Assemblers and Linkers

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."

- Problem set #2 due tonight at 11:59:59pm
- 1st midterm next Monday (10/8)
- Midterm study session first 45 mins of Friday’s lab
A Route from Program to Bits

- Traditional Compilation

High-level, portable (architecture independent) program description

C or C++ program

Compiler

Assembly Code

Assembler

"Object Code"

Library Routines

A collection of precompiled object code modules

Dependent program description with symbolic memory references

Machine language with "some" remaining symbolic memory references

"Executable"

Loader

"Memory"

Program and data bits loaded into memory

Architecture, ISA, Dependent program description with symbolic memory references

Machine language with all memory references resolved
**WHAT AN ASSEMBLER DOES**

Assembly is just a recipe for sequentially filling memory locations.

```assembly
.word 0x03ffffffc, 0x00000020
.space 6
.word 0xE3A00000, 0xE2900001, 0x1AFFFFFD
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>in decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x03ffffffc</td>
<td>6718860</td>
</tr>
<tr>
<td>0x00000004</td>
<td>0x00000020</td>
<td>32</td>
</tr>
<tr>
<td>0x00000008</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x0000000c</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000010</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000014</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000018</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x0000001c</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000020</td>
<td>0xE3A00000</td>
<td>-476053504</td>
</tr>
<tr>
<td>0x00000024</td>
<td>0xE2900001</td>
<td>-493879295</td>
</tr>
<tr>
<td>0x00000028</td>
<td>0x1AFFFFFD</td>
<td>452984829</td>
</tr>
<tr>
<td>0x0000002c</td>
<td>0x00000000</td>
<td>0</td>
</tr>
</tbody>
</table>

You can even assemble and run this program.

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000020</td>
<td>0xE3A00000</td>
<td>.word 0xE3A00000, 0xE2900001, 0x1AFFFFFD ; [MOV R0,#0]</td>
</tr>
<tr>
<td>0x00000024</td>
<td>0xE2900001</td>
<td>[ADDS R0,R0,#1]</td>
</tr>
<tr>
<td>0x00000028</td>
<td>0x1AFFFFFD</td>
<td>[BNE -.4]</td>
</tr>
<tr>
<td>0x0000002c</td>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>
```
**WHAT AN ASSEMBLER DOES**

Assembly is just a recipe for sequentially filling memory locations.

```
.word   0x03ffffffc, 0x00000020
.space  6
main:   mov     r0,#0
loop:   adds    r0,r0,#1
        bne     loop
        andeq   r0,r0,r0
```

And this recipe is equivalent to the first:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000020</td>
<td>0xE3A00000</td>
<td>main: mov r0,#0</td>
</tr>
<tr>
<td>0x00000024</td>
<td>0xE2900001</td>
<td>loop: adds r0,r0,#1</td>
</tr>
<tr>
<td>0x00000028</td>
<td>0x1AFFFFFFD</td>
<td>bne loop</td>
</tr>
<tr>
<td>0x0000002C</td>
<td>0x00000000</td>
<td>andeq r0,r0,r0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>In decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0x03FFFFFFC</td>
<td>67108860</td>
</tr>
<tr>
<td>0x00000004</td>
<td>0x00000020</td>
<td>32</td>
</tr>
<tr>
<td>0x00000008</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x0000000C</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000010</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000014</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000018</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x0000001C</td>
<td>0x00000000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000020</td>
<td>0xE3A00000</td>
<td>-476053504</td>
</tr>
<tr>
<td>0x00000024</td>
<td>0xE2900001</td>
<td>-493879295</td>
</tr>
<tr>
<td>0x00000028</td>
<td>0x1AFFFFFFD</td>
<td>452984829</td>
</tr>
<tr>
<td>0x0000002C</td>
<td>0x00000000</td>
<td>0</td>
</tr>
</tbody>
</table>
# 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
.align 4
.block main

.word 0x03fffffffc, 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,lsl #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
"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.

<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 0x03fffffff, 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>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>8</td>
</tr>
<tr>
<td>total</td>
<td>52</td>
</tr>
<tr>
<td>main</td>
<td>56</td>
</tr>
<tr>
<td>loop</td>
<td>76</td>
</tr>
<tr>
<td>test</td>
<td>92</td>
</tr>
<tr>
<td>halt</td>
<td>104</td>
</tr>
</tbody>
</table>
"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 or unresolved addresses (forward references) are "back-filled" once they are resolved.

State of the symbol table after the instruction `str r3, [r1,r2,lsl #2]` is assembled

<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>
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**

To handle this an object file includes a symbol table with:
1) unresolved references
2) Addresses of labels declared to be "global" (i.e. accessible to other object modules).
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.g. .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]
  ```

Two things:
1) This is the first time we've seen the IP (r12) register used
2) At the mov instruction the PC is pointing to the instruction after the ldr

How does dynamic linking work?

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

Java program

Compiler

JVM bytecodes

"Library Routines"

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
Modern Languages

Intermediate “object code language”

High-level, portable (architecture independent) program description

Java program

Compiler

JVM bytecodes

“Library Routines”

JIT Compiler

Machine code

Today’s JITs are nearly as fast as a native compiled code.

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.
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

- Play with the ARM compiler
- Compiler code optimization
- We look deeper into the Rabbit hole