CPSC 213, Winter 2016, Term 1

Midterm II Sample Questions

Exercise 1

Implement the following C code in assembly. Pass arguments on the stack. Assume that r5 has already been initialized as the stack pointer and assume that some other procedure (not shown) calls `doit()`.
You do not have to show the allocation of x; just use the label x to refer to its address. Comment every line.

```
int x;

void doit () {
      x = addOne (5);
}
int addOne (int a) {
      return a + 1;
}
```

Exercise 2

Implement the following in SM213 assembly. You can use a register for c instead of a local variable. Comment every line.

```
int len;
int* a;
int countNotZero () {
      int c=0;
      while (len>0) {
            len=len-1;
            if (a[len]!=0)
                  c=c+1;
            }
            return c;
}
```

Exercise 3

Given the following code:

```
struct S {
  int*      a;
  int       b[4];
  struct T  c;
  struct S* d;
};
```

4b Convert the assembly into C.

4c The code implements a simple function. What is it? Give the simplest, plain English description you can.

5 **(4 marks)** **Procedure Calls.** Given these global declarations

```
int x;
int (*proc) (int);
```

Give the SM213 assembly for the code below (just for this single statement, not for the procedure itself). Assume that the return value is in `r0`. Comments are not required.

```
x = proc (1);
```

8 **(8 marks)** **Procedures and the Stack.** Answer the following questions about this assembly code.

```
[01]        ld  $-12, r0
[02]        add r0, r5
[03]        ld  $2, r0
[04]        st  r0, 0(r5)
[05]        st  r0, 4(r5)
[06]        st  r0, 8(r5)
[07]        gpc $6, r6
[08]        j   foo
[09]        ld  $12, r1
[10]        add r1, r5
[11]        ld  $0x1000, r1
[12]        st  r0, (r1)
[13]        halt

[14]  foo:  ld  $-8, r0
[15]        add r0, r5
[16]        st  r6, 4(r5)
[17]        ld  8(r5), r0
[18]        ld  12(r5), r1
[19]        ld  16(r6), r2
[20]        add r1, r0
[21]        add r2, r0
[22]        ld  $8, r1
[23]        add r1, r5
[24]        j (r6)
```

8a How many arguments, if any, does `foo()` have?

8b How many local variables, if any, does `foo()` have (count them even if they are not used)?

8c Is `foo()`'s return address saved on the stack at any point. If so, which line saves it?

8d If you can determine the integer value in memory at address `0x1000` following the execution of this code, give its value.

7 **(7 marks)** **Procedure Calls** Implement the following C code in assembly. Pass arguments on the stack. Assume that `r5` has already been initialized as the stack pointer and assume that some other procedure (not shown) calls `doit()`. You do not have to show the allocation of `x`; just use the label `x` to refer to its address. Comment every line.

```
int x;

void doit () {
    x = addOne (5);
}

int addOne (int a) {
    return a + 1;
}
```

11 **(4 marks)** **Loops**. Consider the consequences of eliminating the two conditional branch instructions from SM213 (and not adding any other instructions). Would compilers still be able to compile C programs to run on this modified machine? If so, explain how. If not, carefully explain what feature or features of C would be impossible to execute on the modified machine.

12 **(4 marks)** **Procedure Call and Return**.

12a Is a procedure call a static or dynamic jump? Justify your answer.

12b Is a procedure return a static or dynamic jump? Justify your answer.

13 **(10 marks)** **Writing Assembly Code**. Write SM213 assembly code that implements the following C program. Use labels for static addresses but do not include variable label declarations (i.e. "`.long`" lines). Show only the code for these two procedures. Do not implement a return from `callReplace()`; simply halt at the end of that procedure. Do not use the stack. Comment every line.

```
int* a;
int  searchFor, replaceWith, size, i;

void replace() {                        void callReplace() {
    for (i=0; i<size; i++)                  replace();
        if (a[i]==searchFor)                // halt; do not return
            a[i]=replaceWith;           }
}
```

14 **(20 marks)** The following SM213 assembly code implements a simple procedure. Carefully comment every line, give an equivalent C program that would compile into this assembly, and explain in plan English what this procedure does.

4 **(8 marks)** **Procedure Calls.** Give SM213 assembly for these statements. Assume the `i` is a global variable of type `int`, that `r5` stores the value of the stack pointer, and that arguments are passed on the stack.

4a
```
int foo (int i, int j) {
    return j;
}
```

4b
```
    i = foo (1, 2);
```

```
int* a;
int  searchFor, replaceWith, size, i;

void replace() {                        void callReplace() {
    for (i=0; i<size; i++)                  replace();
        if (a[i]==searchFor)                // halt; do not return
            a[i]=replaceWith;           }
}
```

7 **(10 marks)** Implement the following in SM213 assembly. Pass arguments on the stack. You can use a register for `c` instead of a local variable. **Comment every line.**

```
int countNotZero (int len, int* a) {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

4 **(10 marks)** **Write Assembly Code.** Give the assembly code the compiler would generate to implement the following C procedure, assuming that arguments are passed on the stack. Just this procedure. Use labels for static values. Comment every line of your code.

```
int computeSum (int* a) {
    int sum=0;
    while (*a>0) {
        sum = add (sum, *a);
        a++;
    }
    return sum;
}
```

6 **(12 marks)** **Procedures and Writing Assembly Code.** Consider the following procedure in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers.

```
int sum (int *a, int i, int* b, int j) {
    return a[i] + b[j];
}
```

6a Why would we use the stack to store arguments instead of just using registers? Explain carefully.

6b What is known statically vs dynamically?

- The offset between the stack pointer for sum and the address of a?
- The offset between the stack pointer for sum and the address of i?
- The address of a?
- The address of i?

6c Implement the C procedure above in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers. Comment your code.

CPSC 213, Winter 2016, Term 1

Midterm II Sample Questions - Solutions

Exercise 1

```
doit:
      deca r5           # allocate space for ra on stack
      st r6, (r5)       # save ra on stack
      deca r5           # make room for argument on stack
      ld $5, r0         # r0 = 5
      st r0, (r5)       # arg0 = 5
      gpc $6, r6        # get return address
      j add             # call addOne (5)
      inca r5           # remove argument area
      ld $x, r1         # r1 = &x
      st r0, (r1)       # x = addOne (5)
      ld (r5), r6       # restore ra from stack
      inca r5           # remove ra space from stack
      j (r6)            # return
addOne:
      ld (r5), r0       # r0 = a
      inc r0            # r0 = a + b
      j (r6)            # return a + b
```

Exercise 2

```
      countZero:  ld $len, r1             # r1 = address of len
                  ld (r1), r1             # r1 = len
                  ld $a, r2               # r2 = address of a
                  ld (r2), r2             # r2 = a
                  ld $0, r0               # r0 = c

      loop:       bgt r1, cont            # goto cont if len>0
                  br done                 # goto done if len<=0

      cont:       dec r1                  # len = len - 1
                  ld (r2, r1, 4), r3      # r3 = a[len]
                  beq r3, loop            # goto loop if a[len]==0
                  inc r0                  # c=c+1 if a[len]!=0
                  br loop                 # goto loop
      done:       j (r6)                  # return c
```

Exercise 3

```
      ld $0, r0
      ld $s, r1
      ld (r1), r2
      st r0, 24(r2)
```

```
    int x;
    int (*proc) (int);
```

Give the SM213 assembly for the code below (just for this single statement, not for the procedure itself). Assume that the return value is in `r0`. Comments are not required.

```
    x = proc (1);
```

```
ld   $1, r0
deca r5
st   r0, (r5)
ld   $proc, r0
gpc  $2, r6
j    *(r0)
inca r5
ld   $x, r1
st   r0, (r1)
```

Rubric: 1 for arg, 2 for call (1 for indirect), 1 for store result. One minor error to gpc value or offsets okay; two is -1.

6 **(9 marks)** **C Pointers and Functions** Consider the following C code.

```
void foo(int x, int *p) {
A.     printf("x = %d, *p = %dn", x, *p);
       *p = x + 12;
       --x;
       p = 0;
}
int main (void) {
       int   a = 1;
       int   b = 2;
       int  *p = &a;
       int **q = &p;
B.     printf("**q = %dn", **q);
       foo(*p, *q);
C.     printf("a = %dn", a);
       if (!p)
D.           printf("p is 0n");
       else
E.           printf("*p is %dn", *p);
       *q = &b;
F.     printf("*p = %dn", *p);
       return 0;
}
```

In the above code, what if anything is printed by the `printf` instruction at locations A to F when the program is executed?

6a A

(2) x = 1, *p = 1

6b B

(1) **q = 1

6c C

(2) a = 13

6d D

(1) nothing

6e E

(1) *p is 13

6f F

```
int* copy (int* src) {
  int* dst = malloc (sizeof (int));
  inc_ref (dst);
  *dst = *src;
  return dst;
}

int foo() {
  int  a = 3;
  int* b = copy (&a);
  saveIfMax (b);
  int t = *b;
  dec_ref (b)
  return t;
  ~~return *b;~~

}
```

```
int* max;
void saveIfMax (int* x) {
  if (max==NULL || *x > *max) {
    if (max != NULL)
      dec_ref(max);
    max = x;
    inc_ref(max);
  }
}
```

8 **(8 marks)** **Procedures and the Stack.** Answer the following questions about this assembly code.

```
[01]        ld   $-12, r0
[02]        add  r0, r5
[03]        ld   $2, r0
[04]        st   r0, 0(r5)
[05]        st   r0, 4(r5)
[06]        st   r0, 8(r5)
[07]        gpc  $6, r6
[08]        j    foo
[09]        ld   $12, r1
[10]        add  r1, r5
[11]        ld   $0x1000, r1
[12]        st   r0, (r1)
[13]        halt

[14]  foo:  ld   $-8, r0
[15]        add  r0, r5
[16]        st   r6, 4(r5)
[17]        ld   8(r5), r0
[18]        ld   12(r5), r1
[19]        ld   16(r6), r2
[20]        add  r1, r0
[21]        add  r2, r0
[22]        ld   $8, r1
[23]        add  r1, r5
[24]        j  (r6)
```

8a How many arguments, if any, does `foo()` have?

3

8b How many local variables, if any, does `foo()` have (count them even if they are not used)?

1

8c Is `foo()`'s return address saved on the stack at any point. If so, which line saves it?

Yes; line 16.

8d If you can determine the integer value in memory at address `0x1000` following the execution of this code, give its value.

6

```
int x;

void doit () {
    x = addOne (5);
}

int addOne (int a) {
    return a + 1;
}
```

```
doit:
    deca r5          # allocate space for ra on stack
    st r6, (r5)      # save ra on stack
    deca r5          # make room for argument on stack
    ld $5, r0        # r0 = 5
    st r0, (r5)      # arg0 = 5
    gpc $6, r6       # get return address
    j add            # call addOne (5)
    inca r5          # remove argument area
    ld $x, r1        # r1 = &x
    st r0, (r1)      # x = addOne (5)
    ld (r5), r6      # restore ra from stack
    inca r5          # remove ra space from stack
    j (r6)           # return
addOne:
    ld (r5), r0      # r0 = a
    inc r0           # r0 = a + b
    j (r6)           # return a + b
```

8 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* b;

void set (int i) {
    b [i] = i;
}
```

There is a dangerous bug in this code. Carefully describe what it is. Assume that `b` was assigned a value somewhere else in the program.

There's a potential array overflow. Need to check that `i` is in range (0 .. size of `b` - 1) before writing to `b[i]` and thus this size, which is dynamically determined, should be a parameter to `set` or a global variable.

9 **(3 marks)** **Programming in C.** Consider the following C code.

```
int* one () {
    int loc = 1;
    return &loc;
}

void two () {
    int zot = 2;
}
```

```
void three () {
    int* ret = one();
    two();
}
```

There is a dangerous bug in this code. Carefully describe what it is.

Hint: what is the value of `*ret` just before and just after `two()` is called? Look carefully at the implementation of `one()`, what it returns, and when variables are allocated and deallocated.

Yes; there's a dangling pointer. The procedure `one()` returns a pointer to a local variable, but that local variable is deallocated when the procedure returns. Just before `three()` calls `two()` the value of `*ret` is 1, but after calling `two()` it changes to 2 because `two()`'s local variable `zot` will be allocated in the same location as `one()`'s `loc`, and `*ret` is a dangling pointer pointing to that location.

10 (4 marks) **Branch and Jump Instructions**.

10a What is one important benefit that *PC-relative* branches have over *absolute-address* jumps.

smaller instructions

10b What is the value of the program-counter register (i.e., the PC) following the execution of this unconditional branch instruction at address `0x500`. Justify your answer.

```
0x500: 8005
```

```
0x502 + 5 * 2 == 0x50c
```

11 (4 marks) **Loops**.

Consider the consequences of eliminating the two conditional branch instructions from SM213 (and not adding any other instructions). Would compilers still be able to compile C programs to run on this modified machine? If so, explain how. If not, carefully explain what feature or features of C would be impossible to execute on the modified machine.

If-then statements whose test condition is a dynamic value and loops that execute a bounded and dynamically determined number of times.

12 (4 marks) **Procedure Call and Return**.

12a Is a procedure call a static or dynamic jump? Justify your answer.

Static. The compiler knows the address of every procedure.

12b Is a procedure return a static or dynamic jump? Justify your answer.

Dynamic. A procedure can be called from multiple statements and each of these will have different return addresses. The same return statement must thus be able to jump to many different addresses, depending on which statement called it.

13 (10 marks) **Writing Assembly Code**.

Write SM213 assembly code that implements the following C program. Use labels for static addresses but do not include variable label declarations (i.e. "`.long`" lines). Show only the code for these two procedures. Do not implement a return from `callReplace()`; simply halt at the end of that procedure. Do not use the stack. Comment every line.

```
int* a;
int  searchFor, replaceWith, size, i;

void replace() {
    for (i=0; i<size; i++)
        if (a[i]==searchFor)
            a[i]=replaceWith;
}
```

```
void callReplace() {
    replace();
    // halt; do not return
}
```

```
         ld $i, r0          # r0 = &i
         ld (r0), r1        # r1 = i
         ld $-4, r2         # r2 = -4
         add r2, r1         # r1 = i-4
         bgt r1, L0         # goto L0 if i > 4
         beq r1, L0         # goto L0 if i == 4
         br DEFAULT         # goto L0 if i < 4
L0:      ld $-2, r2         # r2 = -2
         add r1, r2         # r2 = i-6
         bgt r2, DEFAULT    # goto DEFAULT if i > 6
         ld $JT, r2         # r2 = JT
         j *(r2, r1, 4)     # goto jt[i-4]
CASE_4:  ld $0, r2          # t_i = 0
         br L1              # goto L1
CASE_6:  ld $1, r2          # t_i = 1
         br L1              # goto L1
DEFAULT: ld $2, r2          # t_i = 2
L1:      st r2, (r0)        # i = t_i

# The Jump Table
JT:      .long CASE_4
         .long DEFAULT
         .long CASE_6
```

3b Where the global variable `int (*bar)(void)` was previously declared, the statement:

```
    bar();
```

```
ld $bar, r0   # r0 = &bar
gpc $2, r6    # r6 = return address
j *(r0)       # bar()
```

4 **(8 marks)** **Procedure Calls.** Give SM213 assembly for these statements. Assume the `i` is a global variable of type `int`, that `r5` stores the value of the stack pointer, and that arguments are passed on the stack.

4a
```
int foo (int i, int j) {
    return j;
}
```

```
ld 4(r6), r0   # r0 = j
j (r6)         # return j
```

4b
```
    i = foo (1, 2);
```

```
deca r5        # make stack space for arg0
deca r5        # make stack space for arg1
ld $1, r0      # r0 = 1
st r0, 0(r5)   # arg0 = 1
ld $2, r0      # r0 = 2
st r0, 4(r5)   # arg1 = 2
gpc $6, r6     # r6 = return address
j foo          # t_i = foo (1,2)
inca r5        # free stack space for arg1
inca r5        # free stack space for arg0
ld $i, r1      # r1 = &i
st r0, (r1)    # r1 = t_i
```

5 **(12 marks)** Consider the following SM213 assembly code that implements a simple C procedure.

```
replace:      ld   $size, r0          # r0 = &size
              ld   0x0(r0), r0        # r0 = size = i
              ld   $a, r1             # r1 = &a
              ld   0x0(r1), r1        # r1 = a
              ld   $searchFor, r2     # r2 = &searchFor
              ld   0x0(r2), r2        # r2 = searchFor
              not  r2                 # r2 = !searchFor
              inc  r2                 # r2 = -searchFor
              ld   $replaceWith, r3   # r3 = &replaceWith
              ld   0x0(r3), r3        # r3 = replaceWith
loop:         beq  r0, done           # goto done if i==0
              dec  r0                 # i--
              ld   (r1, r0, 4), r4    # r4 = a[i]
              add  r2, r4             # r4 = a[i] - searchFor
              beq  r4, match          # goto match   if a[i]==searchFor
              br   nomatch            # goto nomatch if a[i]!=searchFor
match:        st   r3, (r1, r0, 4)    # a[i] = replaceWith
nomatch:      br   loop               # goto loop
done:         j    0x0(r6)            # return
callReplace:  gpc  $0x6, r6           # ra = pc + 6
              j    replace            # replace()
              halt
```

7 **(10 marks)** Implement the following in SM213 assembly. Pass arguments on the stack. You can use a register for c instead of a local variable. **Comment every line.**

```
int countNotZero (int len, int* a) {
    int c=0;
    while (len>0) {
        len=len-1;
        if (a[len]!=0)
            c=c+1;
    }
    return c;
}
```

```
countZero: ld  0(r5), r1         # r1 = len
           ld  4(r5), r2         # r2 = a
           ld  $0, r0            # r0 = c
loop:      bgt r0, cont          # goto cont if len>0
           dec r1                # len = len - 1
           br  done              # goto done if len<=0
cont:      ld  (r2, r1, 4), r3   # r3 = a[len]
           beq r3, loop          # goto skip if a[len]==0
           inc r0                # c=c+1 if a[len]!=0
           br  loop              # goto loop
done:      j   (r6)              # return c
```

4 **(10 marks)** **Write Assembly Code.** Give the assembly code the compiler would generate to implement the following C procedure, assuming that arguments are passed on the stack. Just this procedure. Use labels for static values. Comment every line of your code.

```
int computeSum (int* a) {
    int sum=0;
    while (*a>0) {
        sum = add (sum, *a);
        a++;
    }
    return sum;
}
```

```
.pos 0x100
computeSum: deca r5          # allocate space on stack for return address
            st r6, 0(r5)     # store return address on stack
            ld $0, r0        # r0 = 0 (sum)
            ld 4(r5), r1     # r1 = a (argument passed on stack)
loop_guard: ld 0(r1), r2     # r2 = *a
            bgt r2 loop_body # goto loop_body if *a > 0
            br loop_end      # executed if the above branch condition fails
loop_body:  deca r5          # allocate arg 2
            deca r5          # allocate arg 1
            st r0, 0(r5)     # arg 1 = sum
            st r2, 4(r5)     # arg 2 = *a
            gpc $6, r6       # r6 = return address
            j add            # call add(sum, *a)
                             # return value is in r0 so sum already = add(...)
            inca r5          # deallocate arg 2
            inca r5          # deallocate arg 1
            inca r2          # a++
            br loop_guard
loop_end:   ld 0(r5), r6     # r6 = return address
            inca r5          # deallocate return address
            j 0(r6)          # return sum (sum is already in r0)
```

You don't have to copy the way I've commented the code verbatim, I'm slightly more descriptive than necessary for teaching purposes. But don't forget to comment your code, and it helps to be explicit about which registers represent which variables for your own sake, and the grader's.

We allocate the return address on the stack at the beginning because `computeSum` is not a leaf procedure (it calls another function `add(...)`).

The way I've written the loop is just one of the ways you could have done it. I prefer translating loops/if statements as closely as possible to the code's structure (unless explicitly asked to do otherwise) because I find it easier to think about the code and convince myself about its correctness. So I structure my conditional branch to represent the *true* case of the loop guard. Again, this is personal preference and you wouldn't lose marks for structuring your loop differently.

There are some subtleties here with how I chose r0 to be the variable `sum`. I could have stored it on the stack, but it's easier to store it in registers when writing assembly under exam conditions. This should be acceptable as long as the question doesn't explicitly ask you to store local variables on the stack. Since `add(...)` stores the return value in r0, when the function returns, it's automatically assigned to be `sum`'s value! Note that if the statement was instead something like `sum = sum + add(...)` then I couldn't do this, because I would have to save `sum` to another register before it gets overwritten so I could add it to the result. Having `sum` in r0 also means that I don't have to do any extra work of preparing r0 before returning at the end of `computeSum`.

Finally, don't forget that `a` is a pointer, so `a++` does pointer arithmetic instead of just +1.

5b Give SM213 assembly code that reads the value of `b->k` into `r0`. Comment your code.

```
ld   $b, r0              # r0 = &b (pointer to struct)
ld   0x0(r0), r0         # r0 = b (address of struct itself)
ld   0x1c(r0), r0        # r0 = b->k (content of address of field in struct)
```

3 marks. Full credit also given for alternate answer using indexed load. Subtle point: while the offset has to fit into one hex digit of machine language, what's actually stored is in the machine language instruction is the number from the assembly language instruction divided by 4. Hex 1c is decimal 28. $28/4 = 7$, and 0x7 will indeed fit into that digit. But full credit also given for slightly more verbose solution:

```
ld  $b, r0       # r0 = &b (pointer to struct)
ld  0x0(r0), r0  # r0 = b (address of struct itself)
ld $0x1c, r1     # r1 = decimal 28 (7 integers * 4 bytes each)
add r1, r0       # r0 = address of struct field b->k
ld 0x0(r0), r0   # r0 = value of b->k field
```

6 **(12 marks)** **Procedures and Writing Assembly Code.** Consider the following procedure in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers.

```
int sum (int *a, int i, int* b, int j) {
    return a[i] + b[j];
}
```

6a Why would we use the stack to store arguments instead of just using registers? Explain carefully.

There are a limited number of registers, so procedures with many arguments would not be able to store all arguments on the stack. Also, the convention in SM213 is to use specific registers to store information like the return address and the stack pointer, so the number of registers available for free use inside a procedure is even more limited. Using the stack in the first place avoids the need to save registers to the stack in subsequent procedure calls.

2 marks. The answer "the stack is better than registers because it's too hard to keep track of which register is used for which parameter" is not correct: the compiler can indeed figure that information out at compile time. The answer "the stack is more efficient than using registers" is not correct: the stack is slower since it requires memory accesses. Most answers involving dangling pointers or polymorphism were incorrect.

6b What is known statically vs dynamically?

- The offset between the stack pointer for sum and the address of a? static
- The offset between the stack pointer for sum and the address of i? static
- The address of a? dynamic
- The address of i? dynamic

6c Implement the C procedure above in SM213 assembly code. Assume that arguments have been passed in on the stack, not in registers. Comment your code.

```
sum:     ld   0(r5), r1       # r1 = a
         ld   4(r5), r2       # r2 = i
         ld   8(r5), r3       # r3 = b
         ld   12(r5), r4      # r4 = j
         ld   (r1,r2,4), r0   # r0 = a[i]
         ld   (r3,r4,4), r1   # r1 = b[j]
         add  r1, r0          # r0 = a[i] + b[j]
         j    (r6)            # return
```

8 marks.

- +1 for using parameters on the stack that were placed by the caller, as opposed to pushing parameters explicitly in this code or using registers.
- +1 for having the correct offsets for stack arguments
- +1 for leaving the teardown to the caller (not doing it explicitly here)
- +1 for correct calculation for a/b
- +1 for correct calculation for i/j
- +1 for add
- +1 for having return value in r0
- +1 for jump to r6 value
- -1/2 for moving result to r0 instead of computing it there in the first place (verbose)
- -1/2 for wrong order of arguments
- -1/2 for changing the stack pointer instead of using offsets from it
- -1/2 for jump to r6 instead of (r6)

7 **(4 marks)** **Static Control Flow.** Consider the following procedure in C.

```
if (a > 2) b = 3;
else if (a < -4) b = 5;
```

In SM213 assembly, how many branch/jump statements are needed to implement this code? How many of these are conditional and how many are unconditional? Justify your answer.

Three total, two conditional and one unconditional. One conditional is needed for each of the two tests in the C code. One unconditional is needed to skip past the second code block. A second unconditional is not needed at the end of that block, because control will simply flow through to the next line.

4 marks: 1 for correct answer of 3 total, 1 for its justification. 1 for correct answer of 2 cond + 1 uncond, 1 for its justification.