UNIVERSITY OF BRISTOL

PRACTICE PAPER

School of Computer Science

Examination for the Degree of
Master of Science in Computer Science (conversion)

COMSM1302
Overview of Computer Architecture

TIME ALLOWED:
2 Hours

Answers

Other Instructions

1. Make sure you carefully read any instructions on the question paper and answer booklet.
2. You must use the answer booklet to answer all questions.
3. This is a closed-book examination; books, notes, or other resources are forbidden.
4. A reference sheet will be given out alongside the question paper.
5. Any of the question paper can be used for rough work, but only the answer booklet will be marked.
6. A non-programmable calculator may be used for this examination.

TURN OVER ONLY WHEN TOLD TO BEGIN WORK

Section 1 — Short-answer questions (50 marks total)

You do not need to justify your answers for any of the questions in this section, and you will not receive partial credit for showing your reasoning. Just write your answers down in the shortest form possible, e.g. "A" for multiple-choice questions, "True" for true/false questions, or "23" for numerical questions. If you do display working, circle or otherwise indicate your final answer, as if it cannot be identified then the question will not be marked.

Question 1 (5 marks)

(a) Write a single Hack assembly instruction that stores the value 32 in `A`.

Solution: `@32`.

(b) Write a single Hack assembly instruction that stores $-$`D` in `M`, then jumps if `D` $= 0$.

Solution: `M=-D;JEQ`.

Question 2 (4 marks)

Which of the following memory segments in Hack VM would be most appropriate, when compiling from a high-level language such as C, to store a global variable?

A. `this`.

B. `that`.

C. `pointer`.

D. `local`.

E. `static`.

F. It depends on the context.

Solution: **E**, no partial credit.

The `static` memory segment is unaffected by function calls and returns and always maps to the same location of physical memory - so e.g. the Hack VM command "push static 1" will push from the same memory location no matter where it appears in the code. This is exactly the behaviour we need for storing global variables. Meanwhile, the `this`, `that` and `local` segments all map to different locations in memory depending on the context, and the `pointer` segment has another important use (and is only two words long in any case).

Question 3 (5 marks)

In an **assembler**, which of the following tasks would typically be carried out during lexing? You can pick more than one option.

A. Converting the program from a string into a sequence of tokens.

B. Constructing a symbol table mapping labels to memory addresses.

C. Constructing a symbol table mapping variables to memory addresses.

D. Converting each line of assembly into machine code.

E. Optimising the resulting machine code.

Solution: **A and B**, one mark per correctly-categorised option. (So 1 mark for including A, 1 mark for not including D, and so on.)

A. This is the main purpose of lexing.

B. In an assembler specifically, it's usually convenient to do this during lexing - since one line of code corresponds to one machine code instruction, it's easy to compute ROM addresses on the first pass through the file.

C. This normally happens after lexing, so we can use the labels symbol table to distinguish between variables and labels.

D. This happens after lexing, in the parsing stage.

E. There is generally no need to optimise the resulting machine code, since one line of assembly already corresponds to one line of machine code. (This is a bit of a lie — for a very large and complex instruction set like x64, sometimes some basic optimisation is possible by replacing one variant of an instruction with a faster or more space-efficient variant — but there is certainly no need to do this with Hack assembly.) If optimisation did happen, it would happen after lexing.

Question 4 (4 marks)

What are the binary values of the comp, dest, and jump operands for the C-instruction `AM=!M`? Your answers should include any leading 0s, where appropriate.

Solution: Comp is 1110001, dest is 101, jump is 000. Two marks for comp, one mark each for dest and jump, one mark partial credit for comp if a was omitted.

We can read this off from the reference sheet: comp is `!M`, which has $a = 1$ and $c = 110001$; dest is `AM`, which translates to 101; and there is no jump, which translates to 000.

Question 5 (5 marks)

Which of the following are properties of an ISA, and which are properties of a microarchitecture?

A. Word length and address space of memory,

B. Energy efficiency.

C. Clock speed.

D. Number of transistors used.

E. Addressing modes supported.

Solution: A and E are ISA properties, B, C and D are microarchitecture properties. One mark per correct choice.

The microarchitecture is the physical design of the computer in hardware — circuit diagrams and PCB layouts. The instruction set architecture (ISA) is the way the computer acts in response to machine code instructions. Therefore word length, address space of memory, and addressing modes supported are each properties of an ISA. Clock speed, number of transistors used, and energy efficiency are each properties of a microarchitecture. NB: different ISAs can be more well-suited to being energy efficient, e.g. MIPS is more well-suited to low energy requirements than x64, but the actual energy efficiency is down to physical implementation.

Question 6 (4 marks)

What value will the following sequence of VM operations leave on top of the stack?

```
push constant 4
push constant 5
sub
not
push constant 0
eq
```

Solution: −1, 0xFFFF, or `true`. No partial credit. Here's the contents of the stack after each command, with the bottom of the stack at the left and the top at the right.

```
Initial stack: empty
push 4: 4
push 5: 4, 5
sub: -1
not: 0
push 0: 0, 0
eq: -1
```

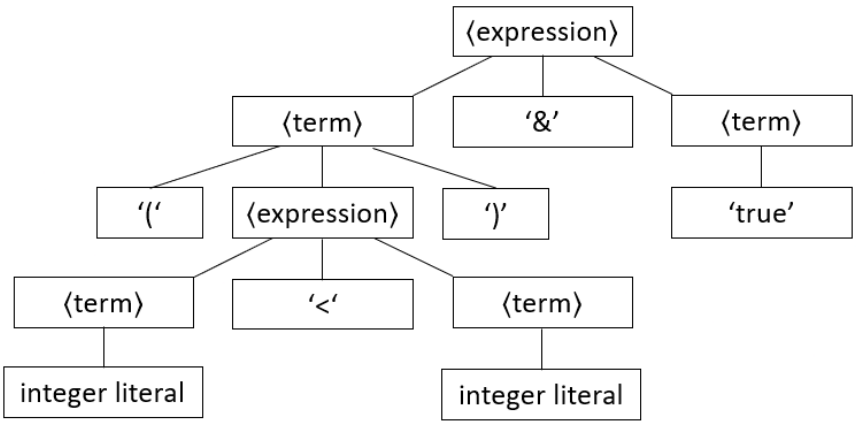
Question 7 (5 marks)

Below is part of the Jack grammar:

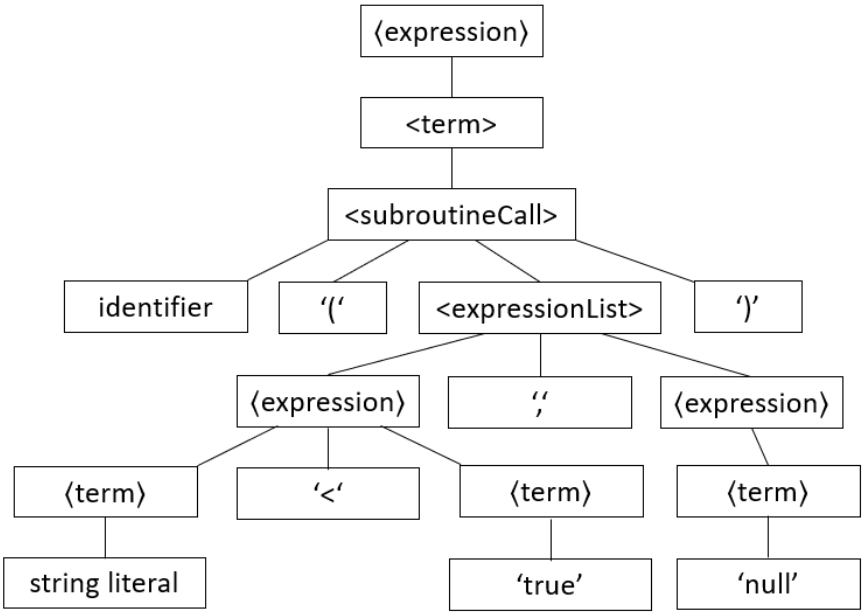
⟨`expression`⟩ ::= ⟨`term`⟩, {(‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`&`’ | ‘`|`’ | ‘`<`’ | ‘`>`’ | ‘`=`’), ⟨`term`⟩};
⟨`term`⟩ ::= integerliteral | stringliteral | ‘`true`’ | ‘`false`’ | ‘`null`’ | ‘`this`’ | identifier, [‘`[`’, ⟨`expression`⟩, ‘`]`’] | ‘`(`’, ⟨`expression`⟩, ‘`)`’ | ((‘`-`’ | ‘`~`’), ⟨`term`⟩) | ⟨`subroutineCall`⟩;
⟨`subroutineCall`⟩ ::= identifier, [‘`.`’, identifier], ‘`(`’, ⟨`expressionList`⟩, ‘`)`’;
⟨`expressionList`⟩ ::= [⟨`expression`⟩, {‘`,`’, ⟨`expression`⟩}];

Which of the below is **not** a valid CST for a valid ⟨`expression`⟩?

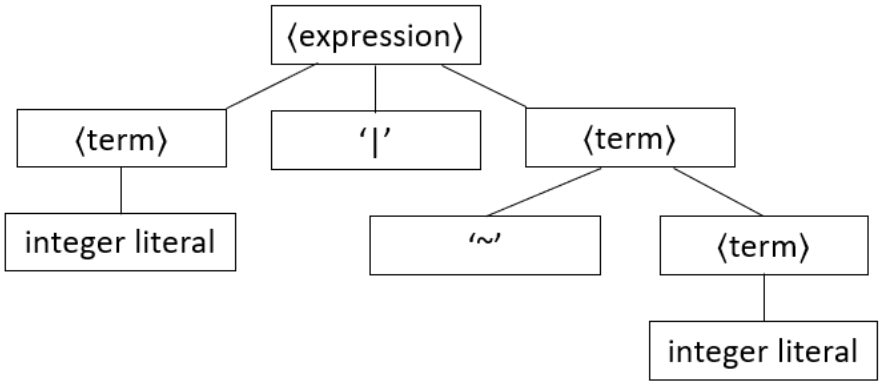
A.



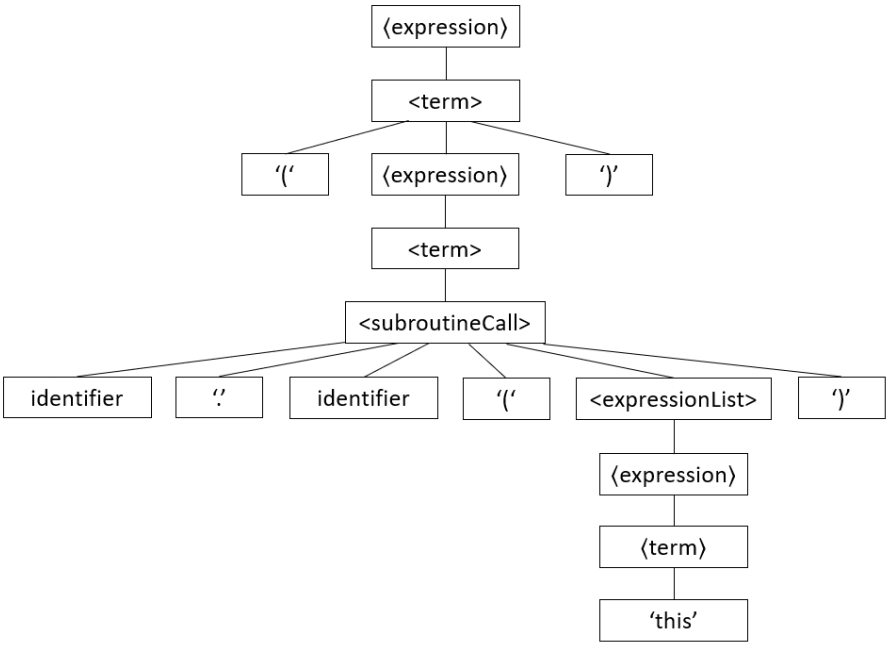
B.

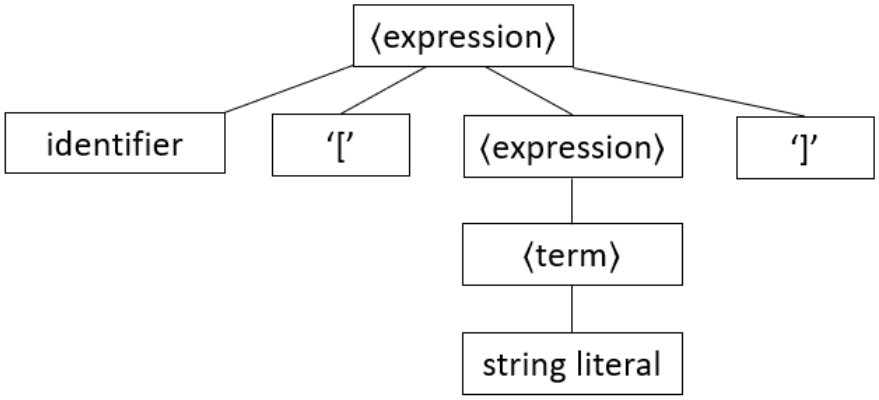

⟨expression⟩
<term>
<subroutineCall>
identifier
'('
<expressionList>
')'
⟨expression⟩
','
⟨expression⟩
⟨term⟩
'<'
⟨term⟩
⟨term⟩
string literal
'true'
'null'

C.


⟨expression⟩
⟨term⟩
'|'
⟨term⟩
integer literal
'~'
⟨term⟩
integer literal

D.


⟨expression⟩
<term>
'('
⟨expression⟩
')'
<term>
<subroutineCall>
identifier
'.'
identifier
'('
<expressionList>
')'
⟨expression⟩
⟨term⟩
'this'

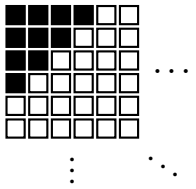


E.

Solution: E. From the rules of the grammar, it's valid to replace an ⟨`expression`⟩ with a ⟨`term`⟩ and then replace the ⟨`term`⟩ by identifier, '`[`', ⟨`expression`⟩, '`]`'. It's not valid to directly replace an ⟨`expression`⟩ by identifier, '`[`', ⟨`expression`⟩, '`]`', though, and that's what this CST represents.

Question 8 (5 marks)

In Hack, which values would need to be written to which RAM addresses in order to draw a right-angled triangle in the top-left corner of the screen as shown, with a height of four pixels? You should assume that initially, all memory is set to zero. You may give your answer in either binary, decimal, or hexadecimal form.



Solution: The values written should be:

$$\begin{aligned}\text{RAM}[0x4000] &= 0xF,\\ \text{RAM}[0x4020] &= 0x7,\\ \text{RAM}[0x4040] &= 0x3,\\ \text{RAM}[0x4060] &= 0x1.\end{aligned}$$

Question 9 (4 marks)

What is the technical term for each of the following concepts? For each part of the question, pick from one of the following options: "call frame", "global stack", "return address", "working stack", and "none of the above".

(a) The entire stack. **(1 mark)**

Solution: Global stack.

(b) A part of the stack dedicated to temporarily storing program state during a function call, to be restored on return. **(1 mark)**

Solution: Call frame.

(c) A part of the stack dedicated to storing values pushed directly via `push` Hack VM commands during the execution of a function. **(1 mark)**

Solution: Working stack.

(d) The value that will be left on top of the stack immediately after returning from a function. **(1 mark)**

Solution: None of the above. This is the return value, not the return address.

Question 10 (4 marks)

Which **one** of the following statements is **false**?

A. EBNF can't be "simulated" using BNF — there are grammars written in EBNF that can't be rewritten in BNF on such a way that they accept the same expressions as valid.

B. In the grammar for a programming language, which strings are considered to be tokens depends on which language the grammar is for.

C. When writing a compiler, symbol tables are typically used to store information about functions and variables.

D. Given the grammar for a programming language and a parser for that grammar, it's always possible to tell which programs have valid syntax.

E. An abstract syntax tree may have fewer nodes than the corresponding concrete syntax tree.

Solution: A. It was explained when EBNF was introduced that EBNF and BNF can express all the same grammars — EBNF just makes things easier.

Question 11 (5 marks)

Which RAM address (or addresses) does the Hack VM code below write the value 500 to?

```
push constant 200
pop this 0
push pointer 0
push pointer 1
add
pop pointer 1
push constant 500
pop that 7
```

You may assume the initial value of RAM[0] to RAM[4] are:

RAM[0] = 300;
RAM[1] = 290;
RAM[2] = 280;
RAM[3] = 1;
RAM[4] = 2;

Solution: 10 and 300. The first two lines write 200 to RAM[1]. The next four lines write 3 to RAM[4]. The last two lines write 500 to RAM[300] (the top of the stack) and RAM[10] (`that 7`).

Section 2 — Long-answer questions (50 marks total)

In this section, you should give the reasoning behind your answers unless otherwise specified. You **will** receive credit for partial answers or for incorrect answers with sensible reasoning.

Question 12 (7 marks)

(a) Explain what is meant by predication.

Solution: Predication is an optimisation technique where a conditional statement in high-level code is replaced by an arithmetic expression.

(b) Give an example of predication.

Solution: The C code `x = (y == 4) ? 3 : 5;` could be rewritten as `x = 3 + (2 * (y != 4));`.

(c) Why can predication make code run faster?

Solution: The original conditional expression requires at least one branch when compiled to assembly, leading to a control hazard. The new expression has no control hazards and can therefore be pipelined more efficiently.

Question 13 (15 marks)

Consider the Hack assembly code X which is a direct translation of the Hack VM instruction `return`. (So X assumes that the current contents of RAM are a state of the Hack VM under the standard memory map given on the reference sheet.)

(a) Explain in words how X "knows" which address to jump back to.

Solution: The assembly code for the corresponding `call` statement ended with a label, and the code put the ROM address of that label on the stack. The `return` code then retrieves that label from the stack and jumps to it.

(b) Give **detailed pseudocode** for what X must do. This should include which values are written to which addresses of memory, and which order the writes happen in.

Solution: There are a few different solutions, but here's one of them. (See also the non-C solutions of week 10.)

```
RAM[13] = RAM[LCL-5];
RAM[ARG] = RAM[SP-1];
SP = ARG+1;
THAT = RAM[LCL-1];
THIS = RAM[LCL-2];
ARG = RAM[LCL-3];
LCL = RAM[LCL-4];
Jump to the ROM address stored in RAM[13].
```

Statements like "Restore the old value of `THIS`" wouldn't contain enough detail for full marks, but would be worth partial credit.

Question 14 (5 marks)

Explain in words how the Hack assembler is able to calculate the ROM address of each label during the lexing process.

Solution: There are a few different ways of doing this — anything that works will get full marks, and there's no requirement to do the same thing as the model answer to the week 8 lab. During lexing, the assembler tracks the ROM address of the line currently being lexed in a variable, say `ROM_address`. Initially `ROM_address = 0`. Every line of assembly will take up exactly one word of ROM except for lines containing only whitespace, comments, or labels. So `ROM_address` is incremented on every line which, ignoring any initial whitespace, starts with anything other than a newline, a '`(`', or '`//`'. Then on reaching a label (as identified by the '`(`' character at the start), the correct ROM address will be stored in `ROM_address`.

Question 15 (5 marks)

Consider the following code in a fictional assembly language:

```
ADD #42, %35, $63
```

In this language, `ADD x, y, z` adds `x` to `y` and stores the value in `z`. The language uses `#` for immediate addressing, `$` for direct addressing, and `%` for indirect addressing. What will running this instruction do?

Solution: It will add 42 (immediate addressing) to RAM[RAM[35]] (indirect addressing) and store the result in RAM[63] (direct addressing).

Question 16 (10 marks)

Write Hack assembly code to solve the following problem. Fill the first RAM[0] memory addresses counting upwards from RAM[256] with the values $0, 1, 2, \ldots$, so that if e.g. RAM[0] = 3 then the desired output is

RAM[256] = 0, RAM[257] = 1, RAM[258] = 2, RAM[259] unchanged.

It's OK if your code changed the value stored in RAM[0].

Solution: There are lots of possible solutions here — here's one of them.

```
// pos = 256, value = 0
@256
D=A
@pos
M=D
@value
M=0
// While (RAM[0] > 0)
(loopstart)
@R0
D=M
@halt
D;JLE
// RAM[pos] = value
@value
D=M
@pos
A=M
```

```
M=D
// pos++, value++, RAM[0]--, and loop
@pos
M=M+1
@value
M=M+1
@R0
M=M-1
@loopstart
0;JMP
// Halt
(halt)
@halt
0;JMP
```

The comments will be useful for you when you check your work, but you don't need to include them to get full marks.

Question 17 (5 marks)

This question is about the first-fit memory allocation algorithm for Hack described in lectures that uses coalescence but not bins (i.e. "attempt 3"). We use -1 to denote free segments and null pointers, and 0 to denote used segments. Recall from lectures that in this algorithm, the structure of a segment's metadata is:

- The segment's first and last words contain its usable length.
- The segment's second word is -1 if the segment is free and 0 if it is used.
- If the segment is free, its third word is a pointer to the next free segment in the linked list and its fourth word is a pointer to the previous free segment in the linked list.

You are given the contents of the following memory addresses, which may or may not align with segment boundaries. Read them carefully.

RAM[0x800] = 0x2202;
RAM[0x801] = 0x9FC;
RAM[0x802] = 0xFFFF;
RAM[0x803] = 0xFFFF;
RAM[0x804] = 0x3003;
...
RAM[0x1200] = 0xDFD;
RAM[0x1201] = 0x0;
RAM[0x1202] = 0x2A02;
RAM[0x1203] = 0x2202;
...
RAM[0x21FE] = 0x844;
RAM[0x21FF] = 0x1FD;
RAM[0x2200] = 0x5FD;
RAM[0x2201] = 0xFFFF;
RAM[0x2202] = 0x3002;
...
RAM[0x29FF] = 0x1FD;

RAM[0x2A00] = 0x5FD;
RAM[0x2A01] = 0xFFFF;
...
RAM[0x2FFF] = 0x7FD;
RAM[0x3000] = 0xFFD;
RAM[0x3001] = 0xFFFF;
RAM[0x3002] = 0x803;

What segments are currently in memory? For each segment, you should give the first address, the last address, and whether it's free or used. If any segments can't be deduced from the information available, explain why.

Solution: Segment 1: 0x2200–0x27FF, free.

Segment 2: 0x3000–0x3FFF, free.

Segment 3: 0x801–0x11FF, free.

Segment 4: 0x1200–0x1FFF, used.

Segment 5: 0x2000–0x21FF, used

Segment 6: 0x2800–0x2FFF, used.

Segment 1 is pointed to by RAM[0x800], and its length is given by RAM[0x2200]. Segment 2 is pointed to by RAM[0x2202], and its length is given by RAM[0x3000]. Segment 3 is pointed to by RAM[0x3002], and its length is given by RAM[0x801]. Since RAM[0x803] = -1 that's the last free segment and all the others are used. We know 0x1200 is the start of a segment since segment 3 ends at address 0x11FF, and the length of segment 4 is given by RAM[0x1200]. We know the length of segment 5 from RAM[0x21FF], since segment 1 starts at 0x2200 so this must be the last address of some segment. Similarly, we can deduce segment 6's existence and length from RAM[0x2FFF] together with the fact that segment 2 starts at address 0x3000. These six segments cover all of heap memory, so there can't be any others.

Question 18 (3 marks)
Write **<u>at most six</u>** lines of Hack assembly that shift the contents of RAM[1] left by four bits.

Solution:

```
@R1
D=M
DM=M+D
DM=M+D
DM=M+D
DM=M+D
```

THIS PAGE IS INTENTIONALLY LEFT BLANK

END OF PAPER