C ARRAYS

The C source code

```c
int hist[100];
int score = 92;
...
hist[score] += 1;
```

might translate to:

```assembly
hist: .space 100
score: .word 92

mov R3, #hist
ldr R2, score
ldr R1, [R3, R2, LSL #2]
add R1, R1, #1
str R1, [R3, R2, LSL #2]
```

Address:

CONSTANT base address + scaled VARIABLE offset
C "STRUCTS"

- C "structs" are lightweight "container objects" - objects with members, but no methods.
- There is special "Java-like" syntax for accessing particular members: `variable.member` (actually, Java's dot operator "." is borrowed from C)
- You can also have pointers to structs.

C provides an new operator to access them:

```c
pointerVariable->member
```

This simplifies the alternative syntax:

```c
(*pointerVariable).member
```

```c
struct Point {
    int x, y;
} P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 123;
```


**Structs in Action**

```c
struct Point {
    int x, y;
} P1, P2, *p;
...
P1.x = 157;
...
p = &P1;
p->y = 123;
```

**Address:**

- VARIABLE base address + CONSTANT offset

```
P1: .space 8
P2: .space 8
p: .space 4
...
mov R1,#P1
mov R0,#157
str R0,[R1,#0] ; P1.x = 157
str R1,#p ; p = &P1
ldr R2,#p
mov R0,#123
str R0,[R2,#4] ; p->y = 123
```
C “if” to Assembly Translation

C code:
if (expr) {
    STUFF
}

C code:
if (expr) {
    STUFF1
} else {
    STUFF2
}

ARM assembly:
(compute expr)
beq Lendif
   (compile STUFF)
Lendif:

Note: the branches used in assembly “SKIP” code blocks rather than cause them to be executed. This often results in a complement test!

ARM assembly:
(compute expr)
beq Lelse
   (compile STUFF1)
b    Lendif
Lelse:
   (compile STUFF2)
Lendif:
C "WHILE" LOOPS

C code:
while (expr) {
    STUFF
}

Assembly:
Lwhile:
    (compute expr)
    beq Lendw
    (compile STUFF)
    b Lwhile
Lendw:

Alternate Assembly:
    b Ltest
    Lwhile:
    (compute STUFF)
    Ltest:
    (compute expr)
    bne Lwhile
Lendw:

Compilers spend a lot of time optimizing in and around loops.
- moving all possible computations outside of loops
- unrolling loops to reduce branching overhead
- simplifying expressions that depend on "loop variables"
C “FOR” LOOPS

- Most high-level languages provide loop constructs that establish and update an iterator, which controls the loop’s behavior

    for (initialization; conditional; afterthought) {
        STUFF;
    }

Assembly:
    (compile initialization)
    Lfor:
        (compute conditional)
        beq Lendfor
        (compile STUFF)
        (compile afterthought)
    B   Lfor
    Lendfor:

For loops are the most commonly used form of iteration found programming languages.

Their advantage is readability. They bring together the three essential components of iteration, setting an initial value, establishing a termination condition, and giving an update rule.

Ahhh, but one other iteration forms there are!
Next time

• The details behind assemblers
• 2-pass and 1-pass assembly
• Linkers and dynamic libraries
Long, long, time ago, I can still remember
How mnemonics used to make me smile...
Cause I knew with just those opcode names
That I could play some assembly games
And I'd be hacking kernels in just awhile.
But Comp 411 made me shiver,
With every new lecture that was delivered,
There was bad news at the doorstep,
I just didn't get the problem sets.
I can't remember if I cried,
When inspecting my stack frame’s insides,
All I know is that it crushed my pride,
On the day the joy of software died.
And I was singing...

When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."
Routes from Programs to Bits

- Traditional Compilation

High-level, portable (architecture independent) program description:

1. C or C++ program
   - Compiler
     - Assembly Code
       - Assembler
         - "Object Code"

Architecture dependent mnemonic program description with symbolic memory references:

1. "Library Routines"
   - A collection of precompiled object code modules
2. Linker
   - "Executable"
     - Loader
       - "Memory"

Machine language with all memory references resolved:

Program and data bits loaded into memory.
How an Assembler Works

Three major components of assembly

1) Allocating and initializing data storage
2) Conversion of mnemonics to binary instructions
3) Resolving addresses

```assembly
.word 0x03fffffff, main
array: .space 11
total: .word 0

main:
mov r1,#array
mov r2,#0
mov r3,#1
ldr r0,total
b test

loop:
add r0,r0,r3
str r3,[r1,r2,lsr #2]
add r3,r3,r3
add r2,r2,#1

test:
cmp r2,#11
blt loop
str r0,total

*halt: b halt
```

Need to figure out this immediate value
This one is a PC-relative offset
This is a forward reference
This offset is completely different than the one a few instructions ago
So is this
RESOLVING ADDRESSES - 1ST PASS

"Old-style" 2-pass assembler approach

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine code</th>
<th>Assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x03FFFFFFFC</td>
<td>.word 0x03ffffffc, main</td>
</tr>
<tr>
<td>4</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0x00000000</td>
<td>array: .space 11</td>
</tr>
<tr>
<td>52</td>
<td>0x00000000</td>
<td>total: .word 0</td>
</tr>
<tr>
<td>56</td>
<td>0xE3A01000</td>
<td>main: mov r1, #array</td>
</tr>
<tr>
<td>60</td>
<td>0xE3A02000</td>
<td>mov r2, #0</td>
</tr>
<tr>
<td>64</td>
<td>0xE3A03001</td>
<td>mov r3, #1</td>
</tr>
<tr>
<td>68</td>
<td>0xE51F0000</td>
<td>ldr r0, total</td>
</tr>
<tr>
<td>72</td>
<td>0xEA000000</td>
<td>b test</td>
</tr>
<tr>
<td>76</td>
<td>0xE0800003</td>
<td>loop: add r0, r0, r3</td>
</tr>
<tr>
<td>80</td>
<td>0xE7813102</td>
<td>str r3, [r1, r2, lsl #2]</td>
</tr>
<tr>
<td>84</td>
<td>0xE0833003</td>
<td>add r3, r3, r3</td>
</tr>
<tr>
<td>88</td>
<td>0xE2822001</td>
<td>add r2, r2, #1</td>
</tr>
<tr>
<td>92</td>
<td>0xE352000B</td>
<td>test: cmp r2, #11</td>
</tr>
<tr>
<td>96</td>
<td>0xBA000000</td>
<td>blt loop</td>
</tr>
<tr>
<td>100</td>
<td>0xE50F0000</td>
<td>str r0, total</td>
</tr>
<tr>
<td>104</td>
<td>0xEA000000</td>
<td>*halt: b halt</td>
</tr>
</tbody>
</table>

- In the first pass, data and instructions are encoded and assigned offsets, while a symbol table is constructed.
- Unresolved address references are set to 0
"Old-style" 2-pass assembler approach

- In the first pass, data and instructions are encoded and assigned offsets, while a symbol table is constructed.
- Unresolved address references are set to 0.
Modern 1-pass Assembler

Modern assemblers keep more information in their symbol table which allows them to resolve addresses in a single pass.

- Known addresses (backward references) are immediately resolved.
- Unknown addresses (forward references) are "back-filled" once they are resolved.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Resolved?</th>
<th>Reference list</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>8</td>
<td>y</td>
<td>56</td>
</tr>
<tr>
<td>total</td>
<td>52</td>
<td>y</td>
<td>68</td>
</tr>
<tr>
<td>main</td>
<td>56</td>
<td>y</td>
<td>4</td>
</tr>
<tr>
<td>loop</td>
<td>76</td>
<td>y</td>
<td>?</td>
</tr>
<tr>
<td>test</td>
<td>?</td>
<td>n</td>
<td>72</td>
</tr>
</tbody>
</table>
 ROLE OF A LINKER

Some aspects of address resolution cannot be handled by the assembler alone.

1. References to data or routines in other object modules
2. The layout of all segments in memory
3. Support for **REUSABLE** code modules
4. Support for **RELOCATABLE** code modules

This final step of resolution is the job of a **LINKER**.
**Static and Dynamic Libraries**

- **LIBRARIES** are commonly used routines stored as a concatenation of "Object files". A global symbol table is maintained for the entire library with **entry points** for each routine.

- When a routine in a LIBRARY is referenced by an assembly module, the routine's address is resolved by the **LINKER**, and the appropriate code is added to the executable. This sort of linking is called **STATIC linking**.

- Many programs use common libraries. It is wasteful of both memory and disk space to include the same code in multiple executables. The modern alternative to STATIC linking is to allow the **LOADER** and **THE PROGRAM ITSELF** to resolve the addresses of libraries routines. This form of lining is called **DYNAMIC linking** (e.x. .dll).
Dynamically Linked Libraries

- C call to library function:
  
  ```c
  printf("sqr[%d] = %d\n", x, y);
  ```

- Assembly code

  ```assembly
  mov     R0,#1
  mov     R1,ctrlstring
  ldr     R2,x
  ldr     R3,y
  mov     IP,__stdio__
  mov     LR,PC
  ldr     PC,[IP,#16]
  ```

  Why are we loading the PC from a memory location rather than branching?

How does dynamic linking work?
Dynamically Linked Libraries

- Lazy address resolution:

  sysload: stmfd sp!, [r0-r10, lr]

  .

  ; check if stdio module
  ; is loaded, if not load it

  .

  ; backpatch jump table
  mov r1, __stdio__
  mov r0, dfopen
  str r0, [r1]
  mov r0, dfclose
  str r0, [r1, #4]
  mov r0, dfputc
  str r0, [r1, #8]
  mov r0, dfgetc
  str r0, [r1, #12]
  mov r0, dfprintf
  str r0, [r1, #16]

Because, the entry points to dynamic library routines are stored in a TABLE. And the contents of this table are loaded on an "as needed" basis!

Before any call is made to a procedure in "stdio.dll"

.globl __stdio__:
__stdio__:
fopen: .word sysload
fclose: .word sysload
fgetc: .word sysload
fputc: .word sysload
fprintf: .word sysload

After the first call is made to any procedure in "stdio.dll"

.globl __stdio__:
__stdio__:
fopen: dfopen
fclose: dclose
fgetc: dfgetc
fputc: dfputc
fprintf: dprintf
Modern Languages

Intermediate "object code language"

- High-level, portable (architecture independent) program description
  - PORTABLE mnemonic program description with symbolic memory references
  - An application that EMULATES a virtual machine. Can be written for any Instruction Set Architecture. In the end, machine language instructions must be executed for each JVM bytecode

```
Java program
```
```
Compiler
```
```
JVM bytecodes
```
```
"Library Routines"
```
```
Interpreter
```
Modern Languages

Intermediate "object code language"

- High-level, portable (architecture independent) program description
  - Java program
  - Compiler
  - JVM bytecodes
  - JIT Compiler
  - Machine code
  - "Library Routines"

PORTABLE mnemonic program description with symbolic memory references

While interpreting on the first pass the JIT keeps a copy of the machine language instructions used. Future references access machine language code, avoiding further interpretation.

Today's JITs are nearly as fast as a native compiled code.
**Assembly? Really?**

- In the early days compilers were dumb
  - literal line-by-line generation of assembly code of "C" source
  - This was efficient in terms of S/W development time
    - C is portable, ISA independent, write once-run anywhere
    - C is easier to read and understand
    - Details of stack allocation and memory management are hidden
  - However, a savvy programmer could nearly always generate code that would execute faster

- Enter the modern era of Compilers
  - Focused on optimized code-generation
  - Captured the common tricks that low-level programmers used
  - Meticulous bookkeeping (i.e. will I ever use this variable again?)
  - It is hard for even the best hacker to improve on code generated by good optimizing compilers
Next Time

- Compiler code optimization
- We look deeper into the Rabbit hole