Programmeringsmanual

Jakob Engblom   Marcus Nilsson

17 augusti 1999

Innehåll

Kapitel 1

Processorn R4300

Processorn i våra *Algorithmics-P4032* maskiner är en R4300, tillverkad av NEC.[1] Denna processor är speciellt avsedd för användning i inbyggda system, men detta påverkar bara hårdvarutillverkare. Programmering är samma som för alla andra processorer i R4000-familjen.

Det finns många varianter i MIPS R-serie. De äldsta processorerna är R2000 och R3000; R4000 är dagens vanligaste variant, och R10000 är den mest avancerade. Sedan finns R6000 och R8000 för flyttalsintensiva program.

R4300 är en "64-bitars processor", vilket betyder att register kan vara 64 bitar stora, till skillnad från 32 bitar som är det vanligaste idag.[2] Normalt arbetar man i 32-bitarsläge, och det kommer vi att göra på datorarkitekturkursen.

Skillnader mellan processorna i R-serien

Processorna i R-serien är ganska lika varandra. Vanliga användarprogram kan köras på alla processorer (med ett undantag: det finns vissa tillägg till instruktionsuppsättningen som bara fungerar på nyare varianter som R4000 och R10000). Hårdvarunära program, som operativsystem och inbyggda system, behöver dock anpassas efter varje processor.

Skillnaderna mellan de olika processorerna finner man framförallt i hanteringen av minnet och avbrott/exceptions. Detta skiljer sig dramatiskt mellan olika processorer i R-serien. Detta innebär för vår del att *ingen dokumentation som rör avbrott och exceptions på R2000 och R3000 kan användas för R4000.*

[1] MIPS-designen är "öppen", dvs det finns flera tillverkare av MIPS-kompatibla processorer; även av SUNs SPARC-processorer finns det andra tillverkare.

[2] Alla Intels processorer sedan 80386, Motorolas 68000-serie, IBM/Motorolas PowerPC-serie, SPARC innan Ultrasparc med flera är 32 bitars

1.1 Registeruppsättningen

R4300 har 32 heltalsregister, alla 32 bitar stora. I teorin är alla dessa register likvärdiga, men de har tilldelats speciella roller (så att alla program använder registren på samma sätt). För assemblatorn heter registren egentligen `$1`, `$2`, etc, men de har också logiska namn som anknyter till deras funktion, som `sp` eller `k0`.[3]

I tabell 1.1 nedan visas registren, deras logiska namn och deras roller. Notera att nästan alla roller har att göra med hur registren används i samband med funktionsanrop.

Att ett register *sparas* innebär att funktionen som använder registret måste se till att dess värde då funktionen avslutas är samma som då funktionen anropades. Om registret inte behöver sparas måste den anropande funktionen tänka på att inte hålla några värden som ska användas efter funktionsanropet i ett av dessa register. Alla värden som du vill behålla över ett funktionsanrop måste alltså ligga i register `s0` till `s7`.

Dessa konventioner är egentligen "frivilliga" – ingenting i hårdvaran tvingar dig att använda registren på detta sätt. Det är dock en bra ide att alltid följa konventionen, eftersom det dels gör din kod lättare att förstå för andra, och dels gör att din kod utan problem kan anropa eller anropas av kod skriven i andra språk (t ex C). Notera att biblioteksfunktionerna ni använder är skrivna i C och därmed följer dessa konventioner.

Förutom heltalsregistren finns även tre specialregister:

- `pc` som innehåller adressen till nästa instruktion[4] (programräknare eller instruktionspekare är vanliga svenska namn). Man kan inte skriva eller läsa `pc` direkt från assemblerkoden.
- `hi` och `lo` används för att ta emot resultaten av multiplikationer och divisioner. De kan skrivas och läsas med specialinstruktioner (`mfhi`, `mthi`, `mflo` och `mtlo`).

  Vid en multiplikation hamnar den höga delen av resultatet i `hi` och den låga delen i `lo`.

  Vid en division hamnar kvoten i `lo` och resten i `hi`.

Dessutom innehåller processorn (som alla processorer idag) en flyttalsdel. Denna har 32 egna register, `fp0` till `fp31`. Dessa använder vi dock inte på den här kursen.

[3] För att använda dessa namn i ett assemblerprogram måste ni ha med `#include regdef.h` i er assemblerfil. I Hennessy & Pattersson kallas registren `$sp`, etc – man kan skriva de symboliska namnen både med och utan dollartecken framför.

[4] Vad programräknaren egentligen pekar på i förhållande till de instruktioner som körs vid ett givet ögonblick är ganska oklart på en pipelinad och superskalär processor. Den bästa definitionen är nog att `pc` pekar på nästa instruktion som processorn inte har hunnit *börja* köra än.

1.1.1 Att använda stacken

På en traditionell processor hanteras stacken med speciella instruktioner, oftast `PUSH` och `POP`. `PUSH` lägger ett ord på stacken och ändrar stackpekare, `POP` läser ett ord från stacken och ändrar stackpekaren.

På en RISC-processor (som MIPS) anser man att dessa instruktioner är för komplicerade. Istället ändrar man stackpekaren i en instruktion och skriver värden till stacken med andra. För att till exempel spara undan registren `ra`, `s0` och `s1` på stacken skriver man följande kod. Dessutom måste stacken alltid peka på en jämn 8-bytesadress[5], vilket betyder att vi minskar stackpekaren "onödigt" mycket i exemplet:

```
subu    sp,16    # room for three 32-byte values, plus padding
sw      ra,0(sp) # save ra
sw      s0,4(sp) # save s0
sw      s1,8(sp) # save s1
```

Notera att `sp` pekar på en plats på stacken som *används*. Du måste alltså *först* göra plats på stacken och *sedan* skriva värden.

[5]Detta är nytt på R4000 – på R2000 och R3000 räcker det att stackpekaren pekar på en jämn fyrabytesadress.

Register	Logiskt namn	Användning
$0	zero	Har alltid värdet noll. Skrivning till detta register har ingen effekt.
$1	`$at`	Assembler Temporary. Används av assemblern för att skapa komplexa instruktioner. Bör ej användas i kod man skriver själv.
$2-3	`v0 - v1`	Resultat från funktioner (heltal). Kan användas för temporära variabler inom en funktion utan att sparas.
$4-7	`a0 - a3`	De första fyra heltalsargumenten till en funktion. Kan användas för temporära variabler inom funktionen utan att sparas.
$8-15	`t0 - t7`	Temporära variabler. Dessa register är speciellt avsedda att användas för temporära variabler inom en funktion. Deras värden behöver inte sparas.
$16-23	`s0 - s7`	Sparade variabler. Dessa register måste sparas av anropade funktioner. Den anropande funktionen kan lita på att värden i `s0` till `s7` inte förstörs (skrivs över) av den anropade funktionen.
$24-25	`t8 - t9`	Ytterligare två temporära variabler.
$26-27	`k0 - k1`	Reserverade för användning av operativsystemet. Dessa registers värden förstörs av avbrottshanterare, vilket innebär att de kan ändras precis när som helst. Använd inte i vanlig kod. "k" betyder "kernel".
$28	`gp`	Globalpekare. Detta register används av program skrivna i C för att hålla reda på vissa globala data. Använd inte i assembler!
$29	`sp`	Stackpekare. Pekar på stackens topp. Om ni vill lägga saker på stacken ska ni först minska stackpekaren och sedan lägga in data på konstanta offsets från stackpekaren. Måste peka på en jämn 8-bytesadress!
$30	`fp`	Rampekare. Används vid funktionsanrop för att hålla reda på var på stacken lokala variabler ligger. Måste sparas.
$31	`ra`	Returadress. När en subrutin anropas med instruktionerna `jal`, `bal`, `bgezal` och `bltzal` läggs återhoppsadressen i `ra`. En funktion avslutas med instruktionen `j ra` för att hoppa tillbaka till anropande funktion. Värdet i `ra` ska sparas.

Tabell 1.1: Registerkonventioner för MIPS

Kapitel 2

Att använda assemblatorn

Assemblatorn (programmet som assemblerar er assemblerkod) kräver litet mer än bara assemblerkoden för att kunna assemblera era program (ungefär som ML kräver `fun` framför funktionsdefinitioner och streck mellan olika klausuler).

2.1 Placera kod på rätt ställe

När man skriver assembler måste kod och data placeras på olika platser i minnet kallade *segment* (för vanliga programmeringsspråk sköts detta automatiskt). Detta görs genom att man markerar vad som är att anse som data och vad som är att anse som kod.

Kod och data som bara ska läsas placeras i `text`-segmentet. Under ett operativsystem som UNIX är detta segment oftast skrivskyddat, dvs koden kan inte ändras. Data som kan ändras placeras i `data`-segmentet, som både kan läsas och skrivas.

Ni talar om för assemblatorn att kod följer genom att skriva `.text`. Data inleds med `.data`. Dessutom finns en del andra segment, men dessa behöver ni inte bry er om – de är främst till för kompilatorer för högnivåspråk.

2.2 Att lägga data rätt

Ibland krävs det att data ligger på en jämn adress. T.ex. måste ord som ligger i minnet och hämtas med `lw` och `sw` ligga på en adress som är jämnt delbart med 4.

För att göra detta använder man `.align`. Denna tar ett argument x, som betyder att efterföljande data börjar på en adress som är jämnt delbar med 2^x.

Ex:

```
        .data

message: .asciiz "Hello world\r\n"

        .align 2        # Align to word

ptr:    .space 4

        .text

        ...code...
```

2.3 Anrop utifrån

För att en funktion ska kunna anropas utifrån (dvs från en annan fil) krävs att dess namn deklareras som en “global symbol”. Detta görs med

```
        .globl <funktionsnamn>
```

2.4 Ordningen kan spela roll

Normalt har assemblatorn frihet att ändra ordning på instruktioner för att koden ska bli effektivare, men naturligtvis utan att ändra på programmets mening. Detta kan vara ett problem när man arbetar med exceptions och avbrott, eftersom en del saker då måste ske i en viss ordning. Ett annat tillfälle man kan vilja hindra assemblatorn från att omordna instruktioner är när man vill spåra exekveringen i PMON, för att instruktionerna ska komma i den ordning man förväntar sig.

Om man vill förhindra att assemblatorn ordnar om koden, vilket man alltid ska göra när man arbetar med exceptions, omger man kodavsnittet med `.set nomove` och `.set move`.

Nedan följer exempel på hur man gör detta. Detaljer om instruktioner etc. finns i kapitel 5 om exceptions.

```
        .set nomove

        # Code that may not be reordered
        mtc0 t0,sr
        mtc0 t1,epc
```

```
# end of critical section

.set move
```

Kapitel 3

Bibliotek

När du assemblerar ditt program med makefilen `withclib`[1] kommer ett bibliotek att länkas med som innehåller I/O-rutiner. Dessutom innehåller det kod som startar ditt program, vilket gör att din kod måste uppfylla en del egenskaper.

Nedan följer en beskrivning av hur du skriver program som fungerar ihop med biblioteket. Efterföljande avsnitt tar upp biblioteksrutinerna och hur du använder dem.

3.1 Hur du skriver program

För att länka ihop och använda biblioteket måste följande göras:

- `asm_clib.h` måste inkluderas.[2]
- Startetiketten måste heta `start`. Här börjar exekveringen av din kod.
- Etiketten `start` måste vara synlig utåt. Detta görs genom att skriva `.globl start`. Om du inte gör detta kommer assemblatorn att klaga på att den inte kan hitta symbolen start när den försöker länka ihop hela ditt program.
- Din kod är egentligen en subrutin som körs av biblioteket. Därför måste du göra ett korrekt återhopp; dvs ett hopp tillbaka till `ra`.

Koden för ett typiskt program kommer att se ut så här:

```
#include <mips/regdef.h>  # Register names
```

[1] Detta är standard, om du använder den makefil du fått ut till kursen – se användarhandledningen för MIPSarna.

[2] Detta görs genom att ha en instruktion `#include <asm_clib.h>` i din assemblerfil.

```
#include <asm_clib.h>     # I/O-library definitions

        .text             # This is CODE -- put in text segment
        .globl start      # Make the start label visible

start:

        # Must save RA in order to be able to return correctly

        subu      sp, 8       # Make room on stack
        sw        ra, 0(sp)   # Save RA on the stack

        # YOUR CODE....

        # Restore RA from the stack

        lw        ra, 0(sp)
        addu      sp, 8       # Restore the value of SP


        # At this point, RA and SP have the same value as they
        # had on entry to start. All saved registers must also be
        # restored.

        jr        ra          # Jump back to caller
```

3.2 Hur du använder biblioteksrutinerna

I biblioteket finns rutiner för att göra enkel I/O. Du använder dem genom att ladda argumentregistren med lämpliga argument och anropa rutinen genom att skriva dess namn. En lista på alla rutiner finns i bilaga A.

Exempel:

```
#include <mips/regdef.h>  # Register names
#include <asm_clib.h>     # I/O-library definitions


#
#  DATA SEGMENT
#
        .data              # Mark beginning of DATA segment
message:
```

```
        .asciiz "Hello world\r\n"

#
# CODE/TEXT SEGMENT
#
        .text             # This is CODE -- put in text segment
        .globl start      # Make the start label visible

start:

        # Must save RA in order to be able to return correctly

        subu      sp, 8        # Make room on stack
        sw        ra, 0(sp)    # Save RA on the stack

        # Perform some library calls

        li        a0, 4711

        PRINT_INT           # Print the integer in a0

        la        a0, message

        PRINT_STRING        # Print the string pointed to by a0

        READ_CHAR           # Read a character

        move      a0, v0     # The character is returned in v0

        PRINT_CHAR          # Print the character read


        # Restore RA from the stack

        lw        ra, 0(sp)
        addu      sp, 8        # Restore the value of SP


        # At this point, RA and SP have the same value as they
        # had on entry to start.

# Register a0 does not have to be restored -- it is not a
        # saved register!

        jr        ra           # Jump back to caller
```

Kapitel 4

I/O

In- och utmatning sköts med hjälp av ett speciellt chip, *Winbond W83777 Combi IO*. I detta chip finns stöd för ett stort antal olika enheter, men vi kommer bara att utnyttja den seriella kommunikationen, vilken sköts av de två UART-modulerna (UART = *Universal Asynchronous Receiver Transmitter*).

UART0 och UART1 har samma registeruppsättning men ligger på olika adresser, UART0 på `$bff80fe0` och UART1 på `$bff80be0`. Adresser för register finns i bilaga D. Normalt använder man bara UART0; det är denna som är kopplad till terminalen.

UART kan användas både för pollad och avbrottsstyrd I/O.

Pollad I/O innebär att man själv måste ta reda på när ett tecken har matats in och när UARTen är upptagen med pågående utmatning. I UARTen finns register som man kan läsa för att ta reda på detta. Innan man matar ut måste man vänta på att UARTen är klar med pågående utmatning. Detta gör man genom att läsa statusregistret gång på gång tills det visar att UARTen är ledig. Därefter skriver man tecknet till utmatningsregistret. För att läsa ett tecken väntar man på motsvarande sätt tills inmatning av tecken har skett, varefter man läser tecknet från inmatningsregistret.

Avbrottsstyrd I/O innebär att CPUn blir avbruten när ett tecken har matats in eller utmatningen är klar. Man får sedan kolla i UART vad som exakt har hänt. Ett hjälpchip som kallas ICU, *Interrupt Control Unit*, tar emot avbrottssignalen från UART och skickar den vidare till CPUn. Mer detaljer om hur UART, CPU och ICU fungerar tillsammans finns i kapitel 5.

4.1 Pollad I/O

Vid pollad I/O används tre register:

UART_RBR, Receive buffer register Innehåller inkommande tecken.

UART_TBR, Transmit buffer register Här skriver man det tecken som ska matas ut.

UART_USR, Uart status register Innehåller status för UART, med bl.a. följande bitar:

- **UART_USR_TBRE, TBR empty** Är satt om `UART_TBR` är tomt, dvs UARTen är redo för utmatning.
- **UART_USR_RDR, RBR data ready** Är satt om det finns ett tecken att läsa i `UART_RBR`.

4.2 Avbrottstyrd I/O

Avbrott i UART0 och UART1 kontrolleras av `UART_ICR` (Interrupt Control Register). I denna ligger bl.a. följande bitar:

UART_ICR_ETBREI När denna är satt genereras avbrott när `UART_TBR` är tomt.

UART_ICR_ERDRI När denna är satt genereras avbrott när det finns tecken i `UART_RBR`.

Mer om precis vad som händer vid ett avbrott finns i kapitlet 5.

Kapitel 5

Exceptionhantering

Exceptionhanteringen på MIPS är relativt primitiv. Då någon form av ovanlig situation inträffar (ett programfel, adresseringsfel, inmatningsavbrott och liknande) avbryter processorn programmet som exekveras och hoppar till en exceptionhanterare. Denna hanterare ska sedan ta reda på vad det var som hände och handla i enlighet med detta.

Exceptions och avbrott

Exceptions är allting som gör att processorn avbryter det den håller på med och hoppar till en speciell exceptionhanterare. Det kan vara adresseringsfel, operativsystemsanrop, inmatning från tangentbordet, med mera. Se tabell C.1 för en lista över alla exceptions.

Avbrott är ett specialfall av exceptions (exception typ 0, enligt tabell C.1). Avbrott skapas då externa enheter vill skicka data till processorn eller behöver information från den, till exempel för I/O till skärm eller disk.

5.1 Hjälpprocessor noll

Avbrott sköts inte av den egentliga MIPS-processorn. Istället har man valt att lägga ut avbrottshanteringen på en specialmodul i processorn, kallad *Coprocessor 0*, CP0.

CP0 styrs genom att skriva och läsa värden i dess register med hjälp av de två instruktionerna:

- `mfc0 reg,cp0reg` hämtar ett värde från register *cp0reg* i CP0 och lägger i register *reg*.
- `mtc0 reg,cp0reg` lägger värdet i register *reg* i CP0-register *cp0reg*.

CP0 register finns definierade i `cp0_regdef.h`.

Exempel:

```
.set    nomove
mtc0    t3,sr
.set    move
```

eller:

```
.set    nomove
mfc0    t4,cause
.set    move
```

5.1.1 Register i CP0

Vi nöjer oss med att presentera de relevanta registren i CP0. De flesta används för att hantera det virtuella minnet, och detta berör vi inte här.

Register	Logiskt namn	Användning
\$9	`Count`	Räknare. Detta register räknas ständigt upp i en konstant takt, närmare bestämt halva den takten som processorn maximalt kan exekvera intruktioner. Se även avsnittet om timeravbrott.
\$11	`Compare`	Jämförelse. När `Count` blir lika med `Compare` genereras ett timeravbrott. Att skriva till detta register ger som sidoeffekt att återställa timeravbrottet. Se även avsnittet om timeravbrott.
\$12	`SR`	Statusregister. Kontrollerar vad processorn kan göra, vilka interrupt som tas omhand, med mera. En översikt över de enskilda fältens storlek och placering finns i avsnitt C.1.
\$13	`Cause`	Causeregistret innehåller information om vad som hänt. Här kan man se vilken typ av exception som inträffade, och, om det var ett avbrott, vilket avbrott som inträffade. Layouten presenteras i avsnitt C.2.
\$14	`EPC`	Exception Program Counter. Innehåller adressen till den instruktion som har orsakat ett exception. Används för att återstarta programmet när ett exception har hanterats.

5.2 När är exceptions tillåtna?

När ett exception inträffat vill man helst ta hand om det innan man tar hand om eventuella senare exceptions. För att hindra exceptions från att avbryta varandra sätts EXL-biten i `sr` (se bilaga C.1) för att markera att ett exception hanteras. Så fort EXL sätts till 0 igen kommer eventuella väntande exceptions att inträffa.

Avbrott har man bättre kontroll över. Även om exceptions är tillåtna kan man stänga av avbrott helt genom att sätta IE-biten i `sr` till 0.

Om man vill stänga av avbrott selektivt kan man behålla IE som 1, men sätta bitar i IM-fältet i `sr` till 0 eller 1. 1 innebär att avbrottet får förekomma, 0 att det är avstängt.

För att tillåta ett visst avbrott ska alltså

- EXL = 0,
- IE = 1 och
- IMx = 1, där x är numret på avbrottet man vill tillåta.

5.3 Att hantera ett exception

När ett exception inträffar hoppar processorn till en viss plats i minnet; förhoppningsvis finns det en exceptionhanterare på denna adress som kan ta hand om detta (se avsnitt 5.4 för mer om var hanteraren ska ligga och hur man placerar den där).

Denna hanterare måste genomföra följande steg:

1. Spara undan alla register som exceptionhanteraren kommer att använda så att de kan återställas senare.
2. Identifiera vilken typ av exception som inträffade.
3. Anropa en funktion för att hantera just denna typ av exception.
4. Återställa exceptionkällan.
5. Återställa register.
6. Återstarta programmet som avbröts.

Alla dessa steg har sina subtila detaljer.

Innan exceptionhanteraren anropas har processorn automatiskt gjort följande:

- Lagrat programräknaren i `epc`.
- Satt EXL-biten i `sr` till 1 så att andra exception inte kan inträffa.

5.3.1 Att spara register

Det enklaste är att spara registren på stacken. Detta fungerar så länge man bara hanterar avbrott – om man ska hantera generella exceptions kan man inte lita på att stackpekaren har ett riktigt värde, och då måste man spara undan dess gamla värde och sedan gå över till att använda en speciell stack för exceptionhanteringen.

Register `k0` och `k1` är reserverade för exceptionhanterare. Dessa kan man alltså använda för att spara undan andra register till minne.[1]

Glöm ej att spara `at`. Man använder oftast detta register utan att tänka på det, eftersom assemblatorn översätter en del instruktioner till kod som använder `at`.

5.3.2 Identifiera ett exception

Eftersom samma exceptionhanterare tar hand olika typer av exceptions (se avsnitt 5.4 nedan) måste man lista ut vad som inträffat genom att:

1. ExcCode-fältet i `cause`-registret (se bilaga C.2) talar om vilken typ av exception som inträffat.
2. Om det var ett avbrott tittar man på IP-bitarna i `cause` för att se vilket avbrott det var.
3. För externa enheter måste man oftast läsa av statusregister i själva enheten för att veta precis vad som hänt.

5.3.3 Återställa register

De register som sparades då avbrottshanteraren anropades måste återställas innan den avslutas. Eftersom ett avbrott kan inträffa precis när som helst måste *alla* register som används av avbrottshanteraren sparas och återställas – annars kan vad som helst hända.

5.3.4 Återställ exceptionkällan

För att inte samma exception ska inträffa igen måste källan (dvs det som orsakade att detta exception inträffade) återställas. Ibland går detta automatiskt i hårdvaran, och ibland måste exceptionhanteraren göra detta själv.

Exempelvis måste IP-bitarna i `cause` sättas till 0 för att inte samma avbrott ska återkomma flera gånger.

[1]Eftersom den enda adresseringsmoden hos R4000 är register + offset måste man använda ett register för att kunna spara undan saker till minne. Utan reserverade register vore detta inte möjligt.

5.3.5 Återstarta programmet

För att återstarta programmet som avbröts använder man instruktionen `eret`. Tidigare (ex på R2000/3000) använde man `rfe` men på nyare modeller finns `eret` som sköter återhoppet på ett snyggare och bättre sätt.

`eret` gör följande:

- EXL-biten i `sr` sätts till 0.
- PC sätts till värdet som lagrats i `epc`.

5.4 Var ligger en exceptionhanterare?

En exceptionhanterare ska ligga på en viss adress i minnet. Det finns fyra olika "ingångar", på följande adresser:

- `$a0000000` – Virtuellt minneshantering.
- `$a0000080` – Virtuellt minneshantering, 64-bitarsläge.
- `$a0000100` – Cachefel.
- `$a0000180` – *General exception vector*; allt annat, inklusive avbrott.

Varje exceptionhanterare är alltså i praktiken maximalt 128 bytes lång, vilket betyder att man inte får plats med några större mängder kod. Detta kommer man runt genom att på adresserna ovan bara lägga en liten kodsnutt som anropar den "riktiga" exceptionhanteraren.

En sådan kodsnutt (*stub*) kan se ut så här:

```
        .text
/* a short stub to transfer execution to the real handler */
        .set    nomove
excptstub:
        la      k0,excpthandler         # jump to correct segment
        jr      k0
excptstubend:
        .set    move
```

Vi antar att den riktiga exceptionhanteraren heter `excpthandler`. Denna ligger tillsammans med övrig kod.

Det går inte att ladda in kod till MIPS som ligger på dessa adresser [2]. Därför måste man själv kopiera exceptionhanteraren till rätt adress. Detta görs av en snutt kod enligt följande:

[2] Detta beror på att PMON vägrar ladda till dessa adresser

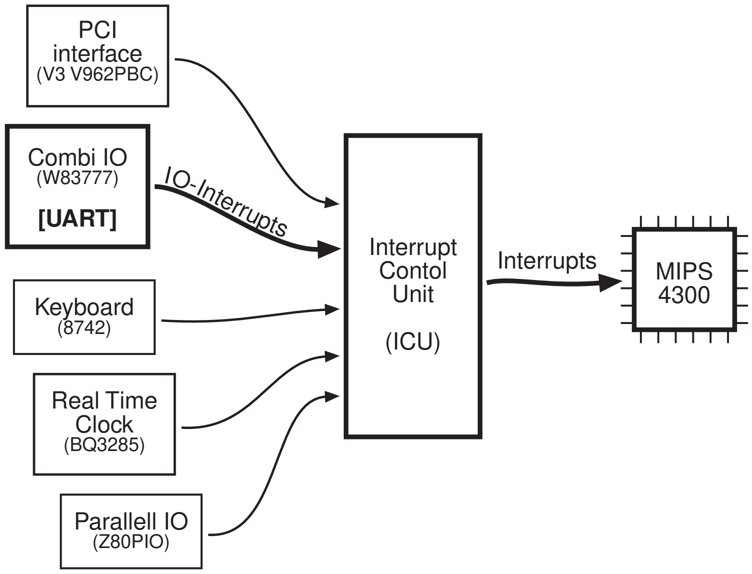
```
#include <mips/regdef.h>
#include <cp0_regdef.h>

#define general_exception_vector 0xa0000180

/* init code -- copy the stub to the correct place in memory */
        .set    nomove
        .globl  excpthandler_init
excpthandler_init:
        la      t0, excptstub
        la      t2, excptstubend
        la      t1, general_exception_vector

        subu    t2,t2,t0        # calculate length of handler

copyloop:
        lw      v0,(t0)         # Copy one word at a time
        sw      v0,(t1)
        addu    t0,4
        addu    t1,4
        subu    t2,4
        bgez    t2,copyloop

        j       ra              # This is a leaf routine, so ra is
                                # unchanged

        .set    move
```

5.5 Avbrottshantering på P4032

Processorns inbyggda avbrottshantering är ganska primitiv. För att ge större flexibilitet finns ett hjälpchip kallat ICU (*Interrupt Control Unit*). Bild 5.1 visar hur interrupt från olika externa enheter samlas ihop i ICU innan de skickas vidare till processorn.

I/O-chippet på våra maskiner innehåller två separata in/utmatningsmoduler, UART0 och UART1. Dessa genererar alltid IRQ3 resp. IRQ4 (IRQ = Interrupt ReQuest) till ICUn. ICUn omvandlar dessa till `INTR 0` till `INTR 3` (INTR = INTeRrupt). INTR går *ut* från ICU; dessa signaler är kopplade till avbrott 2 till 5 *in* i processorn. Bild 5.2 ger en översikt över hur avbrotten skickas från UART till processorn via ICU.

ICUn kan dessutom generera `INTR 4` vilket blir avbrott 6 i processorn (detta används för andra enheter än UARTarna). I processorn finns också avbrott 0 och 1; dessa är "mjukvaruavbrott", samt avbrott 7; detta används för processorns interna timer-funktion.



Figur 5.1: Avbrottshanteringsstrukturen hos P4032. De delar som vi använder till in- och utmatning är markerade med tjockare linjer.

I ICUn måste man alltså ställa in vad avbrotten från UART ska generera för CPU-avbrott. Detta görs med hjälp av `ICU_IXR0/1`. Nedan följer en förteckning över registren i ICUn. Observera att `ICU_IRR` och `ICU_IMR` ligger på samma adress. Vilket register man använder beror på om man läser (`ICU_IRR`) eller skriver (`ICU_IMR`).

ICU_IRR, Interrupt request register Innehåller information om vilka avbrott som är aktiva.

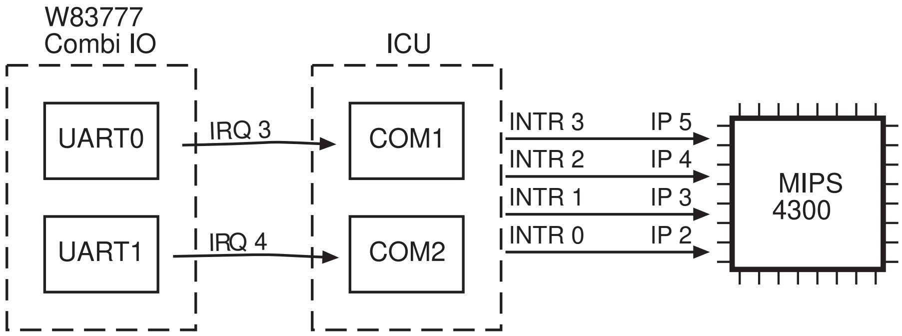
- **ICU_IRR_COM1** Är satt när IRQ 3 är aktiv.
- **ICU_IRR_COM2** Är satt när IRQ 4 är aktiv.

ICU_IMR, Interrupt mask register Anger vilka avbrott som ska skickas vidare till CPUn.

- **ICU_IMR_COM1** Mask för IRQ 3.
- **ICU_IMR_COM2** Mask för IRQ 4.

ICU_IXR0/1, Interrupt crossbar register 0 och 1 Register som anger vilket CPU avbrott som ska genereras för ett visst avbrott. För de flesta avbrott finns det två bitar som anger vilket av fyra CPU avbrott som ska användas (INTR 0-3).

För IRQ 3 används `ICU_IXR0_COM1` och för IRQ 4 används `ICU_IXR1_COM2`.



Figur 5.2: Avbrottshantering i I/O-systemet

Kapitel 6

Timeravbrott

I MIPS finns det en inbyggt stöd för timeravbrott. Detta innebär att det är möjligt att få ett avbrott med jämna mellanrum. Detta mellanrum kan man ställa in själv.

Två register i CP0 sköter om detta, `Count` och `Compare`. `Count` räknas upp med samma hastighet som huvudklockan, vilket i R4300 är 67Mhz, dvs 67 miljoner gånger per sekund. När `Count` är lika med `Compare` genereras ett timeravbrott som är IP 7 i `Cause` (se Exceptionhantering för mer information).

För att återställa timeravbrottet skriver man till `Compare`.

Kapitel 7

Realtidsklockan

MIPS har en inbyggd klocka som understöds av ett batteri så den inte förlorar tiden när inte strömmen är på. Den har en upplösning på 1 sekund och håller reda på veckodag, datum, m.m.

Klockan har ett antal register där sekunder, minuter etc. kan läsas och skrivas. De kan presenteras i dels binär form och i BCD (Binary Coded Decimal). Dessutom finns möjlighet att sätta 12-timmars eller 24-timmar läge.

Man läser och skriver till registren genom att sätta offset i `RTC_ADDR` och läsa/skriva `RTC_DATA`. Registren med offsets finns i bilaga D.

7.1 Att läsa av klockan

Innan man läser av registren sätter man det dataformat och läge man vill ha. Register uppdateras en gång per sekund och när det sker är `RTC_A_UIP` satt. För att läsa av klockan måste man först se till att denna bit ej är satt, vilket man gör med pollning. Biten sätts en liten tid före själva uppdateringen sker, så om man upptäcker att biten ej är satt är man garanterad en viss tid innan uppdateringen kommer igång.

7.2 Att sätta klockan

För att skriva till registren skriver sätter man först `RTC_B_UTI`, som förhindrar att registren uppdateras. Sedan sätter man dataformat och läge, varefter man skriver till registren. Slutligen rensar man `RTC_B_UTI`.

Bilaga A

Biblioteksrutiner

Inkludera `asm_clib.h` för att använda biblioteket.

PRINT_INT

Skriver ut ett heltal på terminalen

Argument: a0 - Heltal att skriva ut

Resultat: -

PRINT_STRING

Skriver ut en sträng på terminalen

Argument: a0 - Nollterminerad sträng att skriva ut

Resultat: -

PRINT_CHAR

Skriver ut ett tecken på skärmen

Argument: a0 - Tecken att skriva ut

Resultat: -

READ_CHAR

Läser in ett tecken från terminalen

Argument: -

Resultat: v0 - Det inlästa tecknet

Bilaga B

Registernamn och registerkonventioner

De logiska registernamnen nedan kan man använda om man inkluderar filen `<mips/regdef.h>`.

Se avsnitt 1.1 för mer information om registren.

Register	**Logiskt namn**	**Användning**
\$0	zero	Har alltid värdet noll.
\$1	`$at`	Assembler Temporary. Använd ej!
\$2-3	`v0 - v1`	Resultat från funktioner.
\$4-7	`a0 - a3`	Funktionsargument.
\$8-15	`t0 - t7`	Temporära variabler.
\$16-23	`s0 - s7`	Sparade variabler. Måste sparas.
\$24-25	`t8 - t9`	Temporära variabler.
\$26-27	`k0 - k1`	Reserverade för avbrottshantering.
\$28	`gp`	Globalpekare. Reserverad för C. Använd ej!
\$29	`sp`	Stackpekare.
\$30	`fp`	Framepointer. Måste sparas.
\$31	`ra`	Returadress. Måste sparas.

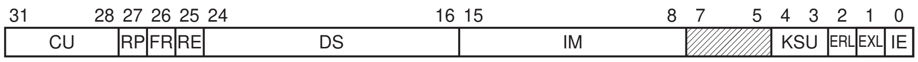
Tabell B.1: Registerkonventioner för MIPS

Bilaga C

CP0 register

C.1 Statusregistret

Statusregistret kontrollerar ett antal funktioner hos processorn. Det ser ut enligt följande:



Figur C.1: Statusregistrets utseende

CU *Coprocessor Usable.* Kontrollerar vilka coprocessorer som kan användas. 1 $\rightarrow$ användbar, 0 $\rightarrow$ får ej användas. En bit per coprocessor.

RP *Reduced Power.* Sätts till 1 för att köra processorn långsammare (drar mindre ström).

FR *extra Floating point Registers.* 1 $\rightarrow$ 32 flyttalsregister, 0 $\rightarrow$ 16 flyttalsregister.

RE *Reverse Endian.* Byter "endianness" hos processorn. Denna bit ska man *inte* röra, om man inte är väldigt säker på vad man håller på med.

DS *Diagnostic Status.* Används för att testa processorn. Beror på den specifika processortypen.

IM *Interrupt Mask.* En bit för varje interrupt 0-7, samma ordning som i `Cause`-registret (se nedan). 1 $\rightarrow$ interrupt på, 0 $\rightarrow$ interrupt avstängt.

KSU *Kernel Supervisor User.* Avgör i vilket läge processorn kör. 10 $\rightarrow$ Usermode, 01 $\rightarrow$ Supervisormode, 00 $\rightarrow$ Kernelmode.

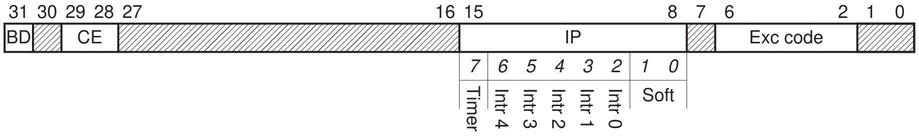
ERL *ERror Level.* 1 $\rightarrow$ felhanteringsläge, 0 $\rightarrow$ normalt.

EXL *EXception Level.* 1 → avbrottshanteringsläge, 0 → normalt.

IE *Interrupt Enable.* 1 → interrupt på, 0 → (alla) interrupt avstängda.

C.2 Causeregistret

Causeregistret talar om för en avbrottshanterare vad som orsakade avbrottet.



Figur C.2: Causeregistrets utseende

BD *Branch Delay*. 1 → senaste avbrottet ägde rum i en *branch delay slot*, 0 → normalt.

CE *Coprocessor Error*. Talar om vilken coprocessor som inte gick att använda (vid ett *coprocessor unusable*-avbrott).

IP *Interrupt Pending*. Talar om vilka avbrott som har inträffat och som väntar på att hanteras. En bit per avbrott: 1 → avbrott har inträffat, 0 → avbrott har inte inträffat.[1]

ExcCode *EXCeption CODE*. Talar om vilken typ av exception som inträffade. Det finns 32 olika avbrott, se tabell C.1.

[1] Notera att flera av dessa bitar kan vara satta samtidigt – inget hindrar processorn från att ta emot flera avbrott samtidigt.

Nummer	**Namn**	**Beskrivning**
0	`Int`	Interrupt
1	`Mod`	TLB modification exception
2	`TLBL`	TLB exception (load or instruction fetch)
3	`TLBS`	TLB exception (store)
4	`AdEL`	Address error exception (load or instruction fetch)
5	`AdES`	Address error exception (store)
6	`IBE`	Bus error exception (instruction fetch)
7	`DBE`	Bus error exception (data reference)
8	`Sys`	Syscall exception
9	`Bp`	Breakpoint
10	`RI`	Reserved instruction
11	`CpU`	Coprocessor unusable exception
12	`Ov`	Arithmetic overflow exception
13	`Tr`	Trap exception
14	`VCEI`	Virtual coherency exception instruction
15	`FPE`	Floating point exception
16-22	-	*Reserved for future use*
23	`WATCH`	Reference to `WatchHi` or `WatchLo` address.
24-30	-	*Reserved for future use*
31	`VCED`	Virtual coherency exception data

Tabell C.1: Exceptionkoder i `ExcCode`-fältet i `Cause`-registret

Bilaga D

Hårdvaruadresser

Adresserna för UART är för UART0.

Relevanta bitar i adressen ges nedanför respektive adress. T ex är UART_USR_TBRE och UART_USR_RDR bitar i UART_USR.

Adresser av typen RTCx betyder att x är en offset som talar om vilket register i RTC som avses (läggs i RTC_ADDR).

Namn	**Adress**	**Bit**	**Beskrivning**
UART_RBR	bff80fe0		Receive buffer register
UART_TBR	bff80fe0		Transmit buffer register
UART_USR	bff80ff4		Uart status register
UART_USR_RDR		0	RBR data ready
UART_USR_TBRE		5	TBR empty
UART_ICR	bff80fe4		Interrupt control register
UART_ICR_ERDRI		0	RBR data ready interrupt enable
UART_ICR_ETBREI		1	TBR empty interrupt enable
ICU_IRR	bff90000		Interrupt request register
ICU_IRR_COM1		3	COM1 interrupt pendning
ICU_IRR_COM2		4	COM2 interrupt pendning
ICU_IMR	bff90000		Interrupt mask register
ICU_IMR_COM1		3	COM1 interrupt mask
ICU_IMR_COM2		4	COM2 interrupt mask
ICU_IXR0	bff9000c		Interrupt crossbar register 0
ICU_IXR0_COM1		6-7	COM1 intr
ICU_IXR1	bff90010		Interrupt crossbar register 1
ICU_IXR1_COM2		0-1	COM2 intr
RTC_ADDR	bff00000		RTC pointer register (offset, 1 byte)
RTC_DATA	bff00004		RTC data register (1 byte)
RTC_SECONDS	RTC0		Seconds (0-59)
RTC_MINUTES	RTC2		Minutes (0-59)
RTC_HOURS	RTC4		Hours (1-12 AM, 129-140 PM eller 0-23
RTC_DAYWEEN	RTC6		Day of week (1-7, 1 = Sunday)
RTC_DAYMONTH	RTC7		Day of month (1-31)
RTC_MONTH	RTC8		Month (1-12)
RTC_YEAR	RTC9		Year (0-99)
RTC_A	RTC10		RTC register A
RTC_A_UIP		7	Update cycle status
RTC_B	RTC11		RTC register B
RTC_B_HF		1	Hour Format. 1 = 24-hour, 0 = 12-hour
RTC_B_DF		2	Data Format. 1 = Binary, 0 = BCD
RTC_B_UTI		7	Update Transfer Inhibit

Tabell D.1: Adresser

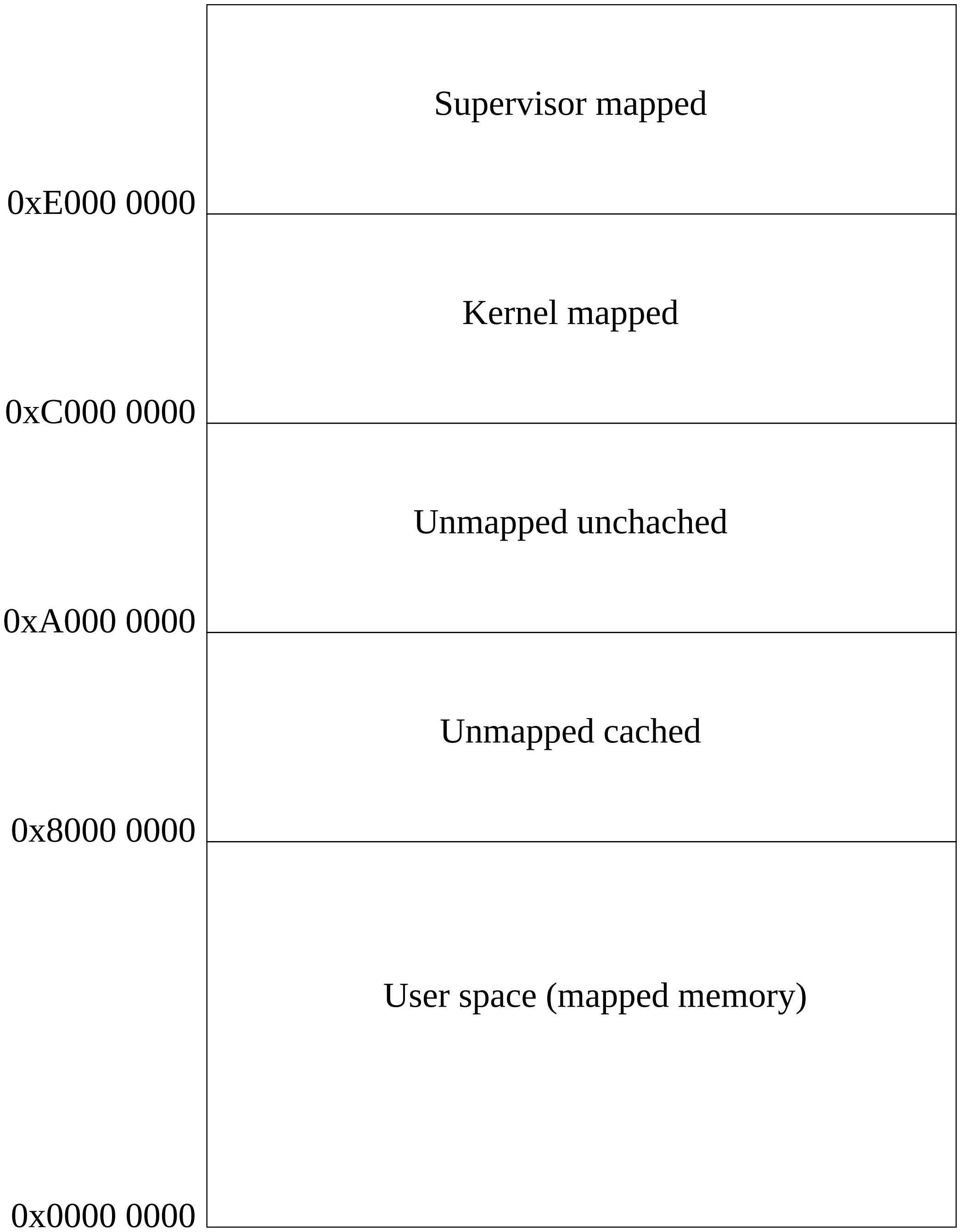
Bilaga E

Minneskarta

I MIPS är alla adresser virtuella, dvs de översätts till fysiska adresser när man läser/skriver till dem. En viss del av adressrymden är dock avsatt för att man ska kunna komma åt det fysiska minnet på ett enkelt sätt. Dessa adresser översätts alltid till samma adresser så att man enkelt kan komma åt en given fysisk adress.

Adresserna 0x80000000-0x9fffffff översätts till de fysiska adresserna 0x00000000-0x1fffffff. Likaså 0xa0000000-0xbfffffff med skillnaden att läsningar/skrivningar inte går genom cachen. Alla hårdvaruadresser läser man via de senare eftersom man inte vill att de cachas. Om man t.ex. skriver till adress 0xb0000004 så skrivs det till den fysiska adressen 0x10000004, utan cachning.

Om man inte använder virtuellt minne är man tvungen att köra i kernel mode, eftersom man från user mode inte kommer åt denna adressrymd som mappas på detta sätt till den fysiska adressrymden.



Figur E.1: Minneskarta hos P4032