Compiler Construction: Practical Introduction

System Course for SRR Engineers

Samsung Research Russia
Moscow 2019

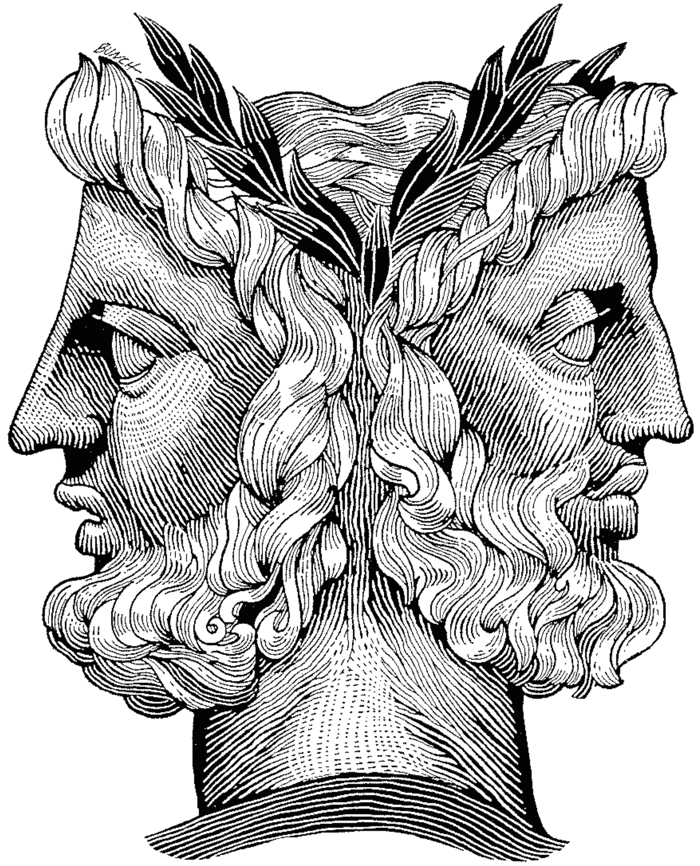
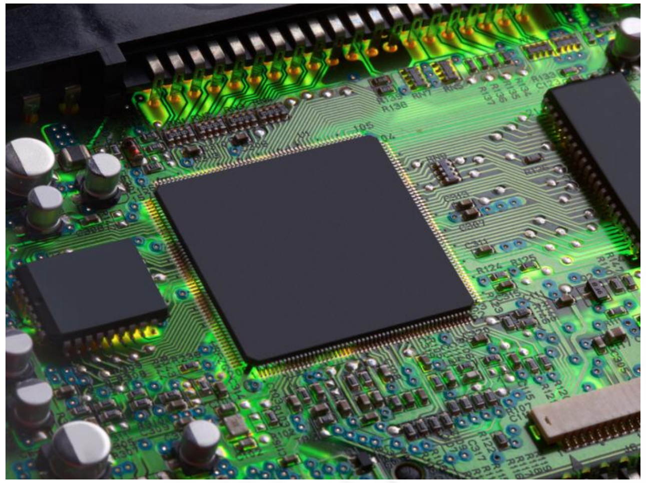
Lecture 6
Back End Essentials

- Compiler Backend
- Optimizations for sequential uniprocessor
- Static Single Assignment form
- Optimizations of parallel computations

Compiler Backend

Role and Responsibilities

Compiler

Compiler Backend

Responsibility

- Optimally map given program to hardware without compromising correctness

Where is compiler backend?

- Regular compilers (gcc GIMPLE, LLVM)
- Virtual Machines for managed languages (JVM, .NET)
- Scripting languages (Python, JavaScript &.c)
  - Even in browser
- Optimizing execution of a Machine Learning model (e.g. neural network)

Tasks and Issues

- Backend should:
  - Select suitable processor instructions
  - Allocate spare hardware resources such as registers
  - Optimize execution sequence
  - Schedule operations to minimize total execution time
  - &c.
- Backend need to know:
  - How program is executed (Control flow analysis)
  - What values are used during execution (Data flow analysis)
  - “Understand” program’s semantics

Optimizations Examples

- Conditional Constant Propagation
  - Constant propagation is a transformation that, given an assignment x = c for a variable x and a constant c, replaces later uses of x with uses of c as long as intervening assignments have not changed the value of x.
- Tail-recursion optimization
  - A call from procedure f ( ) to procedure g ( ) is a tail call if the only thing f ( ) does, after g ( ) returns to it, is itself return. The call is tail-recursive if f ( ) and g ( ) are the same procedure.
- Partial Redundancy Elimination
  - Partial-redundancy elimination inserts and deletes computations in the flowgraph in such a way that after the transformation each path contains no more-and, generally, fewer-occurrences of any such computation than before.
- Register Allocation
  - Register allocation determines which of the values (variables, temporaries, and large constants) that might profitably be in a machine's registers should be in registers at each point in the execution of a program.

Intermediate Representations

- System of structures that collect essential information about program convenient for analysis, transformation and target code generation
- Two kinds
  - High level IR (platform independent)
  - Low level IR (platform dependent)
- E.g.
  - Abstract Syntax Tree (AST)
  - Static Single Assignment (SSA)
  - gcc: GIMPLE, RTL
  - Clang: LLVM
  - V8 Java Script engine: Hydrogen, Litium

Optimizations for sequential uniprocessor

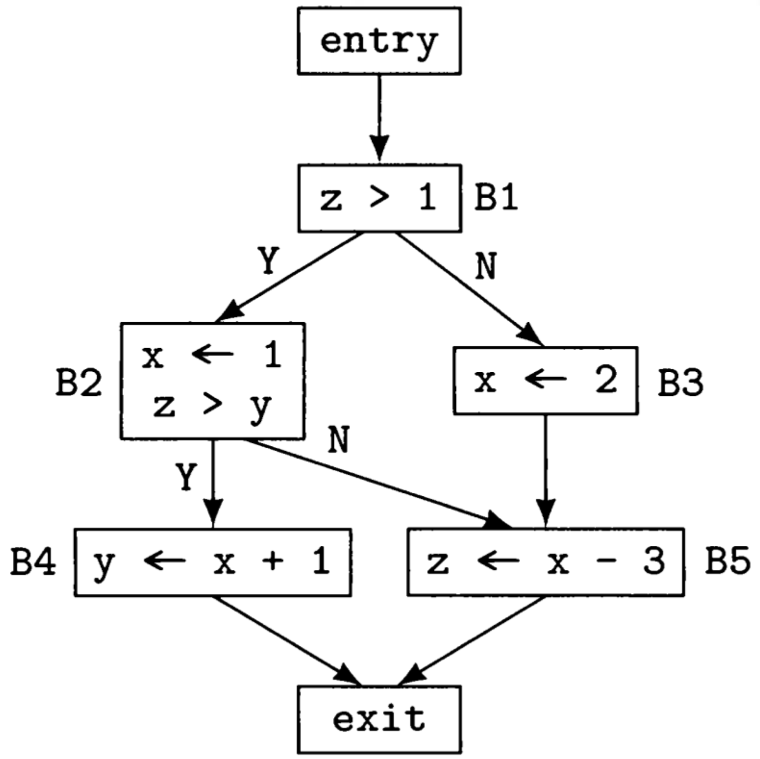
Reaching Definitions Problem

Control Flow Graph

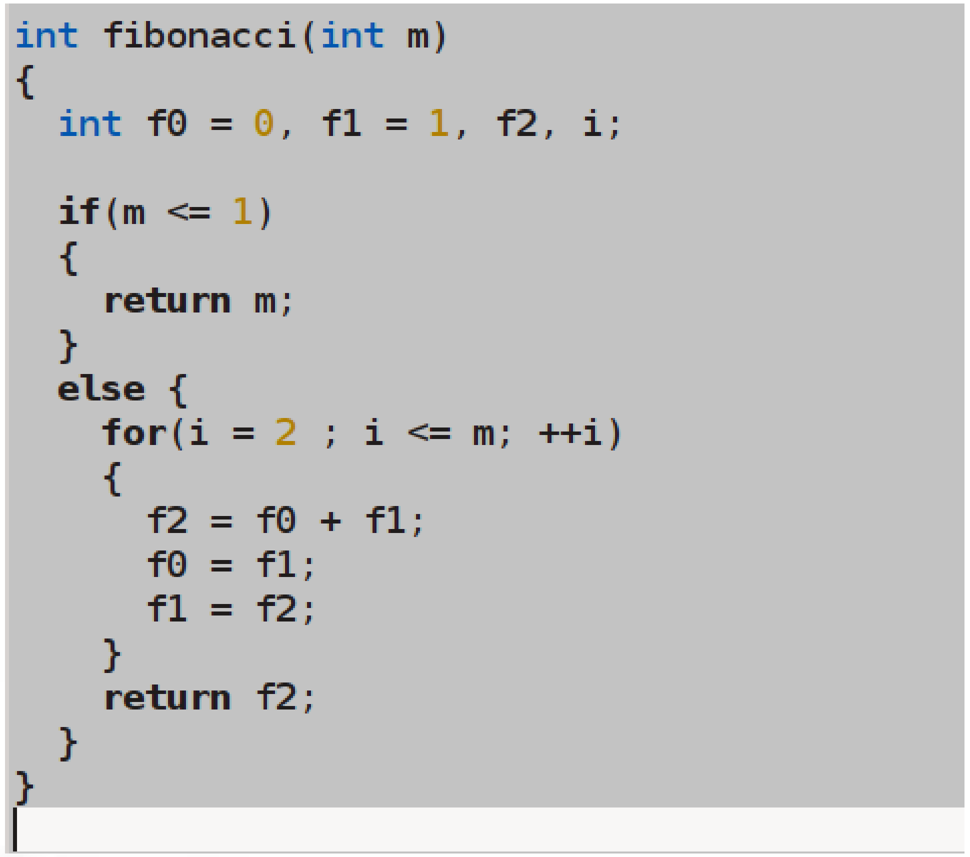
Data Flow Analysis

Defs & Uses

- **Definition** of the variable is a point in program code where it is assigned to.
- **Use** of variable is a point in code where its value is used.
- Du- and ud-chains are a sparse representation of data-flow information about variables.
- **Du-chain** for a variable connects a definition of that variable to all the uses it may flow to.
- **Ud-chain** connects a use to all the definitions that may flow to it.



Reaching Definitions

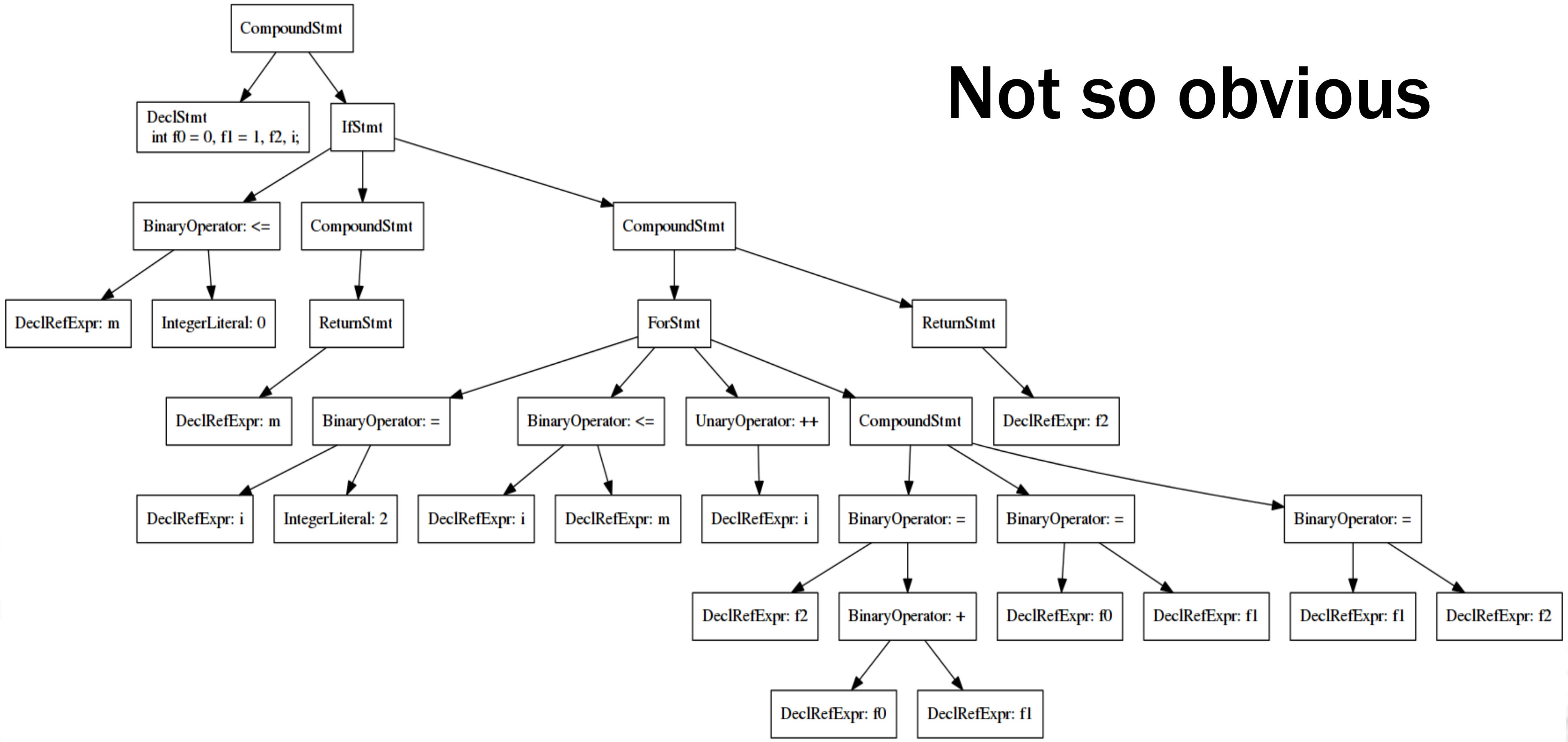


```
int fibonacci(int m)
{
  int f0 = 0, f1 = 1, f2, i;

  if(m <= 1)
  {
    return m;
  }
  else {
    for(i = 2 ; i <= m; ++i)
    {
      f2 = f0 + f1;
      f0 = f1;
      f1 = f2;
    }
    return f2;
  }
}
```

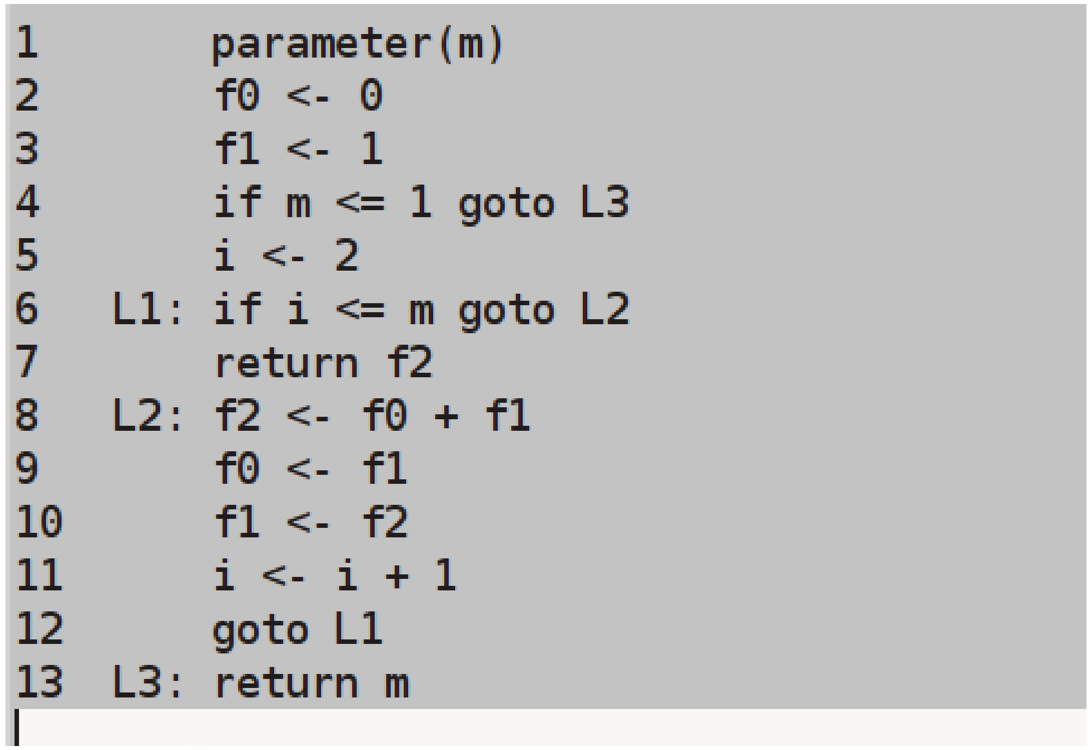
Whether f2 in return statement has definite value?

Abstract Syntax Tree



Not so obvious

Backend Intermediate Representation

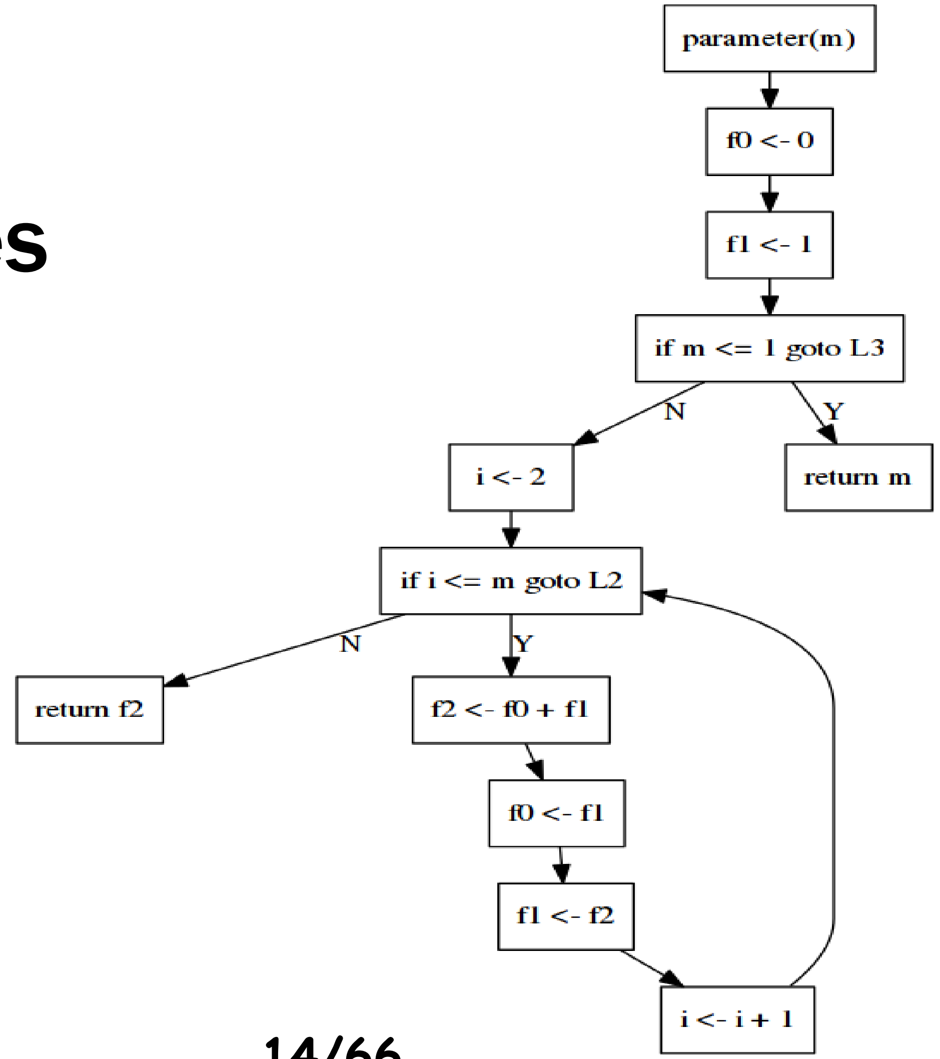


```
    parameter(m)
    f0 <- 0
    f1 <- 1
    if m <= 1 goto L3
    i <- 2
L1: if i <= m goto L2
    return f2
L2: f2 <- f0 + f1
    f0 <- f1
    f1 <- f2
    i <- i + 1
    goto L1
L3: return m
```

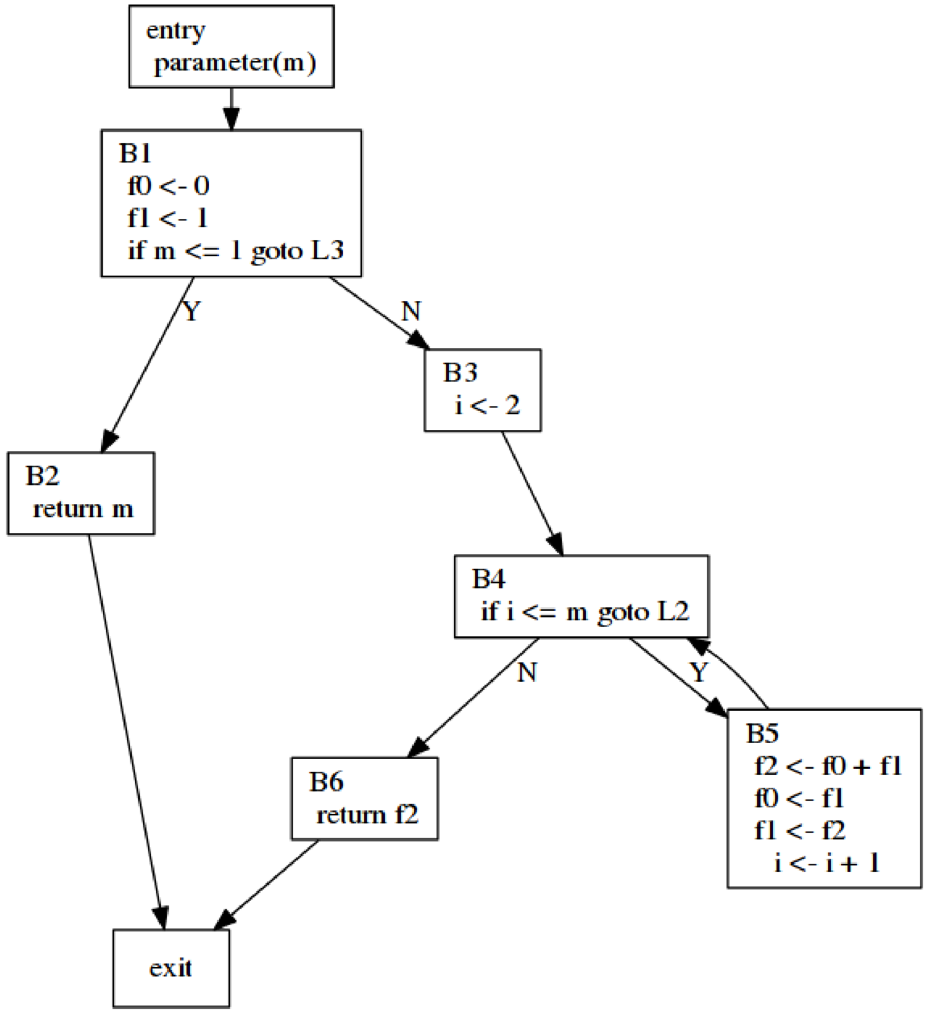
Can perform Symbolic Execution

Control Flow Chart

Too many nodes and edges



Control Flow Graph (CFG)

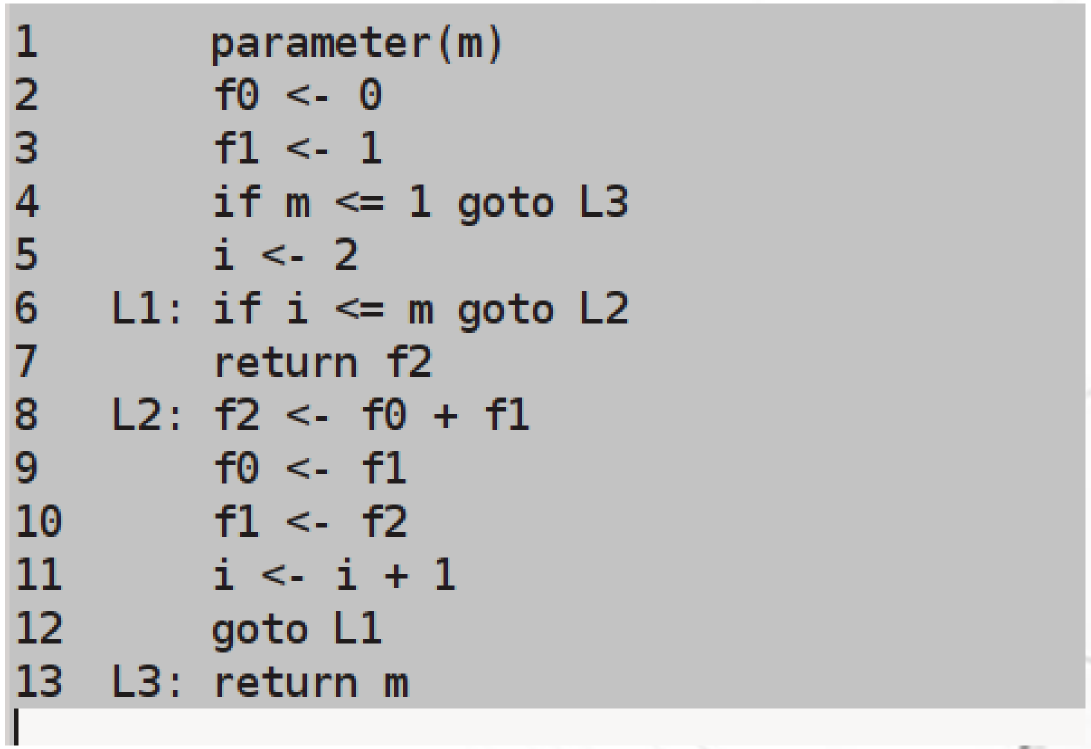


CFG summarizes control flow with Basic Blocks and control edges

Basic Block: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit

Data Flow Analysis: Reaching Definitions

Bit Position	Definition	Basic Block
1	m in line 1	B1
2	f0 in line 2	
3	f1 in line 3	
4	i in line 5	B3
5	f2 in line 8	B5
6	f0 in line 9	
7	f1 in line 10	
8	i in line 11	



```
1        parameter(m)
2        f0 <- 0
3        f1 <- 1
4        if m <= 1 goto L3
5        i <- 2
6    L1: if i <= m goto L2
7        return f2
8    L2: f2 <- f0 + f1
9        f0 <- f1
10       f1 <- f2
11       i <- i + 1
12       goto L1
13   L3: return m
```

Data Flow Analysis: Reaching Definitions

- RCHin – definitions reaching block entry
- RCHout – definitions reaching block exit
- PRSV – definitions preserved by block
- GEN – definitions generated by block

- Summarize PRSV and GEN sets for each Basic Block
- Iteratively distribute information through control edges

Reaching Definitions: Initialization

- RCHin(entry) = <0 0 0 0 0 0 0 0>
- RCHin(i) = <0 0 0 0 0 0 0 0> for ∀ i
- PRSV(B1) = <0 0 0 1 1 0 0 1>
- PRSV(B3) = <1 1 1 0 1 1 1 0>
- PRSV(B5) = <1 0 0 0 0 0 0 0>
- PRSV(i) = <1 1 1 1 1 1 1 1> for other i
- GEN(B1) = <1 1 1 0 0 0 0 0>
- GEN(B3) = < 0 0 0 1 0 0 0 0>
- GEN(B5) = < 0 0 0 0 1 1 1 1>
- GEN(i) = <0 0 0 0 0 0 0 0> for other i
- RCHout(i) = <0 0 0 0 0 0 0 0> for ∀ i

Bit Position	Definition	Basic Block
1	m in line 1	B1
2	f0 in line 2	
3	f1 in line 3	
4	i in line 5	B3
5	f2 in line 8	B5
6	f0 in line 9	
7	f1 in line 10	
8	i in line 11	

Reaching Definitions: Equations

$$RCHout(i) = GEN(i) \cup (RCHin(i) \cap PRSV(i)), \forall i$$

$$RCHin(i) = \bigcup_{j \in \mathrm{Pred}(i)} RCHout(j), \forall i$$

Reaching Definitions: Result

RCHout(entry) = <0 0 0 0 0 0 0 0>	RCHin(entry) = <0 0 0 0 0 0 0 0>
RCHout(B1) = <1 1 1 0 0 0 0 0>	RCHin(B1) = <0 0 0 0 0 0 0 0>
RCHout(B2) = <1 1 1 0 0 0 0 0>	RCHin(B2) = <1 1 1 0 0 0 0 0>
RCHout(B3) = <1 1 1 1 0 0 0 0>	RCHin(B3) = <1 1 1 0 0 0 0 0>
RCHout(B4) = <1 1 1 1 1 1 1 1>	RCHin(B4) = <1 1 1 1 1 1 1 1>
RCHout(B5) = <1 0 0 0 1 1 1 1>	RCHin(B5) = <1 1 1 1 1 1 1 1>
RCHout(B6) = <1 1 1 1 1 1 1 1>	RCHin(B6) = <1 1 1 1 1 1 1 1>
RCHout(exit) = <1 1 1 1 1 1 1 1>	RCHin(exit) = <1 1 1 1 1 1 1 1>

f2 in block B6 has definite value

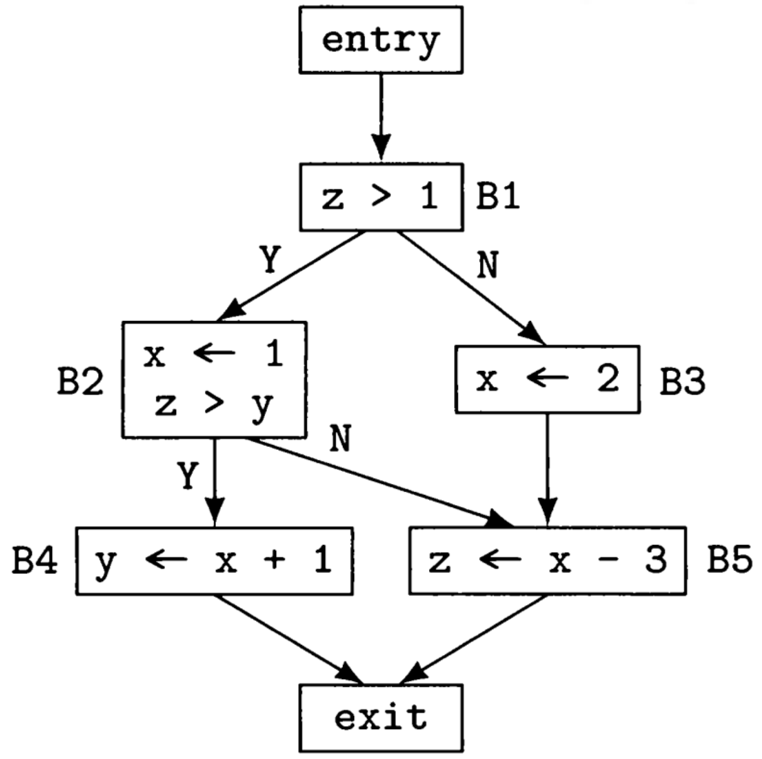
Data Flow Analysis: Applications

Problems for DFA:

- Reaching definitions
- Upward exposed uses
- Live variables
- Available expressions
- Copy propagation
- Constant propagation

Variations of DFA

- Forward analysis: from entry to exit
- Backward analysis: from exit to entry
- Confluence operator: union vs intersection



Static Single Assignment Form

Conditional Constant Propagation

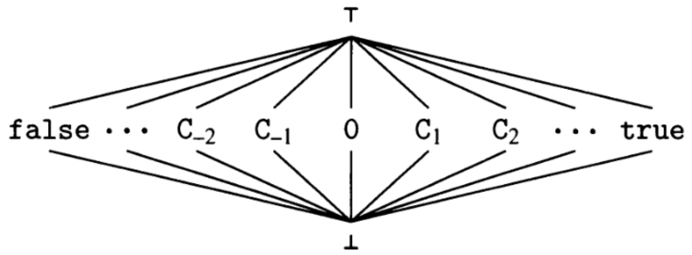
Building SSA

Conditional Constant Propagation

```
x = 1;
y = x + 2;
if(y > x)
  y = 5;
// ...
... y ...
```

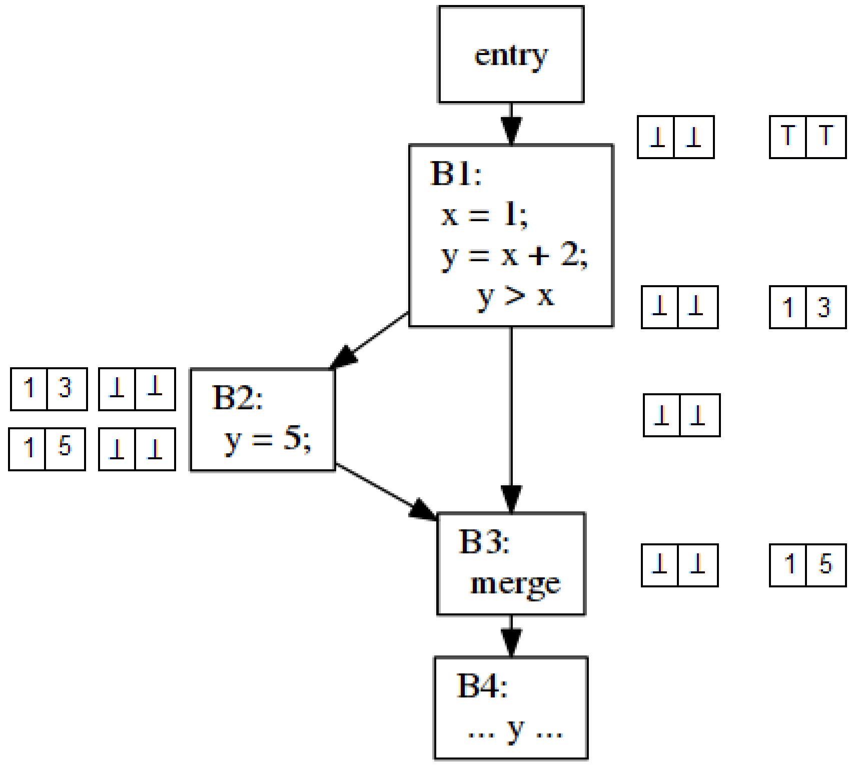
```
x = 1;
y = 3;        // dead code
if(true)
  y = 5;
// ...
... 5 ...
```

Conditional Constant Propagation: Lattice



- Lattice for Constant Propagation
  - ⊥ – yet to be determined
  - T – can’t determine / definitely not a constant
- Operators:
  - join(a, b): lowest value above both a and b (join(T, 0) = T, join(0, -1) = T)
  - meet(a, b): highest value below both a and b (meet(0, -1) = ⊥, meet(T, 1) = 1)
- Evaluation of expressions EVAL(e, Vin):
  - If any argument of e in Vin is T (or ⊥) returns T (or ⊥), otherwise evaluate as usual

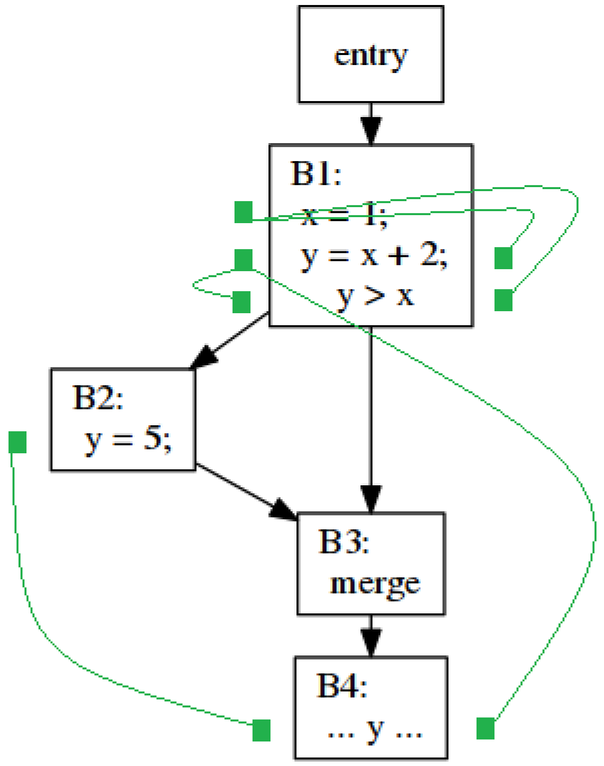
Conditional Constant Propagation: Result



Algorithm Complexity

- Height of lattice == 2 $\Rightarrow$ each vector can change value 2 * V times
- Maximal number of iterations: 2 * V * E times
- Cost of each iteration: O(V)
- Overall algorithm takes $O(EV^2)$ time
- A lot of useless coping unchanged values from one vector to another

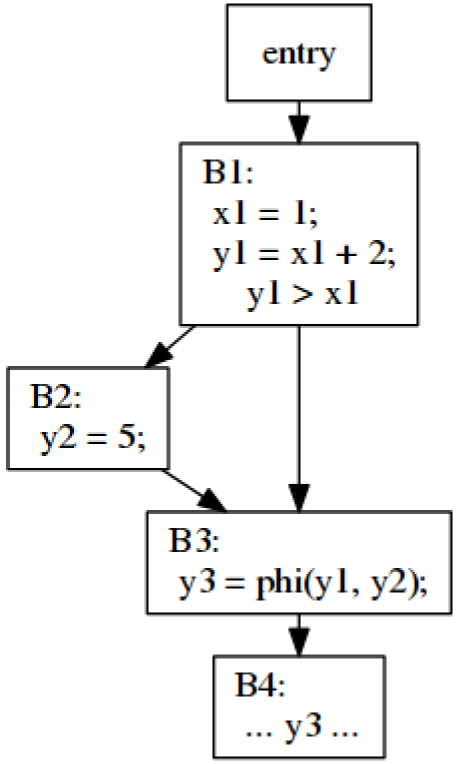
Conditional Constant Propagation: Def-use



- Sparse representation: def-use chains
- Graph for each variable connecting definition to all reachable uses
- **Complexity: O(N^2V)**
- Loss of accuracy: does not take into account control flow!

Conditional Constant Propagation: SSA

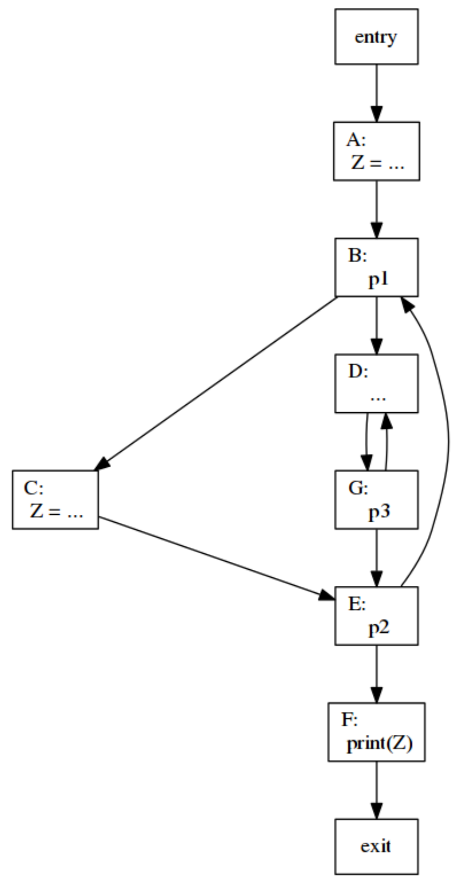
- SSA – Static Single Assignment form
- Each definition creates new version of variable
- φ function – pseudo-operation that combines different definitions at merge point to new version
- SSA is a factorized def-use chain which respects control flow
- Constant Propagation in SSA form has complexity O(EV)

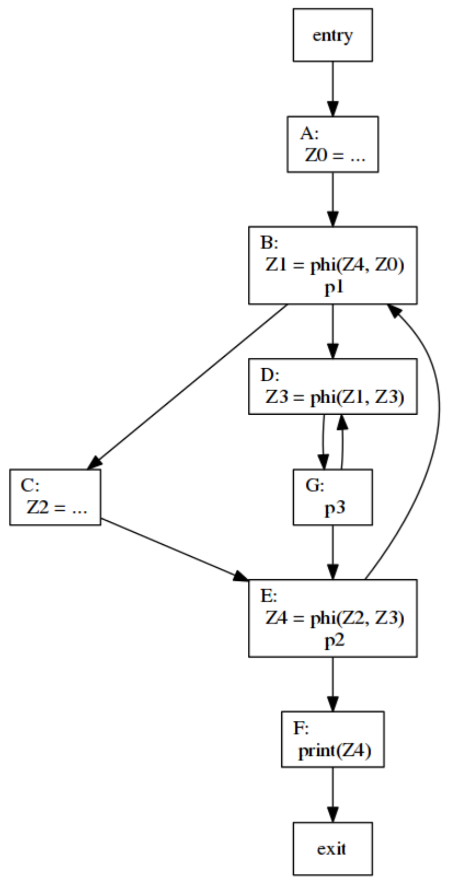


Building SSA

- Intermediate representation of program in which **every use** of a variable is reached by **exactly one definition**.
- Most program do not satisfy this condition. One requires to insert dummy assignments called φ-**functions** to merge multiple definitions.
- Simple algorithm:
  - Insert φ-functions for all variables at all CFG merge points
  - Solve reaching definitions
  - Rename all real and dummy assignments of variables uniquely

Building SSA

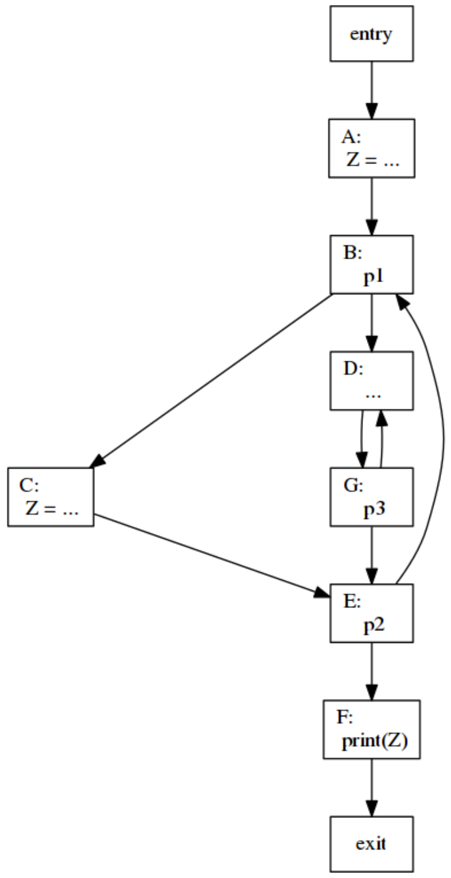


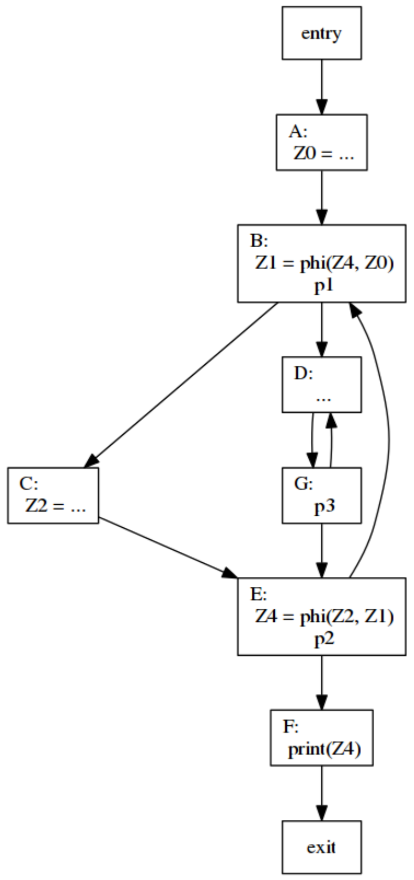


Building SSA

- In the previous slide, dummy assignment Z3 in not really needed since there is no actual assignment to Z in nodes D and G of the original program
- Minimal SSA Form
  - Place φ-functions only where they are really needed
  - Perform direct renaming

Building Minimal SSA





Building Minimal SSA

- Node a dominates node b iff every path from entry to b goes through a
- Node u is in dominance frontier of node w if w
  - dominates a CFG predecessor v of u, but
  - does not strictly dominate u
- Dominance frontier == control dependence in reverse graph!
- Iterated dominance frontier: irreflexive transitive closure of dominance frontier relation
- Where to place φ-functions for a variable Z:
  - Let Assignment = {START} ∪ { nodes with assignment to Z}
  - Find I = iterated dominance frontier of Assignment
  - Place φ-functions in nodes of set I

Optimizations of Parallel Computations

Parallelization overview

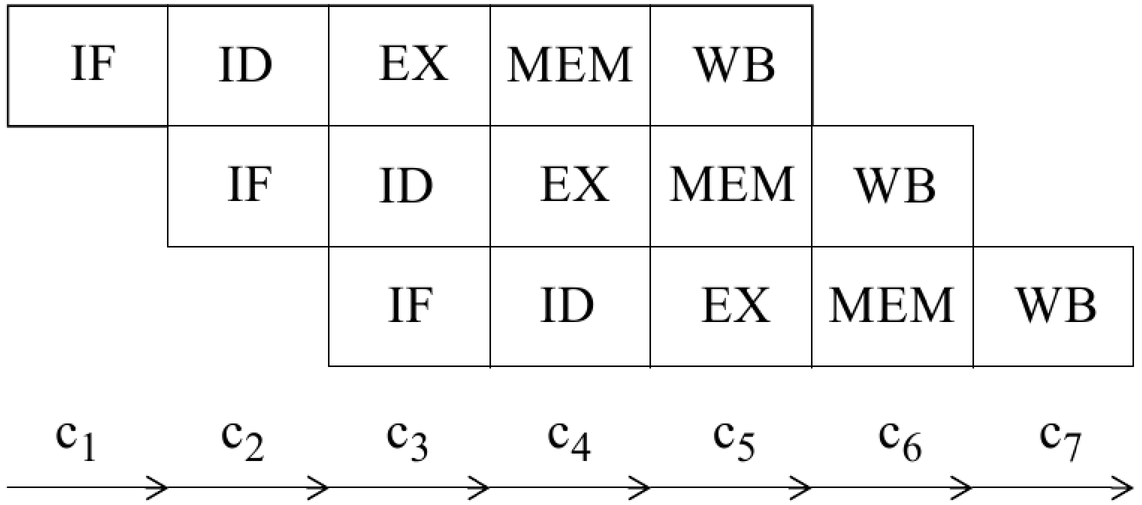
Matrix multiplication case study

Introduction into dependency analysis

Types of Parallelism

- Bit Level Parallelism
  - 8 => 16 => 32 => 64 => …
- Instruction Level Parallelism
  - Pipelining
  - Superscalar and Very Large Instruction Word (VLIW, EPIC)
- Data Level Parallelism
  - Vector Instructions
  - Graphic Processing Unit (GPU)
- Task Level Parallelism
  - Multicore and Manycore
  - Simultaneous multithreading, Hyperthreading
  - Clusters and Datacenters

Pipelined Instruction Unit

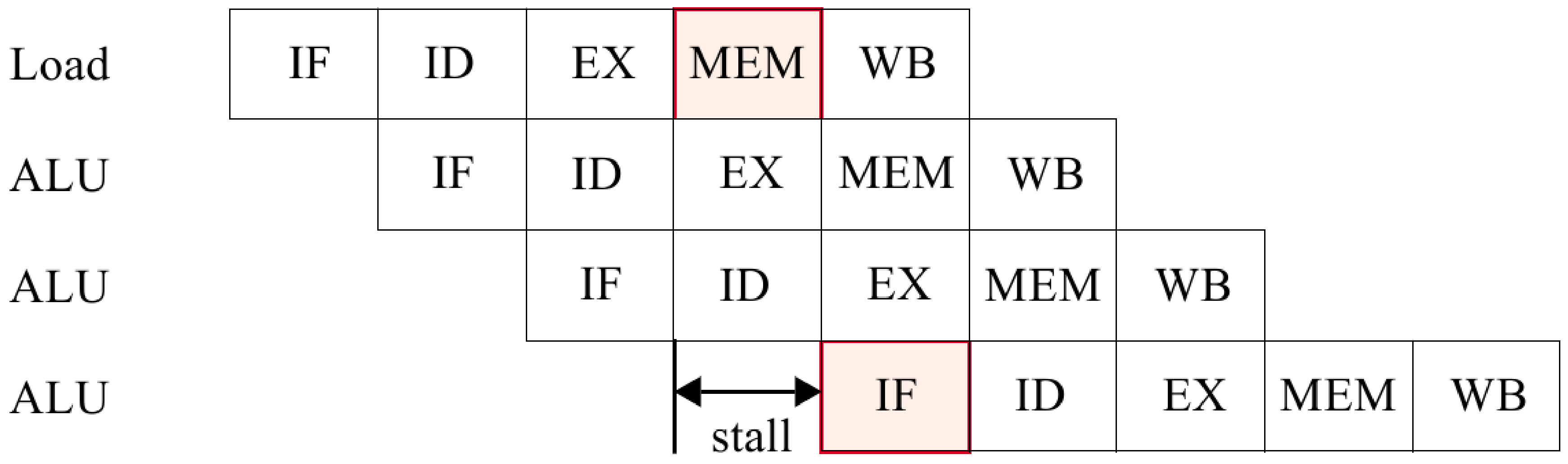


IF – Instruction Fetch
ID – Instruction Decode
EX – Execute
MEM – Memory Access
WB – Write Back

Compiling for Instruction Pipeline

- **Structural hazards** – hardware resources do not support all possible combinations of instruction overlap
- **Data hazards** – result produced by one instruction is needed by subsequent instruction
- **Control hazards** – pipeline stall due to branching instruction

Instruction Pipeline: Structural Hazard



Load	IF	ID	EX	MEM	WB				
ALU		IF	ID	EX	MEM	WB			
ALU			IF	ID	EX	MEM	WB		
ALU				stall	IF	ID	EX	MEM	WB

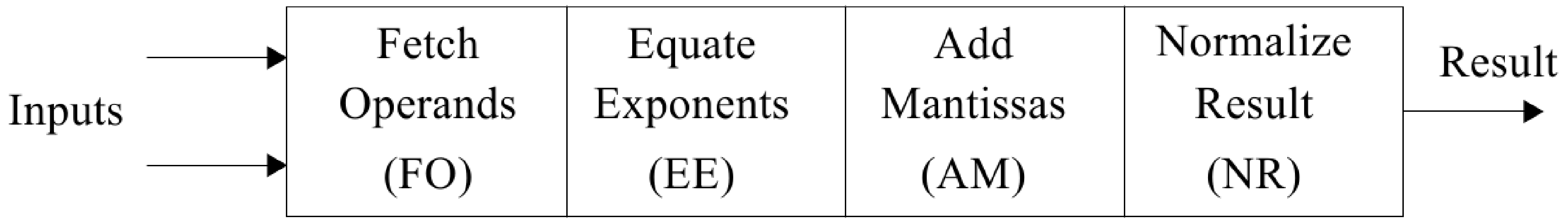
Instruction Pipeline: Data Hazard

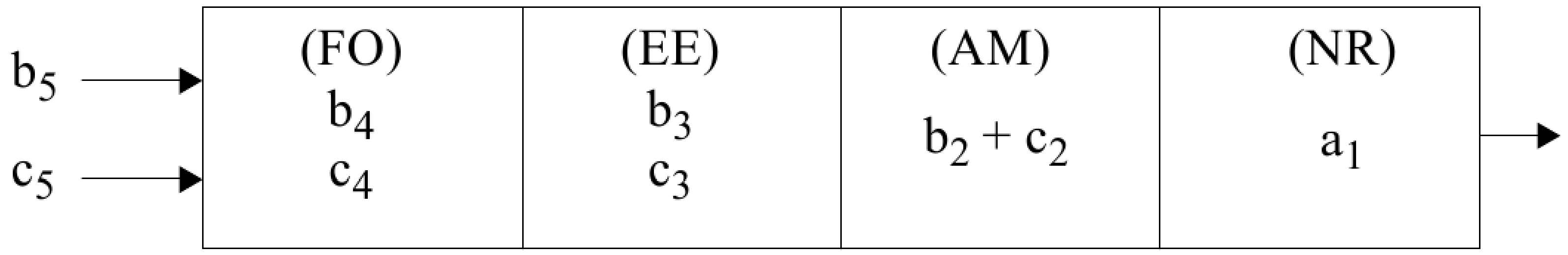
`LW R1,0(R2)`	IF	ID	EX	MEM	WB		
`ADD R3,R1,R4`		IF	ID	stall	EX	MEM	WB

Instruction Pipeline: Control Hazard

Conditional Branch	IF	ID	EX	MEM	WB				
		IF	ID	EX	IF	ID	EX	MEM	WB
		stall	stall	stall					

Pipelined Execution Unit: Floating Adder

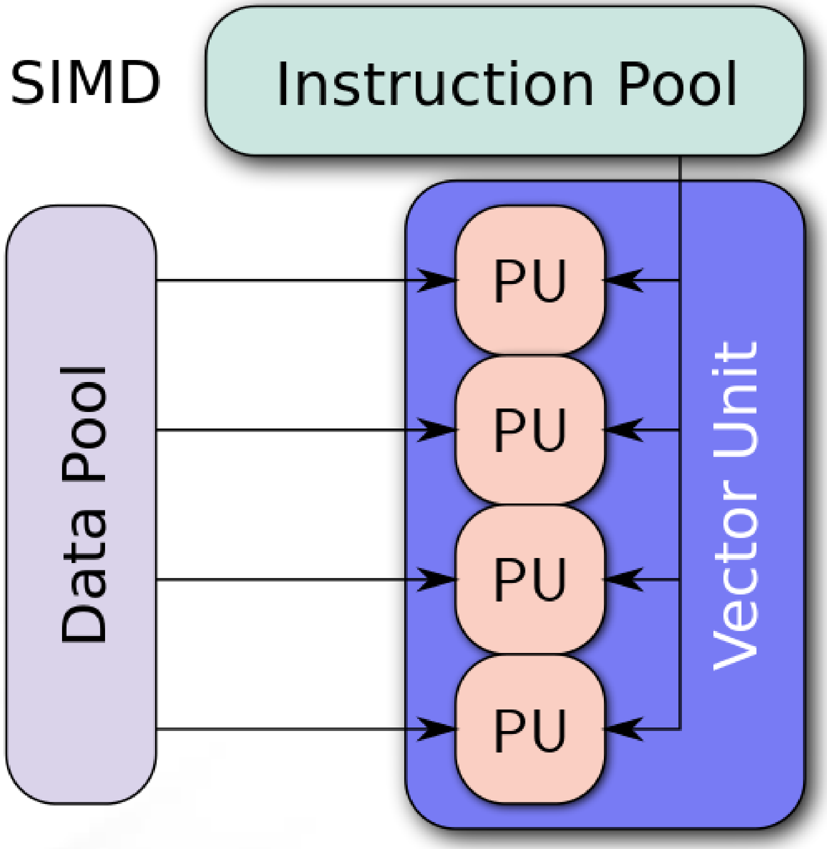




Pipelined Execution Unit: Data Hazard

ADDD R3,R1,R2	IF	ID	EX1	EX2	MEM	WB		
ADDD R3,R3,R4		IF	ID	stall	EX1	EX2	MEM	WB

Vector Instruction Unit



- Vector ALU
- Vector registers
- Vector instructions

```
VLOAD VR1, M1
VLOAD VR2, M2
VADD VR3, VR1, VR2
```

- Intel: MMX, SSE, AVX
  ARM: NEON
  PowerPC: AltiVec
  Sparc: VIS
  MIPS: MSA

Compiling for Vector Processor

- Vector instruction sequence:

```
VLOAD VR1, A
VLOAD VR2, B
VADD VR3, VR1, VR2
VSTORE VR3, C
```

- In language supporting vector instructions (Cilk):

```
C[0:63] = A[0:63] + B[0:63]
```

- In usual language:

```
for( i = 0; i < 64; ++i)
        C[i] = A[i] + B[i]
```

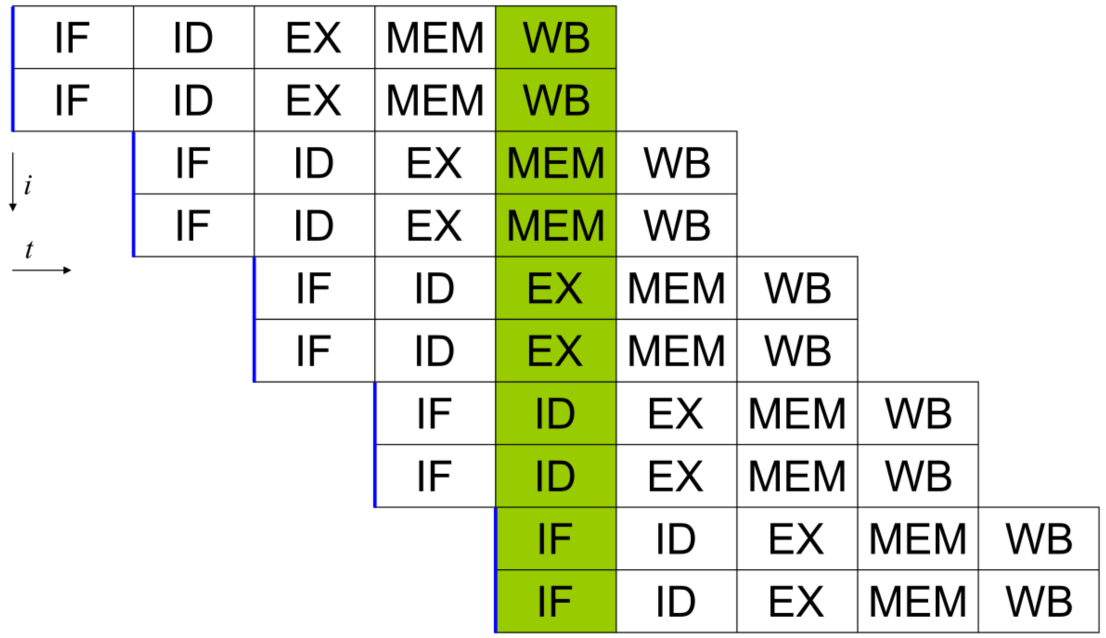
- A slightly modified example:

```
for(i = 0; i < 64; ++i)
        A[i + 1] = A[i] + B[i]
```

- In language supporting vector instructions:

```
A[1:64] = A[0:63] + B[0:63]          // NOT VECTORIZED
```

Multiple-Issue Instruction Unit

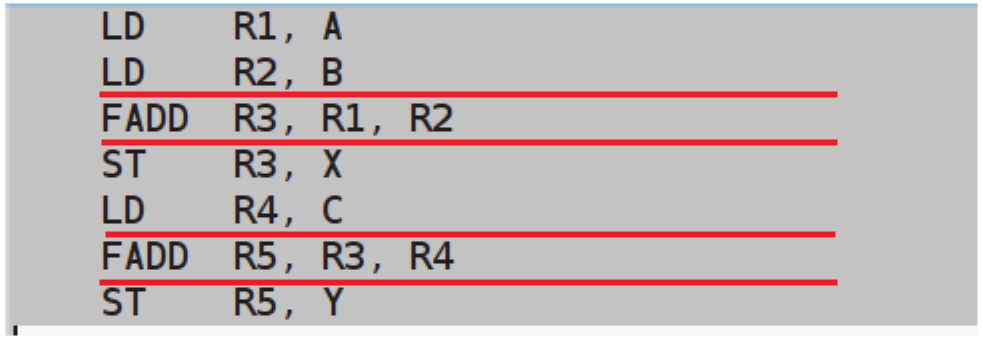


IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

- Multiple functional units
  - Integer ALUs
  - Floating ALUs
  - Address ALUs
- Multiple-issue decoder
- **Superscalar:** look ahead in instruction stream for ready-to-execute instructions
- **VLIW:** execute single wide instruction in each cycle

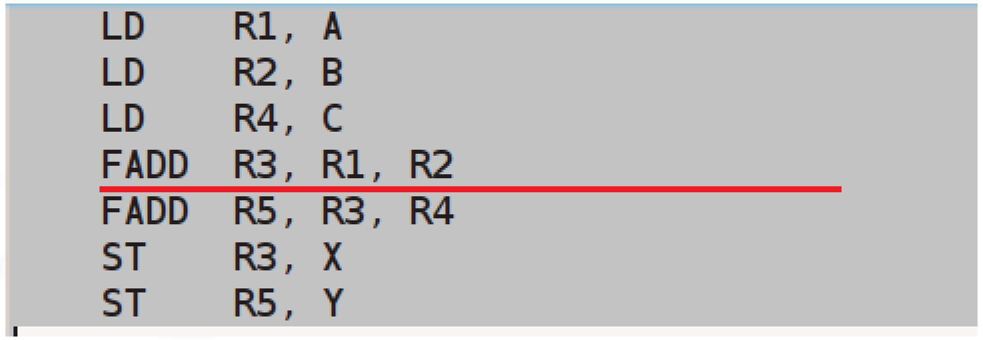
More than one instruction per cycle

Scheduling for Pipelined Processor



```
LD     R1, A
LD     R2, B
FADD   R3, R1, R2
ST     R3, X
LD     R4, C
FADD   R5, R3, R4
ST     R5, Y
```

7 instruction + 4 delays = 11 cycles

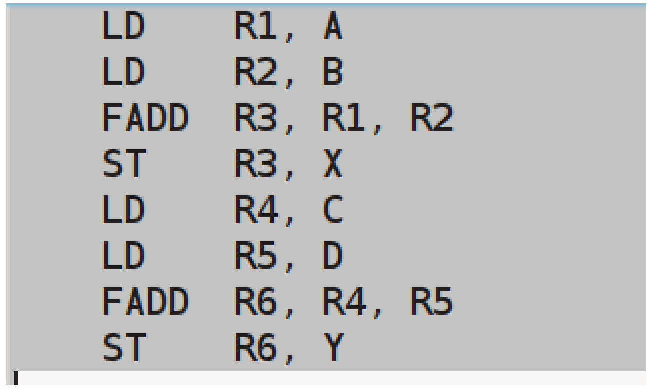


```
LD     R1, A
LD     R2, B
LD     R4, C
FADD   R3, R1, R2
FADD   R5, R3, R4
ST     R3, X
ST     R5, Y
```

7 instruction + 1 delay = 8 cycles

Scheduling for Multiple-issue Processor

Suppose we have processor with 2 loads but 1 addition per cycle

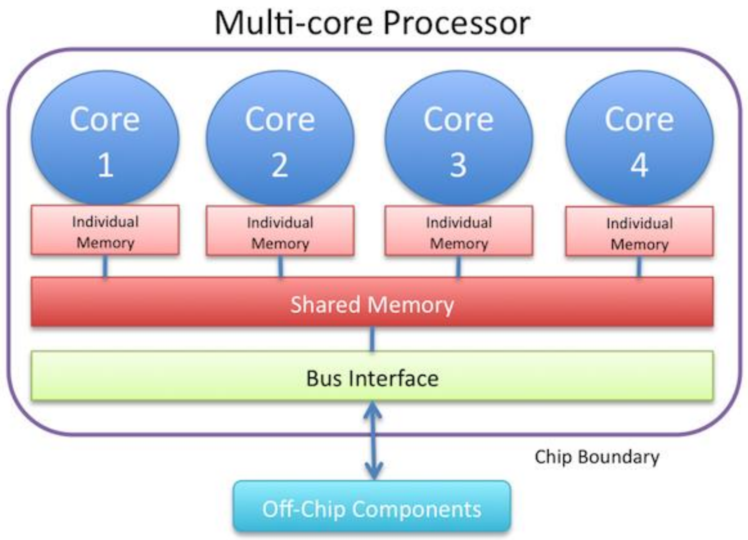


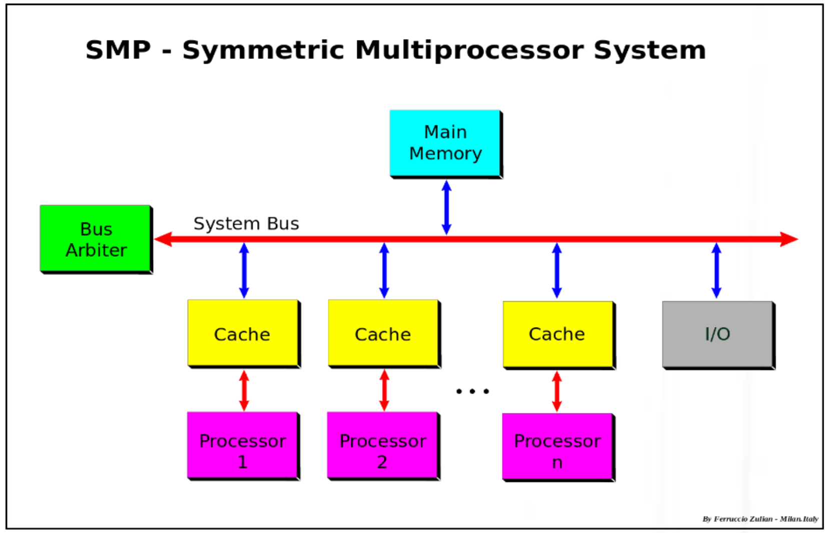
```
LD    R1, A
LD    R2, B
FADD  R3, R1, R2
ST    R3, X
LD    R4, C
LD    R5, D
FADD  R6, R4, R5
ST    R6, Y
```

LD R1, A	LD R4, C
LD R2, B	LD R5, D
delay	delay
FADD R3, R1, R2	delay
STD R3, X	FADD R6, R4, R5
empty	ST R6, Y

VLIW – compiler generates wide instructions with two slots
Superscalar – compiler schedules instructions to fit in look-ahead window

Processor Parallelism





- Multicore processor: single component with two or more independent actual processing unit
- Multiprocessor system: two or more CPUs within a single computer system

Compiling for Processor Parallelism

```
#pragma omp parallel for
  for(i = 0; i < N; ++i)
    C[i] = A[i] + B[i];
```

Parallelized / Vectorized

```
#pragma omp parallel for
  for(i = 0; i < N; ++i)
    A[i + 1] = A[i] + B[i];
```

Non-Parallelized / Non-Vectorized

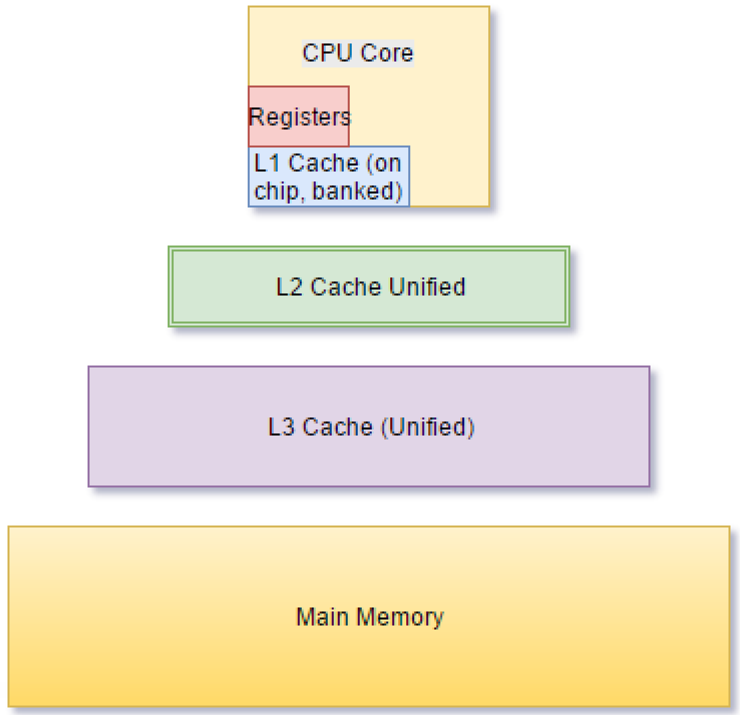
```
#pragma omp parallel for
  for(i = 0; i < N; ++i)
    A[i - 1] = A[i] + B[i];
```

Non-Parallelized / Vectorized

```
#pragma omp parallel for
  for(i = 0; i < N; ++i)
    S = A[i] + B[i];
```

Non-Parallelized / Non-Vectorized

Memory Hierarchy



- Latency – number of cycles required to deliver single data element form memory
- Bandwidth – number of elements delivered on each cycle
- Latency avoidance – reduce latency in computations (caches)
- Latency tolerance – doing something else while data is being fetched

Compiling for Memory Hierarchy

```
for(i = 0; i < N; ++i)
  for(j = 0; j < M; ++j)
    A[i] = A[i] + B[j];
```

```
for(jj = 0; jj < M; jj += L)
  for(i = 0; i < N; ++i)
    for(j = jj; j < jj + L; ++j)
      A[i] = A[i] + B[j];
```

- Access for array A fits into cache
- Access for array B misses for large M
- Suppose cache size larger than L array elements
- Strip-mining loop

Matrix Multiplication: Scalar Uniprocessor

```
for(i = 0; i < N; ++i)
  for(j = 0; j < N; ++j)
  {
    C[i][j] = 0.0;
    for(k = 0; k < N; ++k)
      C[i][j] += A[i][k] * B[k][j];
  }
```

Straightforward algorithm – good performance on scalar uniprocessor

Matrix Multiplication: Pipelined Processor

```
for(i = 0; i < N; i += 4)
  for(j = 0; j < N; ++j)
  {
    C[i][j]   = 0.0;
    C[i+1][j] = 0.0;
    C[i+2][j] = 0.0;
    C[i+3][j] = 0.0;
    for(k = 0; k < N; ++k)
    {
      C[i][j]   += A[i][k]   * B[k][j];
      C[i+1][j] += A[i+1][k] * B[k][j];
      C[i+2][j] += A[i+2][k] * B[k][j];
      C[i+3][j] += A[i+3][k] * B[k][j];
    }
  }
```

Loop unrolling

Matrix Multiplication: Vector Processor

```
for(i = 0; i < N; i += 64)
  for(j = 0; j < N; ++j)
  {
    C[i:i+63][j] = 0.0;
    for(k = 0; k < N; ++k)
      C[i:i+63][j] += A[i:i+63][k] * B[k][j];
  }
```

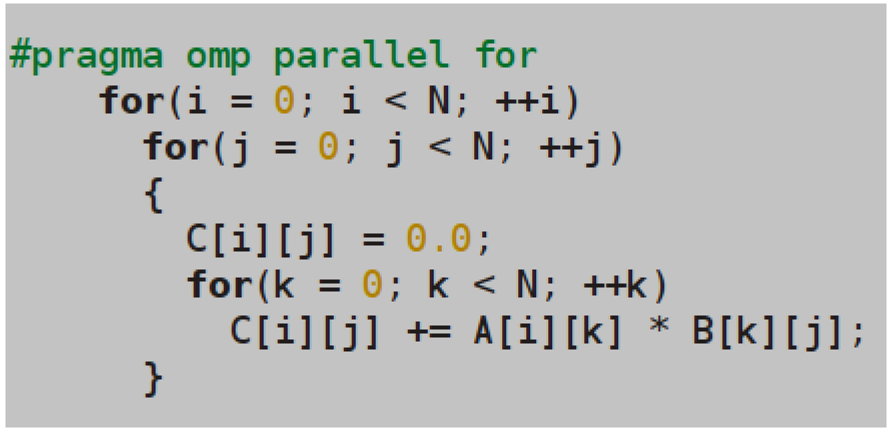
Innermost loop is not vectorized,
So vectorize outermost loop

Matrix Multiplication: Superscalar Processor

```
for(i = 0; i < N; i += 4)
  for(j = 0; j < N; j += 4)
  {
    C[i:i+3][j]   = 0.0;
    C[i:i+3][j+1] = 0.0;
    C[i:i+3][j+2] = 0.0;
    C[i:i+3][j+3] = 0.0;
    for(k = 0; k < N; ++k)
    {
      C[i:i+3][j]   += A[i:i+3][k]  * B[k][j];
      C[i:i+3][j+1] += A[i:i+3][k]  * B[k][j+1];
      C[i:i+3][j+2] += A[i:i+3][k]  * B[k][j+2];
      C[i:i+3][j+3] += A[i:i+3][k]  * B[k][j+3];
    }
  }
```

4-way simultaneous issue, 4 floating-point multiply-adders, 4 pipeline stages

Matrix Multiplication: Parallel Processor



```
#pragma omp parallel for
    for(i = 0; i < N; ++i)
      for(j = 0; j < N; ++j)
      {
        C[i][j] = 0.0;
        for(k = 0; k < N; ++k)
          C[i][j] += A[i][k] * B[k][j];
      }
```

Parallelize outermost loop

Matrix Multiplication: Memory Hierarchy

```
for(ii = 0; ii < N; ii += L)
 for(jj = 0; jj < N; jj += L)
 {
  for(i = ii; i < ii + L; ++i)
   for(j = jj; j < jj + L ; ++j)
    C[i][j] = 0.0;
  for(kk = 0; kk < N; kk += L)
   for(i = ii; i < ii + L; ++i)
    for(j = jj; j < jj + L; ++j)
     for(k = kk; k < kk + L; ++k)
      C[i][j] += A[i][k] * B[k][j];
 }
```

Cache size greater than $3L^2$ data item size

Introduction into Dependency Analysis

- Manual optimization for particular architecture leads to **non-portable** programs
- All these optimizations should be performed by **compiler**
- But optimizations should keep program's **correctness**
- Formalization of correctness: **dependency** between program statements
- Statement S2 **depends** on statement S1:
    - Both S1 and S2 access **the same memory**
    - There is **control path** from S1 to S2
    - At least one of these accesses is **write**

Loop Nest

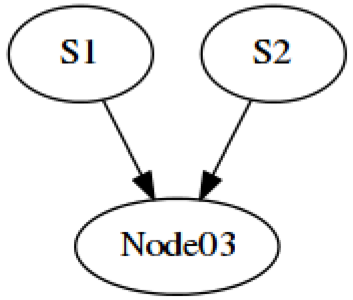
```
    for(i1 = L1; i1 < U1; i1 += ST1)
      for(i2 = L2; i2 < U2; i2 += ST2)
        ....
        for(in = Ln; in < Un; in += STn)
        {
S1:        A[f1(i1, i2, ..., in), ..., fm(i1, i2, ..., in)] = ....
S2:        ... = A[g1(i1, i2, ..., in), ..., gm(i1, i2, ..., in)];
        }
```

Normalized loop:

- Li == 0
- STi == 1

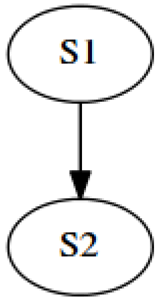
Types of Dependency

```
S1:    pi = 3.141592;
S2:    r = 5.0;
S3:    area = pi * r * r;
```



Data dependency

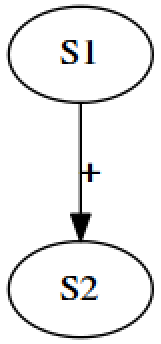
```
S1:   if(x != 0.0)
S2:         a = a / x;
```



Control dependency

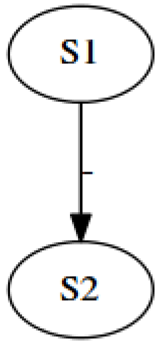
Data Dependency Classification

```
S1:   X = ...
S2:   ... = X
```



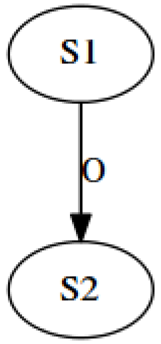
True dependence

```
S1:   ... = X
S2:   X = ...
```



Anti-dependence

```
S1:   X = ...
S2:   X = ...
```



Output dependence

Loop-carried and Loop-independent

```
for(i = 0; i < N; ++i)
{
  A[i + 1] = F[i];
  F[i + 1] = A[i];
}
```

Loop-carried dependence

```
for(i = 0; i < N; ++i)
{
  A[i] = ... ;
  ... = A[i];
}
```

Loop-independent dependence

Parallelization and Vectorization

- It is valid to convert a sequential loop to a **parallel loop** if the loop **carries no dependence**.
- A statement contained in at least one loop can be **vectorized** if the statement is **not** included in any **cycle of dependences**.

Polyhedral Optimization Frameworks

- Graphite: Gimple Represented as Polyhedra
- http://gcc.gnu.org/wiki/Graphite
- Polly: LLVM Framework for High-Level Loop and Data-Locality Optimizations
- http://polly.llvm.org/

Bibliography

1. Steven Muchnick. Advanced Compiler Design and Implementation, 1997
2. Randy Allen, Ken Kennedy. Optimizing Compilers for Modern Architectures: A Dependence-based Approach, 2001.
3. D. Loveman, R. Faneuf. Program Optimization – Theory and Practice. Massachusetts Computer Associates Inc.
4. A. Appel. SSA is Functional Programming.
5. R. Cytron & al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.
6. C. McConnell, R. Johnson. Using SSA Form in a Code Optimizer.
7. R. Kennedy & al. Partial Redundancy Elimination in SSA Form
8. K. Cooper & al. Operator Strength Reduction

Thank you!