EP1200 Introduktion till datorsystemteknik

Tentamen tisdagen den 3 juni 2014, 14.00 till 18.00

- Inga hjälpmedel är tillåtna utom de som följer med tentamenstexten
- Skriv kurskod, namn och personnummer på alla ark som lämnas in.
- Svara på svenska eller engelska. Oläsliga och oförståeliga svar ger ingen poäng. Svara med välstrukturerade, korta och koncisa svar. Alla svar måste vara motiverade; all kod väl kommenterad.
- Poängen anges för varje uppgift.
- Lösningsförslag anslås på kursens hemsida på KTH Social efter tentamens slut.

1. Give a short answer for each question. (Each correct answer gives 0.5 point.)
   a. Explain what CISC and RISC architectures are!
   CISC, complex instruction set computing, refers to architectures with an extensive instruction set supported by the complex CPU hardware, thus achieving computation with little memory needs. RISC, reduced instruction set computing, refers to architectures with a limited instruction set and simple hardware. Complexity here is instead in the software realization of the instructions.
   b. What are the front and back ends of a modern compiler?
   Front-end refers to the compilation from the high-level language to an intermediate representation (the VM in the case of Jack). The back-end is the platform specific compilation from the intermediate representation to assembly language.
   c. Explain the *tokenizing* and *parsing* steps of the Jack compiler!
   Tokenizing divides the code into terminal elements, tokens. The parser evaluates whether the input is valid (correct) Jack code and at the same time identifies non-terminals and terminals and discovers the program structure. (Parsing has a different meaning in the assembler!)
   d. Give three examples of operating system functions that are *not* part of the Hack OS (but found in Linux, Windows)!
   Multi-threading and multi-processing, access to mass storage devices, file system structure, command line or graphical user interface, implementation of security levels, networking. Mathematical functions do not belong to the OS, but are part of the high level language standard libraries.)
   e. What is a context-free grammar of a high-level programming language?
   The context-free grammar gives the rules on how syntactic elements can be formed from other, simpler elements, that is, how non-terminals can be formed from other non-terminals and terminals.
   f. What is the difference between the stack and the heap?
   Both are part of the RAM. Access to the stack is very limited, new elements can be put on the top of the stack, or removed from the top of the stack. The stack is the working memory of the VM. The heap is another part of the RAM, from which memory blocks can be allocated dynamically using memory allocation functions. These memory blocks are used for storing objects.
   g. What is the content of the A- and D-registers and of the PC after the following operation?

```
 @0
0;JMP
```

The first command puts 0 in the A-register. The second is an unconditional jump command, which makes the program continue from the ROM address given in the A register. The execution of the jump operation means that the PC is set to the value held in the A-register, that is, the PC will contain 0 as well. The D the register is not affected, and will keep the value it had before these instructions.

h. What is the expression "*3 4 x * + y x / +*" when written in infix notation?
(3+4*x)+y/x

i. Are the following instructions valid in HACK Assembly? Motivate your answer!

```
@i
M=1
```

```
@i
M=5
```

The first one is valid, since there is a C-instruction, where the ALU computation result is 1, and it can be stored in destination M. The second one is not valid, because the ALU cannot compute 5 as output without any input.

j. Comment on the following line of VM code: *pop constant 128* .
This line of code is not a valid VM command. The argument of pop should give where the value popped from the stack should be stored, and hence it cannot be the constant segment.

2. Write a program in Hack assembly that takes an integer in two's complement and multiplies it by the constant 2. The input value is stored in R0; the result is stored in R1; R2 is true (1) if the multiplication led to overflow (and consequently the result in R1 is not correct), otherwise false (0).
   a. Explain how the two's complement coding scheme is used. What is the largest positive and what is the smallest negative number the Hack computer can handle? (1p)
   b. Write the assembly code! (3p)
   c. Provide the entries of the symbol table for the assembler, in addition to the predefined symbols! (1p)

   a. In two's complement the MSB defines whether the number is positive or negative (0/1). The largest positive number is (0)111…1, the smallest negative number is 1000..00. For n bits representation the largest positive number is 2^{n-1}-1, the largest negative is -2^{n-1}. In the Hack architecture, words are 16 bits long. Therefore, the smallest negative number is -2^{15} (1000 0000 0000 0000), the largest positive number is 2^{15}-1 (0111 1111 1111 1111)
   b. Overflow happens, if the operation gives a number that is smaller than the smallest possible number of larger than the largest possible number. Then the number will be misinterpreted according to the two's complement scheme. We show a possible Assembly code, based on comparison with the smallest and largest numbers that can be processed without overflow. Other nice solution you had was based on the comparison of the signs of the original number and the result. Some of you suggested to subtract from the result and compare, but unfortunately that solution does not work, as even the subtractions will be incorrectly performed.

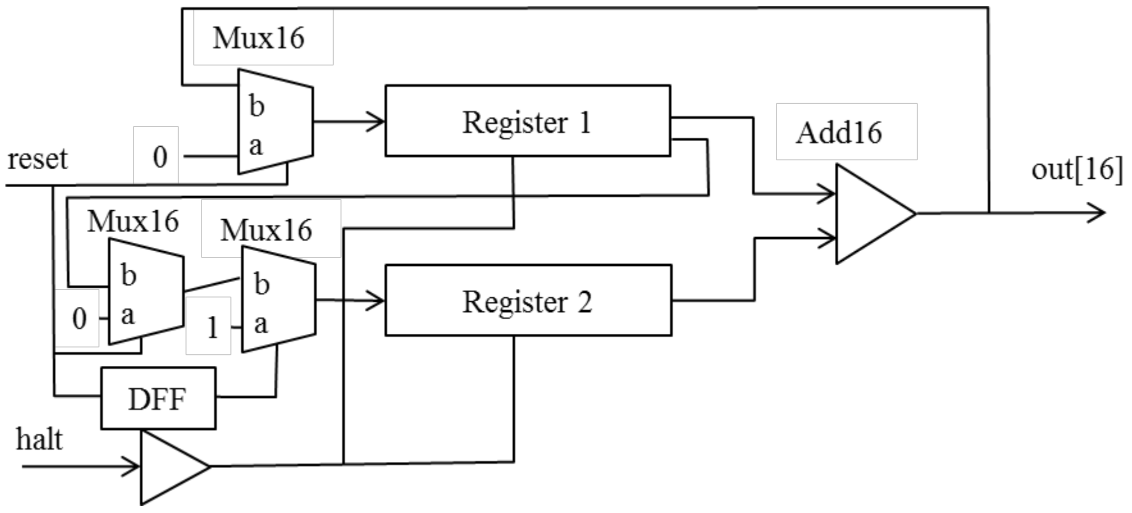
```
//  multiplies R0 by 2, stores in R1, sets R2 true (1)
otherwise false (0)

0           @R0  // here we multiply by two
1           D=M
2           D=D+M
2           @R1
4           M=D
5           @R2   // here we evaluate overflow
6           M=0
7           @R0   // check for positive numbers
8           D=M
9           @16383
10          D=D-A
11          @OVERFLOW
12          D;JGT
13          @R0   // check for negative numbers
14          D=M
15          @16384
16          D=D+A
17          @OVERFLOW
18          D;JLT
19          @END
20          0;JMP
(OVERFLOW)
21          @R2
22          M=1
(END)
23          @END
24          0;JMP   // Infinite loop
```

c. Symbol table for the assembly code, in addition to the pre-defined symbols: the code does not use any user defined variables, only two labels. Labels point to a ROM location, this location is given in the symbol table. For help, we give the ROM addresses in the program.

OVERFLOW	21
END	23

3. Build a chip that counts according to the Fibonacci series. The chip has one control signal, *reset*, that starts the counting and one signal, *halt*, to pause the counting. The Fibonacci series is defined as: $out(0) = out(1) = 1; out(t) = out(t-1) + out(t-2), t \geq 2$. *You can* use the chips available in the Hack chipset.

   a. Block diagram:



   b. Chip in HDL:

```
CHIP FibonacciCounter {
IN reset, halt;
OUT out[16];
PARTS:

Not (in=halt, out=nothalt1, out=nothalt2);
Mux16 (a=reg1in, b[0..15]=false, sel=reset, out=reg1in2);
Mux16 (a=reg2in, b[0..15]=false, sel=reset, out=reg2mux);
DFF (in=reset, out=reset1);
Mux16 (a=reg2mux, b[0]=true, b[1..15]=false, sel=reset1, out=reg2in2);
Register (load=nothalt1, in=reg1in2, out=add1, out=reg2in);
Register (load=nothalt2, in=reg2in2, out=add2);
Add16 (a=add1, b=add2, out=reg1in, out=out);

}
```

4. Consider that you have a recursive implementation for printing the items in a linked list (as the one implemented in the Hack project during the course). The code is the following:

```
method void print() {
  do Output.printString(" -> ");
  do Output.printInt(data);
  if (~(next =  null)) {
    do next.print();
  }
  return;
}
```

The method is defined within the List class. Following standard practice, when the program starts the stack pointer SP is initialized to SP=256 and the base of the heap is set to address 1024, this is where memory for the first List object will be allocated.

a. Assume you want to print all items in the list. How many entries can there be in the linked list at most in order for your program to execute correctly? Give a reasonable upper bound, and motivate your answer! (3p)

b. Provide a different implementation that would be able to print a longer list! (2p)

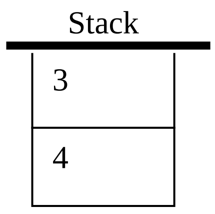
a. The number of entries is limited by the stack size. Every time you call print(), the following happens to the stack in assembly: THIS is pushed, then 5 values are pushed (return address, LCL, ARG, THIS, THAT). That is, every recursive invocation of print() occupies 6 positions on the stack. Assuming your stack was empty when print() was called first, there are 1024-256=768 empty stack positions, thus the method can be invoked at most 768/6=128 times, and this is the maximum number of entries. If there are more you will have a stack overflow.

b. An iterative implementation avoids the above problem. For example:

```
method void print() {
  var List curItem;
  let curItem=this;
  while (!(curItem=null)) {
    do Output.printString(" -> ");
    do Output.printInt(data);
    let curItem = next;
  }
  return;
}
```

5. Consider the following VM code defining myFunction. The content of the stack is as shown in the figure, and the next line to be executed in the VM program is “call myFunction 2”.
   a. What will be the content of the stack after the function returns? (3p)
   b. Observe that in the function definition it says “function myFunction 1“ while when the function is called it says “call myFunction 2”. Why is the integer value that follows the function name different in these two cases? (2p)

```
function myFunction 1
push arg 0
pop local 0
label firstLabel
push arg 1
push local 0
add
pop local 0
push local 0
push constant 10
lt
if-goto firstLabel
push local 0
return
```



a. The function takes argument 0 and adds argument 1 until the sum gets larger or equal than 10. Therefore, the stack content when the function returns is 11.
b. Because in case of the definition “1” stands for the number of local variables, which need to be initialized. When calling the function the number “2” is the number of arguments, which the caller needs to know for properly managing the stack.

6. Consider the following Jack statement and the information available in the symbol table.

```
do fract.initialize(j+1,5);
```

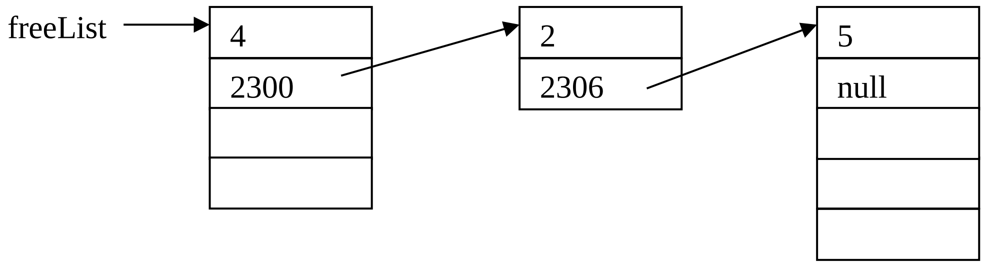
Name	Type	Kind	#
j	int	field	0
i	int	argument	0
fract	Fraction	local	0

a. Explain the information that is available in the given symbol table. (1p)
b. Give the VM code of the Jack statement (4p)

a. The symbol table gives the type of an identifier, which affects the memory space it needs, and the kind of it, which affects its life-cycle. The table below contains information about integer j, which is the first f ield variable, integer i, the first argument, and fract, which is a local variable for an object of the Fraction class.
b. To write the VM code, recognize that object fract has to be passed as argument.

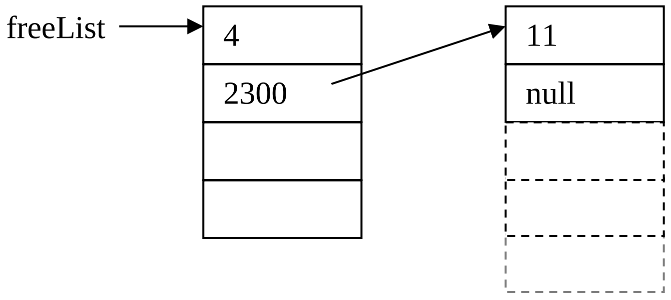
```
push local 0
push this 0
push constant 1
add
push constant 5
call Fraction.initialize 3
```

7. Consider a memory allocation algorithm that, upon allocation, uses the memory word that precedes the returned memory segment to save the size of the allocated block. Such an algorithm is analogous to the one described in the book.
   Consider that during the execution of a program the linked list that keeps track of the free memory segments looks like the following.



   a. Given the free list above, would the call `Memory.alloc(5)` return successfully? (1p)
   b. How does the linked list look like immediately after the function call `Memory.dealloc(2303)` returns? Assume that Memory[2302]=4 and that the allocation algorithm keeps the memory defragmented.

   a. No, in the given free list, a block of 5 memory words can not be allocated, since it requires 6 memory words, the first storing the length of the segment.
   b. The deallocated segment can be merged with the second, and consequently even with the third segment. Then the final list looks like the following.


freeList
4
2300
11
null