Sammanfattning av Datorteknik

Felix Mulder

9 mars 2013

Assembly

```
#include <iregdef.h>        # För att definiera zero, s0-s7, ra etc.

add s0, s0, s1              # Adderar innehållet i s0 med s0 och lägger det i det mest vän
add s0, s1                  # Gör samma sak som föregående

neg t0, s4                  # Gör innehållet i s4 negativt och lägger det i t0
beq zero, zero, L1          # Hoppar till label "L1"
b L1                            # Samma som föregående

addiu t0, zero, s0          # Lägger innehållet i s0 i t0.
move t0, s0                 # Samma som föregående

# Adresseringsmetoder
lw t0, 4(t3)                # t3 håller en register-adress. Lägg till 4 på denna adress oc
                            # innehållet från den resulterande adressen.
# Hoppinstruktioner         # Hoppa till L1 om:
beq t0, t1, L1              # t0 == t1
bne t0, t1, L1              # t0 != t1
bgt t0, t1, L1              # t0 > t1
bge t0, t1, L1              # t0 >= t1
blt t0, t1, L1              # t0 < t1
ble t0, t1, L1              # t0 <= t1
```

Syntetiska hoppinstruktioner kan även ta konstanter som andra argument.

Direktiv Globala variabler

```
.data                       # Tala om att det följer data
tal1:       .word 0         # Reservera plats för ett ord och fyll med 0
f1:         .float 0.0      # Flyttal
tal2:       .word 0
empty_vek:  .space 20*4     # Tom vektor med 20 element, vardera 4 byte
```

I programmet kan man sedan använda talen som adresser. Andra direktiv kan t.ex. vara:

```
.text                       # Det som följer är maskininstruktioner.
```

Vektorer

En vektor definieras om en sammanhängande samling data som i högnivåspråk delas in i fack och åtkomst ges via vektor[i]. I assembly deklarerar man en vektor globalt enligt följande:

```
.data
vek:
    .word 0                 # vek[0] = 0;
    .word 5                 # vek[1] = 5;
    .word 10                # vek[2] = 10;
```

Konstanter

En konstant deklareras ungefär som i C:

```
.data
#define pi 3.14
#define e 2.17
```

Detta är nyttigt att använda t.ex. när man ska lagra structs.

```
//C-kod
typedef struct {
    float re;
    float im;
} complex;

complex vek[3];
float t0 = vek[0].re;
flaot t1 = vek[0].im;

# Assembly
.data
#define re 0
#define im 4
vek:
    .space 3*(4+4)          # Tom vektor med 3 element, vardera håller två om 4 byte
    .text
main:
   lw t0, re(vek)           # t0 = vek[0].re;
   lw t1, im(vek)           # t1 = vek[0].im;
```

Kodkonventioner för rutiner och likande

Ifall en subrutin ska använda något av s-registerna eller kalla på ännu en subrutin, måste de spara undan de använda på stacken. Stacken kan liknas vid en hög tallrikar. Där de tallrikar som ligger högst upp har högst adress. Registret sp", stack pointer, pekar på det element som ligger underst längst ner av de använda minnescellerna. Ifall man ska lägga saker på stacken. Så lägger man dem i praktiken längst ner av de använda adresserna". Detta kan göras i kod enligt:

```
    .globl reserveSpace
    .ent reserveSpace
reserveSpace:
    subu sp, sp, 24         # Flyttar sp "nedåt", måste vara ett tal % 8 == 0
    sw ra, 0(sp)            # Följande rader fyller på "nedifrån och uppåt"
    sw s0, 4(sp)
    sw s1, 8(sp)
    sw s2, 12(sp)
    sw s3, 16(sp)
    sw s4, 20(sp)           # Närmast gamla sp
```

Märk väl att ifall en subrutin tar argument, så måste de gamla argumenten också sparas undan på stacken. Man behöver däremot inte ta hänsyn till t-registerna, så länge man själv inte har använt dem och behöver värdet efter subrutinanropet.

När man börjar en subrutin med att spara undan saker på stacken så kallas detta för **prolog**. Likadant så kallas återställningen i slutet av rutinen för **epilog**. För att återställa stackpekaren använder man:

```
addu sp, sp, 24             # När man använt subu sp,sp,24 i prologen
```

Läsning och skrivning till portar

```
# Läsning och skrivning till en 8-bitars I/O-port
    lui s0, 0xbfa0          # Lägg in de mest signifikanta av de 32 bitarna i s0
    ori s0, s0, 0x0002      # Lägg in de minst signifikanta
L1:
    lb s1, 0(s0)            # Läs in den byte som ligger på porten
    nop
    sll s1, s1, 1           # Multiplicera med 2
    sb  s1, 0(s0)           # Skriv till samma port
    b L1                    # Upprepa
    nop
```

Trådar

Semaforer: En semafor är en datastruktur som fungerar ungefär som ett *lås*, men med mer flexibilitet. Man kan göra två operationer på en semafor: **wait** och **signal**. En semafor används i ett system med flera trådar eller processer. Dess funktion är att skydda gemensamma resurser genom ömsesidig uteslutning.

En implementation av de två operationerna kan se ut så här:

```
void wait(sem S)
{
    while (S.lock <= 0);
    disable_interrupts();
    S.lock--;
    enable_interrupts();
}

void signal(sem S)
{
    disable_interrupts();
    S.lock--;
    enable_interrupts();
}
```

En tråd måste alltså innan den ska använda en gemensam resurs kalla wait på dess semafor, och sedan efter användning kalla på signal.

Trådar befinner sig i tre olika tillstånd: **exekverande, redo** och **blocked**. Ett systemanrop sätter en exekverande tråd till blocked. Därefter blir denne

"unblocked"vid en definierad händelse. Processorn väljer från de som befinner sig i redo-stadiet och låter de exekvera. Tiden för exekvering kallas för **timeslice/tidskvantum**. När man byter mellan trådar kallas detta för **threadswitch/context switch**.

Round-robin: Alla trådar får exekvera lika länge.

Prioritet: Round-robin, men endast mellan trådar med samma prio. Så länge de högre prioriterade trådarna är blockerade så får de tråder med lägre prioritet exekvera.

Ömsesidig uteslutning: för kritiska exekveringsbrott kan man låsa resurser. Som sagt används detta av semaforer.

Övriga begrepp

D/A-omvandlare: Tar ett binärt tal och omvandlar detta till en analog signal t.ex. en spänning

A/D-omvandlare: Tar en analog insignal (spänning, sinuskurva) och omvandlar denne till ett binärt tal

RAM: random access memory

SIMM: Single in-line memory module, RAM-minnes konstruktion

ROM: Read-only memory

EPROM: erasable programmable memory

ISA: instrction set architecture, arkitekturen hos en processor som definierar hur man kan programmera den

Procesor: Processorns uppgift är att utföra logiska och aritmetiska funktioner på data, samt styra flödet av instruktioner. Denna består av:

- **Register:** Används för att temporärt lagra data inne i processorn; så att denne inte behöver hämta från minnet lika ofta. Registeråtkomst är snabbare än minnesåtkomst (de sitter på samma chip som processorn.
- **ALU:** arithmetic logic unit. I moderna processorer finns det flera ALU. Ofta en för flyttal utöver standard-ALU:n.