Computadores e Videogames

Uma Abordagem Prática das Arquiteturas Clássicas

Sumário

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

fones 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 videgame. O uso de formatos

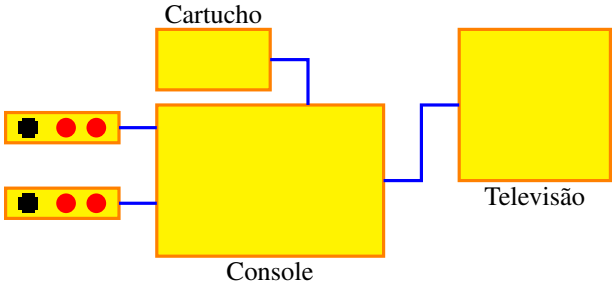


Figura 1: Videogame abstrato

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 sí (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 acontencendo 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 desenhar 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.

Videogames de Primeira Geração

Geralmente considerado como sendo o primeiro videogame, o "Tenis for Two" foi construido 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 tragetórias 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 atraia 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 anos 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 motiva-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.

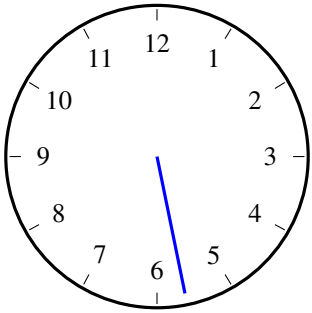


Figura 2: Relógio analógico

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 conseguíssimos, 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

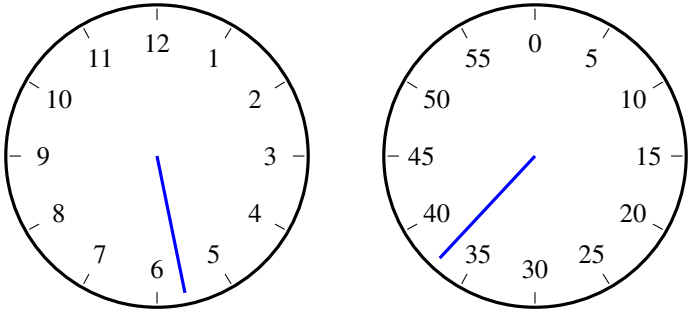


Figura 3: Relógio digital (híbrido)

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 completemente 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 caractersísticas:

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 hidraulica 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 Digital[1] 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`

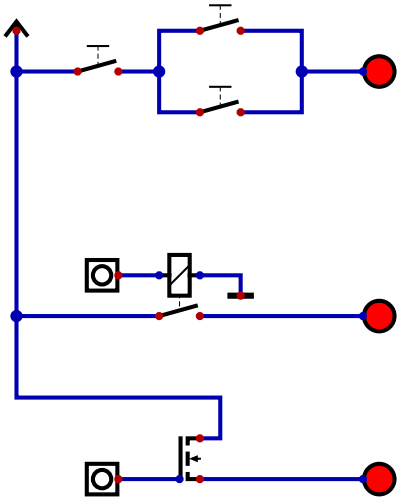


Figura 4: Exemplos do nível de chaves

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 relês 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.

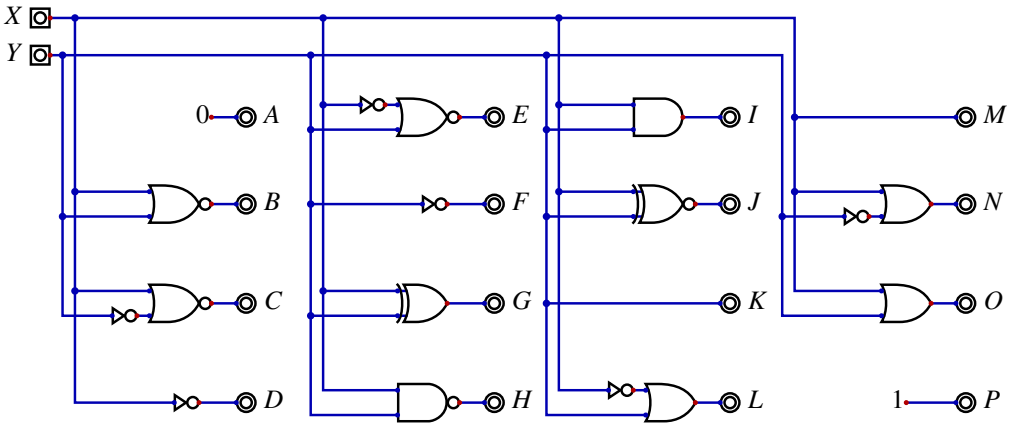
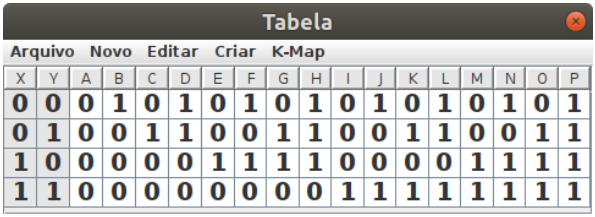


Figura 5: Todos os circuitos combinacionais de duas entradas

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.



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	0	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 AND(X,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

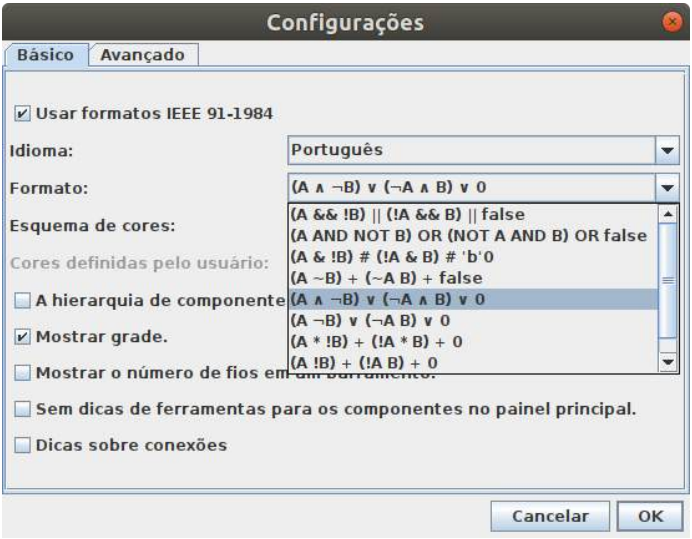


Figura 7: Notações alternativas oferecidas pelo simulador Digital

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 Boleana	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:

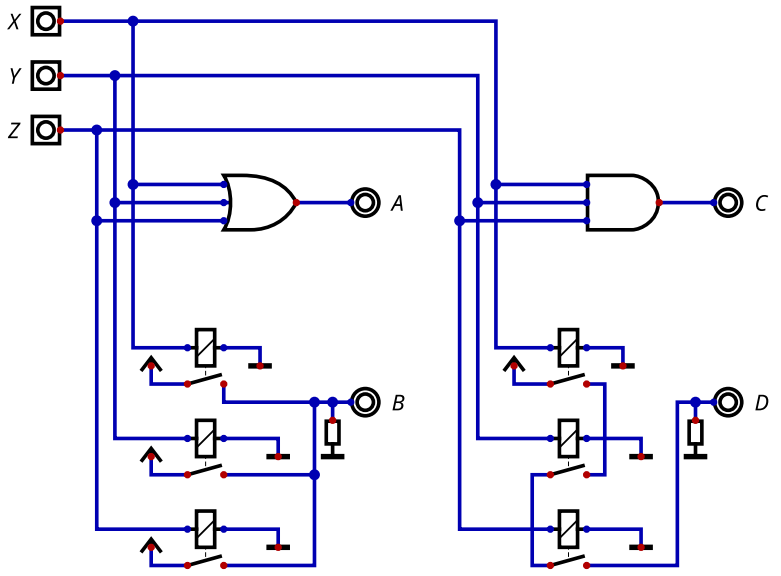
Usando o software Digital para gerar uma tabela verdade para os quatro circuitos mostra que A e B implementam a mesma função, assim como C e D.

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



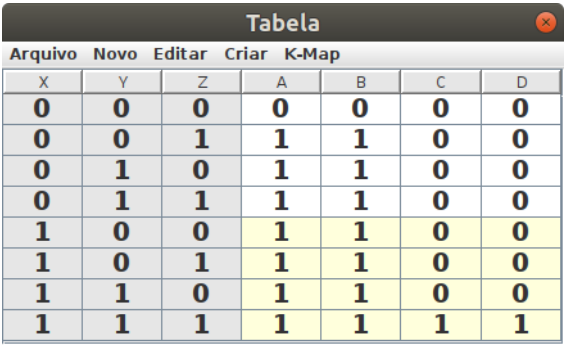
A primeira coluna da Figura 11 apenas repete as duas portas lógicas básicas que vimos na lista completa de todos os 16 circuitos possíveis. A segunda coluna da mesma figura implementa as portas no punho usando apenas portas NOT, AND e OR. Já a terceira coluna faz isso de novo, mas usando apenas portas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

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



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

[3] `https://en.wikipedia.org/wiki/Apollo_Guidance_Computer`

Figura 10: Equações de relé

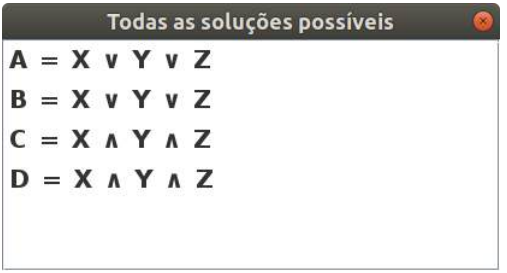


tabela (com algumas entradas possivelmente invertidas) e a saída de todos os AND indo para um OR que gera o resultado final. Como os AND e o OU podem ter qualquer número de entradas, isso funciona para tabelas verdade de qualquer tamanho.

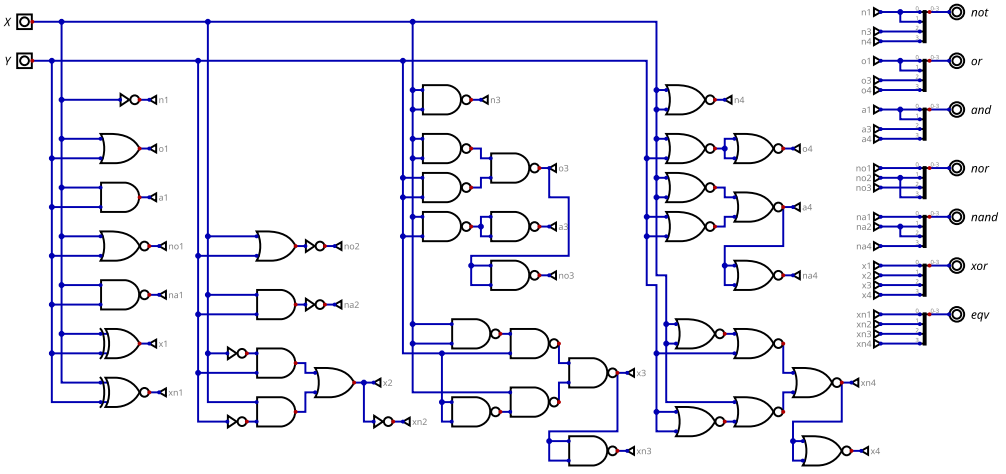
Apesar de não ser prático enumerar todas os circuitos combinacionais de 3 entradas (256), 4 entradas (65536) ou mais, podemos ter certeza que consiguiremos implementar qualquer um deles dado sua tabela verdade usando a soma de produtos.

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

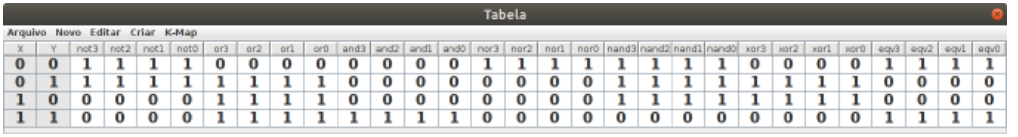
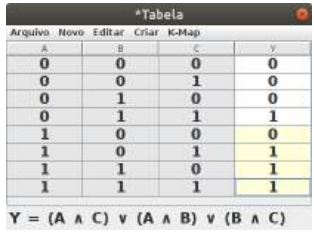


Figura 13: Tabela verdade do circuito de votação

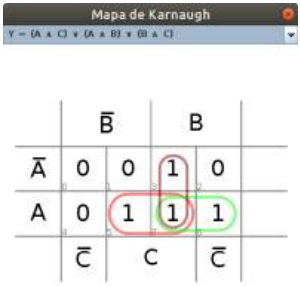


mais de um grupo. Isso reduz o número de portas AND necessárias e também reduz o número de entradas de cada AND. No caso a saída correspondente à entrada 111 pode ser combinada com cada uma das outras saídas um e precisamos de um AND a menos.

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

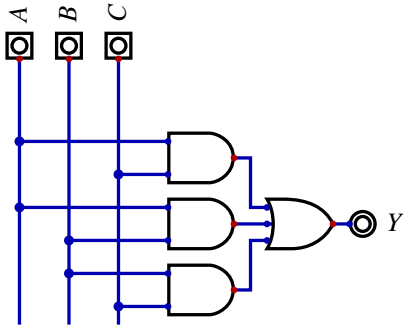


Figura 15: Circuito de votação

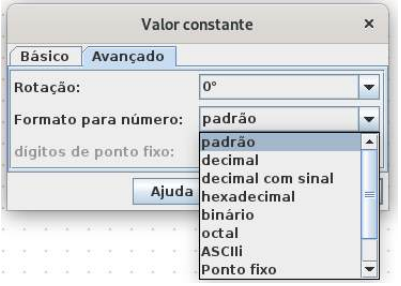
de cada representação numérica. Na fórmula do 729 a base é 10, no caso do hexadecimal a base é 16, para números binários a base é 2 e em octal a base é 8.

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.

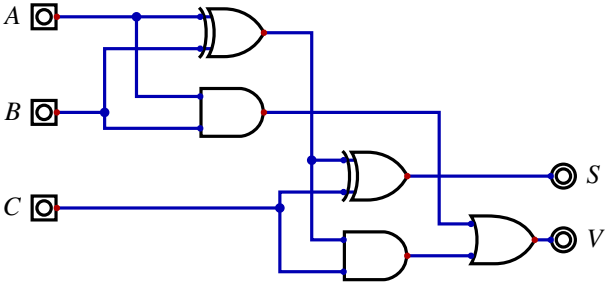


Figura 17: Somador

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 da 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 seguindas. 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 ciruito 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 nos 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 incluiram 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 simplficação 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

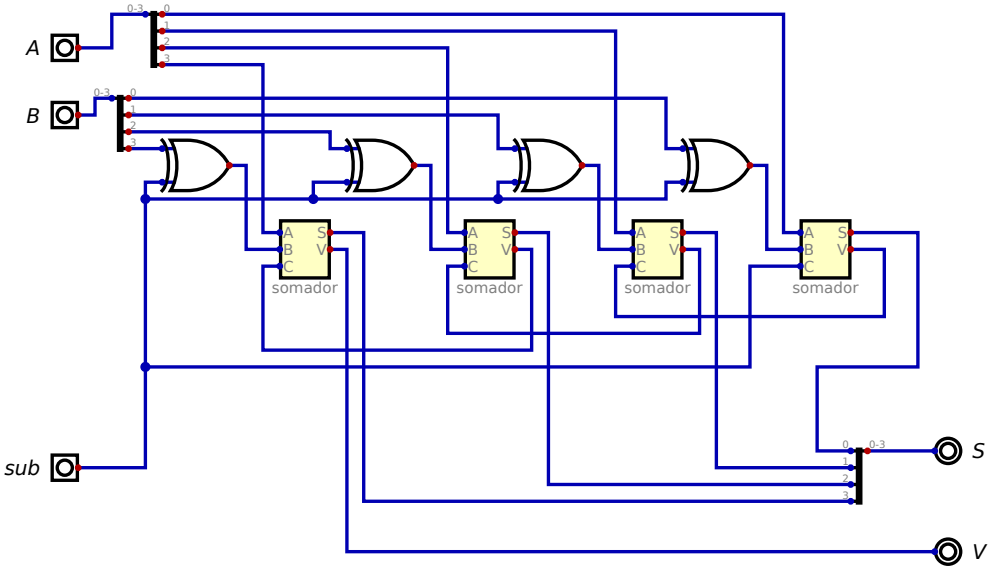


Figura 18: Unidade Aritmética de 4 bits

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 conveção 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.

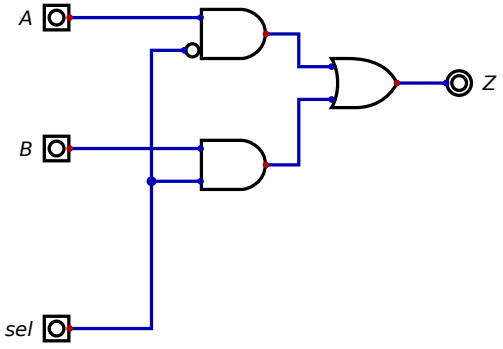


Figura 19: Multiplexador de 2 sinais

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. Podemmos, 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

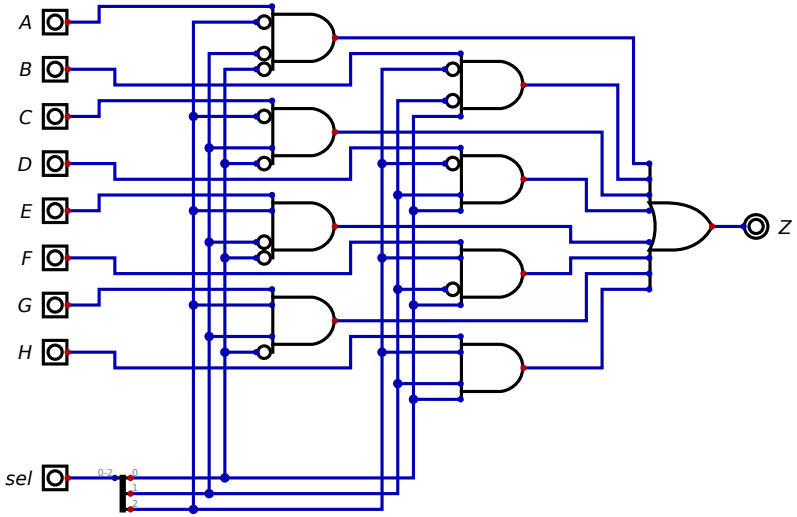


Figura 20: Multiplexador de 8 sinais

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 digito 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

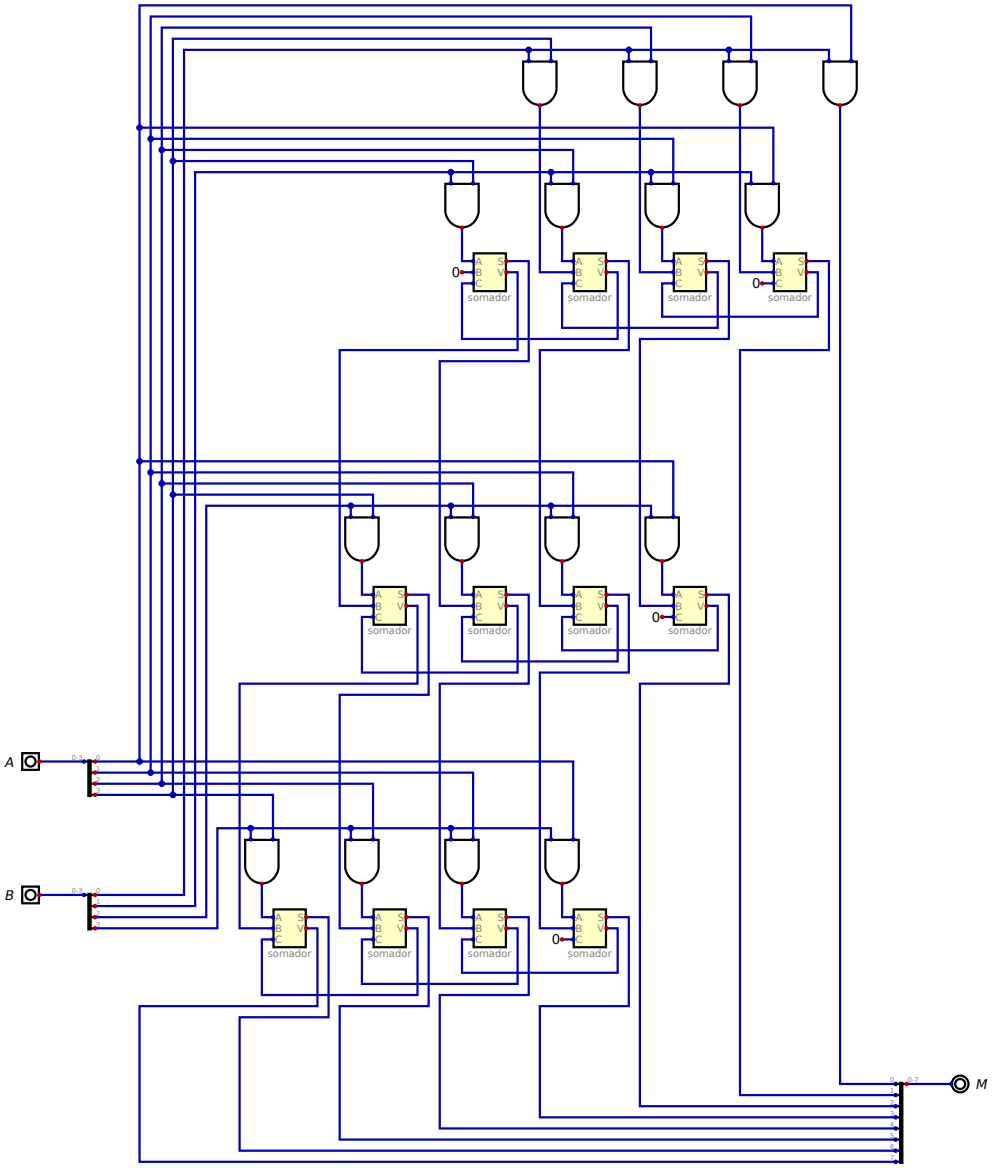


Figura 21: Multiplicador de 4 por 4 bits

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 deslocamente 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.

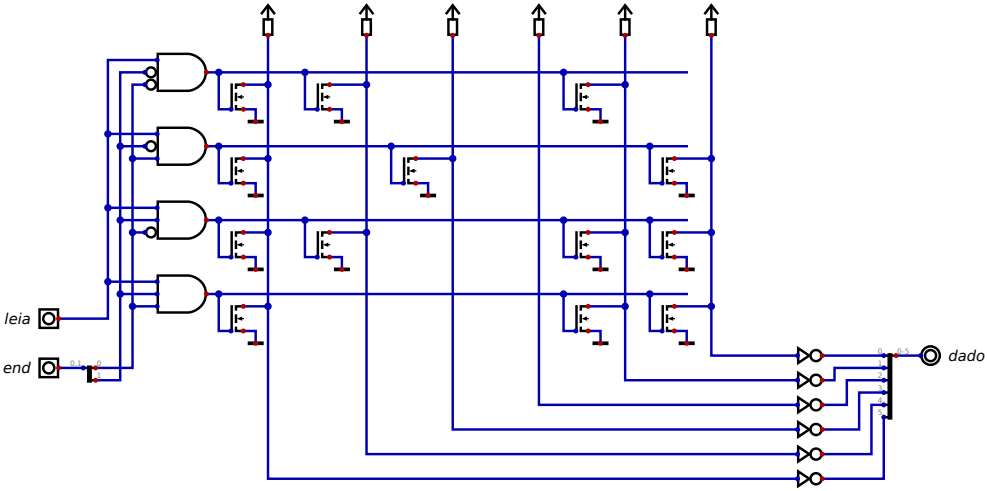


Figura 22: ROM com 4 palavras de 6 bits cada uma

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 superficie 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 transitores 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

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	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	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.

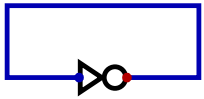


Figura 24: Circuito paradoxal

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 construido 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 construido 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 frequênccia 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, e 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 videgame circuitos assim são os únicos que são feitos para oscilar. Em todo o resto do sistema oscilações serão consideradas erros de projeto.

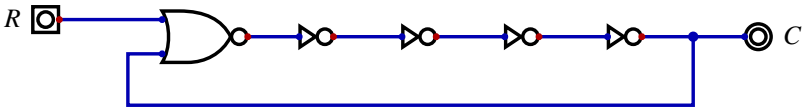


Figura 25: Oscilador em anel com inicialização

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 termos 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.

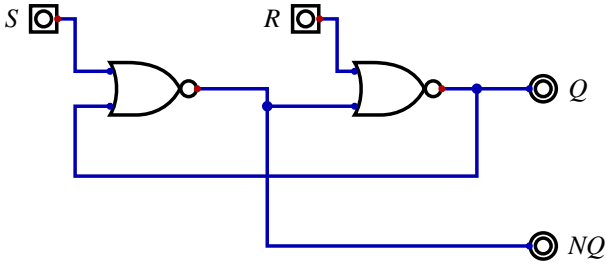


Figura 26: Flip Flop básico

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).

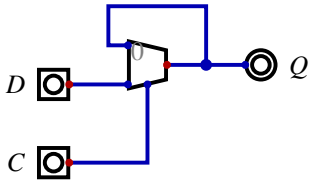


Figura 27: Latch

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.

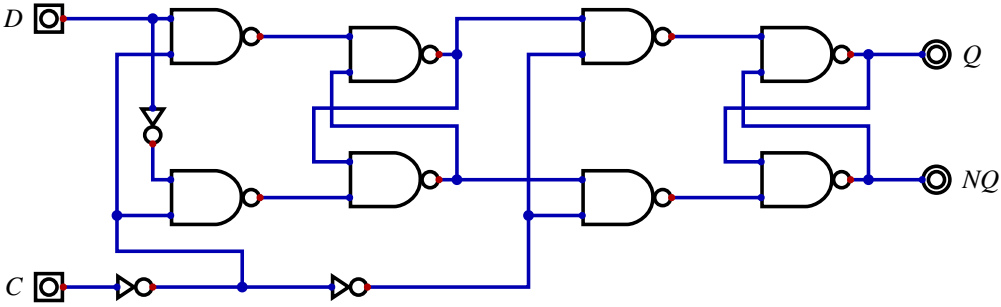


Figura 28: Flip Flop sensível à borda de subida do relógio

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 enderço para selecionar uma palavra para ser alterada ou lida.

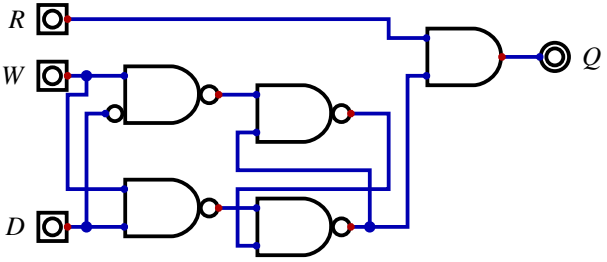


Figura 29: Flip Flop modificado para ser bit de RAM

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

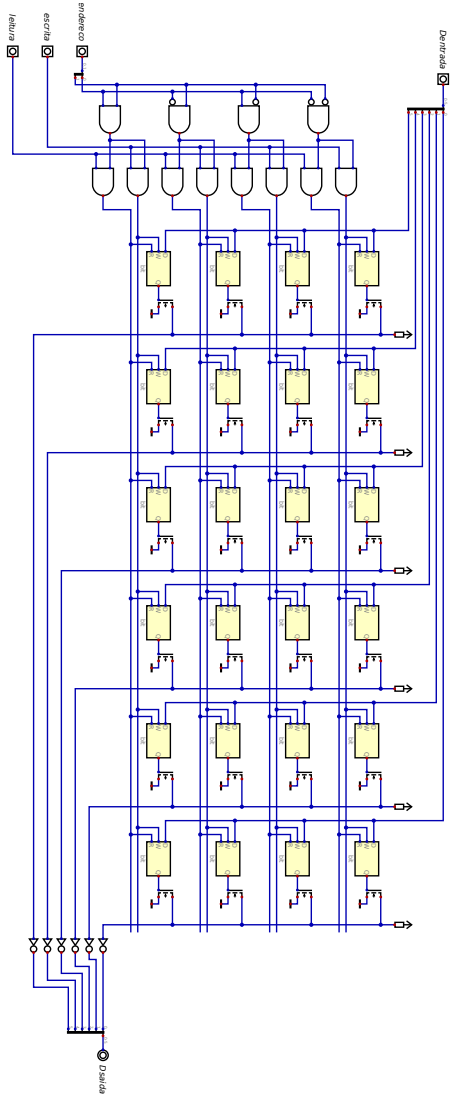


Figura 30: RAM de 4 palavras de 6 bits cada

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 endereco 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 símples. 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.

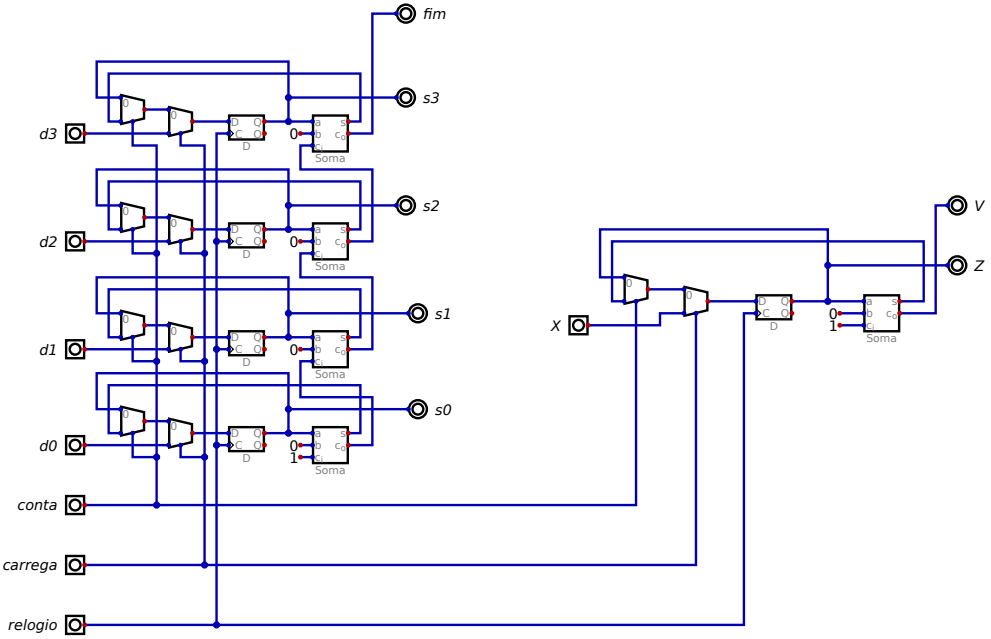


Figura 31: Um contador de 4 bits e outro de 8 bits

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**.

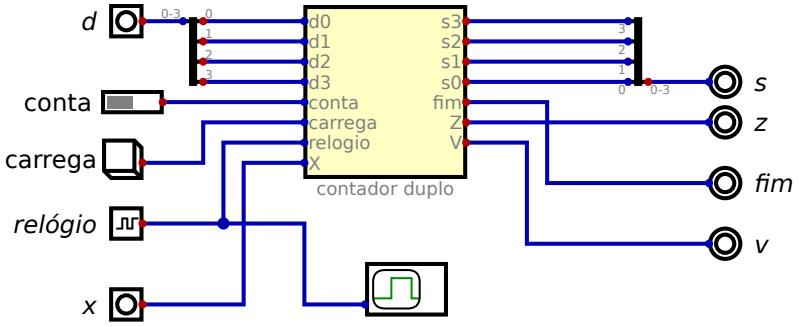


Figura 32: Circuito de teste dos dois contadores

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 dá 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.

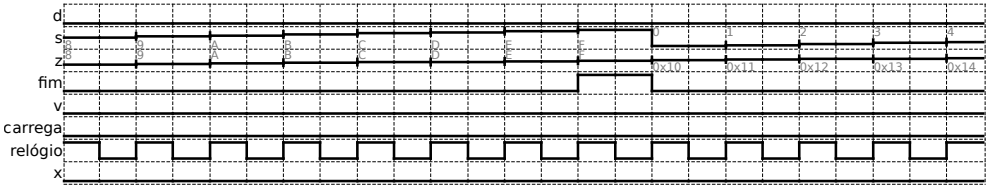


Figura 33: Formas de onda dos dois contadores

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 digitos 9 ou menores então um "0x" é colocado na frente para que o número não seja confundido com decimal. Mas se algum digito 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 ai 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.

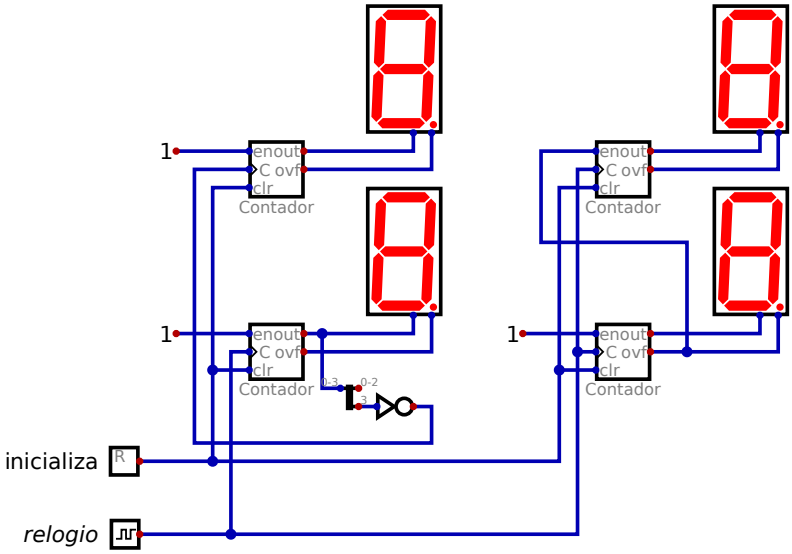


Figura 34: Dois estilos de expandir contadores

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 símples. 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

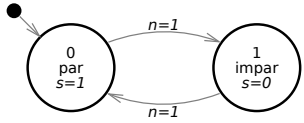


Figura 35: FSM para determinar par ou ímpar

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.

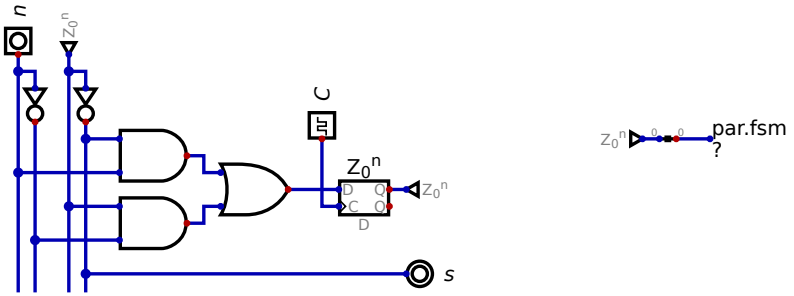


Figura 36: Circuito da FSM para determinar par ou ímpar

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 regjstradores 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 do 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

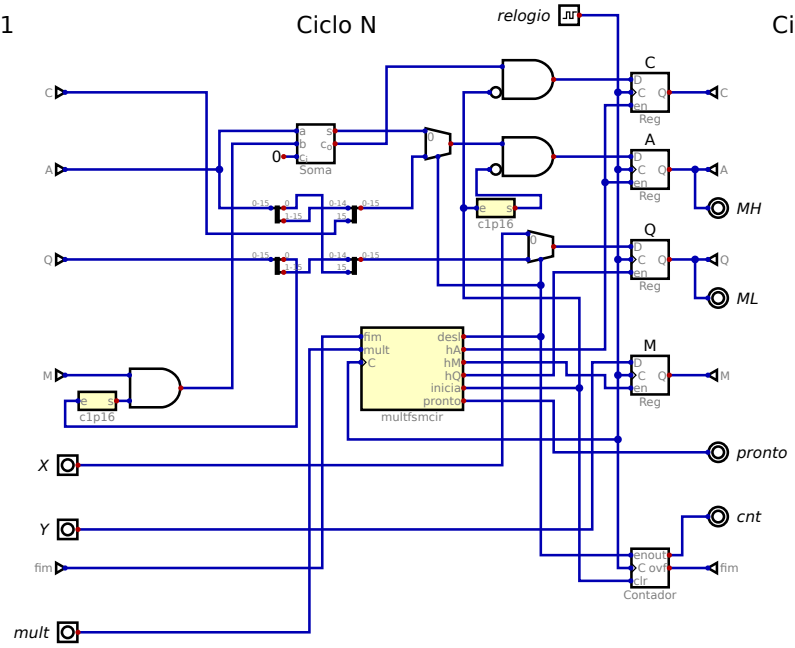


Figura 37: Multiplicador sequencial de 16 por 16 bits

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 c1p16 (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 precisariamos 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 incialmente pelo operando X.

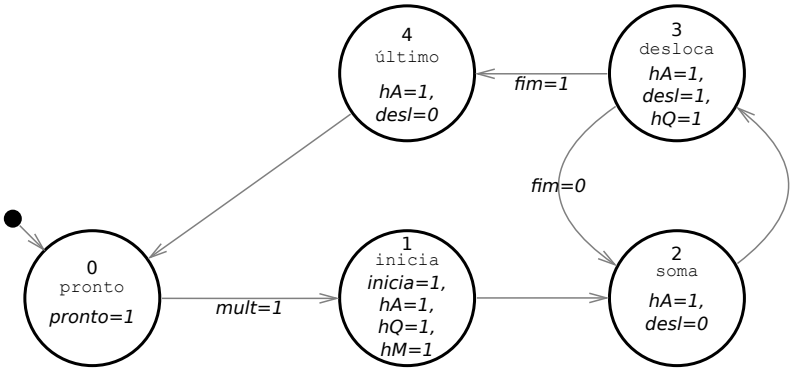


Figura 38: FSM que controla o multiplicador sequencial

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

concluida 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 ai 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 *Uncommited 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.

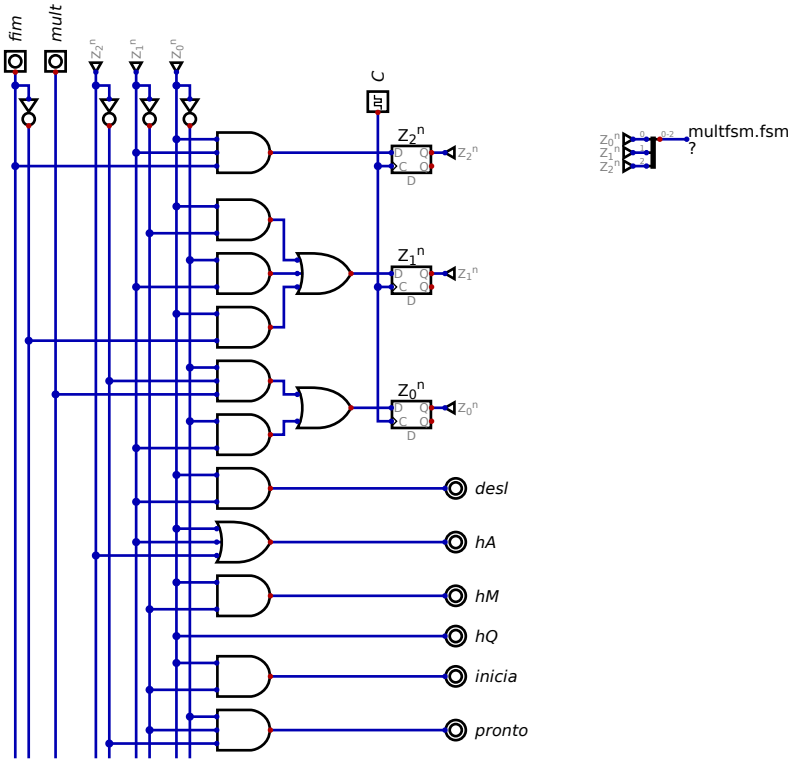


Figura 39: Circuito gerado da FSM de controle do multiplicador

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-

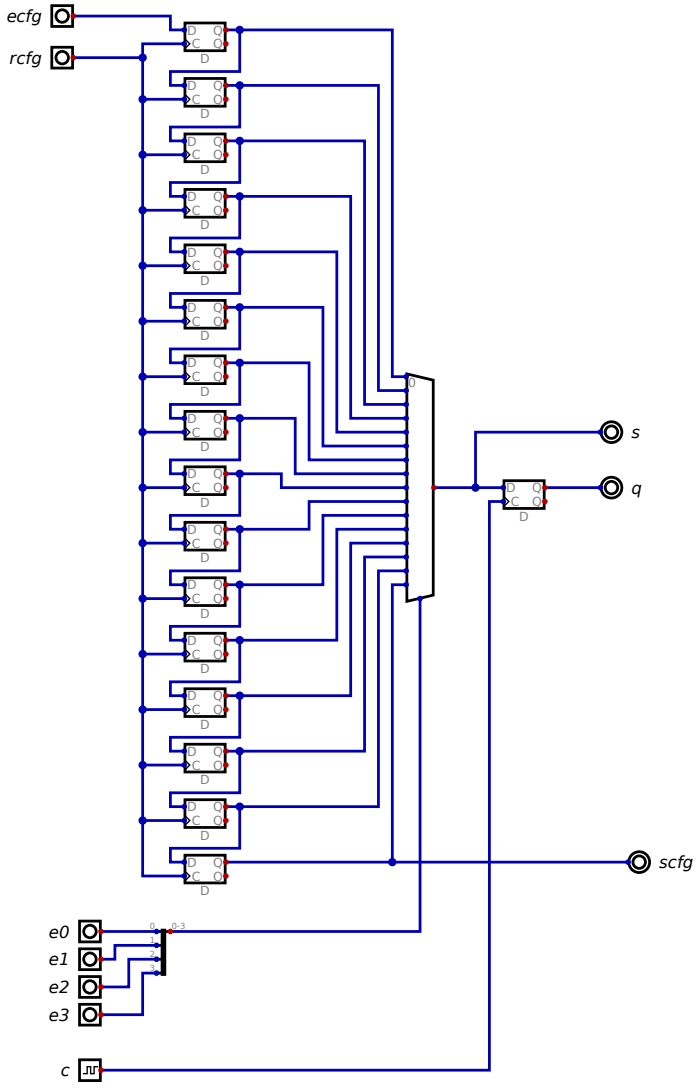


Figura 40: Circuito básico da FPGA (*lookup table*)

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 usam 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 sua 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

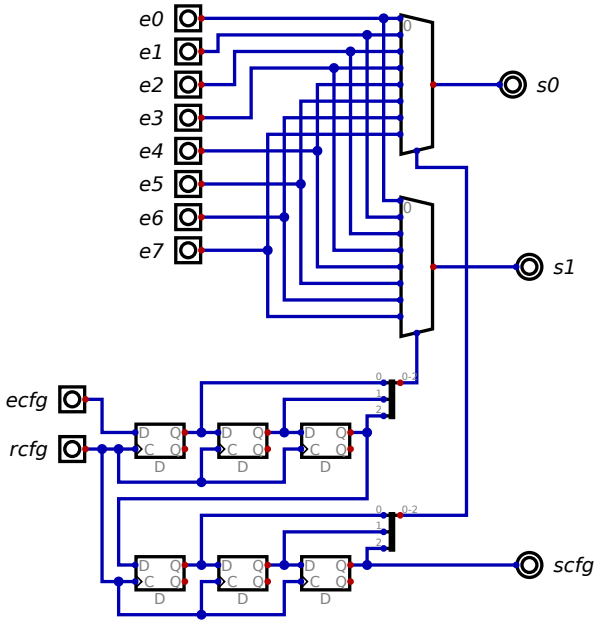


Figura 41: Circuito de roteamento em uma direção

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

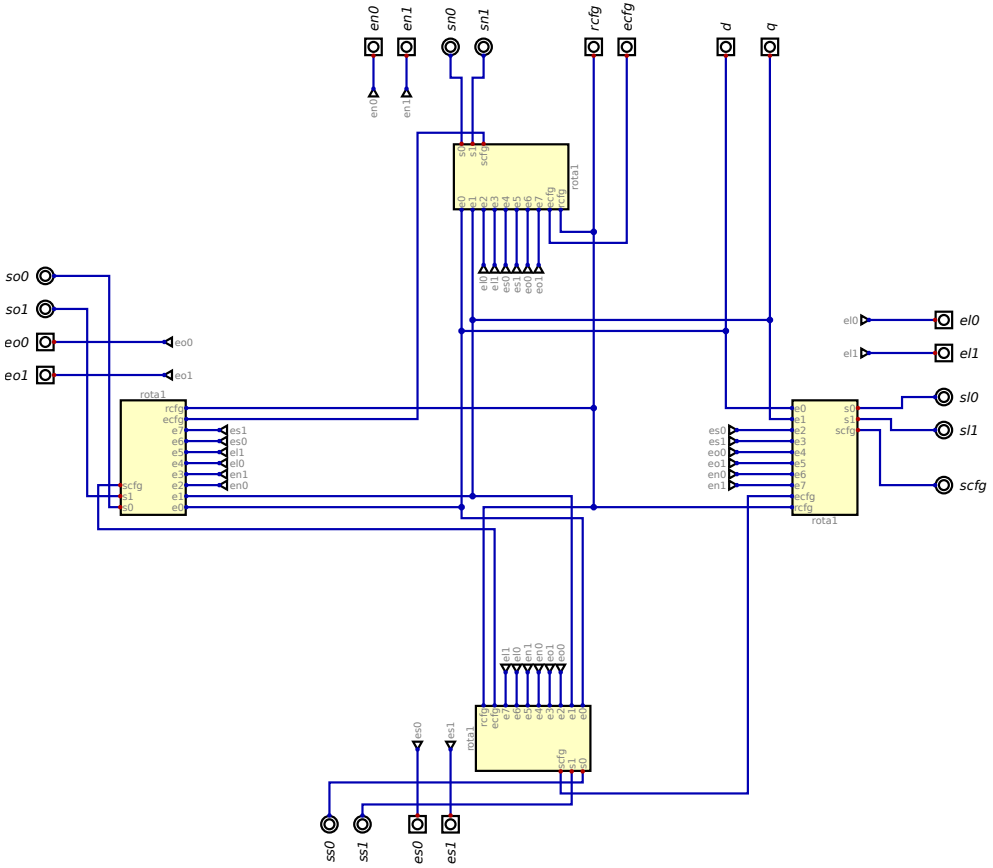


Figura 42: Circuito de roteamento nas quatro direções

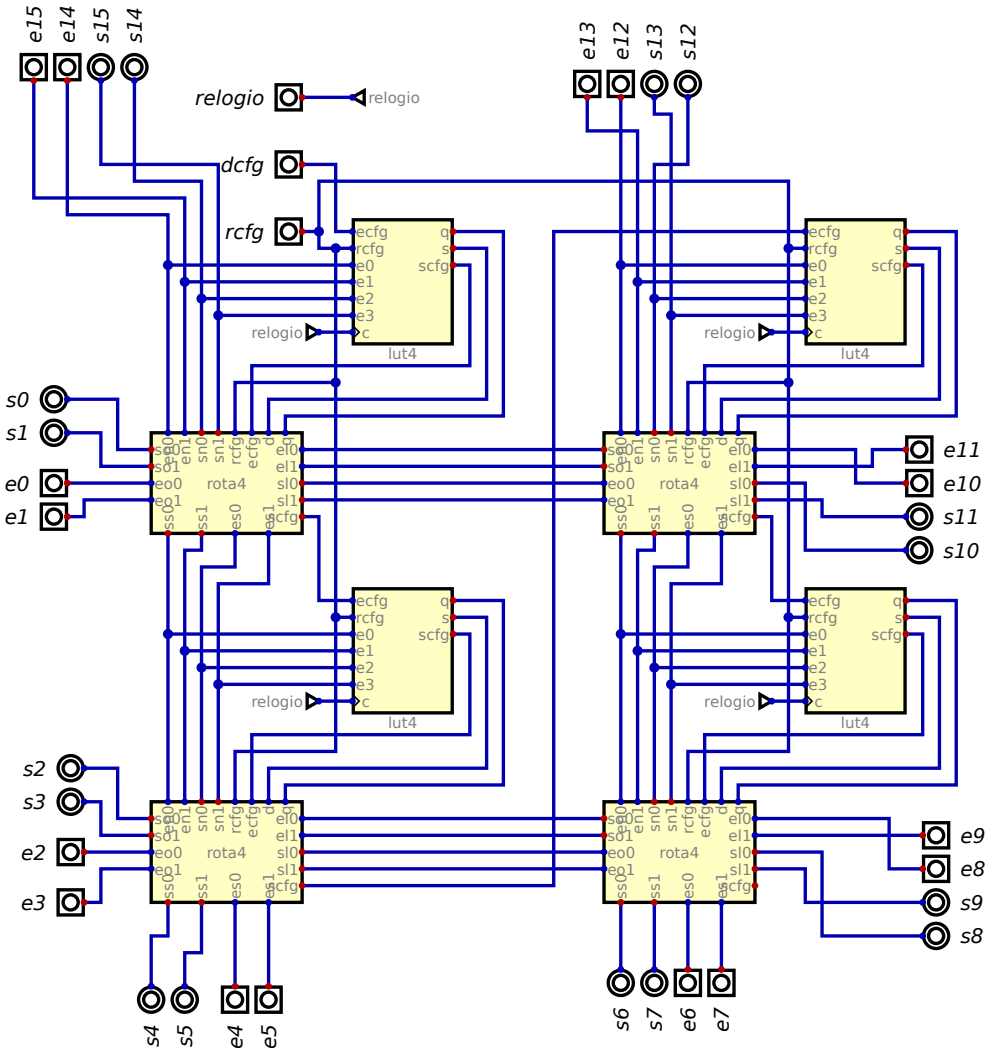


Figura 43: FPGA com 4 elementos (2x2)

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 bit 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 quermos 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 comprir 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 parta 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 "video" 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 shocante 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 denecessária 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

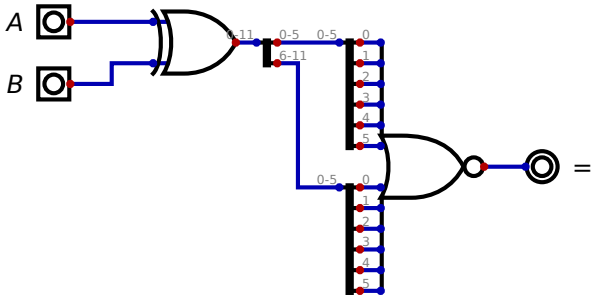


Figura 44: Comparador de igualdade de números inteiros

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 saida 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

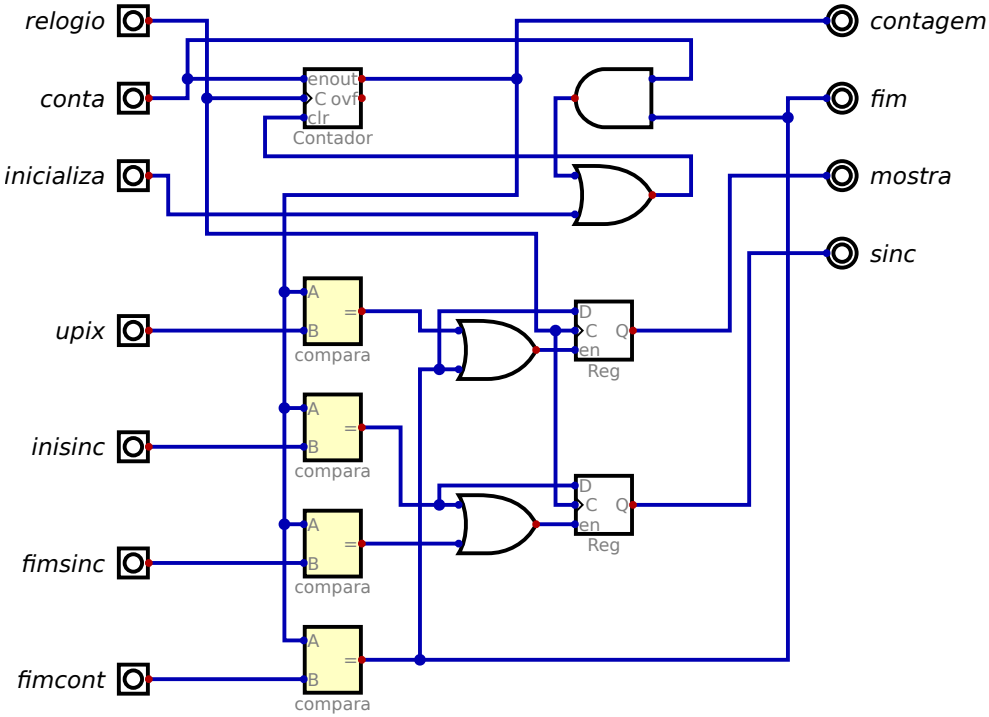


Figura 45: Gera sinais de sincronismo e de habilitação

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, dai **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.

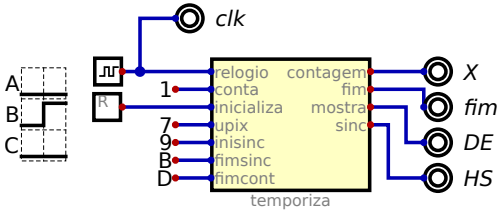


Figura 46: Teste do gerador de sincronismo

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, dai 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.

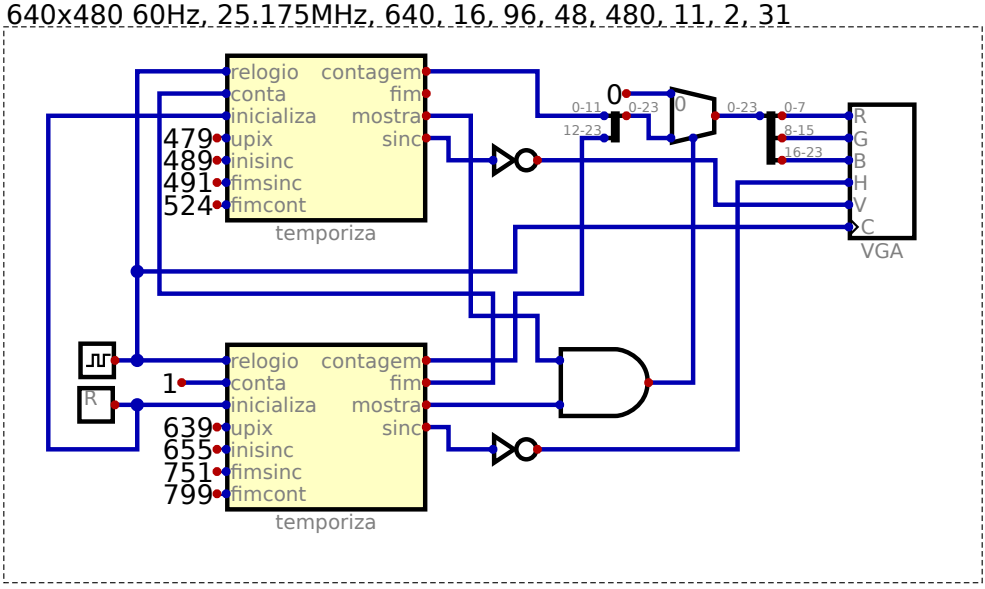
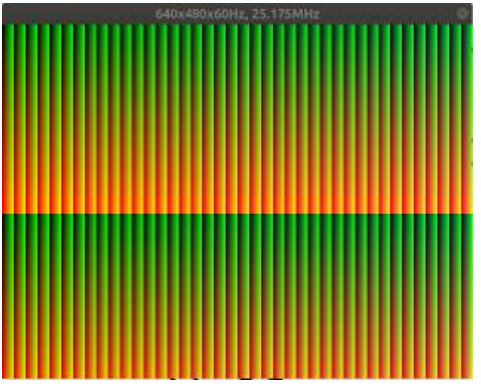


Figura 47: Gera um padrão de cores na tela VGA

Figura 48: Tela do teste de cores



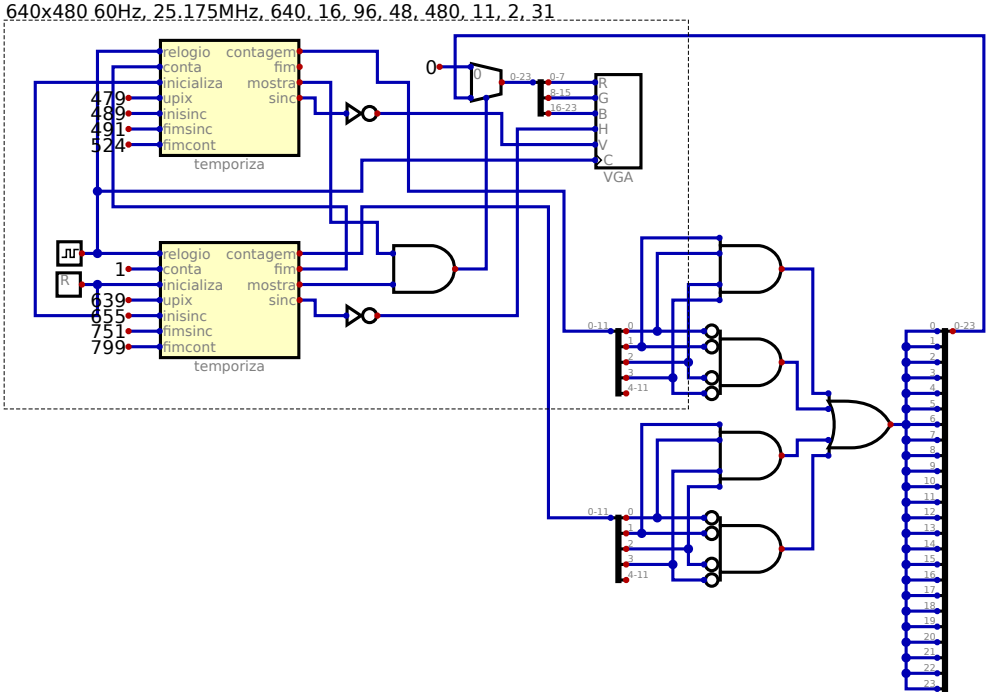
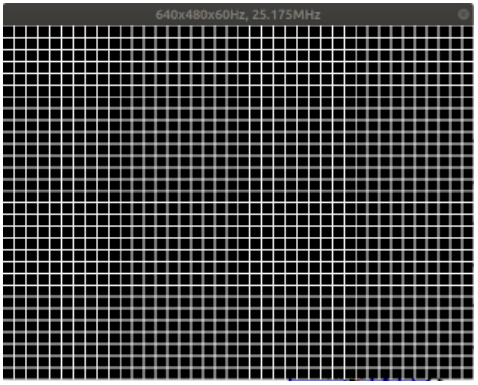


Figura 49: Gera um padrão quadriculado na tela VGA

Figura 50: Tela do teste padrão quadriculado



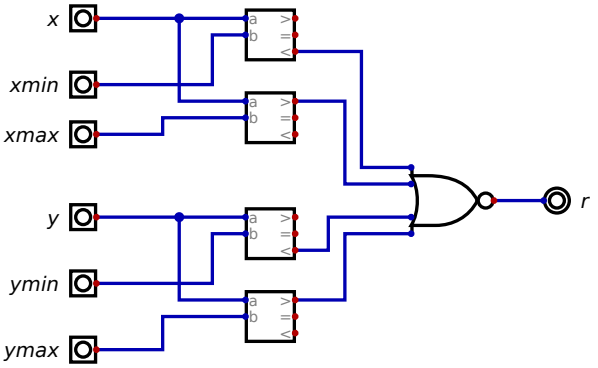


Figura 51: Indica se um ponto está dentro de um retângulo

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 retêngulos 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

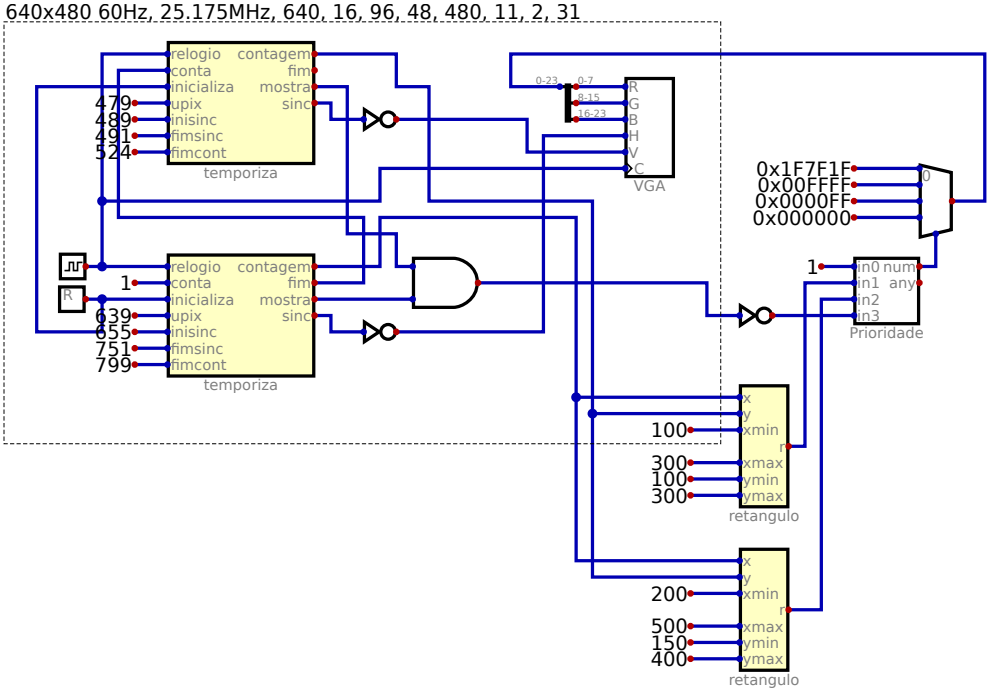
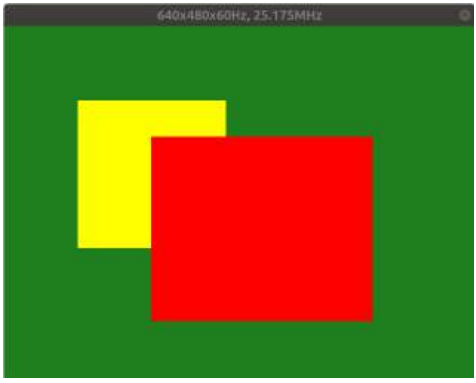


Figura 52: Gera dois retângulos na tela VGA

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 retêngulos (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 of 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 do 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.

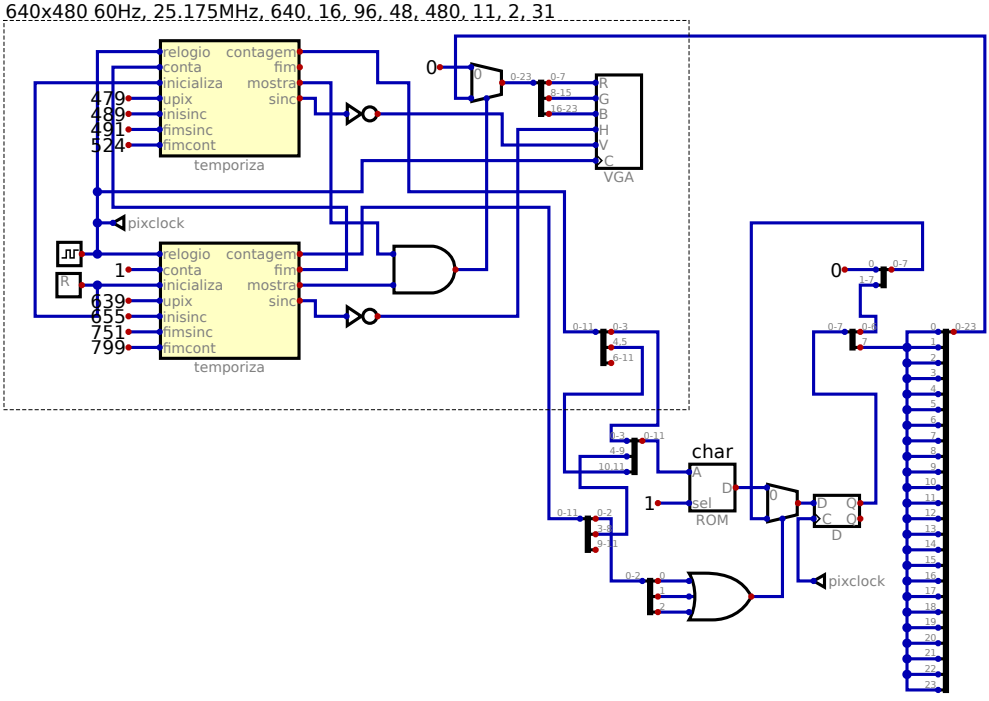
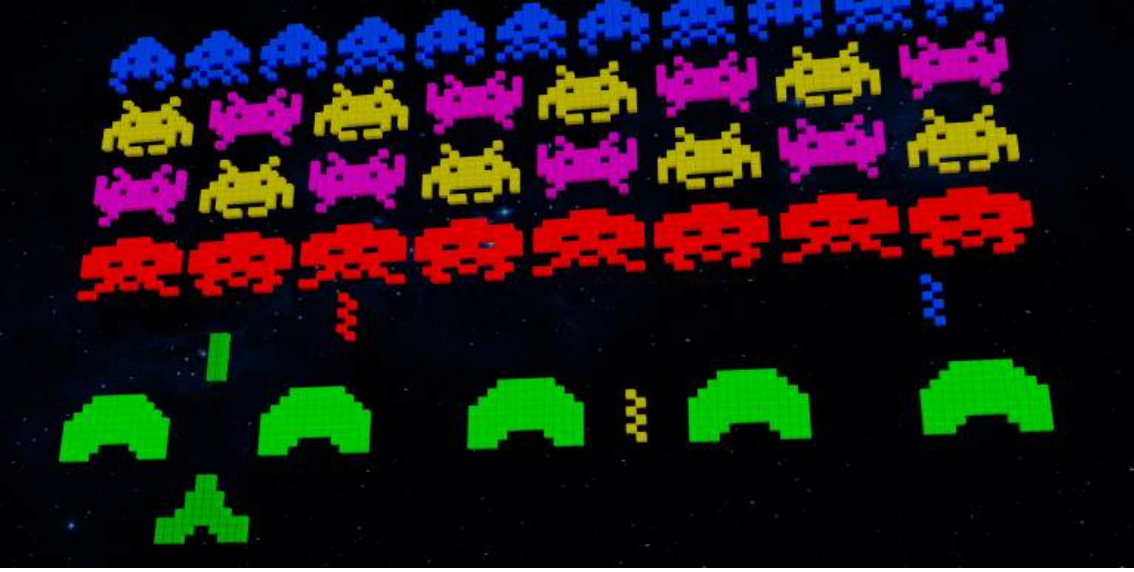


Figura 54: Gera um texto na VGA

Figura 55: Tela do teste de texto



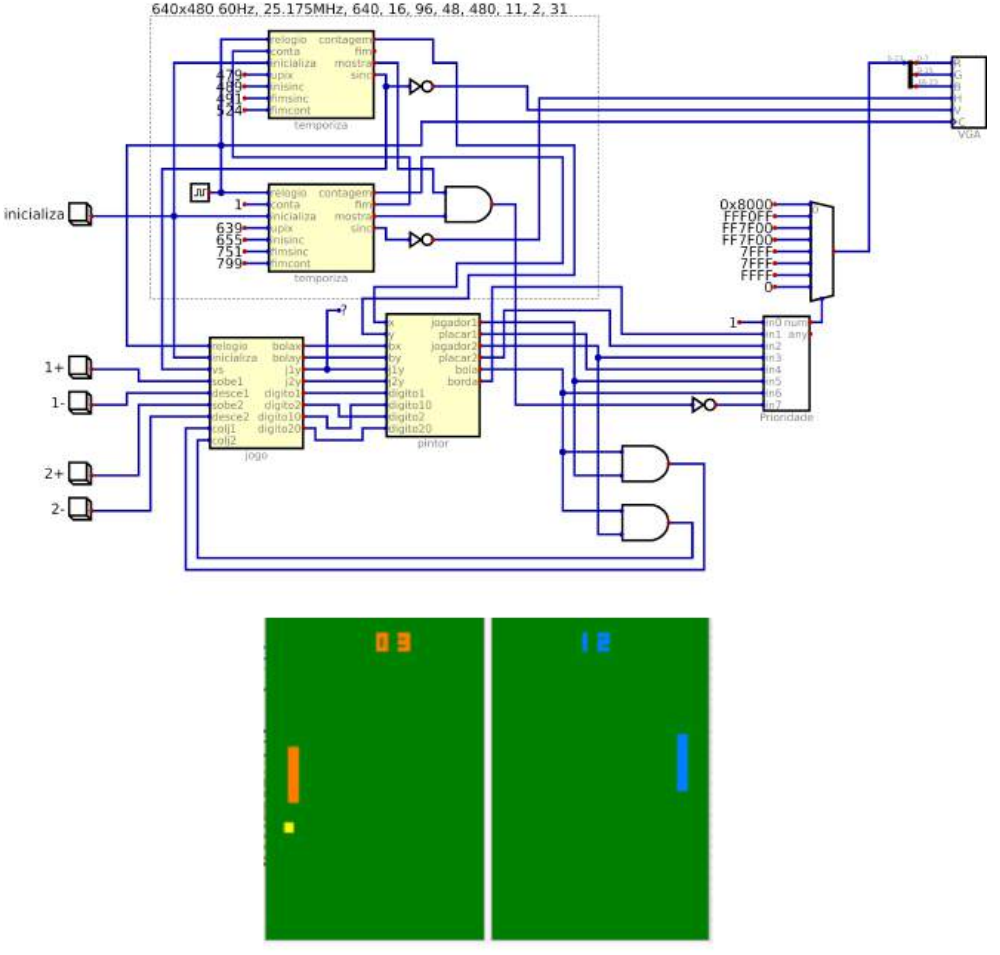
Jogo Pong criado no simulador Digital

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.

---

pixabay.com: MasterTux

Figura 56: Jogo Pong criado no simulador Digital



Linguagens HDL

Imagine ter de projetar um circuito com milhões de portas lógicas, uma por uma. Conforme as tecnologias avançam, a complexidade no design do hardware desses dispositivos também aumenta. E como é possível desenvolver esses sistemas cada vez mais complexos? Através de uma linguagem de descrição de hardware!

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

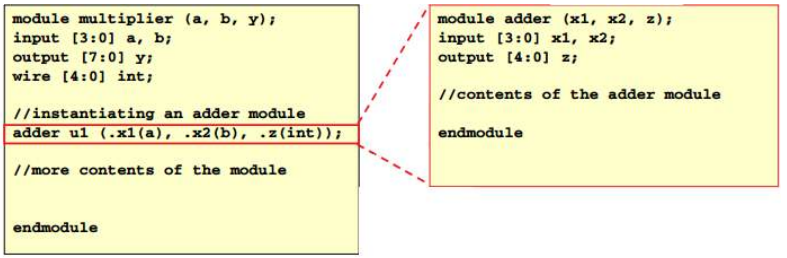
mos HDL's, não estamos programando como em C ou Python, e sim descrevendo um hardware. Dessa forma, diferente das linguagens tradicionais de programação que funcionam de maneira sequencial, ou seja, cada linha é lida pelo compilador, uma após a outra, em HDL os circuitos ali descritos funcionam como 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

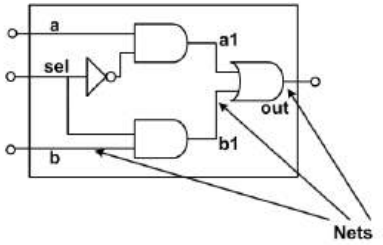
Em linguagens de alto nível como C ou Java possuem tipos de dados como int, char, float, entre outros. Para Verilog, temos três tipos de dados:

- **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

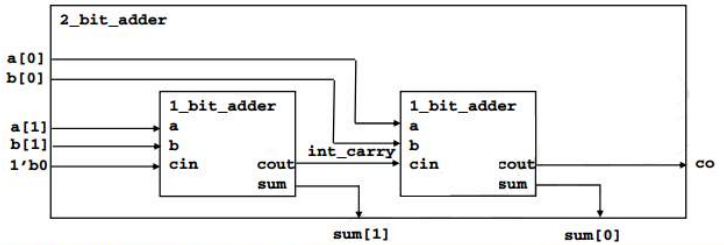
Wires

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

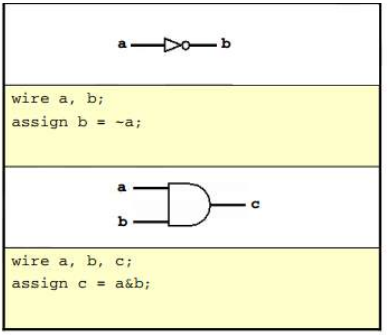



```
module 2_bit_adder (a, b, sum, co);
input [1:0] a, b;
output [1:0] sum;
output co;

wire int_carry;

//instantiating an adder module
1_bit_adder u1 (.a(a[0]), .b(b[0]), .cin(1'b0), .co(int_carry), .sum(sum[0]));
1_bit_adder u2 (.a(a[1]), .b(b[1]), .cin(int_carry), .co(co), .sum(sum[1]));

//more contents of the module
endmodule
```

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.

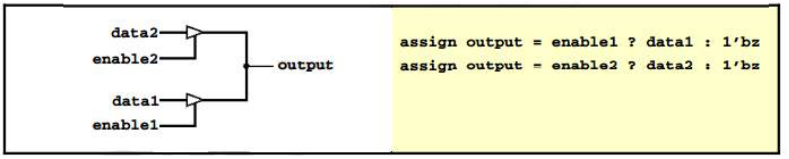
Outra combinação são wires e 'tri-state buffers'. 'Tri-state buffers' são dispositivos que guiam o valor de saída para o valor de sua entrada, ou deixam a saída em um estado de alta impedância. Quando um Net se encontra no estado de alta impedância, pode ser conduzido por

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.

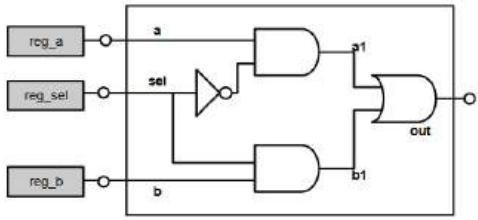
Registers são usados extensivamente em modelagem comportamental e na aplicação de

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.

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

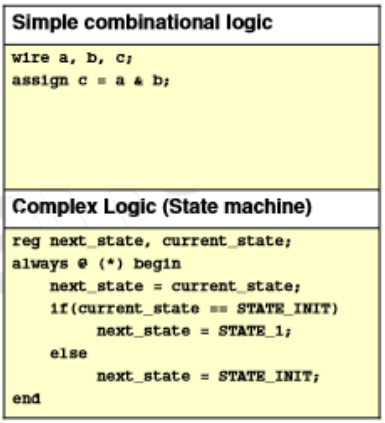
Declarações **blocking** são executadas uma após a outra, semelhante as linguagens de

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



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 eventodentro 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

```
always @ (a or b or sel) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

Event list using the comma operator

```
always @ (a or b or sel) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

Event list using the * wildcard

```
always @ (*) begin
        if (sel == 1)
            op = a;
        else
            op = b;
end
```

Escrevendo um código de apoio não-sintetizável

Sabemos que atrasos e o bloco initial não são sintetizáveis (que não serão traduzidos em circuitos, auxiliando apenas na depuração da saida 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
$display	Imprime sentenças para simulações
$signed	Usado em $display para imprimir variáveis em um formato declarado
$unsigned	Usado em $display para imprimir variáveis em um formato não declarado
real	Tipo de dado para armazenar números decimais
initial block	Usado somente em simulações
gate delays	Usado somente em simulações
always @	Quando usado dentro de blocos initial não é sintetizável
fork, join	A construção fork ... join é usada em blocos initial para executar linhas de código simultaneamente
specify	Especifica delays de portas lógicas
specparam	Especifica uma constante de bloco para armazenar inteiros, hora, números reais, delays ou ASCII char
repeat, for, while	Estes loopings quando usados em blocos initial não sintetizáveis

Tabela 17: Construções em Verilog

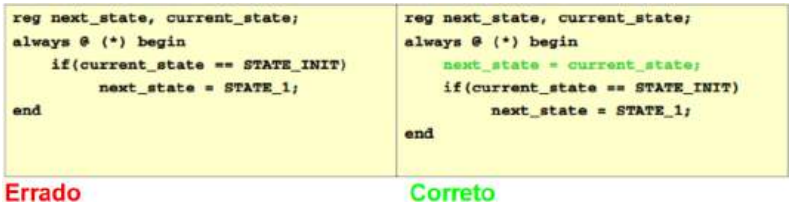
Boas práticas

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.

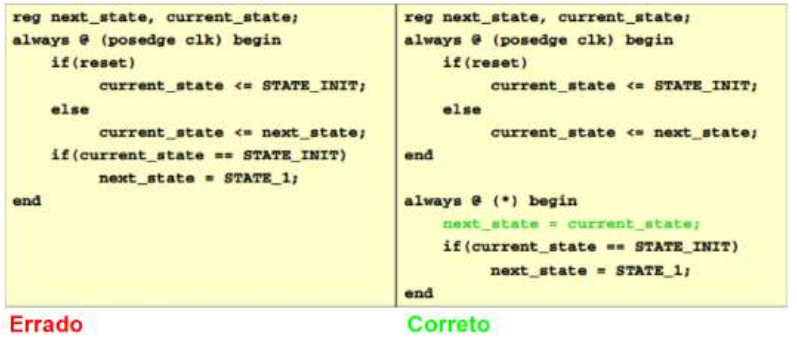
Em um bloco always @ (*), cada reg deve ter um valor para todos os condicionais do case. Se algum deles for omitido, um latch será gerado. Para evitarmos este problema, devemos dar a cada reg um valor de atribuição padrão antes de cada caso ou construção if-else.Veja o exemplo da Figura 66.

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



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



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

```
reg next_state, current_state;
always @ (posedge clk or reset) begin
    if(reset)
        current_state <= STATE_INIT;
    else
        current_state <= next_state;
end
```

Errado

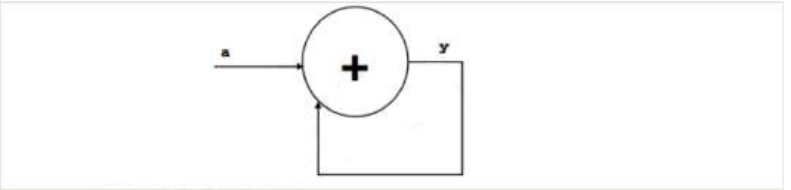
```
reg next_state, current_state;
always @ (posedge clk) begin
    if(reset)
        current_state <= STATE_INIT;
    else
        current_state <= next_state;
end
```

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




```
always @ (*) begin
    y = y + a
end
```

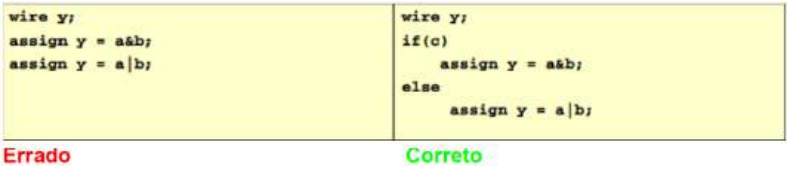
Errado

```
always @ (posedge clk) begin
    y <= y + a
end
```

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



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 sentive** (sensível a diferença entre caracteres maiúculos e minúsculos) e todas as palavras-chaves são apenas aceitas em carcateres 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.	*“Ilegal Left-hand-side assignment”*
Quando o sinal conectado em uma porta de saída é um register	*“Illegal output port specification”*
Quando o sinal conectado em uma porta lógica primitiva é um register	*“Gate has illegal output specification”*
Quando a entrada de um módulo é declarado como um register	*“Incompatible declaration, (signal) defined as input”*

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