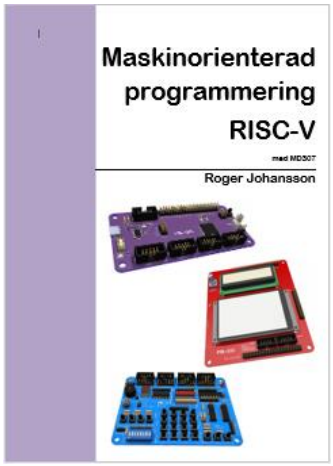


ETERM8 och *MD307*
inledande övning 1

Under denna övning kan du lära dig att använda utvecklingssystemet *ETERM8* tillsammans med simulatorn *SimServer* som ger dig en virtuell miljö för bland andra måldatorn *MD307*.

Vi behandlar den grundläggande programutvecklingsprocessen dvs. hur man redigerar en källtext i assemblerspråk, därefter översätter källtexten till ett maskinprogram, och slutligen hur programmet testas och undersöks i simulatorn.

Anmärkningar:

- Instruktioner i detta häfte förutsätter att du arbetar med en fungerande installation av *ETERM8* och *SimServer*.
- *ETERM8* har utformats för användning med såväl verktygskedjan *binutils* (GNU-projektet), som GMV's assemblatorer. För *MD307* finns dock ingen GMV-assemblator och här beskrivs därför bara användningen av *binutils för RISC-V*. Det förutsätts att du sedan tidigare är bekant med något assemblerspråk och har tillgång till lämpliga läromedel, instruktionslista etc. för RISC-V.

INNEHÅLL

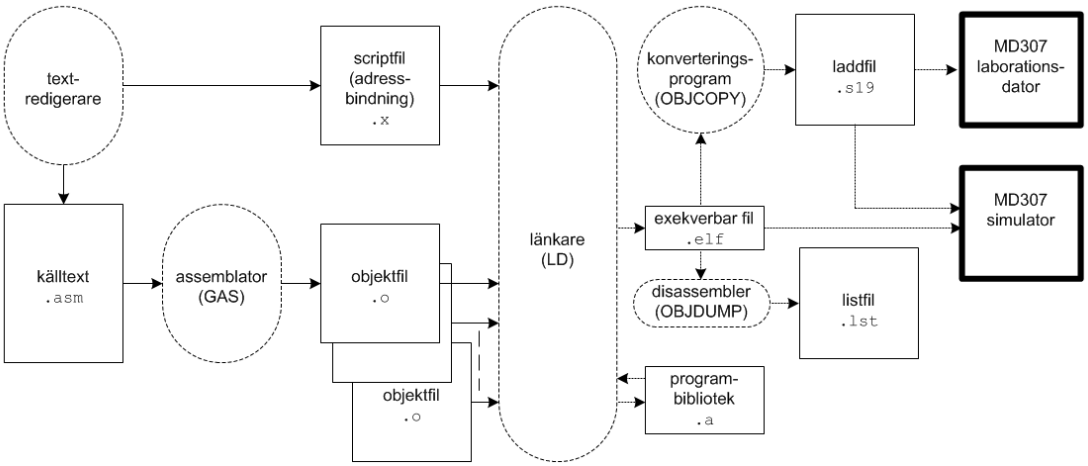
Version: 2024-12-01 17:34:00

ETERM8 och programutvecklingen

ETERM8 är avsett för programutveckling i assemblerspråk och har anpassats för undervisningsändamål. *ETERM8* omfattar funktioner för:

- *Textredigering*, källtexten skrivs/redigeras med hjälp av en *Editor*, färgad syntax används för att hjälpa dig upptäcka enklare stavningsfel.
- *Assemblering*, källtexten översätts till en laddfil som innehåller programmets maskinkod och data.
- *Test*, laddfilen överförs till den inbyggda *simulatorn* eller till laborationsdatorn *MD307* via *ETERM8*:s terminalfunktion .

Programutveckling i assembler skiljer sig inte särskilt mycket från programutveckling i något högnivåspråk. Programmet skrivs i form av källtext, dvs. en textfil som innehåller instruktioner och direktiv till assemblatorn. Då programmet, eller en lämplig del av det, är färdigt måste det översättas till maskinkod innan programmet kan testas i en måldator eller simulator. Översättningen av programmet sker i flera steg med hjälp av olika verktyg. Processen illustreras i figur 1.



FIGUR 1: ÖVERSÄTTNING AV ASSEMBLERPROGRAM TILL EXEKVERBAR KOD

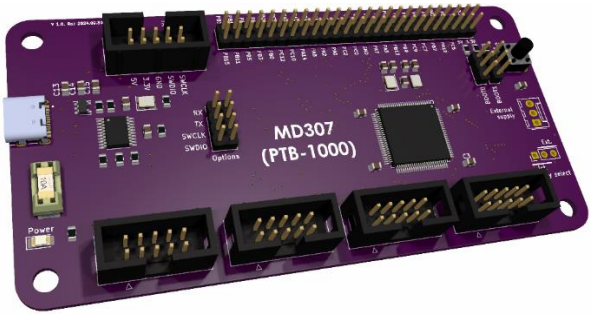
Det första steget kallas assemblering (ung. ”sätta samman”) och utförs av assemblatorn. Assemblatorn översätter programmets källtext till objektkod, till skillnad från källtextens textformat är objektkoden i binärformat.

En, eller eventuellt flera objektfiler, kombineras samman av länkaren till en exekverbar fil med *elf-format* (*executable linkable format*). Man kan också välja att låta länkaren skapa bibliotek med program som senare kan kombineras på nytt med ytterligare objektkod.

För att skapa en exekverbar fil krävs också ett så kallat *länkarskript*, en textfil där man bland annat samlat information om var maskinkoden ska placeras i måldatorns minne, var i minnet utrymme reserverats för variabler etc.

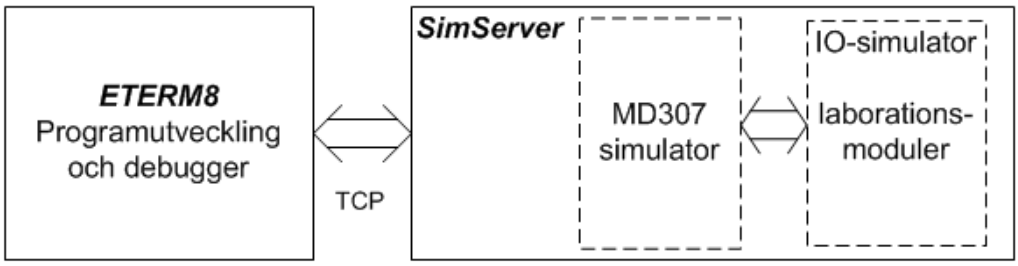
För att kunna överföra en exekverbar fil till laborationsdatorn måste den konverteras till en så kallad *laddfil*, med ett textbaserat format (*Motorola S-format*) detta görs med ett konverteringsprogram. I laddfilen finns programmet representerat på en form som kan överföras till laborationsdatorn och där tolkas som instruktioner och data.

Då programmet, i form av laddfil, överförts till laborationsdatorn *MD307*, se figur 2 nedan, kan det exekveras (utföras) och man kan då kontrollera programmets funktion men man kan också använda *ETERM8*:s simulator för test av program.



FIGUR 2: LABORATIONSDATOR MD307

Funktionerna för utveckling av assemblerprogram har integrerats i programmet *ETERM8*, medan simuleringsfunktioner utförs i ett annat program: *SimServer*. *ETERM8* och *SimServer* kommunicerar via *Transmission Control Protocol (TCP)*, se figur 3



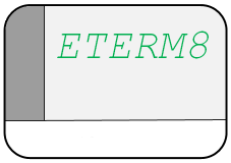
FIGUR 3: ETERM8 OCH SIMSERVER

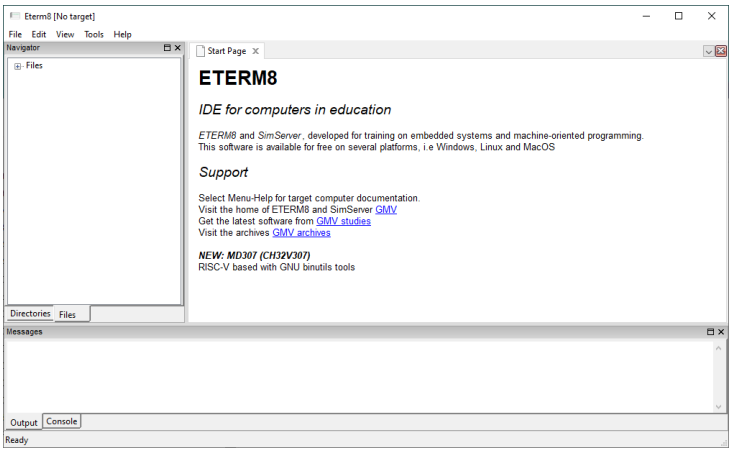
SimServer motsvarar den fysiska laborationsdatorn (bland andra *MD407*) tänk dock på att det alltid finns skillnader mellan simulatorer och hårdvara som gör att likheten inte är fullständig.

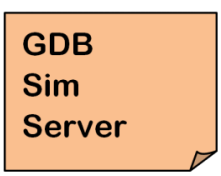
ETERM8 kommunicerar alltså med *SimServer* via *TCP* där *ETERM8* är klienten medan *SimServer* utgör serverfunktionen. Kommunikationsporten kan väljas fritt och standardvärdet är port 1234. Normalt behöver inte detta ändras men det kan vara praktiskt om man vill utföra flera samtidiga instanser av *SimServer*, dessa måste då tilldelas olika portar.

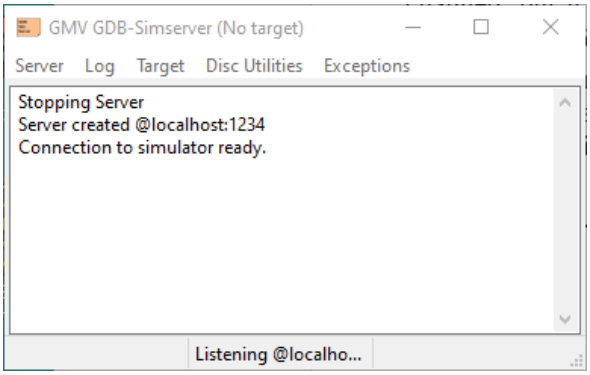
I *SimServer*, finns utöver simulatorer för laborationsdatorer också simulering av laborationsenheter. Detta gör att du kan utföra uppgifter och laborationsövningar utan att ha tillgång till den fysiska hårdvaran. Syftet med detta är att du ska kunna förbereda dig grundligt innan du börjar arbeta med laborationsutrustningarna.

Om du inte redan gjort det så starta nu *ETERM8* och *SimServer*.







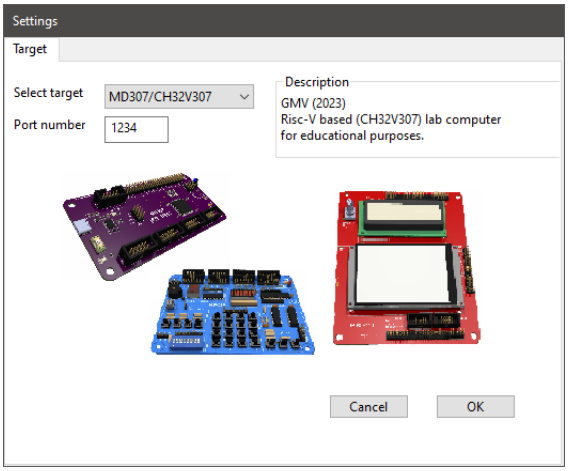


Börja med att kontrollera att *ETERM8* är inställd för rätt laborationsdator och att rätt kommunikationsport används.

Välj Tools | Settings:

I listboxen Select Target, välj MD307/CH32V307.

Kontrollera Port number (TCP-port) detta ska vara samma som i *SimServer*

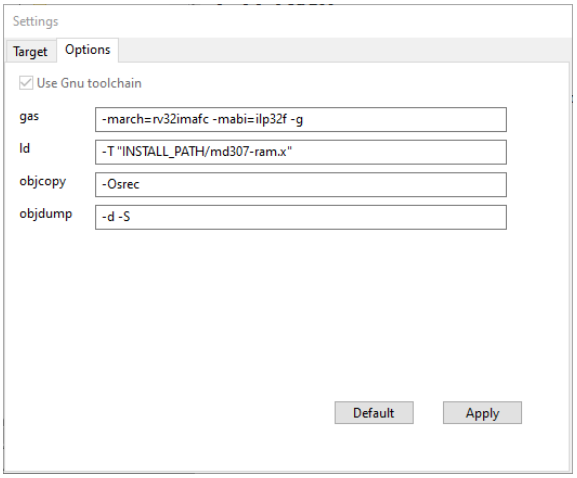


Klicka OK för att spara eventuella ändringar, eller Cancel för att återgå utan att spara ändringar.

Under fliken Options kan du ge speciella flaggor för de använda verktygen.

Texten ”INSTALL_PATH” anger här den plats där *ETERM8* och *SimServer* är installerade. Den kan variera beroende på just din installaton.

Verktygskedjan GNU/binutils används för assemblering, länkning mm.

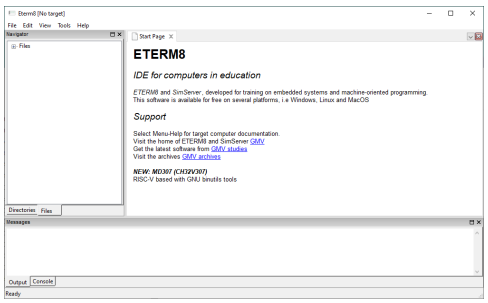


Kontrollera att *SimServer* ställts in för samma kommunikationsport, det gör du med menyvalet Server | Set Target.

Tänk också fortsättningsvis på att bilderna här kommer att kunna avvika mer eller mindre beroende på vilken plattform (Windows, MacOS eller Linux) du använder. På samma sätt kan utseendet avvika beroende på det valda temat, i operativsystemet.

ETERM8 har initialt tre olika arbetsytor med olika flikar:

- Navigator - fliken Directories används för att ange ett arbetsbibliotek, under fliken Files visas alla filer i arbetsbiblioteket.
- Messages - i fliken Output visas meddelanden från assemblatorn, fliken Console rymmer ett terminalfönster som används för att kommunicera med laborationsdatorn *MD407* via någon USB-port.
- Den tredje arbetsytan som initialt visar en startsida, används också som redigeringsfönster för textfiler.

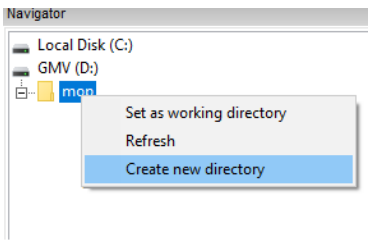


Ytterligare flikar skapas av *ETERM8*:s debugger, vi återkommer med detaljer efterhand som dessa introduceras.

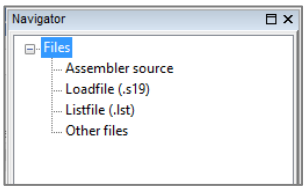
Börja nu med att skapa ett arbetsbibliotek (här D:\mop\lab1) välj själv lämplig plats och namn.

Anm. För ditt arbetsbibliotek bör du undvika sökvägar med svenska tecken eller "mellanslag" eftersom vissa programverktyg som *ETERM8* använder inte hanterar dessa korrekt.

Använd Navigator-Directories för att välja arbetsbiblioteket, välj (vänsterklicka) på arbetsbibliotekets namn, högerklicka därefter, en popup med namnet Set as working directory visas, klicka på denna.



Växla därefter till fliken Files.



Under Files sorteras filerna i arbetsbiblioteket i fyra olika grupper:

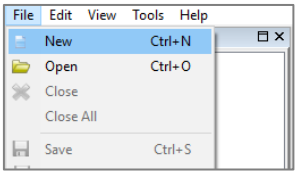
- Assembler source listar filer med ändelse som är typisk för en källtext.
- Loadfile (ändelse `.s19`), laddfil med binär kod/data som kan överföras till laborationsdator eller simulator
- Listfile (ändelse `.lst`) adressinformation tillsammans med assemblerkod, kan ibland vara användbart vid test och felsökning.

Filer som inte har någon speciell betydelse för *ETERM8* sorteras in under Other files.

Skapa ett assemblerprogram

Skapa en ny källtextfil genom att välja:

File | New från menyn.



Därefter anger du namnet på den fil du vill skapa, välj `mom1.asm` ocj klicka på Save. Om du inte anger någon filnamnsändelse (eller anger någon annan ändelse), kommer *ETERM8* att lägga `.asm` till ditt filnamn. Nu skapas ett nytt fönster.

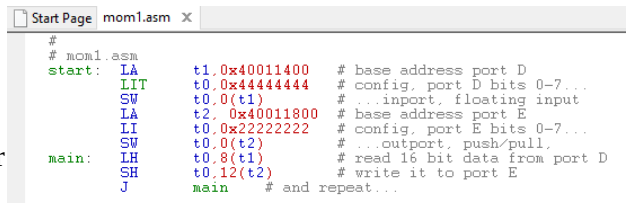
ÖVNING: TEXTREDIGERING

Skriv in följande program i det nya fönstret:

```
start:  LA    t1,0x40011400   # base address port D
        LIT   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...
```

Observera hur texteditorn färglägger din text ("färgad syntax").

- Tecknet '`#`' används för att ange att all påföljande text är kommentarer. Kommentarer ignoreras av assemblatorn, tecknet kan förekomma var som helst på en textrad och kommentarer färgas grå.
- Ett giltigt symbolnamn färgas grönt. Med "giltigt symbolnamn" menas alla kombinationer av teckensträngar där de ingående tecknen är tillåtna för en symbol. Detta innebär självfallet inte att symbolen är korrekt definierad eller refererad.
- Giltiga instruktioner, namn på RISC-V register och assemblerdirektiv färgas blå
- Konstanter och operatorer färgas röda. Teckenkombinationen '0x' (noll-x) anger att påföljande talvärde ska tolkas på hexadecimal form.



Notera speciellt hur instruktionen:

```
LIT   t0,0x44444444
```

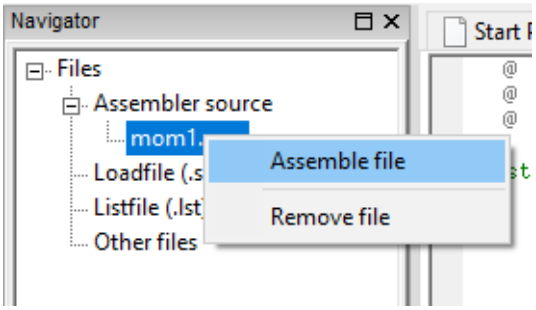
färgas grön, dvs. tolkas som en symbol. Detta beror på att vi (avsiktligt) stavat instruktionen fel, rätt instruktion ska här vara `LI`. Låt felet vara kvar, vi ska strax rätta till det. För att spara filen använder du nu File | Save.

Tänk på att korrekt ”färgad syntax” inte nödvändigtvis innebär att ditt assemblerprogram är korrekt, det är snarare till för att göra dig uppmärksam på enklare stavfel, syntaxfel etc. Därför kan det hända att du får felmeddelanden även om du stavat såväl instruktioner som operander riktigt.

Assemblering

Du assemblerar källtexten genom att:

1. Växla till fliken Files i Navigator fönstret.
2. Öppna Assembler source
3. Välj (vänsterklicka) på filnamnet `mom1.asm` högerklicka därefter, en popup knapp visas...
4. välj Assemble File



alternativt:

Från Menyn väljer du Tools|Assemble, ett dialogfönster låter dig nu välja den fil du vill assemblera.

Utskrifter från assemblatorn visas i sektionen Messages under fliken Output.

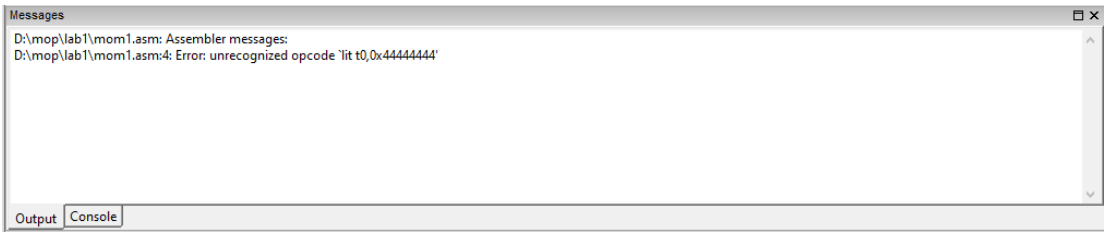
ÖVNING: ASSEMBLERING

1. Assemblera nu filen `mom1.asm`

Assemblatorn kommer att klaga på den felstavade instruktionen. Efter filnamnet, med fullständig sökväg (som kan se annorlunda ut i ditt exempel) skrivs radnummer, därefter typ av fel. Utskriften

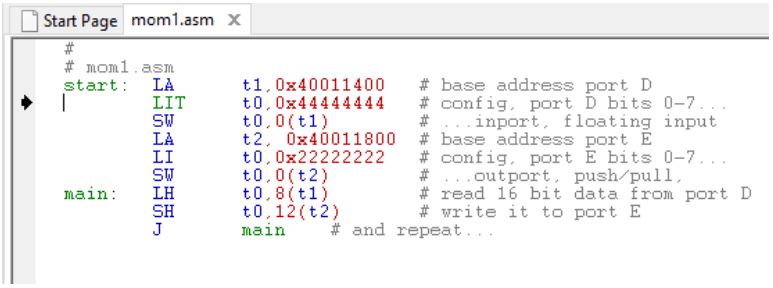
```
"Error: unrecognized opcode 'lit t0,0x44444444'"
```

berättar vad vi redan misstänkte, dvs. att instruktionens namn är felstavat.

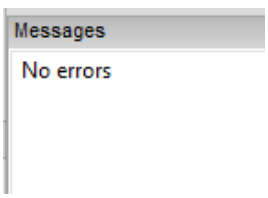


Felutskrift från assemblator

2. Dubbelklicka nu (vänster knapp) på felutskriften. Markören i marginalen pekar ut raden i källtextfilen som genererat felet.



3. Rätta felet (ändra den felaktiga instruktionen till `LI`) och assemblera på nytt. Meddelandet No Errors ska nu visas i Output-fliken.



Simulatorfunktioner

Med *ETERM8:s* debugger och *MD307*-simulator kan du simulera instruktionsutförandet i laborationsdatorn *MD307*. Du kan också övervaka och manuellt ändra såväl register som minnesinnehåll.

Simulering av programexekvering

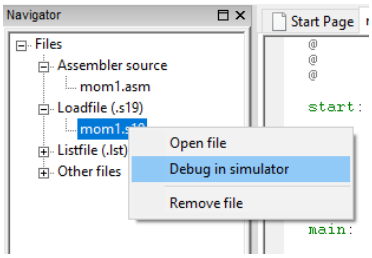
Med debuggerns hjälp kan du få en första inblick i hur assemblerinstruktioner fungerar. Du kan utföra ett assemblerprogram instruktionsvis och i lugn och ro studera effekterna. Glöm inte att spara dina källtextfiler, många moment utgår från att återanvända kod.

Under detta moment kommer vi att introducera användningen av debuggern. Först visar vi hur du kan ladda program till simulatorn och utföra programmet instruktionsvis. Du kommer också att se hur man kan koppla *kringenheter* till simulatorn.

Kontrollera först att du startat *SimServer*.

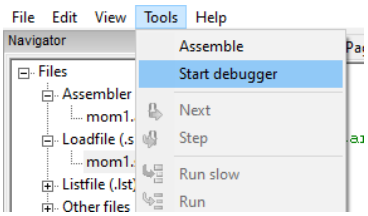
STARTA DEBUGGERN

Vi ska nu prova programmet `mom1.asm`. Debuggern kan startas på olika sätt,



1. I fliken Navigator|Files, välj gruppen Loadfile(.s19)
2. Högerklicka på mom1.s19, en popup med tre alternativ visas:
   Open file - om du vill granska laddfilen med en editor
   Debug in simulator - för att starta simulatorn med denna laddfil.
   Remove file för att ta bort filen.

alternativt:

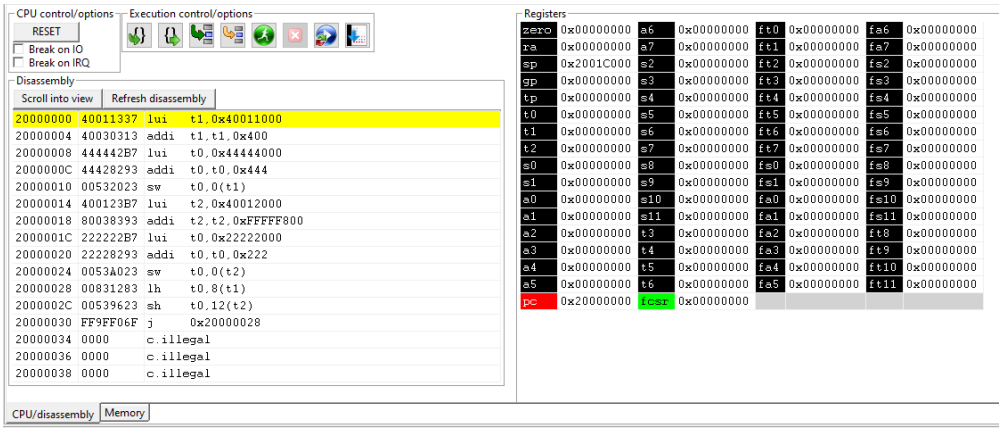


Välj från menyn
Tools |Start debugger
en dialog låter dig nu välja laddfil.

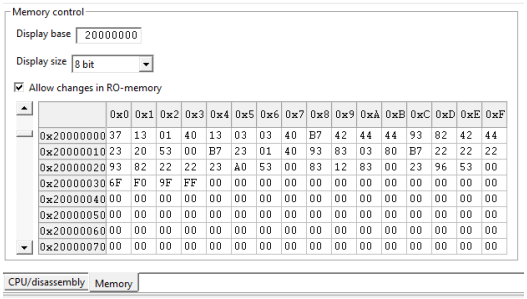
1. Välj från menyn Tools | Start debugger
2. Välj i dialogrutan: `mom1.s19`

Debuggern har två flikar:

- *CPU/disassembly*, *MD307*-simulatorns olika RISC-V-register och en bild av det disassemblerade minnesinnehållet dvs. programmet och debuggerns kontrollfunktioner.



- *Memory*: *MD307*-simulatorns minne

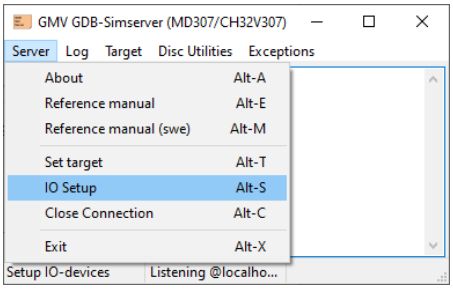


Eftersom programmet utför in- och ut- matning måste vi dock göra vissa förberedelser innan vi provar programmet.

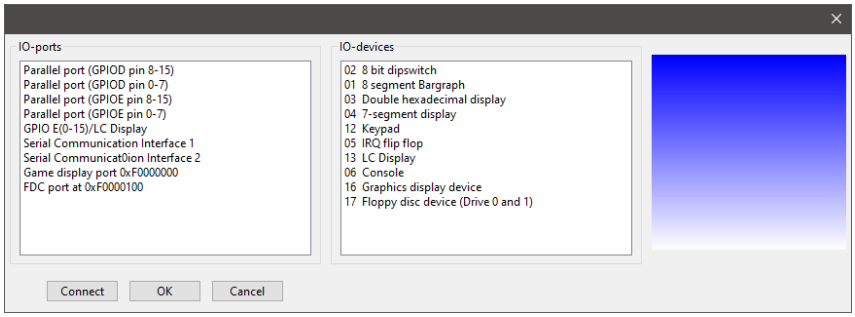
Till simulatorn finns också en fristående del som kallas *IO-simulator* (Input/Output-simulator). Dess uppgift är att simulera olika omgivningar till laborationsdatorn, dvs. de enheter som inmatning sker från och utmatning sker till. Eftersom IO-simulatorn innehåller olika typer av kringenheter och dessa kan kopplas till olika anslutningar hos *MD407* och vi måste göra dessa inställningar manuellt.

I *SimServer*,

välj Server | IO Setup:



Följande dialogruta öppnas:



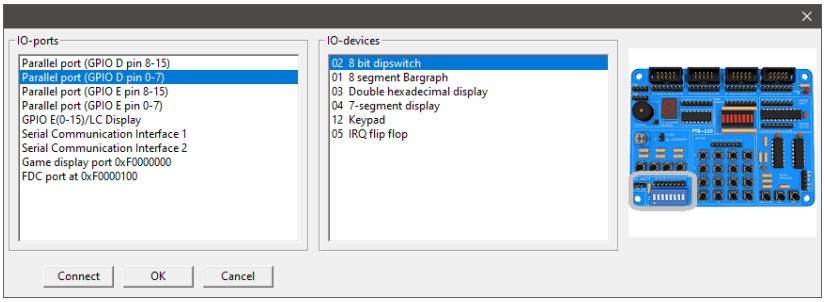
Till vänster, *IO-ports*, visas de anslutningar (*portar*) hos *MD307* som kan användas i simulatorn.

I mitten, *IO-devices*, visas de kringenheter som kan anslutas.

Då en kringenhet väljs, visas den längst till höger.

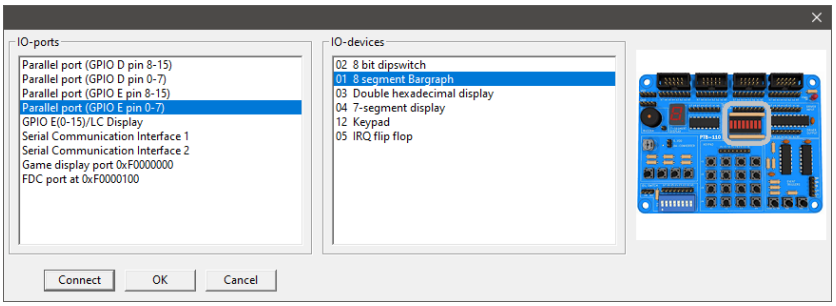
För att prova vårt första program, `mom1`, kopplar vi nu en 8 bitars strömställare (8 bit dipswitch) till den minst signifikanta *byten* av port D (GPIOD pin 0-7).

Märk porten och kringenheten:



Klicka därefter på Connect. Strömställarmodulen öppnas. Den är försedd med 8 st. enpoliga knappar som kan ställas i läge on eller off genom att man klickar på dem. Initialt står knapparna i läge off, vilket innebär att motsvarande logiknivå är 0. Genom att klicka en gång på en knapp ändras läget till on, logiknivån är då 1, klicka en gång till på knappen för att få den att återgå till off, osv. I fönstret skrivs också den aktuella anslutningen ut, i detta fall PD0-7.

Fortsätt nu med att ansluta en diodramp (8 segment Bargraph) till den minst signifikanta byten hos port E, GPIOE pin 0-7.



Diodrampsmodulen öppnas, den innehåller 8 st dioder som kopplas till porten.

Klicka slutligen OK för att behålla anslutningarna.

TEST AV PROGRAM

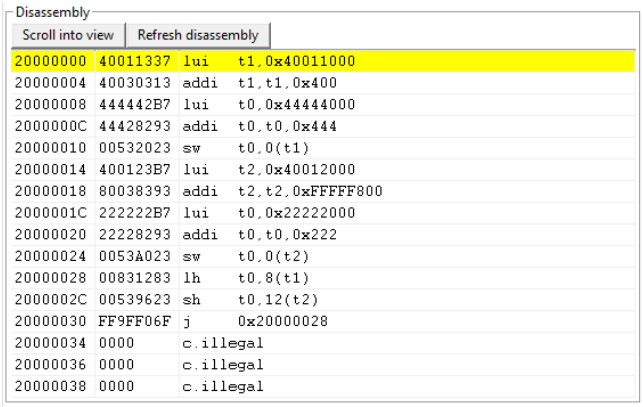
Då du anslutit enheter för såväl in- som utmatning är du redo att testa ditt program. Följande knappar används i *ETERM8* för att kontrollera programexekveringen:

	Step instruction: Utför en maskininstruktion, kallas också *trace*, exakt en maskininstruktion utförs.
	Next instruction: Utför nästa instruktion, skillnaden mot Step instruction är att en tillfällig brytpunkt placeras på nästa adress i det sekventiella programflödet. Detta innebär exempelvis att hela subrutiner kan utföras utan att man behöver stega igenom varje instruktion i subrutinen.
	Step 5 ins/sec: Stega automatiskt, 5 instruktioner per sekund
	Run, stop at breakpoint: Utför programmet, stanna vid brytpunkt.
	Run, ignore breakpoints: Utför programmet, ignorera brytpunkter

Figuren till höger visar adresser, maskinkod och maskininstruktioner i vårt program.

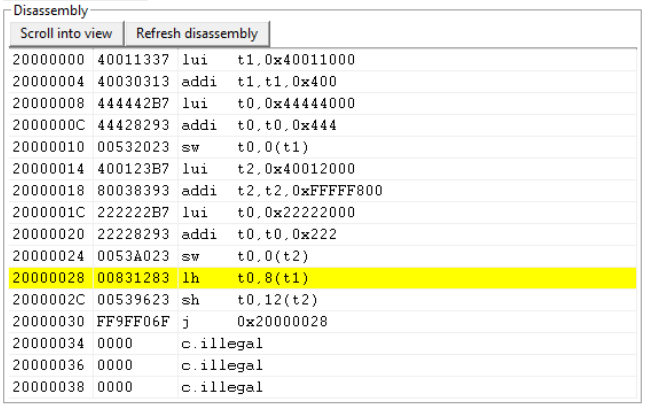
Observera nu hur debuggern har märkt den instruktion som står i tur att utföras med en gul bakgrund.

Notera också att denna instruktionssekvens ser annorlunda ut mot vår källkod, detta beror på att vi använt så kallade *pseudo-instruktioner* i källkoden. Assemblatorn har översatt dessa till de maskininstruktioner vi ser här.



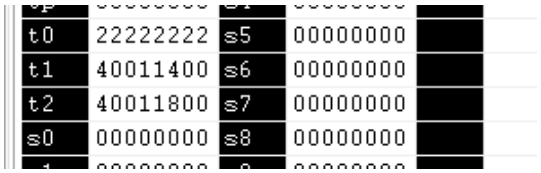
Instruktionen utförs, nästa instruktion märks med gul bakgrund. Notera effekten av instruktionen, register `t1` har fått nytt värde.

Fortsätt stega tills instruktionen på adress 20000028 märks med gul bakgrund. Notera nu hur värdena i processorns register `t1` och `t2` ändrats i register-sektionen.



Adressen till GPIOD finns nu i `t1`, adressen till GPIOE finns i `t2`.

Inporten till GPIO finns på offset 8, utporten finns på offset 12.

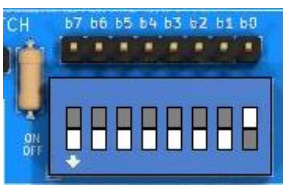


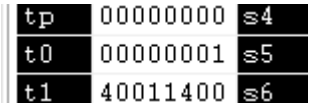
Ställ nu in värdet 1 på strömställarmodulen genom att klicka på den minst signifikanta biten.

Utför nu även nästa instruktion för att läsa ett värde från inporten:

```
lh    t0,8(t1)
```

Observera hur register `t0` laddas med strömställarmodulens värde

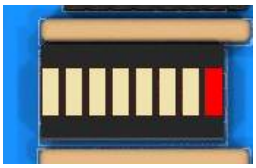




Fortsätt nu och utför nästa instruktion:

```
sh    t0,12(t2)
```

Värdet i register `t0` skrivs till ljusdiodrampen.

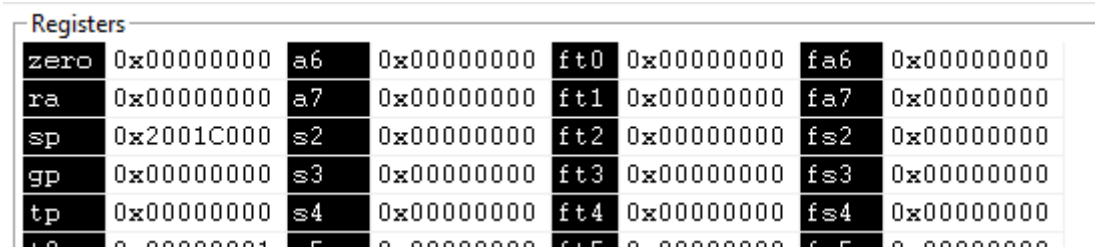


Röda indikatorer tolkas som ’1’, övriga tolkas som ’0’.

Då du testat ett programs funktion genom att utföra det instruktionsvis kan du också utföra det med något Run-kommandot.

Här finns det tre olika varianter, det första alternativet tillåter dig följa programexekveringen i detalj eftersom debuggern uppdaterar vyn mellan varje instruktion. Det båda andra alternativen används för att utföra programmet så snabbt som möjligt, endast programräknaren i registerfönstret uppdateras, det första alternativet, med en gul böjd pil, kontrollerar om det finns brytpunkter i programmet, det andra alternativet ignorerar brytpunkter.

Simulatorns registersektion visar användarregister hos RISC-V.



FIGUR 4: DEL AV SIMULATORNS REGISTERSEKTION

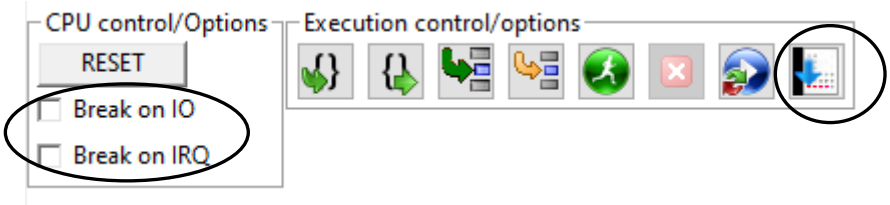
Du kan ändra registrens innehåll manuellt genom att klicka på fönstret för det register du vill ändra och skriva in ett nytt värde, värdet anges på hexadecimal form. Då du ändrat ett registerinnehåll måste du också trycka Enter, för att ändringarna ska registreras i simulatorn.

Brytpunkter

En *brytpunkt* är ett ställe där programexekveringen avbryts. Brytpunkter är bara meningsfyllda då du använder Run, stop at breakpoint. Brytpunkter kan hanteras direkt i programfönstret eller i *brytpunktstabellen* via debuggerns verktygslist. Brytpunkstabellen har plats för upp till 20 samtidiga brytpunkter i programmet.

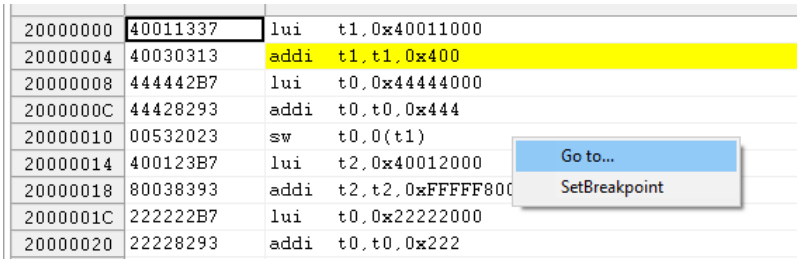
Brytpunkten kan vara *permanent* och läggs då in i brytpunktstabellen. Varje gång programmet ska utföra instruktionen på denna adress kommer simulatorn att avbryta exekveringen. Brytpunkten kan också vara *tillfällig*, programmet exekveras tills denna adress uppträder *nästa gång*. Brytpunkten tas därefter automatiskt bort av simulatorn.

Slutligen finns två speciella brytpunktsfunktioner: Break on IO, då denna aktiverats stoppas programexekveringen då programmet gör någon in- eller utmatning, respektive Break on IRQ, vilken gör att programexekveringen stoppas vid upptakten av avbrottshantering.



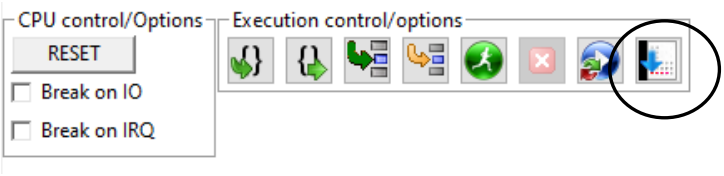
Brytpunkter kan sättas ut på olika sätt;

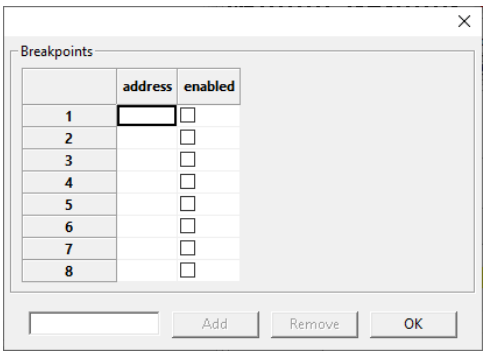
- Om programfönstret visar den adress där du vill sätta brytpunkten: identifiera adressen i programfönstret och klicka på dess rad. En *popup* meny visas då.
  - Go to.. sätter en tillfällig brytpunkt och startar programexekveringen
  - SetBreakpoint sätter en permanent brytpunkt här.



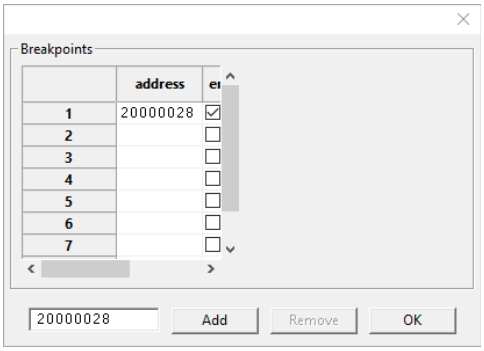
För att stoppa programmet på adresser som inte syns i programfönstret kan du identifiera dessa exempelvis genom att granska listfilen från ditt program.

- För att hantera brytpunkterna öppnar du dialogen för brytpunktstabellen:

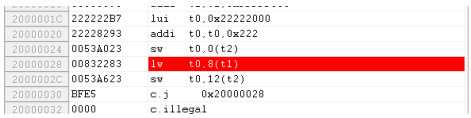




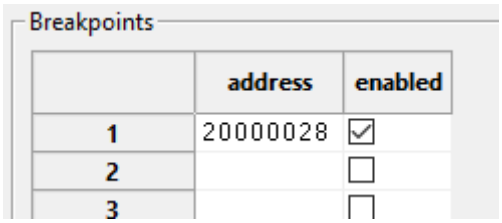
- För att lägga till en ny brytpunkt skriver du in adressen i fönstret längst ned till vänster, exempelvis adress 20000028, och klickar på Add



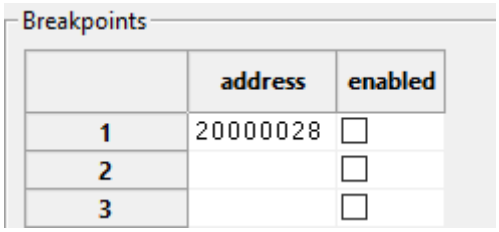
- Programfönstret markerar adresser med brytpunkt med röd bakgrund.



- Brytpunkter kan vara *aktiva* (enabled) eller *inaktiva*. Programmet stoppas inte vid en inaktiv brytpunkt men det kan vara en praktisk funktion då man tillfälligt vill undvika brytpunkten men samtidigt behålla den i brytpunktstabellen.

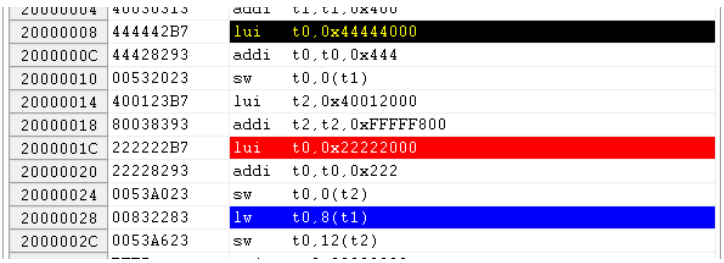


- Du kan tillfälligt inaktivera en brytpunkt genom att märka av enabled-boxen.

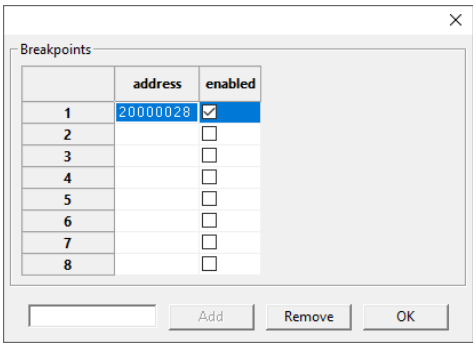


I programfönstret märks raderna enligt:

- Om den instruktion som står i tur att utföras har en brytpunkt visas denna med gul text mot svart bakgrund.
- En aktiv brytpunkt illustreras av röd bakgrund.
- En inaktiv brytpunkt illustreras av blå bakgrund.



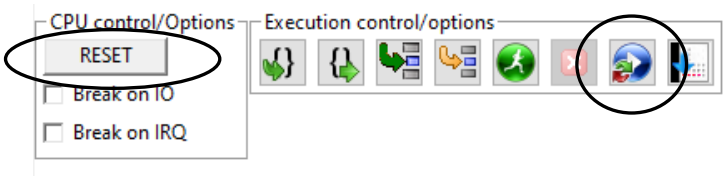
- För att ta bort en synlig brytpunkt klickar du på dess rad och väljer Remove Breakpoint.
- För att ta bort en brytpunkt via brytpunktstabellen klickar du först på dess nummer i tabellen och därefter Remove.



Återstart av program

Programmet kan förberedas för start på två olika sätt

- Restart, här förbereds programmet på samma sätt som om det hade laddats till *MD307*, dvs. hårdvaran (portar och klockor) har initierats och är klar att användas.
- RESET, här genomför simulatorn samma procedur som vid *reset* av *MD307*, dvs. portar och klockor försätts i initialtillstånd, programräknare och stackpekare initieras från minnet, därefter kan programmet startas.



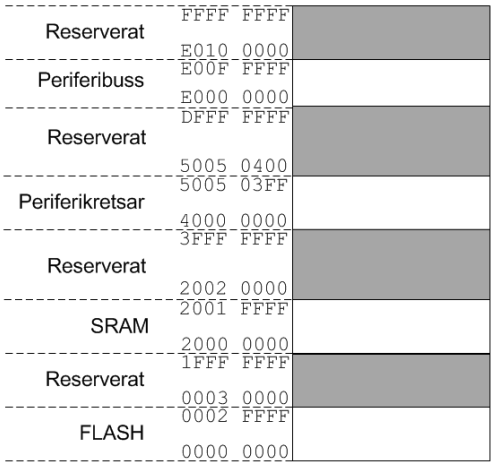
Simulatorns hantering av minnet

Användningen av adressrummet i en *MD307* framgår av figuren till höger.

Adressrymden hos *MD307*-simulatorn används på följande sätt.

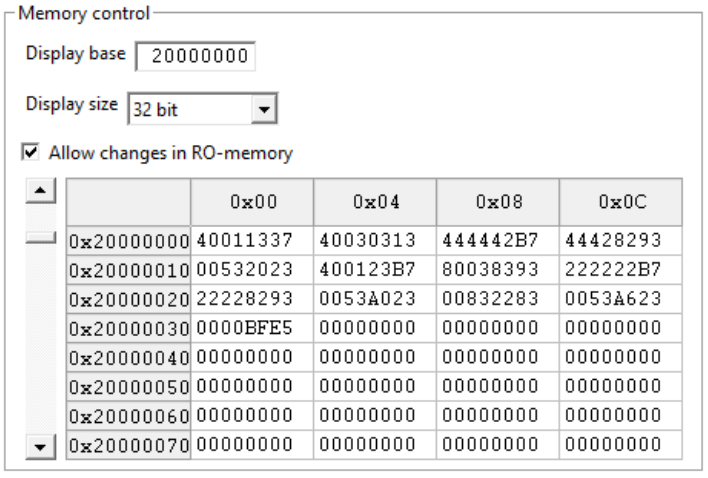
`0x0000 0000 - 0x0002 FFFF`	192 kB Flash
`0x2000 0000 - 0x2001 FFFF`	128 kB SRAM
`0x4000 0000 - 0x5005 03FF`	Periferikretsar
`0xE000 0000 - 0xE00F FFFF`	Systemregister

Adresser utanför dessa block används ej.

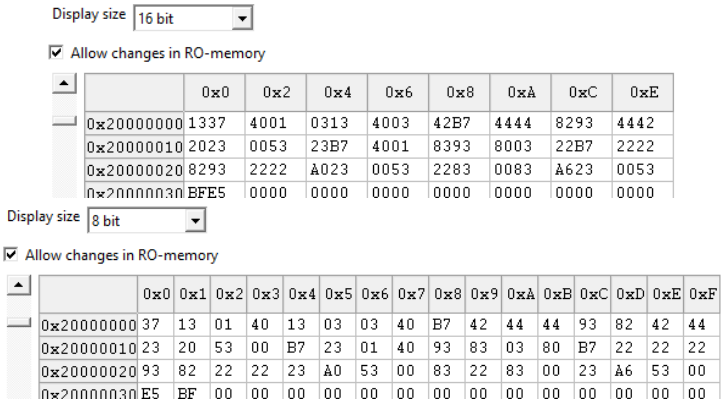


Under fliken Memory kan minnesinnehållet studeras och i vissa fall ändras. Minnesinnehåll visas i block om 128 bytes.

- Du kan ändra visningsintervallet med hjälp av rullningslisten till vänster. Som alternativ kan du ange en startadress för blocket som ska visas (Display base). Eftar att ha skrivit in en ny adress måste du trycka ned Enter-tangenten för att ändringarna ska få effekt.
- Allow changes in RO-memory beskrivs längre ned.



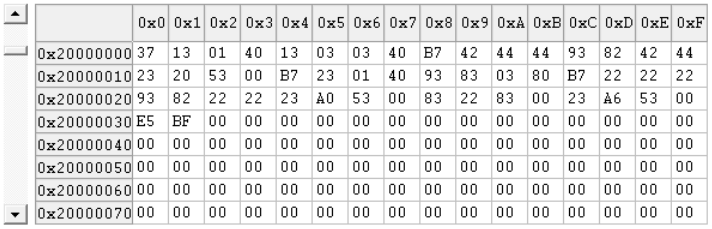
- Du kan välja (Display size) om du vill att minnet ska visas som word (32 bitar), halfword (16 bitar) eller byte (8 bitar).



READ WRITE MEMORY (SRAM)

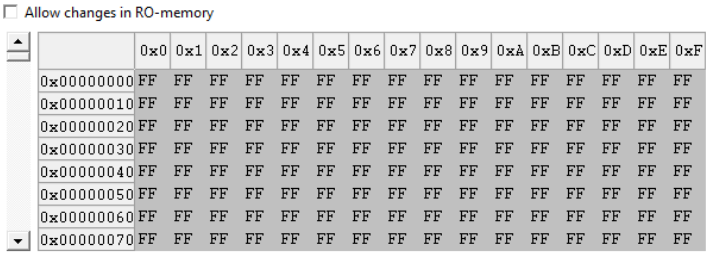
Figuren visar adressutrymmet 2000 0000_{16}-2000 $007F_{16}$, detta minne är av SRAM-typ, dvs. läs och skrivbart minne.

Innehållet i detta minne kan ändras genom att skriva in ett nytt (hexadecimalt) värde. Ändringsbara minnesinnehåll har en *vit* bakgrund.

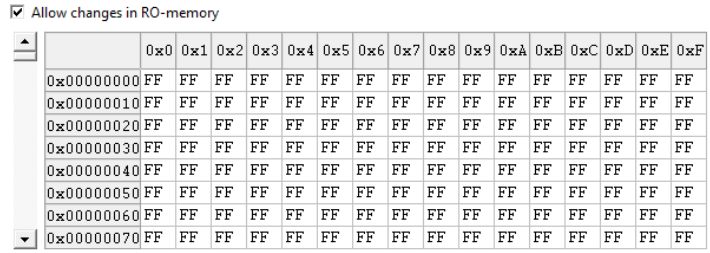


READ ONLY MEMORY (FLASH)

En annan typ är de minnesareor som är reserverade som ROM-typ (*endast läsbart minne*). För denna kategorin minne har fönstren för minnesinnehåll *grå* bakgrund. Innehållet kan då inte ändras.

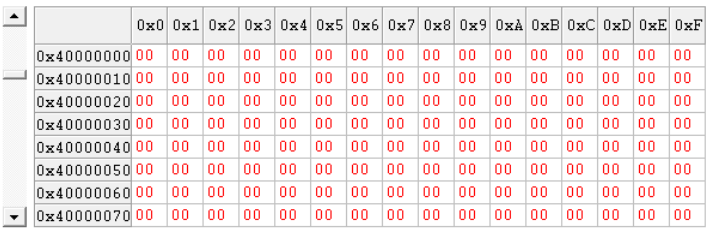


I ett verkligt system kan sådana minnen normalt sett inte skrivas, (en speciell programmeringsrutin krävs) men i simulatorn vill vi i bland kunna initiera detta minne. Av denna anledning finns knappen Allow changes in RO-memory. Då den är aktiv behandlas ROM som skrivbart minne.



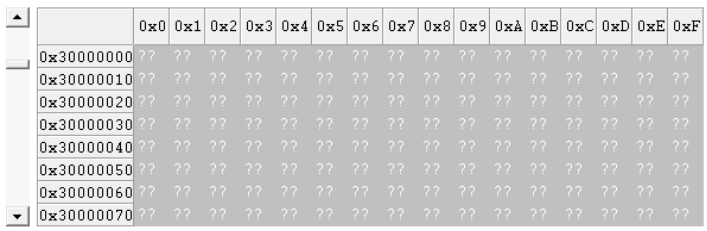
PERIFERIKRETSAR OCH SYSTEMREGISTER

Adressrymder med periferikretsar och systemregister visas med röd färg mot vit bakgrund. Observera att bara en mycket liten del av dessa adressrymder används.
Sådana block är inte skrivbara från *ETERM8*.



ICKE ANVÄNT MINNE

Den sista kategorin i adressrymden är helt enkelt odefinierade block. I *ETERM8* illustreras dessa med frågetecken mot grå bakgrund.



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.