Advanced Micro-controllers

Agenda

- volatile

Introduction

- Nilesh Ghule
- M.Sc. Electronics
- DESD modules
  - Advanced Micro-controllers
  - Embedded OS
  - Linux Device Drivers

register - storage class

- "register" is same as "local" except the storage is in CPU registers.
- "register" characterstics:
  - Storage: CPU register
  - Default value: garbage
  - Scope: block
  - Life: block
- "register" keyword - requests compiler that the variable should be stored in CPU register.
  - If CPU register is not available, the variable will be treated like a local variable (on stack).
- "register" variables are processed faster, because their location (register) is directly accessed by ALU.
- Example:

```
register int i;
for(i=1; i<=100; i++) {
    printf("%d\n", i);
}
```

- Since "i" variable is accessed again and again, it is helpful to keep it in CPU register itself. Each access to the variable will be quick (because no need to load/store variable in RAM).

Compiler optimization

- Modern compilers optimize the code for faster execution.
- If a variable is accessed repeatedly (in loop), then compiler may load the variable from RAM into CPU register at the start of loop, use it repeatedly from the CPU register, and at the end of loop update the value back into RAM.
- This compiler behaviour can be configured with optimization level.
- Example:

```
int i;
for(i=1; i<=100; i++) {
    printf("%d\n", i);
}
```

```
cmd> gcc -O0 main.c
# optimization level = 0 i.e. no optimization
# i.e. for each access fetch variable from RAM into CPU register.
```

```
cmd> gcc -O3 main.c
# optimization level = 3 i.e. full optimization
# compiler fetch variable once into CPU register, use it repeatedly and finally store/update it back to the
RAM.
```

volatile

- volatile - suddenly change
- volatile keyword tells compiler that the variable may suddenly change out of current execution context. So always fetch the variable from its RAM location for each access. In other words, do not cache it in CPU register for a code block.
- Variable may get modified out of the current execution context
  - in another thread of execution.
  - in Interrupt handler or ISR.
  - in memory-mapped IO.
- Example

```
volatile int i;
```

```
// main() function
for(i=1; i<=100; i++) {
    printf("%d\n", i);
}
```

```
// other() function or isr()
i = 200;
```

Memory mapped IO

- Each IO device/peripheral is controlled by special function registers (SFR). They are also called as "IO registers".
- Each IO register is given some address (by manufacturer).
- Memory mapped IO means SFRs are accessed just like RAM addresses i.e. with same assembly language instructions.

```
# ARM assembly -- RAM location or IO SFR
# location = 0x2000
LDR r7, =0x2000
LDR r0, [r7]
ADD r0, #1
STR r0, [r7]
```

```
# ARM assembly -- RAM location or IO SFR
int *r7 = 0x2000;
int r0 = *r7;
r0++;
*r7 = r0;
```

const keyword

- Myths:
  - "const" variables cannot be modified.
  - "const" variables are stored in "rodata" section.
- Facts:
  - "const" keyword tells compiler that the variables is not intended to be modified.
  - Compiler doesn't allow to use any operator on the variable that may modify value of the variable. e.g. ++, --, =, +=, -=, &=, |=, etc.
  - Only string constants/literals "..." are stored in "rodata" section.
  - Other const variables are stored as per their storage class.
- Effects:

```
const int a = 10;
a++; // compiler error
```

```
const int a = 10;
a = 10; // compiler error
```

```
const int a = 10;
int *p = &a; // warning: suspious pointer conversion -- const int* to int*
*p = 20; // "a" is modified -- indirectly
printf("%d\n", a); // 20
```

static keyword

- "static" is same as "global" except the scope is limited (where it is declared).
- "static" characterstics:
    - Storage: Data section/BSS section
    - Default value: zero
    - Life: program
    - Scope: Limited to block or file (as per declaration)

static const volatile combination

- const volatile int a = 10;
    - const tells compiler that variable is not meant to be modified.
        - Do not allow any operators like ++, --, =, +=, etc -- in code.
    - volatile can be modified suddenly out of current execution context.
        - in ISR - C code
        - in another thread - C code
        - in MMIO - Electronics
    - const volatile
        - ~~in ISR - C code~~ <-- not permitted by compiler
        - ~~in another thread - C code~~ <-- not permitted by compiler

        - in MMIO - Electronics
- Typically "const volatile" represents a memory-mapped read-only register.
- static const volatile int a = 10;
    - static const volatile
        - ~~in ISR - C code~~ <-- not permitted by compiler
        - ~~in another thread - C code~~ <-- not permitted by compiler
        - in MMIO - Electronics
        - Scope is limited to file or function in which it is declared.
- Typically "static const volatile" represents a memory-mapped read-only register that is meant to be accessed only in a file/function.

Program vs Process

- Source code

```
// demo01.c
int num1 = 10;
int num4;
int main() {
    static int num2 = 20;
    int num3 = 20;
    printf("%d, %d, %d\n", num1, num2, num3);
    return 0;
}
```

```
> gcc -o demo01.out demo01.c

> objdump -t demo01.out
# displays symbol table

> objdump -S demo01.out
# displays machine level/assembly code
```

```
> objdump -h demo01.out
# displays exe header
```

- Instruction (assembly/machine level)
  - Instruction = Op code + Operands
    - Op code -- e.g. add, mov, sub, ...
    - Operands -- e.g. registers, addresses/values, ...
- Program: Program is a set of instructions given to the computer/machine.
  - Program is an executable file.
  - Executable file is a sectioned binary file. It contains following sections:
    - exe header
      - magic number (ELF --> 0x7f454c46 (?ELF), PE --> 0x4d5a (MZ))
      - address of entry point function (_start)
      - information of all sections
    - text - machine level instructions
    - data - initialized global and static variables
    - bss - uninitialized global and static variables
    - rodata - string constants
    - symbol table - information about symbols (variables and functions)
      - name, address, section, size, flags.

rodata section

- Example:

```
// demo02.c
int main() {
    char *p = "SUN";
    *p = 'F'; // segmentation fault
    puts(p);
    return 0;
}
```

```
> gcc -g -o demo02.out demo02.c
# output: Segmentation Fault

> valgrind ./demo02.out
```

Data section vs BSS section

- Example:

```
// demo03.c
int arr1[1024] = {1, 0, 0, 0, 0};
int arr2[1024];
int main() {
    return 0;
}
```

```
> gcc -o demo03.out demo03.c

> ls -l demo03.*

> objdump -t demo03.out
```

- Data section
  - Space is reserved for the variable in executable file and this space stores given initial values.
  - Data section from file is copied into the data section in RAM (for the process).

- BSS section - Block Started by Symbol
  - Space is not reserved for the variable. Only declaration is done that the variable is to be allocated at runtime.
  - All bytes in BSS section are zero filled.
- Both sections will get memory in RAM at runtime.
- Data section and BSS section are initialized to initial values and zeros (respectively) by the startup routine (i.e. entry point function).