

TDT4160 - Datamaskiner og digitalteknikk

Øvingshefte

Institutt for datateknikk og informasjonsvitenskap



Innhold

1 Teknisk bakgrunn	5
1.1 Mikrokontrolleren EFM32 Giant Gecko	5
1.2 Støtteregistre	8
1.3 Bit-behandling	10
1.4 Programmeringsspråket C	14
1.5 Assembly	23
2 EFM32 Giant Gecko	27
2.1 GPIO	29
2.2 Interrupt	38
2.3 SysTick	43
2.4 Instruksjonssett	45
3 Utviklingsmiljø	55
3.1 Installasjon av Simplicity Studio	55
3.2 Oppsett av Simplicity Studio	56
3.3 Alternativ til Simplicity Studio: kommandolinje	61
3.4 Hvis noe går galt	63
3.5 Rammeverk for assembly	63
3.6 Rammeverk for C	65
4 Øvinger	68
4.1 Øving 1	69
4.2 Øving 2	70
4.3 Øving 3	72

Tillegg	75
A Instruksjonssett	75
B Prosessor-registre	77
C Støtteregistre	79
D Eksterne komponenter	81

Forkortelser

STK Starter Kit

I/O Input/Output

GPIO General Purpose I/O

IRQ Interrupt Request

ISR Interrupt Service Routine

PC Program Counter

SP Stack Pointer

LR Link Register

ALU Arithmetic Logic Unit

Introduksjon

Dette kompendiet fungerer som et tilskudd til pensum og inneholder praktisk og teoretisk informasjon om øvingsopplegget. Kompendiet er delt opp i fire kapitler der de to første dekker det teoretiske grunnlaget og de to siste er mer praktiske og beskriver utviklingsmiljøet og oppgavene.

I kapittel 1 dekkes en bakgrunnsteori som er mer *generelt* anvendbart for mikrokontrollerprogrammering. Her utdypes det litt om hva en mikrokontroller egentlig er, og overordnet om hvordan man programmerer en. I tillegg gis det en innføring i bit-behandling og programmeringsspråket C. Avhengig av bakgrunnen din kan deler av dette kapittelet hoppes over, men les introduksjonen til kapittelet før du bestemmer deg for hva du trenger å lese.

Kapittel 2 er på mange måter kjernen av kompendiet. Her skal vi se konkret på mikrokontrolleren og hvordan vi styrer dens komponenter. Vi skal også studere instruksjonssettet og lære hvordan vi bruker det til å lage programmer.

Kapittel 3 gir en innføring til utviklingsmiljøet. Her finnes en steg-før-steg liste som beskriver alt som skal til for å laste opp og kjøre programmer på mikrokontrolleren. I tillegg beskrives rammeverket som deles ut.

Kapittel 4 inneholder oppgavetekstene til alle øvingene i øvingsopplegget.

Kapittel 1

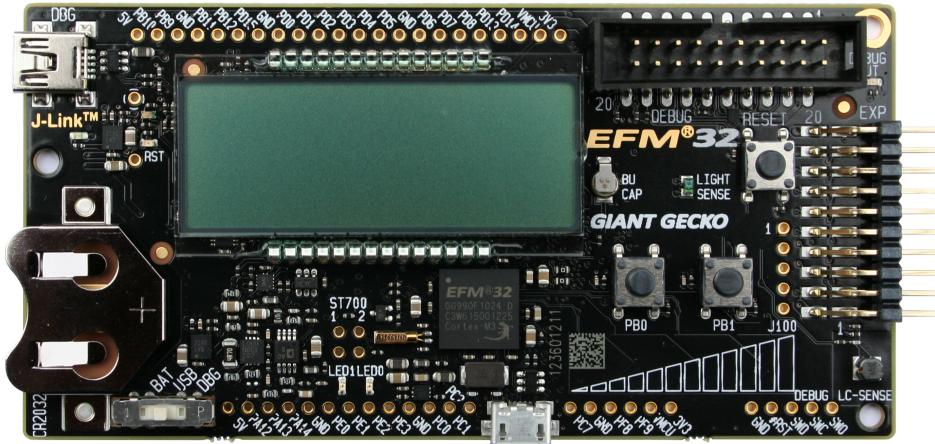
Teknisk bakgrunn

Dette kapittelet gir en introduksjon til konsepter som er relevant for mikrokontrollere generelt, samt noen overordnede detaljer om systemet vi skal jobbe med, en EFM32 Giant Gecko mikrokontroller. Seksjon 1.1 introduserer mikrokontrolleren og går raskt innom hvordan den er bygget opp. Seksjon 1.2 gir en introduksjon til det overordnede prinsippet bak maskinvareinteraksjon med mikrokontrollere, støtteregistrene. Seksjon 1.3 gir en innføring i bit-behandling, og hvordan det kan brukes til å manipulere støtteregistre. Deretter gis en introduksjon til programmeringsspråket C i seksjon 1.4, og til slutt assembly i seksjon 1.5.

Det meste av kapitlet dekker generell bakgrunnsinfo, men seksjon 1.2.1 og 1.3.1 definerer noen konvensjoner for resten av kompendiet som er spesielt viktig å få med seg.

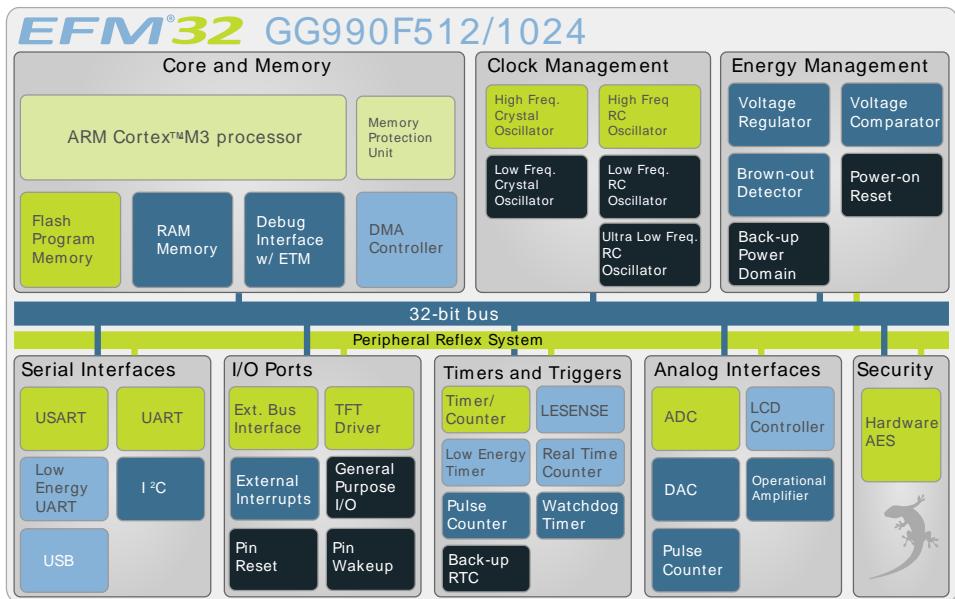
1.1 Mikrokontrolleren EFM32 Giant Gecko

I dette øvingsopplegget skal vi jobbe med en såkalt mikrokontroller. En mikrokontroller er en liten chip som inneholder en prosessor, minne, I/O-porter o.l. Det er altså en fullt fungerende datamaskin samlet inni en chip på størrelse med en fingernegl. Mikrokontrollere er svært billige og energieffektive og brukes i det meste av elektronikk som for eksempel mikrobølgeovner, barbermaskiner, TV-er, mobiltelefoner, biler osv.



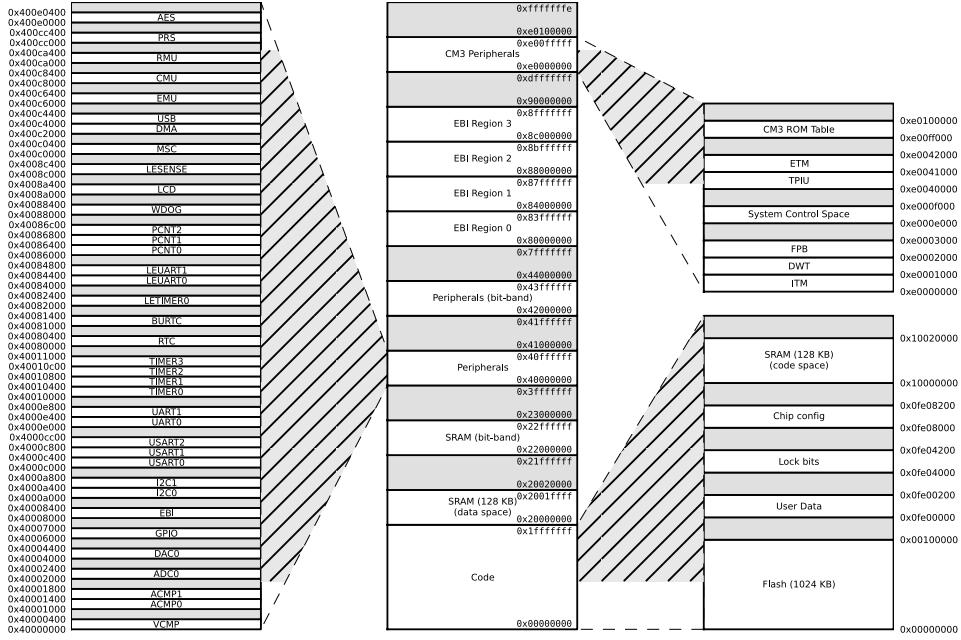
Figur 1.1: STK3700

Maskinen vi skal jobbe med er en Silicon Labs Giant Gecko mikrokontroller med en ARM Cortex-M3-prosessor. Selv om mikrokontrolleren teknisk sett er en fullverdig datamaskin kan den ikke brukes til så veldig mye interessant uten perifere enheter, og vi skal derfor bruke et Starter Kit (STK). Et STK utgjør en praktisk fullverdig maskin og består av mikrokontrolleren, kretskortet som mikrokontrolleren er loddet på og en rekke perifere enheter og koblinger. I dette øvingsopplegget skal vi bruke et Silicon Labs STK3700 som heretter omtales som *kit-et*. Kit-et syr sammen mikrokontrolleren med blant annet knapper, LED, LCD-skjerm og en J-Link-modul som vi kan bruke for å enkelt legge inn programmene våre. STK3700 er avbildet i figur 1.1. Mikrokontrolleren er den svarte chippen som sitter rett under LCD-skjermen.



Figur 1.2: EFM32 Giant Gecko

Figur 1.2 viser en oversikt over komponentene i mikrokontrolleren. Figuren er gruppert i forskjellige deler som for eksempel *Core and Memory* øverst til venstre som blant annet inneholder Cortex-M3-prosessoren. Clock management i midten øverst inneholder en rekke forskjellige klokker. Det høye antallet ulike klokker er nødvendig for å holde det spesielt lave energiforbrukskretsen som Silicon Labs spesialiserer seg på. Nummer to fra venstre nederst er *I/O Ports* som blant annet inneholder General Purpose I/O (GPIO) som vi senere skal se er en svært sentral komponent i øvingsopplegget.



Figur 1.3: Giant Gecko memory map

Figur 1.3 viser minnemappingen til mikrokontrolleren. Observér plaseringen til GPIO mellom adresse 40006000 og 40007000 (til venstre, nær bunnen). Dette betyr at at GPIO er et minneområde som vi kan lese og skrive til. Dette konseptet utdypes i seksjon 1.2.

1.2 Støtteregistre

For å utføre spesielle oppgaver på en mikrokontroller bruker man ofte noe som kalles støtteregistre, heretter omtalt som registre. Disse registrene er en del av prosessorens adresseområde og må ikke forveksles med prosessorens interne registre. Et register brukes til å utføre en oppgave som for eksempel å sette en tilstand eller lese en spesiell verdi. Å skrive til eller lese fra et register kan sammenlignes med et funksjonskall der å skrive til registeret blir som å kalle på en void funksjon med en verdi som input, mens å lese fra registeret blir som å kalle på en eller annen get-funksjon. Detaljene i maskinvaren, der verdiene som sendes til registeret medfører en eller annen maskinvare-funksjon, er abstrahert bort for programmereren. Eksempler på slike registre er GPIO-registrene som brukes for å lese og skrive til eksterne komponenter. Bruken av slike registre er på mange måter kjernen av mikrokontroller-programmering,

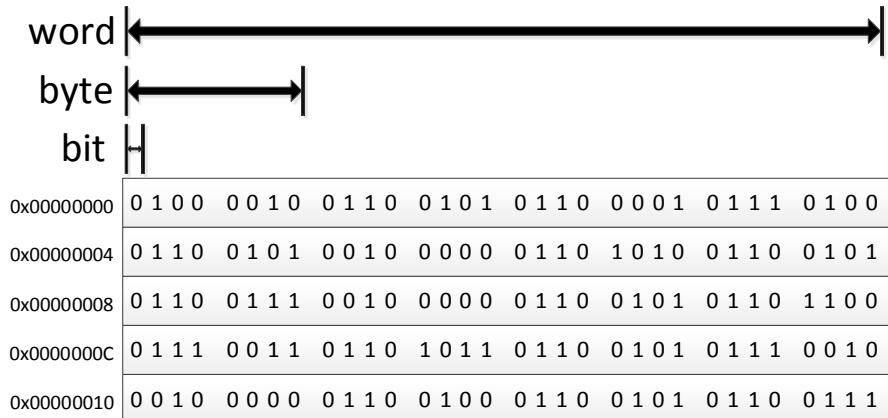
og konkrete eksempler på dette finnes i seksjon 2.1.

1.2.1 Enheter

For å aksessere et register må vi regne ut registerets adresse, og da er det viktig å holde tunga rett i munnen med tanke på enheter. Et register består av 32 bits, mens et byte er som kjent 8 bits. Datamaskiner regner stort sett alltid med bytes når man aksesserer minnet. Siden 32 bits er fire bytes, må hver registeradresse da være et multiplum av fire. Tabellen som følger viser enhetene som blir brukt i denne seksjonen og deres betydning.

Enhett	Navn	Antall bits	Antall bytes	Bruksområde
bit	bit	1	1/8	GPIO-pinner
B	byte	8	1	Minneadresser
w	word	32	4	Registre

Hvis vi skal for eksempel oversette $2w$ til bytes (B), kan vi bruke det faktum at $w = 4B$. Vi erstatter rett og slett w med $4B$, og får følgende: $2w = 2(4B) = 8B$. I bits blir $2w = 2(32\text{ bit}) = 64\text{ bit}$, siden $w = 32\text{ bit}$. Figur 1.4 visualiserer de forskjellige enhetene plassert i minnet. Legg merke til at adressene til hvert ord (her representert heksadesimalt) er et multiplum av fire.



Figur 1.4: Enheter visualisert i minnet

1.3 Bit-behandling

Bit-behandling er ofte svært nyttig når man jobber så tett på maskinvaren. Denne seksjonen er ment for å gi litt tips og triks for å beherske bit-behandling. Standard logiske operasjoner inkluderer AND, OR, EOR (XOR) og NOT¹. I en prosessor er disse logiske operasjonene tilgjengelige som instruksjoner, men som *bitwise* logiske operasjoner. Når en logisk operasjon er *bitwise* betyr det at operasjonen utføres mellom to ord på alle bit-ene i ordene som er på samme indeks. En god måte å visualisere dette på er å skrive ut ordene binært, plassere dem rett ovenfor hverandre, og deretter utføre operasjonen mellom alle bit-verdiene som står rett ovenfor hverandre. I seksjon 1.3.3 til og med 1.3.5 skal vi bruke denne metoden til å løse et par eksemploppgaver. Men først skal vi i seksjon 1.3.1 og 1.3.2 introdusere *binær indeksering*, og en måte å produsere bit-indekser på.

1.3.1 Indeksering av binære ord

Binære verdier som kjent gjort opp av bits. For å forenkle diskusjonen må vi kunne indeksere hvert individuelle bit. Vi kaller den minst signifikante bit-en (den lengst til høyre) for bit 0. De neste er bit 1, bit 2, osv. Tabellen under viser dette på en 8-biters binær verdi 11110000.

Indeks	7	6	5	4	3	2	1	0
Verdi	1	1	1	1	0	0	0	0

1.3.2 Produsere bitx

Vi kaller for eksempel bit 5 for binært 100000 som er lik desimalt 32, men sammenhengen mellom bit 5 og tallet 32 er ikke helt klar. Det er derfor nødvendig med en fast prosedyre for å produsere bitx på generelt basis. Dette kan gjøres med *bitshift*. Et bitshift flytter et binært tall et gitt antall plasser til høyre eller venstre. Hvis vi for eksempel shift-er det binære tallet 000000001 én plass til venstre får vi 00000010. Shift-er vi det én plass til høyre forsvinner bit 0 ut på høyresiden og vi ender opp med 0. Observér at ved left shift blir bit-en helt til høyre fylt inn med 0, og bit-en helt til venstre vil forsvinne siden det ikke er plass til den i den nye verdien. Tilsvarende er det for right shift, men da motsatt. For å produsere bitx på generelt grunnlag kan vi shifte tallet 1, x plasser til venstre. Eksempelet under viser hvordan

¹Boolsk algebra er beskrevet i læreboka på side 138.

bit 5 kan produseres.

Ord	Verdi
1	00000001
bit-indeks	5
1 Left Shift <i>bit-indeks</i>	00010000

1.3.3 Ta bort deler av et register

Vi antar at vi har et 32 bits register, der vi skal fjerne de 4 minst signifikante bit-ene. Det er viktig at vi ikke rører noen av de andre bit-ene. For å få til dette skal vi benytte oss av egenskapene til en bitwise AND.

Ord	Verdi
register	1010101010101010101010101010101010
maske	11111111111111111111111111111111110000
<i>register AND maske</i>	1010101010101010101010101010100000

Her ser vi at alt vi trengte å gjøre var å ANDe med en maske. Observér at på posisjonen til alle bit-ene som vi ville fjerne var verdien 0 i masken. På posisjonene vi ville beholde hadde masken verdien 1. Hver bit i registeret blir ANDet med biten i masken som er på samme posisjon rett under.

1.3.4 Sette inn en bit i et register

Vi bruker det samme registeret som i forrige eksempel. Nå skal vi sette to bits inn i registeret, uten å endre noen av de andre bitene som er der fra før. Vi vil sette inn bit 0 og bit 3. For å få til dette skal vi benytte oss av egenskapene til en bitwise OR.

Ord	Verdi
register	1010101010101010101010101010101010
input	000000000000000000000000000000001001
<i>register OR input</i>	1010101010101010101010101010101011

Når vi ORer inn disse bit-ene i registeret ser vi at disse blir tvunget inn i registeret uten at noen av de andre bit-ene blir påvirket. Observér at bit 3 allerede var satt i registeret. Siden OR gir 1 som output når minst en av bitene er 1, spiller det ingen rolle at den var 1 fra før.

1.3.5 Flippe et bit

Se for deg at det finnes en bit i registeret som vi vil flippe. Vi vet ikke hva bit-en er, men vi vil sette den til det motsatte av hva den er nå. Som eksempel skal vi flippe bit 3 i registeret. For å få til dette må vi benytte oss av egenskapene til Exclusive OR (EOR).

Ord	Verdi
register	10101010101010101010101010101010
bit3	000000000000000000000000000000001000
<i>register EOR bit3</i>	101010101010101010101010100010
<i>(register EOR bit3) EOR bit3</i>	10101010101010101010101010101010

Vi ser her at ved å gjøre et Exclusive OR på registeret og bit 3 får vi endret bit 3 i registeret fra 1 til 0. Enda viktigere ser vi at ved å gjøre akkurat det samme på det nye registeret (*register EOR bit3*), får vi satt den tilbake fra 0 til 1 igjen. Dette er fordi EOR har den egenskapen at når den ene inputen (maske-biten) er 0, beholdes alltid den andre inputen (register-biten), akkurat som med OR. Men når den ene inputen er 1 (maske-biten), vil resultatet bli det motsatte av den andre inputen (register-biten).

1.3.6 Oppgaver

Under følger noen enkle bit-operasjon-oppgaver som kan hjelpe mye til med forståelsen. Fyll inn de tomme cellene.

Ord	Verdi
register	0001010010011000
<i>register Left Shift 3</i>	
<i>register Right Shift 1</i>	

Tabell 1.1: Oppgave 1

Ord	Verdi
register	1000 1100 0011 0101
maske	0100 0100 1010 0000
<i>register AND maske</i>	
<i>register OR maske</i>	
<i>register EOR maske</i>	

Tabell 1.2: Oppgave 2

Ord	Verdi
register	1010 1101 1111 1000
maske	1100 1010 0001 1111
<i>register AND maske</i>	
<i>register OR maske</i>	
<i>register EOR maske</i>	

Tabell 1.3: Oppgave 3

1.3.7 Løsninger

Under følger løsninger på oppgavene over.

Ord	Verdi
register	0001010010011000
<i>register Left Shift 3</i>	1010010011000000
<i>register Right Shift 1</i>	0000101001001100

Tabell 1.4: Løsning oppgave 1

Ord	Verdi
register	1000 1100 0011 0101
maske	0100 0100 1010 0000
<i>register AND maske</i>	0000 0100 0010 0000
<i>register OR maske</i>	1100 1100 1011 0101
<i>register EOR maske</i>	1100 1000 1001 0101

Tabell 1.5: Løsning oppgave 2

Ord	Verdi
register	1010 1101 1111 1000
maske	1100 1010 0001 1111
<i>register AND maske</i>	1000 1000 0001 1000
<i>register OR maske</i>	1110 1111 1111 1111
<i>register EOR maske</i>	0110 0111 1110 0111

Tabell 1.6: Løsning oppgave 3

1.4 Programmeringsspråket C

C er et språk som er svært vanlig å bruke når man programmerer mikrokontrollere. Noen av grunnene til dette er at det, samtidig som å være et høynivåspråk, tillater programmereren å jobbe tett på maskinvaren. Spesielt støtten for minnehåndtering og bit-behandling er viktig her. Denne seksjonen gir en kort introduksjon til C. Dersom du er interessert i å lære mer anbefales andre utgave av “The C Programming Language” av Brian Kernighan og Dennis Ritchie.

1.4.1 Inkludering av bibliotek

For å inkludere filer brukes preprocessor-kommandoen `#include`. Filene som inkluderes kan inneholde funksjoner og definisjoner, der en definisjon kan for eksempel definere en datatype eller en konstantverdi. Eksempelvis må man inkludere filen `stdio.h` for å få tilgang på funksjonen `printf`. Se eksempel under for bruk.

```
#include <stdio.h>
int main(void) {
    printf("Hello, world!");
    return 0;
}
```

Med `#include` holder det å skrive selve filnavnet, da hele pathen til disse filene skal oppgis først ved kompilering. I rammeverket som deles ut blir alle nødvendige paths automatisk gitt til kompilatoren. Som vist i eksempelet over er det, i motsetning til i Java, ikke nødvendig å angi hvilken fil funksjonen er deklarert i når man kaller på en ekstern funksjon.

1.4.2 Variabler

I motsetning til Python og i likhet med Java må alle variabler deklarerdes med en datatype. Eksempler på datatyper er `int` (heltall), `double` (flyttall) og `char` (positivt heltall som ofte representerer en bokstav). I tillegg til å betegne datatypen på en variabel kan man angi en eller flere *specifiers* foran datatypen. Disse kan spesifisere størrelsen på variabelen (hvor store tall den skal ha plass til), hvorvidt den er `unsigned` eller har støtte for negative tall, eller om kompilatoren skal tillates å optimalisere aksessen til variabelen (mer om dette senere). For eksempel kan en variabel som skal kunne holde store positive heltall deklarerdes som

```
unsigned long int tall;
```

Datatypene **int**, **double** og **char** er støttet ut av boksen og krever ingen tillegg til programmet ditt, men ulempen med disse er at størrelsene varierer fra platform til platform. Det er derfor vanskelig å være helt sikker på hvor stor en slik variabel er. For de tilfellene hvor det er nødvendig å vite nøyaktig hvor stor variablene er finnes det alternativer, C støtter nemlig muligheten til å definere egne datatyper. I filen **stdint.h** er det definert forskjellige varianter av **int** der størrelsene er deterministiske, se tabell under.

Datatype	Størrelse	Fortegn
int8_t	8 bit	Ja
int16_t	16 bit	Ja
int32_t	32 bit	Ja
uint8_t	8 bit	Nei
uint16_t	16 bit	Nei
uint32_t	32 bit	Nei

Tabell 1.7: Datatyper i stdint.h

Fortegn

Om tallet har fortagn eller ikke spesifiserer kun hvordan verdien skal tolkes og påvirker ikke hvordan verdien lagres i minnet. Hvis en variabel har fortagn sier vi at den er **signed**. En signed datatype, som for eksempel **int8_t**, gjør at verdien blir tolket i toers komplement. Kjapt forklart er toers komplement at et tall tolkes som negativt dersom den mest signifikante bit-en er 1. I dette tilfellet kan like mange tall representeres, men halvparten av disse er negative. For eksempel kan tallene 0 til 255 representeres av en 8-biters **unsigned** variabel, mens tallene -128 til 127 kan representeres når de samme 8 bitene tolkes med toers komplement i en **signed** variabel.

Boolske verdier

Boolske verdier og datatyper er ikke støttet ut av boksen i C. I stedet tolker kontrollstrukturene boolske verdier på følgende måte: Tallet 0 tilsvarer false, mens alle andre tall tilsvarer true. Det betyr at der hvor man vanligvis ville skrevet **true** i mange andre språk, holder det å skrive for eksempel 1 i C. Datatypen man bruker for å holde en boolsk verdi spiller ingen rolle da alle datatyper kan representer 1 og 0. For de som synes dette er litt kryptisk og

lite intuitivt finnes det en fil `stdbool.h` som definerer datatypen `bool`, og verdiene `true` og `false` som er definert som henholdsvis 1 og 0.

Boolske utrykk bruker følgende operatorer for å sammenligne verdier:

Operator	Beskrivelse
<	Mindre enn
>	Større enn
<=	Mindre enn eller lik
>=	Større enn eller lik
==	Lik
	Boolsk OR
&&	Boolsk AND
!	Boolsk NOT

Tallsystem

Med mindre du oppgir noe annet, tolker C alle konstantverdier du skriver inn som vanlige desimale tall (tall fra titallsystemet). Ved å plassere `0x` eller `0b` rett foran tallet vil det bli tolket som respektivt heksadesimalt eller binært². Eksempelvis tolkes tallet `0xFFFF` som et heksadesimalt tall, `0b1111` som et binært tall, og `9999` som et desimalt tall.

Kompilatoroptimalisering

Kompilatoren vil som regel forsøke å optimalisere aksessen til en variabel. For eksempel hvis du har en variabel som aksesseres ofte, vil kompilatoren kanskje cache denne variablene på en eller annen måte. Dette vil medføre at å skrive til variablene ikke egentlig skriver til minnet. I de aller fleste tilfeller er dette veldig nyttig: Minneaksess er svært tregt, og caching øker ytelsen dramatisk. Men noen ganger er det viktig at verdien faktisk skrives til sin minneadresse umiddelbart.

Hvis en variabel for eksempel representerer et GPIO-register, er det viktig at verdien ikke caches. Da vil verdien kanskje ligge i cachen i lang tid før den skrives til maskinvaren, om den blir det i det hele tatt. Et annet tilfelle er hvis et interrupt endrer verdien til en global variabel som andre deler av koden overvåker. For eksempel kan et interrupt fra en knapp sette en boolsk variabel til true. Med caching vil kanskje ikke dette synes for resten av koden.

²Binære tall prefikset med `0b` støttes av krysskompilatoren for dette øvingsopplegget, men er ikke definert i ANSI C (standard C). Det vil altså ikke fungere med de fleste andre C-kompilatorer.

I slike tilfeller bruker vi specifieren **volatile**. Hvis en variabel er deklarert som volatile, vil kompilatoren *ikke* prøve å optimalisere bruken av den. Dette gjøres på følgende måte:

```
volatile int foo;
```

Pekere

En vanlig variabel representerer en minneadresse. Når vi endrer på variablene, endres *innholdet* på minneadressen. Men en peker er en variabel som *peker* på en minneadresse. Når vi endrer på verdien i en peker, endrer vi på *adressen* til pekeren, altså hvor pekeren peker. For å forklare hvordan dette fungerer er det enklest å se på et eksempel:

```
1 int tall;
2 int* peker;
3 peker = &tall;
4 *peker = 10;
```

Her er to nye tegn introdusert, '*' og '&'. Når en variabel deklarerется med '*' (linje 2), betyr det at den er en peker. Datatypen til en peker betegner datatypen til variablene den skal peke på. Når en variabel er deklarert som en peker vil det å forandre på den forandre på *hvor* den peker (linje 3). Når en vanlig variabel aksesseres med en '&' betyr det *adressen* til variablen. På linje 3 tar vi altså adressen til den vanlige variablen **tall** og legger den i pekeren **peker**. På linje 4 aksesseres pekeren med '*'. Når en peker aksesseres med '*', betyr det at vi *derefererer* den (eng. dereferencing), som betyr å aksessere innholdet til minneadressen som pekes på. Siden **peker** peker på minneadressen som **tall** representerer, vil innholdet i **tall** være verdien 10 etter at linje 4 har kjørt.

Her er selve minneadressene abstrahert bort: Kompilatoren vil være ansvarlig for å velge en minneadresse til **tall**. Det er også mulig å lage en peker som peker til en minneadresse vi velger selv. Kombinert med dereferering er dette en hendig måte å lese og skrive til spesifikke minneposisjoner, for eksmepel et GPIO-register. En peker kan settes til en konstant minneadresse ved normal tilordning. Husk at ved tilordning (for eksempel `int x = 10`), kreves det at verdien på høyresiden av likhetstegnet må ha samme datatype som variablene som tilordnes. Konstanten må dermed også være en peker. Hvis vi vil skrive en 32 bits verdi til minneadressen `0x20000000`, kan vi caste adressen til en 32 bits datatype, f.eks. `uint32_t*`.

Eksempelet under skriver verdien 123 til minneadressen 0x20000000:³

```
volatile uint32_t* minnepeker = (uint32_t*) 0x20000000;
*minnepeker = 123;
```

Arrays

Et *array* er en liste med verdier som ligger etter hverandre i minnet. Et array deklarerется og aksesserется på følgende måte:⁴

```
1 int tall[3];
2 tall[0] = 8;
3 tall[1] = 16;
4 tall[2] = 32;
```

Husk fra seksjon 1.2.1 at minnet adresseres per byte. Adressene til elementene i et array av 32 bits integere, som eksempelet over, er derfor alltid delbare på 4. Tabellen under viser dette arrayet der minneadressene øker med 4 for hvert element:

Minneadresse	Verdi
0	0000 0000 0000 0000 0000 0000 0000 1000
4	0000 0000 0000 0000 0000 0000 0001 0000
8	0000 0000 0000 0000 0000 0000 0010 0000

Siden prosessoren addresserer minnet i bytes, blir det her nødvendig med en oversetting for at vi skal kunne adressere hvert element uten at programmereren må manuelt ta hensyn til størrelsen. Når vi aksesserer et element av et array, for eksempel `tall[2]`, vil adressen til elementet måtte være adressen til arrayet pluss indeksen ganger størrelsen på datatypen angitt i bytes. I dette eksempelet er indeksen 2, og størrelsen på datatypen 4 bytes. Den fysiske adressen blir dermed $2 * 4 + \text{array}$. Merk at C gjør denne utregningen for oss. Hver gang vi offsetter et array vil komplilatoren automatisk ta hensyn til størrelsen på datatypen. Det samme gjelder også for pekere.

³Legg merke til specifiseringen `volatile`. Denne er lagt på for å være sikker på at verdien faktisk blir skrevet til minnet.

⁴Hva ville skjedd dersom vi prøvde å aksessere `tall[3]`? Er du dreven i Java eller Python frykter du kanskje en `ArrayIndexOutOfBoundsException` eller `IndexError`, men i C er det ingenting som automatisk passer på deg på denne måten. Aksesserer du noe utenfor arrayet får du enkelt og greit tilbake det som skulle finnes i minnet på den adressen. Kan du se for deg hvordan en hacker kunne utnyttet dette?

Structs

En struct er en gruppering av flere variabler. For å sammenligne med Java er en struct som et objekt uten metoder. For eksempel kan en struct som representerer en vektor med to elementer `x` og `y` defineres på følgende måte:

```
struct vektor {  
    int x;  
    int y;  
};
```

Eksempel på deklarasjon og aksess av en “instans” av en slik struct følger under:

```
1 | struct vektor v;  
2 | v.x = 3;  
3 | v.y = 4;
```

Datatypen er altså `struct vektor` og her kalte vi instansen `v`. Første linje deklarerer `v`. Elementene i structen aksesseres, på samme måte som objektfelt i Java, med punktum (linje 2 og 3).

For å unngå å ha “struct” som en del av datatype-navnet kan vi lage et type-alias med kommandoen `typedef` i det vi definerer structen. Definisjonen blir da i stedet:

```
typedef struct {  
    int x;  
    int y;  
} vektor;
```

Hvis vi har en peker til en strukt, må vi dereferere pekeren før vi aksesserer elementer. Det kan gjøres på to måter:

```
1 | vektor v;  
2 | vektor* vp = &v;  
3 | (*vp).x = 3;  
4 | vp->x = 3;
```

Pekeren deklarerer på linje 2 og peker på instansen som ble deklarert på linje 1. På linje 3 derefereres pekeren på normal måte og deretter aksesseres elementet fra den derefererte pekeren. Linje 4 viser en snarvei som oppnår nøyaktig det samme på en litt penere måte: Pilen derefererer pekeren til structen *og* aksesserer elementet.

1.4.3 Kontrollstrukturer

Denne seksjonen lister opp kontrollstrukturene som finnes i C med noen enkle eksempler. Svært mange moderne programmeringsspråk, blant andre Java, har arvet sine kontrollstrukturer fra C, og mange av disse er derfor antakeligvis kjente for deg. Hvis du er kjent med kontrollstrukturene i Java kan denne seksjonen hoppes over. Eksempler for hver kontrollstruktur er listet opp under. Etter hvert eksempel følger en beskrivelse av hvordan koden vil bli utført. Ved all bruk av variabler kan det antas at variablen er deklarert tidligere i programmet.

```
if (tall == 2) {  
    foo();  
}
```

Hvis `tall` er lik 2, kall funksjonen `foo`.

```
while (i > 0) {  
    i = i-1;  
}
```

Sjekk først om `i` er høyere enn 0. Hvis ja, trekk 1 fra `i`. Gjenta til `i` ikke lenger er høyere enn 0.

```
do {  
    i = i-1;  
} while (i > 0);
```

Trekk først 1 fra `i`, deretter gjenta hvis `i` er høyere enn 0.

```
for (i=0; i<10; i++) {  
    foo();  
}
```

1. Utfør første operasjon i for-løkken (`i=0`)

2. Test kondisjonen i andre del av for-løkken (`i<10`) Hvis true, utfør kroppen til for-løkken (`foo();`), hvis false avslutt for-løkken.
 3. Utfør tredje operasjon i for-løkken (`i++`).
 4. Gå til steg 2.
-

```
switch(tilstand) {
    case 1:
        foo();
        break;
    case 2:
        bar();
        break;
    default:
        tilstand = -1;
}
```

Hvis variablen `tilstand` er lik 1, kall på funksjonen `foo`, deretter avslutt switch-strukturen. Hvis `tilstand` er lik 2 kall på funksjonen `bar`, deretter avslutt switch-strukturen. Hvis `tilstand` er hverken 1 eller 2, sett `tilstand` til -1.

1.4.4 Bit-behandling

Bit-behandling, som introdusert i seksjon 1.3, utføres i C ved bruk av operatorer, på lik måte som f.eks. + og -. C-operatorene for bit-behandling er vist i tabellen under.

Operasjon	Syntaks	Beskrivelse
Left shift	a «b	a bitshiftes b plasser til venstre.
Right shift	a »b	a bitshiftes b plasser til høyre.
Bitwise AND	a & b	a AND b.
Bitwise OR	a b	a OR b.
Bitwise EOR	a ^ b	a EOR b.
NOT	~a	NOT a

Eksempelvis kan bit0 og bit1 ORes inn i en variabel på følgende måte:

```
uint32_t foo = 0b10000000;
foo = foo | 0b01;
foo = foo | 0b10
```

1.4.5 Funksjoner

Funksjoner i C er svært likt statiske metoder i Java. Bortsett fra tilgangsmodifikatorene (public, private) er det stort sett likt. En funksjon kan returnere enten én eller ingen verdier, der datatypen er bestemt i fuksjonsdeklarasjonen. En funksjon kan også ta inn ingen, én eller flere verdier, der datatypen(e) er bestemt i funksjonsdeklarasjonen. Følgende eksempel demonstrerer dette:

```
1 int add(int a, int b) {  
2     return a + b;  
3 }  
4  
5 int main(void) {  
6     int sum = add(1,2);  
7     return 0;  
8 }
```

Første linje i en funksjonsdeklarasjon har følgende tre deler: Datatype for returverdi, funksjonsnavn og deklarasjon av input-variabler. I eksempelet på linje 1 er returverdien `int`, funksjonsnavnet er `add` og det er to input `a` og `b` som begge skal ha datatypen `int`.

Main er, akkurat som i Java, funksjonen som kjører når programmet starter. Eksempelet på linje 5 adderer to tall ved hjelp av funksjonen `add`, og deretter returnerer den 0 for å signalisere suksess. Observér at input til main er `void`. `void` er et spesielt nøkkelord som betyr *ingenting*. Eksempelvis kan `void` settes som returverdi i en funksjonsdeklarasjon hvis funksjonen ikke skal returnere noe.

Rekkefølgen til funksjoner er viktig. Et funksjonskall må alltid skje *etter* funksjonen den kaller på er deklarert. Det vil si at kompilatoren vil lese nedover linje for linje, og hvis den kommer til et funksjonskall til en funksjon den ikke har sett ennå gir den en feilmelding. I C må vi se på funksjoner som variabler. Den enkleste løsningen på dette problemet er å deklarere funksjonene i begynnelsen av fila, uten å faktisk implementere dem. Da vil man kunne kalle på dem i vilkårlig rekkefølge. Slike deklarasjoner er kjent som prototyper, og tilsvarer å deklarere en variabel uten å gi den en verdi. En prototype ser helt lik ut som en funksjon, bortsett fra at selve implementasjonen er erstattet med et semikolon, for eksempel ville prototypen til `add-funksjonen` over sett slik ut:

```
int add(int a, int b);
```

1.5 Assembly

I denne seksjonen gis en generell innføring i assembly, uten å gå spesielt i dybden. Her ligger fokuset mer på hva assembly egentlig er, og detaljene introduseres først i seksjon 2.4. Denne seksjonen kan trygt hoppes over dersom du vet hva assembly er fra før.

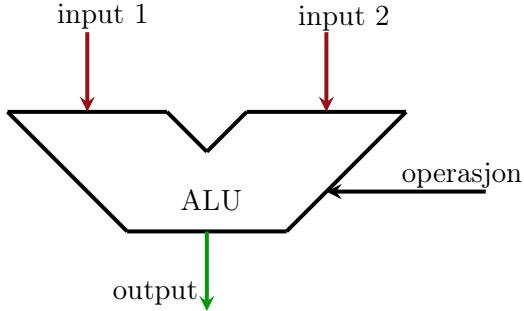
En prosessor utfører binære instruksjoner. Assembly er en tekstlig representasjon av slike instruksjoner. Ordet “assembly” er ikke spesielt beskrivende, ordet kommer fra assembleren, programmet som oversetter assembly til binær maskinkode. Et mer beskrivende (og litt mindre catchy) navn kunne vært “prosessorinstruksjoner i tekstlig form” eller lignende. Assemblykode er altså en mengde prosessorinstruksjoner som sammen utgjør et program.

For å forstå hvorfor vi trenger assembly, kan vi se på en liten kodesekvens fra et høynivåspråk og (feilaktig) anta at prosessoren utfører den direkte.

```
for (i=0; i<10; i++) {  
    a[i] = b[i] + c[i] * d[i];  
}
```

Hvordan skal prosessoren utføre denne kodesekvensen? Tar den hele sekvensen inn på én gang, eller bryter den det opp i flere biter? I såfall, hvordan brytes det opp? Hvordan vet prosessoren når den skal teste kondisjonen og hva den skal gjøre om den utgjør false? Hvordan vet prosessoren hvor stort hvert element i arrayet er, og dermed hvordan den skal tolke indeksen? Hvordan vet prosessoren om den skal addere eller multiplisere først? Svaret på alle disse spørsmålene avgjøres lenge før programmet kjøres på en prosessor. Prosessorer er ikke designet for å kunne ta slike avgjørelser, og om de måtte det ville det gått kraftig utover ytelsen til programmet. I stedet tar kompilatoren av seg alle disse spørsmålene og kan heller produsere korte og enkle prosessorinstruksjoner som prosessoren kan helt entydig forstå. Dette kommer av prinsippet om abstraksjon: Jo lavere ned i abstraksjonslaget, jo enklere er hver individuelle handling. Prosessoren utfører mange små og enkle operasjoner i stedet for få store og kompliserte operasjoner.

At prosessoren er så fryktelig enkel er ikke helt riktig: Moderne prosessorer har mange svært kompliserte teknikker for å presse ytelsen oppover. Men hvis vi reduserer selv slike prosessorer til deres mest grunnleggende virkemåte ser vi at måten den håndterer data på egentlig er svært enkel. Modellen vi sitter igjen med etter å ignorere denne kompleksiteten er akkurat det vi trenger for å forstå assembly.



Figur 1.5: Arithmetic Logic Unit (ALU)

1.5.1 Prosessorens grunnleggende virkemåte

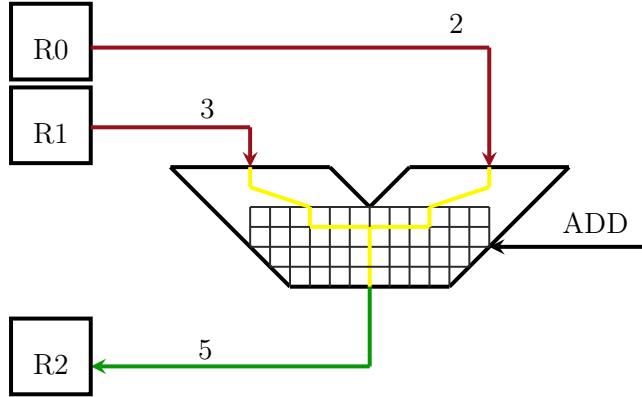
Arithmetic Logic Unit (ALU) er den delen av prosessoren som gjør beregningene til instruksjoner. Den kan ta inn maks to verdier som input, gi én verdi som output, og konfigureres til hvilken operasjon den skal gjøre på inputen. Figur 1.5 viser en enkel ALU. Operasjonen kan for eksempel være addisjon, og output-verdien vil bli summen av de to input-verdiene. Hvert sett med input, output og operasjon kalles en instruksjon. Prosessoren utfører programmer ved å kjøre slike instruksjoner. Instruksjoner som dette er som regel på det man kaller tre-adresse-form:

ADD z, x, y

Denne instruksjonen betyr at ALU-en skal konfigureres til å gjøre addisjon, inputen skal komme fra variabel **x** og **y**, og outputen skal legges i variabel **z**. Slike variabler er ikke vanlige variabler i minnet, men prosessorregistre som sitter tett inntil ALU-en, i denne seksjonen omtalt som registre. Noen slike registre har spesielle bruksområder, som for eksempel programtelleren som inneholder adressen til neste instruksjon som skal utføres. Andre registre er det man kaller “general purpose”, registre man kan bruke til hva man vil. General purpose-registre har normalt navn som R0, R1, R2, osv. For at eksempekkoden over skal kunne forstås av prosessoren, bruker vi registre som input og output:

ADD R2, R0, R1

Figur 1.6 viser hvordan ALU-en ville sett ut i det instruksjonen over utføres, dersom R0 inneholder 2 og R1 inneholder 3. De aktuelle registrene kobles fysisk inn til ALU-en, og ALU-en konfigureres til å lede inputen gjennom det subsettet av kretsene sine som utfører addisjon. Etter at instruksjonen er kjørt vil R2 inneholde verdien 5.



Figur 1.6: ALU-en under utføring av en ADD-instruksjon

1.5.2 Enkle prinsipper, stort bruksområde

Vi kan bygge videre på dette enkle prinsippet for å oppnå nesten alle funksjonene til en prosessor. For eksempel kan all form for **aritmetikk**, inkludert bit-behandling, utføres på denne måten. Hvis vi vil regne ut mer komplisert aritmetiske uttrykk som for eksempel $x * y + z$ kan vi først kjøre $x * y$ gjennom ALU-en med en **MULTIPLY**-operasjon, lagre resultatet i et midlertidig register, for å deretter gjøre en **ADD**-operasjon på det midlertidige registret og z . Det er også mulig å flytte innholdet av ett register til et annet, ved å bare kjøre det gjennom ALU-en uten å gjøre noe operasjon på det (eventuelt addere det med 0), og skrive det til et annet register.

Vi nevnte tidligere at programtelleren er et register. Siden programtelleren bestemmer hvilken instruksjon som skal være den neste som utføres, kan vi implementere **kontrollflyt** (if, while, for, funksjonskall, osv) ved å manipulere programtelleren. Normalt vil prosessoren automatisk øke programtelleren til å peke på den neste instruksjonen i minnet. Men hvis vi manuelt skriver en annen adresse til programteller-registeret vil instruksjonen på den adressen være den neste som utføres, og man sier at vi har “branch”-et dit.⁵ Alle prosessorer har i tillegg mulighet for å utføre såkalte kondisjonelle branches, der branchen kun blir tatt dersom en betingelse er tilfredsstilt.

I seksjon 2.4 skal vi komme tilbake til assembly og studere instruksjonssettet (“assembly-språket”) til mikrokontrolleren. Her vil vi også studere en rekke instruksjoner for å branche både kondisjonelt og ukondisjonelt, og vi

⁵ Ordet “branch” kan være noe misleddende og antyde parallelitet. Å branche betyr kun å hoppe til et annet sted i koden, koden man hoppet fra vil ikke fortsette å utføre før man eventuelt brancher tilbake senere. Andre ord for branch er “jump” og “goto”.

skal også introdusere **minneoperasjoner** som kan lese og skrive fra minneområdet.

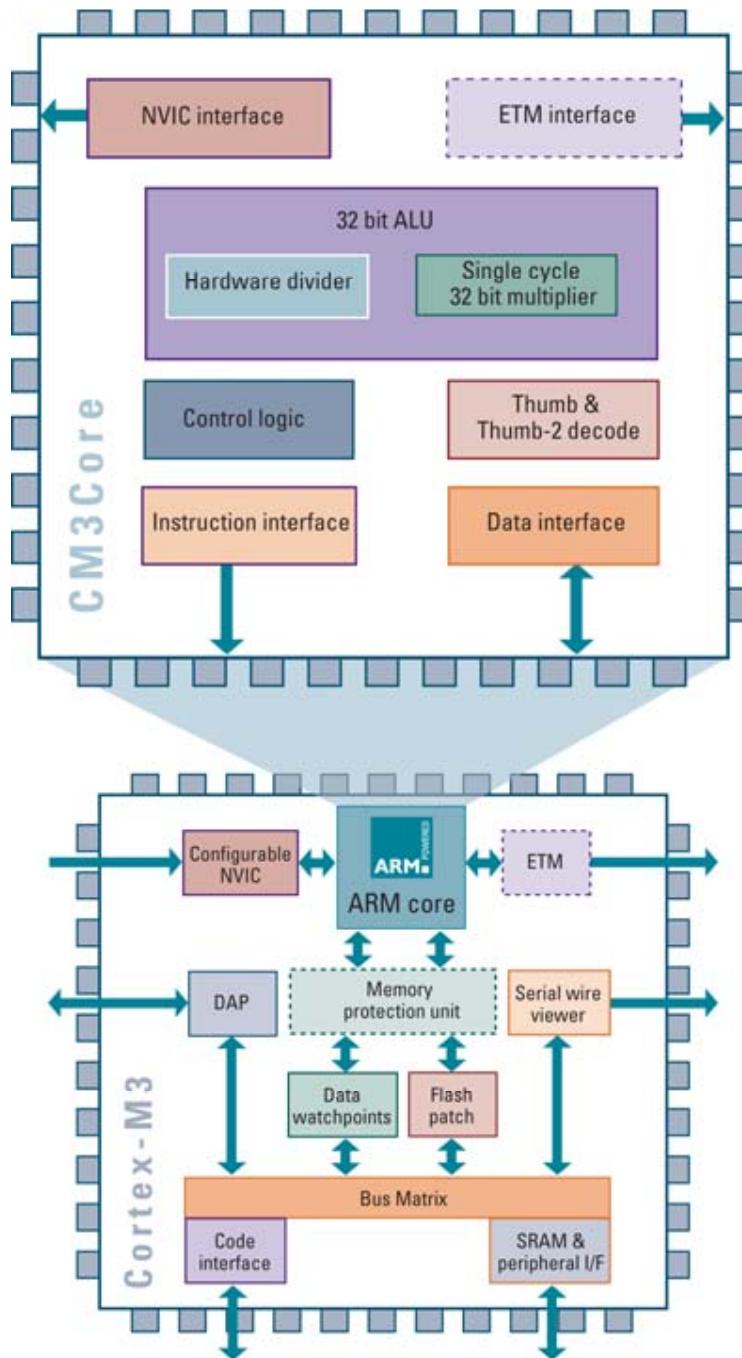
Med disse tre grunnstenene, aritmetikk, kontrollflyt og minneoperasjoner, har vi alt vi trenger for å programmere i assembly.

Kapittel 2

EFM32 Giant Gecko

Dette kapittelet gir en forklaring av alle konseptene bak mikrokontrolleren – EFM32 Giant Gecko – som er nødvendig for å fullføre øvingsopplegget. Seksjon 2.1 beskriver en av de viktigste komponentene vi skal bruke, General Purpose I/O (GPIO). Seksjon 2.2 forklarer konseptet bak interrupts, og hvordan vi kan bruke dette sammen med GPIO. Seksjon 2.3 forklarer Sys-Tick, en komponent som holder styr på tiden. Til slutt skal vi i seksjon 2.4 se på instruksjonssettet, eller “assembly-språket”, som brukes i mikrokontrollerens Cortex-M3-prosessor.

Et block-diagram over mikrokontrolleren ble vist i figur 1.2 (side 7). Prosessoren på mikrokontrollren – Cortex-M3 – er der plassert øverst til venstre. I figur 2.1 ser vi hvordan denne prosessoren ser ut. Kjernen av prosessoren (ARM core), er den som utfører alle instruksjonene. Hvis vi ser på tilkoblingene til kjernen ser vi at den kun er koblet til *Configurable NVIC*, *ETM* og *Memory protection unit*. Disse modulene brukes til henholdsvis interrupts, debugging og minneaksess. Den eneste måte kjernen kan kommunisere med resten av mikrokontrolleren (under normal kjøring) er altså gjennom minneaksess og interrupts. Denne formen for kommunikasjon utdypes i de neste seksjonene.



Figur 2.1: ARM Cortex-M3 med kjerne

2.1 GPIO

General Purpose I/O (GPIO) er et sett med støtteregistre vi skal bruke for å håndtere perifere enheter på kitet. GPIO-registrene utfører oppgaver som å lese eller skrive til forskjellige perifere enheter. For eksempel skal vi ofte bruke output-registrene (DOUT) og input-registrene (DIN). Disse brukes til henholdsvis å skrive til og lese fra perifere enheter. For å sette en verdi på et eller flere signaler trenger man bare å skrive verdien til det det riktige output-registeret og systemet tar seg av resten. Tilsvarende trenger man kun å lese fra et input-register for å se verdiene som ligger på et eller flere signal.

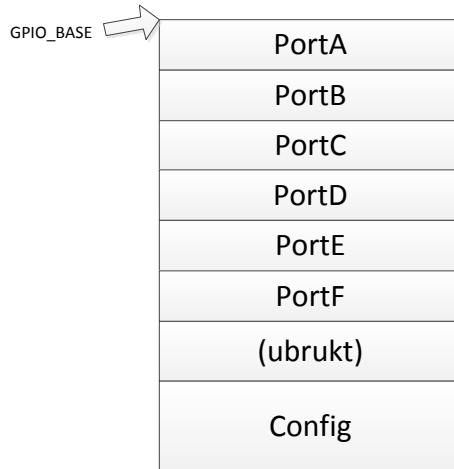
2.1.1 Pinner

Et signal er i denne sammenhengen én enkelt ledning, kabel, signal, eller hvordan man vil tolke det, som kan ha verdien 1 eller 0. Et signal som dette kalles en pin. Vi kan bruke slike pinner til å kontrollere perifere enheter. For eksempel er LED-dioden og begge knappene koblet direkte på hver sin pin. Pinnene er også fysisk mappet ut på kitet, og kan derfor kobles til hva som helst av eksterne enheter. Hvis du ser på alle hullene som går langs kantene på din STK3700, ser du at mange av dem er merket med ting som PB10, PE2 osv. Disse er henholdsvis pin 10 på port B, og pin 2 på port E.

2.1.2 Porter

Hver pin er underordnet en port, og all aksess til en pinne gjøres gjennom porten den hører til. Hver port har en rekke registre, som inkluderer blant andre DOUT og DIN. Siden disse registrene er underordnet hver sin port omtaler vi de heretter som *portregistrene*. De individuelle pinnene er representert i de forskjellige *posisjonene* i disse registrene. Hver pin har altså en gitt posisjon i hvert register. Dette utdypes seksjon 2.1.5.

2.1.3 GPIO-registrene



Figur 2.2: Minneområdet for GPIO

Figur 2.2 viser hvordan GPIO-registrene er gruppert i minnet, der adressene vokser nedover. `GPIO_BASE` er en konstant adresse som representerer starten av GPIO-registrene. Det vil si at det første GPIO-registeret har adresse `GPIO_BASE + 0w`, mens det andre har adresse `GPIO_BASE + 1w` osv.

Portregistrene

De første 54 GPIO-registrene representer hver port og alle dens pinner. Denne delen består av 6 grupper kalt PortA, PortB, ..., PortF. Hver av disse gruppene består av 9 registre. Disse registrene refereres heretter til som *portregisterene*. Plasseringen til hver gruppe av portregister er vist i figur 2.2. Det er 6 porter totalt, port A til og med port F, og dermed er det totale antallet portregister:

$$\text{portregisterPerPort} * \text{antallPorter} = 9w * 6 = 54w$$

Hvert sett med portregister er organisert på følgende måte der offset-er fra portens første register.

Offset	Navn	Beskrivelse
0B	CTRL	Control
4B	MODEL	Mode low
8B	MODEH	Mode high
12B	DOUT	Data output
16B	DOUTSET	Set data output
20B	DOUTCLR	Clear data output
24B	DOUTTGL	Toggle data output
28B	DIN	Data input
32B	PINLOCKN	Lock pin

Config-registrene

Resten av registrene starter 10w etter det siste portregisteret. Disse registrene er i figur 2.2 gruppert som *Config*. Config-registrene brukes blant annet til å sette opp og håndtere GPIO-en. Det første av disse registrene vil ha følgende offset fra `GPIO_BASE`.

$$offset = (\text{antall Portregister} + 10)w = (54 + 10)w = 64w = 256B$$

Disse registrene er organisert som følger, der offset-en er fra fra `GPIO_BASE`.

Offset	Navn	Beskrivelse
256B	EXTIPSELL	External Interrupt Port Select Low Register
260B	EXTIPSELH	External Interrupt Port Select High Register
264B	EXTIRISE	External Interrupt Rising Edge Trigger Register
268B	EXTIFALL	External Interrupt Falling Edge Trigger Register
272B	IEN	Interrupt Enable Register
276B	IF	Interrupt Flag Register
280B	IFS	Interrupt Flag Set Register
284B	IFC	Interrupt Flag Clear Register
288B	ROUTE	I/O Routing Register
292B	INSENSE	Input Sense Register
296B	LOCK	Configuration Lock Register
300B	CTRL	GPIO Control Register
304B	CMD	EM4 Wake-up Clear Register
308B	EM4WUEN	EM4 Wake-up Enable Register
312B	EM4WUPOL	EM4 Wake-up Polarity Register
316B	EM4WUCAUSE	EM4 Wake-up Cause Register

2.1.4 Utregning av register-adresse

For å aksessere et register må vi først regne ut adressen. Adressen er en minneadresse, så dette må altså regnes ut i bytes. For alle andre registre enn portregistrene finnes det et fast offset som listet opp over. Addressen til disse vil være `GPIO_BASE + offset-et`. Men for å aksessere et portregister trenger man i tillegg et offset fra `GPIO_BASE` til porten.

For å regne ut dette offset-et ganger vi antall portregister per port med portnummeret, der portnummer er 0 for port A, 1 for port B osv. Hele adressen vil da være `GPIO_BASE + portoffset + registeroffset`. Antall portregister per port er alltid 9w, siden alle portene har 9 registre og hvert register er 1w stort. Vi kan på generelt grunnlag regne ut basen for en port på følgende måte:

$$GPIO_BASE + 9w * portnummer$$

Når vi har denne port-basen kan vi plusse på et register-offset for å få hele adressen til et register. Eksempelvis er DOUT-registeret (offset 12B) på henholdsvis port A og B:

$$\begin{aligned} pA_{DOUT} &= GPIO_BASE + 9w * 0 + 12B \\ pB_{DOUT} &= GPIO_BASE + 9w * 1 + 12B \end{aligned}$$

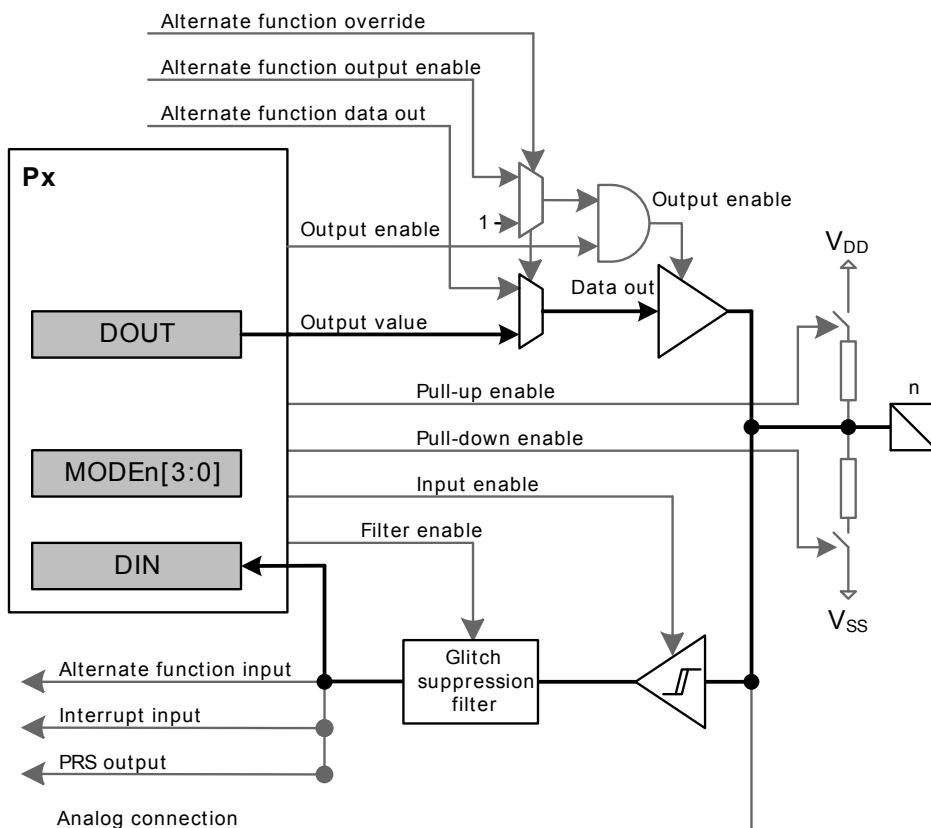
Her er det viktig å holde styr på enhetene! Legg merke til både w og B brukes som enhet i formlene over, og husk at $1w = 4B$. Pseudokoden under viser hvordan man kan regne ut adressen til DOUT-registeret på en vilkårlig port angitt av variabelen `port`. Pseudokoden er på tre-adresseform som introdusert i seksjon 1.5, og kan lett oversettes til assembly.

```
1 LOAD R0, port
2 LOAD R1, 36 // 9w
3 MULTIPLY R2, R0, R1
4 LOAD R3, GPIO_BASE // 0x60004000
5 ADD R4, R2, R3
6 LOAD R5, DOUT // 12B
7 ADD R6, R4, R5
```

Her brukes en LOAD-instruksjon for å laste registre med verdier fra minnet (`port`-variablen) eller konstanter (36, `GPIO_BASE`, `DOUT`). På linje 3 multipliseres portnummeret med antall portregister per port. Dette adderes med `GPIO_BASE` på linje 5, som videre adderes på offsetet for `DOUT` på linje 7. `R6` vil da inneholde adressen til `DOUT` på porten angitt av `port`.

2.1.5 Bruk av Input/Output

For å sette og lese verdiene på pinnene kan vi bruke portregisterene DIN og DOUT, samt hjelprechistarne til DOUT: DOUTSET, DOUTCLR og DOUTTGL. Dette kan i praksis brukes til å for eksempel lese status på en knapp, eller å skru av og på en LED. Denne seksjonen forklarer hvordan vi kan vi bruke disse registrene for å få til ting som dette. Ethvert innhold i et register kan i denne seksjonen antas å være oppgitt binært.



Figur 2.3: DOUT og DIN pin-kobling

Figur 2.3 viser hvordan registrene DOUT og DIN er koblet på pin n på port x .

Registerinnhold

I DIN, DOUT og de fleste andre GPIO-registrene representerer hver bit en pin. Et eksempel for verdien til de første 8 pinnene i et register er represenert i tabellen under.

pin	7	6	5	4	3	2	1	0
verdi	0	0	1	0	1	0	0	0

Pin-nummerne korresponderer med normal binær indeksering som introdusert i seksjon 1.3.1. I eksempelet over er biten 1 på posisjonene som representerer pin 3 og 5, og 0 på resten. Derfor kan vi tolke dette som at pin 3 og 5 har verdi 1, mens resten har verdi 0. Videre i kompendiet vil posisjonen til en viss pin i et register med slik indeksering kalles en **pin-bit**.

DIN - Data IN

DIN er inputregisteret og inneholder hva pinner er satt til eksternt. Eksempelvis kan man bruke DIN for å lese statusen til en knapp uten interrupt. For å finne verdien på en spesifikk pin leser vi DIN-registeret og sjekker verdien på pinnens posisjon.

Hvis vi vil lese verdien til pin 7 starter vi med å lese ut hele DIN-registeret, la oss for eksempel si at $DIN = 10011000$. Vi er ute etter bit 7, altså biten helt til venstre, resten bryr vi oss ikke om. For å lese pin 7 må vi på en eller annen måte nullstille alle andre bits enn bit 7, og så sjekke om resultatet er lik eller ulik 0. Dette kan vi gjøre med å utføre en bitwise AND med DIN og 10000000 (bit 7). Dette fungerer fordi på alle andre posisjoner enn bit 7 ANDes det med 0, og å ANDe med 0 gir alltid 0 uavhengig av hva den andre verdien er. På posisjonen til bit 7 derimot ANDer vi med 1, og når en verdi ANDes med 1 beholdes alltid verdien. Altså gir det verdien 1 dersom bit 7 er 1, og 0 dersom bit 7 er 0. Vi ender opp med uttrykket $10011000 \text{ AND } 10000000$ som, siden $1 \text{ AND } 1 = 1$, gir verdien 10000000 som er ulik 0, og vi vet da at pinnen er 1.

Pseudokoden under viser hvordan vi sjekker pin 7 i DIN-registeret og brancher til koden `pin_er_satt` dersom pinnen er 1:

```
1 LOAD R1, R0 // Anta at R0 = adressen til DIN
2 AND R2, R1, 0b10000000
3 COMPARE R2, 0
4 BRANCH_IF_NOT_EQUAL pin_er_satt
```

Vi antar at adressen til DIN-registeret finnes i R0¹, og innholdet til adressen, altså innholdet i registeret, lastes dermed på første linje til R1. Deretter ANDes registeret med bit 7 på linje 2, og resultatet sammelignes med 0 på linje 3. Instruksjonen på linje 4 bruker resultatet av forrige sammenligning til å branche til koden navngitt `pin_er_satt` dersom sammenligningen fant at verdiene var ulike. *Kondisjonell branching* som dette utdypes når de tilsvarende instruksjonene for mikrokontrolleren introduseres i seksjon 2.4.6.

DOUT - Data OUT

DOUT-registeret brukes til å angi output på porten og er da typisk et register man skriver til. Hvis vi vil sette pin 0 til verdien 1, kan vi skrive 00000001 til DOUT.

Som regel må vi anta at DOUT allerede inneholder et eller annet på de andre bit-posisjonene som vi gjerne ikke vil ødelegge.² Hvis for eksempel pin 7 på forhånd var satt til 1 (altså DOUT = 10000000), ville vi overskrevet dette hvis vi skrev 00000001 til registeret. Derfor bruker vi vanligvis bitwise OR når vi vil skrive til et register. Vi starter med å hente ut verdien i registeret (10000000), og deretter ORer vi inn de pinnene vi vil slå på (00000001) og resultatet blir 10000001 der både pin 7 og pin 0 er på. Alternativt kan vi bruke hjelpperregisteret DOUTSET, som tar i mot 00000001 og ORer dette automatisk inn i registeret for oss.

I de tilfellene hvor vi skal sette en pin til 1, fungerer bitwise OR helt fint. Om vi skal sette noe til 0 derimot, blir det litt vanskeligere. Derfor har vi hjelpperregisteret DOUTCLR. DOUTCLR brukes på akkurat samme måte som DOUTSET, bare at det vi skriver til DOUTCLR ender opp som 0 i DOUT. Hvis vi vil sette pin 0 til verdi 0 kan vi da skrive 00000001 til DOUTCLR, og DOUT vil bli gå fra 10000001 til 10000000. Tilsvarende er det med DOUTTGL, bare at her blir det vi setter som 1 togglet. Altså satt til det motsatte av hva det var fra før.

Pseudokoden under viser hvordan vi kan trygt skrive bit 0 til DOUT ved bruk av hjelpperregisteret DOUTSET:

```

1 | LOAD R1, 0b00000001
2 | STORE R1, R0 // Anta at R0 = adressen til DOUT

```

¹Se seksjon 2.1.4 for pseudokode for å regne ut registeradresser.

²Et slikt type hensyn er et eksempel på god praksis i motsetning til streng nødvendighet. Hvis ingen annen I/O gjøres på porten er det kanskje ikke nødvendig å ta slike hensyn, men dette krever at programmereren har total kontroll over hele systemet. Risiko knyttet til feil som oppstår av slike antakelser er ofte mye større enn innsatsen som kreves for å unngå dem.

På linje 1 laster vi pin 0 til R1, og vi skriver denne direkte til DOUTSET på linje 2 og GPIO-maskinvaren vil gjøre ORingen for oss. Eventuelt kan vi heller gjøre ORingen i koden vår og så skrive direkte til DOUT, uten bruk av hjelpperregisteret DOUTSET:³

```

1 LOAD R1, 0b00000001
2 LOAD R2, R0 // Anta at R0 = adressen til DOUT
3 OR R3, R1, R2
4 STORE R3, R0 // Anta at R0 = adressen til DOUT

```

På linje 1 laster vi innholdet til DOUT til R1. Vi ORer dette med pin 0 på linje 3, og lagrer resultatet tilbake til DOUT på linje 4.

2.1.6 Oppsett av Input/Output

Før vi kan bruke en pin til Input/Output (I/O) må vi angi hvilken *mode* den skal ha.⁴ Moden betegner om pinnen er *input* eller *output*, men det er også mulig å spesifisere forskjellige varianter av disse. Vi gjør dette vi å skrive til portregistrene MODEL, MODEH og DOUT.

DOUT

Verdien til DOUT-registret i det en pin er satt opp påvirker hvordan pin-en virker. Det er derfor viktig å sette opp DOUT-registret *før* MODE-registrene. I begge bruksområdene i dette øvingsopplegget (normal output for LED og aktivt lav input for knapper) skal pin biten i DOUT settes til 0 for dette formålet.

MODE register

I MODE-registrene er hver pin representert av en gruppe med fire bits. Disse fire bitsene skal angi hvilken mode pinen skal ha. Hver mode som systemet støtter har en unik 4-bits ID som er definert av rammeverket. Med opptil 16 pinner på hver port sier en kjapp utregning ($4 * 16 = 64$) at vi faktisk behøver 64 bits for å angi dette. MODE er derfor delt opp i to 32 biters registre: MODEL og MODEH, der den siste bokstaven betegner enten low

³Ved å gjøre dette i koden flytter vi prosesseringen fra GPIO-maskinvarens kretser til prosessoren. På stor skala er ikke dette like effektivt, da GPIO-kretsene vil ligge ubrukt mens prosessoren vil måtte jobbe litt ekstra. Som akademikere har vi heldigvis friheten til å velge den morsomme måten framfor den mest effektive.

⁴Merk at i flere av øvingene blir dette automatisk gjort av det utleverte oppstartsprogrammet.

(L) eller high (H). Low-varianten representerer pin 0 til 7 og high-varianten pin 8 til 15. I tabellene nedenfor vises begge registrene og deres innhold etter at knappen PB0 (på pin 9) er konfigurert til å ha mode-en `GPIO_MODE_INPUT`, definert av rammeverket som 0001.

Pin	pin15	pin14	pin13	pin12	pin11	pin10	pin09	pin08
Register	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0001	xxxx

Tabell 2.1: MODEH

Pin	pin07	pin06	pin05	pin04	pin03	pin02	pin01	pin00
Register	xxxx							

Tabell 2.2: MODEL

Her ser vi at gruppa til pin 9 i MODEH er satt til 0001, som tilsvarer `GPIO_MODE_INPUT`. Alle andre bits bryr vi oss ikke om akkurat nå, og er representert med 'x'.

For å sette MODE-registrene må vi trikse litt med bit-ene. Det er hovedsakelig to problemer vi må ta hensyn til. Først, som alltid, må vi passe på å ikke skrive over noen av de andre delene av registeret. I eksempelet over kan vi altså kun endre på gruppen fra og med bit 4 til og med 7, resten må forbli uendret. Dette fører til det andre problemet, å ORe inn en verdi som kan være 0. Hvis verdien vi vil ORe inn er 0, og posisjonen allerede inneholder 1, vil det originale innholdet forbli. I eksempelet over vil vi skrive 0001 (`GPIO_MODE_INPUT`) til gruppa for pin 9. Hvis denne allerede inneholder for eksempel 0100 (`GPIO_MODE_OUTPUT`), vil resultatet av en vanlig OR bli 0101 som er hverken input eller output. Vi løser dette ved å først nullstille hele gruppa med en bitwise AND over hele registeret, og deretter ORe inn verdien vår.

Vi ANDer registeret med en “maske” med 1111 på alle gruppene vi vil beholde og 0000 på gruppa vi vil nullstille. Vi starter med verdien 1111 og leftshifter 4 posisjoner for å komme til pin 9-gruppa:

1111 << 4
=
0000 0000 0000 0000 0000 0000 1111 0000

Dette inverterer vi for å få maska:

NOT(1111 << 4)
=
1111 1111 1111 1111 1111 1111 0000 1111

Deretter ANDer vi denne maska med registeret ('x' representerer et bit fra registeret):

(NOT(1111 << 4)) AND MODEH
=
xxxx xxxx xxxx xxxx xxxx xxxx 0000 xxxx

Til slutt leftshifter vi verdien vi vil skrive (0001) 4 posisjoner (til pin9-gruppa) og ORer den inn:

((NOT (1111 << 4)) AND MODEH) OR (0001 << 4)
=
xxxx xxxx xxxx xxxx xxxx xxxx 0001 xxxx

Verdien vi får fra dette utrykket kan vi trygt skrive tilbake til MODEH-registeret.

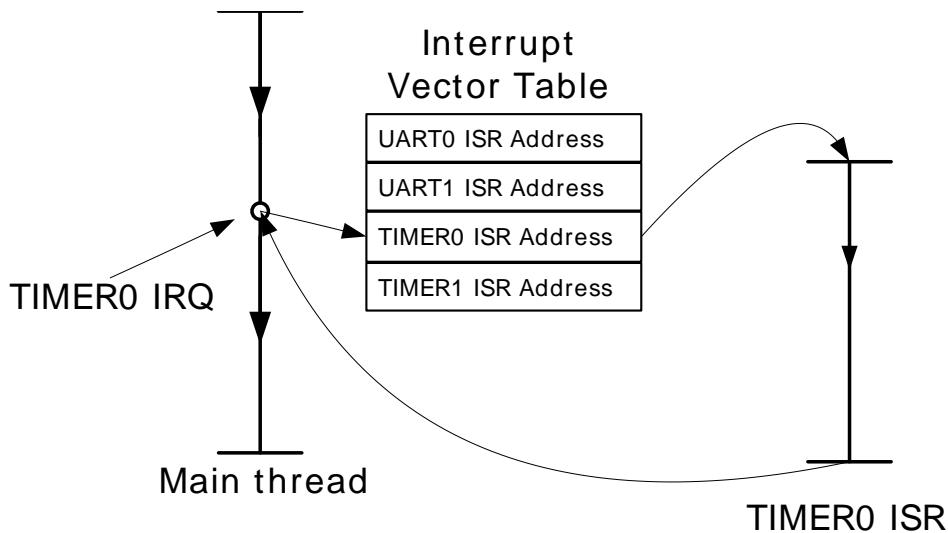
2.2 Interrupt

Denne seksjonen forklarer grunnleggende felles oppsett av alle typer interrupts, oppsett av interrupt-vektoren. Spesifikk oppsett av GPIO-interrupts er forklart i seksjon 2.2.3, og SysTick-interrupts i seksjon 2.3.

2.2.1 Bakgrunn

Interrupt er som navnet tilsier, en hendelse som *avbryter* prosessoren. Når et interrupt forekommer vil prosessoren slutte med det den driver med, finne ut hvalags interrupt det var, og kjøre koden som hører til denne typen interrupt. Alternativet for interrupt er noe som kalles **polling**. Hvis man

for eksempel poller en knapp, har man en løkke som sjekker knappen om og om igjen, helt til den trykkes inn. Bruk av interrupts er nesten alltid en fordel, da prosessoren kan mellom knappetrykkene drive med hva som helst av andre beregninger, eller til og med sove og dermed spare mye strøm. Når knappen trykkes inn blir prosessoren enten avbrutt eller vekket opp, og kan dermed utføre knappens funksjon.



Figur 2.4: Interrupt handling

Figur 2.4 illustrerer et interrupt. En IRQ (**Interrupt ReQuest**), i dette eksempelet *TIMER0*, avbryter koden som kjører (*Main thread*). Deretter blir det slått opp i et **Interrupt Vector Table** og riktig ISR (**Interrupt Service Routine**) blir funnet fram. ISR-en kjøres, og til slutt gjenoptas *Main thread* på samme sted som den ble avbrutt på.

2.2.2 Oppsett av interrupt-vektor

Interrupt Vector Table er en samling med interrupt-vektorer som er ansvarlige for at hver interrupt-kilde skal kalle på riktig funksjon. Felles for oppsett av alle typer interrupts er at vi må sette en interrupt-vektor til å peke fra interrupt-typen til funksjonen vi vil at skal kjøre ved interrupt.

Koden som initialiserer systemet kjører blant annet en assembly-fil som kalles `startup_efm32gg.s`. Denne filen inneholder Interrupt Vector Table-et med alle interrupt-vektorene. Tabell 2.3 viser et eksempel på en interrupt-vektor fra denne.

```

.weak GPIO_ODD_IRQHandler
.globl GPIO_ODD_IRQHandler
.set GPIO_ODD_IRQHandler, _IRQHandlerinterrupt

```

Tabell 2.3: GPIO_ODD_IRQHandler

Denne interrupt-vektoren er satt til å peke på ISRen `_IRQHandlerInterrupt` som er en standard interrupt-rutine som ikke gjør annet enn å returnere. Det rare er at hverken denne vektoren, eller noen av de andre vektorene, kan direkte forandres. Men pseudo OP-en `.weak` angir at en eventuell annen funksjonsdeklarasjon med det samme navnet (`GPIO_ODD_IRQHandler`) vil overstyre deklarasjonen i `startup_efm32gg.s`. Dette betyr at for å sette interrupt-vektoren trenger vi bare å deklarere en funksjon med akkurat dette navnet i vår kode, og den vil automatisk overstyre defaulten (`_IRQHandlerinterrupt`).

I C gjøres denne overridingen kun ved å deklarere en funksjon med det samme navnet. For eksempel:

```

void GPIO_ODD_IRQHandler(void) {
    // Din interrupt-kode her
}

```

I assembly gjøres dette ved å deklarere et label med samme navn, i tillegg til et par pseudo ops. Merk at du også må manuelt returnere fra interrupt:

```

.global GPIO_ODD_IRQHandler
.thumb_func
GPIO_ODD_IRQHandler:
    // Din interrupt-kode her
    BX LR // Returner fra interrupt

```

Pseudo op-en `.global` sier til linkeren at labelet `GPIO_ODD_IRQHandler` skal gjelde for hele programmet. `.thumb_func` sier at den etterfølgende koden er i instruksjonssettet *thumb-2*.

2.2.3 Interrupt-oppsett for GPIO

For å oppdage at for eksempel en knapp er trykket, vil vi helst bruke interrupts. Vi skal sette opp dette slik at når knappen trykkes ned, blir en funksjon i koden vår kalt. For å få til dette må vi sette riktig interrupt-vektor til å peke til vår interrupt handler, og deretter sette GPIO-registrene EXTIPSEL, EXTIFALL, IF og IEN (se seksjon 2.1.3). På kitet er det to

knapper, *PB0* og *PB1*. *PB0* er koblet til pin 9 på port B, og *PB1* er på pin 10 på port B.

Interrupt-vektorene for GPIO

Når et interrupt skjer vil vi at vår interrupt handler skal kalles. GPIOen har to interrupt-vektorer, `GPIO_EVEN_IRQHandler` og `GPIO_ODD_IRQHandler`. Dette begrenser hvor mange interrupt handlere man kan ha for hele GPIOen til to, men det er alltid mulig for interrupt handleren å lese pinnene og finne ut hva den skal gjøre. Førstnevnte interrupt-vektor brukes til alle pinner med partall, og sistnevnte til oddetall. Siden knappen *PB0* har pin-nummer 9, blir det `GPIO_ODD_IRQHandler` som kalles når denne knappen trykkes.

EXTIPSEL - EXternal Interrupt Port SElect

Interrupts i EFM32 fungerer slik at hvert pin-nummer kan kun brukes én gang som interrupt. Så det er altså ikke mulig å bruke for eksempel både pin 3 på port A og pin 3 på port B til å generere interrupt i samme oppsett. På denne måten trenger systemet kun å vite hvilken port hvert pin-nummer skal gi interrupt på, og denne informasjonen ligger i EXTIPSEL-registrene.

I likhet med MODE-registrene (se seksjon 2.1.6) er hver pin representert av en gruppe på fire bits i EXTIPSEL-registrene. Disse fire bitsene skal angi på hvilken port pinnen skal gi interrupt. For eksempel ville vi satt port B på gruppa til pin 9 for å generere interrupts ved knappetrykk på *PB0*, siden denne knappen er på port B pin 9. I likhet med MODE består EXTIPSEL av to registre, EXTIPSELL og EXTIPSELH, som representerer henholdsvis pin 0 til 7 og pin 8 til 15. I tabellene nedenfor vises begge registrene og deres innhold etter at knappen *PB0* er konfigurert til å gi interrupt.

Pin	pin15	pin14	pin13	pin12	pin11	pin10	pin09	pin08
Register	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	0001	xxxx

Tabell 2.4: EXTIPSELH

Pin	pin07	pin06	pin05	pin04	pin03	pin02	pin01	pin00
Register	xxxx							

Tabell 2.5: EXTIPSELL

Her ser vi at gruppa til pin 9 i EXTIPSELH er satt til binært 0001, som

er port-nummeret til port B. Som før er alle bits vi ikke bryr oss representert med 'x'.

EXTIPSEL-registrene modifiseres på samme måte som MODE-registrene. En C-vennlig formel for dette er beskrevet og forklart i seksjon 2.1.6. Under følger assembly-vennlig pseudokode på tre-adresseform for å sette EXTIPSELLH, der vi bruker samme formel som for MODE-registrene:

```
1 LOAD R1, 0b1111
2 LEFT_SHIFT R2, R1, 4
3 NOT R3, R2
4 LOAD R4, R0 // Anta at R0 = adressen til EXTIPSELH
5 AND R5, R3, R4
6 LOAD R6, PORT_B // 0b0001
7 LEFT_SHIFT R7, R6, 4
8 OR R8, R5, R7
9 STORE R8, R0 // Anta at R0 = adressen til EXTIPSELH
```

Masken produseres fra og med linje 1 til og med linje 3. Maska ANDes med EXTIPSELH-registeret på linje 5⁵, som i effekt nullstiller pin 9-gruppa og klargjør ORing av verdien. Verdien vi vil skrive (PORT_B = 0b0001), lastes og left shiftes inn til pin 9-gruppa på henholdsvis linje 6 og 7. På linje 8 ORes denne verdien inn i den klargjorte EXTIPSELH-verdien. Resultatet vi da sitter igjen med i R8 inneholder EXTIPSELH-registeret med verdien 0b0001 korrekt satt inn. Denne verdien skrives tilbake til EXTIPSELH på linje 9.

EXTIFALL - EXTernal Interrupt FALLing Edge Trigger

EXTIFALL-registeret brukes til å angi at interruptet på pinnen skal skje ved falling edge, altså når verdien på pinnen faller fra 1 til 0. Knappene på kitet er aktivt høye, det vil si at de har verdien 1 når de ikke er trykket ned, og verdien 0 når de trykkes ned. Tilsvarende finnes et EXTIRISE-register for rising edge, men det vil ikke bli nødvendig i dette øvingsopplegget.

EXTIFALL-registeret brukes på samme måte som portregistrene DOUT og DIN, der hver bit representerer en pin (se seksjon 2.1.5). For at interrupts skal skje når knappen trykkes inn, må vi sette bit 9 i EXTIFALL-registeret til 1. Denne verdien må settes inn med bitwise OR, som forklart i seksjon 1.3.4. Se DOUT-seksjonen under seksjon 2.1.5 for pseudokode på dette. Det samme gjelder for IF- og IEN-registrene introdusert under.

⁵Vi antar at R0 inneholder adressen til EXTIPSELH, se seksjon 2.1.4 for hvordan man regner ut registeradresser.

IF - Interrupt Flag

IF-registeret representerer hver pin på et bit på lik linje som EXTIFALL. Når et interrupt skjer på en pin, settes automatisk pin-biten til 1 i IF-registeret. Hvis pin-biten til for eksempel pin 9 er 1 i IF, vet vi at et ubehandlet interrupt for pin 9 har skjedd. Heldigvis trenger vi ikke sjekke IF-registeret for interrupts for å vite når det kommer, det ville gjort interrupts meningsløst. Men når vi har behandlet et interrupt må vi oppdatere IF-registeret for å si fra at interruptet er blitt behandlet. Vi må altså sette bit-en tilbake til 0 i interrupt-handlaren vår. IF-registeret har på lik linje med DOUT-registeret hjelperegistre for å gjøre dette. Ved å skrive pin-biten til IFC (Interrupt Flag Clear) forteller vi systemet at interruptet er blitt behandlet.

I tillegg til å gjøre dette i interrupt-handleren er det god praksis å i tillegg gjøre det under oppsettet for å unngå problemer med eventuelle tidligere ubehandledde interrupts. Det er ikke mulig for systemet å motta flere interrupts for en pin mens dens biten er 1 i IF.

IEN - Interrupt ENable

IEN-registeret er det siste vi setter, og er det som enables interruptet på gitt pin. Vi har hittil konfigurert for følgende: Interrupt på pin 9 skal komme fra port B (EXTIPSELH), og pin 9 skal generere interrupts ved falling edge (EXTIFALL). Nå er det på tide å enable interrupts på pin 9. IEN-registeret representerer hver pin på et bit på lik linje som IF, og vi trenger da bare å legge inn bit 9 med en bitwise OR. Nå vil interrupt-vektoren som hører til pin 9 kalles hver gang verdien til pin 9 på port B går fra 1 til 0, gitt at IF-registeret blir riktig oppdatert etter hvert interrupt.

2.3 SysTick

SysTick er et system som brukes for å holde styr på tid ved hjelp av klokkesignalet. I SysTick har man et tellerregister som settes til en verdi ved hjelp av et load-register. Ved hvert klokkesignal vil innholdet til tellerregisteret minke med 1, og når registeret har kommet helt ned til 0 genereres et interrupt. Deretter settes verdien i tellerregisteret tilbake til verdien i load-registeret og nedtellingen starter på nytt. Etter riktig oppsett vil SysTick generere interrupts med en fast frekvens.

SysTick er organisert som de følgende fire registrene, der offset er fra SYSTICK_BASE:

Offset	Navn	Beskrivelse
0B	CTRL	Control and Status Register
4B	LOAD	Reload Value Register
8B	VAL	Current Value Register
12B	CALIB	Calibration Register

For å sette opp SysTick må vi bruke registrene CTRL, LOAD og VAL (CALIB vil ikke bli nødvendig).

2.3.1 CTRL - Control and Status Register

I CTRL-registeret er vi kun interessert i de tre bit-ene helt til høyre: Bit 0, 1 og 2. Bit 0 er ENABLE-bit-en, denne må settes til 1 for at SysTick skal være aktiv. Bit 1 er TICKINT-bit-en, denne må settes til 1 om det skal genereres et interrupt når telleregisteret treffer 0. Bit 2 er CLKSOURCE-bit-en, denne bestemmer om det er core-klokken som skal brukes. Hvis vi ville brukt en annen klokke ville vi satt denne til 0, men i dette tilfelle bruker vi core-klokken og setter denne til 1. I vårt tilfelle skal altså alle disse bit-ene settes til 1, som betyr at CTRL-registeret må lastes med det binære tallett 111.

2.3.2 LOAD - Reload Value Register

LOAD-registeret bestemmer hva telleregisteret skal settes til etter at det har truffet 0, og er dermed det som bestemmer frekvensen av interruptene. LOAD-registeret er 24 bit stort, så det har plass til tall som er lavere enn 2^{24} . For at SysTick skal generere et interrupt én gang i sekundet, må LOAD-registeret settes til antall klokkesykler per sekund, som tilsvarer klokkefrekvensen angitt i Hz som er 14000000. For å generere interrupt 10 ganger i sekundet kan man bruke klokkefrekvensen delt på 10.

2.3.3 VAL - Current Value Register

VAL er registeret vi hittil har omtalt som telleregisteret, og er registeret som brukes til å telle nedover. Når VAL er lik 0 vil SysTick generere et interrupt og deretter sette VAL lik LOAD. Ved oppsett er det ikke kritisk hva vi setter VAL til da det kun vil påvirke forsinkelsen for det første interruptet: Setter vi VAL til 0 vil det wrappes opp til verdien av LOAD-registeret med én gang og det første interruptet vil komme uten forsinkelse rett etter SysTick er satt opp. Setter vi det til samme verdi som LOAD vil det første interruptet komme etter en forsinkelse lik frekvensen. Om frekvensen er ett interrupt per sekund vil det første interruptet altså komme etter ett sekund.

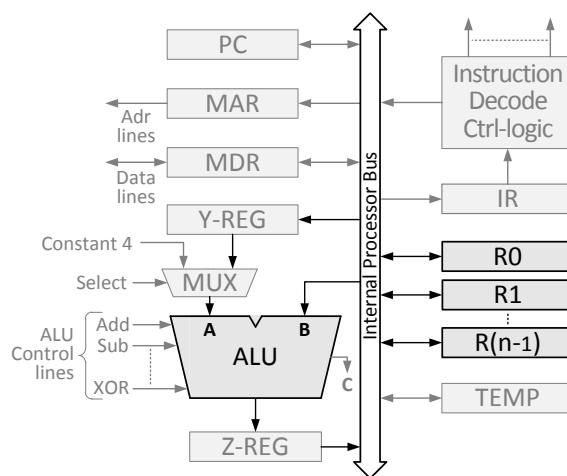
2.3.4 Interrupt-vektor

Siden vi har satt opp SysTick til å generere interrupts, trenger vi naturligvis en interrupt handler. Interrupt-vektoren til SysTick heter `SysTick_Handler`, og det er denne vi må overstyre for å behandle interruptene som blir generert av SysTick. Se seksjon 2.2 for informasjon om overstyring av interrupt-vektorer.

2.4 Instruksjonssett

Et instruksjonssett, eller “assembly-språket”, er settet med instruksjoner som prosessoren forstår. Et kompilert program består av en rekke instruksjoner, og alt en prosessor egentlig gjør er å eksekvere disse instruksjonene.

En slik instruksjon består av en OP-kode, etterfulgt av ingen, ett eller flere operander. OP-kode står for operasjonskode og kan sees på som en enkel funksjon som CPUen har hardkodet støtte for å utføre. Eksempler på operasjoner og deres OP-koder er aritmetiske operasjoner som ADD, boolske operasjoner som CMP (CoMPare), kontrollflyt-operasjoner som BEQ (Branch if EQUAL) og minneoperasjoner som LDR (LoaD Register) og STR (STore Register). Operandene kan eventuelt angi inputen til operasjonen og hvor outputen skal plasseres. Konvensjonen er `OP-kode, output, input 1, input 2`. For eksempel vil instruksjonen `ADD R0, R1, R2` lagre summen av innholdet til R1 og R2 i registeret R0.



Figur 2.5: Datapath-figur fra forelesninger

Et register som for eksempel R0, R1 eller R2 er i denne sammenhengen et **prosessor-register**. Figur 2.5 viser en datapath som brukes mye i forelesningene. Her ser vi at registrene (R0, R1, ..., R(n-1)) er koblet direkte på den interne prosessor-bussen, som igjen er koblet på ALUen. ALUen – enheten som utfører alle instruksjonene – har altså direkte tilgang på disse, og bruker de derfor til å holde all data som den skal jobbe med.

Instruksjonene i instruksjonssettet presenteres her i ren tekst, f.eks. **MOV R5, #3**. Når man programmerer i assembly skrives de på denne måten, men til slutt vil de oversettes til maskinkode av assembleren. Maskinkoden er kun binære tall, men likevel er de tekstlige instruksjonene egentlig svært like som maskinkoden.

	OP-kode	Operand 1	Operand 2
Tekstlig	MOV	R5	#3
Binær	00100	100	00000011

Tabell 2.6: MOV-instruksjonen

Tabell 2.6 viser sammenhengen mellom tekstlig og binær representasjon for instruksjonen **MOV R5, #3**. Observér sammenhengen i operand 1 mellom R5 og det binære tallet 100, og i operand 2 mellom #3 og det binære tallet 00000011.

2.4.1 ARM Thumb-2 instruksjonssett

Prosessoren i EFM32 har en ARM-arkitektur og benytter seg av Thumb-2 standarden som instruksjonssett. Resten av dette kapittelet gir en forenklet innføring i de viktigste delene av instruksjonssettet, og skal dekke det nødvendige for å fullføre øvingsopplegget. Tabell 2.7 forklarer symbolbruken for de følgende tabellene. Mer dokumentasjon finnes i *ARM and Thumb-2 Instruction Set Quick Reference Card*⁶, og mer utfyllende i *Arm Reference Manual*. Se seksjon 2.4.8 for eksempler for bruk av instruksjonene.

⁶<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.qrc0001m/index.html>

Symbol	Forklaring
Rx, Ry, Rz	Et register (R0, R1, R2...).
'direkteverdi'	Direkteverdi (#0, #1, #2...).
'verdi'	Direkteverdi eller register (Rx).
'betingelse'	En betingelse (EQ, NE, PL... (se 2.4.6)).
'label'	En string som representerer en adresse.
'reg-liste'	En liste med registre f.eks. {R1, R2, R3}.

Tabell 2.7: Forklaring av symboler

2.4.2 Flytting av data

Flytting av data innebærer å laste konstanter inn i registre, og å flytte data fra ett register til et annet.

OP-kode	Operander	Beskrivelse
MOV	Rx, 'verdi'	Legg 'verdi' i Rx.

Ved bruk av denne instruksjonen til å laste konstanter inn registre er det viktig å huske på hvordan en instruksjon er representert. Instruksjoner har en fastsatt størrelse, og MOV-instruksjonen for konstanter, som vist i tabell 2.6, er 16 bit stor. Siden den første halvdelen av instruksjonen brukes til å spesifisere OP-koden og destinasjonsregisteret, er det kun satt av 8 bit for konstanten. MOV har derfor kun plass til konstanter lavere enn 2^8 , eventuelt konstanter som kan produseres ved å leftshifte en 8 bits konstant.⁷ For lasting av større konstanter enn dette kan vi bruke LDR-instruksjonen med en “=”-snarvei. Se seksjon 2.4.3 for LDR-instruksjonen og 2.4.8 for eksempler på å jobbe med variabler.

2.4.3 Minnebehandling

Minnebehandling involverer å hente verdier fra minnet inn i et register, og lagring av et registers innhold til minnet.

⁷For å holde den mentale modellen vår av instruksjonssettet mest mulig ren og enkel er flere spesielle egenskaper ved Thumb-2 utelatt fra diskusjonen. I Thumb-2 er det flere versjoner av de samme instruksjonene, noen har variabel lengde, og noen inneholder spesielle felt som gjør dem svært fleksible. En av MOV-instruksjonene har et felt for bitshifting og når instruksjonen utføres på ALU-en vil konstanten leftshiftes antall posisjoner angitt av shift-feltet. Assembleren vil automatisk velge denne instruksjonen når MOV brukes for en konstant som kan produseres av en lefshiftet 8-bits verdi.

OP-kode	Operander	Beskrivelse
STR	Rx, [Ry]	Lagre Rx på minnelokasjon Ry.
STR	Rx, [Ry, Rz]	Lagre Rx på minnelokasjon Ry+Rz.
LDR	Rx, [Ry]	Hent innhold i minnelokasjon Ry og legg i Rx.
LDR	Rx, [Ry, Rz]	Hent innhold i minnelokasjon Ry+Rz og legg i Rx.
LDR	Rx, ='label'	Hent innhold fra minnelokasjon 'label' og legg i Rx.
LDR	Rx, ='direkteverdi'	Last direkteverdi (opptil 32 bits) og legg i Rx.
PUSH	'reg-liste'	Push registrene i 'reg-liste' til stacken
POP	'reg-liste'	Pop fra stacken og legg i registrene i 'reg-liste'

Her betyr "minnelokasjon Ry" adressen som ligger i register Ry.

2.4.4 Bit-behandling

Bit-behandling innebærer operasjoner som bitshifting og bitwise OR og AND.

OP-kode	Operander	Beskrivelse
LSL	Rx, Ry, 'verdi'	Shift Ry 'verdi' plasser til venstre, lagre i Rx.
AND	Rx, Ry, 'verdi'	Gjør et bitwise AND på Ry og 'verdi', lagre i Rx.
ORR	Rx, Ry, 'verdi'	Gjør et bitwise OR på Ry og 'verdi', lagre i Rx.
EOR	Rx, Ry, 'verdi'	Gjør et bitwise Exclusive OR på Rx og 'verdi', lagre i Rx.
MVN	Rx, 'verdi'	Gjør et bitwise NOT på 'verdi', lagre i Rx.

2.4.5 Aritmetikk

Aritmetikk innebærer matematiske operasjoner som multiplikasjon, addisjon og subtraksjon.

OP-kode	Operander	Beskrivelse
MUL	Rx, Ry, Rz	Lagre i Ry ganger Rz i Rx.
ADD	Rx, Ry, Rz	Lagre i Ry pluss Rz i Rx.
SUB	Rx, Ry, Rz	Lagre i Ry minus Rz i Rx.

2.4.6 Kontrollflyt

Kontrollflyt innebærer å hoppe – eller *branche* – rundt om kring i koden, og gjøres med forskjellige *branch-instruksjoner* og med **MOV**-instruksjonen. Slike instruksjoner kan for eksempel brukes der man i et høynivåspråk vil le brukt et funksjonskall eller en struktur som for eksempel **if**, **for** eller **while**. En branch-instruksjon er enten kondisjonell eller ukondisjonell. Når en ukondisjonell branch-instruksjon kjøres branches det i alle tilfeller, mens

en kondisjonell branch-instruksjon vil kun branche dersom en gitt betingelse er oppfylt. Kondisjonelle branches benytter seg av betingelser, som igjen benytter seg av noe som heter *condition flags*, eller betingelsesflagg.

Condition flags

Et condition flag er en bit av et statusregister, der statusregisteret inneholder flere condition flags på faste posisjoner. Condition flag-ene settes av ALUen når en instruksjon har kjørt, og sier noe om hva instruksjonen resulterte i. Disse kan leses i etterkant for å avgjøre noe om instruksjonen, for eksempel om den resulterte i et positivt tall.

Tabellen under viser to flagg Z og N, og hva verdiene deres betyr:

Flagg	Navn	Beskrivelse
Z = 1	Zero	Instruksjon gav 0.
Z = 0	Zero	Instruksjon gav ikke 0.
N = 1	Negative	Instruksjon gav negativt tall.
N = 0	Negative	Instruksjon gav positivt tall eller 0.

Z-flagget vil settes til 1 når for eksempel en subtraksjon resulterer i 0, og N-flagget settes til 0 når for eksempel en addisjon resulterer i et positivt tall.

Betingelser

En betingelse er en slags abstraksjon av condition flag-ene og er det vi bruker i koden. Hver betingelse innebærer et krav til et eller flere condition flags. For eksempel må Z-flagget være 1 for at EQ-betingelsen skal være true.

Betingelse	Navn	Krav til condition flag
EQ	Equal	Z = 1
NE	Not equal	Z = 0
MI	Minus/negative	N = 1
PL	Plus/positive eller 0	N = 0

Compare-instruksjonen

Compare-instruksjonen (CMP) er instruksjonen som sammeligner to verdier. I et høynivåspråk kan du se på CMP som det som er inni parantesene i for eksempel en if-setning. CMP sammenligner verdiene og legger deretter sin output i condition flag-ene.

OP-kode	Operander	Beskrivelse
CMP	Rx, 'verdi'	Setter condition flag-ene basert på sammenligning av Rx og 'verdi'.

En sammenligning setter condition flag-ene på følgende måte, sammenlign flaggene med kravene for betingelsene over!

Resultat	Z-flag	N-flag	Betingelse oppfylt
Rx == 'verdi'	1	0	EQ, PL
Rx < 'verdi'	0	1	NE, MI
Rx > 'verdi'	0	0	NE, PL

Branch-instruksjoner

Branch-instruksjonene er instruksjonene som tar seg av selve hoppingen i koden.

OP-kode	Operand	Beskrivelse
B	'label'	Branch til instruksjonen med addresse 'label'.
B'betingelse'	'label'	Branch til 'label' dersom 'betingelse' er oppfylt.
BL	'label'	Branch til 'label', lagre returadressen i LR (Link Register). Merk at dette skriver over forrige LR!
BX	Rx	Branch til adressen i register Rx.
MOV	PC, Rx	Denne instruksjonen brukes til å returnere fra interrupt. Flytt innholdet i Rx inn i programtelleren. I praksis det samme som en ukondisjonell branch til et register i stedet for label.

I dybden: Sammenheng mellom condition flag og betingelser

EQ-betingelsen signaliserer at to tall er like. Fra beskrivelsen om condition flags og betingelser ser vi at EQ krever at Z-flagget er 1, og at Z kun settes til 1 når en instruksjon resulterer i 0. Dette betyr at hvis en instruksjon resulterer i 0 kan vi tolke det som likhet. Compare-instruksjonen må derfor være en aritmetisk instruksjon, der resultatet 0 betyr likhet: Hvis vi subtraherer en verdi y fra en annen verdi x og resultatet er 0, vet vi at $x = y$, og en sammenligning kan dermed gjøres med subtraksjon. Z-flagget vil i dette tilfellet settes til 1 ved likhet og 0 ved ulikhet, akkurat som krevd av henholdsvis EQ-betingelsen og NE-betingelsen.

Dette vil også fungere for andre slags sammenligninger. N-flagget settes til 1 når en subtraksjon resulterer i et negativt tall. Hvis vi sammenligner x

og y med operasjonen $x - y$ og resultatet er negativt, vet vi at x er mindre enn y . Dette reflekteres i MI-betingelsen (mindre enn): MI krever at N er 1. Det motsatte tilfellet, høyere enn eller lik, reflekteres av PL-betingelsen som krever at N er 0.

2.4.7 I dybden: Implementasjon av funksjoner

Funksjonskall er en litt mer komplisert form for kontrollflyt. Hvordan man implementerer funksjoner er helt avhengig av hvordan man vil definere en funksjon. Hvis man vil ha mulighet til å gi argumenter, returnere verdier og deklarere lokale variabler må man eksplisitt implementere alt dette. Du står fritt til å implementere funksjoner med så lite eller mye funksjonalitet du vil, om i det hele tatt. Vi skal her se på de aller mest grunnleggende måtene å implementere funksjoner på. Det er i hovedsak to problemstillinger vi da må vurdere: Returnering og preservering av registre.

Returnere fra funksjon

Når vi har branchet til en funksjon, hvordan vet denne funksjonen hvor den skal returnere? En funksjon skal tross alt kunne kalles fra forskjellige steder, så returneringen skal ikke nødvendigvis gjøres til det samme stedet hver gang. Vi kan i første omgang løse dette problemet ved å, ved hvert funksjonskall, lagre returadressen et eller annet sted rett før vi brancher til funksjonen. Funksjonen som ble kalt på kan da bruke denne adressen til å returnere. I vårt tilfelle kan LR-registret (Link Register) brukes til å lagre returadressen. Vi definerer et **primitivt funksjonskall** som følgende prosedyre:

1. Sett LR til adressen til den neste instruksjonen etter funksjonskallet og branch til funksjonen (se BL-instruksjonen).
2. Funksjonen som ble kalt kan returnere ved å branche til adressen i LR.

Nøstede funksjonskall

Men hva skjer dersom funksjonen som ble kalt videre kaller på en annen funksjon? Hvis vi bruker primitive funksjonskall vil vi måtte skrive over returadressen i LR for å kunne returnere fra denne andre funksjonen, men da kan vi ikke lenger returnere fra den første! Problemet blir enda større hvis vi vil tillate en nærmest uendelig dybde av nøstede kall, som for eksempel rekursive funksjoner krever. Løsningen er å bruke *stacken*. Vi definerer et **trygt funksjonskall** som følgende prosedyre:

1. Push nåverende verdi av LR til stacken.
2. Utfør et primitivt funksjonskall.
3. Når funksjonen har returnert popper vi tilbake fra stacken inn til LR og får tilbake returadressen vi pushet i punkt 1.

Registerpreservering

Den andre problemstillingen, registerpreservering, er egentlig en generalisering av det samme problemet som med næstede kall: Se for deg at vi lagrer en verdi i register R1, kaller på en funksjon, og etter returnering leser verdien i R1. Hvis funksjonen vi kalte på også har skrevet til R1, eller eventuelt kalt på en annen funksjon som har skrevet til R1, er den opprinnelige verdien blitt skrevet over. Dette problemet kan løses ved at alle funksjoner starter med å pushe alle registrene den kommer til å skrive over og popper dem tilbake igjen rett før den returnerer. Den har da *preservert* registrene. En **trygg funksjon** kan dermed defineres som følgende prosedyre:

1. Push alle registre funksjonen skal bruke, inkludert LR.
2. Utfør koden til funksjonen. Dette kan inkludere et ubegrenset antall *primitive* funksjonskall.
3. Pop alle registrene tilbake.
4. Returner ved å branche til adressen i LR.

Merk at dette er en generalisering av det trygge funksjonskallet vi definerer: Dersom en trygg funksjon skal kalle på andre funksjoner, og dermed bruke LR-registret, pusher den LR, og alle andre registre den vil bruke, til stacken før koden til funksjonen starter, og popper dem tilbake før returnering.

Hvis du også vil gi argumenter og returnere verdier kan du for eksempel bruke et fast register til dette. Det er mange måter å gjøre ting på, prøv deg frem! For en mer systematisk og komplett implementasjon av funksjoner kan du søke opp “activation records”.

2.4.8 Eksempler

Under følger et sett med eksempler for bruk av instruksjonene vi har gått gjennom.

Flytting av data

Assembly	Beskrivelse
MOV R0, #12	Last tallet 12 inn i R0.
MOV R1, R0	Last innholdet i R0 inn i R1.
MOV PC, LR	Last programtelleren med link-registeret.

Minnebehandling

Assembly	Beskrivelse
STR R0, [R1]	Lagre tallet i R0 til minneadressen i R1.
STR R0, [R1, R2]	Lagre tallet i R0 til minneadressen i R1 med offset R2.
LDR R0, [R1]	Hent innhold i minneadressen i R1 og legg i R0.
LDR R0, [R1, R2]	Hent innhold i minneadressen i R1 med offset R2 og legg i R0.
PUSH {R1,R2,R3}	Push R1, R2 og R3 til stacken.
POP {R1,R2,R3}	Pop fra stacken og legg i R1, R2 og R3.
LDR R1, =variabel	Hent minneadressen til variabel og legg i R1.
LDR R0, [R1]	Hent innholdet til variabel og legg i R0.
STR R0, [R1]	Lagre tallet i R0 til variabel.

Bit-behandling

Assembly	Beskrivelse
LSL R0, R1, #2	Shift R1 2 plasser til venstre og lagre i R0.
AND R0, R0, R1	Gjør et bitwise AND på R0 og R1, lagre i R0.
ORR R0, R0, #0b111	Gjør et bitwise OR på R0 og det binære tallet 111, lagre i R0.
MVN R0, #0	Inverter alle bit-ene i tallet 0, lagre i R0.

Aritmetikk

Assembly	Beskrivelse
MUL R0, R0, R1	Lagre R0 ganger R1 i R0.
ADD R0, R0, R1	Lagre R0 pluss R1 i R0.
SUB R0, R0, R1	Lagre R0 minus R1 i R0.

Kontrollflyt

Assembly	C-ekvivalent
Loop: SUB R0, R0, #1 BNE Loop	do { R0--; } while (R0 != 0);
CMP R0, #10 BNE False MOV R0, #1 B Endif False: MOV R0, #2 EndIf:	if (R0 == 10) { R0 = 1; } else { R0 = 2; }
PUSH LR BL funksjon POP LR ... funksjon: ... MOV PC, LR	funksjon(); ... void funksjon(void) { ... return; }
MOV R0, #0 CheckCondition: CMP R0, #10 BPL EndFor ... ADD R0, R0, #1 B CheckCondition EndFor:	for (i = 0; i < 10; i++) { ... }

Kapittel 3

Utviklingsmiljø

Dette kapitlet presenterer utviklingsmiljøet for øvingsopplegget. Seksjon 3.1 til og med 3.2 beskriver oppsett av utviklingsmiljø med Simplicity Studio i Windows steg for steg. Deretter presenterer seksjon 3.4 noen potensielle problemer og hvordan disse eventuelt kan løses. Seksjon 3.5 beskriver hvordan rammeverket for assembly skal brukes. Til slutt gir seksjon 3.6 en innføring i rammeverket som skal brukes i øving 3.

Hvis du ikke vil bruke Simplicity Studio er det også mulig å bruke en vanlig teksteditor og kommandolinjen. Prosessen for dette er beskrevet i seksjon 3.3. Stegene med installasjon og oppsett av Simplicity Studio kan da hoppes over.

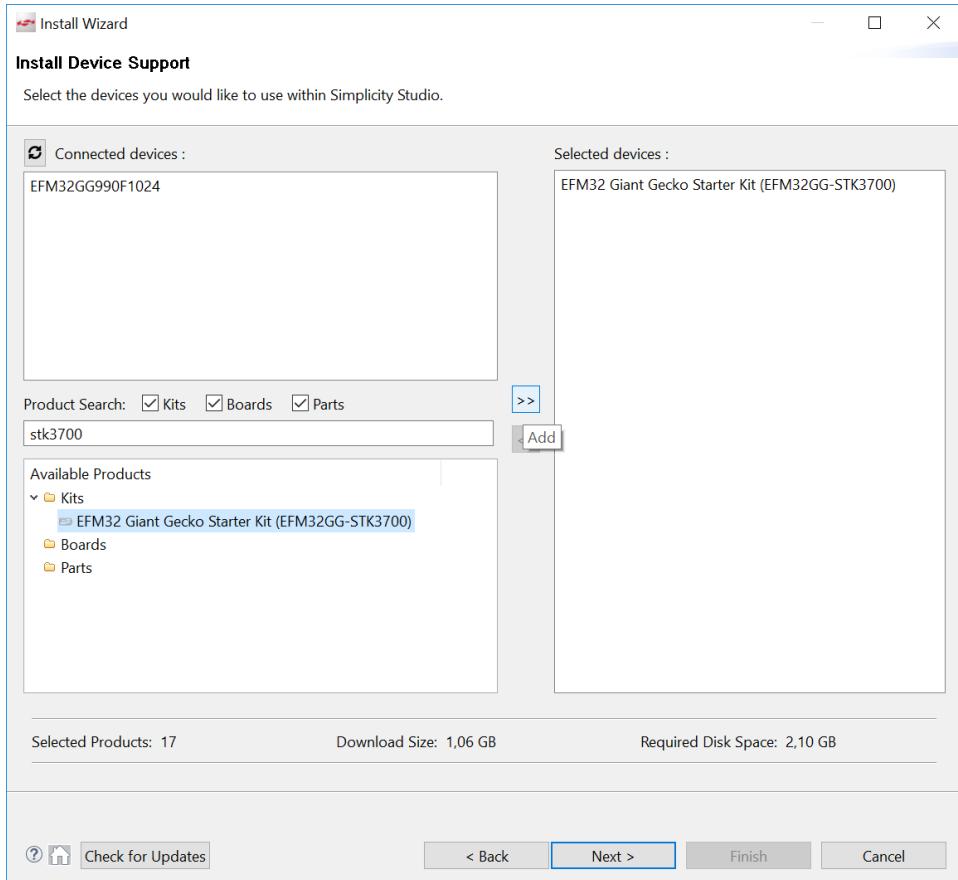
3.1 Installasjon av Simplicity Studio

Simplicity Studio er en slags samlepakke for dokumentasjon, software og firmware fra Silicon Labs. Simplicity Studio 4 kan lastes ned for Windows, Mac og Linux fra Silicon Labs sine hjemmesider¹.

NB: Denne guiden baserer seg på versjon 4 av Simplicity Studio.

Under installasjonen vil du få spørsmål om hvilken metode du vil bruke for å installere. Velg **Install by Device**. I det neste vinduet (figur 3.1) skriv ”stk3700“ i tekstboksen under ”Product Search“, velg **EFM32 Giant Gecko Starter Kit (EFM32GG-STK3700)**. Trykk deretter på knappen » (Add) som vist i figur 3.1. Trykk Next og fortsett installasjonen.

¹<https://www.silabs.com/products/development-tools/software/simplicity-studio>



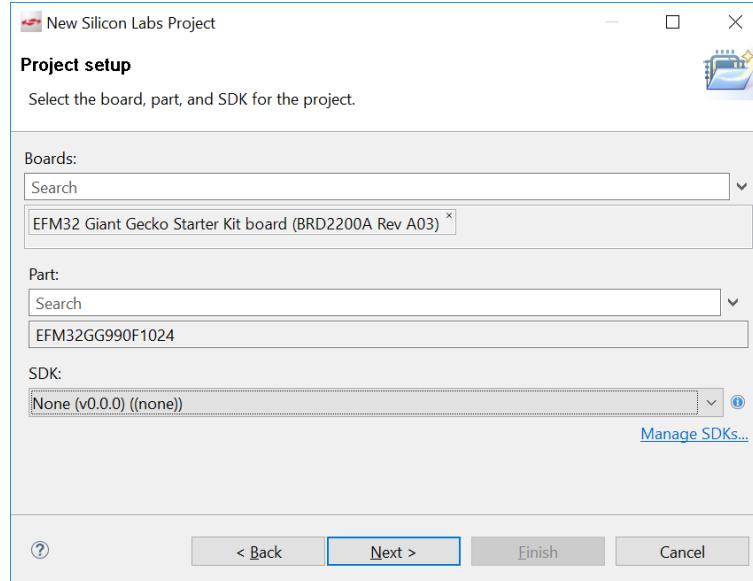
Figur 3.1: Simplicity Studio installasjon

3.2 Oppsett av Simplicity Studio

Vi skal nå sette opp Simplicity Studio for en utdelt prosjektmappe. Prosjektet som refereres til i dette oppsettet er o1 (for øving 1). Ved oppsett for øving 2 og 3 må o1 erstattes med henholdsvis o2 og o3.

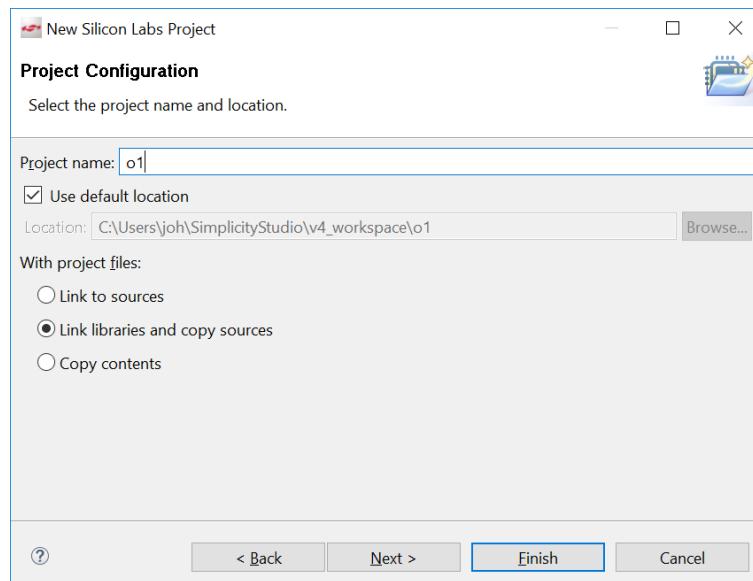
3.2.1 Nytt prosjekt

Klikk **File** → **New** → **Project...**, velg **Silicon Labs MCU Project** og trykk **Next**. I det neste vinduet (figur 3.2) velger du **EFM32 Giant Gecko Starter Kit board** og **SDK: None** (øvingene kommer med SDK inkludert).



Figur 3.2: Oppretting av nytt prosjekt

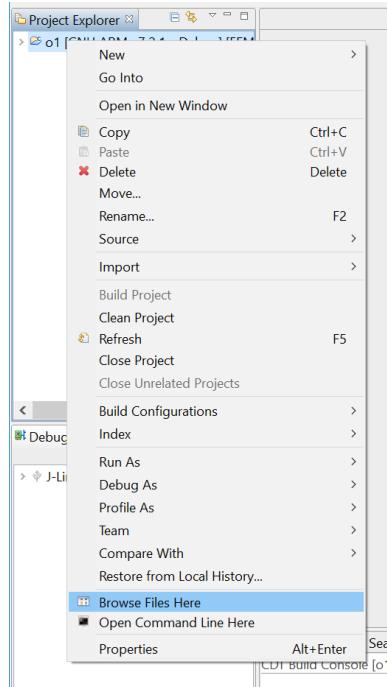
Trykk Next til du kommer til ”Project Configuration“ (figur 3.3). Skriv **o1** i **Project name**. Klikk **Finish**.



Figur 3.3: Oppretting av nytt prosjekt

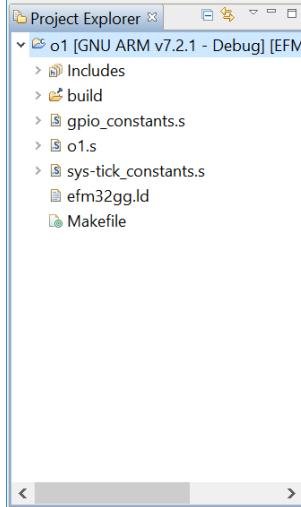
3.2.2 Kopiere inn filer

Vi skal nå kopiere inn filene fra det utdelte rammeverket inn i prosjektet. Høyreklikk på prosjektet og velg "Browse Files Here" for å åpne prosjektmappen (figur 3.4).



Figur 3.4: Kopiere filer inn i prosjektet

Kopier alle filene fra **o1**-mappen i det utdelte rammeverket til prosjekt-mappen. Du skal ende opp med en mappestruktur som vist i figur 3.5.



Figur 3.5: Kopiere filer inn i prosjektet

3.2.3 Project properties

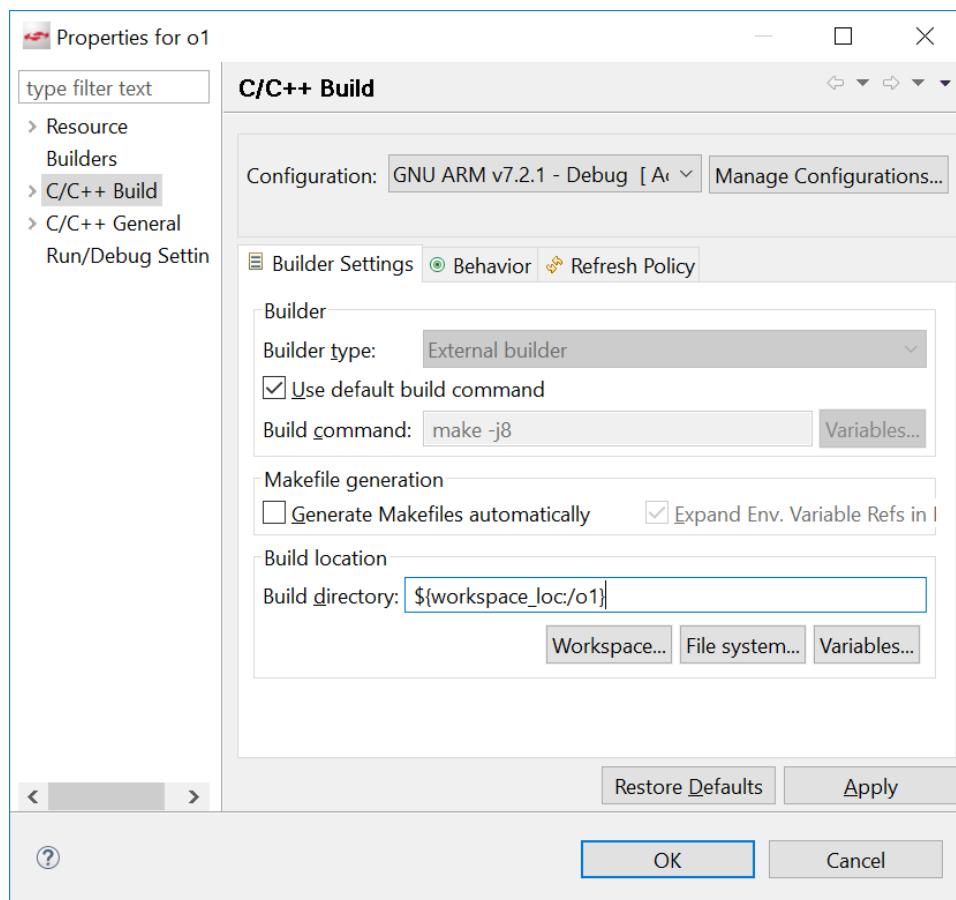
Høyreklikk på prosjektet og velg **Properties**. Velg **C/C++ Build** (figur 3.6). Uncheck **Generate Makefiles automatically**-boksen. Endre så **Build directory** til `${workspace_loc:/o1}`.

3.2.4 Bygge prosjektet

Du kan nå prøve å kompilere programmet ved å velge **Project → Build Project**. Informasjon om kompileringen og eventuelle feilmeldinger finner du i konsollvinduet (figur 3.7). Hvis kompileringen gikk feilfritt, så vil det kompilerte programmet ligge i exe-mappen og ha navnet o1.bin.

3.2.5 Flash til kit

Det kompilerte programmet flashes til kit-et fra Simplicity Studio. Høyreklikk på fila exe/o1.bin og velg **Flash to Device**. I vinduet som kommer opp (figur 3.8), trykk **Program**-knappen for å starte flashingen. Bryteren nederst til venstre på kit-et må stå på **DBG** for at flashingen skal fungere, og kit-et må være koblet til på DBG-porten (mini USB-kabelen, ikke mikro-USB). Husk at programmet må være kompilert (*Build Project*) før det kan flashes til kit-et.



Figur 3.6: Byggeinnstillinger for prosjektet

```

CDT Build Console [o1]
13:53:32 **** Build of configuration GNU ARM v7.2.1 - Debug for project o1 ****
make -j8 all
Assembling o1.s
arm-none-eabi-gcc -x assembler-with-cpp -mcpu=cortex-m3 -mthumb -I.. -I../common/CMS
Assembling gpio_constants.s
arm-none-eabi-gcc -x assembler-with-cpp -mcpu=cortex-m3 -mthumb -I.. -I../common/CMS
Assembling sys-tick_constants.s
arm-none-eabi-gcc -x assembler-with-cpp -mcpu=cortex-m3 -mthumb -I.. -I../common/CMS
Linking target: exe/o1.out
arm-none-eabi-gcc -Xlinker -Map=lst/o1.map -mcpu=cortex-m3 -mthumb -Tefm32gg.1d --sp
Creating binary file
arm-none-eabi-objcopy -O binary exe/o1.out exe/o1.bin

13:53:32 Build Finished (took 594ms)

o1 PROBLEM COUNT: 0 WARNING(S), 0 ERROR(S)

```

Figur 3.7: Konsollvindu inneholder informasjon fra kompilering

3.2.6 Start kodingen

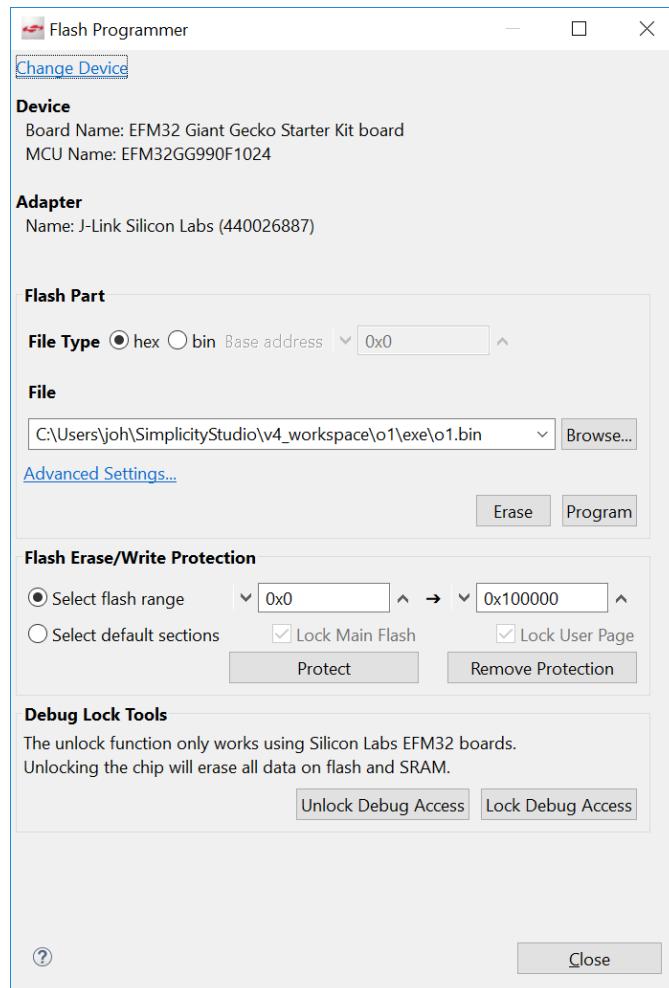
Før du begynner å kode kan det være greit å kontrollere at kompilering og flashing fungerer. Dersom ingen feilmeldinger vises i konsollen er det bare å begynne å kode. Koden skal skrives i filen som heter `o1.s`, `o2.s` eller `o3.c` for henholdsvis øving 1, 2 eller 3.

Merk at Simplicity Studio ikke integrerer like godt med C som med Java. Du vil ofte få røde streker under referanser til deler av koden som ikke finnes direkte i prosjektet. Du må derfor ikke stole så mye på editoren, og heller bygge prosjektet for å sjekke om alt er riktig. Eventuelle feil vil da vises i konsollen. For å få vekk de røde strekene kan du velge Window → Preferences, navigere til C/C++ → Code Analysis, og un-checke de to nederste (“Symbol is not resolved” og “Type cannot be resolved”).

3.3 Alternativ til Simplicity Studio: kommandolinje

Det er også mulig å bruke kommandolinjen istedenfor Simplicity Studio. Naviger til mappa med koden din (`o1`, `o2` eller `o3`) og kjør kommandoene `make` for å bygge/kompilere prosjektet. For å flashe til kit bruker du kommandoen `make flash`. Merk at for å flashe til kortet må du likevel installere Simplicity Studio som inneholder flasheverktøyet.

NB: om du bruker Ubuntu 18.04: Toolchainen som kommer i pak-



Figur 3.8: Flash til kit fra Simplicity Studio

ken gcc-arm-none-eabi fungerer ikke. Du kan enten bruke toolchainen som kommer med Simplicity Studio eller laste ned ny pakke fra PPA².

3.4 Hvis noe går galt

Ved å følge dette oppsettet på en Windows-maskin burde alt gå som det skal, men det er alltid mulig at ting går galt. Ved problemer med installering av et av programmene er det ofte best å google etter en løsning eller spørre din studass. Men hvis noe går galt ved kompilering og/eller opplasting av program til kitet kan justeringer av Makefile-en som regel løse problemet. Merk at hver øving har sin egen Makefile, så eventuelle forandringer må gjøres i alle filene. Øving 1 og 2 har identiske Makefile-er, men i øving 3 er det noen forskjeller.

3.4.1 Could not open flash programmer

Om du får feilmeldingen "Could not open flash programmer" når du velger Flash to Device, betyr det som regel at Flash Programmer-vinduet allerede er oppe et sted. Det ligger antakeligvis bak et annet vindu. Lukk Flash Programmer-vinduet og prøv igjen.

3.5 Rammeverk for assembly

I rammeverket for øving 1 og 2 følger det med to filer, `gpio_constants.s` og `sys-tick_constants.s`, som holder en rekke konstantverdier for henholdsvis GPIO og SysTick. Disse konstantene inneholder adresser til registrene som ble presentert i seksjon 2.1 og 2.3. På bakgrunn av at `MOV`-instruksjonen kun har plass til konstanter lavere enn 2^8 (se seksjon 2.4.2), kan vi ikke laste inn de fleste av disse med `MOV`. I stedet kan vi bruke `LDR`-instruksjonen med "`=`"-snarveien som beskrevet i seksjon 2.4.3. Eksempelvis kan konstanten `GPIO_BASE` lastes inn i register 0 på følgende måte:

```
LDR R0, =GPIO_BASE
```

Det er også mulig å gjøre konstant aritmetikk på konstantverdien. For eksempel kan man laste klokkefrekvensen delt på 10 på følgende måte og assembleren vil gjøre utregningen for oss:

```
LDR R0, =FREQUENCY/10
```

²<https://launchpad.net/ team-gcc-arm-embedded/+archive/ubuntu/ppa>

Adresse	Verdi
foo	0x20000000
main	LDR R1, =foo
	LDR R0, [R1]
...	...

Adresse	Verdi
0x20000000	42
0x20000004	0
0x20000008	0
...	...

(a) Programminne

(b) Dataminne

Tabell 3.1: Lagring og aksess av en heap-variabel

På bunnen av filene ligger det en rekke hjelpekonstanter som kan være relevant for øvingene. Disse er alle små nok til å passe direkte inn i instruksjoner som tilatrer direkte-verdier. For eksempel kan pin-indeks-konstanten for LED-en (definert som `LED_PIN = 2` i `gpio_constants.s`) brukes i shift-instruksjonen som i linje 1 i følgende kode:

```
1 | LSL R0, R1, #LED_PIN
2 | LSL R0, R1, #2
```

Instruksjon 1 og 2 over er ekvivalente.

3.5.1 Variabler

I øving 2 skal vi aksessere globale heap-variabler³ i assembly. Dette kan vi gjøre med LDR-instruksjonen, men først må vi forstå hvordan heap-variabler er lagret i minnet. Tabell 3.1 viser minnet delt opp i programminne og dataminne.⁴ I tabell 3.1a vises et eksempel på hvordan programminnet kan se ut for et gitt program. Verdi-kolonnen representerer innholdet til minnet: Dette er det binære programmet (her representert i tekstlig form) kopiert over i minnet til datamaskinen. (Adressene er her representert av labels.) Innholdet i programminnet kan ikke endres, og globale heap-variabler må derfor være referanser til posisjoner i dataminnet. I tabellen kan vi se at verdien på adressen `foo` inneholder adressen `0x20000000`. Her er `foo` en heap-variabel.

³En “heap”-variabel er en variabel med en konstant adresse og potensielt ubegrenset levetid. Dette i motsetning til stack-variabler som brukes som lokale (midlertidige) variabler i funksjoner.

⁴Selv om den klassiske Von Neumann-arkitekturen ikke skiller mellom program- og data-minne, og moderne datamaskiner ofte plasserer program og data i det samme fysiske minnet, setter vi et klart skille mellom program og data. Den viktigste forskjellen er at vi ikke kan endre på programminnet. Tidligere forsøkte man å skrive programmer som kunne endre seg selv. Man oppdaget fort at slike programmer var utrolig vanskelige å skrive og vedlikeholde, og dermed ble skrivbare programminner stadig mer uvanlig.

I dataminnet, vist i tabell 3.1b, ser vi at adresse 0x20000000 inneholder verdien 42, og dette er verdien til variabelen `foo`. Koden for å laste verdien ser vi i programminnet (figur 3.1a) i koden som starter på adresse `main`. Vi laster altså først verdien til referansen (0x20000000) inn i `R1` (første linje), deretter bruker vi referansen til å laste verdien til `foo` inn i `R0` (andre linje). For å heller skrive inneholdet i `R0` til variabelen, kan vi endre op-koden på den andre linjen til “STR”.

3.6 Rammeverk for C

Akkurat som i assembly utfører vi i C oppgaver på mikrokontrolleren ved å skrive til støtteregistre. Der vi i assembly må bruke flere instruksjoner for å regne ut adressen og deretter eksplisitt lese og skrive dit, kan vi i C benytte oss av variabel-abstraksjonen. Vi kan altså sette registre opp som variabler som representerer minneadressene til sine respektive registre. I denne seksjonen antas det at du kjenner til pekere og structs som er beskrevet i seksjon 1.4.2.

3.6.1 Datatyper

Registrene i mikrokontrolleren er 32 bit store. For å representere disse i C er det mest praktisk å bruke en `unsigned 32 bits integer`. I rammeverket som leveres ut er det definert en datatype `word`. Denne datatypen er et alias for typen `uint32_t` og kan brukes til alle register-formål.

3.6.2 Introduksjon til memory mapping

Memory mapping, eller “minnekartlegging”, er å bruke variabler og datastrukturer til å representere konkrete posisjoner i datamaskinenes adresseområde. La oss starte med et eksempel der vi har et enkelt system med én pin som vi skal kunne lese og skrive til. I dette eksempeletsystemet trenger pinnen ingen konfigurasjon eller lignende, det er kun ett register og dette representerer verdien til pinnen. Som i EFM32 skriver vi til registre for å sette output og leser fra registret for å lese input. Registreret er 32 bit stort, altså et `word`, og en gitt adresse som vi kjenner. Vi kan dermed representerer registreret som en variabel med datatypen `word`. Når vi deklarerer en vanlig variabel vil komplilatoren plassere det i minnet på en eller annen adresse programmereren ikke har kontroll over. For at det skal være et memory map må variabelen heller “plasseres” på registeradressen. Vi trenger dermed en *peker* av type `word`.

Husk fra seksjon 1.4.2 at datatypen til en peker betegner datatypen den peker på. Vi vil at pekeren skal peke på et `word`, og at adressen til pekeren er registeradressen.

```

1 | volatile word* pinregister;
2 | pinregister = (word*) REGISTERADRESSE;
3 | if (vil_lese) print(*pinregister);
4 | if (vil_skrive) *pinregister = VERDI;
```

I eksempelet over deklarerer vi en peker av type `word` med specifiseringen `volatile` (se seksjon 1.4.2 for beskrivelse av `volatile`) på linje 1. På linje 2 setter vi pekeren til å peke på adressen til registeret. Vi behandler videre pekeren som hvilken som helst annen peker og bruker `**` til å dereferere innholdet ved både lesing og skriving på henholdsvis linje 3 og 4.

3.6.3 Memory mapping av datastrukturer

Minnestrukturen for GPIO og SysTick er mer kompleks enn kun ett register. Vi vil helst unngå å måtte lage én peker for hvert register, da dette ville tvunget oss til å regne ut adressene til hvert eneste. En enklere måte er å deklarere en datastruktur som direkte gjenspeiler strukturen av registrene vi vil lage memory map av, og mappe denne til base-adressen til registrene. La oss bruke GPIO-registrene, introdusert i seksjon 2.1.3, som eksempel. GPIO-registrene består av 6 sett med portregister etterfulgt av ett sett med configregister. Vi kan bruke C sin struct-funksjonalitet for å representer dette. Vi definerer først et sett med portregister til å være en struct `gpio_port_map_t` som inneholder portregistrene (CTRL, MODEL, MODEH, DOUT, osv.) i nøyaktig samme rekkefølge som de ligger i minnet. Hvert portregister representeres av en variabel med den 32 biters datatypen `word` og specifiseringen `volatile`:

```

typedef struct {
    volatile word CTRL;
    volatile word MODEL;
    volatile word MODEH;
    volatile word DOUT;
    volatile word DOUTSET;
    volatile word DOUTCLR;
    volatile word DOUTTGL;
    volatile word DIN;
    volatile word PINLOCKN;
} gpio_port_map_t;
```

På samme måte kan vi lage én struct for å representere GPIO i sin helhet. Vi definerer `gpio_map_t` til å være et array av typen `gpio_port_map_t` med nøyaktig 6 elementer, etterfulgt av samtlige config-registre med datatypen `word`:

```
typedef struct {
    volatile gpio_port_map_t ports[6];
    volatile word unused_space[10];
    volatile word EXTIPSELL;
    volatile word EXTIPSELH;
    volatile word EXTIRISE;
    volatile word EXTIFALL;
    volatile word IEN;
    volatile word IF;
    volatile word IFS;
    volatile word IFC;
    volatile word ROUTE;
    volatile word INSENSE;
    volatile word LOCK;
    volatile word CTRL;
    volatile word CMD;
    volatile word EM4WUEN;
    volatile word EM4WUPOL;
    volatile word EM4WUCAUSE;
} gpio_map_t;
```

Merk at vi må inkludere de 10 ubrukte registrene mellom port- og config-registrene. Om vi ikke hadde inkludert de, ville ikke datatypen korresponder med de fysiske posisjonene til registrene. Dermed ville alle config-registrene i vår modell bømmet på de fysiske registrene med 10 word.

Denne datastrukturen kan mappes til minnet på akkurat samme måte som i seksjon 3.6.2: Vi lager en peker av type `gpio_map_t` som vi setter til å peke på `GPIO_BASE`. Registrene kan deretter aksesseres gjennom pekeren på normal måte.

Kapittel 4

Øvinger

Dette øvingsopplegget består av å programmere en EFM32 mikrokontroller fra Silicon Labs. Det er tre øvinger der øving 1 og 2 ligger tett på maskinvaren med assemblyprogrammering, mens øving 3 omhandler C-programmering og tillater en mer abstrahert tilnærming. Her vil du få en verdifull innsikt i hvordan datamaskiner *egentlig* fungerer, og ikke minst hvordan maskinvareinteraksjon fungerer. Øvingene vil ta høyde for at dette er ditt første møte med slik lavnivå-programmering, men de er verdt å sette av mye tid til da dette er en tilvenningsprosess som kan være noe tidkrevende. I tillegg til en del spesifikk kunnskap om EFM32 skal dette øvingsopplegget gi deg følgende mer generelle erfaringer:

- Bruk av prosessorens instruksjoner til å skrive programmer.
- Bruk av minneaksess til å kommunisere med maskinvare.
- Krysskompile og kjøre programmer på et embedded system.
- Programmeringsspråket C.
- Innsikt i sammenhengen mellom høynivåprogrammering og utførelse på prosessoren.

4.1 Øving 1

Du skal skrive **assembly**-kode som **poller** knappen *PB0*. Når knappen trykkes på skal LED-en *LED0* skrus på. Når knappen slippes skal LED-en skrus av. Det skal være mulig å gjøre dette så mange ganger man vil uten å restarte maskinen. Instruksjonssettet (“assembly-språket”) for EFM32 er introdusert i seksjon 2.4.

Når du kjører koden din på kortet vil det kjøre et lite program i forkant av din assembly-kode. Dette programmet setter opp noen deler av mikrokontrolleren så du skal slippe å gjøre alt dette i din kode. Programmet gjør følgende:

1. Skrur på GPIO-klokken.
2. Setter LED-pinnen (PE2) til output.
3. Setter PB0-pinnen (PB9) til input.

Det som gjenstår for din kode er å skrive et program som leser knappen i en evig løkke og etter hver lesning setter LED-en av eller på avhengig av om knappen er trykket inn eller ikke.

Foreslått fremgangsmåte:

1. Sett opp utviklingsmiljøet (se kapittel 3).
2. Skriv et program som skrur på LEDen og sjekk at det fungerer (se seksjon 2.1).
3. Skriv om programmet til å oppfylle kravene til øvingen.

4.2 Øving 2

Du skal skrive **assembly**-kode som bruker SysTick-timeren for å telle **tiendedels** sekunder. Dette skal brukes til å implementere en stoppeklokke. Klokka skal begynne å telle når knappen *PB0* trykkes inn, og stoppes når samme knapp trykkes på igjen. For hvert **sekund** som går mens klokka teller skal LEDen *LED0* toggles. Når maskinen starter skal klokka ikke begynne å telle før knappen trykkes inn.

Når du kjører koden din på kit-et vil det kjøre et lite program i forkant av din assembly-kode. Dette programmet setter opp noen deler av mikrokontrolleren så du skal slippe å gjøre alt dette i din kode. Programmet gjør følgende¹:

1. Skrur på GPIO-klokken.
2. Setter LED-pinnen (PE2) til output.
3. Setter PB0-pinnen (PB9) til input.
4. Enabler interrupt vektoren som vil kalles av knappen.

For å telle både tiendedeler, sekunder og minutter kan du bruke en variabel for hver og øke dem på følgende måte. Tiendededelene økes ved hvert eneste interrupt. Sekundene økes hver gang tiendededelene har telt til 10, tiendededelene kan dermed nullstilles. Minuttene økes hver gang sekundene har telt til 60, sekundene kan dermed også nullstilles.

Programmet som kjører i forkant av koden din setter opp et interrupt som viser minutter, sekunder og tiendedeler på skjermen i formatet **mm-ss-t**. For at dette skal fungere må du bruke en gitt minneposisjon som variabel for tiendels sekunder, sekunder og eventuelt minutter. Variablene som inneholder disse minneposisjonene er deklarert globalt som **tenths**, **seconds** og **minutes**, og kan nås fra din kode uten behov for imports eller lignende. Innholdet av disse variablene vil hele tiden vises på skjermen (skjermen oppdateres 20 ganger i sekundet). Se seksjon 2.4.8 for eksempel på variabelaksess i assembly.

Pass på at det er noe som kjører “i bakgrunnen” mens det ventes på interrupt. Dette kan for eksempel være en evig løkke.

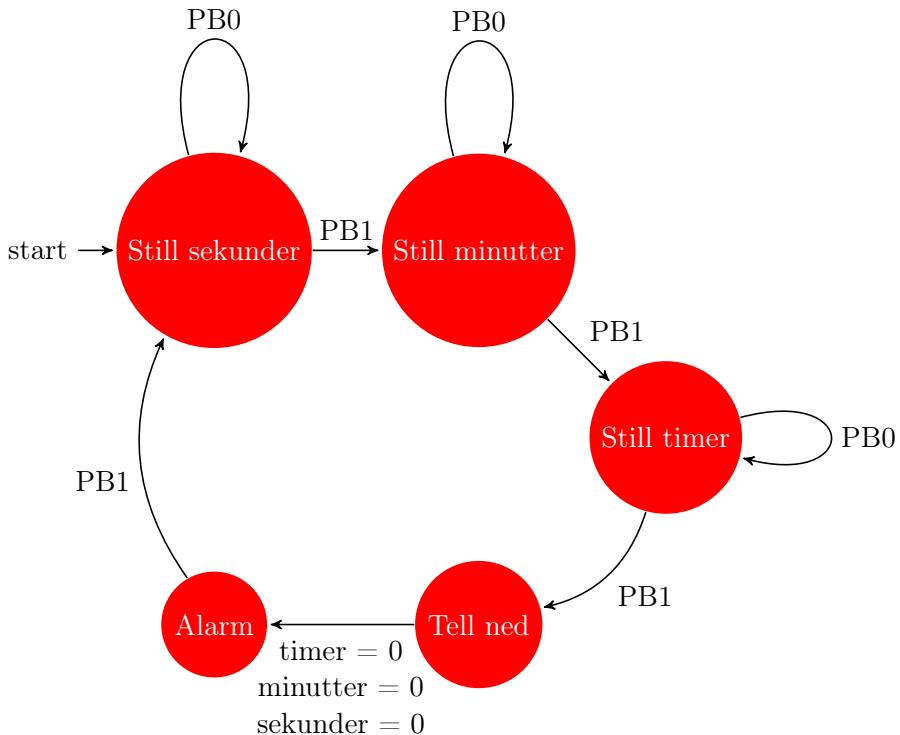
¹Merk at i motsetning til i øving 1 er ikke konfigurering av interrupt inkludert i det utvleverte oppstartsprogrammet.

Foreslått fremgangsmåte:

1. Skriv tallet 1 til `tenths`-variabelen og verifiser at 00-00-1 vises på skjermen.
2. Skriv kode for å telle tiendedels sekunder. Hver gang koden blir kalt skal `tenths`-variabelen øke med 1.
3. Plasser koden for dette i din SysTick interrupt handler.
4. Skriv kode for å sette opp Sys-Tick interrupt, sjekk på skjermen at tallet økes ca hvert tiendedels sekund.
5. Oppdater koden til å toggle LED-en ved hvert sekund basert på hvor mange tiendedels sekunder som har gått.
6. Oppdater koden til å også telle sekunder og minutter.
7. Skriv kode for å sette opp interrupt for knappen (se seksjon 2.2.3).
8. Skriv en interrupt handler for knappen som starter/stopper klokka ved knappetrykk.

4.3 Øving 3

Du skal skrive et program i C som skal utgjøre en alarmklokke. Alarmklokken skal kunne stilles inn til et antall sekunder, minutter og timer og skal deretter telle nedover. Når nedtellingen treffer 0 skal LED-en *LED0* settes på for å simulere en alarm. Alarmklokken skal fungere som den endelige tilstandsmaskinen i figur 4.1.



Figur 4.1: Endelig tilstandsmaskin for alarmklokken

Alarmklokken skal starte i tilstanden **Still sekunder**. I de tre første tilstandene (still sekunder, minutter og timer) fører et trykk på knappen *PB0* at henholdsvis sekunder, minutter eller timer økes med 1, og et trykk på *PB1* fører til at maskinen går til neste tilstand. Når maskinen går i tilstanden **Tell ned** skal alarmklokken begynne å telle nedover. Når alarmklokken har telt ned til 0 skal maskinen gå over til tilstanden **Alarm** der nedtellingen stopper og alarmen går ved at LED-en skrues på. Når knappen *PB1* trykkes på, skal alarmen skrus av og maskinen skal gå tilbake til tilstanden **Still sekunder**. Det er viktig for valideringen at LED-en ikke er aktiv på noe annet

tidspunkt enn under alarm, og at tilstandsmaskinen ellers er implementert som beskrevet her. Knappen *PB1* er på port B pin 10.

I filen som leveres ut er det allerede lagt inn et kall til en funksjon `init`. Denne funksjonen gjør følgende²:

- Skrur på GPIO-klokken.
- Enabler interrupt-vektorene som vil kalles av knappene.
- Enabler LCD-skjermen.

For å skrive til LCD-skjermen kan du bruke funksjonen `lcd_write` (prototypen til denne funksjonen finnes i `o3.h`). For å konvertere timer, minutter og sekunder til en string kan du bruke funksjonen `time_to_string` som legges ved i `o3.c`.

Foreslått fremgangsmåte:

- Skriv et program (i C) som skrur på LED-en, sjekk at det fungerer. Bruk av GPIO i C beskrives i seksjon 3.6 og programmeringspråket C er introdusert i seksjon 1.4.
- Skriv et program som skrur på LED-en når en knapp trykkes inn (med interrupts).
- Implementér øving 2 i C.
- Modifiser implementasjonen til å oppfylle kravene for øving 3.

²Merk at i motsetning til øving 1 og 2 settes ingen av pinnene til input eller output. Se kapittel 2.1.6.

Tillegg

Tillegg A

Instruksjonssett

Instruksjonssettet oppsummeres her. Full beskrivelse finnes i seksjon 2.4.

Symbol	Forklaring
Rx, Ry, Rz	Et register (R0, R1, R2...).
'direkteverdi'	Direkteverdi (#0, #1, #2...).
'verdi'	Direkteverdi eller register (Rx).
'betingelse'	En betingelse (EQ, NE, PL... (se 2.4.6)).
'label'	En string som representerer en adresse.
'reg-liste'	En liste med registre f.eks. {R1,R2,R3}.

Tabell A.1: Forklaring av symboler

Betingelse	Navn	Krav til condition flag
EQ	Equal	Z = 1
NE	Not equal	Z = 0
MI	Minus/negative	N = 1
PL	Plus/positive eller 0	N = 0

Tabell A.2: Betingelser

OP-kode	Operander	Beskrivelse
MOV	Rx, 'verdi'	Legg 'verdi' i Rx.
STR	Rx, [Ry]	Lagre Rx på minnelokasjon Ry.
STR	Rx, [Ry, Rz]	Lagre Rx på minnelokasjon Ry+Rz.
LDR	Rx, [Ry]	Hent innhold i minnelokasjon Ry og legg i Rx.
LDR	Rx, [Ry, Rz]	Hent innhold i minnelokasjon Ry+Rz og legg i Rx.
LDR	Rx, ='label'	Hent innhold fra minnelokasjon 'label' og legg i Rx.
LDR	Rx, ='direkteverdi'	Last direkteverdi (opptil 32 bits) og legg i Rx.
PUSH	'reg-liste'	Push registrene i 'reg-liste' til stacken
POP	'reg-liste'	Pop fra stacken og legg i registrene i 'reg-liste'
LSL	Rx, Ry, 'verdi'	Shift Ry 'verdi' plasser til venstre, lagre i Rx.
AND	Rx, Ry, 'verdi'	Gjør et bitwise AND på Ry og 'verdi', lagre i Rx.
ORR	Rx, Ry, 'verdi'	Gjør et bitwise OR på Ry og 'verdi', lagre i Rx.
EOR	Rx, Ry, 'verdi'	Gjør et bitwise Exclusive OR på Ry og 'verdi', lagre i Rx.
MVN	Rx, 'verdi'	Gjør et bitwise NOT på 'verdi', lagre i Rx.
MUL	Rx, Ry, Rz	Lagre i Ry ganger Rz i Rx.
ADD	Rx, Ry, Rz	Lagre i Ry pluss Rz i Rx.
SUB	Rx, Ry, Rz	Lagre i Ry minus Rz i Rx.
CMP	Rx, 'verdi'	Setter condition flag-ene basert på sammenligning av Rx og 'verdi'.
B	'label'	Branch til instruksjonen med adresse 'label'.
B'betingelse'	'label'	Branch til 'label' dersom 'betingelse' er oppfylt.
BL	'label'	Branch til 'label', lagre returadressen i LR (Link Register). Merk at dette skriver over forrige LR!
BX	Rx	Branch til adressen i register Rx.
MOV	PC, Rx	Denne instruksjonen brukes til å returnere fra interrupt. Flytt innholdet i Rx inn i programtelleren. I praksis det samme som en ukondisjonell branch til et register i stedet for label.

Tabell A.3: Instruksjoner

Tillegg B

Prosessor-registre

Prosessor-registrene vises i tabell B.1. General purpose-registrrene har ingen spesiell betydning for prosessoren og kan brukes til hva som helst. Stack Pointer (SP) peker på toppen av stacken. Den oppdateres automatisk av push- og pull-instruksjoner. SP kan brukes manuelt hvis man for eksempel vil poppe verdier fra stacken uten å ta vare på verdiene. Link Register (LR) inneholder adressen som funksjonen som kjører skal returnere tilbake til. Den settes automatisk når BL-instruksjonen brukes til funksjonskall. Program Counter (PC) inneholder adressen til neste instruksjon som skal utføres. Den vedlikeholdes automatisk av prosessoren. Branching kan oppnåes ved å skrive direkte til PC-registret, men dette er i praksis det samme som å bruke en branch-instruksjon.

Når et register har et alias kan både navnet og aliaset brukes i koden.

Navn	Alias	Bruksområde
R0		General purpose
R1		General purpose
R2		General purpose
R3		General purpose
R4		General purpose
R5		General purpose
R6		General purpose
R7		General purpose
R8		General purpose
R9		General purpose
R10		General purpose
R11		General purpose
R12		General purpose
R13	SP	Stack Pointer
R14	LR	Link Register
R15	PC	Program Counter

Tabell B.1: Prosessor-registrene

Tillegg C

Støtteregistre

Støtteregistrene oppsummeres her. SysTick-registrene beskrives i detalj i seksjon 2.3 og GPIO-registrene i seksjon 2.1.

Offset	Navn	Beskrivelse
0B	SYSTICK_CTRL	Control and Status Register
4B	SYSTICK_LOAD	Reload Value Register
8B	SYSTICK_VAL	Current Value Register
12B	SYSTICK_CALIB	Calibration Register

Tabell C.1: SysTick-registrene. Offset gjelder fra `SYSTICK_BASE`.

Offset	Navn	Beskrivelse
0B	CTRL	Control
4B	MODEL	Mode low
8B	MODEH	Mode high
12B	DOUT	Data output
16B	DOUTSET	Set data output
20B	DOUTCLR	Clear data output
24B	DOUTTGL	Toggle data output
28B	DIN	Data input
32B	PINLOCKN	Lock pin

Tabell C.2: GPIO portregistrene. Offset gjelder fra starten av porten.

Offset	Navn	Beskrivelse
256B	EXTIPSELL	External Interrupt Port Select Low Register
260B	EXTIPSELH	External Interrupt Port Select High Register
264B	EXTIRISE	External Interrupt Rising Edge Trigger Register
268B	EXTIFALL	External Interrupt Falling Edge Trigger Register
272B	IEN	Interrupt Enable Register
276B	IF	Interrupt Flag Register
280B	IFS	Interrupt Flag Set Register
284B	IFC	Interrupt Flag Clear Register
288B	ROUTE	I/O Routing Register
292B	INSENSE	Input Sense Register
296B	LOCK	Configuration Lock Register
300B	CTRL	GPIO Control Register
304B	CMD	EM4 Wake-up Clear Register
308B	EM4WUEN	EM4 Wake-up Enable Register
312B	EM4WUPOL	EM4 Wake-up Polarity Register
316B	EM4WUCAUSE	EM4 Wake-up Cause Register

Tabell C.3: GPIO config-registrene. Offset gjelder fra `GPIO_BASE`.

Tillegg D

Eksterne komponenter

Tabellen under viser på hvilken port og pin knappene og lysene på kit-et er på.

Navn	Port	Pin
LED0	E	2
PB0	B	9
PB1	B	10

Register

bit, 8
bitshift, 9
bitwise, 9
byte, 8

kontrollflyt, 24, 47

mikrokontroller, 4

pin, 28
processor-register, 23, 45

støtteregister, 7

word, 8