Homework 1 Report  
 Disassembling the machine code for GPIO pin toggle program.

Submitted by: Kripa Varma  
Date : 4/17/2015

# Summary

Files committed to git : hw1.c, hw1.h, types.h, makefile, hw1.elf, hw1\_dmp.txt, Homework1\_Report.docx

* Installed **Linaro** arm-none-eabi toolchain for Windows.
* Installed **github** and **gitshell** on windows.
* Wrote a **C program** that toggles the GPIO Port0 Pin#6.
* Wrote a **makefile** and compiled the program.
* Obtained **objectdump** using the command **arm-none-eabi-objdump –S hw1.elf >hw1\_dmp.txt.**
* Tested the program on board by toggling Port1.Bit18 that controls LED0.

**Main Focus** : Detailed study of translation from   
 C statements -> Assembly -> Machine language encoding.

# Learned

1. **GPIO Register programming**:

* Studied the LPC1768 user manual to understand the **function of GPIO Direction, Set, Clear, Pin registers (**0x2009C000 to 0x2009C09E)**.**
* There are 70 GPIO pins.
* There are **five FIODIR** registers each bit defines whether corresponding GPIO pin is input/output (0/1). Some bits are reserved.
* Similarly there are five each of FIOSET, FIOCLR,FIOPIN registers for setting clearing and reading status of GPIO pins.
* There are **2 bits in PINSELn** register corresponding to each GPIO pin, which defines the role of that particular pin. If the 2 bits are **set to Zero** then the pin functions as a GPIO pin controllable **by GPIO registers at 0x2009 C000 - 0x2009 FFFF.**

1. **Studied briefly the elf file objectdump, and program sections**
   * Each section header is associated with
     + Start Address,
     + VMA (Virtual Memory Address),
     + LMA (Load /Physical Memory Address)
     + flag LOAD(whether section is loaded into memory)
     + flag ALLOC (Whether memory is allocated for the section)
   * .text section – is the executable code section & READ ONLY
   * .data section – contains the initialized global and local static variables & is READ\_WRITE
   * .bss section – contains the uninitialized global and local static variables & is READ\_WRITE
   * .heap is of size 0x00000800 and starts at 0x100003a0
     + Space for Dynamically allocated memory
     + Starts at
   * .stack\_dummy is of size 0x00000c00 and starts at 0x100003a0
     + Local variables and arguments are stored in stack
     + It starts at 0x100003a0 + 0xc00 and works backward.
     + In this case max stack size is (0xc00 – 0x800 = 0x400), beyond which stack might overflow and could cross over to possibly malloc-ed memory space.
2. **ARM Assembly and Machine Language Encoding:**

* Learned the syntax and studied machine language encoding coding of ARM 32 bit instructions **LDR,STR,MOV,MVN,ADD,SUB,ORR,BIC,B**.
* -mthumb compile flag that tells compiler to use 16 bit thumb instructions was not used during this homework, so all instructions decoded below are 32 bit.
* Studied the various **addressing modes used in Load/Store**

Load/Store memory address = [BaseReg] +/- Offset

Offset can be one of these:-

1) 12 bit or 8 bit **Immediate value**

2) **Register offset**

3) **Scaled or shifted register offset**

Each offset type can be used in one of 3 ways:-

1. **Offset addressing** - The memory address is generated by applying the offset to the base register value. Base register remains unchanged
2. **PreIndexed**  - Similar to offset ddressing, but calculated m/y addr is written back to Base Register
3. **PostIndexed** - base reg alone is used to compute m/y address. Offset is then applied to Base Reg and written back to Base Reg

* Studied the various **shifter\_operand addressing modes** for Data Processing/Arithmetic Instructions.

1. Immediate operand value – 8-bit operand that is rotated right 2^n times
2. Register operand value
3. Register operand with Logical Left Shift by Immediate value/Register
4. Register operand with Logical Right Shift by Immediate value/Register
5. Register operand with Arithmetic Left Shift by Immediate value/Register
6. Register operand with Arithmetic Right Shift by Immediate value/Register

* Refreshed my memory on **2’s complement arithmetic,** and representation of negative values using 2’s complement.
* Learned the **encoding** of signed\_immed\_24 operand for **branch instruction**.

1. First base address is computed as : BA = (Addr of branch instruction + 8)

For eg: in the while loop below , TA = 0x82a8 + 8 = 0x82b0

1. Then base addr is subtracted from target address to get byte offset., which is a multiple 4 bytes ( in non thumb case)

ByteOffset = FFFFFFD8 (-0x28)

1. signed\_immed\_24 = ByteOffset[25:2 ] = FFFFF6
2. Learned the **use of git** commands add, rm, commit, push.

# The disassembly of m/c code in while loop

while (1)  
 {  
 gpio\_set0->pins\_t.pin6 = 1;

8288: e51b2010 ldr r2, [fp, #-16]

828c: e5d23000 ldrb r3, [r2]

8290: e3833040 orr r3, r3, #64 ; 0x40

8294: e5c23000 strb r3, [r2]

gpio\_set0->pins\_t.pin6 = 0;

8298: e51b2010 ldr r2, [fp, #-16]

829c: e5d23000 ldrb r3, [r2]

82a0: e3c33040 bic r3, r3, #64 ; 0x40

82a4: e5c23000 strb r3, [r2]

}

82a8: eafffff6 b 8288 <main+0x6c>

* The table below, breaks down the 4 byte machine instructions obtained from objdump, identifies the <opcode> <Rd>,<Rn> and the operands.
* Most Signifcant nibble ie., the <cond> field of all instructions is 0xE ( Always execute), which means unconditional execute.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Inst. #** | **addr** | **m/c code word** | **assembly** | **<opcode>** | **<Rd>**  **Dest Reg** | **<Rn>**  **Src Reg** | **Other operands** |
| **Start of while loop :** | | | | | | | |
| 1 | 8288 | e51b2010 | ldr r2, [fp, #-16]  [[1]](#load_store) | <opcode>=0x51  Load r2 word  subtract imm offset  r3<-addr[fp-16] | r2(0x2) | fp(0xb) | offset\_12 = 0x10 |
| 2 | 828c | e5d23000 | ldrb r3, [r2]  [[1]](#load_store) | <opcode>=0x5d  Load byte add imm offset  r3<-byte at addr[r2] | r3(0x3) | r2(0x2) | offset\_12 = 0x0 |
| 3 | 8290 | e3833040 | orr r3, r3, #64  [[2]](#orr) | <opcode>=0x38  OR immediate  r3<-r3|0x40 | r3(0x3) | r3(0x3) | imm\_8= x40  rotate\_imm=0 |
| 4 | 8294 | e5c23000 | strb r3, [r2]  [[1]](#load_store) | <opcode>=0x5c  Store byte  add imm offset  r3->byte at addr[r2] | r3(0x3) | r2(0x2) | offset\_12 = 0x0 |
| 5 | 8298 | e51b2010 | ldr r2, [fp, #-16] | Same as #1 | r2(0x2) | fp(0xb) | offset\_12 = 0x10 |
| 6 | 829c | e5d23000 | ldrb r3, [r2] | Same as #2 | r3(0x3) | r2(0x2) | offset\_12 = 0x0 |
| 7 | 82a0 | e3c33040 | bic r3, r3, #64  [[3]](#bic) | <opcode>=0x3c  Bit Clear  r3<-r3&!0x40 | r3(0x3) | r3(0x3) | imm\_8= x40  rotate\_imm=0 |
| 8 | 82a4 | e5c23000 | strb r3, [r2] | Same as #4 | r3(0x3) | r2(0x2) | offset\_12 = 0x0 |
| 9 | 82a8 | eafffff6 | b 8288  [[4]](#branch) | Unconditional branch, Do not store RA | 2s complement offset operand: fffff6  Sign extending to 30 bits : 3ffffff6  Shifting left by 2 to align to word:ffffffd8  Add to PC, which is 82b0 now : 8288 | | |

# Generic syntax of instructions used in the program

**[1] LOAD/STORE Immediate offset Encoding**

LDR|STR{<cond>} Rd, [<Rn>, #+/-<offset\_12>]



P (0/1) : post/ pre indexed( when W=1) or offset(when W=0)

U (0/1) : Offset Sub/Add

B (0/1) : Word/Byte access

W(0/1): BaseReg unchanged/Addr written back to BaseReg

L (0/1): Load/Store

**[2] ORR Syntax: (Logical OR)**

Rd := Rn OR shifter\_operand



I (0/1) : Register/Immediate shifter operand

S (0/1) : CPSR Unchanged/ CPSR Update after execution

**[3] BIC Syntax: ( Bit Clear)**

Rd := Rn AND NOT(shifter\_operand)



**[4] B (Branch)**

B{L}{<cond>} <target\_address>



L (0/1) : Store / Do not Store return address

<target\_address>:

Specifies the address to branch to. The branch target address is calculated by:

1. Sign-extending the 24-bit signed (two's complement) immediate to 30 bits.

2. Shifting the result left two bits to form a 32-bit value.

3. Adding this to the contents of the PC, which contains the address of the branch

instruction plus 8 bytes.

# Hours:

4 hours : GPIO related reading, coding, makefile.  
4-5 hrs : Study instructions encoding, Assembly syntax.  
4-5 hrs : Creating the report.

# Issues faced:

* The compilation initially gave error : “[undefined reference to `\_exit' when using arm-none-eabi-gcc](http://stackoverflow.com/questions/19419782/exit-c-text0x18-undefined-reference-to-exit-when-using-arm-none-eabi-gcc)“

Google search revealed that when retargeting/crosscompiling on machine other than target the linker **flag --specs=nosys.specs** is to be used to avoid such an error.

* For lack of access to other equipments, the program could be tested only by toggling **Port1.pin18** instead that controls **LED0**. To verify that toggling is actually taking place, for-loop delays and wait\_ms() calls had to be added between toggling.

# References:

1. UM10360 LPC176x/5x User manual ; Section 9.5 – GPIO Registers
2. ARM Architecture Reference Manual ; Section A4, A5