Projeto Pegasus 42

O objetivo deste workshop da Gambiconf 2025 é mostrar como os elementos mais básicos do computadores digitais e videogames funcionam e como podem ser usados para formar blocos mais complexos até um computador completo.

Um simulador (chamado Digital) será usado para que os participantes do workshop possam experimentar todos os circuitos apresentados.

1. Chaves, Portas Lógicas, Circuitos Combinacionais

Estes são os blocos básicos dos circuitos digitais.

2. Circuitos Sequenciais

Circuitos mais complexos precisam de memória.

3. Processadores

O mcpu16h é um processador muito símples com apenas 4 instruções, servindo como uma boa introdução enquanto o drv32h é mais prático, sendo compatível com o padrão RISC-V

4. FPGAs e Shin JAMMA

Com circuitos reconfiguráveis (as FPGA - Field Programmable Gate Arrays ou matrizes de portas programáveis em campo) é possível ver os circuitos simulados funcionando no mundo real. Como existem muitas placas de FPGA diferentes foi criado o padrão Shin JAMMA para que o mesmo projeto funcione em todas elas

5. Vídeo e Áudio

Apesar de ser possível criar jogos interessante usando o terminal texto, saída gráfica colorida e som como aqueles gerados pelo clássico Nintendinho (NES ou Famicom) permitem jogos mais divertidos

6. Pegasus 42

Com memória cache e gráficos de alta resolução fica viável rodar a linguagem Squeak Smalltalk

A. História

Além do material apresentado no workshop, é incluída uma história do projeto Pegasus na forma de um apêndice.

1. Chaves, Portas Lógicas, Circuitos Combinacionais

O objetivo deste workshop é a construção de um retro computador (mais ou menos equivalente ao que as pessoas comprariam para ter em casa no início dos anos 1990) chamado Pegasus 42 e usá-lo para programar uns jogos simples. A idéia é que o projeto possa ser completamente entendido desde o nível mais baixo até o sistema em geral.

Para isso usaremos inicialmente um simulador, e depois passaremos as usar uma FPGA (Field Programmable Gate Array - um chip que pode ser reconfigurado para implementar qualquer circuito digital). Para cada nível de abstração existem vários simuladores que podemos usar. De um alto nível para baixo nível temos:

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

A vantagem de se usar um simulador ao invés do produto de verdade é poder ver detalhes que seriam muito difíceis, senão impossível, de medir no circuito real. Os simuladores de baixo nível mostram muito mais detalhes que os de alto nível, mas são proporcionalmente mais lentos quando executados num mesmo computador. Por isso apesar de ser possível simulador um computador completo usando o Spice, talvez level minutos para o circuito simulado executar uma única instrução. Podemos ter que esperar semanas para saber se ele carrega o sistema operacional ou não. Normalmente usamos os simuladores de baixo nível para pequenos trechos do projeto e simuladores de mais alto nível para o sistema completo.

Quando dissemos que o objetivo era entender o Pegasus 42 ao nível mais baixo exageramos um pouco. Vamos considerar o nível de chaves como sendo o mais baixo neste workshop. Apesar do *Digital* não ser otimizado para este nível, ele é suficiente para ilustrar as idéias que serão apresentas em seguida. Ele também não é otimizado para os níveis mais altos, mas para os projetos reduzidos que serão estudados ele é suficiente (mas é lento demais para mostrar a operação de circuitos que geram vídeo de maneira usável).

Uma representação abstrata de um sistema é um retêngulo com um número de entradas e algumas saídas. Normalmente mostraremos as entradas vindo da

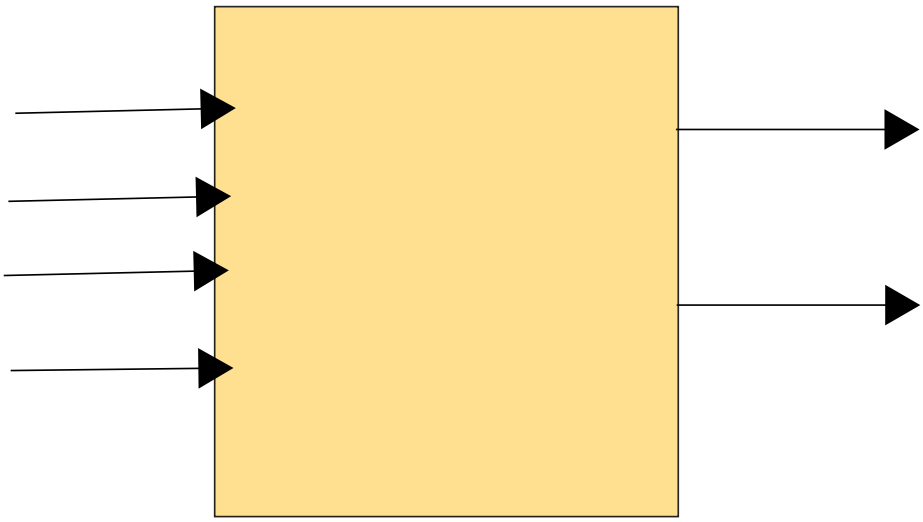

Figure 1: sistema

esquerda e as saídas indo para a direita, mas podemos ignorar esta regra se estiver deixando o desenho mais confuso.

Digital ou Analógica

A primeira escolha que precisamos fazer é a natureza das entradas e saídas do nosso sistema. Nas entradas e saídas analógicas algum valor do nosso circuito (tensão, corrente, etc) é análogo a algum valor do mundo (temperatura, brilho, etc). Nas entradas e saídas digitais uma série de valores do circuito representam um único valor do mundo. Esta série pode usar entradas ou saídas separadas (representação paralela) ou uma mesma entrada ou saída ao longo do tempo (representação serial).

característica	Analógico	Digital
número de circuitos	um	um por dígito
precisão	depende da qualidade do circuito	sempre igual ao número de digitos
ruído	acumula a cada operação	não passa da entrada para saída

Os circuitos analógicos dominaram a computação até a metade do século 20, e a telecomunicação até o fim do século 20. O fator mais importante era o número de circuitos já que os componentes eram muito caros e a ligação deles um processo manual. Com a evolução dos circuitos integrados o custo passou a ser muito

baixo e os outros fatores levaram à digitalização da tecnologia. Nosso projeto é digital.

A humanidade tem usado vários sistemas digitais diferentes para representar números, sendo o mais popular o sistema posicional decimal com digitos indu-arábicos. Quanto mais valores cada dígito pode ter, mas sensível fica aos ruídos. Mas quanto menos valores cada digito poder ter, mais dígitos são necessários para representar o mesmo número. A melhor proteção possível contra o ruído é quando cada dígito pode ter apenas 2 valores, como no sistema posicional binário.

Apesar do sistema binário precisar de mais dígitos (e, portanto, mais circuitos) que as alternativas, cada circuito é mais simples de modo que é a opção que usaremos.

Portas Lógicas

Os circuitos combinacionais são aqueles cuja saída (ou saídas) depende apenas da combinação das entradas. No caso binário, cada digíto só pode ser ou 0 ou 1. Existem várias áreas da matemática que são equivalentes quando são usados apenas dois valores.

Área					
Álgebra Booleana	1	0	inversão	soma	produto
Lógica de Predicados	verdade	falso	não	ou	e
Teoria dos Conjuntos	conjunto universal	conjunto vazio	complemento	união	intersecção
Circuitos de Chaves	5V	0V	normalmente fechado	paralelo	série

Notações de todas estas áreas podem ser usadas para representar os circuitos combinacionais. Uma outra representação possível é simplesmente uma tabela com uma linha para cada combinação de entradas e indicando a saída correspondente. Chamamos isso de "tabela verdade" mesmo quando os valores mostrados são 0 e 1 ao invés de falso e verdadeiro.

Não seremos completamente consistentes, podendo descrever um circuito como tendo a forma de "soma de produtos" (Álgebra Booleana) e outro circuito como contendo "não e" (Lógica de Predicados). Este último é a razão dos circuitos básicos serem conhecidos como "portas lógicas".

Para ilustrar estas idéias usaremos o simulador Digital, como mencionamos anteriormente. O *Digital* foi escrito em Java e por isso é necessário instalar esta linguagem no seu computador antes de poder usá-lo. A vantagem disso é que roda em computadores com diferentes sistemas operacionais e diferentes

processadores. O site indicado é o do código fonte, mas isso só necessário para quem quer modificar o simulador. Na página tem um botão "Download" para baixar *Digital.zip* com a versão mais recente da ferramenta.

Chaves

Como falamos de circuitos de chaves, vamos começar por ai ligando duas chaves em paralelo entre uma lâmpada e uma fonte de alimentação. No menu "Arquivo" selecionamos "Novo". Usando o menu "Componentes", "Chaves", "Chave" posicionamos duas chaves simples como desejamos. Já em "Componentes", "Entradas e Saídas", "LED" temos uma aproximação razoável para a lâmpada que desejamos (o *Digital* tem opções mais sofisticadas mas não as usaremos aqui). Finalmente em "Componentes", "Conexões", "Fonte" temos uma alimentação para o circuito. Note que todos os circuitos precisam de alimentação e de um sinal terra, mas normalmente não mostramos estes e o simulador funciona assim mesmo. Mas se formos construir o circuito de verdade precisamos nos lembrar deles.

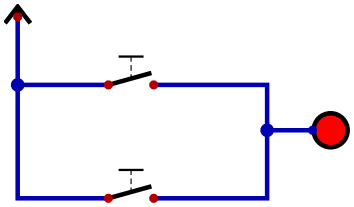

Figure 2: chaves paralelas

Se imaginarmos duas salas com duas portas entre elas, uma ao lado da outra (em paralelo), se uma *ou* outra estiver aberta poderemos ir de uma sala para a outra. Se simularmos este circuito (menu "Simulação", "Iniciar a simulação" ou então o botão com triângulo simples apontando para a direita) veremos que o LED fica apagado. Mas se acionarmos a chave de cima ou a chave de baixo (ou as duas) ele acende.

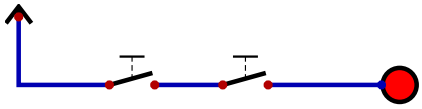

Figure 3: chaves em série

Se imaginarmos duas salas com duas portas entre elas, uma depois da outra via um pequeno corredor (em série), não basta que uma porta esteja aberta. Só será possível passar de uma sala para outra se a primeira *e* e segunda porta estiverem abertas. Neste segundo circuito ligamos as chaves em série e na simulação vemos que o LED permanece apagando a não ser que a primeira *e* a segunda chave tenham sido pressionadas.

Vimos duas das tres equivalências entre chaves e áreas da matemática. A tabela indica a última equivalência (inversão, não, complemento) como sendo uma chave normalmente fechada. Este tipo de chave abre o circuito quando pressionada. Mas aqui iremos mostrar uma alternativa de depende do nível de dispositivos (o único caso que em baixaremos até este nível neste projeto.)

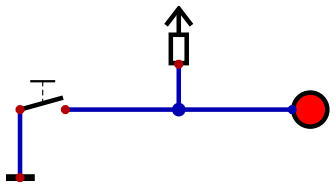

Figure 4: inversão com chave

Só usamos uma chave aqui e no lugar da alimentação usamos "Componentes", "Conexões", "Resistor Pull-Up". Também precisamos de "Componentes", "Conexões", "Terra". Com a chave aberta uma corrente passa pelo resistor e pelo LED, que fica aceso. Ao ser pressionada a chave oferece um caminho para 0V e a corrente vinda do resistor passa por ela ao invés do LED, que se apaga.

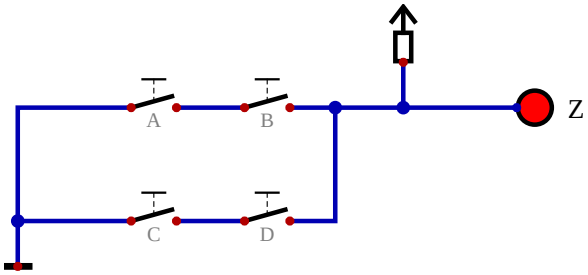

Figure 5: e, ou, inversão com chaves

Aqui temos um exemplo mais complexo usando a mesma idéia. Geralmente a porta AOI ("and/or/invert" - e/ou/inverte) não é considerada porta básica e não a veremos mais neste projeto, mas ela é útil o suficiente para ser incluida em muitas bibliotecas de projeto de circuito integrado. Em álgebra Booleana temos:

```
Z = !(AxB + CxD)
```

enquanto na lógica de predicados seria:

```
Z = não((A e B) ou (C e D))
```

Uma quarta representação do AOI (sendo a primeira a figura ou esquemático, a segunda a álgebra Booleana e a terceira a equação lógica) seria a tabela verdade:

A	B	C	D	Z
aberta	aberta	aberta	aberta	acesa
aberta	aberta	aberta	fechada	acesa
aberta	aberta	fechada	aberta	acesa
aberta	aberta	fechada	fechada	apagada
aberta	fechada	aberta	aberta	acesa
aberta	fechada	aberta	fechada	acesa
aberta	fechada	fechada	aberta	acesa
aberta	fechada	fechada	fechada	apagada
fechada	aberta	aberta	aberta	acesa
fechada	aberta	aberta	fechada	acesa
fechada	aberta	fechada	aberta	acesa
fechada	aberta	fechada	fechada	apagada
fechada	fechada	aberta	aberta	apagada
fechada	fechada	aberta	fechada	apagada
fechada	fechada	fechada	aberta	apagada
fechada	fechada	fechada	fechada	apagada

A tabela mostra um problema - as entradas tem uma natureza (são acionadas por um dedo humano) e a saída tem uma natureza bem diferente (luz saíndo do LED). Se queremos que as saídas de um circuito possam ser usadas como entradas de um outro circuito para construir sistemas maiores elas precisam ser do mesmo tipo. Felizmente foram inventadas chaves que são acionadas por eletricidade: relés (1835), válvulas termiônicas (1904) e transistores (1947). Apesar do *Digital* poder simular relés de modo limitado (como as chaves), vamos trocar as chaves do circuito AOI por transistores MOSFET (Metal/Oxide/Silicon Field Effect Transistor) tipo N (negativo) que é usado em circuitos integrados.

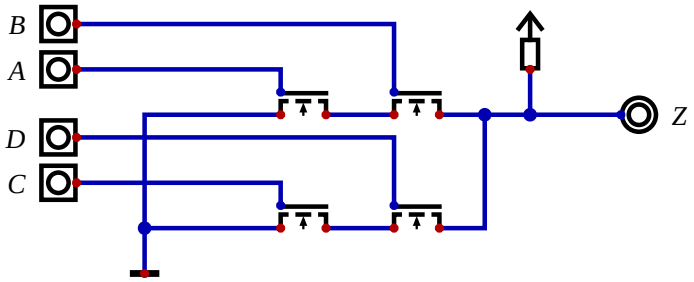

Figure 6: e, ou, inversão NMOS

Além de trocarmos as chaves por “Componentes”,“Chaves”,“FET tipo N” também usamos “Componentes”, “Entradas e Saídas”, “Entrada” e também “Saída” do mesmo menu para indicar que estes sinais podem vir de outro circuito e os resultados podem ir para outro circuito. Na simulação podemos trocar os valores das entradas e observar os valores da saida.

No menu “Análises”, o itém “Analises” cria uma tabela verdade para o circuito e podemos verificar que é a mesma do circuito com chaves. Um problema deste tipo de circuito, que chamamos de NMOS, é que sempre que a saída é 0 existe uma corrente passando pelo resistor e gerando calor (e drenando a bateria se esta for de onde vem a alimentação). Um outro tipo de transistor, o “FET tipo P”, é o oposto do tipo N e conduz corrente quando a entrada é 0. Se trocarmos o resistor por um circuito complementar ao dos transistores N usando transistores P (em série quando o outro é paralelo) o funcionamento do circuito continuará o mesmo mas sem que corrente fique passando sempre.

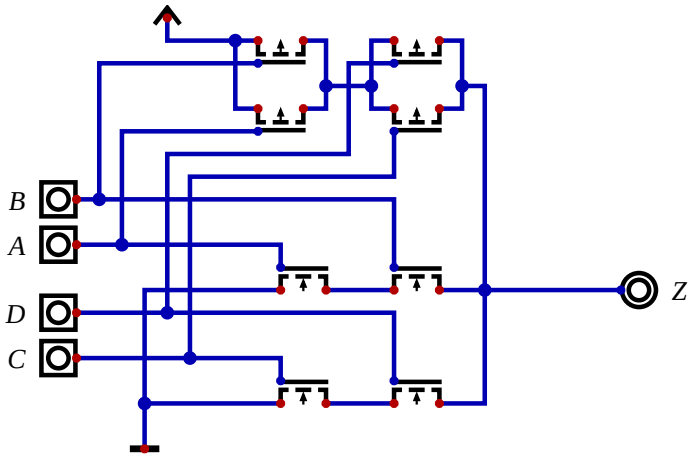

Figure 7: e, ou, inversão CMOS

A complexidade adicional do CMOS limtou este tecnologia a nichos (como relógios digitais) nos anos 1960 e 1970, mas praticamente substituiu todos os outros tipos de circuitos nos anos 1980 quando o crescente número de transistores por chip (Lei de Moore) tornou gastar o dobro de transistores uma boa solução para reduzir a potência.

Portas Lógicas de uma entrada

Até agora vimos apenas uma porta lógica de uma entrada: Não.

Com uma só entrada, apenas duas combinações de entradas são possíveis: ou 0 ou 1. Sua tabela verdade vai ter duas linhas. A saída para cada linha pode ter dois valores de modo que existem 2ˆ2 = 4 tabelas verdades possíveis.

A	Z
0	0
1	0

Na primeira porta a saída é sempre 0. Não é de se surpreender que não falamos dela. Em termos de circuito é só ligar a saída no fio terra.

A	Z
0	1
1	0

Esta é a porta Não que já vimos.

A	Z
0	0
1	1

Aqui a saída é igual à entrada. Como o primeiro circuito dá para fazer isso com apenas um fio.

A	Z
0	1
1	1

A última porta tem como saída sempre um. Isso também pode ser feito como um fio ligado na alimentação.

Então das 4 portas possíveis apenas o Não é interessante. Note que se encararmos o valor de Z em cada tabela como os bits de um número binário com a primeira linha tendo o dígito menos significativo, poderemos chamar estas portas de "porta 0" (Z = 0), "porta" 1 (Z = !A), "porta 2" (Z = A) e "porta 3" (Z = 1).

Portas Lógicas de duas entradas

Usando o mesmo raciocínio, uma porta de duas entradas tem 4 combinações possíveis de entradas e por isso sua tabela verdade tem 4 linhas. Usando o mesmo esquema de enumeração usaremos um número binário de 4 dígitos indicando que existem 2^4 = 16 portas possíveis.

saídas	equação	nome
0 0 0 0	Z = 0	
0 0 0 1	Z = !(A+B)	NOR
0 0 1 0	Z = Ax!B	
0 0 1 1	Z = !B	
0 1 0 0	Z = !AxB	

saídas	equação	nome
0 1 0 1	Z = !A	
0 1 1 0	Z = (!AxB)+(Ax!B)	XOR
0 1 1 1	Z = !(AxB)	NAND
1 0 0 0	Z = AxB	AND
1 0 0 1	Z = (AxB)+(!Ax!B)	XNOR
1 0 1 0	Z = A	
1 0 1 1	Z = A+!B	
1 1 0 0	Z = B	
1 1 0 1	Z = !A+B	
1 1 1 0	Z = A+B	OR
1 1 1 1	Z = 1	

As portas 0000 e 1111 na verdade não tem nenhum entrada, enquanto 0011, 0101, 1010 e 1100 ignoram uma das entradas. Então estas 6 são o que vimos acima.

6 portas tem nomes no menu "Componentes", "Lógica" e um desenho correspondente. Existe um desenho para AND (1000), OR (1110) e XOR (exclusive OR - 0110) e para o inverso deles colocamos uma bolinha na saída e um "N" no início do nome. O *Digital* também permite colocar uma bolinha nas entradas e ai podemos usar um AND para as portas 0010 e 0100 e uma porta OR para 1011 e 1101.

Soma de Produtos Se olharmos as linhas do XOR e XNOR veremos que são as equações mais complicadas. No caso do XOR o primeiro produto (AND) é !AxB e corresponde diretamente ao 1 da esquerda, enquanto Ax!B é que gera o 1 da direita.

saídas	produto
0 0 0 1	!Ax!B
0 0 1 0	Ax!B
0 1 0 0	!AxB
1 0 0 0	AxB

Então o XOR é a soma (OR) do segundo e do terceiro produtos desta tabela. Isso é verdade para qualquer porta lógica e pode ser expandido para qualquer número de entradas. Então nunca irão descobrir no futuro uma porta lógica nova que não sabemos implementar. A tabela verdade pode ser transformada diretamente num circuito.

Isso não quer dizer que o circuito produzido assim será muito bom. Usando este método para a penúltima porta lógica teríamos

```
Z = (Ax!B)+(!AxB)+(AxB)
```

Mas sabemos que uma símples porta OR faz a mesma coisa. Felizmente a álgebra booleana tem regras de simplificação parecidas com as regras da álgebra normal e existe um método gráfico (chamado de Mapa de Karnough, que é uma das coisas que o *Digital* pode gerar) para reduzir ao máximo a lógica mantendo a mesma operação.

Múltiplos Bits

Uma desvantagem dos circuitos digitais em relação aos analógicos é a repetição dos mesmo circuito para cada dígito. Se usarmos 32 bits para representar valores, teremos 32 cópias de cada circuito.

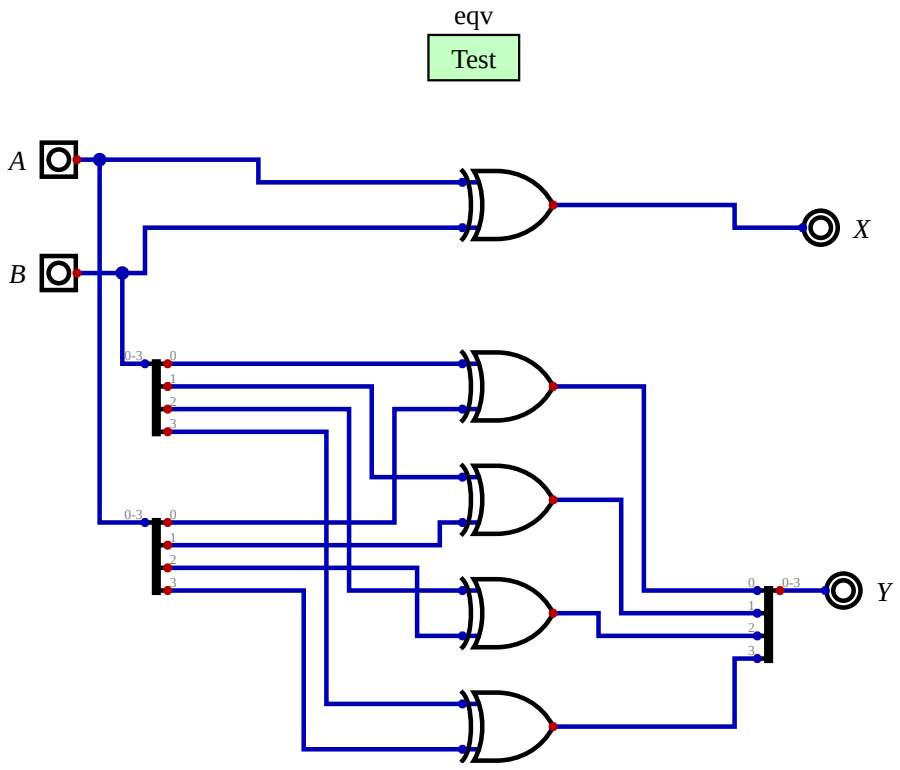

Figure 8: multipĺos bits

O *Digital* tem um recurso para reduzir esta complexidade. Para cada entrada e saída, além de um nome podemos definir uma "largura" em número de bits. No circuito acima mudamos A, B, X e Y para terem 4 bits cada uma. Em "Componentes","Conexão","Distribuidor" temos um meio para ligar sinais com diferentes números de bits. Nos dois da esquerda a entrada foi configurada como "4" e a saída como "1,1,1,1" enquanto no da direita foi o contrário. Além disso

cada componente normal pode ser configurado para uma certa largura. A porta ou exclusivo de cima foi configurada para 4 bits enquanto as quatro de baixo são 1 bit cada.

Os dois circuitos devem ser completamente equivalentes, mas o de cima é mais fácil de se entender por ser bem menor. E isso para 4 bits - a ganho para, por exemplo, 32 bits é proporcionalmente maior. Na edição do circuito o *Digital* não tem nenhuma indicação visual de que a porta de cima é diferente das outras, que as entradas e saídas são múltiplos bits ou que os fios ligando estarão carregando múltiplos bits. Durante a simulação, no entanto, os fios de 1 bit são mostrados em verde escuro (para 0) ou verde claro (para 1) enquanto os com vários bits continuam azul escuro para com o valor indicado acima do fio. E "clicando" numa entrada de 1 bit ela inverte seu valor enquanto numa de múltiplos bits aparece uma caixa de diálogo para definir o novo valor.

Usando "Análises","Análises" veremos a tabela verdade e podemos comparar os bits de X e de Y para confirmar que os circuitos são realmente equivalentes. Com 256 linhas, no entanto, esta confirmação é bem cansativa. E se fizermos qualquer alteração no circuito teremos que repetir este estudo da tabela verdade. Felizmente o *Digital* permite automatizar isso via "Componetes","Diversos","Caso de Teste". Editamos o teste (que chamamos de "eqv") para:

```
A B X Y

loop(a,16)
  loop(b,16)
        (a)  (b)  (a^b)  (a^b)
  end loop
end loop
```

Usando "Simulação","Executar testes" todas as 256 combinações de a e b são geradas para A e B e X e Y são comparadas com a *XOR* b. Todos os testes que passam são mostrados em verde e todos os que falham com um "x" vermelho. É possível examinar os detalhes para saber ondo ocorreu a falha.

Outra maneira que o *Digital* elimina complexidade é o uso de projetos hierarquicos. Vários componentes podem ser combinados num circuito que pode ser mostrado como um bloco único num circuito de mais alto nível. Qualquer projeto prático deve usar isso extensamente e também incluir muitos testes para cada sub-circuito. Mas como o objetivo deste workshop é mostrar a complexidade de um computador, os projetos que serão vistos terão o mínimo de níveis possível. Muitas vezes sub-circuitos são comparados às chamadas de subrotinas nas linguagens de programação mas na verdade são mais equivalentes às macros.

Decisões

Nas linguagens de programação temos construções como "if A then B else C" para usar uma entrada para escolher entre duas outras entradas.

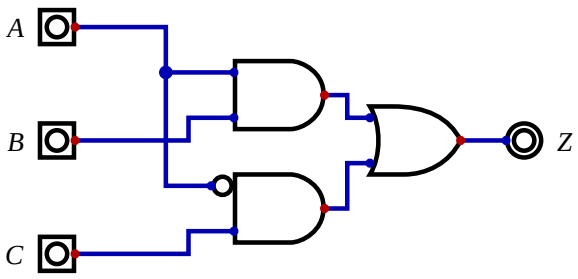

Figure 9: multiplexador de 2 entradas

Olhando a tabela verdade vemos que Z = B sempre que A é 1, mas Z = C se A for 0. Isso significa que circuitos combinacionais podem tomar decisões. Na verdade podem escolher entre mais de duas alternativas, como o "switch/case" nas linguagens de programação.

Isso é mais complicado de testar pois com 8+3 = 11 entradas são 2048 combinações possíveis. O teste exaustivo (t8x256) ainda é possível:

```
S_2 S_1 S_0   A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0  Z

loop(s,8)
  loop(a,256)
        bits(3,s) bits(8,a) bits(1,a>>s)
  end loop
end loop
```

Aqui estamos confirmando que a saída corresponde ao bit selecionado por S (Z é calculado deslocando a entrada à direita de modo que o bit menos significativo é a entrada desejada). Um teste mais cuidadoso é verificar que quando todas as entradas são 0 a saída também é e em seguida tornar uma entrada por vez 1 e a saída deve continuar 0 a não ser quando a entrada desejada é a que vai para 1. Ao invés de 8x256 testes precisamos de apenas 8x9. Neste caso o teste completo é até melhor, mas na fabricação de um produto onde parte do custo é o tempo gasto no equipamento de teste a solução mais reduzida seria mais vantajosa.

Nosso projeto usará muitos multiplexadores e o circuito mostrado é meio grande (e teria que ser repetido 32 vezes para selecionar entre 8 valores de 32 bits cada). Ao nível de portas lógicas esta é a melhor solução, mas se descermos ao nivel de chaves é possível economizar transistores.

Este circuito passa pelos mesmos testes que o circuito anterior. Na prática este uso de transistores de passagem NMOS causa uma redução no sinal de saída mas com dois inversores este problema é eliminado. Usar pares NMOS e PMOS de passagem é outra solução, mas a fiação fica meio complexa. A idéia de mostrar isso é para que possamos usar multiplexadores nos nossos projetos sem nos preocuparmos demais com o custo deles.

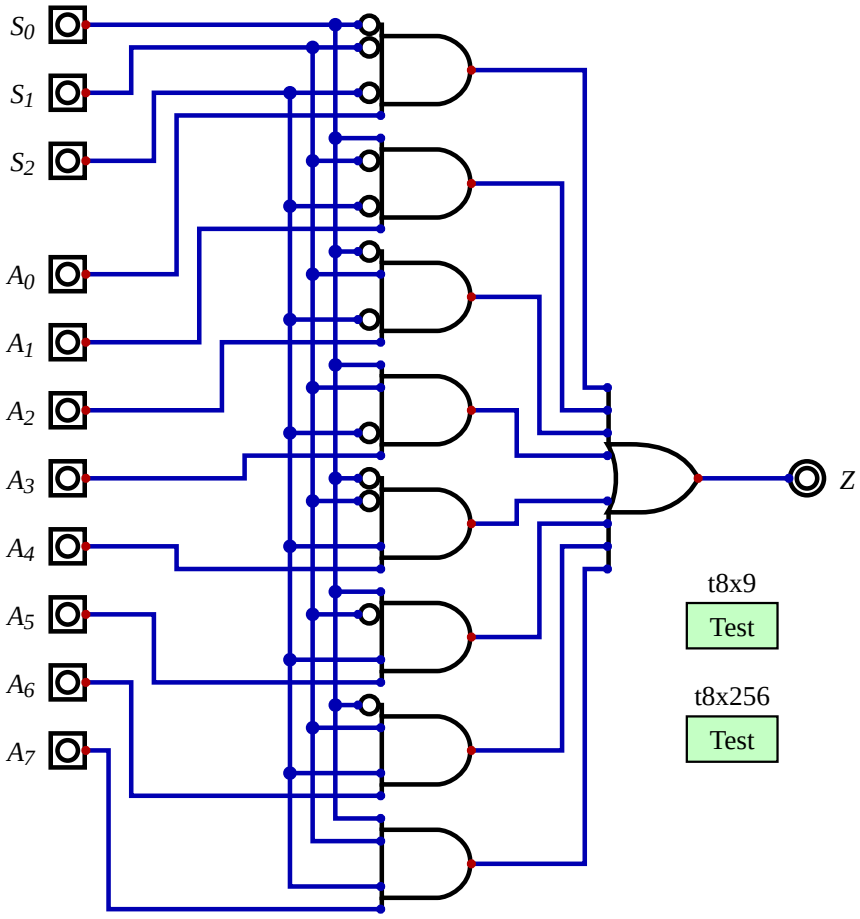

Figure 10: multiplexador de 8 entradas

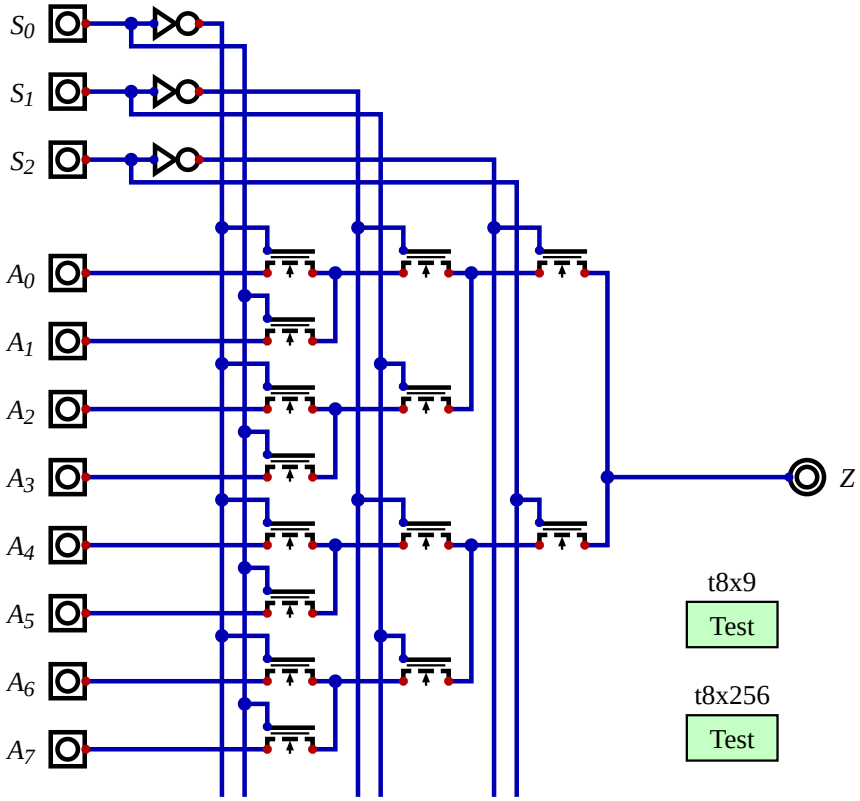

Figure 11: multiplexador de 8 entradas

Números

Uma idéia muito popular é que computadores apenas manipulam números mas nós podemos interpretar estes números como sendo letras, cores, sons, etc. Isso está sutilmente errado - os computadores podem manipular representações de muitas coisas, incluindo números. Não é fácil perceber isso para os números inteiros positivos, mas para números negativos ou de ponto flutuante fica mais claro.

O primeiro detalhe que precisamos observar é a diferença entre "+" na álgebra Booleana e na aritmética binária:

A	B	A+B Booleana	A+B Binária
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	10

O resultado na última linha tem mais digitos que as entradas no caso da aritmética binária. Este também é o caso na aritmética decimal: somando dois números decimais de 5 dígitos pode gerar um resultado de 6 dígitos. E para cada dígito o resultado será de 0 a 18, sendo o último de 2 dígitos.

Note que a última linha da tabela não está dizendo que "um mais um igual a dez". Da mesma forma que o número decimal 307 significa 3x100 + 0x10 + 7x1, o número 1010 binário significa 1x8 + 0x4 + 1x2 + 0x1 que é o número dez. No sistema posicional decimal o dígito mais da direita é o da unidade e cada dígito para a esquerda vale dez vezes mais. No sistema posicional binário o digito mais de direita é o da unidade e cada dígito para esquerda vale duas vezes mais. O número 10 em binário é 1x2 + 0x1 que é dois.

Vamos chamar os dígitos da soma de números binários de 1 bit de S (de "soma") para o menos significativo e C (de "carry", que é "vai um" em inglês) para o mais significativo.

A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

C tem a tabela verdade da porta lógica AND e S da porta OU Exclusivo. Então um somador é:

Para os dígitos menos significativos dos números de entrada isso já serve. Mas para os demais é necessário levar em conta o "vai um" vindo do dígito logo à

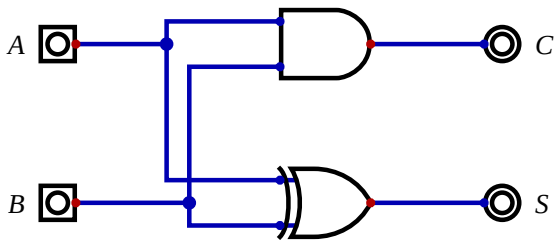

Figure 12: meio somador

direita. Por isso chamamos este circuito de "meio somador" e usamos dois deles para criar um "somador completo", que é um circuito com 3 entradas e 2 saídas.

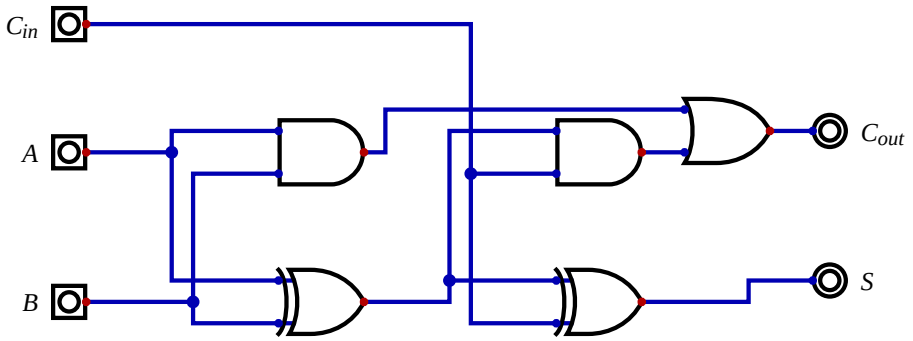

Figure 13: somador completo

Repetindo o somador completo 4 vezes podemos somar dois números binários de 4 bits cada um. Na verdade poderíamos ter usado um meio somador para o dígito menos significativo, mas fazendo deste jeito fica fácil combinar vários blocos destes para números ainda maiores.

Podemos testar que realmente estamos somando números:

```
C_in   A_3 A_2 A_1 A_0   B_3 B_2 B_1 B_0   C_out S_3 S_2 S_1 S_0

loop(a,16)
  loop(b,16)
      0 bits(4,a) bits(4,b) bits(5,a+b)
      1 bits(4,a) bits(4,b) bits(5,a+b+1)
  end loop
end loop
```

Repare que com *C_out* e *S_3* a *S_0* o resultado é de 5 bits, um dígito à mais que as entradas.

Este é um "somador em cascata" e não é muito rápido. O vai um passa de digíto em dígito e só chega no dígito mais significativo com muito atraso. Se dobrarmos

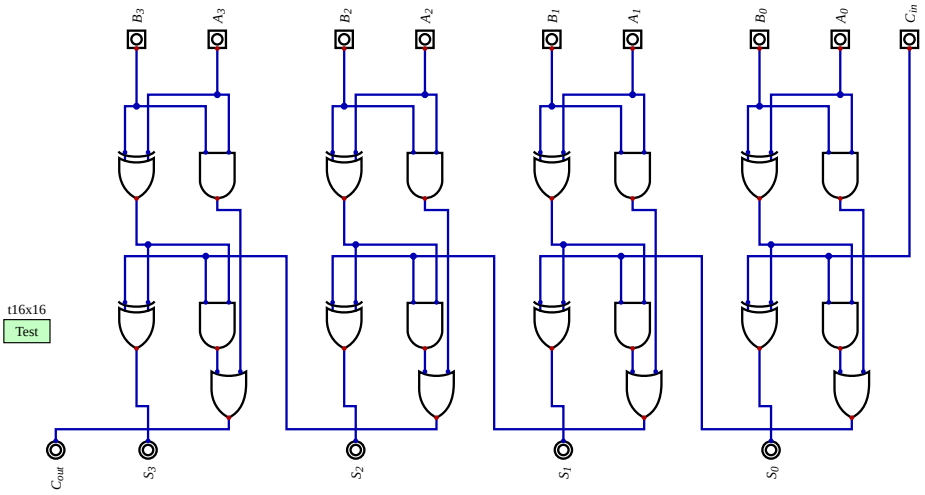

Figure 14: somador de 4 bits

o número de dígitos o somador vai operar com metade da velocidade. Existem circuitos mais complexos que reduzem este problema, mas para o nosso projeto esta solução simples já é suficiente.

Para a subtração aparece um problema novo: números negativos. Mencionamos acima que computadores manipulam representações e não números, mas é fácil ignorar a diferença no caso dos números positivos. Para os números negativos precisaremos escolher um entre várias representações possíveis, cada uma com suas vantagens e complicações.

A representação mais popular para os números decimais é o do sinal/magnitude. Um dígito especial à esquerda indica se o número é positivo ("+" ou nada) ou negativo ("-"). Os demais dígitos indicam o valor absoluto do número. A mesma idéia pode ser usada com números binários, inclusive usando 0 e 1 para indicar números positivos e negativos tornando o sistema ainda mais uniforme. Dois problemas deste sistema é o ocorrência de dois zeros diferentes (+0 e -0) e o fato que a soma e subtração funcionam um pouco diferente e é necessário examinar os sinais dos operandos e o tamanho das magnitudes para escolher a operação certa.

Uma representação que foi usada apenas em algumas das primeiras calculadoras mecânicas foi o complemento de nove. Um número negativo é representado trocando cada dígito por nove menos aquele dígito. O negativo de 307, por exemplo, seria 692 (e no caso de uma cálculadora de seis dígitos 000307 negativo seria 999692 e ai fica mais fácil interpretar o dígito mais da esquerda como indicando o sinal). Agora para subtrair basta inverter o segundo operando e somar (ignorando qualquer "vai um" gerado pela operação): 000531 - 000307 = 000531 + 999692 = (1)000223. Mas este resultado está errado já que 307+223

= 530. Isso é um dos problemas do complemento de nove: precisamos somar um para ter o resultado certo. E ele também tem dois zeros. No caso binário o equivalente é o complemento de um (que é a função NOT).

O complemento de dez é uma pequena modificação do sistema anterior que corrige os dois problemas ao mesmo tempo. Para negar um número subtraimos cada dígito de nove e somamos um ao resultado. O complemento de dez de 000307 é 999693 e agora somas já dão o resultado correto. Além disso 000000 continua zero mas 999999 passa a ser um negativo - não existe mais o zero negativo. Um detalhe é que existem mais números negativos que positivos para determinada quantidade de dígitos, mas isso é apenas uma questão estética. O caso binário equivalente é o complemento de dois.

O último sistema que consideraremos é o de deslocamento ou víes ("bias" em inglês). Aqui somamos um valor a todos os números para serem todos positivos. Escolhendo o deslocamento como sendo metade do maior númeor possível mais um, a soma de dois números transformas os dois deslocamentos em um "vai um", e ai precisamos somar um terceiro deslocamento para ajustar a resposta. 531 é representado por 500531 e -307 por 499693. 500531 + 499693 + 500000 = (1)500224.

Comparando estas representações no caso de números binários de 3 dígitos (valores mostrados em decimal signal/magnitude):

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

Todas estas representações foram usadas na história da computação mas nos anos 1960 o complemento de dois acabou dominando e é o que usaremos no nosso projeto. Mas no padrão IEEE 754 de ponto flutuante a mantissa usa a representação sinal/magnitude enquanto o expoente usa a representação de deslocamento.

Com uma pequena modificação no somador de 4 bits ele também pode subtrair usando a representação de complemento de dois. Dissemos que a NOT faz o complemento de um, mas usando um XOR para cada dígito podemos controlar se fazemos esta inversão ou não. No teste de subtração *-16x16* vemos que na verdade estamos calculando *a-b-1*, que é o problema que vimos no complemento de um. Mas usando *Cin* podemos corrigir isso, obtendo o complemento de dois. No modo subtração o circuito gera um *Cout* invertido e o teste compensa este detalhe.

```
invB  C_in   A_3 A_2 A_1 A_0   B_3 B_2 B_1 B_0   C_out S_3 S_2 S_1 S_0
```

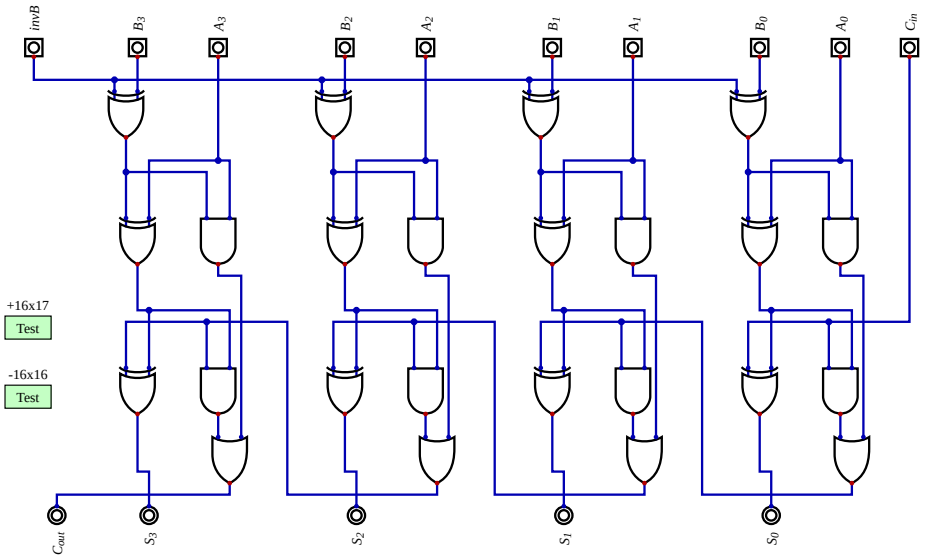

Figure 15: somador e subtrator de 4 bits

```
loop(a,16)
  loop(b,16)
      1 0 bits(4,a) bits(4,b) bits(5,(a-b-1)^16)
      1 1 bits(4,a) bits(4,b) bits(5,(a-b)^16)
  end loop
end loop
```

2. Circuitos Sequenciais

O tempo não é um fator num circuito combinacional. É verdade que não sendo infinitamente rápido, a resposta só fica pronta depois de um certo atraso depois que as entradas recebem seus valores.

Já nos circuitos sequenciais o tempo é fundamental. As entradas chegam em sequência pelos mesmos sinais ao longo do tempo e as respostas também são enviadas como uma sequência de valores ao longo do tempo. Podemos converter o sistema combinacional abstrato que já vimos num sistema sequencial ligando algumas das suas saídas nas entradas. Dizemos que o valor sendo realimentado é o "estado atual" do sistema.

Na prática um circuito destes pode até funcionar, mas seria um tanto instável pois alguns caminhos gerando uns bits do estado atual podem ter atrasos maiores que os de outros bits. A solução mais popular é acrescentar um sinal de "relógio" para que a realimentação ocorra de modo controlada.

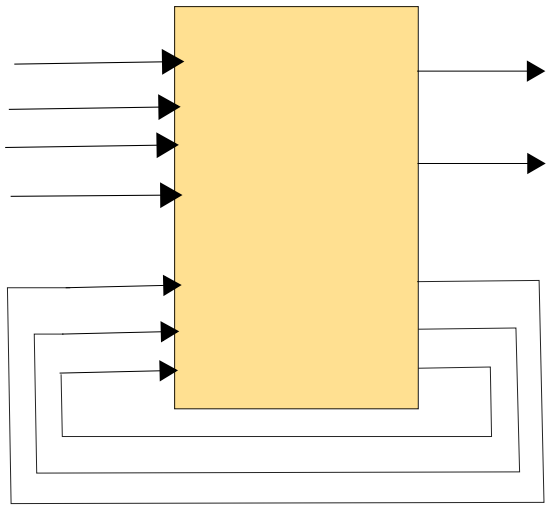

Figure 16: sistema sequencial

Osciladores

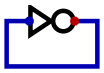

Figure 17: oscilador com inversor

O *Digital* se recusa a simular este circuito, que é o sistema sequencial mais simples. Um problema é que o circuito é contraditório: a saída teria que ser 1 para entrada 0, ou 0 para entrada 1. Mas tem um fio ligando as duas de modo que elas tem o mesmo valor. Se construirmos este circuito ele vai ficar alternando muito rapidamente entre 0 e 1. É o que chamamos de "oscilador". A frequência da oscilação depende da velocidade do inversor e o atraso no fio. O problema principal do ponto de vista do simulador é que o valor inicial é desconhecido, o que pode ser resolvido com um sinal de inicialização (em inglês é "reset").

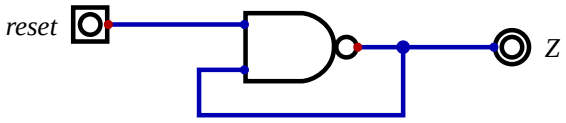

Figure 18: oscilador com nand

Se tentarmos simular parece funcionar, mas assim que mudamos *reset* para 1 aparece um erro. A solução é a simulação passo a passo. Depois de mudar *reset* para 1 a cada passo da simulação a saída alterna entre 0 e 1.

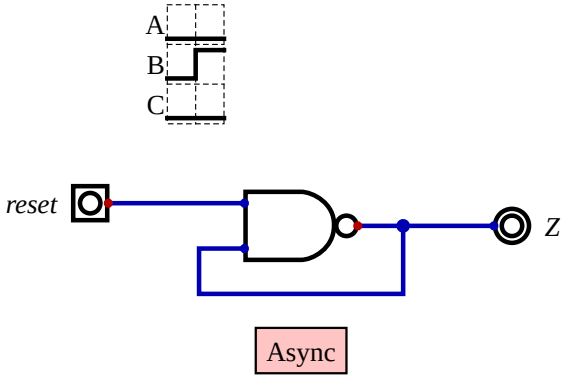

Figure 19: oscilador com nand

Acrescentando "Componentes","Diversos","Async" muda um pouco como a visualização da simulação ocorre, e com "Componentes","Entradas e Saídas","Gráfico de dados" podemos ter uma visualização das entradas e saídas ao longo do tempo, que é para circuitos sequenciais o que a tabela verdade é para circuitos combinacionais.

Memória

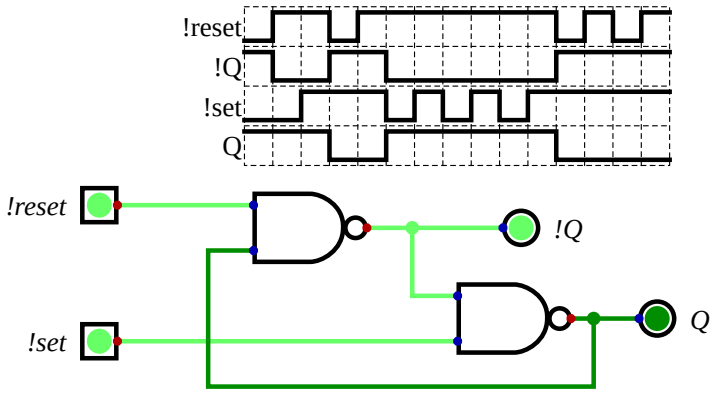

Figure 20: flip-flop com nands

Ligando dois osciladores em série ele deixa de oscilar pois um número par de inversões não é contraditório. O circuito é "bi-estável", o que quer dizer que existem duas situações diferentes em que ele fica parado. A saída da primeira parte é sempre oposta à da segunda parte, por isso usamos uma "!" na frente do nome. E também nos nomes das entradas para indicar que estas devem normalmente ficar em 1 e irem para 0 quando desejamos que façam sua função.

O nome deste circuito é "flip-flop" para indicar que é bi-estável e é a memória mais simples de um computador com capacidade de armazenar 1 bit. O nome mais completo é "flip-flop RS" já que as entradas *!reset* e *!set* são separadas. O circuito "lembra" que um pulso negativo chegou em "!set" mesmo depois que este pulso já não está mais lá. Pulsos adicionais não fazem nada. A mesma coisa para pulsos em *!reset*.

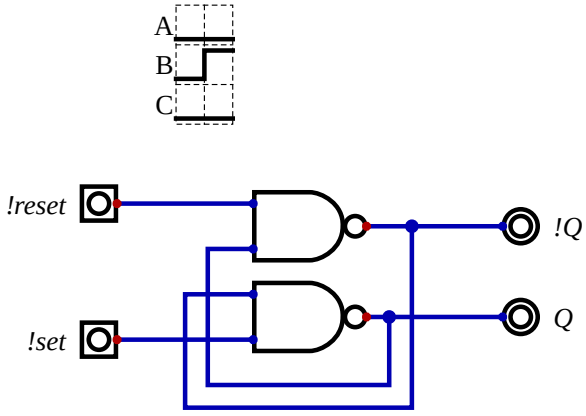

Figure 21: flip-flop com nands

Este é exatamente o mesmo circuito mas desenhado no estilo do sistema sequencial abstrato com as saídas dando a volta no circuito para se ligarem às entradas. Isso é só para que mesmo com as complicações dos próximos circuitos continuemos a ter esta idéia básica em mente.

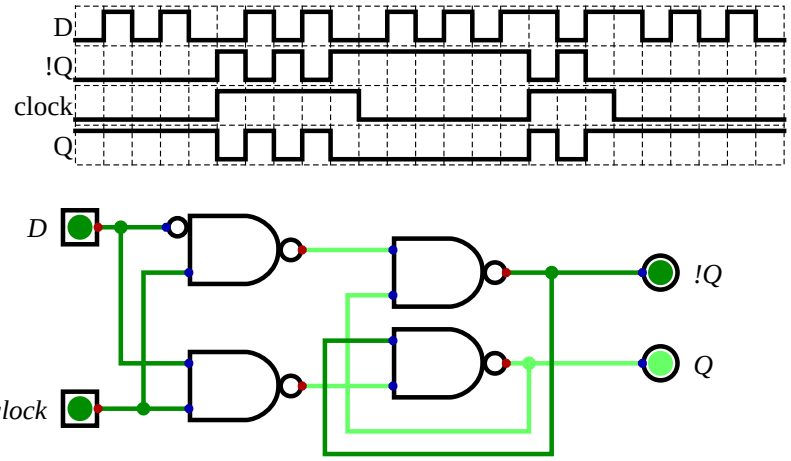

Figure 22: flip-flop tipo D

Ao invés de gerar *!reset* e *!set* diretamente, é mais conveniente ter um sinal de dados D e um de relógio *clock*. Enquanto o relógio está alto, a saída reflete D com um pequeno atraso. Quando relógio está baixo o valor da saída não muda independentemente do que está acontecendo em D. Em inglês este circuito é conhecido como "latch" pois ele "trava" a saída na borda de descida do relógio (o instante em que passa de 1 para 0).

Também podemos usar um multiplexador para ter a mesma funcionalidade. Mas neste caso não temos a saída invertida. Projetos modernos evitam usar latches pois durante metade do ciclo de relógio não podemos confiar na saída. Com dois latches seguidos operando em níveis opostos do relógio podemos ter uma saída que fica estável durante quase o ciclo inteiro, sendo a única incerteza bem próximo de uma das bordas do relógio.

Em um projeto realista poderemos querer amostrar D em certos ciclos de relógio mas não em outros. Uma opção seria não deixar o relógio subir nestes casos, fazendo um AND com alguma condição. Mas isso não é uma boa idéia pois agora o relógio percebido por uma parte do circuito está defasado em relação a outras partes do sistema e este é o tipo de coisa que parece funcionar por milhões de ciclos de relógio mas de vez em quando falha. Com mais um multiplexador poderemos ter um sinal de habilitação *En* (de "enable") que opera no mesmo caminho que D e não no caminho de *clock*.

Esta funcionalidade pode ser obtida em "Componentes","Memória","Registrador". Mas é importante entender o que está acontecendo dentro dele. Trocamos a

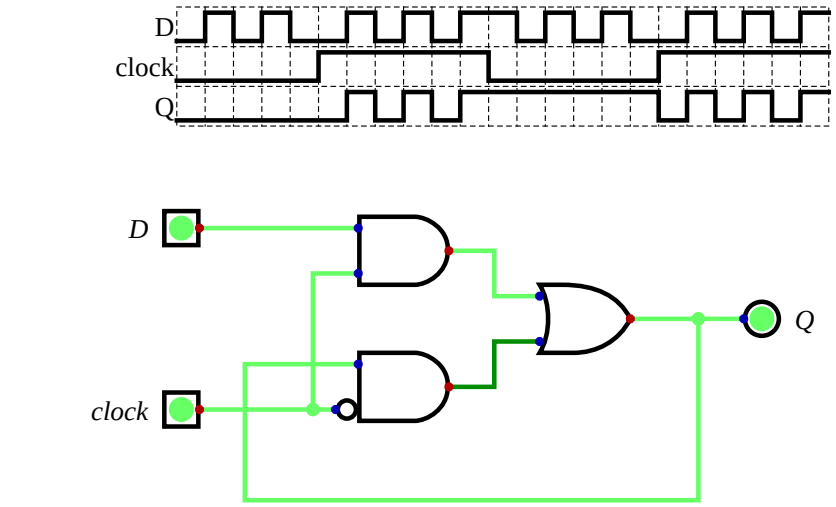

Figure 23: latch

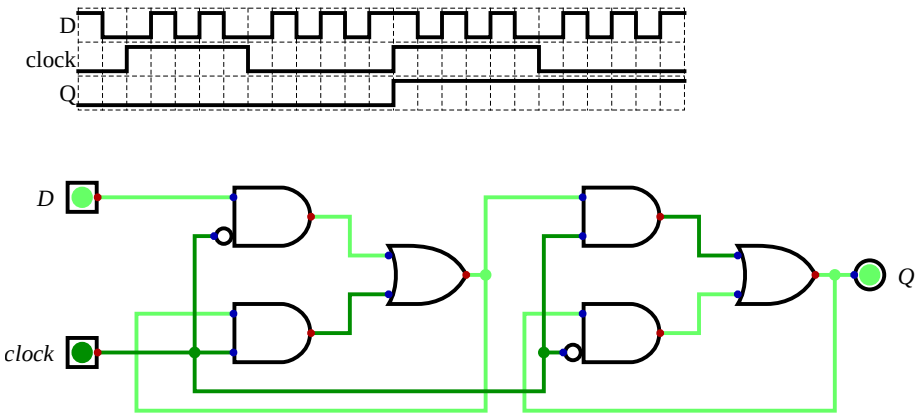

Figure 24: flip-flop tipo D sensível à borda de subida

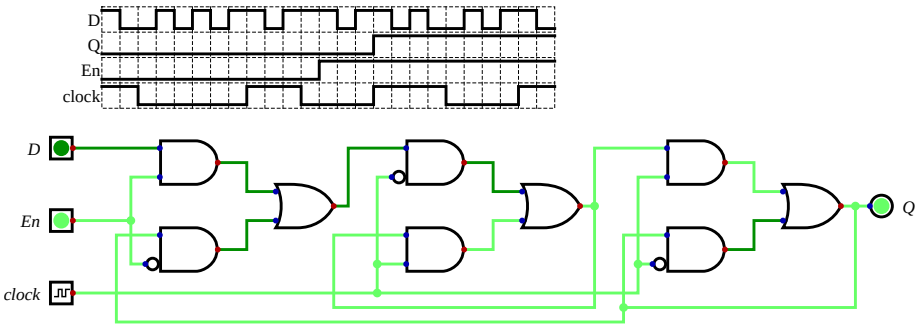

Figure 25: registrador

entrada *clock* de uma entrada normal para uma especial. Aqui não fez nenhuma diferença na simulação, mas podemos configurar para pulsar numa velocidade desejada sem ter que ficar manualmente clicando nela. E nos testes este tipo de entrada pode ter não apenas 0 ou 1 mas também C para indicar uma borda de subida.

Máquinas de Estados Finitos

Com os registradores podemos resolver o problema de instabilidade do circuito sequencial abstrato. Basta passar os sinais das saidas a serem realimentadas por registradores e fazer as saídas dos registradores irem para as entradas. Assim não importa se os diferentes bits são calculados com atrasos diferentes pois serão amostrados todos ao mesmo tempo na borda de subida do relógio. A única preocupação com tempo restante é ver se as bordas de subida são suficientemente espaçadas para que o circuito combinacional termine sua operação. É por isso que dizemos que um processador pode operar a 1,5 GHz enquanto outro pode chegar a 3,2 GHz.

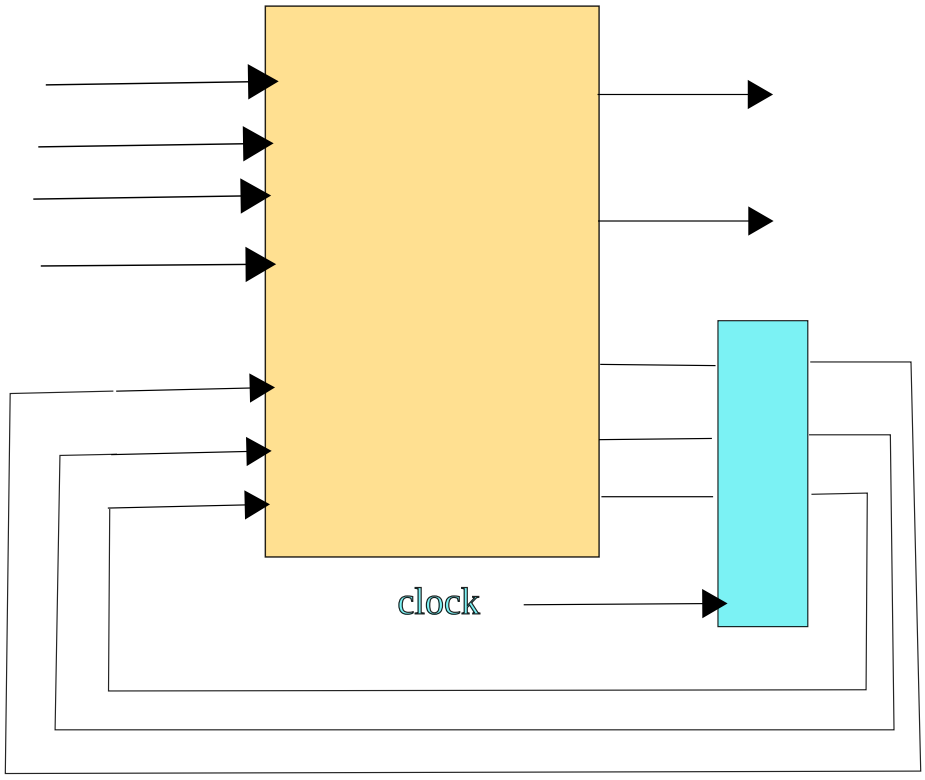

Figure 26: máquina de estados finitos

Com a introdução do relógio, o estado do sistema pode ser pensado como saltando

instantaneamente de um valor para o outro. O número de estados possíveis é limitado pelo número de bits para representar o estado atual, mas o número de estados reais pode ser bem menor que isso. Chamamos este tipo de sistema de "máquina de estados finitos" ("finite state machine" ou "FSM" em inglês).

Uma representação possível para a FSM é uma tabela com, por exemplo, uma linha para cada estado e uma coluna para cada entrada possível. Cada célula indica a saída e também o estado seguinte. O *Digital* tem um editor para uma representação gráfica bem popular: cada estado é mostrado como um círculo e setas ligando os círculos mostram as possíveis transições. As setas indicam quais deve ser as entradas para que salte para o outro estado e também quais devem ser os valores das saídas.

Nos exemplos do *Digital* tem *rotDecoderMealy.fsm* com entradas A e B e saídas L e R. Cada estado tem um nome, mas também tem um número de 0 a 6 que são os valores que os registradores devem ter para cada estado. Este sistema consegue descobrir se um eixo está girando para a esquerda ou para a direita.

O "Mealy" do nome do arquivo é para indicar que esta FSM é do tipo Mealy, que foi definido por George H. Mealy em 1955. Uma característica deste tipo de FSM é que as saídas dependem não apenas do estado atual mas também das outras entradas. Isso significa que as saídas podem mudar entre bordas do relógio se as entradas mudarem.

De dentro do editor de FSM podemos pedir para gerar a tabela verdade (das "transições", já que sendo um sistema sequencial a máquina como um todo não pode ser descrita por uma tabela verdade) e da tabela podemos pedir para gerar um circuito. O estado atual fica armazenado nos registradores 2n, 1n e 0n e podemos ver as entradas A e B e o circuito combinacional na forma de soma de produtos que gera tanto o valor do próximo estado quando as saídas L e R.

As saídas dos registradores deveriam ter fios indo até as entradas do circuito combinacional. O *Digital* tem um recurso chamado *tunel* para fazer um sinal saltar de um ponto para o outro (ou vários outros) sem atravessar por cima do resto do desenho, e isso foi usado aqui.

Em 1956 Edward F. Moore definiu uma outra variante da FSM onde as saídas dependem exclusivamente do estado atual. A vantagem é que as saídas ficam estaveis entre bordas do relógio. No mesmo exemplo implementado como FSM do tipo Moore vemos que são necessários mais estados para se ter o mesmo resultado, neste caso. Como as saídas estão atrasadas um ciclo de relógio em relação às entradas, muitas vezes é necessário fazer um planejamento bem mais cuidadoso para implementar uma FSM tipo Moore do que a correspondente tipo Mealy.

A diferença mais óbvia deste circuito em relação ao anterior é que aqui as saídas L e R não tem ligação com as entradas A e B.

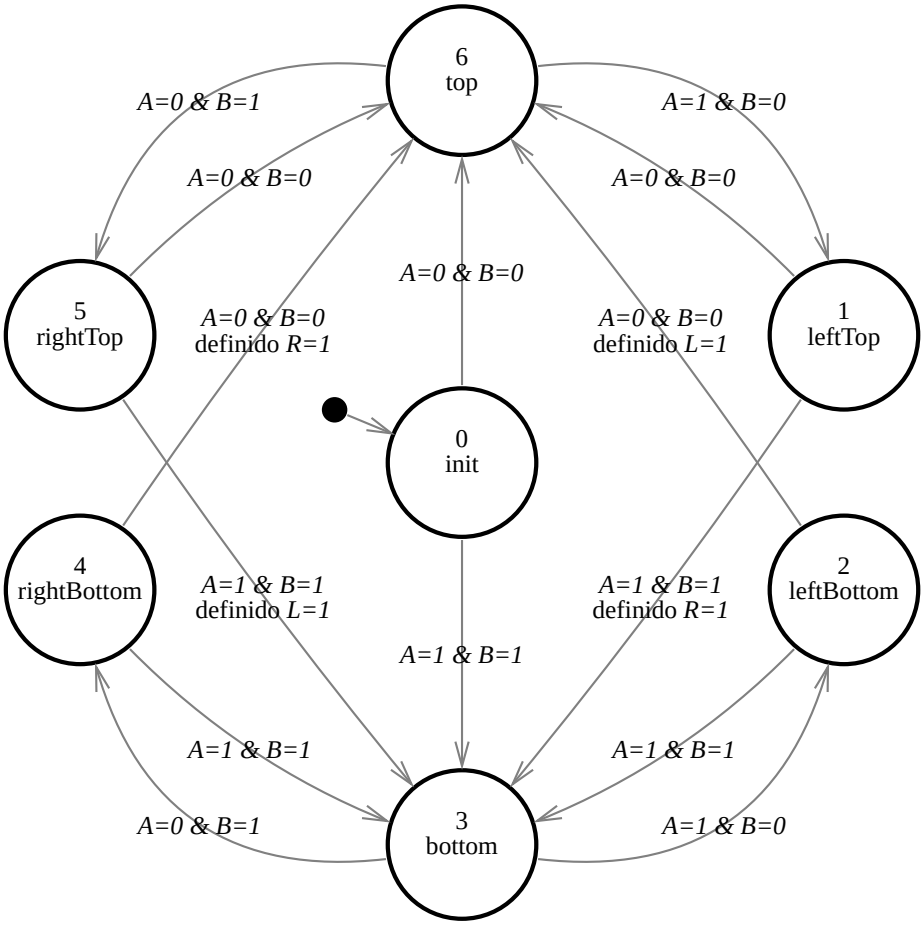

Figure 27: exemplo de FSM Mealy - decodificador rotacional

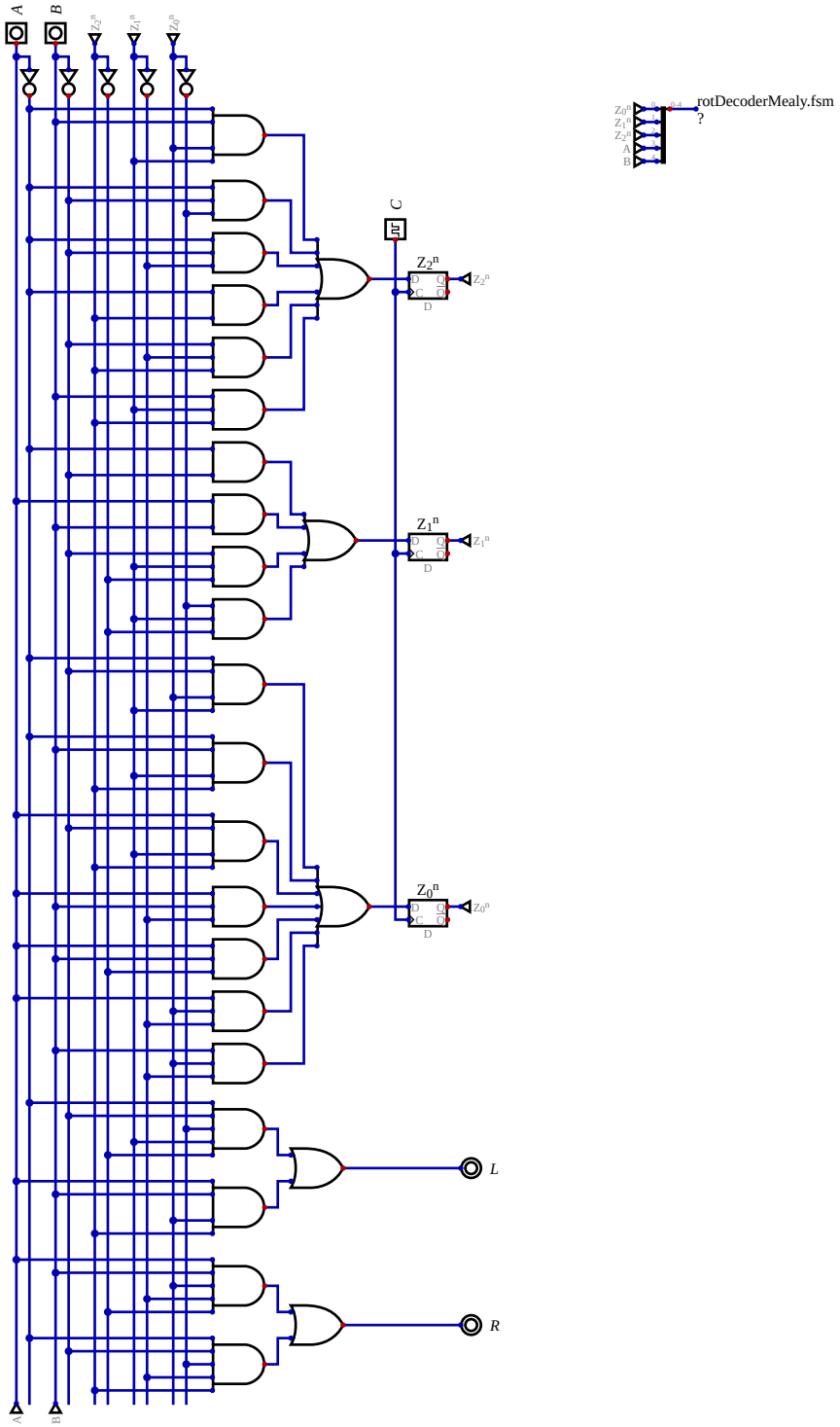

Figure 28: exemplo de FSM Mealy - decodificador rotacional

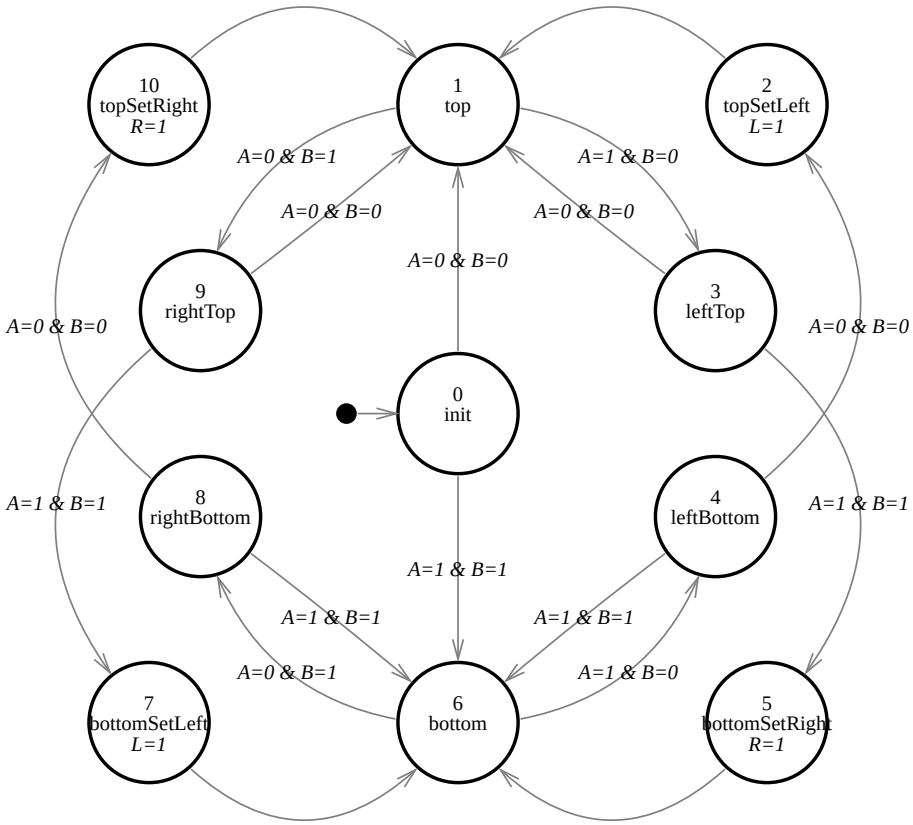

Figure 29: exemplo de FSM Moore - decodificador rotacional

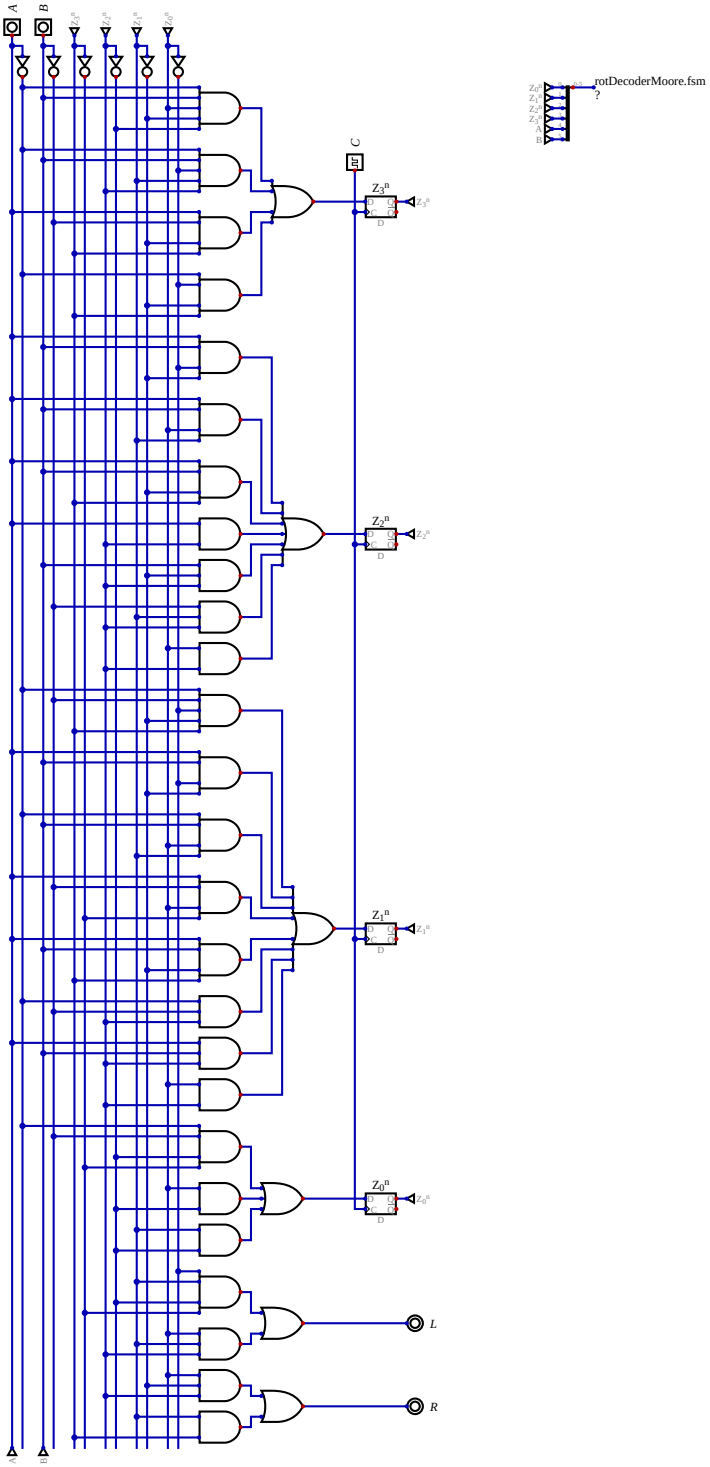

Figure 30: exemplo de FSM Moore - decodificador rotacional

3. Processadores

4. FPGAs e Shin JAMMA

5. Vídeo e Áudio

6. Pegasus 42

A. História