ARM Assembler Referens

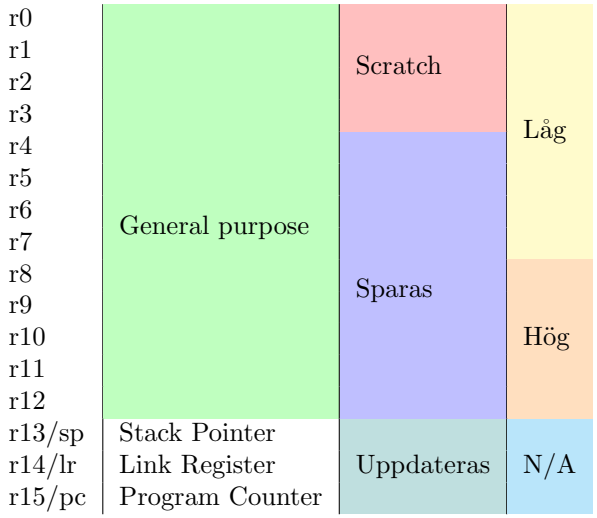
Mars 2023

Detta är tänkt som en referens till ARM assembler och riktar sig till de som programmerar för CPUlator ARMv7 System Simulator. Processorn har 16 register om 32 bitar.

1 Register

Av de 16 register ni har tillgång till är 12 general purpose, d.v.s de kan användas till vad som helst. Dessa register är `r0-r12`. `r13` är stackpekaren, och kan även refereras till som `sp`. `r14` är länkregistret (kan refereras till som `lr`), och innehåller återhoppsadressen efter att vissa instruktioner genomförts. `r15` är programräknaren (kan refereras till som `pc`) och innehåller adressen till nästa instruktion som ska exekveras.

General purpose registren är uppdelade i tre delar. `r0-r3` är s.k. *scratch-register*, vilket innebär att de kan användas fritt utan att man behöver spara undan dem först. Notera att p.g.a det bör dessa **inte** användas för beräkningar eller temporär lagring, eftersom anrop till andra funktioner kan (och troligen kommer) förstöra innehållet i registren. `r0-r7` räknas som de låga registren. Vissa instruktioner kräver att man använder de låga registren (dessa är markerade i tabellen på sidan 3). `r8-r12` är de höga registren.



2 Anropsprocedur och funktioner

Vid funktionsanrop används `BL` instruktionen. Den hoppar till en angiven label (d.v.s. funktionens namn) och sparar återhoppsadressen i `lr` registret. För att skicka argument till funktionen används scratch-registren, alltså `r0-r3` för att skicka de fyra första argumenten, där `r0` innehåller det första argumentet, `r1`

det andra, etc. Behövs det sedan fler pushas dessa till stacken med anropet `PUSH{...}` där argumenten pushas i ordning, d.v.s det första först etc. Ni bör inte göra det dock, utan det bör räcka med `r0-r3`. Funktioner returnerar värden i `r0`.

Alla funktioner ska inledas med en prolog, d.v.s. **alla** register (scratch-registren är undantagna) som används i funktionen ska pushas till stacken tillsammans med link registret. Alla funktioner ska avslutas med en epilog, d.v.s. alla register som pushades ska poppas ifrån stacken. Anropen till `PUSH` och `POP` ska lista registren i samma ordning, med undantag för `lr`. Där `lr` pushas ska `pc` poppas. Detta gör att nästa instruktion som exekveras är den som återhoppsadressen pekar på. Ska något värde returneras ifrån funktionen görs en `MOV` till `r0` *innan* `POP`.

Det är vanligt att argumenten flyttas ifrån scratch-registren till `r4-r12` efter prologen, för att inte riskera att förstöra funktionens argument om en annan funktion behöver kallas på.

Nedan följer ett enkelt exempel på hur en funktion och ett funktionsanrop kan se ut.

```
summarize:
    PUSH {r4,r5,lr} // Prolog
    MOV r4,r0       // Flytta argument till r4-r12
    MOV r5,r1
    ADD r4,r5       // r4 = r4 + r5
    MOV r0,r4       // Flytta r4 till r0 för att returnera
    POP {r4,r5,pc}  // Epilog. Notera att "pc" ersatt "lr"

main:
    MOV r0,#10      // Första argumentet i r0
    MOV r1,#5       // Andra argumentet i r1
    BL summarize    // Kalla på funktionen
                    // r0 är nu 15
```

3 Instruktioner

Tabellen nedan listar de mest användbara instruktionerna i ARM assembler. I tabellen är *#imm* ett *immediate* värde, alltså ett värde som används direkt. Alla immediate värden har # framför sig i ARM assembler. Heltalsdivision har fått ett eget avsnitt i dokumentet, eftersom inget inbyggt stöd finns.

Alla instruktioner i tabellen som är markerade med en röd rad kan enbart användas med de låga registren (`r0-r7`)

Instruktion	**Förklaring**	**Exempel**
	Flytta Data	
MOV rm,rn	rm ← rn	MOV r0,r1
MOV rm,#imm	rm ← *imm*	MOV r0,#10
LDR rm,=<label>	rm ← &label	LDR r0,=tmp
LDR rm,[rn]	rm = *rn	LDR r1,r0
LDR rm,=#imm	rm ← #imm	LDR r0,=#10
STR rm,[rn]	*rn = rm	STR r0,[r1]
	Stacken	
PUSH {...}	Pushar de listade registren till stacken	PUSH {r4,r5,lr}
POP {...}	Poppar de listade registren ifrån stacken	POP {r4,r5,pc}
	Aritmetik	
ADD rm,rn	rm = rm + rn	ADD r1,r0
ADD rm,#imm	rm = rm + *imm*	ADD r1,#10
SUB rm,rn	rm = rm - rn	SUB r1,r0
SUB rm,#imm	rm = rm - *imm*	SUB r1,#10
MUL rm,rn	rm = rm * rn	MUL r1,r0
	Branching	
CMP rm,rn	Jämför värdena i rm och rn	CMP r0,r1
CMP rm,#imm	Jämför värdet i rm med *imm*	CMP r0,#10
BEQ <label>	Hoppar till *label* om jämförda värden är lika	BEQ loop
BNE <label>	Hoppar till *label* om jämförda värden är olika	BNE loop
BLE <label>	Hoppar till *label* om det första jämförda värdet är mindre än eller lika med det andra	BLE loop
BGE <label>	Hoppar till *label* om det första jämförda värdet är större än eller lika med det andra	BGE loop
BLT <label>	Hoppar till *label* om det första jämförda värdet är mindre än det andra	BLT loop
BGT <label>	Hoppar till *label* om det första jämförda värdet är större än det andra	BGT loop
B <label>	Hoppar till *label*	B loop
BL <label>	Hoppar till *label*, sparar återhoppsadressen i lr	BL subroutine
	Logik	
MVN rm,rn	rm ← ¬ rn	MVN r0,r1
MVN rm,#imm	rm ← ¬ *imm*	MVN r0,#10
AND rm,rn	rm = rm ∧ rn	ADD r0,r1
AND rm,#imm	rm = rm ∧ *imm*	ADD r0,#10
ORR rm,rn	rm = rm ∨ rn	ORR r0,r1
ORR rm,#imm	rm = rm ∨ *imm*	ORR r0,#10
EOR rm,rn	rm = rm ⊕ rn	EOR r0,r1
EOR rm,#imm	rm = rm ⊕ *imm*	EOR r0,#10
LSL rm,rn	rm = rm << rn	LSL r0,r1
LSL rm,#imm	rm = rm << *imm*	LSL r0,#10
LSR rm,rn	rm = rm >> rn	LSR r0,r1
LSR rm,#imm	rm = rm >> *imm*	LSR r0,#10

3.1 Division

Heltalsdivision är som nämnt inte inbyggt i ARM assembler. Det är dock bra att ha en sådan funktion om man behöver använda t.ex. modulo-räkning. Därför får ni en funktion given som genomför heltalsdivision. Funktionen tar två argument: täljaren i `r0` och nämnaren i `r1`. Den returnerar sedan två värden: `r0/r1` i `r0` och `r0 % r1` i `r1`. Koden finns nedan.

```
idiv:
    MOV r2, r1
    MOV r1, r0
    MOV r0, #0
    B .Lloop_check
    .Lloop:
            ADD r0, r0, #1
            SUB r1, r1, r2
    .Lloop_check:
            CMP r1, r2
            BHS .Lloop
    BX lr
```

4 Assemblerfilens sektioner

Assemblerfiler är uppdelade i olika segment: globala variabler för sig, kod för sig. Dessa segment definieras med labels, t.ex. `.data` för datasegmentet som innehåller definierade variabler och `.text` som innehåller kod. Ej initierade variabler läggs i `.bss`.

Även variablerna definieras med labels, följt av en label som beskriver datatypen, följt av ett värde. De viktigaste datatyperna är `.word` för `int`, `.skip` för arrayer, `.ascii` för **ej** nullterminerade textsträngar, `.asciz` för nullterminerade textsträgnar och `.byte` för `char`.

Funktioner inleds och namnges via en label. Labels kan också användas internt i en funktion för att kontrollera programflödet. Det är vanligt att ha en punkt framför namnen på dessa för att markera att de är lokala (även om de *kan* användas ifrån andra funktioner i filen). Om man vill göra en variabel eller funktion åtkomlig utanför filen används `.global <label>`. Exempel på detta finns nedan:

```
.data
    // Här läggs alla initerade variabler
    .global i   // Gör i tillgänglig globalt
    i:      .word   0           // int i = 0;
    fmt:    .asciz  "i: %d\n"   // string fmt = "i: %d\n";
    arr:    .skip   10          // char arr[10];
```

```
.text
    // Här skrivs all kod
    .global main    // Gör main till en global funktion
    main:           // main()
        // Kod tillhörande main()
        .mainLoop:  // Lokal label för en loop i main()
            // Kod som ska köras i loopen
```

4.1 Tips och tricks

4.1.1 Jobba med arrayer

Att arbeta med arrayer i assembler är trivialt; hämta adressen till arrayen och addera en offset som motsvarar det index elementet i fråga skulle haft (i vårt fall blir detta `index*4` för en `int`, eftersom varje `int` är 4 bytes). Detta resulterar dock i minst fyra instruktioner; hämta adress, multiplicera, addera och hämta/spara värde:

```
LDR r4,=arr     // Hämta arraypekaren
MUL r5,#4       // Multiplicera indexet i r5 med 4
ADD r4,r5       // Addera offset till basadress
LDR r6,[r4]     // Hämta elementets värde
STR r6,[r6]     // Spara r6's värde i elementet
```

Detta går dock att göra mycket smidigare med en speciell version av `LDR`- och `STR`-instruktionerna:

```
LDR r4, =arr                  // Hämta arrayens adress
LDR r6, [r4, +r5, LSL #2]   // Hämta värdet i indexet i r5
STR r6, [r4, +r5, LSL #2]   // Spara värdet i r6 på index i r5
```

Det ser komplicerat ut, men det är ganska lätt. Strukturen på den andra operanden är `[adress,+/-index,shift]`. `index` shiftas enligt `shift`, och detta adderas (eller subtraheras, om `-` används istället för `+`) sedan till `adress`. Detta används sedan för som adress i operationen. I exemplet ovan sker `r4 + (r5<<2)`, d.v.s. `r5` skiftas två gåner till vänster och mulipliceras alltså med 4. Detta adderas sedan till `r4`.

4.2 Stora tal

Om man vill jobba med tal som är större än 16bit kan man inte använda `MOV`-instruktionen, eftersom ARMs instruktioner är 32bit långa, vilket i `MOV`s fall ska inkludera det 16bit+ långa värdet. Instruktionen `LDR` är dock byggd för att ladda in minnesadresser i ett angivet register, och minnesadresser är 32bit långa. Det går därför att använda `LDR` i de fall då mer än 16bit ska läggas i ett register:

```
LDR rm, =#0xFFFFFFFF
```