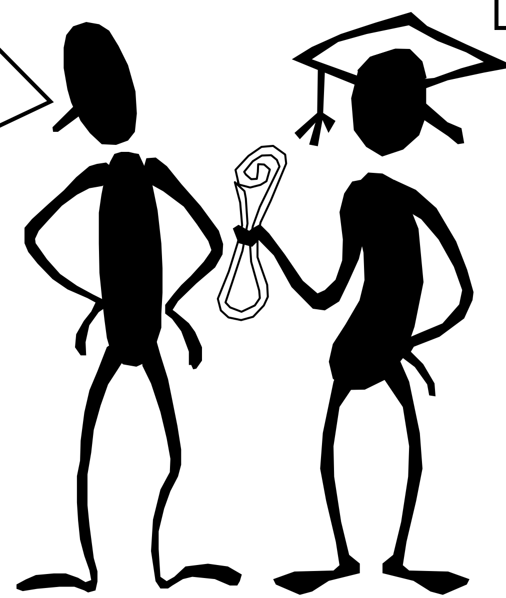
Assemblers and Linkers

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."

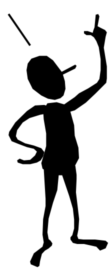
- Stay tuned for updates to problem 4 of Problem set #2

LUI TRICKS

There is a subtle trick required to load large constants using LUI/ADDI combinations. Recall the ADDI *always* sign extends its immediate argument:

```
# load t0 with 0x01234567
lui     t0,0x01234
addi    t0,t0,0x567
```

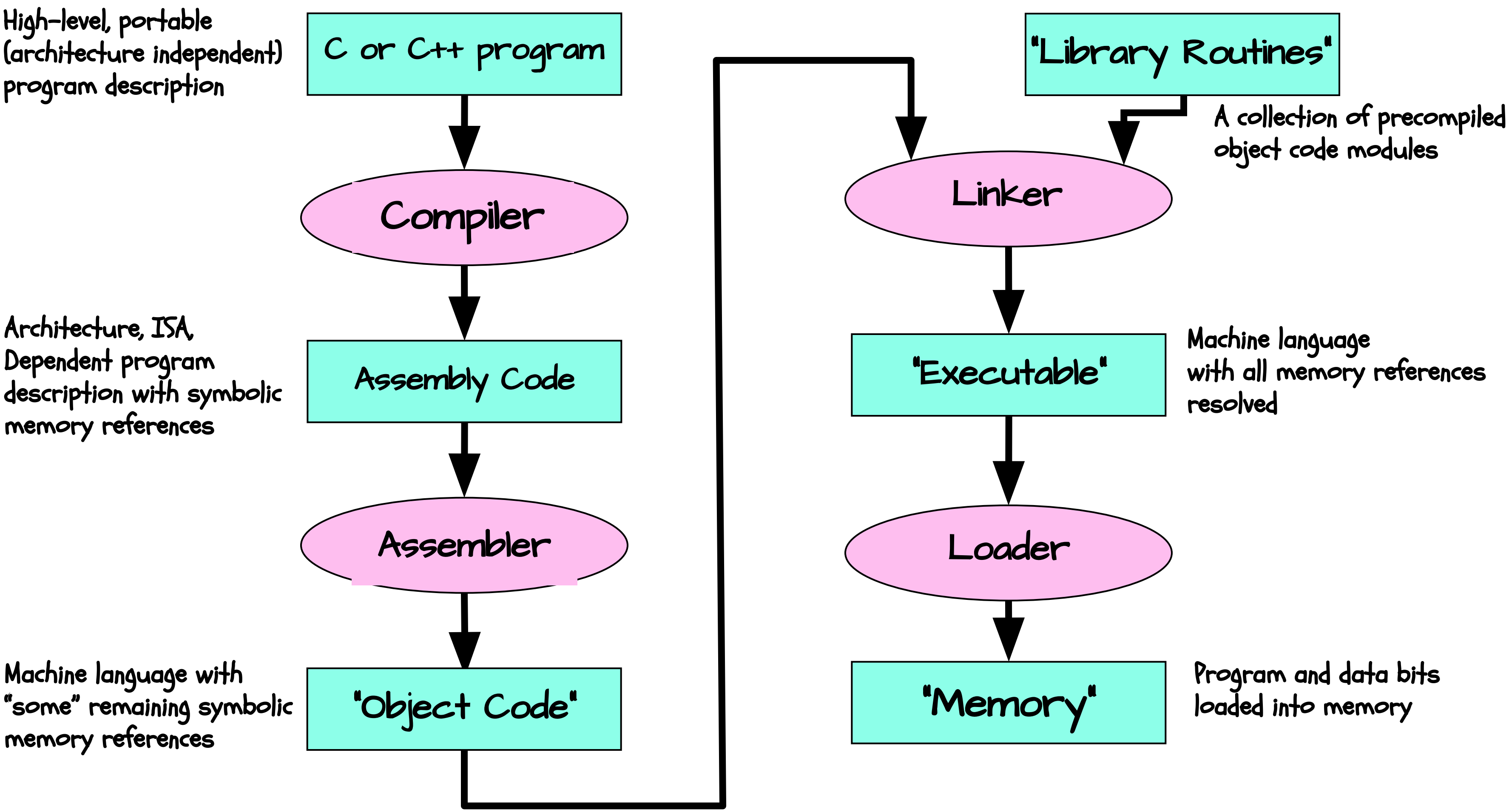
```
# load t0 with 0x89ABCDEF
lui     t0,0x89ABD
addi    t0,t0,0xDEF
```




```
After lui    t0:0x89ABD 000
addi             0xFFFFF DEF
```

A Route from Program to Bits

- Traditional Compilation



What an Assembler does

Assembly is just a recipe for sequentially filling memory locations.

```
.word   0x01000293, 0xFFC28293
.word   0x0052AA23, 0xFE029CE3
.word   0x0000006F
.space  4
```

Address	Contents	
0x00000000	0x01000293	16777875
0x00000004	0xFFC28293	-4029805
0x00000008	0x0052AA23	5417507
0x0000000C	0xFE029CE3	-33383197
0x00000010	0x0000006F	111
0x00000014	0x00000000	0
0x00000018	0x00000000	0
0x0000001C	0x00000000	0
0x00000020	0x00000000	0

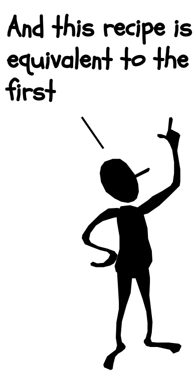
Address	Contents	Instruction
0x00000000	0x01000293	.word 0x01000293, 0xFFC28293 # [addi x5,x0,16]
0x00000004	0xFFC28293	.word 0x01000293, 0xFFC28293 # [addi x5,x5,-4]
0x00000008	0x0052AA23	.word 0x0052AA23, 0xFE029CE3 # [sw x5,20(x5)]
0x0000000C	0xFE029CE3	.word 0x0052AA23, 0xFE029CE3 # [bne x5,x0,.-8]

What an Assembler does

Assembly is just a recipe for sequentially filling memory locations.

```
main:   li      t0,16
loop:   addi    t0,t0,-4
        sw      t0,a(t0)
        bne     t0,x0,loop
halt:   j       halt
a:      .space  4
```

Address	Contents	
0x00000000	0x01000293	16777875
0x00000004	0xFFC28293	-4029805
0x00000008	0x0052AA23	5417507
0x0000000C	0xFE029CE3	-33383197
0x00000010	0x0000006F	111
0x00000014	0x00000000	0
0x00000018	0x00000000	0
0x0000001C	0x00000000	0
0x00000020	0x00000000	0



Address	Contents	Instruction
0x00000000	0x01000293	main: li t0,16
0x00000004	0xFFC28293	loop: addi t0,t0,-4
0x00000008	0x0052AA23	sw t0,a(t0)
0x0000000C	0xFE029CE3	bne t0,x0,loop

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

```
reset:   j       main
array:   .space  11
total:   .word   0

main:    li      t2,0
         li      t3,1
         li      t4,11
         lw      t0,total
         j       test
loop:    add     t0,t0,t3
         slli    t5,t2,2
         sw      t3,array(t5)
         add     t3,t3,t3
         addi    t2,t2,1
test:    blt     t2,t4,loop
         sw      t0,total
*halt:   j       halt
```

So is this

This one is a PC-relative offset

This is a forward reference

Need to figure out this immediate value

This offset is completely different than the one a few instructions ago

Resolving Addresses– 1st Pass

"Old-style" 2-pass assembler approach

Address	Machine Code	Assembly Code		
0	0x0000006f	reset:	j	main
4	0x00000000	array:	.space	11
48	0x00000000	total:	.word	0
52	0x00000393	main:	li	t2,0
56	0x00100E13		li	t3,1
60	0x00B00E93		li	t4,11
64	0x00002283		lw	t0,total
68	0x0000006f		j	test
72	0x01C282B3	loop:	add	t0,t0,t3
76	0x00239F13		slli	t5,t2,2
80	0x01CF2023		sw	t3,array(t5)
84	0x01CE0E33		add	t3,t3,t3
88	0x00138393		addi	t2,t2,1
92	0x01D3C063	test:	blt	t2,t4,loop
96	0x00502023		sw	t0,total
100	0x0000006F	*halt:	j	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	Location
reset	0
array	4
total	48
main	52
loop	72
test	92
halt	100

Resolving Addresses in 2ND pass

"Old-style" 2-pass assembler approach

Address	Machine Code	Assembly Code
0	0x3400006f	reset: j main
4	0x00000000	array: .space 11
48	0x00000000	total: .word 0
52	0x00000393	main: li t2,0
56	0x00100E13	li t3,1
60	0x00B00E93	li t4,11
64	0x30002283	lw t0,total
68	0x1800006f	j test
72	0x01C282B3	loop: add t0,t0,t3
76	0x00239F13	slli t5,t2,2
80	0x01CF2223	sw t3,array(t5)
84	0x01CE0E33	add t3,t3,t3
88	0x00138393	addi t2,t2,1
92	0xFFD3C6E3	test: blt t2,t4,loop
96	0x02502823	sw t0,total
100	0x0000006F	*halt: j 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

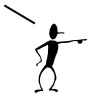
Symbol	Address
reset	0x00000000 (0)
array	0x00000004 (4)
total	0x00000030 (48)
main	0x00000034 (52)
loop	0x00000048 (72)
test	0x0000005C (92)
halt	0x00000064 (100)

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 sw t3, array(t5) is assembled



Symbol	Address	Resolved?	Reference List
reset	0x00000000 (0)	Y	0
array	0x00000004 (4)	Y	80
total	0x00000030 (48)	Y	64,?
main	0x00000034 (52)	Y	0
loop	0x00000048 (72)	Y	?
test	?	N	68
halt	?	N	?

Role of a Linker

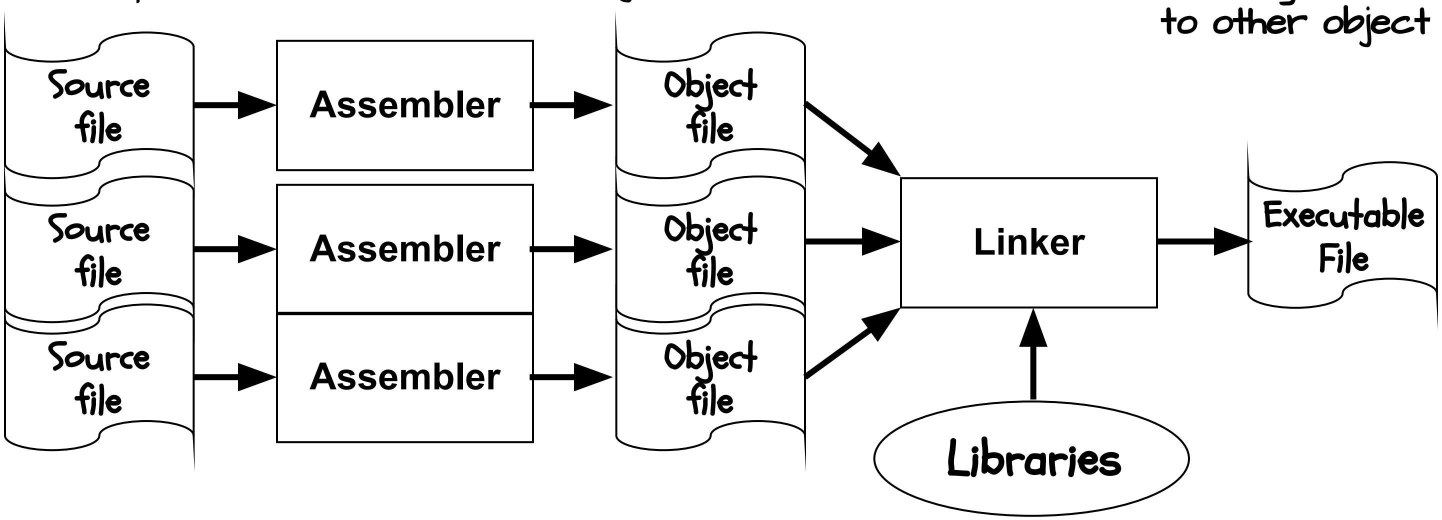
Some aspects of address resolution cannot be handled by the assembler alone.

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

```
li      a0,#1
li      a1,ctrlstring
lw      a2,x
lw      a3,y
auipc   r31,__stdio__
addi    r31,r31,__stdio__
jalr    ra,16(r31)
```

How does dynamic linking work?

Two things:
1) Calling a function using a pointer
2) There is a table of library entry points located at known fixed offsets from the library's index

Dynamically Linked Libraries

- Lazy address resolution:

```
sysload: addi  sp,sp,-4
         sw    ra,(sp)
         .
         .
         # check if stdio module
         # is loaded, if not load it
         .
         .
         # backpatch jump table
         la   t1,__stdio__
         la   t0,dfopen
         sw   t0,(t1)
         la   t0,dfclose
         sw   t0,4(t1)
         la   t0,dfputc
         sw   t0,8(t1)
         la   t0,dfgetc
         sw   t0,12(t1)
         la   t0,dfprintf
         sw   t0,16(t1)
```

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
```

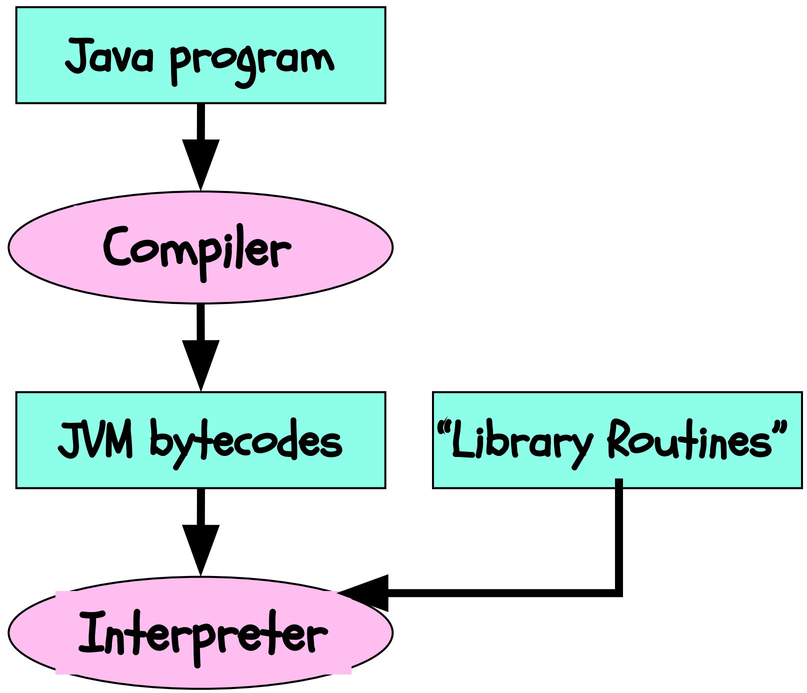
Modern Languages

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



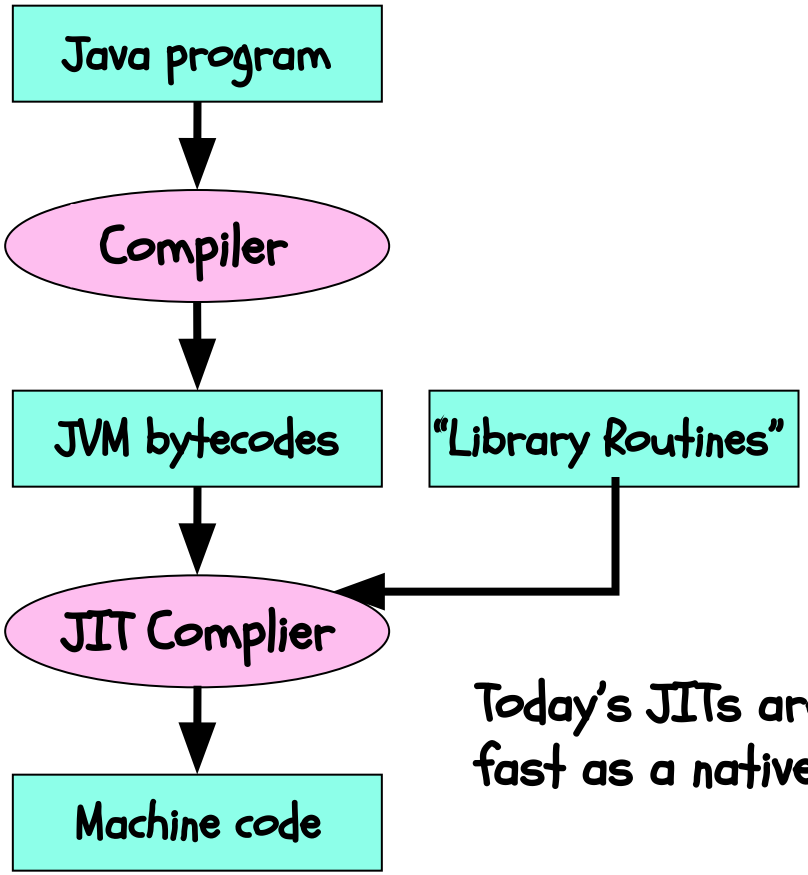
Modern Languages

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 RISC-V compiler
- Compiler code optimization
- We look deeper into the Rabbit hole