Tekstoppgaver del 2:

1. Kalleren har ansvaret for å ta vare på r0-r3, funksjonen har ansvar for Â plassere resultatet i r0. funksjonen må mellomlagre verdier fra r0-r3 i registeret dersom den trenger de. r4-r11 skal være umodifiserte, så disse må evt. også lagres i register mens funksjonen "låner" de.
2. Input-argumentet ble lagret i registeret, mens fib brukte det. Derfor er dette det samme når funksjonen har lastet det tilbake fra register. Men får load kalles, er input argumentet 0, fordi denne variabelen ble brukt som teller.
   3. Siden main ikke har noen ordentlig return-point å gå til, gjorde jeg et kall på SWI som avslutter programmet.

Tekstoppgaver del 3:

1. Logikken ser ut til å være tilsvarende, men det er gjort på en mer uleselig måte. Siden det er gjort om fra C-kode, inneholder den masse includes eller hva man skal kalle det. Men f.eks måten loopen er konstruert ser veldig lik ut. Det er brukt noe save, pop og push som jeg ikke har benyttet meg av.
2. Koden som produseres blir mer leselig, enn den tidligere.
3.
    - skrive kode i ved hjelp av en kompilator er som regel mye raskere enn ved Assembler.
    - Mer overførbart til andre OS, vil være færre endringer som skal til sammenlignet med assembler kode som er spesifikt for prosessorakritektur.
    - Enklere å feilsøke.
    - Assembler gir bedre kontroll over maskinvaren, som kan hjelpe med ytelse og utnyttelse av maskinvarefunksjoner
    - Assembler kode er mer tidkrevende
    - Assembler kode gir bedre muligheter for optimalisering av ytelse.

Tekstoppgaver del 4:

1. 2.0 -> 1.0*2^1 (i binaer)

   Exp: 1 + 127 = 128 -> 10000000 (i binaer)

   IEEE 754: 0 10000000 00000000000000000000000

2. 3.0 -> 1.1*2^1

   Exp: 1 + 127 = 128 -> 10000000

   IEEE 754: 0 10000000 10000000000000000000000

3. 0.50390625 -> 0.10000001 -> 1.000001*2^0

   Exp: 0 + 127 = 127 -> 01111111

   IEEE 754: 0 01111111 00000010000000000000000

4. 2.0 + 0.50390625

   Steg 1:

   10000000 00000000000000000000000

   01111111 00000010000000000000000

   Steg 2:

   10000000 1.00000000000000000000000

   01111111 1.00000010000000000000000

   Steg 3: Shift -> 1.0*2^0

   10000000 1.00000000000000000000000

   01111111 1.00000010000000000000000

   Steg 4:

   10000000 1.00000000000000000000000

   10000000 0.10000010000000000000000

   Steg 5:

   10000000 1.00000000000000000000000

   10000000 0.10000010000000000000000

   1.10000010000000000000000

   Steg 6: Trenger ikke nomarlisere

   Steg 7: Trenger ikke runde av

   Steg 8:

   0 10000000 10000010000000000000000