Lecture Overview

• Last week
  – Type and size of operands
    • (common data types, effect of operand size on complexity)
  – Encoding the instruction set
    • (Fixed, variable and hybrid encoding, stored program)

• This week
  – The role of the compiler
  – Effect of ISA on Compiler Complexity
  – Pipeline performance
  – Pipeline hazards
Typical Compilation

Dependencies
- Language dependent; machine independent
- Somewhat language dependent, largely machine independent
- Small language dependencies; machine dependencies slight (e.g., register counts/types)
- Highly machine dependent; language independent

Function
- Transform language to common intermediate form
- For example, procedure inlining and loop transformations
- Including global and local optimizations + register allocation
- Detailed instruction selection and machine-dependent optimizations; may include...
Let's assume that $A$ is an array of word-length integers and that the compiler has associated the variables $g$, $h$ and $i$ with the registers $s1$, $s2$ and $s4$. Let's assume that the starting address, or base address, of the array is in $s3$. The following is a possible compilation of a segment of a C program to MIPS assembly instructions:

$$g = h + A[i];$$

First convert word-index to byte-index:

\[
\begin{align*}
\text{add} & \quad s1, s4, s4 & \# \text{Temp reg } s1 = 2 \times i \\
\text{add} & \quad s1, s1, s1 & \# \text{Temp reg } s1 = 4 \times i
\end{align*}
\]

To get the address of $A[i]$, we need to add $s1$ to the base of $A$ in $s3$:

\[
\begin{align*}
\text{add} & \quad s1, s1, s3 & \# s1 = \text{address of } A[i] \ (4 \times i + s3)
\end{align*}
\]

Now we can use that address to load $A[i]$ into a temporary register:

\[
\begin{align*}
lw & \quad t0, 0(s1) & \# \text{Temporary register } t0 \text{ gets } A[i]
\end{align*}
\]

Finally add $A[i]$ to $h$ and place the sum in $g$:

\[
\begin{align*}
\text{add} & \quad s1, s2, t0 & \# g = h + A[i]
\end{align*}
\]
Assuming the five variables \( f, g, h, i, \) and \( j \) correspond to the five registers \( $s0 \) through \( $s4 \), what is the compiled MIPS code for the following C if statement:

\[
\text{if (} i == j \text{) } f = g + h; \text{ else } f = g - h; 
\]

**MIPS:**

```
bne  $s3, $s4, Else
    # go to Else if \( i \neq j \)
add  $s0, $s1, $s2
    # \( f = g + h \) (skipped if \( i \neq j \))
    j     Exit
    Else: sub $s0, $s1, $s2
        # \( f = g - h \) (skipped if \( i = j \))
    Exit:
```
Compiling a while Loop

Assume that $i$, $j$ and $k$ correspond to $s3$ through $s5$, and the base of the array “save” is in $s6$. what is the compiled MIPS code for the following C segment:

```c
while (save[i] == k)  i = i + j;
```

**MIPS:**

The first step is to load $\text{save}[i]$ into a temporary register

Loop:  

```mips
add $t1, $s3, $s3  # Temp reg $t1 = 2 * i
add $t1, $t1, $t1  # Temp reg $t1 = 4 * i
add $t1, $t1, $s6  # $t1 = address of $\text{save}[i]
lw $t0, 0($t1)  # Temp reg $t0 = $\text{save}[i]
```

The next instruction performs the loop test, exiting if $\text{save}[i] \neq k$

```mips
bne $t0, $s5, Exit  # go to Exit if $\text{save}[i] \neq k$
```

The next instruction add $j$ to $i$:

```mips
add $s3, $s3, $s4  # i = i + j
```

Finally reaching the loop end

```mips
j Loop
```

```
Exit:
```

```
```
# Major Types of Optimization

<table>
<thead>
<tr>
<th>Optimization Name</th>
<th>Explanation</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High –level</strong></td>
<td>At or near source level; machine indep.</td>
<td></td>
</tr>
<tr>
<td>Procedure integration</td>
<td>Replace procedure call by procedure body</td>
<td>N.M</td>
</tr>
<tr>
<td><strong>Local</strong></td>
<td>Within straight line code</td>
<td></td>
</tr>
<tr>
<td>Common sub-expression elimination</td>
<td>Replace two instances of the same computation by single copy</td>
<td>18%</td>
</tr>
<tr>
<td>Constant propagation</td>
<td>Replace all instances of a variable that is assigned a constant with the constant</td>
<td>22%</td>
</tr>
<tr>
<td>Stack height reduction</td>
<td>Rearrange expression tree to minimize resources needed for expression evaluation</td>
<td>N.M</td>
</tr>
<tr>
<td><strong>Global</strong></td>
<td>Across a branch</td>
<td></td>
</tr>
<tr>
<td>Global common sub-expression elimination</td>
<td>Same as local, but this version crosses branches</td>
<td>13%</td>
</tr>
<tr>
<td>Copy propagation</td>
<td>Replace all instances of a variable A that has been assigned X (i.e., A = X) with X</td>
<td>11%</td>
</tr>
<tr>
<td>Code motion</td>
<td>Remove code from a loop that computes same value each iteration of the loop</td>
<td>16%</td>
</tr>
<tr>
<td>Induction variable elimination</td>
<td>Simplify/eliminate array –addressing calculations within loops</td>
<td>2%</td>
</tr>
<tr>
<td><strong>Machine-dependant</strong></td>
<td>Depends on machine knowledge</td>
<td></td>
</tr>
<tr>
<td>Strength reduction</td>
<td>Many examples, such as replace multiply by a constant with adds and shifts</td>
<td>N.M</td>
</tr>
<tr>
<td>Pipeline Scheduling</td>
<td>Reorder instructions to improve pipeline performance</td>
<td>N.M</td>
</tr>
</tbody>
</table>
Measurements taken on MIPS

**Effect of Optimization**

- **Level 0**: non-optimized code
- **Level 1**: local optimization
- **Level 2**: global optimization, s/w pipelining
- **Level 3**: adds procedure integration
Multimedia Instructions

- Small vector processing targeting multimedia
  - Intel’s MMX, PowerPC AltiVec, Sparc VIS, MIPS MDMX
  - $N \frac{1}{N^{th}}$-word vectors packed into one register
  - Same operations performed on all $N$ vectors

- Plus
  - Little additional ALU hardware
  - Utilize under-used hardware resources

- Minus
  - Extra pack & unpack if data isn’t already arranged perfectly
  - Limited vector sizes, difficult to compile for general code

- Result
  - Mostly used in hand-coded libraries

- Compare to general vector processing
  - Hide memory latency in vector access
  - Strided processing, gather/scatter addressing
Effect of Compilers on ISA

• Promote regularity
  – Limit # register formats and variability of operands
  – Orthogonality in operations, registers & addressing

• Provide primitives, not solutions
  – Common features over specific language features
  – Special-purpose instructions often unusable (except through hand-assembly-coded libraries)

• Simplify trade-offs among alternatives
  – Simplify the analysis of special features such as cache and pipeline
  – Allow simultaneous activities to promote optimization

• Favor static binding
Object files for Unix typically contains:

- **Header**: size & position of components
- **Text segment**: machine code
- **Data segment**: static and dynamic variables
- **Relocation info**: identify absolute memory ref.
- **Symbol table**: name & location of labels, procedures and variables
- **Debugging info**: mapping source to object code, break points, etc.

**Linker**

- Place code & data modules symbolically in memory
- Determine the address of data & instruction labels
- Patch both internal & external ref.
## Linking Object Files

### Object file header

<table>
<thead>
<tr>
<th>Name</th>
<th>Procedure A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text size</td>
<td>$100_{\text{hex}}$</td>
</tr>
<tr>
<td>Data size</td>
<td>$20_{\text{hex}}$</td>
</tr>
</tbody>
</table>

### Text segment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lw</td>
<td>$\text{a0, 0}($gp)$</td>
</tr>
<tr>
<td>4</td>
<td>jal</td>
<td>0</td>
</tr>
</tbody>
</table>

### Data segment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(X)</td>
<td></td>
</tr>
</tbody>
</table>

### Relocation Info

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>lw</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>jal</td>
<td>B</td>
</tr>
</tbody>
</table>

### Symbol table

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
</tr>
</tbody>
</table>

### Executable file header

<table>
<thead>
<tr>
<th>Text size</th>
<th>$300_{\text{hex}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size</td>
<td>$50_{\text{hex}}$</td>
</tr>
</tbody>
</table>

### Text segment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040 0000_{\text{hex}}</td>
<td>lw</td>
<td>$\text{a0, 0}($gp)$</td>
</tr>
<tr>
<td>0040 0004_{\text{hex}}</td>
<td>jal</td>
<td>40 0100_{\text{hex}}</td>
</tr>
</tbody>
</table>

### Data segment

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 0000_{\text{hex}}</td>
<td>(X)</td>
<td></td>
</tr>
</tbody>
</table>

### Relocation Info

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction type</th>
<th>Instruction type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0040 0100_{\text{hex}}</td>
<td>lw</td>
<td>$\text{a1, 0}($gp)$</td>
</tr>
<tr>
<td>0040 0104_{\text{hex}}</td>
<td>jal</td>
<td>40 0000_{\text{hex}}</td>
</tr>
</tbody>
</table>

### Symbol table

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
</tr>
</tbody>
</table>

### Assuming the value in $\text{gp}$ is $1000 8000_{\text{hex}}$
To load an executable, the operating system follows these steps:

1. Read the executable file header to determine the size of text and data segments
2. Create an address space large enough for the text and data
3. Copy the instructions and data from the executable file into memory
4. Copy the parameters (if any) to the main program onto the stack
5. Initialize the machine registers and sets the stack pointer to the first free location
6. Jump to a start-up routines that copies the parameters into the argument registers and calls the main routine of the program