E1.9/E2.19 Assembler Programming Coursework

T. J. W. Clarke, v1.00

December 4, 2011

Equipment

This assignment requires the Keil ARM development tools. The demo version can be downloaded and installed on any windows PC. See course web pages for further information.

1 Introduction

Assembler programming is now seldom used. Better compilers and faster computers mean any speed gain is usually irrelevant. The higher productivity of high level languages makes them preferable.

However there are some niche areas where assembly code is needed, typically in high speed hardware interface, or specialised use of interrupts. This assignment uses both.

The problem is similar to that in the ARM EE2 lab experiment: the number of pulses on a digital input line within a given fixed time must be counted, in order to estimate frequency. To make the problem more challenging we require 4 inputs in parallel to be counted in the same period.

There are many different ways to approach this problem. The assessment will be based on correctness, performance and your ability to analyse your code.

```
volatile int finished;

int count_cycles()
{
   finished = 0;
   // set up interrupt after T ms
   slave_portpin = 0
   while (~finished) {
       if ( portpin() != slave_portpin) {
          count = count + 1
          slave_portpin = portpin()
   }
   return count;
}

interrupt_subroutine()
{
   finished = 1;
}
```

Figure 1: Pseudocode to measure frequency

2 Basic Algorithm

Figure 1 shows the algorithm you will use to measure frequency on a single input pin. This can easily be adapted to measure frequencies on more than one pin in parallel. For this assignment you must measure the frequencies on 4 pins in parallel. The digital value of all 4 pins: p0, p1, p2, p3, can be read by reading a single (32 bit) memory location as shown in Figure 2. Note that `count_cycles` is written here as a subroutine but will be coded in this assignment as non-subroutine code.

Bit numbers in 32 bit word labelled IOPIN

31	24	23	16	15	8	7	0
	p3		p2		p1		p0

NB All other bits in the word are undefined, and may not be 0.

Figure 2: Input signals

3 Implementation and Testing

You are given a Keil project with skeleton code you can use to test your answer. The skeleton code must not be changed except where indicated. The Keil project includes debug code (debug file `io.ini`) that simulates hardware input, and generates an ARM IRQ interrupt after a known number of cycles, see Appendix C for specifics.

Your code needs to implement, in assembler:

- The counting loop, as in Figure 1, in code from **START** onwards.
- Interrupt Service Routine (ISR) code, from **ISR_FUNC** onwards.
- Code which writes the output into specified memory locations at the end of your counting loop.

The ISR will be called when your counting loop code is active, so you can partition registers between the counting loop and ISR yourself.

The output from your counting loop, when, after the ISR has been called to interrupt counting, execution reaches label **LOOP_END** in the assembly code, must be 4 words **P0COUNT**, **P1COUNT**, **P2COUNT**, **P3COUNT** as defined in the skeleton assembly file. These words must each contain the number of cycles measured on the given input pin in Figure 2.

Your counting loop code must terminate, as in Figure 1, when the IRQ interrupt happens. You can add whatever code you like after the main counting loop to make the **PNCOUNT** output words correct, before branching to **LOOP_END** to terminate execution.

The skeleton code you are given includes a Keil debug module (`io.ini`) which provides simulated waveforms on 4 bits of the (single) 32 bit input port, and a monitor which compares the resulting counts with what is expected and outputs the error for each count. The length of the test, and the simulated frequencies of each port input, can be controlled through constants defined in your assembly code.

Code will be considered correct if it is never more than 1%, or 1 cycle, in error, which ever is greater.

The Keil simulator includes a **Logic Analyser** which displays waveforms of the simulation. This has been set up to record the 4 input signals and the signal which generates the IRQ. Refer to Appendix D for details of how to use this and other Keil debug functionality.

4 Applications

The code fragment you will test is artificial but with small changes could solve real-world problems. Written in assembler the solution will be between 2 and 20 times faster than written in C. In a real application the interrupt would be periodic, derived from a hardware timer, and contain code to analyse and process the counted values. Probably the real code would be written as an assembly language fragment inside a larger C or C++ program. The LPC2103 and related NXP ARM7 microcontrollers have clock frequencies of 60MHz. Later ARM processors are faster, e.g. the LPC1768 (used in the ARM mbed module, mbed.org) which has a 120MHz clock. The performance scales with the clock frequency of the processor.

When programming microcontrollers the most difficult code is the startup code that initialises the system and sets up interrupts, etc. The file you are given has a simplified version of this code.

- You can for example trace through the code from reset which sets up the I/O pin on bit 15 of the port to trigger an interrupt.
- You can trace the code from the ARM ISR vector which clears this interrupt source (so that the interrupt is no longer present on return from the ISR).
- The counting code is run in supervisor mode, with the interrupt in IRQ mode. Where are the CPSR bits set so that in supervisor mode the IRQ interrupt is enabled.

5 Code Submission and Assessment

Your answer to this assignment must be a single assembly file, normally the skeleton file `countfreq.s` which you are given with appropriate modifications. This will be assessed using test code like `io.ini`, but with waveforms of different frequency to determine whether your code works and what are the maximum frequencies it can detect correctly.

Include, in comments at the start of your assembler file, answers to the following questions:

- What is the maximum frequency you can reliably measure, given a 100MHz clock?
- How does the expected fractional error from your algorithm change as the simulation time increases?

Appendix A - Advanced Coding Ideas

Here are some optional ideas for clever solutions to this problem. They all involve more assembly language than a basic solution, so don't attempt them unless you have time to spare. The ideas are not fully worked through, they are meant to provide you with ideas for further thought.

- ARM bitwise logical instructions can process all inputs in parallel.
- 32 bits additions can be used to perform 4 8 bit additions in parallel, as long as overflow is detected and processed.
- Inner loops can be unrolled, and extra instructions inserted at each iteration which work in parallel with the unrolled inner loop. This allows multiple instructions to be written processing the output of the inner loop computation while only increasing the inner loop time by one instruction.
- The overhead of a branch instruction can be made less than 1% of the overall by unrolling inner loops sufficiently. This is difficult to do by hand, but the ARM MASM *macro assembler* provides features for compactly writing repetitive sections of code. See Keil documentation. Note that your code is limited to a maximum size of below 0x7f00 bytes, since address 0x7f00 is used by the test code. This will not normally be a problem!
- The LPC2103 datasheet (see web) details **FIOPIN**, a location which can access the GPIO port inputs faster than **IOPIN** as is used in the skeleton code.

Appendix B - Assessment Criteria

Figure 3 shows the marking scheme for this assignment.

Assessment	Details	marks
Correctness	Does the code find the correct frequencies? Error of up to 1% is allowed.	50
Code Style	Does the code where appropriate use the optimisations discussed in the lectures? Is it a time-efficient implementation? Is the operation of the code correctly analysed?	20
Performance	What is the maximum frequency that can be detected?	20
Innovation	Is there anything especially innovative and useful in the implementation?	10

Figure 3: Assessment

Appendix C - Keil Assembly Project Guide

The demo version of the Keil compiler and simulator can be installed free on any windows PC. The code limit of 32K is much larger than needed. The zip file includes a complete Keil project which can be started by double-clicking on the `testassem.uvproj` project file.The project will allow you to assemble and test the `countfreq.x` fassembly file. This is downloaded as a skeleton, which you may change. In order to ensure that the provided debug code works correctly please do not change anything except:

- The code section which you write containing the counting code and the ISR. This section is mapped to device flash ROM and therefore is read only.
- The parameters section controlling the simulation: you may change parameter values
- The data section, in case you need to define additional read/write data.

Your counting loop code (from the **START** label) must branch to the **LOOPEND** label after it has completed. Code you are given will then terminate the simulation and display results. Your ISR must start at label **ISR_FUNC** and correctly return to the main loop with a return from interrupt, after changing a register or memory location which will cause the counting loop to terminate.

Appendix D - Debugging

Assembly language is notoriously difficult to debug. Compared with high level languages an equivalent program can take up to ten times as long. Careful debug strategy is important.

Keil Specific Debug Notes

The Keil assembler will provide errors when code is malformed. The linker will also provide certain errors. You must expect a warning error:

`L6314W: No section matches pattern ...`

This error should be ignored.

However the fatal error message:

`testassem.axf: Error: L6286E:`

indicates that you have used **LDR**, **STR** or **ADR** with a label that is out of range. This is easy to do because the DATA section is located a long way away from the CODE section, and the CODE section is read-only and therefore cannot be used for memory variables. The fix is to define a memory word in the code section to contain the out of range address, load this into a register with **LDR** and then use another memory reference instruction to access the data. You can find an example of this in the given code where **Addr_EXTMODE**, defined in a **DCD** statement, is first loaded into a register and then used to write to the **EXTMODE** I/O register.

You can debug this error by temporarily removing all LDR,STR, ADR statements from your code until you discover which is causing the problem. Because of this error, it is strongly advised that you write your code incrementally checking that the code assembles correctly at frequent intervals.

Debug Strategy

- Write and test (by single-stepping, or running to a breakpoint) small sections of code at a time. You should assume your code will contain many mistakes and verify each instruction. This is not very difficult if you use the Keil debug features described in the next section.
- Get the simplest possible code working first before attempting anything ambitious.
- The ISR must correctly return to the counting loop after it is executed. Single-step carefully through the return sequence to ensure this.
- Use registers rather than memory locations for variables where this is possible: it will be simpler and easier to debug.

Using Keil Instrumentation

The Debug→Start/Stop command controls the simulation. While simulating the various Keil debug tools below can be setup and used. Between simulations most debug tool settings are remembered. See the Keil help pages Help→Uvision Help→Uvision IDE.

- The simulator allows single-step, or run to end. Running can be stopped at a specified instruction by setting a breakpoint.
- The register window allows you to check intermediate results as you step through the simulation.
- The *memory watch windows* (View→ Watch) can be set up to view any memory address, and mark memory locations which have been read (without writing) or written as respectively red or green. Normally a memory location should never be read before it is written but there is an exception if the location is an I/O register.
- The *Logic Analyser* (View→Analysis Tools→ Logic Analyser) provides a simulated oscilloscope view of specified signals. It has been set up for the 4 Port bits you will count, and the port bit which generates the inerrupt signal to end the simulation. These 5 signals need to be displayed in *bit* mode, right-click on *PORT* to the left of each waveform and select bit.

Appendix E - Skeleton Assembly Code

```
; Standard definitions of Mode bits and Interrupt (I & F) flags in PSRs

Mode_USR        EQU     0x10
Mode_FIQ        EQU     0x11
Mode_IRQ        EQU     0x12
Mode_SVC        EQU     0x13
Mode_ABT        EQU     0x17
Mode_UND        EQU     0x1B
Mode_SYS        EQU     0x1F

I_Bit           EQU     0x80            ; when I bit is set, IRQ is disabled
F_Bit           EQU     0x40            ; when F bit is set, FIQ is disabled


;// <h> Stack Configuration (Stack Sizes in Bytes)
;//   <o0> Undefined Mode      <0x0-0xFFFFFFFF:8>
;//   <o1> Supervisor Mode     <0x0-0xFFFFFFFF:8>
;//   <o2> Abort Mode          <0x0-0xFFFFFFFF:8>
;//   <o3> Fast Interrupt Mode <0x0-0xFFFFFFFF:8>
;//   <o4> Interrupt Mode      <0x0-0xFFFFFFFF:8>
;//   <o5> User/System Mode    <0x0-0xFFFFFFFF:8>
;// </h>

UND_Stack_Size  EQU     0x00000000
SVC_Stack_Size  EQU     0x00000080
ABT_Stack_Size  EQU     0x00000000
FIQ_Stack_Size  EQU     0x00000000
IRQ_Stack_Size  EQU     0x00000080
USR_Stack_Size  EQU     0x00000000

ISR_Stack_Size  EQU     (UND_Stack_Size + SVC_Stack_Size + ABT_Stack_Size + \
                         FIQ_Stack_Size + IRQ_Stack_Size)

        AREA     RESET, CODE
ENTRY
;  Dummy Handlers are implemented as infinite loops which can be modified.

Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP                             ; Reserved Vector
                LDR     PC, IRQ_Addr
;               LDR     PC, [PC, #-0x0FF0]      ; Vector from VicVectAddr
                LDR     PC, FIQ_Addr

ACBASE DCD P0COUNT
SCONTR DCD SIMCONTROL

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0                       ; Reserved Address
FIQ_Addr        DCD     FIQ_Handler

Undef_Handler   B       Undef_Handler
SWI_Handler     B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
FIQ_Handler     B       FIQ_Handler


AREA  ARMuser, CODE,READONLY

IRQ_Addr        DCD     ISR_FUNC1
EINT2 EQU  16
Addr_VicIntEn DCD 0xFFFFF010   ; set to (1<<EINT0)
Addr_EXTMODE DCD  0xE01FC148    ; set to 1
Addr_PINSEL0 DCD 0xE002C000 ; set to 2_1100
Addr_EXTINT DCD 0xE01FC140

;  addresses of two registers that allow fast input

Addr_IOPIN DCD 0xE0028000


; Initialise Interrupt System
;  ...
ISR_FUNC1 STMED R13!, {R0,R1}
MOV  R0, #(1 << 2)     ; bit 2 of EXTINT
LDR  R1,Addr_EXTINT
STR R0, [R1]    ; EINT2 reset interrupt
LDMED R13!, {R0,R1}
B  ISR_FUNC

Reset_Handler
; PORT0.1 1->0 triggers EINT0 IRQ interrupt
MOV R0, #(1 << EINT2)
LDR R1, Addr_VicIntEn
STR R0, [R1]
MOV R0, #(1 << 30)
LDR R1, Addr_PINSEL0
STR R0, [R1]
MOV R0, #(1 << 2)
LDR R1, Addr_EXTMODE
STR R0, [R1]

;  Setup Stack for each mode

                LDR     R0, =Stack_Top

;  Enter Undefined Instruction Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_UND:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #UND_Stack_Size

;  Enter Abort Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_ABT:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #ABT_Stack_Size

;  Enter FIQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_FIQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #FIQ_Stack_Size

;  Enter IRQ Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_IRQ:OR:I_Bit:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #IRQ_Stack_Size

;  Enter Supervisor Mode and set its Stack Pointer
                MSR     CPSR_c, #Mode_SVC:OR:F_Bit
                MOV     SP, R0
                SUB     R0, R0, #SVC_Stack_Size
B  START
;----------------------------DO NOT CHANGE ABOVE THIS COMMENT-----------------------------
;-----------------------------------------------------------------------------------------
;-----------------------------------------------------------------------------------------
; constant data used in counting loop

START     ; initialse  counting loop

; main couting loop loops forever, interrupted at end of simulation
LOOP
B  LOOP ; For skeleton code only, replace with counting loop which
; branches to LOOP_END on termination of loop



ISR_FUNC ; Interrupt must set variable to terminate main loop
B  LOOP_END ; for skeleton code only
; replace this by return from interrupt
; branch to LOOP_END will be at end of LOOP code

;-----------------------------------------------------------------------------------------
; PARAMETERS TO CONTROL SIMULATION, VALUES MAY BE CHANGED TO IMPLEMENT DIFFERENT TESTS
;-----------------------------------------------------------------------------------------
SIMCONTROL
SIM_TIME  DCD   10000   ; length of simulation in cycles (100MHz clock)
P0_PERIOD DCD    40        ; bit 0 input period in cycles
P1_PERIOD DCD    34   ; bit 8 input period in cycles
P2_PERIOD DCD   44   ; bit 16 input period in cycles
P3_PERIOD DCD 38   ; bit 24 input period in cycles
;--------------------DO NOT CHANGE AFTER THIS COMMENT-----------------------------------
;-----------------------------------------------------------------------------------------
;-----------------------------------------------------------------------------------------
LOOP_END MOV R0, #0x7f00
LDR R0, [R0]  ; read memory location 7f00 to stop simulation
STOP B  STOP
;-----------------------------------------------------------------------------
  AREA DATA, READWRITE

P0COUNT DCD 0
P1COUNT DCD 0
P2COUNT DCD 0
P3COUNT DCD 0
;-----------------------------------------------------------------------------
                AREA    STACK, NOINIT, READWRITE, ALIGN=3

Stack_Mem       SPACE   USR_Stack_Size
__initial_sp    SPACE   ISR_Stack_Size

Stack_Top


         END                    ; Mark end of file
```