>initial</code> para executar linhas de código simultaneamente
<code>specify</code>	Especifica <code>delays</code> de portas lógicas
<code>specparam</code>	Especifica uma constante de bloco para armazenar inteiros, hora, números reais, <code>delays</code> ou ASCII char
<code>repeat, for, while</code>	Estes loopings quando usados em blocos <code>initial</code> não sintetizáveis

Tabela 17: Construções em Verilog

Boas práticas

Assim como em outras linguagens de programação, é ideal usar algumas convenções. Seguindo essas práticas, podemos poupar horas de depuração e isso torna o código mais fácil para outros projetistas compreenderem. Abaixo seguem algumas recomendações:

- Dê nomes significativos para suas variáveis;
- Recue seu código para torná-lo legível;
- Divida seu código em módulos e funções significativas;
- Use parâmetros (constantes), tanto quanto possível;
- Comente o seu código.

Em um bloco always @ (*), cada reg deve ter um valor para todos os condicionais do case. Se algum deles for omitido, um latch será gerado. Para evitarmos este problema, devemos dar a cada reg um valor de atribuição padrão antes de cada caso ou construção if-else. Veja o exemplo da Figura 66.

Figura 66: Todo registrador deve ter um valor para todos os condicionais de um case

<pre>reg next_state, current_state; always @ (*) begin if(current_state == STATE_INIT) next_state = STATE_1; end</pre>	<pre>reg next_state, current_state; always @ (*) begin next_state = current_state; if(current_state == STATE_INIT) next_state = STATE_1; end</pre>
Errado	Correto

Outro erro muito comum é misturar **blocking assignment** com **non-blocking** em um mesmo bloco always@ (Figura 67). Divida seu código em blocos o quanto for necessário, lembrando que eles irão ser executados paralelamente.

Figura 67: Evitar misturar **blocking assignment** com **non-blocking** em um mesmo bloco always

<pre>reg next_state, current_state; always @ (posedge clk) begin if(reset) current_state <= STATE_INIT; else current_state <= next_state; if(current_state == STATE_INIT) next_state = STATE_1; end</pre>	<pre>reg next_state, current_state; always @ (posedge clk) begin if(reset) current_state <= STATE_INIT; else current_state <= next_state; end always @ (*) begin next_state = current_state; if(current_state == STATE_INIT) next_state = STATE_1; end</pre>
Errado	Correto

Alguns designers usam resets assíncronos (Figura 68), pois é muito arriscado e não é recomendado. Se for necessário utilizar um reset assíncrono, certifique-se pelo menos de que o sinal de reset não viola quaisquer tempos de setup/hold.

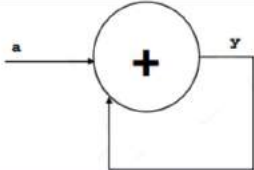
Figura 68: Evitar o uso de resets assíncronos

<pre>reg next_state, current_state; always @ (posedge clk or reset) begin if(reset) current_state <= STATE_INIT; else current_state <= next_state; end</pre>	<pre>reg next_state, current_state; always @ (posedge clk) begin if(reset) current_state <= STATE_INIT; else current_state <= next_state; end</pre>
Errado	Correto

Outro ponto de atenção é não criar loopings infinitos assíncronos. No exemplo da Figura 69 à esquerda, quando Y é incrementado, a lógica detecta a alteração e executa a adição novamente, criando um ciclo interminável.

Já exemplo no lado direito mostra um contador sequencial que incrementa Y a cada ciclo de clock, o qual é a maneira correta.

Figura 69: Evitar a criação de loopings infinitos assíncronos

	
<pre>always @ (*) begin y = y + a end</pre>	<pre>always @ (posedge clk) begin y <= y + a end</pre>
Errado	Correto

Outra boa prática muito importante é a de não atribuir múltiplos valores a um mesmo sinal wire de maneira incondicional (Figura 70), caso contrário ele será resolvido para um 0 lógico.

Figura 70: Evitar múltiplas atribuições a um mesmo wire sem um teste condicional

<pre>wire y; assign y = a&b; assign y = a b;</pre>	<pre>wire y; if(c) assign y = a&b; else assign y = a b;</pre>
Errado	Correto

Já os números podem ser representados em notação decimal, hexadecimal, binário, octal ou em notação científica:

- 12 - decimal;
- 'H83a - hexadecimal;
- 8'b1000_0001 - 8-bit binário;
- 64'hff01 - 64-bit hexadecimal;
- 9'O17 - 9-bit octal;
- 2'B1z - 2-bit número binário com LSB em alta impedância;
- 32'bz - 32-bit Z (valores de X e Z são automaticamente aumentado);
- 6.3 - notação decimal;
- 32e-4 - notação científica para 0.0032.

E para comentários utilizamos o exemplo da Figura 71.

Figura 71: Exemplo de uso de comentário // na linguagem Verilog

```
module MUX2_1 (out,a,b,sel);  
  
    // Port declarations  
  
    output out;  
    input a, b, sel;
```

Para utilizar strings, devemos colocá-las entre aspas “ “. A linguagem Verilog reconhece os caracteres da linguagem em C: \t, \n, \\\, \", e %%. Eles são utilizadas para gerar saídas formatadas.

A criação dos nomes dos objetos (módulos, portas e instâncias) devem começar com uma letra ou um underscore (_) e pode ter no máximo 1023 caracteres.

A linguagem Verilog é **case sensitive** (sensível a diferença entre caracteres maiúsculos e minúsculos) e todas as palavras-chaves são apenas aceitas em caracteres usado letras minúsculas. Já a base de um número pode usar minúscula ou maiúscula.

Podemos executar a síntese de um código em Verilog no modo case-insensitive, especificando -u na linha de comando ou configurações do ambiente de desenvolvimento.

Por fim listamos abaixo os símbolos de monitoramento e depuração mais comuns na linguagem Verilog, e na Tabela 18 a seguir, apresentamos um breve resumo com os erros mais comuns no acesso a seus tipos de dados.

- **\$ <identificador>** o símbolo \$ denota tarefas e funções, como por exemplo:
 - \$time - encontra a simulação corrente;
 - \$display/ \$monitor - mostra ou monitora os valores dos sinais;
 - \$stop - para uma simulação;
 - \$finish - termina uma simulação.
- **# <delay specification>** o símbolo # denota a especificação de delay para entrada de portas lógicas ou sentenças procedurais.

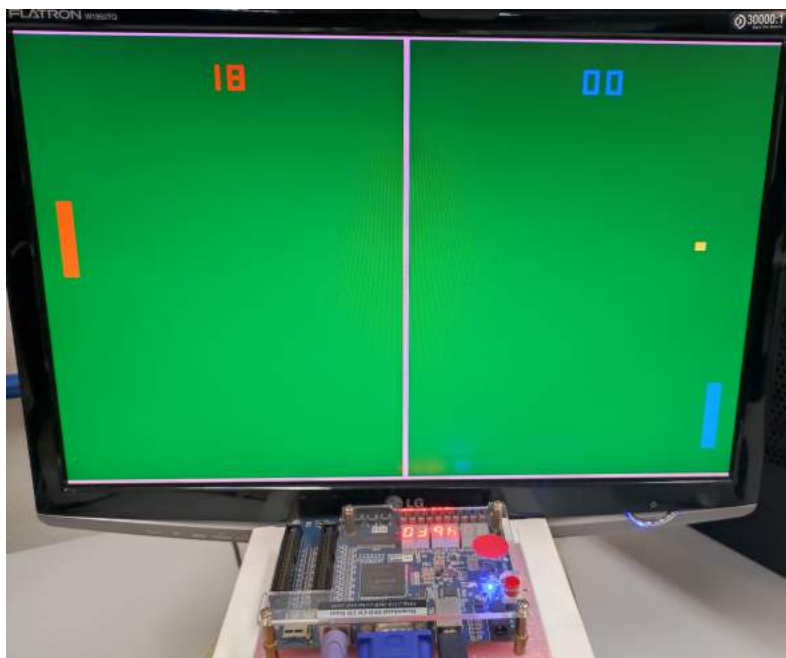
Possível erro	Mensagem
Quando uma atribuição processual é feita para um net ou um sinal é esquecida de ser declarado como reg.	<i>“Illegal Left-hand-side assignment”</i>
Quando o sinal conectado em uma porta de saída é um register	<i>“Illegal output port specification”</i>
Quando o sinal conectado em uma porta lógica primitiva é um register	<i>“Gate has illegal output specification”</i>
Quando a entrada de um módulo é declarado como um register	<i>“Incompatible declaration, (signal) defined as input”</i>

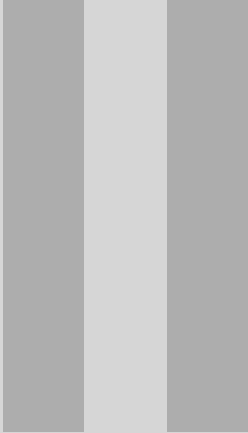
Tabela 18: Erros comuns em Verilog

Jogo Pong em Verilog

Recriamos o jogo Pong usando a linguagem de descrição de hardware Verilog, sendo executado em um kit de FPGA modelo DE0-CV da fabricante Terasic, usando uma FPGA Cyclone V, conforme apresentado na Figura 72. O código fonte completo se encontra no Apêndice A desse livro.

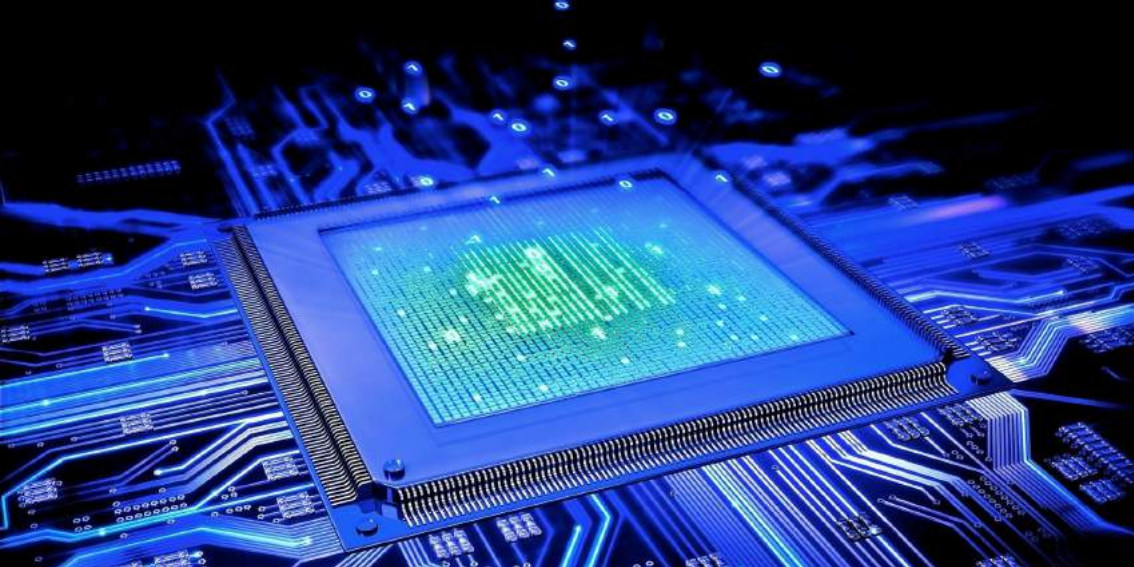
Figura 72: Jogo Pong codificado em linguagem de descrição de hardware Verilog





Parte II Videogames de SEGUNDA geração

Processador AP9	103
Jogo Tetris em Assembler	163



Processadores

Processador AP9

O processador AP9 de 16 bits, foi implementado em linguagem Verilog™ pela equipe do professor Eduardo do Valle Simões, e disponibilizado para uso sobre licença GPL na plataforma Github. Esse processador possui um conjunto de 25 instruções e seu módulo **CPU** possui componentes internos, **ULA** e **UC**, bem como funções de entrada e saída (internas e externas) que realizam a ligação entre os módulos, a arquitetura é apresentada na Figura 73.

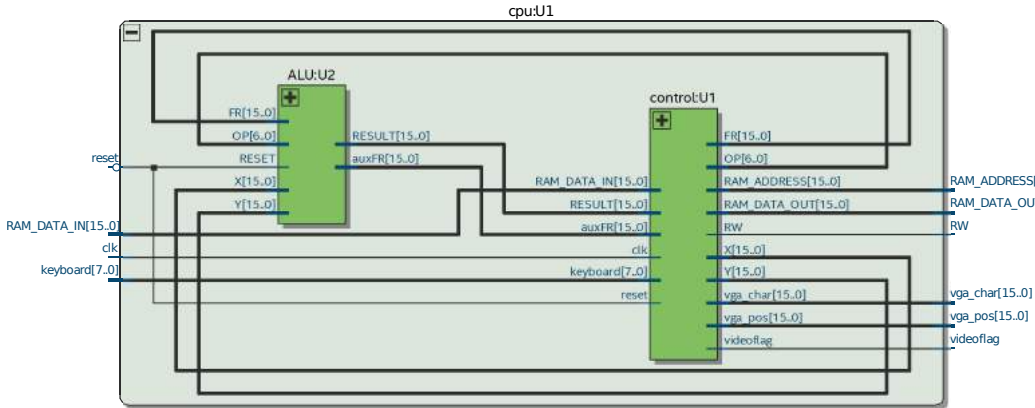


Figura 73: Processador didático AP9

Abaixo segue a declaração do módulo CPU (Código 1) desenvolvido em linguagem VerilogTM, onde foram declaradas, registradores, entradas, saídas e seus componentes internos (ULA e UC):

```

1  module  cpu_v(wire_clock, wire_reset, bus_RAM_DATA_IN, bus_RAM_DATA_OUT,
2
3  input wire          wire_clock;
4  input wire          wire_reset;
5  input wire [15:0]   bus_RAM_DATA_IN;
6  output wire [15:0]  bus_RAM_DATA_OUT;
7  output wire [15:0]  bus_RAM_ADDRESS;
8  output wire         wire_RW;
9  input wire [7:0]    bus_keyboard;
10 output wire         videoflag;
11 output wire [15:0]  bus_vga_pos;
12 output wire [15:0]  bus_vga_char;

```



```

13  output wire [15:0]  data_debug;
14  output wire [15:0]  led;
15
16
17  wire [15:0]          m2;
18  wire [15:0]          m3;
19  wire [15:0]          m4;
20
21
22  wire                enable_alu;
23  wire [5:0]          opCode;
24  wire                useCarry;
25  wire                useDec;
26  wire [15:0]          FR_in_at_control;
27  wire [15:0]          FR_out_at_control;
28
29
30  wire [2:0]           flagToShifthAndRot;
31
32  alu_v alu_v_dut(wire_clock,enable_alu,m2,m3,m4,opCode,
33                FR_out_at_control,FR_in_at_control,useCarry,flagToShifthAndRot,
34                useDec);
35
36  control_unit_v control_v_dut(wire_clock, wire_reset, bus_RAM_DATA_IN,
37                                bus_RAM_DATA_OUT,bus_RAM_ADDRESS,wire_RW, bus_keyboard, videoflag,
38                                bus_vga_pos, bus_vga_char,enable_alu,FR_in_at_control,
39                                FR_out_at_control,opCode,useCarry,flagToShifthAndRot,useDec,m2,m3,m4
40                                ,data_debug, led);
41
42  endmodule

```

Código 1: Módulo CPU

A Figura 74 mostra as variáveis declaradas para o processador AP9 de 16 bits e a finalidade de cada um deles.

Figura 74: Conjunto de variáveis do processador AP9

Nome	Finalidade
wire_clock	Barramento de Entrada de Clock
wire_reset	Barramento de Entrada de Reset do Processador
bus_RAM_DATA_IN	Barramento de 16bits de Entrada de Dados da Memória RAM
bus_RAM_DATA_OUT	Barramento de 16bits de Saída de Dados da Memória RAM
bus_RAM_ADDRESS	Barramento de 16bits de Endereço da Memória RAM
wire_RW	Barramento de Sinalização de Gravação na Memória RAM
bus_keyboard	Barramento de 8bits de Entrada de Dados do Teclado
videoflag	Barramento de Sinalização de Flag de Vídeo
bus_vga_pos	Barramento de 16bits de Saída de Vídeo
bus_vga_char	Barramento de 16bits de Saída de Vídeo de Character
data_debug	Barramento de 16bits de Saída de Depuração de Dados
led	Barramento de 16bits de Saída de LED
M2	Barramento de Registrador de Dados Interno de 16bits
M3	Barramento de Registrador de Dados Interno de 16bits
M4	Barramento de Registrador de Dados Interno de 16bits
enable_alu	Barramento Interno de Flag (Sinalização) de Controle de ULA
opCode	Barramento de Registrador de Dados Interno de 6bits, utilizado para armazenar código de instrução
useCarry	Barramento Interno de Sinalização de Bit de Carry
useDec	Barramento Interno de Sinalização de Bit de Instrução de Decremento
FR_in_at_control	Barramento de Entrada Interno de Flag (Sinalização) da Unidade de Controle de 16bits
FR_out_at_control	Barramento de Saída Interno de Flag (Sinalização) da Unidade de Controle de 16bits
flagToShifthAndRot	Barramento Interno de Flag (Sinalização) de 3bits da Instrução Shift e Rots

Os conjunto de variáveis declaradas no módulo da **CPU** são necessárias para interligação do módulo com os módulos externos que compõem o computador, bem como realiza as ligações interna entre a **ULA** e a **UC**. São realizadas através destas variáveis a entrada e saída de dados do processador, as instruções recebidas pelo processador por meio de uma entrada de teclado, por exemplo, acionam as unidades de **UC** e **ULA** que realizarão o processamento deste dado e reproduziram sua saída em um unidade de vídeo. Este fluxo pode ser exemplificado como um ciclo, o processador recebe uma dada instrução, realiza seu processamento e após a reprodução do resultado, retornando ao estado de espera por uma nova instrução, podemos exemplificar este ciclo na Figura 75.

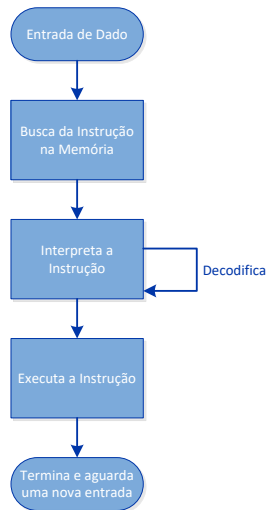


Figura 75: Fluxo básico de um ciclo de instruções

O Processador AP9 foi desenvolvido com um conjunto de 25 instruções, apresentadas na Figura 76.

Tipos de Instruções	Instruções
Instruções de Manipulação de Dados	LOAD
	STORE
	STOREI
	LOADI
	LOADN
	MOV
Instruções de Entrada e Saída (I/O)	INCHAR
	OUTCHAR
Instruções de Controle de Fluxo	PUSH
	POP
	RTS
	JUMP*
	CALL *
Instruções Aritméticas	ADD
	ADDC
	SUB
	MULT
	DIV
	INC e DEC
	MOD
	CMP
Instruções Lógicas	AND
	OR
	XOR
	NOT
	SHIFT e
	ROTS

*As instruções possuem variações que são alteradas conforme sua Flag FR.

Figura 76: Conjunto de Instruções Processador AP9

Como visto anteriormente na Figura 73 a arquitetura da CPU é composta por seguintes componentes:

- Unidade Lógica e Aritmética (ULA)
- Unidade de Controle (UC)

São unidades internas ao módulo CPU que realizam o processamentos das informações fornecidas pelos programas, o processamento destas informações são realizadas em estágios, ou ciclos de clock, que podem ser de carregamento de dados, processamento ou parada.

A Unidade de Controle, como o próprio nome sugere, é responsável por receber e coordenar estas informações, no processador AP9, a Unidade de Controle (UC) foi desenvolvida utilizando uma máquinas de estado.

O Código 2 apresenta à declaração inicial do módulo da Unidade de Controle implementada no processador AP9, nele além das entradas e saídas da CPU apresentados na Figura 73, são declarados os barramentos de entrada e saída, registradores de uso interno e externo.

```

1 module control_unit_v( wire_clock, wire_reset, bus_RAM_DATA_IN,
   bus_RAM_DATA_OUT,
2     bus_RAM_ADDRESS, wire_RW, bus_keyboard, videoflag,
   bus_vga_pos,
3     bus_vga_char, enable_alu, FR_in_at_control, FR_out_at_control,
4     opcode, useCarry, flagToShifthAndRot, useDec, m2, m3, m4,
   data_debug, led);
5   input wire wire_clock;
6   input wire wire_reset;
7   input wire [15:0] bus_RAM_DATA_IN;
8   output reg [15:0] bus_RAM_DATA_OUT;
9   output reg [15:0] bus_RAM_ADDRESS;
10  output reg wire_RW;
11  input wire [7:0] bus_keyboard;
12  output reg videoflag;
13  output reg [15:0] bus_vga_pos;
14  output reg [15:0] bus_vga_char;
15  output reg enable_alu=1'b0;
16  input wire [15:0] FR_in_at_control;
17  output reg [15:0] FR_out_at_control;
18  output reg [5:0] opcode;
19  output reg useCarry;
20  output reg [2:0] flagToShifthAndRot;

```

```
21  output reg          useDec;
22  input  wire  [15:0] m2;
23  output reg  [15:0] m3;
24  output reg  [15:0] m4;
25  output reg  [15:0] data_debug;
26  output [15:0] led;
27
28  reg          resetStage;
29  reg [7:0]    stage=8'h00;
30  reg          processing_instruction=1'b0;
31  reg          definingVariables=1'b1;
32  reg          startedProcessing;
33  reg [15:0]   IR;
34  reg [15:0]   PC;
35  reg [15:0]   Rn [0:7];
36  reg [15:0]   END;
37  reg [15:0]   SP=16'h7ffc;
```

Código 2: Módulo Unidade de Controle

Neste módulo foram declaradas entradas, saídas e registradores da Unidade de Controle, a mudança neste módulo em relação ao módulo CPU e a declaração da variável "*enable_alu=1'b0*" que ativa a ULA, atribui-se valor inicial de 0, são declarados declarados no trecho apresentado registradores aos quais serão apresentados na tabela abaixo.

Nome	Finalidade
resetStage	Registrador de Reset
stage=8'h00	Registrador de 8bits com valor inicial de 0 (Zero)
processing_instruction=1'b0	Registrador de instrução com valor inicial de 0 (Zero)
definingVariables=1'b1	Registrador de variável com valor inicial de 1 (um)
startedProcessing	Registrador de início de processo
IR	Registrador de instrução (IR, do inglês "Instruction Register") de 16bits
PC	Contador de Programas (PC, do inglês "Program Counter") Registrador de 16bits que indica o endereço da próxima instrução a ser executada pelo processador
Rn[0:7]	Registrador de uso geral de 16bits, com referência de 8bits
END	Registrador de Endereço de 16bits
SP=16'h7FFC	Registrador de Ponteiro de Pilha de 16bits (SP, do inglês "Stack Pointer")

Figura 77: Registradores Processador AP9

Está Unidade é responsável por controlar a execução das operações do processador para executar as instruções de um programa, o código 3 apresenta o código utilizado para carregar as instruções de um programa:

```

1      // 'loadInstruction';=====
2      case x ( stage )
3          8'h01: begin
4              bus_RAM_ADDRESS=PC;
5          end
6          8'h02: begin
7              processing_instruction=1'b1;
8              IR=bus_RAM_DATA_IN;
9              PC=PC+1'b1;
10             resetStage=1'b1;
11         end

```

```
12         endcase
13         data_debug=16'hffff;
14     end else begin
15         data_debug=16'h0000;
16         casez(IR)
17             default : begin
18                 casex(stage)
19                     8'h01: begin
20                         wire_RW=1'b0;
21                         processing_instruction=1'b0;
22                         resetStage=1'b1;
23                     end
24                 endcase
25             end
26         end
```

Código 3: Carregamento de Instrução

A operação de Carregamento de Instrução requer dois ciclos de clock, no Código 3 os ciclos de clock são especificados como *stage*, no primeiro ciclo de clock, *8'h01*, o conteúdo do registrador *PC* (*Contador de Programas*) é colocado no barramento de endereço de memória RAM (*bus_RAM_ADDRESS*). É através das informações contidas no barramento (*bus_RAM_ADDRESS*) que Unidade de Controle (UC), sabe onde buscar ou gravar informações na memória. Em seguida, no segundo ciclo de Clock, *8'h02*, o registrador de instruções *processing_instruction* recebe o valor de 1 inicializando/habilitando a Unidade de Controle (UC), o registrador de instruções *IR* recebe do barramento de dados (*bus_RAM_DATA_IN*) o conteúdo da posição de memória informada.

Ao final da instrução o barramento de depuração de dados receberá um conjunto de *16'hffff* em hexadecimal, caso contrário, o barramento receberá um conjunto de **16'h0000** em hexadecimal.

Na linha 16, após um ciclo de clock, é lida a informação do registrador de instruções *IR* que recebeu qual será a instrução há ser executado pela Unidade de Controle, na linha 19 temos o caso **default**, caso a instrução solicitada não tenha sido encontrada, o barramento que sinaliza a gravação em memória RAM (*wire_RW*) e o registrador de instruções (*processing_instruction*) receba o valor de 0, sinalizando o fim da gravação de dados e da instrução, o registrador de estágio (*resetStage*) recebe o valor de 1, voltando a unidade de controle para seu estágio

inicial, caso o registrador (*IR*) possua uma instrução especificada no processador AP9 (Figura 76), está será executada, as instruções do processador são classificadas em categorias, este processador possui um conjunto de 25 instruções que são divididas de acordo com o tipo de operação que executam, iremos apresentar as instruções nas próximos sessões.

Instruções de Manipulação de Dados

O Processador AP9 possui implementado 6 instruções de manipulação de dados, são elas: LOAD, STORE, STOREI, LOADI, LOADN e MOV.

As instruções de manipulação de dados são utilizadas para realizar a transferência de dados entre os registradores ou entre os registradores e a memória principal, seu formato é apresentado na Figura 78.

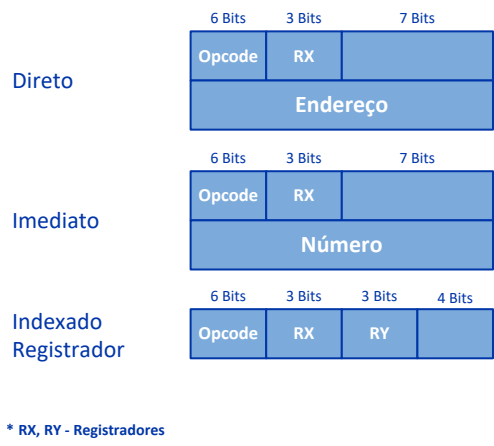


Figura 78: Formato de Instrução de Manipulação de Dados

As instruções a seguir são identificadas através do seu Opcode, que são representados pelos 6 bits mais significativos da informação.

A Instrução LOAD

A instrução LOAD (*16'b110001??????????*) utiliza o modo de endereçamento direto, neste modo de endereçamento o endereço efetivo de memória que deve ser usado pela operação para ler o operando e/ou guardar o resultado é especificado na própria instrução, esta instrução é executada entre a memória e um registrador, realiza a cópia em um registrador da informação contida em uma posição de memória especificada pelo endereço, seu código 4 é declarado a seguir:

```

1      16'b110000?????????: begin
2      // 'instruction_load;=====
3      case x(stage)
4          8'h01: begin
5              bus_RAM_ADDRESS=PC;
6          end
7          8'h02: begin
8              END=bus_RAM_DATA_IN;
9          end
10         8'h03: begin
11             bus_RAM_ADDRESS=END;
12         end
13         8'h04: begin
14             PC=PC+16'h0001;
15             Rn[IR[9:7]]=bus_RAM_DATA_IN;
16             processing_instruction=1'b0;
17             resetStage=1'b1;
18         end
19     endcase
20 end

```

Código 4: Instrução LOAD

Podemos observar que a instrução LOAD necessita de 4 ciclos de clock para ser executada. No primeiro ciclo de clock da instrução, o barramento de endereço de memória (*bus_RAM_ADDRESS*) recebe do contador de programas (*PC*), o endereço de memória da próxima instrução a ser processada. No segundo ciclo de clock da instrução, o registrador de endereço (*END*) recebe do barramento de entrada de dados (*bus_RAM_DATA_IN*) os dados contidos em memória. No terceiro ciclo de clock da instrução, o barramento de endereço de

memória (*bus_RAM_ADDRESS*) recebe do registrador de endereço (*END*) os dados contidos nele. No quarto ciclo de clock da instrução, o contador de programas (*PC*) é incrementado em 1, o registrador *Rn* entre os bits 9:7 do registrador de instruções (*IR*), *Rn[IR[9:7]]*, recebe do barramento de entrada de dados (*bus_RAM_DATA_IN*) os dados contidos em memória, o registrador de instruções (*processing_instruction*) recebe o valor de 0 (Zero) e (*resetStage*) para o valor de 1(um), fazendo assim o processador aguardar uma nova instrução.

Na Figura 79 temos um exemplo do que é realizado pela instrução *LOAD*, neste exemplo é realizado o carregamento da informação *01101111 01101001* na posição 50 da Memória para o Registrador *Rn[IR[9:7]]*.

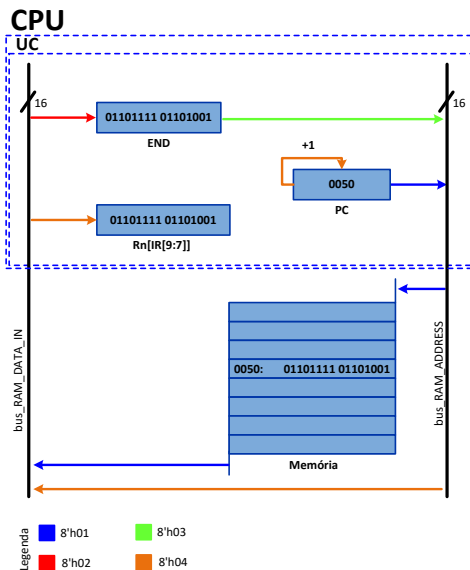


Figura 79: Exemplo: Load 50 em Rn[IR[9:7]]

Instrução STORE

A instrução STORE (16'b110001?????????) utiliza o modo de endereçamento direto, neste modo de endereçamento o endereço efetivo de memória que deve ser usado pela operação para ler o operando e/ou guardar o resultado é especificado na própria instrução, esta instrução é executada entre um registrador e a memória, realiza a cópia na memória de uma informação contida em um registrador, seu código 5 é declarado a seguir:

```

1      16'b110001?????????: begin
2      // 'instruction_store;=====
3      case x(stage)
4          8'h01: begin
5              bus_RAM_ADDRESS=PC;
6          end
7          8'h02: begin
8              END=bus_RAM_DATA_IN;
9          end
10         8'h03: begin
11             bus_RAM_ADDRESS=END;
12         end
13         8'h04: begin
14             wire_RW=1'b1;
15             bus_RAM_DATA_OUT=Rn[IR[9:7]];
16         end
17         8'h05: begin
18             PC=PC+16'h0001;
19             wire_RW=1'b0;
20             processing_instruction=1'b0;
21             resetStage=1'b1;
22         end
23     endcase
24 end

```

Código 5: Instrução STORE

Podemos observar que a instrução STORE necessita de 5 ciclos de clock para ser executada.

No primeiro ciclo de clock da instrução, o barramento de endereço de memória (*bus_RAM_ADDRESS*) recebe do contador de programas (*PC*), o endereço de memória da próxima instrução a ser processada. No segundo ciclo de clock da instrução, o registrador de endereço (*END*) recebe do

barramento de entrada de dados (*bus_RAM_DATA_IN*) os dados contidos em memória. No terceiro ciclo de clock da instrução, o barramento de endereço de memória (*bus_RAM_ADDRESS*) recebe do registrador de endereço (*END*) os dados contidos neste. No quarto ciclo de clock da instrução, o barramento de gravação (*wire_RW*) recebe o valor de 1, habilitando a gravação de dados na memória RAM, o barramento de saída de dados (*bus_RAM_DATA_OUT*) recebe os dados contidos no registrador *Rn[IR[9:7]]*, gravando-os na memória RAM. No quinto ciclo de clock da instrução, o contador de programas (*PC*) é incrementado em 1 (um), o barramento de gravação (*wire_RW*) irá receber o valor de 0 (zero), desabilitando a gravação de dados na memória RAM, o registrador de instruções (*processing_instruction*) recebe o valor de 0 (zero) e (*resetStage*) para o valor de 1 (um), fazendo assim o processador aguardar uma nova instrução.

Na Figura 80 temos um exemplo do que é realizado pela instrução STORE, neste exemplo é realizado o carregamento da informação *01101111 01101001* contida no Registrador *Rn[IR[9:7]]* na posição 4 da Memória RAM.

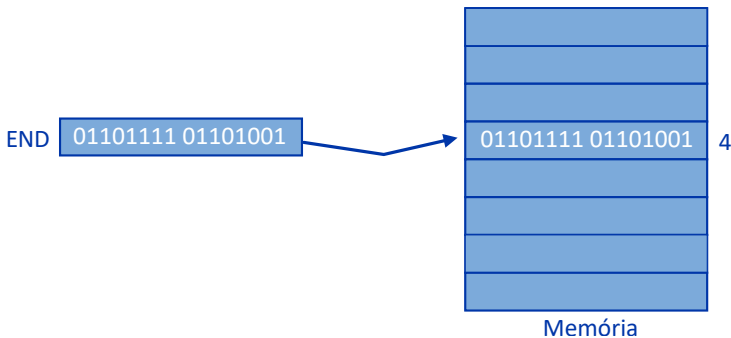


Figura 80: Exemplo: STORE *Rn[IR[9:7]]* em 4

Instrução STOREI

A instrução STOREI (*16'b111101????????*) utiliza o modo de endereçamento indireto, neste modo de endereçamento o endereço efetivo de memória que deve ser usado pela operação

para ler o operando e/ou guardar o resultado está armazenado em um Registrador, ou seja, está instrução carrega na memória a informação contida em um registrador cujo endereço é repassado por outro registrador, seu código 6 é declarado a seguir:

```

1      16'b111101?????????: begin
2          //'instruction_storei;=====
3      case x(stage)
4          8'h01: begin
5              bus_RAM_ADDRESS=Rn[IR[9:7]];
6          end
7          8'h02: begin
8              wire_RW=1'b1;
9              bus_RAM_DATA_OUT=Rn[IR[6:4]];
10         end
11         8'h03: begin
12             wire_RW=1'b0;
13             processing_instruction=1'b0;
14             resetStage=1'b1;
15         end
16     endcase
17 end

```

Código 6: Instrução STOREI

Podemos observar que a instrução STOREI necessita de 3 ciclos de clock para ser executada.

No primeiro ciclo de clock da instrução, o barramento de endereço de memória *bus_RAM_ADDRESS* recebe do registrador *Rn[IR[9:7]]* o endereço de memória da próxima instrução a ser executada pelo processador, neste caso o endereço onde será gravada a próxima informação. No segundo ciclo de clock da instrução, o barramento *wire_RW* receberá o valor de 1 (um), sinalizando a memória RAM que está deve ser habilitada para a gravação de dados, o barramento de saída de dados de memória RAM *bus_RAM_DATA_OUT* receberá os dados contidos no registrador *Rn[IR[6:4]]* gravando-os no endereço de memória repassado no ciclo anterior da instrução. No terceiro ciclo de clock da instrução, o barramento *wire_RW* receberá o valor de 0 (zero), isto irá sinalizar o termino da gravação de dados em memória RAM, desabilitando-a para a gravação de dados, o registrador de instruções (*processing_instruction*) recebe o valor de 0

(zero) e (*resetStage*) para o valor de 1 (um), fazendo assim o processador aguardar uma nova instrução.

Na Figura 81 temos um exemplo do que é realizado pela instrução STOREI, neste exemplo é realizado o carregamento da informação *01101111 01101001* contida no registrador *Rn[IR[6:4]]* na posição de memória cujo endereço é o conteúdo do registrador *Rn[IR[9:7]]*.

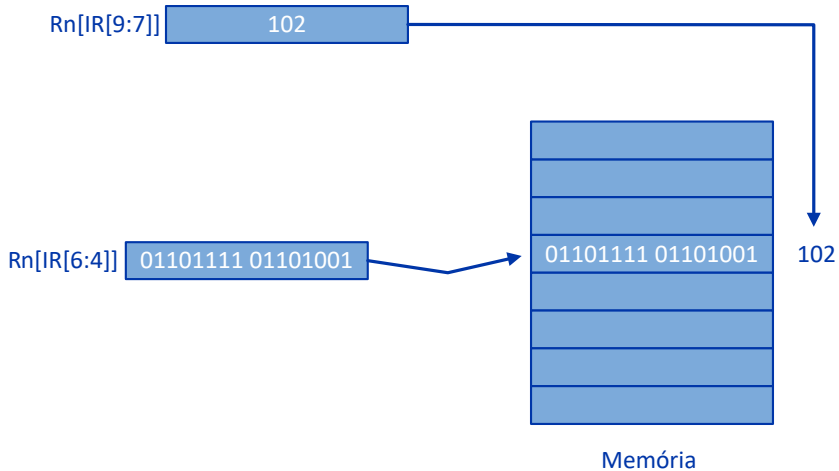


Figura 81: Exemplo: STOREI de *Rn[IR[6:4]]* no endereço *Rn[IR[9:7]]*

Instrução LOADI

A instrução LOADI (*16'b111100????????*) utiliza o modo de endereçamento indireto, neste modo de endereçamento o endereço efetivo de memória que deve ser usado pela operação para ler o operando e/ou guardar o resultado está armazenado em um Registrador, ou seja, está instrução recebe da memória a informação contida em um registrador cujo endereço e repassado por outro registrador, está instrução cópia em um registrador a informação contida em uma posição de memória repassado por outro registrador, seu código 7 é declarado a seguir:

```

1      16'b111100?????????: begin
2      // 'instruction_loadi;=====
3      case x(stage)
4          8'h01: begin
5              bus_RAM_ADDRESS=Rn[IR[6:4]];
6          end
7          8'h02: begin
8              Rn[IR[9:7]]=bus_RAM_DATA_IN;
9              processing_instruction=1'b0;
10             resetStage=1'b1;
11         end
12     endcase
13 end

```

Código 7: Instrução LOADI

A instrução LOADI necessita de 2 ciclos de clock para ser executada.

No primeiro ciclo de clock da instrução, o barramento de endereço de memória *bus_RAM_ADDRESS* recebe do registrador *Rn[IR[6:4]]* o endereço de memória da próxima instrução a ser executada pelo processador, neste caso o endereço onde será gravada a próxima informação. No segundo ciclo de clock da instrução, o registrador *Rn[IR[9:7]]* recebe do barramento de entrada de dados da memória RAM *bus_RAM_DATA_IN* os dados contidos na posição de memória RAM repassados anteriormente, os registradores, de instruções (*processing_instruction*) recebe o valor de 0 (zero) e de estágio (*resetStage*) recebe o valor de 1 (um), fazendo assim o processador aguardar uma nova instrução.

Na Figura 82 temos um exemplo do que é realizado pela instrução LOADI, neste exemplo é realizado o carregamento da informação 01101111 01101001 conteúdo da posição de memória cujo endereço é indicado no registrador *Rn[IR[6:4]]* no registrador *Rn[IR[9:7]]*.

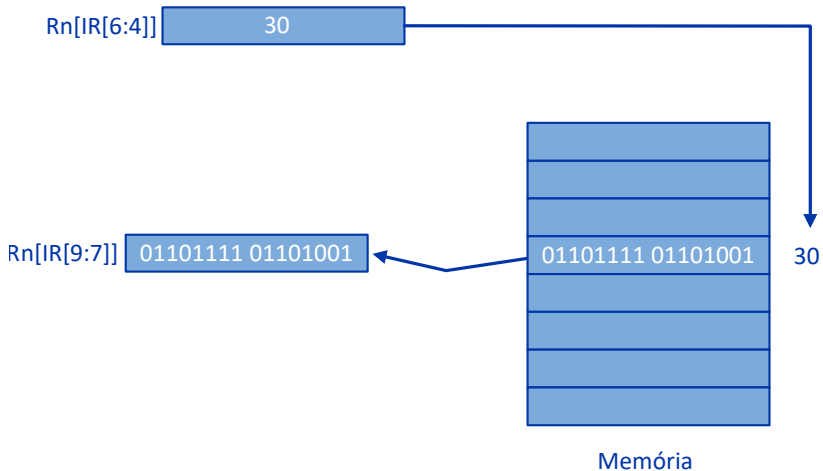


Figura 82: Exemplo: LOADI de `Rn[IR[6:4]]` em `Rn[IR[9:7]]`

Instrução LOADN

A instrução LOADN (*16'b111000????????*) utiliza o modo de endereçamento imediato, neste modo de endereçamento a informação a ser armazenada em um registrador é um operando que se encontra dentro do próprio código da instrução, ou seja, está instrução carrega em um registrador um dado que está imediatamente disponível na própria instrução (sem ir a memória), seu código 8 é declarado a seguir:

```

1      16'b111000?????????: begin
2          // 'instruction_loadn;=====
3      case x(stage)
4          8'h01: begin
5              bus_RAM_ADDRESS=PC;
6          end
7          8'h02: begin
8              Rn[IR[9:7]]=bus_RAM_DATA_IN;
9              PC=PC+16'h0001;
10             processing_instruction=1'b0;

```

```
11         resetStage=1'b1;
12     end
13     endcase
14 end
```

Código 8: Instrução LOADN

A instrução LOADN necessita de 2 ciclos de clock para ser executada. No primeiro ciclo de clock da instrução, o barramento de endereço de memória *bus_RAM_ADDRESS* recebe os dados do contador de programas *PC*, neste caso a informação que será gravada no próximo ciclo. No segundo ciclo de clock da instrução, o registrador **Rn[IR[9:7]]** recebe do barramento de entrada de dados *bus_RAM_DATA_IN* os dados repassados anteriormente, contidos no registrador *PC*, os registradores de instruções (*processing_instruction*) recebe o valor de 0 (zero) e de estágio (*resetStage*) recebe o valor de 1 (um), fazendo assim o processador aguardar uma nova instrução.

Na Figura 83 temos um exemplo do que é realizado pela instrução LOADN, neste exemplo é realizado o carregamento imediato da informação 0198 conteúdo no registrador PC no registrador Rn[IR[9:7]].

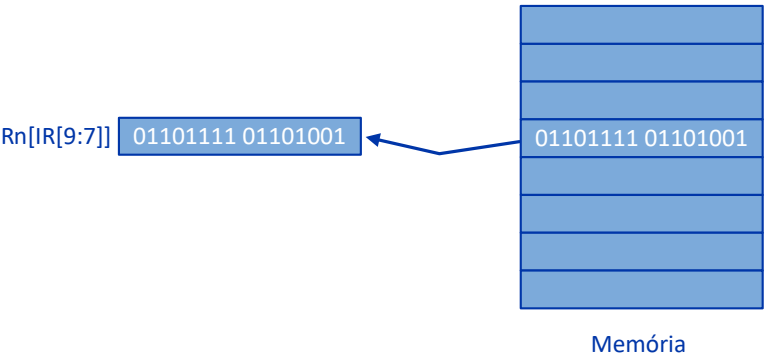


Figura 83: Exemplo: LOADN em Rn[IR[9:7]]

Instrução MOV

A instrução MOV (*16'b110011????????*) é extremamente poderosa e versátil, sendo amplamente usada nos mais diversos tipos de endereçamentos, esta instrução realiza a movimentação (cópia) de dados entre registradores de mesmo tamanho, ou seja, embora a instrução MOV lembre a palavra mover (ou move, em inglês), o que ela faz na verdade é uma cópia da informação de um lugar para outro, registrador-registrador, e não movimentação, pois os dados permanecem inalterada na origem, seu código 9 é declarado a seguir:

```

1      16'b110011?????????: begin
2      // 'instruction_mov;=====
3      casex(stage)
4      8'h01: begin
5          casex(IR[1:0])
6              2'b?0: begin
7                  Rn[IR[9:7]] = Rn[IR[6:4]];
8              end
9              2'b01: begin
10                 Rn[IR[9:7]] = SP;
11             end
12             2'b11: begin
13                 SP = Rn[IR[9:7]];
14             end
15         endcase
16     end
17     8'h02: begin
18         processing_instruction = 1'b0;
19         resetStage = 1'b1;
20     end
21 endcase
22 end

```

Código 9: Instrução MOV

A instrução MOV necessita de 2 ciclos de clock para ser executada e possui 3 casos de execução.

No primeiro ciclo de clock da instrução é verificado o valor contido no registrador de instruções *IR[1:0]* para verificar qual ação será tomada.

No primeiro caso, se o valor de $IR[1:0]$ for igual a $2'b?0$, então, a instrução realizará a ação de copiar os dados do registrador $Rn[IR[6:4]]$ para o registrador $Rn[IR[9:7]]$. No segundo caso, quando o valor de $IR[1:0]$ for igual a $2'b01$, então, a instrução realizará a ação de copiar os dados do registrador de ponteiro SP para o registrador $Rn[IR[9:7]]$. Já no terceiro caso, uma vez que o valor de $IR[1:0]$ for igual a $2'b11$, então, a instrução realizará a ação de copiar os dados do registrador $Rn[IR[9:7]]$ para o registrador de ponteiro SP , nota-se que nos 3 casos os registradores possuem o mesmo tamanho.

No segundo ciclo de clock da instrução, os registradores de instruções (*processing_instruction*) recebe o valor de 0 (zero) e de estágio (*resetStage*) recebe o valor de 1 (um), fazendo assim o processador aguardar uma nova instrução.

Na Figura 84 temos um exemplo do que é realizado pela instrução MOV, neste exemplo é realizado a movimentação dos dados entre os registradores.

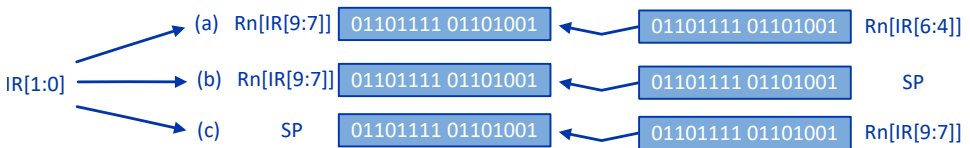


Figura 84: Exemplo: MOV entre Registradores

Instruções Aritméticas

O Processador AP9 possui implementado 9 instruções Aritméticas, são elas: ADD, ADDC, SUB, MULT, DIV, INC, DEC e MOD.

As instruções Aritméticas como o nome sugere, são operações fundamentais da matemática, elas consistem da adição, subtração, multiplicação, divisão e etc entre operandos, são operações Registro-Registro, ou seja, os operandos são informações armazenadas nos registradores, são instruções em que seus operandos são processados pela ULA, são operações que são afetadas pelo estado dos flags e de também alterá-los de acordo com a instrução e o resultado obtido seu formato é apresentado na Figura ??

Instrução ADD/ADDC

A instrução ADD e ADDC (16'b100000?????????) realiza a soma e soma com Carry (+) de dados contidos em registradores, ao receber uma instrução de ADD/ADDC a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução ADD/ADDC no módulo UC

```

1      16'b100000?????????: begin
2          //'instructions_add_and_addc;=====
3      case x(stage)
4          8'h01: begin
5              m3=Rn[IR[6:4]];
6              m4=Rn[IR[3:1]];
7              enable_alu=1'b1;
8              useCarry=IR[0];
9              opcode=IR[15:10];
10         end
11         8'h06: begin
12             Rn[IR[9:7]]=m2;
13             enable_alu=1'b0;
14             processing_instruction=1'b0;
15             resetStage=1'b1;
16         end
17     endcase
18 end

```

Código 10: Instrução ADD/ADDC na UC

A instrução ADD/ADDC no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:
 - (a) O registrador de saída **m3** recebe os dados do registrador **Rn[IR[6:4]]**.
 - (b) O registrador de saída **m4** recebe os dados do registrador **Rn[IR[3:1]]**.
 - (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
 - (d) O barramento **useCarry** irá receber o valor do registrador **IR[0]**.
 - (e) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b100000**.
2. No Segundo estágio da instrução:
 - (a) o registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.

- (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitação a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução ADD/ADDC no módulo ULA

```

1      6'b100000: begin
2          // 'instructions_add_and_addc;=====
3      casex(stage)
4          8'h01: begin
5              if (useCarry==1'b1) begin
6                  temp=32'h00000000+m3+m4+FR_in[11];
7              end
8              else begin
9                  temp=32'h00000000+m3+m4;
10             end
11         end
12         8'h02: begin
13             if (temp>32'h0000ffff) begin
14                 FR_out[11]=1'b1;
15             end
16             else begin
17                 FR_out[11]=1'b0;
18             end
19             m2=temp[15:0];
20             if (temp[15:0]==16'h0000) begin
21                 FR_out[12]=1'b1;
22             end
23             else begin
24                 FR_out[12]=1'b0;
25             end
26             resetStage=1'b1;
27         end
28     endcase
29 end

```

Código 11: Instrução ADD/ADDC na ULA

A instrução ADD/ADDC no módulo ULA possui 2 estágios:

1. No Primeiro estágio da instrução temos 2 casos:

- (a) Se **useCarry** for igual a 1, então, o registrador **temp** recebe a soma entre **32'h00000000**, os registradores **m3**, **m4** e a Flag **FR_in[11]**.
- (b) Caso contrario, se **useCarry** for igual a 0, então, o registrador **temp** recebe a soma entre **32'h00000000**, os registradores **m3** e **m4**.

2. No Segundo estágio da instrução temos 2 verificações:

- (a) Na Primeira verificação, se **temp** é maior que **32'h0000ffff**, então, a Flag **FR_out[11]** recebe 1, caso contrário a Flag **FR_out[11]** recebe 0.
- (b) Então o registrador **m2** recebe os dados do registrador **temp[15:0]**.
- (c) Na Segunda verificação, se **temp[15:0]** for igual a **16'h0000**, a Flag **FR_out[12]** recebe 1, caso contrário recebe 0.
- (d) A variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 85 temos um exemplo do que é realizado pela instrução ADD/ADDC (SOMA), com ou sem Carry, neste exemplo, a Unidade de Controle através dos registradores M3 e M4, repassa a ULA os dados, está realizará a operação de soma e retornará o resultado através do registrador M2 à Unidade de Controle.

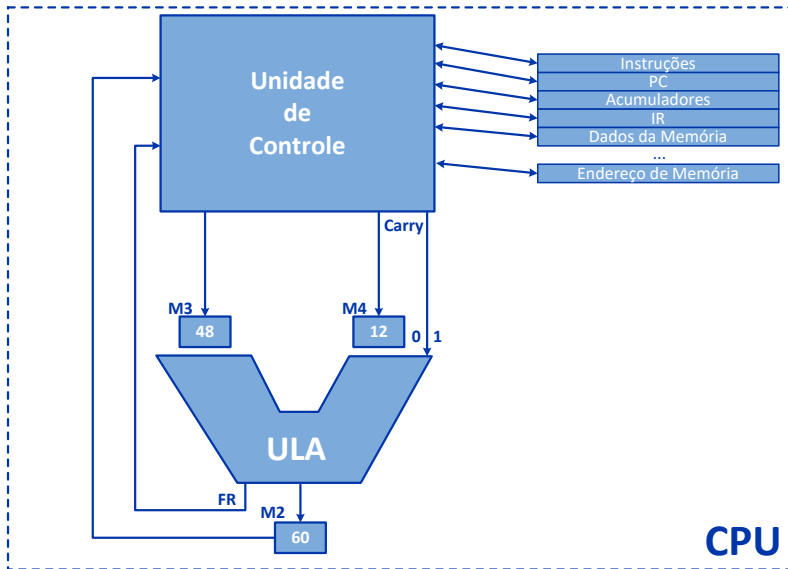


Figura 85: Exemplo: ADD M3=48 e M4=12, retorna M2=60

Instrução SUB

A instrução SUB e SUBC (16'b100001????????) realiza a subtração (-) e subtração com Carry (ADDC) de dados contidos em registradores, ao receber uma instrução de SUB/SUBC a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução SUB no módulo UC

```

1      16'b100001?????????: begin
2          // 'instructions_sub_and_subc;=====
3      case x(stage)
4          8'h01: begin
5              m3=Rn[IR[6:4]];
6              m4=Rn[IR[3:1]];
7              enable_alu=1'b1;

```



```

8      useCarry=IR[0];
9      opcode=IR[15:10];
10
11      end
12      8'h06: begin
13          Rn[IR[9:7]]=m2;
14          enable_alu=1'b0;
15          processing_instruction=1'b0;
16          resetStage=1'b1;
17      end
18      endcase
19  end

```

Código 12: Instrução SUB na UC

A instrução SUB/SUBC no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:

- (a) O registrador de saída **m3** recebe os dados do registrador **Rn[IR[6:4]]**.
- (b) O registrador de saída **m4** recebe os dados do registrador **Rn[IR[3:1]]**.
- (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
- (d) O barramento **useCarry** irá receber o valor do registrador **IR[0]**.
- (e) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b100001**.

2. No Segundo estágio da instrução:

- (a) o registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
- (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitação a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução SUB no módulo ULA

```

1      6'b100001: begin
2          // 'instructions_sub_and_subc ;=====
3          casex(stage)
4              8'h01: begin

```

```
5         addInv=-m4;
6     end
7     8'h02: begin
8         if (useCarry==1'b1) begin
9             temp=m3+addInv+FR_in[11];
10        end
11        else begin
12            temp=m3+addInv;
13        end
14    end
15    8'h03: begin
16        if (useCarry==1'b0) begin
17            if (m4>m3) begin
18                FR_out[6]=1'b1;
19                m2=16'h0000;
20            end
21            else begin
22                FR_out[6]=1'b0;
23                m2=temp[15:0];
24            end
25        end
26        if (useCarry==1'b1) begin
27            if (m4>(m3+FR_in[11])) begin
28                FR_out[6]=1'b1;
29                m2=16'h0000;
30            end
31            else begin
32                FR_out[6]=1'b0;
33                m2=temp[15:0];
34            end
35        end
36        if (temp[15:0]==16'h0000) begin
37            FR_out[12]=1'b1;
38        end
39        else begin
40            FR_out[12]=1'b0;
41        end
42        resetStage=1'b1;
43    end
44 endcase
```

Código 13: Instrução SUB na ULA

A instrução SUB/SUBC no módulo ULA possui 3 estágios:

1. No Primeiro estágio da instrução, o registrador **addInv** recebe **M4** negativo
2. No Segundo estágio da instrução temos 2 casos:
 - (a) Se **useCarry** for igual a 1, então, o registrador **temp** recebe a soma entre os registradores **m3**, **addInv** e a Flag **FR_in[11]**.
 - (b) Caso contrário, se **useCarry** for igual a 0, então, o registrador **temp** recebe a soma entre os registradores **m3** e **addInv**.
3. No Terceiro estágio da instrução temos 3 verificações:
 - (a) Na Primeira verificação, se **useCarry** for igual a 0:
 - i. Se **M4** for maior que **M3**, então, a Flag de saída **FR_out[6]** recebe o valor de 1 e o registrador **M2** recebe o valor de **16'h0000**.
 - ii. Caso contrário a Flag de saída **FR_out[6]** recebe o valor de 0 e o registrador **M2** recebe o registrador **temp[15:0]**.
 - (b) Na Segunda verificação, se **useCarry** for igual a 1:
 - i. Se **M4** for maior que **M3** somado a Flag **FR_in[11]**, então, a Flag de saída **FR_out[6]** recebe o valor de 1 e o registrador **M2** recebe o valor de **16'h0000**.
 - ii. Caso contrário a Flag de saída **FR_out[6]** recebe o valor de 0 e o registrador **M2** recebe o registrador **temp[15:0]**.
 - (c) Na Terceira verificação, se o registrador **temp[15:0]** for igual a **16'h0000** a Flag de saída **FR_out[12]** recebe o valor de 1, caso contrário a Flag de saída **FR_out[12]** recebe o valor de 0.
 - (d) Terminada as verificações a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 86 temos um exemplo do que é realizado pela instrução SUB/SUBC (SUBTRAÇÃO), com ou sem Carry, neste exemplo, a Unidade de Controle através dos registradores M3 e M4, repassa a ULA os dados, está realizará a operação de subtração e retornará o resultado através do registrador M2 à Unidade de Controle.

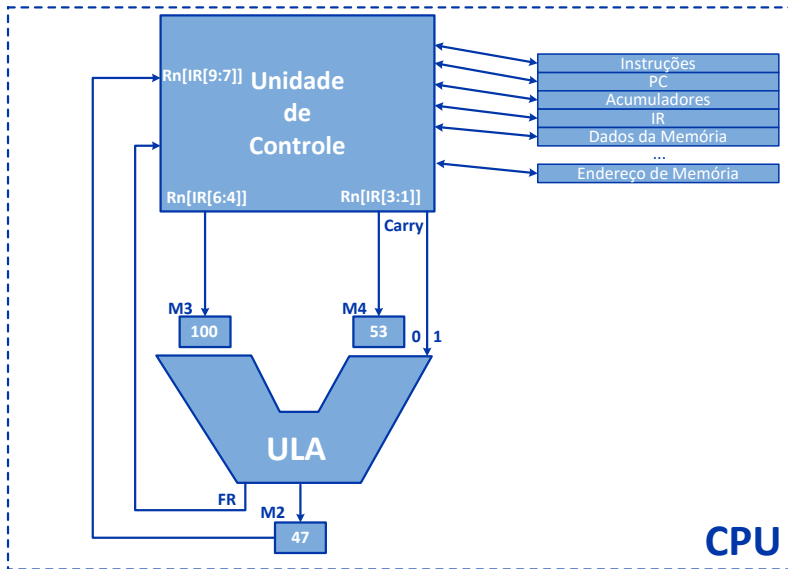


Figura 86: Exemplo: SUB M3=100 e M4=53, retorna M2=47

Instrução MULT

A instrução MULT (16'b100010?????????) realiza a Multiplicação (*) de dados contidos em registradores, ao receber uma instrução de MULT a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução MULT no módulo UC

```

1      16'b100010?????????: begin
2          // 'instruction_mul;=====
3      case x (stage)
4          8'h01: begin
5              m3=Rn [ IR [ 6 : 4 ] ] ;
6              m4=Rn [ IR [ 3 : 1 ] ] ;
7              enable_alu=1'b1;

```

```

8         opcode=IR[15:10];
9     end
10    8'h06: begin
11        Rn[IR[9:7]]=m2;
12        enable_alu=1'b0;
13        processing_instruction=1'b0;
14        resetStage=1'b1;
15    end
16    endcase
17    end

```

Código 14: Instrução MULT na UC

A instrução MULT no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:
 - (a) O registrador de saída **m3** recebe os dados do registrador **Rn[IR[6:4]]**.
 - (b) O registrador de saída **m4** recebe os dados do registrador **Rn[IR[3:1]]**.
 - (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
 - (d) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b100010**.
2. No Segundo estágio da instrução:
 - (a) o registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
 - (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
 - (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitação a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução MULT no módulo ULA

```

1        6'b100010: begin
2            // 'instruction_mul; =====
3            casex(stage)
4                8'h01: begin
5                    temp=m3*m4;
6                end

```

```

7      8'h02: begin
8          m2=temp[15:0];
9          if (temp>32'h0000ffff) begin
10             FR_out[10]=1'b1;
11         end
12         else begin
13             FR_out[10]=1'b0;
14         end
15         if (temp[15:0]==16'h0000) begin
16             FR_out[12]=1'b1;
17         end
18         else begin
19             FR_out[12]=1'b0;
20         end
21         resetStage=1'b1;
22     end
23 endcase
24 end

```

Código 15: Instrução MULT na ULA

A instrução MULT no módulo ULA possui 2 estágios:

1. No Primeiro estágio da instrução, o registrador **temp** recebe a multiplicação dos dados contidos nos registradores **M3** e **M4**.
2. No Segundo estágio da instrução, o registrador **M2** recebe a multiplicação contida no registrador **temp[15:0]** e são realizadas 2 verificações:
 - (a) Na Primeira verificação, se **temp** for maior que **32'h0000ffff**, a Flag de saída **FR_out[10]** recebe o valor de 1, caso contrário, a Flag de saída **FR_out[10]** recebe o valor de 0.
 - (b) Na Segunda verificação, se **temp** for igual a **16'h0000**, então, a Flag de saída **FR_out[12]** recebe o valor de 1, caso contrário, a Flag de saída **FR_out[10]** recebe o valor de 0.
3. Terminada as verificações a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 87 temos um exemplo do que é realizado pela instrução MULT (MULTIPLICAÇÃO), a Unidade de Controle através dos registradores M3 e M4, repassa a ULA os dados,

está realizando a operação de multiplicação e retornará o resultado através do registrador M2 à Unidade de Controle.

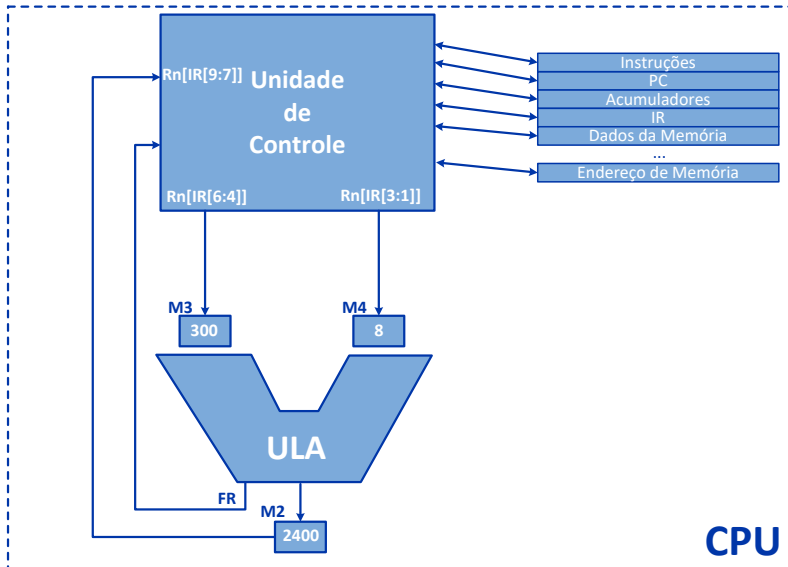


Figura 87: Exemplo: MULT M3=300 e M4=8, retorna M2=2400

Instrução DIV

A instrução DIV (16'b100011?????????) realiza a Divisão (/) de dados contidos em registradores, ao receber uma instrução de DIV a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução DIV no módulo UC

```

1 16'b100011?????????: begin
2  // 'instruction_div;=====
3  casex (stage)

```

```

4      8'h01: begin
5          m3=Rn[IR[6:4]];
6          m4=Rn[IR[3:1]];
7          enable_alu=1'b1;
8          opcode=IR[15:10];
9
10     end
11     8'h06: begin
12         Rn[IR[9:7]]=m2;
13         enable_alu=1'b0;
14         processing_instruction=1'b0;
15         resetStage=1'b1;
16     end
17 endcase
end

```

Código 16: Instrução DIV na UC

A instrução DIV no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:
 - (a) O registrador de saída **m3** recebe os dados do registrador **Rn[IR[6:4]]**.
 - (b) O registrador de saída **m4** recebe os dados do registrador **Rn[IR[3:1]]**.
 - (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
 - (d) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b100011**.
2. No Segundo estágio da instrução:
 - (a) o registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
 - (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
 - (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitação a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução DIV no módulo ULA

```

1      6'b100011: begin
2          // 'instruction_div;=====

```



```

3      casex(stage)
4          8'h01: begin
5              if(m4==16'h0000) begin
6                  FR_out[9]=1'b1;
7                  m2=16'h0000;
8              end
9              else begin
10                 m2=m3/m4;
11                 FR_out[9]=1'b0;
12             end
13         end
14         8'h02: begin
15             if(m2==16'h0000) begin
16                 FR_out[12]=1'b1;
17             end
18             else begin
19                 FR_out[12]=1'b0;
20             end
21             resetStage=1'b1;
22         end
23     endcase
24 end

```

Código 17: Instrução DIV na ULA

A instrução DIV no módulo ULA possui 2 estágios:

1. No Primeiro estágio da instrução, se **M4** for igual a **16'h0000**, então, a Flag de saída **FR_out[9]** recebe o valor de 1 e o registrador **M2** o valor de **16'h0000**, caso contrário, o registrador **M2** recebe a divisão entre os dados contidos nos registradores **M3** e **M4**, e a Flag de saída **FR_out[9]** recebe o valor de 0.
2. No Segundo estágio da instrução, se **M2** for igual a **16'h0000**, então, a Flag de saída **FR_out[12]** recebe o valor de 1, caso contrário, a Flag de saída **FR_out[10]** recebe o valor de 0.
3. Terminada o estágio a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 88 temos um exemplo do que é realizado pela instrução DIV (DIVISÃO), a Unidade de Controle através dos registradores M3 e M4, repassa a ULA os dados, está

realizará a operação de multiplicação e retornará o resultado através do registrador M2 à Unidade de Controle.

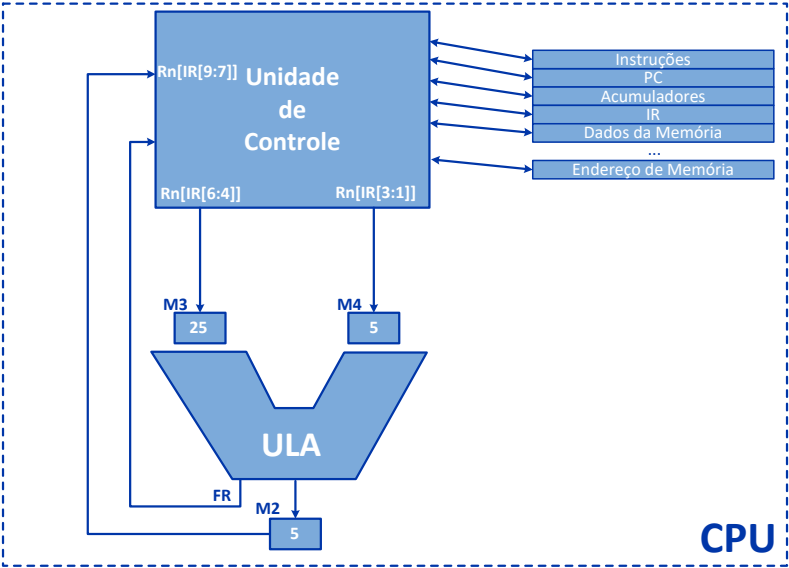


Figura 88: Exemplo: DIV M3=25 e M4=5, retorna M2=5

Instrução INC e DEC

A instrução INC e DEC (16'b100100?????????) realiza o Incremento (++) ou Decremento (–) dos dados contidos em registradores, ao receber uma instrução de INC e DEC a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução INC e DEC no módulo UC

```
1 16'b100100?????????: begin
2 // 'instructions_inc_and_Dec; =====
3 case x (stage)
```

```

4      8'h01: begin
5          m3=Rn[IR[9:7]];
6          enable_alu=1'b1;
7          opcode=IR[15:10];
8          useDec=IR[6];
9
10         end
11     8'h06: begin
12         Rn[IR[9:7]]=m2;
13         enable_alu=1'b0;
14         processing_instruction=1'b0;
15         resetStage=1'b1;
16     end
17 endcase
end

```

Código 18: Instrução INC e DEC na UC

A instrução INC e DEC no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:

- (a) O registrador de saída **m3** recebe os dados do registrador **Rn[IR[9:7]]**.
- (b) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
- (c) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b100100**.
- (d) O barramento **useDec** irá receber o valor do registrador **IR[6]**.

2. No Segundo estágio da instrução:

- (a) O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
- (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução INC e DEC no módulo ULA

```

1      6'b100100: begin
2          // 'instructions_inc_and_Dec;=====

```

```
3      casex(stage)
4      8'h01: begin
5          if(dec ==1'b0) begin
6              m2=m3+16'h0001;
7          end
8          else begin
9              m2=m3-16'h0001;
10         end
11     end
12     8'h02: begin
13         if(dec ==1'b0) begin
14             FR_out[12]=1'b0;
15         end
16         else begin
17             if(m2==16'h0000) begin
18                 FR_out[12]=1'b1;
19             end
20             else begin
21                 FR_out[12]=1'b0;
22             end
23         end
24         resetStage=1'b1;
25     end
26 endcase
27 end
```

Código 19: Instrução INC e DEC na ULA

A instrução INC e DEC no módulo ULA possui 2 estágios:

1. No Primeiro estágio da instrução, se a **dec** for igual a 0, então, o registrador **M2** recebe a soma do valor contido no registrador **M3** mais 1, realizando o Incremento dos dados, caso contrário, o registrador **M2** recebe a subtração do valor contido no registrador **M3** menos 1, realizando o Decremento dos dados.
2. No Segundo estágio da instrução, temos 2 verificações, se a **dec** for igual a 0, então, a Flag de saída **FR_out[12]** recebe o valor de 0, caso contrário:
 - (a) Se o registrador **M2** for igual a **16'h0000**, então, a Flag de saída **FR_out[12]** recebe o valor de 1, caso não seja nenhum dos casos anteriores a Flag de saída **FR_out[12]** recebe o valor de 0.

3. Terminada o estágio a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 89 temos um exemplo do que é realizado na instrução de Incremento e Decremento de dados, a Unidade de Controle através dos registradores **M3**, repassa a ULA o dado, está realizando a operação de incremento ou decremento conforme o bit **useDec** e retornará o resultado através do registrador **M2** à Unidade de Controle.

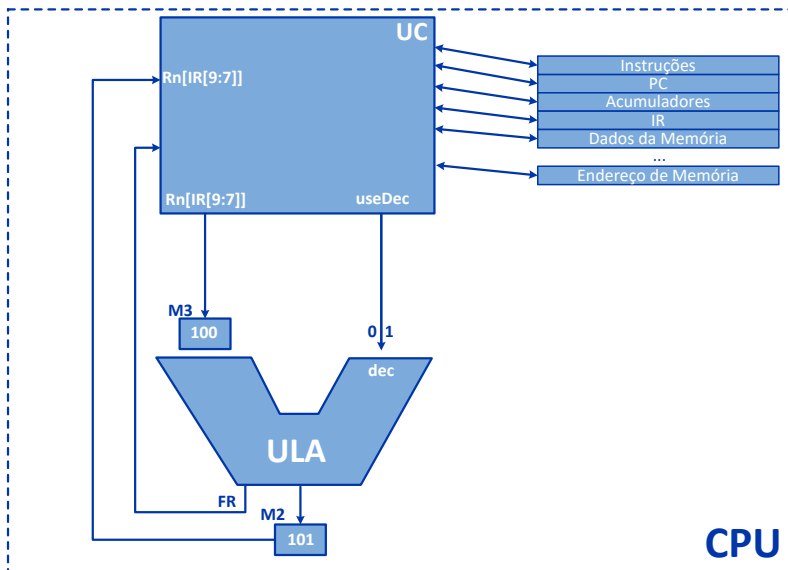


Figura 89: Exemplo: INC M3=100 e useDec=0, retorna M2=101

Instrução MOD

A instrução MOD (16'b100101????????) realiza o resto da divisão entre inteiros (%) dos dados contidos em registradores, ao receber uma instrução de MOD a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução MOD no módulo UC

```

1      16'b100101?????????: begin
2      //'instruction_mod;=====
3      casex(stage)
4          8'h01: begin
5              m3=Rn[IR[6:4]];
6              m4=Rn[IR[3:1]];
7              enable_alu=1'b1;
8              opcode=IR[15:10];
9          end
10         8'h06: begin
11             Rn[IR[9:7]]=m2;
12             enable_alu=1'b0;
13             processing_instruction=1'b0;
14             resetStage=1'b1;
15         end
16     endcase
17 end

```

Código 20: Instrução MOD na UC

A instrução MOD no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:

- O registrador de saída **M3** recebe os dados do registrador **Rn[IR[6:4]]**.
- O registrador de saída **M4** recebe os dados do registrador **Rn[IR[3:1]]**.
- O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
- O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b100101**.

2. No Segundo estágio da instrução:

- O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
- O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução MOD no módulo ULA

```

1      6'b100101: begin
2          // 'instruction_mod;=====
3      casex(stage)
4          8'h01: begin
5              if(m4==16'h0000) begin
6                  FR_out[9]=1'b1;
7                  m2=16'h0000;
8              end
9              else begin
10                 m2=m3%m4;
11                 FR_out[9]=1'b0;
12             end
13         end
14         8'h02: begin
15             if(m2==16'h0000) begin
16                 FR_out[12]=1'b1;
17             end
18             else begin
19                 FR_out[12]=1'b0;
20             end
21             resetStage=1'b1;
22         end
23     endcase
24 end

```

Código 21: Instrução MOD na ULA

A instrução MOD no módulo ULA possui 2 estágios:

1. No Primeiro estágio da instrução é verificado se o registrador **M4** é igual a **16'h0000**, caso positivo, a Flag de saída **FR_out[9]** recebe o valor de 1, o registrador **M2** recebe o valor de **16'h0000**, caso contrário, o registrador **M2** recebe o resto da divisão dos registradores **M3** e **M4**, a Flag de saída **FR_out[9]** recebe o valor de 0.
2. No Segundo estágio da instrução é verificado se o registrador **M2** é igual a **16'h0000**, caso positivo, a Flag de saída **FR_out[12]** recebe o valor de 1, caso contrário, a Flag de saída **FR_out[12]** recebe o valor de 0.
3. Terminado os estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da

instrução e então retornando o processo para a UC.

Na Figura 90 temos um exemplo do que é realizado na instrução de resto da divisão entre inteiros de dados, a Unidade de Controle através dos registradores **M3** e **M4**, repassa a ULA os dados, está realizando a operação de resto da divisão e retornará o resultado através do registrador **M2** à Unidade de Controle.

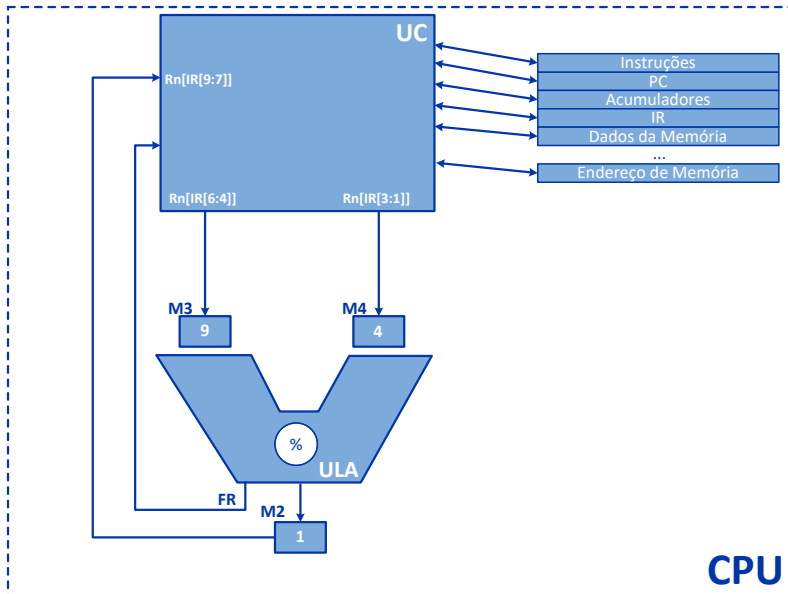


Figura 90: Exemplo: MOD M3=9 e M4=4, retorna M2=1

Instruções Lógicas

O Processador AP9 possui implementado 7 instruções Lógicas, são elas: CMP, AND, OR, XOR, NOT, SHIFT e ROTS.

As instruções lógicas, são instruções em que seus operandos são processados pela ULA,

são operações bit a bit, incluindo AND, NOT, OR, XOR estas operações realizam a mudança de bits armazenadas nos registradores, são operações que são afetadas pelo estado dos flags e de também alterá-los de acordo com a instrução e o resultado obtido, seu formato é apresentado na Figura 91.

Instrução CMP

A instrução CMP (16'b010110????????) realiza comparação entre dados contidos em registradores, ao receber uma instrução de CMP a UC inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução CMP no módulo UC

```

1      16'b010110?????????: begin
2      // 'instruction_cmp;=====
3      casex(stage)
4          8'h01: begin
5              m3=Rn[IR[9:7]];
6              m4=Rn[IR[6:4]];
7              enable_alu=1'b1;
8              opcode=IR[15:10];
9          end
10         8'h06: begin
11             bus_vga_char=FR_in_at_control;
12             enable_alu=1'b0;
13             processing_instruction=1'b0;
14             resetStage=1'b1;
15         end
16     endcase
17 end

```

Código 22: Instrução CMP na UC

A instrução CMP no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:

- O registrador de saída **M3** recebe os dados do registrador **Rn[IR[9:7]]**.
- O registrador de saída **M4** recebe os dados do registrador **Rn[IR[6:4]]**.
- O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.

(d) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b010110**.

2. No Segundo estágio da instrução:

- (a) O registrador **bus_vga_char** recebe o valor da Flag de entrada **FR_in_at_control**.
- (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução CMP no módulo ULA

```

1      6'b010110: begin
2          //'instruction_cmp;=====
3      casex(stage)
4          8'h01: begin
5              if(m3 == m4) begin
6                  FR_out[15:13]=3'b001;
7              end
8              if(m3<m4) begin
9                  FR_out[15:13]=3'b010;
10             end
11             if(m3>m4) begin
12                 FR_out[15:13]=3'b100;
13             end
14             resetStage=1'b1;
15         end
16     endcase
17 end

```

Código 23: Instrução CMP na ULA

A instrução CMP no módulo ULA possui um estágio:

- Neste estágio serão realizadas 3 verificações:
 1. Se **M3** for igual a **M4**, então, a Flag de saída **FR_out[15:13]** recebe o valor de **3'b001**.

2. Se **M3** for menor que **M4**, então, a Flag de saída **FR_out[15:13]** recebe o valor de **3'b010**.
 3. Se **M3** for maior que **M4**, então, a Flag de saída **FR_out[15:13]** recebe o valor de **3'b100**.
- Terminado os estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 91 temos um exemplo do que é realizado na instrução de comparação de dados, a Unidade de Controle através dos registradores **M3** e **M4**, repassa a ULA os dados, está realizando a operação comparação e retornará o resultado através da Flag **FR_out[15:13]** à Unidade de Controle.

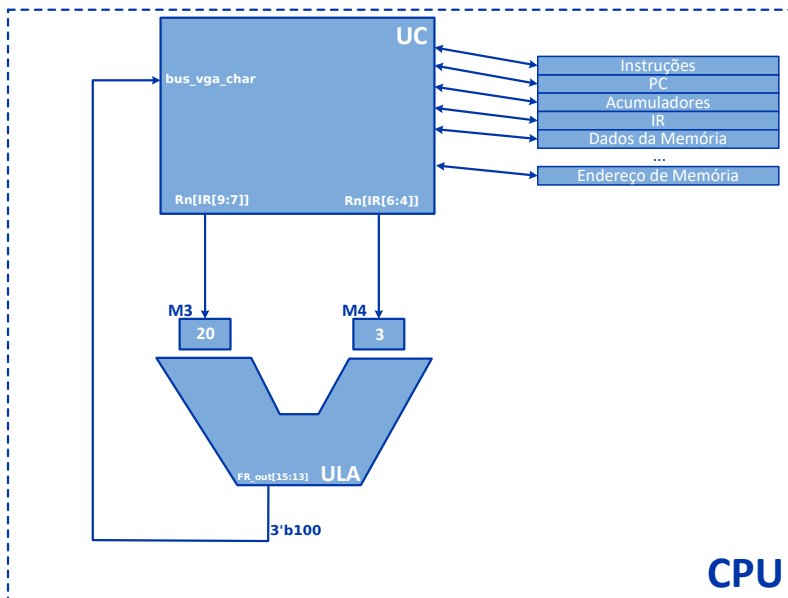


Figura 91: Exemplo: CMP M3=20 e M4=3, retorna M2=3'b100

Instrução AND

A instrução AND (16'b010010?????????) realiza a operação lógica "E", bit a bit, **AND** é usado para a junção de ideias, nesta instrução os receber os dados contidos em registradores, o resultado de dois bits de entrada e um de saída. Para que o bit de saída seja verdadeiro (valor 1) ambos os bits de entrada devem ser verdadeiros, ao receber uma instrução de AND a Unidade de Controle (UC) inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução AND no módulo UC

```

1      16'b010010?????????: begin
2      // 'instruction_and;=====
3      case x (stage)
4      8'h01: begin
5          m3=Rn [IR [6:4]] ;
6          m4=Rn [IR [3:1]] ;
7          enable_alu=1'b1 ;
8          opcode=IR [15:10] ;
9      end
10     8'h06: begin
11         Rn [IR [9:7]]=m2 ;
12         enable_alu=1'b0 ;
13         processing_instruction=1'b0 ;
14         resetStage=1'b1 ;
15     end
16     endcase
17     end

```

Código 24: Instrução AND na UC

A instrução AND no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:
 - (a) O registrador de saída **M3** recebe os dados do registrador **Rn[IR[6:4]]**.
 - (b) O registrador de saída **M4** recebe os dados do registrador **Rn[IR[3:1]]**.
 - (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
 - (d) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b010010**.
2. No Segundo estágio da instrução:

- (a) O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
- (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução AND no módulo ULA

```

1      6'b010010: begin
2          // 'instruction_and;=====
3      casex(stage)
4          8'h01: begin
5              m2=m3 & m4;
6          end
7          8'h02: begin
8              if (m2==16'h0000) begin
9                  FR_out[12]=1'b1;
10             end
11             resetStage=1'b1;
12         end
13     endcase
14 end

```

Código 25: Instrução AND na ULA

A instrução AND no módulo ULA possui 2 estágio:

- No Primeiro estágio, o registrador **M2** recebe os registradores **M3 & M4**.
- No Segundo estágio, se **M2** for igual a **16'h0000**, então a Flag de saída **FR_out[12]** recebe o valor de 1.
- Ao termino dos estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 92 temos um exemplo do que é realizado na instrução de operação lógica "E" de dados, a Unidade de Controle através dos registradores **M3** e **M4**, repassa a ULA os dados, está realizará a operação "E" e retornará o resultado através do Registrador **M2** à Unidade de Controle.

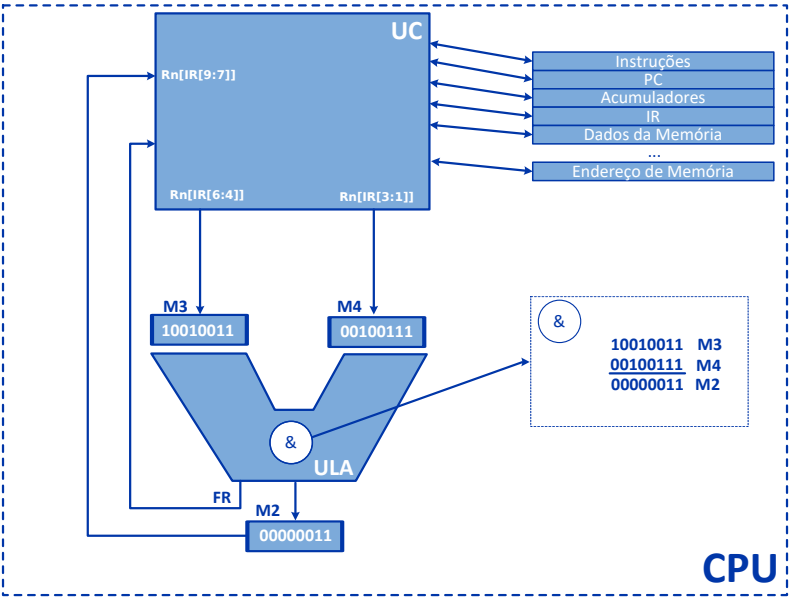


Figura 92: Exemplo: AND M3=10010011 e M4=00100111, retorna M2=00000011

Instrução OR

A instrução OR (16'b010011?????????) realiza a operação lógica "OU", bit a bit, **OR** tem a função de indicar escolha, nesta instrução os receber os dados contidos em registradores, o resultado de dois bits de entrada e um de saída. Para que o bit de saída seja verdadeiro (valor 1), pelo menos um dos bits de entrada precisa ser verdadeiro, ao receber uma instrução de OR a Unidade de Controle (UC) inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução OR no módulo UC

```
1 16'b010011?????????: begin
2  // 'instruction_or;=====
3  case x (stage)
```

```

4      8'h01: begin
5          m3=Rn[IR[6:4]];
6          m4=Rn[IR[3:1]];
7          enable_alu=1'b1;
8          opcode=IR[15:10];
9
10         end
11      8'h06: begin
12          Rn[IR[9:7]]=m2;
13          enable_alu=1'b0;
14          processing_instruction=1'b0;
15          resetStage=1'b1;
16
17         end
18     endcase
19 end

```

Código 26: Instrução OR na UC

A instrução OR no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:
 - (a) O registrador de saída **M3** recebe os dados do registrador **Rn[IR[6:4]]**.
 - (b) O registrador de saída **M4** recebe os dados do registrador **Rn[IR[3:1]]**.
 - (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
 - (d) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b010011**.
2. No Segundo estágio da instrução:
 - (a) O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
 - (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
 - (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução OR no módulo ULA

```

1      6'b010011: begin
2          //'instruction_or';=====

```

```
3      casex(stage)
4          8'h01: begin
5              m2=m3 | m4;
6          end
7          8'h02: begin
8              if(m2==16'h0000) begin
9                  FR_out[12]=1'b1;
10             end
11             resetStage=1'b1;
12         end
13     endcase
14 end
```

Código 27: Instrução OR na ULA

A instrução OR no módulo ULA possui 2 estágio:

- No Primeiro estágio, o registrador **M2** recebe os registradores **M3** | **M4**.
- No Segundo estágio, se **M2** for igual a **16'h0000**, então a Flag de saída **FR_out[12]** recebe o valor de 1.
- Ao termino dos estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 93 temos um exemplo do que é realizado na instrução de operação lógica "**OU**" de dados, a Unidade de Controle através dos registradores **M3** e **M4**, repassa a ULA os dados, está realizando a operação "**OU**" e retornará o resultado através do Registrador **M2** à Unidade de Controle.

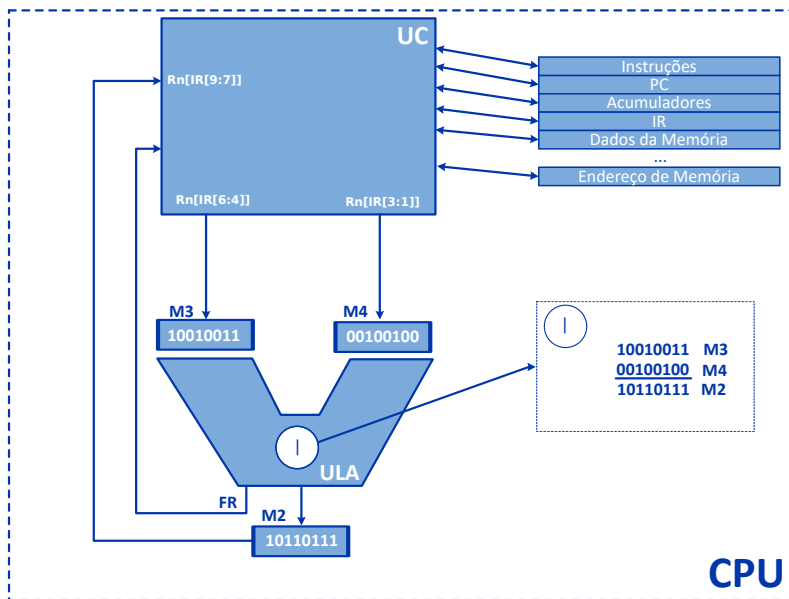


Figura 93: Exemplo: OR M3=10010011 e M4=00100100, retorna M2=10110111

Instrução XOR

A instrução XOR (16'b010100????????) realiza a operação lógica "**OU Exclusivo**", bit a bit, **XOR** está instrução pode ser considerada um caso particular da instrução **OR**, nesta instrução os receber os dados contidos em registradores, o resultado de dois bits de entrada e um de saída. A saída desta instrução será verdadeira apenas quando os bits de entrada forem diferentes, ou seja, se um deles for verdadeiro (1) e o outro falso (0), se ambos os bits de entrada possuir o mesmo valor, o bit de saída será, sempre, falso. Ao receber uma instrução de XOR a Unidade de Controle (UC) inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução XOR no módulo UC

```

1      16'b010100?????????: begin
2          //'instruction_xor;=====
3      case x(stage)
4          8'h01: begin
5              m3=Rn[IR[6:4]];
6              m4=Rn[IR[3:1]];
7              enable_alu=1'b1;
8              opcode=IR[15:10];
9          end
10         8'h06: begin
11             Rn[IR[9:7]]=m2;
12             enable_alu=1'b0;
13             processing_instruction=1'b0;
14             resetStage=1'b1;
15             bus_vga_char=m2;
16         end
17     endcase
18 end

```

Código 28: Instrução XOR na UC

A instrução XOR no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:

- O registrador de saída **M3** recebe os dados do registrador **Rn[IR[6:4]]**.
- O registrador de saída **M4** recebe os dados do registrador **Rn[IR[3:1]]**.
- O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
- O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b010100**.

2. No Segundo estágio da instrução:

- O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
- O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- A variável **processing_instruction** recebe o valor de 0,
- A variável **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução
- O registrador de saída **bus_vga_char** recebe os dados do registrador de entrada **m2**.

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução XOR no módulo ULA

```

1      6'b010100: begin
2          // 'instruction_xor;=====
3      case x(stage)
4          8'h01: begin
5              m2=m3 ^ m4;
6          end
7          8'h02: begin
8              if (m2==16'h0000) begin
9                  FR_out[12]=1'b1;
10             end
11             resetStage=1'b1;
12         end
13     endcase
14 end

```

Código 29: Instrução XOR na ULA

A instrução OR no módulo ULA possui 2 estágio:

- No Primeiro estágio, o registrador **M2** recebe os registradores **M3** ^ **M4**.
- No Segundo estágio, se **M2** for igual a **16'h0000**, então a Flag de saída **FR_out[12]** recebe o valor de 1.
- Ao termino dos estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 94 temos um exemplo do que é realizado na instrução de operação lógica "**OU Exclusivo**" de dados, a Unidade de Controle através dos registradores **M3** e **M4**, repassa a ULA os dados, está realizando a operação "**OU Exclusivo**" e retornará o resultado através do Registrador **M2** à Unidade de Controle.

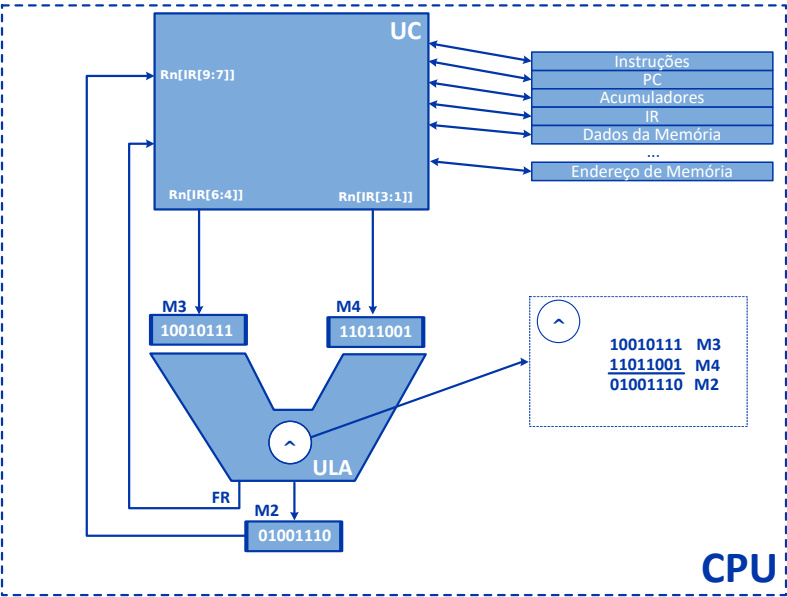


Figura 94: Exemplo: XOR M3=10010111 e M4=11011001, retorna M2=01001110

Instrução NOT

A instrução NOT (16'b010101?????????) é utilizada para realizar a complementação (inversão) dos bits de um registrador, nesta instrução os receber os dados contidos em registradores, literalmente, inverte o bit de entrada, se o bit de entrada for 0, por exemplo, o bit de saída será 0, e vice-versa. Ao receber uma instrução de NOT a Unidade de Controle (UC) inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução NOT no módulo UC

```
1 16'b010101?????????: begin
2  // 'instruction_not;=====
3  case x (stage)
```

```

4      8'h01: begin
5          m3=Rn[IR[6:4]];
6          enable_alu=1'b1;
7          opcode=IR[15:10];
8      end
9      8'h06: begin
10         Rn[IR[9:7]]=m2;
11         enable_alu=1'b0;
12         processing_instruction=1'b0;
13         resetStage=1'b1;
14     end
15     endcase
16 end

```

Código 30: Instrução NOT na UC

A instrução NOT no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:
 - (a) O registrador de saída **M3** recebe os dados do registrador **Rn[IR[6:4]]**.
 - (b) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
 - (c) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b010101**.
2. No Segundo estágio da instrução:
 - (a) O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
 - (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
 - (c) A variável **processing_instruction** recebe o valor de 0,
 - (d) A variável **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução

No primeiro estágio da instrução será habilitada a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução NOT no módulo ULA

```

1      6'b010101: begin
2          // 'instruction_not';=====
3          casex(stage)

```

```
4      8'h01: begin
5          m2=~m3;
6      end
7      8'h02: begin
8          if (m2==16'h0000) begin
9              FR_out[12]=1'b1;
10         end
11         resetStage=1'b1;
12     end
13 endcase
14 end
```

Código 31: Instrução NOT na ULA

A instrução NOT no módulo ULA possui 2 estágio:

- No Primeiro estágio, o registrador **M2** recebe o registrador **M3** (invertido).
- No Segundo estágio, se **M2** for igual a **16'h0000**, então a Flag de saída **FR_out[12]** recebe o valor de 1.
- Ao termino dos estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

Na Figura 95 temos um exemplo do que é realizado na instrução de operação lógica "**NOT**" (inversão) de dados, a Unidade de Controle através do registrador **M3**, repassa a ULA o dado, está realizando a operação "**NOT**" e retornará o resultado através do Registrador **M2** à Unidade de Controle.



Figura 95: Exemplo: NOT M3=10110101, retorna M2=01001010

Instrução SHIFTs e ROTs

A instrução SHIFTs e ROTs (16'b010000????????) são utilizadas para realizar o deslocamento de bits para esquerda/direita e rotação de bits para esquerda/direita, respectivamente, nesta instrução os receber os dados contidos em registradores, literalmente, desloca/rota o bit de entrada por **N** posições, temos um exemplo na Figura 96.

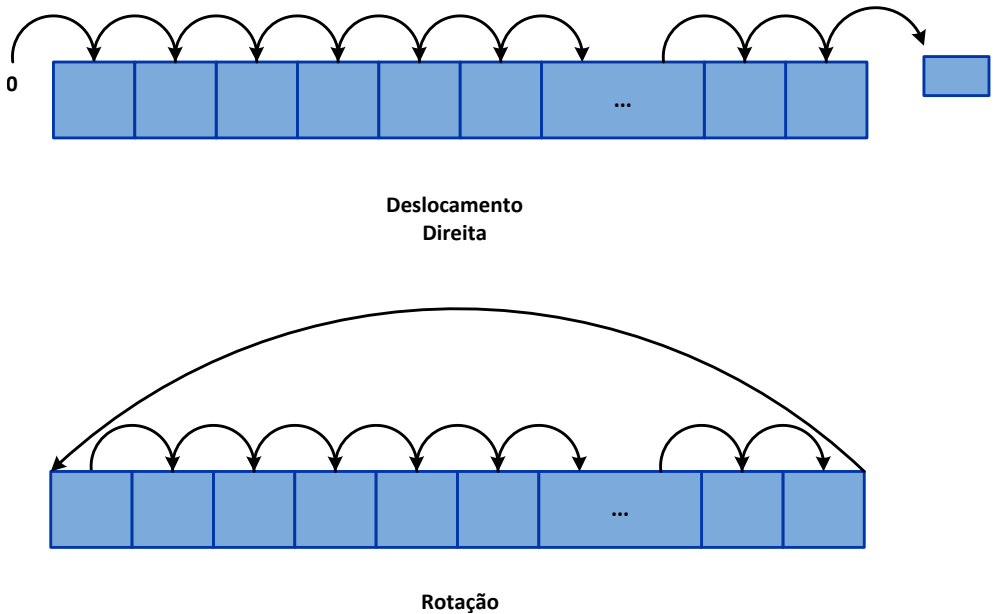


Figura 96: Exemplo: Deslocamento e Rotação de bits

Ao receber uma instrução de SHIFTs e ROTs a Unidade de Controle (UC) inicializa a ULA como veremos nos códigos declarados a seguir:

Instrução SHIFTs e ROTs no módulo UC

```

1      16'b010000?????????: begin
2      //'instructions_shifts_and_rots=====
3      casex(stage)
4      8'h01: begin
5          m3=Rn[IR[9:7]];
6          m4[3:0]=IR[3:0];
7          enable_alu=1'b1;
8          opcode=IR[15:10];

```



```

9         flagToShifthAndRot=IR[6:4];
10     end
11     8'h06: begin
12         Rn[IR[9:7]]=m2;
13         enable_alu=1'b0;
14         processing_instruction=1'b0;
15         resetStage=1'b1;
16     end
17 endcase
18 end

```

Código 32: Instrução SHIFTs e ROTs na UC

A instrução SHIFTs e ROTs no módulo UC possui 2 estágios:

1. No Primeiro estágio da instrução:

- (a) O registrador de saída **M3** recebe os dados do registrador **Rn[IR[9:7]]**.
- (b) O registrador de saída **M4[3:0]** recebe os dados do registrador **IR[3:0]**.
- (c) O barramento **enable_alu** irá receber o valor de 1, habilitando a ULA.
- (d) O registrador **opcode** irá receber o valor do registrador **IR[15:10]** que será o valor da operação a ser executada na ULA, neste caso, **6'b010000**.
- (e) A Flag de saída **flagToShifthAndRot** recebe o dados do registrador **IR[6:4]**, que irá indicar qual a operação a ser executada na ULA.

2. No Segundo estágio da instrução:

- (a) O registrador **Rn[IR[9:7]]** recebe os dados do registrador de entrada **m2**.
- (b) O barramento **enable_alu** irá receber o valor de 0, desabilitando a ULA.
- (c) A variável **processing_instruction** recebe o valor de 0 e **resetStage** recebe o valor de 1, fazendo assim o processador aguardar uma nova instrução.

No primeiro estágio da instrução será habilitação a ULA, após sua habilitação as operações lógicas serão realizadas neste módulo, retornando o resultado no segundo estágio, vejamos o código da instrução no módulo da ULA e o que ela realiza:

Instrução SHIFTs e ROTs no módulo ULA

```

1     6'b010000 : begin
2         // 'instructions_shifts_and_rots=====
3     casez(flagToShifthAndRot)
4         3'b10?: begin

```

```
5         casex(stage)
6             8'h01: begin
7                 m2=((m3<<m4)&16'hffff)|(m3>>(16'h000f-m4));
8                 resetStage=1'b1;
9             end
10        endcase
11    end
12    3'b11?: begin
13        casex(stage)
14            8'h01: begin
15                m2=((m3>>(16'h0010-m4))|(m3<<m4))&(16'hffff);
16                resetStage=1'b1;
17            end
18        endcase
19    end
20    3'b000: begin
21        casex(stage)
22            8'h01: begin
23                m2=m3<<m4;
24                resetStage=1'b1;
25            end
26        endcase
27    end
28    3'b001: begin
29        casex(stage)
30            8'h01: begin
31                m2=m3<<m4;
32                resetStage=1'b1;
33            end
34        endcase
35    end
36    3'b010: begin
37        casex(stage)
38            8'h01: begin
39                m2=m3>>m4;
40                resetStage=1'b1;
41            end
42        endcase
43    end
44    3'b011: begin
```

```
45         case x(stage)
46             8'h01: begin
47                 m2=m3>>m4;
48                 resetStage=1'b1;
49             end
50         endcase
51     end
52 endcase
53 end
```

Código 33: Instrução SHIFTs e ROTs na ULA

A instrução SHIFTs e ROTs no módulo ULA possui vários casos, variando conforme a entrada da Flag **flagToShifthAndRot**, vejamos o que é realizado em cada caso:

- Caso a Flag **flagToShifthAndRot** repassada pela UC 3'b10?, então, a instrução possui um estágio, o registrador **M2** recebe o deslocamento para esquerda («) dos registradores **M3** e **M4** e & um valor de **16'hfff**, ou (l), o deslocamento para direita (») dos registradores **M3** e **M4** - o valor de **16'hfff**.
- Terminado os estágios, a variável **resetStage** recebe o valor de 1, sinalizando o final da instrução e então retornando o processo para a UC.

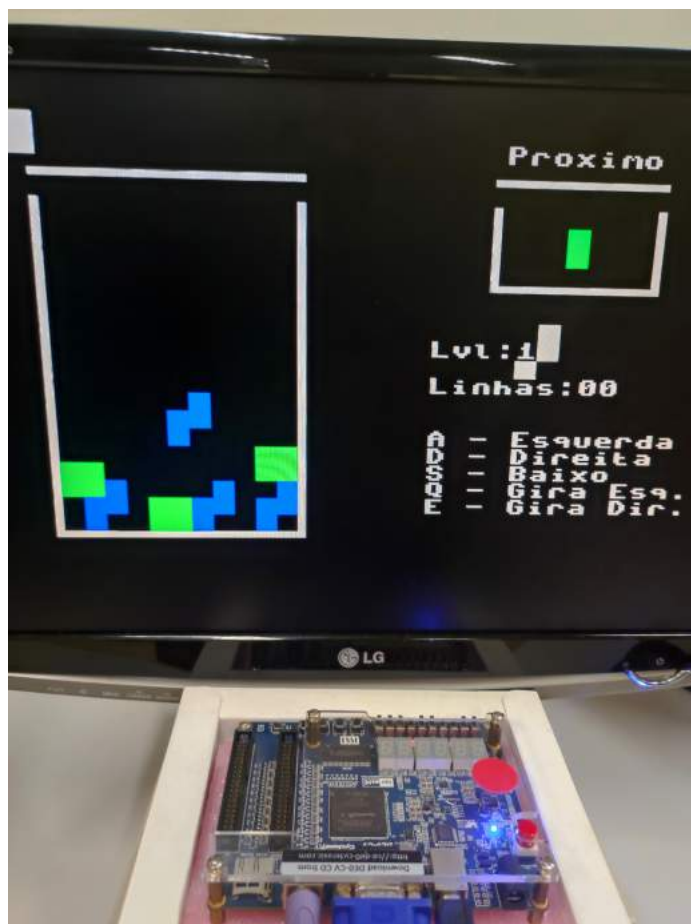


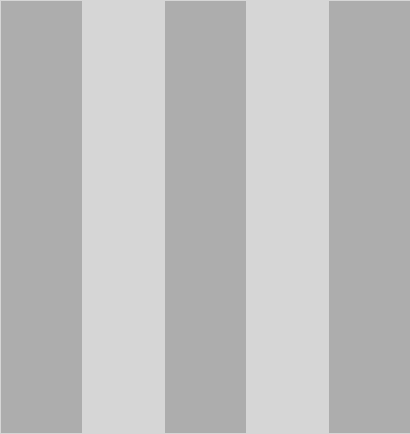
Jogo Tetris em Assembler

Foi elaborado um jogo de segunda geração conhecido como Tetris, no qual um único jogador deve empilhar blocos que caem com formato aleatório (podendo o jogador ajustar a orientação desses blocos), conforme exibido na Figura 97. Foi executado também em um kit de FPGA modelo DE0-CV da fabricante Terasic, usando uma FPGA Cyclone V, assim como o jogo de primeira geração da parte 1 desse livro.

O código fonte completo para o jogo Tetris, codificado em linguagem assembler do processador AP9, se encontra no Apêndice B desse livro.

Figura 97: Jogo Tetris codificado em linguagem assembler do processador AP9





APÊNDICE A

Jogo Pong codificado em Verilog


```

1  module pong(
2      input wire CLOCK_50,
3      output wire [3:0] VGA_R,
4      output wire [3:0] VGA_G,
5      output wire [3:0] VGA_B,
6      output wire VGA_HS,
7      output wire VGA_VS,
8      output wire vga_blank,
9      output wire vga_relogio,
10     output wire syncn,
11     input wire PS2_CLK,
12     input wire PS2_DAT,
13     output wire [6:0] HEX3,
14     output wire [6:0] HEX2,
15     output wire [6:0] HEX1,
16     output wire [6:0] HEX0,
17     input wire [1:0] SW
18 );
19
20     wire relogio50; // VGA
21     reg [7:0] r;
22     reg [7:0] g;
23     reg [7:0] b;
24     wire hsync;
25     wire vsync; // teclado
26     wire psrelogio;
27     wire psdados; // hex
28     wire resetbtn;
29     wire vs;
30     wire hs;
31     wire rst;
32     wire [10:0] x; wire [10:0] y; wire [10:0] y1; wire [10:0] y2;
33     wire hb; wire vb;
34     wire novalinha; reg hd;
35     wire novoquadro; wire fimsincv; reg vd;
36     reg relogiopar;
37     wire borda;
38     wire [10:0] bx; wire [10:0] by;
39     wire bola; wire jogador1; wire jogador2;
40     reg [3:0] rstshft;
41     wire sobel; wire descel; wire sobe2; wire desce2;
42     wire [3:0] placar1; wire [3:0] placar10; wire [3:0] placar2; wire [3:0] placar20;
43     wire plc1; wire plc2;
44     wire [7:0] thischar; wire [7:0] prevchar;
45
46     assign relogio50 = CLOCK_50;
47     // VGA
48     assign VGA_R = r[7:4];
49     assign VGA_B = b[7:4];
50     assign VGA_G = g[7:4];
51     assign VGA_HS = hsync;
52     assign VGA_VS = vsync;
53     // teclado
54     assign psrelogio = PS2_CLK;
55     assign psdados = PS2_DAT;
56     // reset
57     assign resetbtn = SW[0];
58     // gera um pulso de inicializacao sem ajuda de um sinal
59     // externo e que funciona em todas as FPGAs onde os
60     // registradores tem um valor '0' no fim da configuracao
61     always @(posedge relogio50) begin
62         rstshft <= {rstshft[2:0],1'b1};
63     end
64
65     assign rst = (~rstshft[3]) | (~resetbtn);
66     // sinais de sincronismo atrasados por um ciclo de relogio
67     always @(posedge relogio50, posedge rst) begin
68         if(rst == 1'b1) begin
69             hd <= 1'b0;
70             vd <= 1'b0;
71             relogiopar <= 1'b0;
72         end else begin
73             hd <= hs;

```

```

74         vd <= vs;
75         relogiopar <= ~relogiopar;
76     end
77 end
78
79 assign novalinha = ( ~hd) & hs;
80 // borda de subida do sincronismo horizontal
81 assign novoquadro = ( ~vd) & vs;
82 // borda de subida do sincronismo vertical
83 assign fimsincv = vd & ( ~vs);
84 // borda de descida do sincronismo vertical
85 toseg led0(
86     .digit(prevchar[3:0]),
87     .segs(hex0));
88
89 toseg led1(
90     .digit(prevchar[7:4]),
91     .segs(hex1));
92
93 toseg led2(
94     .digit(thischar[3:0]),
95     .segs(hex2));
96
97 toseg led3(
98     .digit(thischar[7:4]),
99     .segs(hex3));
100
101 entrada controles(
102     .relogio50(relogio50),
103     .inicializa(rst),
104     .ps2relogio(psrelogio),
105     .ps2dados(psdados),
106     .sobel(sobel),
107     .desce1(desce1),
108     .sobe2(sobe2),
109     .desce2(desce2),
110     .thischar(thischar),
111     .prev(prevchar));
112
113 // padrao VESA: resolucao, freq vertical, freq pixel, dados horizontais, dados
114 // verticais
115 // 640x480 60Hz, 25.175MHz, 640, 16, 96, 48, 480, 11, 2, 31
116 // 800x600 60Hz, 40.000MHz, 800, 40, 128, 88, 600, 1, 4, 23
117 // 800x600 72Hz, 50.000MHz, 800, 56, 120, 64, 600, 37, 6, 23
118 // 1024x768 60Hz, 65.000MHz, 1024, 24, 136, 160, 768, 3, 6, 29
119 temporizacao #(
120     .ATIVO(640),
121     .PRESINC(16),
122     .LARGURASINC(96),
123     .POSSINC(48))
124 horizontal(
125     .relogio(relogio50),
126     .inicializa(rst),
127     .conta(relogiopar),
128     .contagensaida(x),
129     .inativo(hb),
130     .sinc(hs));
131
132 temporizacao #(
133     .ATIVO(480),
134     .PRESINC(11),
135     .LARGURASINC(2),
136     .POSSINC(31))
137 vertical(
138     .relogio(relogio50),
139     .inicializa(rst),
140     .conta(novalinha),
141     .contagensaida(y),
142     .inativo(vb),
143     .sinc(vs));
144
145 pintor #(
146     .TAMX(640),

```

```

146 .TAMY(480))
147 leonardo(
148     .x(x),
149     .y(y),
150     .bx(bx),
151     .by(by),
152     .j1y(y1),
153     .j2y(y2),
154     .digito1(placar1),
155     .digito10(placar10),
156     .digito2(placar2),
157     .digito20(placar20),
158     .jogador1(jogador1),
159     .placar1(plc1),
160     .jogador2(jogador2),
161     .placar2(plc2),
162     .bola(bola),
163     .borda(borda));
164
165 jogo #(
166     .TAMX(640),
167     .TAMY(480))
168 gustavo(
169     .relogio(relogio50),
170     .inicializa(rst),
171     .novoquadro(novoquadro),
172     .fimsincv(fimsincv),
173     .sobel(sobel),
174     .desce1(desce1),
175     .sobe2(sobe2),
176     .desce2(desce2),
177     .bolax(bx),
178     .bolay(by),
179     .j1y(y1),
180     .j2y(y2),
181     .digito1(placar1),
182     .digito10(placar10),
183     .digito2(placar2),
184     .digito20(placar20));
185
186 // paleta de cores. Os primeiros sinais testados tem
187 // prioridade sobre os ultimos
188 always @(hb, vb, bola, jogador1, plc1, jogador2, plc2, borda) begin
189     if((hb == 1'b1 || vb == 1'b1)) begin
190         r <= 8'b00000000;
191         g <= 8'b00000000;
192         b <= 8'b00000000;
193         // forca preto em todo o retraco
194     end
195     else if((bola == 1'b1)) begin
196         r <= 8'b11111111;
197         g <= 8'b11111111;
198         b <= 8'b00000000;
199         // a bola eh amarela
200     end
201     else if((jogador1 == 1'b1 || plc1 == 1'b1)) begin
202         r <= 8'b11111111;
203         g <= 8'b01111111;
204         b <= 8'b00000000;
205         // jogador 1 eh laranja
206     end
207     else if((jogador2 == 1'b1 || plc2 == 1'b1)) begin
208         r <= 8'b00000000;
209         g <= 8'b01111111;
210         b <= 8'b11111111;
211         // jogador 2 eh azulado
212     end
213     else if((borda == 1'b1)) begin
214         r <= 8'b11111111;
215         g <= 8'b11000000;
216         b <= 8'b11111111;
217         // borda eh branca
218     end

```

```

219     else begin
220         r <= 8'b00000000;
221         g <= 8'b10000000;
222         b <= 8'b00000000;
223         // fundo eh verde escuro
224     end
225 end
226
227 // note que os placares tem a mesma cor que os jogadores, mas eh
228 // muito facil modificar o codigo acima para serem cores diferentes
229 // padrao VESA para a polaridade do sincronismo:
230 // 640x480 = -h -v
231 // 800x600 = +h +v
232 assign vsync = ~vs;
233 assign hsync = ~hs;
234 assign syncn = 1'b0;
235 // DE2 115 nao usa sync no verde
236 assign vga_relogio = relógio50;
237 assign vga_blank = 1'b1;
238 // o codigo acima garante o blanking
239
240 endmodule
241
242
243
244 module jogo(
245     input wire relógio,
246     input wire inicializa,
247     input wire novoquadro,
248     input wire fimsincv,
249     input wire sobel,
250     input wire descsl,
251     input wire sobe2,
252     input wire desce2,
253     output wire [10:0] bolax,
254     output wire [10:0] bolay,
255     output wire [10:0] j1y,
256     output wire [10:0] j2y,
257     output wire [3:0] digito1,
258     output wire [3:0] digito10,
259     output wire [3:0] digito2,
260     output wire [3:0] digito20
261 );
262
263 parameter [31:0] TAMX=800;
264 parameter [31:0] TAMY=600;
265
266
267
268 reg [10:0] bdx; reg [10:0] bdy;
269 reg [10:0] bx; reg [10:0] by;
270 wire [10:0] x; wire [10:0] y; reg [10:0] y1; reg [10:0] y2;
271 reg [3:0] placar1; reg [3:0] placar10; reg [3:0] placar2; reg [3:0] placar20;
272 parameter METADEX = TAMX / 2;
273 parameter QUARTOX = METADEX / 2;
274 parameter TRESQUARTOSX = METADEX + QUARTOX;
275 parameter METADEY = TAMY / 2;
276 parameter LARGURABOLA = TAMX / 60;
277 parameter ALTURABOLA = TAMY / 48;
278 parameter MARGEMBY = TAMY / 96;
279 parameter LARGURAJOGADOR = TAMX / 45;
280 parameter MEIAALTURAJOGADOR = TAMY / 12;
281 parameter POSJ1X = TAMX / 20;
282 parameter POSJ2X = TAMX - POSJ1X;
283 parameter MARGEMJY = TAMY / 10;
284
285 assign digito1 = placar1;
286 assign digito10 = placar10;
287 assign digito2 = placar2;
288 assign digito20 = placar20;
289 assign bolax = bx;
290 assign bolay = by;
291 assign j1y = y1;

```

```

292 assign j2y = y2;
293 always @(posedge relógio, posedge inicializa, posedge novoquadro, posedge fimsincv)
begin
294     if(inicializa == 1'b1) begin
295         bx <= METADEX;
296         by <= METADEY;
297         bdx <= 1;
298         bdy <= 1;
299         y1 <= 11;
300         y2 <= METADEY;
301         placar1 <= 4'b0000;
302         placar10 <= 4'b0000;
303         placar2 <= 4'b0000;
304         placar20 <= 4'b0000;
305     end else begin
306         if(novoquadro == 1'b1) begin
307             // 60 vezes por segundo (na verdade frequencia vertical)
308             if((bx == POSJ1X && (by + (ALTURABOLA + MEIAALTURAJOGADOR)) > y1 && by < (y1 +
MEIAALTURAJOGADOR))) begin
309                 // jogador 1 rebateu
310                 bdx <= (bdx ^ 11'b1111111111) + 1;
311                 // bdx = 0 - bdx
312                 if((sobel == 1'b1)) begin
313                     bdy <= bdy - 1;
314                 end
315                 if((desce1 == 1'b1)) begin
316                     bdy <= bdy + 1;
317                 end
318             end
319             if((bx == (POSJ2X - LARGURABOLA) && (by + (ALTURABOLA + MEIAALTURAJOGADOR)) >
y2 && by < (y2 + MEIAALTURAJOGADOR))) begin
320                 // jogador 2 rebateu
321                 bdx <= (bdx ^ 11'b1111111111) + 1;
322                 // bdx = 0 - bdx
323                 if((sobel2 == 1'b1)) begin
324                     bdy <= bdy - 1;
325                 end
326                 if((desce2 == 1'b1)) begin
327                     bdy <= bdy + 1;
328                 end
329             end
330             if((bx < 1)) begin
331                 // jogador 1 deixou passar
332                 bx <= POSJ2X - LARGURABOLA - 1;
333                 by <= y2;
334                 bdx <= 11'b1111111111;
335                 if((desce2 == 1'b1)) begin
336                     bdy <= 11'b00000000001;
337                 end
338                 else if((sobel2 == 1'b1)) begin
339                     bdy <= 11'b1111111111;
340                 end
341                 else begin
342                     bdy <= 11'b00000000000;
343                 end
344                 if((placar2 == 4'b1001)) begin
345                     // se digito ja eh 9 entao vai um
346                     placar20 <= placar20 + 1;
347                     placar2 <= 4'b0000;
348                 end
349                 else begin
350                     placar2 <= placar2 + 1;
351                 end
352             end
353             if((bx > (TAMX - LARGURABOLA))) begin
354                 // jogador 2 deixou passar
355                 bx <= POSJ1X + 1;
356                 by <= y1;
357                 bdx <= 11'b00000000001;
358                 if((desce1 == 1'b1)) begin
359                     bdy <= 11'b00000000001;
360                 end
361                 else if((sobel == 1'b1)) begin

```

```

362         bdy <= 11'b111111111111;
363     end
364     else begin
365         bdy <= 11'b000000000000;
366     end
367     if((placar1 == 4'b1001)) begin
368         // se digito ja eh 9 entao vai um
369         placar10 <= placar10 + 1;
370         placar1 <= 4'b0000;
371     end
372     else begin
373         placar1 <= placar1 + 1;
374     end
375 end
376 if((by < MARGEMBY || by > (TAMY - MARGEMBY - ALTURABOLA))) begin
377     // bordas horizontais sempre refletem verticalmente
378     bdy <= (bdy ^ 11'b111111111111) + 1;
379     // bdy = 0 - bdy
380 end
381 if((sobel == 1'b1 && y1 > MARGEMJY)) begin
382     // ainda tem espaco para subir?
383     y1 <= y1 - 2;
384 end
385 if((desce1 == 1'b1 && y1 < (TAMY - MARGEMJY))) begin
386     // ainda tem espaco para descer?
387     y1 <= y1 + 2;
388 end
389 if((sobe2 == 1'b1 && y2 > MARGEMJY)) begin
390     // ainda tem espaco para subir?
391     y2 <= y2 - 2;
392 end
393 if((desce2 == 1'b1 && y2 < (TAMY - MARGEMJY))) begin
394     // ainda tem espaco para descer?
395     y2 <= y2 + 2;
396 end
397 end
398 if(fimsincv == 1'b1) begin
399     // no fim do sincronismo movimenta a bola com a velocidade calculada acima
400     bx <= bx + bdx;
401     by <= by + bdy;
402 end
403 end
404 end
405
406 endmodule
407
408
409
410
411 module pintor(
412     input wire [10:0] x,
413     input wire [10:0] y,
414     input wire [10:0] bx,
415     input wire [10:0] by,
416     input wire [10:0] j1y,
417     input wire [10:0] j2y,
418     input wire [3:0] digito1,
419     input wire [3:0] digito10,
420     input wire [3:0] digito2,
421     input wire [3:0] digito20,
422     output wire jogador1,
423     output wire placar1,
424     output wire jogador2,
425     output wire placar2,
426     output wire bola,
427     output wire borda
428 );
429
430 parameter [31:0] TAMX=800;
431 parameter [31:0] TAMY=600;
432
433
434

```

```

435 wire d1; wire d10; wire d2; wire d20;
436 parameter METADEX = TAMX / 2;
437 parameter QUARTOX = METADEX / 2;
438 parameter TRESQUARTOSX = METADEX + QUARTOX;
439 parameter METADEY = TAMY / 2;
440 parameter LARGURABOLA = TAMX / 60;
441 parameter ALTURABOLA = TAMY / 48;
442 parameter MARGEMBY = TAMY / 96;
443 parameter LARGURAJOGADOR = TAMX / 45;
444 parameter MEIAALTURAJOGADOR = TAMY / 12;
445 parameter POSJ1X = TAMX / 20;
446 parameter POSJ2X = TAMX - POSJ1X;
447 parameter MARGEMJY = TAMY / 10;
448 parameter MEIAFAIXAVERT = TAMX / 180;
449
450 digito #(
451     .POSICAOX(QUARTOX - 10))
452 plc10(
453     .x(x),
454     .y(y),
455     .placar(digito10),
456     .caracter(d10));
457
458 digito #(
459     .POSICAOX(QUARTOX + 10))
460 plc1(
461     .x(x),
462     .y(y),
463     .placar(digito1),
464     .caracter(d1));
465
466 digito #(
467     .POSICAOX(TRESQUARTOSX - 10))
468 plc20(
469     .x(x),
470     .y(y),
471     .placar(digito20),
472     .caracter(d20));
473
474 digito #(
475     .POSICAOX(TRESQUARTOSX + 10))
476 plc2(
477     .x(x),
478     .y(y),
479     .placar(digito2),
480     .caracter(d2));
481
482 assign bola = (x > bx && x < (bx + LARGURABOLA) && y > by && y < (by + ALTURABOLA))
483 ? 1'b1 : 1'b0;
484 assign jogador1 = (x > (POSJ1X - LARGURAJOGADOR) && x < POSJ1X && y > (j1y -
485     MEIAALTURAJOGADOR) && y < (j1y + MEIAALTURAJOGADOR)) ? 1'b1 : 1'b0;
486 assign jogador2 = (x > POSJ2X && x < (POSJ2X + LARGURAJOGADOR) && y > (j2y -
487     MEIAALTURAJOGADOR) && y < (j2y + MEIAALTURAJOGADOR)) ? 1'b1 : 1'b0;
488 assign placar1 = (d1 == 1'b1 || d10 == 1'b1) ? 1'b1 : 1'b0;
489 assign placar2 = (d2 == 1'b1 || d20 == 1'b1) ? 1'b1 : 1'b0;
490 assign borda = (x > (METADEX - MEIAFAIXAVERT) && x < (METADEX + MEIAFAIXAVERT)) ?
491     1'b1 : (y < MARGEMBY || y > (TAMY - MARGEMBY)) ? 1'b1 : 1'b0;
492
493 endmodule
494
495 //
496 // Este bloco simula dois controles simples (com apenas sinais "sobe" e
497 // "desce"). Neste caso os sinais estao vindo de um teclado padrao PS/2
498 // mas seria possivel usar diretamente botoes da placa de FPGA
499 //
500 // A interface PS/2 de teclado eh bidirecional, mas se abrimos mao de
501 // podermos reinicializar e reconfigurar o teclado e nao tivermos
502 // interesse em controlado os LEDs podemos fingir que eh uma interface
503 // apenas de leitura com estas formas de onda:
504 // ps2relogio:-----
505 // ps2dados:  xxxxx-----x=====D0====x=====D1====x=====D2====x=====
506 //
507 // ps2relogio:  _____

```

```

504 // ps2dados: D6===x===D7===x===P===x-----
505 //
506 // Sao sempre 11 bits onde o primeiro eh sempre zero e ultimo
507 // eh sempre um, Os bits D0 a D7 sao o dado que queremos ler e
508 // o bit P eh a paridade de D0 a D7.
509 // no timescale needed
510
511 module entrada(
512 input wire relógio50,
513 input wire inicializa,
514 input wire ps2relógio,
515 input wire ps2dados,
516 output reg sobel,
517 output reg descel,
518 output reg sobe2,
519 output reg desce2,
520 output wire [7:0] thischar,
521 output wire [7:0] prev
522 );
523
524 reg [10:0] psshft;
525 reg [7:0] pschar; reg [7:0] prevchar;
526 reg [14:0] pscnt;
527 reg ps2r1; reg ps2r2; reg amostra;
528 reg ps2d1; reg ps2d2;
529
530 // atraso com dois flip-flops para reduzir meta-estabilidade
531 always @(posedge relógio50) begin
532 ps2r1 <= ps2relógio;
533 ps2r2 <= ps2r1;
534 amostra <= ps2r2 & ( ~(ps2r1));
535 ps2d1 <= ps2dados;
536 ps2d2 <= ps2d1;
537 end
538
539 always @(posedge relógio50, posedge amostra) begin
540 if(amostra == 1'b1) begin
541 psshft <= {ps2d2,psshft[10:1]};
542 pscnt <= 15'b000000000000000;
543 end
544 else begin
545 pscnt <= pscnt + 1;
546 if(psshft[0] == 1'b0) begin
547 // chegou o "start bit" la em baixo!! Entao chegaram todos
548 prevchar <= pschar;
549 pschar <= psshft[8:1];
550 // nao pega inicio, fim e nem paridade
551 psshft <= 11'b1111111111;
552 end
553 if(pscnt == 15'b11111111111100) begin
554 psshft <= 11'b1111111111;
555 // forca reinicializacao depois de 0x3ffc * 20ns = 328us sem descida em
ps2relógio
556 end
557 if(pscnt == 15'b11111111111111) begin
558 pscnt <= 15'b11111111111101;
559 // trava a contagem para nao voltar a zero sozinha
560 end
561 end
562 end
563
564 always @(inicializa, pschar, prevchar) begin
565 if(inicializa == 1'b1) begin
566 // "a"
567 sobel <= 1'b0;
568 descel <= 1'b0;
569 sobe2 <= 1'b0;
570 desce2 <= 1'b0;
571 end
572 else begin
573 if(pschar == 8'h1C) begin
574 // "a"
575 if(prevchar == 8'hF0) begin

```



```

576         // keyup
577         sobel1 <= 1'b0;
578     end
579     else begin
580         sobel1 <= 1'b1;
581         descel1 <= 1'b0;
582     end
583 end
584 if(pschar == 8'h1A) begin
585     // "z"
586     if(prevchar == 8'hF0) begin
587         // keyup
588         descel1 <= 1'b0;
589     end
590     else begin
591         descel1 <= 1'b1;
592         sobel1 <= 1'b0;
593     end
594 end
595 if(pschar == 8'h42) begin
596     // "k"
597     if(prevchar == 8'hF0) begin
598         // keyup
599         sobe2 <= 1'b0;
600     end
601     else begin
602         sobe2 <= 1'b1;
603         desce2 <= 1'b0;
604     end
605 end
606 if(pschar == 8'h3A) begin
607     // "m"
608     if(prevchar == 8'hF0) begin
609         // keyup
610         desce2 <= 1'b0;
611     end
612     else begin
613         desce2 <= 1'b1;
614         sobe2 <= 1'b0;
615     end
616 end
617 end
618 end
619
620 assign thischar = pschar;
621 assign prev = prevchar;
622
623 endmodule
624
625 //
626 // Este bloco gera uma contagem e dois sinais auxiliares que dividem
627 // a contagem em 4 regioes. O que esta sendo contado eh controlado por
628 // um sinal "conta" que pode ficar sempre em 1 se for desejado contar
629 // diretamente ciclos de relógio.
630 //
631 // inativo: _____
632 // sinc: _____
633 //      4 | 1 | 2 | 3 | 4 | 1
634 // 1) ATIVO
635 // 2) PRESINC
636 // 3) LARGURASINC
637 // 4) POSSINC
638 //
639 // A contagem comeca em zero no inicio da regio ativa e deve ser menor
640 // que o valor maximo do contador (2047 com 11 bits) ateh o fim da regio 4
641 //
642 // no timescale needed
643
644 module temporizacao(
645     input wire relógio,
646     input wire conta,
647     input wire inicializa,
648     output wire [10:0] contagem_saida,

```

```

649 output wire inativo,
650 output wire sinc
651 );
652
653 parameter [31:0] ATIVO=360;
654 parameter [31:0] PRESINC=13;
655 parameter [31:0] LARGURASINC=54;
656 parameter [31:0] POSSINC=27;
657
658
659 parameter TOTAL = ATIVO + PRESINC + LARGURASINC + POSSINC;
660 reg [10:0] contagem;
661
662 assign contagensaida = contagem;
663 assign inativo = (contagem < ATIVO) ? 1'b0 : 1'b1;
664 assign sinc = (contagem < (ATIVO + PRESINC)) ? 1'b0 : (contagem < (ATIVO + PRESINC +
    LARGURASINC)) ? 1'b1 : 1'b0;
665 always @(posedge relógio, posedge inicializa) begin
666     if(inicializa == 1'b1) begin
667         contagem <= 11'b000000000000;
668     end else begin
669         if(conta == 1'b1) begin
670             if(contagem < (TOTAL - 1)) begin
671                 contagem <= contagem + 1;
672             end
673             else begin
674                 contagem <= 11'b000000000000;
675             end
676         end
677     end
678 end
679
680 endmodule
681
682
683 //
684 // converte um numero de 4 bits em informacoes para LED de
685 // 7 segmentos e depois converte esta informacao em 7 retangulos
686 // na tela, ascetos ou apagados conforme a entrada
687 //
688 // no timescale needed
689
690 module digito(
691 input wire [10:0] x,
692 input wire [10:0] y,
693 input wire [3:0] placar,
694 output wire caracter
695 );
696
697 parameter [31:0] POSICAOX=0;
698 parameter X1 = POSICAOX;
699 parameter X2 = POSICAOX + 5;
700 parameter X3 = POSICAOX + 10;
701 parameter X4 = POSICAOX + 15;
702 parameter Y1 = 30;
703 parameter Y2 = 35;
704 parameter Y3 = 40;
705 parameter Y4 = 45;
706 parameter Y5 = 50;
707 parameter Y6 = 55;
708 reg [6:0] segment;
709
710 // segment encoding
711 //      0
712 //      ---
713 //  5 |   | 1
714 //  ---  <- 6
715 //  4 |   | 2
716 //  ---
717 //      3
718 always @(*) begin
719     case(placar)
720         4'b0001 : segment <= 7'b1111001;

```

```

721 //1
722 4'b0010 : segment <= 7'b0100100;
723 //2
724 4'b0011 : segment <= 7'b0110000;
725 //3
726 4'b0100 : segment <= 7'b0011001;
727 //4
728 4'b0101 : segment <= 7'b0010010;
729 //5
730 4'b0110 : segment <= 7'b0000010;
731 //6
732 4'b0111 : segment <= 7'b1111000;
733 //7
734 4'b1000 : segment <= 7'b0000000;
735 //8
736 4'b1001 : segment <= 7'b0010000;
737 //9
738 4'b1010 : segment <= 7'b0001000;
739 //A
740 4'b1011 : segment <= 7'b0000011;
741 //b
742 4'b1100 : segment <= 7'b1000110;
743 //c
744 4'b1101 : segment <= 7'b0100001;
745 //d
746 4'b1110 : segment <= 7'b0000110;
747 //E
748 4'b1111 : segment <= 7'b0001110;
749 //F
750 default : segment <= 7'b1000000;
751 endcase
752 end
753
754 //0
755 assign character = (segment[0] == 1'b0 && x > X1 && x < X4 && y > Y1 && y < Y2) ?
1'b1 : (segment[1] == 1'b0 && x > X3 && x < X4 && y > Y1 && y < Y4) ? 1'b1 : (
segment[2] == 1'b0 && x > X3 && x < X4 && y > Y3 && y < Y6) ? 1'b1 : (segment[3] ==
1'b0 && x > X1 && x < X4 && y > Y5 && y < Y6) ? 1'b1 : (segment[4] == 1'b0 && x > X1
&& x < X2 && y > Y3 && y < Y6) ? 1'b1 : (segment[5] == 1'b0 && x > X1 && x < X2 &&
y > Y1 && y < Y4) ? 1'b1 : (segment[6] == 1'b0 && x > X1 && x < X4 && y > Y3 && y <
Y4) ? 1'b1 : 1'b0;

756
757 endmodule
758
759
760 module toseg(
761 input wire [3:0] digit,
762 output reg [6:0] segs
763 );
764 // segment encoding
765 //      0
766 //      ---
767 //  5 |   | 1
768 //  --- <- 6
769 //  4 |   | 2
770 //  ---
771 //      3
772 always @(*) begin
773     case(digit)
774         4'b0001 : segs <= 7'b1111001;
775     //1
776         4'b0010 : segs <= 7'b0100100;
777     //2
778         4'b0011 : segs <= 7'b0110000;
779     //3
780         4'b0100 : segs <= 7'b0011001;
781     //4
782         4'b0101 : segs <= 7'b0010010;
783     //5
784         4'b0110 : segs <= 7'b0000010;
785     //6
786         4'b0111 : segs <= 7'b1111000;
787     //7

```

```
788         4'b1000 : segs <= 7'b0000000;
789     //8
790         4'b1001 : segs <= 7'b0010000;
791     //9
792         4'b1010 : segs <= 7'b0001000;
793     //A
794         4'b1011 : segs <= 7'b0000011;
795     //b
796         4'b1100 : segs <= 7'b1000110;
797     //c
798         4'b1101 : segs <= 7'b0100001;
799     //d
800         4'b1110 : segs <= 7'b0000110;
801     //E
802         4'b1111 : segs <= 7'b0001110;
803     //F
804         default : segs <= 7'b1000000;
805     endcase
806 end
807
808 //0
809
810 endmodule
811
```

IV

APÊNDICE B

Tetris

codificado

em Assembler


```

1 ; Jogo tetris elaborado em linguagem assembler do processador AP9 de 32 bits
2
3 jmp32 main
4 ; ----- TABELA DE CORES -----
5 ; adicione ao caracter para Selecionar a cor correspondente
6
7 ; 0 branco                0000 0000
8 ; 256 marrom              0001 0000
9 ; 512 verde               0010 0000
10 ; 768 oliva              0011 0000
11 ; 1024 azul marinho      0100 0000
12 ; 1280 roxo              0101 0000
13 ; 1536 teal              0110 0000
14 ; 1792 prata             0111 0000
15 ; 2048 cinza             1000 0000
16 ; 2304 vermelho          1001 0000
17 ; 2560 lima              1010 0000
18 ; 2816 amarelo           1011 0000
19 ; 3072 azul              1100 0000
20 ; 3328 rosa              1101 0000
21 ; 3584 aqua              1110 0000
22 ; 3840 branco            1111 0000
23
24 ; Lista de numeros aleatorios.
25 rand : var #5
26 static rand + #0, #2
27 static rand + #1, #5
28 static rand + #2, #1
29 static rand + #3, #4
30 static rand + #4, #3
31
32 ; Lista auxiliar de numeros aleatorios.
33 rand_ : var #5
34 static rand_ + #0, #5
35 static rand_ + #1, #1
36 static rand_ + #2, #4
37 static rand_ + #3, #3
38 static rand_ + #4, #2
39
40 ; Tela do jogo.
41 ;----- Desenho da tela -----
42 S1L1 : string " "
43 S1L2 : string " "
44 S1L3 : string " "
45 S1L4 : string " Proximo "
46 S1L5 : string " ! / !((((("/ "
47 S1L6 : string " $ & $ & "
48 S1L7 : string " $ & $ & "
49 S1L8 : string " $ & $ & "
50 S1L9 : string " $ & $ & "
51 S1L10 : string " $ & $ & "
52 S1L11 : string " $ & {})))))# "
53 S1L12 : string " $ & "
54 S1L13 : string " $ & "
55 S1L14 : string " $ & Lvl: "
56 S1L15 : string " $ & "
57 S1L16 : string " $ & Linhas: "
58 S1L17 : string " $ & "
59 S1L18 : string " $ & "
60 S1L19 : string " $ & A - Esquerda "
61 S1L20 : string " $ & D - Direita "
62 S1L21 : string " $ & S - Baixo "
63 S1L22 : string " $ & Q - Gira Esq. "
64 S1L23 : string " $ & E - Gira Dir. "
65 S1L24 : string " $ & "
66 S1L25 : string " {})))))))))# "
67 S1L26 : string " "
68 S1L27 : string " "
69 S1L28 : string " "
70 S1L29 : string " "
71 S1L30 : string " "
72
73 ; Tela de end game.

```

```

74 S2L1 : string "
75 S2L2 : string "
76 S2L3 : string "
77 S2L4 : string "
78 S2L5 : string "
79 S2L6 : string "
80 S2L7 : string "
81 S2L8 : string "
82 S2L9 : string "
83 S2L10 : string "
84 S2L11 : string "
85 S2L12 : string "
86 S2L13 : string "
87 S2L14 : string "
88 S2L15 : string "
89 S2L16 : string "
90 S2L17 : string "
91 S2L18 : string "
92 S2L19 : string "
93 S2L20 : string "
94 S2L21 : string "
95 S2L22 : string "
96 S2L23 : string "
97 S2L24 : string "
98 S2L25 : string "
99 S2L26 : string "
100 S2L27 : string "
101 S2L28 : string "
102 S2L29 : string "
103 S2L30 : string "
104
105 ; Tela de novo jogo.
106 S3L1 : string "
107 S3L2 : string "
108 S3L3 : string "
109 S3L4 : string "
110 S3L5 : string "
111 S3L6 : string "
112 S3L7 : string "
113 S3L8 : string "
114 S3L9 : string "
115 S3L10 : string "
116 S3L11 : string "
117 S3L12 : string "
118 S3L13 : string "
119 S3L14 : string "
120 S3L15 : string "
121 S3L16 : string "
122 S3L17 : string "
123 S3L18 : string "
124 S3L19 : string "
125 S3L20 : string "
126 S3L21 : string "
127 S3L22 : string "
128 S3L23 : string "
129 S3L24 : string "
130 S3L25 : string "
131 S3L26 : string "
132 S3L27 : string "
133 S3L28 : string "
134 S3L29 : string "
135 S3L30 : string "
136
137 ; Matriz de memorização de cores
138 mapa_corL1 : string "
139 mapa_corL2 : string "
140 mapa_corL3 : string "
141 mapa_corL4 : string "
142 mapa_corL5 : string "
143 mapa_corL6 : string "
144 mapa_corL7 : string "
145 mapa_corL8 : string "
146 mapa_corL9 : string "

```

GAME OVER
 ENTER PARA JOGAR NOVAMENTE

TETRIS

PRESSIONE ENTER PARA INICIAR


```

147 mapa_corL10 : string "
148 mapa_corL11 : string "
149 mapa_corL12 : string "
150 mapa_corL13 : string "
151 mapa_corL14 : string "
152 mapa_corL15 : string "
153 mapa_corL16 : string "
154 mapa_corL17 : string "
155 mapa_corL18 : string "
156 mapa_corL19 : string "
157 mapa_corL20 : string "
158 mapa_corL21 : string "
159 mapa_corL22 : string "
160 mapa_corL23 : string "
161 mapa_corL24 : string "
162 mapa_corL25 : string "
163 mapa_corL26 : string "
164 mapa_corL27 : string "
165 mapa_corL28 : string "
166 mapa_corL29 : string "
167 mapa_corL30 : string "
168
169 ; dec32laração de variáveis
170 pos1      : var #1 ;posição atual na tela
171 pos2      : var #1 ;posição atual na tela
172 pos3      : var #1 ;posição atual na tela
173 pos4      : var #1 ;posição atual na tela
174 bloco     : var #1 ;caracter de cada pixel do bloco
175 tipo_bloco : var #1 ;Tipo do bloco (5 opções)
176 posi      : var #1 ;Posição do bloco (4 opções cada)
177 seed      : var #1 ;semente para bloco randomico
178 seed      : var #1 ;semente para posição randomica
179 seed_next  : var #1 ;semente para bloco randomico futuro
180 seed_next1 : var #1 ;semente para posição randomica futura
181
182 cor_atual  : var #1 ;cor de cada bloco
183 dir        : var#1; direção atual
184 last_dir   : var#1; direção anterior
185 last_posi  : var#1; posição anterior
186 last_pos1  : var#1; posição anterior
187 last_pos2  : var#1; posição anterior
188 last_pos3  : var#1; posição anterior
189 last_pos4  : var#1; posição anterior
190 flag_parada : var#1; Flag para averiguação de parada do bloco
191 flag_parou  : var#1; Flag para quando um bloco parar
192 contador : var#1; Contador para temporização da queda automatica do bloco
193 apaga_linha : var#20; vetor de indicação de linhas a apagar
194 ini_linha  : var#1; início da verificação de linhas
195 ini_colun  : var#1; início da verificação de colunas
196 num_linha  : var#1; número de linhas para os blocos
197 num_colun  : var#1; número de colunas para os blocos
198 posi_ponteiro : var#1; posiciona o ponteiro inicial de verificação
199 flag_deleta : var#1; flag que sinaliza para deletar linhas
200 num_cor    : var#1; valor de 1 a 5 que mapeia a cor atual do pixel
201 cor_inst    : var#1; código da cor mapeada
202 flag_impr1  : var#1; flag de impressão inicial
203 flag_impr2  : var#1; flag de impressão inicial
204 flag_impr3  : var#1; flag de impressão inicial
205 flag_impr4  : var#1; flag de impressão inicial
206 score_unid  : var#1;score de pontuação de unidade
207 score_dez   : var#1;score de pontuação de dezenas
208 aux_lín     : var#1; Variável auxiliar para deletar mais de uma linha ao mesmo tempo
209 bloco_next : var#1; variável com o próximo bloco
210 level      : var#1; variável de controle de level
211 constA     : var#1; parâmetro para ajuste de delay
212 constB     : var#1; parâmetro para ajuste de delay
213 ;dec32laração mapa de strings
214
215 numero0 : string "0"
216 numero1 : string "1"
217 numero2 : string "2"
218 numero3 : string "3"
219 numero4 : string "4"

```

```

220 numero5 : string "5"
221 numero6 : string "6"
222 numero7 : string "7"
223 numero8 : string "8"
224 numero9 : string "9"
225
226 main:
227 ;----Inicialização das variáveis----
228 loadn32 r2, #'%' ;caracter do bloco
229 store32 bloco, r2
230 loadn32 r0, #24
231 store32 ini_linha, r0
232 loadn32 r0, #11
233 store32 num_colun, r0
234 loadn32 r0, #7
235 store32 ini_colun, r0
236 loadn32 r0, #20
237 store32 num_linha, r0
238 loadn32 r0, #1
239 store32 level, r0
240 loadn32 r0, #0
241 store32 seed, r0
242 store32 seed, r0
243 loadn32 r0, #0
244 store32 dir, r0
245 store32 last_dir, r0
246 store32 last_pos1, r0
247 store32 last_pos2, r0
248 store32 last_pos3, r0
249 store32 last_pos4, r0
250 store32 last_posi, r0
251 store32 flag_parou, r0
252 store32 flag_deleta, r0
253 store32 flag_impr1, r0
254 store32 flag_impr2, r0
255 store32 flag_impr3, r0
256 store32 flag_impr4, r0
257 store32 aux_lin, r0
258 store32 seed_next, r0
259 store32 seed_next1, r0
260 store32 score_unid, r0
261 store32 score_dez, r0
262
263 jmp32 start_menu ; chama a tela inicial
264
265
266 BACK_MAIN:
267     call32 ClearScreen
268     call32 limpa_memorias
269     loadn32 r1, #S1L1 ; tela inicial
270     loadn32 r2, #0 ; cor da tela
271     call32 PrintScreen
272     call32 print_borda
273
274 loop:
275     loadn32 r6, #10
276     store32 contador, r6
277
278     call32 ajusta_delay
279
280     call32 print_nivel
281
282     call32 print_score; imprime a pontuação de linhas
283
284 ;-----Rotina que prevê o próximo bloco-----
285
286     call32 getRandNumNEXT;Gera aleatoriamente o número do próximo bloco
287     call32 getRandNum1NEXT ;Gera aleatoriamente a posição do próximo bloco
288
289 ;----Inicialização da posição de impressão do próximo
290     loadn32 r0, #309
291     store32 pos1, r0
292

```

```

293 call32 select ;seleciona bloco aleatório e posição inicial
294 call32 clear_next; limpa a tela do próximo bloco
295 call32 print_next; imprime o próximo bloco
296
297 ;-----Rotina gera o bloco atual -----
298
299 call32 getRandNum ;Gera aleatoriamente o número do próximo bloco
300 call32 getRandNum1 ;Gera aleatoriamente a posição do próximo bloco
301
302 ;----Inicialização da próxima posição
303 loadn32 r0, #90
304 store32 pos1, r0
305
306 call32 select ;seleciona bloco aleatório e posição inicial
307
308 loop1:
309
310 call32 move ; rotina de mov32imento e rotação do bloco
311
312 call32 Delay;
313
314 load32 r6, contador
315 dec32 r6
316 store32 contador, r6
317 loadn32 r7, #0
318 cmp32 r6, r7
319 ceq32 queda
320
321 loadn32 r0, #1
322 load32 r1, flag_parou
323 cmp32 r1, r0
324 jeq32 parou
325
326 jmp32 loop1
327 parou:
328 loadn32 r0, #0
329 store32 flag_parou, r0
330 call32 grava_bloco ; grava o bloco que parou no mapa
331 call32 pontos_linha; verifica alinhamento de pontos
332
333 load32 r0, flag_deleta
334 loadn32 r1, #0
335 cmp32 r0, r1
336 jeq32 pula_apagar
337
338 call32 apagar; apaga as linhas verificadas e desloca as linhas de cima
339
340 pula_apagar:
341
342 call32 verifi_gameOver
343 jmp32 loop
344
345 halt32
346
347 ;---- Fim do Programa Princ32ipal -----
348
349 move:
350 push32 r0
351 push32 r1
352 push32 r2
353 push32 r3
354 push32 r4
355 push32 r5
356
357 loadn32 r0, #0
358 store32 flag_parada, r0
359
360 load32 r2, dir
361 store32 last_dir, r2
362
363 load32 r2, pos1
364 store32 last_pos1, r2
365

```

```

366     load32 r3, pos2
367     store32 last_pos2, r3
368
369     load32 r4, pos3
370     store32 last_pos3, r4
371
372     load32 r5, pos4
373     store32 last_pos4, r5
374
375     call32 verifi_apagaIni
376
377     load32 r0, flag_impr1
378     loadn32 r1, #1
379     cmp32 r0,r1
380     ceq32 apaga_pos1; apaga pixel pos1
381     load32 r0, flag_impr2
382     cmp32 r0,r1
383     ceq32 apaga_pos2; apaga pixel pos2
384     load32 r0, flag_impr3
385     cmp32 r0,r1
386     ceq32 apaga_pos3; apaga pixel pos3
387     load32 r0, flag_impr4
388     cmp32 r0,r1
389     ceq32 apaga_pos4; apaga pixel pos4
390
391     call32 Readmove; atualiza posições para o próximo mov32imento
392     call32 IS_VALID_POS
393
394     load32 r4, cor_atual ; cor do bloco
395
396     load32 r5, bloco ; carrega caracter do bloco
397
398     add32 r5, r5, r4 ;deixa o pixel do bloco com a cor
399
400     call32 verifi_apagaIni
401
402     load32 r0, flag_impr1
403     loadn32 r1, #1
404     cmp32 r0,r1
405     ceq32 impr_pos1; imprime pixel pos1
406     load32 r0, flag_impr2
407     cmp32 r0,r1
408     ceq32 impr_pos2; imprime pixel pos2
409     load32 r0, flag_impr3
410     cmp32 r0,r1
411     ceq32 impr_pos3; imprime pixel pos3
412     load32 r0, flag_impr4
413     cmp32 r0,r1
414     ceq32 impr_pos4; imprime pixel pos4
415
416     pop32 r5
417     pop32 r4
418     pop32 r3
419     pop32 r2
420     pop32 r1
421     pop32 r0
422     rts32
423
424 Readmove:
425     push32 r0
426     push32 r1
427     push32 r2
428     push32 r3
429     push32 r4
430     push32 r5
431     push32 r6
432
433     inchar32 r0
434
435     loadn32 r1, #255
436     cmp32 r1, r0
437     jeq32 break1
438

```

```

439     loadn32 r1, #'a'
440     cmp32 r0, r1
441     store32 dir, r1
442     jeq32 switch_A
443
444     loadn32 r1, #'s'
445     cmp32 r0, r1
446     store32 dir, r1
447     jeq32 switch_S
448
449     loadn32 r1, #'d'
450     cmp32 r0, r1
451     store32 dir, r1
452     jeq32 switch_D
453
454     loadn32 r1, #'q';rotação da peça
455     cmp32 r0, r1
456     jeq32 switch_Q
457
458     loadn32 r1, #'e';rotação da peça
459     cmp32 r0, r1
460     jeq32 switch_E
461
462     jmp32 break1
463
464     switch_A:
465     dec32 r2
466     dec32 r3
467     dec32 r4
468     dec32 r5
469     jmp32 break;
470
471     switch_S:
472     loadn32 r6, #40
473     add32 r2, r2, r6
474     add32 r3, r3, r6
475     add32 r4, r4, r6
476     add32 r5, r5, r6
477     loadn32 r0, #1
478     store32 flag_parada, r0
479     jmp32 break;
480
481     switch_D:
482     inc32 r2
483     inc32 r3
484     inc32 r4
485     inc32 r5
486     jmp32 break;
487
488     switch_Q:
489     load32 r1, posi
490     store32 last_posi, r1
491     loadn32 r2, #4
492     cmp32 r1, r2
493     jeq32 switch_q_
494     inc32 r1
495     store32 posi, r1
496     call32 select
497     jmp32 break1;
498     switch_q_:
499     loadn32 r1, #1
500     store32 posi, r1
501     call32 select
502     jmp32 break1;
503
504     switch_E:
505     load32 r1, posi
506     store32 last_posi, r1
507     loadn32 r2, #1
508     cmp32 r1, r2
509     jeq32 switch_e_
510     dec32 r1
511     store32 posi, r1

```

```

512     call32 select
513     jmp32 break1;
514     switch_e_:
515     loadn32 r1, #4
516     store32 posi, r1
517     call32 select
518     jmp32 break1;
519
520     break:
521     store32 pos1, r2
522     store32 pos2, r3
523     store32 pos3, r4
524     store32 pos4, r5
525     break1:
526     pop32 r6
527     pop32 r5
528     pop32 r4
529     pop32 r3
530     pop32 r2
531     pop32 r1
532     pop32 r0
533     rts32
534
535     IS_VALID_POS:
536     push32 fr
537     push32 r0
538     push32 r1
539     push32 r2
540     push32 r3
541     push32 r4
542     push32 r5
543     push32 r6
544
545     load32 r0, pos1
546     load32 r1, pos2
547     load32 r2, pos3
548     load32 r3, pos4
549
550     ;----correção das posições----
551     loadn32 r4, #40
552     div32 r4, r0, r4
553     add32 r0, r4, r0
554
555     loadn32 r4, #40
556     div32 r4, r1, r4
557     add32 r1, r4, r1
558
559     loadn32 r4, #40
560     div32 r4, r2, r4
561     add32 r2, r4, r2
562
563     loadn32 r4, #40
564     div32 r4, r3, r4
565     add32 r3, r4, r3
566
567     loadn32 r4, $S1L1
568     loadn32 r6, #' '
569
570     add32 r5, r4, r0
571     ; r1 contém o caracter valido
572     ; S1L1[POS] == ' '
573     loadi32 r5, r5
574     cmp32 r5, r6
575     jne32 IS_VALID_POS_N
576
577     add32 r5, r4, r1
578     ; r1 contém o caracter valido
579     ; S1L1[POS] == ' '
580     loadi32 r5, r5
581     cmp32 r5, r6
582     jne32 IS_VALID_POS_N
583
584     add32 r5, r4, r2

```

```

585 ; r1 contem o caracter valido
586 ; S1L1[POS] == ' '
587 loadi32 r5, r5
588 cmp32 r5, r6
589 jne32 IS_VALID_POS_N
590
591 add32 r5, r4, r3
592 ; r1 contem o caracter valido
593 ; S1L1[POS] == ' '
594 loadi32 r5, r5
595 cmp32 r5, r6
596 jne32 IS_VALID_POS_N
597
598 jmp32 VALID_END
599
600 IS_VALID_POS_N:
601
602 ;pos = last_pos;
603 load32 r0, last_pos1
604 store32 pos1, r0
605
606 load32 r0, last_pos2
607 store32 pos2, r0
608
609 load32 r0, last_pos3
610 store32 pos3, r0
611
612 load32 r0, last_pos4
613 store32 pos4, r0
614
615 load32 r0, last_dir
616 store32 dir, r0
617
618 load32 r0, last_posi
619 store32 posi, r0
620
621 load32 r0, flag_parada
622 loadn32 r1, #1
623 cmp32 r0, r1
624 jeq32 parada
625 jmp32 VALID_END
626
627 parada:
628 store32 flag_parou, r1
629
630 VALID_END:
631
632 pop32 r6
633 pop32 r5
634 pop32 r4
635 pop32 r3
636 pop32 r2
637 pop32 r1
638 pop32 r0
639 pop32 fr
640 rts32
641
642
643 queda:
644 push32 r0
645 push32 r1
646 push32 r2
647 push32 r3
648 push32 r4
649 push32 r5
650
651 loadn32 r0, #1
652 store32 flag_parada, r0
653
654 load32 r2, dir
655 store32 last_dir, r2
656
657 load32 r2, pos1

```

```

658     store32 last_pos1, r2
659
660     load32 r3, pos2
661     store32 last_pos2, r3
662
663     load32 r4, pos3
664     store32 last_pos3, r4
665
666     load32 r5, pos4
667     store32 last_pos4, r5
668
669     call32 verifi_apagaIni
670
671
672     load32 r0, flag_impr1
673     loadn32 r1, #1
674     cmp32 r0,r1
675     ceq32 apaga_pos1; apaga pixel pos1
676     load32 r0, flag_impr2
677     cmp32 r0,r1
678     ceq32 apaga_pos2; apaga pixel pos2
679     load32 r0, flag_impr3
680     cmp32 r0,r1
681     ceq32 apaga_pos3; apaga pixel pos3
682     load32 r0, flag_impr4
683     cmp32 r0,r1
684     ceq32 apaga_pos4; apaga pixel pos4
685
686
687     load32 r0, pos1
688     loadn32 r5, #40
689     add32 r0, r0, r5
690     store32 pos1, r0
691
692     load32 r1, pos2
693     add32 r1, r1, r5
694     store32 pos2, r1
695
696
697     load32 r2, pos3
698     add32 r2, r2, r5
699     store32 pos3, r2
700
701     load32 r3, pos4
702     add32 r3, r3, r5
703     store32 pos4, r3
704
705
706     call32 IS_VALID_POS
707
708     load32 r4, cor_atual ; cor do bloco
709
710     load32 r5, bloco ; carrega caracter do bloco
711
712     add32 r5, r5, r4 ;deixa o pixel do bloco com a cor
713
714     call32 verifi_apagaIni
715
716     load32 r0, flag_impr1
717     loadn32 r1, #1
718     cmp32 r0,r1
719     ceq32 impr_pos1; imprime pixel pos1
720     load32 r0, flag_impr2
721     cmp32 r0,r1
722     ceq32 impr_pos2; imprime pixel pos2
723     load32 r0, flag_impr3
724     cmp32 r0,r1
725     ceq32 impr_pos3; imprime pixel pos3
726     load32 r0, flag_impr4
727     cmp32 r0,r1
728     ceq32 impr_pos4; imprime pixel pos4
729
730     loadn32 r6, #10

```



```

731     store32 contador, r6
732     pop32 r5
733     pop32 r4
734     pop32 r3
735     pop32 r2
736     pop32 r1
737     pop32 r0
738     rts32
739
740 start_menu:
741     pop32 fr
742     pop32 r0
743     pop32 r1
744     pop32 r2
745     pop32 r3
746
747     call32 ClearScreen ; Limpa a Tela
748     loadn32 r1, #S3L1 ; tela inicial
749     loadn32 r2, #0 ; cor da tela
750     loadn32 r0, #0; Posicao na tela onde a mensagem sera escrita
751     call32 PrintScreen
752
753 ;----Verificação da Tecla Enter----
754 loop_sm:
755
756     loadn32 r0, #13
757     inchar32 r1
758     cmp32 r0, r1
759     jne32 loop_sm
760
761     push32 r3
762     push32 r2
763     push32 r1
764     push32 r0
765     push32 fr
766     jmp32 BACK_MAIN
767
768 ClearScreen:
769     push32 fr ; protege registrador de frags
770     push32 r0 ; contador para percorrer a tela
771     push32 r1 ; valor do espaco em branco
772     push32 r2;
773
774     loadn32 r0, #1200 ; tamanho da tela
775     loadn32 r1, #' ' ; espaco em branco
776
777     ClearScreen_loop: ; de 1200 ate 0
778     dec32 r0 ; dec32remento contador
779     outchar32 r1, r0 ; imprime espaco em branco
780     jnz32 ClearScreen_loop ; jump se zero
781     pop32 r2 ;
782     pop32 r1 ;
783     pop32 r0 ;
784     pop32 fr ;
785     rts32
786
787 PrintScreen:
788     push32 fr ; protege registrador de frags
789     push32 r0 ; contador
790     push32 r1 ; endereco da mensagem
791     push32 r2 ; cor da mensagem
792     push32 r3 ;
793     push32 r4 ;
794     push32 r5 ;
795
796     loadn32 r0, #0 ; contador
797     loadn32 r3, #1199 ; tamanho da tela
798     loadn32 r4, #41 ; inc32remento da memoria
799     loadn32 r5, #40 ; inc32remento do contador
800
801     PrintScreen_loop:
802     call32 Imprimestr
803     add32 r1, r1, r4

```

```

804         add32 r0, r0, r5
805         cmp32 r0, r3
806         jle32 PrintScreen_loop
807
808         pop32 r5
809         pop32 r4
810         pop32 r3
811         pop32 r2
812         pop32 r1
813         pop32 r0
814         pop32 fr
815         rts32
816
817 Imprimestr:
818
819         push32 r0      ; protege o r0 na pilha
820         push32 r1      ; protege o r1 na pilha
821         push32 r2      ; protege o r2 na pilha
822         push32 r3      ; protege o r3 na pilha
823         push32 r4      ; protege o r4 na pilha
824
825         loadn32 r3, #'\\0' ; Critério de parada
826
827 ImprimestrLoop:
828         loadi32 r4, r1
829         cmp32 r4, r3
830         jeq32 ImprimestrSai
831         add32 r4, r2, r4
832         outchar32 r4, r0
833         inc32 r0
834         inc32 r1
835         jmp32 ImprimestrLoop
836
837 ImprimestrSai:
838         pop32 r4      ; Resgata os valores dos registradores
839         pop32 r3
840         pop32 r2
841         pop32 r1
842         pop32 r0
843         rts32
844
845 ajusta_delay:
846         push32 r2
847         push32 r3
848         load32 r2, level
849         loadn32 r3, #1
850         cmp32 r2, r3
851         ceq32 seta1
852         loadn32 r3, #2
853         cmp32 r2, r3
854         ceq32 seta2
855         loadn32 r3, #3
856         cmp32 r2, r3
857         ceq32 seta3
858
859         pop32 r3
860         pop32 r2
861         rts32
862
863
864 Delay:
865                                     ;Utiliza push32 e pop32 para nao afetar os Ristradores do
                                     programa princ32ipal
866         push32 r0
867         push32 r1
868         push32 r5
869         push32 r6
870
871
872         load32 r1, constB
873
874
875         Delay_volta2:                ; contador de tempo quebrado em duas partes (dois

```

```

loops de dec32remento)
876     load32 r0, constA
877     Delay_volta:
878     dec32 r0; (4*a + 6)b = 1000000 == 1 seg em um clock de 1MHz
879
880     jnz32 Delay_volta
881     dec32 r1
882     jnz32 Delay_volta2
883
884     pop32 r6
885     pop32 r5
886     pop32 r1
887     pop32 r0
888
889     rts32                                ;return
890
891 seta1:
892     push32 r0
893     push32 r1
894     loadn32 r1,#5;a
895     store32 constB,r1
896     loadn32 r0, #800;b
897     store32 constA,r0
898     pop32 r0
899     pop32 r1
900     rts32
901
902 seta2:
903     push32 r0
904     push32 r1
905     loadn32 r1,#3;a
906     store32 constB,r1
907     loadn32 r0, #500;b
908     store32 constA,r0
909     pop32 r0
910     pop32 r1
911     rts32
912
913 seta3:
914     push32 r0
915     push32 r1
916     loadn32 r1,#2;a
917     store32 constB,r1
918     loadn32 r0, #300;b
919     store32 constA,r0
920     pop32 r1
921     pop32 r0
922     rts32
923
924 seta4:
925     push32 r0
926     push32 r1
927     loadn32 r1,#2;a
928     store32 constB,r1
929     loadn32 r0, #200;b
930     store32 constA,r0
931     pop32 r0
932     pop32 r1
933     rts32
934
935 getRandNum:
936     push32 fr
937     push32 r0;
938     push32 r1;
939     push32 r2;
940     loadn32 r2, #5
941
942 get_rand loop:
943     load32 r1, seed ; read i
944     loadn32 r0, #rand ; read vector
945     add32 r0, r0, r1 ; r0 = vector[i]
946
947     inc32 r1 ;i++

```

```

948     store32 seed, r1
949     cmp32 r1, r2 ;if (i < 5)
950     jel32 getRandNum_
951     loadn32 r1, #0 ;if (i >= 5), i = 0
952     store32 seed, r1
953     jmp32 get_rand_loop
954     getRandNum :
955     loadi32 r1, r0
956     store32 tipo_bloco, r1
957     pop32 r2
958     pop32 r1;
959     pop32 r0;
960     pop32 fr
961     rts32
962
963     getRandNum1:
964     push32 fr
965     push32 r0;
966     push32 r1;
967     push32 r2;
968     loadn32 r2, #5
969
970     get_rand_loop1:
971     load32 r1, seed ; read i
972     loadn32 r0, #rand ; read vector
973     add32 r0, r0, r1 ; r0 = vector[i]
974
975     inc32 r1 ;i++
976     store32 seed, r1
977     cmp32 r1, r2 ;if (i < 5)
978     jel32 getRandNum_1
979     loadn32 r1, #0 ;if (i >= 5), i = 0
980     store32 seed, r1
981     jmp32 get_rand_loop1
982     getRandNum_1:
983     loadi32 r1, r0
984     cmp32 r1, r2
985     jeq32 get_rand_loop1
986     store32 posi, r1
987     pop32 r2
988     pop32 r1;
989     pop32 r0;
990     pop32 fr
991     rts32
992
993     getRandNumNEXT:
994     push32 fr
995     push32 r0;
996     push32 r1;
997     push32 r2;
998     loadn32 r2, #5
999
1000     get_rand_loopNEXT:
1001     load32 r1, seed_next ; read i
1002     loadn32 r0, #rand ; read vector
1003     add32 r0, r0, r1 ; r0 = vector[i]
1004
1005     inc32 r1 ;i++
1006     store32 seed_next, r1
1007     cmp32 r1, r2 ;if (i < 5)
1008     jel32 getRandNum_NEXT
1009     loadn32 r1, #0 ;if (i >= 5), i = 0
1010     store32 seed_next, r1
1011     jmp32 get_rand_loopNEXT
1012     getRandNum_NEXT:
1013     loadi32 r1, r0
1014     store32 tipo_bloco, r1
1015     pop32 r2
1016     pop32 r1;
1017     pop32 r0;
1018     pop32 fr
1019     rts32
1020

```

```

1021 getRandNum1NEXT:
1022     push32 fr
1023     push32 r0;
1024     push32 r1;
1025     push32 r2;
1026     loadn32 r2, #5
1027
1028 get_rand_loop1NEXT:
1029     load32 r1, seed_next1 ; read i
1030     loadn32 r0, #rand_ ; read vector
1031     add32 r0, r0, r1 ; r0 = vector[i]
1032
1033     inc32 r1 ;i++
1034     store32 seed_next1, r1
1035     cmp32 r1, r2 ;if (i < 5)
1036     jcl32 getRandNum_1NEXT
1037     loadn32 r1, #0 ;if (i >= 5), i = 0
1038     store32 seed_next1, r1
1039     jmp32 get_rand_loop1NEXT
1040 getRandNum_1NEXT:
1041     loadi32 r1, r0
1042     cmp32 r1, r2
1043     jeq32 get_rand_loop1NEXT
1044     store32 posi, r1
1045     pop32 r2
1046     pop32 r1;
1047     pop32 r0;
1048     pop32 fr
1049     rts32
1050
1051 select:
1052     push32 fr
1053     push32 r0
1054     push32 r1
1055     push32 r2
1056     push32 r3
1057     push32 r4
1058     push32 r5
1059
1060     load32 r0,pos1
1061     store32 last_pos1, r0
1062     load32 r0,pos2
1063     store32 last_pos2, r0
1064     load32 r0,pos3
1065     store32 last_pos3, r0
1066     load32 r0,pos4
1067     store32 last_pos4, r0
1068
1069     load32 r0,tiplo_bloco
1070     load32 r1,pos1
1071     load32 r2, pos1
1072     loadn32 r4, #40
1073     loadn32 r5, #1
1074
1075     loadn32 r3, #1
1076     cmp32 r0, r3
1077     jeq32 switch_bloco1
1078
1079     loadn32 r3, #2
1080     cmp32 r0, r3
1081     jeq32 switch_bloco2
1082
1083     loadn32 r3, #3
1084     cmp32 r0, r3
1085     jeq32 switch_bloco3
1086
1087     loadn32 r3, #4
1088     cmp32 r0, r3
1089     jeq32 switch_bloco4
1090
1091     loadn32 r3, #5
1092     cmp32 r0, r3
1093     jeq32 switch_bloco5

```

```

1094      jmp32 final
1095
1096
1097      ;-----Posições Bloco 1-----
1098
1099      switch_bloco1:
1100      loadn32 r3,#2816 ; cor do bloco amarelo
1101      store32 cor_atual, r3
1102
1103      loadn32 r3, #1
1104      cmp32 r1, r3
1105      jeq32 switch_bloco1posi1
1106
1107      loadn32 r3, #2
1108      cmp32 r1, r3
1109      jeq32 switch_bloco1posi2
1110
1111      loadn32 r3, #3
1112      cmp32 r1, r3
1113      jeq32 switch_bloco1posi3
1114
1115      loadn32 r3, #4
1116      cmp32 r1, r3
1117      jeq32 switch_bloco1posi4
1118      jmp32 final
1119
1120      ;---definição de novas coordenadas das posições---
1121      switch_bloco1posi1:
1122      sub32 r3,r2,r4
1123      store32 pos2, r3
1124      add32 r3,r2,r4
1125      store32 pos3, r3
1126      add32 r3,r3,r5
1127      store32 pos4, r3
1128      jmp32 final
1129
1130      switch_bloco1posi2:
1131      sub32 r3,r2,r5
1132      store32 pos2, r3
1133      add32 r3,r2,r5
1134      store32 pos3, r3
1135      sub32 r3,r3,r4
1136      store32 pos4, r3
1137      jmp32 final
1138
1139      switch_bloco1posi3:
1140      add32 r3,r2,r4
1141      store32 pos2,r3
1142      sub32 r3,r2,r4
1143      store32 pos3,r3
1144      sub32 r3,r3,r5
1145      store32 pos4,r3
1146      jmp32 final
1147
1148      switch_bloco1posi4:
1149      add32 r3,r2,r5
1150      store32 pos2,r3
1151      sub32 r3,r2,r5
1152      store32 pos3,r3
1153      add32 r3,r3,r4
1154      store32 pos4,r3
1155
1156      jmp32 final
1157
1158      ;-----Posições Bloco 2-----
1159
1160      switch_bloco2:
1161      loadn32 r3,#2304 ; cor do bloco vermelho
1162      store32 cor_atual, r3
1163
1164      loadn32 r3, #1
1165      cmp32 r1, r3
1166      jeq32 switch_bloco2posi1

```

```

1167
1168 loadn32 r3, #2
1169 cmp32 r1, r3
1170 jeq32 switch_bloco2posi2
1171
1172 loadn32 r3, #3
1173 cmp32 r1, r3
1174 jeq32 switch_bloco2posi3
1175
1176 loadn32 r3, #4
1177 cmp32 r1, r3
1178 jeq32 switch_bloco2posi4
1179 jmp32 final
1180
1181 ;---definição de novas coordenadas das posições---
1182 switch_bloco2posi1:
1183 sub32 r3,r2,r4
1184 store32 pos2, r3
1185 add32 r3,r2,r4
1186 store32 pos3, r3
1187 add32 r3,r3,r4
1188 store32 pos4, r3
1189 jmp32 final
1190
1191 switch_bloco2posi2:
1192 sub32 r3,r2,r5
1193 store32 pos2, r3
1194 add32 r3,r2,r5
1195 store32 pos3, r3
1196 add32 r3,r3,r5
1197 store32 pos4, r3
1198 jmp32 final
1199
1200 switch_bloco2posi3:
1201 add32 r3,r2,r4
1202 store32 pos2,r3
1203 sub32 r3,r2,r4
1204 store32 pos3,r3
1205 sub32 r3,r3,r4
1206 store32 pos4,r3
1207 jmp32 final
1208
1209 switch_bloco2posi4:
1210 add32 r3,r2,r5
1211 store32 pos2,r3
1212 sub32 r3,r2,r5
1213 store32 pos3,r3
1214 sub32 r3,r3,r5
1215 store32 pos4,r3
1216
1217 jmp32 final
1218
1219 ;-----Posições Bloco 3-----
1220
1221 switch_bloco3:
1222 loadn32 r3,#512 ; cor do bloco verde
1223 store32 cor_atual, r3
1224
1225 loadn32 r3, #1
1226 cmp32 r1, r3
1227 jeq32 switch_bloco3posi1
1228
1229 loadn32 r3, #2
1230 cmp32 r1, r3
1231 jeq32 switch_bloco3posi2
1232
1233 loadn32 r3, #3
1234 cmp32 r1, r3
1235 jeq32 switch_bloco3posi3
1236
1237 loadn32 r3, #4
1238 cmp32 r1, r3
1239 jeq32 switch_bloco3posi4

```

```

1240 jmp32 final
1241
1242 ;----definição de novas coordenadas das posições----
1243 switch_bloco3posi1:
1244 sub32 r3,r2,r5
1245 store32 pos2, r3
1246 add32 r3,r3,r4
1247 store32 pos3, r3
1248 add32 r3,r2,r4
1249 store32 pos4, r3
1250 jmp32 final
1251
1252 switch_bloco3posi2:
1253 sub32 r3,r2,r5
1254 store32 pos2, r3
1255 add32 r3,r3,r4
1256 store32 pos3, r3
1257 add32 r3,r2,r4
1258 store32 pos4, r3
1259 jmp32 final
1260
1261 switch_bloco3posi3:
1262 sub32 r3,r2,r5
1263 store32 pos2, r3
1264 add32 r3,r3,r4
1265 store32 pos3, r3
1266 add32 r3,r2,r4
1267 store32 pos4, r3
1268 jmp32 final
1269
1270 switch_bloco3posi4:
1271 sub32 r3,r2,r5
1272 store32 pos2, r3
1273 add32 r3,r3,r4
1274 store32 pos3, r3
1275 add32 r3,r2,r4
1276 store32 pos4, r3
1277
1278 jmp32 final
1279
1280
1281 ;-----Posições Bloco 4-----
1282
1283 switch_bloco4:
1284 loadn32 r3,#1024 ; cor do bloco azul marinho
1285 store32 cor_atual, r3
1286
1287 loadn32 r3, #1
1288 cmp32 r1, r3
1289 jeq32 switch_bloco4posi1
1290
1291 loadn32 r3, #2
1292 cmp32 r1, r3
1293 jeq32 switch_bloco4posi2
1294
1295 loadn32 r3, #3
1296 cmp32 r1, r3
1297 jeq32 switch_bloco4posi3
1298
1299 loadn32 r3, #4
1300 cmp32 r1, r3
1301 jeq32 switch_bloco4posi4
1302 jmp32 final
1303
1304 ;----definição de novas coordenadas das posições----
1305 switch_bloco4posi1:
1306 sub32 r3,r2,r5
1307 store32 pos2, r3
1308 add32 r3,r2,r4
1309 store32 pos3, r3
1310 add32 r3,r3,r5
1311 store32 pos4, r3
1312 jmp32 final

```



```

1313
1314     switch_bloco4posi2:
1315     add32 r3,r2,r4
1316     store32 pos2, r3
1317     add32 r3,r2,r5
1318     store32 pos3, r3
1319     sub32 r3,r3,r4
1320     store32 pos4, r3
1321     jmp32 final
1322
1323     switch_bloco4posi3:
1324     add32 r3,r2,r5
1325     store32 pos2,r3
1326     sub32 r3,r2,r4
1327     store32 pos3,r3
1328     sub32 r3,r3,r5
1329     store32 pos4,r3
1330     jmp32 final
1331
1332     switch_bloco4posi4:
1333     sub32 r3,r2,r4
1334     store32 pos2,r3
1335     sub32 r3,r2,r5
1336     store32 pos3,r3
1337     add32 r3,r3,r4
1338     store32 pos4,r3
1339
1340     jmp32 final
1341
1342     ;-----Posições Bloco 5-----
1343
1344     switch_bloco5:
1345     loadn32 r3,#1280 ; cor do bloco roxo
1346     store32 cor_atual, r3
1347
1348     loadn32 r3, #1
1349     cmp32 r1, r3
1350     jeq32 switch_bloco5posi1
1351
1352     loadn32 r3, #2
1353     cmp32 r1, r3
1354     jeq32 switch_bloco5posi2
1355
1356     loadn32 r3, #3
1357     cmp32 r1, r3
1358     jeq32 switch_bloco5posi3
1359
1360     loadn32 r3, #4
1361     cmp32 r1, r3
1362     jeq32 switch_bloco5posi4
1363     jmp32 final
1364
1365     ;---definição de novas coordenadas das posições---
1366     switch_bloco5posi1:
1367     add32 r3,r2,r4
1368     store32 pos2, r3
1369     add32 r3,r3,r5
1370     store32 pos3, r3
1371     sub32 r3,r3,r5
1372     sub32 r3,r3,r5
1373     store32 pos4, r3
1374     jmp32 final
1375
1376     switch_bloco5posi2:
1377     add32 r3,r2,r5
1378     store32 pos2, r3
1379     add32 r3,r3,r4
1380     store32 pos3, r3
1381     sub32 r3,r3,r4
1382     sub32 r3,r3,r4
1383     store32 pos4, r3
1384     jmp32 final
1385

```

```

1386 switch_bloco5posi3:
1387 sub32 r3,r2,r4
1388 store32 pos2,r3
1389 sub32 r3,r3,r5
1390 store32 pos3,r3
1391 add32 r3,r3,r5
1392 add32 r3,r3,r5
1393 store32 pos4,r3
1394 jmp32 final
1395
1396 switch_bloco5posi4:
1397 sub32 r3,r2,r5
1398 store32 pos2,r3
1399 add32 r3,r3,r4
1400 store32 pos3,r3
1401 sub32 r3,r3,r4
1402 sub32 r3,r3,r4
1403 store32 pos4,r3
1404
1405 jmp32 final
1406
1407 final:
1408     pop32 fr
1409     pop32 r0
1410     pop32 r1
1411     pop32 r2
1412     pop32 r3
1413     pop32 r4
1414     pop32 r5
1415 rts32
1416
1417 grava_bloco:
1418     push32 r0
1419     push32 r1
1420     push32 r2
1421     push32 r3
1422     push32 r4
1423     push32 r5
1424     push32 r6
1425     push32 r7
1426
1427     load32 r0, pos1
1428     load32 r1, pos2
1429     load32 r2, pos3
1430     load32 r3, pos4
1431
1432 ;----correção das posições----
1433     loadn32 r4, #40
1434     div32 r4, r0, r4
1435     add32 r0, r4, r0
1436
1437     loadn32 r4, #40
1438     div32 r4, r1, r4
1439     add32 r1, r4, r1
1440
1441     loadn32 r4, #40
1442     div32 r4, r2, r4
1443     add32 r2, r4, r2
1444
1445     loadn32 r4, #40
1446     div32 r4, r3, r4
1447     add32 r3, r4, r3
1448
1449 ;-----memorização no mapa das posições dos blocos-----
1450
1451     loadn32 r4, #S1L1
1452     load32 r5, bloco
1453
1454     add32 r6, r4, r0
1455     storei32 r6,r5
1456
1457     add32 r6, r4, r1
1458     storei32 r6,r5

```

```

1459
1460      add32 r6, r4, r2
1461      storei32 r6,r5
1462
1463      add32 r6, r4, r3
1464      storei32 r6,r5
1465
1466      ;-----memorização das cores na matriz de memória de cores-----
1467
1468      loadn32 r4, #mapa_corL1
1469      load32 r5, cor_atual
1470
1471      store32 cor_inst, r5; fornece a cor desejada para mapeamento
1472
1473      call32 select_cor_; converte cor para código entre 1 e 5
1474
1475      load32 r5, num_cor; obtém o código da cor
1476
1477      add32 r6, r4, r0
1478      storei32 r6,r5
1479
1480      add32 r6, r4, r1
1481      storei32 r6,r5
1482
1483      add32 r6, r4, r2
1484      storei32 r6,r5
1485
1486      add32 r6, r4, r3
1487      storei32 r6,r5
1488
1489      pop32 r7
1490      pop32 r6
1491      pop32 r5
1492      pop32 r4
1493      pop32 r3
1494      pop32 r2
1495      pop32 r1
1496      pop32 r0
1497
1498      rts32
1499
1500      pontos_linha:
1501      push32 r0
1502      push32 r1
1503      push32 r2
1504      push32 r3
1505      push32 r4
1506      push32 r5
1507      push32 r6
1508      push32 r7
1509
1510      ;-----zera o vetor de linha a serem apagadas-----
1511      loadn32 r0,#apaga_linha
1512      loadn32 r1,#0
1513      loadn32 r2,#0
1514
1515      add32 r3, r0, r1
1516      storei32 r3,r2
1517      inc32 r1
1518      add32 r3, r0, r1
1519      storei32 r3,r2
1520      inc32 r1
1521      add32 r3, r0, r1
1522      storei32 r3,r2
1523      inc32 r1
1524      add32 r3, r0, r1
1525      storei32 r3,r2
1526      inc32 r1
1527      add32 r3, r0, r1
1528      storei32 r3,r2
1529      inc32 r1
1530      add32 r3, r0, r1
1531      storei32 r3,r2

```

```

1532 inc32 r1
1533 add32 r3, r0, r1
1534 storei32 r3,r2
1535 inc32 r1
1536 add32 r3, r0, r1
1537 storei32 r3,r2
1538 inc32 r1
1539 add32 r3, r0, r1
1540 storei32 r3,r2
1541 inc32 r1
1542 add32 r3, r0, r1
1543 storei32 r3,r2
1544 inc32 r1
1545 add32 r3, r0, r1
1546 storei32 r3,r2
1547 inc32 r1
1548 add32 r3, r0, r1
1549 storei32 r3,r2
1550 inc32 r1
1551 add32 r3, r0, r1
1552 storei32 r3,r2
1553 inc32 r1
1554 add32 r3, r0, r1
1555 storei32 r3,r2
1556 inc32 r1
1557 add32 r3, r0, r1
1558 storei32 r3,r2
1559 inc32 r1
1560 add32 r3, r0, r1
1561 storei32 r3,r2
1562 inc32 r1
1563 add32 r3, r0, r1
1564 storei32 r3,r2
1565 inc32 r1
1566 add32 r3, r0, r1
1567 storei32 r3,r2
1568 inc32 r1
1569 add32 r3, r0, r1
1570 storei32 r3,r2
1571 inc32 r1
1572 add32 r3, r0, r1
1573 storei32 r3,r2
1574
1575     load32 r1, num_linha
1576     load32 r2, ini_linha
1577     load32 r3, num_colun
1578
1579
1580 verifi_ini:
1581     call32 ponteiro
1582     loadn32 r4, #0; contador de espaços vazios
1583     call32 verif_colun; verifica cada linha para pontuação
1584     cmp32 r4,r3
1585     ceq32 pula
1586     cmp32 r4,r3
1587     jeq32 finaliza
1588     loadn32 r5, #0
1589     cmp32 r4, r5
1590     ceq32 linha_ok
1591     dec32 r2
1592     dec32 r1
1593     jnz32 verifi_ini
1594
1595 finaliza:
1596     pop32 r7
1597     pop32 r6
1598     pop32 r5
1599     pop32 r4
1600     pop32 r3
1601     pop32 r2
1602     pop32 r1
1603     pop32 r0
1604

```

```

1605         rts32
1606
1607     verif_colun:
1608         push32 r0
1609         push32 r1
1610         push32 r2
1611         push32 r5
1612         push32 r6
1613
1614         loadn32 r0, #0
1615         load32 r1, posi_ponteiro
1616         load32 r2, bloco
1617         loadn32 r6, #S1L1
1618         start:
1619             cmp32 r0, r3
1620             jeq32 stop
1621             add32 r5, r0, r1
1622             add32 r5, r5, r6
1623             loadi32 r5, r5
1624             inc32 r0
1625             cmp32 r5, r2
1626             jeq32 start
1627             inc32 r4
1628             jmp32 start
1629         stop:
1630             pop32 r6
1631             pop32 r5
1632             pop32 r2
1633             pop32 r1
1634             pop32 r0
1635         rts32
1636
1637     ponteiro:
1638         push32 r0
1639         push32 r1
1640         push32 r3
1641         push32 r4
1642         push32 r5
1643
1644         loadn32 r0, #40
1645         load32 r1, ini_colun
1646         dec32 r1
1647         mov32 r5, r2
1648         dec32 r5
1649         mul32 r3, r5, r0
1650         add32 r3, r3, r1
1651
1652         div32 r4, r3, r0
1653         add32 r4, r3, r4
1654
1655         store32 posi_ponteiro, r4
1656
1657         pop32 r5
1658         pop32 r4
1659         pop32 r3
1660         pop32 r1
1661         pop32 r0
1662         rts32
1663
1664     pula: push32 r0
1665         push32 r1
1666         push32 r3
1667         push32 r4
1668         push32 r5
1669
1670         loadn32 r0, #5
1671         loadn32 r5, #2
1672         loadn32 r3, #apaga_linha
1673         sub32 r1, r2, r0
1674         add32 r4, r3, r1
1675         storei32 r4, r5
1676
1677         pop32 r5

```

```

1678         pop32 r4
1679         pop32 r3
1680         pop32 r1
1681         pop32 r0
1682     rts32
1683
1684     linha_ok:
1685         push32 r0
1686         push32 r1
1687         push32 r3
1688         push32 r4
1689         push32 r5
1690
1691         loadn32 r0, #5
1692         loadn32 r5, #1
1693         loadn32 r3, #apaga_linha
1694         sub32 r1, r2, r0
1695         add32 r4, r3, r1
1696         storei32 r4, r5
1697         loadn32 r0, #1
1698         store32 flag_deleta, r0
1699
1700         call32 pontos
1701
1702         pop32 r5
1703         pop32 r4
1704         pop32 r3
1705         pop32 r1
1706         pop32 r0
1707
1708         rts32
1709
1710     apagar:
1711         push32 r0
1712         push32 r1
1713         push32 r2
1714         push32 r3
1715         push32 r4
1716         push32 r5
1717         push32 r6
1718         push32 r7
1719
1720
1721         load32 r6, bloco; printa de branco linhas pontuadas
1722         call32 print_
1723         call32 Delay_
1724
1725         loadn32 r6, #' '; apaga linhas pontuadas
1726         call32 print_
1727         call32 Delay_
1728
1729         load32 r6, bloco; printa de branco linhas pontuadas
1730         call32 print_
1731         call32 Delay_
1732
1733         loadn32 r6, #' '; apaga linhas pontuadas
1734         call32 print_
1735         call32 Delay_
1736
1737         load32 r6, bloco; printa de branco linhas pontuadas
1738         call32 print_
1739         call32 Delay_
1740
1741         loadn32 r6, #' '; apaga linhas pontuadas
1742         call32 print_
1743         call32 Delay_
1744
1745         load32 r6, bloco; printa de branco linhas pontuadas
1746         call32 print_
1747         call32 Delay_
1748
1749         loadn32 r6, #' '; apaga linhas pontuadas
1750         call32 print_

```

```

1751         call32 Delay
1752
1753
1754         call32 deleta_desloca_bloco; apaga de fato os blocos alinhados
1755         loadn32 r6, #0
1756         store32 contador, r6
1757         store32 flag_deleta, r6; zera o flag para deletar linhas
1758
1759         pop32 r7
1760         pop32 r6
1761         pop32 r5
1762         pop32 r4
1763         pop32 r3
1764         pop32 r2
1765         pop32 r1
1766         pop32 r0
1767
1768         rts32
1769
1770     print_:
1771         push32 r0
1772         push32 r1
1773         push32 r2
1774         push32 r3
1775         push32 r4
1776         push32 r5
1777
1778         loadn32 r0, #apaga_linha
1779         load32 r1, num_linha
1780         load32 r2, ini_linha
1781
1782
1783     apag:
1784
1785         dec32 r1
1786         add32 r3, r0, r1
1787         loadi32 r4, r3
1788         loadn32 r3, #2
1789         cmp32 r4, r3
1790         jeq32 fim_apag
1791         loadn32 r3, #1
1792         cmp32 r4, r3
1793         ceq32 printar_
1794         dec32 r2
1795         loadn32 r3, #0
1796         cmp32 r1, r3
1797         jne32 apag
1798     fim_apag:
1799         pop32 r5
1800         pop32 r4
1801         pop32 r3
1802         pop32 r2
1803         pop32 r1
1804         pop32 r0
1805
1806
1807     rts32
1808
1809     printar_:
1810
1811         push32 r0
1812         push32 r1
1813         push32 r3
1814         push32 r4
1815         push32 r5
1816
1817         loadn32 r1, #40
1818         load32 r3, ini_colun
1819         dec32 r3
1820         load32 r4, num_colun
1821         mov32 r5, r2
1822         dec32 r5
1823         mul32 r1, r1, r5

```

```

1824 nprint :
1825     add32 r5, r1, r3
1826     outchar32 r6,r5
1827     inc32 r3
1828     dec32 r4
1829     jnz32 nprint_
1830
1831     pop32 r5
1832     pop32 r4
1833     pop32 r3
1834     pop32 r1
1835     pop32 r0
1836     rts32
1837
1838 deleta_desloca_bloco:
1839     push32 r0
1840     push32 r1
1841     push32 r2
1842     push32 r3
1843     push32 r4
1844     push32 r5
1845
1846     load32 r1, ini_colun
1847     load32 r2, ini_linha
1848     load32 r3, num_linha
1849     store32 aux_lin, r3
1850     Delet:
1851     load32 r3, aux_lin
1852     dec32 r3
1853     store32 aux_lin,r3
1854     loadn32 r0, #apaga_linha
1855     add32 r4,r3, r0
1856     loadi32 r5,r4
1857     loadn32 r4, #2
1858     cmp32 r5, r4
1859     jeq32 fim_delet
1860     dec32 r4
1861     cmp32 r5,r4
1862     ceq32 deletar_linha
1863     dec32 r2
1864     dec32 r4
1865     cmp32 r3,r4
1866     jne32 Delet
1867
1868     fim_delet:
1869     pop32 r5
1870     pop32 r4
1871     pop32 r3
1872     pop32 r2
1873     pop32 r1
1874     pop32 r0
1875     rts32
1876
1877 deletar_linha:
1878     push32 r0
1879     push32 r1
1880     push32 r3
1881     push32 r4
1882     push32 r5
1883     push32 r6
1884     push32 r7
1885
1886     loadn32 r0, #S1L1
1887     loadn32 r1, #mapa_corL1
1888
1889     loadn32 r4,#' '
1890
1891     call32 ponteiro
1892     load32 r5, posi_ponteiro
1893     loadn32 r6, #0
1894     deletando:
1895     add32 r7,r5,r6
1896     add32 r7,r0,r7

```



```

1897     storei32 r7,r4 ; apaga o bloco da memória
1898
1899     add32 r7,r5,r6
1900     add32 r7,r1,r7
1901     storei32 r7,r4 ; apaga a cor da memória de cores
1902
1903     mov32 r3,r6
1904     loadn32 r6, #' ';
1905     call32 print_,apaga linha da tela
1906     mov32 r6,r3
1907     load32 r3, num_colun
1908     inc32 r6
1909     sub32 r7,r3,r6
1910
1911     jnz32 deletando
1912
1913     call32 desloca_linha
1914     inc32 r2
1915     load32 r7, aux_lin
1916     inc32 r7
1917     store32 aux_lin, r7
1918
1919     pop32 r7
1920     pop32 r6
1921     pop32 r5
1922     pop32 r4
1923     pop32 r3
1924     pop32 r1
1925     pop32 r0
1926     rts32
1927
1928
1929 desloca_linha:
1930     push32 r0
1931     push32 r1
1932     push32 r3
1933     push32 r4
1934     push32 r5
1935     push32 r6
1936     push32 r7
1937
1938     mov32 r0, r2
1939     loadn32 r1, #apaga_linha
1940
1941     desloca:
1942         loadn32 r3, #5
1943         dec32 r0
1944         sub32 r4, r0, r3
1945         add32 r5,r1,r4
1946         loadi32 r5,r5
1947         loadn32 r3, #2
1948         cmp32 r5, r3
1949         jeq32 fim_deslocaLinha
1950         call32 deslocando
1951         loadn32 r7,#0
1952         cmp32 r4,r7
1953         jne32 desloca
1954
1955     fim_deslocaLinha:
1956         mov32 r0,r2
1957         loadn32 r3,#5
1958     fim_deslo1:
1959 ;-----rotina de deslocamento no vetor de verificações-----
1960     sub32 r4, r0, r3
1961     mov32 r5,r4
1962     dec32 r5
1963     add32 r5, r1, r5
1964     loadi32 r6, r5
1965     add32 r7,r1,r4
1966     storei32 r7,r6
1967     dec32 r0
1968     cmp32 r0,r3
1969     jgr32 fim_deslo1

```

```

1970                sub32 r4, r0, r3
1971                add32 r5,r1,r4
1972                storei32 r5, r4
1973                pop32 r7
1974                pop32 r6
1975                pop32 r5
1976                pop32 r4
1977                pop32 r3
1978                pop32 r1
1979                pop32 r0
1980                rts32
1981
1982
1983 deslocando:
1984                push32 r1
1985                push32 r2
1986                push32 r3
1987                push32 r4
1988                push32 r5
1989                push32 r6
1990
1991                loadn32 r1, #S1L1
1992                loadn32 r3, #0
1993                load32 r4, bloco
1994                loadn32 r6, #mapa_corL1
1995
1996                mov32 r2, r0
1997                call32 ponteiro
1998                load32 r2, posi_ponteiro
1999 deslocamento:
2000                add32 r5,r1, r2
2001                loadi32 r7, r5
2002                cmp32 r7,r4
2003                ceq32 derruba
2004                cmp32 r7,r4
2005                ceq32 print_cor
2006                inc32 r2
2007                inc32 r3
2008                loadn32 r5,#11
2009                cmp32 r3,r5
2010                jle32 deslocamento
2011
2012                pop32 r6
2013                pop32 r5
2014                pop32 r4
2015                pop32 r3
2016                pop32 r2
2017                pop32 r1
2018
2019                rts32
2020
2021 derruba:
2022
2023                push32 r0
2024                push32 r1
2025                push32 r3
2026                push32 r7
2027
2028                loadn32 r0, #41
2029                loadn32 r3, #' '
2030
2031                storei32 r5,r3;Desloca bloco na memória para baixo
2032                add32 r1,r5,r0
2033                storei32 r1,r4
2034
2035                add32 r1,r2,r6; desloca cor no mapa de cor para baixo
2036                loadi32 r7,r1
2037                store32 num_cor,r7
2038                storei32 r1,r3
2039                add32 r1, r1,r0
2040                storei32 r1,r7
2041
2042                pop32 r7

```

```

2043         pop32 r3
2044         pop32 r1
2045         pop32 r0
2046
2047         rts32
2048
2049
2050     print_cor:
2051         push32 r1
2052         push32 r2
2053         push32 r4
2054         push32 r5
2055         push32 r6
2056
2057         loadn32 r1, #40
2058         load32 r2, ini_colun
2059         loadn32 r5, #', '
2060         load32 r6, bloco
2061
2062         dec32 r2
2063
2064         mov32 r4, r0
2065         dec32 r4
2066         mul32 r4, r4, r1
2067         add32 r2, r4, r2
2068         add32 r2, r2, r3
2069         outchar32 r5, r2
2070         call32 select_cor
2071         load32 r4, cor_inst
2072         add32 r6, r4, r6
2073         add32 r2, r2, r1
2074         outchar32 r6, r2
2075
2076         pop32 r6
2077         pop32 r5
2078         pop32 r4
2079         pop32 r2
2080         pop32 r1
2081         rts32
2082
2083     select_cor:
2084         push32 r0
2085         push32 r1
2086
2087         load32 r0, num_cor
2088
2089         loadn32 r1, #1
2090         cmp32 r0, r1
2091         jeq32 amarelo
2092
2093         loadn32 r1, #2
2094         cmp32 r0, r1
2095         jeq32 vermelho
2096
2097         loadn32 r1, #3
2098         cmp32 r0, r1
2099         jeq32 verde
2100
2101         loadn32 r1, #4
2102         cmp32 r0, r1
2103         jeq32 azul
2104
2105         loadn32 r1, #5
2106         cmp32 r0, r1
2107         jeq32 roxo
2108
2109         jmp32 fim_cor
2110
2111     amarelo:
2112         loadn32 r0, #2816
2113         store32 cor_inst, r0
2114         jmp32 fim_cor
2115

```

```

2116   vermelho:
2117   loadn32 r0, #2304
2118   store32 cor_inst,r0
2119   jmp32 fim_cor
2120
2121   verde:
2122   loadn32 r0, #512
2123   store32 cor_inst,r0
2124   jmp32 fim_cor
2125
2126   azul:
2127   loadn32 r0, #1024
2128   store32 cor_inst,r0
2129   jmp32 fim_cor
2130
2131
2132   roxo:
2133   loadn32 r0, #1280
2134   store32 cor_inst,r0
2135   jmp32 fim_cor
2136
2137   fim_cor:
2138
2139       pop32 r1
2140       pop32 r0
2141
2142   rts32
2143
2144   select_cor_:
2145       push32 r0
2146       push32 r1
2147
2148   load32 r0,cor_inst
2149
2150   loadn32 r1,#2816
2151   cmp32 r0,r1
2152   jeq32 amarelo_
2153
2154   loadn32 r1,#2304
2155   cmp32 r0,r1
2156   jeq32 vermelho_
2157
2158   loadn32 r1,#512
2159   cmp32 r0,r1
2160   jeq32 verde_
2161
2162   loadn32 r1,#1024
2163   cmp32 r0,r1
2164   jeq32 azul_
2165
2166   loadn32 r1,#1280
2167   cmp32 r0,r1
2168   jeq32 roxo_
2169
2170   jmp32 fim_cor_
2171
2172   amarelo_:
2173   loadn32 r0, #1
2174   store32 num_cor,r0
2175   jmp32 fim_cor_
2176
2177   vermelho_:
2178   loadn32 r0, #2
2179   store32 num_cor,r0
2180   jmp32 fim_cor_
2181
2182   verde_:
2183   loadn32 r0, #3
2184   store32 num_cor,r0
2185   jmp32 fim_cor_
2186
2187   azul_:
2188   loadn32 r0, #4

```

```

2189 store32 num_cor,r0
2190 jmp32 fim_cor_
2191
2192 roxo:
2193 loadn32 r0, #5
2194 store32 num_cor,r0
2195 jmp32 fim_cor_
2196
2197 fim_cor_:
2198
2199     pop32 r1
2200     pop32 r0
2201
2202 rts32
2203
2204 verifi_gameOver:
2205     push32 r0
2206     push32 r1
2207     push32 r2
2208     push32 r3
2209     push32 r4
2210     push32 r5
2211     push32 r6
2212     push32 r7
2213
2214     loadn32 r0,#S1L1
2215     load32 r1, ini_colun
2216     loadn32 r2,#4
2217     load32 r3,num_colun
2218     load32 r4, bloco
2219     loadn32 r7, #0
2220
2221     call32 ponteiro
2222     load32 r2,posi_ponteiro
2223
2224 loop_over:
2225     add32 r5,r0,r2
2226     loadi32 r6,r5
2227     cmp32 r6,r4
2228     jeq32 game_over
2229     inc32 r2
2230     inc32 r7
2231     cmp32 r7,r3
2232     jle32 loop_over
2233     pop32 r7
2234     pop32 r6
2235     pop32 r5
2236     pop32 r4
2237     pop32 r3
2238     pop32 r2
2239     pop32 r1
2240     pop32 r0
2241 rts32
2242
2243 game_over:
2244
2245     push32 fr
2246     push32 r0
2247     push32 r1
2248     push32 r2
2249
2250     call32 ClearScreen ; Limpa a Tela
2251     loadn32 r1, #S2L1 ; tela inicial
2252     loadn32 r2, #0 ; cor da tela
2253     loadn32 r0, #0; Posicao na tela onde a mensagem sera' escrita
2254     call32 PrintScreen
2255     call32 getchar
2256     pop32 r2
2257     pop32 r1
2258     pop32 r0
2259     pop32 fr
2260     jmp32 main
2261

```

```

2262 getchar:
2263     push32 fr
2264     push32 r0
2265     push32 r1
2266
2267     loadn32 r1, #13
2268
2269     getchar_loop:
2270         inchar32 r0
2271         cmp32 r0, r1
2272         jne32 getchar_loop
2273
2274
2275     pop32 r1
2276     pop32 r0
2277     pop32 fr
2278     rts32
2279
2280     limpa_memorias:
2281         push32 r0
2282         push32 r1
2283         push32 r2
2284         push32 r5
2285
2286     loadn32 r0, #1
2287     load32 r2, ini_linha
2288
2289
2290     limpa_linhas:
2291     call32 ponteiro
2292     load32 r5, posi_ponteiro
2293     call32 limpa_colunas
2294     dec32 r2
2295     cmp32 r2, r0
2296     jeg32 limpa_linhas
2297
2298         pop32 r5
2299         pop32 r2
2300         pop32 r1
2301         pop32 r0
2302     rts32
2303
2304     limpa_colunas:
2305         push32 r0
2306         push32 r1
2307         push32 r2
2308         push32 r3
2309         push32 r4
2310         push32 r6
2311
2312     load32 r0, num_colun
2313     loadn32 r1, #SIL1
2314     loadn32 r2, #mapa_corL1
2315     loadn32 r3, #' '
2316     loadn32 r4, #0
2317
2318     loop_limpa_colun:
2319     add32 r6, r5, r1
2320     add32 r6, r6, r4
2321     storei32 r6, r3
2322     add32 r6, r5, r2
2323     add32 r6, r6, r4
2324     storei32 r6, r3
2325     inc32 r4
2326     cmp32 r4, r0
2327     jle32 loop_limpa_colun
2328
2329         pop32 r6
2330         pop32 r4
2331         pop32 r3
2332         pop32 r2
2333         pop32 r1
2334         pop32 r0

```

```

2335     rts32
2336
2337
2338     print_borda:
2339         push32 r0
2340         push32 r1
2341         push32 r2
2342         push32 r3
2343         push32 r4
2344         push32 r6
2345         push32 r7
2346
2347     loadn32 r0, #4
2348     load32 r1, ini_colun
2349     dec32 r1
2350     load32 r2, num_colun
2351
2352     loadn32 r3, #40
2353     loadn32 r4, #'('
2354     loadn32 r6, #0
2355     mul32 r5,r0,r3
2356     add32 r5,r5,r1
2357
2358     loop_print_borda:
2359     add32 r7,r5,r6
2360     outchar32 r4,r7
2361     inc32 r6
2362     cmp32 r6, r2
2363     jle32 loop_print_borda
2364
2365         pop32 r7
2366         pop32 r6
2367         pop32 r4
2368         pop32 r3
2369         pop32 r2
2370         pop32 r1
2371         pop32 r0
2372     rts32
2373
2374     verifi_apagaIni:
2375
2376         push32 r0
2377         push32 r1
2378         push32 r2
2379         push32 r3
2380         push32 r4
2381         push32 r6
2382         push32 r7
2383
2384     loadn32 r0, #0
2385     store32 flag_impr1, r0
2386
2387     loadn32 r0, #0
2388     store32 flag_impr2, r0
2389
2390     loadn32 r0, #0
2391     store32 flag_impr3, r0
2392
2393     loadn32 r0, #0
2394     store32 flag_impr4, r0
2395
2396     loadn32 r0, #205
2397
2398     load32 r2, pos1
2399     cmp32 r2,r0
2400     cgr32 imprime_pos1
2401
2402     load32 r2, pos2
2403     cmp32 r2,r0
2404     cgr32 imprime_pos2
2405
2406     load32 r2, pos3
2407     cmp32 r2,r0

```

```

2408 cgr32 imprime_pos3
2409
2410 load32 r2, pos4
2411 cmp32 r2,r0
2412 cgr32 imprime_pos4
2413
2414     pop32 r7
2415     pop32 r6
2416     pop32 r4
2417     pop32 r3
2418     pop32 r2
2419     pop32 r1
2420     pop32 r0
2421 rts32
2422
2423 imprime_pos1:
2424     push32 r0
2425 loadn32 r0, #1
2426 store32 flag_impr1, r0
2427     pop32 r0
2428 rts32
2429
2430 imprime_pos2:
2431     push32 r0
2432 loadn32 r0, #1
2433 store32 flag_impr2, r0
2434     pop32 r0
2435 rts32
2436
2437 imprime_pos3:
2438     push32 r0
2439 loadn32 r0, #1
2440 store32 flag_impr3, r0
2441     pop32 r0
2442 rts32
2443
2444 imprime_pos4:
2445     push32 r0
2446 loadn32 r0, #1
2447 store32 flag_impr4, r0
2448     pop32 r0
2449 rts32
2450
2451 apaga_pos1:
2452     push32 r1
2453 loadn32 r1, #' '
2454 outchar32 r1, r2; apaga pixel do bloco
2455
2456     pop32 r1
2457 rts32
2458
2459 apaga_pos2:
2460     push32 r1
2461 loadn32 r1, #' '
2462 outchar32 r1, r3; apaga pixel do bloco
2463
2464     pop32 r1
2465 rts32
2466
2467 apaga_pos3:
2468     push32 r1
2469 loadn32 r1, #' '
2470 outchar32 r1, r4; apaga pixel do bloco
2471
2472     pop32 r1
2473 rts32
2474
2475 apaga_pos4:
2476     push32 r1
2477 loadn32 r1, #' '
2478 outchar32 r1, r5; apaga pixel do bloco
2479
2480     pop32 r1

```



```

2481 rts32
2482
2483
2484 impr_pos1:
2485     push32 r0
2486     load32 r0, pos1
2487     outchar32 r5, r0 ; imprime pixel bloco
2488     pop32 r0
2489 rts32
2490
2491 impr_pos2:
2492     push32 r0
2493     load32 r0, pos2
2494     outchar32 r5, r0 ; imprime pixel bloco
2495     pop32 r0
2496 rts32
2497
2498 impr_pos3:
2499     push32 r0
2500     load32 r0, pos3
2501     outchar32 r5, r0 ; imprime pixel bloco
2502     pop32 r0
2503 rts32
2504
2505 impr_pos4:
2506     push32 r0
2507     load32 r0, pos4
2508     outchar32 r5, r0 ; imprime pixel bloco
2509     pop32 r0
2510 rts32
2511
2512 print_next:
2513     push32 r0
2514     push32 r1
2515     push32 r2
2516     push32 r3
2517     push32 r4
2518     push32 r5
2519     load32 r0, pos1
2520     load32 r1, pos2
2521     load32 r2, pos3
2522     load32 r3, pos4
2523
2524     load32 r4, cor_atual
2525     load32 r5, bloco
2526
2527     add32 r5, r5, r4
2528
2529     outchar32 r5, r0
2530     outchar32 r5, r1
2531     outchar32 r5, r2
2532     outchar32 r5, r3
2533
2534     pop32 r5
2535     pop32 r4
2536     pop32 r3
2537     pop32 r2
2538     pop32 r1
2539     pop32 r0
2540
2541 rts32
2542
2543 zera_seed:
2544     loadn32 r0, #0
2545 rts32
2546
2547 pula_quatro:
2548     loadn32 r0, #4
2549 rts32
2550
2551 dec32rementa:
2552     dec32 r0
2553 rts32

```

```

2554
2555     clear_next:
2556         push32 r0
2557         push32 r1
2558         push32 r2
2559         push32 r3
2560         push32 r4
2561         push32 r5
2562         push32 r6
2563
2564         loadn32 r1, #226
2565
2566         loadn32 r3, #0
2567         call32 clear_next_linha
2568         loadn32 r3, #40
2569         call32 clear_next_linha
2570         loadn32 r3, #80
2571         call32 clear_next_linha
2572         loadn32 r3, #120
2573         call32 clear_next_linha
2574         loadn32 r3, #160
2575         call32 clear_next_linha
2576
2577         pop32 r6
2578         pop32 r5
2579         pop32 r4
2580         pop32 r3
2581         pop32 r2
2582         pop32 r1
2583         pop32 r0
2584
2585         rts32
2586
2587
2588     clear_next_linha:
2589         push32 r0
2590         push32 r2
2591         push32 r4
2592         push32 r5
2593         push32 r6
2594
2595         loadn32 r4, #0
2596         loadn32 r2, #6
2597         loadn32 r0, #' '
2598         add32 r5,r1,r3
2599
2600     loop_tela:
2601         add32 r6,r5,r4
2602         outchar32 r0,r6
2603         inc32 r4
2604         cmp32 r4,r2
2605         jle32 loop_tela
2606         pop32 r6
2607         pop32 r5
2608         pop32 r4
2609         pop32 r2
2610         pop32 r0
2611     rts32
2612
2613
2614     inc32rementa_seed:
2615         inc32 r0
2616         rts32
2617
2618     pontos:
2619         push32 fr
2620         push32 r0
2621         push32 r1
2622         push32 r2
2623         push32 r3
2624         push32 r4
2625         load32 r0, score_unid
2626         load32 r1, score_dez

```

```

2627 loadn32 r2, #10
2628 loadn32 r4, #5
2629 inc32 r0
2630 cmp32 r0, r2
2631 ceq32 zera_unid
2632 store32 score_unid, r0
2633 store32 score_dez, r1
2634 cmp32 r0, r4
2635
2636
2637 ceq32 aumenta_lv
2638     pop32 r4
2639     pop32 r3
2640     pop32 r2
2641     pop32 r1
2642     pop32 r0
2643     pop32 fr
2644     rts32
2645
2646 zera_unid:
2647 loadn32 r0, #0
2648 inc32 r1
2649 load32 r3, level
2650 loadn32 r5, #3
2651 cmp32 r3, r5
2652 jeq32 fin_zera
2653 inc32 r3
2654 store32 level, r3
2655
2656 fin_zera:
2657
2658
2659 rts32
2660
2661 aumenta_lv:
2662
2663 load32 r3, level
2664 loadn32 r5, #3
2665 cmp32 r3, r5
2666 jeq32 fin_aumentaLv
2667
2668 inc32 r3
2669 store32 level, r3
2670 fin_aumentaLv:
2671 rts32
2672
2673 print_score:
2674     push32 r0
2675     push32 r1
2676     push32 r2
2677
2678 load32 r0, score_unid; imprime a unidade
2679 call32 mapeia_string
2680 loadn32 r0, #631
2681 loadn32 r2, #0
2682 call32 Imprimestr
2683
2684 load32 r0, score_dez; imprime a dezena
2685 call32 mapeia_string
2686 loadn32 r0, #630
2687 loadn32 r2, #0
2688 call32 Imprimestr
2689
2690     pop32 r2
2691     pop32 r1
2692     pop32 r0
2693
2694     rts32
2695
2696 print_nivel:
2697     push32 r0
2698     push32 r1
2699     push32 r2

```

```

2700
2701     load32 r0, level;imprime a unidade
2702     call32 mapeia_string
2703     loadn32 r0, #547
2704     loadn32 r2, #0
2705     call32 Imprimestr
2706
2707
2708         pop32 r2
2709         pop32 r1
2710         pop32 r0
2711
2712         rts32
2713
2714     mapeia_string:
2715
2716
2717     loadn32 r1, #0
2718     cmp32 r0, r1
2719     jeq32 string0
2720
2721     loadn32 r1, #1
2722     cmp32 r0, r1
2723     jeq32 string1
2724
2725     loadn32 r1, #2
2726     cmp32 r0, r1
2727     jeq32 string2
2728
2729     loadn32 r1, #3
2730     cmp32 r0, r1
2731     jeq32 string3
2732
2733     loadn32 r1, #4
2734     cmp32 r0, r1
2735     jeq32 string4
2736
2737     loadn32 r1, #5
2738     cmp32 r0, r1
2739     jeq32 string5
2740
2741     loadn32 r1, #6
2742     cmp32 r0, r1
2743     jeq32 string6
2744
2745     loadn32 r1, #7
2746     cmp32 r0, r1
2747     jeq32 string7
2748
2749     loadn32 r1, #8
2750     cmp32 r0, r1
2751     jeq32 string8
2752
2753     loadn32 r1, #9
2754     cmp32 r0, r1
2755     jeq32 string9
2756
2757     jmp32 finalmapeia
2758
2759     string0:
2760     loadn32 r1, #numero0
2761     jmp32 finalmapeia
2762
2763     string1:
2764     loadn32 r1, #numero1
2765     jmp32 finalmapeia
2766
2767     string2:
2768     loadn32 r1, #numero2
2769     jmp32 finalmapeia
2770
2771     string3:
2772     loadn32 r1, #numero3

```

```
2773     jmp32 finalmapeia
2774
2775     string4:
2776     loadn32 r1, #numero4
2777     jmp32 finalmapeia
2778
2779     string5:
2780     loadn32 r1, #numero5
2781     jmp32 finalmapeia
2782
2783     string6:
2784     loadn32 r1, #numero6
2785     jmp32 finalmapeia
2786
2787     string7:
2788     loadn32 r1, #numero7
2789     jmp32 finalmapeia
2790
2791     string8:
2792     loadn32 r1, #numero8
2793     jmp32 finalmapeia
2794
2795     string9:
2796     loadn32 r1, #numero9
2797     jmp32 finalmapeia
2798
2799     finalmapeia:
2800
2801         rts32
2802
2803
```