Trabalho Final

Gabriella Almeida - 15528121

Kelvin Ribeiro Silva - 16302879

Nicolas Amaral dos Santos - 16304033

Introdução

Os módulos anteriores da prática de sistemas digitais introduziu princípios da lógica por trás das operações e funcionamento de um processador, como por exemplo o armazenamento de informações em memória, registradores e operações aritméticas.

O primeiro passo para entender-se o funcionamento de um processador é pensar no vocabulário do computador que é um conjunto de instruções. Até os softwares mais complexos são compilados em uma série de instruções simples como soma, subtração, jump. Essas instruções são cadeias de bits, 0's e 1's, que variam de tamanho de acordo com a arquitetura da arquitetura.

Neste presente projeto, utilizou-se uma arquitetura com uma palavra de 8 bits onde os 4 bits mais significativos definem a operação e os 4 bits menos significativos definem quais operandos serão utilizados por aquela operação.

Integração teoria e prática

No decorrer das aulas foi apresentado os conceitos necessários para a realização deste projeto. Conceitos como memória (latches, flip-flops, registradores), máquina de estados, operações aritméticas e por fim o processador em si, esses conceitos podem ser observados na construção desse processador.

Partindo do armazenamento das informações, os registradores são os componentes principais (depois da memória) para armazenar os dados temporários durante as operações, como a soma entre os registradores A e B, por exemplo. Eles são formados por flip-flops, que são circuitos de memória de 1 bit controlados por ciclos de clock. Cada flip-flop armazena 1 bit e, quando combinados, formam registradores maiores, como um registrador de 8 bits (composto por 8 flip-flops). Os flip-flops não podem ser alterados antes de um ciclo de clock, garantindo a sincronização e controle dos dados dentro da CPU.

A memória RAM (Memória de Acesso Aleatório) é um tipo de memória volátil usada para armazenar dados temporários enquanto o computador está em uso. É organizada em linhas e colunas, formando uma matriz. Neste processador, utilizamos a memória da própria FPGA (seguindo o que fora apresentado durante as aulas nos exercícios realizados), sendo uma memória de 256 palavras de 8 bits.

Outro componente que foi apresentado na disciplina são as máquinas de estado (FSM - Finite State Machine), essa implementação faz uso do modelo de Moore onde a saída depende somente do estado atual. Essa implementação está no Main Decoder que recebe a entrada (4 primeiros bits da instrução), e de acordo com essa entrada transita entre os estados (operações correspondentes àquela instrução) e realiza a operação solicitada.

O estado inicial da máquina é responsável por resetar/inicializar os registradores (reg_A, reg_B, reg_R, reg_IR, reg_PC) e aguardar as instruções. A instrução é buscada da memória e o registrador `reg_IR`

recebe a instrução da memória. Este é um estado onde a máquina obtém a próxima instrução a ser executada. A instrução no `reg_IR` é decodificada para identificar qual operação deve ser realizada. Os estados são: Operações da ULA, operações de memória (load ou store) onde a RAM é acessada, operações de comparação (cmp), saltos condicionais (jmp, jeq e jgr), estado de entrada (IN) e o estado de saída (OUT). Após a execução de uma instrução, a máquina retorna ao estado inicial ou de busca de uma nova instrução.

Componentes

ULA:

A Unidade Lógica e Aritmética (ULA) é um componente fundamental de um processador, responsável por realizar operações aritméticas, como adição e subtração, e lógicas, como AND, OR, NOT, ADD e SUB. A ULA interage diretamente com os registradores do processador, recebendo dados como entrada e armazenando os resultados.

Ao receber o sinal de controle (recebida do Main Decoder), a ULA executa a operação correspondente, utilizando dois valores armazenados nos registradores a e b com o resultado sendo armazenado no registrador de saída result. Além disso, a ULA também gera duas flags importantes:

Zero: que indica se o resultado da operação é igual a zero.

Sign: que sinaliza se ocorreu uma condição de "número negativo" em operações de subtração (quando b é maior que a).

MAIN DECODER:

O Main Decoder é responsável por interpretar as instruções de 8 bits vindas da memória e coordenar as operações do sistema. Os 4 bits mais significativos determinam a operação (como soma, subtração, armazenamento, movimentação entre registradores ou saltos), enquanto os 4 menos significativos especificam os operandos (registradores, memória ou valores imediatos). Ele gera sinais de controle para a ULA, define acessos à memória em operações LOAD e STORE e ajusta o contador de programa em instruções de salto. Assim, o Main Decoder gerencia a execução das instruções e o fluxo de dados no sistema.

COMPARADOR DE 8 BITS:

O comparador de 8 bits (cmp_8bit) é um componente utilizado para comparar os valores de dois registradores, reg1 e reg2. Ele gera dois sinais de saída:

- **zero**: indica se os valores comparados são iguais.
- **sign**: indica se o primeiro valor (reg1) é menor que o segundo (reg2).

Este componente é utilizado no **main decoder** para implementar as instruções de desvio condicional:

- JEQ (Jump if Equal): Quando o sinal **zero** é ativado (`zero_int = '1'`), o programa salta para o endereço especificado pela instrução.
- JGR (Jump if Greater): Quando o sinal **sign** não está ativado (`sign_int = '0'`), indicando que o primeiro registrador é maior, ocorre o salto para o endereço da instrução.

Essas funcionalidades permitem o controle do fluxo do programa com base nos resultados das comparações entre registradores.

MEMÓRIA RAM:

A RAM 256x8 permite leitura e escrita de dados de 8 bits em 256 endereços. Ela é controlada pelos sinais de leitura, escrita e habilitação, com todas as operações sincronizadas pelo sinal de clock. O componente ram256x8 gerencia o armazenamento e recuperação de dados na memória, enquanto o processo de controle do código assegura a correta operação de leitura e escrita conforme os sinais de controle.

Seu funcionamento:

- Quando a operação de leitura é solicitada (quando readsignal é `'1'` e writesignal é `'0'`), os dados da memória no endereço especificado por address_input são lidos e transferidos para dataBuf.

- Quando a operação de escrita é solicitada (quando readsignal é `'0'` e writesignal é `'1'`), os dados presentes em dataBuf são gravados na memória no endereço especificado por address_input.
- A memória só realiza essas operações quando memory_enable está em `'1'`. Se este sinal estiver em `'0'`, as operações são desabilitadas.

CPU:

Este módulo faz a junção de todos os componentes do processador, memória, main decoder, ULA e comparador de 8 bits. O módulo recebe como entrada os switches da FPGA e a instrução que se deseja realizar. Após a operação o resultado será exibido nos Leds da placa. É importante ressaltar sua dependência do clock e a possibilidade de reset.

Descrição das operações

Operações de ALU

1. **ADD (0000)**
    - **Função:** Soma o conteúdo de dois registradores.
    - **Como funciona:**
    - A instrução define dois registradores como entradas para a ALU (através dos bits menos significativos `instrucao(3 downto 0)`):
        - `reg_A_signal`: Registrador fonte 1.
        - `reg_B_signal`: Registrador fonte 2.
    - O controle da ALU (`ctrl`) é ajustado para `"010"` para executar uma soma.
    - Exemplos:
        - Caso `instrucao(3 downto 0) = "0001"`, soma `reg_A` com `reg_B`.
        - Caso `instrucao(3 downto 0) = "1010"`, soma `reg_R` com ele mesmo.
2. **SUB (0001)**
    - **Função:** Subtrai o conteúdo de dois registradores.
    - **Como funciona:**
    - Similar ao ADD, mas ajusta o controle da ALU (`ctrl`) para `"110"` (subtração).
    - Os registradores de entrada são definidos conforme os bits menos significativos.
    - Exemplos:
        - Caso `instrucao(3 downto 0) = "0100"`, subtrai `reg_A` de `reg_B`.
        - Caso `instrucao(3 downto 0) = "1011"`, subtrai `reg_IR` de `reg_R`.
3. **AND (0010)**

        - **Função:** Realiza uma operação lógica AND entre dois registradores.
        - **Como funciona:**
        - Define `ctrl = "000"` para operação lógica AND.
        - Seleciona os registradores de entrada conforme os bits menos significativos.
        - Exemplos:
            - Caso `instrucao(3 downto 0) = "0111"`, executa `reg_B` AND `reg_IR`.
            - Caso `instrucao(3 downto 0) = "1000"`, executa `reg_R` AND `reg_A`.
4. **`ADDI` (0100)**
    - **Função:** Soma o valor de um registrador com um valor imediato.
    - **Como funciona:**
        - Define `ctrl = "111"` para soma com valor imediato.
        - O registrador fonte é definido pelos bits menos significativos.
        - Exemplos:
            - Caso `instrucao(3 downto 0) = "0000"`, soma o valor imediato com o conteúdo de `reg_A`.
5. **`CMP` (0011)**
    - **Função:** Compara o conteúdo de dois registradores e atualiza as flags `zero` e `sign`.
    - **Como funciona:**
        - Seleciona dois registradores para comparação com base nos bits menos significativos.
        - A comparação é realizada pelo componente `cmp_8bit`.
        - Exemplos:
            - Caso `instrucao(3 downto 0) = "1001"`, compara `reg_B` com `reg_R`.

        - Caso `instrucao(3 downto 0) = "1101"`, compara `reg_B` com `reg_IR`.

6. **LOAD (0101)**
    - **Função:** Carrega dados da memória para um registrador.
    - **Como funciona:**
        - Ativa os sinais de leitura (`read_sig = '1'`) e habilitação (`enable_signal = '1'`).
        - O endereço de memória é definido conforme os bits menos significativos:
            - Endereço imediato no `reg_PC` (caso `instrucao(3 downto 0) = "0000"`).
            - Valor armazenado em um registrador (`reg_B`, `reg_R`, etc.).
        - Exemplos:
            - Carrega para `reg_A` o valor do endereço imediato.
            - Carrega para `reg_B` o valor armazenado no endereço apontado por `reg_A`.
7. **STORE (0110)**
    - **Função:** Armazena o conteúdo de um registrador na memória.
    - **Como funciona:**
        - Ativa os sinais de escrita (`write_sig = '1'`) e habilitação (`enable_signal = '1'`).
        - Define o endereço de memória e o dado a ser armazenado com base na instrução:
            - Caso `instrucao(3 downto 0) = "0000"`, armazena `reg_A` no endereço imediato.
            - Caso `instrucao(3 downto 0) = "1000"`, armazena `reg_R` no endereço armazenado em `reg_A`.
8. **MOV (0111)**
    - **Função:** Move o conteúdo de um registrador para outro.
    - **Como funciona:**

        - Seleciona o registrador fonte e o registrador destino com base nos bits menos significativos.
        - Exemplos:
            - Caso `instrucao(3 downto 0) = "0100"`, move `reg_A` para `reg_B`.
            - Caso `instrucao(3 downto 0) = "1011"`, move `reg_IR` para `reg_R`.

9. **JMP (1000)**
    - **Função:** Altera o valor do `reg_PC` (contador de programa) para realizar um salto.
    - **Como funciona:**
        - Define `reg_PC` com o valor de um registrador (ex.: `reg_A`, `reg_B`, etc.).
        - Exemplos:
            - Caso `instrucao(3 downto 0) = "0000"`, define `reg_PC = reg_A`.
            - Caso `instrucao(3 downto 0) = "0010"`, define `reg_PC = reg_R`.
10. **JEQ (1001)**
    - **Função:** Realiza um salto condicional se a flag `zero` estiver ativada (resultado de comparação igual a zero).
    - **Como funciona:**
        - Verifica a flag `zero_int`.
        - Caso `zero_int = '1'`, define `reg_PC` com um endereço imediato (`instrucao(3 downto 0)`).
11. **JGR (1010)**
    - **Função:** Realiza um salto condicional se a flag `sign` estiver desativada (valor positivo).
    - **Como funciona:**

        Verifica a flag `sign_int`.

Caso `sign_int = '0'`, define `reg_PC` com um endereço imediato (`instrucao(3 downto 0)`).

Considerações finais

O componente CPU é a entidade principal e conecta os módulos necessários. Contudo, apresenta os seguintes problemas:

Uso Incompleto da Memória RAM. O componente MemoriaRAM foi declarado, mas não é utilizado dentro do código. Gerou a ausência do processo que deveria carregar instruções da memória. Sem essa funcionalidade, a CPU não consegue buscar e executar instruções.

O main_dec implementa o controlador principal da CPU. Ele apresenta a decodificação das instruções e o controle dos registros e da ALU. Todas as instruções foram testadas no ModelSim e a maior parte delas funciona como deveria, mas alguns problemas foram identificados:

Controle de fluxo incompleto. Algumas instruções, como JMP e JEQ, não incluem uma lógica clara para tratar saltos imediatos ou endereços fora dos registradores, limitando a funcionalidade do controle de fluxo.

Novamente, integração com Memória RAM. Embora exista suporte para operações de load (LOAD) e armazenamento (STORE), a comunicação com a memória não é funcional.

Conexões internas não implementadas**.** Os sinais intermediários, como adress_signal e data_signal, usados para interagir com a memória, não são conectados adequadamente ao módulo principal (cpu.vhd).

Decodificação de Instruções. Embora a lógica de decodificação cubra diversas instruções (como ADD, SUB, MOV, entre outras), não há suporte para interrupções ou tratamento de erros, como instruções inválidas.

Com base nos problemas identificados, as principais razões para o não funcionamento da CPU são:

1. Falta de integração completa entre os módulos
    - As conexões entre os componentes principais (main_dec,MemoriaRAM, e ALU) são incompletas ou ausentes.

2. Ausência de Ciclo de Clock e Controle de Sequenciamento
    - Não há lógica implementada para controlar o fluxo de execução baseado no clock, fundamental para o funcionamento síncrono da CPU.
3. Inicialização inadequada
    - Sem uma lógica de reset funcional, os estados iniciais da CPU e de seus registradores são indefinidos.
4. Carregamento de Instruções
    - A memória RAM, essencial para o armazenamento e leitura das instruções, não é utilizada no fluxo de dados.

Conclusão:

O projeto apresenta uma estrutura básica funcional, com cada módulo atuando adequadamente de forma isolada nas simulações, mas devido a falta de integração e controle adequados entre os módulos, deixou o projeto um pouco não funcional.