OBLIGATORISK TEORIØVING NR 5 – TFE4105
INNLEVERING I LÅSBARE BOKSER I KJELLER ELEKTRO B. FRIST:
Grupper med oddetallsnummer mandag 5/11 klokken 1700.
Grupper med partallsnummer mandag 12/11 klokken 1700.
BESVARELSEN SKAL INNEHOLDE:
GRUPPENUMMER, NAVN PÅ DE SOM ER I GRUPPEN, ÅRSKURS, STUDIERETNING

Øving 5 – TFE4105 Digitalteknikk og Datamaskiner Høsten 2012

Emner som øvingen omfatter:

- Utførende enhet
- Adresseringsmodi
- Instruksjonssett
- RISC og CISC arkitekturer
- Assemblerprogrammering

Øvingen tar for seg stoff fra kap. 10 og 11 i Mano & Kime.

Oppgave 1 – "Papirsimulering" av utførende enhet (Inntil 3 poeng)

I denne oppgaven skal vi bruke en utførende enhet som er vist i figur 10-11 og tabell 10-5 i Mano & Kime. I tabellen til venstre under er gitt en sekvens av 16-biters styreord til den utførende enheten, og i tabellen til høyre vises registerinnhold før oppstart.

Styreordsekvens

011 011 001 0 0010 0 1
100 100 001 0 1001 0 1
101 101 001 0 1010 0 1
001 001 000 0 1011 0 1
001 001 000 0 0001 0 1
110 110 001 0 0101 0 1
111 111 001 0 0101 0 1
000 111 000 0 0000 0 1

Registerinnhold

R0	00000000
R1	00100000
R2	01000100
R3	01000111
R4	01010100
R5	01001100
R6	01000001
R7	01001001

Simuler en kjøring med disse styreordene og finn registerinnhold i de 8 registrene etter kjøringen.

Oppgave 2 – Operandadressering (Inntil 4 poeng, 2 pr. deloppgave)

I Mano & Kime kap. 11-2 omtales operandadressering. Prosessorer klassifiseres ofte etter maks. antall eksplisitte operander. I begrepet *n-adresse-maskin* refererer *n* til maks. antall operander som kan spesifiseres i en instruksjon. Gitt

X = (A+B)*(D-B)/(C-B). 2

Programmet for beregning av X skal utføres på forskjellige maskiner.

a) Vis hvordan programmet for beregning av X kan se ut for maskinene nedenfor.

- 0-adresse maskin (stakk).
- 1-adresse maskin.
- 1 1/2-adresse maskin. 1 1/2-adresse maskin er en maskin som kan ta to operander, men bare en av operandene kan være en lageroperand den andre må være en registeroperand. To registeroperander er også en lovlig kombinasjon. Anta at maskinen kun har 2 registre.
- 3-adresse maskin, hvis man ikke bruker registre til mellomlagring.

a) Vi antar at hver instruksjon krever én minnereferanse, inkludert eventuelle registeroperander som instruksjonen bruker. I tillegg vil hver enkelt dataverdi man henter ut fra eller skriver til minnet, kreve én minnereferanse. Hvor mange minnereferanser trenger hver av de fire maskinene for å utføre programmet?

Oppgave 3 – Adresseringsmodi (Inntil 4 poeng)

Gitt en maskin med et antall registre R_n som kan benyttes både for data og adresser. Lager- og registerinnhold er vist i figur 1.

Hva blir innholdet i register R1 etter utførelse av hver av instruksjonene under? Vis med en skisse hvordan du kommer frem til resultatet. Skisser også hvilken informasjon operanddelen av instruksjonen må inneholde.

Instruksjonen MOV er definert slik: *MOV <til>,<fra>*

Instruksjoner:

Direkte adressering…….	1) MOV R1,R0
	2) MOV R1,VAR1
Indirekte adressering …..……	3) MOV R1,[R4]
	4) MOV R1,[VAR2]
Indeksert adressering……	5) MOV R1,TABELL(R2)
Postindeksert lager indirekte adressering *……	6) MOV R1,[VAR3](R3)
Preindeksert lager indirekte. **..	7) MOV R1,[VAR3(R3)]

Plassering av variable i lageret

Adr VAR1 =16
Adr VAR2 = 8
Adr VAR3 = 10
Adr TABELL = 11

REGISTRE

R0=0
R1=1
R2=4
R3=3
R4=15

HOVEDLAGER	ADR
233	7
15	8
4	9
15	10
17	11
12	12
17	13
9	14
7	15
8	16
21	17
19	18
7	19
9	20

Figur 1

* Postindeksert betyr at indeksregisteret legges til etter at man har funnet den lagerindirekte adressen.

** Preindeksert betyr at indeksregisteret 3 legges til variabelen som brukes til å finne den lagerindirekte adressen.

Oppgave 4 – RISC og CISC (Inntil 2 poeng, 1 pr. deloppgave)

a) Datamaskiner kan defineres som RISC (Reduced Instruction Set Computer) eller CISC (Complex Instruction Set Computer). Hva skiller de to arkitekturene? Nevn karakteristiske egenskaper/oppbygningsforskjeller ved de to arkitekturene.

b) Prøv å resonnere deg fram til et bruksområde der CISC vil være bedre egnet enn RISC.

Oppgave 5 – Assemblerprogram for RISC (Inntil 7 poeng, 3 for deloppgave a) og 4 for b))

I denne oppgaven skal vi benytte instruksjonssettet PPC-RISC, vedlagt oppgaveteksten.

a) Gitt høynivåkoden (pseudokode):

```
if (((X>0) or (Y≤Z)) and (Z≠1)) then
      Z = Z+1
else
      Z = Z-1;
endif
```

Skriv et assemblerprogram som beregner `Z`, med registerinnhold som gitt i tabellen nedenfor. Returadressen er lagret i registret LR.

Inngangsverdier	**Utgangsverdier**
r3 = X	r3 = Z
r4 = Y	LR = returadresse
r5 = Z	

b) En N-tegns tekststreng B[0:N-1] er lagret på ASCII-format i minnet, en byte per tegn. Skriv et program som søker etter tegnet A i tekststrengen B, og bytter ut alle tegn foran A med ASCII NULL (\0). Altså:

```
I = 0;
while ((I<N) and (A≠B[I])) do
      I = I+1;
done
if (I<N) then
      for (J=0 to I-1) do
            B[J] = '\0';
      done
endif
```

Anta registerinnhold som vist i tabellen nedenfor. Merk at søket i while-løkken avslutter med I=N hvis tegnet ikke fins. Sjekk at tilfellene A=B[0], A=B[0<I<N-1],

A=B[N-1] og A≠B[I] alle fungerer. 4 Bruk kladderegistre etter behov. Velg blant registrene r0, r6 og r7-r13.

r3 = N	Antall tegn i strengen
r4 = A	Tegnet vi søker etter
r5 = B	Adressen til første tegn i den lagrede tekststrengen
LR = returadresse	Retur fra rutinen

PPC-RISC reference

[2011-11-06]

The tables below contain a small subset of the available instructions. Instructions are 32 bits, with a 32-bit architecture and starts on a 4 byte boundary. The operator || concatenates two bit fields, $^{n}0$ is a n-bit field of zeros, and ral0 is register a if a$\neq$0 and $^{32}0$ otherwise. Status bits are always set explicitly, by using the dot-form mnemonics (e.g. add.) or by the compare instruction (cmp).

FIXED-POINT STORAGE ACCESS

name	operation	syntax	description	example
l	load	`l rd,d(ra)`	rd ← M[ral0 + *se* d(15:0)]	`l r3,0(r4)`
lx	load indexed	`lx rd,ra,rb`	rd ← M[ral0 + rb]	`lx r3,r4,r5`
lbz	load byte and zero	`lbz rd,d(ra)`	rd ← $^{24}0$ || M[ral0 + *se* d(15:0)]	`lbz r3,0(r4)`
lbzx	load byte and zero indexed	`lbzx rd,ra,rb`	rd ← $^{24}0$ || M[ral0 + rb]	`lbzx r3,r4,r5`
st	store	`st rd,d(ra)`	M[ral0 + *se* d(15:0)] ← rd	`st r3,0(r4)`
stx	store indexed	`stx rd,ra,rb`	M[ral0 + rb] ← rd	`stx r3,r4,r5`
stu	store with update	`stu rd,d(ra)` [1]	M[ra + *se* d(15:0)] ← rd, ra ← ra + *se* d(15:0)	`stu r3,4(r4)`
stb	store byte	`stb rd,d(ra)`	M[ral0 + *se* d(15:0)] ← rd(7:0)	`stb r3,0(r4)`
stbx	store byte indexed	`stbx rd,ra,rb`	M[ral0 + rb] ← rd(7:0)	`stbx r3,r4,r5`

1: ra = r0 is illegal.

FIXED-POINT ARITHMETIC

name	operation	syntax	description	example
add	add	`add[.] rd,ra,rb` [2]	rd ← ra + rb	`add r3,r4,r5`
addi	add immediate	`addi rd,ra,d`	rd ← ral0 + *se* d(15:0)	`addi r3,r4,-2`
addc	add carrying	`addc[.] rd,ra,rb` [2]	rd ← ra + rb, CA ← C_{out}	`addc r3,r4,r5`
adde	add extended	`adde rd,ra,rb`	rd ← ra + rb + CA	`adde r3,r4,r5`
cmp	compare	`cmp n,ra,rb`	CRn [3] ← ra - rb	`cmp 0,r3,r4`
cmpi	compare immediate	`cmpi n,ra,d`	CRn [3] ← ra - *se* d(15:0)	`cmpi 1,r3,-1`
divw	divide word	`divw[.] rd,ra,rb` [2]	rd ← ra / rb	`divw r3,r4,r5`
mullw	multiply low word	`mullw[.] rd,ra,rb` [2]	rd ← (ra * rb)(L)	`mullw r3,r4,r5`
neg	negate	`neg[.] rd,ra` [2]	rd ← $\overline{\text{ra}}$ + 1	`neg r3,r4`
sub	subtract	`sub[.] rd,ra,rb` [2]	rd ← ra - rb	`sub r3,r4,r5`

2: alternate forms, *inst.* with a dot assigns CR0 bits, while *inst* does not.
3: no registers are saved on a compare, operands are compared against 0 and CRn-bits set.

FIXED-POINT LOGICAL

name	operation	syntax	description	example
and	and	`and[.] rd,ra,rb` [2]	rd ← ra ∧ rb	`and r3,r4,r5`
andi	and immediate	`andi. rd,ra,d`	rd ← ra ∧ $^{16}0$ || d(15:0)	`andi. r3,r4,15`
eqv	equivalent	`eqv[.] rd,ra,rb` [2]	rd ← $\overline{\text{ra} \oplus \text{rb}}$	`eqv r3,r4,r5`
mr	move register	`mr[.] rd,ra`	rd ← ra	`mr r3,r4`
not	not	`not[.] rd,ra` [2]	rd ← $\overline{\text{ra}}$	`not r3,r4`
or	or	`or[.] rd,ra,rb` [2]	rd ← ra ∨ rb	`or r3,r4,r5`
ori	or immediate	`ori rd,ra,d`	rd ← ra ∨ $^{16}0$ || d(15:0)	`ori r3,r4,15`
sl	shift left	`sl[.] rd,ra,rb` [2]	rd ← ra sl rb(4:0)	`sl r3,r4,r5`
sli	shift left immediate	`sli[.] rd,ra,d` [2]	rd ← ra sl d(4:0)	`sli r3,r4,3`
sr	shift right	`sr[.] rd,ra,rb` [2]	rd ← ra sr rb(4:0)	`sr r3,r4,r5`
sri	shift right immediate	`sri[.] rd,ra,d` [2]	rd ← ra sr d(4:0)	`sri r3,r4,r5`
rotlw	rotate left word	`rotlw[.] rd,ra,rb` [2]	rd ← ra rl rb(4:0)	`rotlw r3,r4,r5`
rotlwi	rotate left word immediate	`rotlwi[.] rd,ra,d` [2]	rd ← ra rl d(4:0)	`rotlwi r3,r4,3`
xor	xor	`xor[.] rd,ra,rb` [2]	rd ← ra ⊕ rb	`xor r3,r4,r5`
xori	xor immediate	`xori rd,ra,d`	rd ← ra ⊕ $^{16}0$ || d(15:0)	`xori r3,r4,15`

MOVE TO/FROM SPECIAL PURPOSE REGISTERS[4]

name	operation	syntax	description	example
crand	and CR bits (also or and xor)	`crand bd,ba,bb`	bd ← ba ∧ bb	`crand 2,1,4`
mfctr	move from CTR	`mfctr ra`	ra ← CTR	`mfctr r0`
mflr	move from LR	`mflr ra`	ra ← LR	`mflr r0`
mtctr	move to CTR	`mtctr ra`	CTR ← ra	`mtctr r0`
mtlr	move to LR	`mtlr ra`	LR ← ra	`mtlr r0`
mfcr	move from CR	`mfcr ra`	ra ← CR0-7	`mfcr r0`
mtcrf	move to CRn	`mtcrf` 2^{7-n}`,ra`	CRn ← ra(31-4n:28-4n)	`mtcrf 0x80,r0`

4: supports branching and subroutine calls, refer to the branch table below.

BRANCH FACILITY

name	operation	syntax	description	example
b	branch	`b d`	PC ← PC + *se* d(23:0)‖^{2}0	`b loop`
bxx	branch conditional	`bxx n,d`	CRn xx: PC ← PC + *se* d(13:0)‖^{2}0[5,6]	`bne 1,loop`
bdnz	branch conditional	`bdnz d`	CTR ← CTR - 1, CTR≠0: PC ← PC + *se* d(13:0)‖^{2}0	`bdnz loop`
bdnxx	branch conditional	`bdnxx d`	CTR ← CTR - 1, xx ∧ CTR≠0: PC ← PC + *se* d(13:0)‖^{2}0[7]	`bdneq loop`
bdz	branch conditional	`bdz d`	CTR ← CTR - 1, CTR≡0: PC ← PC + *se* d(13:0)‖^{2}0	`bdz loop`
bdzxx	branch conditional	`bdzxx d`	CTR ← CTR - 1, xx ∧ CTR≡0: PC ← PC + *se* d(13:0)‖^{2}0[7]	`bdzeq loop`
bl	branch	`bl d`	PC ← PC + *se* d(23:0) LR ← PC + 4	`bl .mysub`
blr	branch to LR	`blr`	PC ← PC + LR(31:2)‖^{2}0	`blr`

5: legal values of xx are eq (= 0), gt (> 0), lt (< 0), ne (≠ 0), ge (≥ 0), and le (≤ 0).

6: CR0 can be left out, that is `bxx 0,d` can be simplified to `bxx d`.

7: values of xx as for 5, but reads CR0 only.

PPC-RISC REGISTER SET

name	description
R0, R11-R13	general purpose scratch
R1 (SP)	*reserved for stack pointer*
R2, R14-R31	saved registers, must be saved and restored if used
R3-R10	argument registers, general purpose scratch
R3 (R4)	function results, R3‖R4 for 64-bit results
PC	32-bit program counter
LR	link register, for use with function calls
CTR	count register, for loop control
CR0-7	8 × 4-bit condition registers, individual status bits LT‖GT‖EQ‖SO
CR2-4	saved CR-registers, must be saved and restored if used

HAND MULTIPLICATION EXAMPLE OF M&K FIG. 8-4

```
1          addi    r0,r0,0      # clear r0 = product
2  add2:   andi.   r5,r4,1      # r5 = multiplier & 0x1, save to CR0
3          beq     shift        # branch if multiplier & 0x1 == 0
4          add     r0,r0,r3     # product += multiplicand
5  shift:  sli     r3,r3,1      # multiplicand <<= 1
6          sri.    r4,r4,1      # multiplier >>= 1, save to CR0
7          bne     add2         # continue if multiplier > 0
8          or      r3,r0,r0     # copy to function result
9          blr
```

Computes the 32-bit product (r0) = multiplicand (r3) * multiplier (r4), with r3, r4 < 2^{16} on entry.