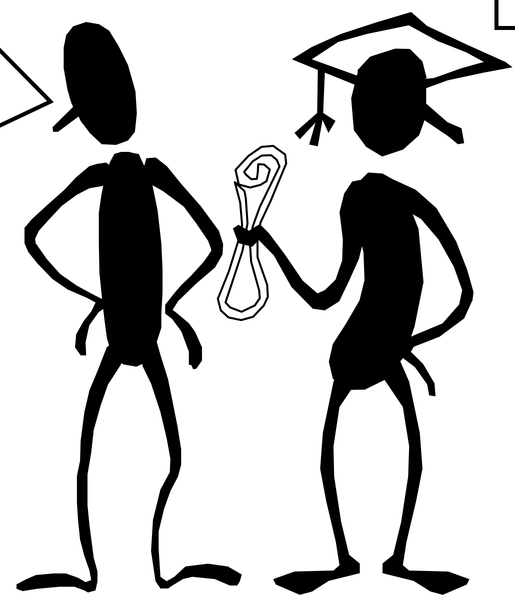
Assemblers and Linkers

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...

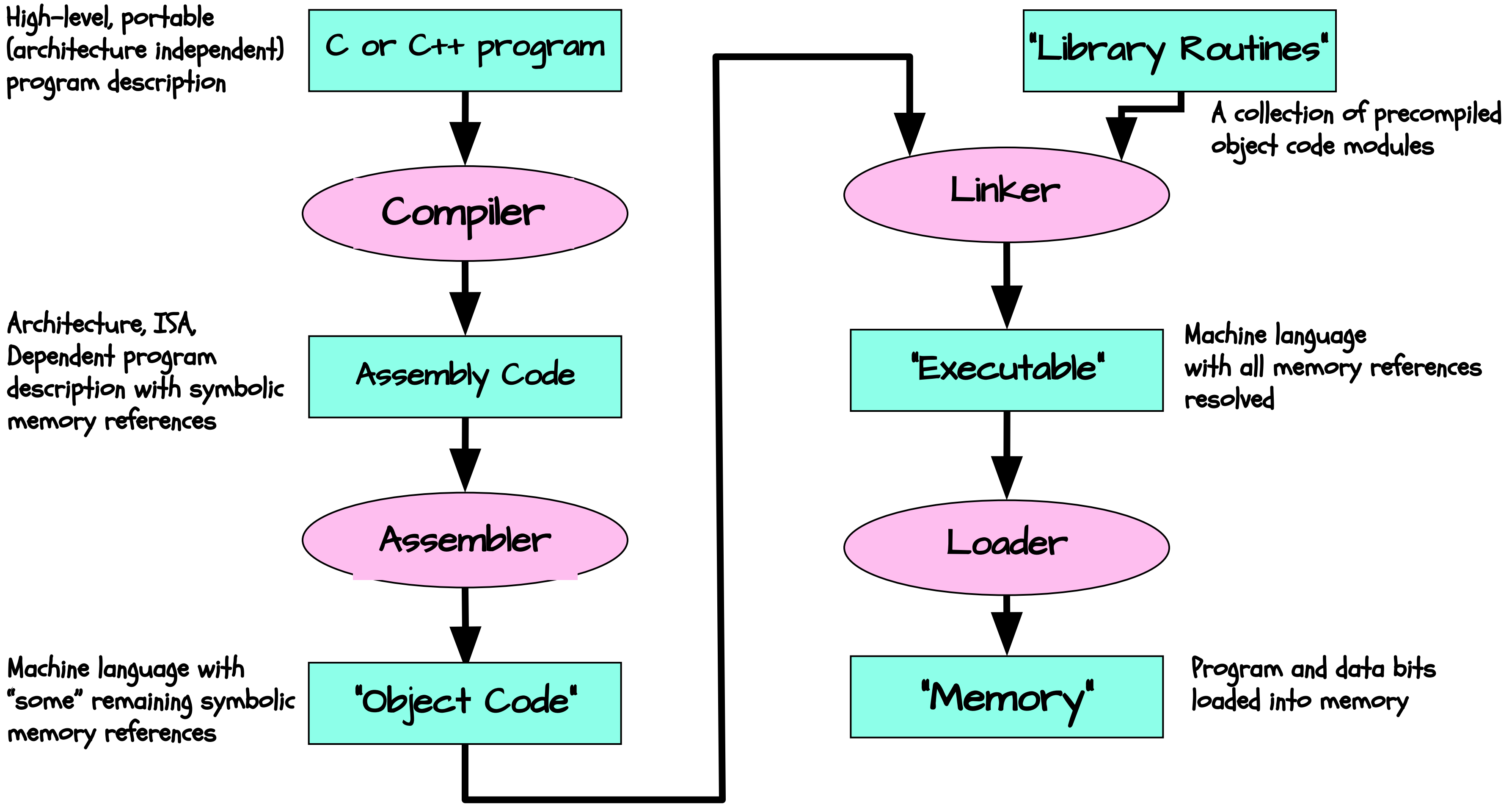
When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."



- 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



What an Assembler does

Assembly is just a recipe for sequentually filling memory locations.

```
.word   0x03fffffc, 0x00000020
.space  6
.word   0xE3A00000, 0xE2900001, 0x1AFFFFFD
```

Address		Contents	in decimal
0x00000000	:	0x03FFFFFC	67108860
0x00000004	:	0x00000020	32
0x00000008	:	0x00000000	0
0x0000000C	:	0x00000000	0
0x00000010	:	0x00000000	0
0x00000014	:	0x00000000	0
0x00000018	:	0x00000000	0
0x0000001C	:	0x00000000	0
0x00000020	:	0xE3A00000	-476053504
0x00000024	:	0xE2900001	-493879295
0x00000028	:	0x1AFFFFFD	452984829
0x0000002C	:	0x00000000	0

Address	Contents	Instruction
0x00000020	0xE3A00000	.word 0xE3A00000, 0xE2900001, 0x1AFFFFFD ; [MOV R0,#0]
0x00000024	0xE2900001	[ADDS R0,R0,#1]
0x00000028	0x1AFFFFFD	[BNE .-4]
0x0000002C	0x00000000	

What an Assembler does

Assembly is just a recipe for sequentually filling memory locations.

```
.word   0x03fffffc, 0x00000020
.space  6
main:   mov     r0,#0
loop:   adds    r0,r0,#1
        bne     loop
        andeq   r0,r0,r0
```

```
Address         Contents       in decimal
0x00000000 : 0x03FFFFFC      67108860
0x00000004 : 0x00000020            32
0x00000008 : 0x00000000             0
0x0000000C : 0x00000000             0
0x00000010 : 0x00000000             0
0x00000014 : 0x00000000             0
0x00000018 : 0x00000000             0
0x0000001C : 0x00000000             0
0x00000020 : 0xE3A00000    -476053504
0x00000024 : 0xE2900001    -493879295
0x00000028 : 0x1AFFFFFD     452984829
0x0000002C : 0x00000000             0
```

And this recipe is equivalent to the first

Address	Contents	Instruction
0x00000020	0xE3A00000	main: mov r0,#0
0x00000024	0xE2900001	loop: adds r0,r0,#1
0x00000028	0x1AFFFFFD	bne loop
0x0000002C	0x00000000	andeq r0,r0,r0

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

```
        .word   0x03fffffc, 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
```

So is this (→ `main` in `.word 0x03fffffc, main`)

Need to figure out this immediate value (→ `r1,#array`)

This one is a PC-relative offset (→ `ldr r0,total`)

This is a forward reference (→ `b test`)

This offset is completely different than the one a few instructions ago (→ `str r0,total`)

Resolving Addresses– 1st Pass

"Old-style" 2-pass assembler approach

Address	Machine code	Assembly code		
0	0x03FFFFFC		.word	0x03fffffc, main
4	0x00000000			
8		array:	.space	11
52	0x00000000	total:	.word	0
56	0xE3A01000	main:	mov	r1,#array
60	0xE3A02000		mov	r2,#0
64	0xE3A03001		mov	r3,#1
68	0xE51F0000		ldr	r0,total
72	0xEA000000		b	test
76	0xE0800003	loop:	add	r0,r0,r3
80	0xE7813102		str	r3,[r1,r2,lsl #2]
84	0xE0833003		add	r3,r3,r3
88	0xE2822001		add	r2,r2,#1
92	0xE352000B	test:	cmp	r2,#11
96	0xBA000000		blt	loop
100	0xE50F0000		str	r0,total
104	0xEA000000	*halt:	b	halt

- 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

Symbol	Address
array	8
total	52
main	56
loop	76
test	92
halt	104

Resolving Addresses in 2nd pass

"Old-style" 2-pass assembler approach

Address	Machine code	Assembly code		
0	0x03FFFFFC		.word	0x03fffffc, main
4	0x00000038			
8		array:	.space	11
52	0x00000000	total:	.word	0
56	0xE3A01008	main:	mov	r1,#array
60	0xE3A02000		mov	r2,#0
64	0xE3A03001		mov	r3,#1
68	0xE51F0018		ldr	r0,total
72	0xEA000003		b	test
76	0xE0800003	loop:	add	r0,r0,r3
80	0xE7813102		str	r3,[r1,r2,lsl #2]
84	0xE0833003		add	r3,r3,r3
88	0xE2822001		add	r2,r2,#1
92	0xE352000B	test:	cmp	r2,#11
96	0xBAFFFFF9		blt	loop
100	0xE50F0038		str	r0,total
104	0xEAFFFFFE	*halt:	b	halt

- 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

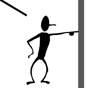
Symbol	Address
array	8
total	52
main	56
loop	76
test	92
halt	104

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



Symbol	Address	Resolved?	Reference list
array	8	y	56
total	52	y	68
main	56	y	4
loop	76	y	?
test	?	n	72

Role of a Linker

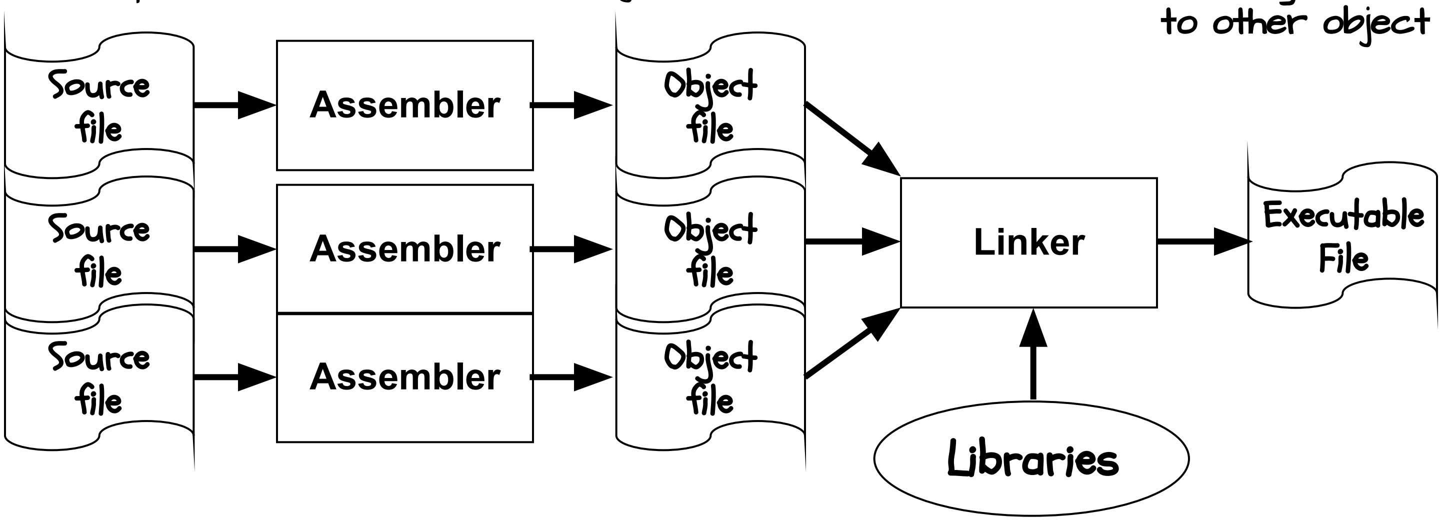
Some aspects of address resolution cannot be handled by the assembler alone.

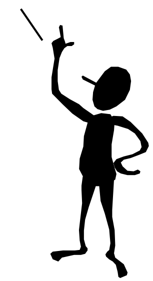
1. References to data or routines in other object modules
2. The layout of all segments in memory
3. Support for **REUSABLE** code modules
4. Support for **RELOCATABLE** code modules

This final step of resolution is the job of a **LINKER**

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.x. .dll).

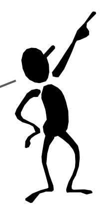
Dynamically Linked Libraries

- C call to library function:

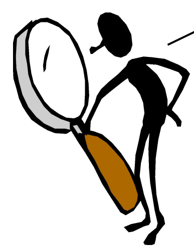
```
printf("sqr[%d] = %d\n", x, y);
```

- Assembly code

```
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



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

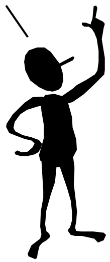
How does dynamic linking work?

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
```

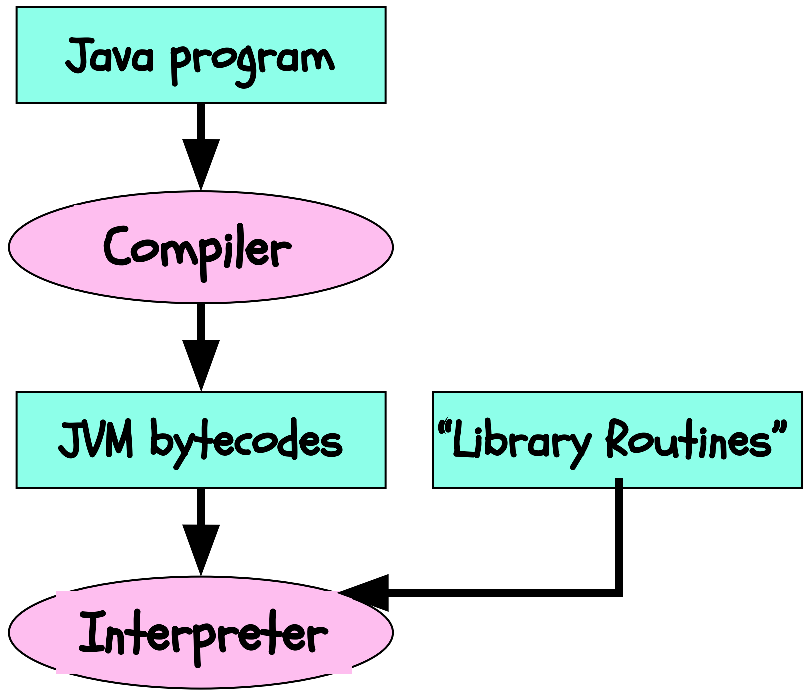
Modern Languages

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



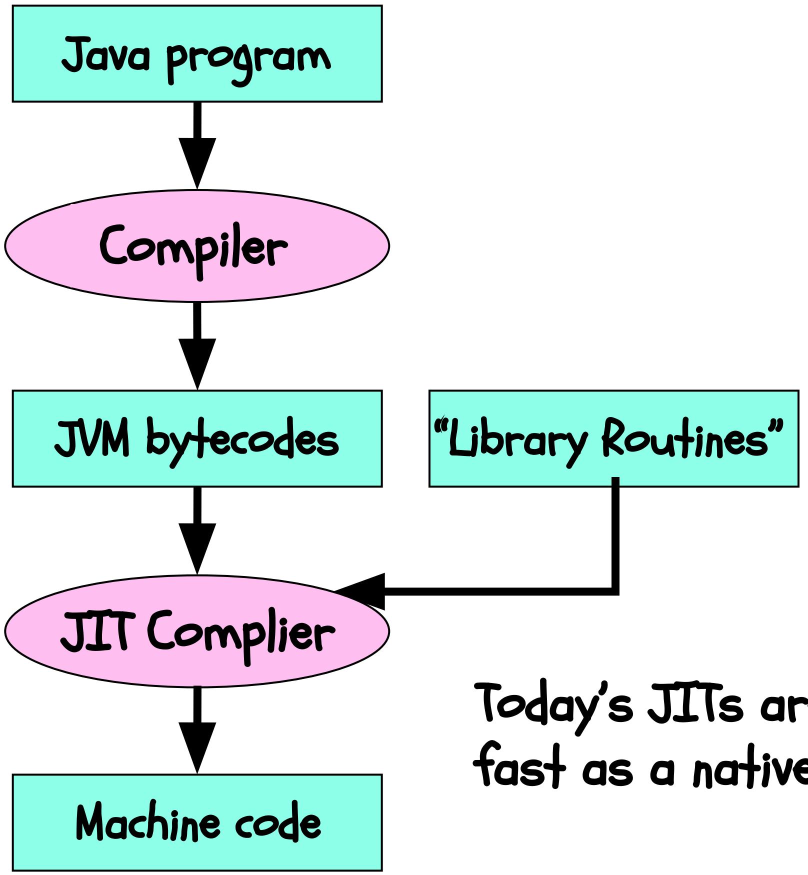
Modern Languages

Intermediate "object code language"

High-level, portable (architecture independent) program description

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

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