ELEC2041

Microprocessors and Interfacing

Lecture 2 : C-Language Review - 1

http://webct.edtec.unsw.edu.au/

March, 2005

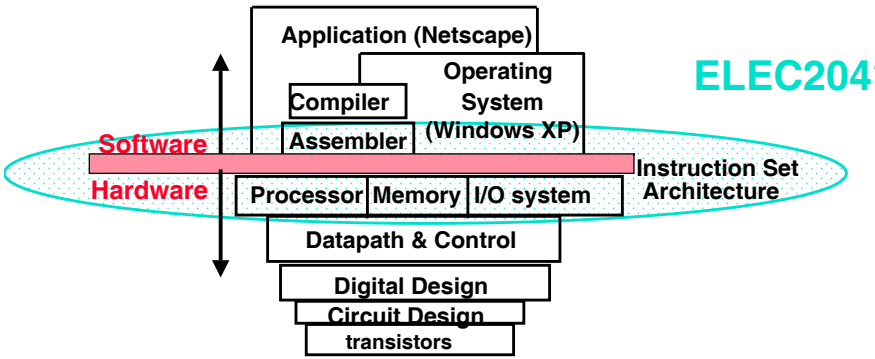
Saeid Nooshabadi

saeid@unsw.edu.au

Overview

- C Syntax
- Important Tidbits in C
- Pointers
- Passing Arguments to Functions

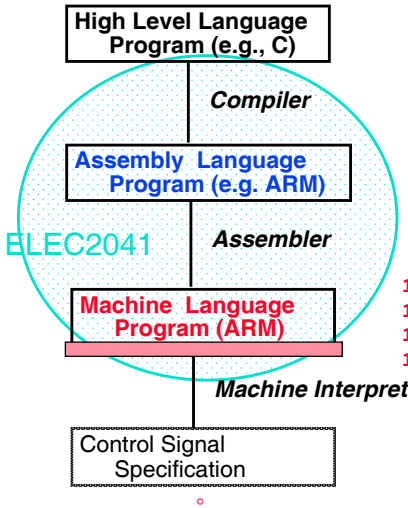
Review: What is Subject about?



ELEC2041

- Coordination of many *levels of abstraction*

Review: Programming Levels of Representation




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
ldr  r0 , [r2, #0]
ldr  r1 , [r2, #4]
str  r1 , [r2, #0]
str  r0 , [r2, #4]
```

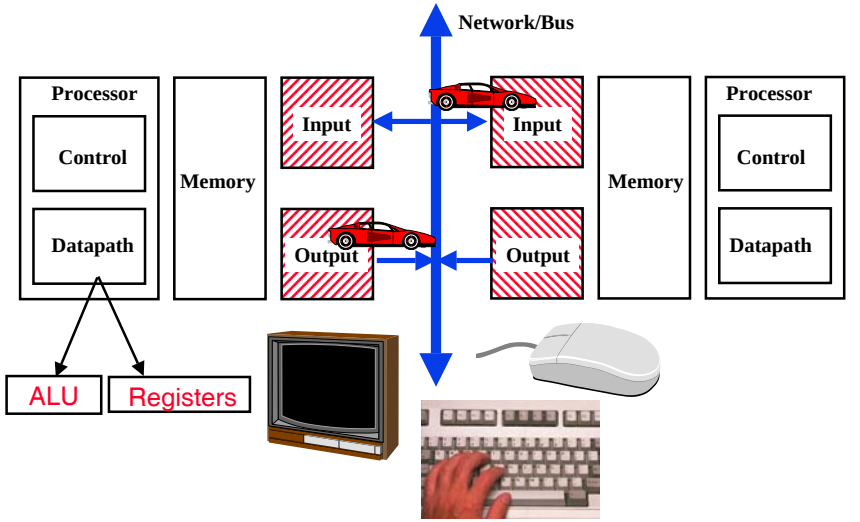
```
1110 0101 1001 0010 0000 0000 0000 0000
1110 0101 1001 0010 0000 0000 0000 0100
1110 0101 1000 0010 0001 0000 0000 0000
1110 0101 1000 0010 0001 0000 0000 0100
```

ALUOP[0:3] <= InstReg[9:11] & MASK

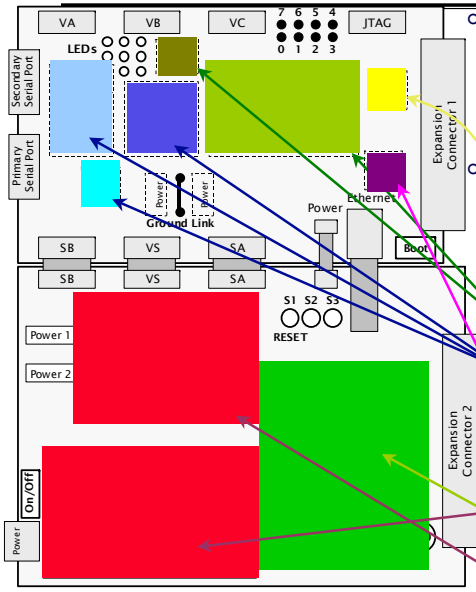
Review: What will You learn in ELEC2041?

- Learn big ideas in Microprocessors & Interfacing
  - 5 Classic components of a Computer
  - Principle of abstraction, used to build systems as layers
  - Data can be anything (integers, floating point, characters): a program determines what it is
  - Stored program concept: instructions just data
  - Principle of stack and stack frames
  - Compilation vs. interpretation thru system layers
  - Principle of Locality, exploited via a memory hierarchy (cache)

Review: 5 Classic Components of a Computer



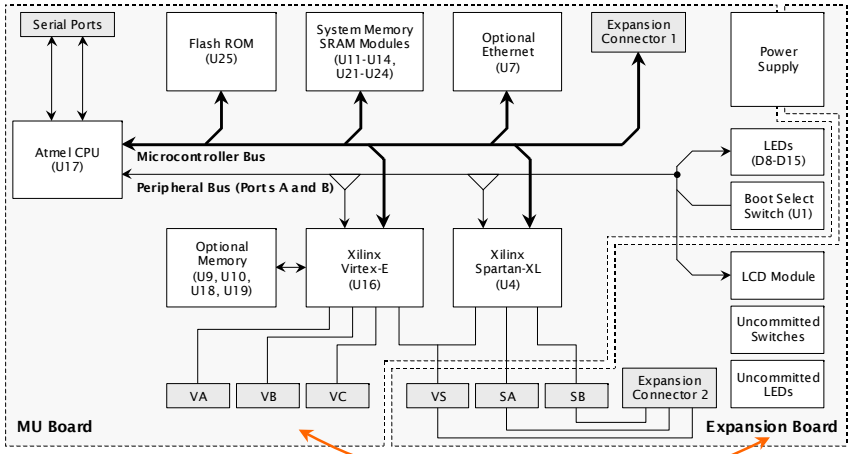
Review: ELEC2041 DSLMU Hardware



Two PCBs

- DSLMU is
  - A Board for 21st Century
  - The State-of-the-Art Development Board
- DSLMU Features:
  - Designed by the University of Manchester with lots of collaboration from UNSW
  - An ARM Microcontroller
  - With 2 MB of Flash and up to 4 MB of SRAM Memory
  - 2 Xilinx FPGAs for extended interfacing and specialised co-processors
  - Optional Ethernet chip
  - LCD Module
  - Lots of uncommitted Switches and LEDs
  - Terminal connector to FPGAs

Review: DSLMU Hardware Block Diagram



Two Printed circuit Boards

All Details and Circuit Diagrams included on the Companion CD-ROM

Quick Survey

- How many of you have experience with:

  Java?

  C++?

  C?

- Important: You will not learn how to code in C in this one lecture! You'll still need some sort of C reference for this course.

Compilation (#1/3)

- C compilers take C and convert it into an architecture specific machine code (string of 1s and 0s).
  - Unlike Java which converts to architecture independent code.
  - Unlike Haskell/Scheme environments which interpret the code.
- But how is it architecture specific?
  - You'll know the answer to this by the end of next week.

Compilation (#2/3)

- Advantages of C-style compilation:
  - Great run-time performance: generally much faster than Haskell or Java for comparable code (because it optimizes for a given architecture)
  - OK compilation time: enhancements in compilation procedure (`Makefiles`) allow only modified files to be recompiled

Compilation (#3/3)

- Disadvantages of C-style compilation:
  - All compiled files (including the executable) are architecture specific, depending on both the CPU type and the operating system.
  - Executable must be rebuilt on each new system.
    - Called "porting your code" to a new architecture.
- The "change→compile→run [repeat]" iteration cycle is slow

Bored@UNSW

- Boring Lectures and Boring Lecturers@UNSW!?
- All Lecturers you have had so far have been boring!
- ... I am a lecturer too. So I must be boring as well!
- I know many boring things. Many very, very boring things. I must be boring. I must bore. I am a lecturer.
- But...

  in another sense, to bore is to dig, to probe under the surface, to uncover that which has been hidden, to view that which has not previously been seen...

From Charles Yablon: "Forms"

C Syntax

```
#include <stdio.h>
int main (void) {
unsigned int exp = 1;
int k;
/* Compute 2 to the 31st.*/
for (k=0; k<31; k++) {
exp = exp * 2;
}
...
return 0;
}
```

Replaces current line by contents of specified file; "< >" means look in system area, " " " user area.

Declare before use; each sequence of variable declarations must follow a left brace.

main is called by OS

/*… */ indicate commnets. gcc accepts "//" comments but they aren't legal C.

C Syntax: General

- Very similar to Java, but with a few minor but important differences
- Header files (.h) contain function declarations, just like in C++.
- .c files contain the actual code.
- `main ()` is called by OS
- `main` can have arguments (more on this later):

```
int main (int argc, char *argv[])
```

- In no argument correct form is:

```
int main (void)
```

- Comment your code:
  - only /* */ works
  - // doesn't work in C
  - gcc accepts //

C Syntax: Declarations

- All declarations must go at the beginning of a C block, before assigning any values.
- Examples of incorrect declarations:

```
int c = 0;
c = c + 1;
d = 23; /* error */
char d;
for (int i = 0; i < 10; i++)
```

C Syntax: Structs

- C uses structs instead of classes, but they're very similar.
- Sample declaration:

```
struct alpha {
        int a;
        char b;
};
```

- To create an instance of this struct:

```
struct alpha inst1;
```

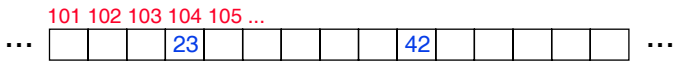
- Read up on more struct specifics in a C reference.

True or False?

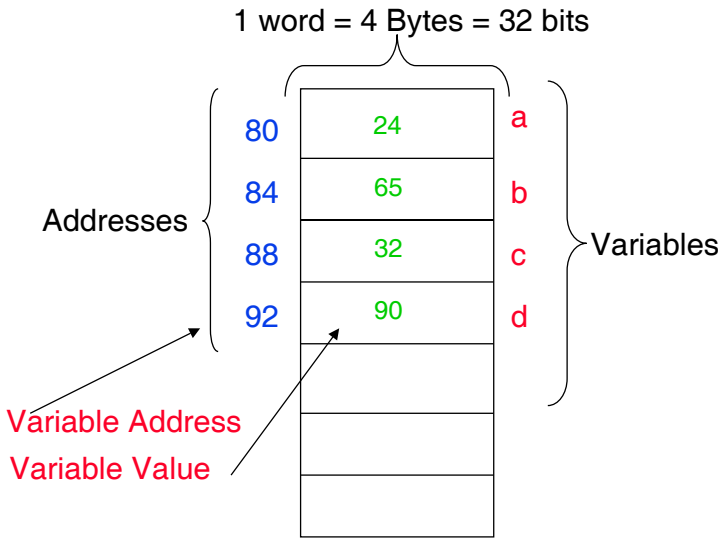
- What evaluates to FALSE in C?
  - 0 (integer)
  - NULL (pointer: more on this later)
- What evaluates to TRUE in C?
  - everything else…
- No such thing as a Boolean type in C.

Address vs. Value (#1/3)

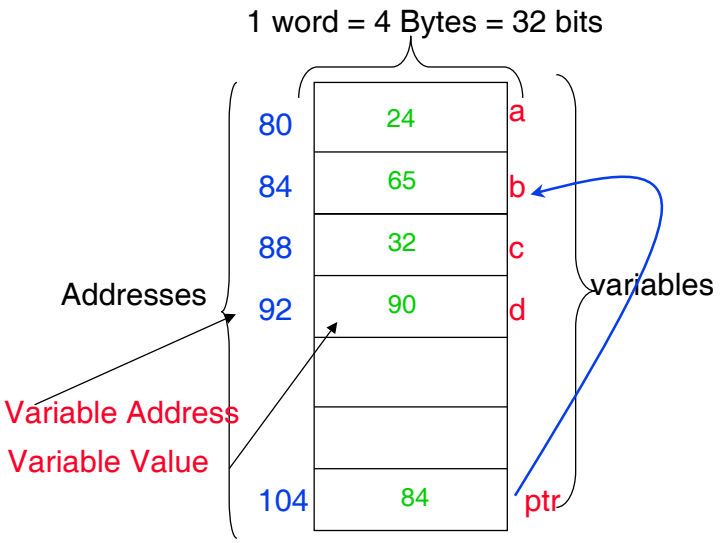
- Consider memory to be a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
- Don't confuse the address referring to a memory location with the value stored in that location.



Address vs. variable (#2/3)

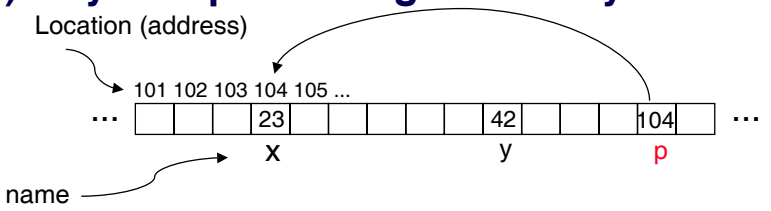


Address vs variable (#3/3)



Pointers in C (#1/7)

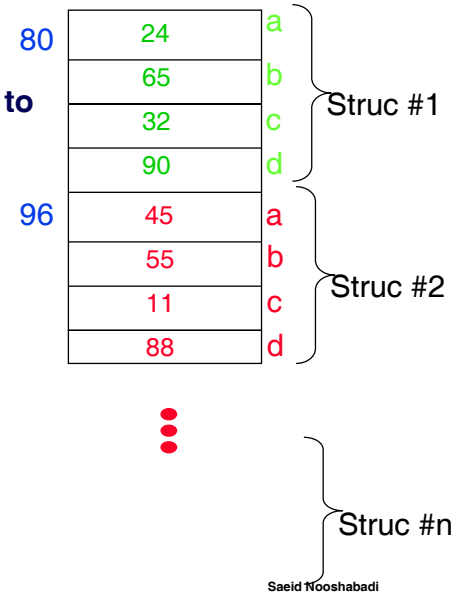
- An address refers to a particular memory location. In other words, it *points* to a memory location.
- **Pointer**: High Level Language (in this case C) way of representing a memory address.



- More specifically, a C variable can contain a pointer to something else. It actually stores the memory address that something else is stored at.

Pointers in C (#2/7)

- Why use pointers?
  - If we want to pass a huge struct or array, it's easier to pass a pointer than the whole thing.
  - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
  - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.



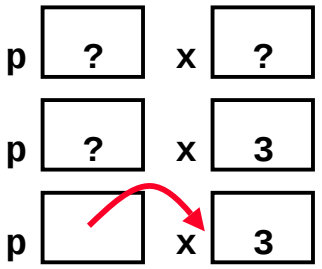
Pointers in C (#3/7)

- How to create a pointer:

  & operator: get address of a variable

```
int *p,
int x;
  x = 3;

  p = &x;
```



Note the "*" gets used 2 different ways in this example. In the declaration to indicate that **p** is going to be a pointer, and in the **printf** to get the value pointed to by **p**.

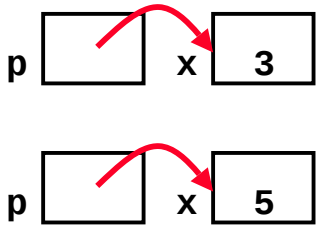
- How get a value pointed to?

  * "dereference operator": get value pointed to

```
printf("p points to %d\n",*p);
```

Pointers in C (#4/7)

- How to change a variable pointed to?
  - Use dereference * operator on left of =

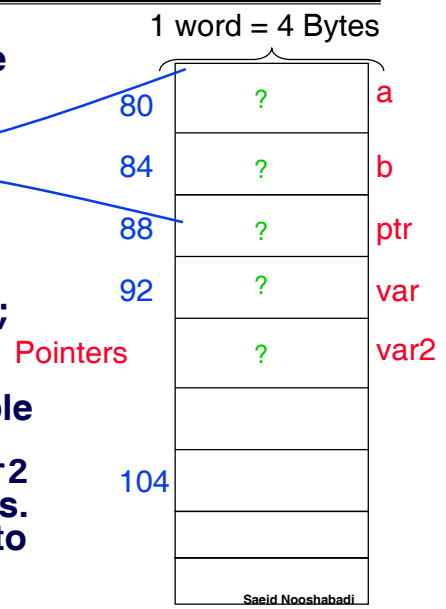


Pointers in C (#5/7)

- To declare a pointer, just precede the variable name with a *
- Examples:

```
int *a;
int  b;
int *ptr, var, var2;
```

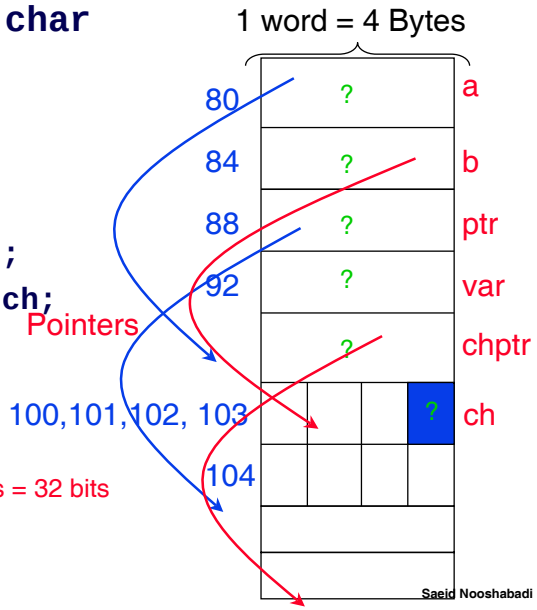
- Warning: In the third example above, the variable `ptr` is a pointer to an integer, while `var` and `var2` are actual integer variables. The asterisk only applies to *one* variable.



Pointers in C (#6/7)

- Mixing `int` and `char` pointers
- Examples:

```
int *a;
char *b;
int *ptr, var;
char *chptr, ch;
```



int Size: 1 word = 4 Bytes = 32 bits
char Size: 1 Byte = 8 bits

Pointers in C (#7/7)

- Notice that a pointer can only point to one type (`int`, `char`, a `struct`, etc.).
- An `int*` cannot point to a character, or vice versa.
- Why not?
  - Safety: Pointers are known to cause problems to careless programmers, so limit what a pointer can do.
- `void *` is a type that can point to anything (generic pointer)
  - Use sparingly to help avoid program bugs!

Arguments to Functions

- Arguments can be:
  - passed by value: Make a copy of the original argument (doesn't really affect types such as integers).

```
int p
 :
addOne (p)
```

  - passed by reference: Pass a pointer, so the called function makes modifications to the original struct.

```
int *p
 :
addOne (p)
```

  - Passing by reference can be dangerous, so be careful.

Pointers and Passing Arguments (#1/2)

- Java and C pass an "by value"
  - procedure/function gets a copy of the parameter, so changing the copy cannot change the original

```
void addOne (int x) {
   x = x + 1;
}
int y = 3;
addOne(y);
```

- y is still = 3

Pointers and Passing Arguments(#2/2)

- How to get a function to change a value?

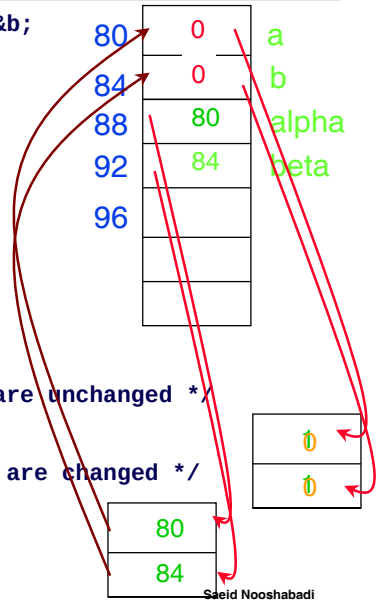
```
void addOne (int *p) {
  *p = *p + 1;
}
int y = 3;

addOne(&y);
```

- y is now = 4

Arguments to Functions: Example

```
int a =1, b=1 *alpha = &a, *beta = &b;
void point_less (int a, int b) {
  a = 0;
  b = 0;
}
void point_full (int *a, int *b) {
  *a = 0;
  *b = 0;
}
point_less(a,b);
/* After calling point_less, and b are unchanged */
point_full(alpha,beta);
/* After calling point_full,a and b are changed */
```



Peer Instruction Question

```
void main(); {
  int *p, x=5, y; // init
  y = *(p = &x) + 10;
  int z;
  flip-sign(p);
  printf("x=%d,y=%d,p=%d\n",x,y,p);
}
flip-sign(int *n){*n = -(*n)}
```

How many errors?

#Errors
1
2
3
4
5
6
7
8
9
(1)0

Peer Instruction Answer

```
void main(); {
  int *p, x=5, y; // init
  y = *(p = &x) + 10;
  int z;
  flip-sign(p);
  printf("x=%d,y=%d,p=%d\n",x,y,*p);
}
flip-sign(int *n){*n = -(*n);}
```

How many errors? I get 7.

#Errors
1
2
3
4
5
6
7
8
9
(1)0

References:

- Nick Parlante: Stanford CS Education Library (http://cslibrary.stanford.edu/); A Collection of very useful material including:
- Essential C: (http://cslibrary.stanford.edu/101/) A relatively quick, 45 page discussion of most of the practical aspects of programming in C.
- Binky Pointer Video: (http://cslibrary.stanford.edu//104) Silly but memorable 3 minute animated video demonstrating the basic structure, techniques, and pitfalls of using pointers.
- Pointer Basics: (http://cslibrary.stanford.edu/106) The companion text for the Binky video.
- Pointers and Memory: (http://cslibrary.stanford.edu/102) A 31 page explanation of everything you ever wanted to know about pointers and memory.
- Linked List Basics: (http://cslibrary.stanford.edu/103) A 26 page introduction to the techniques and code for building linked lists in C.

Things to Remember

- All declarations go at the beginning of each function.
- Only 0 and NULL evaluate to FALSE.
- All data is in memory. Each memory location has an address to use to refer to it and a value stored in it.
- A pointer is a High Level Language version of the address.
  - * "follows" a pointer to its value
  - & gets the address of a value