Machine Description

- The machine description describes valid instructions for a machine.
- It is implemented using a YACC grammar and associated group of semantic actions.
- Typical organization:
  - instructions
  - effects (register transfers) of instructions
  - operands of instructions

Machine Description Applications

- Used as a recognizer any time a new instruction is created or modified.
  - instruction selection
  - common subexpression elimination

```
R[r[13]+i.]=r[7];
R[r[13]+i.]=R[r[13]+i.]+1;
=>
R[r[13]+i.]=r[7];
R[r[13]+i.]=r[7]+1; // legal?
```

  - code motion
  - loop strength reduction

Machine Description Applications (cont.)

- Translate an RTL to an assembly language or machine code instruction.
- Determine an estimated cost of the instruction. This is used by the CSE phase to check if the replacement RTL is cheaper than the original.
- Determine the type of an instruction for instruction scheduling, etc.
- Produce detailed measurements.

Evaluation Order Determination

- Evaluation order determination is performed to reduce the number of registers required by an expression.
- Treats an expression as a tree. Determines the cost (number of registers) of performing each subtree. Orders the generation of code for the tree so that the most expensive subtrees are done first.

Data-Flow Analysis

- Data-flow analysis collects global information about how a function manipulates its data and distributes this information to each block in the control-flow graph.
- Data-flow information can be collected by solving systems of dataflow equations that relate information at various points in the function.
- Information is usually represented as bits in a vector so operations (e.g., union and intersection) can be efficiently applied.

General Types of Data-Flow Analysis

- structural analysis
  - Uses detailed information about control structures to produce data-flow equations.
  - Can be efficient when control-flow graphs are guaranteed to be reducible (only one entry for each loop).
- iterative analysis
  - Solved by iteration until information reaches a fixed point.
  - Easy to implement and works on any flow graph.
  - Most commonly used.
- demand-driven analysis
  - Only obtain needed dataflow information upon demand.
  - Limit the analysis to the portion of the program representation needed to answer the specified query.
  - Usually is accomplished in a recursive manner.

Live Variable Analysis

- For variable x and point p, we wish to know if the value of x at p could be used along some path in the flow graph. If so, then x is live at p, otherwise x is dead at p.
- Used for establishing links for instruction selection and performing register assignment, register allocation, code motion, basic induction variable elimination, etc.

Defs and Uses for Live Variable Analysis

- Live variable analysis in VPO is associated with registers and scalars that are local variables or arguments.
- def[B] - the set of items assigned values in B prior to any use of the item in B
- use[B] - the set of items whose values are used in B prior to any assignment
- We will see that *defs* and *uses* are used to later calculate the ins and outs.

Example of Defs and Uses

- Say block B consists of the following RTLs.

```
r[8]=r[8]+1;
r[5]=13;
r[2]=r[5]+r[8];
r[9]=r[2]*r[11];
```

- The following information can be calculated for this block.

```
def[B]=r[5],r[2],r[9]
use[B]=r[8],r[11]
```


Ins and Outs for Live Variable Analysis

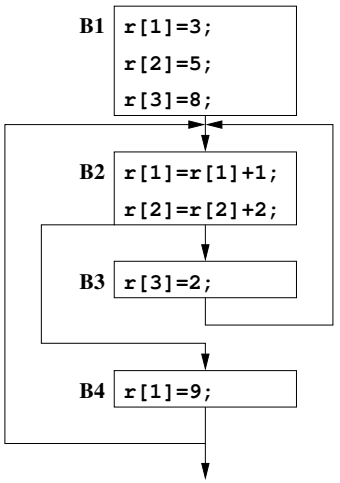
- in[B]
  - the set of items which are live immediately before entering B
  - in[B] = use[B] U (out[B] - def[B])
- out[B]
  - the set of items which are live immediately after exiting B
  - out[B] = U in[S], for each immediate successor S of B
- Note that in[B] depends on out[B] and out[B] depends on in[S].

Iterative Algorithm to Calculate Ins and Outs

```
for all B
   in[B] = empty set;
do {
   change=FALSE;
   for all B {
      out[B] = empty set;
      for all immediate succs S of B
         out[B] = out[B] U in[S];
      oldin = in[B];
      in[B] = use[B] U (out[B]-def[B]);
      if (in[B] != oldin)
         change = TRUE;
   }
} while (change);
```


Example for Live Variable Analysis

- Calculate the live variable information for the example below.



Defining Links within a Basic Block

- VPO determines where each register is set and where the first use of that register is within the block.
- A link is set if it is safe.
- Example of an unsafe link:

```
 r[17]=r[0];          r[0]:
 ...
 ST=HI[foo]+LO[foo];
Sr[0]:...
 ...
 r[18]=r[17];         r[17]:
```


Example of an Unsafe Global Link

- Global links cross basic block boundaries.

```
Before______________        After_______________
1    IC=r[6]?5;             1    IC=r[6]?5;
2    PC=IC==0,L5;           2    PC=IC==0,L5;
3    r[7]=0;                3    r[7]=0;
4    PC=L6;                 4    PC=L6;
 L5:                         L5:
5    r[7]=1;
 L6:                         L6:
6    R[_base]=r[7];         6    R[_base]=1;
```


Rule 1 for Global Links

- Define a link from the use of an item only if just one definition of the item reaches the use.
- VPO uses SSA information to detect if more than one definition reaches a use.
- In the previous example, two definitions of r[7] reach the use at RTL 6.

Another Example of an Unsafe Global Link

```
Before______________        After_______________
1    r[9]=0;
2    IC=r[7]?r[8];          2    IC=r[7]?r[8];
3    PC=IC!=0,L9;           3    PC=IC!=0,L9;
4    r[9]=r[9]+5;           4    r[9]=5;
5    PC=L2;                 5    PC=L2;
 L9:                         L9:
6    r[9]=r[9]+8;           6    r[9]=r[9]+8;
 L2:                         L2:
7    R[_base]=r[9];         7    R[_base]=r[9];
```

Rule 2 for Global Links

- A link can only be defined from the first use if it is the only first use of the definition.
- In the previous example there is a first use at RTL 4 and a first use at RTL 6.

Register Assignment

- Pseudo registers contain temporary values.
- Register assignment is the mapping of pseudo registers to hardware registers.
- Sometimes this phase is called local register allocation and may also be combined with code generation.

Number of Pseudo Registers Used

- The code expander uses on average about 10 pseudo registers for each source-level statement.
- Pseudo registers are never live across source-level statements.
- After the initial instruction selection, this number is reduced to about 5.
- Usually only a maximum of 2 or 3 pseudo registers are live simultaneously at any point in the function.
- Evaluation order determination can decrease this number.

Register Assignment (cont.)

- Typically, the register assigner succeeds in allocating only as many hardware registers as the maximum number of simultaneously live pseudo registers.
- Exceptions
  - dedicated registers for specific operations
  - pseudo register is live across a function call
  - required use of register pairs

Example of Assigning Registers

- Evaluates uses, deads, sets in that order for each RTL.

```
    r[32]=R[_a];
    r[33]=r[32]+1; r[32]:
    R[_m]=r[33];   r[33]:
    r[34]=R[_n];
 =>
    r[1]=R[_a];
    r[33]=r[1]+1;  r[1]:  -- r[1] is free
    R[_m]=r[33];   r[33]:
    r[34]=R[_n];
 =>
    r[1]=R[_a];
    r[1]=r[1]+1;          -- r[1] set, not dead
    R[_m]=r[1];    r[1]:
    r[34]=R[_n];
```


Register Spills

- Register spills are introduced when the number of live pseudo registers exceeds the number of allocable registers available on the target machine.
- The hardware register chosen to spill (store is generated) is the one whose next pseudo register reference is furthest away.
- At the point the next reference to this spilled pseudo register is encountered, a new hardware register is associated with the pseudo register and a load inserted to get the value from memory.

Example of a Register Spill

```
   1  r[33]=R[i];
   2  r[35]=r[33];
      ...
   10 r[38]=R[j];        -- spill needed here
      ...
   16 r[38]=r[38]+r[33];
=>
   1  r[2]=R[i];
   2  r[3]=r[2];
      ...
   9  R[tmp]=r[2];       -- spill is inserted
   10 r[2]=R[j];         -- now can use r[2]
      ...
   15 r[3]=R[tmp];       -- r[3] now available
   16 r[2]=r[2]+r[3];
```


Dedicated Uses of Hardware Registers

- Whenever there is a dedicated use of a hardware register, the code expander needs to insert a USELINE after the instruction indicating the hardware register that has been used.
- Before assigning a pseudo register to a hardware register, the compiler checks that the hardware register is not referenced by a USELINE in the life of the pseudo register.

Example of a Dedicated Use of a Hardware Register

```
   ...=a+foo();
=>
   r[32]=R[_a];
   ST=foo;
  rr[0]                          -- RESLINE
  Ur[0]r[1]...r[15]              -- USELINE
   r[33]=r[0];          r[0]:
   r[33]=r[33]+r[32];
```

- r[32] must be assigned to a nonscratch register.

Detecting Loops

- Detecting loops is important since code inside loops is typically executed much more often than code outside of loops.
- Optimizations associated with loops in VPO include:
  - loop inversion
  - loop-invariant code motion
  - loop strength reduction
  - induction variable elimination
  - recurrence elimination
  - loop unrolling

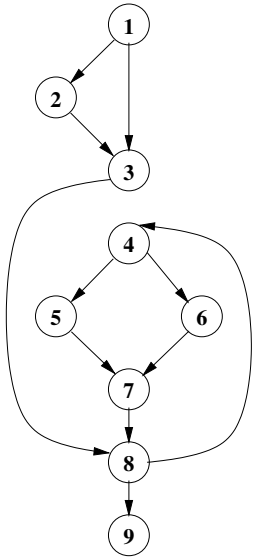
Dominators

- Block *d* dominates block *n* if every path from the initial block of the flow graph to *n* goes through *d*. A block always dominates itself.
- Dominator information is used to calculate natural loops, which will be described later.
- Dominators are also used in a variety of other optimizations.
  - loop-invariant code motion
  - detection of basic induction variables

Calculating Dominator Information

```
dom(top) = {top};
for each block n in N-{top} do
   dom(n) = N;
do {
   for each block n in N-{top} do
      dom(n) = intersection of all dom(p), where p
               are the immediate preds of n;
      dom(n) = dom(n) U {n};
} while (any change to any dom(n));
```

Example of Calculating Dominators



Natural Loops

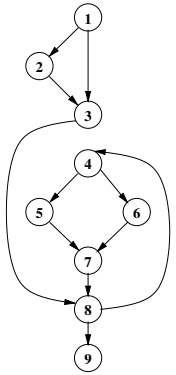
- A natural loop is a loop with a single entry point called the *header*.
- The header dominates all of the nodes in the loop.
- There must be at least one path back to the header.
- A backedge is an edge in a flow graph where the head dominates the tail.
  - $a \rightarrow b$ is an edge where b is the head (successor) and a is the tail (predecessor)
  - if b dominates a then this edge is a backedge

Constructing a Natural Loop

- Put in the loop header (head block of the backedge).
- Put in the loop tail (tail block of the backedge).
- Put in all the blocks in between by following the predecessors starting from the tail.

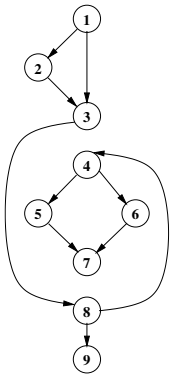
Example of Constructing a Natural Loop

- Look at example of calculating dominators.
- The backedge is $7 \rightarrow 8$ (header is 8 and tail is 7).
  - loop = {8}
  - loop = {7,8}
  - loop = {5,7,8}
  - loop = {4,5,7,8}
  - loop = {4,5,6,7,8}

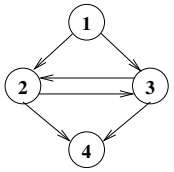


Reducible Flow Graphs

- A reducible flow graph is one whose edges can be partitioned into two sets:
  - backward edges - as defined earlier
  - forward edges form a DAG (directed acyclic graph)



Example of a Nonreducible Flow Graph



- $2 \rightarrow 3$ and $3 \rightarrow 2$ are not backedges since neither node dominates the other.
- However, the graph is still cyclic.
- Most compilers only perform optimizations on natural loops.

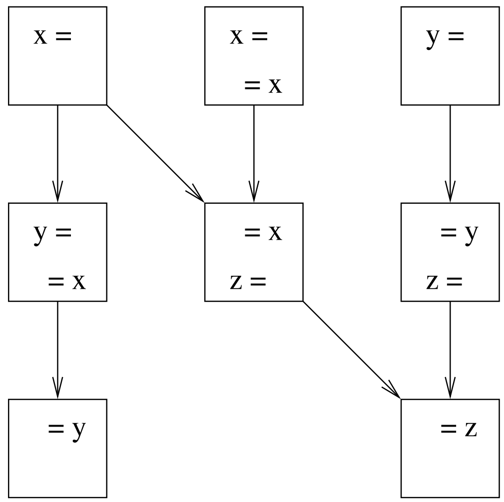
Register Allocation

- Register allocation is a compiler optimization that replaces references to variables with registers.
- The register set used may depend on the type of the variable.
- Register allocation is very beneficial:
  - Directly reduces the number of accesses to the memory system.
  - Indirectly reduces the number of instructions executed.
- Two main types of register allocation:
  - Allocate variables for whole function.
  - Allocate live ranges of variables.

Live Ranges of Variables

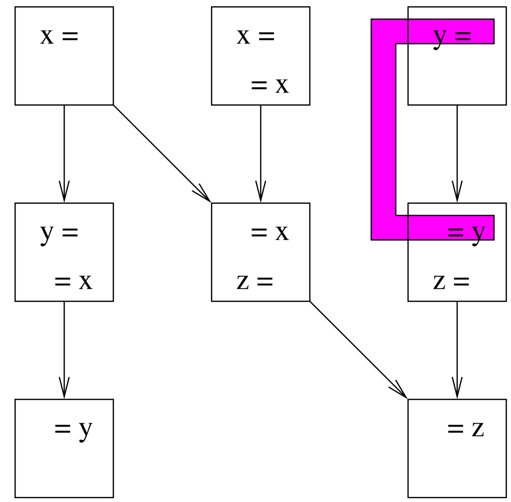
- A live range of a variable represents the points within a function where a variable's value needs to be retained.
- Starts with set(s) of a variable.
- Ends with the last reachable use(s) of the variable that was set.

Example of Live Ranges



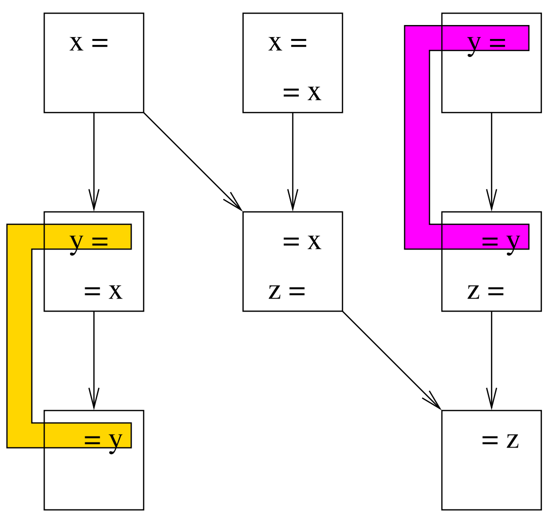
MD EOD DFA Links Reg Assign Loops **Register Allocation** Prop Peep CSE

Example of Live Ranges (cont.)



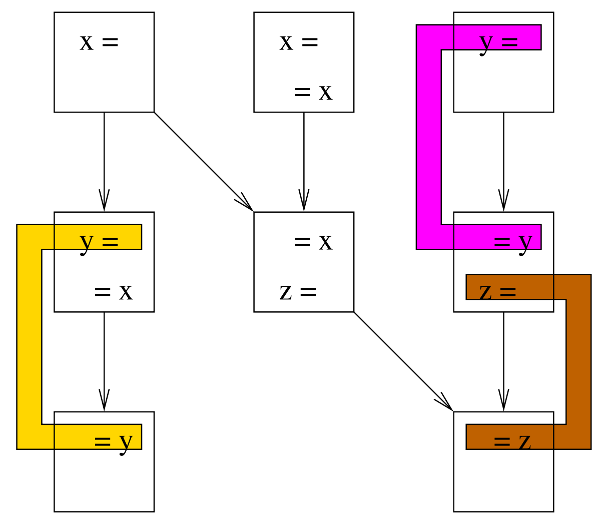
MD EOD DFA Links Reg Assign Loops **Register Allocation** Prop Peep CSE

Example of Live Ranges (cont.)

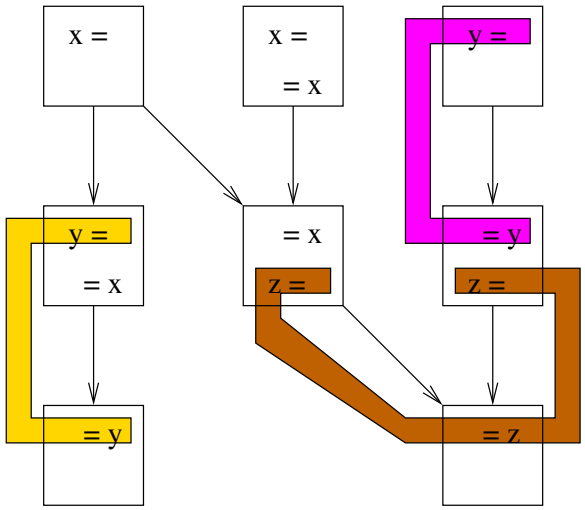


MD EOD DFA Links Reg Assign Loops **Register Allocation** Prop Peep CSE

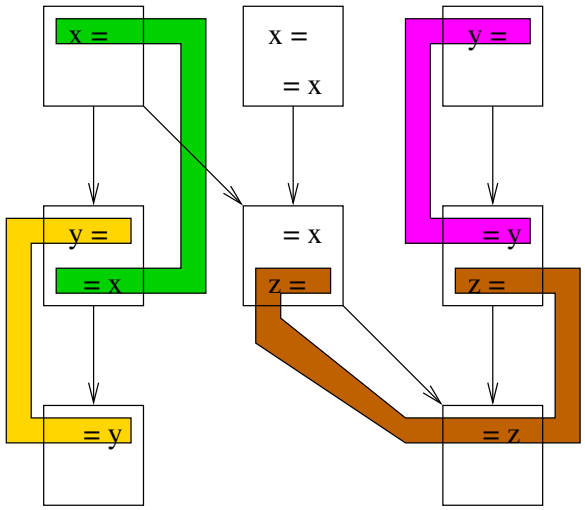
Example of Live Ranges (cont.)



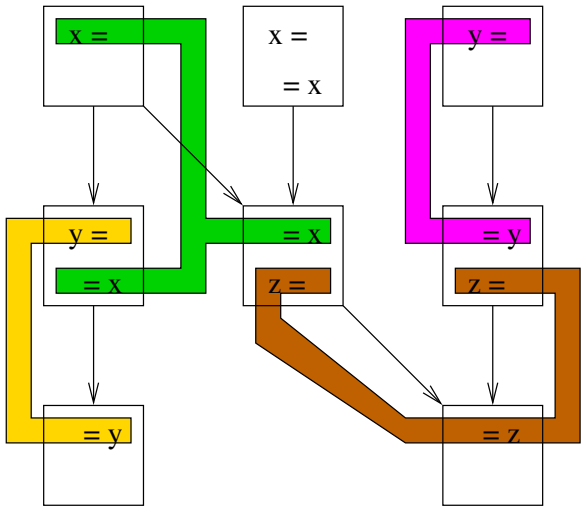
Example of Live Ranges (cont.)



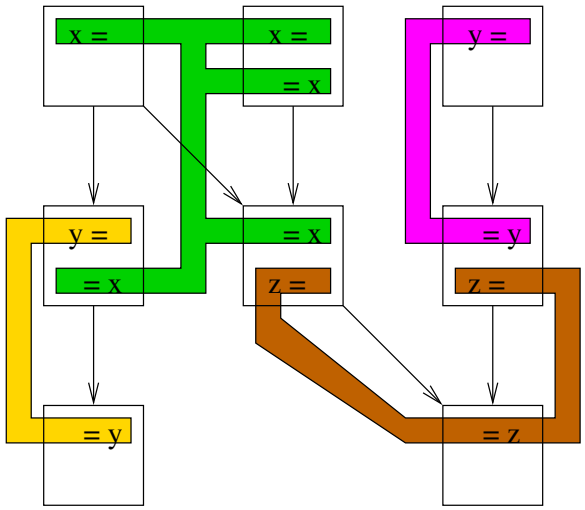
Example of Live Ranges (cont.)



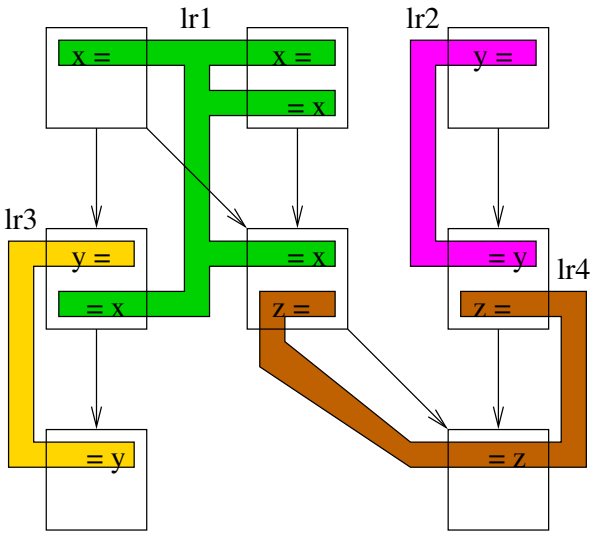
Example of Live Ranges (cont.)



Example of Live Ranges (cont.)



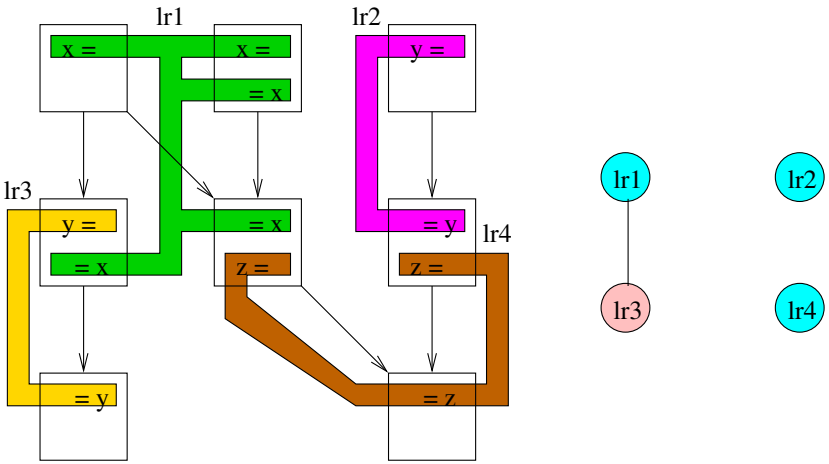
Example of Live Ranges (cont.)



Register Allocation by Graph Coloring

- A coloring of a graph is an assignment of a color to each node of the graph in such a manner that each two nodes connected by an edge do not have the same color.
- In register allocation, each node in the interference graph represents a live range that is a candidate for residing in a register.
- Two nodes in the interference graph are connected if their live ranges overlap (live ranges must reside in different registers).

Example of Live Ranges (cont.)



Register Allocation in VPO

- VPO allocates local variables and arguments to registers.
- Unlike register assignment in VPO, register allocation never results in spills.
- Register allocation in VPO can use all remaining available registers since it is performed after register assignment.

What Variables Can Be Allocated?

- Only scalar variables are candidates for register allocation.
- VPO does not allocate variables that are indirectly referenced.

```
    foo(&a);
=>
    r[8]=r[14]+a.;
    ST=HI[foo]+LO[foo];
```


Benefit Must Outweigh the Cost

- Cost of using a nonscratch registers on most machines is a save and a restore (2 memory references). Benefit would require more than 2 estimated references.
- Parameters have to be loaded from the stack (unless they are passed through a register). Allocating a stack parameter would require another memory reference.

Which Variables Are Allocated First?

- VPO allocates live ranges of variables first that have the greatest potential benefit.
- VPO uses a simple estimate of the frequency for each variable reference based on the loop nesting level in which the reference appears.
  - (loop_nesting_level « 4) + 1
- The estimate for a live range is the sum of the estimates for each reference of the variable within the live range.

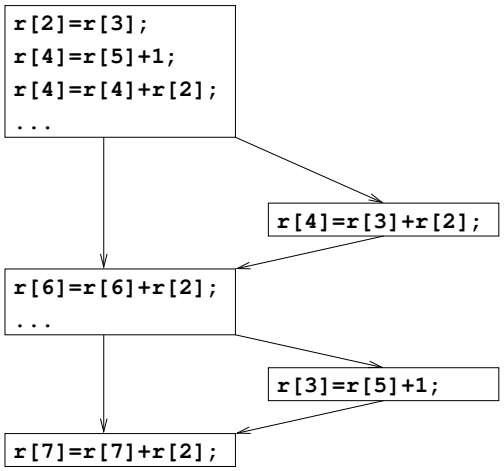
Copy Propagation

- Given an assignment $x \leftarrow y$, replace later uses of x with y as long as intervening instructions have not changed the value of either x or y.

```
r[2]=r[1];              r[2]=r[1];
r[3]=r[3]+r[2];         r[3]=r[3]+r[1];
r[2]=r[4]+r[2];   =>    r[2]=r[4]+r[1];
r[4]=r[4]+r[2];         r[4]=r[4]+r[2];
```

Global Copy Propagation

- Where can r[2] be replaced with r[3]?



Constant Propagation

- Given an assignment $x \leftarrow c$, where c is a constant, replace later uses of x with c as long as intervening instructions have not changed the value of x.

```
r[4]=1;                 r[4]=1;
r[3]=r[3]+r[4];         r[3]=r[3]+1;
r[2]=r[4]+2;       =>   r[2]=1+2;
r[4]=r[4]-r[2];         r[4]=1-r[2];
```


Dead Assignment Elimination

- An assignment is dead if the value assigned is never used.

```
r[2]=r[1];
r[3]=r[3]+r[2];
r[2]=r[4]+r[2];
r[4]=r[4]+r[2];
```

- after copy propagation

```
r[2]=r[1];
r[3]=r[3]+r[1];
r[2]=r[4]+r[1];
r[4]=r[4]+r[2];
```

- after dead assignment elimination

```
r[3]=r[3]+r[1];
r[2]=r[4]+r[1];
r[4]=r[4]+r[2];
```


Peephole Optimization

- Peephole optimization refers to a code-improving transformation that is typically performed after code generation.
- Individual peephole optimizations are often expressed as rules.
- This is accomplished by moving a small window or peephole across the generated code searching for segments of code for which the rules apply.

Peephole Optimization Rules

- A peephole optimization rule is typically expressed with the following parts:
  - matching pattern which can match one or more assembly instructions
  - semantic checks
  - replacement pattern which replaces these matched instructions

Example Peephole Optimization Rule

- Below is a rule to eliminate an unconditional jump by reversing a conditional branch.

```
"      b%0    L%1",
"      ba     L%2",
"L%1:"
invert(%0,%3)
=>
"      b%3    L%2",
"L%1:"
```


Peephole Optimization (cont.)

- Peephole optimization is a convenient method for dealing with a variety of special cases.
- One can quickly specify a number of rules that are appropriate for a particular architecture.
- The validity and completeness of such optimizations would always be a concern.

Common Subexpression Elimination

- Common subexpression elimination replaces code that recalculates an expression that is already currently available in a cheaper form.
- Most implementations work on machine-independent intermediate code (trees, triples, quads, etc) and not on machine instructions.

```
b = a[i];
...
v = i*4;
```

Common Subexpression Elimination in VPO

- VPO symbolically simulates register transfers and records the values that they store.
- When it encounters an instruction that recomputes an existing value and it appears to be beneficial to replace, it will replace the calculation with the earlier value.

Symbolic Expressions

- An s-expr is a symbolic expression. Two s-exprs *a* and *b* are said to match at a given point in the function, if and only if it is determined that *a* and *b* have the same value at that point.

```
r[1]=M[_a];
r[1]=r[1]+M[_b];
```

- After symbolic simulation of the second instruction, `r[1]` matches `M[_a]+M[_b]`.

Example of Local CSE

```
a = c;          r[1]=M[_c];
b = c + 1;  =>  M[_a]=r[1]; r[1]:
                r[2]=M[_c];
                ...
```

- After the first two instructions, we have the equivalence class (`M[_c]`, `r[1]`, `M[_a]`).
- After the 3rd instruction, we find the equivalence class that contains `M[_c]` and replace it with the cheapest member in the class, `r[1]`. So the 3rd instruction is updated as shown below and the death of `r[1]` is moved to the 3rd instruction.

```
                r[1]=M[_c];
            =>  M[_a]=r[1];
                r[2]=r[1];  r[1]:
                ...
```


Dead Assignment Elimination in VPO

- Dead assignment elimination eliminates useless assignments to a register or to memory.
- It is performed during the CSE phase in VPO since the CSE phase keeps track of the number of uses of a variable or a register after it is updated.

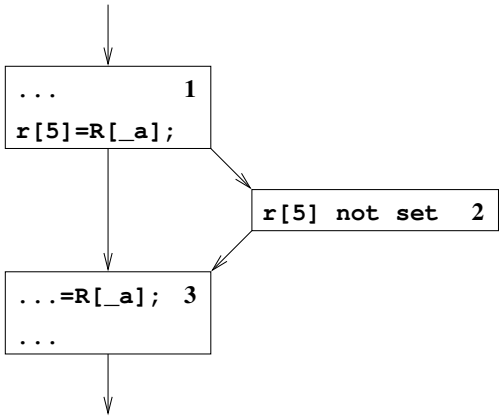
```
r[4]=r[5];          r[4]=r[5];
r[3]=r[4];      =>  r[3]=r[5];      =>  r[3]=r[5];
r[2]=r[4]; r[4]:    r[2]=r[5]; r[4]:    r[2]=r[5];
```

Global Common Subexpression Elimination

- Before operating on a basic block, the CSE phase in VPO operates on all predecessors of that block.
- It merges (intersects) e-lists together from the predecessors and then starts to perform CSE on the current block.

Global Common Subexpression Elimination (cont.)

- By merging e-lists (expression lists) together from both predecessors, `R[_a]` in block 3 can be replaced by `r[5]`.



Global Common Subexpression Elimination (cont.)

- How can VPO perform CSE on loops if it has to operate on all predecessors of a block first?
- VPO makes an initial pass through the loop (dryrun) starting at the header of the loop with an empty e-list.
- Once the initial pass is complete, CSE operates on the loop one more time, and merges the equivalence lists of the predecessors of the header beforehand.