32 bittinen ARM

Tietoa

- RISC
- Load-Store arkkitehtuuri
- 13 ns. General-purpose rekisteriä r0-r12
- r13- r15: Stack Pointer, Link Register, Program Counter
- Application Program Status Register
- 32 bittiset rekisterit
- 16 bittiset ja 32 bittiset käskyt Thumb-2 tilassa, ainoa jota Cortex-M33 prosessori tukee

Rekisterit

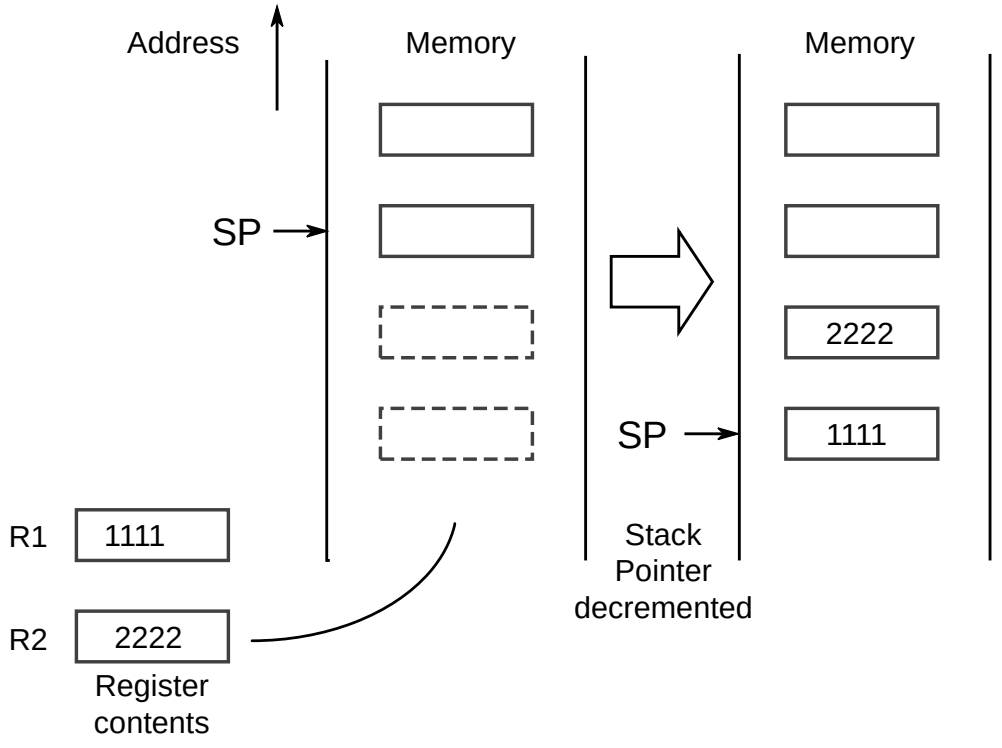
- r0-r12, myös r13 (SP), r14 (LR) ja r15 (PC)
- r0-r3 ovat argumentti/palautusarvo tai yleisiä “scratch” rekistereitä ns. caller-saved rekistereitä
- r4-r11 ovat muuttujarekistereitä ja ns. callee-saved rekistereitä
- r12 on nimeltään Inter-procedure call scratch register (IP), caller-saved rekisteri
- r13, Stack Pointer – osoittaa pinon “päällimmäiseen” arvoon
- r14, Link Register – sisältää paluuosoitteen branch-käskyistä
- r13 ja r14 -- Stack Pointer ja Link Register, voivat tarvittaessa toimia general-purpose rekistereinä, joskin vaikuttaa siltä, että SP:n käyttöä general-purpose rekisterinä ei suositella. On tärkeää muistaa puskea LR:n arvo pinoon, jos sitä käytetään general-purpose rekisterinä. Molemmat ovat callee-saved rekistereitä.
- r15, Program Counter osoittaa nykyiseen käskyyn +8 (ARM) tai +4 (Thumb). Vaikuttaa kuitenkin siltä, että gdb vähentää tästä automaattisesti -4, joten p/x $pc osoittaa nykyiseen käskyyn.

APSR

- ARMin käyttämä lippurekisteri
- Mahdollistaa ehtolauseet
- N = Negative, 1 kun tulos < 0, käytännössä sign bitin arvo
- Z = Zero, 1 kun tulos == 0
- C = Unsigned Carry (overflow), 1 kun vuodimme yli 32 bittisen arvon
- V = Signed Carry (overflow), 1 kun vuodimme yli etumerkillisen laskutoimituksen
- -S käskyt, esimerkiksi ADDS, MOVS päivittävät lippurekisterin
- Vertailukäsky CMP päivittää myös lippurekisterin

Pino

- Toimii kuin oikea pino, Last In First Out (LIFO)
- SP osoittaa pinon “päällimmäiseen” esineeseen
- Pino kasvaa kohti nollaa eli SP:n arvo pienenee
- SP:n **pitää olla aina neljällä jaollinen** tai kahdeksalla jaollinen jos käytetään tiettyjä käskyjä
- Teoriassa voi olla muitakin kuin alaspäin kasvava ns. full-descending stack, mutta ARM ei niitä virallisesti tue



Käskyt

PUSH & POP

- PUSH – puskee rekisterin arvon pinoon
- POP – poistaa arvon pinosta rekisteriin

ARMissa voi puskea ja poistaa monta rekisteriä samalla, esimerkiksi
Huomio, että pop kirjoitetaan tässä samassa järjestyksessä kuin push, mutta oikeasti se tapahtuu päinvastaisessa järjestyksessä

```
push    {r4,r6-r8,lr}       // puskee r4, r6, r7, r8 ja lr rekistereiden arvot pinoon

pop     {r4,r6-r8,pc}       // poistetaan pinosta arvot rekistereihin r4, r6, r7, r8 ja pc
```

LDR(H,B)

- **LDR** – load register. Lataa arvon rekisteriin muistista tai suoraan.
- **LDR** lataa yhden sanan (32 bittiä)
- **LDRH** lataa yhden puolisanan (16 bittiä)
- **LDRB** lataa yhden tavun (8 bittiä)
- LDR on oikeasti pseudokäsky, assembleri luultavasti muuntaa esimerkiksi ldr r0, #2 muotoon mov r0, #2
- 32 bittinen käsky ei voi sisältää 32 bittistä arvoa, joten mov-käsky voi kopioida vain 8 tai 16 bittisiä arvoja riippuen prosessorista.
- Näissä tilanteissa assembleri tallentaa arvon muistiin ja käsittelee sitä PC:n kautta. Tämän takia ARM assemblya purettaessa usein nähdään käskyjä, jotka ovat muotoa ldr r0, [pc, #??] jossa ?? on joku numero

```
ldr    r0, [pc, #4]      // Lataa arvon rekisteriin r0 paikasta pc+4
ldr    r0, [sp]          // Lataa arvon rekisteriin r0 4 tavua paikasta SP
ldr    r0, [r1, #16]     // Lataa arvon rekisteriin r0 paikasta r1+16
ldr    r0, #0x1201       // Lataa arvon 0x1201 rekisteriin r0, tätä et näkisi purettaessa vaan tämän kirjoittaisit itse
```

STR(H,B)

- **STR** – store register. Tallentaa arvon rekisteristä muistiin.
- **STR** tallentaa yhden sanan (32 bittiä)
- **STRH** tallentaa yhden puolisanan (16 bittiä)
- **STRB** tallentaa yhden tavun (8 bittiä)

```
mov     r2, #0x21       // asetetaan r2 arvoksi 0x21
ldr     r1, #0x200      // asetetaan r1 arvoksi 0x200, huomioi ldr
mov     r0, #29         // asetetaan r0 arvoksi 29

str     r0, [r2]        // Tallentaa r0 arvon (29) paikkaan 0x21
str     r0, [r2, #4]    // Tallentaa r0 arvon (29) paikkaan 0x21+4
str     r0, [sp]        // Tallentaa r0 arvon (29) paikkaan sp
```

MOV(S)

- **MOV** *rd, Operand2*
- Kopioi arvon *Operand2* rekisteriin *rd*
- **MOVS** tekee saman, mutta asettaa N (negative) ja Z (zero) flagit arvon mukaan
- Operand2 voi olla 8 bittinen tai 16 bittinen arvo, riippuen prosessorista.

```
mov     r0, #28   // kopioi arvon 28 rekisteriin r0
```

ADD(S) ja SUB(S)

- **ADD(S)** *(rd,) rn, Operand2*
- **SUB(S)** *(rd,) rn, Operand2*
- S-versiot päivittävät APSR rekisterin

```
add r0, r1, #2      // r0 = r1 + 2
add r0, #4          // r0 = r0 + 4
add r0, r1, r2      // r0 = r1 + r2

sub r0, r1, #5      // r0 = r1 - 5
sub r0, #7          // r0 = r0 - 7
sub r0, r3, r4      // r0 = r3 - r4
```

branch

- **b** *label* -- branch
- **bl** *label* – branch & link
- **bx** *Rm* -- branch & exchange instruction set
- **b** hyppää kohtaa label
- **bl** hyppää kohtaan *label* ja tallentaa paluuosoitteen lr rekisteriin
- **bx** hyppää rekisterissä *Rm* olevaan osoitteeseen ja vaihtaa ARM 32/Thumb käskyjen välillä. Tästä ei tarvitse huolehtia.

```
start:
        mov     r0, #1      // r0 = 1
foo:
        add     r0, #1      // r0 = r0 + 1 eli ++r0
        b       foo         // hyppää paikkaan foo
```

Sinne, tänne ja takaisin

- **bl** tallentaa paluuosoitteen lr rekisteriin
- Mahdollisia paluutapoja ovat:

```
bx      lr      // hypätään lr rekisterin sisältämään arvoon
mov     pc, lr  // kopioidaan lr rekisterin arvo pc rekisteriin
```

Nämä kaksi tapaa toimivat vain jos kyseessä on ns. leaf proseduuri eli emme käytä yhtäkään branch käskyä sen sisällä. Jos haluamme palata useammasta branchista, pitää lr säilöä pinoon. Tämä myös mahdollistaa palaamisen **pop** käskyllä

```
push    {r4-r6, lr}  // pusketaan r4-r6 ja lr pinoon
/* jotain... */
bl      //jonnekin....
/* sieltä takaisin*/
pop     {r4-r6, pc} // huom! tässä pc
```

Ehtolauseet

- Kuten aikaisemmin mainittiin, CMP ja -S päättyiset käskyt päivittävät lippuja APSR rekisterissä. Nämä liput mahdollistavat muitakin juttuja, kuten 64-bittisen aritmetiikan, mutta keskitymme tässä vain ehtolauseisiin, jotka toteutetaan **cmp** + **branch** -käskyillä

Muutamia branch käskyjä
beq – branch if equal
bne – branch if not equal
bhs -- branch if unsigned higher or same (sama kuin bcs)
bls -- branch if unsigned less or same
bcc – branch if carry clear
bcs -- branch if carry set
bhi – branch if unsigned higher
blo – branch if unsigned lower (sama kuin bcc)
bge – branch if signed greater or equal
ble – branch if signed less or equal
bgt -- branch if signed greater
blt -- branch if signed lesser

Käytännössä siis

```
cmp     r0, r1
beq     // r0 == r1
bne     // r0 != r1
bhs     // r0 >= r1
bls     // r0 <= r1
bcc     // r0 < r1
bcs     // r0 >= r1
bhi     // r0 > r1
blo     // r0 < r1
bge     // r0 >= r1 (signed)
ble     // r0 <= r1 (signed)
bgt     // r0 > r1   (signed)
blt     // r0 < r1   (signed)
```

- Ohjelmassamme

```
if(var > 20){
        printk("Hello world\n");
}
/* foo() kutsu tässä */
return 0;
```

- Voitaisiin kirjata näin

```
push    lr
ldr     r3, [sp, #4]    // Lataa muuttujan arvo rekisteriin r3
cmp     r3, #20         // r3 > 20?
bhi     .L1             // hyppää .L1

// return 0;
.L0:
mov     r0, #0          // palautusarvo 0
pop     pc              // poistutaan funktiosta

.L1:
ldr     r0, [pc, #4]    // ladataan “Hello world” merkkijonon osoite r0
bl      <printk>        // kutsutaan printk()
b       .L0             // Hypätään .L0
```