SIMULADOR DE CACHE E MEMÓRIA VIRTUAL

Trabalho prático final da disciplina GCC117 – Arquitetura de Computadores I – 2017/2 – Turmas 10AB e 22ª – Prof. André Vital Saúde

Implementar um simulador de memória cache e memória virtual, seguindo os requisitos descritos a seguir.

1 GRUPOS

O trabalho será realizado em grupos de no mínimo 4 e no máximo 5 pessoas. Se aparecerem pessoas sem grupo com este número de pessoas, a primeira coisa a ser feita será a tentativa de aloca-los em grupos que estejam apenas com 4 pessoas, mas se não for possível, serão retirados membros de grupos de 5 pessoas, aleatoriamente, para criar um novo grupo.

2 LINGUAGEM DE PROGRAMAÇÃO E APRESENTAÇÃO

O trabalho pode ser implementado em qualquer linguagem de programação de interesse, desde que atenda aos requisitos de teste e interação com usuário e de operações lógicas bit a bit (ver Seção 1.4 deste link https://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B.

3 ORGANIZAÇÃO E COMPONENTES DO SOFTWARE

Será implementada a hierarquia de memória nos moldes da arquitetura do Core i7, contendo:

1. cache L1 de dados, dentro do núcleo
2. cache L1 de instruções, dentro do núcleo
3. cache L2 de dados e instruções, dentro do núcleo
4. cache L3 de dados e instruções, compartilhado entre núcleos
5. endereçamento virtual e memória virtual

Ver Seção 2.5.4 do documento *Intel® 64 and IA-32 Architectures Optimization Reference Manual, Order Number: 248966-033, June 2016*, disponibilizado.

O software implementado deverá conter:

1. Módulo de cache totalmente associativo (32 bits)
2. Módulo de cache associativo por conjuntos (32 bits)
3. Hierarquia de cache
4. Memória principal com endereçamento virtual e swap
5. Hierarquia de memória com cache e memória principal
6. Processador multinúcleo
7. Leitura de um arquivo com uma sequência de comandos (para testes)
8. Relatório Final

Cada um dos componentes é descrito a seguir. **Os nomes de funções, métodos ou tipos de dados apresentados nas descrições, EXATAMENTE COMO ESTÃO ESCRITOS NESTE DOCUMENTO, são obrigatórios**.

3.1 CACHE TOTALMENTE ASSOCIATIVO

Implementar uma função ou método

```
TACache createTACache(int c, int l);
```

onde `TACache` é o tipo de dado (struct ou classe) que representa uma cache totalmente associativa com capacidade total de `c` bytes, sendo `l` bytes por linha. É obrigatório que `c` e `l` sejam inteiros potências de 2 e que `c` seja um múltiplo de `l` (testar a corretude da entrada e informar o erro, caso exista, é condição bônus).

Implementar também as seguintes funções ou métodos de acesso a informações sobre o `TACache`:

```
int getTACacheCapacity(TACache tac);

int getTACacheLineSize(TACache tac);
```

que retornam, respectivamente, a capacidade e o tamanho da linha da cache totalmente associativa `tac`.

Implementar a seguinte função ou método de acesso a dados da cache:

```
bool getTACacheData(TACache tac, int address, int * value);
```

que busca o valor do endereço `address` na cache totalmente associativa `tac`. O valor é retornado no parâmetro de saída `value` e o método ou função retorna `true`, se o endereço foi encontrado na cache (*hit*) ou `false`, senão (*miss*). A comparação do endereço `address` com o diretório do cache deve ser feita com o uso de operadores lógicos.

Implementar a seguinte função ou método de acesso a dados da cache:

```
void setTACacheLine(TACache tac, int address, int *line);
```

que escreve toda a linha `line`, que contém o endereço `address`, na cache totalmente associativa `tac`. O critério de seleção da linha do cache a ser gravada é de escolha do grupo, mas deve ser, no mínimo o critério FIFO discutido em aula. O tamanho da linha já é o tamanho definido na criação da `tac`.

Implementar a seguinte função ou método de acesso a dados da cache:

```
bool setTACacheData(TACache tac, int address, int value);
```

que sobrescreve o valor `value`, do endereço `address`, na cache totalmente associativa `tac`. Se `address` não está na cache, a função retorna `false`. Se `address` está na cache, `value` deve ser escrita na posição correta (offset) da linha da cache. Nota: todo endereço, imediatamente após escrito, deverá constar em todos os níveis de cache, logo, antes de se escrever um valor na cache toda a linha que contém seu endereço deve ter sido copiada para a cache, ou seja, escrever em um endereço que nunca foi acessado significa chamar `setTACacheData`, receber `false` como resposta, chamar `setTACacheLine` para trazer toda a linha de dados e depois chamar `setTACacheData` novamente para atualizar o valor.

3.2 CACHE ASSOCIATIVO POR CONJUNTOS

Implementar uma função ou método

```
SACache createSACache(int c, int a, int l);
```

onde `SACache` é o tipo de dado (struct ou classe) que representa uma cache associativa por conjuntos com capacidade total de `c` bytes, associatividade `a` e `l` bytes por linha. É obrigatório que `c`, `a` e `l` sejam inteiros potências de 2 e que `c` seja um múltiplo de `a*l` (testar a corretude da entrada e informar o erro, caso exista, é condição bônus).

Cada conjunto da `SACache` deve ser uma `TACache` e, portanto, `createSACache` deve chamar `createTACache`.

Implementar também as seguintes funções ou métodos de acesso a informações sobre o `SACache`:

```
int getSACacheCapacity(SACache sac);

int getSACacheLineSize(SACache sac);
```

que retornam, respectivamente, a capacidade e o tamanho da linha da cache associativa por conjuntos `sac`.

Implementar a seguinte função ou método de acesso a dados da cache:

```
bool getSACacheData(SACache sac, int address, int * value);
```

que busca o valor do endereço `address` na cache associativa por conjuntos `sac`. O valor é retornado no parâmetro de saída `value` e o método ou função retorna `true`, se o endereço foi encontrado na cache (*hit*) ou `false`, senão (*miss*). A verificação dos bits de *lookup* do endereço `address` para extrair o número do conjunto deve ser feita com o uso de operadores lógicos.

Implementar a seguinte função ou método de acesso a dados da cache:

```
void setSACacheLine(SACache tac, int address, int *line);
```

que escreve toda a linha `line`, que contém o endereço `address`, na cache associativa por conjuntos `sac`, usando `setTACacheLine` da `TACache`.

Implementar a seguinte função ou método de acesso a dados da cache:

```
bool setSACacheData(SACache sac, int address, int value);
```

que grava o valor `value`, do endereço `address`, na cache associativa por conjuntos `sac`, utilizando `setTACacheData` da `TACache`, e com o mesmo padrão de retorno de `setTACacheData`.

Implementar a função

```
SACache duplicateSACache(SACache sac);
```

que cria uma nova cache associativa por conjuntos com características idênticas a `sac`.

3.3 HIERARQUIA DE CACHE

Diferente do Core i7, deverá ser implementada uma cache inclusiva de três níveis no formato *write-through* (ver livro texto). A escrita de um dado significa escreve-lo nos três níveis.

Implementar uma função ou método

```
Cache createCache(SACache l1d, SACache l1i, SACache l2, SACache
l3,);
```

onde `Cache` é o tipo de dado (struct ou classe) que representa uma hierarquia de cache contendo uma cache L1 de dados `l1d`, uma cache L1 de instruções `l1i`, uma cache L2 de dados e instruções `l2` e uma cache L3 compartilhada de dados e instruções `l3`. Para simplificar a manutenção da coerência de cache, consideramos que só poderá ser criada a hierarquia de cache se os tamanhos de linha de `l1d` ou de `l1i` forem menores ou iguais ao tamanho de linha de `l2` e o tamanho de linha de `l2` for menor que o tamanho de linha de `l3`.

Implementar as seguintes funções ou métodos de atualização dos dados ou instruções da cache (ver também Seção 3.4):

```
void fetchCacheData(Cache sac, MainMemory mmem, int address, int value);

void fetchCacheInstruction(Cache sac, MainMemory mmem, int address, int
value);
```

que busca na memória principal `mmem` toda a linha de cache correspondente ao endereço `address` para cada nível da hierarquia de cache `c` e atualiza toda a cache. Nota: cada nível pode ter um tamanho de linha diferente e é preciso manter a coerência do cache inclusivo.

Implementar as seguintes funções ou métodos de acesso a dados ou instruções da cache:

```
int getCacheData(Cache c, MainMemory mmem, int address, int * value);

int getCacheInstruction(Cache c, MainMemory mmem, int address, int *
value);
```

que busca o valor do endereço `address` na hierarquia de cache `c`. O valor é retornado no parâmetro de saída `value` e o método ou função retorna `1`, se o endereço foi encontrado na cache L1 de dados; `2`, se o endereço foi encontrado na cache L2, `3`, se o endereço foi encontrado na cache L3 e `4`, se o dado não estava na cache (*miss*) e foi necessário atualizar a cache por meio das respectivas funções ou métodos `fetchCacheData` ou `fetchCacheInstruction`.

Implementar as seguintes funções ou métodos de acesso a dados ou instruções da cache:

```
void setCacheData(Cache c, int address, int value);

void setCacheInstruction(Cache c, int address, int value);
```

que grava o valor `value`, do endereço `address`, na hierarquia de cache `c`.

Ao final de uma leitura ou escrita, o endereço acessado deve ter sido copiado para o cache L1 e deve ser mantida a coerência do cache inclusivo.

Implementar a função

```
Cache duplicateCache(Cache c);
```

que cria uma nova hierarquia de cache na qual L1 e L2 são duplicados por `duplicateSACache` a partir de `c` e L3 é a mesma de `c` (compartilhada).

3.4 MEMÓRIA PRINCIPAL

Implementar uma função ou método

```
MainMemory createMainMemory(int ramsize, int vmsize);
```

onde `MainMemory` é o tipo de dado (struct ou classe) que representa a memória principal, composta de memória RAM, com `ramsize` bytes, e memória virtual, com `vmsize` bytes. Devido à dificuldade adicional detectada na implementação da hierarquia de cache, neste trabalho não iremos implementar a segmentação paginada da memória principal. Portanto, a `MainMemory` deverá ser implementada simplesmente como um único vetor de tamanho `ramsize` + `vmsize`.

Implementar a seguinte função ou método de acesso a dados ou instruções (a memória principal não separa dados de instruções) da memória principal:

```
int getMainMemoryData(MainMemory mem, int address, int * value);
```

que busca o valor do endereço `address` na memória principal `mem`. O valor é retornado no parâmetro de saída `value`. O método ou função retorna `4`, se o endereço é válido e foi lido corretamente; `-1`, se o endereço está fora da faixa de endereços virtuais válidos.

Implementar a seguinte função ou método de acesso a dados da memória principal:

```
void setMainMemoryData(MainMemory mem, int address, int value);
```

que grava o valor do endereço `address` na memória principal `mem`. O método ou função retorna `4`, se o endereço é válido e o dado foi escrito corretamente na memória; `-1`, se o endereço está fora da faixa de endereços virtuais válidos.

3.5 HIERARQUIA DE MEMÓRIA

Implementar uma função ou método

```
Memory createMemory(Cache c, MainMemory mem);
```

onde `Memory` é o tipo de dado (struct ou classe) que representa a hierarquia de memória, composta de uma hierarquia de cache `c` e uma memória principal `mem`.

Implementar as seguintes funções ou métodos de acesso a dados ou instruções da memória:

```
int getData(Memory mem, int address, int * value);

int getInstruction(Memory mem, int address, int * value);
```

que busca o valor do endereço `address` na hierarquia de memória `mem`. O valor é retornado no parâmetro de saída `value`. O método ou função retorna `1`, se o endereço foi lido em L1; `2`, se o endereço foi lido em L2; `3`, se o endereço foi lido em L3; `4`, se o endereço foi lido na memória principal; `-1`, se o endereço está fora da faixa de endereços virtuais válidos.

Implementar as seguintes funções ou métodos de acesso a dados ou instruções da memória:

```
void setData(Memory mem, int address, int value);

void setInstruction(Memory mem, int address, int value);
```

que grava o valor do endereço `address` na hierarquia de memória `mem`. A escrita deve respeitar os critérios de escrita da cache e da memória principal que compõem a hierarquia de memória `mem`.

Implementar a função

```
Memory duplicateMemory(Memory mem);
```

que cria uma nova hierarquia de memória na qual a hierarquia de cache é duplicada por `duplicateCache` (L1 e L2 duplicados e L3 compartilhado) e a memória principal é a mesma de `mem`.

3.6 PROCESSADOR MULTINÚCLEO

Implementar uma função ou método

```
Processor createProcessor(Memory mem, int ncores);
```

onde `Processor` é o tipo de dado (struct ou classe) que representa o processador, composta de `ncores` núcleos, todos usando a hierarquia de memória `mem` ou suas duplicações. **Lembre-se que as caches L1 e L2 são exclusivas de cada núcleo**. Portanto, o primeiro núcleo pode fazer referência exatamente a `mem`, mas cada núcleo adicional deve criar uma nova `Memory` com `duplicateMemory`.

3.7 LEITURA DE ARQUIVO DE COMANDOS

O programa criado deve ser inicializado passando-se como parâmetro um arquivo de comandos (ex: `meuprograma comandos.txt`). O programa irá ler e executar cada comando sequencialmente, **imprimindo na tela uma mensagem para cada comando executado**, com a informação do comando executado e o resultado obtido. Ao final, emitirá um relatório geral da execução de todos os comandos (ver próxima seção) e terminará.

Esta seção explica o formato do arquivo e os comandos que devem ser interpretados. A opção pelo arquivo de comandos permite que cada grupo utilize a linguagem de programação que desejar.

O arquivo é do tipo texto com um comando por linha.

Todos os comandos são no formato "*comando par1 par2*", podendo o número de parâmetros variar conforme o comando. Os comandos que devem ser implementados são listados a seguir. A verificação de corretude do arquivo é condição de bônus.

Comandos de criação da hierarquia de memória:

1. `cl1d c a l`. Cria uma variável `l1d` que é uma cache associativa por conjuntos com capacidade `c`, associatividade `a` e `l` bytes por linha.
2. `cl1i c a l`. Cria uma variável `l1i` que é uma cache associativa por conjuntos com capacidade `c`, associatividade `a` e `l` bytes por linha.
3. `cl2 c a l`. Cria uma variável `l2` que é uma cache associativa por conjuntos com capacidade `c`, associatividade `a` e `l` bytes por linha.
4. `cl3 c a l`. Cria uma variável `l3` que é uma cache associativa por conjuntos com capacidade `c`, associatividade `a` e `l` bytes por linha.
5. `cmp ramsize vmsize`. Cria uma variável `mp` que é uma memória principal com `ramsize` bytes de RAM e `vmsize` bytes de memória virtual.
6. `cmem`. Cria uma variável `mem` que é uma hierarquia de memória criada com `l1d`, `l1i`, `l2`, `l3` e `mp` já criados anteriormente.
7. `cp n`. Cria um processador com *n* núcleos, sendo que cada núcleo terá uma hierarquia de memória baseada em `mem`.

Todos os comandos de criação devem ser executados na ordem acima antes que qualquer outro comando possa ser executado.

Comandos de acesso a dados e instruções:

1. `ri n addr`. Lê a instrução de endereço `addr` na hierarquia de memória do núcleo `n`.
2. `wi n addr value`. Escreve a instrução `value` no endereço `addr` pela hierarquia de memória do núcleo `n`.
3. `rd n addr`. Lê o dado de endereço `addr` na hierarquia de memória do núcleo `n`.
4. `wd n addr value`. Escreve o dado `value` no endereço `addr` pela hierarquia de memória do núcleo `n`.
5. `asserti n addr level value`. Lê a instrução de endereço `addr` na hierarquia de memória do núcleo `n` e verifica se o valor foi lido do nível `level` (variando de 1 a 5 conforme retorno de `getInstruction`) e o valor lido é igual a `value`.
6. `assertd n addr level value`. Lê o dado de endereço `addr` na hierarquia de memória do núcleo `n` e verifica se o valor foi lido do nível `level` (variando de 1 a 5 conforme retorno de `getData`) e o valor lido é igual a `value`.

3.8 RELATÓRIO FINAL

O programa deve ler um arquivo com uma sequência de comandos de usuário e emitir um relatório final contendo as seguintes informações:

1. Descrição de toda a hierarquia de memória, com dados de capacidade, associatividade e tamanho de linhas de cada nível de cache, tamanho da memória RAM e da memória virtual, além do tamanho das páginas.
2. Número total de *hits* para cada nível (variando de 1 a 4 conforme retorno de `getData`).
3. Número total de erros (do tipo -1 conforme retorno de `getData`).

4 AVALIAÇÃO

O trabalho vale 30 pontos. Cada um dos itens da Seção 3 são pontuados individualmente.

Pontuação bônus será obtida nos seguintes casos:

1. Verificação de corretude dos parâmetros de createTACache
2. Verificação de corretude dos parâmetros de createSACache
3. Verificação de corretude dos comandos do arquivo de comandos