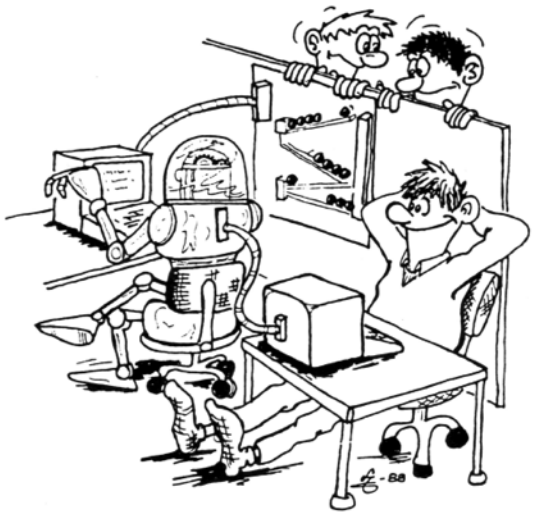
Industriautomation

Tentamen SSY 066, fredag 12/01–2018, fm

Lärare: Kristofer Bengtsson, 0768 979561



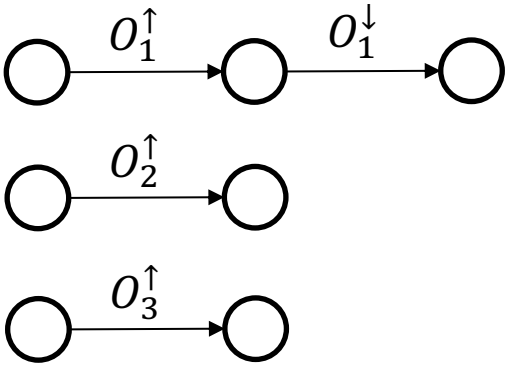
Fullständig lösning ska lämnas på samtliga uppgifter. I förekommande fall av tvetydigt formulerade tentamensuppgifter ska den föreslagna lösningen och eventuella antaganden motiveras. Examinator förbehåller sig rätten att godkänna rimligheten i antaganden och motiveringar.

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 måndag den 5 februari kl. 12:30 – 13:30 på institutionen.

OBS. Inga hjälpmedel är tillåtna.

Uppgift 1

Du har modellerat följande tre operationer:



Rita upp tillstånden och övergångarna som beskriver hur dessa operationer kan exekvera i förhållande till varandra (grafen som bildas när dessa operationer körs) om de tre operationerna har följande guards (här används ==) och actions (här används :=):

$O_1^{\uparrow}$: $a == 0$

$O_1^{\downarrow}$: $a := 1$

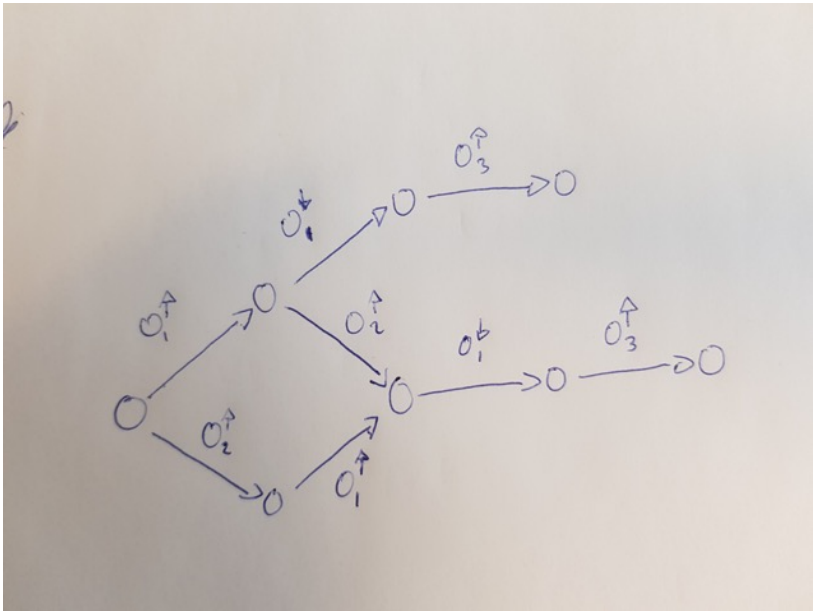
$O_2^{\uparrow}$: $a == 0$ / $b := 1$

$O_3^{\uparrow}$: $a == 1$ / $a := 2$

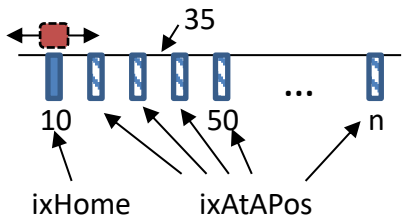
Initialtillstånd på variablerna är 0.

(5 poäng)

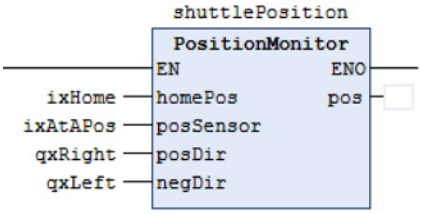
Lösning 1



Uppgift 2



I denna uppgift skall du implementera funktionsblocket PositionMonitor som håller ordning på positionen på en liten skyttel. Blocket skall användas vid olika installationer och skall fungera för olika längder av banor. Längst banan finns många givare. Givaren längst till vänster sätter den digitala insignalen ***ixHome*** till sann när skytteln är ovanför den, vilket definierar hemmaläget. När skytteln befinner sig där, skall dess position vara 10. När den åker åt höger från hemmaläget, och lämnar givaren, skall skyttelns position öka till 15. När den kommer fram till nästa givare blir positionen 20, därefter 25 när den befinner sig mellan andra och tredje givaren, och så vidare tills den passerar den sista givaren. Skytteln har alltså alltid en position som är 5, 10, 15, 20, 25, … Alla givare förutom hemmaläget är inkopplade till samma digitala insignal, ***ixAtAPos***.



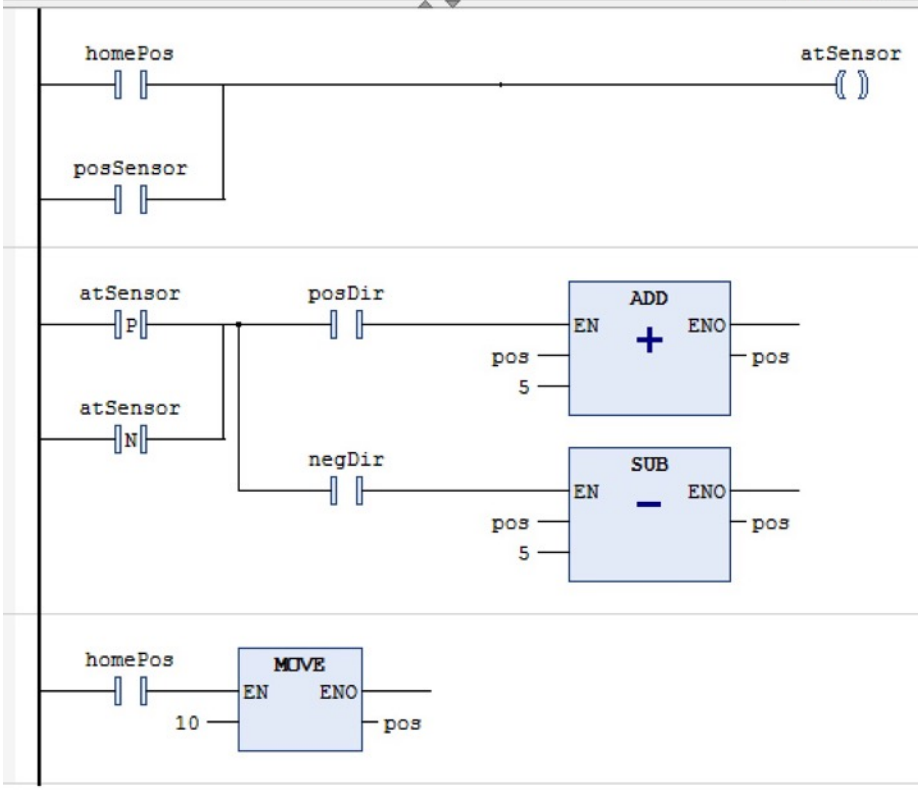
Implementera ladderkoden inne i funktionsblocket PositionMonitor, som är instansierad till höger, så att den beräknar positionen på skytteln. Observera att insignalerna ***ixHome*** och ***ixAtAPos*** är inkopplade till de interna boolska variablerna homePos och posSensor, vilka du använder i din kod. Styrsignalerna ***qxRight*** och ***qxLeft***, som du inte behöver styra i denna kod, är inkopplade på de boolska variablerna ***posDir*** och ***negDir***, vilka du använder för att veta åt vilket håll skytteln rör sig. Utsignalen ***pos***, är en Integer och är den variabel som du skall sätta i din kod

Skytteln börjar alltid vid hemmaläget och din kod skall hålla ordning på positionen oavsett hur många sensorer som är inkopplade. Din kod skall implementeras med hjälp av ladderlogik.

(6 poäng)

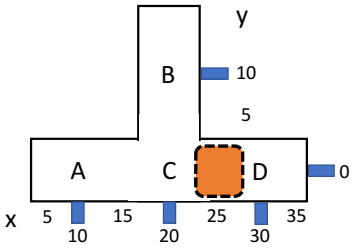
Möjlig lösning uppgift 2

```
FUNCTION_BLOCK PositionMonitor_Solution
VAR_INPUT
    homePos: BOOL;
    posSensor: BOOL;
    posDir: BOOL;
    negDir: BOOL;
END_VAR
VAR_OUTPUT
    pos: INT;
END_VAR
VAR
    atSensor: BOOL;
END_VAR
```



Uppgift 3

Nu skall du implementera ett system som kör runt små autonoma transportörer (AGVer) i en korridor. I bilden till höger finns en schematisk skiss över systemet. AGVn med streckad kant kan åka runt mellan de fyra positionerna A, B, C, och D, och även de mellanliggande positionerna. I X-led kan den befinna sig i position 5, 10, 15, 20, 25, 30 och 35, där A=10, C=20 och D=30. I Y-led kan positionerna vara 0, 5, 10 och 15, där B=10. Koden som uppdaterar positionerna är redan implementerad och du kommer åt positionen på AGVn i variablerna ***agv1_posX*** och ***agv1_posY.*** För att styra AGVn används utgångarna ***qxAGV1Left***, ***qxAGV1Right***, ***qxAGV1Up***, ***qxAGV1Down***.



a) Du skall implementera funktionsblocket DirectionControl som kan styra AGVn till en viss given position. Till höger ser du hur in och utgångarna skall definieras. Ingången ***gotoPos*** säger till vilken position AGVn skall åka, ***currentPos*** säger var AGVn befinner sig just nu, ***limitMin*** och ***limitMax*** sätter en gräns för när den måste stanna (AGVn får inte åka in i väggarna).

```
FUNCTION_BLOCK DirectionControl
VAR_INPUT
    gotoPos: INT;
    currentPos: INT;
    limitMax: INT;
    limitMin: INT;
END_VAR
VAR_OUTPUT
    atPos: BOOL;
    posDirCtrl: BOOL;
    negDirCtrl: BOOL;
END_VAR
```

Utgångarna som blocket skall sätta är ***atPos***, som säger när gotoPos är samma som currentPos, ***posDirCtrl*** styr AGVn i den riktning som ökar värdet på currentPos (höger i X-led, upp i Y-led) och ***negDirCtrl*** styr i riktningen som minskar (vänster i X-led och ner i Y-led).

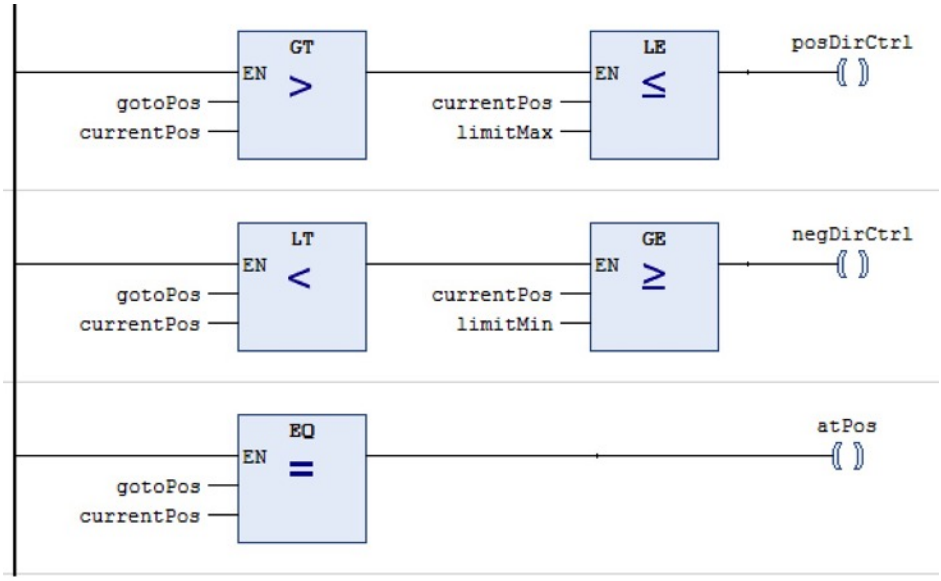
Implementera koden i funktionsblocket så att utgångarna ***posDirCtrl*** och ***negDirCtrl*** styr i rätt riktning givet vart blockets användare vill att den skall åka. Respektera limit-ingångarna så att AGVn inte kan passera övre och undre gräns. Implementera med hjälp av ladderlogik.

(5 poäng)

b) Skapa två instanser av blocket DirectionControl, med lämpliga namn, en för X-led och en för Y-led. Koppla in de in och utgångar som behövs för styrningen. Använd de två globala variablerna ***agv1Ctrl.gotoX*** och ***agv1Ctrl.gotoY*** för att veta till vilken position som AGVn skall åka. Utgången atPos behöver inte kopplas in.

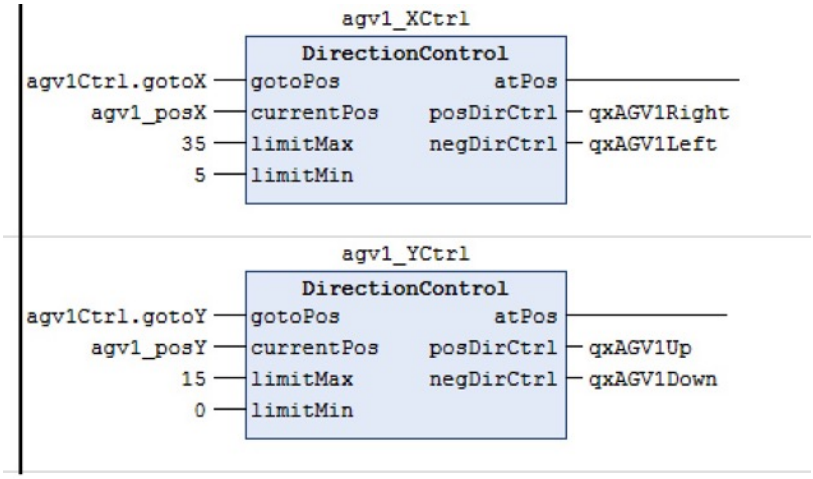
(2 poäng)

Möjlig lösning 4a



Möjlig lösning 4b

Här går också att hantera så att AGVn inte krockar i X och Y genom att dynamiskt ändra på limitMin och limitMax beroende på positionen. Tex så kan inte y vara annat än 0 om x inte är 20.



c) Nu skall du implementera operationen AGVControl som ett funktionsblock. Det styr AGVn mellan de fyra positionerna A – D. Blocket har 7 ingångar, där ***gotoA*** – ***gotoD*** är styrsignaler för vart AGVn skall åka och ***currentX*** och ***currentY*** ger den nuvarande positionen på AGVn. ***stopAGV*** säger till AGVn att stanna sin rörelse där den är så länge insignalen är sann. De tre översta utgångarna skall beskriva operationens tillstånd där ***init*** betyder att AGVn är redo att köra till en position, ***executing*** säger att AGV håller på att åka till en position och ***finished*** säger att den är framme. Den kan endast vara i ett tillstånd åt gången. ***gotoX*** och ***gotoY*** används för att styra AGVn och är inkopplade till dina block från uppgift b.

```
FUNCTION_BLOCK AGVControl
VAR_INPUT
    gotoA: BOOL;
    gotoB: BOOL;
    gotoC: BOOL;
    gotoD: BOOL;
    stopAGV: BOOL;
    currentX: INT;
    currentY: INT;
END_VAR
VAR_OUTPUT
    init: BOOL;
    executing: BOOL;
    finished: BOOL;
    gotoX: INT;
    gotoY: INT;
END_VAR
VAR
END_VAR
```

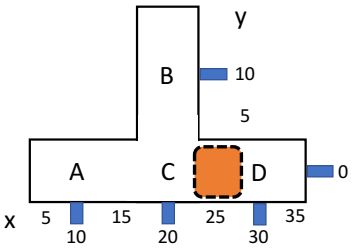
Användaren av ditt block kommer att sätta någon av goto-signalerna till sann. Goto-signalen som startade operationen kommer alltid vara sann till operationen är färdig.

Kom ihåg att AGVn inte får krocka med väggarna vilket innebär att du behöver ha koll på dina goto-signaler. Detta gäller också när din kod initieras.

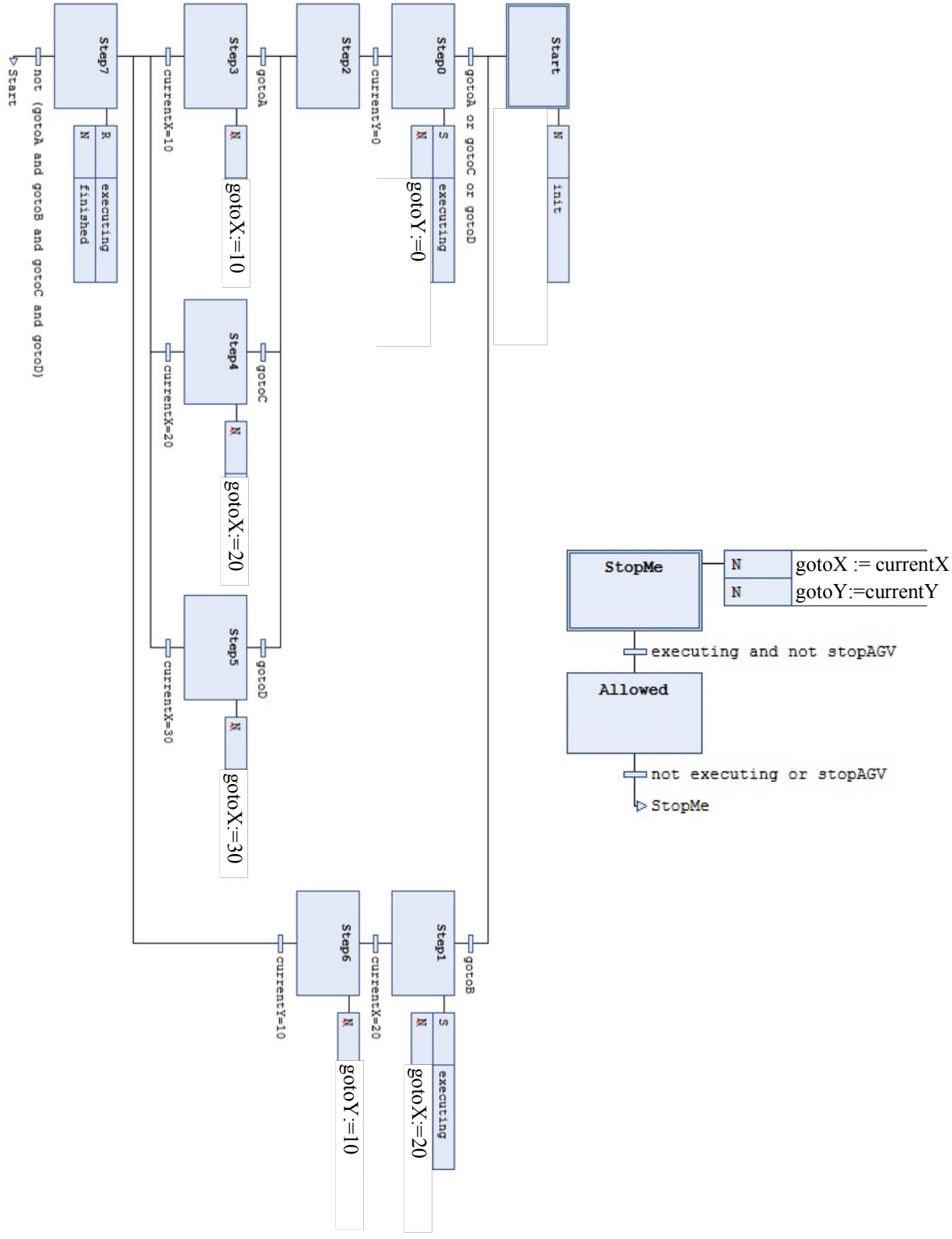
När ***stopAGV*** blir sann, stoppas AGVn direkt och väntar på att åka vidare tills den blir falsk.

Implementera koden med hjälp av en eller flera SFCer. För att kommunicera mellan flera SFCer, deklarera egna variabler i början av uppgiften. Om ordning är viktigt mellan dina SFCer, skriv i vilken ordning de exekveras i början av uppgiften.

(8 poäng)



Möjlig lösning 4c. Observera att man måste sätta gotoX := currentX, för att den inte skall röra sig vid initiering och när man stoppar. Detta sköts med den lilla SFCn som exekveras efter den stora SFCn, vilket gör att goto-signalerna skrivs över när den skall stå still.



d) Nu har vi alla komponenter vi behöver för att styra AGVer i vårt lilla system. Det finns dock ett problem; om vi har flera AGVer så kan de krocka. Du skall därför uppdatera kod nedan så att de två AGVerna agv1 och agv2 aldrig kan krocka.

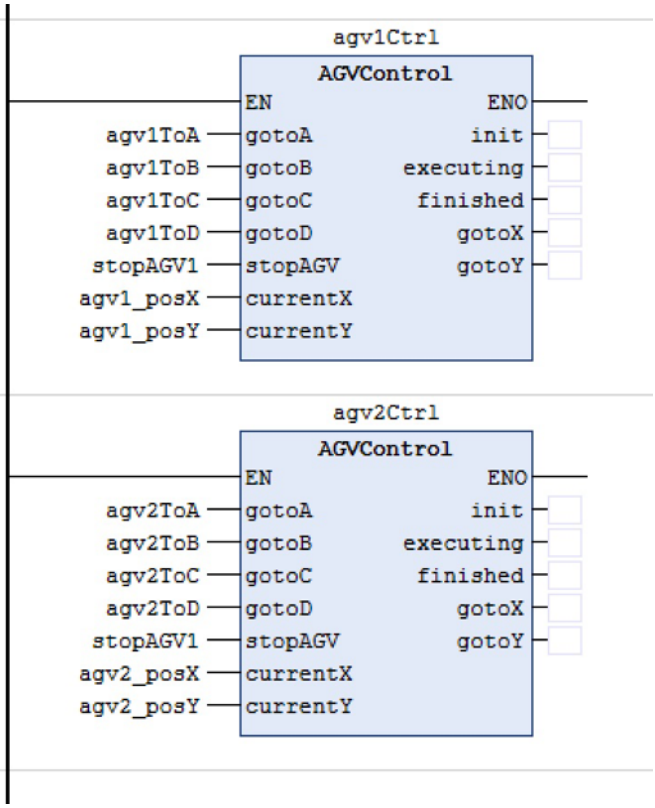
En annan kod sätter variablerna ***agv1ToA*** – ***agv1ToD*** och ***agv2ToA*** – ***agv2ToD***, så dessa kan du inte skriva över. ***stopAGV1*** och ***stopAGV2*** är det dock ingen annan som använder, så dessa kan du skriva till. Du behöver lägga till egna variabler och eventuellt även ändra vilka variabler som är inkopplade till blocken.

Utsignalerna gotoX och gotoY är redan inkopplade till styrningen och inget du behöver implementera här.

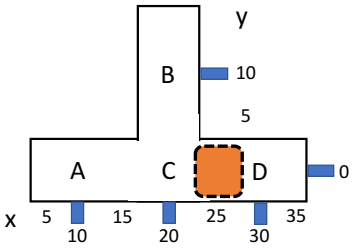
Bara en av AGVerna kommer att åka åt gången. Det är upp till dig att bestämma om en AGV skall stå och vänta på sin startplats tills hela vägen till målet är fritt från den andra AGVn, eller om den skall åka fram och vänta bredvid den andra AGVn

Du behöver inte hantera situationen om AGVerna blockerar varandra så att ingen av dem kan åka till sitt mål. Din uppgift är endast att hindra så att de inte krockar.

Skapa gärna egna funktionsblock för att slippa repetera samma kod. Du får välja om du implementerar i ladder eller SFC (eller en kombination). Instanserna av dina funktionsblock kan du deklarera i ladder.



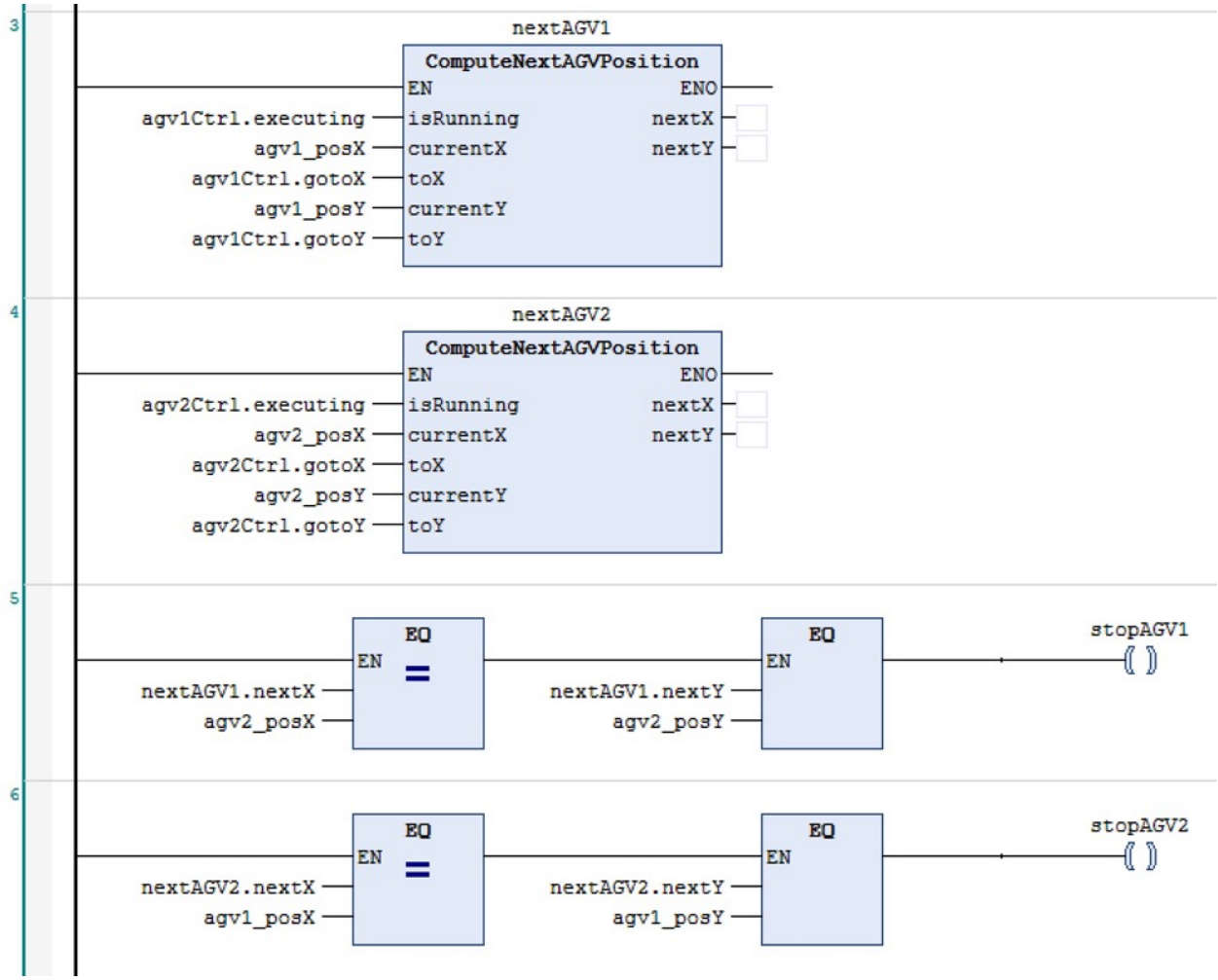
(7 poäng)



Möjlig lösning 4d

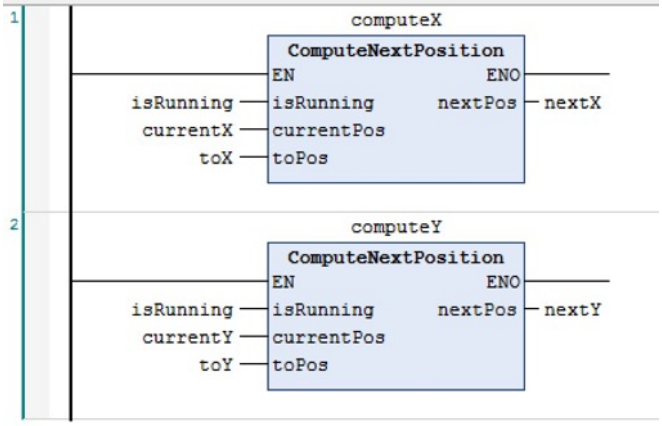
Det finns två sätt att undvika kollision. Antingen kollar man om den andra AGVn befinner sig mellan nuvarande position och målpositionen i både X och Y led. Eller, som följande lösning gör, kollar när AGVn åker om den andra AGVn befinner sig i nästa position som AGV kommer till. Jag har använt några funktionsblock. Blocken agv1Ctrl och agv2Ctrl är oförändrade från uppgiften.

ComputeNextAGVPosition räknar hela tiden ut nästa position i både X och Y led genom att lägga på eller dra ifrån 5 från nuvarande position. Om nästa position är samma som den andra AGVn, stoppas rörelsen, tills den andra AGVn har flyttat på sig. Om AGVn inte rör sig är nästa position samma som dess nuvarande position.

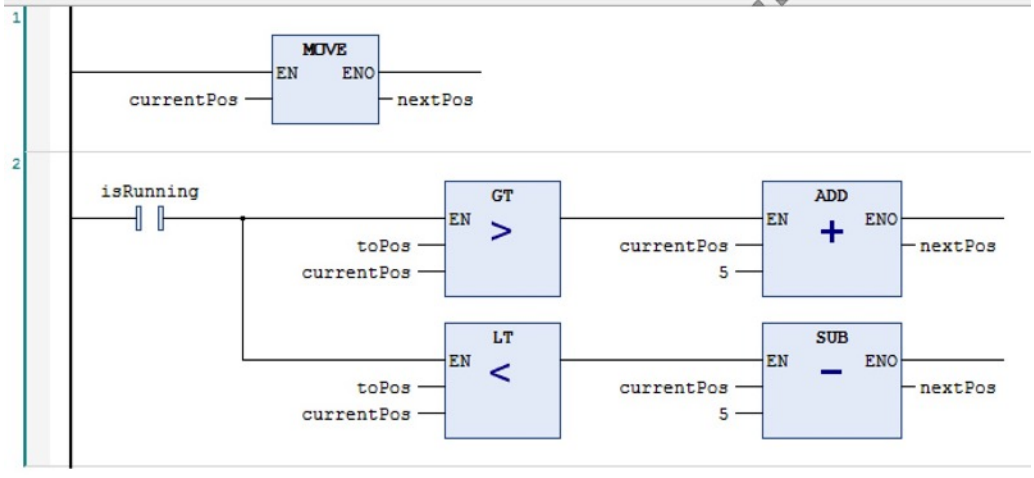


ComputeNextAGVPosition beräknar nästa X och nästa Y position. I computeNextPosition beräknas nästa position beroende på var AGVn är påväg.

```
FUNCTION_BLOCK ComputeNextAGVPosition
VAR_INPUT
    isRunning: BOOL;
    currentX: INT;
    toX: INT;
    currentY: INT;
    toY: INT;
END_VAR
VAR_OUTPUT
    nextX: INT;
    nextY: INT;
END_VAR
VAR
    computeX: ComputeNextPosition;
    computeY: ComputeNextPosition;
END_VAR
```

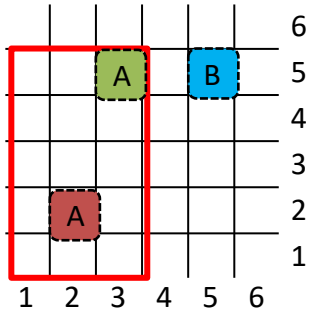



```
FUNCTION_BLOCK ComputeNextPosition
VAR_INPUT
    isRunning: BOOL;
    currentPos: INT;
    toPos: INT;
END_VAR
VAR_OUTPUT
    nextPos: INT;
END_VAR
VAR
    prevNext: INT;
END_VAR
```



Uppgift 4

I denna uppgift skall du modellera operationer för två små AGVer som åker runt i ett rutnät. Din uppgift är att se till att de inte kan krocka.



För att underlätta lite så begränsas de två AGVerna av att de inte får passera varandra i både X och Y-led samtidigt. Så om A åker från Pos(2, 2) till Pos(3,5) får B endast röra sig mellan 4-6 i X-led men till alla positioner i Y-led (om den är i Y 6 så får den röra sig till alla X-positioner). Det är illustrerat i bilden till höger.

Det finns redan två operationer som heter ***gotoPosA*** och ***gotoPosB*** för AGV A och B. Din uppgift är att definiera pre och postconditions för dessa två operationer så att de kan åka runt utan att krocka och aldrig passera varandra enligt ovan regel.

Jag har definierat en ny typ som ni kan använda:

Pos(x: Int, y: Int) - Vilket definierar koorinater i rutnätet. (Du kan använda respektive x eller y koordinat med punktnotation i dina conditions eller funktioner, p(5, 5): Pos; p.x eller p.y)

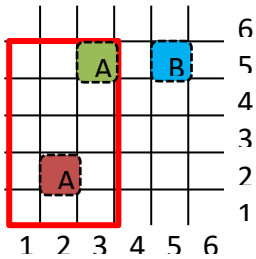
De som använder operationerna sätter insignalerna ***toPosA: Pos*** eller ***toPosB: Pos*** för att beordra vart AGVn skall åka. Den skall åka när det är möjligt att åka till den givna positionen utan att krocka med den andra AGVn eller bryta mot ovan regel. Du styr AGVerna med ***runA: Bool*** och ***runB: Bool*** och när de är framme sätter AGVerna själva ***finishedA: Bool*** och ***finishedB: Bool*** till true. För att kunna köra igen, sätter du run-signalerna till false. Observera att AGVerna kan röra på sig samtidigt, men skall ändå inte krocka.

Du kan använda funktionerna MAX(a: Int, b: Int): Int eller MIN(a: Int, b: Int): Int i dina conditions för att ta reda på vilken int som är störst eller minst. Om du vill kan du skriva egna funktioner i java, pseudokod, eller PLC-kod som du kan använda i dina conditions för att beräkna om något är sant eller falsks eller om du skall uppdatera något i en action.

Så skapa egna variabler och skriv upp de två operationerna pre- och postconditions så att de två operationerna fungerar. **Initialposition är x:2, y:2 för A och x:5, y:5 för B**.

(7 poäng)

Möjlig lösning uppgift 4



För att de inte skall kunna krocka så måste AGVerna boka hela det område som den kommer röra sig inom. Alltså hela den röda rutan i bilden Det är ju ganska begränsande, men då vi inte har en nuvarande position kan vi inte veta var de är inom området. Då de inte kan passera varandra så räcker det med en position för att definiera hela området, då andra hörnet är i kanterna. Beroende på var AGVn startar och vart den skall åka skall alltid den störsa värdet av antingen start eller slutkoordinaten i x respektive y plockas.

För att veta när operationen kan starta räcker det att kolla så att positionen vi skall åka till inte finns inom det andra AGVns bokade område, då nuvarande position alltid finns utanför.

Variabler: *allA: Pos, atA: Pos, allB: Pos, atB: Pos // allX definerar det bokade området, atX definirar start*

$gotoA^{\uparrow}$ $guard: atA \neq toPosA \wedge (MAX(atA.x, toPosA.x) < allB.x \vee MAX(atA.y, toPosA.y) < allB.y)$
// det ger inga minuspoäng med följande enklare guard (även fast den skulle kunna passera den andres // område) $atA \neq toPosA \wedge (toPosA < allB.x \vee toPosA < allB.y)$
$action: runA \coloneqq true$
$allA.x \coloneqq MAX(atA.x, toPosA.x)$
$allA.y \coloneqq MAX(atA.y, toPosA.y)$

$gotoA^{\downarrow}$ $guard: finishedA$
$action: runA \coloneqq false$
$atPosA \coloneqq toPosA$
$allA \coloneqq toPosA$

$gotoB^{\uparrow}$ $guard: atB \neq toPosB \wedge (MIN(atB.x, toPosB.x) > allA.x \vee MIN(atB.y, toPosB.y) > allA.y)$
$action: runB \coloneqq true$
$allB.x \coloneqq MIN(atB.x, toPosB.x)$
$allB.y \coloneqq MIN(atB.y, toPosB.y)$

$gotoB^{\downarrow}$ $guard: finishedB$
$action: runB \coloneqq false$
$atPosB \coloneqq toPosB$
$allB \coloneqq toPosB$