10. Register Allocation

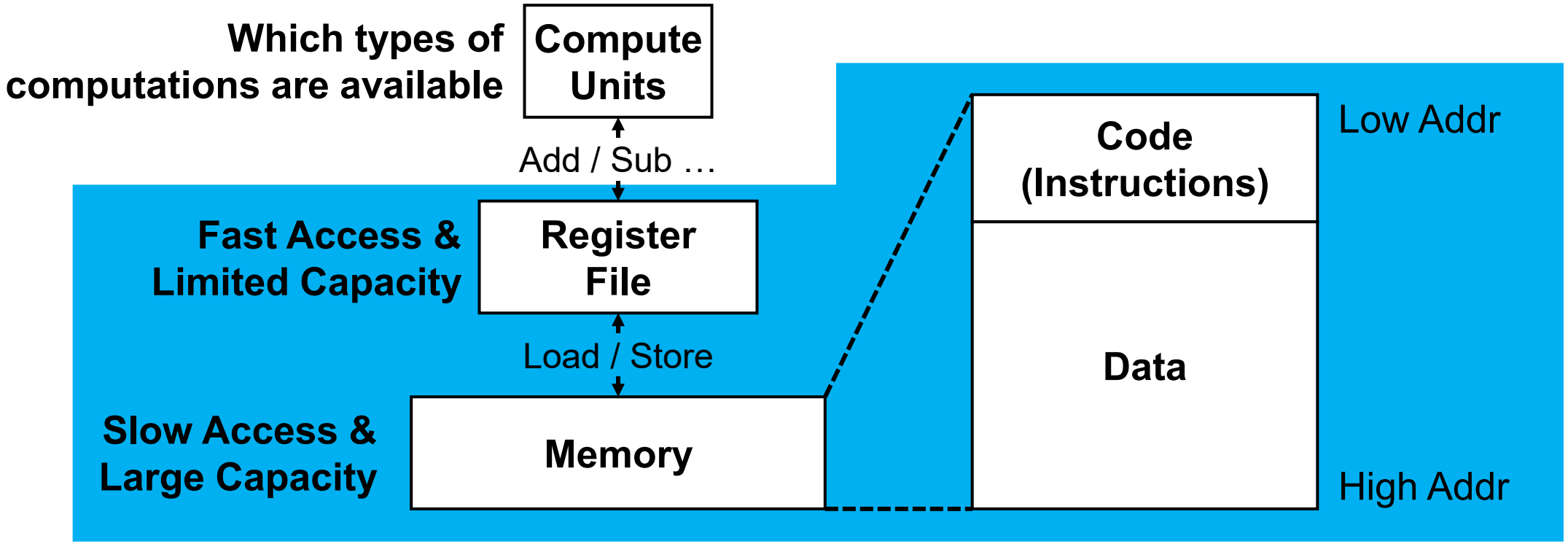
2025 Fall

Hunjun Lee

Hanyang University

ISA: What the Compiler Knows

- In a “software view”, the CPU consists of a compute units, register file, and memory



Storage Class Selection Problem

- **Determines where to place the data (register file vs. memory)**

- **Standard approach:**
  - Globals / Statics ➔ memory
  - Locals:
    - Composite types (structs, arrays, etc) ➔ memory
    - Rest ➔ Virtual register (this will be mapped in later lectures)

- **All memory approach:**
  - Put all variables into memory

Code Generation of IR

- **`igen (e, t)` (you've seen this in the stack machine)**
  - Code to compute the value of e in register t

- **We briefly discuss only a simple example**

```
igen(e1 + e2, t) =
      // t1 and t2 are fresh reg
      igen(e1, t1)
      igen(e2, t2)
      t := t1 + t2
```

- **It is simple (there are unlimited # of registers)**

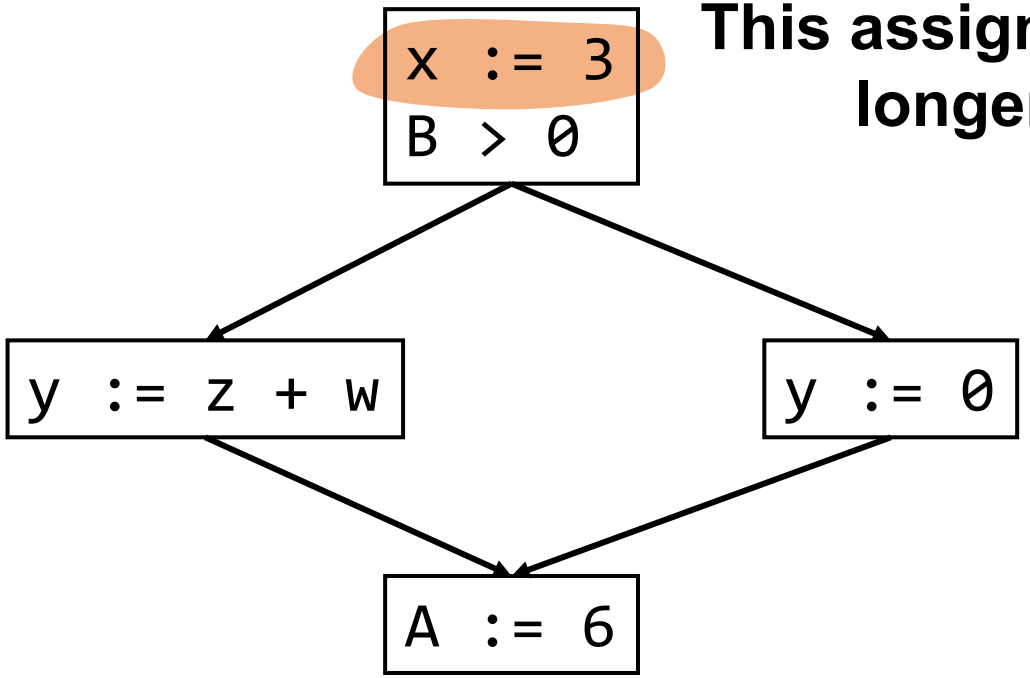
Register Allocation

- **We have assumed an IR with unlimited registers for now**
  - This simplifies the code generation and optimization
  - However, we need an additional process to translate the IR to the final assembly

- **We should rewrite the intermediate code to use no more temporaries than there are machine registers**
  - Ex) 32-bit x86 ISA has eight general-purpose registers
  - We should not change the program behavior after limiting the number of registers

Recall: Liveness Analysis - 1

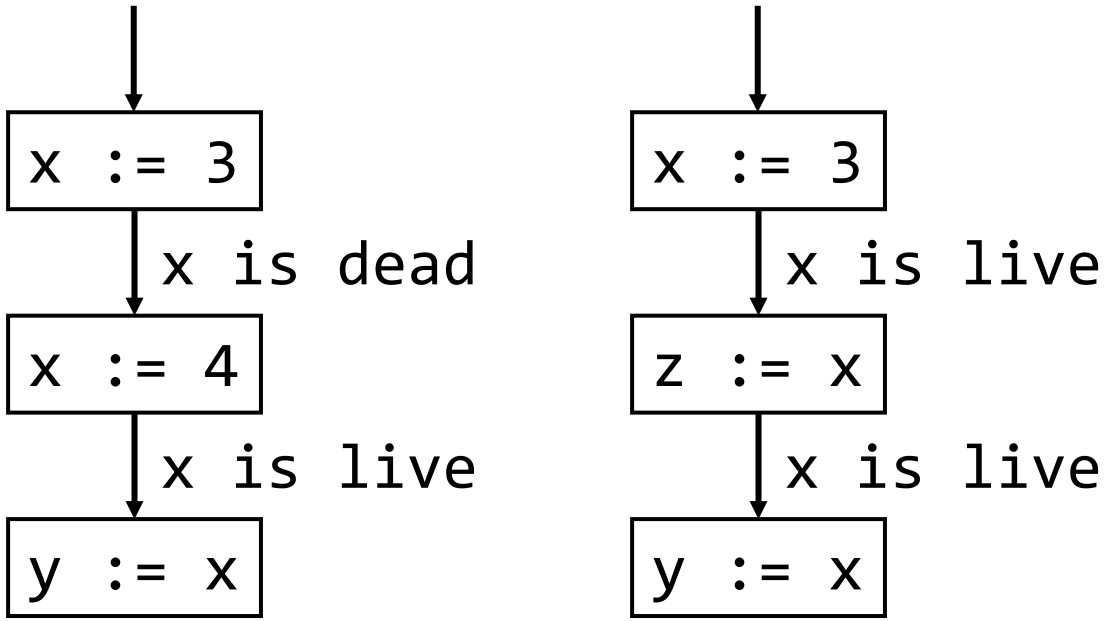
- **Liveness analysis is used to eliminate dead code**
  - liveness indicates that the assigned variable is used in the future
  - An assignment for x is dead if x is dead after the assignment



This assignment is no longer used

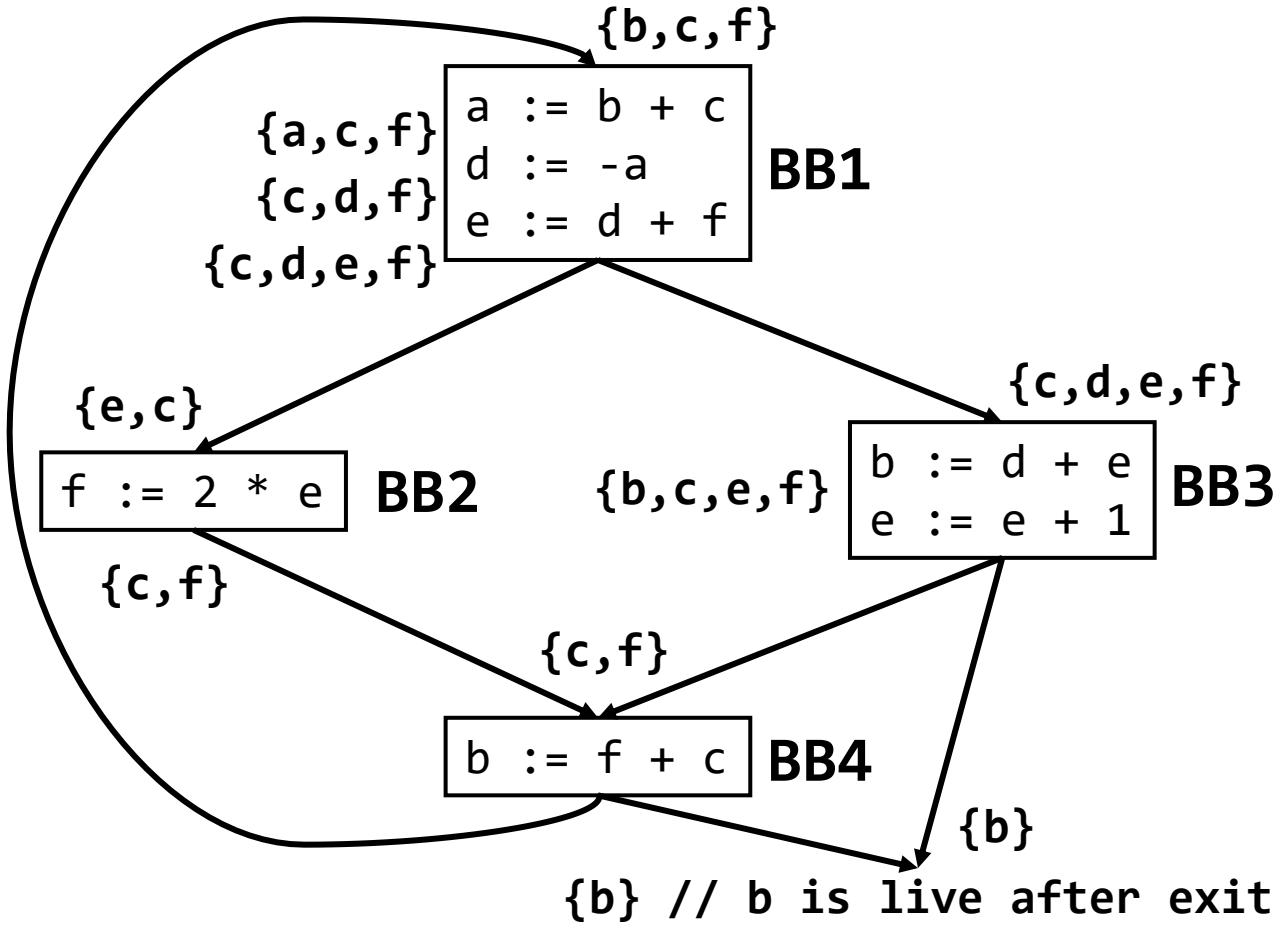
Recall: Liveness Analysis - 2

- **There are necessary functions in liveness analysis**
  - USE[b]: set of variables used in b
  - DEF[b]: set of variables defined in b
  - IN[b] / OUT[b]: set of live variables
- **A transfer function f_b for a basic block b:**
  - IN[b] = USE[b] U (OUT[b] – DEF[b])



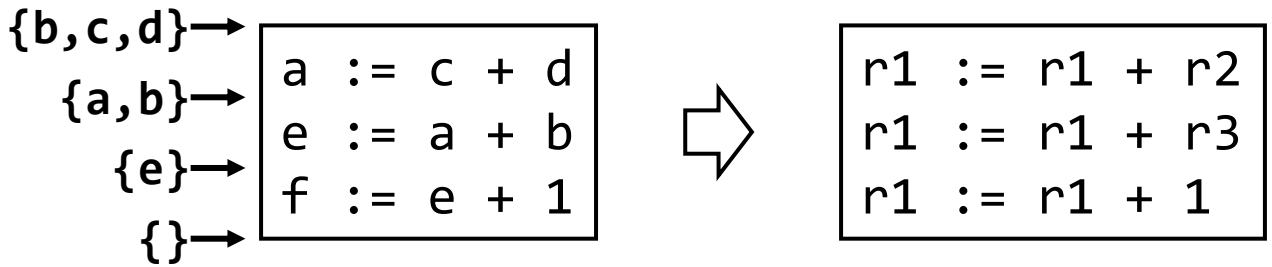
Recall: Liveness Analysis - 3

- Compute interference between registers using liveness



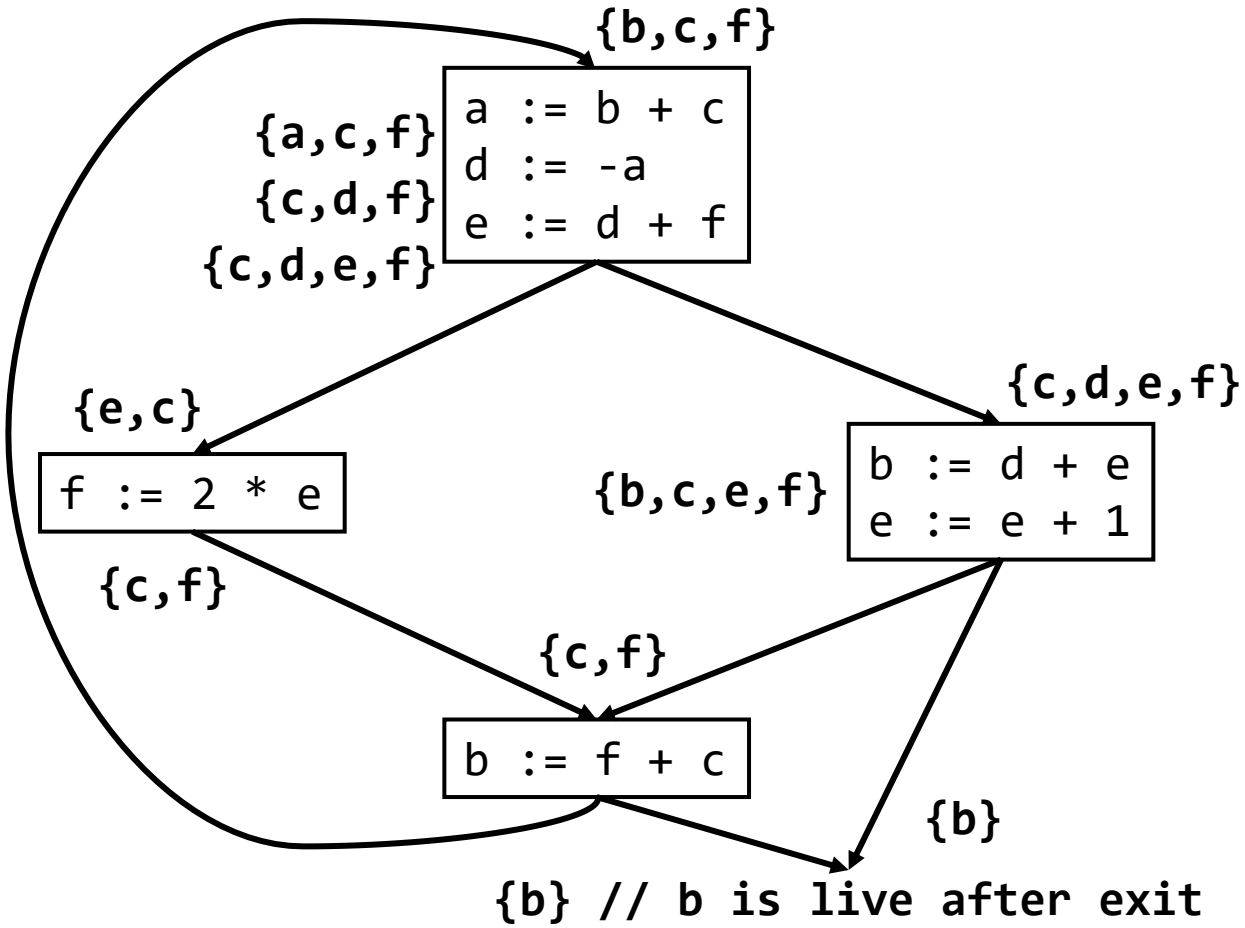
When to Reuse Registers?

- **A register for the dead temporary can be reused**
  - The temporaries t1 and t2 can share the register if at “any point in the program”, both t1 and t2 are not simultaneously live



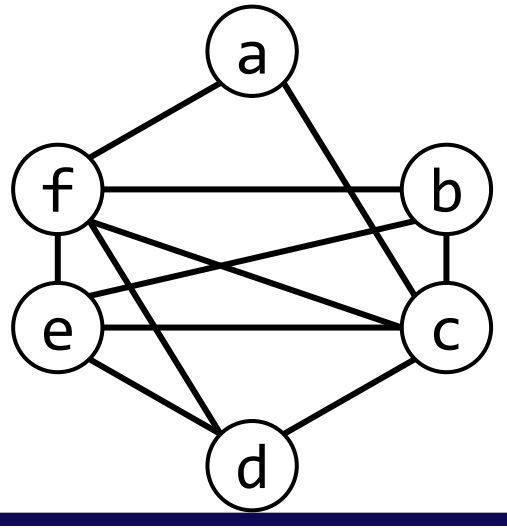
Register Interference Graph (RIG)

- **Compute interference between registers using liveness**



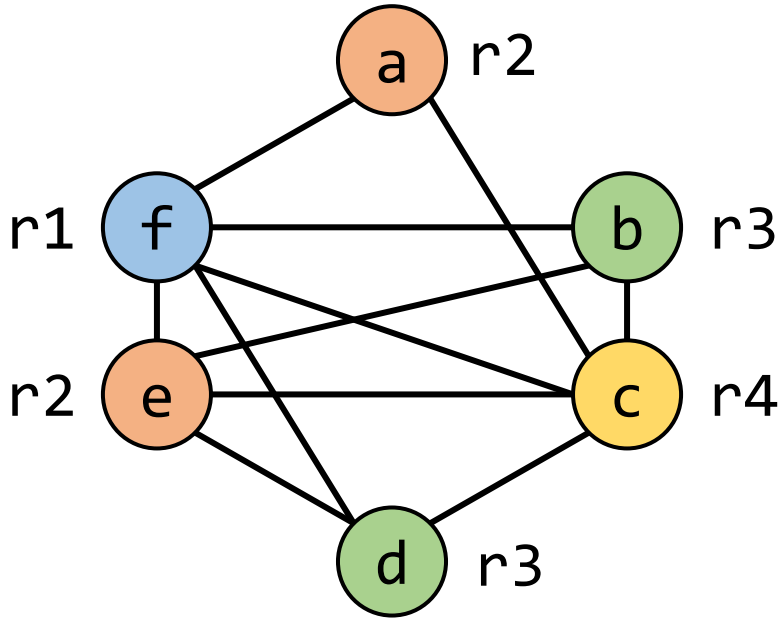
Construct an undirected graph with

- Each temporary as a node
- Connect temporaries if they are live simultaneously at a point



Graph Coloring - 1

- **Register allocation is the same as coloring RIG**
  - Color the nodes in a way that no connected nodes have the same color
  - If there are eight register files, we can assign at most eight colors



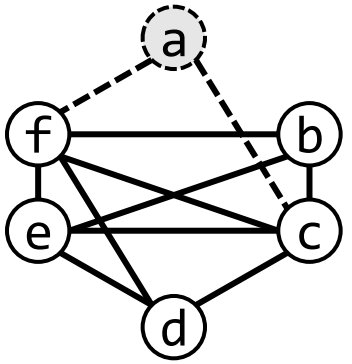
Graph Coloring - 2

- **Graph coloring is NP-hard, and no perfect algorithms are known ➔ we use heuristics**

- **A coloring may not exist for a given number of registers (we will address this later)**
  - We do not have enough registers in the target machine

Graph Coloring - 3

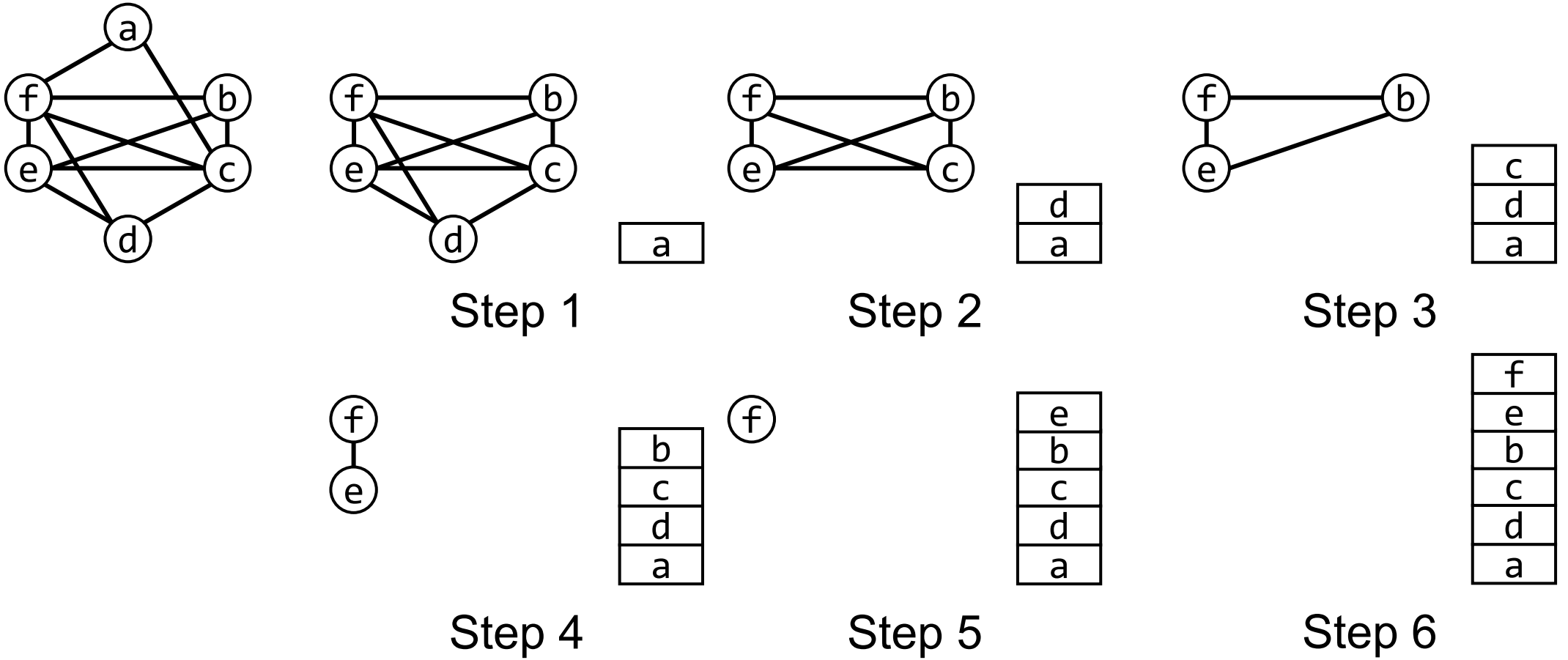
if k == 3,
we can always assign a color to “a”



- **Problem reduction**
  - Pick a node (n) with fewer than k neighbors (edges) in RIG
  - Eliminate the node (n) and its edges from RIG
  - If the resulting graph is k-colorable, so is the original graph
- **Using reduction heuristics**
  - Reduce the problem by removing a node n
  - Put n on a stack and remove it from the RIG
  - Repeat until the graph is empty (assume the graph always becomes empty)
- **Assign colors to the node by popping the stack**

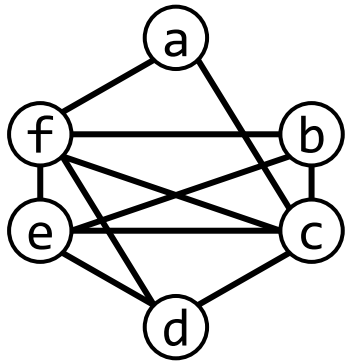
Graph Coloring Example - 1

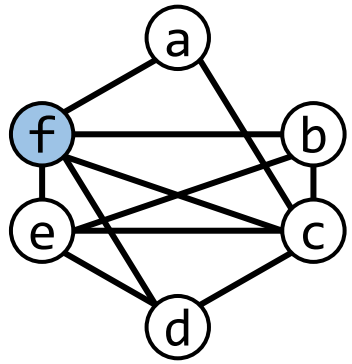
- **Given the RIG below, conduct a k=4 coloring problem**

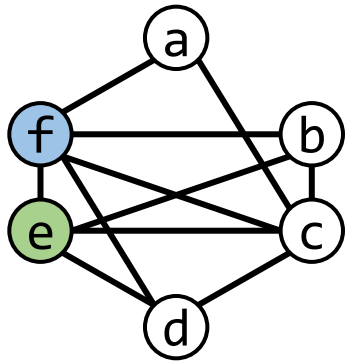


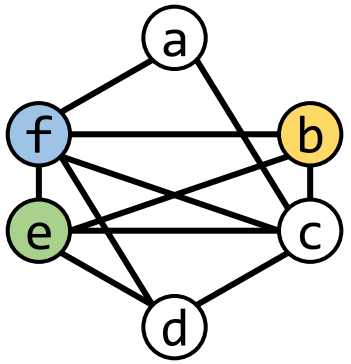
Graph Coloring Example - 2

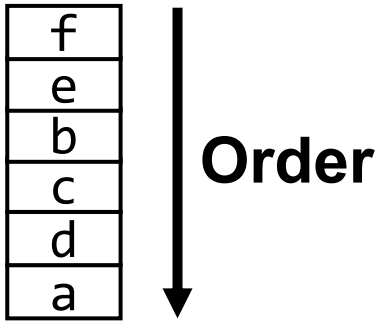
- **Given the RIG below, conduct a k=4 coloring problem**

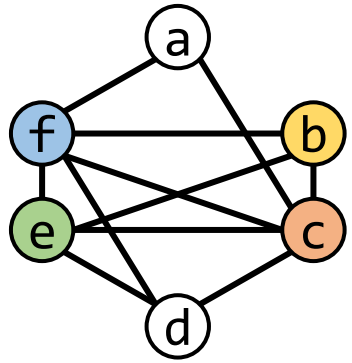


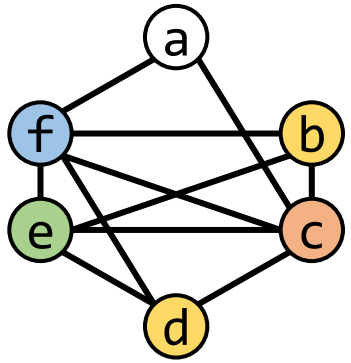


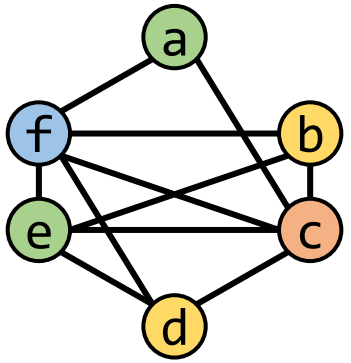










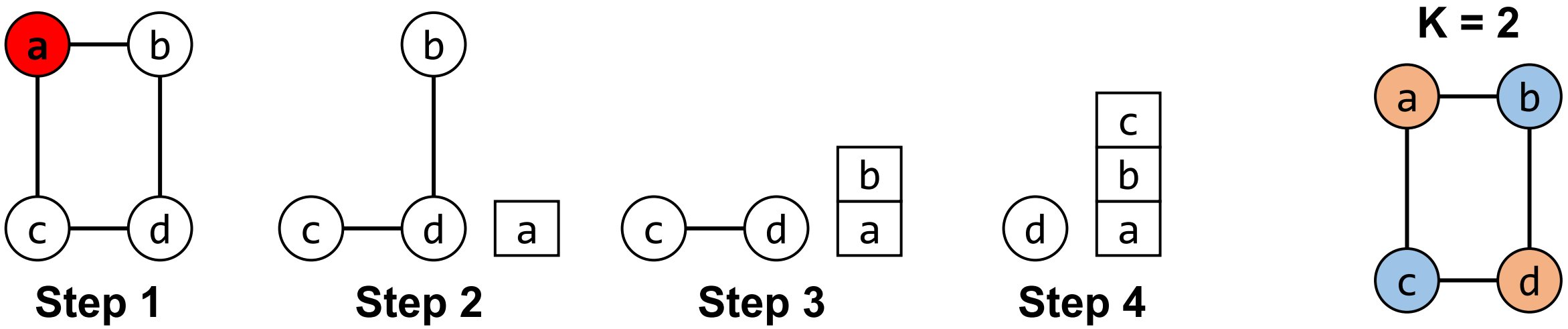


Optimistic Coloring

- If there is a failure, pick a node to spill and continue
- Then, we perform optimistic coloring, in case the heuristics was wrong (a solution may actually exist)

K = 2

Optimistic



Register Spilling

- **There are cases when the heuristic fails to find a coloring**
  - At some point, we fail to find a node n with fewer than k neighbors

- **In this case, we can't hold all values in registers**
  - Some values are spilled to memory

Register Spilling

- **If optimistic coloring fails, we spill f by**
  - allocating a memory location (`fa`) for `f` (typically in the current stack frame)
  - inserting load operation `f := load fa` before reading `f`
  - inserting store operation `store fa := f` after writing `f`
  - and changing variable f to a different one for each usage (e.g., f1, f2, …)

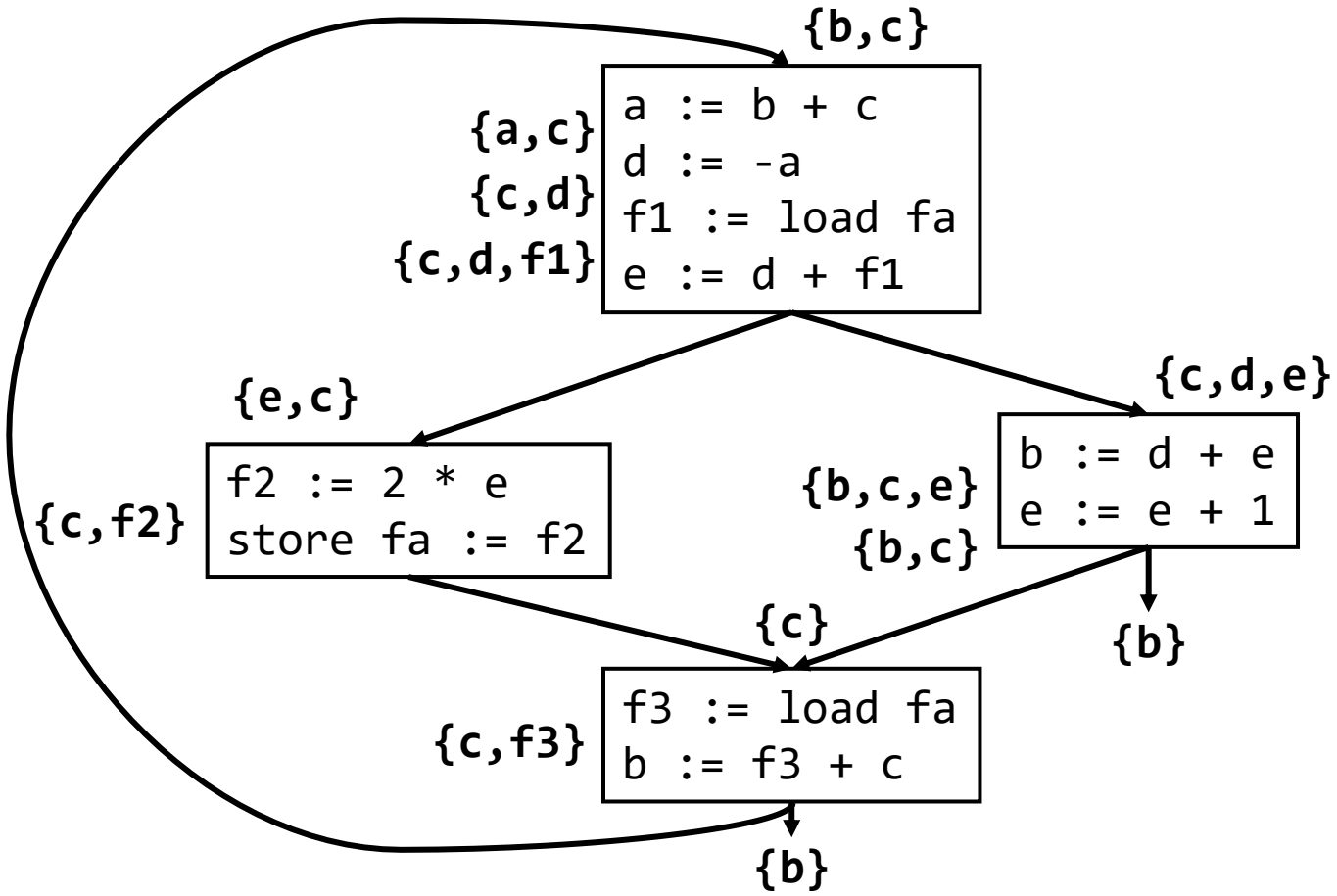
```
a := b + c
d := -a
f1 := load fa
e := d + f1
```

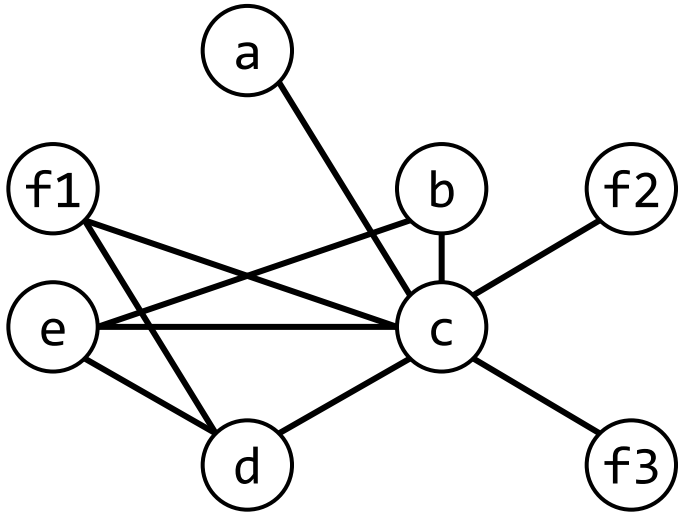
```
f2 := 2 * e
store fa := f2
```

```
f3 := load fa
b := f3 + c
```

Register Spilling

- **Now redraw RIG using the modified variable name**





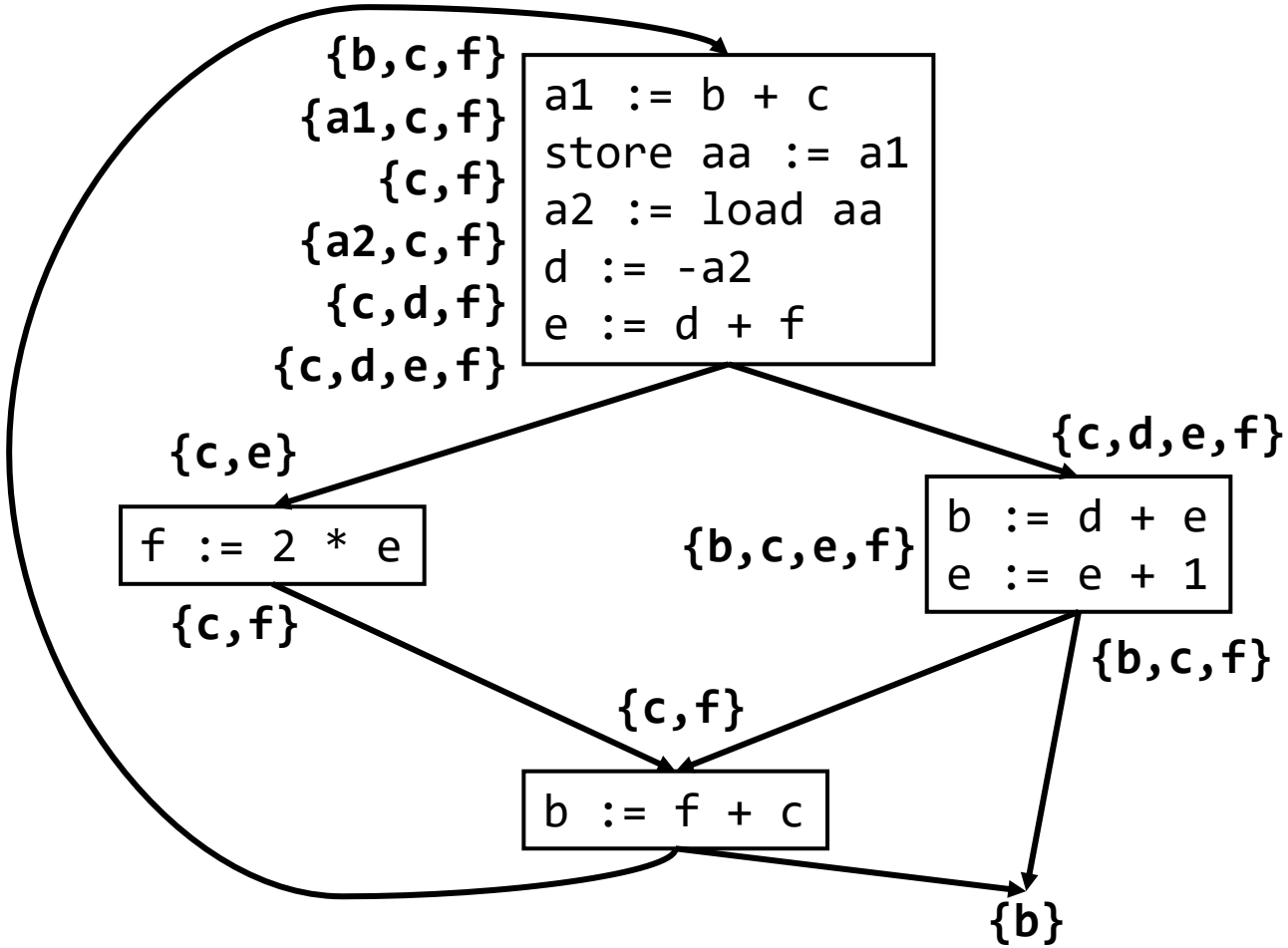
Register Spilling

- **Now the problem is, selecting which register to spill**
  - Any choice is correct, but some may incur (1) additional register spilling and (2) higher overhead due to the register spilling

- **There are some heuristics**
  - Spill temporaries with the most conflicts (the most # of edges)
  - Spill temporaries with few definitions and uses
  - Avoid spilling temporaries in inner loops (incurs large # of loads and stores)

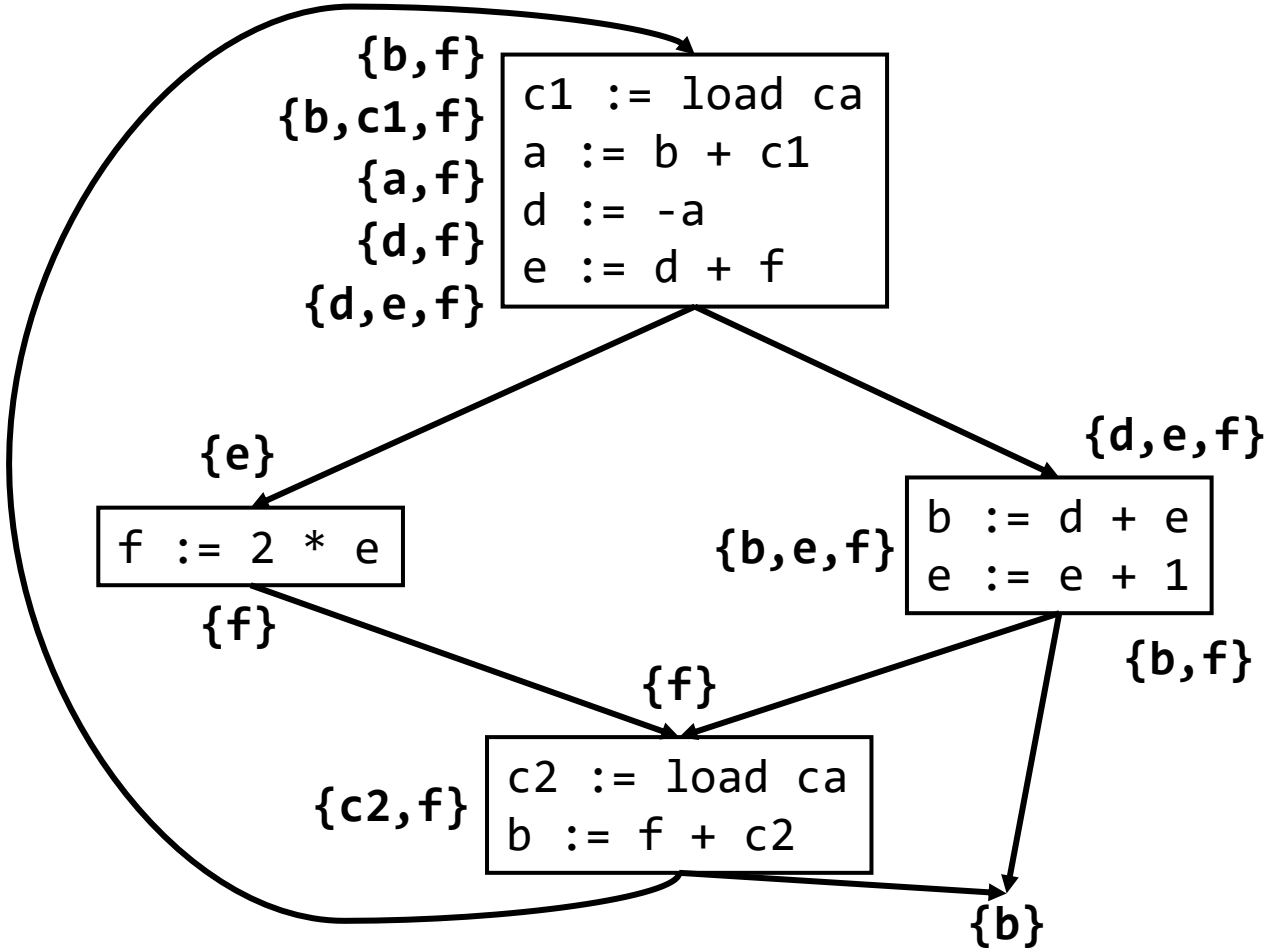
Register Spilling

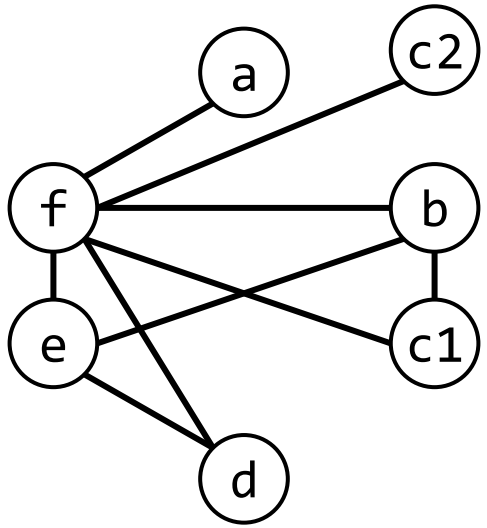
- **What happens if we spill a?**



Register Spilling

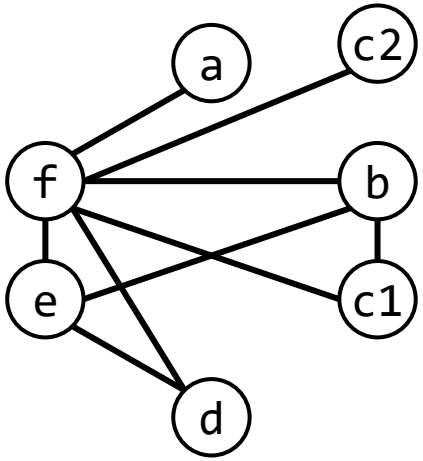
- **What happens if we spill c?**

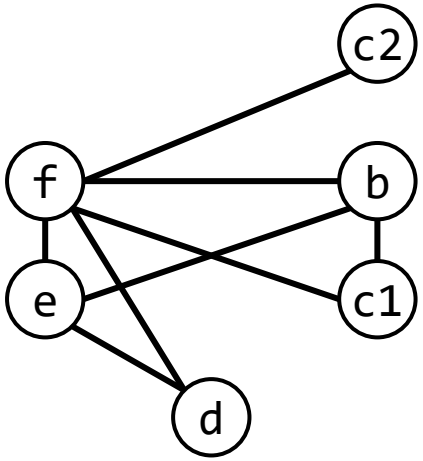


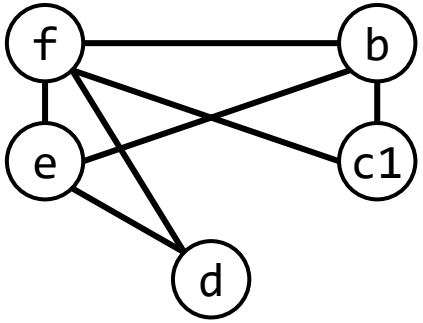


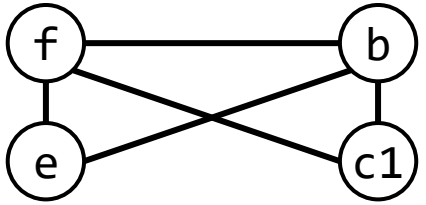
Register Spilling

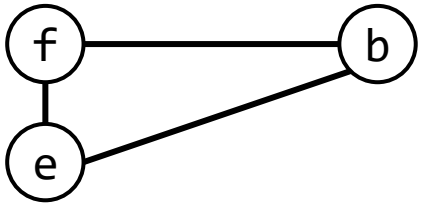
- **Assume k = 3**

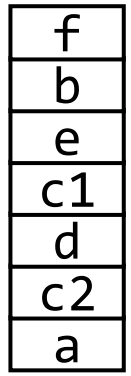


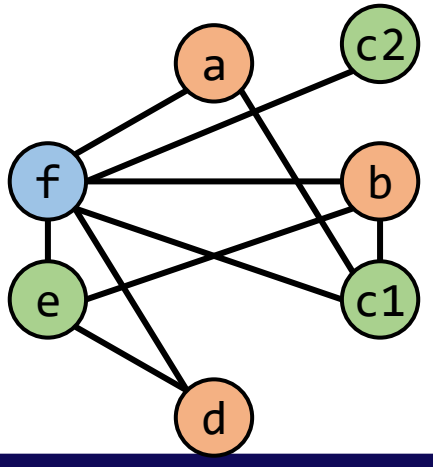






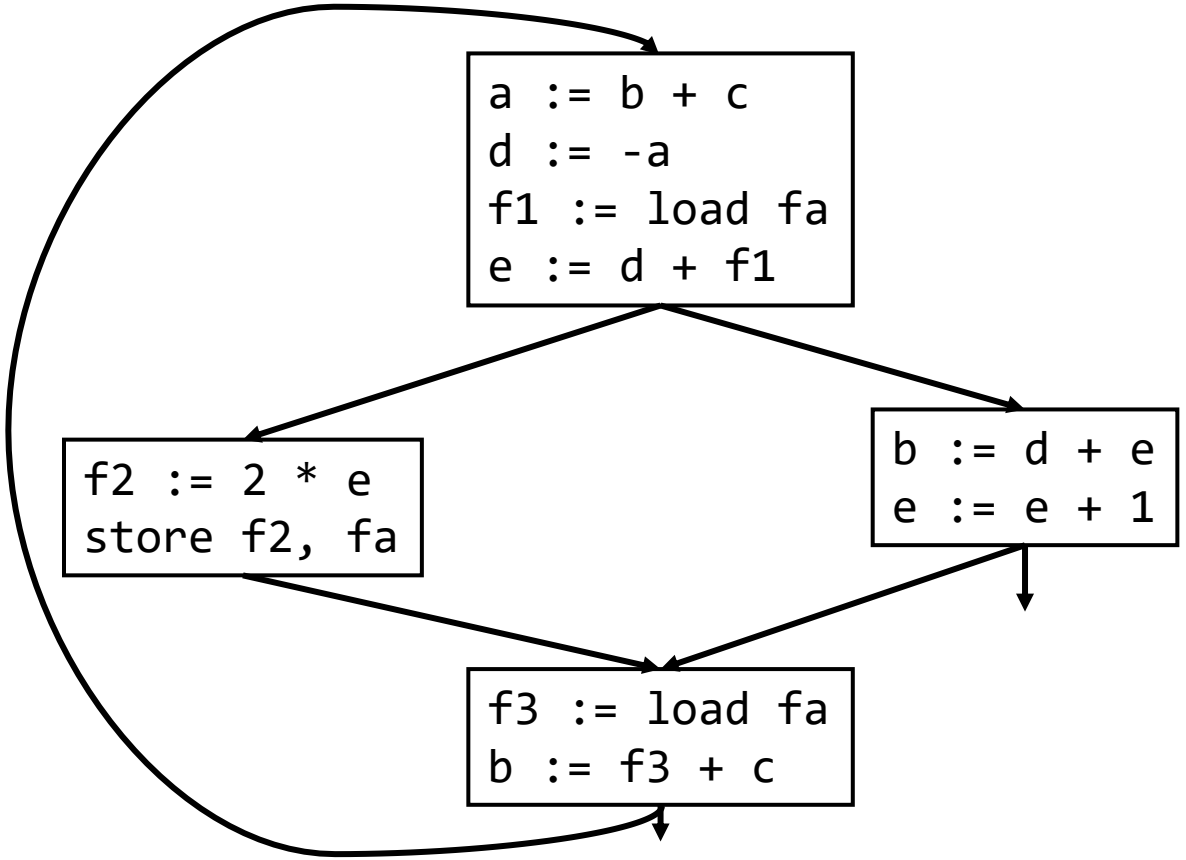


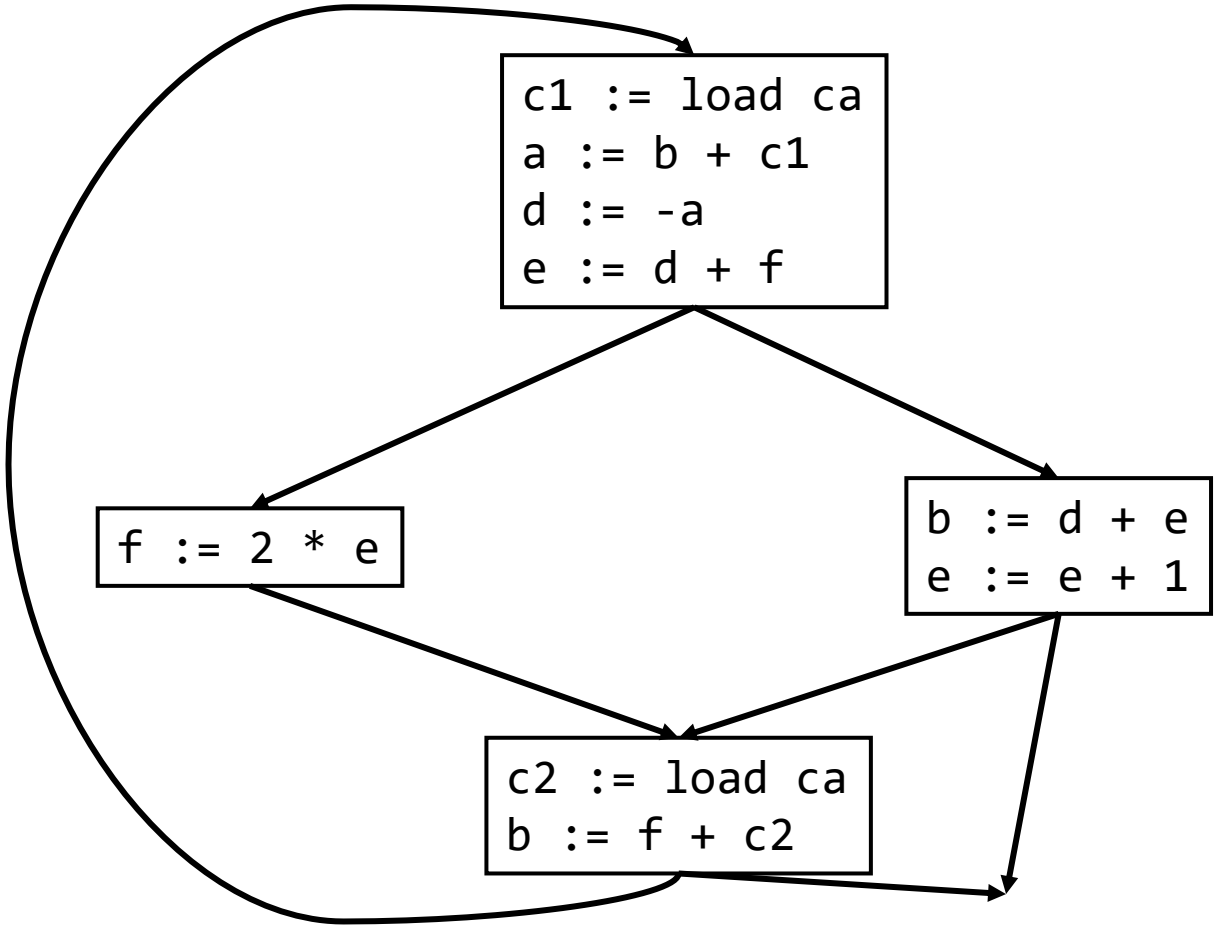




Register Spilling

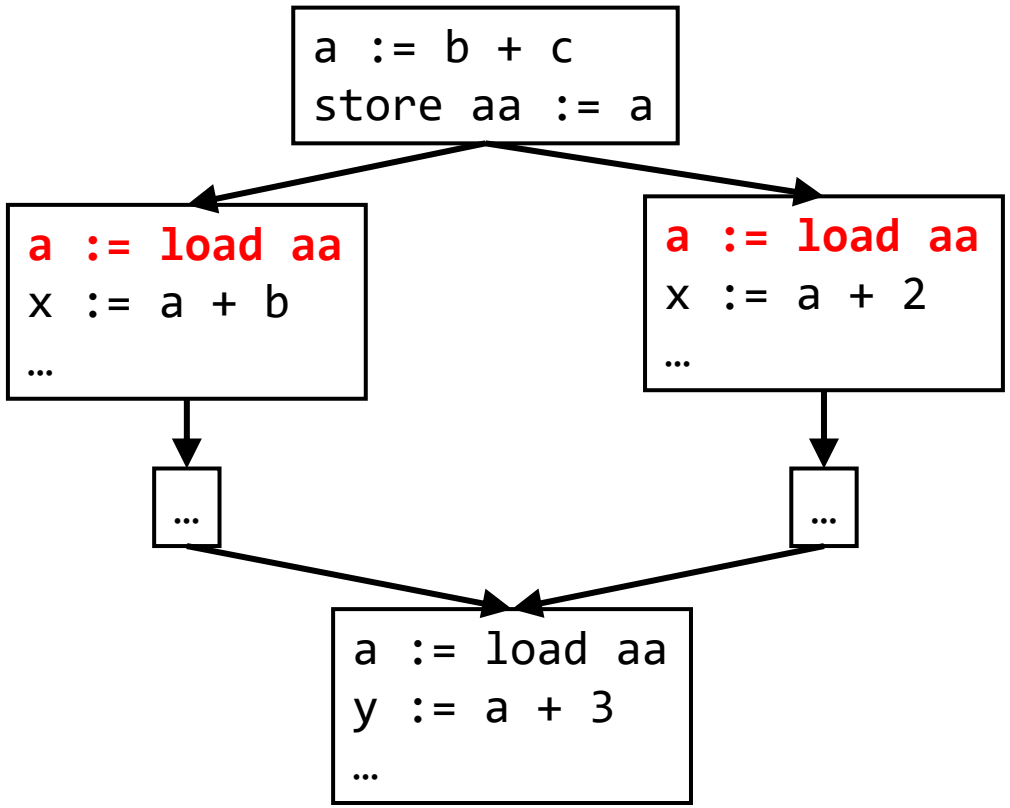
- **Which one is better?**

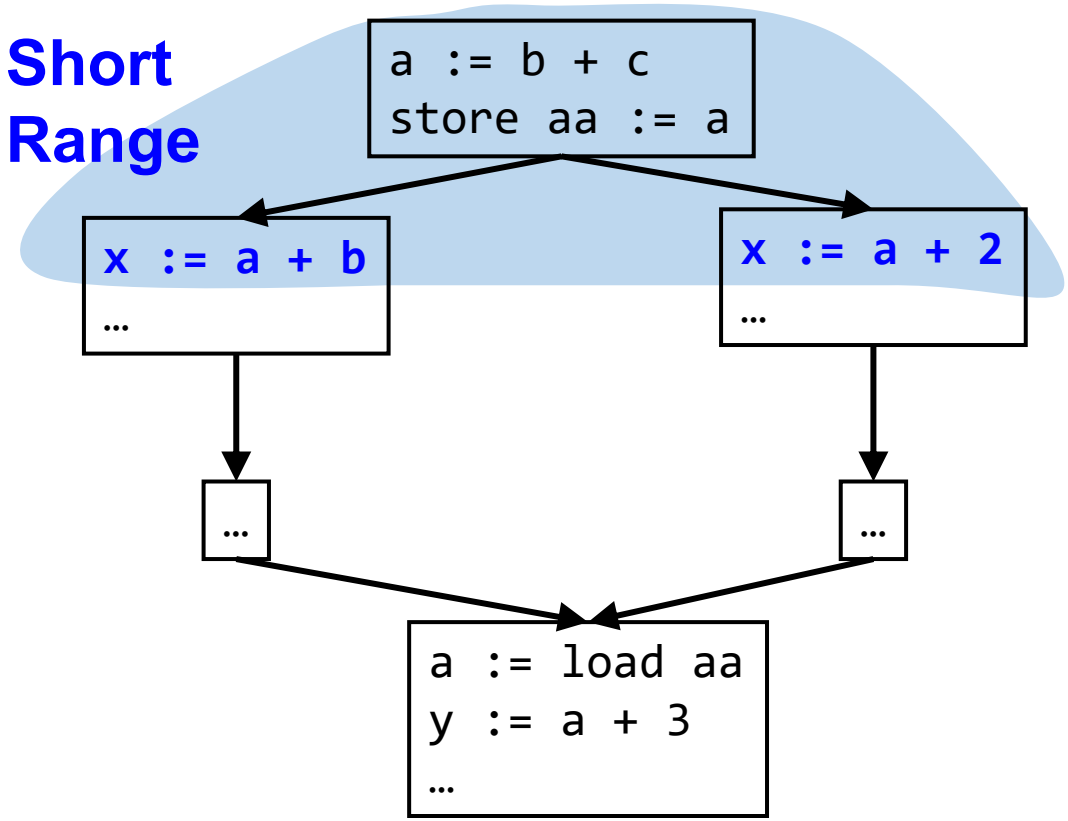




Reducing Spilling Cost - 1

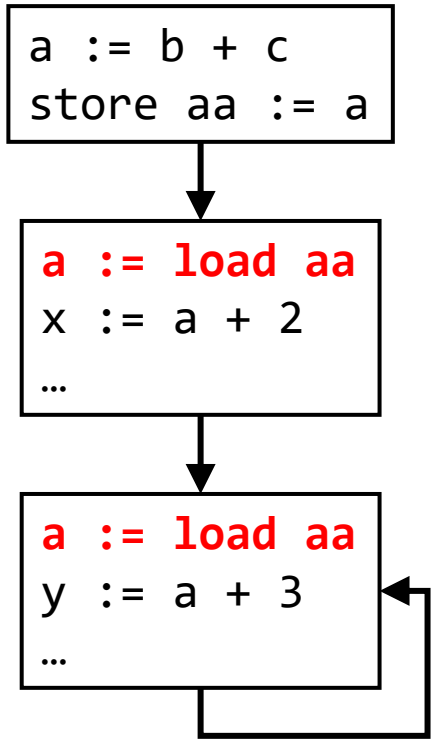
- **Split live range**
  - Instead of choosing variables to spill, choose live ranges to split

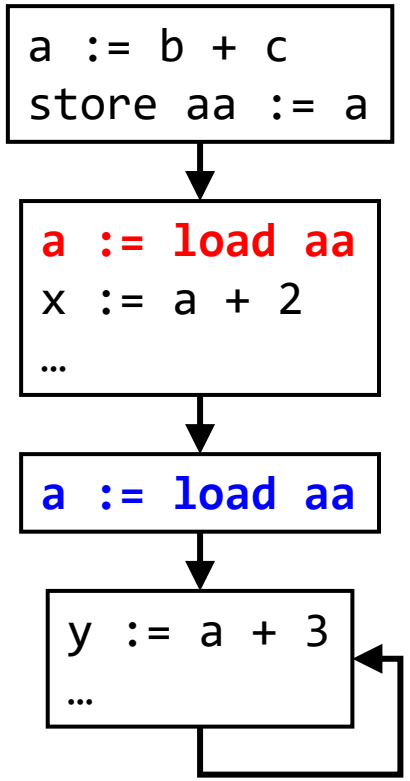




Reducing Spilling Cost - 2

- **Split live range**
  - Instead of choosing variables to spill, choose live ranges to split





Register Keyword

- **There is a "register" keyword in C**
  - We can enforce the compiler to keep a variable in the register file (without spilling the data to the stack)

```
#include <stdio.h>
int main()
{
      register int i = 10;
      int* a = &i;
      printf("%d", *a);
      getchar();
      return 0;
}
```

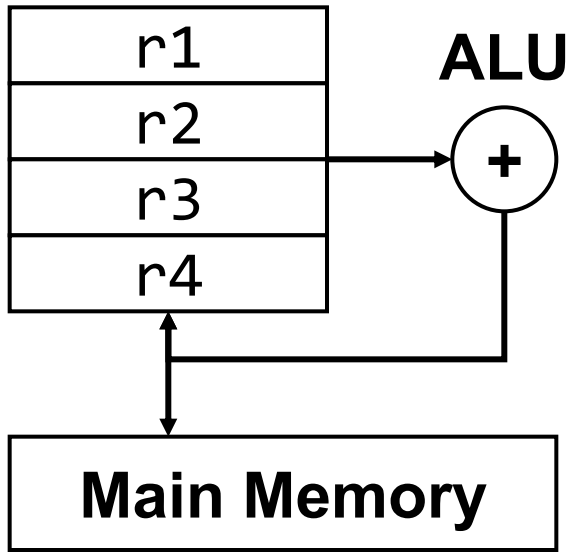
Registers in Computer Architecture - 1

- You can simply imagine having a physical register with four entries ➔ Is this correct?

4-Register Code

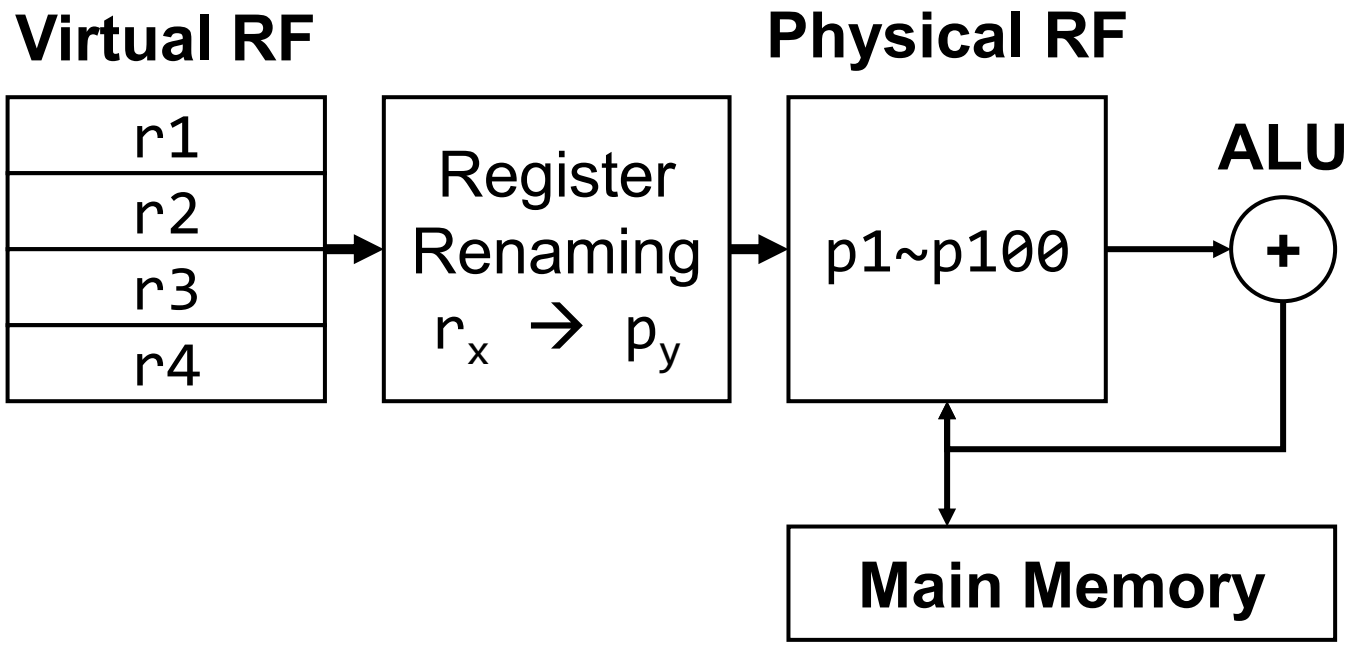
```
r1 := r1 + 1
r2 := r1 + 2
r3 := 4
r1 := r3 + r2
r4 := r3 + 4
…
```

Register File



Registers in Computer Architecture - 2

- There are actually 100+ physical registers in modern CPU while the ISA exposes only eight 64-bit general-purpose registers

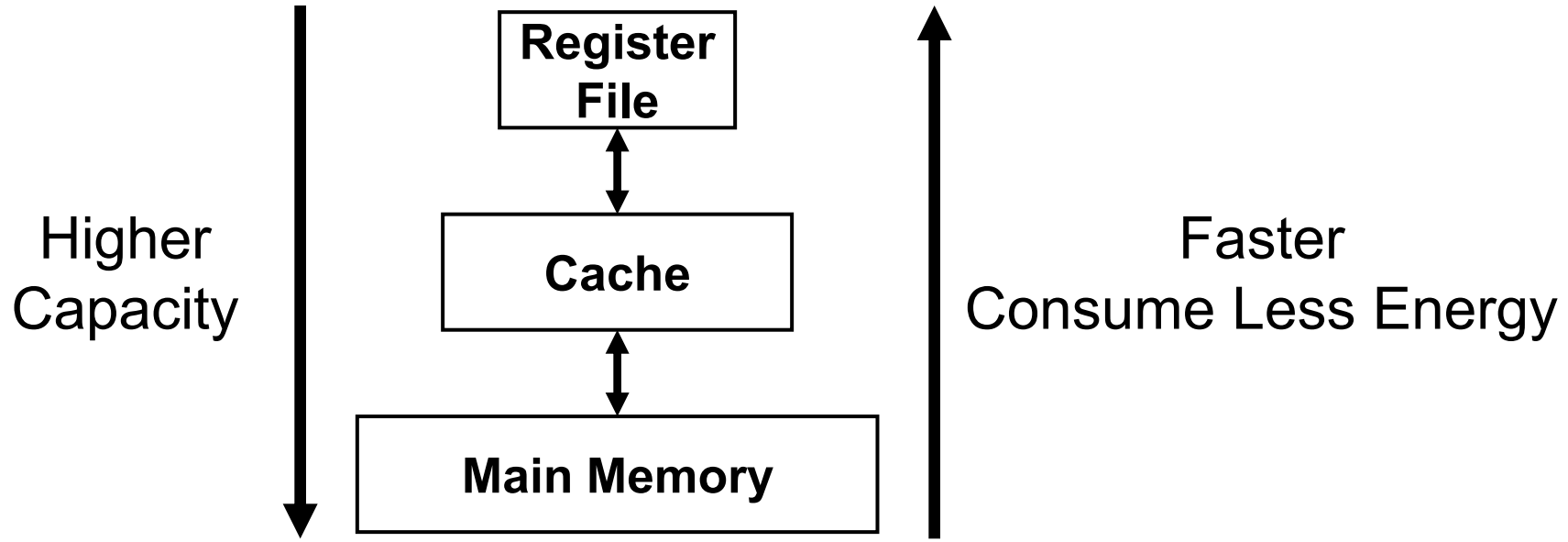


Why do we expose only a small # of registers?

- Backward compatibility
- Reduced compiler complexity
- Reduced instruction size / decoding overhead
- Context switching overhead
- On and on …

Microarchitecture-Aware Compiler - 1

- **CPU memory system utilizes a cache between the register file and main memory**
  - These microarchitecture details are not exposed to the compiler (but some compilers do use the details)



Microarchitecture-Aware Compiler - 2

- **To maximally use the cache, we should minimize the working set**

Unoptimized code

```
for (j = 0; j < 10; j++)
      for (i = 0; i < 10000000; i++)
            a[i] *= b[i];
```

Optimized code

```
for (i = 0; i < 10000000; i++)
      for (j = 0; j < 10; j++)
            a[i] *= b[i];
```