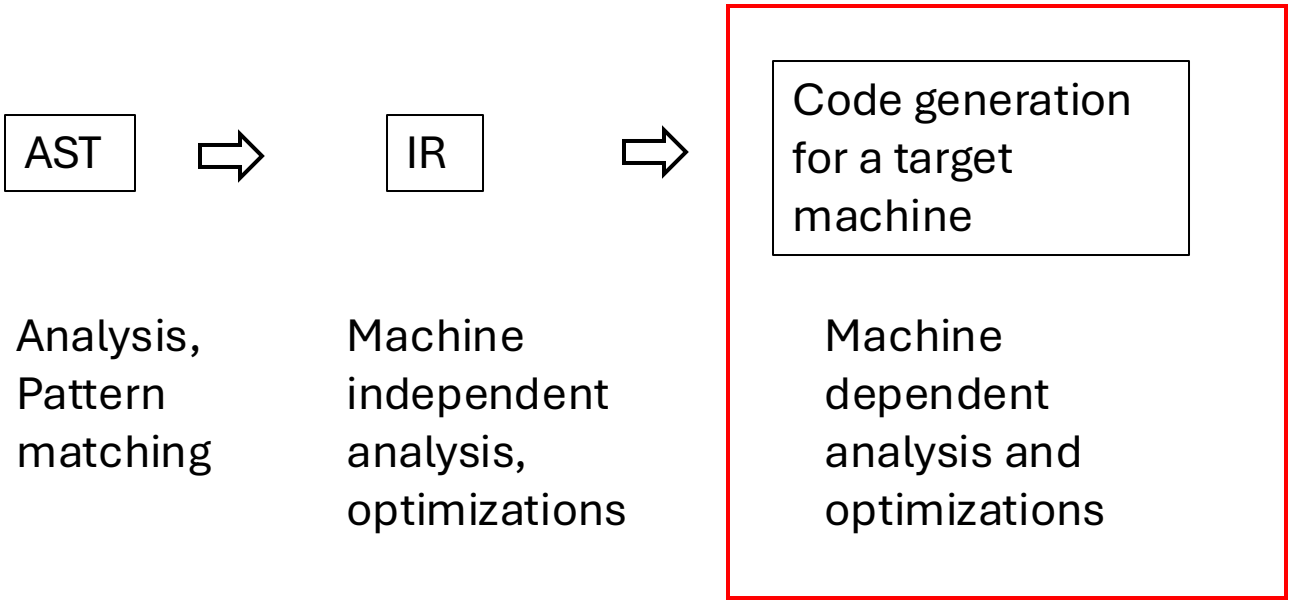
Register Allocation – I

Nikhil Hegde

Compiler Optimizations in LLVM

Lecture series @ QUALCOMM Inc.

Compiler phases so far..



LLVM Code Generator Steps

- Instruction Selection
  - LLVM IR -> DAG
- Scheduling and Formation
  - determine a schedule for DAG nodes
- SSA-based Machine Code Optimization
  - e.g. peephole optimizations
- Register Allocation
  - Unlimited virtual registers to limited machine registers. Introduce spill code if required.
- Prolog/Epilog Code generation
  - Function call conventions, frame pointer elimination
- Late Machine Code Optimization
  - Spill code scheduling, peephole optimizations
- Code Emission
  - Assembler code or direct machine code

https://llvm.org/docs/CodeGenerator.html

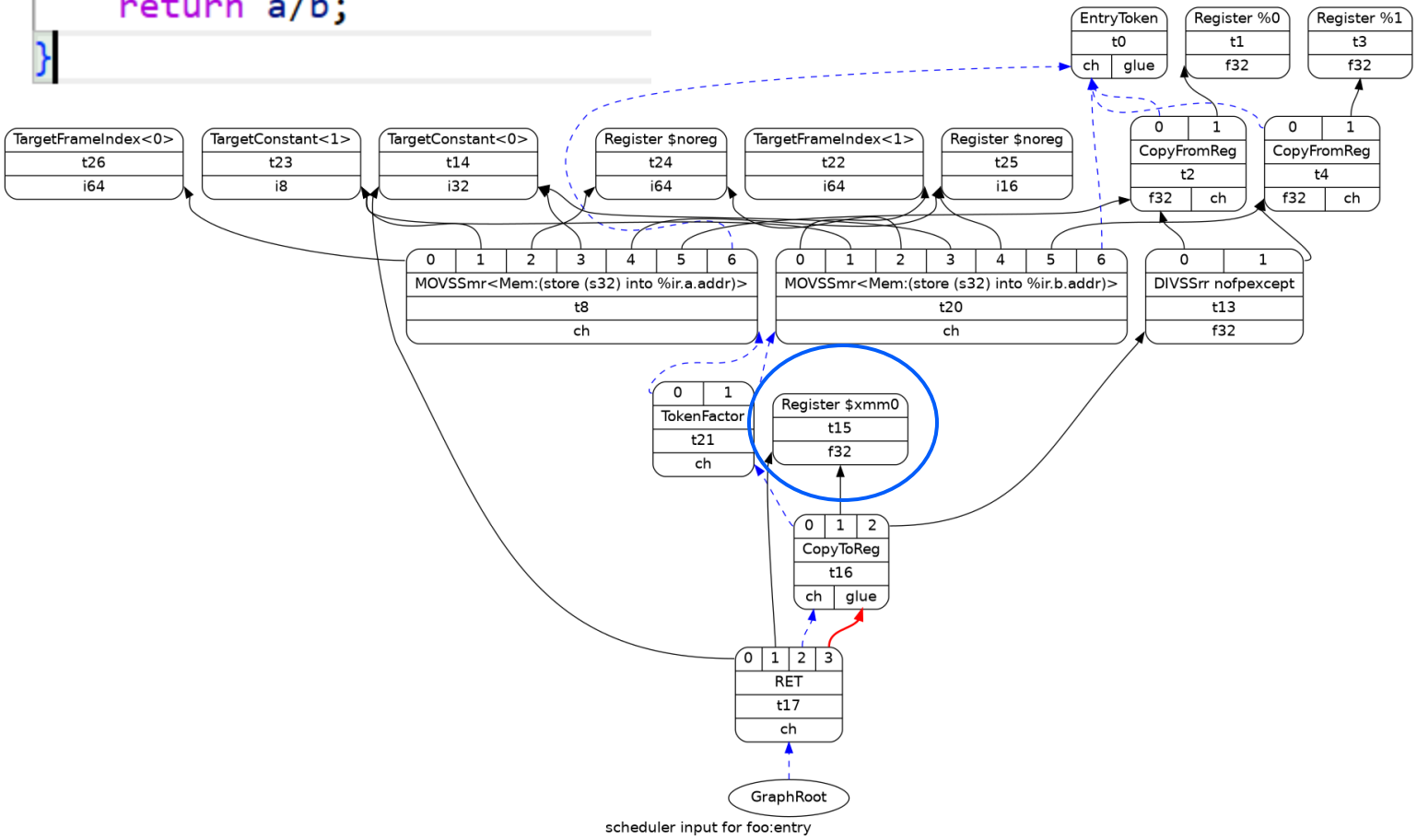
Recall what we got after instruction selection - GlobalISel

```
1  float foo(float a, float b){
2      return a/b;
3  }
```

```
109    %0:fr32 = COPY $xmm0
110    %1:fr32 = COPY $xmm1
111    %7:gr32 = COPY %0
112    MOV32mr %stack.0.a.addr, 1, $noreg, 0,
113    %8:gr32 = COPY %1
114    MOV32mr %stack.1.b.addr, 1, $noreg, 0,
115    %4:fr32 = MOVSSrm_alt %stack.0.a.addr,
116    %5:fr32 = MOVSSrm_alt %stack.1.b.addr,
117    %6:fr32 = nofpexcept DIVSSrr %4, %5, im
118    $xmm0 = COPY %6
119    RET 0, implicit $xmm0
```

Recall what we got after instruction selection - SelectionDAG

```
1  float foo(float a, float b){
2      return a/b;
3  }
```



Register Allocation

• Simple code generation (in CSE example): use a register for each temporary, load from a variable on each read, store to a variable at each write

•What are the problems?

- •Real machines have a limited number of registers – one register per temporary may be too many
- • Loading from and storing to variables on each use may produce a lot of redundant loads and stores

Register Allocation

- Goal: allocate temporaries and variables to registers to:
  - Use only as many registers as machine supports
  - Minimize loading and storing variables to memory (keep variables in registers when possible)
  - Minimize putting temporaries on stack (“spilling”)

Global vs. Local

- Same distinction as global vs. local CSE
  - Local register allocation is for a single basic block
  - Global register allocation is for an entire function

Does inter-procedural register allocation make sense? Why? Why not?

Hint: think about caller-save, callee-save registers

When we handle function calls, registers are pushed/popped from stack

Top-down register allocation

- For each basic block
  - Find the number of references of each variable
  - Assign registers to variables with the most references
- Details
  - Keep some registers free for operations on unassigned variables and spilling
  - Store *dirty* registers at the end of BB (i.e., registers which have variables assigned to them)
    - Do not need to do this for temporaries (why?)

Bottom-up register allocation

- Smarter approach:
  - Free registers once the data in them isn't used anymore
- Requires calculating *liveness*
  - A variable is live if it has a value that *may* be used in the future
- Easy to calculate if you have a single basic block:
  - Start at end of block, all local variables marked dead
    - If you have multiple basic blocks, all local variables defined in the block should be *live* (they may be used in the future)
  - When a variable is used, mark as live, record use
  - When a variable is defined, record def, variable dead above this
  - Creates chains linking uses of variables to where they were defined
- We will discuss how to calculate this across BBs later

Bottom-up register allocation

For each tuple op A B C in a BB, do
  R_x = ensure(A)
  R_y = ensure(B)
  if A *dead* after this tuple, free(R_x)
  if B *dead* after this tuple, free(R_y)
  R_z = allocate(C) //could use R_x or R_y
  generate code for op
  mark R_z *dirty*

At end of BB, for each dirty register
  generate code to store register into appropriate variable

- We will present this as if A, B, C are variables in memory. Can be modified to assume that A, B and C are in virtual registers, instead

Bottom-up register allocation

```
ensure(opr)
    if opr is already in register r
        return r
    else
        r = allocate(opr)
        generate load from opr into r
        return r
```

Bottom-up register allocation - Example

	Live	Registers R1	R2	R3	R4	
1: A = 7	{A}	A*				mov 7 r1
2: B = A + 2	{A, B}	A*	B*			add r1 2 r2
3: C = A + B	{A, B, C}	A*	B*	C*		add r1 r2 r3
4: D = A + B	{B, C, D}	D*	B*	C*		add r1 r2 r1
5: A = C + B	{A, B, C, D}	D*	B*	C*	A*	add r3 r2 r4
6: B = C + B	{A, B, C, D}	D*	B*	C*	A*	add r3 r2 r2
7: E = C + D	{A, B, C, D, E}	D*	E*	C*	A*	add r3 r1 r2
8: F = C + D	{A, B, E, F}	F*	E*		A*	add r3 r1 r1
9: G = A + B	{E, F, G}	F*	E*	G*		add r4 r3 r3
10: H = E + F	{H, G}	H*		G*		add r2 r1 r1
11: I = H + G	{I}	I*				add r1 r3 r1
12: WRITE(I)	{}					write r1

(free r1 – dead)

st r2 B;

(Free dead)

ld b r3;

(spill r2 – farthest, store if live and dirty)

(Load since B not in reg. Free dead regs)

Slides on Graph Coloring and Register Allocation

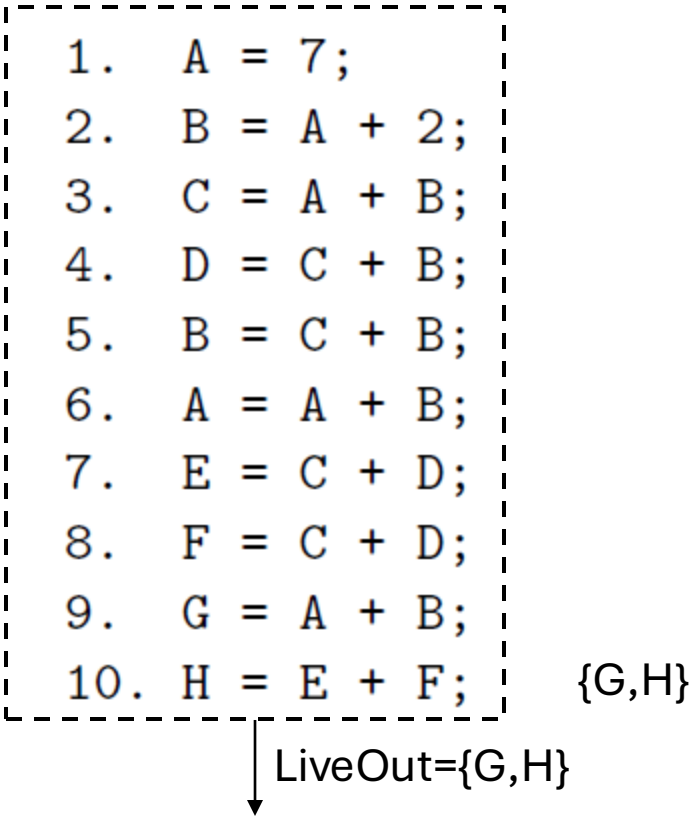
- Refer to global_reg_allocation_andrew_myers.pdf
- Refer to https://www.cs.cmu.edu/afs/cs/academic/class/15745-s18/www/lectures/L12-Register-Allocation.pdf
  (on live ranges. Slides 9-14)

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

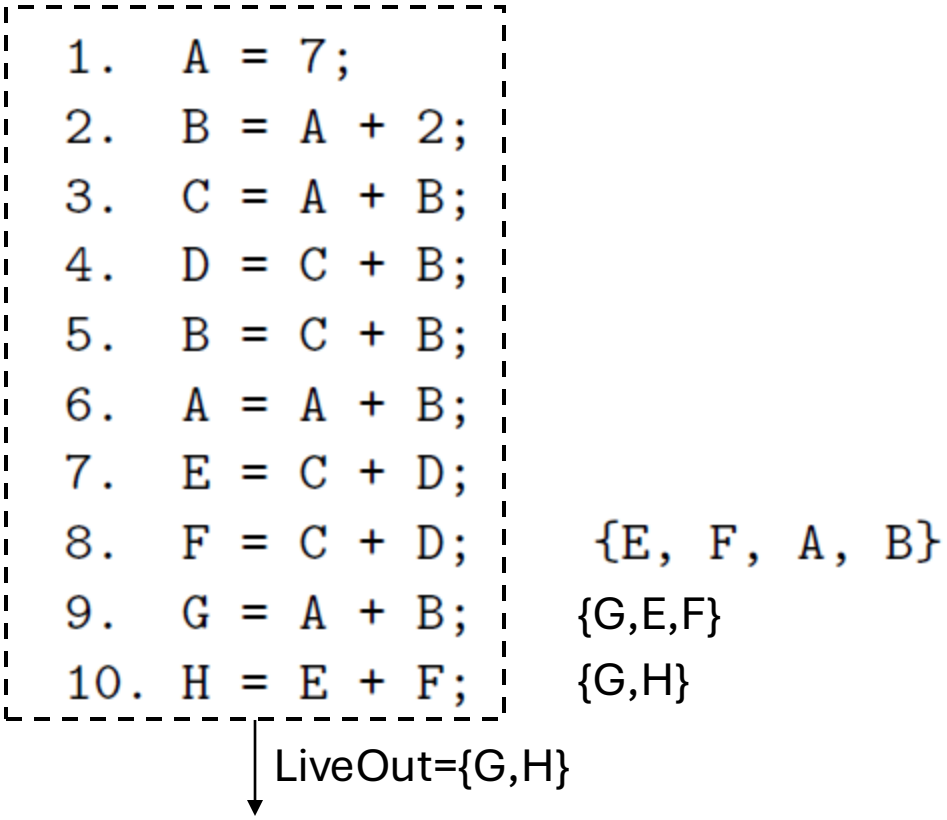
```
1.  A = 7;
2.  B = A + 2;
3.  C = A + B;
4.  D = C + B;
5.  B = C + B;
6.  A = A + B;
7.  E = C + D;
8.  F = C + D;
9.  G = A + B;
10. H = E + F;
```

{G,E,F}

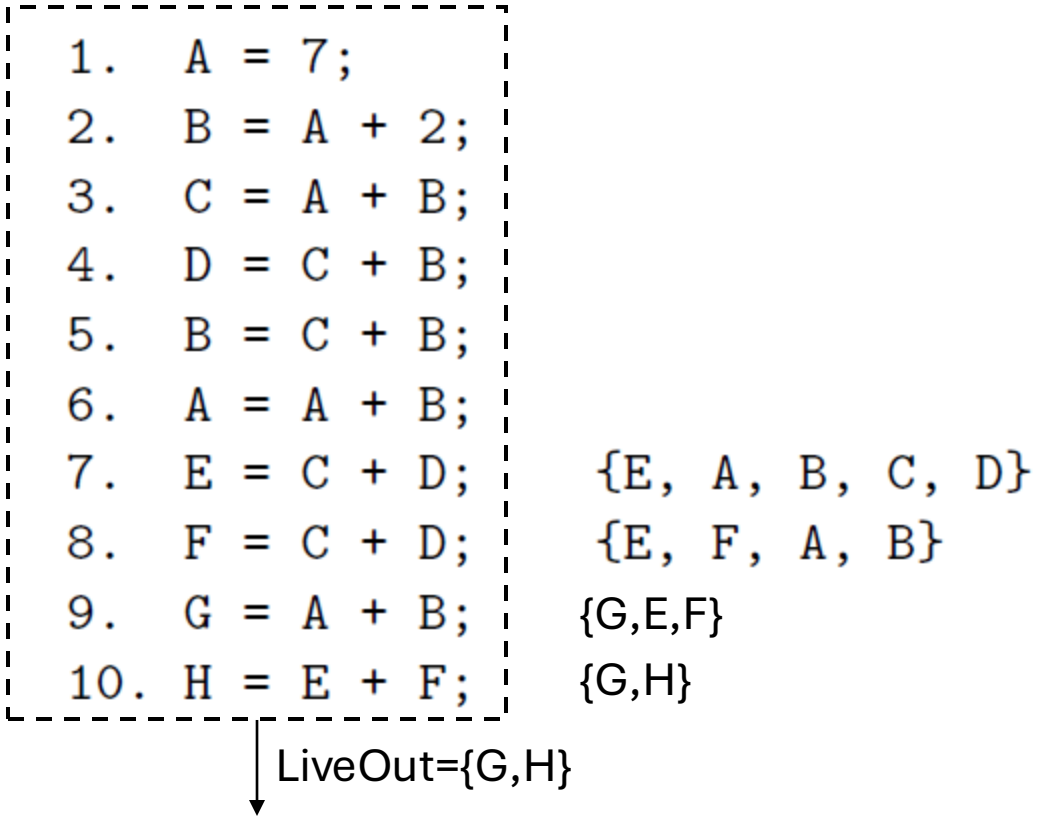
{G,H}

LiveOut={G,H}

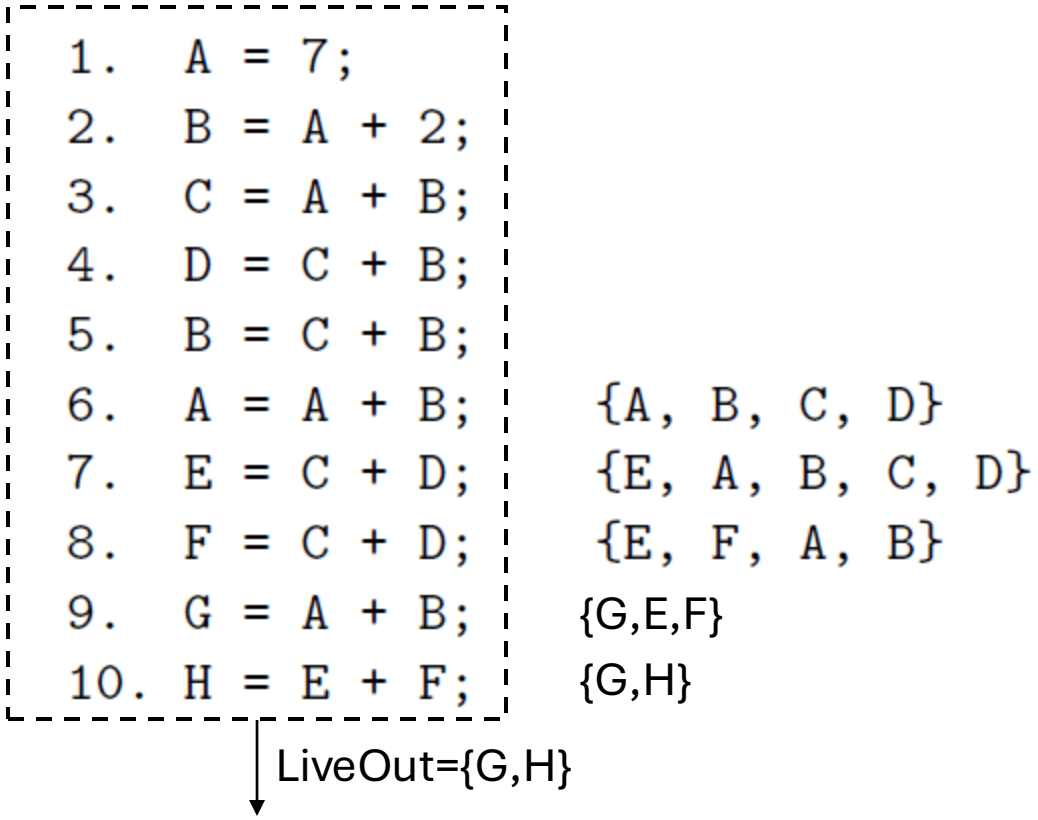
Register Allocation via Graph Coloring - Example



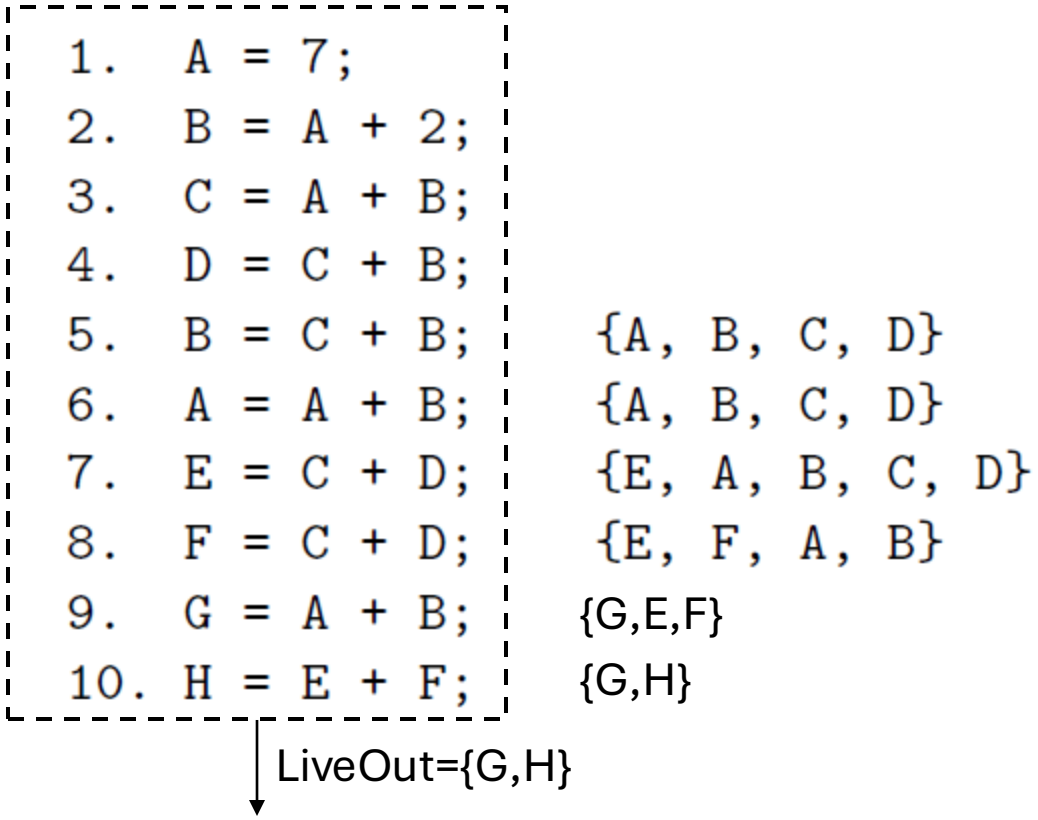
Register Allocation via Graph Coloring - Example



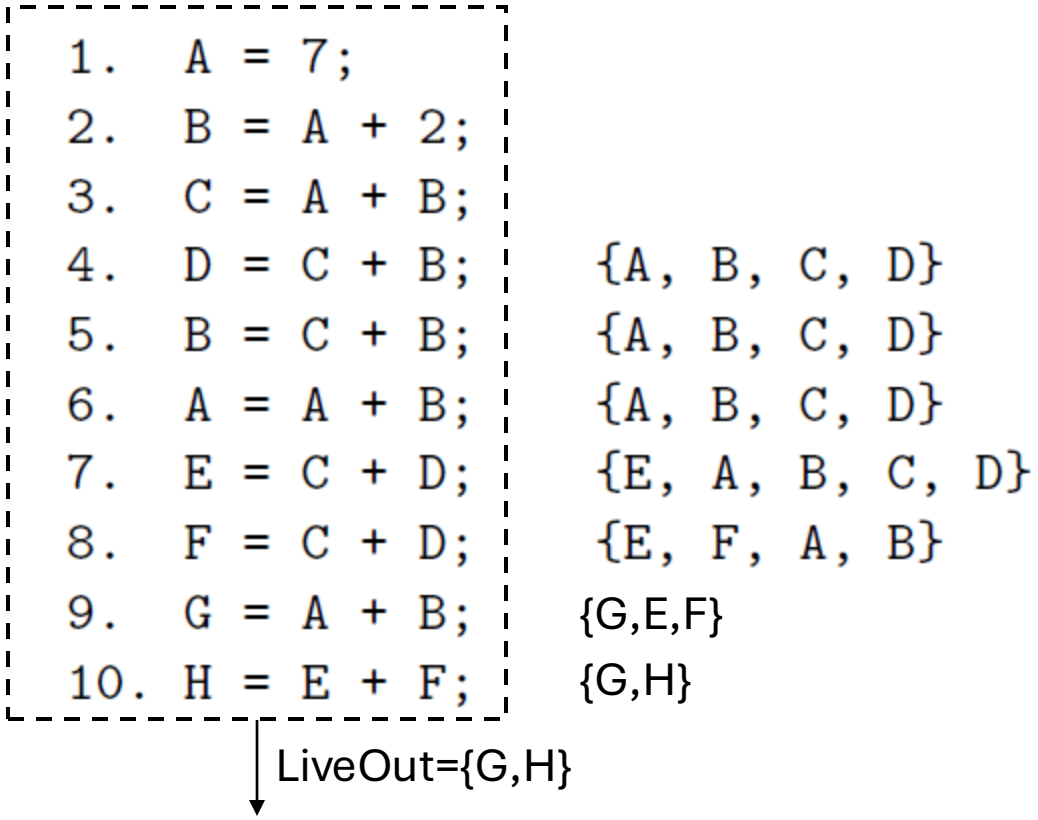
Register Allocation via Graph Coloring - Example



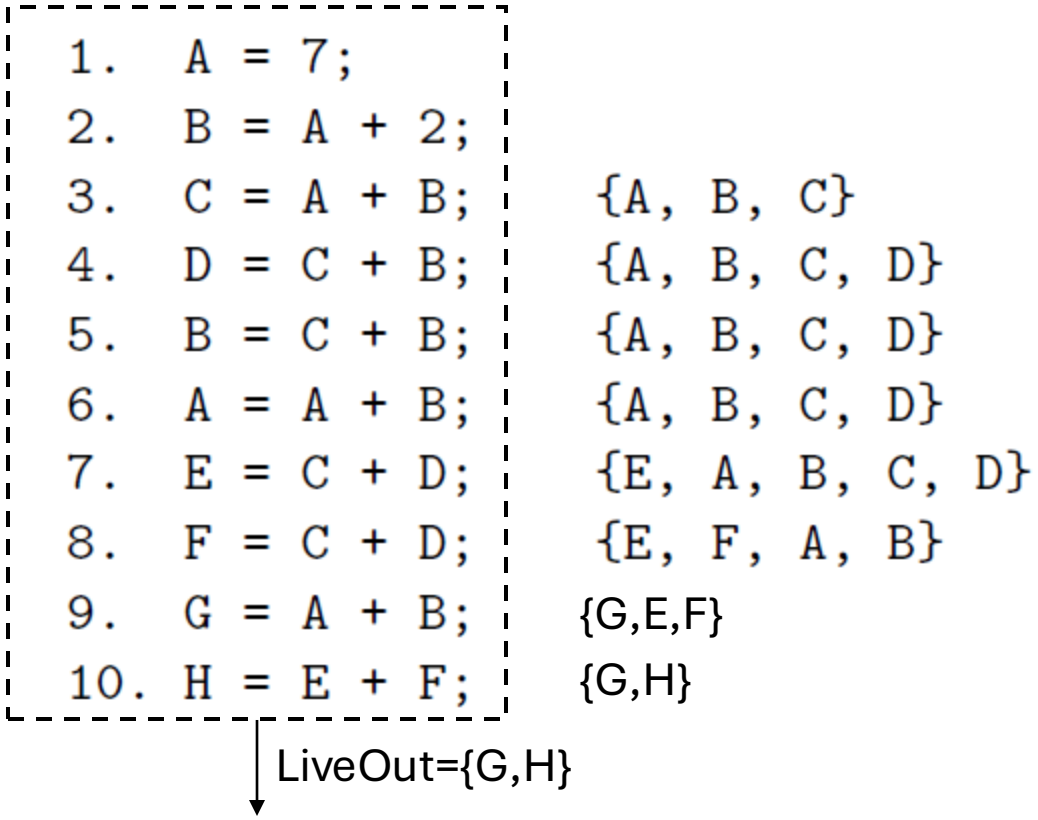
Register Allocation via Graph Coloring - Example



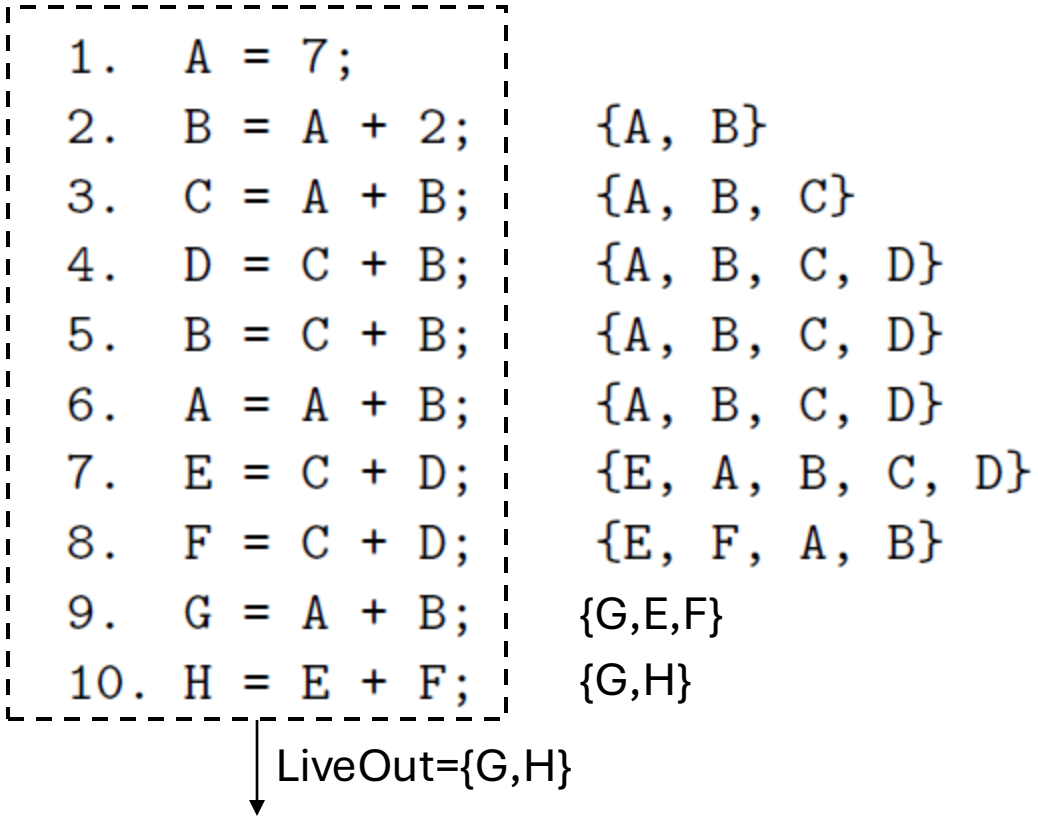
Register Allocation via Graph Coloring - Example



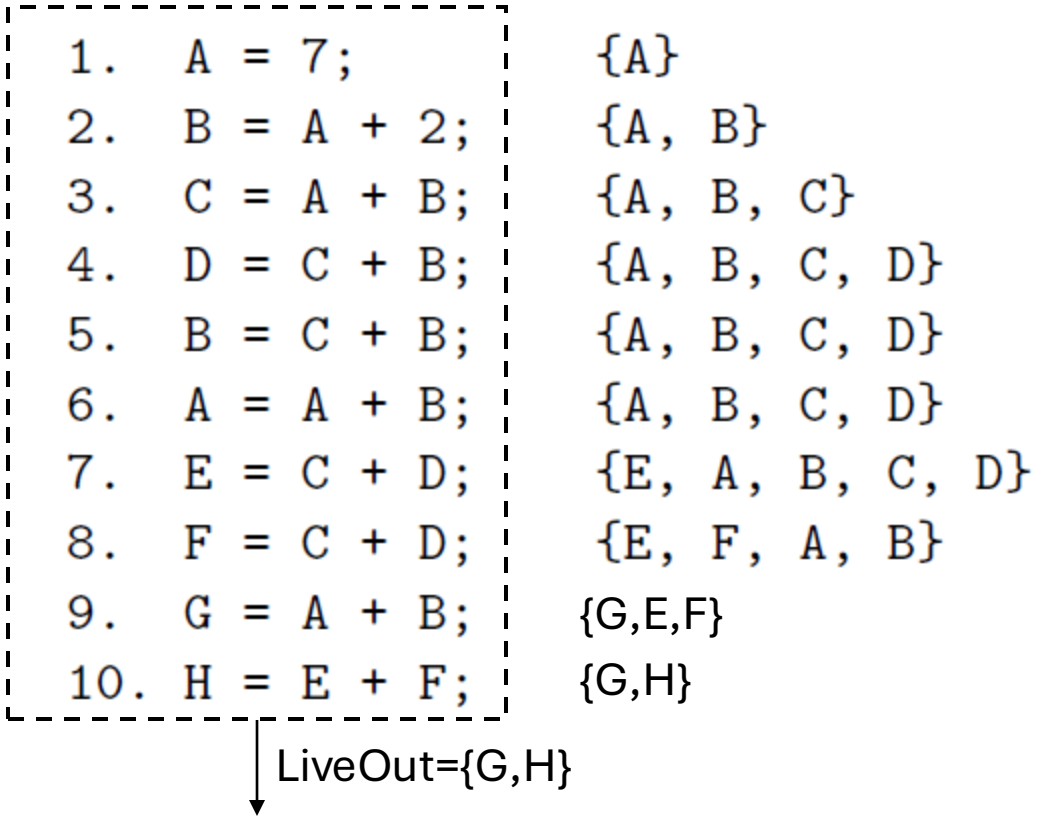
Register Allocation via Graph Coloring - Example



Register Allocation via Graph Coloring - Example

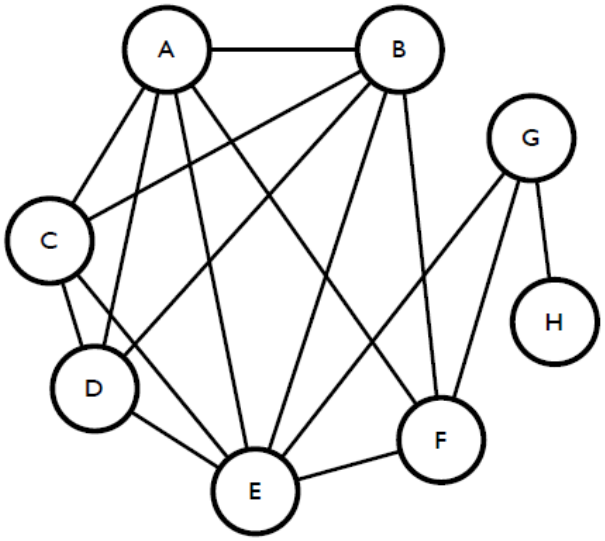


Register Allocation via Graph Coloring - Example



Register Allocation via Graph Coloring - Example

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}



Interference graph

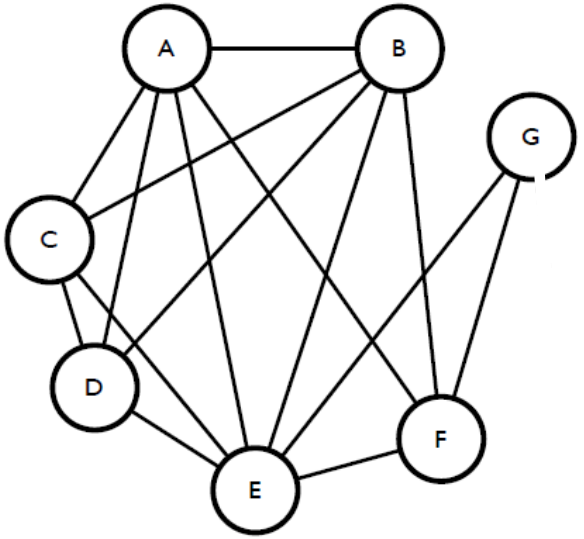
Remove H

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

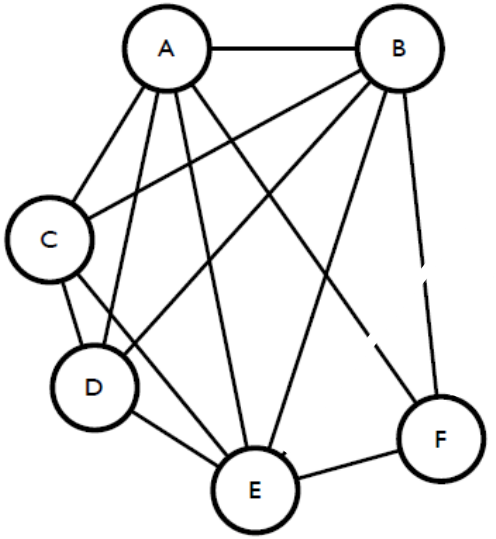
```
1.  A = 7;          {A}
2.  B = A + 2;      {A, B}
3.  C = A + B;      {A, B, C}
4.  D = C + B;      {A, B, C, D}
5.  B = C + B;      {A, B, C, D}
6.  A = A + B;      {A, B, C, D}
7.  E = C + D;      {E, A, B, C, D}
8.  F = C + D;      {E, F, A, B}
9.  G = A + B;      {G,E,F}
10. H = E + F;     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



Remove G

Register Allocation via Graph Coloring - Example

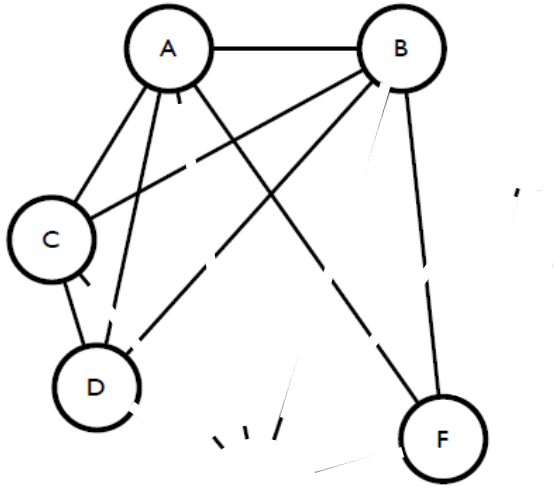
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E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



Remove E

Register Allocation via Graph Coloring - Example

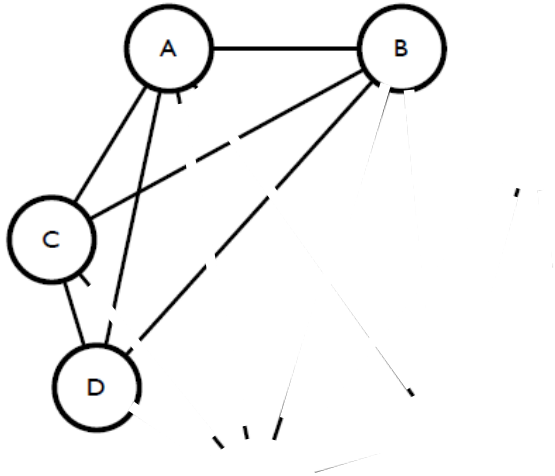
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E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



Remove F

Register Allocation via Graph Coloring - Example

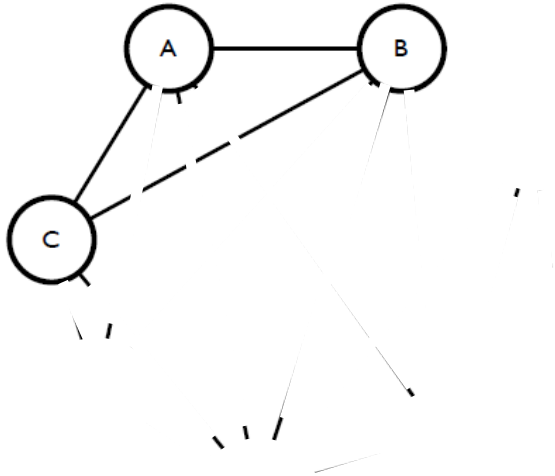
```
1.   A = 7;          {A}
2.   B = A + 2;      {A, B}
3.   C = A + B;      {A, B, C}
4.   D = C + B;      {A, B, C, D}
5.   B = C + B;      {A, B, C, D}
6.   A = A + B;      {A, B, C, D}
7.   E = C + D;      {E, A, B, C, D}
8.   F = C + D;      {E, F, A, B}
9.   G = A + B;      {G,E,F}
10.  H = E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

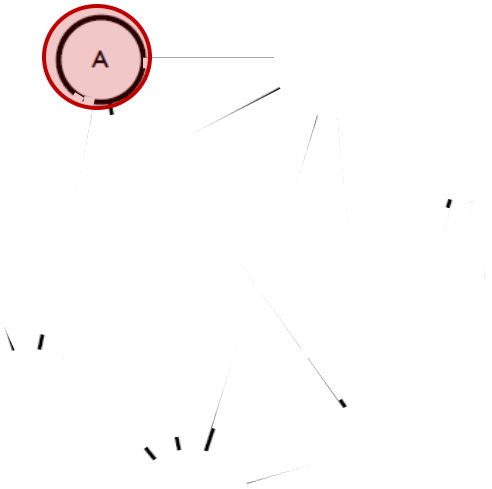
```
1.  A = 7;          {A}
2.  B = A + 2;      {A, B}
3.  C = A + B;      {A, B, C}
4.  D = C + B;      {A, B, C, D}
5.  B = C + B;      {A, B, C, D}
6.  A = A + B;      {A, B, C, D}
7.  E = C + D;      {E, A, B, C, D}
8.  F = C + D;      {E, F, A, B}
9.  G = A + B;      {G,E,F}
10. H = E + F;     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



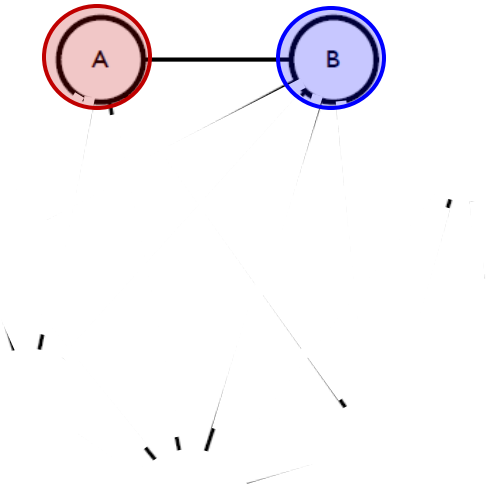
Stack: A - Red

B
C
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:

B -

Blue

C

D

F

E

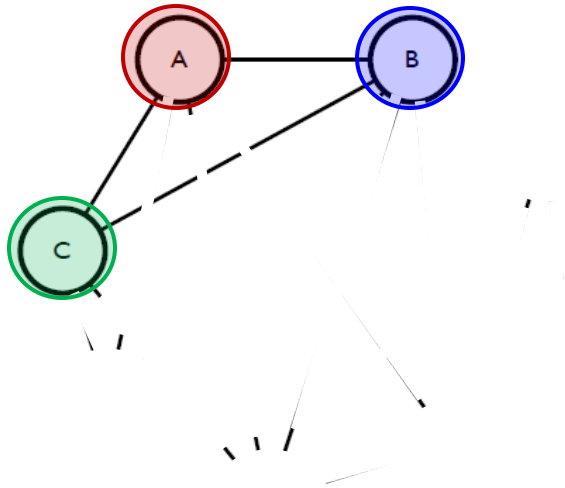
G

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:

C-

Green

D

F

E

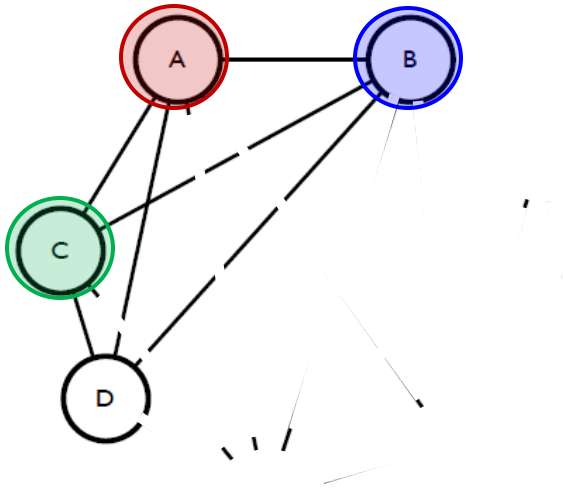
G

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

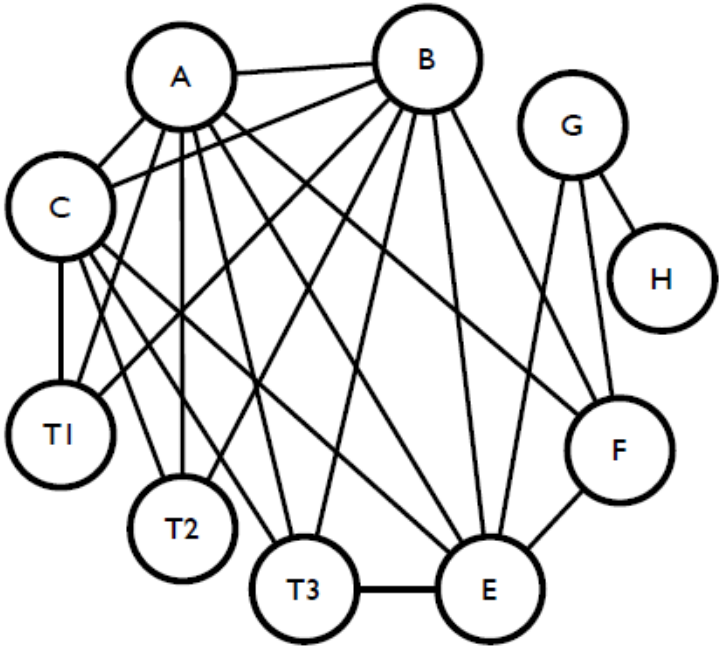
3-Color the variables: Spill D

Register Allocation via Graph Coloring - Example

Earlier code:

```
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;
```

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}

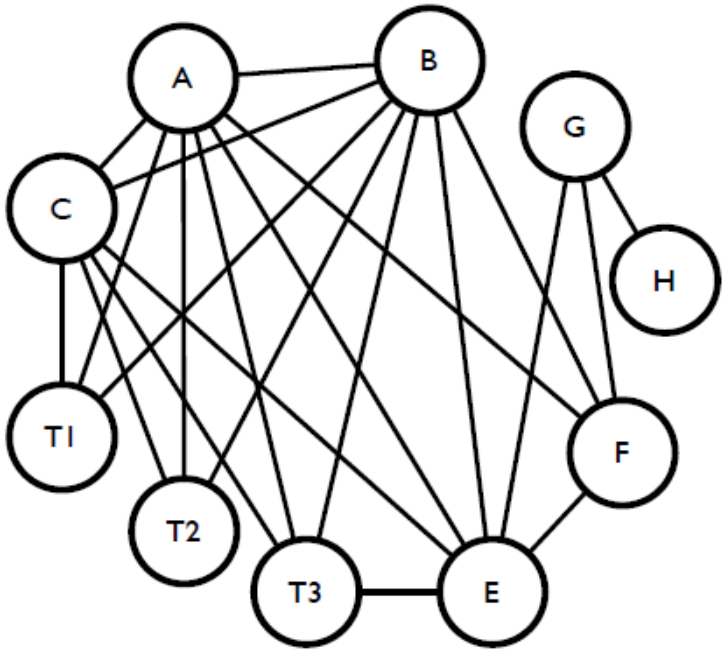


New interference graph

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}
```

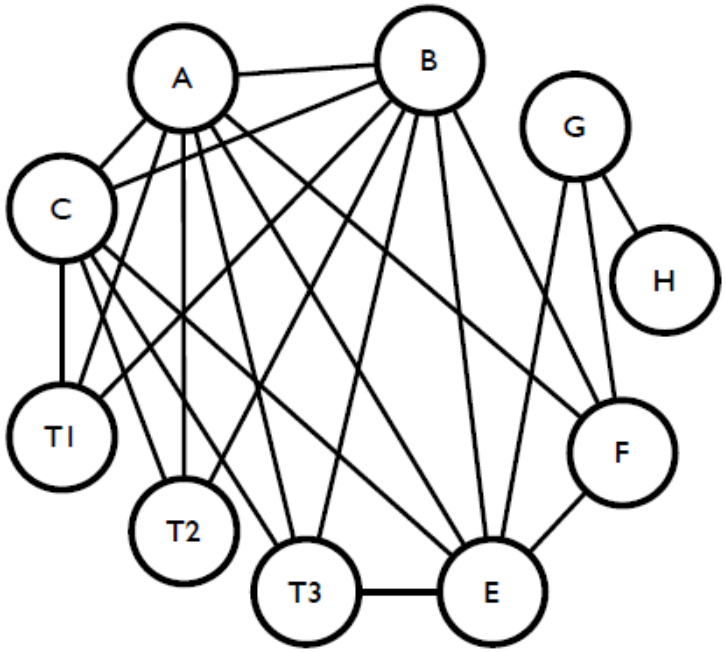


Simplify (step 1)

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}
```

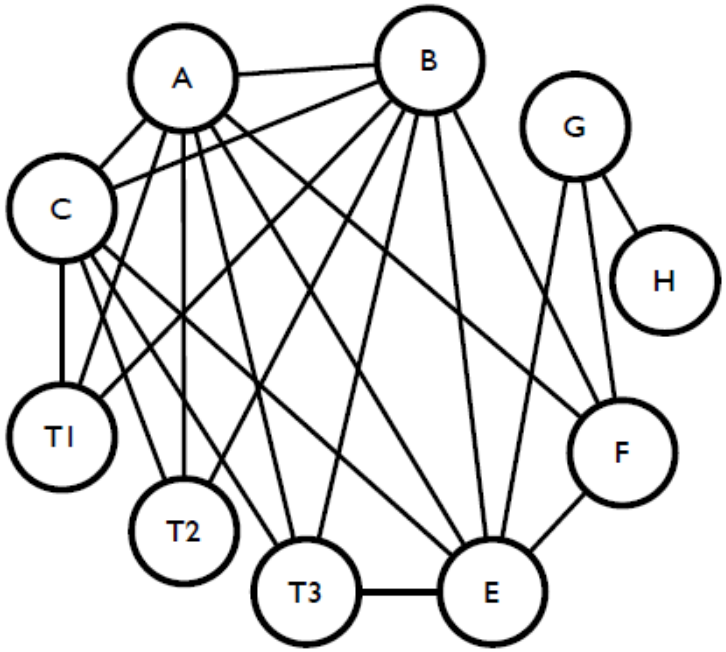


Color (step 2) Stack (left-bottom, right-top): H, G, E, F, C, T1, T2, T3, B, A

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}
```

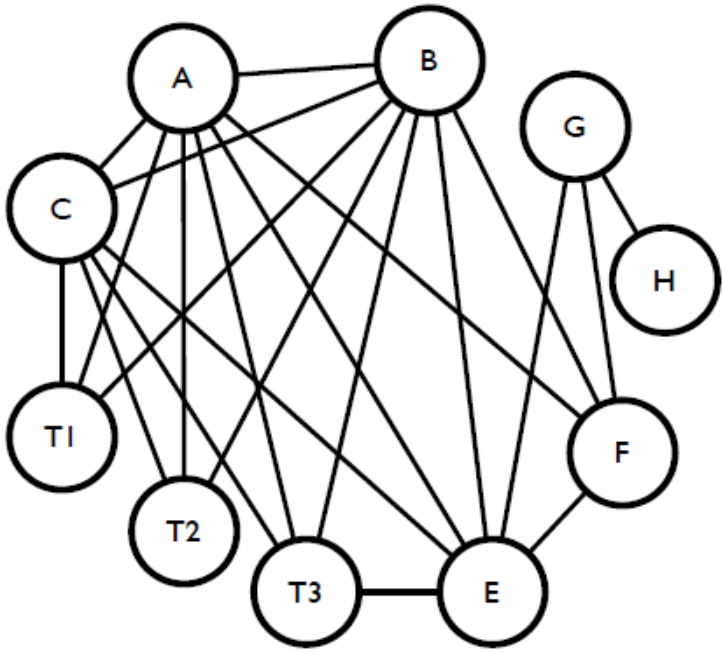


Color (step 2) Stack (left-bottom, right-top): H, G, E, F, C, T1, T2, T3, B, A

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