UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Graduação em Sistemas de Informação

Alunos: Luiz Felipe Lima Costa, Marcos Antônio Dias Júnior e Rodrigo Luiz Macedo Ferreira.

TRABALHO PRÁTICO 2 - CAMINHO DE DADOS EM VERILOG

Relatório de Trabalho Prático apresentado à disciplina DCC 006 Organização de Computadores I, do Curso de graduação em Sistemas de Informação do Departamento de Ciência da Computação do Instituto de Ciências Exatas da UFMG.

Professor: Daniel Fernandes Macedo e Omar Paranaiba Vilela Neto

Belo Horizonte

2025

1. Introdução

Neste trabalho, o objetivo foi modificar um caminho de dados RISC-V de 5 estágios, implementado originalmente em Verilog, para incluir novas instruções e funcionalidades de modo a garantir a execução de operações mais complexas, como multiplicação, divisão, operações lógicas com XOR e desvio condicional. Essas instruções são essenciais para a execução de programas mais diversos e são frequentemente utilizadas em cálculos de alto desempenho, manipulação de dados e controle de fluxo em programas.

- **mul (multiplicação)**: Executa a multiplicação entre dois registradores.
- **div (divisão)**: Executa a divisão entre dois registradores - a não ser que o divisor seja zero (retornando zero como valor default nestes casos).
- **xori (XOR imediato)**: Realiza a operação XOR entre um registrador e um valor imediato.
- **beq (branch equal)**: Realiza um desvio condicional, se dois registradores forem iguais.

Para permitir a implementação das instruções foram feitas alterações no arquivo decode.v e execute.v.

2. Desenvolvimento

2.1 Implementação da instrução mul:

A instrução mul realiza a multiplicação entre dois registradores e armazena o resultado em um terceiro registrador. No RISC-V, esta instrução pertence ao conjunto de extensões M (Multiplicação/Divisão).

Modificações no caminho de dados:

- Arquivo decode.v: Adicionamos o opcode específico da instrução mul e configuramos os sinais de controle necessários para o caminho de dados.
- Arquivo execute.v: Incluímos uma unidade de multiplicação na etapa de execução. A unidade recebe os valores dos registradores Rs1 e Rs2, realiza a multiplicação e envia o resultado para o registrador de destino Rd.

Ajustes no controle:

- Foi necessário ajustar o sinal de controle da ALU para reconhecer a operação de multiplicação. Criamos um código específico para a instrução e garantimos que o controle ativasse a nova unidade de multiplicação apenas quando o opcode correspondente fosse identificado.

Justificativa das escolhas de projeto:

- A multiplicação é uma operação que pode ser realizada em um único ciclo, dependendo da implementação. Para simplificar o projeto, optamos por uma implementação direta que utiliza uma unidade de multiplicação dedicada.

2.2 Implementação da instrução div:

A instrução div realiza a divisão entre dois registradores e armazena o quociente no registrador de destino.

Modificações no caminho de dados:

- Arquivo decode.v: Foi adicionado o opcode específico da instrução div.
- Arquivo execute.v: Adicionamos uma unidade de divisão que realiza a operação entre os valores de Rs1 e Rs2.

Ajustes no controle:

- Assim como na multiplicação, criamos um código de operação específico para a divisão e ajustamos o controle da ALU para ativar a unidade de divisão quando o opcode correspondente fosse encontrado.
- Implementamos um mecanismo para lidar com divisões por zero. Nesses casos, o registrador de destino é configurado para armazenar um valor padrão (zero), evitando exceções.

Justificativa das escolhas de projeto:

- A divisão é uma operação mais complexa, podendo levar mais ciclos dependendo da abordagem. Como simplificação, utilizamos uma unidade de divisão que opera em um ciclo, com verificações para divisões por zero.

2.3 Implementação da instrução xori:

A instrução xori realiza uma operação lógica XOR entre um registrador e um valor imediato.

Modificações no caminho de dados:

- Arquivo decode.v: Adicionamos o opcode da instrução xori e ajustamos os sinais de controle para que a ALU reconhecesse a operação.
- Arquivo execute.v: A ALU foi modificada para incluir a lógica da operação XOR, utilizando o valor imediato fornecido.

Ajustes no controle:

- O controle da ALU foi ajustado para receber o valor imediato corretamente e realizar a operação XOR com o valor de Rs1.

- Implementamos a lógica para garantir que o valor imediato fosse corretamente sinalizado e estendido para 32 bits, conforme necessário.

Justificativa das escolhas de projeto:

- A instrução xori é importante para operações de manipulação de bits, e sua implementação exigiu poucas modificações no caminho de dados existente. Utilizamos a ALU para simplificar o design.

2.4 Implementação da instrução beq:

A instrução beq realiza um desvio condicional se dois registradores forem iguais. Para implementar a comparação de igualdade, optamos por utilizar o operador lógico XOR em vez de subtração. Essa escolha se baseia em benefícios de eficiência e simplicidade no design de hardware. O uso do XOR permite verificar rapidamente se dois valores são idênticos, uma vez que o resultado da operação também será zero quando ambos os registradores forem iguais. Essa abordagem reduz a complexidade da unidade de controle e melhora o desempenho, pois a operação XOR é computacionalmente menos dispendiosa do que a subtração em muitas arquiteturas. Se Rs1 for igual a Rs2 , o programa salta para o endereço especificado pelo rótulo. Além disso, a instrução bne realiza um desvio condicional se dois registros forem diferentes. Assim como na instrução beq, utilizamos a operação XOR para comparar os registradores. Se o resultado da operação XOR for diferente de zero, significa que os valores de Rs1 e Rs2 são diferentes, e o desvio é realizado. As outras instruções do tipo branch anteriormente implementadas não foram alteradas devido à natureza das operações.

Modificações no caminho de dados:

- Arquivo decode.v: Foi adicionado o opcode da instrução beq.
- Arquivo execute.v: Modificamos a lógica de controle de fluxo para verificar a condição Rs1 == Rs2. Se a condição for verdadeira, o program counter (PC) é atualizado para o endereço do rótulo.

Ajustes no controle:

- Foi necessário implementar um comparador que verifica se os valores de Rs1 e Rs2 são iguais.
- Ajustamos o controle de fluxo para garantir que, em caso de igualdade, o PC fosse alterado corretamente.

Justificativa das escolhas de projeto:

- O desvio condicional é fundamental para o controle de fluxo em programas. Escolhemos uma abordagem simples que utiliza um comparador dedicado para verificar a igualdade entre os registradores, garantindo que o salto ocorra corretamente.

Resumo das Modificações

Instrução	Arquivo(s) Modificado(s)	Ajustes Realizados
mul	decode.v, execute.v	Unidade de multiplicação, ajuste da ALU
div	decode.v, execute.v	Unidade de divisão, ajuste da ALU, verificação de divisão por zero
xori	decode.v, execute.v	Ajuste da ALU para operação XOR com valor imediato
beq	decode.v, execute.v	Comparador de igualdade, controle de fluxo para desvio condicional

3. Testes

Para garantir o correto funcionamento das novas instruções implementadas no caminho de dados RISC-V, realizamos uma série de testes em assembly, verificando as operações em diferentes cenários. A seguir, detalhamos os testes realizados, os casos abordados e os resultados esperados.

3.1 Teste das instruções mul, div e xori:

Neste teste, avaliamos o comportamento combinado das instruções de multiplicação, divisão e XOR imediato. O objetivo foi validar as operações aritméticas básicas e a manipulação lógica de bits**.**

Assembly:

```
addi x8, x0, 2          // x8 = 2
addi x9, x0, 3          // x9 = 3
mul x10, x8, x9         // x10 = x8 * x9 = 2 * 3 = 6
div x10, x10, x9        // x10 = x10 / x9 = 6 / 3 = 2
xori x11, x10, 63 // x11 = x10 ^ 63 = 2 ^ 63
```

Dump gerado: 0x00200413 0x00300493 0x02940533 0x02954533 0x03F54593

Resultado esperado:
Registrador x10 armazena o valor 2 após as operações de multiplicação e divisão.
Registrador x11 armazena o valor resultante da operação XOR entre 2 e 63, que é 61.

3.2 Teste da instrução mul operando com 0:

Este teste foi projetado para verificar o comportamento da instrução mul em um cenário com um dos operandos igual a zero. Operações envolvendo zero são casos limites importantes para testar.

```
li x8, 0          // x8 = 0
li x9, 3          // x9 = 3
mul x10, x8, x9   // x10 = x8 * x9 = 0 * 3 = 0
```

Dump gerado: 0x00000413 0x00300493 0x02940533

Resultado esperado: Registrador x10 armazena o valor 0, pois qualquer número multiplicado por zero resulta em zero.

3.3 Teste da instrução beq:

Este teste valida o comportamento da instrução de desvio condicional beq. Verificamos se o processador salta corretamente para o endereço especificado quando os valores dos registradores comparados são iguais.

Assembly:

```
addi x5, x0, 5    // x5 = 5
addi x6, x0, 5    // x6 = 5
beq x5, x6, 12    // Salta para PC + 12 se x5 == x6
addi x10, x0, 0   // x10 = 0 (não executado devido ao salto)
jal x0, 8         // Salta para PC + 8
addi x10, x0, 1   // x10 = 1
ecall             // Chamada de sistema
```

Dump gerado: 0x00500293 0x00500313 0x00628663 0x00000513 0x0080006F 0x00100513 0x00000073

Resultado esperado:
Quando os valores de x5 e x6 são iguais, o processador salta para PC + 12, ignorando a instrução addi x10, x0, 0.
A execução continua no endereço ajustado, seguindo o fluxo para a próxima instrução jal.
Registrador x10 armazena o valor 1, confirmando que o salto foi realizado corretamente.

4. Conclusão

Neste trabalho, foi realizada a modificação e expansão de um caminho de dados RISC-V de 5 estágios, implementado em Verilog, com o objetivo de adicionar novas instruções essenciais: mul, div, xori e beq. As modificações exigiram ajustes no design do processador para garantir que essas operações fossem corretamente executadas no pipeline, sem comprometer o funcionamento das instruções originais.

O principal desafio durante a implementação foi garantir a integração das novas instruções no pipeline do processador, respeitando a ordem de execução das operações e mantendo a consistência do controle de fluxo. As instruções mul e div exigiram a inclusão de unidades de execução especializadas, enquanto a instrução xori exigiu a adição de um módulo para manipulação de operações lógicas com valores imediatos. A instrução de desvio condicional beq foi integrada de forma a garantir a alteração do fluxo de controle quando as condições fossem atendidas.

A realização dos testes foi uma parte fundamental do processo, permitindo validar o funcionamento das novas instruções. Utilizando a plataforma Venus, conseguimos visualizar a execução do pipeline e verificar os resultados das operações, garantindo que o comportamento das instruções estivesse correto em diferentes cenários. A execução bem-sucedida dos testes, incluindo casos com multiplicação por zero e desvios condicionais, confirmou que as modificações no caminho de dados estavam funcionando conforme esperado.

Este trabalho proporcionou uma experiência prática valiosa na implementação de modificações em arquiteturas de processadores, além de reforçar a importância de testes rigorosos para a validação de sistemas complexos. O desafio de integrar novas operações em um processador de pipeline foi um aspecto central do trabalho, e os resultados alcançados indicam que as modificações feitas estão corretas e robustas.