UNIVERSIDADE FEDERAL DO PARANÁ – UFPR

RAFAEL MORAIS DA CUNHA

PAGINAÇÃO NO XINU-CMIPS

CURITIBA
2017

RAFAEL MORAIS DA CUNHA

PAGINAÇÃO NO XINU-CMIPS

Trabalho de Graduação apresentado ao Curso de Graduação em Ciência da Computação, Setor de Ciências Exatas, Universidade Federal do Paraná, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Roberto André Hexsel

CURITIBA
2017

1. SUMÁRIO

1 INTRODUÇÃO

Dois assuntos importantes do curso de Ciência da Computação são o funcionamento de sistemas operacionais e em especial, de memória virtual. Para que os alunos consigam compreender melhor os conceitos apresentados, são geralmente ministradas aulas de laboratório onde utilizam de modelos para que visualizem o funcionamento desses sistemas.

Na UFPR, se tem usado o cMIPS[6], um modelo sintetizável de processador, nas aulas de arquitetura de computadores. Para as aulas de sistemas operacionais, foi feito o porte do sistema operacional XINU para o cMIPS[7], que foi depois validado[5]. Nas aulas de laboratório de memória virtual, o seu funcionamento é visto através de paginação com programas de teste escritos para o cMIPS[8].

O sistema operacional XINU é um sistema operacional feito para ser usado em sala de aula, no aprendizado de cursos de computação. O XINU usa threads como se fossem processos Unix. Portanto, qualquer parte da memória é acessível para qualquer processo que esteja rodando na máquina, e fica aberto o potencial de processos visualizarem ou modificarem dados não pertencentes à eles.

Apesar de já serem usados esses dois últimos materiais em aula, eles ficam de certa forma desconectados, porque o XINU não possui gerenciamento de memória virtual. Existia então a necessidade de unir o uso de paginação com o XINU. Dessa forma seria possível ter uma visão melhor do funcionamento de um sistema operacional moderno, e aumentam as possibilidades do que pode ser feito no modelo XINU cMIPS.

O objetivo deste trabalho é descrever a implementação de memória virtual com paginação no XINU cMIPS, e verificar o seu funcionamento no sistema operacional, fazendo então a união destes dois materiais, processador e sistema operacional, para uso didático.

Para implementar a paginação no cMIPS, foram usados como base os códigos do XINU cMIPS, e os códigos de teste usados em laboratório, fazendo as adaptações necessárias. Para verificar o funcionamento do sistema, foram usados os códigos de teste de memória virtual do cMIPS, todos rodando na nova versão do XINU cMIPS que foi desenvolvida pelo autor. Os testes foram todos executados através de um simulador de código VHDL. Os resultados foram observados pela

saída padrão do cMIPS e através do diagramas de tempo dos sinais produzidos pela simulação do cMIPS.

2 MEMÓRIA VIRTUAL

Em processadores antigos, a memória era diretamente acessada através do seu endereço físico por quaisquer processos. Isso tornava fácil que processos acessassem posições indevidas da memória, inclusive dados do próprio SO, caso houvesse um erro no processo, ou ele fosse propositalmente malicioso. Além disso, vários processos fazendo uso concorrente da memória ela acaba ficando fragmentada e pode tornar difícil a sua alocação. Para contornar esses problemas, surgiram os modelos de memória virtual[2].

Outro motivo para o uso de memória virtual é fazer com que o sistema operacional simule uma memória física para cada processo, podendo ela inclusive ter um tamanho maior que a memória física real. Isso facilita o funcionamento de processos que ocupem muita memória, no processo de salvar ou recuperar partes dele em memória secundária.

Em um processador com suporte à memória virtual, os processos acessam à memória usando um endereço virtual, que é mapeado para um endereço da memória física. A *memory management unit* (MMU), junto com o sistema operacional, fica responsável por traduzir os endereços virtuais usados pelos processos em endereços físicos quando é solicitado um acesso de memória pelo processador. A MMU também é responsável por verificar se o processo possui a permissão necessária para o uso da memória naquele endereço. Os dois principais modelos de memória virtual são os de segmentação e o de paginação.

2.1 SEGMENTAÇÃO

O modelo de memória virtual com segmentação é definido através da divisão da memória em segmentos [2]. Cada segmento é um bloco contínuo de memória, e pode ter permissões de leitura escrita e execução para que os processos só possam fazer o que foi definido para cada segmento. Cada segmento pode ter um tamanho diferente, por isso seu mapeamento deve ter um apontador para o início do segmento e um registrador com o tamanho do segmento.

No modelo de segmentação o endereço virtual é composto pelo identificador do segmento e o deslocamento a partir do início do segmento. Para fazer o

mapeamento dos endereços, a MMU faz uso de uma tabela de segmentos para calcular o endereço físico, verificar se o deslocamento é menor que o tamanho total do segmento e se o processo tem permissão para o uso requerido do segmento.

Tipicamente os processos são divididos em 3 segmentos, de código, de dados, e de pilha. Já no caso de processos que ocupam muita memória, ou mais complexos são usados mais segmentos. Outro modelo de divisão dos processos segmentação que também foi muito usado é a divisão de cada função ou objeto em um segmento diferente.

2.2 PAGINAÇÃO

Assim como no modelo de segmentação, o de paginação também divide a memória física em partes, mas aqui elas tem o mesmo tamanho, sempre potências de 2, e são chamadas de páginas. Cada página é uma bloco contínuo de memória, sempre iniciado num endereço múltiplo do tamanho das páginas. Por todas terem um tamanho igual pré definido, todos os bits do deslocamento são usados. Isso faz com que é possível ter um espaço de endereçamento virtual contínuo e contíguo, o tornando mais simples o uso de memória pelos programadores. Para restringir o acesso de cada página, elas também possuem as suas permissões de leitura, escrita e execução.

O mapeamento dos endereços virtuais em físicos é feito com o número da página e o deslocamento dentro da página. Os mapeamentos são guardados na tabela de páginas, onde a MMU busca o endereço da página física, verifica as suas permissões e concatena o número da página física com o deslocamento dentro da página.

Em ambos os modelos, é comum ter uma tabela de endereçamento individual para cada processo. Com isso, cada processo pode ter seu próprio espaço de endereçamento virtual, facilitando a alocação de grandes blocos contínuos de memória se comparado com um espaço de endereçamento compartilhado, que pode ficar mais fragmentado. Como existem vários processos executando ao mesmo tempo, a memória física, que é compartilhada entre todos os processos, a soma dos espaços de endereçamento virtuais dos processos pode ser maior que a memória física, e esta pode ser esgotada antes da memória virtual de um processo individual.

O esgotamento da memória virtual pode pode ser resolvido de forma mais prática que no caso do processador usar apenas um mapeamento direto, onde é usado o endereço físico, sem memória virtual. Geralmente apenas uma parte da memória física de um computador está sendo usada em um dado momento. Como apenas os segmentos ou páginas que estiverem sendo usados precisam ficar na memória física, é possível liberar mais memória física movendo o que não estiver sendo usado para a memória secundária.

Com um mapeamento direto, o endereço usado pelo programa é sempre o endereço físico, e ao movimentar uma página física para uma posição diferente todas as suas referências teriam de ser atualizadas. Já usando memória virtual, o endereço virtual é mapeado para um endereço físico diferente pela MMU. Com isso, os segmentos podem ser reposicionados dentro da memória, tendo apenas seu mapeamento atualizado, sem que os processos sintam a mudança feita pelo reposicionamento da página.

2.3 PAGINAÇÃO SOB DEMANDA

Para evitar que a memória física seja esgotada, um processo pode salvar as páginas ou segmentos que não estão sendo usados no momento em memória secundária e usar a memória principal para o que estiver sendo usado com mais frequência. Inicialmente isso era feito pelo programador, mas nos sistemas operacionais modernos é usada a paginação sob demanda. Esse é um recurso do SO que faz com que as páginas de memória principal que não estão sendo usadas sejam passadas para a memória secundária automaticamente quando não houver memória física disponível para uma página que estiver sendo acessada.

Isso permite que o programador escreva seus processos sem ter que remanejar explicitamente as páginas em memória principal para secundária quando a primeira estiver completamente alocada. Assim, cada processo fica limitado apenas pela própria tabela de páginas e cada processo tem a impressão de estar trabalhando com a memória virtual toda à sua disposição.

As rotinas paginação sob demanda são chamadas quando um processo tenta acessar uma página que foi salva na memória secundária, ou tenta usar uma nova página, mas não há espaço disponível na memória física. Nesse momento o sistema

operacional escolhe uma das páginas da memória principal, a salva na memória secundária e marca na tabela de páginas que ela está em disco, e a posição onde foi salva. O SO reserva uma parte da memória secundária para salvar as páginas. A partição do disco usada com esse fim é chamada de *swap* (área de trocas). Para a página que estava em memória secundária é feito o processo inverso, copiando do disco para a memória principal e marcando na tabela de páginas o endereço da memória física para onde ela foi copiada.

2.4 TLB

Para fazer as traduções de endereço virtual para físico, a MMU usa os dados das tabelas de mapeamento que ficam armazenadas na memória principal. Porém, adicionar um acesso à memória para cada leitura ou escrita (da memória de instruções ou de dados) tornaria a execução lenta. Para que a tradução possa ser feita de forma mais rápida é usado a Translation Lookaside Buffer (TLB). Ela funciona como um cache da tabela de páginas.

Quando um mapeamento se encontra na TLB, ocorre um acerto (TLB HIT), e o endereço virtual é traduzido para o físico no mesmo ciclo do processador. Quando o mapeamento não se encontra na TLB, ocorre uma falta (TLB MISS) e o sistema operacional toma controle do processador para que alguma das entradas da TLB seja preenchida com o mapeamento do endereço virtual faltante. Depois que o mapeamento é carregado na TLB, o SO devolve o controle do processador para o processo, para que ele continue como se não tivesse sido parado. Essas operações, escondidas pelo sistema operacional, permitem que o programador não se preocupe com o mapeamento da memória quando escreve seu código.

3 MIPS

O MIPS (microprocessor without interlocked pipeline stages) é uma arquitetura de processador que segue a filosofia RISC (reduced intructions set computer). O processador cMIPS (classical MIPS), que foi usado neste trabalho, é uma implementação em VHDL do conjunto de instruções MIPS32r2[6]. Ele é um processador pipeline de 5 estágios, que usa um coprocessador para controlar as interrupções e uso da TLB.

O cMIPS tem sido usado nas aulas de laboratório da UFPR para que os alunos consigam visualizar o funcionamento de um processador, e experimentar a programação em assembly para o MIPS. O modelo usado em aula e para este trabalho é simulado em um computador e está disponível no repositório Git [6]. Há também uma versão do cMIPS sintetizável, que pode ser usado numa FPGA Altera Cyclone IV.

3.1 COPROCESSADORES

O manual do MIPS32 [3] define 4 coprocessadores, chamados de CP0, CP1, CP2, e CP3.

O CP0 é responsável pelo uso de memória virtual, exceções, trocas entre os modos de usuário, supervisor e kernel, pelo subsistema de cache, pelos diagnósticos do processador, e pelos sistemas de tratamento de erro. O CP0 foi parcialmente implementado no cMIPS[6] permitindo o uso de memória virtual por paginação.

O CP1 é usado para fazer as operações de ponto flutuante. CP2 é reservado para implementações específicas do MIPS32. O CP3 é usado como unidade de ponto flutuante em algumas implementações de diferentes versões da arquitetura. Os coprocessadores CP1, CP2 e CP3 não foram implementados no cMIPS.

3.2 MEMÓRIA VIRTUAL NO MIPS

A implementação de memória virtual no cMIPS atende ao especificado no manual do fabricante[4]. Para isso, além do “processador” que executa as instruções

comuns, o Coprocessador-0 é composto de um conjunto de registradores invisíveis ao programador de aplicação, e de lógica que gerencia os eventos ligados à memória virtual, interrupções e exceções.

Para fazer a tradução de endereço virtual para físico o cMIPS usa uma TLB totalmente associativa de 8 posições. Quando é necessário alterar alguma das entradas da TLB o processador executa instruções para que o coprocessador do cMIPS as modifique.

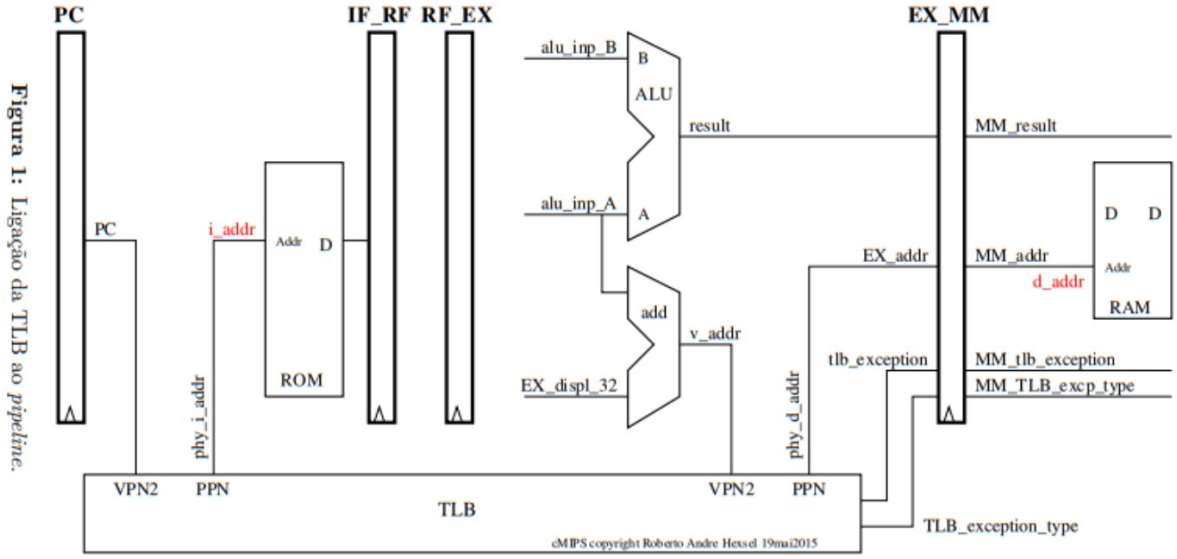
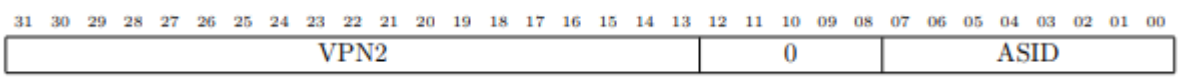


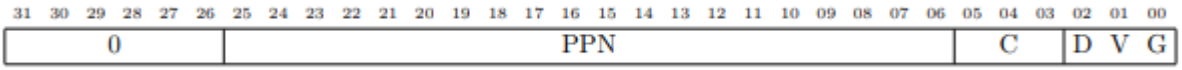
Figura 1: Ligação da TLB ao *pipeline*.

Diagrama da TLB no cMIPS fonte: [8]

Cada entrada da TLB possui três campos de 32 bits, VPN2, PPN0 e PPN1. O VPN2 contém os 19 bits mais altos do número da página virtual (campo VPN2), e o ASID (Address Space Identifier) que é o identificador do processo. Os campos PPN0 e PPN1 são compostos pelo número da página física (PPN), o bit “dirty” (D), que diz se uma página tem permissão para ser modificada, o bit “valid” (V) que diz se o mapeamento é válido e a página pode ser acessada, e o bit “global” (G) que diz se a página é uma página global (pode ser acessada por qualquer processo) ou se apenas o processo com o ASID do VPN2 pode usar a página. Em PPN0 e PPN1 também há o atributo de coerência da cache (C), que não foi implementado no cMIPS.



VPN2



PPN0 e PPN1

Como o VPN2 não contém o bit mais baixo do número da página virtual, os campos PPN0 e PPN1 contém o endereçamento de duas páginas virtuais consecutivas.

3.3 CP0

Para ler ou alterar as entradas da TLB, o processador usa alguns registradores e instruções para interagir com a TLB através do CP0.

Os registradores usados são o EntryHi, EntryLo0, EntryLo1, Index, Random e Wired.

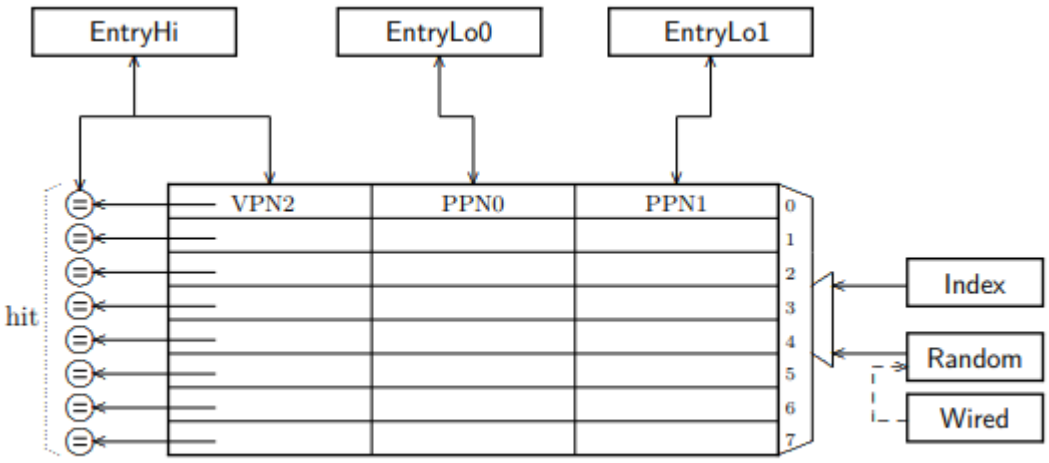
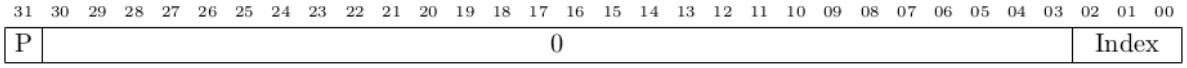


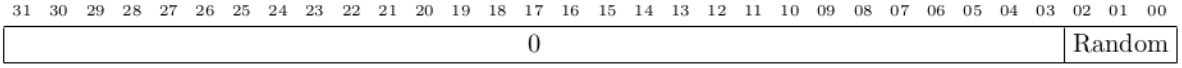
Figura 2: Acesso à TLB através dos registradores do CP0.

TLB do MIPS fonte: [8]

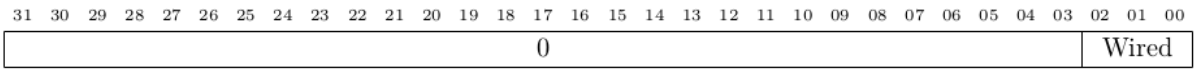
EntryHi, EntryLo0 e EntryLo1 são os registradores usados para ler ou escrever nos campos VPN2, PPN0 e PPN1 de uma entrada da TLB, e tem o mesmo respectivo formato. Index é usado para indicar o índice da entrada da TLB da qual foi ou será feita a leitura ou escrita. Random indica o índice da entrada da TLB onde foi feita uma escrita no caso de uma escrita em uma entrada aleatória, e Wired indica quais entradas devem ser protegidas no caso de uma escrita em uma posição aleatória.



Registrador Index



Registrador Random



Registrador Wired

Para fazer uma escrita nos registradores do coprocessador é usada a instrução mtcp0 (move to CP0), e para fazer a leitura é usada a instrução mfcp0 (move from CP0).

Para alterar uma entrada específica da TLB os campos EntryHi, EntryLo0 e EntryLo1 devem ser preenchidos com os dados a serem salvos em VPN2, PPN0 e PPN1, e o registrador Index deve ser preenchido com a posição da TLB na qual será feita a escrita. Feito isso, deve ser usada a instrução tlbwi (TLB write indexed) para que o CP0 salve os dados do registrador na TLB.

Para fazer a leitura de uma entrada da TLB, o registrador Index deve ser preenchido com a posição da TLB de onde será feita a leitura, e então deve ser executada a instrução tlbr (TLB read), para que o CP0 salve os campos VPN2, PPN0 e PPN1 da TLB, nos registradores EntryHi, EntryLo0 e EntryLo1.

Para fazer a escrita em uma posição aleatória da TLB, devem ser preenchidos os registradores EntryHi, EntryLo0 e EntryLo1, e usada a instrução tlbwr (TLB write random). Essa instrução faz com que o CP0 escolha uma posição aleatória de índice entre o valor de Wired e 7, grava o índice escolhido em Random e os conteúdos dos registradores EntryHi, EntryLo0 e EntryLo1 em VPN2, PPN0 e PPN1.

Caso se deseje saber qual posição da TLB possui o mapeamento de uma página virtual, EntryHi deve ser preenchido com o VPN2 e o ASID da página virtual a ser procurada. Quando for executada a instrução tlbp (TLB probe), caso o mapeamento da página esteja na TLB o CP0 irá preencher o registrador Index com o índice da posição que contém o mapeamento, e caso contrário, ele irá colocar o valor 1 no bit mais alto de Index (P).

3.4 TABELA DE PÁGINAS

A tabela de páginas que foi utilizada é composta por entradas com os campos eLo0 e eLo1 que tem o mesmo formato que PPN0 e PPN1, e os campos intLo0 e intLo1, de 32 bits que são usados pelo SO.

TP	0 3	4 7	8 11	12 15
[0]	eLo0	intLo0	eLo1	intLo1
[1]	eLo0	intLo0	eLo1	intLo1
[2]	eLo0	intLo0	eLo1	intLo1
...		...		
[256K−2]	eLo0	intLo0	eLo1	intLo1
[256K−1]	eLo0	intLo0	eLo1	intLo1

Estrutura da tabela de páginas. Fonte: [8]

Em intLo0 e intLo1 são guardadas as permissões de escrita (W) e execução (X), os bits para indicar se a página foi modificada (M), ou usada (U), o seu status (ST) e o endereço da página em memória secundária (sec), caso ela não esteja em memória principal. Os valores possíveis para o status são 00, para páginas não mapeadas, 01, para paginas mapeadas na memória principal, 10 para páginas guardadas na região de swap, e 11 é um status não definido.

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06	05 04 03 02	01 00
sec	M U W X	ST

intLo0 e intLo1

Na tabela de páginas atual, não há um campo para guardar qual é o processo ao qual a página pertence. Para trabalhos futuros onde for adicionado o controle de acesso das páginas por processo poderia ser usado o campo sec de intLo0 e intLo1 para guardar o ASID do processo de cada página. Caso já seja implementada a

paginação sob demanda, esse campo vai ser usado, e os elementos da TP precisariam ser aumentados para incluir o ASID. Outra opção seria usado uma TP separada para cada thread, dispensando o necessidade de salvar o ASID para cada página das tabelas de mapeamento.

3.5 EXCEÇÕES NO MIPS

As exceções no MIPS são tratadas pelo CP0, que é responsável por desviar o fluxo para o código de tratamento de exceção, impedir que instruções que causaram uma exceção sejam executadas, e colocar os valores adequados nos registradores para que o sistema operacional possa descobrir a causa da exceção e então tratá-la.

Os registradores usados no caso de uma exceção são:

Cause, que indica a causa da exceção. Ele é congelado após uma exceção e só é atualizado quando é feita uma leitura com a instrução mfc0 r, c0_cause ou quando a instrução causadora da exceção é executada novamente após a instrução eret (exception return).

31	30	29	28	27	26	25	24	23	22 21 20 19 18 17 16	15 14 13 12 11 10 09 08	07	06 05 04 03 02	01 00
BD	TI	0	0	DC	0	0	0	IV	x	$IP_7..IP_0$	0	ExcCode	0

Registrador Cause

Os campos de Cause são

BD, se for igual a 1 indica se a ultima exceção ocorreu num branch delay slot.

TI, se for igual a 1 indica que há uma interrupção do temporizador pendente.

DC, se igual a 1 desabilita o contador interno.

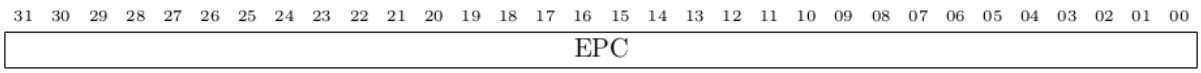
IP_i, indicam que a interrupção de nível i está pendente.

ExcCode, contém o código da ultima exceção, conforme a tabela 1.

binário	dec	mnemônico	causa da excessão
00000	0	Int	interrupção
00001	1	Mod	modificação de página (*store* em página protegida)
00010	2	TLBL	excessão da TLB (*load* ou busca)
00011	3	TLBS	excessão da TLB (*store*)
00100	4	AdEL	erro de endereçamento (*load* ou busca)
00101	5	AdES	erro de endereçamento (*store*)
00110	6	IBE	erro no barramento de instruções
00111	7	DBE	erro no barramento de dados
01000	8	Sys	*syscall*
01001	9	Bp	*breakpoint*
01010	10	RI	instrução reservada (opcode inválido)
01100	12	Ov	*overflow*
01101	13	Tr	*trap*
11111	31	–	nenhuma excessão pendente

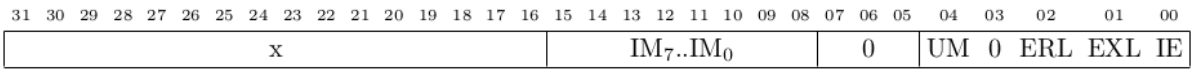
tabela 1: Códigos de exceção de Cause

EPC (Exception PC), que é atualizado por hardware com o endereço da instrução causadora ao ocorrer uma exceção. Também pode ser atualizado por software.



Registrador EPC

Status, esse registrador pode ser alterado pelo programador. Ele controla quais interrupções estão habilitadas, e indica qual o modo do sistema (usuário ou kernel), nivel de exceção (erro ou exceção). Quando o processador está em “modo exceção”, as interrupções são ignoradas.



Registrador Status

Os campos de Status são:

Im_i, se igual a 1, habilita a interrupção de nível i.

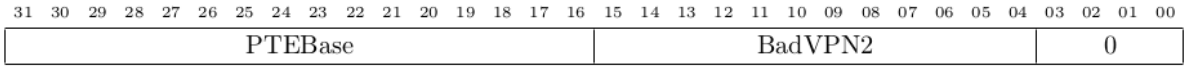
UM, se igual a 1 o processador está no modo usuário, se 0 está no modo kernel.

ERL, se igual a 1, indica que está em error level.

EXL, se igual a 1, indica que está em exception level. Quando está em exception level as interrupções são ignoradas.

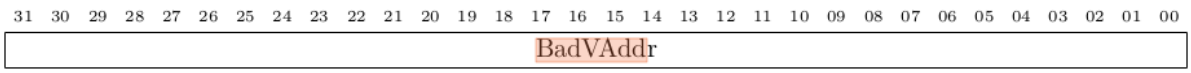
IE, Se for igual a 1, as interrupções estão habilitadas.

Context, guarda o endereço da base da tabela de páginas (PTEBase), que deve ser inicializado por software, e o índice da entrada na TP onde ocorreu a exceção (BadVPN2), que é preenchido por hardware. Unindo esses dois valores, Context é aponta para o endereço da entrada na tabela de páginas que corresponde à exceção de memória virtual.



Registrador Context

BadVAddrs, guarda o endereço virtual que causou a exceção mais recente



Registrador BadVaddrs

4 XINU

O sistema operacional XINU[1] (XINU is not Unix) é um sistema operacional desenvolvido para ser simples e modular. Ele foi criado para ser usado em sala de aula, no aprendizado de sistemas operacionais, e começou a ser usado em aulas de laboratório na UFPR[9] com o porte dele para o processador cMIPS[7][5].

O XINU suporta o uso dos recursos do computador por vários programas concorrentes, e usa threads como se fossem processos Unix. Deixando qualquer parte da memória acessível para todos as threads ou processos que estejam executando na máquina. Neste trabalho o objetivo foi fazer uma implementação inicial de memória virtual no SO, deixando a região de instruções protegida, e os endereços de memória indisponíveis para o processador não acessíveis para os processos através do XINU.

Esse SO é implementado na linguagem C e possui partes em assembly para código que é dependente de máquina. A implementação de memória virtual foi toda feita em assembly, pois é preciso fazer uso dos recursos do coprocessador-0 para controlar a TLB e lidar com as exceções disparadas por ele.

4.1 MEMÓRIA NO XINU

O sistema XINU divide a memória em 4 regiões continuas ao ser iniciado. O segmento de texto, onde fica o código executável e podem ser colocadas as constantes. O segmento de dados, onde ficam as variáveis globais inicializadas. O segmento Bss (block started by symbol) que contém as variáveis globais não inicializadas. E a ultima região é a memória livre, que cobre todo o resto da memória física. A pilha é alocada no topo da memória física/virtual instalada.

A memória livre é usada pelo SO de duas formas diferentes, como heap e como stack. O heap é a região de onde os programas que estão sendo executados podem alocar memória dinamicamente. O stack é a pilha de uma thread, onde são guardados os registros de ativação e as variáveis locais. Ao ser iniciado uma nova thread no SO, ele aloca uma nova pilha exclusiva para este processo.

O XINU não possui nenhum mecanismo de controle de acesso aos dados em memória, deixando toda a memória física aberta para o uso de qualquer thread.

Como não temos um carregador para iniciar novos processos, é possível apenas criar novas threads, que compartilham os segmentos de texto, dados e Bss.

O programador é responsável por proteger estruturas de dados compartilhadas entre processos com o uso de semáforos.

5 IMPLEMENTAÇÃO DE PAGINAÇÃO NO XINU-CMIPS

A implementação de paginação no XINU-cMIPS faz com que seja possível controlar o acesso às páginas da memória virtual. Nesta implementação, existe uma única tabela de páginas, compartilhada com todos os processos. Para tornar a simulação mais rápida, o SO usa apenas 8 páginas de ROM de instruções, 16 páginas de RAM de memória de dados, e mais 2 páginas de RAM com a tabela de páginas. Como o tamanho da tabela de páginas é de 2 páginas, ela possui 32 entradas (4 kBytes / (4 * 32 Bytes)), e pode mapear 64 páginas. Para evitar que as páginas restantes do mapeamento sejam utilizadas, elas todas tem um mapeamento inválido. Os endereços dos periféricos do cMIPS ficam em um endereço muito alto para serem mapeados por uma TP desse tamanho. Para que eles possam ser acessados, eles são carregados diretamente na TLB, e ficam protegidos por wired.

A tabela de páginas fica gravada nas duas páginas mais altas da memória (simulada) do cMIPS, criando um novo segmento na memória inicial do XINU. Foi preciso definir o limite da região de memória livre, para que o segmento da TP não fosse sobrescrito com a pilha de uma thread, ou com a alocação dinâmica do heap.

Neste trabalho, o mapeamento da tabela de páginas é a identidade, portanto qualquer tradução de um endereço virtual retorna o mesmo valor como endereço físico. A tabela de páginas é iniciada com a região de texto (localizada na ROM do cMIPS) mapeada, e com permissão de execução. O restante da ROM é marcada como não mapeada. Os mapeamentos da RAM são todos feitos com permissão de leitura e execução, e marcados como mapeados. Todos os mapeamentos feitos são marcados como páginas globais, e podem ser acessados por quaisquer threads que estejam executando.

Para que seja implementado a memória virtual com paginação no cMIPS, precisaram ser incluídos os tratadores para as exceções da TLB. Esses tratadores foram escritos em assembly porque são dependentes da arquitetura do processador, e as instruções de controle do CP0 não são acessíveis em linguagem C.

5.1 EXCEÇÕES DA TLB

Quando ocorre uma falta na TLB, o CP0 dispara uma exceção e o fluxo de execução desvia automaticamente para as rotinas de tratamento das exceções. As exceções pode ser de diferentes tipos, dependendo do evento que a causou.

Quando uma exceção é detectada, o registrador Context é carregado com com o número da página onde ocorreu a exceção, e também salva o endereço que causou a falta em BadVAddr. O registrador EPC recebe o endereço da instrução causadora da falta. Ele pode ser diferente de BadVaddr caso a instrução seja uma leitura ou escrita de memória. O campo EXL de Status recebe o valor 1 para indicar que o processador está em exception level. O campo ExcCode de Cause recebe o código da exceção sofrida. EntryHi é preenchido por hardware com o número da página que causou a falta.

Além disso, o valor de PC é alterado para o endereço do tratador da exceção causada, e as instruções nos estágios de busca, decod e exec do pipeline são anuladas, porque a exceção só é detectada no estágio de memória.

As faltas na TLB são tradas pelo tratador 0x0000, exclusivo para isso, e as outras exceções causadas pela TLB são tratadas pelo tratador geral de exceções 0x0180. Estes nomes correspondem aos endereços em que os tratadores residem em um SO para o MIPS32r2.

Para retornar de uma exceção, o programador deve usar a instrução eret, que carrega o PC com o endereço em EPC, para que a instrução que causou a exceção seja buscada e executada novamente. Eret muda o campo EXL de Status para 0, para indicar que saiu do exception level, e também anula as instruções nos estagio de busca, decod, e exec (*pipeline flush*).

5.1.1 TLBrefill – Falta na TLB

Ocorre quando há uma falta na TLB. Essa exceção faz com que o CP0 redirecione o processador para o tratador de exceções 0x0000, que é usado exclusivamente para o tratamento de faltas na TLB. O campo ExcCode de causa é preenchido com o valor 2 ou 3, dependendo se foi uma falta causada por um load,

busca ou store. Esse campo é atualizado para 31 quando executada a instrução eret.

Para resolver essa exceção basta o SO usar o valor de Context para buscar na tabela de páginas os mapeamentos das páginas faltantes. Os valor de EntryLo0 e EntryLo1 são copiados da tabela de páginas para uma entrada aleatória da TLB. Como o Valor de EntryHi é preenchido por hardware, não há necessidade dele ser preenchido por código.

Após o mapeamento ser carregado na TLB, é usada a instrução eret para que a instrução causadora da falta seja executada novamente. Este tratador é implementado em oito instruções.

mfc0 k1, c0_context	Carrega context em k1
lw k0, 0(k1)	Carrega eLo0 da página em k0
lw k1, 8(k1)	Carrega eLo1 da página em k1
mtc0 k0, c0_entrylo0	Carrega k0 em entryLo0
mtc0 k1, c0_entrylo1	Carrega k1 em entryLo1
ehb	Garante que as escritas nos entryLo foram concluídas
tlbwr	Escreve em uma posição aleatória da TLB
eret	Retorna do tratador de exceção

Código do tratador de TLBrefill

Existe a possibilidade de a mesma instrução causar mais de uma falta, caso ela seja uma leitura ou escrita. Neste caso a primeira falta seria causada pelo endereçamento da instrução. A segunda ocorre quando a instrução for executada novamente, mas dessa vez uma falta no endereçamento na memória de dados de onde deveria ser feito o load ou store.

5.1.2 TLBmod – Página Modificada

Essa exceção ocorre quando é feito uma escrita em uma página pela primeira vez (bit dirty de EntryLo = 0 na TLB). Com essa exceção, PC é alterado para o endereço do tratador geral de exceções 0x0180. Esse tratador inicia saltando para o tratador correto da exceção, baseado no campo ExcCode de Cause.

No tratador de TLBmod, é buscado na tabela de páginas o valor de eLo (eLo0 ou eLo1) da página causadora da exceção. Ele é usado para verificar se a página está mapeada em memória, se tem permissão de escrita, se está na memória principal, e se não é uma instrução. Caso algum dos testes tenha resultado negativo,

a simulação é abortada. Num sistema realista, ocorreria uma *segmentation fault* e o processo seria abortado.

Após passar com sucesso pelos testes, a entrada na tabela de página tem o bit dirty alterado para 1, indicando que ela pode ser escrita, e os bits de modificado e usado da tabela de páginas são marcados com 1, indicando que a página foi usada, e que seus dados podem ser modificados. Com isso concluído, o tratador volta para a instrução causadora com a instrução eret.

O código fonte do tratador da exceção está no apêndice deste documento.

O abortamento da simulação é um ponto que ainda pode ser melhorado no sistema. No caso de a página não estar em memória principal, ela deveria ser trazida da memória secundária, da posição apontada pelo campo sec de eLo0 ou eLo1. Isso deverá ser feito quando for implementado paginação sob demanda. Nos outros casos, o SO deveria matar o processo que causou a exceção, e não abortar o sistema inteiro.

5.1.3 Double Fault – Falta Dupla

Este é um caso especial, que ocorre no caso de ter ocorrido uma exceção TLBrefill e o mapeamento da tabela de páginas não estar presente na TLB. Isso causa mais uma falta na TLB, desta vez quando o SO tentar ler o valor de eLo0 da página que causou a primeira falta. Caso essa exceção que ocorre durante o tratamento de uma exceção de TLBrefill fosse tratada pelo mesmo tratador, o SO poderia entrar em um loop infinito.

O modo que o processador trata isso é desviado para o tratador geral de exceções 0x1800, e atualiza os valores de BadVaddrs com o endereço causador da exceção e Context com o elemento da tabela de páginas que causou a falta na TLB. valor de EPC é mantido apontando para a instrução que causou a primeira exceção.

O tratador geral então direciona para a rotina que examina o valor de context, e busca na tabela de páginas o mapeamento da própria tabela de páginas, e o salva na TLB. Com isso a instrução eret é executada, e o processador executa a instrução causadora da primeira falta. Como esse mapeamento ainda não foi carregado na TLB, a instrução gera uma nova exceção TLBrefill.

Desta vez, o mapeamento da tabela de páginas já está na TLB, e o tratador consegue executar normalmente. O código fonte do tratador dessa segunda exceção está no apêndice.

6 TRABALHOS FUTUROS

Com a implementação de memória virtual desse trabalho, o problema de escrita na memória de instruções, ou de uso de páginas não mapeadas foi resolvido. Mas, para tornar o XINU mais interessante poderia ser modificado o método que aloca a pilha de cada processo para que a página que a contém seja marcada para ser usada apenas pelo seu pai. O mesmo poderia ser feito também com os métodos que fazem a alocação dinâmica de memória para as threads. Dessa forma teríamos uma maior segurança no sistema, embora a memória de instruções e variáveis ainda sejam todos compartilhados.

Outro ponto a ser melhorado no gerenciamento de memória virtual do XINU-cMIPS é o abortamento da simulação em caso de exceções que deveriam abortar apenas a thread causadora da exceção.

Futuramente também pode ser implementada a paginação sob demanda no sistema, tornando ele mais próximo de um sistema operacional atual, que fornece muitas facilidades para o programador com o gerenciamento de memória.

APÊNDICE

1 TRATADOR DA EXCEÇÃO TLBMOD

```
handle_Mod:            # EntryHi points to offending TLB entry
      tlbp             # what is the offender's index?
      lui  k1, %hi(_excp_saves)
      ori  k1, k1, %lo(_excp_saves)
      sw   a0,  9*4(k1) # save registers
      sw   a1, 10*4(k1)
      sw   a2, 11*4(k1)
      mfc0 a0, c0_badvaddr
      andi a0, a0, 0x1000    # check if even or odd page
      beq  a0, zero, M_even
      mfc0 a0, c0_context
M_odd:
      addi a2, a0, 12   # address for odd entry (intLo1)
      mfc0 k0, c0_entrylo1
      ori  k0, k0, TLB_DIRTY # mark TLB entry as dirty/writable
      j    M_test
      mtc0 k0, c0_entrylo1
M_even:
      addi a2, a0, 4    # address for even entry (intLo0)
      mfc0 k0, c0_entrylo0
      ori  k0, k0, TLB_DIRTY # mark TLB entry as dirty/writable
      mtc0 k0, c0_entrylo0
M_test:
      lw   a1, 0(a2)    # read PT[badVAddr].intLo{0,1}
      mfc0 k0, c0_badvaddr    # get faulting address
      andi a0, a1, TP_MAPPED  # check if page is mapped
      nop
      beq  a0, zero, M_seg_fault    # no, abort simulation
      nop
      andi a0, a1, TP_WR_ABLE # check if page is writable
      nop
      beq  a0, zero, M_prot_viol    # no, abort simulation
      nop
      andi a0, a1, TP_SEC_MEM # check if page is in secondary memory
```

```
      nop                             #
      bne  a0, zero, M_sec_mem     # yes, abort simulation
      nop
      mfc0 a0, c0_epc   # check if fault is on an instruction
      nop                             # wait, what?
      beq  a0, k0, M_prot_viol     # k0 is badVAddr, if so, abort
      nop
      ori  a1, a1, (TP_USED | TP_MODIF) # mark PT entry as modified, used
      sw   a1, 0(a2)
      tlbwi             # write entry with dirty bit=1 back to TLB
      lw   a0,  9*4(k1) # restore saved registers and return
      lw   a1, 10*4(k1)
      lw   a2, 11*4(k1)
      j    _excp_0180ret
      nop
M_seg_fault:      # print message and abort simulation
      la   k1, x_IO_BASE_ADDR
      sw   k0, 0(k1)
      jal  cmips_kmsg
      la   k1, 3        # segmentation fault
      nop
      nop
      nop
      wait 0x31
M_prot_viol:      # print message and abort simulation
      la   k1, x_IO_BASE_ADDR
      sw   k0, 0(k1)
      jal  cmips_kmsg
      la   k1, 2        # protection violation
      nop
      nop
      nop
      wait 0x32
M_sec_mem:  # print message and abort simulation
      la   k1, x_IO_BASE_ADDR
      sw   k0, 0(k1)
      jal  cmips_kmsg
      la   k1, 4        # secondary memory
      nop
```

```
      nop
      nop
      wait 0x33
```

2 TRATADOR DA EXCEÇÃO TLBL

```
handle_TLBL:      # EntryHi points to offending TLB entry
      tlbp        # probe it to find the offender's index
      lui  k1, %hi(_excp_saves)
      ori  k1, k1, %lo(_excp_saves)
      sw   a0,  9*4(k1)
      sw   a1, 10*4(k1)
      sw   a2, 11*4(k1)
      mfc0 a0, c0_badvaddr
                  # check is fault is to address below the PT
      la   a1, (_PT + (x_INST_BASE_ADDR >>13)*16)
      slt  a2, a0, a1   # a2 <- (badVAddr <= PageTable_bottom)
      bne  a2, zero, L_chks         #fault is not to PageTable
      nop
      # get physical page number for two pages at the bottom of PageTable
      la    a0, ( MIDDLE_RAM >>13 )<<13
      mtc0  a0, c0_entryhi           # tag for bottom double-page
      la    a0, ( (MIDDLE_RAM + 0*4096) >>12 )<<6
      ori   a1, a0, 0b00000000000000000000000000000111 # ccc=0, d,v,g1
      mtc0  a1, c0_entrylo0          # bottom page (even)
      la    a0, ( (MIDDLE_RAM + 1*4096) >>12 )<<6
      ori   a1, a0, 0b00000000000000000000000000000111 # ccc=0, d,v,g1
      mtc0  a1, c0_entrylo1          # bottom page + 1 (odd)
      # and write it to TLB[8]
      li    k0, 4
      mtc0  k0, c0_index
      tlbwi
      j L_ret           # all work done, return
      nop
L_chks:
      andi a0, a0, 0x1000     # check if even or odd page
      nop
      beq  a0, zero, L_even
      mfc0 a0, c0_context
```

```
L_odd:
      j    L_test
      addi a2, a0, 12   # address for odd intLo1 entry
L_even:
      addi a2, a0, 4    # address for even intLo0 entry
L_test:
      lw   a1, 0(a2)    # get intLo{0,1}
      mfc0 k0, c0_badvaddr    # get faulting address for printing
      andi a0, a1, TP_MAPPED # check if page is mapped
      nop
      beq  a0, zero, M_seg_fault    # no, abort simulation
      nop
      andi a0, a1, TP_SEC_MEM # check if page is in secondary memory
      nop
      bne  a0, zero, M_sec_mem      # yes, abort simulation
      nop
      ori  a1, a1, TP_USED    # mark PT entry as used
      # sw   a1, 0(a2)
      # if this were handler_TLBS, now is the time to also mark the
      #    PT entry as Modified
      # mark PT entry as used, writable and modified
      ori  a1, a1, (TP_USED | TP_MODIF | TP_WR_ABLE)
      sw   a1, 0(a2)
L_ret:      lw   a0,  9*4(k1)       # nothing else to do, return
      lw   a1, 10*4(k1)
      lw   a2, 11*4(k1)
      j    _excp_0180ret
      nop
```

3 ROTINAS DE TESTE DE MEMÓRIA VIRTUAL [6]

```
//
// This program references a wide range of addresses to do a walk over
//   several entries of the Page Table (PT).
// As the pages are referenced, the TLB must be refilled with entries from
//   the PT.
//
// This program needs a large (simulated) RAM.  RAM must be allocated
//   so that the bottom half is "usable memory" and the top half is
//   allocated to the Page Table.  The base address and RAM size must
//   be set on file  vhdl/packageMemory.vhd.
//
```

```
//   x_DATA_BASE_ADDR : reg32   := x"00040000";
//   x_DATA_MEM_SZ    : reg32   := x"00020000";
//
// With this much memory, simulations run slowly.  No free lunch.


//-----------------------------------------------------------------------
// decide on the tests to run
#define WALK_THE_PT  1
#define TLB_MODIFIED 0
#define DOUBLE_FAULT 0

// these will abort the simulation when the fault is detected/handled
#define PROT_VIOL 0
#define SEG_FAULT 0
//-----------------------------------------------------------------------


#include "cMIPS.h"

extern void PT_update(void *V_addr, int component, int new_value);
extern  int TLB_purge(void *V_addr);
static void print_str(char *);

#define FALSE (0==1)
#define TRUE  !FALSE

#define MAX_RAM ( x_DATA_BASE_ADDR + (x_DATA_MEM_SZ / 2) )

#define NUM_RAM_PAGES  ( (x_DATA_MEM_SZ / 2) / 4096 )




void main(void) {
  int i, rsp, new_value;
  int *walker;


#if WALK_THE_PT
  //-------------------------------------------------------------
  // write to the middle of all datapages
  //   this will cause some TLB refill exceptions, which should be
  //   handled smoothly by the handler at excp_0000 (include/start/s)
  //-------------------------------------------------------------
  walker = (int *)(x_DATA_BASE_ADDR + 1024);

  for (i = 0 ; i < NUM_RAM_PAGES; i++) {
    *walker = i;
    walker = (int *)((int)walker + 4096);
  }

  // and now read what was written
  walker = (int *)(x_DATA_BASE_ADDR + 1024);

  for (i = 0 ; i < NUM_RAM_PAGES; i++) {
    print( *walker );
    walker = (int *)((int)walker + 4096);
```

```
  }

  print_str("\n\twalked\n");
#endif



#define PG_NUM 10



#if TLB_MODIFIED
  //---------------------------------------------------------------------
  // let's change a mapping to cause a TLB-Modified exception
  //
  // in fact, there will be TWO exceptions:
  // (1) a TLB-Refill will copy the mapping from the PT and retry the
  //     instruction;
  // (2) when the sw is retried, it will cause a TLB-Modified
  //     exception, which checks PT's protection bits, then fixes the
  //     dirty bit in the TLB, and retries again, succeeding this time.
  //---------------------------------------------------------------------
  // ( ( (x_DATA_BASE_ADDR + n*4096) >>12 )<<6 ) || 0b000111  d,v,g

  walker = (int *)(x_DATA_BASE_ADDR + PG_NUM*4096);

  // first, remove V_addr from the TLB, to ensure the PT is searched
  if ( TLB_purge((void *)walker) == 0 ) {
    print_str("\n\tTLB entry purged\n\n");
  } else {
    print_str("\n\tTLB miss\n\n");
  }

  new_value = ( ((x_DATA_BASE_ADDR + PG_NUM*4096)>>12) <<6) | 0b000011; //
d=0
  PT_update( (int *)walker, 0, new_value);

  new_value = 0x00000009;                    // writable, mapped
  PT_update( (int *)walker, 1, new_value);

  *walker = 0x99;              // cause a TLBrefill, then a TLBmod

  if ( *walker == 0x99 ) {     // this load is optimized away by gcc
    print( *walker );
    print_str("\n\tMod ok\n");
  } else {
    print_str("\n\tMod err\n");
 }
#endif



#if DOUBLE_FAULT
  //---------------------------------------------------------------------
  // let's remove from the TLB the mapping for the PT itself and cause
  // a double-fault:
  //
```

```
  // (1) on the 1st reference, TLB-refill does not find a mapping for
  //     the PT on the TLB; this causes a TLBL (load) exception;
  // (2) routine handle_TLBL writes a new mapping on the PT, refills
  //     the TLB, then retries the reference;
  // (3) the page referenced is not on the TLB, so there is another
  //     TLB-refill, which loads the mapping, the store is retried
  //     and succeeds.
  //-----------------------------------------------------------------

  // remove the TLB entry for datum to be referenced
  walker = (int *)(x_DATA_BASE_ADDR + PG_NUM*4096 + 1024);
  if ( TLB_purge((void *)walker) == 0 ) {
    print_str("\taddr purged from TLB\n");
  } else {
    print_str("\tTLB miss\n");
  }

  // this is the base of the page table
  walker = (int *)(x_DATA_BASE_ADDR + (x_DATA_MEM_SZ/2));

  // remove the TLB entry which points to the PT
  if ( TLB_purge((void *)walker) == 0 ) {
    print_str("\tPT purged from TLB\n");
  } else {
    print_str("\twtf?\n");
  }

  // now reference a mapped page, to cause the double fault
  walker = (int *)(x_DATA_BASE_ADDR + PG_NUM*4096 + 1024);
  *walker = 0x88;              // cause a TLBrefill then a TLBload

  if ( *walker == 0x88 ) {     // this load is optimized away by gcc
    print( *walker );
    print_str("\tdouble ok\n");
  } else {
    print_str("\tdouble err\n");
 }
#endif



#if PROT_VIOL
  //-------------------------------------------------------------------------
  // let's cause a protection violation -- write to a write-protected page
  //   this will abort the simulation
  //-------------------------------------------------------------------------

  walker = (int *)(x_DATA_BASE_ADDR + PG_NUM*4096);

  // first, remove V_addr from the TLB, to ensure the PT is searched
  if ( TLB_purge((void *)walker) == 0 ) {
    print_str("\tpurged\n");
  } else {
    print_str("\tTLB miss\n");
  }

  // change a PT element so it is data, NON-writable, page is mapped in PT
```

```
  new_value = ( ((x_DATA_BASE_ADDR + PG_NUM*4096)>>12) <<6) | 0b000011; //
d=0
  PT_update( (int *)walker, 0, new_value);
  new_value = 0x00000001;                          // NOT-writable, mapped
  PT_update( (int *)walker, 1, new_value);

  *walker = 0x77;

  // will never get here -- protection violation on the store
  if ( *walker == 0x77 ) {    // this load is optimized away by gcc
    print( *walker );
    print_str("\tprot viol not ok\n");
  } else {
    print_str("\tprot viol err\n");
 }
#endif




#if SEG_FAULT
  //---------------------------------------------------------------------
  // let's cause a segmentation fault -- reference to page not mapped
  //   this will abort the simulation
  //---------------------------------------------------------------------

#define PG_UNMAPPED 20

  // pick a page that is not mapped
  walker = (int *)(x_DATA_BASE_ADDR + PG_UNMAPPED*4096); // page not mapped

  // first, remove V_addr from the TLB, to ensure the PT will be searched
  if ( TLB_purge((void *)walker) == 0 ) {
    print_str("\tpurged\n");
  } else {
    print_str("\tTLB miss\n");
  }

  // add a new PT element for an address range with RAM but UN-mapped
  //   this address is above the page table
  new_value =
    (((x_DATA_BASE_ADDR + PG_UNMAPPED*4096)>>12) <<6) | 0b000011; // d=0
  PT_update( (int *)walker, 0, new_value);
  PT_update( (int *)walker, 1, 0);              // mark as unmapped
  new_value =
    (((x_DATA_BASE_ADDR + (PG_UNMAPPED+1)*4096)>>12) <<6) | 0b000011; //
d=0
  PT_update( (int *)walker, 2, new_value);
  PT_update( (int *)walker, 3, 0);              // mark as unmapped

  *walker = 0x66;

  // will never get here -- seg fault on the store
  if ( *walker == 0x66 ) {    // this load is optimized away by gcc
    print( *walker );
    print_str("\tseg fault not ok\n");
  } else {
    print_str("\tseg fault err\n");
```

```
 }
#endif


  to_stdout('\n');
}
//-----------------------------------------------------------------------


void print_str(char *s) {
  int i;
  i = 0;
  while (s[i] != '\0') {
    to_stdout(s[i]);
    i = i + 1;
  }
}
```

REFERÊNCIAS

[1] Douglas E Comer, **Operating System Design - The XINU Approach**. segunda edição. CRC Press. 2015.

[2] Peter J. Denning. Virtual Memory. **ACM Computing Surveys**. Vol. 2, p. 153-189, 1970.

[3] MIPS Technologies INC. **MIPS32 Architecture for Programmers, Volume I: Introduction to the MIPS32 Architecture**. Revisão 2.50. 2005.

[4] MIPS Technologies INC. **MIPS32 Architecture for Programmers, Volume III: Privileged Resource Architecture**. Revisão 2.50. 2005.

[5] Vanessa B. A. Oliveira. **Porte do XINU para cMIPS**. 61f. Trabalho de Graduação (Ciência da Computação) – Setor de Ciências Exatas, Universidade Federal do Paraná, Curitiba, 2016.

[6] Roberto A. Hexsel. GitHub. **cMIPS**. Disponível em: <https://github.com/rhexcel/cmips>. Acesso em: 20 jun. de 2019.

[7] Roberto A. Hexsel. GitHub. **xinu-cMIPS**. Disponível em: <https://github.com/rhexcel/xinu-cmips>. Acesso em: 20 jun. de 2019.

[8] Roberto A. Hexsel. **Laboratório sobre Tratamento de Exceções de Memória Virtual**. Disponível em: <http://www.inf.ufpr.br/roberto/ci212/labTLB.pdf>. Acesso em: 20 jun. de 2019.

[9] Roberto A. Hexsel. **Projeto de Sistemas Operacionais CI315**. Disponível em: <http://www.inf.ufpr.br/roberto/CI315-17-1.html>. Acesso em: 20 jun. de 2019.