

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 simples 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 a 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 leve 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 apresentadas 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 retângulo 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 dígitos
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. Nossa projeto é digital.

A humanidade tem usado vários sistemas digitais diferentes para representar números, sendo o mais popular o sistema posicional decimal com dígitos indiárabicos. Quanto mais valores cada dígito pode ter, mais sensível fica aos ruídos. Mas quanto menos valores cada dígito 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 dígito só pode ser 0 ou 1. Existem várias áreas da matemática que são equivalentes quando são usados apenas dois valores.

Área	1	0	inversão	soma ou	produto e
Álgebra Booleana	verdade	falso	não		
Lógica de Predicados					
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.

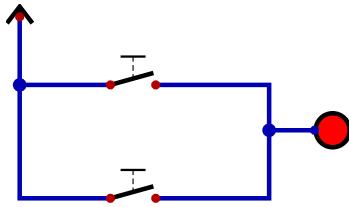


Figure 2: chaves paralelas

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.

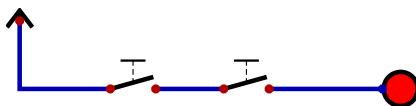


Figure 3: chaves em série

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* a 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 três 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.)

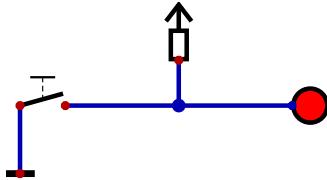


Figure 4: inversão com chave

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.

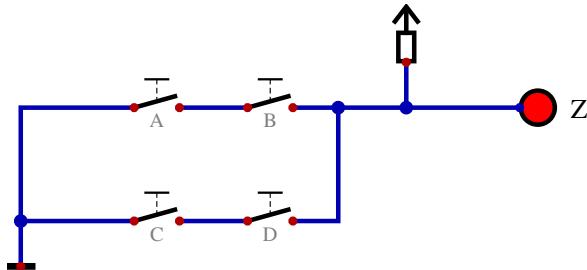


Figure 5: e, ou, inversão com chaves

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 incluída em muitas bibliotecas de projeto de circuito integrado. Em álgebra Booleana temos:

$$Z = ! (A \cdot B + C \cdot D)$$

enquanto na lógica de predicados seria:

$$Z = \text{não}((A \wedge B) \vee (C \wedge D))$$

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.

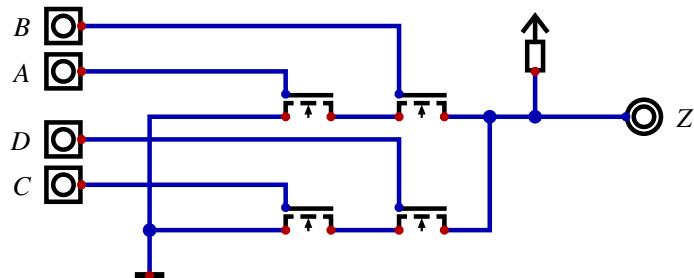


Figure 6: e, ou, inversão NMOS

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 saída.

No menu “Análises”, o ítem “Analyses” 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.

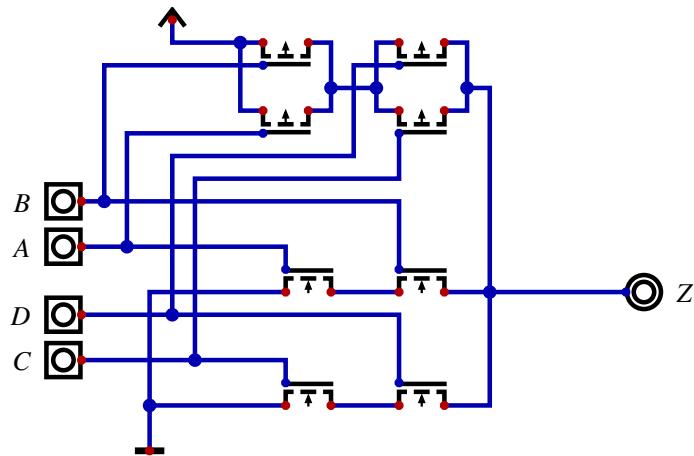


Figure 7: e, ou, inversão CMOS

A complexidade adicional do CMOS limitou 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 $2^2 = 4$ tabelas verdade 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 simples 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 Karnaugh, 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.

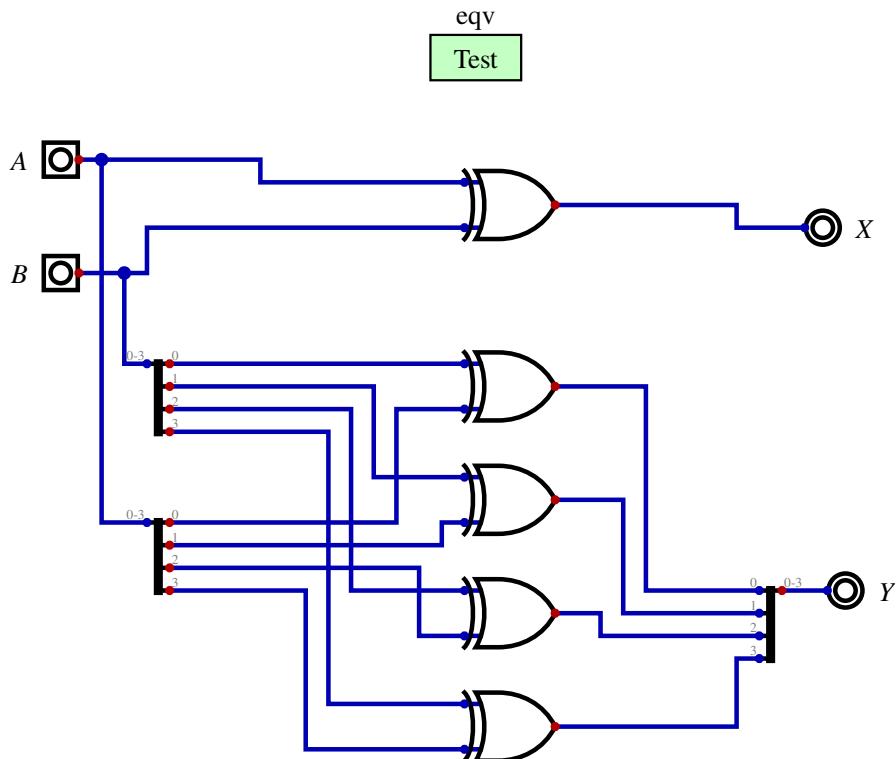


Figure 8: múltiplos bits

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 $a \text{ XOR } b$. 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 onde ocorreu a falha.

Outra maneira que o *Digital* elimina complexidade é o uso de projetos hierárquicos. 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.

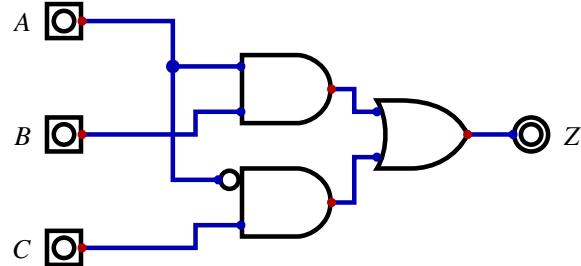


Figure 9: multiplexador de 2 entradas

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 (18×256) 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 8×256 testes precisamos de apenas 8×9 . 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 nível 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.

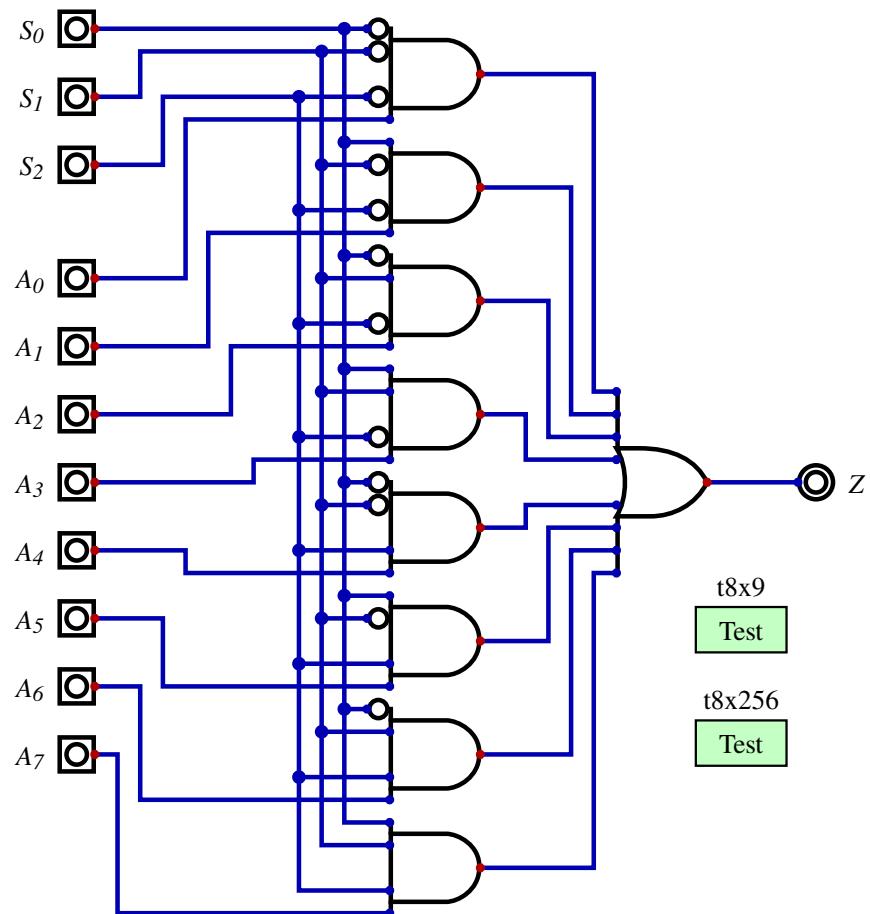


Figure 10: multiplexador de 8 entradas



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 dígitos 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 dígioto 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 $3 \times 100 + 0 \times 10 + 7 \times 1$, o número 1010 binário significa $1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$ que é o número dez. No sistema posicional decimal o dígioto mais da direita é o da unidade e cada dígioto para a esquerda vale dez vezes mais. No sistema posicional binário o dígioto mais da direita é o da unidade e cada dígioto para esquerda vale duas vezes mais. O número 10 em binário é $1 \times 2 + 0 \times 1$ 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 dígioto logo à

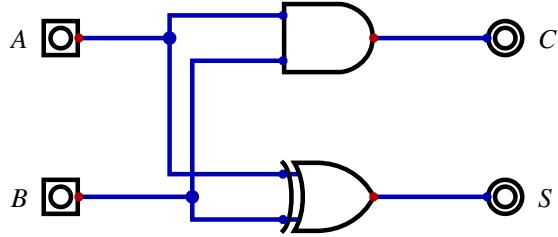


Figure 12: meio somador

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.

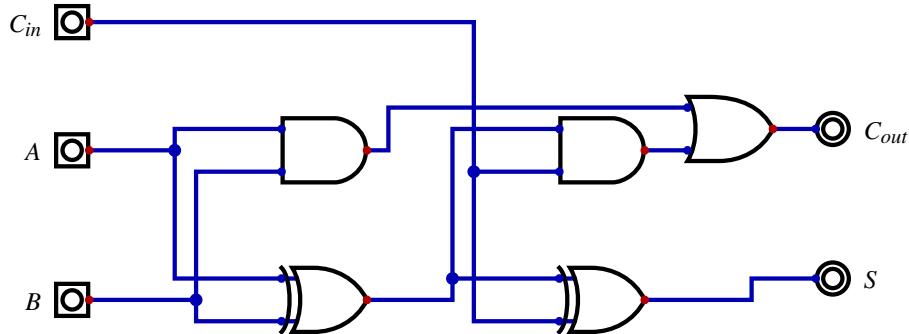


Figure 13: somador completo

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 dígito em dígito e só chega no dígito mais significativo com muito atraso. Se dobrarmos

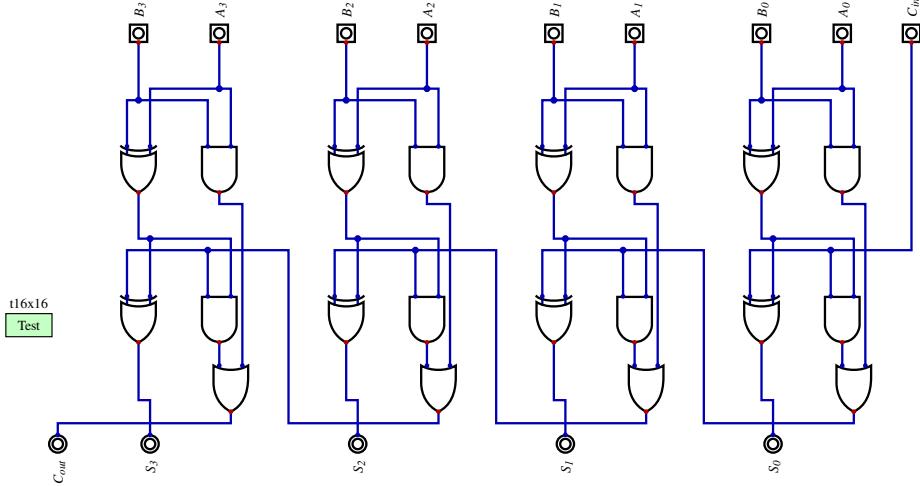


Figure 14: somador de 4 bits

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 calculadora 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 subtraímos 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 vies (“bias” em inglês). Aqui somamos um valor a todos os números para serem todos positivos. Escolhendo o deslocamento como sendo metade do maior número possível mais um, a soma de dois números transforma 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
```

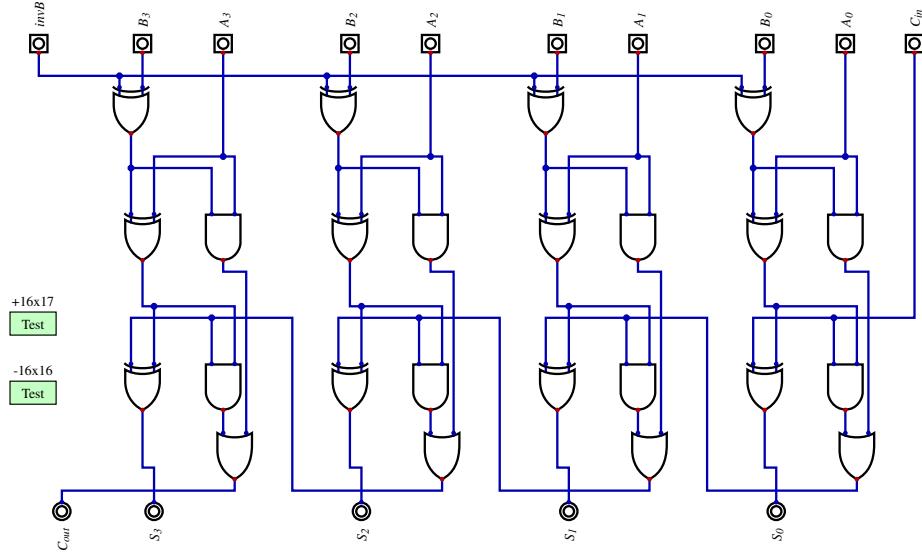


Figure 15: somador e subtrator de 4 bits

```

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.

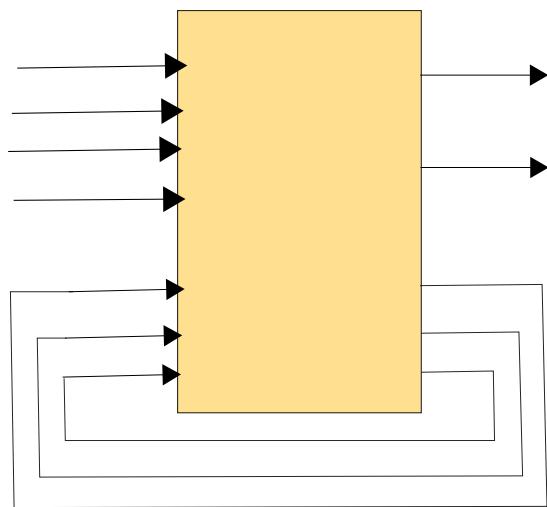


Figure 16: sistema sequencial

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”).

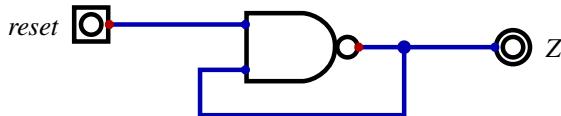


Figure 18: oscilador com nand

Se tentarmos simular parece funcionar, mas assim que mudarmos *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.

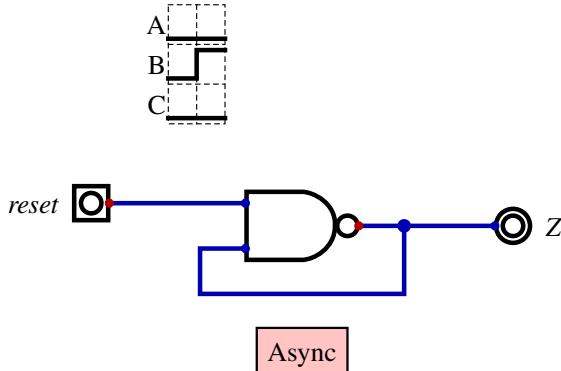


Figure 19: oscilador com nand

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

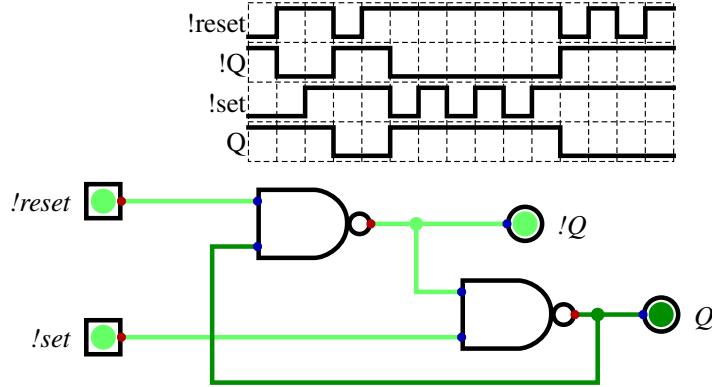


Figure 20: flip-flop com nands

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. Pulses adicionais não fazem nada. A mesma coisa para pulses em *!reset*.

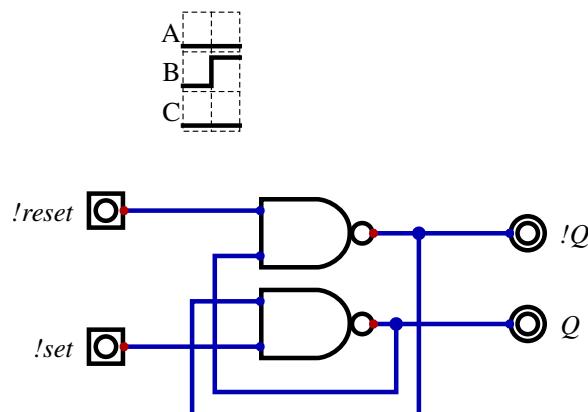


Figure 21: flip-flop com nands

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.

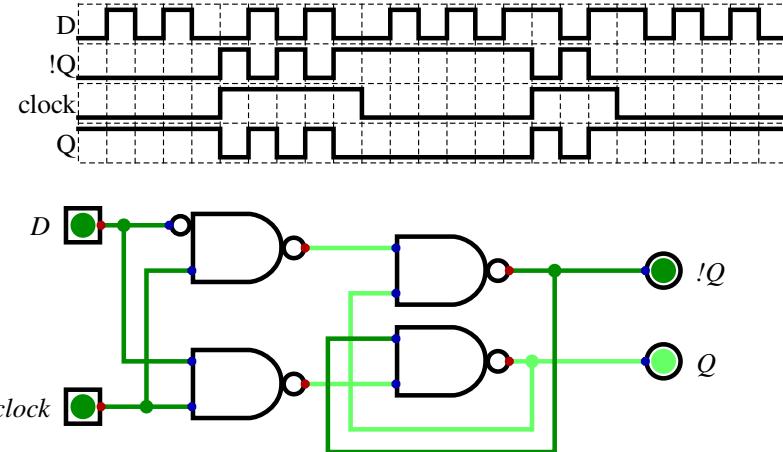


Figure 22: flip-flop tipo D

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

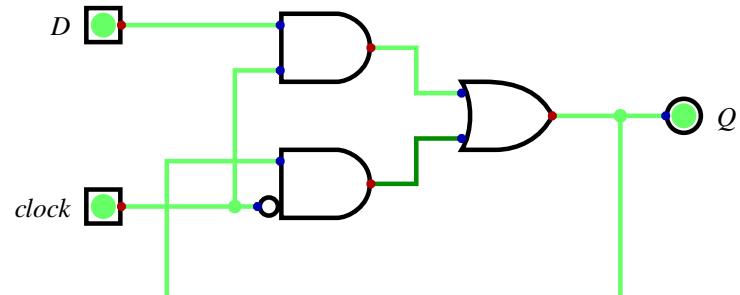
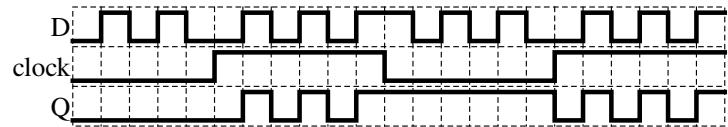


Figure 23: latch

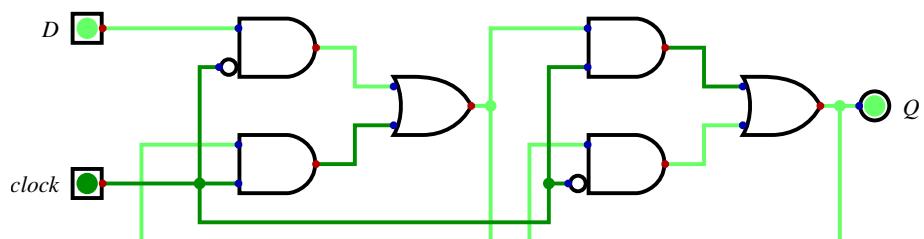
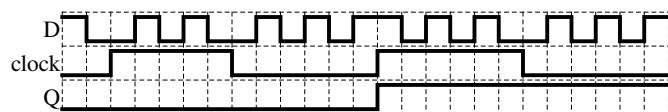


Figure 24: flip-flop tipo D sensível à borda de subida

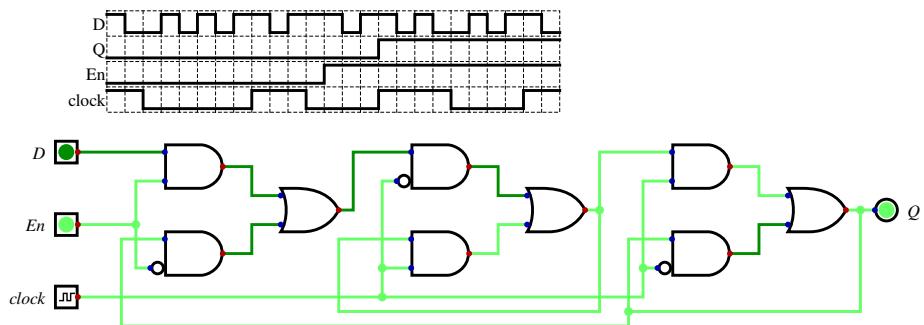


Figure 25: registrador

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 saídas 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.

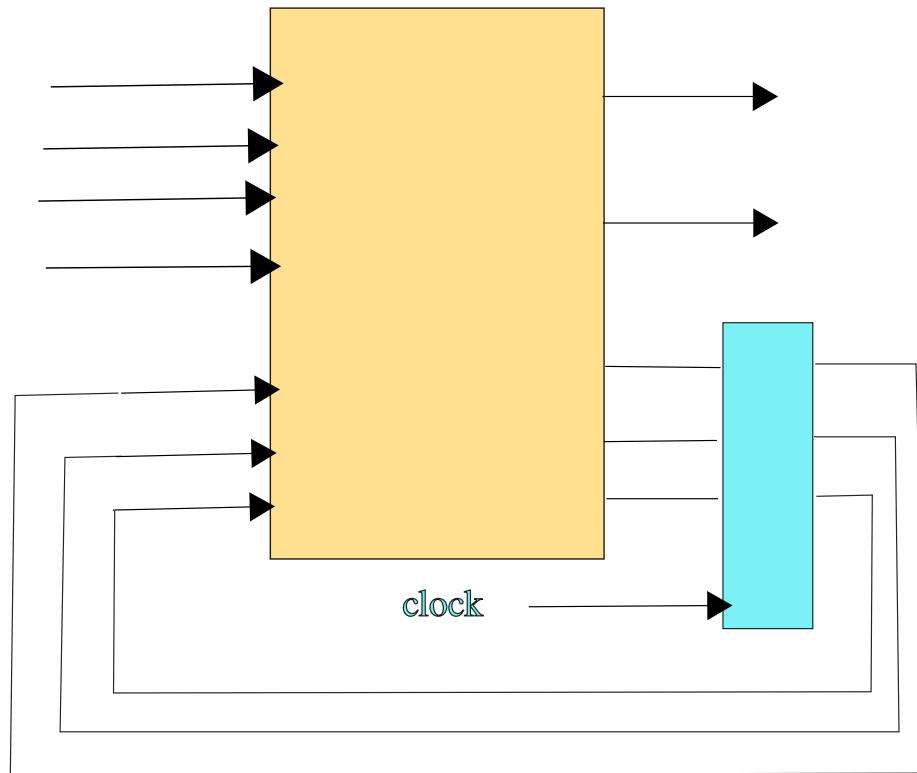


Figure 26: máquina de estados finitos

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*.

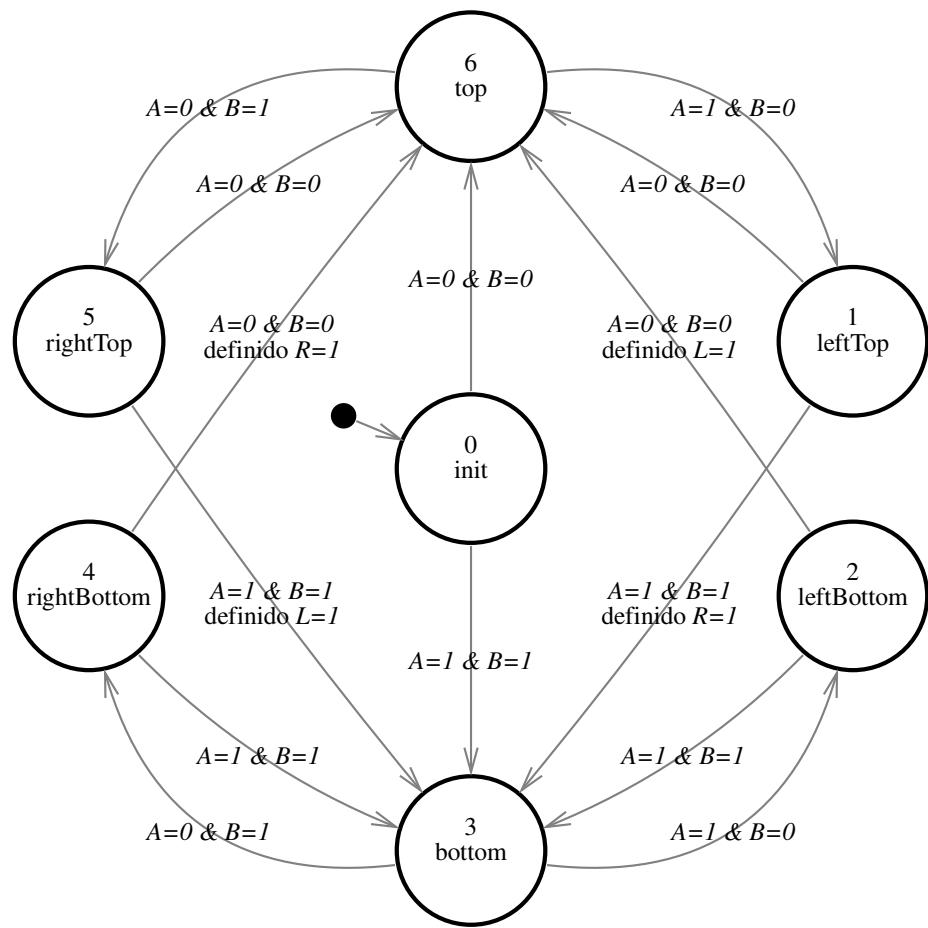


Figure 27: exemplo de FSM Mealy - decodificador rotacional

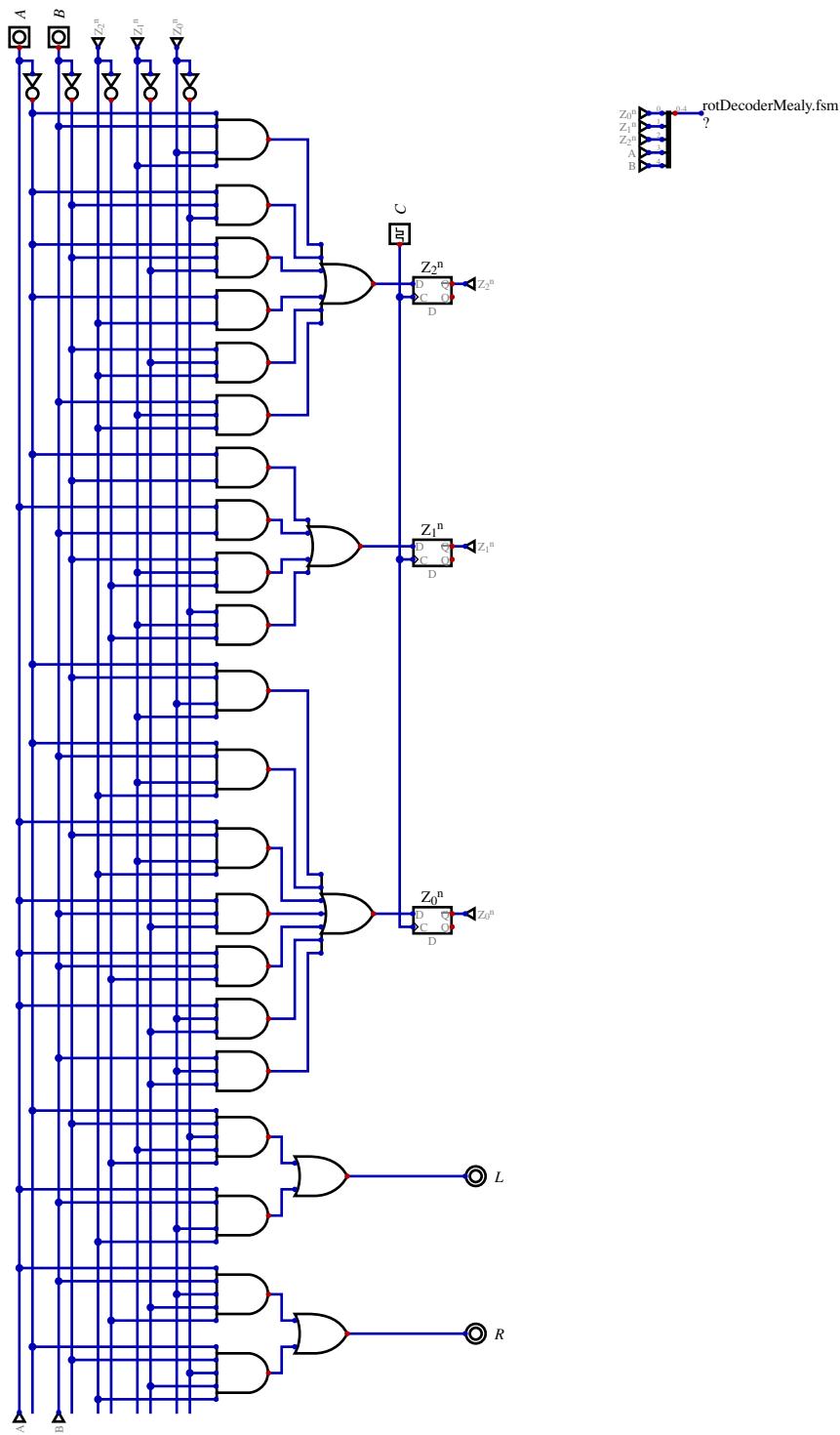


Figure 28: exemplo de FSM Mealy - decodificador rotacional

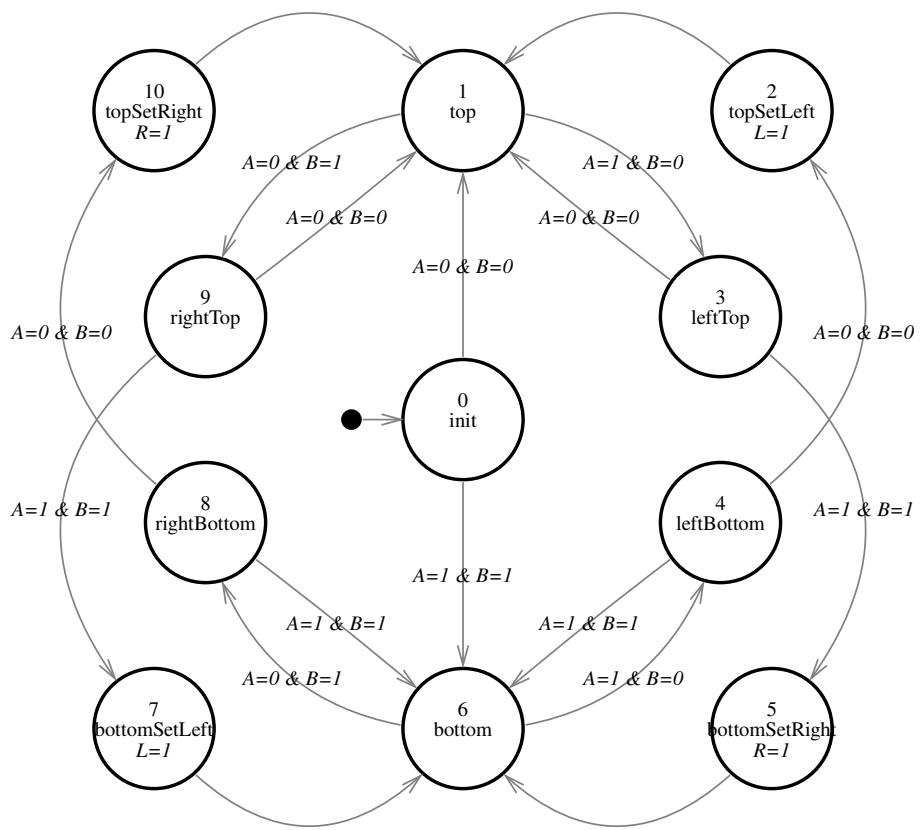


Figure 29: exemplo de FSM Moore - decodificador rotacional

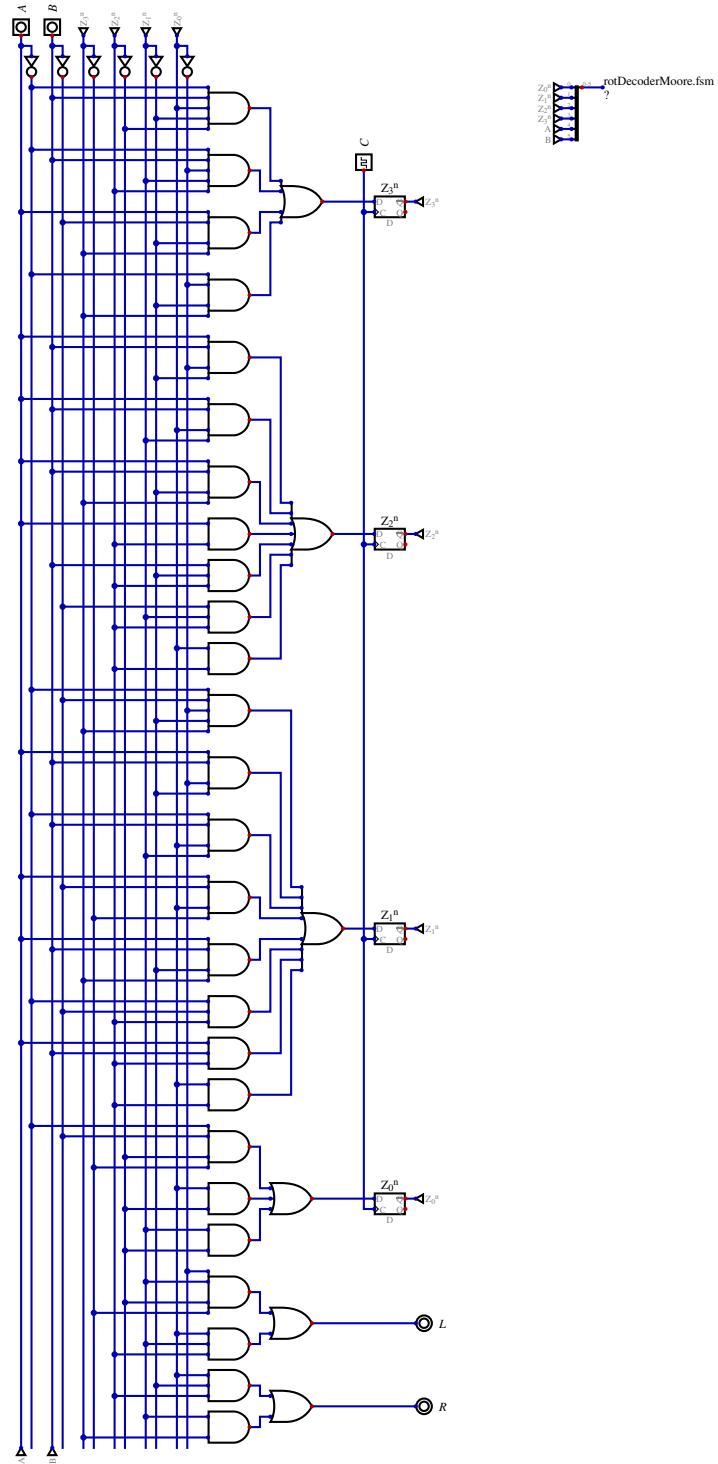


Figure 30: exemplo de FSM Moore - decodificador rotacional
31

3. Processadores

Todos os problemas computacionais podem ser resolvidos com uma FSM. Teoricamente. Na prática até problemas relativamente reduzidos podem precisar de um número de estados absurdo e a máquina correspondente seria impossivelmente cara para se construir (se não absolutamente impossível, se precisa de mais componentes do que existem átomos no universo, por exemplo).

Imagine uma FSM para receber seis caracteres de 7 bits cada cuja função é imprimir estes caracteres na ordem invertida. Ela vai ter 8_865_353_597_185 (8 trilhões) de estados. O problema é que a única memória do sistema é o registrador do estado atual e usar isso para guardar que caracteres já forma vistos não é eficiente. $6 \times 7 = 42$ bits para guardar os caracteres enquanto 8 trilhões de estados precisam de 44 bits para representá-los.

Se usarmos uma FSM ligada a uma pequena memória externa o resultado seria bem melhor. Uma memória de 8 palavras de 8 bits cada estaria sobrando e uma FSM com 12 estados seria suficiente para controlá-la para resolver o problema.

Em seu trabalho de 1936 o Alan Turing imaginou algo ainda mais simples que uma memória: ele ligou a FSM dele (que representou como tabelas no texto) a uma fita infinita com casa individuais que podem conter um símbolo escolhido entre um certo alfabeto. Existe uma cabeça de leitura e gravação que está posicionada numa das casas da fita. A entrada da FSM é o símbolo da casa atual da fita e as saídas são um símbolo a ser gravado (possivelmente o mesmo se não quisermos alterar a fita neste momento) e opcionalmente um comando para mover a cabeça para a casa da esquerda ou da direita.

Hoje isso é conhecido como “Máquina de Turing”. Um simulador muito interessante disponível na web é (<https://turingmachine.io/>) que já inclue vários exemplos.

Aqui temos uma multiplicação de dois números binários. A mesma coisa como uma FSM pura teria um número enorme de estados enquanto aqui só foram necessários 21 estados. Mas isto foi criado apenas para estudar um problema matemático do tipo “existe uma Máquina de Turing capaz de ...” e não ser algo prático. Cada problema requer a construção de uma Máquina de Turing própria, mas no final do texto aparece uma proposta interessante: uma Máquina Universal de Turing que receberia na mesma fita os dados de entrada e uma representação, na forma de uma sequência de símbolos, de uma Máquina de Turing que resolva o problema desejado. Hoje chamamos isso de “interpretador”. Uma vez construída esta máquina, trocando a fita muda o funcionamento. É o que chamamos de “software”.

Arquitetura de von Neumann e de Harvard

O primeiro computador (ou “cérebro eletrônico” conforme a imprensa da época) divulgado para o público foi o Eniac em 1946 (projetos anteriores foram mantidos

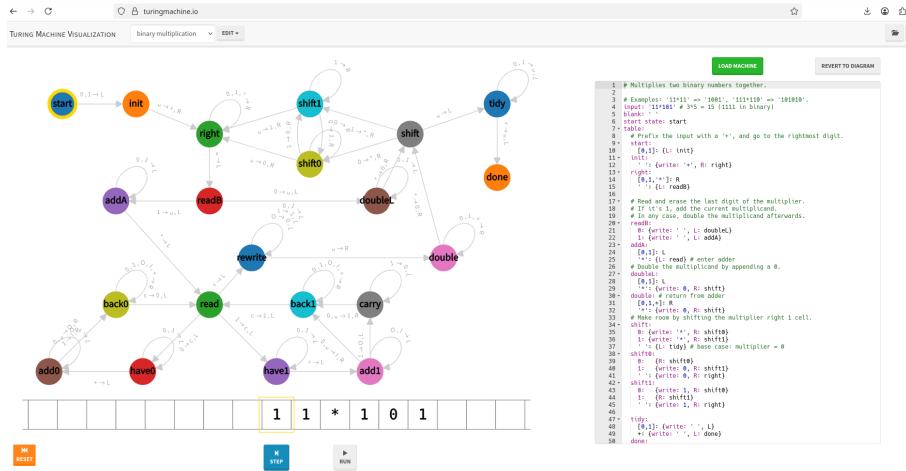


Figure 31: simulador de Máquinas de Turing

em segredo por muitos anos). Projetado por John Mauchly e J. Presper Eckert na Universidade da Pennsylvania, a principal limitação do ENIAC era a necessidade de reconfigurar o hardware via painéis de fios para cada novo problema. Então mesmo durante o seu desenvolvimento eles passaram a discutir o sucessor, a ser chamado de EDVAC.

Um dos participantes destes debates era o John von Neumann e ele escreveu um relatório detalhado com estas idéias. Outro participante, o Herman Goldstine, acabou distribuindo o relatório para grupos externos com apenas o von Neumann como autor, por isso este estilo de computação é conhecido como “arquitetura von Neumann” apesar de ter sido criada por um grupo.

O John von Neumann gostava de analogias com a biologia e por isso chamava as partes do computador de “órgãos” e onde os dados ficavam de “memória” (outros, especialmente a IBM, preferiam termos como “armazenagem” mas acabaram perdendo esta batalha).

A unidade de control é uma FSM, que já vimos. A unidade lógica e aritmética é uma versão mais complicada so somador/subtrator que também já vimos. A entrada e a saída é diferente para cada computador, então vamos ignorá-los por enquanto.

A memória guarda dados e os programas (como a fita da Máquina Universal de Turing). Um bit desta memória é como o registrador, que já vimos. O que não falamos é como escolher um bit entre muitos, mas o multiplexador é bem parecido com o mecanismo usado. Um aspecto interessante da arquitetura de von Neumann é que o processador central (CPU) pode ser feito com uma tecnologia completamente diferente da usar pela memória. Alguns exemplos:

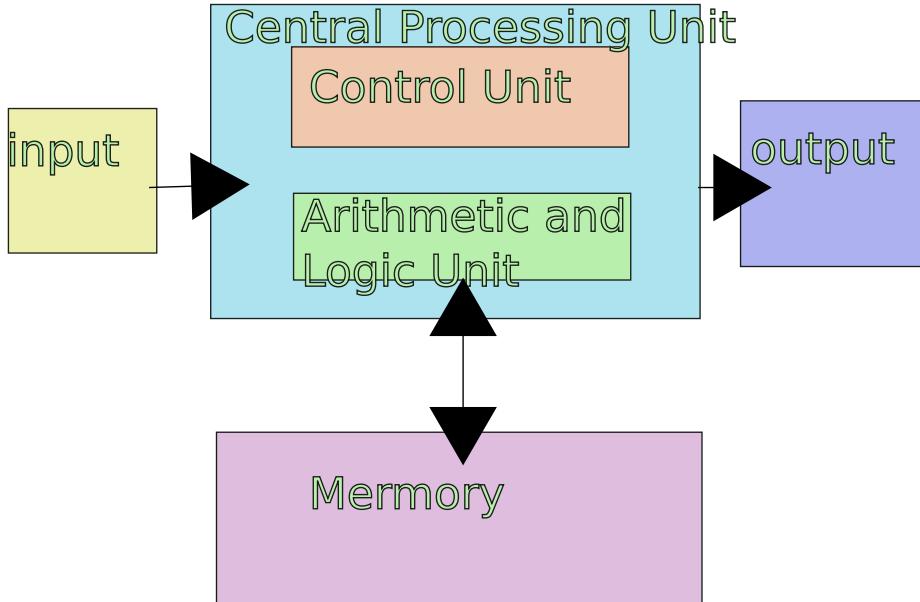


Figure 32: arquitetura von Neumann

Computador	CPU	Memória
EDSAC	válvulas	tanques de mercúrio
IAS	válvulas	tubos de Williams
LGP-30	válvulas	tambor magnético
PDP-8	transistores	núcleos de ferrite
PC moderno	chip digitais	chip com capacitores verticais

O IAS é “Institute for Advanced Studies” de Princeton onde John von Neumann e equipe construiram o computador do seu relatório. Por isso também é mais raramente conhecido como “arquitetura de Princeton”. Isso é em contraste com “arquitetura de Harvard” cujo nome é baseado no computador que a IBM construiu para Howard Aiken da Universidade de Harvard. A única diferença é que a memória de dados e a memória de programas são separadas. A separação permite uma instrução e um dado serem lidos ao mesmo tempo, o que facilita o projeto. Mas impede que um programa modifique a si mesmo.

Hoje todos os computadores menos os mais simples são híbridos: diretamente no processador temos duas pequenas memórias conhecidas como cache de instrução de nível 1 e cache de dados de nível 1. Estas memórias apenas guardam cópias das informações mais recentes usadas pelo computador. Quando a informação desejada não está nelas, um cache unificado de nível 2 é acessado. E pode existir um nível 3 de cache e finalmente chegamos à memória principal única como na

figura do von Neumann. Isto combina as vantagens de hardware da arquitetura Harvard com as vantagens de programação da arquitetura von Neumann.

Mencionamos que as entradas e saídas são específicas para cada computador. Nos primeiros computadores isso se refletia no conjunto de instruções. Um computador podia ter uma instrução para ler uma tecla e outra instrução para gravar numa fita. Por volta de 1970 começou a ficar popular a idéia de fazer os dispositivos aparecerem como posições especiais de memórias. Ai o processador podia usar as instruções “normais” para tudo. Dos processadores mais usados, apenas o x86 (Intel e AMD) ainda usam instruções especiais para entrada e saída e os outros usam “periféricos mapeados na memória”, o que nós também faremos.

MCPU16h

No relatório do EDVAC foi apresentada a idéia de representar um programa como uma série de números, onde os bits mais significativos representariam a “ordem” sendo dada ao computador (hoje chamamos isso de “código de operação”, ou “opcode” reduzido em inglês) e os menos significativos o endereço da posição de memória a ser usada nesta instrução. Muitas operações precisam de dois operandos e produzem um resultado que precisa ser armazenado em algum lugar. Isso exigiria 3 endereços, mas o EDVAC herdou das calculadoras mecânicas a idéia de um registrador especial chamado “acumulador” que serve de um dos operando e o destino do resultado na maioria das instruções.

Quantas instruções diferentes vamos ter? Isso define quantos bits iremos precisar para o “opcode”. Na verdade é possível fazer tudo com uma única instrução, mas isso não é uma boa idéia do ponto de vista didático. Um exemplo é as instrução SUBLEQ que tem 3 endereços, e subtrai o valor no primeiro endereço do que está no segundo endereço (guardando o resultado ali) e se o resultado foi negativo ou igual ele salta para o terceiro endereço. Programas escritos para um computador assim são quase tão difíceis de se entender quanto os da Máquina de Turing.

Uma alternativa mais razoável é o MCPU com suas 4 instruções (dois bits para o “opcode”). O Tim Böscke foi inspirado pelo MPROZ com suas 3 instruções, mas a troca das operações de memória para memória por um acumulador estilo EDVAC simplificou bastante o projeto.

O MCPU16h tem duas diferenças: enquanto o MCPU original tem 8 bits de largura e só sobravam 6 bits para o endereço (apenas 64 bytes, o suficiente para os exemplos mais simples) o MCPU16h, como o nome diz, é de 16 bits e os 14 bits de endereço permitem 16 mil palavras de 16 bits cada uma (32KB). E o “h” no fim do nome é de Harvard. Só que como o EDVAC, o MCPU16h depende de poder modificar um programa enquanto executa mas isso não é possível com a arquitetura Harvard. Só que o *Digital* tem uma memória de duas portas e vamos usar isso para parecer que são duas memórias separadas mas o que é escrito numa pode ser lido na outra.

A escolha da arquitetura Harvard é para permitir o uso de um circuito combinacional para a unidade de controle. Na arquitetura von Neumann (como o MCPU original) a leitura da instrução tem que ocorrer num ciclo e a leitura de um operando no outro. Por isso a unidade de controle precisa ser um circuito sequencial, que é um pouco mais complexo.

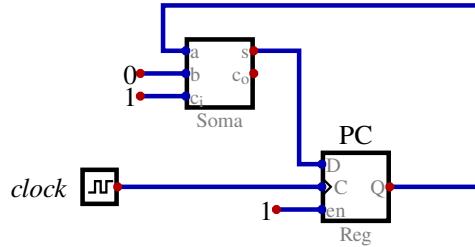


Figure 33: PC do MCPU

Com este circuito simples, a cada borda de subida do relógio o PC avança para a próxima palavra. Neste caso ligamos o “enable” do registrador PC em 1 já que nunca deixamos de alterar o PC. Mas num projeto mais completo vão existir situações em que isso ocorre (se tivermos que esperar pela memória de instruções, por exemplo).

Vamos usar a memória com duas portas. A porta 2 que é limitado à leitura vai servir como memória de instruções. Os sinais de controle da porta 1 estão ligados a valores constantes que não atrapalhem a operação (não gravando nada, por exemplo).

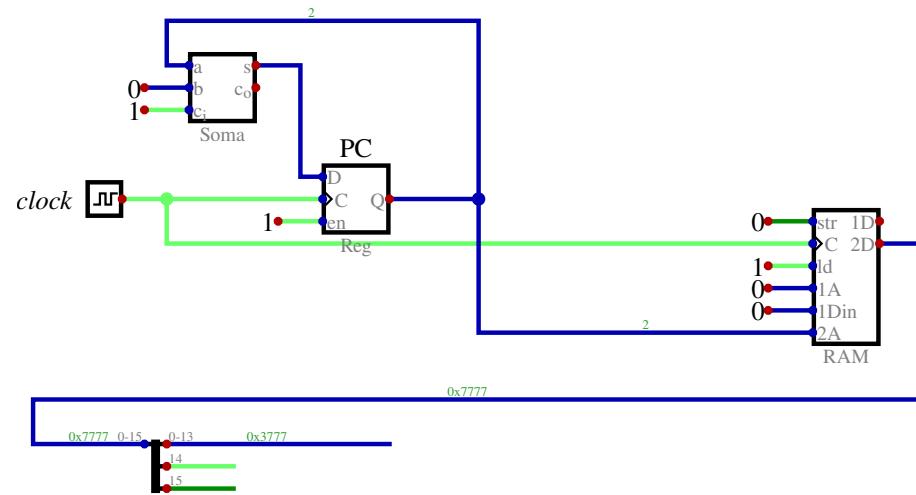


Figure 34: memória de duas portas

Quando a simulação começa a memória está cheia de 0s. Mais para a frente

usaremos o conteúdo de um arquivo para preencher a memória, mas por enquanto estamos editando os valores das primeiras palavras da memória cada vez que a simulação começa só para ver que algo está acontecendo. A figura mostra o que acontece depois de duas subidas do relógio.

O dado vindo da memória (0x7777 indicando que é o hexadecimal equivalente ao binário 0111011101110111) é dividido num endereço de 14 bits (0x3777) e em dois bits de “opcode” (0 e 1). Estamos decodificando as instruções usando apenas fios.

Vamos implementar a primeira das quatro instruções: JCC (“Jump if Carry Clear” - pule se o vai um estiver 0). Precisamos escolher um “opcode” para esta instrução e escolhemos 1 e 1.

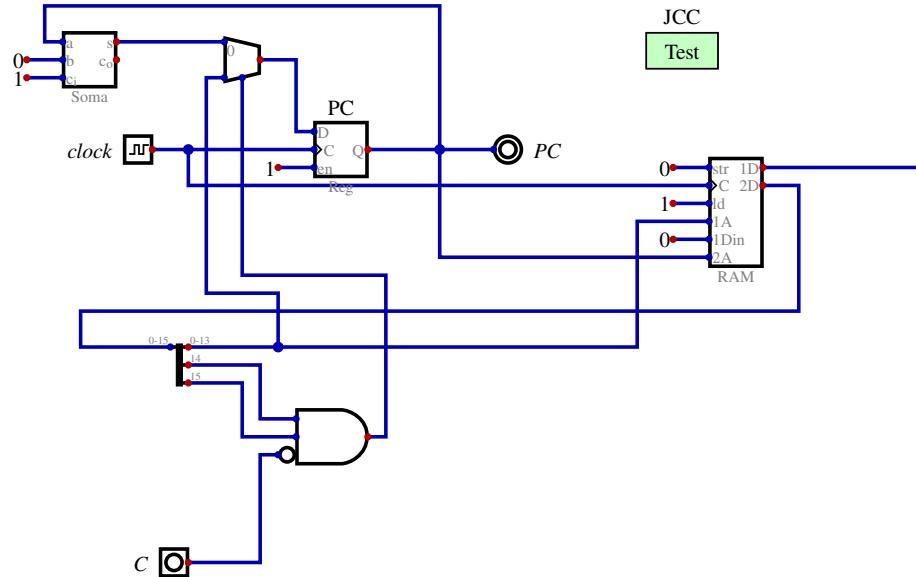


Figure 35: instrução JCC

Um multiplexador escolhe entre pular para o endereço indicado pela instrução ou somar 1 ao PC atual. O primeiro caso só acontece se o opcode é o do JCC e se o bit C é 0. A porta AND de 3 entradas determina esta situação e controla o multiplexador.

Aqui estamos fazendo um único circuito, mas normalmente a porta AND faria parte da unidade de controle enquanto o multiplexador, registrador de PC e o somador fariam parte do que chamamos de “fluxo de dados”. E a memória seria algo separado da CPU.

```
clock C PC
program(0x0000, 0xC006, 0xC001, 0xC008)
0 1 0
```

```

C 1 1
C 1 2
C 0 1
C 1 2
C 0 1
C 0 6

```

Para testar o circuito precisamos tornar o PC uma saída para que possa ser comparado durante o teste. Inicializamos as 4 primeiras palavras da memória com um pequeno programa de teste. A instrução 0 vai ser ignorada por enquanto e as instruções 1, 2 e 3 são todas JCC. A primeira vez que a instrução 1 é executada (linha 5 do teste) o C é 1 e PC é incrementado para 2. Na linhas 7 e 9 a instrução 1 é executada outra vez, com C igual a 1 e C igual a 0 respectivamente. Apenas no último caso ele não continua para 2, mas sim salta para 6.

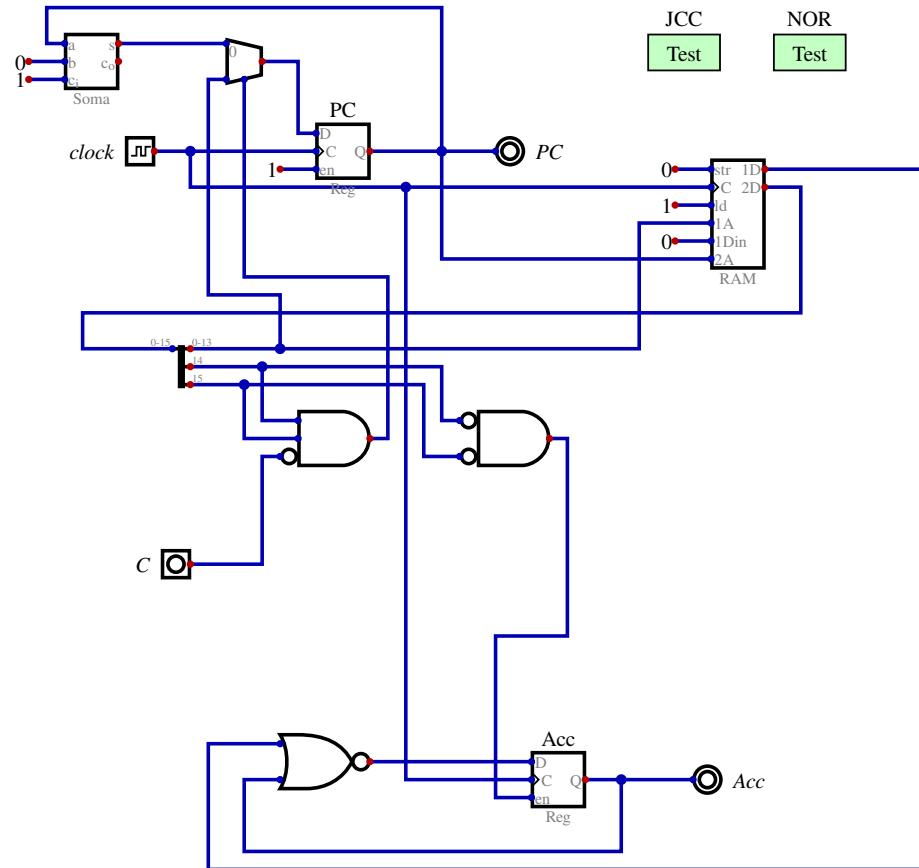


Figure 36: instruções JCC e NOR

A próxima instrução que vamos implementar é a NOR, para a qual escolhemos o opcode 0 e 0. Para isso precisamos de um novo registrador de 16 bits que chamaremos de Acc. Teremos uma saída com este nome para poder usar nos testes. O circuito NOR com largura 16 recebe um dado de Acc e o outro da memória e seu resultado volta para Acc.

Primeiro testamos JCC para ver se continua funcionando. Depois criamos um novo teste para NOR. O circuito precisa passar pelos dois.

```
clock Acc
program(0x0004, 0x0000, 0x0001, 0x0002, 0xFFFF)
0 0
C 0
C 0xFFFFB
C 4
C 0xFFFFA
```

Curiosamente as palavras 0, 1 e 2 servem inicialmente como instruções mas depois são usadas como dados também.

A instrução STA (“STore Accumulator”) vai ter o opcode 1 e 0. A saída de Acc vai ser o dado de entrada da memória (que até agora era sempre 0). E o sinal *str* da memória vai ser acionado para esta instrução, gravando o dado no endereço indicado.

Depois de verificar que os testes de JCC e do NOR continuam funcionando, criamos um novo teste para STA.

```
clock Acc
program(0x0004, 0x8004, 0xC000, 0x0000, 0xFFFF)
0 0
C 0
C 0
C 0
C 0xFFFF
C 0xFFFF
C 0xFFFF
C 0
C 0
C 0
C 0xFFFF
```

Não é possível testar STA sem também usar as outras instruções. Os resultados de gravar um dado só é visível quando se lê de volta o dado e para isso precisamos do NOR. O teste usa JCC também só para manter o código curto - uma sequência NOR, STA, NOR, STA, NOR... já serviria de teste.

A última instrução, o ADD, seria quase igual ao NOR se não fosse pela complicação do vai um. Mas como tanto o ADD quanto o NOR escrevem em Acc precisamos de um multiplexador para decidir entre eles.

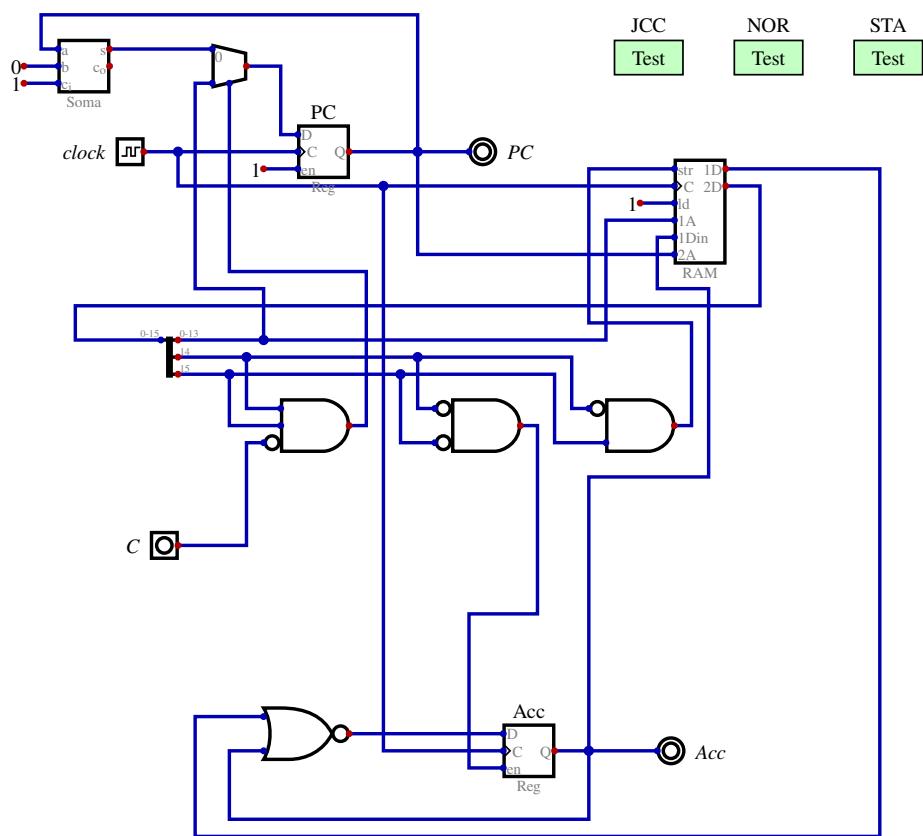


Figure 37: instruções JCC, NOR e STA

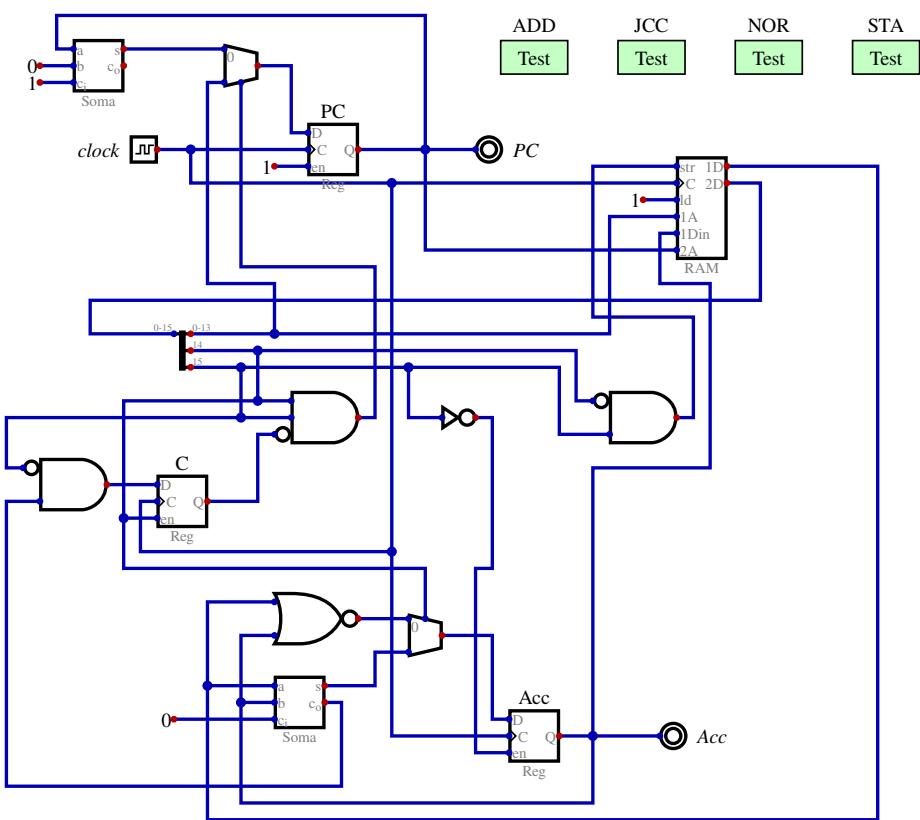


Figure 38: mcpu completo

Agora Acc precisa ser habilitado tanto para ADD (opcode 01) quanto para NOR (opcode 00), então podemos trocar a porta AND com as duas entradas invertidas por um inverso. O novo somador tem as mesmas entradas que o NOR.

A entrada C é trocada por um registrador C de apenas um 1 bit de largura. Isso vai nos obrigar a alterar o teste JCC. A definição de JCC é que C é zerado independentemente se o salto acontece ou não. Isso é usado para que duas instruções JCC seguidas sejam um desvio incondicional (que em outros processadores seria uma instrução separada). A saída vai um do somador é forçada a ser 0 no caso de uma instrução JCC (ou de um STA, mas neste caso não importa).

C precisa ser habilitado para as instruções ADD (opcode 01) e JCC (opcode 11), então basta usar diretamente o bit mais de baixo do opcode.

Com isto o MCPU está completo e poderá executar programas complexos.

Software

Já escrevemos pequenos programas para o MPCU16h nos testes que criamos. Mas indicar diretamente cada instrução como um número hexadecimal ou binário, o que chamamos de “linguagem de máquina”, fica inviável para programas com mais de umas 10 instruções. Felizmente um programa chamado “montador” (“assembler” em inglês) pode ler um texto em que cada linha corresponde a uma instrução em linguagem de máquina e letras indicam a operação (“NOR”, “ADD”, “STA” e “JCC” no caso do MCPU16h) e gerar a linguagem de máquina correspondente. Podemos marcar algumas linhas usando rótulos textuais e ai usar este mesmo texto no campo de endereço de outras instruções e ai o assembler cuida de calcular o valor real dos endereços associados aos rótulos.

Montadores mais avançados, como o *GNU as*, permitem a definição de “macros” que são textos (possivelmente com parâmetros) que depois são expandidos em todos os lugares onde são usados no programa. Neste projeto usamos este recurso para fazer uma versão do *as* (para o processador x86, por exemplo) gerar código de máquina para o MCPU16h. Estas macros ficam no arquivo *mcpu16.inc* que tem em seu início:

```
absStart:  
    .macro NOR a  
    .word (0x3FFF & ((\a-absStart)/2))  
    .endm  
  
    .macro ADD a  
    .word (0x3FFF & ((\a-absStart)/2)) | 0x4000  
    .endm  
  
    .macro STA a  
    .word (0x3FFF & ((\a-absStart)/2)) | 0x8000
```

```

.endm

.macro JCC a
.word (0x3FFF & ((\a-absStart)/2)) | 0xC000
.endm

```

Um programa que inclua este arquivo vai poder escrever algo como “JCC myLoop” e os 16 bits correspondentes serão gerados. Uma complicação é que *as* encara que um rótulo como “myLoop” é um valor relativo e não conseguiria, em princípio, calcular a expressão que gera a instrução. A idéia é que um programa pode ser feito de vários módulos e o montador traduz cada um e uma etapa posterior junta os módulos no programa final. Ai os endereços só serão conhecidos depois disso. Mas os programas que escreveremos para o MCPU16h são relativamente simples e não teremos mais de um módulo. Por isso definimos “absStart” como sendo o início do programa para que a expressão “myLoop-absStart” seja um valor conhecido pelo montador.

Além das 4 instruções que o hardware entende, podemos aproveitar as macros para definir novas instruções como pequenas sequências destas instruções. Por exemplo: CLR (limpa o acumulador para zero), LDA (carrega um valor no acumulador), LDP (carrega um ponteiro no acumulador), LDI (carrega indiretamente no acumulador), NOT (inverte o acumulador), JMP (pula incondicionalmente), JCS (pula se o *C* for 1), SUB (subtrai o acumulador do valor) e CALL (chamada de subrotina). O uso destas “pseudo-instruções” gera programas um tanto grandes. Os programas do MCPU16h são umas 10 vezes maiores do que os para processadores mais razoáveis.

As pseudo-instruções IN e OUT leem e gravam do último endereço da memória e o circuito deve detectar isso e ler ou gravar de um periférico no lugar da memória. Também foram definidas as macros STARTCOUNT e COUNT para controlar um circuito para medir a velocidade para comparação de diferentes processadores. Neste projeto não implementaremos este circuito e estas pseudo-instruções irão mexer com a penúltima palavra da memória sem nenhum efeito.

O comando “make”, entre outras coisas, gera os arquivos .hex que usaremos para inicializar a memória em nossas simulações no diretório *hex* usando os códigos fontes .S encontrados no diretório *soft*. Os programas *as* e *objcopy* são usados pelo *make*. Mais para frente usaremos *riscv32-unknown-linux-gnu-as* no lugar do *as* para cuidar de programas para o RISC-V. Usaremos .s (minúsculo) nos nomes do código fonte neste caso para o *make* poder decidir qual montador usar.

O novo circuito está na parte inferior à direita. Um AND de 14 entradas indica quando o último endereço da memória (0x3FFF) é acessado e neste caso o novo multiplexador seleciona o dado vindo do teclado para as instruções NOR e ADD (enquanto STA e JCC nem usam este dado) ao invés da memória, que é selecionado para qualquer outro endereço. Duas portas AND separadas o acesso ao último endereço entre os casos de escrita (STA) e leitura (qualquer outra instrução). A escrita vai para o terminal e a leitura vem do teclado.

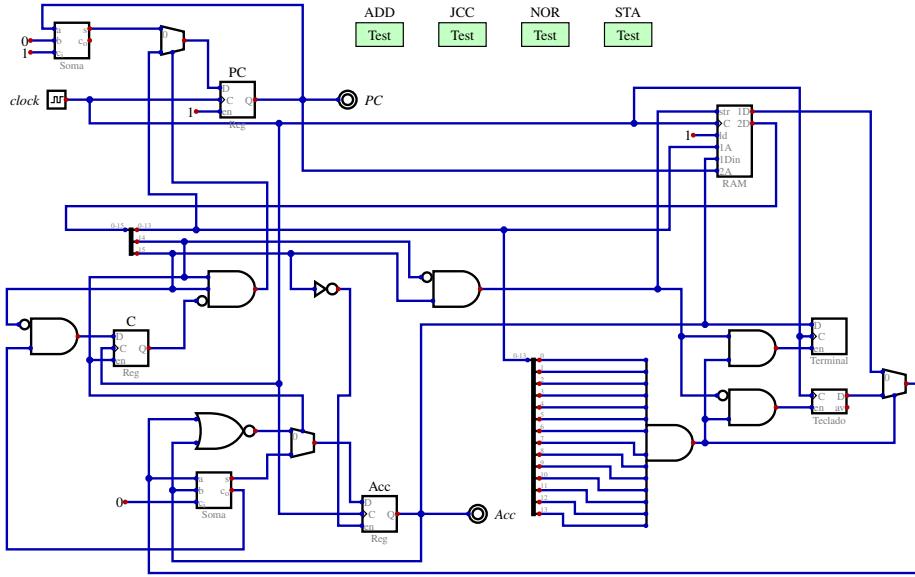


Figure 39: MCPU16h com terminal e teclado

A memória pode ser configurada como “Memória de programa” e na “Configuração específicas do circuito” podemos indicar o arquivo *hex/3.01.testTerminal.hex* como “Arquivo de programa”. Isto foi gerado de

```
.include "mcpu16.inc"

loop:
    LDA cp
    ADD adInst
    STA Of
    CLR
    0:   .word 0 /* will be replaced */
    STA char
    ADD minusOne
    JCC halt /* zero terminated string */
    LDA char
    OUT
    LDA cp
    ADD one
    STA cp
    JMP loop

halt:  JMP halt
```

```

text:    .string16 "Hello world!"
cp:      .word (text-absStart)/2
char:   .word 0

```

Podemos executar este programa passo a passo clicando repetidamente na entrada *clock*. Se o *Digital* abrir uma janela e escrever o texto esperado podemos considerar que o processador está funcionando completamente (a instrução NOR não aparece na listagem mas é usada por pseudo-instruções como LDA).

Para uma segunda execução deste programa podemos configurar a entrada *clock* para pulsar em tempo real.

Trocando a inicialização da memória por *hex/3.02.sine.hex* veremos uma senoide desenhada na tela textualmente. O algoritmo CORDIC evita usar multiplicações.

Um terceiro exemplo é o jogo interativo 2048. O terminal mostra uma matriz de 4 por 4 números que podem ser deslocado para cima, esquerda, baixo ou direita pressionando teclas na pequena janela intitulada “Teclado”. O arquivo para a memória é *hex/3.03.term2048.hex*.

drv32h

Em 2010 um grupo de pesquisa da Universidade da Califórnia Berkeley estava desenvolvendo um circuito que precisava incluir um processador. Depois de avaliarem as alternativas disponíveis eles resolveram que a melhor opção seria um projeto próprio. Afinal, este mesmo grupo tinha criado o processador RISC em 1982 e o RISC II em 1983. O SOAR (“Smalltalk On A RISC”) de 1984 e o SPUR de 1988 tinham continuado nesta tradição. Então o novo processador seria o quinto desta linhagem: o RISC-V.

Eles definiram um conjunto bem pequeno de instruções mas com espaço para ter extensões opcionais. A base pode ser de 32 bits (RV32I ou RV32E), 64 bits (RV64I ou RV64E) ou ainda 128 bits (RV128I). As variantes “I” e “E” são idênticas fora o fato da “I” ter 32 registradores e a “E” apenas 16.

Um exemplo de extensão é a “M” que adiciona instruções de multiplicação e divisão. As instruções básicas já são suficientes para implementar estas funções, mas instruções próprias podem acelerar bastante certas aplicações ao custo de um hardware mais caro. A maioria das extensões são como esta onde uma coisa que já era possível fica mais rápida. Uma extensão que traz nova funcionalidade é a “A” que introduz instruções atômicas. Sem estas instruções não dá para vários processadores ligados numa única memória coordenarem suas atividades.

Com o interesse crescente de empresas e pesquisadores fora de Berkeley pelo projeto foi criada uma fundação independente para organizar a padronização e evolução da arquitetura.

No drv32h (RISC-V no Digital com base RV32I e arquitetura Harvard) iremos implementar apenas as instruções da base e nenhuma das extensões.

Instruções

Detalhes sobre as instruções do RISC-V podem ser encontradas na especificação oficial ou nos slides deste curso de 2022.

As instruções básicas do RISC-V são de 32 bits, mas os dois bits de baixo são sempre 1 e 1. As outras 3 combinações são usadas pela extensão “C” que usa instruções de 16 bits para tornar os programas mais compactos. Os 7 bits de baixo são o código de operação e sem a extensão “C” temos 32 combinações possíveis:

	00...11	01...11	10...11	11...11
..00011	LOAD	STORE	MADD	BRANCH
..00111	LOAD-FP	STORE-FP	MSUB	JALR
..01011	cust0	cust1	NMSUB	reserved
..01111	MISC-MEM	AMO	NMADD	JAL
..10011	OP-IMM	OP	OP-FP	SYSTEM
..10111	AUIPC	LUI	reserved	reserved
..11011	OP-IMM32	OP32	cust2	cust3
..11111	48 bits	64 bits	48 bits	>=80 bits

Apenas iremos implementar as instruções individuais JALR, JAL, AUIPC e LUI e os grupos de instruções LOAD, STORE, BRANCH, OP-IMM e OP. Estas são apenas 9 das 32 possibilidades do código principal de operação. Os grupos de instruções usam campos auxiliares de código de operação para escolher as instrução individual.

As instruções MADD, MSUB, NMSUB e NMADD e os grupos LOAD-FP, STORE-FP e OP-FP são para as extensões de ponto flutuante: “F” (ponto flutuante de 32 bits), “D” (ponto flutuante de 64 bits) e “Q” (ponto flutuante de 128 bits). Não usaremos isso no nosso projeto.

Os 5 bits de baixo são todos 1 para instruções maiores que 32 bits, mas nenhuma foi definida. Os grupos OP-IMM32 e OP32 são para permitir que um RV64I ou RV128I possam também fazer operações de 32 bits, mas num RV32I como o drv32h estas instruções não são usadas.

As marcadas como “reserved” serão usadas por extensões oficiais enquanto as “custX” são onde extensões não oficiais devem ser implementadas.

Eliminando a maior parte do circuito do MCPU16h voltamos a ter praticamente só o PC. Para o drv32h aumentamos o PC de 14 para 32 bits (e também o multiplexadore o somador) e trocamos o valor a ser somado para 4. Antes de enviar o valor de PC para ser um endereço de memória descartamos os dois bits de baixo que serão usados para escolher um byte dentro de uma palavra de 32 bits. A largura da memória também é 32 bits mas só aumentamos o endereço para 21 bits (o que dá 8MB no total). Assim, quando o PC endereçar o byte

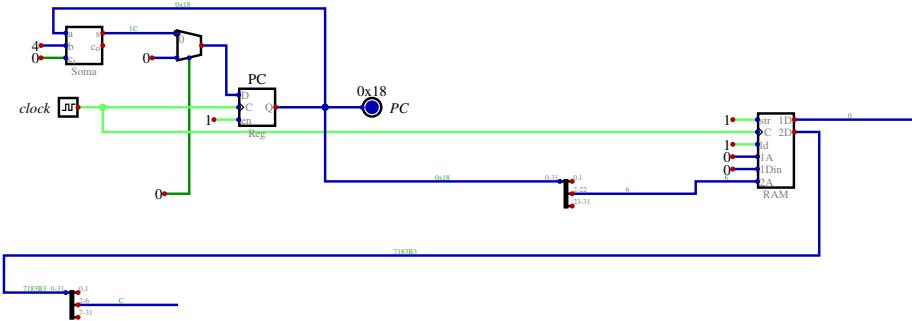


Figure 40: PC do drv32h

0x18 a memória vai ler a palavra 6. Como os 9 bits de cima do endereço não estão ligados, a memória vai se repetir 512 vezes pelo espaço de 4GB.

Inicializamos a memória com *hex/3.04.sine.hex* que é a versão para RISC-V do mesmo programa que o MCPU16h rodou como *hex//3.02.sine.hex*. Enquanto este último arquivo tem 6874 bytes, a versão RISC-V tem apenas 373 bytes.

Separamos a instrução lida nos dois bits de baixo (que devem sempre ser 1 e 1), 5 bits de código principal de operação correspondentes à tabela logo acima e os demais bits.

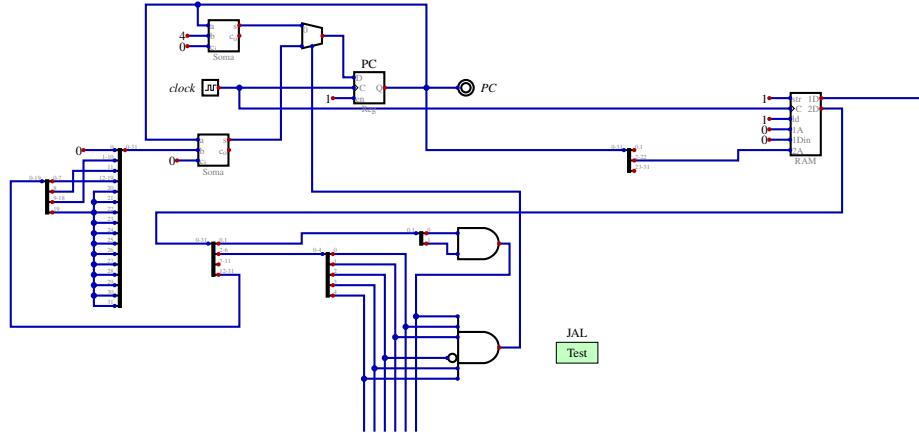


Figure 41: instrução JAL

A primeira instrução a ser implementada é a JAL (“Jump And Link”) que serve para desvios incondicionais e chamadas de subrotina. Ela tem um campo com o registrador de destino mas como estes ainda não existem o campo não está ligado a nada por enquanto. Um valor imediato de 20 bits a ser somado ao PC é codificado de maneira meio complicada. Um segundo somador está sendo usado para addicionar o valor imediato ao PC e enviar o resultado ao multiplexador

que já tínhamos deixado dos projetos anteriores. Uma idéia que parece viável seria ter o multiplexador na entrada do somador original selecionando entre a constante 4 e o valor imediato. Mas isso só daria certo pelo JAL ainda estar incompleto. O valor a ser armazenado no registrador de destino é justamente o PC+4 e o único jeito de ter isso e o PC+imediato é termos dois somadores separados.

Uma porta AND de 6 entradas mais outra de 2 entradas detectam que os 7 bits de baixo da instrução correspondem ao JAL. A saída delas é usada para controlar o multiplexador.

O teste corresponde a:

```
loop:    nop
        jal x31, loop
```

O PC alterna entre 0 e 4.

Em seguida veremos dois grupos de instruções bem parecidos: OP e OP-IMM.

grupo	instrução	funct7		funct3	função
OP	ADD	0000000	rs2	000	rd := rs1 + rs2
OP-IMM	ADDI	imm	imm	000	rd := rs1 + imm
OP	SUB	0100000	rs2	000	rd := rs1 - rs2
OP	SLL	0000000	rs2	001	rd := rs1 « rs2
OP-IMM	SLLI	0000000	imm	001	rd := rs1 « imm
OP	SLT	0000000	rs2	010	rd := rs1 < rs2
OP-IMM	SLTI	imm	imm	010	rd := rs1 < imm
OP	SLTU	0000000	rs2	011	rd := rs1 < rs2 (sem sinal)
OP-IMM	SLTIU	imm	imm	011	rd := rs1 < imm (sem sinal)
OP	XOR	0000000	rs2	100	rd := rs1 XOR rs2
OP-IMM	XORI	imm	imm	100	rd := rs1 XOR imm
OP	SRL	0000000	rs2	101	rd := rs1 » rs2 (sem sinal)

grupo	instrução	funct7		funct3	função
OP-IMM	SRLI	0000000	imm	101	rd := rs1 » imm (sem sinal)
OP	SRA	0100000	rs2	101	rd := rs1 » rs2
OP-IMM	SRAI	0100000	imm	101	rd := rs1 » imm
OP	OR	0000000	rs2	110	rd := rs1 OR rs2
OP-IMM	ORI	imm	imm	110	rd := rs1 OR imm
OP	AND	0000000	rs2	111	rd := rs1 AND rs2
OP-IMM	ANDI	imm	imm	111	rd := rs1 AND imm

A única diferença entre os dois grupos é a falta de um SUBI, mas como o valor imediato é com sinal esta instrução é desnecessária.

Duas novas portas AND de 6 entradas indicam os grupos OP e OP-IMM. Trocamos a saída *PC* por uma ponta de prova *PC*. Ocupa menos espaço e os testes podem usar do mesmo jeito. Também colocamos sondas *JAL*, *OP* e *OP-IMM* para poder testar estes sinais.

Para funcionarem corretamente uma das coisas que estas instruções precisam é de um banco de registradores. O RV32I tem 32 registradores de 32 bits cada, mas o registrador *x0* (também conhecido como “zero”) ignora todas as escritas e retorna sempre 0 quando lido. Depois de contruído o banco repetindo 4 vezes blocos de 8, o registrador *x0* foi apagado e onde estava sua saída foi inserida a constante 0.

Com os registradores ligados a saídas e entradas dá para testá-los manualmente independentemente do resto do circuito.

Escrever programas diretamente em hexadecimal é muito tedioso. Para desenvolver os testes foi chamado o *as* para RV32I na linha de comando de modo que ele leia da entrada padrão e gere a saída quando é digitado control-D. Dai é só copiar o código como comentários no teste. Mesmo assim é fácil cometer erros ao converter da notação “little endian” da listagem para números de 32 bits.

Os 32 bits da instrução são separados em campos conforme o padrão RISC-V. 3 campos de 5 bits são os endereços para o banco de registradores. Um campo de 3 bits pode ser usado diretamente como controle de um multiplexador para escolher qual circuito vai produzir o resultado. A primeira entrada vem de um somador mas a segunda entrada do somador pode ser invertida para uma subtração. Os 7 bits mais significativos da instrução tem apenas dois padrões

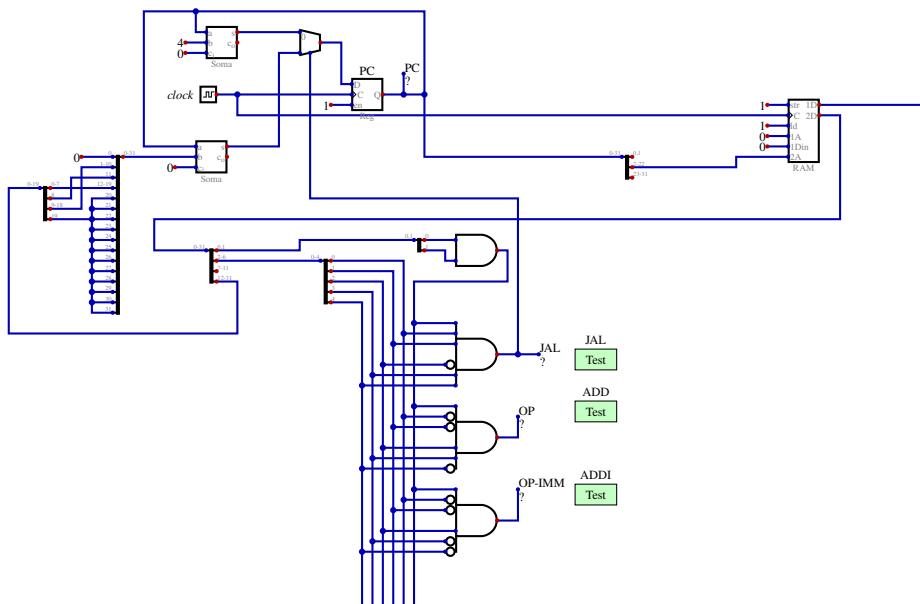


Figure 42: sondas

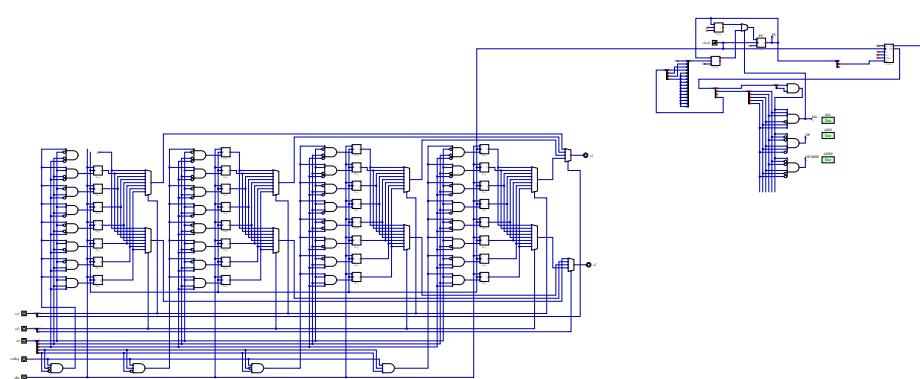


Figure 43: registradores

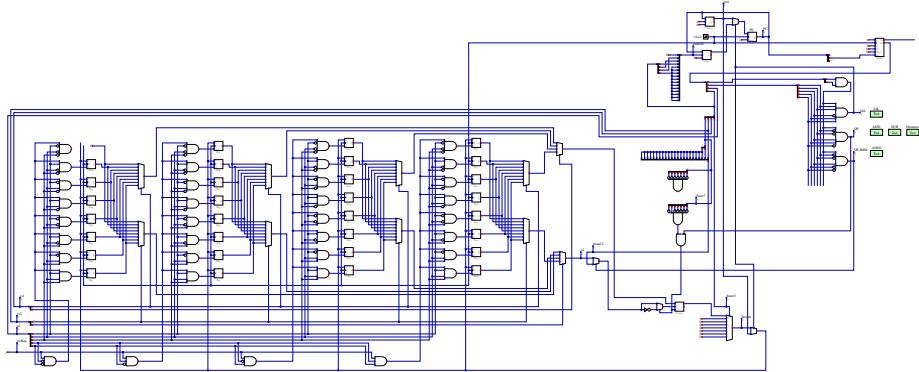


Figure 44: soma e subtração

válidos para as instruções que estamos implementando: 0000000 ou 0100000. O segundo caso (combinado com a indicação de que é OP e não OP-IMM já que estes bits podem ocorrer num valor imediato) transforma a soma em subtração.

Um multiplexador seleciona o valor a ser escrito num registrador entre o resultado calculado e o somador PC+4 (para a instrução JAL).

Com estas instruções já é possível ter um programa de testes que calcula os números de Fibonacci.

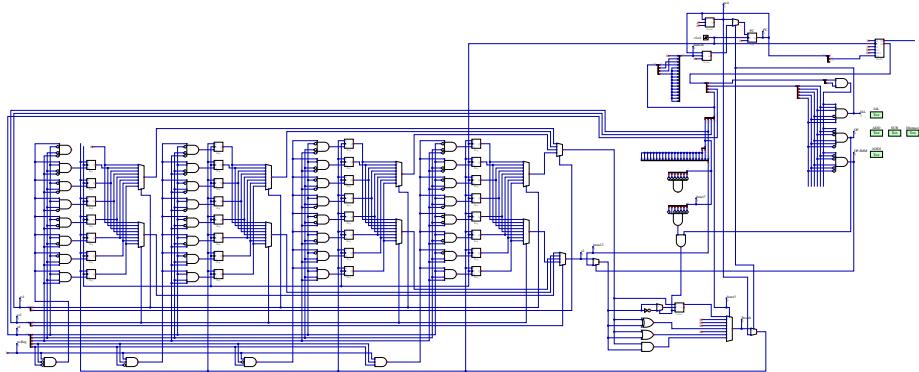


Figure 45: operações lógicas

Com toda esta infraestrutura implementada, adicionar as seis operações lógicas (3 OP e 3 OP-IMM) é muito facil: são as 3 portas de largura 32 bits na parte inferior direita.

Já a comparação de dois números de 32 bits existem dois resultados diferentes possíveis: se os dois números forem considerados sem sinal ou se os dois números forem considerados como sendo a representação em complemento de dois. Não

faz sentido comparar números de representações diferentes.

Estes são os resultados de subtrair dois números de 3 bits cada um ao nível binário (com o vai um do bit 2 mostrado à esquerda antes da vírgula e o vai 1 do bit 1 mostrado à direita entre parênteses):

	000	001	010	011	100	101	110	111
000	1,000 (1)	1,001 (1)	1,010 (1)	1,011 (1)	1,100 (1)	1,101 (1)	1,110 (1)	1,111 (1)
001	0,111 (0)	1,000 (1)	1,001 (1)	1,010 (1)	1,011 (0)	1,100 (1)	1,101 (1)	1,110 (1)
010	0,110 (0)	0,111 (0)	1,000 (1)	1,001 (1)	1,010 (0)	1,011 (0)	1,100 (1)	1,101 (1)
011	0,101 (0)	0,110 (0)	0,111 (0)	1,000 (1)	1,001 (0)	1,010 (0)	1,011 (0)	1,100 (1)
100	0,100 (1)	0,101 (1)	0,110 (1)	0,111 (1)	1,000 (1)	1,001 (1)	1,010 (1)	1,011 (1)
101	0,011 (0)	0,100 (1)	0,101 (1)	0,110 (1)	0,111 (0)	1,000 (1)	1,001 (1)	1,010 (1)
110	0,010 (0)	0,011 (0)	0,100 (1)	0,101 (1)	0,110 (0)	0,111 (0)	1,000 (1)	1,001 (1)
111	0,001 (1)	0,010 (0)	0,011 (0)	0,100 (1)	0,101 (0)	0,110 (0)	0,111 (0)	1,000 (1)

No caso da comparação sem sinal, os resultados equivalentes são:

	0	1	2	3	4	5	6	7
0	=	>	>	>	>	>	>	>
1	<	=	>	>	>	>	>	>
2	<	<	=	>	>	>	>	>
3	<	<	<	=	>	>	>	>
4	<	<	<	<	=	>	>	>
5	<	<	<	<	<	=	>	>
6	<	<	<	<	<	<	=	>
7	<	<	<	<	<	<	<	=

Comparando as duas tabelas vemos que “menor sem sinal” (LTU das instruções SLTU e BLTU) é só o NOT do vai um do bit 2.

Para comparações com sinal os resultados são:

	+0	+1	+2	+3	-4	-3	-2	-1
+0	=	>	>	>	<	<	<	<
+1	<	=	>	>	<	<	<	<
+2	<	<	=	>	<	<	<	<
+3	<	<	<	=	<	<	<	<
-4	>	>	>	>	=	>	>	>
-3	>	>	>	>	<	=	>	>
-2	>	>	>	>	<	<	=	>
-1	>	>	>	>	<	<	<	=

É muito mais complicado ver em que casos a resposta é menor com sinal, mas se observarmos o vai um do bit 2, o vai um do bit 1 e o bit 2 da soma veremos que a resposta é menor sempre que o número de 1s destes 3 bits é ímpar. O XOR dos dois primeiros bits está disponível em vários processadores como indicador *V* (“overflow”) que significa que os bits produzidos no resultado não são os corretos. Com 3 bits não é possível representar todas as respostas. Somando 5 e 5 sem sinal, a resposta será dez e isso precisaria de 4 bits. Somando 2 e 3 com sinal a resposta seria cinco, mas isso representa -3 em complemento de dois. Um valor positivo de *V* indica estas situações para que o software descarte o resultado. Já um XOR de *V* com o sinal da resposta indica a operação “menor com sinal”.

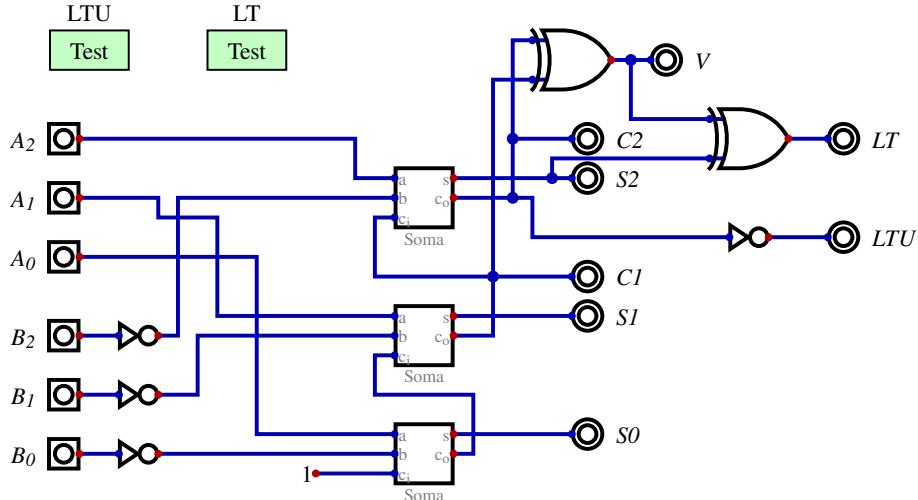


Figure 46: teste de comparações com e sem sinal

Normalmente faríamos um esforço para reaproveitar o circuito de SUB para as instruções SLT, SLTU, SLTI e SLTIU (além dos desvios condicionais, que ainda não vimos). Mas aqui usamos um somador separado e ai tem um para os 31 bits de baixo e outro só para o bit mais significativo, o que gera todos os sinais que precisamos para as comparações.

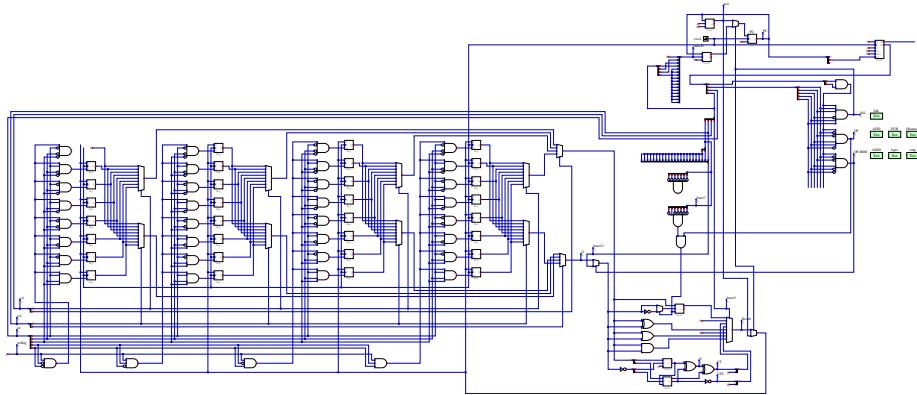


Figure 47: instruções de comparação

As instruções de deslocamento exigem muito hardware adicional. Para deslocar à esquerda por apenas um dígito basta somar um valor a si mesmo. Mas para deslocar para a direita podemos usar um multiplexador para selectionar uma versão deslocada do valor (o que pode ser feito com apenas fios). O dígito mais da direita vai desaparecer e precisamos escolher que bit colocar no dígito mais da esquerda. Se considerarmos números sem sinal o novo bit deverá ser 0 enquanto para números com sinal o certo é copiar o valor anterior deste mesmo bit.

Muitos processadores ou tinham instruções que deslocavam por um único dígito ou então permitiam qualquer número de dígitos mas demoravam um ciclo de relógio por dígito. A primeira opção é incompatível com o padrão RISC-V enquanto a segunda não se encaixa no estilo do resto do drv32h que executa todas as instruções num único ciclo.

Uma maneira de se deslocar por qualquer número de dígitos em um único ciclo de relógio é trocar o deslocador único pelo qual o dado passa N vezes por N deslocadores em série. Só que para 8 bits seriam necessários 8 deslocadores (multiplexadores) e para 32 bits seriam 32 deslocadores, o que é um circuito muito grande e lento.

Uma solução mais elegante é fazer alguns destes deslocadores moverem os bits mais de um dígito de cada vez. O primeiro deslocador poderia mover por 1 dígito, o segundo por 2 dígitos, o terceiro por 4 dígitos e depois por 8, 16 e assim por diante. Ai acontece uma coisa interessante - cada bit do segundo operando pode controlar diretamente um dos deslocadores. Se o operando for 5 (00101 em binário) nós deslocamos por 1, mas não por 2, sim por 4, mas não por 8 e nem por 16. Fica muito simples o circuito. No circuito de teste vemos 3 deslocadores para 8 bits.

Na metade de cima do circuito de teste temos o deslocamento para esquerda. Em cada estágio o multiplexador passa o dado inalterado se o bit correspondente do segundo operando for 0 ou seleciona um deslocamento criado apenas com fios.

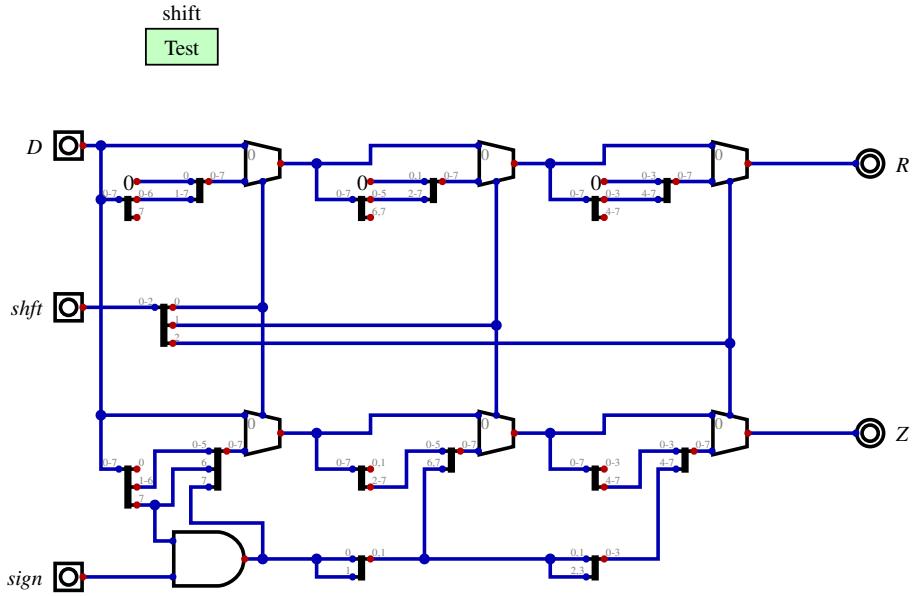


Figure 48: teste de deslocamentos

Em termos de desenho os 3 estágios parecem iguais, mas os fios e as contantes 0 tem larguras diferentes.

Na metade de baixo o deslocamento para a direita é um pouco mais complicado, pois ao invés do valor a ser inserido ser sempre 0 (mas com o dobro de bits a cada estágio) ele é o bit 7 do dado original ou (via porta AND) 0 dependendo de se queremos uma instrução SRL ou SRA. Expandindo a idéia para 5 estágios de 32 bits teremos as últimas instruções OP e OP-IMM.

A próxima instrução será o JALR que fica meio termo entre o JAL e as OP-IMM. Na verdade iremos querer ser como o OP-IMM para o JALR, LOAD e STORE e por isso vamos modificar o sinal de controle do multiplexador que seleciona entre $R[rs2]$ e $imm12$. Da mesma forma, modificaremos o sinal para o multiplexador que seleciona entre o resultado e $PC+4$ para serem escritos no registrador. Um novo multiplexador seleciona entre o cálculo que o JAL já fazia e o resultado da ALU (que vai ser uma soma já que a instrução JALR tem um funct3 igual a 0).

No teste da instrução JALR já podemos ter uma pequena subrotina que é chamava usando a instrução JAL e depois retorna usando JALR. O outro uso de JALR é para seguir ponteiros para funções e implementar desvios calculados como em estruturas switch/case de linguagens de programação.

Ainda no desvios, podemos implementar os desvios condicionais para encerrar esta parte. Estes compararam dois registradores e se a comparação for a desejada ele soma o valor imediato (também de 12 bits, mas tirados de bits diferentes

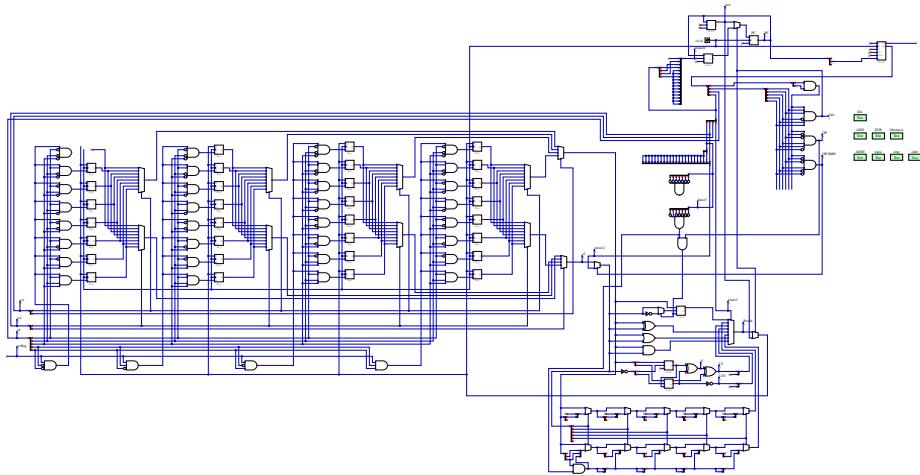


Figure 49: deslocamentos

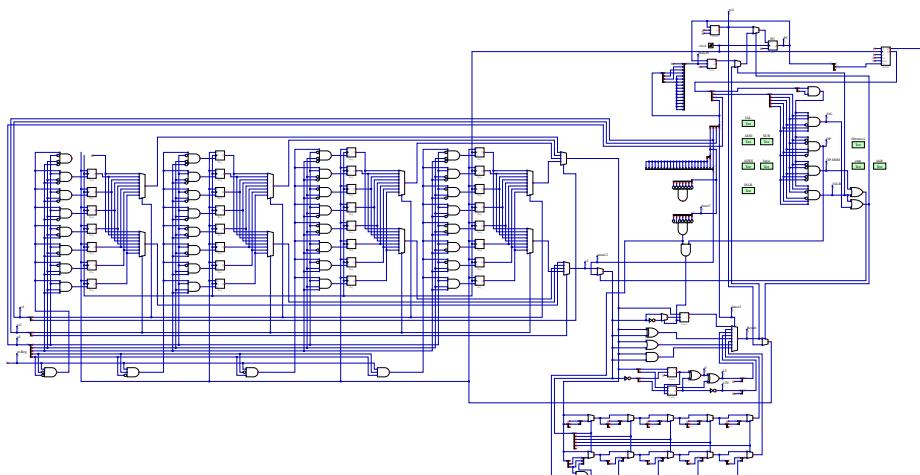


Figure 50: instrução JALR

de OP-IMM e JALR) ao PC mas se a comparação não der certo a instrução não faz nada. Já implementamos 4 das 6 comparações necessárias como parte de SLT e SLTU. Falta poder comparar se dois números são iguais. Geralmente isso é feito vendendo-se a subtração (que fizemos para as outras duas comparações) teve resultado zero. Mas também fizemos o XOR dos dois operandos e se algum bit (testando com um OR de 32 entradas) disso for 1 então os números não são iguais. A mesma funct3 que escolhe os resultados da ALU controla um novo multiplexador de 8 entradas que escolhe a comparação a ser feita. Duas das comparações não estão definidas (por isso as forçamos a sempre serem falso) e as outras são diferentes, LT e LTU bem como seus inversos (igual, GE e GEU).

Os desvios condicionais são as primeiras instruções que estamos implementando que não gravam um resultado no banco de registradores (a STORE é a outra que veremos). Usamos o sinal *branch* antes deste ser combinado com a comparação já que deixamos de gravar no registrador independentemente de desviarmos ou não.

Podemos aproveitar o somador do JAL com um multiplexador para escolher entre o imm20 que ele soma ao PC e o imediato especial dos desvios. O novo imediato é escolhido quando temos um desvio e a condição é verdadeira. O OR de JAL e JALR é expandido para também receber este sinal e selecionar PC+imm no lugar de PC+4.

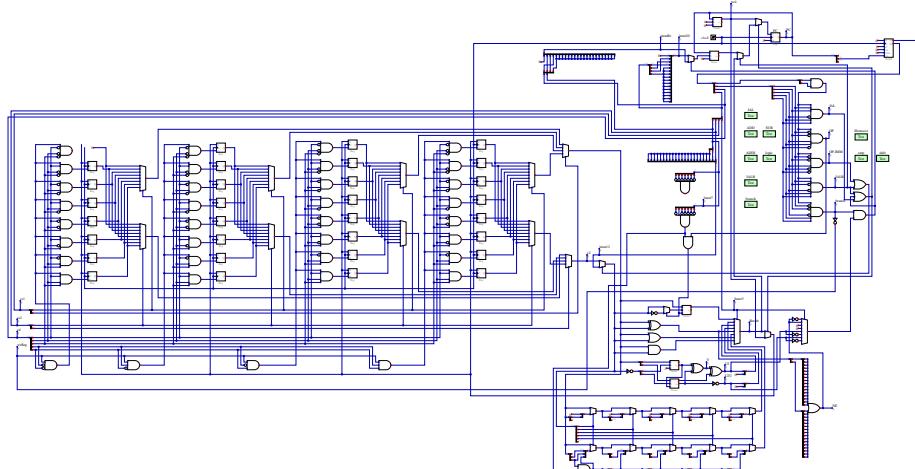


Figure 51: instruções de desvio

Na criação de um teste para os desvios foi percebido um erro que já estava desde o início: o sinal *str* da memória estava acionado sempre, de modo que a cada instrução a segunda porta estava escrevendo 0 no endereço 0. Como o teste de desvios foi o primeiro a tentar executar a instrução em 0 uma segunda vez foi a primeira vez que o problema foi visto (e agora corrigido). Isso é, infelizmente, típico. Os testes são escritos para tentar observar problemas que imaginamos,

mas os problemas que não imaginamos e não testamos diretamente existem.

As duas instruções seguintes (LUI e AUIPC) não são necessárias, mas são convenientes. A maioria dos outros processadores não tem algo parecido. O uso de valores imediatos de apenas 12 ou 20 bits parece um grande limite, mas com as instruções já implementadas podem ser usadas para carregar um valor qualquer de 32 bits (0x12345678, por exemplo) em um registrador:

```
addi x6,zero,(0x12345678>>22)&0x7FF
slli x6,x6,11
addi x6,x6,(0x12345678>>11)&0x7FF
slli x6,x6,11
addi x6,x6,0x123445678&0x7FF
```

Implementar LUI não cria funcionalidade nova, mas reduz o código acima para apenas duas instruções:

```
lui x6,0x12345678>>12
addi x6,x6,0x12345678&0x0FFF
```

O montador oferece a pseudo-instrução “li x6,0x12345678” que vira o código acima. Também omite o LUI se a constante couber em 12 bits. E também leva em conta que o valor do imediato do ADDI é com sinal e ajusta o imediato do LUI se for necessário para o resultado final ficar certo.

Como podemos carregar qualquer valor de 32 bits num registrador, a instrução JALR pode pular para qualquer lugar na memória e o limite do JAL não é realmente um problema. Mas o JAL é um desvio relativo, o que permite um trecho de código ser copiado para outro endereço e continuar funcionando (necessário se formos juntar vários módulos num único executável). Podemos calcular um destino relativo ao custo de um registrador à mais:

```
li x6,0x01000000
jal x7,next
next: addi x6,x6,x7
jalr zero,-4(x6)
```

As 3 primeiras instruções podem ser trocadas por um AUIPC e ai o JALR não precisa compensar o PC+4 do JAL e sim fornecer os 12 bits de baixo do desvio (e o valor do imediato do AUIPC pode ter que ser ajustado se o JALR tiver um imediato negativo).

Para verificar que estmos indo na direção certa:

instrução	PC :=	reg[rd] :=
OP	PC+4	Result
OP-IMM	PC+4	Result imm12
JAL	PC+imm20	PC+4
JALR	Result imm12	PC+4

instrução	PC :=	reg[rd] :=
desvio cond sim	PC+immBr	x
desvio cond não	PC+4	x
AUIPC	PC+4	PC+immUI
LUI	PC+4	immUI
LOAD	PC+4	dMem (rs1+imm12)
STORE	PC+4	x (rs1+immS)

Baseado nisso introduzimos mais um somador (para PC+immUI) e dois multiplexadores: um para selecionar entre PC+4 e o segundo multiplexador que seleciona entre immUI e PC+immUI. Mais duas porta OR aumentam os casos em que dois sinais de controle são acionados para incluir as novas instruções.

A tabela tem os grupos de instruções LOAD e STORE. Sem elas tudo funciona mas podemos usar apenas 31 dados de 32 bits cada em nosso programa. O termo “RISC” é um tanto ambíguo: é “computador com conjunto reduzido de instruções” ou “computador com conjunto de instruções reduzidas”? Uma alternativa é falar de “arquitetura load/store” para indicar que apenas instruções específicas fazem acesso à memória. O MCPU16h tem tanto um conjunto reduzido de instruções (apenas 4) e as instruções são bem simples. Mas o NOR e o ADD não só fazem o que o nome diz mas também fazem acesso à memória, então não é uma arquitetura load/store. O drv32h que estamos terminando aqui com certeza é.

O sinal STORE pode diretamente controlar o sinal *str* da memória enquanto o LOAD pode ser ligado ao *ld* da memória. A entrada de dados da memória pode vir da saída *s2* dos registradores e o endereço pode vir do somador da ALU pois vamos usar o funct3 para definir a variante do LOAD e STORE de modo que a saída da ALU pode não ser o que queremos.

O NOT que impedia a escrita no banco dos registradores para a instruções de desvio condicional foi trocado por um NOR para que o STORE também não escreva nos registradores, como indicado na tabela acima.

O LOAD, por sua vez, escreve no banco de registradores mas não o resultado da ALU nem PC+4. Precisamos de um novo multiplexador para receber o dado vindo da memória. Mas a tabela mostra que enquanto o load usa o mesmo imm12 do OP-IMM e JALR, o STORE tem um imediato quase idêntico ao dos desvios condicionais. Resolvemos isso com mais um multiplexador.

Agora podemos escrever e ler da memória, mas apenas palavras inteiras. Mas talvez seja mais conveniente trabalhar com informações de 16 bits ou até acessar o bytes (8 bits) individuais. O campo funct3 indica o tamanho a ser transferido. Na leitura da memória é só ter um novo circuito que usa os dois bits de baixo do endereço e a informação da largura para extrair os bits relevantes transformando num número de 32 bits a ser enviado para o registrador.

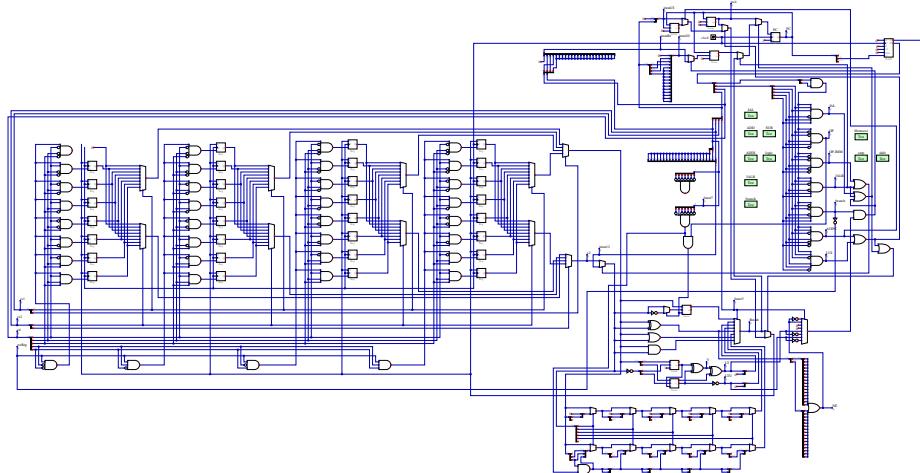


Figure 52: instruções LUI e AUIPC

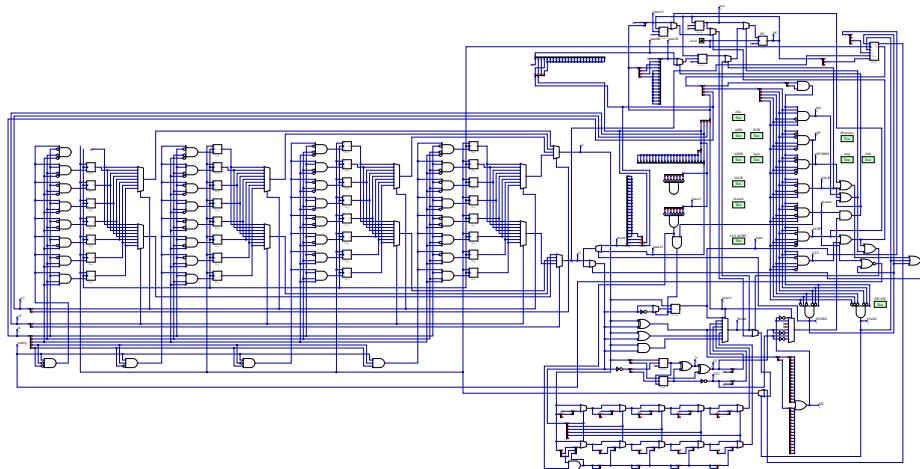


Figure 53: instruções SW e LW

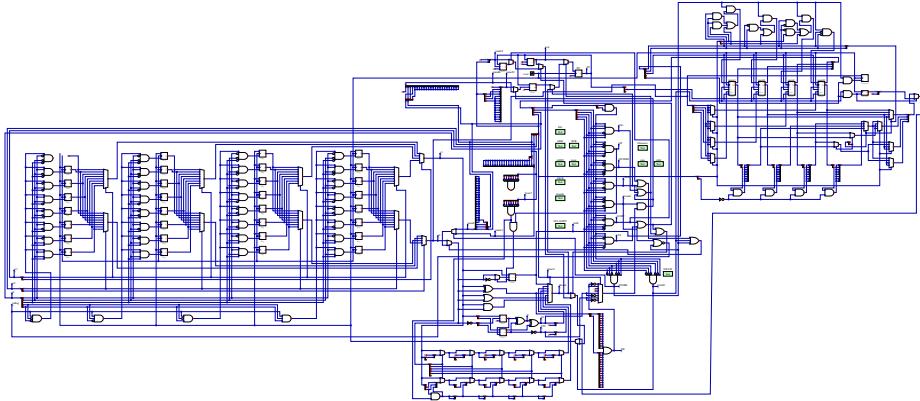


Figure 54: drv32h completo

A escrita de parte de uma palavra é mais complicada. Uma opção é ler a palavra inteira, modificar os bits desejados e escrever a palavra inteira de volta. Ao invés disso usaremos 4 circuitos de memória individuais de 8 bits cada uma. Agora temos 4 sinais *str* separadas que podemos controlar individualmente.

funct3_1	funct3_0	A_1	A_0	str_3	str_2	str_1	str_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	1	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	1	1
0	1	1	0	1	1	0	0
0	1	X	1	X	X	X	X
1	0	0	0	1	1	1	1
1	0	X	1	X	X	X	X
1	0	1	X	X	X	X	X
1	1	X	X	X	X	X	X

Estes sinais são todos 0 se não for uma instrução STORE. Além dos sinais *str* é necessário reorganizar os dados. Se estivermos gravando um byte, por exemplo, então os bits 7 a 0 precisar ir para todas as 4 memórias. Isso é feito com 4 multiplexadores de 4 entradas de 32 bits cada uma e controlados pelos dois bits de baixo de funct3.

Tudo isso é supondo que apenas acessos alinhados são permitidos. Bytes podem ser lidos ou escritos em qualquer endereço enquanto palavras de 16 bits apenas em endereços pares e palavras de 32 bits apenas em endereços múltiplos de 4.

Para a leitura também usamos multiplexadores:

funct3_1	funct3_0	A_1	A_0	byte3	byte2	byte1	byte0
0	0	0	0	sign0	sign0	sign0	mem0
0	0	0	1	sign1	sign1	sign1	mem1
0	0	1	0	sign2	sign2	sign2	mem2
0	0	1	1	sign3	sign3	sign3	mem3
0	1	0	0	sign1	sign1	mem1	mem0
0	1	1	0	sign3	sign3	mem3	mem2
0	1	X	1	X	X	X	X
1	0	0	0	mem3	mem2	mem1	mem0
1	0	X	1	X	X	X	X
1	0	1	X	X	X	X	X
1	1	X	X	X	X	X	X

O bit mais significativo do endereço é usado para selecionar os periféricos (teclado e terminal). Reservar 2GB completos para isso é um desperdício mas o foco neste projetos é simplicidade e não eficiência.

4. FPGAs e Shin JAMMA

5. Vídeo e Áudio

6. Pegasus 42

A. História

De 1977 a 1991 o Brasil teve uma “reserva de mercado” para computadores de pequeno porte. Na época praticamente toda importação era proibida, de automóveis a chocolate incluindo computadores. Mas para os outros produtos uma empresa estrangeira como a Volkswagen ou a Nestlé poderia abrir fábricas no país para vender localmente. A reserva impedia isso para o caso de mini e microcomputadores.

Numa época de alta inflação e sem crescimento econômico, o setor de informática era um dos poucos em franca expansão. Isso atraiu investimento de áreas não relacionadas, especialmente pela oportunidade de poder copiar produtos estrangeiros sem ter que enfrentar a concorrência dos originais. Neste contexto um grupo do setor bancário procurou o advogado Amaro Moraes e Silva Neto em busca de idéias e o Amaro disse que uma coisa que ninguém estava fazendo era um computador para crianças. Ele disse que conhecia um gênio da computação que poderia desenvolver tal produto.

Ele estava pensando em Fábio Cavalcante da Cunha, que era oficialmente estudante de engenharia eletrônica da Escola Politécnica da Universidade de São



Figure 55: Amaro Moraes e Silva Neto

Paulo (Poli-USP) mas estava na época mais focado no seu emprego numa loja de computadores.

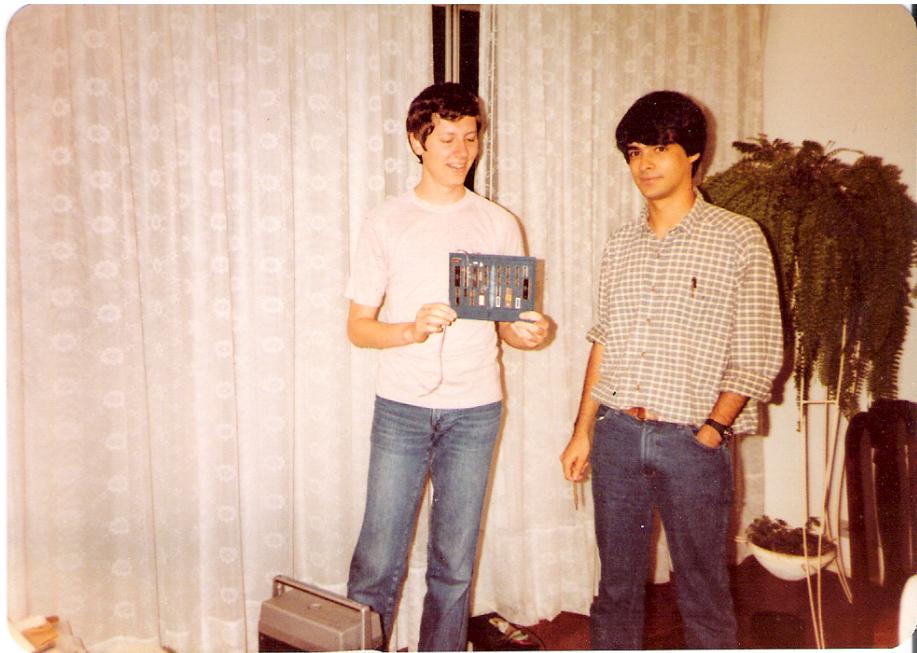


Figure 56: Jecel e Fábio e frente do protótipo

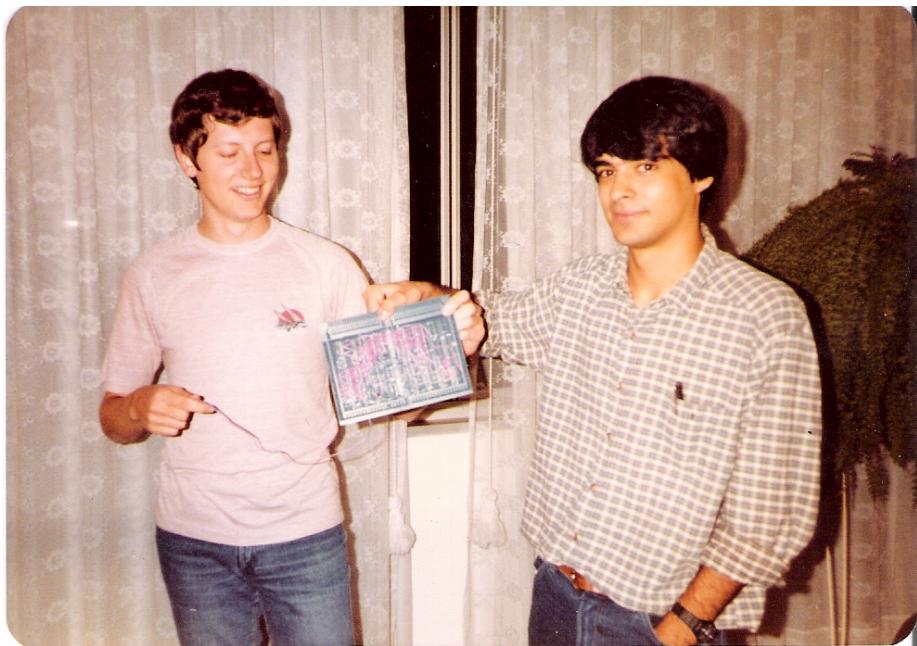


Figure 57: Jecel e Fábio e fiação do protótipo

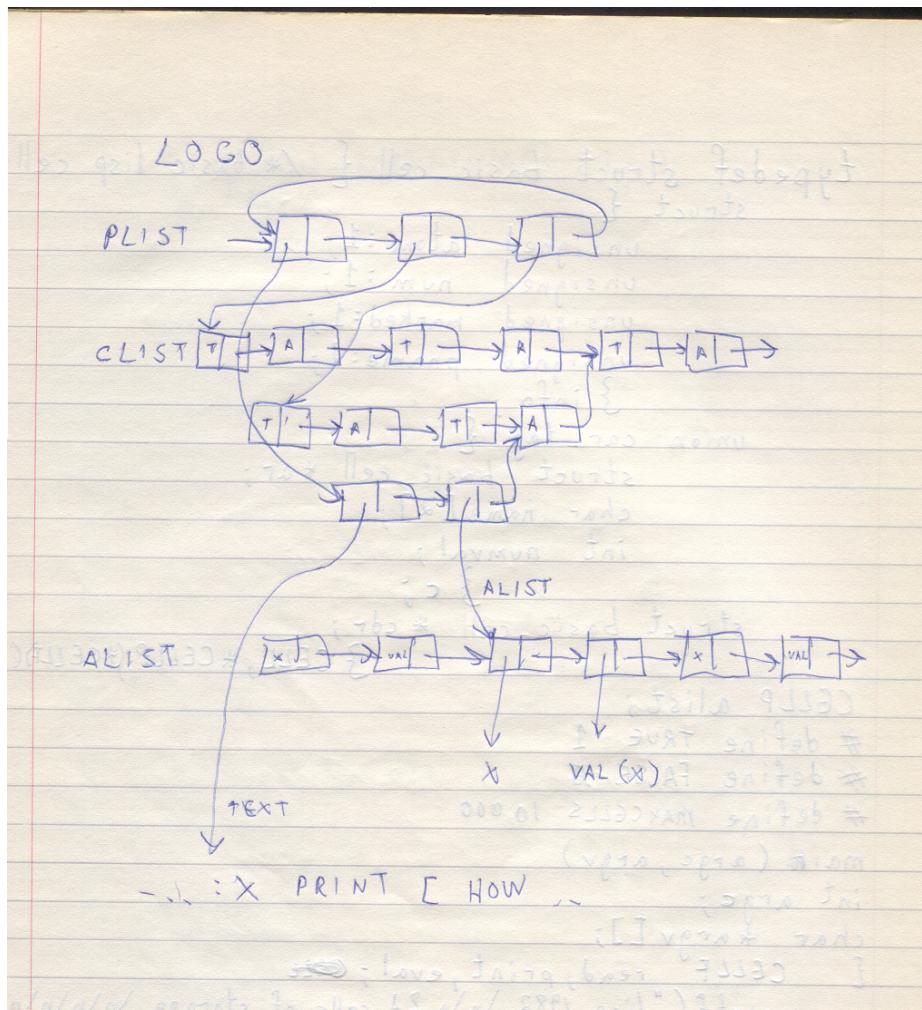


Figure 58: estrutura do Logo

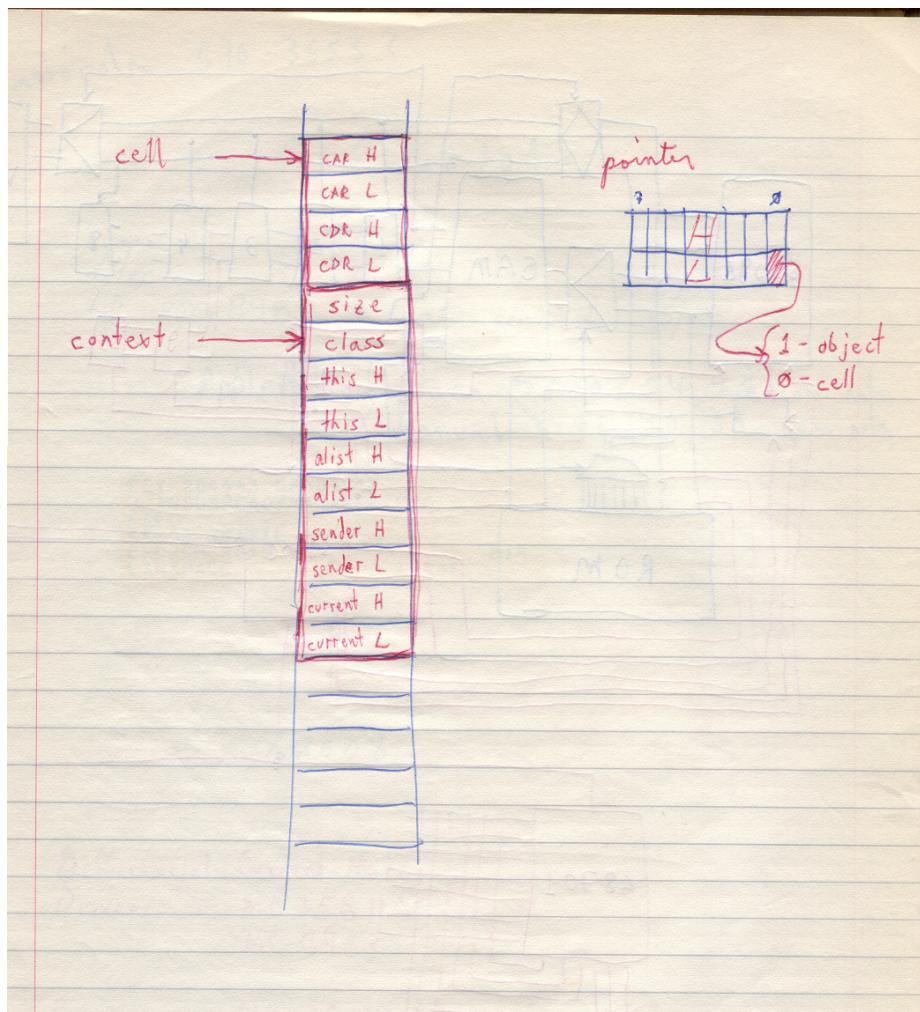


Figure 59: formato dos objetos

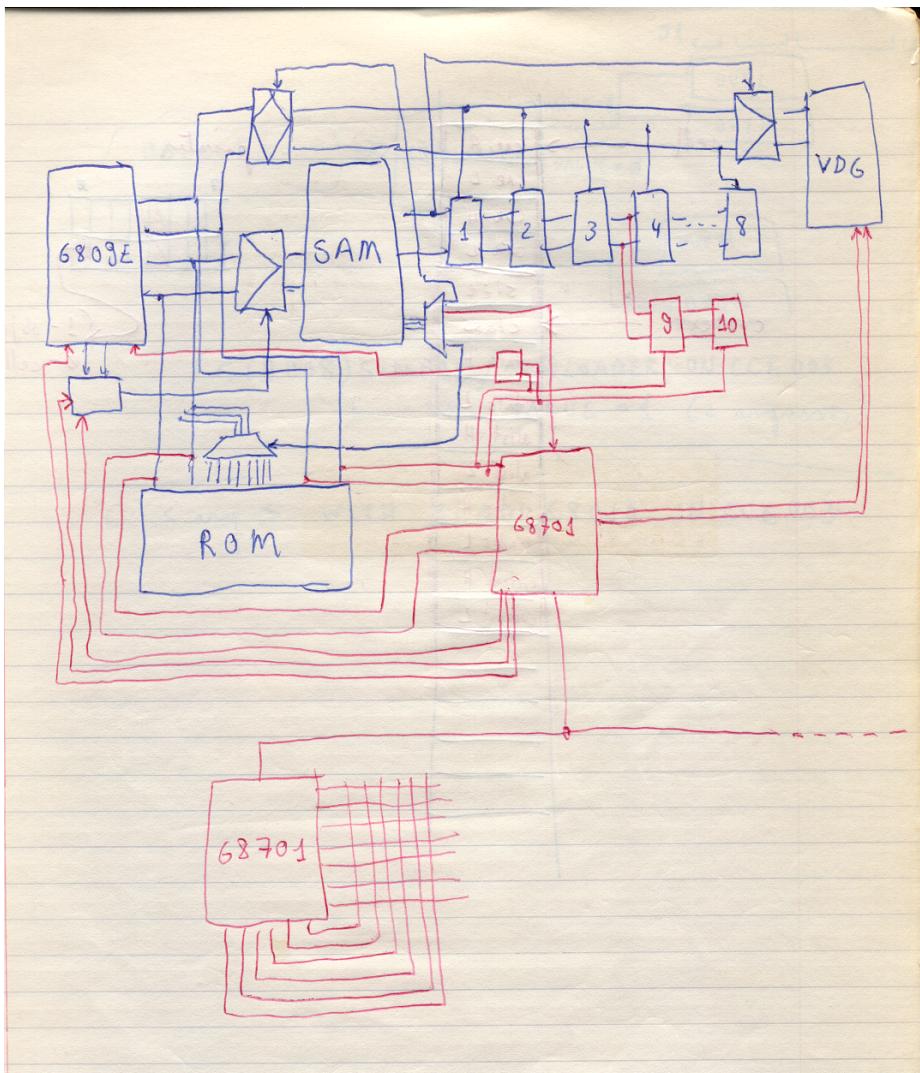
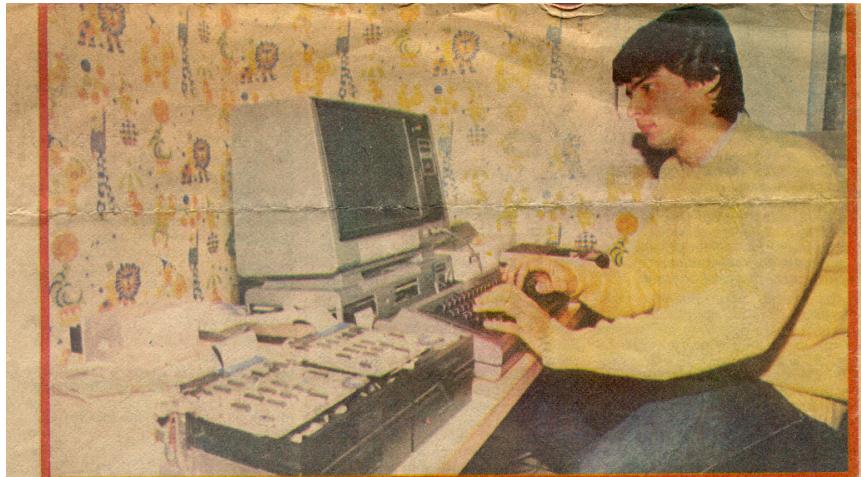


Figure 60: esquemático do Pegasus



Com linguagem Logo, a máquina de Fábio é inédita

19 anos, aficionado, Fábio inventa a nova eletrônica

Com seu computador, ele se torna um pioneiro

JOSÉ EDUARDO MENDONÇA

Influenciado pelo pai, um rádiodoador, Fábio Cavalcanti resolveu, aos sete anos de idade, desmontar um rádio para ver como as válvulas funcionavam. Ganhou uma bronca, e o pai fez um rádio que nunca mais funcionaria. Além disso descobriu, com a experiência, a paixão pela eletrônica que o levou agora, aos 19 anos, a fabricar seu próprio computador.

Aos 13 anos, Fábio fez um curso de computação e o "arquivou na cabeça", devido a interesses que julgava mais apropriados no momento, como seu autorama e os aeromodelos. Mais tarde, no colégio, começou de novo a "ler revistas especializadas e montar pequenos circuitos". Tentou fazer um curso mais avançado e os que encontrou "limitavam demais o conhecimento". Mais uma vez, esqueceu temporariamente sua paixão para enfrentar o cursinho e entrar na Politécnica no curso de engenharia eletrônica, depois de já ter conseguido uma vaga na universidade de São Carlos, para onde migrou abandonando sua cidade natal, Londrina.

Seu computador, o Projeto Pegasus, para a linguagem Logo (ver página "Computadores"), começou a nascer com o exercício de programação em calculadoras e a busca constante de maiores informações. Na época, "78/79, ainda não existia este clima de hoje, com os computadores, só mexia com eles o pessoal de processamento de dados em grandes computadores". Fábio aprendeu "tudo que podia saber uma calculadora, depois de fugir durante um ano", e optou pelo caminho natural: em 1983 foi para os Estados Unidos, mais especificamente para o coração da indústria de computação, o laboratório do Massachusetts Institute of Technology, na Universidade de Stanford, em pleno Vale do Silício.

Um de seus encontros lá foi tão emocionante quanto o encontro, digamos, de um guitarrista daqui com Frank Zappa. Fábio trocou horas de papo com seu guru Marvin Minski, um dos papas da inteligência artificial, e conseguiu provar para si mesmo "que podia estar lá". A mágica foi não ter conseguido uma bolsa de graduação e ter "assumido o caminho de volta".

"Se eu tinha de fazer alguma coisa aqui, senti que teria de fazer como sempre fiz: sozinho." Armado de dezenas de livros, estimulado por dois ou três colegas de escola também aficionados pelos computadores, utilizando seu TRS-80 da Radio Shack, Fábio partiu para seu projeto. Antes, outra deceção: ofereceu seus serviços à Poli, disse que tinha estado no MIT e que gostaria de colaborar. Ouviu como resposta que suas idéias não tinham aplicação comercial imediata. "Parece que dentro da escola não acreditam muito no valor dos alunos, e assim nós temos de criar fora dela."

O computador de Fábio será, diz ele, "possivelmente o primeiro do mundo a vir já com a linguagem Logo, e não com a Basic." A idéia: "Aproximar as crianças dos computadores. Tem cor, som, pode até mesmo programar jogos e serve para crianças a partir de 4 anos até a adolescência, ou mais. Há também excelentes recursos para aplicações mais maduras." O projeto Pegasus ainda não foi enviado para a necessária avaliação e aprovação da Secretaria Especial de Informática, mas, segundo um de seus autores (o outro é o colega Jacel Mattos de Assumpção, também um autodidata, e "um dos poucos gênios" que Fábio conhece), "a máquina já está muito bem definida".

O projeto Pegasus deverá ser comercializado por cerca de 300 mil cruzeiros, tem 64 k bytes de memória, 8 bits e utiliza um chip 6809 da Motorola. Fábio enfatiza que seu computador

não é de forma nenhuma um "game", passatempo que critica: "O videogame pode estimular, mas para uma posição errada. As pessoas podem aprender a utilizar o computador como brinquedo, e apenas assim. Ai ele não estará servindo como ferramenta de aprendizagem."

Além de estudar na Poli e dar aula de computação na Emarés para crianças, Fábio Cavalcanti "troca figurinhas" com grupos de usuários de computadores Apple e TRS-80. Há, no entanto, poucos jovens. "Isto está apenas começando. Os mais novos aprendem diferente. Os mais velhos vai falar como o computador funciona, e a garotada vai perguntar como é que eu faço."

Como profissionais de outras áreas, Fábio e seus colegas têm sua galeria de ídolos e seus livros e passatempos prediletos. Exemplos de pessoas quentíssimas: Wolzniack, um dos inventores do Apple (mais "fera" que o outro, Steven Jobs, tído pelos mais ruros como comerciante), Scott Adams, genial escritor de softwares para games em computador; Leo Chrissopherson, programador de animações e Nolan Bushnell, inventor da Atari.

Todos leem Sherlock Holmes por seu gênio de dedução, os livros de Tolkien por sua magia, o já clássico "Godel, Escher e Bach", de Douglas Hofstadter, um estudo comparativo entre o matemático, o arquiteto e ilustrador e o músico. Adoram assistir "Star Trek" (Jornada nas Estrelas) e cultuam em forma de livros ou filmes a série "The Twilight Zone" (Além da Imaginação). Jogam horas seguidas o jogo de computador chamado "Adventure", tentando sempre superar suas próprias marcas. Com uma cultura semelhante a esta em todo o mundo, estes jovens aficionados dos micros e da robótica começam a criar a tecnologia do futuro.

Figure 61: reportagem da Folha de São Paulo
68

LINGUAGEM LOGO

Dois estudantes da Escola Politécnica da USP, Fábio da Cunha e Jecel Mattos Jr., desenvolveram e estarão lançando na Feira de Informática o PÉGASSOS, primeiro microcomputador que vem com linguagem LOGO. Segundo Fábio da Cunha, a linguagem LOGO é a mais adequada para se aprender computação por sua facilidade de uso, daí sua grande aplicação junto a crianças. O PÉGASSOS funciona com microprocessador 6809, da Eletrola, com velocidade de 1MHz. A memória do sistema é de 16Kbytes de ROM, podendo chegar até 64K, e a memória do usuário é de 64Kbytes. O novo micro tem teclado alfanumérico, com todos os elementos do português, como ç e todos os acentos. O equipamento já vem com interface embutida para ligação com gravador cassete e com televisor comum, inclusive TV a cores (8 cores no sistema Pali/M). O PEGASSOS possui sistema operacional gráfico, som, e tem expansões para disco, impressora e para ligação a um outro micro. Pode também receber cartuchos com jogos e com outras linguagens.

Figure 62: reportagem na revista Microsistemas
69

Brasil coleciona casos de sucessos e fracassos

A saga de Steven Jobs e Stephen Wozniack não contagiou apenas jovens americanos. A idéia de confrontar uma grande companhia, tomando como ariste apenas o cérebro, um punhado de chips e um micro doméstico, ganhou entusiastas adeptos no Brasil, e aqui, como nos EUA, os herdeiros de Jobs e Wozniack podem ser divididos entre vitoriosos e fracassados: Ivan Nazarenko, 21, por exemplo, que estará concluindo o curso no Instituto Tecnológico da Aeronáutica (ITA) em 1986, pode ser perfeitamente enquadrado no primeiro caso. Em 1983, quando os micros fabricados no país ainda não tinham caracteres específicos da língua portuguesa, como o cedilha e acentos, Nazarenko, com apenas 18 anos, criou uma placa que "ensinava português" aos computadores nacionais, batizada de Ivana.

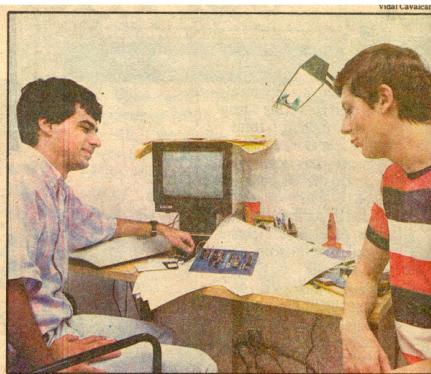
"Foi um invento simples, mas que não tinha precedentes no mundo", explica Nazarenko, que já vendeu através de sua microempresa 1.700 placas, que hoje são comercializadas a 17 ORTNS. "Não deu para ganhar muito dinheiro, mas alimentou certos luxos que tenho. Sem a Ivana, eu não poderia ter um Passat 1.8, que adoro, e seria obrigado a me contentar com o modelo 1.6", diz. No momento, a empresa de Nazarenko assiste uma queda de vendas da Ivana —que teve seu ápice de comercialização no começo deste ano—, mas deve voltar à carga no final de 86. "Estou projetando um equipamento de controle para automação industrial, como trabalho final de curso, que deverá ser o

próximo produto da empresa".

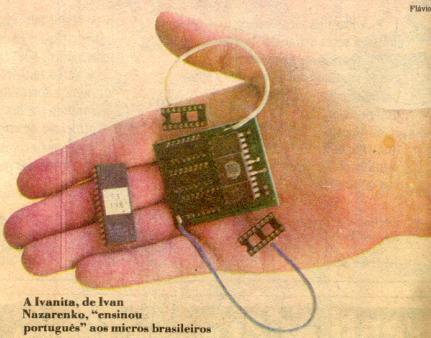
Christovão Manuel Baptista, 32, diretor da Blump, também começa a vislumbrar o retorno de vinte anos investidos em pesquisa. Criando seu primeiro robô aos doze anos —"com lata de talco no corpo e cabeça de tampa de aerosol"— Baptista começa a engatinhar num mercado milionário: o de braços mecânicos e robôs industriais. Dos Cr\$ 50 mil iniciais, abocanhados no programa do Silvio Santos com sua primeira engenhoca, Baptista pode saltar para Cr\$ 100 milhões —o preço estimado de seu modelo mais simples de robô industrial.

Já Fábio Cavalcanti, 24, e Jecel Mattos de Assumpção Jr., 24, criaram há dois anos atrás um computador baseado na filosofia educacional Logo, que prevê o uso de micros em salas de aula, batizado de Logo Machine. Na época, quando os micros domésticos ainda não tinham cor, a máquina de Fábio e Jecel, após seis meses de pesquisa, operava com oito cores e sintetizador de sons. Resultado: ninguém quis fabricar a Logo Machine. "Nos antecipamos as necessidades do mercado", reconhece Fábio.

Atualmente, Fábio e Jecel têm um projeto bem mais ambicioso: estão projetando um micro de 32 bits voltado à linguagem Small Talk —uma poderosa filosofia de programação criada pela Xerox dos EUA, que não requer qualquer conhecimento de computação. "Agora estamos no caminho certo", diz Jecel. "Conceitos de Small Talk são usados no Macintosh da Apple, no sistema operacional Unix, mas queremos criar uma máquina puramente Small Talk".



Fábio (esq.), Jecel e a placa de um microcomputador frustrado



A Ivana, de Ivan Nazarenko, "ensinou português" aos micros brasileiros

Figure 63: reportagem da Folha de São Paulo dois anos depois