Relatório - Arquitetura Microprocessada

Disciplina: Microcontroladores
2024.2

Equipe Responsável

Este projeto foi desenvolvido por estudantes do curso de **Engenharia de Computação - UFC**, sob a orientação do Prof. Dr. Eng. Nícolas de Araújo Moreira da disciplina de **Microcontroladores**.

- **Lucas de Oliveira Sobral**
  Matrícula: 556944
- **Carlos Vinícius dos Santos Mesquita**
  Matrícula: 558171
- **Caio Vinícius Pessoa Freires**
  Matrícula: 558169
- **Mateus Andrade Maia**
  Matrícula: 552593
- **Matheus Simão Sales**
  Matrícula: 555851
- **Herik Mario Muniz Rocha**
  Matrícula: 558167

1 Introdução

O projeto desenvolvido consiste na implementação de um processador simples utilizando Verilog, projetado para executar operações aritméticas e lógicas básicas, além de gerenciar registradores e flags. Todo o código está consolidado em um único arquivo chamado ProcessadorALL.v, que contém todos os módulos necessários para o funcionamento do sistema e o compilador está implementado no arquivo Compiler.cpp. Essa abordagem foi adotada devido à dificuldade encontrada em importar arquivos separados no ambiente de desenvolvimento utilizado.

Link do Repositório: Projeto no GitHub

2 Compilador

O compilador desenvolvido tem como objetivo converter instruções em linguagem de montagem (assembly) para um formato binário que pode ser utilizado diretamente por hardware ou máquinas virtuais. Este processo é baseado em um mapeamento pré-definido de comandos (mnemonics) e seus respectivos códigos binários (opcodes).

2.1 Mapeamento de Códigos

O compilador utiliza uma estrutura `map` para associar comandos aos seus respectivos opcodes e número de argumentos esperados. Por exemplo:

- Comando `ADD`: opcode `00000001`, espera 2 argumentos.
- Comando `SUB`: opcode `00000010`, espera 2 argumentos.
- Comando `NOT`: opcode `00000100`, espera 1 argumento.

2.2 Funcionamento do Compilador

O compilador lê um arquivo de entrada no formato `.asm`, processa cada linha, e escreve os comandos traduzidos em um arquivo binário no formato `.bin`. O fluxo de execução é detalhado a seguir:

1. **Validação de Entrada**: O programa verifica se os arquivos `.asm` (entrada) e `.bin` (saída) foram fornecidos corretamente e se podem ser abertos.
2. **Leitura do Arquivo `.asm`**:
   - Cada linha é lida e processada.
   - Comentários (após `;`) e espaços são removidos.
   - O comando é validado; se for inválido, um erro semântico é gerado.
3. **Processamento de Argumentos**:
   - Para cada comando, verifica-se o número de argumentos esperado.
   - Cada argumento é validado:
     - Deve ser binário (apenas `0` e `1`).
     - Deve ter o tamanho máximo permitido (8 bits, definido por `Word_size`).
   - Argumentos inválidos ou ausentes geram erros sintáticos ou de fim de arquivo (EOF).
4. **Escrita no Arquivo Binário**:
   - O opcode do comando é escrito no arquivo `.bin`.
   - Argumentos válidos são escritos em sequência.

2.3 Mensagens de Erro

O compilador trata e exibe mensagens detalhadas para facilitar a identificação de problemas. Alguns exemplos incluem:

- **Erro de Uso**: Indica que os arquivos de entrada ou saída não foram fornecidos corretamente.
- **Comando Inválido**: Quando um comando não está definido no mapeamento.
- **Erro Sintático**: Argumentos inválidos (não binários ou fora do tamanho permitido).
- **Fim de Arquivo Antecipado (EOF)**: O arquivo termina antes que todos os argumentos sejam processados.

2.4 Exemplo de Entrada e Saída

Considere o seguinte exemplo de entrada no arquivo `programa.asm`:

```
ADD
10101010
11001100
; Comentário: operação de soma

SUB
00001111
10101010
```

A saída gerada no arquivo `programa.bin` será:

```
00000001
10101010
11001100
00000010
00001111
10101010
```

2.5 Conclusão

Este compilador simples demonstra como transformar comandos assembly em códigos binários. Sua estrutura modular permite fácil expansão com novos comandos e maior robustez na validação de entradas.

3 Processador (linhas 3-132)

Este processador foi projetado para realizar operações aritméticas e lógicas básicas, além de implementar a funcionalidade `MOV`, que copia valores entre operandos diretamente no processador. O processador se comunica com uma **memória de instruções** externa, que armazena todas as instruções a serem executadas. A memória organiza cada instrução em registradores de 24 bits, divididos em `opcode` (8 bits) e `operand1` e `operand2` (8 bits cada).

Ele é composto por três principais componentes: **Controller**, **ULA (Unidade Lógica e Aritmética)**, e o módulo principal **Processor**.

Componentes do Processador

- **Controller:** O *Controller* é responsável por decodificar o `opcode` recebido e determinar a operação correspondente. Ele gera sinais de controle específicos para cada tipo de operação, como enviar comandos para a ULA ou realizar diretamente a operação `MOV`.
- **ULA (Unidade Lógica e Aritmética):** A ULA executa as operações aritméticas e lógicas, como soma, subtração, multiplicação, divisões, e operações lógicas (`AND`, `OR`, `XOR`). No entanto, a operação `MOV` é realizada exclusivamente pelo processador e não pela ULA.
- **Processor:** O módulo principal integra todos os componentes e coordena o fluxo de dados. Ele realiza as seguintes funções:
  - Lê as instruções armazenadas na memória.
  - Decodifica e executa cada instrução com base no `opcode`.
  - Coordena a comunicação com a ULA para operações aritméticas e lógicas.
  - Executa diretamente a operação `MOV`, copiando o valor de `operand1` para `result`.
  - Sincroniza o processamento com o sinal de *clock* (`clk`).
  - Exibe o resultado e as *flags* no console ao final de cada instrução.

Ciclo de Operação

O processador opera de forma síncrona, seguindo o ciclo descrito abaixo:

1. Durante o primeiro ciclo (`cycle_counter = 0`), o **instruction loader** lê uma instrução de 24 bits da memória de instruções e carrega o `opcode`, `operand1` e `operand2`.
2. Se a instrução requer o uso da ULA (por exemplo, operações aritméticas ou lógicas), o `Controller` envia o comando correspondente e a ULA processa os valores nos ciclos subsequentes (`cycle_counter < 3`).
3. Caso o `opcode` seja o de `MOV`, o processador executa a operação diretamente, copiando o valor de `operand1` para `result`, sem passar pela ULA.
4. Após a execução da instrução, o `result` e as *flags* são atualizados e exibidos no console.
5. O contador de ciclos (`cycle_counter`) é resetado para permitir a execução da próxima instrução.

Operações Suportadas

O processador suporta as seguintes operações:

- **Operações Aritméticas e Lógicas (ULA):**
- **Operação `MOV` (Processor):** A operação `MOV` copia diretamente o valor de `operand1` para `result`, sem realizar cálculos ou passar pela ULA.

Sinais de Saída

- `result`: Contém o resultado da operação executada. No caso de `MOV`, este valor será igual a `operand1`.
- `flags`: Indica o estado do processador após a execução da instrução. Cada bit possui um significado específico:
  - `Z` (Zero Flag): Indica se o resultado é zero.
  - `S` (Sign Flag): Indica o sinal do resultado (1 para negativo).
  - `C` (Carry Flag): Indica *carry* ou *borrow*, gerado em operações aritméticas.
  - `V` (Overflow Flag): Indica *overflow* aritmético.

4 Memória Externa (linhas 342-358)

Memória de Instruções

A **memória de instruções** é responsável por armazenar o código binário a ser executado pelo processador. Cada instrução é armazenada em uma linha de 24 bits, sendo organizada da seguinte forma:

- **8 bits para o `opcode`:** O `opcode` define a operação a ser realizada pelo processador. Ele é interpretado pelo **Controller** para gerar os sinais de controle necessários para a execução da instrução.
- **8 bits para `operand1`:** `operand1` contém o primeiro operando a ser utilizado na operação. Dependendo da operação, este operando pode ser manipulado pela ULA ou diretamente pelo processador (como no caso da operação `MOV`).
- **8 bits para `operand2`:** `operand2` contém o segundo operando, que será utilizado em operações que exigem dois operandos (como `ADD`, `SUB`, `MUL`, entre outras).

A cada ciclo de execução, o **instruction loader** é responsável por ler as instruções armazenadas na memória de instruções e enviá-las ao processador. O `opcode` é decodificado pelo **Controller**, que determina a operação a ser realizada, enquanto os operandos são fornecidos ao processador para o cálculo ou manipulação desejada. O processador então executa a operação, e o resultado é armazenado no registrador de `result`.

Durante a execução, a memória de instruções funciona como um repositório centralizado, garantindo que as instruções sejam carregadas e processadas sequencialmente pelo processador.

5 Leitura Arquivo binário (linhas 360-500)

O processo de leitura das instruções começa com a leitura do arquivo binário contendo o código de operação, operandos e o endereço de memória, que são interpretados e executados pelo processador. Essa funcionalidade é dividida em dois módulos principais: **InstructionLoader** e **TopLevel**, que trabalham em conjunto com a memória externa.

Módulo InstructionLoader

O **InstructionLoader** é responsável por ler as instruções do arquivo binário e prepará-las para o processador. As instruções são armazenadas em uma memória interna de 256 bytes. A leitura do arquivo binário ocorre no início da simulação, através do comando `$readmemb`, que carrega as instruções do arquivo `instructions.bin` para a memória.

Cada instrução possui 24 bits, organizados da seguinte forma:

- `opcode` (8 bits): Código da operação a ser executada.
- `operand1` (8 bits): Primeiro operando.
- `operand2` (8 bits): Segundo operando.

O módulo `InstructionLoader` opera em uma máquina de estados que realiza a leitura dos dados e os envia para o processador. A sequência de estados é a seguinte:

- **LOAD_OPCODE**: Carrega o `opcode` da memória.
- **LOAD_OPERAND1**: Carrega o primeiro operando da memória.
- **LOAD_OPERAND2**: Carrega o segundo operando da memória.
- **STORE**: Escreve os dados no banco de dados, para posteriormente serem utilizados pelo processador.
- **DONE**: Indica que todas as instruções foram processadas.

Módulo TopLevel

O módulo **TopLevel** é responsável pela integração entre o `InstructionLoader`, a memória externa e o processador. Ele garante que o fluxo de dados entre esses componentes ocorra de forma sincronizada, possibilitando a execução das instruções armazenadas na memória de instruções.

O `TopLevel` recebe o sinal de `reset` e `clk`, controlando o processo de leitura e execução das instruções. Quando o carregamento das instruções é concluído, o módulo também monitora o sinal de `done` para indicar que todas as instruções foram processadas e o processamento foi finalizado.

Dentro do módulo, a `InstructionMemory` armazena e fornece as instruções lidas, enquanto o `InstructionLoader` realiza o carregamento dessas instruções do arquivo binário para a memória. O `Processor` então executa as operações baseadas nas instruções carregadas.

A integração entre esses módulos permite a execução de uma sequência de instruções programadas, com um controle eficiente de início e término de cada ciclo de operação.

6 Unidade Lógica e Aritmética (ULA) (linhas 176-338)

A Unidade Lógica e Aritmética (ULA) é um componente fundamental de um processador, responsável pela execução de operações aritméticas e lógicas. Ela desempenha um papel crucial na execução de instruções, sendo projetada para realizar cálculos matemáticos, manipulação de bits e comparações lógicas.

Principais Funções da ULA

A ULA é projetada para executar as seguintes operações básicas:

- **Operações aritméticas:**
  - Soma (`ADD`): $C = A + B$
  - Subtração (`SUB`): $C = A - B$
  - Multiplicação (`MUL`): $C = A \times B$
  - Divisão (`DIV`): $C = \frac{A}{B}$
  - Módulo (`MOD`): Resto da divisão inteira
- **Operações lógicas:**
  - AND: $C = A \wedge B$
  - OR: $C = A \vee B$
  - XOR: $C = A \oplus B$
  - NOT: $C = \neg A$
  - NOR: $C = \neg(A \vee B)$
  - NAND: $C = \neg(A \wedge B)$
  - XNOR: $C = \neg(A \oplus B)$
  - MOD: $C = A \bmod B$
  - SLL: $C = A << 1$ (Deslocamento lógico para a esquerda)
  - SRL: $C = A >> 1$ (Deslocamento lógico para a direita)

Arquitetura da ULA

A ULA consiste em um conjunto de circuitos combinacionais que recebem como entrada:

- Até dois operandos (`operand1` e `operand2`);
- Um código de operação (`Opcode`) que define a operação a ser realizada;
- Sinais de controle para configurar a execução.

A saída da ULA é o resultado da operação solicitada, juntamente com **flags** de status que indicam condições especiais, como:

- **Zero Flag (Z):** Indica se o resultado é zero;
- **Carry Flag (C):** Indica um carry-out (transbordo) em operações aritméticas;
- **Overflow Flag (O):** Indica ocorrência de overflow;
- **Sign Flag (S):** Indica se o resultado é negativo.

Importância da ULA

A ULA é um dos componentes mais críticos de qualquer unidade de processamento. Sua eficiência e capacidade de executar operações complexas diretamente influenciam o desempenho geral do processador. Em arquiteturas modernas, a ULA também é projetada para suportar instruções vetoriais e paralelismo, permitindo um aumento significativo na performance em aplicações específicas.

7 Registrador (linhas 134-148)

O módulo `Register` implementa um registrador básico de 8 bits, utilizado para armazenar e reter dados entre ciclos de clock. Ele é projetado para operar de forma síncrona com o sinal de `clock` e oferece uma funcionalidade de `reset` para inicialização assíncrona. Abaixo, detalhamos o funcionamento e as características do registrador.

7.1 Entradas e Saídas

O módulo possui os seguintes sinais de entrada e saída:

- `clk`: Entrada de clock, utilizada para sincronizar as operações do registrador.
- `reset`: Entrada de reset assíncrono. Quando ativada, força o registrador a zerar (`data_out = 8'b0`).
- `data_in`: Entrada de 8 bits que contém o dado a ser armazenado no registrador.
- `data_out`: Saída de 8 bits que mantém o valor armazenado no registrador.

7.2 Funcionamento do Registrador

O comportamento do módulo é definido pelo bloco `always`, que responde a mudanças no sinal de clock (`posedge clk`) ou no sinal de reset (`posedge reset`). O funcionamento é descrito abaixo:

- Quando o sinal de `reset` está ativo (`1`), o registrador é imediatamente zerado (`data_out <= 8'b0`), independentemente do sinal de clock.
- Quando o `reset` está inativo (`0`), o registrador armazena o valor presente em `data_in` no flanco de subida do `clk`.

7.3 Vantagens e Aplicações

O registrador é um elemento fundamental nos sistemas digitais, com diversas aplicações, incluindo:

- **Armazenamento Temporário**: Manter dados intermediários em processadores e circuitos digitais.
- **Sincronização de Dados**: Garantir que os dados sejam processados em momentos específicos com base no sinal de clock.
- **Implementação de Pipelines**: Utilizado para dividir operações complexas em etapas menores e síncronas.

7.4 Conclusão

O módulo de registrador exemplifica a construção de um elemento de memória simples e eficiente em Verilog. Sua utilização permite a criação de circuitos digitais complexos, sendo indispensável em arquiteturas computacionais modernas.

8 Diagrama de blocos

O diagrama de blocos de um processador é uma representação visual que ilustra a organização e a interação dos principais componentes dentro de um processador. Ele serve para demonstrar como os dados fluem entre os diferentes módulos e como as operações são realizadas. Através desse diagrama, é possível visualizar as conexões entre a Unidade Lógica e Aritmética (ULA), registradores, barramentos de dados, memória, e os módulos de controle.

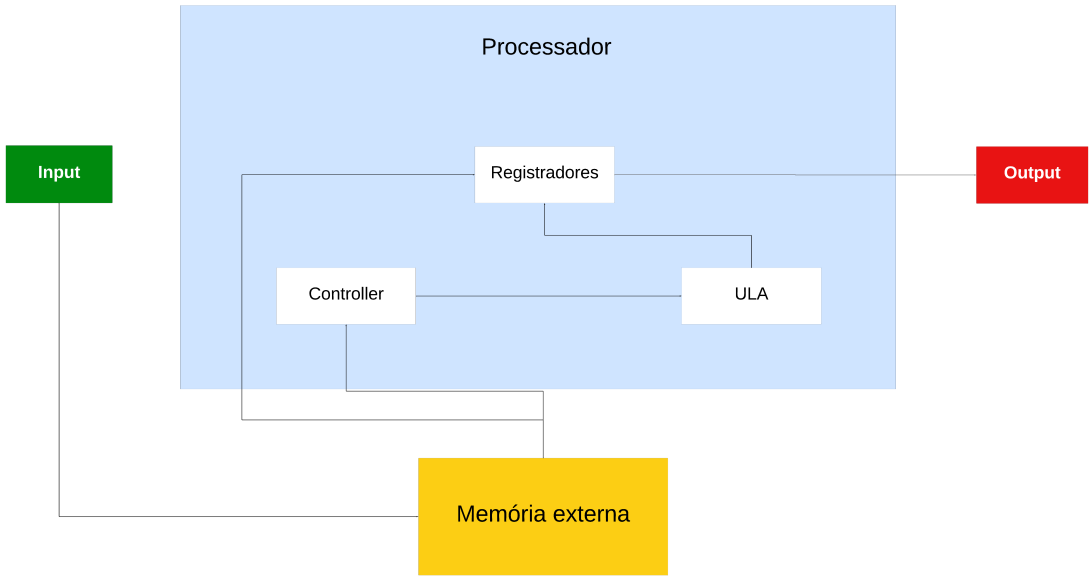

Figure 1: Enter Caption

9 Tabela de Opcodes

A tabela de opcodes é uma estrutura fundamental na arquitetura de processadores, que mapeia códigos binários (opcodes) para operações específicas executadas pelo processador. Cada opcode corresponde a uma operação aritmética, lógica ou de controle que o processador deve realizar. Essa tabela facilita a execução do código, pois traduz as instruções codificadas em um formato compreensível para o processador.

Opcode	Instrução	Operação	Tipo
0001	ADD	Soma operandos	Aritmética
0010	SUB	Subtração dos operandos	Aritmética
0011	MUL	Multiplicação dos operandos	Aritmética
0100	DIV	Divisão dos operandos	Aritmética
0101	MOD	Resto da divisão	Aritmética
0110	AND	Operação lógica "E"	Lógica
0111	OR	Operação lógica "Ou"	Lógica
1000	XOR	Operação lógica "Ou exclusivo"	Lógica
1001	NOT	Operação lógica "Negado"	Lógica
1010	NOR	Operação lógica "Ou negado"	Lógica
1011	NAND	Operação lógica "E negado"	Lógica
1100	XNOR	Operação lógica "Não-OU Exclusivo"	Lógica
1101	MOV	Transferência de dados	Lógica
1110	SLL	Deslocamento lógico à esquerda	Lógica
1111	SRL	Deslocamento lógico à direita	Lógica

Table 1: Tabela de opcodes da ULA com tipo de operação

10 Tabela dos Registradores

A tabela de registradores é uma estrutura essencial para entender como os dados são armazenados e manipulados dentro de um processador. Os registradores são pequenas áreas de armazenamento de alta velocidade localizadas dentro da Unidade de Processamento Central (CPU) e são usadas para armazenar dados temporários, instruções, endereços e resultados intermediários das operações.

Registrador	Sinal de Entrada	Descrição
regA	operand1	Armazena o primeiro operando utilizado nas operações da ULA.
regB	operand2	Armazena o segundo operando utilizado nas operações da ULA.
regC	ula_result	Armazena o resultado das operações realizadas pela ULA.

Table 2: Tabela de Registradores

11 Mapa de Memória e Endereçamento

Endereço (Hexadecimal)	Registrador/Conteúdo	Descrição
0x0000	RegA	Primeiro operando da ULA.
0x0001	RegB	Segundo operando da ULA.
0x0002	RegC	Resultado das operações da ULA.
0x0003	Opcode	Código da operação a ser executada.
0x0004	Flags	Armazena indicadores (Zero, Carry, Overflow, etc.).
0x0005 - 0x000F	RegX - RegY	Registradores de uso geral.
0x0100	Início do Programa	Início do código executável.
0x0101	Opcode da Instrução Atual	Código da operação a ser executada.
0x0102	Endereço do 1º Operando	Endereço do primeiro operando.
0x0103	Endereço do 2º Operando	Endereço do segundo operando.
0x0104 - 0x07FF	Instruções Subsequentes	Contém as instruções subsequentes do programa.

Table 3: Mapa de Memória com Registradores, Flags e Instruções

12 Considerações finais

O desenvolvimento deste projeto de arquitetura microprocessada, no contexto da disciplina de **Microcontroladores**, proporcionou uma experiência prática essencial para consolidar os conceitos teóricos

aprendidos em sala de aula. A implementação dos componentes, como registradores, unidade lógica e aritmética (ULA), e o sistema de controle, permitiu aos integrantes explorar a complexidade e a importância dos microprocessadores na computação moderna.

Além disso, a elaboração do relatório técnico promoveu a organização e a clareza na documentação de sistemas, uma habilidade indispensável para futuros engenheiros de computação. Durante o projeto, enfrentamos desafios relacionados à integração dos módulos e à validação do funcionamento correto das operações lógicas e aritméticas, o que nos estimulou a trabalhar em equipe e a buscar soluções eficientes.

Por fim, o projeto reforçou a importância do planejamento e da atenção aos detalhes no desenvolvimento de sistemas computacionais. A capacidade de mapear e solucionar problemas em cada etapa do processo é uma competência que será amplamente utilizada em nossa trajetória acadêmica e profissional. Estamos confiantes de que o aprendizado adquirido contribuirá significativamente para nossa formação como engenheiros.