CS 61C:
Great Ideas in Computer Architecture
Assemblers, Linkers, Compilers

Instructors:
Krste Ananovic, Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa12

...
### Performance Comparison

<table>
<thead>
<tr>
<th>Processor Speed</th>
<th>MacAir 3.1: 1.6 GHz</th>
<th>MacAir 5.1: 2.0 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Name</td>
<td>MacBookAir 3.1</td>
<td>MacBookAir 5.1</td>
</tr>
<tr>
<td>Model Identifier</td>
<td>M11Y2</td>
<td>M12Y3</td>
</tr>
<tr>
<td>CPU Type</td>
<td>Core i5</td>
<td>Core i7</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>1.6 GHz</td>
<td>2.0 GHz</td>
</tr>
<tr>
<td>Total Number of Cores</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Memory</td>
<td>8 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>Bus Speed</td>
<td>800 MHz</td>
<td>800 MHz</td>
</tr>
<tr>
<td>Graphics</td>
<td>Intel Iris 640</td>
<td>Intel Iris 645</td>
</tr>
<tr>
<td>Hardware Version</td>
<td>10.9.1</td>
<td>11.1.2</td>
</tr>
</tbody>
</table>

### Performance Equation

\[ \text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}} \]

- \( \text{Time}_{3.1} = 1/1.6 \text{GHz} = 625 \text{ ns} \)
- \( \text{Time}_{5.1} = 1/2.0 \text{GHz} = 500 \text{ ns} \)
- \( \text{Performance}_{3.1} = \frac{\text{Time}_{3.1}}{625} = 1.25 \)
- \( \text{Performance}_{5.1} = \frac{\text{Time}_{5.1}}{500} = 1.25 \)

### gcc Optimization Experiment

<table>
<thead>
<tr>
<th></th>
<th>BubbleSort.c</th>
<th>Dhrystone.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacAir 3.1</td>
<td>3.2 s</td>
<td>7.4 s</td>
</tr>
<tr>
<td>-O2</td>
<td>1.5 s</td>
<td>2.7 s</td>
</tr>
<tr>
<td>MacAir 5.1</td>
<td>1.9 s (1.7x)</td>
<td>3.0 s (2.4x)</td>
</tr>
<tr>
<td>-O2</td>
<td>0.8 s (1.9x)</td>
<td>0.7 s (3.8x)</td>
</tr>
<tr>
<td></td>
<td>25000000 dhrys/s (3x)</td>
<td></td>
</tr>
</tbody>
</table>

### Agenda

- Review: Performance
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion,...
### Converting C to MIPS Machine code

\[
\begin{align*}
&\text{Add} \quad \text{Address} \\
&816 \\
&812 \\
&808 \\
&804 \\
&800 \\
&\text{Exit:} \\
&\text{Add Loop:}
\end{align*}
\]

#### Format?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>reg</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>35</td>
<td>n.a</td>
</tr>
<tr>
<td>addu</td>
<td>0</td>
<td>$t1$</td>
<td>$s2$, $t0$</td>
<td>43</td>
<td>n.a</td>
<td>n.a</td>
</tr>
<tr>
<td>sw</td>
<td>0</td>
<td>$t0$,1200($t1$)</td>
<td>0</td>
<td>33</td>
<td>n.a</td>
<td>n.a</td>
</tr>
</tbody>
</table>

### Converting to MIPS Machine code

\[
\begin{align*}
&\text{Add} \quad \text{Loop:} \\
&\text{Format?} \\
&800 \\
&804 \\
&808 \\
&812 \\
&816 \\
&820 \\
&\text{Exit:}
\end{align*}
\]

#### Format?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>reg</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>slt</td>
<td>0</td>
<td>$t1$, $s3$, $s2$</td>
<td>0</td>
<td>19</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>addu</td>
<td>0</td>
<td>$t1$, $s3$, $s2$</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>$t0$,0($t1$)</td>
<td>35</td>
<td>9</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>bne</td>
<td>0</td>
<td>$t0$, $s5$, Exit</td>
<td>5</td>
<td>8</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>addu</td>
<td>0</td>
<td>$s3$, $s3$, $s1$</td>
<td>8</td>
<td>19</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>j Loop</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>280</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### 32-bit Constants in MIPS

- Can create a 32-bit constant from two 32-bit MIPS instructions
- Load Upper Immediate (`lui` or “Louie”) puts 16 bits into upper 16 bits of destination register
- MIPS to load 32-bit constant into register $s0$: $0000 0000 0011 1101 0001 0001 0000 0000$ (two)
- lui $s0$, 61 \# 61 = 0000 0000 0011 1101, 2\_two
- ori $s0$, $s0$, 2304 \# 2304 = 0000 1001 0000 0000, 2\_two

### Assembler

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

Many compilers produce object modules directly

Assembly and Pseudo-instructions

- Turning textual MIPS instructions into machine code called **assembly**, program called **assembler**
  - Calculates addresses, maps register names to numbers, produces binary machine language
  - Textual language called **assembly language**
- Can also accept instructions convenient for programmer but not in hardware
  - **Load immediate** \( (li) \) allows 32-bit constants, assembler turns into \( lui + ori \) (if needed)
  - **Load double** \( (ld) \) uses two lw1 instructions to load a pair of 32-bit floating point registers
  - Called **Pseudo-Instructions**

Assembler Directives (P&H B-5 to B-7)

- Give directions to assembler, but do not produce machine instructions
  .text: Subsequent items put in user text segment
  .data: Subsequent items put in user data segment
  .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 \( w_1 \ldots w_n \): Store the \( n \) 32-bit quantities in successive memory words

Assembler Pseudo-instructions

- Most assembler instructions represent machine instructions one-to-one
- Pseudo-instructions: figments of the assembler’s imagination
  - move \( $t0, $t1 \) \( \rightarrow \) \( add \ $t0, \ $zero, \ $t1 \)
  - \( blt \ $t0, \ $t1, L \) \( \rightarrow \) \( slt \ $at, \ $t0, \ $t1 \)
  - \( bne \ $at, \ $zero, L \)
  - \( $at \) (register 1): assembler temporary

More Pseudo-instructions

- Asm. treats convenient variations of machine language instructions as if real instructions
  Pseudo: Real:
  \( addu \ $t0, $t6,1 \) \( \rightarrow \) \( addu \ $t0, $t6,1 \)
  \( subu \ $sp,$sp,32 \) \( \rightarrow \) \( subu \ $sp,$sp,32 \)
  \( sd \ $a0,32($sp) \) \( \rightarrow \) \( sd \ $a0,32($sp) \)
  \( la \ $a0,str \) \( \rightarrow \) \( la \ $a0,str \)
More Pseudoinstructions

- Asm. treats convenient variations of machine language instructions as if real instructions
- **Pseudo:**
  - `addu $t0,$t6,1`
  - `subu $sp,$sp,32`
  - `sd $a0,32($sp)`
  - `la $a0,str`
- **Real:**
  - `addiu $t0,$t6,1`
  - `addiu $sp,$sp,-32`
  - `sw $a0,32($sp)`
  - `lui $at,left(str)`
  - `ori $a0,$at,right(str)`

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
  - Header: described contents of object module
  - Text segment: translated instructions
  - Static data segment: data allocated for the life of the program
  - Relocation info: for contents that depend on absolute location of loaded program
  - Symbol table: global definitions and external refs
  - Debug info: for associating with source code

Agenda

- Review
- Assemblers
- Administrivia
- Linkers
- Compilers vs. Interpreters
- And in Conclusion, …

Administrivia

- Midterm! October 9th, Evening, 8 PM – 10 PM
  - 1 hour exam with 2 hours to complete it
  - Closed notes, book; one 8.5x11” crib sheet, MIPS green card provided
  - Comprehensive from course start
    - Lectures, Labs, Projects
  - There will be a TA-led review session
  - Special accommodations, contact instructors
Agenda

- Review
- Assemblers
- Linkers
- Administrivia
- Compilers vs. Interpreters
- And in conclusion, ...

Separate Compilation and Assembly

- No need to compile all code at once
- How to put pieces together?

Translation and Startup

Many compilers produce object modules directly. The linker searches a collection of object files and program libraries to find nonlocal routines used in a program, combines them into a single executable file, and resolves references between routines in different files.

Linking Object Modules

- Produces an executable image
  1. Merges segments
  2. Resolve labels (determine their addresses)
  3. Patch location-dependent and external refs
- Often slower than compiling—All the machine code files must be read into memory and linked together
Loading a Program

• Load from image file on disk into memory
  1. Read header to determine segment sizes
  2. Create virtual address space (covered later in semester)
  3. Copy text and initialized data into memory
  4. Set up arguments on stack
  5. Initialize registers (including $sp, $fp, $gp)
  6. Jump to startup routine
    • Copies arguments to $a0, ... and calls main
    • When main returns, do "exit" systems call

What’s a Compiler?

• Compiler: a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text.
• Text must comply with the syntax rules of whichever programming language it is written in.
• Compiler’s complexity depends on the syntax of the language and how much abstraction that programming language provides.
  – A C compiler is much simpler than C++ Compiler
• Compiler executes before compiled program runs

What is Typical Benefit of Compiler Optimization?

• gcc compiler options
  -O1: the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time
  -O2: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to -O, this option increases both compilation time and the performance of the generated code
  -O3: Optimize yet more. All -O2 optimizations and also turns on the -finline-functions, ...

Agenda

• Review
• Assemblers
• Administrivia
• Linkers
• Compilers vs. Interpreters
• And in Conclusion, ...

What’s a Compiler?

• Compiler: a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text.
• Text must comply with the syntax rules of whichever programming language it is written in.
• Compiler’s complexity depends on the syntax of the language and how much abstraction that programming language provides.
  – A C compiler is much simpler than C++ Compiler
• Compiler executes before compiled program runs

Compiled Languages: Edit-Compile-Link-Run

- Editor
- Source code
- Compiler
- Object code
- Linker
- Executable program

What is a Compilation Language?

• For now, try a toy program:
  BubbleSort.c

```c
#define ARRAY_SIZE 20000
int main() {
  int array[ARRAY_SIZE], x, y, holder;
  for(x = 0; x < ARRAY_SIZE; x++)
  for(y = 0; y < ARRAY_SIZE-1; y++)
    if(array[y] > array[y+1]) {
      holder = array[y+1];
      array[y+1] = array[y];
      array[y] = holder;
    }
}
```
Unoptimized MIPS Code

-02 optimized MIPS Code

What’s an Interpreter?

- Reads and executes source statements executed one at a time
  - No linking
  - No machine code generation, so more portable
- Starts executing quicker, but runs much more slowly than compiled code
- Performing the actions straight from the text allows better error checking and reporting to be done
- Interpreter stays around during execution
  - Unlike compiler, some work is done after program starts
- Writing an interpreter is much less work than writing a compiler

Compiler vs. Interpreter

- Advantages
  - Faster Execution
  - Single file to execute
  - Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
  - Can find syntax errors before run program
  - Compiler can optimize code

- Disadvantages
  - Harder to debug program
  - Takes longer to change source code, recompile, and relink
  - Slower execution times
  - No optimization
  - Need all of source code available
  - Source code larger than executable for large systems
  - Interpreter must remain installed while the program is interpreted

Interpreted Languages: 

Editor 
Source code 
Interpreter

Editor-Run

Compilation:

- Faster Execution
- Single file to execute
- Compiler can do better diagnosis of syntax and semantic errors, since it has more info than an interpreter (Interpreter only sees one line at a time)
- Can find syntax errors before run program
- Compiler can optimize code

Interpreter:

- Easier to debug program
- Faster development time

Compiler vs. Interpreter
Java’s Hybrid Approach: Compiler + Interpreter

- A Java compiler converts Java source code into instructions for the Java Virtual Machine (JVM)
- These instructions, called bytecodes, are same for any computer / OS
- A CPU-specific Java interpreter interprets bytecodes on a particular computer

Why Bytecodes?

- Platform-independent
- Load from the Internet faster than source code
- Interpreter is faster and smaller than it would be for Java source
- Source code is not revealed to end users
- Interpreter performs additional security checks, screens out malicious code

JVM uses Stack vs. Registers

\[ a = b + c; \]
\[ => \]
\[ iload \ b; \ push \ b \ onto \ Top \ Of \ Stack \ (TOS) \]
\[ iload \ c; \ push \ c \ onto \ Top \ Of \ Stack \ (TOS) \]
\[ iadd ; \ Next \ to \ Top \ Of \ Stack \ (NOS) = \]
\[ ; \ Top \ Of \ Stack \ (TOS) + \ NOS \]
\[ istore \ a; \ store \ TOS \ into \ a \ and \ pop \ stack \]

Java Bytecodes (Stack) vs. MIPS (Reg.)

Starting Java Applications

- Simple portable instruction set for the JVM
- Just In Time (JIT) compiler translates bytecode into machine language just before execution
And, in Conclusion, ...

- Assemblers can enhance machine instruction set to help assembly-language programmer
- Translate from text that easy for programmers to understand into code that machine executes efficiently: Compilers, Assemblers
- Linkers allow separate translation of modules
- Interpreters for debugging, but slow execution
- Hybrid (Java): Compiler + Interpreter to try to get best of both
- Compiler Optimization to relieve programmer