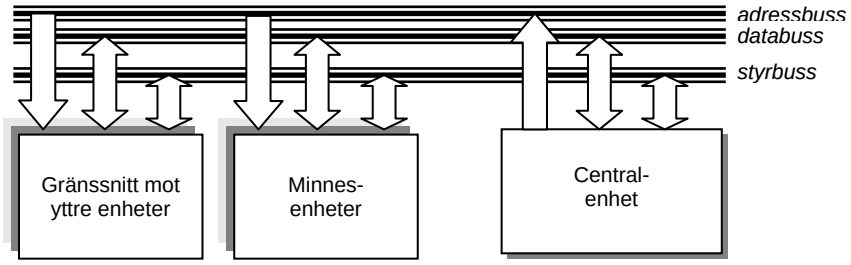
2 UPPBYGGNAD OCH FUNKTION

Detta kapitel ägnas åt metoder och principer som används för att bygga upp ett komplett datorsystem bestående av centralenhet, minne och in- ut-/matningsenheter.

- *Bussystem, intern kommunikation i datorsystemet*
- *Adressavkodning, hur primärminne och I/O-enheter kan anslutas*
- *Olika principer för intern kommunikation (busskommunikation)*

2.1 Bussystem

Datorn är uppbyggd av enheter som kommunicerar med varandra via elektriska ledare. En grupp av sådana ledare kallas för *buss*. Begreppet *buss* anger alltså ett antal olika elektriska signalförare som funktionellt kan grupperas tillsammans. Flera olika bussar används för att koppla samman *centralenhet, minne* och olika typer av *gränssnitt*. Beteckningen *gränssnitt* är här en gemensam beteckning på enheter för in- och/eller ut-/matning från/till elektroniska system som kopplats till datorn. Olika typer av gränssnitt är ofta utförda som integrerade kretsar (eller sammanbyggda med centralenhet och minne) och de kallas också i bland för *periferikretsar*. Allt informationsutbyte som sker i datorsystemet görs normalt via tre bussar: *databuss*, *adressbuss* och *styrbuss*.



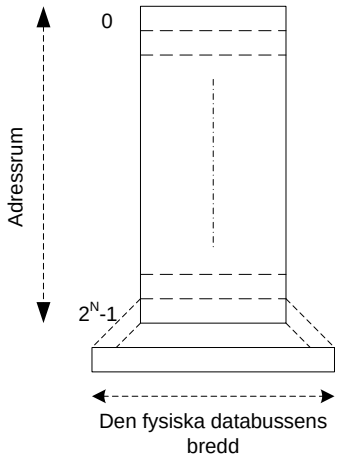
FIGUR 2.1 BUSSYSTEM

Centralenheten initierar kommunikationen genom att *välja* ett gränssnitt mot någon yttre enhet eller en minnesenhet via adressbussen. Varje enskild enhet har således ett unikt adressintervall inom vilket endast denna enhet aktiveras för att överföra data till eller från centralenheten via databussen. Signaler på styrbussen anger i vilken riktning data överförs. Observera hur vi i Figur 2.1 indikerar att adressbussens innehåll bestäms enbart av centralenheten. Detta är inte alltid fallet men för diskussioner i detta kapitel förutsätter vi att endast centralenheten kan generera adresser på adressbussen. Då det gäller databussen är situationen en annan. Såväl centralenheten som minnesenheter och yttre enheter tillåts placera data på databussen. Detta avgörs dock av signaler på styrbussen vilka i sin tur genereras av centralenheten. Av denna anledning har vi ritat databussen dubbelriktad i figuren. Slutligen innehåller styrbussen signaler som kan ha genererats antingen av centralenheten eller av någon yttre enhet och vi ritar av denna anledning även styrbussen som dubbelriktad.

2.1.1 Adressbussen

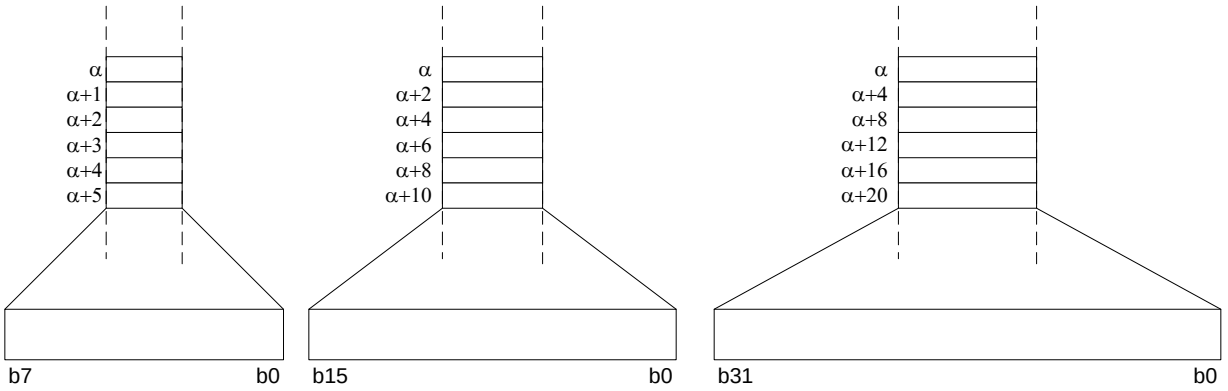
Adressbussens bredd (storlek) bestäms av centralenhetens konstruktion. Antalet adressledningar varierar följaktligen mellan olika processorer men, som exempel, är 16 bitar vanligt för mindre processorer, dessa har då oftast *ordbredden* 8 bitar, dvs. 8 bitars databuss.

För större processorer förekommer 20, (exempelvis Intel 8086), 24 (exempelvis Motorola MC68000), 32 (exempelvis Motorola MC68020) adressledningar. För större processorer är också ordbredden vanligtvis större, typiskt 16 eller 32 bitar, men även 64 bitars databuss förekommer. Adressbussens totala omslutning kallas processorns adressrum.



2.2 ADRESSRUM

Adressbussen är vanligtvis utformad för att adressera *bytes*, dvs. varje enskild adress pekar ut en unik byte (8 bitar) i primärminnet. Hos vissa processorer har adressering i stället utformats enbart för jämna bytes, för adressering av 16-bitars data, detta kallas då ofta ”word-adressering”. Fördelen med det sistnämnda är att 16 bitar då kan läsas/skrivas vid varje minnesoperation och prestanda blir alltså högre. För att ytterligare förbättra prestanda finns också ”longword-adressering” , dvs. 32 bitar behandlas i en arbetscykel, vilket oftast innebär att läsning eller skrivning endast kan ske på adresser som slutar med en med en multipel 4, se Figur 2.3.



FIGUR 2.3 OLIKA SÄTT ATT ORGANISERA MINNET, SOM 8,16 ELLER 32 BITARS ENTITETER

Processorer med dessa egenskaper introducerar då restriktioner på hur data kan lagras i minnet, dvs 16-bitars data måste lagras på en jämn adress, etc. Sådana restriktioner kallas ”alignment conditions”, dvs. villkor för hur data kan lagras i minnet.

2.1.2 Databussen

Medan syftet med adressbussen som tidigare sagts är att ”peka ut” den del av datorn som centralenheten avser att utväxla information med är databussens uppgift att överföra informationen mellan dessa delar. Databussens bredd bestämmer hur mycket information som kan överföras under en arbetscykel. Ju mer information som kan överföras desto högre blir förstås prestanda, givet att arbetscykelns tid inte ändras. Databussens bredd är oftast bestämd av ordbredden hos centralenhetens aritmetikenhet (ALU). Ordbredder är alltid multipler av bytes (8 bitar). Vanliga ordbredder är 8,16,32 och 64 bitar. Vissa processorer kan inte distinkt placeras i någon sådan grupp. Exempelvis är Motorola 68HCS12 något av en ”hybrid”, där aritmetikenheten behandlar såväl 8 som 16-bitars tal och databussen kan vara 8 eller 16 bitar beroende på implementering, dvs. ”derivat” av processorarkitekturen.

Eftersom databussen används för att överföra information såväl till som från centralenheten är bussen också dubbelriktad. Bussens innehåll bestäms därför av att antingen centralenheten, en yttre enhet eller någon minnesenhet *driver* bussen. Det är viktigt att inse att två enheter aldrig samtidigt kan (får) driva databussen eftersom dess innehåll då för det första är att betrakta som nonsens, men det kan dessutom medföra att de kretsar som kopplats till bussen kan förstöras. *Bussarbitreringen,* dvs. avgörandet av vilken enhet som för tillfället får driva databussen, bestäms av centralenheten.

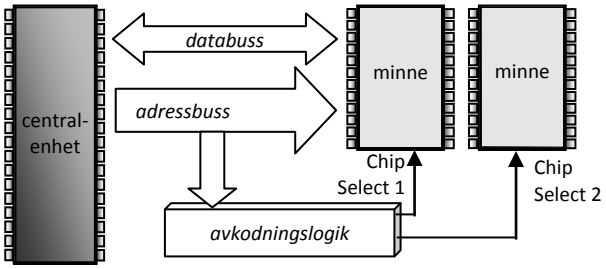
2.1.3 Styrbussen

Styrbussen omfattar, precis som namnet antyder, ett antal kontrollsignaler avsedda att dirigera informationsöverföringen. I denna grupp ingår samtliga styrsignaler som krävs för kommunikationen mellan enheterna anslutna till bussystemet. Ett exempel på en styrsignal från centralenheten är skriv- och lässignalen (*R/W, Read/Write*) där centralenheten anger att den ska läsa (hämta data från) minne eller gränssnitt respektive skriva till (överföra data till) minne eller gränssnitt. Funktionerna är i bland implementerade som en signal, där exempelvis hög nivå innebär att centralenheten förväntar sig att någon enhet driver bussen (hämta data från...) medan en låg nivå innebär att centralenheten driver bussen (överföra data till...). Det är kanske ännu mer vanligt att dessa styrsignaler implementerats som separata styrledningar där lässignalen (*RE, Read Enable*) då indikerar dataöverföring från extern enhet (eller minne) till centralenheten respektive skrivsignalen (*WE, Write Enable*) anger att centralenheten driver databussen för en överföring till någon extern enhet (eller minne). Uppenbarligen kan inte dessa signaler vara aktiva samtidigt. De kan dock vara passiva samtidigt och indikerar då att centralenheten är upptagen med internt arbete som inte kräver kommunikation med någon yttre enhet eller minnet.

2.2 Adressavkodning

Med ”adressavkodning” menar man den den teknik som används för att avkoda adressbussen och bilda den speciella signal, ofta kallad ”chip-select” (CS) som används för att aktivera en speciell enhet så som minne eller gränssnitt. Vi har sett att centralenheten kommunicerar med såväl minne som gränssnitt via

systemets bussar. Låt oss nu titta närmre på hur detta går till, hur man kan vara säker på att rätt minneskrets eller rätt gränssnitt adresseras av centralenheten. Oftast består datorn av flera minnestyper organiserade i block, och vanligtvis ingår också flera typer av gränssnitt. Vi ska nu inledningsvis, som ett enkelt exempel, redogöra för hur ett komplett datorsystem, med centralenhet, två minneskretsar och två gränssnitt kan byggas upp. Vi gör detta genom att konstruera ett block bestående av grindar. Detta logikblock kallas *adress-avkodningslogik* (Figur 2.4).

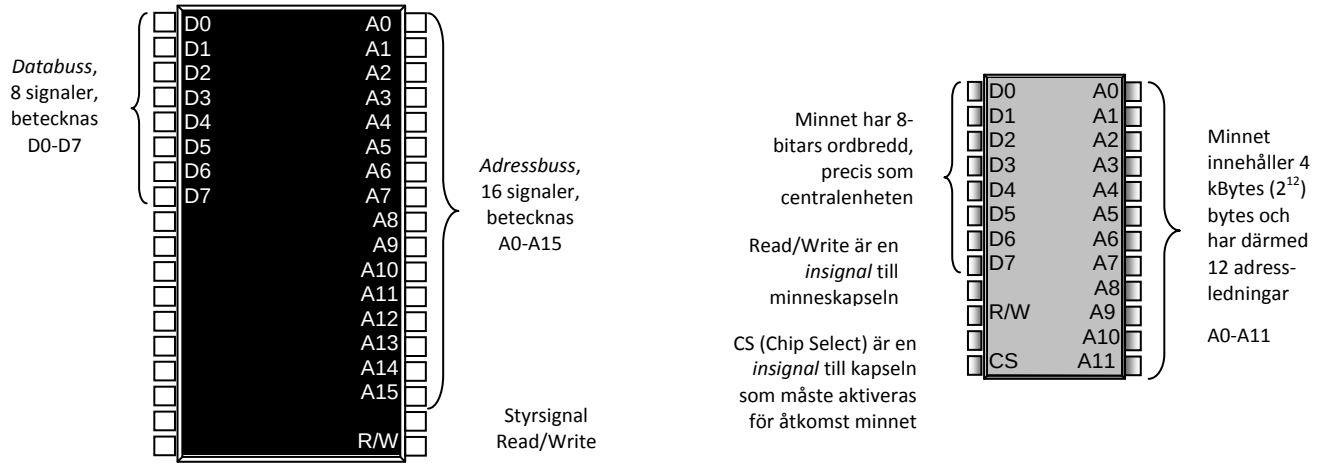


FIGUR 2.4 ADRESSAVKODNINGSLOGIK

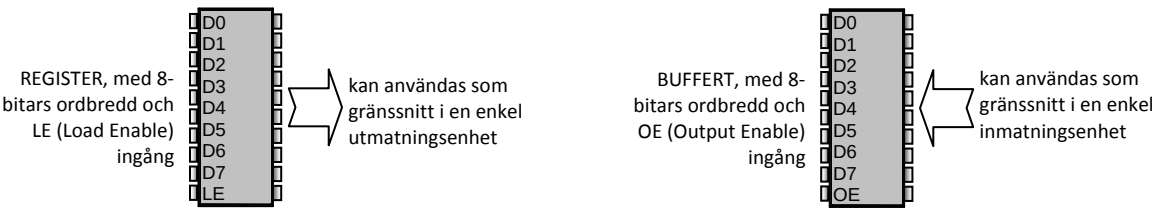
För exemplet antar vi att vi har tillgång till följande komponenter.

- Centralenhet, 64kByte adressrum (16 bitars adressbuss), 8-bitars ordbredd (databuss),
- Minneskapsel RWM 4 kbyte,
- Minneskapsel ROM 4 kbyte
- Register, 8 bitar som inport, Register, 8 bitar som utport.
- Standardkretsar med diverse grindar.

Med de givna förutsättningarna kan vi ge förenklade bilder av de ingående komponenterna. I dessa bilder utelämnar vi signaler och komponenter som inte behövs för adressavkodningen.



FIGUR 2.5 FÖRENKLADE FIGURER AV CENTRALENHET OCH MINNE



FIGUR 2.6 FÖRENKLADE FIGURER AV REGISTER OCH BUFFERT

Vi börjar med att bestämma var, i centralenhetens adressrum, vi vill placera minnen och register. För exemplet väljer vi följande minnesdisposition (adresserna anges i hexadecimal form):

Kapsel	**Startadress**	**Slutadress**
Minneskapsel RWM 4 kbyte	0000	0FFF
Minneskapsel ROM 4 kbyte	C000	CFFF
Buffert, 8 bitar som inport	A000	A000
Register, 8 bitar som utport	8000	8000

TABELL 2.1 MINNESDISPOSITION

Av minnesdispositionen framgår att:

- RWM-minnet ska aktiveras om centralenheten genererar någon av adresserna 0-0FFF.
- ROM-minnet ska aktiveras om centralenheten genererar någon av adresserna C000-CFFF.
- Inporten ska aktiveras om centralenheten genererar adress A000.
- Utporten ska aktiveras om centralenheten genererar adress 8000.

Följande tabell visar de värden adressbussens signaler får anta för respektive kapsel.

Kapsel		**Adressbuss**															
		A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
RWM	$0FFF	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
	$0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ROM	$CFFF	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1
	$C000	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
inport	$A000	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	$A000	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
utport	$8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	$8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

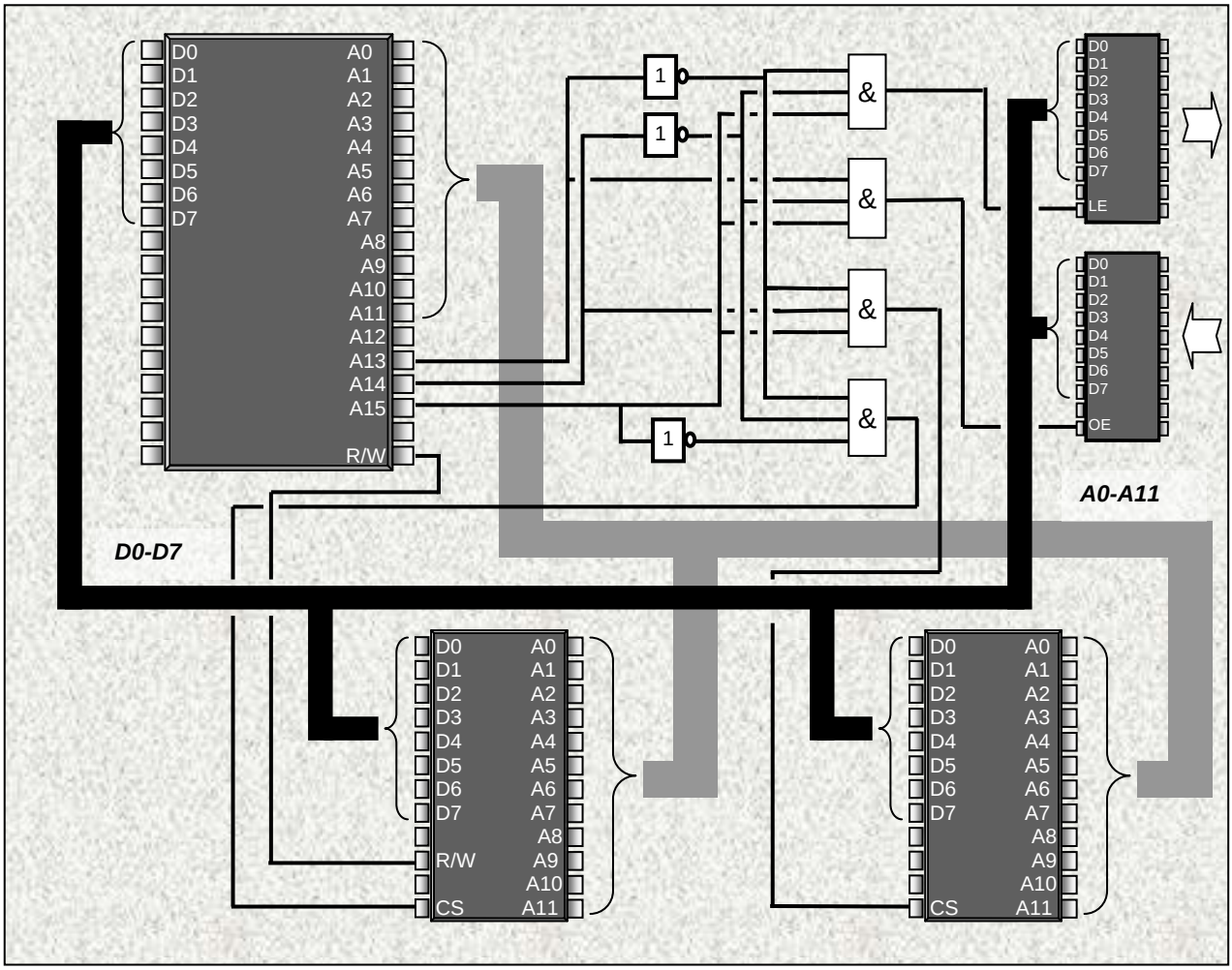
Vi måste nu, konstruera *logik* som kan jämföra adressbussens värde med den del av adressrummet vi tilldelat respektive kapsel och skapa en signal CS (*chip select*) för varje kapsel. För att göra detta använder vi grindar av typen NOT och AND. Av tabellen framgår att vi kan göra en unik CS-signal för varje kapsel genom att bara använda adressledningarna A15, A14 och A13. Vi ser detta tydligare i följande uppställning:

Kapsel		**Adressbuss**		
		A15	A14	A13
RWM	$0000-$0FFF	0	0	0
ROM	$C000-$CFFF	1	1	0
inport	$A000	1	0	1
utport	$8000	1	0	0

För exempelvis RWM-kapseln ska dessa tre adressignaler vara 0, för ROM-kapseln ska vi ha A15=1, A14=1 och A13=0, osv. Genom att bara låta dessa signaler ingå i adressavkodningen får vi minsta möjliga kretslogik. Eftersom vi lämnar adressledningar från A12 och nedåt som ”don’t care” kommer CS-signalen att vara aktiv för ett större adressintervall än det som från början avsågs för respektive kapsel. Man säger att samma modul avbildas på flera olika adressintervall och kallar detta *ofullständig adressavkodning*. Följande tabell visar nu vårt slutliga val av adressavkodning

Kapsel	**Adressbuss**			
	A15	A14	A13	Adressintervall
RWM	0	0	0	0000-1FFF
ROM	1	1	0	C000-DFFF
inport	1	0	1	A000-BFFF
utport	1	0	0	8000-9FFF

Figur 2.7 nedan visar kopplingarna för adressavkodninglogiken. Dataledningarnas anslutningar (D0 till D0, D1 till D1 osv.) är markerade med en svart buss. På motsvarande sätt har adressledningar A0-A11 markerats av en grå buss. Utöver dessa adressledningar, databussen och CS-signalerna ansluts även R/W mellan centralenheten och RWM.



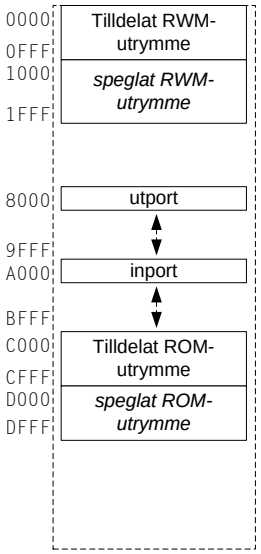
FIGUR 2.7 KOPPLINGAR FÖR ADRESSAVKODNINGSLOGIKEN

2.2.1 Ofullständig adressavkodning

Ofullständig adressavkodning innebär att vi utnyttjar det minsta antal adressbitar som krävs, i vårt aktuella system, för att avkoda adressbussen och skapa en unik CS-signal. Innebörden är att det krävs minimalt med logik (grindar) för att implementera systemet. Samtidigt kan det bli svårt, ibland omöjligt, att utvidga systemet. En ofullständig adressavkodning innebär en moduls adress (adresser) replikeras, i adressrymden.

Se figuren till höger, den beskriver resultatet av adressavkodningen i Figur 2.7. Vi ser att ROM-minneskapseln som har tilldelats adressintervallet C000-CFFF dessutom replikeras i adressintervallet D000-DFFF. På liknande sätt kommer RWM-kapseln att aktiveras för adresser i intervallen 0-FFF och 1000-1FFF. Detta beror på att vi utelämnat A13 i avkodningslogiken.

Då det gäller inporten, vars tilldelade adress är A000, kommer den att aktiveras av avkodningslogiken för *varje* adress i intervallet A000-BFFF, dvs. A000, A001, A002, A003 osv. t.o.m. BFFF. Slutligen aktiveras, på motsvarande sätt, utporten för varje adress i adressintervallet 8000 t.o.m. 9FFF.



2.8 ADRESSRUMMETS UTNYTTJANDE VID OFULLSTÄNDIG ADRESSAVKODNING

Exempel 2.1

Vi har ett system med 16 bitars adressbuss och 8 bitars databuss. Till centralenheten vill vi ansluta en ROM-modul (32 kbyte), en RWM-modul (16 kbyte) och en I/O-port (en inport och en utport). I/O-porten skall placeras på adress $4000, ROM-modulen placeras med start på adress $8000, RWM-modulen placeras med start på adress 0000 i adressrummet.

Konstruera adressavkodningslogiken, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna. Använd ofullständig adressavkodning. Alla ”chip select” signaler (CS_{ROM}, CS_{RWM}, CS_{IN} och CS_{UT}) är aktiva låga.

Lösning:
Vi har ett enkelt sätt att bestämma det antal adressbitar som krävs utgående från storleken av ett tilldelat adressutrymme, kom bara ihåg att 1 kbyte = 2^{10} och skriv:

32 kbyte = 32● 2^{10} = 2^5 ● 2^{10} = 2^{15}, dvs. 15 adressledningar kopplas direkt till kapseln.
16 kbyte = 16● 2^{10} = 2^4 ● 2^{10} = 2^{14}, dvs. 14 adressledningar kopplas direkt till kapseln.

Vi skapar en tabell med adressbussens ledningar (16 st.) och de ingående modulerna, vi bestämmer adressutrymmen för respektive modul enligt följande (startadress + storlek = slutadress+1):

ROM: (32 kbyte = $8000), slutadress = $8000+$8000-1 = $10000-1 = **$FFFF**
RWM: (16 kbyte =$4000), slutadress = 0+$4000-1 = **$3FFF**
I/O-port: (1 Byte), slutadress = $4000+1-1 = **$4000**

Modulernas start respektive slutadresser förs in i tabellen enligt följande:

Modul		**Adressbuss**															
		A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
ROM	$FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RWM	$3FFF	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IO-port	$4000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	$4000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

En ofullständig adressavkodning innebär att vi vill använda så få bitar som möjligt för att därigenom minimera adressavkodningslogiken. Vi börjar därför med att inspektera den mest signifikanta adressbiten (A15) och därefter successivt inspektera adressbitar med lägre ordning (A14, A13, A12 osv.). Då vi ser att adressbitarna bara förekommer för en enda modul har vi också tillräckligt många bitar för adressavkodningen.

Vi börjar med ROM-modulen och ser att detta är den enda kapsel som kräver att A15 =1. För både RWM och I/O-port ska denna adressbit vara 0, det räcker därför med att använda enbart denna bit i adressavkodningen för CS_{ROM}. CS- signalerna ska vara aktivt låga, varför vi får: $\overline{CS_{ROM}} = \overline{A15}$.

I tabellen ser vi att vi måste ta till både A15 och A14 för att skilja RWM-modulen från I/O-modulen. För RWM-modulen gäller att både A15 och A14 måste vara 0 och vi får därför: $\overline{CS_{RWM}} = \overline{\overline{A15} \cdot \overline{A14}}$.

Modul		**Adressbuss**		
		A15	A14	A13....
ROM	$FFFF	1	1	1
	$8000	1	0	0
RWM	$3FFF	0	0	1
	0	0	0	0
IO-port	$4000	0	1	0
	$4000	0	1	0

För IO-porten gäller att A15 ska vara 0 och A14 ska vara 1.

Samtidigt observerar vi att porten är dubbelriktad, vi använder därför $R/\overline{W}$-signalen som en extra adressledning och aktiverar CS_{IN} vid en läsning från adress $4000, respektive CS_{UT} vid en skrivning till adress $4000. För IO-porten får vi därför: $\overline{CS_{IN}} = \overline{\overline{A15} \cdot A14 \cdot R/\overline{W}}$ och $\overline{CS_{UT}} = \overline{\overline{A15} \cdot A14 \cdot \overline{R/\overline{W}}}$.

Vi sammanfattar resultaten:

$$\overline{CS_{ROM}} = \overline{A15}$$
$$\overline{CS_{RWM}} = \overline{\overline{A15} \cdot \overline{A14}}$$
$$\overline{CS_{IN}} = \overline{\overline{A15} \cdot A14 \cdot R/\overline{W}}$$
$$\overline{CS_{UT}} = \overline{\overline{A15} \cdot A14 \cdot \overline{R/\overline{W}}}$$

2.2.2 Fullständig adressavkodning

Vid fullständig adressavkodning används alla adressbitar som krävs för att aktivera en modul enbart inom dess avsedda adressområde. Här tillåter man alltså ingen replikering av adressutrymmet för minneskapslar eller I/O-portar. Det följer direkt att adressavkodningen kommer att kräva fler (större) grindar för att generera CS-signalerna.

Exempel 2.2

En fullständig adressavkodning innebär att samtliga bitar som inte direkt krävs för att adressera modulen internt används i avkodningslogiken. Om vi valt att i stället konstruera fullständig avkodningslogik i Exempel 2.1 hade lösningen sett ut på följande sätt:

Modul		**Adressbuss**															
		A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
ROM	$FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RWM	$3FFF	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IO-port	$4000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	$4000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Avkodningslogiken blir:

$$\overline{CS_{ROM}} = \overline{A15}$$

$$\overline{CS_{RWM}} = \overline{\overline{A15} \cdot \overline{A14}}$$

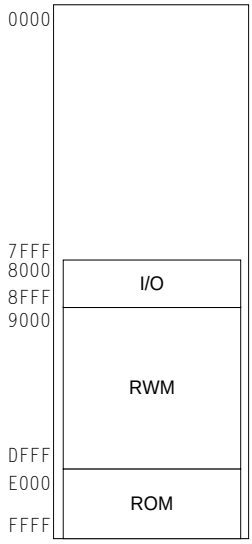
$$\overline{CS_{IN}} = \overline{\overline{A15} \cdot A14 \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \overline{A7} \cdot \overline{A6} \cdot \overline{A5} \cdot \overline{A4} \cdot \overline{A3} \cdot \overline{A2} \cdot \overline{A1} \cdot \overline{A0} \cdot R/\overline{W}}$$

$$\overline{CS_{UT}} = \overline{\overline{A15} \cdot A14 \cdot \overline{A13} \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \overline{A7} \cdot \overline{A6} \cdot \overline{A5} \cdot \overline{A4} \cdot \overline{A3} \cdot \overline{A2} \cdot \overline{A1} \cdot \overline{A0} \cdot \overline{R/\overline{W}}}$$

Det framgår att minneskapslarna i själva verket redan är fullständigt avkodade medan avkodningslogiken för portarna blir betydligt mer omfattande.

Fullständig adressavkodning innebär att adressrummet utnyttjas maximalt. I vårt systemexempel (Exempel 2.1 och Exempel 2.2) kan vi exempelvis bygga ut med fler portar eller eventuellt någon mindre minneskrets om vi använt fullständig adressavkodning redan från början.

Exempel 2.3



Vi har ett system med 16 bitars adressbuss och 8 bitars databuss. Konstruera fullständig adressavkodningslogik (ange booleska uttryck för CS-signalerna) så att adressrummet disponeras enligt figuren till vänster.

Alla ”chip select” signaler är aktiva låga.

Lösning:

Start och slutadresser framgår av figuren och vi kan ställa upp tabellen direkt:

Modul		**Adressbuss**															
		A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
ROM	$FFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$E000	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
RWM	$DFFF	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
	$9000	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
IO	$8FFF	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
	$8000	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

De booleska uttrycken för ”chip select” signalerna blir:

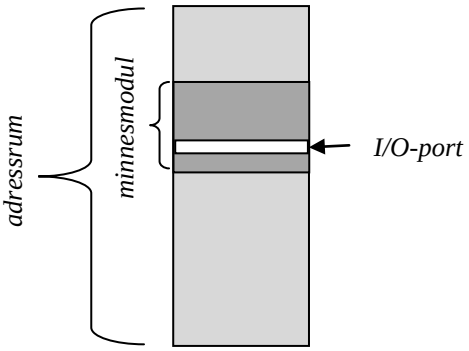
$$\overline{CS_{ROM}} = \overline{A15 \cdot A14 \cdot A13}$$

$$\overline{CS_{RWM}} = \overline{A15 \cdot \overline{A13} \cdot A12}$$

$$\overline{CS_{IO}} = \overline{A15 \cdot \overline{A14} \cdot \overline{A13} \cdot \overline{A12}}$$

2.2.3 Överlagrad minnesavbildning

I sammanhang där bara enstaka I/O-portar behövs och man i stället vill maximera det tillgängliga minnet kan man använda en överlagringsteknik där I/O-porten aktiveras i en del av adressrummet för någon minneskrets.



FIGUR 2.9 I/O-PORT I MINNESMODULS ADRESSRUM

Tekniken innebär att man först skapar en fullständig adressavkodning för I/O-porten och därefter använder dess invers som kompletterande villkor för minnesmodulens CS-signal.

Exempel 2.4

Vi har ett system med 20 bitars adressbuss och 8 bitars databuss. Till centralenheten finns ansluten en 128 kbytes ROM-modul med start på adress $20000. En I/O-port, som upptar 32 bytes, ska överlagras på adress $22000.

Konstruera adressavkodningslogiken, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna. Båda ”chip select” signalerna (CS_{ROM} och CS_{IO}) är aktiva låga.

Lösning:

Bestäm start och slutadresser och ställ upp en tabell:

Modul		**Adressbuss**																			
		A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
ROM	$3FFFF	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$20000	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IO-port	$2201F	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1	1	1
	$22000	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Ur tabellen får vi direkt den fullständiga avkodningslogiken för I/O-porten:

$$\overline{CS_{IO}} = \overline{\overline{A19} \cdot \overline{A18} \cdot A17 \cdot \overline{A16} \cdot \overline{A15} \cdot \overline{A14} \cdot A13 \cdot \overline{A12} \cdot \overline{A11} \cdot \overline{A10} \cdot \overline{A9} \cdot \overline{A8} \cdot \overline{A7} \cdot \overline{A6} \cdot \overline{A5}}$$

Avkodningslogiken för ROM-kapseln fås på samma sätt men vi grindar dessutom in inversen av $\overline{CS_{IO}}$.

$$\overline{CS_{ROM}} = \overline{\overline{A19} \cdot \overline{A18} \cdot A17 \cdot \overline{CS_{IO}}}$$

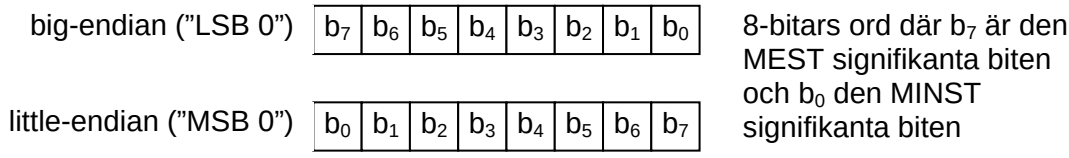
2.3 Buss-kommunikation

Datorns buss-kommunikation måste vara fastlagd i ett *protokoll*. Med protokoll menas helt enkelt regler för hur kommunikationen sker, dvs. hur information, *data*, utbytes mellan olika enheter i systemet. Regelverket inbegriper alltså överenskommelser om:

- ”Språket”, betydelsen av hög respektive låg logiknivå
- Organisation av bitarna i ett ord, ”endian bit order”
- Organisation av bytes i ett ord ”endian byte order”
- Tidsegenskaper, händelser korrekt ordnade i tid

I ett digitalt datorsystem är ”språket” enkelt, vi har logiknivån 1 och logiknivån 0. I bland betyder ”1” sant, annars betyder det falskt. Enda återstående alternativet är att ”1” betyder falskt och sålunda ”0” betyder sant. Fler kombinationer finns inte. För att ytterligare förenkla detta har vi infört uttrycket ”aktiv”. Med ”aktiv hög” menar vi att en signal ska betraktas som ”sann” om den är 1. Med uttrycket ”aktiv låg” är det precis tvärtom.

Då ettor och nollor sätts samman till kompletta ord som utväxlas via bussen måste man också ha fastlagt i vilken ordning bitarna lagras. Vi känner till exempel till formen där den mest signifikanta biten placeras i ordets första bit osv. med den minst signifikanta biten i ordets sista bit. Denna form kalla ”big-endian bit order”. Ordningen kan växlas, så att den minst signifikanta biten lagras i den första positionen och den mest signifikanta biten lagras i den sista positionen, Detta kallas då ”little-endian bit order”, se Figur 2.10.

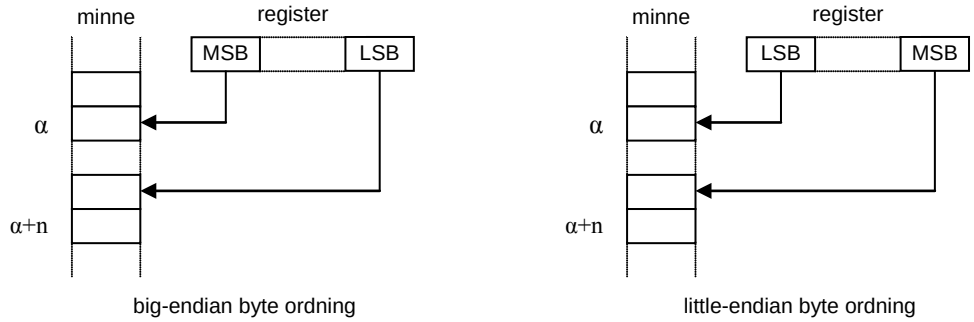


FIGUR 2.10 ”BIG/LITTLE –ENDIAN BIT ORDER”

Exempel 2.5 Bit-ordning

big-endian bit ordning är det absolute vanligast förekommande, bland andra Motorola (Freescale) 68xx, 68xxx, ColdFire,Intel x86. Det finns också processorer där bit-ordningen är programmerbar, exempelvis PowerPC.

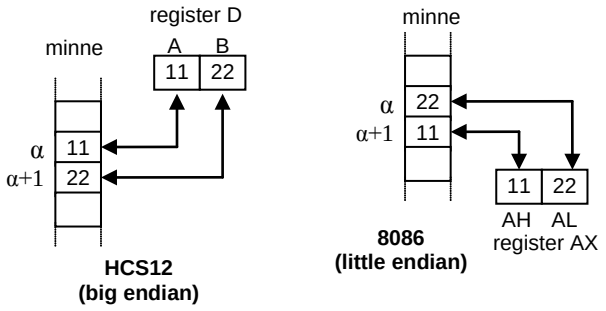
På motsvarande sätt kan delarna av ett ord bestående av flera bytes, organiseras med antingen den mest signifikanta byten först (*Most Significant Byte, MSB*), ”big endian”, eller den minst signifikanta byten först (*Least Significant Byte, LSB*), ”little endian”. Med ”först” menas då den lägsta minnesadressen. Se Figur 2.11



FIGUR 2.11 ORGANISATION VID ”BIG/LITTLE-ENDIAN BYTE ORDER”, *n*-BYTES ORD MED STARTADRESS α

Exempel 2.6 Motorola HCS12/Intel 8086 byte ordning

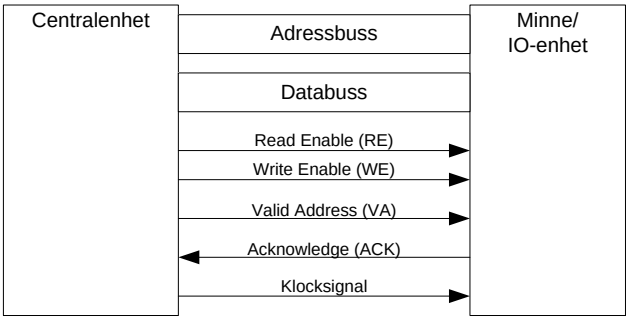
Det hexadecimala ordet $(1122)_{16}$ uppträder i minnet (adress α) respektive register enligt följande figurer:



Den sista viktiga aspekten på ett buss-protokoll är dess tidsmässiga egenskaper, dvs. att data utväxlas i en arbetstakt som är gemensam för alla enheter på bussen. Här skiljer vi mellan tre principiellt olika metoder, asynkron-, synkron- och multiplex-buss. Vi behandlar dessa i de följande avsnitten.

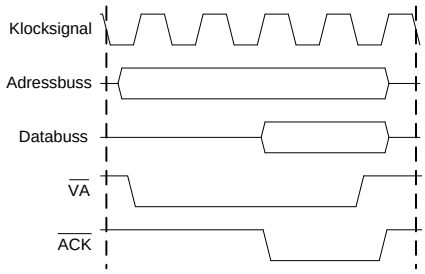
2.3.1 Asynkron buss

Med en *asynkron* buss används så kallade *handskakningssignaler* för kommunikationen (dataöverföringen). Med detta menas att centralenheten signalerar till de övriga enheterna att en adress nu finns på bussen genom att aktivera en speciell signal, *Valid Address* (VA). Övriga enheter kan läsa av adressbussen och därefter måste den enhet som adresseras generera en annan signal, *Acknowledge* (ACK) tillbaks till centralenheten, för att på så sätt tala om för centralenheten att adressen är igenkänd. Denna signal tolkas av centralenheten som att dataöverföringen nu kan ske. Följaktligen inväntar centralenheten svar från den enhet den kommunicerar med (Figur 2.12). Signalerna VA och ACK ingår i styrbussen.



FIGUR 2.12 ASYNKRONT BUSSPROTOKOLL

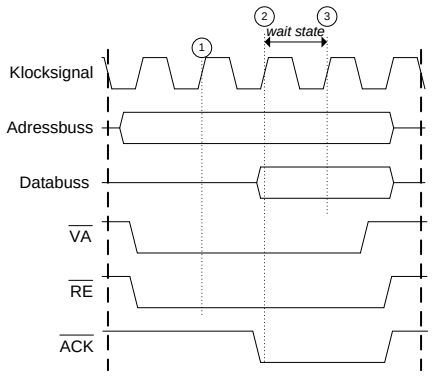
Figur 2.13 illustrerar principen för överföring via en asynkronbuss. Observera att de signaler som ingår i styrbussen är aktivt låga. Då periferienheten upptäckt den aktiva VA-signalen kommer den att avkoda adressbussen. Vid en läsning från minnet kommer periferienheten att placera data på databussen och därefter aktivera signalen ACK. När centralenheten upptäckt en aktiv ACK-signal läser den in data från databussen. Centralenheten indikerar sedan att den läst databussen genom att återställa VA. Slutligen avlägsnar centralenheten adressen på adressbussen. Periferienheten som ser att VA inte längre är aktiv avlägsnar ACK-signalen och data från databussen.



FIGUR 2.13 SIGNALERING VID ÖVERFÖRING PÅ EN ASYNKRON BUSS

Protokollet fungerar på liknande sätt vid en skrivning till periferienheten. I detta fall innebär VA-signalen att såväl innehållet på adressbussen som på databussen är giltigt. Periferienheten avkodar, precis som tidigare, adressbussen, och läser därefter databussens innehåll. Slutligen aktiverar periferienheten ACK-signalen som signal till centralenheten att skrivcykeln kan avslutas.

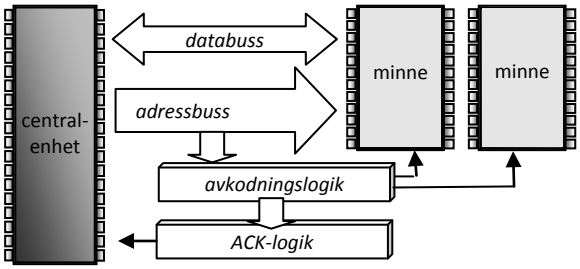
Vår principiella beskrivning i Figur 2.13 är visserligen korrekt men inte fullständig. I verkligheten sker alla tillståndsövergångar då systemet klockas, vanligtvis vid en positiv klockflank. Det innebär att signaländringar på den asynkrona bussen inte är momentant kända. Låt oss illustrera detta med en ny bild av hur en läscykel kan gå till (Figur 2.14).



FIGUR 2.14 LÄSCYKEL MED ASYNKRON BUSS

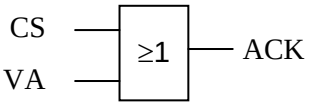
Vid ”1” börjar periferienheten avkoda adressen från adressbussen, detta tar en kort stund, varefter databussen drivs av periferienheten och ACK-signalen aktiveras. ACK-signalen upptäcks av centralenheten vid ”2” och databussen läses av vid nästa positiva klockflank ”3”. Antalet klockcykler från det att centralenheten upptäcker ACK, tills data klockas in från databussen kallas *väntecykler* (”*wait states*”).

Centralenhetens kommunikation med minnesenheter måste oftast utformas på speciellt sätt eftersom minneskretsar normalt inte är asynkrona, dvs. de genererar ingen ACK-signal. Man måste då införa extra logik som genererar ACK-signal till centralenheten när minnet adresseras.



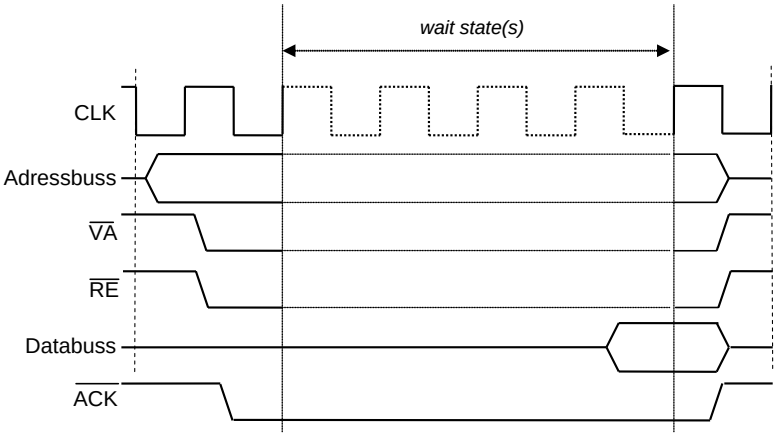
FIGUR 2.15 INKOPPLING AV MINNE SOM SAKNAR ACK-SIGNAL TILL ASYNKRON BUSS

ACK-logiken måste alltså utformas på så sätt att tidsegenskaperna hos det använda minnet respekteras. Detta kräver då att logiken blir speciell för någon familj av minneskretsar. Dessbättre är detta inget större problem i praktiken eftersom centralenheter som arbetar med asynkronbuss oftast kan programmeras för att sätta in väntecykler för olika regioner av adressrummet. ACK-logiken kan då utformas extremt enkel (Figur 2.16) medan tidsegenskaperna kan uppfylls genom en lämpligt vald initieringssekvens för centralenheten (Figur 2.17).



där CS genereras av adressavkodningslogiken för minneskapseln

FIGUR 2.16 FÖRENKLAD ACK-LOGIK FÖR MINNESKRETS



FIGUR 2.17 PROGRAMMERBART ANTAL VÄNTECYKLER

De flesta centralenheter med asynkron buss bevakar att VA-signalen inte är aktiv för länge, typiskt något hundratal klockcykler vilket är tillräckligt för att alla olika typer av kringenheter ska kunna hinna svara. Vid utebliven ACK-signal, dvs adressbussen anger en adress som inte finns representerad hos någon annan enhet kommer ett internfel ”bus error” att genereras.

Exempel 2.7

Vi har ett asynkront system med 24 bitars adressbuss och 8 bitars databuss. Till centralenheten finns ansluten en 512 kbytes ROM-modul med start på adress $0 och en 512 kBytes RWM-modul med start på adress $F80000.

Konstruera adressavkodningslogiken, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna och en ACK-signal. Båda ”chip select” signalerna (CS_{ROM}, CS_{RWM}) och ACK-signalen är aktiva låga.

Lösning:

Bestäm start och slutadresser och ställ upp en tabell:

Modul		**Adressbuss**																							
		A23	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
ROM	$07FFFF	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RWM	$FFFFFF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$F80000	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Ur tabellen får vi direkt den fullständiga avkodningslogiken ROM och RWM moduler:

$$\overline{CS_{ROM}} = \overline{\overline{A23} \cdot \overline{A22} \cdot \overline{A21} \cdot \overline{A20} \cdot \overline{A19} \cdot \overline{VA}}$$

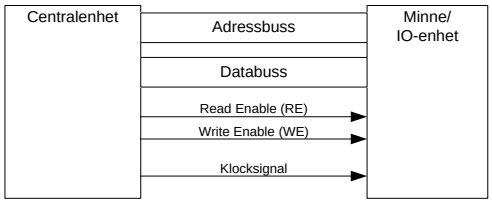
$$\overline{CS_{RWM}} = \overline{A23 \cdot A22 \cdot A21 \cdot A20 \cdot A19 \cdot \overline{VA}}$$

ACK-signalen (utan ”wait-states”) bildas på enklaste sätt:

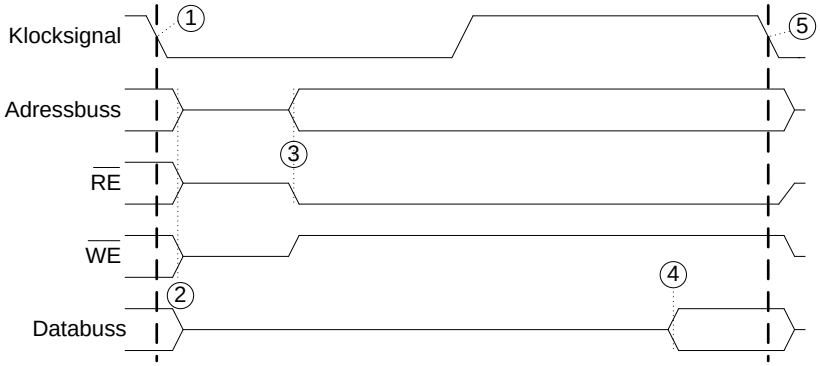
$$\overline{ACK} = \overline{CS_{ROM} + CS_{RWM}}$$

2.3.2 Synkron buss

En buss sägs arbeta *synkront* om signalerna överförs vid en förutbestämd tidpunkt. Fördelen med synkron buss är att handskakningssignaler inte behövs längre. Nackdelen är att det kan vara svårt att blanda olika typer av minnen etc. om dessa har olika tidsegenskaper. Vid dataöverföring på en synkron buss *förutsätter* centralenheten att den andra enheten hinner med i den arbetstakt som används och det krävs inte längre någon ACK- signal.

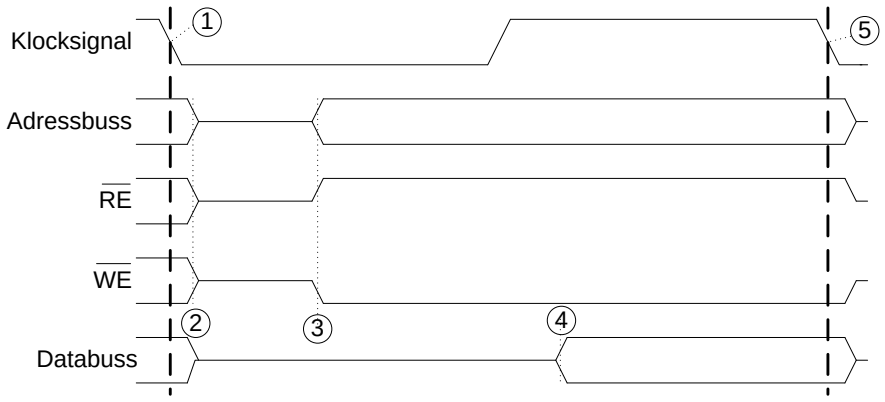


FIGUR 2.18 SIGNALER VID ÖVERFÖRING PÅ SYNKRON BUSS



FIGUR 2.19 LÄSCYKEL PÅ SYNKRON BUSS

1. Läscykeln inleds med en negativ flank hos klocksignalen.
2. Signaler som var giltiga under föregående cykel tas bort och bussarna är nu i odefinierade tillstånd.
3. Centralenheten sätter upp adress och styrsignaler för minnet.
4. Minnet har avkodat adressbussen och lägger ut data på databussen.
5. Läscykeln avslutas med en negativ flank hos klocksignalen. Data klockas in till centralenheten från databussen.



FIGUR 2.20 SKRIVCYKEL PÅ SYNKRON BUSS

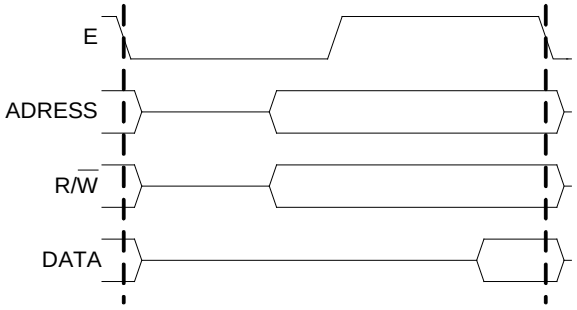
1. Skrivcykeln inleds med en negativ flank hos klocksignalen.
2. Signaler som var giltiga under föregående cykel tas bort och bussarna är nu i odefinierade tillstånd.
3. Centralenheten sätter upp adress och styrsignaler för minnet.
4. Centralenheten lägger ut data på databussen, adressbussen avkodas och aktiverar minneskapsel.
5. Skrivcykeln avslutas med en negativ flank hos klocksignalen. Data klockas in till minnet från databussen.

Exempel 2.8

Vi har ett synkront system med 16 bitars adressbuss och 8 bitars databuss. Data klockas i systemet vid negativ flank hos signalen E.

Till centralenheten ska anslutas:

- 8 kbyte ROM med start på adress $E000
- 16 kbyte RWM med start på adress $8000
- 512 byte I/O, med start på adress $E000



Konstruera fullständig adressavkodningslogik, dvs. ange booleska uttryck för ”chip select” (CS)- signalerna. Alla ”chip select” signalerna (CS_{ROM}, CS_{RWM} och CS_{IO}) är aktiva låga.

Lösning:

Eftersom avkodningen ska vara fullständig bestämmer vi först antalet adressledningar som krävs för de respektive modulerna:

ROM: 8 kbyte = 8● 2^{10} = 2^3 ● 2^{10} = 2^{13}, dvs. 13 adressledningar kopplas direkt till ROM-kapseln.
RWM: 16 kbyte = 16● 2^{10} = 2^4 ● 2^{10} = 2^{14}, dvs. 14 adressledningar kopplas direkt till RWM-kapseln.
I/O: 512 byte = 2^9, dvs. 9 adressledningar används för en generell ”I/O-select” signal.

Nu för vi in start- och slutadresser i en tabell:

Modul		**Adressbuss**															
		A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
ROM	$FFFF	1	1	1	Kopplas direkt till ROM-kapsel												
	$E000	1	1	1													
RWM	$BFFF	1	0	Kopplas direkt till RWM-kapsel													
	$8000	1	0														
I/O	$E1FF	1	1	1	0	0	0	0	Används för att bilda ytterligare CS-signaler för enskilda I/O-enheter								
	$E000	1	1	1	0	0	0	0									

Av tidsdiagrammet över bussens karakteristik kan vi utläsa att adressbuss och R/W-signal är giltig åtminstone den tid som E-signalen är hög. Vi använder därför denna som en *Valid Address*-signal, dvs VA=E (Anm: VA är alltså aktiv hög).

Vi börjar med I/O-blockets generella CS-signal:

$$\overline{CS_{IO}} = \overline{A15 \cdot A14 \cdot A13 \cdot \overline{A12} \cdot \overline{A12} \cdot \overline{A10} \cdot \overline{A9} \cdot VA}$$

därefter får vi enkelt CS-signalen för ROM-modulen:

$$\overline{CS_{ROM}} = \overline{A15 \cdot A14 \cdot A13 \cdot CS_{IO}}$$

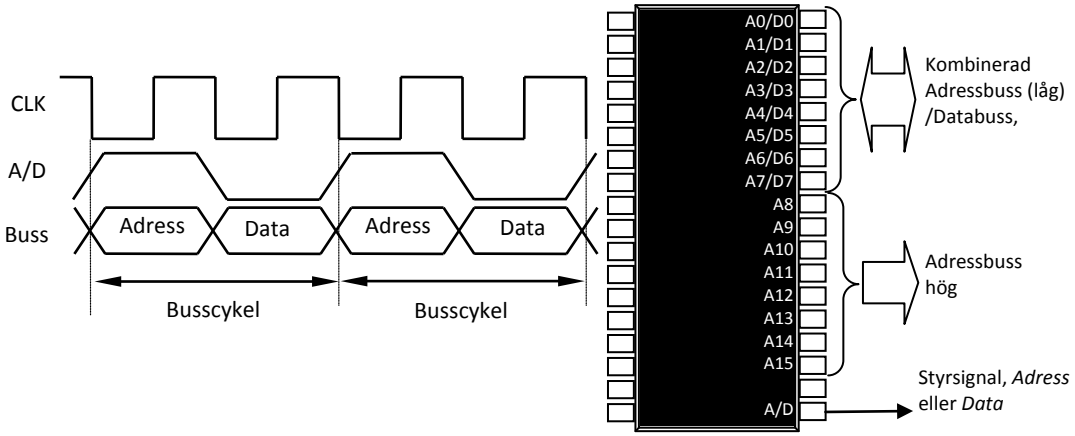
slutligen RWM-modulen:

$$\overline{CS_{RWM}} = \overline{A15 \cdot \overline{A14} \cdot VA}$$

2.3.3 Multiplexad buss

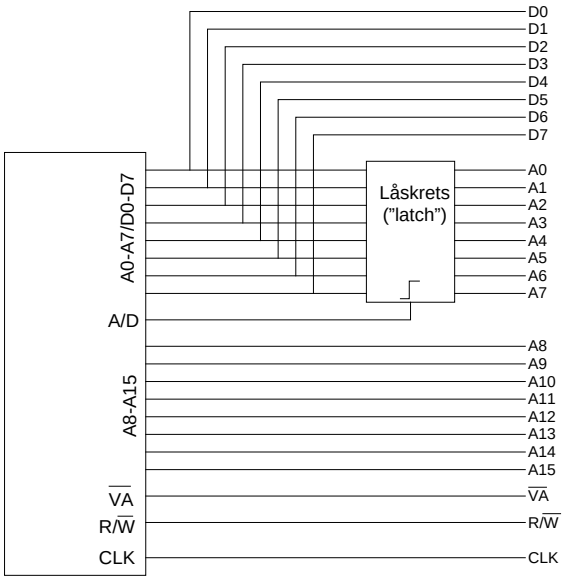
Vanligtvis är adressbussar och databussar fysiskt separerade. För att reducera antalet pinnar (signaler) i en processor's kapsel kan man använda *multiplexad* buss. Med detta menas att bussen används för att växelvis överföra adresser respektive data.

I Figur 2.21 illustreras ett multiplexat system med 16 bitars adressbuss och 8 bitars databuss och med den *fysiska* busbredden 16 bitar. (Användes bland annat i den första 8 bitars mikroprocessorn Intel 8080). Som figuren visar finns en signal A/D som anger huruvida bussen används för att överföra adress eller data. Hela adressen överförs under den första busscykeln. Under den andra busscykeln används bara de 8 minst signifikanta bitarna för data.



FIGUR 2.21 ÖVERFÖRING MED EN MULTIPLEXAD BUSS, 16-BITAR ADRESS, 8-BITAR DATA

I Figur 2.22 visas hur en multiplexad processor kopplas samman med periferienheter (minne). Observera speciellt låskretsen, då A/D signalen växlar från låg till hög nivå, "låses" innehållet på kretsens ingångar, dessa utgör då adressbussens värde för A0-A7. I nästa klockcykel, går A/D-signalen låg och indikerar att data nu finns (vid skrivning) eller ska läggas ut av minnet (vid läsning).



FIGUR 2.22 GRÄNSSNITT MELLAN MULTIPLEXAD PROCESSOR OCH MINNE