CS142B Language Processor Construction

Register Allocation

Yeoul Na

UCI

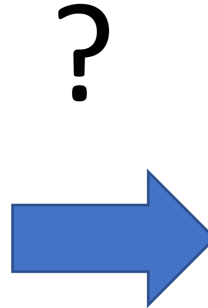
May 14, 2019

Intermediate Representation

```
BB0: (Entry)

BB4:
%3 = PHI (BB0,%1),(BB9, %7)
%4 = PHI (BB0,%2),(BB9, %6)
%5 = ICMP %3, #5
BR_GE %5, BB19, BB9

BB9:
%6 = IADD %4, %3
%7 = IINC %3
BR BB4

BB19:
CALL printInt(%4)
RETURN
```

IR -> Machine Code

```
BB0: (Entry)

BB4:
%3 = PHI (BB0,%1),(BB9, %7)
%4 = PHI (BB0,%2),(BB9, %6)
%5 = ICMP %3, #5
BR_GE %5, BB19, BB9

BB9:
%6 = IADD %4, %3
%7 = IINC %3
BR BB4

BB19:
CALL printInt(%4)
RETURN
```

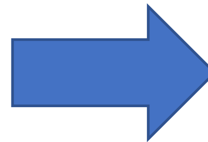



```
BB0: (Entry)
MOV R1, #0
MOV R2, #0

BB4:
CMP R1, #5
BR_GE BB19

BB9:
ADD R2, R1
INC R1
JMP BB4

BB19:
PUSH R2
CALL “printInt”
RETURN
```

IR -> Machine Code

```
BB0: (Entry)

BB4:
%3 = PHI (BB0,%1),(BB9, %7)
%4 = PHI (BB0,%2),(BB9, %6)
%5 = ICMP %3, #5
BR_GE %5, BB19, BB9

BB9:
%6 = IADD %4, %3
%7 = IINC %3
BR BB4

BB19:
CALL printInt(%4)
RETURN
```

Instruction Selection

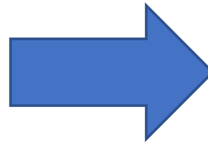


```
BB0: (Entry)
MOV R1, #0
MOV R2, #0

BB4:
CMP R1, #5
BR_GE BB19

BB9:
ADD R2, R1
INC R1
JMP BB4

BB19:
PUSH R2
CALL “printInt”
RETURN
```

IR -> Machine Code

```
BB0: (Entry)

BB4:
%3 = PHI (BB0,%1),(BB9, %7)
%4 = PHI (BB0,%2),(BB9, %6)
%5 = ICMP %3, #5
BR_GE %5, BB19, BB9

BB9:
%6 = IADD %4, %3
%7 = IINC %3
BR BB4

BB19:
CALL printInt(%4)
RETURN
```

Register Allocation



```
BB0: (Entry)
MOV R1, #0
MOV R2, #0

BB4:
CMP R1, #5
BR_GE BB19

BB9:
ADD R2, R1
INC R1
JMP BB4

BB19:
PUSH R2
CALL "printInt"
RETURN
```

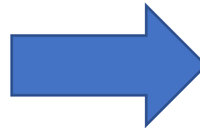
Steps for IR -> Machine Code

- Instruction Selection
- Register Allocation

Register Allocation

- Mapping an infinite number of virtual registers to a finite number of physical registers

```
BB0: (Entry)


BB4:
%3 = PHI (BB0,%1),(BB9, %7)
%4 = PHI (BB0,%2),(BB9, %6)
%5 = ICMP %3, #5
BR_GE %5, BB19, BB9


BB9:
%6 = IADD %4, %3
%7 = IINC %3
BR BB4


BB19:
CALL printInt(%4)
RETURN
```



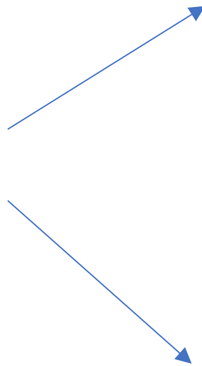
```
BB0: (Entry)
MOV R1, #0
MOV R2, #0

BB4:
CMP R1, #5
BR_GE BB19


BB9:
ADD R2, R1
INC R1
JMP BB4


BB19:
PUSH R2
CALL “printInt”
RETURN
```

Basic Idea of Register Allocation

- # Virtual Registers >> # Physical Registers
  - Some virtual registers must share the same physical register
  - What if there are conflicts?

Basic Example

```
i0: A = 4;
i1: B = 10;
i2: C = A + B;
i3: D = C * 3;
```

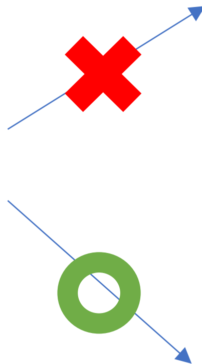


```
i0: R1 = 4;
i1: R1 = 10;
i2: C = R1 + R1;
i3: D = C * 3;
```

```
i0: R1 = 4;
i1: B = 10;
i2: C = R1 + B;
i3: R1 = C * 3;
```

Basic Example

```
i0: A = 4;
i1: B = 10;
i2: C = A + B;
i3: D = C * 3;
```



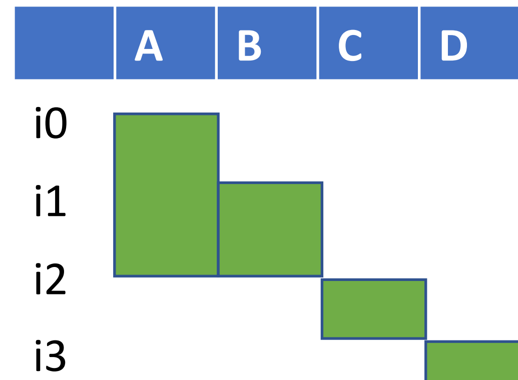
```
i0: R1 = 4;
i1: R1 = 10;
i2: C = R1 + R1;
i3: D = C * 3;
```

```
i0: R1 = 4;
i1: B = 10;
i2: C = R1 + B;
i3: R1 = C * 3;
```

Live Intervals

- A SSA value is live from its definition to its last use

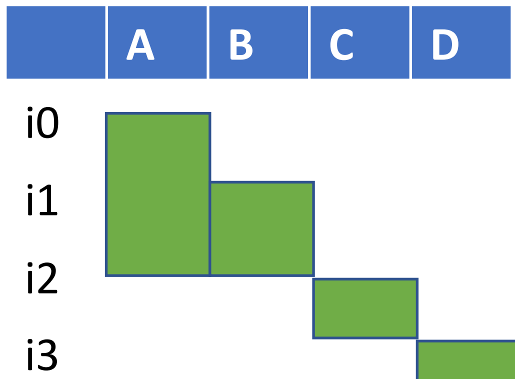
```
i0: A = 4;
i1: B = 10;
i2: C = A + B;
i3: D = C * 3;
```

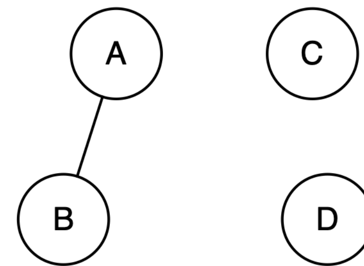


Interference Graph

- Variables that live at the same time are connected in the graph

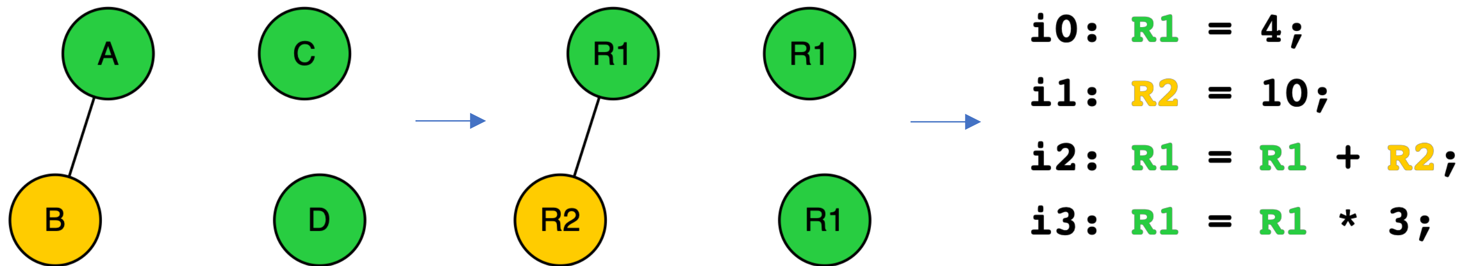
```
i0: A = 4;
i1: B = 10;
i2: C = A + B;
i3: D = C * 3;
```





K-Coloring Problem

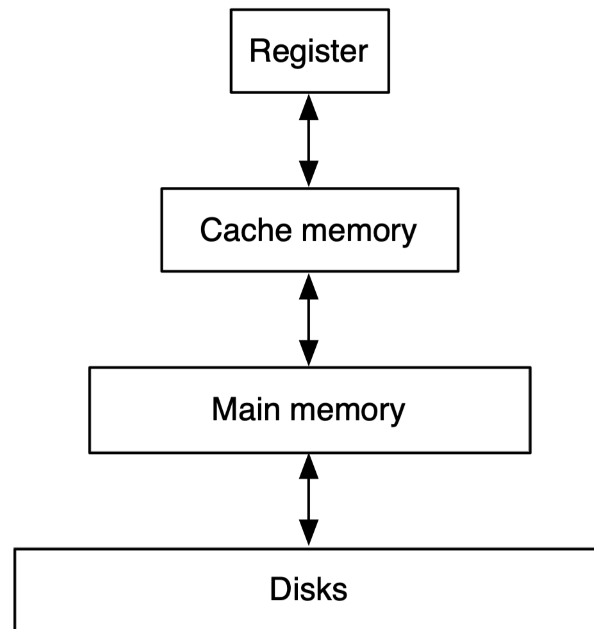
- Color the nodes in the graph with at most k colors
- Neighbors should be assigned different colors



- Optimal solution : NP-Complete -> Use heuristics
- What if it cannot be colored with k colors? spill

Memory hierarchy

- Higher = smaller, faster, closer to CPU
- Avoid register spilling! Store variables in registers as possible



Register Allocation in a Nutshell

- Compute live intervals
- Build an interference graph
- Solve K-coloring problem with K registers
- Output: Variables with colors (assigned physical registers)

Liveness Analysis

- A type of data-flow analysis
  - Data flow facts: Live-in / Live-out
    - *Use[s]:* the set of variables used in s
    - *Def[s]:* the set of variables that are assigned values in s
    - *Live-in[s]:* the set of variables that are simultaneously live *before* s is executed
    - *Live-out[s]:* the set of variables that are simultaneously live *after* s is executed
  - Backward data-flow analysis
  - Transfer functions
    - Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])
    - Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

Liveness Analysis Equations

- Initial Setup
  - Live-out[final] = ∅
  - Use[a:= b + c] = {b, c}
  - Def[a:= b + c] = {a}
- Transfer functions
  - Lives-in[s] = Use[s] ∪ (Lives-out[s] – Def[s])
  - Lives-out[s] = $\bigcup_{p \in succ(s)}$ Live-in[p]

Liveness Analysis Example

```
L1: a = 3;
L2: b = 5;
L3: d = 4;
L4: x = 100;
L5:   if a > b then
L6:       c = a + b;
L7:       d = 2;
L8:   endif
L9: c = 4;
L10:return b * d + c;
```

Live-out = {b, d, c}   Live-in = {b, d}

Live-out = {}   Live-in = {b, d, c}

Lives-in[s] = Use[s] ∪ (Lives-out[s] – Def[s])

Lives-out[s] = $\bigcup_{p \in succ(s)}$ Live-in[p]

Worklist = {L8}

Liveness Analysis Example

```
L1: a = 3;
L2: b = 5;
L3: d = 4;
L4: x = 100;
L5:   if a > b then
L6:        c = a + b;
L7:        d = 2;
L8:   endif
L9: c = 4;
L10:return b * d + c;
```

Live-out = {b, d}   Live-in = {b, d}

Live-out = {b, d, c}   Live-in = {b, d}

Live-out = {}   Live-in = {b, d, c}

Lives-in[s] = Use[s] ∪ (Lives-out[s] – Def[s])

Lives-out[s] = $\bigcup_{p \in succ(s)}$ Live-in[p]

Worklist = {L7, L5}

Liveness Analysis Example

L1: a = 3;
L2: b = 5;
L3: d = 4;
L4: x = 100;
L5: if a > b then
L6: c = a + b; — Live-out = {b} Live-in = {b, a}
L7: d = 2; — Live-out = {b, d} Live-in = {b}
L8: endif — Live-out = {b, d} Live-in = {b, d}
L9: c = 4; — Live-out = {b, d, c} Live-in = {b, d}
L10:return b * d + c; — Live-out = {} Live-in = {b, d, c}

Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])
Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

Worklist = {L5}

Liveness Analysis Example

Code	Live-out	Live-in
L1: a = 3;		
L2: b = 5;		
L3: d = 4;		
L4: x = 100;		
L5: if a > b then	Live-out = {b, a, d}	Live-in = {b, a, d}
L6: c = a + b;	Live-out = {b}	Live-in = {b, a}
L7: d = 2;	Live-out = {b, d}	Live-in = {b}
L8: endif	Live-out = {b, d}	Live-in = {b, d}
L9: c = 4;	Live-out = {b, d, c}	Live-in = {b, d}
L10:return b * d + c;	Live-out = {}	Live-in = {b, d, c}

Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])

Lives-out[s] = $\bigcup_{p \in succ(s)}$ Live-in[p]

Worklist = {L4}

Liveness Analysis Example

```
L1: a = 3;
L2: b = 5;
L3: d = 4;
L4: x = 100;
L5:   if a > b then
L6:        c = a + b;
L7:        d = 2;
L8:   endif
L9: c = 4;
L10:return b * d + c;
```

Live-out = {b, a, d}   Live-in = {b, a, d}

Live-out = {b, a, d}   Live-in = {b, a, d}

Live-out = {b}   Live-in = {b, a}

Live-out = {b, d}   Live-in = {b}

Live-out = {b, d}   Live-in = {b, d}

Live-out = {b, d, c}   Live-in = {b, d}

Live-out = {}   Live-in = {b, d, c}

Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])

Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

Worklist = {L3}

Liveness Analysis Example

Code	Live-out	Live-in
L1: a = 3;	Live-out = {a}	Live-in = { }
L2: b = 5;	Live-out = {b, a}	Live-in = {a}
L3: d = 4;	Live-out = {b, a, d}	Live-in = {b, a}
L4: x = 100;	Live-out = {b, a, d}	Live-in = {b, a, d}
L5: if a > b then	Live-out = {b, a, d}	Live-in = {b, a, d}
L6: c = a + b;	Live-out = {b}	Live-in = {b, a}
L7: d = 2;	Live-out = {b, d}	Live-in = {b}
L8: endif	Live-out = {b, d}	Live-in = {b, d}
L9: c = 4;	Live-out = {b, d, c}	Live-in = {b, d}
L10:return b * d + c;	Live-out = {}	Live-in = {b, d, c}

Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])

Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

Worklist = {}

Liveness Analysis on SSA

- Transfer functions
  - Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])
  - Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

- Special handling of Phi functions
  - Do not add Phi operands to live sets
  - Lives-out[b] = $\cup_{p \in succ(b)}$ {Phi(p).inputOf(b), Lives-in[p]}

Liveness Analysis on SSA Example

Code	Liveness
BB0: (Entry)	
L0: %1 = MOV #0	Live-out = {%1} Live-in = {}
L1: %2 = MOV #0	Live-out = {%1,%2} Live-in = {%1}
BB4:	
L2: %3 = PHI (BB0,%1),(BB9, %7)	Live-out = {%3} Live-in = {}
L3: %4 = PHI (BB0,%2),(BB9, %6)	Live-out = {%3,%4} Live-in = {%3}
L4: %5 = ICMP %3, #5	Live-out = {%3,%4,%5} Live-in = {%3,%4}
L5: BR_GE %5, BB19, BB9	Live-out = {%3,%4} Live-in = {%3,%4,%5}
BB9:	
L6: %6 = IADD %4, %3	Live-out = {%3,%4, %6} Live-in = {%3,%4}
L7: %7 = IINC %3	Live-out = {%6,%7} Live-in = {%3, %6}
L8: BR BB4	Live-out = {%6,%7} Live-in = {%6,%7}
BB19:	
L9: CALL printInt(%4)	Live-out = {} Live-in = {%4}
L10: RETURN	Live-out = {} Live-in = {}

Lives-in[s] = Use[s] ∪ (Lives-out[s] – Def[s])

Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

Compute Live Intervals on SSA

- Generate live intervals during liveness analysis
- Live-out(BB) : add the entire BB range
- At SSA definition : set the range start from the def
- At use : add (merge) a live range from BB to the use
- Special handling for
  - Phi functions
  - Loops

Liveness Analysis on SSA Example

```
BB0: (Entry)
L0: %1 = MOV #0
L1: %2 = MOV #0

BB4:
L2: %3 = PHI (BB0,%1),(BB9, %7)
L3: %4 = PHI (BB0,%2),(BB9, %6)
L4: %5 = ICMP %3, #5
L5: BR_GE %5, BB19, BB9

BB9:
L6: %6 = IADD %4, %3
L7: %7 = IINC %3
L8: BR BB4

BB19:
L9: CALL printInt(%4)
L10: RETURN
```

Live-out = {} Live-in = {%4}

$$\text{Lives-in}[s] = \text{Use}[s] \cup (\text{Lives-out}[s] - \text{Def}[s])$$

$$\text{Lives-out}[s] = \bigcup_{p \in succ(s)} \text{Live-in}[p]$$

Range(%4)= (L9, L9)

Liveness Analysis on SSA Example

```
BB0: (Entry)
L0: %1 = MOV #0
L1: %2 = MOV #0

BB4:
L2: %3 = PHI (BB0,%1),(BB9, %7)
L3: %4 = PHI (BB0,%2),(BB9, %6)
L4: %5 = ICMP %3, #5
L5: BR_GE %5, BB19, BB9

BB9:
L6: %6 = IADD %4, %3
L7: %7 = IINC %3
L8: BR BB4

BB19:
L9: CALL printInt(%4)
L10: RETURN
```

Live-out = {} Live-in = {%4}

Live-out = {} Live-in = {%4}

Lives-in[s] = Use[s] $\cup$ (Lives-out[s] – Def[s])

Lives-out[s] = $\cup_{p \in succ(s)}$ Live-in[p]

Range(%4)= (L6, L9)

Liveness Analysis on SSA Example

```
BB0: (Entry)
L0: %1 = MOV #0
L1: %2 = MOV #0

BB4:
L2: %3 = PHI (BB0,%1),(BB9, %7)
L3: %4 = PHI (BB0,%2),(BB9, %6)
L4: %5 = ICMP %3, #5
L5: BR_GE %5, BB19, BB9

BB9:
L6: %6 = IADD %4, %3
L7: %7 = IINC %3
L8: BR BB4

BB19:
L9: CALL printInt(%4)
L10: RETURN
```

Live-out = {%4}   Live-in = {%4}

Live-out = {}   Live-in = {%4}

Live-out = {}   Live-in = {%4}

Lives-in[s] = Use[s] ∪ (Lives-out[s] – Def[s])

Lives-out[s] = $\bigcup_{p \in succ(s)}$ Live-in[p]

Range(%4)= (L2, L9)

Liveness Analysis on SSA Example

```
BB0: (Entry)
L0: %1 = MOV #0
L1: %2 = MOV #0

BB4:
L2: %3 = PHI (BB0,%1),(BB9, %7)
L3: %4 = PHI (BB0,%2),(BB9, %6)      Live-out = {%4}   Live-in = {}
L4: %5 = ICMP %3, #5                 Live-out = {%4}   Live-in = {%4}
L5: BR_GE %5, BB19, BB9              Live-out = {%4}   Live-in = {%4}

BB9:
L6: %6 = IADD %4, %3                 Live-out = {}   Live-in = {%4}
L7: %7 = IINC %3
L8: BR BB4

BB19:
L9: CALL printInt(%4)                Live-out = {}   Live-in = {%4}
L10: RETURN
```

$$\text{Lives-in}[s] = \text{Use}[s] \cup (\text{Lives-out}[s] - \text{Def}[s])$$

$$\text{Lives-out}[s] = \bigcup_{p \in succ(s)} \text{Live-in}[p]$$

Range(%4)= (L3, L9)

```
BUILDINTERVALS
for each block b in reverse order do
    live = union of successor.liveIn for each successor of b

    for each phi function phi of successors of b do
        live.add(phi.inputOf(b))

    for each opd in live do
        intervals[opd].addRange(b.from, b.to)

    for each operation op of b in reverse order do
        for each output operand opd of op do
            intervals[opd].setFrom(op.id)
            live.remove(opd)
        for each input operand opd of op do
            intervals[opd].addRange(b.from, op.id)
            live.add(opd)

    for each phi function phi of b do
        live.remove(phi.output)

    if b is loop header then
        loopEnd = last block of the loop starting at b
        for each opd in live do
            intervals[opd].addRange(b.from, loopEnd.to)

    b.liveIn = live
```

Christian Wimmer, and Michael Franz. "Linear scan register allocation on SSA form."
In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 170-179. ACM, 2010.

Interference Graph

- Nodes with overlapping intervals are connected
- Using the live intervals

```
for each SSA value si do
  for each SSA value sj (j < i) do
    if (sj interferes live-range(si) or
        si interferes live-range(sj)) then
      E ← E U (si,sj)
```

Allocating Registers with the Interference Graph

- K-coloring
  - Color graph nodes using up to k colors
  - Adjacent nodes must have different colors

- Allocating to k registers -> finding a k-coloring of the interference graph
  - Adjacent nodes must be allocated to distinct registers

- Getting optimal solution is NP-Complete : Heuristics

Simple Greedy Algorithm

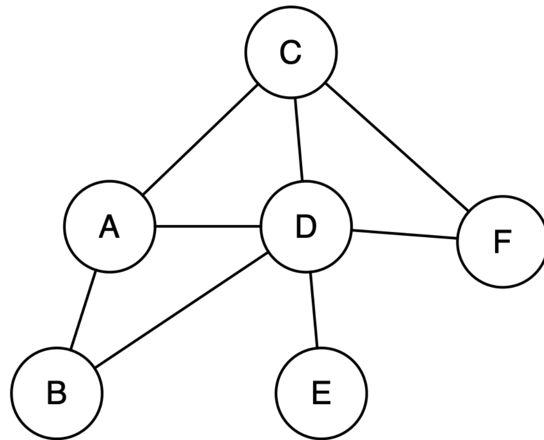
```
for each n ∈ N do  // select n in decreasing order of weight
  if n can be colored then
    reserve a register for n
  else
    Remove n (and its edges) from graph & allocate n to stack (spill)
```

Improved Algorithm by Chaitin 81

- Idea
  - Nodes with < k neighbors are guaranteed colorable
- Remove them from the graph first
  - Reduces the degree of the remaining nodes
- Must spill only when all remaining nodes have degree >= k

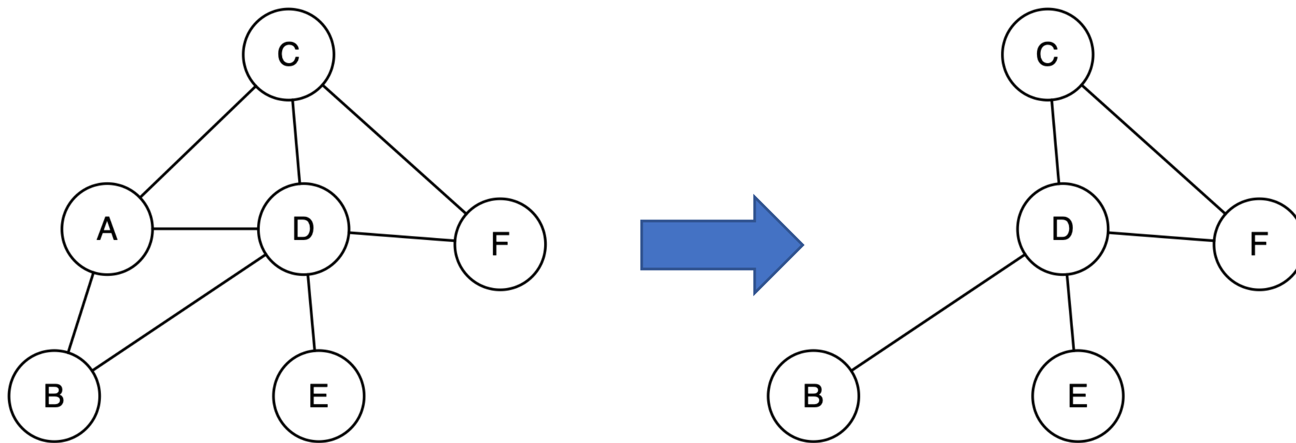
Example with k=4

- Degree(D) > 4 : should it be spilled?



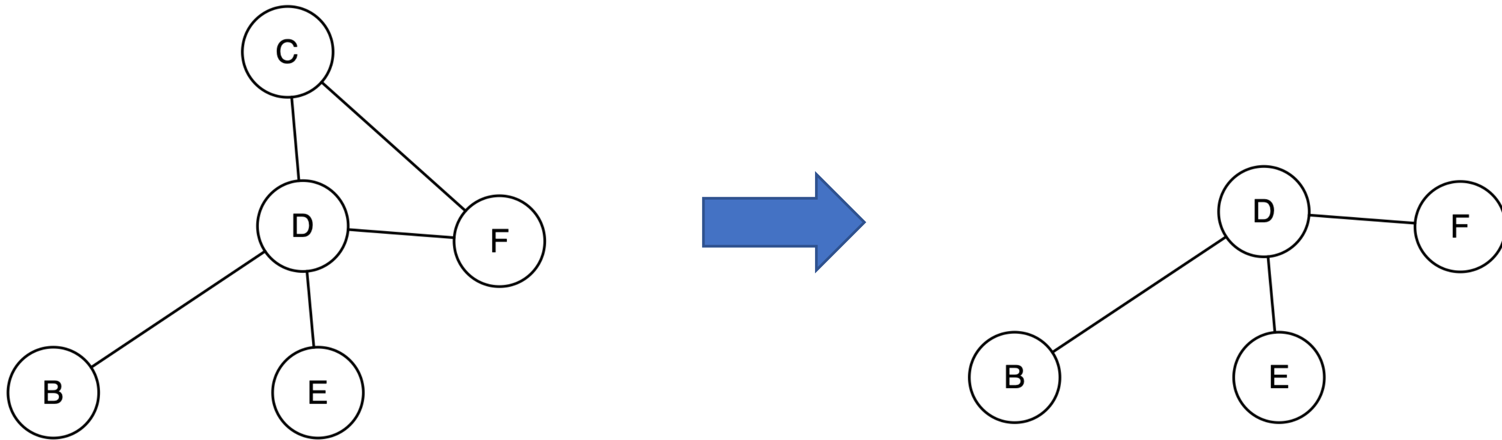
Example

- Degree(A) < k, so remove A from G



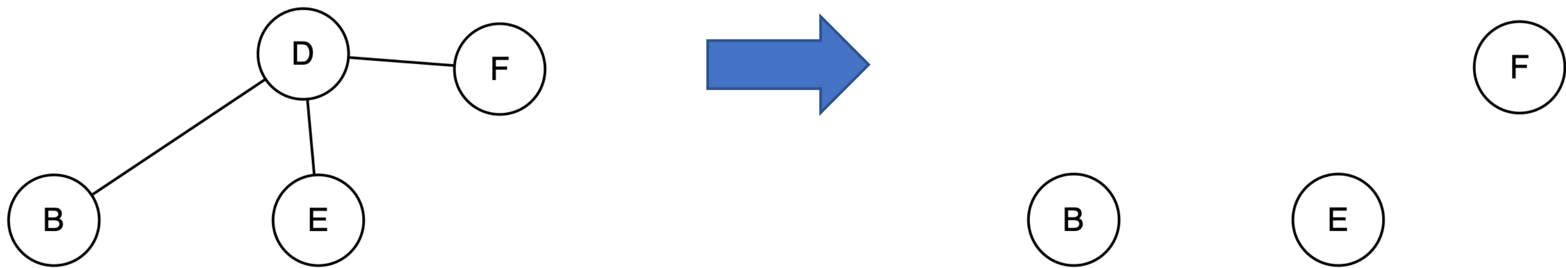
Example

- Degree(C) < k, so remove C from G

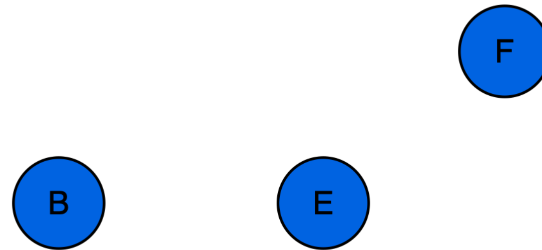


Example

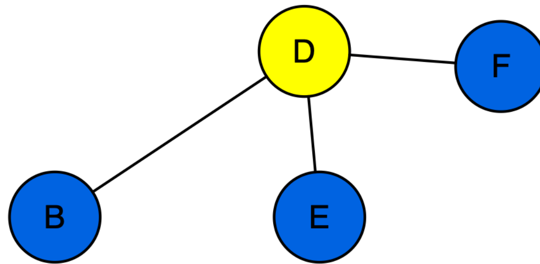
- Degree(D) < k, so remove D from G



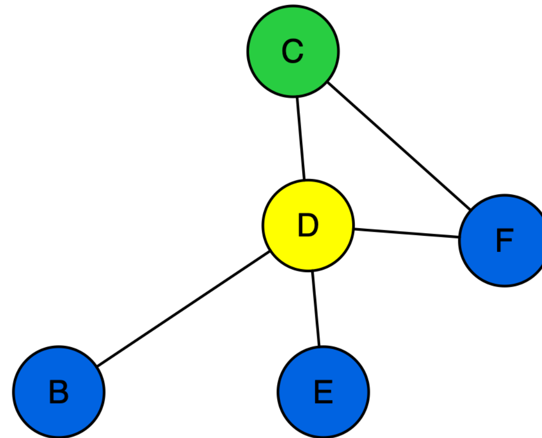
Example – Coloring Subgraph



Example – Coloring Subgraph

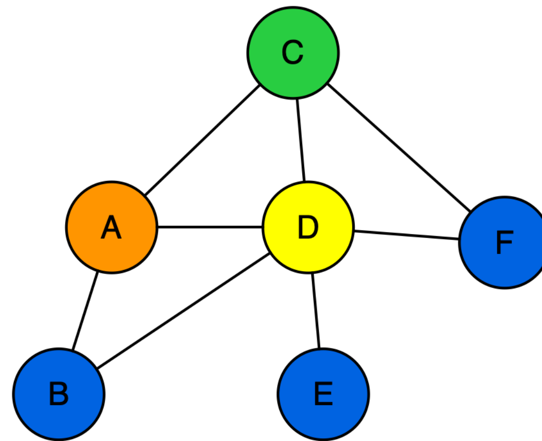


Example – Coloring Subgraph



Example Coloring the graph

- Graph colored with 4 colors!



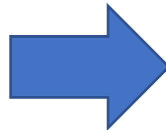
Improved Algorithm by Chaitin 81

```
while interference graph not empty do
  while ∃ a node n with < k neighbors do          } simplify
    Remove n from the graph
    Push n on a stack
  if any nodes remain in the graph then          { blocked with >= k edges }
    Pick a node n to spill                       { lowest spill-cost or }      } spilling
    Add n to spill set                           { highest degree }
    Remove n from the graph
if spill set not empty then
  Insert spill code for all spilled nodes { store after def; load before use }
  Reconstruct interference graph & start over
while stack not empty do                    } allocate register
  Pop node n from stack
  Allocate n to a register
```

Register Spilling

- Allocate a new stack slot
- Store to the slot right after the definition
- Load from the slot right before all uses
- Update operands of the instructions accordingly

```
BB9:
%6 = IADD %4, %3

BB19:
%7 = ISUB %6, 1
```



```
BB9:
%6 = IADD %4, %3
STORE [SP], %6

BB19:
%8 = LOAD [SP]
%7 = ISUB %8, 1
```

Register Allocation Step by Step

- Compute live intervals for SSA values based on liveness analysis (post-order traversal)
- Construct the interference graph for SSA values using the live intervals
- Allocate registers using the interference graph
  - Solving K-coloring problem (Chaitin 81)
- Output: maps (SSA value -> Physical register)

X86 Registers

- General registers
  - EAX, EBX, ECX, EDX, ESI, EDI
- Stack & base pointers
  - EBP, ESP
- Never use EBP and ESP

Tips: Loop Detection

- Assume a well-structured loop
- Simply use DFS to detect a cycle (the target becomes the loop header, the source becomes the loop end)

Tips: Spilling

- Add Instruction classes: LoadInst and StoreInst
- Maintain stack slots in the Method class

```
class StackSlot : public Value {
   int Index;
  …
}

class Method {
  …
  vector<StackSlot*> NativeStack;
}
```