Week 22/24 Assessed Coursework

C/DEBUGGING

Deadline Friday Week 24 (Friday, 19 May 2023, 16:00)

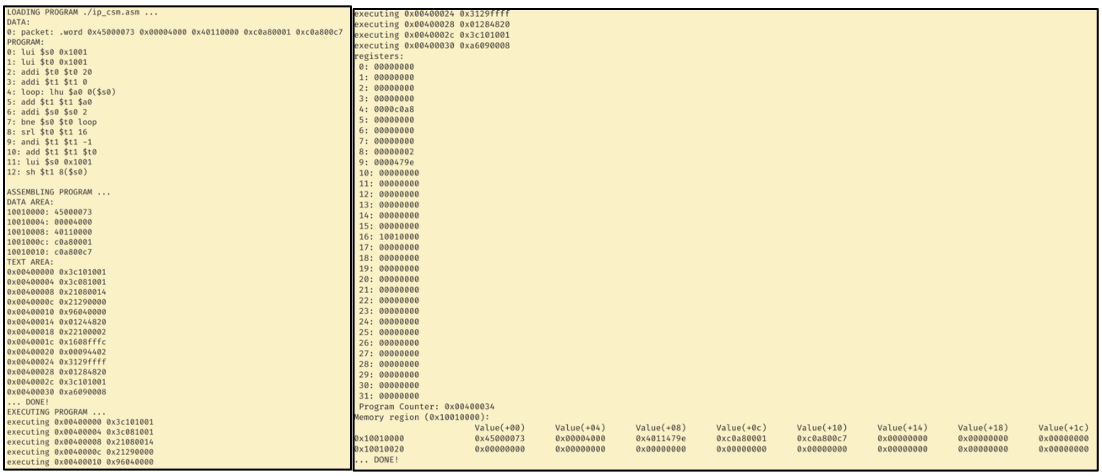
In week 20, you were introduced on the task of implementing a C function which translates MIPS assembly code into bytecode. You task for the SCC.150 week-24 coursework is to implement the *exec_bytecode* function, and implement a MIPS CPU emulator. The function should read the bytecode generated by the *make_bytecode* function and stored in the integer array `text`. Your program should unmarshal 32-bit instructions and modify appropriately the state of the registers contained in the *registers* integer array, the *pc* global variable and the global *data* array. Your implementation should support the following instructions:

NOP - no operation, ADD - addition, LUI – load upper immediate, ADDI – addition immediate,
ANDI - bitwise immediate and, SRL - shift right logical, SLL - shift left logical,
BNE - branch on not equal, LW - Load word, SW - Store word,
LHU - Load half-word unsigned, SH - Store half-word

Your implementation should manipulate the pc register value and emulate addresses in the text memory area (0x00400000 – 0x1001000, start from address 0x00400000) and should provide a way to terminate the execution of a program (e.g. nop instruction). Furthermore, you should emulate a simple global memory region by emulating memory area starting at address 0x1001000 using the array *data* and store information in big-endianness.

Important notes:

- Marks will be awarded only for the exec_bytecode function implementation and any new functions that you will define/implement to improve code readability.
- You should use the update template, which can be found at https://modules.lancaster.ac.uk/pluginfile.php/3150453/mod_label/intro/scc150_template_week24.zip?time=1682314742407 to implement your solution, which contains a correct implementation of the make_bytecode code.



```
LOADING PROGRAM ./ip_csm.asm ...
DATA:
0: packet: .word 0x45000073 0x00004000 0x40110000 0xc0a80001 0xc0a800c7
PROGRAM:
0: lui $s0 0x1001
1: lui $t0 0x1001
2: addi $t0 $t0 20
3: addi $t1 $t1 0
4: loop: lhu $a0 0($s0)
5: add $t1 $t1 $a0
6: addi $s0 $s0 2
7: bne $s0 $t0 loop
8: srl $t0 $t1 16
9: andi $t1 $t1 -1
10: add $t1 $t1 $t0
11: lui $s0 0x1001
12: sh $t1 8($s0)

ASSEMBLING PROGRAM ...
DATA AREA:
10010000: 45000073
10010004: 00004000
10010008: 40110000
1001000c: c0a80001
10010010: c0a800c7
TEXT AREA:
0x00400000 0x3c101001
0x00400004 0x3c081001
0x00400008 0x21080014
0x0040000c 0x21290000
0x00400010 0x96040000
0x00400014 0x01244820
0x00400018 0x22100002
0x0040001c 0x1608fffc
0x00400020 0x00094402
0x00400024 0x3129ffff
0x00400028 0x01284820
0x0040002c 0x3c101001
0x00400030 0xa6090008
... DONE!
EXECUTING PROGRAM ...
executing 0x00400000 0x3c101001
executing 0x00400004 0x3c081001
executing 0x00400008 0x21080014
executing 0x0040000c 0x21290000
executing 0x00400010 0x96040000
```

```
executing 0x00400024 0x3129ffff
executing 0x00400028 0x01284820
executing 0x0040002c 0x3c101001
executing 0x00400030 0xa6090008
registers:
 0: 00000000
 1: 00000000
 2: 00000000
 3: 00000000
 4: 0000c0a8
 5: 00000000
 6: 00000000
 7: 00000000
 8: 00000002
 9: 0000479e
 10: 00000000
 11: 00000000
 12: 00000000
 13: 00000000
 14: 00000000
 15: 00000000
 16: 10010000
 17: 00000000
 18: 00000000
 19: 00000000
 20: 00000000
 21: 00000000
 22: 00000000
 23: 00000000
 24: 00000000
 25: 00000000
 26: 00000000
 27: 00000000
 28: 00000000
 29: 00000000
 30: 00000000
 31: 00000000
 Program Counter: 0x00400034
Memory region (0x10010000):
                  Value(+00)   Value(+04)   Value(+08)   Value(+0c)   Value(+10)   Value(+14)   Value(+18)   Value(+1c)
0x10010000        0x45000073   0x00004000   0x4011479e   0xc0a80001   0xc0a800c7   0x00000000   0x00000000   0x00000000
0x10010020        0x00000000   0x00000000   0x00000000   0x00000000   0x00000000   0x00000000   0x00000000   0x00000000
... DONE!
```

Figure 1 Output of the MIPS emulator when running the IP checksum calculator.

Marking Scheme

Aspect	Weighting
Functionality	65%
Code elegance, clarity, organization	30%
Self-marking	5%

Functionality

A+	The emulator executes correctly complex MIPS programs, *pc* register holds realistic text area addresses, support for half-word data access (*lhu, sh*).
A	All functionality in B plus support for branch (*bne*) and word-wise data access (*lw, sw*) instructions parsing and execution, registers are handled correctly in all cases, there is a method to terminate the program (e.g. *nop*).
B	All functionality in C plus support for R-type instructions parsing and execution (*addi, andi, lui*).
C	All functionality in D plus support for R-type instructions parsing and execution (`add, srl, sll`).
D	Program is read from the int array, a loop is implemented to execute the program (program counter is used), code is executed based on instruction opcode and func fields. A single instruction is implemented.

(*) This is an example of a marking scenario and marks are awarded proportionally for each functionality. If your code only implements the *lui*, *addi* and *andi* instructions, then you will get a D. Marking will test implementation correctness for each instruction independently.

Comments/clarity/organization

A	Clear structure (and helpful comments) about variable use. Well commented throughout. Spaced and organised for clarity. Appropriate use of function to organize code.
B	Good use of comments. Reasonable organisation, can follow program reasonably well. Variable use is sensible.
C	Sparse or sometimes unhelpful comments. Some consistency in use of variables.
D	No comments. Difficult to read code.

Self-marking

A	Clear and concise, functionality is accurate, comments reflect marking criteria
B	List of mostly accurate grades, comments are there but not entirely clear or unnecessarily long, or functionality grade is slightly off
C	Just a list of grades, mostly accurate (or close) but no explanation
D	Just a list of grades, inaccurate