Liveness analysis

Simone Campanoni
simonec@eecs.northwestern.edu
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
A register allocator structure

```
Register allocator

Map heuristic

f without varX

Current f, varX

Spill

f without variables
```
A graph-coloring register allocator structure

f

Register allocator

Code analysis

Graph coloring

f with var spilled

spill(f, var, prefix)

Spill

f without variables and with registers
Task: From Variables to Registers

(:MyVeryImportantFunction 0
  %MyVar1 <- 2
  %MyVar2 <- 40
  %MyVar3 <- %MyVar1
  %MyVar3 += %MyVar2
  print %MyVar3
)

- Why can we map MyVar1 and MyVar3 to r8?
- Why can’t we map MyVar1 and MyVar2 to r8?

We built the interference graph.
To compute it automatically, we need the liveness analysis.
Liveness analysis

**Goal:**
Identify the variables whose values might be used in the future just before and just after a given instruction $i$, for every $i$ in a function $f$:

```plaintext
(myF 0
  (%myVar1 <- 2)
  (%myVar2 <- 40)
  (%myVar3 <- %myVar1)
  (%myVar3 += %myVar2)
  rax <- %myVar3
  return)
```

Live ranges

**IN (just before) and OUT (just after) sets**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{myVar1}</td>
<td>{myVar1, myVar2}</td>
<td>{myVar3, myVar2}</td>
<td>{myVar3}</td>
</tr>
</tbody>
</table>

**Interference graph**

Graph coloring

:myF w/o variables

myVar1 — myVar2 — myVar3
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
Variables in the liveness analysis

• GP 64-bit registers are seen as variables for the liveness analysis
  • `rsp` is not included

• Every time we say “variable” in the context of liveness analysis, we mean either L2 variables or GP 64-bit registers

• IN and OUT sets of the liveness analysis includes variables
  • Hence, they include L2 variables or GP 64-bit registers
  • `IN[i] = {r10}`
Execution path

Let $i$ be an instruction, we need to identify the set of variables with values that will be used just before and just after $i$. 

It is a successor of. 

It is a predecessor of.
Successors of an instruction

\[ i ::= \begin{align*}
& w \leftarrow s \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \mid w \leftarrow \text{stack-arg} \ M \\
& \quad w \ \text{aop} \ t \mid w \ \text{sop} \ sx \mid w \ \text{sop} \ N \mid \text{mem} \times M \ + t \mid \text{mem} \times M \ - t \mid w \ + = \ \text{mem} \times M \mid w \ - = \ \text{mem} \times M \\
& \quad w \leftarrow t \ \text{cmp} \ t \mid \text{cjum}p \ t \ \text{cmp} \ t \ \text{label} \mid \ \text{label} \mid \text{goto} \ \text{label} \\
& \quad \text{return} \mid \text{call} \ u \ N \mid \text{call} \ \text{print} \ 1 \mid \text{call} \ \text{input} \ 0 \mid \text{call} \ \text{allocate} \ 2 \mid \text{call} \ \text{tensor-error} \ F \\
& \quad w++ \mid w-- \mid w @ w \ w \ E
\end{align*} \]

An instruction \( i \) that has only one successor \( s \) and \( s \) is the instruction stored just after \( i \)

\[ \begin{align*}
& \text{rdi} \leftarrow 5 \\
& \text{call} \ \text{print} \ 1 \\
& \text{r10} \leftarrow \text{rax} < 5
\end{align*} \]
Successors of an instruction (2)

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem x M} \mid \text{mem x M} \leftarrow s \mid w \leftarrow \text{stack-arg M} \mid w \text{ aop t} \mid w \text{ sop sx} \mid w \text{ sop N} \mid \text{mem x M }+= t \mid \text{mem x M }-= t \mid w += \text{mem x M} \mid w -= \text{mem x M} \mid w \leftarrow t \text{ cmp t} \mid \text{cjump t cmp t label} \mid \text{label} \mid \text{goto label} \mid \text{return} \mid \text{call u N} \mid \text{call print 1} \mid \text{call input 0} \mid \text{call allocate 2} \mid \text{call tensor-error F} \mid w++ \mid w-- \mid w @ w w E \]

An instruction \( i \) that has only one successor \( s \) but \( s \) is not necessarily the instruction stored just after \( i \)

\[ \text{goto :MY_LABEL_0} \]
\[ \text{got o :MY_LABEL_0} \]
\[ \text{rdi }\leftarrow 5 \]
\[ \text{call print 1} \]
\[ \text{goto :MY_LABEL_0} \]
\[ :\text{MY_LABEL_0} \]
\[ :\text{MY_LABEL_0} \]
Successors of an instruction (3)

An instruction $i$ that has no successor
Successors of an instruction (4)

\[ i ::= w \leftarrow s \mid w \leftarrow \text{mem} \times M \mid \text{mem} \times M \leftarrow s \mid w \leftarrow \text{stack-arg} \times M \mid w \leftarrow \text{aop} \times t \mid w \leftarrow \text{sop} \times \text{sx} \mid w \leftarrow \text{sop} \times N \mid \text{mem} \times M \leftarrow t \mid \text{mem} \times M \leftarrow -t \mid w \leftarrow \text{mem} \times M \mid w \leftarrow \text{mem} \times M \mid \text{w} \leftarrow \text{cmp} \times t \mid \text{cmp} \times t \mid \text{cmp} \times \text{label} \mid \text{label} \mid \text{goto} \times \text{label} \mid \text{return} \mid \text{call} \times u \times N \mid \text{call} \times \text{print} \times 1 \mid \text{call} \times \text{input} \times 0 \mid \text{call} \times \text{allocate} \times 2 \mid \text{call} \times \text{tensor-error} \times F \mid \text{w} \leftarrow + \mid \text{w} \leftarrow - \mid \text{w} \leftarrow \text{w} \times \text{w} \times E \]

An instruction \( i \) that has two successors

\[ \text{cjump} \times \text{rax} \times < 5 : L1 \]
\[ \text{rdi} \leftarrow 1 \]
\[ \text{rsi} \leftarrow 3 \]
\[ : L1 \]
Now with knowledge about paths and successors we can compute IN and OUT sets of each instruction of a function automatically.
Liveness analysis

A variable is **alive** at a particular point in the program if its value at that point will be used in a path that starts from there (the future).

A variable is **dead** if it is not alive.

- To compute liveness at a given point, we need to look into the future
- A variable $v$ is alive at a given point of a program $p$ if
  - Exist a directed path from $p$ to a use of $v$ and
  - that path does not contain any definition of $v$

```plaintext
%v1 <- 5
%v3 <- 3
%v2 <- %v1
return
```

```plaintext
%v1 <- 5
%v1 <- 3
%v2 <- %v1
return
```
Liveness analysis algorithm

1. Identify which variables are define and which ones are read (used) by an instruction
   • GEN and KILL sets (local information)

2. Specify how instructions transmit live values around the program
   • How to compute IN and OUT sets from GEN and KILL sets (global information)

3. Iterate (2) until nothing (i.e., IN and OUT set) changes
   • Notice that (1) is performed only once!
   • GEN and KILL sets are constants and, therefore, path independent!
GEN and KILL sets

• GEN\[i\] = \{all variables read (used) by instruction i\}

  %myVar3 <- %myVar1     \// GEN[i] = \{%myVar1\}

• KILL\[i\] = \{all variables defined by instruction i\}

  %myVar3 <- %myVar1     \// KILL[i] = \{%myVar3\}

%myVar3 += %myVar1

KILL\[i\] = \{%myVar3\}    GEN\[i\] = \{%myVar1, %myVar3\}
GEN and KILL sets: more examples

• GEN[i] = \{all variables read (used) by instruction i\}
• KILL[i] = \{all variables defined by instruction i\}

rdi++
KILL[i] = \{rdi\}
GEN[i] = \{rdi\}
GEN and KILL sets: more examples

• \( \text{GEN}[i] = \{\text{all variables read (used) by instruction } i\} \)
• \( \text{KILL}[i] = \{\text{all variables defined by instruction } i\} \)

\[
\text{cjump rdi} \leq \%v2 : \text{true}
\]
\( \text{KILL}[i] = \{ \} \)
\( \text{GEN}[i] = \{\text{rdi, } \%v2\} \)
Liveness analysis algorithm

1. Define which variables are defined and which ones are read (used) for each instruction
   • GEN and KILL sets

2. Specify how instructions transmit live values around the program
   • How to compute IN and OUT sets from GEN and KILL sets

3. Iterate (2) until nothing changes
IN and OUT sets

• \( \text{IN}[i] = \{ \text{all variables live right before instruction } i \} \)
  \[
  \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
  \]

• \( \text{OUT}[i] = \{ \text{all variables live right after instruction } i \} \)
  \[
  \text{OUT}[i] = \bigcup_{\text{s a successor of } i} \text{IN}[s]
  \]

- If \( \text{OUT}[i] = {} \) then \( \text{IN}[i] = \{ %v1 \} \)
- If \( \text{OUT}[i] = \{ %v2 \} \) then \( \text{IN}[i] = \{ %v1 \} \)
- If \( \text{IN}[i+1] = \{ %v1 \} \), then \( \text{OUT}[i] = \{ %v1 \} \)
- If \( \text{IN}[i+1] = \{ %v1 \} \) and \( \text{IN}[i+j] = \{ %v2 \} \), then \( \text{OUT}[i] = \{ %v1, %v2 \} \)
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}
for (each instruction $i$) IN[$i$] = OUT[$i$] = { }

do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] ∪ (OUT[$i$] − KILL[$i$])
        OUT[$i$] = ∪$_s$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
Outline

• Introduction to register allocation

• Liveness analysis

• Calling convention
Code example

(:myF
  0
  %a <- 2   // 1   { }   {%a}
  rax <- %a  // 2   { %a }   { rax }
  return  // 3   { rax }   { } )

• Are GEN and KILL sets correct?
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...  
    KILL[$i$] = ...
}

for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };

    do{

        for (each instruction $i$){
            IN[$i$] = GEN[$i$] U (OUT[$i$] – KILL[$i$])
            OUT[$i$] = $U_s$ a successor of $i$ IN[$s$]
        }
    }

} while (changes to any IN or OUT occur);
Code example

(defun myF
  (0
   (%a <- 2)  ; 1
   (rax <- %a)  ; 2
   (return)  ; 3
  )
)

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

• Are GEN and KILL sets correct?
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}

for (each instruction $i$)  IN[$i$] = OUT[$i$] = { };

do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] ∪ (OUT[$i$] – KILL[$i$])
        OUT[$i$] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
    }
} while (changes to any IN or OUT occur);
Code example

```plaintext
(:myF
  0
  %a <- 2    // 1
  rax <- %a  // 2
  return     // 3
)
```

<table>
<thead>
<tr>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>{%a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{%a}</td>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

• Are GEN and KILL sets correct?

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_s \text{a successor of } i \text{ IN}[s]
\]
Code example

( :myF
   0
   %a <- 2    // 1
   rax <- %a  // 2
   return    // 3
)

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN</td>
<td>{ }</td>
<td>{%a}</td>
<td>{ }</td>
<td>{}</td>
</tr>
<tr>
<td>KILL</td>
<td>{rax}</td>
<td>{ }</td>
<td>{rax}</td>
<td>{ }</td>
</tr>
<tr>
<td>IN</td>
<td>{ }</td>
<td>{ }</td>
<td>{rax}</td>
<td>{ }</td>
</tr>
<tr>
<td>OUT</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]
\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Code example

`:myF
0
%a <- 2  // 1
rax <- %a  // 2
return  // 3
)`

• Are GEN and KILL sets correct?

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEN</td>
<td>{}</td>
<td>{%a}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>KILL</td>
<td>{%a}</td>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>IN</td>
<td>{}</td>
<td>{rax}</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>OUT</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Code example

```c
(:myF
  0
  %a <- 2  // 1
  rax <- %a  // 2
  return  // 3
)
```

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>rax</td>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a}</td>
<td>{rax}</td>
</tr>
</tbody>
</table>

• Are GEN and KILL sets correct?

\[
\begin{align*}
\text{IN}[i] &= \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \\
\text{OUT}[i] &= \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\end{align*}
\]
### Code example

```plaintext
(:myF
  0
  %a <- 2    // 1
  rax <- %a  // 2
  return     // 3
)
```

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>{}</td>
<td>{%a}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>rax</td>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a}</td>
<td>{rax}</td>
</tr>
<tr>
<td>rax</td>
<td>{}</td>
<td>{rax}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Are GEN and KILL sets correct?

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] \setminus \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Code example

```plaintext
(:myF
0
%a <- 2    // 1
rax <- %a   // 2
return      // 3
)
```

- Are GEN and KILL sets correct?

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
<th>IN</th>
<th>OUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%a</td>
<td></td>
<td>{%a}</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td>rax</td>
<td>{%a}</td>
<td>{rax}</td>
<td>{%a}</td>
<td>{rax}</td>
</tr>
<tr>
<td>return</td>
<td>{rax}</td>
<td>{ }</td>
<td>{rax}</td>
<td>{ }</td>
</tr>
</tbody>
</table>

\[\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i])\]
\[\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]\]
Algorithm

for (each instruction $i$) {
    GEN[$i$] = ...
    KILL[$i$] = ...
}
for (each instruction $i$) IN[$i$] = OUT[$i$] = { };
do{
    for (each instruction $i$){
        IN[$i$] = GEN[$i$] U (OUT[$i$] − KILL[$i$])
        OUT[$i$] = $U_s$ a successor of $i$ IN[$s$]
    }
} while (changes to any IN or OUT occur);
Code example

( :myF
  0
  %a <- 2  // 1
  rax <- %a // 2
  return    // 3
)

• Are GEN and KILL sets correct?

\[
\begin{array}{|c|c|c|c|}
\hline
\text{GEN} & \text{KILL} & \text{IN} & \text{OUT} \\
\hline
\{\} & \{\%a\} & \{\} & \{\%a\} \\
\{\%a\} & \{\text{rax}\} & \{\%a\} & \{\text{rax}\} \\
\{\text{rax}\} & \{\} & \{\text{rax}\} & \{\} \\
\hline
\end{array}
\]

\[
\text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] \setminus \text{KILL}[i])
\]

\[
\text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s]
\]
Algorithm

for (each instruction \( i \)) {
    GEN\([i]\) = ...  
    KILL\([i]\) = ...
}

for (each instruction \( i \))  \( \text{IN}[i] = \text{OUT}[i] = \{ \} \);
do{
    for (each instruction \( i \)){
        \( \text{IN}[i] = \text{GEN}[i] \cup (\text{OUT}[i] - \text{KILL}[i]) \)
        \( \text{OUT}[i] = \bigcup_{s \text{ a successor of } i} \text{IN}[s] \)
    }
} while (changes to any IN or OUT occur);
### Code example

```plaintext
(:myF 0
     %a <- 2 // 1 { } {%a} { } {%a}
     rax <- %a  // 2 {%a} {rax} {%a} {rax}
     return  // 3 {rax} { } {rax} { }
)
```
Steps

1. Compute IN and OUT sets

2. Compute interference graph from IN and OUT sets
Generating the interference graph

• 1 node per variable or register

• Connect each pair of variables that belong to the same IN or OUT set
• Connect a register to all other registers
• ... (see Interference graph slides for other steps)

• Meaning of an edge: 2 connected nodes must use different registers
Code example

(:myF
  0
  %a <- 2  // 1
  rax <- %a  // 2
  return  // 3
)

• Graph coloring can assign r12 to %a
Code example

(:myF
  0
  r12 <- 2 // 1
  rax <- r12 // 2
  return // 3
)

• Are GEN and KILL sets correct?
• Graph coloring can assign r12 to %a
• Is there any problem?
Code example

(:myF
  0
  r12 <- 2  // 1
  rax < r12  // 2
  return    // 3
)

• The calling convention counts as definitions and uses
• When adding them as such, we automatically enforce the calling convention

• Are GEN and KILL sets correct?
• Graph coloring can assign r12 to %a
• Is there any problem?
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Return instruction in a 2 registers CPU

\[(\text{myF } 0)\]
\[\%	ext{a} \leftarrow 2\]
\[\text{return}\]

\(<\text{w/o calling convention}\>
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{w/o calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{w/ calling convention}\]
\%	ext{a} \quad \text{r12}
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
\]
\[\text{Graph coloring}\]
\(%\text{a} \quad \text{r12}\)
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{ }</td>
</tr>
</tbody>
</table>
Call instructions

• Which register should we use for %a?
  r10

• Is it correct? (r10 is a caller save register)

(:myF 0
 %a <- 2
 call :f 0
 %a *= %a
 rax <- %a
 return)

(:myF 0
 r10 <- 2
 call :f 0
 r10 *= r10
 rax <- r10
 return)
# Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u, args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return</td>
<td>{ rax, callee save registers}</td>
<td>{}</td>
</tr>
</tbody>
</table>
## Calling convention in GEN/KILL

<table>
<thead>
<tr>
<th></th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>call u N</td>
<td>{ u,</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td></td>
<td>args used}</td>
<td></td>
</tr>
<tr>
<td>call RUNTIME N</td>
<td>{ args used}</td>
<td>{ caller save registers}</td>
</tr>
<tr>
<td>return</td>
<td>{ rax,</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>callee save registers}</td>
<td></td>
</tr>
</tbody>
</table>
Homework #1

• Compute the IN and OUT sets of all instructions of an L2 function given as input

```plaintext
(:myF
  0
  %myVar1 <- 5
  %myVar2 <- 0
  %myVar2 += %myVar1
  return
)
```

Your work

```
   ( (in
      IN[0] → (r13 r15 rax rbp r12 rbx )
      IN[1] → (r13 r15 rax rbp r12 %myVar1 rbx )
      IN[2] → (r13 r15 rax rbp %myVar2 r12 %myVar1 rbx )
      IN[3] → (r13 r15 rax r14 rbp r12 rbx )
    ) (out
      OUT[0] → (r13 r15 rax r14 rbp r12 %myVar1 rbx )
      OUT[1] → (r13 r15 rax r14 rbp %myVar2 r12 %myVar1 rbx )
      OUT[2] → (r13 r15 rax r14 rbp r12 rbx )
      OUT[3] → ( )
    )
)```

Testing your homework #1

• Under L2/tests/liveness there are the tests you have to pass

• A new compiler argument: -l
  • Check L2compiler.cpp on Canvas

• To test:
  • To check all tests: make test_liveness
  • To check one test: ./liveness test/liveness/test1.L2f

• Check out each input/output for each test if you have doubts
  • For example, the correct output for the test
    test/liveness/test1.L2f
    is
    test/liveness/test1.L2f.out
Debugging suggestion

• Don’t forget you have our L2 compiler binary

• So, to help you debug your work:
  • you can write your own test (a new MyTest.L2f)
  • Generate the output of our L2 compiler by invoking
    ./liveness MyTest.L2f > MyTest.L2f.out
  • Compare our output with the output generated by your L2 compiler