Written Exam / Tentamen

Computer Organization and Components / Datorteknik och komponenter (IS1500), 9 hp
Computer Hardware Engineering / Datorteknik, grundkurs (IS1200), 7.5 hp

KTH Royal Institute of Technology

2020-03-11

08.00-13.00

Teacher on duty / Ansvarig lärare: David Broman, dbro@kth.se, +46 73 765 20 44
Examiner / Examinator: David Broman

Note that the exam questions are available both in English and in Swedish.

Instructions in English

- Allowed aids: One sheet of A4 paper with handwritten notes. You may write on both sides of the paper. You may bring this specific paper home after the exam. It is not a scrap paper.
- Explicitly forbidden aids: Textbooks, electronic equipment, calculators, mobile phones, machine-written pages, photocopied pages, pages of different size than A4.
- Please write and draw carefully. Unreadable text may lead to zero points.
- You may write your answers in either Swedish or English.

The exam consists of two parts:

- **Part I: Fundamentals:** The maximal number of points for Part I is 48 points (for IS1500) and 40 points (for IS1200). There are 8 points for each of the six course modules. All questions in Part I expect only short answers. At most a few sentences are needed.
- **Part II: Advanced:** There are in total three advanced questions, where you should discuss, analyze, or construct, and where the answers require clear motivations. For each exercise, the result can be fail (F), satisfactory (S), good (G), or very Good (VG).

Grades

To get a pass grade (A, B, C, D, or E), it is required to pass Part I of the exam. For IS1500 students, it is required to get at least 2 points on each module (excluding bonus points), and in total at least 36 points on Part I (including bonus points). For IS1200 students, it is required to get at least 2 points on each module (excluding bonus points), and to get at least 30 points in total on questions 1, 2, 4, 5, and 6 on Part I (including bonus points).

Grading scale (For both IS1200 and IS1500) [1]:

- A: Passed Part I, advanced project, and three VG **or** two VG and one G on part II.
- B: Passed Part I, advanced project, and one VG and two G **or** two VG and one S on part II.
- C: Passed Part I, and three G **or** two G and one S **or** one VG and two S **or** one VG and one G and one F **or** two VG and one F **or** one VG and one G and one S on part II.
- D: Passed Part I, and three S **or** one G and one S and one F **or** two G and one F **or** one VG and one S and one F **or** one VG and two F **or** one G and two S on part II.
- E: Passed Part I.
- FX: At least 36 points (for IS1500) or at least 30 points (for IS1200) on Part I, and at most one module with less than 2 points.
- F: otherwise

[1] For IS1500, the grading rules changed because of the grading criteria requirements. They have been updated after the exam to give fairer grades (see course PM). The IS1200 retake exam follows the rules from VT 2019.

Results

The result will be announced at latest 2020-04-01. If the students receives FX, it is possible to request a complementary examination. The examiner decides if it is oral or in written form. The student must request such examination at latest 2020-04-22 via email to `dbro@kth.se`.

Instruktioner på Svenska

- Tillåtna hjälpmedel: En A4-sida med handskrivna anteckningar. Det är tillåtet att skriva på båda sidorna. Det är tillåtet att få tillbaka detta papper efter examen. Det är inget kladdpapper.
- Förbjudna hjälpmedel: Läroböcker, elektroniska hjälpmedel, miniräknare, mobiltelefoner, maskinskrivna sidor, kopierade papper, sidor av andra storlekar än A4.
- Skriv och rita noggrant. Oläsbar text kan resultera i noll poäng.
- Du kan skriva dina svar på antingen engelska eller svenska.

Tentamen består av två delar:

- **Del I: Fundamentala delen:** Maximalt antal poäng för del I är 48 poäng (för IS1500) och 40 poäng (för IS1200). Totalpoängen per kursmodul är 8 poäng (6 moduler totalt). För del I förväntas det endast korta svar på frågorna. Endast ett fåtal meningar krävs.
- **Del II: Avancerade delen:** Det finns totalt tre avancerade övningar där det krävs att diskuterar, analyserar och konstruerar. Vidare krävs motiveringar. Varje övning kan resultera i icke godkänt (F), tillfredsställande (S), bra (G) och mycket bra (VG).

Betyg

För att erhålla godkänt betyg (A, B, C, D eller E) krävs att man får godkänt på del I. För IS1500-studenter krävs det minst 2 poäng på varje modul (exklusive bonuspoäng) samt 36 poäng eller mer på del I (inklusive bonuspoäng) för att få godkänt på tentamen. För IS1200-studenter krävs det minst 2 poäng på varje modul (exklusive bonuspoäng) samt 30 poäng eller mer på frågorna 1, 2, 4, 5 och 6 på del I (inklusive bonuspoäng) för att bli godkänd.

Betygsskala (För både IS1200 och IS1500):

- A: Godkänt del I, avancerat projekt, och tre VG **eller** två VG och ett G på del II.
- B: Godkänt del I, avancerat projekt, och ett VG och två G **eller** två VG och ett S på del II.
- C: Godkänt del I, och tre G **eller** två G och ett S **eller** ett VG och två S **eller** ett VG och ett G och ett F **eller** två VG och ett F **eller** ett VG och ett G och ett S på del II.
- D: Godkänt del I, och tre S **eller** ett G och ett S och ett F **eller** två G och ett F **eller** ett VG och ett S och ett F **eller** ett VG och två F **eller** ett G och två S på del II.
- E: Godkänt del I.
- FX: Minst 36 poäng (för IS1500) eller minst 30 poäng (för IS1200) på del I och som mest en modul med mindre än 2 poäng.
- F: i övriga fall

Resultat

Resultaten kommer att meddelas senast 2020-04-01. Vid FX är det möjligt att begära en komplementerande examination. Examinatorn bestämmer om examinationen är muntlig eller skriftlig. Studenten måste begära examination senast 2020-04-22 via epost till `dbro@kth.se`.

Part I: Fundamentals

1. **Module 1: C and Assembly Programming**

   (a) Suppose a 32-bit MIPS machine code with value `0x8c73fff8` is located at address `0x4310bb10`. Write out the MIPS assembly instruction representing the machine code. (3 points)

   (b) Suppose a function `foo` written in the C programming language has two parameters. The first parameter is an integer pointer, and the second parameter is an integer value. The function does not return any value. The function iterates over an integer array, and increments each element with the value given in the second argument. Assume that the array is always null terminated, i.e., the last element in the array has value 0. For instance, suppose we have an array `a` as follows:

   ```
   int a[] = {4,10,100,23,99,0};
   ```

   If function `foo` is called as

   ```
   foo(a,10);
   ```

   then array `a` is updated, and contains the following values after that it has finished executing the call to `foo`:

   ```
   14,20,110,33,109,0
   ```

   Write down C function `foo` according to the specification above. You must use pointer arithmetic, meaning that you are not allowed to use array indexing (using syntax `[]`). (5 points)

2. **Module 2: I/O Systems**

   (a) Suppose you are configuring a 16-bit timer that is clocked at 80 MHz. The prescale factor has been set to `1:8`. You would like the timer to have a period of 5 ms, that is, the timer wraps and starts on value zero again 200 times every second. What value should then the period register have? (2 points)

   (b) Suppose a 32-bit MIPS processor is used on a simple embedded system. The system has 4 push buttons, named A, B, C, and D. The push buttons are memory mapped to address `0xffffe440`. The status of the buttons are given at bit indices 7 through 4, where button A is given at index 7 and button D at index 4. A bit value 1 means that the button is currently pushed.

   The system has 8 LED lights, memory mapped at address `0x17880010`. The LED lights are located at bit indices 12 through 5. Note that index value 0 means the least significant bit, as usual. A bit value 1 means that the LED light is turned on. When writing to the memory mapped I/O port, you do not have to take into consideration if some of the other unused bits of the port are changed.

   Create a MIPS assembly program that loops forever. In the loop, read the status of the push buttons A and D, and turn on the LED lights to show the binary representation of the decimal value 13 if A is pressed. If A is released, value 13 should continue to be displayed, until button D is pressed. If D is pressed, all LED lights should be turned off. If button A is pressed again, value 13 should be displayed again, etc. If both A and D are pressed at the same time, the LEDs should be turned off. When the program starts, and no buttons are pressed, all LEDs should be turned off. Note that you can get points for a partially correct solution. (6 points)

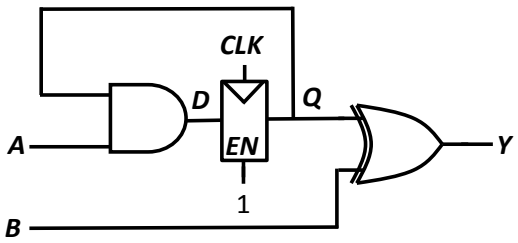
3. **Module 3: Logic Design (for IS1500 only)**

(a) Consider the following truth table, where A, B, and C are boolean variables. Write down a boolean algebra expression, using A, B, and C, that correctly represents the values of Y. The solution needs to be correct, but does not have to be in a simplified form. (2 points)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

(b) Suppose you have a register file with 3 read ports and 1 write port. The capacity is 128 bytes, that is, the number of bytes the register file can store in total. The address signals are 4 bits wide. How many bits wide is the write port, i.e., how many bits can be stored in parallel during one write? (2 points)

(c) Consider the circuit and the truth table below.



clock	*A*	*B*	*Q*	*D*	*Y*
init	1	1	1		
1st	0	0			

The first line of the truth table is the initial state before a clock tick. The second line is the stabilized values after the first clock tick. Copy the truth table and fill in the missing values. (4 points)

4. **Module 4: Processor Design**

(a) Consider the following MIPS assembly code, executing on a 5-stage pipeline without branch delay slots.

```
        sll     $s4,$s4,0
        sll     $s4,$s4,0
        sll     $s4,$s4,0
        sll     $s4,$s4,0
        addi    $s1,$zero,0xfff4    # F D E M W
        sw      $t0,4($s1)          #   F D E M W
        beq     $t0,$t1,skip        #     F D E M W
        lw      $t0,4($s2)          #       F D E M W
skip:
```

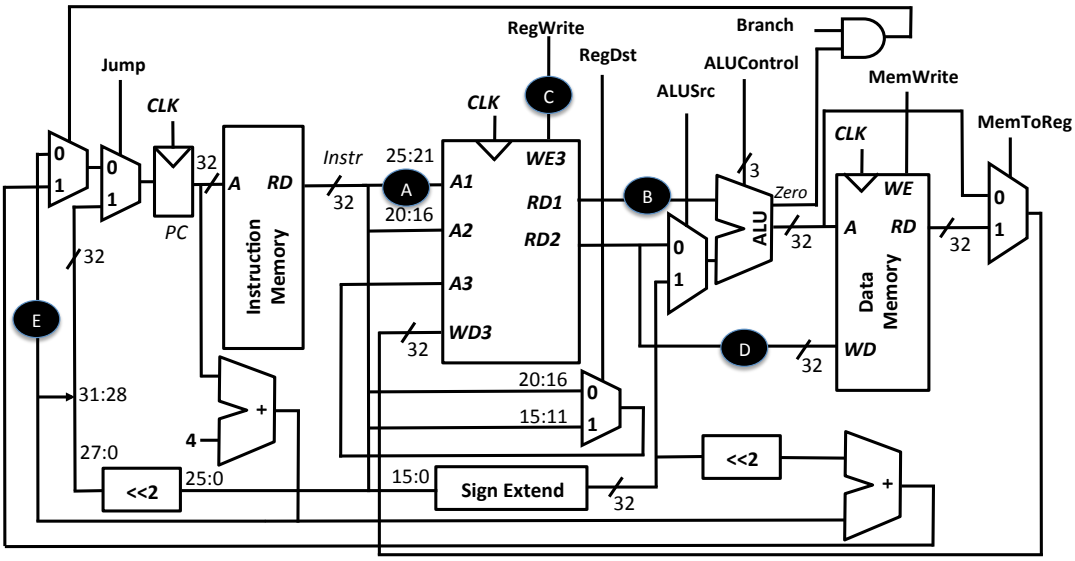
For each data hazard found in the last four instructions (exclude `sll` instructions), answer the following questions (3 points):

i. Between which instructions does the hazard occur?

ii. Which register is involved in the hazard?

iii. How can the hazard be solved with the lowest penalty of extra clock cycles?

(b) Consider the following datapath for a single-cycle 32-bit MIPS processor.



Suppose the `add` instruction is executing in the following program:

```
        lui     $t1,0x22
        ori     $t0,$t1,0x11
        sw      $t0,8($s1)
        addi    $t1,$s1,0
        lw      $s2,0($t1)
        add     $t8,$s2,$t0
```

Instruction `lui` is located at address `0x40004000`. What are then the signal values for A, B, C, D, and E when the `add` instruction is executed? For each signal, answer with either a **hexadecimal number** (not as decimal or binary numbers), or write `unknown` for a signal value where it is not possible to determine an exact value with the given information. (5 points)

5. **Module 5: Memory Hierarchy**

(a) Suppose you have a 4-way set associative instruction cache with the capacity 2048 bytes. The cache is used on a 32-bit MIPS processor. The cache has in total 256 blocks.

i. How many instructions are fetched if a cache miss occurs?

ii. How many sets are there in the cache?

iii. What is the tag size, in bits?

iv. How many valid bits are there totally in the cache?

(4 points)

(b) Suppose we have a 32-bit MIPS processor with separate data and instruction caches. Both caches are direct mapped, each with a capacity of 4096 bytes. The instruction cache has 256 sets, whereas the data cache has 512 sets. Consider the following code:

```
addi    $s0,0xe50
lw      $t0,4($s0)
lw      $t1,8($s0)
```

Assume that both caches are empty before the code above starts to execute. The `addi` instruction is located at address `0x40001004`. What is (i) the instruction cache miss rate, and (ii) the data cache miss rate when executing the three lines of code? (4 points)

6. **Module 6: Parallel Processors and Programs**

   (a) For each of the following five statements, determine if the statement is true or false. If you claim that the statement is false, answer shortly why the statement is false (max two sentences). You do not need to give a motivation if the statement is true.

      i. Data-level parallelism means that one instruction operates on one data item.

      ii. Advanced Vector Extension (AVX) is a well-known example of hardware multithreading.

      iii. Hardware multithreading results in concurrency and not in parallelism.

      iv. Very Long Instruction Word (VLIW) is a form of instruction-level parallelism (ILP).

      v. False sharing means that two software threads share the same page in virtual memory.

      (5 points)

   (b) Suppose you have a machine learning application that can be parallelized, but you are unsure to what degree. After some theoretical reasoning and some simple measurements you conclude that the theoretical maximal speedup is 5. When you run the program on a single core, the execution time is 30s. What is then the proportion (in percentage) of the program's execution time on one core that can be parallelized? (3 points)

Part II: Advanced

7. For each of the following three items, clearly explain: i) what the concepts mean, ii) the main differences, and iii) the similarities.

   (a) Advanced Vector Extension (AVX) vs. superscalar processors

   (b) Hyperthreading vs. multicore

   (c) Instruction-level parallelism vs. data-level parallelism

   The question is graded in three levels S, G, and VG, with the following criteria:

   - Satisfactory (S): Some of the concepts in the question are clearly explained.
   - Good (G): Basically all concepts in the question are clearly explained, and some of the concepts are related to each other, by discussing similarities and differences.
   - Very Good (VG): Basically all concepts in the question are clearly explained, and all of the concepts are related to each other, by discussing similarities and differences.

   If none of the three levels is achieved, the exercise is considered as failed (F).

8. Consider the following C code:

```
#include <stdio.h>

#define S_WIDTH 40    // Screen width
#define S_HEIGHT 10   // Screen height

#define F_CHARS 11    // Number of font characters (in this case A-K)
#define F_WIDTH 5     // Witdth of each font character, including space
#define F_HEIGHT 5    // Height of teach font character

char font[] = " xx  xxx   xxx xxx  xxxx xxxx  xx  x  x  x      x x  x "\
              "x  x x  x x    x  x x    x    x    x  x  x      x x x  "\
              "xxxx xxx  x    x  x xx   xxx  x xx xxxx  x      x xx   "\
              "x  x x  x x    x  x x    x    x  x x  x  x   x  x x x  "\
              "x  x xxx   xxx xxx  xxxx x     xx  x  x  x    xx  x  x ";

char screen[S_WIDTH*S_HEIGHT];
```

The array `font` declares the font for the first 11 characters in the alphabet (A-K). Note how the character `x` is used to show positions that should be filled. Note also how one long string is written on multiple lines by using character `'\'` on each line.

The defines made with `#define` can be used in the C program instead of the declared constant. For instance `S_WIDTH` can be used instead of the constant `40`. The array `screen` allocates space for the actual screen to where the text will be printed.

The following main program clears the screen (inserts period `'.'` characters for clarity), calls the function `myprint`, and prints out the whole `screen` buffer to standard output.

```
int main(){
  // Clear screen
  for(int i=0;i<S_WIDTH*S_HEIGHT;i++)
    screen[i] = '.';

  // Call my print with a user defined message
  myprint(5,2,"FAKED");

  // Print the screen
  char* spos = screen;
  for(int i=0; i<S_HEIGHT; i++){
    for(int j=0; j<S_WIDTH; j++)
      printf("%c",*spos++);
    printf("\n");
  }
  return 0;
}
```

Note that the function `myprint` takes three arguments: (1) the X position where the text should be printed out, (2) the Y position, and (3) the null-terminated string, which shows the text that will be printed out.

If the main program executes, the program prints out the following to the standard output:

```
........................................
........................................
.....xxxx  xx  x  x xxxx xxx  ..........
.....x    x  x x x  x    x  x ..........
.....xxx  xxxx xx   xx   x  x ..........
.....x    x  x x x  x    x  x ..........
.....x    x  x x  x xxxx xxx  ..........
........................................
........................................
........................................
```

Your task is to construct the C function `myprint` that prints out the message to buffer `screen`, using the font defined in buffer `font`. Note that you are only allowed to use pointer arithmetic, not array indexing (using the syntax `[]`). Hint: ASCII character `'A'` is encoded using the decimal value 65. The rest of the capital letters follows directly after.

The question is graded in three levels S, G, and VG, with the following criteria:

- Satisfactory (S): Some minor parts of a C or Assembly program are constructed correctly, but there are major errors or missing parts of the program.
- Good (G): The majority of the program is constructed correctly, including the main flow and structure of the program, but there are a number of minor errors within the program.
- Very Good (VG): The program is correct, with basically no errors.

If none of the three levels is achieved, the exercise is considered as failed (F).

9. Consider the following MIPS assembly code:

```
.data
lst:    .word  10,7

.text
main:
        la      $a0,lst
        la      $a1,2
        jal     sort

stop:   j       stop


sort:   addi    $a1,$a1,-1
        addi    $t0,$0,1
loop_outer:
        beq     $t0,$0,finished
        addi    $t0,$0,0
        addi    $t1,$0,0
loop_inner:
        slt     $t3,$t1,$a1
        beq     $t3,$0,loop_outer

        sll     $t4,$t1,2
        add     $t4,$t4,$a0
        lw      $t5,4($t4)
        lw      $t6,0($t4)
        slt     $t3,$t5,$t6
        beq     $t3,$0,no_action
        sw      $t5,0($t4)
        sw      $t6,4($t4)
        addi    $t0,$0,1
no_action:
        addi    $t1,$t1,1
        j       loop_inner

finished:
        jr      $ra
```

The assembly function `sort` performs sorting of elements in an array. The first argument to the function points to the array, and the second argument is the length of the array. Assume that the assembly code is executing on a 32-bit 5-stage pipelined MIPS processor without branch delay slots.

The following question is divided into three different sub-problems:

- L1: State the number of times a store word (`sw`) instruction is executed? Explain shortly why.
- L2: Identify all locations in the `sort` function where data hazards cannot be solved by only using forwarding. Explain why and write out explicitly in between which instructions these hazards occur, and which registers that are involved.

- L3: Assume that the processor has a direct mapped instruction cache with a capacity of 4096 bytes with 256 sets. The first `addi` instruction in the `sort` function is located at address `0x4000200C`. Assume that the instruction cache is empty before `main` is called. How many instruction cache misses occur when the `sort` function is executed (starting with the first `addi` instruction and ending with the `jr` instruction)?

The question is graded in three levels S, G, and VG, with the following criteria:

- Satisfactory (S): The task at level L1 is solved correctly.
- Good (G): Either the task at level L2 or the task at level L3 is solved correctly.
- Very Good (VG): Both the tasks at level L2 and L3 are solved correctly.

If none of the three levels is achieved, the exercise is considered as failed (F).

Del I: Grundläggande

1. **Modul 1: C-programmering och assemblerspråk**

   (a) Anta att minnescell med adress `0x4310bb10` innehåller 32-bitars MIPS-maskinkod med värdet `0x8c73fff8`. Skriv ut den MIPS-assemblerinstruktion som representerar maskinkoden. (3 poäng)

   (b) Anta att funktionen `foo` skriven i programspråket C har två parametrar. Den första parametern är en heltalspekare och den andra parametern är ett heltalsvärde. Funktionen returnerar inget värde. Funktionen itererar över en array med heltal, och räknar upp (inkrementerar) varje element med det värde som ges av det andra argumentet. Anta att arrayen alltid är nollterminerad, det vill säga att det sista elementet i arrayen har värdet 0. Som exempel, anta att vi har en array `a` enligt följande:

   ```
   int a[] = {4,10,100,23,99,0};
   ```

   Om funktionen foo anropas som

   ```
   foo(a,10);
   ```

   så uppdateras arrayen `a`, och innehåller följande värden efter att exekveringen av anropet till `foo` har avslutats:

   ```
   14,20,110,33,109,0
   ```

   Skriv ner C-funktionen `foo` enligt specifikationen ovan. Du måste använda pekararitmetik, vilket innebär att du inte får använda array-indexering (med syntaxen `[]`). (5 poäng)

2. **Modul 2: In- och utmatningssystem**

   (a) Anta att du konfigurerar en 16-bitars timer som klockas vid 80 MHz. Skalfaktorn (prescale factor) är satt till `1:8`. Du önskar att timern ska ha periodtiden 5 ms, det vill säga att timern ska slå runt och starta på värdet noll igen 200 gånger varje sekund. Vilket värde ska då periodregistret ha? (2 poäng)

   (b) Anta att en 32-bitars MIPS-processor används i ett enkelt inbyggt system. Systemet har 4 tryckknappar kallade A, B, C och D. Tryckknapparna är minnesmappade till adress `0xffffe440`. Status för knapparna ges på bitindex 7 till och med 4, där knapp A ges på index 7 och knapp D på index 4. Ett bitvärde på 1 betyder att knappen för tillfället är intryckt.

   Systemet har 8 LED-lampor, minnesmappade på adress `0x17880010`. LED-lamporna är placerade på bitindex 12 till och med 5. Notera att indexvärde 0 som vanligt betyder den minst signifikanta biten. Ett bitvärde på 1 betyder att LED-lampan tänds. Vid skrivning till den minnesmappade in- och utporten behöver du inte ta hänsyn till om några av portens oanvända bitar förändras.

   Skapa ett MIPS-assemblerprogram som går i en evig slinga (och loopar i evighet). I slingan, läs status för knapparna A och D, och tänd LED-lamporna för att visa den binära representationen av decimaltalet 13 om A är intryckt. Om A släpps, ska värdet 13 fortsätta visas tills knapp D trycks in. Om D trycks in ska alla LED-lamporna släckas. Om knapp A trycks in igen så ska värdet 13 visas igen, och så vidare. Om både A och D är intryckta samtidigt ska LED-lamporna släckas. När programmet startar och inga knappar är intryckta ska alla LED-lampor släckas. Observera att du kan få poäng för en lösning som är delvis riktig. (6 poäng)

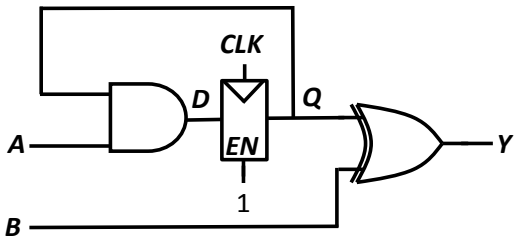
3. **Modul 3: Digital Design (endast för IS1500)**

(a) Betrakta följande sanningstabell, där A, B och C är Booleska variabler. Skriv ner ett uttryck i Boolesk algebra med A, B och C, som på ett riktigt sätt representerar värdena på Y. Lösningen måste vara korrekt men behöver inte vara i förenklad form. (2 poäng)

A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

(b) Anta att du har en registeruppsättning (register file) med 3 läsportar och 1 skrivport. Kapaciteten är 128 byte, det vill säga antalet byte som registeruppsättningen totalt kan lagra. Adressignalerna är 4 bitar breda. Hur många bitar bred är skrivporten, det vill säga hur många bitar kan lagras parallellt vid en skrivning? (2 poäng)

(c) Betrakta kretsen och sanningstabellen här nedanför.



clock	*A*	*B*	*Q*	*D*	*Y*
init	1	1	1		
1st	0	0			

Första raden i sanningstabellen är starttillståndet innan någon klockpuls. Andra raden är det stabiliserade värdet efter den första klockpulsen. Kopiera sanningstabellen och skriv in de värden som saknas. (4 poäng)

4. **Modul 4: Processorkonstruktion**

(a) Betrakta följande MIPS-assemblerkod som exekveras på en 5-stegs pipeline utan hoppfördröjning (without branch delay slots).

```
        sll     $s4,$s4,0
        sll     $s4,$s4,0
        sll     $s4,$s4,0
        sll     $s4,$s4,0
        addi    $s1,$zero,0xfff4    # F D E M W
        sw      $t0,4($s1)          #   F D E M W
        beq     $t0,$t1,skip        #     F D E M W
        lw      $t0,4($s2)          #       F D E M W
skip:
```

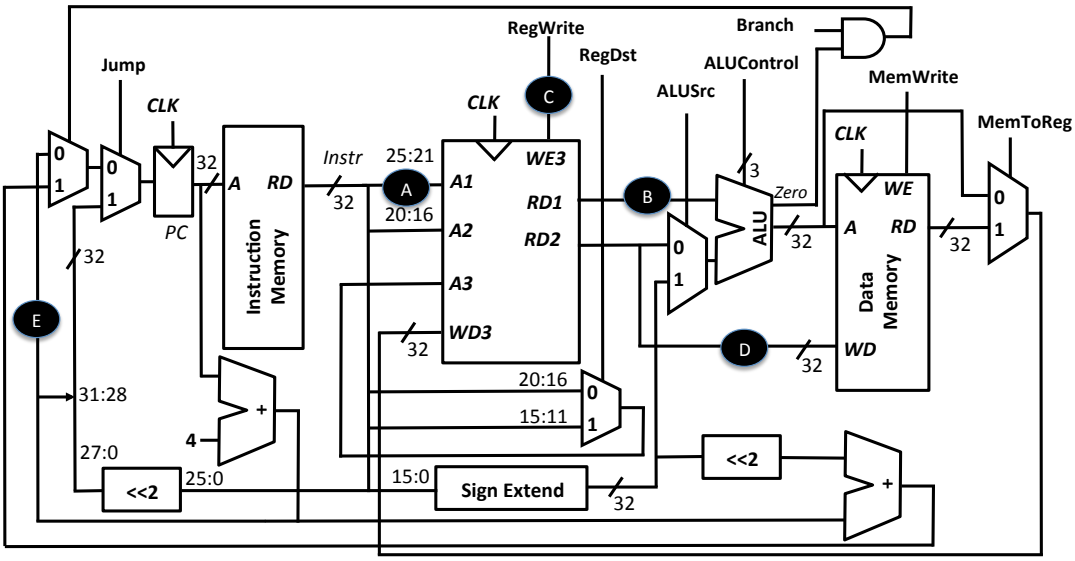
För varje data hazard som finns i de fyra sista instruktionerna (exkludera `sll` instruktioner), besvara följande frågor (3 poäng):

i. Mellan vilka instruktioner uppträder hazarden?

ii. Vilket register är involverat i hazarden?

iii. Hur kan hazarden lösas med lägsta straff i form av extra klockcyklers?

(b) Betrakta nedanstående dataväg för en 1-cykels, 32-bitars MIPS-processor.



Anta att instruktionen `add` exekveras i följande program:

```
        lui     $t1,0x22
        ori     $t0,$t1,0x11
        sw      $t0,8($s1)
        addi    $t1,$s1,0
        lw      $s2,0($t1)
        add     $t8,$s2,$t0
```

Instruktionen `lui` finns på adress `0x40004000`. Vad är då signalvärdena för A, B, C, D och E när `add`-instruktionen exekveras? För varje signal, svara med antingen ett **hexadecimalt tal** (inte ett decimalt eller binärt tal), eller skriv `okänd` för ett signalvärde där det inte går att avgöra ett exakt värde utifrån den givna informationen. (5 poäng)

5. **Modul 5: Minneshierarkier**

(a) Anta att du har ett 4-vägsassociativt instruktionscacheminne med kapaciteten 2048 byte. Cacheminnet används i en 32-bitars MIPS-processor. Cacheminnet har totalt 256 block.

i. Hur många instruktioner hämtas om en cachemiss inträffar?

ii. Hur många mängder (sets) finns det i cacheminnet?

iii. Vad är adressetikettens storlek (the tag size), räknat i bitar?

iv. Hur många giltigbitar (valid bits) finns det i cacheminnet?

(4 poäng)

(b) Anta att vi har en 32-bitars MIPS-processor med separata instruktions- och datacacheminnen. Båda cacheminnena är direktmappade, och vart och ett har en kapacitet på 4096 byte. Instruktionscacheminnet har 256 rader (sets), medan datacacheminnet har 512 rader (sets). Betrakta följande kod:

```
addi    $s0,0xe50
lw      $t0,4($s0)
lw      $t1,8($s0)
```

Anta att båda cacheminnena är tomma innan ovanstående kod börjar exekvera. Instruktionen `addi` finns på adress `0x40001004`. Vad är (i) misskvoten (miss rate) i instruktionscacheminnet, och (ii) misskvoten i datacacheminnet när de tre raderna kod exekveras? (4 poäng)

6. **Modul 6: Parallella processorer och program**

   (a) För vart och ett av följande fem påståenden, avgör om påståendet är sant eller falskt. Om du hävdar att påståendet är falskt, svara kort varför påståendet är falskt (med högst två meningar). Du behöver inte ge någon motivering om påståendet är sant.

      i. Dataparallellitet (data-level parallelism) betyder att en instruktion opererar på ett datavärde.

      ii. Advanced Vector Extension (AVX) är ett välkänt exempel på multitrådning i hårdvara (hardware multithreading).

      iii. Multitrådning i hårdvara (hardware multithreading) resulterar i samtidighet (concurrency) och inte i parallellism.

      iv. VLIW (Very Long Instruction Word) är en form av instruktionsnivåparallellitet (ILP, Instruction-Level Parallelism).

      v. Falsk delning (false sharing) betyder att två programvarutrådar delar på samma sida i det virtuella minnet.

      (5 poäng)

   (b) Anta att du har en tillämpning för maskininlärning som kan parallelliseras, men att du är osäker på i vilken utsträckning. Efter en del teoretiskt resonerande och några enkla mätningar drar du slutsatsen att den teoretiskt maximala speedup:en är 5. När du kör programmet på en enda kärna är exekveringstiden 30 s. Vad är då andelen (i procent) av programmets exekveringstid som kan parallelliseras (räknat på om man kör på en core)? (3 poäng) [2]

[2] Update March 18, 2020: The Swedish version of the task has been slightly updated, to better correspond to the English version.

Del II: Avancerat

7. För vart och en av följande tre punkter, förklara tydligt: i) vad begreppen betyder, ii) de huvudsakliga skillnaderna, och iii) likheterna.

   (a) Advanced Vector Extension (AVX), jämfört med superskalära processorer

   (b) Hyperthreading, jämfört med flerkärnighet (multicore)

   (c) Instruktionsnivåparallellitet (instruction-level parallelism), jämfört med dataparallellitet (data-level parallelism)

   Frågan betygsätts i tre nivåer S, G och VG med följande kriterier:

   - Tillfredsställande (Satisfactory, S): Vissa av begreppen i frågan är tydligt förklarade.
   - Bra (Good, G): I grund och botten är alla begrepp i frågan tydligt förklarade, och vissa av begreppen har relaterats till varandra genom att likheter och skillnader har diskuterats.
   - Mycket bra (Very Good, VG): I grund och botten är alla begrepp i frågan tydligt förklarade, och alla begrepp har relaterats till varandra genom att likheter och skillnader har diskuterats.

   Om ingen av de tre nivåerna har uppnåtts, betraktas uppgiften som underkänd (Failed, F).

8. Betrakta följande C-kod:

```
#include <stdio.h>

#define S_WIDTH 40    // Screen width
#define S_HEIGHT 10   // Screen height

#define F_CHARS 11    // Number of font characters (in this case A-K)
#define F_WIDTH 5     // Witdth of each font character, including space
#define F_HEIGHT 5    // Height of teach font character

char font[] = " xx  xxx   xxx xxx  xxxx xxxx  xx  x  x  x      x x  x "\
              "x  x x  x x    x  x x    x    x    x  x  x      x x x  "\
              "xxxx xxx  x    x  x xx   xxx  x xx xxxx  x      x xx   "\
              "x  x x  x x    x  x x    x    x  x x  x  x   x  x x x  "\
              "x  x xxx   xxx xxx  xxxx x     xx  x  x  x    xx  x  x ";

char screen[S_WIDTH*S_HEIGHT];
```

Arrayen `font` deklarerar teckensnittet för de första 11 tecknen i alfabetet (A-K). Notera att tecknet `x` används för att visa positioner som ska fyllas. Notera också hur en long sträng skrivs på flertalet rader genom att använda tecknet `'\'` på varje rad.

Definitioner gjorda med `#define` kan användas i C-programmet i stället för den deklarerade konstanten. Exempelvis kan `S_WIDTH` användas i stället för konstanten `40`. Arrayen `screen` reserverar plats för den verkliga skärmen där texten ska skrivas ut.

Följande huvudprogram rensar skärmen (lägger in punkt-tecken `'.'` för tydlighetens skull), anropar funktionen `myprint` och skriver ut hela bufferten `screen` till standard output.

```
int main(){
  // Clear screen
  for(int i=0;i<S_WIDTH*S_HEIGHT;i++)
    screen[i] = '.';

  // Call my print with a user defined message
  myprint(5,2,"FAKED");

  // Print the screen
  char* spos = screen;
  for(int i=0; i<S_HEIGHT; i++){
    for(int j=0; j<S_WIDTH; j++)
      printf("%c",*spos++);
    printf("\n");
  }
  return 0;
}
```

Notera att funktionen `myprint` tar tre argument: (1) X-positionen där texten ska skrivas ut, (2) Y-positionen och (3) den nullterminerade sträng som visar den text som ska skrivas ut.

Om huvudprogrammet exekveras så skriver programmet ut följande på standard output:

```
........................................
........................................
.....xxxx  xx  x  x xxxx xxx  ..........
.....x    x  x x x  x    x  x ..........
.....xxx  xxxx xx   xx   x  x ..........
.....x    x  x x x  x    x  x ..........
.....x    x  x x  x xxxx xxx  ..........
........................................
........................................
........................................
```

Din uppgift är att konstruera C-funktionen `myprint`, som skriver ut meddelandet till bufferten `screen`, med användning av teckensnittet definierat i bufferten `font`. Notera att du endast får använda pekararitmetik, inte arrayindexering (med syntaxen `[]`). Tips: Ascii-tecknet `'A'` kodas med det decimala värdet 65. Resten av de stora bokstäverna följer direkt efter.

Frågan betygsätts i tre nivåer S, G och VG med följande kriterier:

- Tillfredsställande (Satisfactory, S): Vissa mindre delar av ett program i C eller assembler har konstruerats korrekt, men det finns antingen stora fel i programmet eller stora delar som saknas.
- Bra (Good, G): Huvuddelen av programmet har konstruerats korrekt, inklusive huvudflödet och strukturen i programmet, men det finns ett antal mindre fel i programmet.
- Mycket bra (Very Good, VG): Programmet är korrekt och har i grund och botten inga felaktigheter.

Om ingen av de tre nivåerna har uppnåtts, betraktas uppgiften som underkänd (Failed, F).

9. Betrakta följande assemblerkod för MIPS:

```
.data
lst:    .word  10,7

.text
main:
        la      $a0,lst
        la      $a1,2
        jal     sort

stop:   j       stop


sort:   addi    $a1,$a1,-1
        addi    $t0,$0,1
loop_outer:
        beq     $t0,$0,finished
        addi    $t0,$0,0
        addi    $t1,$0,0
loop_inner:
        slt     $t3,$t1,$a1
        beq     $t3,$0,loop_outer

        sll     $t4,$t1,2
        add     $t4,$t4,$a0
        lw      $t5,4($t4)
        lw      $t6,0($t4)
        slt     $t3,$t5,$t6
        beq     $t3,$0,no_action
        sw      $t5,0($t4)
        sw      $t6,4($t4)
        addi    $t0,$0,1
no_action:
        addi    $t1,$t1,1
        j       loop_inner

finished:
        jr      $ra
```

Assemblerfunktionen `sort` utför sortering av elementen i en array. Det första argumentet till funktionen pekar på arrayen, och det andra argumentet är längden på arrayen. Anta att assemblerkoden exekveras på en 32-bitars MIPS-processor med 5-stegs pipeline, men utan hoppfördröjning (without branch delay slots).

Följande fråga är uppdelad i tre olika delproblem:

- L1: Ange antalet gånger som en store-word-instruktion (`sw`) exekveras? Förklara kort varför.
- L2: Ange alla platser i funktionen `sort`, där data-hazarder inte kan lösas med enbart forwarding. Förklara varför och skriv uttryckligen mellan vilka instruktioner dessa hazarder uppträder, och vilka register som är inblandade.

- L3: Anta att processorn har ett direktmappat instruktionscacheminne med kapaciteten 4096 byte i 256 rader (sets). Den första `addi`-instruktionen i funktionen `sort` finns på adress `0x4000200C`. Anta att instruktionscacheminnet är tomt innan `main` anropas. Hur många missar inträffar i instruktionscacheminnet när funktionen `sort` exekveras (där den första `addi`-instruktionen utförs först och instruktionen `jr` utförs sist)?

Frågan betygsätts i tre nivåer S, G och VG med följande kriterier:

- Tillfredsställande (Satisfactory, S): Uppgiften på nivå L1 har lösts korrekt.
- Bra (Good, G): Antingen uppgiften på nivå L2 eller uppgiften på nivå L3 har lösts korrekt.
- Mycket bra (Very Good, VG): Både uppgiften på nivå L2 och uppgiften på nivå L3 har lösts korrekt.

Om ingen av de tre nivåerna har uppnåtts, betraktas uppgiften som underkänd (Failed, F).

MIPS Reference Sheet

David Broman, KTH Royal Institute of Technology
Version 1.16, December 21, 2018

INSTRUCTIONS (SUBSET)

Name (format, op, funct)	Syntax	Operation
add (R,0,32)	`add rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
add immediate (I,8,na)	`addi rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add immediate unsigned (I,9,na)	`addiu rt,rs,imm`	reg(rt) := reg(rs) + signext(imm);
add unsigned (R,0,33)	`addu rd,rs,rt`	reg(rd) := reg(rs) + reg(rt);
and (R,0,36)	`and rd,rs,rt`	reg(rd) := reg(rs) & reg(rt);
and immediate (I,12,na)	`andi rt,rs,imm`	reg(rt) := reg(rs) & zeroext(imm);
branch on equal (I,4,na)	`beq rs,rt,label`	if reg(rs) == reg(rt) then PC = BTA else NOP;
branch on not equal (I,5,na)	`bne rs,rt,label`	if reg(rs) != reg(rt) then PC = BTA else NOP;
jump and link register (R,0,9)	`jalr rs`	$ra := PC + 4; PC := reg(rs);
jump register (R,0,8)	`jr rs`	PC := reg(rs);
jump (J,2,na)	`j label`	PC := JTA;
jump and link (J,3,na)	`jal label`	$ra := PC + 4; PC := JTA;
load byte (I,32,na)	`lb rt,imm(rs)`	reg(rt) := signext(mem[reg(rs) + signext(imm)]$_{7:0}$);
load byte unsigned (I,36,na)	`lbu rt,imm(rs)`	reg(rt) := zeroext(mem[reg(rs) + signext(imm)]$_{7:0}$);
load upper immediate (I,15,na)	`lui rt,imm`	reg(rt) := concat(imm, 16 bits of 0);
load word (I,35,na)	`lw rt,imm(rs)`	reg(rt) := mem[reg(rs) + signext(imm)];
multiply, 32-bit result (R,28,2)	`mul rd,rs,rt`	reg(rd) := reg(rs) * reg(rt);
nor (R,0,39)	`nor rd,rs,rt`	reg(rd) := not(reg(rs) | reg(rt));
or (R,0,37)	`or rd,rs,rt`	reg(rd) := reg(rs) | reg(rt);
or immediate (I,13,na)	`ori rt,rs,imm`	reg(rt) := reg(rs) | zeroext(imm);
set less than (R,0,42)	`slt rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than unsigned (R,0,43)	`sltu rd,rs,rt`	reg(rd) := if reg(rs) < reg(rt) then 1 else 0;
set less than immediate (I,10,na)	`slti rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0;
set less than immediate unsigned (I,11,na)	`sltiu rt,rs,imm`	reg(rt) := if reg(rs) < signext(imm) then 1 else 0; (inequality < compares using unsigned values)
shift left logical (R,0,0)	`sll rd,rt,shamt`	reg(rd) := reg(rt) << shamt;
shift left logical variable (R,0,4)	`sllv rd,rt,rs`	reg(rd) := reg(rt) << (reg(rs))$_{4:0}$;
shift right arithmetic (R,0,3)	`sra rd,rt,shamt`	reg(rd) := reg(rt) >>> shamt;
shift right logical (R,0,2)	`srl rd,rt,shamt`	reg(rd) := reg(rt) >> shamt;
shift right logical variable (R,0,6)	`srlv rd,rt,rs`	reg(rd) := reg(rt) >> (reg(rs))$_{4:0}$;
store byte (I,40,na)	`sb rt,imm(rs)`	mem[reg(rs) + signext(imm)]$_{7:0}$:= reg(rt)$_{7:0}$;
store word (I,43,na)	`sw rt,imm(rs)`	mem[reg(rs) + signext(imm)] := reg(rt);
subtract (R,0,34)	`sub rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
subtract unsigned (R,0,35)	`subu rd,rs,rt`	reg(rd) := reg(rs) - reg(rt);
xor (R,0,38)	`xor rd,rs,rt`	reg(rd) := reg(rs) ^ reg(rt);
xor immediate (I,14,na)	`xori rt,rs,imm`	reg(rt) := reg(rs) ^ zeroext(imm);

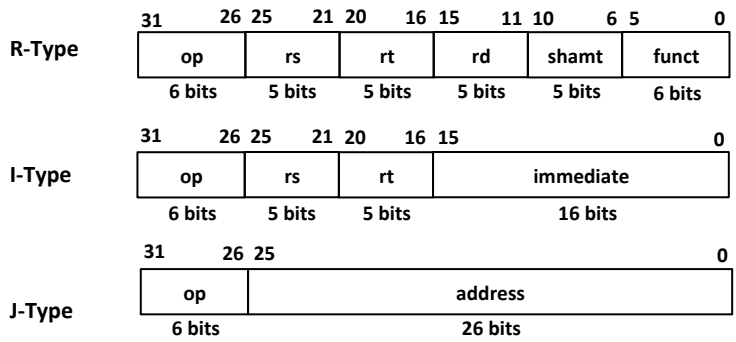
PSEUDO INSTRUCTIONS (SUBSET)

Name	Example	Equivalent Basic Instructions
load address	`la $t0,label`	`lui $at,hi-bits-of-address` `ori $t0,$at,lower-bits-of-address`
load immediate	`li $t0,0xabcd1234`	`lui $at,0xabcd` `ori $t0,$at,0x1234`
branch if less or equal	`ble $t0,$t1,label`	`slt $at,$t1,$t0` `beq $at,$zero,label`
move	`move $t0,$t1`	`add $t0,$t1,$zero`
no operation	`nop`	`sll $zero,$zero,0`

ASSEMBLER DIRECTIVES (SUBSET)

data section	`.data`
ASCII string declaration	`.ascii "a string"`
word alignment	`.align 2`
word value declaration	`.word 99`
byte value declaration	`.byte 7`
global declaration	`.global foo`
allocate X bytes of space	`.space X`
code section	`.text`

INSTRUCTION FORMAT



REGISTERS

Name	Number	Description
$0, $zero	0	constant value 0
$at	1	assembler temp
$v0	2	function return
$v1	3	function return
$a0	4	argument
$a1	5	argument
$a2	6	argument
$a3	7	argument
$t0	8	temporary value
$t1	9	temporary value
$t2	10	temporary value
$t3	11	temporary value
$t4	12	temporary value
$t5	13	temporary value
$t6	14	temporary value
$t7	15	temporary value
$s0	16	saved temporary
$s1	17	saved temporary
$s2	18	saved temporary
$s3	19	saved temporary
$s4	20	saved temporary
$s5	21	saved temporary
$s6	22	saved temporary
$s7	23	saved temporary
$t8	24	temporary value
$t9	25	temporary value
$k0	26	reserved for OS
$k1	27	reserved for OS
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Definitions

- Jump to target address:
  JTA = concat((PC + 4)$_{31:28}$, address(label), 00_2)
- Branch target address:
  BTA = PC + 4 + signext(imm) * 4

Clarifications

- All numbers are given in decimal form (base 10).
- Function signext(x) returns a 32-bit sign extended value of x in two's complement form.
- Function zeroext(x) returns a 32-bit value, where zero are added to the most significant side of x.
- Function concat(x, y, …, z) concatenates the bits of expressions x, y, …, z.
- Subscripts, for instance $X_{8:2}$, means that bits with index 8 to 2 are spliced out of the integer X.
- Function *address*(x) is the 26-bit address field value of the J-Type instruction for an address label x.
- NOP and na mean "no operation" and "not applicable", respectively.
- shamt is an abbreviation for "shift amount", i.e. how many bits that should be shifted.
- `addu` and `addiu` are misnamed *unsigned* because an add operation handles both signed and unsigned numbers in the same way. The term unsigned is actually used to describe that the instruction does not throw overflow exceptions.