PCS3225 - Sistemas Digitais II - Trabalho 1

por Bruno de Carvalho Albertini

21/04/2021

Neste trabalho você irá implementar uma hierarquia de memória completa, com *cache* e memória RAM.

Introdução

As *caches* são a solução que os arquitetos de computadores encontraram para contornar o *memory wall* [1]. O problema é decorrente da evolução do desempenho: processadores evoluíram em velocidade muito mais rápido que as memórias. Nesse sentido, a grosso modo, os processadores são capazes de processar mais informações do que as memórias são capazes de entregar.

[1] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995

O tipo de memória mais rápido que conhecemos é a SRAM (*Static RAM*), cuja célula de memória é um *flip-flop*, implementado atualmente com 6-8 transistores por bit, sem levar em conta os circuitos de decodificação e saída. No entanto, esta memória é cara e complexa, além de ser volátil.

SRAM: cara e complexa, mas rápida.

Outra memória comum é a DRAM (*Dynamic RAM*), cuja célula de memória é um capacitor. Como é possível estrategicamente projetar um transistor para que a capacitância intrínseca em um dos terminais seja utilizável como célula de memória, cada célula é implementada exatamente com 1 transistor por bit, sem levar em conta os circuitos de decodificação e saída. Esta memória é mais barata que a SRAM, mais densa (menos transistores por bit, portanto mais bits por área), porém é mais lenta pois é necessário recarregar o capacitor antes que sua carga se esvaia, o que chamamos de *refresh*. Como esta memória é volátil também, ainda usamos mais um elemento na hierarquia de memória.

DRAM: custo médio, mas lenta.

Para o armazenamento não volátil, usamos atualmente discos magnéticos mecânicos (alta densidade e baixo custo), ou SSD (*Solid State Disk*), que são na verdade memórias *flash* (um tipo de memória EEPROM), com custo maior que os dos discos magnéticos mas consideravelmente menor que as DRAMs. Como o acesso à esses dispositivos é lento e passa por controladores de entrada e saída, normalmente o sistema operacional é que faz a interface entre a DRAM e o armazenamento não volátil. Por este motivo, neste trabalho focaremos nas interfaces processador-SRAM-DRAM.

Atividades

T1A1 Faça uma descrição em VHDL de uma memória RAM cuja entidade está na Figura 1.

Trabalho 1, Atividade 1, 10 envios, maior nota

```
entity ram is
  generic(
    address_size_in_bits : natural := 64;
    word_size_in_bits    : natural := 32;
    delay_in_clocks      : positive := 1
  );
  port(
    ck, enable, write_enable : in bit;
    addr : in    bit_vector(address_size_in_bits-1 downto 0);
    data : inout std_logic_vector(word_size_in_bits-1 downto 0);
    bsy  : out   bit
  );
end ram;
```

Figura 1: Listagem: Entidade para a memória do T1A1.

A memória é uma RAM, mas não é possível determinar qual tipo (SRAM ou DRAM) pois isso é definido na síntese, o que não faremos nesta disciplina. No entanto, esta memória deve ser projetada tendo em mente uma DRAM síncrona. Os parâmetros são:

SDRAM: *Synchronous* DRAM, não confundir com SRAM.

- `address_size_in_bits`: tamanho do barramento de endereços da memória, o que implica diretamente no número de palavras que esta memória é capaz de armazenar;
- `word_size_in_bits`: o tamanho de cada palavra de memória; e
- `delay_in_clocks`: o atraso para uma operação ser realizada (explicação detalhada no texto abaixo), em ciclos de *clock*.

A memória é síncrona, sensível à borda de subida do sinal de entrada `ck`. Quando o `enable` é baixo, esta memória não realiza operação alguma, o sinal de ocupado (*busy*) `bsy` é baixo e a saída `data` fica em *tri-state*. Já ao levantar o `enable`, na próxima borda de subida o sinal `bsy` vai para alto e a memória começa a operação de leitura ou escrita, dependendo se o `write_enable` é baixo ou alto, respectivamente.

Note que a porta `data` representa um barramento compartilhado para entrada e saída de dados, portanto deve ser um `inout`. Ainda, deve suportar *tri-state*, então usa-se o tipo `std_logic`. Para usar este tipo, inclua a biblioteca `ieee.std_logic_1164`.

Veja mais sobre a biblioteca 1164 aqui ou no livro texto.

Em relação à temporização, a memória não responde imediatamente. Qualquer operação demora n ciclos de *clock*, defindo pelo parâmetro citado anteriormente. Quando a memória termina a operação, abaixa o `bsy` e o dado estará disponível corretamente no barramento de dados (leitura) ou pode ser considerado escrito. O dado fica disponível por um ciclo de *clock* apenas pois, se mantiver o *enable* habilitado, a memória começará outra operação (de escrita ou leitura) na próxima borda de subida.

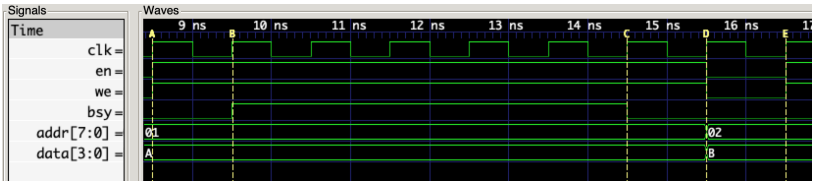

Figura 2: Ciclo completo de escrita na RAM T1A1.

Na Figura 2 podemos ver um ciclo completo de escrita para uma memória instanciada com atraso de 5 *clocks*. O endereço e o dado estão estáveis nas portas correspondentes da memória antes do marcador A, onde o sinal de *enable* e a escrita são habilitados. Após um ciclo de *clock*, em B, a memória levanta o *busy* e começa a operação. Extamente 5 ciclos de *clock* depois, a memória abaixa o *busy* em C, indicando que terminou a escrita. Antes da próxima borda, para evitar que a memória começe outra escrita, deve-se desabilitá-la, o que é feito abaixando o *enable* (e opcionalmente a escrita), o que pode ser visto em D. Note que outra operação de escrita (em outro endereço) pode ser iniciada, mas é necessário manter o endereço e o dado estáveis durante todo o ciclo de leitura ou escrita. Na figura, só depois de abaixar o *enable* é que um novo dado e endereço foi colocado no barramento, para iniciar uma nova operação, o que podemos ver a partir do marcador E.

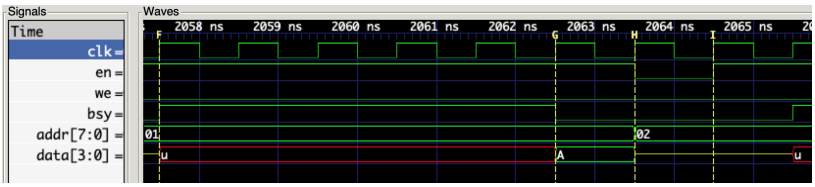

Figura 3: Ciclo completo de leitura na RAM T1A1.

Na Figura 3 podemos ver um ciclo completo de leitura para o mesmo endereço que foi escrito no exemplo anterior, na mesma memória. Note que antes de começar uma leitura, o usuario deve colocar o barramento em *tri-state* pois a memória irá escrever no barramento. Antes de F, a linha amarela indica que o barramento está em *tri-state*, ou seja, ninguém está escrevendo no barramento. Além disso, nota-se que o endereço já está estável no barramento antes de habilitar a memória para leitura, o que pode ser visto em F. De F até G a memória está realizando a operação (atraso de 5 clocks nesta instância), mas o valor do barramento é indefinido (valor `U` do `std_logic`). Somente quando a memória termina a operação é que o valor é válido, quando ela abaixa o *busy* em G. O valor permanece válido por um ciclo de *clock*, mas vai para *tri-state* em H pois a memória foi desabilitada neste ponto. O valor do endereço deve permanecer válido durante todo o ciclo. Entre H e I um novo endereço foi colocado, preparando para iniciar outra operação de leitura, o que podemos ver em I.

Para esta atividade, entregue uma memória conforme a entidade da Figura 1, e honre os parâmetros de tamanho e temporização con-

Na verdade o valor `U` significa *undefined* (não inicializado), mas usamos no exemplo para não confundir com o `X`, que significa *unknown* (desconhecido) mas confunde-se com o *don't care* que usamos nos mapas de Karnaugh. Se você entendeu a diferença, use o `X` pois está semanticamente correto. É comum os projetistas usarem `U` para evitar confusão.

forme exemplificado. Esta atividade vale 3/10 pontos neste trabalho.

3 pontos

T1A2 Faça uma descrição em VHDL de uma memória memória *cache* cuja entivade está na Figura 4.

Trabalho 1, Atividade 2, 10 envios, maior nota

```
entity cache is
  generic(
    address_size_in_bits: natural := 16;
    cache_size_in_bits: natural := 8;
    word_size_in_bits: natural := 8
  );
  port(
    ck, enable, write_enable: in bit;
    addr_i     : in  bit_vector(address_size_in_bits-1 downto 0);
    data_i     : in  bit_vector(word_size_in_bits-1 downto 0);
    data_o     : out bit_vector(word_size_in_bits-1 downto 0);
    bsy        : out bit;
    nl_data_i  : in  std_logic_vector(word_size_in_bits-1 downto 0);
    nl_enable, nl_write_enable: out bit;
    nl_bsy     : in bit
  );
end cache;
```

Figura 4: Listagem: Entidade para a memória do T1A2.

A memória cache é parametrizável, e os parâmetros são:

- `address_size_in_bits`: O tamanho do espaço de endereçamento do processador, em número de bits, ou seja, o tamanho do barramento de endereços. É esperado que este tamanho seja igual ao tamanho da memória RAM disponível. O valor padrão é 16, ou seja, o processador tem um barramento de endereços de 16 bits ou 64ki.

Semantica igual à memória.

- `cache_size_in_bits`: O tamanho da *cache* em bits. É esperado que este tamanho seja **menor** que o tamanho do espaço de endereçamento. O valor padrão é 8, ou seja, a *cache* pode armazenar $2^8 = 256$ palavras.

Note que o espaço de endereçamento, no valor padrão, é muito maior.

- `word_size_in_bits`: O tamanho da palavra de memória armazenada pela *cache*. É esperado que a palavra de memória tenha tamanho **idêntico** para o processador, para a *cache* e para a memória.

As entradas de controle são: `ck`: o *clock* da *cache* que pode ser diferente do processador e da memória, a *cache* também é síncrona e sensível à borda de subida; `enable`: quando baixo desabilita a *cache*, quando alto habilita; e *write_enable*: somente relevante se a *cache* estiver habilitada, quando baixo realiza uma leitura e quando alto realiza uma escrita.

A interface com o processador é feita pelos sinais: `addr_i`: o endereço da palavra que está sendo acessada (escrita ou leitura); `data_i`: a palavra de dado que vem do processador para a memória *cache*; `data_o`: a palavra de dado que sai da *cache* para o processador; e `bsy`: um sinal de ocupado (*busy*) que é alto se a *cache* está realizando

Tecnicamente o sinal de `bsy` (*busy*) faz parte do controle.

alguma operação e baixo se não está. Note que o processador não envia nem recebe palavras para a memória, mas sim para a *cache*, e o barramento de dados não é compartilhado (veja a Figura 5).

A interface da *cache* com a memória é feita por quatro sinais: `nl_data_i`: recebe os dados da memória; `nl_enable`: habilita a memória para que realize alguma operação; `nl_write_enable`: se baixo indica uma operação de leitura na memória e se alto uma operação de escrita na memória; e `nl_bsy`: um sinal de controle da memória para a *cache* com a mesma semântica do sinal `bsy`.

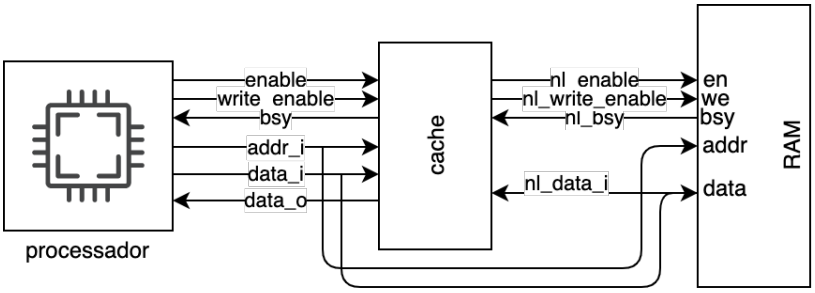

Figura 5: Hierarquia de memória para o T1A2.

Na Figura 5 pode-se observar todos os sinais da hierarquia de memória e suas ligações. O barramento de endereços `addr_i` é único e compartilhado entre todos, ou seja, quando o processador escreve o endereço, tanto a *cache* quanto a memória recebem-no pelo mesmo barramento.

Já o barramento de dados tem uma peculiaridade: é separado na interface do processador com a *cache* mas compartilhado na interface da *cache* com a memória. Note que a *cache* nunca escreve na memória. A *cache* de fato sinaliza para a memória que ela deve realizar uma escrita, mas o endereço vem diretamente do processador e o dado a ser escrito também. O processador nunca recebe dados da memória, sempre da *cache*. Podemos dizer então que o processador escreve dados no barramento compartilhado (escrevendo diretamente na *cache* e indiretamente na memória), mas lê dados somente da *cache*. Para facilitar o entendimento, vamos ver alguns exemplos.

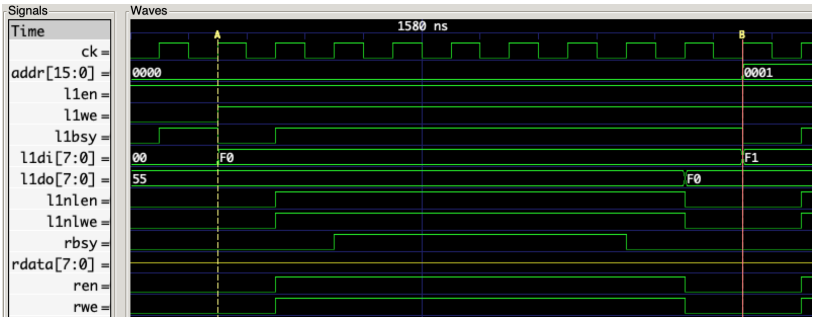

Figura 6: Escrita para o T1A2.

O primeiro deles é uma escrita na memória, que pode ser vista na Figura 6. O processador coloca o endereço e o dado no barramento e habilita a *cache* para escrita. A *cache* levanta o seu *busy* e sinaliza

Esta política de escrita em *cache* chama-se *write-through*, que é assunto de estudo de outra disciplina.

para a memória que é uma escrita. A memória também levanta o seu *busy* quando sinalizada. Tanto a *cache* quando a memória realizam a escrita neste momento, guardando a palavra internamente. Porém, a memória é muito mais lenta que a *cache*, portanto a *cache* deve esperar que a memória abaixe o *busy* para abaixá-lo também, indicando ao processador que a escrita terminou.

Note que o processador não sinaliza a memória diretamente.

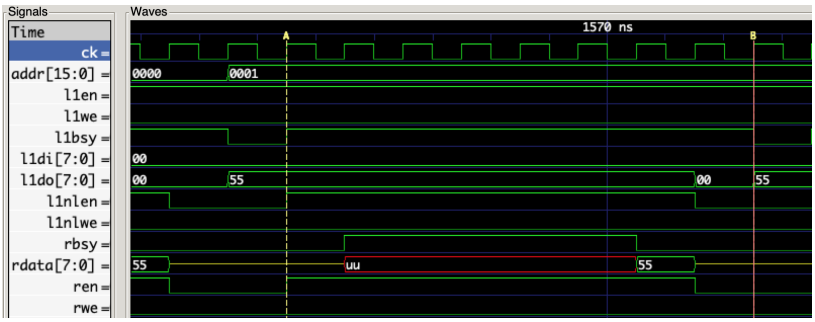

Figura 7: Leitura (*miss*) para o T1A2.

O segundo exemplo é uma leitura de um dado que não está copiado na *cache*, e pode ser visto na Figura 7. O processador coloca o endereço no barramento e habilita a *cache* para leitura. A *cache* percebe que não tem o dado armazenado e habilita a memória para leitura. Tanto a *cache* quando a memória leêm o endereço do barramento vindo do processador. A *cache* espera a memória responder com o dado armazenado, copia-o para sua memória interna e responde ao processador com o dado solicitado.

Esta é uma operação chamada de *cache miss*.

Note que o processador lê o dado da *cache* e não da memória.

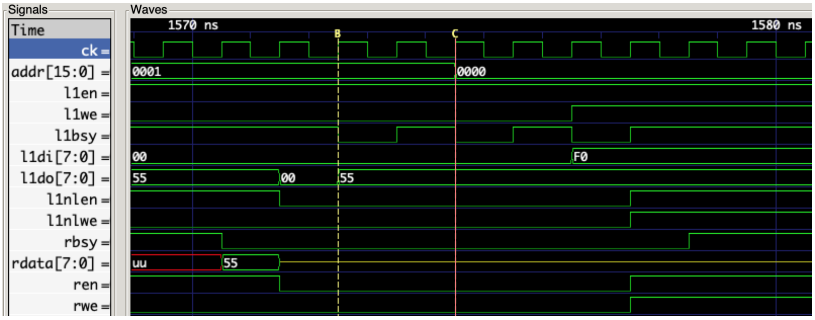

Figura 8: Leitura (*hit*) para o T1A2.

O último exemplo é uma leitura de um dado que já foi copiado na *cache* anteriormente, e pode ser visto na Figura 8. O processador coloca o endereço no barramento e habilita a *cache* para leitura. No entanto, neste caso a *cache* possui o dado armazenado internamente e retorna já no próximo ciclo de *clock*. Note que a memória não é habilitada em nenhum momento quando a *cache* já possui o dado armazenado.

Esta é uma operação chamada de *cache hit*.

Para esta atividade, entregue uma *cache* que implemente a entidade da Figura 4 e honre os parâmetros de temporização e tamanho exemplificados. Você pode considerar que o tamanho da *cache* será sempre menor que o tamanho do espaço de endereçamento e o tamanho da palavra de memória será idêntica para o processador, *cache* e

Os parâmtros podem ser diferentes do exemplo ou dos valores padrão.

memória. Esta atividade vale 7/10 pontos neste trabalho.

7 pontos

Armazenando dados na cache

Esta seção possui dicas não obrigatórias para seu projeto. Se você respeitar a temporização explicada anteriormente, seu projeto é válido independentemente de adotar as dicas a seguir.

A *cache* é uma memória pequena muito rápida com uma máquina de estados simples. As palavras de memória da *cache* na verdade são maiores que as palavras do sistema, pois a *cache* armazena algumas outras informações.

00	v	tag	data
01	v	tag	data
10	v	tag	data
11	v	tag	data

Figura 9: Exemplo de *cache* simples com quatro palavras (linhas).

Na Figura 9 podemos ver uma *cache* com quatro palavras. Suponha que esta *cache* está trabalhando com um processador cujo espaço de endereçamento tem 4 bits, ou seja, 16 palavras. A palavra `0000` é armazenada na linha de *cache* `00`, a `0001` na `01` e assim por diante. Podemos dizer que neste exemplo (com estes tamanhos de espaço de endereçamento e *cache*), os dois bits menos significativos do endereço indicam a posição da *cache* na qual o dado será copiado.

Quando o processador solicita a palavra de memória `0000` pela primeira vez, a *cache* guarda o dado vindo da memória na sua posição interna `00`. Se em algum momento após este acesso o processador solicitar o dado do endereço `0100`, a *cache* também guardará na posição `00`, sobrescrevendo a palavra que estava lá anteriormente.

Se você seguiu a política de escrita deste documento, não precisa se preocupar pois qualquer escrita feita pelo processador é realizada simultaneamente na *cache* e na memória, então qualquer dado alterado já está salvo na memória principal.

No entanto, a *cache* precisa saber se a palavra no seu endereço `00` corresponde ao endereço `0000` ou ao endereço `0100` (ou ainda ao `1000` ao `1100` que também são mapeados nessa linha da *cache*). Para isso, ela armazena a parte ambígua do endereço na própria linha da *cache*, em um espaço chamado *tag*. Neste exemplo, se a linha `00` contém uma cópia da posição de memória `0000`, o *tag* é `00`, mas se contém algo do endereço `0100`, o *tag* é `01`.

O campo *data* contém a cópia do dado e o campo *v* é um bit de validade, que indica para a *cache* que o conteúdo armazenado em *data* é válido. O bit de validade é importante pois ao ligar a *cache*, ela não tem nenhuma cópia em nenhuma linha, portanto precisa desta informação para não retornar lixo para o processador. Por exemplo: sem o campo *v*, é impossível saber se a palavra do endereço `0000` está na *cache* ou não.

Normalmente a *cache* é muito menor que o espaço de endereçamento, então o campo *tag* acaba armazenando mais bits que o dado em si. Este esquema de endereçamento de *cache* é chamado de *direct-mapped*, pois cada endereço do espaço de endereçamento é mapeado

Caso decida implementar outro esquema de endereçamento, não envie seu arquivo para o juiz e avise os professores.

em uma posição da *cache*. Este é o esquema mais simples que é usado na prática e sugerimos que atenha-se a ele. Você estudará outros esquemas de endereçamento de *cache* mais complexos em outras disciplinas.

Instruções para Entrega

Para cada atividade deste trabalho, há um *link* específico no ambiente da disciplina no e-Disciplinas. Acesse-o somente quando estiver confortável para enviar sua solução. Em cada atividade, você pode enviar apenas um único arquivo texto em UTF-8 contendo sua solução em VHDL. O nome do arquivo não importa, mas sim a descrição que está dentro. As entradas e saídas devem ser como as especificadas ou o juiz te atribuirá nota zero.

Se você usar somente caracteres ASCII, não faz diferença ser UTF-8 ou não pois as codificações são compatíveis.

Para este trabalho, o juiz verifica a temporização e não julgará seu trabalho caso não a respeite, atribuindo nota zero para a submissão mesmo que seu projeto esteja correto. Tenha certeza que suas duas soluções honram a temporização definida pela parametrização. Sugerimos que monte um *testbench* para cada atividade e analise com cuidado as formas de onda antes de enviar para o juiz.

Use o GTKWave, o ModelSIM ou o Scansion para visualizar a forma de onda. O parâmetro `--vcd` do GHDL habilita a geração da forma de onda.

Quando acessar o *link* no e-Disciplinas, o navegador abrirá uma janela para envio do arquivo. Selecione-o e envie para o juiz. Jamais recarregue a página de submissão pois seu navegador pode enviar o arquivo novamente, o que vai ser considerado pelo juiz como um novo envio e pode prejudicar sua nota final. Caso desista do envio, feche a janela antes do envio.

Depois do envio, a página carregará automaticamente o resultado do juiz, quando você poderá fechar a janela. Se não quiser esperar o resultado, feche a janela após o envio e verifique sua nota no e-Disciplinas posteriormente. A nota dada pelo juiz é somente para a submissão que acabou de fazer. Sua nota na atividade poderá ser vista no e-Disciplinas e pode diferir da nota dada pelo juiz dependendo da estratégia de atribuição de notas utilizada pelo professor que montou o problema.

Pode demorar alguns segundos até o juiz processar seu arquivo.

Atenção: não atualize a página de envio e não envie a partir de conexões instáveis (e.g. móveis) para evitar que seu arquivo chegue corrompido no juiz.

Referências

[1] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.