Organização de Computadores II
MIPS Pipeline - Relatório final

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais
2014/1

Grupo 4:
Gabriel Miranda
Rafael Luis Caldas
Junio Cezar
Leandro Noman
Pedro Thomas

Introdução:

Este relatório mostra como foi o desenvolvimento de todo o trabalho prático do MIPS pipeline. A explicação do trabalho foi dividida duas partes: a primeira, referente ao desenvolvimento de cada módulo de maneira individual, e a segunda, parte em que houve maior dificuldade, referente à etapa de integração de todos estes módulos. O foco do relatório é nas dificuldades encontradas pelo grupo, nas mudanças realizadas no decorrer do processo e nos motivos de tais mudanças. A maioria das modificações giraram em torno de problemas de sincronia do pipeline e serão apontados no decorrer do relatório.

Descrição do MIPS implementado:

Neste trabalho implementamos uma versão simplificada do MIPS pipeline visto, com detalhes, na disciplina de Organização de Computadores. Nesta simplificação, nós temos um total de 24 instruções reconhecidas, além da instrução NOP (sem operação). Sistemas de encaminhamento de dados não foram criados. Desta forma, para se evitar problemas de Hazard de Dados, é preciso esperar o WriteBack de instruções que podem gerar dependência. O pipeline possui 5 etapas, que são, em ordem de execução:

- *Fetch* - busca as instruções da memória;
- *Decode* - decodifica a instrução com o auxílio do Controle e envia dados e sinais às próximas etapas;
- *Execução* - faz toda a parte aritmética e de processamento de dados;
- *Memória* - realiza acessos à memória para escrita e leitura de dados;
- *WriteBack* - escreve o resultado final em algum registrador, se for o caso.

Desenvolvimento de módulos individuais:

- **Fetch:**

Este módulo corresponde ao primeiro estágio do pipeline e é responsável por enviar um sinal de permissão de leitura e o valor do Contador de Programa (PC) para o Controlador de Memória (CM). A implementação inicial não nos trouxe grandes problemas, tendo em vista que nós apenas precisávamos enviar essas duas informações ao CM e depois repassar a instrução recebida e o valor do próximo PC para o estágio de decodificação. Inicialmente, como não sabiamos que o nosso MIPS deveria reconhecer instruções do tipo NOP, nós ignorávamos as instruções deste tipo. Isso era necessário porque, devido a atrasos de sinais, o fetch sempre recebia um NOP como primeira instrução. Por isso, para não incrementarmos o PC de maneira incorreta, nós ignorávamos todos os NOP's, não incrementando o PC ao encontrar uma instrução deste tipo.

Como não conseguimos solucionar o problema do atraso, nós criamos uma flag neste módulo. Com isso, apenas o NOP devido ao atraso de sinais era ignorado e os próximos eram reconhecidos normalmente. Desta forma, mesmo que a primeira instrução de um programa MIPS fosse um NOP, ela seria reconhecida normalmente.

Outro problema foi encontrado neste módulo já na fase de integração do MIPS. Como poderíamos ter *Stall* devido ao acesso a memória por alguma outra instrução que precisasse ler ou escrever dados, o nosso fetch atrapalhava a instrução anterior, que já estava na fase de decodificação. Isso ocorria porque, quando tinhamos um *Stall*, o valor de PC não era alterado, e o registrador que guardava as instruções recebia 32 bits 0. Com isso, devido ao repasse rápido de valores, o decode recebia de maneira incorreta esses 32 bits 0 e realizava a decodificação da instrução anterior de maneira incorreta. Esse problema foi contornado adicionando-se um registrador auxiliar que mantinha a instrução anterior correta. Desta forma, quando um *Stall* é necessário, o registrador que guarda as instruções recebe primeiro o valor da instrução anterior correta, que estava guardado e, posteriormente, recebe os 32 bits 0, caso houvesse mais de um *Stall* em sequência.

Também tivemos outros problemas relacionados a sincronia e consistência de dados quando fomos unir todas as unidades para formar o MIPS. Sobre este problema, que afetou quase todos os módulos do pipeline, nós falaremos mais abaixo no tópico dedicado ao MIPS.

- **Decode:**

Essa unidade é responsável por receber uma instrução do Fetch e decodificá-la com o auxílio do módulo de Controle. O Controle, referenciado neste módulo, opera de maneira combinacional, pois não depende diretamente do clock para operar. Desta forma, no momento em que a instrução chega no Decode, os 6 bits iniciais e os 6 finais, opcode e funct, respectivamente, são repassados ao Controle que identifica qual é a instrução e manda para o Decode os sinais corretos para cada unidade do MIPS. Posteriormente, os dados decodificados e os sinais de controle são repassados para o próximo estágio do pipeline. Nós também não tivemos grandes dificuldades na criação deste módulo. Mesmo ele sendo maior e um pouco mais complexo que o estágio de Fetch, nós apenas tivemos problemas em saber se deveríamos atribuir valores às portas de saída por assinalamentos ou se deveríamos sincronizar a saída com o clock. Mesmo colocando as principais portas recebendo dados de forma síncrona, nós enfrentamos problema de sincronia com outras unidades. Este foi o mesmo erro encontrado no Fetch e será melhor detalhado posteriormente.

- **Execute:**

Aqui é onde são realizadas as operações lógicas e aritméticas do nosso MIPS. São referenciados, neste módulo, dois outros fundamentais para o funcionamento do nosso processador. Os módulos referenciados aqui são os que contém a ULA (Unidade Lógica Aritmética) e o Shifter, que realiza operações de deslocamento de bits. Uma verificação importante que ocorre neste estado é se haverá a necessidade de se realizar *Stall* na segunda instrução posterior a atual*. Esse *Stall* ocorrerá se a instrução atual necessitar realizar algum acesso a memória, seja para ler ou escrever dados. Há este problema, pois há apenas uma memória para armazenar dados e instruções. Desta forma, quando houver algum acesso a memória para transferência de dados, o Fetch não poderá acessar a memória para buscar uma instrução. Como foi dito no módulo do Fetch, o *Stall* inicialmente nos causava problemas, mas felizmente o problema foi solucionado.

Um problema que tivemos neste e em alguns outros módulos era o de atribuição proibida. Ele ocorria, porque nós criamos dois blocos *always* independentes que tinham o poder de alterar o valor dos mesmos registradores. Como estes blocos poderiam rodar em paralelo, haveria ali uma inconsistência de dados, pois não poderíamos garantir que o valor esperado seria repassado para os registradores. Um fato interessante é que nós desconhecíamos este problema e, quando realizávamos a compilação com o Icarus

Verilog, nós obtínhamos sucesso em nossos testes. Somente após a entrega parcial deste módulo é que descobrimos este erro com o auxílio do software Quartus.

Para solucionar este problema, nós criamos um único bloco *always* sensível aos dois sinais dos blocos anteriores. Quando entramos neste bloco, nós temos uma verificação *if-else* que nos mostra qual o valor correto devemos repassar aos registradores.

**Quando dizemos "atual", estamos nos referindo a instrução que se encontra naquele momento na fase de execução. O stall deve ocorrer nessa segunda instrução, porque no momento em que ela quiser fazer fetch, a memória já estará em uso para dados, que tem prioridade.*

- **Memory:**

Este módulo trata a leitura e escrita de dados na memória. São enviados para o Memory Controller os sinais que indicam se haverá uma leitura ou escrita na memória, o endereço e o dado, que será enviado ou recebido, dependendo da operação. Algumas instruções apenas passam por esse estágio, sem realizar nenhuma operação com a memória, por exemplo, instruções aritméticas e jumps.

Inicialmente, tivemos alguns problemas para trabalhar com portas do tipo *inout,* devido ao seu modo de assinalamento.

Neste módulo, também tivemos o problema da dupla atribuição com blocos independentes mostrado no módulo de execução. Para solucionar este problema, nós realizamos o mesmo processo descrito para aquele estágio.

Por fim, também tivemos problemas de sincronia neste estágio quando fomos integrar todo o MIPS. A técnica utilizada para resolver este problema será mostrada logo abaixo, na parte dedicada ao MIPS.

- **WriteBack:**

O módulo do WriteBack é bastante simples. Ele apenas repassa ao banco de registradores que dado deve ser escrito em qual registrador, além, é claro, do sinal que permite a escrita. Não houve significativas alterações dos módulos feitos para as entregas parciais e para a entrega final. Podemos destacar aqui apenas a inserção da saída de dados de forma assincrona no banco de registradores.

Neste módulo, felizmente não tivemos muita dificuldade e nem problemas.

- **MemController:**

O controlador da memória não é mostrado como um dos estágios do pipeline, mas é um dos mais importantes módulos implementados. É ele quem realiza a interface de comunicação entre os nossos módulos de Fetch e Memory com a memória RAM.

No início de seu desenvolvimento, nós tivemos muitos problemas por sua forma de trabalho. O módulo precisa de dois ciclos de *clock* para receber/passar os dados para/da memória. Isso ocorre porque o nosso MIPS trabalha com uma memória de 32 bits e nós teriamos que implementá-lo em uma FPGA que trabalha com palavras de 16 bits. Por conta disso, tivemos alguns problemas para entender como o controlador de memória devia ser feito. E, até consertá-lo por inteiro, às vezes o dado ia certo, às vezes não, às vezes os sinais de controle vinham errado, etc. Novamente, os problemas ocorriam por não alterarmos os registradores no tempo correto. Quando entendemos a ordem cronológica das alterações de valor de registradores, o MemController funcionou.

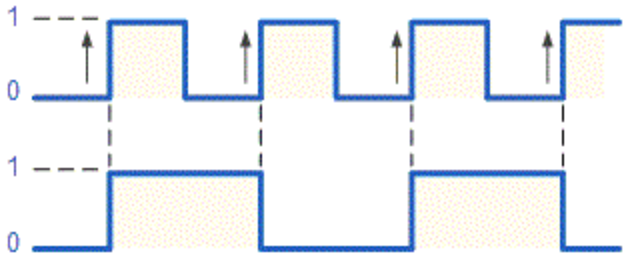
Para contornarmos o problema nós utilizamos uma flag, chamada *estado,* que nos permitia realizar operações distintas nos dois ciclos de *clock*. Com isso, nós conseguimos trabalhar corretamente com essa diferença dos 32 para os 16 bits.

Outro problema que tivemos que contornar nesse módulo foi a forma de endereçamento da memória. No MIPS, o endereçamento é feito por byte, o que nos mostrava que, para termos endereço válidos, eles deveriam ser múltiplos de 4. Por outro lado na FPGA o endereçamento era feito por palavra.

Para este problema, nós pegamos o endereço fornecido no formado do MIPS, dividimos o seu valor por 2 por meio de um deslocamento de 1 bit para a direita. Com isso, de acordo com o nosso estado definido pela nossa flag, nós acessávamos as posições consecutivas de 16 bits para termos os 32 bits pedidos pelos módulos do MIPS. Inicialmente, tivemos dificuldade para descobrir como faríamos essa conversão.

Desenvolvimento da segunda parte: União dos módulos no projeto do MIPS

O Módulo do MIPS foi um dos que foram desenvolvidos com mais facilidade, embora ele tenha a mais importante função, que é a integração das unidades. A maior dificuldade encontrada nele, foi a determinação de dois *clocks* distintos. O primeiro, que chamaremos de *clock* normal é o que nós poderiamos inserir de forma mecânica na FPGA. Este *clock* é repassado aos módulos de Memória e de Controle de Memória, pois, como foi dito acima, eles precisam de 2 ciclos de *clock* para completarem suas operações corretamente. O segundo é um *clock* mais lento, que opera com uma frequência 50% menor. Este clock é repassado aos demais módulos criados com o objetivo de se manter a sincronia. Uma representação destes *clocks* pode ser vista abaixo:



Para a criação deste *clock* mais lento, nós criamos alguns registradores. Um deles atuava como um contador de um bit, um outro recebia o valor desse contador e, por fim, o registrador do *clock* lento recebia esse último valor negado:

clock normal	contador	registrador result	clock lento (negado)
1	**1**	**1**	**0**
0	**1**	**1**	**0**
1	**0**	**0**	**1**
0	**0**	**0**	**1**

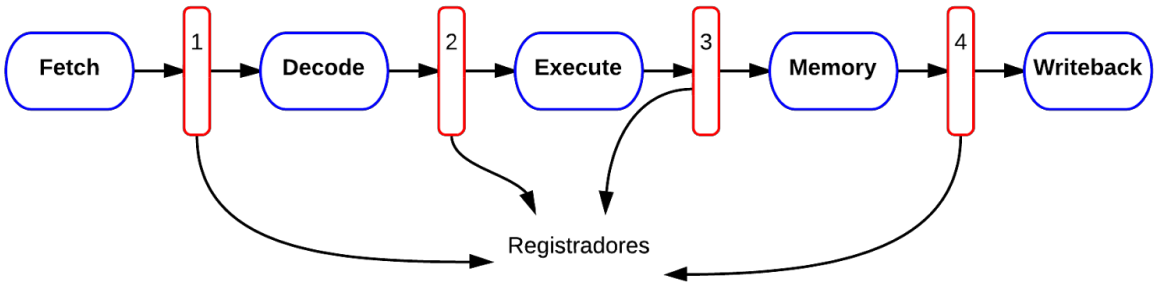
- **Problemas encontrados na Simulação:**

Como já citamos neste relatório, todas as unidades representadas no pipeline apresentaram problemas de sincronia e consistência de valores e sinais. Esse problema ocorria porque nós não isolamos corretamente cada módulo. Com isso, valores de uma instrução acabavam interferindo no funcionamento de outras. Um exemplo simples de problemas que enfrentamos é quando tínhamos duas operações consecutivas de soma.

Neste caso, constantemente a primeira instrução não escrevia no banco de registradores e a segunda instrução escrevia os dados calculados corretamente por ela no registrador de destino da primeira operação.

Para contornar esses tipos de problemas, nós criamos alguns registradores com o objetivo de guardar valores de maneira correta entre os ciclos.

Uma simples representação gráfica pode ser vista abaixo:



Desta forma, conseguimos manter uma sincronia correta que nos possibilitou testar uma variada quantidade de códigos no nosso simulador. Explicando de maneira simplista como esses registradores operam de acordo com os ciclos de *clock*:

CICLO 1: Fetch busca a instrução e a joga nos registradores ‘1’ junto do valor do próximo PC.

CICLO 2: Valores chegam ao Decode que, após realizar a decodificação, guarda os dados em ‘2’.

CICLO 3: Os dados saem de ‘2’, entram no Execute e, depois, os resultados vão para ‘3’.

CICLO 4: Caso seja um *STORE*, a instrução grava na memória e termina aqui, tendo nada para fazer no ciclo 5. *LOAD* pega os valores da memória e junta com outros dados, como sinais de controle e registradores de destino para escrita e repassa eles para ‘4’. Outras operações, como aritméticas apenas tem as informações repassadas para ‘4’.

CICLO 5: Os dados que chegam de ‘4’ são repassados diretamente para o banco de registradores. Caso a escrita seja autorizada, é verificado o dado e o registrador de destino e assim o dado é escrito.

- **Problemas encontrados na FPGA:**

Partindo dos resultados obtidos na simulação, nós acreditávamos que não teríamos grandes problemas na implementação do MIPS na FPGA. O primeiro problema encontrado pelo grupo foi o modelo da placa que tínhamos a disposição. O modelo DE2-115 que nos foi emprestado possuía uma memória diferente do modelo DE2 disponibilizado a outros grupos. Com a 115, nós não obtivemos sucesso no acesso à memória. Das poucas vezes que conseguimos realizar leituras, os valores obtidos nem sempre eram os esperados.

Com um pouco de persistência, tivemos acesso a uma FPGA do modelo DE2. Nela, o acesso a memória ocorria perfeitamente e todos as buscas de instruções obtiveram sucesso, inclusive instruções de NOP eram reconhecidas.

Porém, mais problemas vieram. Novamente, encontramos problemas de sincronia de sinais e dados. Algumas instruções executavam corretamente na FPGA, porém outras não. Com isso, como não tínhamos uma boa maneira de debugar sobre este problema na placa, nós fomos obrigados a retornar a nossa atenção para a Simulação. Pela primeira vez no trabalho decidimos utilizar gráficos de ondas a fim de verificar onde poderiam estar ocorrendo atrasos. Utilizamos os sistemas de WaveForms dos softwares ModelSim 10.1d e Icarus Verilog + GTKwave, porém em ambos os sinais eram repassados nos momentos corretos e as instruções executavam corretamente.

Com isso, tendo em vista o grande desgaste dos membros do grupo e o curto prazo que tínhamos para concluir o trabalho, decidimos por entregá-lo com o funcionamento limitado.

- **Lições aprendidas e comentários finais**

O trabalho foi muito interessante por nos permitir ter um contato prático com vários conceitos mostrados na disciplina de Organização de Computadores.

Também foi aprendido pelo grupo que nós não devemos encarar as linguagens de descrição de Hardware da mesma forma que encaramos linguagens de programação, como por exemplo o C. O fator paralelismo é um importante diferencial nessa área.

Algo fundamental observado foi a importância da ordem cronológica e do funcionamento em etapas de certos processos em Hardware. Como foi dito diversas vezes durante esse

relatório, quando errávamos na sincronia de algum sinal, alguma operação ou etapa era executada de maneira errada. Com isso, vimos outra parte importante do trabalho com hardware, que é a visualização geral do projeto como um todo enquanto se implementa os pequenos módulos. É primordial pensar na integração e no funcionamento conjunto para, assim, se evitar problemas.

Outro ponto importante no trabalho com hardware é que deve-se ter muita atenção, visto que, dependendo do projeto, a quantidade de sinais envolvidos pode ser grande demais, o que dificulta a correção de erros.