1 **Digital representasjon**

Gjør om desimaltallet $(25)_{10}$ til et 8 bits binærtall.

Velg ett alternativ

- Ingen av alternativene er korrekte.
- 00011001 ✔
- 00111011
- 00101010
- 00010111

Maks poeng: 2

2 **Digital representasjon**

Gjør om desimaltallet $(-35)_{10}$ til et 8 bits binærtall på 2'ers kompliment form.

Velg ett alternativ

- 11011101 ✔
- Ingen av alternativene er korrekte.
- 11010011
- 01011110
- 01110111

Maks poeng: 3

3 Forenkle følgende uttrykk maksimalt.

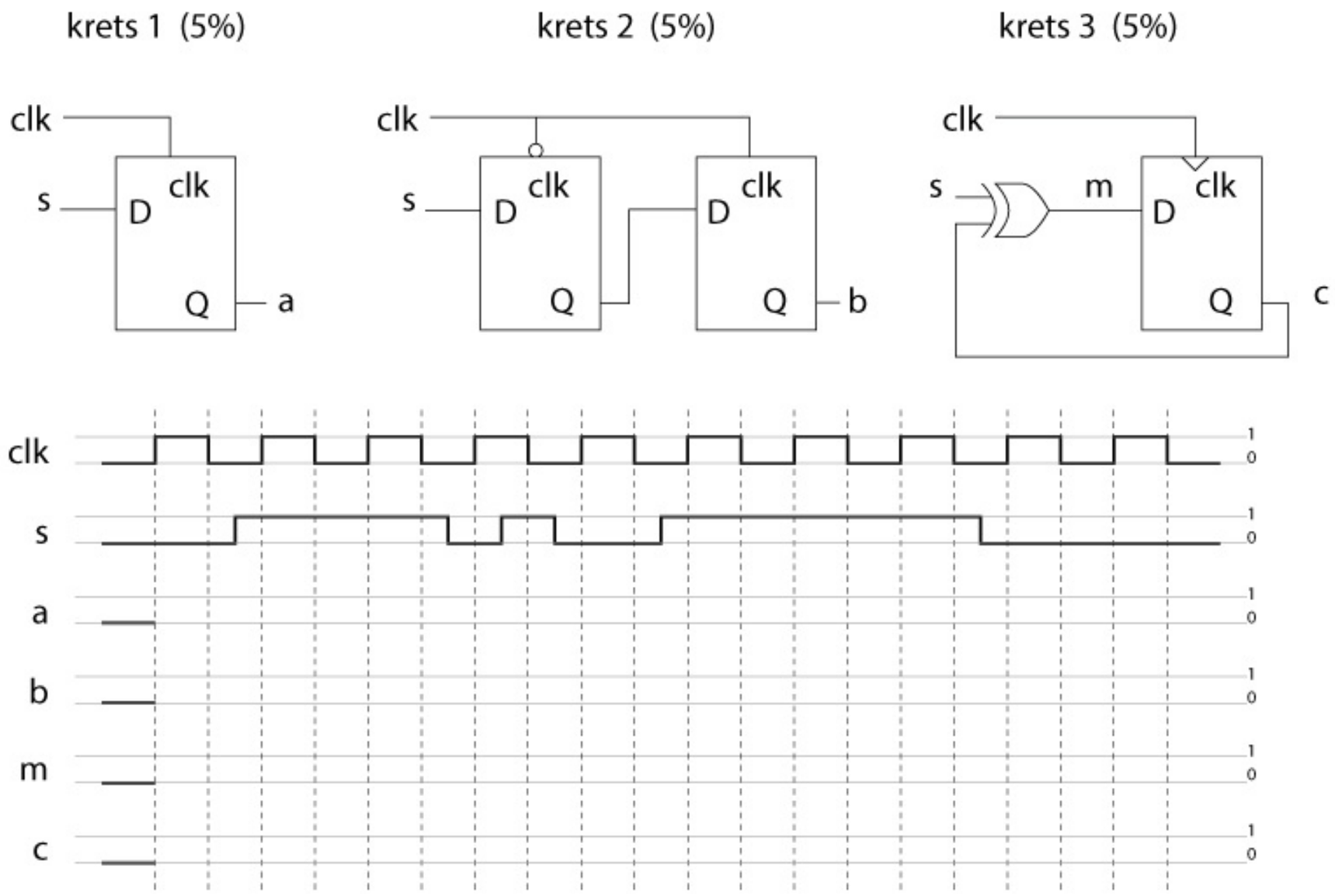
F = XYZ + XY' + XYZ'

Velg ett alternativ

- XY'+Z
- Y'+Z
- X ✔
- XY
- XY'

Maks poeng: 5

4 Hver av de tre kretsene under tar inn klokkesignalet clk og inngangsignalet s. Anta at a, b, c og m har startverdien 0. Tegn inn det manglende tidsforløpet for signalene a, b, m og c. Du skal ikke ta hensyn til portforsinkelse. Hint: Merk forskjellen på latcher og fliflop i illustrasjonen. **Besvarelsen skal tegnes inn på et eget ark.**



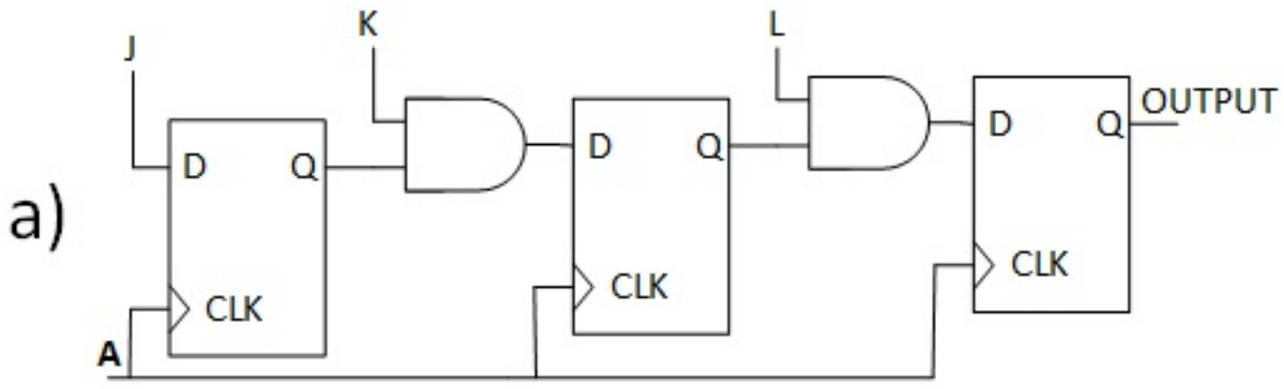
I denne oppgaven kan du svare med digital håndtegning. Bruk eget skisseark (utdelt). Se instruksjon for utfylling av skisseark på pult.

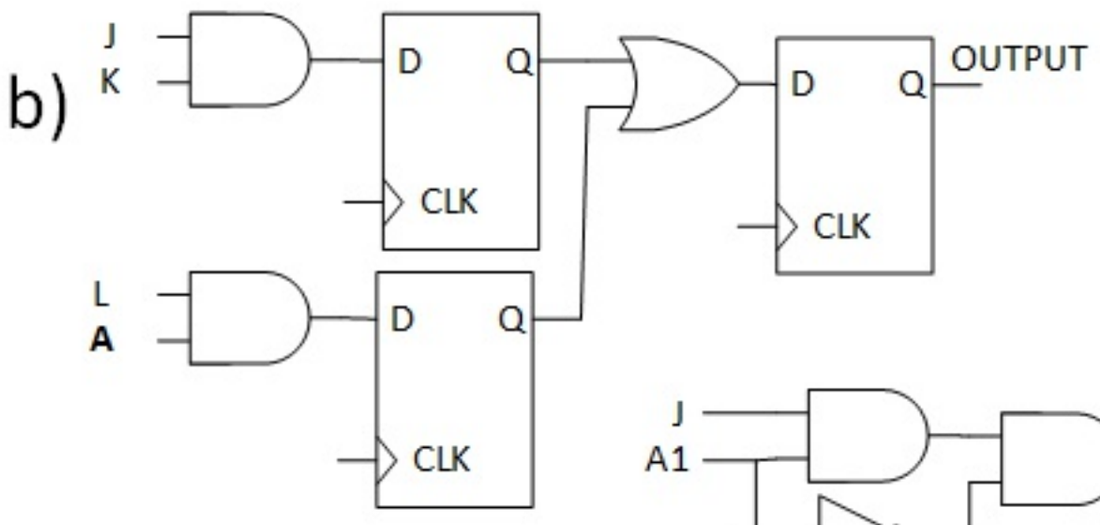
Maks poeng: 15

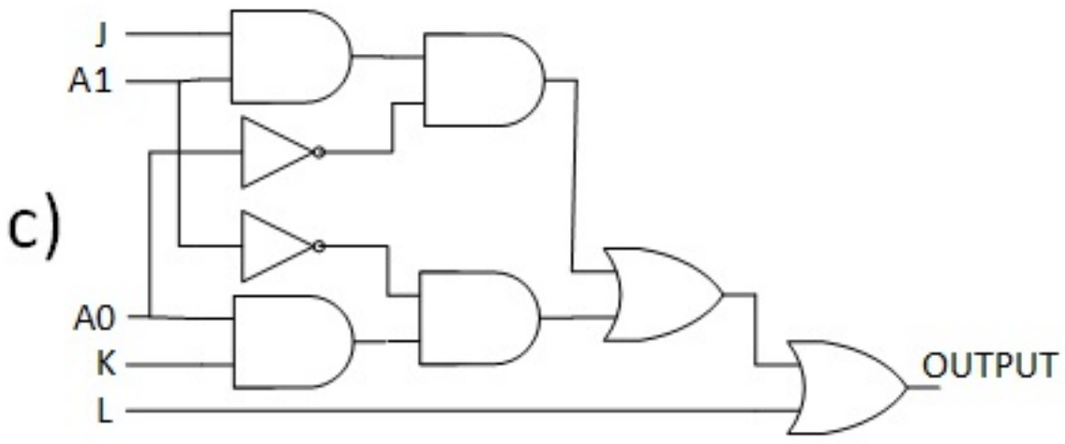
5

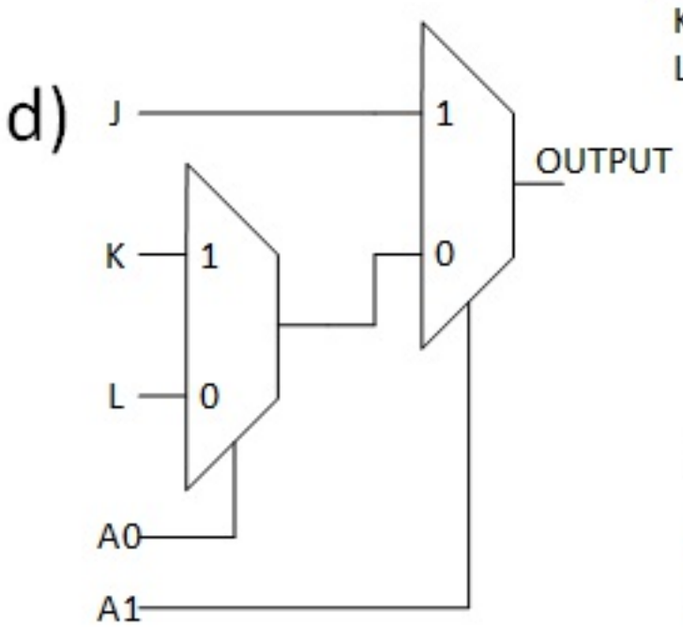
```
library IEEE;
   use IEEE.STD_LOGIC_1164.all;

entity my_thing is
   port( J,K,L  : IN STD_LOGIC;
         A      : IN STD_LOGIC_VECTOR(1 downto 0);
         OUTPUT : OUT STD_LOGIC);
end entity my_thing;

architecture ex of my_thing is
begin
   process(all)
   begin
      if    A(1) = '1' then OUTPUT <= J;
      elsif A(0) = '1' then OUTPUT <= K;
      else                  OUTPUT <= L;
      end if;
   end process;
end architecture ex;
```

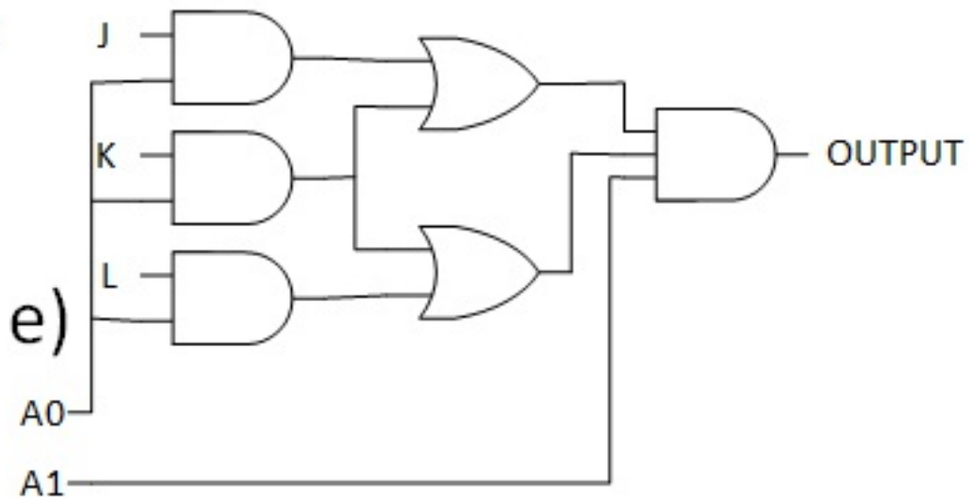
Eksamen IN2060 H2018











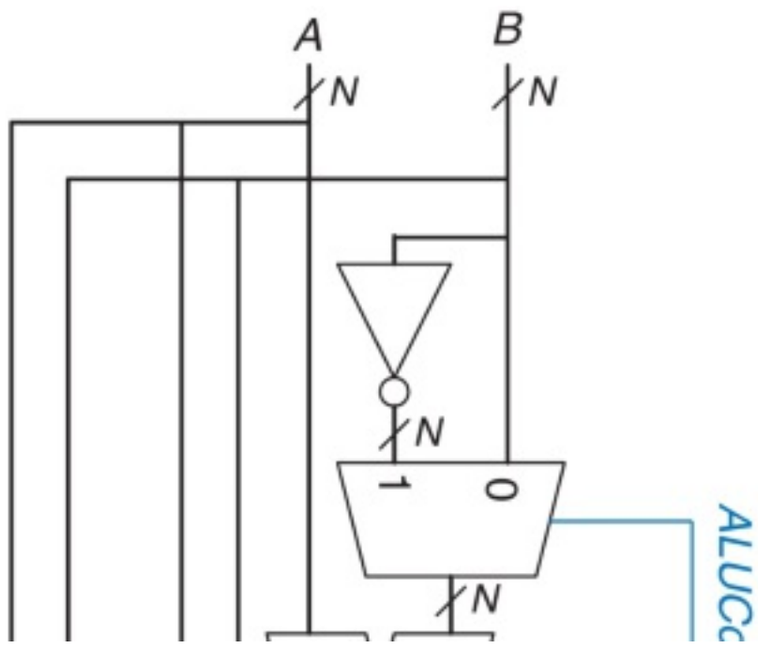
Hvilke(t) av diagrammene passer til koden?

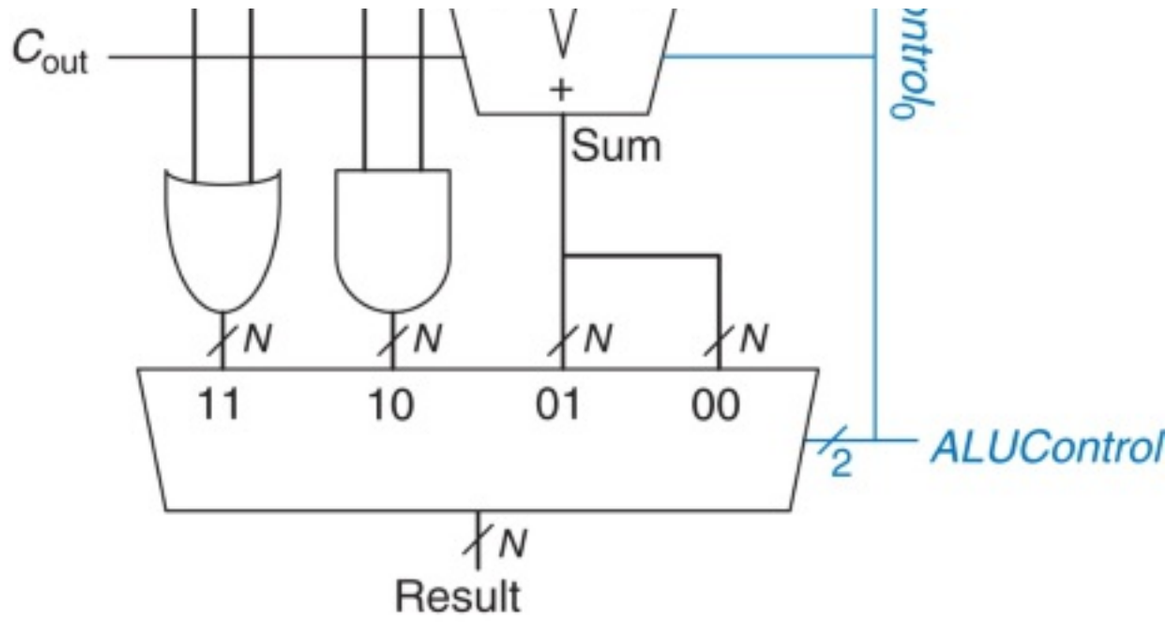
Velg ett eller flere alternativer

- a)
- b)
- c)
- d) ✔
- e)

Maks poeng: 5

6





Figuren viser en N-bit ALU.

Sett kryss ved påstander som er sanne

- Kontrollsignal 10 gir A OR B
- Kontrollsignal 01 må brukes for å legge sammen 2-kompliment tall (signed add)
- C_{out} vil alltid være '1' når vi legger sammen to tokompliments tall
- Med to slike ALU-er kan vi multiplisere vilkårlige N-bits tall på to klokkesykler.
- Adderen er av typen Ripple-Carry
- Bruker vi ALU-en til logiske operasjoner vil C_{out} alltid være 0
- Vi kan ikke utføre A XOR B med denne ALU-en direkte ✔
- ALU-en har to multipleksere ✔
- ALU-en har 3 multipleksere
- ALU-en er hjertet i mikrokontrolleren
- Kontrollsignal 10 gir A AND B ✔
- Kontrollsignal 00 brukes til addisjon ✔

Maks poeng: 10

7

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder2_4 is
  port(a: in STD_LOGIC_VECTOR(1 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end entity;

architecture synth of decoder2_4 is
begin
   process(all) begin
      case a is
         when "00" => y <= "0001";
         when "01" => y <= "0010";
         when "10" => y <= "0100";
         when "11" => y <= "1000";
         when others => y <= "0000";
      end case;
   end process;
end architecture;
```

Over ser du fullstendig VHDL-kode til en 2 til 4 dekoder.
Lag fullstendig kode til en 3 til 8 dekoder.
Kopier og modifisér gjerne elementer fra koden over.

Skriv ditt svar her...

Maks poeng: 5

8

```
library IEEE;
  use IEEE.STD_LOGIC_1164.all;

entity fsm is
  port( CLK, RESET : IN  STD_LOGIC;
        A, B       : IN  STD_LOGIC;
        P, Q, R    : OUT STD_LOGIC);
end entity fsm;

architecture ex of fsm is
begin
  type statetype is (S0, S1, S2);
  signal state, nextstate : statetype;

process(CLK, RESET)
  begin
    if RESET then state <= S0;
    elsif rising_edge(CLK) then
      state <= nextstate;
  end if;
end process;
```

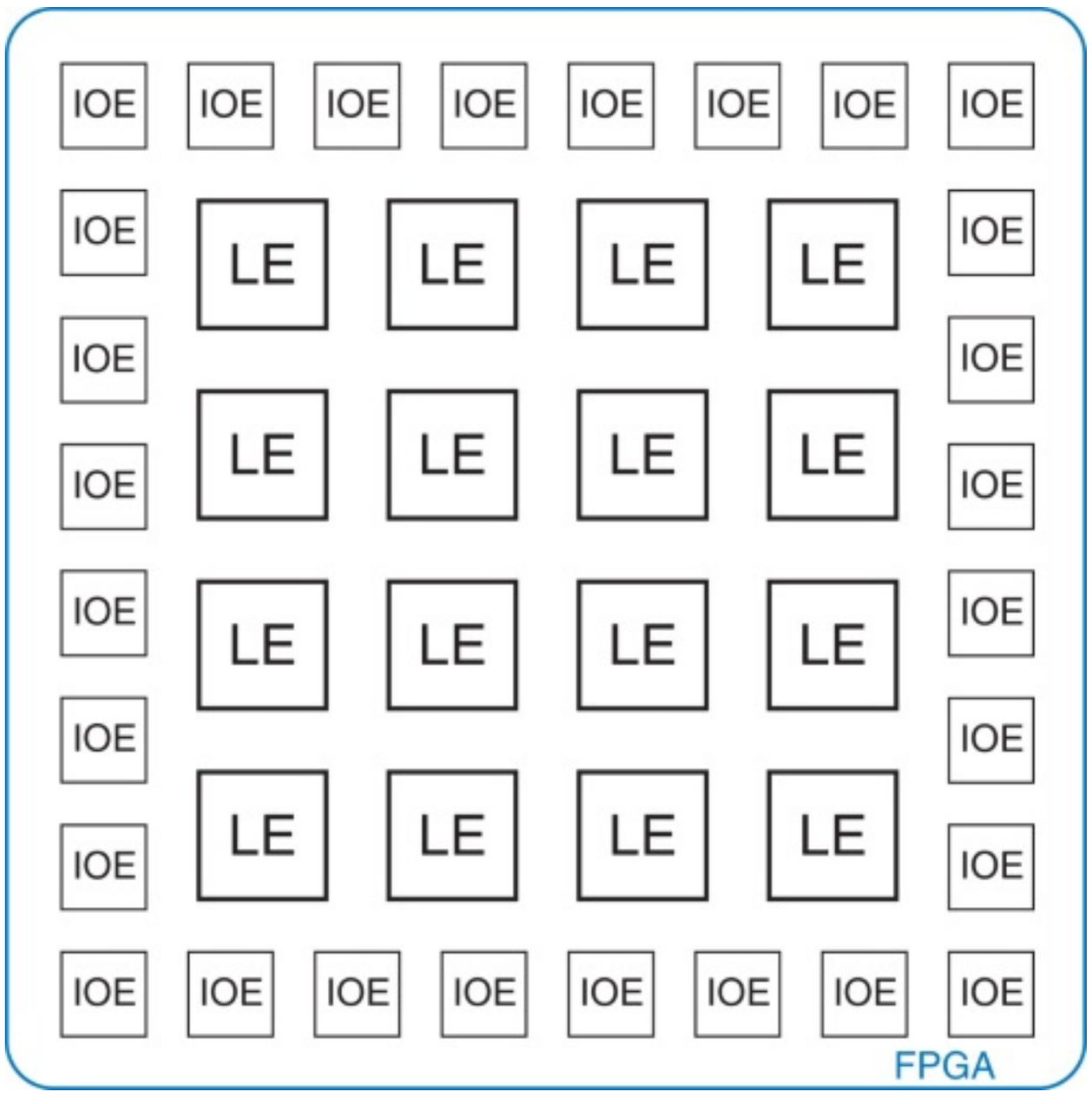
```
process(all)
  begin
    case state is
      when S0 =>
        if ((A = '1') or (B='1')) then nextstate <= S0;
        else                           nextstate <= S1;
        end if;
      when S1 =>
        if    (A= '1')     then nextstate <= S0;
        elsif (B= '1')     then nextstate <= S1;
        else                    nextstate <= S2;
        end if;
      when S2 =>
        if (A='1')      then nextstate <= S1;
        elsif (B='1')   then nextstate <= S2;
        else                 nextstate <= S0;
      end if;
    when others => nextstate <= S0;
  end process;

  -- concurrent statements
  P = '1' when state = S0 else '0';
  Q = '1' when state = S1 else '0';
  R = '1' when state = S2 else 'Z';

end architecture ex;
```

Tegn et tilstandsdiagram for tilstandsmaskinen beskrevet med VHDL-koden over på et eget ark.

I denne oppgaven kan du svare med digital håndtegning. Bruk eget skisseark (utdelt). Se instruksjon for utfylling av skisseark på pult.

Maks poeng: 10

9



Figuren over viser en generell arkitektur for en FPGA bestående av logiske elementer (LE) og input-output elementer (IOE).

Ta stilling til følgende påstander og kryss av for alle som er riktige.

- [ ] Når vi programmerer en FPGA med VHDL skjer instruksjonene sekvensielt.
- [ ] IOE blokker kan ikke ha tristate-buffere
- [x] FPGA kan brukes til både kombinatorisk og sekvensiell logikk
- [ ] En FPGA kan ikke inneholde RAM
- [ ] Avstanden mellom logiske elementer som kobles sammen er ubetydelig for klokkehastigheten
- [x] Tristate-buffere brukes gjerne til busser med flere drivere
- [x] Det er lurt å unngå bruk av typen INOUT for interne signaler, fordi det kan føre til unødig bruk av IOE blokker
- [x] Når vi programmerer en FPGA bestemmer vi hvilke deler av de logiske elementene som brukes og hvordan disse kobles sammen
- [x] Logiske elementer kan inneholde oppslagstabeller (LUT-er)

Maks poeng: 5

10

```
.text
.global main
main:
        MOV r5, #10
        MOV r7, lr
        BL add_one
        BX lr

add_one:
        ADD r0, r5, #1
        BX lr
```

Velg ett eller flere alternativer

- [x] 'main' bruker feil register til funksjonargument
- [ ] 'main' lagrer link-registeret riktig før funksjonskall
- [ ] 'add_one' returnerer verdi i feil register
- [x] Programmet vil aldri avslutte

Maks poeng: 5

11 Vi ønsker å oversette følgende program til ARM assembler. Du kan anta at '*g*' ligger i '*R0*' og '*h*' ligger i '*R1*'.

```
if(g < h) {
    h = h + 1;
} else {
    h = h * 2;
}
```

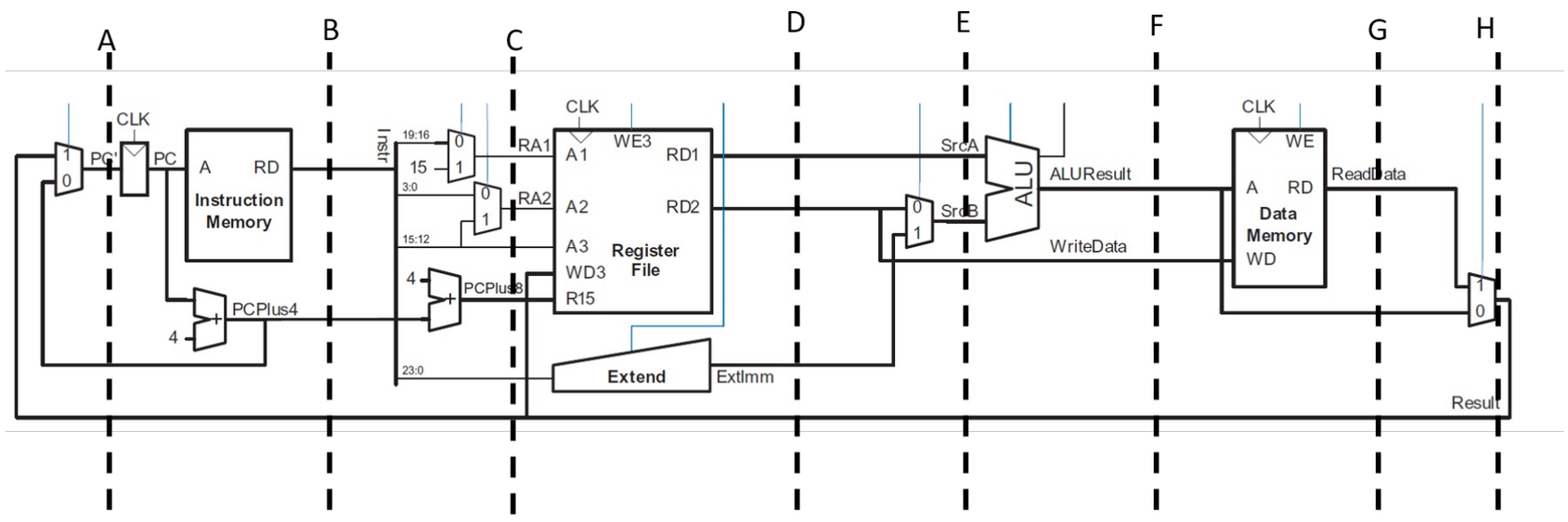
Besvar følgende spørsmål om antall instuksjoner i den oversatte assembler koden:

a. Hvis man bare kan benytte betingetkjøring (*conditional execution*) på hopp (*branch*), trenger man minimum [ ] (5) instruksjoner.

b. Hvis man kan benytte betingetkjøring (*conditional execution*) på alle instuksjoner trenger man minimum [ ] (3) instruksjoner.

Maks poeng: 10

12 På hvilke steder er det best å sette inn pipeline-registre for å få en 5-stegs pipelinet prosessor?



Velg ett eller flere alternativer

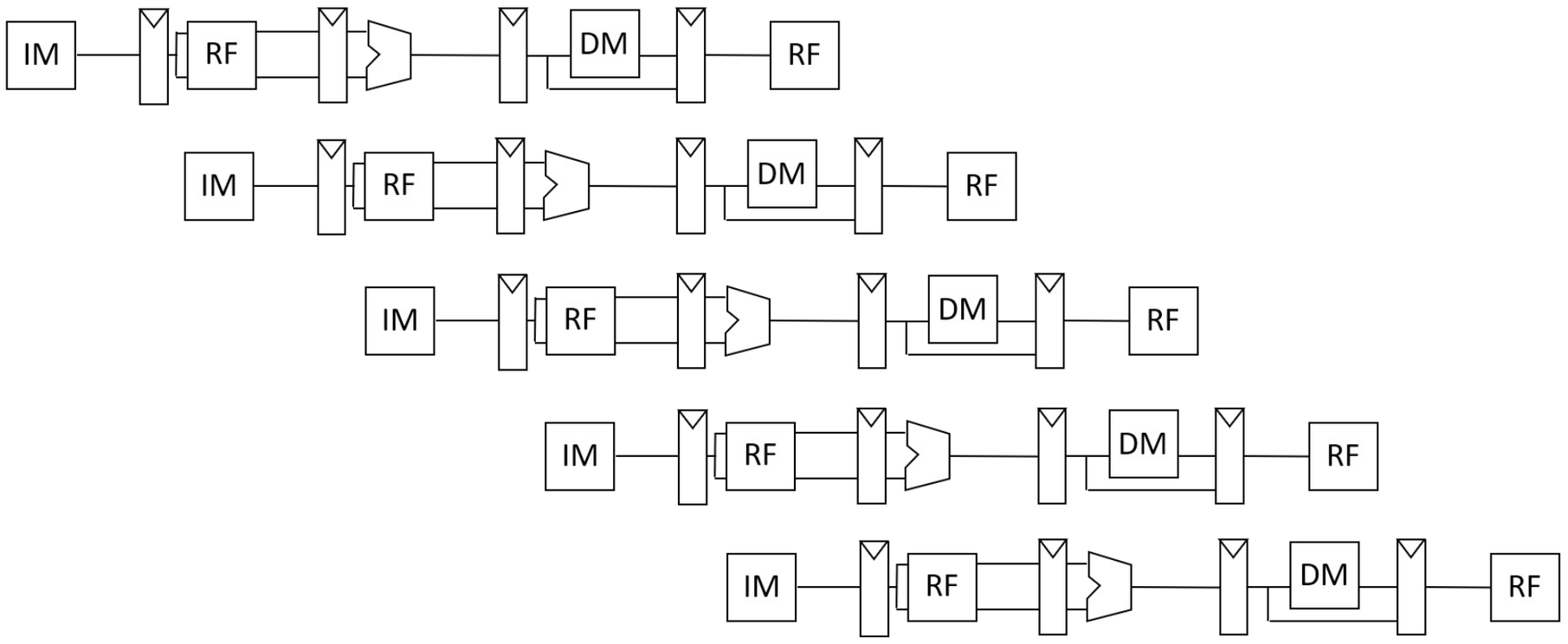
- [ ] ACDFG
- [ ] ACEFH
- [ ] CEFH
- [ ] ABCEF
- [x] BDFG
- [ ] BCDG

Maks poeng: 5

13 Gitt ARM-assemblerprogrammet:

```
MOV R3, #28
SUB R0, R3, #3
LDR R4, [R0, #14]
STR R5, [R3, #36]
AND R2, R0, R4
```

Hvilke(t) register(/registre) skrives og / eller leses i sykel 5?
Anta 5-stegs pipelinet prosessor med hasardenhet som i boka.
Du kan bruke illustrasjonen som tankehjelp / for å kladde.

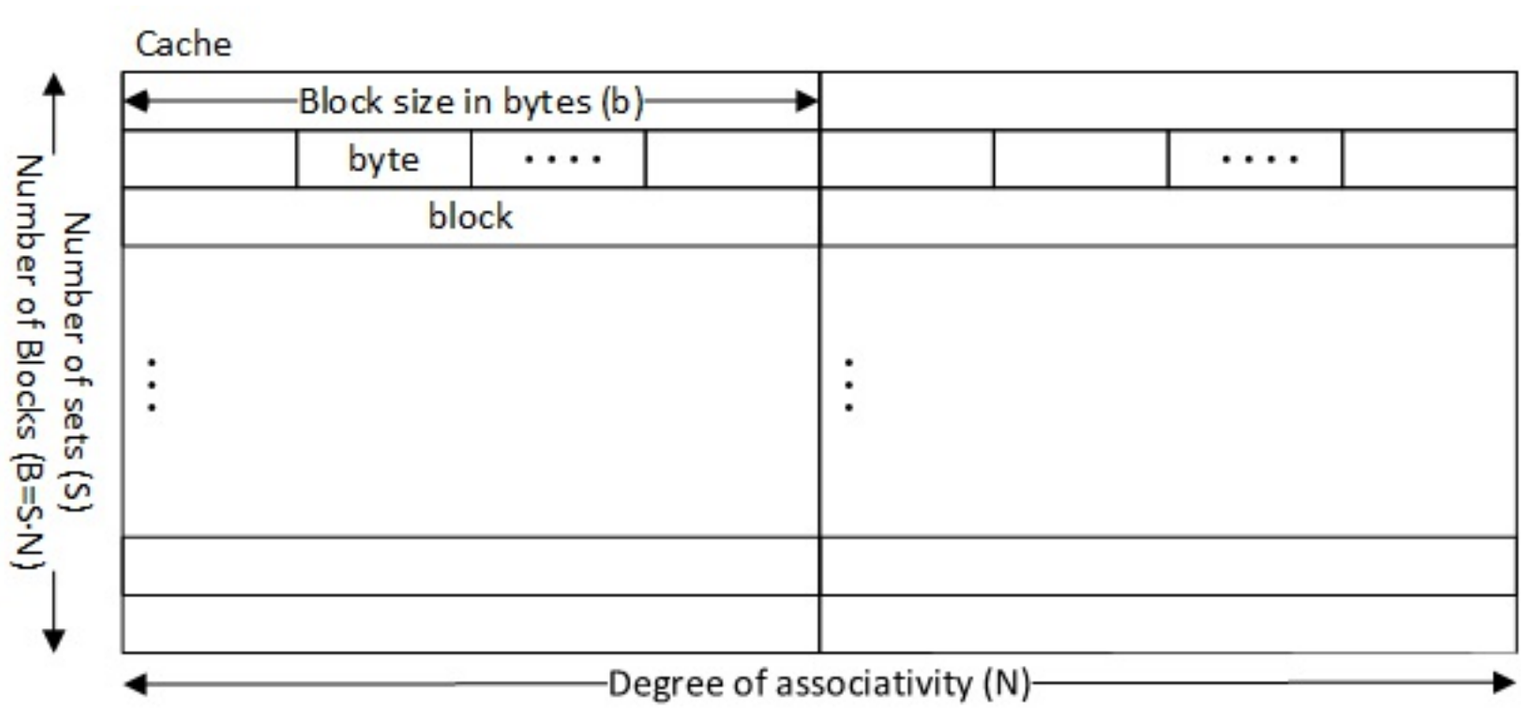


Velg ett eller flere alternativer

- [ ] R5
- [ ] R4
- [ ] R0
- [ ] R2
- [x] R3 ✔

Maks poeng: 5

14



Figur av en generisk cache.

Gitt følgende ARM assembler kode:

```
(1)              MOV  R0, #5
(2)              MOV  R1, #0
(3) LOOP         CMP  R0, 0
(4)              BEQ  DONE
(5)              LDR  R2, [R1, #0x4]
(6)              LDR  R3, [R1, #0x8]
(7)              LDR  R4, [R1, #0x24]
(8)              SUB  R0, R0, #1
(9)              B    LOOP
(10) DONE
```

I denne oppgaven skal du sammenligne ytelsen til to cachesystemer:

1) En direktekoblet (direct mapped) cache med blokkstørrelse på 4 ord.

2) En 2-way set associative cache med blokkstørrelse på ett ord.

Begge cachene har ordstørrelse på 4 byte og 8 sett.

a) Hva blir miss-rate for henholdsvis cache 1) og 2) ? (Skriv som brøk)

Vi endrer linje 6 til

```
(6) LDR   R3, [R1, #0x88]
```

b) Vil miss-ratene for de to cachene endre seg med den nye koden? Begrunn svaret.

For hver av cachene innfører vi en nivå 2 cache (level 2) med 16 ganger så mange sett (128 sett), men ellers samme oppbygning som 1) og 2).
Vi antar at miss penalty er på 100 klokkesykler om vi må lese fra RAM og 10 klokkesykler om vi må lese fra level 2 cache.

c) I hvilken grad vil cachesystemene 1) og 2) dra nytte av nivå 2 cachen, gitt at vi benytter den samme assemblerkoden som i b)? Begrunn svaret.

Skriv ditt svar her...

Maks poeng: 15

Question 10
Attached

Data-processing instructions

Name	Description	Operation
`ADD Rd, Rn, Src2`	Add (`+`)	`Rd = Rn + Src2`
`SUB Rd, Rn, Src2`	Subtract (`-`)	`Rd = Rn - Src2`
`AND Rd, Rn, Src2`	Bitwise AND (`&`)	`Rd = Rn & Src2`
`ORR Rd, Rn, Src2`	Bitwise OR (`|`)	`Rd = Rn | Src2`
`EOR Rd, Rn, Src2`	Bitwise Exclusive OR (`^`)	`Rd = Rn ^ Src2`
`BIC Rd, Rn, Src2`	Bitwise Clear	`Rd = Rn & ~Src2`
`MVN Rd, Rn, Src2`	Bitwise NOT (`~`)	`Rd = ~Rn`
`LSL Rd, Rn, Src2`	Logical Shift Left (`<<`)	`Rd = Rn << Src2`
`LSR Rd, Rn, Src2`	Logical Shift Right (`>>`)	`Rd = Rn >> Src2`
`MOV Rd, Src2`	Move (`=`)	`Rd = Src2`
`CMP Rd, Src2`	Compare	Set flags (see below) based on `Rd - Src2`

Remember that we can also set condition flags by appending an `S` to the end of our Data-processing instructions.

Name	Description
`ADDS Rd, Rn, Src2`	Add (as above) **and** set condition flags
`SUBS Rd, Rn, Src2`	Subtract (as above) **and** set condition flags
`ANDS Rd, Rn, Src2`	Bitwise AND (as above) **and** set condition flags

Multiply instructions

Name	Description	Operation
`MUL Rd, Rn, Rm`	Multiply (`*`)	`Rd = Rn * Rm`
`MULS Rd, Rn, Rm`	Multiply (`*`) **and** set condition flags	`Rd = Rn * Rm`
`MLA Rd, Rn, Rm, Ra`	Multiply and Accumulate	`Rd = (Rn * Rm) + Ra`

Memory instructions

Name	Description	Operation
`STR Rd, [Rn, ± Src2]`	Store Register	`Mem[Adr] = Rd`
`LDR Rd, [Rn, ± Src2]`	Load Register	`Rd = Mem[Adr]`

Branch instructions

Name	Description	Operation
`B label`	Branch	`PC = (PC + 8) + imm24 << 2`
`BL label`	Branch and Link	`LR = (PC + 8) - 4; PC = (PC + 8) + im24 << 2`
`BX Rd`	Branch and eXchange	Branch to address pointed to in `Rd` (used for return)

Condition flags

Flag	Name	Description
`N`	Negative	Instruction result is negative
`Z`	Zero	Instruction result is zero
`C`	Carry	Instruction caused a carry out
`V`	oVerflow	Instruction caused an overflow

Condition mnemonics

Mnemonic	Name	CondEx
`EQ`	Equal	`Z`
`NE`	Not Equal	`!Z`
`CS/HS`	Carry set / unsigned higher or same	`C`
`CC/LO`	Carry clear / unsigned lower	`!C`
`MI`	Minus / negative	`N`
`PL`	Plus / Positive *or* zero	`!N`
`VS`	Overflow	`V`
`VC`	No overflow	`!V`
`HI`	Unsigned higher	`!Z AND C`
`LS`	Unsigned lower or same	`Z OR !C`
`GE`	Signed greater than or equal	`!N XOR !V`
`LT`	Signed less than	`N XOR V`
`GT`	Signed greater than	`!Z AND (!N XOR !V)`
`LE`	Signed less than or equal	`Z OR (N XOR V)`

Question 11
Attached

Data-processing instructions

Name	Description	Operation
`ADD Rd, Rn, Src2`	Add (`+`)	`Rd = Rn + Src2`
`SUB Rd, Rn, Src2`	Subtract (`-`)	`Rd = Rn - Src2`
`AND Rd, Rn, Src2`	Bitwise AND (`&`)	`Rd = Rn & Src2`
`ORR Rd, Rn, Src2`	Bitwise OR (`|`)	`Rd = Rn | Src2`
`EOR Rd, Rn, Src2`	Bitwise Exclusive OR (`^`)	`Rd = Rn ^ Src2`
`BIC Rd, Rn, Src2`	Bitwise Clear	`Rd = Rn & ~Src2`
`MVN Rd, Rn, Src2`	Bitwise NOT (`~`)	`Rd = ~Rn`
`LSL Rd, Rn, Src2`	Logical Shift Left (`<<`)	`Rd = Rn << Src2`
`LSR Rd, Rn, Src2`	Logical Shift Right (`>>`)	`Rd = Rn >> Src2`
`MOV Rd, Src2`	Move (`=`)	`Rd = Src2`
`CMP Rd, Src2`	Compare	Set flags (see below) based on `Rd - Src2`

Remember that we can also set condition flags by appending an `S` to the end of our Data-processing instructions.

Name	Description
`ADDS Rd, Rn, Src2`	Add (as above) **and** set condition flags
`SUBS Rd, Rn, Src2`	Subtract (as above) **and** set condition flags
`ANDS Rd, Rn, Src2`	Bitwise AND (as above) **and** set condition flags

Multiply instructions

Name	Description	Operation
`MUL Rd, Rn, Rm`	Multiply (`*`)	`Rd = Rn * Rm`
`MULS Rd, Rn, Rm`	Multiply (`*`) **and** set condition flags	`Rd = Rn * Rm`
`MLA Rd, Rn, Rm, Ra`	Multiply and Accumulate	`Rd = (Rn * Rm) + Ra`

Memory instructions

Name	Description	Operation
`STR Rd, [Rn, ± Src2]`	Store Register	`Mem[Adr] = Rd`
`LDR Rd, [Rn, ± Src2]`	Load Register	`Rd = Mem[Adr]`

Branch instructions

Name	Description	Operation
`B label`	Branch	`PC = (PC + 8) + imm24 << 2`
`BL label`	Branch and Link	`LR = (PC + 8) - 4; PC = (PC + 8) + im24 << 2`
`BX Rd`	Branch and eXchange	Branch to address pointed to in `Rd` (used for return)

Condition flags

Flag	Name	Description
`N`	Negative	Instruction result is negative
`Z`	Zero	Instruction result is zero
`C`	Carry	Instruction caused a carry out
`V`	oVerflow	Instruction caused an overflow

Condition mnemonics

Mnemonic	Name	CondEx
`EQ`	Equal	`Z`
`NE`	Not Equal	`!Z`
`CS/HS`	Carry set / unsigned higher or same	`C`
`CC/LO`	Carry clear / unsigned lower	`!C`
`MI`	Minus / negative	`N`
`PL`	Plus / Positive *or* zero	`!N`
`VS`	Overflow	`V`
`VC`	No overflow	`!V`
`HI`	Unsigned higher	`!Z AND C`
`LS`	Unsigned lower or same	`Z OR !C`
`GE`	Signed greater than or equal	`!N XOR !V`
`LT`	Signed less than	`N XOR V`
`GT`	Signed greater than	`!Z AND (!N XOR !V)`
`LE`	Signed less than or equal	`Z OR (N XOR V)`

Question 13
Attached

Data-processing instructions

Name	Description	Operation
`ADD Rd, Rn, Src2`	Add (`+`)	`Rd = Rn + Src2`
`SUB Rd, Rn, Src2`	Subtract (`-`)	`Rd = Rn - Src2`
`AND Rd, Rn, Src2`	Bitwise AND (`&`)	`Rd = Rn & Src2`
`ORR Rd, Rn, Src2`	Bitwise OR (`|`)	`Rd = Rn | Src2`
`EOR Rd, Rn, Src2`	Bitwise Exclusive OR (`^`)	`Rd = Rn ^ Src2`
`BIC Rd, Rn, Src2`	Bitwise Clear	`Rd = Rn & ~Src2`
`MVN Rd, Rn, Src2`	Bitwise NOT (`~`)	`Rd = ~Rn`
`LSL Rd, Rn, Src2`	Logical Shift Left (`<<`)	`Rd = Rn << Src2`
`LSR Rd, Rn, Src2`	Logical Shift Right (`>>`)	`Rd = Rn >> Src2`
`MOV Rd, Src2`	Move (`=`)	`Rd = Src2`
`CMP Rd, Src2`	Compare	Set flags (see below) based on `Rd - Src2`

Remember that we can also set condition flags by appending an `S` to the end of our Data-processing instructions.

Name	Description
`ADDS Rd, Rn, Src2`	Add (as above) **and** set condition flags
`SUBS Rd, Rn, Src2`	Subtract (as above) **and** set condition flags
`ANDS Rd, Rn, Src2`	Bitwise AND (as above) **and** set condition flags

Multiply instructions

Name	Description	Operation
`MUL Rd, Rn, Rm`	Multiply (`*`)	`Rd = Rn * Rm`
`MULS Rd, Rn, Rm`	Multiply (`*`) **and** set condition flags	`Rd = Rn * Rm`
`MLA Rd, Rn, Rm, Ra`	Multiply and Accumulate	`Rd = (Rn * Rm) + Ra`

Memory instructions

Name	Description	Operation
`STR Rd, [Rn, ± Src2]`	Store Register	`Mem[Adr] = Rd`
`LDR Rd, [Rn, ± Src2]`	Load Register	`Rd = Mem[Adr]`

Branch instructions

Name	Description	Operation
`B label`	Branch	`PC = (PC + 8) + imm24 << 2`
`BL label`	Branch and Link	`LR = (PC + 8) - 4; PC = (PC + 8) + im24 << 2`
`BX Rd`	Branch and eXchange	Branch to address pointed to in `Rd` (used for return)

Condition flags

Flag	Name	Description
`N`	Negative	Instruction result is negative
`Z`	Zero	Instruction result is zero
`C`	Carry	Instruction caused a carry out
`V`	oVerflow	Instruction caused an overflow

Condition mnemonics

Mnemonic	Name	CondEx
`EQ`	Equal	`Z`
`NE`	Not Equal	`!Z`
`CS/HS`	Carry set / unsigned higher or same	`C`
`CC/LO`	Carry clear / unsigned lower	`!C`
`MI`	Minus / negative	`N`
`PL`	Plus / Positive *or* zero	`!N`
`VS`	Overflow	`V`
`VC`	No overflow	`!V`
`HI`	Unsigned higher	`!Z AND C`
`LS`	Unsigned lower or same	`Z OR !C`
`GE`	Signed greater than or equal	`!N XOR !V`
`LT`	Signed less than	`N XOR V`
`GT`	Signed greater than	`!Z AND (!N XOR !V)`
`LE`	Signed less than or equal	`Z OR (N XOR V)`

Question 14
Attached

Data-processing instructions

Name	Description	Operation
`ADD Rd, Rn, Src2`	Add (`+`)	`Rd = Rn + Src2`
`SUB Rd, Rn, Src2`	Subtract (`-`)	`Rd = Rn - Src2`
`AND Rd, Rn, Src2`	Bitwise AND (`&`)	`Rd = Rn & Src2`
`ORR Rd, Rn, Src2`	Bitwise OR (`|`)	`Rd = Rn | Src2`
`EOR Rd, Rn, Src2`	Bitwise Exclusive OR (`^`)	`Rd = Rn ^ Src2`
`BIC Rd, Rn, Src2`	Bitwise Clear	`Rd = Rn & ~Src2`
`MVN Rd, Rn, Src2`	Bitwise NOT (`~`)	`Rd = ~Rn`
`LSL Rd, Rn, Src2`	Logical Shift Left (`<<`)	`Rd = Rn << Src2`
`LSR Rd, Rn, Src2`	Logical Shift Right (`>>`)	`Rd = Rn >> Src2`
`MOV Rd, Src2`	Move (`=`)	`Rd = Src2`
`CMP Rd, Src2`	Compare	Set flags (see below) based on `Rd - Src2`

Remember that we can also set condition flags by appending an `S` to the end of our Data-processing instructions.

Name	Description
`ADDS Rd, Rn, Src2`	Add (as above) **and** set condition flags
`SUBS Rd, Rn, Src2`	Subtract (as above) **and** set condition flags
`ANDS Rd, Rn, Src2`	Bitwise AND (as above) **and** set condition flags

Multiply instructions

Name	Description	Operation
`MUL Rd, Rn, Rm`	Multiply (`*`)	`Rd = Rn * Rm`
`MULS Rd, Rn, Rm`	Multiply (`*`) **and** set condition flags	`Rd = Rn * Rm`
`MLA Rd, Rn, Rm, Ra`	Multiply and Accumulate	`Rd = (Rn * Rm) + Ra`

Memory instructions

Name	Description	Operation
`STR Rd, [Rn, ± Src2]`	Store Register	`Mem[Adr] = Rd`
`LDR Rd, [Rn, ± Src2]`	Load Register	`Rd = Mem[Adr]`

Branch instructions

Name	Description	Operation
`B label`	Branch	`PC = (PC + 8) + imm24 << 2`
`BL label`	Branch and Link	`LR = (PC + 8) - 4; PC = (PC + 8) + im24 << 2`
`BX Rd`	Branch and eXchange	Branch to address pointed to in `Rd` (used for return)

Condition flags

Flag	Name	Description
`N`	Negative	Instruction result is negative
`Z`	Zero	Instruction result is zero
`C`	Carry	Instruction caused a carry out
`V`	oVerflow	Instruction caused an overflow

Condition mnemonics

Mnemonic	Name	CondEx
`EQ`	Equal	`Z`
`NE`	Not Equal	`!Z`
`CS/HS`	Carry set / unsigned higher or same	`C`
`CC/LO`	Carry clear / unsigned lower	`!C`
`MI`	Minus / negative	`N`
`PL`	Plus / Positive *or* zero	`!N`
`VS`	Overflow	`V`
`VC`	No overflow	`!V`
`HI`	Unsigned higher	`!Z AND C`
`LS`	Unsigned lower or same	`Z OR !C`
`GE`	Signed greater than or equal	`!N XOR !V`
`LT`	Signed less than	`N XOR V`
`GT`	Signed greater than	`!Z AND (!N XOR !V)`
`LE`	Signed less than or equal	`Z OR (N XOR V)`

Question 3
Attached

Theorems

Number	Theorem	Dual	Name
T1	B • 1 = B	B + 0 = B	Identity
T2	B • 0 = 0	B + 1 = 1	Null Element
T3	B • B = B	B + B = B	Idempotency
T4	(B')' = B		Involution
T5	B • B' = 0	B + B' = 1	Complements

#	Theorem	Dual	Name
T6	B•C = C•B	B+C = C+B	Commutativity
T7	(B•C) • D = B • (C•D)	(B + C) + D = B + (C + D)	Associativity
T8	B • (C + D) = (B•C) + (B•D)	B + (C•D) = (B+C) (B+D)	Distributivity
T9	B • (B+C) = B	B + (B•C) = B	Covering
T10	(B•C) + (B•$\overline{C}$) = B	(B+C) • (B+$\overline{C}$) = B	Combining
T11	(B•C) + ($\overline{B}$•D) + (C•D) = (B•C) + ($\overline{B}$•D)	(B+C) • ($\overline{B}$+D) • (C+D) = (B+C) • ($\overline{B}$+D)	Consensus

#	Theorem	Dual	Name
T12	$\overline{B_0 \bullet B_1 \bullet B_2 \ldots}$ = $\overline{B_0}+\overline{B_1}+\overline{B_2}\ldots$	$\overline{B_0+B_1+B_2\ldots}$ = $\overline{B_0}\bullet\overline{B_1}\bullet\overline{B_2}\ldots$	DeMorgan's Theorem