Laborationsmiljön Darma
v0.86

Kent Palmkvist

Februari 2024

Contents

1 Hårdvaran Darma **1**
1.1 Introduktion 1
1.2 Kom igång med hårdvaran 1
1.3 Tillgängliga IO-enheter 2
1.3.1 Tryckknappar 2
1.3.2 Flerfärgad LED 2
1.3.3 Expansionskontakt 3
1.3.4 Seriell kommunikation 3
1.4 Emulerade IO-enheter 3
1.4.1 Laboration 1, tangentbord och flerfärgad LED 3
1.4.2 Laboration 2, tryckknappar och 8 LED 4
1.4.3 Laboration 3, tidbas och 7-segment display 4

2 Mikrokontrollern TM4C123G **7**
2.1 Minneskarta 7
2.2 IO-enheter 7
2.2.1 GPIO 7
2.2.2 Serieport 9
2.2.3 NVIC avbrottshanterare 10

3 Code Composer Studio **11**
3.1 workspace och project 11
3.2 Starta Code Composer Studio 11
3.3 Skapa ett nytt projekt 12
3.4 Assemblera kod 12
3.5 Starta assemblerprogrammet 13
3.6 Undersöka minnesinnehåll 13
3.6.1 Format på disassembly lista 13
3.7 Stega igenom ett program 14
3.8 Undersöka och ändra interna register 15
3.9 Undersöka I/O-enheternas registervärden 15

3.10 Seriell kommunikation 15

4 Processorn ARM Cortex-M4F **19**
4.1 Register 19
4.2 Instruktionsuppsättning 20
4.2.1 Notation 20
4.2.2 Adresseringsmoder 21
4.2.3 Flyttinstruktioner 22
4.2.4 Beräkningsinstruktioner 23
4.2.5 Hoppinstruktioner 25
4.3 Avbrottshantering 26
4.3.1 Avbrottsvektorer 26
4.3.2 Inställning för val av avbrott 27
4.3.3 Händelser vid avbrott 27
4.3.4 Återhopp från avbrott 28
4.3.5 Stanna processor i väntan på avbrott 28
4.4 Assembler pseudoinstruktioner 28
4.5 Fullständig instruktionslista 29

5 Vanliga frågor **35**

Chapter 1

Hårdvaran Darma

Detta kapitel introducerar laborationshårdvaran Darma.

1.1 Introduktion

Laborationsmaterialet kallat Darma är anpassad för att kunna användas tillsammans med 5V-baserad TTL-kompatibel hårdvara, såsom tangentbord, lysdioddrivare etc. Som extra skydd sitter det resistanser till varje anslutningsstift. Varje anslutningsstift har dessutom en lysdiod ansluten som indikerar om värdet är 0 eller 1.

Huvudkomponenten på kortet är en modul kallad TM4C123G LaunchPad evaluation kit. Denna modul består av både en mikrokontrollerkrets (TM4C123GH6PM) som innehåller processor (ARM Cortex-M4F), minne (256KB flash, 32KB RAM) och I/O-enheter (GPIO, UART, etc.). Dessutom finns programmerings och felsökningsstöd, ett par tryckknappar samt en flerfärgs lysdiod på denna modul.

Programmerings och felsökningsstödet ansluts via USB till den dator som används för programmering och felsökning. Det finns även stöd för en seriell kommunikation till den anslutna datorn via denna USB-anslutning.

Strömförsörjning sker via en separat strömmatningsenhet. Om LaunchPad-kortet används separat kan strömförsörjningen istället ske via den anslutna USB-kabeln.

1.2 Kom igång med hårdvaran

Anslut USB-kabeln från PCn till micro-USB kontakten markerad debug (placerad längst upp på det röda kretskortet).

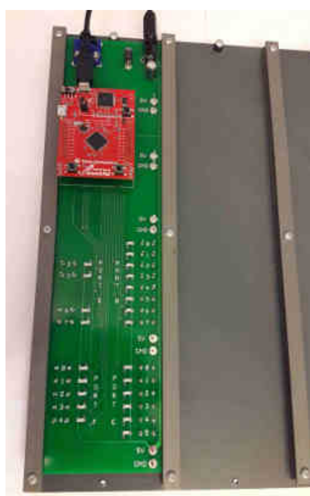

Figure 1.1: Darma-kortet monterat i labbplatta

När spänningen slås på kommer den gröna lysdioden placerad nedanför USB-kabeln att lysa.

1.3 Tillgängliga IO-enheter

Mikrokontrollern ansluts till externa enheter via portarna A-F. Varje port består av 8 bitar, men inte alla är anslutna i DARMA. En del portar har också redan anslutna signaler direkt på LaunchPad-kortet.

1.3.1 Tryckknappar

De två tryckknapparna placerade längst ned på det röda LaunchPad-kortet är anslutna till port F, med vänster knapp ansluten till bit 4, och höger knapp ansluten till bit 0.

Tryckknapparna är aktivt låga, vilket betyder att dom ger värdet 1 när man inte trycker på knappen, och ger värden 0 när knappen är tryckt.

1.3.2 Flerfärgad LED

En 3-färgers LED sitter på LaunchPad-kortet. Den kontrolleras via 3 bitar på port F. Bit 1 i port F styr röd färg, bit 2 styr blå färg, och bit 3 styr grön färg. En nolla på en bit stänger av färgen, och en etta på en bit aktiverar motsvarande färg.

1.3.3 Expansionskontakt

Till många av portarna finns anslutningspinnar monterade på Darma-kortet. Även om varje port är 8 bitar stor, så har inte alla bitar gjorts tillgängliga på Darma-kortet. Varje tillgänglig bit kan konfigureras som ingång eller utgång.

1.3.4 Seriell kommunikation

Inuti mikrokontrollern finns en seriell IO-enhet. Denna kan skicka och ta emot data från den anslutna PC:n via USB-kabeln.

1.4 Emulerade IO-enheter

För att kunna köra utrustningen på distans så ersätts de fysiska enheterna med en emulator som ansluts till Darma-systemet och som styrs med ett program på datorn.

1.4.1 Laboration 1, tangentbord och flerfärgad LED

Programmet tsea28lab1 motsvarar ett 16-knappars tangentbord tillsammans med avläsning av den flerfärgade LED. Programmet startas efter att modulen courses/TSEA28 laddats.

```
module load courses/TSEA28
tsea28lab1 &
```

Övre halvan av programmet motsvarar tangentbordet, där bitmönstret för knapparna som trycks läggs ut på bitarna 3-0 i port E, och bit 4 i port E anger om knappen trycks ned eller inte. För att kunna hålla nere en knapp samtidigt som man använder code composer studio kan ”Sticky buttons” aktiveras. Om ”Sticky buttons” är vald kommer varje tryckknapp vara av toggeltyp, dvs knappen fastnar i nedtryckt läge när den trycks, och släpper om man tycker på knappen en gång till.

I rutan Port E visas värdet på pinnarna 0 till och med 4, där en gul cirkel visar att värdet är en etta, och en svart cirkel visar att värdet är en nolla. På motsvarande sätt visar rutan Port F värdet på pinnarna 0 till och med 3. Slutligen visar den LED färgen på den flerfärgade LED som sitter på LaunchPad-kortet (som styrs av Port F bit 1-3).

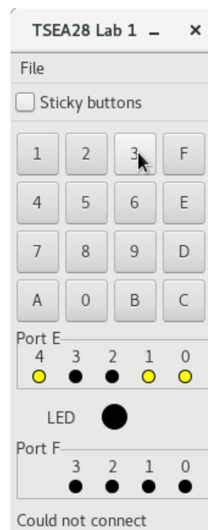

Figure 1.2: Programmet tsea28lab1 med knapp 3 nedtryckt

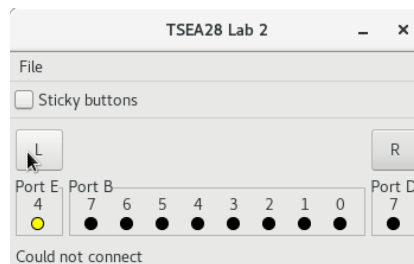

Figure 1.3: Programmet tsea28lab2 med knapp L nedtryckt

1.4.2 Laboration 2, tryckknappar och 8 LED

Programmet tsea28lab2 motsvarar två tryckknappar anslutna till pin 7 port D och pin 4 port E, samt 8 stycken LED anslutna till port B.

Genom att aktivera ”Sticky buttons” behövs ett extra tryck på knappen för att en nedtryckt knapp ska släppa.

1.4.3 Laboration 3, tidbas och 7-segment display

Programmet tsea28lab3 motsvarar en tidbasgenerator och en 4-siffror 7-segment multiplexad display. Port B väljer segment som ska lysa, där bit 0 styr segment a och segment g styrs av bit 6. Bit 7 hos port B styr decimalpunkten. Bara en siffra kan lysa åt gången, vilket styrs av bit 1 och bit 0 hos port F.

Klockpulser genereras genom att trycka på en av knapparna. Den vänstra

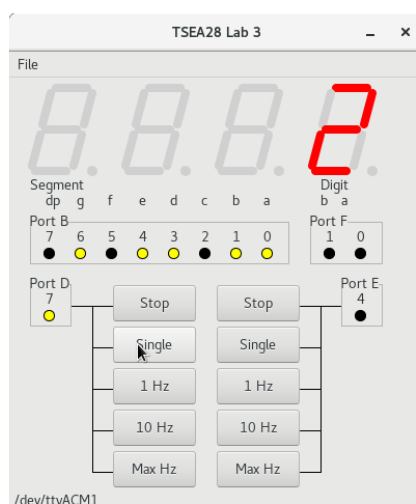

Figure 1.4: Programmet tsea28lab3 med knapp Single nedtryckt

kolumnen styr bit 7 på port D, och den högra kolumnen styr bit 4 på port E. Den senast tryckta knappen är den funktion som fortsätter vara aktiv tills någon annan knapp trycks.

Knappen 1 Hz ger en kontinuerlig fyrkantsvåg med frekvens 1 Hz. På motsvarande sätt fås 10 Hz fyrkantvåg om 10 Hz knappen trycks,

Knappen Max Hz försöker generera så snabb klockfrekvens som möjligt. Denna frekvens begränsas både av hastigheten hos datorn, och av hastighet/fördröjning hos nätverket som använts för att logga in på datorn.

Om knappen Stop trycks sätts signalen på porten till 0. Knappen Single ger en 1:a så länge som den är intryckt.

Tips: Den snabbaste och bästa kvaliteten på körningen fås om programmet körs direkt på den maskin som har hårdvaran ansluten. Detta är dock inte möjligt vid undervisning i distansläge. Näst bästa kvalitet fås om thinlinc används. Utseendet blir bli lite ojämnt och flimrigt, men tillräckligt bra för att det ska synas tydlig skillnad mellan 10 Hz och Max Hz knapparna. Sämst kvalitet fås vid inloggning via vpn och ssh.

Chapter 2

Mikrokontrollern TM4C123G

Detta kapitel ger en översikt över nödvändiga delar av mikrokontrollern TM4C123GH6PM. För de intresserade finns en fullständig beskrivning i Tiva C Series TM4C123GH6PM Microcontroller Data Sheet.

2.1 Minneskarta

Adressrymden i mikrokontrollern visas i figur 2.1. Längst ned i adressrymden (på låga adresser) finns programminnet, som består av 256 KB FLASH-minne. Som arbetsminne för stack, variabler etc. finns 32 KB SRAM.

Det finns även andra enheter och funktioner i minnesarean, men dessa beskrivs inte här.

2.2 IO-enheter

Mikrokontrollern innehåller många olika IO-enheter. Varje IO-enhet kan konfigureras på många olika sätt, vilket sker via konfigurationsregister. Vilka enheter som ska anslutas till portarna bestäms via ett antal konfigurationsregister.

2.2.1 GPIO

Den enklaste typen av IOenhet är GPIO (General Purpose IO) vilken tillåter att värdet hos varje bit på porten kan läsas och skrivas. Varje bit kan även ställas in som ingång eller utgång, och med hjälp av pull-up eller pull-down kan man välja vad en oansluten ingång ska ha för värde.

Varje port har en uppsättning register. De vanligast använda är dataregistret GPIODATA samt datariktningsregistret GPIODIR. I GPIODIR väljer

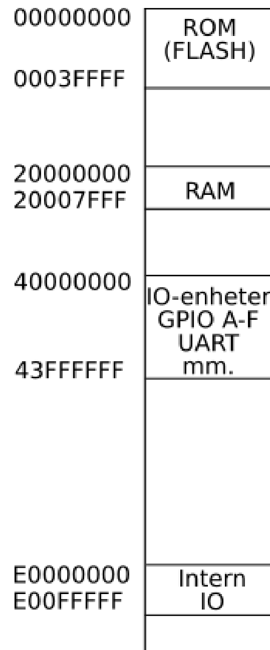

Figure 2.1: Minneskarta för TM4C123GH6PM

en nolla att motsvarande pinne på porten ska vara en ingång, och en etta sätter motsvarande pinne till utgång.

Dataregistrens adress bestämmer vilka bitar i registret som kan skrivas. Adresserna i tabellen har valts så endast de bitar som finns tillgängliga som kopplingspunkter på Darma-kortet kan ändras.

Port	Register	Address	Funktion
B	GPIOB_GPIODATA	0x400053fc	dataregister port B
	GPIOB_GPIODIR	0x40005400	riktningsregister port B
D	GPIOD_GPIODATA	0x40007330	dataregister port D
	GPIOD_GPIODIR	0x40007400	riktningsregister port D
	GPIOD_GPIOICR	0x4000741C	rensa avbrottsbegäran port D
E	GPIOE_GPIODATA	0x400240fc	dataregister port E
	GPIOE_GPIODIR	0x40024400	riktningsregister port E
	GPIOE_GPIOICR	0x4002441C	rensa avbrottsbegäran port E
F	GPIOF_GPIODATA	0x4002507c	dataregister port F
	GPIOF_GPIODIR	0x40025400	riktningsregister port F
	GPIOF_GPIOICR	0x4002541C	rensa avbrottsbegäran port F

Avbrottshantering i GPIO-port

För GPIO-portarna kan för varje enskild pinne bestämmas om och hur ett avbrott kan genereras. Följande kombinationer finns

Funktion	GPIOIS	GPIOIBE	GPIOIEV
Avbrott när pinnens värde är 0	1	X	0
Avbrott när pinnens värde är 1	1	X	1
Avbrott när pinnens värde går från 0 till 1	0	0	1
Avbrott när pinnens värde går från 1 till 0	0	0	0
Avbrott när pinnens värde går från 0 till 1 eller från 1 till 0	0	1	X

För de tre fall där en förändring av en pinnes värde ska ge avbrott kommer denna begäran vara aktiv tills en nollställning görs genom att skriva till GPIOICR registret. För att nollställa en pinnes avbrottsbegäran ska då en 1:a skrivas till motsvarande bitposition i registret.

GPIO-porten kan även styras så att avbrott förhindras från att skickas vidare till processorn (via NVIC-enheten). Detta görs genom att skriva en nolla i GPIOIM på motsvarande bit. Det går även att kontrollera om avbrottsbegäran är på väg från en pinne genom att titta på registren GPIORIS och GPIOMIS som anger om en avbrottsbegäran skickas respektive om en avbrottsbegäran skickas och är tillåten att skickas vidare till NVIC-enheten.

2.2.2 Serieport

Serieporten (UART0) kommunicerar med skrivbordsdatorn via seriell kommunikation. Tecken som ska skickas skrivs till dataregistret UART0_UARTDR och mottagna tecken läses ur samma register. Statusregistret UART0_UARTFR anger om det finns mottaget tecken som inte lästs ännu, och om det går att skicka nästa tecken.

Port	Register	Address	Funktion
UART0	UART0_UARTDR	0x4000c000	dataregister för uart 0
	UART0_UARTFR	0x4000c018	statusregister för uart 0

Registret UART0_UARTFR kan endast läsas. Bitarna i registret har följande funktioner:

Bit	Namn	Funktion
7	TXFE	0: tecken håller på att skickas 1: inga tecken håller på att skickas
6	RXFF	0: mottagaren kan ta emot fler tecken 1: mottagaren kan inte ta emot fler tecken (nya tecken kommer förloras)
5	TXFF	0: Kan ta emot fler tecken att skicka 1: sändaren kan inte ta emot fler tecken (nya tecken kommer förloras)
4	RXFE	0: Finns mottagna tecken 1: Finns inga mottagna tecken att hämta
3	BUSY	0: Inga tecken håller på att skickas 1: Tecken håller på att sändas

2.2.3 NVIC avbrottshanterare

I nära anslutning till processorn finns Nested Vectored Interrupt Controller (NVIC) som hanterar de avbrottsbegäran som de olika I/O-enheterna kan generera. I denna enhet kan prioritet för olika I/O-enheters avbrott definieras.

NVIC-enheten tar emot alla avbrott, och beroende på respektive avbrotts prioritet kan nuvarande huvudprogram eller avbrottsrutin avbrytas för att hantera ett avbrott med högre prioritet.

NVIC-enheten skickar vidare information till processorn vilken avbrottsvektor som ska köras, och en signal som anger om ett avbrott ska ske.

Chapter 3

Code Composer Studio

Programmeringen av Darma görs i utvecklingsmiljön Code Composer Studio. Detta är en komplett miljö med editor, kompilator och assemblator, samt simulering och avslusningsverktyg.

3.1 workspace och project

Alla filer och inställning samlas i en workspace. Placeringen av denna mapp väljs när Code Composer Studio startas. Använd den föreslagna platsen i hemkatalogen (t ex /edu/abcde123/workspace-v12).

Inuti workspace skapas projekt som är en samling av källkodsfiler, scriptfiler och annat som systemet behöver för att kunna assemblera och programmera Darmakortet. Varje enskild programmeringsuppgift bör placeras i ett eget project.

3.2 Starta Code Composer Studio

I denna handledning antas att linux (RHEL8) används. Programvaran finns även för Windows och MacOS, men då kan vissa skillnader finnas i hur program startas och används. För att få tillgång till programvaran i linux behöver först modulen laddas.

```
module load courses/TSEA28
```

Om programvaran används på distans och ska använda ett Darma-system som är anslutet till denna dator behöver även en kontroll göras så ingen annan redan använder programmet. Programmet tsea28active skriver ut eventuella inloggningar som använder ccstudio.

```
tsea28active
```

Om ingen annan använder ccstudio på maskinen kan programmet startas med:

```
ccstudio &
```

Programmet frågar efter en workspace, vilket är den mapp i vilken assemblerprogrammen placeras. Välj ett lämpligt namn (t ex workspace_v12) som ska ligga i ditt hemkonto (dvs /home/ditt-liuid/workspace_v12).

Om du får en fråga om uppdateringar ska installeras väljer du att bara stänga detta.

När programmet startat stänger du fliken ”Getting Started”.

3.3 Skapa ett nytt projekt

För att skapa ett körbart program för TM4C123GH6PM (mikrokontrollern) behöver man först skapa ett projekt.

Välj Project->New CCS Project... Nu ska rätt hårdvara väljas. Sätt Target till Tiva C Series och Tiva TM4C123GH6PM. Välj Stellaris In-Circuit Debug Interface som Connection. Välj Empty Project under project templates and examples. Ange slutligen ett namn på projektet.

Redan befintliga filer (t ex mallfiler för labbar) kan läggas till projektet med menyvalet Project->Add files. Se till att projektets namn är markerat. Välj den mallfil som hör till uppgiften du ska lösa, t ex lab1.asm för 1:a labbet vilken hämtas från laborationshemsidan. Ange att filen ska kopieras.

3.4 Assemblera kod

Enklaste sättet att assemblera koden är att använda Project->Build All.

I Console-fönstret nere i mitten visas de assemblerings och länkningssteg som görs, och eventuella felmeddelanden visas där. I editeringsfönstret visas även en röd markering på de platser där fel hittats.

Notera att länksteget utförs även om assemblering genererat varningar. Programmet kan därför ofta köras, men inte utföra förväntad funktion. Kontrollera därför alltid om varningar genererats i assembleringssteget.

3.5 Starta assemblerprogrammet

För att ladda ned det assemblerade programmet till Darma används Run->debug. Efter nedladdning körs automatiskt en initieringsrutin av processorn och stannar när den når 1:a instruktionen i main.

Det går nu att undersöka registervärden, minnesinnehåll och lägga till brytpunkter. Alla register i TM4C123GH6PM finns i fönstret uppe till höger i fliken "Registers". Processorns interna register (R0-R15 etc.) finns där under Core Registers.

Även I/O-enheternas register finns tillgängliga. Kom dock ihåg att det bara går att läsa en I/O-enhets register om den startats, dvs att initieringsrutinen körts.

Välj sedan Run->Resume för att köra programmet.

För att tillfälligt stanna programmet används Run->Suspend. Processorn stannar då och aktuell position i programmet visas.

För att avsluta programmet och återgå till editering används Run -> terminate.

3.6 Undersöka minnesinnehåll

Minnesinnehåll kan undersökas och ändras i fönstret som öppnas med Window->Show view->Memory Browser. Ange en adress, t ex 0x20000000, för att få minnesinnehållet beskrivet i hexadecimal form. Det går även att ange ett namn för att hitta dess placering i minnet, t ex main eller stack. Detta kan användas för att hitta var i programminnet en viss rutin har placerats.

Beskrivningen av minnesinnhållet kan göras på olika format, inklusive enskilda byte respektive 32-bitars värden, och med olika radix (hexadecimalt, decimalt etc.).

Även innehåll hos I/O-enheterna kan ses i Memory Browser, men är lättare att hanter i registervyn (se "Undersöka I/O-enherternas registervärden" nedan).

3.6.1 Format på disassembly lista

I disassembly-fönstret visas hur instruktionerna kodas i programminnet, och var de hamnat. Detta öppnas med Windows->Show view->Disassembly. För varje instruktion i assemblerkoden anges först radnummer (t ex 385 i figuren nedan) samt opcode och argument. Om det fanns en label anges den på raden efter (i exemplet printchar). Slutligen anges vilken adress (0x00000540) och värdet på den adressen som motsvarar instruktionen (0xF85F1170) och vad

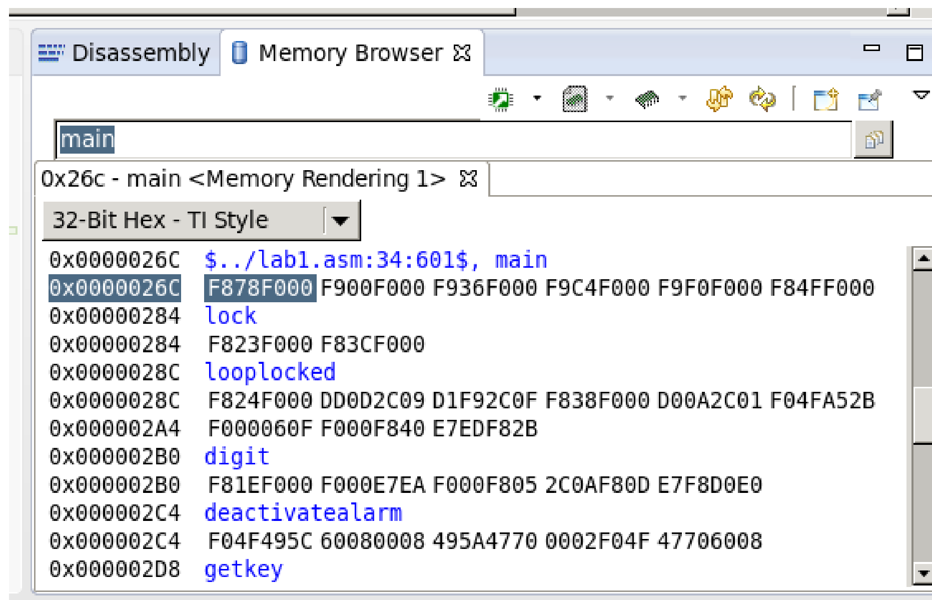

Figure 3.1: Minnesinnehåll i hexadecimal form

koden egentligen betyder (ldr.w r1,[pc, #-0x170]). Notera att assemblern ibland har bytt instruktion eller argument (här byttes ldr till ldr.w vid assembleringen).

3.7 Stega igenom ett program

I debugläget går det att sätta brytpunkter och att stega sig igenom programmet. Det går även att påverka registervärden och IO-enheter i Registersfönstret.

För att köra en maskininstruktion används Run->Assembly Step Into. Efter varje tryck uppdateras registervärdena i registervyn.

En annan version av steg består i att köra en hel subrutin och sedan stanna. Detta är lämpligt t ex för initieringsrutiner som måste köras i full fart, och rutiner som tar lång tid att köra. Om funktionen Run->Assembly Step Over väljs när nästa instruktion är ett subrutinanrop (bl-instruktion) så körs hela den subrutinen innan processorn stannar igen. Om instruktionen inte är ett subrutinanrop körs bara en instruktion precis som för Assembly Step Into.

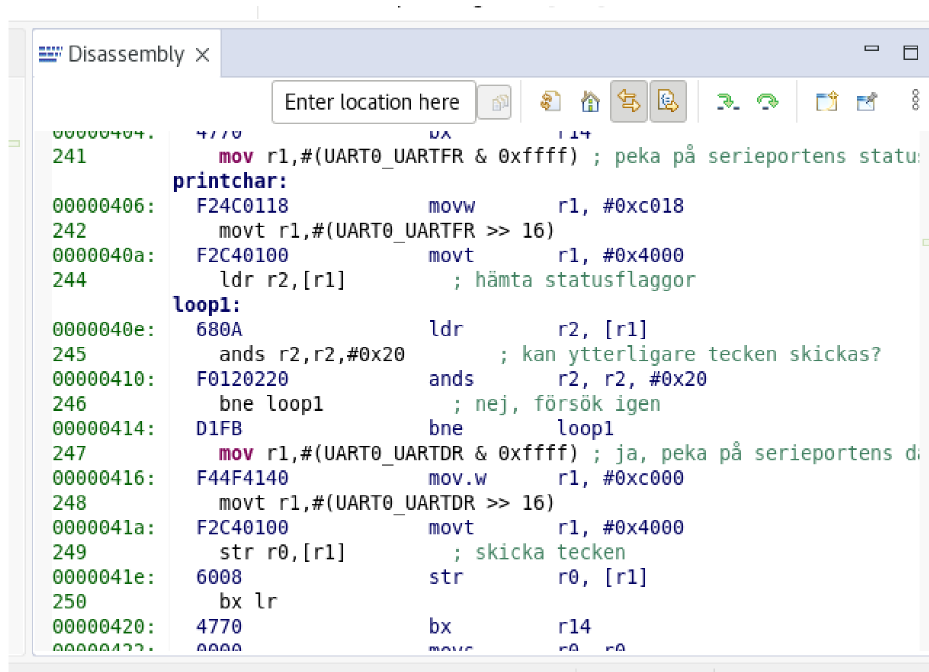

Figure 3.2: Disassembly vy med adresser, label och kod

3.8 Undersöka och ändra interna register

Alla register i processorn, inklusive I/O-enheter kan undersökas i registerfönstret. Öppna fönstret med Windows->Show view->Registers. Register r0-r15 (inkluderar SP,LR och PC) finns under ”core registers”.

Registervärden kan ändras genom att mata in nya värden. Eventuellt behöver uppdateraikonen (gula pilar i övre kanten av registerfönstret) väljas för att få det nya värdet att visas.

3.9 Undersöka I/O-enheternas registervärden

Alla I/O-enheter (GPIOB, GPIOC, etc.) har alla konfigurerings och dataregister tillgängliga i samma fönster (Registers) som de interna registren hos processorn. Skrolla ned en bit så hittas GPIO_PORTB, GPIO_PORTD etc.

3.10 Seriell kommunikation

Den seriella överföringen kan visas i ett fönster i Code Composer Studio. För att öppna detta trycker väljs först Window->Show view -> Other ->

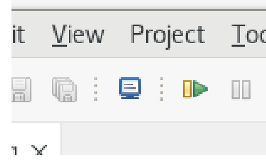

Figure 3.3: Ikon för att öppna terminal

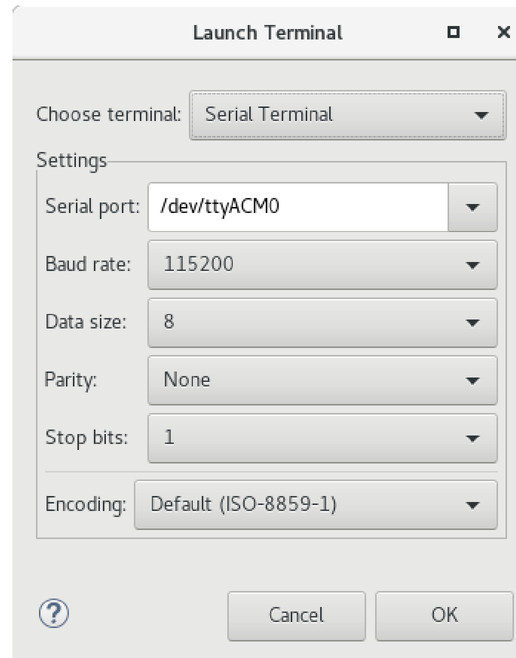

Figure 3.4: Inställning av seriell kommunikation i terminal

Terminal. Ett fönster nere till höger med titeln Terminal visas. Tryck på den blåvita skärmsymbolen, och fyll sedan i inställningar enligt figur 3.4 (dvs ”Serial Terminal”, Port: /dev/ttyACM0 samt baudrate 115200).

Ibland ska en annan port än ttyACM0 användas. Enklaste sättet att ta reda på rätt port är att i en terminal titta på innehållet i mappen /dev/serial/by-id. Dvs ge kommandont ”ls -l /dev/serial/by-id” och se till vilken ttyACM som filen /dev/serial/by-id/usb-Texas_Instruments_In-Circuit_Debug* pekar på.

Symbolen som ser ut som ett stort N med punkter på ändarna visar om porten har kontakt med kortet. Om denna symbol är röd har kortet kontakt med terminal, men om den istället är grå saknas kontakt. Då behöver en ny terminal öppnas.

Ett alternativ till interna seriella terminalen i ccstudio är gtkterm. Detta är ett externt program som har samma funktionalitet som seriell terminal i

ccstudio. Starta programmet mha kommandot tsea28terminal. Programmet väljer själv rätt anslutning etc. Notera att det inte går att både köra den interna seriella terminalen i ccstudio och tsea28terminal samtidigt.

Chapter 4

Processorn ARM Cortex-M4F

Mikrokontrollern TM4C123G är byggd runt processorn Cortex-M4. Detta kapitel ger en introduktion till denna processor, men kommer utelämna mycket detaljer som inte är nödvändiga för att kunna arbeta med Darma. En fullständig beskrivning av instruktionsuppsättningen finns i kapitel 3 i ARM Cortex-M4 Devices Generic User Guide (ARM DUI 0553A). Den hittas på https://developer.arm.com/documentation/dui0553/latest/

Denna processorfamilj (Cortex-M) stödjer enbart instruktionsuppsättningen kallad Thumb. Detta är en mindre version av den kompletta instruktionsuppsättningen definierad för ARM. T ex har villkorliga operationer fler begränsningar i Thumb än in full ARM.

4.1 Register

Programmeringsmodellen för processorn Cortex-M4 består av 17 stycken 32-bitar stora register. Det finns 16 generella register R0-R15, där tre av dessa register har speciella funktioner: R13 är stackpekare (SP), R14 är länkregister (LR), samt R15 är programräknare (PC). De alternativa namnen till R13-R15 används oftast. Det finns även ett statusregister (PSR) som innehåller flaggor samt information om aktuell avbrottsnivå och exekveringsläge.

Statusregistret innehåller flaggorna N (negative), Z (zero), C (carry) och V (overflow). Flaggorna påverkas inte av alla instruktioner, utan kan i vissa fall väljas genom att namnet avslutas med S (t ex ADDS). Z-flaggan sätts om resultatet var 0. N kopierar MSB-biten av resultatet. C-flaggan sätts om en minnessiffra generades ut från 32-bitarsaddition, om en subtraktion ger ett positivt (inklusive värdet 0) resultat, eller vid skiftoperationer.

V-flaggan sätts om resultatet gav spill, dvs om resultatets teckenbit (bit 31) inte stämmer med tecknet om beräkningen gjorts med oändlig nog-

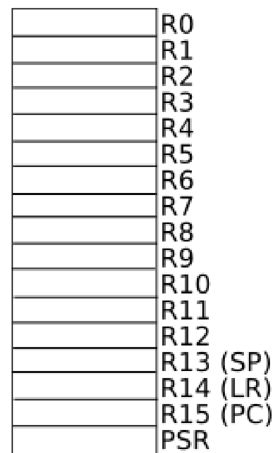

Figure 4.1: Registeruppsättning hos Cortex-M4

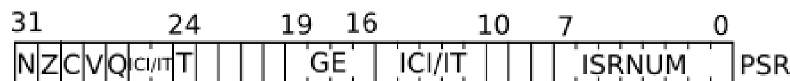

Figure 4.2: Bitdefinition hos statusregistret PSR hos Cortex-M4

grannhet. Exempel på detta är när addition av två negativa värden ger ett positivt resultat, när addition av två positiva tal ger ett negativt resultat, när subtraktion av ett negativt tal från ett positivt tal ger ett negativt resultat, och när subtraktion av ett positivt tal från ett negativt tal ger ett positivt resultat.

Övriga bitar i statusregistret inkluderar flaggan T som anger att processorn kör i Thumb mode (enda möjliga läget i Cortex-M), och ISRNUM anger vilken avbrottsnivå som körs förnärvarande.

Stackpekaren R13 minskar i värde när data läggs på stacken. Dvs den minskar värde vid PUSH, och ökar i värde vid POP.

4.2 Instruktionsuppsättning

Det finns väldigt många definierade instruktioner för Cortex-M4. De vanligaste kommer beskrivas här.

4.2.1 Notation

rd	destinationsregister
rn,rm	källregister
#imm8	32-bitars värde skapat genom valfritt antal vänsterskift

	av ett 8-bitars värde
#imm16	32-bitars värde skapat genom tillägg av 16 nollor till vänster om ett 16-bitars värde

4.2.2 Adresseringsmoder

Förutom omedelbar adressering är de flesta andra adresseringsmoderna endast använda i samband med instruktionerna LDR och STR.

omedelbar (literal, immediate)

#konstant Konstant med begränsad ordlängd. Endast vissa 32-bitars värden kan användas (se #imm8 och #imm16 ovan).

absolut, direkt

Stöds ej av ARM.

indirekt register

[rn] Adress i rn

indirekt register med offset

[rn,#offset] Adress beräknad som rn + offset

indirekt register med postincrement

[rn],#increment Använd adress i rn, öka sedan rn med increment

indirekt register med preincrement

[rn,#increment]! Beräkna först nytt rn = rn+increment, använd sedan nya rn som adress

register indirekt med register offset

[rn,rm] Använd adress = rn+rm

Register indirekt med skiftad register offset

[rn,rm,LSL #steps] Använd adress = rn+(rm << steps), dvs skifta rm steps steg åt vänster och lägg till rn.

4.2.3 Flyttinstruktioner

Processorn har en LOAD-STORE arkitektur, varför läsning/skrivning i minnet begränsas till LDR och STR instruktionerna (POP och PUSH kan ses som specialfall av dessa).

Ladda konstantvärde

```
MOV    rd,#imm8     ; rd = 32-bitars värde baserat på 8-bitars konstant
MVN    rd,#imm8     ; rd = 32-bitars inverterade versionen av 8-bitars
                    ; konstant
MOV    rd,#imm16    ; rd = 16-bitars konstant
MOVT   rd,#imm16    ; övre 16 bitarna av rd = konstant
ADR    rd,label     ; rd = label. Kan bara använda adresser
                    ; 0-4095 från instruktionens position
LDR    rd,label     ; Hämta konstant lagrad i programminnet på
                    ; adress label (max 4095 positioner från
                    ; instruktionens position). Detta implementeras
                    ; som register indirekt med offset addressering
                    ; relativt PC.
```

Flytt mellan register

```
MOV    rd,rm        ; kopiera 32-bitars värde i rm till rd
MOVS   rd,rm        ; kopiera 32-bitars värde i rm till rd, uppdatera
                    ; flaggorna N,Z,C
MVN    rd,rm        ; kopiera och bitvis invertera 32-bitars värde i rm
                    ; till rd
```

Flytt mellan register och minne

Adress i instruktionerna nedan beskrivs av någon av adresseringsmoderna utom omedelbar adressering.

Processorn använder omvänd byteordning (little-endian), så vid en läsning av ett 32-bitars ord från adress A till register rn placeras byte från adress A i de 8 minst signifikanta bitarna av rn (bitarna 7-0), från adress A+1 hämtas bitarna 15-8, från adress A+2 hämtas bitarna 23-16, samt från adress A+3 hämtas en byte för de mest signfikanta bitarna (31-24).

```
LDR    rd,adress    ; hämta 32-bitars värde från minne
LDRB   rd,adress    ; hämta 8-bitars värde från minne, nollställ bit
                    ; 31-8 i rd
```

```
LDRSB  rd,adress     ; hämta 8-bitars värde och teckenförläng till 32-bitar
LDRH   rd,adress     ; hämta 16-bitars värde från minne, nollställ bit
                     ; 31-16 i rd
LDRSH  rd,adress     ; hämta 16-bitars värde och teckenförläng till 32-bitar
STR    rm,adress     ; kopiera värdet i rm till minnet
STRB   rm,adress     ; spara 8 lägsta bitarna i rm i minnet
STRH   rm,adress     ; spara 16 lägsta bitarna i rm i minnet
```

Stackhantering

chapter04

```
PUSH   registerlista ; Placera alla register i registerlistan på stacken
POP    registerlista ; Hämta alla register i registerlistan från stacken
```

4.2.4 Beräkningsinstruktioner

Aritmetiska funktioner

Beräkningarna görs med 32-bitars värden.

```
ADD    rd,rn,#imm8   ; Addera, rd = rn + imm8
ADDS   rd,rn,#imm8   ; Addera, rd = rn + imm8, uppdatera flaggor
                     ; N,Z,C,V
ADD    rd,rm,rn      ; Addera, rd = rm + rn
ADDS   rd,rm,rn      ; Addera, rd = rm + rn, uppdatera flaggor
                     ; N,Z,C,V
ADC    rd,rn,#imm8   ; Addera, rd = rn + imm8 + C-flaggan
ADCS   rd,rn,#imm8   ; Addera, rd = rn + imm8 + C-flaggan, uppdatera
                     ; flaggor N,Z,C,V
ADC    rd,rm,rn      ; Addera, rd = rm + rn + C-flaggan
ADCS   rd,rm,rn      ; Addera, rd = rm + rn + C-flaggan, uppdatera
                     ; flaggor N,Z,C,V
SUB    rd,rn,#imm8   ; Subtrahera, rd = rn - imm8
SUBS   rd,rn,#imm8   ; Subtrahera, rd = rn - imm8, uppdatera flaggor
                     ; N,Z,C,V
SUB    rd,rm,rn      ; Subtrahera, rd = rm - rn
SUBS   rd,rm,rn      ; Subtrahera, rd = rm - rn, uppdatera flaggor
                     ; N,Z,C,V
SBC    rd,rn,#imm8   ; Subtrahera, rd = rn - imm8 - NOT(C-flaggan)
SBCS   rd,rn,#imm8   ; Subtrahera, rd = rn - imm8 - NOT(C-flaggan),
                     ; uppdatera flaggor N,Z,C,V
```

```
SBC     rd,rm,rn        ; Subtrahera, rd = rm - rn - NOT(C-flaggan)
SBCS    rd,rm,rn        ; Subtrahera, rd = rm - rn - NOT(C-flaggan),
                        ; uppdatera flaggor N,Z,C,V
```

Skift och rotation

```
ASR     rd,rm,#imm8; rd = rm aritmetiskt skiftad imm8 steg åt höger
ASR     rd,rm,rn   ; rd=rm skiftad rn steg åt höger
ASRS    rd,rm,#imm8; rd = rm aritmetiskt skiftad imm8 steg åt höger,
                   ; uppdatera flaggor N, Z,
LSL     rd,rm,#imm8; rd = rm logiskt skiftad imm8 steg åt vänster
LSLS    rd,rm,rn   ; rd = rm logiskt skiftad rn steg åt vänster,
                   ; påverka flaggor N,Z,C
LSR     rd,rm,#imm8; rd = rm logiskt skiftad imm8 steg åt höger
LSR     rd,rm,rn   ; rd = rm logisk skiftad rn steg åt höger
LSRS    rd,rm,#imm8; rd = rm logiskt skiftad imm8 åt höger,
                   ; påverka flaggor N,Z,C
LSRS    rd,rm,rn   ; rd = rm logiskt skiftad rn steg åt höger,
                   ; påverka flaggor N,Z,C
ROR ; Rotation höger
```

Jämförelse och test

```
CMP   rn,#imm8    ; Uppdatera flaggor N,Z,C,V beräknat på
                  ; rn - imm8
CMP   rn,rm       ; Uppdatera flaggor N,Z,C,V beräknat på
                  ; rn - rm
TST   rn,#imm8    ; Uppdatera flaggor N,Z,C,V beräknat på
                  ; rn AND imm8
TST   rn,rm       ; Uppdatera flaggor N,Z,C,V beräknat på
                  ; rn AND rm
```

Logiska funktioner

```
AND   rd,rn,#imm8 ; Bitvis AND, rd = rn AND imm8
AND   rd,rn,rm    ; Bitvis AND, rd = rn AND rm
ANDS  rd,rn,#imm8 ; Bitvis AND, rd = rn AND imm8, uppdatera
                  ; flaggor N,Z,C
ANDS  rd,rn,rm    ; Bitvis AND, rd = rn AND rm, uppdatera
                  ; flaggor N,Z,C
ORR   rd,rn,#imm8 ; Bitvis eller, rd = rn OR imm8
ORR   rd,rn,rm    ; Bitvis eller, rd = rn OR rm
```

```
ORRS  rd.rn,#imm8 ; Bitvis eller, rd = rn OR imm8,
                  ; uppdatera flaggor N,Z,C
ORRS  rd.rn,rm    ; Bitvis eller, rd = rn OR rm,
                  ; uppdatera flaggor N,Z,C
EOR   rd,rn,#imm8 ; Bitvis exklusiv eller, rd = rn EOR imm8
EOR   rd,rn,rm    ; Bitvis exklusiv eller, rd = rn EOR rm
EORS  rd.rn,#imm8 ; Bitvis exklusiv eller, rd = rn EOR imm8,
                  ; uppdatera flaggor N,Z,C
EORS  rd.rn,rm    ; Bitvis exklusiv eller, rd = rn EOR rm,
                  ; uppdatera flaggor N,Z,C
```

4.2.5 Hoppinstruktioner

ovillkorliga hopp

```
B     label       ; Hoppa till programrad indikerad av label
```

vilkorliga hopp

Villkorliga hopp kan göras på en mängd olika kombinationer av flaggor.

```
Bcc   label       ; hoppa till label om villkor cc är uppfyllt
```

cc	Flaggor	Hoppa om
EQ	Z=1	Lika
NE	Z=0	Olika
CS / HS	C=1	Unsigned större än eller lika
CC / LO	C=0	Unsigned mindre än
MI	N=1	negativt
PL	N=0	positivt
VS	V=1	Spill
VC	V=0	Inget spill
HI	C=1 och Z=0	Unsigned större än
LS	C=0 eller Z=1	Unsigned mindre än eller lika
GE	(N=1 och V=1) eller (N=0 och V=0)	större än eller lika
LT	(N=1 och V=0) eller (N=0 och V=1)	mindre än
GT	Z=0 och ((N=1 och V=1) eller (N=0 och V=0))	större än
LE	Z=1 eller (N=1 och V=0) eller (N=0 och V=1)	mindre än eller lika med

Subrutinanrop

```
BL      label       ; Spara PC i LR, hoppa sedan till label
```

BL sätter även LSB i LR till 1 då processorn kör i thumbläge. Om LR redan innan innehåller en återhoppsadress som ska användas senare, så måste LR först sparas på stack (med en extra instruktion) innan subrutinanropet görs.

Återhopp från subrutin

```
BX      LR          ; återställ PC från LR (LR får PC:s nuvarande
                    ; värde).
```

Diverse instruktioner

Korta fördröjningar kan skapas genom att utföra instruktioner. De flesta instruktioner tar 1 klockcykel att utföra.

```
NOP                 ; Instruktion som inte gör någon operation.
```

4.3 Avbrottshantering

Avbrott är en viktig del av en processors funktion. Detta system möjliggör hantering av händelser som identifierats av olika I/O-enheter eller interna tester under körning av programmet.

4.3.1 Avbrottsvektorer

Det finns en stor mängd avbrott som kan genereras, dels sådana som genereras internt i processorn som reset när strömmen slås på och när en okänd instruktion utförs, dels avbrott från I/O-enheter som GPIO och uart. Varje enskild källa till avbrott har en egen avbrottsvektor som ligger lagrad i avbrottvektortabellen i början av programminnet. Avbrottvektorn innehåller då adressen där avbrottsrutinen startar.

För att styra vilken enhet som får generera ett avbrott och vilken prioritetsordning som ska gälla finns ett antal register i I/O-enheten NVIC (Nested Vectored Interrupt Controller) som kontrollerar och prioriterar avbrott.

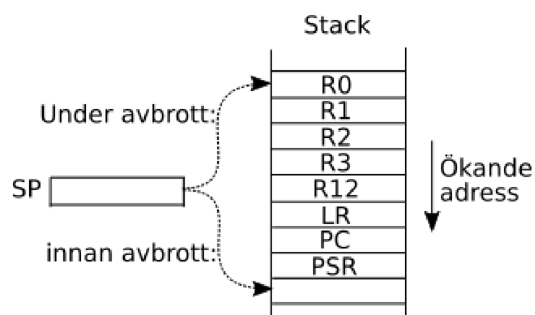

Figure 4.3: Stackens utseende före och under avbrott om stackens adress jämnt delbar med 8

4.3.2 Inställning för val av avbrott

För att kunna generera avbrott måste processorn först sätta upp I/O-enheten så den kan producera avbrott. Beroende på typ av I/O-enhet kan detta vara mer eller mindre komplicerat. Se avsnitt 2.2.1 för en närmare beskrivning av GPIO-portens inställningar.

Alla olika I/O-enheter som kan generera avbrott skickar avbrottssignalerna vidare till NVIC-enheten, som definierar vilken prioritet respektive I/O-enhet har, och om avbrott är tillåtna från enheten. Detta används för att bestämma om en ny avbrottsbegäran ska nå processorn, och vilken avbrottsnivå som då ska gälla. Ett avbrott på en lägre avbrottsnivå kan då avbryta ett pågående avbrott på en högre avbrottsnivå.

Processorn kan också själv bestämma om avbrott ska få ske genom att använda de två instruktionerna

```
CPSIE I        ; tillåt avbrott
CPSID I        ; tillåt inte avbrott
```

4.3.3 Händelser vid avbrott

När processorn tillåter att avbrott sker kommer ett avbrott från en I/O-enhet att först avbryta pågående program/lägre prioriterat avbrott. Om stackens adress då inte är jämnt delbar med 8 läggs ett extra värde (aligner) på stacken och bit 9 i PSR sätts till 1. Därefter lagras 8 registervärden på stacken. Dessa är r0-r3, r12, LR, PC och PSR vilket visas i figur 4.3. Samtidigt hämtas i vektortabellen motsvarande avbrottsrutins startadress och placeras i PC-registret. Dessutom sätts LR-registret till en speciell adress som indikerar att det har skett ett avbrott (0xfffffff1 eller 0xfffffff9).

Avbrottsrutinen kan nu köras, och eftersom flertalet register redan sparats kan r0-r3 och r12 användas utan att påverka huvudprogrammet senare. Däremot behöver lr sparas ifall subrutiner behöver anropas.

Det är lämpligt för avbrottsrutinen att så fort som möjligt inaktivera källan till avbrottet, t ex genom att nollställa avbrottsflaggan i GPIO porten.

När avbrottsrutinen är färdig återlämnas kontrollen till huvudprogrammet genom att återställa PC till LR-registrets värde. Eftersom denna adress är en speciell adress som inte finnns i vanliga program kan rätt antal register återställas inklusive statusregistret. Huvudprogrammet som avbröts kommer då fortsätta som om inget har hänt.

4.3.4 Återhopp från avbrott

Avbrott avslutas lämpligen med att återställa PC från LR. Enligt rekommendationer från ARM bör detta göras med följande instruktion.

```
BX      LR              ; återställ PC från LR
```

4.3.5 Stanna processor i väntan på avbrott

Processorn har möjlighet att sluta utföra instruktion i väntan på att ett avbrott sker. Detta gör att energiförbrukningen minskar eftersom inga nya instruktioner hämtas och utförs. Detta kan göras med följande instruktion.

```
WFI                     ; Stanna processorn, vänta på interrupt
```

När avbrott sedan sker kommer avbrottsrutinen startas, och när avbrottet avslutas fortsätter processorn med instruktionen som ligger efter WFI.

4.4 Assembler pseudoinstruktioner

Det finns en del instruktioner som inte utförs av processorn utan styr assemblatorn. Dessa kan påverka minnesinnehåll i programminnet eller definera konstanter som kan användas istället för numeriska värden i instruktionerna.

```
.thumb              ; Koden ska köras i thumb läge (vissa
                    ; processorer kan stödja fler
                    ; instruktionsuppsättningar)
.text               ; Definitionerna ska placeras i flashminnet
.align  4           ; Starta på en adress jämnt delbar med 4
.global main        ; Startadress för main ska vara
```

```
                                       ; tillgänglig för andra filer
namn1  .equ    0x2323                  ; konstanten namn1 får värdet 0x2323
namn2  .field  0xdeadbeef, 32          ; placera 32-bitars värdet 0xdeadbeef i
                                       ; programminnet. Konstanten namn2
                                       ; innehåller adressen där värdet ligger i
                                       ; programminnet. Konstanter längre än 8 bitar
                                       ; lagras i little-endian ordning
namn3  .string "hej",10,13             ; Placera en sträng med avslutande retur
                                       ; och radmatning i programminnet
namn4  .byte   0,1,0x2,0x3,0x4         ; Placera enskilda byte i minnet
```

Numeriska beräkning kan göras av konstanter, vilket är användbart för att t ex ladda ett 32-bitars värde i ett register. Exempel:

```
       mov     r1,#(0x12345678 & 0xffff)
                                       ; nollställ översta 16 bitarna i r1 och
                                       ; placera värdet 0x00005678 i r1
       movt    r1,#(0x12345678 >> 16)
                                       ; placera värdet 0x1234 i de högsta
                                       ; 16 bitarna av r1
       ; värdet 0x12345678 har nu placerats i r1
```

4.5 Fullständig instruktionslista

I nedanstående tabell står Op2 för något av registren R0-R15, eller en konstant.

Table 4.1: Fullständig instruktionslista.

Mnemonic	Operander	Kort beskrivning	Flaggor
ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V
ADD, ADDW	{Rd,} Rn , #imm12	Add	-
ADR	Rd, label	Load PC-relative address	-
AND, ANDS	{Rd,} Rn, Op2	Logical AND	N,Z,C
ASR, ASRS	Rd, Rm, ⟨Rs|#n⟩	Arithmetic shift right	N,Z,C
B	label	Branch	-
BFC	Rd, #lsb, #width	Bit field clear	-
BFI	Rd, Rn, #lsb, #width	Bit field insert	-
BIC, BICS	{Rd,} Rn, Op2	Bit clear	N,Z,C
BKPT	#imm	Breakpoint	-
BL	label	Branch with link	-
BLX	Rm	Branch indirect with link	-
BX	Rm	Branch indirect	-

Fortsätter på nästa sida

Table 4.1 – fortsättning från föregående sida

Mnemonic	Operander	Kort beskrivning	Flaggor
CBNZ	Rn, label	Compare and branch if non-zero	-
CBZ	Rn, label	Compare and branch if zero	-
CLREX	-	Clear exclusive	-
CLZ	Rd, Rm	Count leading zeros	-
CMN	Rn, Op2	Compare negative	N,Z,C,V
CMP	Rn, Op2	Compare	N,Z,C,V
CPSID	i	Change processor state, disable interrupts	-
CPSIE	i	Change processor state, enable interrupts	-
DMB	-	Data memory barrier	-
DSB	-	Data synchronization barrier	-
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR	N,Z,C
ISB	-	Instruction synchronization barrier	-
IT	-	If-Then condition block	-
LDM	Rn{!}, reglist	Load multiple registers, increment after	-
LDMDB, LDMEA	Rn{!}, reglist	Load multiple registers, decrement before	-
LDMFD, LDMIA	Rn!, reglist	Load multiple registers, increment after	-
LDR	Rt, [Rn, #offset]	Load register with word	-
LDRB, LDRBT	Rt, [Rn, #offset]	Load register with byte	-
LDRD	Rt, Rt2, [Rn, #offset]	Load register with two bytes	-
LDREX	Rt, [Rn, #offset]	Load register exclusive	-
LDREXB	Rt, [Rn]	Load register exclusive with byte	-
LDREXH	Rt, [Rn]	Load register exclusive with halfword	-
LDRH, LDRHT	Rt, [Rn, #offset]	Load register with halfword	-
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load register with signed byte	-
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load register with signed halfword	-
LDRT	Rt, [Rn, #offset]	Load register with word	-
LSL, LSLS	Rd, Rm, ⟨Rs|#n⟩	Logical shift left	N,Z,C
LSR, LSRS	Rd, Rm, ⟨Rs|#n⟩	Logical shift right	N,Z,C
MLA	Rd, Rn, Rm, Ra	Multiply with accumulate, 32-bit result	-
MLS	Rd, Rn, Rm, Ra	Multiply and subtract, 32-bit result	-
MOV, MOVS	Rd, Op2	Move	N,Z,C
MOV, MOVW	Rd, #imm16	Move 16-bit constant	N,Z,C
MOVT	Rd, #imm16	Move top	-
MRS	Rd, spec_reg	Move from special register to general register	-
MSR	spec_reg, Rm	Move from general register to special register	N,Z,C,V
MUL, MULS	{Rd,} Rn, Rm	Multiply, 32-bit result	N,Z
MVN, MVNS	Rd, Op2	Move NOT	N,Z,C
NOP	-	No operation	-
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT	N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR	N,Z,C
PKHTB, PKHBT	{Rd,} Rn, Rm, Op2	Pack halfword	-
POP	reglist	Pop registers from stack	-
PUSH	reglist	Push registers onto stack	-
QADD	{Rd,} Rn, Rm	Saturating add	Q
QADD16	{Rd,} Rn, Rm	Saturating add 16	-
QADD8	{Rd,} Rn, Rm	Saturating add 8	-
QASX	{Rd,} Rn, Rm	Saturating add and subtract with exchange	-
QDADD	{Rd,} Rn, Rm	Saturating double and add	Q

Fortsätter på nästa sida

Table 4.1 – fortsättning från föregående sida

Mnemonic	Operander	Kort beskrivning	Flaggor
QDSUB	{Rd,} Rn, Rm	Saturating double and subtract	Q
QSAX	{Rd,} Rn, Rm	Saturating subtract and add with exchange	-
QSUB	{Rd,} Rn, Rm	Saturating subtract	Q
QSUB16	{Rd,} Rn, Rm	Saturating subtract 16	-
QSUB8	{Rd,} Rn, Rm	Saturating subtract 8	-
RBIT	Rd, Rn	Reverse bits	-
REV	Rd, Rn	Reverse byte order in a word	-
REV16	Rd, Rn	Reverse byte order in each halfword	-
REVSH	Rd, Rn	Reverse byte order in bottom halfword and sign extend	-
ROR, RORS	Rd, Rm, ⟨Rs|#n⟩	Rotate right	N,Z,C
RRX, RRXS	Rd, Rm	Rotate right with extend	N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse subtract	N,Z,C,V
SADD16	{Rd,} Rn, Rm	Signed add 16	GE
SADD8	{Rd,} Rn, Rm	Signed add 8	GE
SASX	{Rd,} Rn, Rm	Signed add and subtract with exchange	GE
SBC, SBCS	{Rd,} Rn, Op2	Subtract with carry	N,Z,C,V
SBFX	Rd, Rn, #lsb, #width	Signed bit field extract	-
SDIV	{Rd,} Rn, Rm	Signed divide	-
SEL	{Rd,} Rn, Rm	Select bytes	-
SEV	-	Send event	-
SHADD16	{Rd,} Rn, Rm	Signed halving add 16	-
SHADD8	{Rd,} Rn, Rm	Signed halving add 8	-
SHASX	{Rd,} Rn, Rm	Signed halving add and subtract with exchange	-
SHSAX	{Rd,} Rn, Rm	Signed halving add and subtract with exchange	-
SHSUB16	{Rd,} Rn, Rm	Signed halving subtract 16	-
SHSUB8	{Rd,} Rn, Rm	Signed halving subtract 8	-
SMLABB, SMLABT, SMLATB, SMLATT	Rd, Rn, Rm, Ra	Signed multiply accumulate long (halfwords)	Q
SMLAD, SMLADX	Rd, Rn, Rm, Ra	Signed multiply accumulate dual	Q
SMLAL	RdLo, RdHi, Rn, Rm	Signed multiply with accumulate (32x32+64), 64-bit result	-
SMLALBB, SMLALBT, SMLALTB, SMLALTT	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long (halfwords)	-
SMLALD, SMLALDX	RdLo, RdHi, Rn, Rm	Signed multiply accumulate long dual	-
SMLAWB, SMLAWT	Rd, Rn, Rm, Ra	Signed multiply accumulate, word by halfword	Q
SMLSD, SMLSDX	Rd, Rn, Rm, Ra	Signed multiply subtract dual	Q
SMLSLD, SMLSLDX	RdLo, RdHi, Rn, Rm	Signed multiply subtract long dual	
SMMLA	Rd, Rn, Rm, Ra	Signed most significant word multiply accumulate	-
SMMLS, SMMLR	Rd, Rn, Rm, Ra	Signed most significant word multiply subtract	-
SMMUL, SMMULR	Rd, Rn, Rm	Signed most significant word multiply	-
SMUAD, SMUADX	Rd, Rn, Rm	Signed dual multiply add	Q
SMULBB, SMULBT, SMULTB, SMULTT	Rd, Rn, Rm	Signed multiply halfwords	-
SMULL	RdLo, RdHi, Rn, Rm	Signed multiply (32x32), 64-bit result	-
SMULWB, SMULWT	Rd, Rn, Rm	Signed multiply by halfword	-
SMUSD, SMUSDX	Rd, Rn, Rm	Signed dual multiply subtract	-
SSAT	Rd, #n, Rm {,shift #s}	Signed saturate	Q

Fortsätter på nästa sida

Table 4.1 – fortsättning från föregående sida

Mnemonic	Operander	Kort beskrivning	Flaggor
SSAT16	Rd, #n, Rm	Signed saturate 16	Q
SSAX	{Rd,} Rn, Rm	Saturating subtract and add with exchange	GE
SSUB16	{Rd,} Rn, Rm	Signed subtract 16	-
SSUB8	{Rd,} Rn, Rm	Signed subtract 8	-
STM	Rn{!}, reglist	Store multiple registers, increment after	-
STMDB, STMEA	Rn{!}, reglist	Store multiple registers, decrement before	-
STMFD, STMIA	Rn{!}, reglist	Store multiple registers, increment after	-
STR	Rt, [Rn {, #offset}]	Store register word	-
STRB, STRBT	Rt, [Rn , #offset]	Store register byte	-
STRD	Rt, Rt2, [Rn , #offset]	Store register two words	-
STREX	Rt, Rt, [Rn , #offset]	Store register exclusive	-
STREXB	Rd, Rt, [Rn]	Store register exclusive byte	-
STREXH	Rd, Rt, [Rn]	Store register exclusive halfword	-
STRH, STRHT	Rt, [Rn {, #offset}]	Store register halfword	-
STRSB, STRSBT	Rt, [Rn {, #offset}]	Store register signed byte	-
STRSH, STRSHT	Rt, [Rn {, #offset}]	Store register signed halfword	-
STRT	Rt, [Rn {, #offset}]	Store register word	-
SUB, SUBS	{Rd,} Rn, Op2	Subtract	N,Z,C,V
SUB, SUBW	{Rd,} Rn, #imm12	Subtract 12-bit constant	N,Z,C,V
SVC	#imm	Supervisor call	-
SXTAB	{Rd,} Rn, Rm, {,ROR #}	Extend 8 bits to 32 and add	-
SXTAB16	{Rd,} Rn, Rm, {,ROR #}	Dual extend 8 bits to 16 and add	-
SXTAH	{Rd,} Rn, Rm, {,ROR #}	Extend 16 bits to 32 and add	-
SXTB16	{Rd,} Rm {,ROR #n}	Signed extend byte 16	-
SXTB	{Rd,} Rm {,ROR #n}	Sign extend a byte	-
SXTH	{Rd,} Rm {,ROR #n}	Sign extend a halfword	-
TBB	[Rn, Rm]	Table branch byte	-
TBH	[Rn, Rm, LSL #1]	Table branch halfword	-
TEQ	Rn, Op2	Test equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C
UADD16	Rd, Rn, Rm	Unsigned add 16	GE
UADD8	Rd, Rn, Rm	Unsigned add 8	GE
UASX	{Rd,} Rn, Rm	Unsigned add and subtract with exchange	GE
UHADD16	Rd, Rn, Rm	Unsigned halving add 16	-
UHADD8	Rd, Rn, Rm	Unsigned halving add 8	-
UHASX	Rd, Rn, Rm	Unsigned halving add and subtract with exchange	-
UHSAX	Rd, Rn, Rm	Unsigned halving subtract and add with exchange	-
UHSUB16	{Rd,} Rn, Rm	Unsigned halving subtract 16	-
UHSUB8	Rd, Rn, Rm	Unsigned halving subtract 8	-
UBFX	Rd, Rn, #lsb, #width	Unsigned bit field extract	-
UDIV	Rd, Rn, Rm	Unsigned divide	-
UMAAL	RdLo, RdHi, Rn, Rm	Unsigned multiply accumulate accumulate long (32x32+64), 64-bit result	-
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned multiply with accumulate (32x32+32+32), 64-bit result	-
UMULL	RdLo, RdHi, Rn, Rm	Unsigned multiply (32x 2), 64-bit result	-
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16	-
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8	-

Fortsätter på nästa sida

Table 4.1 – fortsättning från föregående sida

Mnemonic	Operander	Kort beskrivning	Flaggor
UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange	-
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange	-
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16	-
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8	-
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences	-
USADA8	{Rd,} Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate	-
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate	Q
USAT16	Rd, #n, Rm	Unsigned Saturate 16	Q
USAX	{Rd,} Rn, Rm	Unsigned Subtract and add with Exchange	GE
USUB16	{Rd,} Rn, Rm	Unsigned Subtract 16	GE
USUB8	{Rd,} Rn, Rm	Unsigned Subtract 8	GE
UXTAB	{Rd,} Rn, Rm, {,ROR #}	Rotate, extend 8 bits to 32 and Add	-
UXTAB16	{Rd,} Rn, Rm, {,ROR #}	Rotate, dual extend 8 bits to 16 and Add	-
UXTAH	{Rd,} Rn, Rm, {,ROR #}	Rotate, unsigned extend and Add Halfword	-
UXTB	{Rd,} Rm, {,ROR #n}	Zero extend a Byte	-
UXTB16	{Rd,} Rm, {,ROR #n}	Unsigned Extend Byte 16	-
UXTH	{Rd,} Rm, {,ROR #n}	Zero extend a Halfword	-
VABS.F32	Sd, Sm	Floating-point Absolute	-
VADD.F32	{Sd,} Sn, Sm	Floating-point Add	-
VCMP.F32	Sd, ⟨Sm|#0.0⟩	Compare two floating-point registers, or one floating-point register and zero	FPSCR
VCMPE.F32	Sd, ⟨Sm|#0.0⟩	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check	FPSCR
VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer	-
VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point	-
VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding	-
VCVT⟨B|H⟩.F32.F16	Sd, Sm	Converts half-precision value to single-precision	-
VCVTT⟨B|T⟩.F32.F16	Sd, Sm	Converts single-precision register to half-precision	-
VDIV.F32	{Sd,} Sn, Sm	Floating-point Divide	-
VFMA.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Accumulate	-
VFNMA.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Accumulate	-
VFMS.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Subtract	-
VFNMS.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Subtract	-
VLDM.F⟨32|64⟩	Rn{!}, list	Load Multiple extension registers	-
VLDR.F⟨32|64⟩	⟨Dd|Sd⟩, [Rn]	Load an extension register from memory	-
VLMA.F32	{Sd,} Sn, Sm	Floating-point Multiply Accumulate	-
VLMS.F32	{Sd,} Sn, Sm	Floating-point Multiply Subtract	-
VMOV.F32	Sd, #imm	Floating-point Move immediate	-

Fortsätter på nästa sida

Table 4.1 – fortsättning från föregående sida

Mnemonic	Operander	Kort beskrivning	Flaggor
VMOV	Sd, Sm	Floating-point Move register	-
VMOV	Sn, Rt	Copy ARM core register to single precision	-
VMOV	Sm, Sm1, Rt, Rt2	Copy 2 ARM core registers to 2 single precision	-
VMOV	Dd[x], Rt	Copy ARM core register to scalar	-
VMOV	Rt, Dn[x]	Copy scalar to ARM core register	-
VMRS	Rt, FPSCR	Move FPSCR to ARM core register or APSR	N,Z,C,V
VMSR	FPSCR, Rt	Move to FPSCR from ARM Core register FPSCR	-
VMUL.F32	{Sd,} Sn, Sm	Floating-point Multiply	-
VNEG.F32	Sd, Sm	Floating-point Negate	-
VNMLA.F32	{Sd,} Sn, Sm	Floating-point Multiply and Add	-
VNMLS.F32	{Sd,} Sn, Sm	Floating-point Multiply and Subtract	-
VNMUL	{Sd,} Sn, Sm	Floating-point Multiply	-
VPOP	list	Pop extension registers	-
VPUSH	list	Push extension registers	-
VSQRT.F32	Sd, Sm	Calculates floating-point Square Root	-
VSTM	Rn{!}, list	Floating-point register Store Multiple	-
VSTR.F3⟨32|64⟩	Sd, [Rn]	Stores an extension register to memory	-
VSUB.F⟨32|64⟩	{Sd,} Sn, Sm	Floating-point Subtract	-
WFE	-	Wait for event	-
WFI	-	Wait for interrupt	-

Chapter 5

Vanliga frågor

- Jag hittar inte någon Tiva C series i Target, var finns den?
  Svar: Ibland spårar system ur, och kan behöva rensas. För att få allt som det var från början kan man ta bort filerna i ~/ti, ~/.ti, ~/.TI-trace, och ~/.eclipse. Eventuellt kan även filer i workspace tas bort, men se då till att först kopiera/flytta de .asm-filer som du vill behålla.

- När jag provkör programmet hamnar jag i rutinen FaultISR, varför?
  Svar: Det finns ett flertal anledningar till att man hamnar här. Följande saker bör kontrolleras: a) Stackpekaren pekar utanför RAM-minnets adressrymd, vilket kan kontrolleras i Registers-fliken. b) En otillåten adress läses/skrivs (t ex udda adress i programminnet eller en adress utanför program, data och I/O-minnena) c) Är stackstorleken satt vid länkning (markera projektet i Project Explorer, File->Properties, välj ARM Linker, kontrollera att i rutan Summary of flags set finns argumentet –stack_size=512).

- Ingen utskrift syns trots att jag ser (mha stegning genom koden) att koden skriver data till serieporten.
  Svar: Ibland kan terminalen i ccstudio missa att koppla sig mot serieporten. Detta syns främst som att ikonen som liknar ett stort N med fyllda cirklar i ändarna är grå. Försök i så fall att öppna terminalen på nytt.

- Fråga: När jag försöker undersöka GPIO-portens värden anges värdet ”Error: unable to read”.
  Svar: En port och dess register kan bara läsas och skrivas till om porten har startats. Initieringsrutinen för porten måste därför köras först innan du kan titta på och ändra portens register.

- När jag försöker köra programmet får jag en ruta med texten "Error connecting to the target. Frequency is out of range."
  Svar: Om du kör på en maskin i MUX-labbet beror det troligen på att strömmen inte är på. Slå på spänningsaggregatet. Om strömmen är på (den gröna lysdioden på det röda kretskortet lyser) så är det problem med USB-anslutningen. Försök i så fall att först stänga av strömmen till Darma, vänta 10 sekunder, och slå sedan på strömmen igen.

- När jag startar ccstudio kan jag inte välja den workspace jag använt tidigare och jag får ett felmeddelande att workspace inte kan skapas eller används av någon annan.
  Svar: Detta kan bero på att programmet inte stängts korrekt (t ex strömmen brutits på datorn). Enklaste sättet att korrigera detta är att börja om med ett nytt workspace, och kopiera från det gamla workspace de få filer som du skapat/ändrat själv.

- Jag får felmeddelandet "No source available for main"
  Svar: Detta beror på att du placerat kod/definitioner ovanför definition av main. Flytta allt du lagt till ovanför main till någonstans nedanför main.

- Vid assemblering fås felmeddelandet "Destination must be on a 4 byte boundary"
  Svar: Instruktionen refererar mot en adress som inte jämnt delbar med 4. Positionen för den adressen behöver därför placeras på en adress som är jämnt delbar med 4 mha kommandot .align 4. Notera att instruktionen i sig ligger på en jämn adress, men datakonstanten som instruktionen refererar mot har hamnat snett.

- Kompilering ger felmeddelande "ERROR label defined differently in each pass"
  Svar: En möjlig lösning kan vara att placera en .align med ett större tal (t ex 0x10) innan konstantdefinitionen. En annan möjlighet är att ändra ordningen på subrutiner och definitioner.

Revisioner

0.23	Avbrott tillagt, mer information om GPIO-enheten
0.3	Alignment i avbrottsstack, tagit bort mov pc,lr, lagt till movt
0.4	fix constant load code && => &
0.5	Fixed ORR instruction descriptions
0.6	Correct name for SCIPIO ICR-register
0.7	CentOS 7, ccstudio 9.3
0.8	Distansfunktion, bättre val ttyACM, korr. flaggor
0.81	Mer information om tsea28lab1
0.82	Korrigerat modulnamn och start med &
0.83	Utökat vanliga frågor, lagt till WFI
0.84	Mer information om tsea28lab2
0.85	Mer information om tsea28lab3
0.86	RHEL8, ccstudio 12.5