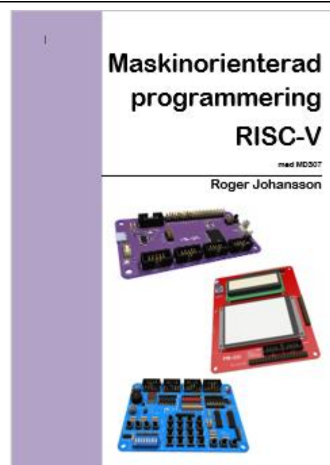


GDB och *SimServer* med *MD307*

inledande övning 2

Under inledande övning 1 använde vi *ETERM8* som ett enkelt och grundläggande verktyg för programutveckling i assemblerspråk. Med *ETERM8*:s debugger tillsammans med simulatorn *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 genom att följa programmet via dess källtext.

Anmärkningar:
Anvisningar i detta häfte förutsätter att du har fungerande installationer av *ETERM8* och *SimServer*.

INNEHÅLL

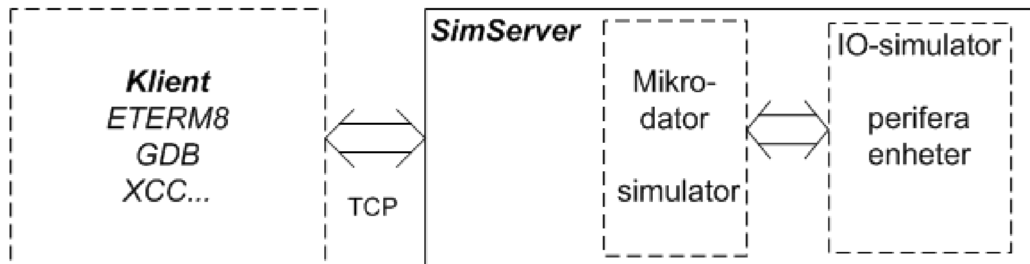
Användning av GDB 2
Snabb start med GDB och SimServer 2
Grundläggande kommandon 3
Instruktionsvis exekvering 5
Programexekvering 5
Brytpunkter 7

Version: 2024-12-01 17:35:00

Användning av GDB

Programmet *SimServer* kan användas tillsammans med olika typer av debuggers, så kallade *klienter*. Under inledande övning 1 kunde du se hur *ETERM8* används tillsammans med *SimServer*. *ETERM8* erbjuder dock bara debugging på *maskinnivå*, dvs. det finns ingen direkt koppling till programmets källtext. Under denna övning ska vi använda *SimServer* tillsammans med *GNU GDB*, en mångsidig och mycket användbar debugger för program på *källtextnivå*.

SimServer kommunicerar alltså med någon debugger via *sockets*, kortfattat en metod för processkommunikation i operativsystemet. Utan 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 ”värddator:unikt ID”. Om man vill upprätta processkommunikation i samma maskin blir alltid värddatornamnet `localhost`. Programmet *SimServer* ”lyssnar” exempelvis på porten med ID 1234. Den fullständiga kanalen för kommunikation med *SimServer* blir då `localhost:1234`. För en utförlig beskrivning av *SimServer*, se ”*SimServer/IO-simulator referenshandbok*”



FIGUR 1

Den variant av *GDB* vi nu använder till *MD307* (riscv32-unknown-elf-gdb) installerades tillsammans med *ETERM8*. Det är en terminalapplikation som kan startas via en länk från skrivbordet (*Windows*).

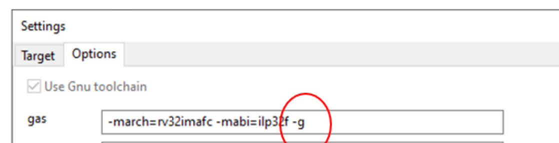
ÖVNING 1

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

```
# gdbtest1.asm
start:
   li     t0,1      # t0<-1
main:
   add    t0,t0,1   # t0<-t0+1
   j      main      # jump 'main'
```

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

Anm flaggan `'-g'` som ges vid assembleringen.
(Tools|Settings|Options).
... krävs för att assemblatorn ska generera den speciella källtextinformation som krävs för källtextdebugging.

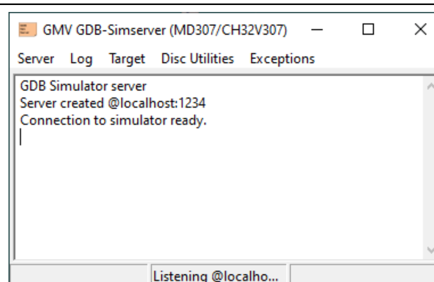


...

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.

- **Anm**. Versionsinformation i figurens text kan skilja sig från din aktuella version.

```
GDB for RISC-V
GNU gdb ('riscv32-embecosm-gcc-win64-20230702') 14.0.50.20230702-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-w64-mingw32 --target=riscv32-unknown-e
lf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<'https://www.embecosm.com'>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

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...
(gdb)
```

Detta betyder att *GDB* läst in objektfilen, med symbolinformationen.

För att koppla ihop *GDB* med vår *MD307*-simulator *SimServer*, ger du kommandot:

(gdb) target extended-remote localhost:1234

GDB svarar:

```
(gdb) target extended-remote localhost:1234
Remote debugging using localhost:1234
warning: Architecture rejected target-supplied description
0x00000000 in ?? ()
(gdb)
```

Här ser vi en varning från *GDB*, den innebär kortfattat att *SimServer* inte delar samma beskrivning av RISC-V registren. I vårt fall har varningen ingen betydelse.

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

(gdb) load

GDB svarar:

```
(gdb) load
Loading section .text, size 0x8 lma 0x20000000
Start address 0x20000000, load size 8
Transfer rate: 2 KB/sec, 8 bytes/write.
(gdb)
```

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

(gdb) monitor restart

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

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

(gdb) list 1

GDB svarar:

```
(gdb) list 1
1       # gdbtest1.asm
2       start:
3               li      t0,1     # t0<-1
4       main:
5               add     t0,t0,1  # t0<-t0+1
6               j       main     # jump `main'
(gdb)
```

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

(gdb) info registers

GDB svarar:

```
(gdb) info registers
ra             0x0      0x0
sp             0x2001c000       0x2001c000
gp             0x0      0x0
tp             0x0      0x0
t0             0x0      0
t1             0x0      0
t2             0x0      0
fp             0x0      0x0
s1             0x0      0
a0             0x0      0
a1             0x0      0
a2             0x0      0
a3             0x0      0
a4             0x0      0
a5             0x0      0
a6             0x0      0
a7             0x0      0
s2             0x0      0
s3             0x0      0
s4             0x0      0
s5             0x0      0
s6             0x0      0
s7             0x0      0
s8             0x0      0
--Type <RET> for more, q to quit, c to continue without paging--
s9             0x0      0
s10            0x0      0
s11            0x0      0
t3             0x0      0
t4             0x0      0
t5             0x0      0
t6             0x0      0
pc             0x20000000       0x20000000 <start>
(gdb)
```

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

(gdb) info register t0

GDB svarar:

```
(gdb) info register t0
t0             0x0      0
(gdb)
```

Instruktionsvis exekvering

För att initiera simulatorn för programexekvering och återstarta programmet börjar du alltid med kommandot:

(gdb) monitor restart

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

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

(gdb) stepi

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

```
(gdb) stepi
main () at D:\mop\lab1\gdbtest1.asm:5
5               add     t0,t0,1 # t0<-t0+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.

```
(gdb) stepi 5
6               j       main    # jump `main'
(gdb)
```

Kontrollera nu hur register `t0` påverkats av de utförda instruktionerna genom att ge kommandot:

(gdb) info register t0

```
(gdb) info register t0
t0              0x4     4
(gdb)
```

Programexekvering

ÖVNING 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:   la    t1,0x40011400     # base address port D
         li    t0,0x44444444     # config, port D bits 0-7...
         sw    t0,0(t1)          # ...inport, floating input
         la    t2, 0x40011800    # base address port E
         li    t0,0x22222222     # config, port E bits 0-7...
         sw    t0,0(t2)          # ...outport, push/pull,
main:    lh    t0,8(t1)          # read 16 bit data from port D
         sh    t0,12(t2)         # write it to port E
         j     main              # and repeat...
```

- 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...
(gdb)
```

Den nya filen måste också laddas till *SimServer* och förberedas för start:

(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*.

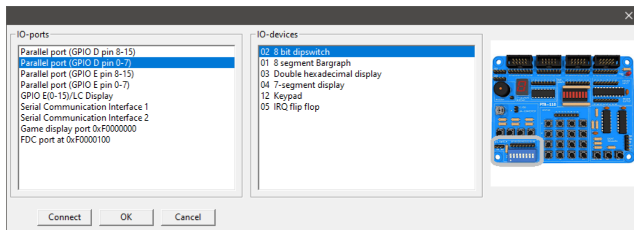
Vill du veta mer om list-kommandot, ge *GDB* kommandot

(gdb) help list

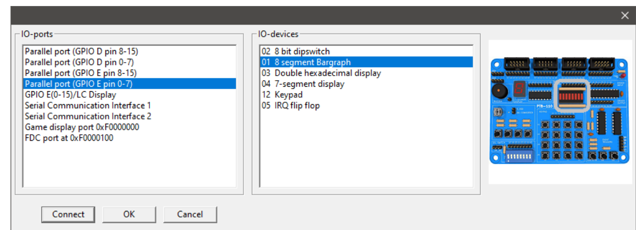
KOPPLA IO-ENHETER TILL SIMSERVER

I *SimServer*: (Server | IO Setup)

- Koppla en strömställare till port D0-7



- Och en diodramp till port E0-7.



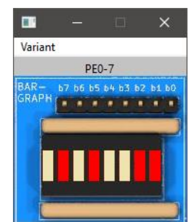
Klicka OK för att stänga dialogen.

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 simulatorn ger du kommandot <ctrl-c>:

```
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
main () at D:\mop\lab1\gdbtest2.asm:11
11              j       main            # and repeat...
(gdb)
```

Anm: I detta fall råkade programmet stannas på rad 1, 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 progammet 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) load
Loading section .text, size 0x34 lma 0x20000000
Start address 0x20000000, load size 52
Transfer rate: 12 KB/sec, 52 bytes/write.
(gdb) monitor restart
User restart target
(gdb) disassemble main
Dump of assembler code for function main:
   0x20000028 <+0>:     lw      t0,8(t1)
   0x2000002c <+4>:     sw      t0,12(t2)
   0x20000030 <+8>:     j       0x20000028 <main>
   0x20000032 <+10>:    unimp
End of assembler dump.
(gdb)
```

Observera att från adress 0x20000032 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 `sw`-instruktionen:

(gdb) break *0x2000002C

GDB svarar:

```
End of assembler dump.
(gdb) break *0x2000002c
Breakpoint 1 at 0x2000002c: file D:\mop\lab1\gdbtest2.asm, line 10.
(gdb)
```

Observer hur vi använder '*' för att ange för *GDB* att efterföljande text ska tolkas som ett heltal.

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   0x2000002c D:\mop\lab1\gdbtest2.asm:10
(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:10
10              sw      t0,12(t2)       # write it to port E
(gdb)
```

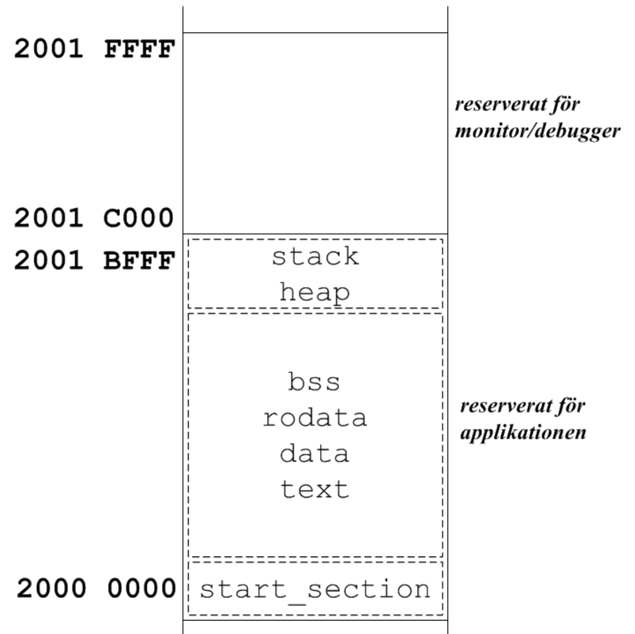
MINNESANVÄNDNING MD307, APPLIKATION I RWM

Låt oss nu studera den omgivning där våra program ska exekveras, dvs.hur applikationen byggs upp.

I figur 2 visas hur *RWM* disponeras i *MD307*. Vi har valt att placera applikationsprogrammet med start på minnets första adress, 20000000, och uppåt. Applikationens olika delar, *segment*, placeras här i den ordning figuren illustrerar. Detta styrs med *länkarskriptet* (`md307-ram.x`. Övre delen av minnet, är reserverat för *MD307*:s inbyggda monitor/debugger. Hos *MD307* används alltså maximalt 112kB av *RWM* för applikationsprogrammet medan resterande 16 kB har reserverats för det residenta programmet.

Programkod och data indelas i olika sektioner, figur 3 beskriver hur minnesdispositionen för ett komplett program, under exekvering, kan se ut. Figuren förstås bäst mot bakgrund av hur ett program översätts, sparas, laddas till primärminnet för att därefter utföras.

Applikationen består av en initierad del, *Image*, bestående av programkod och data. Den andra delen kallar vi *Run-Time*, Såväl programkod som data placeras i namngivna *sektioner* som kan ha godtyckliga namn. I figuren används exempelvis sektionerna `bss`, `data`, `rodata` och `text`. Dessa namn används också av *GCC* vid översättningen.



FIGUR 2: ANVÄNDNING AW RWM I MD3407

prefix

Prefixet, det som också kallas *startupkod*, placeras först. Prefixet ingår ofta i den så kallade *runtime-miljön* som installeras tillsammans med kompilatorn. Ofta kan det finnas olika sådana miljöer och de kallas typiskt `crt`, (*c-run-time*).

applikationskod

Här placeras all övrig programkod. Den får inte vara självmodifierande, dvs sektionen förutsätts vara *read-only*. Kompilatorn gör en "bild" av maskinkod som laddas i minnet. Av tradition namnges en sektion med programkod `text`.

initierade variabler

Deklarationer som `int a = 2; char array[]={”String”};` på toppnivå kan användas för att deklarera globala variabler. Innehållet är definierat från start, men kan komma att ändras under exekvering. Kompilatorn måste göra en "bild" även av detta segment för att dessa initialvärden ska kunna laddas till minnet före exekvering. *GCC* använder regelmässigt två olika segment `data` och `rodata` (*read-only data*) för initierade globala variabler.

icke initierade variabler

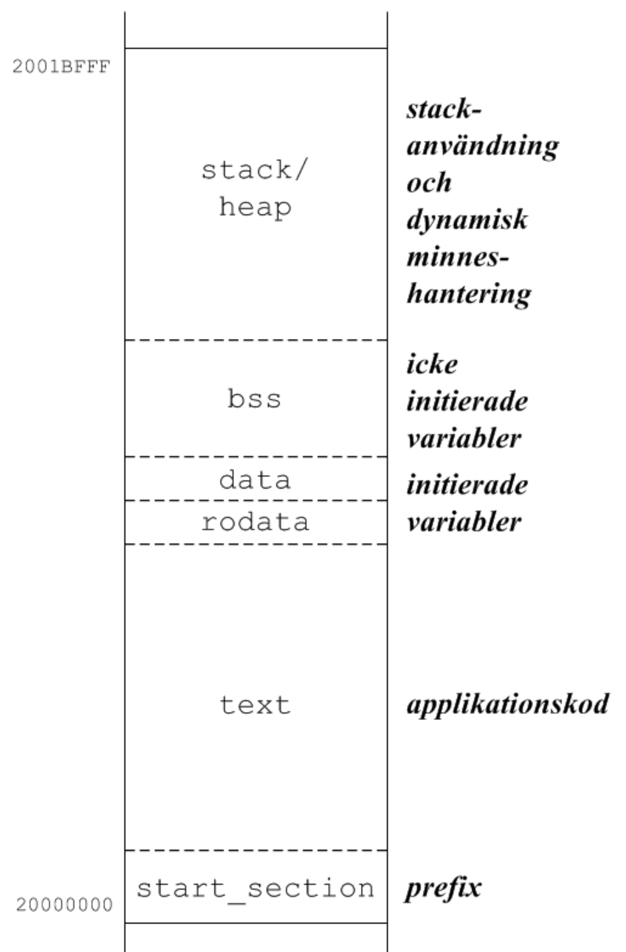
Deklarationer som: `int a; char array[34];` osv, används för att deklarera variabler utan att samtidigt ge dessa initiala värden. Eftersom variablerna inte har något definierat innehåll från start behöver kompilatorn bara hålla reda på var, i minnet dessa hamnar. Det behövs alltså ingen "bild". Oinitierade variabler placeras därför i ett särskilt segment. Av tradition kallas detta `bss`, *block started by symbol,* även namnet `COMMON` förekommer.

stack

Stacken används av program under exekvering. Minne måste avdelas för att rymma den maximala stackanvändningen.

heap

Heapen benämns ofta det minnesutrymme som avdelats för dynamisk minneshantering. Exempelvis `malloc(),free()` i standard-C biblioteket kräver ett sådant minnesutrymme.



FIGUR 3: MINNESANVÄNDNING FÖR PROGRAM UNDER EXEKVERING

Vid kompilering av en källtextfil skapas en objektmodul med följande information/innehåll:

- `text`-sektion innehållande en "bild" av programkoden.
- `data`-sektion innehållande en "bild" av initierade data som kan komma att ändras under programexekvering.
- `rodata`-sektion innehållande en ”bild” av initierade data som *inte* kan ändras under programexekvering.
- Storleken av `bss`-sektionen.
- Symboltabell innehållande alla globala symbolers relativa adresser (offset till sektionens början) i respektive sektion. Observera att alla symboler är relokerbara, dvs absoluta adresser i minnet har ännu ej bestämts.

Sammanställning av de olika segmenten för program och data som ska överföras till laborationsdatorns minne sker vid *länkningen*. Instruktionerna till länkaren kan förmedlas via en script-fil, *länkarscript*. Här använder vi ett enkelt länkarscript `md307-ram.x` som anpassats för assembler- och C-program. Vi går här inte närmre in på länkarscriptets utformning

```
/*
 Default linker script for MD307
 Yuilds three resulting segments, .text, .bss and .data, everything goes to RAM.
*/

MEMORY
{ /* MD307 RAM memory space, 0x20000000 .. 0x2001FFFF */
  RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{   /* Collect sections and merge into final segments */
     .text :
    {
        . = ALIGN(4);
        *(.start_section)   /* startup code */
        *(.text)            /* remaining code sections */
        *(.text.*)
        *(.rodata)          /* read-only data (constants) */
        *(.rodata.*)
    } >RAM

     .data :
    {
        . = ALIGN(4);
        *(.data)           /* user initialised data */
        *(.data.*)
    } >RAM

    .bss :
    {
        . = ALIGN(4);
        __bss_start__ = .;
        *(.bss)            /* zero intialised data */
        *(.bss.*)
        *(COMMON)
        . = ALIGN(4);
        __bss_end__ = .;
        . = ALIGN(4096);
        heap_low = .;
        . = . + 0x800;
        heap_top = .;
        . = . + 0x400;
        stack_top = .;
    } >RAM
}
```

Variabler

För att indikera att en symbol utgör en variabeldeklaration måste den alltså hänföras till något lämpligt segment, `.bss` för icke-initierade variabler eller `.data` för initierade variabler.

Observera speciellt att C-standarden säger att hela bss-segmentet ska initieras till noll innan programmet startas, dvs. benämningen ”icke-initierade” stämmer då inte riktigt, dock denna initiering utförs av run-time systemet. Här konstaterar vi att vi ännu inte använder något sådant men vi återkommer till det i senare moment. Våra variabler i bss-segment kommer därför att innehålla godtyckliga värden, de är än så länge inte initierade.

ÖVNING 3

Spara filen `gdbtest2.asm` under namnet `gdbtest3.asm`. Redigera programmet enligt följande:

```
# gdbtest3.asm
start:      la t1,0x40011400        # base address port D
         li t0,0x44444444       # config, port D bits 0-7...
         sw t0,0(t1)         # ...inport, floating input
         la t2,bssvar           # address of variable 'bssvar'
main:    lh t0,8(t1)         # read16 bit data from port D
         sh t0,0(t2)         # store value
         j  main             # and repeat...
   .bss                      # switch segment…
bssvar:  .space 4
   .data                     # switch segment…
datavar:.word  100
```

- Assemblera och rätta eventuella fel.

Tala nu om för GDB att du arbetar med en ny fil:

(gdb) file gdbtest3.elf

```
(gdb) file gdbtest3.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 "gdbtest3.elf"? (y or n) y
Reading symbols from gdbtest3.elf...
(gdb)
```

Ladda programmet till GDB och återstarta det:

```
(gdb) load
Loading section .text, size 0x26 lma 0x20000000
Loading section .data, size 0x4 lma 0x2000002c
Start address 0x20000000, load size 42
Transfer rate: 5 KB/sec, 21 bytes/write.
(gdb) monitor restart
User restart target
(gdb)
```

Undersök värdet i variabeln `bssvar`, observera att du måste typkonvertera denna eftersom *GDB* inte har typinformationen från assemblerfilen:

(gdb) print (int) bssvar

```
(gdb) print (int) bssvar
$12 = 17007507
(gdb)
```

Variabeln har ett odefinierat innehåll, dvs. kan vara vad som helst. Undersök också variabeln datavar:

(gdb) print (int) datavar

```
(gdb) print (int) datavar
$13 = 100
(gdb)
```

Vi ser att denna initierats korrekt. Forsätt nu med att ställa in någor värde på strömställaren, stega igenom programslingan och undersök på nytt variabeln `bssvar`.

APPENDIX: NÅGRA ANVÄNDBARA GDB-KOMMANDON

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

Studiematerial, Maskinorienterad programmering

Studiematerialet kring *MD307* består huvudsakligen av:

- Laborationsdator *MD307* (*PTB-1000*)
- IO-kort -*PTB-110*
- Display-kort *PTB-111*
- Arbetsbok *Maskinorienterad programmering RISC-V med MD307.*

Länkar till handledningar och handböcker kan nås från *ETERM8*'s hjälpmeny.

Programvaror (Windows, Linux och MacOS)

- *ETERM8*
- *SimServer*
- *CodeLite*
- *GCC för Windows (Mingw64).*
- *GCC korskompilator för RISC-V*, distribution som kompletterats för användning tillssammans med *MD307.*

Handböcker (PDF):

- *Quick Guide – MD307*
- *SimServer/IO-simulator*, användarhandbok.
- *PTB-1000* användarhandbok
- *PTB-110* användarhandbok
- *PTB-111* användarhandbok

Handledningar (PDF)

- *ETERM8 och MD307, inledande övning 1* – introducerande övning inför laborationer.
- *GDB och SimServer med MD307, inledande övning 2* – introducerande övning inför laborationer.
- *GCC och SimServer med MD307, inledande övning 3* – introducerande övning inför laborationer.
- *Laborationsuppgifter med simulator, MD307* – 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.