CS 61C: Great Ideas in Computer Architecture

Running a Program - CALL (Compiling, Assembling, Linking, and Loading)

Instructors:

Nick Weaver & Vladimir Stojanovic

http://inst.eecs.Berkeley.edu/~cs61c/sp16
Integer Multiplication (1/3)

• Paper and pencil example (unsigned):

  Multiplicand  1000  8
  Multiplier    x1001  9
               1000
               0000
               0000
               +1000
               01001000  72

• \( m \) bits \( \times n \) bits = \( m + n \) bit product
Integer Multiplication (2/3)

• In MIPS, we multiply registers, so:
  – 32-bit value x 32-bit value = 64-bit value

• Syntax of Multiplication (signed):
  – mult register1, register2
  – Multiplies 32-bit values in those registers & puts 64-bit product in special result registers:
    • puts product upper half in hi, lower half in lo
  – hi and lo are 2 registers separate from the 32 general purpose registers
  – Use mfhi register & mflo register to move from hi, lo to another register

• This isn’t just about size but also performance:
  – Multiplication is slow, by separating out when you start from when you need it this can potentially improve performance
Integer Multiplication (3/3)

• Example:
  – in C:   a = b * c;
  – in MIPS:
    • let b be $s2; let c be $s3; and let a be $s0 and $s1 (since it may be up to 64 bits)
      mult $s2,$s3  # b*c
      mfhi $s0     # upper half of product into $s0
      mflo $s1     # lower half of product into $s1

• Note: Often, we only care about the lower half of the product
  – Pseudo-inst. mul expands to mult/mflo
Integer Division (1/2)

• Paper and pencil example (unsigned):

\[
\begin{array}{c|c}
\text{Divisor} & 1000 \\
\hline
\text{Dividend} & 1001010 \\
\hline
\text{Quotient} & 1001 \\
\hline
\text{Remainder} & 10
\end{array}
\]

\[
\begin{array}{c}
\text{Step 1:} 1001010 - 1000 = 1010
\end{array}
\]

\[
\begin{array}{c}
\text{Step 2:} 1010 - 1000 = 10
\end{array}
\]

\[
\begin{array}{c}
\text{Step 3:} 10 - 10 = 0
\end{array}
\]

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

• Syntax of Division (signed):
  – \texttt{div register1, register2}
  – Divides 32-bit register 1 by 32-bit register 2:
  – puts remainder of division in \texttt{hi}, quotient in \texttt{lo}

• Implements C division (/) and modulo (%)

• Example in C: \[ a = c / d; \quad b = c \% d; \]

• in MIPS: \[ a\leftarrow$s0; \quad b\leftarrow$s1; \quad c\leftarrow$s2; \quad d\leftarrow$s3 \]
  \texttt{div $s2,$s3} \quad \# \texttt{lo}=c/d, \texttt{hi}=c\%d
  \texttt{mflo $s0} \quad \# \texttt{get quotient}
  \texttt{mfhi $s1} \quad \# \texttt{get remainder}
Levels of Representation/Interpretation

High Level Language Program (e.g., C)

Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

Compiler

Assembler

Machine Interpretation

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

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

Compiler

Assembly

Machine Interpretation

Hardware Architecture Description

Architecture Implementation

Logic Circuit Description

temp = v[k];

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

v[k+1] = temp;

lw $t0, 0($2)

lw $t1, 4($2)

sw $t1, 0($2)

sw $t0, 4($2)

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
An **Interpreter** is a program that executes other programs. 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?
• MARS—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 (e.g., MARS)
  – Translator reaction: add extra information to help debugging (line numbers, names)
• Interpreter slower (10x?), code smaller (2x?)
• 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
Steps in compiling a C program

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
10. Library: `lib.o`
Compiler

• Input: High-Level Language Code (e.g., \texttt{foo.c})

• Output: Assembly Language Code (e.g. MAL) (e.g., \texttt{foo.s} for MIPS)

• Note: Output \textit{may} contain pseudo-instructions

• \textbf{Pseudo-instructions}: instructions that assembler understands but not in machine

  For example:

  \begin{itemize}
  \item \texttt{move $s1,$s2} $\Rightarrow$ \texttt{add $s1,$s2,$zero}
  \end{itemize}
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

16
Assembler

• Input: Assembly Language Code (e.g. MAL) (e.g., `foo.s` for MIPS)
• Output: Object Code, information tables (TAL) (e.g., `foo.o` for MIPS)

• Reads and Uses Directives
• Replace Pseudo-instructions
• Produce Machine Language
• Creates Object File
Assembler Directives (p. A-51.. A-53)

• 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

  .asciiz 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

#### Pseudo:  
- `subu $sp,$sp,32`
- `sd $a0, 32($sp)`
- `mul $t7,$t6,$t5`
- `addu $t0,$t6,1`
- `ble $t0,100,loop`
- `la $a0, str`

#### Real:  
- `addiu $sp,$sp,-32`
- `sw $a0, 32($sp)`
- `sw $a1, 36($sp)`
- `mult $t6,$t5`
- `mflo $t7`
- `addiu $t0,$t6,1`
- `slti $at,$t0,101`
- `bne $at,$0,loop`
- `lui $at,left(str)`
- `ori $a0,$at,right(str)`
Clicker/Peer Instruction

Which of the following is a correct TAL instruction sequence for `la $v0, FOO`?* (aka store the 32-bit immediate value FOO in $v0)

%hi(label), tells assembler to fill upper 16 bits of label’s addr
%lo(label), tells assembler to fill lower 16 bits of label’s addr

A: `ori $v0, %hi(FOO)`
   `addiu $v0, %lo(FOO)`

B: `ori $v0, %lo(FOO)`
   `lui $v0, %hi(FOO)`

C: `lui $v0, %lo(FOO)`
   `ori $v0, %hi(FOO)`

D: `lui $v0, %hi(FOO)`
   `ori $v0, %lo(FOO)`

E: `la $v0, FOO` is already a TAL instruction

*Assume the address of FOO is 0xABCD0124
Administrivia

• Starting next week, section 124 will be W 3-4 in 310 soda
  – This section should be rather light, so if you want questions answered, this may be a good choice to attend

• Project 2-1 due date will be firmed up in the staff meeting today (probably a Saturday due date)
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:

```
or $v0, $0, $0

L1: slt $t0, $0, $a1
beq $t0, $0, L2
addi $a1, $a1, -1
j 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 require **absolute address**
  – So, forward or not, still can’t generate machine instruction without knowing the position of instructions in memory

• What about references to static data?
  – **la** gets broken up into **lui** and **ori**
  – 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 label jumped to: \texttt{j} or \texttt{jal}
    • internal
    • 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 MS)
  
Where Are We Now?

C program: *foo.c*

Compiler

Assembly program: *foo.s*

Assembler

Object (mach lang module): *foo.o*

Linker

Executable (mach lang pgm): *a.out*

Loader

Memory

lib.o
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.