CS406: Compilers

Spring 2022

Week 14:

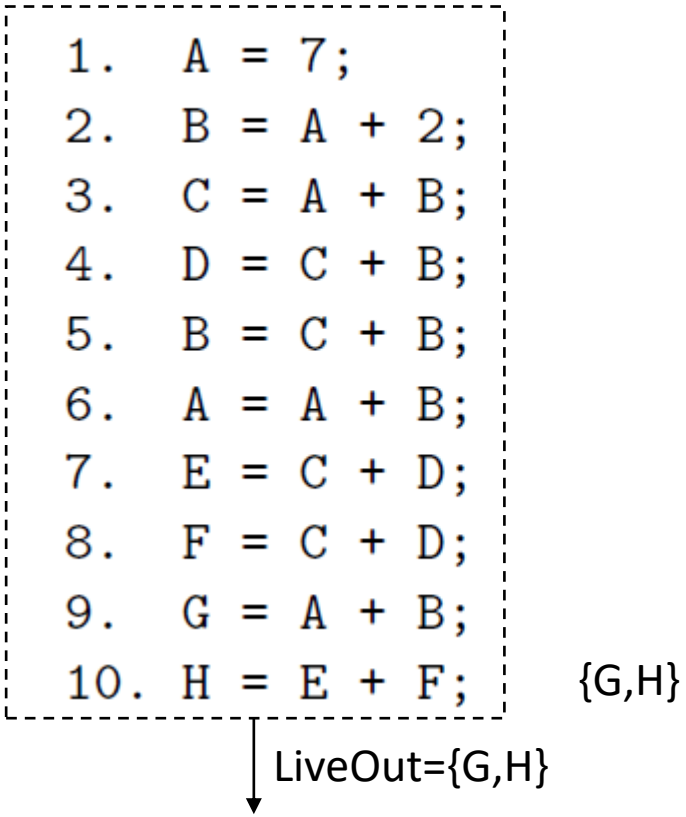
Register allocation via Graph coloring (example), Loop Dependence Analysis

Register Allocation via Graph Coloring - Example

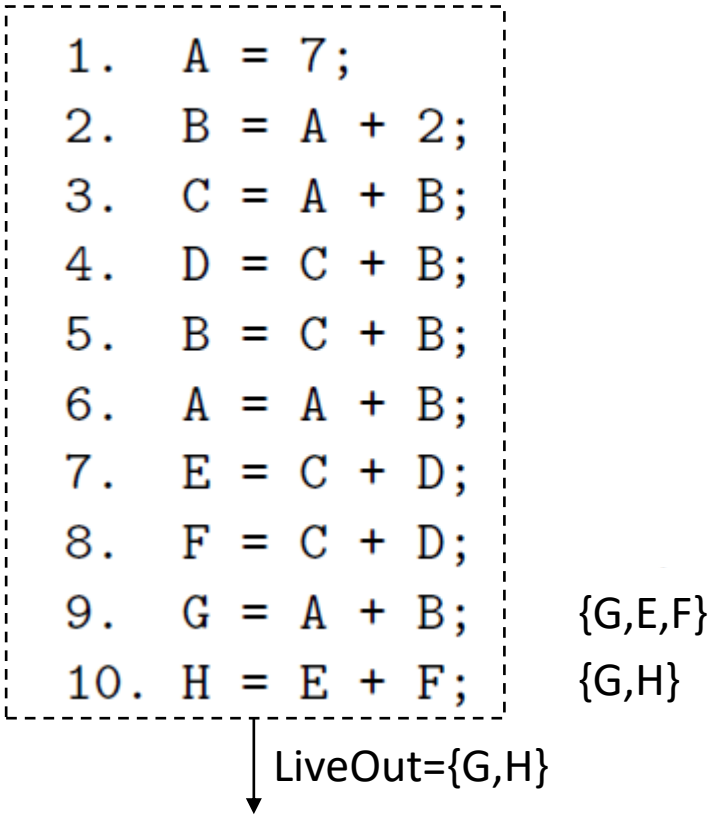
```
1.  A = 7;
2.  B = A + 2;
3.  C = A + B;
4.  D = C + B;
5.  B = C + B;
6.  A = A + B;
7.  E = C + D;
8.  F = C + D;
9.  G = A + B;
10. H = E + F;
```

LiveOut={G,H}

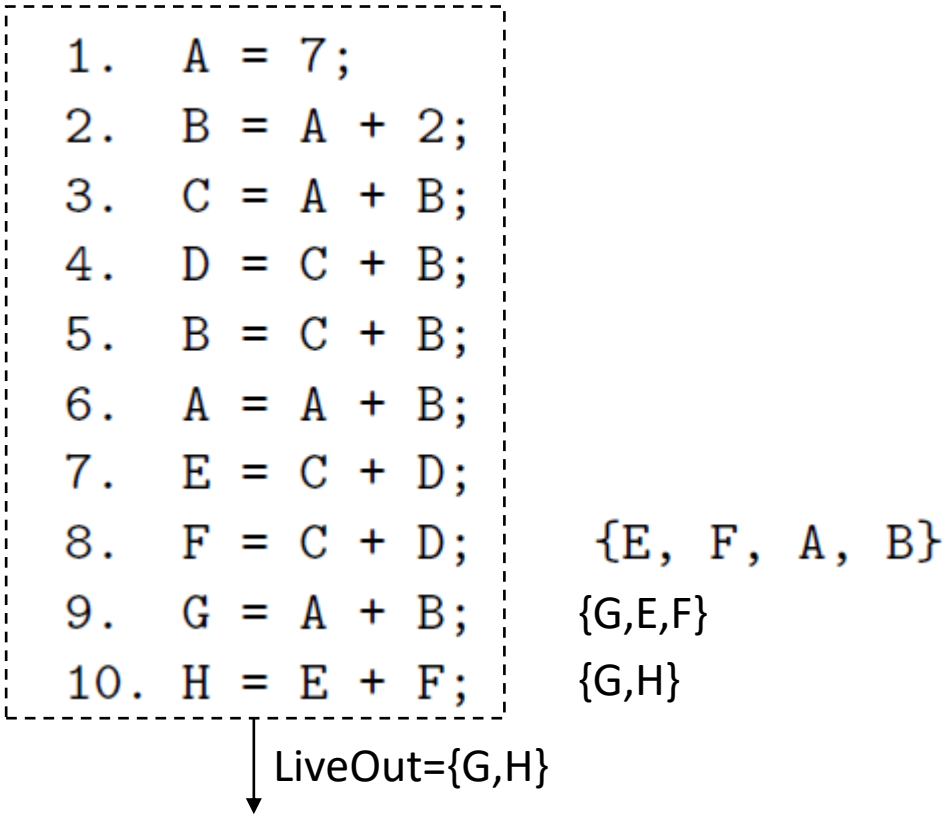
Register Allocation via Graph Coloring - Example



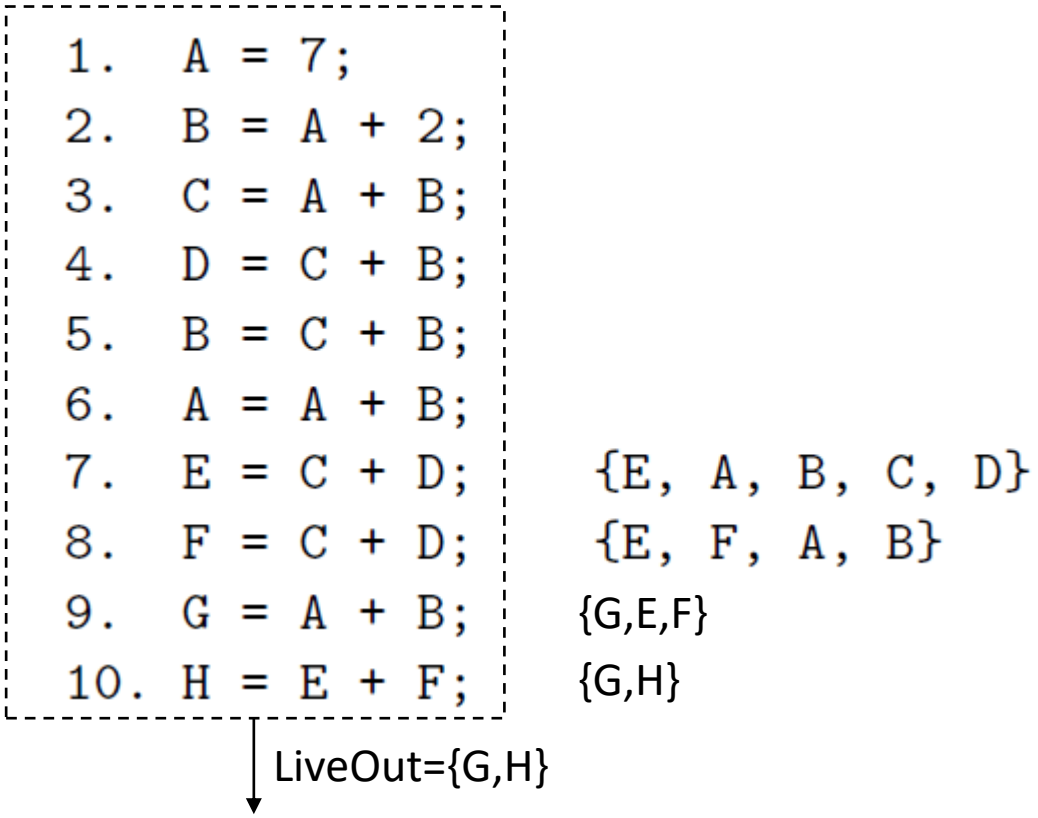
Register Allocation via Graph Coloring - Example



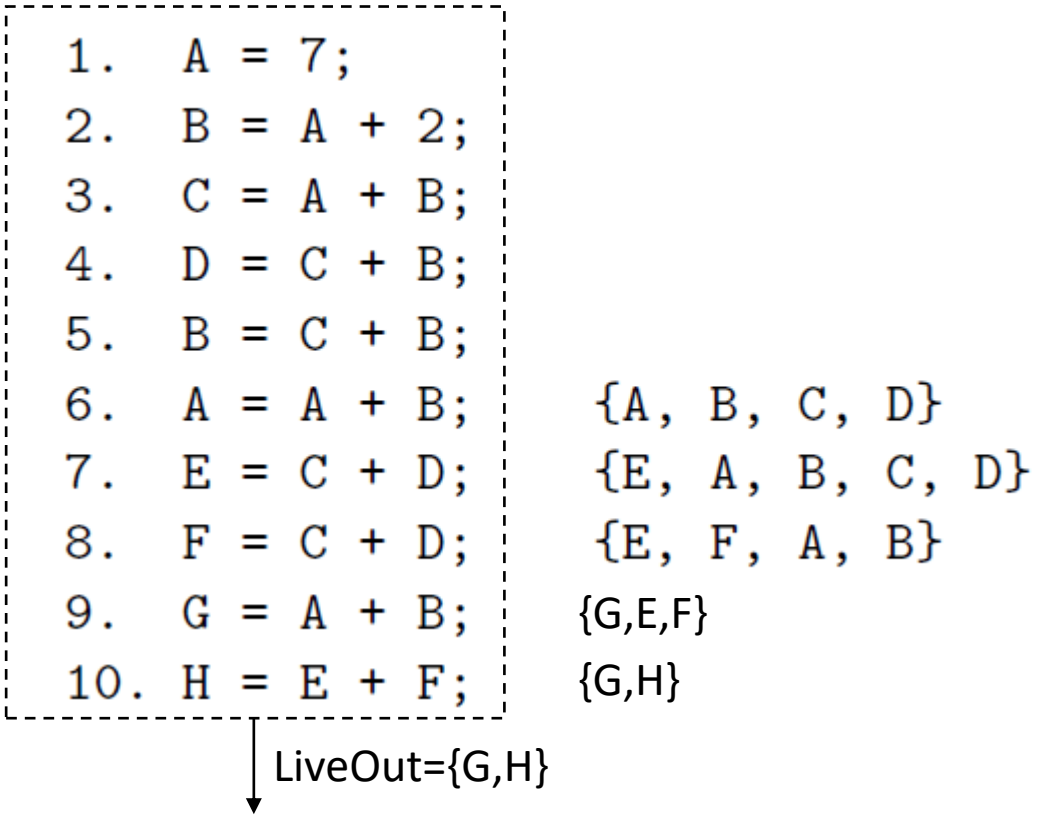
Register Allocation via Graph Coloring - Example



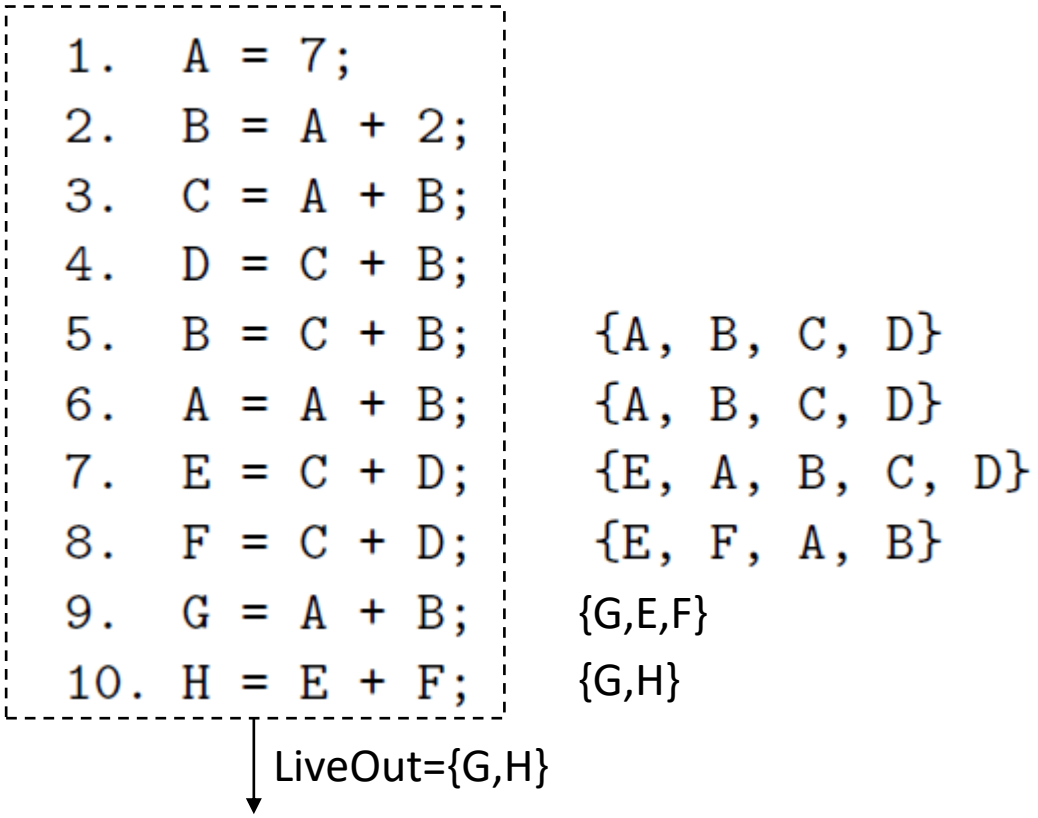
Register Allocation via Graph Coloring - Example



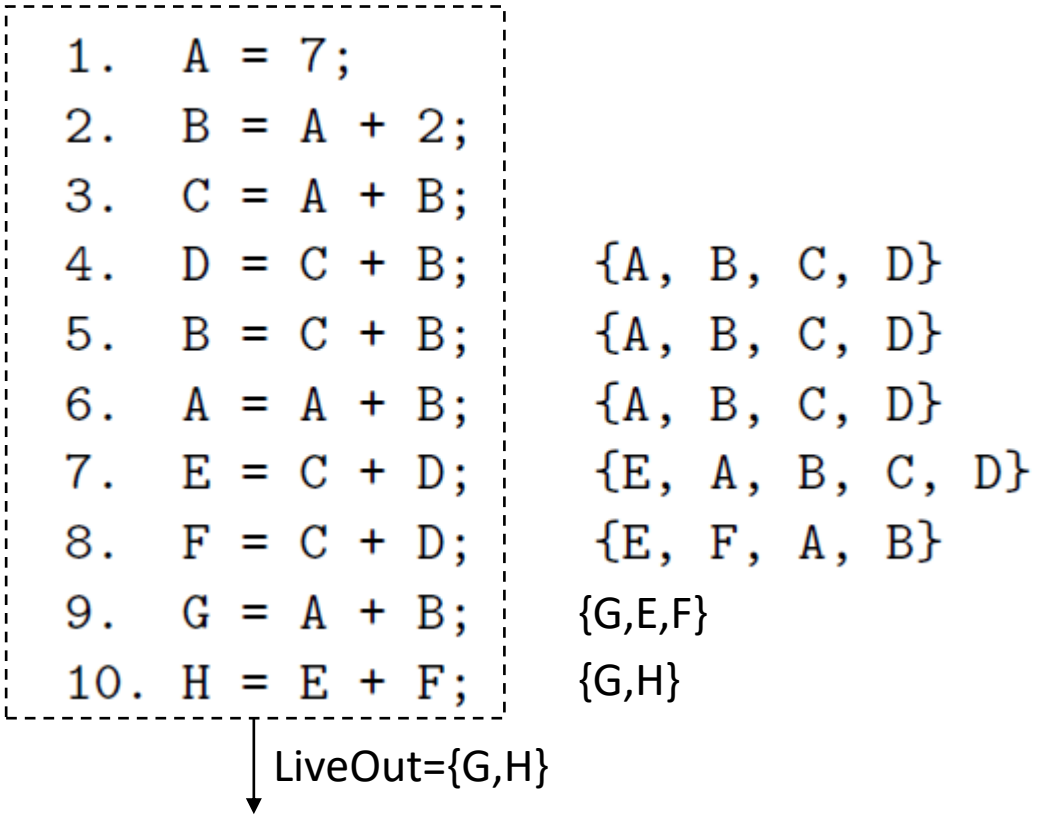
Register Allocation via Graph Coloring - Example



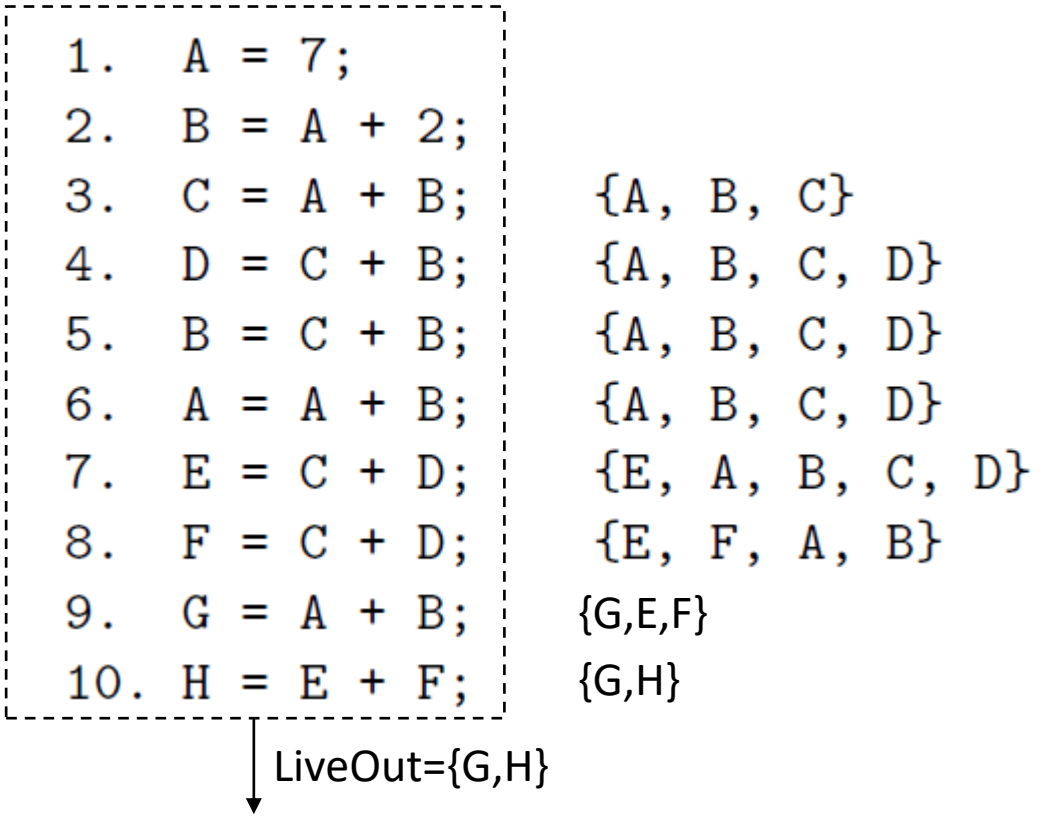
Register Allocation via Graph Coloring - Example



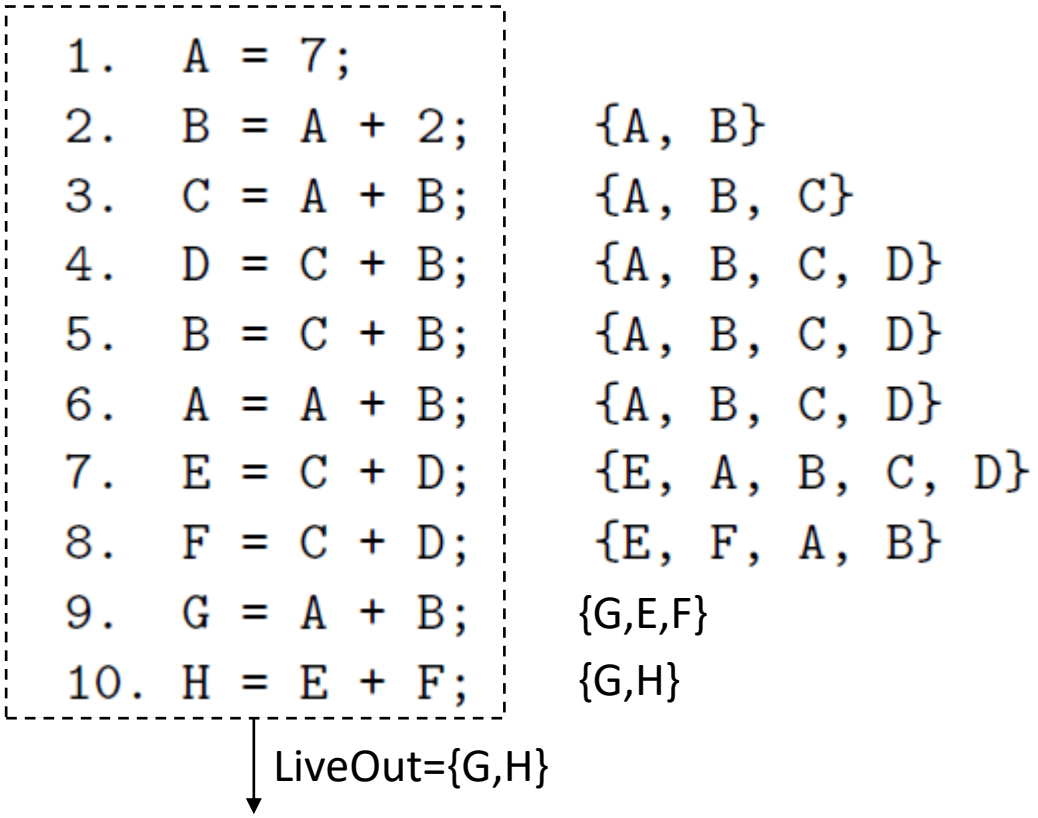
Register Allocation via Graph Coloring - Example



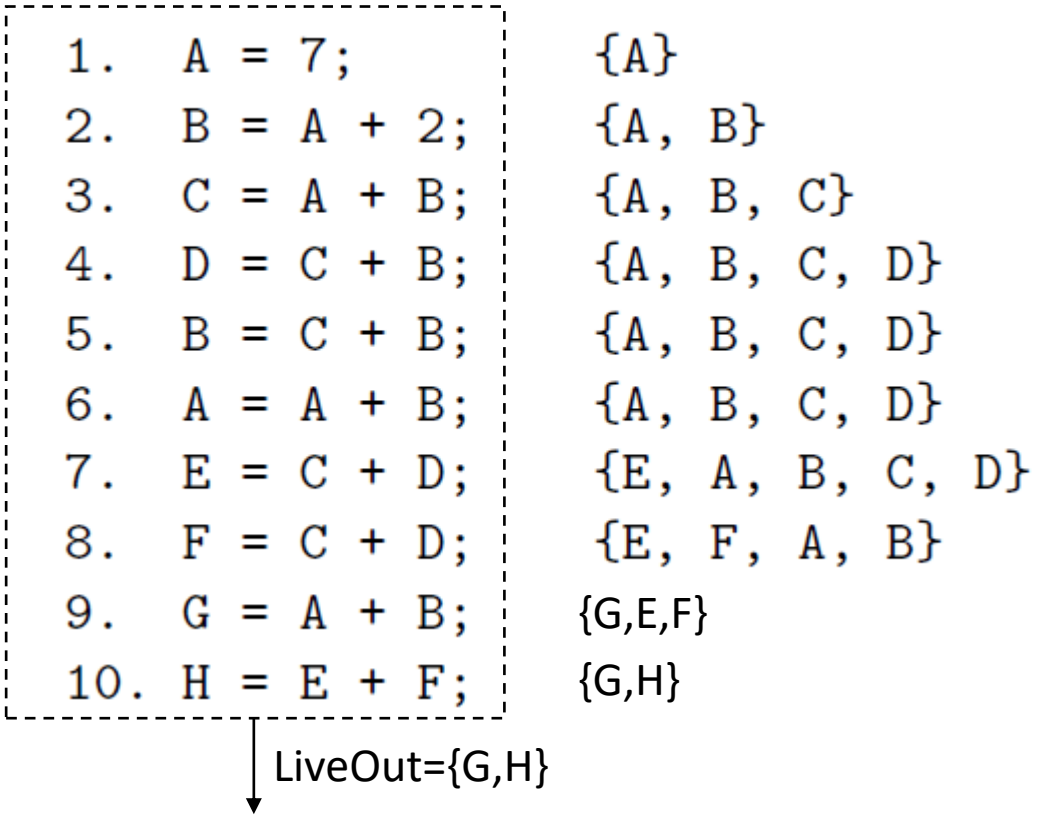
Register Allocation via Graph Coloring - Example



Register Allocation via Graph Coloring - Example

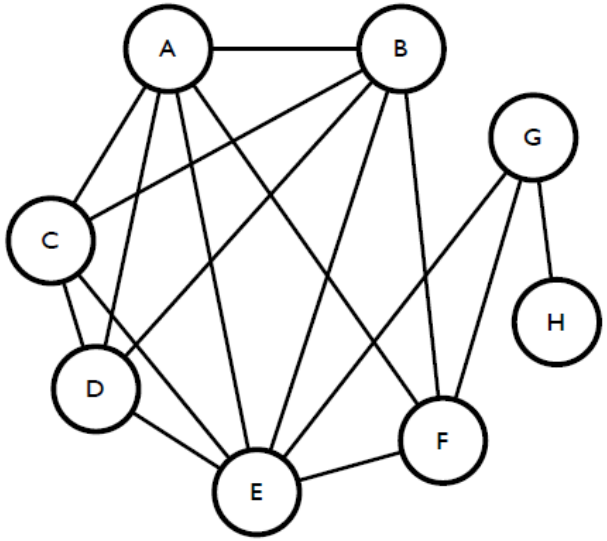


Register Allocation via Graph Coloring - Example



Register Allocation via Graph Coloring - Example

{A}
{A, B}
{A, B, C}
{A, B, C, D}
{A, B, C, D}
{A, B, C, D}
{E, A, B, C, D}
{E, F, A, B}
{G,E,F}
{G,H}



Interference graph

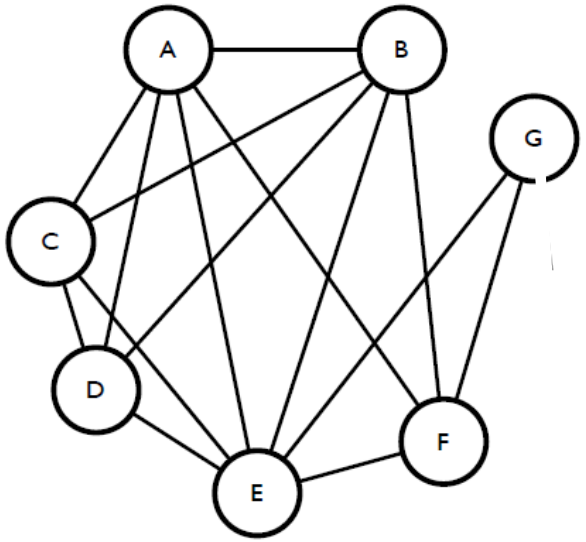
Remove H

Customized rules (3-coloring):

- Remove nodes in reverse alphabetical order
- Spill variables that are used least (spill the variable with most number of edges in case of a tie)

Register Allocation via Graph Coloring - Example

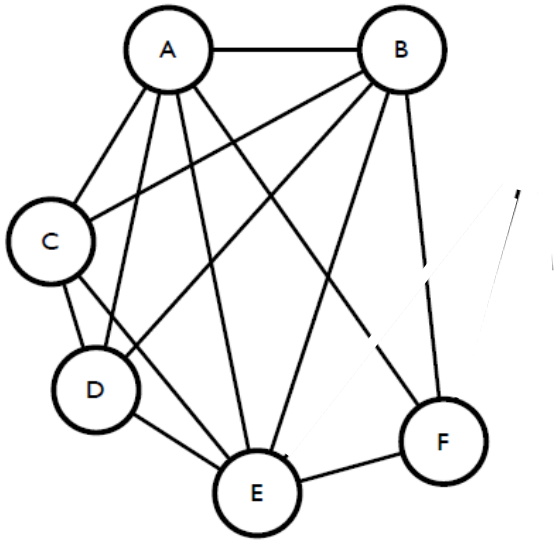
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove G

Register Allocation via Graph Coloring - Example

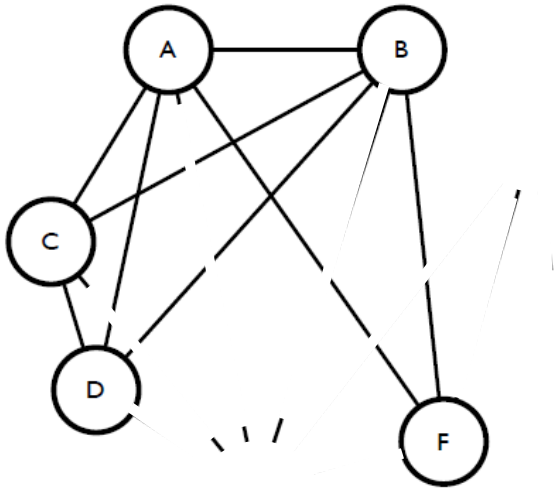
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove E

Register Allocation via Graph Coloring - Example

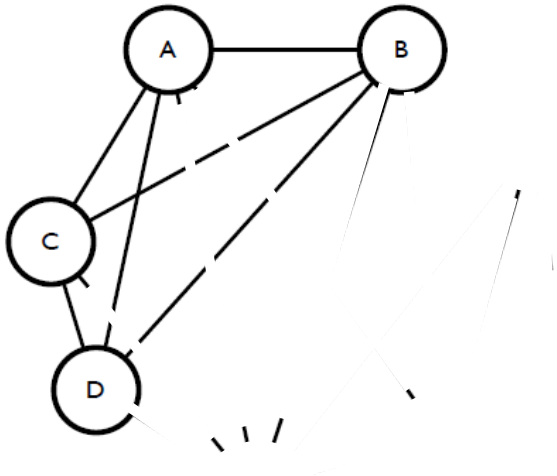
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove F

Register Allocation via Graph Coloring - Example

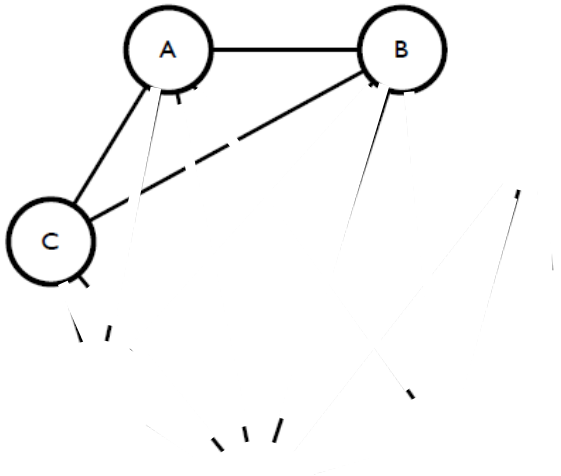
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove D

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Remove C then B then A

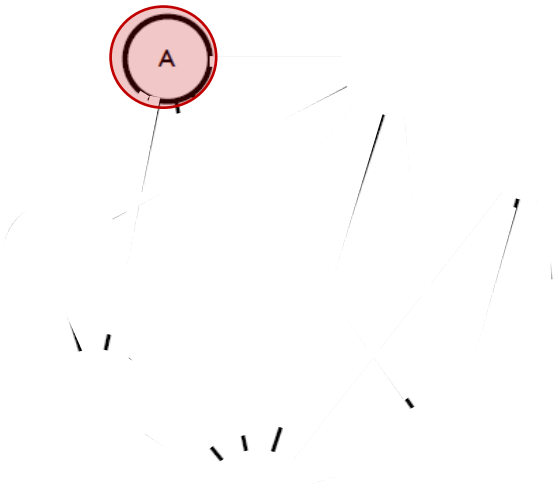
Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```

Stack:
A
B
C
D
F
E
G
H

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```

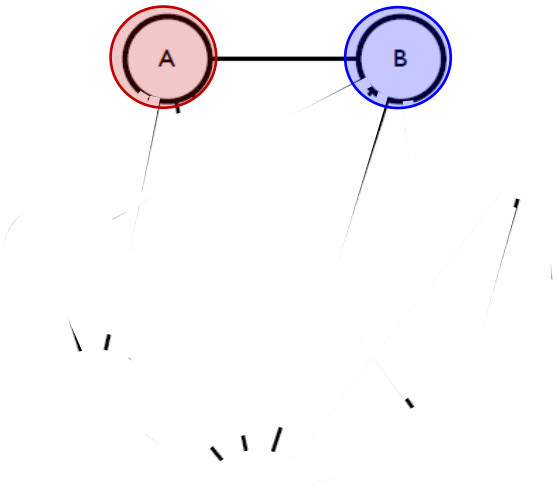


3-Color the variables:

Stack: A - Red
B
C
D
F
E
G
H

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



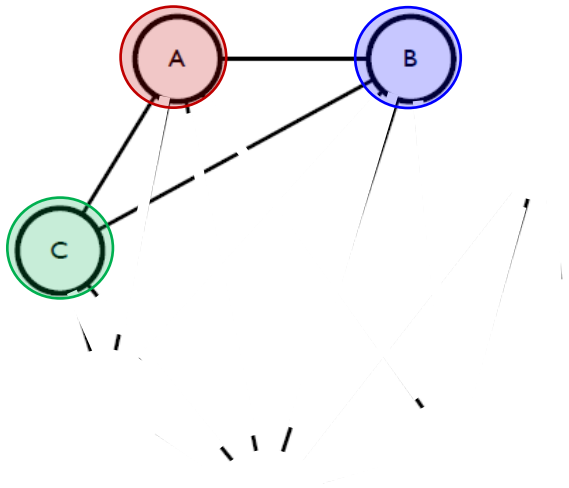
3-Color the variables:

Stack:

B - Blue
C
D
F
E
G
H

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



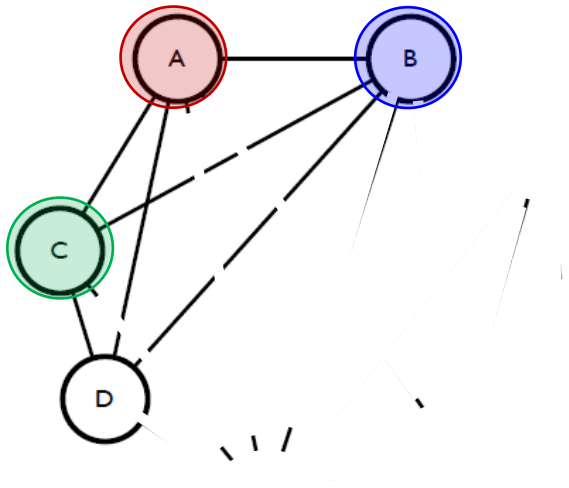
Stack:

C-Green
D
F
E
G
H

3-Color the variables:

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E + F;      {G,H}
```



Stack:

D - ??
F
E
G
H

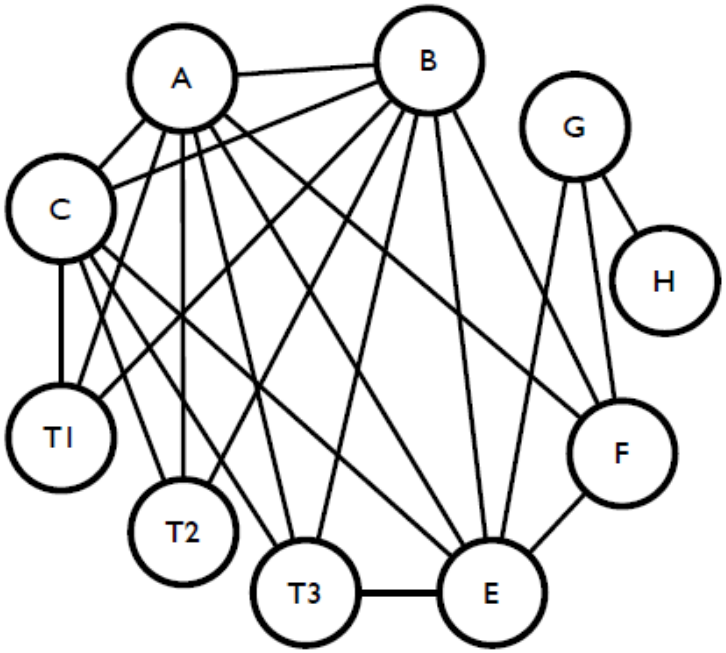
3-Color the variables: Spill D

Register Allocation via Graph Coloring - Example

Earlier code:

```
A = 7;
B = A + 2;
C = A + B;
D = C + B;
B = C + B;
A = A + B;
E = C + D;
F = C + D;
G = A + B;
H = E + F;
```

	Rewritten code:	Liveness info:
1.	A = 7;	{A}
2.	B = A + 2;	{A, B}
3.	C = A + B;	{A, B, C}
4.	T1 = C + B;	{A, B, C, T1}
4'.	ST T1, D	{A, B, C}
5.	B = C + B;	{A, B, C}
6.	A = A + B;	{A, B, C}
6'.	LD D, T2	{A, B, C, T2}
7.	E = C + T2;	{A, B, C, E}
7'.	LD D, T3	{A, B, C, E, T3}
8.	F = C + T3;	{A, B, E, F}
9.	G = A + B;	{G, E, F}
10.	H = E + F;	{G, H}

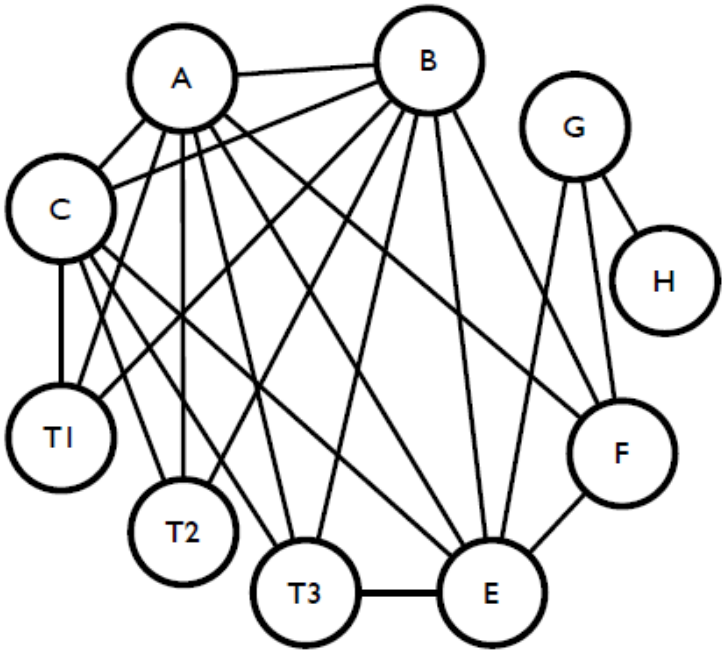


New interference graph

3-Color the variables: Spill D, rewrite code and recalculate liveness

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   T1 = C + B;     {A, B, C, T1}
4'.  ST T1, D        {A, B, C}
5.   B = C + B;      {A, B, C}
6.   A = A + B;      {A, B, C}
6'.  LD D, T2        {A, B, C, T2}
7.   E = C + T2;     {A, B, C, E}
7'.  LD D, T3        {A, B, C, E, T3}
8.   F = C + T3;     {A, B, E, F}
9.   G = A + B;      {G, E, F}
10.  H = E + F;      {G, H}
```

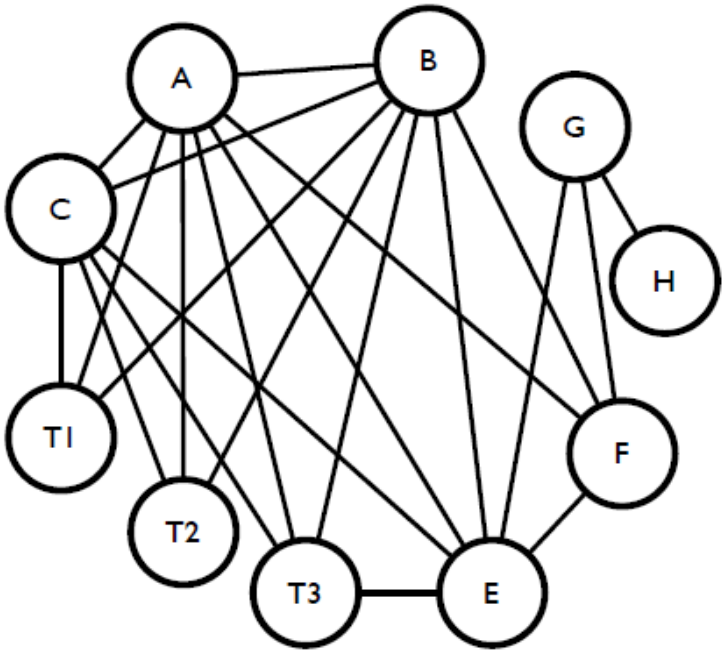


Simplify (step 1)

Stack (left-bottom, right-top): H, G, E, F, C, T1, T2, T3, B, A

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   T1 = C + B;     {A, B, C, T1}
4'.  ST T1, D        {A, B, C}
5.   B = C + B;      {A, B, C}
6.   A = A + B;      {A, B, C}
6'.  LD D, T2        {A, B, C, T2}
7.   E = C + T2;     {A, B, C, E}
7'.  LD D, T3        {A, B, C, E, T3}
8.   F = C + T3;     {A, B, E, F}
9.   G = A + B;      {G, E, F}
10.  H = E + F;      {G, H}
```

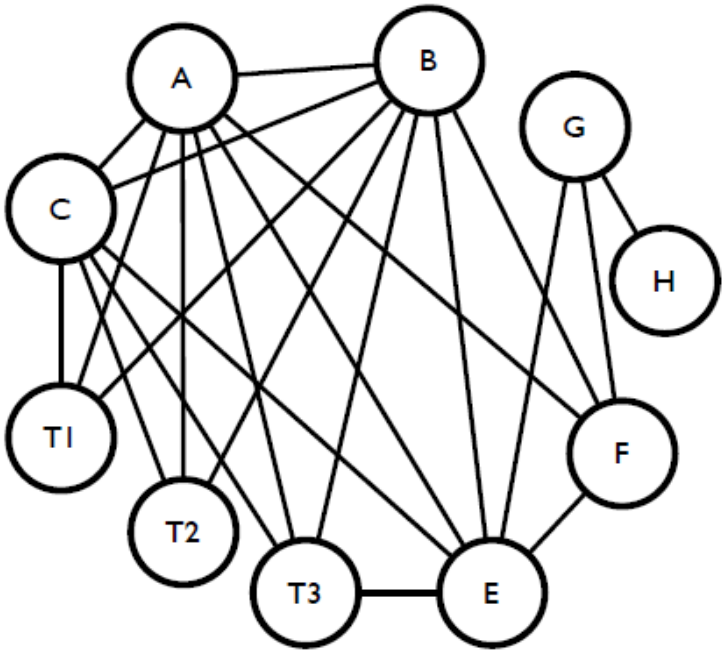


Color (step 2) Stack (left-bottom, right-top): H, G, E, F, C, T1, T2, T3, B, A

Which node must be Spilled now?

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   T1 = C + B;     {A, B, C, T1}
4'.  ST T1, D        {A, B, C}
5.   B = C + B;      {A, B, C}
6.   A = A + B;      {A, B, C}
6'.  LD D, T2        {A, B, C, T2}
7.   E = C + T2;     {A, B, C, E}
7'.  LD D, T3        {A, B, C, E, T3}
8.   F = C + T3;     {A, B, E, F}
9.   G = A + B;      {G, E, F}
10.  H = E + F;      {G, H}
```

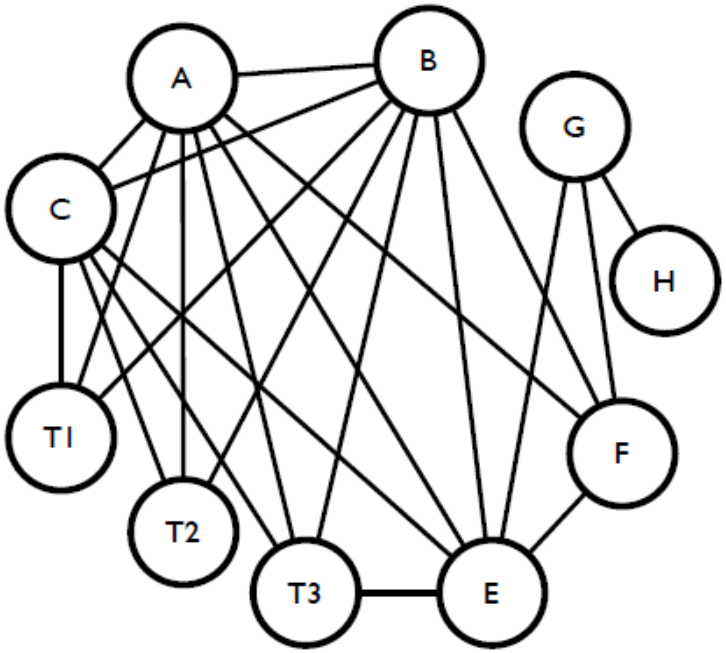


Color (step 2) Stack (left-bottom, right-top): H, G, E, F, C, T1, T2, T3, B, A

Which node must be Spilled now? (i.e. which node can't be colored?)

Register Allocation via Graph Coloring - Example

```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   T1 = C + B;     {A, B, C, T1}
4'.  ST T1, D        {A, B, C}
5.   B = C + B;      {A, B, C}
6.   A = A + B;      {A, B, C}
6'.  LD D, T2        {A, B, C, T2}
7.   E = C + T2;     {A, B, C, E}
7'.  LD D, T3        {A, B, C, E, T3}
8.   F = C + T3;     {A, B, E, F}
9.   G = A + B;      {G, E, F}
10.  H = E + F;      {G, H}
```

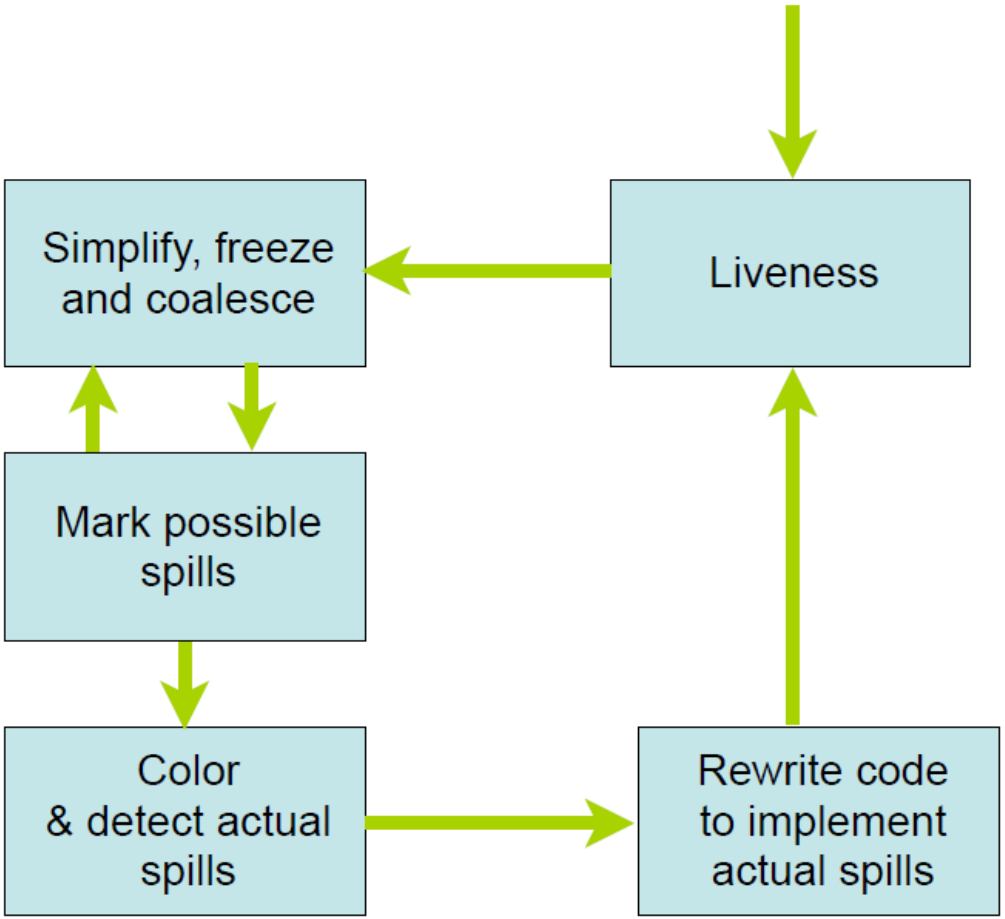


Color (step 2)

Stack (left-bottom, right-top): H, G, E, F, C, T1, T2, T3, B, A

Which node must be Spilled now? (C. Now repeat the steps starting from rewriting the code to spill C, calculating liveness, drawing iteration graph and then simplifying the iteration graph.)

Overall Algorithm



Recap: Optimize Loops

- Low level optimization
  - Moving code around in a single loop
  - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
  - Restructuring loops, often affects multiple loops
  - Examples: loop fusion, loop interchange, loop tiling

Slide Courtesy: Milind Kulkarni

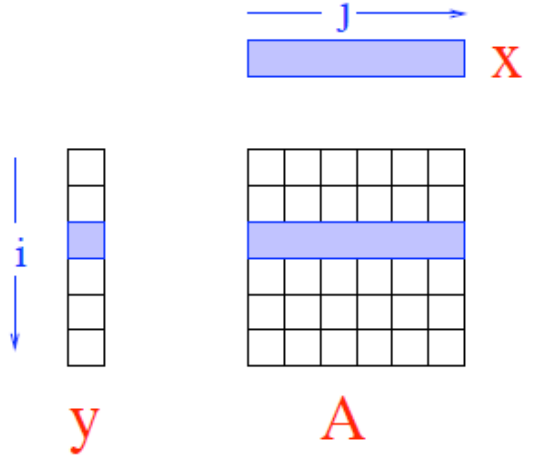
High level loop optimizations

- Many useful compiler optimizations require *restructuring* loops or sets of loops
  - Combining two loops together (*loop fusion*)
  - Switching the order of a nested loop (*loop interchange*)
  - Completely changing the traversal order of a loop (*loop tiling*)
- These sorts of high level loop optimizations usually take place at the AST level (where loop structure is obvious)

Slide Courtesy: Milind Kulkarni

Cache behavior

- Most loop transformations target cache performance
  - Attempt to increase *spatial* or *temporal* locality
  - Locality can be exploited when there is *reuse* of data (for temporal locality) or recent access of nearby data (for spatial locality)
- Loops are a good opportunity for this: many loops iterate through matrices or arrays
- Consider matrix-vector multiply example
  - Multiple traversals of vector: opportunity for spatial and temporal locality
  - Regular access to array: opportunity for spatial locality



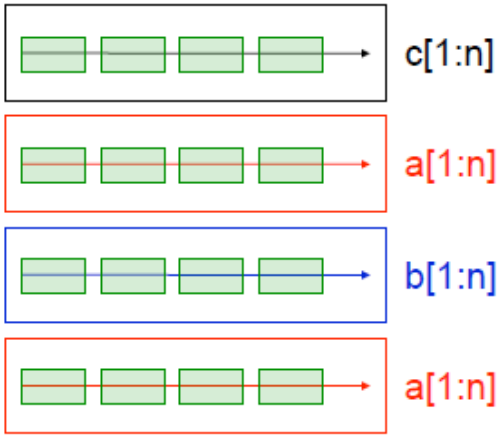
$$y = Ax$$

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```


Slide Courtesy: Milind Kulkarni

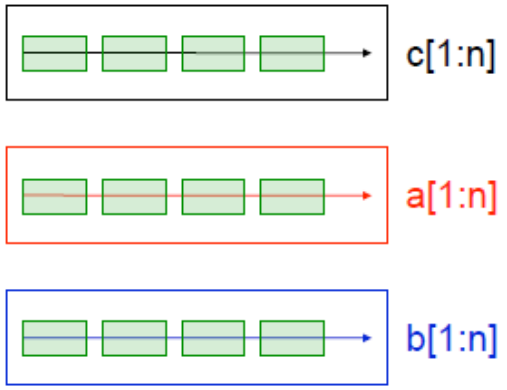
Loop fusion

```
do I = 1, n
   c[i] = a[i]
end do
do I = 1, n
   b[i] = a[i]
end do
```



- Combine two loops together into a single loop
- Why is this useful?
- Is this always legal?

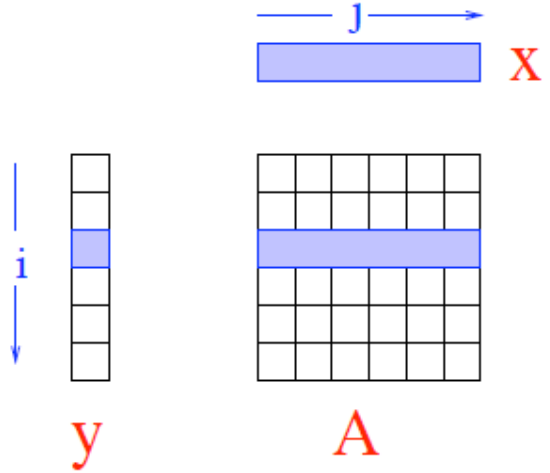
```
do I = 1, n
   c[i] = a[i]
   b[i] = a[i]
end do
```



Slide Courtesy: Milind Kulkarni

Loop interchange

- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
  - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)

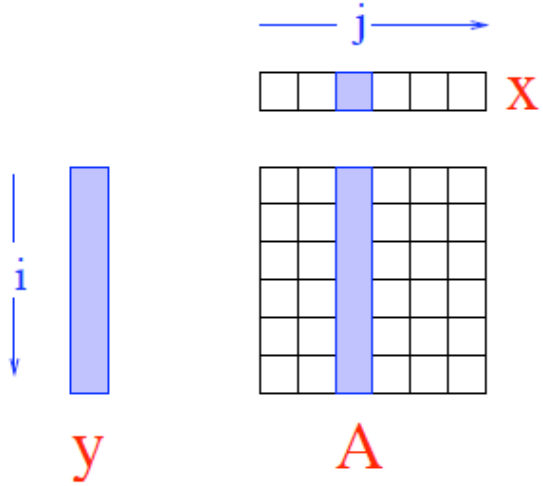



```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```


Slide Courtesy: Milind Kulkarni

Loop interchange

- Change the order of a nested loop
- This is not always legal – it changes the order that elements are accessed!
- Why is this useful?
  - Consider matrix-matrix multiply when A is stored in column-major order (i.e., each column is stored in contiguous memory)




```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    y[i] += A[i][j] * x[j]
```

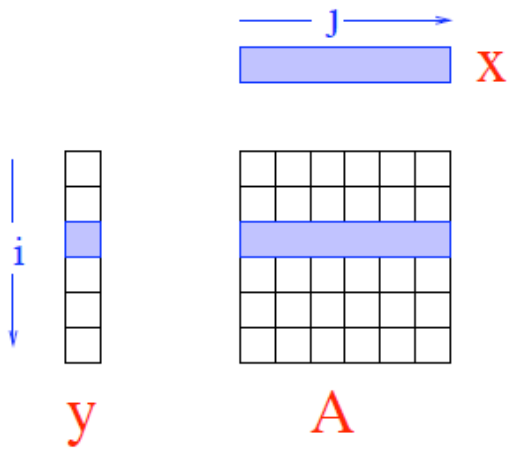

Slide Courtesy: Milind Kulkarni

Loop tiling

- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



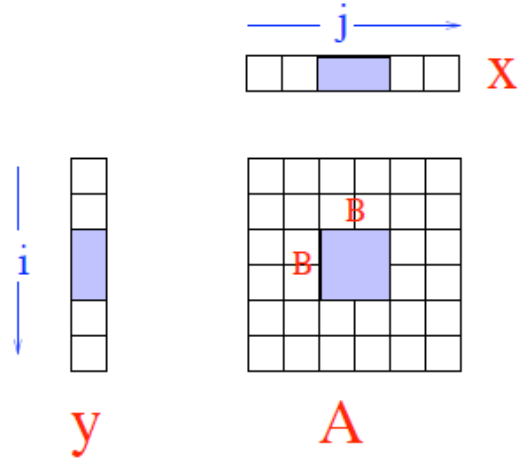
Slide Courtesy: Milind Kulkarni

Loop tiling

- Also called "loop blocking"
- One of the more complex loop transformations
- Goal: break loop up into smaller pieces to get spatial and temporal locality
  - Create new inner loops so that data accessed in inner loops fit in cache
- Also changes iteration order, so may not be legal

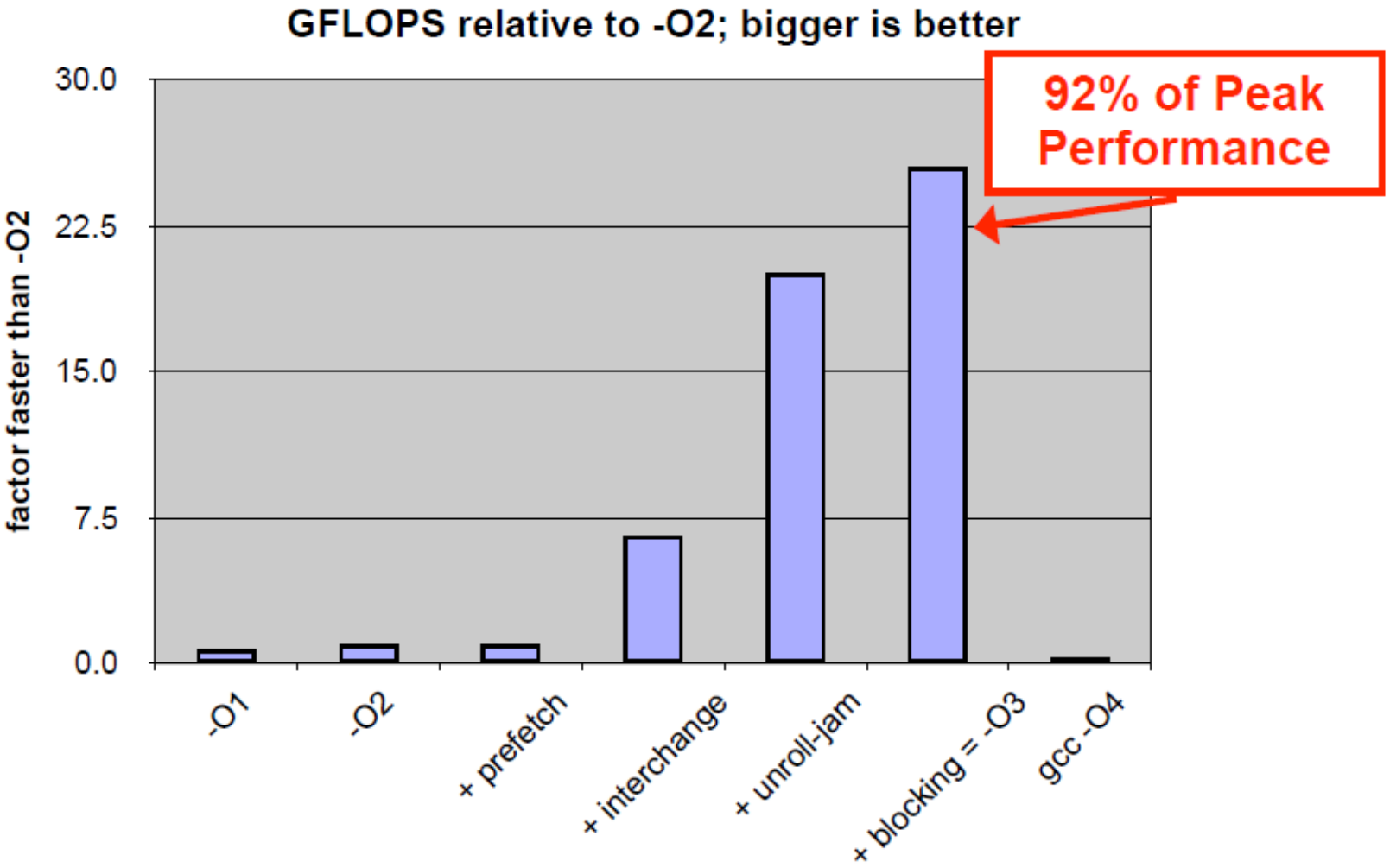
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    y[i] += A[i][j] * x[j]
```

```
for (ii = 0; ii < N; ii += B)
  for (jj = 0; jj < N; jj += B)
    for (i = ii; i < ii+B; i++)
      for (j = jj; j < jj+B; j++)
        y[i] += A[i][j] * x[j]
```



Slide Courtesy: Milind Kulkarni

In a real (Itanium) compiler



Slide Courtesy: Milind Kulkarni

Loop transformations

- Loop transformations can have dramatic effects on performance
- Doing this legally and automatically is very difficult!
- Researchers have developed techniques to determine legality of loop transformations and automatically transform the loop
  - Techniques like *unimodular transform framework* and *polyhedral framework*
  - These approaches will get covered in more detail in advanced compilers course

Slide Courtesy: Milind Kulkarni

Dependence Analysis

(separate set of slides posted)