INSTITUTO FEDERAL DO ESPÍRITO SANTO

ENGENHARIA DE CONTROLE E AUTOMAÇÃO

**LUCAS COELHO GARBOSSA**

**GUILHERME OLIVEIRA**

**IMPLEMENTAÇÃO EM HARDWARE DE INSTRUÇÃO DE PROCESSADOR ARM MONOCÍCLO**

SERRA

2025

LUCAS COELHO GARBOSSA

GUILHERME OLIVEIRA

**IMPLEMENTAÇÃO EM HARDWARE DE INSTRUÇÃO DE PROCESSADOR ARM MONOCÍCLO**

Projeto de desenvolvimento apresentado ao professor Rafael Emerick Z. De Oliveira como requisito para aprovação na disciplina de Arquitetura de Computadores.

SERRA 2025

**SUMÁRIO**

**1 Introdução..........................................................................................................7**

**2 2 Desenvolvimento............................................................................................8**

2.1 Testando o processador ARM Monocíclico......................................................8

2.1.1 Esquemático ARM monocíclico.....................................................................8

2.1.2 Simulação e preenchimento da Tabela 1......................................................10

**2.2 Modificando o processador ARM Monocíclico............................................13**

2.2.1 Função MOV.................................................................................................14

2.2.2 Função CMP..................................................................................................15

2.2.3 Função TST....................................................................................................16

2.2.4 Função EOR..................................................................................................17

**2.3 Testando as novas funções implementadas................................................17**

**3 Conclusão...........................................................................................................20**

**1 Introdução**

A Arquitetura de um hardware é definida pelo conjunto de instruções ou comandos que uma máquina deve saber obedecer e o conjunto de regras de manipulação dos operandos que esses comandos sabem utilizar e operar. Ou seja, ela é uma perspectiva do ponto de vista do programador do computador.

Para executar tais tarefas, é preciso um processador, que pode ser classificado como RISC (que trabalha com um conjunto de instruções reduzidas, sendo menor e mais rápido, ou seja, são projetados para eficiência e desempenho, além de ter um hardware mais simples e um software mais complexo) ou CISC (que tem instruções e hardware mais complexos, o que permite um software mais simples e menor).

Vistos esses aspectos e, devido às suas características, podemos classificar os processadores ARM Monocíclicos como RISC.

O presente relatório da disciplina de Arquitetura de Computadores, do Curso de Engenharia de Controle e Automação do Instituto Federal do Espírito Santo, tem como objetivo descrever a implementação em hardware das instruções MOV, CMP, TST e EOR através de ferramentas que permitam simulação e desenvolvimento de novas instruções em código HDL após entendimento do funcionamento da hierarquia e módulos internos de um processador ARM.

**2 Desenvolvimento**

**2.1 Testando o processador ARM Monocíclico**

Como o processador ARM monocíclico é classificado como RISC, já sabemos que ele contará com um software mais complexo quando pensarmos em desenvolver funções que seu hardware não pode executar por si só, já que é desenvolvido para ser mais simples, menor e mais eficiente.

Portanto, é imprescindível entender afundo o funcionamento de suas partes internas, incluindo sua hierarquia, como o Testbench, e seus módulos mais internos, presentes dentro do DUT (Device Under Test). O Testbench pode ser entendido como nosso ambiente de simulação, sendo capaz de testar e verificar o comportamento do DUT, para dessa forma, sabermos o que esperar da resposta e prever o que deve acontecer em cada ciclo executado no programa. Assim, poderemos garantir que tudo está funcionando conforme esperado. O DUT inclui todos os módulos internos do processador, como a ALU (unidade lógica aritmética), os registradores, flags, etc, que trabalham juntos para executar corretamente as instruções do processador.

**2.1.1 Esquemático ARM monocíclico**

Para entender todo esse funcionamento, foi desenvolvido um esquemático que representa todas as partes e módulos internos de um processador ARM Monocíclico. Ele se mostrou muito eficaz em facilitar o entendimento do funcionamento do processador, além de auxiliar na visualização dos pontos que deveriam ser alterados para a implementação das novas funções. Ele pode ser visto na Figura 1, Figura 2 e Figura 3 que aparecem logo abaixo. Veja como fica didático e compreensível o funcionamento de todos os componentes do processador, desde a unidade de controle até os registradores. É possível visualizar como todos os dados são tratados em cada módulo, sendo possível perceber o que é entrada, saída, etc em cada um deles. Por exemplo, na Figura 1, no módulo “arm”, não temos dificuldade em perceber que as entradas são “Instr” e “ReadData”, ambas de 32 bits, e as saídas são “MemWrite”, de 1 bit, e “WriteData”, “ALUResult” e “PC” de 32 bits. Assim podemos ver como cada um dos blocos tem funções definidas e como eles devem se conectar aos outros.

Diagrama, Esquemático

Descrição gerada automaticamente

Figura 1 - Esquemático processador ARM (parte 1)

Diagrama

Descrição gerada automaticamente

Figura 2 - Esquemático processador ARM (parte 2)

Diagrama, Esquemático

Descrição gerada automaticamente

Figura 3 - Esquemático processador ARM (parte 3)

**2.1.2 Simulação e preenchimento da Tabela 1**

Antes de tudo, foi necessário entender o significado de cada coluna da Tabela 1:

* **SrcA**: Esta coluna representa a fonte A, que é um dos operandos utilizados pela ALU (Unidade Lógica Aritmética) para realizar operações aritméticas e lógicas.
* **SrcB**: Esta coluna representa a fonte B, que é o segundo operando utilizado pela ALU para realizar operações aritméticas e lógicas.
* **ALUResult**: Esta coluna mostra o resultado da operação realizada pela ALU com os operandos SrcA e SrcB.
* **MemWrite**: Esta coluna indica se uma operação de escrita de memória está sendo realizada. Se for 1, a memória está sendo escrita; se for 0, não está sendo escrita.
* **Branch**: Esta coluna indica se uma instrução de desvio condicional está sendo executada. Se for 1, o desvio será realizado; se for 0, não será realizado.
* **ReadData**: Esta coluna mostra os dados que foram lidos da memória durante uma operação de leitura.
* **WriteData**: Esta coluna representa os dados que estão sendo escritos na memória durante uma operação de escrita.

Com os caminhos traçados no esquemático, uma visão mais macro de todo o processo e o entendimento do significado dos parâmetros de entrada e saída o preenchimento da tabela se tornou uma tarefa mais simples e dinâmica.

Então, foi feito o preenchimento da Tabela 1 de acordo com nossas previsões e de acordo com o significado de cada coluna.

Gráfico

Descrição gerada automaticamente com confiança média

Tabela 1 - Previsões de resultados da execução do código memfile.s

Preenchida a tabela, percebemos que a instrução final “STR”, escreverá o **valor 7** (em decimal) na **posição de memória 100** (em decimal).

Seguimos para a simulação no Modelsim (Figura 4). Para incluir os valores desejados, o esquemático das Figuras 1,2 e 3 ajudou bastante, pois com ele foi possível navegar dentro dos módulos do processador ARM no Modelsim com facilidade, visto que é possível identificar a localização de cada variável seguindo os blocos lógicos do desenho.

Assim, com todos os sinais adicionados na janela de ondas, foi possível fazer toda a conferência dos valores da Tabela 1 com a simulação, rolando a barra em amarelo, clock após clock, conferindo valor a valor e numerando cada ciclo na tabela para um clock diferente. Observe como o primeiro ciclo da tabela (1) tem os valores em concordância com a simulação:

* Scra = 8;
* Scrb = 8;
* AluResult = 0, e assim por diante.

Interface gráfica do usuário

Descrição gerada automaticamente

Figura 4 - Simulação no Modelsim

Agora, podemos confrontar nossa previsão em tabela com a simulação do processador. Através da Figura 5, visualizamos a última instrução (“str r2, [r0, #0x64]”, que significa: escreva o valor de r2 (7) na posição de memória r0+100 (100)), que nos mostra que a ALUResult indica o valor 100, WriteData indica 7 e MemWrite indica 1 (habilitando a função de escrita na memória). Tudo isso resulta no valor 7 sendo gravado na posição de memória 100, confirmando perfeitamente nossas previsões.

Tela de computador com jogo

Descrição gerada automaticamente

Figura 5 - Última instrução da simulação que confirma as previsões feitas em tabela

**2.2 Modificando o processador ARM Monocíclico**

Após total compreensão do processo, principalmente em nível de microarquitetura, foi possível iniciar o processo de desenvolvimento das novas instruções a serem processadas, dentre elas: MOV, CMP, TST e EOR.

Para isso, precisamos saber também a ordem de interpretação e o que cada bit representa em uma instrução. É possível ter essa informação através da Figura 6.

Diagrama

Descrição gerada automaticamente

Figura 6 - Codificação de instruções de processamento de dados de um processador ARM monocíclico

Além disso, consultaremos também a Tabela 2, para seguir o padrão de comandos da arquitetura arm.

Tabela

Descrição gerada automaticamente

Tabela 2 - Instruções de processamento de dados

**2.2.1 Função MOV**

A função MOV em assembly é usada para mover. Em um processador ARM, a instrução MOV copia o valor de um registrador ou um valor imediato (a depender de como a instrução será passada) para outro registrador. Por exemplo, MOV R1, R2 copia o valor do registrador R2 para o registrador R1.

Para implementar a função MOV em HDL (Hardware Description Language) no hardware de um processador ARM monocíclico, é necessário modificar a unidade de controle para reconhecer a instrução MOV e direcionar os sinais de controle adequados para os registradores. Basicamente, a unidade de controle deve gerar os sinais que permitem a leitura do valor do registrador de origem (ou imediato) e a escrita desse valor no registrador de destino.

Podemos dividir em três etapas:

* **Decodificação:** a unidade responsável pelo controle deve ser capaz de identificar quando uma função MOV deverá ser executada. Para isso, ele deve saber identificá-la através dos bits 20:27, que carregam os dados do Opcode. Nos preocuparemos mais especificamente com a secção funct da Figura 6, onde nos atentaremos para o I (registrador (0) ou imediato (1)) e o cmd (operação a ser executada, nesse caso, o MOV - 1101 de acordo com a Tabela 2 - );
* **Geração de sinais de controle:** com a unidade de controle sendo capaz de identificar a função MOV, devemos alterá-la para que consiga também gerar sinais de controle, ou seja, o “controller”, através dos seus submódulos “decoder” e “condlogic”, deverá gerar sinais de saída corretos (RegSrc, RegWrite, ImmSrc, etc – ver esquemático das Figuras 1, 2 e 3 - ) para o “datapath”;
* **Execução da Instrução:** no “datapath”, onde os sinais de controle são recebidos, os diversos submódulos, que compõem multiplexadores, registradores, etc (como “regfile”, “alu”, “mux2”, “adder”, etc) são direcionados para que as operações aritméticas e movimentação de dados sejam executadas corretamente.

Com as etapas bem definidas, podemos iniciar o desenvolvimento das novas funções. Para que o “controller” seja capaz de identificar a função MOV, criamos o “MovFlag” (1 bit). Com nosso esquemático bem estabelecido, ficou mais simples expandir esse bit para os locais que precisam identificá-lo para o correto funcionamento.

Como já dito e podendo ser verificado no esquemático, ela é passada como parâmetro inicialmente no bloco arm, sendo sinalizada em seguida para o controller, para o datapath, e vai sendo expandida de fora para dentro em todos os outros locais nos quais é necessária. Além disso, foi criada uma variável interna para “espelhar” o MovFlag, a “FlagM”, para servir como barramento interno e ser passada como parâmetro para o decoder e condlogic. Vale ressaltar que é importante seguir uma ordem de variáveis para passar o MovFlag, respeitando a regra de big ou little endian a qual o processador respeita.

Além disso, foi necessário criar mais um mux2, o “movmux” para passar os parâmetros da instrução MOV e fazer a indicação de escrita ou não, ou seja, se o MOV irá acontecer.

Na parte do decoder da ALU, foi implementada a parte capaz de interpretar o cmd referente ao MOV (1101). Também precisamos acrescentar a informação sobre nossa FlagM (que indica se o MOV irá acontecer ou não) em todos os outros casos, em estado baixo e em estado alto somente para o caso do MOV.

**2.2.2 Função CMP**

A função CMP em assembly é usada para comparar dois valores. Em um processador ARM, a instrução CMP compara o valor de um registrador com outro registrador ou um valor imediato (dependendo de como a instrução será passada). Por exemplo, CMP R1, R2 compara o valor do registrador R1 com o valor do registrador R2.

Da mesma forma que foi feito com MOV, para implementar a função CMP é necessário modificar a unidade de controle para que ela possa reconhece-la e direcionar os sinais de controle de forma adequada. Basicamente precisaremos que os valores dos registradores de origem sejam lidos e comparados na ALU, onde será executada uma subtração, que será tratada da seguinte forma: se o resultado for zero, os valores são iguais, se não, são diferentes. Dessa forma, dependeremos da flag “Z”, que será setada ou não em caso de resultado igual a zero.

Como o processador que temos já tem a função “sub” implementada, trabalharemos com os sinais “ALU control” de forma e ativar a subtração (b01) e não precisaremos modificar muitas coisas. Atentaremos principalmente em indicar a utilização e atualização da flag “Z”.

Assim, criaremos um caso no decoder para o CMP (cmd 1010, da mesma forma que foi feito com MOV), que dará sinal de controle para uma subtração, porém sem que esse valor seja escrito, pois aqui não importa o resultado e sim se os valores são iguais ou não, ou seja, precisaremos criar mais uma flag para indicar isso, no caso, “NoWrite”. Assim, no ALU decoder, indicaremos essa flag para todos os outros casos em estado baixo, ou seja, “escreva”, e no caso do CMP em estado alto, ou seja, “não escreva”. Ela deverá ser passada como parâmetro também nos outros módulos, conforme feito anteriormente, como o condlogic, condcheck, etc.

No condcheck (cc), havia uma lógica de sempre escrever, mas agora temos uma condição que não faz isso, então, devemos acrescentar uma lógica que não escreva caso eu sinalize. Assim, na linha “assign RegWrite = RegW & CondEX” colocamos “& ~NoWrite”, assim, ele não escreverá somente nos casos em que NoWrite = 1.

Não precisaremos indicar que “Z” deve ir para estado alto em caso de valores iguais, pois a função “sub” já fazia essa função.

**2.2.3 Função TST**

A função TST em assembly é usada para testar bits específicos de um valor. Em um processador ARM, a instrução TST realiza uma operação AND bit a bit entre dois registradores ou entre um registrador e um valor imediato, e atualiza as flags do processador com base no resultado. Por exemplo, TST R1, R2 realiza a operação AND entre os valores dos registradores R1 e R2, e atualiza as flags sem armazenar o resultado.

Da mesma forma que o CMP, o TST não fará escrita, então aproveitaremos a flag NoWrite para indicar isso ao criarmos mais um caso no decoder para executar o TST (cmd = 1000) quando solicitado, onde teremos um and sendo executado quando NoWrite for igua a 1.

**2.2.4 Função EOR**

A função EOR em assembly é usada para realizar uma operação XOR (Exclusive OR) bit a bit entre dois valores. Em um processador ARM, a instrução EOR realiza a operação XOR entre dois registradores ou entre um registrador e um valor imediato, e armazena o resultado em um registrador de destino. Por exemplo, EOR R1, R2, R3 realiza a operação XOR entre os valores dos registradores R2 e R3, e armazena o resultado no registrador R1.

Nesse caso, o cmd para o EOR será 0001, porém, para esse caso, a ALU de nosso processador ARM não executa um “ou exclusivo”, tendo somente soma, subtração, and e or, logo, precisaremos acrescentar uma linha que execute essa função. A operação original para o EOR é com um “^”, ou seja, Rd <- Rn ^ Src2.

Além disso, o ALUControl tem somente 2 bits para selecionar as operações desejadas, então, para acrescentar mais uma (EOR), deveremos acrescentar mais um bit, totalizando 3 bits, pois com 2 conseguimos selecionar apenas 4 possibilidades e agora teremos 5. Ou seja, teremos aqui uma “limitação física” e precisaremos alterar nosso hardware. Isso também implicará na mudanças em todos os pontos em que temos parâmetros relacionados a ALUControl, como nos módulos arm, controller, decoder, etc. Acrescentaremos um bit também em todos os casos já existentes e que criamos no ALU decoder até chegarmos ao ponto de incluir mais um caso para a nova função, o XOR.

**2.3 Testando as novas funções implementadas**

O passo a passo da implementação de todas as funções, pode ser acompanhado em diretório específico com todas as linhas modificadas e implementações no código versionados em github (**https://github.com/Guilherme227/armsinglecycle**).

Agora, com todas as funções implementadas, precisaremos implementar um novo arquivo de testes, o memfile2.dat para validarmos se nossas funções estão funcionando corretamente. Segue abaixo na Figura 7.

Tabela

O conteúdo gerado por IA pode estar incorreto.

Figura 7 - Arquivo memfile2.s

Para testarmos a função MOV, utilizamos a simulação do modelsim novamente, para demonstrar que MOV R1, #10 (e3a0100a) foi executado corretamente. Na próxima instrução podemos ver o comando MOV R6, R1 também funcionando corretamente. Veja Figura 8. Além disso, pode ser verificado o MovFlang com valor 1.

**Diagrama

O conteúdo gerado por IA pode estar incorreto.**

Figura 8 - Função MOV funcionando em simulação

Mais abaixo do código da Figura 7 implementamos o teste para a função CMP, que compara R1 com o valor 10 (CMP R1, #10 – e351000a - ). Se o CMP estiver funcionando, ele deve pular para linha MOV R1, #20 (e3a01014), através da instrução BEQ CMP\_OK (0a000000), já que pelo código R1 incialmente seria igual a 10. E isso acontece de formar correta, veja que após o CMP, a próxima instrução executada é exatamente a “e3a01014”. Veja na Figura 9.

Tela de computador com fundo verde

O conteúdo gerado por IA pode estar incorreto.

Figura 9 - Função CMP funcionando em simulação

Da função CMP\_ok já podemos seguir para testar o TST, na próxima linha, com instrução “e3110004” (TST R1, #4), que fará um and bit a bit entre R1, que recebeu 10100 (20 em decimal) anteriormente e 0100 (4 em decimal). O resultado deve ser 00100, veja na Figura 10 como o TST foi executado corretamente após visualizar a, b e result:

Interface gráfica do usuário

O conteúdo gerado por IA pode estar incorreto.

Figura 10 - Função TST funcionando em simulação

Através da próxima instrução (BNE TST\_OK), pulamos para a instrução MOV R1, #50 (e3a01032), movendo o valor 50 em decimal para R1. Na próxima linha testamos o EOR R1, R1, R2 (e0211002), onde, de acordo com as operações do código, temos R2 sendo igual a 7. Veja na Figura 11, o XOR sendo executado de forma correta, pois 50 em binário equivale a 110010 e 7 a 0111. Logo o resultado deve ser 110101.

Tela de computador com jogo

O conteúdo gerado por IA pode estar incorreto.

Figura 11 - Função EOR funcionando em simulação

**3 Conclusão**

Neste relatório, foi descrita a implementação em hardware das instruções MOV, CMP, TST e EOR em um processador ARM monocíclico. Através do desenvolvimento e simulação das novas funções, foi possível validar o funcionamento correto de cada uma delas, conforme demonstrado nas figuras e tabelas apresentadas ao longo do documento.

A implementação das instruções MOV, CMP, TST e EOR exigiu modificações na unidade de controle e nos sinais de controle do processador, além da criação de novas flags e multiplexadores para garantir a correta execução das operações. A simulação no Modelsim confirmou que todas as instruções foram implementadas com sucesso, atendendo aos requisitos estabelecidos.

Com isso, conclui-se que o processador ARM monocíclico foi capaz de executar as novas instruções de forma eficiente, demonstrando a viabilidade da implementação em hardware dessas operações. Este trabalho contribui para o entendimento e desenvolvimento de processadores RISC, destacando a importância da simulação e verificação no processo de design de hardware.