Example of generated MIPS code (using virtual registers):

```
.data
x: .space 4
y: .space 4

.text
la v0, x
lw v1, (v0)
add v2, v0, v1
la v3, y
lw v4, (v3)
sub v5, v4, v2
add v6, v2, v4
sw v5, (v0)
sw v6, (v3)
```

After "proper" register allocation (possible output):

```
.data
x: .space 4
y: .space 4

.text
la $t0, x
lw $t1, ($t0)
add $t2, $t0, $t1
la $t3, y
lw $t4, ($t3)
sub $t5, $t4, $t2
add $t6, $t2, $t4
sw $t5, ($t0)
sw $t6, ($t3)
```

What if less than 7 architectural registers available for allocation?

- Need to know which values is going to be used in the future.
Liveness

**Definition**
A variable (virtual register) is *live* at some point in the program if it has previously been *defined* by an instruction and will be *used* by an instruction in the future. It is *dead* otherwise.

💡 Two variables can use the same architectural register if they are never used at the same time, *i.e.* never simultaneously live.

⇒ Register allocation use liveness information.
Example:

```
.data
x: .space 4
y: .space 4
.text
la    v0, x
lw    v1, (v0)
add   v2, v1, v1
la    v3, y
lw    v4, (v3)
sub   v5, v4, v2
add   v6, v2, v4
sw    v5, (v0)
sw    v6, (v3)
```

Live after instruction:

```
v0
v0 v1
v0  v2
v0  v2 v3
v0  v2 v3 v4
v0  v2 v3 v4 v5
v0  v3  v5  v6
v3  v6
```

Question: how many architectural registers are needed?
Computing liveness is more complicated in the presence of control flow (e.g. loops, if-then-else).

**Assembly pseudo-code:**

```
a = 0
L1: b = a + 1
c = c + b
a = b*2
if (a<9) goto L1
return c
```

Question: what is the live range of b?

To answer this question we need to understand the **dynamic flow** of the program execution.

---

1We illustrate concepts at a slightly higher level than assembly from this point on.
Control-Flow Graph (CFG)

Concept invented in 1970 by:

Frances Allen (1932–2020), IBM, (1st woman to receive Turing Award in 2006!)

a = 0
L1: b = a + 1
   c = c + b
   a = b * 2
if (a < 9) goto L1
return c

Directed graph:

entry
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
5: a < 9
True
6: return c
False
What is the live range of b?

- b is used in statement 4, so b is live on the 3 → 4 edge
- since statement 3 does not define b, b is also live on the 2 → 3 edge
- statement 2 defines b, so any value of b on the 1 → 2 and 5 → 2 edges are not needed, so b is dead along these edges

b live range is 2 → 3 → 4
Live range of $a$:
- $1 \rightarrow 2$ and $4 \rightarrow 5 \rightarrow 2$

Live range of $b$:
- $2 \rightarrow 3 \rightarrow 4$

Live range of $c$:
- $entry \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2$ and $5 \rightarrow 6$

⚠️ Since $a$ and $b$ never simultaneously live, can share a register.
Flow Graph

- A Control Flow Graph (CFG) has out-edges that lead to successor nodes and in-edges that come from predecessor nodes
- $\text{pred}(n) =$ set of all predecessors of node $n$
- $\text{succ}(n) =$ set of all successors of node $n$

Examples

- Out-edges of node 5: $5 \rightarrow 6$ and $5 \rightarrow 2$
- $\text{succ}(5) =$ \{2,6\}
- $\text{pred}(5) =$ \{4\}
- $\text{pred}(2) =$ \{1,5\}
Uses and Defs

**Def (definition)**
- A **write** of a value to a variable
- \( \text{def}(v) = \text{set of CFG nodes that define variable } v \)
- \( \text{def}(n) = \text{set of variables defined at node } n \)

**Use**
- A **read** of a variable’s value
- \( \text{use}(v) = \text{set of CFG nodes that use variable } v \)
- \( \text{use}(n) = \text{set of variables used at node } n \)
A variable $v$ is live on a CFG edge if

- $\exists$ a directed path from that edge to a use of $v$ (node $\in$ use(b)) and
- that path does not go through any def of $v$ (nodes $\notin$ def(v)).
Computing Liveness
Flow of Liveness

Data-flow

- Liveness of variables is a property that flows through the edges of the CFG

Direction of flow

- Liveness flows \textit{backward} in the CFG: behaviour of future nodes determines liveness at a given node
Example: flow of liveness for a

1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
5: a < 9
6: return c

Example: flow of liveness for b

1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
5: a < 9
6: return c
We have liveness on edges

- before and after each node

Two more definitions:

- A variable is **live-out** at a node if it is live on any of that node’s out-edges
- A variable is **live-in** at a node if it is live on any of that node’s in-edges
Computing Liveness

Rules for computing liveness

1. Generate liveness:
   \( \nu \in \text{use}(n) \Rightarrow \nu \in \text{LIVE}_{in}(n) \)

2. Push liveness across edges:
   \( \nu \in \text{LIVE}_{in}(n) \Rightarrow \forall p \in \text{pred}(n) \ \nu \in \text{LIVE}_{out}(p) \)

3. Push liveness across nodes:
   \( \nu \in \text{LIVE}_{out}(n) \land \nu \notin \text{def}(n) \Rightarrow \nu \in \text{LIVE}_{in}(n) \)

Data-flow equations

\[
\begin{align*}
\text{LIVE}_{in}(n) &= \text{use}(n) \cup (\text{LIVE}_{out}(n) - \text{def}(n)) \\
\text{LIVE}_{out}(n) &= \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{in}(s)
\end{align*}
\]
Solving the Data-flow equations

1: for all node $n \in CFG$ do
2: \[ \text{LIVE}_{in}(n) = \emptyset \]
3: \[ \text{LIVE}_{out}(n) = \emptyset \]
4: end for
5: repeat
6: for all node $n \in CFG$ do
7: \[ \text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n) \]
8: \[ \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n) \]
9: \[ \text{LIVE}_{in}(n) = \text{use}(n) \cup (\text{LIVE}_{out}(n) - \text{def}(n)) \]
10: \[ \text{LIVE}_{out}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{in}(s) \]
11: end for
12: until \[ \text{LIVE}'_{in}(n) = \text{LIVE}_{in}(n) \land \text{LIVE}'_{out}(n) = \text{LIVE}_{out}(n) \forall n \]

This is a fix-point algorithm for iterative liveness analysis.
Example

```
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
5: a < 9
6: return c
```

Data-flow equations

\[
\text{LIVE}_{\text{in}}(n) = \text{use}(n) \cup (\text{LIVE}_{\text{out}}(n) - \text{def}(n))
\]

\[
\text{LIVE}_{\text{out}}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{\text{in}}(s)
\]
There is something inefficient about this process.

For instance, consider the 3 → 4 edge in the graph:

- $\text{LIVE}_{\text{out}}(4)$ is used to compute $\text{LIVE}_{\text{in}}(4)$
- $\text{LIVE}_{\text{in}}(4)$ is used to compute $\text{LIVE}_{\text{out}}(3)$

💡 The algorithm would converge faster if we process the nodes backwards.
Backward Liveness Analysis

1: \textbf{for all} node $n \in \text{CFG} \textbf{do}
2: \quad \text{LIVE}_{in}(n) = \emptyset
3: \quad \text{LIVE}_{out}(n) = \emptyset
4: \textbf{end for}
5: \textbf{repeat}
6: \quad \textbf{for all} node $n \in \text{CFG}$ \textbf{in reverse topological order do}
7: \quad \text{LIVE}^{'\prime}_{in}(n) = \text{LIVE}_{in}(n)
8: \quad \text{LIVE}^{'\prime}_{out}(n) = \text{LIVE}_{out}(n)
9: \quad \text{LIVE}_{out}(n) = \bigcup_{\forall s \in \text{succ}(n)} \text{LIVE}_{in}(s)
10: \quad \text{LIVE}_{in}(n) = \text{use}(n) \cup (\text{LIVE}_{out}(n) - \text{def}(n))
11: \textbf{end for}
12: \textbf{until} \text{LIVE}^{'\prime}_{in}(n) = \text{LIVE}_{in}(n) \land \text{LIVE}^{'\prime}_{out}(n) = \text{LIVE}_{out}(n) \forall n
Example with Backward Liveness Analysis

Converges in only 3 iterations!

Data-flow equations

\[
\text{LIVE}_{\text{out}}(n) = \bigcup_{s \in \text{succ}(n)} \text{LIVE}_{\text{in}}(s)
\]

\[
\text{LIVE}_{\text{in}}(n) = \text{use}(n) \cup (\text{LIVE}_{\text{out}}(n) - \text{def}(n))
\]
More performance considerations

**Basic Block**

A straight sequence of assembly instruction which (usually) finishes with a branch/jump instruction.

Key property: Either *all* the instructions in the sequence execute or none execute.

Can significantly decrease the size that a CFG occupies in memory by grouping nodes that have a single predecessor and a single successor into basic blocks.

The instructions in a basic block can be simply represented as a list (rather than a graph).
Example

No basic blocks:

```
entry
1: a = 0
2: b = a + 1
3: c = c + b
4: a = b * 2
5: a < 9
True
6: return c
False
```

With basic blocks:

```
entry
1: a = 0
2: b = a + 1
   c = c + b
   a = b * 2
   a < 9
True
3: return c
False
```

22
Next lecture

- Proper register allocation