Register Allocation
Compilers

Source code

```
int main() {
    int x = 4 + 5;
    return x;
}
```

Tokens

Lexing (P2)

Syntax tree

Parsing (P3)

Analysis (P4)

Checked AST + Symtables

Optimized Linear IR

Optimization Passes

Machine Code Gen

Current focus
Machine Code Gen (Ch. 11-13)

- Translate from (usually linear) IR to machine code
  - Often, compilers will just emit assembly
  - Use built-in system assembler and linker to create final executable

- Issues:
  - Translation from IR instructions to machine code instructions
    - Instruction selection (Ch. 11) - example in y86.c provided w/ P5
  - Arrangement of machine code instructions for optimal pipelining
    - Instruction scheduling (Ch. 12) - algorithm next week; no implementation
  - Assignment of registers to minimize memory accesses
    - Register allocation (Ch. 13) - primary focus of P5
Instruction Selection

- Choose machine code instructions to replace IR
  - Complexity is highly dependent on target architecture
  - CISC provides more options than RISC (e.g., x86 vs. ARM)
    - Tradeoff: *(possible)* performance improvement vs. compiler complexity

- Algorithms:
  - Treewalk routine (similar to P4)
  - Tree-pattern matching / tiling (variant implemented in y86.c in P5)

\[
d = a + c\times4 + 8
\]

Source Language  | Intermediate Representation (IR)  | Target Language
--- | --- | ---
leaq 0x8(%rax,%rcx,4), %rdx | mov %rcx, %rdi | d
mulI c, 4 => t1 | imul $4, %rdi | add a, t1 => t2
addI t2, 8 => d | add %rax, %rdi | add $8, %rdi
Instruction Scheduling

• Modern CPUs expose many opportunities for optimization
  – Some instructions require fewer cycles
  – Instruction pipelining
  – Branch prediction and speculative execution
  – Multicore shared-memory processors

• **Scheduling**: re-order instructions to improve speed
  – Must not modify program semantics
  – Maximize utilization of CPU and memory resources
  – Main algorithm: **list scheduling** (next week!)
Register Allocation

- Maximizing register use is very important
  - Registers are the lowest-latency memory locations
  - Issue: limited number of registers
    - Everything not in registers must be stored in cache or main memory
    - Need to reduce the # of registers used to match the target system
    - Program using $n$ registers $\Rightarrow$ Program using $m$ registers ($n \gg m$)

- Allocation vs. assignment
  - **Allocation**: map a virtual register space to a physical register space
    - This is hard (NP-complete for any realistic situation)
  - **Assignment**: map a valid allocation to actual register names
    - This is easy (linear or polynomial)
Which virtual registers should be allocated to “real” physical registers and which must be allocated elsewhere?
Local Allocation

- **Top-down local register allocation**
  - Compute a priority for each virtual register
    - Frequency of access to that register
  - Sort by priority, highest to lowest
  - Assign registers in order, highest priority first
  - Rewrite the code

- **General idea: prioritize most-often-accessed virtual registers**
  - Allocate to physical registers in priority order
  - Very simple to implement
  - Static per-block allocations are not always optimal
    - Access patterns may change throughout block
    - Especially in SSA form where registers aren’t often re-used
Local Allocation

- **Bottom-up local register allocation**
  - Scan each block instruction-by-instruction
    - Essentially, simulate running the program
  - Maintain physical-to-virtual register mapping ("Name")
    - Initialize registers to empty ("INVALID") at beginning of block
  - For each instruction:
    - Assign virtual registers to physical registers
    - Ensure operands are in physical registers (load them if not)
    - Greedy algorithm: choose best allocation available at each instruction
      - Track next reference and free physical registers as soon as possible
Example

Suppose we have three physical registers:

\[
\begin{align*}
\text{Name}[r0] &= \text{INVALID} & r1 &= \text{INVALID} & r3 &= \text{INVALID} \\
\text{Name}[r1] &= \text{INVALID} & r2 &= \text{INVALID} \\
\text{Name}[r2] &= \text{INVALID} \\
\end{align*}
\]

add:
- loadAI [bp+16] => r1
- loadAI [bp+24] => r2
- add r1, r2 => r3
- i2i r3 => ret
- return

add:
- loadAI [bp+16] => r0
- loadAI [bp+24] => r1
- add r0, r1 => r0
- i2i r0 => ret
- return
Example

add:
  loadAI [bp+16] => r1
  loadAI [bp+24] => r2
  add r1, r2 => r3
  i2i r3 => ret
  return

main:
  loadI 3 => r4
  storeAI r4 => [bp-8]
  loadAI [bp-8] => r5
  loadI 2 => r6
  param r6
  param r5
  call add
  i2i ret => r7
  i2i r7 => ret
  return

add:
  loadAI [bp+16] => r0
  loadAI [bp+24] => r1
  add r0, r1 => r0
  i2i r0 => ret
  return

main:
  loadI 3 => r0
  storeAI r0 => [bp-8]
  loadAI [bp-8] => r0
  loadI 2 => r1
  param r1
  param r0
  call add
  i2i ret => r0
  i2i r0 => ret
  return

Only needed two physical registers for this example!
Example

```plaintext
gcd:
l1:
  loadAI [bp+24] => r1
  loadI 1 => r2
  cmp_GE r1, r2 => r3
  cbr r3 => l2, l3
l2:
  loadAI [bp+24] => r4
  loadI 0 => r5
  store r4 => [r5]
  loadAI [bp+16] => r6
  loadAI [bp+24] => r7
  div r6, r7 => r8
  mult r7, r8 => r9
  sub r6, r9 => r10
  storeAI r10 => [bp+24]
  loadI 0 => r11
  load [r11] => r12
  storeAI r12 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r13
  i2i r13 => ret
  return
```

```plaintext
gcd:
l1:
  loadAI [bp+24] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r0
  cbr r0 => l2, l3
l2:
  loadAI [bp+24] => r0
  loadI 0 => r1
  store r0 => [r1]
  loadAI [bp+16] => r0
  loadAI [bp+24] => r1
  div r0, r1 => r2
  mult r1, r2 => r1
  sub r0, r1 => r0
  storeAI r0 => [bp+24]
  loadI 0 => r0
  load [r0] => r0
  storeAI r0 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r0
  i2i r0 => ret
  return
```

What if we only had two physical registers?
• If no physical registers are free, **spill** one!
  – Store its value to memory and re-load it later
  – For optimal results, spill register that will be accessed the furthest in the future
    • Store Next[pr] for this purpose or just re-calculate when needed
• This is the hardest part of P5 (leave it for last!)
  – Allocate slot in stack frame for each spilled register
    • It’s essentially a new local variable
  – Track the offset for each virtual register
  – Emit load/store instructions as needed
    • **Significant helper code is provided!**
Bottom-up local register allocation

gcd:
l1:
  loadAI [bp+24] => r1
  loadI 1 => r2
  cmp_GE r1, r2 => r3
  cbr r3 => l2, l3
l2:
  loadAI [bp+24] => r4
  loadI 0 => r5
  store r4 => [r5]
  loadAI [bp+16] => r6
  loadAI [bp+24] => r7
  div r6, r7 => r8
  mult r7, r8 => r9
  sub r6, r9 => r10
  storeAI r10 => [bp+24]
  loadI 0 => r11
  load [r11] => r12
  storeAI r12 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r13
  i2i r13 => ret
  return

gcd:
l1:
  loadAI [bp+24] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r0
  cbr r0 => l2, l3
l2:
  loadAI [bp+24] => r0
  loadI 0 => r1
  store r0 => [r1]
  loadAI [bp+16] => r0
  loadAI [bp+24] => r1
  div r0, r1 => ???
  mult r1, ??? => r1
  sub r0, r1 => r0
  storeAI r0 => [bp+24]
  loadI 0 => r0
  load [r0] => r0
  storeAI r0 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r0
  i2i r0 => ret
  return
Bottom-up local register allocation

gcd:
l1:
  loadAI [bp+24] => r1
  loadI 1 => r2
  cmp_GE r1, r2 => r3
  cbr r3 => l2, l3
l2:
  loadAI [bp+24] => r4
  loadI 0 => r5
  store r4 => [r5]
  loadAI [bp+16] => r6
  loadAI [bp+24] => r7
  div r6, r7 => r8
  mult r7, r8 => r9
  sub r6, r9 => r10
  storeAI r10 => [bp+24]
  loadI 0 => r11
  load [r11] => r12
  storeAI r12 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r13
  i2i r13 => ret
  return

gcd:
l1:
  loadAI [bp+24] => r0
  loadI 1 => r1
  cmp_GE r0, r1 => r0
  cbr r0 => l2, l3
l2:
  loadAI [bp+24] => r0
  loadI 0 => r1
  store r0 => [r1]
  loadAI [bp+16] => r0
  loadAI [bp+24] => r1
  storeAI r0 => [bp-8] // store r6
  div r0, r1 => r0
  mult r1, r0 => r1
  loadAI [bp-8] => r0 // load r6
  sub r0, r1 => r0
  storeAI r0 => [bp+24]
  loadI 0 => r0
  load [r0] => r0
  storeAI r0 => [bp+16]
  jump l1
l3:
  loadAI [bp+16] => r0
  i2i r0 => ret
  return
Full algorithm

for each instruction $i$ in program:

if $i$ is a leader, reset data structures

for each $vr$ read in $i$:

$pr = \text{Ensure}(vr)$
replace $vr$ with $pr$ in $i$
if $vr$ is not needed after $i$ then free $pr$

for each $vr$ written in $i$:

$pr = \text{Allocate}(vr)$
replace $vr$ with $pr$ in $i$

Ensure($vr$):
if $vr$ is in $pr$:
    return $pr$
else:
    $pr = \text{Allocate}(vr)$
    emit load from $vr$
    return $pr$

Allocate($vr$):
if $pr$ is available:
    return $pr$
else:
    find furthest-used $pr$ to spill
    emit spill for $pr$
    return $pr$
Textbook vs. reference compiler

- Textbook algorithm uses a stack to store free registers
  - Must remember to add registers to stack when freeing them
  - $O(1)$ access to a free register if one is available
- Reference compiler scans physical registers for first free one
  - $O(k)$ where $k$ is number of physical registers, which is essentially a small constant
  - Only need Name[$pr$]
    - $pr$ is free if Name[$pr$] == INVALID

**TEXTBOOK:**

- loadI 1 => r0
- loadI 2 => r1
- add r0, r1 => r1

**REFERENCE:**

- loadI 1 => r0
- loadI 2 => r1
- add r0, r1 => r0
How many registers does it take to evaluate an arbitrary expression without any spilling?

- Is there an easy way to determine this?

For example:

\[
\text{a + b + c + d}
\]

vs.

\[
(a+b) + (c+d)
\]
How many registers does it take to evaluate an arbitrary expression without any spilling?

- Examine the expression tree (e.g., parse tree)
- Calculate the Strahler number:
  - If the node is a leaf (has no children), its Strahler number is one.
  - If the node has one child with Strahler number i, and all other children have Strahler numbers less than i, then the Strahler number of the node is i.
  - If the node has two or more children with Strahler number i, and no children with greater number, then the Strahler number of the node is i + 1.

\[
\begin{align*}
\text{Strahler number of node } a &= 1 \\
\text{Strahler number of node } b &= 1 \\
\text{Strahler number of node } c &= 1 \\
\text{Strahler number of node } d &= 1 \\
\text{Strahler number of node } e &= 1
\end{align*}
\]
Systems design tradeoff

• Parallelism vs. register pressure
  – Balanced trees provide more parallelism and (as we’ll see next week) better pipelining
    • However, more spills => worse performance
  – Unbalanced trees require fewer registers
    • Fewer spills => better performance
    • However, fewer opportunities for parallelism and pipelining
  – Usually the parallelism is worth the increased register pressure
    • Especially in the presence of forwarding and robust caches
Local vs. global allocation

• Local allocation handles each basic block separately
  – Will miss inter-block dependencies

• Global allocation handles all basic blocks in a procedure
  – Does NOT consider inter-procedural dependencies
  – This is why calling conventions are important
    • I.e., caller-save vs. callee-save and return value

• Decaf project
  – Because we used SSA in P4 and always load/store to memory, no virtual registers will be live at the entrance or exit of any block (so no inter-block dependencies)
  – Thus, we can use local register allocation in P5
Global Allocation

• Discover global live ranges of related uses and definitions
  – For each use, any reaching definitions must be in the same range
  – For each definition, any reachable uses must be in the same range
  – Simple disjoint-set union-find algorithm over SSA form

• Build interference graph
  – Node for each live range and edges between interfering live ranges

• Attempt to compute graph $k$-coloring
  – $k$ is the number of physical registers
  – Greedy algorithm: order the colors (registers)
    • For each vertex, choose smallest color not shared by neighbors
  – If successful, done!
  – If not successful, spill some values and try again
    • Need a robust way to pick which values to spill
    • Alternatively, split live ranges at carefully-chosen points
Global Allocation

\[
x = 4 \\
y = 10
\]

\[
x = 6 \\
y = 5
\]

\[
y = 7
\]

\[
z = x + y
\]

\[
w = z + 1
\]
LR₁: x={4, 6}

\[ x = 4 \]
\[ y = 10 \]

\[ x = 6 \]
\[ y = 5 \]

\[ y = 7 \]

\[ z = x + y \]

\[ w = z + 1 \]
Global Allocation

LR₁: \( x = \{4, 6\} \)

\[
\begin{align*}
x &= 4 \\
y &= 10
\end{align*}
\]

LR₂: \( y = 10 \)

\[
\begin{align*}
x &= 6 \\
y &= 5
\end{align*}
\]

\[
\begin{align*}
y &= 7
\end{align*}
\]

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

\[
\begin{align*}
w &= z + 1
\end{align*}
\]
Global Allocation

\[ x = 4 \]
\[ y = 10 \]

\[ x = 6 \]
\[ y = 5 \]

\[ y = 7 \]

\[ z = x + y \]

\[ w = z + 1 \]

\[ LR_1: \ x = \{4, 6\} \]

\[ LR_2: \ y = 10 \]

\[ LR_3: \ y = \{5, 7\} \]
Global Allocation

\[ x = 4 \]
\[ y = 10 \]

\[ x = 6 \]
\[ y = 5 \]

\[ y = 7 \]

\[ z = x + y \]

\[ w = z + 1 \]

LR\(_1\): \( x = \{4, 6\} \)
LR\(_2\): \( y = 10 \)
LR\(_3\): \( y = \{5, 7\} \)
LR\(_4\): \( z = x + y \)
Global Allocation

\[ x = 4 \]
\[ y = 10 \]
\[ x = 6 \]
\[ y = 5 \]
\[ y = 7 \]
\[ z = x + y \]
\[ w = z + 1 \]
Global Allocation

\[ x = 4 \]
\[ y = 10 \]
\[ z = x + y \]
\[ w = z + 1 \]