Finding Dead Code
how do we find dead code?

- Easy answer: if the variable being written by an instruction is not live, the code is dead
- Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

```
1:   A = B + C
2:   C = A + B
3:   T1 = A + B
4:   D = T1 + C
5:   T2 = D + T1
6:   D = A + B
7:   WRITE(D)
```
how do we find dead code?

• Easy answer: if the variable being written by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

```
1:   A = B + C    {A, B}
2:   C = A + B    {A, B}
3:   T1 = A + B   {A, B, C}
4:   D = T1 + C   {A, B, C, T1}
5:   T2 = D + T1  {A, B, D, T1}
6:   D = A + B    {A, B}
7:   WRITE(D)     {D}
```
how do we find dead code?

• Easy answer: if the variable being written by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

```plaintext
1:   A = B + C   {A, B}
2:   C = A + B   {A, B}
3:   T1 = A + B  {A, B, C}
4:   D = T1 + C  {A, B, C, T1}
5:   T2 = D + T1  {A, B, D, T1}
6:   D = A + B   {A, B}
7:   WRITE(D)    {D}
```
how do we find dead code?

• Easy answer: if the variable *being written* by an instruction is not live, the code is dead

• Intuition: the value you are generating is not being used anywhere else, so generating this value is pointless

1: \( A = B + C \) \( \{A, B\} \)
2: \( C = A + B \) \( \{A, B\} \)
3: \( T1 = A + B \) \( \{A, B, C\} \)
4: \( D = T1 + C \) \( \{A, B, C, T1\} \)
5: \( T2 = D + T1 \) \( \{A, B, D, T1\} \)
6: \( D = A + B \) \( \{A, B\} \)
7: WRITE(D) \( \{D\} \)
how do we find dead code?

- After you remove dead code, it changes liveness information
- Recompute and iterate

1: \[ A = B + C \] \{A, B\}
2: \[ C = A + B \] \{A, B, C\}
3: \[ T1 = A + B \] \{A, B, C, T1\}
4: \[ D = T1 + C \] \{A, B\}

6: \[ D = A + B \] \{D\}
7: \[ \text{WRITE}(D) \] \{\}
how do we find dead code?

• After you remove dead code, it changes liveness information
• Recompute and iterate

1: \( A = B + C \) \( \{A, B\} \)
2: \( C = A + B \) \( \{A, B, C\} \)
3: \( T1 = A + B \) \( \{A, B, C, T1\} \)
4: \( D = T1 + C \) \( \{A, B\} \)
6: \( D = A + B \) \( \{D\} \)
7: WRITE(D) \( \{\} \)
how do we find dead code?

- After you remove dead code, it changes liveness information
- Recompute and iterate

```plaintext
1:  A = B + C    {A, B}
2:  C = A + B    {A, B}
3:  T1 = A + B    {A, B}
6:  D = A + B    {D}
7:  WRITE(D)    {}
```
how do we find dead code?

- After you remove dead code, it changes liveness information
- Recompute and iterate

1: \( A = B + C \) \{A, B\}
2: \( C = A + B \) \{A, B\}
3: \( T1 = A + B \) \{A, B\}

6: \( D = A + B \) \{D\}
7: WRITE(D) \{\}
how do we find dead code?

• After you remove dead code, it changes liveness information
• Recompute and iterate

```
1:   A = B + C       {A, B}

6:   D = A + B       {D}
7:   WRITE(D)        {D}
```
can we do this faster?

• Recomputing and iterating is slow!

• We can speed this up by computing **use-def** chains
  
  • Track how uses of variables are connected to definitions of those variables

• Can trace backwards from live code along use-def chains
  
  • Instruction is “backwards reachable” from live code → instruction is live

  • Instruction is *not* backwards reachable → no definition from this instruction eventually propagates to live code, instruction is dead

• This generalizes to a program analysis technique called **program slicing**
next: register allocation