COMP 512
Rice University
Spring 2015

# *Overview of Optimization, Part I*

## *Local Value Numbering, Terminology, EBBs*

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved

## Optimization

**Content from § 8 EaC2e**

Compilers operate at many ***granularities*** or ***scopes***

- Local techniques
  - ♦ Work on a single basic block
  - ♦ Maximal length sequence of straight-line code
- Regional techniques
  - ♦ Consider multiple blocks, but less than whole procedure
  - ♦ Single loop, loop nest, dominator region, …
- Intraprocedural (or global) techniques
  - ♦ Operate on an entire procedure (*but just one*)
  - ♦ Common unit of compilation
- Interprocedural (or whole-program) techniques
  - ♦ Operate on > 1 procedure, up to whole program
  - ♦ Logisitical issues related to accessing the code (*link time?*)

## Optimization

At each of these scopes, the compiler uses different graphs

- Local techniques
  - ♦ Dependence graph (*instruction scheduling*)
- Regional Techniques
  - ♦ Control-flow graph (*natural loops*)
  - ♦ Dominator tree
- Intraprocedural (or global) techniques
  - ♦ Control-flow graph
  - ♦ Def-use chains, sparse evaluation graphs, SSA as graph
- Interprocedural (or whole-program) techniques
  - ♦ Call (multi) graph

*Compiler writers must be able to perform graph traversals in their sleep – preorder, postorder, reverse postorder, depth first, …*

## Optimization

At each of these scopes, the compiler uses different kinds of techniques

- Local techniques
  - ♦ Simple walks of the block
- Regional Techniques
  - ♦ Find a way to treat multiple blocks as a single block (*EBBs, dominators*)
  - ♦ Work with an entire loop nest
- Intraprocedural (or global) techniques
  - ♦ Data-flow analysis to determine safety and opportunity
  - ♦ Separate transformation phase to rewrite the code
- Interprocedural (or whole-program) techniques
  - ♦ Need a compilation framework where optimizer can see all the relevant code
  - ♦ Sometimes, limit to all procedures in a file
  - ♦ Sometimes, perform optimization at link time

# Optimization

We need to differentiate between analysis and transformation

- Analysis reasons about the code's behavior
- Transformation rewrites the code to change its behavior

Local techniques can interleave analysis and transformation

- Property of basic block: operations execute in defined order

Over larger regions, the compiler typically must complete its analysis before it transforms the code

- Analysis must consider all possible paths, including cycles
  - ♦ Cycles typically force compiler into offline analysis
- Leads to confusion in terminology between "optimization", "analysis", and "transformation"

# Optimization

## Terminology

*Optimization*

- We will use “optimization” to refer to a broad technique or strategy, such as code motion or dead code elimination

*Transformation*

- We will use “transformation” to refer to algorithms & techniques that rewrite the code being compiled

*Analysis*

- We will use the term “analysis” to refer to algorithms & techniques that derive information about the code being compiled

This subtle distinction in usage was suggested by Vivek Sarkar.

## Redundancy Elimination as an Example

**Covered in 412**

*An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.*

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

- Proving that x+y is redundant
- Rewriting the code to eliminate the redundant evaluation

One single-pass technique for accomplishing both is called value numbering

## Value Numbering — An Old Idea

### The key notion (Balke 1967 or Ershov 1954)

- Assign an identifying number, V(n), to each expression
  - ♦ V(x+y) = V(j) iff x+y and j always have the same value
  - ♦ Use hashing over value numbers to make it efficient
- Use the value numbers to "improve" the code

*V(n) is n's "value number"*

Improving the code

- Replace redundant expressions
- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them

- This technique was invented for low-level, linear IRS
- Equivalent methods exist for trees (*build a DAG, § 8.3.1 in EaC**1e***)

# Local Value Numbering

Local ⇒ one block at a time
Block ⇒ straight-line code

## The algorithm

For each operation *o* in the block

1 Get value numbers for the operands from a hash lookup

2 Hash <operator,VN($o_1$),VN($o_2$)> to get a value number for *o*

3 If *o* already had a value number, replace *o* with a reference

4 If $o_1$ & $o_2$ are constant, evaluate it & use a "load immediate"

If hashing behaves, the algorithm runs in linear time

- ♦ If you don't believe in hashing, try multi-set discrimination

Minor issues

- Commutative operator ⇒ hash operands in each order *or* sort the operands by VN before hashing (*either works, sorting is cheaper*)
- Looks at operand's value number, not its name

EaC2e: digression on page 256 or reference [65]

# Local Value Numbering

## An Example

| Original Code | With VNs | Rewritten |
|---|---|---|
| $a \leftarrow x + y$ | $a^3 \leftarrow x^1 + y^2$ | $a^3 \leftarrow x^1 + y^2$ |
| * $b \leftarrow x + y$ | * $b^3 \leftarrow x^1 + y^2$ | * $b^3 \leftarrow a^3$ |
| $a \leftarrow 17$ | $a^4 \leftarrow 17$ | $a^4 \leftarrow 17$ |
| * $c \leftarrow x + y$ | * $c^3 \leftarrow x^1 + y^2$ | * $c^3 \leftarrow a^3$ (oops!) |

Two redundancies:

- Eliminate stmts with a *
- Coalesce results ?

Options:

- Use $c^3 \leftarrow b^3$
- Save $a^3$ in $t^3$
- Rename around it

$a \leftarrow 17$ **kills** the copy of value 3 in a

# Local Value Numbering

## Example (continued)

| Original Code | With VNs | Rewritten |
|---|---|---|
| $a_0 \leftarrow x_0 + y_0$ | $a_0^3 \leftarrow x_0^1 + y_0^2$ | $a_0^3 \leftarrow x_0^1 + y_0^2$ |
| * $b_0 \leftarrow x_0 + y_0$ | * $b_0^3 \leftarrow x_0^1 + y_0^2$ | * $b_0^3 \leftarrow a_0^3$ |
| $a_1 \leftarrow 17$ | $a_1^4 \leftarrow 17$ | $a_1^4 \leftarrow 17$ |
| * $c_0 \leftarrow x_0 + y_0$ | * $c_0^3 \leftarrow x_0^1 + y_0^2$ | * $c_0^3 \leftarrow a_0^3$ |

Renaming:
- Give each value a unique name
- Makes it clear

Notation:
- While complex, the meaning is clear

Result:
- $a_0^3$ is available
- rewriting works

*

## Simple Extensions to Value Numbering

### Constant folding

- Add a field to the table that records when a value is constant
- Evaluate constant values at compile-time
- Replace with load immediate or immediate operand
- No stronger local algorithm

### Algebraic identities

- Must check (many) special cases
- Replace result with input VN
- Build a decision tree on operation

Identities: (Click)

x←y, x+0, x-0, x*1, x÷1, x-x, x*0, x÷x, x∨0, x ∧ 0xFF…FF, max(x,MAXINT), min(x,MININT), max(x,x), min(y,y),

and so on ...

(*over values, not names*)

## Optimization

*In discussing any optimization, we look for three issues*

**Safety** – *Does it change the results of the computation?*

- Safety is proven with results of analysis
- Data-flow analysis or other special case analysis

**Profitability** – *Is it expected to speed up execution?*

- Many authors assume transformations are always profitable
- Use either heuristics or a careful algebra of costs

**Opportunity** – *Can we efficiently locate places to apply it?*

- Can we find find all the places where the transformation works?
- Do we need to update safety information afterward?

## Safety

The first principle of optimization

***The compiler must preserve the code's "meaning"***

When can the compiler transform the code?

- Original & transformed code must have the same final state
- Variables that are visible at exit
- Equality of result, not equality of method (*ignore temporaries*)

Formal notion

*For two expressions, M and N, we say that M and N are* <u>*observationally equivalent*</u> *if and only if, in any context C where both M and N are closed (that is, have no free variables), evaluating C[M] and C[N] either produces identical results or neither terminates.* Plotkin, 1975

$\Rightarrow$ Different translations with identical results are fine

## Safety

In practice, compilers use a simpler notion of equivalence

> ***If, in their actual program context, the result of evaluating e' cannot be distinguished from the result of evaluating e, the compiler can substitute e' for e.***

- This restatement ignores divergence
- If e' is faster than e, the transformation is profitable

**Equivalence and context**

- Compiled code always executes in some context
- Optimization is the art of capitalizing on context
- Lack of context ⇒ fully general (*i.e.*, slow) code

**Some compilers employ a worse standard** (FORTRAN)

- Correct behavior for "standard conforming" code
- Undefined behavior for other code

## Safety

### My favorite bad quote on safety

> *You, as a compiler writer, must decide if it's worth the risk of doing this kind of optimization. It's difficult for the compiler to distinguish between the safe and dangerous cases, here. For example, many C compilers perform risky optimizations because the compiler writer has assumed that a C programmer can understand the problems and take steps to remedy them at the source code level. It's better to provide the maximum optimization, even if it's dangerous, than to be conservative at the cost of less efficient code. A Pascal programmer may not have the same level of sophistication as a C programmer, however, so the better choice in this situation might be to avoid the risky optimization entirely or to require a special command-line switch to enable the optimization.*

Allen Holub, *Compiler Design in C,* Prentice Hall, 1990, p. 677

The point

- You must not violate the first principle
- Without the first principle, just compile a return and be done

This statement was gone by the 2nd Edition.

# Safety & Value Numbering

## Why is <u>local value numbering</u> safe?

Critical property of a basic block:
If any statement executes, they all execute, in a predetermined order

- Hash table starts empty
- Expressions placed in table as processed
- If <operator,VN($o_1$),VN($o_2$)> is in the table, then
  - ♦ It has already occurred <u>at least once</u> in the block
  - ♦ Neither $o_1$ nor $o_2$ have been subsequently redefined
    - → The mapping uses VN($o_1$) and VN($o_2$), not $o_1$ and $o_2$
    - → If one was redefined, it would have a new VN

If <operator,VN($o_1$),VN($o_2$)> has a VN, the compiler can safely use it

- Algorithm incrementally constructs a proof that <operator,VN($o_1$),VN($o_2$)> is redundant
- Algorithm modifies the code, but does not invalidate the table

# Profitability

**The compiler should only transform the code when it helps!**

- Eliminating one or more operations
- Replacing an operation with a cheaper one
- Moving an operation to a place where it will execute fewer times

Sometimes, we can prove profitability

- ♦ Fold a constant expression into an immediate operation

Sometimes, we must guess

- ♦ Eliminating a redundant operation in a loop

Sometimes, we cannot tell …

- ♦ Inlining in a Fortran compiler

We should know when we cannot tell if some transformation is profitable !

Compiler writers need to think explicitly about profitability ...

# Profitability & Value Numbering

**When is local value numbering profitable?**

- If reuse is cheaper than re-computation
  - ♦ Does not cause a spill or a copy (*hard to determine*)
  - ♦ In practice, assumed to be true
- Local constant folding is always profitable
  - ♦ Re-computing uses a register, as does load immediate
  - ♦ Immediate form of operation avoids even that cost
- Algebraic identities
  - ♦ If it eliminates an operation, it is profitable (x + 0)
  - ♦ Profitability of simplification depends on target ($2x \Rightarrow x+x$)
  - ♦ Easy to factor target machine costs into the implementation
    - → ***don't apply it unless it is profitable!***

## Opportunity

To perform an optimization, the compiler must locate all the places in the code where it can be applied

- Allows compiler to evaluate each possible application
- Leads to efficient application of the transformation
- Avoids additional search

Approaches

- Perform analysis to find opportunities
  - ♦ VERYBUSY expressions & code hoisting
- Look at every operation
  - ♦ Value numbering, loop invariant code motion
- Iterate over subset of the IR
  - ♦ Operator strength reduction on SSA

# Opportunity & Value Numbering

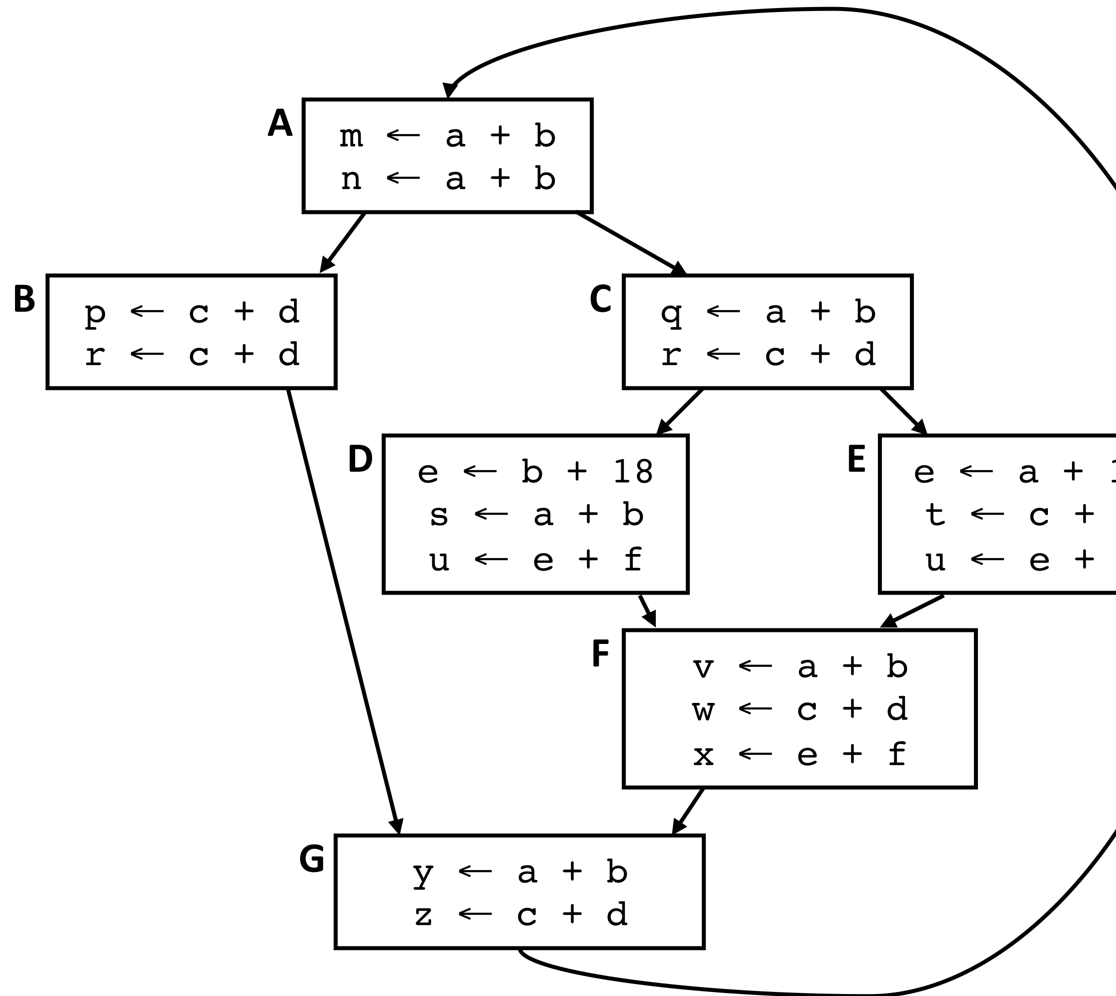
**How does local value numbering find opportunity?**

- Linear scan over block, in execution order
- Constructs a model of program state
- At each operation, check for several opportunities

Summary

- It performs an exhaustive search of the opportunities
- This answer is not satisfying, but it is true
  - ♦ Must limit cost of checking each operation
  - ♦ For example, use a tree of algebraic identities by operator
- Hashing keeps cost down to O(1) per operand + per operation

# A Multi-Block Example



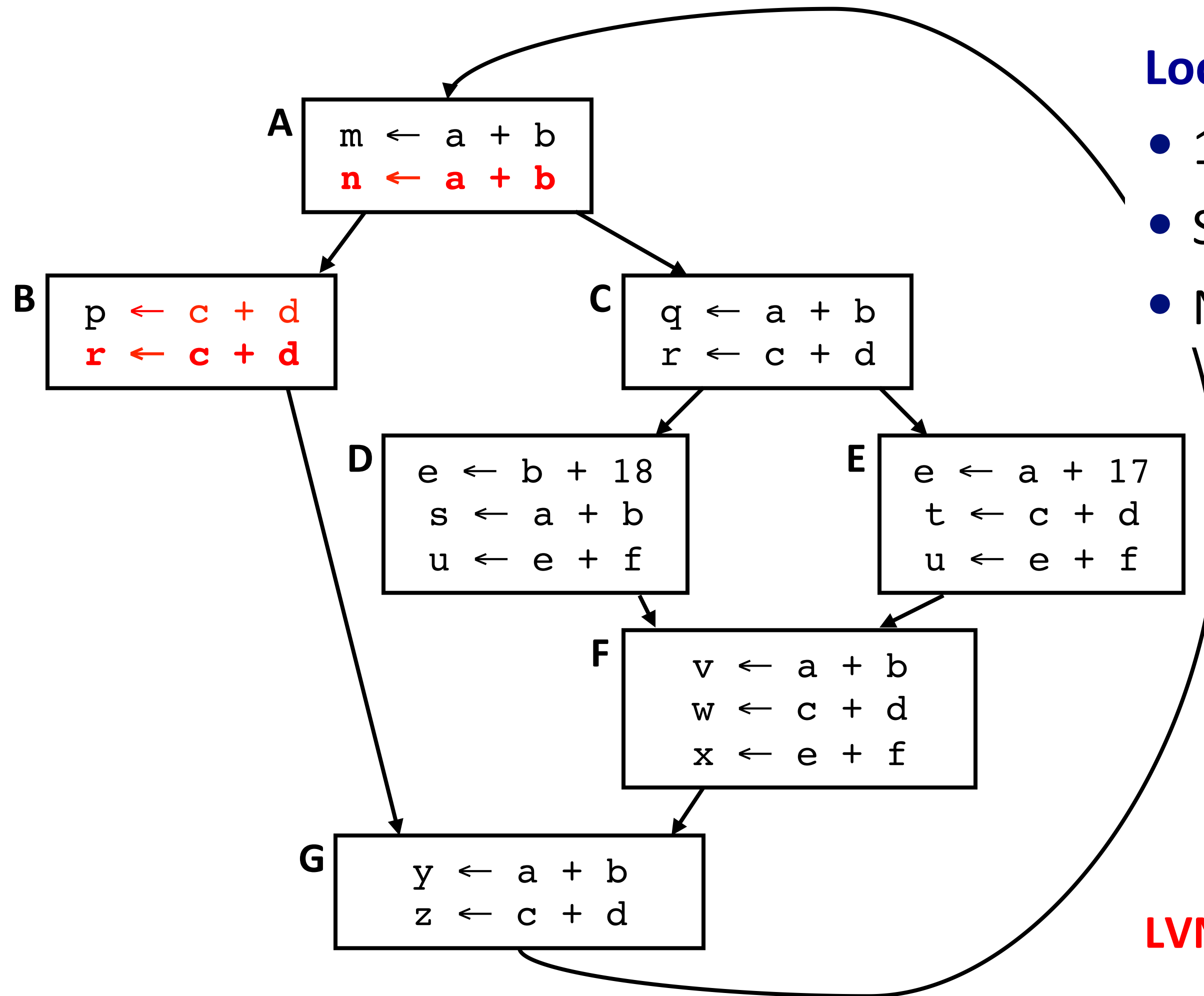

**Control-flow graph (CFG)**

- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

**G = (N,E)**

- **N** = {A, B, C, D, E, F, G}
- **E** = { (A,B), (A,C), (B,G),( C,D), (C,E), (D,F), (E,F), (F,E) }
- |**N**| = 7, |**E**| = 8

# A Multi-Block Example

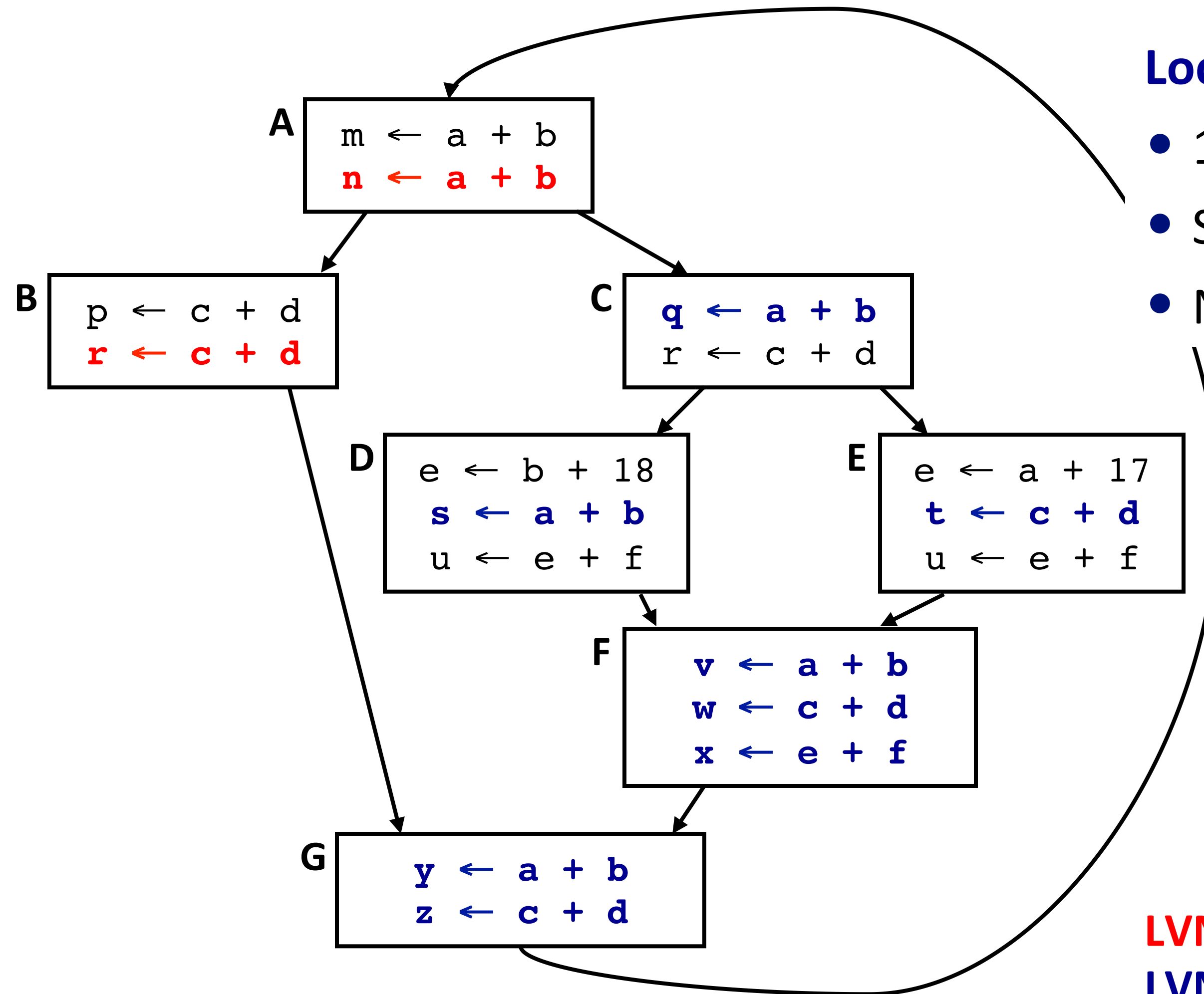



**Local Value Numbering (LVN)**

- 1 block at a time
- Strong local results
- No inter-block effects

**LVN finds redundant ops in red**

# A Multi-Block Example

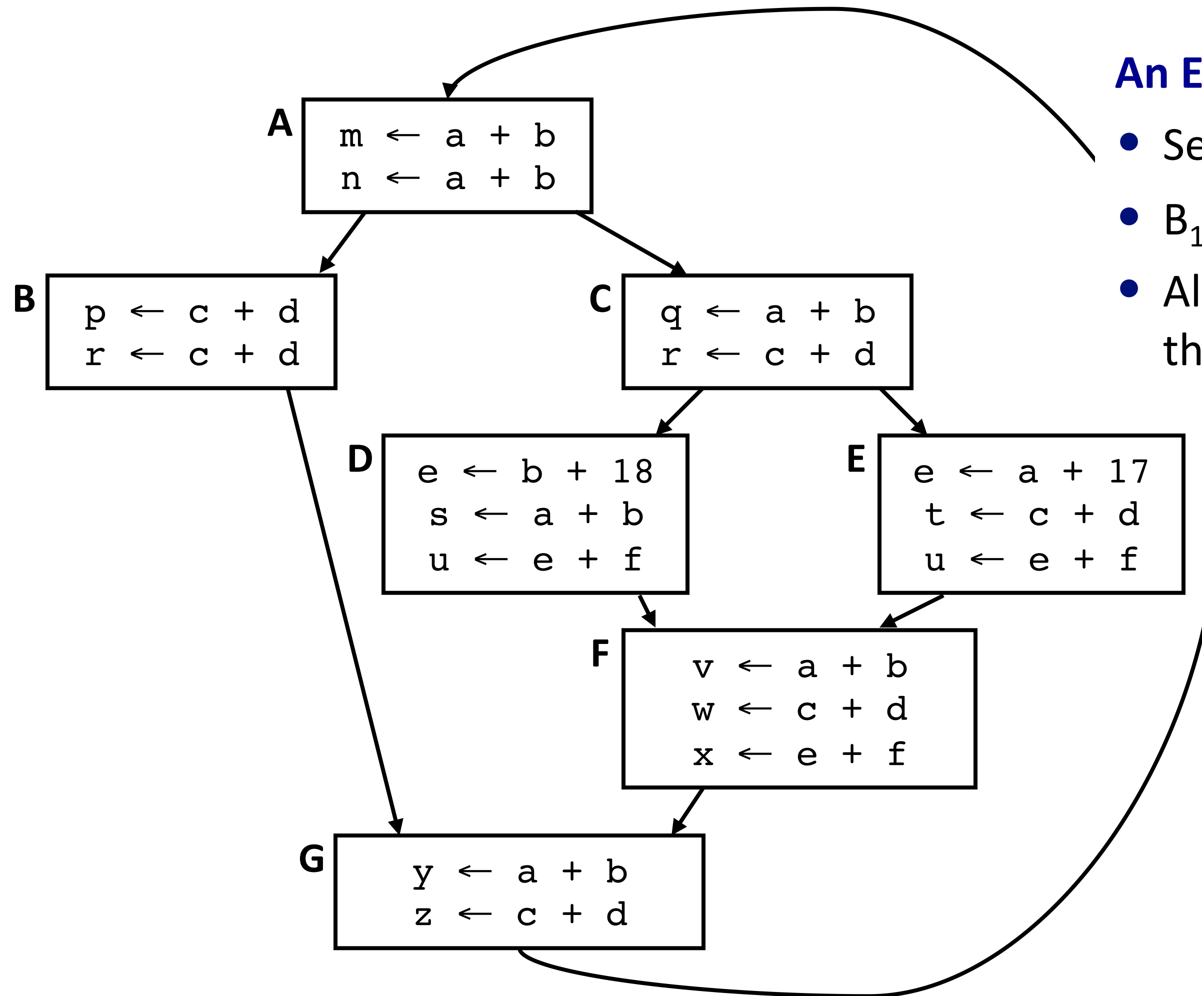



**Local Value Numbering (LVN)**

- 1 block at a time
- Strong local results
- No inter-block effects

**LVN finds redundant ops in red**
**LVN misses redundant ops in blue**

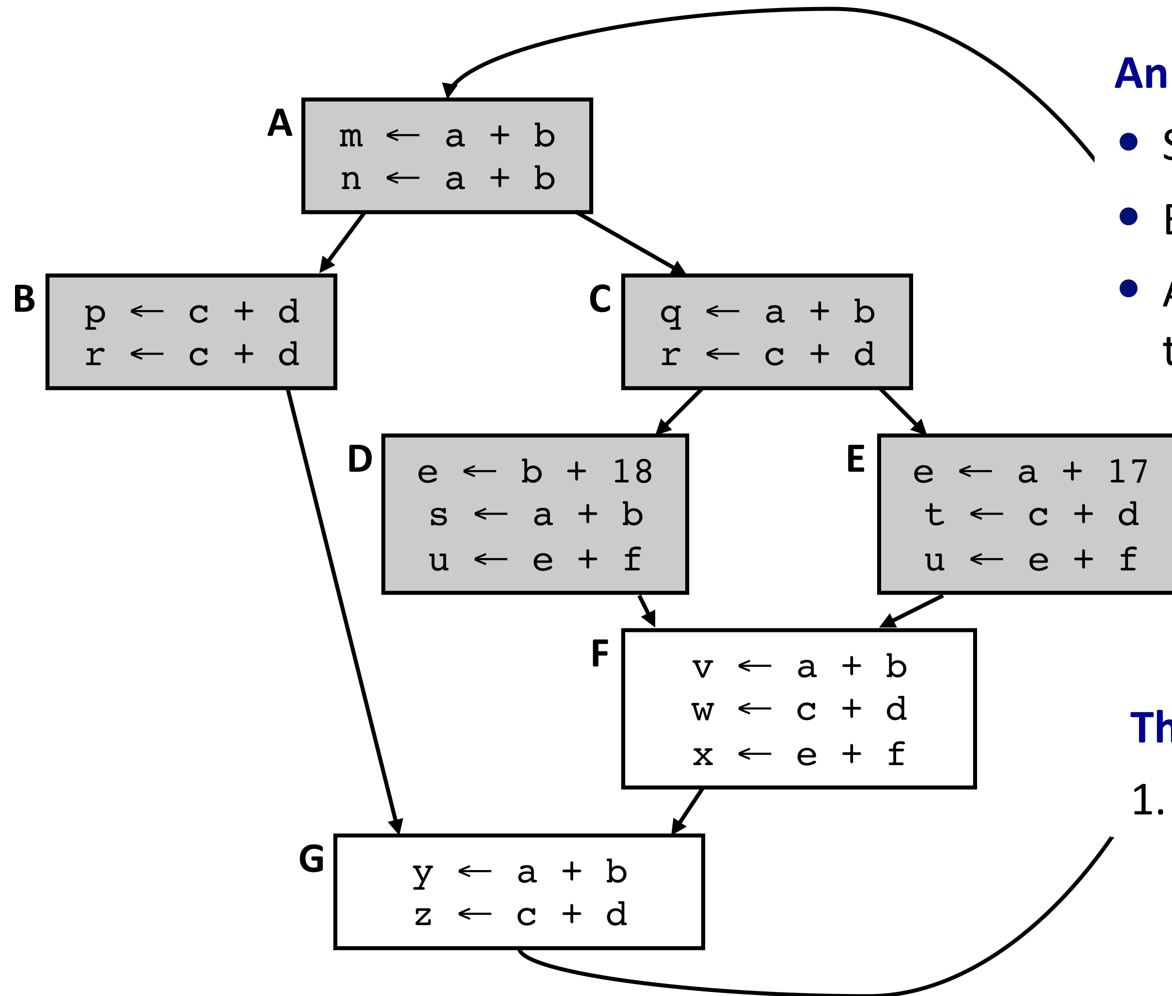
# Beyond Basic Blocks: Extended Basic Blocks




### An Extended Basic Block (EBB)

- Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

# Extended Basic Blocks



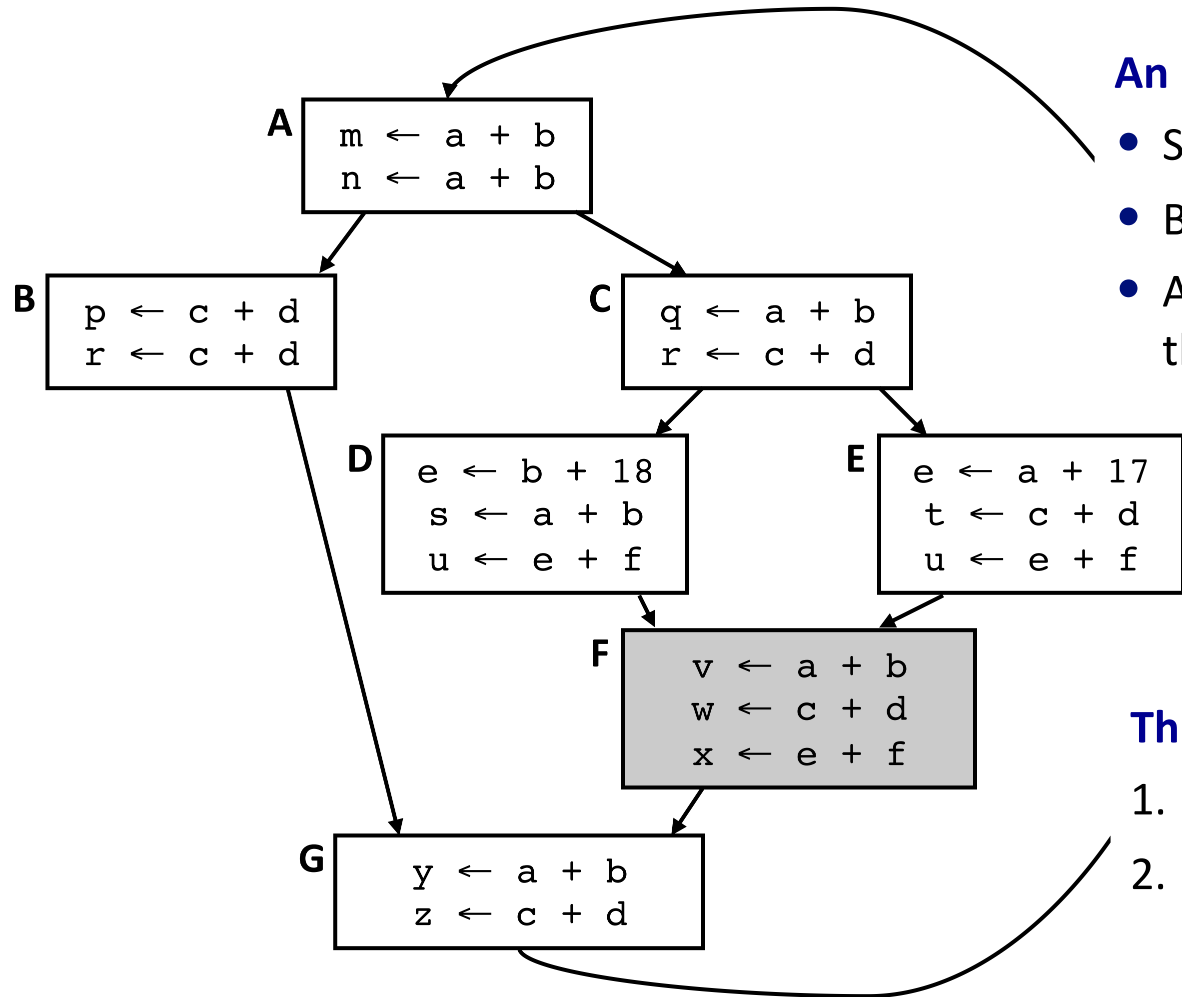

## An Extended Basic Block (EBB)

- Set of blocks $B_1$, $B_2$, …, $B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

## Three EBBs in this CFG

1. {A, B, C, D, E}

# Extended Basic Blocks



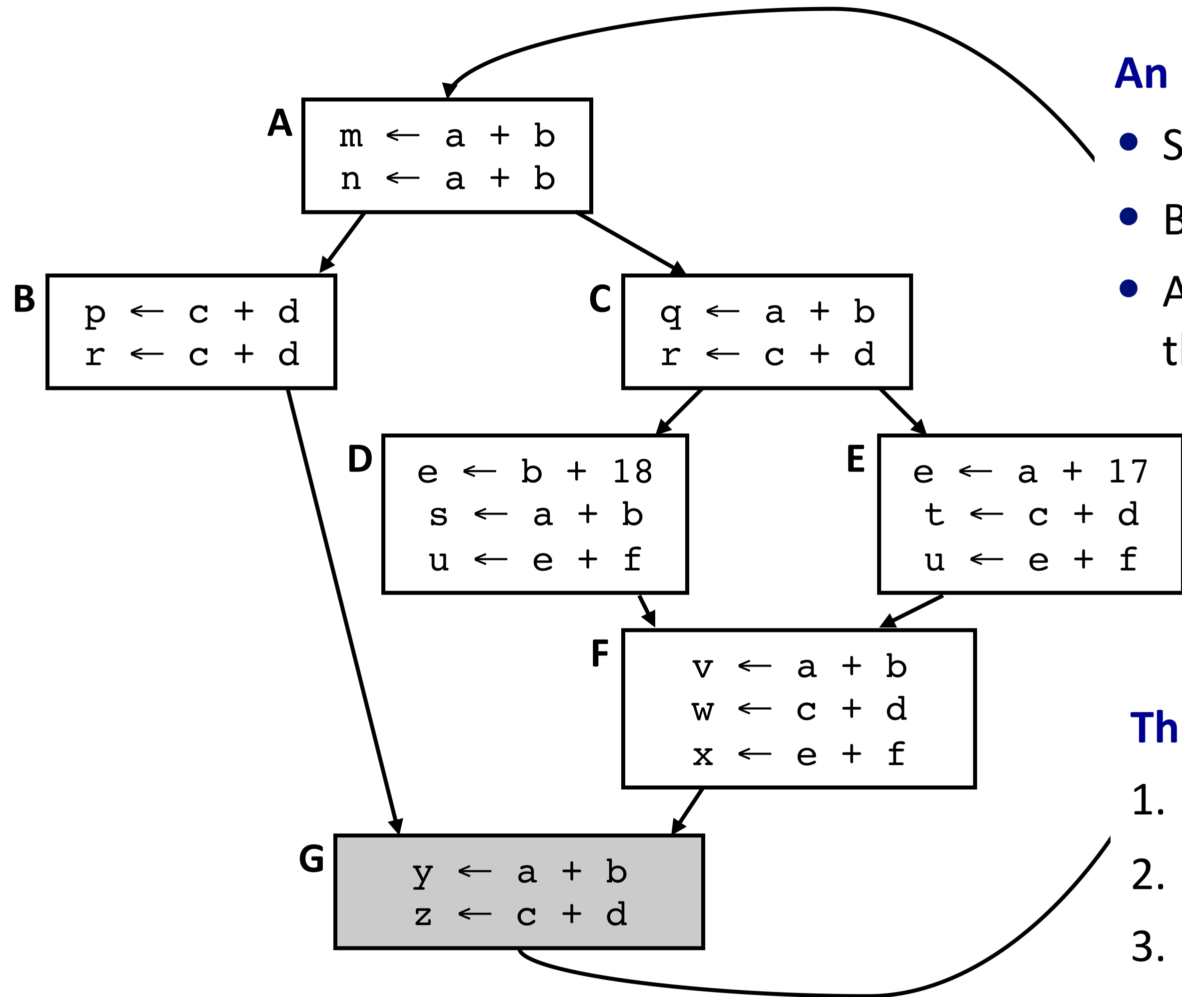

## An Extended Basic Block (EBB)

- Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

## Three EBBs in this CFG

1. { A, B, C, D, E }
2. { F }

# Extended Basic Blocks




**An Extended Basic Block (EBB)**

- Set of blocks $B_1, B_2, \ldots, B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

**Three EBBs in this CFG**

1. { A, B, C, D, E }
2. { F }
3. { G }

# Extended Basic Blocks

**A**
```
m ← a + b
n ← a + b
```

**B**
```
p ← c + d
r ← c + d
```

**C**
```
q ← a + b
r ← c + d
```

**D**
```
e ← b + 18
s ← a + b
u ← e + f
```

**E**
```
e ← a + 17
t ← c + d
u ← e + f
```

**F**
```
v ← a + b
w ← c + d
x ← e + f
```

**G**
```
y ← a + b
z ← c + d
```

**An Extended Basic Block (EBB)**

- Set of blocks $B_1$, $B_2$, …, $B_n$
- $B_1$ has > 1 predecessor
- All other $B_i$ have 1 pred. & that pred. is in the **EBB**

**Three EBBs in this CFG**

1. { A, B, C, D, E }
2. { F }
3. { G }

**Degenerate or trivial EBBs**