ECE 353 Introduction to Microprocessor Systems

Week 5

Eric Hoffman

Topics

- ARM7TDMI Programming
  - LUTs
  - Branches and Loops
- Structured Programming
  - Flowcharts
- Stacks
  - Hardware versus Memory Stacks
  - ARM7TDMI Stack Management

Look-Up Tables

- In an embedded system, the code area is usually in nonvolatile memory (Flash)(ROM)
- Data in the code area will also be placed in ROM – so the data is constant
- Look-up tables (LUTs) are used to translate a value from one domain to another when...
  - There is no convenient mathematical relationship between the two domains
  - The calculations are too expensive for the application constraints

Dew Point From RH & Temp

LUT Examples

- Exercises: Write code fragments that...
  - 1. Use a lookup table to get a 32-bit mask that will be used to clear individual bytes as directed by a 4-bit value (i.e. 2_1001 means that we want to clear the middle two bytes and preserve the most/least significant bytes)
  - 2. Use a lookup table to get the square root of an unsigned half-word value
    - Disassembly fragment
  - In both cases,
    - Assume that the value to convert is in R0, result in R1
    - Use addressing modes efficiently
    - Identify how large the complete data table will be

Branches

- Branches redirect program flow by loading a new PC value
  - In many processors, a branch that does not return is called a *jump*
- ARM7TDMI branch instructions
  - **B *target_address***
    - Target_address is a label
    - Instruction encodes a signed 24-bit offset
  - **BX <Rm>**
    - Branch address is in a register
    - Can be used to jump to Thumb code
- Jump tables
  - Be sure index is bounds-checked!

Signed number, shifted left "promoted" 2-bits

Is there a similar control structure to a jump table In HLL?

Conditional Branches and Looping

- Conditional branches allow program flow to change in response to conditions
  - Action is based on current state of flags
  - Prior instruction must be used to set flags
- Can use backwards conditional jumps to create a loop that can be terminated -
  - By a condition
  - After a predetermined number of iterations
  - Or a combination of both
- Looping examples
  - Incrementing vs Decrementing loops
  - While (or for) vs Do While Loops

Implementing Structured Programming Constructs

- Structured programming basics
  - One entry point, one exit point per subroutine
  - Based on three basic control structures
    - Sequence
    - Selection
      - If, If/Else, Switch/Case
    - Repetition
      - While
      - Do-While
- Flowchart Basics

Stack Implementation

- The stack is a LIFO data structure
- What is the stack used for?
- Two basic stack operations
  - PUSH
  - POP (aka PULL)
- Hardware stacks vs. memory stacks
  - Hardware stack
  - Memory stack
    - Stack pointer (SP)
    - Stack topologies and SP operation

ARM7TDMI Stack Operation

- By convention, the stack pointer is R13
- ARM stack terminology
  - Empty versus full
  - Ascending versus descending
- Allocating stack space
  - SP initialization
- Stack operations
  - LDR/STR
  - LDM/STM
  - PUSH/POP
- ARM operating modes and stacks

Assumes a FD stack

Wrapping Up

- Homework #3 is due Wednesday, 10/15
- Quiz #1 will be held Thursday, 10/8 at 7:15pm in room 2317 EH
  - Coverage will be over modules 1 and 2.
  - Calculators are not permitted. You may have a 3x5 card with handwritten notes.
  - The instruction set documentation will be provided.
  - If you have a conflict, please send me the details by email.

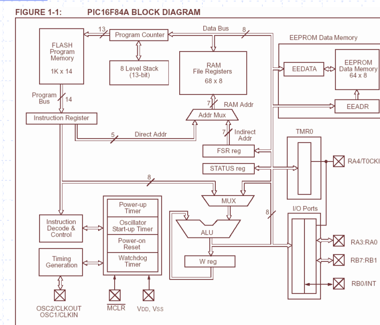
```
UND_Stack_Size  EQU     0x00000000
SVC_Stack_Size  EQU     0x00000008
ABT_Stack_Size  EQU     0x00000000
FIQ_Stack_Size  EQU     0x00000040
IRQ_Stack_Size  EQU     0x00000040
USR_Stack_Size  EQU     0x00000400

Stack_Size      EQU     (UND_Stack_Size + SVC_Stack_Size + \
                         ABT_Stack_Size + FIQ_Stack_Size + \
                         IRQ_Stack_Size + USR_Stack_Size)

                AREA    SRAM, NOINIT, READWRITE, ALIGN=3
Stack_Mem       SPACE   Stack_Size

; 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
```

PIC16F84 Hardware Stack



Jump Tables

```
start
        MOV  R0, ???    ; assume index is in R0

        MOV  R1, jmptbl ; get base address

        LDR  R0, [R1,R0,LSL #2] ; do look-up
        BX   R0         ; branch to target
;
task0                   ; stub
        NOP
        B    start
;
task1                   ; stub
        NOP
        B    start
;
jmptbl DCD task0, task1; ...
```

B Instruction Reference

- Syntax
  - B{<cond>} <target_address>
- RTL
  - if (cond is true)
    - PC ← PC + offset
- Flags are not affected

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	0	signed_immediate_24																							

BL Instruction Reference

- Syntax
  - BL{<cond>} <target_address>
- RTL
  - if (cond is true)
    - R14 ← next instruction address
    - PC ← PC + offset
- Flags are not affected

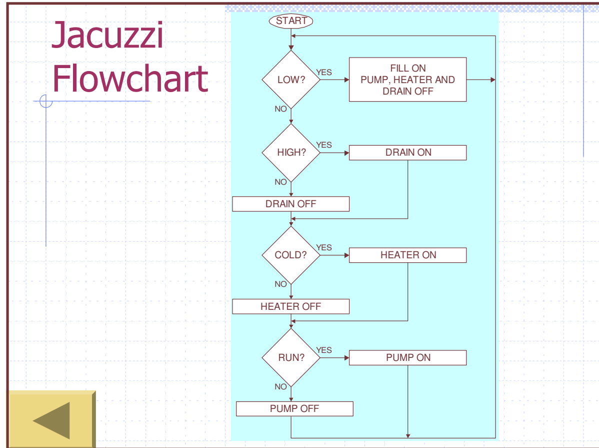
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				1	0	1	1	signed_immediate_24																							

BX Instruction Reference

- Syntax
  - BX{<cond>} <Rm>
- RTL
  - if (cond is true)
    - T flag ← Rm[0]
    - PC ← Rm & 0xFFFFFFFE
- Flags are not affected
- In ARM v5 and higher, there is also a BLX instruction

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cond				0	0	0	1	0	0	1	0	SBO				SBO				SBO				0	0	0	1	Rm			

Jacuzzi Flowchart



Disassembly Fragment

```
   57:         ADR      R2, sqrt_LUT
0x00080104  E24F2024  SUB       R2,PC,#0x00000024
   58:         LDR      R1, sqrt_MASK
0x00080108  E51F1020  LDR       R1,[PC,#-0x0020]
```

```
            AREA    FLASH, CODE, READONLY

mask_LUT    DCD 0x00000000, 0x000000FF, 0x0000FF00, 0x0000FFFF
            DCD 0x00FF0000, 0x00FF00FF, 0x00FFFF00, 0x00FFFFFF
            DCD 0xFF000000, 0xFF0000FF, 0xFF00FF00, 0xFF00FFFF
            DCD 0xFFFF0000, 0xFFFF00FF, 0xFFFFFF00, 0xFFFFFFFF
sqrt_LUT    DCB 0, 1, 1, 2, 2                       ;would need 65536 entries
; assembler adds 3 bytes of padding here
sqrt_MASK   DCD         0x0000FFFF

__main
            MOV         R0, #3                      ; assume index is in R0

            ADR         R2, mask_LUT                ; get base address
            AND         R0, #0x0F                   ; mask index
            LDR         R1, [R2,R0,LSL #2]          ; do look-up

            ADR         R2, sqrt_LUT                ; get base address
            LDR         R1, sqrt_MASK               ; appears to be direct addressing
            AND         R0, R1                      ; mask index
            LDRB        R1, [R2,R0]                 ; do look-up
```

Looping Example

HLL:

```
unsigned long data[100];
unsigned long num_elem;
      .
      .
      .
for (i=0; i<num_elem; i++) {
   data[i+1]=data[i];
}
```

ARM Assembly:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; R0 pointer to element 0 of data
;; R1 used as our index for the loop (ie. i)
;; R2 stores num_elem (# of elements)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                MOV     R1, #0
insrt_loop      LDR     R3, [R0,R1 LSL #2]
                ADD     R1, R1, #1
                CMP     R1, R2
                BLS     insrt_loop
```

Are these really the same loop structure?

No, the assembly is really a do..while loop, not a for loop.