Long, long, time ago, I can still remember
How mnemonics used to make me smile...
Cause I knew that with 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 door step,
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."

Study sections 2.10-2.15
Path from Programs to Bits

- Traditional Compilation

C or C++ program

Compiler

Assembly Code

Assembler

“Object Code”

Linker

“Executable”

Loader

“Memory”

“Library Routines”

A collection of precompiled object code modules

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

```
.data
array: .space 40
total: .word 0

.text
.globl main
main:
    la    $t1, array
    move  $t2, $0
    move  $t3, $0
    beq   $0, $0, test
loop:
    sll   $t0, $t3, 2
    add   $t0, $t1, $t0
    sw    $t3, ($t0)
    add   $t2, $t2, $t3
    addi  $t3, $t3, 1
test:
    slti  $t0, $t3, 10
    bne   $t0, $0, loop
    sw    $t2, total
jr     $ra
```

$lui$ $9$, arrayhi
ori $9$, $9$, arraylo
0x3c09????
0x3529????
Resolving Addresses- 1\textsuperscript{st} Pass

- "Old-style" 2-pass assembler approach

<table>
<thead>
<tr>
<th>Segment offset</th>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x3c090000</td>
<td>la $t1, array</td>
</tr>
<tr>
<td>4</td>
<td>0x35290000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0x00005021</td>
<td>move $t2, $</td>
</tr>
<tr>
<td>12</td>
<td>0x00005821</td>
<td>move $t3, $0</td>
</tr>
<tr>
<td>16</td>
<td>0x10000000</td>
<td>beq $0, $0, test</td>
</tr>
<tr>
<td>20</td>
<td>0x000b4080</td>
<td>loop: sll $t0, $</td>
</tr>
<tr>
<td>24</td>
<td>0x01284020</td>
<td>add $t0, $t1, $t0</td>
</tr>
<tr>
<td>28</td>
<td>0xad0b0000</td>
<td>sw $t0, ($t0)</td>
</tr>
<tr>
<td>32</td>
<td>0x014b5020</td>
<td>add $t0, $t1, $t0</td>
</tr>
<tr>
<td>36</td>
<td>0x216b0001</td>
<td>addi $t3, $t3, 1</td>
</tr>
<tr>
<td>40</td>
<td>0x2968000a</td>
<td>test: slti $t0, $</td>
</tr>
<tr>
<td>44</td>
<td>0x15000000</td>
<td>bne $t0, $0, loop</td>
</tr>
<tr>
<td>48</td>
<td>0x3c010000</td>
<td>sw $t2, total</td>
</tr>
<tr>
<td>52</td>
<td>0xac2a0000</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0x03e00008</td>
<td>j $ra</td>
</tr>
</tbody>
</table>

- In the first pass, data and instructions are encoded and assigned offsets within their segment, while the symbol table is constructed.
- Unresolved address references are set to 0

Symbol table after Pass 1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Segment</th>
<th>Location pointer offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>data</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>data</td>
<td>40</td>
</tr>
<tr>
<td>main</td>
<td>text</td>
<td>0</td>
</tr>
<tr>
<td>loop</td>
<td>text</td>
<td>20</td>
</tr>
<tr>
<td>test</td>
<td>text</td>
<td>40</td>
</tr>
</tbody>
</table>
Resolving Addresses in 2 Passes

- “Old-style” 2-pass assembler approach

**Symbol table after Pass 1**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Segment</th>
<th>Location pointer offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>data</td>
<td>0</td>
</tr>
<tr>
<td>total</td>
<td>data</td>
<td>40</td>
</tr>
<tr>
<td>main</td>
<td>text</td>
<td>0</td>
</tr>
<tr>
<td>loop</td>
<td>text</td>
<td>20</td>
</tr>
<tr>
<td>test</td>
<td>text</td>
<td>40</td>
</tr>
</tbody>
</table>

---

Resolving Addresses in 2 Passes

**Pass 2**

<table>
<thead>
<tr>
<th>Segment offset</th>
<th>Code</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x3c091001</td>
<td>la $t1,array</td>
</tr>
<tr>
<td>4</td>
<td>0x35290000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0x00005021</td>
<td>move $t2,$</td>
</tr>
<tr>
<td>12</td>
<td>0x00005821</td>
<td>move $t3,$0</td>
</tr>
<tr>
<td>16</td>
<td>0x10000006</td>
<td>beq $0,$0,test</td>
</tr>
<tr>
<td>20</td>
<td>0x000b4080</td>
<td>loop:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sll $t0,$t3,2</td>
</tr>
<tr>
<td>24</td>
<td>0x01284020</td>
<td>add $t0,$t1,$t0</td>
</tr>
<tr>
<td>28</td>
<td>0xad0b0000</td>
<td>sw $t0,($t0)</td>
</tr>
<tr>
<td>32</td>
<td>0x014b5020</td>
<td>add $t0,$t1,$t0</td>
</tr>
<tr>
<td>36</td>
<td>0x216b0001</td>
<td>addi $t3,$t3,1</td>
</tr>
<tr>
<td>40</td>
<td>0x2968000a</td>
<td>test:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>slti $t0,$t3,10</td>
</tr>
<tr>
<td>44</td>
<td>0x1500ffff</td>
<td>bne $t0,$0,loop</td>
</tr>
<tr>
<td>48</td>
<td>0x3c011001</td>
<td>sw $t2,total</td>
</tr>
<tr>
<td>52</td>
<td>0xac2a0028</td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0x03e00008</td>
<td>j $ra</td>
</tr>
</tbody>
</table>

---

In the second pass, the appropriate fields of those instructions that reference memory are filled in with the correct values if possible.
Modern Way – 1-Pass Assemblers

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>SEGMENT</th>
<th>Location pointer offset</th>
<th>Resolved?</th>
<th>Reference list</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>data</td>
<td>0</td>
<td>y</td>
<td>null</td>
</tr>
<tr>
<td>total</td>
<td>data</td>
<td>40</td>
<td>y</td>
<td>null</td>
</tr>
<tr>
<td>main</td>
<td>text</td>
<td>0</td>
<td>y</td>
<td>null</td>
</tr>
<tr>
<td>loop</td>
<td>text</td>
<td>20</td>
<td>y</td>
<td>null</td>
</tr>
<tr>
<td>test</td>
<td>text</td>
<td>?</td>
<td>n</td>
<td>16</td>
</tr>
</tbody>
</table>

State of the symbol table after the instruction `sw $t0, ($t0)` is assembled.
The 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 (.text, .data) 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.g. .dll).
Dynamically Linked Libraries

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

- **Assembly code**

  ```asm
  addi $a0, $0, 1
  la $a1, ctrlstring
  lw $a2, x
  lw $a3, y
  call fprintf
  ```

- **Maps to:**

  ```asm
  addi $a0, $0, 1
  lui $a1, ctrlstringHi
  ori $a1, ctrlstringLo
  lui $at, globaldata
  lw $a2, x($at)
  lw $a3, y($at)
  lui $at, fprintfHi
  lw $at, fprintfLo($at)
  jalr $at, $31
  ```

How does dynamic linking work?

Yet another pseudoinstruction

Why are we loading the function’s address into a register first, and then calling it?
Dynamically Linked Libraries

• Lazy address resolution:

```assembly
sysload:    addui $sp,$sp,16

.globl stdio:
stdio:
fopen:     dfopen
fclose:    dclose
fgetc:     dfgetc
fputc:     dfputc
fprintf:   dprintf
```

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”

```assembly
.globl stdio:
stdio:
fopen:     .word sysload
fclose:    .word sysload
fgetc:     .word sysload
fputc:     .word sysload
fprintf:   .word sysload
```

• After first call is made to any procedure in “stdio.dll”

```assembly
.globl stdio:
stdio:
fopen:     dfopen
close:     dclose
getc:      dfgetc
putc:      dfputc
printf:    dprintf
```
Modern Languages

- Intermediate “object code language”

High-level, portable (architecture independent) program description

Java program

Compiler

JVM bytecodes

“Library Routines”

Interpreter

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”

Java program

Compiler

JVM bytecodes

“Library Routines”

JIT Compiler

“Memory”

High-level, portable (architecture independent) program description

PORTABLE mnemonic program description with symbolic memory references

While interpreting on the first pass it 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 (ex. .NET).
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
Example Compiler Optimizations

- Example “C” Code:

```c
int array[10];
int total;

int main( ) {
    int i;

    total = 0;
    for (i = 0; i < 10; i++) {
        array[i] = i;
        total = total + i;
    }
}
```
Unoptimized Assembly Output

With debug flags set:

```assembly
.globl main
.text
main:
    addiu $sp,$sp,-8       # allocates space for ra and i
    sw $0,total            # total = 0
    sw $0,0($sp)           # i = 0
    lw $8,0($sp)           # copy i to $t0
    b  L.3                 # goto test
L.2:                        # for(...) {
    sll $24,$8,2           # make i a word offset
    sw $8,array($24)       # array[i] = i
    lw $24,total           # total = total + i
    addu $24,$24,$8        # i = i + 1
L.3:                        # update i in memory
    sw $8,0($sp)           # update i in memory
    slti $1,$8,10          # (i < 10)?
    bne $1,$0,L.2          #} if TRUE loop
    addiu $sp,$sp,8        # update i in memory
jr $31                   # update i in memory
```

Why does turning on debugging generate the worse code?
Ans: Because the compiler reverts back to line-by-line translation.

103, that's not so bad
Register Allocation

- Assign local variable “i” to a register

```assembly
.globl main
.text
main:
    addiu $sp,$sp,-4    #allocates space for ra
    sw $0,total        #total = 0
    move $8,$0         #i = 0
    b L.3               #goto test
L.2:
    sll $24,$8,2        # make i a word offset
    sw $8,array($24)   # array[i] = i
    lw $24,total       # total = total + i
    addu $24,$24,$8
    sw $24,total
    addi $8,$8,1        # i = i + 1
L.3:
    slti $1,$8,10       # (i < 10)?
    bne $1,$0,L.2      #} if TRUE loop
    addiu $sp,$sp,4
    jr $31
```

Two instructions outside the loop are replaced with one

91, I can play in public.
Loop-Invariant Code Motion

- Temporarily allocate temp registers to hold global values to avoid loads inside the loop, yet mirroring changes.

```
.globl main
.text
main:
    addiu $sp,$sp,-4       #allocates space for ra
    sw $0,total           #total = 0
    move $9,$0             #temp for total
    move $8,$0             #i = 0
    b  L.3                 #goto test
L.2:                        #for(...) {
    sll $24,$8,2           #  make i a word offset
    sw $8,array($24)       #  array[i] = i
    addu $9,$9,$8          #  i = i + 1
    sw $9,total
L.3:                        #} if TRUE loop
    slti $1,$8,10          # (i < 10)?
    bne $1,$0,L.2          #} if TRUE loop
    addiu $sp,$sp,4        #make i a word offset
    jr $31                #array[i] = i
```

We've added an instruction here outside of the loop and eliminated an `lw` inside of loop.

82! Side-bets anyone?
Remove Unnecessary Tests

Since “i” is initially set to “0”, we already know it is less than “10”, so why bother testing it the first time?

```
.globl main
.text
main:
    addiu $sp,$sp,-4       #allocates space for ra
    sw $0,total           #total = 0
    move $9,$0             #temp for total
    move $8,$0             #i = 0
L.2:                        #for(...) {
    sll $24,$8,2           #  make i a word offset
    sw $8,array($24)       #  array[i] = i
    addu $9,$9,$8          #i = i + 1
    sw $9,total
    slti $1,$8,10          #loads const 10
    bne $1,$0,L.2          #} loops while i < 10
    addiu $sp,$sp,4
    jr $31
```

Eliminated a branch here and the label it referenced.
Remove Unnecessary Stores

- All we care about it the value of total after the loop finishes, so there is no need to update it on each pass

```assembly
.globl main
.text
main:
    addiu $sp,$sp,-4       #allocates space for ra and i
    sw $0,total            #total = 0
    move $9,$0             #temp for total
    move $8,$0             #i = 0
L.2:
    sll $24,$8,2           #for(...) {
    sw $8,array($24)       #  array[i] = i
    addu $9,$9,$8          # i = i + 1
    addi $8,$8,1           # loads const 10
    slti $1,$8,10          #} loops while i < 10
    bne $1,$0,L.2          #}
    sw  $9,total
    addiu $sp,$sp,4
jr $31
```

70, ready for the PGA!
Unrolling Loops

- By examining the function we can see it is always executed 10 times. Thus, we can make 2, 5, or 10 copies of the inner loop reduce the branching overhead.

```assembly
.globl main
.text
main:
    addiu $sp,$sp,-4       #allocates space for ra and i
    sw $0,total            #total = 0
    move $9,$0             #temp for total
    move $8,$0             #i = 0

L.2:
    sll $24,$8,2           #for(...) {
    sw $8,array($24)       #  array[i] = i
    addu $9,$9,$8          #  i = i + 1
    addi $8,$8,1
    sll $24,$8,2
    sw $8,array($24)
    addu $9,$9,$8
    addi $8,$8,1
    slti $24,$8,10
    bne $24,$0,L.2
    sw $9,total
    addiu $sp,$sp,4
    jr $31 #} loops while i < 10
```

60, watch out Tiger!
Next Time

- We go deeper into the rabbit hole…

- Quiz on Friday
  - Multiple Choice
  - Open book/open notes
  - No computers or calculators