

UiO : Department of Informatics
University of Oslo

Forelesning 6a - IN2060

Programmering via ARM assemblerspråk

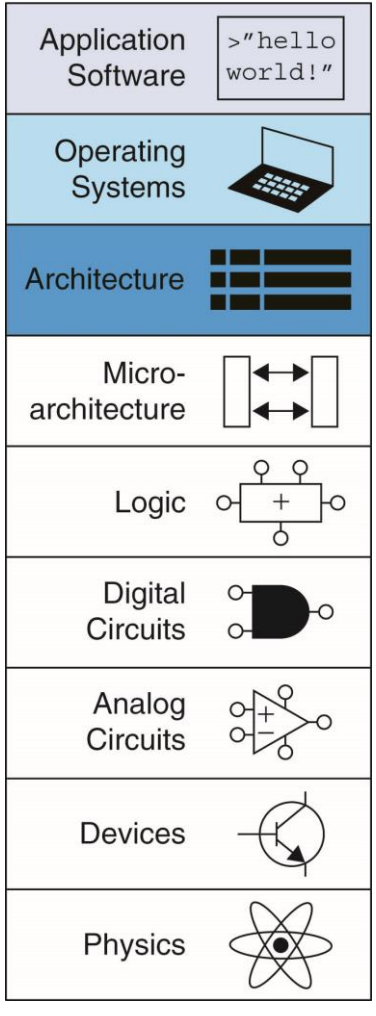


Datamaskinarkitektur Kapittel 6 del 1

Til nå: digitale porter → digitale byggeblokker.

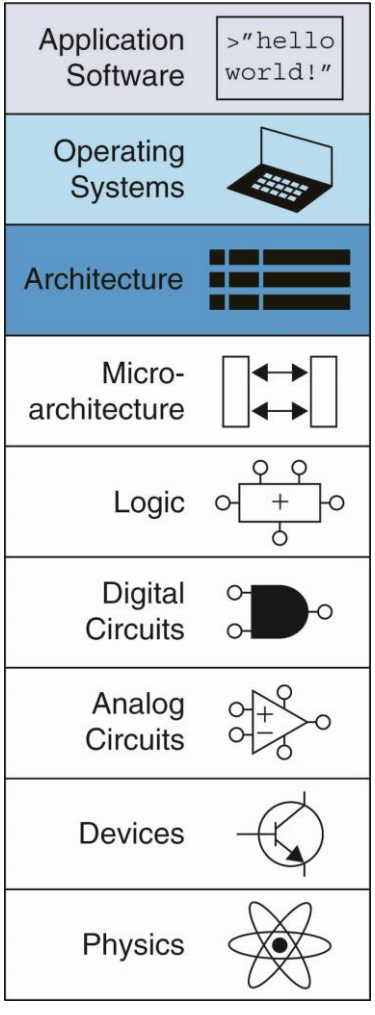
Nå skal vi hoppe opp et abstraksjonsnivå og se på hva en datamaskinen tilbyr oss av funksjonalitet.

- Del 1: ARM assemblerspråk
  - Introduksjon til datamaskinarkitektur
  - Bli kjent med *instruksjoner* og *registeret (!)*
  - Se på noen høynivå konstruksjoner
    - Hvordan man programmere med assemblerspråk.
- Del 2: Maskinkode (neste forelesning)
  - Hvordan instruksjoner kodes til 0 og 1



Datamaskinarkitektur

- Forskjellen mellom arkitektur og mikroarkitektur?
  - **Arkitektur**: Programmerens perspektiv av en datamaskin
    - Definert av ***instruksjoner*** og ***registeret***
    - Definerer hvordan datamaskinen fungerer «utenifra»
  - **Mikroarkitektur**: En spesifikk implementasjon av en type arkitektur i hardware (Kapittel 7)

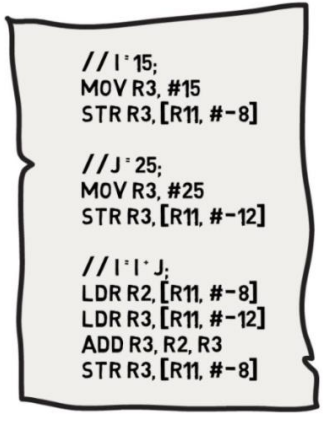


Instruksjoner

Kommandoene som datamaskinen faktisk utfører.

- **Assemblerspråk**: Instruksjonene på et format som vi kan forstå (human-readable)
  - De første programmeringsspråk lignet på assemblerspråk (Fortran)
- **Maskinspråk**: computer-readable, kun 1'ere og 0'ere (neste forelesning)

Høynivå programmeringsspråk: Et abstraksjonslag over arkitekturen som gjør at det er lettere å programmere (fjerner unødvendige detaljer)



ASSEMBLY LANGUAGE



MACHINE CODE

ARM Arkitekturen

- Utviklet på 80-tallet av *Advanced RISC Machines* (idag ARM Holdings)
- Mer enn 350 milliarder ARM-baserte brikker er produsert (2026)
- Nesten alle mobiltelefoner og nettbrett er basert på ARM prosessorer (stadig større rolle i PC-er og datasentre).
- **100 % av den tilkoblede verdensbefolkningen** bruker produkter basert på ARM-teknologi. (74%)
- Også brukt i servere, roboter, biler, spillmaskiner, osv. Verdens 9. raskeste superdatamaskin (Fugaku). (Raskest i 2020)

  – Utbredelsen gjør ARM arkitekturen til et naturlig valg for IN2060.

Arkitektur design

Hvilke designvalg ligger bak en datamaskinarkitektur?

Design prinsipper, som beskrevet av *Hennesy* og *Patterson*:

1. “Regularity supports design simplicity”
2. Få de vanligste instruksjonene til å kjøre raskt
3. Kompakt design er raskere
4. Et godt design krever gode kompromisser

Instruksjon: Addisjon

C kode

```
a = b + c;
```

ARM assemblerkode

```
ADD a, b, c
```

- **ADD:** navn på instruksjonen (*mnemonic*)
- **b, c:** “inn”-variabler (*source operands*)
- **a:** hvor det skal lagres (*destination operand*)

Instruksjon: Subtraksjon

C kode

```
a = b - c;
```

ARM assemblerkode

```
SUB a, b, c
```

- **SUB:** navn på instruksjonen (*mnemonic*)
- **b, c:** “inn”-variabler (*source operands*)
- **a:** hvor det skal lagres (*destination operand*)

Designprinsipp 1

```
ADD a, b, c
SUB a, b, c
```

"Regularity supports design simplicity"

- Formatet på instruksjonene er like
- Like antall variabler (two sources and one destination)
- Dette gjør at det er enklere å lage hardware som skal tolke og behandle disse instruksjonene.

Hvordan løse større utrykk?

C kode

```
a = b + c - d;
```

ARM assemblerkode

```
ADD t, b, c; t = b + c
SUB a, t, d; a = t - d
```

Mer komplekse utrykk løser vi ved å dele opp i flere små instruksjoner

Designprinsipp 2 - “Make the common case fast”

Å dele opp større instruksjoner til flere små, er typisk for en RISC arkitektur.

- ARM er et *Reduced Instruction Set Computer* (RISC)
  - Ønsker *få* og *enkle* instruksjoner
  - Kompliserte (sjeldne) instruksjoner løses med flere enkle instruksjoner.
- Andre arkitekturer, som Intel’s x86, er *Complex Instruction Set Computers* (CISC)
  - Støtter store kompliserte instruksjoner som krever mer komplisert hardware.

Operand Location

Hvor vi kan hente og lagre data?

- *Registers*
- Konstanter (*immediates*)
  – som vi legger direkte inn i instruksjonen.
- Minne (tar lengre tid)

Operands: Registers

- ARM har plass til 16 variabler i registeret
- Registeret er mye raskere enn minne
- Variablene i registeret er 32 bits
  - Her antar vi at vi opererer med en “32-bits arkitektur” (64-bit ARM finnes også).

Designprinsipp 3 - “Smaller is Faster”

- ARM har et lite register.

ARM Register Set

Name	Use
R0	Argument / return value / temporary variable
R1-R3	Argument / temporary variables
R4-R11	Saved variables
R12	Temporary variable
R13 (SP)	Stack Pointer
R14 (LR)	Link Register
R15 (PC)	Program Counter

Merk:

- Stor R før nummeret.

Gjensyn med addisjon instruksjonen

C kode

```
a = b + c;
```

ARM assemblerkode

```
; R0 = a, R1 = b, R2 = c
ADD RO, R1, R2
```

- Når vi skal skrive assemblerkode må vi hovedsakelig holde oss til *register-variablene*.
- Men vi kan også benytte oss av konstanter.

Operands: Konstanter (Immediates)

- Mange instruksjoner kan bruke konstanter
  - *immediate operands*
- For eksempel: ADD and SUB
- Verdien er *direkte* tilgjengelig for instruksjonen (*immediate*).

C Code	ARM Assembly Code
	`; R0 = a, R1 = b`
`a = a + 4;`	`ADD R0, R0, #4`
`b = a - 12;`	`SUB R1, R0, #12`

Generere konstanter

- Vi kan lagre *små* konstanter til registeret ved å benytte oss av *move* (`MOV`) instruksjonen:

C Kode

```
//int: 32-bit signed word
int a = 23;
int b = 0x45;
```

ARM Assembly Kode

```
; R0 = a, R1 = b
MOV R0, #23
MOV R1, #0x45
```

  - Kan bruke heksadesimale tall direkte (`0x45`)
  - Men vi har ikke støtte for store tall (< 8 bits)
  - `MOV` kan også bruke registeret: `MOV R7, R9`

```
#0x7E000000
#0x00DC0000
#0x00008700
#0x00000065
```

Generere store konstanter

- Vi kan danne større konstanter ved å benytte oss av `MOV` og `ORR` instruksjonene:

C kode

```
int a = 0x7EDC8765;
```

ARM assemblerkode

```
# R0 = a
MOV R0,      #0x7E000000
ORR R0, R0, #0xDC0000
ORR R0, R0, #0x8700
ORR R0, R0, #0x65
```

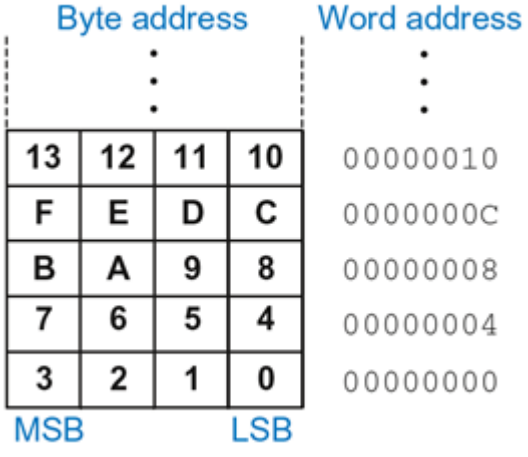
– Bygger opp 32-bit tallet med 4 x 8-bits tall.

Operands: Minne

- Vi må kunne jobbe med mer data enn de 16 variablene som registeret tilbyr.
- Løsningen er å lagre resten i minne.
- Minne har mye større plass men er *tregere*.
- Ønsker derfor å beholde de mest brukte variablene I registeret.
  - Gir raskere kjøring av kode

Minne er *byte-addressable*

- Hvert data **byte** har en unik addresse
- 32-bit word = 4 bytes,
  - Derfor øker word addressene med 4 for hvert word.



Hvordan lese fra minne

En minnelesning kaller vi for *load register* eller (`LDR`)

- Format:

```
LDR R0, [R1, #12]
```

Hva skjer her?

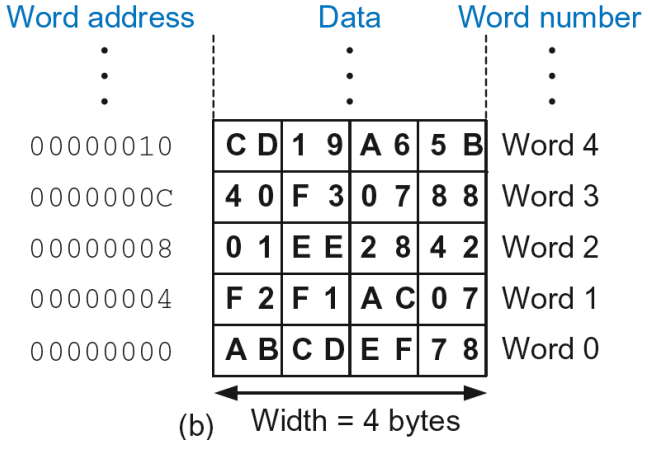
- Vi legger til *offset* 12 til baseadressen (R1)
  - Adressen blir = (R1 + 12)
- Vi laster inn data fra adressen (R1 + #12) til variabelen R0.
  - Alle variablene i registeret kan vi bruke som baseadresser.

Hvordan lese fra minne: Eksempel

La oss lese 32-bit data (word) fra minnneadresse 8 til R3

ARM assemblerkode

```
MOV R2, #0
LDR R3, [R2, #8]
```



- Her vil R3 få verdien 0x01EE2842
- Hvorfor må vi bruke R2?

Hvordan skrive til minne?

- Å skrive til minne: *store register* (`STR`)
- Format:

```
STR R0, [R1, #..]
```

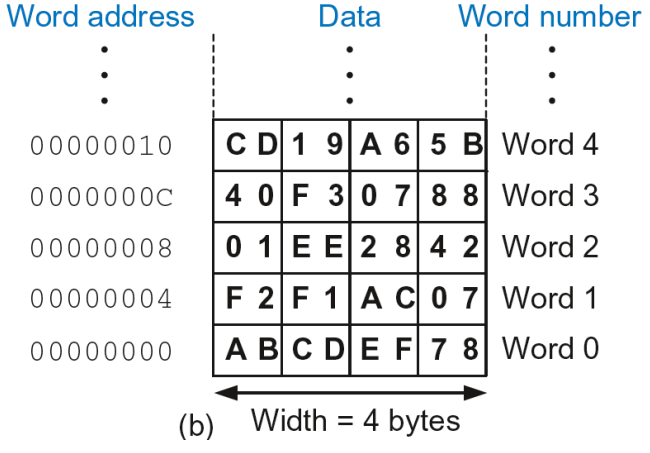
Hvordan skrive til minne: Eksempel

La oss lagre verdien i R7 til memory word 21.

- Minneaddresse = 4 x 21 = 84 = 0x54

ARM assemblerkode

```
MOV R5, #0
STR R7, [R5, #0x54]
```

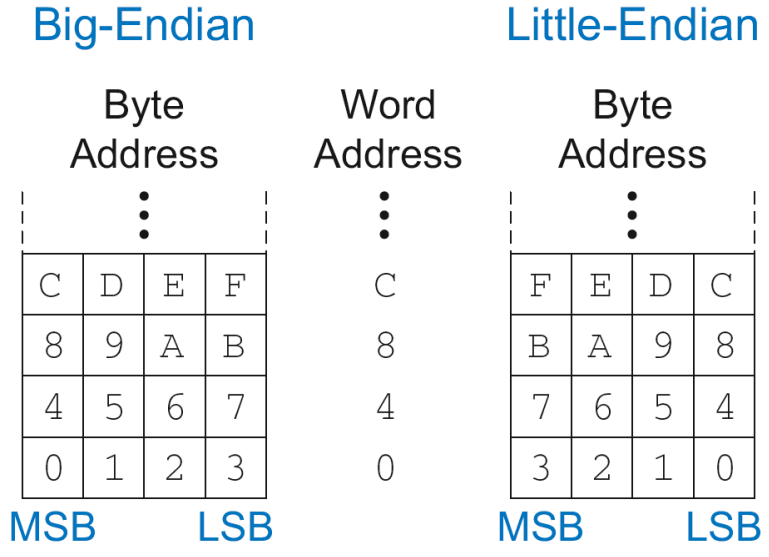


– Merk, offset kan også skrives som desimaltall (`#84`)

Minneadressering: Big-Endian & Little-Endian

Hvordan nummeres bytes innenfor et 32-bit word?

- **Little-endian**:
  - byte nummeringen starter i "little" enden (least significant)
- **Big-endian**:
  - byte nummeringen starter i "big" enden (most significant)



Viktig detalj når man deler data mellom to forskjellige datasystemer

Høynivåspråk

- Som vi skal se, det er mange detaljer å holde rede på når man programmerer direkte i assemblerspråk.
- Dette slipper man med *høynivåspråk*
  - Som f.eks, C, Java, Python osv.
- Her programmer vi på et høyere abstraksjonsnivå, og slipper å bry oss om hvordan arkitekturen utfører instruksjonene.
- Kompilator
  - Oversetter til riktig instruksjoner/maskinspråk
  - Optimaliserer også koden til den aktuelle arkitekturen

Første dataprogrammerer: Ada Lovelace (1815-1852)

- Begavet britisk matematiker
- Fulgte forelesningene til Charles Babbage
- Beskrev Babbages mekaniske, generelle datamaskin, *Den analytiske maskinen*.
- Hun hadde flere tanker om hvordan maskinen kunne mates med ulike instruksjoner for å foreta beregninger.
- Blir derfor ansett som verdens første dataprogrammerer.



Programmering: Nødvendige elementer

- Dataprosessering
- Betinget utførelse (Conditional execution)
- Branches (hopp i programkoden)
- Høynivå konstruksjoner
  - if/else statements
  - for loops
  - while loops
  - arrays
  - funksjonskall

Instruksjoner for dataprosessering

- Vi har allerede sett på *addisjon* og *subtraksjon*
- Logiske operasjoner
- Shifts / rotate
- Multiplikasjon

Logiske instruksjoner

- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
- `MVN` **(MoVe and NOT)**

Logiske instruksjoner: Eksempler

Source registers

R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Assembly code

```
AND  R3, R1, R2
ORR  R4, R1, R2
EOR  R5, R1, R2
BIC  R6, R1, R2
MVN  R7, R2
```

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

`EOR` = XOR
`BIC` = bitwise clear
`MVN` = inverter og move

Logiske instruksjoner: Eksempler 2

AND or **BIC**: Nyttig for “masking” av bits

- Eksempel: “Maske” alle unntatt den minst signifikante byte verdien
  - `0xF234012F AND 0x000000FF = 0x0000002F`
  - `0xF234012F BIC 0xFFFFFF00 = 0x0000002F`

Merk: `0x000000FF = 000..0011111111`

ORR: Nyttig for å kombinere bitfelter

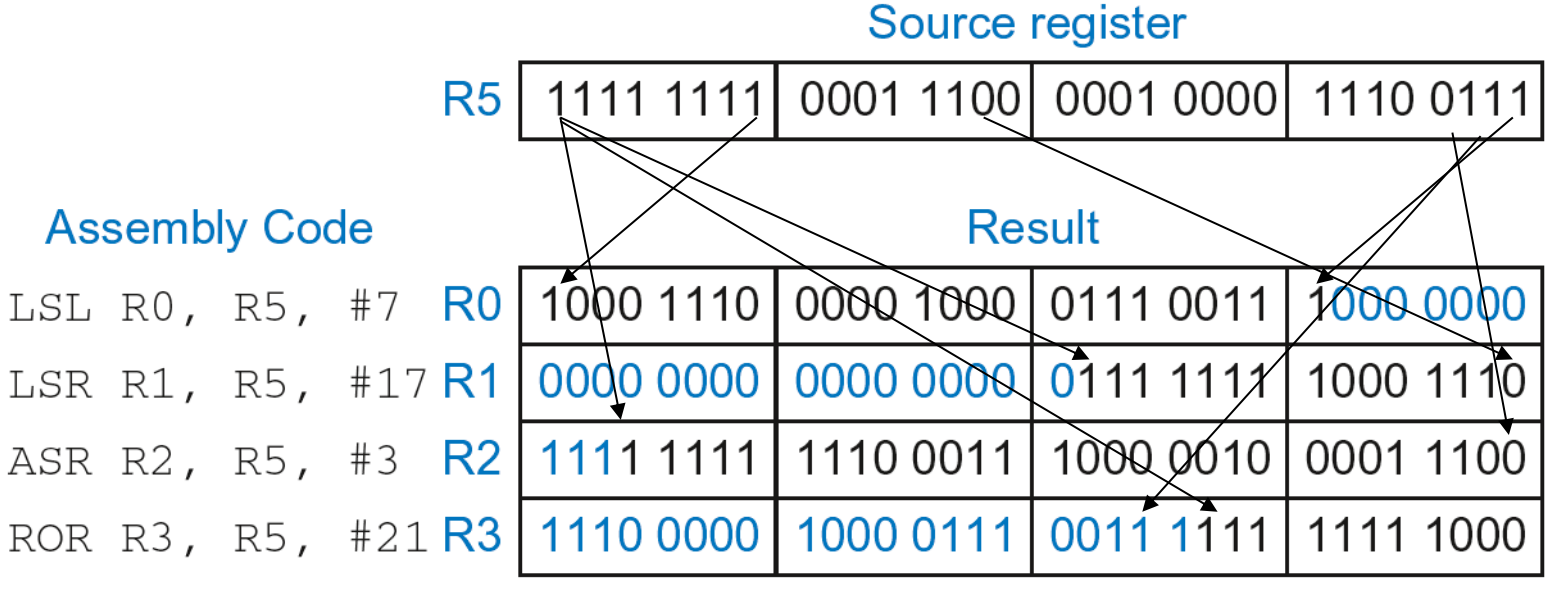
- Eksempel: Kombinere 0xF2340000 med 0x000012BC:
  - `0xF2340000 ORR 0x000012BC = 0xF23412BC`

Shift instruksjoner

- **`LSL`**: *logical shift left*
  - Eksempel: `LSL R0, R7, #5  ; R0 = R7 << 5`
- **`LSR`**: *logical shift right*
  - Eksempel: `LSR R3, R2, #31 ; R3 = R2 >> 31`
- **`ASR`**: *arithmetic shift right*
  - Eksempel: `ASR R9, R11, R4 ; R9 = R11 >>>` $\mathbf{R4_{7:0}}$
- **`ROR`**: *rotate right*
  - Eksempel: `ROR R8, R1, #3  ; R8 = R1 ROR 3`

Shift instruksjoner: Eksempler

- Immediate shift, med (5-bit immediate)
- Gir en mulighet til å flytte: 0-31



Multiplikasjon

`MUL`: 32 × 32 multiplikasjon, 32-bit resultat

- `MUL R1, R2, R3`
- R1 = (R2 x R3)31:0

`UMULL`: *Unsigned multiply long*: 32 × 32 multiplication, 64-bit result

- `UMULL R1, R2, R3, R4`
- {R1,R2} = R3 x R4 (R3, R4 unsigned)

`SMULL`: Signed multiply long: 32 × 32 multiplication, 64-bit result

- `SMULL R1, R2, R3, R4`
- {R1,R2} = R3 x R4 (R3, R4 signed)

Betinget utførelse (Conditional execution)

Det er ikke alltid vi ønsker å utføre neste instruksjon

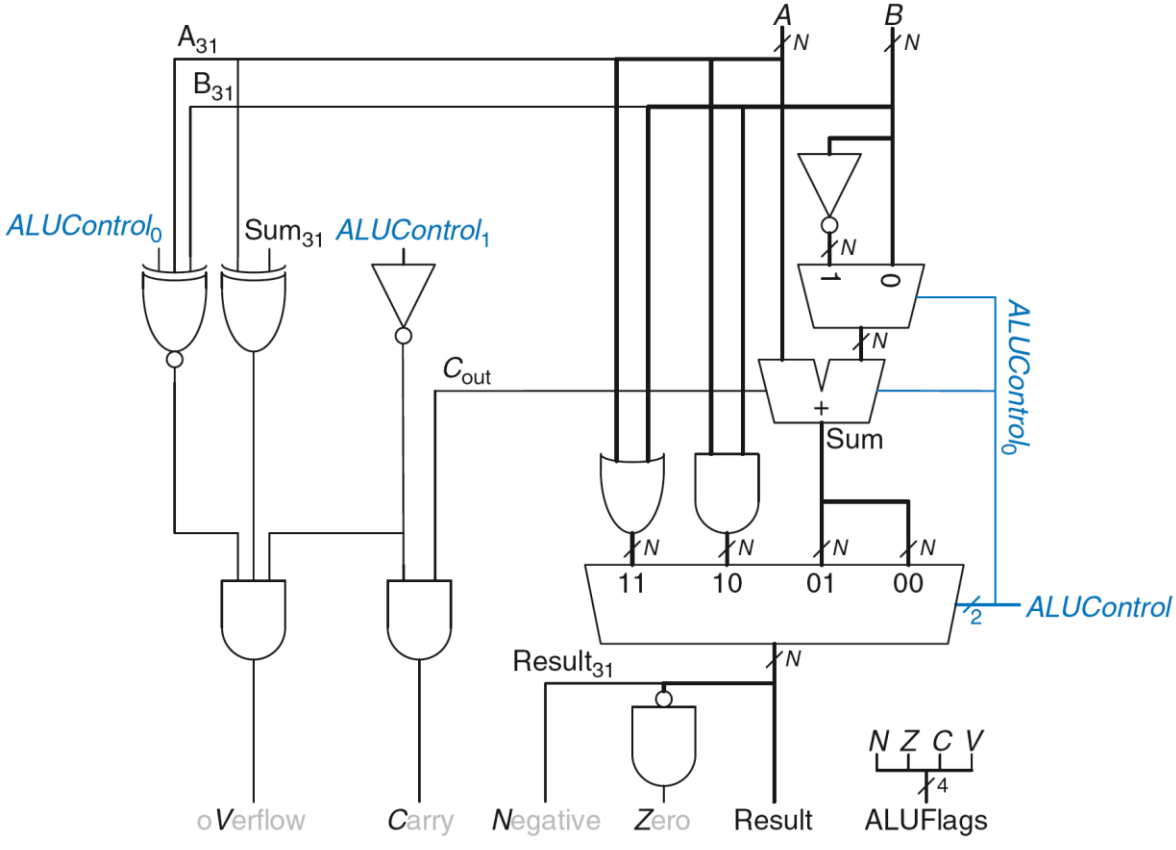
- F.eks.
  - *if/else* statements:
    - Vi ønsker kun å utføre koden dersom betingelsen er sann.
  - *Branching*:
    - Hoppe til et annet sted i koden dersom en betingelse er sann.
- ARM tilbyr dette vha. *condition flags*:
  - Blir satt av noen instruksjoner.
  - Brukes for å styre om instruksjoner skal kjøres eller ikke.

ARM Condition Flags

Flag	Name	Description
N	**N**egative	Instruksjonens resultat er negative
Z	**Z**ero	Instruksjonens resultat er null (zero)
C	**C**arry	Instruksjonen ga *unsigned carry out*
V	o**V**erflow	Instruksjonen ga en *overflow*

- Settes av ALU (kapittel 5)
- Holdes i *Current Program Status Register* (CPSR, ved siden av R0–R15).

ARM ALU


A
B
N
A31
B31
ALUControl0
Sum31
ALUControl1
Cout
1
0
+
Sum
ALUControl0
11
10
01
00
2
ALUControl
Result31
oVerflow
Carry
Negative
Zero
Result
N Z C V
4
ALUFlags

Hvordan *Condition Flags* (NZCV) settes

Metode 1: *Compare* : `CMP`

- Eksempel: `CMP R5, R6`
  - Utfører: R5-R6
  - Lagrer ikke resultatet men setter flaggverdier.
  - Dersom resultatet er
    - 0, Z=1
    - negativt, N=1
    - carry out, C=1
    - signed overflow, V=1

Hvordan *Condition Flags* (NZCV) settes

Metode 2: Legge til `S` etter instruksjonsnavn

- Eksempel: `ADDS R1, R2, R3`
  - Utfører R2 + R3
  - Sette flaggverdier:
    - Dersom resultatet er 0 (Z=1), negativ (N=1), osv.
- Lagrer i tillegg resultatet til R1

Betingede instruksjoner

- Instruksjoner med *betinget kjøring* basert på *condidtion flags*
- Gjenkjennes via navnetillegg (*mnemonic.)*
  - Eksempel:

```
CMP   R1, R2
SUBNE R3, R5, R8
```

  - NE: *condition mnemonic (**n**ot **e**qual)*
  - Her vil *SUB* kun kjøres dersom R1 ≠ R2 (dvs. Z = 0)

Betingede instruksjonsutvidelser (*condition mnemonic*)

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\bar{Z}$
0010	CS / HS	Carry set / Unsigned higher or same	C
0011	CC / LO	Carry clear / Unsigned lower	$\bar{C}$
0100	MI	Minus / Negative	N
0101	PL	Plus / Positive of zero	$\bar{N}$
0110	VS	Overflow / Overflow set	V
0111	VC	No overflow / Overflow clear	$\bar{V}$
1000	HI	Unsigned higher	$\bar{Z}C$
1001	LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
1110	AL (or none)	Always / unconditional	ignored

Betinget utførelse

- **Eksempel:**

```
– CMP   R5, R9          ; utfører R5-R9
                        ; og setter condition flags

– SUBEQ R1, R2, R3      ; utføres dersom R5==R9 (Z=1)
– ORRMI R4, R0, R9      ; utføres dersom R5-R9 er
                        ; negativ (N=1)
```

- Anta at R5 = 17, R9 = 23, hva skjer?
  - `CMP` gir: 17 – 23 = -6, setter flagg: *N*=1, *Z*=0, *C*=0, *V*=0
  - `SUBEQ` blir ikke kjørt siden de ikke var like (*Z*=0)
  - `ORRMI` blir kjørt siden resultatet var negativt (*N*=1)

Branches - instruksjons-hopp

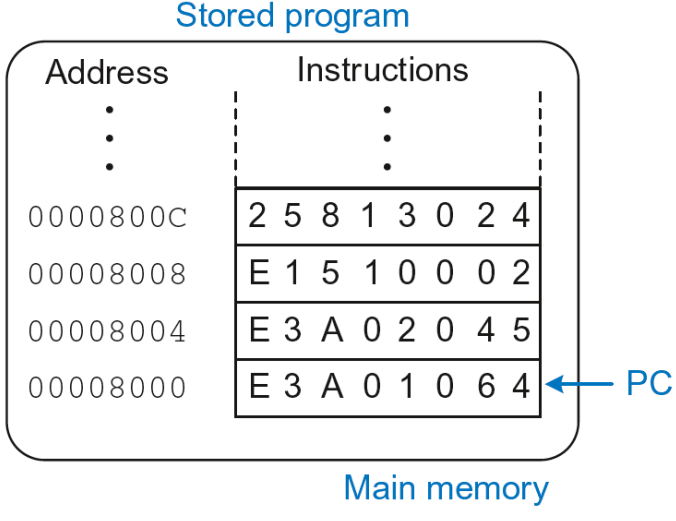
- Branches muliggjør hopp i kjøringen til forskjellige steder i programkoden
  - Ikke tvunget til å følge koden slavisk.
- Typer av branches:
  - **Branch (B)**
    - Hopper til en annen instruksjon (til en annen linje i programkoden)
  - **Branch and link (BL)**
    - Kommer vi tilbake til
  - Begge kan være betinget eller ubetinget

Assembly code

```
MOV   R1, #100
MOV   R2, #69
CMP   R1, R2
STRHS R3, [R1, #0x24]
```

Lagrede programkoden

Assembly code	Machine code
`MOV R1, #100`	`0xE3A01064`
`MOV R2, #69`	`0xE3A02045`
`CMP R1, R2`	`0xE1510002`
`STRHS R3, [R1, #0x24]`	`0x25813024`



Ubetinget Branch: Eksempel

ARM assemblerkode

```
   MOV R2, #17              ; R2 = 17
   B   TARGET               ; branch til target
   ORR R1, R1, #0x4         ; ikke utført
TARGET
   SUB R1, R1, #78          ; R1 = R1 + 78
```

- Branch sørger for at vi hopper over ORR instruksjonen til den *labelen* vi har spesifisert.
- Vi står fritt til å velge navn på *labels*.
  - Unntatt reserverte navn som `ADD`, `ORR`, osv.

Betinget Branch

ARM Assemblerkode

```
  MOV   R0, #4              ; R0 = 4
  MOV   R1, #8              ; R1 = 8
  CMP   R0, R1              ; setter flags med R0-R1
  BEQ    THERE              ; branch ikke utført(Z=0)
  ADD   R1, R1, #1          ; R1 = 8 + 1 = 9
THERE
  ADD R1, R1, #78           ; R1 = 9 + 78 = 87
```

Hvis ikke branch blir utført følger vi koden som normalt.

if Statement

C Kode

```
if (i == j)
  f = g + h;

f = f - i;
```

ARM Assemblerkode

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4       ; setter flag med R3-R4
 BNE L1           ; hvis i!=j, hopp over
 ADD R0, R1, R2   ; f = g + h
L1
 SUB R0, R0, R2   ; f = f - i
```

- Legg merke til at assemblerkoden tester for det motsatte (`i != j`) av C-koden (`i == j`)
- Kan vi gjør dette på en mer elegant måte?

If Statement: Alternativ kode med betinget *add*

C Kode

```
if (i == j)
  f = g + h;


f = f - i;
```

Assemblerkode fra forrige side

```
 CMP R3, R4
 BNE L1
 ADD R0, R1, R2
L1
 SUB R0, R0, R2
```

C Kode

```
if (i == j)
  f = g + h;
f = f - i;
```

Alternativ assemblerkode

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP   R3, R4       ; setter flag med R3-R4
 ADDEQ R0, R1, R2   ; if (i==j) f = g + h
 SUB   R0, R0, R2   ; f = f - i
```

if/else Statement

C Kode

```
if (i == j)
  f = g + h;

else
  f = f - i;
```

ARM Assemblerkode

```
;R0=f, R1=g, R2=h, R3=i, R4=j

 CMP R3, R4      ; setter flag med R3-R4
 BNE L1          ; hvis  i!=j, hopp til L1
 ADD R0, R1, R2  ; f = g + h
 B   L2          ; branch forbi else del
L1
 SUB R0, R0, R2  ; f = f - i
L2
```

Kan vi gjøre dette enklere?

if/else Statement: Alternativ kode

C Kode

```
if (i == j)
  f = g + h;
else
   f = f - i;
```

ARM Assemblerkode

```
;R0=f, R1=g, R2=h, R3=i, R4=j

CMP   R3, R4       ; setter flag with R3-R4
ADDEQ R0, R1, R2   ; if (i==j) f = g + h

SUBNE R0, R0, R2   ; else f = f - i
```

- Merk at det er kun `CMP` som setter *conditional flags.*
- Siden vi bruker betingete instruksjoner slipper vi å *branche*.

While loops

C Code

```
// skal finne x
// slik at 2^x = 128
int pow = 1;
int x   = 0;


while (pow != 128) {

  pow = pow * 2;
  x = x + 1;
}
```

ARM Assemblerkode

```
; R0 = pow, R1 = x
  MOV    R0, #1            ; pow = 1
  MOV    R1, #0            ; x = 0

WHILE
  CMP R0, #128             ; R0-128
  BEQ DONE                 ; hvis (pow==128)
                           ; hopp ut av løkken
  LSL R0, R0, #1           ; pow=pow*2
  ADD R1, R1, #1           ; x=x+1
  B   WHILE                ; repeter løkken
DONE
```

– Assemblerkoden tester for det motsatte tilfellet (`pow == 128`) av C koden (`pow != 128`).

for løkker

```
for (initialization; condition; loop operation)
   statement
```

- **initialization:** utføres før løkken starter
- **condition:** blir testet i begynnelsen av hver iterasjon
- **loop operation:** utføres på slutten av hver iterasjon
- **statement:** utføres hver gang *condition* er oppfylt

```
for (i = 0; i < 10; i = i + 1)
  print i
```

```
resultat = 0 1 2 3 4 5 6 7 8 9
```

for Loops

C Kode

```
// summer tallene fra 1-9
int sum = 0


for (i=1; i!=10; i=i+1)
   sum = sum + i;
```

ARM Assemblerkode

```
; R0 = i, R1 = sum
  MOV    R0, #1            ; i = 1
  MOV    R1, #0            ; sum = 0

FOR
  CMP R0, #10              ; R0-10
  BEQ DONE                 ; hvis (i==10)
                           ; hopp ut av løkke

  ADD R1, R1, R0           ; sum=sum + i
  ADD R0, R0, #1           ; i = i + 1
  B   FOR                  ; repeter løkke
DONE
```

For løkker ARM: *nedadgående* løkker mer effekltive

C kode

```
// adds numbers from 1-9
int sum = 0


for (i=9; i!=0; i=i-1)
   sum = sum + i;
```

ARM Assemblerkode

```
; R0 = i, R1 = sum
  MOV    R0, #9          ; i = 9
  MOV    R1, #0          ; sum = 0

FOR
  ADD  R1, R1, R0        ; sum = sum + i
  SUBS R0, R0, #1        ; i = i - 1
                         ; og setter flag
  BNE  FOR               ; hvis (i!=0)
                         ; repeater løkke
```

- Vi sparer 2 instruksjoner per iterasjon:
  - `SUBS R0, R0, #1` erstatter `CMP og` indeks justeringen
  - Har kun 1 branch – istedet for 2

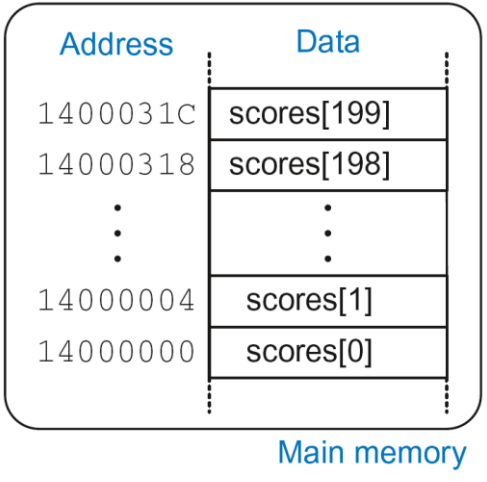
Arrays

- Når vi trenger å ha store mengder tilsvarende data tilgjengelig
  - Index: gir tilgang til hvert enkelt data element
  - Size: antall elementer (må allokere)

Array eksempel

Array med plass til 200 elementer

- Base addressen = 0x14000000
  - Adressen til det første elementet `scores[0]`).
- Resten av innholdet av Array'et hentes ut relativt til *base addressen.*



Hvordan aksessere Array'et (fra minne)

C Kode

```
int array[5];
array[0] = array[0] * 8;
array[1] = array[1] * 8;
```

ARM Assemblerkode

```
; R0 = array base address
  MOV R0, #0x60000000          ; R0 = 0x60000000

  LDR R1, [R0]                 ; R1 = array[0]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0]                 ; array[0] = R1

  LDR R1, [R0, #4]             ; R1 = array[1]
  LSL R1, R1, 3                ; R1 = R1 << 3 = R1*8
  STR R1, [R0, #4]             ; array[1] = R1
```

Funksjonskall

C Kode

```
void main()
{
  int y;
  y = sum(42, 7);
  ...
}


int sum(int a, int b)
{
  return (a + b);
}
```

- **Caller:** Den som kaller funksjonen
- **Callee:** Den som blir kalt på

Konvensjoner for funksjonskall

- **Caller:**
  - Gir **argumenter** til *callee*
  - Hopper/brancher til callee
- **Callee:**
  - **Utfører** funksjonen
  - **Returner resultatet** til *caller*
  - **Returnerer til** *point of call*
  - **Må passe på å ikke overskrive** registeret eller minne som *caller* trenger videre.

ARM funksjonskall konvensjoner

- **Kallefunksjon:**
  - **`BL`** (branch and link, setter verdien til `LR`)
- **Returnere** fra funkjsonen:
  - **`MOV PC, LR`** (når ferdig, hopp tilbake til caller via `LR`)
- **Argumenter:**
  - **`R0-R3`**
- **Returverdi:**
  - **`R0`**

ARM Funksjonskall eksempel

C Kode

```
int main() {
  simple();
  a = b + c;
}

void simple() {
  return;
}
```

void = ingen returverdi

ARM Assemblerkode

```
0x00000200 MAIN     BL   SIMPLE
0x00000204          ADD  R4, R5, R6
...

0x00401020 SIMPLE   MOV  PC, LR
```

brancher til `SIMPLE` og setter `LR` = `PC` + 4 = 0x00000204

Setter `PC` = `LR`
(neste instruksjon blir da BL 0x00000204)

Inngangsargumenter og returverdier: Eksempel

C Kode

```
int main()
{
  int y;
  ...
  y = diffofsums(2, 3, 4, 5);  // 4 argumenter
  ...
}

int diffofsums(int f, int g, int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result;                // returverdi
}
```

Inngangsargumenter og returverdier: Eksempel

ARM Assembly Code

```
; R4 = y
MAIN
  ...
  MOV R0, #2              ; argument 0 = 2
  MOV R1, #3              ; argument 1 = 3
  MOV R2, #4              ; argument 2 = 4
  MOV R3, #5              ; argument 3 = 5
  BL DIFFOFSUMS           ; kall funksjonen
  MOV R4, R0              ; y = returnert verdi
  ...

DIFFOFSUMS
  ADD R8, R0, R1          ; R8 = f + g
  ADD R9, R2, R3          ; R9 = h + i
  SUB R4, R8, R9          ; result = (f+g)-(h+i)
  MOV R0, R4              ; legg returverdi i R0
  MOV PC, LR              ; returner to caller
```

Problem!

- DIFFOFSUMS overskriver R4, R8 og R9
- Disse vil få en annen verdi etter DIFFOFSUMS kallet.
- Hva gjør vi hvis *caller* trenger verdiene?
- *Løsning*: Bruke *stack'en* for å midlertidig lagre unna verdiene.

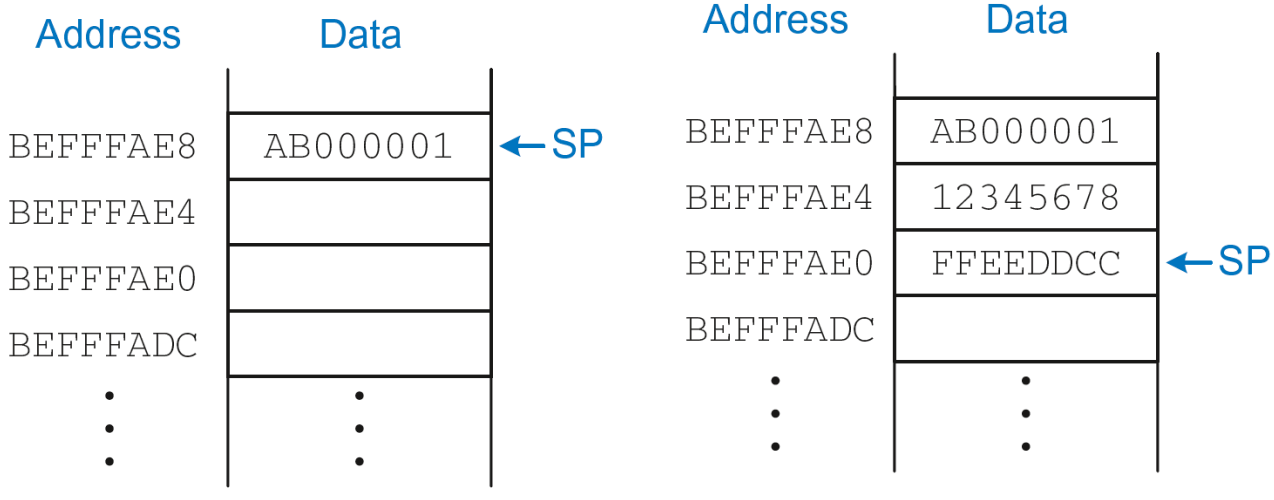
Stack

- Minne som brukes til å midlertidig lagre verdier.
- Som en “stack” av tallerkener:
  - *last-in-first-out* (LIFO)
- ***Expands*:** bruker **mer** minne når mer plass er nødvendig
- ***Contracts*:** bruker **mindre** minne når plassen ikke lengre er nødvendig



Stack

- Vokser nedover (fra høyere til lavere minne adresse)
- Stack pointer: SP peker på siste element som ble lagt til



Her vokser stack'en med 2 words

Hvordan funksjoner bruker Stack'en

- De funksjonene som vi kaller på, må ikke forstyrre for prosessen vi er i.
- Men som vi så
  - `diffofsums` overskrev 3 registerverdier: `R4`, `R8`, `R9`

ARM Assemblerkode

```
; R4 = result
DIFFOFSUMS
  ADD R8, R0, R1          ; R8 = f + g
  ADD R9, R2, R3          ; R9 = h + i
  SUB R4, R8, R9          ; result = (f + g) - (h + i)
  MOV R0, R4              ; legg returverdi i R0
  MOV PC, LR              ; returner til caller
```

Kan lagre registerverdiene på stack'en

ARM Assembly Code

```
; R2 = result
DIFFOFSUMS
  SUB SP, SP, #12          ; lage plass til 3 reg.
  STR R4, [SP, #8]         ; lagre R4 til stack
  STR R8, [SP, #4]         ; lagre R8 til stack
  STR R9, [SP]             ; lagre R9 til stack
  ADD R8, R0, R1           ; R8 = f + g
  ADD R9, R2, R3           ; R9 = h + i
  SUB R4, R8, R9           ; result = (f + g) - (h + i)
  MOV R0, R4               ; legg returverdi i R0
  LDR R9, [SP]             ; gjenopprette R9 from stack
  LDR R8, [SP, #4]         ; gjenopprette R8 from stack
  LDR R4, [SP, #8]         ; gjenopprette R4 from stack
  ADD SP, SP, #12          ; deallokere stack plassen
  MOV PC, LR               ; returnere til caller
```

push {R4,R8,R9}

pop {R4,R8,R9}

Stack’ens forløp iløpet av funksjonskallet



før kallet **mens funksjonen kjører** **etter kallet**

Merk at slik *stacking* koster oss mye ressurser

Reserverte og ikke-reserverte registre

Disse må funksjonen som blir kalt sørge for å ikke overskrive. Dvs. *stacke* ved behov.

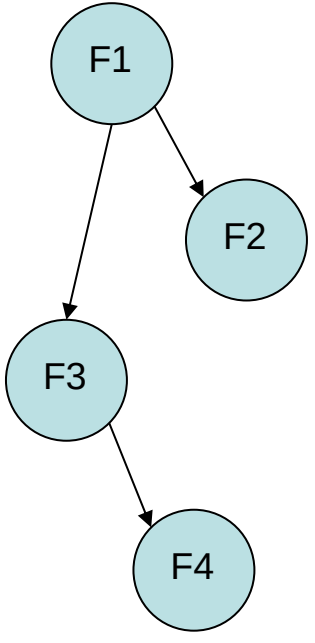
Preserved *Callee-Saved*	Nonpreserved *Caller-Saved*
R4-R11	**R12**
R14 (LR)	**R0-R3**
R13 (SP)	**CPSR**
stack above SP	**stack below SP**

Dersom man er avhengig av at disse verdiene ikke skal bli overskrevet av et funksjonskall, må man sørge for å bevare dem før kallet.

Hensikten med inndelingen er å **minimerer** behovet for å mellomlagre variabler på stack'en.

Leaf- og *Nonleaf*-funksjoner

- **Leaf-funksjon**: funksjon som ikke kaller en annen funksjon.
  - Her må vi kun sørge for å ikke overskrive *preserved registers*.
- **Nonleaf-funkjson**: funksjon som kaller andre funksjoner.
  - Må **også** sørge for å ikke overskrive *preserved registers*.
  - Hvis den er avhengig av verdiene til *nonpreserved registers,* er det nødvendig å lagre på dem stacken før funksjonskall, og gjenopprette verdiene etter funksjonskallet.



Nonleaf-funksjon: Eksempel

C Kode

```
int f1(int a, int b) {
  int i, x;
  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}
int f2(int p) {
  int r;
  r = p + 5;
  return r + p;
}
```

ARM Assemblerkode

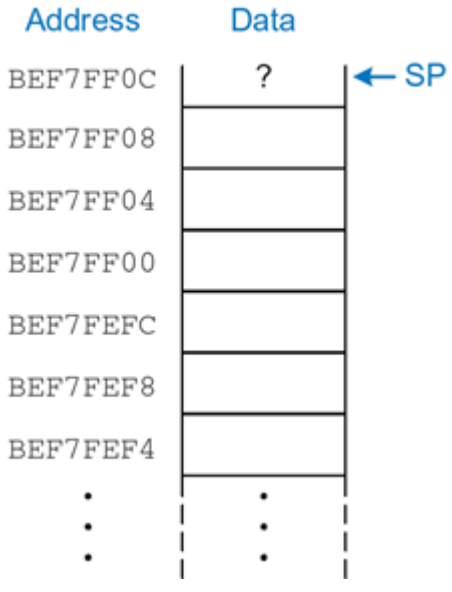
```
; R0=a, R1=b, R4=i, R5=x
F1
  PUSH {R4,  R5, LR}
  ADD   R5,  R0, R1
  SUB   R12, R0, R1
  MUL   R5,  R5, R12
  MOV   R4,  #0
FOR
  CMP   R4, R0
  BGE   RETURN
  PUSH {R0, R1}
  ADD   R0, R1, R4
  BL    F2
  ADD   R5, R5, R0
  POP  {R0, R1}
  ADD   R4, R4, #1
  B     FOR
RETURN
  MOV   R0, R5
  POP  {R4, R5, LR}
  MOV   PC, LR
```

```
; R0=p, R4=r
F2
  PUSH {R4}
  ADD   R4, R0, 5
  ADD   R0, R4, R0
  POP  {R4}
  MOV   PC, LR
```

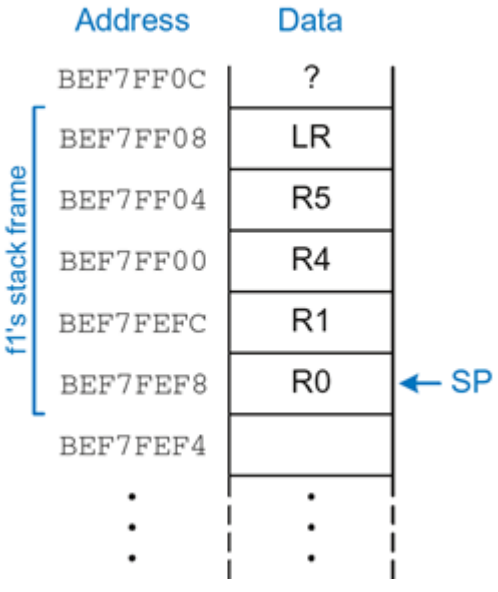
`F2` er *leaf*-funksjon og trenger kun å stacke *preserved* registers.

`F1` er *nonleaf*-funksjon og må både stacke *preserved* og *nonpreserved* registers.

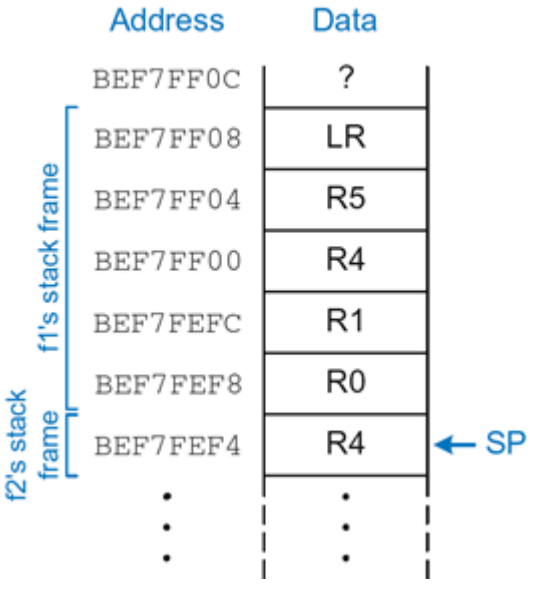
Stack’ens forløp



I begynnelsen av `f1`



Rett før kallet på `f2`



Etter kallet til `f2`

Oppsummering

Har sett på hva ARM arkitekturen tilbyr av funksjonalitet via *instruksjoner* og *registeret*.

- Dataprosessering : `ADD,SUB,MUL,LSL,..`
- Branches: `B TARGET`
- Betinget utførelse vha. flag og betingete instruksjoner: `ADDEQ,SUBNE,BEQ,..`
- Og hvordan vi kan bygge *høynivå konstruksjoner* som:
  - if/else, for og while løkker, arrays
  - Funksjonskall (stack'en er viktig).

Neste forelesning

- Maskinkode
  - Hvordan instruksjonene er kodet til 32-bits word.

Ukeoppgaver uke 6a:

6.1, 6.2, 6.3 (a, b), 6.16, 6.17, 6.18, 6.37, 6.44, 6.45