

# CSCI131

## Introduction to Computer Systems

### Task Group 2: Assembly language programming

*Complete the exercises, and try to get them marked, before attempting the assignment!*

There are several exercise tasks in this task group as detailed below. At the end of this document there is a link to the assignment. You should try to complete all the exercise tasks before attempting the assignment task.

Some tasks require that you simply run a supplied assembly language program on the simulator. When running the program on the simulator, you are supposed to be following how the program works by reference to the lecture notes. For these tasks, you need only take a couple of screenshots to prove completion. Other tasks require that you enter and run code.

When you have completed all the tasks, show your results to a tutor to be credited with marks.

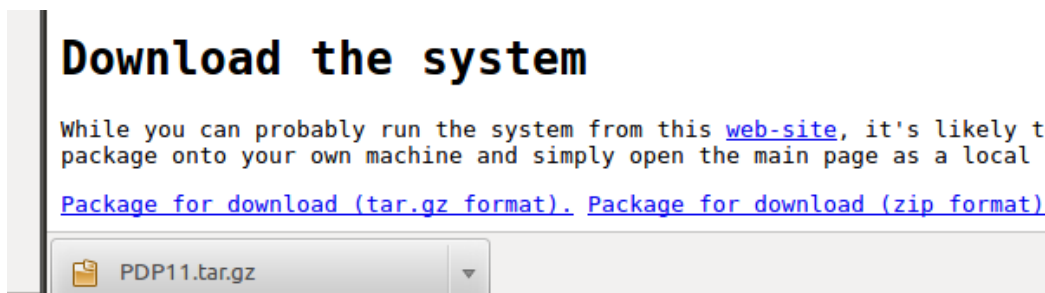
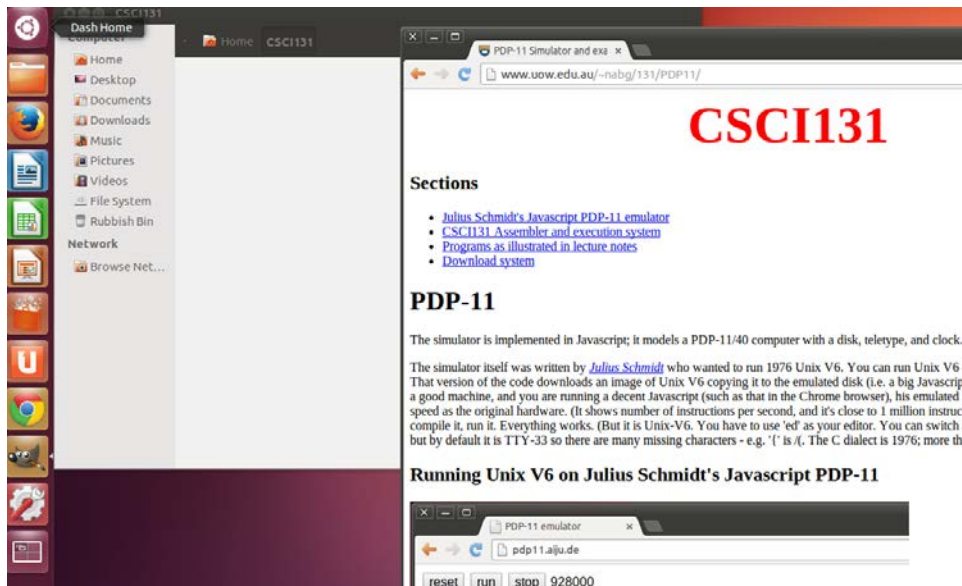
You will be writing and running PDP11 code.

Your PDP11 computer is actually a Javascript program written by [Julius Schmidt](#), a German physics student who had an interest in computing and Unix. On a good modern PC, when run with a decent Javascript interpreter such as that in the Chrome browser, Schmidt's simulator runs at close to the speed of the original hardware! (Challenge: Schmidt, a physics student, wrote his Javascript simulator using the PDP11-40 system manual as a guide; can you, a computer science student, write another version in Python/Java/C/C++ without peeking too much at Schmidt's code?)

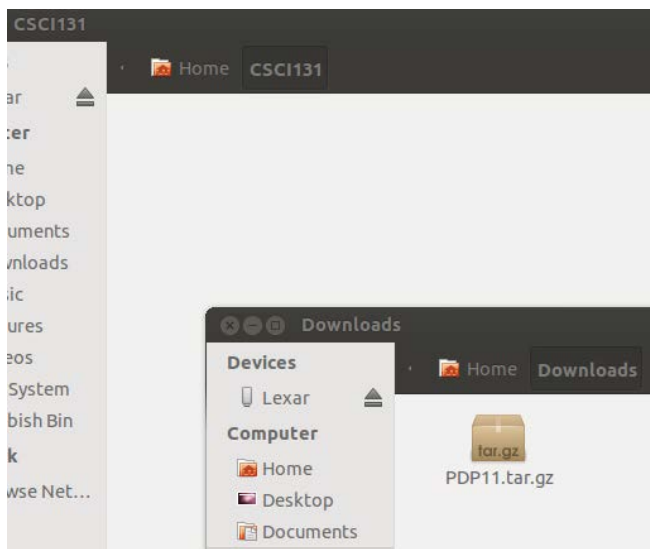
Because the simulator is in Javascript, it is run as a web application. You can use it over the Internet by connecting to <http://www.uow.edu.au/~nabg/131/PDP11/Page1.html>; but it will probably be easier for you to download the entire application and run it locally on your own machine.

## Task 1: Copying a string and counting the characters: 2 marks

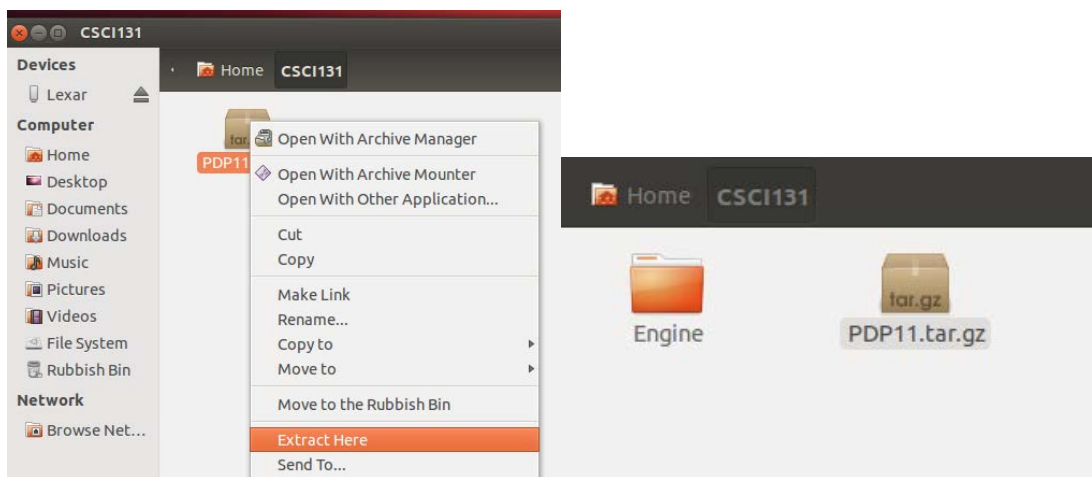
Create a sub-directory for you work on this task group. Connect to the UOW server with the code ([www.uow.edu.au/~nabg/PDP11/](http://www.uow.edu.au/~nabg/PDP11/)) and download the system.



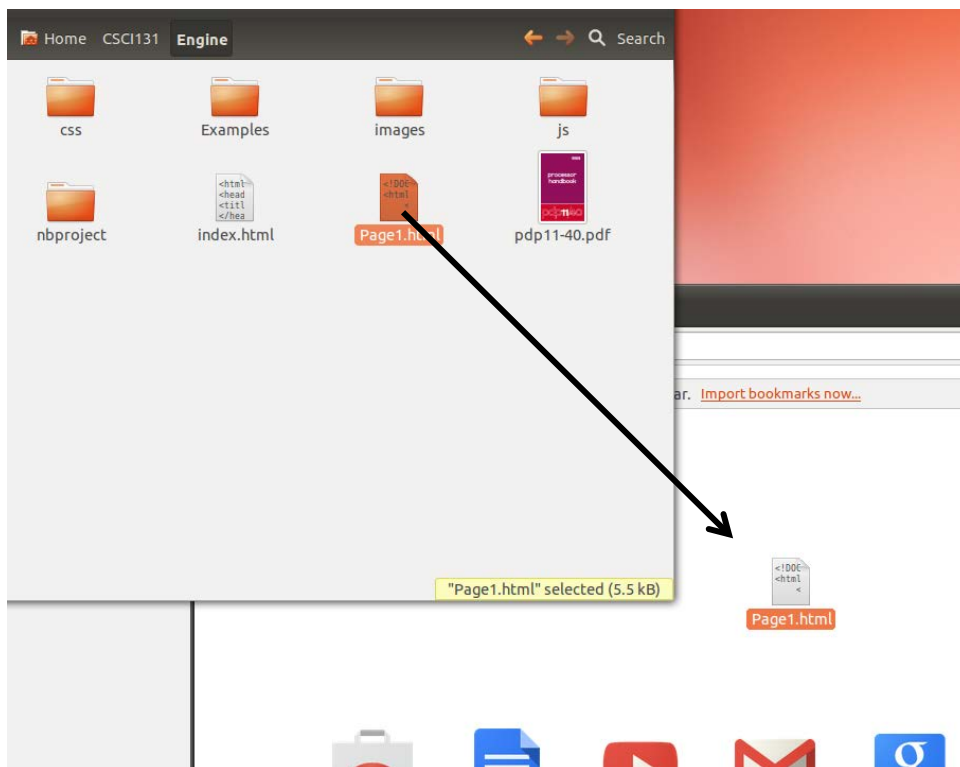
The code is provided as a “tar.gz” Unix archive (or a zip file for Microsofties).



Unpack the file



You run it by using a browser to open the HTML page that contains the Javascript code ("Page1.html"). For the Chrome browser in the labs, you drag the file into a Chrome window:



The HTML page consists of four tabs:

- Editing and Assembly
- Symbol table
- Program execution
- Dump memory



For this first task, you will run the lecture example of a program that copies a string and counts the number of characters. A roughly equivalent C++ program is:

```
* Author: nabg
*
* Created on 29 May 2013, 3:48 PM
*/

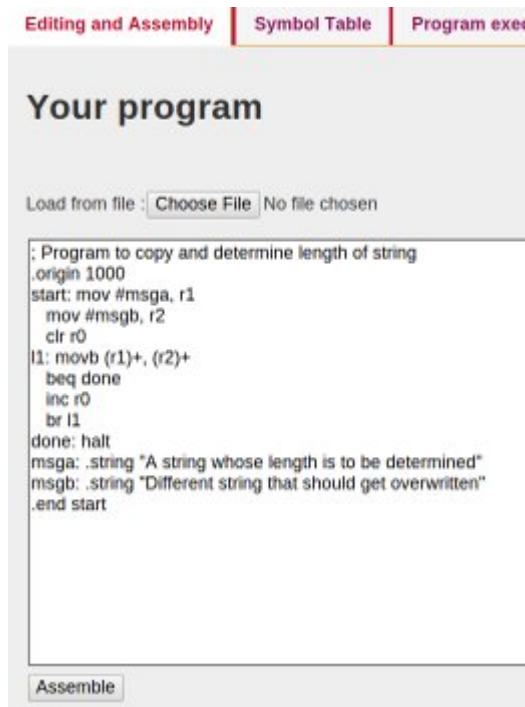
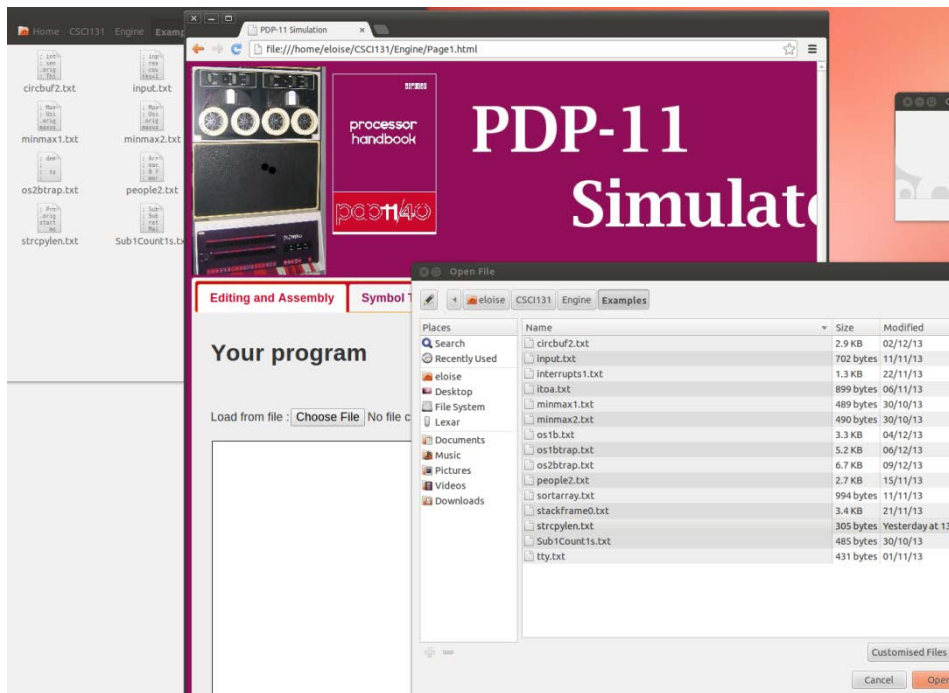
#include <cstdlib>

using namespace std;

/*
 *
 */
int main(int argc, char** argv) {
    int r0; // count of characters
    char msga[] = "A string whose length is to be determined";
    char msgb[100]; // Place for a copy of the string
    char *r1 = msga; // Set pointers appropriately
    char *r2 = msgb;
    r0 = 0; // zero out count
    while(*r1) {
        *r2 = *r1;
        r1++;
        r2++;
        r0++;
    }
    // Need to put null at end of msgb
    *r2 = 0;
    return 0;
}
```

(You could try creating a C++ NetBeans project for that code and use “Debug” to run the C++ under control of gdb. You should see that it operates almost exactly like the assembly language version.)

The source for the assembly language code is included in the Engine/Examples folder. You can open the file from the displayed HTML page, or just type in the assembly language code:



Assemble the code and switch to the program execution window.

# Generated code

```

; Program to copy and determine length of string
.origin 1000
start: mov #msga, r1

      mov #msgb, r2

      clr r0
l1:    movb (r1)+, (r2)+
      beq done
      inc r0
      br l1
done:  halt
msga:  .string "A string whose length is to be determined"

      msgb: .string "Different string that should get overwritten"

      .end start

```

Address	Content
001000	012701
001002	001024
001004	012702
001006	001076
001010	005000
001012	112122
001014	001402
001016	005200
001020	000774
001022	000000
001024	020101
001026	072163
...	...
001074	000144
...	...
001076	064504
001100	063146
...	...
001150	067145
001152	000000

(both shown as octal numbers – encoding bit patterns)

Editing and Assembly
Symbol Table
Program execution
Dump memory

## Generated code

```


; Program to copy and determine length of string
.origin 1000
001000 012701 start: mov #msga, r1
001002 001024
001004 012702 mov #msgb, r2
001006 001076
001010 005000 clr r0
001012 112122 l1: movb (r1)+, (r2)+
001014 001402 beq done
001016 005200 inc r0
001020 000774 br l1
001022 000000 done: halt
001024 020101 msga: .string "A string whose length is to be determined"
001026 072163
001030 064562
001032 063556
001034 073440
001036 067550

```

## Execution

R0=000000	R1=000000	R2=000000	R3=000000	R4=000000	R5=000000	R6 (SP)=001000	R7 (PC)=001000
Status bits	N <input type="checkbox"/>	Z <input type="checkbox"/>	V <input type="checkbox"/>	C <input type="checkbox"/>	Priority	0	

Step
StepStep
Restart



Use “Step” to single step through the execution of the program (not the whole way, just for two or three cycles of the loop after which you can switch to “Step Step” continuous execution to let the program finish). You can switch back from “StepStep” to single step mode by clicking the Step button.

As the program executes, the line with the last executed instruction is highlighted in the "Generated code" text area:

## Generated code

```

; Program to copy and determine length of string
.origin 1000
001000 012701    start: mov #msga, r1
001002 001024
001004 012702        mov #msgb, r2
001006 001076
001010 005000        clr r0
001012 112122    l1: movb (r1)+, (r2)+
001014 001402        beq done
001016 005200        inc r0
001020 000774        br l1
001022 000000    done: halt
        msga: .string "A string whose length is to be determined"
001024 020101

```

The registers, status flags, and priority settings are updated as each instruction is executed.

## Execution

R0=000000	R1=001025	R2=001077	R3=000000	R4=000000	R5=000000	R6 (SP) =001000	R7 (PC) =001014
<b>Status bits</b>	N <input type="checkbox"/>	Z <input type="checkbox"/>	V <input type="checkbox"/>	C <input type="checkbox"/>	Priority	0	

Run the program to completion. (Step-step mode slows down Schmidt's simulator greatly to allow the steps to be observed.)

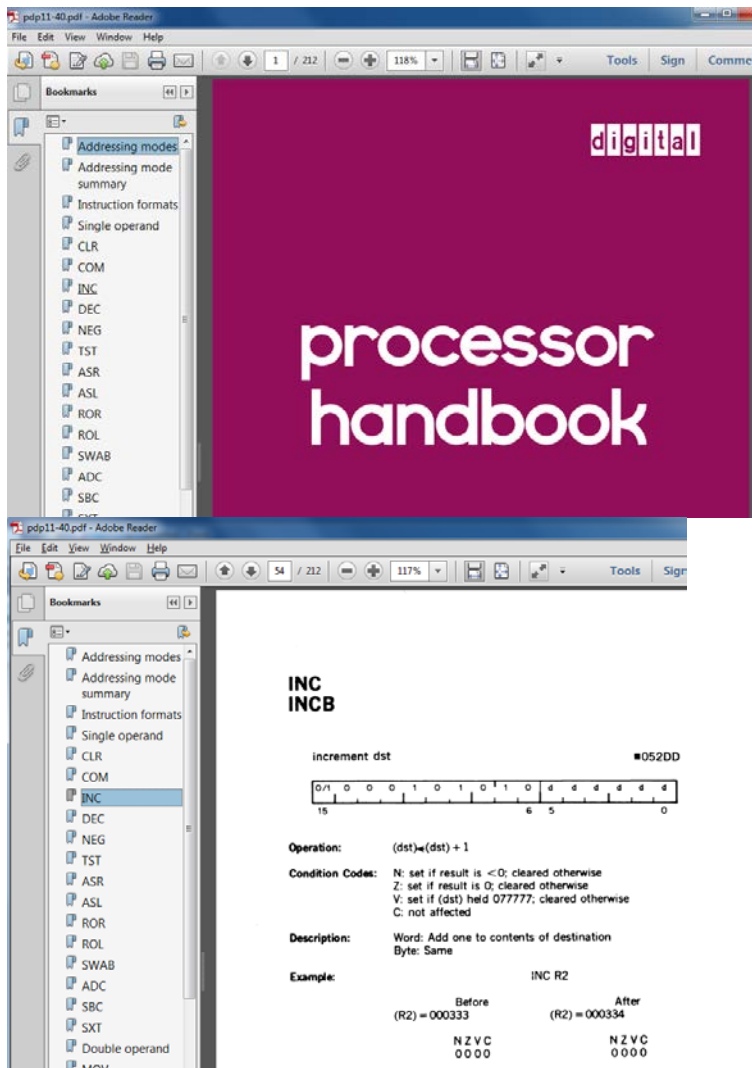
Check the results. The length of the string will be shown as the final value in register r0 (remember it is displayed in octal, work out the length and check that it is correct for the message string).

Dump the memory. Inspect the memory dump and confirm that the characters from msga have been used to overwrite those originally in msgb.

Check that you understand the assembly directives `.origin`, `.string`, `.end` (see slide 45 in [05PDP11Details.pdf](#)).

Look up the documentation on each of the instructions used in the program. (`mov`, `clr`, `movb`, `beq`, `inc`, `br`, `halt`) The PDP11-40 manual is included in your download:





Use [the lecture notes](#) (slides around 50) to get an explanation of the addressing modes used when accessing data:

001000 012701 start: mov #msga, r1

- Move word instruction – 4 bits - 01

MOV MOVB .origin 1000  
001000 012701 start: mov #msga, r1  
001002 001024

move source to destination

• Source: immediate addressing mode – mode 2 register 7 - 6 bits for ss - 27

• Destination – register mode – mode 0, register 1 – 6 bits for dd - 01

• Code word 012701

• Next word has address of msga

001012 112122 l1: movb (r1)+, (r2)+

- Move byte instruction – 4 bits - 11

MOV MOVB

move source to destination

• Source: autoincrement addressing mode – mode 2 register 1 - 6 bits for ss - 21

• Destination – autoincrement addressing mode – mode 2 register 2 – 6 bits for dd - 22

• Code word 112122

001014 001402 beq done

- Branch instruction – 8 bits - 014

BEQ

branch if equal (to zero) 001400 Plus offset

• Offset – 8 bits for a word count

– Here assembler has worked out it's a branch forward for two words (4 bytes)

- Instruction at 1014 was fetched
- PC (r7) was incremented to 1016
- On execution add 4 to 1016 getting 1022 (yes – that is right, this is odd)
- 1022 is address for "done"

001014 001402 beq done  
001016 005200 inc r0  
001020 000774 br l1  
001022 000000 done: halt

001020 000774 br l1

- Branch instruction – 8 bits - 004

BR

branch (unconditional) 000400 Plus offset

• Offset – 8 bits for a word count

– Here assembler has worked out it's a branch back for four words

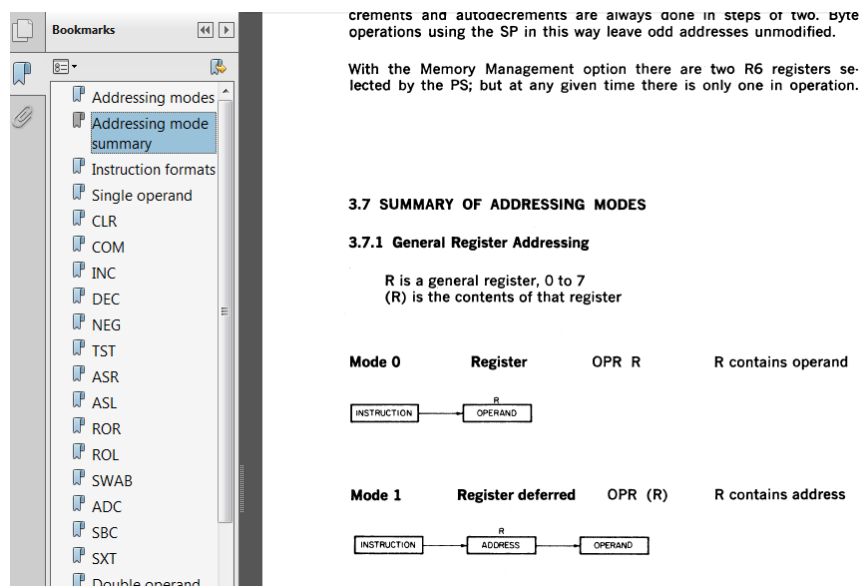
- Instruction at 1020 was fetched
- PC (r7) was incremented to 1022
- On execution add -10 to 1022 getting 1012
- 1012 is address for "done"

001012 112122 l1: movb (r1)+, (r2)+  
001014 001402 beq done  
001016 005200 inc r0  
001020 000774 br l1  
001022 000000 done: halt

Offset value is 374 as an 8 bit signed value 11111100 which is -4



Read the explanations of addressing modes in the manual:



## Task 2: Data processing examples: 1 mark

The lecture notes contain a number of examples that illustrate how familiar programming constructs in C/C++ work at the assembly language level. The examples include array access (both using pointers in classical C style, and using array indexing), subroutines (functions/procedures/methods/... whatever name you prefer), recursive functions, and structs.

Pick **one** of the following examples from the lectures:

minmax1.txt

Example from [04PDP11.pdf](#), about slide 47, finding minimum and maximum values in an array. Loops through array using address pointer style code (like C/C++ `int data[16]; int* ptr = data; - int val = *ptr;`.)

minmax2.txt

Example from [04PDP11.pdf](#), about slide 59, finding minimum and maximum values in an array. Loops through array using indexed addressing (like C/C++ `int data[16]; int ndx=0; int val = data[ndx]` array subscripting code.)

Sub1sCount1s.txt

Example from [04PDP11.pdf](#), about slide 70, that illustrates call to a subroutine (with return address placed on stack etc).

sortarray.txt

Example from [05PDP11Details.pdf](#), about slide 80, sorting array. Illustrates array access with indexed addressing mode.

itoa.txt

Example from [05PDP11Details.pdf](#), about slide 90, showing recursion (recursive implementation of integer to ascii string function).

people2.txt

Example from [05PDP11Details.pdf](#), about slide 120. Just illustrating how C structs, and access to fields of struct, might be handled in assembly. It's a different way of using indexed addressing mode

Run your chosen example in the simulator. Make sure that you understand (and could explain to the tutor) the different instructions and addressing modes used in the code.

### ***Task 3: Examples relating to I/O: 1 mark***

The lecture notes contain a number of examples that illustrate how input and output are handled using both naïve “wait-loop” mechanisms and more sophisticated interrupt-driven approaches that allow overlap of I/O tasks and continued computation.

Pick **one** of the following examples from the lectures:

tty.txt

Example from [04PDP11.pdf](#), about slide 78, showing wait loop I/O being used while printing a message.

input.txt

Example from [05PDP11Details.pdf](#), about slide 120, showing wait loop I/O being used reading and echoing characters.

Interrupts1.txt

Example from [05PDP11Details.pdf](#), about slide 160, that illustrates output of message using interrupts while supposedly completing some compute task.

Run your chosen example in the simulator. Make sure that you understand (and could explain to the tutor) how I/O can be handled by polling (wait loops) or by using interrupts and why interrupt driven mechanisms allow for more efficient use of a computer system.

### ***Task 4: Proto operating systems: 1 mark***

The lecture notes contain a series of examples that illustrate aspects of a primitive operating system including “supervisor calls” (trap/emt/svc/...) that provide I/O related services (possibly implemented using interrupts).

Pick **one** of the following examples from the lectures:

os1b.txt

Example from [05PDP11Details.pdf](#), about slide 220, illustrating a simple non-interrupt driven "operating system".

The earliest operating systems incorporated utility functions such as `ascii<->integer` conversion functions as well as more fundamental things like i/o handling. In later systems, utility functions became part of standard libraries that were linked with user code. Also, in early operating systems, OS functions were invoked with simple subroutine calls. This code mimics such an early OS.

os1btrap.txt

Example from [05PDP11Details.pdf](#), about slide 260, illustrating "traps" or "supervisor calls". As operating systems became more sophisticated, extra instructions (`emt/trap/svc/...`) were added that were to be used in calls to the OS. This example is the same as the previous one but using "trap" instructions.

os2btrap.txt

Example from [05PDP11Details.pdf](#), about slide 280, illustrating "traps" or "supervisor calls" and an interrupt driven "OS".

Run your chosen example in the simulator. Make sure that you understand (and could explain to the tutor) how supervisor calls work.

### ***Task 5: Simple coding in assembly language: 1 mark***

You can never have enough practice in manipulating bits with logical operations. You did it in C; now try in assembler.

You are to implement an assembly language version of a subset of the following C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/*
 *
 */
int main(int argc, char** argv) {
    unsigned short a, b;
    a = (unsigned short) (getpid() % 65536);
    b = (unsigned short) (getuid() % 65536);
    printf("a %ho; b %ho\n", a, b);
    unsigned short c, d, e;
    c = a & b;
    d = a | b;
    e = a ^ b;
    printf("a & b %ho\n", c);
    printf("a | b %ho\n", d);
    printf("a ^ b %ho\n", e);
    return (EXIT_SUCCESS);
}
```

You should run the C code. Then you will have known data values, in octal, for 'a', 'b', 'c', 'd' and 'e'.

```

Logic (Build, Run) x Logic (Run) x
a 52350; b 1750
a & b 350
a | b 53750
a ^ b 53400

```

(Your data will differ.) This will help you check your assembly language code.

The assembly version will have the values of 'a' and 'b' defined using .word directives. It doesn't use any input and output; you check the results in the "Dump memory" display.

Try to work out the code for the operations yourself. Inclusive or is essentially the PDP11's "bit set" instruction – modify the destination by setting all those bits that are in the source. There is an XOR instruction. There isn't an "AND". But you can use "bit clear". Instead of doing "AND" with a bit pattern, you complement the pattern and then do "bit clear" (try it on paper and convince yourself that that will work.)

Load from file :  No file chosen

```

.origin 1000
; and - there is no and instruction
; but can get there using bic (bit clear)
; c = a & b;
start: mov a, c
      mov b, r0
      com r0
      bic r0, c
; or - well that is really just bit set
; d = a | b;
      mov a, d
      bis b, d
; and xor - well do have an xor
      mov a, e
      mov b, r0
      xor r0, e
      halt
a: .word 52350
b: .word 1750
c: .word 0

```

Symbol	Value
start	001000
a	001054
b	001056
c	001060
d	001062
e	001064

```

...      ...      ....
001000 016767 000050 000052 016700 000044 005100 040067 000040
001020 016767 000030 000034 056767 000024 000026 016767 000014
001040 000022 016700 000010 074067 000012 000000 052350 001750
001060 000350 053750 053400 000000 000000 000000 000000 000000
...      ...      ....

```

Look carefully in the dump; 'c' at 1060 contains 0350 – that's correct. Others are correct as well.

### Task 6: Processing some data: 2 marks

For this task, you will use the supplied “operating system” with its I/O handlers and utility functions for reading and writing strings and decimal integers.

A C version of the program that you are to implement is:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
static char input[128];
static char output[128];

void createreverse() {
    char* r1 = input;

    while(*r1) r1++;

    char *r2 = output + (r1 - input) - 1;
    r1=input;
    while(*r1) {
        *r2 = *r1;
        r1++;
        r2--;
    }
}

int main(int argc, char** argv) {
    memset(output,0, 128);
    gets(input);
    createreverse();
    puts(input);
    puts(output);
    return (EXIT_SUCCESS);
}
```

Output - ReverseApp (Run)

```
Hello World
Hello World
dlrow olleH
```

(The char[] will be in uninitialized memory; that is why it is necessary to clear at least the output array via memset().)

The functions gets() and puts() are defined in the stdio library. The function gets() reads in a line of characters from stdin, storing them in the array supplied (which is assumed to be of sufficient size). It reads until a newline character; this is not stored in the array. A nul byte is added to terminate the string. When working with stdin connected to a terminal, Unix/Linux automatically echoes input characters. The function puts() writes the contents of a character array to stdout; if connected to a terminal, it automatically appends a newline character. The gets() and puts() functions in stdio are part of “libc” – Linux’s standard library functions. Their implementation uses Linux’s read and write supervisor calls.

The function createreverse() reverses the string. You should make sure that you understand how the C code works. The use of pointers here is typical of C. You have probably been shown code like this, but you will have been encouraged to use a rather different style in your C++ coding.

The first while loop in `createreverse()` is finding the end of the string (keep incrementing the pointer `r1` until it is referencing a nul byte. The length of the string is the difference between the final value in `r1` and the start address of the input buffer. The pointer `r2` is then set to an appropriate offset position in the output buffer while `r1` is reset to the start of the input. The second while loop copies the bytes, decrementing `r2` and incrementing `r1` as it goes.

The assembly language version will be closer to the following version of `createreverse()`:

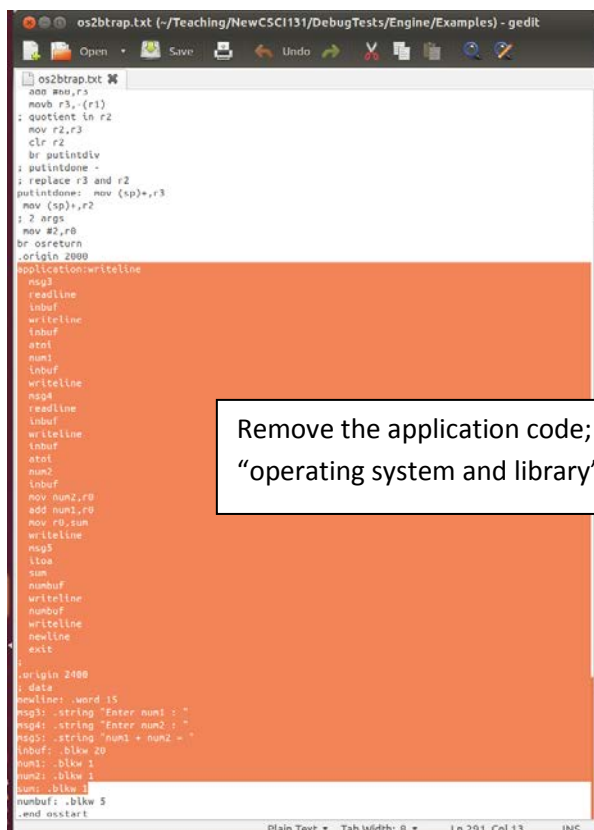
```
void createreverse() {
    char* r1 = input;

    while(*r1++) ;
    r1--;

    char *r2 = output + (r1 - input);
    r1=input;
    while(*(--r2)=*(r1++));
}
```

(Convince yourself that this code has the same effect as that shown previously.)

Start your assembly language version by taking the code of the example `os2btrap.txt` and cutting out the example application code to leave just the “operating system and library” components.



```
os2btrap.txt (~/Teaching/NewCSCI131/DebugTests/Engine/Examples) - gedit
; application code
; replace r3 and r2
; putintdone: mov (sp)+,r3
; mov (sp)+,r2
; 2 args
; mov #2,r0
; br osreturn
; origin 2000
; application: writeline
; msg3
; readline
; inbuf
; writeline
; inbuf
; atoi
; num1
; inbuf
; writeline
; msg4
; readline
; inbuf
; writeline
; inbuf
; atoi
; num2
; inbuf
; mov num2,r0
; add num1,r0
; mov r0,sum
; writeline
; msg5
; ltoa
; sum
; numbuf
; writeline
; numbuf
; writeline
; newline
; exit
; origin 2000
; data
; newline: .word 15
; msg3: .string "Enter num1 : "
; msg4: .string "Enter num2 : "
; msg5: .string "num1 + num2 = "
; inbuf: .blkw 20
; num1: .blkw 1
; num2: .blkw 1
; sum: .blkw 1
; numbuf: .blkw 5
; end osstart
```

Remove the application code; leave the “operating system and library” code.



The operating system functions `readline` and `writeline` are similar to `gets` and `puts`. (The `readline` function differs from `gets` in that it does include the newline character in the input buffer.) Of course, input is NOT automatically echoed.

Create a new application:

```
.origin 2000
application:readline
input
writeline
input
; still to do
; call createreverse
; writeline
;output
exit
.origin 3000
; data
input: .blkw 100
output: .blkw 100
.end osstart
```

Assemble

This has the input and output static arrays similar to the C code. It simply reads in a line of text, then outputs it. Run this part to make sure that you have correctly copied the code from `os2btrap.txt`.

Remember to click in the input text area before attempting to provide input to the program:



This part should run.

### Generated code

```

001246 001324
        write
001250 001374
        getint
001252 001466
        putint
        ;
        ; exit from os
001254 000240  exitos: nop
001256 000240  nop
001260 000000  halt
001262 000774  br exitos ; no escape
        ; read
        ; takes one argument - address of buffer where input to be
stored
        ; (argument will be in location immediately after trap
instruction)
        ; uses r0 and r1
        ; read characters (echoing them to teleprinter) and store until
newline dealt with
        ; (and add a null byte for safety before returning)

```

### Execution

R0=000000	R1=002014	R2=
Status bits	N <input type="checkbox"/>	Z <input checked="" type="checkbox"/>

Now implement the createreverse function:

```

application:readline
input
writeline
input
call createreverse
writeline
output
exit
createreverse: mov #input,r1
strlen:tstb (r1)+
bne strlen
dec r1
mov #output,r2
add r1,r2
sub #input,r2
mov #input,r1
cpy:movb (r1)+,-(r2)
bne cpy
return
.oriain 3000

```

Check that you understand all the instructions and addressing modes used. Confirm that it does implement the second version of the C code.

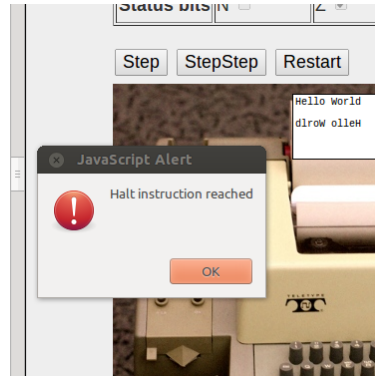
Try running.

(Hack. Sometimes it is useful to be able to switch from “StepStep” to single step mode at a specific point in a program. One of the unused instruction op-codes has been usurped to allow this! It is the FADD instruction 075000. The simulator does not have the “floating point” hardware so this is an invalid op-code – it was floating point add. Just insert that value in your code where you want single-step to start. Apart from changing simulator execution mode, it is a “no-op”.)

```

        getint
001252 001466 putint
        ;
        ; exit from os
001254 000240 exitos: nop
001256 000240 nop
001260 000000 halt
001262 000774 br exitos ; no escape
        ; read
        ; takes one argument - address of buffer where input to be
stored
        ; (argument will be in location immediately after trap
instruction)
        ; uses r0 and r1
        ; read characters (echoing them to teleprinter) and store until
newline dealt with
        ; (and add a nul byte for safety before returning)

```




---

## Exercises complete

Show the tutor that you have completed all the parts.

(That was 8 very easy marks wasn't it – just copy the code. Hopefully, you learnt a little about how assembly language works and how a primitive operating system can provide I/O and other services to applications.)

---

## Assignment

The [assignment](#): implement a little data processing program (using the supplied “operating system and library” to handle I/O).