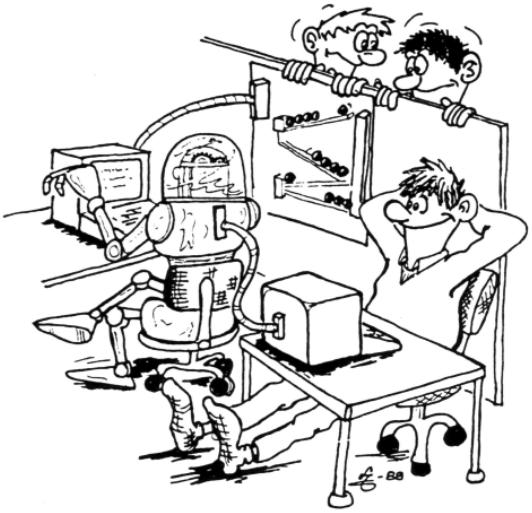
Industriautomation

Tentamen SSY 066, fredag 18/01–2019, fm

Lärare: Kristofer Bengtsson, 0768-979561



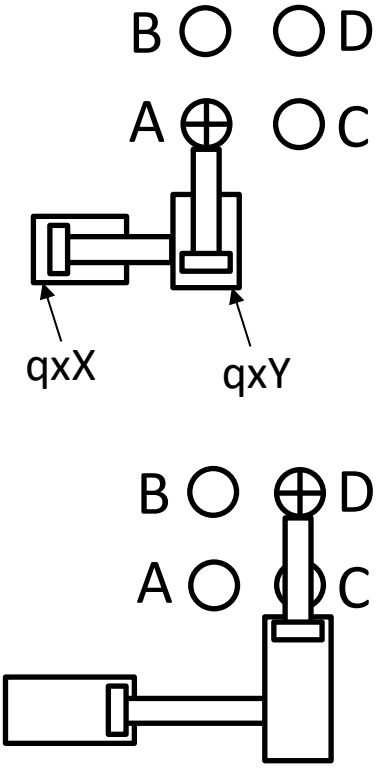
Fullständig lösning ska lämnas på samtliga uppgifter. I förekommande fall av tvetydigt formulerade tentamensuppgifter ska den föreslagna lösningen och eventuella antaganden motiveras. Examinator förbehåller sig rätten att godkänna rimligheten i antaganden och motiveringar.

Totalt omfattar tentamen 40 poäng. För betygen tre, fyra och fem krävs 16, 24 resp. 32 poäng. Lösningar anslås direkt efter tentatillfället på kursens hemsida i Pingpong. Granskning av rättningen sker fredag den 8 februari kl. 12:30 – 13:30 på institutionen.

OBS. Inga hjälpmedel är tillåtna.

Uppgift 1

I denna uppgift skall vi styra ett litet system som kan flytta sitt verktyg till fyra olika positioner, A, B, C, D. Systemet består av två cylindrar, X och Y där cylinder X håller i cylinder Y som i sin tur håller i verktyget. När utsignalen ***qxX*** blir sann åker cylinder X ut och flyttar cylinder Y till höger. När ***qxY*** blir sann åker verktyget upp till övre raden. När både ***qxX*** och ***qxY*** är falska befinner sig verktyget i position A, som på övre bilden, och när båda är sanna, befinner sig verktyget i position D, som på nedre bilden (den är vid B när bara **qxY** är sann och vid C då bara **qxX** är sann).



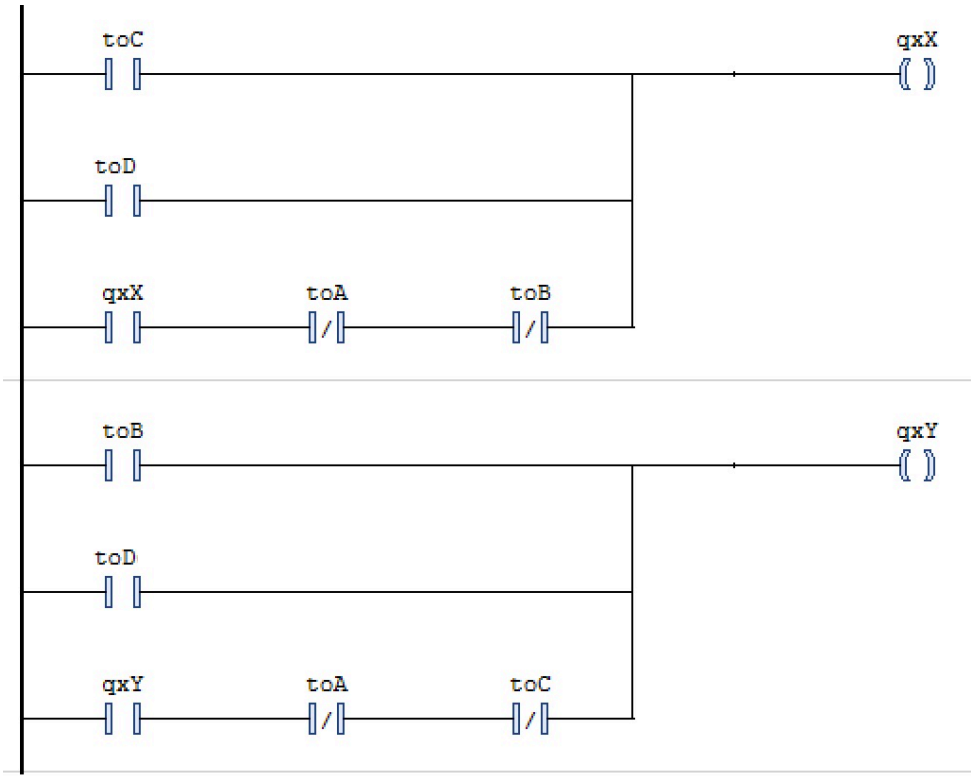
Din uppgift är att implementera ladderkod som styr verktyget till de olika positionerna beroende på de fyra insignalerna, ***toA***, ***toB***, ***toC***, och ***toD***. Det är alltså en annan kod som sätter dessa signaler och din uppgift är att sätta utsignalerna ***qxX*** och ***qxY*** korrekt.

Om tex ***toD*** blir sann, skall båda cylindrarna åka ut. Om ingen av insignalerna är sanna skall verktyget behålla sin nuvarande position (alltså behålla värdet på ***qxX*** och ***qxY***). I exemplet med ***toD***, skall båda cylindrarna stanna ute ända tills en ny insignal blir sann, även om toD blir falsk. Endast en insignal kommer att vara sann åt gången.

Implementera styrningen med ladderlogik, men du får **inte använda set- / reset-logik**. Det enklaste är att implementera två laddernätverk där ett av dem sätter **qxX** och det andra **qxY**.

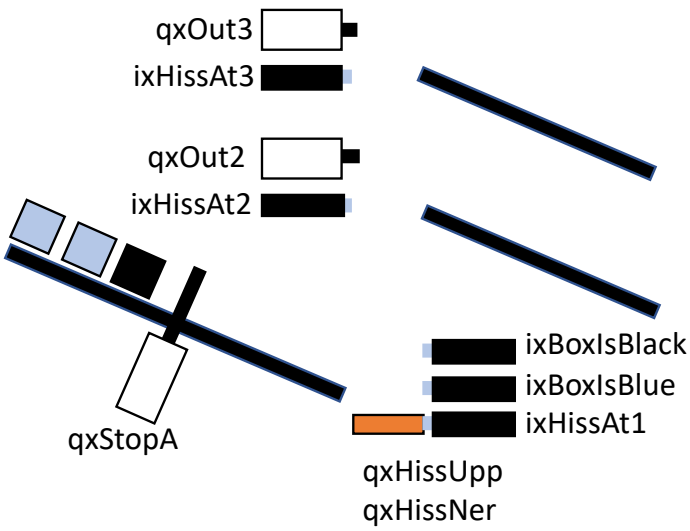
(5 poäng)

Möjlig lösning uppgift 1



Uppgift 2

Nu skall vi styra ett system med ladder och SFC. Till höger ser du en skiss över systemet. Klossar, som antingen är blåa eller svarta glider in på banan längst till vänster av sig själva. De skall transporteras till olika våningar med hjälp av en hiss (den oranga rutan), blåa klossar skall till våning 2 och svarta till våning 3.



Först skall systemet se till att endast en kloss åker in i hissen åt gången. Det görs med StopA som styrs med **qxStopA**. När **qxStopA** är sann är dess stop-pinne indragen i cylindern, så att en kloss kan åka förbi och när de är falska är den som på bilden ute och stoppar klossarna. **qxStopA** skall vara sann i 1 sekund så att bara en kloss hinner ner förbi stoppet. Observera att detta endast får ske när hissen är på plats på våning 1 och då det inte finns någon kloss på hissen. Direkt när det är ledigt igen, skall en ny kloss släppas fram. Det tar mer än 1 sekund för en kub att glida från StopA till hissen.

Att en kloss är på hissen vid våning 1 syns genom att antingen sensorn **ixBoxIsBlue** (klossen är blå) eller sensorn **ixBoxIsBlack** (klossen är svart) är sann. Bara en är sann åt gången. Om klossen är blå skall hissen åka upp till våning 2 (vilket visas med sensorn **ixHissAt2**) eller om den är svart till våning 3 (vilket visas med sensorn **ixHissAt3**). Hissen styrs med **qxHissUpp** och **qxHissNer**, som när upp är sann åker den upp, när ner är sann åker den ner och om båda är falska så stannar den på den positionen den befinner sig.

När en kloss (och hissen) är på våning 2 eller 3 skall respektive cylinder (**qxOut2** eller **qxOut3**) putta ut cylindern under en sekund. Sedan skall hissen åka ner och vänta på nästa kloss.

Din uppgift är att implementera styrningen av **qxStopA** med hjälp av ladderkod.

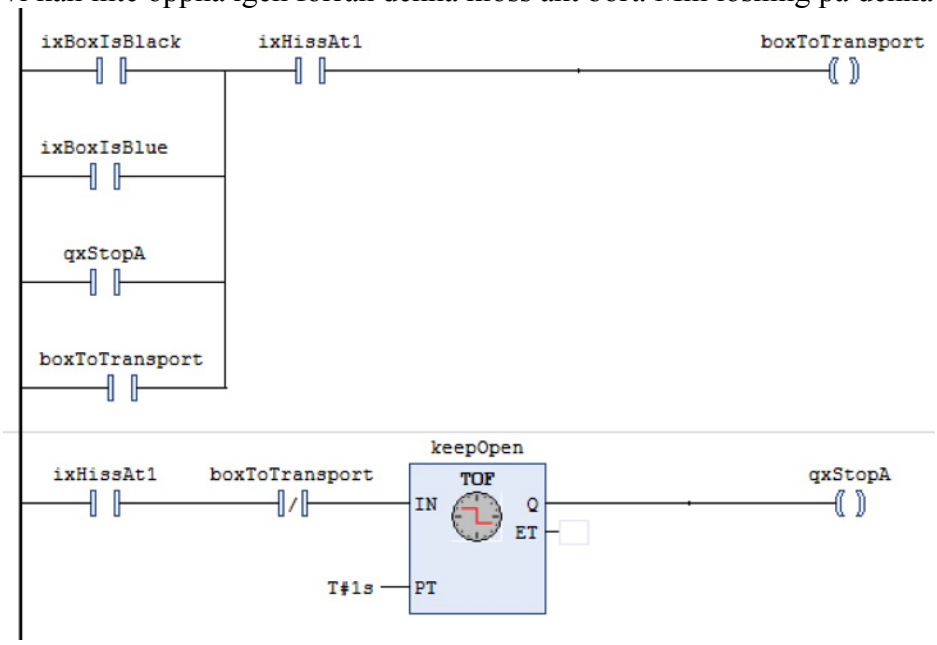
Hissen och puttarnas utgångar, **qxHissUpp, qxHissNer**, **qxOut2** och **qxOut3,** skall styras med en SFC.

Du kan använda alla insignaler i båda ladder och SFCn och du kan även deklarera egna variabler som du kan använda i båda koderna. Du behöver inte skapa funktionsblock utan kan direkt skriva din ladder kod och sedan din SFC kod.

(6 poäng)

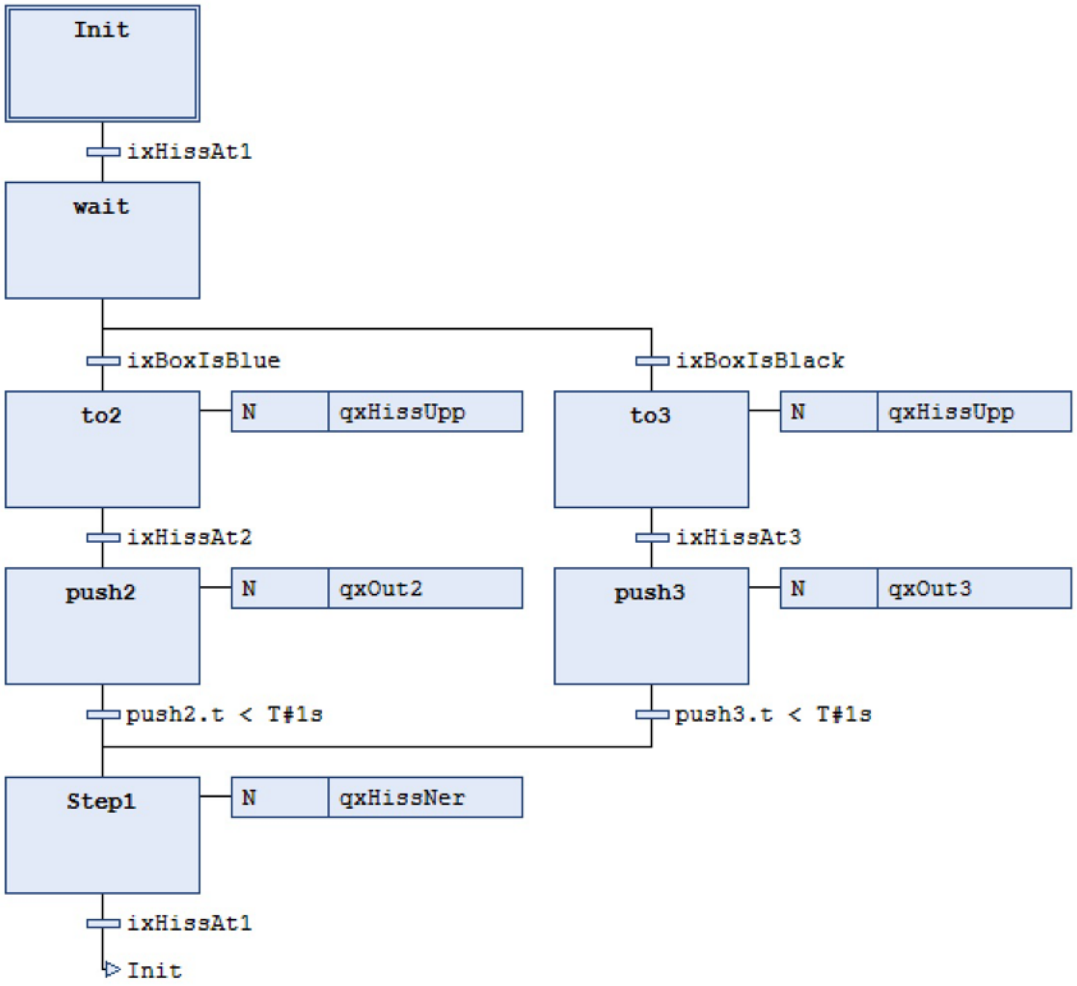
Möjligt svar uppgift 2

För att styra när en kloss skall släppas fram behöver vi hålla ordning på om hissen är på våning 1, att det inte finns en kloss på hissen redan och att det inte finns en kloss som just nu åker fram mot hissen. Direkt efter att vi släppt på en kloss vet vi ju inte när den är framme, så vi kan inte öppna igen förrän denna kloss åkt bort. Min lösning på denna styrning blev såhär:



Där boxToTransport är en egen variabel som håller iordning på om det finns en kloss på hissen eller påväg till hissen. qxStopA styrs med en off-timer. När qxStop drar ner sitt stop kommer boxToTransport bli sann och IN på blocket keepOpen blir direkt falskt. Men keepOpen håller utsignalen sann 1 sekund.

Se SFC på nästa sida. Den är ganska självförklarande.


Init
ixHissAt1
wait
ixBoxIsBlue
ixBoxIsBlack
to2
N
qxHissUpp
to3
N
qxHissUpp
ixHissAt2
ixHissAt3
push2
N
qxOut2
push3
N
qxOut3
push2.t < T#1s
push3.t < T#1s
Step1
N
qxHissNer
ixHissAt1
Init

Uppgift 3

En man som hade en varg, en get och ett kålhuvud skulle passera en flod med en liten båt. Han kunde inte ta med sig mer än en av dem per gång. Vargen och geten kunde inte lämnas ensamma, inte heller geten och kålhuvudet. I denna uppgift skall vi undersöka hur han gick till väga för att komma över floden.

a) I första uppgiften får geten, vargen och kålhuvudet vara tillsammans utan restriktioner då vi skall studera alla möjliga tillstånd som kan uppstå. Du skall beskriva problemet med hjälp av variabler och operationer. Skriv ner vilka variabler du har och dess domän. Skriv också ner lämpliga operationer med guards och actions baserat på dina variabler så att tillståndet uppdateras på lämpligt sätt när en operation sker.

(4 poäng)

b) Beskriv ditt system och hur dina operationer uppdaterar tillståndet från uppgift **a** med hjälp av en graf. Men för att du skall slippa rita upp alla tillstånd så behöver du inte flytta kålhuvudet utan det är alltid kvar på en sida. En operation tar ingen tid utan är färdig direkt. Markera också de tillstånd där ditt system bryter mot specifikationen: *”Vargen och geten får inte lämnas ensamma, inte heller geten och kålhuvudet.”*

(4 poäng)

c) Uppdatera nu alla dina operationer från **a** om det behövs, så att det inte är möjligt att bryta mot specifikationen från **b**. Skriv ner dina nya operationer och svara sedan på frågan: Vilken är den kortaste operationssekvensen för att få över alla till andra sidan?

(4 poäng)

Möjlig lösning uppgift 3a

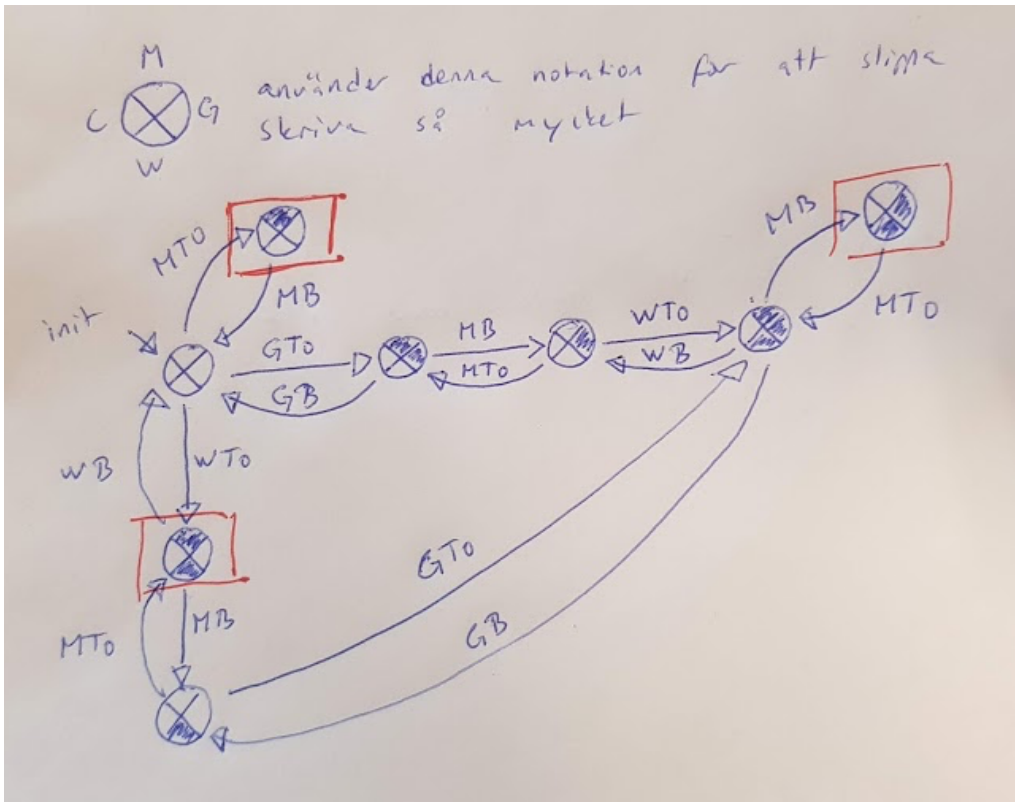
Det enklaste är att använda en boolsk variabel per ”sak” som är falsk om ”saken” är på första sidan och sann om den är på andra sidan. Målet är att alla kan komma över till andra sidan.

M: Bool // falsk om mannen är på första sidan och sann om mannen är på andra sidan
G: Bool // geten
W: Bool // vargen
C: Bool // kålhuvudet

Operationer
(använder bara ett precondition på formen *guard / actions* där actions separeras med ;)

MTo: $\neg M \ / \ M := true$ // Mannen åker själv över ån utan någon annan (Man To)
MB: $M \ / \ M := false$ // Mannen åker tillbaka över ån utan någon annan (Man Back)
GTo: $\neg G \wedge \neg M \ / \ G := true; M := true$ // Mannen ror över geten till andra sidan
GB: $G \wedge M \ / \ G := false; M := false$ // Mannen ror tillbaka geten till första sidan
WTo: $\neg W \wedge \neg M \ / \ W := true; M := true$ // Mannen ror över vargen till andra sidan
WB: $W \wedge M \ / \ W := false; M := false$ // Mannen ror tillbaka vargen till första sidan
CTo: $\neg C \wedge \neg M \ / \ C := true; M := true$ // Mannen ror över kålhuvudet till andra sidan
CB: $C \wedge M \ / \ C := false; M := false$ // Mannen ror tillbaka kålhuvudet till första sidan

Möjlig lösning uppgift 3b



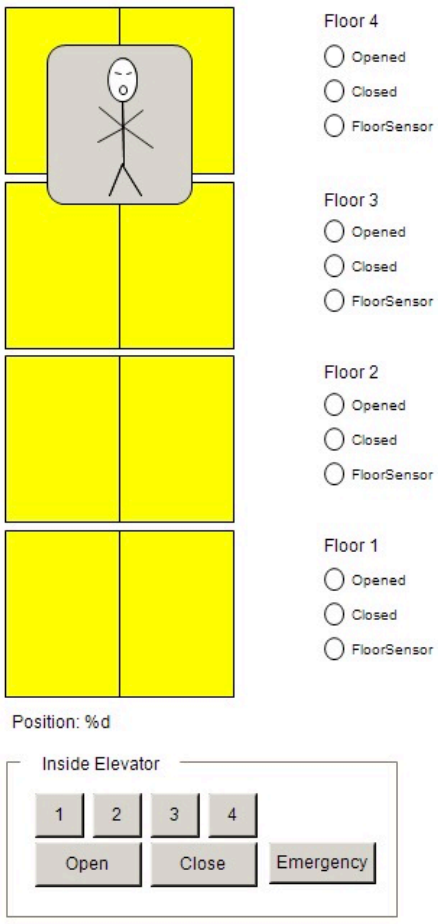
Möjlig lösning uppgift 3c
Geten får aldrig vara själv med vargen eller kålhuvudet så alla operationer som för mannen bort från geten utan att ha koll på de andra är problematiska. Tex för att mannen skall kunna åka över med MTo får inte geten vara kvar tillsammans med någon av de andra, alltså: $\neg(\neg G \wedge \neg W) \wedge \neg(\neg G \wedge \neg C)$ vilket förkortat blir $G \vee (W \wedge C)$. Geten kan vi dock flytta utan restriktioner.

MTo: $\boldsymbol{G} \vee (\boldsymbol{W} \wedge \boldsymbol{C}) \wedge \quad \neg M \;/\; M \coloneqq true$
MB: $\neg(\boldsymbol{G} \wedge (\boldsymbol{W} \vee \boldsymbol{C}) \wedge \; M \;/\; M \coloneqq false$
GTo: $\neg G \wedge \neg M \;/\; G \coloneqq true; M \coloneqq true$
GB: $G \wedge M \;/\; G \coloneqq false; M \coloneqq false$
WTo: $(\boldsymbol{G} \vee \boldsymbol{C}) \wedge \quad \neg W \wedge \neg M \;/\; W \coloneqq true; M \coloneqq true$
WB: $\neg(\boldsymbol{G} \wedge \boldsymbol{C}) \wedge \quad W \wedge M \;/\; W \coloneqq false; M \coloneqq false$
CTo: $(\boldsymbol{G} \vee \boldsymbol{W}) \wedge \quad \neg C \wedge \neg M \;/\; C \coloneqq true; M \coloneqq true$
CB: $\neg(\boldsymbol{G} \wedge \boldsymbol{W}) \wedge \quad C \wedge M \;/\; C \coloneqq false; M \coloneqq false$

Möjliga sekvenser:
$GTo \rightarrow MB \rightarrow WTo \rightarrow GB \rightarrow CTo \rightarrow MB \rightarrow GTo$
eller
$GTo \rightarrow MB \rightarrow CTo \rightarrow GB \rightarrow WTo \rightarrow MB \rightarrow GTo$

Uppgift 4

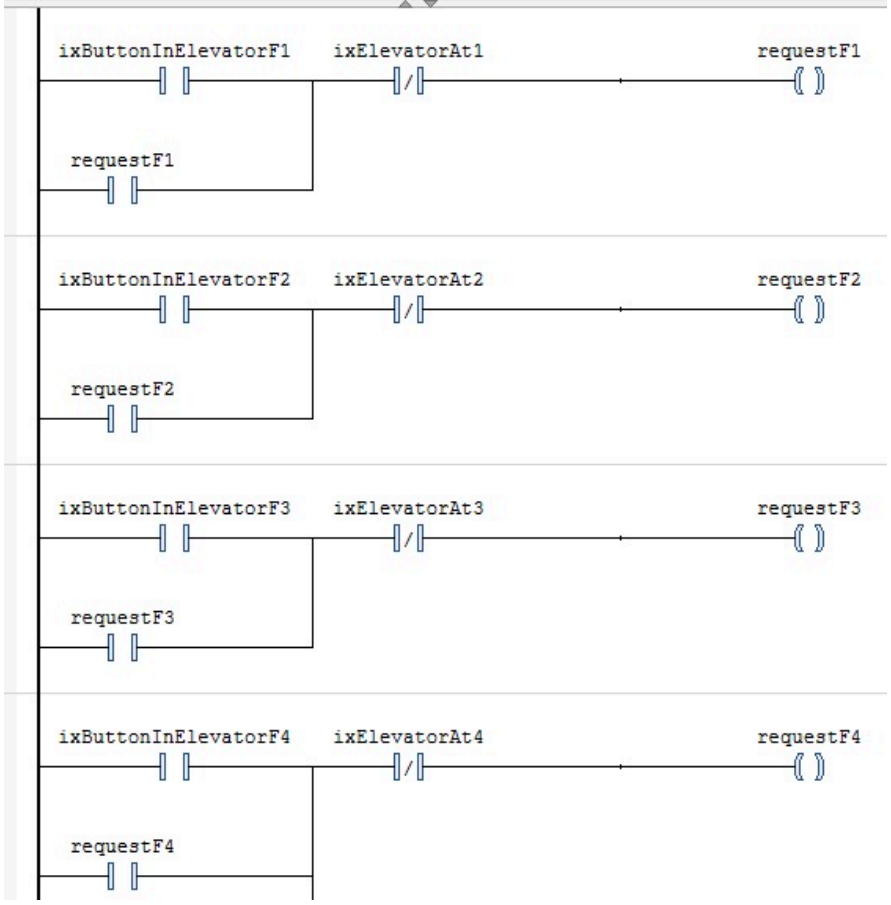
I denna sista uppgift skall vi jobba med styrningen av en hiss. Hissen har 4 våningar och dörrar på varje våning. Den styrs upp och ner med två utgångar: **qxElevatorGoUp** och **qxElevatorGoDown**. Om hissen är på en våning indikeras med variablerna **ixElevatorAt1, ixElevatorAt2, ixElevatorAt3, ixElevatorAt4**. Inne i hissen finns fyra knappar **ixButtonInElevatorF1, ixButtonInElevatorF2, ixButtonInElevatorF3, ixButtonInElevatorF4**, vilka användaren trycker på för att åka till respektive våning.

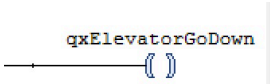


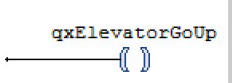
a) I första uppgiften skall du implementera de två nätverken som styr **qxElevatorGoUp** och **qxElevatorGoDown**. Någon annan har redan implementerat delar av koden som du hittar på nästa sida för att hantera knappar och skapa "requests" för att gå till olika våningar. Hissen kommer alltid starta vid en våning och er kod i denna uppgift behöver bara hantera en knapptryckning åt gången.

(6 poäng)

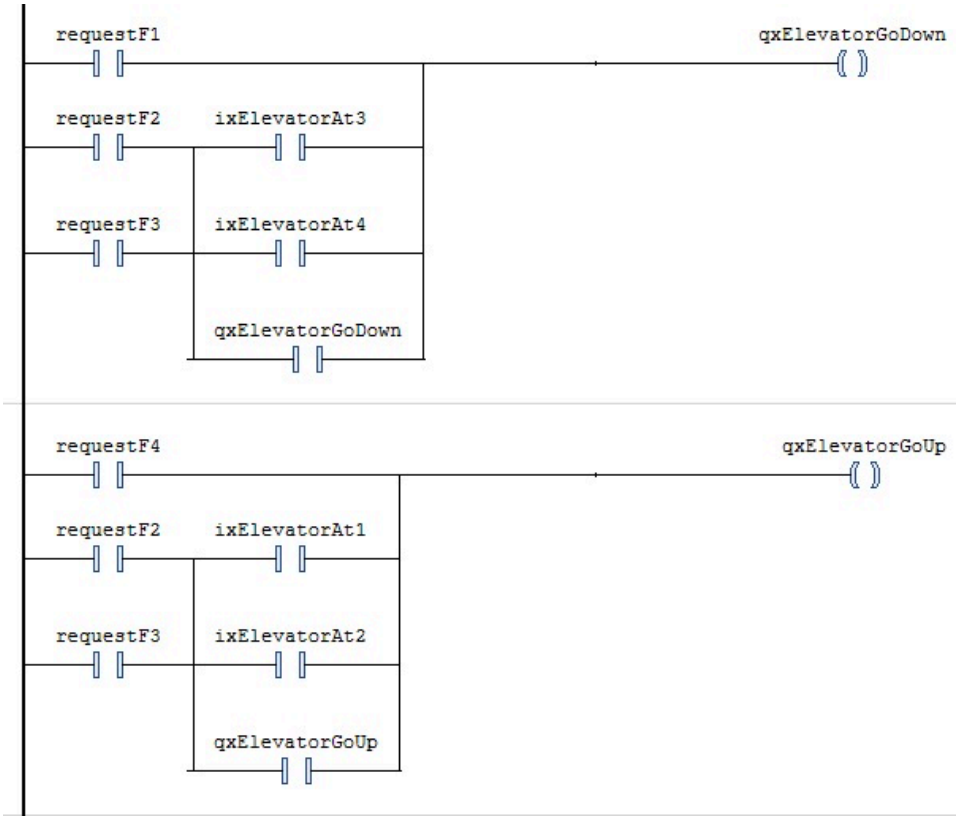
```
PROGRAM ElevatorControl
VAR
    requestF1: BOOL;
    requestF2: BOOL;
    requestF3: BOOL;
    requestF4: BOOL;
    init: BOOL := TRUE;
END_VAR
```







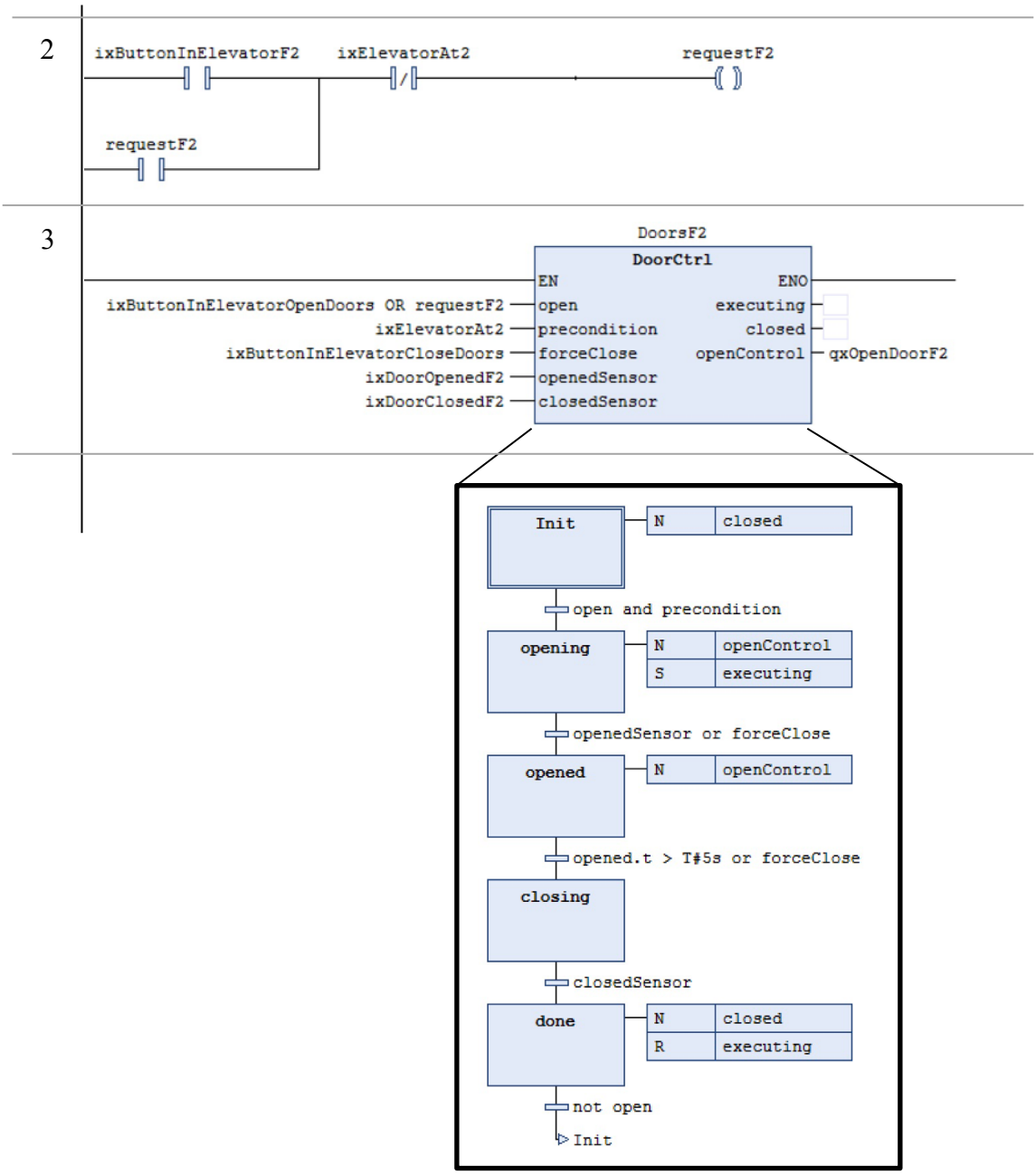
Möjlig lösning 4a:



Uppgift 4

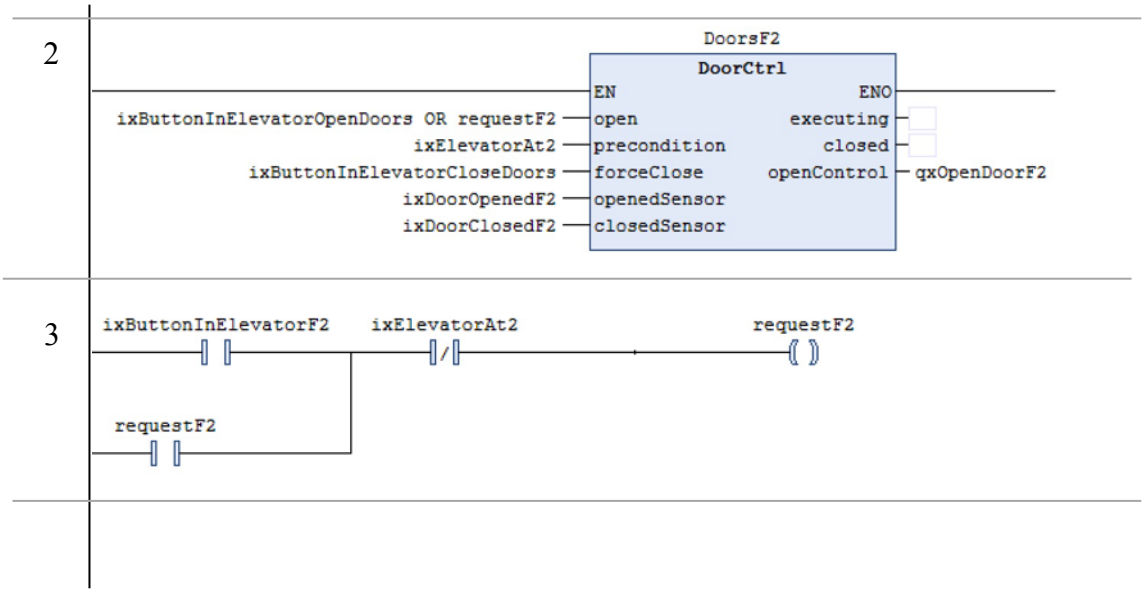
b) När nu hissen kan åka mellan våningar har något implementerat styrningen av dörrarna, se delar av koden nedan. För varje dörr finns funktionsblocket DoorCtrl (i koden visas endast blocket för dörr 2) som styr dörren och är implementerat med SFC. **ixButtonInElevatorOpenDoors** och **ixButtonInElevatorCloseDoors** är två knappar inne i hissen för att öppna och stänga dörren igen om den är vid en våning och **ixDoorOpenedF2**, **ixDoorClosedF2** är sensorer om dörren är öppen eller stängd och **qxOpenDoorF2** styr dörren. När en hiss kommer fram till en våning med en request som är sann (tex requestF2 i koden nedan), skall dörren öppnas automatiskt. Problemet är att koden inte fungerar som det är tänkt. Vad är det som är fel och hur kan det åtgärdas?

(5 poäng)

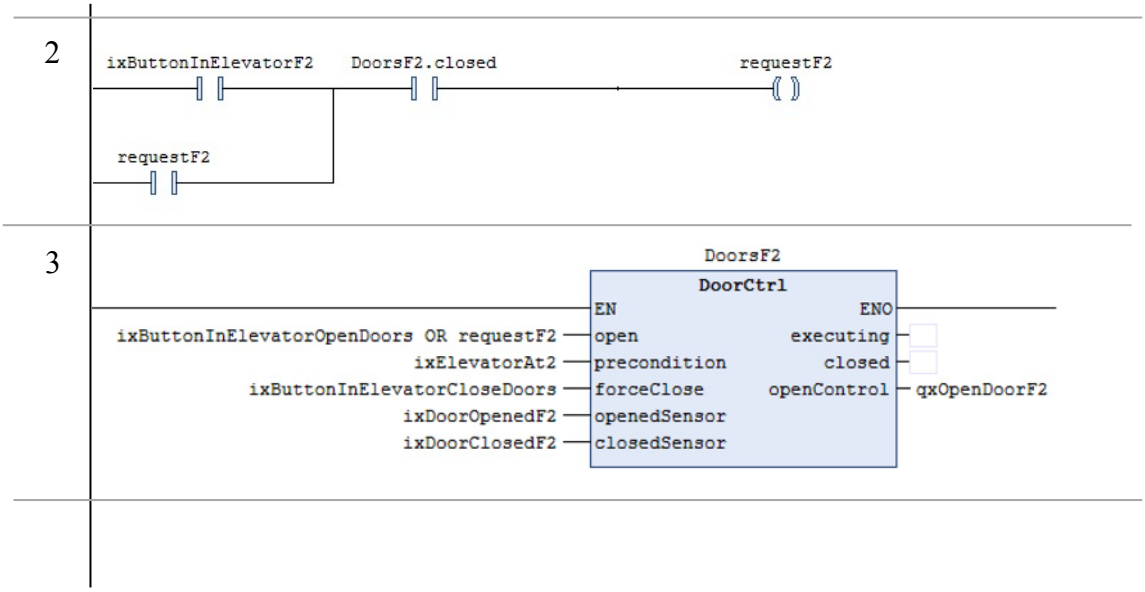


Möjlig lösning 4b

Problemet med koden är att requestF2 blir falsk när vi kommer fram till våning två då ixElevatorAt2 blir sann. Detta sker i nätverket ovan vårt funktionsblock. Så när DoorsF2 evalueras så kommer aldrig requestF2 (insignal open) och ixElevatorAt2 (insignal precondition) vara sanna samtidigt så SFC startar aldrig automatiskt utan vi behöver trycka på knappen öppna när hissen står på en våning. En enkel lösning är att helt enkelt byta ordning på våra två nätverk:



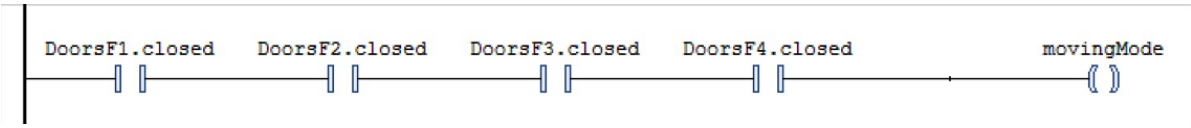
Eller kanske ännu bättre, ändra logiken så att requesten inte blir falsk förrän dörren har börjat att öppnas:



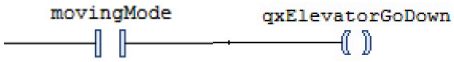
Uppgift 4

c) En sak som inte fungerar så bra med vår kod är om personen i hissen har klickat på flera knappar samtidigt. Om vi till exempel är vid våning 1 och klickar på 2 och 3 så skall hissen först åka till 2 och öppna dörren där. Om någon kommer in och klickar på 1 så skall den ändå fortsätta upp till 3. Det samma gäller så klart om vi är på väg ner. Troligen fungerar inte detta med din nuvarande kod.

För att hissen skall stanna på en våning för att öppna dörren har följande kod implementerats:



Movdär **movingMode** är en intern variabel som läggts in som en kontakt på dina två nätverk som styr **qxElevatorGoUp** och **qxElevatorGoDown** så att dessa inte kan vara sanna om **movingMode** är falsk. Då kommer hissen stanna på en våning och inte köra vidare förrän dörren har stängt.



Din uppgift är att uppdatera din ladderkod från uppgift **a** så att hissen kan hantera flera requests och fortsätter i den riktning den är på väg. Du kan använda set- och reset-logik om du vill och skapa nya nätverk efter behov. Glöm inte heller att använda **movingMode** så att hissen stannar på våningarna. Alla dörrar har var sitt funktionsblock som styr respektive dörr och ditt ändringsförslag från uppgift **b** är också implementerat.

För att förtydliga hissens funktion, din hiss skall klara följande how-to demo:
Börja vid våning 4. Tryck 3 och 1. När hissen stannar vid 3 och öppnar dörrarna, tryck 4. Hissen skall fortsätta ner till 1. När hissen vänder tillbaka upp från 1, tryck 2 och 3. När hissen stannar på våning 2, tryck 1. Hissen fortsätter upp till 3 och sedan 4 och sedan åker den till 1.

(6 poäng)

Möjlig lösning 4c:

Problemet är att lösningen i a och b tappar bort vilket håll hissen var på väg när den stannar vi en våning för att öppna dörren. Så länge som det finns request kvar åt det hållet vi åker skall vi komma ihåg det. Lösningen är ganska enkel, det är bara att i princip flytta logiken från nätverken som direkt styr hissen och införa ett minne som inte blir falsks när vi stannar (goingUP och goingDown nedan). Det är även viktigt att de två nätverken hindrar varandra. Det som behövs extra är att hindra tex down att fortsätta vara down om det kommer en request 2 eller 3 när vi är påväg ner och har paserat dem.
Även om man har lite fel i uppgift a kan man få full poäng på denna uppgift om man kommer på vad som behöver göras.