TEIS
Teknisk rapport

TEIS TEKNISK RAPPORT

Författare Lasse Karagiannis	Uppgift Task5
E-post adress lasse.l.karagiannis@gmail.com	Filnamn Lasse_Karagiannis_C_task5_komplettering_2.pdf

Low level C

CPU architecture

Lasse Karagiannis

16-10-16

Innehåll

KRAVSPECIFIKATION....3

Sammanfattning teorilektion 5 (Krav_001)....3

2.1 CPU....4

2.2 Programkonstruktion....4

2.3 Stack och heap....4

3 Förevisning av programexekvering från debuggern (Krav_003)....5

4 Förevisning av funktionsanrop (Krav_004)....7

KRAVSPECIFIKATION

Tabell 1 Kravspecifikation från kund

Krav	Beskrivning	Utfört, (ja/nej)
	Förstudie	
Krav_001	Gå igenom teorilektion 5 (Theory 5) och skriv en kort sammanfattning av lektionen vilken ska ingå i rapporten som eget kapitel, enligt krav_005.	
	Funktionskrav	
Krav_002	Skapa ett applikationsprojekt med C-programmet i bilagan. Samma BSP som i CASE 1C kan användas.	
Krav_003	Visa med debuggern och beskriv ett antal assembler instruktioner från en del av C-koden i ett separat kapitel. Kommentar: Frivilligt är att beskriva vad assembler-instruktionerna gör för att utföra en C-instruktion.	
Krav_004	Beskriv i ett separat kapitel i rapporten hur ett funktionsanrop sker. Visa programräknaren (pc) och stackpekaren (sp) före, under och efter ”delay_function”. Stega genom hela ”`main`” med en mycket kort vänteloop i funktionen ”`void delay_function(int time_delay)`”. Kommentar: *En total analys behöver inte utföras utan bara visa och få en känsla för hur ett anrop fungerar i praktiken. Svara på frågorna i kodexemplet i bilagan.*	
	Dokumentationskrav	
Krav_005	Sammanfoga dokumentationen från krav_001 till krav_004 till en läsbar rapport. Framsida med titel, en kort sammanfattning, innehållsförteckning och separata kapitel enligt krav_001till krav_004 ovan. Lägg även till eventuella slutsatser och referenser.	
	Leveranskrav	
Krav_006	Leveransen ska ske till plattformen Itslearning. Leveransen ska vara en rapport. Namnet på filen ska vara ”förnamn_efternamn_C_task_5”. Sista leveransdag se kursschema (för VG).	

Sammanfattning teorilektion 5 (Krav_001)

Avsnittet handlade om datorsystem arkitektur och programkonsruktion.

2.1 CPU

Nios II processorn har en separata bussar för data och instruktioner. Processorn (CPU) har en ALU -aritmetisk logisk enhetet, programräknare, interna register samt en styrenhet som avkodar instruktionerna. Koden som processorn exekverar adresseras ur instruktionsminnet med programräknaren PC.

2.2 Programkonstruktion

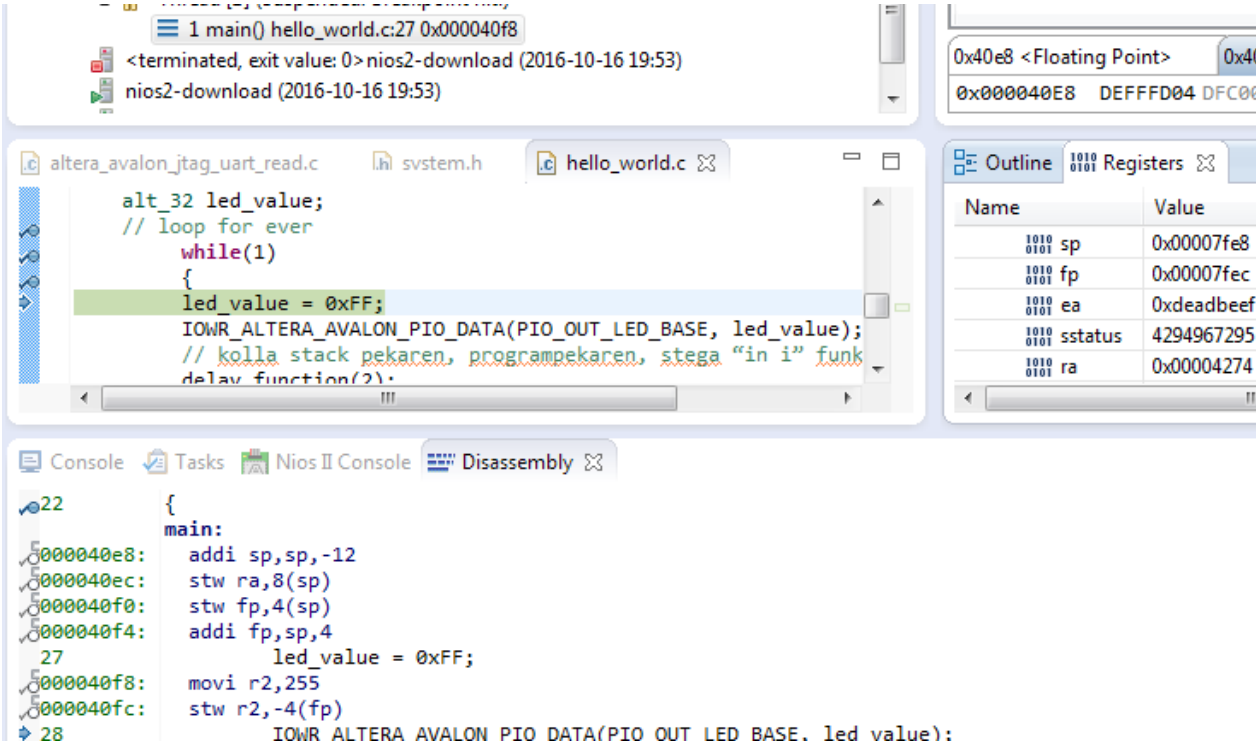
Programutveckling sker för Nios II I programspråket C. Vid utveckling används en kompilator som översätter programspråket till assemblykod. Denna kan assembleras i assemblatorn, som oftast är en integrerad del av kompilatorn och objektkod med symboliska adresser för variabler och funktioner. Länkaren länkar in bibliotek som programmet använder och ersätter de symboliska adresserna med fysiska adresser.

2.3 Stack och heap

Stacken används vid funktionsanrop och lagrar återhopps-adressen, funktionsargument och lokala variabler. Stacken växer emot minskande adresser och dess senaste dit pushade värde är adresserat med stackpekaren SP. För debuggning så används också FP (Frame Pointer), men den kan optimeras bort och dess register kan användas som ett generellt register.

Dynamiskt skapade variabler allokeras däremot på den s.k. Heapen. Denna får man hålla ordning på själv som programutvecklare, dvs. se till att man frigör minne som inte längre behövs. I programspråket C används funktionerna malloc och calloc.

3 Förevisning av programexekvering från debuggern (Krav_003)



Diskuterar här några få assemblerinstruktioner för main-funktionen. När man starta debuggern så stannar inte exekvering förräns man är inne i while-loopen. Programmet ignorerar samtliga brytpunkter innan dess, vidare är inte disassembleringen synkroniserad med C-koden. Debuggern pekar led_value = 0xFF som nästa instruktion att exekvera, medan debuggern pekar på rad 28. En diskrepans med programmet som återses i andra Eclipse-baserade programvaror för inbyggda system, såsom Atolic Studio.
Vi kan se att kompilatorn har genererat kod som skapar utrymme på stacken. För addi – instruktionen kan man läsa från instruktionslistan som återfinns på
https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii51017.pdf
att addi-instuktionen "Sign-extends the 16-bit immediate value" and adds it to ra and stores it in rb, enligt ekvationen addi rb ra imm(16).

Därefter görs stw ra 8(sp). Instuktionslistan ger följande förklaring till stw rB, byte_offset(rA) :
"Computes the effective byte address specified by the sum of rA and the instruction's signed 16-bit immediate value. Stores rB to the memory location specified by the effective byte address ".
Detta betyder att innehållet I ra lagras på adressen 8+sp.
Manualen berättar att ra är synonymt med register 31 och är återhopps adressen. Men återhoppsadressen för vad? Är det återhopps adressen för main funktionen? Ra har värdet 0x4274. Skrollar man neråt så ser man att det är adressen till instruktionen som kommer efter anropeet av main

00004270: call 0x40e8 <main>

00004274: stw r2,-4(fp)

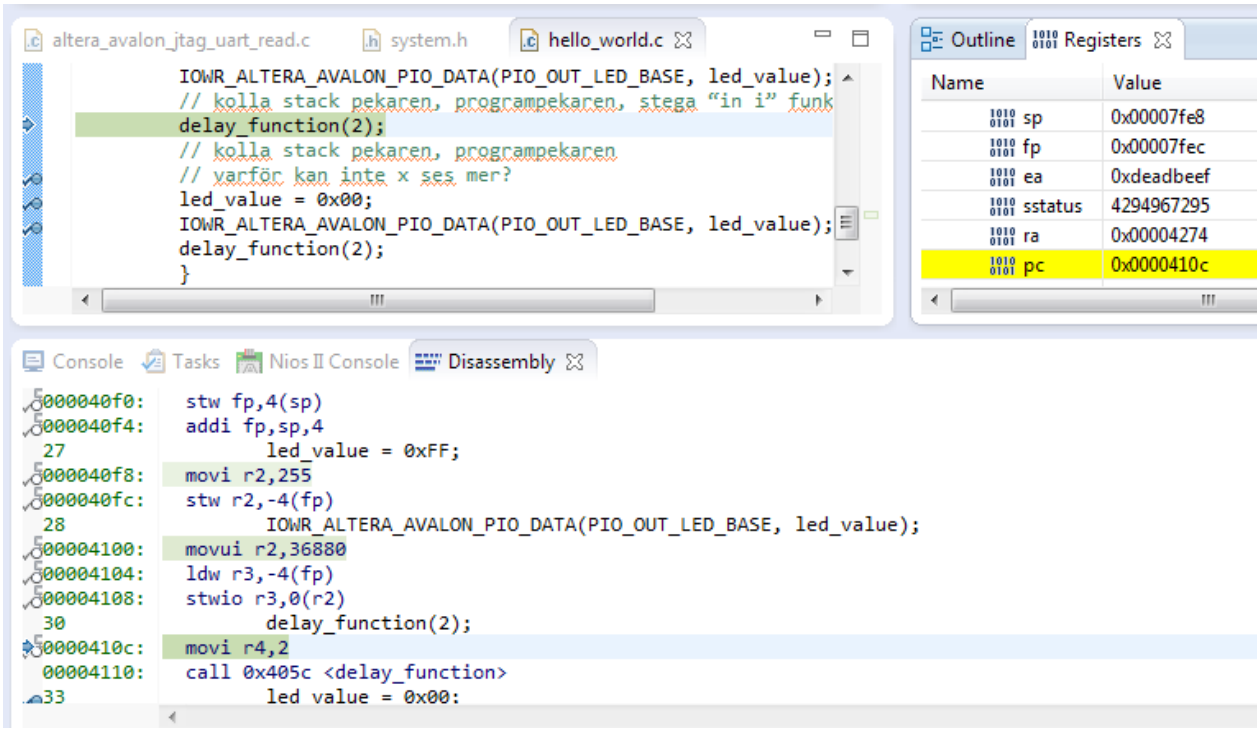
00004278: movi r4,1

Vi ser alltså att C-kompilatorn har genererat kod enligt konventionen att den anropade funktionen har ansvaret för att skapa utrymme på stacken och lagra undan register.

Går vi vidare så ser vi att stw fp, 4(sp), dvs. Innehållet I fp sparas undan på adress SP+4, därefter får FP SP+4. Vi kommer sedan till C-instruktionen `led_value = 0xFF;`
Denna implementeras med en movi r2, 255 (move immediate) dvs. R2 laddas med talet 255, därefter lagras detta på stacken stw r2,-4(fp). Detta betyder att adressen för variabeln led_value är -4+FP = -4+0x7FEC = 0x7FE8.

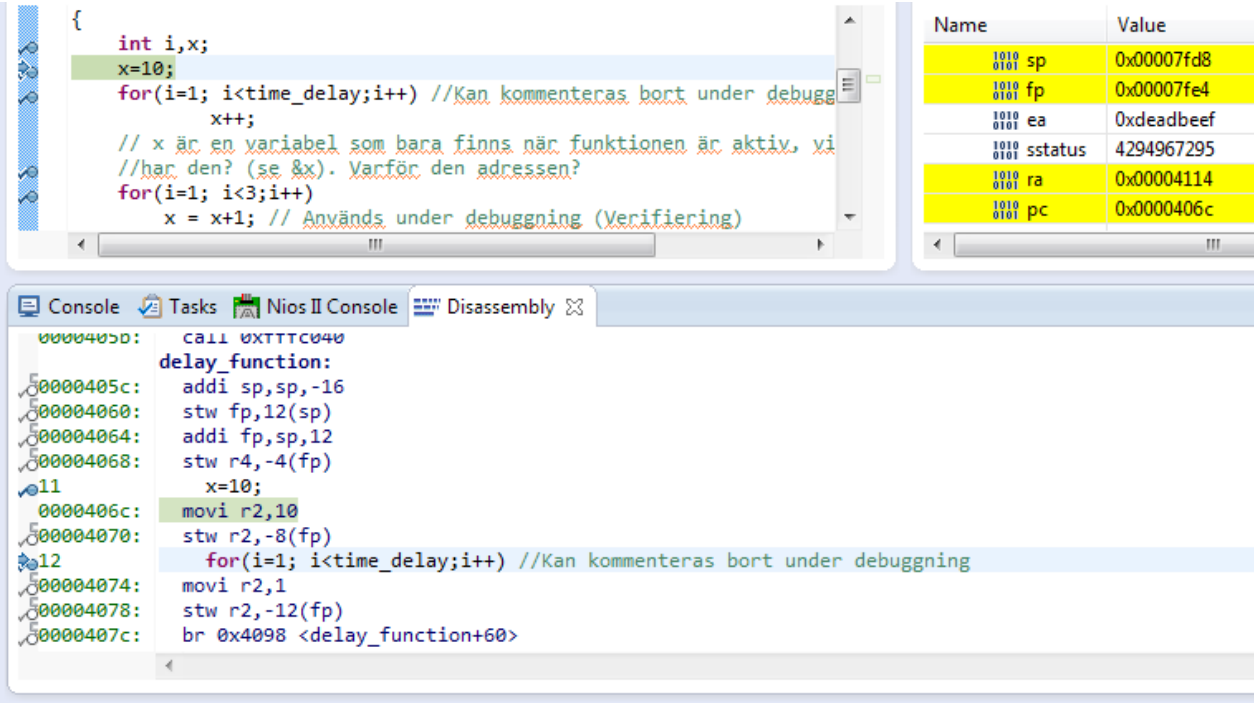
4 Förevisning av funktionsanrop (Krav_004)

Nedan ser vi debugger-vyn strax innan funktionsanropet görs:



Vi ser att att argumentet till delay_funnction, vilket var satt till två hanteras med instruktionen movi r4, 2, dvs. register r4 laddas med talet 2. Programräknaren befinner sig på 0x410C och SP och FP är oförändrade sedan de modifierades i början av main.

Vi stegar med ”step-into” och skrollar bakåt i disassemblyn för att även få med instruktionerna debuggern inte stannar vid och får då följande debugger vy:



Vi att stackpekarens värde har minskat med decimalt 16 till 0x7FD8, vilket stämmer om man subtraherar 16 från det gamla värdet 0x7FE8 (0x7FE8 – 0x10 = 0x7FD8).
PC pekar inne i subrutinen på adress 0x406C vars minnesinnehåll innehåller instruktionen movi r2,10, dvs. implementeringen av C-koden x = 10.
Sedan lagras r2 på stacken på adressen -8 + FP = -8 + 0x7FE4 = 0x7FDC, vilket betyder att adressen för variabeln x är 0x7FDC, och det faktum att den befinner sig på stacken, betyder att variabeln kommer att vara oåtkomlig efter att funktionen returnerat (SP återställs till tidigare värde)

Vi ser att CALL har uppdaterat ra vilken innehåller återhopps-adressen, men den anropade funktionen har inte sparat undan ra på det nyskapade stack-utrymmet, vilket main gjorde då den blev anropad, vilket är en smula konstigt.
Tittar vi på hur funktionen returnerar så gjör den det med instruktionen ”ret”, vilken gör att ra överförs till PC.
Vidare kan vi se att argumentet till delay_function som lades i r4 lagras på stacken.