ICAR Software Laboratory : VAX

The aim of this lab is to introduce the VAX-11/780 computer - a Virtual Address eXtension to the PDP-11 family of computers. The VAX-11/780 can be classified as a Complex Instruction Set Computer (CISC), dating from the 1970's – 1980's. At this time computer memory was slow and expensive, therefore code generation and code density were important design considerations in its development.

"A major goal of the VAX-11 instruction set was to provide for effective compiler generated code. Four decisions helped to realize this goal:

1) A very regular and consistent treatment of operators.
2) An avoidance of instructions unlikely to be generated by a compiler.
3) Inclusions of several forms of common operators.
4) Replacement of common instruction sequences with single instructions."

William Strecker 1978

To highlight the advantages of a CISC instruction set the VAX-11/780 will be compared to the PicoBlaze processor used in the previous lab i.e. a RISC like processor. To start the VAX11/78 instruction set simulator (ISS) select:

⊞ `-> VAX11Simulator`

Note, the original software and additional documentation can be downloaded from:

http://softlab-pro-web.technion.ac.il/projects/VAX11_Simulator/html/UserGuide.htm

The VAX11/780 is a 32-bit architecture i.e. 32-bit wide address bus, data bus and registers. There are 16 registers, `r0, r1, ..., r15`, however, registers `r12` to `r15` are system registers i.e. used by the processor for the program counter, stack pointer etc, so should not be used by the programmer. An instruction's *operation* field defines the instruction's function e.g. `Add`. An instruction's *data type* is assigned a single letter label, as shown in figure 1.

Number of Bits	Data type	Name	Label
8	Integer	Byte	B
16	Integer	**Word**	W
32	Integer	Long word	L
32	Floating point Single precision	F_floating	F
64	Integer	Quad word	Q
64	Floating point Double precision	D_floating	D
8n	Character string	Character	C

Figure 1: VAX11/78 data types

The VAX11/780 has a standard assembler opcode syntax, containing three fields:

```
<operation> <data type> <2 / 3>
```

The final field defines the number of operands in the instruction. A '2' indicates that the last operand is a source and destination i.e. similar to the PicoBlaze. A '3' indicates separate source and destination operands. Examples of both instruction formats are shown below (immediate and register addressing modes):

```
Addb2 R1, R0        ; R0 <= R1 + R0
Addb3 R1, R2, R3    ; R3 <= R2 + R1
Subw2 $100, R1      ; R1 <= R1 - 100
```

Note, the destination operand is specified last in the instruction.

Questions: when reading an instruction from memory what is the advantage of specifying the destination register last? Why could an `Addb2` instruction execute faster than an `Addb3` instruction? What is the advantage / disadvantage of an instruction set that allows the numbers of operands to be varied i.e. 2 or 3 operand formats?

The VAX11/780 is a good example of a CISC, supporting a large number of instructions and addressing modes, as shown in figure 3. Unlike the PicoBlaze almost any addressing mode can be used to specify the location of each operand e.g. memory, register or constant. Example:

```
addl3 15(r2),(r3)[r4], r1     ;3 operand Long word ADD
```

Figure 2: complex instruction

This instruction fetches two operands from memory. The first uses the displacement addressing mode: `15(r2)`. This adds the constant 15 to the base address stored in register `r2`. The second operand uses the indexed addressing mode: `(r3)[r4]`. Here the base address is stored in register `r3`, offset by the value stored in `r4`. This offset is multiplied by the data type size, in bytes e.g. long = 32bit, the multiplier = 4 (data type dependent). In the above example if `r4`=2 the offset would be 8 (memory is byte addressable).

Operand Addressing mode	Syntax	Example	Description
Immediate	$value	$100	Constant 100 (hex)
Absolute	*$address	*$100	Memory[100]
Register	rn	r3	Resister R3
Register deferred	(rn)	r3	Memory[r3]
Displacement	Offset(rn)	100(r3)	Memory[r3 + 100]
Deferred displacement	*Offset(rn)	*100(r3)	Memory[Memory [r3 + 100]]
Indexed	(rn)[rm]	(r3)[r4]	Memory[r3 + r4 × d]
Autoincrement	(rn)+	(r3)+	Memory[r3]; r3 = r3 + d
Autodecrement	– (rn)	–(r3)	r3 = r3 – d ; Memory[r3]

Figure 3: VAX11/78 addressing modes

Questions: assuming that the opcode field is 8bits long and constants use the word data type (refer to figure 1), how many bits will be required to encode the `addl3` instruction shown in figure 2? If operands can be either register, memory, or constants, what is the maximum and minimum number of bits required to represent this instruction?

Questions: the example shown in figure 2 uses indexed addressing to access one of its operands i.e. two registers: base address `(r3)` and offset `[r4]`, where the offset value is data type dependent:

```
    accessed_memory_location = r3 + (r4 × d)
```

If the instruction data type is changed from `addl3` to `addw3` and the base register is 100 with a offset of 2 what memory location will be accessed?

A full description of the instructions supported in the ISS can be found by left clicking on the pulldown:

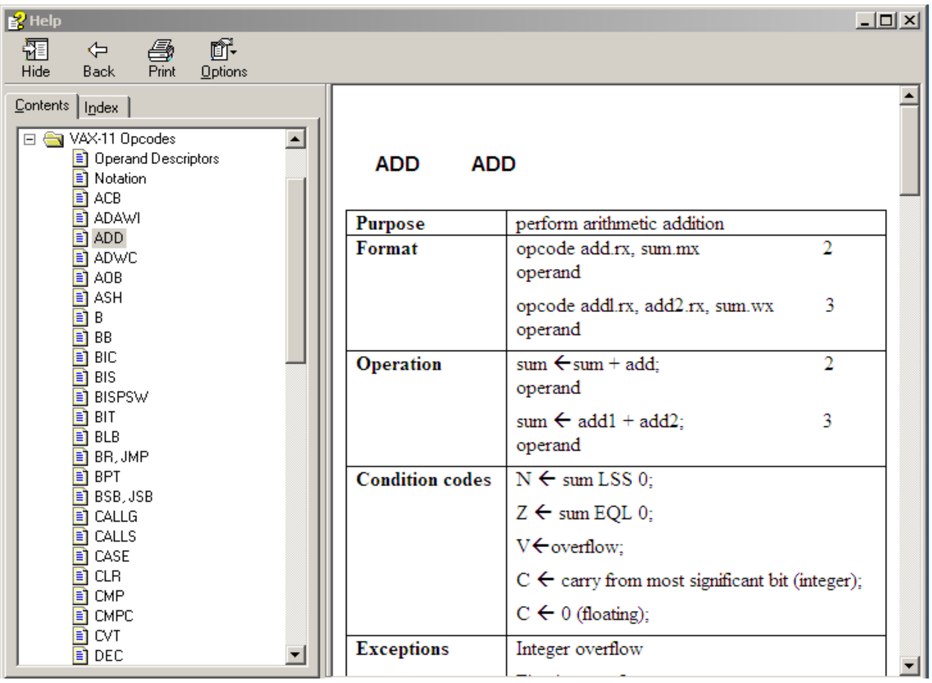
```
Help -> Contents -> Vax-11  Opcodes
```

for more information on the instructions in the previous examples expand this folder and select the relevant instruction, as shown in figure 4.

IMPORTANT: for security reasons(?) the help menu may not work from a network drive, therefore, copy the file `VAX11Simulator.chm` from :

```
T:\Computer Science\Apps\VAX11Simulator
```

to `C:\temp`, then double click on this file to read the help menu shown in figure 4.



ADD ADD

Purpose	perform arithmetic addition
Format	opcode add.rx, sum.mx 2 operand opcode addl.rx, add2.rx, sum.wx 3 operand

Operation	sum ← sum + add; 2 operand
	sum ← add1 + add2; 3 operand

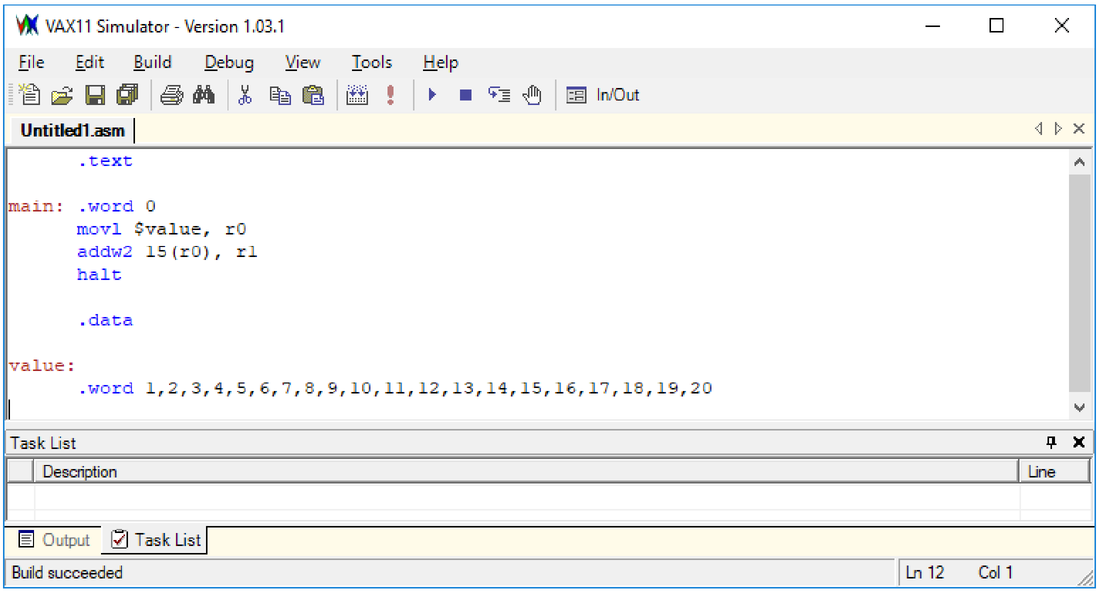
Condition codes	N ← sum LSS 0;
	Z ← sum EQL 0;
	V ← overflow;
	C ← carry from most significant bit (integer);
	C ← 0 (floating);

Exceptions	Integer overflow

Figure 4: Opcode syntax

Task 1

Start the VAX11/78 simulator (ISS), then within the main textbox enter the program shown in figure 5. The VAX11/78 uses a Von-Neumann architecture, therefore the assembler directives `.text` and `.data` are used to define where instructions and data are stored in memory. Labels must end with a '**:**', **all** values are decimal, for a hexadecimal numbers prefix with `0x`. Data can be embedded within the program using the `.word` assembler directive (16bit), for more information on these commands refer to the Assembler's Directives section within the help menu.



```
        .text

main:   .word 0
        movl $value, r0
        addw2 15(r0), r1
        halt

        .data

value:
        .word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
```

Figure 5: Test program

Note, your program must always start with the `.word 0` command, this pads the start of the program such that the computer starts at the first instruction. To compile this program left click on the Compile icon , or the pulldown:

```
Build -> Compile
```

any errors will be displayed in the task list, in the bottom panel. To simulate this program left click on the step icon , or the pulldown:

```
Debug -> Step
```

this will launch the ISS as shown in figure 6, the next instruction to be simulated is highlighted in yellow. Single step through this program by clicking on the step icon or the above pulldown menu.

As the program executes, register and memory values will be updated in the right hand side and bottom panels. To restart the program click on the Restart Program icon ■, or the pulldown:

```
Debug -> Restart Program
```

<u>Questions</u>: Why is a move `long` instruction used to load `R0`? **Hint**, how will this data be used? The operand `$value` represents the address of a symbolic label.

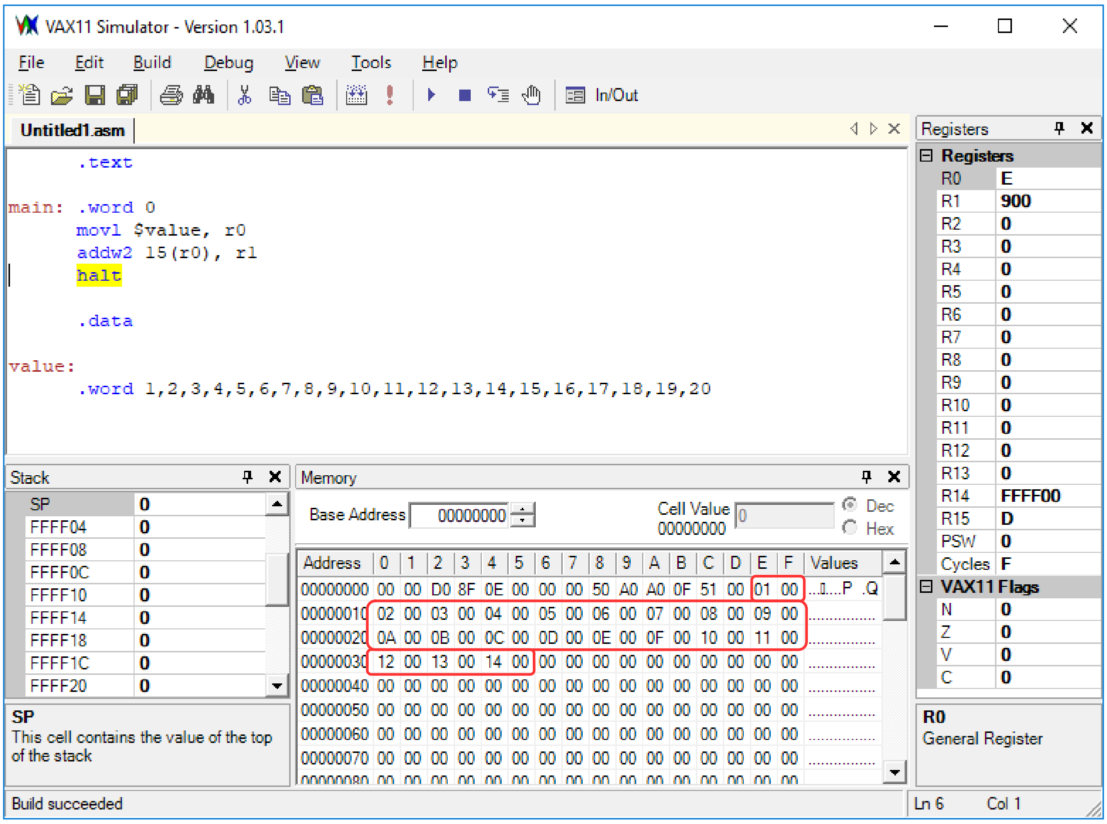
Where and how are the data values 1 – 20 stored in memory? How many bytes (memory locations) are used to store each number?

Hint, this data is stored at address 0x0E onwards, as shown in figure 6 (register `R0`). What data type is used? Try changing the `.word` assembler directive to `.byte` and reload the code, what has happened to the values stored in memory?

If the intention of this program was to read the 15th element from the list of numbers, why is the result incorrect? Ensure you understand these questions before you proceed. The value 0x900 is not correct, can you see why?

Hint: the operand `15(r0)` should point to the first byte of a number stored in memory e.g. data value 01, 02, 03 etc. Data is stored in memory using the `.word` data type i.e. a 16bit representation. Each number will be allocated two memory locations (two bytes) of storage. What would happen if the address generated is incorrect and points into the 'middle' of a number? What order are bytes read from memory for the `.word` data type, LSB or MSB first?

Edit your program, replacing the displacement offset 15 with the value 28 i.e. the offset needed to really access the 15th element from the list. Re-compile and run this program. Ensure program produce the correct result.


VAX11 Simulator - Version 1.03.1
File Edit Build Debug View Tools Help
In/Out
Untitled1.asm
Stack
Memory
Registers

```
        .text

main:   .word 0
        movl $value, r0
        addw2 15(r0), r1
        halt

        .data

value:
        .word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20
```

Stack	
SP	0
FFFF04	0
FFFF08	0
FFFF0C	0
FFFF10	0
FFFF14	0
FFFF18	0
FFFF1C	0
FFFF20	0

SP
This cell contains the value of the top of the stack

Base Address 00000000 Cell Value 00000000 0 ● Dec ○ Hex

```
Address  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  Values
00000000 00 00 D0 8F 0E 00 00 00 50 A0 A0 0F 51 00 01 00 ...Ð...P .Q
00000010 02 00 03 00 04 00 05 00 06 00 07 00 08 00 09 00 ................
00000020 0A 00 0B 00 0C 00 0D 00 0E 00 0F 00 10 00 11 00 ................
00000030 12 00 13 00 14 00 00 00 00 00 00 00 00 00 00 00 ................
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Registers	
R0	E
R1	900
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
R13	0
R14	FFFF00
R15	D
PSW	0
Cycles	F
VAX11 Flags	
N	0
Z	0
V	0
C	0

R0
General Register

Build succeeded Ln 6 Col 1

Figure 6: Simulator (data block highlighted in red)

Task 2

Write a **PicoBlaze** program to add the constant 0x10 to the external memory location 0x25. The address 0x25 can be hard-coded as a constant within the program. How many instructions are required? Now write the same program in the VAX11/78 instruction set.

Hint, you should be able to perform the same function on the VAX11/78 in **one** instruction using the `Add` instruction combined with absolute and immediate addressing modes, similar to that shown in figure 2. Refer to figure 3 for addressing mode syntax. Note, use ‘0x’ for hex, don't forget the ‘*’ for an address.

Immediate	$value	$100	Constant 100 (hex)
Absolute	*$address	*$100	Memory[100]

Task 3

To illustrate the power of a CISC instruction set we shall use the vector addition algorithm from the previous lab. This program performs a vector addition, adding together 16 pairs of data values stored in external memory, as shown in figures 7 and 8. From the previous laboratory it was calculated that the instruction count (IC) for this program on the PicoBlaze processor was :

$$IC = 3 + 11 \times 16 + 1 = 180 \text{ instructions}$$

Now write a VAX11/78 program to implement the functionality shown in figure 7.

Remember when writing CISC programs we are looking to replace groups of simple instructions with a single complex instruction, otherwise the resultant code will not use the CISC's specialised hardware and therefore, will take longer to run. To implement this program you may wish to consider using the instructions `CLR` and `ACB`. For more information on how to use them and examples of their use refer to the help menu (`.chm` file)

Note, data values use the `.Byte` data type. In this computer, data can not be stored at address 0x00 i.e. its a Von-Neumann not a Harvard architecture, therefore, use labels to define read and write start addresses within memory i.e. define a `.data` region after your program as shown in figure 6, then define and use symbolic labels similar to `$value`.

Hint, you should be able to implement this program using six instructions, three to initialise variables, two in the main loop and one to stop the program. You can also use the counter register in the `ACB` instruction as your index register i.e. add two to the counter each loop.

Assuming that the PicoBlaze and the VAX11/78 use the same clock speed what is the speedup of the VAX11/78 compared to the PicoBlaze processor. What are the possible advantages / disadvantage of the VAX11/78 processor compared to the PicoBlaze processor? If you were unable to find a solution refer to Appendix A.

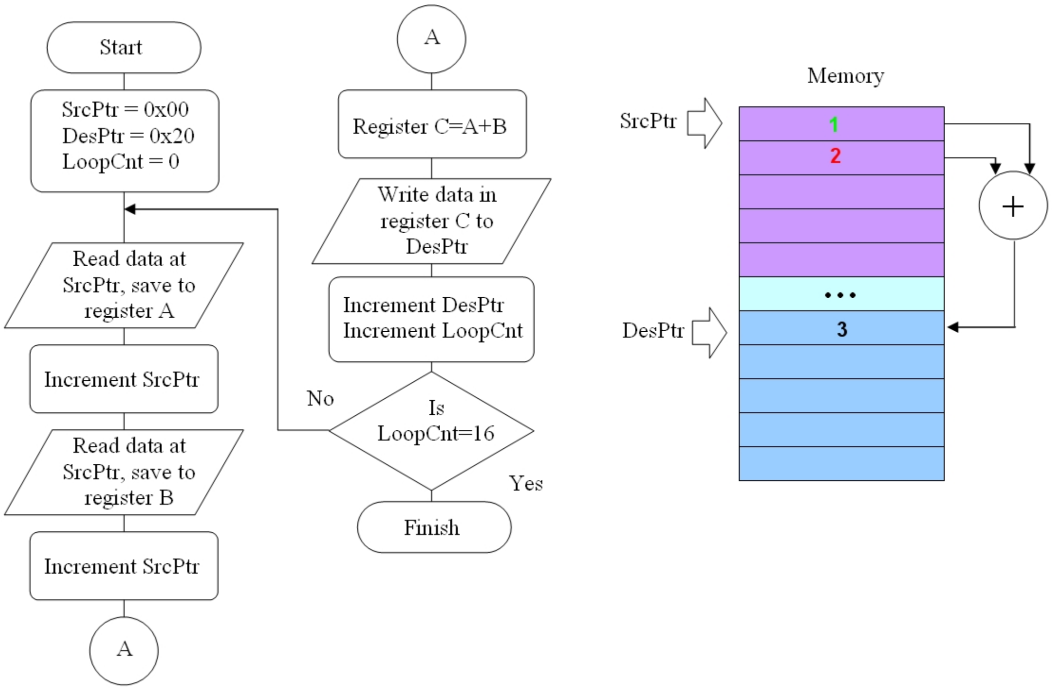


Figure 7: Vector addition flowchart

```
; RAM
#EQU ram_src_addr, 0x00     ; read address
#EQU ram_des_addr, 0x20     ; write address

main:
  load S0, ram_src_addr     ; set source / destination
  load S1, ram_des_addr     ;
  load S2, 00               ; zero loop counter

loop:
  rdprt S3, (S0)            ; read par 0
  add S0, 01                ; inc src ptr
  rdprt S4, (S0)            ; read par 1
  add S0, 01                ; inc src ptr
  add S4, S3                ; result = par 0 + par 1
  wrprt S4, (S1)            ; store at des ptr
  add S1, 01                ; inc des ptr
  add S2, 01                ; inc loop counter

  load S3, S2               ; copy count
  sub S3, 10                ; have 16 values been processed
  jump NZ, loop

trap:
  jump trap
```

Figure 8: Vector addition PicoBlaze program

```
.text
main: .word 0
      locc $SPACE, $LENGTH, str
      subw3 r0, $LENGTH, r0
      pushl r0
      pushal message
      calls $2, .printf

finish:
      halt

.set SPACE, 0x20
.set LENGTH, 33

.data
str:     .asciz "abcd ef ghi jklm nop qrst uvwx yz"
message: .asciz "First word is %d characters long\n"
```

Figure 9: application specific instructions

Task 5

In addition to complex instructions and addressing modes the VAX11/78 also supported application specific instructions e.g. character processing.

CMPC	: compare two character strings
MOVC	: move character string
LOCC	: locate character within a string
MATCHC	: find substring within character string

The VAX simulator also supports system calls to standard-in and standard-out i.e. the keyboard and monitor (console window). To write characters to the display the `printf` system subroutine is called, data to be displayed is pushed onto the processor’s data stack in reverse order, as shown in figure 9. A full description of the instructions, assembler directives and system calls can be found in the help menu by left clicking on:

```
Help -> Contents -> Vax-11  Opcodes
                 -> Assembler’s directives
                 -> System Calls
```

Enter the program in figure 9, single step through these instruction, what function does this program perform? Refer to the Help menu for more information on each instruction and assembler directive.

Hint, the ASCII code for the space character is 0x20. Try removing the space between the `d` and `e` characters.

Task 6

Modify the previous program to search for a user defined sub-string i.e. the string labelled '`key`', as shown in figure 10. If a match is found print out the locations of each match i.e. character offset from the start of the string. If no match is found print out an error message.

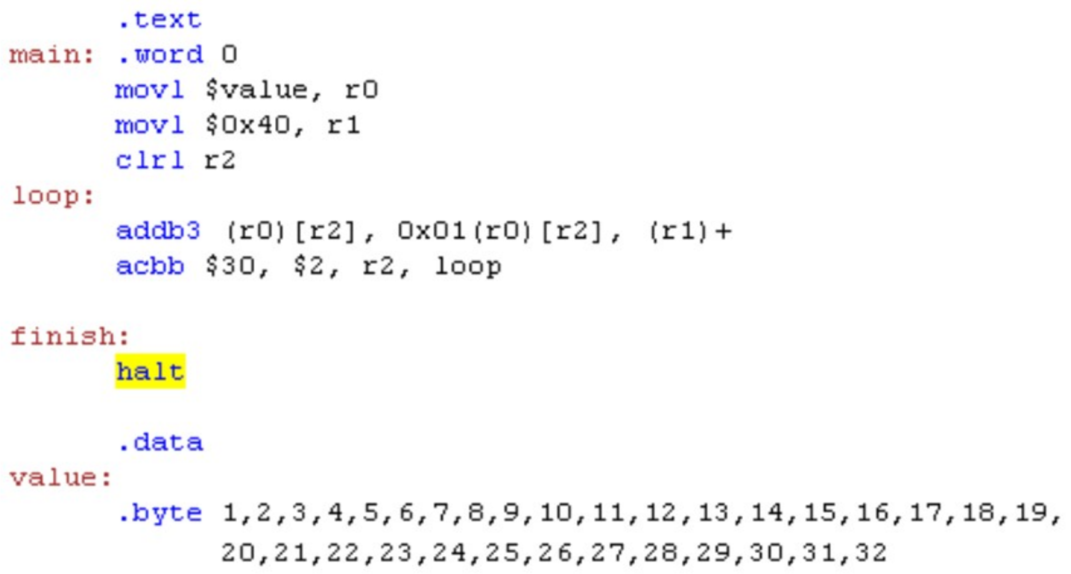
```
.set SPACE, 0x20
.set LENGTH, 33

.data
str:     .asciz "abcd ef ghi jklm nop qrst uvwx yz"
key:     .asciz "nop"
message: .asciz "Match: %d\n"
```

Figure 10: application specific instructions

Hint, you may wish to look at the `MATCHC` instruction. There may be a software bug in the simulator for this instruction, but do give it a try. Alternatively, you could use the `CMPC` and `LOCC` instructions. As with all CISC programs try to get as much functionality as possible into each instruction.

Appendix A



```
      .text
main: .word 0
      movl $value, r0
      movl $0x40, r1
      clrl r2
loop:
      addb3 (r0)[r2], 0x01(r0)[r2], (r1)+
      acbb $30, $2, r2, loop

finish:
      halt

      .data
value:
      .byte 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,
            20,21,22,23,24,25,26,27,28,29,30,31,32
```

Figure A1: solution for task 3