Tratador de interrupção em MIPS

Anderson Cristiano Sassaki Gonçalves

11 de fevereiro de 2025

ÍNDICE

1 Introdução **2**
2 Tratadores de interrupção **4**
3 Implementação em MIPS **5**
3.1 Entrando no tratador 6
3.2 Detalhando as instruções 7
4 Dentro do tratador **11**
4.1 Detalhes estruturais do coprocessador 0 11
4.2 Registradores CP0 11

1 Introdução

Exceções e interrupções são eventos que alteram o o fluxo normal da execução de instruções de um programa. Apesar de serem semelhantes, elas têm diferenças que podem ser entendidas ao considerarmos suas origens.

- Uma interrupção é um evento assíncrono, não associado ao trabalho realizado pelo processador no instante em que ocorre, geralmente solicitada por dispositivos externos.

- Uma exceção é um evento síncrono, diretamente associado ao trabalho realizado pelo processador no instante em que ocorre, causada pelo próprio programa que estava sendo executado.

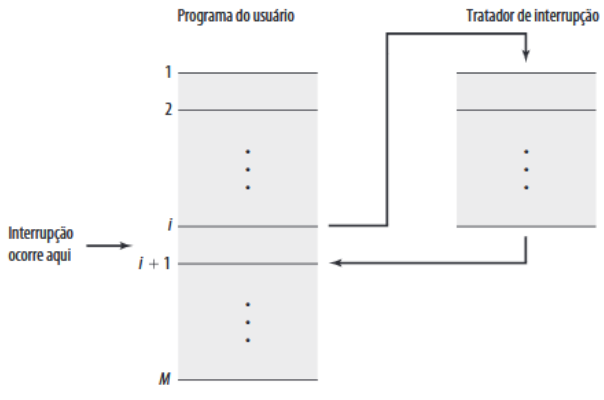
Trap	Interrupt
It's a signal emitted by a user program	It's a signal emitted by a hardware device
Synchronous process	Asynchronous process
Can occur only from software device	Can occur from a hardware or a software device
Only generated by a user program instruction	Generated by an OS and user program instruction
Traps are subset of interrupts	Interrupts are superset of traps
Execute a specific functionality in the OS and gives the control to the trap handler	Force the CPU to trigger a specific interrupt handler routine

Figura 1: Diferença entre exceção e interrupção

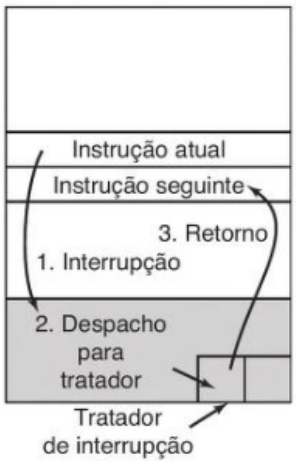
Exceções, algumas vezes chamadas de *traps* (Figura 1), podem ser acionadas por algum durante a execução do programa. Casos mais comum incluem overflow aritmético, divisão por zero ou acesso inválido à memória. Esses erros acontecem de forma síncrona com a execução de um programa. Uma vez resolvidas, o processador retorna à sua atividade anterior.

Interrupções por outro lado, acontecem de forma assíncrona e podem chegar à qualquer instante. Para exemplificar, considere o caso de dispositivos de E/S. No método mais simples, um driver de dispositivo seria acionado e permaneceria em um laço, verificando se o dispositivo terminou a operação. Neste método, a CPU se mantém ocupada o tempo todo até o término da operação, um processo conhecido como espera ocupada.

No entanto, uma abordagem mais eficiente é gerar uma interrupção ao término da operação, sinalizando para o processador que o trabalho foi concluído. Dessa forma, a CPU pode ser alocada para realizar outras tarefas enquanto aguarda o término da operação.



(a) Diagrama A



(b) Diagrama B

Figura 2: Fluxo de controle de uma interrupção

Os diagramas A e B presentes na Figura 2 apresentam um fluxo resumido de uma interrupção. Baseado neles podemos descrever um cenário comum de interrupção e um fluxo mais próximo da implementação real.

Considere um dispositivo de E/S como exemplo. Inicialmente, o dispositivo recebe algum trabalho através do driver e de seu controlador. Assim que a operação é concluída, o controlador gera uma interrupção para o processador. Se o sistema estiver pronto para lidar com a interrupção, o tratador de interrupções recebe o sinal e identifica qual dispositivo fez a solicitação.

Com a interrupção aceita, o contador de programa (PC), e a palavra de estado do programa (PSW) são armazenados na pilha, e a CPU entra em modo *kernel*. O número do dispositivo informado é utilizado como índice no vetor de interrupções para localizar o endereço do tratador específico para aquele dispositivo.

Após concluir seu trabalho, o tratador restaura o estado original do processador e retoma a execução do programa a partir da instrução interrompida.

Apesar das diferenças entre interrupções e exceções, o mecanismo utilizado pelos computadores geralmente é o mesmo. Ao chegar ao tratador, o evento é identificado como interrupção ou exceção e processado conforme necessário.

Na próxima seção, exploraremos com mais detalhes como os tratadores de interrupção lidam com esses eventos, desde sua identificação até a execução das ações necessárias para garantir a continuidade do sistema.

2 Tratadores de interrupção

Processar interrupções e exceções não é uma tarefa tão simples, há bem mais trabalho envolvido para o SO. No momento, examinaremos com mais detalhes o papel do processador na E/S controlada por interrupção. O surgimento de tal interrupção dispara uma cadeia de eventos, tanto no *hardware* quanto no *software*. Para lidar com tal evento a seguinte sequência de passos é executada:

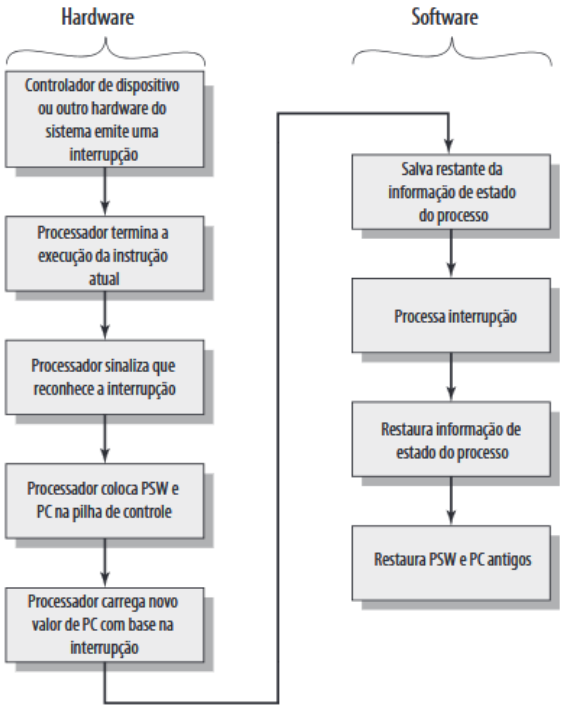


Figura 3: Processamento de interrupção simples

1. O processador termina a execução da instrução atual.
2. Salvar informações necessárias para retornar ao programa atual no ponto de interrupção. (PSW e PC + 4)
3. Estabelecer um contexto para a rotina de tratamento de interrupção.
4. Estabelecer uma pilha para a rotina de tratamento da interrupção.
5. Confirmar o recebimento ao controlador de interrupções.
6. Executar a rotina de tratamento de interrupção.
7. Escolher qual processo executar em seguida.
8. Carregar os registradores do novo processo, incluindo sua PSW.
9. Começar a execução do novo processo.

3 Implementação em MIPS

Com todo o material teórico abordado nas seções anteriores, ainda não somos capazes de transformar o conhecimento em prática e implementar um tratador de interrupções na arquitetura MIPS. Para alcançar tal feito, precisamos entender as nuances da arquitetura MIPS em relação ao tratamento de exceções, tais como suas características únicas de registradores especiais, vetores de interrupções e coprocessadores utilizados para lidar com os eventos descritos.

No capítulo 4 do livro **MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture** temos a descrição de alguns aspectos do modelo de programação da CPU MIPS. A primeira descrição relevante para o projeto é a do coprocessador 0 (CP0), visto na Figura 4, que está incorporado no chip da CPU e dá suporte ao sistema de memória virtual e tratamento de exceções. CP0 também é conhecido como Coprocessador de Controle de Sistema.

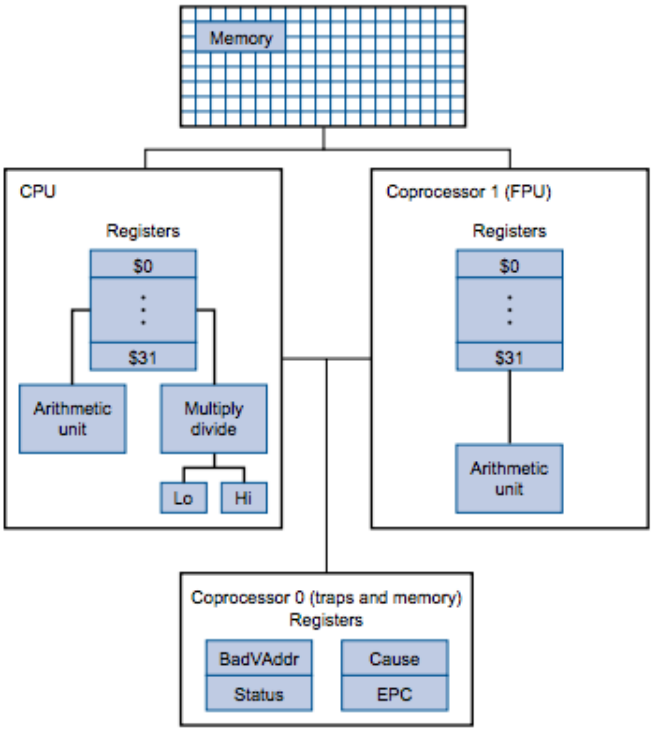
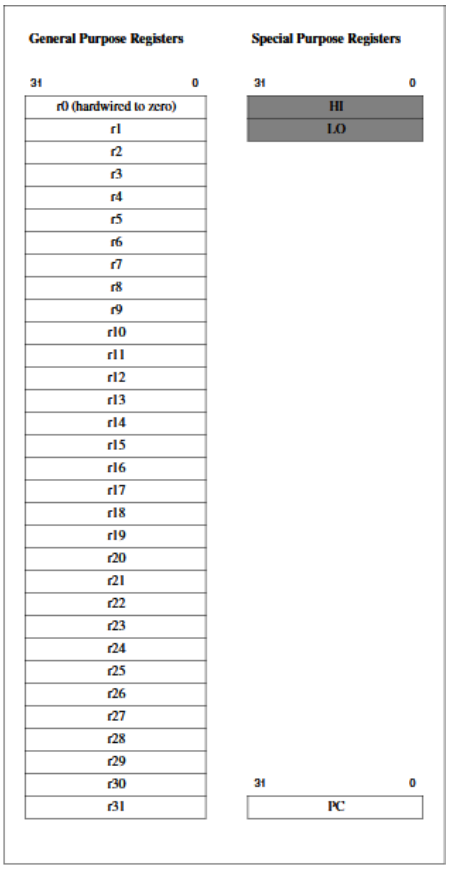


Figura 4: Organização da CPU

Além disso, também está presente a descrição de outros elementos básicos que também fazem parte do processador. Tais como o coprocessador 1, que é reservado para operações de ponto flutuante. Os 32 registradores de propósito geral e três de propósito especial, o PC, que armazena o contador de programa e um par de registradores HI e LO que armazenam os resultados das operações de multiplicação e divisão de inteiros podem ser vistos na Figura 5.

Vale notar que dos 32 registradores de propósito geral, dois tem funções designadas. O r0 é conectado diretamente a um valor zero, ele pode ser usado como registrador alvo para qualquer instrução cujo resultado deva ser descartado e pode ser usado como fonte quando um valor zero for necessário. O r31 é o registrador de destino usado pelas instruções de ramificação/salto e link de chamada de procedimento (por exemplo, Jal) sem ser explicitamente especificado na palavra de instrução. Caso contrário, r31 é usado como registro normal



(a) Registradores de propósito geral

Table 2-7: CP0 Registers

Register Number	Register Name	Function
0-6	Reserved	Reserved in the M4K® Microprocessor core.
7	HWREna	Enables access via the RDHWR instruction to selected hardware registers in Non-privileged mode.
8	BadVAddr	Reports the address for the most recent address-related exception.
9	Count	Processor cycle count.
10	Reserved	Reserved in the M4K® Microprocessor core.
11	Compare	Core timer interrupt control.
12	Status	Processor status and control.
	IntCtl	Interrupt control of vector spacing.
	SRSCtl	Shadow register set control.
	SRSMap	Shadow register mapping control.
13	Cause	Describes the cause of the last exception.
14	EPC	Program counter at last exception.
15	PRID	Processor identification and revision
	Ebase	Exception base address of exception vectors.
16	Config	Configuration register.
	Config1	Configuration register 1.
	Config2	Configuration register 2.
	Config3	Configuration register 3.
17-22	Reserved	Reserved in the M4K® Microprocessor core.
23	Debug	Debug control/exception status.
	TraceControl	EJTAG trace control.
	TraceControl2	EJTAG trace control 2.
	UserTraceData	User format type trace record trigger.
	TraceBPC	Control tracing using an EJTAG Hardware breakpoint.
	Debug2	Debug control/exception status 1.
24	DEPC	Program counter at last debug exception.
25-29	Reserved	Reserved in the M4K® Microprocessor core.
30	ErrorEPC	Program counter at last error.
31	DeSAVE	Debug handler scratchpad register.

(b) Registradores do coprocessador 0

Figura 5: Tabela de registradores

3.1 Entrando no tratador

Além das instruções vistas em aula, o projeto fará uso de instruções especiais, desenvolvidas especificamente para o tratamento de interrupção. Essas instruções transferem o controle para um tratador de interrupções no software no modo kernel. A arquitetura separa as instruções em dois conjuntos: condicionais e incondicionais. Abaixo segue a descrição das instruções e suas classificações.

Table 5.22 System Call and Breakpoint Instructions

Mnemonic	Instruction	Defined in MIPS ISA
BREAK	Breakpoint	MIPS32
SYSCALL	System Call	MIPS32

Figura 6: Instruções de chamada de sistema e breakpoint, causam exceções incondicionais.

Table 5.23 Trap-on-Condition Instructions Comparing Two Registers

Mnemonic	Instruction	Defined in MIPS ISA
TEQ	Trap if Equal	MIPS32
TGE	Trap if Greater Than or Equal	MIPS32
TGEU	Trap if Greater Than or Equal Unsigned	MIPS32
TLT	Trap if Less Than	MIPS32
TLTU	Trap if Less Than Unsigned	MIPS32
TNE	Trap if Not Equal	MIPS32

Figura 7: Instruções de trap, causam exceções condicionais, baseadas no resultado de uma comparação

Table 5.24 Trap-on-Condition Instructions Comparing an Immediate Value

Mnemonic	Instruction	Defined in MIPS ISA
TEQI	Trap if Equal Immediate	MIPS32 Removed in Release 6
TGEI	Trap if Greater Than or Equal Immediate	MIPS32 Removed in Release 6
TGEIU	Trap if Greater Than or Equal Immediate Unsigned	MIPS32 Removed in Release 6

Mnemonic	Instruction	Defined in MIPS ISA
TLTI	Trap if Less Than Immediate	MIPS32 Removed in Release 6
TLTIU	Trap if Less Than Immediate Unsigned	MIPS32 Removed in Release 6
TNEI	Trap if Not Equal Immediate	MIPS32 Removed in Release 6

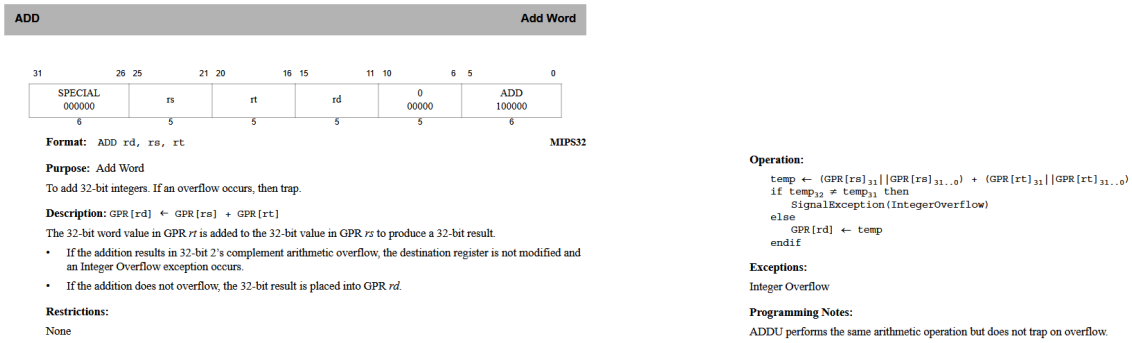
Figura 8: Instruções de trap, causam exceções condicionais, baseadas no resultado de uma comparação com um valor imediato

Vale destacar que funções de load/store não são definidas para o CP0, para escrever e ler dos registradores é necessário utilizar as instruções *move to and from*.

3.2 Detalhando as instruções

Na subseção anterior vimos algumas funções que podem ser utilizadas para transferir o controle da execução ao nosso tratador e entrar no modo kernel. Agora, veremos com mais detalhes o funcionamento de algumas das instruções que serão utilizadas na implementação do nosso tratador, seja para causar ou para tratar uma exceção. Todas as definições foram extraidas do livro **MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual**.

- **ADD** Realiza a soma de dois inteiros de 32 bits. Note que, ao detectar um overflow, a operação automaticamente sinaliza uma exceção de overflow. Algumas operações fazem essa sinalização de forma automática e a partir disso conseguimos entrar no tratador e resolver o problema. Semelhante a essa operação, na aula vimos a operação ADDU, a única diferença das duas é que a operação ADDU não realiza a etapa de verificar o overlflow.



ADD — **Add Word**

31–26	25–21	20–16	15–11	10–6	5–0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format: `ADD rd, rs, rt` **MIPS32**

Purpose: Add Word

To add 32-bit integers. If an overflow occurs, then trap.

Description: `GPR[rd] ← GPR[rs] + GPR[rt]`

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result.

- If the addition results in 32-bit 2's complement arithmetic overflow, the destination register is not modified and an Integer Overflow exception occurs.
- If the addition does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

None

Operation:

```
temp ← (GPR[rs]31||GPR[rs]31..0) + (GPR[rt]31||GPR[rt]31..0)
if temp32 ≠ temp31 then
   SignalException(IntegerOverflow)
else
   GPR[rd] ← temp
endif
```

Exceptions:

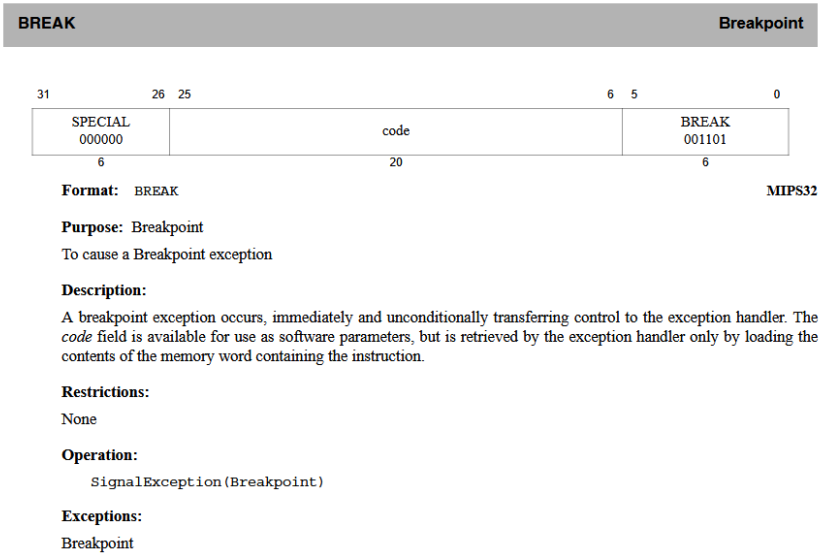
Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but does not trap on overflow.

Figura 9: Especificação da instrução ADD

- **BREAK** Causa uma exceção de *Breakpoint* incondicional e instantaneamente, transferindo o controle para o tratador de exceção.



BREAK — **Breakpoint**

31–26	25–6	5–0
SPECIAL 000000	code	BREAK 001101
6	20	6

Format: `BREAK` **MIPS32**

Purpose: Breakpoint

To cause a Breakpoint exception

Description:

A breakpoint exception occurs, immediately and unconditionally transferring control to the exception handler. The *code* field is available for use as software parameters, but is retrieved by the exception handler only by loading the contents of the memory word containing the instruction.

Restrictions:

None

Operation:

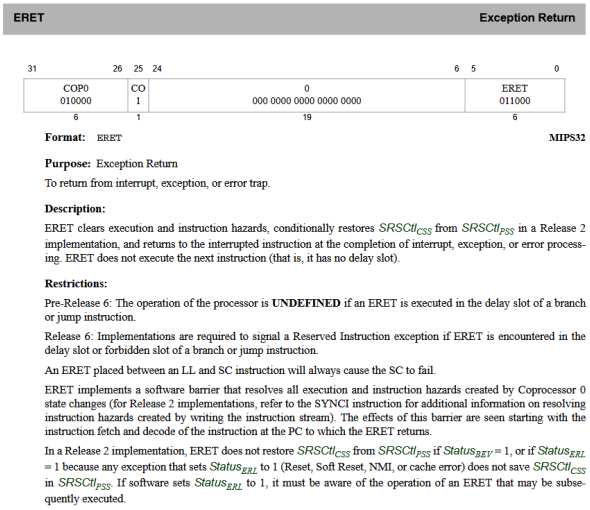
```
SignalException(Breakpoint)
```

Exceptions:

Breakpoint

Figura 10: Especificação da instrução BREAK

- **ERET** *Exception Return.* Retorna do tratador, devolvendo o controle da execução ao programa previamente interrompido. Efetivamente, restaura o PC com o valor armazenado no EPC (Exception Program Counter), restaura o Status (bit EXL = 0) e permite que o programa continue de onde havia parado.



ERET — **Exception Return**

31–26	25	24–6	5–0
COP0 010000	CO 1	0 000 0000 0000 0000 0000	ERET 011000
6	1	19	6

Format: `ERET` — **MIPS32**

Purpose: Exception Return

To return from interrupt, exception, or error trap.

Description:

ERET clears execution and instruction hazards, conditionally restores $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ in a Release 2 implementation, and returns to the interrupted instruction at the completion of interrupt, exception, or error processing. ERET does not execute the next instruction (that is, it has no delay slot).

Restrictions:

Pre-Release 6: The operation of the processor is **UNDEFINED** if an ERET is executed in the delay slot of a branch or jump instruction.

Release 6: Implementations are required to signal a Reserved Instruction exception if ERET is encountered in the delay slot or forbidden slot of a branch or jump instruction.

An ERET placed between an LL and SC instruction will always cause the SC to fail.

ERET implements a software barrier that resolves all execution and instruction hazards created by Coprocessor 0 state changes (for Release 2 implementations, refer to the SYNCI instruction for additional information on resolving instruction hazards created by writing the instruction stream). The effects of this barrier are seen starting with the instruction fetch and decode of the instruction at the PC to which the ERET returns.

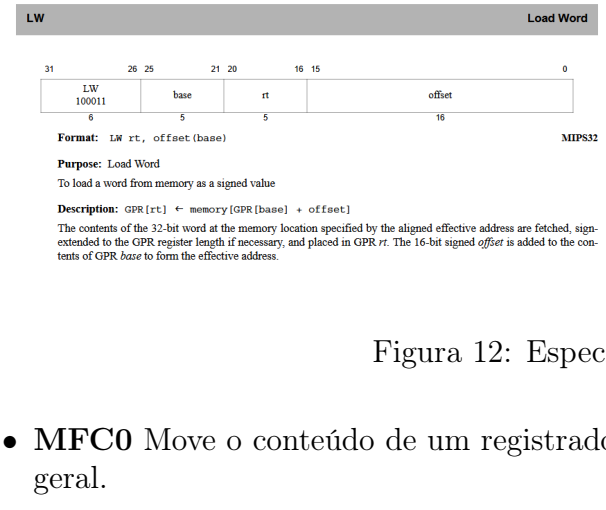
In a Release 2 implementation, ERET does not restore $SRSCtl_{CSS}$ from $SRSCtl_{PSS}$ if $Status_{BEV} = 1$, or if $Status_{ERL} = 1$ because any exception that sets $Status_{ERL}$ to 1 (Reset, Soft Reset, NMI, or cache error) does not save $SRSCtl_{CSS}$ in $SRSCtl_{PSS}$. If software sets $Status_{ERL}$ to 1, it must be aware of the operation of an ERET that may be subsequently executed.

Operation:

```
if StatusERL = 1 then
   temp ← ErrorEPC
   StatusERL ← 0
else
   temp ← EPC
   StatusEXL ← 0
   if (ArchitectureRevision() ≥ 2) and (SRSCtlHSS > 0) and (StatusBEV = 0) then
      SRSCtlCSS ← SRSCtlPSS
   endif
endif
if IsMIPS16Implemented() | (Config3ISA > 0) then
   PC ← temp31..1 || 0
   ISAMode ← temp0
else
   PC ← temp
endif
LLbit ← 0
ClearHazards()
```

Figura 11: Especificação da instrução ERET

- **LW** Carrega uma palavra da memória. Pode causar diversas exceções, focaremos no caso de acesso a um endereço desalinhado (não multiplo de 4) e acesso a um endereço inválido (endereço fora dos limites, por exemplo).



LW — **Load Word**

31–26	25–21	20–16	15–0
LW 100011	base	rt	offset
6	5	5	16

Format: `LW rt, offset(base)` — **MIPS32**

Purpose: Load Word

To load a word from memory as a signed value

Description: `GPR[rt] ← memory[GPR[base] + offset]`

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 allows hardware to provide address misalignment support in lieu of requiring natural alignment.

Note: The pseudocode is not completely adapted for Release 6 misalignment support as the handling is implementation dependent.

Operation:

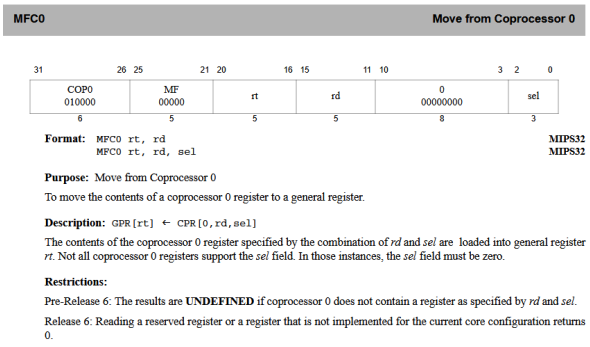
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (CCA, WORD, pAddr, vAddr, DATA)
GPR[rt]  memword
```

Exceptions:

TLB Refill, TLB Invalid, Bus Error, Address Error, Watch

Figura 12: Especificação da instrução LW

- **MFC0** Move o conteúdo de um registrador do coprocessador 0 para um registrador de propósito geral.



MFC0 — **Move from Coprocessor 0**

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MF 00000	rt	rd	0 00000000	sel
6	5	5	5	8	3

Format: `MFC0 rt, rd` — **MIPS32**
`MFC0 rt, rd, sel` — **MIPS32**

Purpose: Move from Coprocessor 0

To move the contents of a coprocessor 0 register to a general register.

Description: `GPR[rt] ← CPR[0,rd,sel]`

The contents of the coprocessor 0 register specified by the combination of *rd* and *sel* are loaded into general register *rt*. Not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be zero.

Restrictions:

Pre-Release 6: The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

Release 6: Reading a reserved register or a register that is not implemented for the current core configuration returns 0.

Operation:

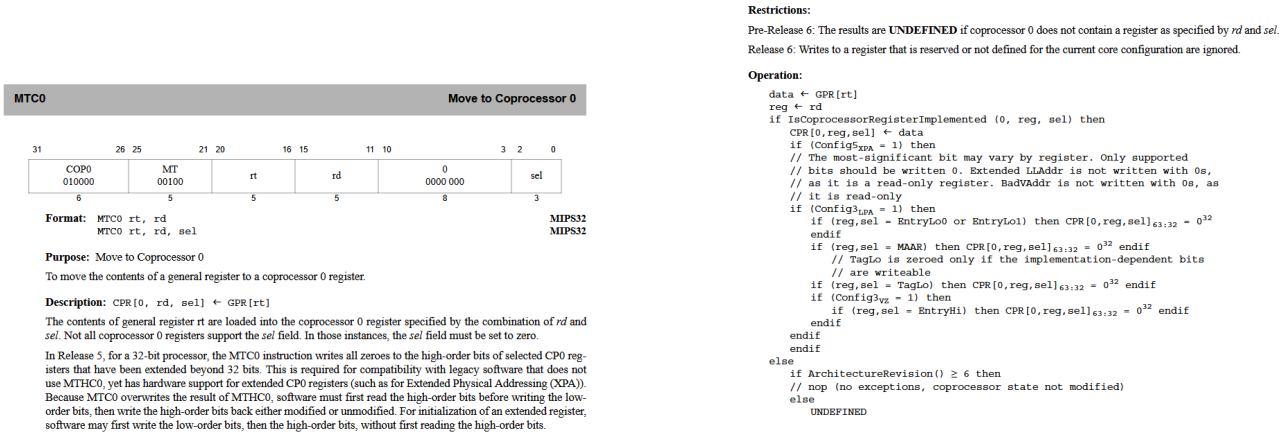
```
reg = rd
if IsCoprocessorRegisterImplemented(0, reg, sel) then
   data ← CPR[0, reg, sel]
   GPR[rt] ← data
else
   if ArchitectureRevision() ≥ 6 then
      GPR[rt] ← 0
   else
      UNDEFINED
   endif
endif
```

Exceptions:

Coprocessor Unusable, Reserved Instruction

Figura 13: Especificação da instrução MFC0

- **MTC0** Move o conteúdo de um registrador de propósito geral para um registrador do copressador 0.



MTC0 — **Move to Coprocessor 0**

31–26	25–21	20–16	15–11	10–3	2–0
COP0 010000	MT 00100	rt	rd	0 0000 000	sel
6	5	5	5	8	3

Format: `MTC0 rt, rd` **MIPS32**
`MTC0 rt, rd, sel` **MIPS32**

Purpose: Move to Coprocessor 0

To move the contents of a general register to a coprocessor 0 register.

Description: `CPR[0, rd, sel] ← GPR[rt]`

The contents of general register rt are loaded into the coprocessor 0 register specified by the combination of *rd* and *sel*. Not all coprocessor 0 registers support the *sel* field. In those instances, the *sel* field must be set to zero.

In Release 5, for a 32-bit processor, the MTC0 instruction writes all zeroes to the high-order bits of selected CP0 registers that have been extended beyond 32 bits. This is required for compatibility with legacy software that does not use MTHC0, yet has hardware support for extended CP0 registers (such as for Extended Physical Addressing (XPA)). Because MTC0 overwrites the result of MTHC0, software must first read the high-order bits before writing the low-order bits, then write the high-order bits back either modified or unmodified. For initialization of an extended register, software may first write the low-order bits, then the high-order bits, without first reading the high-order bits.

Restrictions:

Pre-Release 6: The results are **UNDEFINED** if coprocessor 0 does not contain a register as specified by *rd* and *sel*.

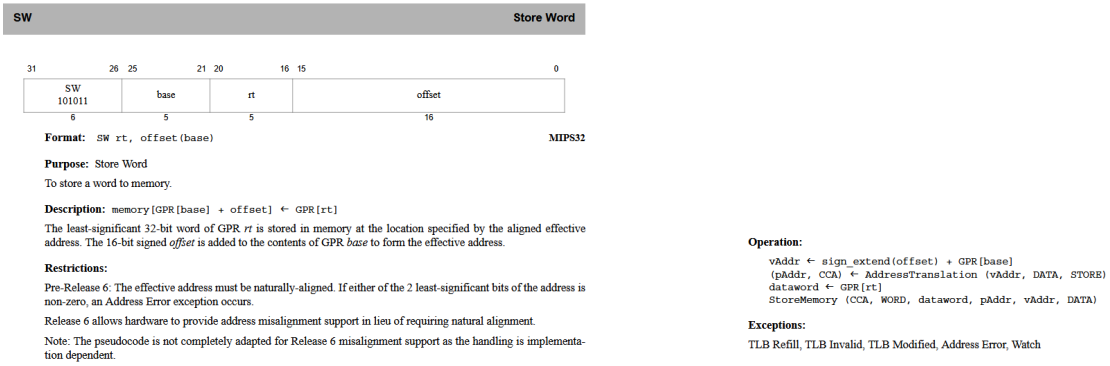
Release 6: Writes to a register that is reserved or not defined for the current core configuration are ignored.

Operation:

```
data ← GPR[rt]
reg ← rd
if IsCoprocessorRegisterImplemented (0, reg, sel) then
   CPR[0,reg,sel] ← data
   if (Config5_XPA = 1) then
   // The most-significant bit may vary by register. Only supported
   // bits should be written 0. Extended LLAddr is not written with 0s,
   // as it is a read-only register. BadVAddr is not written with 0s, as
   // it is read-only
   if (Config3_LPA = 1) then
      if (reg,sel = EntryLo0 or EntryLo1) then CPR[0,reg,sel]63:32 = 0^32
      endif
      if (reg,sel = MAAR) then CPR[0,reg,sel]63:32 = 0^32 endif
         // TagLo is zeroed only if the implementation-dependent bits
         // are writeable
      if (reg,sel = TagLo) then CPR[0,reg,sel]63:32 = 0^32 endif
      if (Config3_VZ = 1) then
         if (reg,sel = EntryHi) then CPR[0,reg,sel]63:32 = 0^32 endif
      endif
   endif
   endif
else
   if ArchitectureRevision() ≥ 6 then
   // nop (no exceptions, coprocessor state not modified)
   else
      UNDEFINED
```

Figura 14: Especificação da instrução MTC0

- **SW** Armazena uma palavra na memória. Pode causar as mesmas exceções que LW.



SW — **Store Word**

31–26	25–21	20–16	15–0
SW 101011	base	rt	offset
6	5	5	16

Format: `SW rt, offset(base)` **MIPS32**

Purpose: Store Word

To store a word to memory.

Description: `memory[GPR[base] + offset] ← GPR[rt]`

The least-significant 32-bit word of GPR *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

Pre-Release 6: The effective address must be naturally-aligned. If either of the 2 least-significant bits of the address is non-zero, an Address Error exception occurs.

Release 6 allows hardware to provide address misalignment support in lieu of requiring natural alignment.

Note: The pseudocode is not completely adapted for Release 6 misalignment support as the handling is implementation dependent.

Operation:

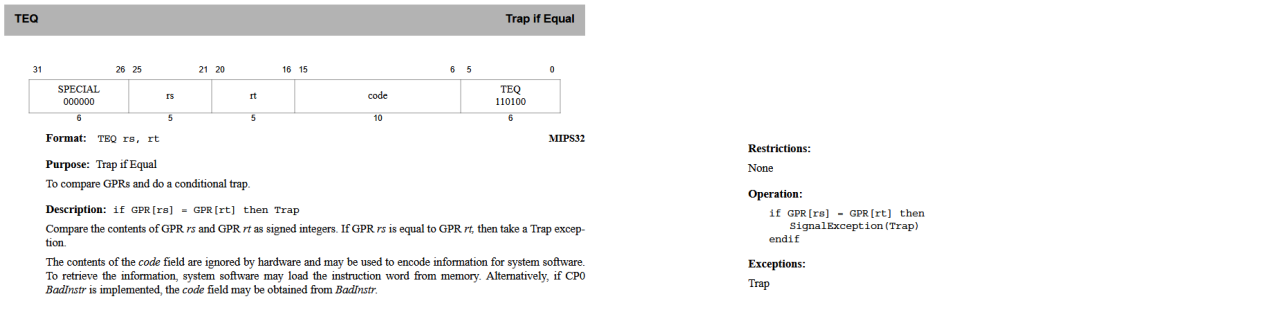
```
vAddr ← sign_extend(offset) + GPR[base]
(pAddr, CCA) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (CCA, WORD, dataword, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid, TLB Modified, Address Error, Watch

Figura 15: Especificação da instrução SW

- **TEQ** Compara os conteúdos de dois registradores de propósito geral e realiza um *Trap* condicional.



TEQ — **Trap if Equal**

31–26	25–21	20–16	15–6	5–0
SPECIAL 000000	rs	rt	code	TEQ 110100
6	5	5	10	6

Format: `TEQ rs, rt` **MIPS32**

Purpose: Trap if Equal

To compare GPRs and do a conditional trap.

Description: `if GPR[rs] = GPR[rt] then Trap`

Compare the contents of GPR *rs* and GPR *rt* as signed integers. If GPR *rs* is equal to GPR *rt*, then take a Trap exception.

The contents of the *code* field are ignored by hardware and may be used to encode information for system software. To retrieve the information, system software may load the instruction word from memory. Alternatively, if CP0 *BadInstr* is implemented, the *code* field may be obtained from *BadInstr*.

Restrictions:

None

Operation:

```
if GPR[rs] = GPR[rt] then
   SignalException(Trap)
endif
```

Exceptions:

Trap

Figura 16: Especificação da instrução MFC0

4 Dentro do tratador

Uma vez compreendidos os conceitos por trás de um tratador de exceção, os detalhes que fazem chegar nele efetivamente e as instruções que serão utilizadas, nos resta apenas mais uma área a ser analizada antes de concluirmos a análise da implementação em MIPS.

Todos os conceitos vistos até agora nos dão detalhes de causas de uma interrupção, fluxo de execução quando ocorre uma interrupção, instruções que podem ser utilizadas para gerar uma interrupção, ou até mesmo erros que podem acontecer e que ativarão uma exceção automaticamente. Aprendemos a como um programa deve se comportar ao detectar uma interrupção. Entretanto, nada nos diz como agir uma vez que entramos no tratador. Nesta seção, detalharemos com base no livro **MIPS32™ Architecture For Programmers Volume III: The MIPS32™ Privileged Resource Architecture** os detalhes que acontecem dentro do tratador, e como de fato tratar as interrupções que ocorreram.

4.1 Detalhes estruturais do coprocessador 0

O CP0 fornece uma abstração das funções necessárias para suportar um SO: tratamento de exceções, gerenciamento de memória, e controle de recursos. A interface com o CP0 é feita por meio de diversas instruções, incluindo a capacidade de movver dados de e para os registradores do CP0 e funções espécificas que modificam o estado do CP0. Os registradores do CP0 e a interação com eles constituem grande parte da Arquitetura de Recurso Privilegiado (PRA).

A PRA MIPS requer dois modos de operação: Modo Usuário e Modo Kernel. Quando em modo usuario, o programador tem acesso a registradores da CPU e FPU que são providos pela ISA, além de acesso a um endereço de memória virtual. Quando em modo kernel, o programado do sistema tem acesso completo as capacidades do processador, incluindo a habilidade de alterar o mapeamento da memoria virtual, controlar o ambiente do sistema e trocar o contexto entre processos.

O processador está operando em modo kernel quando o bit DM no registrador *Debug* é zero. E qualquer um das condições abaixo é verdadeira:

- O campo KSU no registrador Status contem 2#00
- O bit EXL no registrador Status e 1
- O bit ERL no registrador Status e 1

O processador entra em modo kernel ao ligar, ou como resultado de uma interrupcao, exceção ou erro. O processador sai de modo kernel e entra no modo usuario quando todas as tres condicoes anteriores são falsas, normalmente como resultado de uma instrução ERET.

O processador está operando em modo usuario quando todas as condições abaixo são verdadeira:

- O campo KSU no registrador Status contem 2#10
- o bit DM no registrador *Debug* é zero
- Os bits EXL e ERL no registrador Status são ambos zero

4.2 Registradores CP0

Os registradores do CP0 fornecem uma interface entre a arquitetura de conjunto de instruções (ISA) e a PRA. Todos os registradores estão listados abaixo, seguidos de uma breve descrição dos registradores importantes para o tratador de exceção.

Table 6-1 Coprocessor 0 Registers in Numerical Order

Register Number	Sel	Register Name	Function	Reference	Compliance Level
0	0	Index	Index into the TLB array	Section 6.3 on page 41	Required (TLB MMU); Optional (others)
1	0	Random	Randomly generated index into the TLB array	Section 6.4 on page 42	Required (TLB MMU); Optional (others)
2	0	EntryLo0	Low-order portion of the TLB entry for even-numbered virtual pages	Section 6.5 on page 43	Required (TLB MMU); Optional (others)
3	0	EntryLo1	Low-order portion of the TLB entry for odd-numbered virtual pages	Section 6.5 on page 43	Required (TLB MMU); Optional (others)
4	0	Context	Pointer to page table entry in memory	Section 6.6 on page 45	Required (TLB MMU); Optional (others)
5	0	PageMask	Control for variable page size in TLB entries	Section 6.7 on page 46	Required (TLB MMU); Optional (others)
6	0	Wired	Controls the number of fixed (“wired”) TLB entries	Section 6.8 on page 47	Required (TLB MMU); Optional (others)
7	all		Reserved for future extensions		Reserved
8	0	BadVAddr	Reports the address for the most recent address-related exception	Section 6.9 on page 48	Required
9	0	Count	Processor cycle count	Section 6.10 on page 49	Required
9	6-7		Available for implementation dependent user	Section 6.11 on page 49	Implementation Dependent
10	0	EntryHi	High-order portion of the TLB entry	Section 6.12 on page 51	Required (TLB MMU); Optional (others)
11	0	Compare	Timer interrupt control	Section 6.13 on page 52	Required
11	6-7		Available for implementation dependent user	Section 6.14 on page 52	Implementation Dependent
12	0	Status	Processor status and control	Section 6.15 on page 53	Required
13	0	Cause	Cause of last general exception	Section 6.16 on page 58	Required
14	0	EPC	Program counter at last exception	Section 6.17 on page 61	Required
15	0	PRId	Processor identification and revision	Section 6.18 on page 62	Required
16	0	Config	Configuration register	Section 6.19 on page 63	Required
16	1	Config1	Configuration register 1	Section 6.20 on page 65	Required
16	2	Config2	Configuration register 2	Section 6.21 on page 69	Optional
16	3	Config3	Configuration register 3	Section 6.22 on page 70	Optional

16	6-7		Available for implementation dependent user	Section 6.23 on page 71	Implementation Dependent
17	0	LLAddr	Load linked address	Section 6.24 on page 72	Optional
18	0-n	WatchLo	Watchpoint address	Section 6.25 on page 73	Optional
19	0-n	WatchHi	Watchpoint control	Section 6.26 on page 74	Optional
20	0		XContext in 64-bit implementations		Reserved
21	all		Reserved for future extensions		Reserved
22	all		Available for implementation dependent use	Section 6.27 on page 76	Implementation Dependent
23	0	Debug	EJTAG Debug register	EJTAG Specification	Optional
24	0	DEPC	Program counter at last EJTAG debug exception	EJTAG Specification	Optional
25	0-n	PerfCnt	Performance counter interface	Section 6.30 on page 79	Recommended
26	0	ErrCtl	Parity/ECC error control and status	Section 6.31 on page 82	Optional
27	0-3	CacheErr	Cache parity error control and status	Section 6.32 on page 83	Optional
28	0	TagLo	Low-order portion of cache tag interface	Section 6.33 on page 84	Required (Cache)
28	1	DataLo	Low-order portion of cache data interface	Section 6.34 on page 85	Optional
29	0	TagHi	High-order portion of cache tag interface	Section 6.35 on page 86	Required (Cache)
29	1	DataHi	High-order portion of cache data interface	Section 6.36 on page 87	Optional
30	0	ErrorEPC	Program counter at last error	Section 6.37 on page 88	Required
31	0	DESAVE	EJTAG debug exception save register	EJTAG Specification	Optional

Figura 18: Registradores do CP0

BadVAddr: É um registrador de leitura que captura o endereço virtual mais recente que causou uma excecao de memoria.

Figure 6-8 BadVAddr Register Format

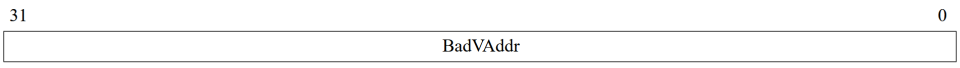


Table 6-11 BadVAddr Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
BadVAddr	31..0	Bad virtual address	R	Undefined	Required

Figura 19: Registrador BadVAddr

Status: É um registrador de leitura/escrita que contem o modo de operação, habilitação de interrupção e os estados de diagnostico do processador. Os campos desse registrador se combinam para criar

modos de operação para o processador.

Figure 6-12 Status Register Format

31 28	27	26	25	24	23	22	21	20	19	18	17 16	15 8	7	6	5	4	3	2	1	0
CU3..CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	Impl	IM7..IM0	KX	SX	UX	UM KSU	R0	ERL	EXL	IE

KSU	4..3	If Supervisor Mode is implemented, the encoding of this field denotes the base operating mode of the processor. See Chapter 3, "MIPS32 Operating Modes," on page 9 for a full discussion of operating modes. The encoding of this field is: Note: This field overlaps the UM and R0 fields, described below.	R/W	Undefined	Required if Supervisor Mode is implemented; Optional otherwise
UM	4	If Supervisor Mode is not implemented, this bit denotes the base operating mode of the processor. See Chapter 3, "MIPS32 Operating Modes," on page 9 for a full discussion of operating modes. The encoding of this bit is: **Encoding** / **Meaning** 0 — Base mode is Kernel Mode 1 — Base mode is User Mode Note: This bit overlaps the KSU field, described above.	R/W	Undefined	Required
R0	3	If Supervisor Mode is not implemented, this bit is reserved. This bit must be ignored on write and read as zero. Note: This bit overlaps the KSU field, described above.	R	0	Reserved
ERL	2	Error Level; Set by the processor when a Reset, Soft Reset, NMI or Cache Error exception are taken. **Encoding** / **Meaning** 0 — Normal level 1 — Error level When ERL is set: • The processor is running in kernel mode • Interrupts are disabled • The ERET instruction will use the return address held in ErrorEPC instead of EPC • The lower 2^{29} bytes of kuseg are treated as an unmapped and uncached region. See Section 4.6 on page 16. This allows main memory to be accessed in the presence of cache errors. The operation of the processor is **UNDEFINED** if the ERL bit is set while the processor is executing instructions from kuseg.	R/W	1	Required

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
EXL	1	Exception Level; Set by the processor when any exception other than Reset, Soft Reset, NMI or Cache Error exception are taken. **Encoding** / **Meaning** 0 — Normal level 1 — Exception level When EXL is set: • The processor is running in Kernel Mode • Interrupts are disabled. • TLB Refill exceptions use the general exception vector instead of the TLB Refill vector. • EPC and $Cause_{BD}$ will not be updated if another exception is taken	R/W	Undefined	Required
IE	0	Interrupt Enable: Acts as the master enable for software and hardware interrupts: **Encoding** / **Meaning** 0 — Interrupts are disabled 1 — Interrupts are enabled	R/W	Undefined	Required

Figura 20: Registrador Status

Cause: Primariamente descreve a causa da exceção mais recente. Seus campos tambem controlam requisições de interrupção de software e o vetor por onde as interrupções são enviadas. Quase todos seus campos são apenas para leitura.

Figure 6-13 Cause Register Format

31	30	29 28	27 24	23	22	21 16	15 8	7	6 2	1 0
BD	0	CE	0	IV	WP	0	IP7:IP0	0	Exc Code	0

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
WP	22	Indicates that a watch exception was deferred because $Status_{EXL}$ or $Status_{ERL}$ were a one at the time the watch exception was detected. This bit both indicates that the watch exception was deferred, and causes the exception to be initiated once $Status_{EXL}$ and $Status_{ERL}$ are both zero. As such, software must clear this bit as part of the watch exception handler to prevent a watch exception loop. Software should not write a 1 to this bit when its value is a 0, thereby causing a 0-to-1 transition. If such a transition is caused by software, it is **UNPREDICTABLE** whether hardware ignores the write, accepts the write with no side effects, or accepts the write and initiates a watch exception once $Status_{EXL}$ and $Status_{ERL}$ are both zero. If watch registers are not implemented, this bit must be ignored on write and read as zero.	R/W	Undefined	Required if watch registers are implemented
IP[7:2]	15..10	Indicates an external interrupt is pending: Encoding / Meaning: 15 Hardware interrupt 5, timer or performance counter interrupt; 14 Hardware interrupt 4; 13 Hardware interrupt 3; 12 Hardware interrupt 2; 11 Hardware interrupt 1; 10 Hardware interrupt 0	R	Undefined	Required
IP[1:0]	9..8	Controls the request for software interrupts: Encoding / Meaning: 9 Request software interrupt 1; 8 Request software interrupt 0	R/W	Undefined	Required
ExcCode	6..2	Exception code - see Table 6-17	R	Undefined	Required

Figura 21: Registrador Cause

EPC: É um registrador de leitura/escrita que contem o endereco no qual o processamento deve ser retomado assim que a exceção foi tratada. Todos os bits do registrador EPC são significativos e devem ser graváveis.

Figure 6-14 EPC Register Format

31	0
EPC	

Table 6-18 EPC Register Field Descriptions

Fields		Description	Read/Write	Reset State	Compliance
Name	Bits				
EPC	31..0	Exception Program Counter	R/W	Undefined	Required

Figura 22: Registrador EPC

ErrorEPC: É um registrador de leitura/escrita, similar ao EPC, exceto que o ErrorEPC é usado em exceções de erro. Todos os bits do registrador ErrorEPC são significativos e devem ser graváveis.

Figure 6-25 ErrorEPC Register Format

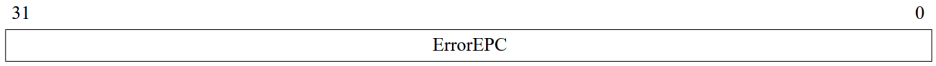


Table 6-30 ErrorEPC Register Field Descriptions

Fields		Description	Read/ Write	Reset State	Compliance
Name	Bits				
ErrorEPC	31..0	Error Exception Program Counter	R/W	Undefined	Required

Figura 23: Registrador ErrorEPC

O registrador Cause é o responsável por armazenar o código da exceção mais recente. Para descobrir qual foi a exceção devemos comparar o código armazenado em Cause com a lista de possiveis códigos, representada na tabela abaixo.

Table 6-17 Cause Register ExcCode Field

Exception Code Value		Mnemonic	Description
Decimal	Hexadecimal		
2	16#02	TLBL	TLB exception (load or instruction fetch)
3	16#03	TLBS	TLB exception (store)
4	16#04	AdEL	Address error exception (load or instruction fetch)
5	16#05	AdES	Address error exception (store)
6	16#06	IBE	Bus error exception (instruction fetch)
7	16#07	DBE	Bus error exception (data reference: load or store)
8	16#08	Sys	Syscall exception
9	16#09	Bp	Breakpoint exception
10	16#0a	RI	Reserved instruction exception
11	16#0b	CpU	Coprocessor Unusable exception
12	16#0c	Ov	Arithmetic Overflow exception
13	16#0d	Tr	Trap exception
14	16#0e	-	Reserved
15	16#0f	FPE	Floating point exception
16-17	16#10-16#11	-	Available for implementation dependent use
18	16#12	C2E	Reserved for precise Coprocessor 2 exceptions
19-21	16#13-16#15	-	Reserved
22	16#16	MDMX	Reserved for MDMX Unusable Exception in MIPS64 implementations.
23	16#17	WATCH	Reference to WatchHi/WatchLo address
24	16#18	MCheck	Machine check
25-29	16#19-16#1d	-	Reserved
30	16#1e	CacheErr	Cache error. In normal mode, a cache error exception has a dedicated vector and the Cause register is not updated. If EJTAG is implemented and a cache error occurs while in Debug Mode, this code is used to indicate that re-entry to Debug Mode was caused by a cache error.
31	16#1f	-	Reserved

Figura 24: Tabela de códigos de exceções

Com todo o conhecimento teórico adquirido até o momento, já se torna possível realizar a implementação de um tratador de interrupções simples que funciona para a arquitetura MIPS. Para desenvolver tal aplicação, utilizaremos o simulador MARS, que além de simular o conjunto de instruções, disponibiliza inumeras ferramentas que nos auxiliarão no processo de desenvolvimento.