EE234

Microprocessor Systems

Final Exam

Dec. 15, 2020. (11am – 1:30pm)

Instructor: Dae Hyun Kim (daehyun@eecs.wsu.edu)

Name:

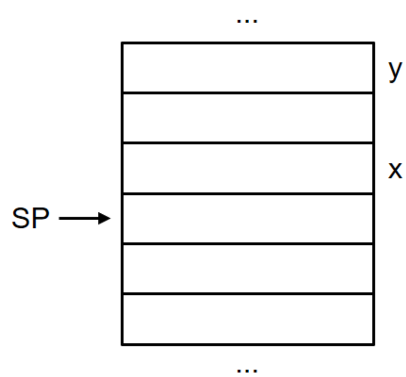
WSU ID:

Problem	Points	
1	20	
2	20	
3	30	
4	20	
5-1	20	
5-2	30	
6	20	
Total	160	

Problem 1 (1-D Array, 20 points)

All the registers R# are 32-bit registers. “int” is a 32-bit signed integer data type. Write an assembly code for the “for” loop in the following C code. The memory figure shows the stack pointer (SP) and the locations of the variables x and y.

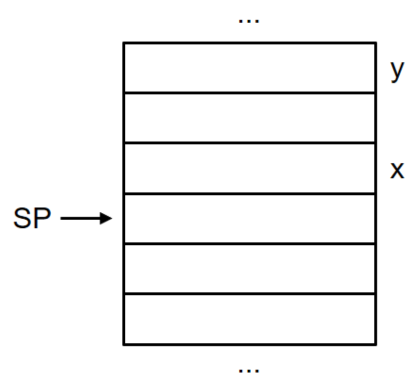
```
int y[10];
int* x = new int[10];

...

for ( int k = 0 ; k < 10 ; k++ )
  x[k] = y[k];
```



Problem 2 (1-D Array, 20 points)

All the registers R# are 32-bit registers. “long” is a 64-bit signed integer data type. Write an assembly code for the “for” loop in the following C code. The memory figure shows the stack pointer (SP) and the locations of the variables x and y.

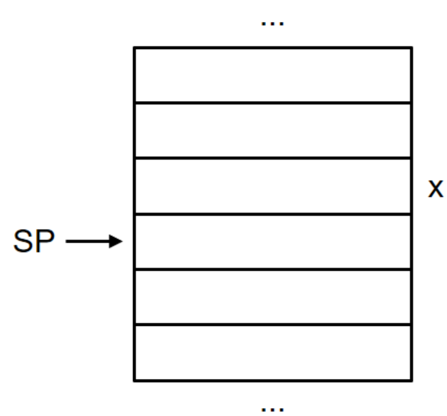
```
long y[10];
long* x = new long[10];

...

for ( int k = 0 ; k < 10 ; k++ )
  x[k] = y[k];
```



Problem 3 (2-D Array, 30 points)

All the registers R# are 32-bit registers. “int” is a 32-bit signed integer data type. Write an assembly code for the nested “for” loop in the following C code. The memory figure shows the stack pointer (SP) and the locations of the variables x and y.

```
int** x = new int*[2];

for ( int k = 0 ; k < 2 ; k++ )
  x[k] = new int[4];

...

for ( int i = 0 ; i < 2 ; i++ ) {
  for ( int k = 0 ; k < 4 ; k++ ) {
    x[i][k] = 0;
  }
}
```



Note: Try to minimize the number of memory access instructions (LDR, STR) executed. However, you should strictly follow the flow of the program (i.e., you should have two nested loops, etc.).

Problem 4 (Estimation of Memory Consumption, 20 points)

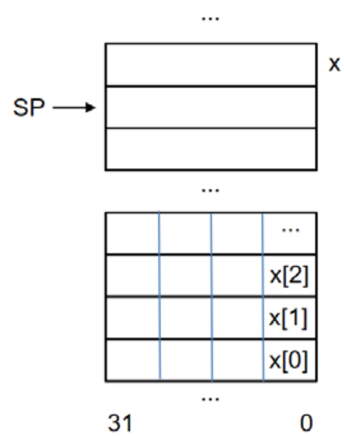
Estimate how many bytes are used for the array x in the following C code. You should include the memory space used for variable x itself. “int” is a 32-bit signed integer data type.

```
int**** x = new int***[3];

for ( int k = 0 ; k < 3 ; k++ )
  x[k] = new int**[4];

for ( int i = 0 ; i < 3 ; i++ ) {
  for ( int k = 0 ; k < 4 ; k++ ) {
    if ( k % 2 == 0 ) {
      x[i][k] = new int*[5];
      for ( int m = 0 ; m < 3 ; m++ )
        x[i][k][m] = new int[6];
    }
  }
}
```

Problem 5-1 (Array Manipulation I, 20 points)

The “char” data type in C is used to represent 1 byte. If you need an array of M char-type variables, you will ideally need M bytes. However, all the memory addresses for LDR and STR instructions should be integer multiples of 4 in the 32-bit ARM architecture (so, for example, you cannot use 0x0001 for a target memory address). Now, let’s take a look at the following C code. It reserves memory space for 80 characters, so ideally it should reserve 80 Bytes in the heap memory. However, it requires some bit manipulations. Thus, a compiler can reserve 320 Bytes in the heap memory and use only the lease significant 1B in each word for each x[k] as follows.

```
char* x = new char[80];

for ( int k = 0 ; k < 80 ; k++ )
  x[k] = k;
```

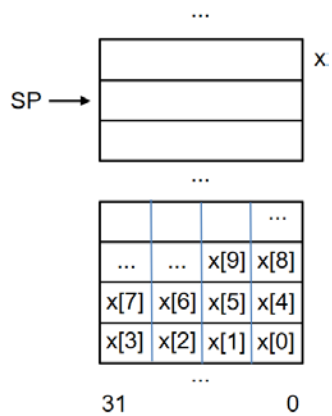


Write an assembly code for the “for” loop in the C code shown above. The memory management should be the same as the compiler above.

Problem 5-2 (Array Manipulation II, 30 points)

For the C code in Problem 5-1, a different compiler reserves exactly 80 Bytes in the heap space as follows.

```
char* x = new char[80];

for ( int k = 0 ; k < 80 ; k++ )
  x[k] = k;
```



Write an assembly code for the “for” loop in the C code shown above. The memory management should be the same as the new compiler explained above.

Note: If you know what to do, but don’t know how to implement it, you can just explain (in English) what you should do to implement the above C code (to get some partial credit).

Problem 6 (C, 20 points)

All the registers R# are 32-bit registers. <u>“int” is a 32-bit signed integer data type and “long” is a 64-bit signed integer data type.</u> The following table shows the main memory.

```
int** x = new int*[a];

for ( int i = 0 ; i < a ; i++ )
  x[i] = new int[b];
```

“a” and “b” are some constants. Currently, the value of x is 0x4000 as shown in the figure.

Address	Data (31 … 0)	
0x8000	0x4000	x
...		
0x402C	0x4000	
0x4028	0x402C	
0x4024	0x4028	
0x4020	0x4024	
0x401C	0x4020	
0x4018	0x401C	
0x4014	0x4018	
0x4010	0x4014	
0x400C	0x4010	
0x4008	0x4024	
0x4004	0x4014	
0x4000	0x400C	

(a) What is the value of *((int*) x)?

(b) What is the value of *((long*) x)?

(c) What is the value of x[2]?

(d) What is the value of x + 3?

(e) What is the value of (x[0]+2)?

(f) What is the value of x[1][2]?

(g) int* y = x[1]. What is the value of y[2]?

(h) long** y = (long**) x. What is the value of y[1]?

(i) What is the value of &(x[2])?

(j) long* y = (long*) x[0]. What is the value of y[1]?

Assembly Instructions

R# is a register. (# = 0 ~ 12)

<table>
<tr><th>Instruction</th><th>Meaning</th></tr>
<tr><td>INV Rd</td><td>Bitwise inversion.
<table>
<tr><td>Before</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>After</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
</table></td></tr>
<tr><td>AND Rd, Ra, Rb
AND Rd, Ra, #imm
AND Rd, #imm</td><td>Bitwise AND. (Rd = Ra AND Rb), (Rd = Ra AND #imm), (Rd = Rd AND #imm)
<table>
<tr><td>Ra</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td>Rb</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Rd</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table></td></tr>
<tr><td>OR Rd, Ra, Rb
OR Rd, Ra, #imm
OR Rd, #imm</td><td>Bitwise OR. (Rd = Ra OR Rb), (Rd = Ra OR #imm), (Rd = Rd OR #imm).
<table>
<tr><td>Ra</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>Rb</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Rd</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
</table></td></tr>
<tr><td>EOR Rd, Ra, Rb
EOR Rd, Ra, #imm
EOR Rd, #imm</td><td>Bitwise exclusive-OR. (Rd = Ra ⊕ Rb), (Rd = Ra ⊕ #imm), (Rd = Rd ⊕ #imm)
<table>
<tr><td>Ra</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>Rb</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>Rd</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr>
</table></td></tr>
<tr><td>LSR Rd, Ra, #imm
LSR Rd, #imm</td><td>Logical shift right by (#imm) bits. (Rd = Rd >> #imm), (Rd = Rd >> #imm)
Ex) #imm = 3
<table>
<tr><td>Before</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>After</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
</table></td></tr>
<tr><td>LSL Rd, Ra, #imm
LSL Rd, #imm</td><td>Logical shift left by (#imm) bits. (Rd = Ra << #imm), (Rd = Rd << #imm)
Ex) #imm = 3
<table>
<tr><td>Before</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td></tr>
<tr><td>After</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
</table></td></tr>
<tr><td>MOV Rd, Ra
MOV Rd, #imm</td><td>Rd = Ra
Rd = #imm</td></tr>
<tr><td>ADD Rd, Ra, Rb
ADD Rd, Ra, #imm
ADD Rd, #imm</td><td>Rd = Ra + Rb
Rd = Ra + #imm
Rd = Rd + #imm</td></tr>
<tr><td>SUB Rd, Ra, Rb
SUB Rd, Ra, #imm
SUB Rd, #imm</td><td>Rd = Ra - Rb
Rd = Ra - #imm
Rd = Rd - #imm</td></tr>
<tr><td>MUL Rd, Ra, Rb
MUL Rd, Ra, #imm</td><td>Rd = Ra * Rb
Rd = Ra * (#imm)</td></tr>
<tr><td>CMP Rd, #imm
CMP Rd, Ra</td><td>Set Z = 1 if Rd == #imm. Otherwise, Z = 0. (Z is the Zero field of the CPSR.)
Set Z = 1 if Rd == Ra. Otherwise, Z = 0.
Notice that N != V is Rd < #imm or Rd < Ra.</td></tr>
<tr><td>BEQ [addr]</td><td>Branch to [addr] if Z = 1. Ex) CMP R1, R2. BEQ tar → Go to tar if R1 == R2.</td></tr>
<tr><td>BNE [addr]</td><td>Branch to [addr] if Z = 0. Ex) CMP R1, R2. BNE tar → Go to tar if R1 != R2.</td></tr>
<tr><td>BLT [addr]</td><td>Branch to [addr] if N != V. Ex) CMP R1, R2. BLT tar → Go to tar if R1 < R2.</td></tr>
<tr><td>LDR Rd, [Ra, #imm]</td><td>Load the data stored at [Ra + #imm] to Rd.</td></tr>
<tr><td>STR Rd, [Ra, #imm]</td><td>Store the data stored in Rd to [Ra + #imm].</td></tr>
</table>