



Introduktion till assemblerprogrammering

ARM Assembler och arkitektur

Primärminne

Minnesoperationer

- **LOAD** – Läs data från minne
 - data läses från en given plats i minnet (adress) och hamnar i processorns register
- **STORE** – Skriv data till minne
 - data skrivs in i minnet på en given adress

Adress	Innehåll
0	01011001
1	11110000
2	01011000
3	11011100
4	11111111
5	00001111
6	11110000
7	10101010

Variabler och minne

Exempel, deklaration i C:

```
unsigned char x; //kan ha värde 0-255
```

1 byte (8 bitar) ger följande tal som kan representeras:

Binärt	Decimalt
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
.....	
1111 1101	253
1111 1110	254
1111 1111	255

Variabler och minne forts.

Exempel, deklaration i C:

```
signed char x; //värde mellan -128 och 127
```

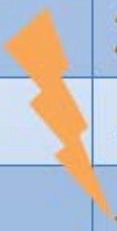
1 byte (8 bitar): Hur representeras talen?

Binärt	Decimalt
0000 0000	0
0000 0001	1
0000 0010	2
0000 0011	3
.....	
1111 1101	?
1111 1110	?
1111 1111	?

Variabler och minne, forts.

- Alternativ 1: Använd MSB som teckenbit
(MSB = Most Significant Bit)
- Då får vi 2 nollor, +0 och -0

Binärt	Decimalt
0000 0000	0
0000 0001	1
0000 0010	2
...
1000 0000	-0
....
1111 1110	-126
1111 1111	-127



Variabler och minne, forts

- Alternativ 2: 2-komplement

Binärt	Decimalt
0000 0000	0
0000 0001	1
0000 0010	2
....
1000 0000	-128
....
1111 1110	-2
1111 1111	-1

- Exempel $-1(\text{dvs. } 11111111) + 1 = 0$

Beteckningar på olika bitgrupper

- Byte – 8 bitar
- Ord (word) – 32 bitar (på används på PI, beror på arkitektur)
- Nibble – 4 bitar
- Halfword – 16 bitar

Ett dataord (32 bitar) tar upp fyra konsekutiva (direkt på varandra följande) adresser i minnet

Byte- eller word-adresserat minne

- Byteadresserat:
 - Måste hålla koll på vad som ska anses vara ord
 - Exempel läs adress 4, som är en del i ett ord.
- Wordadresserat:
 - Fragmentering om en byte ska lagras (tar upp 4 byte)
 - Läser word (4 byte) i taget

Adress	Byte	Data
0	0	1111 0000
1	1	1010 0101
2	2	1100 0011
3	3	0011 0011
4	4	1111 1111
5	5	0000 1111
6	6	1111 0000
7	7	1010 1010

Binärt → Hexadecimalt

Ofta skriver man binära bitmönster på hexadecimal form för att öka läsbarheten

Binärt	Hexadecimalt
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binärt	Hexadecimalt
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

CPU-tid

$\text{CPU-tid} = \text{Antal klockcykler} \times \text{klockperiod}$

- Prestanda kan förbättras genom att:
 - **Minska antal klockcykler**
 - **Öka klockfrekvensen**

Exekveringstid

En instruktion genomgår ett antal steg som vardera tar en klockcykel

Exempel: **ADD r1, r1, r3**

1. Hämta instruktion
2. Avkoda instruktion
3. Beräkning
4. Minnesläsning/skrivning
5. Skriv resultat till register

Summa 5 cykler (CPI, clocks per instruction 4 för **ADD**)

Exekveringstid

- Antal klockcykler för att exekvera en maskininstruktion
 - Antal klockcykler per instruktion (CPI)
- Tid för en klockcykel
 - Tid för en klockperiod (T)
 - Frekvens (f) är $f = 1/T$
- Antal maskininstruktioner
 - Instruction count (IC)
- Exekveringstid räknad i klockcykler: $CPI \times T \times IC$

Prestanda

- Prestanda kan förbättras genom
 - Öka klockfrekvensen (f), dvs minska T
 - Minska antal instruktioner (IC)
 - Minska antal klockcykler per instruktion (CPI)

Sådant som påverkar prestanda

- Algoritm
 - Bestämmer vilka och hur många *operationer* som ska utföras
- Programmeringsspråk, kompilator
 - Bestämmer hur många *maskininstruktioner* som ska utföras *per operation*
- Processor och minnessystem
 - Bestämmer hur snabbt instruktioner exekveras
- I/O och operativsystem
 - Bestämmer hur snabbt I/O-operationer kan utföras

Maskininstruktioner

Definition av instruktionens olika delar

- Vad ska göras? (operationskod)
- Vem är inblandad? (källoperander, *eng. source*)
- Vart ska resultatet? (destination)

Typer av instruktioner (ARMv6)

- Alla instruktioner är 32 bitar långa
(1 ord = 4 byte)
- Olika typer av instruktioner
 - Aritmetiska och logiska (ALU)
 - Minnesinstruktioner (load, store)
 - Programflödesinstruktioner (hopp)

ARM arkitektur

(ur programmerarens synvinkel)

- Arkitektur av RISC-typ (RISC = Reduced Instruction Set Computer)
- ARMv6 har 16 st 32-bits register

ARMv6 registerfil med konventionsnamn

Register	Namn	Kommentar
\$0	r0	Primärt argument och returvärde
\$1	r1	Sekundärt argument och högt word för 64-bit returvärde
\$2-\$3	r2-r3	Argument 3 och 4
\$4-\$11	r4-r11	Generella register, får ej förändras av subrutiner.
\$12	r12	Scratchregister för funktioner
\$13	sp	Stackpekare
\$14	lr	Länkregister, används vid subrutinanrop
\$15	pc	Programräknare

Aritmetiska operationer

- Aritmetiska operationer (+ och -)

–Två källoperander och en destination

```
ADD rd, rn, rm    /* rd = rn + rm */
```

Alla aritmetiska operationer följer samma mönster eftersom regelbundenhet ger enklare/effektivare implementation i hårdvaran

Aritmetiska operationer

- C-kod:
`f = (g + h) - (i + j);`

För variablerna `f`, `g`, `h`, `i`, `j` används register `r0`, `r1`, `r2`, `r3`, `r4`

- Kan kompileras till assembler-kod (ARM):
`ADD r5, r1, r2 /* temp r5 = g + h */`
`ADD r6, r3, r4 /* temp r6 = i + j */`
`SUB r0, r5, r6 /* r4 = r5 - r6 */`

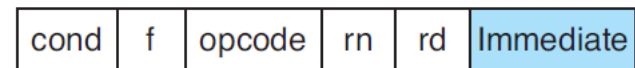
Adressering "ALU"-operationer

- Omedelbar adressering (*immediat*)

ADD r4, r4, #3 **/* r4 <- r4+3 */**

Ena operanden finns direkt i instruktionen

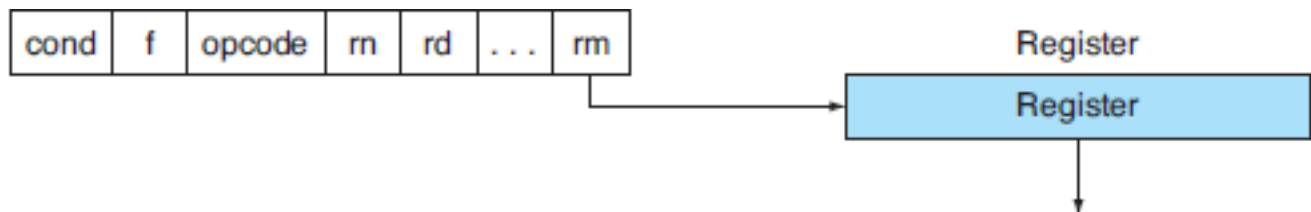
Operanden kan var högst 12 bitar



- Registeradressering

ADD r4, r4, r3 **/* r4 <- r4+r3 */**

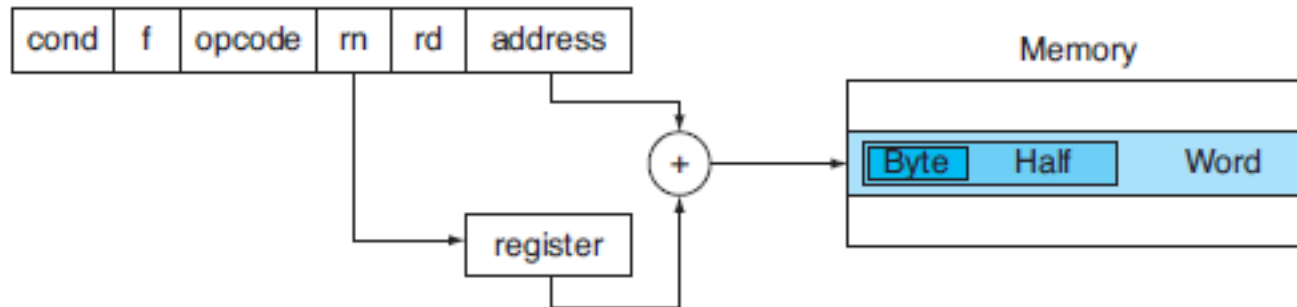
Alla operander finns direkt i register



Adressering minnesinstruktioner

- Adressering med basadress och förflyttning (*immediate offset*)

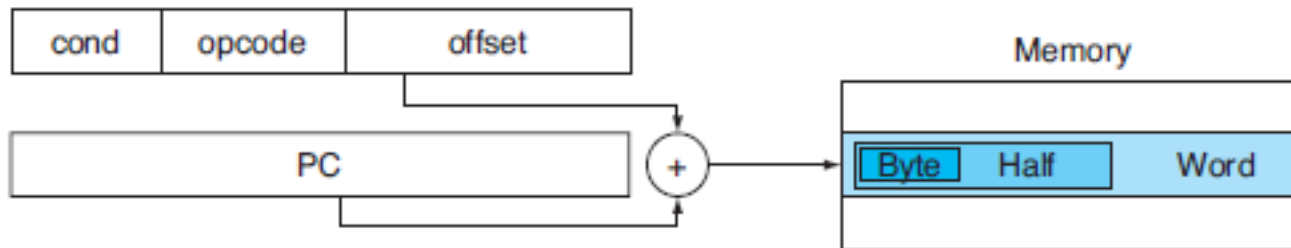
LDR r4, [r3,#4] **/* r4 <- [r3 + 4] */**



Ett register håller en basadress, och siffran i hakparentesen anger hur långt från denna basadress man avser att accessa minne (LDRB för byte och LDRH för half word)

Adressering hoppinstruktioner

- PC-relativ adressering
- **B label**



PC får nytt värde som är summa av nuvarande värde och offset till "hoppdestinationen" (label)

ARM har flera och mer komplexa adresseringsmoder (12st sammanlagt, se kap 2.10 på It's om du vill veta mer)

Statusregistret CPSR

Många instruktioner påverkar detta register.

Registret innehåller främst följande flaggor (bitar)

- Negative (N)
- Zero (Z)
- Carry (C)
- Overflow (V)

ARM-instruktion - addition

ADD r8,r9,r10 /* addera talen i register r9 och
r10 och lägg summan i r8 */

ARM-instruktion - subtraktion

SUB r8,r9,r10 /* subtrahera r9 - r10
och lägg resultatet i r8 */

Instruktionerna påverkar även flaggorna statusregistret CPSR

Hårdvarumultiplikation

- Multiplikation består av en serie additioner och skift
- Kan göras relativt snabbt i hårdvara, men är långsammare än addition

ARM-instruktion

MUL r8,r9

/* multiplicera innehållet i register r8 och r9 och
lägg resultatet i r8.
Klarar bara 32bits resultat */

Vad händer vid en **ADD**-instruktion?

- En instruktion i ARM är 32 bitar. Vad gör processorn med bitmönstret?

- Ex

ADD r3,r4,r5

/* addera innehållet i **r4** med innehållet i **r5** och lägg
resultatet **r3** */

Instruktioner och bitmönster i ARM

Alla instruktioner för ARM är 32 bitar = 4 byte
Instruktionen **ADD** kodas på följande sätt

Exempel:

ADD{*S*}<*c*> <*Rd*> , <*Rn*> , <*Rm*> { , <*shift*> }

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
cond				0	0	0	0	1	0	0	S	Rn			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rd				imm5				typ		0	Rm				

Generellt instruktionsformat ARMv6

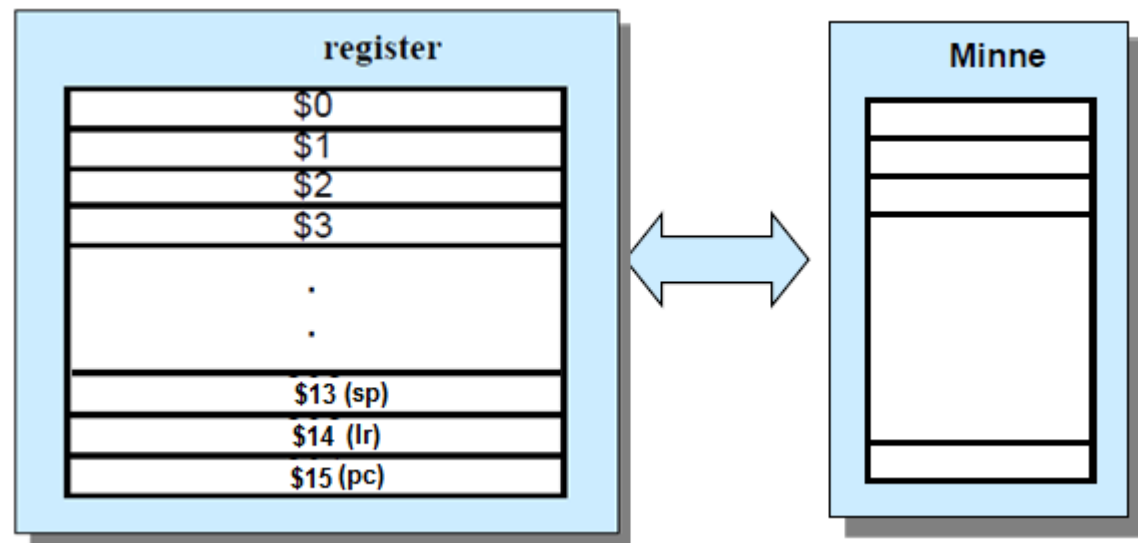
Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

Förklaring till de olika fälten:

- **Cond:** Villkor relaterade till villkorliga hopp
- **F:** Instruktionsformat, det finns flera varianter
- **I:** Immediate. Om 0 är andra operanden ett register annars en konstant
- **Opcode:** Talar om vilken instruktion det är frågan om
- **S:** Set condition code, relaterad till villkorliga hopp
- **Rn:** Första registeroperanden
- **Rd:** Destinationregister
- **Operand2:** Andra operanden

ARM

- 16 register varav 13 generella
- Programräknaren ($pc=r15$), innehåller adressen till nästa instruktion som ska hämtas
- Minnet innehåller både instruktioner och data



Minnesinstruktioner

- Minnesaccesser kan göras med 8, 16 eller 32 bitar
- Adressen räknas ut som innehållet i ett register plus en *offset*

Minnesinstruktioner (forts.)

Load (Byte/Halfword/Word)

ARM-instruktion

LDRB *r1*, [*r2*,#20]

LDRH *r1*, [*r2*,#20]

LDR *r1*, [*r2*,#20]

Förklaring

$r1 = M[r2 + 20]$ (8 bitar)

$r1 = M.h[r2 + 20]$ (16 bitar)

$r1 = M.w[r2 + 20]$ (32 bitar)

Store (Byte/Halfword/Word)

ARM-instruktion

STRB *r1*, [*r2*,#20]

STRH *r1*, [*r2*,#20]

STR *r1*, [*r2*,#20]

Förklaring

$M[r2 + 20] = r1$ (8 bitar)

$M.h[r2 + 20] = r1$ (16 bitar)

$M.w[r2 + 20] = r1$ (32 bitar)

Aritmetiska instruktioner

- Aritmetiska instruktioner har två *källoperander* och en *destinationsoperand*
- Operanderna ligger alltid i *register* utom om ena källoperanden är en konstant. Konstanten får vara högst 12 bitar lång.

Aritmetiska instruktioner (forts.)

Add

ARM-instruktion

ADD rd, rn, rm

ADD rd, rn, #K

Förklaring

Addera två tal i register

Addera ett tal i register med en konstant

Subtract

ARM-instruktion

SUB rd, rn, rm

SUB rd, rn, #K

Förklaring

Subtrahera två tal i register

Addera ett tal i register med en konstant

Förklaring till logiska operationer

- Sanningsvärden: 1 = sant, 0 = falskt
- **AND**: sant om båda operanderna är sanna
- **OR**: sant om någon av operanderna är sanna
- **NOT**: inverterar
- **XOR**: sant om operanderna är olika
- bitvis operation opererar på var och en av bitpositionerna samtidigt

Exempel på bitvis operationer

- Bitvis **AND** (logiskt OCH) mellan 01100011 och 01001010 ger resultatet 01000010
- Bitvis **OR** (logiskt ELLER) mellan 01100011 och 01001010 ger resultatet 01101011
- Bitvis **NOT** på 01100011 ger resultatet 10011100
- Bitvis **XOR** (exklusivt ELLER) mellan 01100011 och 01001010 ger resultatet 00101001