Lookup Tables

Example of Arbitrary Waveform Generator

- Clock Generator Output Increments a Counter
- Counter Output used as Memory (lookup table) Address
- Memory Stores One Period (or portion of period if symmetry is present) of a Waveform
- Output of Memory is Digitized Waveform Value
- This Value is Input to DAC
- DAC Output Filtered to Remove High Frequency Effects
Example of Arbitrary Waveform Generator

- Microcontroller can comprise several of these blocks
- Lookup table data allows any waveform to be generated
- Could also use an ARM with Pre- (or Post-) indexed addressing and let the clock interrupt the processor

Computing the Table Values

- Consider a Sine Wave Generator
- Could use a C program to calculate the values of a Sine Wave from 0 to 90 degrees
  - Take advantage of symmetry of Sine function
  - Take advantage of C math.h library
- These values stored in the data section of an ARM program
- Since floating point is not supported, use Qm.n notation – a fixed point representation
  - Called 'Q notation'
  - In this example, we use Q.31 (‘.’ is missing on p. 137 of book)
  - Number of integral bits often optional so no '.' used – wordsize assumed to be known
- Example C code on page 136-7 of textbook
Qm.n Notation

- Q Notation Refers to Fixed Point Number with m Integral Bits and n Fractional Bits
- Used for Hardware with No Floating Point and has Constant Resolution – Unlike Floating Point
- Resolution – Smallest Incremental Value Between Two Subsequent Values, \( f(x+e) - f(x) \)
  - \( e \) is Smallest Possible Value
  - \( e \) is the “Resolution”
- Integral Part is in 2’s Complement Form (signed form)
- \( m \) is Number of Integral Bits NOT Including the Sign Bit
- Qm.n Requires \( m+n+1 \) Bits

Qm.n Notation (cont)

- Qm.n has a Range of \([-2^m, 2^m-2^n]\)
- Resolution is \( 2^{-n} \)

**EXAMPLE:** Q14.1

Requires 14+1+1=16 Bits
Resolution is \( 2^{-1} = 0.5 \)
Maximum Value is: \( 2^{14} - 0.5 = 16K - 0.5 \)
Minimum Value is: \( -2^{14} = -16K \)
All Values are \([-2^{14}, 2^{14} - 2^{-1}]\)
  = \([-16384.0, +16383.5]\)
  = \([0x8000, 0x8001, 0x8001, ..., 0xFFFF, 0x0000, 0x0001, ..., 0xFFF, 0xFFE, 0xFFF] \)
Qm.n Notation (cont)

- Qm.n has a Range of \([-2^m, 2^m - 2^n]\)
- Resolution is \(2^{-n}\)

**EXAMPLE:** Q6.1
Requires 6+1+1=8 Bits
Resolution is \(2^{-1}=0.5\)
Maximum Value is: \(2^6-0.5=64-0.5\)
Minimum Value is: -\(2^6=-64\)
All Values are \([-2^6, 2^6-2^{-1}]\)
\[= [-64.0, +63.5]\]
\[= [0x80, 0x81, 0x81, ..., 0xFF, 0x00, 0x01, ..., 0xFD, 0xFE, 0xFF]\]

Qm.n Notation Conversion

- From Floating Point to Qm.n
  1) Multiply Floating Point Number by \(2^n\)
  2) Round to the Nearest Integer

- From Qm.n to Floating Point
  1) Convert to Floating Point as if Q-value were Integer
  2) Multiply Converted Floating Point Number by \(2^{-n}\)
Q31 For Sine Wave

- Q31 Allows for Accurate Representation of sin(0) through sin(89) Degrees
- What is the minimum Q31 Representation?

Q31 Range is [0x80000000, 0x7FFFFFFF]

0x80000000 is Minimum Value, in Binary:
1000 0000 0000 0000 0000 0000 0000 0000
0111 1111 1111 1111 1111 1111 1111 1111 +1

1000 0000 0000 0000 0000 0000 0000 0000

Red Bits are the 31 Fractional Bits
This Bit String Represents 0.0 Decimal (as does 0x00000000)

---

Q31 For Sine Wave

- Q31 Allows for Accurate Representation of sin(0) through sin(89) Degrees
- What is the minimum non-zero Q31 Representation?

Q.31 Range is [0x80000000, 0x7FFFFFFF]

0x80000001 is Minimum Value + 1, in Binary:
1000 0000 0000 0000 0000 0000 0000 0001
0111 1111 1111 1111 1111 1111 1111 1110 +1

1111 1111 1111 1111 1111 1111 1111 1111

Red Bits are the 31 Fractional Bits
This Bit String Represents -SUM(2^{-1}+2^{-2}+\ldots+2^{-31}) Decimal
Q31 For Sine Wave

- What is the Q31 for zero?
  \[0.0 = 0x00000000\]
- What is the Q31 Representation for \(\sin(90)\)?
  \[\sin(90^\circ) = 1, \text{ so must represent } #1 \text{ in Q31 Format, but this is impossible in Q31 (fractions only), must use closest value}\]

Q31 Range is \([0x80000000, 0x7FFFFFFF]\)

\(0x7FFFFFFF\) is Maximum Value, in Binary:

\[0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\]

\[= \text{SUM}(2^{-1} + 2^{-2} + \ldots + 2^{-31}) = (15/16 + 15/16^2 + 15/16^3 + \ldots + 15/16^8)\]

\[= 1 - 2^{-32} \text{ (EASY WAY TO COMPUTE)}\]

\[= 0.9375 + 0.05859375 + 0.00366211 + 0.00022888 + 0.00001431 + 0.00000089 + 0.00000005 + 0.000000003\]

\[= 0.999999999767169\]

\[\text{MUST USE THIS FOR } \sin(90) = 1\]

---

Computing the Table Values

```c
#include <stdio.h>
#include <string.h>
#include <math.h>

main()
{
    int i;
    int index=0;
    signed int j[92];
    float sin_val;
    FILE *fp;

    if ((fp = fopen("sindata.twt","w")) == NULL)
    {
        printf("File could not be opened for writing\n");
        exit(1);
    }
```
Computing the Table Values

```c
for (i=0; i<=90; i++) {  /* index i is in units of degrees */
    /* convert to radians */
    sin_val = sin(M_PI*i/180.0);  /* M_PI is pi constant - math.h */
    /* convert to Q31 notation */
    j[i] = sin_val * (2147483648); /* Q31 conversion factor - 2^31 */
}
for (i=1; i<=23; i++) {  /* Generate a line of 4 sin values */
    fprintf(fp, "DCD     ");
    fprintf(fp,"0x%x," ,j[index]);
    fprintf(fp,"0x%x," ,j[index+1]);
    fprintf(fp,"0x%x," ,j[index+2]);
    fprintf(fp,"0x%x," ,j[index+3]);
    fprintf(fp,"\n");
    index += 4;
}
fclose(fp);
```

Sine Wave – Only need 0 through 90 deg

\[
\begin{align*}
\sin(180 - \theta) &= \sin(\theta) \\
\sin(\theta - 180) &= -\sin(\theta) \\
\sin(360 - \theta) &= \sin(\theta) \\
\sin(\theta + 360) &= \sin(\theta) \\
\sin(-\theta) &= -\sin(\theta)
\end{align*}
\]
Using Symmetry with Sine Lookup Table

- First Part of Program Converts to Angle Between 0 and 90
  - Compares to 90 then 180 then 270
  - If Angle <= 90, lookup sine value
  - If Angle <= 180, Angle < 180-Angle
  - If Angle <= 270, Angle < Angle-180
  - If Angle <= 360, Angle < 360-Angle
- Assumes Angle NOT Greater than 360
- Lookup Value
  - Negate Value if Original Angle > 180

\[
\sin(\theta - 180) = -\sin(\theta) \quad \sin(360 - \theta) = -\sin(\theta)
\]

ARM Program with Sine Lookup Table

```
AREA SINETABLE, CODE
ENTRY
; Registers used:
;   r0 <- return value in Q.31 notation
;   r1 <- sin argument (in degrees from 0 to 360)
;   r2 <- temp
;   r4 <- starting address of sine table
;   r7 <- copy of argument

main
  mov r7, r1 ; make copy of argument
  ldr r2, =270 ; const. won’t fit with rotation scheme
  adr r4, sin_data ; load address of sin data table
  ; check if angle is less than or equal to 90
  cmp r1, #90 ; set flags based on 90 degrees
  ble retvalue ; if N=1, value is LT 90, go to table
  ; check if angle is less than or equal to 180
  cmp r1, #180 ; set flags based on 180 degrees
  rsble r1, r1, #180 ; if N=1, subtract angle from 180
```
ARM Program with Sine Lookup Table

```
ble  retvalue  ;if N=1, value is LT 180, go to table
;check if angle is less than or equal to 270
cmp r1, r2  ;set flags based on 270 degrees
suble r1, r1, #180  ;if N=1, subtract 180 from angle
ble  retvalue  ;if N=1, value is LT 270, go to table
;angle must be GT 270 and LT 360
rsb r1, r1, #360  ;subtract angle from 360
retvalue

ldr r0, [r4, r1, LSL #2]  ;lookup Q.31 sine value
;must now determine whether or not to negate value
cmp r7, #180  ;set flags to determine need to negate
rsbgt r0, r0, #0  ;negate value if N=0
```

done  b  done

ALIGN

sin_data

```
DCD 0x00000000, 0x023be164, 0x04779630, 0x06b2f1d8
DCD 0x08edc7b0, 0x02b7eb50, 0x0d613050, 0x0f996a30
DCD 0x11d06ca0, 0x14060b80, 0x163a1a80, 0x186c6de0

... ...

DCD 0x7fec0a00, 0x7ffb0280, 0x7fffffff
```

END

- Complete Table on Page 138
- This Part of File Generated by the C Program
ARM Program with Sine Lookup Table

```assembly
ldr r0, [r4, r1, LSL #2]; get sin value from table
```

- Actual Table Lookup Instruction
- r4 Contains Starting Address of Table
- r1 Contains Computed Degrees < 90
- Pre-Indexed Addressing
- LSL #2 Multiplies by 4 Since Byte Addressable
- Angle is Multiplied by 4 Since 4 sine Values per Word (DCD)

FUTURE PROJECT – DIGITAL RECORDER

- Add ADC & Audio Amp Chip to Evaluator7T Board
- Interface ADC to Samsung Processor
- 2 Programs – RECORD PLAYBACK
  - RECORD
    - Digitizes Speech From Microphone – store in memory
  - PLAYBACK
    - Read Recorded Speech From Memory
    - Send to ADC
    - Output ADC to Audio Amplifier & Speaker
Jump Tables – Another Lookup Table

- Jump Tables Contain *Addresses* Instead of *Data*
- Allows a Microcontroller to Execute One of Several Subroutines Based on an Input Value; A CASE Statement
- Replaces a Series of Comparisons and Branches
- Microcontroller Receives/Computes a Value, Then Based on Value, Accesses the Jump Table
- Jump Table then “Points” to the Appropriate Subroutine to Execute

Jump Table Example

- Three Input Values:
  - First Value (0 or 1) Used to Determine Whether to ADD or SUBTRACT
  - Second and Third Values are Operands to be Added or Subtracted
- Uses Equate Directive (\texttt{EQU}) to Set the Number of Jump Targets
- Calls High-Level Function that Processes the Control Argument and Uses Jump Table to Call Appropriate Lower-Level Function
- Lower Level Function Executes and Returns to Main Program
Jump Table Example

AREA Jump, CODE, READONLY ;Name this block of code
CODE32 ;Following code is ARM code
num EQU 2 ;Number of entries in jump table
ENTRY ;Mark first instr to execute

start ;First instruction to call
mov r0, #0 ;Set up the three parameters
mov r1, #3
mov r2, #2
bl arithfunc ;Call the function

stop b stop ;Processing complete

arithfunc ;Label function
cmp r0, #num ;function code is unsigned int.
movhs pc, lr ;if code >=num (C=1) return
adr r3, JumpTable ;load address of jump table
ldr pc, [r3,r0,LSL #2] ;Jump to appropriate routine

Jumptable

DCD DoAdd
DCD DoSub

DoAdd add r0, r1, r2 ;Operation 0
mov pc, lr ;Return

DoSub sub r0, r1, r2 ;Operation 1
mov pc, lr ;Return

END ;Mark the end of file

• Starting Address of Jump Table in Register r3
• r0 Contains 0 or 1 Condition
• Multiplied by 4 (LSL #2) to Account for Word Alignment
  – DCD Do??? Are Word Aligned Addresses in Memory
Searching Lists/Tables

- Classical Problem Involving a Table of “Keys” and “Information”
- A “Key” is Input and a Table is Accessed to
  1) Determine if the Key and Information is Present
  2) Possibly Return the Data Associated with the Key
- Straightforward Approach is to Perform a Linear Search, Could Require Worst Case of Searching all N Keys
- If Table is Pre-processed, Other Method Can be Used Allowing "almost" Direct Access

Table Pre-Processing

- No Pre-Processing
  - Sequential Search, O(N)
- Storing Using a Hash Function
  - Near Constant-time Access, O(1)
- Storing in Sorted Key Order
  - Binary Search, O[\lg(N)]
- We Will Consider the Binary Search Method in More Detail
Binary Search Table

- Assume Keys in Ascending Numerical Order
- First Compare Input Key to Middle Key
- Can Immediately Rule Out Top or Bottom Half Based on Input Key GT or LT Middle Key
- Next Set Area to Search Equal to Top or Bottom Half of Table and Repeat
- First Discard Half, then One-Fourth, One-Eighth, etc.
C Program Description

```c
first = 0;
last = num_entries - 1;
index = 0;
while ((index == 0) & (first <= last)) {
    middle = (first + last)/2;
    if (key == table[middle]) index = middle;
    else if (key < table[middle]) last = middle - 1;
    else first = middle + 1;
}
```

Example – Two Passes in Search

<table>
<thead>
<tr>
<th>First</th>
<th>100</th>
<th>206</th>
<th>412</th>
<th>800</th>
<th>Key</th>
<th>992</th>
</tr>
</thead>
<tbody>
<tr>
<td>Middle</td>
<td>947</td>
<td></td>
<td></td>
<td></td>
<td>First</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>968</td>
<td>992</td>
<td></td>
<td></td>
<td>Middle</td>
<td>947</td>
</tr>
<tr>
<td>Last</td>
<td>1064</td>
<td></td>
<td></td>
<td></td>
<td>Last</td>
<td>1078</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

First Pass

<table>
<thead>
<tr>
<th>First</th>
<th>100</th>
<th>206</th>
<th>412</th>
<th>800</th>
<th>Key</th>
<th>992</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Middle</td>
<td>947</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Last</td>
<td>1078</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Second Pass
**Table Data Example**

- Assume Table Start Address is \(0x4000\)
- Each Entry is 16 Bytes
  - First Word (4 Bytes) is Key
  - Remaining 12 Bytes are Character Data

\[
\text{address} = \text{table_addr} + i \times \text{size_of_entry}
\]

- Example Second Entry Address is

\[
0x4000 + 2 \times 16 = 0x4020
\]

---

**Example Table Structure**

<table>
<thead>
<tr>
<th>Table base address</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldr r7, [r6, r2, LSL #4]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4 Byte Key</th>
<th>12 Bytes of Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000034</td>
<td>Vacuums</td>
</tr>
<tr>
<td>0x00000234</td>
<td>Clothes</td>
</tr>
<tr>
<td>0x00003403</td>
<td>Candy</td>
</tr>
<tr>
<td>0x0010382C</td>
<td>Telephones</td>
</tr>
</tbody>
</table>

...
Binary Search Implementation

- Use `ldr` Instruction with Pre-indexed Addressing
- Table Base Address is Offset with a Scaled Index
- Scaling Specified as Constant `ESIZE`
- Assumes Table Size is Power of Two
  - If not Power of Two, Use a Two-Level Table
  - Entry is Key and Address Pointing to Data in Memory
- Load Table Entries with Single Pre-indexed Instruction
  
  ```
  ldr r7, [r6, r2, LSL #ESIZE]
  ```
- Using `ESIZE` Equate Allows for Easily Changing Table Size

Example Code

```plaintext
NUM       EQU  14 ;Number of entries in table
ESIZE     EQU  4 ;log(Size) of Entry (in bytes)
AREA      BIN-SRCH, CODE
ENTRY     ;Mark first instr to execute

; Registers used:
; r0 - first entry
; r1 - last entry
; r2 - middle entry
; r3 - index
; r4 - size of the entries (log 2)
; r5 - the key (what you’re searching for)
; r6 - address of the list
; r7 - temp

  ldr r5, =0x200 ;look for key-data 0x200 (PINEAPPLE)
  adr r6, table_start ;load address of the table
  mov r0, #0 ;first = 0
  mov r1, #NUM-1 ;last - number of entries
                 ;in list-1
```
Example Code (cont)

```assembly
loop   cmp    r0, r1          ;compare first and last (first-last)
       movgt  r2, #0          ;first>last, no key found, middle=0
       bgt    done           ;<calulate arithmetic middle index
       add    r2, r0, r1      ;first + last
       mov    r2, r2, ASR #1  ;first <- first + last/2
       ;get key of middle entry
       ldr    r7, [r6, r2, LSL #ESIZE]  ;load the entry
       cmp    r5, r7          ;compare key to middle value loaded
       ;conditionally pick to update either 'last' or 'first'
       addgt  r0, r2, #1      ;first = middle + 1
       sublt  r1, r2, #1      ;last = middle - 1
       bne    loop            ;if first NE last, continue search

done   mov    r3, r2          ;move middle to 'index' - entry found
stop   b      stop
```

Example Code (cont)

```assembly
table_start
   DCD  0x004
   DCB  "PEPPERONI  
   DCD  0x005
   DCB  "ANCHOVIES  
   DCD  0x010
   DCB  "OLIVES      
   DCD  0x012
   DCB  "GREEN PEPPER"
   DCD  0x018
   DCB  "BLACK OLIVES"
   DCD  0x022
   DCB  "CHEESE      
   DCD  0x024
   DCB  "EXTRA SAUCE 
   DCD  0x026
   DCB  "CHICKEN     
```
Example Code (cont)

DCD 0x030
DCB "CANADIAN BAC"
DCD 0x035
DCB "GREEN OLIVES"
DCD 0x038
DCB "MUSHROOMS"
DCD 0x100
DCB "TOMATOES"
DCD 0x200
DCB "PINEAPPLE"
DCD 0x300
DCB "PINE NUTS"
END