Tentamen i kursen EDA330

Datorsystemteknik D

21/8 1999

Tentamensdatum: Lördag 21/8 1999 kl. 8.45 i sal MG

Examinator: Jonas Vasell

Institution: Datorteknik

Förfrågningar: Peter Rundberg (ankn. 1682)

Lösningar: Anslås måndag 23/8 på institutionens anslagstavla utanför laboratoriet

Resultat: Anslås senast fredag 3/9 på institutionens anslagstavla utanför laboratoriet

Rättningsgranskning: Tid och plats anslås i samband med resultaten.

Betygsgränser: 3: 24-35 poäng, 4: 36-47 poäng, 5: 48-60 poäng

Tillåtna hjälpmedel: inga

Allmänt: För full poäng på en uppgift krävs både ett korrekt svar och en motivering. En bra motivering är minst lika viktig som ett korrekt svar. Redovisa noggrannt alla gjorda antaganden utöver de som anges i uppgiftstexten. Skriv tydligt och använd gärna figurer. Maximal poäng på varje deluppgift anges inom parentes efter uppgiftstexten.

Välgångsönskning: Lycka till!

Uppgifter (1-5):

1.

a. Ange intruktionskoderna i binär form för det minsta antal MIPS-instruktioner som krävs för att ladda flyttalskonstanten 2,375 i register 7. Konstanten ska representeras i flyttalsformatet IEEE 754 med enkel precision (32 bitar). Exponenten i detta flyttalsformat är 8 bitar med bias 127. Konstanten får ej lagras i dataminnet. Använd endast de MIPS-instruktioner som anges i bilaga 2. (6 p)

b. Redogör för de fyra huvudstegen vid flyttalsaddition. Beskriv för varje steg vilken funktion det har. Definiera alla använda begrepp som är specifika för flyttalsrepresentation. (8 p)

2. Följande MIPS-kod utför en funktion som förekommer på många ställen i en viss tillämpning:

```
        sll    $a0, $a0, 2
        lw     $v1, XTAB($a0)
        lw     $v0, 0($v1)
        beq    $v0, $zero, L1
        addi   $v0, $v0, -1
        beq    $zero, $zero, L2
L1:     addi   $v0, $zero, 7
L2:     sw     $v0, 0($v1)
```

a. Vad gör denna funktion? (2 p)

b. Från vilket/vilka register hämtas indata (argument), och i vilket/vilka register finns resultat som kan användas efter funktionen? (2 p)

c. Om funktionen utförs i en pipeline som den i bilaga 1, med data forwarding där så är möjligt och predict-not-taken som hoppstrategi, hur många extra cykler orsakade av pipeline-konflikter kommer det att krävas för att exekvera funktionen? Delayed branch används ej, dvs instruktionen efter en hoppinstruktion utförs bara om hoppet inte tas. Om antalet extra cykler kan vara olika i olika fall, ange vilka dessa fall är, och hur många extra cykler som krävs i respektive fall. (4 p)

d. Eftersom rutinen utförs på många ställen i tillämpningen, så vill man göra en subrutin av den. Visa minsta nödvändiga tillägg till koden ovan för att göra om den till en subrutin, och skriv en så kort kod som möjligt som laddar argumentregistren med konstantvärden och anropar subrutinen. (4 p)

3. Du har fått i uppgift att konstruera ett cacheminne till en specialprocessor för ett datainsamlingssystem. Specialprocessorn implementeras som en ASIC (tillämpningsspecifik integrerad krets) med VLSI-teknik. Tillgängliga utvecklingsverktyg och begränsningar hos kretsen gör att du måste bygga upp cacheminnet med färdiga RAM-komponenter som vardera kan lagra 256x16 bitar, dvs varje komponent har ett adressrum på 256 adresser och kan lagra 16 bitar per adress.Totalt kan du använda som mest 10 sådana RAM-komponenter. Din uppgift är nu att hitta en konfiguration för cacheminnet som ger så låg miss rate som möjligt. De parametrar du kan variera är associativitet (1 eller 2), datalagringskapacitet (1KB, 2 KB, eller 4KB), och antal ord per block (1 eller 2). Eftersom det är mycket viktigt att begränsa trafiken till primärminnet så ska cacheminnet använda write-back som skrivstrategi. Dessutom gör det faktum att cacheminnet inte kan bli så stort att LRU bör användas som utbytesalgoritm, vilket kräver $\log_2$(associativitet) replacement-bitar per block. Primärminnesadresserna är 28 bitar breda.

   Som ett första steg använder du en simulator för att mäta upp miss rate för de olika konfigurationerna och kommer då fram till följande:

Datalagringskapacitet	Associativitet	Ord/block	Miss rate
1 KB	1	1	32,5%
1 KB	1	2	24,8%
1 KB	2	1	27,4%
1 KB	2	2	20,9%
2 KB	1	1	25,2%
2 KB	1	2	19,2%
2 KB	2	1	20,7%
2 KB	2	2	15,9%
4 KB	1	1	18,5%
4 KB	1	2	14,2%
4 KB	2	1	16,2%
4 KB	2	2	12,4%

   a. Varför minskar miss rate när datalagringskapaciteten ökar? (1 p)

   b. Varför minskar miss rate när antalet ord/block ökar? (1 p)

   c. Varför minskar miss rate när associativiteten ökar? (1 p)

   d. Ibland kan en konfiguration med lägre miss rate ge sämre systemprestanda än en konfiguration med högre miss rate. Vad kan ha hänt då om man antar att endast cachekonfigurationen ändrats? (1 p)

   e. Vilken är den lägsta miss rate du kan åstadkomma med de givna förutsättningarna? TIPS: Räkna ut antal block, sets, tagbitar, indexbitar med mera för olika konfigurationer och se efter hur många RAM-komponenter som krävs i respektive fall för att hitta de möjliga konfigurationerna. Det är inte säkert att du behöver gå igenom alla konfigurationer. (8 p)

4. Systemet för låsningsfria bromsar (ABS) i en bil är ytterst säkerhetskritiskt. I ett visst sådant system körs med regelbundna intervall en kalibreringsrutin för varje broms. Denna rutin läser 100 mätvärden från en sensor. Från dessa värden och diverse värden som hämtas från inbyggda tabeller beräknas en kalibreringsfaktor som påverkar hur bromsverkan regleras. Kalibreringsrutinen körs på en mikroprocessor i bromssystemets styrenhet. Programmet initierar en avläsning av sensorn och väntar sedan på ett svar från sensorn. Processorn har 100 MHz klockfrekvens och är kopplad till ett relativt litet RAM som primärminne. Cacheminne för data och instruktioner är inbyggt i processorn. Exekveringen av kalibreringsrutinen har följande karaktäristik:
   - Antal exekverade instruktioner: 2×10^8
   - Antal dataminnesåtkomster (primärt tabelluppslagningar): 4×10^7
   - Kalibreringsrutinen är så liten att effekter av instruktionscachemissar kan försummas.
   - Genomsnittlig väntetid från initiering av avläsning per sensorvärde: 10 ms

   Innan kalibreringsrutinen börjar användas provkörs den i ett testsystem med samma typ av styrenhet som i en färdig bil, men med en testsensor istället för den riktiga bromssensorn. Med testsensorn blir den genomsnittliga väntetiden på sensorvärden 1 ms. Man passar också på att prova med två typer av RAM som primärminne; en ny typ som gör att åtkomsttiden för ett cacheblock från primärminnet blir 100 ns, och en lite billigare och långsammare standardtyp som ger åtkomsttiden 200 ns för ett cacheblock. Med den första, snabbare typen av RAM tar kalibreringsrutinen 3,1 s att köra vid testningen. Med den långsammare typen av RAM blir körningstiden istället 3,5 s.

   a. Hur lång tid kommer det att ta köra kalibreringsrutinen med den riktiga bromssensorn och standardtypen av RAM? (2 p)

   b. Med en bättre kodning av rutinen visar testkörningar att sannolikheten för datacachemissar halveras. Dock ändras inte antalet instruktioner eller instruktionsmixen nämnvärt. Vad blir körningstiden med den riktiga bromssensorn och standardtypen av RAM om denna förändring införs? (4 p)

   c. Utöver förbättringen som man får med ändringen i deluppgift b vill man uppnå ytterligare uppsnabbning för att klara de tidskrav som ställs på systemet. Ett uppenbart förslag är att byta till den snabbare nya RAM-typen, men det har nackdelen att det kostar ganska mycket. Då kommer någon på att man kan försöka med att öka processorns klockfrekvens vilket blir mycket billigare. Dock visar det sig att klockfrekvensen bara kan höjas med 10% till 110 MHz utan att effektutvecklingen i styrenheten blir för stor. Är det möjligt att uppnå samma prestandaförbättring med en justering av klockfrekvensen som med det dyrare bytet av RAM-typ? (4 p)

5. Nedan följer ett antal frågor med tre svarsalternativ (1, X, 2) vardera, varav endast ett är rätt. Ställ upp svaren som en tipsrad. Varje rätt svar ger en poäng. (12 p)

   a. Minneskoherens innebär (1) att alla tillgängliga kopior av en del av minnet alltid är lika. (X) att det bara får finnas en kopia av varje del av minnet. (2) att minnet är skrivskyddat.

   b. Booths algoritm (1) är en metod för flyttalsmultiplikation. (X) är en metod för heltalsmultiplikation. (2) är en metod för flyttalsdivision.

   c. MIPS-arkitekturen tillåter adressering av (1) 2^{30} ord. (X) 2^{30} byte. (2) 2^{32} ord.

   d. SPEC är (1) en uppsättning benchmarks för linjära ekvationssystem. (X) ett kernel-baserat benchmark. (2) en standardiserad uppsättning benchmarks baserad på riktiga program.

   e. Write-invalidate är (1) en teknik för att stoppa felaktiga skrivningar. (X) ett cache-koherensprotokoll. (2) en typ av minnesåtkomst.

   f. Hög rumslokalitet i minnessystem innebär att (1) minneskretsarna är tätt packade. (X) om en adress refereras så är det stor sannolikhet att en närliggande adress snart refereras. (2) att få adresser refereras.

   g. SIMD står för (1) Single In-line Memory Device. (X) Single Instruction Multiple Data. (2) Single Interrupt Multiple Devices.

   h. PCI är en vanlig standard för (1) processor-minnesbussar. (X) bakplansbussar. (2) I/O-bussar.

   i. En superskalär processor (1) kan starta exekvering av flera instruktioner samtidigt. (X) har en extra lång pipeline. (2) har en speciellt kraftfull ALU.

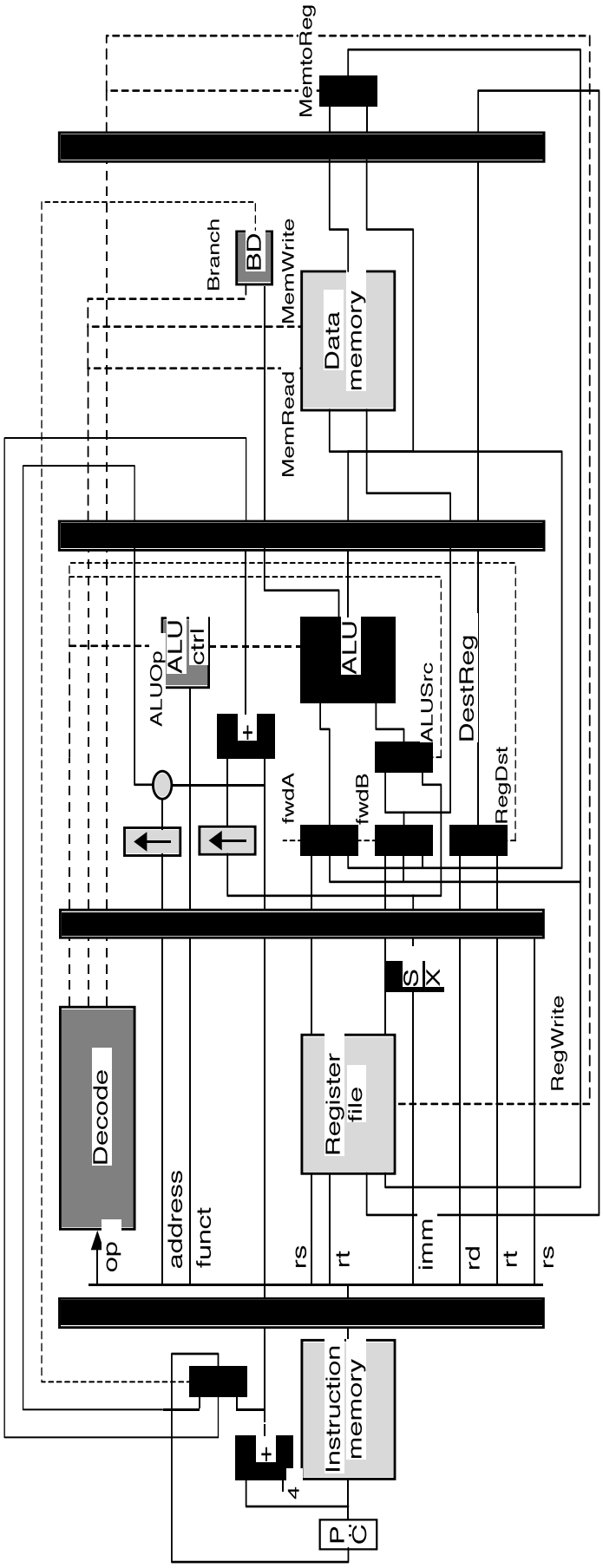
   j. En atomisk operation (1) är en processorinstruktion som utför en serie minnesåtkomster som inte får avbrytas av andra instruktioner. (X) är en processorinstruktion som utför en serie minnesåtkomster som får avbrytas av andra instruktioner. (2) är en processorinstruktion som utför en enda minnesläsning.

   k. CPI för en processor beror på (1) kompilator och arkitektur. (X) arkitektur och implementeringens organisation. (2) implementeringens organisation och hårdvaruteknologi.

   l. Ett “fat tree” är en nätverkstopologi (1) som är trädbaserad och har högre bandbredd närmare löven än vid roten. (X) som är trädbaserad och har högre bandbredd än en vanlig trädtopologi. (2) som är trädbaserad och har högre bandbredd närmare roten än vid löven.

SLUT

Bilaga 1: MIPS pipeline



Bilaga 2: MIPS maskininstruktioner

Notes:*op*, *funct*, *rd*, *rs*, *rt*, *imm*, *address*, *shamt* refer to fields in the instruction format. The program counter PC is assumed to point to the next instruction (usually 4 + the address of the current instruction). M is the byte-addressed main memory.

Assembly instruction	Instr. format	*op* *op*/*funct*	Meaning	Comments
`add $rd, $rs, $rt`	R	0/32	`$rd = $rs + $rt`	Add contents of two registers
`sub $rd, $rs, $rt`	R	0/34	`$rd = $rs - $rt`	Subtract contents of two registers
`addi $rt, $rs, imm`	I	8	`$rt = $rs + imm`	Add signed constant
`addu $rd, $rs, $rt`	R	0/33	`$rd = $rs + $rt`	Unsigned, no overflow
`subu $rd, $rs, $rt`	R	0/35	`$rd = $rs - $rt`	Unsigned, no overflow
`addiu $rt, $rs, imm`	I	9	`$rt = $rs + imm`	Unsigned, no overflow
`mfc0 $rt, $rd`	R	16	`$rt = $rd`	*rd* = coprocessor register (e.g. epc, cause, status)
`mult $rs, $rt`	R	0/24	`Hi, Lo = $rs * $rt`	64 bit signed product in Hi and Lo
`multu $rs, $rt`	R	0/25	`Hi, Lo = $rs * $rt`	64 bit unsigned product in Hi and Lo
`div $rs, $rt`	R	0/26	`Lo = $rs / $rt, Hi = $rs mod $rt`	
`divu $rs, $rt`	R	0/27	`Lo = $rs / $rt, Hi = $rs mod $rt` (unsigned)	
`mfhi $rd`	R	0/16	`$rd = Hi`	Get value of Hi
`mflo $rd`	R	0/18	`$rd = Lo`	Get value of Lo
`and $rd, $rs, $rt`	R	0/36	`$rd = $rs & $rt`	Logical AND
`or $rd, $rs, $rt`	R	0/37	`$rd = $rs | $rt`	Logical OR
`andi $rt, $rs, imm`	I	12	`$rt = $rs & imm`	Logical AND, unsigned constant
`ori $rt, $rs, imm`	I	13	`$rt = $rs | imm`	Logical OR, unsigned constant
`sll $rd, $rs, shamt`	R	0/0	`$rd = $rs << shamt`	Shift left logical (shift in zeros)
`srl $rd, $rs, shamt`	R	0/2	`$rd = $rs >> shamt`	Shift right logical (shift in zeros)
`lw $rt, imm($rs)`	I	35	`$rt = M[$rs + imm]`	Load word from memory
`sw $rt, imm($rs)`	I	43	`M[$rs + imm] = $rt`	Store word in memory
`lbu $rt, imm($rs)`	I	37	`$rt = M[$rs + imm]`	Load a single byte, set bits 8-31 of $*rt* to zero
`sb $rt, imm($rs)`	I	41	`M[$rs + imm] = $rt`	Store byte (bits 0-7 of $*rt*) in memory
`lui $rt, imm`	I	15	`$rt = imm * 2^16`	Load constant in bits 16-31 of register $*rt*
`beq $rs, $rt, imm`	I	4	`if($rs==$rt) PC = PC + imm` (PC always points to next instruction)	
`bne $rs, $rt, imm`	I	5	`if($rs!=$rt) PC = PC + imm` (PC always points to next instruction)	

Notes:*op*, *funct*, *rd*, *rs*, *rt*, *imm*, *address*, *shamt* refer to fields in the instruction format. The program counter PC is assumed to point to the next instruction (usually 4 + the address of the current instruction). M is the byte-addressed main memory.

Assembly instruction	Instr. format	*op* *op*/*funct*	Meaning	Comments
`slt $rd, $rs, $rt`	R	0/42	`if($rs<$rt) $rd = 1; else $rd = 0`	
`slti $rt, $rs, imm`	I	10	`if($rs<imm) $rt = 1; else $rt = 0`	
`sltu $rd, $rs, $rt`	R	0/43	`if($rs<$rt) $rd = 1; else $rd = 0` (unsigned numbers)	
`sltiu $rt, $rs, imm`	I	11	`if($rs<imm) $rt = 1; else $rt = 0` (unsigned numbers)	
`j destination`	J	2	`PC = address*4`	Jump to *destination*, *address* = *destination*/4
`jal destination`	J	3	`$ra = PC; PC = address*4` (Jump and link, *address* = *destination*/4)	
`jr $rs`	R	0/8	`PC = $rs`	Jump to address stored in register $*rs*

MIPS registers

Name	Number	Usage
`$zero`	0	constant 0
`$at`	1	reserved for assembler
`$v0 - $v1`	2-3	expression evaluation and function results
`$a0 - $a3`	4-7	arguments
`$t0 - $t7`	8-15	temporary, saved by caller
`$s0 - $s7`	16-23	temporary, saved by called function
`$t8 - $t9`	24-25	temporary, saved by caller
`$k0 - $k1`	26-27	reserved for kernel (OS)
`$gp`	28	points to middle of a 64K block in the data segment
`$sp`	29	stack pointer (top of stack)
`$fp`	30	frame pointer (beginning of current frame)
`$ra`	31	return address
`Hi, Lo`	-	store partial result of mult and div operations
`PC`	-	contains the address of the next instruction to be fetched (this is not a real MIPS register, and is only used to define instructions)
`status`	-	register 12 in coprocessor 0, stores interrupt mask and enable bits
`cause`	-	register 13 in coprocessor 0, stores exception type and pending interrupt bits
`epc`	-	register 14 in coprocessor 0, stores address of instruction causing exception

MIPS Instruction formats

Format	Bits 31-26	Bits 25-21	Bits 20-16	Bits 15-11	Bits 10-6	Bits 5-0
R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	imm		
J	op	address				

MIPS Assembler syntax

```
                                # This is a comment
            .data               # Store following data in the
data
                                # segment
    items:                      # This is a label connected to
the
                                # next address in the current
segment
            .word 1, 2          # Stores values 1 and 2 in next
two
                                # words
    hello:  .asciiz “Hello”     # Stores null-terminated string
in
                                # memory
            .text               # Store following instructions in
                                # the text segment
    main:   lw $t0, items($zero)  # Instruction that uses a label
to
                                # address data
```