CALL

(Compiler/Assembler/Linker/Loader)
Integer Multiplication (1/3)

• Paper and pencil example (unsigned):

  \[
  \begin{array}{r}
  \text{Multiplicand} & 1000 & 8 \\
  \text{Multiplier} & \times 1001 & 9 \\
  \hline
  1000 \\
  0000 \\
  0000 \\
  +1000 \\
  \hline
  01001000 & 72
  \end{array}
  \]

  \[m \text{ bits} \times n \text{ bits} = m + n \text{ bit product}\]
• In RISC-V, we multiply registers, so:
  • 32-bit value x 32-bit value = 64-bit value

• Multiplication is **not** part of standard RISC-V…
  • Instead it is an *optional* extra: The compiler needs to produce a series of shifts and adds if the multiplier isn't present

• Syntax of Multiplication (signed):
  • `mul rd, rs1, rs2`
  • `mulh rd, rs1, rs2`
  • Multiplies 32-bit values in those registers and returns either the lower or upper 32b result
    • If you do mulh/mul back to back, the architecture can fuse them
  • Also unsigned versions of the above
Integer Multiplication (3/3)

- Example:
  - in C:   \( a = b \times c; \)
  - \texttt{int64_t} a; \texttt{int32_t} b, c;
  - Aside, these types are defined in C99, in \texttt{stdint.h}

- in RISC-V:
  - let \( b \) be \( s2 \); let \( c \) be \( s3 \); and let \( a \) be \( s0 \) and \( s1 \) (since it may be up to 64 bits)
  - \texttt{mulh} \( s1, s2, s3 \)
  - \texttt{mul} \( s0, s2, s3 \)
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c}
\text{1001} \\
\hline
1000 | 1001010
\end{array}
\]

\[
\begin{array}{c}
\text{Dividend} \\
-1000 \\
10 \\
101 \\
1010 \\
-1000 \\
10 \text{ Remainder} \\
\hline
\text{(or Modulo result)}
\end{array}
\]

• Dividend = Quotient \times Divisor + Remainder
Integer Division (2/2)

• Syntax of Division (signed):
  • \texttt{div rd, rs1, rs2}
  \texttt{rem rd, rs1, rs2}
  • Divides 32-bit \texttt{rs1} by 32-bit \texttt{rs2}, returns the quotient (/) for \texttt{div}, remainder (%) for \texttt{rem}
  • Again, can fuse two adjacent instructions

• Example in C: \texttt{a = c / d; b = c \% d;}

• RISC-V:
  • \texttt{a\tiny\rightarrow}s0; b\tiny\rightarrow}s1; c\tiny\rightarrow}s2; d\tiny\rightarrow}s3
  • \texttt{div s0, s2, s3}
  \texttt{rem s1, s2, s3}
Agenda

• Interpretation vs Compilation
• The CALL chain
• Producing Machine Language
Levels of Representation/Interpretation

- High Level Language Program (e.g., C)
- Assembly Language Program (e.g., RISC-V)
- Machine Language Program (RISC-V)

**Compiler**

**Assembler**

**Machine Interpretation**

**Hardware Architecture Description (e.g., block diagrams)**

**Architecture Implementation**

- Logic Circuit Description (Circuit Schematic Diagrams)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw t0, 0(a2)
lw t1, 4(a2)
sw t1, 0(a2)
sw t0, 4(a2)
```

Anything can be represented as a number, i.e., data or instructions

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

+ How to take a program and run it
Language Execution Continuum

- An **Interpreter** is a program that executes other programs.

  - **Scheme**  
  - **Java**  
  - **C++**  
  - **C**  
  - **Assembly**  
  - **Java bytecode**  
  - **Machine code**

  - Easy to program  
  - Inefficient to interpret  
  - Difficult to program  
  - Efficient to interpret

- Language **translation** gives us another option

- In general, we **interpret** a high-level language when efficiency is not critical and **translate** to a lower-level language to increase performance

- Although this is becoming a “distinction without a difference”
  Many interpreters do a “just in time” runtime compilation to bytecode that either is emulated or directly compiled to machine code (e.g. LLVM)
Interpretation vs Translation

• How do we run a program written in a source language?
  • **Interpreter**: Directly executes a program in the source language
  • **Translator**: Converts a program from the source language to an equivalent program in another language

• For example, consider a Python program `foo.py`
Python interpreter is just a program that reads a python program and performs the functions of that python program.

Well, that’s an exaggeration, the interpreter converts to a simple bytecode that the interpreter runs… Saved copies end up in .pyc files.
Interpretation

• Any good reason to interpret machine language in software?

• Simulators: Useful for learning / debugging

• Apple Macintosh conversion
  • Switched from Motorola 680x0 instruction architecture to PowerPC.
    • Similar issue with switch to x86
  • Could require all programs to be re-translated from high level language
  • Instead, let executables contain old and/or new machine code, interpret old code in software if necessary (emulation)
Interpretation vs. Translation? (1/2)

• Generally easier to write interpreter
• Interpreter closer to high-level, so can give better error messages
• Translator reaction: add extra information to help debugging (line numbers, names):
  This is what `gcc -g` does, it tells the compiler to add all the debugging information
• Interpreter slower (10x?), code smaller (2x? or not?)
• Interpreter provides instruction set independence: run on any machine
Interpretation vs. Translation? (2/2)

- Translated/compiled code almost always more efficient and therefore higher performance:
  - Important for many applications, particularly operating systems.
- Compiled code does the hard work once: during compilation
  - Which is why most “interpreters” these days are really “just in time compilers”: don’t throw away the work processing the program
Agenda

• Interpretation vs Compilation
• The CALL chain
• Producing Machine Language
Steps Compiling a C program
Compiler

- Input: High-Level Language Code (e.g., `foo.c`)
- Output: Assembly Language Code (e.g. MAL) (e.g., `foo.s` for RISC-V)
- Code matches the calling convention for the architecture
- Note: Output may contain pseudo-instructions
- **Pseudo-instructions**: instructions that assembler understands but not in machine
  For example:
  
  ```
  j label \Rightarrow jal x0 label
  ```
Steps In The Compiler

- **Lexer:**
  - Turns the input into "tokens", recognizes problems with the tokens

- **Parser:**
  - Turns the tokens into an "Abstract Syntax Tree", recognizes problems in the program structure

- **Semantic Analysis and Optimization:**
  - Checks for semantic errors, may reorganize the code to make it better

- **Code generation:**
  - Output the assembly code
Where Are We Now?

1. C program: foo.c
2. Compiler
3. Assembly program: foo.s
4. Assembler
5. Object (mach lang module): foo.o
6. Linker
7. Executable (mach lang pgm): a.out
8. Loader
9. Memory

CS164
Assembler

- Input: Assembly Language Code (e.g. MAL) (e.g., foo.s)
- Output: Object Code, information tables (TAL) (e.g., foo.o)
- Reads and Uses Directives
- Replace Pseudo-instructions
- Produce Machine Language
- Creates Object File
Assembler Directives

- Give directions to assembler, but do not produce machine instructions
  - `.text`: Subsequent items put in user text segment (machine code)
  - `.data`: Subsequent items put in user data segment (binary rep of data in source file)
  - `.globl sym`: declares `sym` global and can be referenced from other files
  - `.string str`: Store the string `str` in memory and null-terminate it
  - `.word w1...wn`: Store the $n$ 32-bit quantities in successive memory words
### Pseudo-instruction Replacement

Assembler treats convenient variations of machine language instructions as if real instructions.

<table>
<thead>
<tr>
<th>Pseudo</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nop</code></td>
<td><code>addi x0, x0, 0</code></td>
</tr>
<tr>
<td><code>not rd, rs</code></td>
<td><code>xori rd, rs, -1</code></td>
</tr>
<tr>
<td><code>beqz rs, offset</code></td>
<td><code>beq rs, x0, offset</code></td>
</tr>
<tr>
<td><code>bgt rs, rt, offset</code></td>
<td><code>blt rt, rs, offset</code></td>
</tr>
<tr>
<td><code>j offset</code></td>
<td><code>jal x0, offset</code></td>
</tr>
<tr>
<td><code>ret</code></td>
<td><code>jalr x0, x1, offset</code></td>
</tr>
<tr>
<td><code>call offset</code></td>
<td><code>auipc x6, offset[31:12]</code></td>
</tr>
<tr>
<td></td>
<td><code>jalr x1, x6, offset[11:0]</code></td>
</tr>
<tr>
<td><code>tail offset</code></td>
<td><code>auipc x6, offset[31:12]</code></td>
</tr>
<tr>
<td></td>
<td><code>jalr x0, x6, offset[11:0]</code></td>
</tr>
</tbody>
</table>
So what is "tail" about...

- Often times your code has a convention like this:
  - `{ ...
    lots of code
    return foo(y);
  }
  - It can be a recursive call to \texttt{foo()} if this is within \texttt{foo()}, or call to a different function...

- So for efficiency...
  - Evaluate the arguments for \texttt{foo()} and place them in a0-a7...
  - Restore \texttt{ra}
  - Restore the stack and all callee saved registers
  - Then call \texttt{foo()} \textit{with} \texttt{j or tail}

- Then when \texttt{foo()} returns, it can return \textbf{directly} to where it needs to return to
  - Rather than returning to wherever \texttt{foo()} was called and returning from there
Agenda

- Interpretation vs Compilation
- The CALL chain
- Producing Machine Language
Producing Machine Language (1/3)

- **Simple Case**
  - Arithmetic, Logical, Shifts, and so on
  - All necessary info is within the instruction already
- **What about Branches?**
  - PC-Relative
  - So once pseudo-instructions are replaced by real ones, we know by how many instructions to branch
  - So these can be handled
“Forward Reference” problem

Branch instructions can refer to labels that are “forward” in the program:

```assembly
or   s0, x0, x0
L1:  slt  t0, x0,  $a1
     beq  t0, x0, L2
     addi a1, a1, -1
     jal  x0, L1
     L2:  add  $t1, $a0, $a1
```

Solved by taking 2 passes over the program

First pass remembers position of labels
Second pass uses label positions to generate code
Producing Machine Language (3/3)

- What about jumps (j and jal)?
  - Jumps within a file are PC relative (and we can easily compute)
  - Jumps to other files we can’t

- What about references to static data?
  - la gets broken up into lui and addi
  - These will require the full 32-bit address of the data
  - These can’t be determined yet, so we create two tables…
Symbol Table

• List of “items” in this file that may be used by other files

• What are they?
  • Labels: function calling
  • Data: anything in the `.data` section; variables which may be accessed across files
Relocation Table

- List of “items” this file needs the address of later
- What are they?
  - Any external label jumped to: \texttt{jal}
    - external (including lib files)
  - Any piece of data in static section
    - such as the \texttt{la} instruction
Object File Format

- **object file header**: size and position of the other pieces of the object file
- **text segment**: the machine code
- **data segment**: binary representation of the static data in the source file
- **relocation information**: identifies lines of code that need to be fixed up later
- **symbol table**: list of this file’s labels and static data that can be referenced
- **debugging information**

A standard format is ELF (except Microsoft)

http://www.skyfree.org/linux/references/ELF_Format.pdf
Linker (1/3)

- **Input:** Object code files, information tables (e.g., `foo.o`, `libc.o`)
- **Output:** Executable code (e.g., `a.out`)
- Combines several object (.o) files into a single executable (“linking”)
- Enable separate compilation of files
  - Changes to one file do not require recompilation of the whole program
    - Windows 7 source was > 40 M lines of code!
  - Old name “Link Editor” from editing the “links” in jump and link instructions
Linker (2/3)

The diagram illustrates the process of linking two object files (.o file 1 and .o file 2) to create an executable file (a.out). The linker takes the relocatable sections (text 1, data 1, info 1, text 2, data 2, info 2) from the object files and combines them into the a.out file, which contains the final relocatable text, data, and output information.
Linker (3/3)

- Step 1: Take text segment from each `.o` file and put them together
- Step 2: Take data segment from each `.o` file, put them together, and concatenate this onto end of text segments
- Step 3: Resolve references
  - Go through Relocation Table; handle each entry
  - That is, fill in all absolute addresses
Three Types of Addresses

- **PC-Relative Addressing** (beq, bne, jal)
  - never relocate
- **External Function Reference** (usually jal)
  - always relocate
- **Static Data Reference** (often auipc and addi)
  - always relocate
- RISC-V often uses auipc rather than lui so that a big block of stuff can be further relocated as long as it is fixed relative to the pc
Absolute Addresses in RISC-V

- Which instructions need relocation editing?
  - Jump and link: ONLY for external jumps
    
    \[
    \begin{array}{c|c|c}
    \text{jal} & \text{rd} & \text{xxxxx} \\
    \end{array}
    \]

- Loads and stores to variables in static area, relative to the global pointer
  
  \[
  \begin{array}{c|c|c|c}
  \text{lw/sw} & \text{gp} & \text{x?} & \text{xxxxx} \\
  \end{array}
  \]

- What about conditional branches?
  
  \[
  \begin{array}{c|c|c|c}
  \text{beq} & \text{rs} & \text{rt} & \text{xxxxxx} \\
  \end{array}
  \]

- PC-relative addressing preserved even if code moves
Resolving References (1/2)

- Linker assumes first word of first text segment is at address \(0x04000000\).
- (More later when we study “virtual memory”)
- Linker knows:
  - length of each text and data segment
  - ordering of text and data segments
- Linker calculates:
  - absolute address of each label to be jumped to and each piece of data being referenced
Resolving References (2/2)

• To resolve references:
  • search for reference (data or label) in all "user" symbol tables
  • if not found, search library files (for example, for `printf`)
  • once absolute address is determined, fill in the machine code appropriately

• Output of linker: executable file containing text and data (plus header)
In Conclusion…

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A .s file becomes a .o file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several .o files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.
Loader Basics

• Input: Executable Code (e.g., a.out)
• Output: (program is run)
• Executable files are stored on disk
• When one is run, loader’s job is to load it into memory and start it running
• In reality, loader is the operating system (OS)
  • loading is one of the OS tasks
  • And these days, the loader actually does a lot of the linking
Loader … what does it do?

- Reads executable file’s header to determine size of text and data segments
- Creates new address space for program large enough to hold text and data segments, along with a stack segment
- Copies instructions and data from executable file into the new address space
- Copies arguments passed to the program onto the stack
- Initializes machine registers
  - Most registers cleared, but stack pointer assigned address of 1st free stack location
  - Jumps to start-up routine that copies program’s arguments from stack to registers & sets the PC
  - If main routine returns, start-up routine terminates program with the exit system call
At what point in process are all the machine code bits determined for the following assembly instructions:

1) `addu x6, x7, x8`
2) `jal fprintf`

A: 1) & 2) After compilation
B: 1) After compilation, 2) After assembly
C: 1) After assembly, 2) After linking
D: 1) After compilation, 2) After linking
E: 1) After compilation, 2) After loading
Example: \(\text{C} \rightarrow \text{Asm} \rightarrow \text{Obj} \rightarrow \text{Exe} \rightarrow \text{Run}\)

C Program Source Code: \texttt{prog.c}

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i, sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("The sum of sq from 0 .. 100 is \%d\n", sum);
}
```

"\texttt{printf}" lives in "\texttt{libc}"
Compilation: MAL:
\[ i = t0, \text{sum} = a1 \]

```
.text
.align 2
.globl main
main:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv t0, x0
    mv a1, x0
    li t1, 100
    j check
loop:
    mul t2, t0, t0
    add a1, a1, t2
    addi t0, t0, 1
    check:
        blt t0, t1 loop:
        la $a0, str
        jal printf
        mv a0, x0
        lw ra, 0(sp)
        addi sp, sp 4
        ret
.data
.align 0
str:
    .asciiz "The sum of sq from 0 .. 100 is %d\n"
```

Pseudo-Instructions?
Compilation: MAL:
i = t0, sum = a1

```assembly
.text
.align 2
.globl main
main:
    addi sp, sp, -4
    sw ra, 0(sp)
    mv t0, x0
    mv a1, x0
    li t1, 100
    j check
loop:
    mul t2, t0, t0
    add a1, a1, t2
    addi t0, t0, 1
    add a1, a1, t2
    addi t0, t0, 1
    blt t0, t1 loop:
    la $a0, str
    jal printf
    mv a0, x0
    lw ra, 0(sp)
    ret
.data
.globl str
str:
    .asciiz "The sum of sq from 0 .. 100 is %d\n"
```

Pseudo-Instructions?
Underlined
Assembly step 1:
Remove Pseudo Instructions, assign jumps

```assembly
.text
.align 2
.globl main
main:
  addi sp, sp, -4
  sw ra, 0(sp)
  addi t0, x0, 0
  addi a1, x0, 0
  addi t1, x0, 100
  jal x0, 12

loop:
  mul t2, t0, t0
  add a1, a1, t2
  addi t0, t0, 1
```

**Pseudo-Instructions? Underlined**

```assembly
check:
  blt t0, t1 -16
  lui a0, l.str
  addi a0, a0, r.str
  jal printf
  mv a0, x0
  lw ra, 0(sp)
  addi sp, sp 4
  jalr x0, ra

.data
.ascii "The sum of sq from 0 .. 100 is %d\n"
```

`l.str` and `r.str` are pseudo instructions which are not to be translated to assembly code.
Assembly step 2

Create relocation table and symbol table

• Symbol Table

<table>
<thead>
<tr>
<th>Label</th>
<th>address (in module)</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td>0x0000000000</td>
<td>global text</td>
</tr>
<tr>
<td>loop:</td>
<td>0x000000014</td>
<td>local text</td>
</tr>
<tr>
<td>str:</td>
<td>0x0000000000</td>
<td>local data</td>
</tr>
</tbody>
</table>

• Relocation Information

<table>
<thead>
<tr>
<th>Address</th>
<th>Instr. type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000002c</td>
<td>lui</td>
<td>l.str</td>
</tr>
<tr>
<td>0x00000030</td>
<td>addi</td>
<td>r.str</td>
</tr>
<tr>
<td>0x00000034</td>
<td>jal</td>
<td>printf</td>
</tr>
</tbody>
</table>
Assembly step 3

• Generate object (.o) file:
  • Output binary representation for
    • text segment (instructions)
    • data segment (data)
    • symbol and relocation tables
  • Using dummy “placeholders” for unresolved absolute and external references
  • And then… We link!
Linking Just Resolves References...

• So take all the .o files
  • Squish the different segments together

• For each entry in the relocation table:
  • Replace it with the actual address for the symbol table of the item you are linking to

• Result is a single binary
  •
Static vs. Dynamically Linked Libraries

• What we’ve described is the traditional way: **statically-linked** approach
  • Library is now part of the executable, so if the library updates, we don’t get the fix (have to recompile if we have source)
  • Includes the **entire** library even if not all of it will be used
  • Executable is self-contained
• Alternative is **dynamically linked libraries** (DLL), common on Windows & UNIX platforms
Dynamically Linked Libraries

- Space/time issues
  - + Storing a program requires less disk space
  - + Sending a program requires less time
  - + Executing two programs requires less memory (if they share a library)
  - – At runtime, there’s time overhead to do link

- Upgrades
  - + Replacing one file (libXYZ.so) upgrades every program that uses library “XYZ”
  - – Having the executable isn’t enough anymore

---

Overall, dynamic linking adds quite a bit of complexity to the compiler, linker, and operating system. However, it provides many benefits that often outweigh these.
Dynamically Linked Libraries

• Prevailing approach to dynamic linking uses machine code as the “lowest common denominator”
  • Linker does not use information about how the program or library was compiled (i.e., what compiler or language)
  • Can be described as “linking at the machine code level”
  • This isn’t the only way to do it ...

• Also these days will **randomize layout** (Address Space Layout Randomization)
  • Acts as a defense to make exploiting C memory errors substantially harder, as modern exploitation requires jumping to pieces of existing code (“Return oriented programming”) to counter another defense (marking heap & stack unexecutable, so attacker can’t write code into just anywhere in memory).
Final Review C Program: Hello.c

#include <stdio.h>

int main()
{
    printf("Hello, %s\n", "world");
    return 0;
}

# Directive: enter text section
# Directive: align code to 2^2 bytes
# Directive: declare global symbol main
# label for start of main
# allocate stack frame
# save return address
# compute address of
#   string1
# compute address of
#   string2
# call function printf
# restore return address
# deallocate stack frame
# load return value 0
# return
# Directive: enter read-only data section
# Directive: align data section to 4 bytes
# label for first string
# Directive: null-terminated string
# label for second string
# Directive: null-terminated string
Assembled Hello.s: Linkable Hello.o

00000000 <main>:

0: ff010113 addi sp,sp,-16
4: 00112623 sw ra,12(sp)
8: 00000537 lui a0,0x0     # addr placeholder
c: 00050513 addi a0,a0,0  # addr placeholder
10: 000005b7 lui a1,0x0    # addr placeholder
14: 00058593 addi a1,a1,0  # addr placeholder
18: 00000097 auipc ra,0x0  # addr placeholder
1c: 000080e7 jalr ra       # addr placeholder
20: 00c12083 lw ra,12(sp)
24: 01010113 addi sp,sp,16
28: 00000513 addi a0,a0,0
2c: 00008067 jalr ra
Hello.o: a.out

000101b0 <main>:
  101b0: ff010113 addi sp,sp,-16
  101b4: 00112623 sw ra,12(sp)
  101b8: 00021537 lui a0,0x21
  101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
  101c0: 000215b7 lui a1,0x21
  101c4: a1c58593 addi a1,a1,-1508 # 20alc <string2>
  101c8: 288000ef jal ra,10450    # <printf>
  101cc: 00c12083 lw ra,12(sp)
  101d0: 01010113 addi sp,sp,16
  101d4: 00000513 addi a0,0,0
  101d8: 00008067 jalr ra