Oppgave 1

a) Memory:
- cache
- (også register i mye mindre grad)

Control ~~[illegible]~~:
- mikroinstruksjonsminne

~~[illegible]~~ ALU:
- register (accumulator)

b) i) ILP:

ILP går ut på å ~~[illegible]~~ flere instruksjoner samtidig på samme kjerne. Dette gjøres ved å dele opp utføreksen i mindre steg og kompartimentalisere prosessoren i tilsvarende steg.
F. eks. mens ALU-en gjør beregninger kan en "fetch-unit" hente neste instruksjon, og en "decode-unit" dekode en instruksjon.

Fagtrykk Trondheim AS 12 2015

ii) Prosessor nivå parallellitet:
Dette går ut på å ha flere kjerner som kan operere på ét felles delt minne, (+ evt. også sitt eget).
Dette er en

c)
Busy wait går ut på å overvåke en eller flere inngangs~~portsignaler~~, f.eks en buss, Dette er en veldig aktiv prosess ettersom systemet alltid må være i en eller annen løkke og vente på aktivitet.
Interrupt drevet I/O bruker en dedikert interrupt-inngang/-enhet for å aktivere systemet når det forekommer input. Systemet slipper dermed å overvåke inngangen og må kun behandle inputet når det forekommer. (mye mer effektivt).

d) Det finnes to typer lokalitet som utnyttes i minnesystemer; temporal og spatial lokalitet. Temporal lokalitet betyr at data som nylig har blitt aksessert har høy sannsynlighet for å bli aksessert igjen kort tid senere.

Spatial lokalitet betyr at etterfølgende data ofte aksesseres etter hverandre. Disse fenomenene stammer fra måten spesielt arrays lagres og bruken av løkker i de fleste algoritmer.

I forbindelse med minnesystemer betyr det at hvis data nylig har blitt aksessert er det lurt å holde den dataen og nærliggende data lagret lett tilgjengelig (cache m.m.).

e) Først og fremst er en chip multiprosessor et system med flere prosessorer/kjerner. Disse kjernene kan enten alle være like eller ulike.
Hvis de er like er prosessoren homogen.
Typisk eksempel er GPU-er som må gjøre mange av de samme operasjonene parallellt.
Hvis de er ulike er systemet heterogent.
F. eks. DVD-spillere som har egne prosessorer for dekoding av filmformatet (MPEG) og andre for lesing av disk / I/O osv.

Denne kolonnen er forbeholdt sensor

This column is for external examiner

Oppgave 2

a)

F_2	F_1	F_0	$f(A,B)$	$C(A,B,C)$	kommentar
0	0	0	0	$(A \cdot B) \oplus C \oplus (A \cdot B)$	
0	0	1	A+B+C	$(A+B)+C+(A \cdot B)$	adder med carry inngått
0	1	0	$A \oplus B$	— " —	xor
0	1	1	$A \cdot B$	— " —	AND
1	0	0	A + B	— " —	OR
1	0	1	$\overline{A \cdot B}$	— " —	NAND
1	1	0	$\overline{A}$	— " —	NOT
1	1	1	A	— " —	"identity" to A

b) i) Ved å analysere bussportene kommer vi frem til følgende adresseområder:

RAM: 0x2000 ↔ 0x3FFF

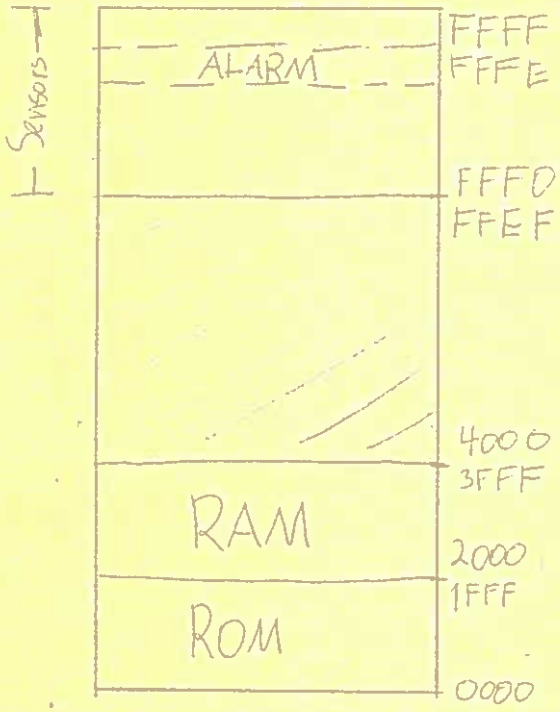
Sensors: 0xFFF0 ↔ 0xFFFF

Alarm: 0xFFFE

ROM: 0x0000 ↔ 0x1FFF

iii) Alarmen ligger i minneområdet til sensorene, men dette er ikke et problem fordi alarm er kun aktiv ved $\overline{R/W} = 1$ og sensorene ved $\overline{R/W} = 0$, som aldri skjer samtidig.

iii) Minne kart:



Fagtrykk Trondheim AS 12 2015

c) $D_0 = (\overline{Q_2} \cdot \overline{Q_1} \cdot \overline{Q_0} \cdot OK) + (\overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} \cdot OK)$

$D_1 = (\overline{Q_2} \cdot Q_1 \cdot \overline{Q_0} \cdot OK) + (\overline{Q_2} \cdot \overline{Q_1} \cdot Q_0 \cdot OK)$

$D_2 = (\overline{Q_2} \cdot Q_1 \cdot Q_0 \cdot OK)$

$Q_0(next) = D_0$

$Q_1(next) = D_1$

$Q_2(next) = D_2$

$Y = \overline{Q_0} \cdot \overline{Q_1} \cdot Q_2$

	Tilstand			neste tilstand						utgang Y
	Q_2	Q_1	Q_0	OK=0			OK=1			
				$\overline{Q_2}$	Q_1	Q_0	Q_2	Q_1	$\overline{Q_0}$	
	0	0	0	0	0	0	0	0	1	0
	0	0	1	0	0	0	0	1	0	0
	0	1	0	0	0	0	0	1	1	0
	0	1	1	0	0	0	1	0	0	0
	1	0	0	0	0	0	0	0	0	1
*	1	0	1	0	0	0	0	0	0	0
*	1	1	0	0	0	0	0	0	0	0
*	1	1	1	0	0	0	0	0	0	0

* tilstand kan ikke nås ~~maskinen starter~~ fra annen tilstand (uggyldig)

Fagtrykk Trondheim AS 12 2013

d) Dette er en tilstandsdrevet FSM siden Y kun er drevet av vippene, altså en Moore type.

Oppgave 3

a) MAR:

Brukes for lesing/skriving av minnet. MAR inneholder adressen til minnelokasjonen som skal aksessares.

MDR:

Brukes for lesing/skriving, i samspill med MAR. Ved lesing vil data i lokasjon i MAR overføres til MDR, slik at MDR inneholder dataen. Ved skriving vil dataen som skal skrives legges i MDR.

PC:

Program Counter (PC) inneholder en peker inn i programminnet. Den peker på neste instruksjon som skal hentes.

MBR:

Inneholder instruksjonen som nettopp ble hentet. Dvs. kun opkoden blir plassert her siden opkoden er en byte stor. Ved bruk av

Fagtrykk Trondheim AS 12 2013

immediate aksess vil også data bli plassert her.

b) Ved å skrive til flere registe samtidig kan det gjøres med én mikroinstruksjon.

ALU = A = 00 011000

C = 011100000

Mem = 000

B = 0000 ("don't care" egentlig)

Så mikroinstruksjonen hadde vært:

00 011000 011100000 000 0000

c) Hvis MDR og LV er like vil MDR − LV = 0 og da vil zero-flagget aktiveres. Z = 1 som kan bli detektert.

Trenger to instruksjoner.

1: ALU = B = 00 01010

C = A = 1 0000 0000

Mem = 0

B = MDR = 0000

2: ALU = B − A = 00 111111

C = H = 1 0000 0000

Mem = 0

B = LV = 0101 (5)

Så ~~~~ mikroinstruksjonene er:

1: 00 010100 10000000 000 0000

2: 00 111111 10000000 000 0101

|– ALU –| |– C –| |MBR| |– B –|

d) Det ekstra bitet brukes for branching. Når det settes til 1 ~~~~ vil MPC peke på øvre halvdel av adresseområdet til control store. Vanligvis er det satt til 0, men vi sier at signaler fra ALU-en (Z og N) brukes sammen med JAMZ og JAMN for å oppnå kondisjonell branching. Kun også ha ukondisjonell branch ved å bruke JMPC.

e) Lengden er 36-bit og maksimal mengde er 512 mikroinstruksjoner.

Fagtrykk Trondheim AS 12 2013

Oppgave 4

a) Instruksjonslengden er lik for alle instruksjonene (32-bit) opkoden har samme størrelse (6-bit) på tvers. Instruksjonene er enkle i den forstand at det ikke ligger/forekommer komplekse algoritmer bak instruksjone. (simple og generelle).

b) MOVC bruker imidiate.

c) NOP og RT er 0-adr. instruksjoner.

d) Går gjennom koden linje for linje.

1) R_1 lastes med verdien 0x CC CC CC 55

2) R_2 —— 11 —— 0x CC CC CC AA

3) $R_1 = R_1 + R_2$ = 0x [illegible]

4) 0x CC CC CC 55 lagres i minnelokasjon 0xCCCCCCAA

=> 0x CC CC CC AA ⟶ 0x CC CC CC 55
(adr.)

4) $R_1 = R_1 + R_3$ = 0x 55 + 0xFFFF0000 = 0xFFFF CC 55

5) 0xFFFFCC55 lagres i minnelokasjon 0xFFFF0000

=> 0xFFFF 0000 ⟶ 0xFFFF CC55
(adr.)

6) R_1 = 0x00 CC CC 55 = 0x55

7) R16 = 0x55

8) $R_1 = R_1 + R_{16} = 0x55 + 0x55 = 0xAA$

9) Sammenligner R_1 og R_2.
Vi har $R_1 = 0xAA$
og $R_2 = 0xAA$
så $R_1 = R_2$ og Z-flagget settes til 1.

10) Branch til 0xAA siden Z=1. PC=0xAA

R_1 har verdien 0xAA etter koden har kjørt.

Oppgave 5

a) Gj. snitt. aksesstid blir

$$\bar{t} = t_{cache} + (1 - \mu_{hit}) t_{ram}$$

$$\Rightarrow \bar{t} = 10\,ns + (1 - 0{,}9) \cdot 100\,ns$$

$$= 30\,ns$$

b) Maximal klokkefrekvens gitt av treigeste enhet.

1) $f_{max} = \frac{1}{t_{execute}} = \frac{1}{15\,ns} = 66{,}7\,MHz$

2) $f_{max} = \frac{1}{t_{operand}} = \frac{1}{7{,}5\,ns} = 133{,}3\,MHz$

3) Vi må da se på summen av alle. For både fig. 7 og 8 er summen (totaltid)

Denne kolonne er forbeholdt sensor

This column is for external examiner

37,5 ns. Det gir

$$f_{maks} = \frac{1}{37{,}5\,ns} = 26{,}7\ MHz$$

4) Det gir spesielt utfordringer ift. branchling, kondisjonell
Det er fordi man må vite resultatet av kondisjonstesten, før man kan utføre neste instruksjon. Hvis man ikke vet det eller tipper feil må pipelinen tømmes som kaster bort mange klokkesykler. ~~Jo dypere pipelinen er jo større sjanse for~~ avhengighet

Vi kan også ha problemer med avhengigheter. Hvisen linje med kode er avhengig av at den forrige linjen er ferdig må write back delen av den forrige bli utført ofte før operand delen av den nåværende.

Jo dypere pipelinen er jo dyrere er det tømme den og jo større sjanse er det for avhengigheter.

Denne kolonne er forbeholdt sensor

This column is for external examiner

c) Vi kan legge inn en dedikert fetch enhet som i henter instruksjoner og inkrementerer PC.

Vi kan innføre et samletånd (ved å bruke vipper). Dette vil la oss dele opp utførelsen i flere steg og øke klokkefrekvensen.