Roteiro – Projeto 01 – Processador

Olá, estou aqui hoje para falar a respeito do projeto 01 da matéria de sistemas digitais.

O projeto consiste na implementação (OLHAR PDF) ...

Mas você sabe o que é um processador?  
  
Um processador programável, também conhecido como processador de propósitos gerais é um circuito digital cuja tarefa de processamento em particular, ao invés de ser construída no próprio circuito, fica armazenada em uma memória. A representação dessa tarefa de processamento é conhecida como programa. Um processador pode ser construído seguindo duas arquiteturas existentes: A de Harvard, no qual, as memorias de instrução e de dados são separadas e a de Von-Neumann, no qual, as memorias de instrução e de dados são unidas em um único bloco. Além disso, ele consiste em duas partes principais: um bloco operacional (datapath) e uma unidade de controle.  
  
No bloco operacional básico, temos dentro deles a memória de dados (irá guardar todos os dados que um processador programável irá acessar), um banco de registradores e uma unidade lógica e aritmética (ULA). Assim, podemos ver um processamento como sendo dividia em três partes: primeiro temos a carga de dados, significando ler os dados que estão guardados em qualquer local da memoria de dados, e coloca-los em qualquer um dos registradores do banco de registradores. Logo em seguida, temos a transformação desses dados no qual iremos passar os dados de 2 desses registradores através da ULA que está configurada para qualquer uma das operações suportada por ela, e colocando o resultado de volta em qualquer um dos registradores do banco de registradores. É por último, temos o armazenamento (escrever) dos dados, em que, pegamos os dados que estão em qualquer um dos registradores do banco de registradores e depositamos em qualquer lugar da memória de dados.

É necessário descrever uma sequência das operações que desejamos executar no bloco operacional. Essa descrição é chamada de instruções e uma coleção de instrução é conhecida como programa. Esses programas são armazenados na chamada memória de instruções e é aí que a unidade de controle desempenha sua função, ela irá lê cada instrução da memoria de intrusões e então executa essa instrução no bloco operacional. Do mesmo modo que o bloco operacional, a unidade de controle é dividida em três etapas: Busca, Decodificação e Execução.

Na busca, a unidade de controle começa lendo a instrução na memória de instrução, para isso faz-se necessário ter um registrador, chamado de registrador de instrução para salvar essa instrução e um contador, chamado de contador de programa para saber a próxima posição que deverá ser buscada a instrução subsequente na memoria de instrução. O próximo passo seria a decodificação no qual irá determinar qual a operação para aquela instrução desejada lendo o seu opcode. E assim, por ultimo entrando na parte de execução, onde enviará caso necessário os linhas de controle apropriados para o bloco operacional. Para isso tudo funcionar corretamente, a unidade de controle precisa de um bloco de controle que vai efetuar repetidamente o processor de busca, decodificação e execução.

Vale lembrar, que tipicamente, um certo número de bits da instrução é reservado pelos conjuntos de instruções para indiciar qual é a operação que deverá ser realizada. Os bits restantes especificam informações adicionais que são necessárias à execução da operação, como registradores de origem e destino. Esses bits são conhecidos como código de operação da instrução (opcode), e os bits restantes representam os operandos, indicando com quais dados deve-se operar.

Voltando agora para o nosso problema inicial, foi pedido para criarmos um processador que atende a todas as funções previstas nessa tabela. Apesar de haver 19 funções, nosso projeto é composto por 22 instruções;  Isso se deve pelo fato da função “MOV” poder desempenhar 3 instruções diferentes: **“Carregar” (LOAD)** um dado de 8 bits da memória de dados no banco de registradores, **“Salvar” (SAVE)** um dado de 8 bits do banco de registradores  na memória de dados e **“Carregar uma Constante” (LOAD CONST)** fornecida na instrução diretamente no banco de registradores. Além disso, como pedido, a função “JMP” deve poder ser acionada de duas formas diferentes, uma forma de instrução condicional **“Pular quando 0” (JUMP if “0”),** pulando para um endereço quando o dado de um registrador for nulo e, de forma incondicional, **“Pular”** **(JUMP UN)** um endereço dado na instrução.

Nosso código de instruções possui 32 bits, para que operações que necessitem a presença de três registradores mais o vetor de operação (opcode), ele consiga carregar toda informação necessária para realização das operações. Dessa forma a instrução **IN[31:0]** é didaticamente dividida em **A[7:0]B[7:0]C[7:0]D[7:0]**, de acordo com o barramento I/O que alimenta o processador**.** Assim, ficou divido que o A[7:0] ficará guardada o opcode para a escolha da função desejada na ULA, e os demais B[7:0],C[7:0] e D[7:0] ficam guardando as informação para os registradores ou posição da memória de dados. Para todas as instruções, a sintaxe e forma de representação de acordo com o barramento de instrução fica:

A partir disso, projetamos nossa unidade lógica e aritmética (ULA) para atender todos os requisitos do projeto do microcontrolador. Ademais, é visível que a ULA realiza todas as operações lógicas e aritméticas assincronamente. A saída do componente é ditada pelos bits de seleção, que funcionam para informar a ULA que resultado de qual operação deve ser liberado. Assim, as representações em componentes de cada bloco da ULA são:

→ **ADD**: Somador de dois dados de 8 Bits;

→ **SUB**: Substrato de dois dados de 8 Bits;

→ **COMP**: Comparador de igualdade entre dois dados de 8 Bits;

→ **MULT**: Multiplicador de dois dados de 8 Bits;

→ **INC**: Somador de dois dados de 8 Bits, sendo o primeiro dado **A** e o segundo um vetor de 8 bit com valor decimal de “1”;

→ **DEC**: Subtrator de dois dados de 16 Bits, sendo o primeiro dado **A** e o segundo um vetor de 8 bit com valor decimal de “1”

→ **AND**: Componente que realiza a operação booleana “And”, bit a bit, para dois vetores de 8 bits.

→ **OR**: Componente que realiza a operação booleana “Or”, bit a bit, para dois vetores de 8 bits.

→ **XOR**: Componente que realiza a operação booleana “Xor”, bit a bit, para dois vetores de 8 bits.

→ **NOT**: Componente que realiza a operação booleana “Not”, em cada bit de um vetor de 8 bits (**A**).

→ **SHL**: Multiplicador de dois dados de 8 Bits, sendo o primeiro dado **A** e o segundo um vetor de 16 bit com valor decimal de “2”;

→ **SHR**: Divisor de dois dados de 8 Bits, sendo o primeiro dado **A** e o segundo um vetor de 16 bit com valor decimal de “2”;

Vale ressaltar que os vetores A e B, ambos de 8bits não são os mesmos usados para o barramento. Esses vetores são exclusivamente representações de saídas dos registradores do banco de registradores.

Como foi apresentado anteriormente nesse vídeo, a máquina de estado do bloco de controle irá ficar em um ciclo de busca -> decodificação -> execução. O nosso não é diferente disso. Começando no estado de START que serve para iniciar o processador e a contagem de instruções, logo em seguindo indo para o FETCH que atua capturando a instrução da sequência e assim indo para o DECODE que direciona para o estado de atuação da função, de acordo com o opcode. E assim gera a EXECULÇÃO no bloco operacional. Após, ficará circulando entre FETCH > DECODE > FUNCTION. Essas instruções so possuem duas exceções, na instrução de HLT e JUMP IF 0. Ao chegar no estado de HLT, o processador deve parar completamente, sendo assim este estado circula em si mesmo, impedindo que seja lida e executada uma nova instrução. Já o **JUMP IF ‘0’** e um exceção pelo fato de necessitar dois pulsos de clock caso a condição do salto seja satisfeita, um pulso para verificar a condição e outro para executar o pulo.

Assim, nosso bloco operacional e bloco de controle ficaram da seguinte maneira. Como podem ver, existe duas memorias separadas, uma para dados e outro para instruções denotando a escolha de uma arquitetura Harvard. Ademais, podem ser vistas outros blocos que não foram apresentados nesse vídeo como: **Pilha (LIFO)** para realizar as ações de **PUSH** e **POP,** acionadas pelos bits de execução provindos do controle. **Registrador de Retorno (RET REG)** para guardar o endereço posterior a um **CALL** e necessário para execução de um **RET e** Por fim, ainda iremos adicionar um bloco que realiza a operação “**A + B -1**” para realizar a soma do offset da ação de **JUMP.**

Mas como esse registrador irá funcionar, imaginemos que na memória de instrução existe a instrução para somar dois valores R[2] = R[1] + R[0]. Primeiramente, o registrador de instrução irá pegar essa instrução e a partir do opcode do barramento, irá saber que é uma função de soma. A partir disso, irá mandar os códigos de comandos específicos que iram para o banco de registradores, e o bit de seleção para a ULA especificando que é uma soma, como também o bit seleção do mux para salvar o resultado da soma novamente no banco de registradores. Ao final, você terá o resultado da soma esperada salva no registador escolhido.