

Notater: INF2270

Veronika Heimsbakk
veronahe@student.matnat.uio.no

10. juni 2014

Innhold

1	Binære tall og tallsystemer	3
1.1	Tallsystemer	3
1.2	Konvertering	3
1.3	Komplement	4
1.4	Binær lagring og registre	5
2	Boolsk algebra	5
2.1	Definisjoner	5
2.2	Boolske funksjoner	5
2.3	Mintermer og makstermer	6
2.3.1	Konvertering mellom kanoniske former	7
2.4	Andre boolske uttrykk	7
3	Port minimering	7
3.1	Karnaugh diagram	7
4	Kombinatorisk logikk	8
4.1	Binær adder	8
4.1.1	Half adder	8
4.1.2	Full adder	8
5	Sekvensiell logikk	8
5.1	SR-latch	9
5.2	D-latch	9
5.3	Flip-flop	10
5.4	D flip-flop	10
5.5	JK flip-flop	10

5.6	T flip-flop	11
5.7	Tilstandsmaskin	11
6	Datamaskinarkitektur	11
6.1	Von Neumann Arkitektur	11
6.1.1	Data- og instruksjonsbus	12
6.2	Aritmetisk logisk enhet (ALU)	13
7	Input/Output	13
7.1	Bus	13
7.2	Eksterne hendelser	14
7.2.1	Polling	14
7.2.2	Avbrudd	14
8	Pipeline	15
8.1	Én-sykel implementasjon	15
8.2	Multicycle	15
8.3	Pipelining	15
8.4	Speedup	15
8.5	Komplikasjoner ved pipelining	16
8.5.1	Resource Hazard	16
8.5.2	Data Hazard	16
8.5.3	Control Hazard	16
9	Minnehierarki	16

Introduksjon

Dette er mine eksamensnotater i kurset INF2270 – Datamaskinarkitektur som jeg tok våren 2014 ved Institutt for informatikk, Universitetet i Oslo. Notatene er basert på forelesningsfoiler, egne notater (hovedsakelig fra INF1400 – Digital design) og pensumbøkene. Disse inneholder sikkert en del feil og mangler, ikke stol på de. Men send meg gjerne en e-post om det er noen innvendinger.

1 Binære tall og tallsystemer

Elektroniske signaler (spenning, volt) representerer informasjon i et digitalt system. Signalene er representert ved to verdier, derav binært. Et binært tall, kalt bit, har to verdier: 0 og 1.

1.1 Tallsystemer

Desimaltallet 7392 er en kort notasjon for:

$$7 \times 10^3 + 3 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$$

Det binære tallet 11010,11 vist med multiplikasjon og koeffisient 2:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 26,75_{10}$$

Et oktalt nummer:

$$127,4_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} = 87,5_{10}$$

Et hexadesimalt nummer:

$$B65F_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = 46687_{10}$$

1.2 Konvertering

Konvertere desimalnummer 41 til binær. Kan leses slik:

$$41_{10} = a_5 \times a_4 \times a_3 \times a_2 \times a_1 \times a_0 = 101001_2$$

Dette ved å se på tallene i rest (tabell 2). Det første (a_0) er *least significant byte* (LSB). Binære tall kan bli konvertert opp til hexadesimale tall ved å gruppere tallene i grupper på 4:

$$10110001101011,1111 = 2C6B, F_{16}$$

Desimal base 10	Binær base 2	Oktal base 8	Hexadesimal base 16
00	0000	00	0
01	0001	01	1
02	0010	02	2
03	0011	03	3
04	0100	04	4
05	0101	05	5
06	0110	06	6
07	0111	07	7
08	1000	10	8
09	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tabell 1: Tall med forskjellige baser.

Dette får vi fordi 10 er 2, 1100 er 12 (eller C), 0110 er 6, 1011 er B og 1111 er F. Når man grupperer binære tall, så starter man alltid med gruppen lengst til høyre. Derfor inneholder den første gruppen bare to siffer. Denne metoden fungerer for oktale tall også, da med grupper på tre og tre.

	Rest	Koeffisient
$41 : 2 = 20 + \frac{1}{2}$	$\frac{1}{2}$	$a_0 = 1$
$20 : 2 = 10 + 0$	0	$a_1 = 0$
$10 : 2 = 5 + 0$	0	$a_2 = 0$
$5 : 2 = 2 + \frac{1}{2}$	$\frac{1}{2}$	$a_3 = 1$
$2 : 2 = 1 + 0$	0	$a_4 = 0$
$1 : 2 = 0 + \frac{1}{2}$	$\frac{1}{2}$	$a_5 = 1$

Tabell 2: Konvertering fra desimal til binær.

1.3 Komplement

1-er komplementet av det binære tallet 1011000 er 0100111. 1-er komplementet til binære tall formes ved å bytte 1-ere til 0 og 0-ere til 1. Komplementet av komplementet gir den originale verdien.

1.4 Binær lagring og registre

En *binær celle* er en enhet som har to tilstander og er kapabel til å lagre en bit (0 eller 1) av informasjon. Et *register* er en gruppe av binære celler. Et register med 16 celler kan være i en av 2^{16} mulige tilstander.

2 Boolsk algebra

2.1 Definisjoner

En *mengde* elementer er hvilken som helst samling objekter, som har som regel en felles egenskap. For eksempel hvis S er en mengde elementer, og x og y er objekter. Da betyr notasjonen $x \in S$ at x er et *element* i S . Og $y \notin S$ betyr at y ikke er et element i S . En mengde elementer skrives slik: $A = \{1, 2, 3, 4\}$.

2.2 Boolske funksjoner

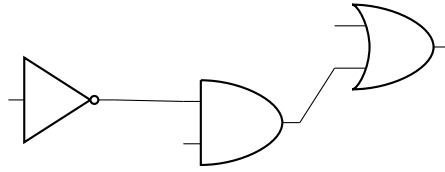
Boolsk algebra opererer med binære variable og logiske operatorer. Ta for eksempel funksjonen $F_1 = x + y'z$. Denne funksjonen kan leses som «x ELLER IKKE-y OG z» (se sannhetsverditabellen i tabell 3). Antall rader i sannhetsverditabellen er 2^n , hvor n er antall variable i funksjonen. En boolsk funksjon kan blir oversatt til en *krets* (se figur 1). De logiske portene og deres funksjoner kan ses i tabell 5.

x	y	z	F_1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Tabell 3: Sannhetsverditabell for funksjonen F_1 .

Ved å manipulere boolske uttrykk kan man av og til få enklere uttrykk for samme funksjonen. Da reduserer man antall porter som trengs for å gjøre det samme. For eksempel:

$$F_2 = x'y'z + x'yz + xy'$$



Figur 1: Kretsen til funksjonen F_1 .

Kan bli redusert ved å gjøre dette:

$$\begin{aligned}
 F_2 &= x'y'z + x'yz + xy' \\
 &= x'z(y' + y) + xy' \\
 &= x'z + xy'
 \end{aligned} \tag{1}$$

Her er det viktig å huske enkelte postulater og teoremer (tabell 4).

	a	b
Postulat 2	$x + 0 = x$	$x \cdot 1 = x$
Postulat 5	$x + x' = 1$	$x \cdot x' = 0$
Teorem 1	$x + x = x$	$x \cdot x = x$
Teorem 2	$x + 1 = 1$	$x \cdot 0 = 0$
Teorem 3	$(x')' = x$	
Teorem 3, kommutativ	$x + y = y + x$	
Teorem 4, assosiativ	$x + (x + z) = (x + y) + z$	$x(yz) = (xy)z$
Teorem 4, distributiv	$x(y + z) = xy + xz$	$x + yz = (x + y)(x + z)$
Teorem 5, DeMorgan	$(x + y)' = x'y'$	$(xy)' = x' + y'$
Teorem 6, absorbering	$x + xy = x$	$x(x + y) = x$

Tabell 4: Postulater og teoremer.

2.3 Mintermer og makstermer

Det fins fire mulige kombinasjoner av x og y : $x'y'$, $x'y$, xy' og xy . Hver av disse AND termene er kalt *mintermer*, eller standard produkt. n variable kan kombineres til 2^n mintermer. Hver minterm er satt sammen av en AND term av de n variablene.

Makstermer er de n variablene formet av en OR term. Det er også 2^n kombinasjoner av makstermer, eller standard summer.

Boolske funksjoner som er uttrykt som en sum av mintermer eller et produkt av makstermer, sies å være på *canonical form*. Se tabell 6 for eksempel på minterm og maksterm.

2.3.1 Konvertering mellom kanoniske former

Komplementet av en funksjon uttrykt som summen av mintermer er lik summen av mintermene som mangler fra den originale funksjonen. For eksempel funksjonen:

$$F(A, B, C) = \sum(1, 4, 5, 6, 7)$$

Denne funksjonen har komplement som kan uttrykkes som:

$$F'(A, B, C) = \sum(0, 2, 3) = m_0 + m_2 + m_3$$

Hvis vi tok komplementet av F' med DeMorgans teorem, får vi F i en annen form:

$$F = (m_0 + m_2 + m_3)' = m'_0 \cdot m'_2 \cdot m'_3 = M_0 M_2 M_3 = \prod(0, 2, 3)$$

Derfor holder følgende:

$$m'_j = M_j$$

2.4 Andre boolske uttrykk

3 Port minimering

Gate-level minimization er design oppgaven ved å finne den optimale implementasjonen av porter.

3.1 Karnaugh diagram

Karnaugh diagram er bygd opp av ruter, hvor hver rute representerer en minterm fra funksjonen som skal minimeres. K-diagram for funksjonen $m_1 + m_2 + m_3 = x'y + xy' + xy = x + y$:

		y	
		0	1
x	0	0	1
	1	1	1

I et 3-variabel K-diagram er mintermene arrangert tilnærmet lik Grey code. Som dette:

	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

4 Kombinatorisk logikk

En kombinatorisk krets består av en samtrafikk av logiske kretser. For n variable, er det 2^n mulige kombinasjoner av binære inputs. For hver mulige input-kombinasjon, er det en mulig verdi for output variabelen. Dette illustreres ved en sannhetsverditabell.

4.1 Binær adder

En kombinatorisk krets som legger sammen to bits er kalt *half adder*, en som gjør dette med tre bits kalles *full adder*.

4.1.1 Half adder

Denne kretsen trenger to input og to output. Boolsk funksjon for output variablene:

$$\begin{aligned} S &= x \oplus y \\ C &= xy \end{aligned} \tag{2}$$

Hvor S er summen og C er *carry*. C er 1 bare når begge inputene er 1. Summen representerer LSB av summen.

4.1.2 Full adder

Å legge sammen n -binære tall krever bruken av en full adder. Prosessen går på en bit-by-bit basis, høyre til venstre, starter med LSB. En full adder er en kombinatorisk krets som former symmen av tre bit. Tre input og to outputs. Funksjon:

$$\begin{aligned} S &= x'y'z + x'yz' + xy'z' + xyz \\ C &= xy + xz + yz \end{aligned} \tag{3}$$

5 Sekvensiell logikk

I synkrone sekvensielle kretser skjer endringene i output samtidig med endringen i et *klokkesignal*. I asynkrone sekvensielle kretser skjer endringene i output uten noe klokkesignal. Nesten alle kretser er synkrone. Et klokkesignal er et digitalt signal som veksler mellom 0 og 1 med fast takt.

Den omvendte av klokkeperioden kalles klokkefrekvensen:

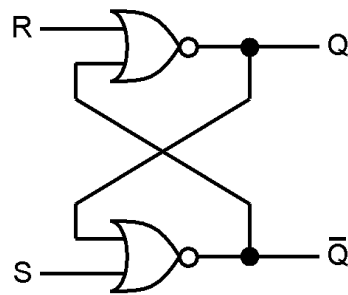
$$\text{frekvens} = \frac{1}{\text{klokkeperioden}}$$

Man ønsker så høy klokkefrekvens som mulig, fordi hver enkelt operasjon bruker da kortere tid. Maksimal klokkefrekvens bestemmes av flere faktorer:

- Lengde på signalveiene
- Last
- Forinkelse gjennom porter (delay)

Merk; hastighet er ikke direkte proporsjonal med klokkefrekvens.

5.1 SR-latch



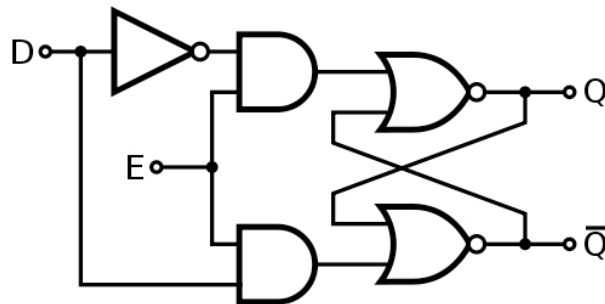
Figur 2: SR-latch.

1. Kretsen skal sette Q til 1 hvis den får 1 på inngang S. Når inngang S går tilbake til 0, skal Q forbli på 1.
2. Kretsen skal resette Q til 0 når den får 1 på inngang R. Når inngang R går tilbake til 0, skal Q forbli på 0.
3. Tilstanden 1 på både S og R brukes normalt ikke.

5.2 D-latch

Dataflyten gjennom en D-latch kontrolleres av et klokkesignal.

1. Slipper gjennom et digitalt signal så lenge klokkeinngangen er 1.
2. I det øyeblikket klokkeinngangen går fra 1 til 0 låser utgangen seg på sin nåværende verdi. Forandringer på inngangen vil ikke påvirke utgnagsverdien så lenge klokkesignalet er 0.



Figur 3: D-latch.

5.3 Flip-flop

Flip-floper kommer i to variander: positiv flanketrigget og negativ flanketrigget. På en positiv flanketrigget flip-flop kan utgangen kun skifte verdi i det øyeblikket klokkesignalet går fra 0 til 1. På en negativ flanketrigget flip-flop kan utgangen kun skifte verdi i det øyeblikket klokkesignalet går fra 1 til 0.

5.4 D flip-flop

En positiv flanketrigget D flip-flop sampler verdien på D i det øyeblikket Clk går fra 0 til 1. Denne verdien holdes fast på utgangen helt til neste positive flanke.

For flip-floper kan man generelt beskrive neste utgangsverdi $Q(t+1)$ som funksjon av nåværende inngangsverdier, og nåværende utgangsverdi $Q(t)$.

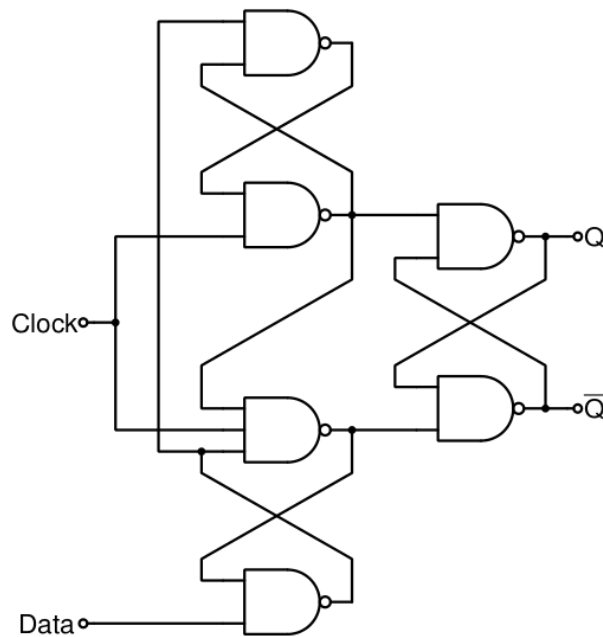
En positiv flanketrigget D flip-flop kan lages av to D-latcher (Master-Slave).

5.5 JK flip-flop

En JK flip-flop har følgende egenskaper:

- $J=0, K=0$: Utgang låst.
- $J=0, K=1$: Resetter utgang til 0.
- $J=1, K=0$: Setter utgang til 1.
- $J=1, K=1$: Inverterer utgang.

Utgangen kan kun forandre verdi på stigende klokkeflanke. En JK flip-flop er den mest generelle vi har.



Figur 4: D flip-flop.

5.6 T flip-flop

En T flip-flop har følgende egenskaper:

- $T=0$: Utgang låst.
- $T=1$: Inverterer utgang.

Utgangene kan kun forandre verdi på stigende klokkeflanke.

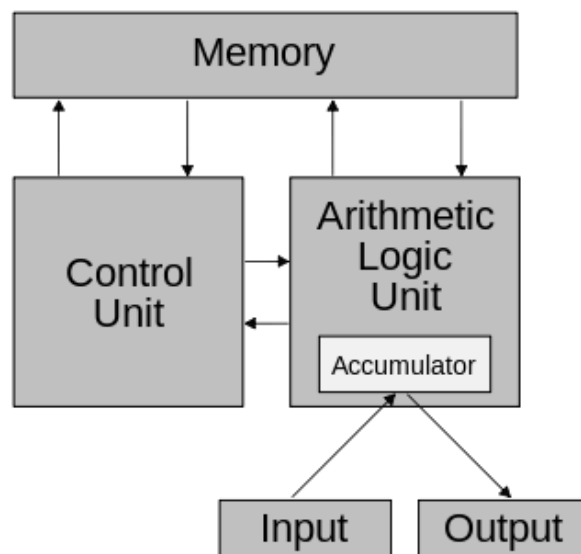
5.7 Tilstandsmaskin

En *tilstandsmaskin* er et sekvensielt system som gjennomløper et sett med tilstander styrt av verdiene på inngangssignalene.

6 Datamaskinarkitektur

6.1 Von Neumann Arkitektur

John von Neumann publiserte i 1945 en modell for datamaskinarkitektur som brukes fortsatt den dag i dag. Hovedbidraget hans og unikheten i denne



Figur 5: Von Neumann arkitektur.

modellen er å bruke et enkelt minneelement som både skal brukes for program og data.

I *control uniten* (figur 5) har man flere registre:

IR Instruksjonsregister. Her lagres maskinkoden av instruksjonen som skal eksekveres.

PC Program counter. Her lagres minneadressen for den neste maskinkodeinstruksjonen.

MAR Memory address register. Halvparten av registeret er dedikert til å lagre minneadressen som det skal leses fra eller skrives til fra CPU.

MBR Memory buffer register. Den andre halvparten av registeret som lagrer selve dataen som er lest fra minne eller skal skrive inn på minne fra CPU.

6.1.1 Data- og instruksjonsbus

Bus er kommunikasjonskanalen mellom registre, funksjonelle enheter (ALU), minne og I/O enheter. Bus kan deles mellom flere enheter, men kun en som kan sende av gangen. I en von Neumann arkitektur er det en bus mellom minne og CPU som skal både overføre instruksjon og data, dette vil da være flaskehalsen. Internt i en CPU er det en eller flere bus(er) som overfører data mellom interne registre.

6.2 Aritmetisk logisk enhet (ALU)

Den delen av CPU hvor logiske og aritmetiske beregninger utføres. Operasjoner som en ALU skal utføre er eksempelvis:

- AND
- OR
- XOR
- Addisjon
- Subtraksjon

Da trenger man følgende byggeblokker: fulladder, multiplekser, AND, OR og NOT. Moderne CPUer kan inneholde flere ALUer. ALUer kan designes til å håndtere flere funksjoner per trinn og mer komplekse. Alternativt kan man bruke software aktivt til å designe slik at en ALU kan utføre komplekse operasjoner over flere trinn. Ulemper ved å sette kompleksitet i hardware er høyere kostnader i strømforbruk, areal og produksjonskostnad. Designet av en ALU spiller en stor rolle med hensyn på CPU sin ytelse, da det mest komplekse operasjonen setter maskimal klokkefrekvens.

7 Input/Output

7.1 Bus

En bus består av datalinjer og kontrollinjer. Kontrollinjene regulerer bruken av busen, spesifiserer hva busen inneholder, synkroniserer overføring osv. Datalinjene inneholder de data som sendes over busen, både adresser og faktiske data.

Buser er enten *synkrone* eller *asynkrone*.

Synkron Endringer på busen skjer etter en fast protokoll, relativt til et klokkesignal i kontrollinjene.

- Raskere enn asynkrone.
- Knytter sammen enheter med samme klokkehastighet.
- Enhetene må ligge fysisk nær hverandre.

Asynkron Ingen klokkesignal blant kontrollinjene. Overføring av data skjer etter regler avtalt mellom enhetene.

- Knytter sammen enheter med ulik hastighet.
- Gir færre begrensninger i bus-lengde.
- Mer komplisert protokoll for synkronisering.

Asynkrone buser og handshaking Handshaking brukes for å koordinere sending av data mellom sender og mottaker. Kontrollinjer brukes for å utveksle informasjon om hvor langt de to enhetene har kommet (eksempel `read_request`, `data_ready`, `ack`).

7.2 Eksterne hendelser

Prosessoren kan lese innholdet av rad- og kolonneregisteret for å finne ut hvilken tast som er trykket ned. Hvordan finne ut når en tast er trykket ned? Kan løses på to måter: polling og avbrudd.

7.2.1 Polling

Prosessoren kan med jevne mellomrom lese av innholdet i rad- og kolonneregisterne og sjekke om det er en endring fra forrige gang.

Fordel Enkelt å implementere (endless-loop, les av og sjekk forrige verdi).

Ulempe Prosessoren får ikke gjort så mye annet enn å sjekke disse registerne hele tiden.

7.2.2 Avbrudd

I stedet for å sjekke jevnlig, kan tastaturet selv si fra at en tast er trykket ned. Prosessoren finner ut hvilken tast ved å sammenligne gammelt og nytt innhold i rad- og kolonneregisteret.

Fordel Prosessoren kan løse andre oppgaver enn å bare vente på at en tast skal trykkes ned.

Ulempe Det kreves mer av hardware; må ha egne signaler inn til prosessoren som kan brukes til for eksempel å si fra at en tast er trykket ned.

Ved avbrudd så avslutter prosessoren den instruksjonen den holder på med å eksekvere. Alle registre som er i bruk må lagres unna. Avhengig av hvilken kilde som genererte avbruddet vil det bli startet opp en avbruddsrutine som prosesserer avbruddet. Når avbruddsrutinen er ferdig, gjenopprettes registerne som ble lagret unna, og prosessoren fortsetter å eksekvere det programmet den kjørte før avbruddet skjedde.

8 Pipeline

8.1 Én-sykel implementasjon

1. Alle instruksjoner bruker en klokkesykel Klokkeperioden må være like lang som den lengste forsinkelsen for enhver instruksjon gjennom data-stien. Resultat: reaskere instruksjoner tvinges til å bruke mer tid enn nødvendig. Mulig løsning: klokke med variabel periode. Vurdering: vanskelig å implementere.

2. Bruke flere identiske komponenter Trenger flere like komponenter som egentlig gjør samme jobb. Resultat: trenger mer plass på CPUen og øker effektforbruket. Mulig løsning: endre kontroll-logikk slik at samme komponent kan brukes til flere oppgaver. Vurdering: vanskelig, umulig eller lite effektivt hvis man krever at alle instruksjoner skal ta en klokkesykel.

3. Instant memory og data memory aksesseres samtidig Må kunne aksessere to forskjellige lokasjoner i samme minne-enhet samtidig. Resultat: må enten ha to separate minne-enheter, eller kunne adressere flere ord samtidig i en minne-enhet. Mulig løsning: operere med en minne-enhet for program, og en annen for data. Vurdering: lite fleksibelt ved blant annet dynamisk data-allokering, og mer plasskrevende.

8.2 Multicycle

Multicycle bruker mer enn en klokkesykel per instruksjon hvis nødvendig. De ulike trinnene i en instruksjon kan dele samme komponent. Trenger ikke separat data- og instruksjonsminne. Trenger ekstra registre for å mellomlagre data.

8.3 Pipelining

Innfører samlebåndsprinsipp for eksekvering av instruksjoner. Hver instruksjon må splittes opp i uavhengige deler som utføres etter hverandre. Hver subinstruksjon kan utføres uavhengig av de andre subinstruksjonene. Neste instruksjon settes i gang før forrige instruksjon er helt ferdig.

8.4 Speedup

Det å ha en fire trinns pipeline betyr ikke at man får fire ganger raskere prosessering. Det går alltid noe tid bort til administrering av instruksjoner.

Effektiv speedup En k -trinns pipeline som trenger en klokkesyker per trinn med eksekvering av n instruksjoner vil ha en speedup på:

$$\frac{kn}{k + n - 1}$$

8.5 Komplikasjoner ved pipelining

Til enhver tid kan det være subinstruksjoner fra opptil fire instruksjoner i en pipeline. Noen ganger er ikke alle subinstruksjonene gyldige. Neste instruksjon kan ikke eksekveres rett etter hvis hopp-betingelsen slår til. En slik situasjon kalles *hazard*.

8.5.1 Resource Hazard

Kan oppstå hvis to subinstruksjoner i pipelinen ønsker å aksessere samme ressurs. Andre ressurser kan også gi hazards, litt avhengig av hvordan CPU arkitekturen er bygget opp (minne, cache, register, busser, ALU osv).

8.5.2 Data Hazard

Dette oppstår fordi to forskjellige instruksjoner trenger å aksessere samme data samtidig. Trenger resultat fra forrige instruksjon før denne har produsert gyldig svar.

8.5.3 Control Hazard

I utgangspunktet har vi tenkt en pipeline som innhenter neste instruksjoner fortløpende. Men problemet oppstår når vi får en JUMP instruksjon. Da må vi tømme pipelinen for andre instruksjoner og lese inn den nye.

9 Minnehierarki

Register Integrert på CPU. Intern kladdeblokk med rask aksess til innholdet.

Cache Hurtig mellomlager for både instruksjoner og data for å jevne ut hastighetsforskjellen mellom CPUen og hurtigminnet.

RAM Buffer mellom eksternt lagringsmedium og CPUen med rask lese- og skriveaksess.

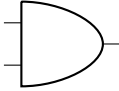
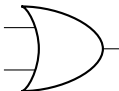
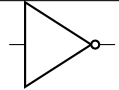
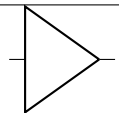
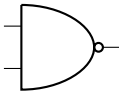
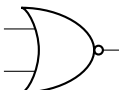
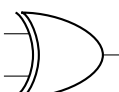
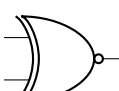
SSD/Flash/HD Høykapasitetsmedium for program/data.

9.1 Cache

Cache forbedrer von Neumann arkitekturs flaskehals med hensyn på minneaksessering. Cache ligger nærme CPU for å øke hastighet, men små størrelser for å redusere produksjonskost og derfor er det en trade-off diskusjon mellom størrelse og hastighet. Cache inneholder en kopi av et subsett av hurtigminne (RAM). CPUen henter data og/eller instruksjoner fra cache istedenfor RAM.

Siden cache er mindre enn RAM, må det bestemmes hvilke data/instruksjoner som skal ligge i cache. Cache baserer seg på at instruksjoner/data aksesseres nær hverandre i tid og rom.

Hvis data/instruksjonen CPUen trenger ligger i cache, kalles dette for en *cache hit*. Hvis den ikke ligger i cache, kalles det for *cache miss*. En essensiell del for utnyttelsen av cache er å kunne raskt sjekke for hit eller miss.

Navn	Symbol	Funksjon	Sannhetstabell		
AND		$F = x \cdot y$	x	y	F
			0	0	0
			0	1	0
			1	0	0
			1	1	1
OR		$F = x + y$	x	y	F
			0	0	0
			0	1	1
			1	0	1
			1	1	1
NOT (inverter)		$F = x'$	x	F	
			0	1	
			1	0	
Buffer		$F = x$	x	F	
			0	0	
			1	1	
NAND		$F = (xy)'$	x	y	F
			0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR		$F = (x + y)'$	x	y	F
			0	0	1
			0	1	0
			1	0	0
			1	1	0
XOR		$F = xy' + x'y = x \oplus y$	x	y	F
			0	0	0
			0	1	1
			1	0	1
			1	1	0
XNOR		$F = xy + x'y' = (x \oplus y)'$	x	y	F
			0	0	1
			0	1	0
			1	0	0
			1	1	1

Tabell 5: Oversikt over logiske porter.

			Minterm		Maksterm	
x	y	z	Term	Betegnelse	Term	Betegnelse
0	0	0	$x'y'z'$	m_0	$x + y + z$	M_0
0	0	1	$x'y'z$	m_1	$x + y + z'$	M_1
0	1	0	$x'yz'$	m_2	$x + y' + z$	M_2
0	1	1	$x'yz$	m_3	$x + y' + z'$	M_3
1	0	0	$xy'z'$	m_4	$x' + y + z$	M_4
1	0	1	$xy'z$	m_5	$x' + y + z'$	M_5
1	1	0	xyz'	m_6	$x' + y' + z$	M_6
1	1	1	xyz	m_7	$x' + y' + z'$	M_7

Tabell 6: Mintermer og makstermer for tre variable.

Boolsk funksjon	Operator	Navn	Kommentar
xy	$x \cdot y$	AND	x og y
xy'	x/y	hemming	x, men ikke y
$x'y$	y/x	hemming	y, men ikke x
$xy' + x'y$	$x \oplus y$	exclusive-OR	x eller y, men ikke begge
$x + y$	$x + y$	OR	x eller y
$(x + y)'$	$x \downarrow y$	NOR	ikke-OR
$xy + x'y'$	$(x \oplus y)'$	ekvivalens	x er lik y
$x + y'$	$x \subset y$	implikasjon	hvis y, så x
$x' + y$	$x \supset y$	implikasjon	hvis x, så y
$(xy)'$	$x \uparrow y$	NAND	ikke-AND

Tabell 7: Andre operasjoner.

x	y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Tabell 8: Sannhetsverditabell for half adder.

S	R	Q
0	0	låst
0	1	0
1	0	1
1	1	0

Tabell 9: Sannhetsverdien for SR-latch.