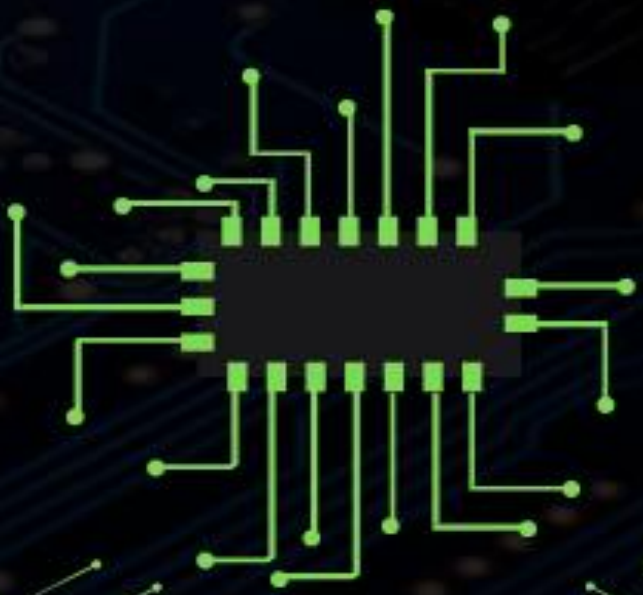
Ebook por
Wenderson Anjos

UM PROCESSADOR DIGITAL FEITO DO ZERO - COMO FUNCIONA?



SUMÁRIO

INTRODUÇÃO

Os computadores digitais trouxeram uma variada fonte de conhecimento para a vida humana, possibilitando que as pessoas trabalhassem em suas tarefas cotidianas de maneira produtiva e criasse novos produtos para diversos tipos de consumidores. Desde máquinas industriais, até dispositivos portáteis e computadores de mesa, administra uma vasta produção de inúmeros outros projetos, físicos e digitais.

O software de computador é um destes projetos digitais que melhorou significativamente a vida das pessoas e empresas. Entretanto, para ser possível um software digital funcionar, é necessário uma série de componentes físicos interligados, compondo um sistema ordenado de execução a fim de executar uma tarefa. Por sua vez, tais componentes integram uma lógica de funcionamento interno.

Esta integração foi estabelecida em várias gerações da computação, que se originaram das máquinas de Charles Babbage como a máquina diferencial que realizava cálculos trigonométricos através de um programa matemático criado por Ada Lovelace e a máquina de Alan Turing, que salvou várias pessoas na guerra alemã, através de processos de decodificação de dados criptográficos.

A arquitetura de John Von Neumman, revolucionou a computação, definindo um modelo de comunicação memória-processador e a capacidade de executar rotinas de programação imperativa (Assembly). Tudo isto só foi possível pela criação de processadores digitais – O cérebro da máquina. Um processador opera com bilhões de transistores, no qual usando um rearranjo transistorizado específico é possível construir componentes lógicos (portas lógicas).

No momento que criamos uma organização planejada destas portas, criamos circuitos combinacionais para processar uma tarefa e circuitos sequenciais para armazenar um dado. Quando tudo isto é integrado de maneira estratégica, construímos pequenas arquiteturas capazes de realizar tarefas mais complexas, onde a responsabilidade de procedimentos específicos é dividida em vários componentes ou arquiteturas.

Quando reunimos cada micro arquitetura em uma só, desenvolvemos um mecanismo central de um computador, que chamamos de CPU (Central Processing Unit). É através desta CPU que programas finais em código de máquina são processados, códigos estes idealizados durante a construção do processador por cada fabricante.

O código de máquina, melhor representado textualmente, se transformou em uma linguagem chamada Assembly, no qual cada fabricante possui os seus próprios formatos, seguindo certas convenções padronizadas. Com o nascimento deste modelo de programação, o desenvolvimento de sistemas operacionais e firmwares se tornou tarefas mais versáteis, possibilitando que outros projetistas criasse os compiladores para aproximar a sintaxe de códigos mais da vida humana e criação de programas maiores.

Tais programas maiores foram exercidos pelos programadores e desenvolvedores. Os desenvolvedores de softwares desempenham um papel fundamental na criação de softwares usando estas sintaxes de programação, o que só foi possível pela existência de sistemas operacionais e drivers rodando em cima de processadores. No entanto, há alguns impasses no ramo do conhecimento de computação.

Mesmo que desenvolvedores criem softwares e produtos empresariais facilitando a vida do usuário e do consumidor, é possível existir uma certa “indiferença” no conhecimento de como a computação realmente funciona. Isto se dá pelo motivo da exigência de mercado, que espera mais produtividade para melhor lucratividade em um dado período de tempo. Entretanto, conhecer melhor a computação, proporciona a “intimidade” com a máquina, colaborando com uma ciência de algoritmos mais eficientes.

Visando este contexto, o intuito deste E-book é apresentar um mundo extremamente importante, mas que não recebeu total atenção pelos novos desenvolvedores. Neste mundo, você aprenderá a linguagem das máquinas – Os circuitos lógicos digitais. Mostraremos um processador criado do zero, usando um simulador, ferramenta necessária para a projeção de hardware.

Bons Estudos!

1. CIRCUITOS PARA DESLOCAMENTO DE BITS

Na área de processamento digital, nós temos inúmeros tipos de circuitos combinacionais que podem ser feitos usando portas lógicas, tais circuitos são utilizados em microcontroladores, processadores, controladores e diversos outros componentes que irá processar uma tarefa e entregar um resultado. O deslocador de bit é um destes circuitos essenciais para o funcionamento de tarefas triviais, que veremos logo adiante.

Em programação alto nível e baixo nível utilizamos operadores e instruções para processar o deslocamento de bits em um registrador ou variável. Se estamos falando de C por exemplo, podemos criar uma variável com o número 160 e querer realizar uma "divisão" rápida por um número múltiplo de 2, exemplo: 16. Se dividimos 160 / 16, o resultado é 10, já que 16 x 10 é igual a 160. Isto é o mesmo que deslocar 4 bits a direita, como 160 >> 4 = 10, ou minha_variavel >> 4 (Onde "minha_variavel" contém o número 160).

Também é o mesmo que dizer em binário: 10100000b >> 4 = 00001010, perceba-se que o dado "1010" que estava nos 4 bits mais significativos (mais altos) do valor, se deslocou 4 bits a direita, ou seja, foram para os 4 bits menos significativos do valor.

E se quisermos fazer uma "multiplicação" rápida por um múltiplo de 2? Isto também é possível, mas ao invés de utilizar deslocamento pra direita, agora é o oposto: Deslocamento para a "Esquerda". Pegamos como exemplo o dado 8 em binário (00001000b), se quisermos multiplicar 8 por 2, dando o número 16, apenas desloque 1 bit para a esquerda, como: 00001000b << 1 = 00010000b (Número 16 em binário). Perceba-se que o estado lógico alto (1) que estava na posição 3 (bit<3> contando da direita para esquerda), ele agora está na posição 4 (bit<4>, lembrando que o primeiro bit começa na posição 0).

Estes cálculos também são utilizados quando você pretende trabalhar apenas com os bits mais altos de um dado valor, então você normalmente "isola" por uma instrução AND os 4 bits mais altos do valor e desloca ele para a direita, por exemplo: Digamos que você tenha o valor 11000011 (Número 195 em

decimal). Você quer "descartar" os 4 bits mais baixos, que é o 0011, e quer apenas "considerar" os 4 bits mais altos, que é o 1100.

Neste cenário, basta realizar uma operação AND de 11000011 com 11110000b (Nesta operação, apenas os 4 bits mais baixos são zerados, deixando os mais altos), e depois deslocando 4 bits para a direita, como em Assembly: Se o dado a ser deslocado estiver no registrador AL por exemplo, basta utilizar a instrução SHR (Shift Right) - SHR AL, 4 ou 11000000 >> 4. AL terá o valor 00001100.

Pronto! Agora você pode trabalhar com o número 12 em binário de forma isolada/separada. Isto é muito utilizado quando você quer fazer novas operações aritméticas ou extensões de operações, como criar um multiplicador de 32 bits para uma arquitetura que só trabalha com 8 bits.

1.1. Registrador de Deslocamento (Shift Register)

Compreendido como funciona os deslocamentos, vamos entender as imagens abaixo. Nós temos 2 tipos de deslocadores: O deslocador via clock, também chamado de "Shift Register" ou "Registrador de Deslocamento" e o Deslocador Barrel via multiplexadores, primeiro entenderemos o shift register apresentado nesta 1ª imagem:

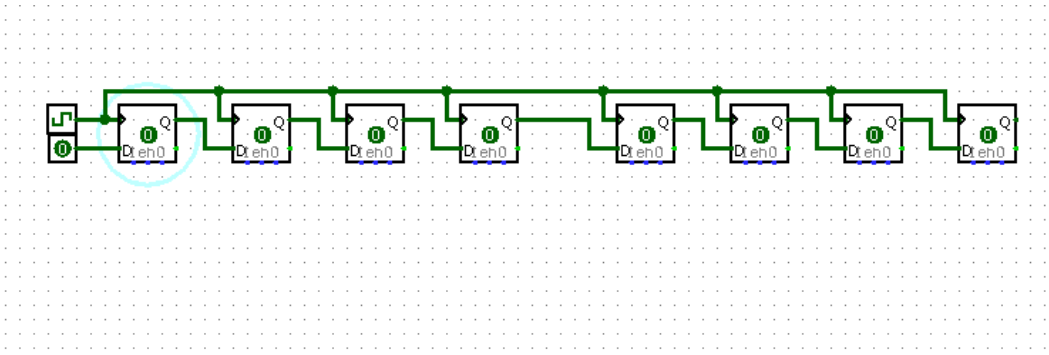


Figura 1 - Registrador de Deslocamento de 8 bits via Clock

Este tipo de deslocador, utiliza uma série de Flip-Flops tipo D encadeados, ou seja, cada flip-flop é uma célula de memória (Com 2 latches de portas NOR) que pode armazenar o dado 1 ou 0, sua saída Q é enviada para a entrada D do próximo Flip-Flop, no entanto, o pulso de clock que oscila de 0 pra 1 e de 1 pra 0, constantemente, irá se conectar na entrada "Clock" de todos os Flips-Flops.

Isto significa que se cada Flip-Flop está configurado pra armazenar a 1ª entrada D na "transição de subida" (Borda de Subida, de 0 para 1), neste gatilho de transição, o 1ª Flip-Flop irá armazenar o que está na sua entrada D, digamos, 1. No 2ª pulso de Clock, o próximo valor da entrada irá ser armazenado no 1ª Flip-Flop, porém o 2ª Flip-Flop vai armazenar o que está no 1ª Flip-Flop, ou seja, a saída do 1ª vai pra entrada do 2ª.

Portanto, a cada pulso, a saída da célula anterior vai para a entrada da célula posterior, e assim nós temos um deslocamento de bits 1 vez pra esquerda a cada pulso.

Este tipo de deslocador via clock é muito utilizado no armazenamento serial, quando você pretende utilizar apenas uma linha de conexão pra enviar um dado de 16 bits por exemplo, logo você precisaria de 16 Flip-Flops, um enviando pro outro, em 16 pulsos, cada pulso escrevendo 1 bit. Isto é, um armazenamento em série. Mas e se você quiser fazer igual a instrução SHR ou SHL em Assembly? Ou nos demais operadores de deslocamento em programação, como eu deslocaria vários bits de uma vez só via circuito? São estas perguntas que vamos responder no próximo tópico.

1.2. Deslocador Barrel

Considerando que cada pulso de clock determina os estágios de execução de uma CPU (Pra cada instrução lida da memória), não é trivial utilizar este o shift register se você quiser criar uma "instrução de deslocamento", pois você teria um atraso muito grande, concorda? É aí que utilizamos outros tipos, como o: O Deslocador Barrel.

Este tipo de deslocador utiliza multiplexadores (Denominado MUX), portanto, precisamos entender como um multiplexador funciona usando portas lógicas.

1.2.1. Multiplexadores

Os multiplexadores são componentes que servem para *selecionar* uma linha de conexão e transmitir para a saída. Eles são utilizados para “chavear” um

dos dados de várias fontes em uma única saída. Este chaveamento é feito via endereçamento das entradas, que veremos a diante.

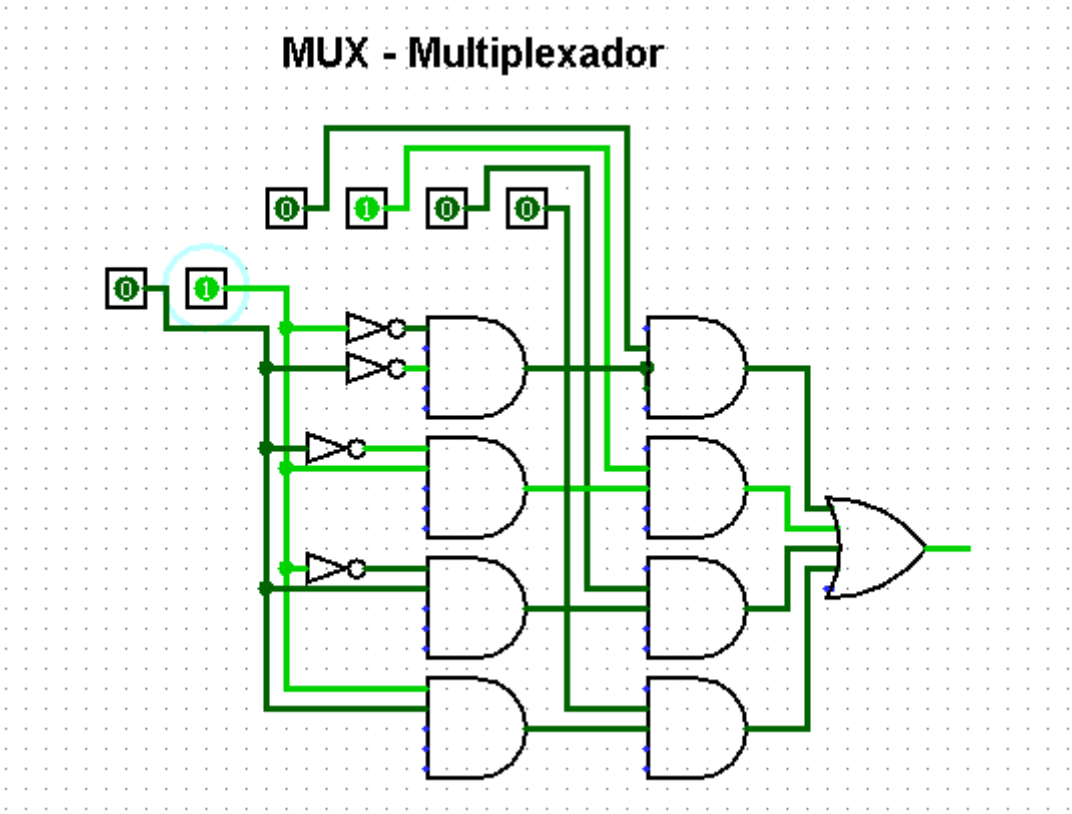


Figura 2 - Multiplexador 4x1 (Endereçamento de seleção de 2 bits)

Nesta imagem vemos 8 portas AND e 1 porta OR conectadas em 6 entradas (botões de bits). A 1ª entrada de 2 bits é o endereçamento da combinação de seleção, ou seja, quando for 00, vai selecionar a 1ª entrada de 4 bits, quando for 01 seleciona a 2ª entrada de 4 bits, 10 a 2ª entrada e 11 a 3ª. Ou seja, a entrada selecionada será enviada para a saída, portanto, precisamos de um "seletor" que seleciona uma porta AND específica baseado no endereçamento, podendo ter até 4 combinações específicas, ou seja, 4 portas AND, cada uma relacionada a 1 entrada.

A porta AND que tiver selecionada, vai pegar a sua entrada e enviar para um OR. Através desta "Mescla" de resultados via OR, é que nós temos um multiplexador. Então em termos simples, um multiplexador serve para endereçar uma entrada, aquela que tiver selecionada, será enviada para a saída, se for 1, a saída será 1 e se for 0, a saída será 0, da entrada selecionada.

Sabendo como funciona um multiplexador, agora vamos para a 3ª imagem, que mostra 2 deslocadores separados - Um deslocador para a esquerda e o outro sendo deslocador para a direita:

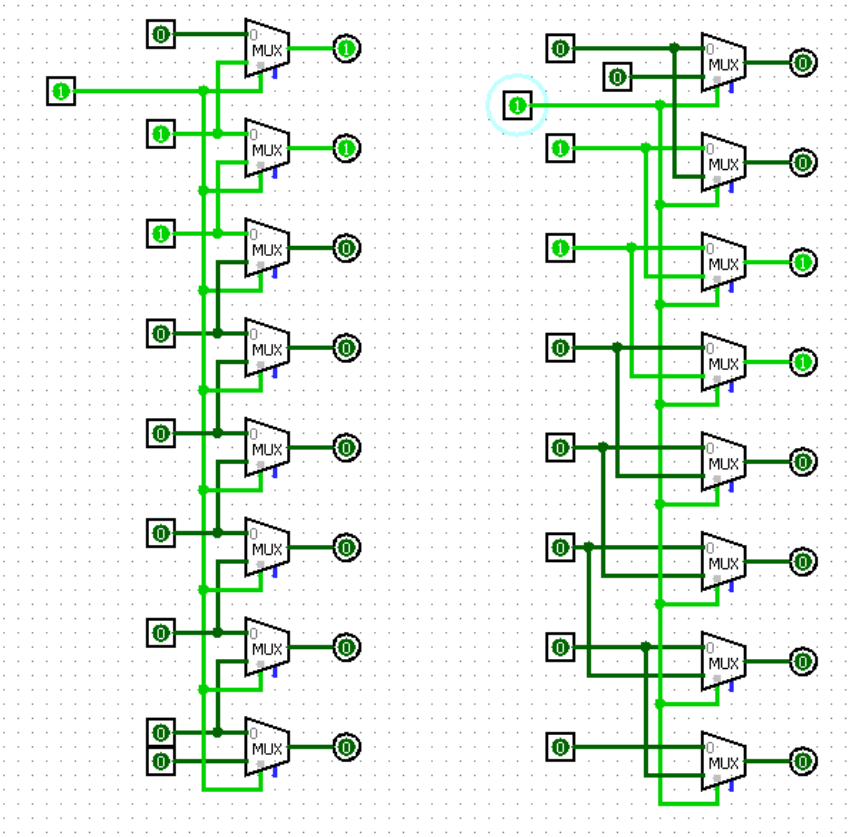


Figura 3 - Deslocador de 1-bit para a esquerda e deslocador de 1-bit para a direita

Considerando que estes dois deslocadores são de 8 bits cada, então teremos 8 multiplexadores. Nos dois tipos de deslocadores, a entrada do botão a ser enviada, está na entrada 0 do multiplexador. E embaixo de cada multiplexador, nós temos os bits de endereçamento, então considerando que este é um multiplexador de 1 bit, significa que ele pode endereçar até 2 entradas (2 estados - 0 ou 1). Se ele for estado 0, a entrada 0 do multiplexador é enviado pra saída, mas se ele for 1, a entrada 1 do multiplexador é enviado pra saída, logo o estado sendo 0, não teríamos nenhum deslocamento.

No entanto, se o endereço for estado 1, o que tiver na entrada 1 será enviada e podemos perceber que no 1ª deslocador a esquerda (que está na figura 3) , a entrada 1 do 1ª multiplexador está conectada na entrada 0 do 2ª multiplexador, então a entrada 0 do 2ª vai pra saída do 1ª, a entrada 0 do 3ª vai pra saída do 2ª, a 0 do 4ª vai pra saída do 3ª, e assim por diante.

Vocês podem ver que temos estas conexões entre as entradas. Isto é o que faz um conjunto de multiplexadores deslocar 1 bit para a esquerda (Considerando que nós estamos lendo de baixo para cima). Como pode ser visto, o dado 110b se transformou em 011b (O mesmo que dividir 6 por 2 = 3).

Já o deslocador para a direita da *figura 3*, o processo de conexão interligada é o inverso, ou seja, se o 1ª deslocador conectava cada entrada '1' de multiplexador na entrada '0' do próximo multiplexador, o deslocador a direita irá conectar cada entrada 0 de multiplexador na entrada 1 do próximo multiplexador, perceberam a inversão? É isto que caracteriza o deslocamento para direita, como podemos ver, o dado 110b se transformou em 1100b (O mesmo que multiplicar 6 por 2 = 12).

Mas existe algo que não foi dito, é que se a gente tem uma interconexão entre entradas de cada multiplexador, logo vai chegar uma hora que o último multiplexador não terá uma entrada a quem se interligar, correto? Significa que por padrão ele deve ser 0, da mesma forma, é o deslocador pra direita, o outro extremo inverso do deslocador, deve ser por padrão 0, já que não teria nenhum multiplexador depois dele.

Isto é uma lógica fundamental para a construção de circuitos de deslocamento de bits, já que um deslocamento para a direita, é determinada pelo zeramento dos bits mais à esquerda e o deslocamento para a esquerda, determina o zeramento dos bits mais a direita (O inverso). Ou seja, deslocar para direita faz com que perdemos os últimos bits a direita (Eles praticamente somem) e deslocar para esquerda faz com que perdemos os últimos bits a esquerda (A partir daí, é possível criar flags de Carry Out, da mesma forma que, quando uma soma ultrapassa o tamanho máximo de bits).

1.2.2. Conexão de Deslocadores Unidirecionais

Okay, mas afinal como o deslocador barrel realmente funciona? Simples, sabendo que nós temos um deslocador da figura 3, usando multiplexadores, basta você interligar os dois deslocadores, no entanto, sendo apenas de uma única direção - Ou pra esquerda ou pra direita (Por enquanto). Claro que podemos utilizar as duas direções em um deslocador barrel, mas vamos pôr

como exemplo apenas uma direção. Na figura 4 conseguimos perceber esta interligação:

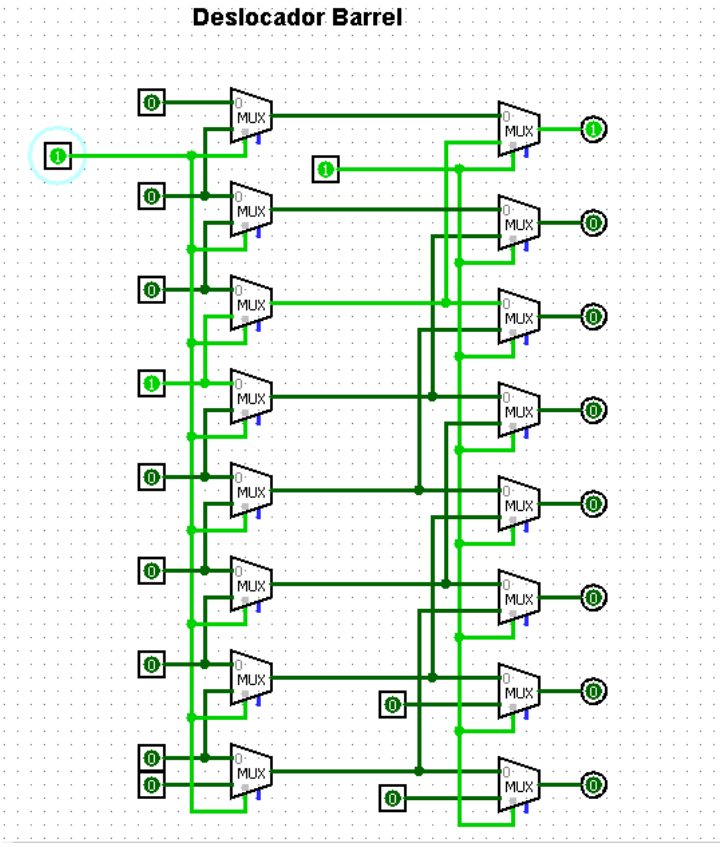


Figura 4 - Interconexão de 2 deslocadores múltiplos de 2 (deslocamento de até 3 bits)

A saída do 1ª deslocador, vai para a entrada do 2ª deslocador. Mas é tão simples assim? Quase! Ainda existe uma pegadinha: O 2ª deslocador tem que ser múltiplo de 2, seguindo a base binária. Por exemplo: Se você tem 3 deslocadores interligados, o 1ª deslocará 1 bit para a esquerda, o 2ª deslocará 2 bits para a esquerda e o 3ª deslocará 4 bits para a esquerda, reparou na quantidade de bits de cada deslocador? Você vai seguindo essa sequência: 1, 2, 4, 8, 16, 32, ... etc.

Logo, se um deslocador de 1 bit, interligava sua entrada ‘1’ na entrada ‘0’ do próximo multiplexador, o deslocador de 2 bits vai interligar sua entrada 0 no "próximo do próximo" multiplexador, ou seja, você "salta" um multiplexador. O deslocador de 1 bit não saltava, o deslocador de 2 bits saltava 1 multiplexador, logo, o deslocador de 4 bits para a esquerda vai saltar (4-1) multiplexadores, e assim por diante.

Nesta interligação do deslocador barrel, o que de fato acontece é a "soma" das bases binárias na potência da posição do deslocador, isto é, se lembram do

estado que "endereçava" as entradas do multiplexador? Então, se no 1ª deslocador este estado for 0, a entrada 0 será selecionada, se for 1, a entrada 1 será selecionada. Portanto, se o estado for 0, é o mesmo que "desabilitar" o deslocamento de bits do deslocador, pois sua saída será a mesma da entrada.

Então, se você tem este mesmo estado de endereçamento para cada deslocador, o estado que for 1, irá habilitar aquele deslocamento. Vamos colocar como exemplo o que acontece na figura 4, o dado de entrada foi 1000b (8 em decimal). O 1ª deslocador está habilitado, então ele vai deslocar 1 bit para a direita, resultando em sua saída o valor 0100b.

O 2ª deslocador também está habilitado, no entanto, ele desloca 2 bits para a direita, portanto ele vai pegar a saída do 1ª (que é 0100b) e vai deslocar 2 bits, se tornando o valor 0001b, logo no total eles deslocaram quantos bits? 3 bits, pois 1 bit de deslocamento do 1ª + 2 bits de deslocamento do 2ª = 3 bits. Ele realizou a soma de 1 + 2, devido aos dois deslocadores estarem habilitados.

Esta sequência de habilitação, forma um dado numérico binário que se corresponde ao "valor" de deslocamento, pois pode-se ver na imagem que este valor nos "estados de endereçamento" forma o binário 11, ou seja, o decimal 3. Se quisermos deslocar 5 bits para a esquerda? Teremos que ter o valor 101 nos estados de endereçamento de 3 deslocadores, sendo o 1ª **habilitado** (1), o 2ª **desabilitado** (0) e o 3ª **habilitado** (1), assim ele somaria 1 + 0 + 4 = 5 bits de deslocamentos (Esta é a soma da base binária exponenciado na posição do deslocador).

1.2.3. Deslocadores Bidirecionais

E se quisermos deslocar tanto para a direita, quanto para a esquerda, usando deslocadores de até 7 bits? Isto é possível se você fizer igual está na Figura 5. Considerando tudo que falamos até aqui, este deslocador barrel é o mais completo, pois ele terá tanto a seleção de qual "direção" ele vai deslocar, quanto que a "quantidade" de bits que vai deslocar:

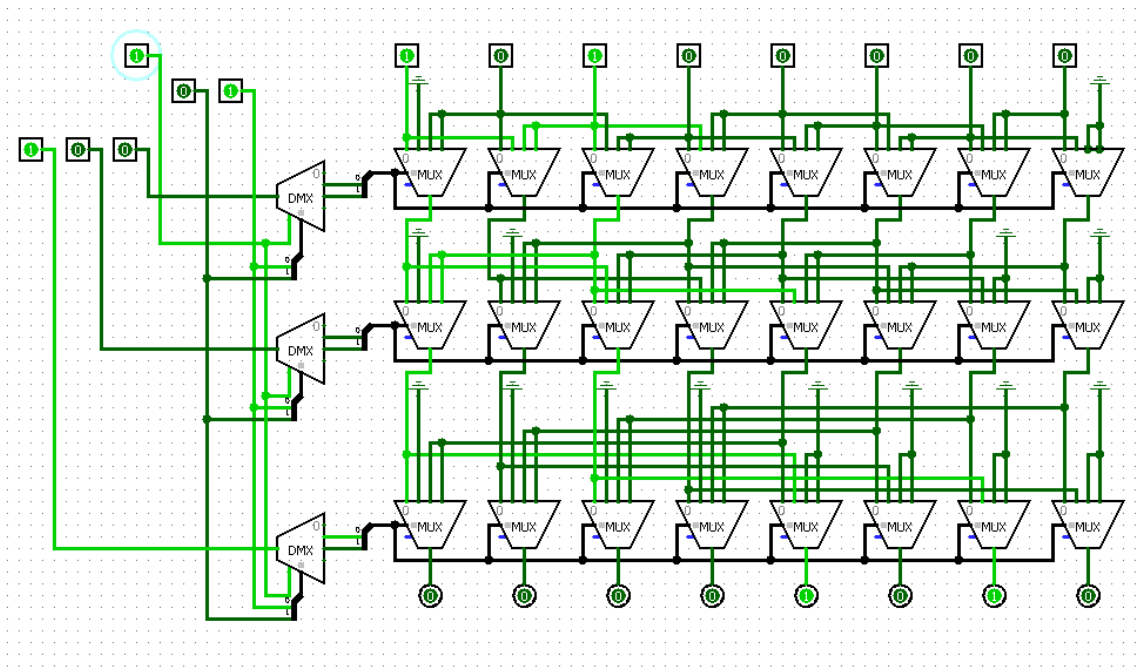


Figura 5 - Deslocador Barrel completo de 7 bits de deslocamento (Right/Left)

O conceito é o mesmo que explicamos, porém com os deslocadores alinhados horizontalmente, desta forma, o 1ª deslocador está na linha 0, o 2ª deslocador na linha 1 e o 3ª na linha 2. A linha 0 vai receber o dado inicial nas entradas 0, onde ele terá 4 entradas para cada multiplexador, logo, elas serão selecionadas por um endereçamento de 2 bits. Cada entrada Nº 1 na entrada Nº 0 do MUX mais à esquerda (deslocamento para a direita) e cada entrada Nº 2 e Nº 3 na entrada Nº 0 do MUX mais à direita (deslocamento para a esquerda), isto se tiver deslocando 1 bit para uma das direções, mas se for 2 bits, como na linha 1, você usa a mesma conexão anterior mas saltando 1 multiplexador, e se for na linha 2, saltando 3 multiplexadores.

A saída de cada deslocador, será o dado de entrada do próximo deslocador. Então, se no endereçamento tivermos o valor 00b, ele não vai deslocar para nenhuma direção, sendo um resultado constante, se for 01b, vai deslocar para a direita, e se for 10b para a esquerda e se for 11b, valor constante também (igual no caso do 00b).

Pode-se ver que no 1ª deslocador, nós temos um Ground (GND) que zera os MUX do lado mais extremo (Esquerda e Direita), já no 2ª deslocador, nós temos 2 GNDs de cada lado e no 3ª temos 4 GNDs de cada lado, logo, se temos 8 bits totais e 7 deslocamentos, e quisermos adicionar um 4ª deslocador, poderíamos deslocar até 15 bits para a direita, que neste caso teríamos 8 GNDs de cada lado, ou seja, 8 GNDs na entrada 0 e 8 GNDs na entrada 2 e 3 (Este é

o conceito que eu expliquei sobre "zerar" os bits numa direção oposta a que está deslocando, é a ênfase do deslocamento de bits).

Agora, para selecionar as entradas, que é no caso do endereçamento (O que vai determinar se é instrução SHR ou SHL), ou utilizaríamos portas AND ou um Demultiplexador (DEMUX). Utilizar um DEMUX é mais prático e rápido, já que a lógica é a mesma. O DEMUX vai funcionar como um "Seletor". Assim para cada linha de deslocamento, teríamos um DEMUX na entrada, os dados de saída de cada DEMUX vão para o endereçamento da linha de deslocamento.

A entrada de cada DEMUX, será o estado lógico que vai determinar se aquela linha será habilitada ou não, e como sabemos, se ela for desabilitada, não irá deslocar nenhum dado, mas se habilitar, o seu deslocamento será somado com outros deslocamentos (O que expliquei sobre as somas). No entanto, o endereçamento "fonte" que vai determinar a direção de deslocamento, ele deve se conectar no endereçamento de seleção de todos os DEMUX, pois se não fosse assim, teríamos erros de usar 2 direções opostas em deslocadores diferentes, significando em erro no resultado.

Portanto todos os endereçamentos são iguais, pois todos os deslocamentos irão deslocar para a mesma direção, com a diferença na quantidade de deslocamentos, incluindo se ele estará habilitado ou não.

Para finalizarmos, pode-se ver na figura 5 que o dado de entrada é 10100000b (160 em decimal), selecionamos o endereçamento 01b (Deslocamento para a direita) e determinamos a quantidade de deslocamento sendo 100b (4 deslocamentos), e o resultado na saída é 00001010b (10 em decimal), pois 160 >> 4 = 160 / 16 = 10 (São 4 deslocamentos pois 100b é 2^2x1 + 2^1x0 + 1x0 ou 4 + 0 + 0 = 4).

Se quisermos pegar o valor 10 em decimal e deslocar 4 bits para a esquerda pra recuperar o valor original, basta alterar o dado de entrada para 00001010 e mudar o endereçamento de 01b para 10b em binário e pronto! Temos um deslocador barrel dinâmico!

Através do deslocador, será possível deslocar até 7 bits de um registrador acumulador no processador, e armazenar dados de 8 bits para uma arquitetura que só permite o armazenamento de 4 bits por vez. Portanto, usando uma

instrução como esta: **SHL 7** (Ou 10100111 no formato de instrução), é o mesmo que fazer **DR << 7** (Acumulador deslocado 7 bits a esquerda), e com 4 bits na parte de dados do formato de instrução, será possível deslocar até 15 bits, ou seja, temos uma possibilidade de termos registradores acima de 8 bits, com uma instrução de deslocamento já pré-adaptada para estes casos.

2. CIRCUITOS DE ENTRADA E SAÍDA

Focaremos no funcionamento da "Unidade de I/O", com suas portas de comunicação de dispositivos externos (Entrada e Saída - E/S). Abaixo como demonstração, apresento um programa Assembly que exibe um *Hello World* na tela executando no WR80, lendo as Strings da memória RAM e escrevendo no monitor, usando portas de I/O.

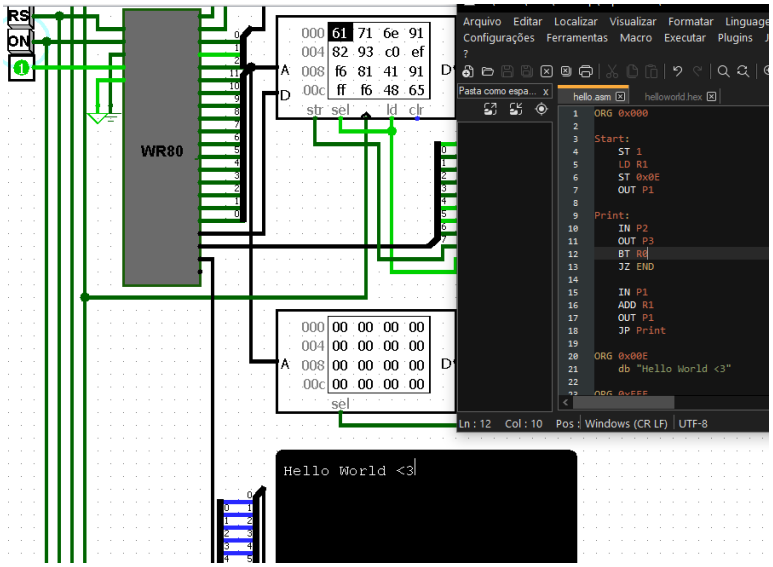


Figura 6 - Hello World no próprio processador

Quando pensamos em "portas" de um processador, nos lembramos na questão de "controladores", que são circuitos integrados (CIs) externos do processador responsáveis por se comunicar com algum dispositivo de hardware. Cada dispositivo de hardware tem a sua função - Ela recebe entradas por uma solicitação, que passam por um processamento, e envia suas respostas para quem solicitou, imagine isso como uma arquitetura cliente-servidor.

Mas por si só, os dispositivos de hardware precisam de um controle mais inteligente, de um mecanismo intermediário que vai "interfacear" a comunicação

entre o solicitador (A CPU por exemplo) e os próprios dispositivos (Um HD por exemplo), e neste meio da comunicação, nós teríamos um controlador (como o controlador SATA).

Normalmente um controlador não é tão difícil de projetar comparado com o processador, uma vez que ele desempenha um papel muito específico de controle, com poucos comandos decodificados, no entanto, são dispositivos fundamentais que "complementam" o funcionamento de uma CPU, ou seja, separando as responsabilidades em mecanismos distintos reduzindo a carga de tarefas de uma unidade de processamento.

Observando este cenário, a CPU precisa de pinos externos enumerados, que possam ser acessados pelo Assembly via instruções de I/O, como as instruções da Intel: IN e OUT. Estas são as mesmas que estou projetando, porém numa arquitetura RISC e não CISC (como Intel), ou seja, é um modelo de processador SAP (Simple As Possible), falarei melhor sobre isso adiante.

No Assembly dos microcontroladores PIC, é comum utilizar registradores associados com os pinos externos (as chamadas "portas") e cada registrador tem um nome: PORTA, PORTB, PORTC, ... etc. Os dados nestes registradores de portas são movimentados utilizando a instrução movf, movlw e movfw, ou seja, entre o acumulador W e as portas (ou outros registradores como as "GPRs").

Os bits que estiverem definidos neles, irão para os pinos correspondentes, portA terá seus 8 pinos, enquanto que portB terá seus 8 pinos, cada pino equivalente a 1 bit daquele registrador. É através destes registradores que dados podem ser enviados para outros dispositivos, seja para ligar um LED, ler um botão, uma transmissão serial (bit-a-bit), até mesmo comunicação com controladores mais complexos (rede, vga, etc.).

Percebam-se que nas tarefas que são desempenhadas, que mencionei, eu disse "ligar um LED" e logo em seguida, "ler um botão". Conseguimos observar que há dois direcionamentos diferentes: O primeiro sendo saída, do processador para o dispositivo; O segundo sendo entrada, do dispositivo para o processador. Este direcionamento de entrada ou saída normalmente pode ser configurado no microcontrolador utilizando "registradores de funções especiais"

(SFRs) e cada porta poderá ter seu direcionamento, em muitos casos, cada pino individualmente.

Isto acontecem principalmente nos casos de microprocessadores RISC, agora no cenário dos processadores CISC como Intel e AMD, eles só podem definir um direcionamento por vez, já que quem define este direcionamento é a própria instrução de I/O, onde tal instrução, ou vai ler um "número" de porta (IN), ou vai escrever neste "número" de porta (OUT) e cada número de porta se relaciona com um único controlador, que por sua vez se relaciona com o dispositivo de Hardware.

Perceba-se que eu destaquei duas vezes a palavra "número", justamente porque esta é uma das diferenças das arquiteturas CISC com a RISC, pois normalmente Intel utiliza enumerações de portas de dispositivos, que são endereçamentos. Já no RISC, como o PIC, eles utilizam aqueles registradores nomeados, podendo também ser números endereçáveis.

Sabendo destas informações, vamos conhecer como funciona a unidade de I/O do meu próprio microprocessador de portas lógicas - O WR80.

2.1. Funcionamento das Portas de I/O

Na minha arquitetura projetei 4 registradores de portas: P0, P1, P2 e P3. A abreviação vem de Port 0, Port 1, etc. P0 e P1 são portas de endereços, enquanto que P2 e P3 são portas de dados. Cada um deles são registradores de 8 bits, que funcionam de uma maneira diferente dos registradores de usuário: R0, R1, R2 e R3, além do registro acumulador DR (Data Register).

Os registradores de usuário, também chamados de GPR (Registradores de propósitos gerais), são utilizados pra armazenar resultados de cálculos e símbolos, enquanto que os registradores de I/O (ou portas), são usados pra se comunicar com periféricos externos da CPU. As duas portas de endereçamento formam o par P0:P1, onde apenas 12 bits são utilizados para endereçar a memória RAM externa (mapeando até 4KB de RAM), e os 4 bits que sobraram? Então, eles são chamados de "bits de controle" e são eles que vão determinar o "direcionamento" das portas de dados (P2 e P3), direcionamento este que pode ser entrada ou saída.

Se temos dois estados, logo para saída será 1 e para entrada será 0, e se temos duas portas de dados, usaremos 2 bits de controle, cada um controlando uma porta: Bit 6 de P0 controla a porta 2 e bit 7 de P0 controla a porta 3. Assim como os bits 5 e 4 de controle, onde o bit 5 de P0 vai servir para enviar um sinal de acesso a memória RAM, dizendo a ela que a CPU quer ler ou escrever um dado e quem vai informar automaticamente neste bit 5 será as instruções IN e OUT.

Já o bit 4 servirá para seleção de memórias distintas, exemplo: A mesma pinagem de endereçamento de 12 bits P0:P1 poderá se conectar tanto em uma memória RAM, quanto em uma memória ROM, de mesmo tamanho, portanto, quando o bit 4 = 0, por padrão é a RAM que é selecionada, e quando o bit 4 = 1, vai selecionar a ROM. Então ao invés de termos um endereçamento de 12 bits (de 0x000 até 0xFFF), teremos de 13 bits, analisando de forma completa (Até 0x1FFF), que seria 4096 bytes para cada tipo de memória.

Já o registrador P2 será a porta de dados da RAM ou da ROM, por onde os dados serão lidos ou escritos, considerando o endereçamento de P0:P1, no entanto, a busca de instruções da memória para o decodificador, não escreverá os endereços em P0:P1, ao invés disso, manterá o valor constante destas portas e o endereçamento enviará direto de PC (Program Counter) para os pinos, que é um contador que endereça cada byte/instrução da memória e alimenta o IR (Registrador de Instrução), além de conduzir as 4 seleções de estágios.

O que de fato acontece é um "chaveamento" entre PC e P0:P1 de forma dinâmica. A porta P3, por sua vez, vai poder se comunicar com outros dispositivos do LogiSIM, como: Um monitor, um teclado, ou outro controladores que tem como seu dado de entrada 8 bits. P3 normalmente será um monitor TTY, parecido com um terminal, apenas para imprimir textos. Desta forma, através das instruções IN e OUT, o microcontrolador vai poder movimentar dados de entrada, da porta P3 referente a um dispositivo, para uma memória RAM ou ROM através da porta P2.

Utilizamos 2 bits no formato de instrução para mapear as portas (no IR), o que nos permite trabalhar com até 4 portas. No entanto, temos mais 2 bits sobrando, totalizando 4 bits, o que nos possibilita "escalonar" o nosso

processador para mais 12 portas (total = 16). Isto nos permite conversar com muitos dispositivos interconectados na CPU.

Tudo isso pra funcionar existe uma lógica, que vem da forma interna do banco de registradores de I/O, seus pinos e suas condições externas de escrita ou leitura, até a divisão do processamento em etapas (dos 4 estágios do pipeline) e o chaveamento de mecanismos controlados por bits. Esta mecânica vamos compreender analisando as próximas imagens.

2.2. Movimentação de Dados

Neste tópico será visto sobre o funcionamento das instruções de movimento de dados, visando se aprofundar nas características de I/O.

Na imagem abaixo é possível notar 4 dispositivos apontados pelas setas vermelhas. Cada dispositivo é um alternador/habilitador de uma instrução de movimentação, que apenas realiza a habilitação da entrada com a saída, através do 1ª pino que se for 1, o circuito estará habilitado e se for 0, desabilitado.

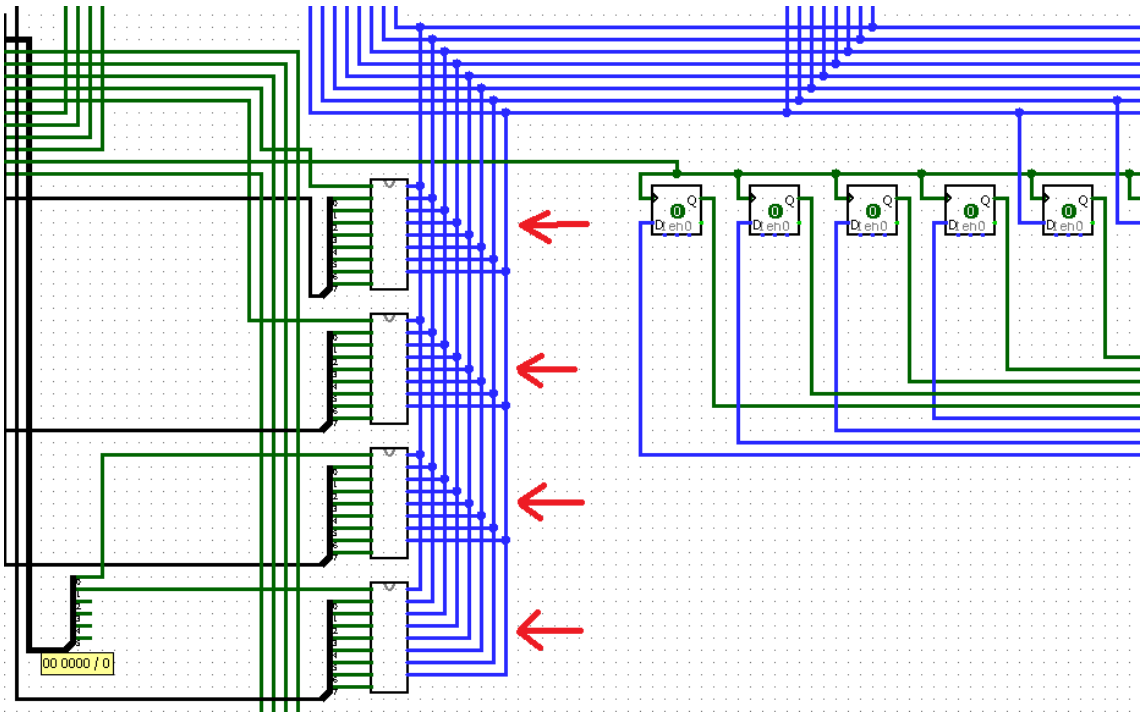


Figura 7 - Alternadores de instruções de movimentação na ULA

Este mecanismo utiliza "Buffers Controlados", que é como o funcionamento de uma porta AND, no entanto com um estado a mais, além de 0 ou 1, que é o estado "Desabilitado". Pode-se ver que quando as trilhas/conexões

ficam na cor azul, é porque está desabilitado (Nem é 0 e nem é 1), este tipo de modelo é chamado de "Circuito Tri-State", que como o próprio nome já diz, contém "3 estados": 0, 1 e desabilitado (Uma faixa elétrica desconhecida pelos estados).

O 1ª alternador habilita a saída dos dados da instrução ST (Store), o 2ª habilita os dados da instrução LD (Load), o 3ª da instrução IN (Input) e o 4ª da instrução OUT (Output). Todas as 4 operações são instruções da categoria "Movimentação".

Perceba-se que todas as 4 instruções estão interconectadas em um mesmo barramento de 8 bits, que está em cor azul, ou seja, se um alternador é desabilitado, logo sua entrada não interfere nas saídas de outros alternadores, podendo realizar inúmeras conexões em um mesmo barramento de saída.

Todas as instruções lógico-aritméticas, inclusive as de salto, enviam para este mesmo barramento, que irá por sua vez, enviar o resultado para um registro chamado "Registrador de Resultado da ULA" (Um registro de 12 bits que pode ser visto logo a direita, 8 bits para instruções comuns e 12 bits para saltos). No final do ciclo de execução, o resultado é salvo temporariamente neste registro, possibilitando copiar este resultado para destinos específicos, no estágio de escrita, baseado no tipo de instrução.

Na próxima imagem vemos dois habilitadores de entradas, das instruções IN e OUT, respectivamente, vindo na fase de leitura no início do estágio de execução. A diferença dos "habilitadores" com os "alternadores" nesta microarquitetura, é que habilitadores consideram utilizar "apenas" portas AND, pois obrigatoriamente se o estado não for 1, ele será 0, impedindo que ocorram conflitos de estados inexistentes em entradas de circuitos combinacionais.

Já os alternadores, eles consideram um 3ª estado apenas pelo fato de se conectar em um mesmo barramento, também impedindo conflitos de "vários estados 0s" nas mesmas conexões, podendo ocasionar leituras erradas. No caso das saídas de barramento dos alternadores, estamos usando Flip-Flops, que são dispositivos prontos do LogiSIM, possibilitando esse tipo de conexão, já os habilitadores, eles se conectam diretamente no circuito combinacional da operação na ULA.

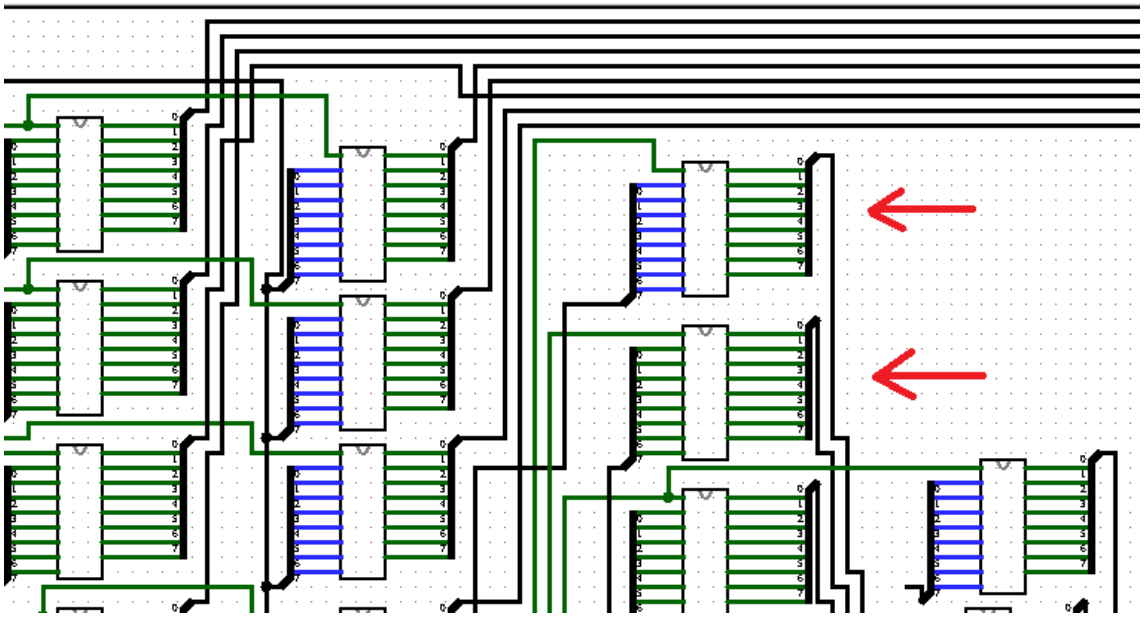


Figura 8 - Habilitadores de dados das instruções IN e OUT

Considerando isto, cada instrução tem seu próprio chip habilitador de entrada, a entrada poderá ser lida tanto que diretamente do "formato de instrução" (Como ST, SHR, SHL), quanto que "indiretamente", ou seja, dos registradores selecionados pelo formato de instrução (Como ADD, SUB, LD, IN, OUT).

Há também uma origem alternativa, de uma instrução que se diferencia de todas - As instruções de salto: JC, JZ e JP. A origem de leitura delas vem de PC - Program Counter, pois o processamento de salto é apenas somar PC = PC + OFFSET, onde o OFFSET é um registrador de 12 bits que armazena o endereço em 2 ciclos distintos, mas vamos deixar esta explicação para um outro tópico.

Já a saída dos habilitadores, se espalham pelo circuito em linhas de barramento de 8 bits, individualmente, cada uma enviando para uma operação combinacional da ULA, que por sua vez vão utilizar alternadores na saída pro registrador de resultado da ULA.

2.3. Registradores de I/O

As entradas dos habilitadores das instruções IN e OUT, como nesta próxima imagem, vem exatamente dos "Registradores de Portas de I/O".

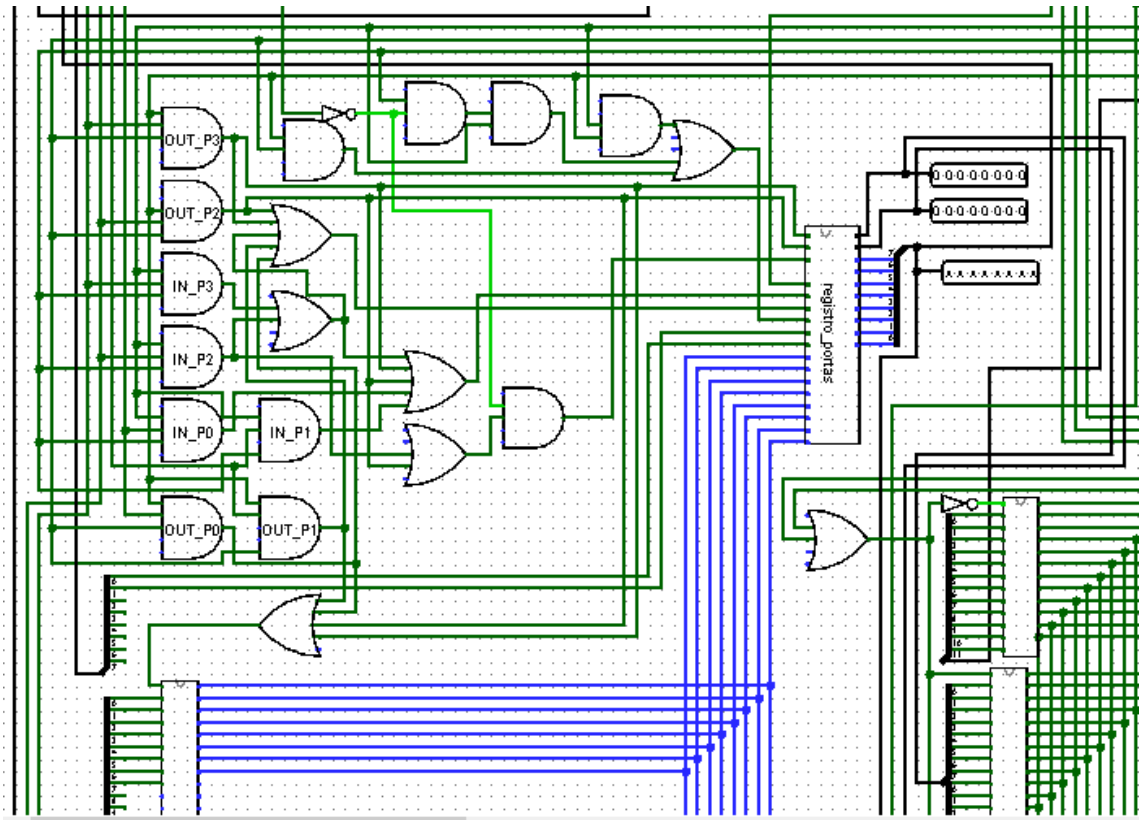


Figura 9 - Condições de escrita e leitura do banco de registros de I/O

Verifique que na imagem acima, nós temos um CHIP com 17 entradas e 3 conjuntos de saídas: 8 trilhas separadas + 1 barramento de 8 bits + outro barramento de 8 bits (Total = 24 trilhas de saídas). Este é o banco de registradores de portas de I/O.

Enquanto que em seu lado lateral esquerdo, temos o circuito combinacional que representa as "condições" de leitura ou escrita destes registradores, o que na verdade poderá também ser entendido como "Operações Condicionais" das instruções IN e OUT, já que eles também são processados tanto no estágio de execução, como no estágio de escrita.

A organização deste CHIP é necessária para o funcionamento das instruções de I/O, uma vez que o mecanismo dele se diferencia de qualquer outro registrador existente neste circuito, então antes de nos aprofundarmos nas condições, veremos mais internamente o que acontece nestes registradores.

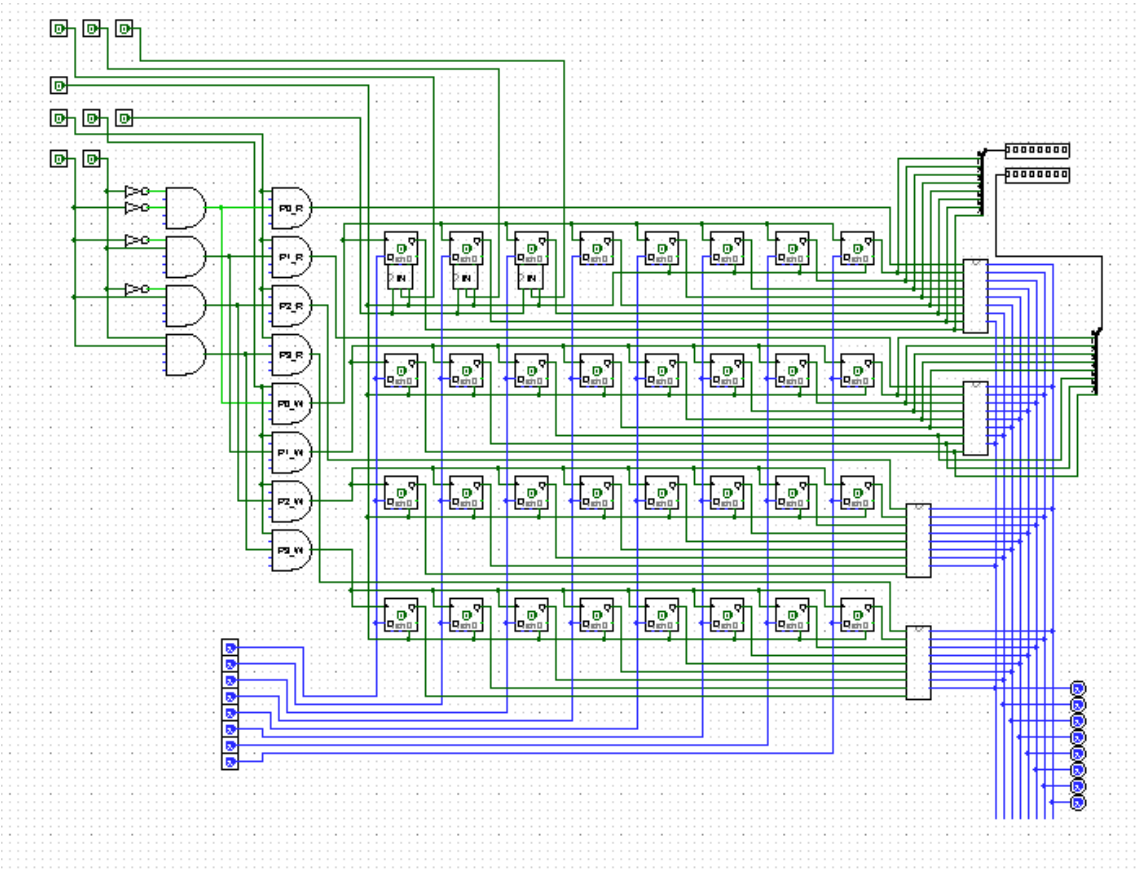


Figura 10 - Circuito interno da memória do banco de registros

Na imagem, temos a microarquitetura de um banco de registradores de I/O. São exatamente 32 Flip-Flops tipo D, que escrevem a entrada no "Nível Alto" (Não por transição de subida), e são agrupados por 4 linhas de 8 Flip-Flops, isto é, 8 bits. Cada linha se refere a 1 registrador: P0, P1, P2 e P3, respectivamente.

Os 3 bits mais altos de P0, contém dois tipos de entrada:

1. A entrada padrão "síncrona" no pino D que depende do nível alto de habilitação de escrita no pino CLOCK;
2. A entrada do tipo "Assíncrona" que utiliza outra via de habilitação, que não é a do clock, e escreve diretamente na célula (pelos 3 primeiros botões), através de um mecanismo de seleção (Pode-se ver um pequeno chip embaixo delas). Estes 3 bits são os bits de controle, onde o Bit<7> controla o direcionamento da porta P3, o Bit<6> controla o direcionamento da porta P2 e o Bit<4> define o acesso a memória.

A saída de P0 e P1 são sempre lidas, independente da habilitação de leitura, pelos dois barramentos separados de 8 bits. Estes dois barramentos vão endereçar a memória RAM na utilização das instruções IN e OUT, e elas por padrão devem ser sempre "Saídas".

No entanto, P0 e P1 também são selecionáveis, e podem se conectar em alternadores de saídas, via habilitação de leitura, assim como as outras portas P2 e P3. O próximo botão de entrada é o "Clear", e é responsável por limpar toda a memória dos registros assincronamente.

Nos próximos 3 botões de entrada , nós temos o 1ª que habilita a leitura de um dado registrador, enviando para uma das 4 portas AND iniciais que por sua vez irá habilitar o alternador de saída daquela linha, e o 2ª que habilita a escrita em um dado registrador, que também vão para portas AND, que são as últimas 4 portas AND, no entanto, ao invés de ir pros alternadores de saída como os habilitadores de leitura, desta vez o de escrita vai pro pino de clock da linha corresponde de Flip-Flops.

Já o 3ª botão de entrada também é habilitação de escrita, porém apenas para os 3 bits de controle de P0, mencionados anteriormente. Ou seja, eles habilitam o mecanismo de seleção assíncrona. Os últimos 2 botões de entrada, são os 2 bits de endereçamento dos registradores, que passam por um seletor (Juntando as portas AND, nós temos um DEMUX) que vai selecionar a porta AND daquela linha, ou seja, daquele registrador.

Se a gente mescla os bits de habilitação de leitura/escrita + os bits de seleção do registrador, temos um circuito completo para entregar na saída, via alternadores em um mesmo barramento, o dado do registrador selecionado ou entregar um dado ao registrador. Temos também +8 botões de entrada, que será literalmente o nosso dado a ser escrito, cada botão se conecta no pino D de cada coluna de todas as linhas, logo a linha que não for selecionada, a entrada será descartada.

2.4. Condições de R/W

Compreendido como funciona o circuito interno do banco de registradores, agora veremos sobre as "Condições de Escrita e Leitura". Na imagem abaixo, percebemos os destaques em vermelho, laranja e roxo. O que está apontado pela seta vermelha é o CI do circuito de registradores que acabamos de ver.

O destaque em vermelho, representa as condições para habilitar os bits de controle, que são elas que vão determinar tanto o acesso em memória, quanto a via por onde este acesso ocorrerá e a direção dos dados. Temos uma porta

OR que mescla o resultado de 4 portas AND, a condição pode ser lida como: "SE (o estágio de escrita for ativado E a instrução OUT for selecionada) OU (O estágio de execução for ativado + o nível baixo de CLOCK E a instrução IN for selecionada) OU (A instrução IN for selecionada E o estágio de escrita ativado) ENTÃO habilite a escrita dos bits de controle.".

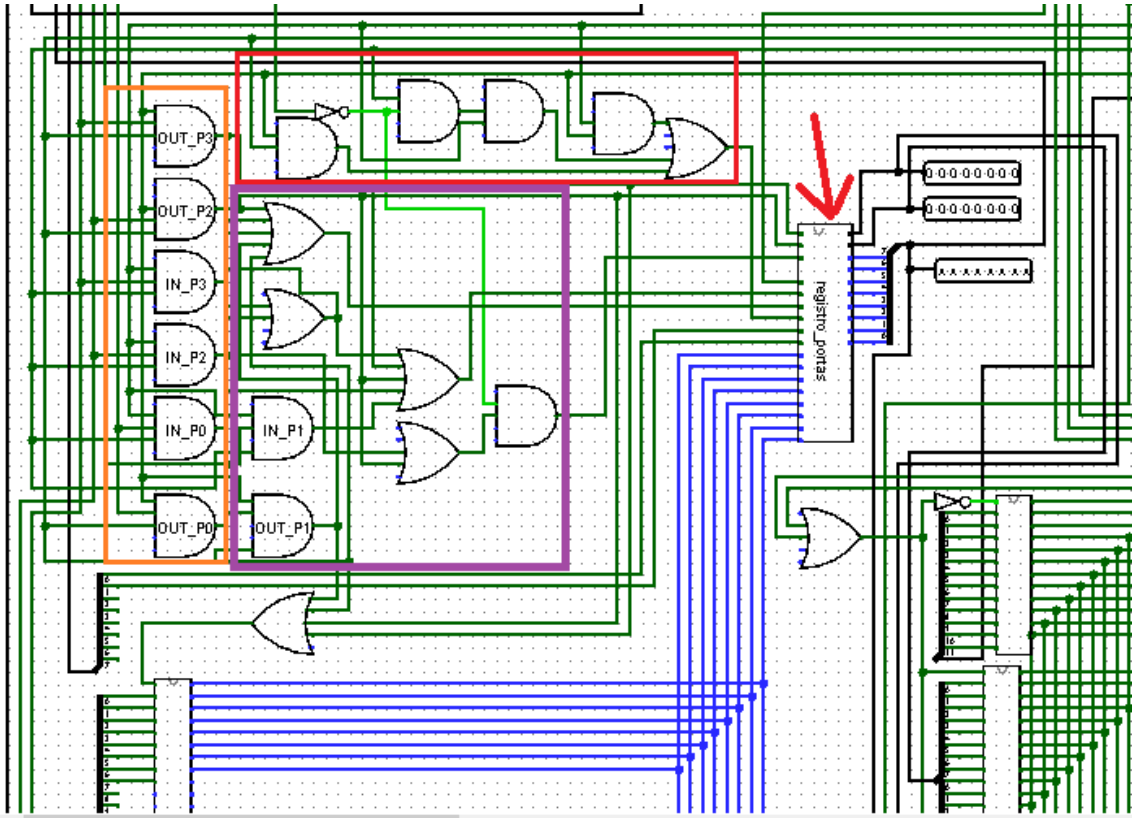


Figura 11 - Condições de R/W destacadas referente ao registrador de portas

Em um destes 3 casos, habilitaremos um bit de controle, que neste caso, vamos ler/escrever na memória ou ler/escrever em uma porta de dados. Percebemos que a instrução OUT age apenas no estágio de escrita, já que esta é a função da própria instrução - Escrever.

Já a instrução IN, age tanto no estágio de execução, como também de escrita, no entanto, ela considera o nível baixo de CLOCK na execução, devido a um "atraso" que devemos respeitar sobre as outras condições. Cada estágio tem dois níveis de clock: alto e baixo.

No caso das operações de I/O, no nível alto, realizamos a habilitação de escrita e leitura simultânea a uma porta selecionada, no nível baixo, acessamos a memória RAM lendo o conteúdo que está no endereço P0:P1 na porta P2 definida como entrada, porém no registrador sendo entrada e saída ao mesmo tempo, enviando ao habilitador. Com este atraso definido o Clock em nível baixo,

permitimos uma interconexão síncrona entre o "Final" do estágio de execução e o "Início" do estágio de escrita, para uma operação de leitura da porta de I/O escrevendo para o registrador de resultado da ULA.

Já que habilitamos a escrita dos bits de controle, precisamos definir "Quem" e "Onde" vão escrever estes bits, e para isto temos as próximas condições, que se referem ao destaque roxo e laranja.

Começando pelo laranja: "***SE** for estágio de escrita **E** seleção do registro P3 **E** instrução OUT **ENTÃO** ativar bit<7> de P0*", neste cenário, ele vai permitir que a porta P3 tenha sua direção como "Entrada" na instrução "OUT P3". No entanto, "***SE** for estágio de escrita **E** seleção do registro P2 **E** instrução OUT **ENTÃO** ativar bit<6> de P0*", aqui ele já direciona a porta P2 como entrada, ao invés de P3, na instrução "OUT P2". Vamos chamar estas duas condições de ***O3*** e ***O2***, respectivamente.

Ainda no destaque laranja, temos as condições do IN: "***SE** for instrução IN **E** seleção de P3 **E** estágio de execução **OU** instrução IN **E** seleção P3 **E** execução **ENTÃO** ative uma mesma saída.*", vamos nomear estas duas condições separadas pelo **OU** como apenas uma - ***I3_2***.

E pra finalizar, as últimas duas condições: "***SE** for estágio de escrita **E** seleção de P0 **E** instrução OUT ative a saída*", chamaremos a condição de ***O-0***; "***SE** for estágio de escrita **E** seleção de P1 **E** instrução OUT ative a saída*", chamaremos de ***O-1***.

Perceba que nós temos 5 condições nomeadas: ***O3***, ***O2***, ***I3_2***, ***O-0*** e ***O-1***, todas elas passaram por um OR, no destaque roxo, e se for UMA delas, a gente vai ativar a "Escrita" dos registradores de I/O, ou seja, isso significa que OUT P3, OUT P2, IN P3, IN P2, OUT P0 e OUT P1, armazenam dados nos registradores de I/O, já que **OUT P3** e **OUT P2**, escreve o dado de DR para a porta mencionada (Escreve do registrador acumulador para a memória RAM), **IN P3** e **IN P2**, escreve da memória RAM para a porta selecionada (Que posteriormente vai pra DR, é o oposto de OUT).

Enquanto que **OUT P0** e **OUT P1**, escrevem de DR para P0 ou P1, ou seja, é aqui que é formado os endereços da RAM. No entanto, OUT P0 e OUT P1 não definem o bit<5> de acesso a RAM, já que elas serão "somente" para

formar o endereçamento, e não pra de fato escrever dados (Como é feito em P2 e P3). **IN P0** e **IN P1** a mesma coisa, elas não acessam a RAM, pois só leem o endereço dos registradores e escrevem em DR, para possíveis cálculos do endereço, portanto, também não definem o bit<5>. Perceba que IN P0 e IN P1 não estão dentro destas 5 condições, o que significa que elas não vão habilitar a escrita dos registradores de I/O.

Já o destaque roxo, mostra uma das condições que já mencionamos, que é a junção de IN P2 e IN P3 numa só condição (***I2_3***) e a junção das 5 condições pra habilitar a escrita dos registros, no entanto, ainda faltam mais dois OR's para representar, e eles são simples, pois já seguem as 5 condições anteriores, apenas trocando "OUT P0/OUT P1" por "IN P0/IN P1", dizendo ao banco de registros que será selecionado apenas como "Leitura" no caso do 1ª OR, pois "OUT P0/OUT P1" não leem os registros (Só escrevem).

Enquanto que o 2ª OR, representa o OUT P2 e o IN P2, ativando um AND que também depende do nível baixo de CLOCK, neste cenário, ele ativa o Bit<5> de acesso a RAM. O que nos faz concluir que, somente a porta P2 irá acessar memórias externas, sendo leitura (IN) ou escrita (OUT).

Uma outra condição que está abaixo do destaque roxo, é uma porta OR que verifica se a instrução for OUT P1, OUT P2, OUT P3 ou OUT P2 (as instruções de escrita), porque se for uma delas, um alternador de entrada irá ser ativado, levando sua entrada de 8 bits para o barramento de 8 bits que escreve no banco de registradores, mas de onde vem estes 8 bits? Simples - **DR**.

Como foi mencionado antes, instrução OUT ler DR e escreve nos registros de portas, logo a comunicação é "inversa" da instrução IN, que no caso iria ler do dispositivo externo para os registros de portas. Neste caso, nós temos um só barramento que vem de duas origens diferentes: Mecanismo Externo (RAM, ROM, Teclado, etc.) ou Mecanismo Interno (DR - Registro Acumulador).

Portanto, se a instrução OUT for ativada, ela nem vai habilitar a entrada dos pinos externos, apenas saída, isto faz com que o mesmo barramento de entrada dos registros não ocorra conflito de dados, já que os dados já estão sendo utilizados de DR.

2.5. Chaveamento de Portas Externas

Para finalizarmos, nós temos as últimas duas imagens abaixo, ambas demonstrando o funcionamento do chaveamento dos pinos externos. A próxima imagem se refere ao chaveamento dos pinos P1 e P0, nesta ordem.

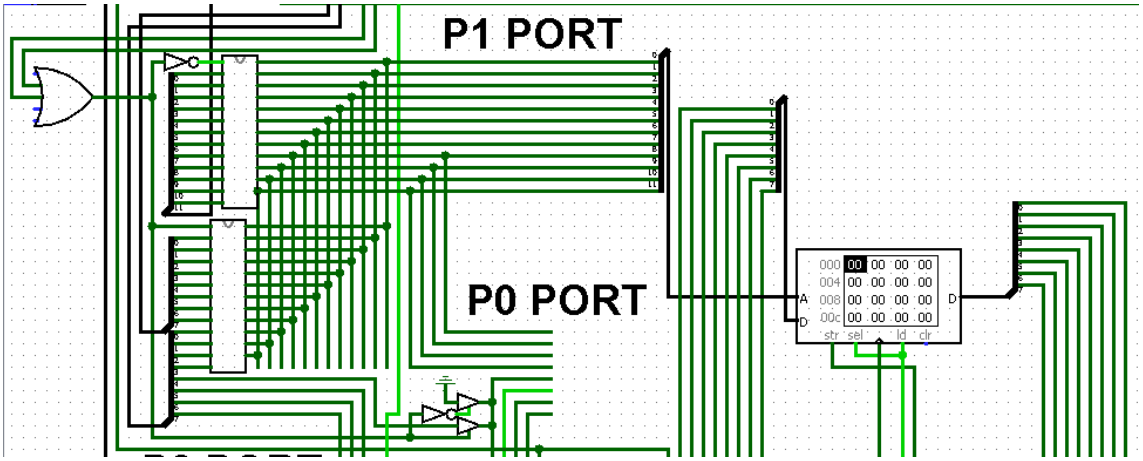


Figura 12 - Pinagem das portas P0 e P1 para endereçamento de memória RAM

Nós temos 2 alternadores de 12 bits: O de baixo é o endereçamento que vem do barramento de P0:P1 (que é sempre saída) e o de cima é o endereçamento de PC. Se lembram que eu falei do chaveamento entre P0:P1 e PC? Pois então, é aqui que ocorre este mecanismo.

Existe uma porta OR que verifica se é uma das instruções de acesso: IN ou OUT, se for, ele ativa o alternador de P0:P1 e desativa o alternador de PC, usando uma porta NOT, mas se não for nenhuma das duas instruções, se mantém ativado o alternador de PC e desativado o de P0:P1. Mas o contador de programa sempre precisa ler cada instrução da memória RAM, como que ele volta pra leitura original de PC? Fácil, a cada vez que o estágio de busca se reinicia no ciclo, automaticamente os flip-flops, que armazenaram a ativação do resultado de decodificação das instruções IN e OUT, são zerados, fazendo com que o OR ative o alternador de PC.

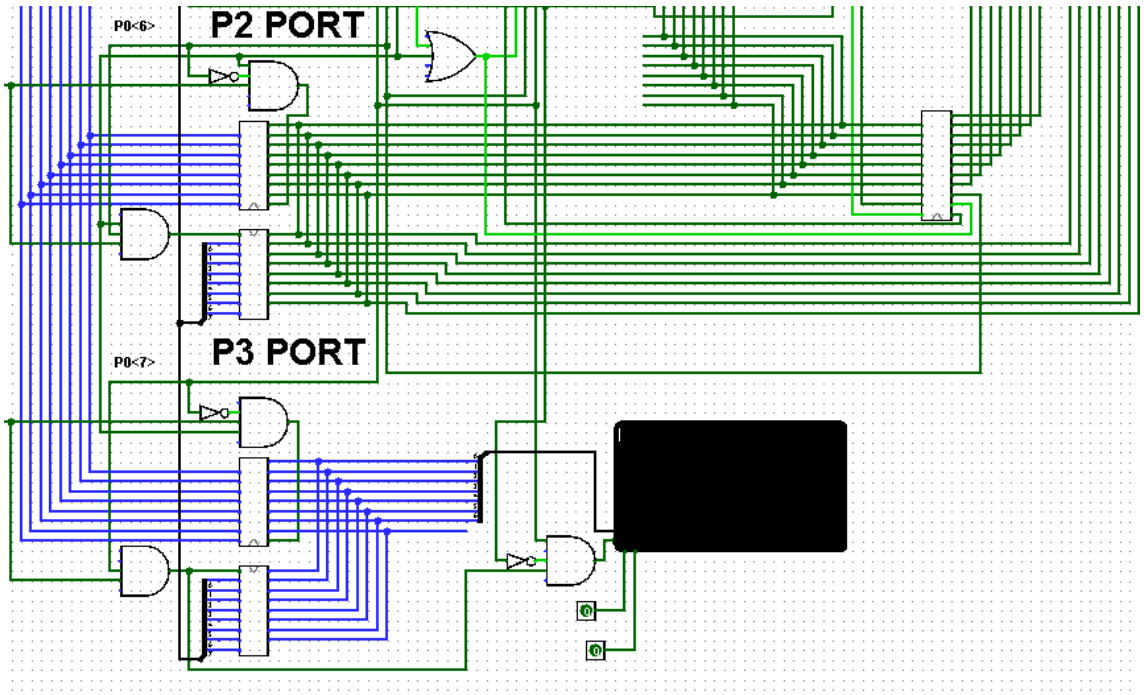


Figura 13 - Pinagem das portas de dados P2 e P3 (RAM e Monitor, respectivamente)

Mencionando a última imagem acima, nós temos o mesmo mecanismo de alternância semelhante a P0:P1 com PC, no entanto, desta vez a alternância se trata de "Direções" de dados.

Em P2 PORT, o 1ª alternador é ativado para entrada de dados (Vindo da RAM para os registros de I/O) quando o acesso da memória está definido, o bit<6> for 0 (Vem do distribuidor de 12 bits) e a seleção de registradores for do número 2 (10b - o mesmo que seleciona R0... R3, endereço que é lido do formato de instrução). No entanto, um pequeno controlador de acesso externo também efetua seu papel (O CHIP mais a direita), que terá um conjunto básico de portas AND pra verificar se o Bit<5> = 1 com o Bit<6> = 0, efetuando a seleção de memória.

O 2ª alternador é ativado para saída de dados, mantendo exatamente as mesmas condições, com a diferença que o Bit<6> deve ser igual a 1. A entrada do 1ª alternador, se conecta a saída do 2ª alternador, e assim, nós temos uma pinagem de duas direções selecionáveis.

Em P3 PORT é seguido o mesmo mecanismo de condições, mas verificando o Bit<7> ao invés do Bit<6>, e o Bit<5> não é influenciador (Acabei identificando um erro durante as explicações, porque conectei o Bit<5> no P3, mas é só pro P2). A porta P3 se conecta em um monitor TTY, que se atualiza

quando o Bit<7> de P0 é 1 e o Clock no nível baixo, diferentemente da memória RAM que faz suas atualizações com o clock no nível alto.

Os dados de 8 bits de P3 são enviados ao monitor e escritos, na finalização do estágio de escrita. Feito isso, devemos ter um pino de clock como saída alternativa do microprocessador, para sincronizar a comunicação com outros componentes, ou até mesmo, um pino de entrada externo que conecta um clock separado, permitindo ao usuário se ele quer utilizar o clock interno ou externo.

3. ARQUITETURA DE COMPUTADOR WR80

Neste artigo, vou explicar de forma completa como o processador funciona, de acordo com a arquitetura específica dele!

O WR80 é um processador digital de 8 bits, isto significa que seu formato de instrução contém 8 bits de tamanho, incluindo o barramento de dados de leitura da memória RAM. No entanto, ele utiliza endereçamento de 12 bits, podendo endereçar até 4 KB (4096 bytes) de memória externa!! Desta forma, ele possui um registrador interno para saltos condicionais e incondicionais que constrói este endereço de 12 bits em 2 ciclos de CPU, durante a leitura de estágios, portanto, vamos compreender primeiramente como funciona a divisão em estágios!

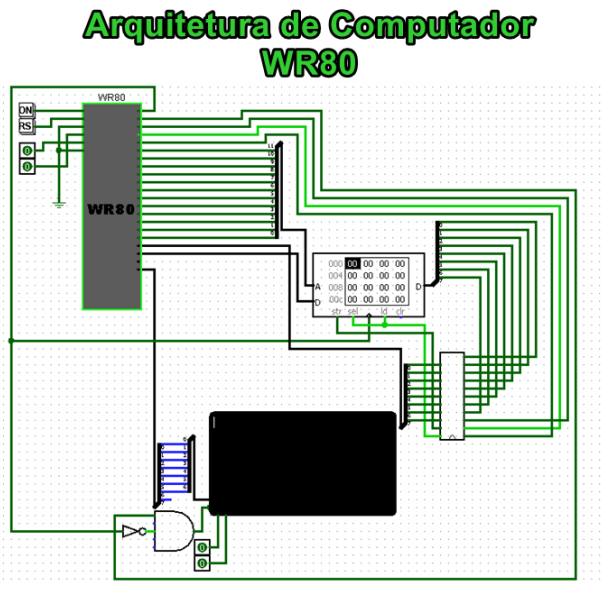


Figura 14 - Arquitetura Externa do WR80 Computer

3.1. Estágios da CPU - Buscando Instruções

Na imagem abaixo, temos um seletor usando portas lógicas, que vai ler os 2 primeiros flip-flops do contador. Este contador é formado por 14 bits: 12 bits para um contador de programa, abreviado para "PC' (Program Counter) + 2 bits para o contador de estágios.

A cada sequência binária contabilizada - 00, 01, 10 e 11 - Ele soma +1 em PC, ou seja, PC = 0, isto significa que ele endereça o 1ª byte da memória RAM, depois que contou os 4 estágios (00,01,10,11), ele soma +1 e PC = 1, onde endereça o 2ª byte da memória RAM. Cada um destes bytes é uma instrução de 8-bit. A cada 4 estágios contados, nós temos 1 ciclo de CPU!

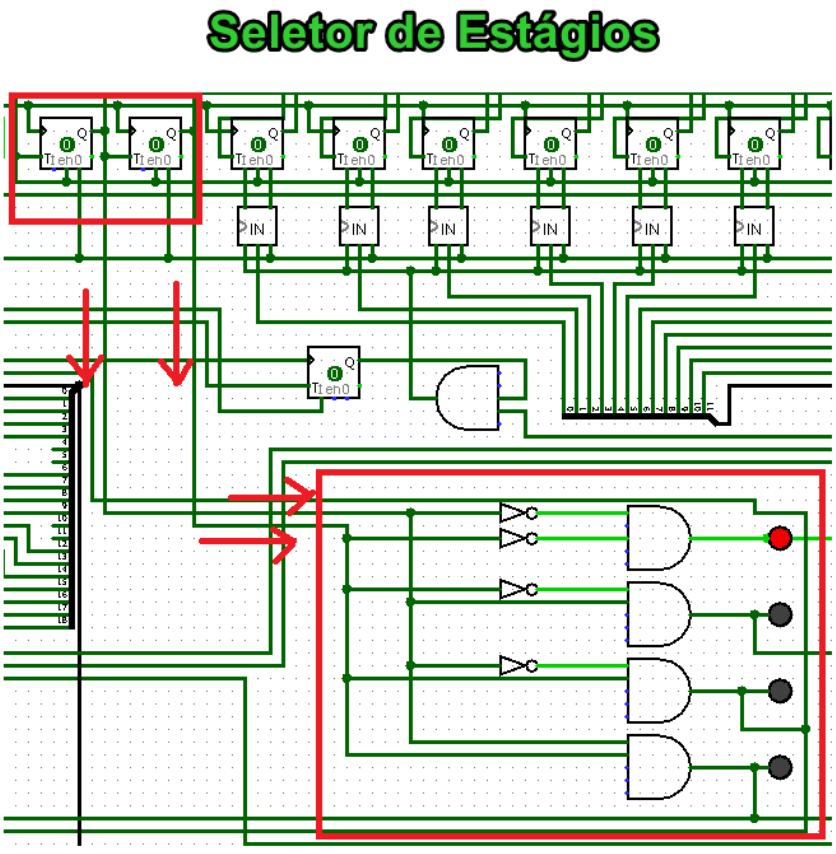


Figura 15 - Seleção de Estágios por meio de 2 bits

O contador funciona por meio de Flip-Flops tipo T, que é uma junção de entradas do Tipo J-K, quando o Flip-Flop está em sua entrada o estado 1, a cada ciclo de Clock, ele inverte o seu estado atual (Se for 0, é 1, se for 1, é 0). Se interligamos estes flip-flops em cadeia + portas lógicas AND, nós temos um contador!

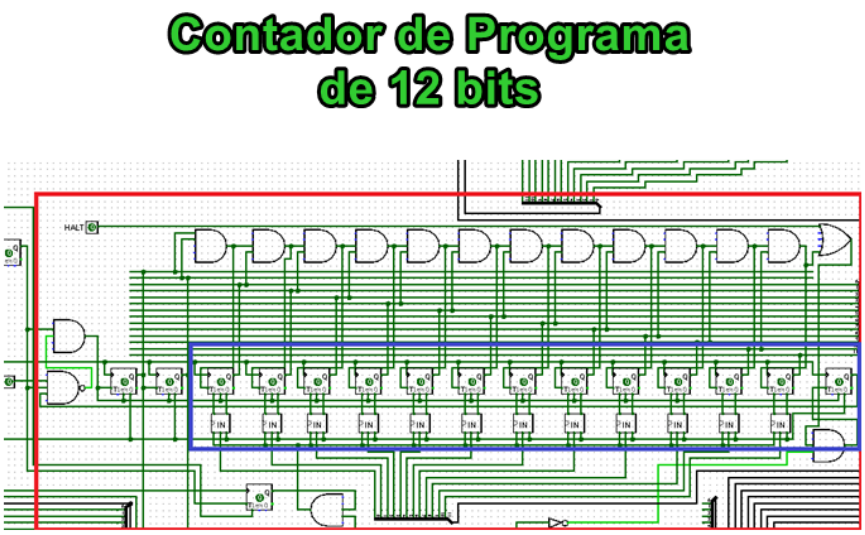


Figura 16 - Contador de Programa (PC) de 12 bits

Mas afinal, que estágios são estes? Eles são expostos e caracterizados a seguir:

1. **Estágio de Busca:** Ler a memória RAM apontado por "PC" e armazena o byte da instrução em "IR".
2. **Estágio de Decodificação:** Ler "IR" e divide o "opcode" do "endereço" e ativa uma linha específica.
3. **Estágio de Execução:** Ler a linha ativada e seleciona uma "operação" específica, habilitando dados.
4. **Estágio de Escrita:** Pega o dado de saída da operação e armazena em registradores específicos.

Cada estágio da CPU também é conhecido como "Micro operação", pois ela a partir de agora se torna a "Menor" unidade de processamento de uma instrução, portanto, uma instrução é dividida em 4 micro operações nesta arquitetura. Cada micro operação, percebemos uma tarefa que é realizada e nestas tarefas, nós temos dados a ser lidos e escritos, porém de forma sincronizada!

Por este motivo, cada estágio é selecionado pelo seletor de estágios de 4 saídas, cada saída sendo uma seleção de um estágio específico, executados em ordem. Se o seletor é alimentado pelo contador, que por sua vez, é alimentado pelo CLOCK, significa que nós temos um circuito SÍNCRONO, isto é, tudo age em relação ao tempo de CLOCK (Frequência = 4.1KHz), no momento certo!

Perceba-se que na lista de estágios, temos algumas palavrinhas aspadas, e elas são fundamentais para entendermos os processos de uma CPU. Como já explicamos do que se trata o "PC", que é o contador de programa que endereça instruções da memória, vamos saltar para o IR - Instruction Register ou Registro de Instrução.

Este registrador está na "Unidade de Controle", que é composta por: "Circuito de Controle" que recebe o sinal de clock e controla os estágios + o "Decodificador de Instruções", que compreende a instrução ativando uma linha específica. O registrador de instrução armazena o dado lido da memória no estágio de busca, deixando-o pronto para ser decodificado. O IR de 8 bits, ainda é dividido em duas partes, que detalharemos a seguir:

1. **OR (Opcode Register - 4 bits):** O opcode significa "Operation Code" ou "Código de Operação", e é nele que contém a verdadeira "identificação" de qual operação a CPU vai realizar, se é soma, saltos, subtração, movimentação, etc. Se temos 4 bits de opcode, 2^4 = 16, portanto, 16 operações possíveis!
2. **AR (Addressing Register - 4 bits):** O registro de endereço contém o endereçamento de registradores internos, portanto, podemos endereçar até 16 registradores! Dentre eles, os "registradores de usuário", que são: Os registradores de portas e os registradores de cálculos. Existem operações que usam esta parte como um "Dado literal", como no caso das operações ST, SHR e SHL.

Todas as operações diferentes das que foram mencionadas acima, utilizam o AR como endereçamento de registros, como as operações: ADD, SUB, AND, XOR, etc. O par OR:AR formam o IR, que de forma completa chamamos de "Instrução". Portanto, de forma a caracterizar uma instrução, dizemos que cada operação (16 opcodes possíveis), analisa e age por uma combinação de registros/dados lendo ou escrevendo (16 números/endereços possíveis).

Um exemplo é a operação aritmética ADD (0100 - a operação 4) que pode calcular os registradores R0, R1, R2 e R3 (Endereçados por 2 bits - 00, 01, 10, 11, respectivamente), deixando 2 bits altos reservados para futuros escalonamentos do processador (de forma mais garantida, 1 bit de extensão de opcode), ou seja, possibilitar que mais registradores sejam calculados! Desta

forma, se quisermos por exemplo somar o que está em R1 com o Acumulador, utilizamos a instrução completa: 0100 0001, ou ADD R1.

Se quisermos subtrair R2, utilizamos SUB R2 (0101 0010 - A operação 5 com o Registrador 2), e se quisermos efetuar uma operação lógica? Também é simples, temos: AND, OR, NOT e XOR; Cada uma podendo utilizar R0, R1, R2 e R3, da mesma forma que ADD e SUB. O resultado será escrito em um outro registrador, que é chamado de "Acumulador", vamos detalhar isto mais adiante em nosso E-book, quando falarmos do Estágio de Execução e Escrita.

3.7. Estágio de Decodificação – Mapeando Operações

No primeiro tópico, falamos sobre os estágios existentes e focamos no estágio de busca, já nesta segunda parte, vamos entender como a CPU decodifica as instruções no estágio de decodificação. Após o IR receber a instrução de 8 bits e o clock dar o segundo pulso, o seletor de estágios vai selecionar o próximo, que é a decodificação. O sinal do seletor vai para o decodificador de instrução a fim de habilitá-lo, que é dividido em duas partes: Decodificador de Opcodes & Decodificador de Dados.

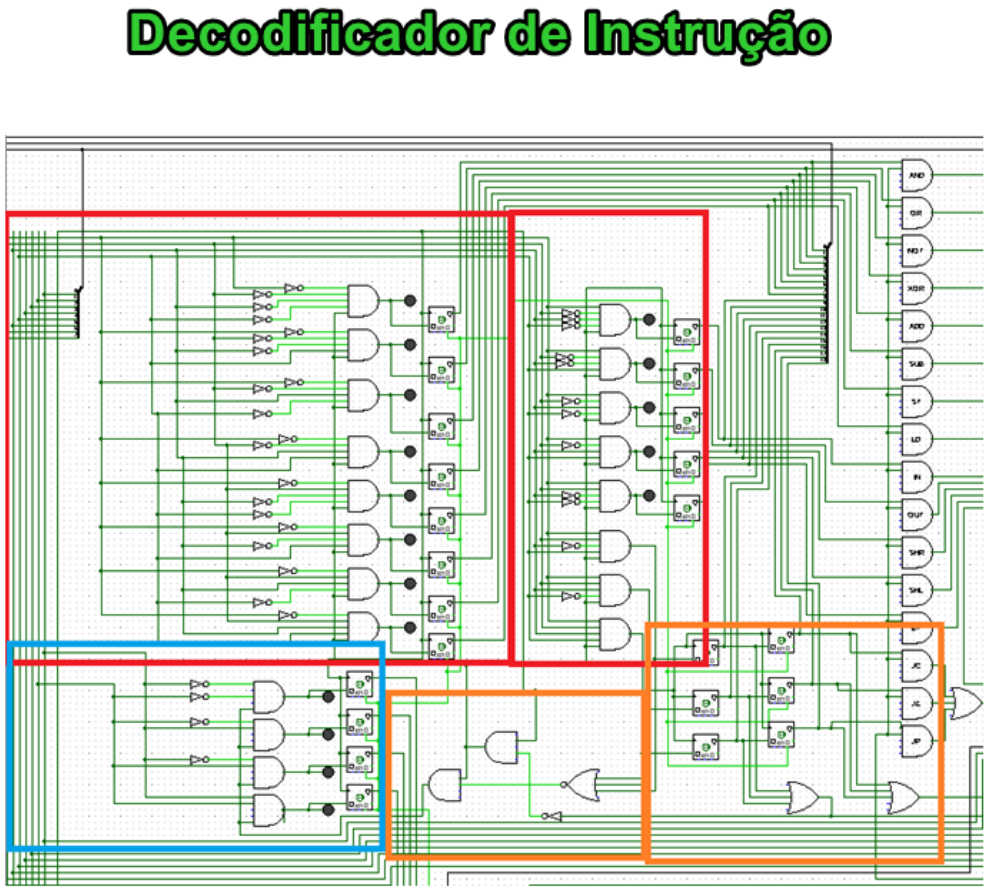


Figura 17 - Decodificador de Opcodes e Endereços

O decodificador de instrução como pode ser visto na imagem acima, realiza algumas etapas, então primeiramente, se analisarmos o destaque em vermelho, nós temos 8 portas AND do lado esquerdo e 8 portas do lado direito.

O lado esquerdo representa as primeiras 8 instruções: AND, OR, NOT, XOR, ADD, SUB, ST e LD. Já o lado direito representa as últimas 8 instruções: IN, OUT, SHR, SHL, BT, JC, JZ e JP. Isto utiliza um demultiplexador como seleção de portas AND baseado na combinação de entrada, exemplo: Se for 0000, que é a operação AND, irá selecionar o 1ª AND, se for 0010, que é a operação NOT, será selecionado o 3ª AND, e assim por diante. Assim, temos o decodificador de opcodes em ação, baseado em apenas 4 bits.

Cada uma destas linhas de ativações, armazenam em um Flip-Flop Tipo D, já que uma vez que o próximo estágio é selecionado, perdemos a habilitação do decodificador pelo seletor de estágios, desabilitando a linha que foi ativada, então precisamos armazenar este bit de ativação, que vai representar a própria operação que foi lida.

No destaque azul, temos o decodificador de dados, que também vai selecionar uma linha e armazenar em um Flip-Flop. Este seletor de dados, ou vai selecionar um dos registradores de usuário, que chamamos de GPR - General Propose Register (R0, R1, R2 e R3) ou vai selecionar um dos 4 registros de portas: P0, P1, P2 e P3. O conjunto de registros que será de fato lido ou escrito, vai depender da operação decodificada.

Os destaques em laranja, demonstra a organização de duas colunas de Flip-Flops: A 1ª coluna contém 3 Flip-Flops, armazenando os bits de ativação das instruções de salto, como: JC, JZ e JP, respectivamente. Enquanto que a 2ª coluna contém +3 Flip-Flops, armazenando a ativação que foi armazenada anteriormente na 1ª coluna (No Flip-Flop correspondente), então sabendo que estes bits são armazenados a cada ativação do Flip-Flop (CLOCK), e nesta entrada de clock, apenas o sinal de controle do estágio "Decodificação" que ativa estes flip-flops, significa que, cada coluna armazena seu respectivo bit em um ciclo separado.

Cada coluna passa por um OR, que vai verificar se é uma das 3 instruções de salto previamente armazenada, se for, ela ativa uma "parte" do registro de

Offset, possibilitando que copiamos dados: Ou dos 4 bits LSB (Menos significativos) do formato de instrução para os 4 MSB (Mais significativos) do OFFSET no 1ª ciclo, Ou, dos 8 bits completos do formato de instrução para os 8 bits LSB do OFFSET no 2ª ciclo.

Concluímos que os 4 + 8 bits do registro de Offset, formando 12 bits, é o endereço de memória para instruções de salto, que é formado em 2 ciclos de CPU distintos! É no 2ª ciclo que de fato, a instrução de salto é executada, ou seja, o valor do contador de programa PC é "somado" mais o registro de OFFSET, o resultado é armazenado no próprio contador PC. No entanto, a instrução JC só salta se CARRY estiver ativo (Bit C de STATUS), ou seja, é um salto condicional, logo se CARRY no registro STATUS não tiver ativo, PC não armazenará o resultado da soma.

Da mesma forma, a instrução JZ, que salta se ZERO estiver ativo (Bit Z de STATUS), ou seja, também é um salto condicional que só soma PC se ZERO for 1, no contrário, PC não armazena o resultado. Desta forma, teremos o funcionamento de um desvio de memória (Ou salto de memória). JP é um salto "incondicional", o que significa que independentemente de ZERO ou CARRY estiver ativo ou não, ele soma PC de qualquer forma.

Por que devemos aliás dividir a instrução de salto em 2 ciclos de CPU? Justamente porque o formato de instrução é de 8 bits, e o estágio de busca só ler 8 bits de cada vez devido ao tamanho do barramento (8 trilhas), por este motivo, devemos dividir a instrução de salto em 2 partes:

1. Opcode de 4 bits + 4 bits MSB do OFFSET no LSB da instrução = 8 bits;
2. 8 bits LSB restantes do OFFSET no formato de instrução inteira = 8 bits;

Portanto, instruções de salto terá no total 16 bits no formato de instrução. Já o próximo destaque laranja, a esquerda, representa as "condições" de habilitação do decodificador, ou seja, se for uma das instruções de salto no 2ª ciclo, NÃO ativa o decodificador, já que a CPU entenderia que a 2ª parte da instrução de salto, iria se referir a uma nova instrução, e é sobre isto que devemos evitar, já que a 2ª parte só é a parte do endereço de salto, e não uma nova instrução.

As 16 portas AND que estão mais a direita, irão ler o conteúdo armazenado nos flip-flops (bits de ativação) referente a instrução + o sinal de controle do estágio de "Execução", isto significa que, quando for execução, a instrução que foi selecionada no estágio anterior, irá de fato selecionar um circuito combinacional da operação lógico-aritmética, enviando dados para habilitadores de entrada, de acordo com cada instrução.

Os habilitadores de entrada, por sua vez, irão enviar dados de origens diferentes (ou semelhantes) para um circuito combinacional específico, que vai de fato processar dois dados enviados para os habilitadores. No próximo tópico, veremos o funcionamento deste processamento e entenderemos como são determinados os operandos de cada instrução através dos habilitadores de entrada.

3.3. Estágio de Execução – Efetuando Operações

No tópico anterior, demonstramos como funciona o decodificador de instruções no WR80, e finalizamos isto especificando os habilitadores de entrada, que basicamente é um conjunto de portas AND encapsulados, que vão determinar os operandos, então nesta parte, vamos verificar estes operandos e os circuitos combinacionais por onde eles passam.

Como podem ver na imagem, temos os dados lidos que serão processados. Estes habilitadores respondem a seguinte pergunta: Qual é a fonte dos dados que serão processados para cada instrução? Para entender isto, devemos verificar que, esta arquitetura de microprocessador, trabalha com instruções monádicas, que basicamente são instruções de apenas 1 operando, já que o nosso modelo escolhido é o SAP (Simple As Possible), isto é, uma forma muito minimalista de se construir um processador.

Quase todos os habilitadores, atuam em cima de um registrador chamado DR (Data Register) que é o acumulador desta CPU. Portanto, DR será um operando "Implícito" de cada instrução, onde ela não será "exposta" na programação Assembly, ao invés disso, sabemos que a própria instrução armazenará o resultado nela, assim sendo, as instruções leem de DR mas na programação Assembly, a gente especifica qual é a "fonte" de dados que DR irá atuar junto.

3.3.1. Habilitadores de Dados & Set de Instruções

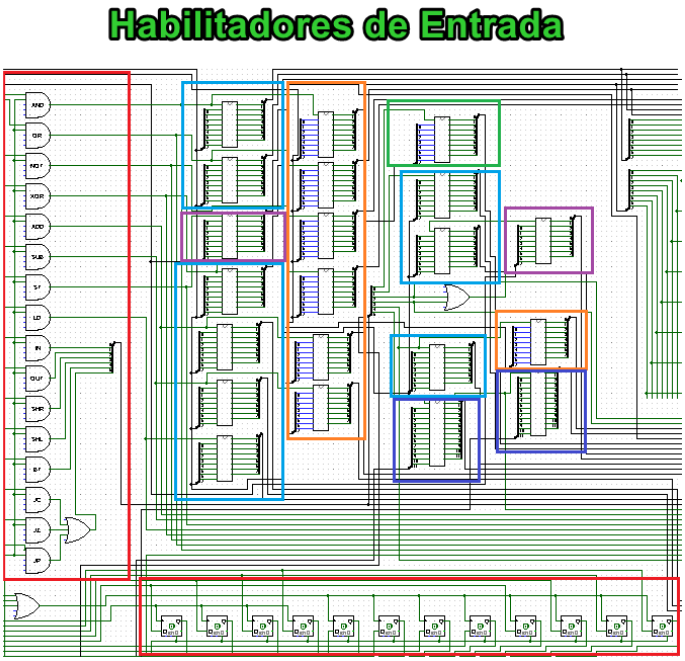


Figura 18 - Habilitadores de Dados nas Entradas para a ULA

Um outro motivo para usarmos habilitadores destas "fontes", é que nós precisamos "bloquear" os dados que passam pelos circuitos combinacionais, quando este circuito não está sendo utilizado, ou seja, se você quer "somar" um dado, você não precisa habilitar os dados da instrução de movimentação, por exemplo, apenas o de soma. Antes de analisarmos a imagem acima, vamos entender o set de instruções do WR80:

- **AND** - Operação Lógica AND que realiza um AND de DR com Rx (R0, R1, R2 ou R3) e armazena o resultado em DR.
- **OR** - Operação Lógica OR que realiza um OR de DR com Rx, armazenando o resultado em DR.
- **NOT** - Operação Lógica NOT realizando uma INVERSÃO do que está em Rx, armazenando em DR (Sem alterar Rx).
- **XOR** - Operação Lógica XOR que realiza um XOR de DR com Rx, armazenando em DR.
- **ADD** - Operação Aritmética ADD que SOMA o valor de DR com Rx, armazenando em DR.
- **SUB** - Operação Aritmética SUB que SUBTRAI o valor de DR com Rx, armazenando em DR.
- **ST** - Operação de movimentação STORE que COPIA/MOVE um número literal de 4 bits para DR.

- **LD** - Operação de movimentação LOAD que COPIA/MOVE um dado de DR para Rx.
- **IN** - Operação de movimentação INPUT que COPIA/MOVE um dado de Px (P0, P1, P2 ou P3) para DR, onde Px irá "Ler" de um dispositivo de entrada (RAM, teclado, etc.).
- **OUT** - Operação de movimentação OUTPUT que COPIA/MOVE um dado de DR para Px, onde Px irá "Escrever" em um dispositivo de saída (RAM, Monitor, etc.).
- **SHR** - Operação de deslocamento de bits SHIFT RIGHT que DESLOCA N bits de DR para a direita, onde N é um número literal de 3 bits (até 7 deslocamentos).
- **SHL** - Operação de deslocamento de bits SHIFT LEFT que DESLOCA N bits de DR para a esquerda, onde N é um número literal de 3 bits.
- **BT** - Operação de comparação BIT TEST que realiza uma subtração de DR com Rx, sem armazenar o resultado em DR, mas armazenando em SR (STATUS Register) o bit Z - Zero e o bit C - Carry.
- **JC** - Operação de salto condicional JUMP CARRY que salta para endereço especificado de memória (Como número literal) se o bit C = 1.
- **JZ** - Operação de salto condicional JUMP ZERO que salta para endereço especificado se o bit Z = 1.
- **JP** - Operação de salto incondicional JUMP que salta para endereço de memória especificado.

Compreendendo o set de instruções, então analisaremos a imagem. O primeiro destaque em vermelho, são as portas AND mencionadas no tópico anterior, que vai ativar um destes habilitadores. O segundo destaque em vermelho, é o próprio registro de offset de 12 bits para saltos (explicado no tópico anterior). O que está em destaque azul claro, são as entradas habilitadas vindas de DR, como as instruções lidas em ordem dos habilitadores na imagem: AND, OR, XOR, ADD, SUB, LD, OUT, SHR/SHL e BT.

Já o destaque em Laranja, são as entradas vindas de Rx, como das instruções anteriores, exceto das instruções OUT e SHR/SHL. O destaque em violeta, vem dos números literais (4 bits no formato de instrução), como: ST e SHR/SHL. O destaque em verde, é o único habilitador que ler da fonte Px, da instrução IN, já que ela precisa "escrever" em DR. Por último, o destaque em

roxo, são os dados de PC e OFFSET, nas instruções: JC, JZ e JP, usando apenas 2 habilitadores.

Perceba que nós temos um OR entre dois destaques azul claro no lado direito, este OR ativa dois habilitadores, que se referem a instrução SHR e SHL, mas os dois habilitadores são para as mesmas instruções, já que os dados de operando delas são iguais (2 habilitadores para SHR e os mesmos 2 habilitadores para SHL), não só por este motivo, mas também pelo fato das duas instruções utilizar um mesmo circuito combinacional de deslocamento de bits.

Da mesma forma que o ADD e o SUB que utilizam um mesmo Full-Adder. Portanto, é possível fazer a mesma coisa pro ADD e SUB, usar um só habilitador. Em otimizações futuras deste circuito isto será feito. Okay, já sabemos de onde vem estes dados e quais são eles, mas... Para onde eles vão? Eles vão para circuitos combinacionais específicos que veremos a partir do próximo tópico.

3.3.2. Operações Lógicas

As operações lógicas desempenham um papel crucial na construção de processadores, veremos abaixo como são estas operações.

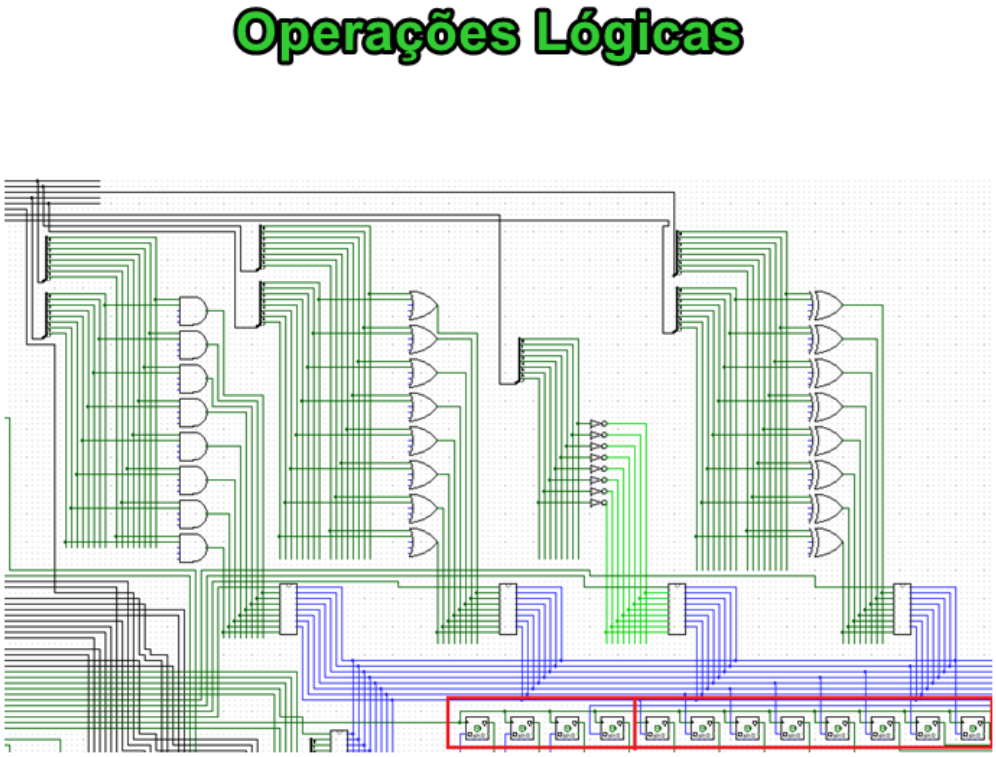


Figura 19 - Quatro Operações Lógicas da ULA e o Registro de Resultado

Na imagem acima, vemos as 4 operações lógicas separadas em colunas: AND, OR, NOT e XOR. Cada uma delas, operam DR com Rx e enviam o resultado para um alternador, dispositivo este formado por circuitos tri-state (buffer controlados = portas AND com um sinal de habilitação) que se conecta em um mesmo barramento de saída, possibilitando que mais operações utilizem o mesmo barramento.

O dado de saída, será o resultado da operação lógica bit-a-bit que foi selecionada e será armazenada no "Registro de Resultado da ULA". Como podem ver, este destaque em vermelho, contém 12 bits (devido as instruções de salto), no entanto, separei o agrupamento de 8 bits, para dizer que as instruções lógicas só armazenam nestes 8 bits. Deste registro resultado, será copiado para DR no estágio de escrita.

3.3.3. Operações de Movimentação

Na imagem abaixo, temos os 4 alternadores, referente às instruções: ST, LD, IN e OUT, respectivamente. Dos habilitadores destas instruções, eles vão diretamente para os alternadores, sem nenhuma operação especial, apenas, "Copiar". Isto no caso de ST e LD, mas para IN e OUT, temos condições árduas no outro lado da CPU, próximo aos registros de I/O, pois elas são mais complexas devido a forma especial que é tratado as portas (Visto no capítulo 2 - **Circuitos de Entrada e Saída**).

O dado copiado de ST é um número literal, o de LD é de DR, do IN vem de Px que foi lido de um dispositivo externo e o OUT vem de DR para escrever em algum dispositivo, no entanto, todos eles tem um mesmo direcionamento neste estágio - O barramento do registro de resultado. Pois é deste registro, que os destinos serão determinados em outras condições.

Os alternadores de movimento permitem que várias conexões sejam conectadas em um mesmo barramento, sem que interfira em outros dados, pois quando cada dado for selecionado pelo alternador, os outros automaticamente serão bloqueados. O alternador opera com um buffer controlado que impede a entrada de ir para a saída quando o estado é 0.

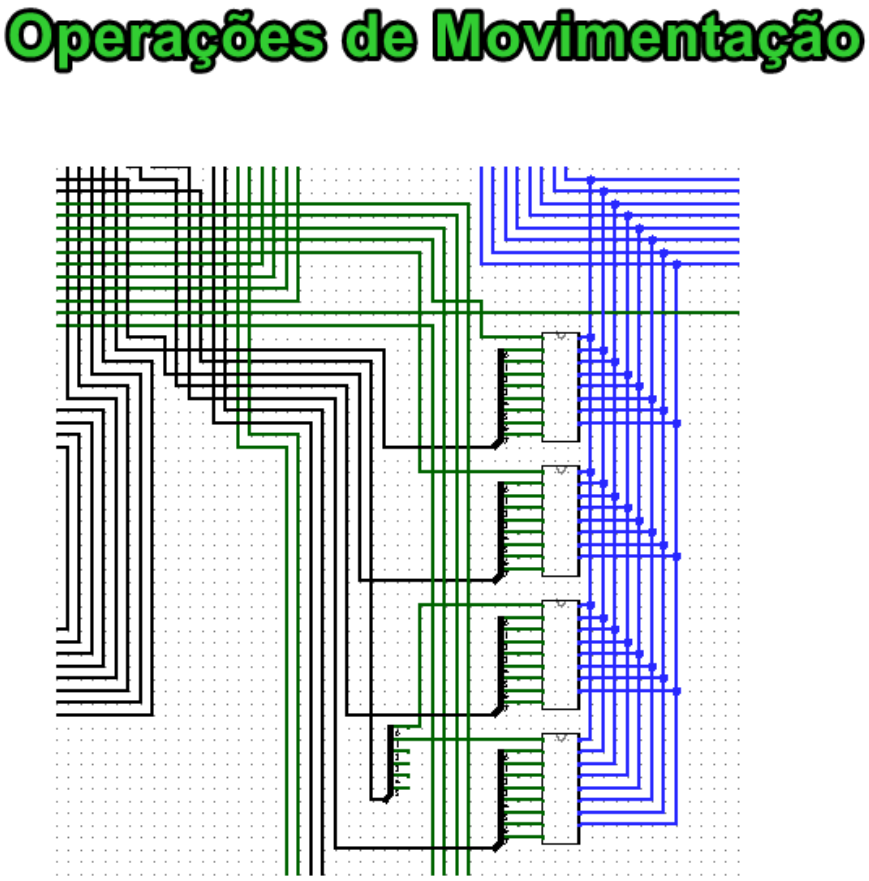


Figura 20 - Quatro Operações de Movimentação usando Alternadores

3.3.4. Operações de Deslocamento

Na última e próxima imagem, temos as operações de deslocamento, que utilizam um conjunto de X multiplexadores interligados entre si Y vezes, onde X é o tamanho do dado (8 bits) e Y é a quantidade de bits referente a quantidade de deslocamentos (3 bits para 8 - 1 deslocamentos). O dado resultante, vai para o alternador, que por sua vez, escreve no barramento de registro de resultado.

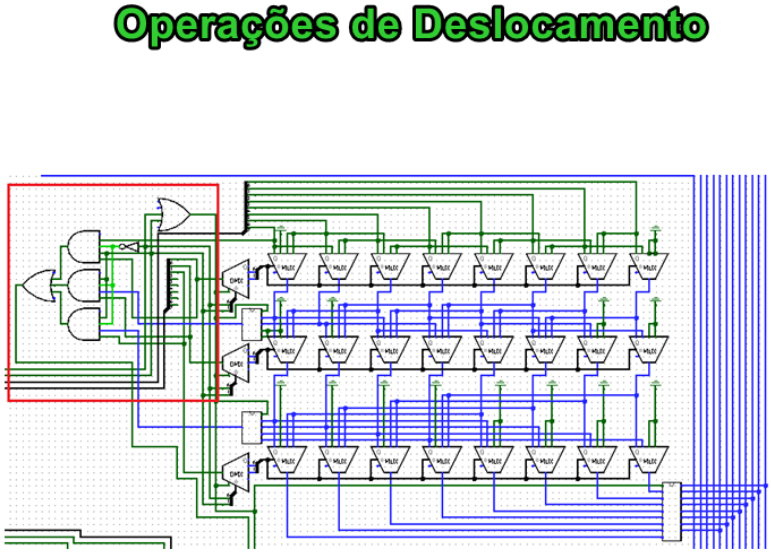


Figura 21 - Operação de Deslocamento de Bits usando MUX

O destaque em vermelho representa a adaptação do deslocador (Visto no capítulo 1 – **Circuitos Para Deslocamento de Bits**) para compreender em qual cenário o deslocador "estoura", ou seja, ele cria o Carry Out para ser armazenado em STATUS (SR). Ele apena verifica a quantidade de bits que será deslocada e se os últimos bits daquele conjunto está setado. Isto significa que ele "Prevê" o estouro, já que estas condições são realizadas na entrada, e não na saída.

No próximo tópico, finalizaremos com o estágio de execução especificando o Full-Adder (Somador-Completo), para assim, iniciarmos sobre o estágio de escrita.

3.3.5. Operação de Soma e Suas Aplicabilidades Matemáticas

Até aqui, já vimos sobre operações lógicas, de movimentação e deslocamento da ULA no estágio de execução, além dos habilitadores de entrada de dados para os circuitos combinacionais da ULA, que representa a "fonte" dos dados para processamento. Neste tópico, será visto sobre o Full-Adder (Somador-Completo) e suas aplicações em diversas operações!

Para entendermos o conceito da utilização do Full-Adder em outras operações, precisamos saber como é possível utilizar a soma para determinar outras funções matemáticas. Eis uma pequena introdução: Se nós queremos somar, basta usar o sinal de '+', simples, não? 1 + 1 = 2. Okay, isso todo mundo já sabe, e na verdade, por trás da soma, há todo um mistério matemático envolvido, mas não vamos focar nisso agora.

Mas, e se quisermos subtrair? Simples também, basta somar com um número negativo! 1 + (-1) = 1 - 1 = 0... 2 + (-3) = 2 - 3 = -1. Isto no mundo digital também é válido, já que se você pega o valor 1 em binário (00000001) e soma mais um número -1 em binário (11111111), é o mesmo que "estourar" a capacidade de 8 bits, ou seja, 11111111 + 1 = 00000000. Isto acontece porque, se a capacidade estoura, ele zera todo o registro ou determina uma "sobra" (A sobra seria os dados restantes do estouro).

A multiplicação é a soma sucessiva de números por ele mesmo, ou seja, 4 x 3 = 4 + 4 + 4 (Somando 4 três vezes). E a divisão? Ela é a subtração de X por Y, N vezes, até o resultado ser igual ou menor que 0. N será o resultado da

divisão, e sabemos que subtração pode nascer da soma. Portanto, 16 / 2 = 16 + (-2) + (-2) + (-2) + (-2) + (-2) + (-2) + (-2) + (-2) = 0 -> 8 subtrações.

Exponenciação é a multiplicação de um número sucessiva vezes por ele mesmo, e multiplicação utiliza soma. Portanto, a partir da exponenciação, você pode criar logaritmos e raízes, já que são as operações inversas. E através de todas estas operações, você pode criar funções trigonométricas e mais outras equações complexas, ou seja, qual foi a origem? Soma. Toda esta explicação é só para nos ambientar de que, através de um simples somador-completo, como abordaremos no próximo tópico, conseguimos implementar diversas outras operações da ULA, como: Instruções de Salto.

3.3.6. Operações Aritméticas – Full-Adder

A imagem abaixo representa o mecanismo de circuitos aritméticos para soma & subtração, envolvendo demais outras instruções.

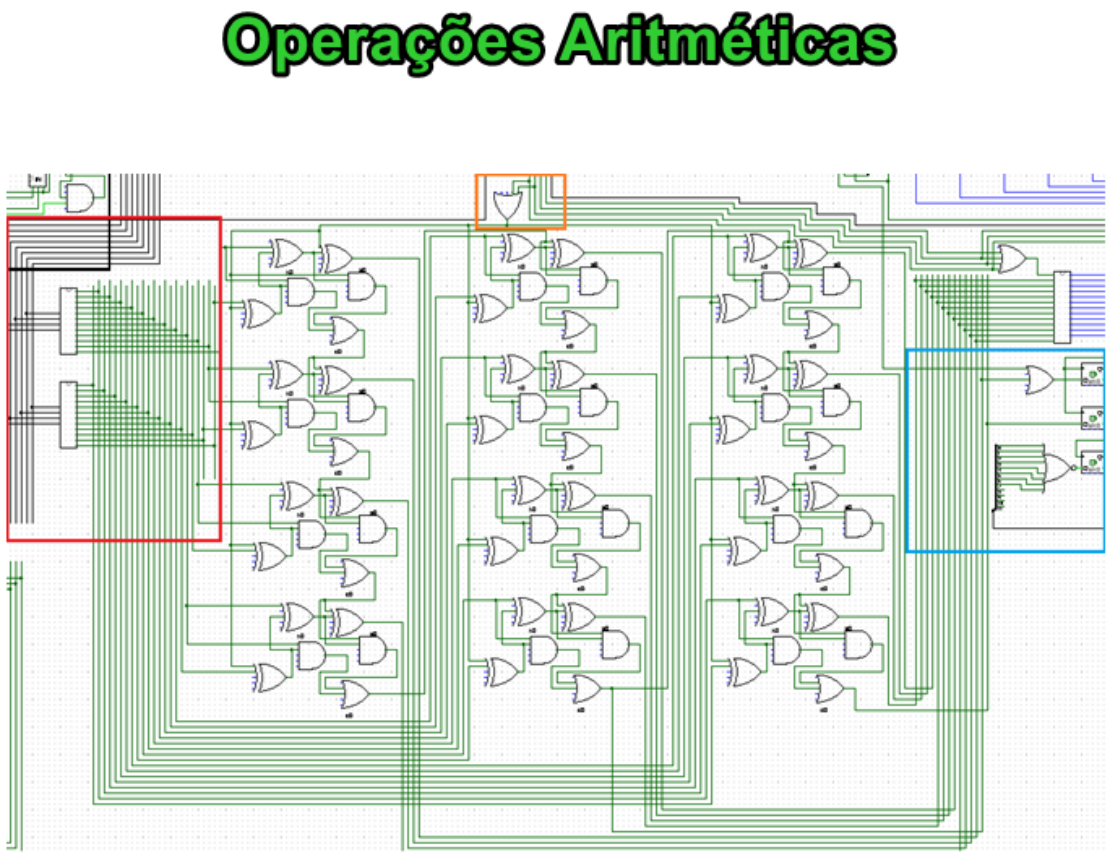


Figura 22 - Operações Aritméticas de Soma, Subtração e Saltos usando Full-Adder

Saltar para uma região de memória RAM, é o simples ato de "somar" o contador PC (que endereça a instrução atual) + o número de bytes a serem

saltados (OFFSET - Deslocamento) e atualizar PC com este resultado. E se este número de deslocamento for negativo? A gente já sabe o que acontece, como explicado anteriormente, ele vai "subtrair" PC durante a soma. Portanto, se o deslocamento for positivo, ele salta para frente da instrução atual, se for negativo, ele salta para trás da instrução atual.

Desta forma, conseguimos percorrer toda a memória RAM, como se fosse um plano cartesiano, com seu lado negativo e positivo, a partir do ponto 0 que é - O endereço atual da instrução de salto.

Já na operação de subtração, usamos um método diferente, sem precisar "converter explicitamente" o outro número em negativo, mas ao invés isso, usamos uma porta XOR no Full-Adder conectada a saída do Carry IN e aos bits do dado B, e assim, ele naturalmente converte o segundo número em negativo, de forma "implícita", dando um resultado subtraído. O Carry IN será ativo na instrução "SUB", isto significa que nós não usamos uma instrução ADC (ADD com Carry) como nas arquiteturas mais modernas.

E como realizamos uma comparação de dados? Então, existe duas formas de se fazer isto:

1. Se você quer verificar se um valor é igual/diferente ao outro, basta utilizar XNOR (XOR negado), porque ele zera a saída se os valores são diferentes, no contrário (se não zerar), são iguais, ou usar XOR porque ele zera se os valores são iguais, no contrário, são diferentes.
2. Utilizar a subtração de valores, que inclusive este método é utilizado nos microcontroladores PIC direto no código Assembly. A subtração é ainda melhor, porque além de analisar operações de igualdade (igual e diferente), ele analisa operações relacionais: Menor que, Maior que, etc. Portanto, se você subtrai 2 - 4 = -2, o resultado deu negativo? Sim, então 2 é menor que 4. Subtraindo 3 - 3 = 0, o resultado deu zero? Sim, então são iguais. Não, então são diferentes. 8 - 2 = 6, se o resultado é positivo, portanto, 8 é maior que 2! (Viu? isto se trata de uma inferência lógica).

Como saber se o valor é "maior ou igual"? Responderemos a esta pergunta abaixo com mais inferências.

Se a instrução BT realiza a subtração, ela não pode, de forma alguma, armazenar o resultado em DR, mas ela precisa "informar" ao processador o STATUS da operação. Sabemos que o Full-Adder também entrega o status destes bits, chamados de "Carry Out", e posteriormente, eles são armazenados em SR - STATUS register. Se a soma por um número ultrapassar o tamanho máximo de um registrador, Carry Out = 1, se não, Carry Out = 0. Se a subtração der negativa, Carry Out = 0, se não, sendo positiva, Carry Out = 1 (É o oposto da soma).

Se usarmos este carry out para escrever no bit Carry (bit<0>) de STATUS, conseguimos saber apenas lendo este bit, se a subtração deu negativa ou positiva. Mas para identificar se é "maior ou igual", nós temos duas observações: Ou Carry deve ser 1 ou Zero deve ser 1. Se o resultado deu negativo, de forma alguma Zero será 1, portanto, se ZERO = 0, não será "Maior ou Igual", será menor, já que o resultado NÃO deu zero (Zero = 0).

E para identificar se é "Menor ou Igual", Ou Carry deve ser 0 ou Zero deve ser 1. A missão da instrução BT é apenas definir o estado destes 2 bits, mas quem de fato vai "verificar" eles, são as instruções JC e JZ.

Visto isso, a união de BT com JC/JZ na codificação em Assembly, é formado uma "condicional", como vocês já fazem no IF, ELSE, WHILE em programação alto-nível, podendo simular todas as operações relacionais em programação.

Na imagem acima, podemos ver no destaque em azul, a escrita de 3 Flip-Flops, onde o 1ª será Carry, o 3ª o Zero e o 2ª o "Overflow" de salto (Na verdade, este overflow de saltos ainda não é muito bem trabalhado no circuito, então eu deixo ele reservado para futuras aplicações). Destes 3 Flip-Flops, eles serão levados ao SR, que é o registro de STATUS, visto no destaque em azul da próxima imagem, Registro este armazenado no estágio de escrita.

Já o destaque em vermelho na primeira imagem, representa um circuito encapsulado de portas OR, que vai alternar entre os dados possíveis a serem lidos pelo Full-Adder, ou seja, se for saltos, os dados são PC e Offset, se for soma, será DR e Rx, se for subtração, o mesmo que soma, e se for BT, o mesmo

que soma. A porta OR no destaque em laranja, representa a ativação do subtrator (Ativar Carry IN).

O Full-Adder de 1 bit, é composto por 2 portas XOR, 2 portas AND e 1 porta OR (+1 porta XOR adicional para subtração), onde a 2ª porta XOR determina o resultado do cálculo e a porta OR determina o bit Carry Out, mas e se quisermos interligar vários bits? Basta pegar este Full-Adder de 1 bit e conectar o seu Carry Out no Carry IN do próximo Full-Adder e assim por diante.

Então é isto que fazemos para este Full-Adder de 12 bits (8 bits são somados para instruções comuns e 12 bits para saltos). O 1ª somador é o bit 0, o 2ª o bit 1, 3ª o 2, ... até o bit 11, então o resultado vem de forma inversa, portanto, você deve reorganizar este resultado no barramento de saída do alternador, escrevendo no registro de resultado.

3.4. Estágio de Escrita – Armazenando Resultados

Na próxima imagem abaixo, temos os registradores de fontes e destinos das operações: "quem" pode "armazenar" o resultado da ULA ou "quem" pode "levar" para os habilitadores de entradas, isto é, DR destacado em vermelho, SR destacado em azul claro e Rx destacado em roxo.

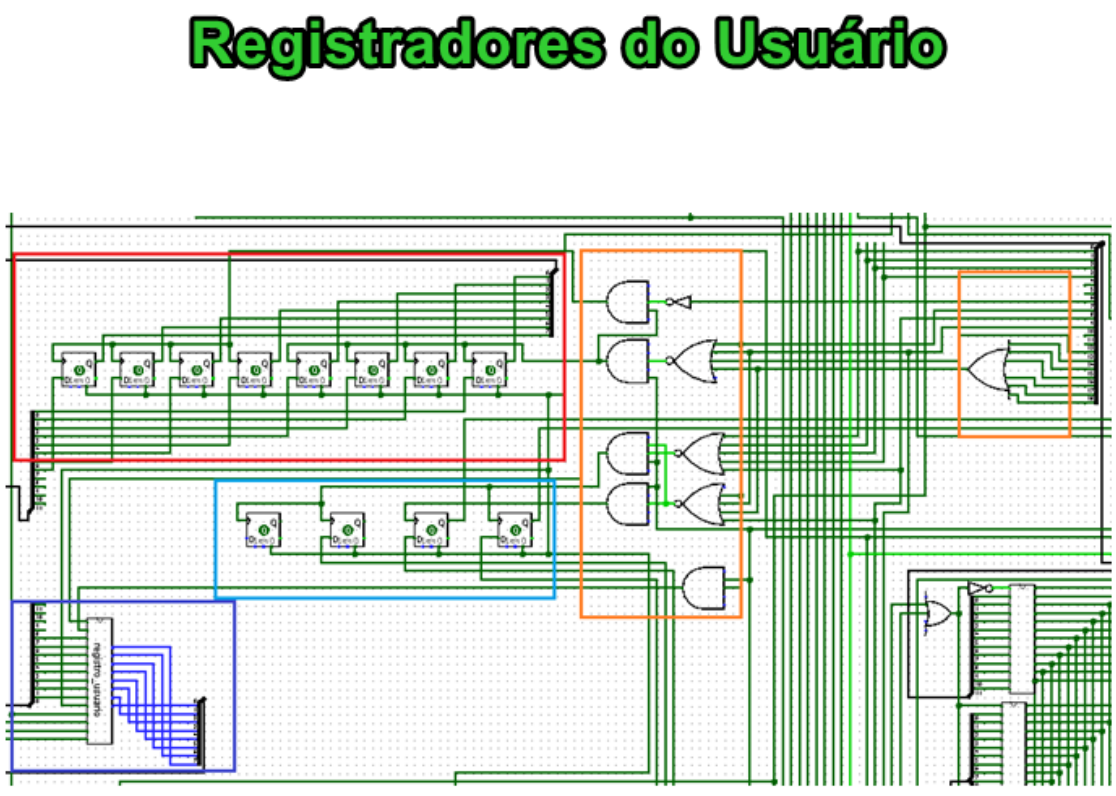


Figura 23 - Registradores DR, SR e R0-R3 (Rx)

O acumulador DR, de 8 bits, recebe o valor que está no registrador de resultado da ULA, a depender das instruções que já sabemos quem são. Rx também vem deste mesmo registrador, no entanto, SR vem dos 3 Flip-Flops que mencionamos do Full-Adder, para armazenamento do bit 'Carry' e do bit 'Zero'.

No destaque em laranja, temos as condições de escrita no estágio de escrita para cada registrador. Estas condições determinam se aquele registrador vai receber o resultado baseado na instrução selecionada pelo decodificador. Por exemplo, ST só armazena nos 4 bits LSB de DR, portanto, a condição sabendo que é ST, ela não ativa os 4 bits MSB (Só os LSBs).

Na imagem abaixo, temos as condições de saltos, destacado em laranja, que verificam o bit C e Z de SR e verifica se é uma das instruções de saltos, se for validado, ele escreve do registro de resultados em PC. O registrador PC naturalmente soma +1, devido a uma sincronização de um Flip-Flop com 1 CLOCK, é uma forma de "regularizar" os deslocamentos de saltos.

Condições de Saltos

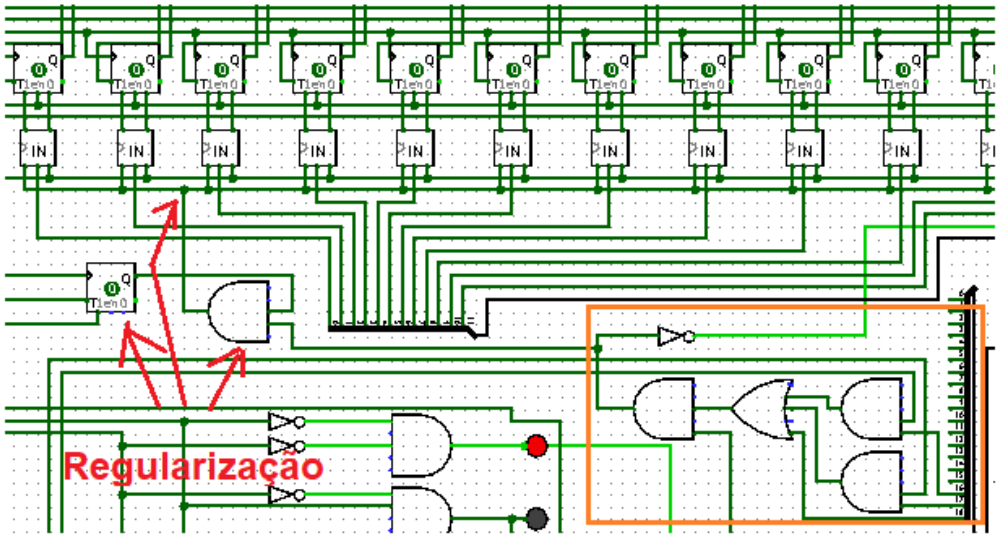


Figura 24 - Condições de Saltos para Escrita em PC

Na penúltima imagem deste tópico, temos os registradores de portas + suas condições (Veja o capítulo 2 – **Circuitos de Entrada e Saída**), isto é, P0 até P3, onde P0:P1 endereça a memória RAM/ROM, 4 bits MSB de P0 define o

controle de acesso a RAM e P2 será a porta de dados para leitura ou escrita na memória RAM. Enquanto que P3, será para outros dispositivos.

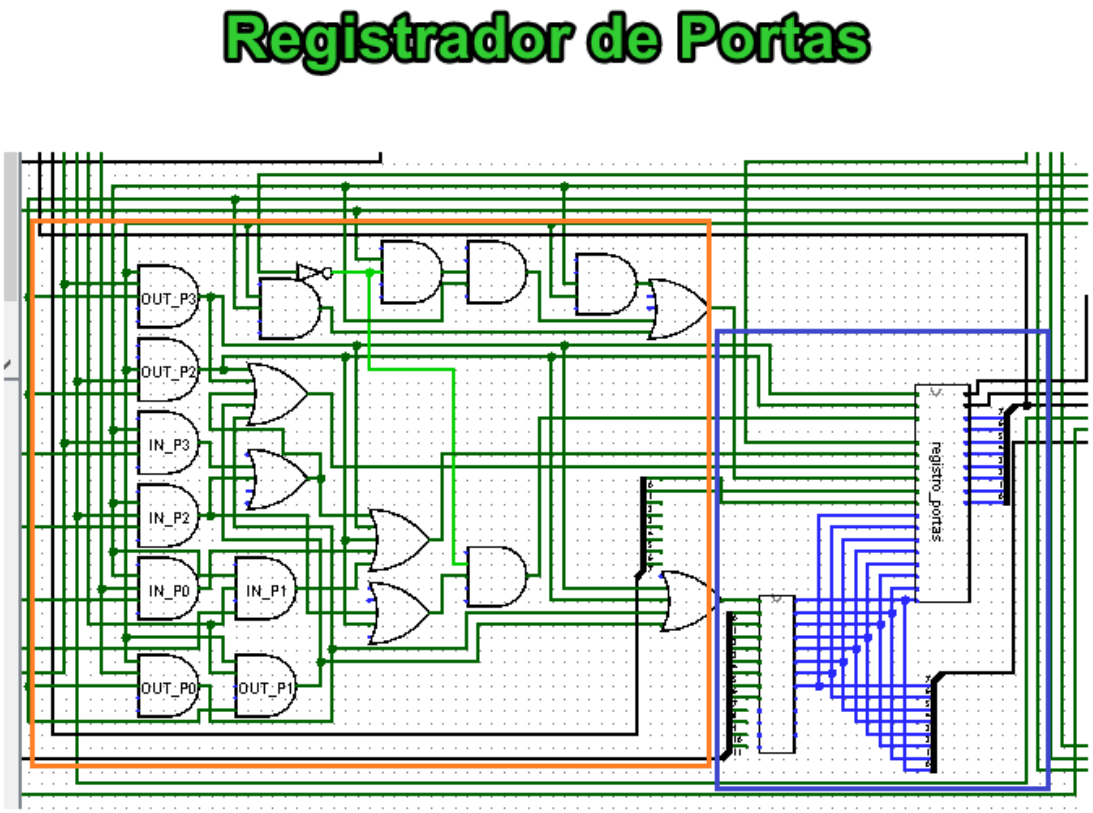


Figura 25 - Registrador de Portas de I/O com suas condições

Na última imagem deste tópico, temos a pinagem externa destas portas, que foram adaptadas do segundo capítulo mencionado para determinar o encapsulamento do WR80.

As pinagens também recebem sinais de controle dos 4 bits MSB de P0, para chavear os seus direcionamentos e determinar se o que será feito é leitura de RAM, escrita de RAM, leitura de outro dispositivo ou escrita em outro dispositivo, ou até mesmo, ao invés de ser uma RAM, poder ser uma "ROM". Já sobre P0 e P1, constantemente precisam se chavear com PC.

O chaveamento é feito pois ora o acesso a memória RAM será por instruções de I/O pelo programador Assembly, ora o acesso a memória será feito pelo contador de programa, no entanto, o contador realizará este acesso a cada estágio de busca, portanto, é necessário um mecanismo no qual o chaveamento de PC seja padrão e sempre retornável.

Pinagem externa das Portas

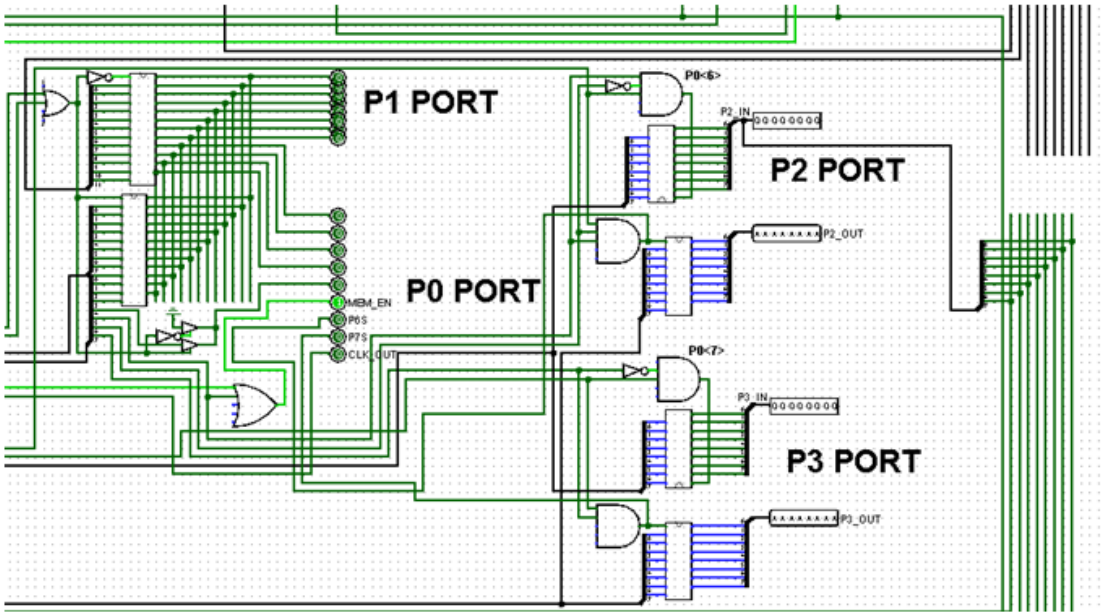


Figura 26 - Pinagem das portas de I/O

Para finalizar este capítulo, falaremos no próximo tópico sobre o circuito como um todo, analisar a questão de sincronização e velocidade de clock, como também falar de possíveis investimentos e implementações futuras.

3.5. Arquitetura de Comunicação Interna

Se você chegou até aqui, parabéns, você é um verdadeiro guerreiro!! Muito obrigado por literalmente acompanhar este conteúdo da forma mais genuína e estudiosa! O que eu vou mostrar aqui é uma análise geral de toda a arquitetura interna, ou seja, a comunicação das unidades. Além de identificar possíveis melhorias e implementações. Vamos lá!

A arquitetura interna é toda a organização de circuitos para o funcionamento do processador. Ela é composta por um rearranjo combinatório de portas lógicas e outros dispositivos formando componentes e encapsulando em uma ou mais camadas. Cada componente desempenha uma tarefa importante para o processamento de um dado e entrega de resultados para outros componentes. Quando criamos um sistema de componentes interligados a fim de atingir um resultado, criamos uma arquitetura.

Microoperações do WR80

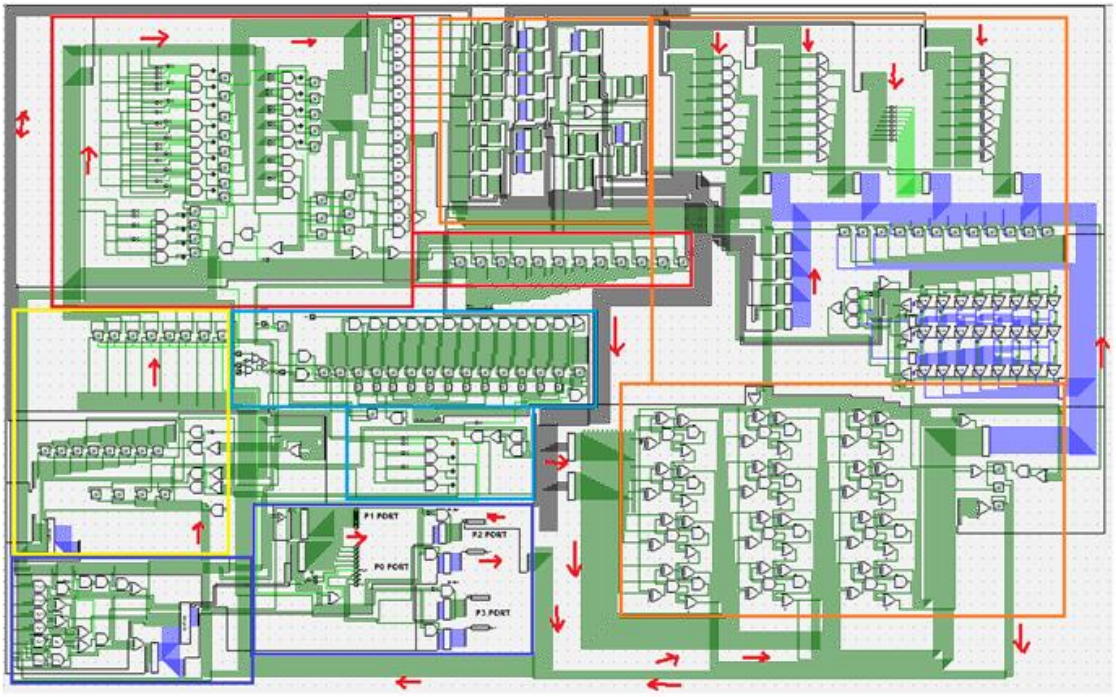


Figura 27 - Micro operações de todo o WR80 e suas divisões

Na figura 27, mostramos as divisões das "Micro operações do WR80", de forma que elas estão soltas, sem encapsulamento. O destaque em azul claro é o "circuito de controle" que contém um clock de 4.1KHz máximos do LogiSIM + suas condições de entrada para verificar no chip externo do WR80 se o usuário quer usar o clock interno ou um clock externo. Também contém o contador de programa PC e o seletor de estágios, que envia sinais de controle para as micro operações seguintes. Estas que vão receber os sinais de controle, são:

- O destaque em vermelho, que é o "decodificador de instruções" + o "registrador de offset de saltos";
- O destaque em laranja, composta pelos habilitadores de entrada, as operações lógico-aritméticas, deslocamento, movimentação e saltos.
- O destaque em amarelo são os "bancos de registradores" para serem lidos ou escritos pelo Assembly;
- Enquanto que o destaque em roxo, são as "portas de I/O" para se comunicar com o mundo externo.

As setas vermelhas ao lado das linhas de conexões, demonstram o direcionamento dos "barramentos internos" do WR80, se os dados vão para a esquerda, direita, cima ou baixo, muitas vezes realizando uma volta inteira no circuito, ou se desviando para outras direções.

Se decidíssemos encapsular uma ou mais micro operações em um mesmo componente, criamos as "Unidades" (Será visto adiante). O que está destacado em vermelho e azul claro, poderá fazer parte da "Unidade de Controle" (UC), que vai se comunicar com o circuito destacado em laranja, que se trata da "Unidade Lógico-Aritmética" (ULA), que por sua vez, escreve ou lê valores ou resultados de/para o "Banco de Registradores" ou "Unidade de Entrada ou Saída" (E/S) que está destacado em amarelo e roxo, respectivamente.

Nesta arquitetura, a entrada e saída de dados é dividido em duas partes: Comunicação Interna e Comunicação Externa. A interna se trata dos bancos de registradores - registradores de propósitos gerais (R0, R1, R2 e R3) + registradores dedicados/especiais (IR, PC, SR e Registro de Saltos) e a externa se trata dos registradores de E/S (P0, P1, P2 e P3) + as pinagens conectados nestes registradores.

Na próxima imagem, observamos uma representação do encapsulamento das unidades, cada uma contendo uma microarquitetura interna identificada por cores que se associam com as cores destacadas da imagem anterior, exceto as cores violeta e verde, onde a violeta são os registradores internos de dados, associadas em uma constância maior com a entradas da ULA e a cor verde que referencia cada operação da ULA.

3.5.1. Diagramação das Unidades

Veremos um pouco sobre a estrutura diagramática do processador WR80. Os diagramas fornecem um modelo claro do funcionamento de um sistema. Por este motivo, é necessário estabelecer os componentes principais que fluem por esta comunicação. Cada componente pode ser posteriormente encapsulado, visando uma melhor manutenção e escalabilidade do hardware.

Diagramação das Unidades

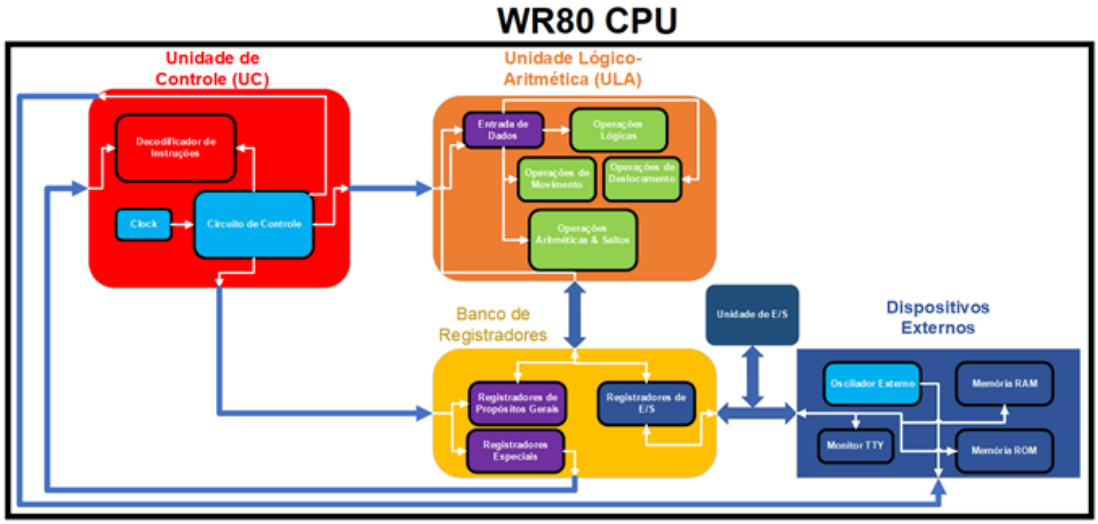


Figura 28 - Comunicação das Unidades pelo Diagrama

O WR80 CPU é uma organização que é composta por: Uma unidade de controle que é alimentada, ou por um clock interno, ou por um clock externo (oscilador externo). Ela divide os seus sinais de controle para as outras unidades, inclusive para o próprio decodificador que está dentro dela, onde ele recebe o sinal no estágio de decodificação.

A ULA recebe esse sinal no estágio de execução, o banco de registradores recebe no estágio de escrita e o dispositivo externo - Memória RAM - Recebe no estágio de busca, no entanto, quando se trata de operações rotineiras no Assembly, os dispositivos externos receberão o sinal do estágio de escrita através da unidade de E/S (Que por sua vez recebeu o sinal dos registradores de E/S).

3.5.2. Encapsulamento das Unidades

Para tornar prático o conceito da diagramação, compreenderemos como tudo é encapsulado, dividindo melhor a responsabilidade dos componentes.

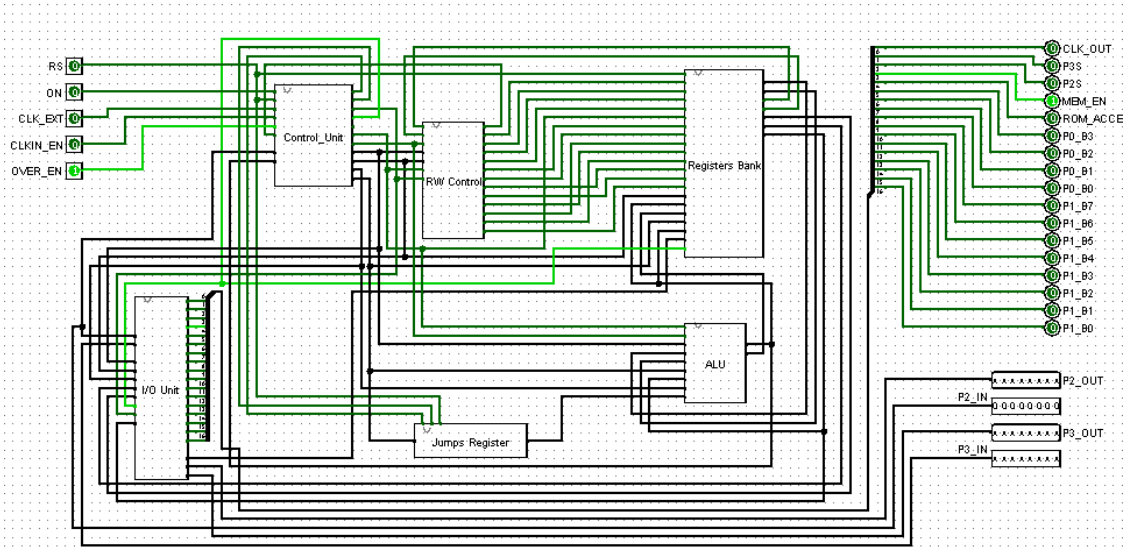


Figura 29 - Encapsulamento das Unidades do processador

É possível perceber uma boa divisão das unidades, encapsulando as micro operações e facilitando na manutenção do circuito. Temos os CIs:

- **Control_Unit** – Responsável pelo controle de decodificação e contagem do endereço do programa, além do controle de circuitos.
- **R/W Control** – Que recebe os sinais da Unidade de Controle (Control_Unit), no entanto, para controlar operações de leitura/escrita (R/W) de registros.
- **Registers Bank** – Sendo o banco de registradores de usuário e portas, no qual é controlado pela unidade de controle e o controle de leitura/escrita.
- **ALU** – A unidade lógico-aritmética, controlado pela unidade de controle e pelo controlador R/W, recebe dados do banco de registradores, unidade de I/O e do registro de saltos, a fim de realizar as operações.
- **I/O Unit** – A unidade de I/O (E/S – Entrada e Saída), no qual realiza uma interface entre o mundo externo e o mundo interno, recebendo e enviando dados de/para periféricos, como monitores, memórias, etc. Os dados são armazenados e operadores posteriormente.
- **Jump Register** – Um registrador auxiliar para armazenamento de endereços de saltos, a fim de mesclar os endereços de offsets coletados no formato de instrução e determinar a localização de desvios de memória.

O CI principal que recebe entradas iniciais do usuário é o Control_Unit, podendo aceitar um clock externo, operações de on/off e reset, além de configurações extras. A saída de dados é manuseada apenas pela I/O Unit, no entanto, mantendo entrada de dados operáveis pelo Control_Unit.

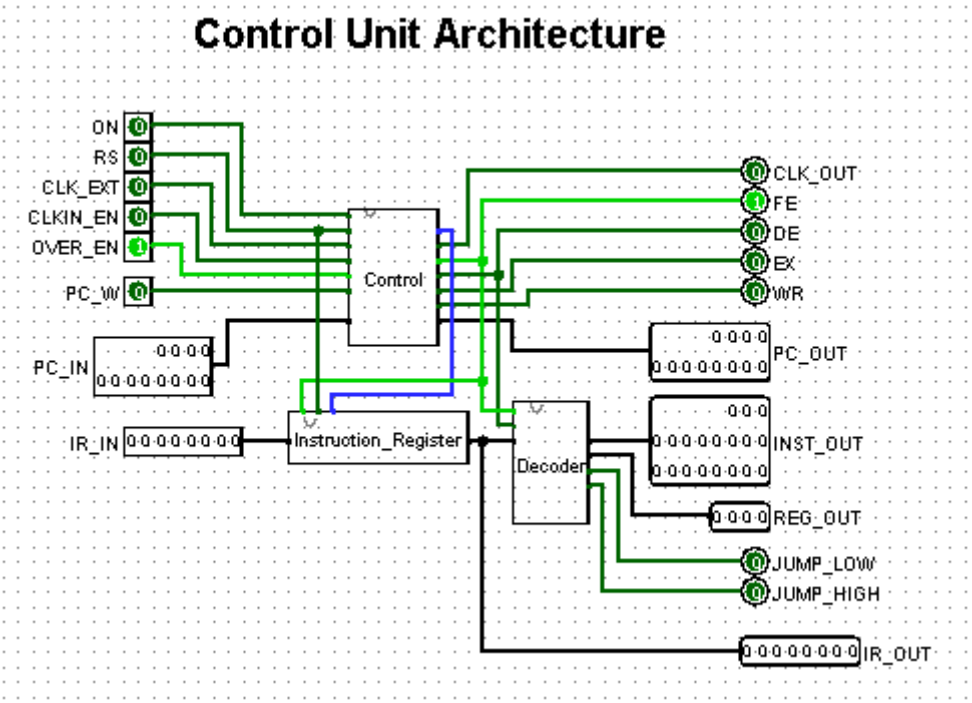


Figura 30 - Encapsulamento da arquitetura da unidade de controle

Esta arquitetura, sendo a parte interna do Control_Unit, é a própria unidade de controle, composta pelos circuitos de controle, que pode tanto receber sinais/dados externos como enviar sinais/dados para a camada acima. Também encontramos o **Instruction_Register** (Registrador de Instrução) que armazena o formato de instrução vindo da memória RAM e **Decoder** (Decodificador) que separa o formato de instrução a fim de selecionar uma linha específica.

Os dados de PC (Program Counter) são administrados por esta arquitetura, recebendo o endereço a fim de armazenar os cálculos das instruções de salto e enviando o resultado caso solicitado pelas mesmas instruções. Da mesma forma que os dados de IR (Instruction Register), no qual é necessário à sua **recepção** e **transmissão**, pelo armazenamento de instruções a serem decodificadas e depois pelo envio para instruções que solicitam dados que estão no formato de instrução, respectivamente.

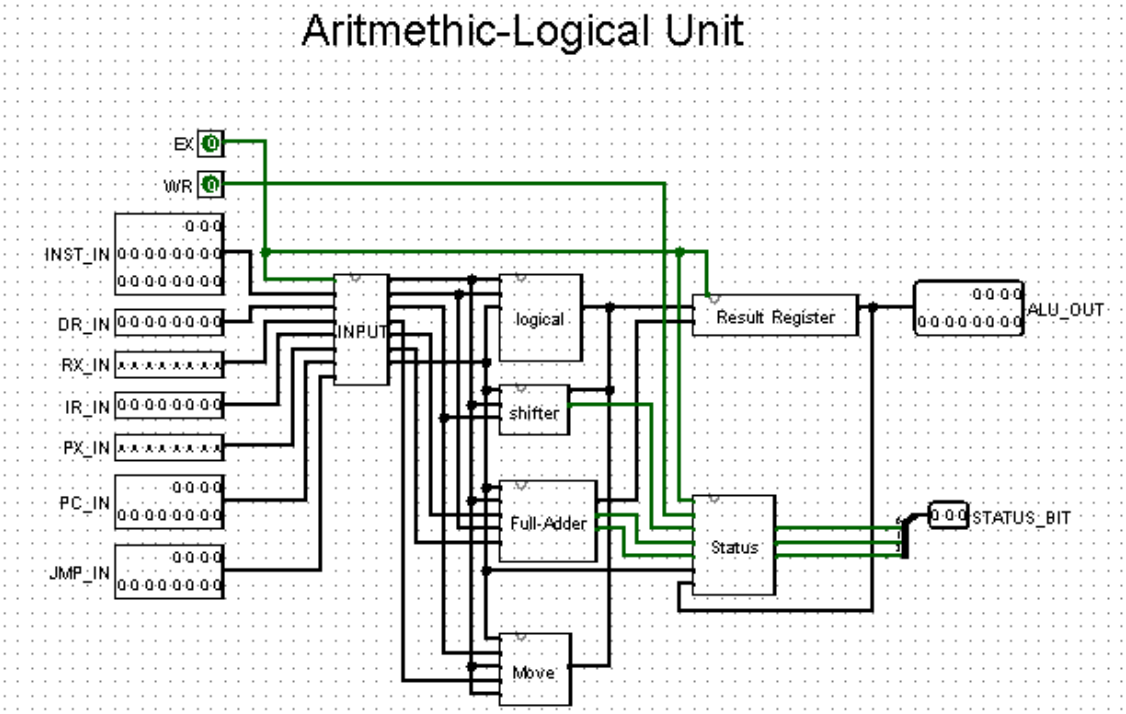


Figura 31 - Encapsulamento da arquitetura da unidade lógico-aritmética (ULA)

Acima vemos a unidade lógico-aritmética, composta pelo encapsulamento de Cis que realizam operações lógicas, de aritmética, de deslocamento e de movimento. O conteúdo destes Cis fora abordado em tópicos anteriores. Esta arquitetura apresenta o modelo de comunicação muito conhecido na computação:

- **Entrada** – Realizada pelo CI **input** que seleciona dados de diversas origens;
- **Processamento** – Realizado por todos os Cis: logical, shifter, full-adder e move.
- **Saída** – São os resultados entregues pelo processamento, armazenado temporariamente no **Result Register** (Registrador de Resultado) e **Status** (Estados das Operações).

3.5.3. Programação Assembly

A linguagem Assembly é uma representação direta para o código de máquina (opcodes) e se torna mais fácil criar um programa do que programar diretamente usando números hexadecimais ou binários. No entanto, numa arquitetura simplificada, a programação em opcodes se torna mais simples, devido ao formato de organização simétrica no modelo RISC.

O Assembly é uma camada acima de todas as abordadas e está na camada mais externa de um computador – ISA. Esta camada é uma arquitetura de conjunto de instruções (Instruction Set Architecture) que define o “Assembly” de um processador. Cada fabricante pode construir seu próprio esquema de circuitos, consequentemente, novas organizações de decodificação, que por sua vez, define um novo formato de Assembly.

Primeiramente, veremos de forma explícita quais são as instruções deste processador em formato binário:

1. Instruções do Processador:

1.1. Operações lógicas

1.1.1. Operação AND - 0000
1.1.2. Operação OR - 0001
1.1.3. Operação NOT - 0010
1.1.4. Operação XOR - 0011

1.2. Operações aritméticas

1.2.1. Operação ADD - 0100
1.2.2. Operação SUB - 0101

1.3. Operações de movimento

1.3.1. Operação ST - 0110
1.3.2. Operação LD - 0111
1.3.3. Operação IN - 1000
1.3.4. Operação OUT - 1001

1.4. Operações de Deslocamento

1.4.1. Operação SHR - 1010
1.4.2. Operação SHL - 1011

1.5. Operações de Comparação & Saltos

1.5.1. Operação BT - 1100
1.5.2. Operação JC - 1101
1.5.3. Operação JZ - 1110
1.5.4. Operação JP - 1111

2. Registradores/Armazenamento (4ª estágio do pipeline)

2.1. Registradores de dados/usuários

2.1.1. R0 – Registrador 0 – 00
2.1.2. R1 – Registrador 1 – 01
2.1.3. R2 – Registrador 2 – 10
2.1.4. R3 – Registrador 3 – 11

2.2. Registradores de portas E/S

2.2.1. P0 – Porta 0 – 00 (Endereço)
2.2.2. P1 – Porta 1 – 01 (Endereço)
2.2.3. P2 – Porta 2 – 10 (Dados)
2.2.4. P3 – Porta 3 – 11 (Dados)

2.3. Registradores Internos (Overhead)

2.3.1. IR – Registrador de Instrução (Instrução)
2.3.1.1. OR – Registrador de Opcode

2.3.1.2. AR – Registrador de Endereço/Dado
2.3.2. DR – Registrador Acumulador
2.3.3. SR – Registrador de STATUS (4 bits)
- Bit Carry (0) = Bit C
- Bit Zero (1) = Bit Z
- +2 bits reservados (Interrupção, Modos, etc. etc..)

2.3.4. PC – Contador de Programa
2.3.5. Offset – Registro de deslocamento de saltos
2.3.6. RR – Registrador de Resultados da ULA

Note que temos os binários de instruções, que especifica as operações e os binários de endereçamentos para o armazenamento de dados. Considerando que nossos opcodes está nos 4 bits mais significativos do formato de instrução e o endereçamento de registradores estão nos 4 bits menos significativos, é possível formar os pares de **instrução – registrador** (Onde o registrador é um operando).

Vamos colocar como exemplo o cenário onde precisamos armazenar o valor 2 no acumulador. Como a instrução que realiza este procedimento é o ST (Store), aceitando nos seus 4 bits LSB um número literal, que em nosso caso é o 2, logo a instrução completa seria **ST 2,** no qual DR armazena o valor 2. Se checarmos os binários das instruções, teremos a instrução: **0110 0010.** Onde **0110** é o opcode **ST** e **0010** é o número literal **2**.

Já que o acumulador se encontra com o valor 2, poderemos "carregar" este valor em um registrador, e agora precisamos se atentar ao formato de par **instrução – registrador**. O opcode **LD** (Load) é responsável por realizar este carregamento, então iremos como exemplo carregar no registrador **R1.** A instrução completa seria **LD R1,** isto é, "Carrega o valor que está no acumulador para o registrador 1". Isto em binário seria **0111 0001**, onde **0111** é o opcode do **LD** e **0001** é um endereçamento do registrador 1 (R1).

Perceba que agora ao invés de utilizarmos os 4 bits LSB como um número literal, tal como aconteceu com o ST, agora usamos como um endereçamento. A maior parte das instruções do WR80 opera sobre este último formato. E se quisermos adicionar o valor que já está em DR (2) pelo valor que foi carregado em R1 (2)? Para isto, apenas utilize a operação **ADD.**

O novo comando ficaria **ADD R1**, onde DR = DR + R1, desta forma, o acumulador vai somar o valor 2 em DR com o valor 2 em R1 e entregar o resultado no próprio acumulador DR, que é o valor 4. No formato de instrução,

ficaria **0100 0001**. Onde **0100** é o **ADD** e **0001** é o **R1.** Perceberam a "relação direta" entre um mnemônico Assembly com os opcodes binários? O Assembly RISC proporciona uma maior identificação, enquanto que o Assembly CISC, por ser mais complexo, contém outros elementos de identificação nos bits do formato de instrução.

Este programa completo em binário na memória RAM ficaria: **01100010 01110001 01000001**. O mesmo programa em formato hexadecimal, facilitaria na visualização, se tornando: **62 71 41**. Mas para trabalhar em Assembly, precisamos de um sistema operacional com um simples editor no qual possamos escrever nossos mnemônicos (símbolos Assembly), o **assembler** (Montador – não confunda com **Assembly**) seria a ferramenta no qual faria a conversão destes mnemônicos da linguagem Assembly para o programa binário que demonstramos. Portanto, em um arquivo, o programa em Assembly ficaria:

MeuPrograma:

ST 2

LD R1

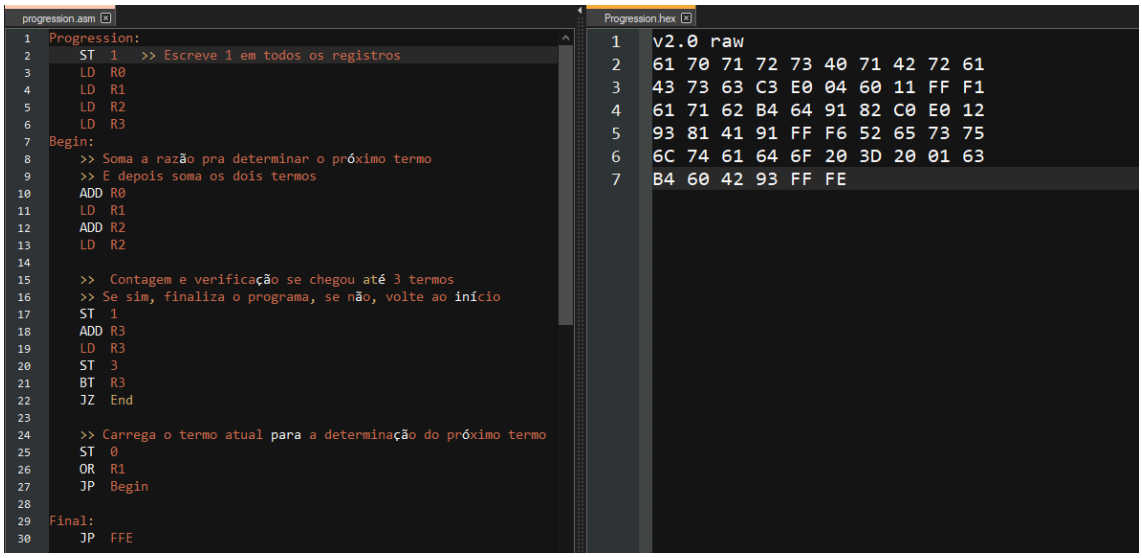
ADD R1

É claro que a label *MeuPrograma* é só uma forma de delimitar o código Assembly, facilitando na visualização e operações de salto. O montador converteria esta label em um formato de endereço no qual instruções como **JC** ou **JZ** poderia operar de maneira mais clara sem precisar especificar diretamente o endereço de um programa Assembly. Porém a label *MeuPrograma* não iria ser convertido para algum tipo de comando binário no código final (Apenas ajuda o desenvolvedor).

A missão deste E-book não é detalhar o conceito interno de montadores & compiladores de linguagens baixo-nível e alto-nível, pois poderia ficar muito extenso. Já que montadores e compiladores são projetos elaborados por desenvolvedores que querem colaborar com a arquitetura, criando facilidades para os outros desenvolvedores que fossem criar programas mais complexos para este processador. No entanto, este é um tema que podemos reservar para um próximo E-book dedicado inteiramente a este assunto.

Para enfatizarmos melhor a programação Assembly do WR80, após um possível montador existir, mostraremos um exemplo de um programa Assembly para calcular a progressão aritmética na razão de 1, ou seja, a razão vai determinar de quantas em quantas unidades a sequência seria, como: **1, 2, 3...** (razão de 1), **0, 2, 4, 6,...** (razão de 2), etc... E após isto, realizaria uma somatória de toda a sequência. Em nosso caso, seria **3 termos** na razão de '1': **1, 2, 3**. E sua somatória seria: **1 + 2 + 3 = 6.**

O programa vai mostrar na tela uma String e o resultado do cálculo. Apresentaremos o formato mnemônico do código e o formato hexadecimal deste programa, e depois o seu resultado no circuito após carregar o formato hexadecimal na memória RAM do simulador LogiSIM.



progression.asm

```
Progression:
    ST  1   >> Escreve 1 em todos os registros
    LD  R0
    LD  R1
    LD  R2
    LD  R3
Begin:
    >> Soma a razão pra determinar o próximo termo
    >> E depois soma os dois termos
    ADD R0
    LD  R1
    ADD R2
    LD  R2

    >>  Contagem e verificação se chegou até 3 termos
    >> Se sim, finaliza o programa, se não, volte ao início
    ST  1
    ADD R3
    LD  R3
    ST  3
    BT  R3
    JZ  End

    >> Carrega o termo atual para a determinação do próximo termo
    ST  0
    OR  R1
    JP  Begin

Final:
    JP  FFE
```

Progression.hex

```
v2.0 raw
61 70 71 72 73 40 71 42 72 61
43 73 63 C3 E0 04 60 11 FF F1
61 71 62 B4 64 91 82 C0 E0 12
93 81 41 91 FF F6 52 65 73 75
6C 74 61 64 6F 20 3D 20 01 63
B4 60 42 93 FF FE
```

Figura 32 - Formato mnemônico e hexadecimal do programa de progressão aritmética

No lado esquerdo, temos o nosso programa de progressão aritmética em formato Assembly, incluindo **comentários**, **labels** e os nossos **mnemônicos** das instruções, isto é, os símbolos de programação. No lado direito, temos o mesmo programa, no entanto, escrito diretamente em código de máquina (no formato hexadecimal), usando o termo inicial **v2.0 raw,** que é o termo de cabeçalho exigido pelo simulador LogiSIM. Em seguida, o resultado na arquitetura externa do circuito (Computador WR80):

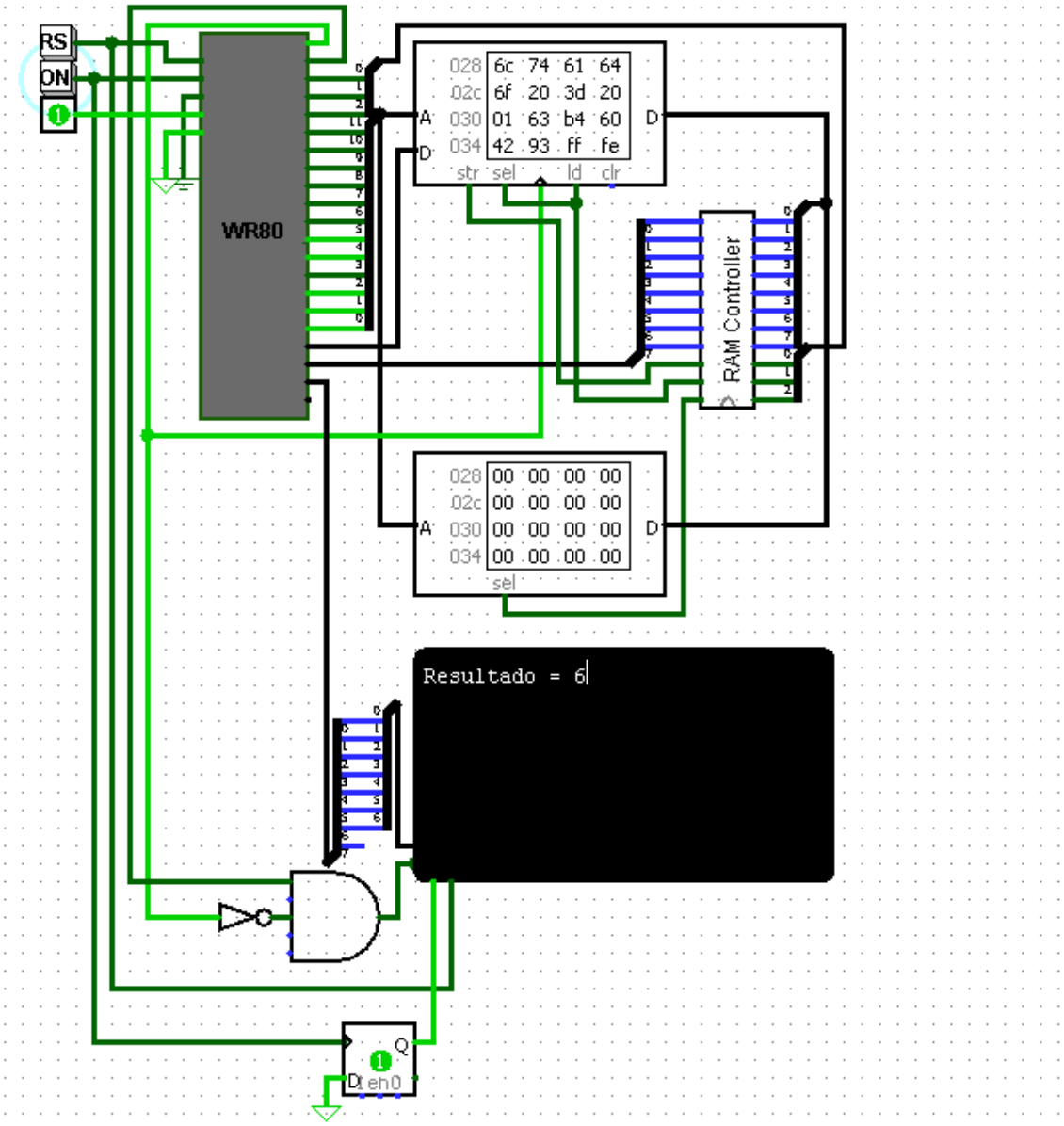


Figura 33 - resultado do programa de progressão aritmética na arquitetura externa

Desta forma, após carregar o programa em formato hexadecimal na memória RAM do LogiSIM, temos após a execução do código o valor “Resultado = 6” escrito no monitor TTY. Para executar, é necessário pressionar o botão **RS** para resetar os registradores, o botão **ON** para ligar o processador (simulando a liberação de corrente) e no 3ª botão de estado lógico **1**, para converter seu estado de 0 para 1, ou seja, este último é o botão para informar ao processador que queremos que o processador trave após finalizar o programa.

Perceba que nós temos uma memória ROM que está vazia e um controlador RAM Controller que irá controlar a comunicação externa de um destes dispositivos com o processador WR80, que se encontra no lado esquerdo. O programa foi executado no Clock de 4.1KHz do LogiSIM.

3.6. Análise e Planejamento de Hardware

Neste tópico será abordado sobre as análises de clock do LogiSIM, comparando com o Clock do Proteus, também faremos um paralelo com a análise de pipeline, compreendendo um pouco sobre o paralelismo vs sequencial. Para finalizar, mostraremos os planejamentos de implementações futuras.

3.6.1. Análise de Frequência e Clock

Faremos uma observação quanto ao Clock do LogiSIM: A velocidade real baseada na frequência configurada neste simulador não é a esperada e adequada, explicaremos o porquê.

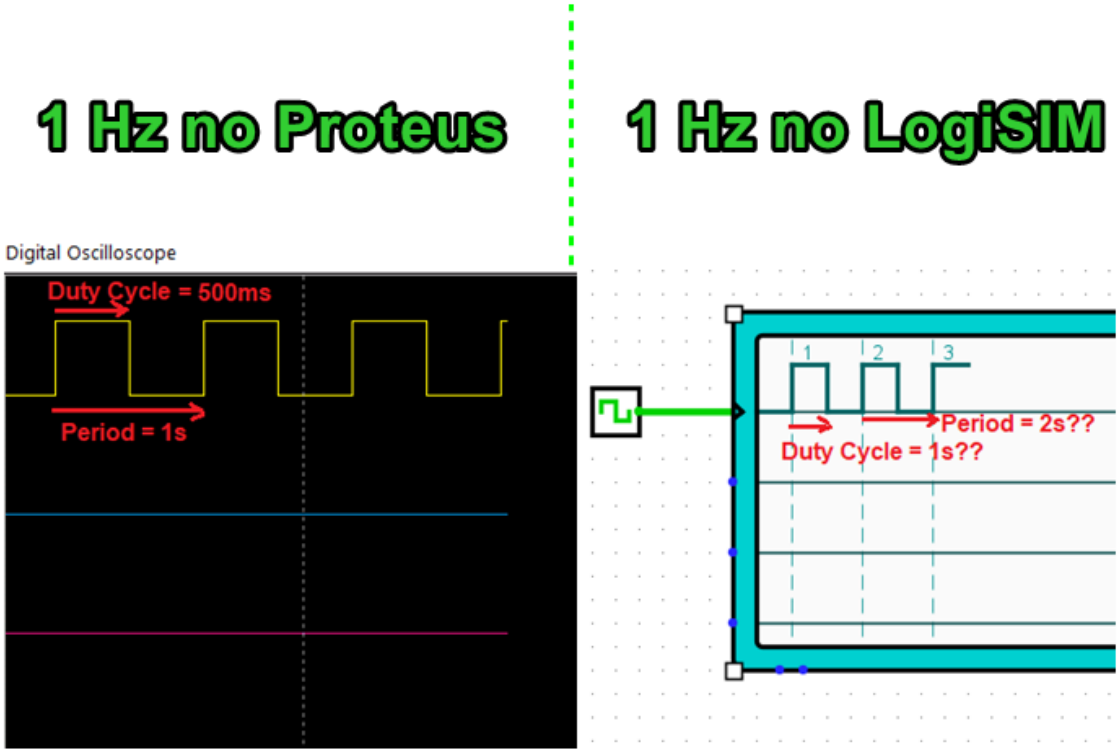


Figura 34 - Comparação de frequências do Proteus e LogiSIM

Na vida real, Se temos 1 Hz de Frequência, logo sabemos que é 1 pulso por segundo, ou seja, no 1ª segundo ocorre uma transição de subida do Clock, que se mantém neste estado 1 por 1/2 do tempo, que é 500ms. Nesta 2ª metade, o clock ocorre uma transição de descida, se mantendo no estado 0 por +500ms. Quando iniciar o próximo segundo uma nova transição de subida deve ocorrer, portanto, no segundo atual temos 1 pulso e no segundo posterior temos outro pulso e assim por diante, o que faz termos 1 pulso por segundo.

O tempo que ele se mantém naquele estado, dentro do segundo, é chamado de "Duty Cycle" (pode ser entendido como um percentual relativo ao período), já o tempo entre a primeira transição de subida para a próxima transição de subida, é chamada de "Período".

A frequência desse clock determina este período, então se falarmos que um Clock tem a frequência de 2 Hz (2 Hertz), significa que ele deve dá 2 pulsos a cada segundo, portanto, para a obtenção desta velocidade, o tempo de Duty Cycle deve ser a metade de 500ms = 250ms (ou 25% do período anterior e 50% do período atual). É o mesmo que dividir 1/4.

O cálculo para identificar o tempo de permanência de um estado de Clock (0 ou 1) dentro de 1 segundo, seria as variáveis: *1_SEC / (QUANT_ESTADOS QUANT_PULSOS)*, vamos chamar *QUANT_PULSOS* de "Freqz" e *QUANT_ESTADOS* da constante '2', já que este número não muda (Clock só pode ter 2 estados), e *1_SEC* pela constante '1' que é a base do segundo:

1/ (freqz x 2) = Duty Cycle. Portanto, se temos uma frequência de 4 Hz (4 pulsos por segundo), *1 / (4*2) = 0,125s* de Duty Cycle (125ms). Os cálculos aqui é considerando que usamos um Duty Cycle de 50% (*High Time* / *Period*).

Logo, se você divide um tempo de permanência de um estado por 2, você duplica a velocidade ou frequência do Clock e se você multiplica a frequência, divide o Duty Cycle. Porém, o LogiSIM faz com que, na frequência de 1 Hz, a cada segundo uma nova transição ocorra, seja de subida ou descida, o que não está certo, pois o pulso na vida real considera que apenas a transição de subida ocorra no tempo exato, e não duas transições a cada tempo.

Portanto, no LogiSIM 1 Hz na verdade se torna 1 segundo de Duty Cycle, fazendo com que a frequência em termos reais e práticos, seja dividido ao meio e o Período seja igual a 2 segundos. Logo, se você utiliza a frequência máxima do LogiSIM, que é 4.1 KHz (4100 Hertz), na verdade você estará utilizando 2.05 KHz (2050 Hertz). Veja a comparação da Figura acima, que identificamos a velocidade correta no Proteus, mas incorreta do LogiSIM!

Testei a versão mais recente do LogiSIM e por incrível que pareça, isto ainda ocorre. Enquanto que no Simulador Proteus 8, este erro não ocorre, mantendo a velocidade real do clock. Tudo que preciso fazer, é migrar todo o

circuito para a versão mais nova, não pelos motivos do Clock (Já que isto não se resolveria), mas por outros recursos mais avançados, inclusive para outros simuladores também.

3.6.2. Análise de Pipelines – Sequencial ou Paralelo?

O nosso circuito utiliza uma execução de instruções de forma sequencial (4 estágios executando um após o outro, sem pipeline), o que significa que uma próxima instrução só executa quando de fato os 4 estágios finalizam.

E se cada estágio executa em 1 pulso de clock, nossas instruções executam completamente em 4 pulsos, portanto, dividimos a frequência real máxima por 4 -> *2050 Hz / 4 estágios = 512,5 Hz*. Isto é, nossa frequência real de instruções (Ciclo de Instruções) é 512 Hz e não 4.1 KHz.

Mesmo se utilizássemos a versão mais nova do LogiSIM, que utiliza uma frequência máxima de Clock de 2 MHz, considerando que o erro foi resolvido, da mesma forma dividiríamos por 4, por causa da forma sequencial, sendo a frequência real = 500 KHz (Como ainda tem o erro, então é 250 KHz).

Na próxima imagem, mostro um exemplo de 3 estágios, de como seria SEM pipeline, onde I é o estágio de busca, E de execução e D de operações de escrita/leitura de Dados. Uma próxima instrução só executa no próximo "I" quando I E D finalizaram.

Considere que alguns dos estágios podem não ser finalizados, como no caso do 3ª e 5ª ‘E’ (Execução), pois há cenários em que instruções de salto possa não ser satisfeitas em certas condições, evitando a escrita de dados no estágio ‘D’. Veremos esta ilustração.

Estágios Sem Pipeline

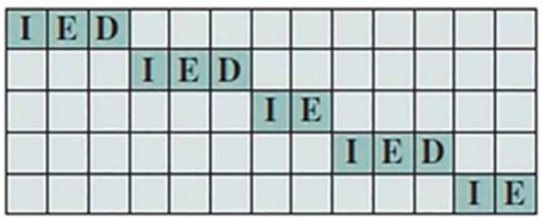


Figura 189 – Execução de instruções sequencialmente

Figura 35 - Três estágios executando de forma sequencial

Agora, eu mostro de como seria COM pipeline na próxima imagem. Nós temos um paralelismo de micro operações diferentes (Pré-paralelismo), ou seja, busca a primeira instrução e vai para execução, enquanto a busca está livre, busca a segunda instrução, enquanto executa a primeira, e quando a primeira efetua sua escrita/leitura de dados no estágio D, executa a segunda instrução e já busca a terceira.

Estágios Com Pipeline

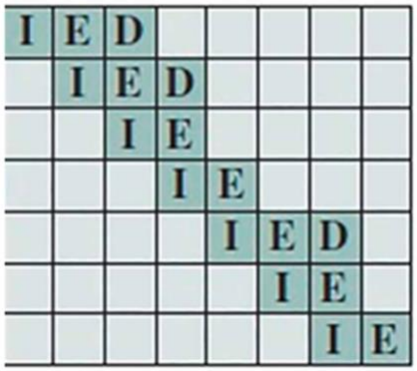


Figura 191 – *Pipeline* de três estágios

Figura 36 - Três estágios executando de forma pré-paralela (Paralelismo de Estágios)

Desta forma, no final do estágio D de cada linha, temos 1 pulso de Clock, que seria a instrução completamente executada, o que nos diz que, cada instrução executaria em 1 pulso de clock, determinando a frequência real de Clock para o Ciclo de Instruções. Isto é, se o Clock = 2MHz, teríamos 2 milhões de instruções executadas a cada segundo e não 500 mil instruções, como na divisão por 4 estágios sequenciais.

A implementação do Pipeline seria válida, se libertássemos os estágios anteriores para novas operações, criando N seletores de X estágios, onde N = X. O "Superpipeline" já consideraria esta mesma lógica, mas efetuando cada estágio no "Duty Cycle" e não no período, isto também corrigiria o problema da frequência errada do LogiSIM, tornando um Pipeline virtualmente e um Superpipeline fisicamente (em termos de circuitos reais).

3.6.3. Implementações Futuras & Investimentos

Sobre implementações futuras, pretendo utilizar o bit de extensão de opcodes, que é 1 bit reservado/zerado em algumas instruções, normalmente ele é o bit<3> em instruções que usam Rx.

Através deste bit, será possível "estender" a quantidade de opcodes, assim podemos criar as instruções **call** e **ret** para chamada e retorno de rotinas. Podemos aumentar a quantidade de registros de I/O, como de P4 a P7, onde P4 e P5 poderão ser o endereço de pilha da memória RAM.

As Instruções **push**, **pop**, **call** e **ret** iriam controlar P4 e P5, enquanto que **call** e **ret** utilizaria a mesma lógica das instruções de salto, tanto na soma, quanto na divisão de ciclos, com a diferença de que, utilizariam a pilha em P4:P5 para armazenar/ler os endereços de retorno, respectivamente.

A integração do circuito em uma placa FPGA seria essencial, convertendo toda a nossa arquitetura em um VHDL (Linguagem de Descrição de Hardware), realizando otimizações para vermos o funcionamento na vida real da melhor forma possível. No caso do FPGA, poderíamos utilizar um Clock de até 50 MHz! Outra forma é projetar usando transistores a fim de aprendizado, criando placas de circuitos impressos manualmente.

Compreendendo o processo manual e fazendo do zero, como fazemos até agora, posteriormente seria interessante passar a "automatizar" estes processos, enviando os circuitos pré-otimizados para fábricas de produção para o retorno de nossos CIs. É neste sentido que poderíamos ter um microprocessador bem mais elaborado, com mecanismos de conversão AD e DA (Analog-Digital e Digital-Analog), transmissão serial via TX e RX (Pinos de transmissão e recepção), como também, inúmeras portas de I/O!

A conversão DA e AD poderiam ter instruções próprias para realizar leitura de dados analógicos do mundo externo ou envio de dados digitais para o mundo externo. Instruções como **dac** e **adc**, possibilitariam que novos registradores nascessem para armazenar os resultados das conversões, podendo ser uma das portas de I/O.

A transmissão serial focaria no envio e recepção de dados bit a bit por uma linha de conexão. A configuração poderia ser feita por instruções próprias como **txr** e **rxr**, que operaria em cima de bits de registradores especiais, a fim de traçar uma comunicação entre dois dispositivos. Adicionar mais portas de I/O poderia facilitar neste processo de configuração e armazenamento de dados.

CONCLUSÃO

Neste E-book foi possível explorar o tema de processadores digitais na forma mais íntima possível, abordando desde circuitos individuais para processamento e controle de bits, até a junção de circuitos formando um hardware gerenciável, capaz de entregar tarefas comuns no ramo da computação.

Primeiro foi enfatizado os deslocadores de bits, adotando as formas síncronas usando circuitos sequenciais (flip-flop) e as formas assíncronas, através de componentes multiplexados. Com o deslocamento de bits, compreendemos como certas operações aritméticas executaria de forma mais rápida, além de entender o desvio do fluxo de dados digitais e uma pequena introdução ao armazenamento serial.

Após isto, focamos na capacidade do processador em se comunicar com o mundo externo, através das Portas E/S. Foi possível apresentar da maneira mais arcaica o funcionamento de uma interação, isto é, a forma com que o processador possibilita que um usuário utilize um computador usando periféricos. A unidade de I/O com suas portas e instruções de movimentação, desempenha um papel crucial na interação do usuário com um software.

Por fim, abordamos uma arquitetura inteira de um processador RISC e SAP, isto é, a maneira simplificada de circuitos processarem comandos binários. Percebemos que para a execução de tarefas programáticas, bastaria um seletor de estágios, dividindo o fluxo de execução em unidades específicas, cada um comunicando entre si.

Compreendemos no último capítulo como a CPU busca as instruções da memória, decodifica, processa e armazena os resultados em pequenos espaços, como também vimos quais são estas instruções e de que maneira elas poderiam ser utilizadas para se construir pequenos programas, elementos essenciais para compor firmwares e softwares.

Foi apresentado na finalização a arquitetura das unidades através de diagramas e encapsulamentos, além de análises de futuras implementações de hardware.

REFERÊNCIAS

Malvino, Albert Paul; Brown, Jerald A. (1993). Digital Computer Electronics (3 ed.). McGraw-Hill. pp. 140–212. ISBN 0-02-800594-5.

Malvino, Albert Paul; Brown, Jerald A. (1993). Digital Computer Electronics (3 ed.). McGraw-Hill. pp. 143–144. ISBN 0-02-800594-5.

Eater, Ben. Build an 8-bit computer from scratch. Ben Eater. Disponível em: <https://eater.net/8bit/control>. Acessado em: 10 Aug. 2024.

KarenOk. Designing and Implementing a SAP-1 Computer. KarenOk. Disponível em: <https://karenok.github.io/SAP-1-Computer>. Acessado em: 11 Aug. 2024.

Malvino, Albert Paul; Brown, Jerald A. (1993). Digital Computer Electronics (3 ed.). McGraw-Hill. Disponível em: < https://drive.google.com/file/d/1g9VK7DQuiCMUlins-J77bPicDu4BEXPz/view>. Acessado em: 20 Aug. 2024.

Tanenbaum, Andrew S; Austin, Todd. Organização estruturada de computadores. Tradução Daniel Vieira. Revisão Técnica. Wagner Luiz Zucchi. São Paulo: Pearson Prentice Hall, 2013. Disponível em: < https://github.com/free-educa/books/blob/main/books/Organização%20estruturada%20de%20computadores%20-%20Tanenbaum.pdf>. Acessado em: 22 Aug. 2024.

Souza, Pedro. Circuitos Digitais. 2020. Disponível em: <https://www.youtube.com/playlist?list=PLXyWBo_coJnMYO9Na3t-oYsc2X4kPJBWf>. Acessado em: 22 Aug. 2024.

Sichman, Jaime S. Engenharia de Computação – Circuitos Lógicos – 10ª bimestre. 2016. Disponível em: <https://www.youtube.com/playlist?list=PLxI8Can9yAHeWyA5-3n4TrLZMa1YgaBAS>. Acessado em: 22 Aug. 2024.