
KAPITTEL 1

Logikkstyring

1.1 Innledning

Logikkstyring er den del av et regulerings- eller styringssystem som tar seg av logiske funksjoner, som for eksempel Av/På, sekvensiell oppstart og nedkjøring, betingelses-kontroll på videre progresjon i en trinnvis opp-starting (forrigling), osv.

Det er påstått fra industri-hold at rundt 80 % av et regulerings- og styringssystem er logikkstyring, mens bare 20 % er kontinuerlig eller analog regulering. Det finnes knapt noe styringssystem som *ikke* inneholder logikkstyring, mens det er svært mange systemer som bare inneholder det, dvs. de inneholder *ikke* kontinuerlig regulering.

Når vi i dag i utstrakt grad realiserer styrings- og reguleringssystemer ved hjelp av datamaskiner eller datateknisk baserte styringsenheter, ligger dette spesielt naturlig til rette for logikkstyring. For en digital datamaskin er logikkstyring mye mer naturlig enn kontinuerlig regulering. Kontinuerlig regulering må behandle systemet som samplet for å kunne realisere reguleringen på en datamaskin, mens logikkstyring benytter logiske funksjoner, som er direkte naturlig for datamaskinen.

Logiske funksjoner kan inndeles i to grupper:

- Kombinatoriske
- Sekvensielle

I det etterfølgende vil vi først se på hva som er særtrekkene for disse to grupper. Deretter vil vi gjennom fire kapitler gi en konsentrert oversikt over grunnlaget for kombinatoriske funksjoner, svitsj-teori og boolsk algebra samt kretser for realisering av kombinatoriske funksjoner og sekvenskretser. I resten av kompendiet blir kombinatoriske og sekvensielle funksjoner for en stor del behandlet sammen. Man kan nemlig vanskelig tenke seg den ene gruppen uten at den er sterkt knyttet til den andre, og kombinatoriske og sekvensielle funksjoner er gjerne innvevet i hverandre.

Kombinatoriske funksjoner:

Når vi skal studere kombinatoriske funksjoner, er det enklest å ta utgangspunkt i elektriske brytere eller *svitsjer*.

Svitsjer og svitsjing opptrer i mange fysiske former: elektriske brytere og vendere, releer, signal eller ikke signal i en elektrisk krets, AV og PÅ, osv. Det dreier seg altså om noe som har én av to tilstander, og datateknikken er som kjent basert på kretser som opptrer på denne måten. I styringssystemer manifesterer dette “noe” seg som *variable* som kan ha én av to tilstander. Slike variable er altså *binære variable*, og i styringssystemer direkte realisert med elektronikk er de gjerne representert av et elektrisk signal som kan anta én av to distinkte spenningsnivåer.

I kombinatoriske funksjoner genereres en ut-variabel entydig av funksjonens øyeblikkelige inn-variable (fri variable). De fri variable er enten binære (boolske¹) variable eller relasjoner med binært utkomme (verdi), som t.eks. $x > y$. Generelt er dette boolske *uttrykk* (eng. *expression*). Den genererte utvariabelen (avhengige variable) er boolsk, dvs. med verdi SANN eller FALSK, 1 eller 0, osv.

Sekvensielle funksjoner:

Sekvensielle funksjoner bygger på kombinatoriske, men innfører noe om det som har foregått før, dvs. *historien*; det huskes hvilken *tilstand* systemet har vært i. Vi vil se mer grundig på dette i Kap. 1.6.

1. Hvorfor det kalles *boolsk* blir forklart i kapittel 1.2.

1.2 Grunnleggende om kombinatoriske funksjoner

1.2.1 Svitsjer og svitsj-teori

For å uttrykke relasjoner mellom størrelser som hver kan anta en av to verdier, må vi ha et sett av binære variable, et sett av svitsjefunksjoner av de binære variablene, og en algebra til behandling av disse funksjonene av binære variable. Dette er nært knyttet til **logikk**. I logikk har vi *påstander*, og en påstand er enten sann eller falsk og kan representeres ved en binær variabel. I logikken har vi en grunnleggende unær operasjon negasjon, to binære operasjoner disjunksjon (logisk addisjon) og konjunksjon (logisk multiplikasjon) og en algebra som kan formuleres helt basert på disse elementæroperasjonene.

I svitsj-teori har vi liknende fenomener: Vi har en unær operasjon komplementering svarende til negasjon, to binære operasjoner addisjon og multiplikasjon, gjerne kalt henholdsvis ELLER og OG og svarende til disjunksjon og konjunksjon, og vi har en teori som binder disse sammen, *boolsk algebra*, oppkalt etter George Boole som først formulerte denne i et banebrytende dokument publisert i 1854. Overensstemmelsen med logikk er grunnen til at vi ofte omtaler svitsjteori som svitsjlogikk, og elektroniske kretser som realiserer svitsjer som *logiske kretser*. Dette innebærer at svitsjkretser kan betraktes som realiseringer av logiske relasjoner, dvs. som anvendelse av logikk.

Fordi svært komplekse logiske kretser kan realiseres, og fordi dette inngår i en lang rekke forskjellige systemer i styrings- og reguleringsteknikk, telefoni, datateknikk osv., har svitsjteorien blitt utviklet til en uavhengig og vel etablert disiplin. Det er også verdt å merke seg at svitsjteorien også er nært beslektet med mengdelære. Operasjonene ELLER og OG svarer til *union* og *snitt* i mengdelæren.

1.2.2 Kort om Boolsk algebra

En **binær variabel** x er en variabel som kan anta én av to mulige verdier. Hva verdiene er, er egentlig av underordnet interesse når bare operasjonene på verdiene er definert. Således kan settet av de to verdiene eksempelvis være $\{\text{NEI, JA}\}$, $\{\text{FALSK, SANN}\}$, $\{\text{FALSE, TRUE}\}$, $\{0, 1\}$, $\{\text{AV, PÅ}\}$,

{OFF, ON}, {LAV, HØY}, eller det generiske {a, b}. I det følgende vil vi bruke settet (mengden) $V = \{0, 1\}$ hvis det ikke uttrykkelig er sagt noe annet.

For de to elementene i V definerer vi først to binære¹ operasjoner

1. Addisjon eller ELLER (OR), med symbol $+$ eller \vee :

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 1$$

I tabellform:

	0	1
0	0	1
1	1	1

To elementer behandlet med en addisjonsoperator blir kalt en *boolsk sum*, eller ganske enkelt en *sum*, av elementene.

En elektronisk krets som utfører denne operasjonen kalles en *ELLER-krets* (eng. *OR-circuit*).

2. Multiplikasjon eller OG (AND), med symbol \bullet eller \wedge :

$$0 \bullet 0 = 0$$

$$0 \bullet 1 = 0$$

$$1 \bullet 0 = 0$$

$$1 \bullet 1 = 1$$

I tabellform:

	0	1
0	0	0
1	0	1

To elementer behandlet med en multiplikasjonsoperator blir kalt et *boolsk produkt*, eller ganske enkelt et *produkt*, av elementene.

En elektronisk krets som utfører denne operasjonen kalles en *OG-krets* (eng. *AND-circuit*).

1. *binær* i denne sammenheng betyr at operasjonen virker på *to objekter*

Den tredje viktige operasjonen er *unær*, dvs. den opererer på bare én variabel:

3. *Komplementering*, ofte også kalt *negasjon*:

$$\bar{0} = 1$$

$$\bar{1} = 0$$

En elektronisk krets som utfører denne operasjonen kalles en *inverter*.

1.2.3 Svitsj-funksjoner

I likhet med vanlig algebra, har vi også i boolsk algebra funksjoner av en eller flere variable. I boolsk algebra må alle variable som inngår i en funksjon være binære variable. I tråd med vår anvendelse kaller vi disse funksjonene svitsj-funksjoner.

En svitsjfunksjon kan formelt uttrykkes:

$$f = \text{func}(x_1, x_2, x_3, \dots)$$

hvor *func* er en nærmere definert funksjon, *f* er boolsk (dvs. ender opp som en verdi 0 eller 1), og alle argumentene *x* er boolske variabler, dvs. enten fri variable eller boolske funksjoner. Eksempel på en enkel funksjon:

$$f = a \cdot c + b \cdot d$$

Hvis f.eks. $c = 1$ og $d = 0$, vil dette gi $f = a$.

Vi vil gi mange flere eksempler etter hvert, men først skal vi se litt på praktiske muligheter for realisering, og deretter på egenskaper og “regneregler” ved boolsk algebra.

Realisering skjer enten i programmer som kjører på datamaskin, eller direkte i fysisk utstyr, “hardware”, som ofte er elektroniske kretser. Siden elektroniske kretser kan realisere boolske funksjoner på en meget direkte måte, egner disse seg for å *beskrive* og *forklare* svitsjefunksjoner, selv om man til syvende og sist kanskje velger endelig realisering i datamaskinprogram.

1.3 Logikk-kretser

1.3.1 Krets-symboler

Kretselementer som realiserer elementære svitsjoperasjoner symboliseres:

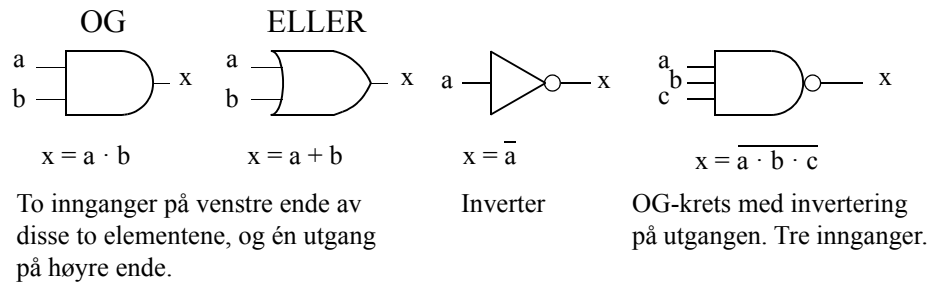


Fig. 1 Basis kretselementer

Basis-kretsene er OG, ELLER og inverter, og OG- og ELLER-kretsene har to eller flere innganger. Alle kretser kan kombineres med invertering, som symboliseres ved en ring. Invertert utgang er vist på kretselementet til høyre, men både innganger og utganger kan kombineres med invertering, og vi skal se at dette gir mulighet for svært fleksible løsninger.

Faktisk brukes oftere OG og ELLER-kretser *med* invertert utgang enn *uten*. På engelsk kalles disse henholdsvis NAND (Negated AND) og NOR (Negated OR). På norsk har vi ingen tilsvarende innarbeidete uttrykk¹. De to vanligste grunnelementene er altså:

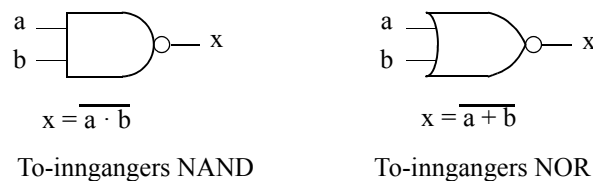


Fig. 2 Basis-elementene NAND og NOR

Som vi skal se senere, gir disse større fleksibilitet til kombinasjoner, og de er også litt enklere å realisere, dvs. de kan realiseres med litt færre elementære komponenter på den integrerte brikken.

1. Man kunne tenke seg "NOG" og "NELLER" eller "OG-IKK" og "ELLER-IKK", men slike uttrykk har ikke slått an.

1.3.2 Elektroniske kretselementer

Elektroniske kretselementer som realiserer svitsjfunksjoner er i dag vanligvis integrerte kretselementer, eller de inngår i integrerte kretser. De logiske variable er oftest elektriske likespenningssignaler med to distinkte spenningsnivåer til å representere **0** og **1**. I en vanlig tradisjonell klasse slike elementer er gjerne boolsk **0** representert ved $u_L = 0$ volt (egentlig $0\text{ V} < u_L < 0,4\text{ V}$) og boolsk **1** representert ved $u_H = +5$ volt (egentlig $2,4\text{ V} < u_H < V_{DD}$, der V_{DD} er forsyningsspenningen, nominelt 5 volt).

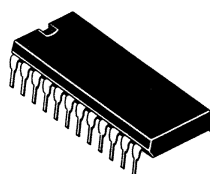


Fig. 3 Integrert krets, brikke (“chip”)

Vi kan også ha representasjon motsatt av det som ble forklart foran, dvs. boolsk 0 representert ved nominelt +5 volt og boolsk 1 representert ved 0 volt. I sistnevnte tilfelle snakker vi om “Lav” representasjon, og i det første, om “Høy” representasjon. Det er oftest “Høy” representasjon som brukes når kretser og logikkfunksjoner beskrives, og hvis intet annet sies, mener vi “høy” representasjon. Siden et signal (her boolsk) vanligvis varierer med tiden, dvs. det er en tidsfunksjon, avbilder vi et slikt tidsforløp gjerne slik:

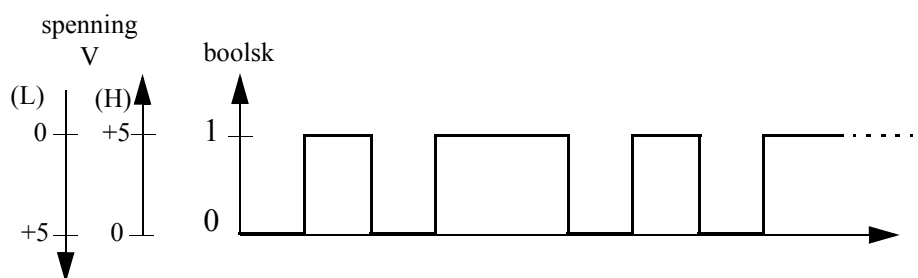


Fig. 4 Tidsdiagram for binær variabel

I stedet for Høy og Lav representasjon finner man i litteraturen også “positiv logikk” og “negativ -”, men dette er egnet til misforståelse, for i begge tilfeller er det vanligvis positiv spenning eller 0. I tilfellet “negativ logikk” er det spennings-svinget som går i negativ retning når signalet går fra 0 til 1 boolsk.

1.3.3 Intern realisering av elektroniske svitsj-kretser

Dette avsnittet er egentlig overflødig, for intet i det vi vil behandle i de etterfølgende kapitler avhenger av hva som her skal beskrives. Ikke desto mindre, så kan det hjelpe på forståelsen å vite hvordan de integrerte kretselementene virker internt, derfor skal vi ta en kort gjennomgang.

Moderne svitsjkretser er oftest utført i teknologien CMOS (Complementary Metal-Oxide Semiconductor), og grunnkretsen er inverteren:

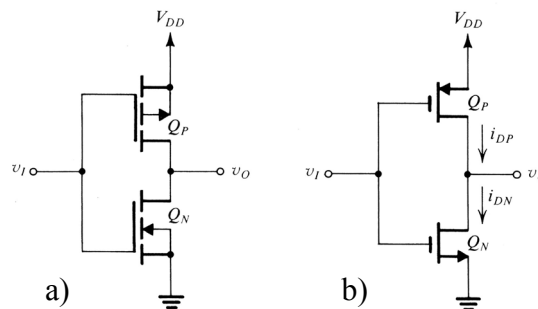


Fig. 5 Inverter realisert i CMOS: a) Detaljer b) Forenklet kretsskjema

Virkemåte for inverteren:

Inverteren består av to CMOS-transistorer, én P-kanal (Q_P) og én N-kanal (Q_N). Figur 6 viser inverteren med et ekvivalentskjema. Med “høy” spenning v_I på gate-inngangen blir Q_N ledende (har liten motstand source-drain), svarende til bryteren S_N lukket. Øverste transistor Q_P blir som en åpen bryter (høy motstand), dvs. S_P åpen. Dermed blir utgangen v_O lav. Motsatt med “lav” spenning på inngangen: Da blir Q_N (ekvivalent S_N) åpen og Q_P ledende (S_P lukket), og v_O blir høy.

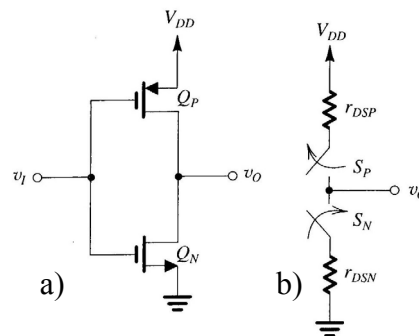


Fig. 6 CMOS-inverteren (a) og dens ekvivalent-skjema (b)

La oss nå utvide koplingen i Figur 6 med å dublere transistoren Q_N med en maken i parallell. Vi doublerer også transistoren Q_P med en maken, men her plassert i serie:

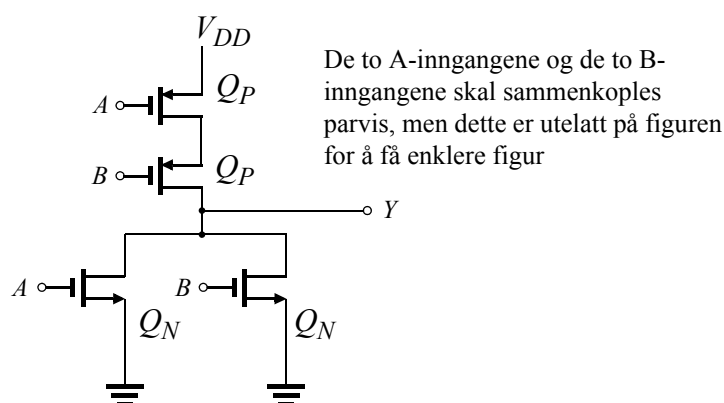


Fig. 7 To-inngangs NOR-port

På samme måte som i Figur 6 vil høy inngang gjøre Q_N -transistoren ledende, og dermed vil punktet Y trekkes *ned* hvis *enten* A - eller B -inngangen blir høy. Tilsvarende må *begge* Q_P -transistorene, som står i seriekopling, bringes ledende for at utgangen Y skal trekkes *opp*, dvs. at inngangene på begge Q_P -transistorene må være lav for at dette skal skje. Og motsatt, for at Y ikke skal trekkes opp, er det tilstrekkelig at én av Q_P -inngangene er høy. Når vi så danner par av én Q_P - og én Q_N -transistor ved å sammenkople deres innganger, vil det for at Y skal trekkes lav være tilstrekkelig at enten A - eller B -inngangene er høy. Vi har altså fått en krets som gir den boolske funksjonen

$$Y = \overline{A + B}$$

altså en NOR-krets, dvs. ELLER med invertert utgang.

Hvis vi trenger flere innganger, legger vi bare til flere Q_N - Q_P transistorpar, en Q_N i parallell med de andre Q_N , og en Q_P innskutt i serie med de opprinnelige Q_P . Jeg sier “vi”, men det er riktignok fabrikanten av integrertkretsen som gjør dette, vi “brukere” kan bare velge mellom forskjellige kretser med gitte antall innganger.

Forresten kaller vi ofte disse kretsene *porter*, idet de virker som porter eller dører som åpnes og lukkes. “Gate” på engelsk.

En NAND-krets kan tilsvarende realiseres ved å bytte om parallell- og seriekoplingene, altså seriekople Q_N -transistorene og parallellkople Q_P -transistorene:

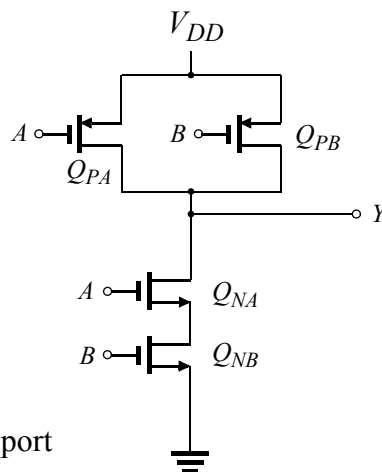


Fig. 8 To-inngangs NAND-port

Nå når vi har sett litt på hvordan elementære svitsjkretser eller porter virker og er oppbygget, er vi godt rustet til å bygge disse videre ut til å løse mer sammensatte boolske funksjoner. Da må vi først se litt mer på boolsk algebra, som vi så vidt introduserte i Kap. 1.2 side 3.

1.4 Regneregler innen boolsk algebra

1.4.1 Postulater for boolsk algebra

La oss ta for oss en mengde variable $B = \{a, b, c, \dots\}$ med ekvivalens-relasjonen $=$ og de to binære operasjonene $+$ og \cdot , samt den unære komplement-operasjonen.

Boolsk algebra baserer seg på et sett postulater, og på grunnlag av disse kan vi deretter utlede en del matematiske lover. Vi kan lett konstatere at reglene i postulatene samsvarer med talleksempelene med de to mulige verdiene 0 og 1 vist i innledningen.

P1: Lukket system: Resultatet av en boolsk operasjon er en verdi innen det lukkede sett av verdier $\{0, 1\}$:

$$\text{For alle } a \in B, \quad a + 1 = 1, \quad a \cdot 0 = 0$$

P2: Assosiativ: Operasjonene $+$ og \cdot er assosiative:

$$(a + b) + c = a + (b + c) = a + b + c$$

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) = a \cdot b \cdot c$$

P3: Kommutativ: Operasjonene $+$ og \cdot er kommutative:

$$a + b = b + a$$

$$a \cdot b = b \cdot a$$

P4: Distributiv: De to operasjonene er distributive overfor hverandre:

$$a + b \cdot c = (a + b) \cdot (a + c)$$

$$a \cdot (b + c) = a \cdot b + a \cdot c$$

P5: Identitetselementer: Det finnes et *identitetselement* for operasjonen $+$, betegnet 0 og kalt *null*, og et annet for operasjonen \cdot , betegnet 1 og kalt *én* eller *enhet* (unity) innenfor mengden B slik at

$$a + 0 = a$$

$$a \cdot 1 = a$$

P6: Komplement: Hvert element i B har et *komplement* innenfor B slik at hvis \bar{a} er komplementet av a , vil:

$$a + \bar{a} = 1$$

$$a \cdot \bar{a} = 0$$

Bemerk at 0 og 1, som betegner de to identitetslementene i boolsk algebra, ikke må forveksles med tallene 0 og 1 i vanlig algebra. Samtidig gjør vi oppmerksom på at i boolsk algebra, likesom i vanlig algebra, utføres operasjonen \cdot før operasjonen $+$. Altså:

$$a + b \cdot c = a + (b \cdot c) \neq (a + b) \cdot c$$

Komplementoperasjonen innebærer altså, at hvis $x = 0$, så vil $\bar{x} = 1$, og hvis $x = 1$, vil $\bar{x} = 0$.

1.4.2 Utledele lover eller teoremer

Fra postulatene foran kan vi utvikle følgende lover:

L1: Likhet: For alle

$a, b, c \in B$, hvis $a + b = a + c$ og $a \cdot b = a \cdot c$,
da er $b = c$.

L2: Komplementær: For alle

$a, b \in B$, hvis $a + b = 1$ og $a \cdot b = 0$,
da er $a = \bar{b}$ og $b = \bar{a}$

L3: Identitetslementene 0 og 1 er komplementar av hverandre:

På grunnlag av P4 har vi, siden 0 og 1 er $\in B$,
 $1 + 0 = 1$ og $0 \cdot 1 = 0$. Dermed gir L2:
 $0 = \bar{1}$ og $1 = \bar{0}$.

L4: Identitetslementene 0 og 1 er unike:

Hvis dette ikke hadde vært tilfelle, ville det vært to eller flere 0'er, la oss anta de to 0_1 og 0_2 . Siden $0_1 \in B$ og 0_2 er et identitetslement, vil P4 gi $0_1 + 0_2 = 0_1$. Samme resonnering for 0_2 gir $0_2 + 0_1 = 0_2$, og dermed vil
 $0_1 = 0_1 + 0_2 = 0_2 + 0_1 = 0_2$.
På samme måten kan lett vises at
 $1_1 = 1_1 \cdot 1_2 = 1_2 \cdot 1_1 = 1_2$.

L5: Komplementet er unikt:

Hvis dette ikke hadde vært tilfelle, ville det vært (minst)

to komplementer av en variabel a , la oss kalle dem a_1 og a_2 . P5 ville da gi $a + a_1 = 1$ og $a + a_2 = 1$. Vi ville også ha $a \cdot a_1 = 0$ og $a \cdot a_2 = 0$. Altså vil $a + a_1 = a + a_2$ og $a \cdot a_1 = a \cdot a_2$. Dermed får vi, fra L2: $a_1 = a_2$. Bemerk: L5 kunne vært utviklet fra dette.

- L6: Involusjon: For alle $a \in B$ er $\overline{\overline{a}} = a$.
 Dette lar seg lett vise, f.eks. ut fra P5 og L2.
- L7: Idempotent: Et boolsk element er *idempotent* overfor de binære operasjonene $+$ og \cdot , dvs. en operasjon utført gjentatte ganger på samme variabel endrer ikke variabelens verdi.
 D.v.s. $x + x = x$, $x \cdot x = x$. Hvis $x = 0$, så vil $x + x = 0 + 0 = 0$, $x \cdot x = 0 \cdot 0 = 0$, og hvis $x = 1$, vil $x + x = 1 + 1 = 1$, $x \cdot x = 1 \cdot 1 = 1$.
- L8: Absorpsjon: For alle $a, b \in B$, $a + ab = a$ og $a \cdot (a + b) = a$.
- L9: DeMorgan's teorem: Dette er et viktig teorem for realisering av porter.
 $\overline{(a + b)} = \overline{a} \cdot \overline{b}$ og $\overline{(a \cdot b)} = \overline{a} + \overline{b}$.

Disse teoremene kan se "tørre og kjedelige" ut, men de danner et uunnværlig fundament for videreutvikling og sammenstilling av elementære logikkretser. De forteller oss blant annet at om vi utfører en rekke operasjoner etter hverandre på boolske variable, holder resultatet seg alltid innenfor det lukkede settet $\{0, 1\}$. Hvis vi komplementerer en variabel a flere ganger etter hverandre, veksler resultatet bare mellom a og \overline{a} . Altså $\overline{\overline{a}} = a$.

1.4.3 Konsekvenser av De Morgan's teorem

En svært nyttig lov er De Morgan's teorem. Under gjennomgangen foran om hvordan elementære kretselementer er realisert, ble det nevnt at porter med invertert utgang er vanligst. Dette skal vi nå utdype i sammenheng med De Morgan's teorem. NAND-kretsen realiserer $\overline{(a \cdot b)}$, og De Morgan's teorem sier oss at dette er det samme som $\overline{a} + \overline{b}$. Altså kan den samme elektronikk-kretsen som realiserer NAND også betraktes som en OR med in-

verterte innganger. Likeså vil NOR også være det samme som AND med inverterte innganger:

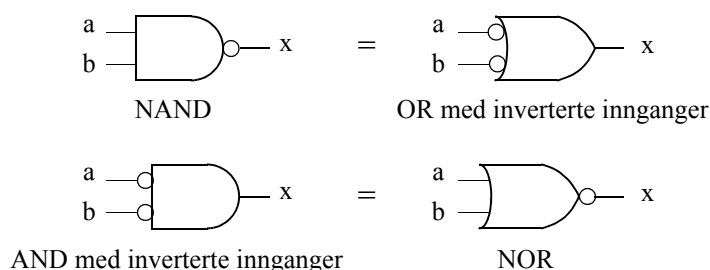


Fig. 9 To sider av samme sak v.hj.a. De Morgan

1.4.4 Høy og lav representasjon

På side 7 ble forklart hva vi mener med “høy representasjon” og “lav representasjon” og forskjellen mellom disse. Et “høyt” signal (nominelt 5 V) uttrykker boolsk 1 i høy representasjon. Men vi ser nå at høy signalverdi like gjerne kan oppfattes som 0 i lav representasjon. Vi kan gjerne blande disse representasjonene hvis vi markerer representasjonen: 1 (H) = 0 (L) og 0 (H) = 1 (L). Dette er det samme som når vi, med høy representasjon, skriver $1 = \bar{0}$, og $0 = \bar{1}$. Altså er $a(L) = \bar{a}(H)$ og $\bar{a}(L) = a(H)$.

Fordelen som oppnås med dette er at vi i mange tilfeller kan beholde en mer “naturlig” oppfatning av et signal slik det passerer gjennom en rekke trinn av kretser. Hvis et navngitt signal (la oss kalle det a) representerer en hendelse, av varighet mye kortere og sjeldnere enn “ikke hendelse”, vil $a = 1$ uttrykke at hendelsen inntreffer, representert ved en kortvarig positiv puls på en signalledning. Når dette signalet passerer gjennom en NAND-port, med konstant 1 (H) på portens andre inngang, kommer det på utgangen et signal som i hviletilstand ligger høyt, men når hendelsen inntreffer kommer det en puls som svinger negativt, kortvarig til 0 V. Det er da mer naturlig å se dette som bare et skifte av representasjon, ikke en negasjon. Det er fremdeles samme aktive hendelse, det er unaturlig å være tvunget til å betrakte høy signalverdi som aktiv eller “SANN” og et “ikke” eller FALSK som hendelsen.

Med denne betraktningsmåte får vi en notasjon som hjelper oss med å skille mellom logisk verdi og elektrisk representasjon. En logisk variabel har da en øyeblikksverdi a eller \bar{a} , uavhengig av om den på et visst sted i kretsskjemaet er i høy eller lav representasjon, (H) eller (L). I kretsskjemaet lar vi en ring symbolisere både boolsk invertering og skifte av elektrisk representasjon.

Eksempel: En kretskombinasjon vi ofte har bruk for er multiplekseren. Dette er en krets hvor vi ved hjelp av et boolsk styresignal velger ut ett av to signaler. Dette svarer til en elektrisk vender med to stillinger, hvor et boolsk styresignal bestemmer venderstillingen.

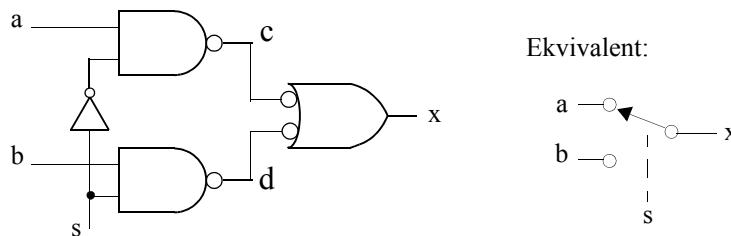


Fig. 10 To-stillings multiplekser

En morsom detalj er at denne koplingen, som benytter fire elementære kretselementer: to NAND, én OR med inverterte innganger og én inverter, kan realiseres med bare én integrert krets, “chip”, som inneholder 4 identiske elementer, i katalogen oppført som fire NAND. En av disse benyttes som inverter ved å sammenkople de to inngangene, og en av NAND-kretsene benyttes som utgangens OR med innganger i lav representasjon.

$$c(L) = a \cdot \bar{s}; \quad d(L) = b \cdot s; \quad x(H) = c(L) + d(L) = a \cdot \bar{s} + b \cdot s$$

Konsentrerer vi oss om logikken fremfor elektrisk representasjon, kan vi droppe (L) og (H) og får ganske enkelt

$$c = a \cdot \bar{s}; \quad d = b \cdot s; \quad x = c + d = a \cdot \bar{s} + b \cdot s$$

Det er imidlertid viktig å bemerke at det aller meste av litteraturen *ikke* drar nytte av denne notasjon men blander representasjon og logisk tilstand ved å konsekvent betrakte alle signalledninger i høy representasjon. Med denne vanlige uttrykksmåte blir ovenstående likninger:

$$c = \overline{a \cdot \bar{s}}, \quad d = \overline{b \cdot s}, \quad x = \bar{c} + \bar{d} = a \cdot \bar{s} + b \cdot s$$

Begge uttrykksmåter er naturligvis *korrekte*, men jeg vil påstå at den første av disse former viser mest direkte hva vi ønsker å uttrykke.

1.5 Vipper eller “Flip-flop”

I de foregående kapitlene har vi behandlet kombinatorisk logikk. I neste kapittel skal vi ta for oss sekvensielle funksjoner, som vi så vidt nevnte helt i begynnelsen, på side 2. For å kunne ha noe konkret å knytte behandlingen av sekvensielle funksjoner til, vil vi i dette kapitlet gi en kort gjennomgang av elektroniske *komponenter* som trengs for å kunne realisere *historie* eller *tilstander*, som **sekvensielle funksjoner** baserer seg på.

“Historie” eller “tilstander” innebærer en form for hukommelse, og for å oppnå det må vi ha noen kretser som kan skiftes mellom forskjellige stabile tilstander. En type grunnelementer for dette er bistabile vipper, ofte kalt “flip-flops”. Det finnes også monostabile og astabile vipper.

Bistabile vipper har to tilstander, SATT og RESATT (eng. SET og RESET), og de kan forbli i disse tilstander ubegrenset i tid.

Vippen er også den elementære byggeblokken for tellere, registre og andre kretser for å styre sekvensiell logikk.

1.5.1 SR (SETT-RESETT)-vippen

En aktiv-HØY input SR-vippe kan enkelt realiseres ved å krysskople to NOR-kretser. En aktiv-LAV input SR-vippe realiseres tilsvarende ved å bruke to NAND-kretser:

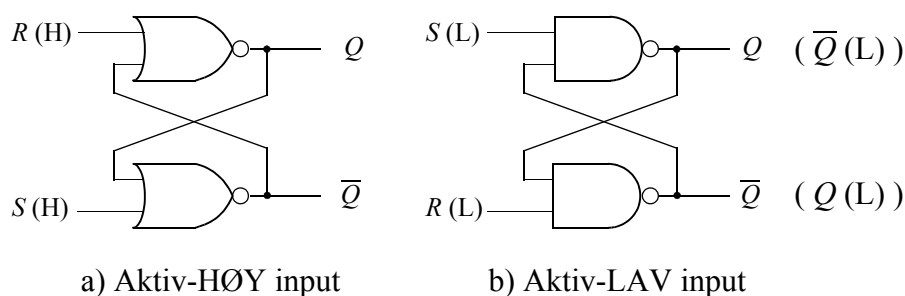


Fig. 11 SR-vippe, realisert med enkle portelementer

For Aktiv-HØY input-kretsen skal begge inngangene normalt (dvs. når de er passive) være LAV. Etter høy puls på S-inngangen vil utgangen $Q = 1$,

uansett hva den var før S-(SETT-)pulsen. Dette innebærer tilstand 1. Tilsvarende vil en HØY puls på R-inngangen tvinge utgangen Q til LAV, dvs. tilstand 0. Utgangen \bar{Q} vil alltid være komplementet av Q når begge inngangene er passive, dvs. lave.

Aktiv-LAV input -kretsen virker tilsvarende, men her er passiv verdi for inngangene HØY, og kretsen settes, henholdsvis resettes, av en LAV inngangspuls.

I overensstemmelse med figur 9 kan kretsene i figur 11 tegnes slik:

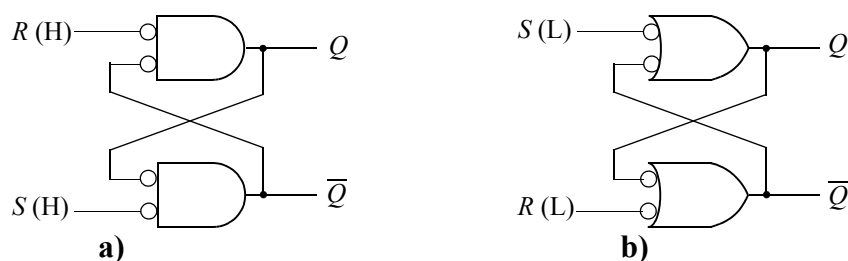


Fig. 12 Alternative tegnemåter for kretsene i figur 11

Vi kan noen ganger ha behov for å styre hvorvidt S eller R-signalet skal få påvirke vippen. Vippen kan da utstyres med OG-porter foran inngangene:

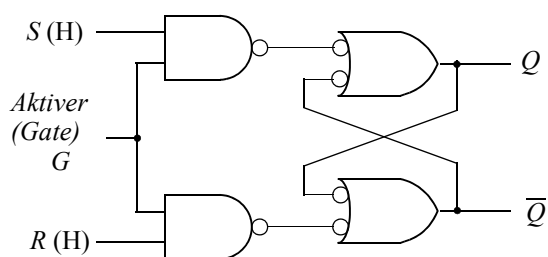


Fig. 13 SR-vippe med aktiveringsinngang

Signalet *Aktiver* må være 1 for at S eller R skal kunne ha noen virkning.

Tilstandstabell

Det kan være illustrerende med en tabell over tilstandene til innganger og utganger for SR-vippen. Vi tar for oss vippen med Aktiv-LAV innganger:

Innganger		Utganger		Forklaring
S	R	Q	\overline{Q}	
H	H	IE	IE	Ingen Endring. Vippen forblir i eksisterende tilstand
L	H	1 (H)	0 (L)	Vippen blir SATT
H	L	0	1	Vippen blir RESATT
L	L	1	1	Ikke tillatt tilstand for inngangene

Hva betyr “Ikke tillatt”?

Hvis vi analyserer virkemåten for kretsen i figur 12-b og undersøker hva som hender hvis begge inngangene er LAV, ser vi at det skjer ikke noe spesielt dramatisk, men utgangene Q og \overline{Q} er begge HØY. De er altså ikke komplementære, som forutsetningen er. Når inngangene deretter begge går til HØY vil de to utgangene igjen være komplementære, men hvis inngangene skifter nøyaktig samtidig, vil det være tilfeldig hvilken tilstand Q havner i.

Vi vil litt senere i dette kapittelet se hvordan denne anomaliteten unngås.

Logikksymbol

Vipper er brukt så ofte at vi vanligvis benytter et funksjonelt, dvs. mer overordnet, kretssymbol for dem, i stedet for de detaljerte kretsskjemaene foran:



Fig. 14 SR-vipper, funksjonelt symbol

Vi kan også se dette som et *logikk-symbol*, som fremhever logisk funksjon fremfor elektrisk virkemåte.

1.5.2 D-vippen (lås)

D-vippen kan betraktes som en videreutvikling fra SR-vippen med aktiveringsporter:

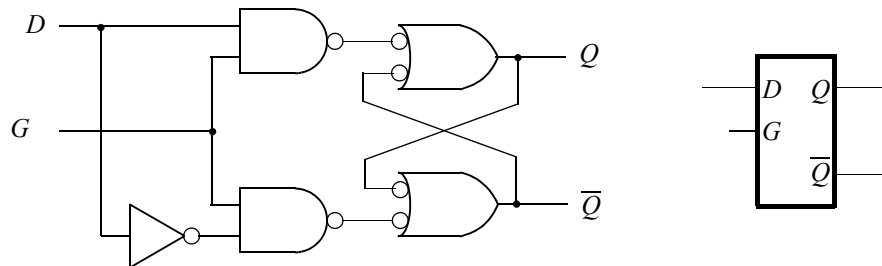


Fig. 15 D-vippe

D-vippen atskiller seg fra SR-vippen ved at den har bare én inngang utenom G . Denne inngangen kalles D , som står for Data. Figur 15 viser et logikkdiagram (logisk kretsdiagram) samt logikksymbol for D-vippen. Virkemåten er at når G kortvarig blir 1, vil verdien av D innsettes i vippen, dvs. $Q = D$. Når G går tilbake til 0, beholder Q sin verdi.

Hvis G blir stående =1 en stund mens D varierer, vil Q følge D 's variasjoner. Idet G går til 0, vil Q bli stående slik den var i dette øyeblikk. M.a.o. Q låses til verdien D hadde idet G gikk til 0, dvs. fra HØY til LAV. Av denne grunn kalles D-vippen også en **lås** (eng. latch). Endringer i D mens $G=0$ påvirker ikke Q .

Med denne koplingen unngår vi anomaliteten beskrevet foran, dvs. en ikke-tillatt kombinasjon av inngangene. Siden de to datainngangene til NAND-portene foran vippen alltid er komplementære, vil NAND-portenes utganger aldri kunne være LAV samtidig.

1.5.3 Flanke-triggete flip-flop'er

En annen variant får vi hvis G -inngangen erstattes med en flanketrigget klokkepuls-inngang C , hvor vippens tilstand (utgang Q) bare kan endres idet C skifter fra 0 til 1, eller fra 1 til 0. I førstnevnte tilfelle har vi en vippe som trigges på positiv flanke, i sistnevnte er det triggering på negativ flanke. En gitt flip-flop er enten positivflanke trigget eller negativflanke trigget.

Den vanligste bruken av disse kretsene er at C er tilknyttet en felles klokkepuls-generator. Dermed vil de forskjellige flipflop’ene skifte tilstand bare ved disse samme tidspunkt. Kretsene opererer altså *synkront*, og vi har å gjøre med *synkron logikk*.

Det finnes flere forskjellige typer flanketriggete flip-flop’er, men her skal vi bare beskrive de to viktigste: D-flipflop og JK-flipflop.

Synkron D-vippe

Den synkrone D-vippen (også kalt *klokket D-vippe*) likner på D-låsen beskrevet i kapittel 1.5.2, med den forskjell at Q’s tilstand blir bestemt av D idet øyeblikk C skifter fra 0 til 1 (positiv-trigget flipflop) eller fra 1 til 0 (gjelder for negativ-triggete flipflop’er). Utenom dette bestemte øyeblikk, er tilstand Q upåvirket av D. Logikksymbolet for D vippen er:

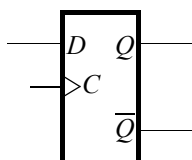


Fig. 16 Logikksymbol for synkron D-vippe

Tilstandstabell:

Innganger		Utganger		Forklaring
D	C	Q	\bar{Q}	
1	↑	1	0	SETT (lagrer en 1)
0	↑	0	1	RESETT (lagrer en 0)

JK-vippe

JK-vippen er allsidig og mye brukt. I tillegg til klokkeinngangen har den to innganger J og K og likner på flanketrigget RS-vippe, men med den forskjell at J- og K-inngangene er fullstendig uavhengige av hverandre. JK-vippen har altså ingen “ikke-tillatt” kombinasjon av inngangenenes tilstander.

Logikksymbol:

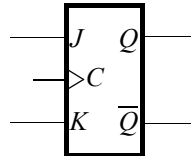


Fig. 17 Logikksymbol for synkron JK-vippe

Tilstandstabell:

Innganger			Utganger		Forklaring
J	K	C	Q	\bar{Q}	
0	0	\uparrow	Q_0	\bar{Q}_0	Ingen endring
0	1	\uparrow	0	1	RESETT (lagrer en 0)
1	0	\uparrow	1	0	SETT (lagrer en 1)
1	1	\uparrow	\bar{Q}_0	Q_0	Skift til motsatt (eng. toggle)

Asynkron Preset og Clear (Sett og Resett)

I tillegg til de synkrone inngangene, har de synkrone D- og JK-vippene gjerne også to asynkrone innganger. Ved hjelp av disse kan man tvinge vippen til henholdsvis 1 eller 0, uavhengig av D eller JK eller klokke-innganger. Disse spesielle inngangene er vanligvis Aktiv-LAV, dvs. de skal normalt holdes i den inaktive stillingen HØY. Med en LAV puls på en av disse kan man altså tvangssette vippen til 1 eller 0. Det er konstruktørens plikt å påse at disse spesielle inngangene aldri kan bli LAV (dvs. *aktive*) samtidig.

Logikksymboler:

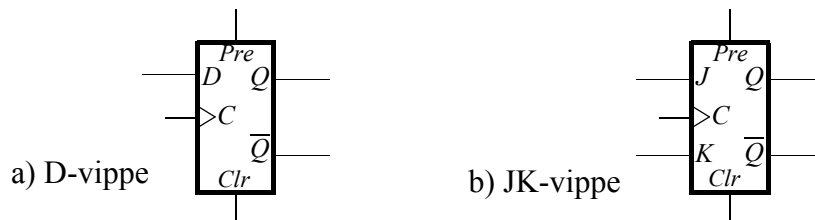


Fig. 18 Asynkrone Sett og Resett-innganger i synkrone vipper

Eksempel med JK-vippe

En JK-vippe med negativ flanketrigging tilføres klokkepulser som vist i tidsdiagrammet nedenfor. Diagrammet viser også tilført signal til J- og K-inngangene, og vi kan se resultatet som forløpet av utsignalet Q.

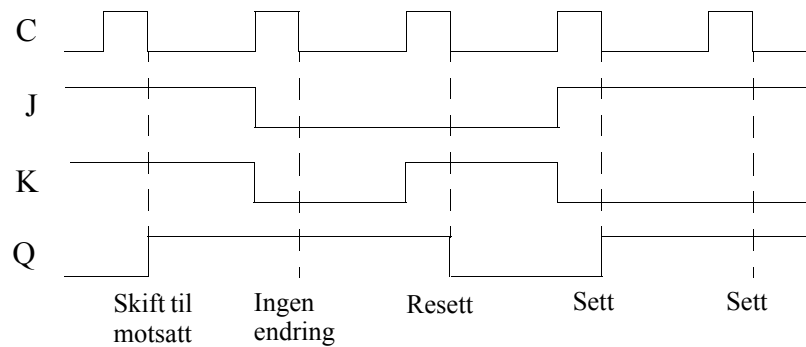


Fig. 19 JK-vippe: Signaler på innganger og derav utgangssignal Q

1.6 Sekvensielle funksjoner

Sekvensielle funksjoner:

Sekvensielle funksjoner avhenger av kombinasjoner av inndata og historie. Historien uttrykker i hvilken grad og på hvilken måte nåværende funksjonsverdi påvirkes av hva som har foregått tidligere. Dette kommer til syne gjennom funksjonens øyeblikkelige *tilstand*, som igjen kan avhenge av *tidligere tilstand*. Hvis nåværende tilstand, sammen med kombinatoriske funksjoner, entydig uttrykker funksjonsverdien, og tilstanden er uavhengig av hvordan systemet kom i denne tilstanden, sier vi at systemet, i denne tilstanden, følger Markov-regler. Hvis dette gjelder alle tilstander, er systemet i helhet et Markov-system.

En systemmodell basert på tilstander illustreres gjerne i form av en *graf*, med distinkte noder som står for tilstandene. En tilstandstransisjon, dvs. overgang fra en tilstand til en annen, forårsakes av en *diskret hendelse* (eng. “discrete event”) og skjer momentant, ideelt betraktet. Eksempler på diskrete hendelser er diskrete forandringer av signalverdier, som forandring av binære verdier, at en kontinuerlig variabel passerer en gitt grense ($x > x_0$), etc. Vi vil ta for oss dette, med prinsipiell angrepsmåte, i kapittel 1.6.3, men først vil vi se litt på et viktig fenomen ved praktiske koblinger.

1.6.1 Tids-usikkerhet

I avsnittet over ble *diskret hendelse* introdusert og forklart som momentan, ideelt sett. Praksis er imidlertid ikke ideell, og dette gjelder naturligvis også praktiske realiseringer av logikkretser. En diskret tilstandsændring i elektroniske logikkretser er et plutselig nivåskifte, vanligvis i elektrisk spenning, f.eks. fra 0V til nær 5 volt. Denne “plutselige” nivåændringen skjer ikke med uendelig hastighet, men i løpet av gjerne noen nanosekunder eller mer, altså med en skrå flanke, ikke rett firkant. Dessuten forplanter denne spenningsændringen seg videre i kretskoblingen, gjennom forskjellige kretselementer, og dette tar også litt tid. I løpet av et kort tidsintervall er altså mange av kretselementene i en overgangsfase, og først når det hele har kommet i ro, vil kretsen som helhet være i den nye tilstanden. Hvis man ikke tar spesielle forholdsregler, kan logiske kombinasjoner av forskjellige kretselementer gi helt gale “svar”, og disse kan også være upredikterbare. Et eksempel:

En binær tellerkrets kan lages slik at når vi teller oppover og går fra 011 til neste trinn, så skal dette bli 100. Hvert binært siffer (bit) realiseres med en JK-vippe:

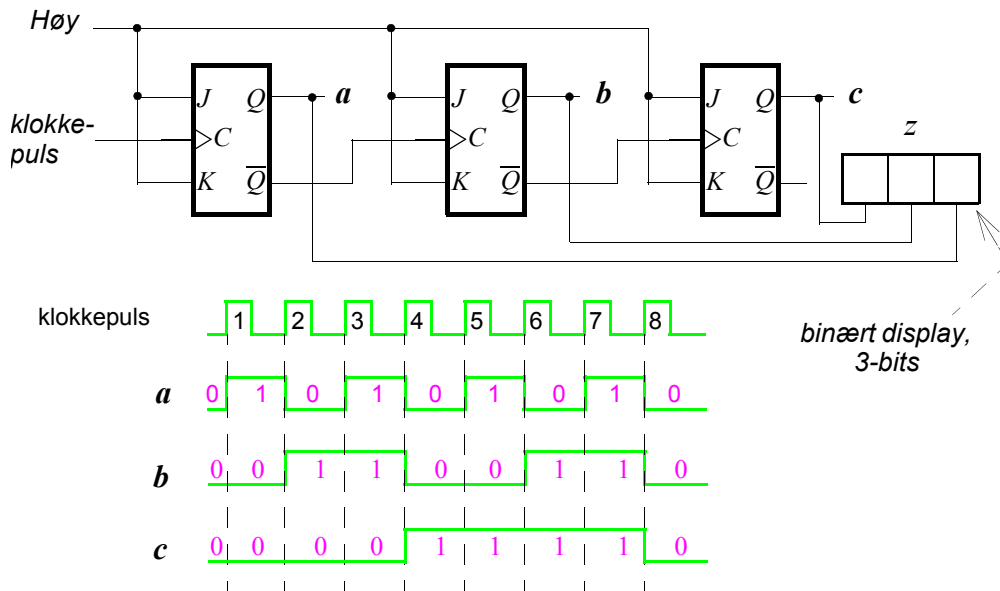


Fig. 20 3-bits asynkron teller og kappløp

Hvert trinn “klokkes” fra foregående trinn i denne koblingen. Den nevnte overgangen realiseres slik at når begge utgangene både $a=1$ og $b=1$ og vi får en ny tellepuls, vil $a \rightarrow 0$ og $b \rightarrow 0$ samtidig som det går en mente videre til neste vippe (c).

Disse hendelsene skjer innenfor et kort øyeblikk i forkanten av puls nr. 4: Først går $a \rightarrow 0$, så $b \rightarrow 0$ og deretter $c \rightarrow 1$. I løpet av dette korte overgangsintervallet har vi altså flere gale kombinasjoner, vist i rekkefølge ovenfra - ned:

c	b	a	z
0	1	1	3
0	1	0	2
0	0	0	0
1	0	0	4

overgangs-faser

Uttrykt som et trebits tall $z = cba$, så vil z i et kort øyeblikk gjennomløpe fra 3 til 2 og så til 0 før det ender på det riktige 4 (100 binært). Denne forplantningen gjennom kretselementer mens de stabiliserer seg kalles på engelsk “ripple” eller “race”, altså rippel eller kappløp. Asynkrontelleren kalles derfor også “rippel-teller”.

For å hindre at logikksystemet gjør gale ting under slike overgangssituasjoner, bruker vi gjerne **synkron** logikk, hvor en *klokkepuls* styrer slik at alle trinnene klokkes samtidig. I følgende kobling kan vi la $C = Lav$ (dvs. $C(L)=1$) være tidsintervallene når alle tilstandene er stabile og korrekte. Tellerkretsen skifter på positiv flanke av C , dvs. når C går fra L til H (logisk 1 til 0). Før de enkelte kretstrinnene vinner å stille seg om som følge av at $C \rightarrow H$, har korrektbetingelsen opphørt å være sann, dvs. $C(L) = 0$.

Vi kan ta utgangspunkt i den asynkrone telleren i Figur 20 og forandre koblingen litt og får en **synkron**-teller:

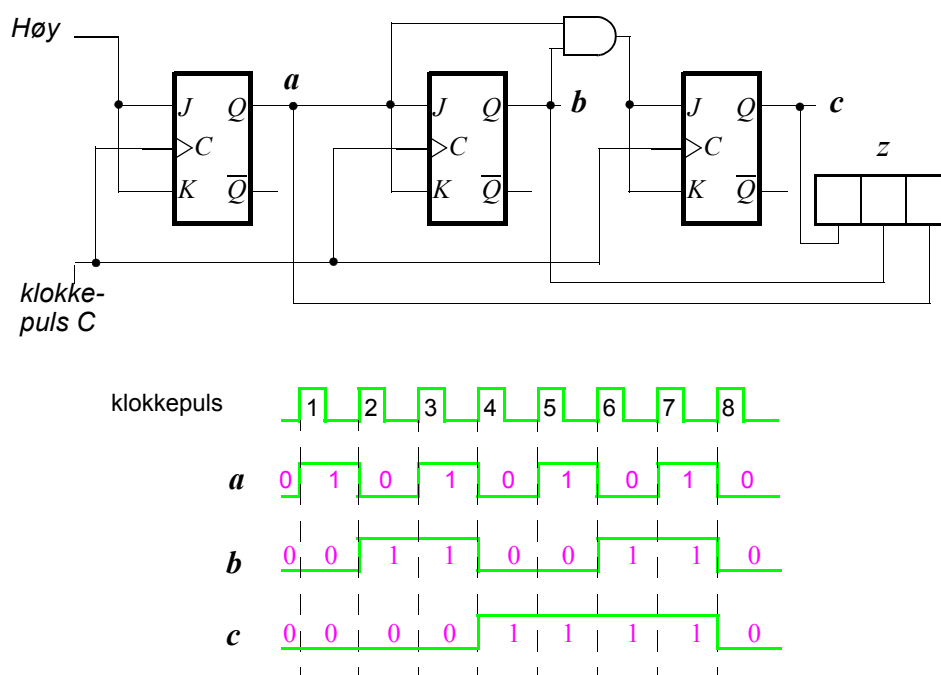


Fig. 21 3-bits synkron teller uten kappløp

Hver enkelt JK-krets stiller seg om samtidig, når $C \rightarrow H$, og de gjør dette i henhold til sine J-K -innganger i dette klokkeøyeblikket. Alle J og K er fast koblet til Høy, og dette innebærer at vippene stiller seg om til motsatt tilstand. Se forklaringen om JK-vipper side 20.

I den synkrone telleren skifter alle vippene “samtidig”, i motsetning til den asynkrone telleren, hvor vippe nr. n skifter som følge av at nr. $(n-1)$ har skiftet. Likevel setter jeg ordet “samtidig” i anførselstegn, for her har vi å gjøre med en annen type kappløp, som faktisk kan være enda verre enn ved rippeltelleren: Et kappløp som skyldes små ulikheter fra én integrert krets til en annen, og som gjør at to vipper som skulle skifte samtidig endrer seg med kanskje noen få nanosekunders forskjell. Moderne elektroniske kretser er så raske at noen få nanosekunder kan ha betydning, og det farlige med et slikt kappløp er at det er upredikterbart. Denne usikkerheten unngås ved å sørge for at vi ikke aksjonerer i den nye tilstanden før et kort øyeblikk etter klokkepuls-flanken som trigget tilstandsovergangen. Det er en enkel måte å gjøre dette på: Hvis triggering skjer på stigende pulsflanke, lar vi fallende pulsflanke være det tidspunkt hvor vi foretar en (ny) aksjon som følge av den nye tilstanden.

Dermed kan vi vende tilbake til innledningen av dette kapittelet, Sekvensielle funksjoner på side 23: Siden tiden i et digitalt system er diskret og går fremover skrittvis med ett skritt pr. systemklokke-puls, vil **en diskret hendelse skje momentant, og vi trenger ikke ta forbeholdet "ideelt betraktet"**.

1.6.2 Sekvenstyper

Sekvensfunksjoner kan være *tidssekvenser*, *beregningsbetingede* eller *prosessbetingede* sekvenser og blandinger av disse. Ved prosessbetingede sekvenser er signaler fra prosessen bestemmende for overgang til en annen tilstand, og dette gjør denne typen *ikke-deterministisk*. Hvis sekvensen bare er bestemt av beregninger og av fastlagte tidspunkt, er sekvensen deterministisk.

Ved tidssekvenser skiftes fra en tilstand til en annen ved forutbestemte tidspunkt, ikke nødvendigvis ekvidistante. Mange sekvensstyringer følger et slikt prinsipp, i hvert fall delvis. Ved beregningsbetingede sekvenser vil beregningsresultater under veis være bestemmende for videre gang, dvs. vi-

dere sekvens av tilstander. Mange styringssystemer følger også delvis et slikt prinsipp. Begge disse typer kan imidlertid karakteriseres som *åpen sløyfe* -styring, siden det ikke skjer noen påvirkning av sekvensen ut fra hvordan prosessen oppfører seg. Styring etter blanding av disse to deterministiske prinsippene kan man betrakte som styring *etter oppskrift*: Oppskriften foreskriver for eksempel: Start motor, vent 2 sekunder, åpne ventil V, vent 30 sekunder mens en tank fylles, sett på oppvarming, varm i 15 minutter, åpne tømmeventil, steng etter 30 sek., osv.

En slik “blind” styring som antydnet over kan være enkel, men den har naturligvis en stor svakhet i at den ikke tar hensyn til hvordan *prosessen* utvikler seg. I de aller fleste tilfeller er derfor sekvensen i betydelig grad bestemt av *målte (innleste) prosess-signaler*, og dermed har vi en **prosess-betinget** sekvens. Som regel har vi altså en blanding av disse tre typer.

Ved blandet sekvenstype kan prosess-signaler bryte inn i en tidssekvens og endre denne, eller forskjellige undersekvenser kan innkoples ved forskjellige tidspunkt som bestemt av signaler innlest utenfra.

1.6.3 Generell formulering av sekvens-forløp

Sekvensielt forløp beskrives generelt gjerne med henvisning til sekvensmodell formulert av Mealy eller Moore. Vi skal her se på Mealys formulering:

En sekvensiell prosess kan beskrives ved:

1. Et endelig sett Q interne tilstander.
2. Et endelig sett P innganger.
3. Et endelig sett W utganger (ut-signaler).
4. Tilstandsforandring τ fra q_m til q_n hvor $q \in Q$
5. Utgangstransformasjon ω fra (q_m, p_i) til w hvor $q \in Q$, $p \in P$ og $w \in W$

Begge modellene illustreres ved en rettet graf, med knutepunkter for

tilstandene og linjer med piler for transisjonene mellom tilstandene:

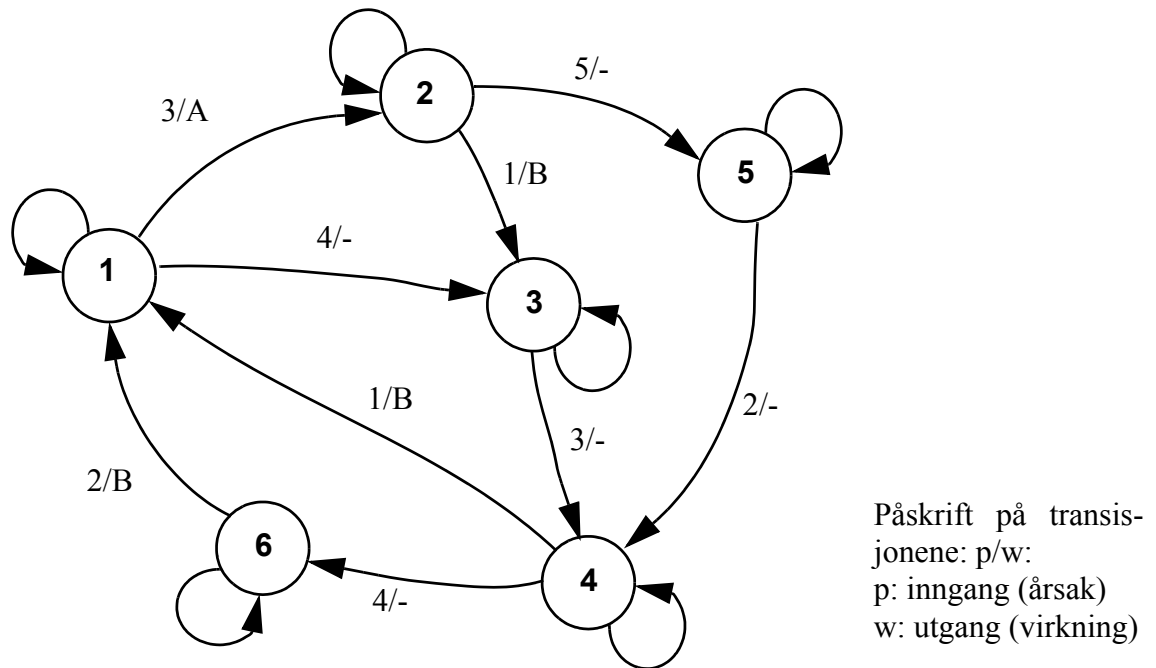


Fig. 22 . Tilstandsforandringer og tilhørende aksjoner

En sekvensiell prosess “er” i en og bare en tilstand til enhver tid. Om denne øyeblikkelige tilstanden kan vi si at den er *aktiv*. Egenskapen *aktiv* forflytter seg blant knutepunktene i den rettede grafen langs transisjonslinjene, det vil si prosessen går over fra en tilstand til en annen, som følge av “innganger” både i Moores og i Mealys modell. Slike “innganger” kan være hvilke som helst fri variable i et program, slik at logiske inn-signaler kan bearbejdes numerisk og påvirke modellens “innganger”. Tilstandsoverganger ledsages av en **aksjon**, karakterisert ved dannelsen av “utganger” (Mealy). En “utgang” er generelt altså en aksjon, noe som har en effekt. I vår sammenheng kan dette typisk være generering av et fysisk utgangssignal.

For en gitt tilstand $q_j \in Q$ og inngang $p_i \in P$, vil transformasjonen τ definere neste tilstand:

$$q_s = \tau(q_j, p_i) \in Q$$

Transformasjonen ω definerer utgangssignalene:

$$w = \omega(q_j, p_i) \in W$$

Tilstander og transformasjonene τ og ω kan illustreres med figur 22.

Det er altså knyttet en aksjon til *tilstandstransisjonen*, ikke til selve tilstanden. Ved Moores modell er derimot aksjonen knyttet til tilstanden, og det kan teoretisk vises at de to modellene er ekvivalente, dvs. et system fremstillet etter én av modell-typene kan transformeres til den andre modell-typen. Vi kunne derfor gjerne brukt Moores modell, men vi vil her til å begynne med bruke Mealys modell.

Et komplekst sekvenssystem må fremstilles hierarkisk, hvis vi skal kunne både behandle detaljer og samtidig beholde total-oversikten. Dette innebærer at *en enkelt aksjon* på høyt betraktningsnivå ofte må detaljeres ved å beskrives som et undersystem og fremstilles som en egen sekvensiell modell med tilhørende graf. Dette undersystemet trer i funksjon når og bare når aksjonen på nivået over utløses. I Moores modell er aksjonen knyttet til *tilstandene*, og dette betyr i denne sammenheng at den mer detaljerte undersekvensen aktiveres mens tilhørende tilstand på nivået over er *aktiv*.

1.6.4 Analyse av en sekvensiell krets

Vi tar for oss en kretskobling som ved første øyekast kan se enkel ut, men som inneholder noen interessante egenskaper:

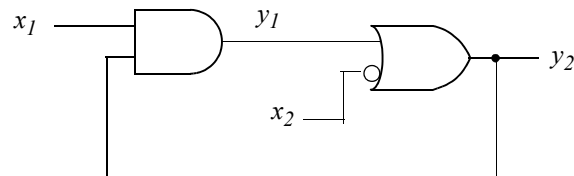


Fig. 23 Eksempel på sekvensiell krets

Når vi skal analysere en slik krets, kan vi starte med å sette opp en tabell over alle kombinasjoner av inngangsverdier (x) og tilhørende avhengig variable (y), slik at vi kan finne systemets tilstander og tilstandsoverganger. I analysen beveger vi oss stort sett nedover i tabellen, men det er også avmerket en overgang oppover, merket med stiplet pil:

x_1	x_2	y_1	y_2	Tilstand
0	0	0	1	A
1	0	1	1	B
0	0	0	1	A
1	0	1	1	B
1	1	1	1	B
0	1	0	0	C
1	1	0	0	C
0	1	0	0	C
0	0	0	1	A
0	1	0	0	C

Eksempel på transisjon uten tilstandsending

Eksempel på mulig race hvis begge inngangene x_1 og x_2 skifter samtidig.

Vi starter med en tilfeldig valgt kombinasjon av inngangene, her $\langle x_1, x_2 \rangle = 0,0$, finner verdiene av de avhengig variable, her $\langle y_1, y_2 \rangle = 0,1$, og setter et navn på denne tilstanden, her kalt A. Så endrer vi én av inngangene, her $x_1 \rightarrow 1$. Av figur 23 ser vi da at $\langle y_1, y_2 \rangle = 1,1$, altså har vi en ny tilstand, og vi kaller den B. Vi kan etter hvert tegne et tilstandsdiagram, liknende det som ble gjennomgått på mer prinsipielt vis i kapittel 1.6.3, en rettet graf med tilstandene som noder illustrert som sirkler, og tilstandsoverganger som transisjoner illustrert med piler mellom nodene. Ved siden av pilene kan vi påføre hvilke endringer i inngangene som forårsaker transisjonen. Innen i hver sirkel angir vi tilstandens navn, samt verdien av de avhengig variable, her y_1, y_2 . Etter hvert oppstår diagrammet nedenfor:

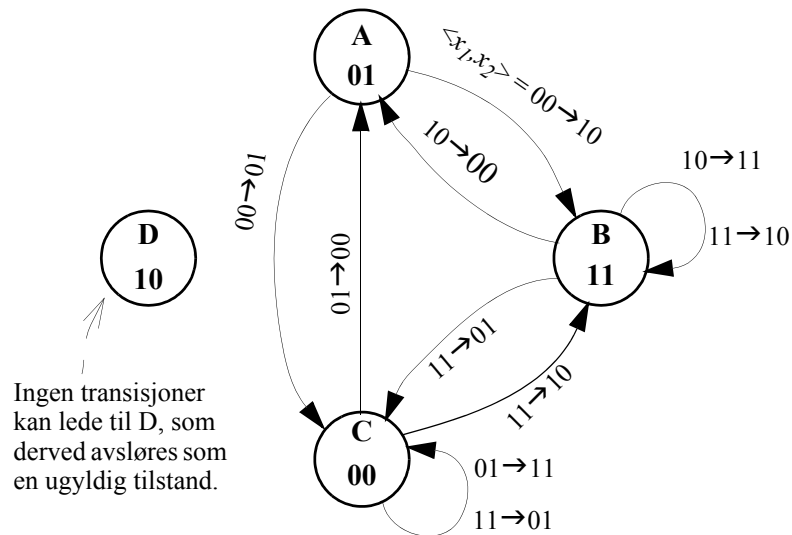


Fig. 24 Tilstandsdiagram, eksempel-krets

Vi har her definert tilstandene ut fra systemets utgangsvariable, og dette er jo naturlig fordi det er disse variablene som systemet genererer og som påvirker omgivelsene. Imidlertid kan vi under analysen få litt problemer fordi inn-variablenes nåtilstand kan være bestemmende for neste systemtilstand. M.a.o.: Noen av de tilstandene vi har definert kan ha indre tilstander. I vårt eksempel gjelder dette tilstandene B og C, så la oss trekke inn disse

interne tilstandene. Da blir diagrammet figur 24 videreutviklet:

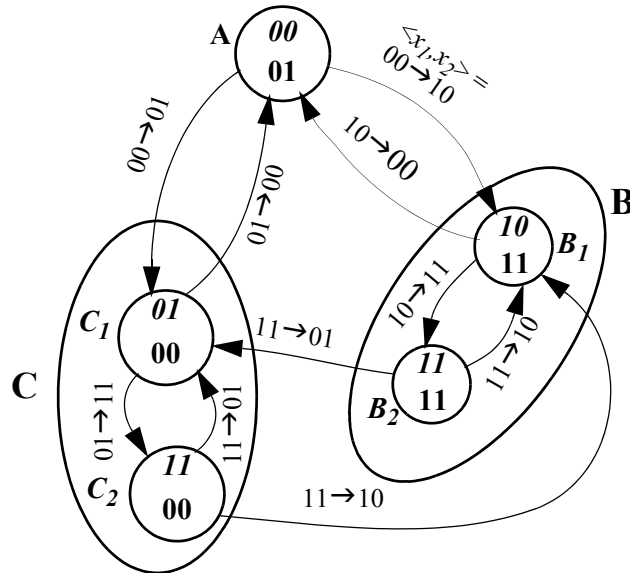


Fig. 25 Tilstandsdiagram med undertilstander

Her er de ovale tilstandsmarkeringene for B og C identiske med figur 24, men inne i både B og C har vi to interne tilstander, henholdsvis B_1 , B_2 og C_1 , C_2 . I både disse og i A har vi nå tilføyd inngangsverdiene (skrevet i kursiv for å gi forskjell fra utgangs-tilstandene). Transisjonene i figur 24 som fører tilbake til samme tilstand er nå inne i tilstandene B og C, som overganger mellom de *interne* tilstandene. Et naturlig spørsmål nå: Hvorfor beskrive dette i to trinn, som her først metoden i figur 24 og deretter figur 25? Svaret er at det er førstnevnte metode som er viktig, idet denne beskriver systemets egentlige tilstander. De interne transisjonene skyldes ofte *virkningsløse endringer av inngangsvariable*. Metoden med de "interne" tilstandene er bare noe som i visse tilfeller kan være nyttig for å hjelpe oss med å sikre at vi har dekket alle mulige transisjoner: Vi har her to (n) uavhengig variable, og i prinsippet skal det da kunne gå $2^n = 4$ transisjoner ut fra hver node. Hvis bare én av dem endrer seg av gangen (se avsnitt 1.6.1), reduseres antall transisjoner til n . Med kjennskap til kretsen og tidsforholdene, vil dette diagrammet hjelpe oss til å ha kontroll med at vi har fått med oss alle transisjonene. I figur 25 er det eksakt 2 transisjoner ut fra hver av tilstandene A, B_1 , B_2 , C_1 , C_2 , og det samme også i de ytre tilstandene A, B og C.

1.7 Tilstandstabeller for sekvensstyring

Overgang mellom de aktuelle tilstander, med derav følgende utlesning av utgangssignaler, kan oppstilles i forskjellige former for *tilstandstabeller*. Tilstandstabeller er hensiktsmessige for systematisk og oversiktlig behandling under programmeringen, og programmet kan hensiktsmessig arbeide direkte på disse tabellene. Vi skal nedenfor se på noen viktige former for tilstandstabeller.

1.7.1 Huffman-tabell

En av de viktigste typer sekvenstabeller er Huffman-tabellen. Huffman-tabellen uttrykker meget direkte teorien for tilstandstransformasjon, gitt foran. Forutsatt at Huffman-tabellen holdes på moderat størrelse, vil den også være hensiktsmessig som datastruktur for sekvensstyring innen sann-tidsprogram. Dette kan nødvendiggjøre tilstrekkelig system-oppdeling.

En Huffman-tabell fremstilles som en tre-dimensjonal matrise med $n \times m \times 2$ elementer, hvor n =antall tilstander og m =antall innganger. De to elementer $k = 1, 2$ for hver verdi i, j ($i \leq m, j \leq n$) inneholder informasjon om henholdsvis **neste tilstand** og **aksjon** eller **utgang**. De m innganger i tabellen representerer alle forekommende "inngangene" i Mealys modell. Disse kan være, og er ofte, avledet av *inn-signalene*, som en kombinatorisk funksjon. Inngangene i modellen, og dermed kolonnene i Huffmantabellen, representerer altså de *diskrete hendelsene* som ble omtalt i innledningen.

Huffman-tabellen kan oppstilles slik:



Tilstander q	Innganger					
	p_1	p_2	p_i	p_m
q_1	-/-	-/-		-/-		-/-
q_2	-/-	-/-		-/-		-/-
q_3	-/-	-/-		-/-		-/-
...						
q_j	-/-	-/-		q_s/w		-/-
...						
q_n	-/-	-/-		-/-		-/-

Fig. 26 . Huffman-tabell, skjematisk.

Element (i,j) inneholder to komponenter: neste tilstand q_s og utgang (aksjon) w .

Eksempel: Gardintrekk-mekanisme.

Som et meget enkelt eksempel på bruk av Huffman-tabeller skal vi betrakte en “gardintrekk”-mekanisme, av den typen som brukes for gardiner i auditorier, hvor motorstyring skjer fra en trykknappsats med tre knapper:

1. Gå mot venstre () (lukk)
2. Gå mot høyre () (åpne)
3. Stopp

Videre er mekanismen forsynt med en endebryter i hver ende av gardinføringen, slik at motoren stanser når gardinen er helt lukket, henholdsvis helt åpen. Disse gir altså to signaler:

4. Venstre ende (dvs. helt lukket)
5. Høyre ende (dvs. helt åpen)

Proessen representerer en rent Prosessbetinget sekvens. Systemet har i alt følgende fem mulige tilstander:

1. I ro ved venstre ende
2. I ro ved høyre ende
3. I ro i en mellomstilling
4. Underveis mot venstre
5. Underveis mot høyre

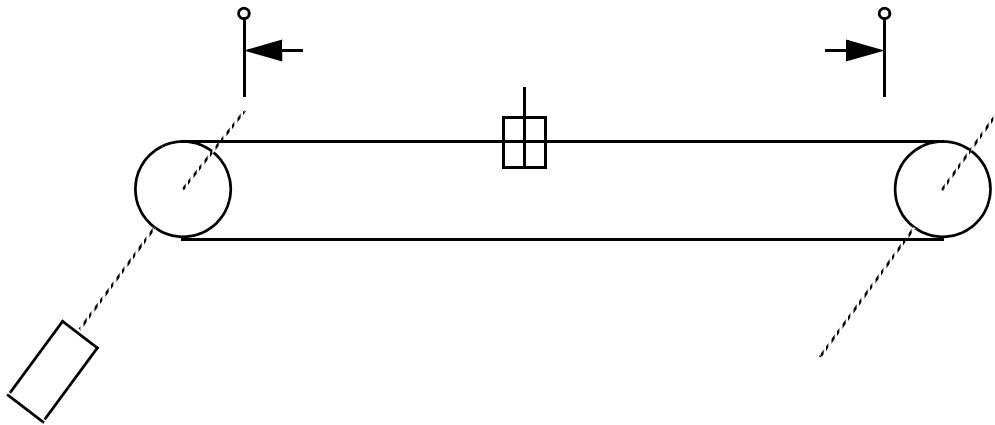


Fig. 27 . Gardintrekk-mekanisme, skjematisk

Gardintrekk-mekanismen kan beskrives med følgende Huffman-tabell, som uttrykker overgangen τ fra en tilstand til en annen som følge av forandringer i inngangene. Samtidig med hver overgang angis tilhørende utgangsfunksjon w .

Parentesmerkete situasjoner forekommer normalt ikke, bortsett fra eventuelt transient. Disse "situasjoner" bør likevel gis fornuftig aksjon, da de uttrykker feil-tilfeller, først og fremst utstyrsfeil.

Tegnforklaring for utganger:

V: Start mot venstre

H: Start mot høyre

S: Stopp motor

-: Ingen aksjon



Tilstander q	Innganger				
	Trykknapper			Endebrytere	
			STOPP	venstre	høyre
	1	2	3	4	5
1. I ro venstre	1/-	5/H	1/-	1/-	(1/-)
2. I ro høyre	4/V	2/-	2/-	(2/-)	2/-
3. I ro mellom	4/V	5/H	3/-	1/-	2/-
4. Mot venstre	4/-	4/-	3/S	1/S	(3/S)
5. Mot høyre	5/-	5/-	3/S	(3/S)	2/S

Fig. 28 Huffman-tabell for gardintrekk-mekanismen

For store systemer blir det gjerne stort antall tabell-innganger og dermed stor og uoversiktlig Huffman-tabell. Det kan da være hensiktsmessig å oppdele tabellen i flere mindre Huffman-tabeller. Dette innebærer imidlertid at modellen oppdeles i flere deler, som igjen er en forbundet graf, på et høyere hierarkisk nivå. En slik hierarkisk inndeling må være mulig ut fra systemets (prosessens) egenskaper, hvis en oppdeling av modellen skal være gjennomførbar. Hvis en slik inndeling synes vanskelig, henger det sammen med sterk og kompleks kopling innen systemets logiske struktur. Prøv å omarbeide og forenkle denne.

Det er viktig å merke seg forskjellen mellom innleste prosess-signaler og tabellinnganger. Selv om binære inn-signaler ofte opptrer direkte som tabellinnganger, som i eksempelet ovenfor om gardintrekk-mekanisme, er

dette slett ingen regel. Tabellinnganger svarer direkte til Mealy-modellens innganger, og disse kan prinsipielt like gjerne være boolske program-variable, beregnet i en eller annen program-del. I praksis har tabellinngangene dog ofte en forholdsvis nær sammenheng med innleste prosess-variable.

Forskjellen mellom Mealys og Moores modeller er som nevnt utgangene. Mealys modell knytter utgangene til transisjonene, mens Moore knytter dem til tilstandene. I eksempelet med gardintrekk-mekanismen vil etter Mealys modell aksjonene være å *starte* drivmotor til høyre eller til venstre, samt *stopp* av motor. Ved Moores modell ville tilsvarende aksjoner være *kjør* motoren. Ingen stopp er nødvendig som egen aksjon, for man forlater bare tilstanden hvor det kjøres. Realiseringen etter Mealys modell innebærer derfor utgangssignal i form av en puls til en (f.eks.) relekopling¹ med innebygget **hold**: For starting legg inn motorrele med holdekopling, og for stopp: frigjør holdekoplingen. I dette tilfelle kan det ut fra realiseringen med en viss rett argumenteres i favør av Moores modell: En digital utgang (en bit) fra datamaskinen går direkte til motorreleet, som derfor ligger inne så lenge som utgangen genereres. Ved feil som medfører at datamaskinen stopper kan man da utforme program og utgangskretser slik at utgangssignalet går til 0, hvorved motoren stopper. Ved Mealys modell må den samme sikkerheten oppnås ved at f. eks. drivspenningen for releene faller bort sammen med at datamaskinen faller ut.

Et annet eksempel:

En kontinuerlig variabel x innleses og sammenliknes med en grenseverdi x_0 , lagret som parameter. Hvis grensen overskrides, skal alarm gis. I Huffman-tabellen inngår da den boolske

$$b_1 = ((x - x_0) > 0)$$

som tabellinngang. I elementet for aktuell tilstand og denne inngang angis alarmtilstanden og tilhørende aksjon.

1. En slik relekopling med hold kan naturligvis gjerne realiseres i programvare. Det er den *prinsipielle* virkemåten vi poengterer her.

1.7.2 Beslutningstabell

Beslutningstabell (eng.: "decision table") er et hjelpemiddel som gir en overskuelig fremstilling av sammenhengen mellom betingelser og aksjoner. Den egner seg derfor godt for valg mellom en rekke *kombinatoriske* funksjoner.

Beslutningstabellen er særlig fordelaktig ved komplekse valgsituasjoner og gir, som få andre metoder, programmereren en god oversikt over problemet, samtidig som den gir lett mulighet for kontroll av at alle kombinasjoner av inngangene er dekket. I så måte er den på linje med Huffman-tabellen. Beslutnings-tabellen kan også benyttes direkte som programunderlag. Benyttes den sammen med Huffman-tabell i sekvensstyringsprogram, vil disse to kunne utfylle hverandre godt.

En beslutningstabell oppstilles, på papir, gjerne i 4 deler slik:

betingelsesstamme ("condition stub")	Regler							betingelsesdel ("condition value")
	1	2	3	4	5	6	7	
Liste over betingelser								
Liste over aksjoner								
aksjonsstamme ("action stub")	Aksjoner som utføres							aksjonsdel ("action value")

Fig. 29 . Beslutningstabell, skjematisk

Metoden kan best beskrives gjennom et enkelt eksempel, og vi kan da ta for oss en heis-styring, forholdene rundt en etasje. Senere vil vi utbygge eksempelet til å omfatte hele heisstyringen, men for å ikke gjøre eksempelet for komplekst i begynnelsen, tar vi som et utdrag forholdene rundt en etasje, som vi antar er en “typisk”, dvs. en mellometasje, hverken den helt nederste eller helt øverste. Vi kan kalle dette etasje nr. e_1 . På denne bakgrunn gjør vi den forenkling at vi her ikke tar hensyn til oppkall fra andre etasjer, idet dette hører med til den utvidede betraktningen.

Etasjene nummererer vi fra 1 (nederste) og oppover. Symbolsk betegner vi etasjen heisen står i som e_1 , og etasje som heisen skal gå til: e_2 . Dette betyr da at heisen skal gå oppover hvis $e_2 > e_1$, og nedover hvis $e_2 < e_1$.

Vi har her følgende mulige normale inngangssignaler:

Dør lukket (dvs. mekanisk kontakt koplet til dør)	
Etasjeindikator (heisen befinner seg innen etasjen)	
Knapp i kupé:	Gå til etg. e_2
Knapp i kupé:	Stopp
Knapp i etasjen:	Jeg vil OPP
Knapp i etasjen:	Jeg vil NED

Aksjonene er:

- Start nedover
- Start oppover
- Stopp
- Feilindikering

For enkelhets skyld vil vi ved feil kun gi Feilindikering, uten å forfølge dette videre. I et virkelig tilfelle må dette behandles komplett.

Vi kan sette opp følgende beslutningstabell:

	Regler					
	1	2	3	4	5	6
Dør lukket	1	1	-	0	0	1
Etasjeindikator	1	1	-	0	1	0
Kupéknapp $e_2 > e_1$	-	1	-	-	-	-
Kupéknapp $e_2 < e_1$	1	0	-	-	-	-
Kupéknapp $e_2 = e_1$	-	-	-	-	-	-
Kupéknapp STOPP	0	0	1	-	-	0
Etasjeknapp OPP	-	-	-	-	-	-
Etasjeknapp NED	-	-	-	-	-	-
Start nedover	X					
Start oppover		X				
Stopp			X	X		
Feilindikering				X		

Fig. 30 . Beslutningstabell for heiskupé

For inngangene betyr - at det ikke tas hensyn til dette, spørsmålet er irrelevant. Svar 1 eller 0 er da likegyldig. For aksjoner betyr X at aksjonen skal utføres, mens blankt betyr ingen aksjon. Signalet fra kupéknappene angir numerisk ønsket etasje. Dette kunne angis som en heltallsvariabel, ikke boolsk. Imidlertid kan det forekomme at noen trykker på knapper både opp og ned samtidig. Dette må tas hånd om på en fornuftig måte, og her er det valgt å la passende elektroniske sammenlikningskretser gi de tre signalene som direkte inngår som innganger i tabellen, dvs. der står det boolske resultatet av sammenlikningen med den etasje heisen befinner seg i. Vi kommer for øvrig tilbake til dette spørsmålet om samtidig trykk på flere knapper.

Med et annet system, kunne det meget vel tenkes at det bare var én mulighet, uttrykt som en numerisk verdi. Eksempel: Nivåmåling i en tank.

Målesignalet gir entydig nivået. Da ville det være naturlig å sette de boolske relasjonene inn i tabellen, slik som her, men hvor samtidig “klaff” på flere enn én samtidig ikke var mulig, fordi sammenlikningen foregår i programmet, dvs. en beregning på grunnlag av *én* foreliggende innlest verdi. Dette er en viktig detalj og viser hvordan numeriske variable benyttes til å gi innganger i beslutningstabellen, og dette er også et eksempel på prosessbetingelser, omtalt i avsnitt 1.6.2, side 26.

Tabellen må sies å gi et godt og oversiktlig bilde av forholdene, og det er forholdsvis lett å kontrollere at alle kombinasjoner er tatt med.

Kolonnene i tabellens høyre del representerer en rekke **regler**: En kolonne angir kombinasjoner av inngangene, og hvis et foreliggende tilfelle har 0 på alle de steder hvor regelen foreskriver 0, og 1 alle steder hvor det står 1 i kolonnen, har vi “klaff”, og tilhørende aksjoner nedenfor skal utføres. For de innganger hvor det står - i kolonnen, tas ikke hensyn til inngangsverdien, dvs. inngangsverdien er *irrelevant*.

Sammenlikner man de forskjellige reglene, kan det tenkes at flere gir samsvar med foreliggende situasjon. Denne *flertydighet* løses vanligvis slik at vi foreskriver undersøkelse av regel etter regel fra venstre mot høyre, inntil en regel med samsvar er funnet. Da stanser sammenlikningen. Det kan også tenkes at ingen regler passer: Beslutningstabellen ville da være *man-gelfull*. Dette løses ved at man tar med, som en spesiell regel til slutt, en *ellers*-regel, som dekker “default”-situasjon. Denne regelen settes altså opp slik at den gir klaff i alle tilfeller der de øvrige ikke passet. Ofte vil denne regelen foreskrive aksjonen “feil-reaksjon”, eller den kan bringe kontrollen i programsløyfen tilbake til utgangspunktet uten å gjøre noe. Gjennomløpet av tabellen blir altså uten virkning. Denne algoritmen om å undersøke fra venstre mot høyre innebærer at innbyrdes rekkefølge mellom reglene kan være av betydning.

Når vi skal lage et program som behandler beslutningstabellen, går vi frem slik:

1. Oppstill en boolsk BETINGELSESMATRISSE svarende til betingelser og regler, hvor alle JA angis med 1 og alle NEI eller IKKE RELEVANT med 0.

2. Oppstill en boolsk MASKEMATRISE av samme form som Betingelses-matrisen. Her innsettes 1 tilsvarende alle relevante spørsmål og 0 i alle ikke-relevante.
3. En gitt situasjon karakteriseres med en DATAVEKTOR med boolske (binære) komponenter som angir svar på betingelsene.

Vi får derved følgende algoritme:

I tur og orden undersøkes slik om datavektoren svarer til en av reglene: Dann OG-funksjonen (logisk produkt) av datavektor og vedkommende kolonne i maske-matrisen. Hvis resultatet er identisk med tilsvarende kolonne i betingelsesmatrisen, er overensstemmelse funnet. Tilsvarende aksjon skal utføres.

For dette aktuelle eksempelet blir betingelsesmatrise og maskematrise slik:

BETINGELSEMATRISE

1	1	0	0	0	1
1	1	0	0	1	0
0	1	0	0	0	0
1	0	0	0	0	0
0	0	0	0	0	0
0	0	1	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

MASKEMATRISE

1	1	0	1	1	1
1	1	0	1	1	1
0	1	0	0	0	0
1	1	0	0	0	0
0	0	0	0	0	0
1	1	1	0	0	1
0	0	0	0	0	0
0	0	0	0	0	0

Antar vi eksempelvis at situasjonen kan beskrives slik: Vi er i 2. etasje. Heisdøren er lukket, etasjeindikatoren viser at heisen befinner seg i denne etasjen, og det trykkes på kupéknapp for å gå til 4. etg. Da er datavektor slik:

1
1
1
0
0
0
0
0
0

Bemerk at datavektoren beskriver foreliggende tilfelle, derfor vil aldri noen komponent her være “ikke-relevant”, “ubestemt” eller liknende.

For denne datavektoren blir OG-funksjonen mellom datavektor og hver spalte i maskematrisen:

1	1	0	1	1	1
1	1	0	1	1	1
0	1	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Undersøkes overensstemmelsen med Betingelsesmatrisen, ser man at regel 2 gir overensstemmelse, de andre regler ikke. Aksjon oppført under regel 2 skal da utføres, og dette er “Start oppover”.

En nyttig regel for oppsetting av en betingelsestabell: Utnytt eksisterende innbyrdes forhold mellom betingelsene. Hvis JA for en betingelse nødvendigvis medfører NEI for en annen, bør én av disse angis med -, dvs. LIKEGYLDIG (Ikke relevant).

Eksempel: Betingelser:

Alder > 68 år	Det er åpenbart ikke mulig med JA på
" > 50 "	alle disse betingelser samtidig, dvs.
" < 18 "	de utelukker hverandre (delvis).

I tilfellet med heisen har vi noe tilsvarende med kupé-knappene: Det gir ingen mening å gå både opp og ned samtidig. I dette tilfelle er imidlertid slikt motstridende *signal* absolutt mulig, det viser at en person har trykket flere knapper samtidig, evt. det har oppstått f. eks. overledning på kablingen eller annen liknende systemfeil. Dette må det tas hensyn til, og det kan være nærliggende i første omgang å anta at det er trykket samtidig på flere knapper. Systemfeil kan vi ta hånd om ved at hvis situasjonen varer over en viss tid, f.eks. en time, kan dette utløse feil-alarm. Vi går ikke inn på muligheten for systemfeil her, men derimot må vi ta hensyn til samtidig trykking av flere knapper. Dette har vi gjort i tabellene over: Man har her bestemt at knappen for egen etasje helt neglisjeres. Da har vi tre muligheter igjen:

entydig Neste etasje er ovenfor,
entydig Neste etasje er nedenfor,
Oppkall om både opp og ned

I dette eksempelet viser tabellen at i tilfellet “både opp og ned” er “ned” gitt prioritet over “opp”, dvs. det skal da reageres som om bare “ned” er trykket. M.a.o. for at kommandoen skal oppfattes som “oppover”, må trykkes entydig knapp for høyere etasje.

Med Beslutningstabeller kan blant annet oppnås følgende:

1. Logikken fremtrer presist og på kompakt form.
2. Kompliserte tilfeller blir oversiktlig fremstilt.
3. Tydelig relasjon mellom variable.
4. Forenklet programmering.
5. Enkel og oversiktlig dokumentasjon.
6. Overflødighet, tvetydighet og mangelfull tabell avsløres systematisk.

Beslutningstabellen egner seg godt til evaluering av betingelsene i kombinatoriske funksjoner som bestemmende for tilstandstransisjon i sekvensstyringsprogram.

1.8 Eksempel på sekvensstyre-system: Heis-styring

Vi vil nå utbygge eksempelet med heisstyring og velger å benytte Huffman-tabell og beslutningstabeller i kombinasjon: Huffman-tabell for det overordnede system og behandling av tilstander, mens beslutningstabellen benyttes til å avgjøre tilstandstransisjonene. Dette bygger videre på de oppstillinger av beslutningstabell vi laget i avsnitt 1.7.2 .

1.8.1 Heis-systemet

Heis-systemet er av de vanlige moderne systemene som kan beskrives slik:

Heisen skal betjene et ubestemt antall etasjer. Ved hver etasje (mellom nederste og øverste) finnes to trykknapper merket **OPP** og **NED**. Ved nederste og øverste etasje er det kun én knapp, henholdsvis **OPP** og **NED**. Trykk på en etasjeknapp kan gjøres når som helst, og dette registreres av styreprogrammet som oppfører dette som "bestilling" i to køer, *oppover-kø* og *nedover-kø*.

Heis-styresystemet kan beskrives med tilstandsmodellen i Fig. 31 på side 48. Utgangsposisjonen er tilstand nr. 1: Heisen står i ro, og intet er bestilt. Den står i en tilfeldig etasje. Som man ser av figuren, skiller ikke tilstandene mellom de enkelte etasjer, men programmet holder naturligvis rede på hvilken etasje heisen står i. Hvis det nå kommer "bestilling" på å gå oppover, enten ved at en person går inn i heiskupeen og trykker for en etasje ovenfor, eller det trykkes på en av etasjeknappene ovenfor, får vi først tilstands-transisjon til tilstand 2. En intern variabel som angir retning settes derved til OPP. Så startes heisen og vi får tilstandstransisjon til tilstand 3. Idet heisen passerer en etasje, går systemet til tilstand 4. Hvis dette var den bestilte etasje, går tilstand over til nr. 2, samtidig med aksjon **stopp**. Hvis etasjen derimot ikke var bestilt, går tilstanden over til nr. 3 og det fortsettes oppover. Slik hoppes det mellom tilstand 3 og 4 for hver "uvedkommende" etasje, inntil riktig etasje nås, og tilstanden altså blir 2. Hvis det etter dette foreligger flere bestillinger oppover (oppover-køen), fortsettes fra tilstand 2 til 3, og det hele gjentas.

Modellen er symmetrisk, så beskrivelsen ovenfor gjelder tilsvarende for retning **nedover**. Mellomtilstanden 1: **Nøytral** er dels hvor systemet befinner seg når intet er bestilt, og dels overgang mellom "opp" og "ned". Som vi ser av denne modellen, vil systemet "låse seg opp" i én retning så lenge det finnes bestillinger i denne. Dette er ønskelig, og overensstemmende med vanlige regler.

La oss nå se nærmere på bestillingskøene, kort omtalt på side 45. Egentlig bør vi kalle dem *bestillingstabeller*, for å unngå misforståelsen at de opererer etter "først inn - først ut"-prinsippet, som er vanlig for *køer*. Det er altså to enkelt-tabeller, en for hver retning heisen går. Vi organiserer dette som en todimensjonal tabell (matrise eller array), hvor den ene dimensjonen svarer til retning OPP eller NED, og den andre til etasjene, i dette tilfelle 1...5. Vi illustrerer dette slik:

Bestillingstabell:

Etasje	OPP	NED
5		
4		
3		
2		
1		

I utgangspunktet er bestillingstabellens celler tomme. Når en bestillingsknapp trykkes, plasseres et merke, flagg, i tilhørende celle, og dette skal bevirke at heisen stopper ved vedkommende etasje når den er på vei henholdsvis oppover eller nedover. Idet heisen stopper ved en etasje, nullstilles tabellcellen.

Trykkes en etasjeknapp, skal flagg settes i OPP- eller NED-cellen for vedkommende etasje, uavhengig av heisens umiddelbare posisjon eller tilstand (bortsett fra hvis heisen allerede står i vedkommende etasje). Trykkes en kupéknapp (bestillingsknapp i heiskupéen), skal flagg settes dersom man trykker på knapp for en høyere liggende etasje når heisen er under veis oppover. Tilsvarende for nedover. Derimot skal ikke registreres bestilling til lavere etasje når heisen er under veis oppover, og tilsvarende høyere etasje mens den er på vei ned. Dette er valgt fordi slike bestillinger kan anses for feilbetjening: Det kommer inn i heiskupeen passasjerer som egentlig skal fraktes i motsatt retning av heisens rute, eller det er trykket feil. På denne

måten får vedkommende passasjer mulighet for å trykke om igjen. Evt. må han/hun bare bli med heisen i "feil retning", og kan gjenta bestillingen når heisen er kommet til høyeste, henholdsvis laveste, bestilte etasje. På denne måten oppnås også en enkel indikering på lysende knapper i kupeen av fremtidige stopp, slik at passasjerer får et klart bilde av stoppene fremover.

Forbindelsen mellom trykknappene og tabellen er forholdsvis triviell og realiseres nokså direkte med programmert innlesning på lavt nivå av signaler fra knappene. Dette blir derfor ikke berørt videre her; vi forutsetter i det følgende at slik forbindelse er realisert, slik at knapptrykking straks kommer til syne som logiske variable (SANN) i tabellens celler.

Ved hver etasje er en sensor som gir et logisk signal SANN når heisen ankommer en etasje. Signalet blir SANN like før ankomst til etasjen, slik at heismotoren vinner å stanse. Det går over til FALSK idet heisen forlater etasjen. Dette signalet endrer en tellervariabel `s_etg` som derved til enhver tid viser siste etasje heisen var ved, enten heisen da stoppet eller bare passerte. Variabelen `s_etg` tvangssettes til 1 når heisen kommer til 1. etg., for å sikre synkronisering.

En Huffman-tabell for denne heis-styringen kan nå settes opp, som i Fig. 32 på side 49. Tabellen (og modellen) tar bare for seg de "overordnede" funksjoner, tilstandsovergang i forbindelse med gang mellom etasjene, og den er noe forenklet, idet den bl.a. ser bort fra muligheten for trykk på stoppknappen mens heisen er under veis mellom etasjene. Huffmantabellen må ses i sammenheng med beslutningstabellen, liknende Fig. 30 på side 40. Vi kan sette opp en slik beslutningstabell for hver av tilstandene, og i programmet realiseres beslutningstabellen som et array av strukturer (struct i C, record i Pascal). En peker skifter mellom de enkelte beslutningstabellene, i henhold til tilstanden.

På noen følgende sider er vist beslutningstabeller for noen av tilstandene, og systematikken skulle fremgå av disse. Beskrivelsen er forhåpentlig tilstrekkelig til at man ser sammenhengen, men for fullstendighets skyld bør kanskje påpekes at her er "ellers"-regelen, som ble nevnt på side 41, at intet gjøres. Systemet forblir i sin tilstand så lenge ingen av reglene tilfredsstilles.

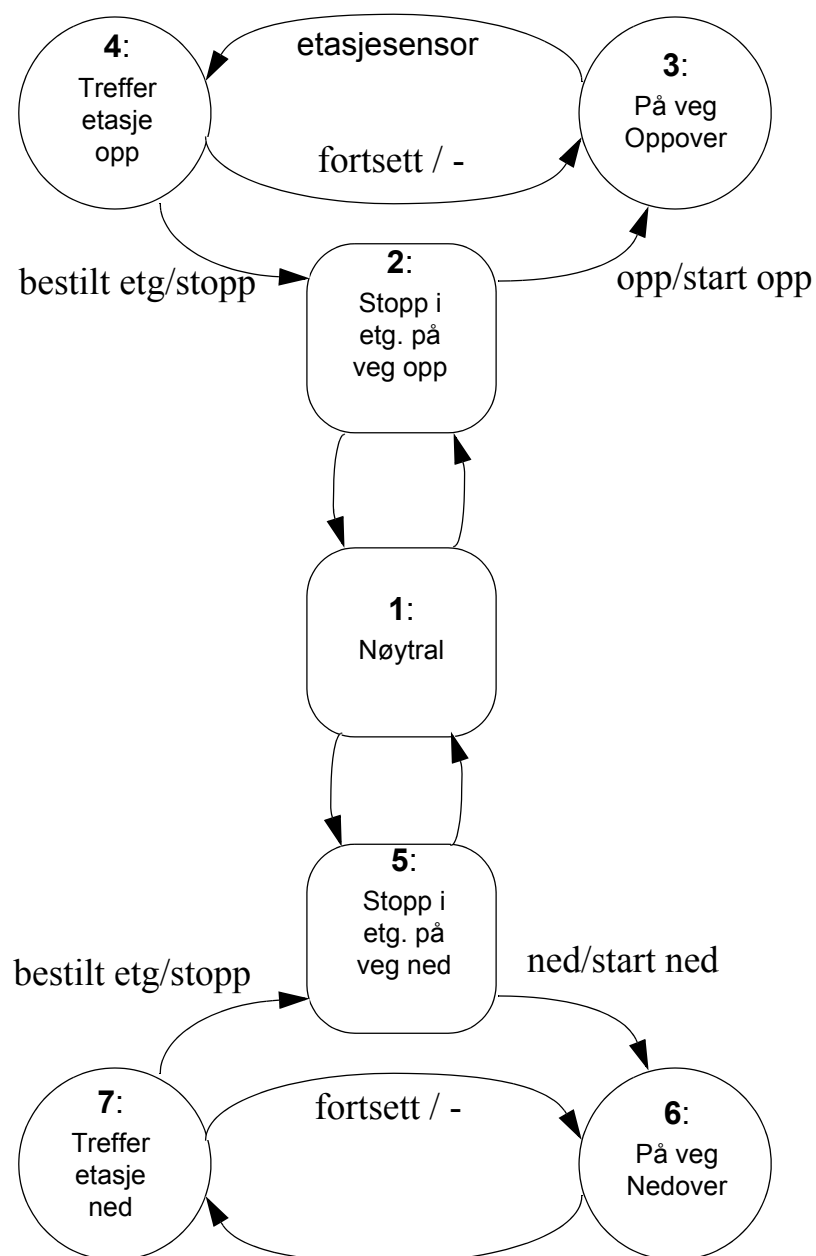


Fig. 31 . Tilstandsmodell for heis

Tilstander q	Innganger					
	Bestillingstabell		Etasjebryter	Programfunksjoner		
	Best.flagg finnes i celle >s_etg	Best.flagg finnes i celle <s_etg		tref-fer bestilt etasje	tref-fer "gal" etasje	tom bestil-lings-kø
1. Nøytral	2/-	5/-	-/-	-/-		
2. Stopp på veg opp	3/MO	-/-	-/-	-/-		1/-
3. På veg oppover	-/-	-/-	4/-	-/-		
4. Treffer etasje opp	-/-	-/-	-/-	2/MS	3/-	
5. Stopp på veg ned	-/-	6/MN	-/-	-/-		1/-
6. På veg nedover	-/-	-/-	7/-	-/-		
7. Treffer etasje ned	-/-	-/-	-/-	5/MS	6/-	
Tegnforklaring: MO: Motor Opp; MN: Motor Ned; MS: Motor Stopp						

Fig. 32 Huffmantabell for heisstyring

	Regler	
	1	2
Dør lukket	1	1
Etasjeindikator	-	-
Bestilling i celle >s_etg	-	1
Bestilling i celle <s_etg	1	0
Kupéknapp STOPP	0	0
Til tilstand 5 (ned)	X	
Til tilstand 2 (opp)		X

Fig. 33 . Beslutningstabell for tilstand 1

	Regler		
	1	2	3
Dør lukket	-	-	0
Etasjeindikator	1	1	0
Bestilt etasje	1	0	-
Til tilstand 2, Stopp: MS	X		
Til tilstand 3		X	
Feilindikering			X

Fig. 34 Beslutningstabell for tilstand 4

	Regler		
	1	2	3
Dør lukket	1	-	-
Etasjeindikator	1	1	0
Ventetid i etg. ferdig	1	1	1
Best.flagg finnes i celle >s_etg	1	0	-
Kupéknapp STOPP	0	-	-
Til tilstand 3, Start: MO	X		
Til tilstand 1		X	
Feilindikering			X

Fig. 35 . Beslutningstabell for tilstand 2

For beslutningstabellene er det nærliggende å spørre om regler for å kontrollere om alle kombinasjoner er med. En systematisk måte er Karnaugh-diagram, som vil bli gjennomgått i et senere kurs i logikk-design. Karnaugh-diagram er imidlertid ikke praktisk, hvis antall variable overstiger 5. En annen metode er Quine-McClusky's, men denne er ganske omstendelig. Teoretisk har man, med n uavhengig variable, 2^n kombinasjoner. Dette blir imidlertid alt for tungvint. I de aller fleste tilfeller ønsker vi tilstandstransisjon i noen få tilfeller, når bestemte betingelser inntreffer. Før dette skjer, foretas det intet. Dermed blir de interessante kombinasjonene ganske fåtallige, slik at systemet blir oversiktlig. Karnaugh-diagram kan da benyttes, hvis man ikke innser kombinasjonene direkte. Karnaugh-diagrammet gir bl.a. en metode til å finne "don't cares", altså - (strek) i diagrammet. Disse kan også finnes direkte, der to regler med samme aksjon(er) har like betingelser unntatt én, hvor vi har 0 og 1.

Som det fremgår av tabellene ovenfor, har vi egentlig ikke benyttet oss av Huffmantabellen. Beslutningstabellene inneholder selv aksjoner for skifting av tilstand, og det hører én beslutningstabell til hver tilstand. Tilstanden er gjennom dette *realisert* gjennom sin beslutningstabell. Dermed er det tilstrekkelig med settet av beslutningstabeller, organisert innen styre-

programmet på en systematisk måte med peker som skifter mellom tabellene i henhold til tilstandene.

Til slutt skal vi se hvordan vi kan stille opp en beslutningstabell for setting av flagg i bestillingstabellen, basert på knappetrykk (umerkede rubrikker: likegyldig ("dont'care")).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Kupéknapp 5	1															
Kupéknapp 4		1			1											
Kupéknapp 3			1			1										
Kupéknapp 2				1			1									
Kupéknapp 1								1								
Etasjeknapp ↑ i 4. etg.									1							
Etasjeknapp ↑ i 3. etg.										1						
Etasjeknapp ↑ i 2. etg.											1					
Etasjeknapp ↑ i 1. etg.												1				
Etasjeknapp ↓ i 5. etg.													1			
Etasjeknapp ↓ i 4. etg.														1		
Etasjeknapp ↓ i 3. etg.															1	
Etasjeknapp ↓ i 2. etg.																1
Knapp > s_etg	1	1	1	1												
Knapp < s_etg					1	1	1	1								
Knapp = s_etg	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bestillingstab. OPP, 5.etg.	X															
Bestillingstab. OPP, 4.etg.		X							X							
Bestillingstab. OPP, 3.etg.			X							X						
Bestillingstab. OPP, 2.etg.				X							X					
Bestillingstab. OPP, 1.etg.												X				
Bestillingstab. NED, 5.etg.													X			
Bestillingstab. NED, 4.etg.					X									X		
Bestillingstab. NED, 3.etg.						X									X	
Bestillingstab. NED, 2.etg.							X									X
Bestillingstab. NED, 1.etg.								X								

Fig. 36 Beslutningstabell for oppsetting av Bestillingstabell

1.9 IEC-848: Grafisk notasjon for sekvensstyring

Mens de grunnleggende Mealy og Moore -modellene gjerne fremstilles grafisk i form av generelle rettede grafer, slik som Fig. 22 på side 28 og eksempelet i Fig. 31 på side 48, finnes det spesielle grafiske notasjoner som er mer "skreddersydd" for sekvensstyring. Disse inneholder en del spesifikke detaljer som gir mer detaljert uttrykksfullhet. En viktig blant disse er IEC-standard 848 (Preparation of function charts for control systems) som også bygger på den franske standarden Grafcet og den tyske DIN 40719. For enkelhets skyld omtaler vi denne IEC-måten derfor som Grafcet-metoden, selv om det korrekte egentlig er "Standard IEC-848".

IEC-848 benytter også rettede grafer. En slik graf kalles her *funksjonsdiagram* (Function chart). Hierarkisk oppdeling (se side 29) er meget enkelt å gjennomføre med denne metoden. Det som i Mealy og Moores modeller kalles *tilstand* heter *trinn* i IEC-848. Den svarer til Moores modell ved at aksjoner er knyttet til trinnene (tilstandene).

Den noe nyere standarden IEC1131-3, beskrevet i kapittel 1.10, har tatt opp i seg IEC-848 med noen modifikasjoner. Denne del av IEC1131-3 kalles "Sequential Function Charts" (SFC) og er én av de fem metodene som inngår i IEC1131-3.

Et funksjonsdiagram består av **trinn**, **transisjonsbetingelser** og **rettede forbindelser**, som sammenknytter trinn og transisjoner.

1.9.1 Trinn

Et trinn tegnes som et rektangel (helst kvadrat) med etikett (nummer og/eller tekst). Teksten kan vise hva som skjer i trinnet. Ett bestemt trinn i et sekvenssystem er *starttrinnet*. Dette representerer starten i sekvensen og tegnes som et rektangel med dobbel ramme.

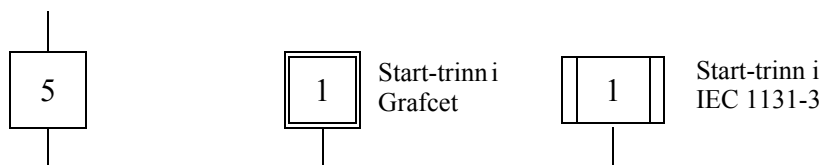


Fig. 37 . Symbol for trinn i Grafcet og 1131-3

Under beskrivelsen av et sekvensielt system kan vi ha bruk for å illustrere en øyeblikkssituasjon. Da vil systemet befinne seg i en bestemt tilstand, dvs. ett bestemt trinn i funksjonsdiagrammet er *aktivt*. Dette kan illustreres med en prikk i trinn-rektangelet:

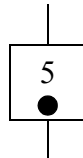


Fig. 38 .

1.9.2 Kommandoer og aksjoner

Til et trinn kan knyttes *kommandoer* og *aksjoner* (begge går under begrepet *aksjoner* i Moore) som utføres mens trinnet er aktivt. Det skilles ikke strengt mellom kommandoer og aksjoner, men kommandoer er noe som utføres i trinnet, mens en aksjon er noe som forårsakes i trinnet. Eksempel: "Åpne ventil 2" er en kommando, mens "Åpning av ventil 2" er en aksjon. Det er ellers ikke grunn til å utdype denne forskjellen videre, og i de fleste tilfeller vi i det følgende omtaler en *kommando*, kan man like gjerne erstatte den med en *aksjon*.

En kommando eller aksjon angis i en rektangulær boks knyttet til trinnet:

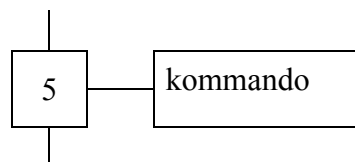


Fig. 39 .

En kommando iverksettes når det tilhørende trinnet blir aktivt. Idet trinnet blir deaktivert idet et etterfølgende trinn overtar som aktivt, vil kommandoen enten

- terminere, eller
- beholde sin tilstand.

Det er m.a.o. anledning til å skille mellom en "enkelt-handling" (transient) og en aksjon som strekker seg over tid ("steady state"). Forskjellen fremgår av følgende lille eksempel:

- a. En ventil åpnes i trinn 5, og det er underforstått at ventilen lukkes idet trinnet forlates:

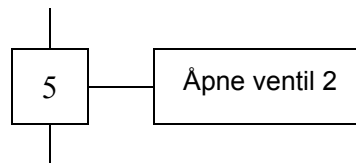


Fig. 40 .

- b. Hvis lukkingen skal knyttes til et etterfølgende trinn kan vi angi selve åpningen slik:



Fig. 41 .

Vi skal senere vise en mer systematisk måte til å angi dette på.

Flere aksjoner eller kommandoer kan knyttes til samme trinn, tegnet i hver sin boks. Innbyrdes arrangering av boksene er uten betydning for rekkefølgen av utførelse. Her vises en form, hvor b) er en forenklet form av a):

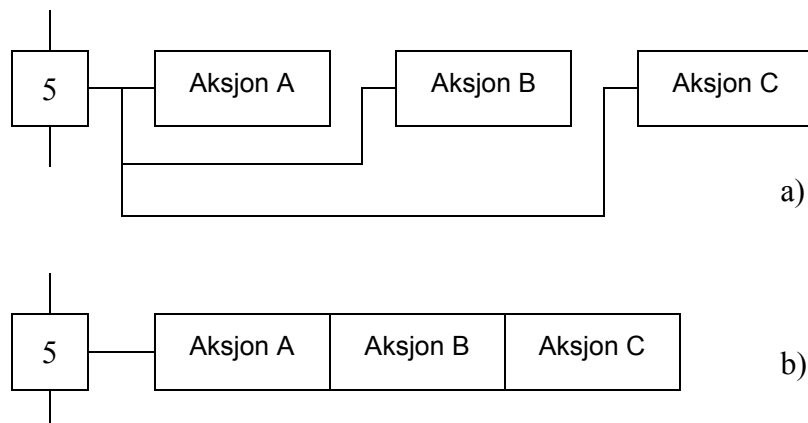


Fig. 42 .

Boksene kan gjerne plasseres under hverandre. Her er også form b) en forenklet tegnemåte for formen a):

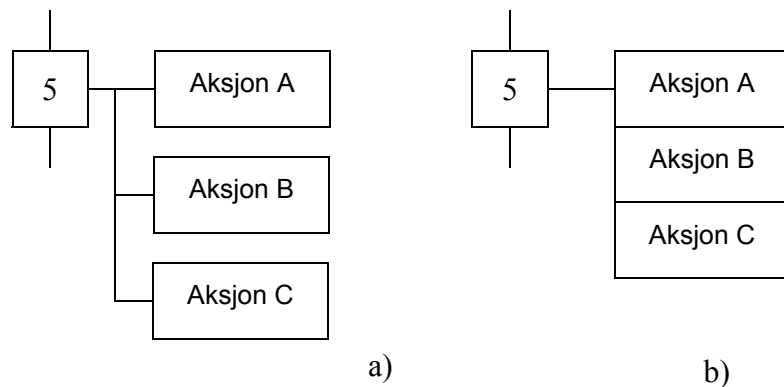


Fig. 43 .

En transisjon foregår langs forbindelseslinjen mellom to trinn og går "normalt" nedover eller mot høyre i figuren. Retningen trenger da ikke å angis med piler. Settes på pilspiss, kan transisjonen gå i hvilken som helst retning i skjemaet. Transisjonen symboliseres ved et lite tverrstrekk på forbindelseslinjen. Ved siden av tverrstreken angis *betingelsen* som fører til transisjonen. Transisjonen foregår *bare* når *både* foregående trinn er *aktivt* og betingelsen (B) er *sann*:

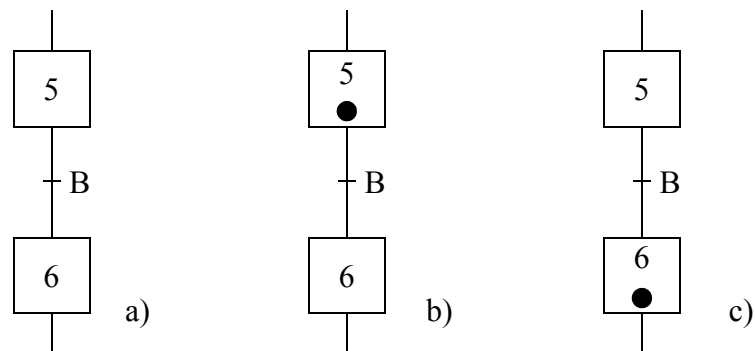
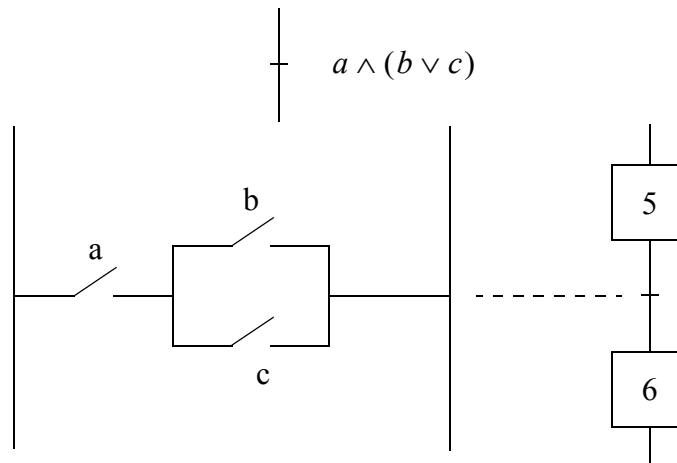


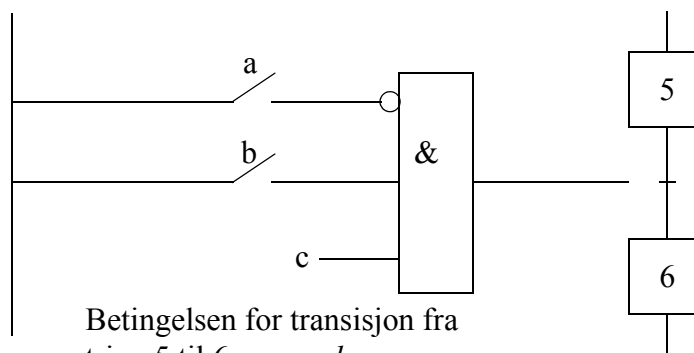
Fig. 44 .

I tilfelle a) foregår ingen transisjon, uansett om B er sann eller ikke, fordi trinn 5 ikke er aktivt. I b) er det klart for transisjon som inntreffer idet B blir sann. Tilfelle c) viser situasjonen etter en transisjon når B var sann, og vi kan betrakte c) som en situasjon som etterfølger b).

Betingelser angis matematisk, som boolske uttrykk, eller fysisk, som eksempelvis sammenkopling av relekontakter eller elektroniske (logiske) kretselementer, eller som en kombinasjon. Noen eksempler:



Betingelsen for transisjon fra
trinn 5 til 6: $a \wedge (b \vee c)$



Betingelsen for transisjon fra
trinn 5 til 6: $\neg a \wedge b \wedge c$

Fig. 45 .

Vi har naturligvis behov for å uttrykke valg mellom alternative transisjonsveier. I figurene 22 og 31 uttrykkes dette ved at flere transisjonslinjer

kan føre ut fra en tilstand, hver med forskjellig betingelse som utløsende årsak. I den formen vi beskriver her uttrykkes dette litt annerledes: Ut fra selve trinnet fører bare 1 linje, men den forgrener seg til flere, hver med sin egen og forskjellig betingelses-strek på tvers av transisjonslinjen:

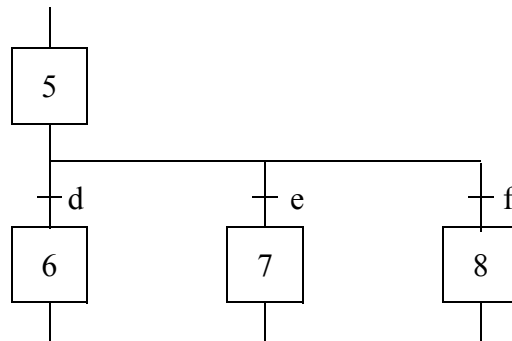


Fig. 46 .

I figuren over er d, e og f tre forskjellige betingelser, som ved oppfyllelse fører til hver sin alternative gren. Det vil være klart at før eller senere må $d \vee e \vee f = 1$, og de må også være *gjensidig utelukkende*.

I likhet med grafformen i figurene 22 og 31 er ett og bare ett trinn *aktivt* i de uttrykksformene som er vist hittil. I Grafcet-formen har vi imidlertid i tillegg også mulighet for å uttrykke *parallelitet*, dvs. to eller flere deler av diagrammet hvor prosessering foregår samtidig. Dette innebærer at flere enn ett trinn er aktivt samtidig. Dette uttrykkes på liknende måte som i figuren over, bare med den forskjell at den horisontale linjen som fører til alternativ forgrening i figuren over erstattes med en dobbel-linje, samt at vi trenger en betingelses-strek ovenfor, rett etter utløpet fra foregående trinn, mens det ikke skal være betingelses-strek mellom den horisontale dobbelt-linjen og de etterfølgende trinn-boksene (se nedenfor). Hvorfor?

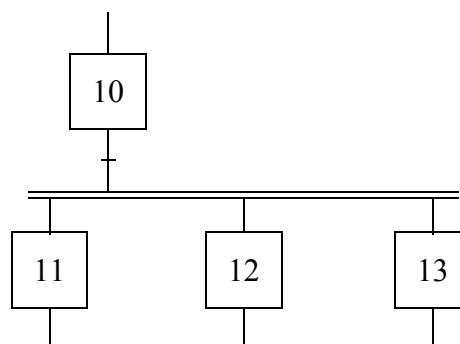


Fig. 47 .

Etter parallelle grener må grenene føres sammen igjen, der parallell operering slutter. Dette uttrykkes også med en horisontal dobbeltlinje. Samling av alternative grener uttrykkes tilsvarende, med horisontal enkeltlinje. Følgende figur er et eksempel som inneholder begge disse typene forgrening, samt samling igjen:

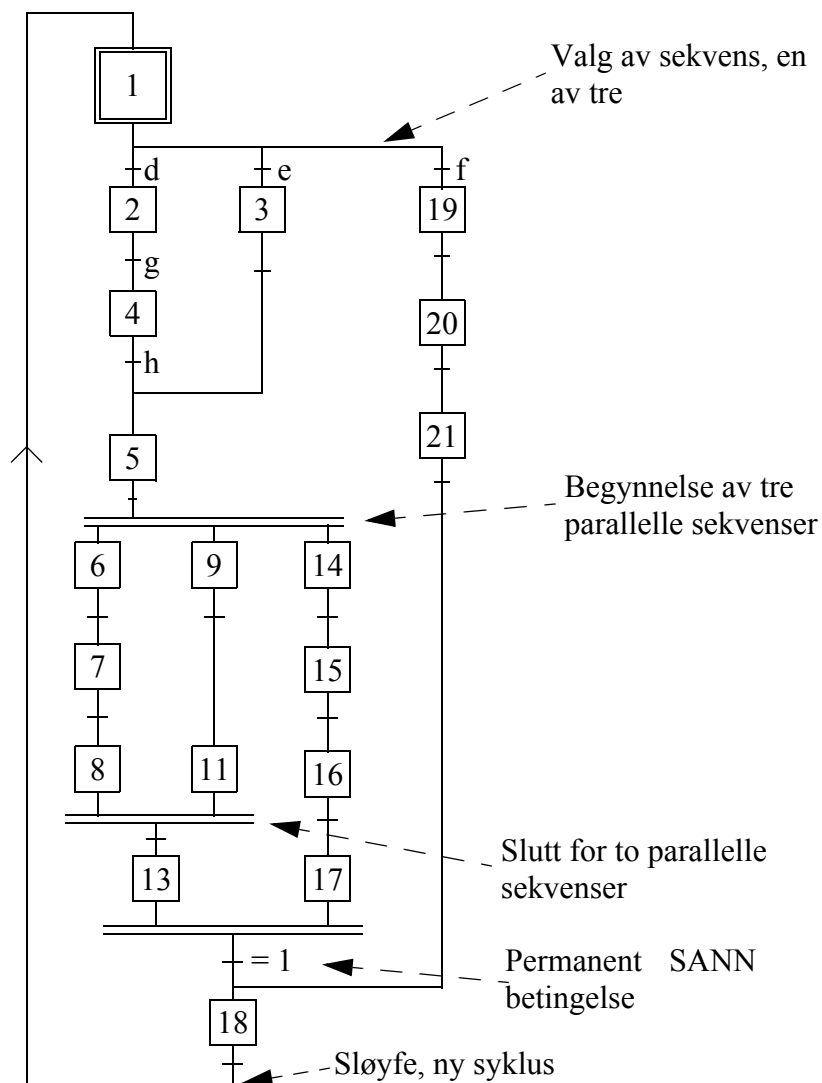


Fig. 48 Eksempel på sekvens-system

1.9.3 Detaljert beskrivelse av kommandoer

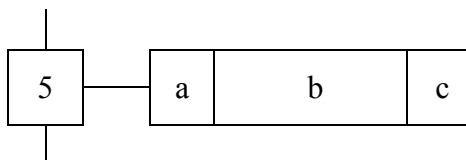


Fig. 49 Trinn med kommandoblokk

En kommando har generelt tre felt innenfor rektangelet. Feltene a og c utelates gjerne hvis de ikke brukes. Felt b skal være minst dobbelt så stort som a og b.

Felt b inneholder selve kommandoen. c er en etikett som kan brukes som referanse. Felt a kan inneholde bokstavsymboler for å vise behandlingen av binære signaler, som forklart nedenfor.

På side 55 ble vist forskjell mellom lagrede og ikke lagrede kommandoer. Varigheten av "ikke-lagret" kommando er forutsatt å være lik lengden av perioden vedkommende trinn er aktivt. I praksis vil mange ikke-lagrede kommandoer være begrenset i tid og kan også være forsinket i forhold til tidspunktet trinnet blir aktivt. Tilsvarende forhold gjelder for lagrede kommandoer.

Den korrekte sammenheng mellom varigheten av kommandoene og varigheten tilhørende trinn er aktivt uttrykkes ved å merke et detaljert kommandosymbol med et bokstavsymbol i felt a (se figur 49) med en av følgende bokstaver:

- S (Stored, lagret)
- R (Resett en lagret aksjon, *kun i IEC 1131-3*)
- D (Delayed, forsinket)
- L (time Limited, tidsbegrenset)
- C (Conditional, betinget)

Hvis L er svært kort, kan L erstattes med

- P (Pulse shaped, pulsformet)

Vi kan også bruke en kombinasjon av bokstavsymboler. Vanlig lese- rekkefølge svarer da til hvordan et binært signal fra vedkommende trinn behandles for å forme den ønskede kommando. Eksempelvis vil symbolet

DSL betyr at det binære signalet blir forsinket, lagret og så gitt en viss varighet. Virkemåten fremgår for øvrig best av en rekke eksempler vist i det følgende.

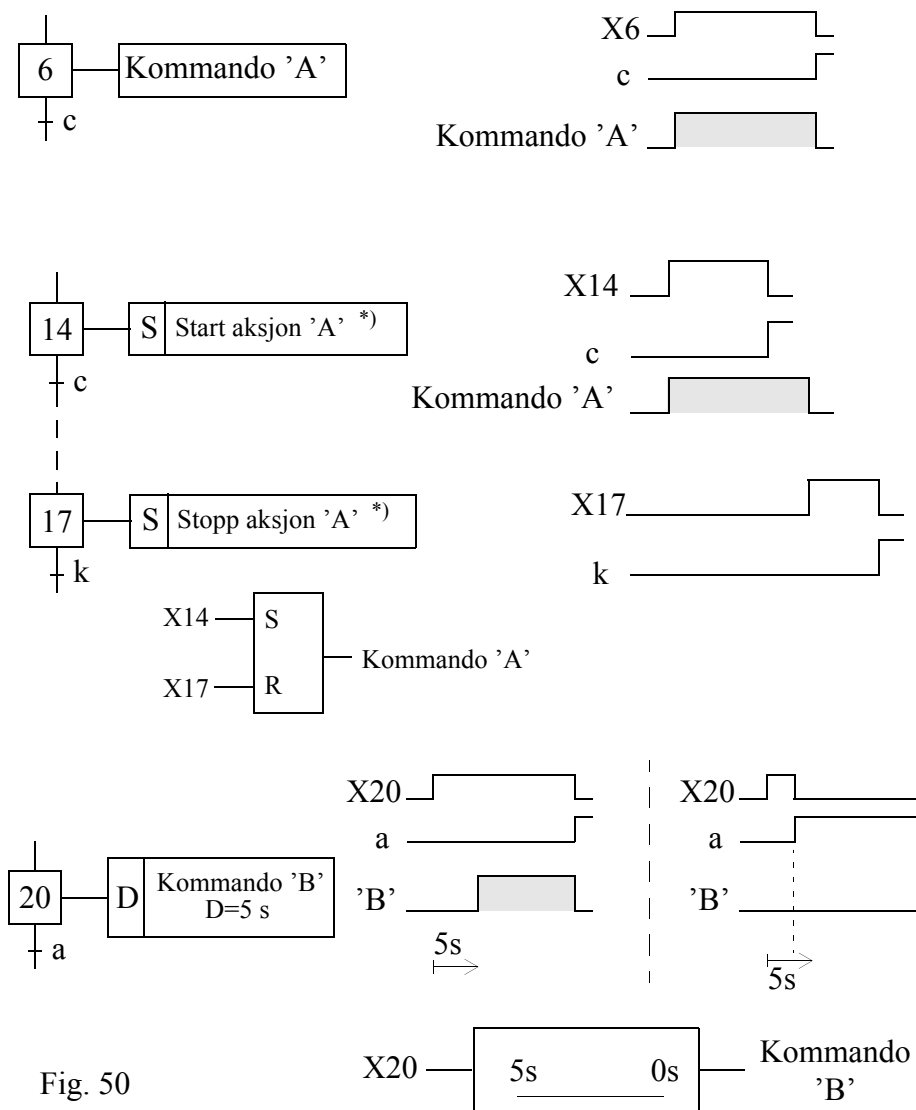
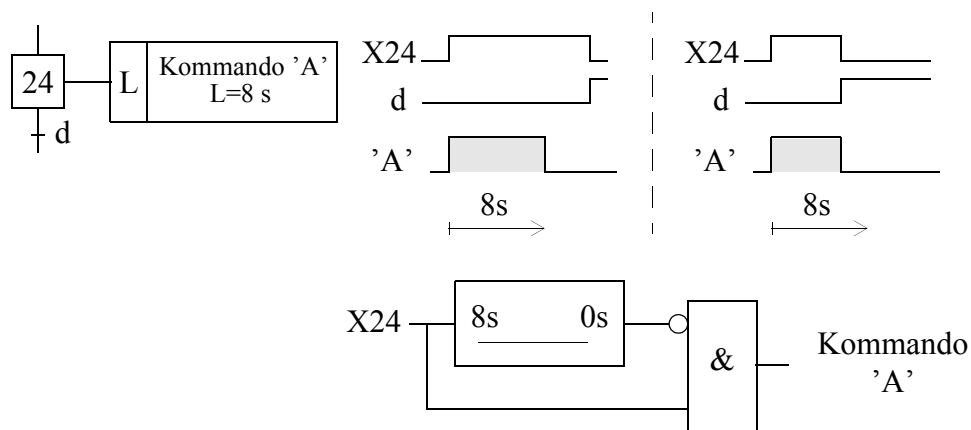


Fig. 50

*) Med IEC1131-3 brukes for trinn 14 og 17 henholdsvis S: Aksjon 'A' og R: Aksjon 'A' for Start og Stopp.

Tidsbegrenset, uten lagring:



Lagret og forsinket aksjon:

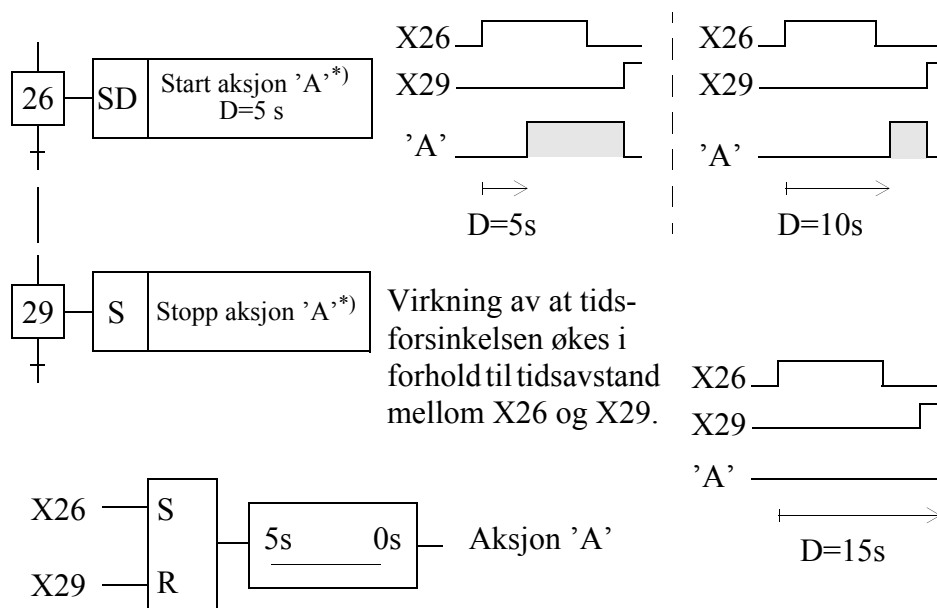
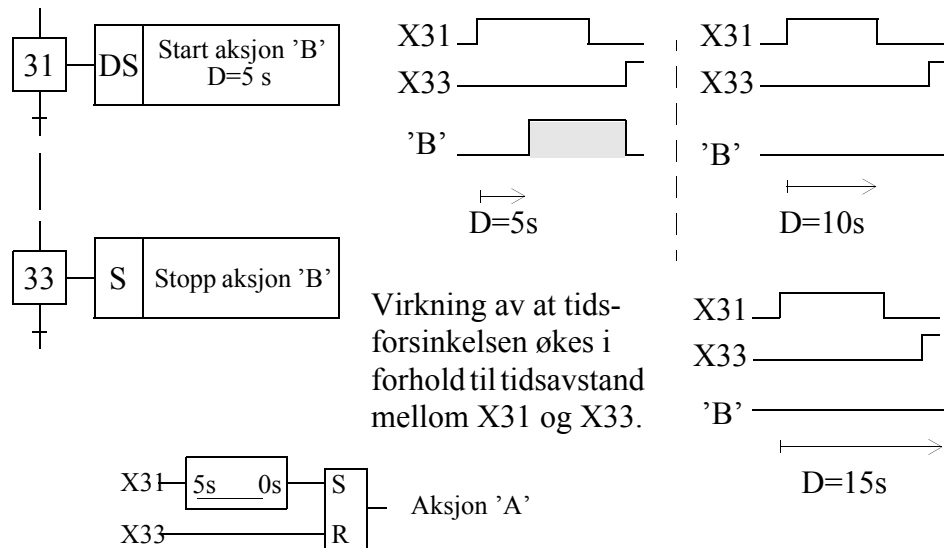


Fig. 51

*) For trinn 26 og 29 gjelder for IEC 1131-3 tilsvarende som nevnt side 61.

Forsinket og lagret aksjon:



Lagret og tidsbegrenset aksjon:

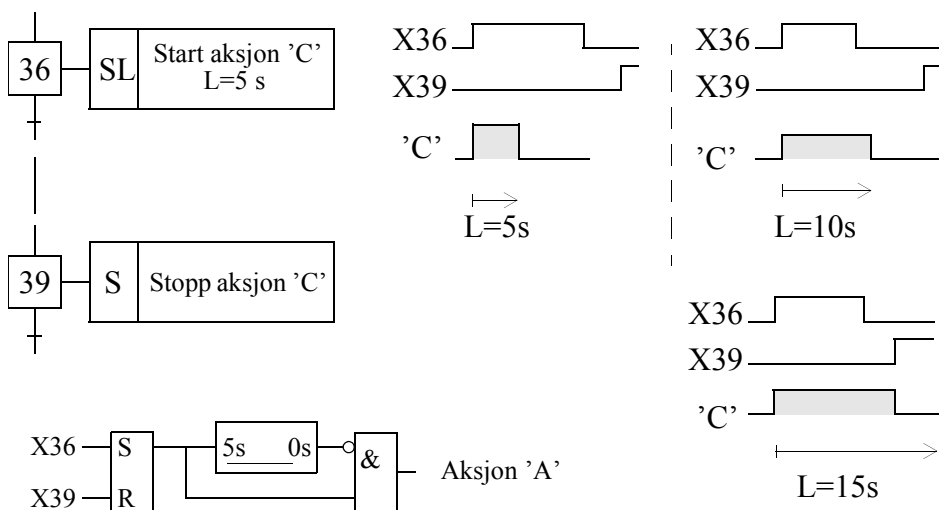


Fig. 52

En kommando kan sendes gjennom en *logisk aktiverende betingelse* før eller etter den angitte prosesseringen av flankesignalet (felt "a" i kommando-symbolet i Fig. 49 på side 60). Dette er særlig nyttig når kommandoen "lagres" (holding). Slik betingelse kan angis innenfor eller utenfor kommando-symbolet, avhengig av tilgjengelig tekstplass. Eksempler:

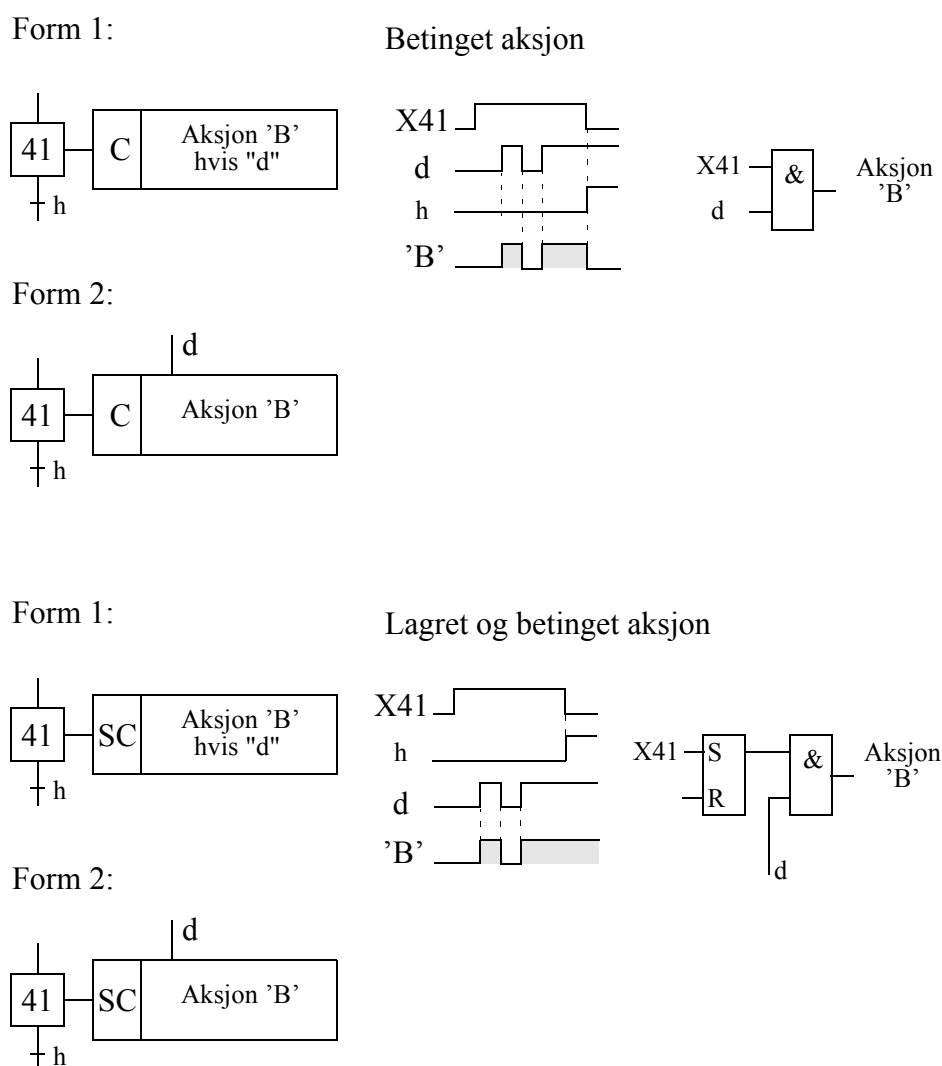


Fig. 53

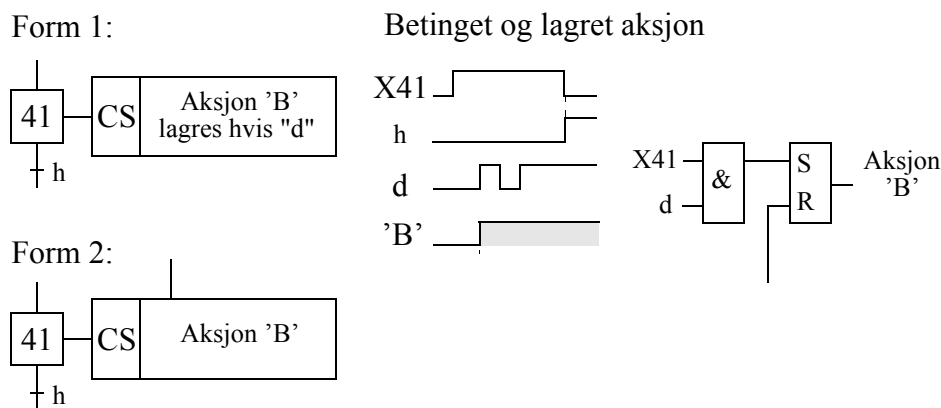


Fig. 54

1.9.4 Detaljert beskrivelse av transisjoner

Som omtalt innledningsvis i avsnitt 1.9.2, side 54, kan transisjoner angis bl.a. med tekst, som boolske uttrykk eller grafiske symboler. Vi skal nå utdype dette nærmere.

Tidsavhengighet

Vi kan ha behov for å uttrykke at en betingelse skal utløses en viss tid etter at en logisk variabel har endret seg. Mer presist: Hvis en logisk variabel **a** endrer seg fra logisk 0 til 1 og en tid etter tilbake til 0, kan dette fremstilles som en firkantpuls. *Virkningen* av denne pulsen kan benyttes som en betingelse, som eventuelt kan være forsinket i forhold til "inngangen" **a**. Forsinkelsen kan være i forkant eller bakkant av pulsen **a**, eller eventuelt ved begge pulsflanker. Dette sees lettest i en figur:

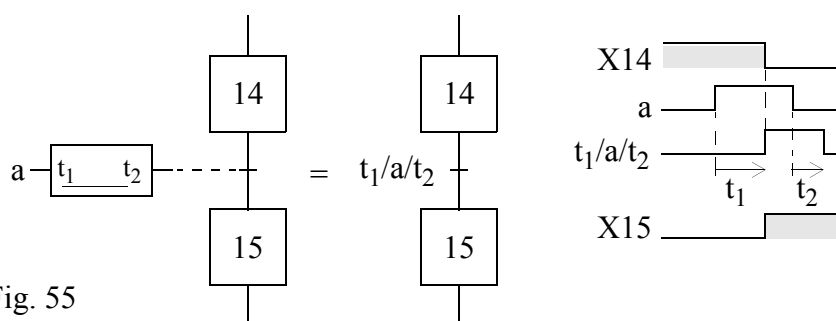


Fig. 55

I figur 55 er vist to alternative måter å angi en forsinket betingelse: enten ved hjelp av et logikksymbol for forsinkelse, eller ved den sammensatte beskrivelse $t_1/a/t_2$. Hvis en av tidsforsinkelsene t_1 eller t_2 er 0, sløyfes leddet. Dvs. hvis $t_1=0$ men $t_2 \neq 0$, skriver vi helst a/t_2 , og hvis bare $t_2=0$, uttrykkes tidsforsinkelsen som t_1/a . Hvis t_1 eller t_2 er konstante tall, skrives tallverdiene, altså ikke som symbolene t_1 og t_2 .

I figur 55 ser vi at $t_1/a/t_2$, dvs. betingelsen for transisjon fra trinn 14 til 15, inntreffer forsinket t_1 etter at a går 0 til 1. Transisjonen innebærer at X14 opphører og X15 begynner, slik det ble forklart i avsnitt 1.9.3.

Helt i overensstemmelse med reglene vist foran kan man eksempelvis angi at varigheten av trinn 27, dvs. Aksjon 'B', skal være 4 sekunder på følgende måte:

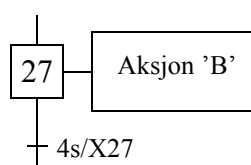


Fig. 56

Notasjon for logiske tilstander og tilstandsoverganger:

Notasjon	Sann ved	Signalform
c	$c=1$	
$\neg c, \bar{c}$	$c=0$	
$\uparrow c$	positiv flanke av c	
$\downarrow c$	negativ flanke av c	

1.9.5 Gjenbruk av sekvens

En sekvens av trinn, som opptrer flere steder, kan representeres hvert sted med ett enkelt trinn-symbol av generell type, som henviser til en detaljert fremstilling av disse trinnene. Eksempel:

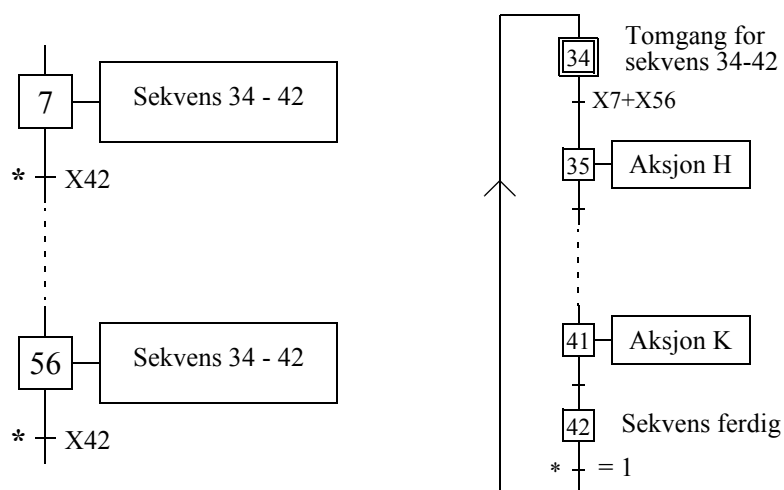


Fig. 57

Stjernesymbolet (*) brukes til å markere transisjoner, plassert i forskjellige grafer, som skal nullsettes samtidig.

Egentlig er dette en subrutine, og fremstillingen som en egen sekvens (aktivitet eller prosess) slik høyre side av figur 57 viser er ikke helt god: Den forutsetter at det er en subrutine, og som sådan starter utføringen idet subrutinen kalles, og utføringen stopper idet det returneres til kallende sekvens. I ovennevnte figur er subrutinen fremstilt som en egen sekvens eller prosess, og hvis dette skal virke korrekt, *må* "sekvensen" stå i ro i trinn 34 når den ikke er kalt. Dessuten er betingelsen ut fra trinn 34 bundet: Her *må* stå den logiske eller-funksjonen av de mulige kallene. Det er med andre ord spesielle bindinger som ikke eksisterer i "normale" sekvenser. Denne måten tas imidlertid med her, fordi den er med i standarden, og den virker helt fint, hvis reglene følges fullt ut. Ikke sett en vilkårlig betingelse etter trinn 34, imidlertid!

1.9.6 Flere nivåer

Et skjema kan gjerne utformes i flere nivåer med forskjellig detaljeringsgrad. Eksempel:

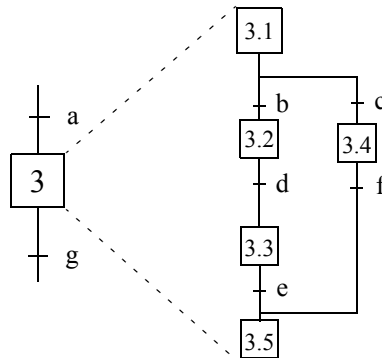


Fig. 58

I figur 58 er sekvensen til høyre en detaljering av trinn 3. Derfor er det også naturlig å la nummeret til hovedtrinnet komme frem i trinn-numrene, altså som under-trinn til trinn 3. For øvrig er denne fremstillingsformen bedre enn den som ble brukt i figur 57: detaljskjemaet kan godt oppfattes som en subrutine og er ikke beheftet med de problemene som det ble advart mot ved figur 57.

1.9.7 Eksempler

Start/stopp av stor maskin

Noen maskiner må startes gjennom en sekvens av enkelt-aksjoner. Dette kan også gjelde stopp. Mange maskintyper kommer under denne kategorien. Noen få typiske eksempler: Store dieselmotorer, elektriske sleperingsmotorer, vannturbiner med elektriske generatorer som må fases inn mot nettet. I de trinnvise oppstart- og stopp-prosedyrene inngår gjerne **forrigling**, som betyr at vi går videre til neste trinn først etter at visse kriterier er "bevist" oppfylt. Kriteriene er ofte bestemt av sikkerhetskrav.

En slik overordnet sekvens kan fremstilles slik:

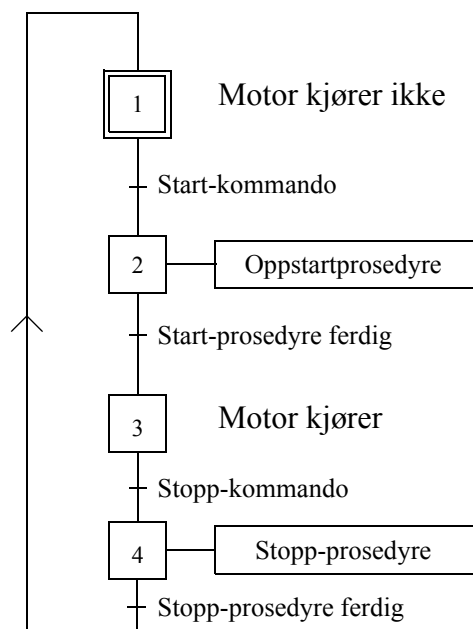


Fig. 59

Hver av aksjonene "Oppstartprosedyre" og "Stopp-prosedyre" kan så videreutvikles mer detaljert, etter samme prinsipp som i figur 58. Standarden IEC-848 viser et eksempel på dette, men vi tar det ikke med her.

Automatisk veiing og blanding

Systemet er vist skjematisk i Fig. 60 på side 70 og består av to pulverbeholdere A og B plassert over en vekt C. Videre et transportbelte hvor det ankommer "briketter", samt en roterende blander. Beholderne A og B er utstyrt med hver sin mateinnretning MA og MB på undersiden, slik at pulver kan slippes ned på vekten i passende hastighet. Brikettene ankommer på transportbåndet en etter en og "tippes" over kanten og ned i blanderen, hvor de blir knust og blandet med pulveret. figur 60 viser disse enhetene.

En operasjonsyklus foregår slik:

Etter trykk på en startknapp starter oppveiing av først pulver A opp til en mengde p_A , deretter oppveies pulver B inntil vekten viser p_B . Etter dette

tømmes vekt-innholdet ned i blanderen ved hjelp av materen MC. Samtidig med at oppveiingen begynner, starter transportbåndet og en spesiell mater som slipper briketter ned på båndet. Hastighetsforskjellen mellom transportbåndet og brikettmateren er slik at brikettene blir liggende adskilt, en og en.

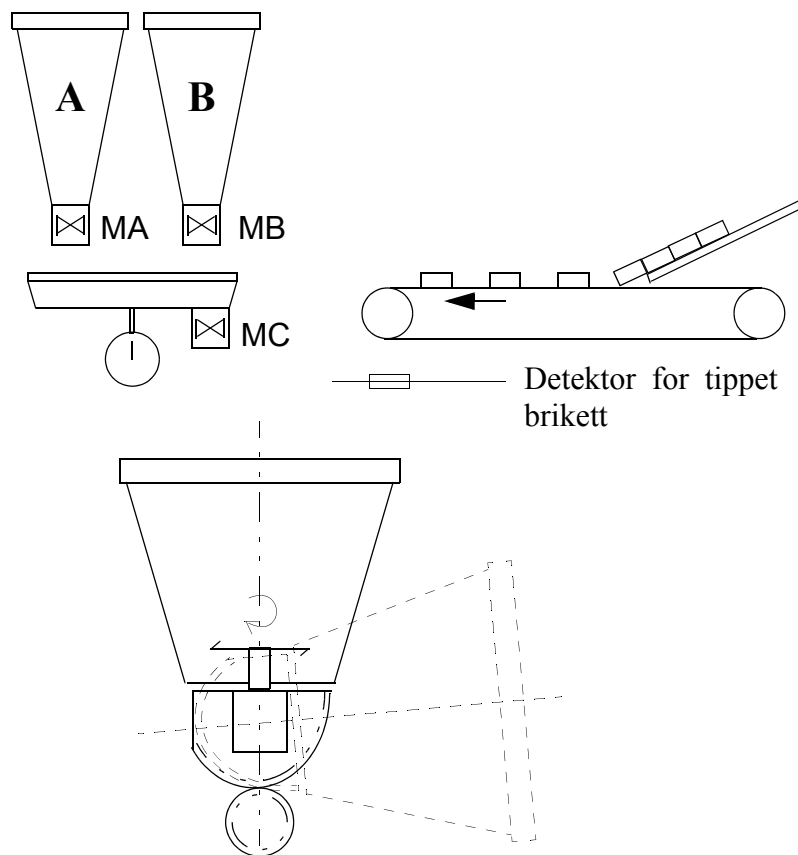


Fig. 60

Idet en brikett tipper ned i blanderen registreres dette av en detektor. Etter at foreskrevet antall briketter og oppmålt pulver er kommet ned i blanderen starter denne. Etter en viss blandetid tippes blanderen som vist stiplet, slik at den tømmes, og etter en viss tid i tippet stilling anses blanderen tom. Da stoppes motoren som roterer blanderen, og blanderen reises til opprettet

stilling. Når blanderen er kommet opp, stoppes syklusen.

Dette kan fremstilles som i figur 61:

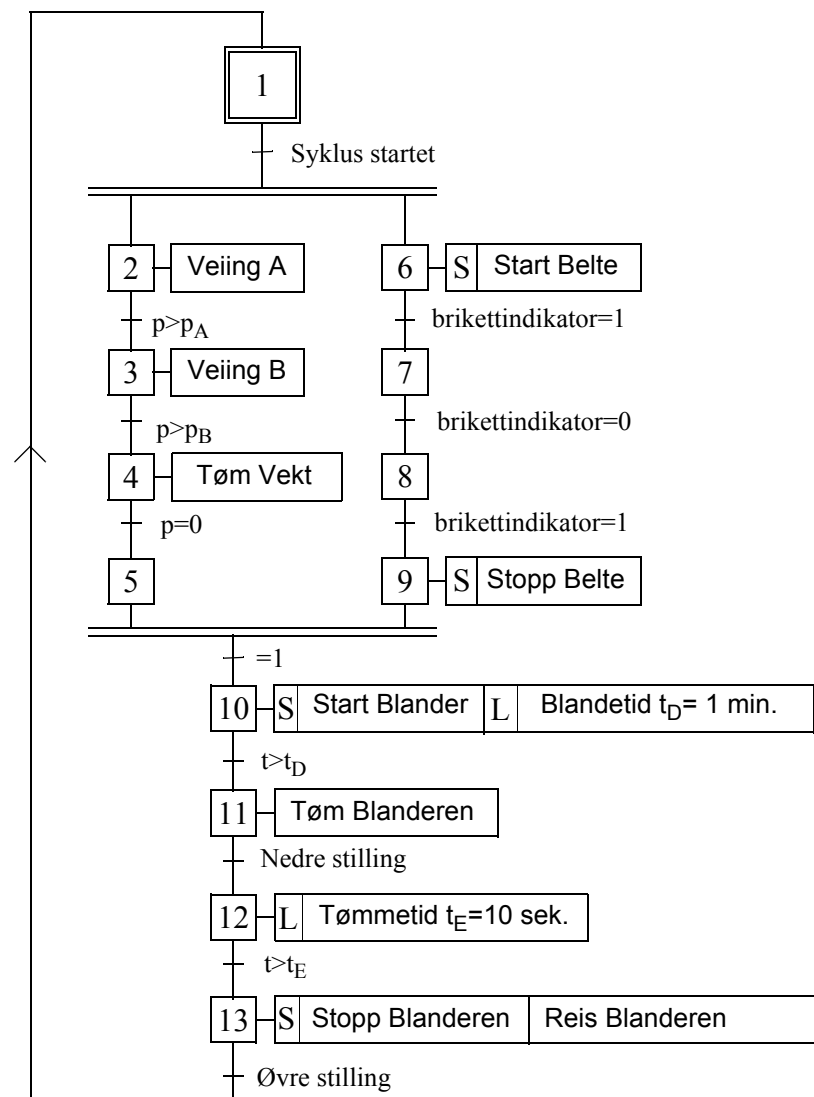


Fig. 61 Veie- og blandeopprosess

1.10 Metoder og utstyr for realisering av logikkstyring

1.10.1 Oversikt

Logikkstyring kan realiseres med én av to forskjellige prinsipper:

- Trådbundet, og
- Programmert eller programmerbart

I “gamle dager” var alle styresystemer “trådbundet”. Det vil si at logikksystemet var oppbygget basert på et antall logiske elementer som var fast forbundet, på en slik måte at selve forbindelsene var bestemmende for den logiske oppførsel av systemet. Et enkelt eksempel på dette er sammenknytning mellom noen enkle logiske funksjonsblokker:

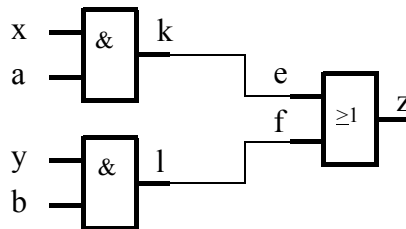


Fig. 62 . Trådbundet styring med 3 funksjonsblokker

Her har vi tre funksjonsblokker: To “OG”-funksjoner, og én “ELLER”-funksjon. Hvis de sammenkoples som vist, ved at inngangene **e** og **f** til ELLER-blokken koples til utgangene **k** og **l** fra OG-blokkene, oppnås at “utgangsvariabelen” $z = (x \& a) \mid (y \& b)$.

Samlet er dette en *multiplekser*: Hvis **a** og **b** er to gjensidig utelukkende styresignaler, vil disse velge ut én av inngangssignalene **x** eller **y** og presentere den valgte ut på utgangen **z**. Sammen med de elementære OG- og ELLER-blokkene er ledningene fra **k** til **e** og fra **l** til **f** bestemmende for den samlede oppførselen av dette “systemet”: Multiplekserfunksjonen **z** er fast og “trådbundet”, den er knyttet til de fast oppkoblede ledningstrådene. Dette prinsippet kan utbygges til store systemer, og dette var tidligere eneste måten man hadde mulighet for.

Det store problemet med slike løsninger var at endringer var svært komplisert da det medførte forandring av kabelopplegg, og i den sammenheng vanligvis også ombygging av elementære funksjonsblokker.

Denne store begrensningen ble man ikke kvitt før man kunne bruke *programmerbare* innretninger, og denne mulighet fikk man først da man tok i bruk programmerbare prosessorer, dvs. datamaskin-teknologi, og dette begynte å bli vanlig fra midten av 1960-årene av. Fremdeles, etter overgangen til nytt århundre, benyttes trådbundet styring, men nå bare i helt spesielle tilfeller, til realisering av små, begrensede, faste og veldefinerte styringsbehov. Ett av de siste områdene som tok i bruk programmerbar styring var til biler, men her benyttes fremdeles trådbundet styring i stor utstrekning.

En type systemer som fremdeles er basert på trådbundet styring, og som vel alltid vil være det, er datamaskiner internt: En datamaskin er realisert med kretskort, som hvert består av et antall integrerte kretser og andre elektronikkomponenter, fast sammenkoplet gjennom kretskortmønsteret. Kretskortet kan bare i begrenset grad endres til ny oppførsel. Det benyttes altså fremdeles trådbundet teknologi, men nå helst begrenset til lavere abstraksjonsnivå, til å realisere komplekse byggelementer som er laget programmerbare slik at de tilsammen utgjør et programmerbart styresystem.

Logikkfunksjons-elementer

I figur 62 foran ble brukt to typer byggelementer uten nærmere presentasjon: OG og ELLER. Det er lett og naturlig å tenke seg disse som elementære integrertkrets-brikker, eller deler av slike. Før vi fikk integrerte kretser, eller i alle fall før halvlederteknologien kom i vanlig bruk, var releer den eneste komponenttype man hadde som byggelementer til logiske styringer. Releer blir ikke nå lengre mye brukt til realisering av logikkfunksjoner. Vi skal likevel ta med en oversikt over denne komponenttype, av to grunner:

- Releer blir fremdeles brukt til svært enkle formål, der digitale styringsenheter ville være "å skyte spurver med kanoner", samt til å styre store strømstyrker, som eksempelvis å slå på lyskasterne på en bil, styrt av en liten bryter ved førersetet over tynne kabler, eller motorer. I sistnevnte tilfelle kalles releet gjerne en *kontaktor*.
- Releer danner konseptuell basis for en måte å programmere PLS² på, s.k. stigediagrammer. Se Fig. 66 på side 77 og omtale side 81.

2. PLS = Programmerbar Logisk Styring: se forklaring side 76.

1.10.2 Elektromekaniske releer

Et elektromekanisk rele består av en elektromagnet, et jernanker som kan tiltrekkes av elektromagneten, og av ett eller flere sett kontaktfjærer som legges om når ankeret opereres av elektromagneten. Prinsipp og skjematisk:

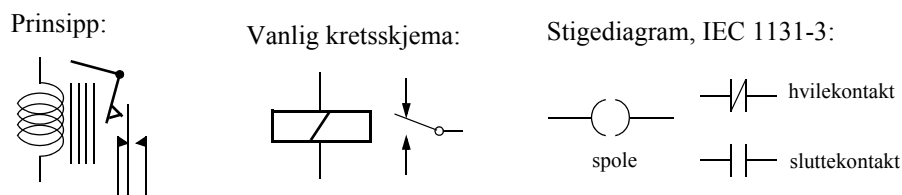


Fig. 63 Elektromekanisk rele

Et lite utvalg releer:

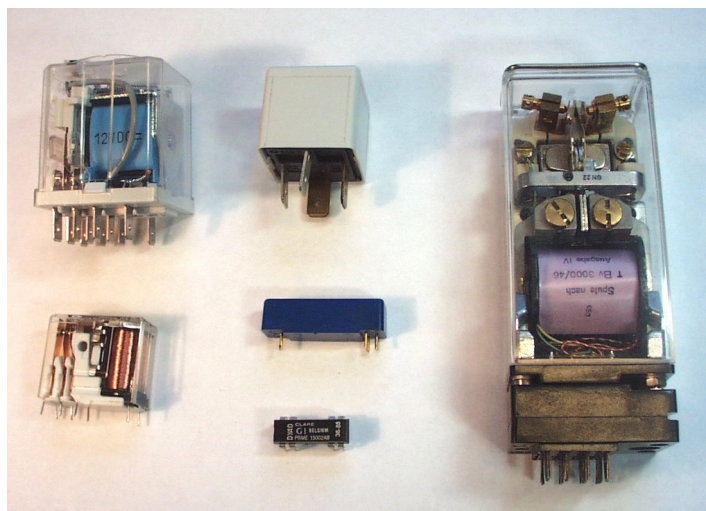


Fig. 64 Noen eksempler på releer

Venstre kolonne, ovenfra: “General purpose”, Tre venderfjærsett med kontakter som tåler 10A ved 250 V AC. Spole: 12 V DC.

Miniatyr, for montering på kretskort: To venderfjærsett 5 A 250 V AC, Spole 24 V DC.

Midtre kolonne: Enkelt rele med ett slutte-fjærsett. Spole: 24 V DC.
 Reedrele for kretskort (fra ca. 1980). Enkel sluttekontakt.
 Reedrele for kretskort (fra slutten av 1990-tallet). Enkel sluttekontakt.
 Helt til høyre: Polarisert rele med stor følsomhet. Enkel venderfjærsett.
 (Siemens, konstruert på 1940-tallet).

figur 63 viser to symbolstandarder for relekontakter og -spoler.
 “Hvilekontakt” er en kontakt som danner forbindelse i strømløs tilstand og som bryter forbindelsen når relespolen energiseres. Motsatt for sluttekontakt, hvor forbindelse dannes når releet slår til, energiseres. En venderkontakt er en kombinasjon av en hvilekontakt og en sluttekontakt, og kontaktfjæren som enten danner forbindelse mot hvilefjæren eller slutfjæren kalles venderfjær. Venderkontakten finnes i to utgaver: break-before-make og make-before-break, som vist i figur 65. Fjæren som beveges av ankeret kalles arbeidsfjær, og for både hvilekontakt, sluttekontakt og break-before-make -typen av venderkontakt er det venderfjæren som er arbeidsfjær, mens det er slutfjæren som er arbeidsfjær ved make-before-break.

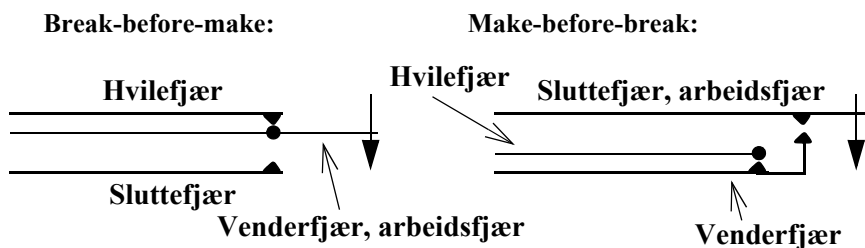


Fig. 65 To venderfjær-prinsipper

Reledigrammer viser **alltid** relekontaktene slik de er når spolen er strømløs.

Man realiserte ganske store systemer basert på releer. Typiske eksempler på slike store systemer som fikk stor utbredelse var de automatiske telefonsentralene. Systemene ble beskrevet og dokumentert med kretsskjemaer, og det var vanskelig å praktisere hierarkisk systembeskrivelse, slik at skjemaene ble store og lite oversiktlige. Likevel var systemkunnskapen stor hos de som arbeidet med slike systemer, inkludert montører og driftspersonell. Disse var vel vant med å lese kretsskjemaer, forstå dem til bunns og i

stand til å gjøre endringer, som da medførte endring av kabling.

I midten av 1960-årene begynte man å ta i bruk datamaskiner til styring av både kontinuerlige og logiske prosesser. Siden dette var helt nytt og man ikke hadde hverken hensiktsmessige programmeringsverktøy eller designverktøy, ble dette rene forskningsprosjekter til å begynne med. Etter at et slikt styresystem var ferdig utviklet og satt i drift, var det fremdeles meget tungvint å endre på, for man måtte da bruke programmerere, og helst de samme som hadde utviklet systemet. Man innså fort at det var behov for metodikk og utstyr som kombinerte datateknikkens potensielle fleksibilitet med mulighet til at montører og prosessingeniører selv raskt kunne gjøre alle nødvendige endringer, slik som de hadde kunnet gjøre med relebaserte systemer. Dette ga på slutten av 1960-tallet opphav til PLC, Programmable Logical Controller, eller på norsk PLS, Programmerbar Logisk Styring.

Siden det var sterkt ønskelig at montører og driftspersonell selv skulle kunne gjøre endringer i et system, ble det utviklet programmeringsverktøy hvor logikken ble uttrykt ved hjelp av rele-kretsskjemaer, selv om realiseringen var med elektroniske blokker og programmerbare prosessorer. Man *simulerer* altså relekoplingen. Et slikt skjema består av mange relekontakter i sammenkopling slik at de tilstrebe logiske operasjoner oppnås. For eksempel vil to seriekoblede sluttekontakter kunne gi en sluttet strømkrets hvis begge kontaktene er operert, dvs. de danner en OG-funksjon. En parallellkopling av to kontakter skaper tilsvarende en ELLER-funksjon. Hvis en slik sammenkopling innkopler strøm til en relespole, vil releet operere og bevirke videre koplinger. Den ene enden av en slik seriekopling må da tilknyttes en spenningskilde, og den andre til kraftforsynings 0-punkt. Når vi så har en rekke slike kretser, koplet mellom den samme spenningskildens poler, blir dette av utseende som en "stige", som illustrert i figur 66. Se også Fig. 67 på side 78, som viser et komplett styresystem, av moderat størrelse, og hvor logikkfunksjonene er realisert med en blanding av transistorer og releer.

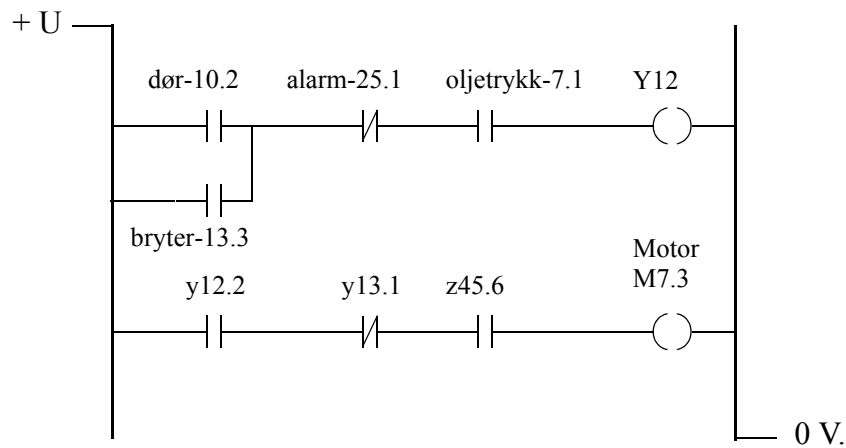
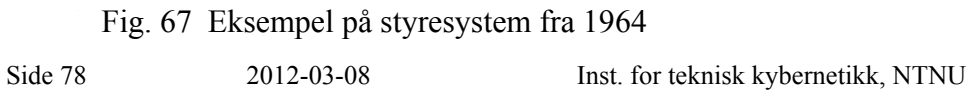


Fig. 66 Eksempel på stigediagram

En grunnleggende vanskelighet, som oftest blir oversett, er forskjellen mellom parallelle, logisk kombinatoriske problemer og metoder på den ene side, og sekvensielle, tilstands-betingede problemer og metoder på den andre. Fordi denne forskjell sjelden blir berørt, vil vi behandle dette spesielt i kapittelet Parallele og sekvensielle funksjoner, side 79.



1.10.3 Parallelle og sekvensielle funksjoner

Releer og logikkmoduler er naturlig parallelle, distribuerte. Releer og deres kontakter ett sted i et system kan gjerne operere samtidig med releer andre steder i systemet. En prosessor, derimot, gjør jobben ved å utføre en rekke enkeltoperasjoner, én etter en. Dette har som konsekvens at det er enklest hvis metoden direkte tilsvarer problemets art:

- Releer og logikkmoduler samsvarer med parallelle og kombinatoriske problemer, og realiseringen av disse problemer blir ganske direkte. Det gir derimot problemer å realisere sekvenser og tilstandsbetingede funksjoner. Disse må realiseres med “simulering” av sekvenser, enten ved tilbakekoplinger eller ved at kun en liten del av koplingen er “aktiv” til enhver tid. Den “aktive” delen forflyttes gjennom systemet, mens resten av systemet er i en “ventetilstand”. Et enkelt eksempel på dette blir vist senere.
- Prosessorer, datamaskiner, utfører jobben sekvensielt og bearbeider problemet som en algoritme som består av en rekke enkelt-trinn. Etter sin natur er det derfor i prinsippet enkelt å realisere sekvens-problemer. Parallelle systemer, derimot, må simuleres i datasystemet. Et typisk eksempel er sanntids operativsystem, som utnytter prosessorens høye hastighet til å betjene en rekke “prosesser” som sett utenfra arbeider i parallell.

Når vi i det etterfølgende skal ta for oss forskjellige programmeringsmetoder for PLS'er, vil jeg henvise til disse grunnleggende forskjeller og påpeke vansker som følger av dem.

1.10.4 Standarden IEC 1131-3³

Standarden IEC 1131, Programmable Controllers, fra IEC (International Electrotechnical Commission) kom i begynnelsen av 1990-årene og er den mest autoritative samlingen av regler og metoder for programmering og

3. God oversikt, mange eksempler og sunn vurdering finnes i flg. bok, som er brukt ved utarbeidelsen av dette kapittelet:

R.W. Lewis: Programming industrial control systems using IEC 1131-3.
IEC Control Engineering series, vol. 50, 1995. ISBN 0-85296-827-2

bruk av PLS. Selv om ikke alle PLS-fabrikat følger standarden helt, så er IEC 1131 den standarden som alle PLS bør vurderes mot. Før IEC 1131 kom, hadde alle som laget PLS sin egen måte å programmere og å bruke dem på, og det var spesielt problematisk å knytte et bestemt fabrikat sammen med utstyr levert av andre, enten det nå var andre PLS, eller sensorer, pådragsorganer etc. IEC 1131 tar sikte mot et felles rammeverk som skal lette slike problemer.

IEC 1131 er inndelt i fem deler:

- Part 1: General information
- Part 2: Equipment requirements and tests
- Part 3: Programming languages
- Part 4: User guidelines
- Part 5: Messaging service specification

Her vil vi bare ta for oss del 3, som altså omhandler programmerings-"språk", dvs. metoder for hvordan en PLS programmeres. Første revisjon av IEC 1131-3 ble publisert i 1993 og er gjeldende versjon.

IEC 1131-3 omfatter fem forskjellige programmerings-"språk", hvorav to er tekstbasert og tre er grafiske:

- Strukturert tekst (Structured text)
- Funksjonsblokkdiagram (Function Block Diagram)
- Stigediagram (Ladder diagram)
- Sekvensielle funksjonsdiagram (Sequential Function Chart)
- Instruksjonslister (Instruction List)

Strukturert tekst (ST)

Dette er et høynivåspråk med syntaks som likner på det ordinære programmeringsspråket PASCAL men er også påvirket av ADA. Et eksempel:

```
IF SPEED1 > 100.0 THEN
    Flow_Rate := 50.0 + Offset_A1;
ELSE
    Flow_Rate := 100.0;
    Steam := ON;
END_IF;
```

Funksjonsblokk-diagram (FBD)

Et grafisk språk for avbildning av signal- og dataflyt gjennom funksjonsblokker, som skal oppfattes som gjenbrukbare programelement:

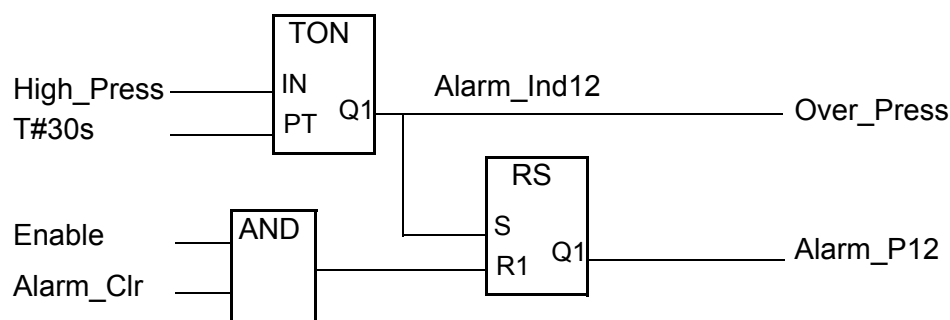


Fig. 68 Eksempel på FBD

Stigediagram (LD)

Dette er et grafisk språk basert på stigediagram for releer, som beskrevet foran, blant annet rundt Fig. 66 på side 77. Releer kan fritt suppleres med logikk-funksjonselementer:

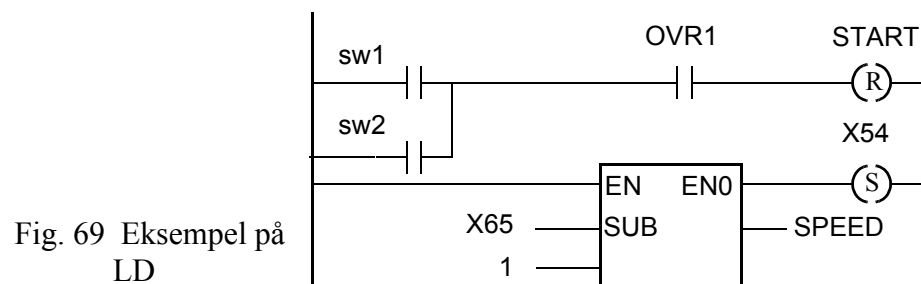


Fig. 69 Eksempel på LD

Instruksjonsliste (IL)

Et lav-nivå språk som likner på assembly, og som er typisk for mange liknende språk som finnes i eksisterende PLS'er. Eksempel:

```
LD    R1
JMPC  RESET
LD    PRESS_1
ST    MAX_PRESS
RESET: LD    0
      ST    A_X43
```

Sekvensielle funksjonsdiagram (SFC)

Dette grafiske språket er omtalt flere ganger allerede; det kan ses som en videreutvikling fra IEC 848 eller Grafcet. Det er velegnet for definering av tids- og hendelsesdrevne styresekvenser:

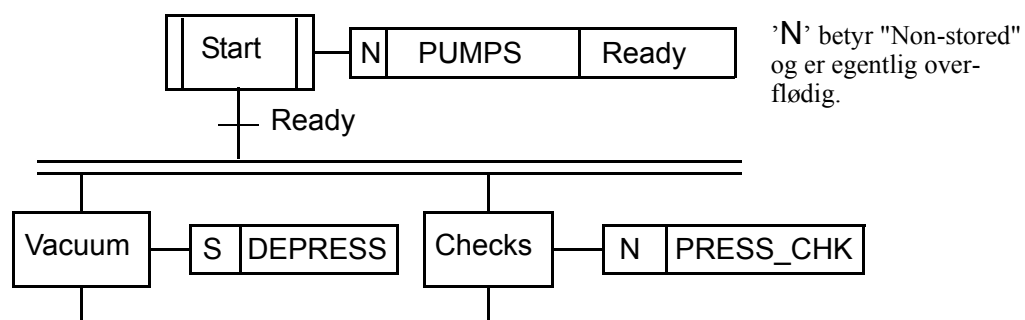


Fig. 70 Eksempel på SFC

Se *Lewis* (fotnote side 79) for mer detaljert beskrivelse.

Vurdering av de fem metodene

Ved vurdering av hvilken metode og språk som egner seg for et gitt tilfelle, bør man ha klart for seg forskjellen mellom parallelle og sekvensielle funksjoner, som behandlet i et eget avsnitt foran, på side 79. Der nest er abstraksjonsnivået viktig. Hvis man velger en uhensiktsmessig metode eller språk, blir det en dårlig løsning.

To av språkene er tekstbasert: Strukturert tekst (ST) og Instruksjon-

sliste (IL). Begge er sekvensielle, så de egner seg bare for sekvensielle funksjoner, dvs. systemer som må beskrives av tilstander. Programmeringen blir som konvensjonell sekvensiell datamaskinprogrammering, og man har intet uttrykksmiddel for parallell operasjon. IL kan best sammenliknes med assemblykode og er på lavt abstraksjonsnivå. Begge språkene kan lett realisere sub-funksjoner (subrutiner).

De to språkene Funksjonsblokk-diagram (FBD) og Stigediagram (SD) er begge grafiske og parallelle. Begge egner seg best til beskrivelse av systemer som skal realiseres direkte i hardware, dvs. FBD som elektronikk på kretskort og SD med fysiske releer. Brukt for PLS, kan de også være hensiktsmessige til detaljer i et sekvensielt funksjonsdiagram (SFC), til beskrivelse av dannelsen av *betingelser* i et SFC. Siden de kan blandes, kan disse to språkene like greit behandles under ett (se bemerkning under avsnitt Stigediagram (LD), side 81).

Som hovedspråk til programmering av PLS er disse to språkene lite egnet. Her er noen avveininger:

SD har egentlig bare én fordel:

- Det er kjent for montører og har intuitiv virkemåte for små systemer.

Listen over ulemper er lang:

- Det er vanskelig å lage velstrukturerte program p.g.a. begrenset mulighet til subrutiner, programblokker og objekter:
 - Vanskelig å oppbygge program hierarkisk
 - Vanskelig gjenbruk
 - Begrensede muligheter til overføring av parametre mellom programblokker
- En del i stigediagram kan lett lese og sette kontakter og utganger hvor som helst i diagrammet. Dette vanskeliggjør innkapsling av data, som er viktig for god programkvalitet og vedlikeholdbar programvare.
- Problemer ved programutvikling med flere programmerere.
- Programmeringsenheter viser grafisk bare lite utsnitt av stigediagrammet. Det blir som å programmere gjennom et kikk-hull.
- Eksisterende PLS bruker lite standardiserte symboler og virkemidler.
- Dårlige muligheter til adressering og manipulering av datastrukturer.

- Dårlig datastruktur, tungvinte aritmetiske operasjoner.
- Tungvint realisering av sekvenser. Dermed er det begrensede muligheter til å bygge komplekse sekvenser.
- Liten styring med program-eksekvering.

Mange av problemene over kan omgås med diverse knep og design av PLS, men de kan aldri elimineres.

Her er en figur som viser hvordan sekvensforløp kan realiseres med releer. Som vi ser, er ikke sekvensforløpet særlig iøynefallende:

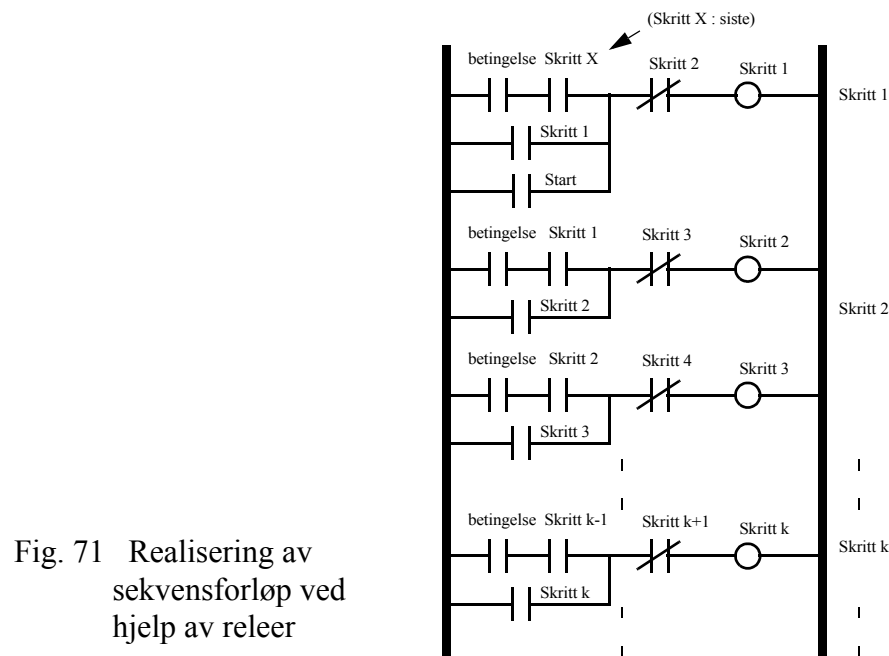


Fig. 71 Realisering av sekvensforløp ved hjelp av releer

Med alle begrensninger og innvendinger mot de fire nevnte språkene, står vi igjen med ett: **Sekvensielle funksjonsdiagram (SFC)**. Dette er egentlig det eneste som i dag kan anbefales som **generelt programmeringsspråk for PLS**. Det er grafisk og kan brukes hierarkisk slik at det er tjenlig både til oversikt i et stort system, og til detaljnivå. Det er både sekvensielt og parallelt, idet man kan legge inn parallelle deler etter behov. Funksjonsblokker og stigediagram kan gjerne benyttes til *detaljer* innen et SFC, spesielt for å uttrykke *betingelser*.