1 TLB no cMIPS

A Figura 1 mostra um diagrama de blocos do cMIPS com a TLB. O diagrama não mostra a interface de programação da TLB, que é discutida adiante.

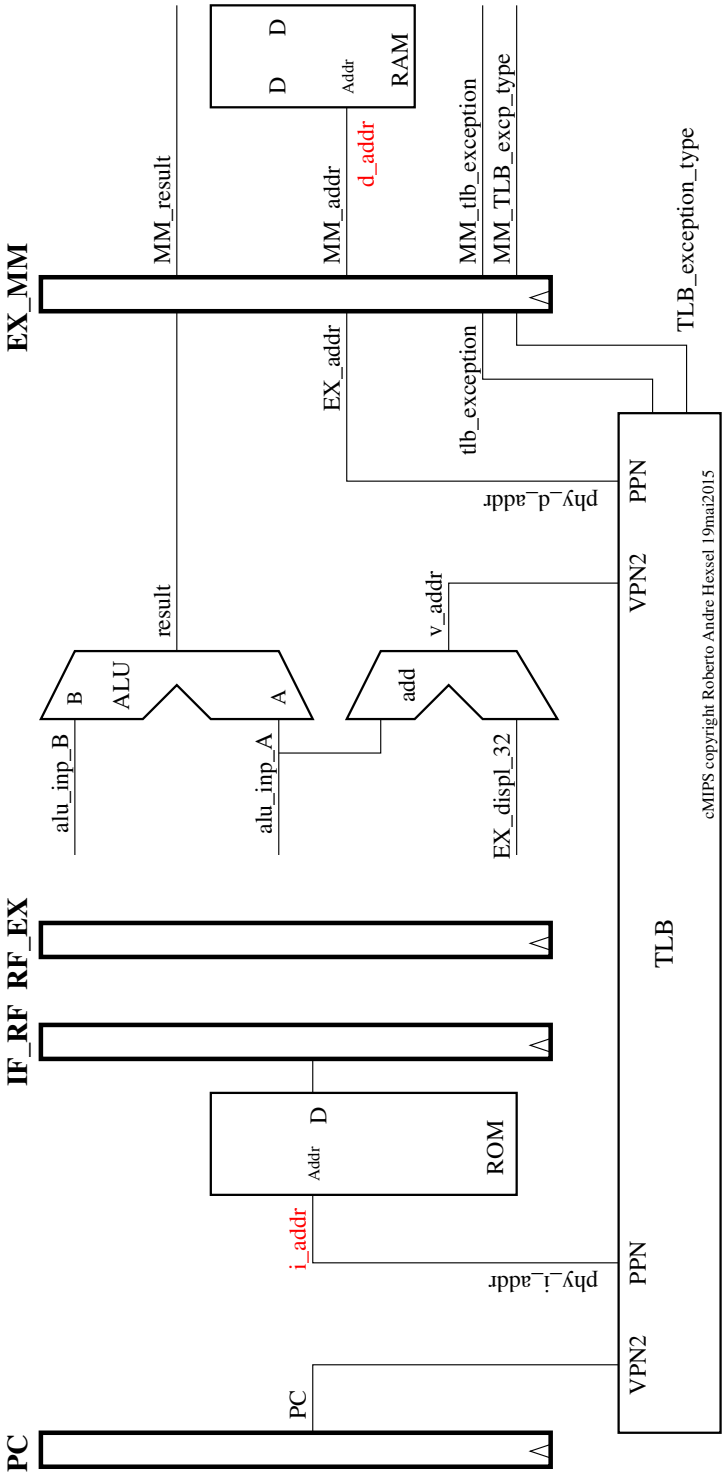


Figura 1: Ligação da TLB ao *pipeline*.

São duas as instruções usadas para acessar os registradores do CP0: `mfc0` *move from coprocessor 0*, e `mtc0` *move to coprocessor 0*. As duas tem dois argumentos, um dos 32 registradores de uso geral, e o número de um registrador do CP0. Além dessas, as instruções específicas à TLB são descritas adiante.

A interface de programação da TLB consiste dos registradores Index, Random, Wired, EntryLo0, EntryLo1, e EntryHi. Os conteúdos de cada um destes são descritos nas próximas seções. A tripla EntryHi, EntryLo0 e EntryLo1 é usada para acessar – leitura e escrita – um elemento da TLB. A Figura 2 mostra um diagrama de blocos da TLB do cMIPS.

A TLB no cMIPS tem oito elementos e é totalmente associativa. Cada elemento mapeia um par de páginas físicas; a cada etiqueta (VPN2) correspondem um par de páginas virtuais contíguas, que mapeiam as duas páginas físicas (PPN0 e PPN1) que lhes correspondem.

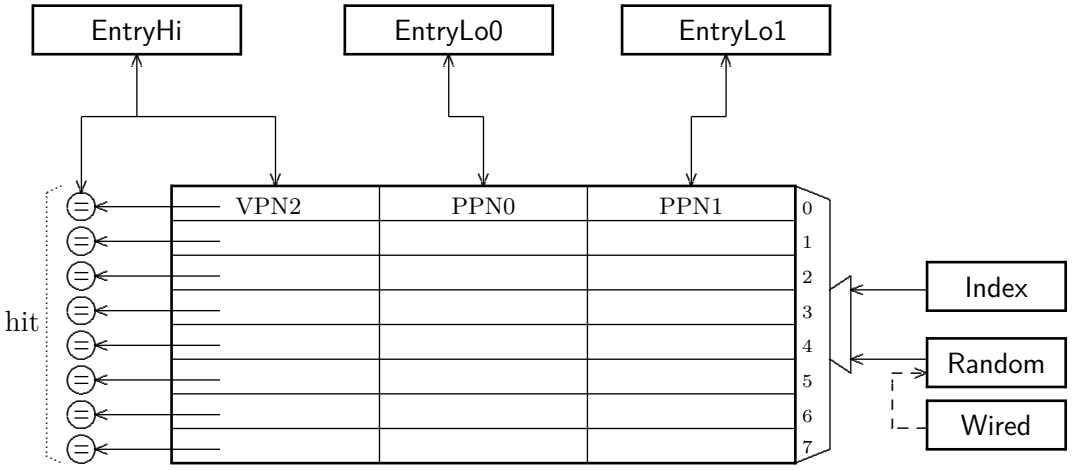


Figura 2: Acesso à TLB através dos registradores do CP0.

A TLB pode ser encarada como uma memória de 8 posições. O registrador Index permite escolher um dos 8 elementos da TLB para leitura ou atualização. O registrador Random escolhe, aleatoriamente, um dos elementos, e é usado na atualização dos conteúdos da TLB. O registrador Wired protege um subconjunto da TLB das reposições aleatórias; quando da reposição aleatória, os elementos de número menor que conteúdo de Wired não são apontados por Random. Posto de outra forma, Random conta de 7 até Wired.

ASID é o *Address Space IDentifier* do processo que executa no processador. Em sistemas Unix, o ASID é computado com alguma função do PID do processo, porque o ASID é um campo de 8 bits, enquanto que o PID tem 15 bits – $|\,\mathsf{ASID}\,| \ll |\,\mathsf{PID}\,|$.

Considere a leitura do elemento apontado por Index. Depois que a instrução `tlbr` (*TLB read*) é executada, o registrador EntryHi contém a etiqueta (VPN2) e o ASID do elemento. Os registradores EntryLo0 e EntryLo1 contém os números das duas páginas físicas (PPN0 e PPN1) associadas à VPN2, mais os bits de estado de cada página física. O trecho de código abaixo mostra a leitura de um elemento da TLB.

```
1  li    t1, 5           # deseja-se ler o elemento 5
2  mtc0  t1, c0_index    # grava 5 em Index
3  tlbr                  # lê elemento em TLB[5]
4  mfc0  s0, c0_ehi      # copia conteúdos para registradores s0,s1,s2
5  mfc0  s1, c0_elo0
6  mfc0  s2, c0_elo1
```

Os valores em `s0`, `s1`, `s2` podem ser usados para, por exemplo, contabilizar referências às páginas virtuais ou físicas. Em breve veremos trechos completos de código com estas instruções para manusear os conteúdos da TLB.

No caso da escrita numa posição indicada por Index, os três registradores devem ser preenchidos e então gravados na TLB com um **tlbwi** (*TLB write indexed*).

```
1  ...                  # prepara novo elemento da TLB em s0-s2
2  mtc0  s0, c0_ehi     # copia novos conteúdos para os registradores
3  mtc0  s1, c0_elo0    #   da interface com TLB
4  mtc0  s2, c0_elo1
5  li    t1, 5          # escrita será do elemento 5
6  mtc0  t1, c0_index   # grava 5 em Index
7  tlbwi               # atualiza elemento em TLB[5]
```

Se, ao invés de um elemento determinado, deseja-se escolher um elemento a esmo para a reposição, o procedimento de escrita na TLB é similar ao acima, exceto que escrever em Index é desnecessário, e a instrução de escrita é **tlbwr** (*TLB write random*). Os candidatos à reposição são os elementos de número maior ou igual que o conteúdo de Wired.

```
1  ...                  # prepara novo elemento da TLB em s0-s2
2  mtc0  s0, c0_ehi     # copia novos conteúdos para os registradores
3  mtc0  s1, c0_elo0    #   da interface com TLB
4  mtc0  s2, c0_elo1
5  tlbwr               # atualiza elemento N da TLB, N>=Wired
```

A TLB pode ser consultada, com a instrução **tlbp** (*TLB probe*). O registrador EntryHi deve ser preenchido com os VPN2 e ASID desejados, e então uma **tlbp** preenche o registrador Index com o número do elemento da TLB que contém a etiqueta com os VPN2 e ASID gravados em EntryHi.

```
1  ...                  # VPN2 e ASID gravados em s0
2  mtc0  s0, c0_ehi     # copia para EntryHi
3  tlbp                # busca na TLB
4  mfc0  t1, c0_index   # Index(31)=0 indica que Index(2..0) aponta
5                       #   para o elemento com a etiqueta buscada
```

Se o par ⟨VPN2,ASID⟩ não é encontrado, então Index_{31}=1 e o campo $\mathsf{Index}_{2..0}$ é indeterminado.

2 Control Processor 0 no cMIPS

O 'processador' do MIPS executa as "instruções de usuário" dos programas, que são as instruções que estudamos até agora.

O *Control Processor 0* (CP0) é o 'co-processador' que executa as instruções que controlam o ambiente (de *hardware*) em que os programas de usuário executam. Estas "instruções de controle" são usadas para implementar sistemas operacionais, através de trocas – bem-comportadas – de execução em *modo usuário* para execução *modo sistema*, além do tratamento de todas as condições excepcionais que possam ocorrer durante a execução de um programa.

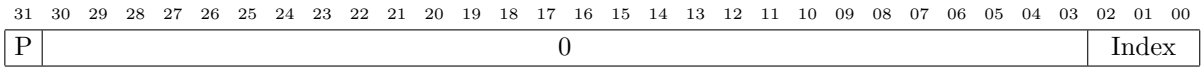
2.1 Registradores do CP0

Os títulos das próximas seções indicam o número do registrador no CP0, e a página de sua definição completa, em

> *MIPS32 Architecture for Programmers, Volume III: The MIPS32 Privileged Resource Architecture*, MIPS Technologies, Rev. 2.50, 2005.

Campos marcados com 'x' são irrelevantes para estas atividades. Campos marcados com '0' (zero) devem ser escritos com '0'.

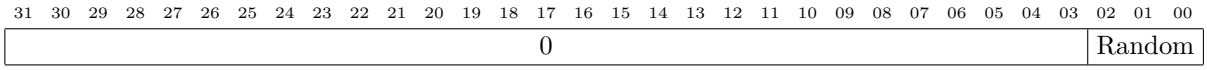
2.1.1 Index (CP0-0, pg.61)



P — *Probe failure* após a execução de `tlbp` (probe);
P=0: indica que o elemento foi encontrado e Index aponta para o elemento;
P=1: indica que elemento não foi encontrado e o conteúdo de Index é indefinido.

Index — índice do elemento na TLB encontrado após uma `tlbp` (probe) se P=0; usado para indexar a TLB nas escritas com a instrução `tlbwi` (write-indexed), e nas leituras com `tlbr` (read).

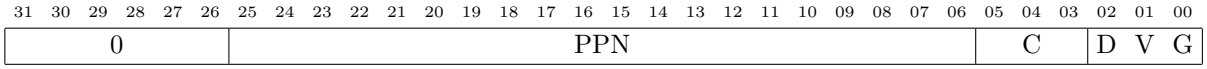
2.1.2 Random (CP0-1, pg.62)



Random — número usado para a reposição aleatória na TLB. Random varia no intervalo [Wired,7].

2.1.3 EntryLo0 (CP0-2) e EntryLo1 (CP0-3, pg.63)

Estes registradores devem ser preenchidos por *software* antes de uma escrita na TLB. Após um `tlbp`, se a etiqueta desejada foi encontrada, estes registradores são preenchidos por *hardware*.



PPN — número da página física, 20 bits;

C — atributo de coerência de caches, não implementado no cMIPS;

dirty — D=1: página pode ser atualizada;
D=0: uma escrita na página causa excessão TLB-modified;

valid — V=1: mapeamento é válido;
V=0: acesso à página causa excessão TLB-invalid;

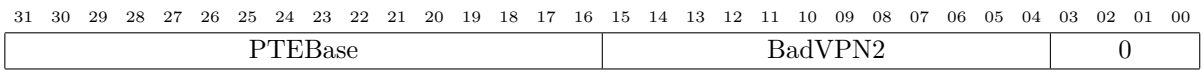
global — G=1: página é global e ASID é ignorado;
G=0: o ASID é comparado junto com VPN2 numa busca na TLB.

Para preencher este registrador com o número de uma página física, elimina-se os 12 bits do deslocamento na página, e abre-se espaço para os 6 bits de status: `PPN = ( P_addr >>12 ) << 6`.

2.1.4 Context (CP0-4, pg.67)

Este registrador difere da especificação para que a TP possa ser alocada em endereço 'baixo'.

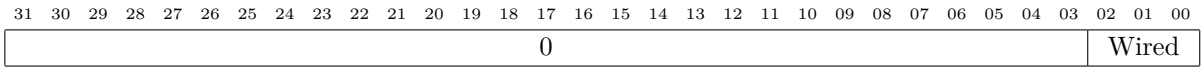
O campo PTEBase deve ser inicializado por *software* com o endereço da base da Tabela de Páginas (TP), e o campo BadVPN2 é o índice na TP, preenchido por *hardware* na ocorrência de uma excessão. A TP é um vetor de pares de números de páginas físicas, indexado por Context.



PTEBase — endereço da base da TP – *este campo difere da especificação;*

BadVPN2 — endereço ($VA_{31..13}$) do par de páginas no qual ocorreu exceção.

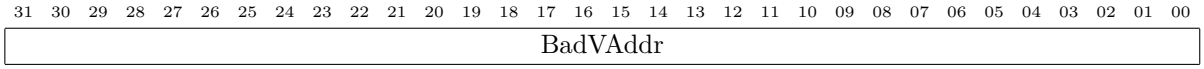
2.1.5 Wired (CP0-6, pg.72)



Wired número de elementos da TLB que estão 'grudados' na TLB e não serão sobrescritos por uma instrução `tlbwr` (*write-random*).

2.1.6 BadVAddr (CP0-8, pg.74)

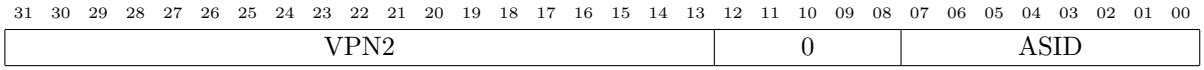
Mantém o endereço virtual que sofreu a exceção mais recente, que é uma dentre AddressError (AdEL ou AdES), TLB-refill, TLB-invalid (TLBL ou TLBS) ou TLB-modified. BadVAddr pode ser diferente de EPC em *loads* e *stores*.



2.1.7 EntryHi (CP0-10, pg.76)

Este registrador deve ser preenchido por *software* antes de uma escrita na TLB, ou antes de uma busca (*probe*) na TLB com `tlbp`.

Quando ocorre uma falta na TLB (excessão TLB-refill), este registrador é preenchido por *hardware*, com VPN2 apontando para a página virtual causadora da falta. O tratador desta excessão (TLB-refill) não precisa alterar EntryHi.



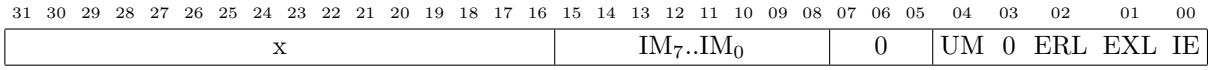
VPN2 endereço virtual $VA_{31..13}$ para o par de páginas virtuais nas quais ocorreu a excessão;

ASID *address space identifier*. Este campo deve ser preenchido por *software* para definir o ASID do processo que está a executar. Quando um elemento da TLB é lido, este campo é sobrescrito com o ASID daquele elemento, que pode ser distinto do ASID do processo corrente. Se EntryLo0.G=1 ou EntryLo1.G=1, a página é *global* e o ASID é ignorado.

Para preencher este registrador com o número de uma página virtual, elimina-se os 13 bits do deslocamento (*par de páginas*) e acrescenta-se o ASID: `VPN2 = ((V_addr >>13)<<13) | ASID`.

2.1.8 Status (CP0-12, pg.79)

Este registrador pode ser alterado por *software*; os bits *UM, ERL, EXL, IE* são alterados por *hardware* na ocorrência de uma excessão.



$IM_7..IM_0$ *interrupt mask*; se bit IM_i=1 então a interrupção de nível i está habilitada;

UM *user mode*, UM=1 indica modo usuário, UM=0 indica modo sistema;

ERL *at error level*, ERL=1 quando ocorre um dentre Reset, SoftReset, NMI ou CacheError;

EXL *at exception level*, EXL=1 quando ocorre uma exceção que não seja Reset, SoftReset, NMI ou CacheError. Se Status.EXL=1, as interrupções são ignoradas;

IE *interrupt enable*, IE=1 indica que as interrupções estão habilitadas.

2.1.9 Cause (CP0-13, pg.92)

Este registrador indica a causa da última excessão ou os níveis das interrupções pendentes. Os campos *TI, DC, IV* podem ser alterados por *software*; os demais são atualizados, a cada ciclo por *hardware*. Quando ocorre uma excessão, o campo ExcCode 'congela' e só é atualizado após uma leitura, com `mfc0 r, c0_cause`, ou pela execução de **`eret`**.

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

BD *in branch delay*, BD=1 indica que a última exceção ocorreu num *branch delay slot*;

TI *timer interrupt*, TI=1 indica que há uma interrupção do temporizador pendente;

DC *disable Count register*, DC=1 desabilita o contador interno; DC=0 habilita o contador. Usado para reduzir dissipação de energia em aplicações de baixo consumo;

IV *interrupt uses general exception vector*, IV=0 para tratamento de interrupções como se fossem exceções (`0x0180`) ; IV=1 para o tratamento das interrupções ser desviado para o vetor de interrupções (`0x0200`) – este é o modo recomendado para o cMIPS;

$IP_7..IP_0$ interrupção pendente; se bit IP_j=1 então a interrupção de nível j está pendente;

ExcCode *exception code*, identifica a exceção, conforme a Tabela 1.

Tabela 1: Códigos de excessão, bits 6 a 2 do registrador Cause.

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

2.1.10 EPC (CP0-14, pg.97)

Contém o endereço no qual o processamento reinicia após o tratamento de uma excessão. Se Status.EXL=0, o conteúdo do EPC é atualizado pelo *hardware* na ocorrência de uma excessão ou interrupção.

O conteúdo de EPC pode ser alterado e/ou consultado por *software*.

O conteúdo de EPC pode ser distinto de BadVAddr em *loads* e *stores*.

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
EPC

2.2 Alterações no estado do processador

Suponha que uma excessão da TLB é detectada ao acessar a memória de dados, durante a execução da instrução no endereço `0x1004`, que é um *load* do endereço `0x0040.0000`. As seguintes alterações no CP0 ocorrem automaticamente e no mesmo ciclo:

EPC ← `0000.1004`	salva endereço da instrução causadora
BadVAddr ← `0040.0000`	salva endereço com a excessão
Context.BadVPN2 ← `0040.0000/(4K*2)`	salva VPN2 com a excessão
Status.EXL ← 1	entra em *EXception Level*
Cause.ExcCode ← {1,2,3}	sinaliza excessão (cfe. Tabela 1)
PC ← `&( tratador() )`	salta para tratador geral

anula instruções em BUSCA, DECOD, EXEC

Depois do tratamento da excessão, quando o processador executar a instrução **`eret`**, o estado se altera para:

PC ← EPC	busca novamente a instrução causadora
Status.EXL ← 0	sai do *EXception Level*

anula instruções em BUSCA, DECOD, EXEC

3 Tratadores de Exceção `0x0000`, `0x0180`

Os projetistas do MIPS definiram um modelo *sui generis* para o tratamento de faltas na TLB e faltas na TP. O modelo emprega um tratador específico para faltas na TLB, e outro para todas as demais excessões, incluindo aí aquelas associadas à memória virtual.

O evento mais frequente são as faltas na TLB, e estes são tratados da forma mais rápida e simples possível. Os demais eventos são relativamente raros, e seu tratamento é mais custoso. Esta divisão atende à máxima *make the common case fast* [PH14].

Vejamos como é a Tabela de Páginas, e então as excessões causadas pelos vários tipos de referências à memória – faltas na TLB, faltas de página, falta de segmentação e violação de proteção.

3.1 Tabela de Páginas

A TP é organizada como mostra a estrutura `PTentry`. Os componentes de tipo `entryLo` são formatados *exatamente* como os registradores EntryLo0 e EntryLo1. Além destes, dois inteiros são reservados para uso pelo SO, um para cada um dos mapeamentos.

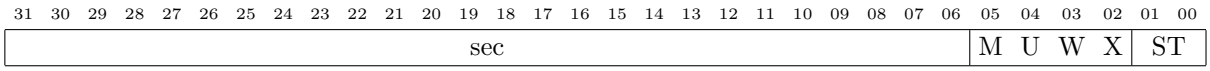
```
typedef struct entry {
  entryLo eLo0;     // mesmo formato que registrador entryLo0, 32 bits
  int     intLo0;   // inteiro para uso do SO
  entryLo eLo1;     // mesmo formato que registrador entryLo1, 32 bits
  int     intLo1;   // inteiro para uso do SO
} PTentry;

PTentry PT[262144]; // tabela de páginas cobre 2GB (2G/(4K*2) = 256K)
```

A tabela de páginas tem 256K elementos porque, no SO definido pela MIPS, somente 2Gbytes são mapeados para modo usuário na TP, e 2G/(4K×2)=256K. Lembre que cada elemento da TP cobre um par de páginas virtuais contíguas.

O diagrama abaixo mostra a disposição, em memória, dos elementos da TP. Os números pequenos são os deslocamentos, em bytes, dos elementos da estrutura.

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

Os inteiros `intLo0` e `intLo1` mantém as permissões de alto nível, como seriam definidas pelo SO. No nosso protótipo de SO, estes inteiros são assim formatados:



sec — este campo conteria um apontador para a página em memória secundária;

M — *modificado*, M=1 indica que a página em memória física difere da sua cópia na memória secundária e portanto está suja;

U — *usado*, U=1 indica que a página foi referenciada. Esta informação é usada para implementar o 'envelhecimento' das páginas em memória física, como alguma aproximação para LRU;

W — *writable*, W=0 indica que a página não pode ser alterada. Se for modificada, ocorre uma violação de proteção e a simulação é encerrada;

X — *executable*, X=0 indica que a página não pode ser executada. Se ocorrer uma busca nesta página, ocorre uma violação de proteção e a simulação é encerrada;

ST — *status* da página. 'fatal' significa que a simulação é encerrada;
ST=00 página não é mapeada e ocorre uma falta de segmentação, fatal;
ST=01 página está mapeada no espaço de endereçamento do processo;
ST=10 página está "armazenada em memória secundária", fatal nestas simulações;
ST=11 comportamento indefinido.

3.2 Ambiente de Simulação e o protoOS

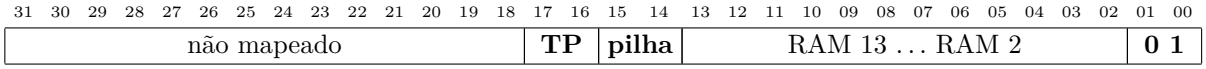
Nosso sistema operacional contém um *mínima minimorum* para que o material das seções anteriores, e das próximas, faça algum sentido.

Mantenho-me fiel ao nível de violência padrão GoT empregado nas aulas sobre este assunto. Não há dó nem piedade. Nosso sistema não emprega memória secundária, e qualquer referência que não esteja em memória principal aborta a simulação. Violações de privilégio e faltas de segmentação abortam a simulação. Como não há um SO de verdade, uma vez detectado o crime, a única penalidade plausível é a capital.

Para reduzir o tempo de simulação, são mapeadas somente quatro páginas de ROM, e mais 32 páginas de RAM. Fui obrigado a empregar um truque sujo para permitir simulações com pouca memória: a metade de baixo da RAM (páginas 0 a 15) é "memória usável", e a metade de cima (páginas 16 a 31) é usada para a tabela de páginas. Isso agride brutalmente a especificação contida no livro sagrado. O mundo é imperfeito e você deve aprender a lidar com as coisas como elas podem ser. Do contrário, as simulações ficam insuportavelmente lentas.

Nos exercícios que seguem, somente excessões em acessos à memória RAM são investigados.

O mapa da memória RAM é mostrado abaixo. A metade inferior, com 16 páginas, é a "memória usável" pelos programas de usuário. A pilha está alocada nas duas páginas mais altas da memória usável. A metade superior contém a tabela de páginas nas duas primeiras páginas, e todo o resto é não mapeado.



O endereço base da RAM é `x_DATA_BASE_ADDR=0x0004.0000`, definido em `include/cMIPS.h`.

O programa `include/start.s` inicializa a TLB com os seguintes mapeamentos:

`TLB[0]`	`ROM 0,1`	início da ROM, com o executável
`TLB[1]`	`E/S 0,1`	endereços dos periféricos
`TLB[2]`	`RAM 0,1`	início da RAM, dados inicializados
`TLB[3]`	`RAM 14,15`	pilha, no topo da RAM usável
`TLB[4]`	`TP 0, 1`	base da tabela de páginas

Lembre que a TLB mapeia um par de páginas físicas para cada etiqueta (VPN2), e é por isso que a tabela mostra um elemento da TLB (`TLB[0]`) como mapeando as duas páginas de ROM (`ROM 0,1`), nos endereços 0 e 4K. Para os fins desta aula, todos os mapeamentos são a *identidade*, porque não faz sentido re-alocar as páginas físicas e complicar ainda mais os testes. Num sistema real, os mapeamentos seriam totalmente associativos e dependentes da ordem em que as páginas fossem carregadas em memória física.

Estes mapeamentos são 'grudados' na TLB, Wired é inicializado com 5 e assim ficam disponíveis para reposição aleatória os elementos 5, 6 e 7.

O programa `testes/pt_walk.c` contém os programas de teste.

Faça bom uso da compilação condicional para reduzir o nível de ruído nos testes.

espaço em branco proposital

3.3 TLBrefill – Falta na TLB

Quando uma referência causa uma falta na TLB, isso provoca uma excessão TLB-refill que é desviada para o `Tratador 0x0000`, dedicado exclusivamente ao tratamento de faltas na TLB.

O código do tratador é extremamente simples. Ao detectar a excessão, o *hardware* carrega o registrador Context com o VPN2 (índice na TP) da página na qual ocorreu a falta.

Context é usado como apontador na TP – Context.BadVPN2 é o índice na TP da página com o endereço causador. O elemento apropriado da TP é carregado para os registradores EntryLo0 e EntryLo1, e então estes são copiados para uma posição aleatória da TLB. Este trecho de código foi copiado de [Swe07], pág. 145.

Programa 1: Tratador da Excessão TLB-refill.

```
_excp_0000:
    mfc0 k1, c0_context     # read Context for PTbase[VPN2]
    lw   k0, 0(k1)          # k0 <- PT[Context].Entrylo0
    lw   k1, 8(k1)          # k1 <- PT[Context].EntryLo1
    mtc0 k0, c0_Elo0        # EntryLo0 <- k0 = even element
    mtc0 k1, c0_Elo1        # EntryLo1 <- k1 = odd element
    ehb                     # exception hazard barrier
    tlbwr                   # update random TLB element
    eret                    # and return (Status.EXL <- 0)
```

A instrução **ehb** (*exception hazard barrier*) garante que a carga de todos os registradores completou antes da escrita na TLB.

Ao detectar a excessão, além de Context, o *hardware* carrega o registrador BadVAddr com o endereço que causou a falta.

O registrador EPC é carregado pelo *hardware* com o endereço da instrução causadora (se Status.EXL=0). EPC pode ser distinto de BadVAddr.

Ao executar **eret**, o PC é carregado com o conteúdo de EPC, e a instrução causadora é buscada novamente. A causadora é buscada novamente para encher o *pipeline*, e porque um *load* ou um *store* podem causar *duas* faltas na TLB: uma na busca e outra no acesso ao dado. Além disso, salvar o PC da causadora é mais econômico do que salvar o estado de todos os registradores de segmento do processador.

Durante a execução deste tratador o processador fica em *exception level* (Status.EXL=1). Quando a **eret** é executada, o processador sai do *exception level* (Status.EXL $\leftarrow 0$).

O registrador Cause indica a causa da excessão em Cause.ExcCode=$\{2,3\}$, e este campo é reinicializado em 31 com o **eret**.

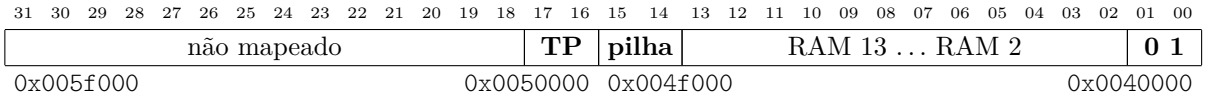
espaço em branco proposital

3.4 *Page Table Walk*

Habilite este teste 'ligando' somente o `#define WALK_THE_PT` no topo de `tests/pt_walk.c`.

O Programa 2 faz uma 'caminhada' por todas as 16 páginas de dados usáveis, primeiro escrevendo em todas elas, e então lendo o que foi escrito.

O programa causa várias excessões TLB-refill porque a TLB não é grande o suficiente para acomodar os mapeamentos para todas as páginas de dados – três mapeamentos são usados para ROM, E/S e TP, mais dois para a base da RAM e para a pilha. Sobram 3 mapeamentos livres para acomodar os 6 VPN2 das 12 páginas que não são a base da RAM ou a pilha.



Programa 2: Percorre todas as páginas de dados, escrevendo e lendo.

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

walker = (int *)(x_DATA_BASE_ADDR + 1024);

for (i = 0; i < NUM_RAM_PAGES; i++) { // write to each page
  *walker = i;
  walker = (int *)((int)walker + 4096);
}

walker = (int *)(x_DATA_BASE_ADDR + 1024);

for (i = 0; i < NUM_RAM_PAGES; i++) { // and now read them back
  print( *walker );
  walker = (int *)((int)walker + 4096);
}
```

Todas as referências que não estão na TLB causam somente uma TLB-refill porque não ocorre nenhuma violação de proteção e nem de segmento – veremos estes casos adiante.

Acompanhe a execução deste trecho com `gtkwave` para se familiarizar com o que ocorre nas excessões TLB-refill. Para tanto, reveja o Programa 1, e diga

```
run.sh -v tlb.sav -t 50 -u u -w &
```

para executar a simulação e invocar `gtkwave`. Desligou os `#define`s indesejados?

Preste atenção aos registradores EPC, Cause, Status, Context, BadVAddr, Index e Random. Na parte de baixo do diagrama de tempo são mostrados todos os elementos da TLB, suas etiquetas e os mapeamentos correspondentes.

O executável é tão grande que tenho preferido usar um editor (`zile`) para acompanhar as simulações. O comando que eu uso para compilar e então ler o executável está abaixo e foi copiado de `/bin/compile.sh`. Na linha de comando, troque `zile` pelo editor de sua preferência.

```
compile.sh -O2 pt_walk.c && mv prog.bin data.bin .. && \
  mips-objdump -z -D -EL \
      -M reg-names=mips2r2 -M cp0-names=mips2r2 --show-raw-insn \
      --section .text  --section .data --section .rodata \
      --section .sdata --section .sbss --section .bss --section .PT \
      pt_walk.elf > xxxx && zile xxxx
```

O programa `nm` (*names*) ajuda a encontrar os nomes das variáveis e funções, em `testes` diga:

```
nm pt_walk.elf | egrep 'walker|_excp_0180|_excp_0000'
```

3.5 Outras Excessões

As outras excessões relacionadas com a memória virtual são tratadas no *tratador geral de excessões* (*general exception handler*), que inclui o tratamento de todas as excessões que não sejam (i) interrupções (se Cause.IV=1), (ii) falta na TLB e (iii) erros na cache.

Por exemplo, se o elemento recém carregado na TLB, pela TLB-refill for inválido – porque a página não está mapeada (EntryLo0.v=0) – então quando a instrução causadora for re-executada após o **eret**, o mapeamento inválido provoca uma nova excessão, uma falta na TP, que é tratada pelo *tratador geral de excessões*.

O código do tratador deve ler Cause.ExcCode para descobrir qual a causa da excessão e se for o caso invocar os serviços do SO para carregar a página faltante em memória física. Neste exemplo, a página não está mapeada, (EntryLo0.v=0 e `intLo0.ST=00`), portanto esta é uma falta de segmentação que redunda no término sumário da 'execução' (simulação) do processo.

O trecho de pseudocódigo abaixo indica como separar e tratar as excessões provocadas pela TLB. Veja a Tabela 1 para os códigos das excessões apontadas pelo registrador Cause.ExcCode.

Programa 3: Pseudocódigo do Tratador Geral de Excessões.

```
switch ( Cause.ExcCode ) {  //

   case 0: PANIC();  // interrupt, should never get here

   case 1:           // TLB modified
      copy_TPentry_to_entryLo0_entryLo1();
      check_if_user_can_write_to_this_page();
      then_fix_EntryLo_V_D_or_segmentation_fault();
      mark_page_as_modified_in_PT();
      break;

   case 2:           // TLBL: load or instruction fetch
      copy_TPentry_to_entryLo0_entryLo1();
      check_if_EPC_equals_badVAddr_for_ifetch_or_load();
      check_if_mapping_is_valid_and_invoke_SO_to_load_page();
      else_kill_process();
      break;

   case 3:           // TLBS: store
      copy_TPentry_to_entryLo0_entryLo1();
      check_if_user_can_write_to_this_page();
      then_fix_EntryLo_D_or_segmentation_fault();
      check_if_mapping_is_valid_and_invoke_SO_to_load_page();
      else_kill_process();
      break;
   ...
   case 31:
   default: PANIC(); // should never get here
}
```

De novo: suponha que ocorreu uma excessão de falta na TLB (TLB-refill), e o mapeamento faltante foi carregado na TLB no tratamento da TLB-refill. Suponha ainda que o mapeamento carregado na TLB é inválido (EntryLo0.V=0). Quando instrução causadora é buscada novamente, após o **eret**, esta instrução causa outra excessão, dessa vez uma TLB-load ou TLB-store (TLBmod), que é tratada no tratador geral do Programa 3.

Vejamos a TLB-modificada – tentativa de escrita em página protegida.

3.6 *TLBmod*

Habilite este teste 'ligando' somente `#define TLB_MODIFIED` no topo de `tests/pt_walk.c`.

Este programa provoca uma excessão de "primeira escrita" numa página. Da preparação:

(a) no início do teste, o mapeamento da página onde ocorrerá a escrita é expurgado da TLB, para garantir que a TP será consultada quando ocorrer o *store* e a TLB-refill;

(b) o primeiro `PT_update()` marca o componente EntryLo0 do elemento da TP como protegido contra a escrita (d=0);

(c) no segundo `PT_update()`, o componente `intLo0` do elemento da TP é alterado, adicionando-lhe a permissão para escrita.

Programa 4: Provoca uma excessão TLB-refill e uma TLB-modified.

```
#define NUM_RAM_PAGES  ( (x_DATA_MEM_SZ / 2) / 4096 )
#define PG_NUM 10

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");
} else {
  print_str("\n\tTLB␣miss\n");
}

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

new_value = 0x00000009;            // mark page as 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("\tMod␣ok\n\n");
} else {
  print_str("\tMod␣err\n\n");
}
```

Veja em `include/handlers.s`, a rotina `handle_TLBmod`. Finalmente, o teste:

(1) a primeira tentativa de escrita causa uma falta na TLB, e um TLB-refill;

(2) o tratamento da TLB-refill carrega o mapeamento na TLB; este mapeamento diz que a página não pode ser atualizada – por causa do EntryLo0.d ←0 no primeiro `PT_update()`. TLB-refill retorna para o *store*, cujo endereço fora armazenado em EPC;

(3) o *store* é buscado novamente, e agora ocorre uma excessão TLB-modified porque o mapeamento na TLB diz que a página é protegida (d=0);

(4) o tratador da TLB-modified consulta a TP e descobre que a página pode ser atualizada (intLo0.W=1). O mapeamento na TLB é consertado (EntryLo0.d ←1), a TP é marcada como Usada e Modificada. TLB-modified retorna para o *store*, que é o endereço em EPC;

(5) o *store* é buscado pela terceira vez, e agora sua execução completa porque o mapeamento está na TLB e indica que a escrita é permitida.

Duas excessões distintas, três buscas e tentativas de execução. Fica ainda mais emocionante.

3.7 *Inception* – A Excessão Dentro da Excessão

Esta é, na minha opinião, a 'sacada' mais genial dos projetistas do MIPS, embora ela seja um tanto indigesta. Vamos pois com calma.

A tabela de páginas de um processo ocupa *muito* espaço em memória, e um espaço de endereçamento (que é mapeado pela TP) mapeia, nos SOs da MIPS, 2 Gbytes para programas de usuário. Cada VPN2 (2×4 Kbytes) corresponde a um elemento da TP com 16 bytes; portanto, cada TP necessita

$$2\,\text{GB}/(2 \times 4\,\text{K}) \times 16\,\text{bytes} = 4\,\text{Mbytes}\,, \quad 4\,\text{Mbytes}/4096 = 1024\,\text{páginas}$$

e, potencialmente, são muitos os processos executando concorrentemente. O $|\,\mathsf{ASID}\,| = 8$ bits, o que permite acomodar 2^8 espaços de endereçamento ativos, sendo assaz conveniente que partes das inúmeras TPs possam ser armazenadas em memória secundária,

Lembre que um espaço de endereçamento contém alguns mapeamentos válidos em endereços baixos, nas seções de código e dados, e mais alguns em endereços altos, para a pilha. No meio, há um enorme vazio, e a maior parte da TP contém mapeamentos inválidos porque uma faixa grande dos endereços não são referenciados pelo processo. Além disso, a localidade garante que, num dado intervalo, somente um subconjunto pequeno das páginas necessita estar mapeado em memória física – é justamente por isso que a TLB é efetiva.

As páginas da TP que mapeiam o "espaço vazio" não precisam estar em memória primária, e a localidade indica que as páginas impopulares da TP também podem ficar fora da memória primária, liberando espaço precioso para páginas populares e que foram referenciadas pelos processos num passado recente.

Quando o tratador de TLB-refill (Programa 1) é executado, pode ocorrer uma falta na TLB na execução de um dos `lw`. Quer dizer, o mapeamento do elemento da TP apontado por Context *não está na TLB*, e o acesso àquela página (da própria TP) causa uma *segunda* falta na TLB – uma excessão TLB-refill ocorre durante o tratamento de uma excessão TLB-refill. Sinistro!

A saída é tão simples quanto elegante: a segunda excessão é transferida diretamente para o tratador geral, que acessa a TP e carrega o mapeamento (da TP) na TLB. Pela ordem:

(1) quando ocorre a primeira excessão TLB-refill, os registradores EPC, BadVAddr e Context são carregados com os endereços gerados pela instrução causadora (busca ou memória), o processador muda para *exception level* (Status.EXL←1), e o controle desvia para o tratador de TLB-refill (Programa 1);

(2) ao executar o primeiro `lw`, ocorre a segunda TLB-refill, BadVAddr e Context são carregados com o endereço efetivo acessado por este `lw` – que é um elemento da Tabela de Páginas. EPC *não é atualizado* porque o processador já se encontra em *exception level*. O controle desvia para o tratador geral;

(3) o tratador geral examina Context para descobrir o endereço da instrução causadora da (segunda) excessão, e acessa a TP para obter o mapeamento da VPN2 que contém a própria TP. Este mapeamento é inserido na TLB;

(4) quando o tratador universal executar `eret`, o controle retornará para a instrução causadora da primeira excessão porque o EPC ainda aponta para aquela instrução. Ao retornar com `eret`, o processador sai do *exception level*;

(5) a instrução causadora é buscada novamente e desta vez o mapeamento da TP está na TLB;

(6) se o mapeamento da causadora original (o primeiro TLB-refill) ainda está na TLB, a causadora é buscada pela segunda vez e finalmente encontra seu mapeamento na TLB e completa;

(7) se o mapeamento da causadora original (o primeiro TLB-refill) não está na TLB, então algum outro elemento da TLB é sobrescrito com o mapeamento para a causadora original, que é buscada pela terceira vez e finalmente encontra seu mapeamento na TLB e completa.

Se, na segunda excessão. o tratador detecta que a página não está mapeada em memória física, a página faltante deve ser carregada da memória secundária, o que implicaria numa troca de contexto. Ao voltar a executar, o processo retorna para a instrução causadora, que sofre uma nova excessão TLB-refill. Com a página carregada e mapeada, o tratador copia o elemento da TP para a TLB e a execução prossegue.

Acredito que agora seja uma boa hora para reler esta seção desde o início.

espaço em branco proposital

3.8 *Double Fault*

Habilite este teste 'ligando' somente `#define DOUBLE_FAULT` no topo do arquivo.

Este programa provoca uma excessão de "falta na TP" durante o tratamento de uma TLB-refill.

(a) No início do teste, o mapeamento da página onde ocorrerá a escrita é expurgado da TLB, para garantir que a TP será consultada quando ocorrer o *store* e a TLB-refill;

(b) o mapeamento da página onde está a tabela de páginas é removido da TLB. *No tratamento da TLB-refill do store, ocorrerá uma* segunda *falta na TLB, quando TP for consultada*;

(c) a excessão é um TLB-store porque provocada por uma escrita; o tratador é essencialmente o mesmo que para uma TLB-load, e a mesma rotina é usada para escritas e leituras.

Programa 5: Provoca uma *double fault*: TLB-refill seguida de TLB-store.

```
// 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");
}

// point to the base of the page table
//   and remove the TLB entry which points to the PT
walker = (int *)(x_DATA_BASE_ADDR + (x_DATA_MEM_SZ/2));
if ( TLB_purge((void *)walker) == 0 ) {
  print_str("\tPT␣purged␣from␣TLB\n");
} else {
  print_str("\twtf?\n");      // this ought not to ever happen
}

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

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

Veja em `include/handlers.s`, a rotina `handle_TLBload`. Ao teste:

(1) a primeira tentativa de escrita causa uma falta na TLB, e um TLB-refill. O endereço da causadora é armazenado em EPC;

(2) o tratamento da TLB-refill causa uma nova excessão porque a tabela de páginas não está mapeada na TLB. Esta segunda excessão ocorre 'dentro' da primeira, cujo tratamento não foi completado;

(3) o controle desvia diretamente para o tratador geral de excessões (`_excp_0180`), e de lá, para `handle_TLBL`. No protoOS, a mesma rotina é usada para tratar *loads* e *stores*;

(4) o tratador da TLB{L,S} verifica se a excessão é mesmo uma *double fault* (endereço causador é na TP) e recarrega o mapeamento da TP em `TLB[4]`;

(5) o endereço em EPC não foi alterado em `handle_TLBL`; quando retornar do tratador geral, retorna para o *store* apontado por EPC;

(6) o *store* é buscado pela segunda vez, o que causa uma segunda TLB-refill;

(7) a segunda TLB-refill completa porque o mapeamento da TP está na TLB;

(8) depois que seu mapeamento é carregado na TLB, o *store* é buscado pela terceira vez, e agora sua execução completa porque todos os mapeamentos relevantes estão na TLB.

4 Da tarefa

Sua tarefa é fazer análises similares às das Seções 3.6 e 3.8. Você deve seguir a execução destes programas com `gtkwave` e anotar a sequência de eventos dos programas nas Seções 4.1 e 4.2.

4.1 Violação de Proteção

Habilite este teste 'ligando' somente `#define PROT_VIOL` no topo do arquivo.

Programa 6: Violação de proteção: escrita numa página protegida.

```
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\tpurged\n");
} else {
  print_str("\t\tTLB␣miss\n");
}

// change a PT element so it is mapped but NON-writable
new_value = ( ((x_DATA_BASE_ADDR + PG_NUM*4096)>>12) <<6) | 0b000011;
PT_update( (int *)walker, 0, new_value);                          // d=0
new_value = 0x00000001;                          // NON-writable, mapped
PT_update( (int *)walker, 1, new_value);


// cause a protection violation
*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");
}
```

espaço em branco proposital

4.2 Falta na Segmentação

Habilite este teste 'ligando' somente *#define SEG_FAULT* no topo do arquivo.

Programa 7: Falta na Segmentação: referência à página não mapeada.

```
// pick a page that is not mapped, at an address above the PT
#define PG_UNMAPPED 20
walker = (int *)(x_DATA_BASE_ADDR + PG_UNMAPPED*4096);

// remove V_addr from the TLB, to ensure the PT will be searched
if ( TLB_purge((void *)walker) == 0 ) {
  print_str("\n\tpurged\n");
} else {
  print_str("\n\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;
PT_update( (int *)walker, 0, new_value);                    // d=0
PT_update( (int *)walker, 1, 0);               // mark as unmapped
new_value =
  (((x_DATA_BASE_ADDR + (PG_UNMAPPED+1)*4096)>>12) <<6) | 0b000011;
PT_update( (int *)walker, 2, new_value);                    // d=0
PT_update( (int *)walker, 3, 0);               // mark as unmapped


// cause a segmentation fault
*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");
}
```

espaço em branco proposital

Referências

[PH14] David A Patterson and John L Hennessy. *Computer Organization & Design: The Hardware/-Software Interface*. Morgan Kaufmann, 5th edition, 2014.

[Swe07] Dominic Sweetman. *See MIPS Run – Linux*. Morgan Kaufmann, 2nd edition, 2007. ISBN 0120884216.

EOF

Histórico das Revisões:
18out2016: tamanho da TP, gramática, ajustes no texto;
12mai2016: primeira versão.