COMP 520 - Compilers

Lecture 18 (April 11)

RISC Code Generation

- Lecture covers material outside of the text
  - No additional reading beyond slides
Topics

• Code generation for RISC architectures
  – RISC: Reduced Instruction Set Architecture
  – Sample MIPS code for simple expressions and statements

• Register use in expression evaluation
  – temporary values
  – lifetime of temporaries
  – expression evaluation order and number of temporaries

• Basic optimization steps
  – tuple code
  – common subexpression elimination
  – register allocation
  – spill code
  – linkage optimization
Code generation for RISC instruction sets

• How does a RISC machine compare to a stack-oriented machine
  • all arithmetic and logical functions operate between registers
    – no explicit stack
  • minimal support for procedure/function call
    – no frame maintenance
  • registers are a compiler-managed cache

• Compiler challenges related to a RISC instruction set
  – large but finite set of registers (32 – 128)
    • register allocation – a problem for the compiler
      – which values should be placed in registers?
      – what should we do when we run out of registers?
  – procedure and function calling conventions
    • no direct support for execution stack
      – frame pointer, stack pointer, and other linkage conventions must be defined
    • flexible strategy for register use
      – needed to limit register save/restore work in call/return
      – parameters and results passed in registers instead of on the stack
MIPS memory organization

- **Key areas**
  - **Reserved**
    - for use by operating system
  - **Text segment**
    - compiler-generated program is loaded here
  - **Stack segment**
    - procedure invocation stack
      - expands downwards
      - holds some local variables, some linkage information
  - **Data segment**
    - static constants and variables are placed at the bottom
      - their locations are known by the compiler
    - dynamically allocated values are placed above the static data
      - this area is the heap
      - heap allocation addresses cannot be predicted by the compiler
      - heap expands upwards
      - memory for deleted values can be reused/compacted
## MIPS register conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Symbolic name</th>
<th>Value</th>
<th>Set by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$zero</td>
<td>Constant zero</td>
<td>Not set</td>
</tr>
<tr>
<td>1</td>
<td>$at</td>
<td>ASM use only</td>
<td>ASM</td>
</tr>
<tr>
<td>2-3</td>
<td>$v0, $v1</td>
<td>Function results</td>
<td>Callee</td>
</tr>
<tr>
<td>4-7</td>
<td>$a0, $a3</td>
<td>Functions args 1-4</td>
<td>Caller</td>
</tr>
<tr>
<td>8-15</td>
<td>$t0, $t7</td>
<td>Temp – not preserved</td>
<td></td>
</tr>
<tr>
<td>16-23</td>
<td>$s0, $s7</td>
<td>Temp – preserved</td>
<td></td>
</tr>
<tr>
<td>24-25</td>
<td>$t8, $t9</td>
<td>Temp – not preserved</td>
<td></td>
</tr>
<tr>
<td>26-27</td>
<td>$k0, $k1</td>
<td>OS use only</td>
<td>OS</td>
</tr>
<tr>
<td>28</td>
<td>$gp</td>
<td>Global pointer</td>
<td>OS</td>
</tr>
<tr>
<td>29</td>
<td>$sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>30</td>
<td>$fp</td>
<td>Frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>31</td>
<td>$ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
</tbody>
</table>

- $s0 - $s7 are expected to be preserved across a function call
- $t0 - $t9 are not expected to be preserved across a function call
Example Code 1 – simple statement

miniJava

```java
x = y + z;
```

MIPS assembler

```mips
lw $t0, @y
lw $t1, @z
add $t2, $t0, $t1
sw $t2, @x
```

• Notes
  – x, y, z are local int vars
    • represented as 32-bit binary values
    • word-aligned
    • @y stands for address of y in memory
      – typically some offset added to $fp (local vars) or $gp (global vars)
  – all operations take place between registers
    • values of y, z are fetched from frame
    • value of result is stored into x in frame
Example Code 2 – multiple statements

\[
\begin{align*}
  x &= y + z; \\
  w &= y - z;
\end{align*}
\]

\[
\begin{align*}
  lw &\quad t0, \ y \\
  lw &\quad t1, \ z \\
  add &\quad t2, \ t0, \ t1 \\
  sw &\quad t2, \ x \\
  lw &\quad t0, \ y \\
  lw &\quad t1, \ z \\
  sub &\quad t2, \ t0, \ t1 \\
  sw &\quad t2, \ w
\end{align*}
\]

register allocation within a statement

register allocation across two statements
Code Example 3 – multiple execution paths

```c
if (x < y)
    x = y + 1;
```

```assembly
lw $t0, @x
lw $t1, @y
bge $t0, $t1, L10
addi $t0, $t1, 1
sw $t1, @x
L10:
```

```c
while (x != y) {
    if (x > y)
        x = x - y;
    else
        y = y - x;
}
```

```assembly
lw $t0, @x
lw $t1, @y
Ltop:
    beq $t0, $t1, Lend
    ble $t0, $t1, Lelse
Lthen:
    sub $t0, $t0, $t1
    b Ltop
Lelse:
    sub $t1, $t1, $t0
    b Ltop
Lend:
    sw $t0, @x
    sw $t1, @y
```

register allocation across execution paths

register allocation across loops – large payoff
Example Code 4 – function call

```plaintext
x = x + f(x)+ x;

lw   $s0, @x  
mv   $a0, $s0  
jal  @f  
aw   $t0, $s0, $v0  
aw   $t0, $t0, $s0  
sw   $t0, @x

int f(x){ return 3 + x;}

f:  
    subiu $sp, $sp, 8  
    sw   $ra, 8($sp)  
    sw   $fp, 4($sp)  
    addiu $fp, $sp, 8  
    addiu $v0, $a0, 3  
    lw   $ra, 8($sp)  
    lw   $fp, 4($sp)  
    addiu $sp, $sp, 8  
    jr   $ra

Function invocation in an expression

Function body

explicit frame/stack management can be omitted in this case!
Expression evaluation

• How many registers are needed to evaluate an expression?
  – using our stack-based expression evaluation strategy we could use a new register for each temporary value

\[(A - B) + ((C + D) + (E * F))\]

We may easily run out of registers using this strategy!
Expression evaluation

- Minimizing the number of registers needed
  - Sethi-Ullman labeling
    - each subexpression is labeled with the minimum number of registers needed to evaluate it
    - defined inductively by the rules below
  - rearrange expression evaluation order to minimize register use
    - evaluate the subexpression requiring more registers first

A simple variable

\[ e \]

A unary expr

\[ op_k e \]

A binary expr

\[ op_k e1 \]
\[ op_k e2 \]

A binary expr

\[ op_k e1 \]
\[ op_k e2 \]

where

- \( h < k \)
Sethi-Ullman labeling and expression reordering

- Questions
  - Is evaluation order rearrangement always possible?
  - Is minimal register usage always best?

\[(A - B) + ((C + D) + (E * F))\]

\[
\begin{align*}
    t1 & := A \\
    t2 & := D \\
    t1 & := t1 + t2 \\
    t2 & := B \\
    t3 & := F \\
    t2 & := t2 * t3 \\
    t1 & := t1 + t2 \\
    t2 & := A \\
    t3 & := B \\
    t2 & := t2 - t3 \\
    t1 & := t2 + t1
\end{align*}
\]
Expression evaluation

- Reusing registers
  - Consider tuple code - how long is each temporary value needed?
    - Lifetime starts at definition, ends at last use
  - values with disjoint lifetimes may use the same register

\[(A - B) + ((A + D) + (B * F))\]

<table>
<thead>
<tr>
<th></th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
<th>t8</th>
<th>t9</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 :=</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2 :=</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3 :=</td>
<td>t1 - t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4 :=</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5 :=</td>
<td>t1 + t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6 :=</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7 :=</td>
<td>t2 * t6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t8 :=</td>
<td>t5 + t7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t9 :=</td>
<td>t3 + t8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Register allocation for expressions

- AST → tuple code generation
- tuple code → common subexpression elimination
- tuple code → lifetime analysis
- AST → interference graph
- interference graph → interference analysis
- interference graph → register allocation
- tuple code with machine registers → object code generation
- object code generation → object code
- lifetime analysis → object code
Tuple code generation

- Tuple code
  - list of tuples
  - each tuple represents a machine instruction
    - (destination, operation, operands)
  - use temporaries rather than registers
  - operations at level of target instruction set

- Tuple code generation
  - postorder traversal of AST

Example: \((A - B) + ((A + D) + (A - B))\)
Common subexpression elimination

• Replace redundant loads and computations

Example: \((A - B) + ((A + D) + (A - B))\)

<table>
<thead>
<tr>
<th>Tuple code</th>
<th>repl tuples by copy</th>
<th>subst, del copy</th>
<th>renumb temps</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 := A</td>
<td>t1 := A</td>
<td>t1 := A</td>
<td>t1 := A</td>
</tr>
<tr>
<td>t2 := B</td>
<td>t2 := B</td>
<td>t2 := B</td>
<td>t2 := B</td>
</tr>
<tr>
<td>t3 := t1 - t2</td>
<td>t3 := t1 - t2</td>
<td>t3 := t1 - t2</td>
<td>t3 := t1 - t2</td>
</tr>
<tr>
<td>t4 := A</td>
<td>t4 := t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5 := D</td>
<td>t5 := D</td>
<td>t5 := D</td>
<td>t4 := D</td>
</tr>
<tr>
<td>t6 := t4 + t5</td>
<td>t6 := t4 + t5</td>
<td>t6 := t1 + t5</td>
<td>t5 := t1 + t4</td>
</tr>
<tr>
<td>t7 := A</td>
<td>t7 := t1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t8 := B</td>
<td>t8 := t2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>t9 := t7 - t8</td>
<td>t9 := t3</td>
<td>t9 := t3</td>
<td></td>
</tr>
<tr>
<td>t10 := t6 + t9</td>
<td>t10 := t6 + t9</td>
<td>t10 := t6 + t3</td>
<td>t6 := t5 + t3</td>
</tr>
<tr>
<td>t11 := t3 + t10</td>
<td>t11 := t3 + t10</td>
<td>t11 := t3 + t10</td>
<td>t7 := t3 + t6</td>
</tr>
</tbody>
</table>
Lifetime analysis

• For each temporary
  – construct interval from first definition to last use

```
<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1 := A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t2 := B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t3 := t1 - t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t4 := D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5 := t1 + t4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6 := t5 + t3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7 := t3 + t6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Allocating temporaries to registers

• Create interference graph
  – node for each temporary
  – edge (u,v) if live range of u overlaps live range of v

• Allocate tuple code temporaries to registers
  – temporaries connected by an edge in the interference graph must be in distinct registers
    • because the lifetimes of the two temporaries at the ends of the edge overlap
  – can all temporaries be allocated to k registers?
    • possible iff interference graph can be k-colored
Interference graph

t1 := A

2 := B

t3 := t1 - t2

t4 := D

t5 := t1 + t4

t6 := t5 + t3

t7 := t3 + t6

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
</table>

Diagram:

- t1
- t2
- t3
- t4
- t5
- t6
- t7

Connections:
- t1 to t7
- t2 to t7
- t3 to t5
- t4 to t5
- t4 to t6
- t6 to t7
Register Allocation

- **k-coloring interference graph**
  - k is number of available registers
  - k-coloring is an NP-complete problem
  - linear time greedy heuristic algorithm (may fail)
    - repeatedly remove a node and all associated edges from a graph and add to a list
      - choose a node with indegree < k whenever possible
    - when graph is empty, color nodes in reverse of order removed
  - if heuristic algorithm fails
    - pick a temporary t with a long lifetime and indegree ≥ k
    - Generate spill code
      - insert tuple operation to store t when generated and fetch t where needed
      - repeat lifetime analysis, interference analysis

- **Additional considerations**
  - constrain temporaries for function arguments to appropriate machine registers
  - constrain temporaries live across a call to callee-preserved registers (i.e. s0-s7)
**Greedy k-coloring algorithm [Kempe 1879]**

- Given undirected graph $G = (V,E)$, try to color the nodes in $V$ using $k$ colors so that for any $(u,v) \in E$, $\text{color}(u) \neq \text{color}(v)$

Stack of nodes $S$ initially empty
Undirected graph $G = (V,E)$

```plaintext
while $G \neq \{\}$ do
    choose some minimum degree node $t$ in $V$
    $S$.push($t$)
    $G$.remove($t$)  // remove $t$ and edges incident on $t$ from $G$
end
```

```plaintext
while $S \neq \{\}$ do
    $t = S$.pop()
    $G$.insert($t$)  // add $t$ and edges in $G$ incident on $t$
    color $t$, if possible
end
```
Kempe algorithm – simplify step

Stack of nodes $S$ initially empty

Undirected graph $G = (V, E)$

while $G \neq \emptyset$ do
    choose some minimum degree node $t$ in $V$
    $S$.push($t$)
    $G$.remove($t$)

$S = (t_1, t_3, t_4, t_5, t_6, t_2, t_7)$

$G =$

Diagram showing the steps of the Kempe algorithm:
- Initially, $G$ contains all the nodes.
- After each iteration, a node is removed from $G$ and added to $S$.
- Nodes are connected to form triangles and a final single node.

Diagrams illustrate the progression of the algorithm with removals and additions.
Kempe algorithm – color step

Stack of nodes $S$ to color in order
Undirected graph $G = (V,E)$ initially empty

while $S \neq {}$ do
  $t = s$.pop()
  $G$.insert($t$)  // add $t$ and edges in $G$ incident on $t$
  color $t$, if possible
end

$k = 3$ colors: ▢ □ ▪
Register Allocation

- rewrite tuple code with register allocation

\[
\begin{align*}
t1 & := A \\
t2 & := B \\
t3 & := t1 - t2 \\
t4 & := D \\
t5 & := t1 + t4 \\
t6 & := t5 + t3 \\
t7 & := t3 + t6
\end{align*}
\]

\[
\begin{align*}
r1 & := A \\
r2 & := B \\
r2 & := r1 - r2 \\
r3 & := D \\
r1 & := r1 + r3 \\
r3 & := r1 + r2 \\
r1 & := r2 + r3
\end{align*}
\]
Putting it together: RISC code generation

- AST → tuple code generation → tuple code
- tuple code → common subexpression elimination → tuple code
- tuple code → interference analysis → interference graph
- interference graph → register allocation → tuple code with machine registers
- tuple code with machine registers → object code generation → object code
- object code → lifetime analysis → live ranges
- live ranges → object code generation

RISC Code Generation