GCC for Parallelization

Uday Khedker, Supratim Biswas, Prashant Rawat

GCC Resource Center,
Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

January 2010

Outline

- GCC: The Great Compiler Challenge
- Configuration and Building
- Introduction to parallelization and vectorization
- Observing parallelization and vectorization performed by GCC
- Activities of GCC Resource Center

About this Tutorial

- Expected background
  Some compiler background, no knowledge of GCC or parallelization
- Takeaways: After this tutorial you will be able to
  - Appreciate the GCC architecture
  - Configure and build GCC
  - Observe what GCC does to your programs
  - Study parallelization transformations done by GCC
  - Get a feel of the strengths and limitations of GCC

The Scope of this Tutorial

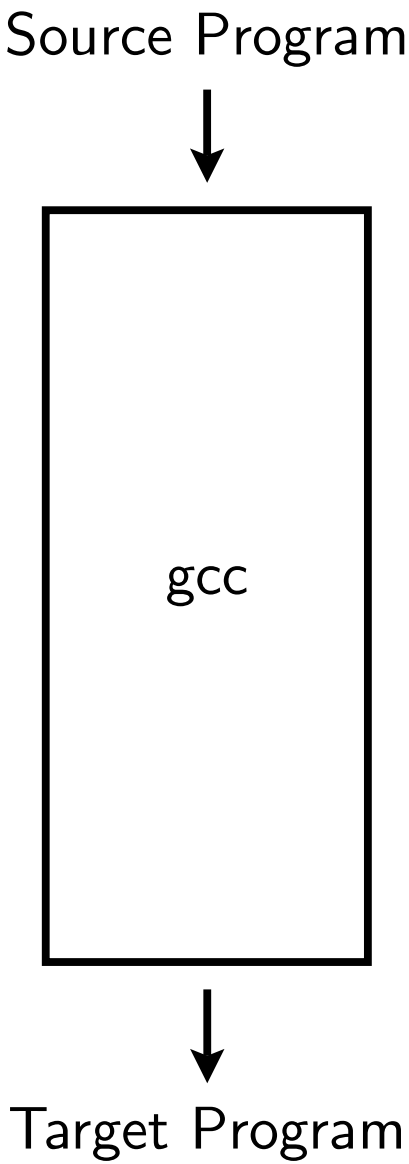
- What this tutorial does not address
  - Code or data structures of gcc
  - Algorithms used for parallelization and vectoriation
  - Machine level issues related to parallelization and vectoriation
  - Advanced theory such as polyhydral approach
  - Research issues
- What this tutorial addresses

Basics of Discovering Parallelism using GCC

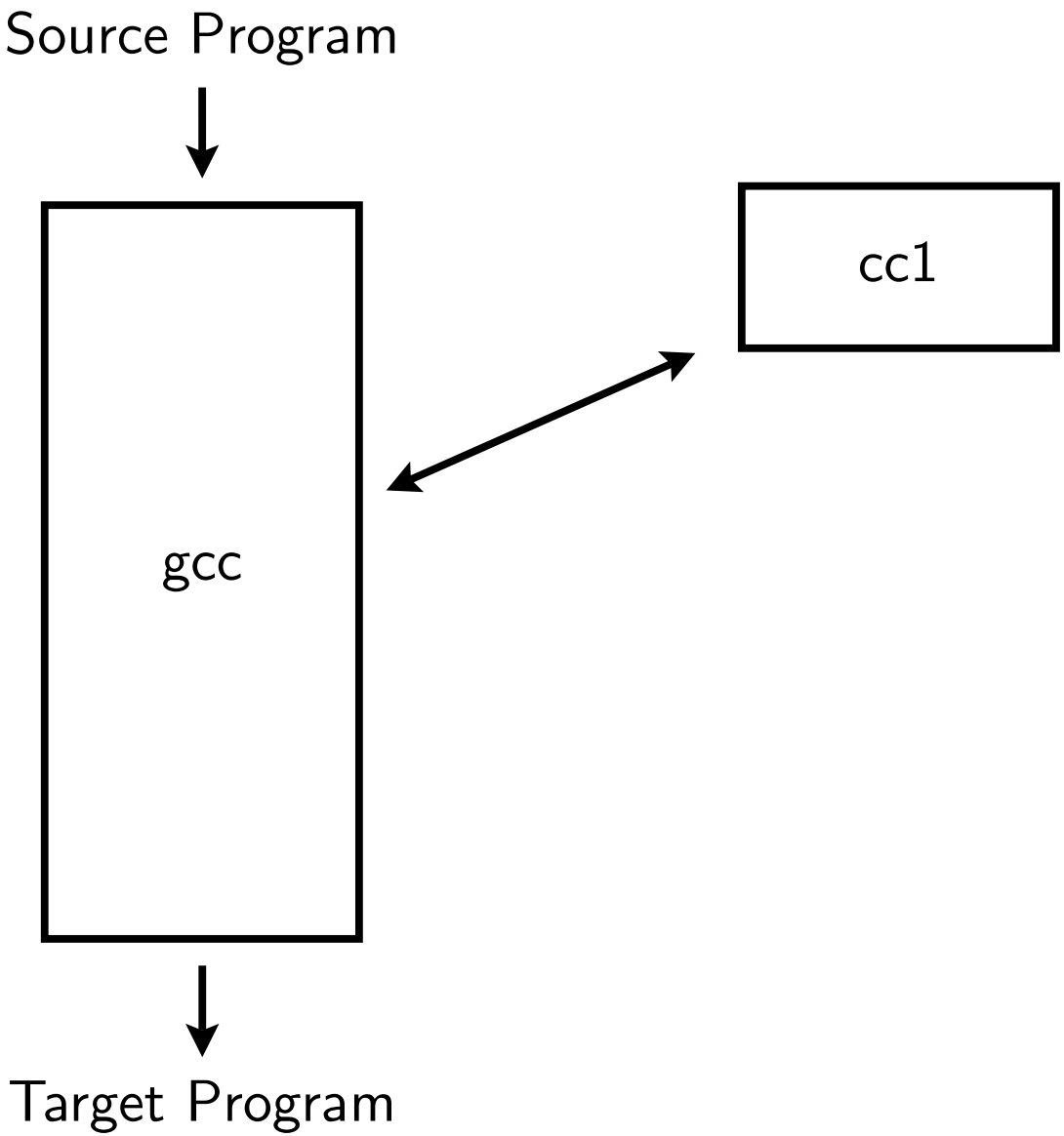
Part 1

GCC $\equiv$ *The Great Compiler Challenge*

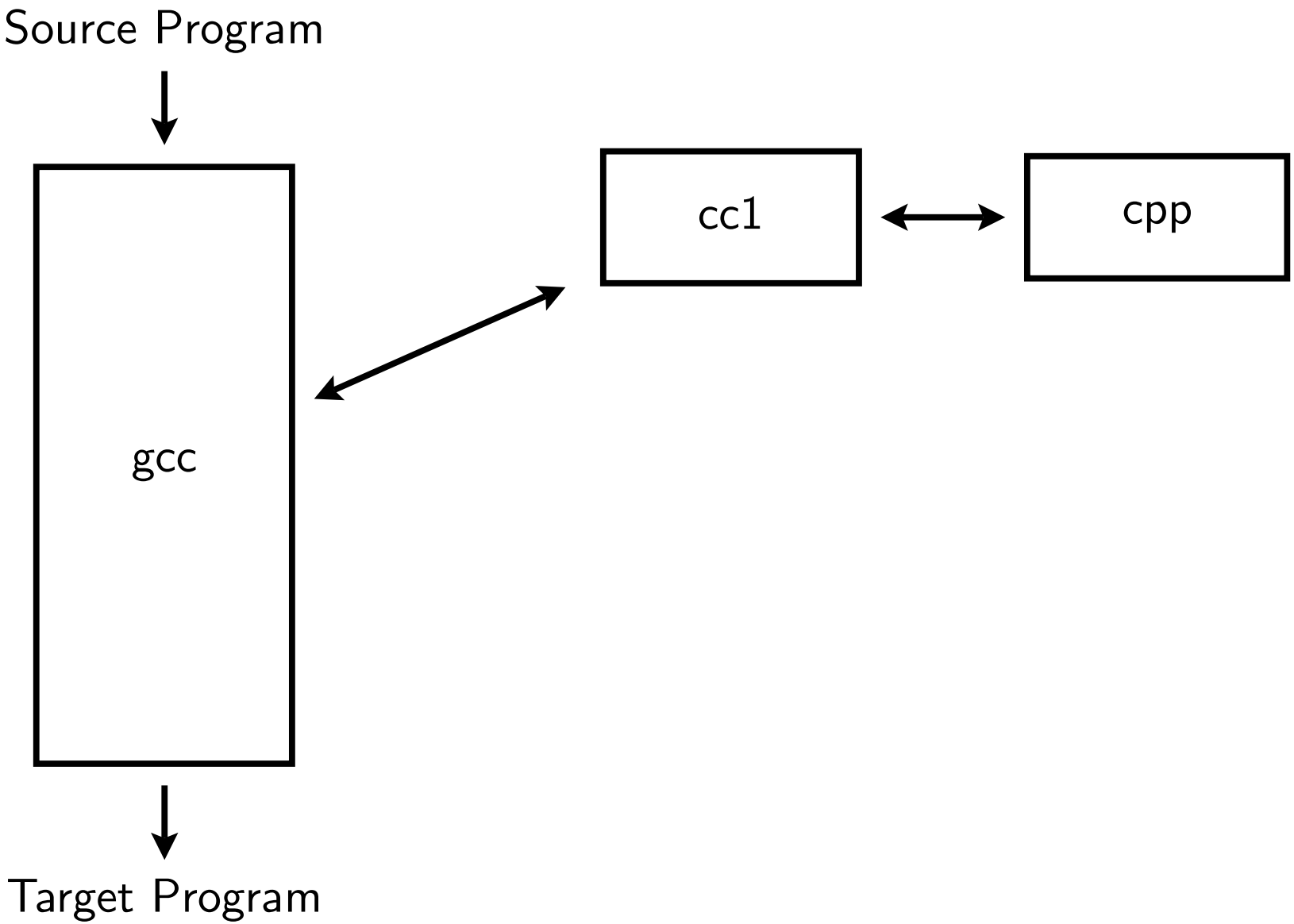
The Gnu Tool Chain



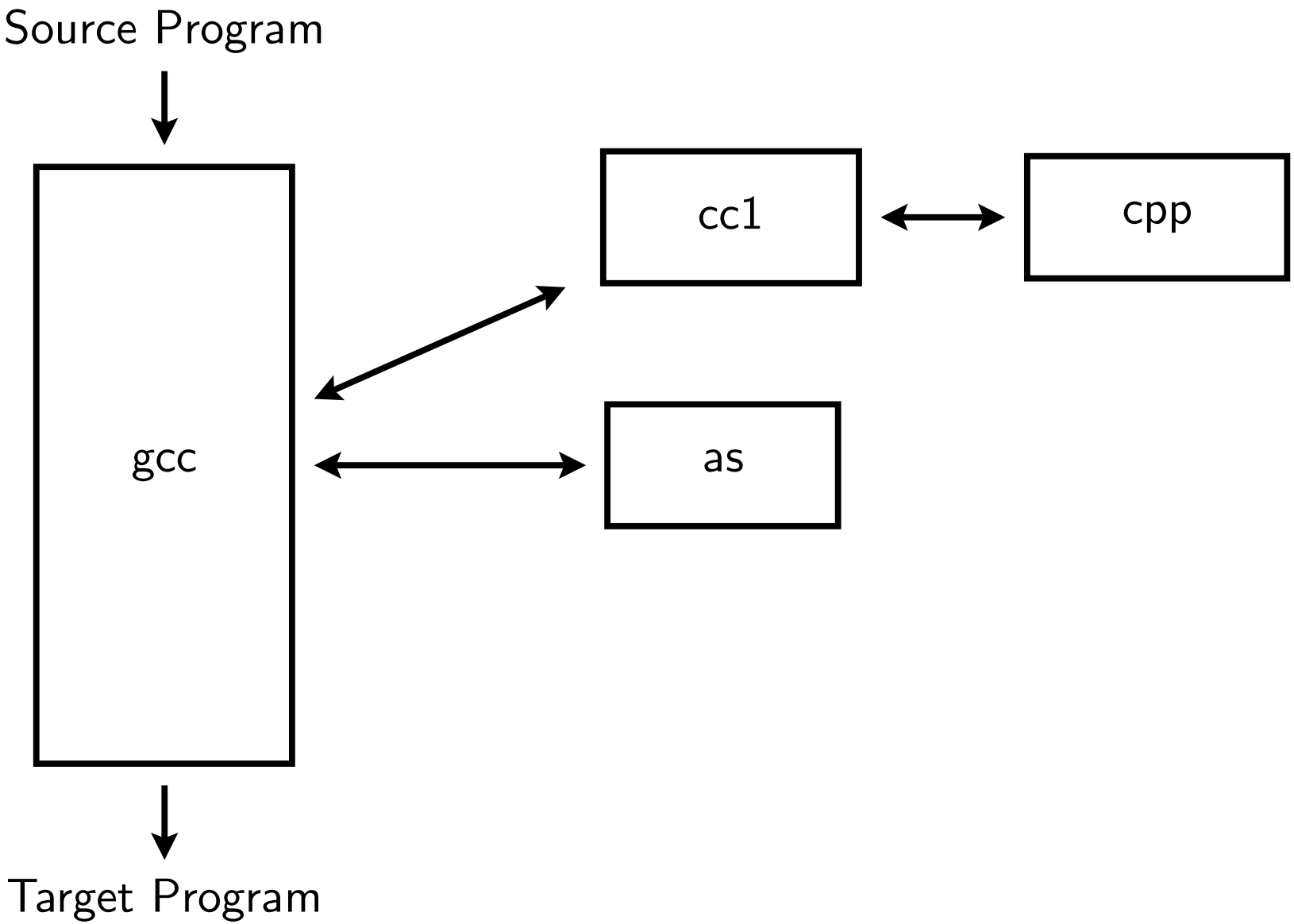
The Gnu Tool Chain



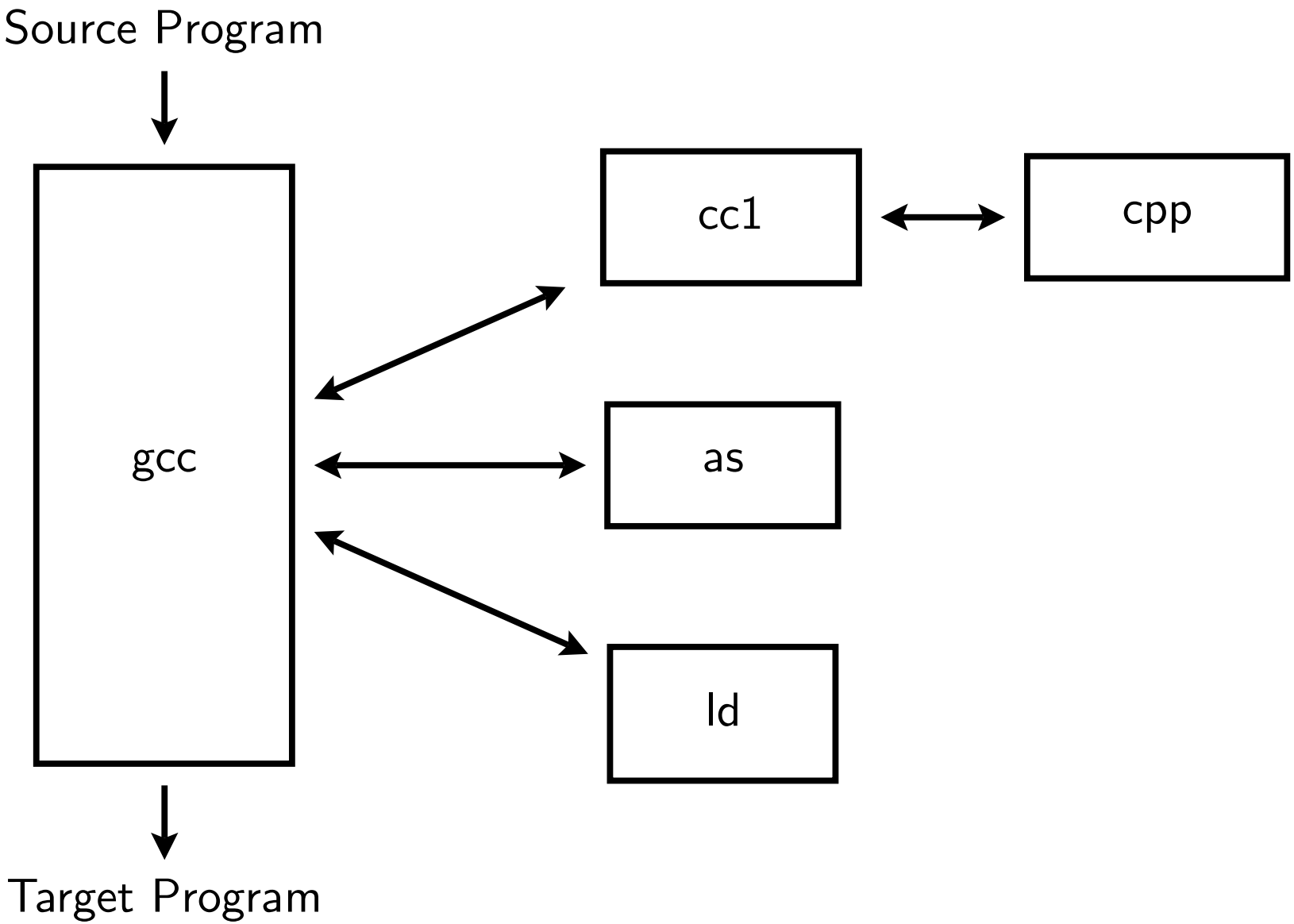
The Gnu Tool Chain



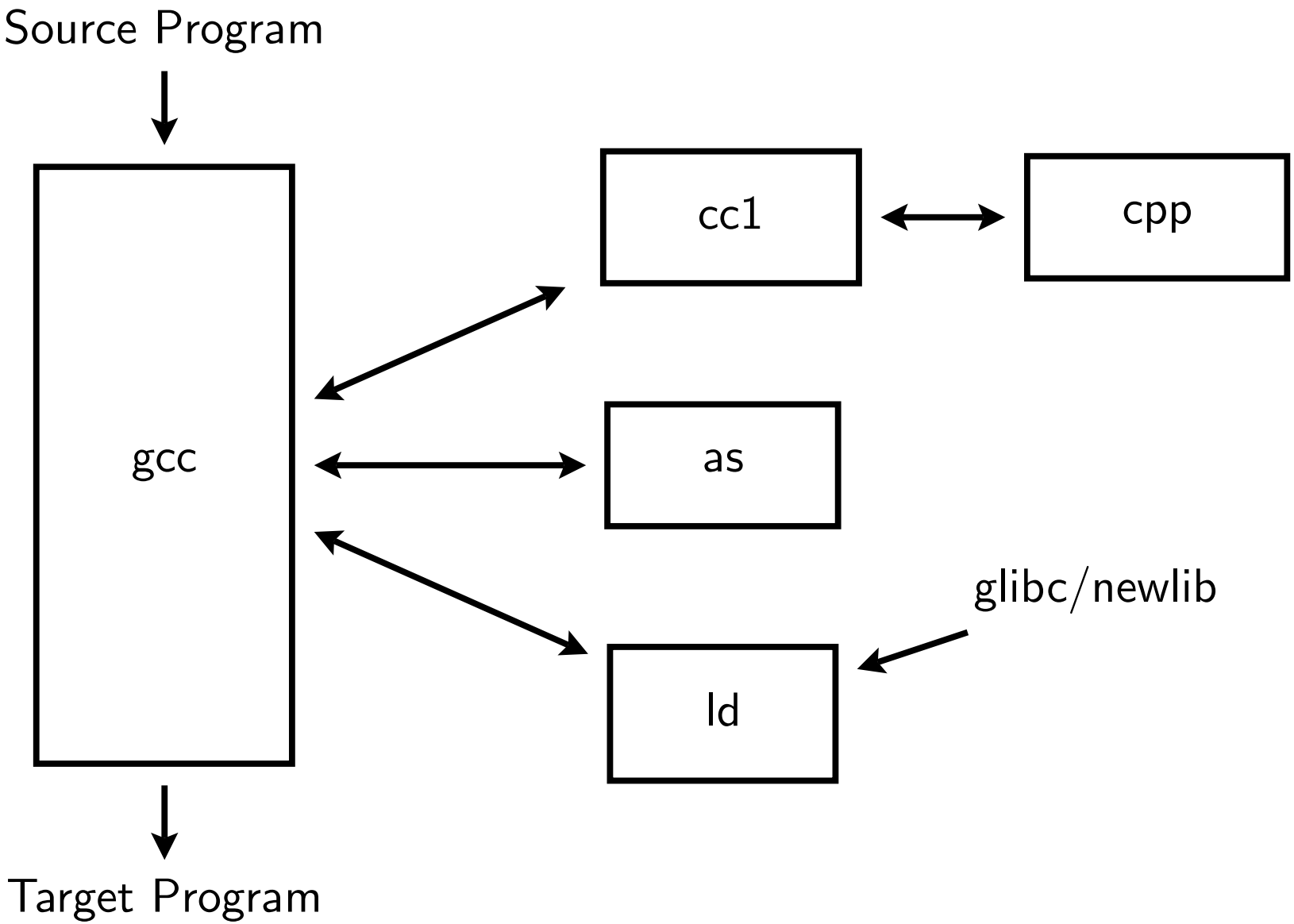
The Gnu Tool Chain



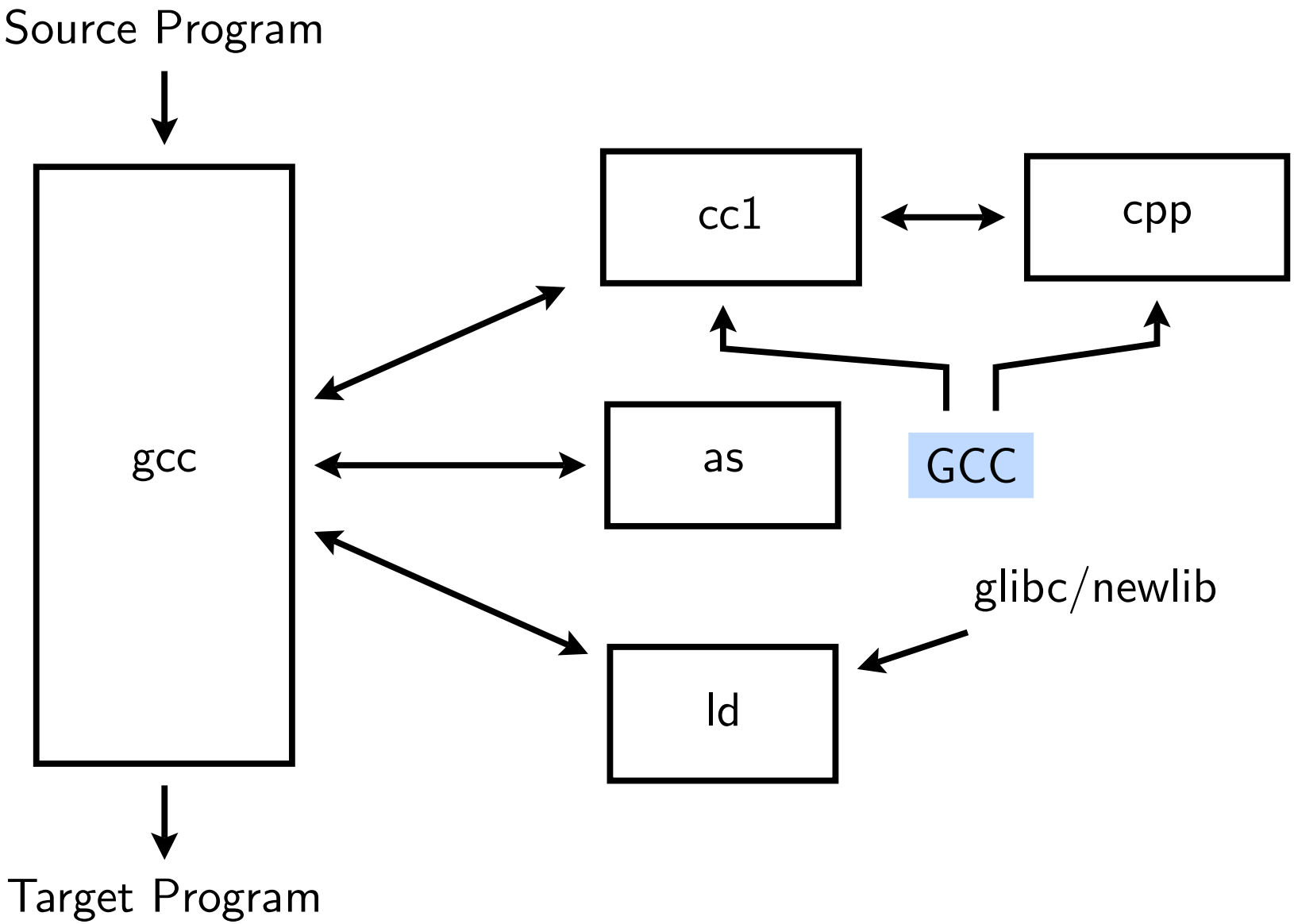
The Gnu Tool Chain



The Gnu Tool Chain



The Gnu Tool Chain



Why is Understanding GCC Difficult?

Some of the obvious reasons:

- Comprehensiveness

  GCC is a production quality framework in terms of completeness and practical usefulness

- Open development model

  Could lead to heterogeneity. Design flaws may be difficult to correct

- Rapid versioning

  GCC maintenance is a race against time. Disruptive corrections are difficult

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86),
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC,
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa,
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10, TIGCC (m68k variant),

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10, TIGCC (m68k variant), Z8000,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10, TIGCC (m68k variant), Z8000, PIC24/dsPIC,

Comprehensiveness of GCC 4.3.1: Wide Applicability

- Input languages supported:
  C, C++, Objective-C, Objective-C++, Java, Fortran, and Ada
- Processors supported in standard releases:
  - Common processors:
    Alpha, ARM, Atmel AVR, Blackfin, HC12, H8/300, IA-32 (x86), x86-64, IA-64, Motorola 68000, MIPS, PA-RISC, PDP-11, PowerPC, R8C/M16C/M32C, SPU, System/390/zSeries, SuperH, SPARC, VAX
  - Lesser-known target processors:
    A29K, ARC, ETRAX CRIS, D30V, DSP16xx, FR-30, FR-V, Intel i960, IP2000, M32R, 68HC11, MCORE, MMIX, MN10200, MN10300, Motorola 88000, NS32K, ROMP, Stormy16, V850, Xtensa, AVR32
  - Additional processors independently supported:
    D10V, LatticeMico32, MeP, Motorola 6809, MicroBlaze, MSP430, Nios II and Nios, PDP-10, TIGCC (m68k variant), Z8000, PIC24/dsPIC, NEC SX architecture

Comprehensiveness of GCC 4.3.1: Size

Source Lines	Number of lines in the main source	2,029,115
	Number of lines in libraries	1,546,826
Directories	Number of subdirectories	3527
Files	Total number of files	57825
	C source files	19834
	Header files	9643
	C++ files	3638
	Java files	6289
	Makefiles and Makefile templates	163
	Configuration scripts	52
	Machine description files	186

(Line counts estimated by the program `sloccount` by David A. Wheeler)

Open Source and Free Software Development Model

The Cathedral and the Bazaar [Eric S Raymond, 1997]

Open Source and Free Software Development Model

The Cathedral and the Bazaar [Eric S Raymond, 1997]

- Cathedral: Total Centralized Control
  Design, implement, test, release

Open Source and Free Software Development Model

The Cathedral and the Bazaar [Eric S Raymond, 1997]

- Cathedral: Total Centralized Control
  Design, implement, test, release
- Bazaar: Total Decentralization
  Release early, release often, make users partners in software development

Open Source and Free Software Development Model

The Cathedral and the Bazaar [Eric S Raymond, 1997]

- Cathedral: Total Centralized Control
  Design, implement, test, release
- Bazaar: Total Decentralization
  Release early, release often, make users partners in software development

  "Given enough eyeballs, all bugs are shallow"

Open Source and Free Software Development Model

The Cathedral and the Bazaar [Eric S Raymond, 1997]

- Cathedral: Total Centralized Control
  Design, implement, test, release
- Bazaar: Total Decentralization
  Release early, release often, make users partners in software development

  "Given enough eyeballs, all bugs are shallow"
  Code errors, logical errors, and architectural errors

Open Source and Free Software Development Model

The Cathedral and the Bazaar [Eric S Raymond, 1997]

- Cathedral: Total Centralized Control
  Design, implement, test, release
- Bazaar: Total Decentralization
  Release early, release often, make users partners in software development

  "Given enough eyeballs, all bugs are shallow"
  Code errors, logical errors, and architectural errors

A combination of the two seems more sensible

The Current Development Model of GCC

GCC follows a combination of the Cathedral and the Bazaar approaches

- GCC Steering Committee: Free Software Foundation has given charge
  - Major policy decisions
  - Handling Administrative and Political issues
- Release Managers:
  - Coordination of releases
- Maintainers:
  - Usually area/branch/module specific
  - Responsible for design and implementation
  - Take help of reviewers to evaluate submitted changes

Why is Understanding GCC Difficult?

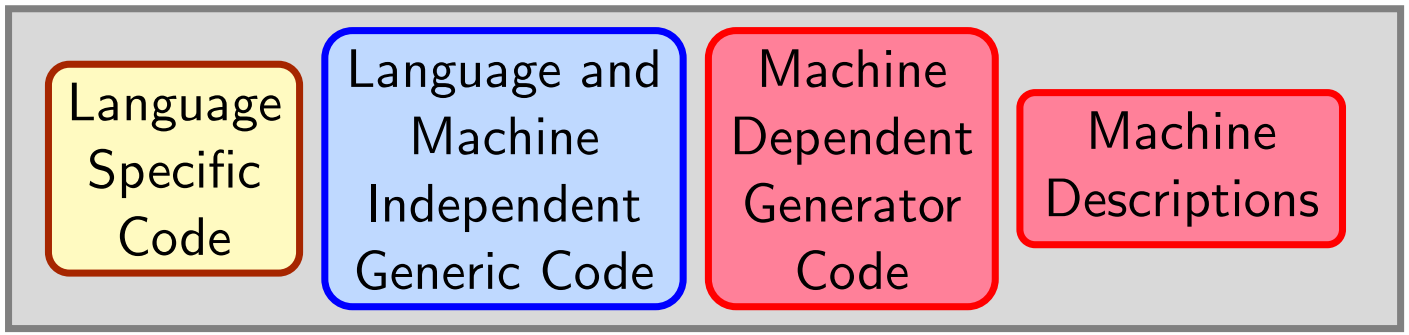
Deeper reason: GCC is not a *compiler* but a *compiler generation framework*

There are two distinct gaps that need to be bridged:

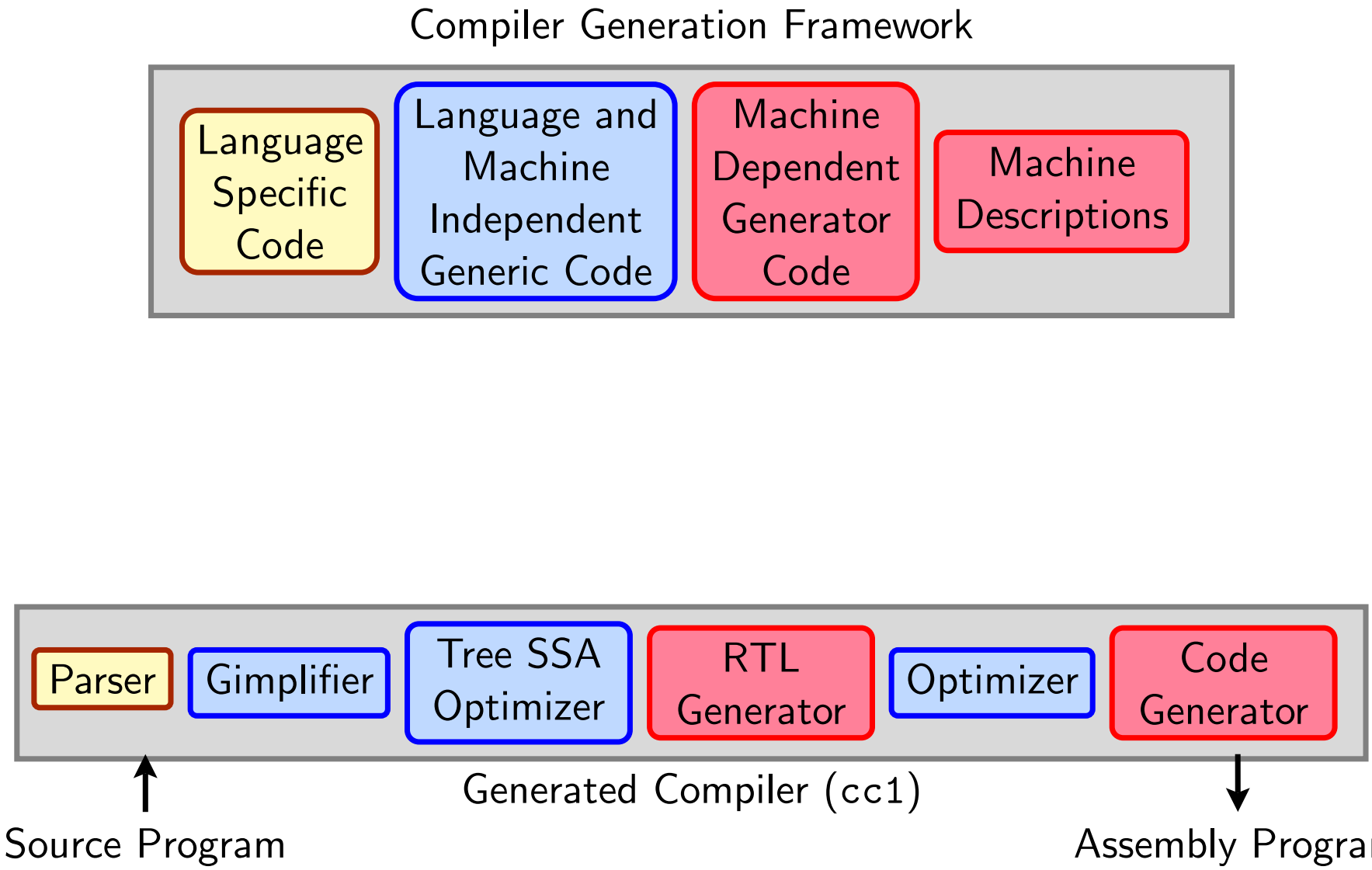
- Input-output of the generation framework: The target specification and the generated compiler
- Input-output of the generated compiler: A source program and the generated assembly program

The Architecture of GCC

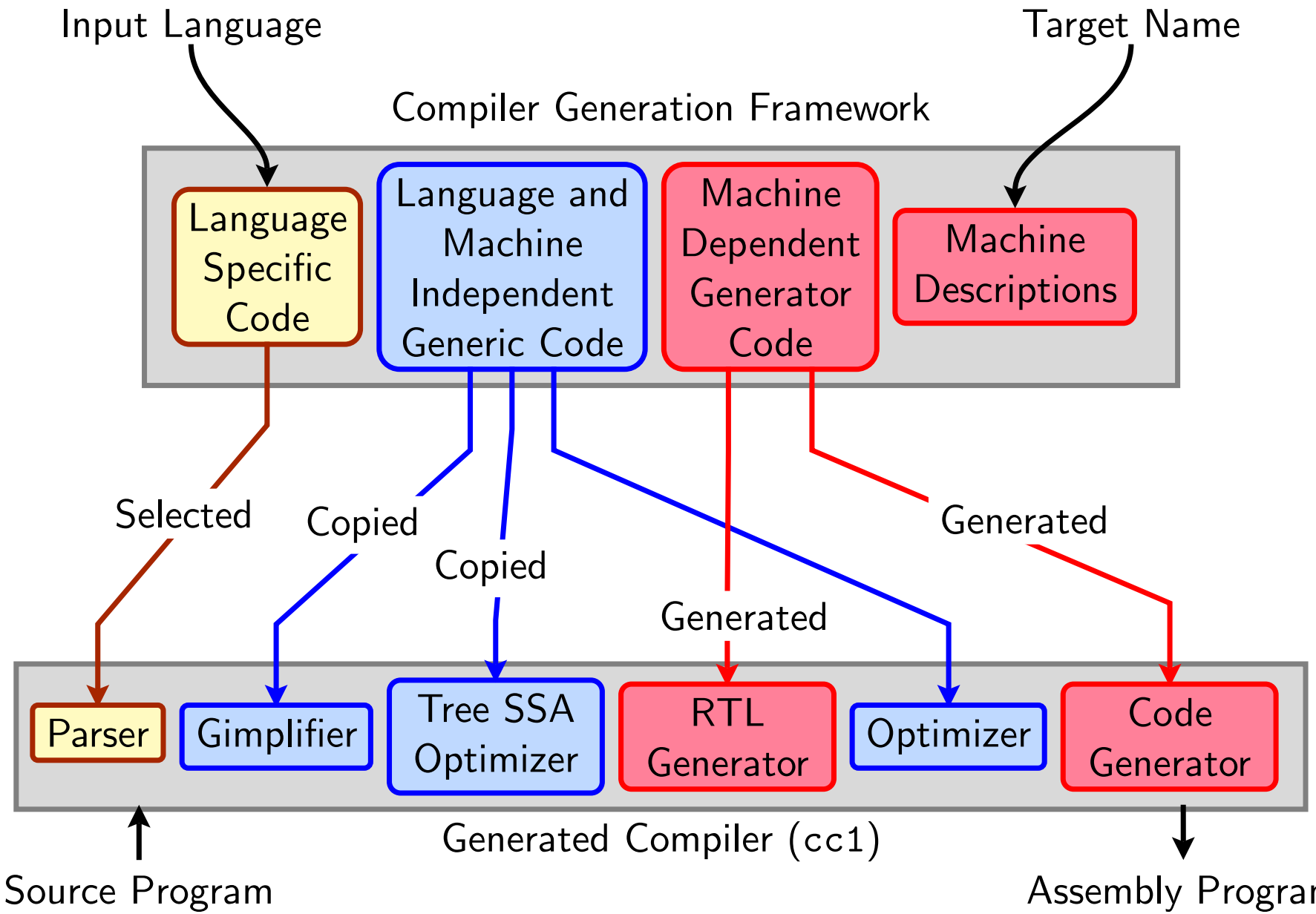
Compiler Generation Framework



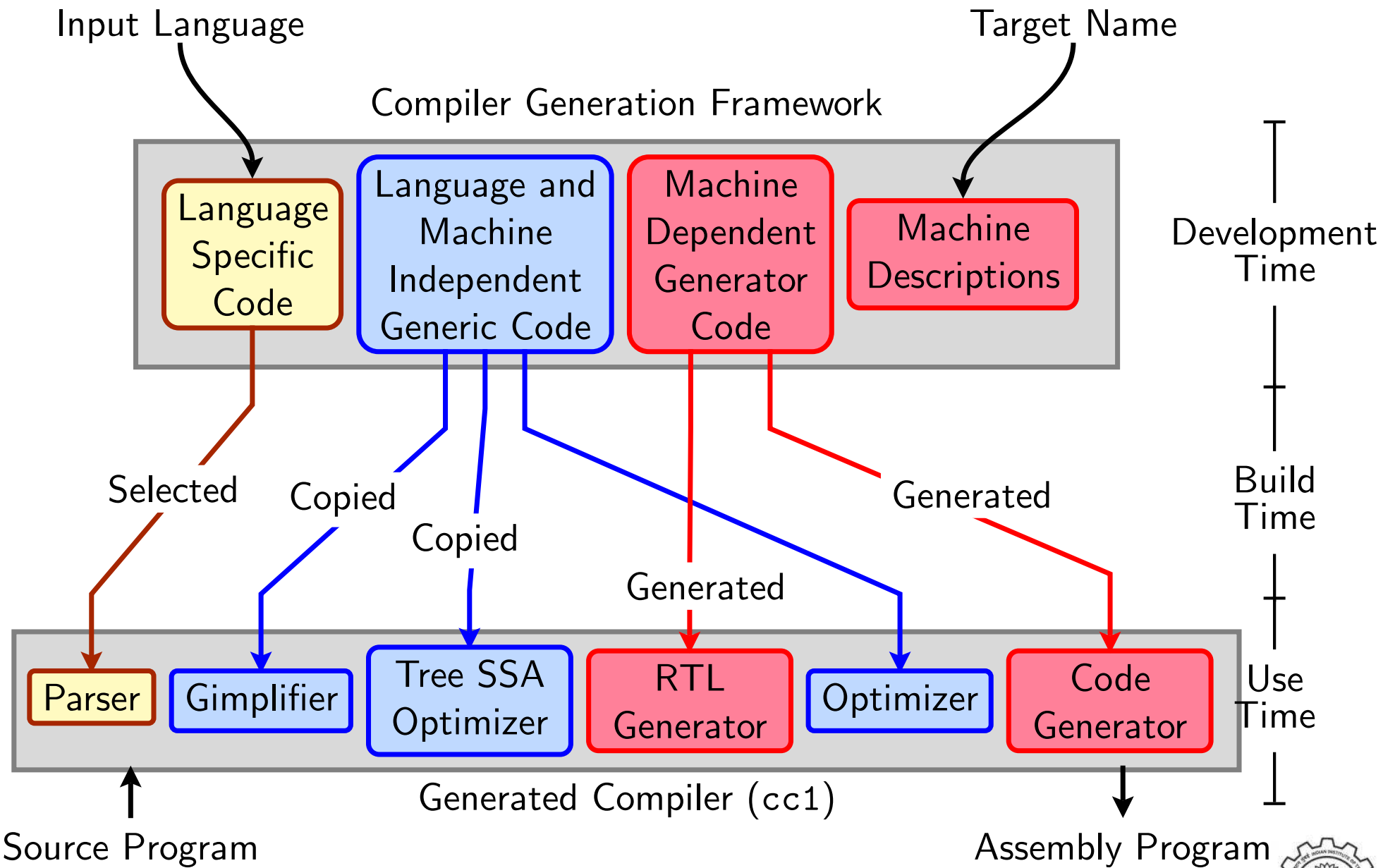
The Architecture of GCC



The Architecture of GCC



The Architecture of GCC



An Example of The Generation Related Gap

- Predicate function for invoking the loop distribution pass

```
static bool
gate_tree_loop_distribution (void)
{
  return flag_tree_loop_distribution != 0;
}
```


An Example of The Generation Related Gap

- Predicate function for invoking the loop distribution pass

```
static bool
gate_tree_loop_distribution (void)
{
  return flag_tree_loop_distribution != 0;
}
```

- There is no declaration of or assignment to variable `flag_tree_loop_distribution` in the entire source!

An Example of The Generation Related Gap

- Predicate function for invoking the loop distribution pass

```
static bool
gate_tree_loop_distribution (void)
{
  return flag_tree_loop_distribution != 0;
}
```

- There is no declaration of or assignment to variable `flag_tree_loop_distribution` in the entire source!
- It is described in `common.opt` as follows

```
ftree-loop-distribution
Common Report Var(flag_tree_loop_distribution) Optimization
Enable loop distribution on trees
```


An Example of The Generation Related Gap

- Predicate function for invoking the loop distribution pass

```
static bool
gate_tree_loop_distribution (void)
{
  return flag_tree_loop_distribution != 0;
}
```

- There is no declaration of or assignment to variable `flag_tree_loop_distribution` in the entire source!
- It is described in `common.opt` as follows

```
ftree-loop-distribution
Common Report Var(flag_tree_loop_distribution) Optimization
Enable loop distribution on trees
```

- The required C statements are generated during the build

Another Example of The Generation Related Gap

Locating the `main` function in the directory `gcc-4.4.2/gcc` using cscope

Another Example of The Generation Related Gap

Locating the `main` function in the directory `gcc-4.4.2/gcc` using cscope

```
  File              Line
0 collect2.c         766 main (int argc, char **argv)
1 fix-header.c      1074 main (int argc, char **argv)
2 fp-test.c           85 main (void )
3 gcc.c             6216 main (int argc, char **argv)
4 gcov-dump.c         76 main (int argc ATTRIBUTE_UNUSED, char **argv
5 gcov-iov.c          29 main (int argc, char **argv)
6 gcov.c             355 main (int argc, char **argv)
7 gen-protos.c       130 main (int argc ATTRIBUTE_UNUSED, char **argv
8 genattr.c           89 main (int argc, char **argv)
9 genattrtab.c      4438 main (int argc, char **argv)
a genautomata.c     9321 main (int argc, char **argv)
b gencheckSum.c       65 main (int argc, char ** argv)
c gencodes.c          51 main (int argc, char **argv)
d genconditions.c    209 main (int argc, char **argv)
e genconfig.c        261 main (int argc, char **argv)
f genconstants.c      50 main (int argc, char **argv)
```

Another Example of The Generation Related Gap

Locating the `main` function in the directory `gcc-4.4.2/gcc` using cscope

```
g genemit.c          820 main (int argc, char **argv)
h genextract.c       394 main (int argc, char **argv)
i genflags.c         231 main (int argc, char **argv)
j gengenrtl.c        350 main (int argc, char **argv)
k gengtype.c        3584 main (int argc, char **argv)
l genmddeps.c         45 main (int argc, char **argv)
m genmodes.c        1376 main (int argc, char **argv)
n genopinit.c        472 main (int argc, char **argv)
o genoutput.c       1005 main (int argc, char **argv)
p genpeep.c          353 main (int argc, char **argv)
q genpreds.c        1399 main (int argc, char **argv)
r genrecog.c        2718 main (int argc, char **argv)
s main.c              33 main (int argc, char **argv)
t mips-tdump.c      1393 main (int argc, char **argv)
u mips-tfile.c       655 main (void )
v mips-tfile.c      4690 main (int argc, char **argv)
w protoize.c        4373 main (int argc, char **const argv)
```

The GCC Challenge: Poor Retargetability Mechanism

- Symptom of poor retargetability mechanism

  Large size of specifications

The GCC Challenge: Poor Retargetability Mechanism

- Symptom of poor retargetability mechanism

  Large size of specifications

- Size in terms of line counts

Files	i386	mips
*.md	35766	12930
*.c	28643	12572
*.h	15694	5105

Part 2

Meeting the GCC Challenge

Meeting the GCC Challenge

Goal of Understanding	Methodology	Needs Examining		
		Makefiles	Source	MD
Translation sequence of programs	Gray box probing	No	No	No
Build process	Customizing the configuration and building	Yes	No	No
Retargetability issues and machine descriptions	Incremental construction of machine descriptions	No	No	Yes
IR data structures and access mechanisms	Adding passes to massage IRs	No	Yes	Yes
Retargetability mechanism		Yes	Yes	Yes

What is Gray Box Probing of GCC?

- Black Box probing:
  Examining only the input and output relationship of a system
- White Box probing:
  Examining internals of a system for a given set of inputs
- Gray Box probing:
  Examining input and output of various components/modules
  - Overview of translation sequence in GCC
  - Overview of intermediate representations
  - Intermediate representations of programs across important phases

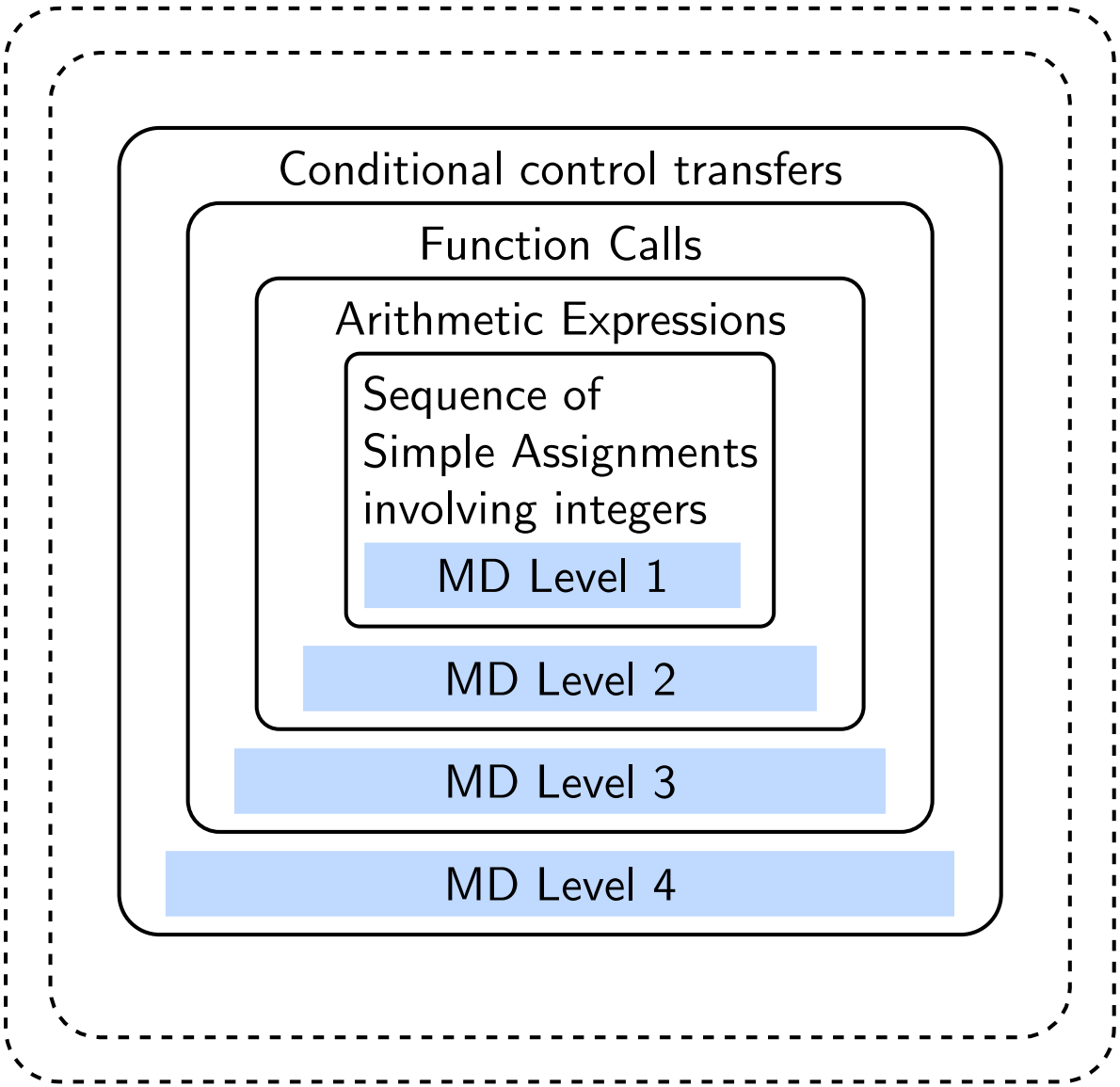
Customizing the Configuration and Build Process

- Creating only `cc1`
- Creating bare metal cross build
  Complete tool chain without OS support
- Creating cross build with OS support

Incremental Construction of Machine Descriptions

- Define different levels of source language
- Identify the minimal set of features in the target required to support each level
- Identify the minimal information required in the machine description to support each level
  - Successful compilation of any program, and
  - correct execution of the generated assembly program
- Interesting observations
  - It is the increment in the source language which results in understandable increments in machine descriptions rather than the increment in the target architecture
  - If the levels are identified properly, the increments in machine descriptions are monotonic

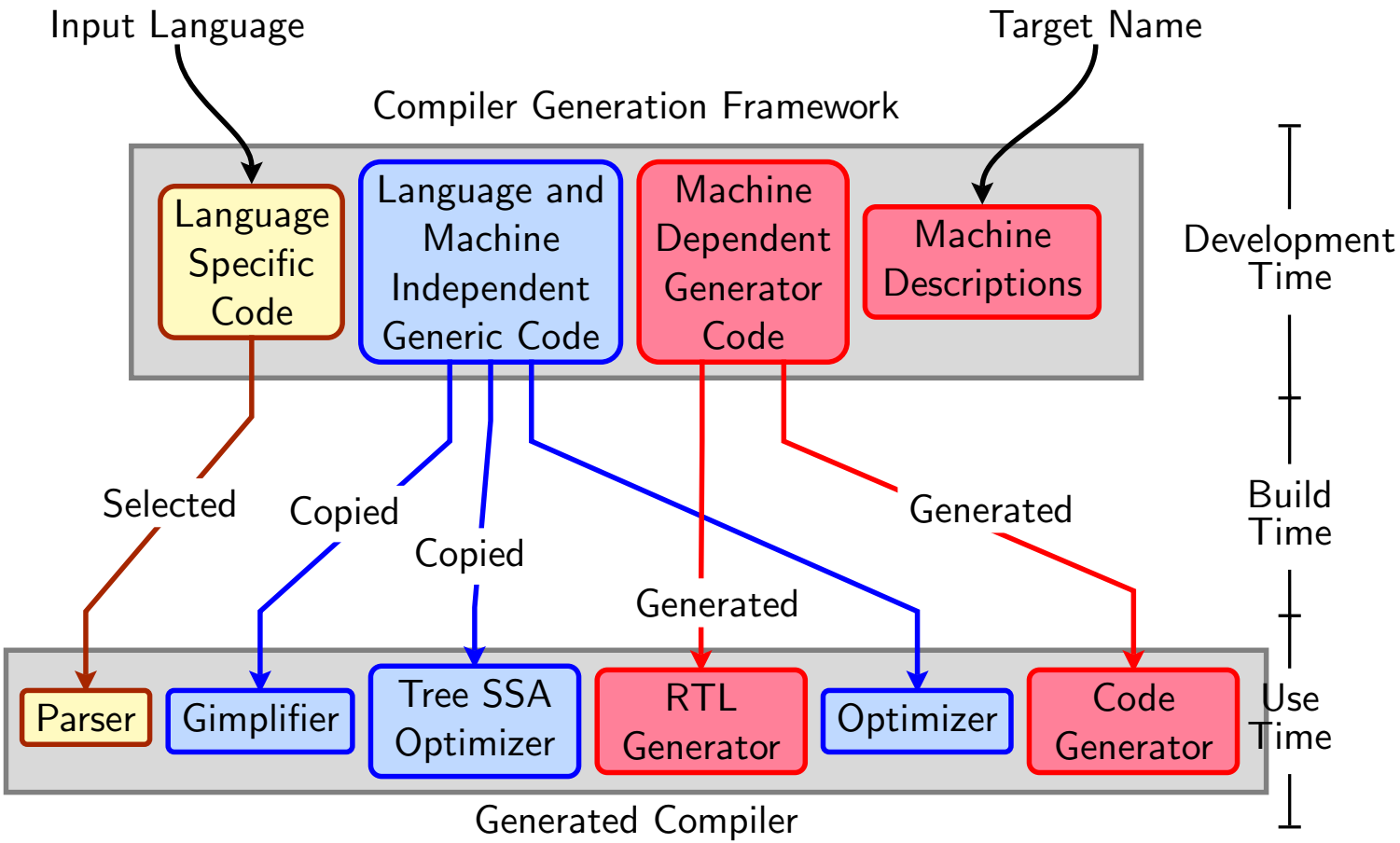
Incremental Construction of Machine Descriptions



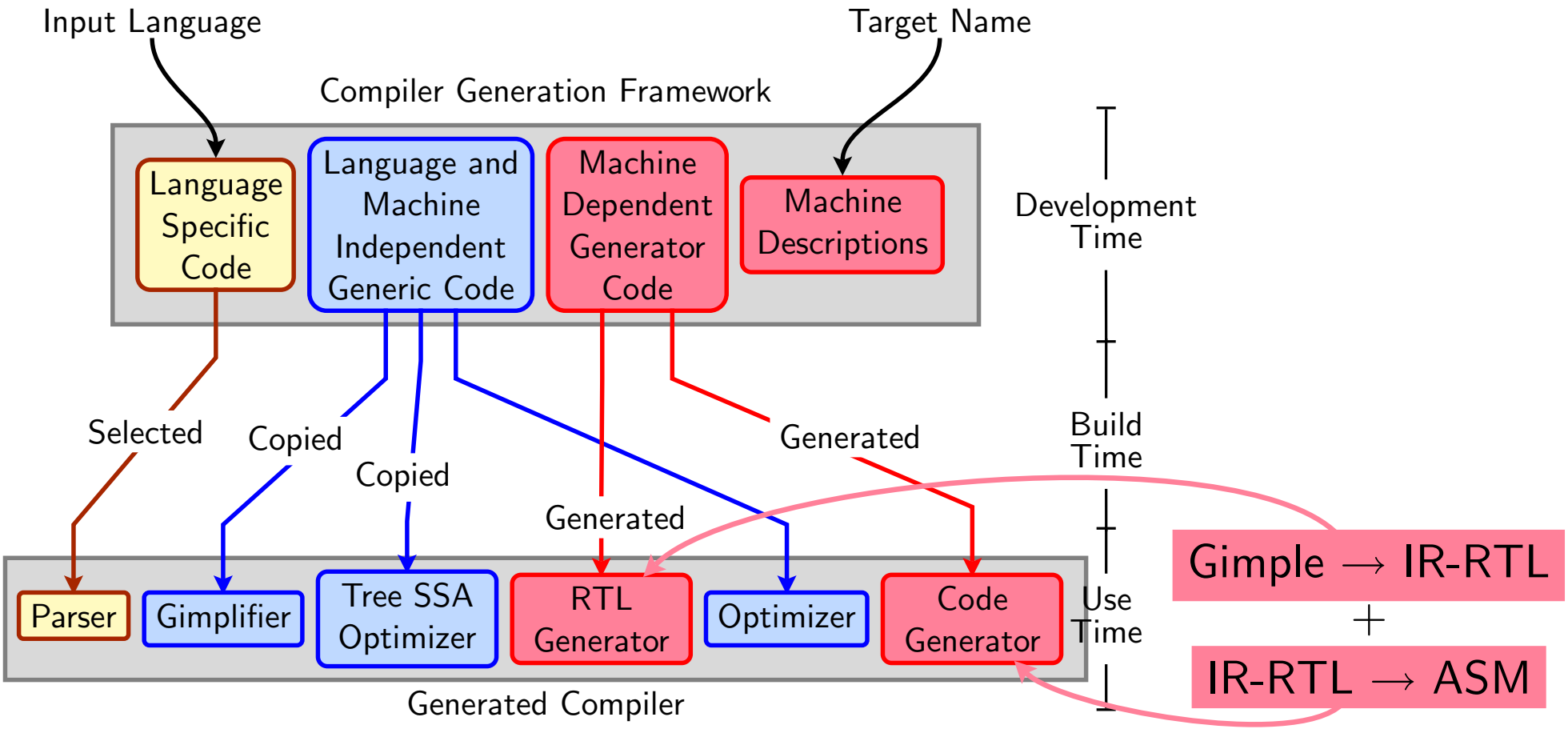
Adding Passes to Massage IRs

- Understanding the pass structure
- Understanding the mechanisms of traversing a call graph and a control flow graph
- Understanding how to access the data structures of IRs
- Simple exercises such as:
  - Count the number of copy statements in a program
  - Count the number of variables declared "const" in the program
  - Count the number of occurances of arithmatic operators in the program
  - Count the number of references to global variables in the program

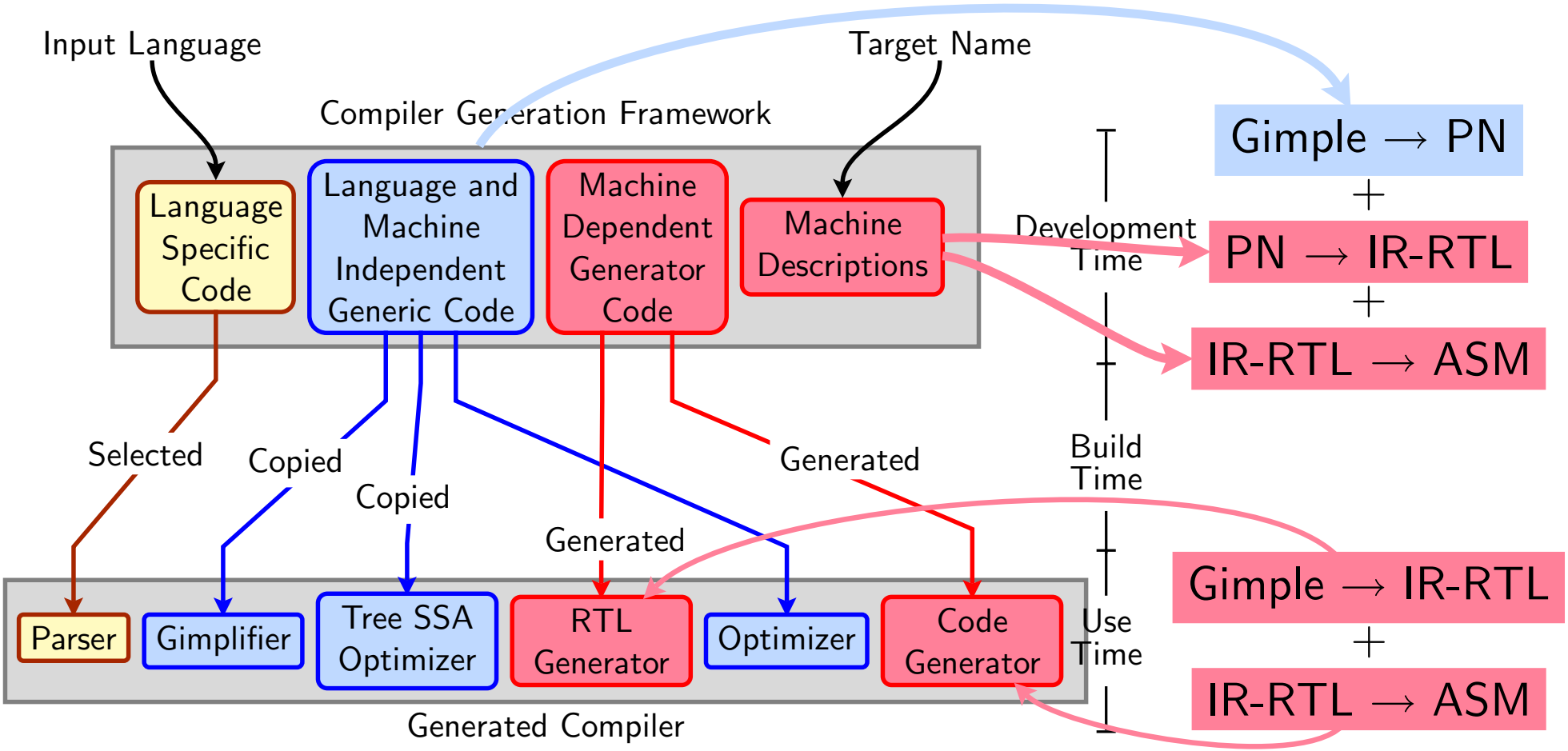
Understanding the Retargetability Mechanism



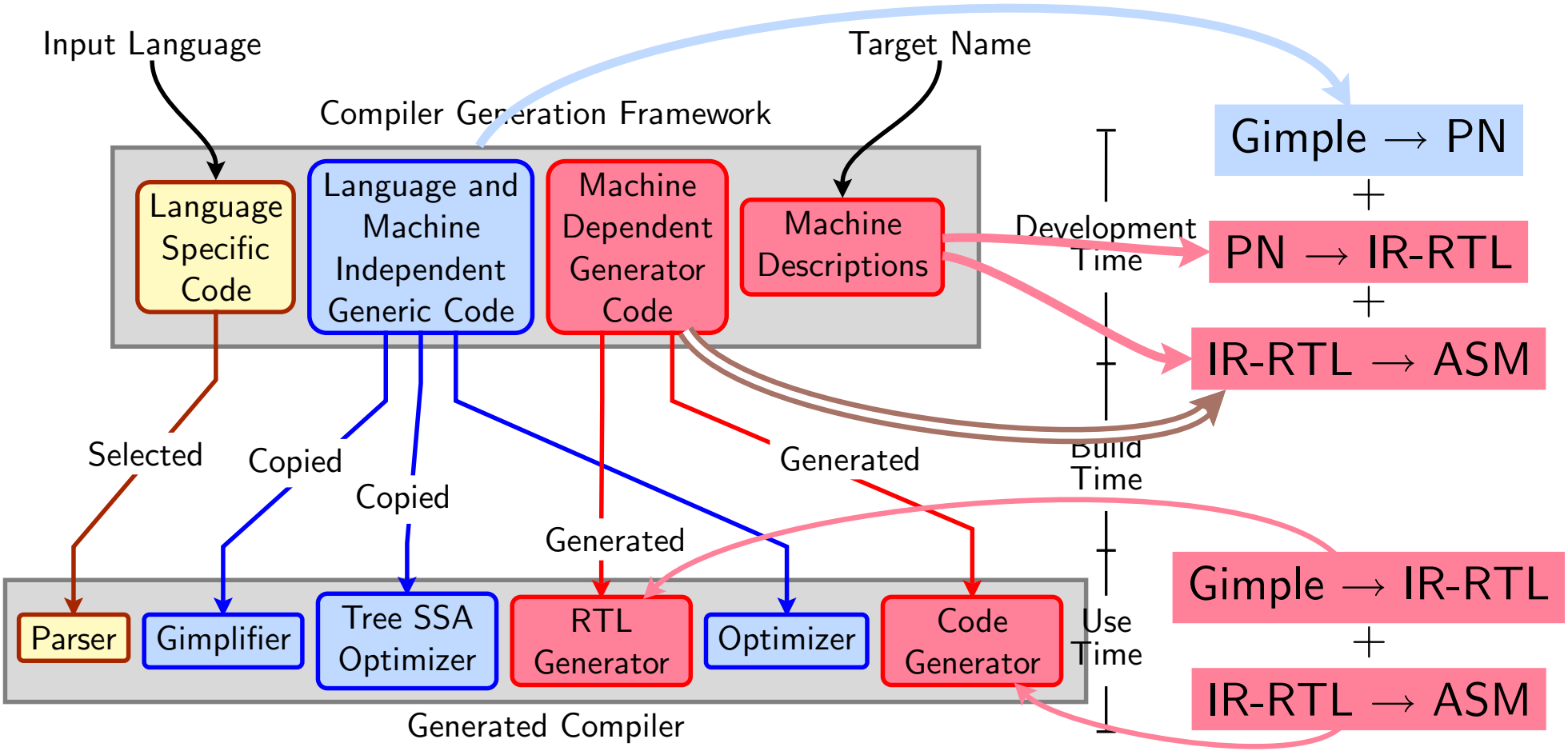
Understanding the Retargetability Mechanism



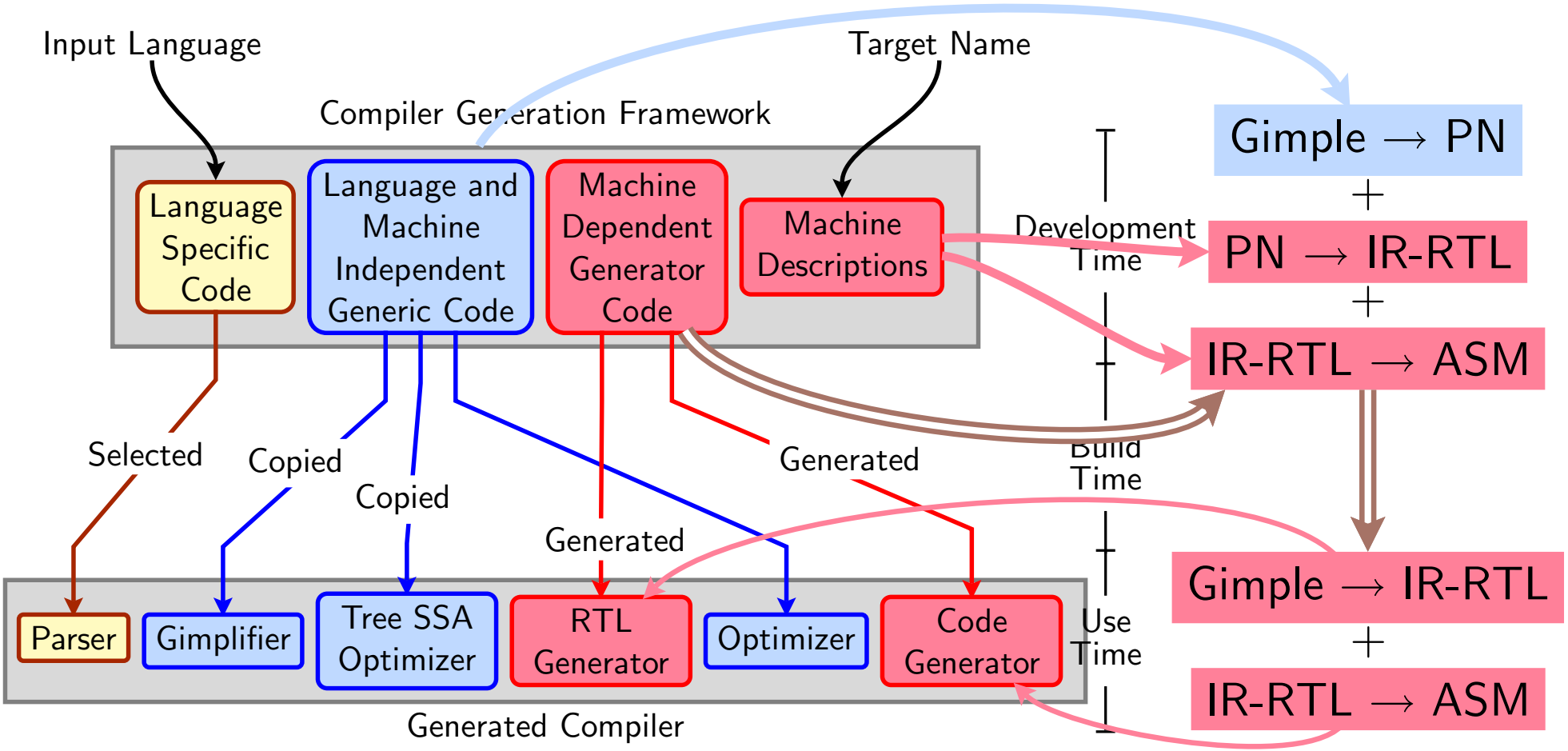
Understanding the Retargetability Mechanism



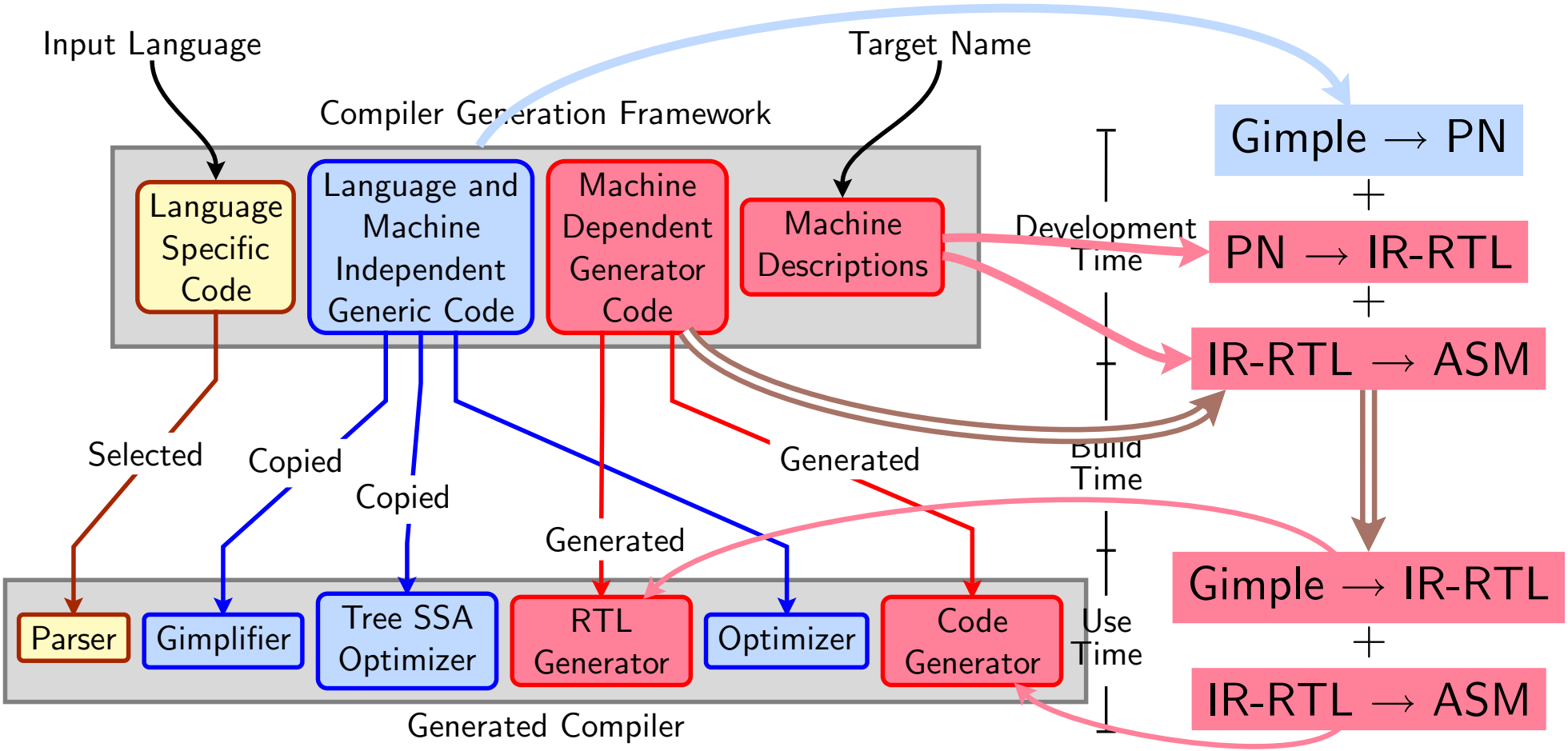
Understanding the Retargetability Mechanism



Understanding the Retargetability Mechanism



Understanding the Retargetability Mechanism



Many more details need to be explained ☹

Our Current Focus

Goal of Understanding	Methodology	Needs Examining		
		Makefiles	Source	MD
Translation sequence of programs	Gray box probing	No	No	No
Build process	Customizing the configuration and building	Yes	No	No
Retargetability issues and machine descriptions	Incremental construction of machine descriptions	No	No	Yes
IR data structures and access mechanisms	Adding passes to massage IRs	No	Yes	Yes
Retargetability mechanism		Yes	Yes	Yes

Part 3

Configuration and Building

Configuration and Building: Outline

- Code Organization of GCC
- Configuration and Building
- Native build Vs. cross build
- Testing GCC

GCC Code Organization

Logical parts are:

- Build configuration files
- Front end + generic + generator sources
- Back end specifications
- Emulation libraries
  (eg. `libgcc` to emulate operations not supported on the target)
- Language Libraries (except C)
- Support software (e.g. garbage collector)

GCC Code Organization

Front End Code

- Source language dir: `$(SOURCE)/<lang dir>`
- Source language dir contains
  - Parsing code (Hand written)
  - Additional AST/Generic nodes, if any
  - Interface to Generic creation

  Except for C – which is the "native" language of the compiler

  C front end code in: `$(SOURCE)/gcc`

Optimizer Code and Back End Generator Code

- Source language dir: `$(SOURCE)/gcc`

Back End Specification

- `$(SOURCE)/gcc/config/<target dir>/`
  Directory containing back end code
- Two main files: `<target>.h` and `<target>.md`,
  e.g. for an `i386` target, we have
  `$(SOURCE)/gcc/config/i386/i386.md` and
  `$(SOURCE)/gcc/config/i386/i386.h`
- Usually, also `<target>.c` for additional processing code
  (e.g. `$(SOURCE)/gcc/config/i386/i386.c`)
- Some additional files

Configuration

Preparing the GCC source for local adaptation:

- The platform on which it will be compiled
- The platform on which the generated compiler will execute
- The platform for which the generated compiler will generate code
- The directory in which the source exists
- The directory in which the compiler will be generated
- The directory in which the generated compiler will be installed
- The input languages which will be supported
- The libraries that are required
- etc.

Pre-requisites for Configuring and Building GCC 4.4.2

- ISO C90 Compiler / GCC 2.95 or later
- GNU bash: for running configure etc
- Awk: creating some of the generated source file for GCC
- bzip/gzip/untar etc. For unzipping the downloaded source file
- GNU make version 3.8 (or later)
- GNU Multiple Precision Library (GMP) version 4.2 (or later)
- MPFR Library version 2.3.2 (or later)
  (multiple precision floating point with correct rounding)
- MPC Library version 0.8.0 (or later)
- Parma Polyhedra Library (PPL) version 0.10
- CLooG-PPL (Chunky Loop Generator) version 0.15
- jar, or InfoZIP (zip and unzip)
- libelf version 0.8.12 (or later) (for LTO)

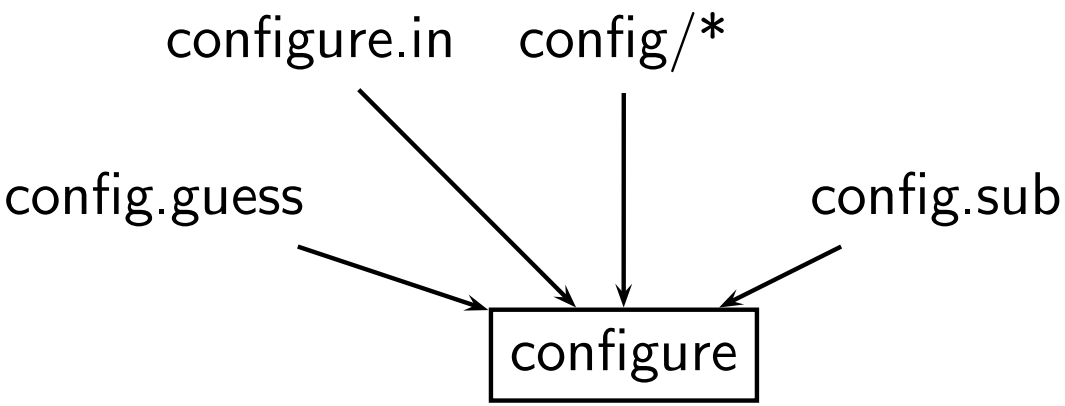
Our Conventions for Directory Names

- GCC source directory : `$(SOURCE)`
- GCC build directory : `$(BUILD)`
- GCC install directory : `$(INSTALL)`
- Important
  - `$(SOURCE)` $\neq$ `$(BUILD)` $\neq$ `$(INSTALL)`
  - None of the above directories should be contained in any of the above directories

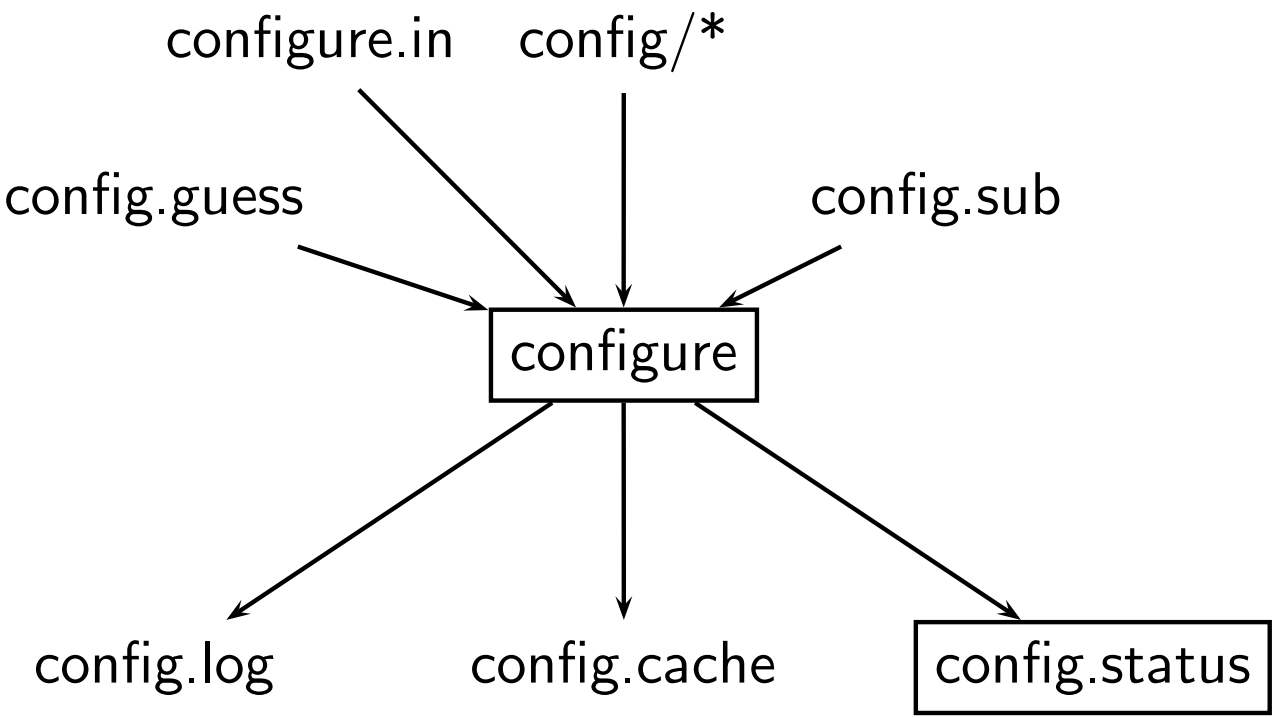
Configuring GCC

configure

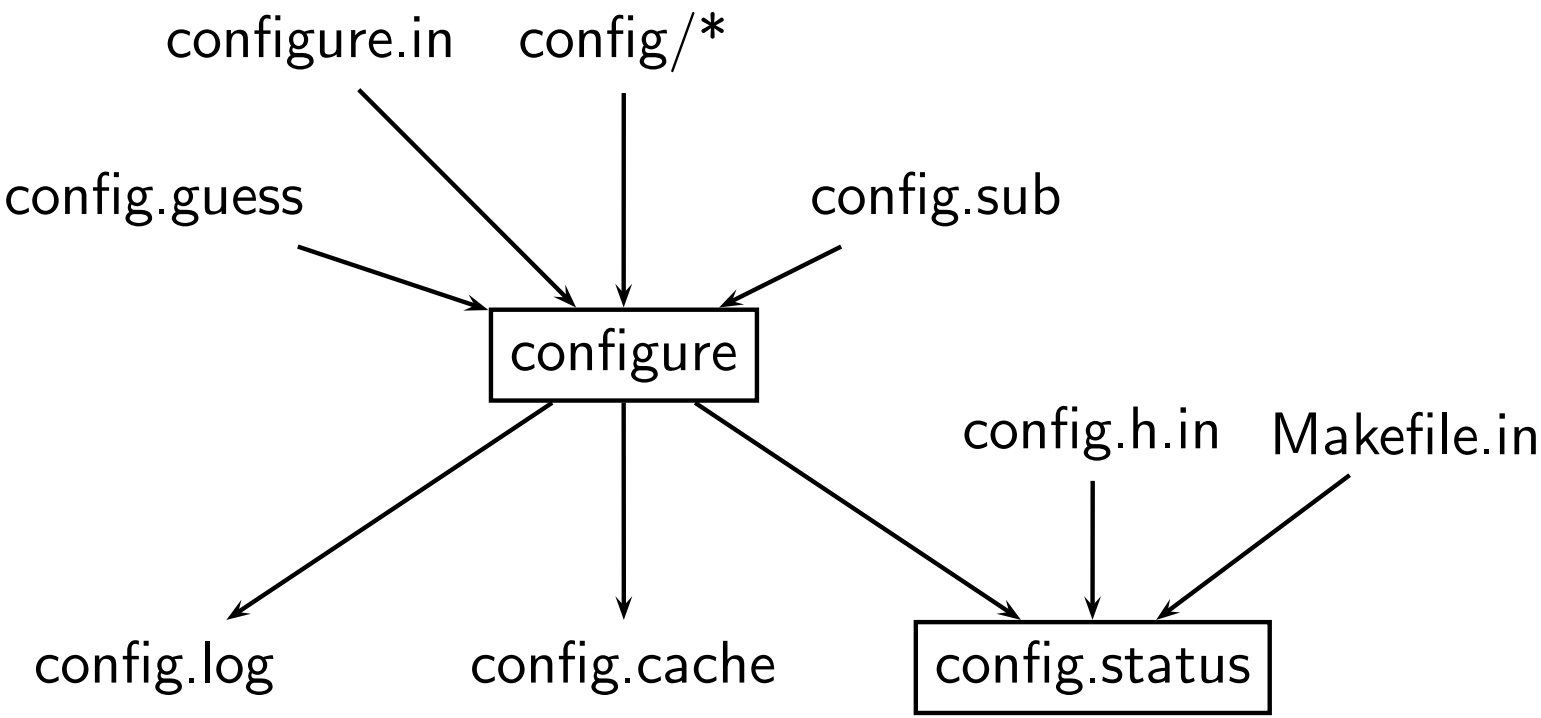
Configuring GCC



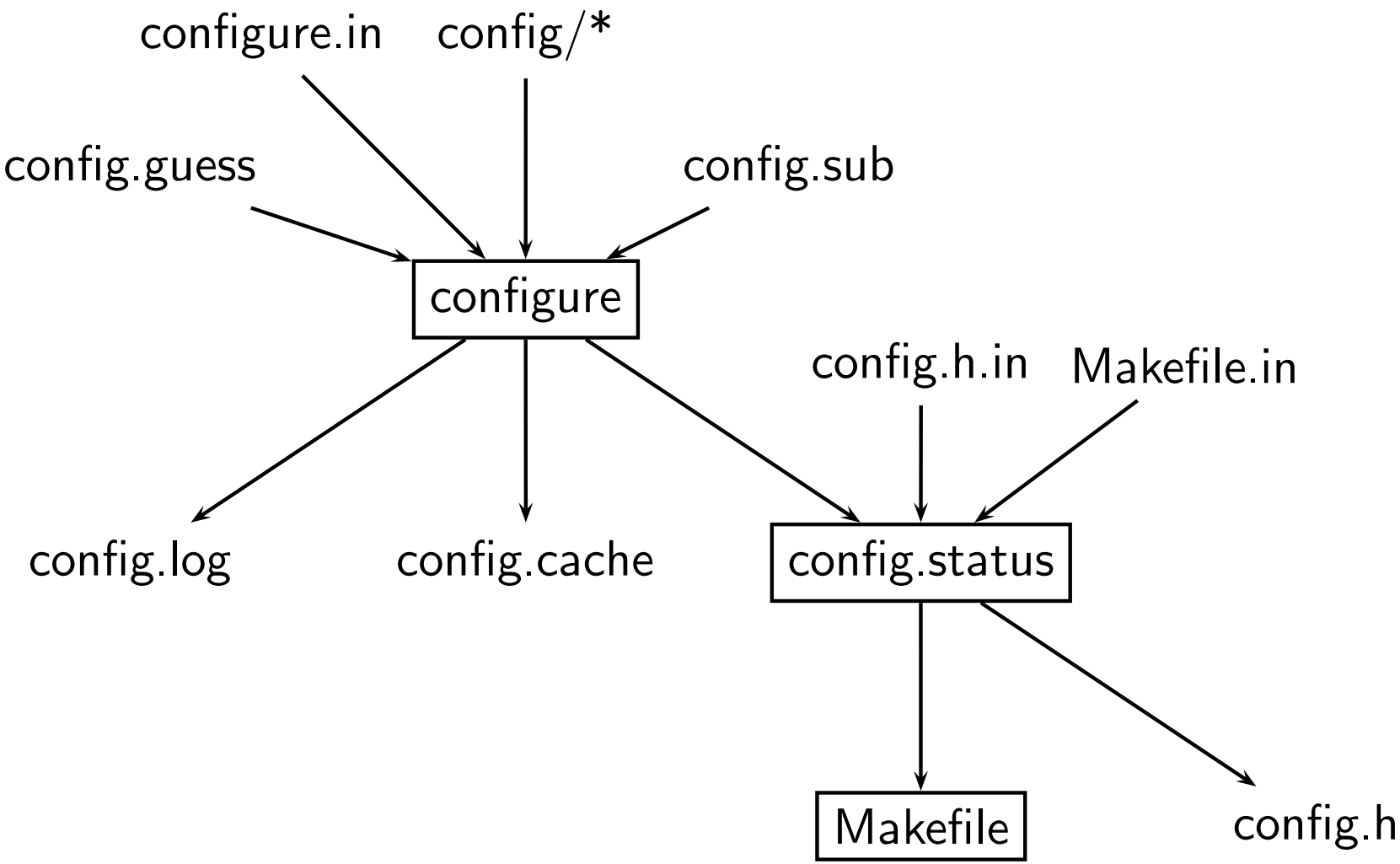
Configuring GCC



Configuring GCC



Configuring GCC



Steps in Configuration and Building

Usual Steps	
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make` • `make install`	

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make` • `make install`	• Download and untar the source • `cd $(BUILD)` • `$(SOURCE)/configure` • `make` • `make install`

Steps in Configuration and Building

Usual Steps	Steps in GCC
• Download and untar the source • `cd $(SOURCE)` • `./configure` • `make` • `make install`	• Download and untar the source • `cd $(BUILD)` • `$(SOURCE)/configure` • `make` • `make install`

GCC generates a large part of source code during a build!

Building a Compiler: Terminology

- The sources of a compiler are compiled (i.e. built) on *Build system*, denoted BS.
- The built compiler runs on the *Host system*, denoted HS.
- The compiler compiles code for the *Target system*, denoted TS.

The built compiler itself runs on HS and generates executables that run on TS.

Variants of Compiler Builds

BS = HS = TS	Native Build
BS = HS ≠ TS	Cross Build
BS ≠ HS ≠ TS	Canadian Cross

Example

Native i386: built on i386, hosted on i386, produces i386 code.

Sparc cross on i386: built on i386, hosted on i386, produces Sparc code.

Bootstrapping

A compiler is just another program

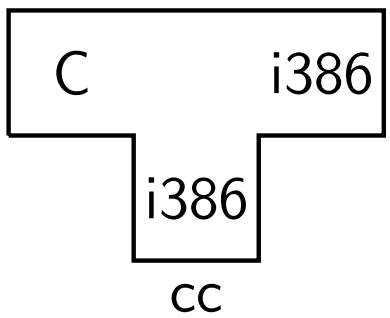
It is improved, bugs are fixed and newer versions are released

To build a new version given a built old version:

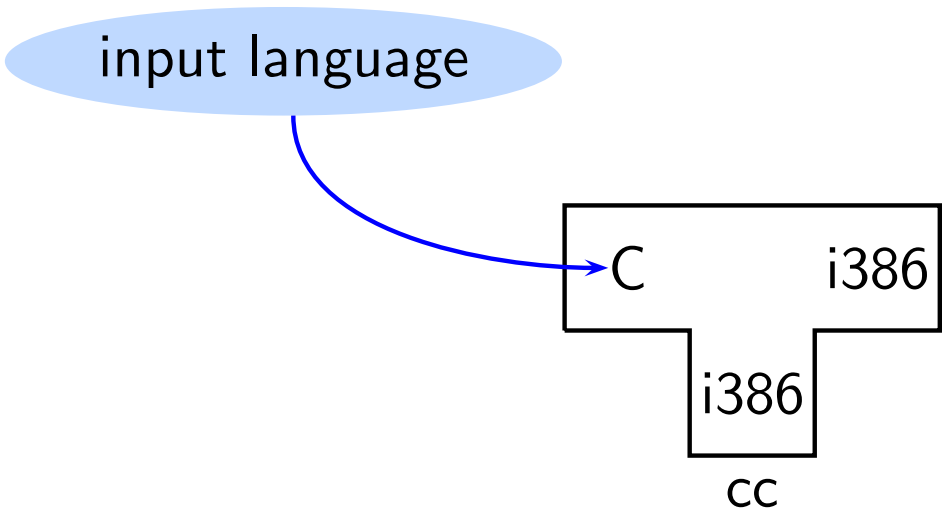
1. Stage 1: Build the new compiler using the old compiler
2. Stage 2: Build another new compiler using compiler from stage 1
3. Stage 3: Build another new compiler using compiler from stage 2
   Stage 2 and stage 3 builds must result in identical compilers

$\Rightarrow$ Building cross compilers stops after Stage 1!

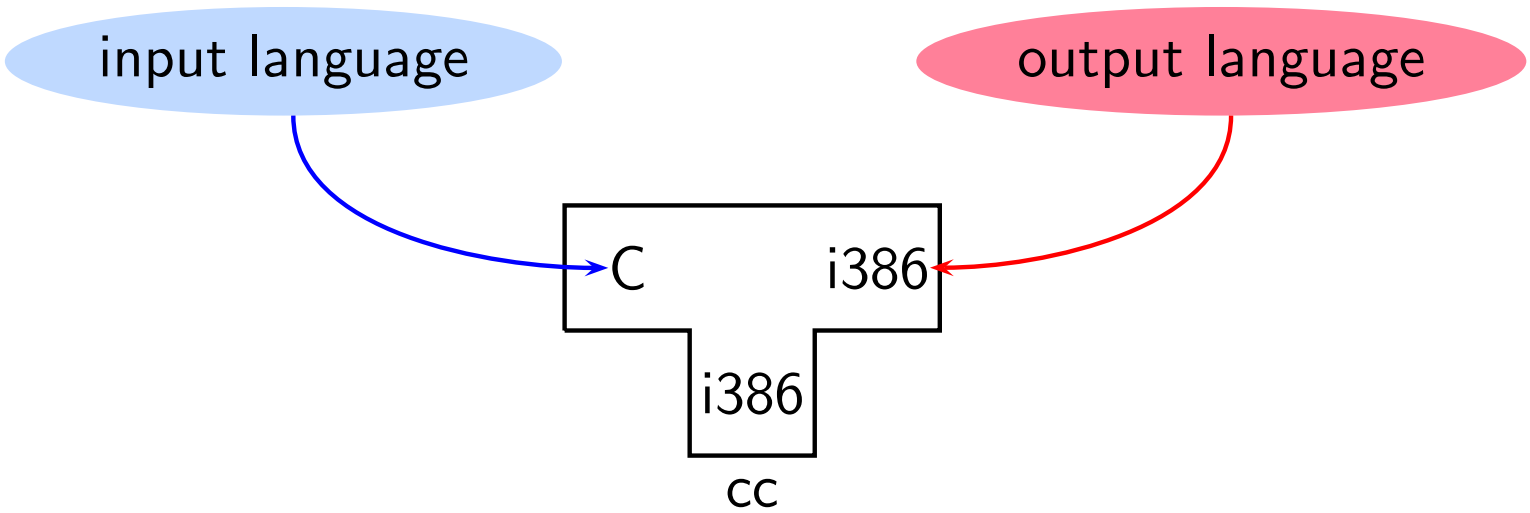
T Notation for a Compiler



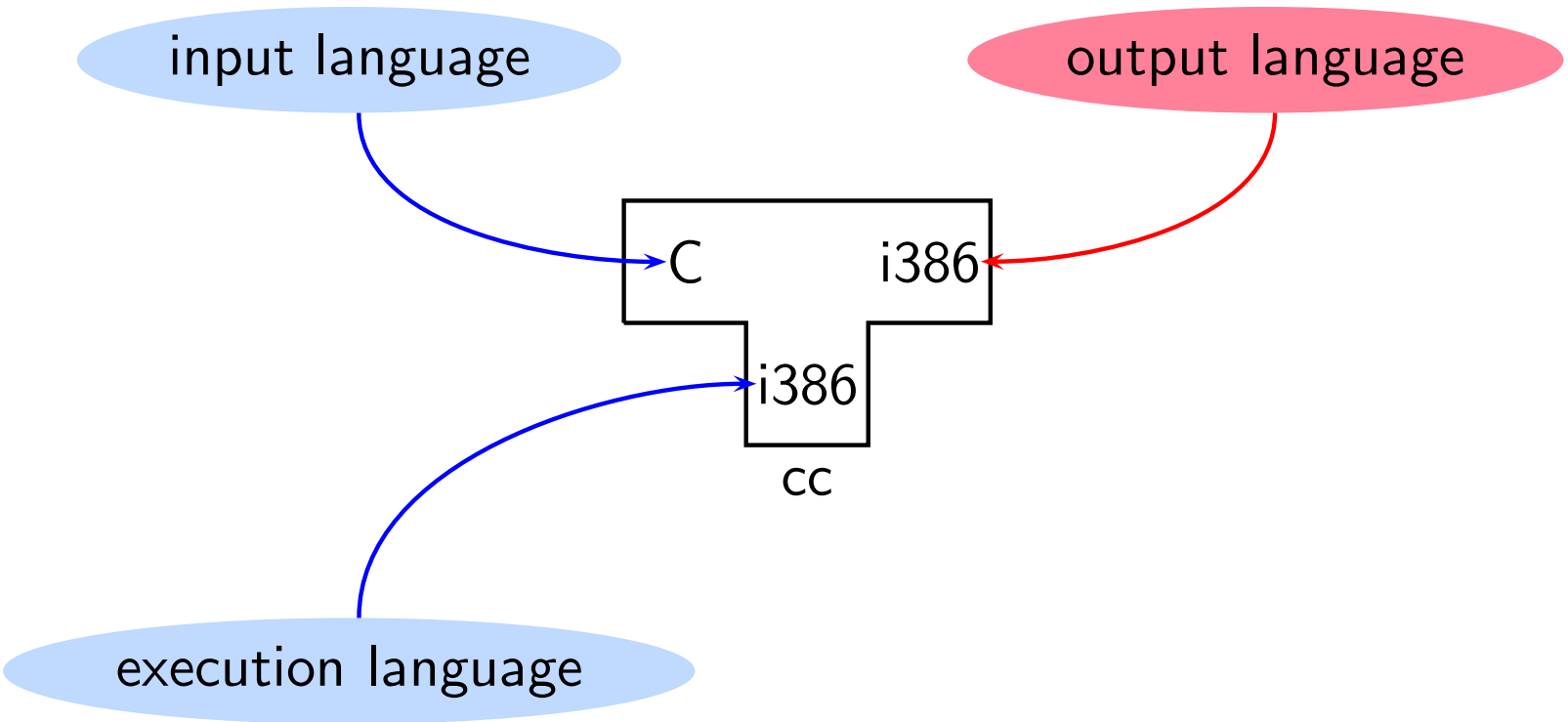
T Notation for a Compiler



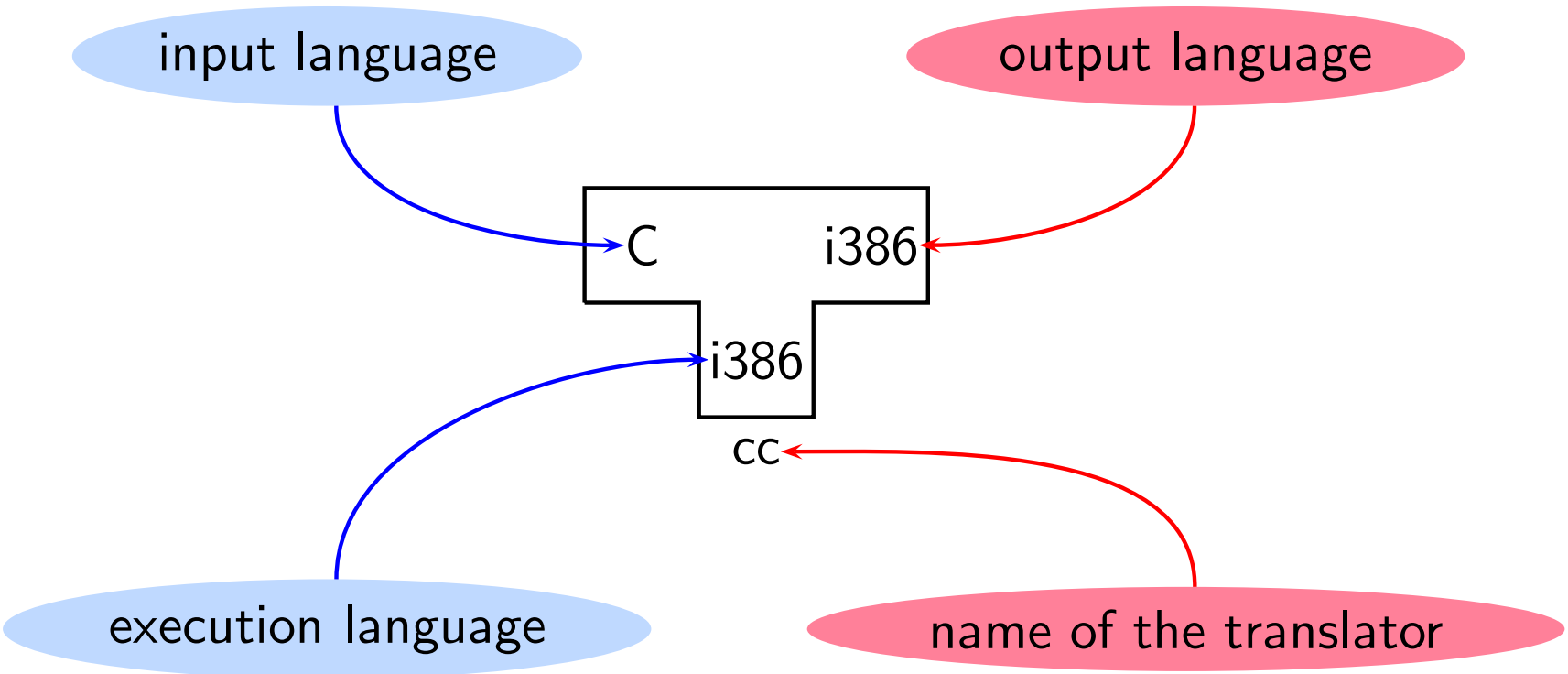
T Notation for a Compiler



T Notation for a Compiler



T Notation for a Compiler

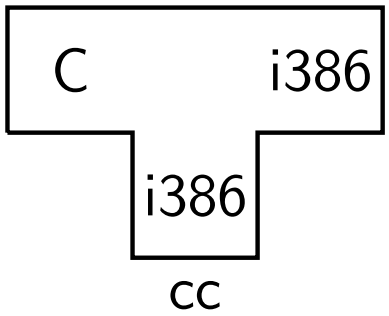


A Native Build on i386

GCC
Source

Requirement: BS = HS = TS = i386

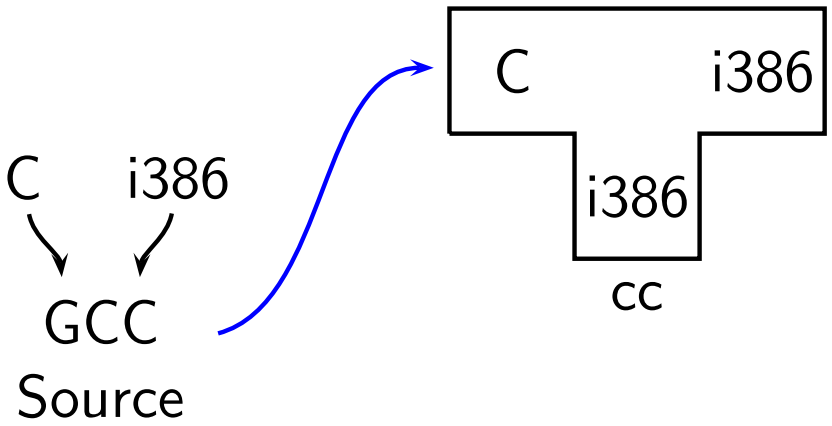
A Native Build on i386



GCC
Source

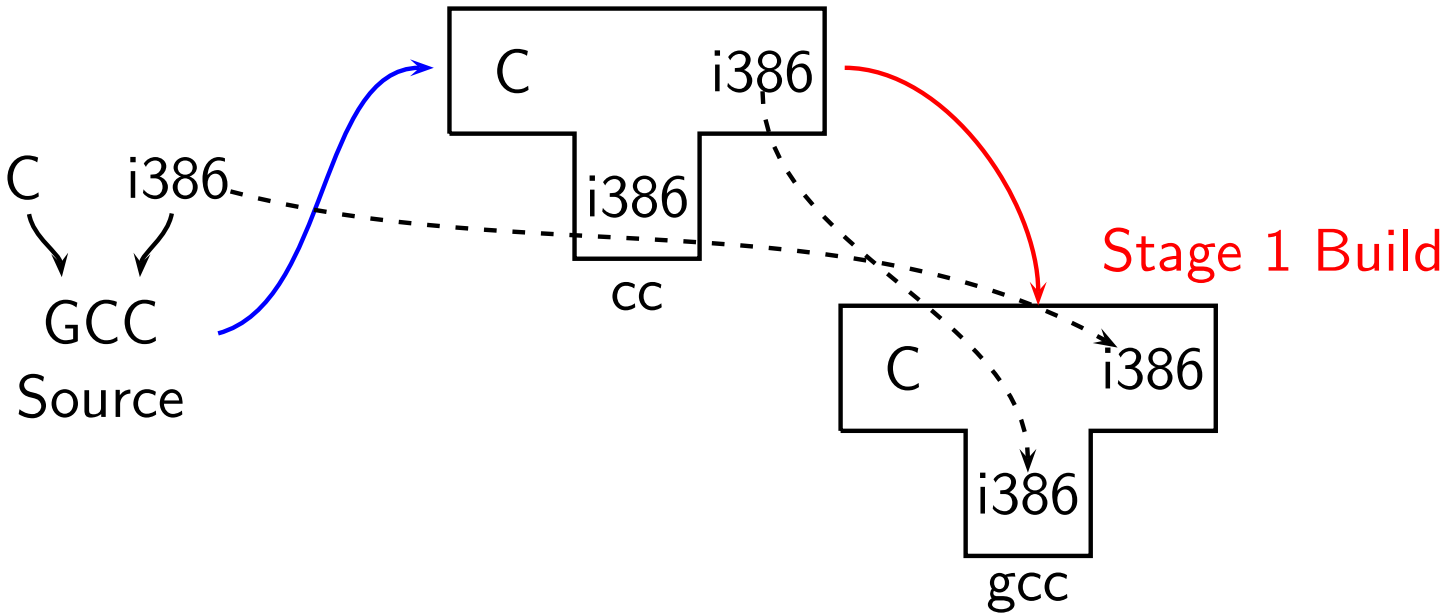
Requirement: BS = HS = TS = i386

A Native Build on i386



Requirement: BS = HS = TS = i386

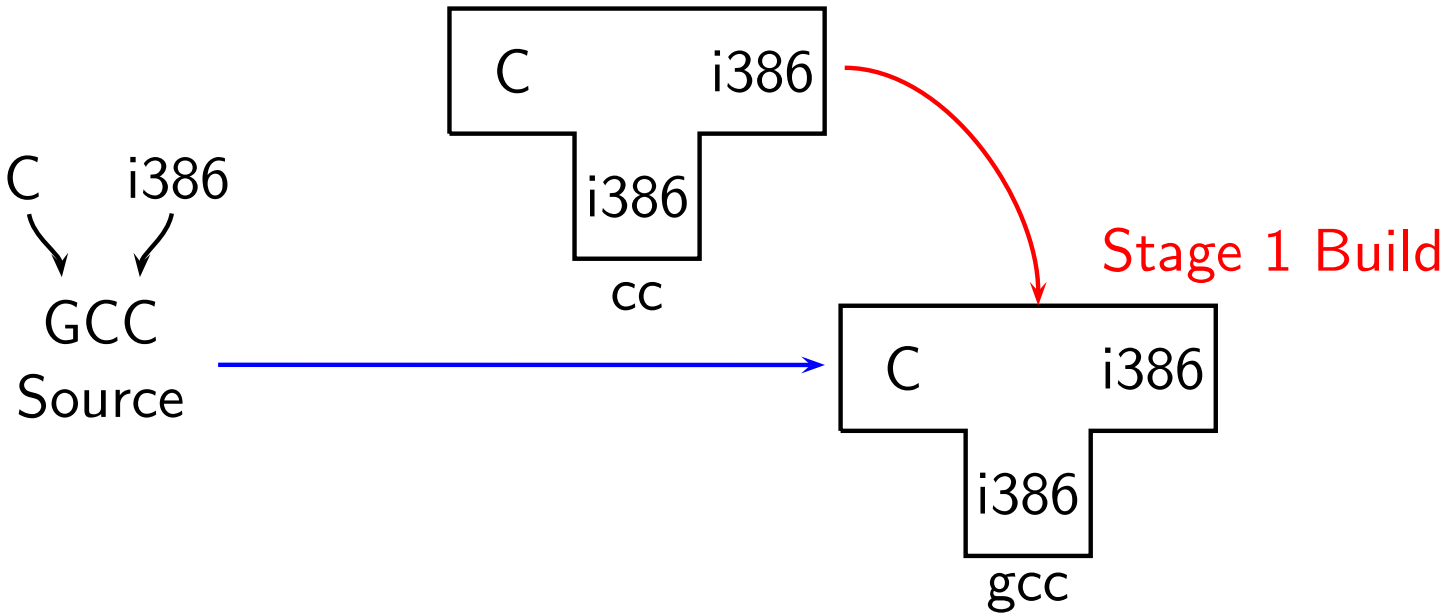
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

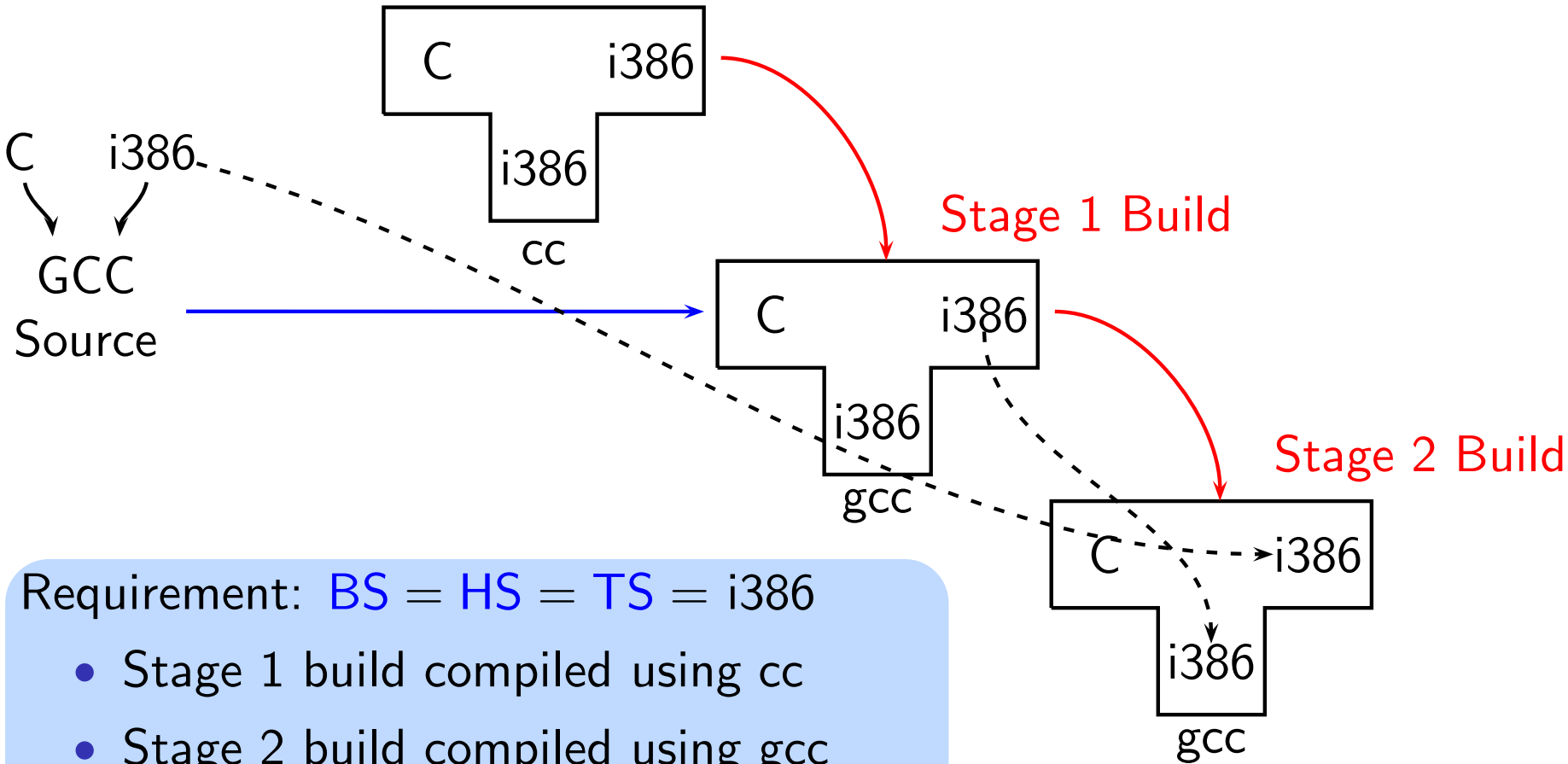
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc

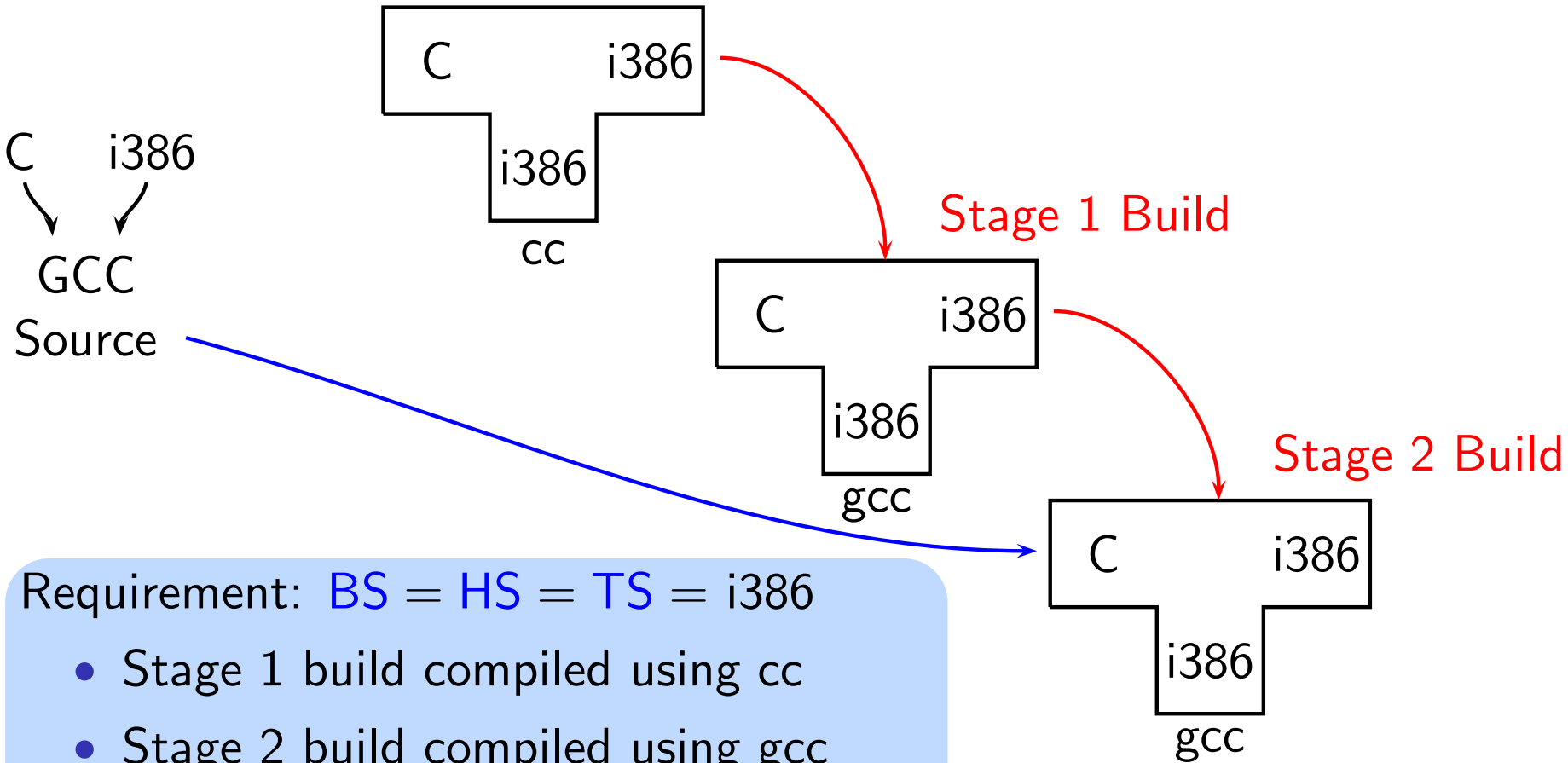
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc

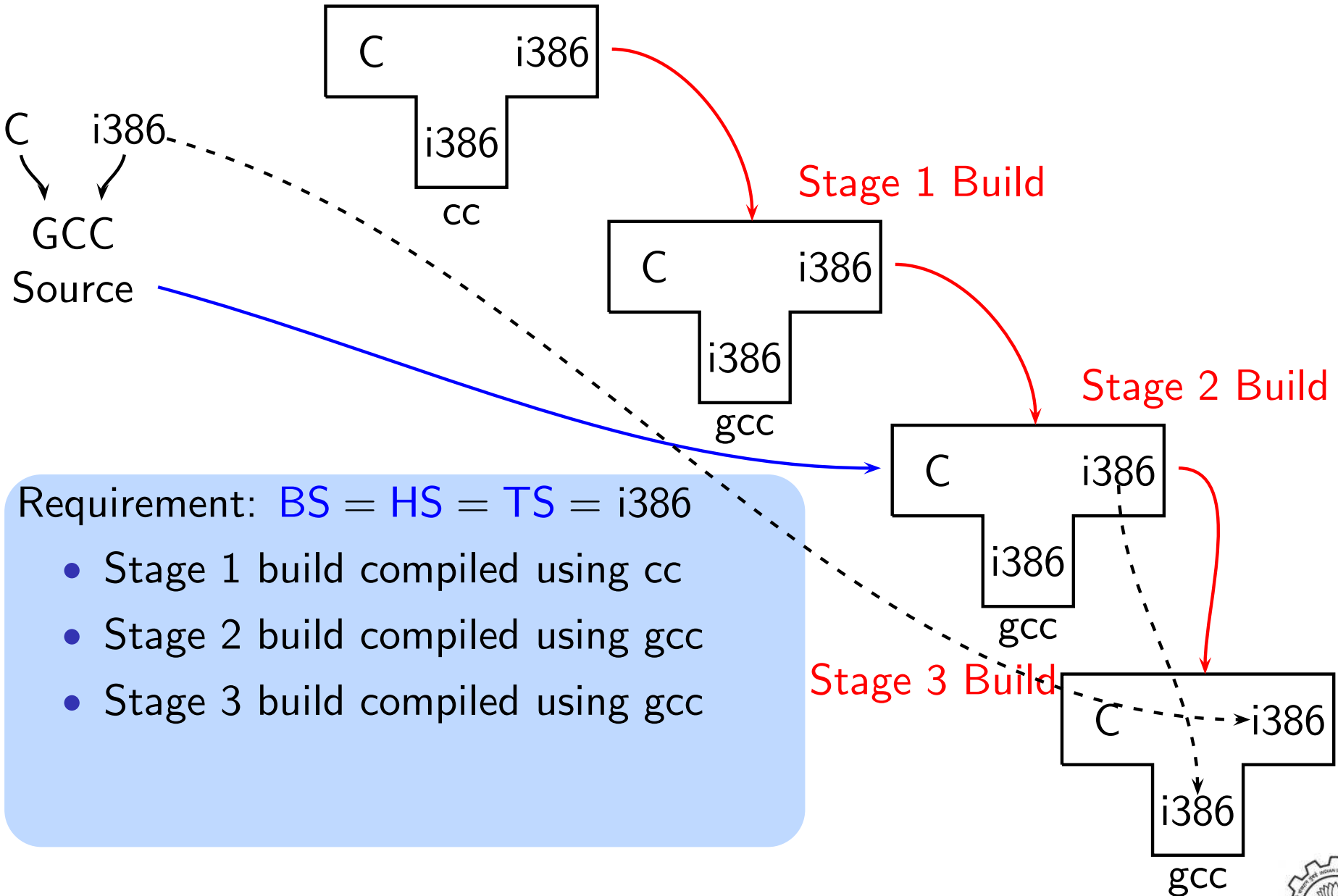
A Native Build on i386



Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc

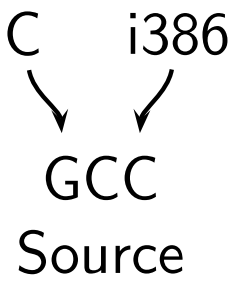
A Native Build on i386

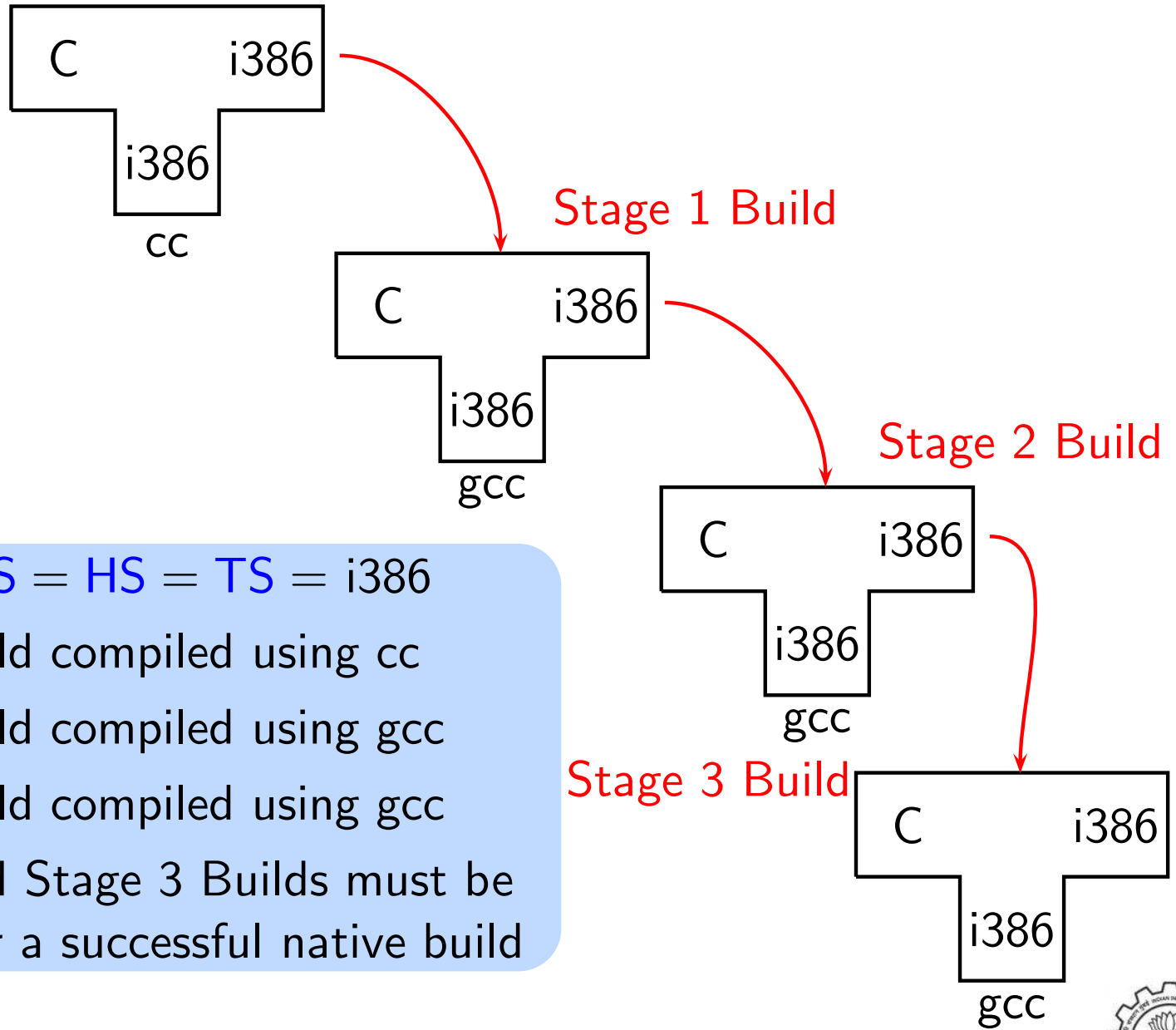


Requirement: BS = HS = TS = i386

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc

A Native Build on i386





Requirement: BS = HS = TS = i386

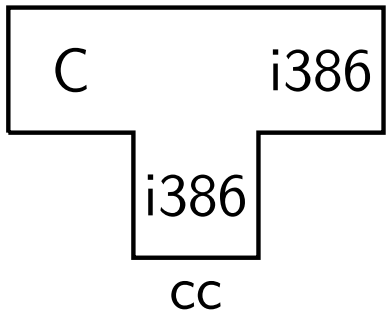
- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
- Stage 3 build compiled using gcc
- Stage 2 and Stage 3 Builds must be identical for a successful native build

A Cross Build on i386

GCC
Source

Requirement: BS = HS = i386, TS = mips

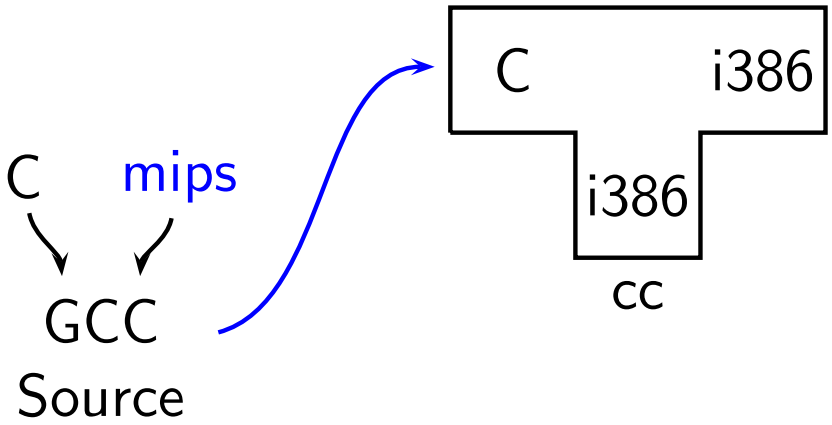
A Cross Build on i386



GCC
Source

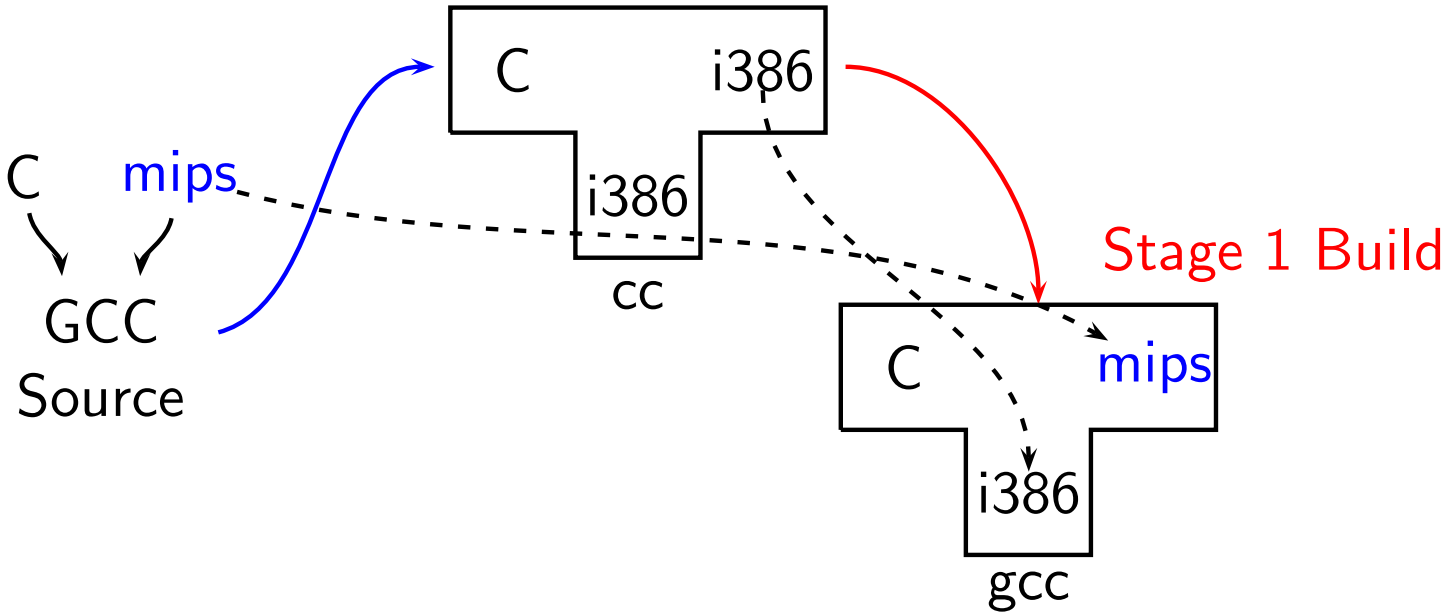
Requirement: BS = HS = i386, TS = mips

A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

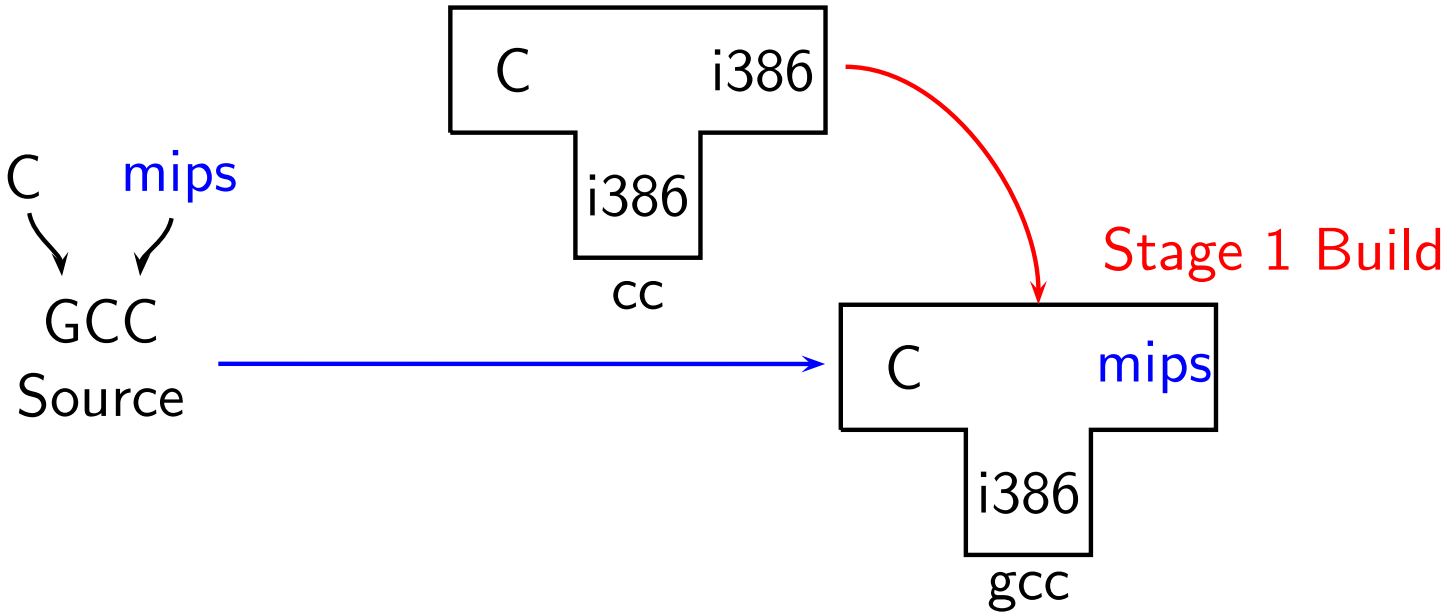
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc

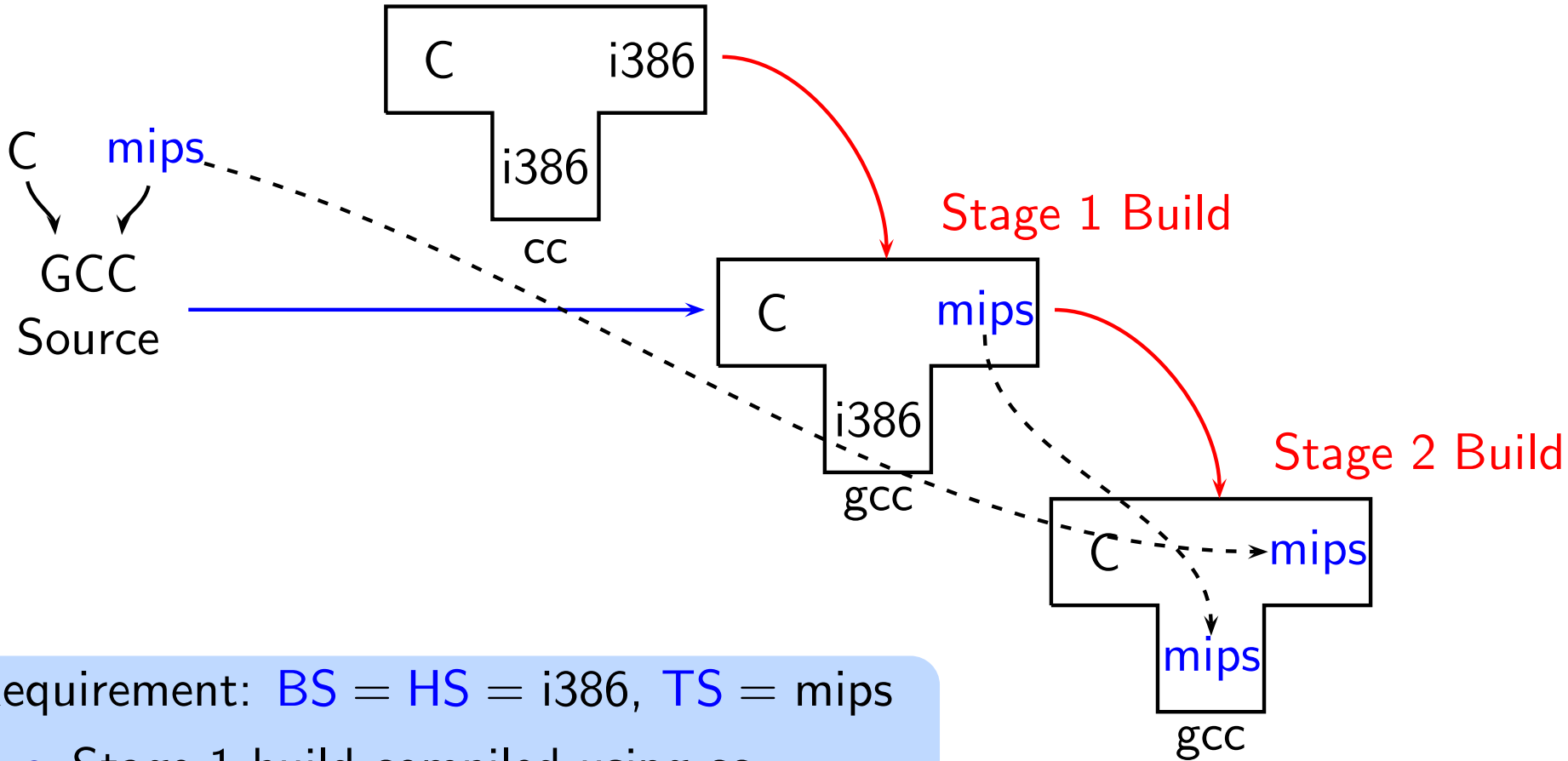
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc

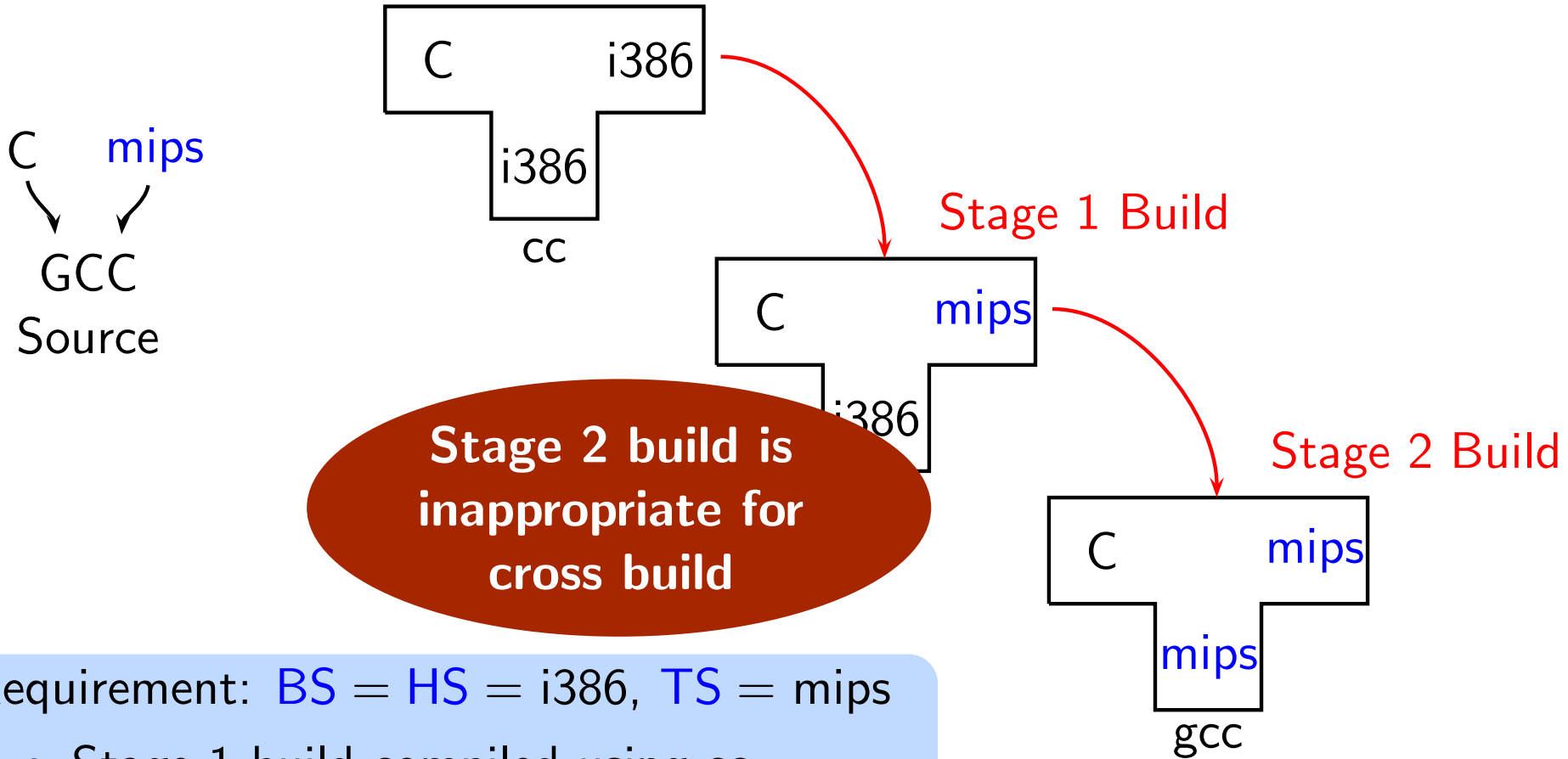
A Cross Build on i386



Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
  Its HS = mips and not i386!

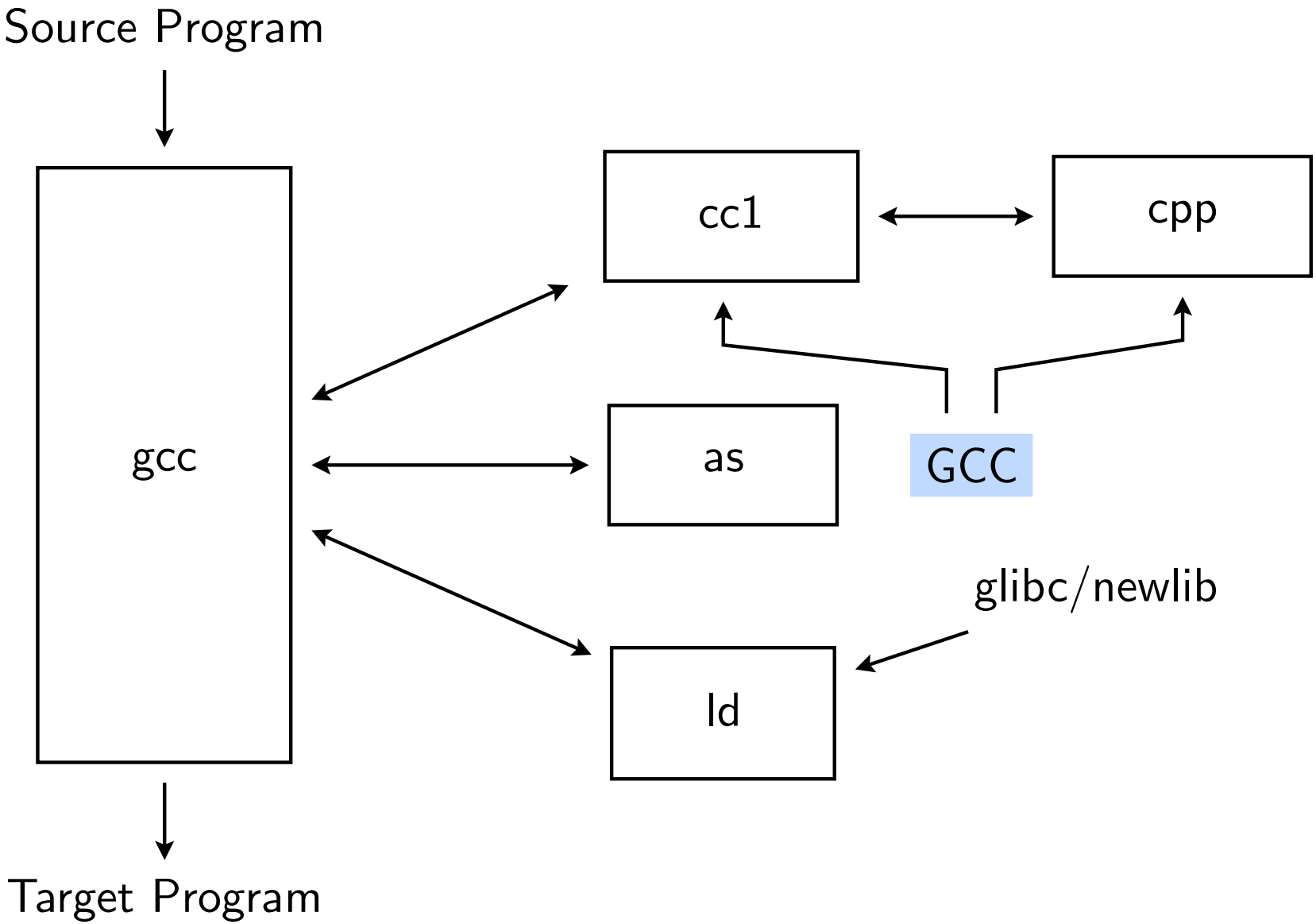
A Cross Build on i386



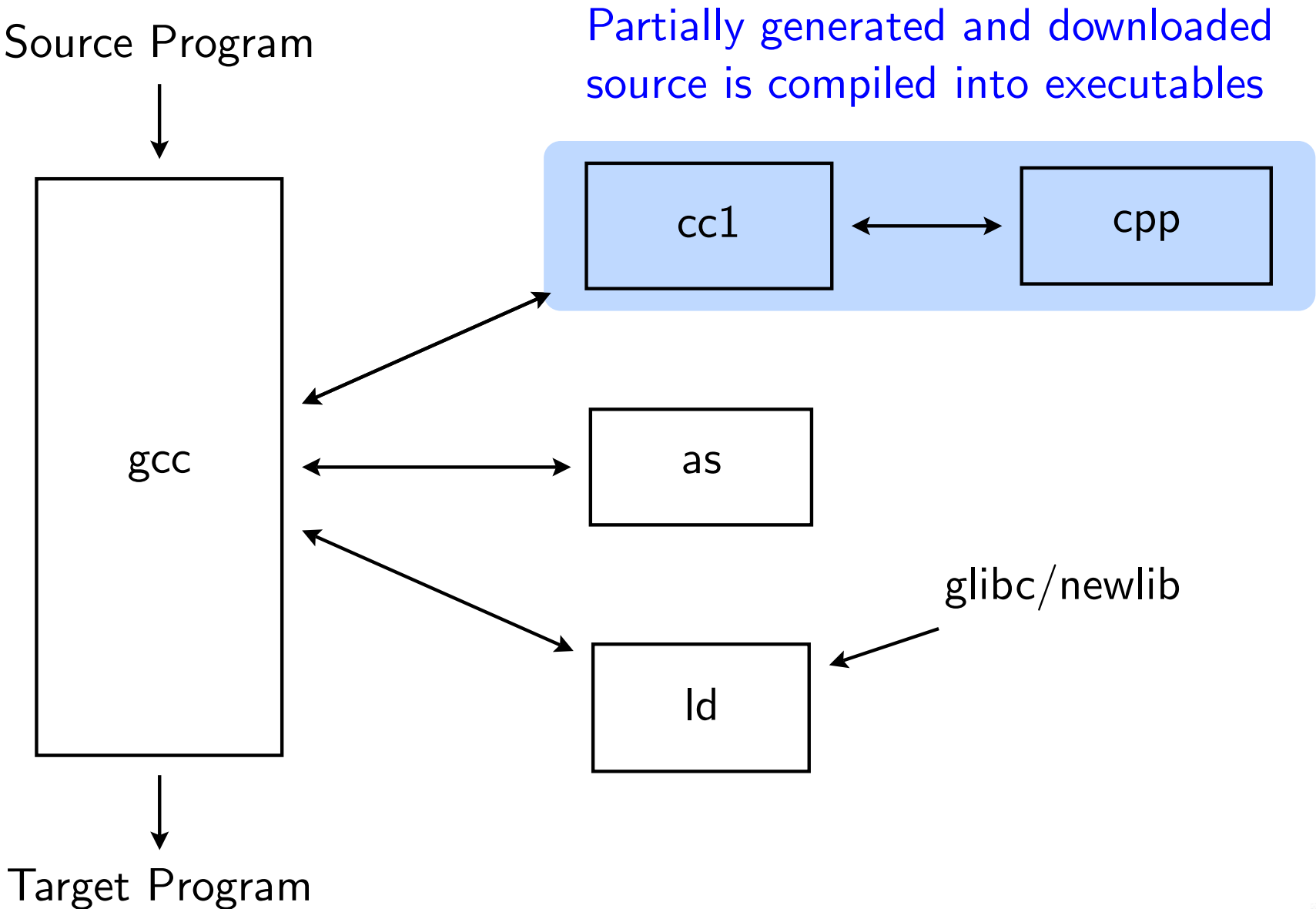
Requirement: BS = HS = i386, TS = mips

- Stage 1 build compiled using cc
- Stage 2 build compiled using gcc
  Its HS = mips and not i386!

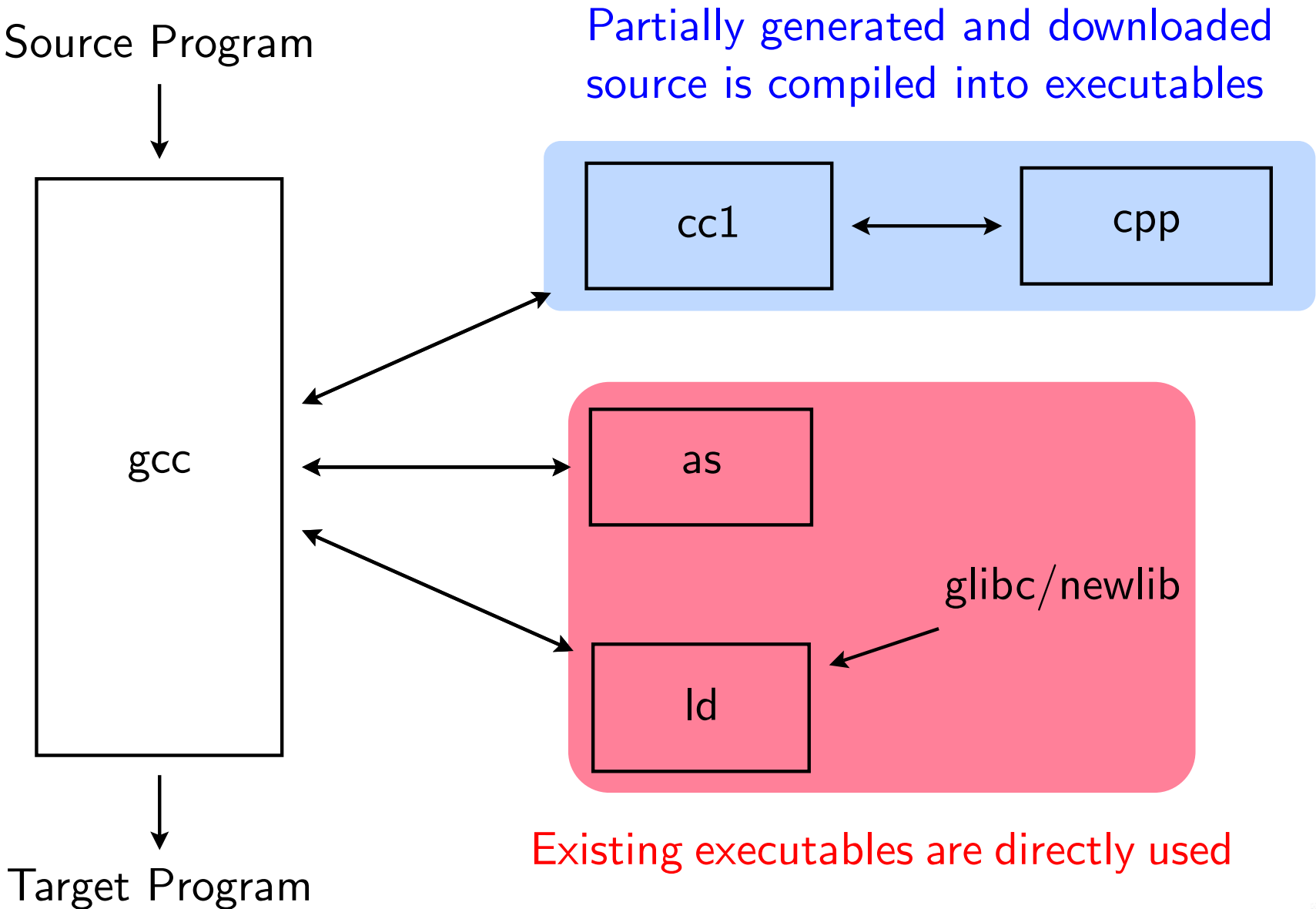
A More Detailed Look at Building



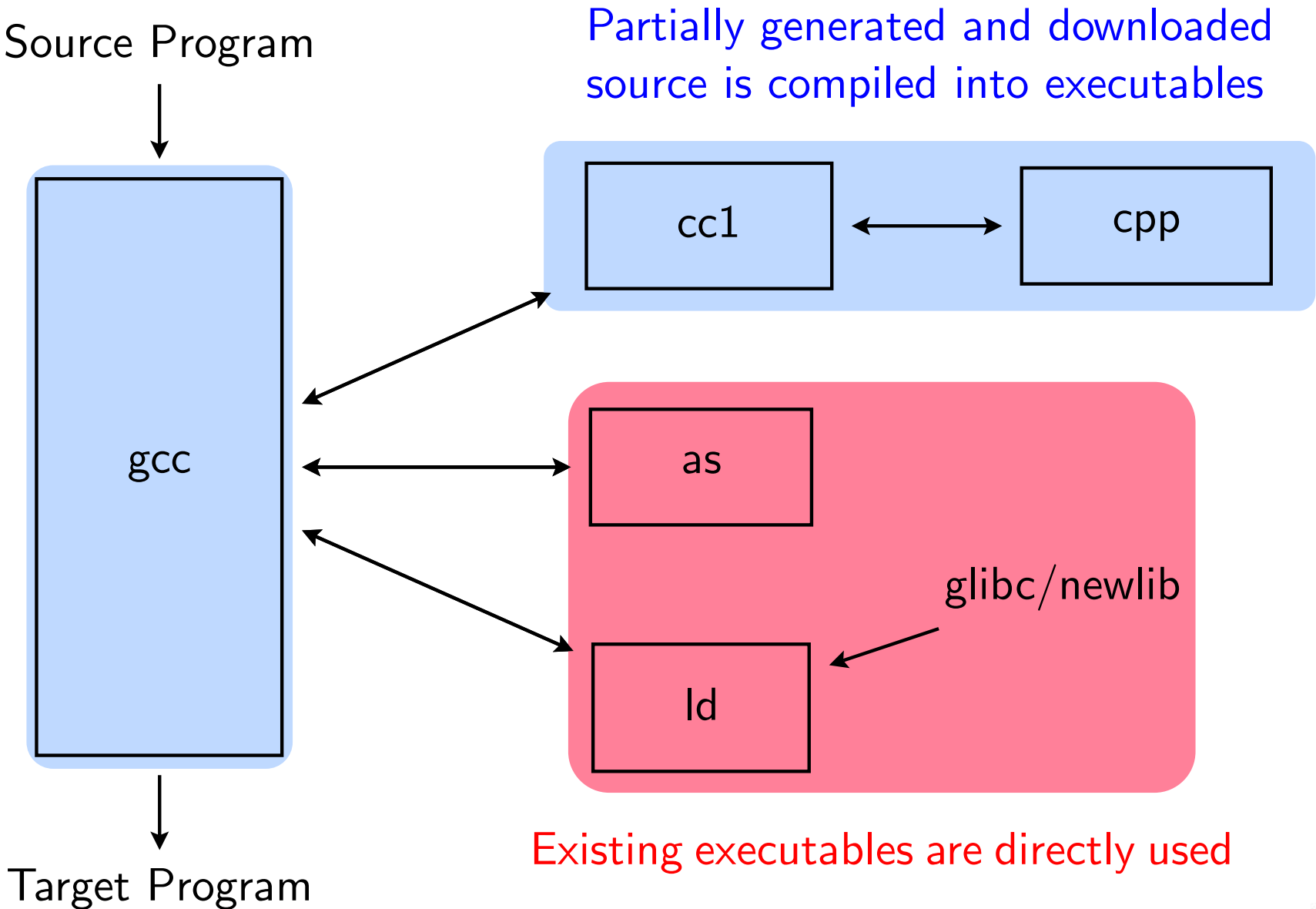
A More Detailed Look at Building



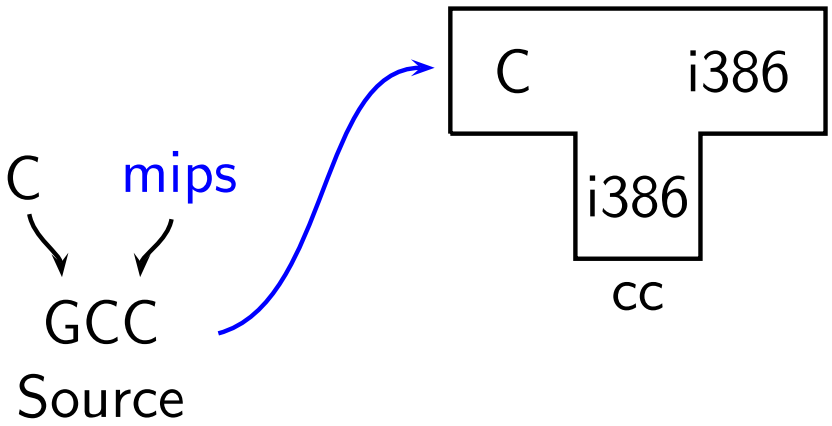
A More Detailed Look at Building



A More Detailed Look at Building

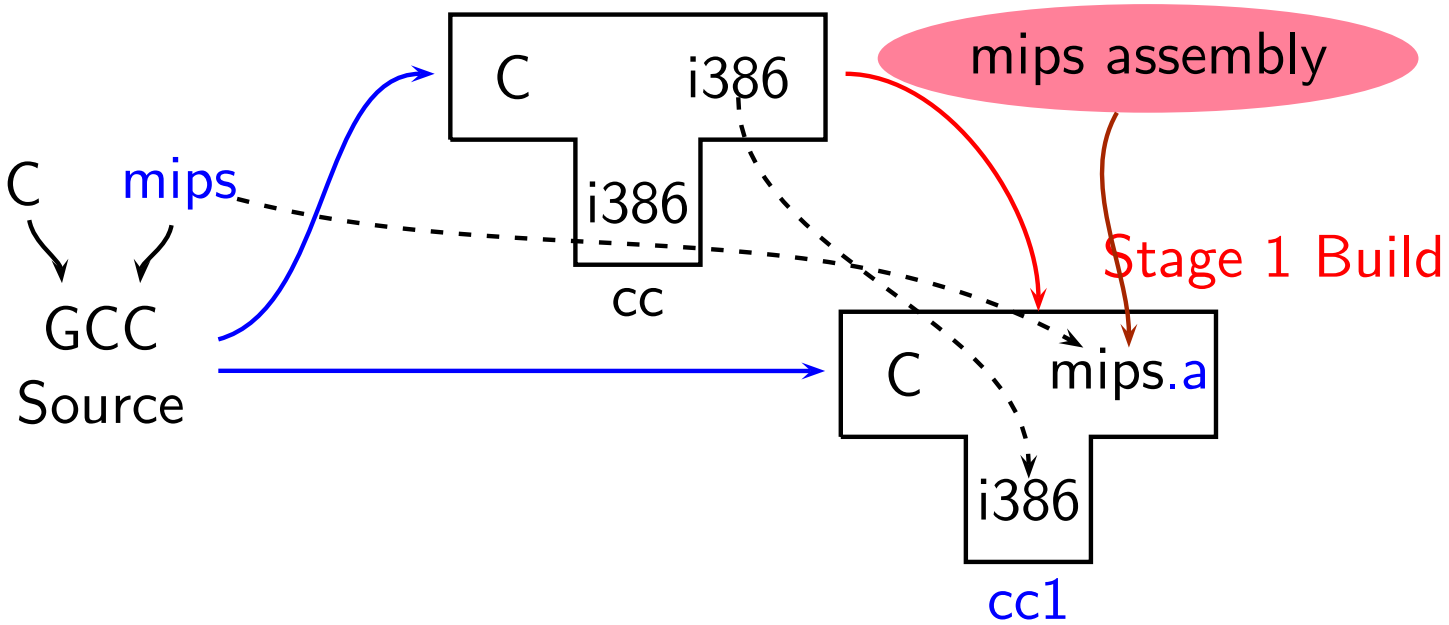


A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

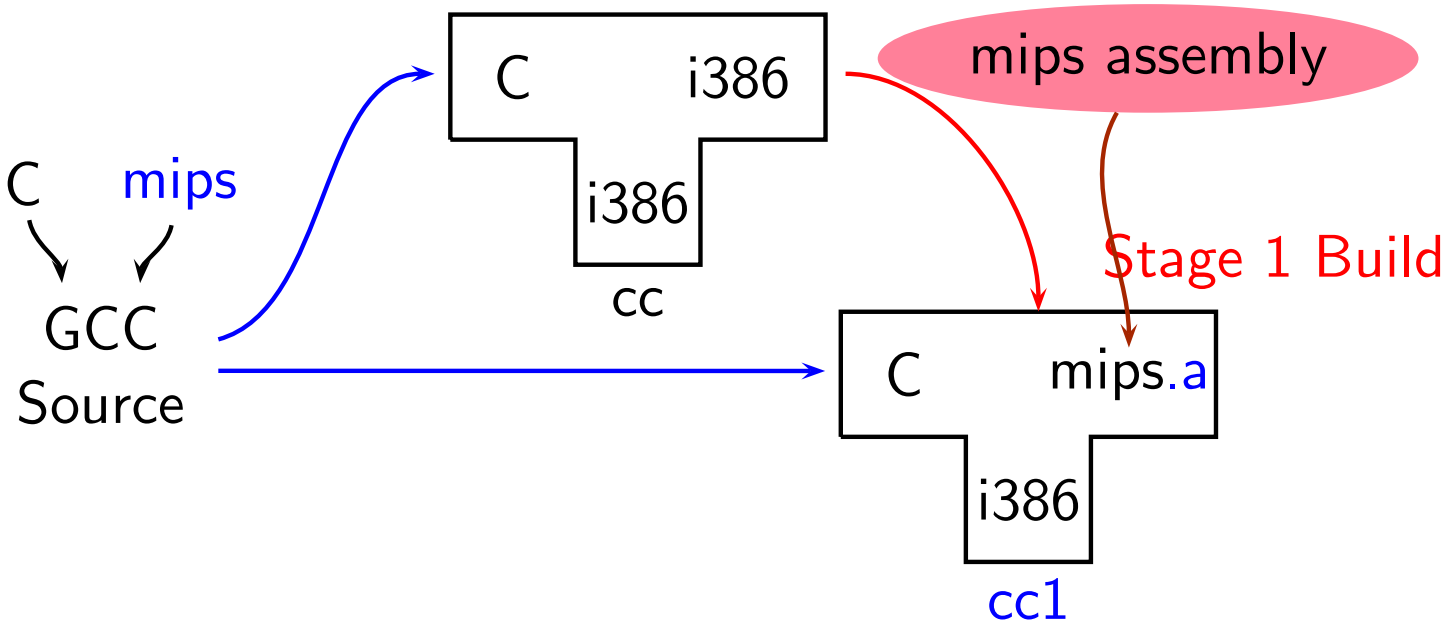
A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

- Stage 1 build consists of only cc1 and not gcc

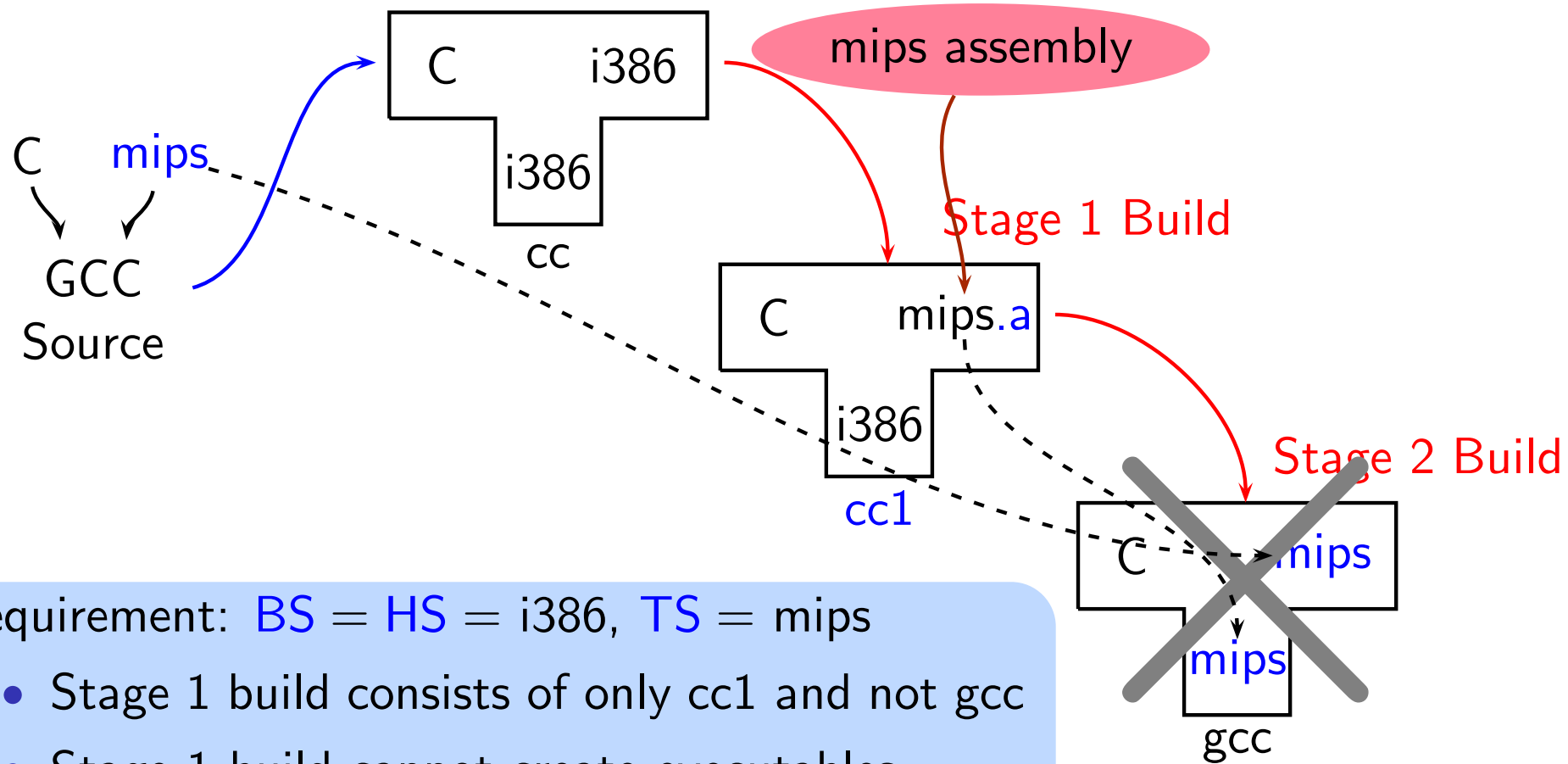
A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

- Stage 1 build consists of only cc1 and not gcc
- Stage 1 build cannot create executables
- Library sources cannot be compiled for mips using stage 1 build

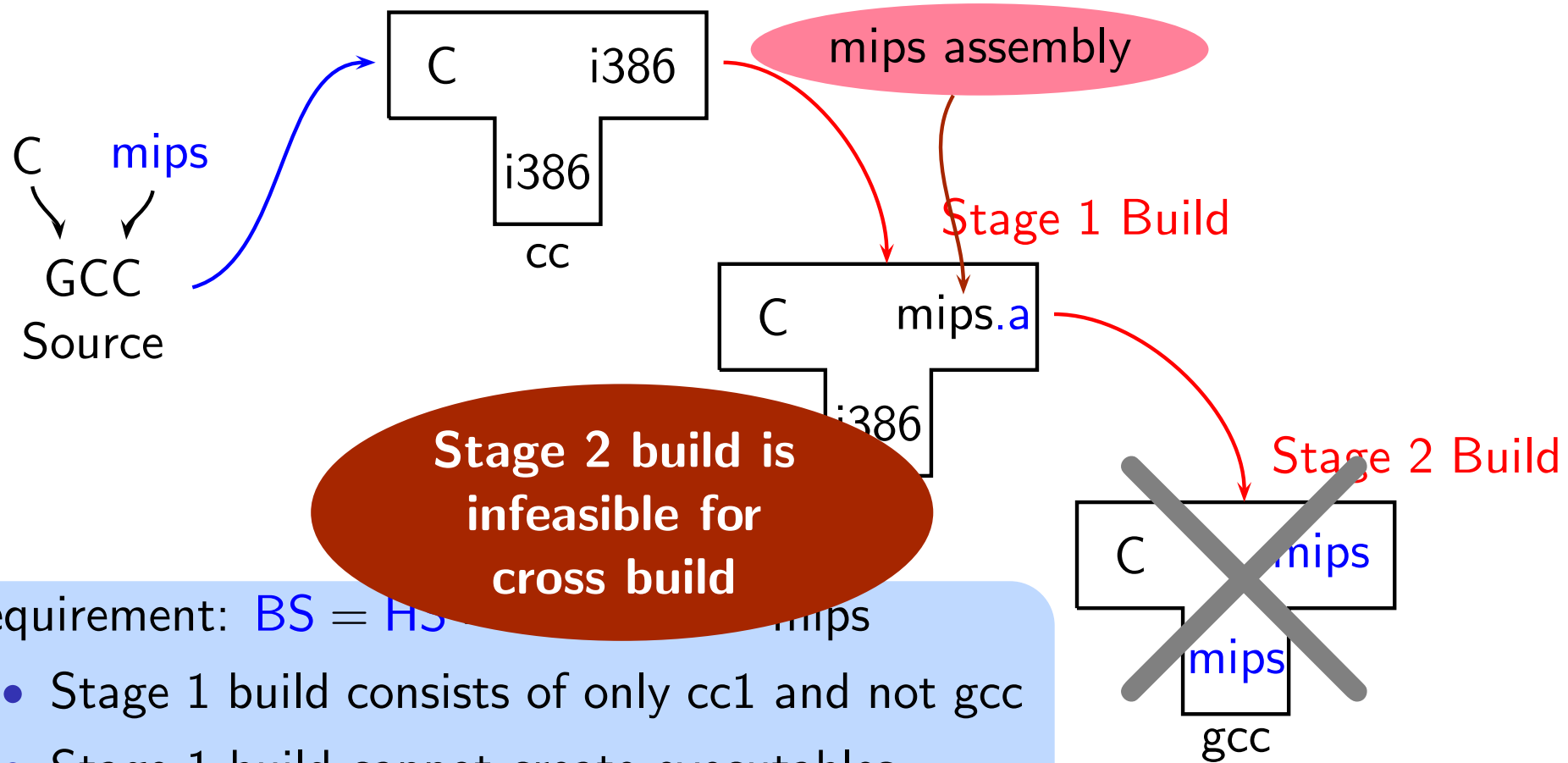
A More Detailed Look at Cross Build



Requirement: BS = HS = i386, TS = mips

- Stage 1 build consists of only cc1 and not gcc
- Stage 1 build cannot create executables
- Library sources cannot be compiled for mips using stage 1 build
- Stage 2 build is not possible

A More Detailed Look at Cross Build



Requirement: BS = HS ... mips

- Stage 1 build consists of only cc1 and not gcc
- Stage 1 build cannot create executables
- Library sources cannot be compiled for mips using stage 1 build
- Stage 2 build is not possible

Cross Build Revisited

- Option 1: Build binutils in the same source tree as `gcc`
  Copy binutils source in `$(SOURCE)`, configure and build stage 1
- Option 2:
  - Compile cross-assembler (`as`), cross-linker (`ld`), cross-archiver (`ar`), and cross-program to build symbol table in archiver (`ranlib`),
  - Copy them in `$(INSTALL)/bin`
  - Build stage GCC
  - Install newlib
  - Reconfigure and build GCC
    Some options differ in the two builds

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE)/configure <options>`
  `configure` output: customized `Makefile`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE)/configure <options>`
  `configure` output: customized `Makefile`
- `make 2> make.err > make.log`

Commands for Configuring and Building GCC

This is what *we* specify

- `cd $(BUILD)`
- `$(SOURCE)/configure <options>`
  `configure` output: customized `Makefile`
- `make 2> make.err > make.log`
- `make install 2> install.err > install.log`

Build for a Given Machine

This is what actually happens!

- Generation
  - Generator sources (`$(SOURCE)/gcc/gen*.c`) are read and generator executables are created in `$(BUILD)/gcc/build`
  - MD files are read by the generator executables and back end source code is generated in `$(BUILD)/gcc`
- Compilation
  Other source files are read from `$(SOURCE)` and executables created in corresponding subdirectories of `$(BUILD)`
- Installation
  Created executables and libraries are copied in `$(INSTALL)`

Build for a Given Machine

This is what actually happens!

- Generation
  - Generator sources (`$(SOURCE)/gcc/gen*.c`) are read and generator executables are created in `$(BUILD)/gcc/build`
  - MD files are read by the generator executables and back end source code is generated in `$(BUILD)/gcc`
- Compilation
  Other source files are read from `$(SOURCE)` and executables created in corresponding subdirectories of `$(BUILD)`
- Installation
  Created executables and libraries are copied in `$(INSTALL)`

```
gcov-iov
genoutput
genmddeps
gencheck
gengenrtl
gencheckSum
genconditions
genmodes
genflags
genattr
genopinit
genrecog
gencondmd.c
genattrtab
genautomata
genpreds
genemit
gencondmd
gengtype
gencodes
genconfig
genpeep
genconstants
genextract
```

Build failures due to Machine Descriptions

Incomplete MD specifications	$\Rightarrow$	Unsuccessful build
Incorrect MD specification	$\Rightarrow$	Successful build but run time failures/crashes (either ICE or `SIGSEGV`)

Building `cc1` Only

- Add a new target in the `Makefile.in`

```
cc1:
	make all-gcc TARGET-gcc=cc1$(exeext)
```

- Configure and build with the command `make cc1`.

Common Configuration Options

`--target`

- Necessary for cross build
- Possible `host-cpu-vendor` strings: Listed in `$(SOURCE)/config.sub`

`--enable-languages`

- Comma separated list of language names
- Default names: `c`, `c++`, `fortran`, `java`, `objc`
- Additional names possible: `ada`, `obj-c++`, `treelang`

`--prefix=$(INSTALL)`

`--program-prefix`

- Prefix string for executable names

`--disable-bootstrap`

- Build stage 1 only

Registering New Machine Descriptions

- Define a new system name, typically a triple.
  e.g. `spim-gnu-linux`
- Edit `$(SOURCE)/config.sub` to recognize the triple
- Edit `$(SOURCE)/gcc/config.gcc` to define
  - any back end specific variables
  - any back end specific files
  - `$(SOURCE)/gcc/config/<cpu>` is used as the back end directory

  for recognized system names.

Tip

Read comments in `$(SOURCE)/config.sub` & `$(SOURCE)/gcc/config/<cpu>`.

Testing GCC

- Pre-requisites - `Dejagnu`, `Expect` tools
- Option 1: Build GCC and execute the command
  `make check`
  or
  `make check-gcc`
- Option 2: Use the configure option `--enable-checking`
- Possible list of checks
  - Compile time consistency checks
    `assert`, `fold`, `gc`, `gcac`, `misc`, `rtl`, `rtlflag`, `runtime`, `tree`, `valgrind`
  - Default combination names
    - `yes`: `assert`, `gc`, `misc`, `rtlflag`, `runtime`, `tree`
    - `no`
    - `release`: `assert`, `runtime`
    - `all`: all except `valgrind`

GCC Testing framework

- `make` will invoke `runtest` command
- Specifying `runtest` options using `RUNTESTFLAGS` to customize torture testing
  `make check RUNTESTFLAGS="compile.exp"`
- Inspecting testsuite output: `$(BUILD)/gcc/testsuite/gcc.log`

Interpreting Test Results

- PASS: the test passed as expected
- XPASS: the test unexpectedly passed
- FAIL: the test unexpectedly failed
- XFAIL: the test failed as expected
- UNSUPPORTED: the test is not supported on this platform
- ERROR: the testsuite detected an error
- WARNING: the testsuite detected a possible problem

GCC Internals document contains an exhaustive list of options for testing

Configuring and Building GCC – Summary

- Choose the source language: C (`--enable-languages=c`)
- Choose installation directory: (`--prefix=<absolute path>`)
- Choose the target for non native builds: (`--target=sparc-sunos-sun`)
- Run: `configure` with above choices
- Run: `make` to
  - generate target specific part of the compiler
  - build the entire compiler
- Run: `make install` to install the compiler

Tip

Redirect all the outputs:

```
$ make > make.log 2> make.err
```

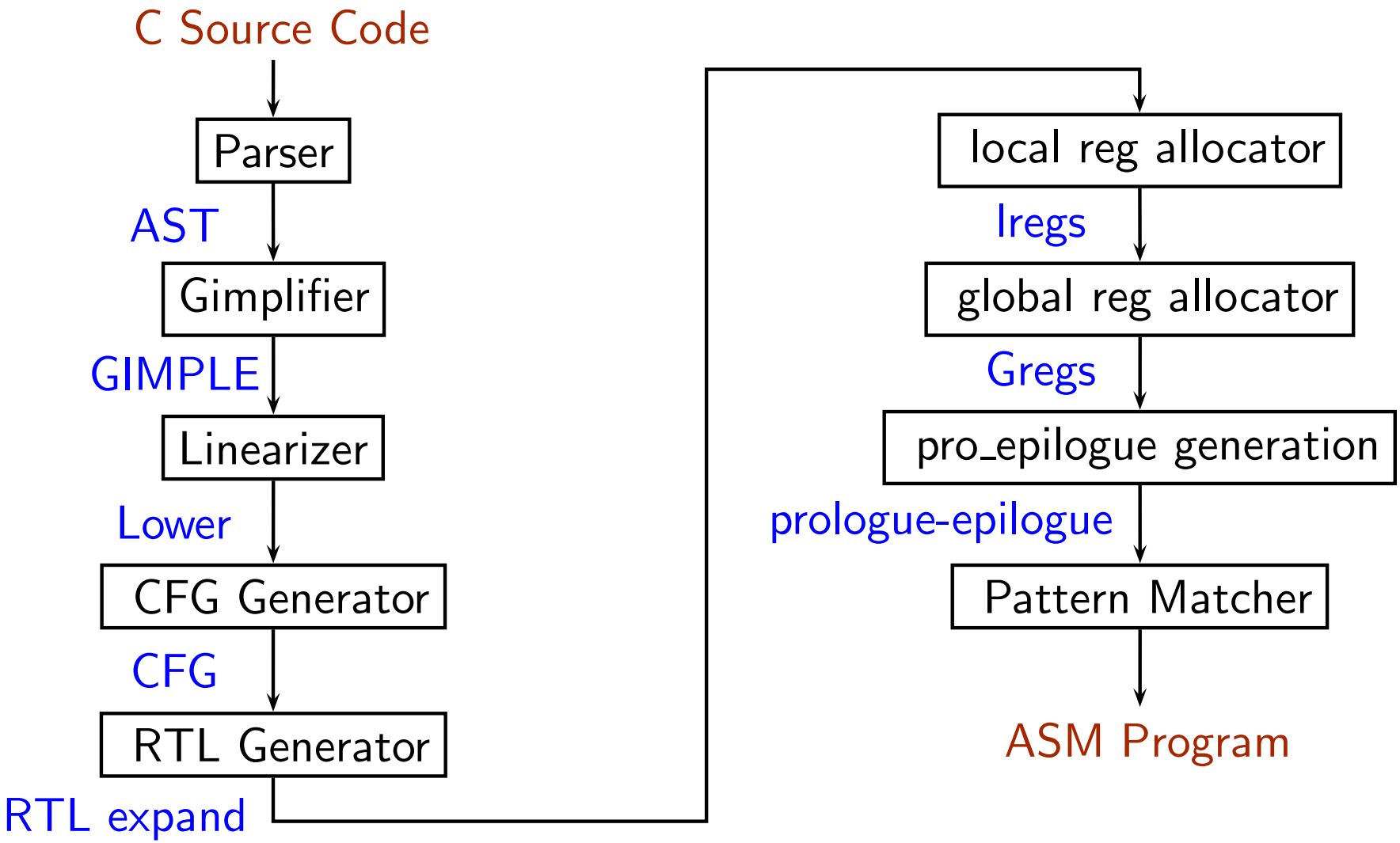

Part 4

First Level Gray Box Probing of GCC

Gray Box Probing of GCC: Outline

- Overview of translation sequence in GCC
- Overview of intermediate representations
- Intermediate representations of programs across important phases

Important Phases of GCC



Lowering of abstraction!

Phases of GCC

To see the output after each pass use the option

`-fdump-<ir>-<pass>`

where `<ir>` is

- `-tree-<pass>`
  `<pass>` could be: `gimple` , `cfg` etc.
  Use `-all` to see all dumps
- `-rtl-<pass>`
  `<pass>` could be: `expand`, `greg`, `vreg` etc.
  Use `-all` to see all dumps
  We can also use `-da` option

Example:

```
gcc -fdump-tree-all -fdump-rtl-all test.c
```


Example Program

```
int main()
{
    int a=2, b=3, c=4;
    while (a<=7)
    {
        a = a+1;
    }
    if (a<=12)
        a = a+b+c;
}
```

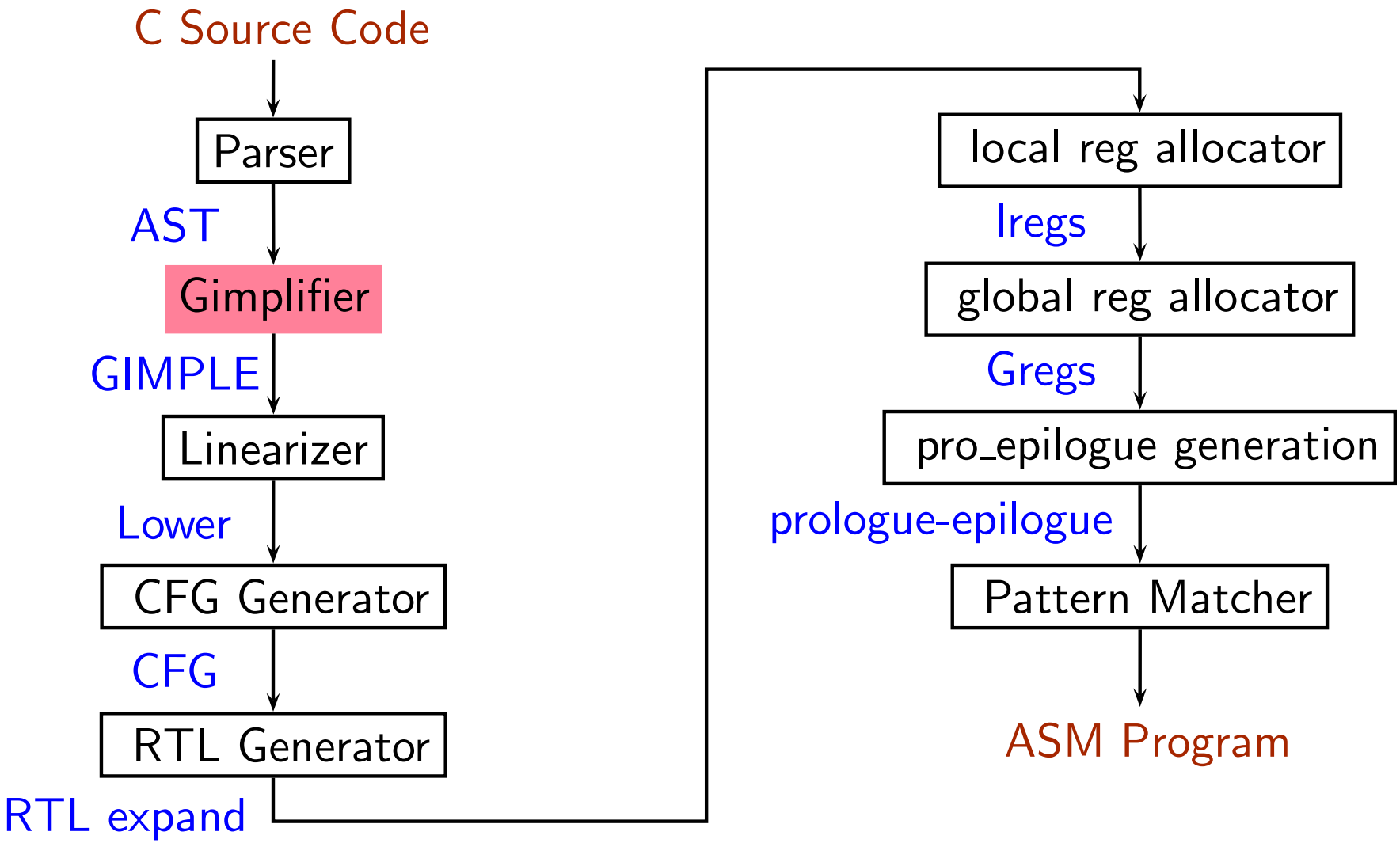

GCC 4.3.3 Dumps for Our Example Program

```
test.c.001t.tu
test.c.003t.original
test.c.004t.gimple
test.c.006t.vcg
test.c.007t.useless
test.c.010t.lower
test.c.011t.ehopt
test.c.012t.eh
test.c.013t.cfg
test.c.014t.cplxlower0
test.c.015t.veclower
test.c.021t.cleanup_cfg1
test.c.051t.apply_inline
test.c.131r.expand
test.c.132r.sibling
test.c.134r.initvals
test.c.135r.unshare
test.c.136r.vregs
test.c.137r.into_cfglayout
test.c.138r.jump
test.c.157r.regclass
test.c.160r.outof_cfglayout
test.c.166r.split1
test.c.168r.dfinit
test.c.169r.mode-sw
test.c.171r.asmcons
test.c.174r.subregs_of_mode_init
test.c.175r.lreg
test.c.176r.greg
test.c.177r.subregs_of_mode_finish
test.c.180r.split2
test.c.182r.pro_and_epilogue
test.c.196r.stack
test.c.197r.alignments
test.c.200r.mach
test.c.201r.barriers
test.c.204r.eh-ranges
test.c.205r.shorten
test.c.206r.dfinish
test.s
```

Selected Dumps for Our Example Program

test.c.001t.tu
test.c.003t.original
test.c.004t.gimple
test.c.006t.vcg
test.c.007t.useless
test.c.010t.lower
test.c.011t.ehopt
test.c.012t.eh
test.c.013t.cfg
test.c.014t.cplxlower0
test.c.015t.veclower
test.c.021t.cleanup_cfg1
test.c.051t.apply_inline
test.c.131r.expand
test.c.132r.sibling
test.c.134r.initvals
test.c.135r.unshare
test.c.136r.vregs
test.c.137r.into_cfglayout
test.c.138r.jump
test.c.157r.regclass
test.c.160r.outof_cfglayout
test.c.166r.split1
test.c.168r.dfinit
test.c.169r.mode-sw
test.c.171r.asmcons
test.c.174r.subregs_of_mode_init
test.c.175r.lreg
test.c.176r.greg
test.c.177r.subregs_of_mode_finish
test.c.180r.split2
test.c.182r.pro_and_epilogue
test.c.196r.stack
test.c.197r.alignments
test.c.200r.mach
test.c.201r.barriers
test.c.204r.eh-ranges
test.c.205r.shorten
test.c.206r.dfinish
test.s

Important Phases of GCC



Gimplifier

- Three-address representation derived from GENERIC
  - Computation represented as a sequence of basic operations
  - Temporaries introduced to hold intermediate values
- Control construct are explicated into conditional jumps

Gimple: Translation of Composite Expressions

File: `test.c.004t.gimple`

```
if (a <= 12)
{
    a = a+b+c;
}
```

```
if (a <= 12)
{
    D.1199 = a + b;
    a = D.1199 + c;
}
else
{
}
```


Gimple: Translation of Higher Level Control Constructs

File: `test.c.004t.gimple`

```
while (a <= 7)
{
      a = a+1;
}
```

```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
{
      goto <D.1196>;
}
else
{
      goto <D.1198>;
}
<D.1198>:;
```

Gimple: Translation of Higher Level Control Constructs

File: test.c.004t.gimple

```
while (a <= 7)
{
     a = a+1;
}
```

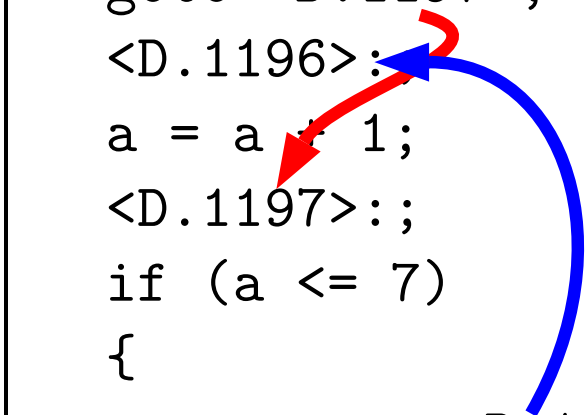
```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
{
      goto <D.1196>;
}
else
{
      goto <D.1198>;
}
<D.1198>:;
```

Gimple: Translation of Higher Level Control Constructs

File: `test.c.004t.gimple`

```
while (a <= 7)
{
     a = a+1;
}
```

```
goto <D.1197>;
<D.1196>:
a = a + 1;
<D.1197>:;
if (a <= 7)
{
      goto <D.1196>;
}
else
{
      goto <D.1198>;
}
<D.1198>:;
```

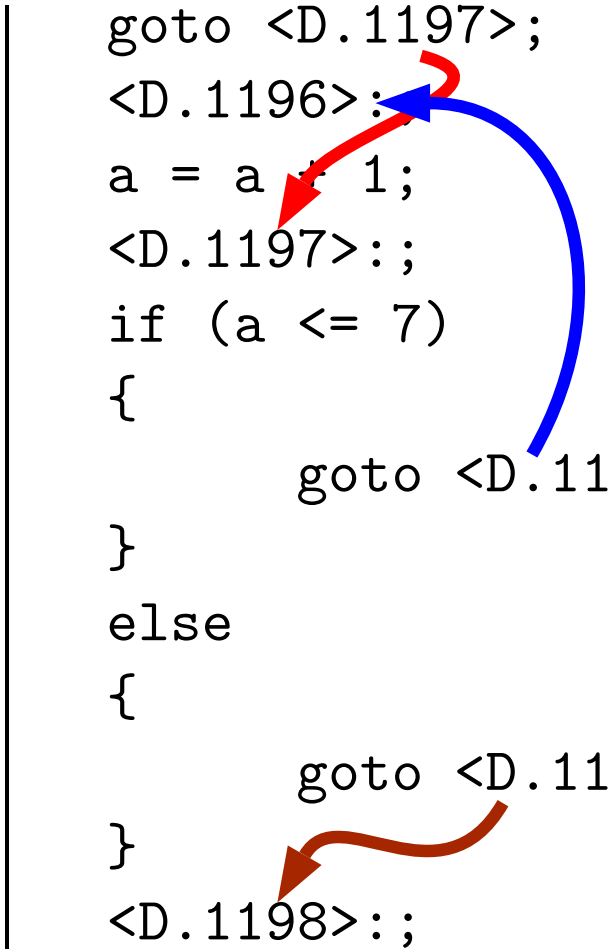


Gimple: Translation of Higher Level Control Constructs

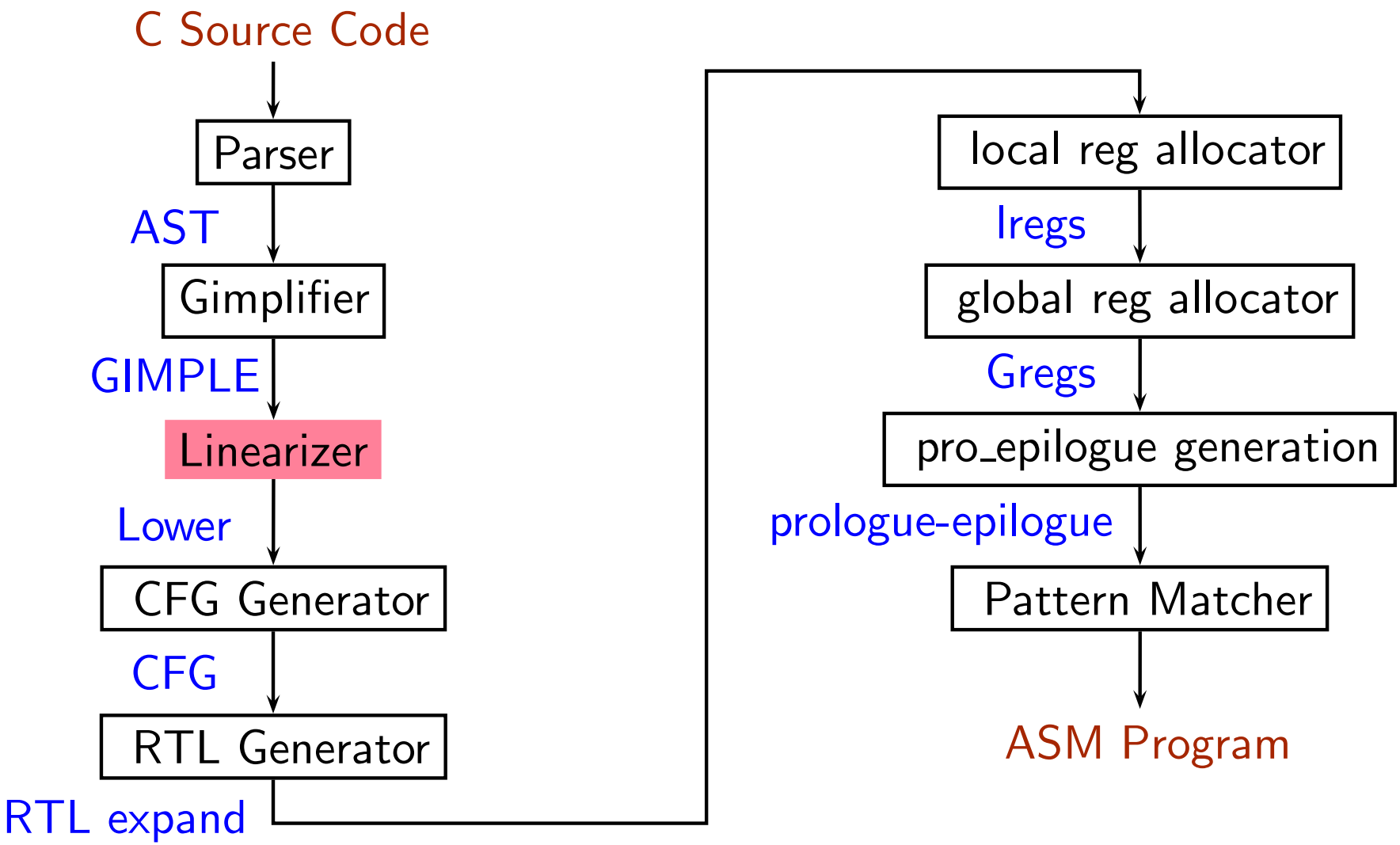
File: `test.c.004t.gimple`

```
while (a <= 7)
{
     a = a+1;
}
```

```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
{
      goto <D.1196>;
}
else
{
      goto <D.1198>;
}
<D.1198>:;
```



Important Phases of GCC



Lowering Gimple

File: `test.c.010t.lower`

```
if (a <= 12)
{
      D.1199 = a + b;
      a = D.1199 + c;
}
```

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:;
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:;
return;
```


Lowering Gimple

File: `test.c.010t.lower`

```
if (a <= 12)
{
      D.1199 = a + b;
      a = D.1199 + c;
}
```

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:;
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:;
return;
```

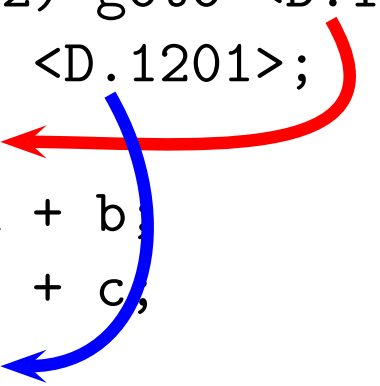
`if-then` translated in terms of conditional and unconditional gotos

Lowering Gimple

File: `test.c.010t.lower`

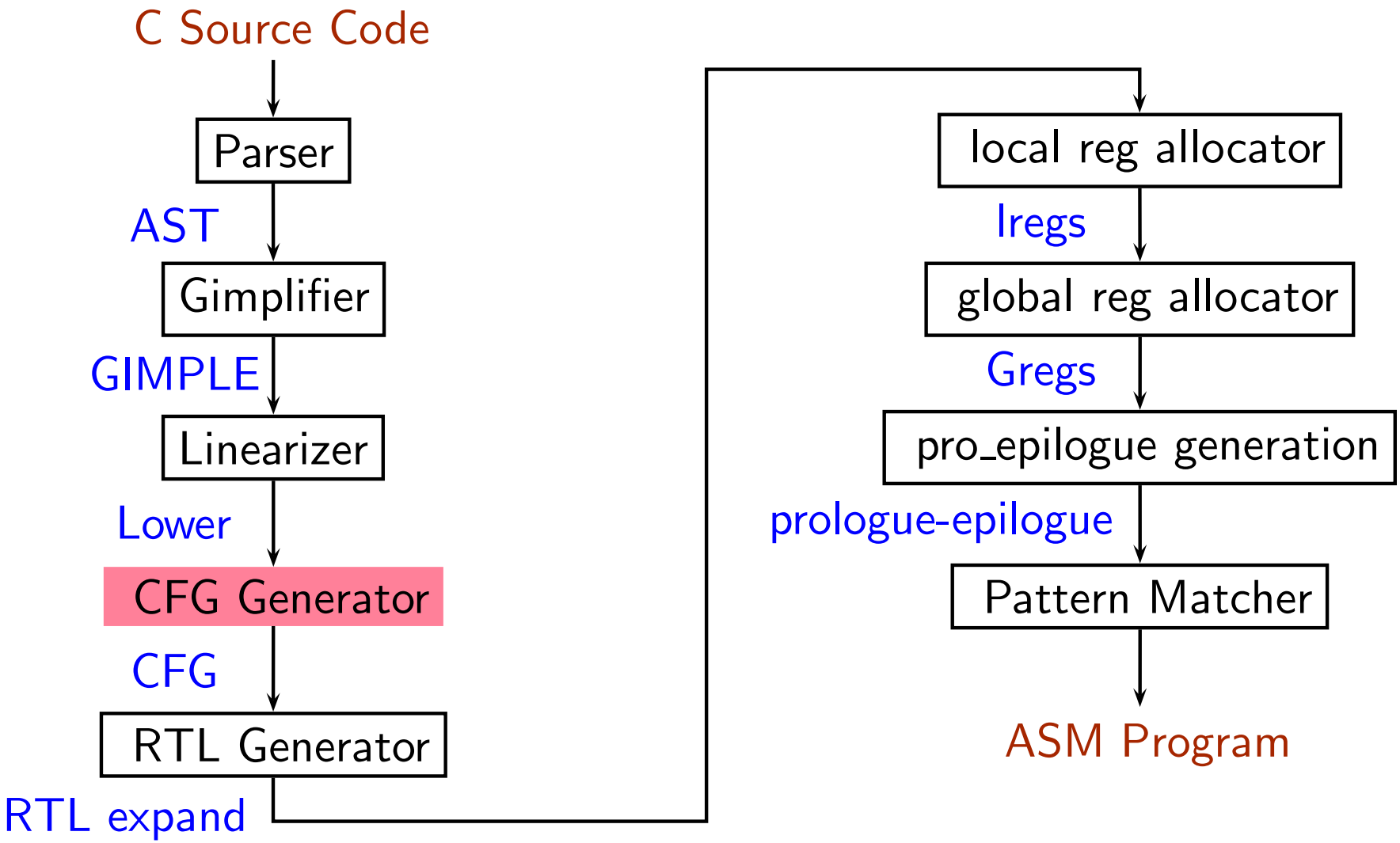
```
if (a <= 12)
{
      D.1199 = a + b;
      a = D.1199 + c;
}
```

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:;
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:;
return;
```



`if-then` translated in terms of conditional and unconditional gotos

Important Phases of GCC



Constructing the Control Flow Graph

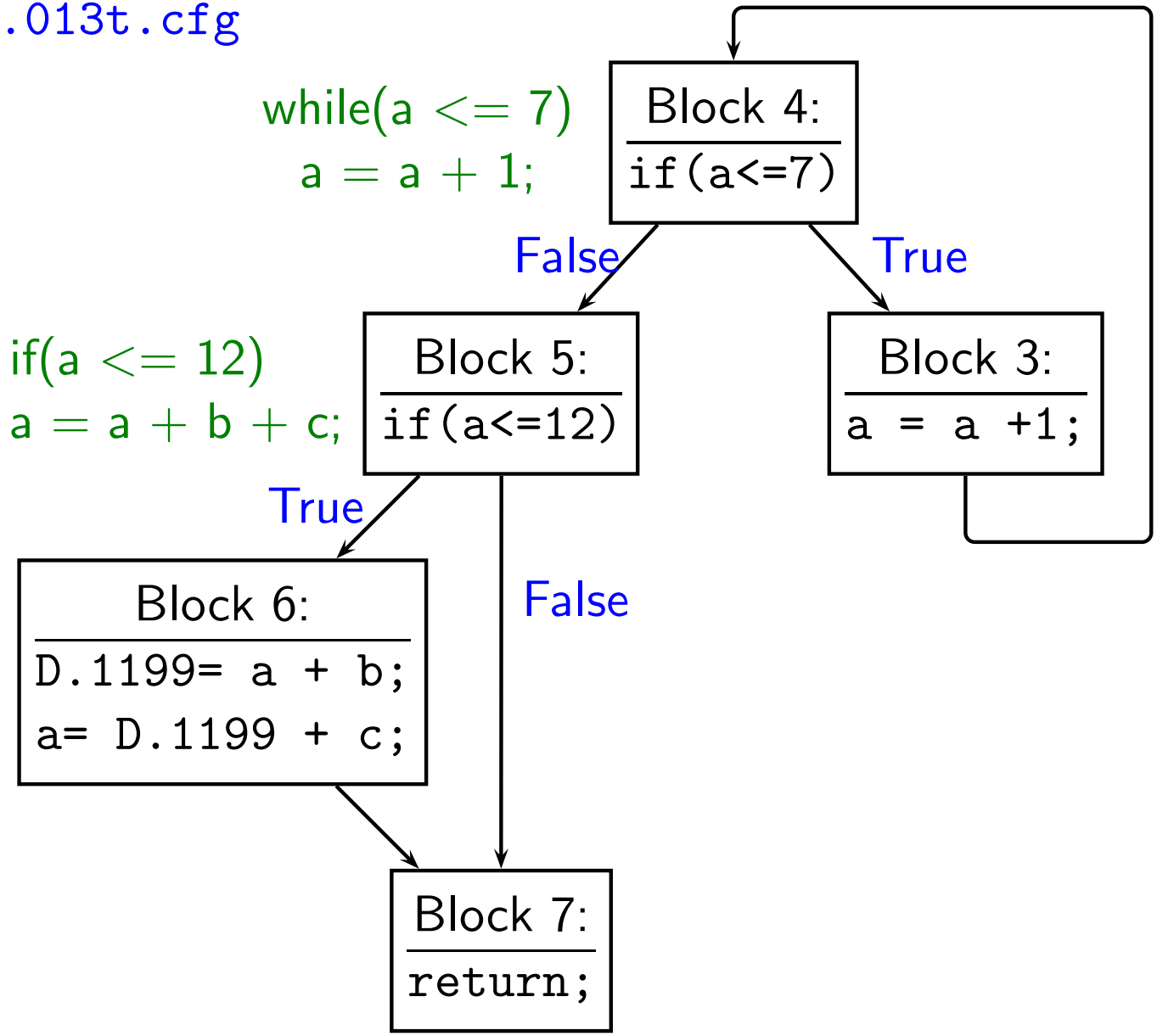
File: test.c.013t.cfg

```
if (a <= 12) goto <D.1200>;
else  goto <D.1201>;
<D.1200>:;
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:;
return;
```

```
# BLOCK 5
if (a <= 7)
    goto <bb 6>;
else
    goto <bb 7>;
# SUCC: 6 (true) 7 (false)
# BLOCK 6
D.1199 = a + b;
a = D.1199 + c;
# SUCC: 7 (fallthru)
# BLOCK 7
return;
# SUCC: EXIT
```

Control Flow Graph

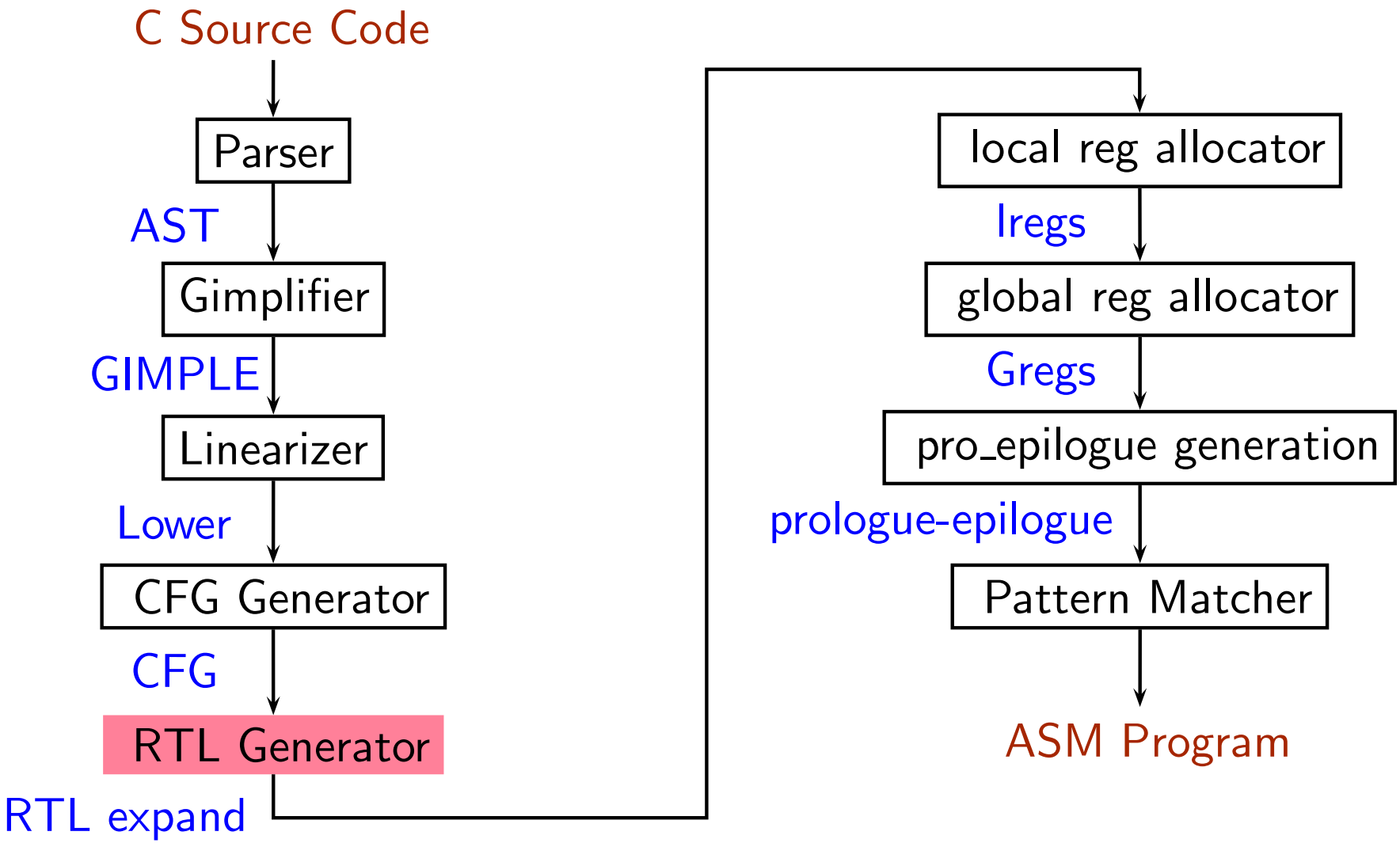
File: `test.c.013t.cfg`



Decisions that have been taken

- Three-address representation is generated
- All high level control flow structures are made explicit.
- Source code divided into interconnected blocks of sequential statements.
- This is a convenient structure for later analysis.

Important Phases of GCC



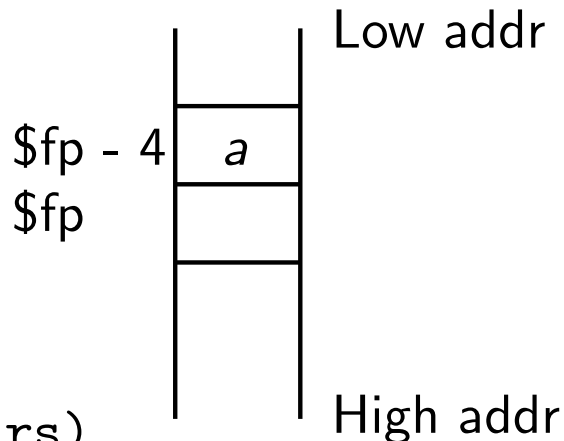
Expansion into RTL for i386 Port

Translation of $a = a + 1$

File: `test.c.031r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 0 (parallel [
    (set (mem/c/i:SI (plus:SI
                (reg/f:SI 54 virtual-stack-vars)
                (const int -4 [...])) [...])
            (plus:SI
                (mem/c/i:SI (plus:SI
                    (reg/f:SI 54 virtual-stack-vars)
                    (const int -4 [...])) [...])
                (const int 1 [...])))
    (clobber (reg:CC 17 flags))
    ]) -1 (nil))
```



Plus operation computes `$fp - 4` *as the address of variable* `a`

Expansion into RTL for i386 Port

Translation of $a = a + 1$

File: `test.c.031r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 0 (parallel[
    ( set (mem/c/i:SI (plus:SI
            (reg/f:SI 54 virtual-stack-vars)
            (const int -4 [...])) [...])
        (plus:SI
            (mem/c/i:SI (plus:SI
                (reg/f:SI 54 virtual-stack-vars)
                (const int -4 [...])) [...])
            (const int 1 [...])))
  (clobber (reg:CC 17 flags))
  ]) -1 (nil))
```

Set denotes assignment

Expansion into RTL for i386 Port

Translation of $a = a + 1$

File: `test.c.031r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 0 (parallel [
    ( set (mem/c/i:SI (plus:SI
              (reg/f:SI 54 virtual-stack-vars)
              (const int -4 [...])) [...])
          (plus:SI
              (mem/c/i:SI (plus:SI
                  (reg/f:SI 54 virtual-stack-vars)
                  (const int -4 [...])) [...])
              (const int 1 [...])))
   (clobber (reg:CC 17 flags))
   ]) -1 (nil))
```

1 is added to variable a

Expansion into RTL for i386 Port

Translation of $a = a + 1$

File: `test.c.031r.expand`

stack($fp - 4) = stack($fp - 4) + 1
|| flags=?

```
(insn 12 11 0 (parallel [
    ( set (mem/c/i:SI (plus:SI
            (reg/f:SI 54 virtual-stack-vars)
            (const int -4 [...])) [...])
        (plus:SI
            (mem/c/i:SI (plus:SI
                (reg/f:SI 54 virtual-stack-vars)
                (const int -4 [...])) [...])
            (const int 1 [...])))
   (clobber (reg:CC 17 flags))
   ]) -1 (nil))
```

Condition Code register is clobbered to record possible side effect of plus

Flags in RTL Expressions

Meanings of some of the common flags

`/c` memory reference that does not trap
`/i` scalar that is not part of an aggregate
`/f` register that holds a pointer

Expansion into RTL in spim Port

Translation of $a = a + 1$

File: `test.c.031r.expand`

r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40

```
(insn 7 6 8 test.c:6 (set (reg:SI 39)
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
            (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 test.c:6 (set (reg:SI 40)
      (plus:SI (reg:SI 39)
           (const_int 1 [...]))) -1 (nil))
(insn 9 8 0 test.c:6 (set
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
           (const_int -4 [...])) [...])
      (reg:SI 40)) -1 (nil))
```

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated

Expansion into RTL in spim Port

Translation of $a = a + 1$

File: `test.c.031r.expand`

r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40

```
(insn 7 6 8 test.c:6 (set (reg:SI 39)
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
            (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 test.c:6 (set (reg:SI 40)
      (plus:SI (reg:SI 39)
           (const_int 1 [...]))) -1 (nil))
(insn 9 8 0 test.c:6 (set
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
           (const_int -4 [...])) [...])
      (reg:SI 40)) -1 (nil))
```

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated

Expansion into RTL in spim Port

Translation of $a = a + 1$

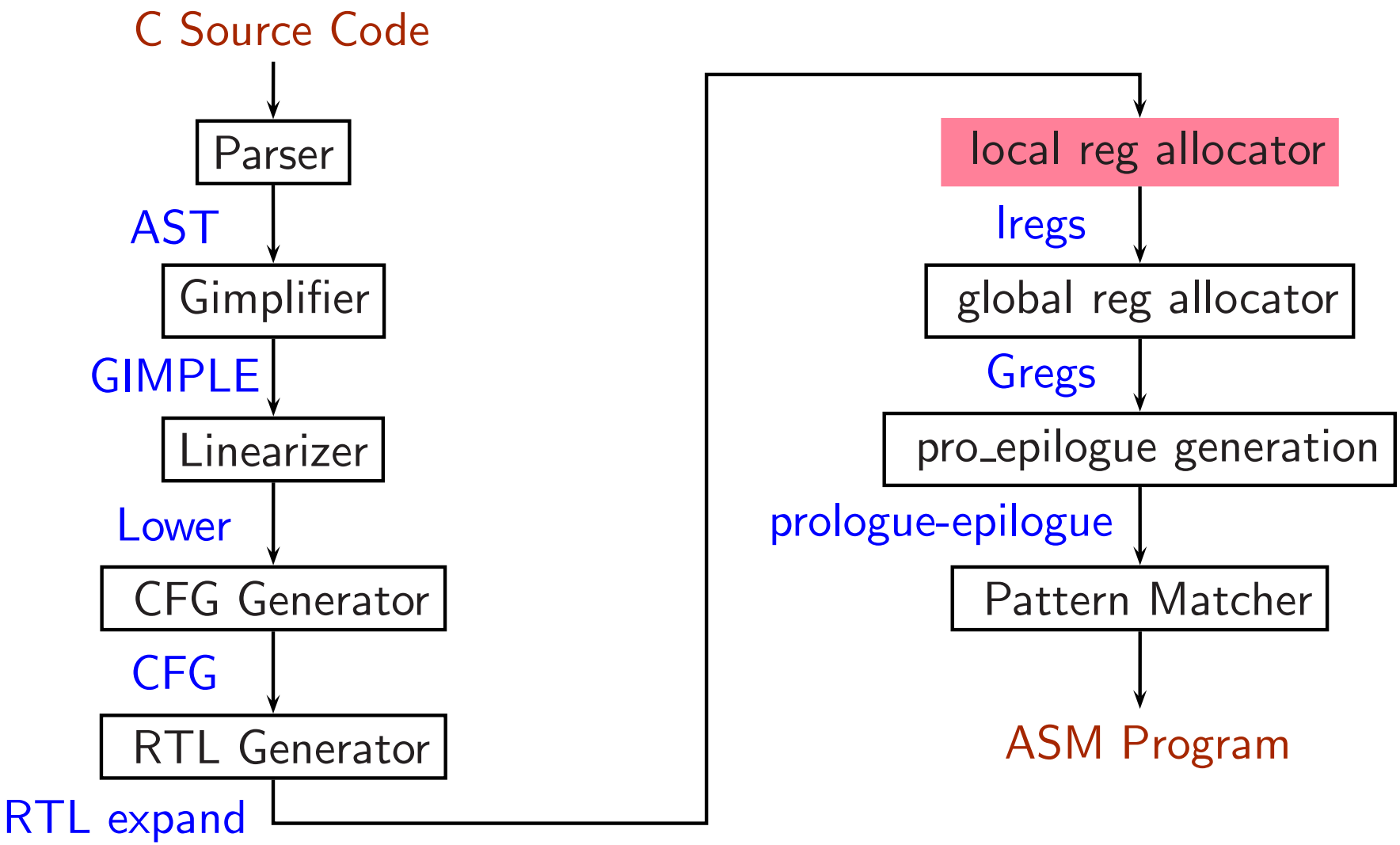
File: `test.c.031r.expand`

r39=stack($fp - 4)
r40=r39+1
stack($fp - 4)=r40

```
(insn 7 6 8 test.c:6 (set (reg:SI 39)
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
            (const_int -4 [...])) [...])) -1 (nil))
(insn 8 7 9 test.c:6 (set (reg:SI 40)
      (plus:SI (reg:SI 39)
           (const_int 1 [...]))) -1 (nil))
(insn 9 8 0 test.c:6 (set
      (mem/c/i:SI (plus:SI (reg/f:SI 33 virtual-stack-vars)
           (const_int -4 [...])) [...])
      (reg:SI 40)) -1 (nil))
```

In spim, a variable is loaded into register to perform any instruction, hence three instructions are generated

Important Phases of GCC



Local Register Allocation: i386 Port

File: `test.c.175r.lreg`

```
(insn 12 11 13 4 (parallel [
        (set (mem/c/i:SI (plus:SI
                                (reg/f:SI 20 frame)
                                (const_int -4 [...])) [...])
                (plus:SI (mem/c/i:SI
                        (plus:SI
                                (reg/f:SI 20 frame)
                                (const_int -4 [...])) [...])
                        (const_int 1 [...])))
            (clobber (reg:CC 17 flags))
        ]) 249 *addsi_1
                (expr_list:REG_UNUSED (reg:CC 17 flags) (nil))
```

Identifies candidates for local register allocation ≡
Definitions which have all uses within the same block

Local Register Allocation: i386 Port

File: `test.c.175r.lreg`

```
Basic block 3 , prev 2, next 4, loop_depth 0, count 0, freq 0.
Predecessors:  4
;; bb 3 artificial_uses:  u-1(6) u-1(7) u-1(16) u-1(20)
;; lr  in         6 [bp] 7 [sp] 16 [argp] 20 [frame]
;; lr  use        6 [bp] 7 [sp] 16 [argp] 20 [frame]
;; lr  def        17 [flags]
```

Identifies candidates for local register allocation ≡
Definitions which have all uses within the same block

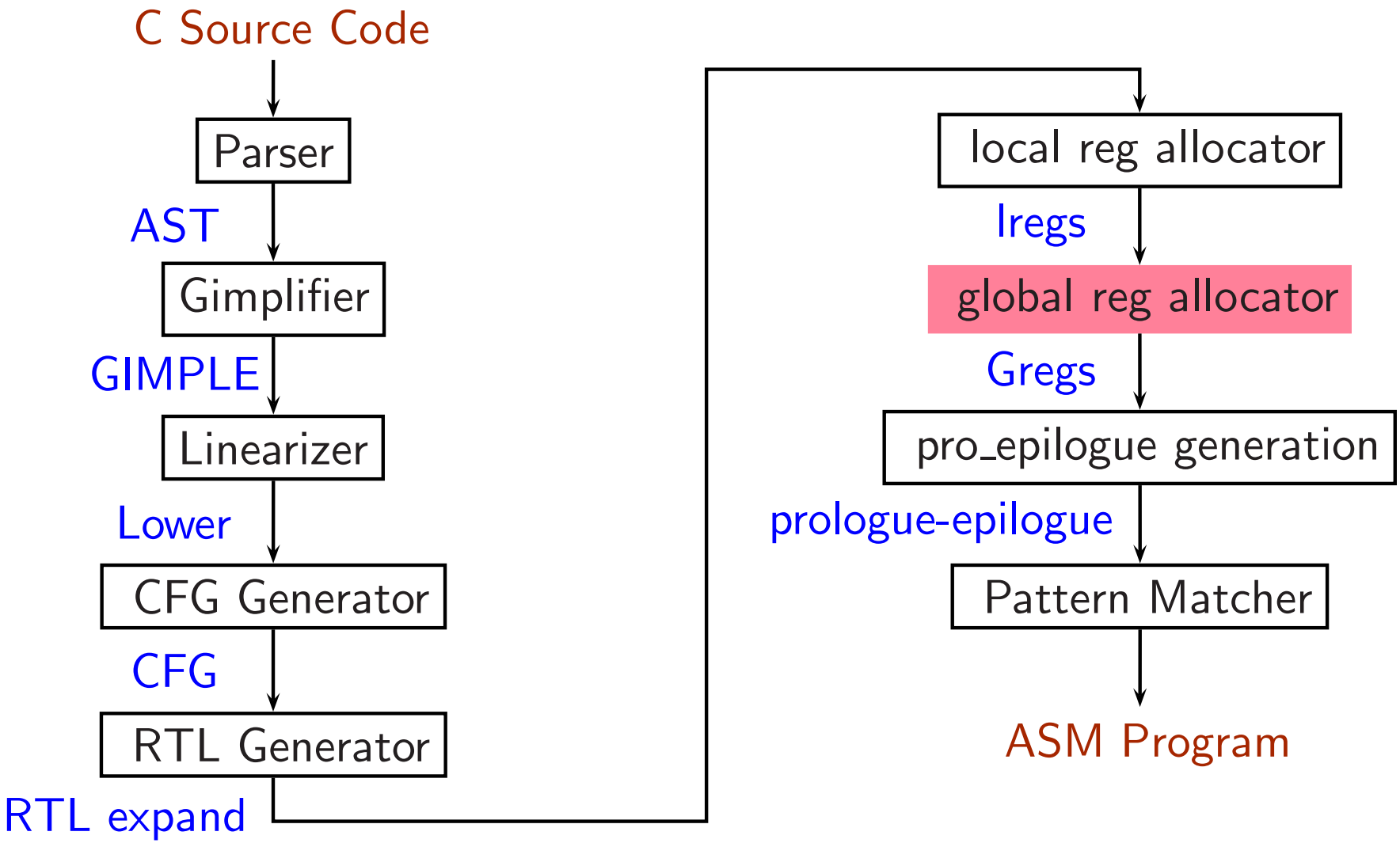
Local Register Allocation: spim Port

File: test.c.175r.lreg

```
(insn 7 6 8 2 (set (reg:SI 39)
            (mem/c/i:SI (plus:SI (reg/f:SI 1 $at)
                                 (const_int -4 [...])) [...]))
                                       4 *IITB_move_from_mem (nil))
(insn 8 7 9 2 (set (reg:SI 40)
                   (plus:SI (reg:SI 39)
                            (const_int 1 [...]))) 12 addsi3
                   (expr_list:REG DEAD (reg:SI 39)(nil)))
(insn 9 8 12 2 (set
                  (mem/c/i:SI (plus:SI (reg/f:SI 1 $at)
                                       (const_int -4 [...])) [...])
                  (reg:SI 40)) 5 *IITB_move_to_mem
                 (expr_list:REG DEAD(reg:SI 40) (nil)))
```

Discovers the last uses of reg:SI 39 *and* reg:SI 40

Important Phases of GCC



Global Register Allocation: i386 Port

File: test.c.176r.greg

```
(insn 12 11 13 4 (parallel [
        (set (mem/c/i:SI (plus:SI
                                (reg/f:SI 20 bp)
                                (const_int -4 [...])) [...])
                (plus:SI (mem/c/i:SI
                        (plus:SI
                                (reg/f:SI 20 bp)
                                (const_int -4 [...])) [...])
                        (const_int 1 [...])))
            (clobber (reg:CC 17 flags))
        ]) 249 *addsi_1 (nil))
```

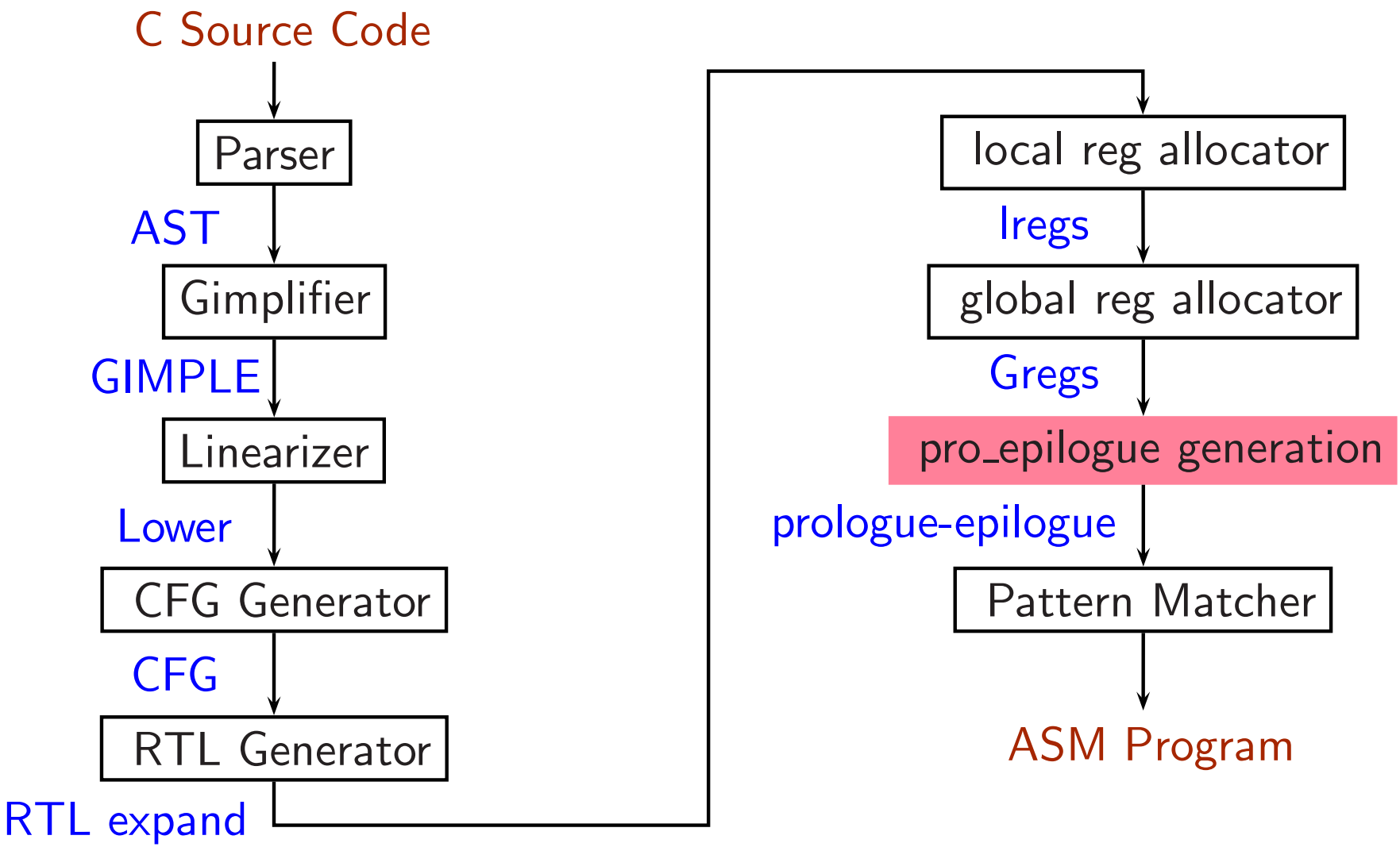

Global Register Allocation: spim Port

File: test.c.176r.greg

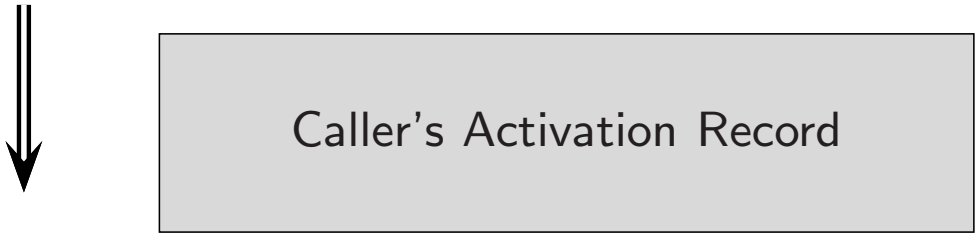
```
(insn 7 6 8 3 test.c:4 (set (reg:SI 2 $v0 [39])
          (mem/c/i:SI (plus:SI (reg/f:SI 1 $fp )
                (const_int -4 [...])) [...]))
                     4 *IITB_move_from_mem (nil))
(insn 8 7 9 3 test.c:4 (set (reg:SI $v0 [40])
          (plus:SI (reg:SI $v0 [39])
                (const_int 1 [...]))) 12 addsi3 (nil))
(insn 9 8 18 3 test.c:4 (set
                (mem/c/i:SI (plus:SI (reg/f:SI 1 $fp )
                      (const_int -4 [...])) [...])
                (reg:SI $v0 40)) 5 *IITB_move_to_mem (nil))
```

$v0 is used for both reg:SI 39 *and* reg:SI 40

Important Phases of GCC

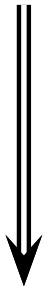


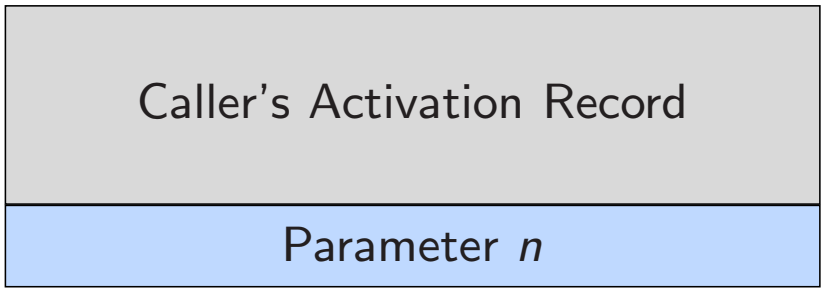
Activation Record Structure in Spim



Activation Record Structure in Spim

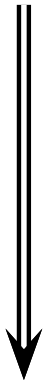
Caller's Responsibility

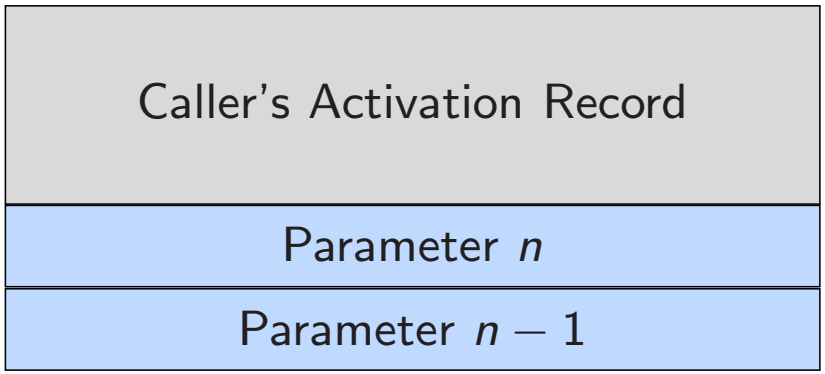




Activation Record Structure in Spim

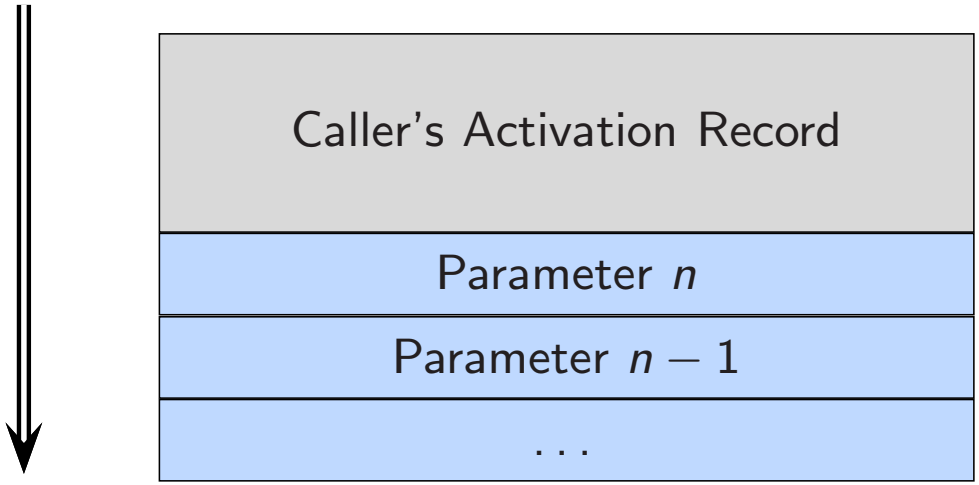
Caller's Responsibility





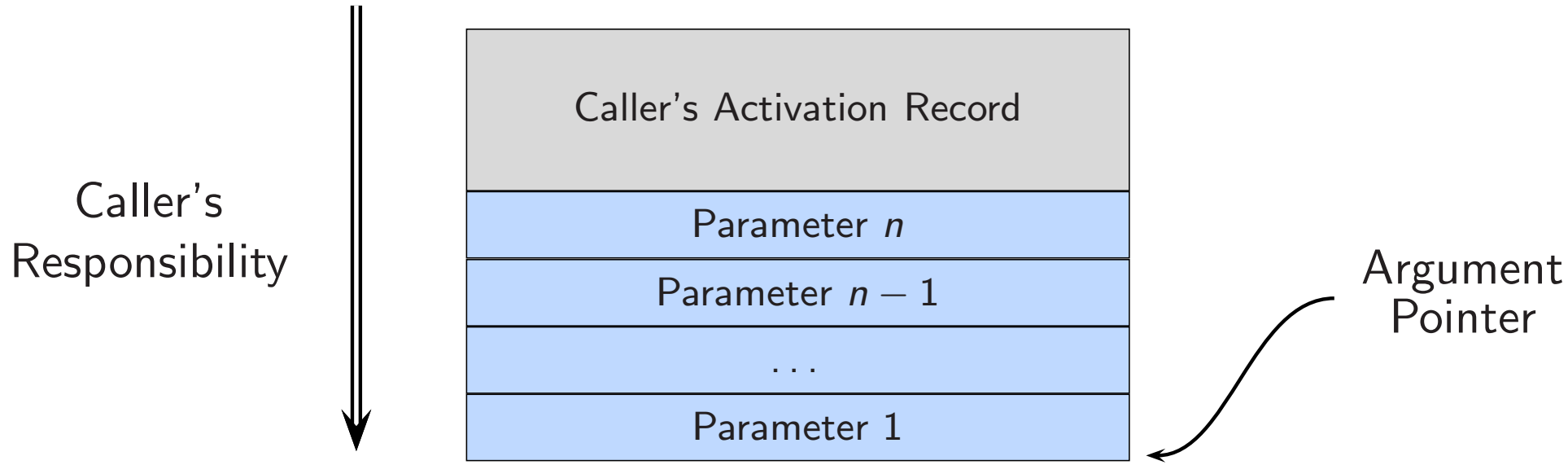
Activation Record Structure in Spim

Caller's Responsibility

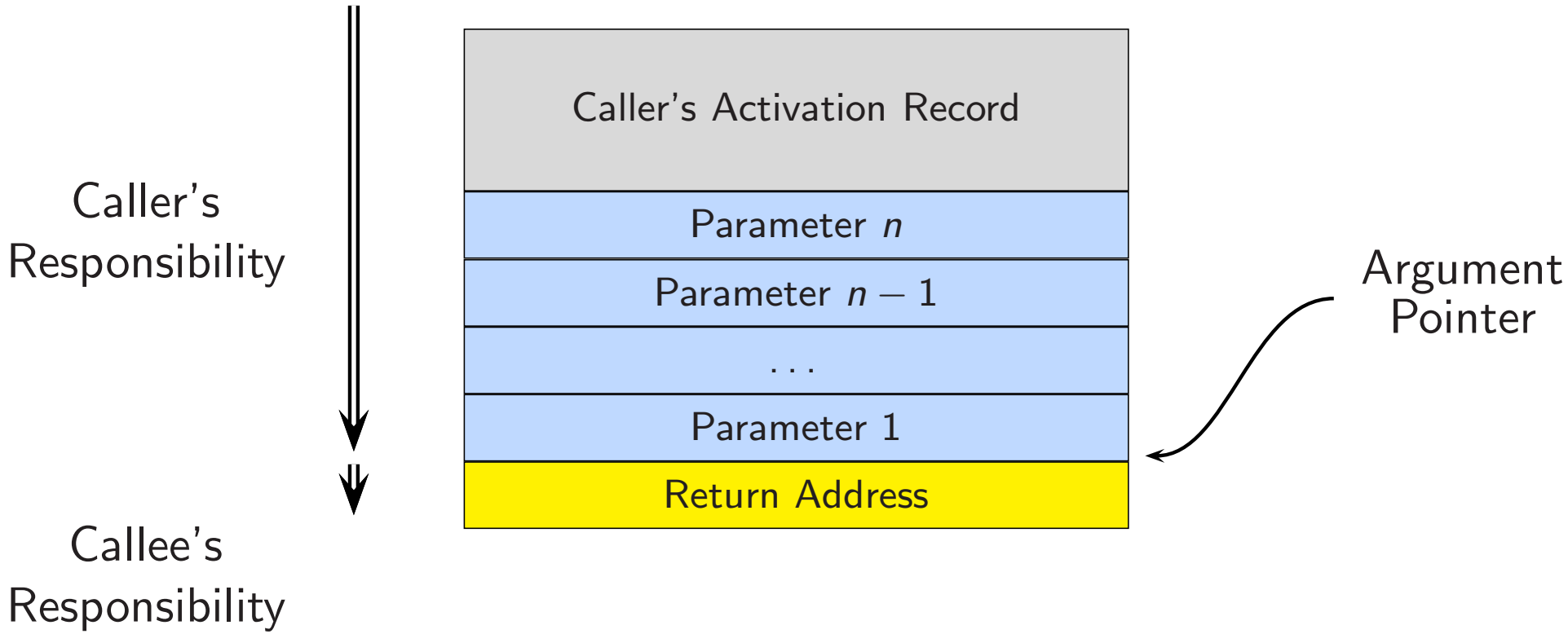


Argument Pointer

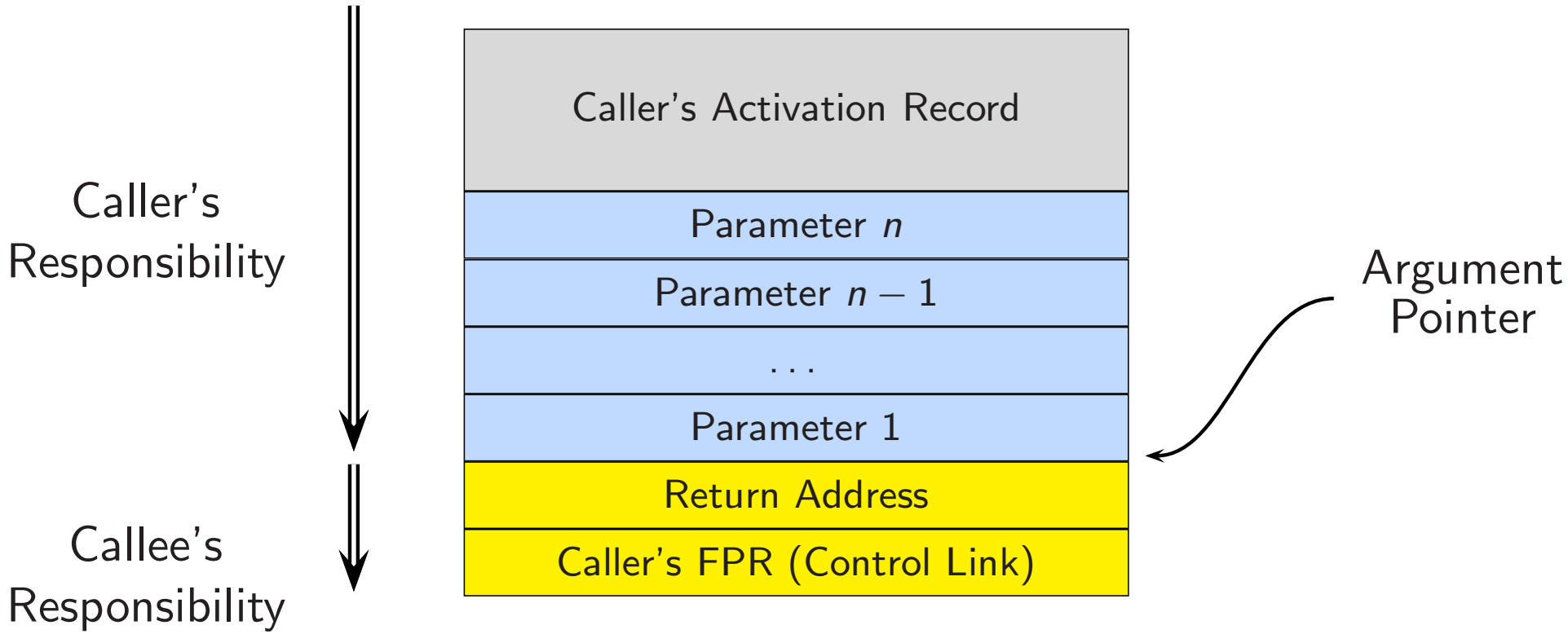
Activation Record Structure in Spim



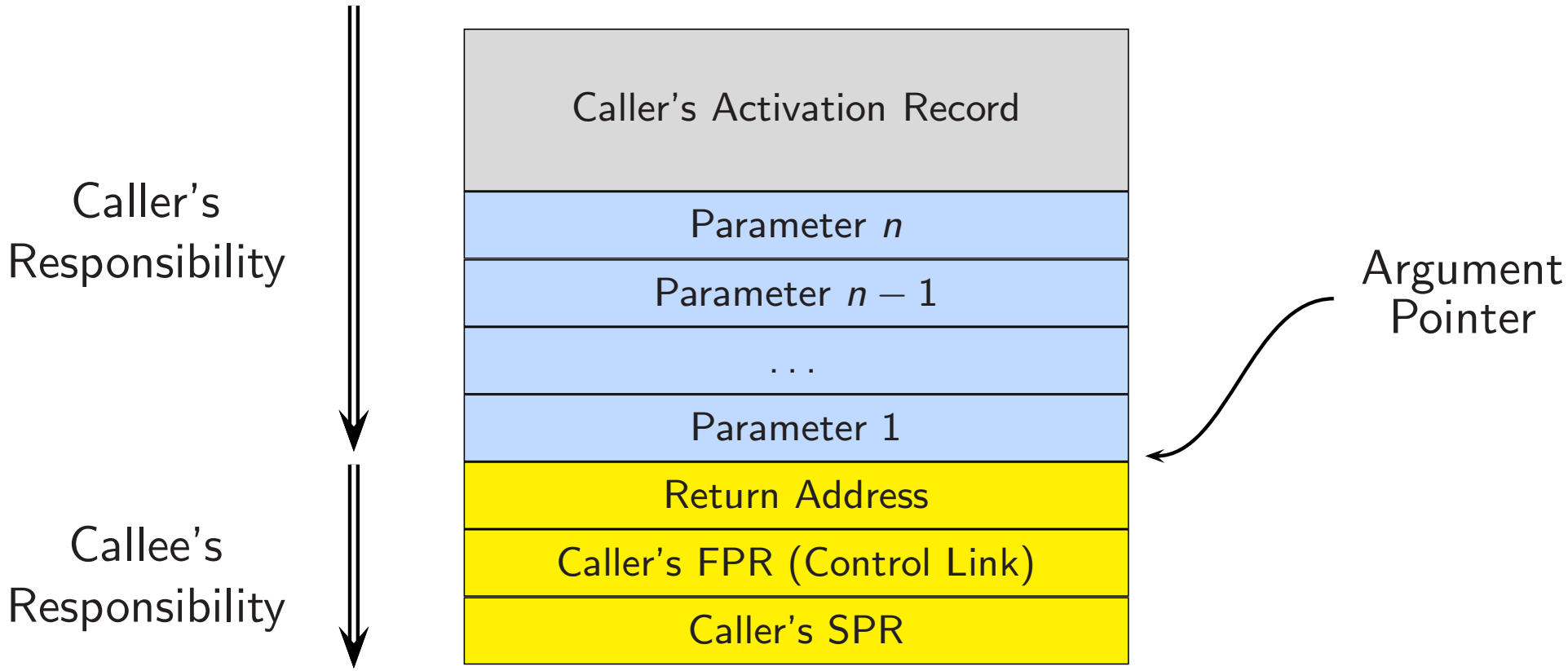
Activation Record Structure in Spim



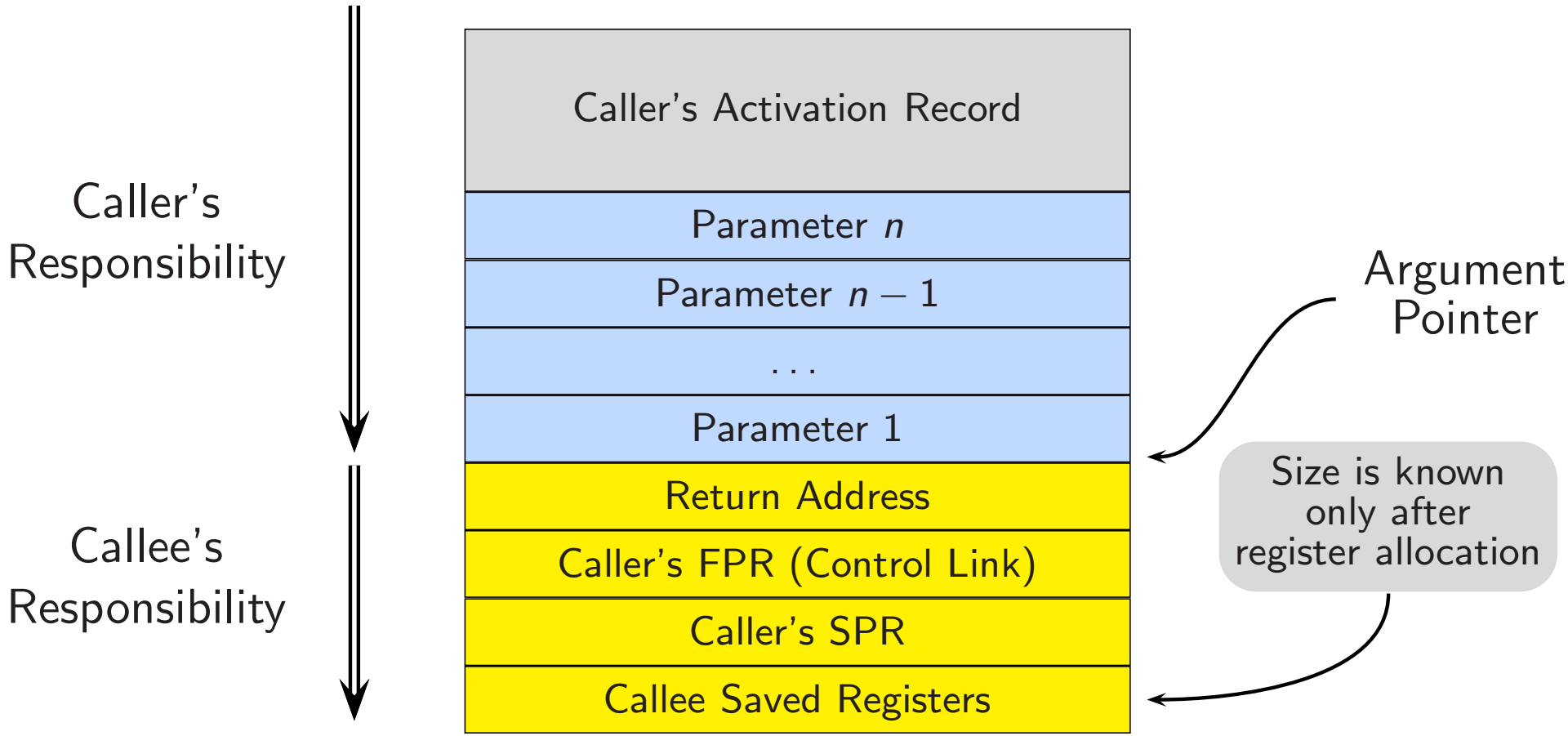
Activation Record Structure in Spim



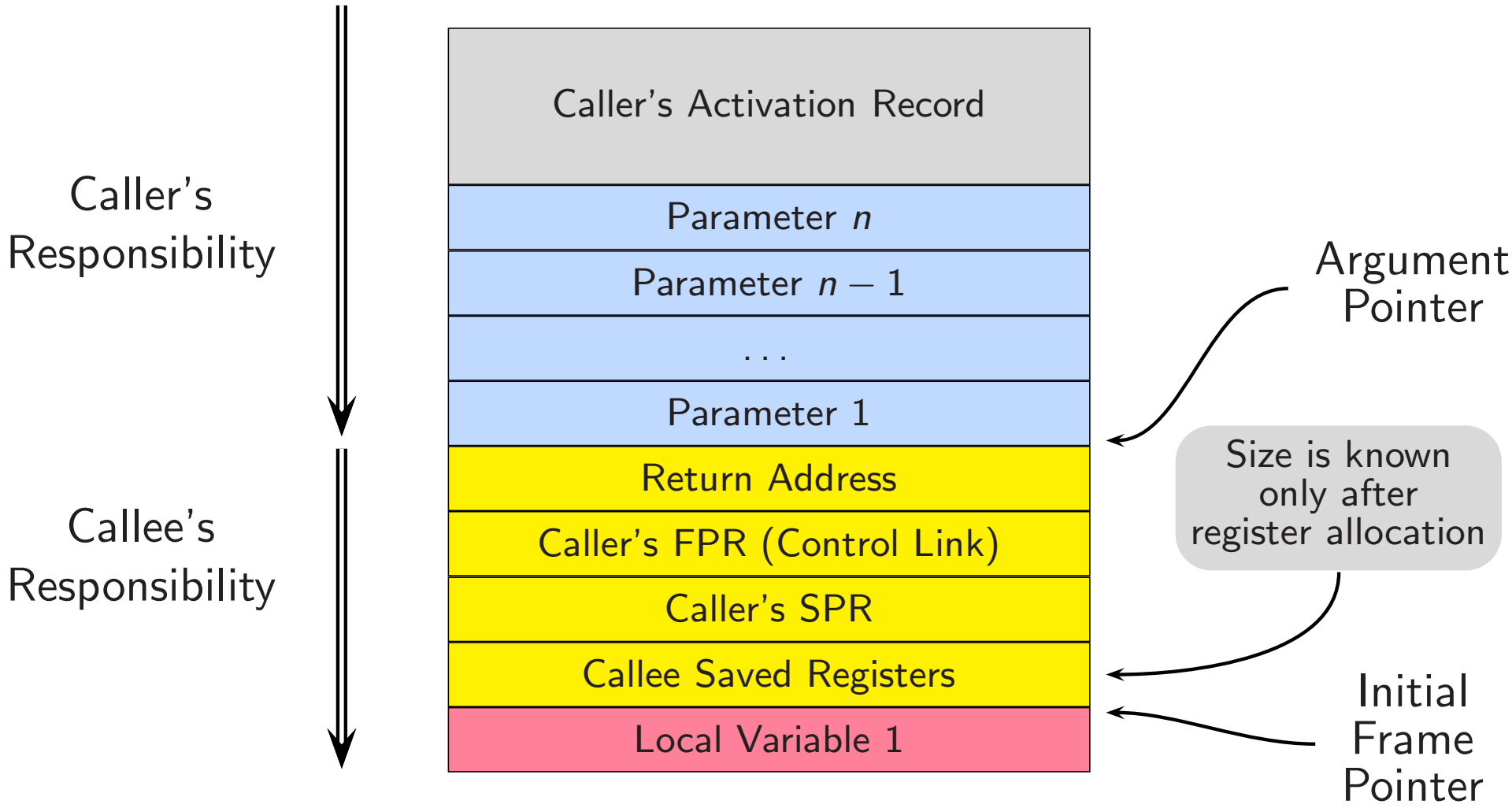
Activation Record Structure in Spim



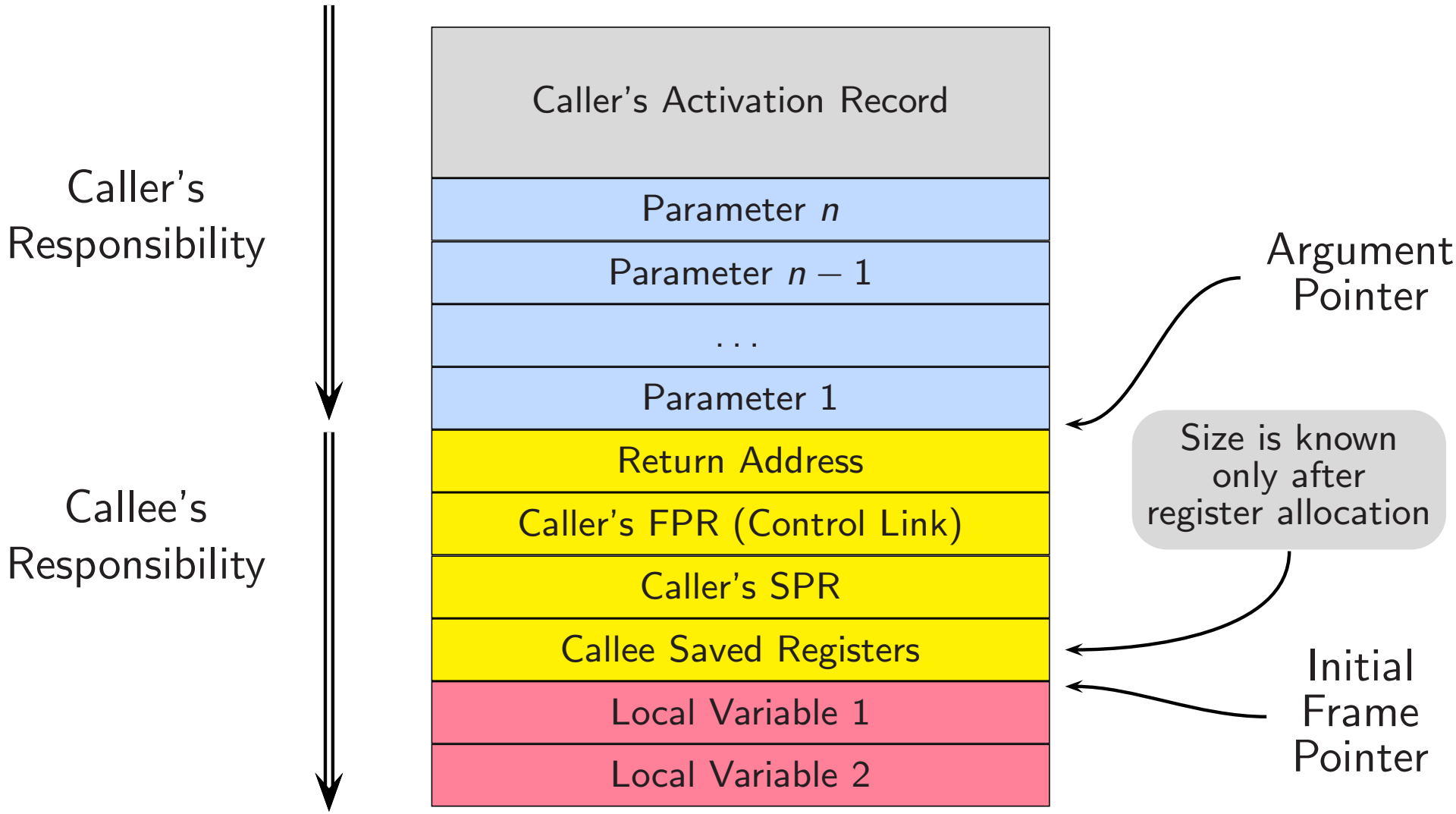
Activation Record Structure in Spim



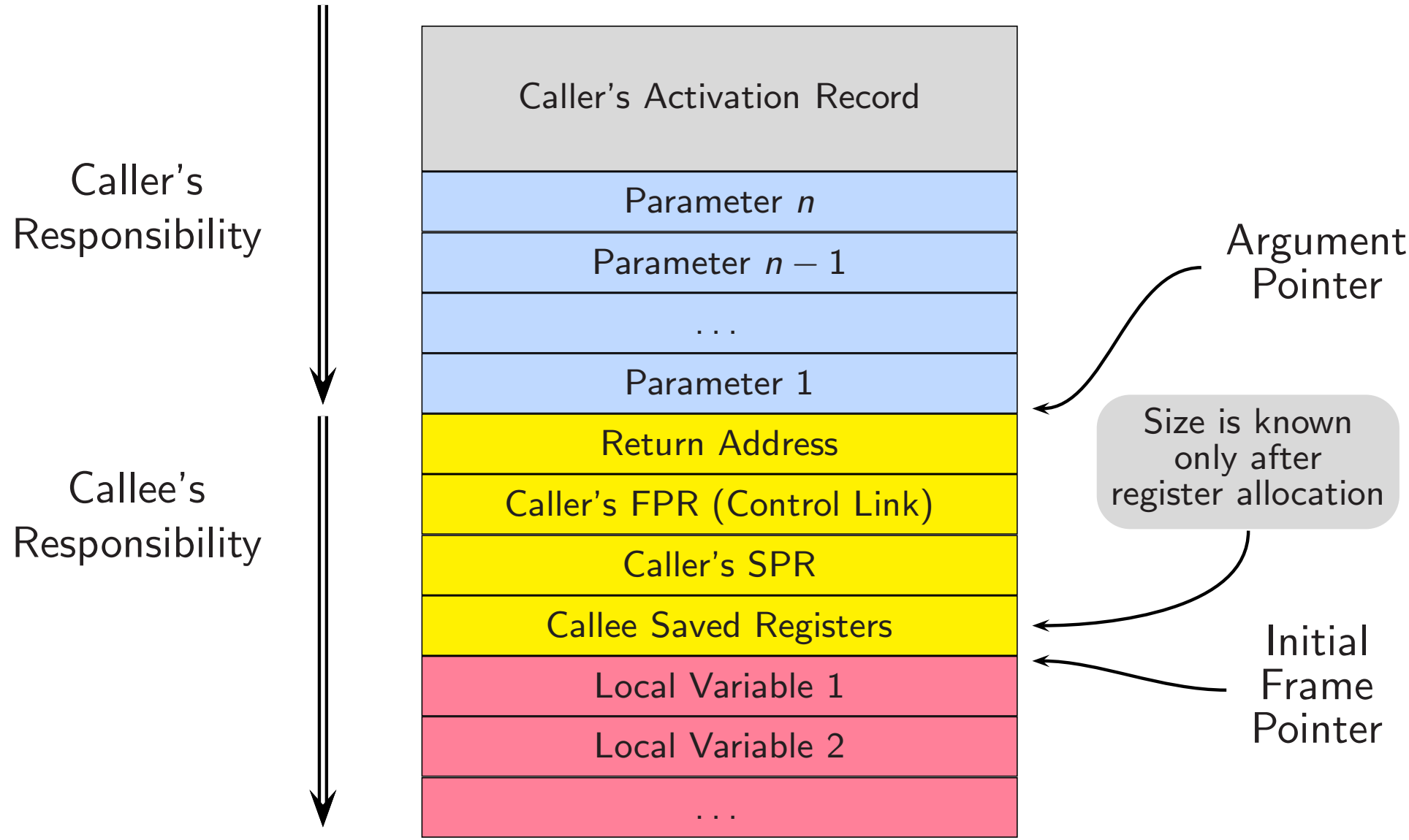
Activation Record Structure in Spim



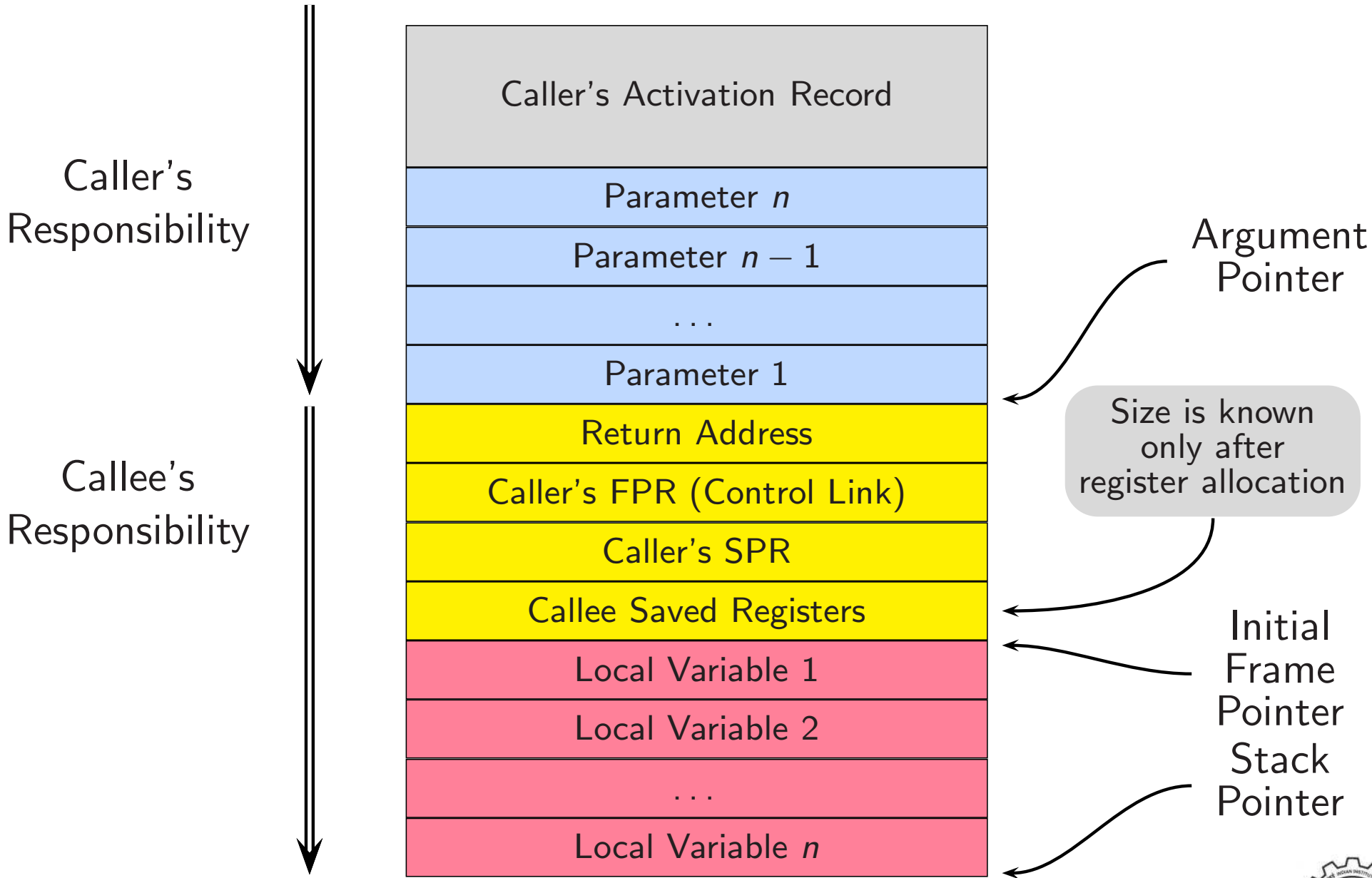
Activation Record Structure in Spim



Activation Record Structure in Spim



Activation Record Structure in Spim



RTL for Function Calls in spim

Calling function	Called function
• Allocate memory for actual parameters on stack • Copy actual parameters • **Call function** • Get result from stack (pop) • Deallocate memory for activation record (pop)	• Allocate memory for return value (push) • Store mandatory callee save registers (push) • Set frame pointer • Allocate local variables (push) • **Execute code** • Put result in return value space • Deallocate local variables (pop) • Load callee save registers (pop) • Return

Prologue and Epilogue: spim

File: test.c.182r.pro_and_epilogue

```
(insn 17 3 18 2 test.c:2
   (set (mem:SI (reg/f:SI 29 $sp) [0 S4 A8])
        (reg:SI 31 $ra)) -1 (nil))
(insn 18 17 19 2 test.c:2
   (set (mem:SI (plus:SI (reg/f:SI 29 $sp)
                    (const_int -4 [...])) [...])
      (reg/f:SI 29 $sp)) -1 (nil))
(insn 19 18 20 2 test.c:2 (set
      (mem:SI (plus:SI (reg/f:SI 29 $sp)
                  (const_int -8 [...])) [...])
      (reg/f:SI 30 $fp)) -1 (nil))
(insn 20 19 21 2 test.c:2 (set (reg/f:SI 30 $fp)
       (reg/f:SI 29 $sp)) -1 (nil))
(insn 21 20 22 2 test.c:2 (set (reg/f:SI 29 $sp)
    (plus:SI (reg/f:SI 30 $fp)
             (const_int -32 [...]))) -1 (nil))
```

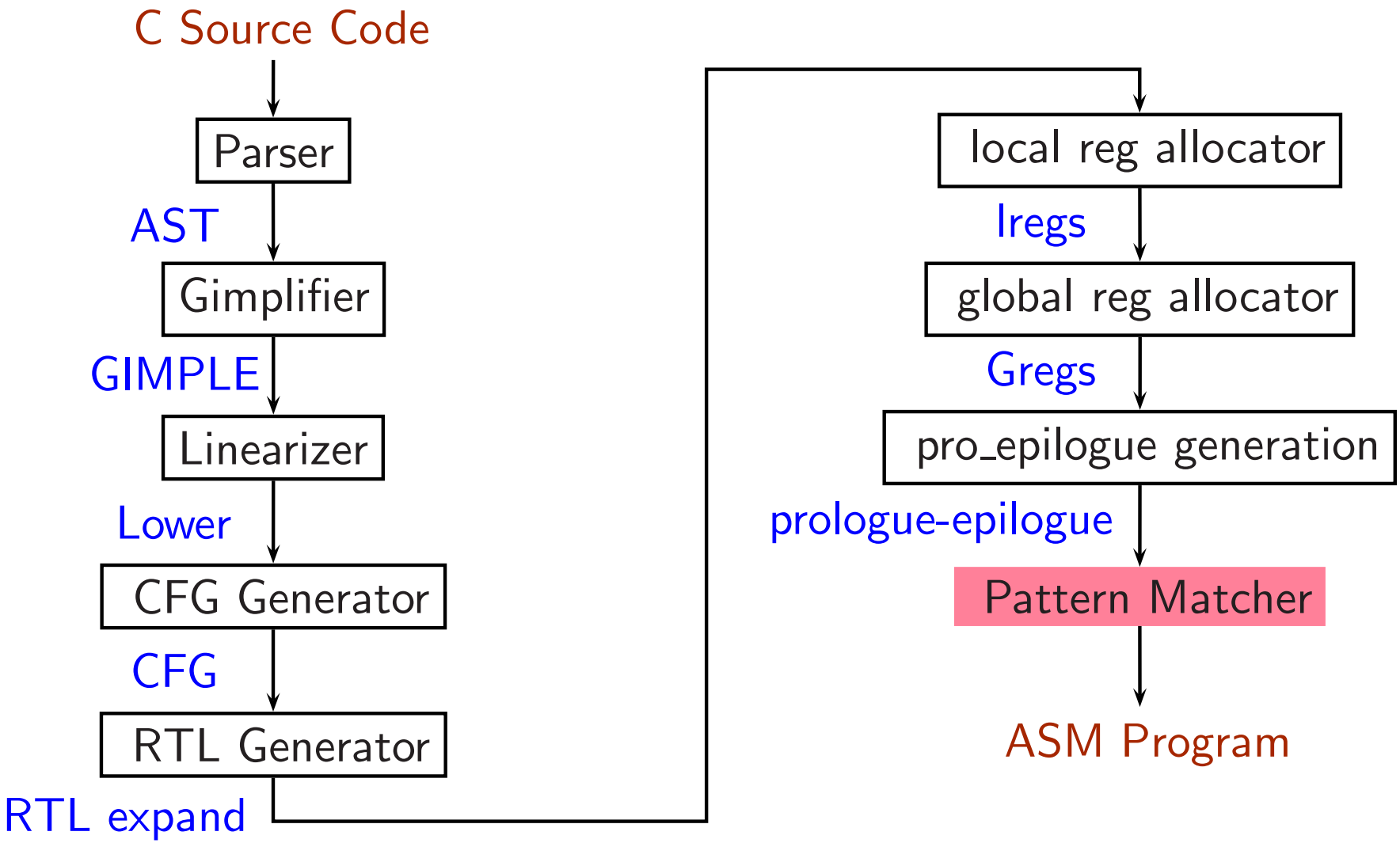

Prologue and Epilogue: spim

File: test.c.182r.pro_and_epilogue

```
(insn 17 3 18 2 test.c:2
   (set (mem:SI (reg/f:SI 29 $sp) [0 S4 A8])
        (reg:SI 31 $ra)) -1 (nil))
(insn 18 17 19 2 test.c:2
   (set (mem:SI (plus:SI (reg/f:SI 29 $sp)
                    (const_int -4 [...])) [...])
      (reg/f:SI 29 $sp)) -1 (nil))
(insn 19 18 20 2 test.c:2 (set
      (mem:SI (plus:SI (reg/f:SI 29 $sp)
                  (const_int -8 [...])) [...])
      (reg/f:SI 30 $fp)) -1 (nil))
(insn 20 19 21 2 test.c:2 (set (reg/f:SI 30 $fp)
       (reg/f:SI 29 $sp)) -1 (nil))
(insn 21 20 22 2 test.c:2 (set (reg/f:SI 29 $sp)
    (plus:SI (reg/f:SI 30 $fp)
             (const_int -32 [...]))) -1 (nil))
```

```
sw $ra, 0($sp)
sw $sp, 4($sp)
sw $fp, 8($sp)
move $fp,$sp
addi $sp,$fp,32
```


Important Phases of GCC



Assembly

Assembly Code for a= a + 1;

File: `test.s`

For spim	For i386

`lw $v0, -8($fp)`
`addi $v0, $v0, 1`
`sw $v0, -8($fp)`
	`addl $1, -8(%ebp)`

3 instruction required in spim and one in i386

Gray Box Probing of GCC: Conclusions

- Source code is transformed into assembly by lowering the abstraction level step by step to bring it close to machine architecture
- This transformation can be understood to a large extent by observing inputs and output of the different steps in the transformation
- In gcc, the output of almost all the passes can be examined
- The complete list of dumps can be figured out by the command

```
man gcc
```


Part 5

Introduction to Parallelization and Vectorization

A Taxonomy of Parallel Computation

	Single Program	Multiple Programs
Single Data	SPSD	MPSD
Multiple Data	SPMD	MPMD

A Taxonomy of Parallel Computation

	Single Program		Multiple Programs
	Single Instruction	Multiple Instructions	
Single Data	SISD	MISD	MPSD
Multiple Data	SIMD	MIMD	MPMD

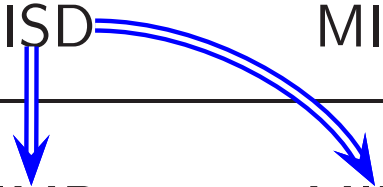
A Taxonomy of Parallel Computation

	Single Program		Multiple Programs
	Single Instruction	Multiple Instructions	
Single Data	SISD	?	?
Multiple Data	SIMD	MIMD	MPMD

Redundant computation for validation of intermediate steps

A Taxonomy of Parallel Computation

	Single Program		Multiple Programs
	Single Instruction	Multiple Instructions	
Single Data	SISD	MISD	MPSD
Multiple Data	SIMD	MIMD	MPMD



Transformations performed by a compiler

Vectorization: SISD ⇒ SIMD

- Parallelism in executing operation on shorter operands (8-bit, 16-bit, 32-bit operands)
- Existing 32 or 64-bit arithmetic units used to perform multiple operations in parallel
  A 64 bit word ≡ a vector of 2×(32 bits), 4×(16 bits), or 8×(8 bits)

Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

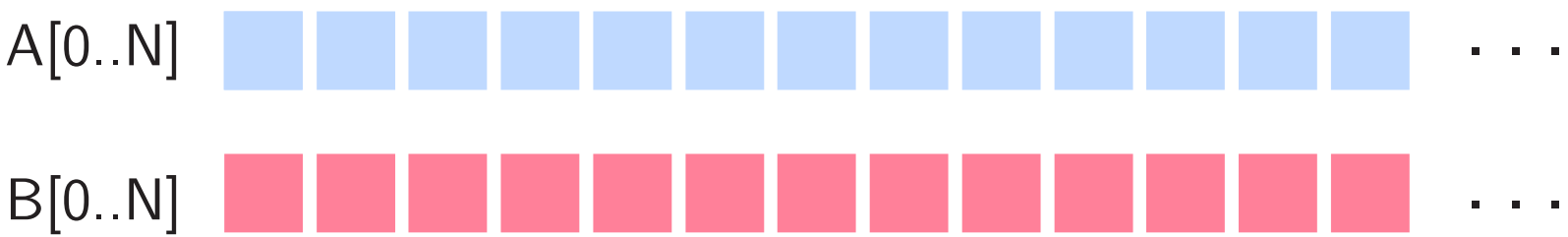

Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



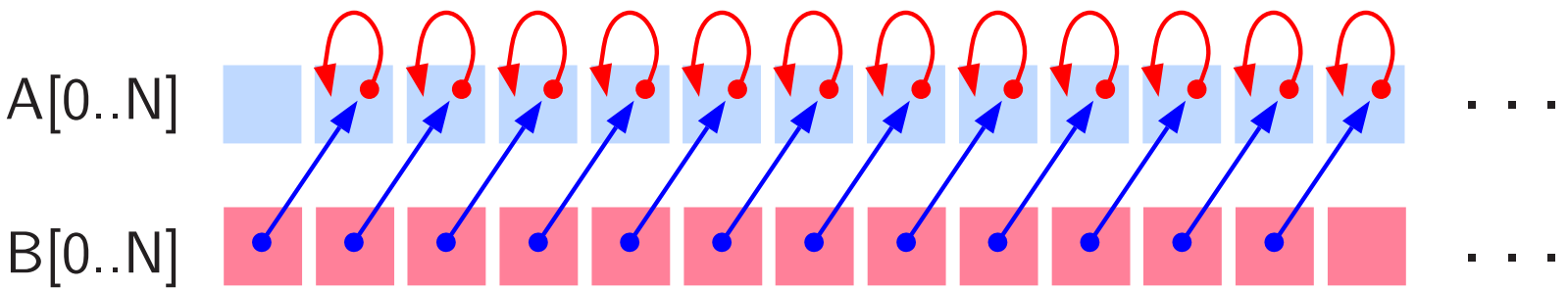
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



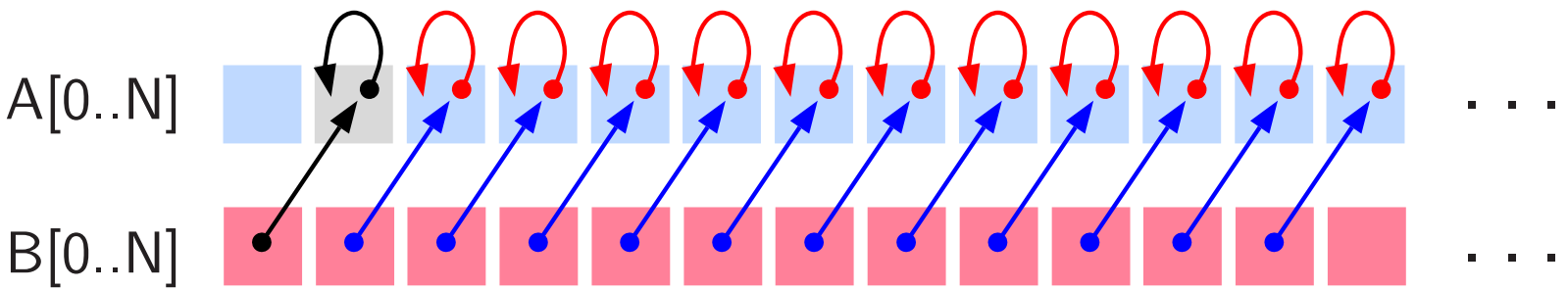
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Iteration # 1

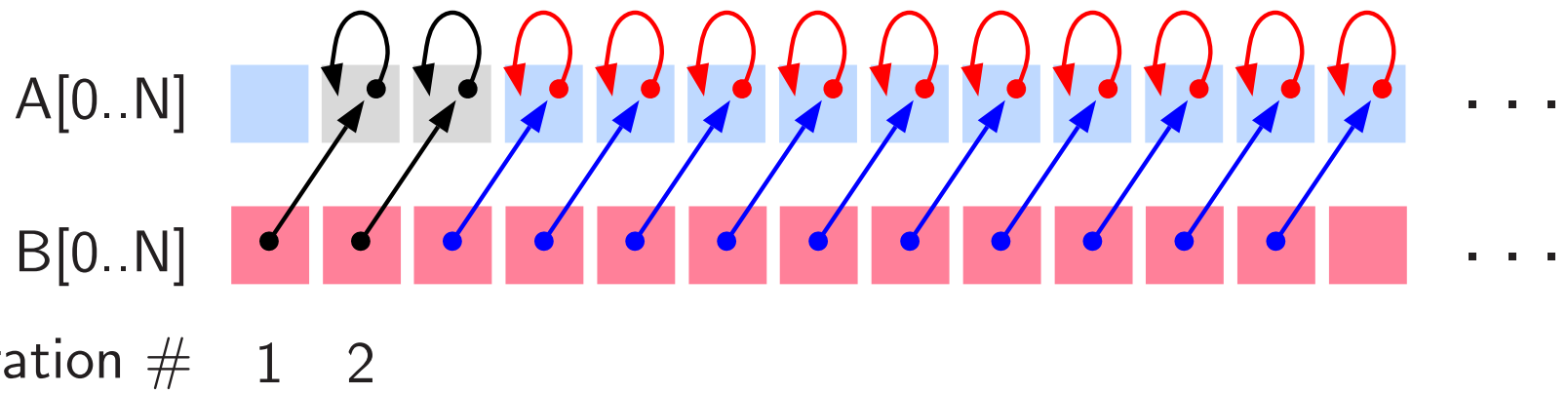
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



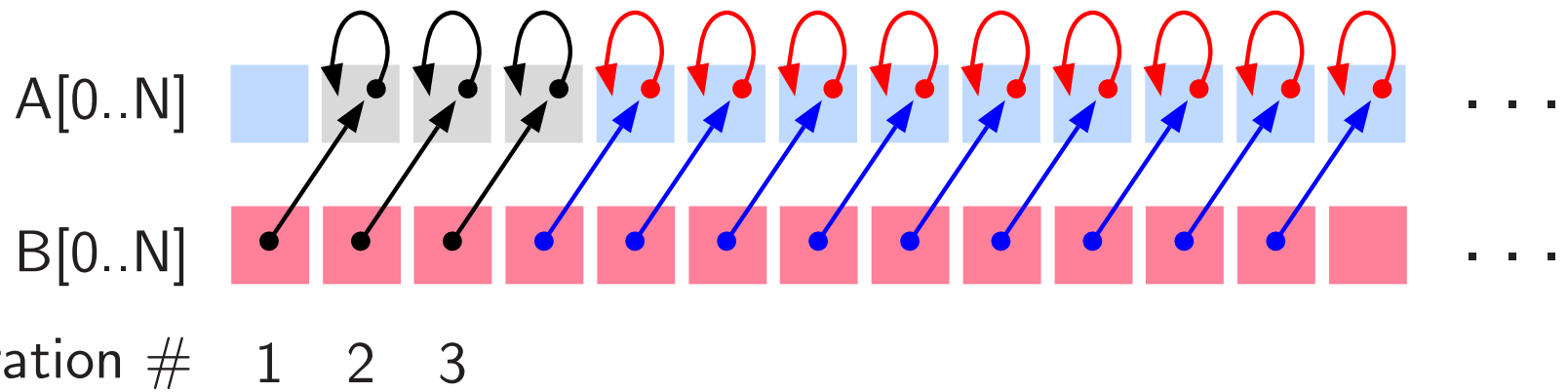
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



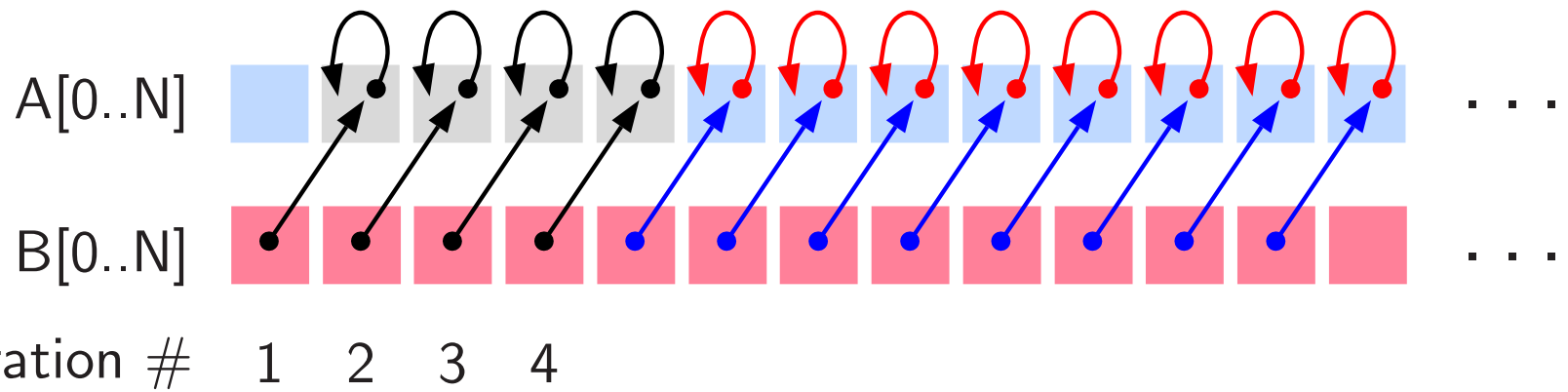
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



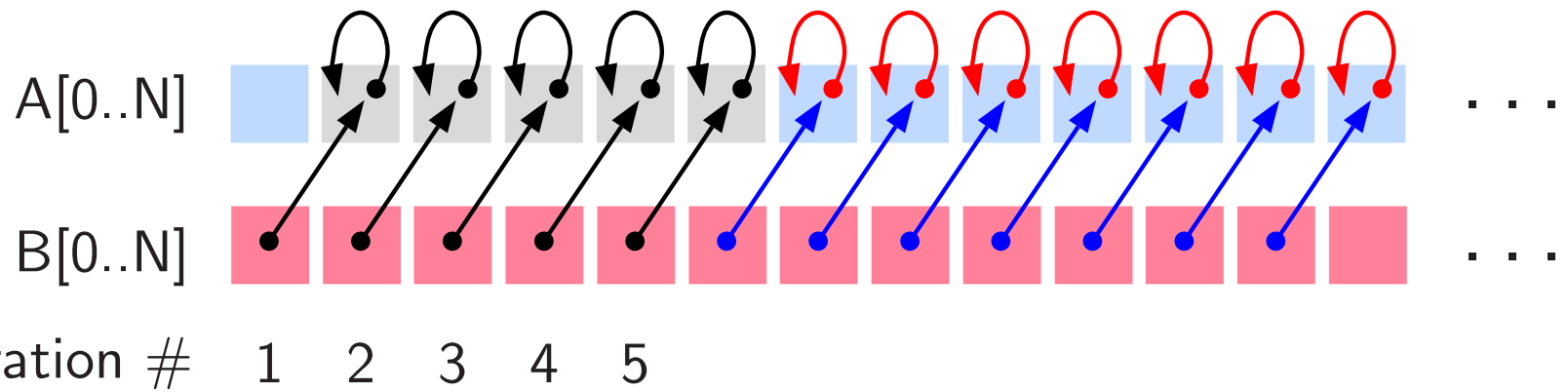
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



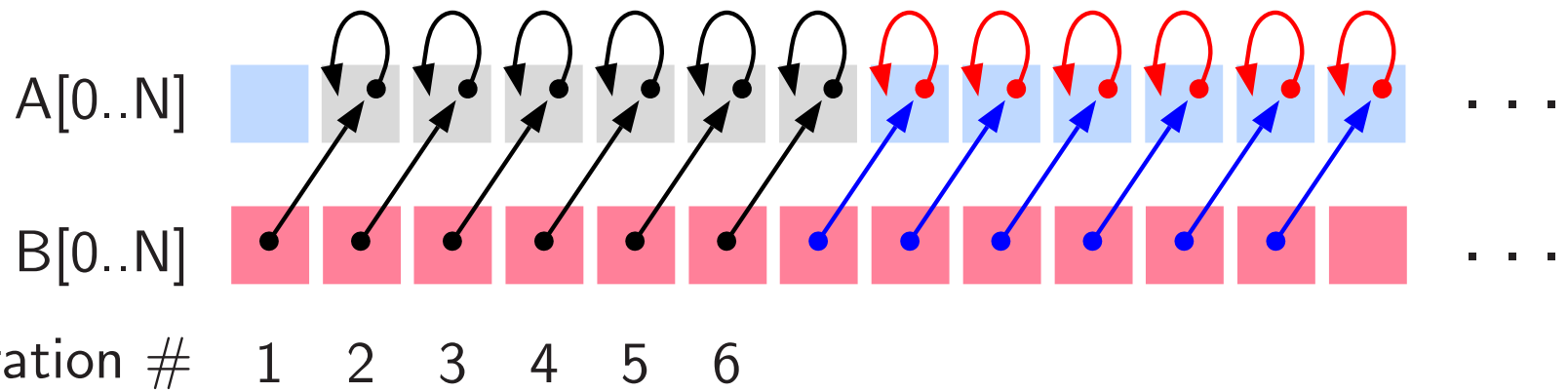
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



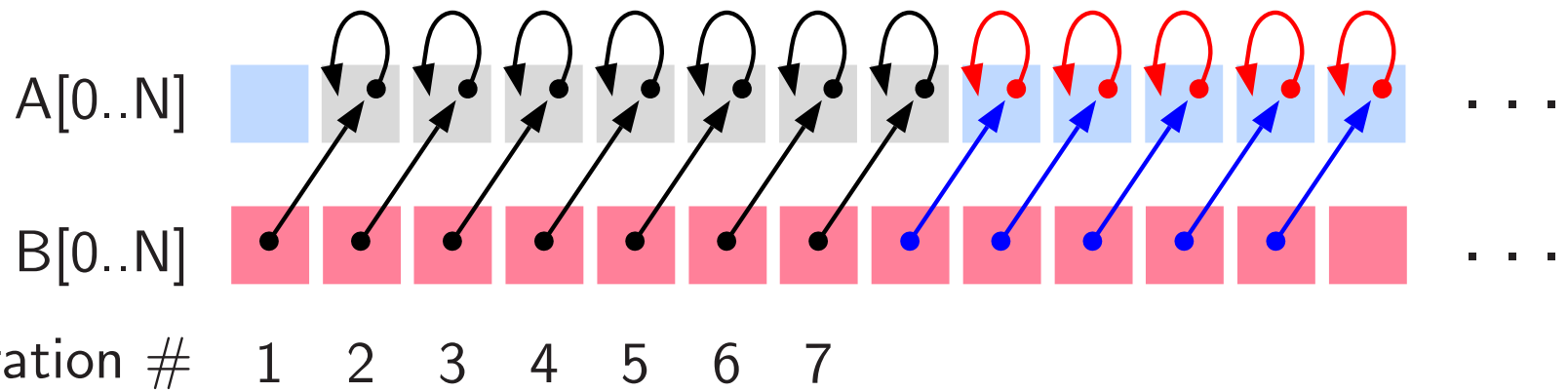
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



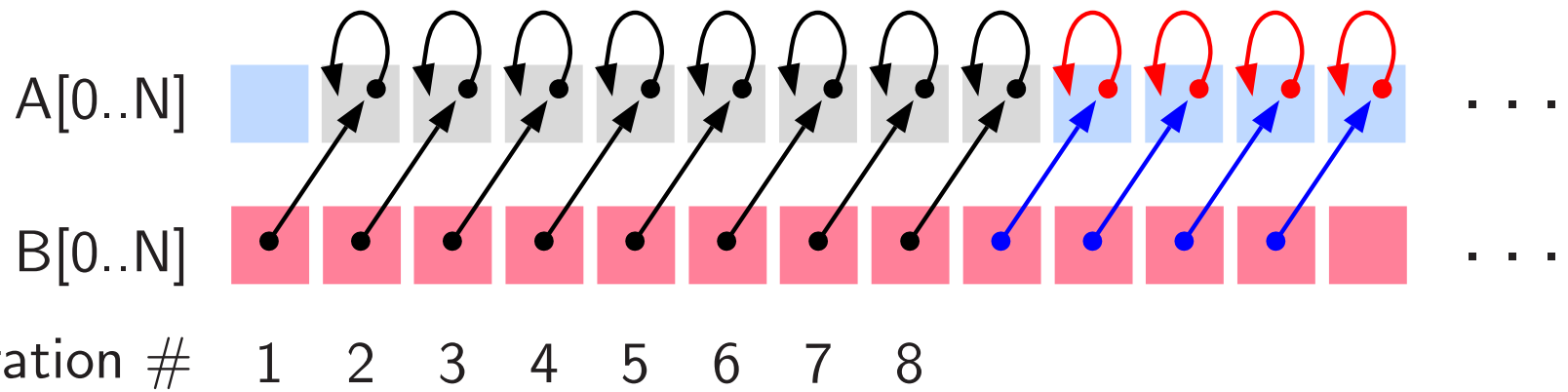
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



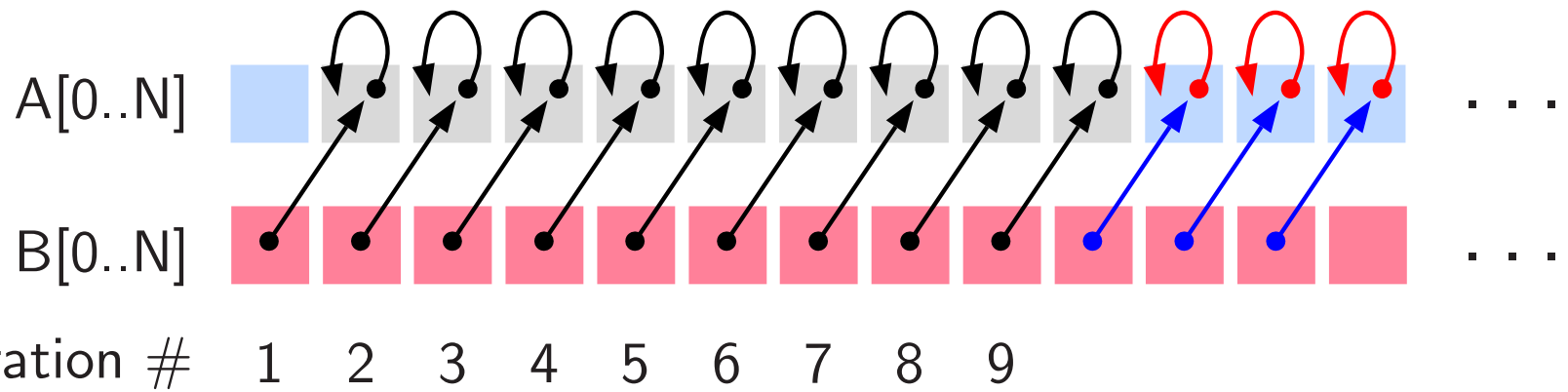
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



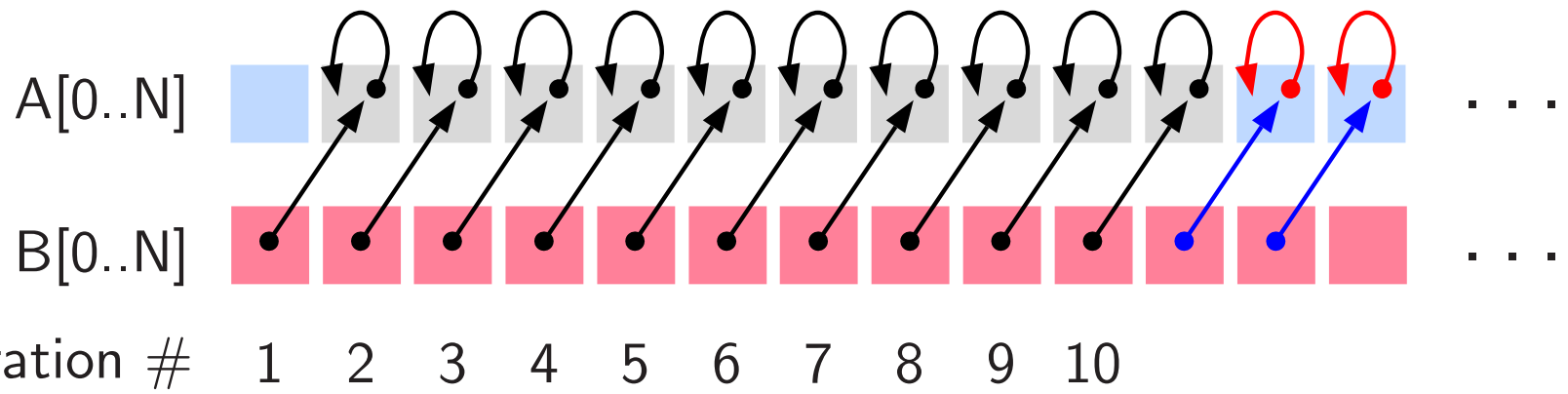
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



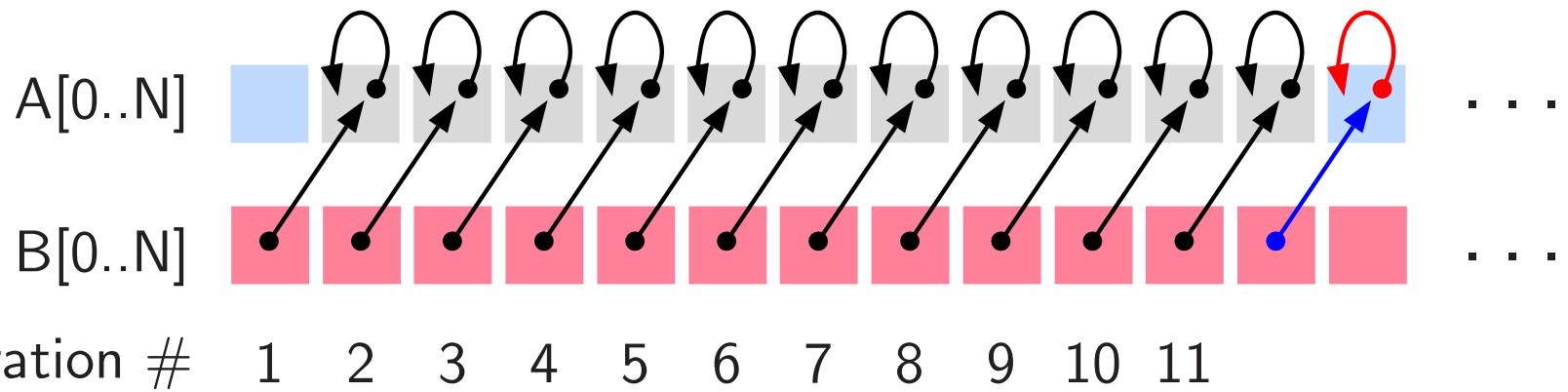
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



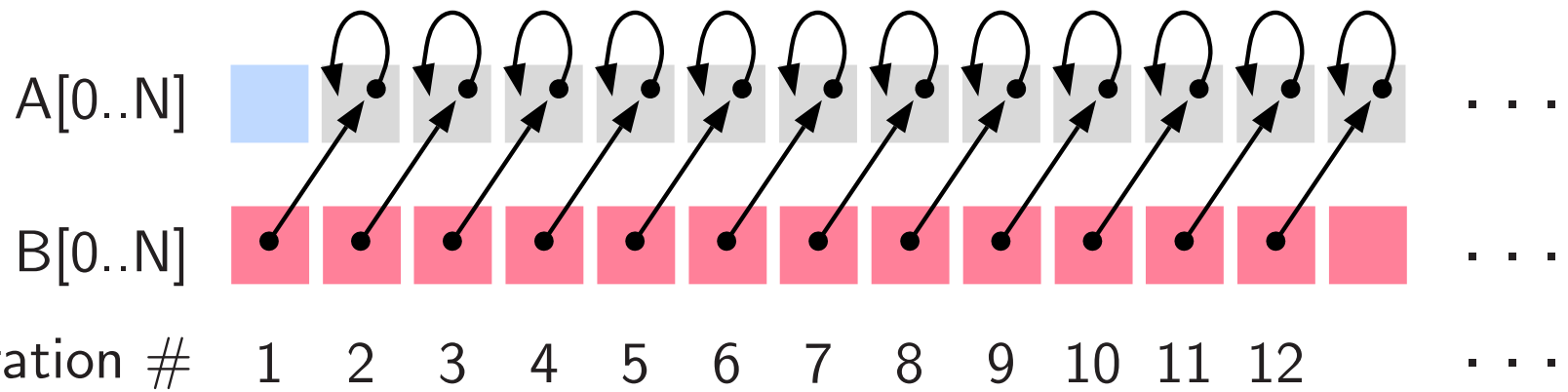
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

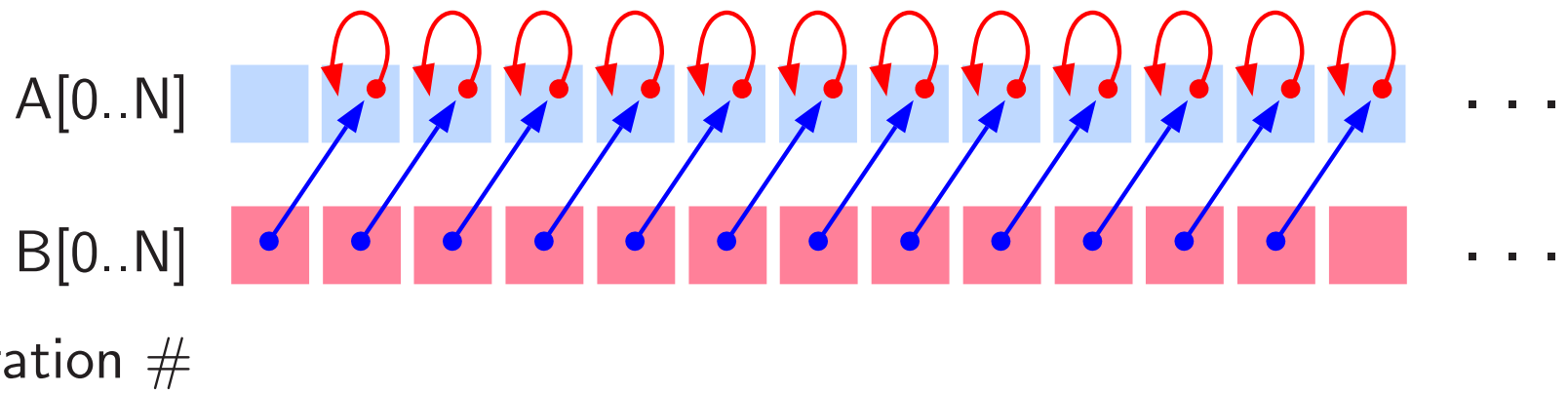
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

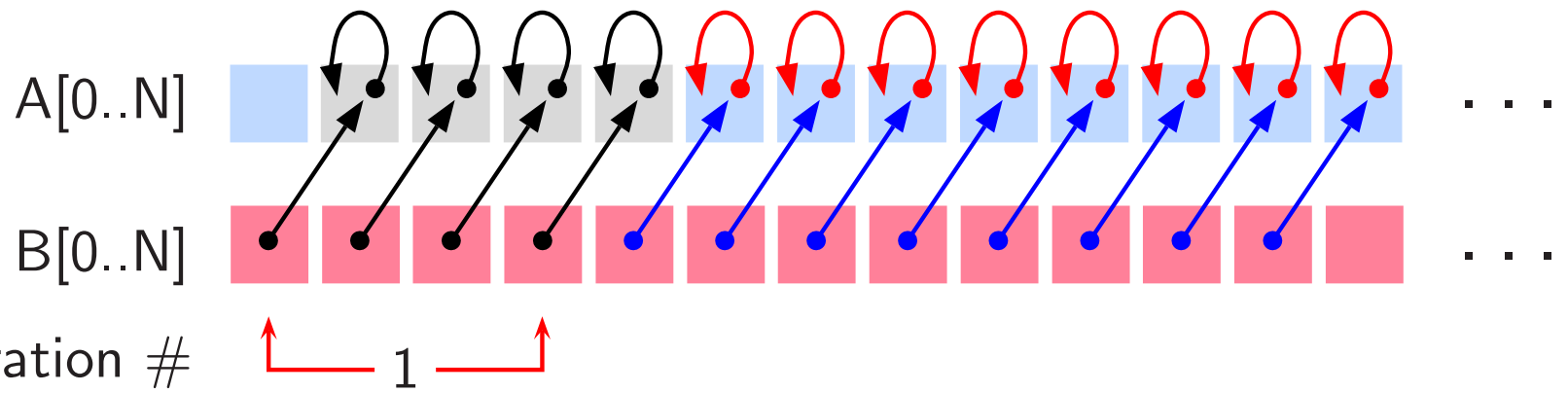
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

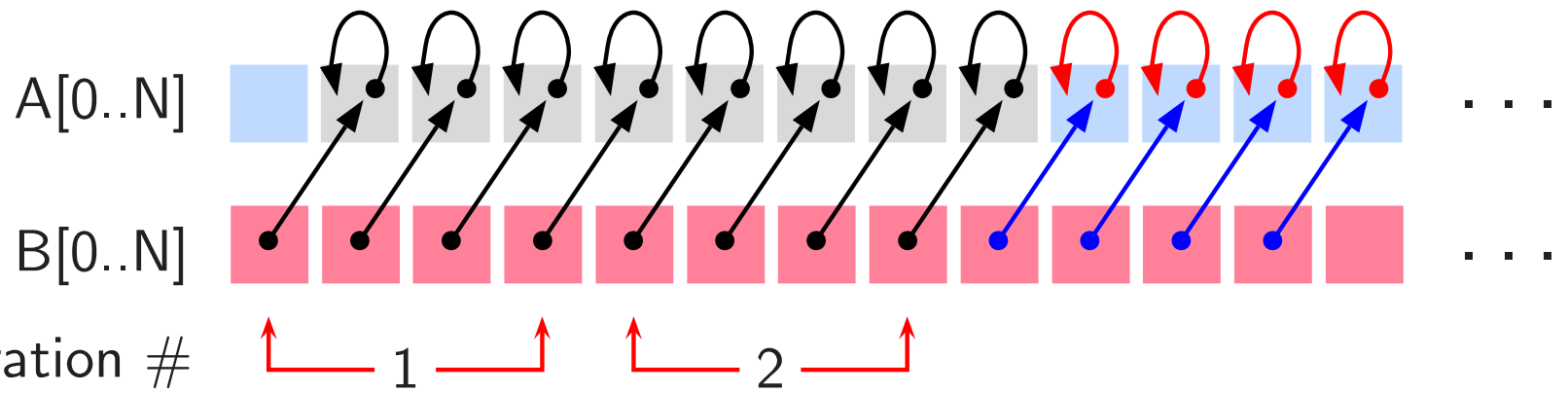
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

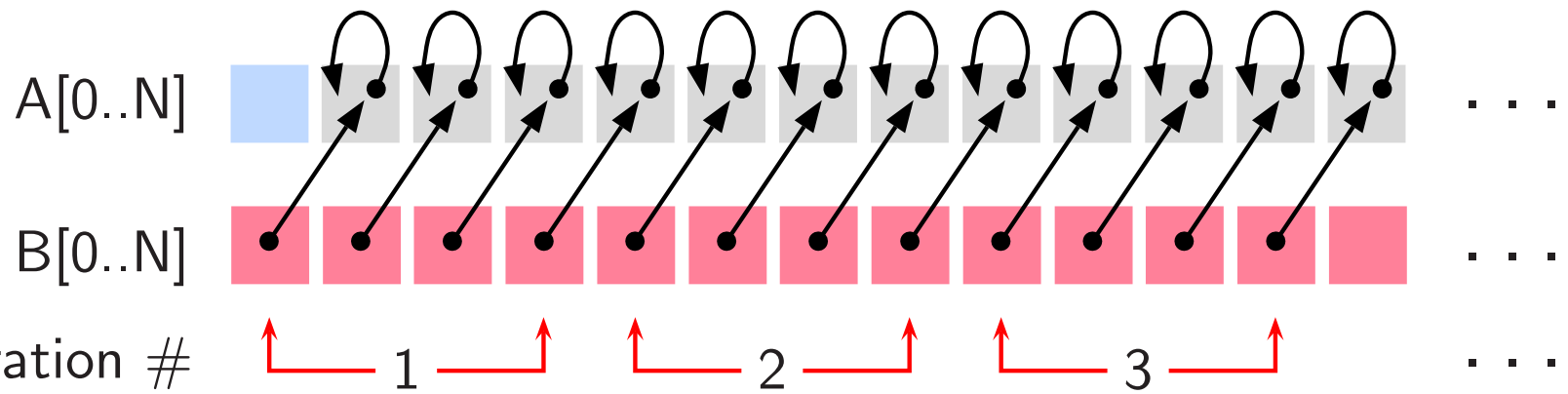
Vectorization Factor

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Vectorized Code

```
int A[N], B[N], i;
for (i=1; i<N; i=i+4)
  A[i:i+3] = A[i:i+3] + B[i-1:i+2];
```



Example 1

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location

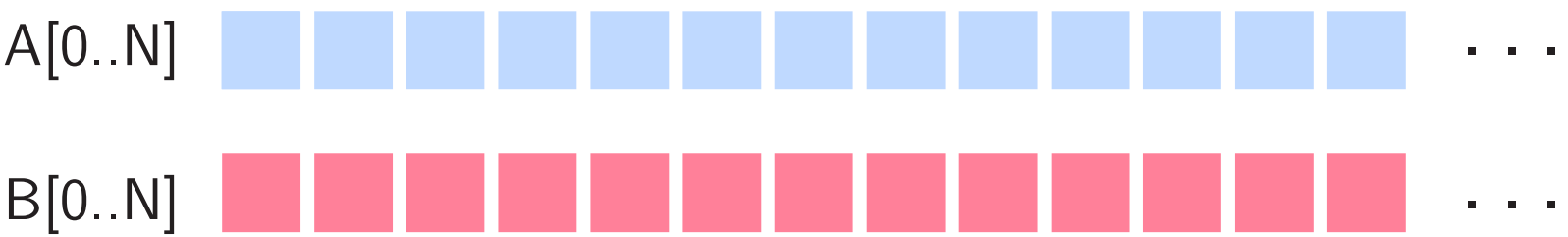
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



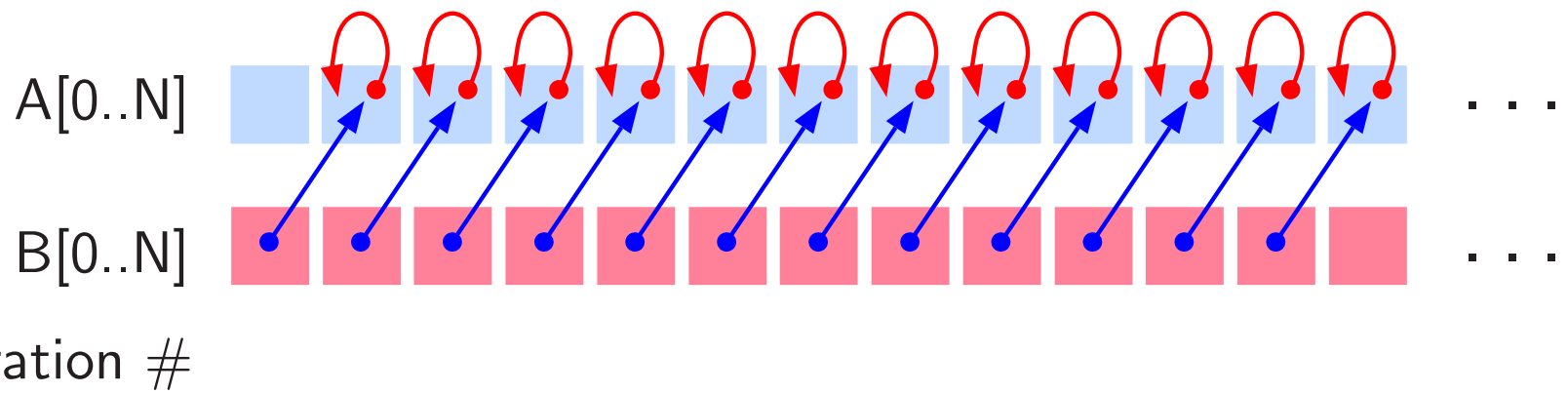
Example 1

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



Example 1

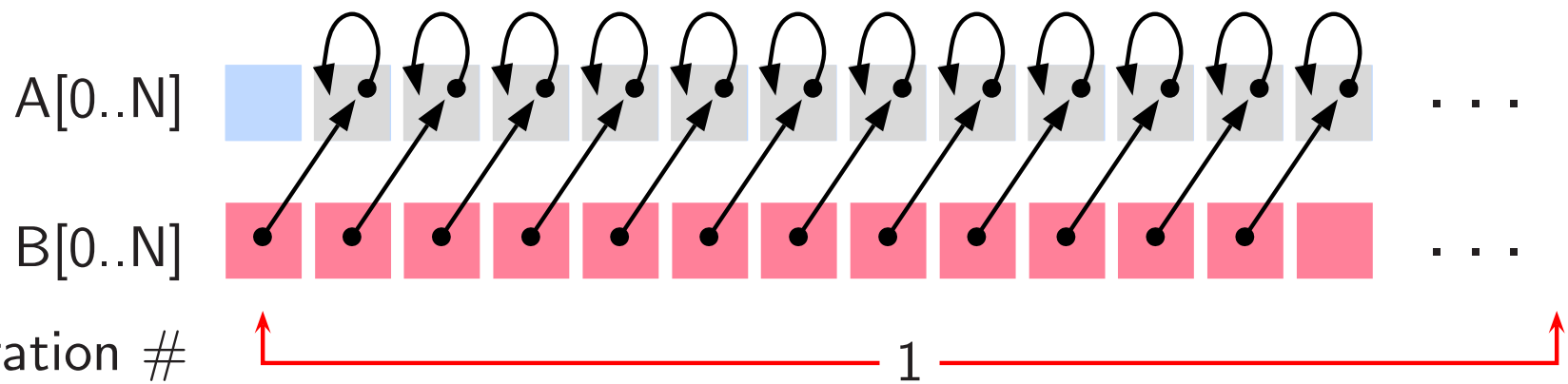
Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

Original Code

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Parallelized Code

```
int A[N], B[N], i;
foreach (i=1; i<N; )
  A[i] = A[i] + B[i-1];
```

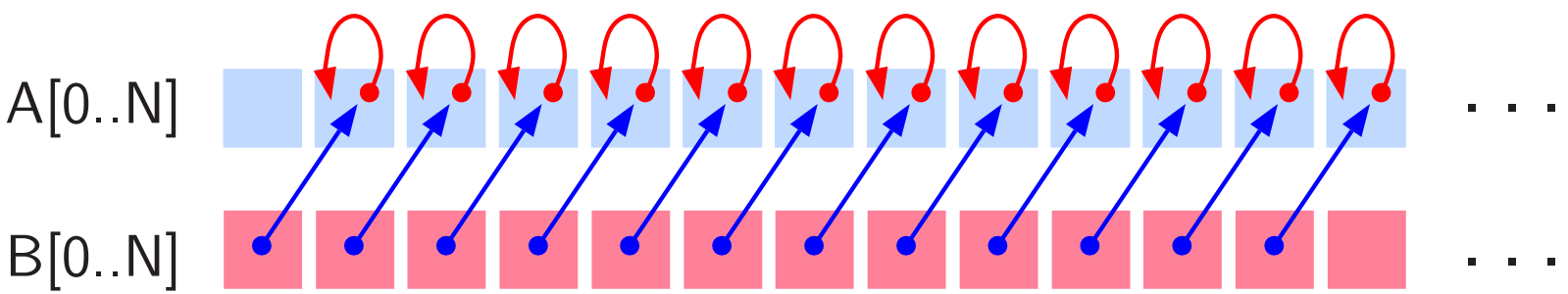


Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

```
int A[N], B[N], i;
for (i=1; i<N; i++)
  A[i] = A[i] + B[i-1];
```

Observe reads and writes into a given location



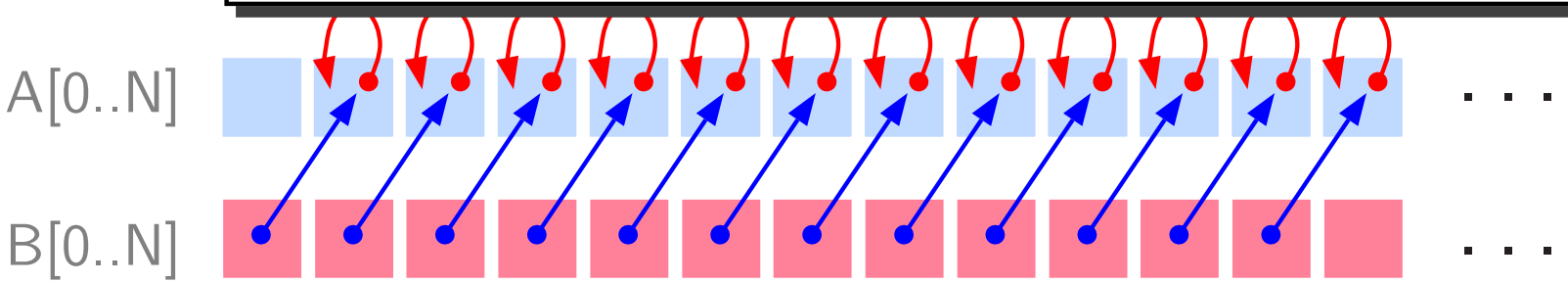
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	
Write	Read	
Write	Write	
Read	Read	



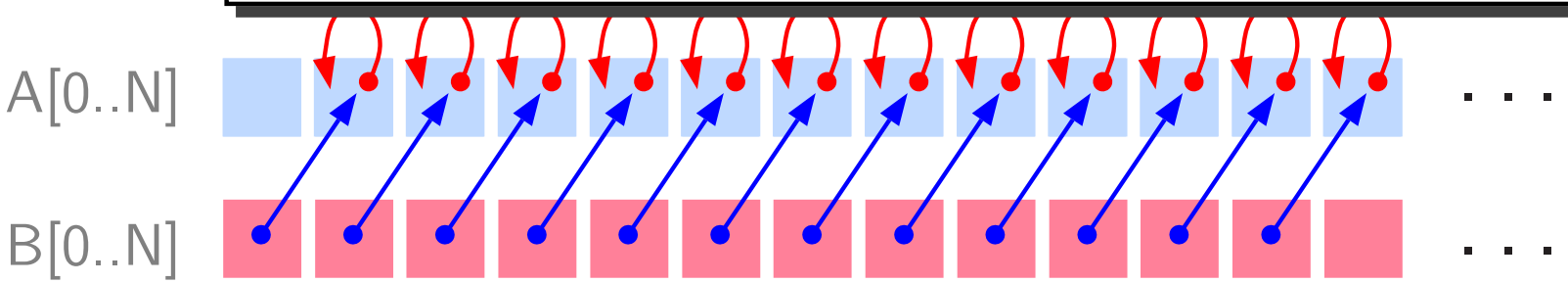
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	
Write	Write	
Read	Read	



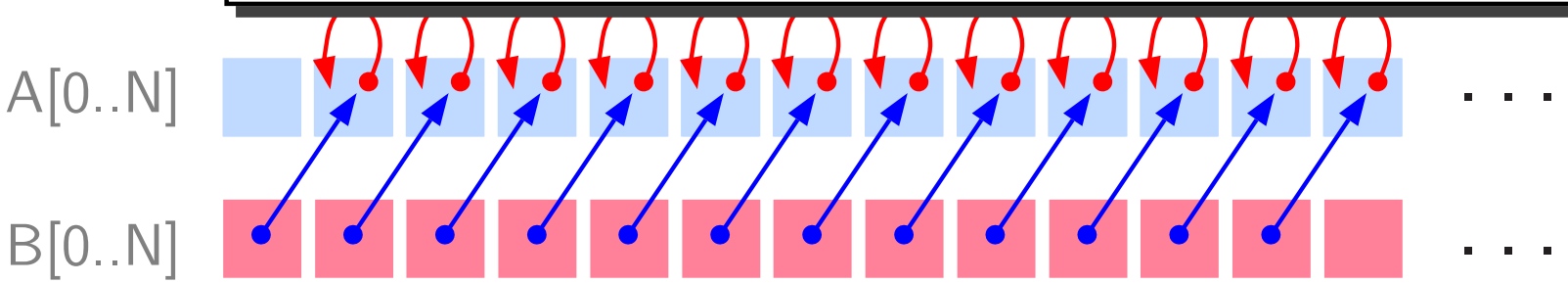
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	No
Write	Write	
Read	Read	



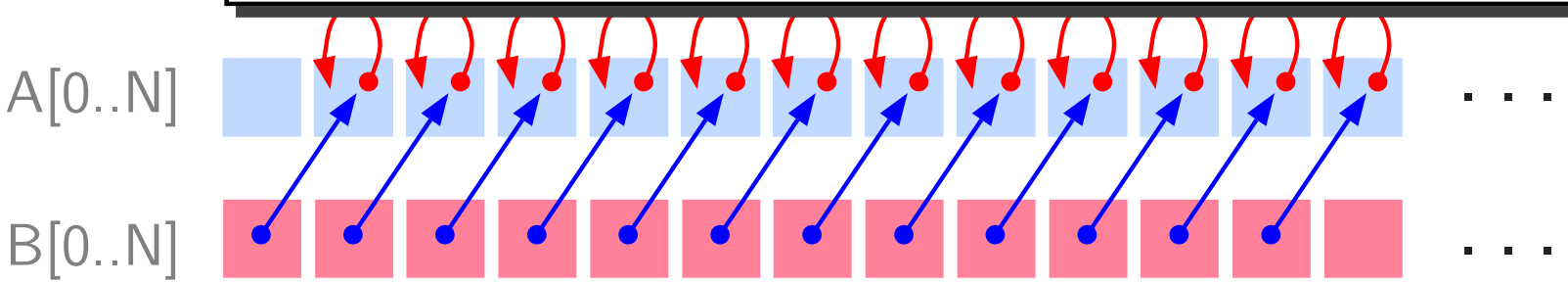
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	No
Write	Write	No
Read	Read	



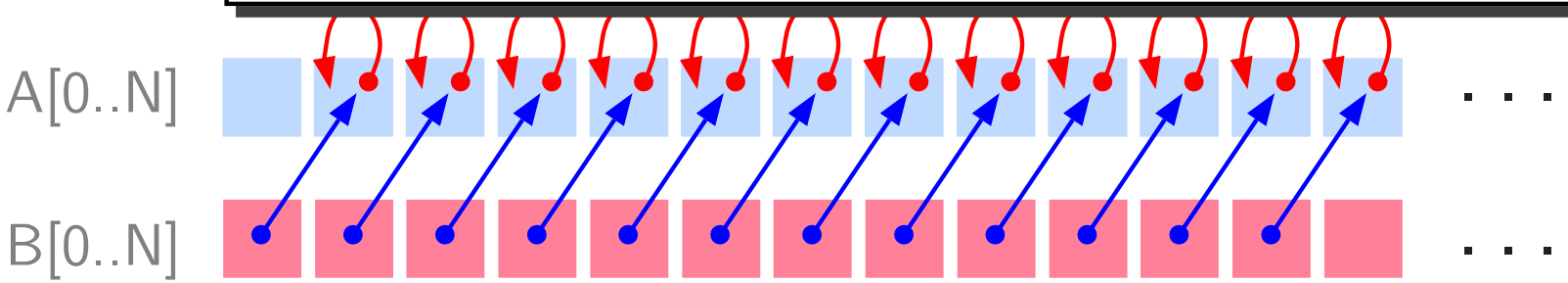
Example 1: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : Yes

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	No
Write	Read	No
Write	Write	No
Read	Read	Does not matter



Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

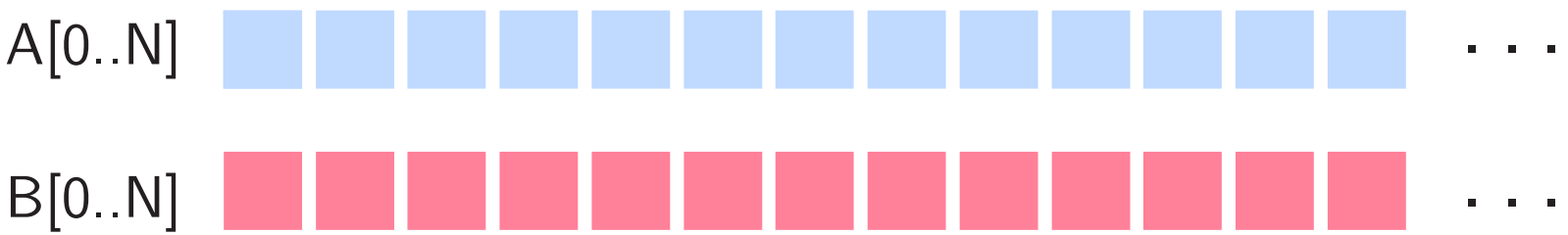

Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



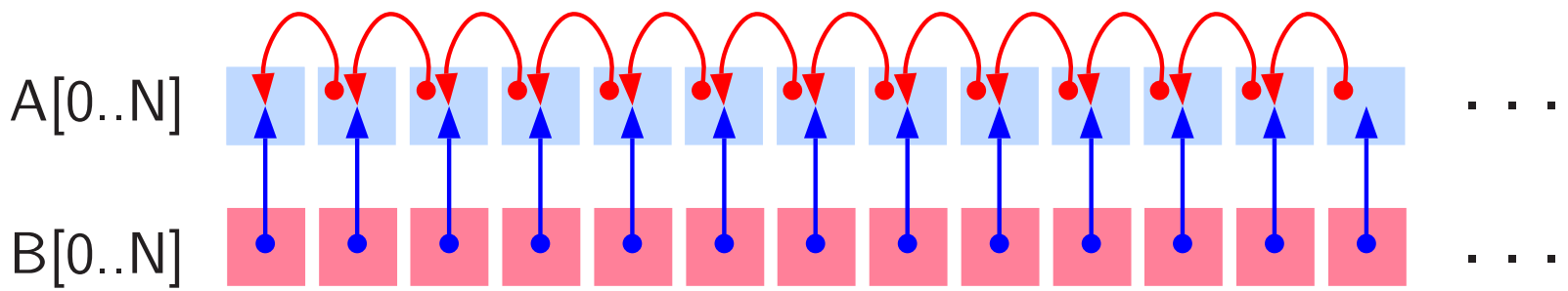
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



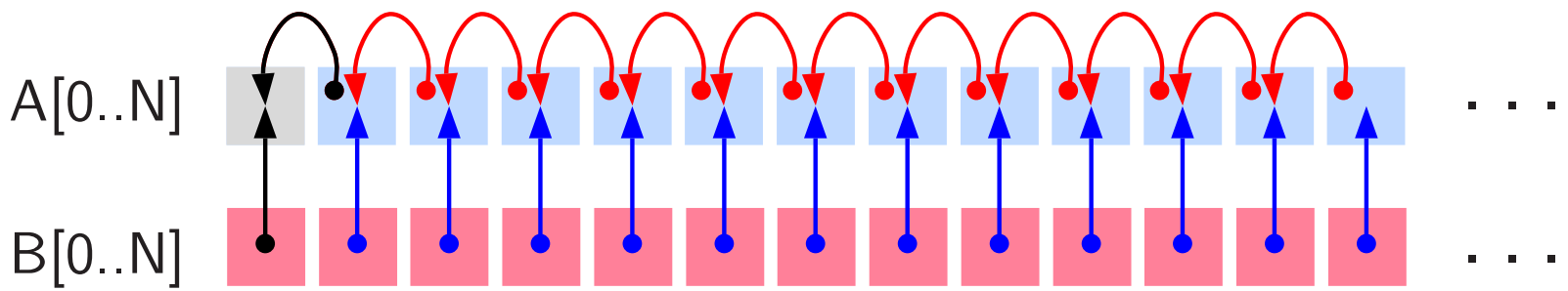
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



Iteration # 1

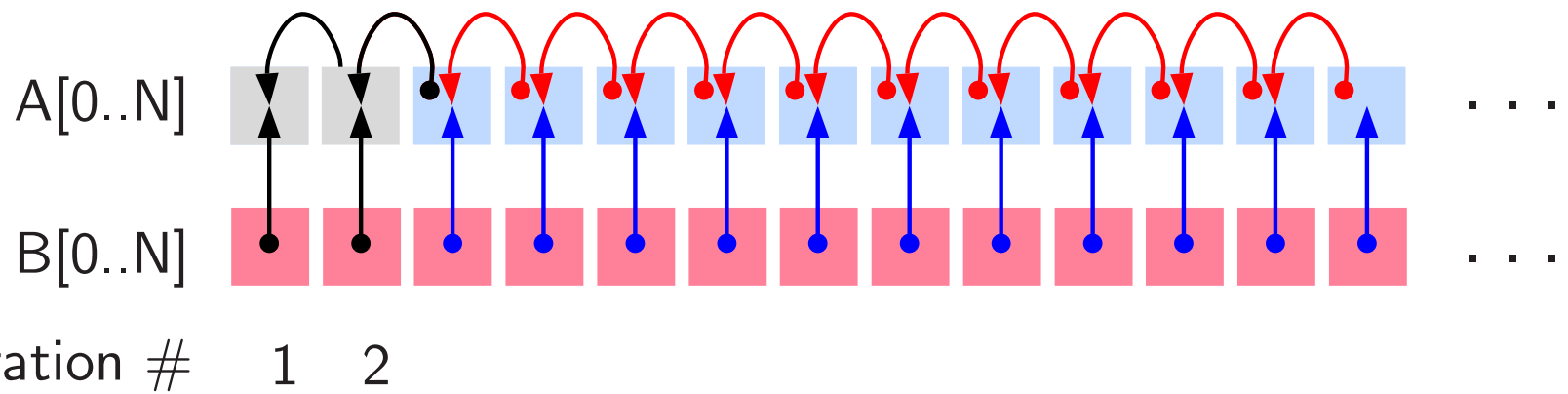
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



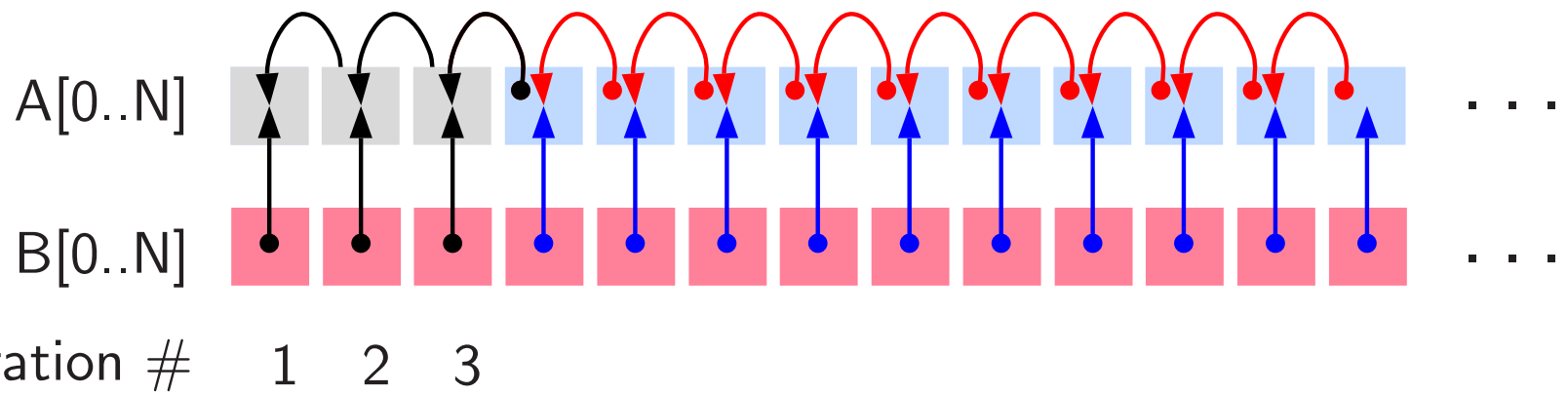
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



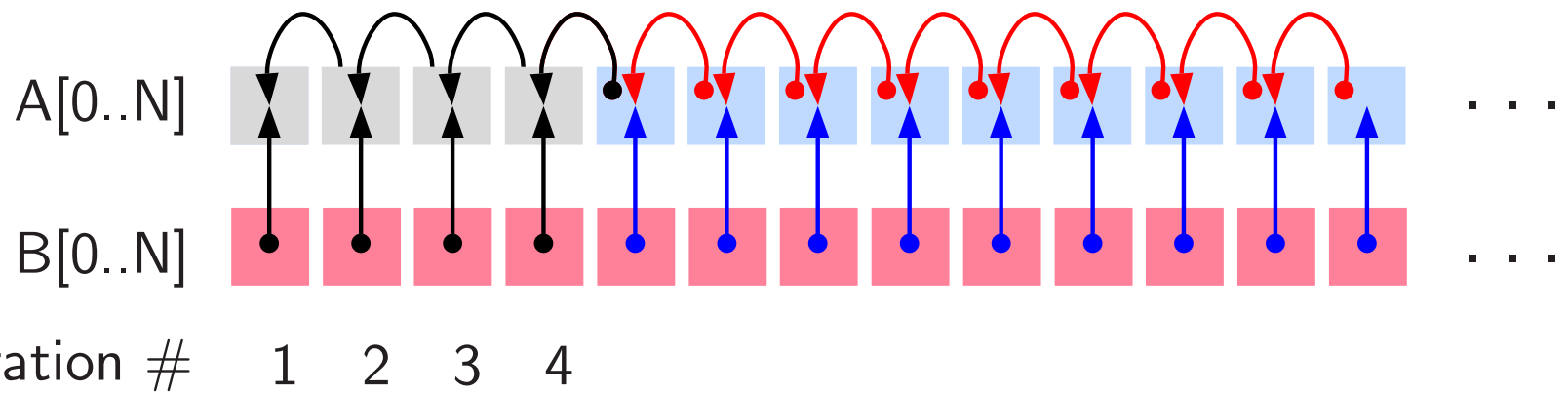
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



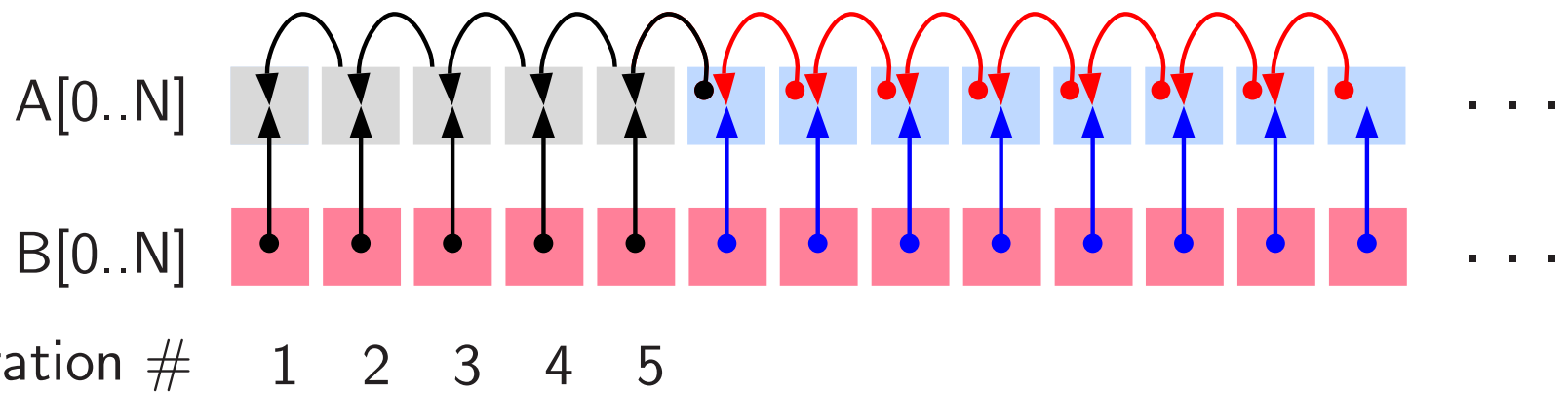
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



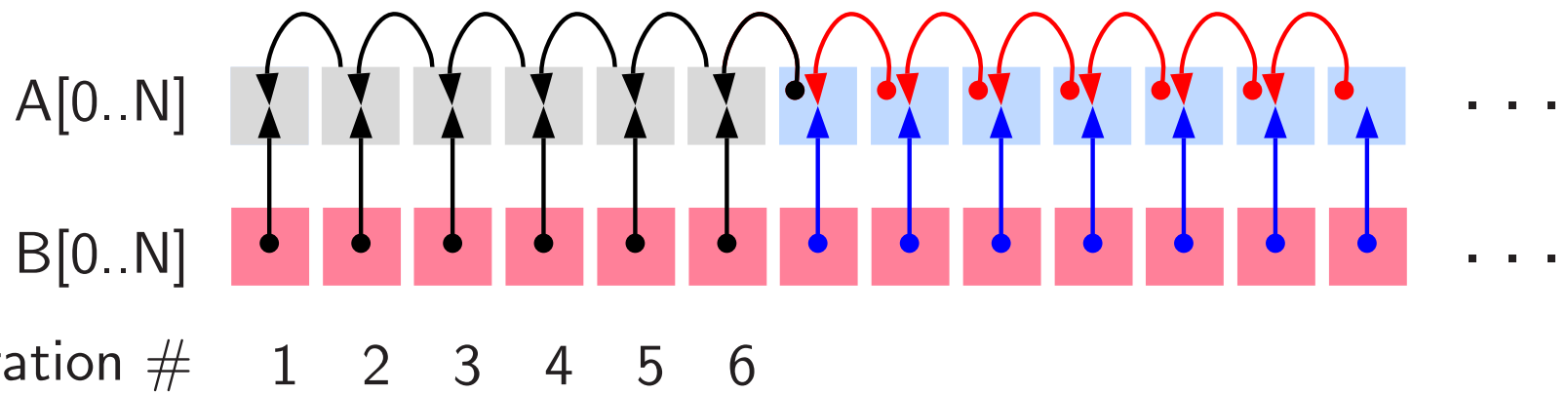
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



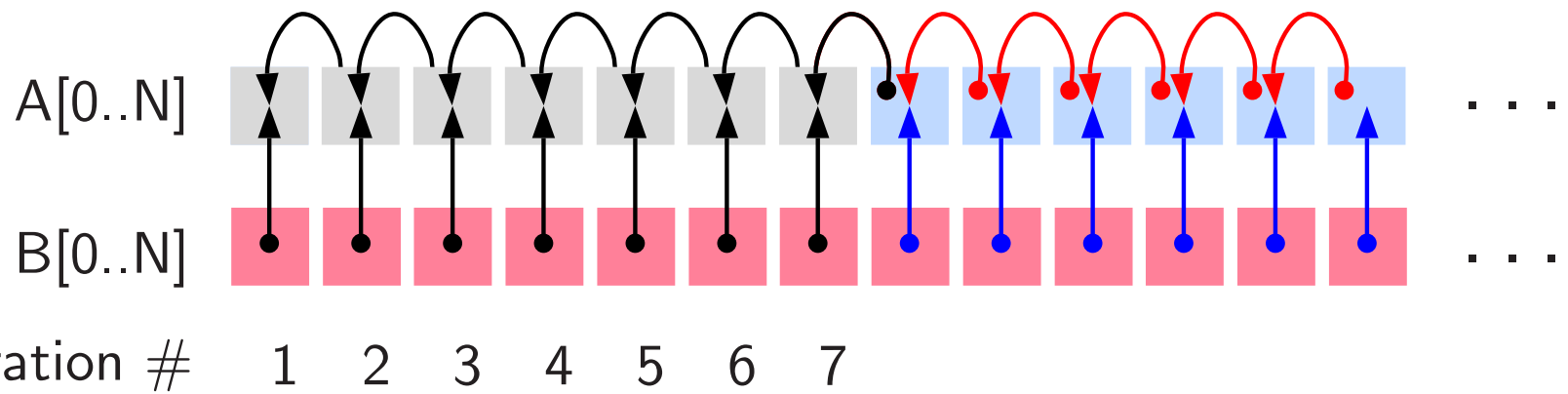
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



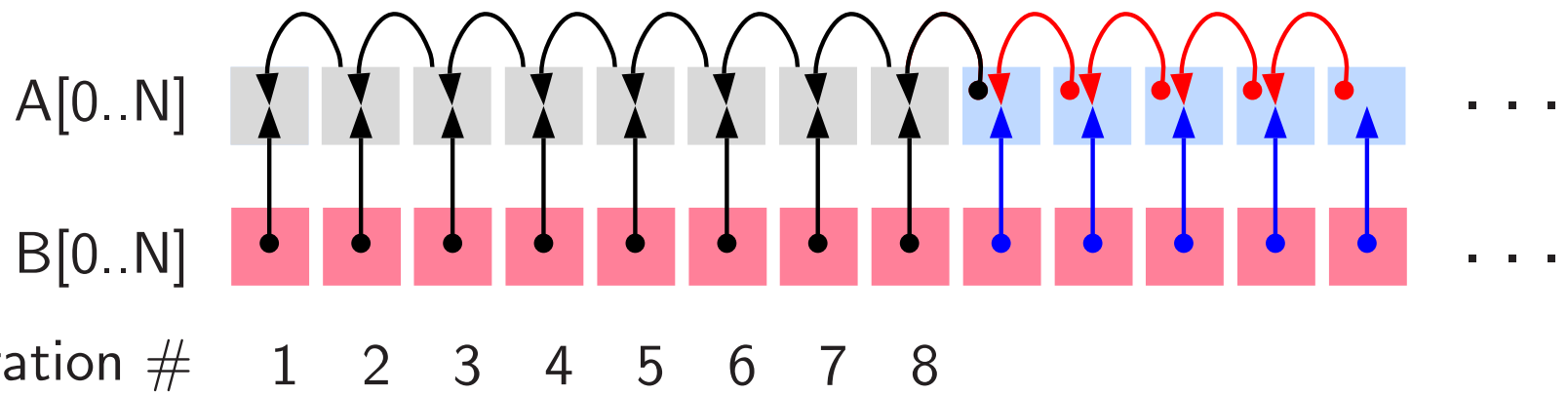
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



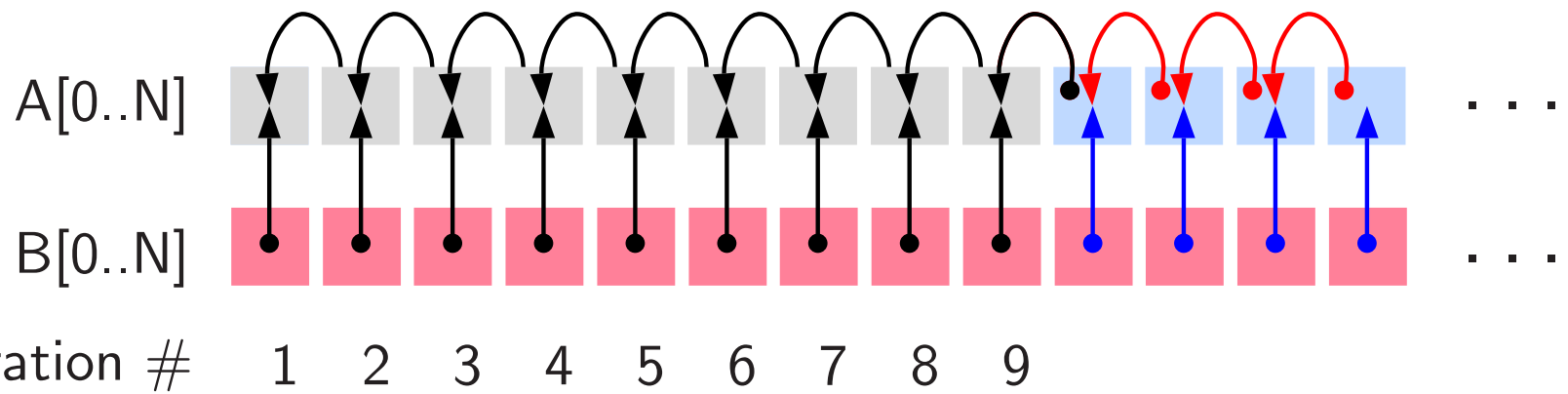
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



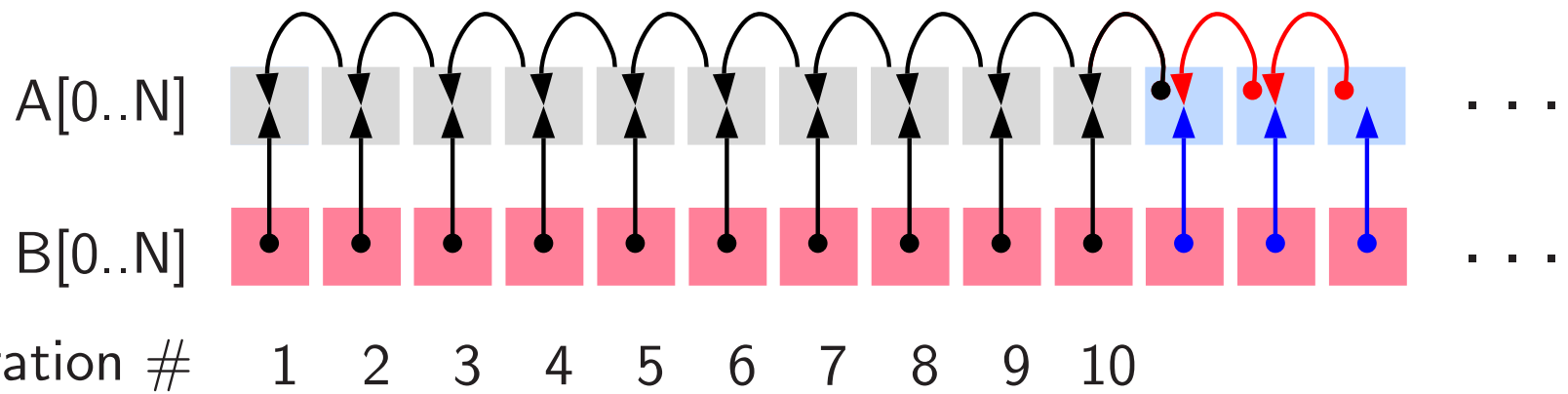
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



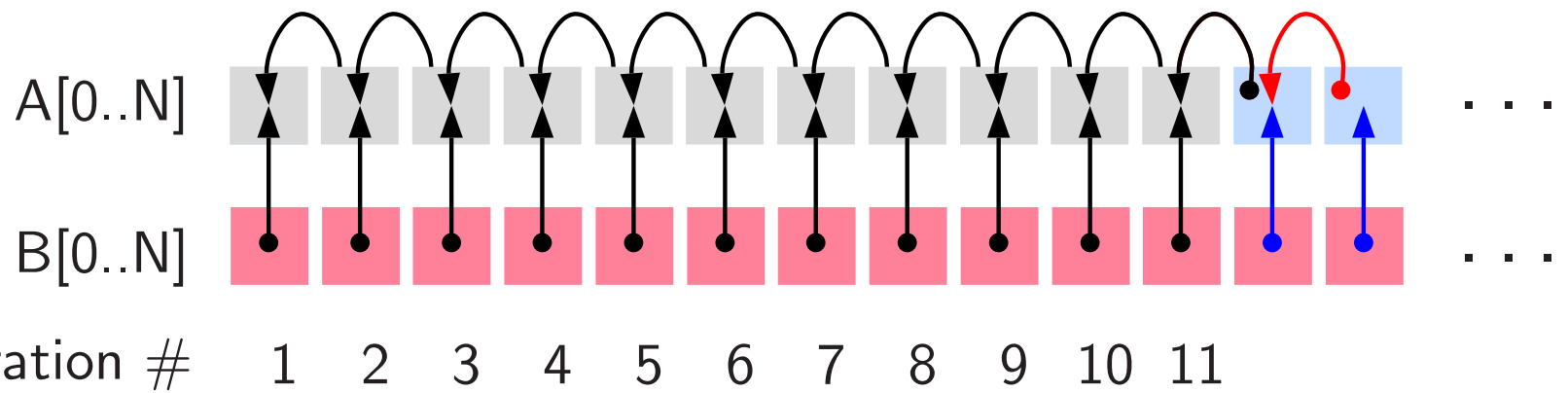
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



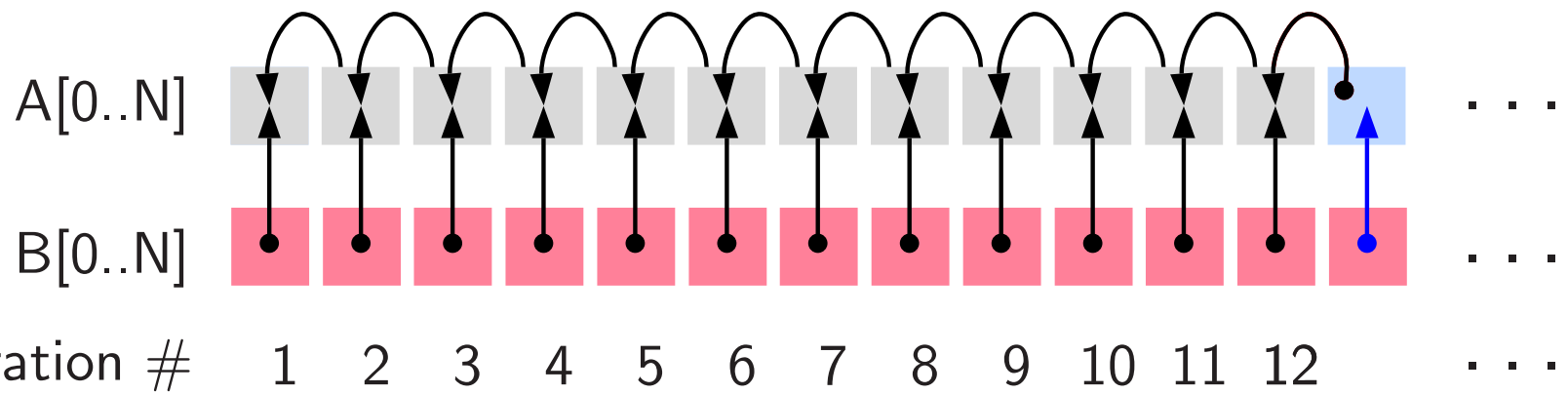
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



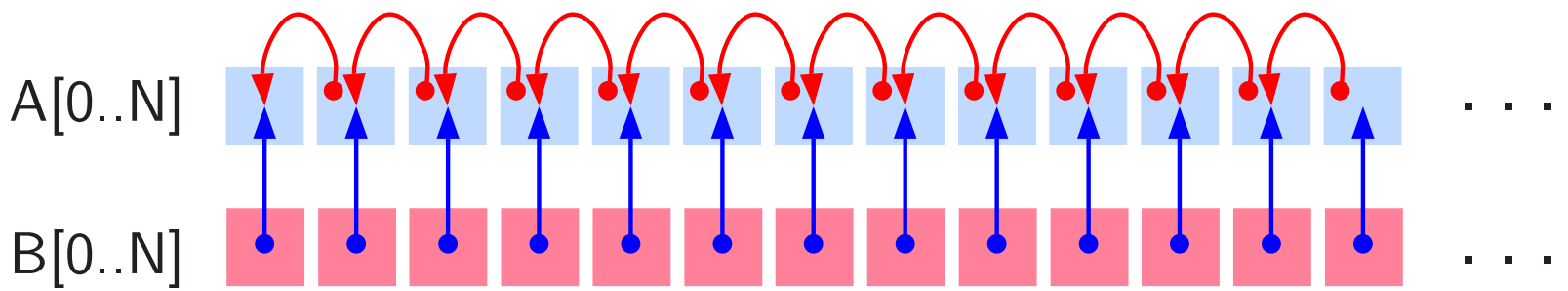
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



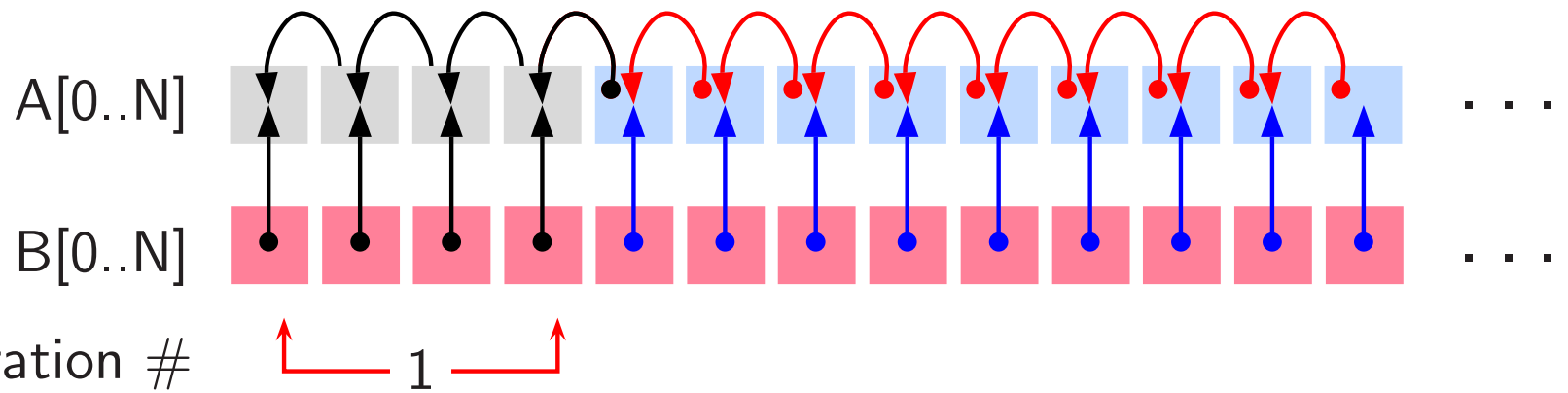
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



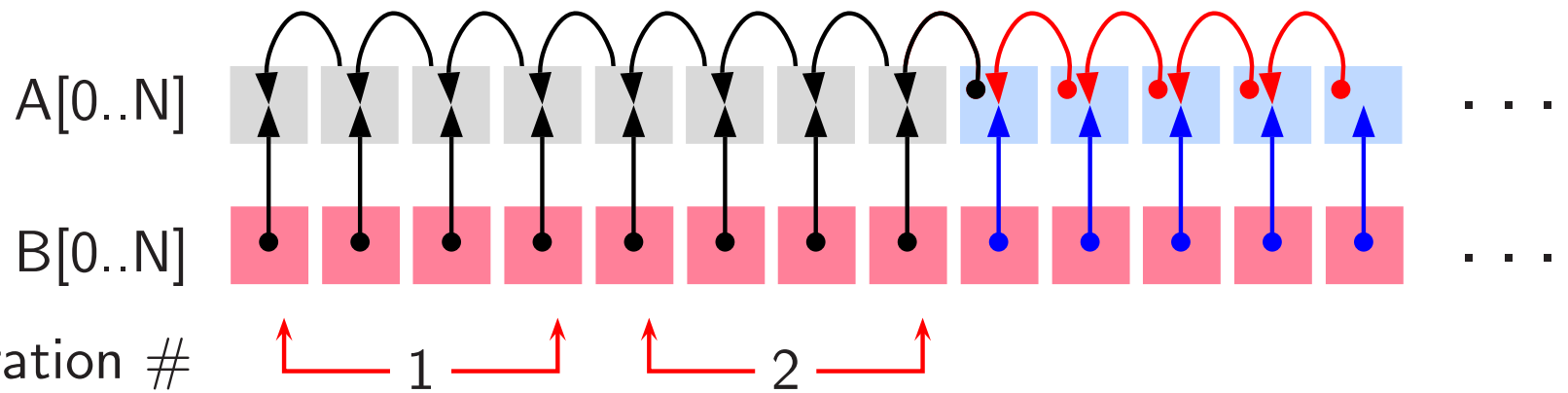
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



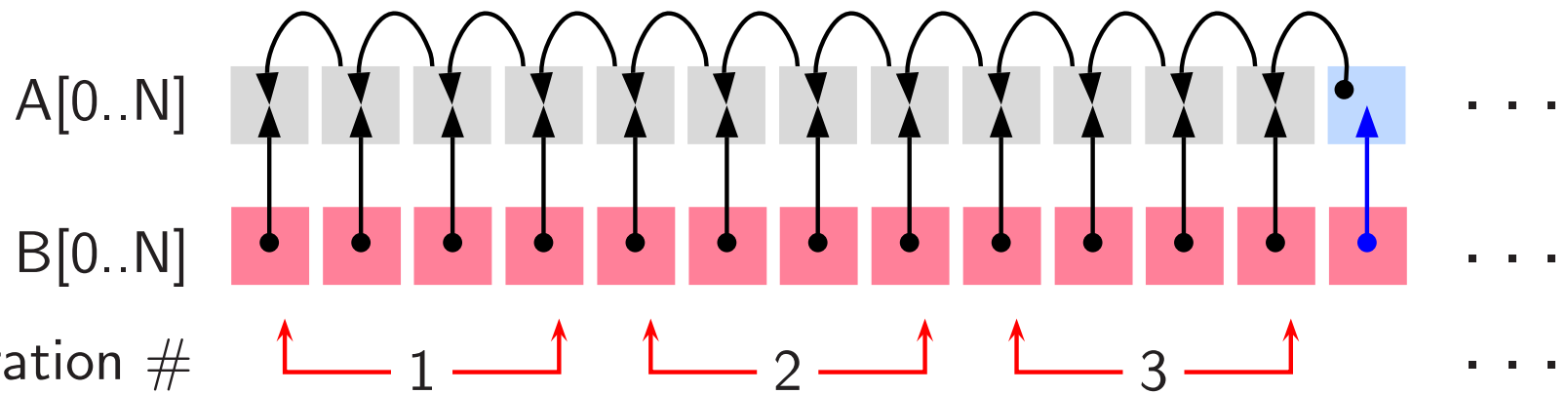
Example 2

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction



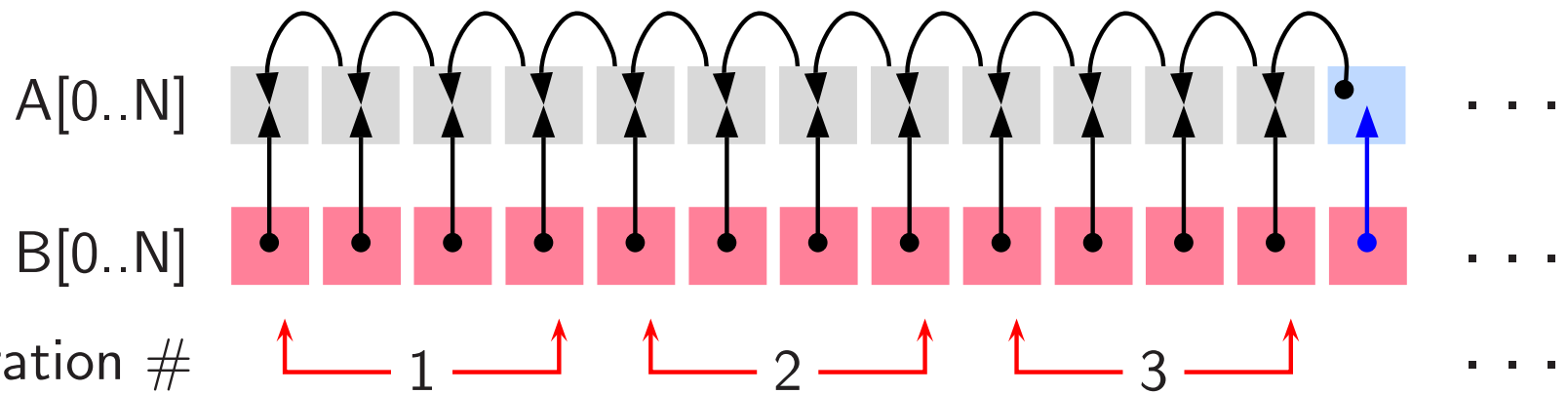
Example 2

Vectorization (SISD $\Rightarrow$ SIMD) : Yes
Parallelization (SISD $\Rightarrow$ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

- Vector instruction is synchronized: All reads before writes in a given instruction
- Read-writes across multiple instructions executing in parallel may not be synchronized



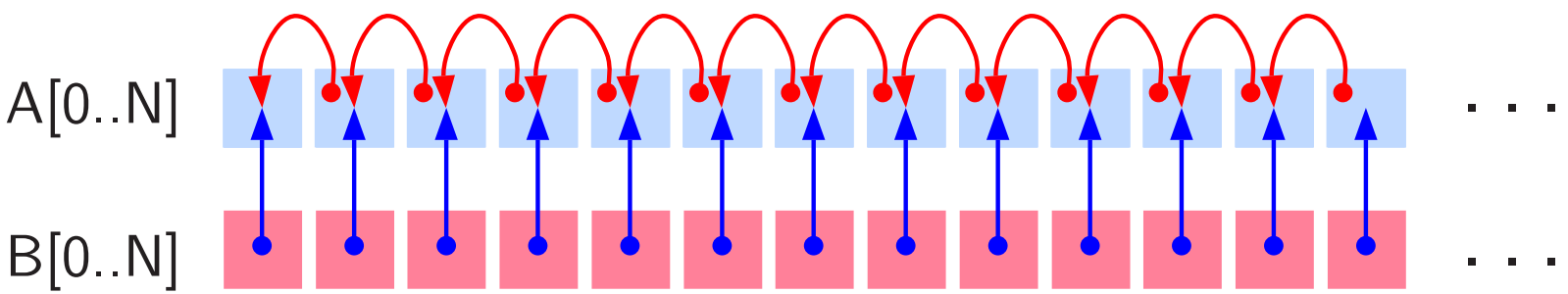
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

Original Code

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i] = A[i+1] + B[i];
```

Observe reads and writes into a given location



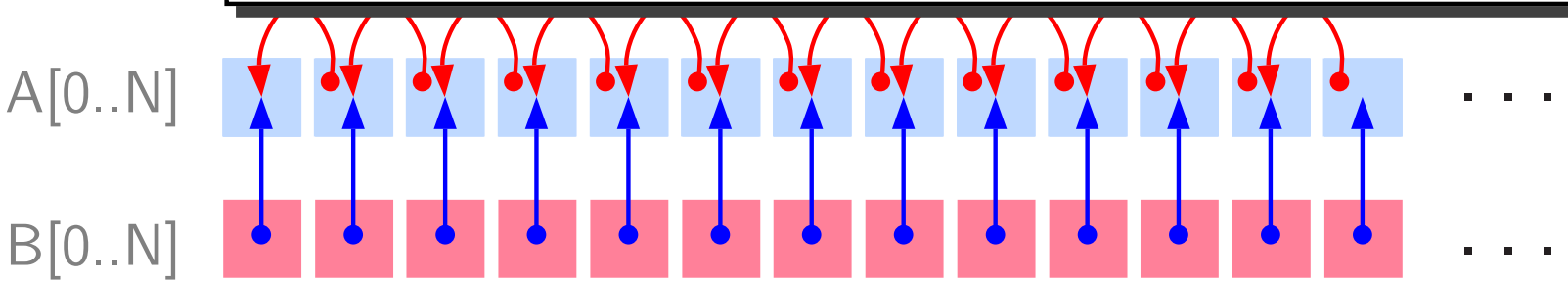
Example 2: The Moral of the Story

Vectorization (SISD ⇒ SIMD) : Yes
Parallelization (SISD ⇒ MIMD) : No

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	
Write	Read	
Write	Write	
Read	Read	



Example 2: The Moral of the Story

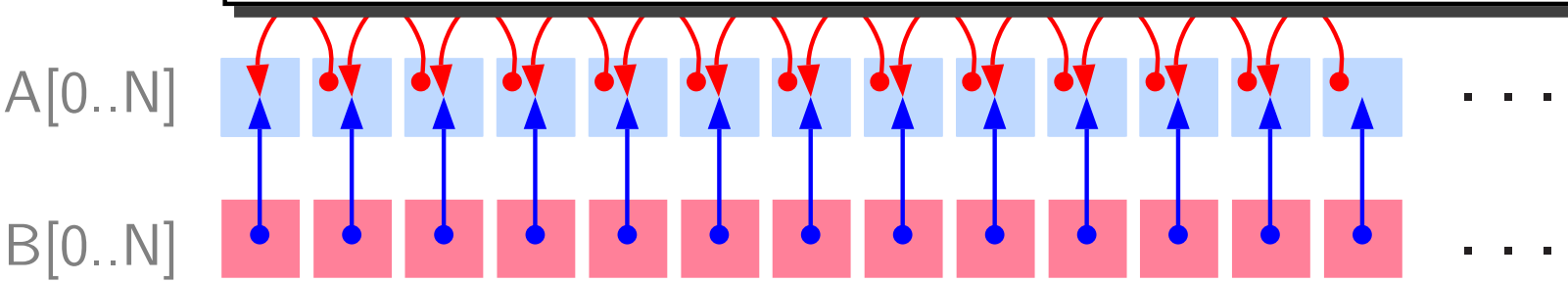
Vectorization (SISD ⇒ SIMD) : Yes

Parallelization (SISD ⇒ MIMD) : **No**

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	
Write	Write	
Read	Read	



Example 2: The Moral of the Story

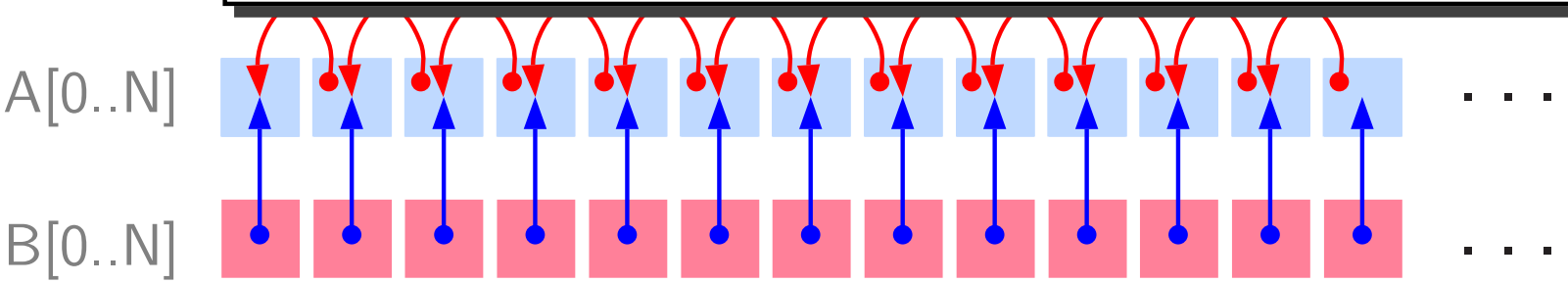
Vectorization (SISD ⇒ SIMD) : Yes

Parallelization (SISD ⇒ MIMD) : **No**

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	
Read	Read	



Example 2: The Moral of the Story

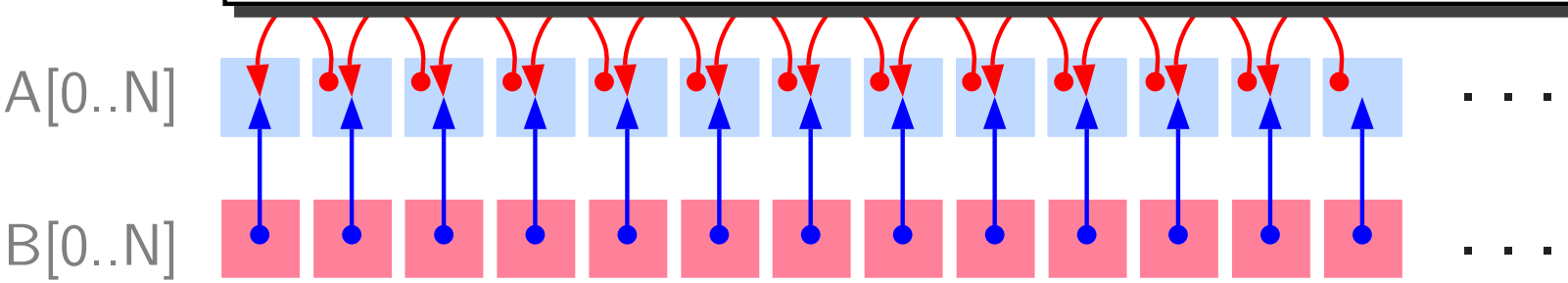
Vectorization (SISD ⇒ SIMD) : Yes

Parallelization (SISD ⇒ MIMD) : **No**

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	No
Read	Read	



Example 2: The Moral of the Story

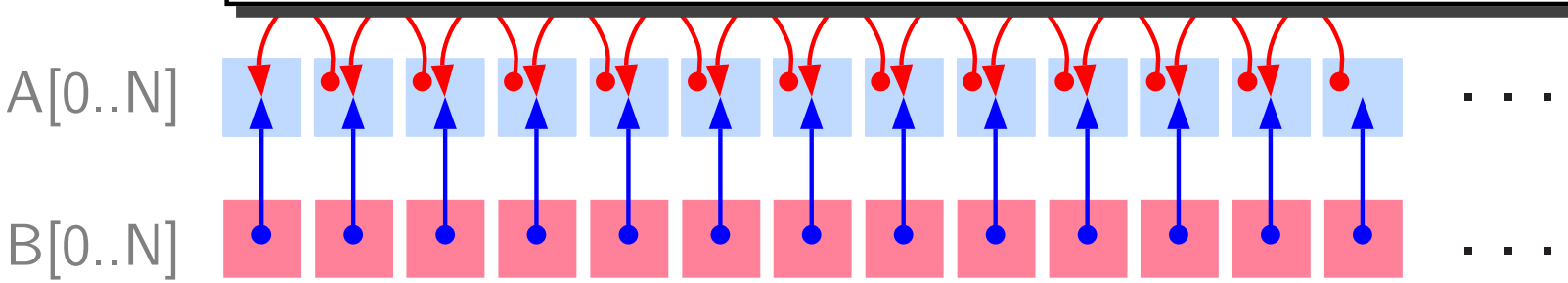
Vectorization (SISD ⇒ SIMD) : Yes

Parallelization (SISD ⇒ MIMD) : **No**

When the same location is accessed across different iterations, the order of reads and writes must be preserved

```
int A[
for (i
  A[i]
```

Nature of accesses in our example		
Iteration i	Iteration $i + k$	Observation
Read	Write	Yes
Write	Read	No
Write	Write	No
Read	Read	Does not matter



Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

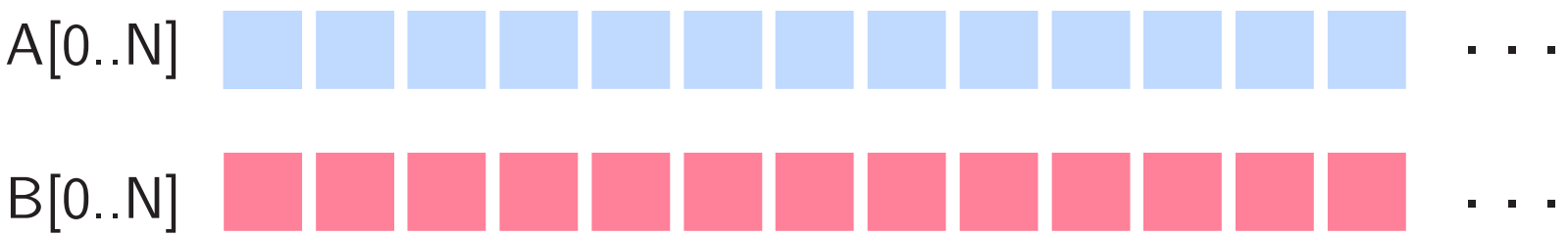
Observe reads and writes into a given location

Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

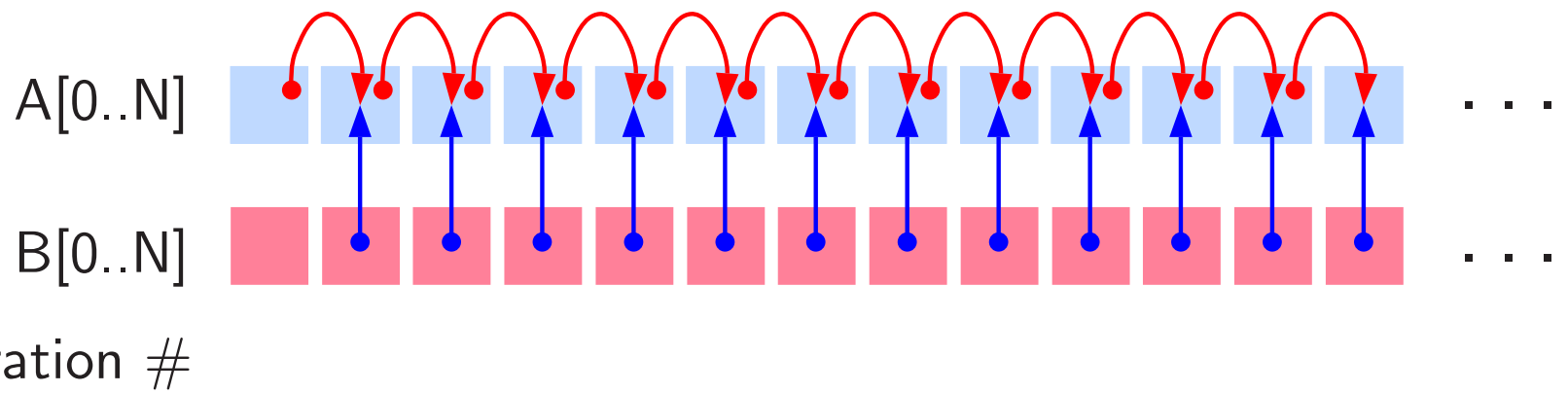


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

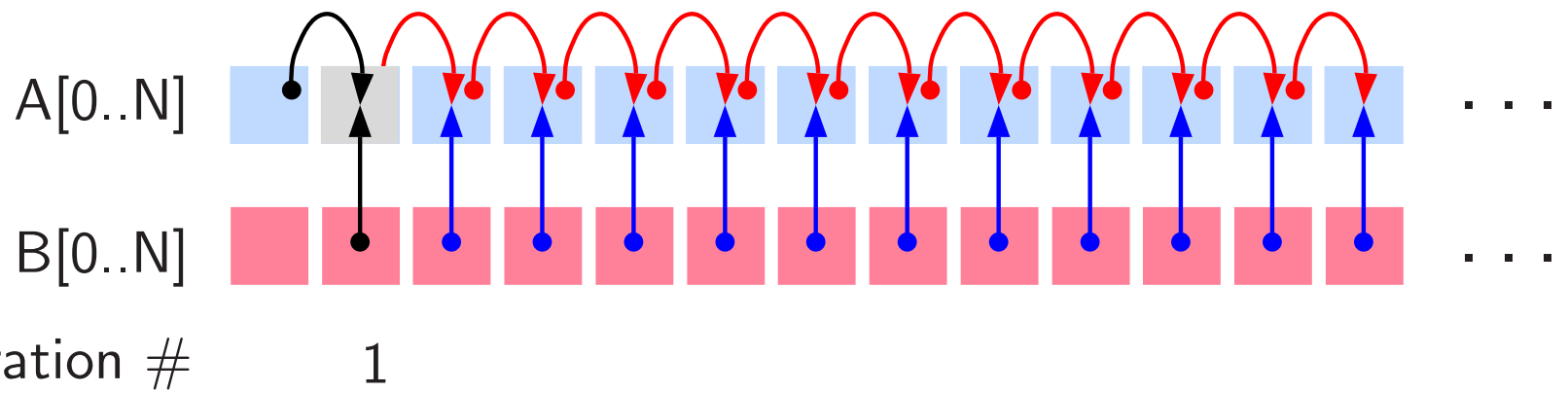


Example 3

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

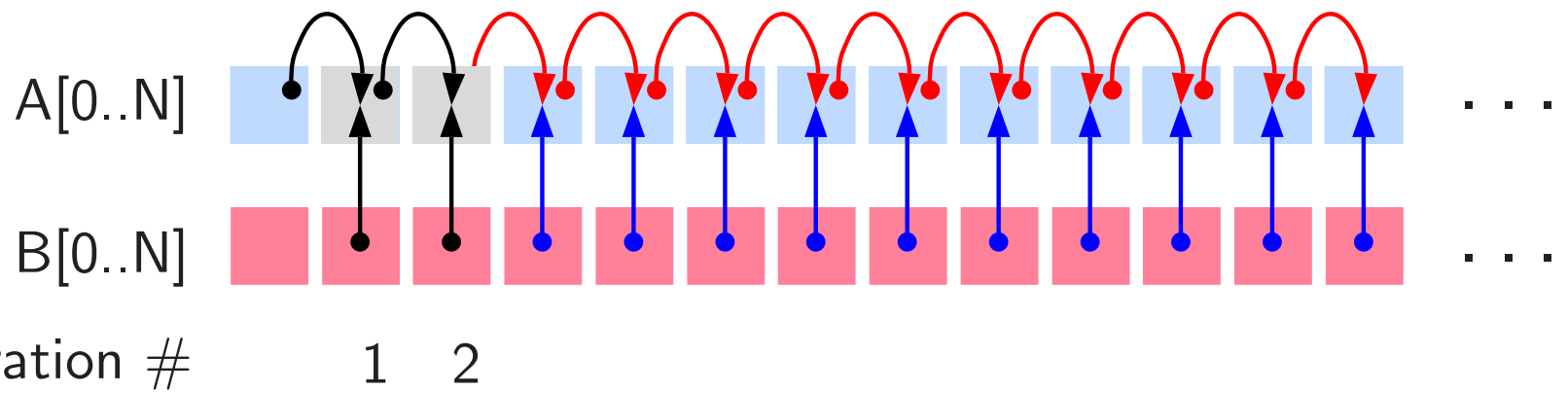


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

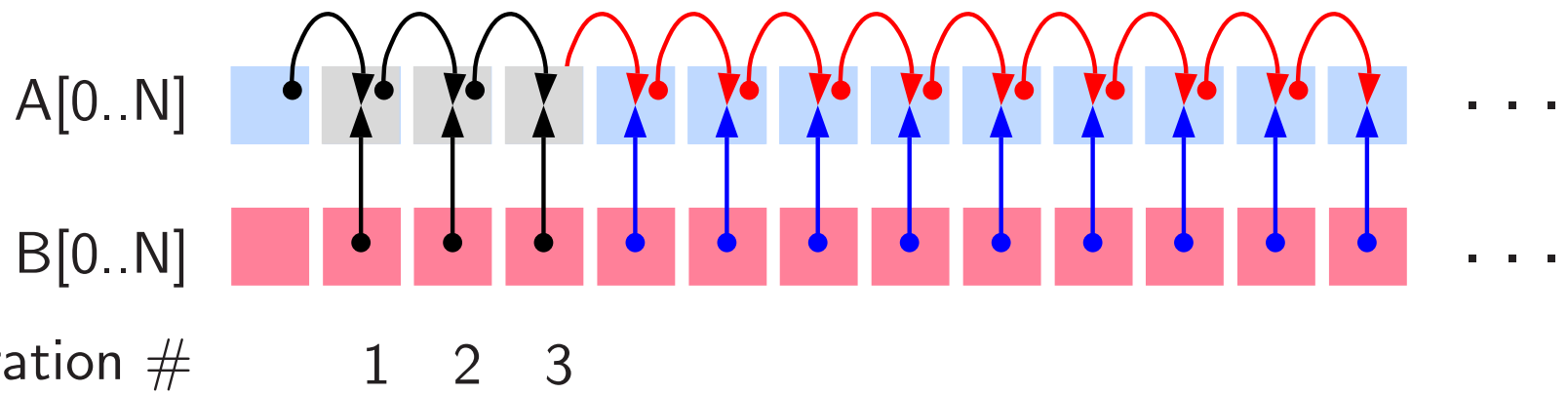


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

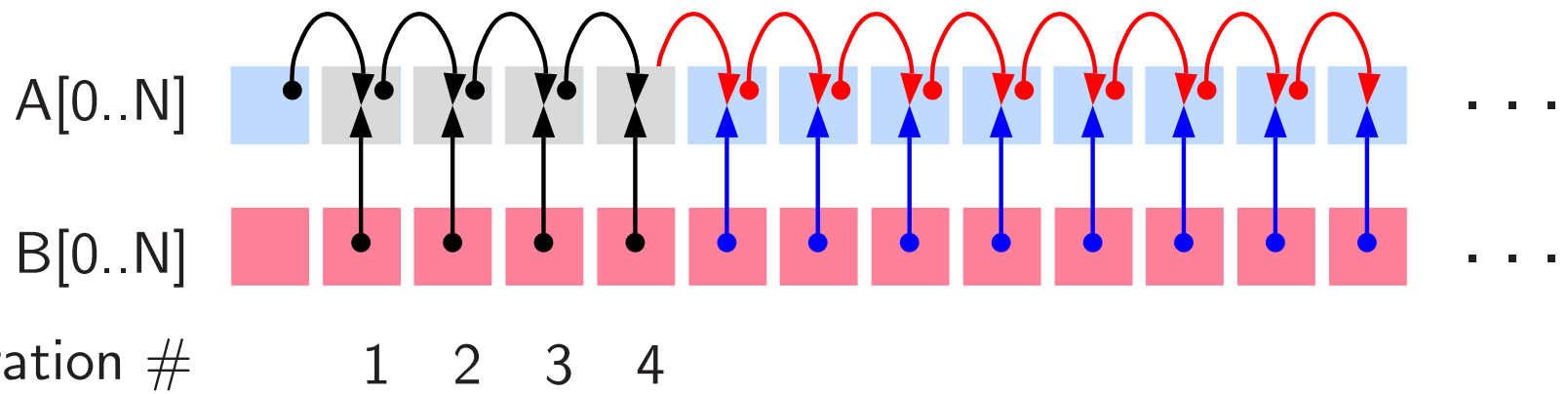


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

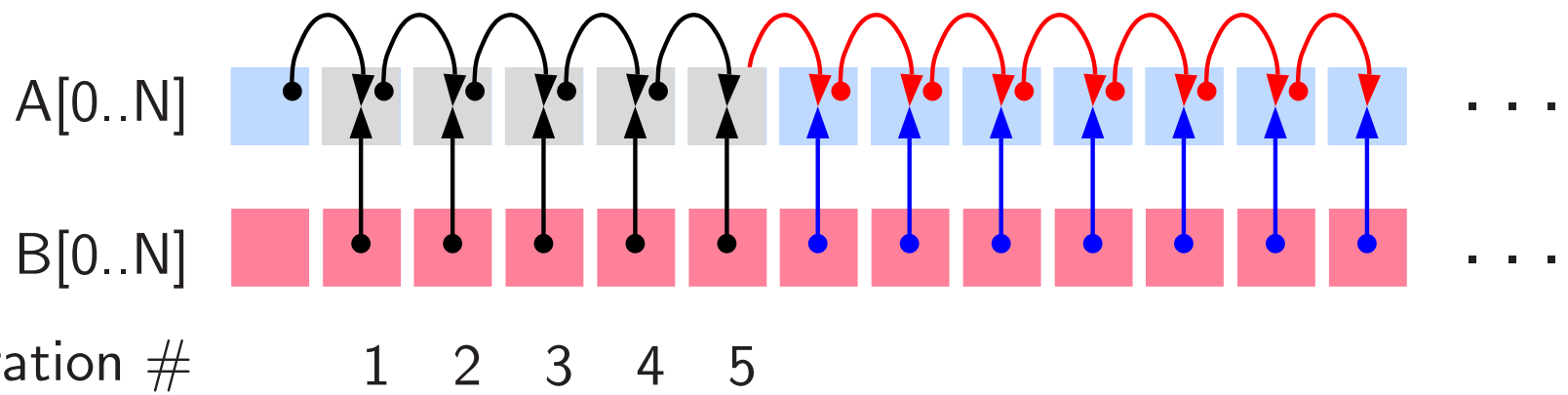


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

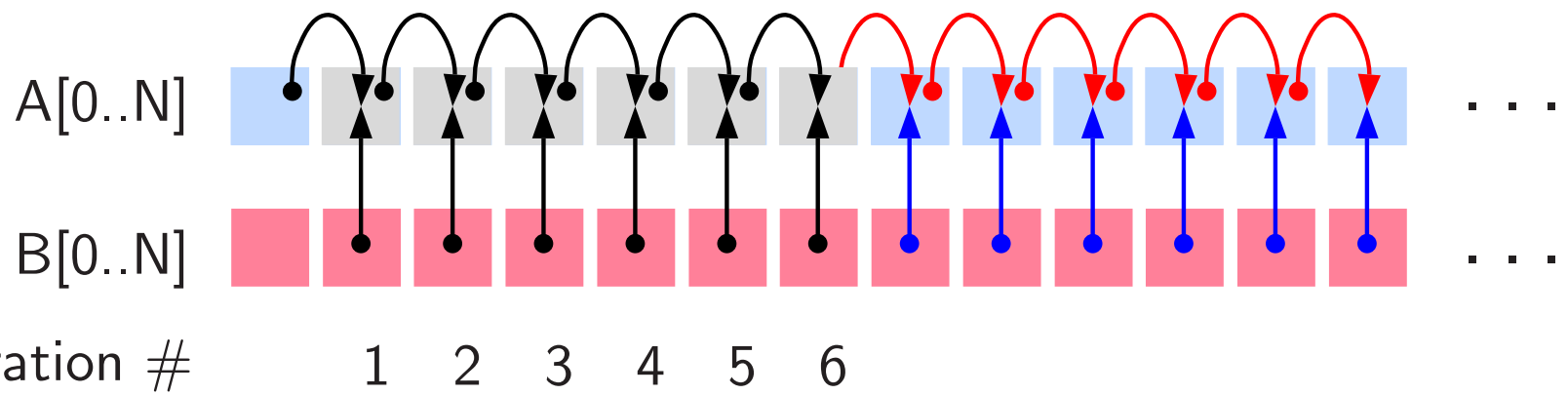


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

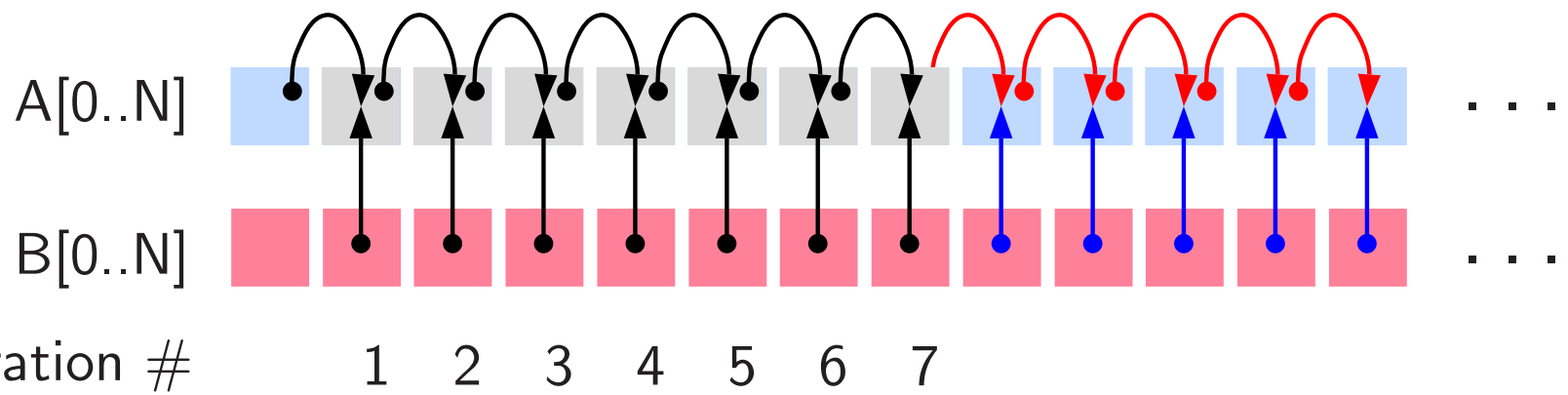


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

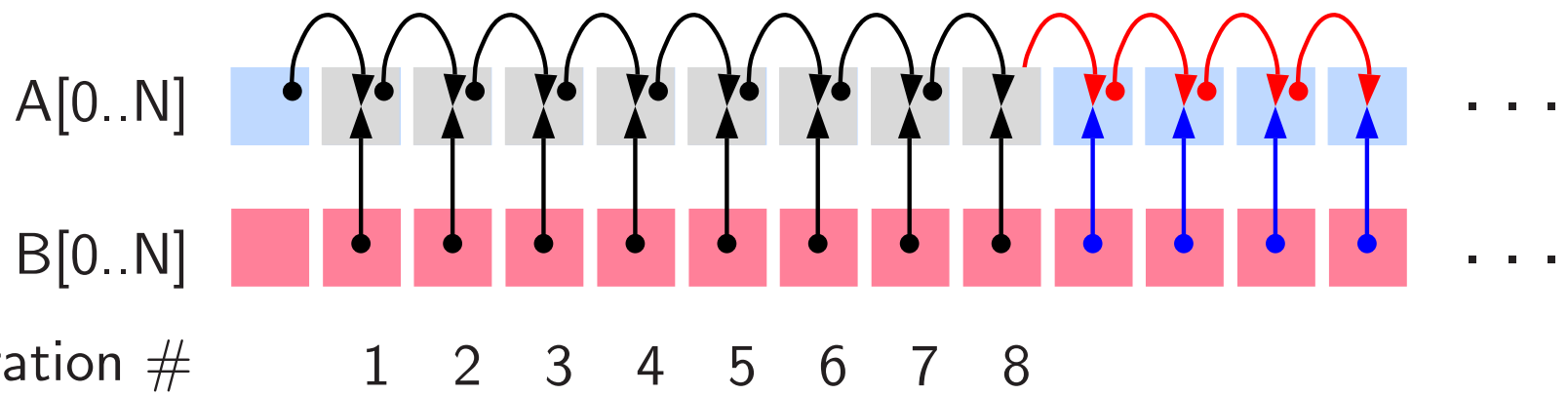


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

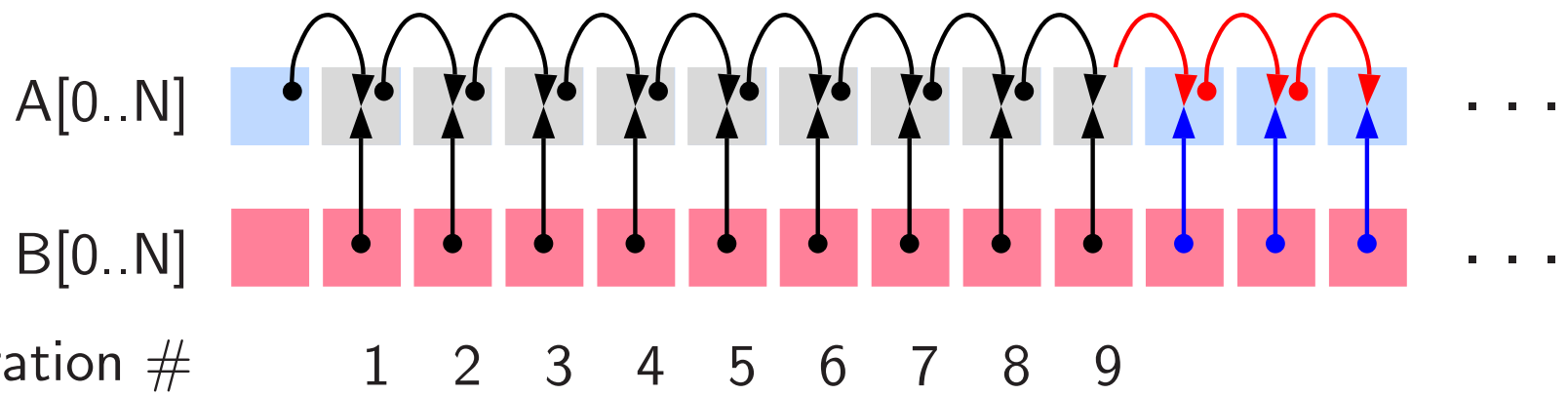


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

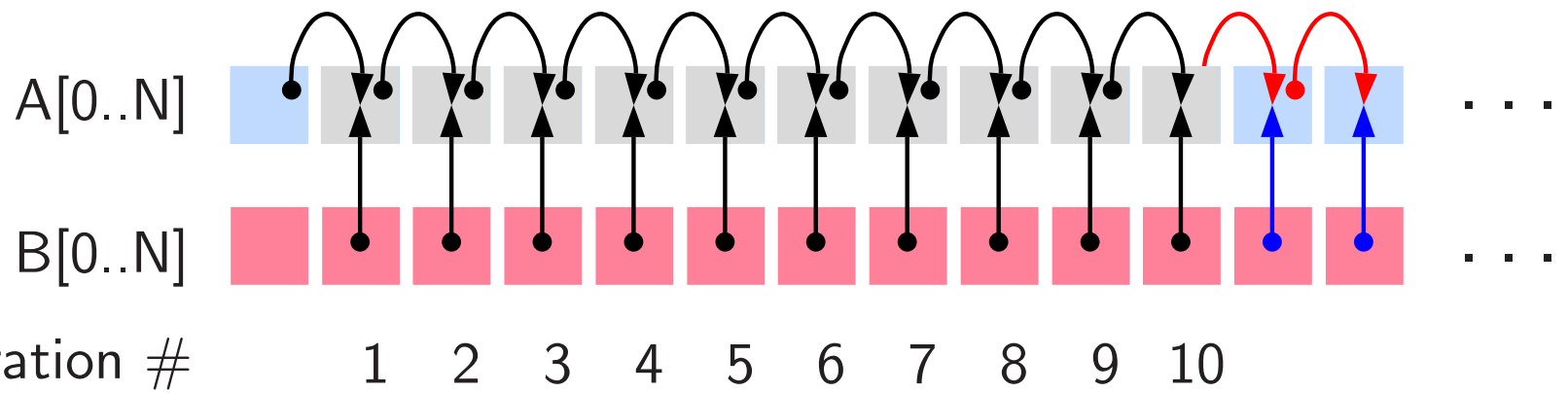


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

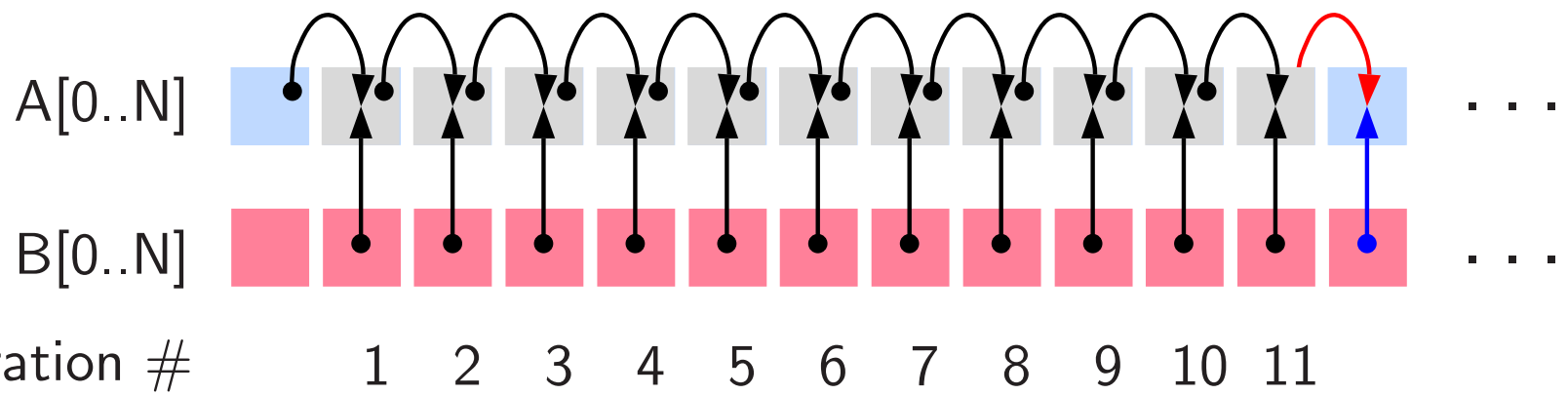


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location

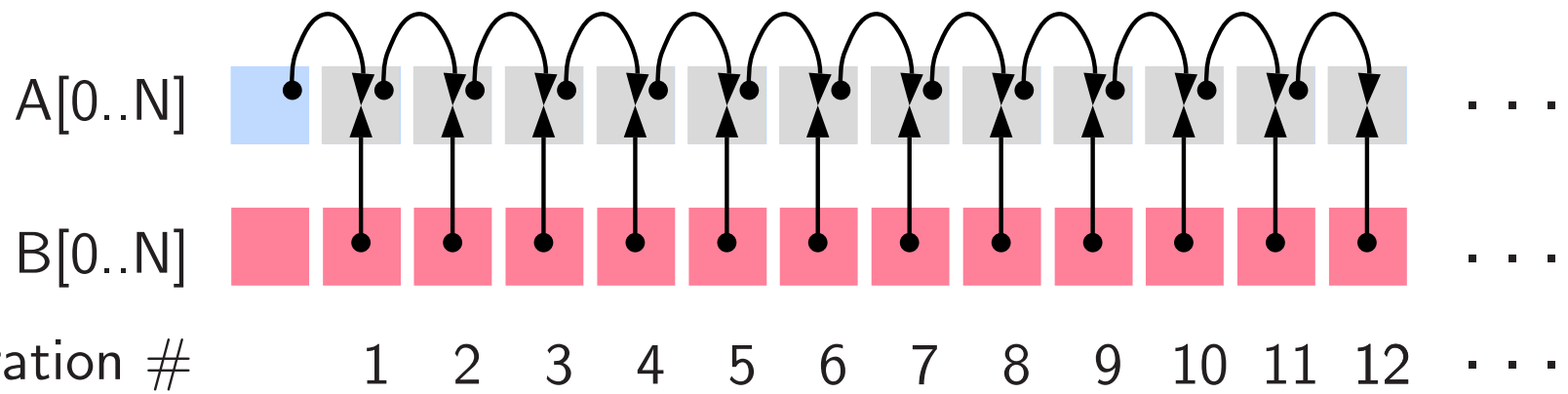


Example 3

Vectorization (SISD ⇒ SIMD) : No
Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B[N], i;
for (i=0; i<N; i++)
  A[i+1] = A[i] + B[i+1];
```

Observe reads and writes into a given location



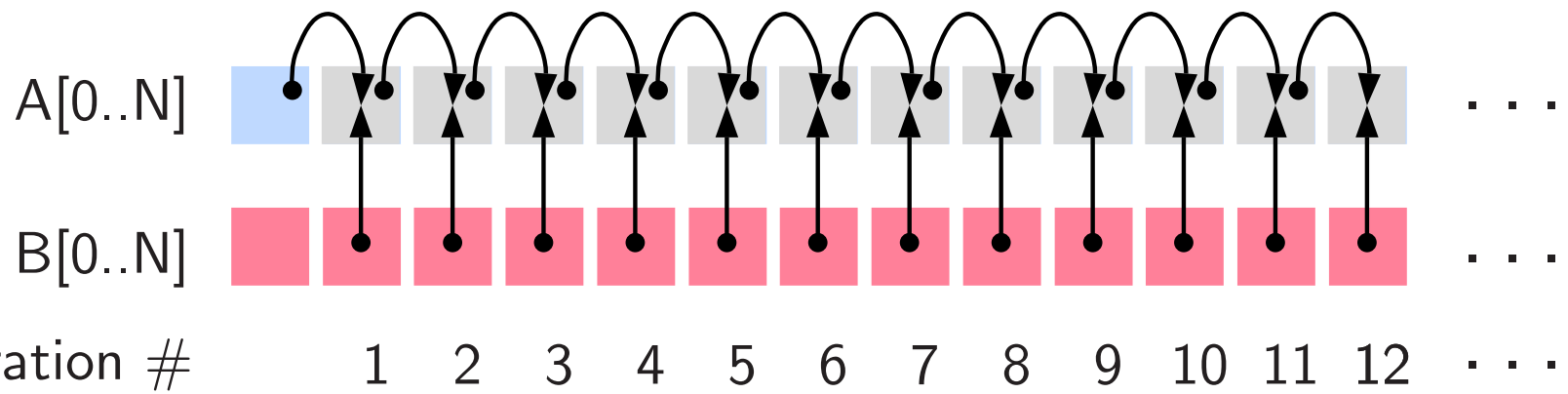
Example 3

Vectorization (SISD ⇒ SIMD) : No

Parallelization (SISD ⇒ MIMD) : No

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	
Write	Write	
Read	Read	



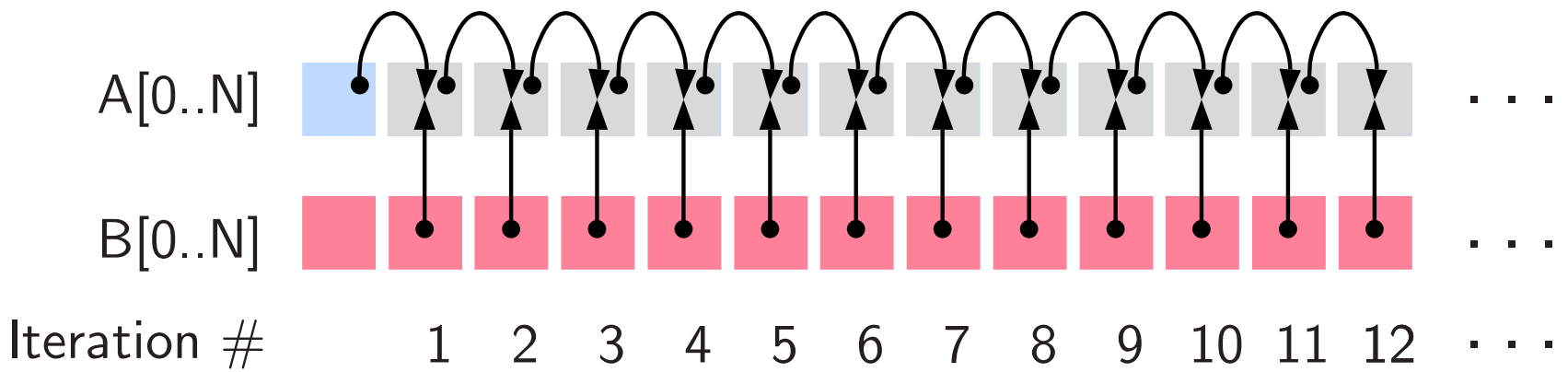
Example 3

Vectorization (SISD ⇒ SIMD) : **No**

Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	
Read	Read	



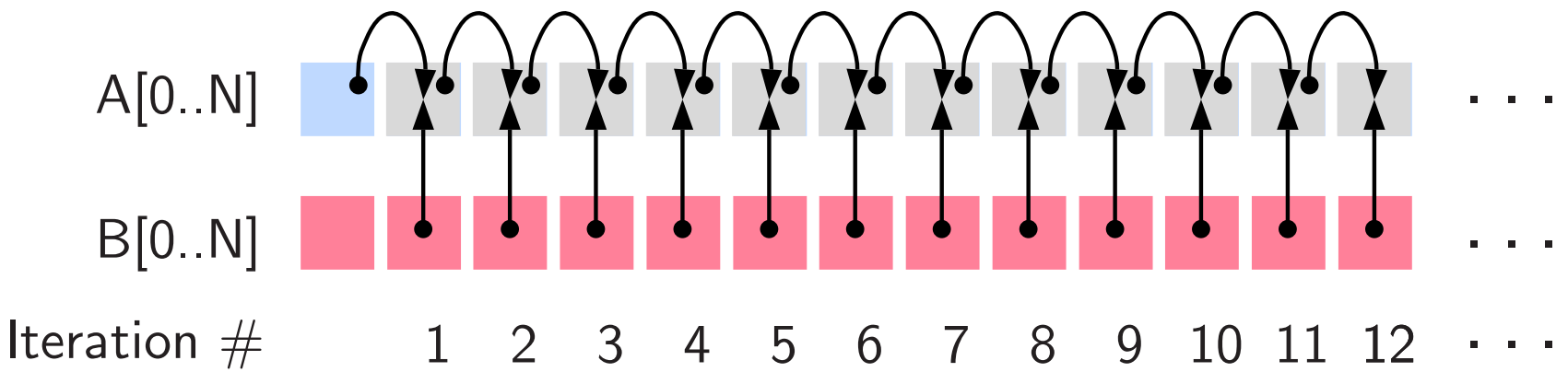
Example 3

Vectorization (SISD ⇒ SIMD) : **No**

Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	No
Read	Read	



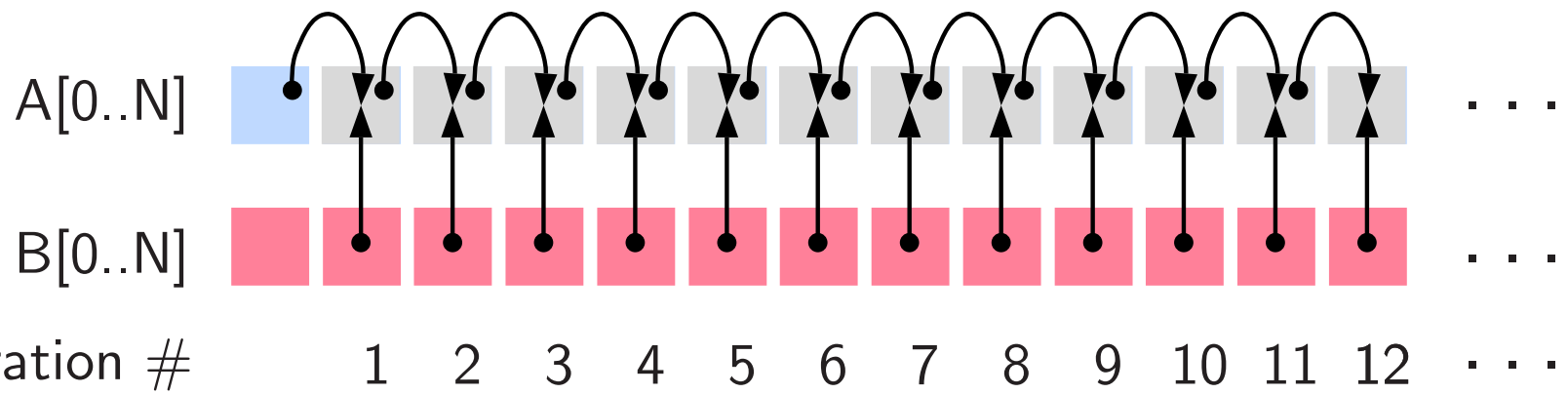
Example 3

Vectorization (SISD ⇒ SIMD) : **No**

Parallelization (SISD ⇒ MIMD) : **No**

```
int A[N], B
for (i=0; i
  A[i+1] =
```

Nature of accesses in our example		
Iteration i	Iteration $i+k$	Observation
Read	Write	No
Write	Read	Yes
Write	Write	No
Read	Read	Does not matter



Example 4

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

Example 4

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

- This case is not possible

Example 4

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

- This case is not possible
- Vectorization is a limited granularity parallelization

Example 4

Vectorization (SISD $\Rightarrow$ SIMD) : No
Parallelization (SISD $\Rightarrow$ MIMD) : Yes

- This case is not possible
- Vectorization is a limited granularity parallelization
- If parallelization is possible then vectorization is trivially possible

Data Dependence

Let statements S_i and S_j access memory location m at time instants t and $t + k$

Access in S_i	Access in S_j	Dependence	Notation
Read m	Write m	Anti (or Pseudo)	$S_i \ \bar{\delta} \ S_j$
Write m	Read m	Flow (or True)	$S_i \ \delta \ S_j$
Write m	Write m	Output (or Pseudo)	$S_i \ \delta^O \ S_j$
Read m	Read m	Does not matter	

- Pseudo dependences may be eliminated by some transformations
- True dependence prohibits parallel execution of S_i and S_j

Loop Carried and Loop Independent Dependences

Consider dependence between statements S_i and S_j in a loop

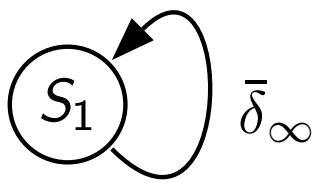
- Loop independent dependence. t and $t + k$ occur in the same iteration of a loop
  - S_i and S_j must be executed sequentially
  - Different iterations of the loop can be parallelized
- Loop carried dependence. t and $t + k$ occur in the different iterations of a loop
  - Within an iteration, S_i and S_j can be executed in parallel
  - Different iterations of the loop must be executed sequentially
- S_i and S_j may have both loop carried and loop independent dependences

Dependence in Example 1

- Program

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];    /* S1 */
```

- Dependence graph



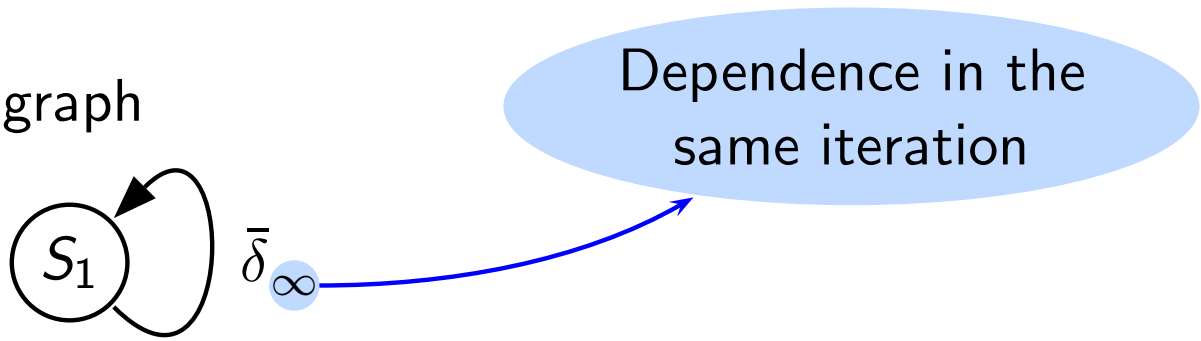
- No loop carried dependence
  Both vectorization and parallelization are possible

Dependence in Example 1

- Program

```
int A[N], B[N], i;
for (i=1; i<N; i++)
    A[i] = A[i] + B[i-1];    /* S1 */
```

- Dependence graph

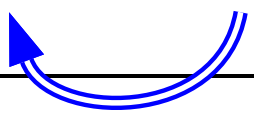


- No loop carried dependence
  Both vectorization and parallelization are possible

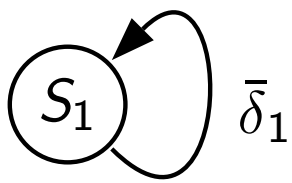
Dependence in Example 2

- Program

```
int A[N], B[N], i;
  for (i=0; i<N; i++)
     A[i] = A[i+1] + B[i];  /* S1 */
```



- Dependence graph



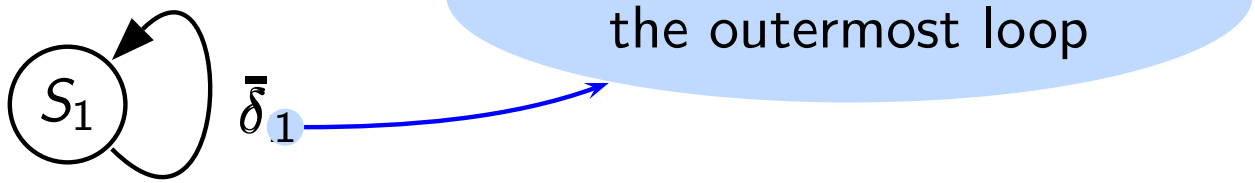
- Loop carried anti-dependence
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes

Dependence in Example 2

- Program

```
int A[N], B[N], i;
  for (i=0; i<N; i++)
     A[i] = A[i+1] + B[i];  /* S1 */
```

- Dependence graph

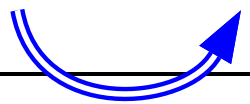


- Loop carried anti-dependence
  Parallelization is not possible
  Vectorization is possible since all reads are done before all writes

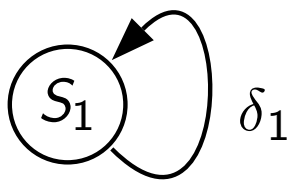
Dependence in Example 3

- Program

```
int A[N], B[N], i;
  for (i=0; i<N; i++)
     A[i+1] = A[i] + B[i+1];  /* S1 */
```



- Dependence graph



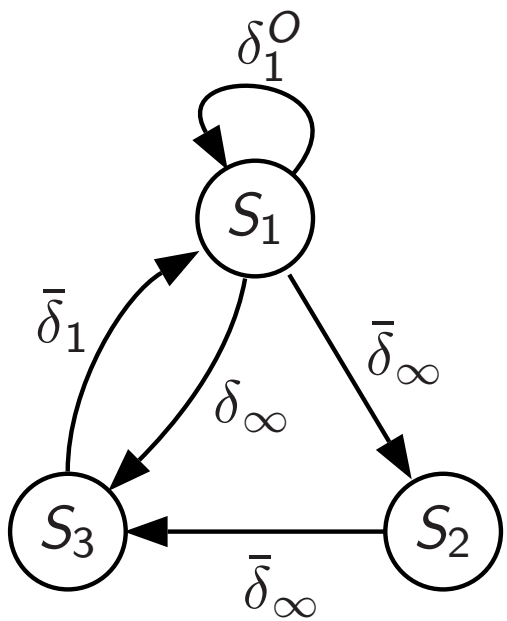
- Loop carried flow-dependence
  Neither parallelization not vectorization is possible

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph

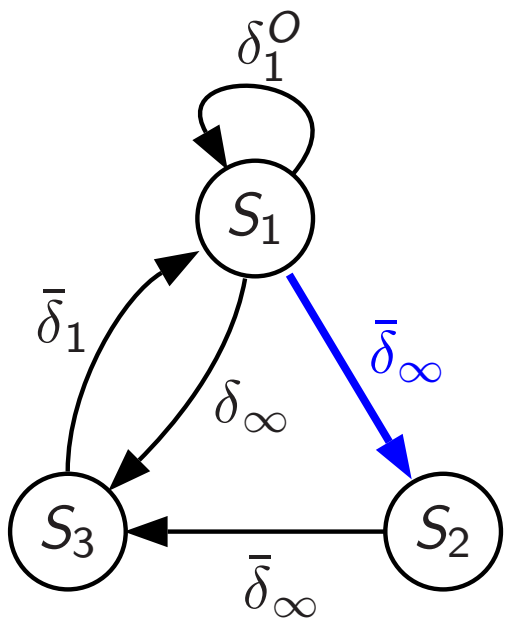


Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph



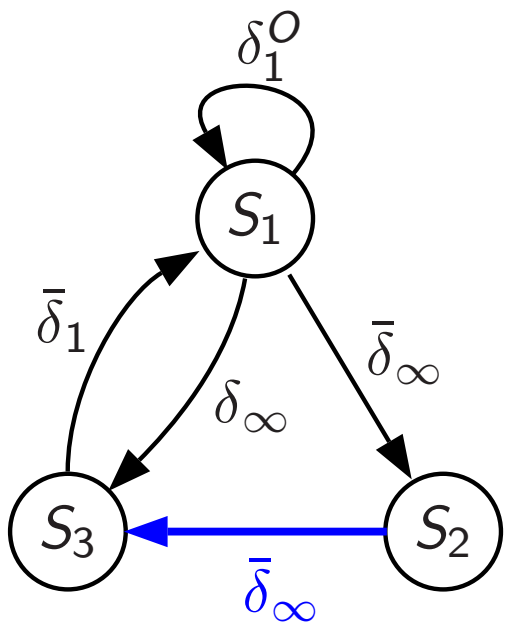
Loop independent anti dependence due to `A[i]`

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph



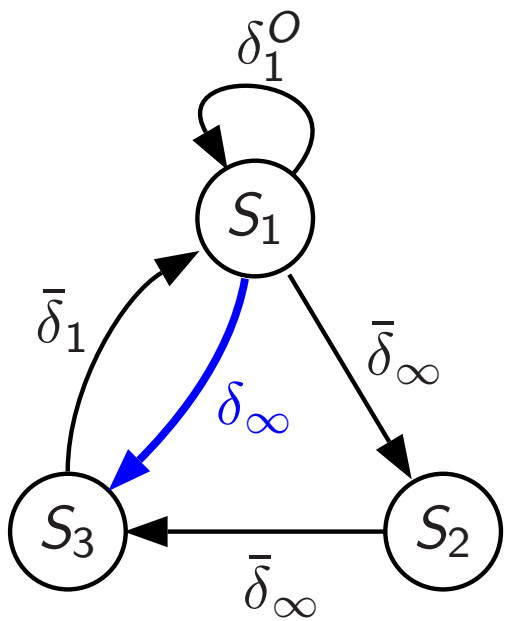
Loop independent anti dependence due to `B[i]`

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph



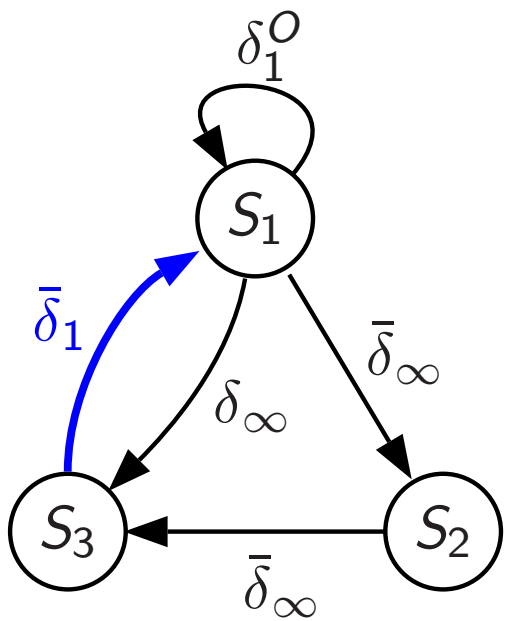
Loop independent flow dependence due to T

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph



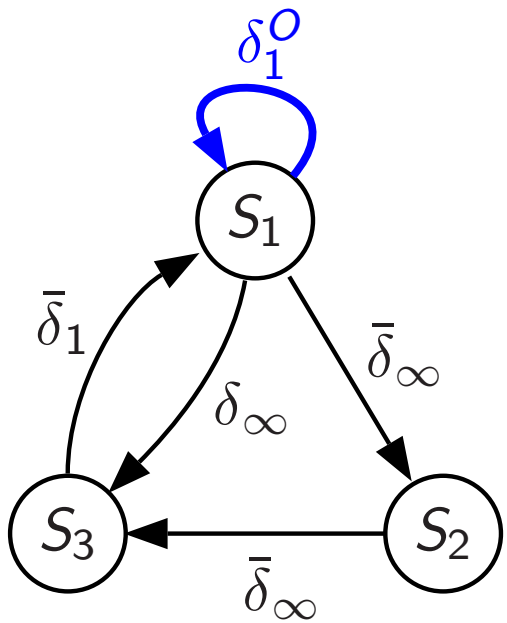
Loop carried anti dependence due to T

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph



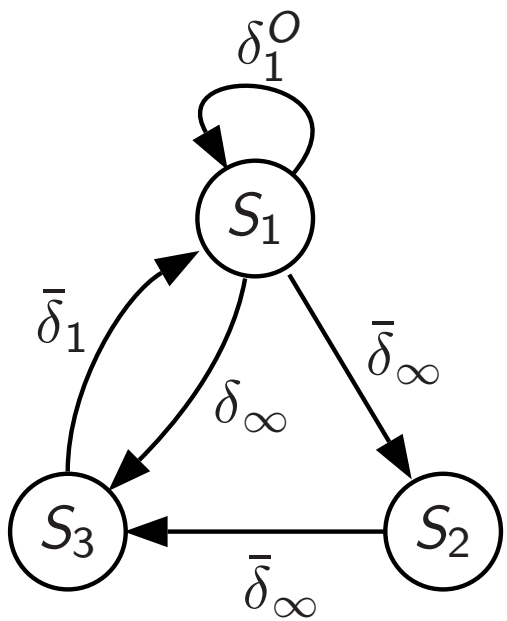
Loop carried output dependence due to T

Example 4: Dependence

Program to swap arrays

```
for (i=0; i<N; i++)
{    T = A[i];          /* S1 */
     A[i] = B[i];       /* S2 */
     B[i] = T;          /* S3 */
}
```

Dependence Graph



Tutorial Problem for Discovering Dependence

Draw the dependence graph for the following program
(Earlier program modified to swap 2-dimensional arrays)

```
for (i=0; i<N; i++)
{
    for (j=0; j<N; j++)
    {   T = A[i][j];          /* S1 */
        A[i][j] = B[i][j]; /* S2 */
        B[i][j] = T;          /* S3 */
    }
}
```

Data Dependence in Loops

- Analysis in loop is tricky, as
  - Loops may be nested
  - Different loop iterations may access same memory location
  - Arrays occur frequently
  - Far too many array locations to be treated as independent scalars

Data Dependence in Loops

- Analysis in loop is tricky, as
  - Loops may be nested
  - Different loop iterations may access same memory location
  - Arrays occur frequently
  - Far too many array locations to be treated as independent scalars
- Consider array location `A[4][9]` in the following program

```
for(i = 0; i <= 5; i ++)
  for(j = 0; j <= 4; j ++)
  {
    A[i+1][3*j] = ... ;           /* S1 */
          ...   = A[i+3][2*j+1]; /* S2 */
  }
```


Data Dependence in Loops

- Analysis in loop is tricky, as
  - Loops may be nested
  - Different loop iterations may access same memory location
  - Arrays occur frequently
  - Far too many array locations to be treated as independent scalars
- Consider array location `A[4][9]` in the following program

```
for(i = 0; i <= 5; i ++)
  for(j = 0; j <= 4; j ++)
  {
    A[i+1][3*j] = ... ;           /* S1 */
          ...   = A[i+3][2*j+1]; /* S2 */
  }
```

S2 accesses in iteration (1,4), S1 accesses in iteration (3,3)

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 1

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i+1][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: Dependence exists

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	1, 0	0, 0
0, 1	1, 1	0, 1
0, 2	1, 2	0, 2
0, 3	1, 3	0, 3
1, 0	2, 0	1, 0
1, 1	2, 1	1, 1
1, 2	2, 2	1, 2
1, 3	2, 3	1, 3
2, 0	3, 0	2, 0
2, 1	3, 1	2, 1
2, 2	3, 2	2, 2
2, 3	3, 3	2, 3
3, 0	4, 0	3, 0
3, 1	4, 1	3, 1
3, 2	4, 2	3, 2
3, 3	4, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Iteration Vectors and Index Vectors: Example 2

```
for (i=0, i<4; i++)
  for (j=0; j<4; j++)
  {
     a[i][j] = a[i][j] + 2;
  }
```

Loop carried dependence exists if

- there are two distinct iteration vectors such that
- the index vectors of LHS and RHS are identical

Conclusion: No dependence

Iteration Vector	Index Vector	
	LHS	RHS
0, 0	0, 0	0, 0
0, 1	0, 1	0, 1
0, 2	0, 2	0, 2
0, 3	0, 3	0, 3
1, 0	1, 0	1, 0
1, 1	1, 1	1, 1
1, 2	1, 2	1, 2
1, 3	1, 3	1, 3
2, 0	2, 0	2, 0
2, 1	2, 1	2, 1
2, 2	2, 2	2, 2
2, 3	2, 3	2, 3
3, 0	3, 0	3, 0
3, 1	3, 1	3, 1
3, 2	3, 2	3, 2
3, 3	3, 3	3, 3

Data Dependence Theorem [KA02]

There exists a dependence from statement S_1 to statement S_2 in common nest of loops if and only if there exist two iteration vectors **i** and **j** for the nest, such that

1. **i** $<$ **j** or **i** $=$ **j** and there exists a path from S_1 to S_2 in the body of the loop,
2. statement S_1 accesses memory location M on iteration **i** and statement S_2 accesses location M on iteration **j**, and
3. one of these accesses is a write access.

Part 6

Parallelization and Vectorization in GCC

Parallelization and Vectorization in GCC

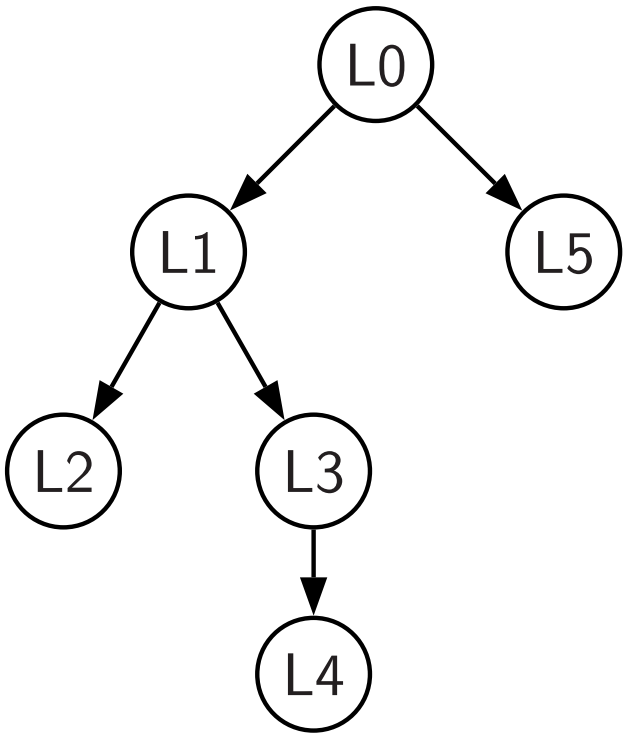
Implementation Issues

- Getting loop information (Loop discovery)
- Finding value spaces of induction variables, index expressions, and pointer accesses
- Analyzing data dependence
- Performing transformations

Loop Information

```
Loop0
{    Loop1
     {    Loop2
          {
          }
          Loop3
          {  Loop4
             {
             }
          }
     }
     Loop5
     {
     }
}
```

Loop Tree



Representing Value Spaces of Variables and Expressions

Chain of Recurrences: 3-tuple ⟨Starting Value, modification, stride⟩ [BWZ94, KMZ98]

```
for (i=3; i<=15; i=i+3)
{
    for (j=11; j>=1; j=j-2)
    {
        A[i+1][2*j-1] = ...
    }
}
```

Entity	CR
Induction variable i	$\{3, +, 3\}$
Induction variable j	$\{11, +, -2\}$
Index expression i+1	$\{4, +, 3\}$
Index expression 2*j-1	$\{21, +, -4\}$

Advantages of Chain of Recurrences

CR can represent any affine expression

$\Rightarrow$ Accesses through pointers can also be tracked

```
int A[32], B[32];
int i, *p;
p = &B
for(i = 2; i<N; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```


Advantages of Chain of Recurrences

CR can represent any affine expression

$\Rightarrow$ Accesses through pointers can also be tracked

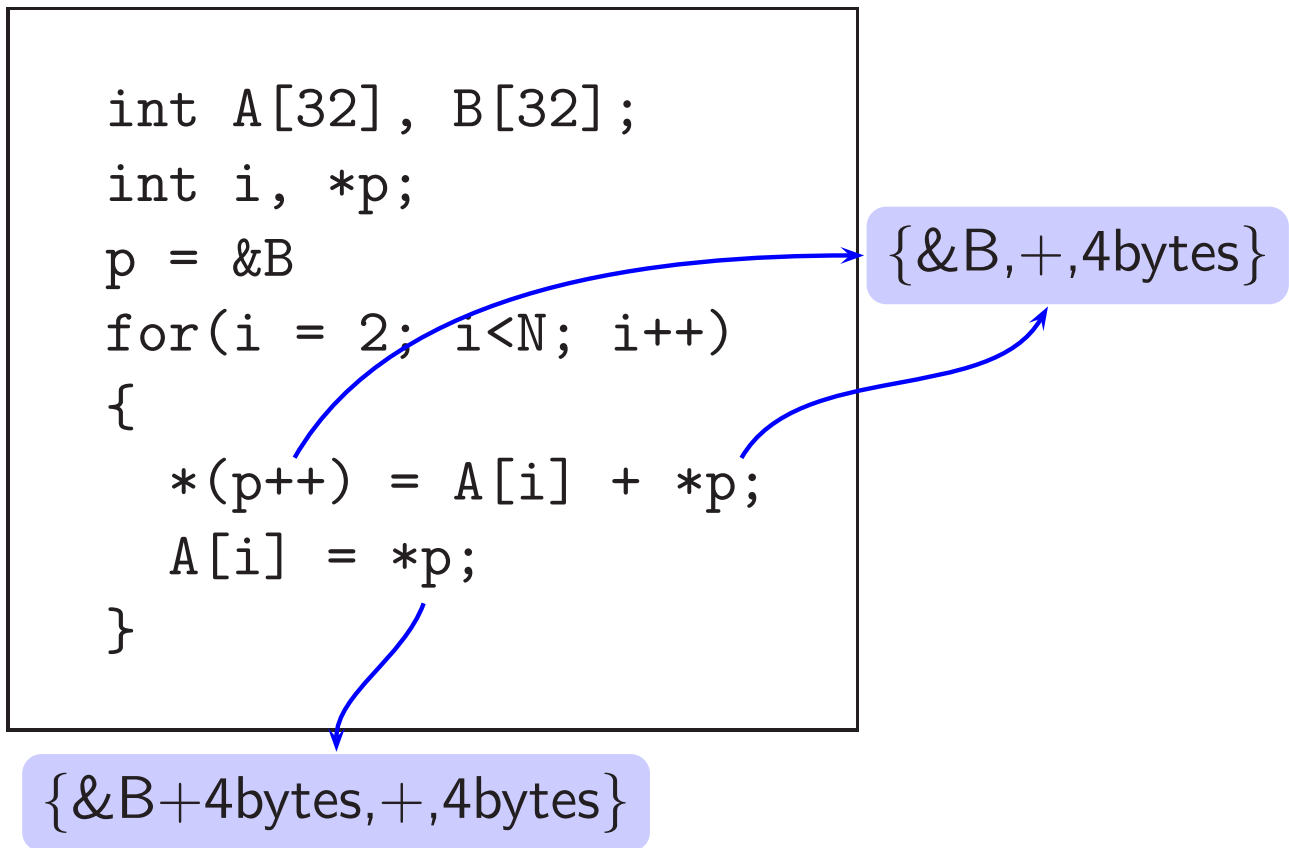
```
int A[32], B[32];
int i, *p;
p = &B
for(i = 2; i<N; i++)
{
  *(p++) = A[i] + *p;
  A[i] = *p;
}
```

{&B,+,4bytes}

Advantages of Chain of Recurrences

CR can represent any affine expression

⇒ Accesses through pointers can also be tracked



Transformation Passes in GCC

- A total of 196 unique pass names initialized in `${SOURCE}/gcc/passes.c`
  - Some passes are called multiple times in different contexts
    Conditional constant propagation and dead code elimination are called thrice
  - Some passes are only demo passes (eg. data dependence analysis)
  - Some passes have many variations (eg. special cases for loops)
    Common subexpression elimination, dead code elimination
- The pass sequence can be divided broadly in two parts
  - Passes on Gimple
  - Passes on RTL
- Some passes are organizational passes to group related passes

Passes On Gimple

Pass Group	Examples	Number of passes
Lowering	Gimple IR, CFG Construction	12
Interprocedural Optimizations	Conditional Constant Propagation, Inlining, SSA Construction	36
Intraprocedural Optimizations	Constant Propagation, Dead Code Elimination, PRE	40
Loop Optimizations	Vectorization, Parallelization	24
Remaining Intraprocedural Optimizations	Value Range Propagation, Rename SSA	23
Generating RTL		01
Total number of passes on Gimple		136

Passes On Gimple

Pass Group	Examples	Number of passes
Lowering	Gimple IR, CFG Construction	12
Interprocedural Optimizations	Conditional Constant Propagation, Inlining, SSA Construction	36
Intraprocedural Optimizations	Constant Propagation, Dead Code Elimination, PRE	40
Loop Optimizations	Vectorization, Parallelization	24
Remaining Intraprocedural Optimizations	Value Range Propagation, Rename SSA	23
Generating RTL		01
Total number of passes on Gimple		136

Our Focus is Vectorization and Parallelization

Passes On RTL

Pass Group	Examples	Number of passes
Intraprocedural Optimizations	CSE, Jump Optimization	15
Loop Optimizations	Loop Invariant Movement, Peeling, Unswitching	7
Machine Dependent Optimizations	Register Allocation, Instruction Scheduling, Peephole Optimizations	59
Assembly Emission and Finishing		03
Total number of passes on RTL		84

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_dce_loop);
    NEXT_PASS (pass_lim);
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_tree_unswitch);
    NEXT_PASS (pass_scev_cprop);
    NEXT_PASS (pass_empty_loop);
    NEXT_PASS (pass_record_bounds);
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_linear_transform);
    NEXT_PASS (pass_graphite_transforms);
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_dce_loop);
    NEXT_PASS (pass_lim);
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_tree_unswitch);
    NEXT_PASS (pass_scev_cprop);
    NEXT_PASS (pass_empty_loop);
    NEXT_PASS (pass_record_bounds);
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_linear_transform);
    NEXT_PASS (pass_graphite_transforms);
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC

```
NEXT_PASS (pass_tree_loop);
  {
    struct opt_pass **p = &pass_tree_loop.pass.sub;
    NEXT_PASS (pass_tree_loop_init);
    NEXT_PASS (pass_copy_prop);
    NEXT_PASS (pass_dce_loop);
    NEXT_PASS (pass_lim);
    NEXT_PASS (pass_predcom);
    NEXT_PASS (pass_tree_unswitch);
    NEXT_PASS (pass_scev_cprop);
    NEXT_PASS (pass_empty_loop);
    NEXT_PASS (pass_record_bounds);
    NEXT_PASS (pass_check_data_deps);
    NEXT_PASS (pass_loop_distribution);
    NEXT_PASS (pass_linear_transform);
    NEXT_PASS (pass_graphite_transforms);
    NEXT_PASS (pass_iv_canon);
    NEXT_PASS (pass_if_conversion);
    NEXT_PASS (pass_vectorize);
      {
        struct opt_pass **p = &pass_vectorize.pass.sub;
        NEXT_PASS (pass_lower_vector_ssa);
        NEXT_PASS (pass_dce_loop);
      }
    NEXT_PASS (pass_complete_unroll);
    NEXT_PASS (pass_parallelize_loops);
    NEXT_PASS (pass_loop_prefetch);
    NEXT_PASS (pass_iv_optimize);
    NEXT_PASS (pass_tree_loop_done);
  }
```

- Passes on tree-SSA form
  A variant of Gimple IR
- Discover parallelism and transform IR
- Parameterized by some machine dependent features (Vectorization factor, alignment etc.)
- Mapping the transformed IR to machine instructions is achieved through machine descriptions

Loop Transformation Passes in GCC: Our Focus

Data Dependence	Pass variable name	`pass_check_data_deps`
	Enabling switch	`-fcheck-data-deps`
	Dump switch	`-fdump-tree-ckdd`
	Dump file extension	`.ckdd`
Loop Distribution	Pass variable name	`pass_loop_distribution`
	Enabling switch	`-ftree-loop-distribution`
	Dump switch	`-fdump-tree-ldist`
	Dump file extension	`.ldist`
Vectorization	Pass variable name	`pass_vectorize`
	Enabling switch	`-ftree-vectorize`
	Dump switch	`-fdump-tree-vect`
	Dump file extension	`.vect`
Parallelization	Pass variable name	`pass_parallelize_loops`
	Enabling switch	`-ftree-parallelize-loops=n`
	Dump switch	`-fdump-tree-parloops`
	Dump file extension	`.parloops`

Compiling for Emitting Dumps

- Other necessary command line switches
  - `-O3 -fdump-tree-all`
    `-O3` enables `-ftree-vectorize`. Other flags must be enabled explicitly
- Processor related switches to enable transformations apart from analysis
  - `-mtune=pentium -msse4`
- Other useful options
  - Suffixing `-all` to all dump switches
  - `-S` to stop the compilation with assembly generation
  - `--verbose-asm` to see more detailed assembly dump
  - `-fno-predictive-commoning` to disable predictive commoning optimization

Example 1: Observing Data Dependence

Step 0: Compiling

```
#include <stdio.h>
int a[200];
int main()
{
   int i, n;
   for (i=0; i<150; i++)
   {
      a[i] = a[i+1] + 2;
   }
   return 0;
}
```

```
gcc -fcheck-data-deps -fdump-tree-ckdd-all -O3 -S datadep.c
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
#include <stdio.h>
int a[200];
int main()
{
    int i, n;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
}
```

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
#include <stdio.h>
int a[200];
int main()
{
    int i, n;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
}
```

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
#include <stdio.h>
int a[200];
int main()
{
    int i, n;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
}
```

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
#include <stdio.h>
int a[200];
int main()
{
    int i, n;
    for (i=0; i<150; i++)
    {
        a[i] = a[i+1] + 2;
    }
    return 0;
}
```

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(evolution_function = 0, +, 1_1)
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

(scalar_evolution = 1, +, 1_1)

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                    address: 4
aligned to: 128
(chrec = 1, +, 1_1)
```

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                    address:  0
aligned to: 128
base_object: a[0]
(chrec = 0, +, 1_1)
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
(evolution_function = 0, +, 1_1)
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

(scalar_evolution = 1, +, 1_1)

Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                    address: 4
aligned to: 128
(chrec = 1, +, 1_1)
```


Example 1: Observing Data Dependence

Step 2: Understanding the chain of recurrences

```
<bb 3>:
  # i_13 = PHI <i_4(4), 0(2)>
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  if (i_4 <= 149)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

```
base_address: &a
offset from base address: 0
constant offset from base
                    address:  0
aligned to: 128
base_object: a[0]
(chrec = 0, +, 1_1)
```


Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$ • Solve for $0 \leq x, y < 150$ $y = x + 1$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$ • Solve for $0 \leq x, y < 150$ $y = x + 1$ $\Rightarrow \quad x - y + 1 = 0$	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View	CFG View
• Relevant assignment is $a[i] = a[i+1] + 2$ • Solve for $0 \leq x, y < 150$ $y = x + 1$ $\Rightarrow \quad x - y + 1 = 0$ • Find min and max of LHS	

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View	CFG View

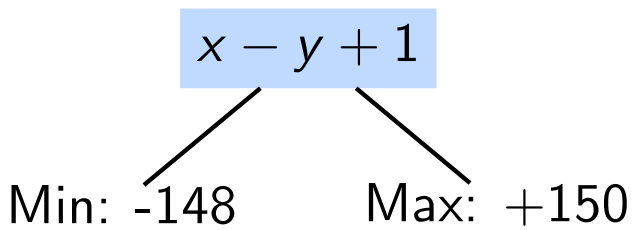
- Relevant assignment is

  $a[i] = a[i+1] + 2$

- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View	CFG View

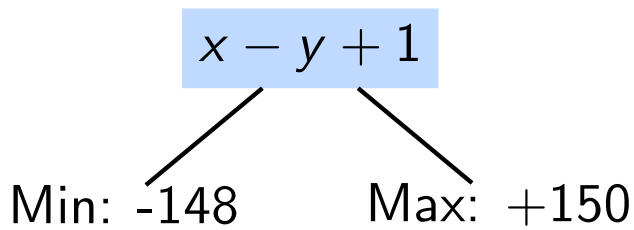
- Relevant assignment is

  $a[i] = a[i+1] + 2$

- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



RHS belongs to $[-148, +150]$ and dependence may exist

Example 1: Observing Data Dependence

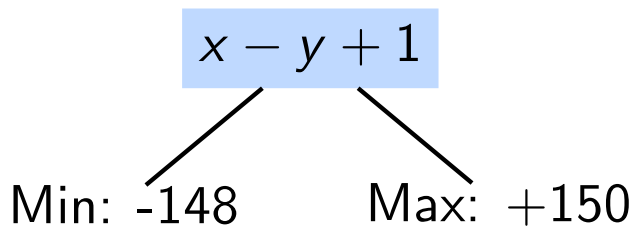
Step 3: Understanding Banerjee's test [Ban96]

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View

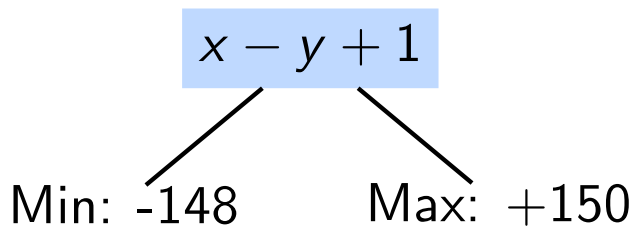
- Relevant assignment is

  $a[i] = a[i+1] + 2$

- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```

- Chain of recurrences are

  For `a[i_4]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`
  ```

Example 1: Observing Data Dependence

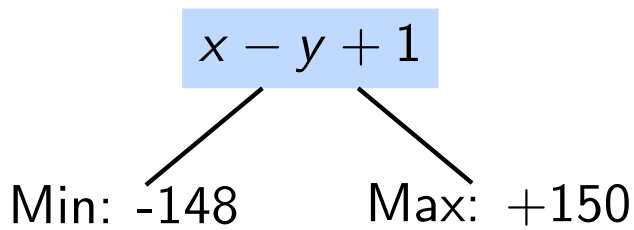
Step 3: Understanding Banerjee's test [Ban96]

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
- Chain of recurrences are

  For `a[i_4]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`
- Solve for $0 \leq$ `x_1` < 150

  `1 + 1*x_1 - 0 + 1*x_1` $= 0$


  ```

Example 1: Observing Data Dependence

Step 3: Understanding Banerjee's test [Ban96]

Source View

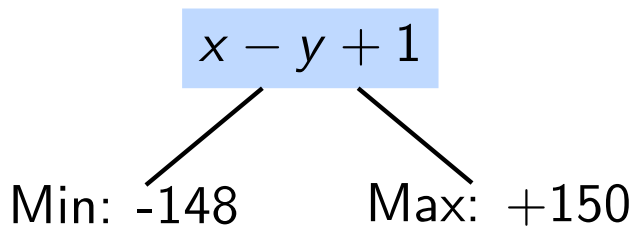
- Relevant assignment is

  $a[i] = a[i+1] + 2$

- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```

- Chain of recurrences are

  For `a[i_4]`: {1, +, 1}_1

  For `a[i_13]`: {0, +, 1}_1

- Solve for $0 \leq$ x_1 < 150

  `1 + 1*x_1 - 0 + 1*x_1` $= 0$

- Min of LHS is -148, Max is +150


  ```

Example 1: Observing Data Dependence

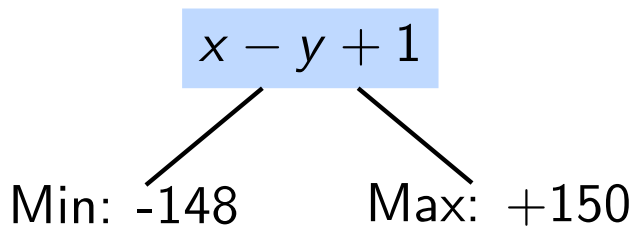
Step 3: Understanding Banerjee's test [Ban96]

Source View

- Relevant assignment is
  $a[i] = a[i+1] + 2$
- Solve for $0 \leq x, y < 150$

$$\begin{aligned} y &= x+1 \\ \Rightarrow \quad x - y + 1 &= 0 \end{aligned}$$

- Find min and max of LHS



RHS belongs to $[-148, +150]$ and dependence may exist

CFG View

- ```
  i_4 = i_13 + 1;
  D.1240_5 = a[i_4];
  D.1241_6 = D.1240_5 + 2;
  a[i_13] = D.1241_6;
  ```
- Chain of recurrences are

  For `a[i_4]`: `{1, +, 1}_1`

  For `a[i_13]`: `{0, +, 1}_1`
- Solve for $0 \leq$ `x_1` < 150

  `1 + 1*x_1 - 0 + 1*x_1` $= 0$
- Min of LHS is -148, Max is +150
- Dependence may exist
  ```

Example 2: Observing Vectorization and Parallelization

Step 0: Compiling with `-fno-predictive-commoning`

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=4 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4`

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 1: Examining the control flow graph

Program	Control Flow Graph

```
int a[256], b[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = b[i];
    }
    return 0;
}
```

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 2: Observing the final decision about vectorization

```
parvec.c:9: note: LOOP VECTORIZED.
parvec.c:6: note: vectorized 1 loops in function.
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 3: Examining the vectorized control flow graph

Original control flow graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Transformed control flow graph

```
...
vect_var_.31_18 = *vect_pb.25_16;
*vect_pa.32_21 = vect_var_.31_18;
vect_pb.25_17 = vect_pb.25_16 + 16;
vect_pa.32_22 = vect_pa.32_21 + 16;
ivtmp.38_24 = ivtmp.38_23 + 1;
if (ivtmp.38_24 < 64)
  goto <bb 4>;
else
  goto <bb 5>;
...
```

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads

Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
      /* execute the loop body to be parallelized */
}
```


Example 2: Observing Vectorization and Parallelization

Step 4: Understanding the strategy of parallel execution

- Create threads t_i for $1 \leq i \leq$ MAX_THREADS
- Assigning start and end iteration for each thread
  $\Rightarrow$ Distribute iteration space across all threads
- Create the following code body for each thread t_i

```
for (j=start_for_thread_i; j<=end_for_thread_i; j++)
{
      /* execute the loop body to be parallelized */
}
```

- All threads are executed in parallel

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```


Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Get the number of threads

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Get thread identity

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Perform load calculations

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Assign start iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Assign end iteration to the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 5: Examining the thread creation in parallelized control flow graph

```
D.1299_7 = __builtin_omp_get_num_threads ();
D.1300_9 = __builtin_omp_get_thread_num ();
D.1302_10 = 255 / D.1299_7;
D.1303_11 = D.1302_10 * D.1299_7;
D.1304_12 = D.1303_11 != 255;
D.1305_13 = D.1304_12 + D.1302_10;
ivtmp.28_14 = D.1305_13 * D.1300_9;
D.1307_15 = ivtmp.28_14 + D.1305_13;
D.1308_16 = MIN_EXPR <D.1307_15, 255>;
if (ivtmp.28_14 >= D.1308_16)
  goto <bb 3>;
```

Start execution of iterations of the chosen thread

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

<bb 3>:
# i_14 = PHI <i_6(4), 0(2)> 	
D.1666_5 = b[i_14];
a[i_14] = D.1666_5;
i_6 = i_14 + 1;
if (i_6 <= 255)
goto <bb 4>;
else
goto <bb 5>;
<bb 4>:
goto <bb 3>;
	<bb 4>:
	i.29_21 = (int) ivtmp.28_18;
	D.1312_23 = (*b.31_4)[i.29_21];
	(*a.32_5)[i.29_21] = D.1312_23;
	ivtmp.28_19 = ivtmp.28_18 + 1;
	if (D.1308_16 > ivtmp.28_19)
	goto <bb 4>;
	else
	goto <bb 3>;

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

<bb 3>:
# i_14 = PHI <i_6(4), 0(2)> 	
D.1666_5 = b[i_14];
a[i_14] = D.1666_5;
i_6 = i_14 + 1;
if (i_6 <= 255)
goto <bb 4>;
else
goto <bb 5>;
<bb 4>:
goto <bb 3>;
	<bb 4>:
	i.29_21 = (int) ivtmp.28_18;
	D.1312_23 = (*b.31_4)[i.29_21];
	(*a.32_5)[i.29_21] = D.1312_23;
	ivtmp.28_19 = ivtmp.28_18 + 1;
	if (D.1308_16 > ivtmp.28_19)
	goto <bb 4>;
	else
	goto <bb 3>;

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 4>:
  i.29_21 = (int) ivtmp.28_18;
  D.1312_23 = (*b.31_4)[i.29_21];
  (*a.32_5)[i.29_21] = D.1312_23;
  ivtmp.28_19 = ivtmp.28_18 + 1;
  if (D.1308_16 > ivtmp.28_19)
    goto <bb 4>;
  else
    goto <bb 3>;
```

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph	Parallel loop body

<bb 3>:
# i_14 = PHI <i_6(4), 0(2)> 	
D.1666_5 = b[i_14];
a[i_14] = D.1666_5;
i_6 = i_14 + 1;
if (i_6 <= 255)
goto <bb 4>;
else
goto <bb 5>;
<bb 4>:
goto <bb 3>;
	<bb 4>:
	i.29_21 = (int) ivtmp.28_18;
	D.1312_23 = (*b.31_4)[i.29_21];
	(*a.32_5)[i.29_21] = D.1312_23;
	ivtmp.28_19 = ivtmp.28_18 + 1;
	if (D.1308_16 > ivtmp.28_19)
	goto <bb 4>;
	else
	goto <bb 3>;

Example 2: Observing Vectorization and Parallelization

Step 6: Examining the loop body to be executed by a thread

Control Flow Graph

```
<bb 3>:
  # i_14 = PHI <i_6(4), 0(2)>
  D.1666_5 = b[i_14];
  a[i_14] = D.1666_5;
  i_6 = i_14 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;
<bb 4>:
  goto <bb 3>;
```

Parallel loop body

```
<bb 4>:
  i.29_21 = (int) ivtmp.28_18;
  D.1312_23 = (*b.31_4)[i.29_21];
  (*a.32_5)[i.29_21] = D.1312_23;
  ivtmp.28_19 = ivtmp.28_18 + 1;
  if (D.1308_16 > ivtmp.28_19)
    goto <bb 4>;
  else
    goto <bb 3>;
```

Example 3: Vectorization but No Parallelization

Step 0: Compiling with
`-fno-predictive-commoning -fdump-tree-vect-all -msse4`

```
int a[256];
int main()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i] = a[i+4];
    }
    return 0;
}
```

Example 3: Vectorization but No Parallelization

Step 1: Observing the final decision about vectorization

```
vecnopar.c:8: note: LOOP VECTORIZED.
vecnopar.c:5: note: vectorized 1 loops in function.
```


Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

Step 2: Examining vectorization

Control Flow Graph

```
<bb 3>:
  # i_13 = PHI <i_6(4), 0(2)>
  D.1665_4 = i_13 + 4;
  D.1666_5 = a[D.1665_4];
  a[i_13] = D.1666_5;
  i_6 = i_13 + 1;
  if (i_6 <= 255)
    goto <bb 4>;
  else
    goto <bb 5>;

<bb 4>:
  goto <bb 3>;
```

Vectorized Control Flow Graph

```
  a.31_11 = (vector int *) &a;
  vect_pa.30_15 = a.31_11 + 16;
  vect_pa.25_16 = vect_pa.30_15;
  vect_pa.38_20 = (vector int *) &a;
  vect_pa.33_21 = vect_pa.38_20;

<bb 3>:
  vect_var_.32_19 = *vect_pa.25_17;
  *vect_pa.33_22 = vect_var_.32_19;
  vect_pa.25_18 = vect_pa.25_17 + 16;
  vect_pa.33_23 = vect_pa.33_22 + 16;
  ivtmp.39_25 = ivtmp.39_24 + 1;
  if (ivtmp.39_25 < 64)
    goto <bb 4>;
```

Example 3: Vectorization but No Parallelization

- Step 3: Observing the conclusion about dependence information

```
inner loop index: 0
loop nest: (1 )
distance_vector:   4
direction_vector:     +
```

- Step 4: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```


Example 4: No Vectorization and No Parallelization

Step 0: Compiling with `-fno-predictive-commoning`

```
int a[256], b[256];
int main ()
{
    int i;
    for (i=0; i<256; i++)
    {
        a[i+2] = b[i] + 5;
        b[i+3] = a[i] + 10;
    }
    return 0;
}
```

- Additional options for parallelization
  `-ftree-parallelize-loops=4 -fdump-tree-parloops-all`
- Additional options for vectorization
  `-fdump-tree-vect-all -msse4`

Example 4: No Vectorization and No Parallelization

- Step 1: Observing the final decision about vectorization

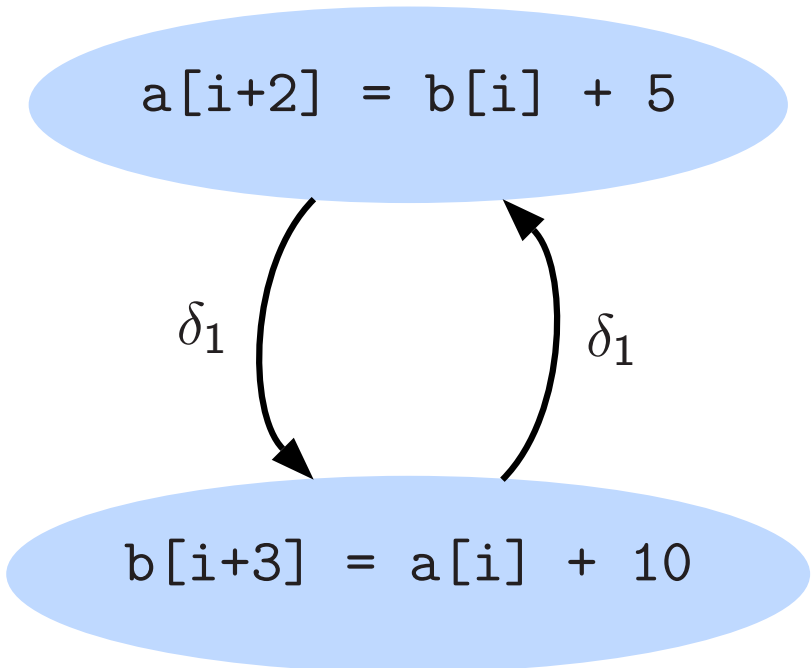
```
noparvec.c:5: note: vectorized 0 loops in function.
```

- Step 2: Observing the final decision about parallelization

```
FAILED: data dependencies exist across iterations
```


Example 4: No Vectorization and No Parallelization

Step 3: Understanding the dependencies that prohibit vectorization and parallelization



GCC Parallelization and Vectorization: Conclusions

- Chain of recurrences seems to be a useful generalization
- Data dependence information is not stored across passes
- Interaction between different transformations is not clear
  Predictive commoning and SSA seem to prohibit many opportunities
- Scalar dependences are not reported. Not clear if they are computed
- May report dependence where there is none

Other passes need to be studied to arrive at a better judgement

References


Utpal K. Banerjee.
Dependence Analysis.
Kluwer Academic Publishers, Norwell, MA, USA, 1996.


Ken Kennedy and John R. Allen.
Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

References


Olaf Bachmann, Paul S. Wang, and Eugene V. Zima.
Chains of recurrences - a method to expedite the evaluation of closed-form functions.
In *In International Symposium on Symbolic and Algebraic Computing*, pages 242–249. ACM Press, 1994.


V. Kislenkov, V. Mitrofanov, and E. Zima.
Multidimensional chains of recurrences.
In *ISSAC '98: Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, pages 199–206, New York, NY, USA, 1998. ACM.

Part 7

GCC Resource Center

National Resource Center for F/OSS, Phase II

- Sponsored by Department of Information Technology (DIT), Ministry of Information and Communication Technology
- CDAC Chennai is the coordinating agency
- Participating agencies

Organization	Focus
CDAC Chennai	SaaS Model, Mobile Internet Devices on BOSS, BOSS applications
CDAC Mumbai	FOSS Knowledge Base, FOSS Desktops
CDAC Hyderabad	E-Learning
IIT Bombay	Gnu Compiler Collection
IIT Madras	OO Linux kernel
Anna University	FOSS HRD

Objectives of GCC Resource Center

1. To support the open source movement
   Providing training and technical know-how of the GCC framework to academia and industry.
2. To include better technologies in GCC
   Whole program optimization, Optimizer generation, Tree tiling based instruction selection.
3. To facilitate easier and better quality deployments/enhancements of GCC
   Restructuring GCC and devising methodologies for systematic construction of machine descriptions in GCC.
4. To bridge the gap between academic research and practical implementation
   Designing suitable abstractions of GCC architecture

Broad Research Goals of GCC Resource Center

- Using GCC as a means
  - Adding new optimizations to GCC
  - Adding flow and context sensitive whole program analyses to GCC (In particular, pointer analysis)
- Using GCC as an end in itself
  - Changing the retargetability mechanism of GCC
  - Cleaning up the machine descriptions of GCC
  - Facilitating specification driven optimizations
  - Improving vectorization/parallelization in GCC

GRC Training Programs

Title	Target	Objectives	Mode	Duration
Workshop on Essential Abstractions in GCC	People interested in deploying or enhancing GCC	Explaining the essential abstractions in GCC to ensure a quick ramp up into GCC Internals	Lectures, demonstrations, and practicals (experiements and assignments)	Three days
Tutorial on Essential Abstractions in GCC	People interested in knowing about issues in deploying or enhancing GCC	Explaining the essential abstractions in GCC to ensure a quick ramp up into GCC Internals	Lectures and demonstrations	One day
Workshop on Compiler Construction with Introduction to GCC	College teachers	Explaining the theory and practice of compiler contruction and illustrating them with the help of GCC	Lectures, demonstrations, and practicals (experiements and assignments)	Seven days
Tutorial on Demystifying GCC Compilation	Students	Explaining the translation sequence of GCC through gray box probing (i.e. by examining the dumps produced by GCC)	Lectures and demonstrations	Half day

GRC Training Programs

CS 715: The Design and Implementation of GNU Compiler Generation Framework

- 6 credits semester long course for M.Tech. (CSE) students at IIT Bombay
- Significant component of experimentation with GCC
- Introduced in 2008-2009

Progress on Research Work

- Released GDFA (Generic Data Flow Analyser)
  Currently: Intraprocedural data flow analysis for any bit vector framework
- Identified exact changes required in the machine descriptions and instruction selection mechanism
  New constructs in machine descriptions
  May reduce the size of specifications by about 50%

July 2009 Workshop

July 2009 Workshop

- Conduct of the workshop:
  - Number of lecture sessions: 12
  - Number of lab sessions: 7
  - Assignments were done in groups of 2
  - Number of lab TAs: 15
- Participants:
  - External Participants: 60
  - Participants from private industry: 34
    (KPIT, ACME, HP, Siemens, Google, Bombardier, Morgan Stanley, AMD, Merceworld, Selec, Synopsys)
  - Participants from academia: 16
  - Participants from govt. research org.: 10
    (NPCIL, VSSC)

Part 8

Conclusions

Conclusions

GCC is a strange paradox

- Practically very successful
  - Readily available without any restrictions
  - Easy to use
  - Easy to examine compilation without knowing internals
  - Available on a wide variety of processors and operating systems
  - Can be retargeted to new processors and operating systems

Conclusions

GCC is a strange paradox

- Practically very successful
  - Readily available without any restrictions
  - Easy to use
  - Easy to examine compilation without knowing internals
  - Available on a wide variety of processors and operating systems
  - Can be retargeted to new processors and operating systems
- Quite adhoc

Conclusions

GCC is a strange paradox

- Practically very successful
  - Readily available without any restrictions
  - Easy to use
  - Easy to examine compilation without knowing internals
  - Available on a wide variety of processors and operating systems
  - Can be retargeted to new processors and operating systems
- Quite adhoc
  - Needs significant improvements in terms of design
    Machine description specification, IRs, optimizer generation

Conclusions

GCC is a strange paradox

- Practically very successful
  - Readily available without any restrictions
  - Easy to use
  - Easy to examine compilation without knowing internals
  - Available on a wide variety of processors and operating systems
  - Can be retargeted to new processors and operating systems
- Quite adhoc
  - Needs significant improvements in terms of design
    Machine description specification, IRs, optimizer generation
  - Needs significant improvements in terms of better algorithms
    Retargetability mechanism, interprocedural optimizations, parallelization, vectorization,

Conclusions

- The availability of source code and the availability of dumps makes GCC a very useful case study for compiler researchers

Conclusions

GCC Resource Center at IIT Bombay

- Our Goals
  - Demystifying GCC
  - A dream to improve GCC
  - Spreading GCC know-how
- Our Strength
  - Synergy from group activities
  - Long term commitment to challenging research problems
  - A desire to explore real issues in real compilers
- On the horizon
  - Enhancements to data flow analyser
  - Overall re-design of instruction selection mechanism

Last but not the least ...

Thank You!