TDDD55 Compilers and Interpreters

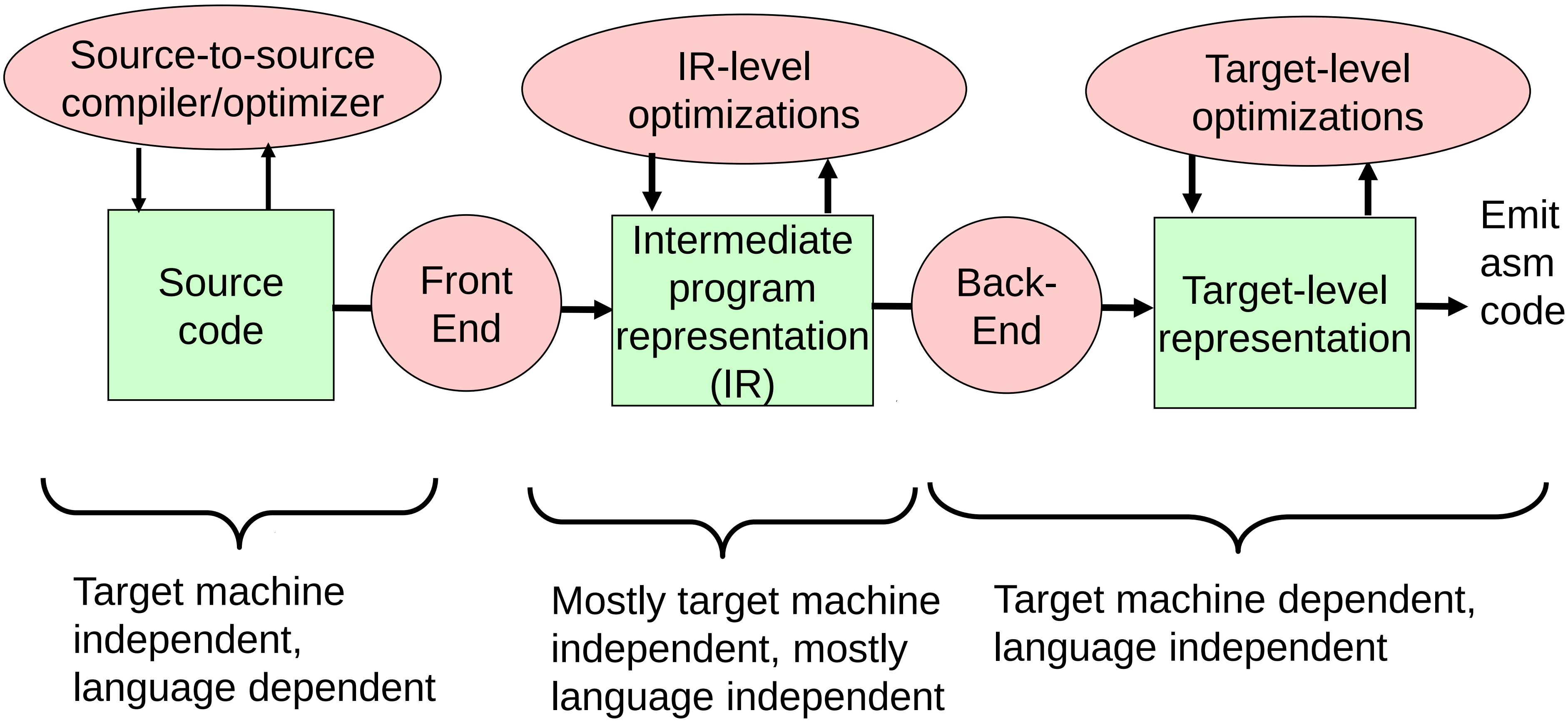
TDDB44 Compiler Construction

Code Optimization

P. Fritzson, C. Kessler, M. Sjölund, A. Pop
IDA, Linköpings universitet, 2023.

Code Optimization – Overview

Goal: Faster code and/or smaller code and/or low energy consumption



Remarks

- Often multiple levels of IR:
  - high-level IR (e.g. abstract syntax tree AST),
  - medium-level IR (e.g. quadruples, basic block graph),
  - low-level IR (e.g. directed acyclic graphs, DAGs)

  → do optimization at most appropriate level of abstraction

  → code generation is continuous lowering of the IR towards target code

- "Postpass optimization":
  done on *binary code* (after compilation or without compiling)

Disadvantages of Compiler Optimizations

- Debugging made difficult
  - Code moves around or disappears
  - Important to be able to switch off optimization
  - Note: Some compilers have **–Og** optimization level to avoid optimization that makes debugging hard
- Increases compilation time
- May even affect program semantics
  - **A = B*C – D + E** è **A = B*C + E – D**
    may lead to overflow if **B*C+E** is too large

Optimization at Different Levels of Program Representation

- **Source-level optimization**
  - Made on the source program (text)
  - Independent of target machine
- **Intermediate code optimization**
  - Made on the intermediate code (e.g., on AST trees, quadruples)
  - Mostly target machine independent
- **Target-level code optimization**
  - Made on the target machine code
  - Target machine dependent

Source-level Optimization

At source code level, independent of target machine

- Replace a slow algorithm with a quicker one,
  e.g. Bubble sort è Quick sort
- Poor algorithms are the main source of inefficiency but is difficult to automatically optimize
- Needs pattern matching, e.g. [K.'96] [di Martino, K. 2000]

Intermediate Code Optimization

At the intermediate code (e.g., trees, quadruples) level.

In most cases is target machine independent

- Local optimizations within basic blocks (e.g. common subexpression elimination)
- Loop optimizations (e.g. loop interchange to improve data locality)
- Global optimization (e.g. code motion, within procedures)
- Interprocedural optimization (between procedures)

Target-level Code Optimization

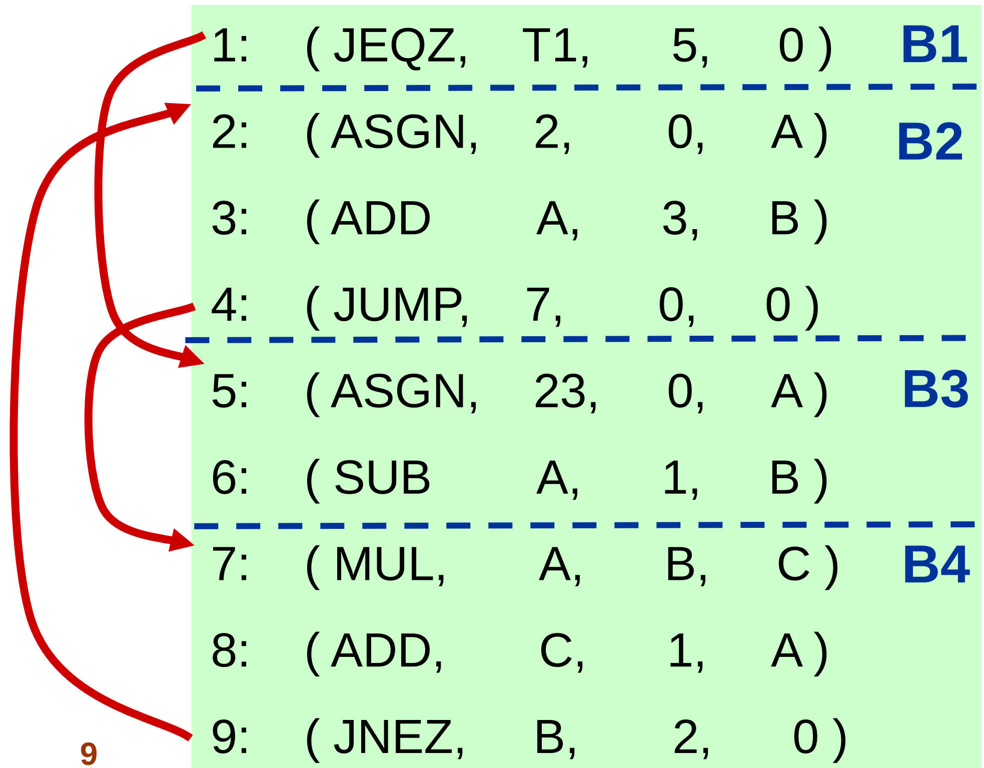
At the target machine binary code level.

Dependent on the target machine

- Instruction selection, register allocation, instruction scheduling, branch prediction
- Peephole optimization

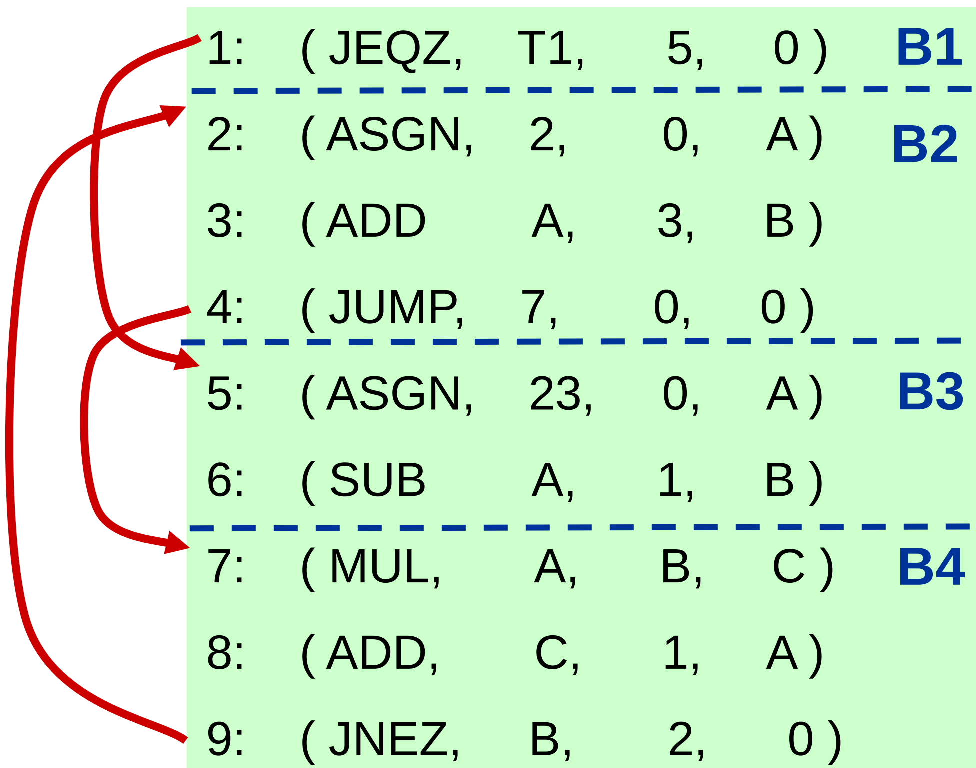
Basic Block

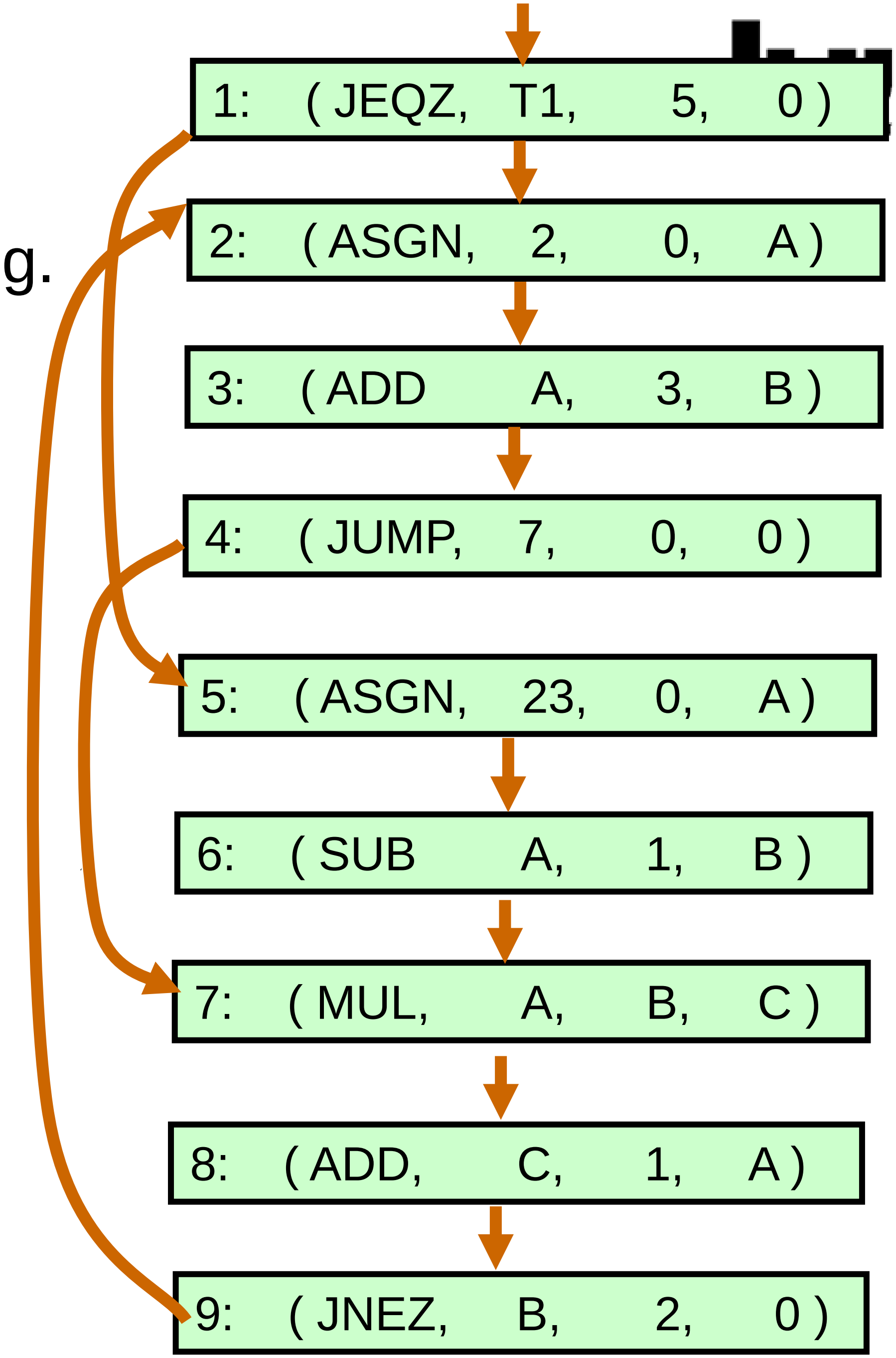
- A **basic block** is a sequence of textually consecutive operations (e.g. quadruples) that contains no branches (except perhaps its last operation) and no branch targets (except perhaps its first operation).
  - Always executed in same order from entry to exit
  - A.k.a. *straight-line code*



Control Flow Graph

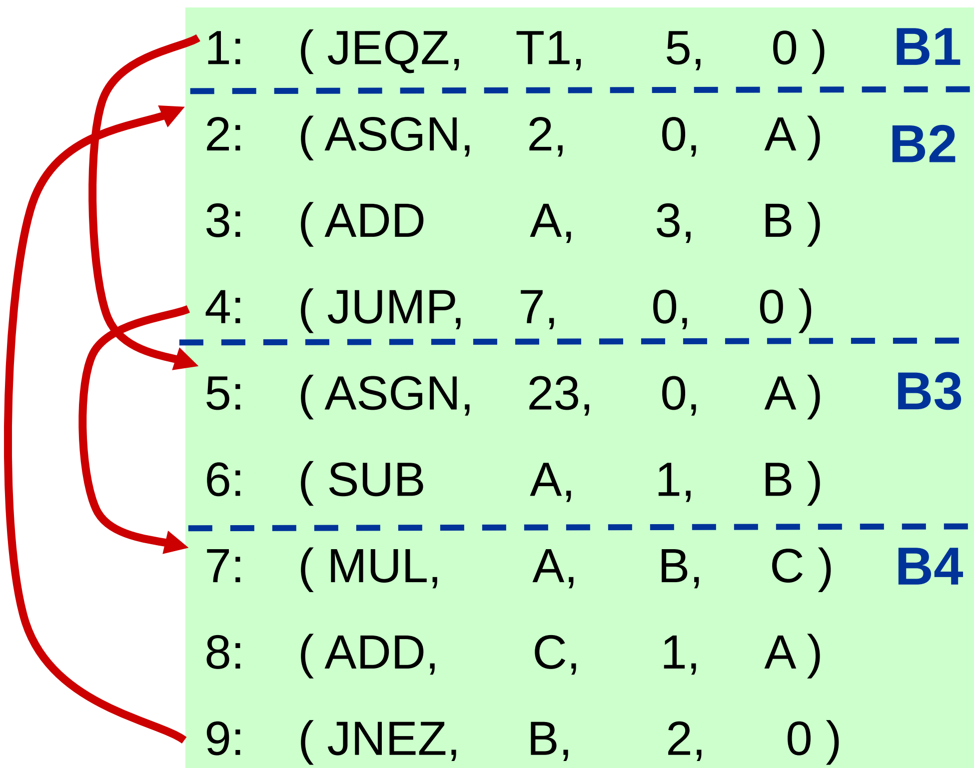
- Nodes: primitive operations (e.g. quadruples), or basic blocks.
- Edges: control flow transitions

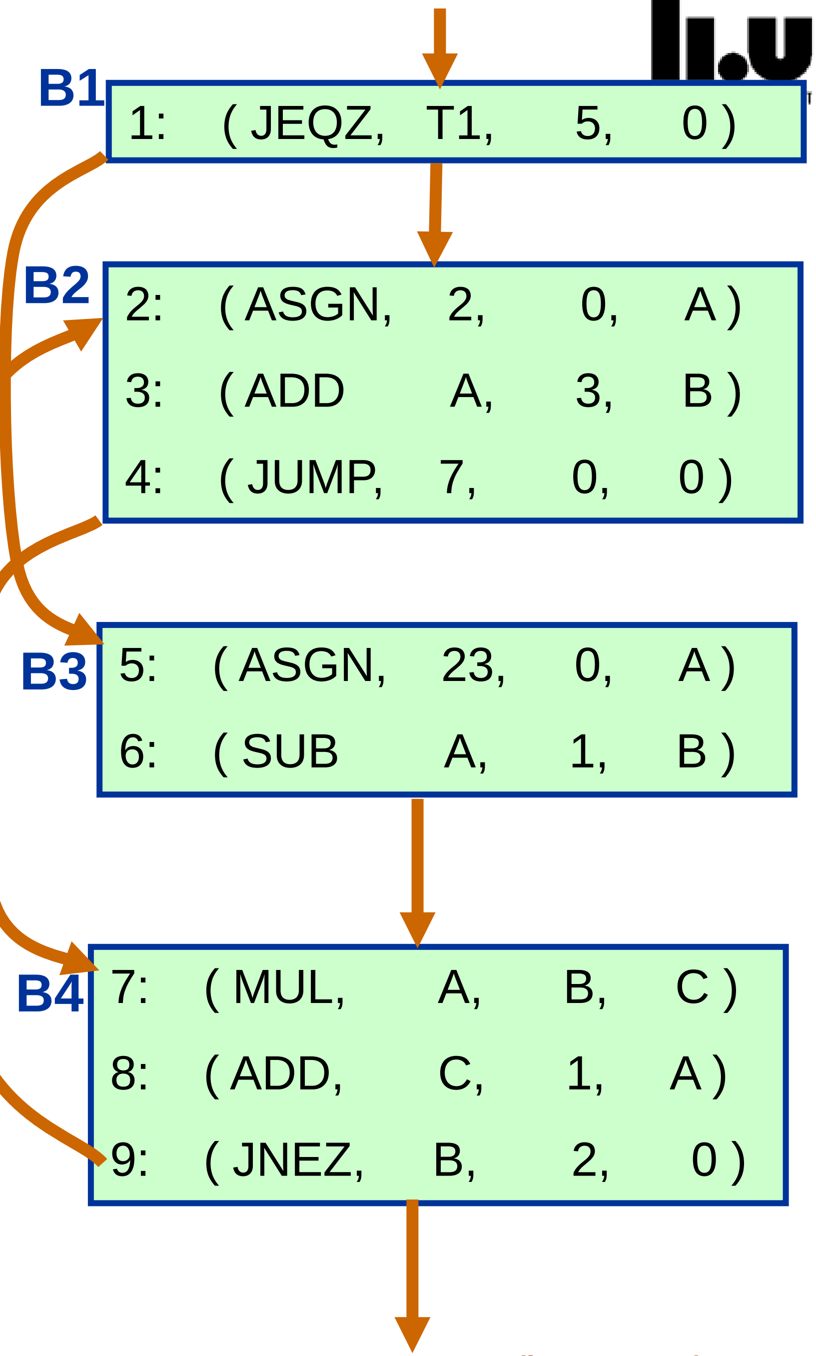




Basic Block
Control Flow Graph

- Nodes: basic blocks
- Edges: control flow transitions





Local Optimization

(within single Basic Block)

Local Optimization

- Within a single basic block
  - Needs no information about other blocks
- Example: **Constant folding** (Constant propagation)
  - Compute constant expressions at compile time

```
const int NN = 4;
…
i = 2 + NN;
j = i * 5 + a;
```

```
const int NN = 4;
…
i = 6;
j = 30 + a;
```

Local Optimization (cont.)

- Elimination of common subexpressions

```
A[ i+1 ]  =  B[ i+1 ];
```

→

```
tmp = i+1;
A[ tmp ]  =  B[ tmp ];
```

```
D = D + C * B;
A = D + C * B;
```

→

```
T = C * B;
D = D + T;
A = D + T;
```

NB: Redefinition of D
à D+T is *not* a common subexpression! (does not refer to the same *value*)

Common subexpression elimination builds **DAGs** (**directed acyclic graphs**) from expression trees and forests

Local Optimization (cont.)

- Reduction in operator strength
  - Replace an expensive operation by a cheaper one (on the given target machine)

Examples:

```
x = y ^ 2.0;          à   x = y * y;
x = 2.0 * y;          à   x = y + y;
x = 8 * y;            à   x = y << 3;

(S1+S2).length()  à   S1.length() + S2.length()
```

Some Other Machine-Independent Optimizations

q Array-references

  § `C = A[I,J] + A[I,J+1]`

  § Elements are beside each other in memory. Ought to be ''*give me the next element*''.

q Inline expansion of code for small routines

  § `x = sqr(y)     Þ     x = y * y`

q Short-circuit evaluation of tests

  § `while (a > b) and (c-b < k) and ...`

  § If **false** the rest does not need to be evaluated if they do not contain side effects (or if the language demands it for this op)

More examples of machine-independent optimization

- See for example the OpenModelica Compiler (https://github.com/OpenModelica/OpenModelica/blob/master/OMCompiler/Compiler/FrontEnd/ExpressionSimplify.mo) optimizing abstract syntax trees

```
// listAppend(e1,{}) => e1 is O(1) instead of O(len(e1))
case DAE.CALL(path=Absyn.IDENT("listAppend"),
              expLst={e1,DAE.LIST(valList={})})
     then e1;
// atan2(y,0) = sign(y)*pi/2
case (DAE.CALL(path=Absyn.IDENT("atan2"),expLst={e1,e2}))
guard Expression.isZero(e2)
algorithm
  e := Expression.makePureBuiltinCall(sign", {e1}, DAE.T_REAL_DEFAULT);
then DAE.BINARY(
  DAE.RCONST(1.570796326794896619231321691639751442),
  DAE.MUL(DAE.T_REAL_DEFAULT),
  e);
```

Exercise 1:
Draw a basic block control flow graph (BB CFG)

Loop Optimization

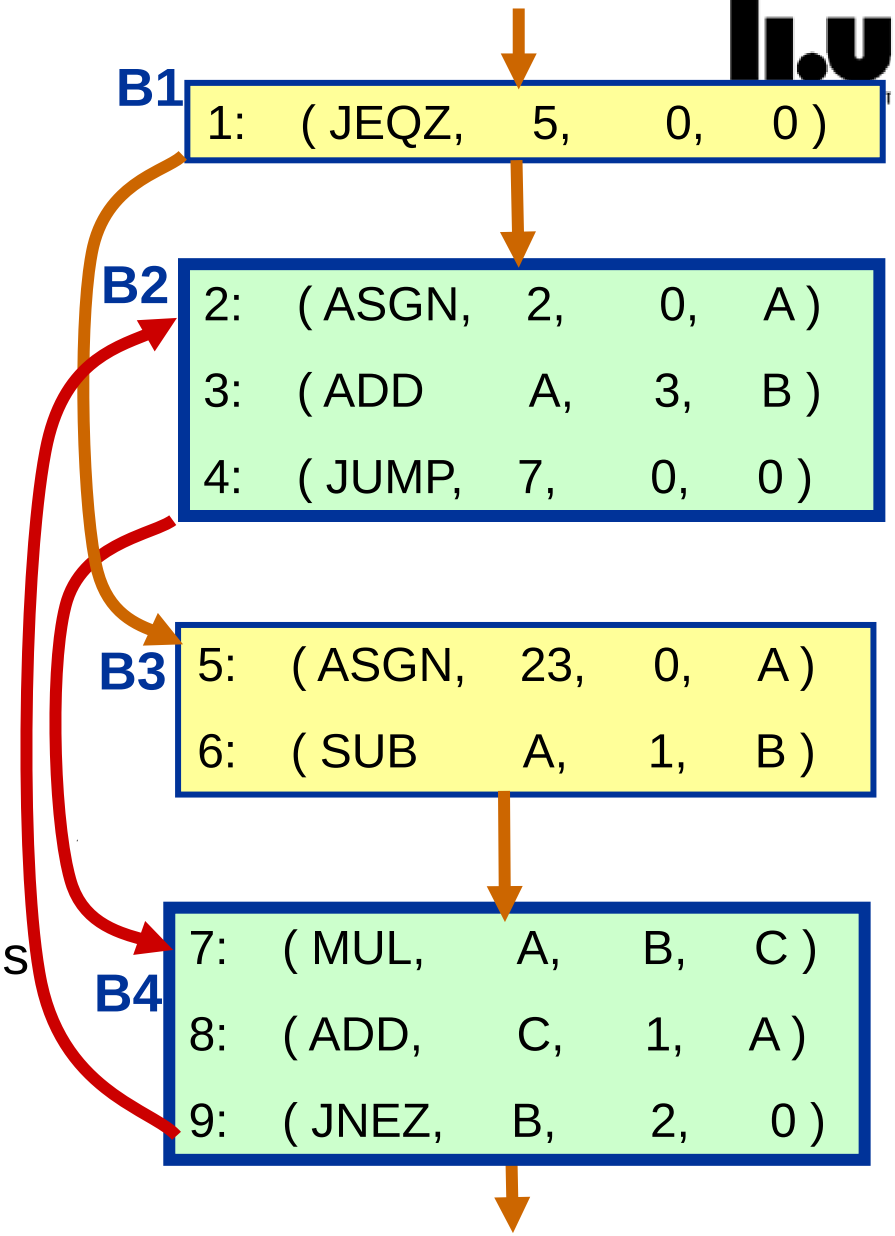
Loop Optimization

Minimize time spent in a loop

- q Time of loop body
- q Data locality
- q Loop control overhead

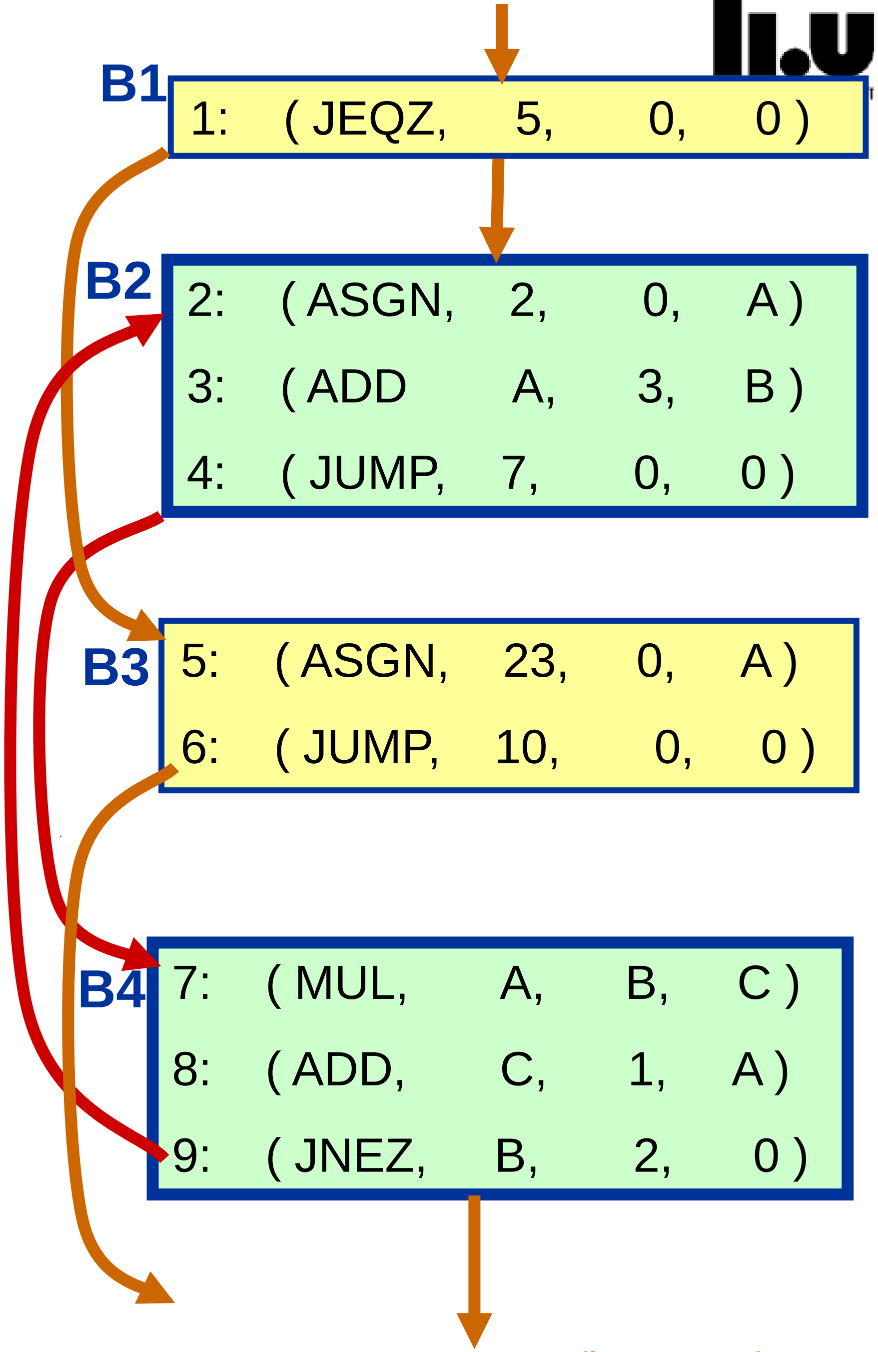
What is a **loop**?

- q A **strongly connected component** (SCC) in the control flow graph resp. basic block graph
- q SCC strongly connected, i.e., all nodes can be reached from all others
- q Has a **unique** entry point
- q Example: { B2, B4 } is an SCC with 2 entry points à not a loop in the strict sense (spaghetti code)



Loop Example

- Removed the 2nd entry point from the previous example
- Example: { B2, B4 } is an SCC with 1 entry points à is a loop!



Loop Optimization Examples (1)

- Loop-invariant code hoisting
  - Move loop-invariant code out of the loop
  - Example:

```
for (i=0;  i<10;  i++)
  a[i] = b[i]  + c / d;
```

→

```
tmp = c / d;
for (i=0;  i<10;  i++)
  a[i] = b[i]  + tmp;
```

Loop Optimization Examples (2)

- Loop unrolling
  - Reduces loop overhead (number of tests/branches) by duplicating loop body. Faster code, but code size expands.
  - In general case, e.g. when odd number loop limit – make it even by handling 1st iteration in an if-statement before loop.
  - Example:

```
i = 1;
while (i <= 50) {
  a[i] = b[i];
  i = i + 1;
}
```

⇒

```
i = 1;
while (i <= 50) {
  a[i] = b[i];
  i = i + 1;
  a[i] = b[i];
  i = i + 1;
}
```

Loop Optimization Examples (3)

- Loop interchange
  - To improve data locality, change the order of inner/outer loop to make data access sequencial
  - This makes accesses within a cache block (reduce cache misses / page faults)
  - Example:

```
for (i=0;  i<N;  i++)
  for (j=0;  j<M;  j++)
    a[ j ][ i ] = 0.0;
```

```
for (j=0;  j<M;  j++)
  for (i=0;  i<N;  i++)
    a[ j ][ i ] = 0.0;
```

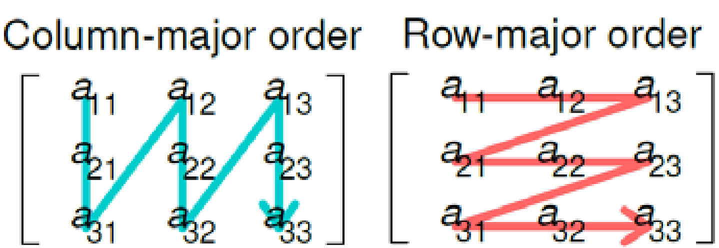
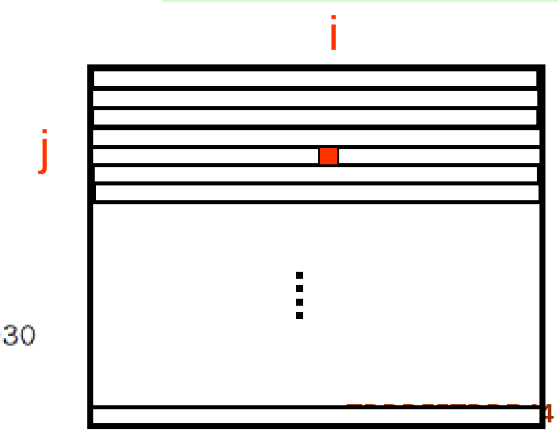


Figure: By Cmglee – Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=65107030



Faster with consecutive data accesses for inner loop

Loop Optimization Examples (4)

- Loop fusion
  - Merge loops with identical headers
  - To improve data locality and reduce number of tests/branches
  - Example:

```
for (i=0;  i<N;  i++)
   a[ i ] = /* … */;
for (i=0;  i<N;  i++)
   f(a[ i ]);
```

⇒

```
for (i=0;  i<N;  i++) {
    a[ i ] = /* … */;
    f(a[ i ]);
}
```

Loop Optimization Examples (5)

- Loop collapsing
  - Flatten a multi-dimensional loop nest
  - May simplify addressing
    (relies on consecutive array layout in memory)
  - Cons: Loss of structure
  - Example:

```
for (i=0;  i<N;  i++)
  for (j=0;  j<M;  j++)
    f( a[ i ][ j ] );
```

⇒

```
for ( ij=0;  ij<M*N;  ij++) {
  f( a[ ij ] );
}
```

Exercise 2:
Draw CFG and find possible loops

Global Optimization

(within a single procedure)

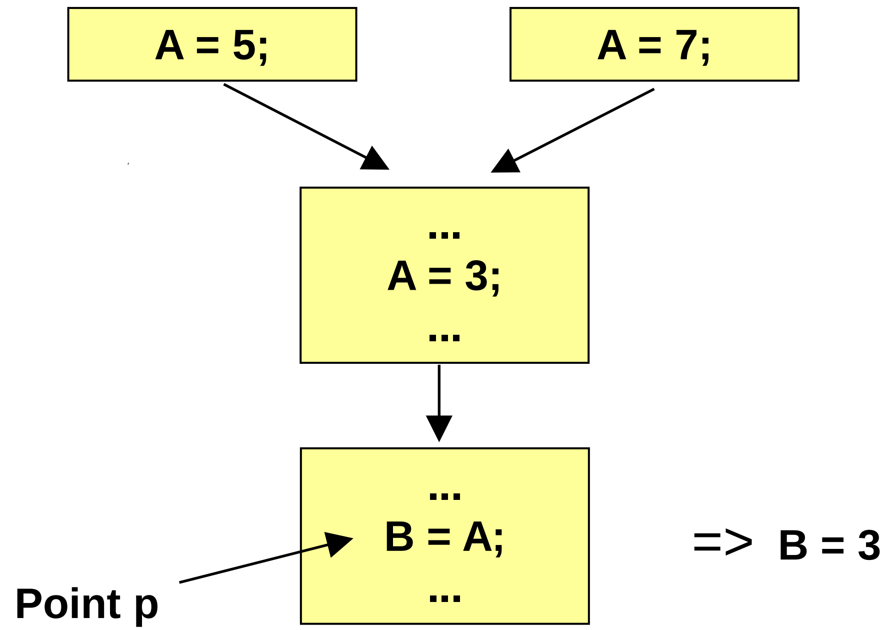
Global Optimization

- More optimization can be achieved if a *whole procedure* (=global optimization) is analyzed
  (Whole program analysis = interprocedural analysis)
  - Global optimization is done within a single procedure
  - Needs *data flow analysis*

- Example of global optimizations
  - Remove variables which are never referenced.
  - Avoid calculations whose results are not used.
  - Remove code which is not called or reachable (i.e., *dead code elimination*).
  - Code motion.
  - Find uninitialized variables.

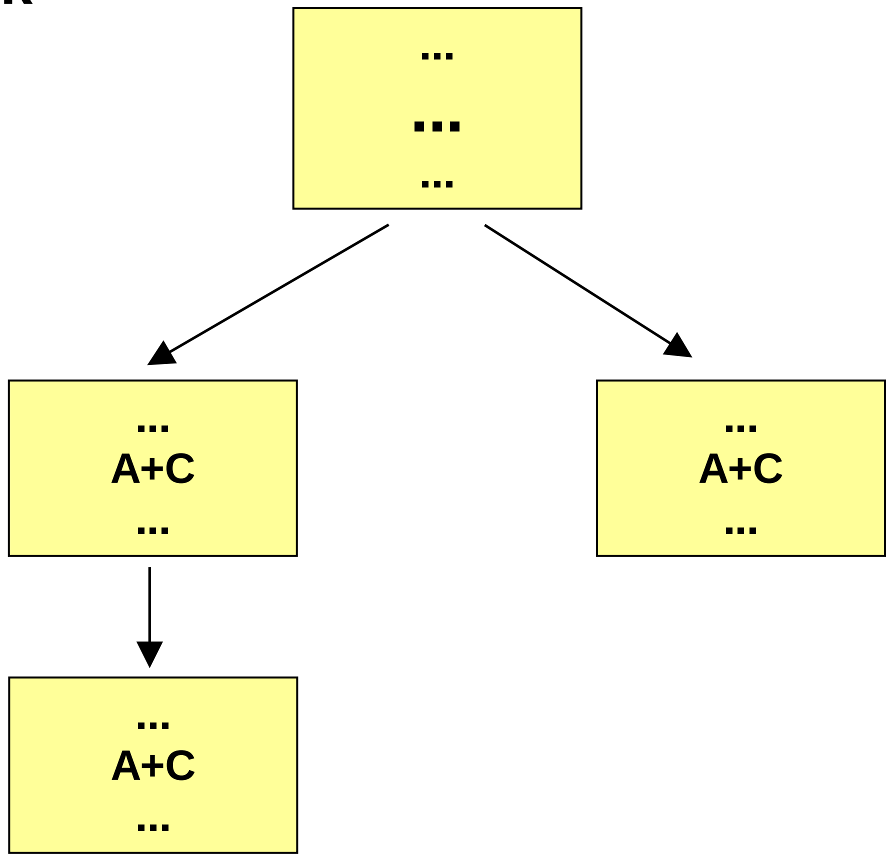
Data Flow Analysis (1)

- Concepts: Data is flowing from definition to use
  - *Definition*: A = 5 *A is defined*
  - *Use*: B = A * C *A is used*
- The flow analysis is performed in two phases, forwards and backwards
- **Forward analysis:**
  - Finds *Reaching definitions*
  - Which definitions apply at a point *p* in a flow graph?



Data Flow Analysis (2), Forward

- *Available expressions*
  - Used to eliminate common subexpressions **over block boundaries**



Example:
An available expression
A+C

Data Flow Analysis (3), Backward

- *Live variables*
  - A variable v is *live* at point *p* if its value is used after *p* before any new definition of v is made.

```
...
v = A;
...
...
c = v;
...
```

Definition of v

Point p

Is there a new definition of v before is used?

v is *live* at point p since there is no new definition of v in between (and v is used after this line)

```
...
v = A;
x = 35;    <- p
c = v;
...
```

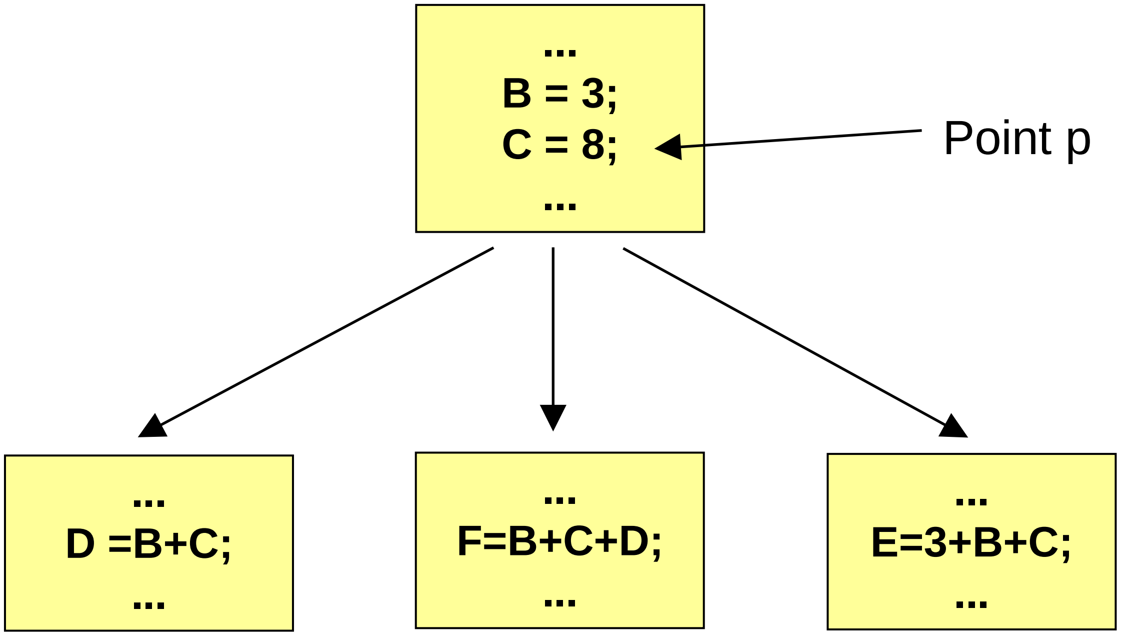
First v is *not live* at point p, since v was redefined before next use

```
...
v = A;
...        <- p
v = 999;
c = v;
...
```

- Example:
  - If variable A is in a register and is dead (not live, will not be referenced) the register can be released

Data Flow Analysis (4), Backward

- *Very-Busy Expressions* or *Anticipated Expressions*
- An expression B+C is *very-busy* at point p if all paths leading from the point p eventually compute the value of the expression B+C from the values of B and C available at p.



Remarks

- Need to analyze **data dependences** to make sure that transformations do not change the semantics of the code
- **Global transformations**
  need control and data flow analysis (within a procedure – *intra*procedural)
- ***Inter*procedural analysis** deals with the whole program
- Covered in more detail in courses
  (Discontinued) TDDC86 Compiler optimizations and code generation
  (9 hp Ph.D. student level) DF00100 Advanced Compiler Construction

Target Optimizations on Target Binary Code

Target-level Optimizations

Often included in main code generation step of back end:

- q Register allocation
  - § Better register use à less memory accesses, less energy
- q Instruction selection
  - § Choice of more powerful instructions for same code
    à faster + shorter code, possibly using fewer registers too
- q Instruction scheduling à reorder instructions for faster code
- q Branch prediction (e.g. guided by profiling data)
- q Predication of conditionally executed code

à See lecture on code generation for RISC and superscalar processors (TDDB44)

à Much more in TDDC86 Compiler optimizations and code generation

Postpass Optimizations (1)

- **"postpass"** = done after target code generation
- **Peephole optimization**
  - Very simple and limited
  - Cleanup after code generation or other transformation
  - Use a window of very few consecutive instructions
  - Could be done in hardware by superscalar processors…

Cannot remove LD instruction since the peephole context is too small (3 instructions). The INC instruction which also loads A is not visible!

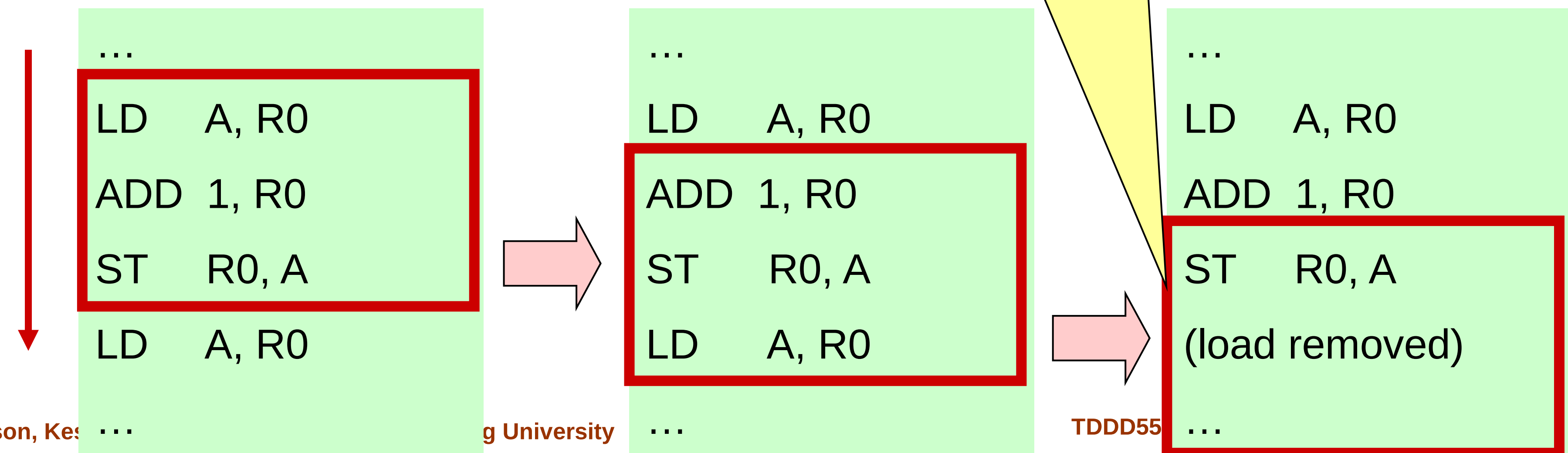
…	…	…
LD A, R0	INC A, R0	INC A, R0
ADD 1, R0	(removed)	(removed)
ST R0, A	(removed	(removed)
LD A, R0	LD A, R0	LD A, R0
…	…	…

Postpass Optimizations (1)

- **"postpass"** = done after target code generation

- **Peephole optimization**
  - Very simple and limited
  - Cleanup after code generation or oth… …formation
  - Use a window of very few consecutive … …tions
  - Could be done in hardware by supersca… …ocessors…

Greedy peephole optimization (as on previous slide) may miss a more profitable alternative optimization (here, removal of a load instruction)



Postpass Optimizations (2)

- **Postpass instruction (re)scheduling**
  - Reconstruct control flow, data dependences from binary code
  - Reorder instructions to improve execution time
  - Works even if no source code is available
  - Can be *retargetable*
    (parameterized in processor architecture specification)
  - E.g., aiPop™ tool by AbsInt GmbH, Saarbrücken

References

- Beniamino Di Martino and Christoph Kessler. “Two program comprehension tools for automatic parallelization”. In: *IEEE Concurrency* 8.1 (2000), pp. 37–47. DOI: 10.1109/4434.824311.
- Christoph Kessler. “Pattern-Driven Automatic Parallelization”. In: *Sci. Program.* 5.3 (Aug. 1996), pp. 251–274. DOI: 10.1155/1996/406379.

Questions?

- Next lecture: L11 - Code Generation