Recap
Subroutines, the stack, switches, LEDs

Overview
Pointers
Indexed mode addressing
TCNT (free running 16-bit time)
Introduction to Lab 3

Can we collect data to prove it works?
Input, output, time

Read sections 4.5, 6.1, 6.2, 6.3 and 6.11

Figure 6.1. Pointers are addresses pointing to objects. The objects may be data, functions, or other pointers.

6.1. Indexed addressing modes used in implementing pointers

Figure 6.2. Examples of data structures that utilize pointers.

If Register X or Y contains an address, we say it points into memory

Definitions in C
\[
\text{unsigned char data;} \quad // 8\text{-bit value} \\
\text{unsigned char out;} \quad // 8\text{-bit value} \\
\text{unsigned char *pt;} \quad // 16\text{-bit address}
\]

Definitions in assembly
\[
data \text{ rmb 1} \quad ; 8\text{-bit value} \\
out \text{ rmb 1} \quad ; 8\text{-bit value} \\
pt \text{ rmb 2} \quad ; 16\text{-bit address}
\]

Initialization in C
\[
pt = \&\text{data}; \quad // \text{pointer to data}
\]

Initialization in assembly
\[
\text{ldx} \ #\text{data} \quad // \text{pointer to data} \\
\text{stx} \ pt
\]

Dereference in C
\[
\text{out} = \*\text{pt}; \quad // \text{fetch value at pointer}
\]
Dereference in assembly

```assembly
ldx pt ; X points to data
ldaa 0,x ; fetch value at pointer
staa out

; read 8-bit contents pointed to by X
```

16-bit definitions in C

```c
unsigned short data; // 16-bit value
unsigned short out; // 16-bit value
unsigned short *pt; // 16-bit address
```

16-bit definitions in assembly

```assembly
data rmb 2    ; 16-bit value
out rmb 2    ; 16-bit value
pt rmb 2    ; 16-bit address
```

Initialization in C

```c
pt = &data;  // pointer to data
```

Initialization in assembly

```assembly
ldx #data    // pointer to data
stx pt
```

16-bit dereference in C

```c
out = *pt;  // fetch value at pointer
```

16-bit dereference in assembly

```assembly
ldy pt ; Y points to data
ldd 0,y ; 16-bit fetch at pointer
std out

; read 16-bit contents pointed to by Y
```

### 6.1.1. Indexed addressing mode

**Indexed** addressing uses a fixed offset with the 16-bit registers: X, Y, SP, or PC.

- 5-bit (-16 to +15),
- 9-bit (-256 to +127), or
- 16-bit

<table>
<thead>
<tr>
<th>Machine</th>
<th>Opcode</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6A5C</td>
<td>staa</td>
<td>-4,Y</td>
<td>[Y-4] = RegA</td>
</tr>
</tbody>
</table>

Let \( n, R \) be the indexed address

- fixed offset \( n \)
- index register \( R \)

Then

\[ EAR = R + n \]
16-bit data structures with indexed addressing is different in assembly versus in C.

Prime fdb 1,2,3,5,7,11,13,17,19,23

The equivalent ROM-based definition is C would be

```c
unsigned short const Prime[10] =
{1,2,3,5,7,11,13,17,19,23};
```

Want to fetch the 7 from Prime[4] In assembly.

```
ldx #Prime ; pointer to the structure
ldd 8,x ; read element number 4
```

or if we could have fetched it directly as

```
ldd Prime+8 ; read Prime[4]
```

Either way, manipulating addresses in assembly always involves the physical byte-address regardless of the precision of the data.

Want to increment the pointer to the next element.

In C, we define the pointer as

```c
unsigned short const *Pt;
```

and initialize it as

```c
Pt = Prime;
```

To increment the pointer to the next element

In C, use the expression `Pt++`.

In assembly, we can define the pointer in RAM as

```assembly
Pt rmb 2 ; 16-bit pointer to Prime
```

and initialize it as

```
ldx #Prime
stx Pt ; pointer to Prime[0]
```

However, to increment the pointer to the next element we have to add 2 to the pointer. E.g.,

```
ldx Pt ; previous pointer
inx
inx ; next element in the 16-bit structure
stx Pt
```

6.1.2. Auto Pre/Post Decrement/Increment Indexed addressing mode

**Optimized addressing modes to make it run fast**

Not on Exam!

Not really needed for Lab

Regular access of an array

```assembly
staa 0,Y ; Store RegA at 2345,
iny ; Reg Y=2346
```

Post-increment addressing first accesses the data then adds to the index register:

```assembly
staa 1,Y+ ; Store at 2345, then Reg Y=2346
```

Regular access of an array, Y points to 16-bit element at 2344

```assembly
iny ; Reg Y=2345
iny ; Reg Y=2346
std 0,Y ; Store RegD at 2346,2347
```
Pre-increment addressing first adds to the index register then accesses the data:
```
std 2,+Y ;Reg Y=2346, then store at 2346
```

Post-decrement addressing first accesses data then subtracts from index register:
```
staa 1,Y- ;Store at 2345, then Reg Y=2344
```

Pre-decrement addressing first subtracts from index register then accesses the data:
```
staa 1,-Y ;Reg Y=2344, then store at 2344
```

### 6.1.3. Accumulator Offset Indexed addressing mode

Two registers combined to make effective address
- One register points to array, other register has array index
- Not on Exam1
- Not really needed for Lab

The offset is located in one of the accumulators A, B or D, and the base address is in one of the 16-bit registers: X, Y, SP, or PC.

```
ldab #4
ldy #2345
staa B,Y  ;Store at 2349 (B & Y unchanged)
```

### 6.1.4. Indexed Indirect addressing mode

Optimized addressing modes for complex data structures
- Not on Exam1
- Not needed for Lab

```
ldy #2345
staa [-4,Y] ;fetch 16-bit address from 2341, store 56 at 1234
```

![Figure 6.6. Example of the 9S12 indexed-indirect addressing mode.](image)

### 6.1.5. Accumulator D Offset Indexed Indirect addressing mode

Optimized addressing modes for complex data structures
- Not on Exam1
- Not needed for Lab

```
ldd #4
ldy #2341
stx [D,Y] ;Store copy of value in Reg X at 1234 (D & Y unchanged)
```

![Figure 6.7. Example of the 9S12 accumulator-offset indexed-indirect addressing mode.](image)

### 6.1.6. Post-byte machine coded for indexed addressing

For more information see Tables 6.1 and 6.2 in the book

*Show xb- table*

1) Open CPU12rg.pdf

2) Click on Indexed Addressing Mode Postbyte Encoding (xb)
6.1.6. Load effective address instructions

\[
\begin{align*}
\text{leax } & \text{idx } ; \text{RegX=EA} \\
\text{leay } & \text{idx } ; \text{RegY=EA} \\
\text{leas } & \text{idx } ; \text{RegS=EA}
\end{align*}
\]

In each of the following cases, the effective address, EA, is loaded into Register X.

\[
\begin{align*}
\text{leax } & m,r ; \text{IDX 5-bit index, X=r+m (-16 to 15)} \\
\text{leax } & v,+r ; \text{IDX pre-inc, r=r+v, X=r (1 to 8)} \\
\text{leax } & v,-r ; \text{IDX pre-dec, r=r-v, X=r (1 to 8)} \\
\text{leax } & v,r+ ; \text{IDX post-inc, X=r, r=r+v (1 to 8)} \\
\text{leax } & v,r- ; \text{IDX post-dec, X=r, r=r-v (1 to 8)} \\
\text{leax } & A,r ; \text{IDX Reg A offset, X=r+A, zero padded} \\
\text{leax } & B,r ; \text{IDX Reg B offset, X=r+B, zero padded} \\
\text{leax } & D,r ; \text{IDX Reg D offset, X=r+D} \\
\text{leax } & q,r ; \text{IDX1 9-bit index, X=r+q (-256 to 255)} \\
\text{leax } & W,r ; \text{IDX2 16-bit index, X=r+W (-32768 to 65535)}
\end{align*}
\]

where \( r \) is Reg X, Y, SP, or PC, and the fixed constants are

\( m \) is any signed 5-bit -16 to +15
\( q \) is any signed 9-bit -256 to +255
\( v \) is any unsigned 3 bit 1 to 8
\( W \) is any signed 16-bit -32768 to +32767 or any unsigned 16-bit 0 to 65535

4.5. 16-bit timer

\[
\begin{array}{cccccccc}
\text{b15} & \text{b14} & \text{b13} & \text{b12} & \text{b11} & \text{b10} & \text{b9} & \text{b8} & \text{b7} & \text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} & \text{b0} & \text{TCNT} \\
\hline
\text{S0044} & & & & & & & & & & & & & & & \\
\text{S0046} & \text{TEN} & \text{TSWAI} & \text{TSFRZ} & \text{TFFCA} & 0 & 0 & 0 & 0 & 0 & \text{TSCR1} \\
\text{S004D} & \text{TOI} & 0 & 0 & 0 & \text{TCRE} & \text{PR2} & \text{PR1} & \text{PR0} & \text{TSCR2} \\
\text{S004F} & \text{TOF} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{TFLG2} \\
\end{array}
\]

Table 4.11. 9S12 timer ports.

<table>
<thead>
<tr>
<th>PR2</th>
<th>PR1</th>
<th>PR0</th>
<th>Divide by</th>
<th>E = 8 MHz</th>
<th>E = 24 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>TCNT period</td>
<td>TCNT frequency</td>
<td>TCNT period</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>125 ns</td>
<td>8 MHz</td>
<td>41.7 ns</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>250 ns</td>
<td>4 MHz</td>
<td>83.3 ns</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>500 ns</td>
<td>2 MHz</td>
<td>167 ns</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1 µs</td>
<td>1 MHz</td>
<td>333 ns</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2 µs</td>
<td>500 kHz</td>
<td>667 ns</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>4 µs</td>
<td>250 kHz</td>
<td>1.33 µs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>8 µs</td>
<td>125 kHz</td>
<td>2.67 µs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>16 µs</td>
<td>62.5 kHz</td>
<td>5.33 µs</td>
</tr>
</tbody>
</table>

Table 4.12. Given an E clock frequency, the PR2 PR1 and PR0 bits define the TCNT rate.

; 9S12DP512 at 8 MHz
; Enable TCNT at 1us
Timer_Init
    movb #$80,TSCR1 ;enable
    movb #$03,TSCR2 ;divide by 8
    rts

Jonathan W. Valvano
6.11. Functional Debugging

6.11.1. Instrumentation: dump into array without filtering

Assume `happy` is a strategic 8-bit variable.

```
SIZE   equ 20
Buf    rmb SIZE
Pt     rmb 2
```

`Pt` will point into the buffer. `Pt` must be initialized to point to the beginning, before the debugging begins.

```
ldx #Buf
stx Pt
```

The debugging instrument saves the strategic variable into the `Buffer`.

```
Save
  pshb
  pshx    ;save
  ldx  Pt   ;X=>Buf
  cpx  #Buf+SIZE
  bhs  done ;skip if full
  ldab happy
  stab 0,X ;save happy
  inx      ;next address
  stx  Pt
  done
  pulx
  pulb
  rts

void Save(void){
  if(Pt < &Buf[SIZE]){
    (*Pt) = happy;
    Pt++;
  }
}
```

Similar to Program 6.37. Instrumentation dump.

Next, you add `jsr Save` statements at strategic places within the system.

Use the debugger to display the results after the program is done.

6.2. Arrays

**Random access**

An array

```
equal precision and allows random access.
```

The **precision** is the size of each element.

The **length** is the number of elements (fixed or variable).

The **origin** is the index of the first element.

**zero-origin indexing.**

In general, let `n` be the precision of a zero-origin indexed array in bytes.

If `I` is the index and `Base` is the base address of the array, then the address of the element at `I` is

```
Base+n*I
```

In the previous examples, the length of the array was known.

One simple mechanism to save the length of the array as the first element.

```
const char Data[5]={4,0x05,0x06,0x0A,0x09};
const short Powers[6]={5,1,10,100,1000,10000};
```

We could define these variable length arrays in assembly as

```
Data   fcb 4,$05,$06,$0A,$09
Powers fdb 5,1,10,100,1000,10000
```
Another common mechanism to handle variable length is a termination code.

<table>
<thead>
<tr>
<th>ASCII</th>
<th>code</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>$00</td>
<td>null</td>
</tr>
<tr>
<td>ETX</td>
<td>$03</td>
<td>end of text</td>
</tr>
<tr>
<td>EOT</td>
<td>$04</td>
<td>end of transmission</td>
</tr>
<tr>
<td>FF</td>
<td>$0C</td>
<td>form feed</td>
</tr>
<tr>
<td>CR</td>
<td>$0D</td>
<td>carriage return</td>
</tr>
<tr>
<td>ETB</td>
<td>$17</td>
<td>end of transmission block</td>
</tr>
</tbody>
</table>

Table 6.3. Typical termination codes

### 6.3. Strings

A **string** is a data structure with equal size elements that only allows sequential access.

**Example 6.4.** Write software to output an ASCII string to the serial port.

**Solution**

Because the length of the string may be too long to place all the ASCII character into the registers at the same time, call by reference parameter passing will be used. With call by reference, a pointer to the string will be passed. The function `OutString`, shown in Program 6.6, will output the string data to the serial port. The function `SCI_OutChar` will be developed later in Chapter 8 and shown as Program 8.2. For now all we need to know is that it outputs a single ASCII character to the serial port. The main program calls this function twice, with different ASCII strings.

```assembly
Hello fcc "Hello World"
fcb 0
CRLF fcb 13,10,0
;Reg X points to the string data
OutString
  ldaa 1,x+ ;next data
  beq done ;0 means end
  jsr SCI_OutChar
  bra OutString
done rts
main lds #$4000
  bsr SCI_Init
mloop ldx #Hello ;first string
  bsr OutString
  ldx #CRLF ;second string
  bsr OutString
bra mloop
```

Program 6.6. A variable length string contains ASCII data.

**The bottom line**

- **Pointers are addresses**
- **Indexed addressing mode used for pointers**
- **Precision: 8 bit or 16-bit**
- **Arrays and strings have equal precision elements**
- **TCNT is a 16-bit free-running clock**

Jonathan W. Valvano