Wyjątki i przerwania, oraz instrukcje wielocyklowe

na przykładzie procesora MIPS

Jan Mazur

Nomenklatura MIPSowa

Wyjątki (*exceptions)* - zdarzenia inne niż skoki zmieniające przepływ sterowania

Przerwania (*interrupts)* - wyjątki pochodzenia zewnętrznego (PIN w procesorze)

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Procedura obsługi przerwania (*Interrupt Service Routine; ISR*) - procedura obsługująca zdarzenie które wywołało wyjątek.

Przykłady wyjątków

- błąd strony
- odwołanie do niewyrównanej pamięci
- odwołanie do chronionej pamięci
- breakpoint
- I/O
- instrukcja trap
- przepełnienie całkowitoliczbowe
- anomalia w jednostce zmiennoprzecinkowej (FPU - *floating point unit*)
- próba wykonania niezidentyfikowanej instrukcji
- błąd sprzętu
- ...

Zastosowania

Początkowo do reagowania na niespodziewane zdarzenia z procesora, np. takie jak przepełnienie przy operacjach arytmetycznych.

Później ten sam mechanizm rozszerzony na obsługę urządzeń I/O.

W MIPS mechanizm wyjątków jest używany również przy starcie i restarcie procesora.

Związany z TLB, błędami cache, błędami pamięci.

Klasyfikacja wyjątków

Jeśli wyjątek występuje za każdym razem w tym samym miejscu programu, dysponując tymi samymi danymi i pamięcią, to jest on **synchroniczny**. (syscall, integer overflow)

Asynchroniczne wyjątki pochodzą od bytów zewnętrznych względem CPU i pamięci. (I/O, błąd HW)

Asynchroniczne wyjątki zazwyczaj można obsłużyć po wykonaniu obecnie przetwarzanej instrukcji. Zatem są łatwiejsze w obsłudze.

Klasyfikacja wyjątków

Wyjątki *explicite* wywoływane przez użytkownika (np. syscall) nazywamy **programowymi** (*user-requested*). Są to w pełni przewidywalne i pożądane zdarzenia, aczkolwiek wykorzystują mechanizm wyjątków do swoich celów.

Mogą zostać obsłużone po całkowitym wykonaniu instrukcji.

Wymuszone (*coerced*) wyjątki pochodzą od zdarzeń w sprzęcie, które nie są kontrolowane przez program. Nieprzewidywalne, **trudne** do obsługi, mogą wymagać obsługi w trakcie przetwarzania instrukcji.

Klasyfikacja wyjątków

Niemaskowalne przerwania (*NMI - Non-Maskable Interrupts*) - specjalne przerwania których nie można zignorować przez ustawienie odpowiednich bitów w masce przerwań. Zazwyczaj oznaczają nieodwracalnych błędów.

Maskowalne przerwania - przerwania które można zignorować odpowiednio ustawiają maskę bitową.

Używane również przy debugowaniu. Można np. zrzucić pamięć procesu, z którym coś jest nie tak.

Klasyfikacja wyjątków

Niektóre wyjątki muszą zostać obsłużone **w trakcie** wykonania danej instrukcji, a inne **pomiędzy** instrukcjami.

Wyjątki pojawiające się **w trakcie** wykonania instrukcji są zawsze synchroniczne, ponieważ powoduje je wykonanie danej instrukcji.

Wyjątki występujące **w trakcie** wykonania instrukcji są **trudniejsze** do obsługi, ponieważ instrukcja musi być zatrzymana i później wznowiona.

Klasyfikacja wyjątków

IO	async.	coerced	NMI	beetwen	resumable
syscall	sync.	user-requested	NMI	beetwen	resumable
breakpoint	sync.	user-requested	Maskable	beetwen	resumable
int overflow	sync.	coerced	Maskable	within	resumable
FPU	sync.	coerced	Maskable	within	resumable
page fault	sync.	coerced	NMI	within	resumable
misaligned	sync.	coerced	Maskable	within	resumable
mem viol.	sync.	coerced	NMI	within	resumable
undef. instr.	sync.	coerced	NMI	within	term
HW	async.	coerced	NMI	within	term

Tabela C.31, *Computer Architecture - A Quantitative Approach*

Trudności

- rozpoznanie wszystkich możliwych scenariuszy wyjątków
- wznawianie kontekstu wykonania
- wyjątki wymagające obsługi w czasie wykonania instrukcji
- związanie wyjątku z konkretną instrukcją
- co jeśli w procedurze obsługi wyjątku otrzymamy wyjątek?
- co jeśli późniejsza (w potoku) instrukcja spowoduje wyjątek wcześniej niż instrukcja wcześniejsza?
- problemy z potokowaniem
- instrukcje wielocyklowe dodatkowo komplikują

Poprawne i szybkie obsługiwanie wyjątków jest **TRUDNE**.

Koprocesor

Koprocesor to układ spełniający pewne wyspecjalizowane zadania. Może mieć różne stopnie swobody.

Jego instrukcje mogą być osadzone w instrukcjach CPU (FPU) bądź może on pracować niezależnie.

Często opcjonalne (FPU w Intel 8087).

Przetwarzanie audio, grafiki, sygnałów, łańcuchów znaków, szyfrowanie...

Koprocesor 0 w MIPS - CP0

Koprocesor systemowy - obowiązkowy. Stanowi międzymordzie między ISA , a PRA (*Privileged Resource Architecture).*

Pobieranie instrukcji i logika wykonania osadzone w CPU.

Dostępny tylko w trybie jądra i trybie debugowania.

Daje interfejs do **obsługi wyjątków**, zarządzania pamięcią, zarządzania cache, konfiguruje CPU, inne.

Manipulować CP0 można za pomocą instrukcji z opcode COP0.

```
mfc0    d,$n      # Move from coprocessor 0
mtc0    s, <n>    # Move to coprocessor 0
```

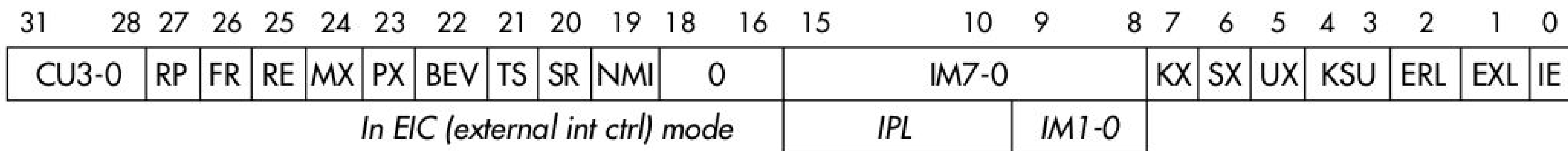
Specjalne instrukcje do niwelowania hazardów:

- EHB - usuwa hazardy wykonania
- ERET, DERET - usuwa hazardy instrukcji i wykonania

Najważniejsze rejestry CP0

- **Cause** - co spowodowało wyjątek bądź przerwanie?
- **Status** - konfiguracja CP0, maska przerwań, tryby pracy CPU.
- **EPC** (*exception program counter) -* adres następnej instrukcji po wyjątku
- **EBase** - adres wejścia do obsługi wyjątków
- **BadVAddr** - ostatni adres wirtualny, który spowodował wyjątki związane z pamięcią (złe wyrównanie, brak dostępu, poza kuseg)
- Count & Compare - przerwania cykliczne, czasowe, “zegar”
- NestedEPC - używany przy zagnieżdżonych wyjątkach
- SRSCtl &SRSMap - tzw. *shadow registers*
- Config1-3 - rejestry konfiguracji CPU, informacje o możliwościach
- IntCtl - między innymi podaje odległości między wektorami wejścia

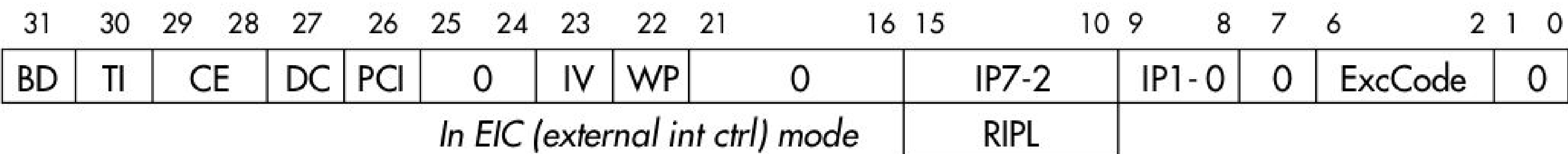
Status



BEV	Boot Entry Vectors. Używamy punktu wejścia do wyjątków z niekeszowanego i niezmapowanego* obszaru pamięci.
SR	Wystąpił Soft Reset.
NMI	Wystąpiło NMI.
IMx	Maska przerwań. 7-2 sprzętowe, 0-1 programowe
ERL	Tryb pracy procesora Error Level. Ustawiany gdy Reset, Soft Reset, NMI, Cache error. Procesor w trybie jądra, przerwania zablokowane.
EXL	Tryb pracy procesora Exception Level. Ustawiany na wszystkie wyjątki poza powyższymi. Tryb jądra, przerwania zablokowane.
IE	Blokada przerwań.

*Trywialna translacja adresu wirtualnego na adres fizyczny - odrzucenie części bitów.

Cause



BD	Czy winowajca ostatniego wyjątku był w *Branch Delay Slot.*
TI	Wyjątek spowodowany był przez Count & Compare. Wywłaszczanie.
PCI	Przerwanie spowodowane przez przepełnienie licznika wydajnościowego.
IV	Czy punkt wejścia dla przerwań pod offsetem 0x200.
IPx	Oczekujące przerwania. 7-2 sprzętowe, 1-0 programowe.
ExcCode	Kod wyjątku który się wydarzył.

EBase & IntCtl

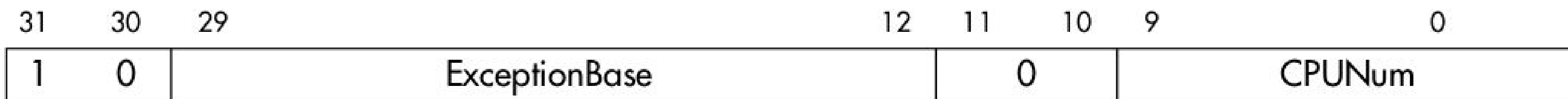


FIGURE 3.7 Layout of the **EBase** register.

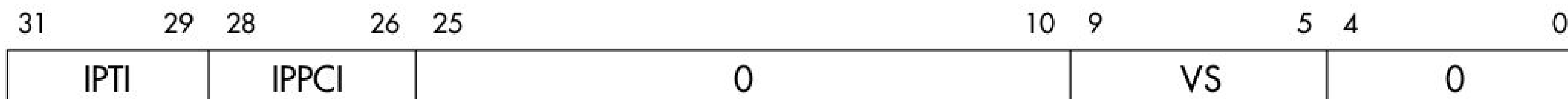


FIGURE 3.8 Layout of the **IntCtl** register.

IPTI	Mapowanie przerwań zegara na bity IPx.
IPPCI	Mapowanie przerwań PCI na bity IPx.
VS	Vector Spacing. Odległość między adresami ISR.
12-29	Adres względem którego zdefiniowane są adresy ISR.

Bity 30,31 wymuszają aby ExceptionBase był w pewnej konkretnej sekcji pamięci.
Brak bitów 0-11 w ExceptionBase definiuje jego ziarnistość.

SRSCtl & SRSMap

Procedura obsługi przerwania powinna zapisać stan rejestrów głównego przeznaczenia, aby po zakończeniu przywrócić ich wartości.

Architektura MIPS oferuje do 8 zestawów rejestrów głównego przeznaczenia zwanych *shadow registers.*

Rejestry SRSCtl i SRSMap służą do zarządzania zestawami rejestrów. Do obsługi wyjątku może zostać przypisany nowy zestaw rejestrów, aby uniknąć zapisywania stanu rejestrów ogólnego przeznaczenia używanych w przerwanym kontekście wykonania.

Dostępne tylko w przypadku *Vectored Interrupts* oraz *EIC.*

Priorytetyzacja

Podstawowa implementacja architektury MIPS **nie oferuje** priorytetyzacji przerwań.

Robione jest to **programowo** na podstawie bitów IPx w rejestrze Cause.

Opcjonalne rozszerzenie mechanizmu obsługi przerwań *External Interrupt Controller* pozwala na sprzętową priorytetyzację przerwań.

Wyjątki mają zdefiniowane priorytety w sprzęcie. Przerwania są w górnej części stawki ale nie najwyżej.

Table 6.6 Priority of Exceptions

Exception	Description
Reset	The Cold Reset signal was asserted to the processor
Soft Reset	The Reset signal was asserted to the processor
Debug Single Step	An EJTAG Single Step occurred. Prioritized above other exceptions, including asynchronous exceptions, so that one can single-step into interrupt (or other asynchronous) handlers.
Debug Interrupt	An EJTAG interrupt (EjtagBrk or DINT) was asserted.
Imprecise Debug Data Break	An imprecise EJTAG data break condition was asserted.
Nonmaskable Interrupt (NMI)	The NMI signal was asserted to the processor.
Machine Check	An internal inconsistency was detected by the processor.
Interrupt	An enabled interrupt occurred.
Deferred Watch	A watch exception, deferred because *EXL* was one when the exception was detected, was asserted after *EXL* went to zero.

Potok

Gdy pojawi się wyjątek, w potoku znajduje się kilka instrukcji w różnym stanie. W przypadku wystąpienia wyjątku chcemy, aby część instrukcji była zakończona, a część nigdy się nie zaczęła.

Architektura implementuje **prezycyjne wyjątki** jeśli potrafi wskazać instrukcję, taką że wszystkie wcześniejsza się wykonały a późniejsze nie zaczęły. Iluzja punktowego zatrzymania potoku.

MIPS posiada precyzyjne wyjątki. Jak to zrobić? Pomyśleć o wyjątkach jak o skokach. Wyjątki to inna forma hazardu przepływu sterowania. Możemy reużyć mechanizmu używanego przy źle przewidzianych skokach.

Wyjątki na różnych etapach wykonania w potoku

- IF - Naruszenie pamięci, dostęp do niewyrównanej pamięci, page fault
- ID - Niezidentyfikowana instrukcja
- EX- przepełnienie
- Mem - Naruszenie pamięci, dostęp do niewyrównanej pamięci, page fault

```
lw     F   D   X   M*  W
add        F*  D   X    M   W
```

Najpierw dowiemy się o wyjątku z późniejszej instrukcji. Co wtedy?

Przy pierwszym wyjątku wyłączamy zapis do pamięci i rejestrów. Dla każdej instrukcji w potoku trzymamy wyjątki które przyszły.

W fazie W przetwarzamy wyjątki w programowej kolejności instrukcji.

Możliwe optymalizacje, gdzie przetwarzamy wyjątki na innych fazach potoku.

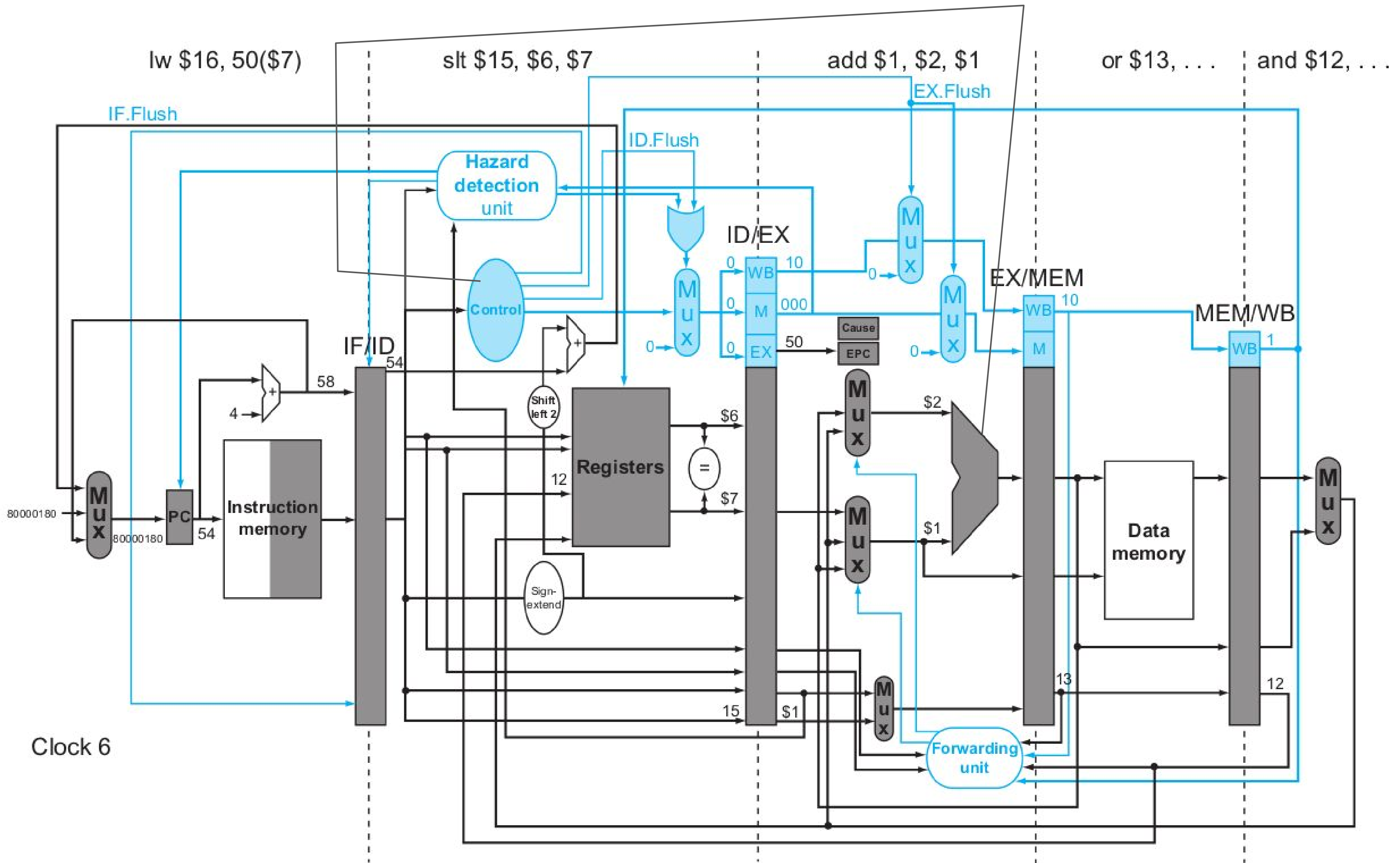
Przykład

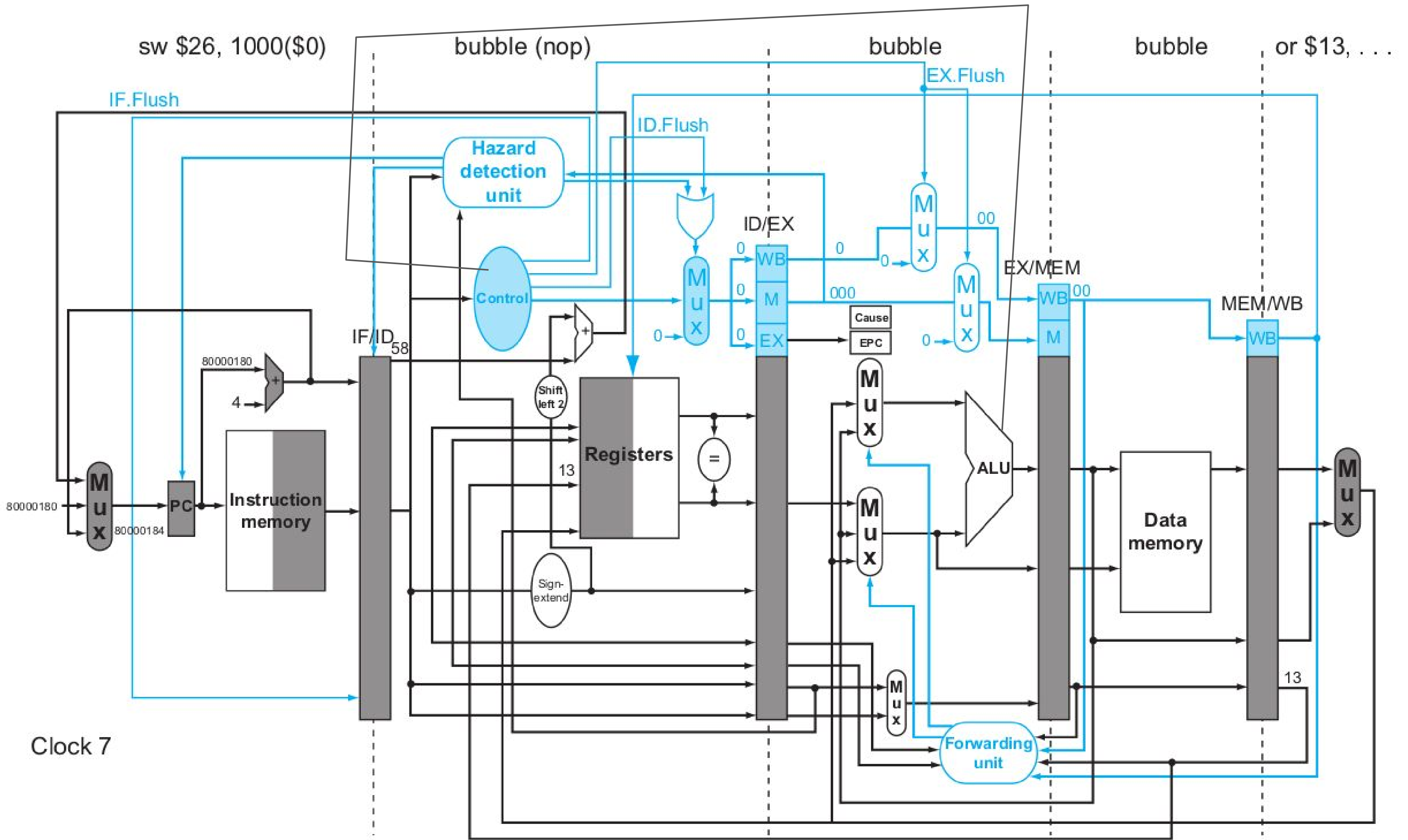
Niech instrukcja add powoduje przepełnienie.

Trzeba pozwolić skończyć się instrukcją sub and i or, oraz wycofać instrukcje slt i lw.

Użyjemy do tego mechanizmu do obsługi skoków.

40_{hex}	sub	\$11, \$2, \$4
44_{hex}	and	\$12, \$2, \$5
48_{hex}	or	\$13, \$2, \$6
$4C_{hex}$	add	\$1, \$2, \$1
50_{hex}	slt	\$15, \$6, \$7
54_{hex}	lw	\$16, 50(\$7)
. . .		


lw $16, 50($7)
slt $15, $6, $7
add $1, $2, $1
or $13, . . .
and $12, . . .
IF.Flush
EX.Flush
ID.Flush
Hazard detection unit
ID/EX
EX/MEM
MEM/WB
IF/ID
Control
Shift left 2
Registers
Sign-extend
Instruction memory
Data memory
PC
Cause
EPC
Forwarding unit
80000180
Clock 6



sw $26, 1000($0)
bubble (nop)
bubble
bubble
or $13, . . .
IF.Flush
EX.Flush
ID.Flush
Hazard detection unit
Control
ID/EX
EX/MEM
MEM/WB
IF/ID
WB
M
EX
Cause
EPC
Registers
Instruction memory
Data memory
ALU
PC
Shift left 2
Sign-extend
Forwarding unit
80000180
80000184
58
13
Clock 7

Wektor wejścia

Procesor musi wiedzieć pod jaki adres skoczyć, żeby ostatecznie zacząć przetwarzać dobrą procedurę wyjątku. Rejestr EBASE zawiera adres względem którego procesor wykonuje skok. Zazwyczaj 0x8000.000. W systemach wieloprocesorowych każdy procesor może mieć ten adres inny.

Dla większości wyjątków offset wynosi 0x180.

Gdy jest ustawiona flaga IV w rejestrze Cause, to przerwania mają osobny offset 0x200.

Wektor wejścia

MIPS w release 2 dodaje:

- Vectored Interrupts - powód przerwania związany z adresem pod który się skacze. Dla każdego przerwania inny adres. Nie jest wymagana procedura która najpierw zidentyfikuje powód przerwania a potem skoczy do docelowej procedury obsługi przerwania.
- EIC (*External Interrupt Controller*) - możliwość zdefiniowania 64 różnych przerwań, oraz przypisanych do nich wektorów wejścia.

W obu przypadkach możemy wykorzystać mechanizm *shadow registers* i przypisać zestaw rejestrów do przerwań. W obu możemy także priorytetyzować przerwania sprzętowo.

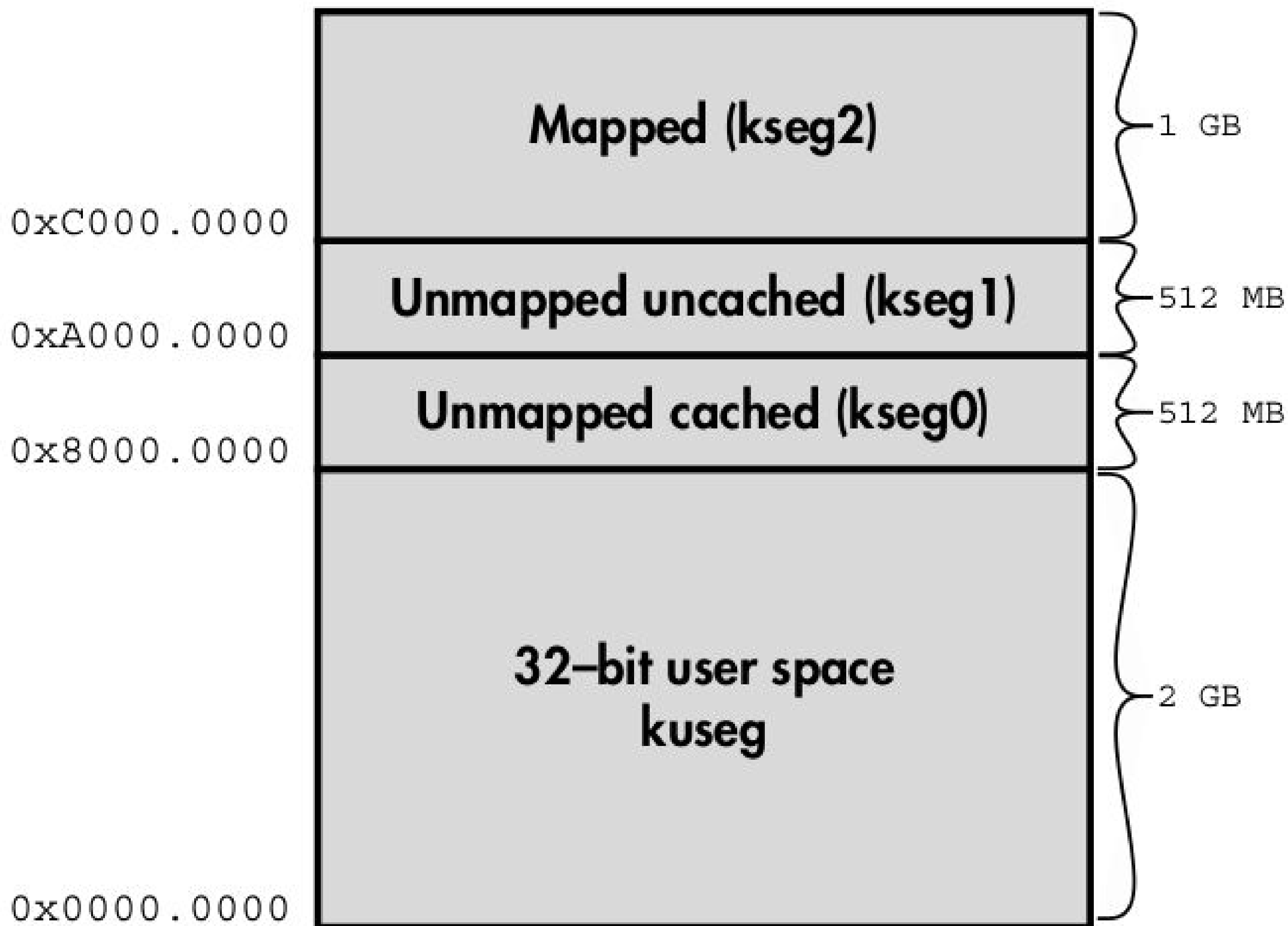
Przydatne w niskopoziomowych systemach wbudowanych, które muszą implementować złożoną logikę obsługi wielu przerwań.

TABLE 5.1 Exception Entry Points

Memory region	*Entry point*	*Exceptions handled here*
On-chip debug	`0xFF20.0200`	EJTAG debug, when mapped to "probe" memory. See section 12.1 for some notes on the EJTAG on-chip debug system.
	`0xBFC0.0480`	EJTAG debug, when using normal ROM memory.
Reset (ROM-only)	`0xBFC0.0000`	Reset and NMI entry point.
ROM entry points (when **SR(BEV)** is one)	`0xBFC0.0400`	Dedicated to interrupts—only used when **Cause(IV)** is set, not available in all CPUs.
	`0xBFC0.0380`	All others.
	`0xBFC0.0300`	Cache Error.
	`0xBFC0.0200`	Simple TLB Refill (**SR(EXL)** is zero).
"RAM" entry points (**SR(BEV)** is zero)	`BASE+0x200+...`	Multiple interrupt entry points—seven more in VI mode, 62 in EIC mode.
	`BASE+0x200`	Interrupt special (**Cause(IV)** is one).
	`BASE+0x180`	All others.
	`BASE+0x100`	Cache error—in RAM but always through uncached kseg1 window.
	`BASE+0x000`	Simple TLB Refill (**SR(EXL)** is zero).

Table 6.9 Exception Vector Offsets

Exception	Vector Offset
TLB Refill, *EXL* = 0	`0x000`
Cache error	`0x100`
General Exception	`0x180`
Interrupt, $Cause_{IV}$ = 1	`0x200` (In Release 2 implementations, this is the base of the vectored interrupt table when $Status_{BEV}$ = 0)
Reset, Soft Reset, NMI	None (Uses Reset Base Address)


Mapped (kseg2)
Unmapped uncached (kseg1)
Unmapped cached (kseg0)
32–bit user space
kuseg
0xC000.0000
0xA000.0000
0x8000.0000
0x0000.0000
1 GB
512 MB
512 MB
2 GB

Obsługa przerwania w podstawowym trybie off 0x200

Procedura obsługujące konkretne przerwanie może wykonać się na poziomie przerwania. Może zapisać GPR, przy kasowaniu przerwania na urządzeniu. Musi zawołać eret.

```
IVexception:
    mfc0   k0, C0_Cause          /* Read Cause register for IP bits */
    mfc0   k1, C0_Status         /* and Status register for IM bits */
    andi   k0, k0, M_CauseIM     /* Keep only IP bits from Cause */
    and    k0, k0, k1            /* and mask with IM bits */
    beq    k0, zero, Dismiss     /* no bits set - spurious interrupt */
    clz    k0, k0                /* Find first bit set, IP7..IP0; k0 = 16..23
    xori   k0, k0, 0x17          /* 16..23 => 7..0 */
    sll    k0, k0, VS            /* Shift to emulate software IntCtlVS */
    la     k1, VectorBase        /* Get base of 8 interrupt vectors */
    addu   k0, k0, k1            /* Compute target from base and offset */
    jr     k0                    /* Jump to specific exception routine */
    nop
```

Procedura obsługujące przerwanie, może również odpowiednio ustawić bity IMx w Status, aby zablokować “mniej ważne” przerwania, ale pozostawić ważniejsze. Zapisać stan i odblokować zagnieżdżone przerwania.

Zagnieżdżone wyjątki

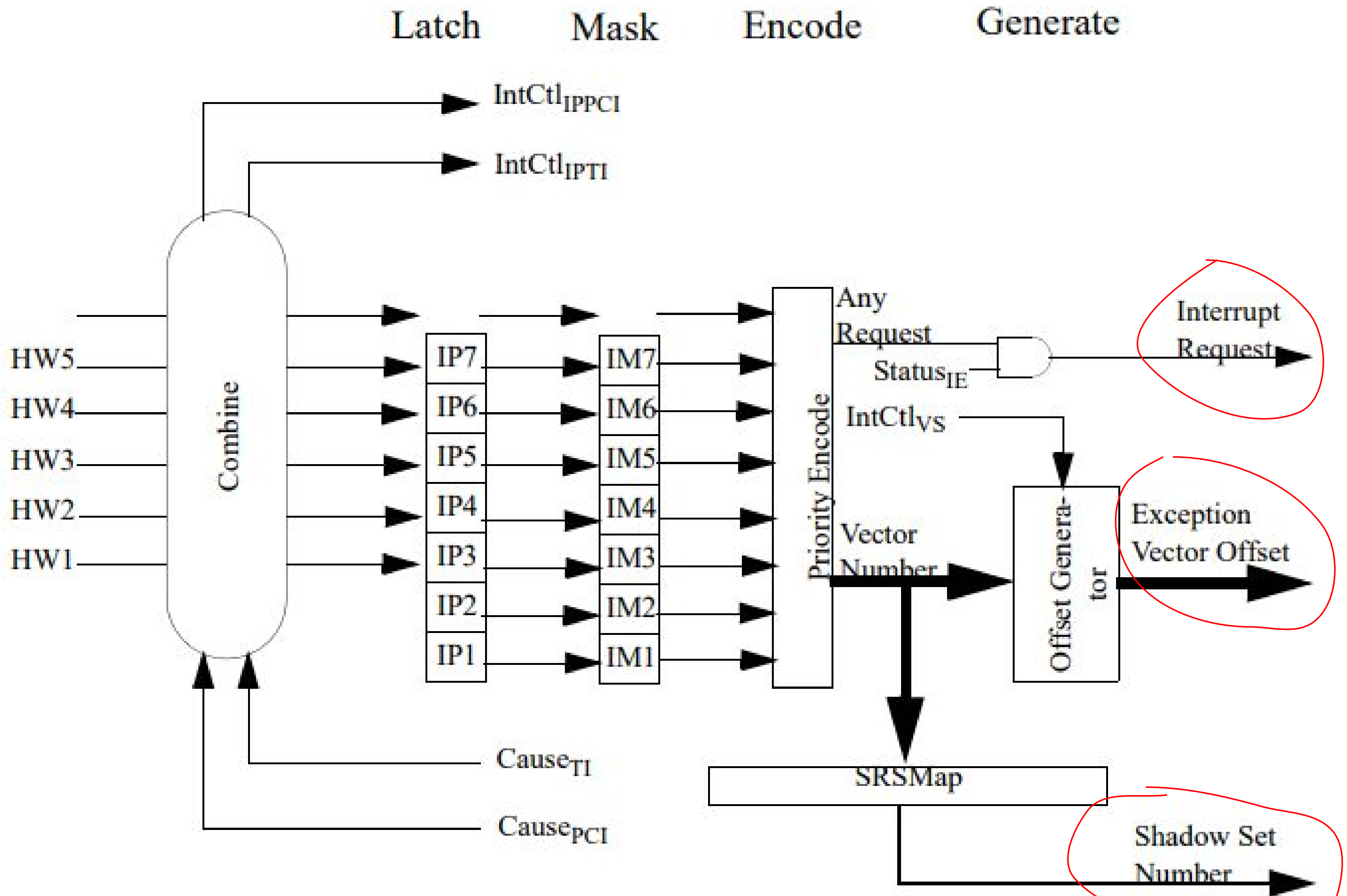
```
/* Save GPRs here, and setup software context */
mfc0   k0, C0_EPC             /* Get restart address */
sw     k0, EPCSave            /* Save in memory */
mfc0   k0, C0_Status          /* Get Status value */
sw     k0, StatusSave         /* Save in memory */
li     k1, ~IMbitsToClear     /* Get Im bits to clear for this interrupt */
                    <-----    /*   this must include at least the IM bit */
                              /*   for the current interrupt, and may include
                              /*   others */
and    k0, k0, k1        <----   /* Clear bits in copy of Status */
ins    k0, zero, S_StatusEXL, (W_StatusKSU+W_StatusERL+W_StatusEXL)
                                 /* Clear KSU, ERL, EXL bits in k0 */
mtc0   k0, C0_Status             /* Modify mask, switch to kernel mode, */
                                 /*   re-enable interrupts */

/*
 * Process interrupt here, including clearing device interrupt.
 * In some environments this may be done with a thread running in
 * kernel or user mode. Such an environment is well beyond the scope of
 * this example.
 */
 /*
  * To complete interrupt processing, the saved values must be restored
  * and the original interrupted code restarted.
  */

    di                          /* Disable interrupts - may not be required
    lw      k0, StatusSave      /* Get saved Status (including EXL set) */
    lw      k1, EPCSave         /*   and EPC */
    mtc0    k0, C0_Status       /* Restore the original value */
    mtc0    k1, C0_EPC          /*  and EPC */
    /* Restore GPRs and software state */
    eret                        /* Dismiss the interrupt */
```

Potem należy zablokować przerwania, przywrócić Status i EPC, oraz wywołać eret.

Table 6.3 Relative Interrupt Priority for Vectored Interrupt Mode

Relative Priority	Interrupt Type	Interrupt Source	Interrupt Request Calculated From	Vector Number Generated by Priority Encoder
Highest Priority	Hardware	HW5	$Cause_{IP7}$ and $Status_{IM7}$	7
		HW4	$Cause_{IP6}$ and $Status_{IM6}$	6
		HW3	$Cause_{IP5}$ and $Status_{IM5}$	5
		HW2	$Cause_{IP4}$ and $Status_{IM4}$	4
		HW1	$Cause_{IP3}$ and $Status_{IM3}$	3
		HW0	$Cause_{IP2}$ and $Status_{IM2}$	2
	Software	SW1	$Cause_{IP1}$ and $Status_{IM1}$	1
Lowest Priority		SW0	$Cause_{IP0}$ and $Status_{IM0}$	0

Figure 6.1 Interrupt Generation for Vectored Interrupt Mode



Ogólny scenariusz obsługi wyjątku

```
/* If Status_EXL is 1, all exceptions go through the general exception vector
/* and neither EPC nor Cause_BD nor SRSCtl are modified */
if Status_EXL = 1 then
    vectorOffset ← 0x180
else
    if InstructionInBranchDelaySlot then
        EPC ← restartPC/* PC of branch/jump */
        Cause_BD ← 1
    else
        EPC ← restartPC                    /* PC of instruction */
        Cause_BD ← 0
    endif

    /* Compute vector offsets as a function of the type of exception */
    NewShadowSet ← SRSCtl_ESS              /* Assume exception, Release 2 only */
    if ExceptionType = TLBRefill then
        vectorOffset ← 0x000
    elseif (ExceptionType = Interrupt) then
```

```
    if (Cause_IV = 0) then
        vectorOffset ← 0x180
    else
        if (Status_BEV = 1) or (IntCtl_VS = 0) then
            vectorOffset ← 0x200
        else
            if Config3_VEIC = 1 then
                if (EIC_option1)
                    VecNum ← Cause_RIPL
                elseif (EIC_option2)
                    VecNum ← EIC_VecNum_Signal
                endif
                NewShadowSet ← SRSCtl_EICSS
            else
                VecNum ← VIntPriorityEncoder()
                NewShadowSet ← SRSMap_{IPL×4+3..IPL×4}
            endif
            if (EIC_option3)
                vectorOffset ← EIC_VectorOffset_Signal
            else
                vectorOffset ← 0x200 + (VecNum × (IntCtl_VS || 0b00000))
            endif
        endif /* if (Status_BEV = 1) or (IntCtl_VS = 0) then */
    endif /* if (Cause_IV = 0) then */
endif /* elseif (ExceptionType = Interrupt) then */
```

```
    /* Update the shadow set information for an implementation of */
    /* Release 2 of the architecture */
    if (ArchitectureRevision ≥ 2) and (SRSCtl_HSS > 0) and (Status_BEV = 0) then
        SRSCtl_PSS ← SRSCtl_CSS
        SRSCtl_CSS ← NewShadowSet
    endif
endif /* if Status_EXL = 1 then */

Cause_CE ← FaultingCoprocessorNumber
Cause_ExcCode ← ExceptionType
Status_EXL ← 1


/* Calculate the vector base address */
if Status_BEV = 1 then
    vectorBase ← 0xBFC0.0200
else
    if ArchitectureRevision ≥ 2 then
        /* The fixed value of EBase_31..30 forces the base to be in kseg0 or kseg1
        vectorBase ← EBase_31..12 || 0x000
    else
        vectorBase ← 0x8000.0000
    endif
endif
```

Pseudokod z MIPS PRA, strona 87.

Powrót z wyjątku

Chcemy wrócić do przerwanego kontekstu wykonania.

Co gdy najpierw wrócimy a potem obniżymy uprawnienia? Powrót z obsługi przerwania powinien być operacją atomową.

Atomowa instrukcja eret m. in. skacze do adresu w EPC i zeruje Status(EXL) zmieniając tryb pracy procesora.

Nested exceptions

Co z EPC i Status? Trzeba je zapisać przed włączeniem zagnieżdżonych wyjątków. Jak już je włączymy to nie możemy polegać na rejestrach k0 i k1, które są zarezerwowane dla obsługi wyjątków. Tworzymy exception frame na stosie do zapisywania rejestrów.

Skoro stos rośnie to nie chcemy dużej ilości zagnieżdżeń. Wprowadzamy priorytety i pozwalamy przerywać obsługę wyjątku tylko wyjątkom o wyższym priorytecie.

Stos jest wtedy tak duży, jak dużo jest różnych priorytetów. Może być statyczny.

Przykład ISR *interrupt service routines*

Nie jest to zbyt mądra ISR bo nie robi nic na rzecz usunięcia problemu który spowodował wyjątek.

```
    .set noreorder
    .set noat
xcptgen:
    la      k0,xcptcount    # get address of counter
    lw      k1,0(k0)        # load counter
    addu    k1,1            # increment counter
    sw      k1,0(k0)        # store counter
    eret                    # return to program
    .set at
    .set reorder
```

Xcptcount powinien być w kseg0 żeby nie dostać wyjątku TLB miss przy czytaniu czy zapisie.

Instrukcje wielocyklowe

Nie powinniśmy oczekiwać że każda instrukcja wykona się w jednym cyklu. Dzielenie i mnożenie zmiennoprzecinkowe są zdecydowanie trudniejszymi operacjami niż np. dodawanie czy zwykły zapis do rejestrów.

Gdybyśmy chcieli każdą instrukcję wykonać w stałej ilości cykli, to zegar procesora musiałby być na tyle wolny aby każda instrukcja zdążyłą się wykonać w jednym / dwóch cyklach.

Lepszym rozwiązaniem jest pozwolić instrukcjom wykonywać się w różnych długościach czasu. Można myśleć o takich instrukcjach jakby miały normalny potok ale faza EX była wykonywana wiele razy.

Dobrym przykładem są instrukcje arytmetyki zmiennoprzecinkowej. Mogą je realizować osobne jednostki funkcyjne.

Niepotokowane jednostki funkcyjne

W danej jednostce funkcyjnej na raz może być tylko i wyłącznie jedna instrukcja.

Inne próbujące użyć danej jednostki funkcyjnej muszą zaczekać.

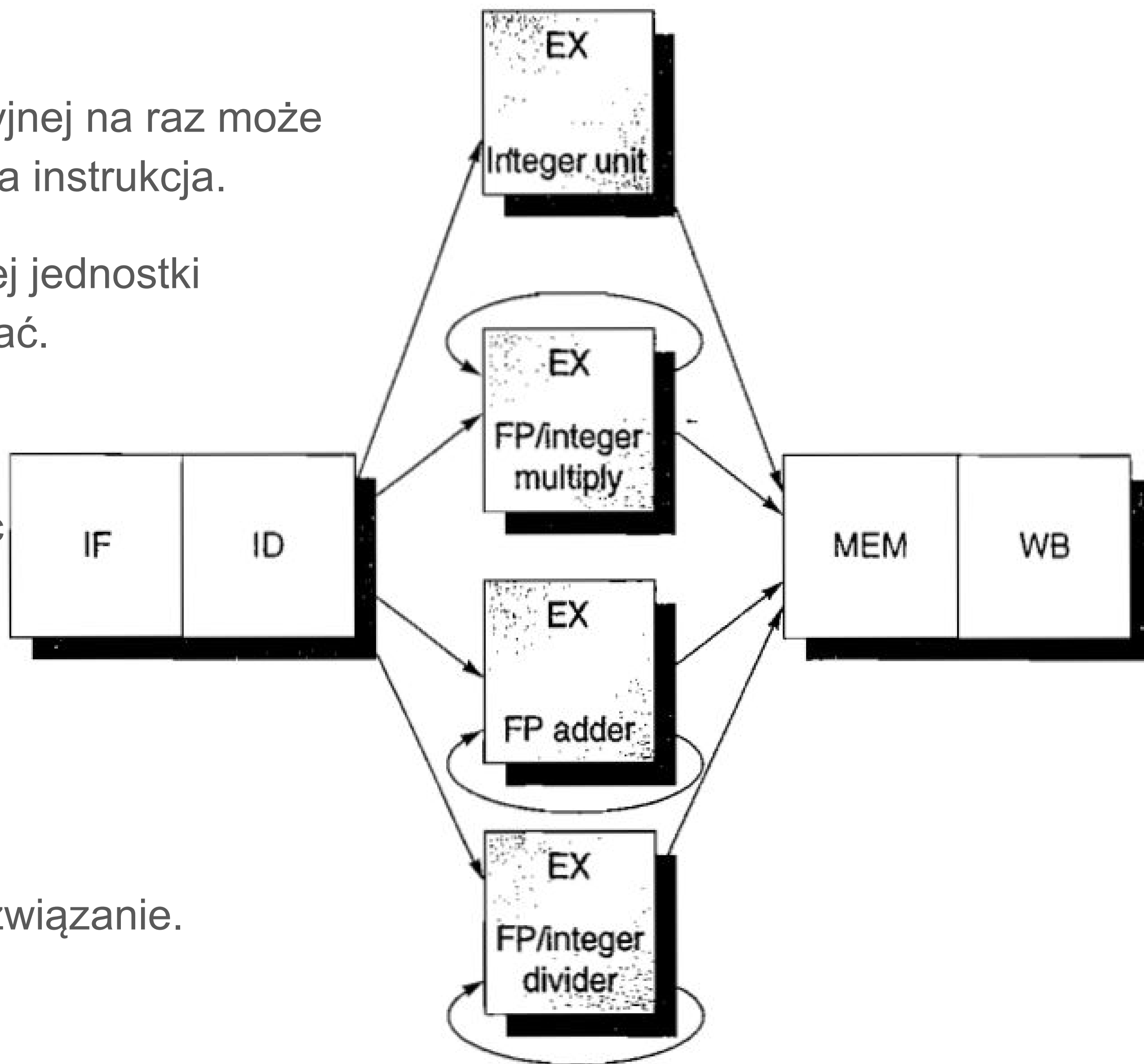
W takim wypadku

instrukcja nie może wejść

do fazy EX więc blokuje

cały potok!

Przydałoby się lepsze rozwiązanie.

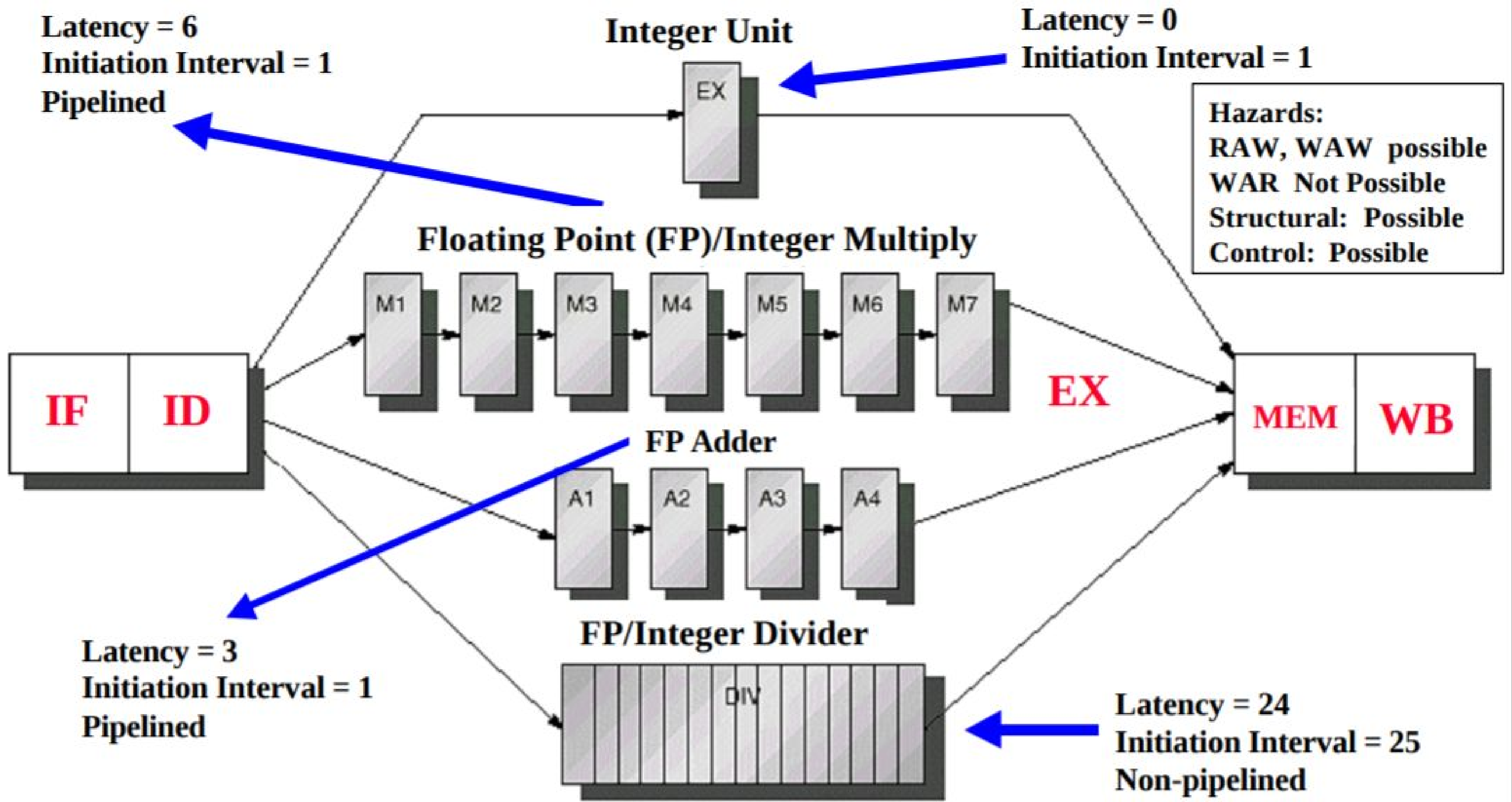


Potokowane instrukcje wielocyklowe

- **opóźnienie** (*latency*)- minimalna ilość cykli między instrukcją produkującą wynik a funkcją używającą jej wyniku
- **interwał inicjacji** (*initiation interval*) - minimalna ilość cykli pomiędzy dwoma takimi instrukcjami

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Extending The MIPS Pipeline: Multiple Outstanding Floating Point Operations



A pipeline that supports multiple outstanding FP operations.

(In Appendix A)

Problemy

- Dzielenie nie jest w pełni potokowane. Mogą pojawić się hazardy strukturalne na jednostce dzielenia.
- Hazardy strukturalne w fazie WB
- Hazardy WAW, ponieważ faza WB wykonuje się out of order.
- Instrukcje będą kończyć się w innej kolejności niż zostały zlecone. Problem z wyjątkami.
- Dłuższe instrukcje, więc zapobieganie hazardom RAW będzie bardziej kosztowne.

Hazardy RAW

L.D	F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D	F0,F4,F6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D	F2,F0,F8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
S.D	F2,0(R2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Zapobieganie hazardom RAW jest dużo bardziej kosztowne ze względu na długość instrukcji.

Hazardy strukturalne

Instruction	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

Można pozwolić na wiele portów zapisu do rejestrów, ale zazwyczaj używany będzie tylko jeden. Drugą opcją jest opóźnianie instrukcji. W jednym z rozwiązań rozpoznajemy instrukcje w fazie ID i jeśli wykryjemy, że faza WB się pokrywa z jakąś inną instrukcją to opóźniamy instrukcję o 1 cykl (bubble).

Hazardy WAW

Gdyby na poprzednim obrazku instrukcja L.D została zlecona jeden cykl szybciej to miałaby fazę Mem wcześniej niż faza Mem wcześniejszej instrukcji jaką jest ADD.D. Powoduje to nadpisanie wyniku instrukcji L.D poprzez instrukcję wcześniejszą!

Rozwiązaniem może być znowu opóźnianie instrukcji, w tym wypadku L.D.

Sytuacja ta może zostać wykryta w fazie ID, a odpowiednia instrukcja opóźniona.

Precyzyjne wyjątki z instrukcjami wielocyklowymi

```
DIV.D    F0,F2,F4
ADD.D    F10,F10,F8
SUB.D    F12,F12,F14
```

Instrukcja SUB.D skończy się jako pierwsza.

Co jeśli jej wykonanie da wyjątek?

Instrukcja ADD.D mogła się już skończyć wykonywać, natomiast DIV.D nie!

Nieprecyzyjny wyjątek!

Albo jeśli DIV.D wygeneruje wyjątek po tym jak instrukcja ADD.D już się skończyła. Nie odtworzymy stanu ADD.D sprzed DIV.D ponieważ ADD.D już zmodyfikowała swoje operandy!

Jednym z rozwiązań jest pamiętanie wyników wszystkich operacji dopóki wcześniej zlecone instrukcje nie wykonają się w pełni.

Źródła

- *Computer Architecture: A Quantitative Approach* - John L. Hennessy
- *See MIPS Run* - Dominic Sweetman
- *Computer Organization and Design: The Hardware/Software Interface* - David A. Patterson, John L. Hennessy
- MIPS® Architecture For Programmers Vol. III: MIPS32® / microMIPS32™ Privileged Resource Architecture