INF1400 – Kap4rest

Kombinatorisk Logikk

Hovedpunkter

- Komparator
- Dekoder/enkoder
- MUX/DEMUX
- Kombinert adder/subtraktor
- ALU
- FIFO
- Stack
- En minimal RISC - CPU

Komparator

Komparator – sammenligner to tall A og B

- 3 utganger: A=B, A>B og A<B

Eksempel: 4-bits komparator

Utgang A=B

Slår til hvis $A_0=B_0$ og $A_1=B_1$ og $A_2=B_2$ og $A_3=B_3$

Kan skrives: $(A_0 \oplus B_0)' (A_1 \oplus B_1)' (A_2 \oplus B_2)' (A_3 \oplus B_3)'$

Komparator - eksempel

Utgang $A>B$ slår til hvis:

$(A_3>B_3)$ eller

$(A_2>B_2$ og $A_3=B_3)$ eller

$(A_1>B_1$ og $A_2=B_2$ og $A_3=B_3)$ eller

$(A_0>B_0$ og $A_1=B_1$ og $A_2=B_2$ og $A_3=B_3)$

Kan skrives:

$$(A_3B_3') + (A_2B_2')(A_3 \oplus B_3)' + (A_1B_1')(A_2 \oplus B_2)'(A_3 \oplus B_3)' +$$

$$(A_0B_0')(A_1 \oplus B_1)'(A_2 \oplus B_2)'(A_3 \oplus B_3)'$$

Komparator - eksempel

Utgang $A<B$ slår til hvis:

$(A_3<B_3)$ eller

$(A_2<B_2$ og $A_3=B_3)$ eller

$(A_1<B_1$ og $A_2=B_2$ og $A_3=B_3)$ eller

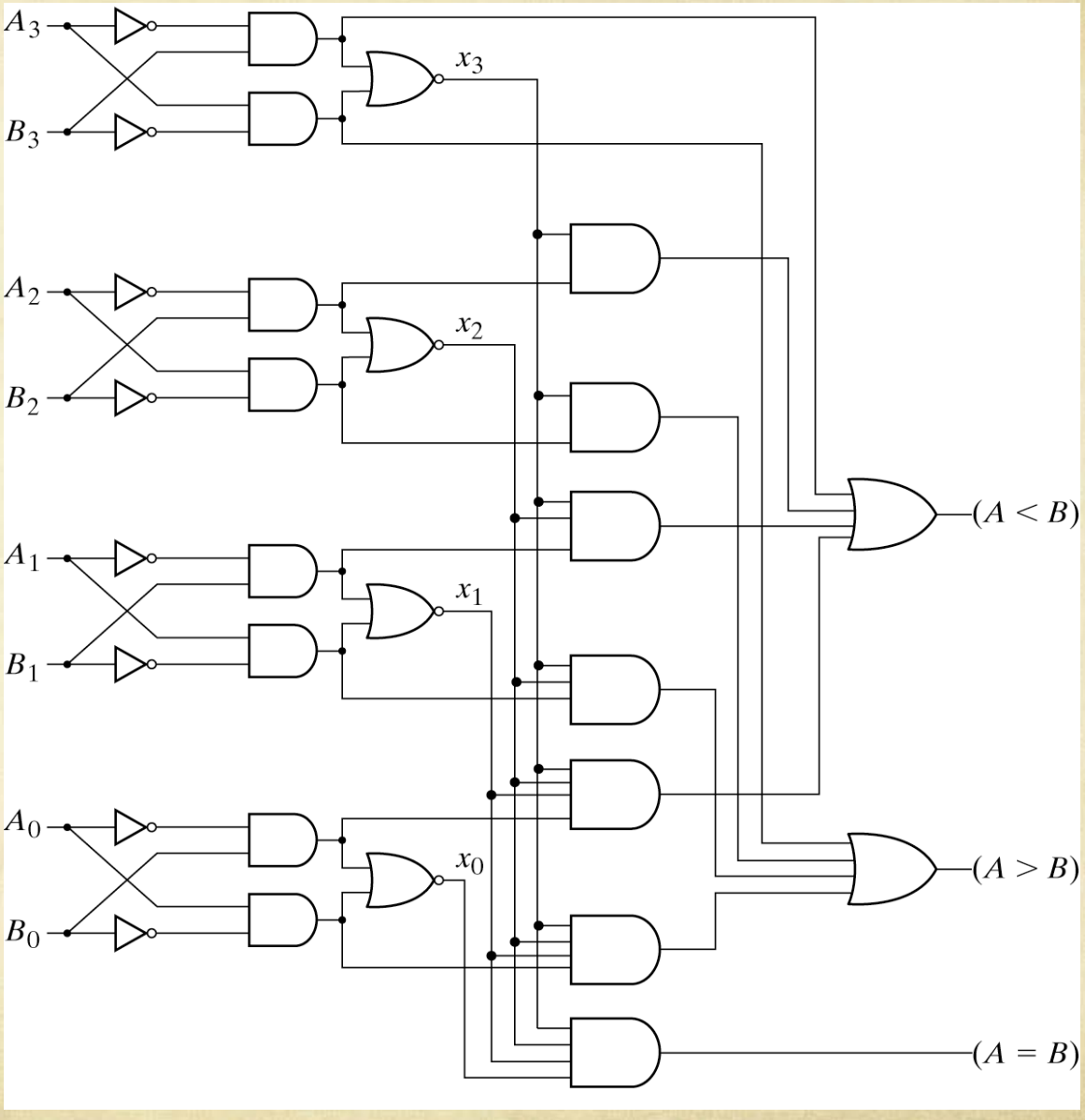
$(A_0<B_0$ og $A_1=B_1$ og $A_2=B_2$ og $A_3=B_3)$

Kan skrives:

$$(A_3´B_3) + (A_2´B_2)(A_3 \oplus B_3)´ + (A_1´B_1)(A_2 \oplus B_2)´(A_3 \oplus B_3)´ +$$

$$(A_0´B_0)(A_1 \oplus B_1)´(A_2 \oplus B_2)´(A_3 \oplus B_3)´$$

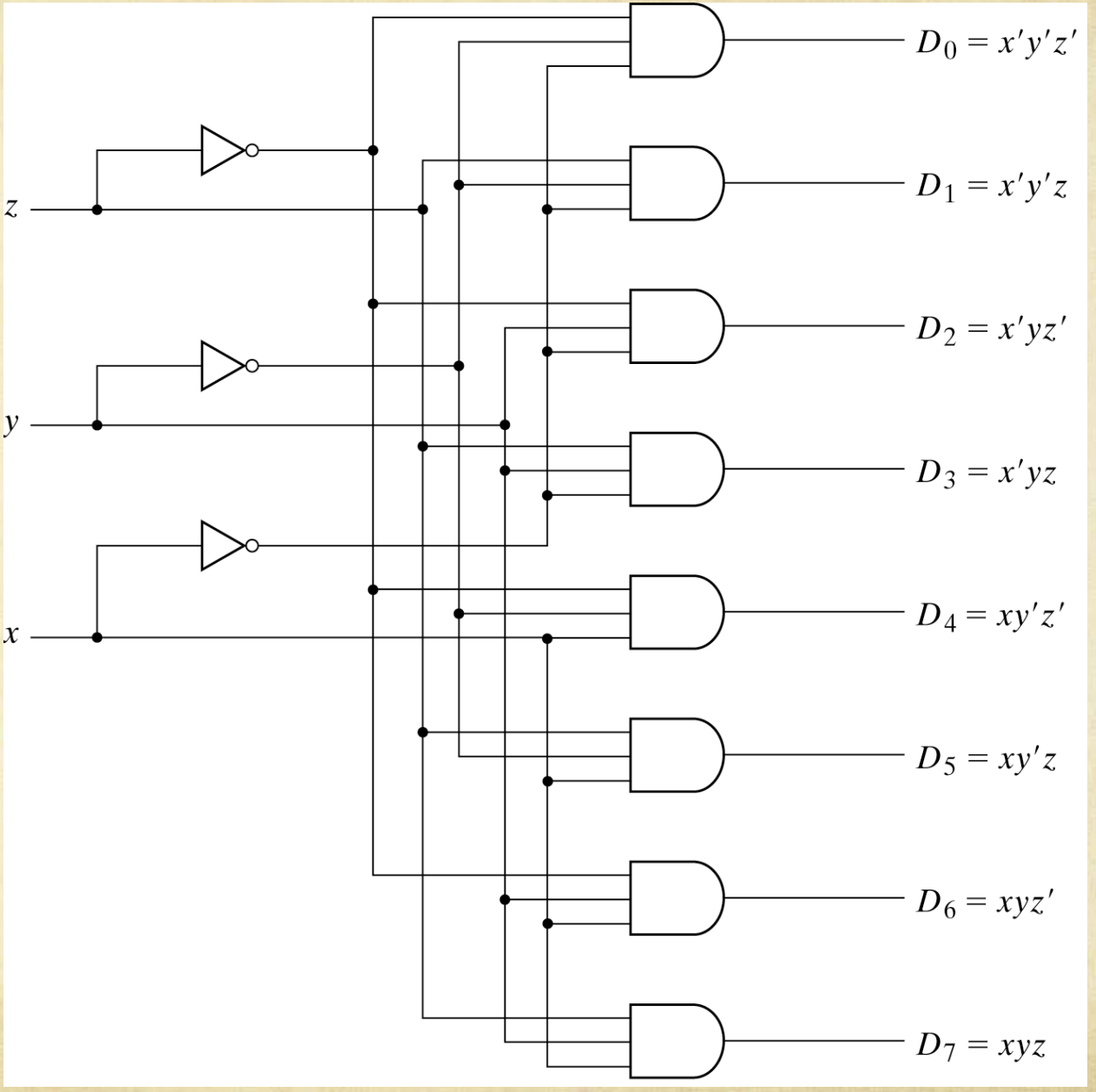
Komparator - eksempel



Dekoder

Dekoder – tar inn et binært ord, gir ut alle mintermer

Eksempel: 3bit inn / 8bit ut



Dekoder - sannhetstabell

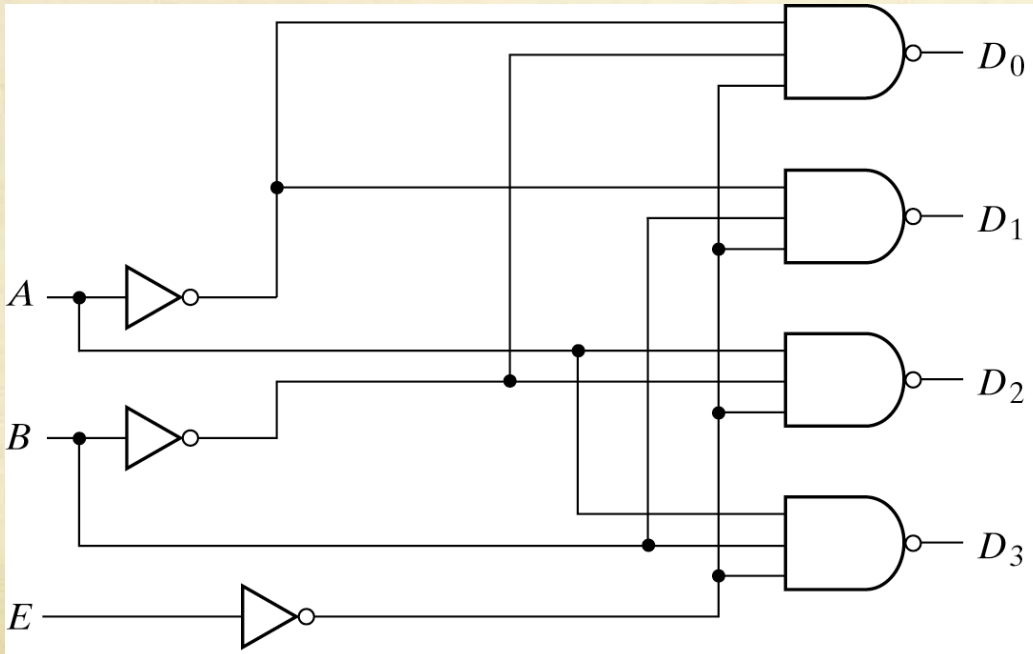
Eksempel: 3bit inn

Innganger			Utganger							
x	y	z	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Dekoder – varianter

Enable input: Enable aktiv - normal operasjon.
Enable inaktiv - alle utganger disablet

NAND logikk: Inverterte utganger



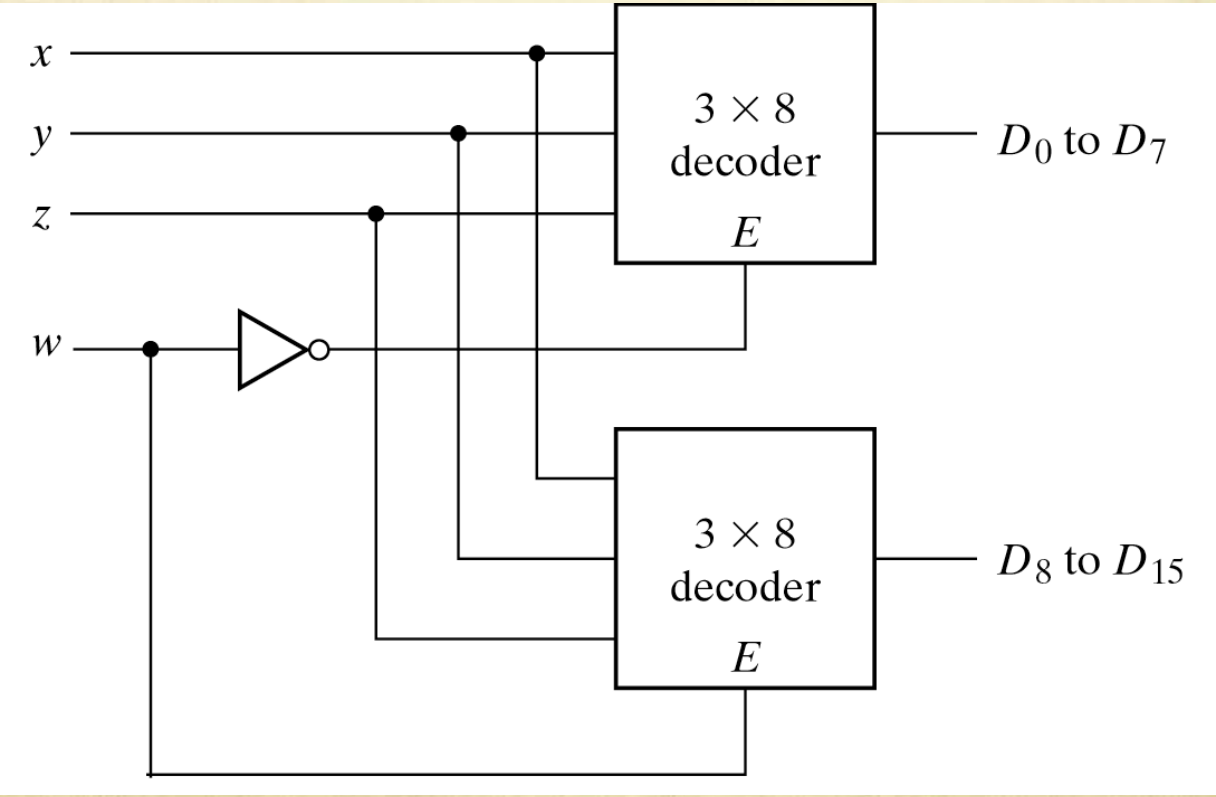
E	A	B	D_0	D_1	D_2	D_3
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

Eksempel

Aktiv ”lav” enable inngang

Dekoder - parallellkobling

Eksempel: Lager en 4x16 dekoder fra 2stk 3x8 dekodere med enable innganger

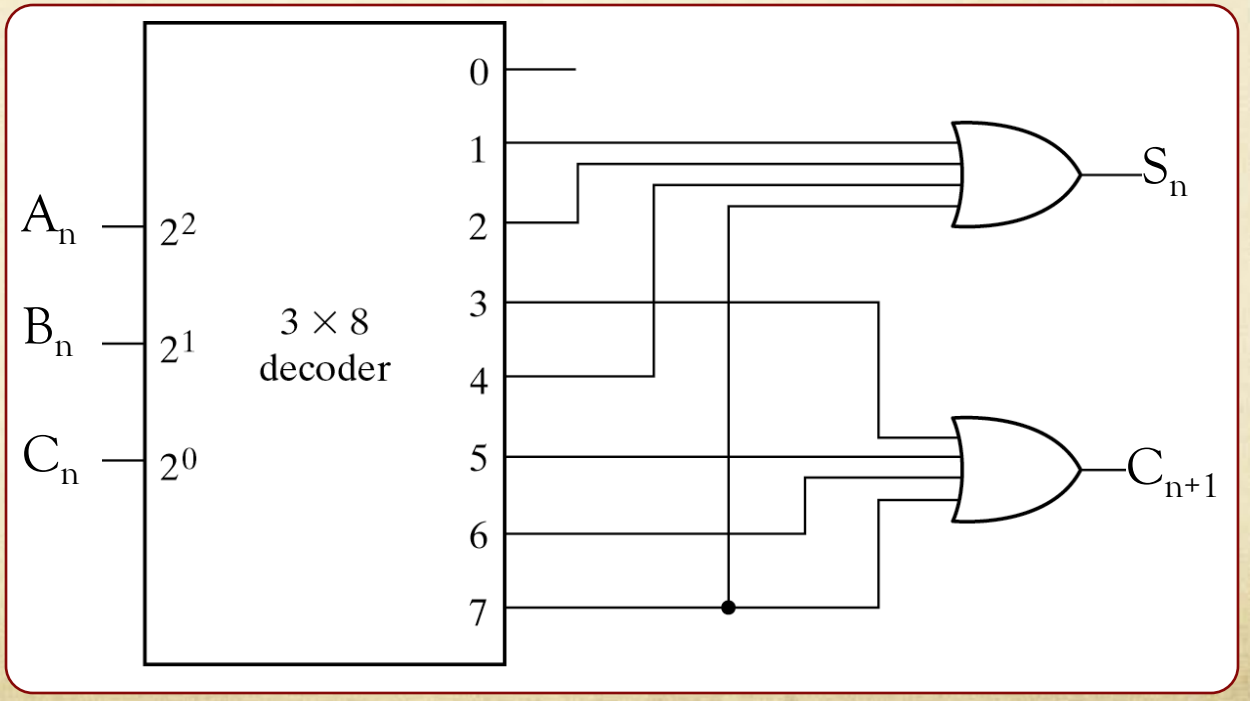


Dekoder – generering av logiske funksjoner

Dekoder - elektrisk sannhetstabell. Kan generere generelle logiske funksjoner direkte fra mintermene på utgangen

Eksempel:

Fulladder



Enkoder

Enkoder – motsatt av dekoder

Eksempel: 8x3 enkoder

Innganger								Utganger		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

$$x = D_4 + D_5 + D_6 + D_7$$
$$y = D_2 + D_3 + D_6 + D_7$$
$$z = D_1 + D_3 + D_5 + D_7$$

Antar at det ikke eksisterer andre inngangskombinasjoner

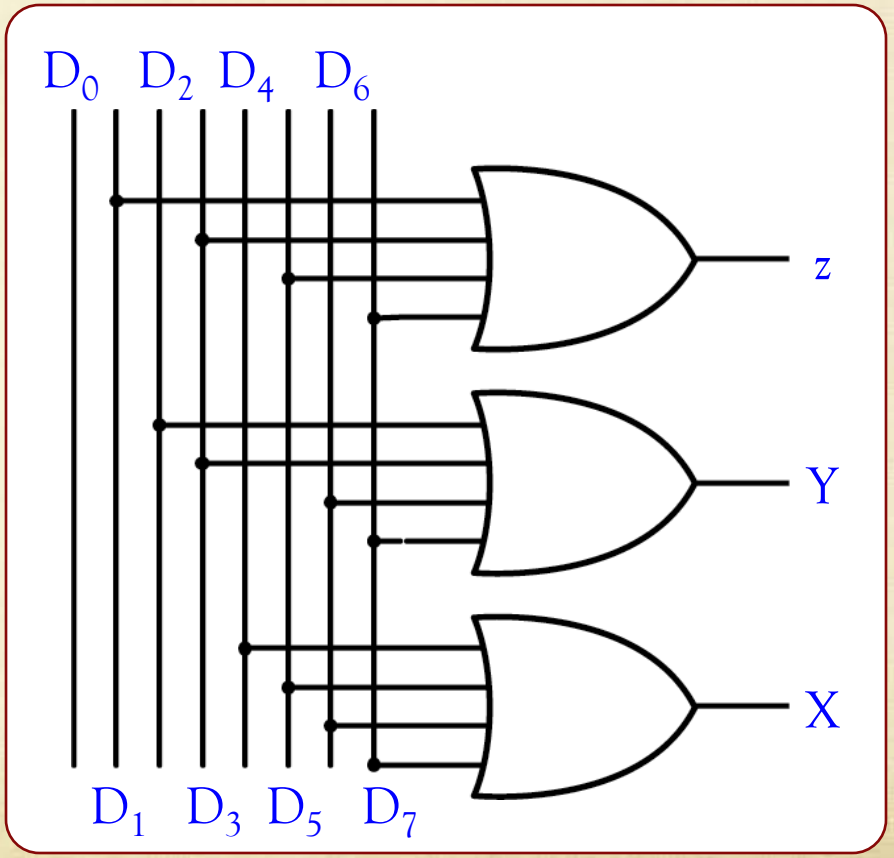
Enkoder

Eksempel

$x = D_4 + D_5 + D_6 + D_7$

$y = D_2 + D_3 + D_6 + D_7$

$z = D_1 + D_3 + D_5 + D_7$



Prioritets-enkoder

Problem i enkodere: Hva hvis man får flere "1"ere inn samtidig?

Løsning: Prioritets-enkoder

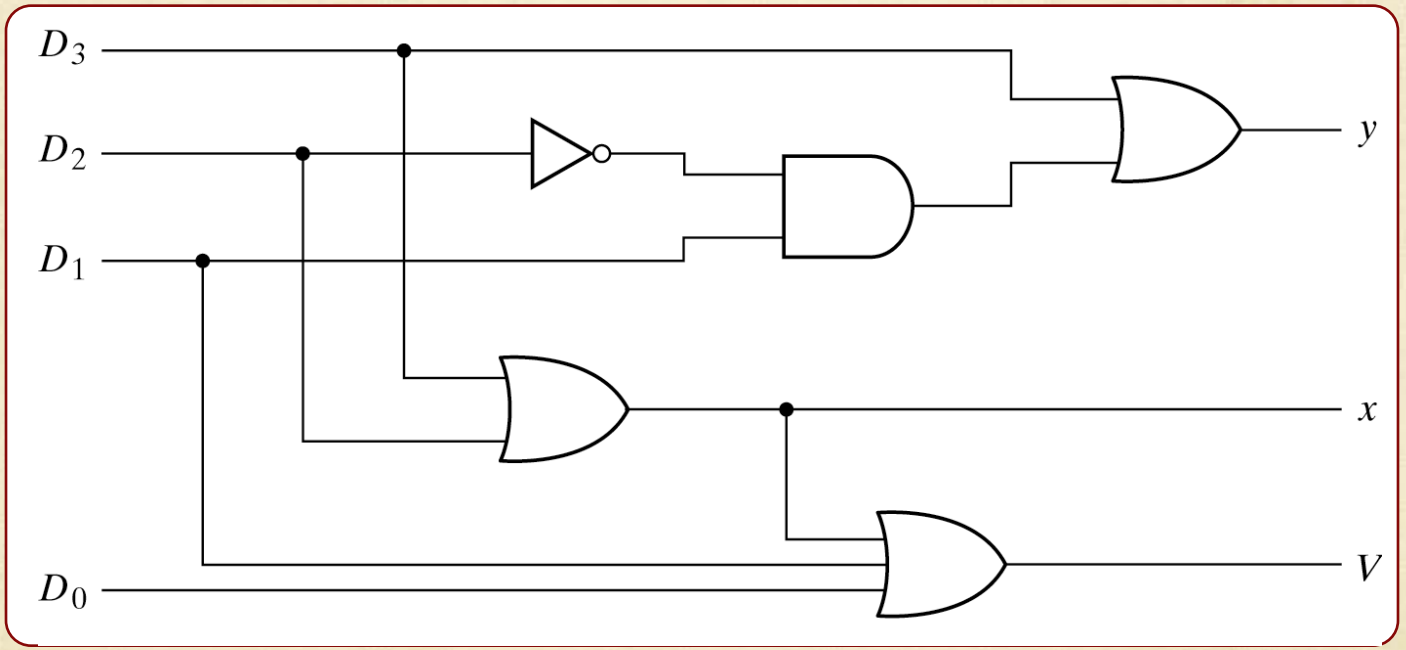
Hvis flere "1"ere inn - ser kun på inngang med høyst indeks (prioritet)

Eksempel: 8x3 prioritets-enkoder

Innganger								Utganger		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
x	1	0	0	0	0	0	0	0	0	1
x	x	1	0	0	0	0	0	0	1	0
x	x	x	1	0	0	0	0	0	1	1
x	x	x	x	1	0	0	0	1	0	0
x	x	x	x	x	1	0	0	1	0	1
x	x	x	x	x	x	1	0	1	1	0
x	x	x	x	x	x	x	1	1	1	1

Prioritets-enkoder

Eksempel: 4x2 prioritets-enkoder med "valid" utgang

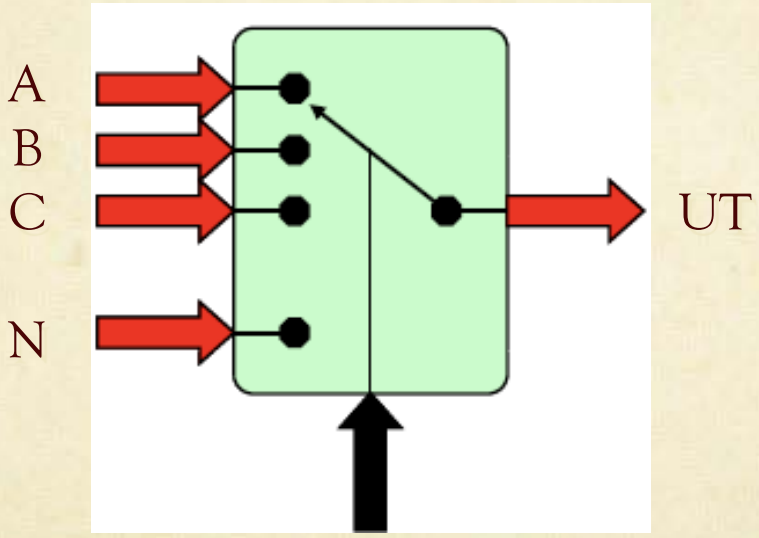


"V" signaliserer at minst en inngang er "1"

Multiplekser

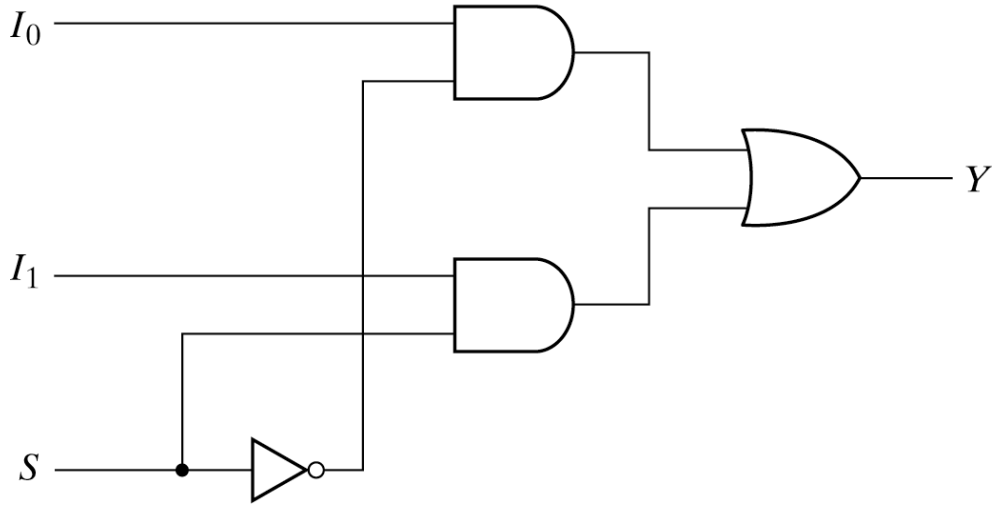
Multiplekser (MUX) – velger hvilke innganger som slippes ut

Hver inngang kan bestå av ett eller flere bit

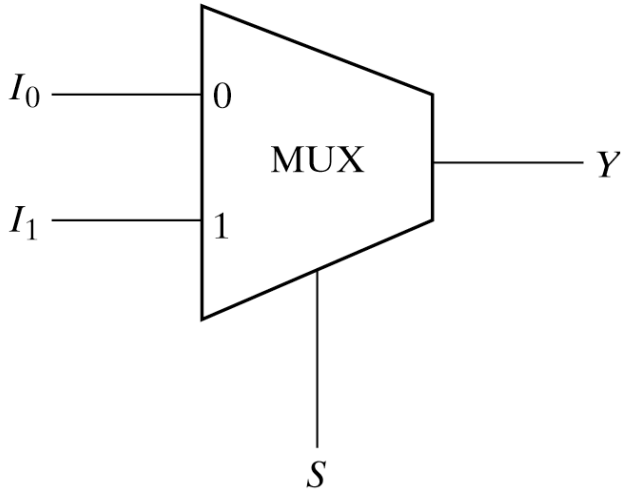


MUX

Eksempel: 2-1 MUX

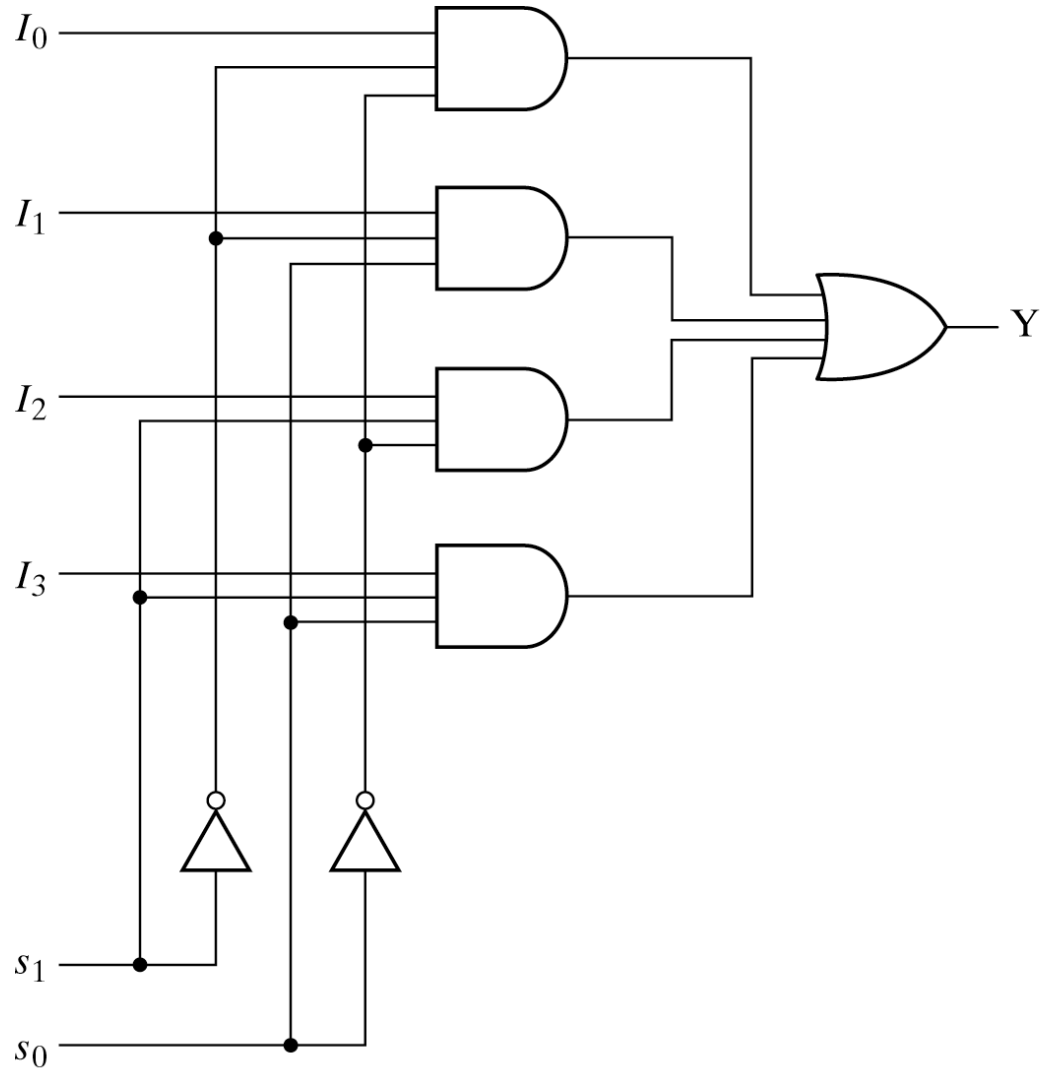


Implementasjon



Symbol

MUX

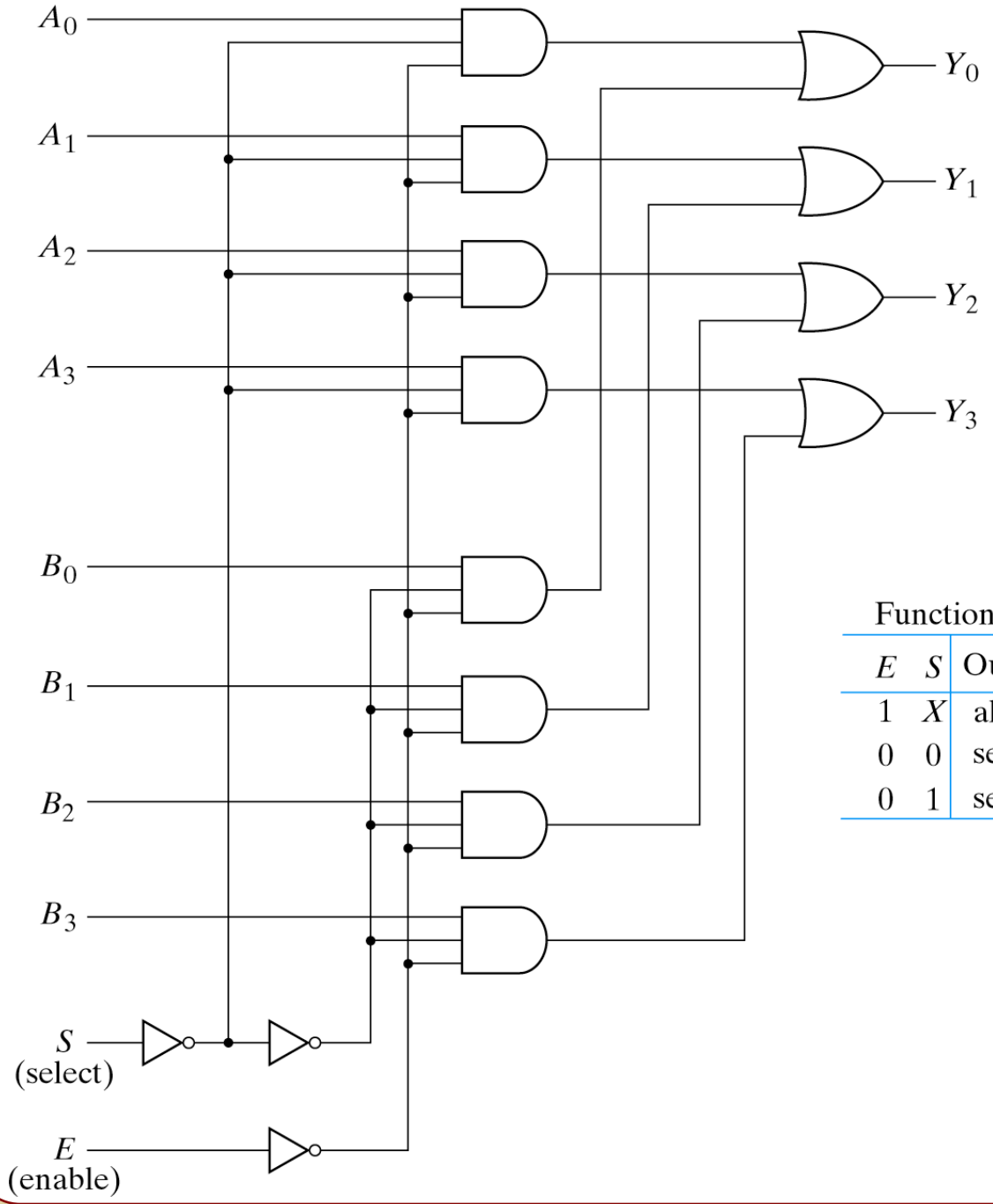


s_1	s_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) Function table

MUX

Eksempel: 2-1 MUX

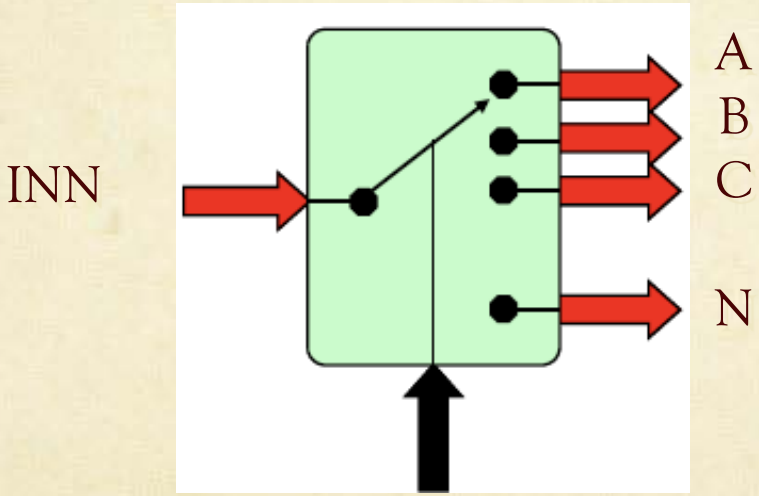


Function table

E	S	Output Y
1	X	all 0's
0	0	select A
0	1	select B

Demultiplekser

Demultiplekser – motsatt av multiplekser



ALU

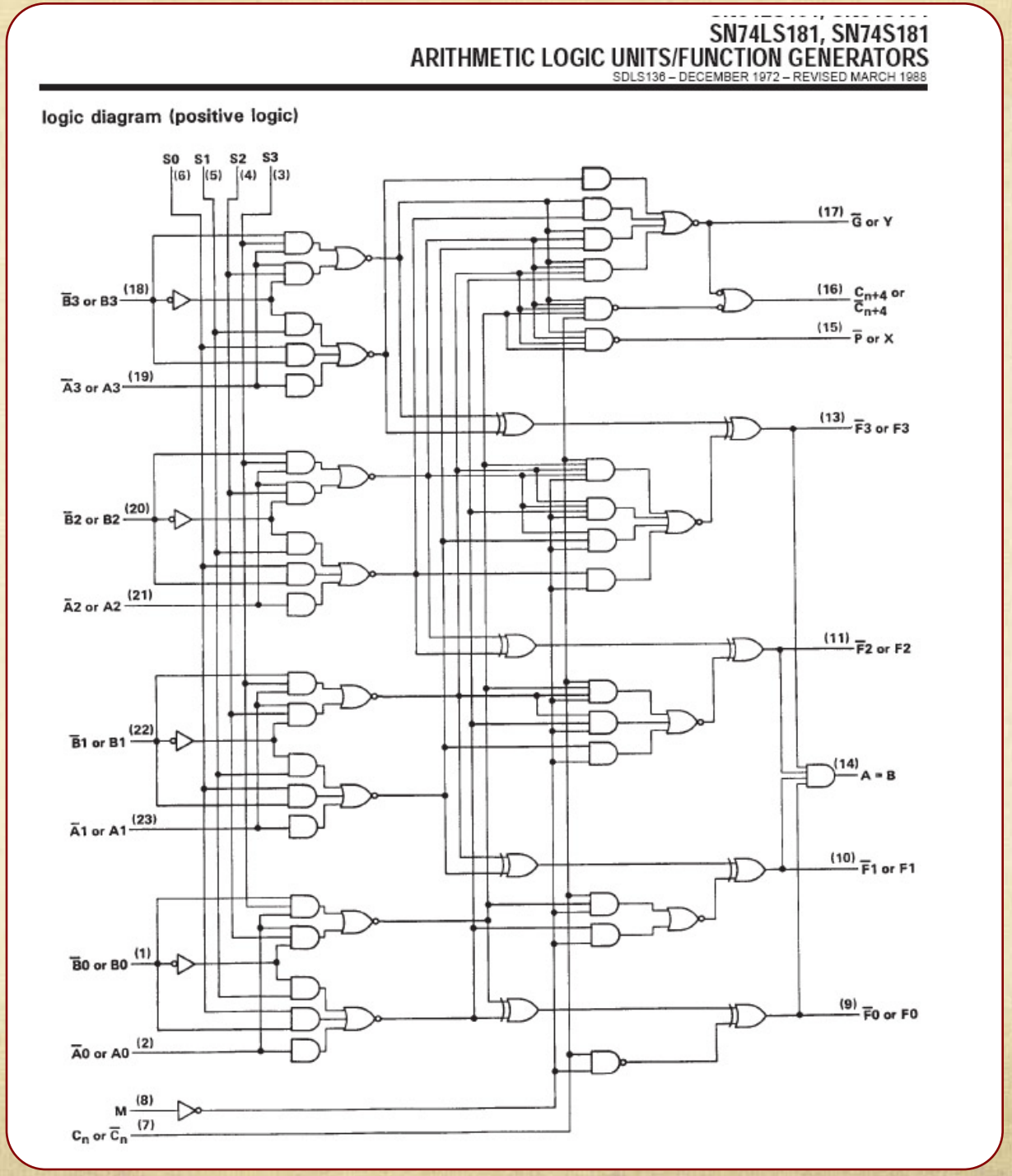
ALU – Arithmetic Logic Unit

Generell regneenhet

Eksempel: SN74LS181

4bit utbyggbar ALU

30 forskjellige operasjoner

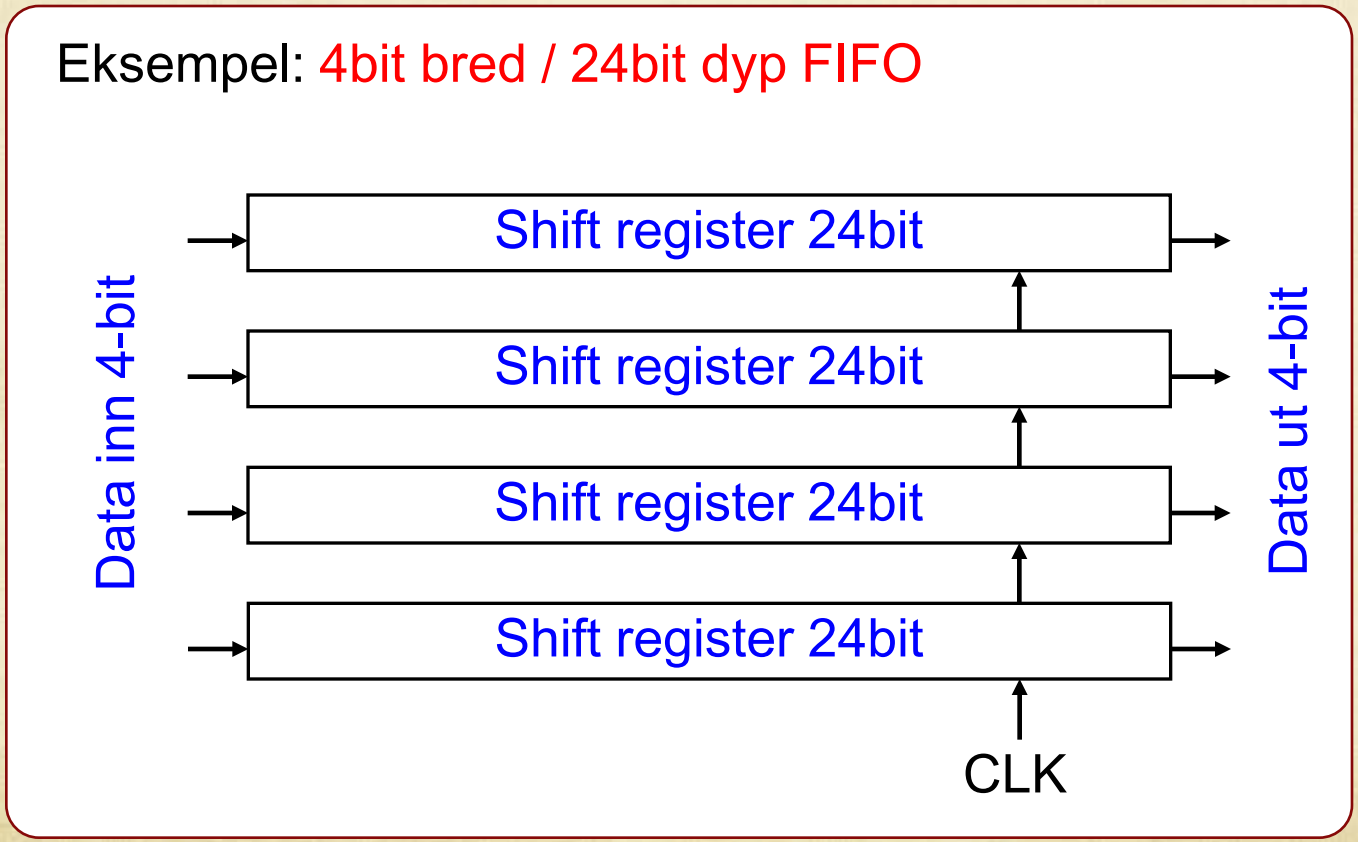


ALU - SN74LS181

SELECTION				ACTIVE-HIGH DATA		
				M = H	M = L; ARITHMETIC OPERATIONS	
S3	S2	S1	S0	LOGIC FUNCTIONS	$\overline{C_n}$ = H (no carry)	$\overline{C_n}$ = L (with carry)
L	L	L	L	$F = \overline{A}$	F = A	F = A PLUS 1
L	L	L	H	$F = \overline{A + B}$	F = A + B	F = (A + B) PLUS 1
L	L	H	L	$F = \overline{A}B$	$F = A + \overline{B}$	$F = (A + \overline{B})$ PLUS 1
L	L	H	H	F = 0	F = MINUS 1 (2's COMPL)	F = ZERO
L	H	L	L	$F = \overline{AB}$	$F = A$ PLUS $A\overline{B}$	$F = A$ PLUS $A\overline{B}$ PLUS 1
L	H	L	H	$F = \overline{B}$	$F = (A + B)$ PLUS $A\overline{B}$	$F = (A + B)$ PLUS $A\overline{B}$ PLUS 1
L	H	H	L	$F = A \oplus B$	F = A MINUS B MINUS 1	F = A MINUS B
L	H	H	H	$F = A\overline{B}$	$F = A\overline{B}$ MINUS 1	$F = A\overline{B}$
H	L	L	L	$F = \overline{A} + B$	F = A PLUS AB	F = A PLUS AB PLUS 1
H	L	L	H	$F = \overline{A \oplus B}$	F = A PLUS B	F = A PLUS B PLUS 1
H	L	H	L	F = B	$F = (A + \overline{B})$ PLUS AB	$F = (A + \overline{B})$ PLUS AB PLUS 1
H	L	H	H	F = AB	F = AB MINUS 1	F = AB
H	H	L	L	F = 1	F = A PLUS $A^{\dagger}$	F = A PLUS A PLUS 1
H	H	L	H	$F = A + \overline{B}$	F = (A + B) PLUS A	F = (A + B) PLUS A PLUS 1
H	H	H	L	F = A + B	$F = (A + \overline{B})$ PLUS A	$F = (A + \overline{B})$ PLUS A PLUS 1
H	H	H	H	F = A	F = A MINUS 1	F = A

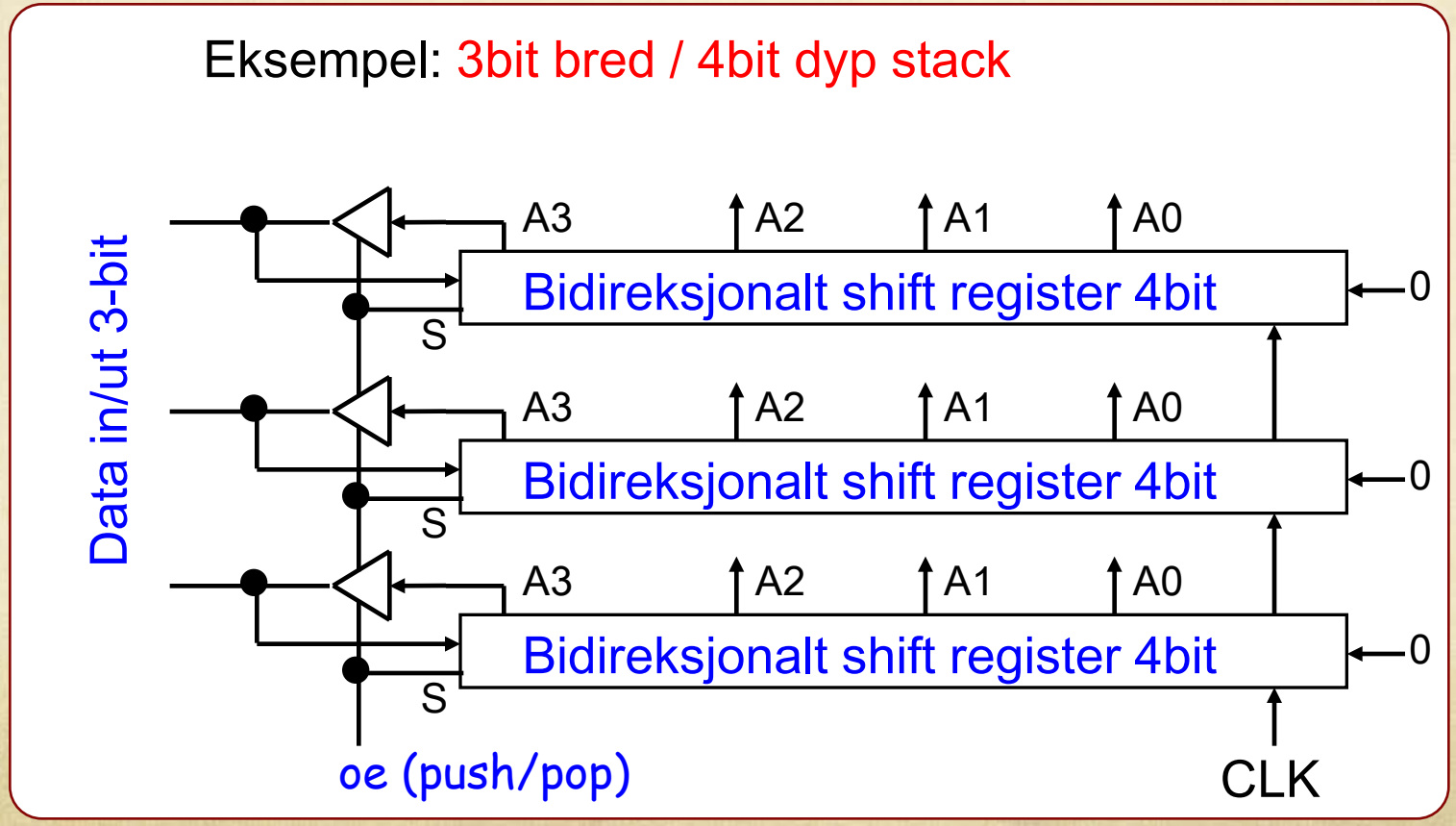
FIFO

FIFO – ”First in first out” (mellomlagringsenhet, buffer)



Stack

Stack – Mellomlagringsenhet for data, data kommer inn (push) og ut (pop) i samme ende



CPU

- CPU – Central Processing Unit

- "Hjernen" i en vanlig seriell datamaskin
- Von Neuman prinsippet – all data som skal behandles må innom CPU'en
- En CPU styres av programkode. Denne koden består av et sett med lavnivå (maskinkode) instruksjoner
- Maskinkode-instruksjoner er det laveste nivået man kan programmere på
- Programmering direkte i maskinkode kan i teorien gi optimale programmer (hastighet/plass), men blir fort ekstremt tungvint og uoversiktlig for større program

RISC - CPU

- RISC – Reduced Instruction Set Computer (få maskinkode-instruksjoner)

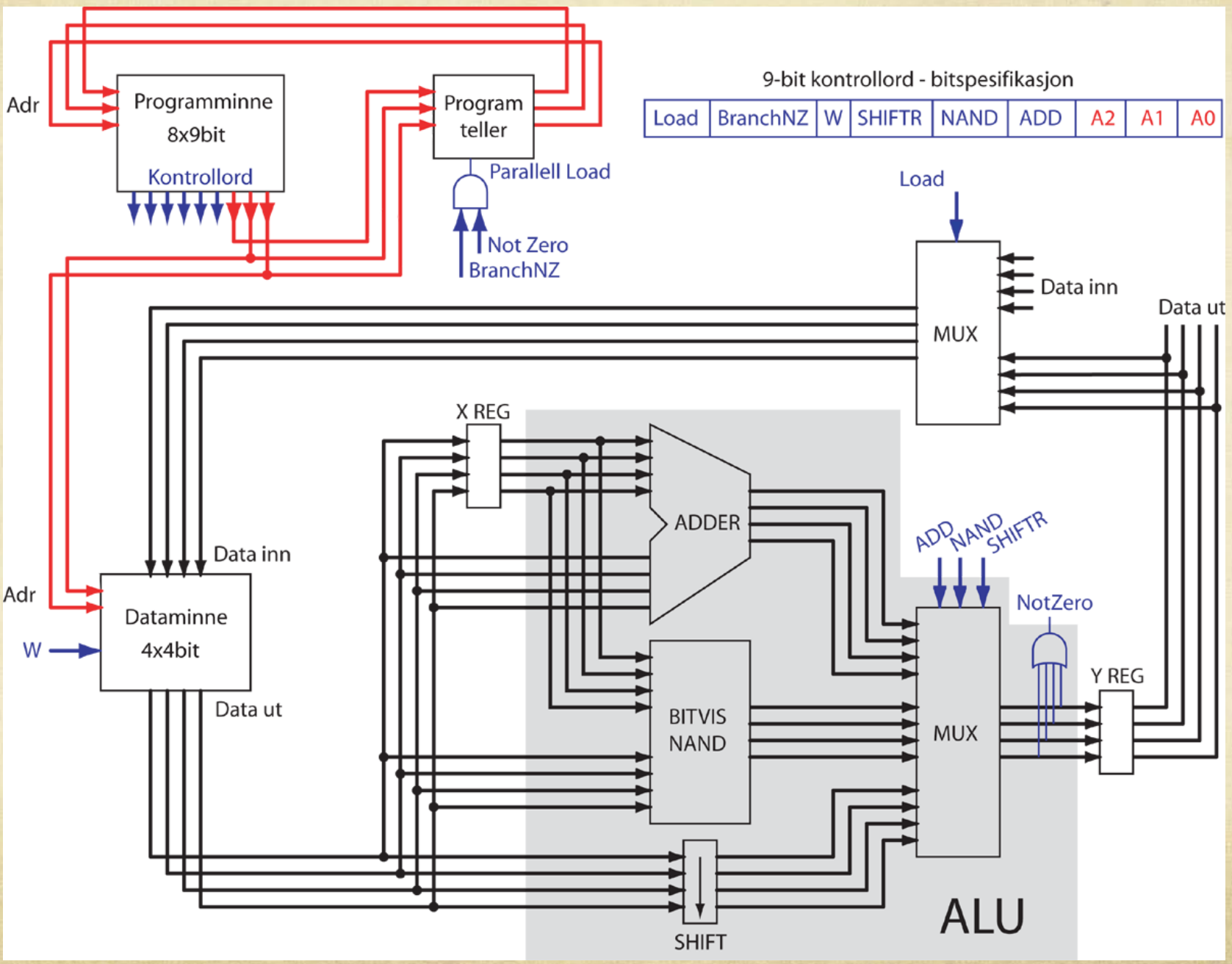
- RISC-prinsippet regnes i dag (av mange) som meget effektivt

- Dess færre maskinkodeinstruksjoner man har til rådighet dess flere mellomoperasjoner må man utføre
- Dess færre maskinkodeinstruksjoner man har dess raskere kan de utføres (til en viss grad)
- RISC prosessorer tar i utgangspunktet mindre plass enn CISC (Compleks Instruction Set Computer) prosessorer

En minimal RISC - CPU

- Vi vil nå presentere en ekstrem RISC prosessor med kun 6 instruksjoner. (Pentium4 CISC har >200)
- Den følgende RISC prosessoren er kraftig forenklet, men er fullt funksjonell, og med mer minne kan den utføre alle oppgaver man kan forvente av en vanlig CPU
- Den følgende RISC prosessoren kunne ha vært forenklet ytterligere men dette er ikke gjort av pedagogiske grunner

Komplett CPU: 4-bit databuss / 3bit adressebuss



Virkemåte

- Alle flip-flopper, registre og programteller klokkes av ett felles klokkesignal
- Innholdet i programtelleren angir adressen til neste instruksjon i programminnet.
- Programtelleren øker instruksjonsadressen med 1 for hver klokkeperiode (hvis den ikke skal gjøre et hopp)
- Når en ny instruksjon (kontrollord) lastes ut av programminnet vil de 6 første bittene i kontrollordet (OP-koden) direkte styre hva som skjer i systemet:
- Siste del av kontrollordet (3bit) brukes til å angi adresse i dataminnet for hvor data skal inn/ut evt. adresse i programminnet for hopp

Kontrollord (OP kode)

- Bit nr. 8 (load) styrer en MUX som velger om data inn til dataminne skal komme utenifra (IO) eller fra Y register
- Bit nr. 7 (branchNZ) (hopp hvis resultat ikke er 0) avgjør om programteller skal telle opp eller laste inn ny adresse (parallell load) hvis siste beregning i ALU ikke gir 0

- Bit nr. 6 (write) styrer om dataminnet skal lese ut data eller skrive inn ny data. (data som blir lest inn vil samtidig være tilgjengelig ut i dette systemet)

Kontrollord (ALU)

ALU-en utfører 3 operasjoner samtidig

1) Addisjon av nåværende data med forrige data (fra X registret)

2) Bitvis NAND av nåværende data med forrige data
3) Høyre shift av nåværende data

- Bit nr. 5 (shift) velger det høyre-shiftede resultatet ut fra ALU-en

- Bit nr. 4 (NAND) velger det bitvis NANDede resultatet ut fra ALU-en
- Bit nr. 3 (ADD) velger addisjons resultatet ut fra ALU-en

Kontrollord (adressedel)

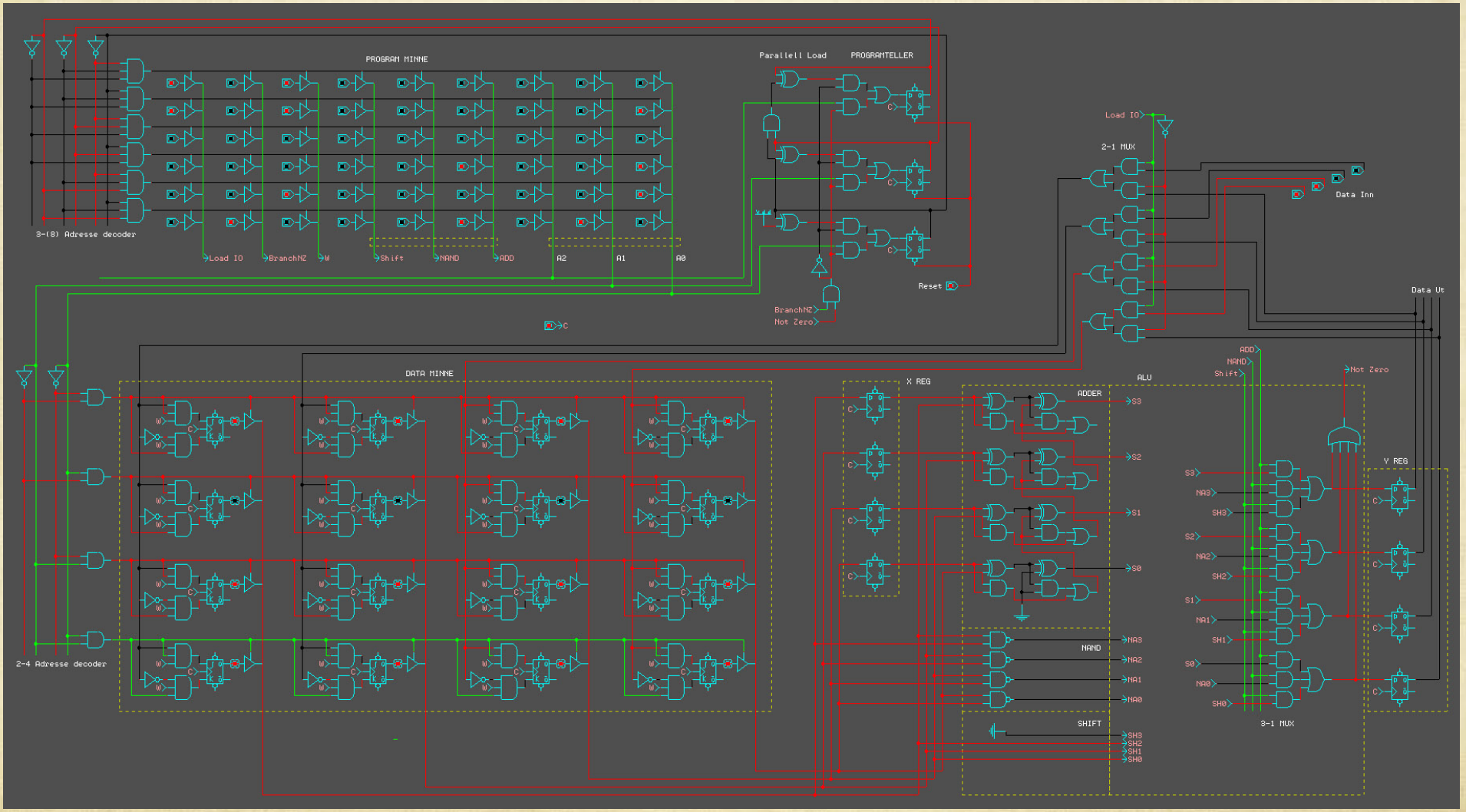
- Bit nr. 2 (A2) bit nr.2 i adresse for hopp
- Bit nr. 1 (A1) bit nr.1 i adresse for hopp / data
- Bit nr. 0 (A0) bit nr.0 i adresse for hopp / data

En minimal RISC - CPU

Kommentarer:

- Ordbredden på databussen er her satt til 4bit. Man kan uten videre utvide denne til 32-64bit hvis behov
- Antall ord i data/programminnet (her valgt til 4/6) kan utvides til hva man måtte ønske
- Lengden på OP-koden kan reduseres hvis man bruker binær koding
- ALU-en kan utvides med flere direkte regneoperasjoner for å spare mellomregninger
- Man kan legge inn muligheten av å direkte legge inn faste verdier (immediate) i dataminnet for å spare mellomregninger

Komplett CPU: Funksjonell Diglog-implementasjon



Maskinkode-instruksjoner (kontrollordet i programminnet)

Bit nr.	8	7	6	5	4	3	2	1	0
	Load	BranchNZ	W	ShiftR	NAND	ADD	A2	A1	A1
Hent data fra angitt minneadresse og utfør ADD med forrige data	0	0	0	0	0	1	X	A1	A0
Hent data fra angitt minneadresse og utfør BIT-NAND med forrige data	0	0	0	0	1	0	X	A1	A0
Hent data fra angitt minneadresse og shift data ett bit til høyre	0	0	0	1	0	0	X	A1	A0
Skriv data til angitt minneadresse	0	0	1	X	X	X	X	A1	A0
Load data fra IO til angitt minneadresse	1	0	1	X	X	X	X	A1	A0
Hopp til angitt programadresse hvis forrige data ikke ble 0	0	1	0	X	X	X	A2	A1	A0

RISC program-eksempler

Eksempel 1: Ta inn to tall fra IO, adder tallene og gi svaret til IO. Løsning:

0. Les inn første tall fra IO, og legg det i dataminne nr. 00

   Maskinkode: 101 000 000

1. Les inn neste tall fra IO, og legg det i dataminne nr. 01

   Maskinkode: 101 000 001

2. Hent ut siste tall fra dataminne 01 (legges i X reg)

   Maskinkode: 000 000 001

3. Les ut første tall fra dataminne 00, og utfør ADD med forrige tall (som er innholdet i dataminne 01)

   Maskinkode: 000 001 000

4. Skriv resultat til (for eksempel) dataminne nr. 10

   Maskinkode: 001 000 010

(Databussen er direkte synlig for IO)

RISC program-eksempler

Eksempel 2: Implementasjon av for-løkke

Java kode	*Hva vi setter CPU til å gjøre*
int k;	// velger dataadresse 01
int n;	// velger dataadresse 10
k = IO.inInt();	// Henter k fra IO
for (n=0; n<k; n++)	
... ;	// Her kan man legge hva som helst
	// k = k-1, hopper tilbake hvis k≠0
...	// Neste instruksjon

*Vår RISC kan ikke subtrahere direkte. Legger derfor først inn 1111 (2er komplement for -1) i dataminne 00. Kan så addere -1 til k underveis i løkken

RISC program-eksempler

Eksempel 2: Implementasjon av for-løkke

0. Les inn ”-1” (1111) fra IO (for enkelhetens skyld) og legg det i dataminne nr. 00. Maskinkode: 101 000 000
1. Les inn ”k” fra IO, og legg det i dataminne nr. 01. Maskinkode: 101 000 001
2. Hent fram ”-1” fra dataminne. Maskinkode: 000 000 000
3. Hent fram ”k” fra dataminne og ADD med forrige data (-1). Maskinkode: 000 001 001
4. Lagre resultat til ”k”. Maskinkode: 001 000 001
5. Hopp til program linje nr. 2 hvis resultatet ikke var 0. Maskinkode: 010 001 010

Hovedpunkter

- Komparator
- Dekoder/enkoder
- MUX/DEMUX
- Demultiplekser
- Kombinert adder/subtraktor
- ALU
- FIFO
- Stack
- En minimal RISC - CPU