UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
SISTEMAS DIGITAIS

BananaCore

Processador em VHDL

Rogiel Sulzbach, Jefferson Johner, Matheus Oliveira

14 de Dezembro de 2015

Conteúdo

Capítulo 1

Introdução

O presente projeto versa sobre o processador BananaCore, desenvolvido como trabalho final da disciplina ENG04461 - Sistemas Digitais. Faz-se importante ressaltar que nosso processador foi desenvolvido com fins didáticos, não tentando priorizar desempenho ótimo.

Escrever capítulo de introdução

Capítulo 2

Especificação

A especificação inicial para o nosso projeto consiste em um processador de arquitetura do tipo Von Neumann[1] de 16 bits com instruções básicas do tipo:

- Operações de memória: Carregar dado da memória, armazenar dado na memória
- Operações aritméticas: adição, subtração, multiplicação e divisão
- Operações de IO: Write port e Read port
- Operações bit-a-bit: AND, NAND, OR, NOR, XOR e NOT

Como o objetivo do trabalho é desenvolver apenas um processador – não estamos interessados em como armazenar o programa no FPGA – escolhemos por gravar o programa de forma fixa; isto é, o programa é armazenado direto na memória RAM como um valor inicial.

Sabendo da limitação desta implementação, o design deve permitir que seja fácil substituir esta implementação inicial por outra mais funcional e completa.

[1] A arquitetura Von Neumann consiste em uma única memória compartilhada tanto para dados como para programas.

Capítulo 3

Implementação

3.1 Controlador de Memória

O controlador de memória é a unidade que faz o intermédio ao acesso a memória do processador.Nele, a operação de escrita é validada e então enviada para um respectivo memory bank (ver seção 3.1.3).

O controlador é operado utilizando 7 sinais:

Endereço (entrada) Recebe um valor de 16 bits que representa o endereço que memória que se deseja operar

Leitura de dados (saída) Retorna o valor equivalente ao byte armazenado no endereço solicitado ou alta impedância (caso esteja executando uma operação se escrita)

Escrita de dados (entrada) Recebe o byte que deseja-se escrever na memória

Operação (entrada) Indica a operação desejada na memória RAM (escrita ou leitura)

Pronto (saída) Uma flag que indica que a operação de leitura/escrita foi concluída com sucesso.

3.1.1 Leitura na memória RAM

Para ler um byte da memória RAM, primeiramente seleciona-se o endereço desejado e atribui-se este valor ao barramento de endereços. Em seguida, o bit de operação é definido em modo leitura e aguarda-se um número arbitrário de clocks até que o sinal de pronto transicione para nível alto. No

instante em que este sinal transiciona, o dado já está disponível na porta de leitura.

3.1.2 Escrita na memória RAM

Para escrever um byte na memória RAM, primeiramente seleciona-se o endereço desejado e atribui-se este valor ao barramento de endereços. Em seguida, no barramento de escrita de dados, o valor do byte é atribuido e o bit de operaçao é definido para uma operação do tipo *WRITE*. Devido ao fato da memória RAM possuir um clock diferenciado do processador central, é necessário aguardar um intervalo arbitrário de ciclos de clock até que o sinal de pronto seja colocado em alto. Neste momento, já é possível iniciar a escrita do próximo byte da mesma maneira.

3.1.3 Memory Bank

O memory bank é uma entidade simples e serve como uma abstração para um módulo de memória RAM genérico. O otimizador do Quartus II, ao detectar a presença de um grande bloco de dados, automaticamente executa uma otimização e substitui este por um bloco de memória RAM.

3.2 Controlador de Registrador

Para evitar que fosse necessário injetar um grande número de sinais de acesso a dados, controle e status dos registradores, uma abstração semelhante ao acesso à memória foi criada para simplificar este desenvolvimento. Detalhes desta implementação serão omitidos, pois são muito semelhantes ao barramento de memória.

3.3 Controlador de Instruções

O controlador de instruções é, sem dúvida, uma das partes com maior quantidade de código descrevendo hardware; isto se dá devido a uma dificuldade que encontramos ao implementar um barramento único de acesso à memória e aos registradores.

3.3.1 Decodificador de instruções

Detalhar a implementação atual

3.3.2 Executor de instruções

Cada instrução foi dividida em uma entidade chamada de *executor*. Esta entidade é responsável por fazer o carregamento de dados, execução da instrução e armazenamento do resultado final. Devido a repetição de código nestes executores, utilizamos geradores para gerar grande parte do código de forma automática e simples (ver seção 3.5).

3.3.3 Acesso à memória

Inicialmente, pretendíamos implementar o acesso global a memória por via de um barramento delimitado por *buffers tri-state*, contudo, esta implementação se mostrou muito complexa, pois ao incrementar as implementações de instruções do processador o barramento entrava em um estado inválido pois mais de dois sinais tentavam ser escritos no barramento em simultâneo. Acreditamos que estes problemas são devidos a falhas de design da arquitetura e que poderiam ser resolvidas escolhendo uma forma alternativa de implementação das instruções.

A solução desde problema, embora não seja ideal, foi simples: um grande MUX foi implementado de forma a fazer o "controle"de acesso ao barramento principal. Esta solução tem um grave problema: a necessidade de escrever código cresce muito em função da quantidade de instruções implementadas. Para um processador simples como o BananaCore isto pode não ser um problema muito relevante. Para implementações maiores, porém, isto pode ganhar uma faceta muito mais adversa. Como forma de solucionar, parcialmente, este empecilho, fizemos uso de artefatos de geração de código para gerar os muxes e demais condições do decodificador de instruções (ver seção 3.5).

Detalhar a implementação atual

3.4 Exemplo de funcionamento

Os bits correspondentes ao programa são definidos na memória inicial no endereço 0. Ao inicializar o processador, o **InstructionController** (seção 3.3) é responsável por carregar os primeiros 4 bytes de dados da memória. Após o carregamento inicial, o controlador é responsável por decodificar o opcode[1] e extrair cada argumento das instruções. Estes dados são salvos em um flip-flop para uso posterior.

[1] Opcode é o código de instrução, um código de 8 bits que dá "nome"a cada instrução do processador

No próximo estágio, o **InstructionExecutor** (seção 3.3.2) correspondente a instrução decodificada é ativado. Nele, os argumentos da instrução são passados e a execução da instrução começa. Cada instrução tem um comportamento diferente: instruções que dependem de dados registrados, primeiramente carregam seus registradores um a um. Instruções que armazenam a saída em algum registrador, fazem o armazenamento ao final da execução.

Ao final da execução da instrução, uma flag de pronto é levantada e o InstructionController continua com a execução da próxima instrução na memória (repetindo o processo especificado acima).

Para simplificar o entendimento do processador, utilizaremos um programa simples utilizado para os testes do processador. O programa faz o cálculo da operação de fatorial utilizando o seguinte programa:

Listing 3.1: Exemplo de código utilizado pelo Assembler

```
1  Assembler::Assembler(memoryStore)
2          .readIO(REGISTER_0)
3          .loadConstant(REGISTER_1, 0)
4          .loadConstant(REGISTER_2, 1)
5          .loadConstant(REGISTER_3, 1)
6
7          .mark(loopAddress)
8
9          .multiply(REGISTER_2, REGISTER_0)
10         .loadRegister(REGISTER_2, REGISTER_ACCUMULATOR)
11
12         .subtract(REGISTER_0, REGISTER_3)
13         .loadRegister(REGISTER_0, REGISTER_ACCUMULATOR)
14
15         .notEqual(REGISTER_0, REGISTER_1)
16         .jumpIfCarry(loopAddress)
17
18         .writeIO(REGISTER_2)
19
20         .jump(0);
```

O programa completo montado, em notação binária, é expresso abaixo:

```
00000011 00000000 00000000 00100001 00000000 00000000
00000000 00100010 00000000 00000001 00000000 00100011
00000000 00000001 00010010 00100000 00000000 00000010
```

```
00001110 00010001 00000011 00000000 00000000 00001110
00110101 00000001 00100010 00000000 00001110 00000010
00100000 00100000 00000000 00000000
```

No exemplo anterior, o *assembler* feito em C++ foi usado para gerar um código binário para o processador. Este é um algoritmo muito simples para cálculo de fatorial.

Primeiramente, o processador carrega os primeiros 4 bytes da memória; estes bytes correspondem a:

```
00000011 00000000 00000000 00100001
```

O decodificador extrai os seguintes parâmetros:

Opcode `00000011` (READ_IO)

Argumento 0 ~~`0000`~~`0000`

Nesta instrução (de acordo com a documentação, ver seção 5), os primeiros 4 bits do argumento 0 são ignorados. E os últimos 4 bits são correspondentes ao registrador de destino em que o conteúdo da porta externa deve ser copiado para. O decodificador também armazena o tamanho correspondente a instrução, neste caso, 2 bytes.

Após decodificado, o próximo estágio para para a ativação do executor da instrução, em que um bit interno é setado e a máquina de estados inteira da instrução inicia a execução.

Para esta instrução, o conteúdo da porta de entrada é primeiramente lido para um registrador interno e, no próximo ciclo e movido para o registrador 0000 de acordo com o argumento da instrução.

Ao final da execução, o registrador interno chamado **program counter** é incrementado o valor correspondente ao tamanho da instrução atual (isto é, pula para o endereço 2) onde a próxima instrução a ser executada se encontra.

O próximo ciclo de execução executa o seguinte bloco de memória:

```
00000000 00100001 00000000 00000000
```

O decodificador extrai os seguintes parâmetros:

Opcode `00000000` (LOAD)

Argumento 0 `0010`

Argumento 1 `0001`

Argumento 2 `00000000 00000000`

No argumento 0, o tipo de LOAD é especificado. De acordo com a documentação (ver seção 5), o tipo 0010 corresponde ao carregamento de uma constante. No argumento 1, o registrador de destino é especificado e no argumento 2 a constante de 16-bits a ser carregada é especificada.

Ao iniciar a execução da instrução, o executor imediatamente escreve no registrador `0001` o valor `00000000 00000000` e assim que completado, o controlador de instruções procede para a próxima instrução.

3.5 Geração de código

Para gerar os códigos repetitivos e forma algorítmica, fizemos usos de duas ferramentas distintas: Cog[2] e um script personalizado de geração de executores de instruções.

Mais detalhes da geração de código podem ser extraídas do código fonte do projeto disponível no GitHub.

3.6 Especificação final

A especificação final para o nosso projeto consiste em um processador de arquitetura do tipo Von Neumann[3] de 16 bits com instruções básicas do tipo:

- Operações de memória: Carregar dado da memória, armazenar dado na memória
- Operações aritméticas: adição, subtração, multiplicação e divisão
- Operações de IO: Write port e Read port
- Operações bit-a-bit: AND, NAND, OR, NOR, XOR e NOT

Escrever sobre os registradores

[2] Cog é um aplicativo que executa programas Python contidos em comentários do código fonte e substitui sua saída ao final da execução. O programa está disponível publicamente em `http://nedbatchelder.com/code/cog/`

[3] A arquitetura Von Neumann consiste em uma única memória compartilhada tanto para dados como para programas.

Capítulo 4

Testes

Escrever capítulo de testes

Capítulo 5

Código desenvolvido

Todo código VHDL, C++ e Python desenvolvido para este projeto está disponível livremente no GitHub.

BananaCore `https://github.com/Rogiel/BananaCore`

- **Documentação das instruções** `https://github.com/Rogiel/BananaCore/blob/master/Documentation/Instruction%20Set.md`

BananaVM `https://github.com/Rogiel/BananaVM`

- **Assembler** `https://github.com/Rogiel/BananaVM/tree/master/Source/BananaVM/Assembler`

Capítulo 6

Melhoramentos futuros

Tempos de espera redundantes

Na implementação atual, o processador gasta muitos ciclos de clock esperando por sinais internos. Muitas destas esperas podem ser removidas ou simplificadas.

Aglutinação de leituras consecutivas

Também, é possível aglutinar operações de forma simultânea. Por exemplo, é possível reduzir um ciclo de clock a cada leitura de 2 bytes consecutivos da memória ao iniciar a leitura do próximo imediatamente após a finalização da leitura anterior.

Acesso a memória em palavras maiores

Para melhorar a performance é possível aumentar o tamanho da palavra retornada pelo controlador de memória. Atualmente o controlador retorna palavras com 8 bits. Este limite pode ser aumentado (ou configurável) de forma a permitir que seja possível ler mais bytes em apenas uma solicitação.

Ineficiência do divisor

Após a análise de "slow-model"to TimeQuest, percebe-se claramente que o caminho crítico do processador está no divisor. Ele é responsável por um circuito síncrono, isto faz com que o clock do processador inteiro seja

delimitado pelo clock deste caminho crítico. Uma possível solução seria implementar pipeline no cálculo da divisão para segmentar este caminho crítico em múltiplos caminhos menores.

Controlador de clock interno

Se possível na placa de desenvolvimento, a criação de um controlador de clock interno permitiria que o clock fosse ajustado para velocidade mínima necessária no momento, bem como permitiria a redução (ou total desligamento) do clock do processador para economia de energia ou dissipação de calor.

Controlador de interrupções

Uma futura melhoria interessante é a adição de interrupções por timers e externas.

Capítulo 7

Conclusão

Escrever capítulo de conclusão

O processador é, dos componentes de um computador ou sistema digital, o componente mais caro e complexo. Trata-se de uma poderosa máquina de calcular.Tendo em vista sua função e complexidade, conclui-se que como projeto de conclusão de disciplina um processador é um elemento extremamente adequado, abordando grande parte dos conceitos estudados durante o semestre.