Register Allocation

CS 502

Lecture 14

11/25/08

Where we are...

- Reasonably low-level intermediate representation:
  - sequence of simple instructions followed by a transfer of control.
  - a representation of static environments
- Still need to:
  - determine how variables are mapped to registers
  - formulate a representation for state saving and restoration.

Other Considerations

- Interaction with runtime system:
  - Memory management (garbage collection)
  - Foreign function interfaces
  - Concurrency, distributed programming
- Data type layout
  - How should objects be represented?
  - What about type checks, casts?
- Procedure calling conventions

Goals

- Want a virtual machine that
  - Is a reasonably close match to "real" architectures, but
  - Exposes important language-specific features
    - Types
    - Object structure
    - Closure representation
  - Enables retargetable code generation

Code Generation

- Specify translation using so-called “two-level language”
  - Distinguish between parts that can be done at compile-time
    - Computing offsets of free variables
  - From parts to be executed at runtime
    - Actual code sequence
    - Memory allocation
- Distinguish between structures manipulated by the compiler and those manipulated by the runtime system
  - Environments that associate continuation variables with offsets on a frame
  - Stack frames and heaps

Data Layout

- What information should heap-allocated objects record?
  - Vectors: length of vector
  - Closures: label of beginning of code sequence
  - Constructors: identity
- In a realistic compiler, would require more information:
  - Where are pointers stored? Important for garbage collection.
- Must also add a pointer object that distinguishes references to heap allocated object from the objects themselves.

Approach

- Relationship between code generation and virtual machine architecture:
  - VM provides basic support for
    - Allocation
    - Jumps
    - Load and Store
    - Garbage collection (still need to discuss)
  - Goal: define code generator in terms of locally defined rules
    - Implications for instruction scheduling and selection

Abstraction

- Mapping to a real architecture:
  - Caller vs. callee saves registers
    - How do we deal with this issue?
  - Header information:
    - How do distinguish between pointers and integers?
      - GC masks in stack frames
      - What about polymorphism?
    - What about other heap-allocated objects?
  - Infinite number of registers
    - Where should these registers be mapped when we do register allocation?
  - Is one stack sufficient? Do we need a stack?
  - Is one heap sufficient? What about multiple heaps?

Next steps

- Register Allocation
  - Mapping virtual registers onto a finite set of physical registers
- Data Representation issues:
  - Closure and datatype representation
- Instruction selection
  - Mapping virtual machine instructions to sequence of real machine instructions
- Instruction scheduling
  - Ordering instructions to take advantage of machine-specific characteristics (e.g., branch delays, cache misses, etc.)
- Peephole optimization
  - Eliminating redundant loads/stores
  - Dead-code elimination
  - Loop hoisting (localized)
  - Simple arithmetic simplification on registers
- Which order should these transformations be applied?

Register Allocation

- Separation of concerns:
  - During optimization/code generation, assume an infinite collection of registers, treat register allocation as a separate problem.
  - Physical registers are expensive:
    - Small numbers
    - Fast access

Where should it be performed?

- Expression
- Local (basic block)
- Loop
- Global (procedure)
- Interprocedural

Assumptions:

- Low-level intermediate or assembly code
- Unlimited number of virtual registers
- Global optimization (flow analysis) has been performed

Why is it hard?

- Control-flow (especially loops)
- Machine dependent
- IR instructions may not match assembly
- Calling conventions (caller vs. callee saves)
- Restricted instructions

Practical considerations:

How much time and space do we expend to find a good mapping?

Terminology

- A value corresponds to a definition (or the result of a definition)
- A variable is live if it holds a value that may be needed in the future
- A live range is composed of one or more values, connected by common uses.
  - All values comprising a live range will be read by the same virtual register
  - A single virtual register may have several live ranges
- A single virtual register may comprise several live ranges
- Interference graph:
  - Vertices represent variables (more precisely distinct definitions)
  - Edges represent interference between variables with overlapping live ranges (i.e., both live ranges are live at some point, and cannot use the same physical register)
  - A coloring represents a register assignment

Example: CFG

a ← 0

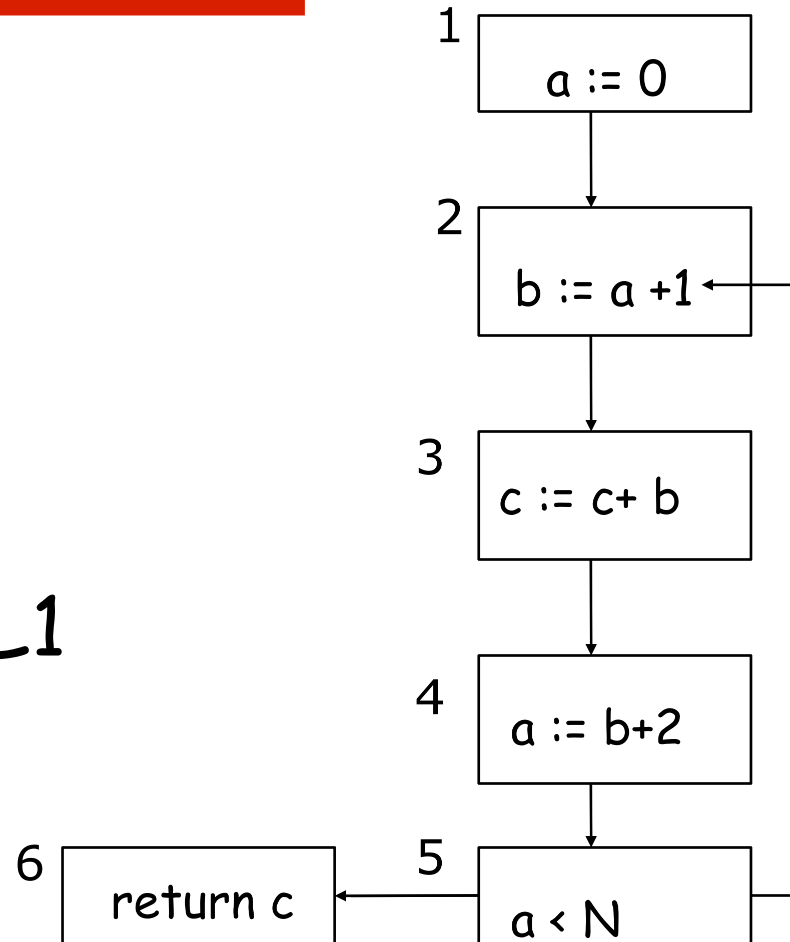
L1: b ← a + 1

c ← c + b

a ← b * 2

if a < N goto L1

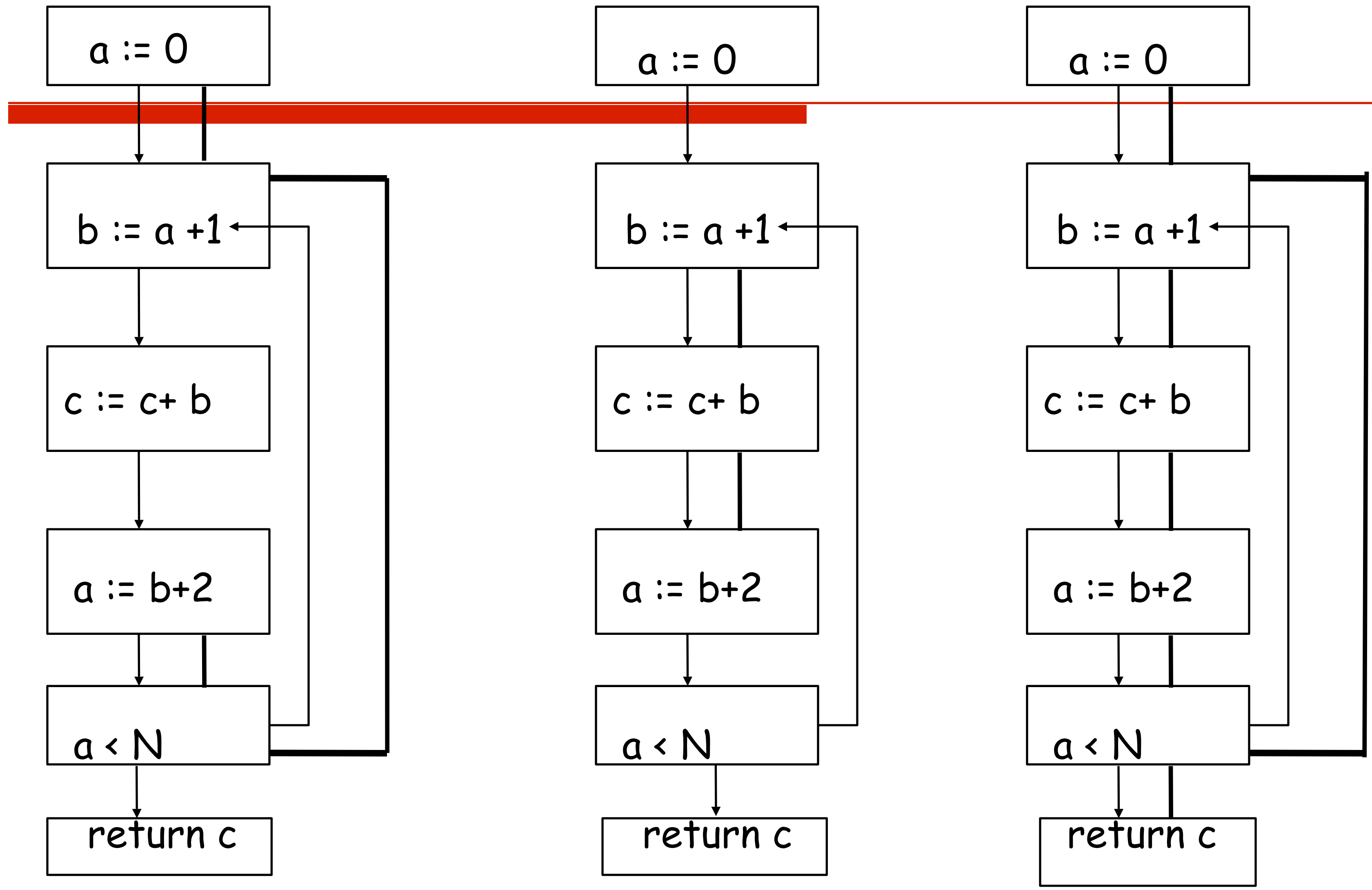
return c



Flow Graph Terminology

- Out-edges: lead to successor nodes
- In-edges: come from predecessor nodes.
- Example:
  - out-edges of node 5 (a < N) are 5 -> 6 and 5 -> 2. succ[5] = {2,6}
  - in-edges of 2 are 5 -> 2 and 1 -> 2. pred[2] = {1,5}
- The def of a variable is the set of graph nodes that define it. The use of a variable is the set of graph nodes that use it. def(3) = {c}; use(3) = {b,c}
- A variable is live on an edge if there is a directed path from that edge to a use of a variable that does not go through another def.

Live Ranges



Interference

- Two live ranges interfere if at some point in the routine:
  - Both live ranges have been defined
  - Both live ranges will be used
  - The live ranges have different values
- Generally undecidable
- Conservative approximation:
  - At each definition in the routine, make the defined live range interfere with all live ranges that
    - Are available
    - Are live

Copy Propagation

- At a copy instruction, check if source and target live ranges interfere. If not, remove copy instruction

```
x ← ...
  ...
y ← x
z ← x + 1
  ...
  ← y
  ← z
```

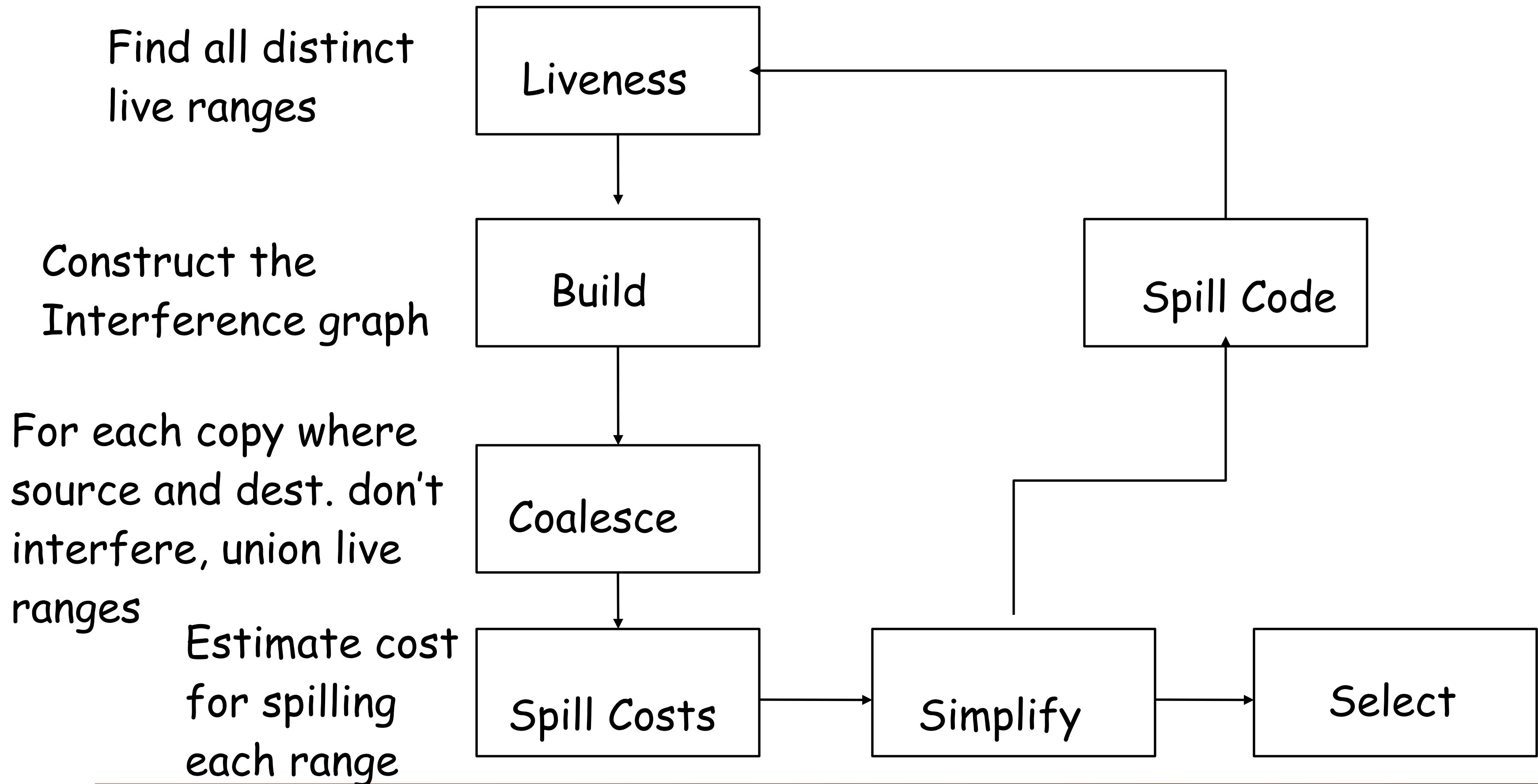
```
xy ← ...
  ...
z ← xy + 1
  ...
  ← xy
  ← z
```

y and z interfere, but x and y don't. Why?

Interference Edges

- At any non-move instruction that defines a variable a, where the live-out variables are b1,…,bn, add interference edges (a,b1), (a,b2),.., (a,bn).
- At any move instruction a <- c where variables b1,…,bn are live-out, add interference edges (a,b1),(a,b2),…, (a,bj) for any bi not the same as c.

Basic Structure



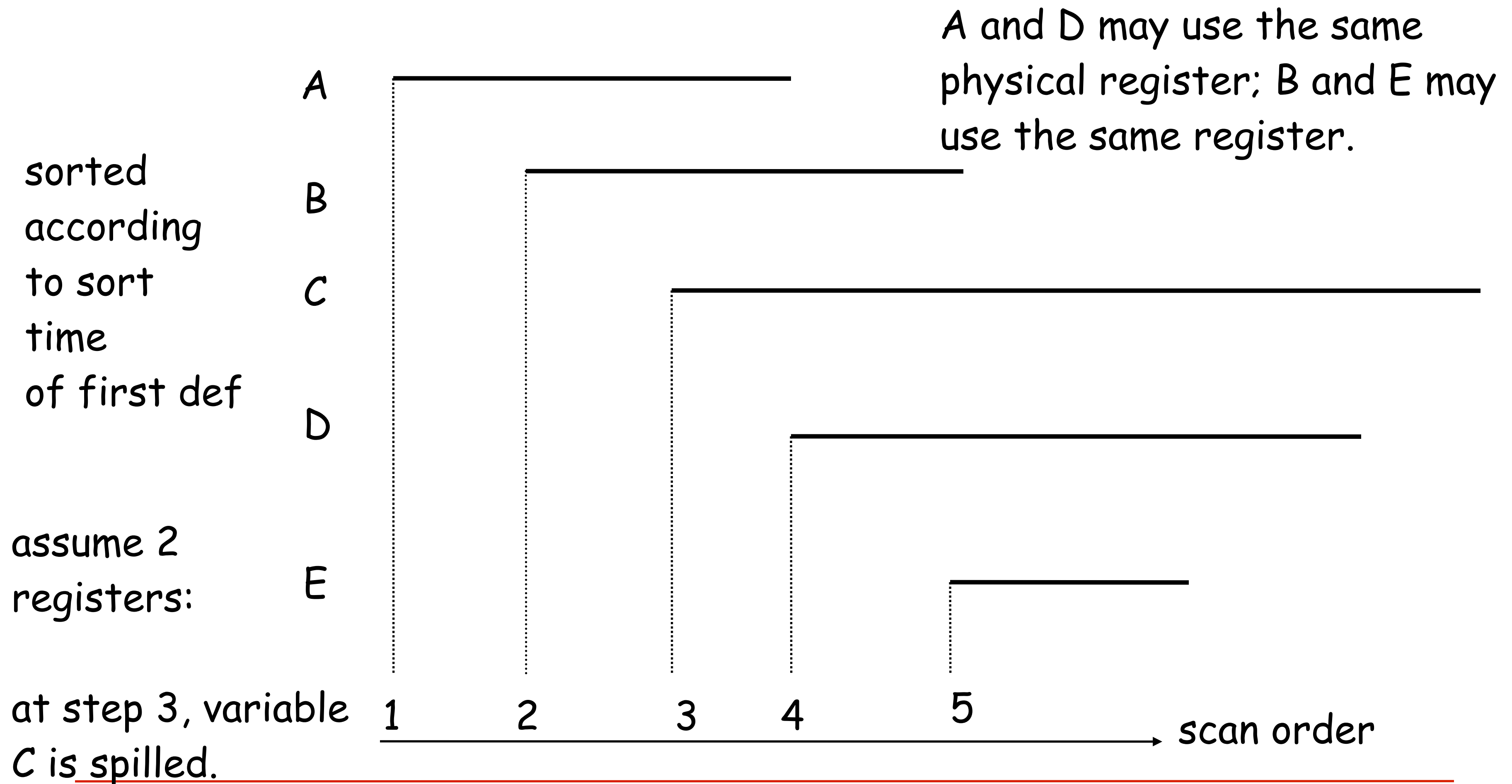
Linear-Scan Register Allocation

- A simple local allocation algorithm
- Assume code is already scheduled
- Build a linear ordering of live ranges (also called live intervals or lifetimes)
- Scan the live range and assign registers until you run out of them – then spill

Live Interval

- Assume some numbering of instructions in the IR
- [i,j] is said to be a live interval for variable v if
  - there is no instruction with number $j' > j$ such that v is live at j'
  - No instruction with number $i' < i$ such that v is live at i'
- Conservative approximation of live ranges:
  - There may be subranges of [i,j] in which v is not live.

Basic Idea



Summary

- Linear scan register allocation is composed of 4 simple steps:
  - Order the instructions in linear fashion
    - Many have proposed heuristics for finding the best linear order
  - Calculate the set of live intervals
    - Each temporary is given a live interval
  - Allocate a register to each interval
    - If a register is available, then allocation is possible
    - Otherwise, an already allocated register is chosen (register spill occurs)
  - Rewrite the code according to the allocation
    - Actual registers replace temporary or virtual registers
    - Spill code is generated

The Algorithm

```
LinearScanRegisterAllocation
  active ← {}
  foreach live interval i in order of increasing
              start point
    ExpireOldIntervals(i)
    if length(active) = R then
      SpillAtInterval(i)
    else
     register[i] ← a free register
     add i  to active, sorted by increasing end point
```

The Algorithm

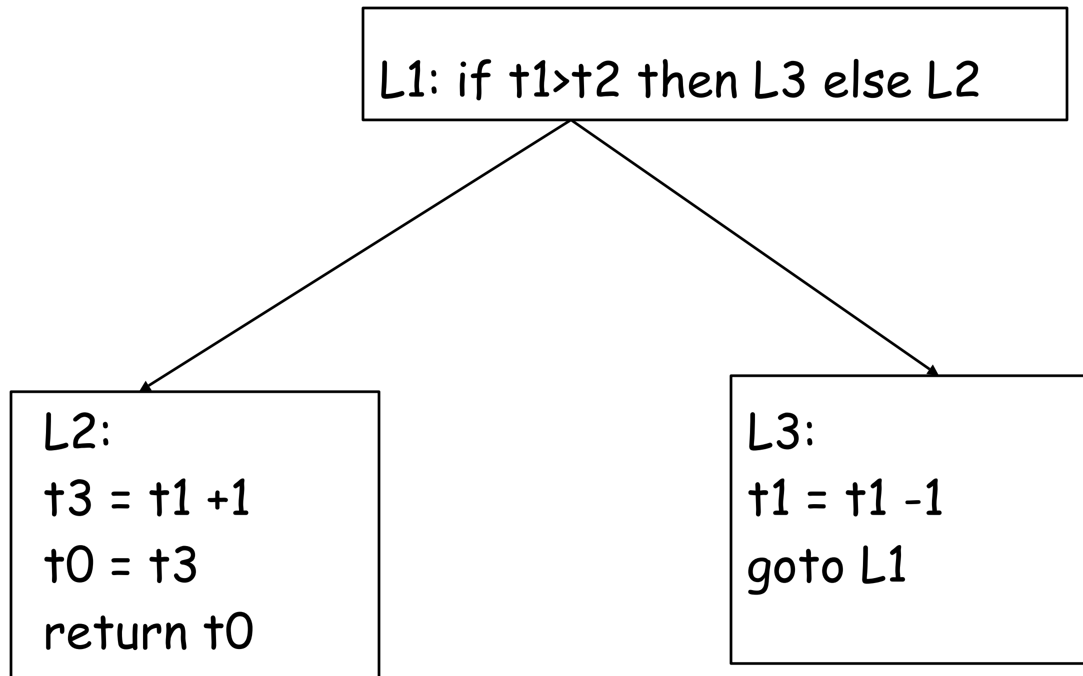
```
ExpireOldIntervals(i)
  foreach interval j in active, in order of
    increasing end point:
      if endpoint[j] >= startpoint[i]
        then return
      remove j from active
      add register[j] to pool of free registers
```

The Algorithm

```
SpillAtInterval(i)
 spill ← last interval in active
 if endpoint[spill] >= endpoint[i]
   then register[i] ← register[spill]
        location[spill] ← new stack loc.
        remove spill from active
        add i to active, sorted by increasing end point
   else location[i] ← new stack location
```

Example

- Given the following CFG:



Example

- Find a linear ordering of instructions:
  - The a priori choice of ordering affects performance
  - Exhaustive search not feasible
  - One possible ordering: (based on IR)

```
0: L1: if t1 > t2 then L3 else L2;
1: L3: t1 = t1 - 1
2:     goto L1;
3: L2: t3 = t1 + 1;
4:     t0 = t3;
5:     return t0
```

Computing Live Intervals

```
0: L1: if t1 > t2 then L3 else L2;
1: L3: t1 = t1 – 1
2:       goto L1;
3: L2: t3 = t1 + 1;
4:       t0 = t3;
5:       return t0
```

Live Intervals:

t0[4,5] t2[0,2] only overlap is t2 and t1

t1[0,3] t3[3,4]

Allocate registers to intervals

- Maintain three lists:
  - Free: set of available registers
  - Alloc: set of allocated registers
  - Active: list of active intervals ordered by increasing end points
- Assign registers:
  - Order the intervals in order of increasing end points
  - Scan the list of intervals; select the next ti
  - Free all registers assigned to intervals in Active whose interval is less than or equal to start of ti
  - If a free register exists in Free, allocate it
  - If Free is empty, then spill:
    - If last interval on the Active list end beyond the interval for ti, then ti is given that register, and ti's interval is added to active
    - If ti's interval ends at the same point or beyond the last interval in Active, then ti is given a stack location

Allocate registers to intervals

t2[0,2]

t1[0,3]

t3[3,4]

t0[4,5]

Free = {r1,r2}

Active = {}

Alloc = {}

-- Looking at t2,Allocate r1

Free = {r2}

Active = {t2:[0,2]}

Alloc = {r1:t2}

-- Looking at t1, Allocate r2

Active = {t1:[0,3], t2:[0,2]

Free = {}

Alloc = {r1:t2,r2:t1}

Allocate registers to intervals

t2[0,2]

t1[0,3]

t3[3,4]

t0[4,5]

More intervals to process

start of interval 3

Free = {r1}

Alloc = {r2:t1}

-- Looking at t3, allocate r1

Active = {r1:t3,r2:t1}

start of interval 4

Free = {r2}

Alloc = {r1:t3}

-- Looking at t0, allocate r2

Active = {r1:t3,r2:t0}

Consider a different ordering

t1[0,5]

t2[0,5]

t3[1,2]

t0[2,3]

Free = {r1,r2}

Active = {}

Alloc = {}

-- Looking at t1,Allocate r1

Free = {r2}

Active = {t1:[0,5]}

Alloc = {r1:t1}

-- Looking at t2, Allocate r2

Active = {t1:[0,5], t2:[0,5]

Free = {}

Example (cont)

t1[0,5]

t2[0,5]

t3[1,2]

t0[2,3]

More intervals to process

Active = {t2:[0,5], t1:[0,5]

start of interval 1

Alloc = {r2:t2, r1:t1}

-- Looking at t3

-- Free is empty

-- End of t1 > t3

-- t3 allocated to r1, t1 on stack

Free = {}

Active = {t3:[1,2], t2:[0,5]}

Alloc = {r1:t3,r2:t2}

Example (cont)

t1[0,5]

t2[0,5]

t3[1,2]

t0[2,3]

Alloc = {r1:t3, r2:t2}

-- start of interval 2

-- Looking at t0

-- Free is empty (spill needed)

-- end of t2 > end of t0

-- t0 is allocated to r2, t2 on stack

Rewriting the code

- Code is rewritten with assigned registers and spill code inserted
- Spill code is additional code that may increase cycle time:

```
0: L1: t1(r1) > t2(r2) then L3 else L2
1:   spill(t1(r1)) and free r1
2: L2: t3(r1) = t1(stk) + 1
3:   spill(t0(r2)) and free r2
4:       t0(r2) = t3(r1)
5:       return t0(r2)
6: L3: t1(stk) = t1(stk) – 1
7:       goto L1
```