Datorteknik
Labbserie med Arduino/KPAD
Kodgranskning

Uppdateras löpande. Se datumet:

14 december 2020

Inledning

Det här dokumentet är både för dig som student, labassistent och examinator.

Kontrollera din kod gentemot reglerna nedan. Din kod måste inte följa dem till punkt och pricka men andemeningen i dem skall följas. Du har en viss ”konstnärlig frihet” med din kod men den friheten sträcker sig inte till att skriva onödigt komplicerad kod.

Kodreglerna är resultatet av åtskilliga års assemblererfarenhet med studenter.

Återkoppling på inlämnad kod kan ske med nummer under respektive labb.

Allmänt om koden

Står valet mellan lättläst kod eller mindre kodmängd vinner alltid lättläst.

1 Programrader För att ge ett lättläst program skall varje programrad disponeras så att

- *Labels* börjar i kolumn 0 och på egen rad om inte läsbarheten försämras
- Instruktioner, `.equ`, `.org` och motsv börjar ett tab-stop in på raden
- Argument kan börja efter mellanslag eller ytterligare ett tab-stop
- Kommentarer utförs som

`;`	för kommentar till radslut
`//`	för kommentar till radslut
`/* */`	för längre block

  Kommentarer med ’`;`’ börjar senare i en bestämd kolumn, helst samma för alla kommentarer.

2 Struktur Koden måste vara strukturerad. Det betyder här att den skall vara uppbyggd av subrutiner. Nu räcker det inte med att göra om allt till subrutiner utan rutinerna måste vara ändamålsenliga och ”göra vad de heter”.

Man kan dra det hela så långt att huvudprogrammet kanske enbart består av tre subrutin-anrop à la:

```
    call    INIT_HARDWARE
MAIN:
    call    GET_INPUT
    call    PROCESS
    jmp     MAIN
```

Det är fullt rimligt att koden består av flera subrutiner under programframtagning och felsökning som mot slutet slås ihop till färre rutiner för att snygga upp programflödet och ge tydligare kod.[1]

[1] Kod som man kanske inte skulle kunnat ta fram om man skrev för stora oorganiserade kodstycken

3 Ansvarsområde Tänk på vilket *ansvarsområde* en rutin har, det skall framgå av rutinens namn (*label*). En subrutin som heter `LINE_PRINT:` förväntas skriva ut en `LINE` och inget annat. Det är generalfel om den bara skriver ut en *del av* en rad, eller skriver ut *flera* rader, eller skriver ut en rad *och* håller reda på klockan *och* tutar ett larm om klockan är 8. För att göra programmeringen enkel: Låt varje rutin göra en enda tydlig sak och namnge den därefter.

4 Subrutiner Varje subrutin skall disponeras enligt följande ('|' ditsatta för att markera andra programrader)

```
EN_RUTIN:
    push    r16
    |
AAA:
    |
    jmp     BBB
CCC:
    call    DDD
    |
    rjmp    FFF
BBB:
    |
    breq    AAA
    rjmp    CCC
FFF:
    pop     r16
    ret
```

- Notera att hopp (`jmp`/`br*`) inte får ske *mellan* subrutiner, enbart *inom* en rutin.
- Notera också speciellt att varje subrutin har *en* väldefinierad ingång och endast ***en*** (1) utgång `ret`, och denna `ret` skall vara sista raden i subrutinen. Det är alltså inte tillåtet att göra returer från flera ställen i rutinen.[2]
- Huvudflödet i en rutin skall vara nedåt.
- Håll ihop rutiner som har med varann att göra. Sprid inte kod överallt.
- Det är helt i sin ordning att ha korta subrutiner för ökad läslighet:

  ```
  NEXT_BIT:
      lsl     r16
      ret
  ```

- Använd så få hopp som möjligt. Med bättre villkor kan det bli färre rader kod som här

  ```
      |                       |
      breq    DIT             brne    BORT
      rjmp    BORT        DIT:
  DIT:                        |
      |                       rjmp    EXIT
      rjmp    EXIT        BORT:
  BORT:                       |
      |                   EXIT:
  EXIT:                       ret
      ret
  ```

  Med reglerna ovan undviker du att koden "tarmar" ut sig, utan håller sig inom sin rutin.

från början? Det blir en individuell uppfattning om tycke och smak. För många rutiner förvirrar, för få försvårar felsökning.

[2]Detta ger också mindre programstorlek i och med att uppsamlingskod (`pop` &c) enbart behöver förekomma på ett enda ställe per subrutin.

5 Variabler med mera för ökad läslighet och minimera sidoeffekter genom att

- *Undvika* globala variabler.
- *Använda* lokala variabler där det krävs.
- *Använda* `.equ` och `def` där dessa är motiverade.
- *Använda* labels även i `.dseg`.
- *Använda* pekare där det är rimligt.
- *Använda* ASCII-tecken (`ldi r16,'A'`) i stället för hexkod för tydlighets skull.
- *Använda* `push` och `pop` men glöm inte variabler på stacken.

6 Generalitet Koden skall vara generell där så är möjligt. Det är exempelvis begränsande (och i denna kurs alltså fel) att skriva kod som bara kan göra AD-omvandling på en kanal. Skriv istället en rutin som tar kanalnumret som argument. Vill man sedan göra en rutin för omvandling av till exempel ett tempsensorvärde gör man detta i en egen rutin med lämpligt beskrivande namn:

```
AD_READ:                ;  in: channel i r16
        |               ; out: measured value 0-255 i r16
        ret

READ_TEMP:              ; get temp in r16 (0-255 deg C)
    ldi     r16,3       ; sensor is channel 3
    call    AD_READ     ; read value
    call    VALUE_TO_C  ; convert to human readable form
    ret
```

Nu kan `AD_READ` användas för att läsa av joystickar och tryckknappar osv också.

7 Ett tips är att markera kodavsnitt och subrutiner som hänger ihop så de är lätta att hitta i kodmassan när man scrollar. Till exempel:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; LCD stuff
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;
    ; LCD_OPEN
    ;
LCD_OPEN:
    |
    ret

    ;
    ; LCD_PRINTZ, print asciiz at Y
    ;
LCD_PRINTZ:
    |
    ret
```

Lab2 Morsesändare

Erfarenheter och tips från feedback till studenter vid tidigare kursomgångar ger detta:

1. Det finns inget krav att programmet ska sluta längst ner i assemblerfilen. Det kan sluta där det känns enklast:

```
    cpi     r16,0
END:
    breq    END
```

```
END:
    rjmp    END
```

2. `LOOKUP:` gör en tabelluppslagning. Det är onödigt komplicerat att låta denna rutin också hantera eventuella specialfall. Det är lättare att skilja bort specialfall innan anropet.

3. Längre `DELAY:` löses med *hoprullade* loopar inte *utrullade*

```
    ; inte detta
LDELAY:
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    call    DELAY
    ret
```

```
    ; hellre detta
LDELAY:
    ldi     r16,10
LD:
    call    DELAY
    dec     r16
    brne    LD
    ret
```

```
    ; men helst detta
    ; anrop med argument
    ldi     r16,10
    call    LDELAY

    ;
    ;
LDELAY:
    call    DELAY
    dec     r16
    brne    LDELAY
    ret
```

4. Programmet skall kunna sända godtyckligt lång text (generalitet igen). För att vara ett generellt program skall hela den textsträng som ska sändas skrivas in

```
.db 'MICJO MICJO MICJO',0
```

   och inte

```
.db 'MICJO',0
```

   kört tre gånger eller motsvarande.

   Bestämmer du dig för att sända strängen om och om igen är det OK att inleda varje sändning med en längre paus som

```
.db '   MICJO MICJO MICJO',0
```

Lab3 Digitalur

Erfarenheter och tips från feedback till studenter vid tidigare kursomgångar ger detta:

1. Konfigurera avbrott innan du släpper på dem.
2. Avbrottsrutiner skall spara `SREG`.
3. Avbrottsrutinen i sin helhet ska inte ha några sidoeffekter. Även kod av följande typ ska fungera

```
AVBROTT:
    :
    call    TIME_TICK
    :
    reti

MAIN:
    call    TIME_FORMAT
    call    LINE_PRINT
    jmp     MAIN
```

4. `LINE_PRINT:` måste kunna skriva ut en hel rad om 16 tecken.
5. Både BCD-uppräkningen och `TIME_FORMAT:` är så repetetiva i sin natur att de bara ber om att loopas. De skall alltså rullas ihop. Den försiktige kodaren gör de dock först utrullade för att få sin funktion.
6. Avbrottsvektor 0 är programstart. Lägg inte övrig kod här. Riskera speciellt inte att använda samma adresser som `OCR1Aaddr`.
7. Använd pekare! Det är ofta lämpligt.

Lab4 Radeditor