



GDB och *SimServer* med *MD407* inledande övning 2

Under inledande övning 1 använder vi *ETERM8* som ett enkelt och grundläggande verktyg för programutveckling i assemblerspråk. Med *ETERM8*:s debugger tillsammans med simulatoren *SimServer* har vi där sett hur man kan studera processorns registerinnehåll, även följa instruktionsexekveringen i laborationsdatorn och samtidigt undersöka minnets innehåll. Detta kallas allmänt *machine level debugging*.

Under denna övning introducerar vi kraftfullare verktyg avsedda att användas för ingående test och felsökning. Vi visar med exempel hur du kan använda *GDB*, *GNU Debugger* för test, felsökning och felavhjälpning i program. Med *GDB* tar vi steget upp till *source level debugging* (SLD) vilket gör att vi kan följa programexekvering och ändringar i processorns register, ändringar i minnesinnehåll etc. utgående från vår källtext. Vi använder samma simulator, *SimServer*, som tidigare, men gränssnittet är nu annorlunda.

Under övningen visar vi hur du kommer i gång med *GDB* och använder några enkla grundläggande kommandon. Det här är en väldigt liten del av vad du kan göra med *GDB* och övningen ska i första hand ska ge dig insikter i hur ett applikationsprogram kan övervakas och kontrolleras enbart genom att följa dess källtexter.

Anmärkningar:

Anvisningar i detta häfte förutsätter att du arbetar med en fungerande installation av *ETERM8* och *SimServer*.

INNEHÅLL

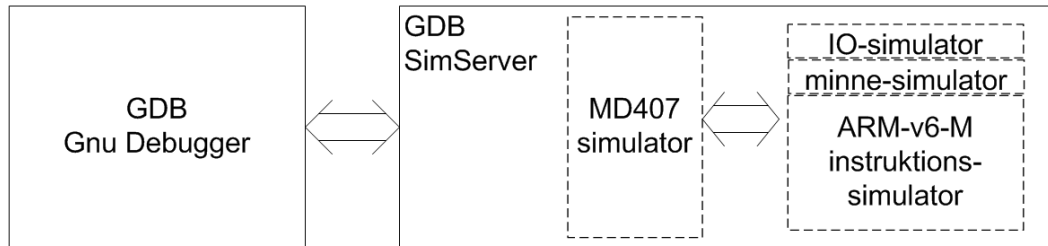
Användning av <i>GDB</i>	2
Snabb start med <i>GDB</i> och <i>SimServer</i>	2
Grundläggande kommandon.....	3
Instruktionsvis exekvering.....	4
Programexekvering.....	5
Brytpunkter.....	7

Version: 2024-02-18 10:19:00

Användning av GDB

I programmet *SimServer* finns samma simulator som du arbetade med i kapitel 1 och 2. Skillnaden är att du nu inte har *ETERM8*:s debugger som gränssnitt, utan *GDB*. *GDB* och *SimServer* kommunicerar via *sockets*, kortfattat en metod för processkommunikation i operativsystemet.

Utän att gå in på detaljer innebär detta att två program startas med förutsättningen att dom kan utväxla data via en *port*. Porten anges, i båda programmen i formatet "värd dator:unik ID". Om man vill upprätta processkommunikation i samma maskin blir alltid värd datornamnet *localhost*. Programmet *SimServer* "lyssnar" exempelvis på porten med ID 1234. Den fullständiga kanalen för kommunikation med *SimServer* blir då *localhost:1234*.



FIGUR 0.1

UPPGIFT 1

Använd *ETERM8* och förbered följande enkla program för att undersöka *GDB* och *SimServer*.

```
@ gdbtest1.asm
start:
    MOV    R0,#0
main:
    ADD    R0,R0,#1
    B     main
```

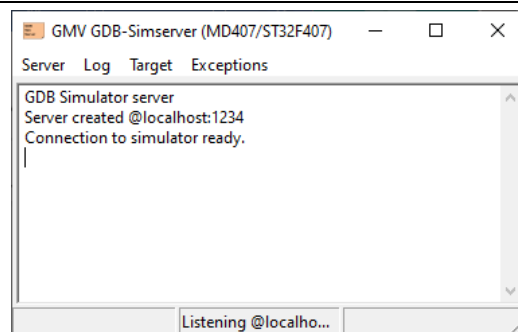
- Redigera en källtext, *gdbtest1.asm* med programmet och spara denna.
- Assemblera och rätta eventuella fel. Du behöver *gdbtest1.elf* senare.

...

Snabb start med GDB och SimServer

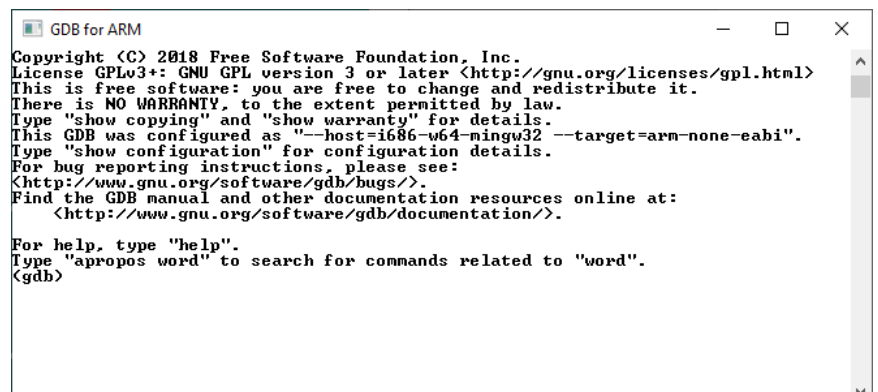
- Starta *SimServer*

Simulatorn startar och väntar nu på att *GDB* ska börja kommunicera. Här använder vi porten 1234 för kommunikationen.



- Starta *GDB*

Ett konsollfönster med *GDB* öppnas: *GDB* identifierar sig och skriver ut sin prompter (*gdb*) vilket betyder att man kan börja arbeta med debuggern.



Grundläggande kommandon

Det första man bör göra är att ändra arbetsbibliotek till det ställe där man har sina filer. I vårt fall där vi tidigare skapade `gdbtest1.asm`, arbetsbiblioteket: `D:\mop\lab1`, vi ger därför kommandot:

```
(gdb) cd d:\mop\lab1
```

GDB svarar:

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) cd d:\mop\lab1
Working directory d:\mop\lab1.
(gdb)
```

Nu anger man den fil man vill arbeta med. I vårt fall är det objektfilen `gdbtest1.elf`.

I GDB ger vi därför kommandot:

```
(gdb) file gdbtest1.elf
```

GDB svarar:

```
Working directory d:\mop\lab1.
(gdb) file gdbtest1.elf
Reading symbols from gdbtest1.elf...done.
(gdb)
```

Detta betyder att GDB läst in objektfilen, med symbolinformationen.

För att koppla ihop GDB med vår MD407-simulator *SimServer*, ger du kommandot:

```
(gdb) target extended-remote localhost:1234
```

GDB svarar:

```
Reading symbols from gdbtest1.elf...done.
(gdb) target extended-remote localhost:1234
Remote debugging using localhost:1234
0xffffffff in ?? (<)
(gdb)
```

Med `load`-kommandot, laddas nu kod och data till *SimServer*:

```
(gdb) load
```

GDB svarar:

```
Remote debugging using localhost:1234
0xffffffff in ?? (<)
(gdb) load
Loading section .text, size 0x8 lma 0x20000000
Start address 0x20000000, load size 8
Transfer rate: 64 bits in <1 sec, 8 bytes/write.
(gdb)
```

Du kan nu låta *SimServer* göra nödvändiga förberedelser för debuggning med kommandot:

```
(gdb) monitor restart
```

Med `list`-kommandot kan du studera programmets källtext, exempelvis med

```
(gdb) list 1
```

GDB svarar:

```
Transfer rate: 64 bits in <1 sec, 8 bytes/write.
(gdb) list 1
1      @ gdbtest1.asm
2      start:
3          MOU      R0,#0
4      main:
5          ADD      R0,R0,#1
6          B        main
7
(gdb)
```

Ett annat användbart kommando är `info`, för att studera processorns registerinnehåll:

```
(gdb) info registers
```

GDB svarar:

```
(gdb) info registers
R0      0x0      0x0
R1      0x0      0x0
R2      0x0      0x0
R3      0x0      0x0
R4      0x0      0x0
R5      0x0      0x0
R6      0x0      0x0
R7      0x0      0x0
R8      0x0      0x0
R9      0x0      0x0
R10     0x0      0x0
R11     0x0      0x0
R12     0x0      0x0
SP      0xffffffff 0xffffffff
LR      0xffffffff 0xffffffff
PC      0x20000000 0x20000000 <start>
XPSR    0x10000000 0x10000000
MSP     0xffffffff 0xffffffff
PSP     0x0       0x0
CONTROL 0x0       0x0
FAULTMASK 0x0      0x0
BASEPRI 0x0       0x0
PRIMASK 0x0       0x0
(gdb)
```

Är man bara intresserad av ett enstaka registers innehåll anger man det, exempelvis:

```
(gdb) info register r0
```

GDB svarar:

```
PRIMASK 0x0      0x0
(gdb) info register r0
r0      0x0      0x0
(gdb)
```

Instruktionsvis exekvering

För att initiera simulatorm för programexekvering ger du kommandot:

```
(gdb) monitor restart
```

```
(gdb) monitor restart
User restart target
(gdb)
```

För att utföra en maskininstruktion används kommandot `steppi` (step instruction):

```
(gdb) steppi
```

GDB låter *SimServer* utföra en assemblerinstruktion och svarar med att skriva ut nästa instruktion, dvs. den som nu står i tur att utföras:

```
r0      0x0      0x0
(gdb) steppi
main () at D:\mop\lab1\gdbtest1.asm:5
5      ADD    R0,R0,#1
(gdb)
```

Genom att ge argument (antal instruktioner) till `stepi` kan vi få *GDB* att utföra flera instruktioner i ett kommando, exempelvis kan vi stega ytterligare 5 instruktioner med kommandot:

```
(gdb) stepi 5
```

Återigen svarar *GDB* med att skriva ut den instruktion som står i tur att utföras.

```
5      ADD     R0, R0, #1
(gdb) stepi 5
6      B       main
(gdb)
```

Kontrollera nu hur register `R0` påverkats av de utförda instruktionerna genom att ge kommandot:

```
(gdb) info register r0
```

```
6      B       main
(gdb) info register r0
r0     0x3     0x3
(gdb)
```

Programexekvering

UPPGIFT 2

Använd *ETERM8* och förbered nu nästa program för att undersökas med *GDB* och *SimServer*. Observera att detta är samma program som i `mom1.asm`.

```
@ gdbtest2.asm
start:
    LDR    R0, =0x55555555
    LDR    R1, =0x40020C00 @ konfig port D
    STR    R0, [R1]

    LDR    R1, =0x40020C14 @ utport GPIO D
    LDR    R2, =0x40021010 @ inport GPIO E

main: LDR    R0, [R2]
      STR    R0, [R1]
      B     main
```

- Redigera en källtext, `gdbtest2.asm` med programmet och spara denna.
- Assemblera och rätta eventuella fel.

Vi övergår nu till programmet i `gdbtest2.asm`, ge ett nytt *file*-kommando till *GDB*:

```
(gdb) file gdbtest2.elf
```

Denna gång kommer *GDB* att fråga om du verkligen vill byta fil och dess symboltabell, svara 'y' på frågorna:

```
(gdb) file gdbtest2.elf
A program is being debugged already.
Are you sure you want to change the file? (y or n) y
Load new symbol table from "gdbtest2.elf"? (y or n) y
Reading symbols from gdbtest2.elf...done.
(gdb)
```

Den nya filen måste också laddas till *SimServer*:

```
(gdb) load
```

```
(gdb) monitor restart
```

Prova nu några alternativa list-kommandon, vi har exempelvis två symboler definierade, start och main, ge kommandot:

(gdb) list start

studera utskriften från *GDB*, ge därefter kommandot:

(gdb) list main

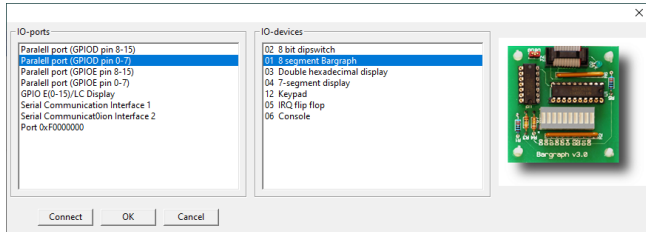
Vill du veta mer om list-kommandot, ge *GDB* kommandot

(gdb) help list

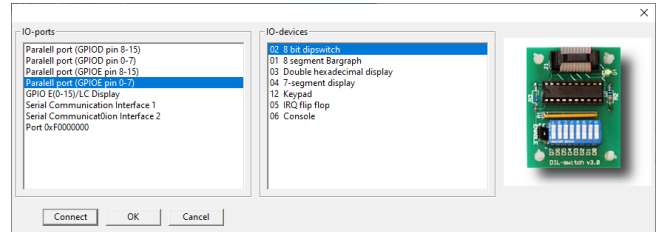
KOPPLA IO-ENHETER TILL SIMSERVER

I *SimServer*: (Server | IO Setup)

- Koppla en diodramp till port D0-7



- Och en strömställare till port E0-7.



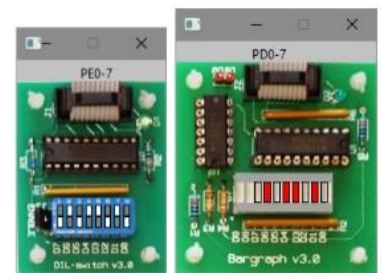
Klicka OK för att stänga dialogen.

Ställ nu in värdet 0x5A på strömställaren. Starta därefter programmet med c- (continue) kommandot:

(gdb) c

```
<gdb> c
Continuing.
```

Programmet exekveras nu oavbrutet, du kan verifiera det genom att ändra strömställarens lägen och observera IO-enheterna, strömställarens inställda värde ska kopieras till diodrampen.



Du har inte fått någon ny prompter från *GDB*, du kan därför inte ge några nya kommandon under tiden programmet exekveras av *GDB*. För att få *GDB* att avbryta simulatören ger du kommandot <ctrl-c>:

```
<gdb> c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
main (<) at D:\mop\lab1\gdbtest2.asm:10
10 main: LDR R0, [R2]
<gdb>
```

Anm: I detta fall råkade programmet stannas på rad 10, men det kan stoppa på vilken rad som helst i programslingan.

Prova med att starta (c = continue):

(gdb) c

och stoppa (<ctrl-c>) programmet några gånger.

Brytpunkter

För att slippa stega sig fram instruktionsvis till en bestämd punkt i programmet kan man först sätta ut brytpunkter och därefter använda c-kommandot.

Vi ska nu starta om programmet och ger därför på nytt kommandon:

```
(gdb) load
```

```
(gdb) monitor restart
```

Vill vi veta adresserna till instruktionerna i 'main' ger vi kommandot:

```
(gdb) disassemble main
```

GDB svarar nu:

```
(gdb) load
Loading section .text, size 0x20 lma 0x20000000
Start address 0x20000000, load size 32
Transfer rate: 15 KB/sec, 32 bytes/write.
(gdb) disassemble main
Dump of assembler code for function main:
0x2000000a <+0>:    ldr    r0, [r2, #0]
0x2000000c <+2>:    str    r0, [r1, #0]
0x2000000e <+4>:    b.n   0x2000000a <main>
0x20000010 <+6>:    strb  r5, [r2, r5]
0x20000012 <+8>:    strb  r5, [r2, r5]
0x20000014 <+10>:   lsr   r0, r0, #16
0x20000016 <+12>:   ands  r2, r0
0x20000018 <+14>:   lsr   r4, r2, #16
0x2000001a <+16>:   ands  r2, r0
0x2000001c <+18>:   asrs  r0, r2, #32
0x2000001e <+20>:   ands  r2, r0
End of assembler dump.
(gdb)
```

Observera att från adress 0x20000010 och framåt finns det instruktioner vi inte känner till från vårt program. Allt är i sin ordning, det är i själva verket data som tolkas som instruktioner av disassemblatorn.

Vi kan nu enkelt sätta en brytpunkt, exempelvis på STR-instruktionen:

```
(gdb) break *0x2000000c
```

GDB svarar:

```
(gdb) break *0x2000000c
Breakpoint 1 at 0x2000000c: file D:\mop\lab1\gdbtest2.asm, line 11.
(gdb)
```

För att få en översikt av brytpunktstabellen ger du kommandot

```
(gdb) info break
```

GDB svarar:

```
(gdb) info break
Num   Type           Disp Enb Address      What
1     breakpoint     keep y   0x2000000c  D:\mop\lab1\gdbtest2.asm:11
(gdb)
```

Starta programmet och exekvera fram till brytpunkt med c-kommandot, vid brytpunkten svarar GDB:

```
(gdb) c
Continuing.

Breakpoint 1, main () at D:\mop\lab1\gdbtest2.asm:11
11          STR    R0, [R1]
(gdb)
```

APPENDIX: NÅGRA ANVÄNDBARA GDB-KOMMANDON

Kommando	Beskrivning
help	Lista hjälp-ämne
help <i>ämne-klass</i>	Lista hjälp-klasser
help <i>kommando</i>	Ge hjälp om specifikt kommando
info break	Lista brytpunkter
info registers	Lista registerinnehåll
info register <rn>	Lista specifikt registerinnehåll
info line <i>number</i>	Visa start och slutposition för rad number i källtexten
continue, c	Exekvera till nästa brytpunkt
<ctrl-c>	Stoppa programexekvering
quit, q	Avsluta GDB
disassemble <i>0xstart</i>	Disassemblera från en adress i minnet
stepi, si	Utför en instruktion (stega in i subrutin)
nexti, ni	Utför en instruktion fullständigt (hel subrutin)
x <i>0xaddress</i>	Undersök minnesinnehåll
Brytpunkt	
break <i>function-name</i>	Sätt en brytpunkt i programmet
break <i>line-number</i>	Sätt en brytpunkt i programmet
break <i>*address</i>	Sätt en brytpunkt på en adress, (ingen källtext given)
clear	Ta bort brytpunkter
clear <i>function</i>	Ta bort alla brytpunkter i <i>function</i>
clear <i>line-number</i>	Ta bort brytpunkt på rad <i>line-number</i>
delete, d	Ta bort alla typer av brytpunkter
continue, c	Fortsätt programexekvering till nästa brytpunkt
finish	Exekvera till slutet av denna funktion
Programexekvering	
stepi, si	Utför en assemblerinstruktion
step, s	Stega en programrad, in i funktioner vid funktionsanrop
step <i>antal-steg</i>	Stega en programrad, in i funktioner vid funktionsanrop
next, n	Stega en programrad, över funktioner vid funktionsanrop
next <i>number</i>	Stega en programrad, över funktioner vid funktionsanrop
where	Visa radnummer och funktion
Stack	
frame, f	Visa aktuell aktiveringspost Använd "up/down" för att flytta mellan aktiveringsposter
up	Visa föregående aktiveringspost
down	Visa nästa aktiveringspost
info args	Visa parametrar och lokala variabler
info locals	Visa parametrar och lokala variabler
Källkod	
list, l	Lista källkod
list <i>line-number</i>	Lista källkod
list <i>function</i>	Lista källkod
list <i>start#,end#</i>	Lista källkod
list <i>filename:function</i>	Lista källkod
set listsize <i>count</i>	Sätt/visa antalet källkodsraderna som ska visas av list-kommandot
show listsize	Sätt/visa antalet källkodsraderna som ska visas av list-kommandot
Undersöka variabler	
print <i>variable-name</i>	Visa variablers värde
p <i>variable-name</i>	Visa variablers värde
p <i>file-name::variable-name</i>	Visa variablers värde
p ' <i>file-name</i> :: <i>variable-name</i>	Visa variablers värde
ptype <i>variable</i>	Visa variablers typ
ptype <i>data-type</i>	Visa variablers typ

Studiematerial, Maskinorienterad programmering

Tryckt häfte

- *Quick Guide* – avsedd att användas under tentamen eftersom det normalt inte är tillåtet att använda utskrifter av PDF-version.

Programvaror: (Windows, Linux och MacOS)

- *ETERM8*
- *SimServer*
- *CodeLite*, lämplig distribution till detta material.
- *GCC för Windows (Mingw64)*.
- *GCC korskompilator för ARM*, distribution som kompletterats med programbibliotek för *MD407*.

Dokument (PDF):

- *Quick Guide* - elektronisk form av det tryckta häftet
- *IO-simulator*, referenshandbok.
- *En snabb introduktion till maskinorienterad C* – kompendium som behandlar C speciellt som ett maskinnära programspråk.
- *ETERM8 och MD407, inledande övning 1* – introducerande övning inför laborationer.
- *GDB och SimServer med MD407, inledande övning 2* – introducerande övning inför laborationer.
- *GCC och SimServer med MD407, inledande övning 3* – introducerande övning inför laborationer.
- *Laborationsuppgifter med simulator, MD407* – anvisningar för att genomföra en hel laborationsserie. Samtliga uppgifter har här utformats för att kunna genomföras enbart med hjälp av de programvaror som omfattas i kursen. Det behövs alltså inte tillgång till laborationsutrustningen. För flertalet uppgifter hänvisas direkt till arbetsboken, du behöver därför även denna för att arbeta med detta häfte.