GCC – Code Generation

Matthias Hannen
Vladimir Parashin

Overview

1. Intoduction
2. GCC Generation Architecture
3. GIMPLE
4. Register Transfer Language (RTL)
5. Summary

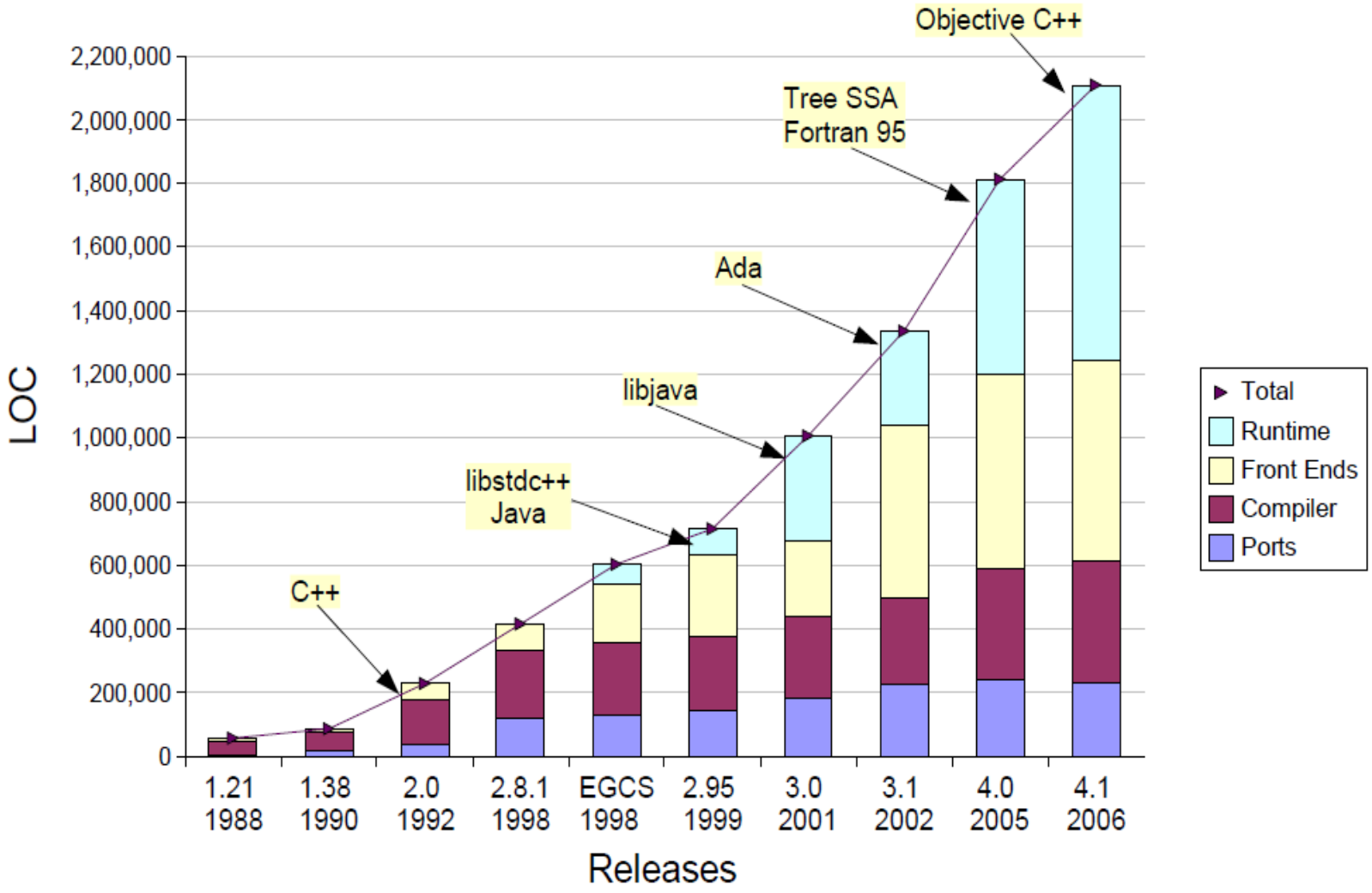
Intoduction

GNU Compiler Collection(GCC)

- Inspired on Pastel Compiler
- Cross compiler

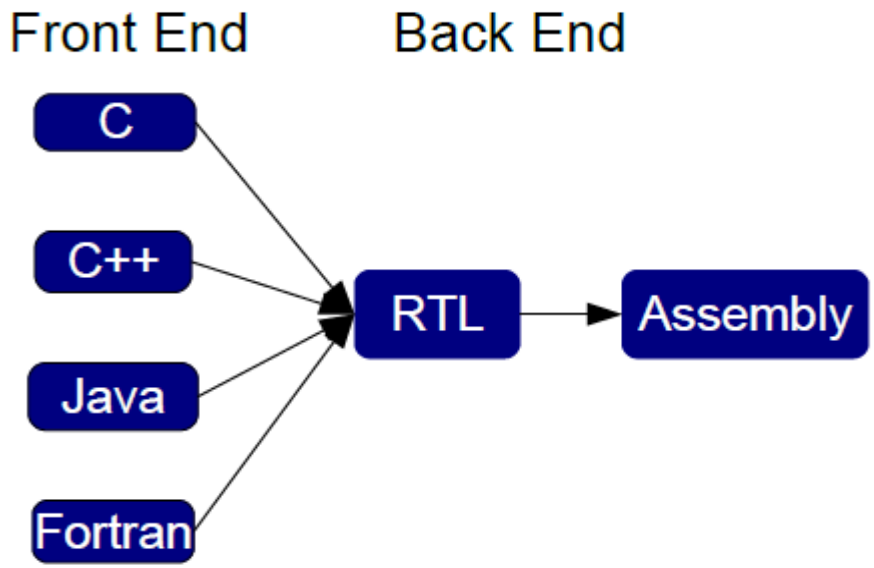
- GCC 1(1987) – only for C
- GCC 2(1992) – RISC architecture support
- GCC 3(2001) – Trees
- GCC 4(2005) – internals overhaul (Tree SSA)

Intoduction



Intoduction

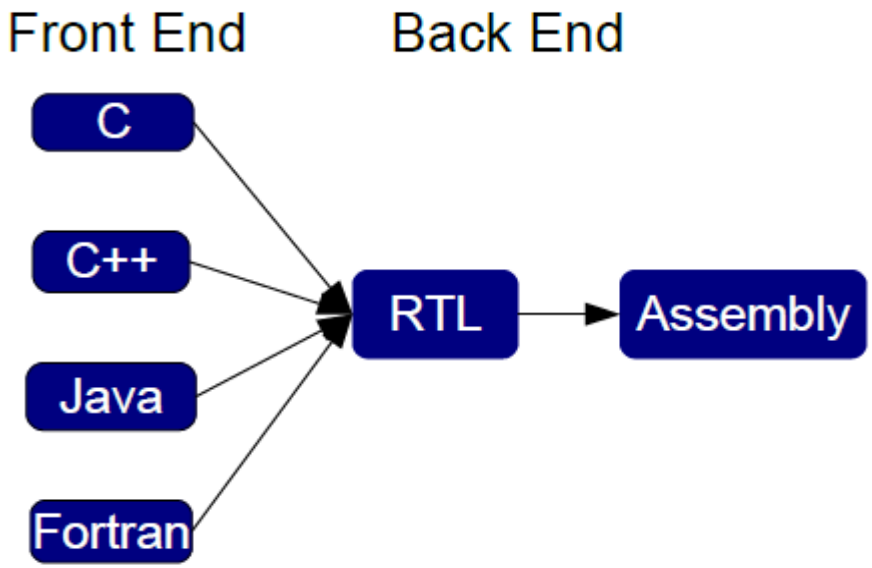
Internals till 2005:



Why change Architecture?

Intoduction

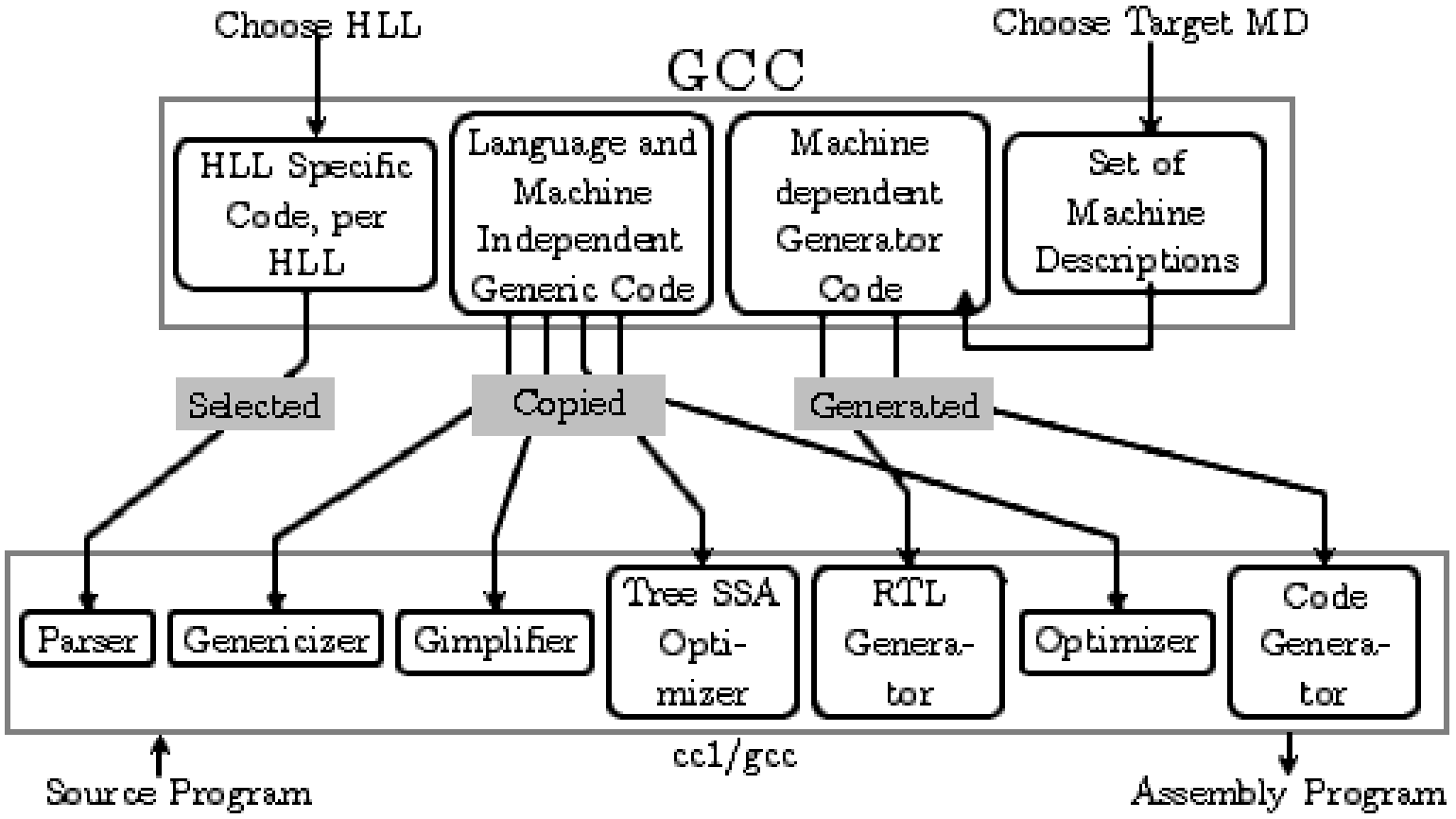
Internals till 2005:



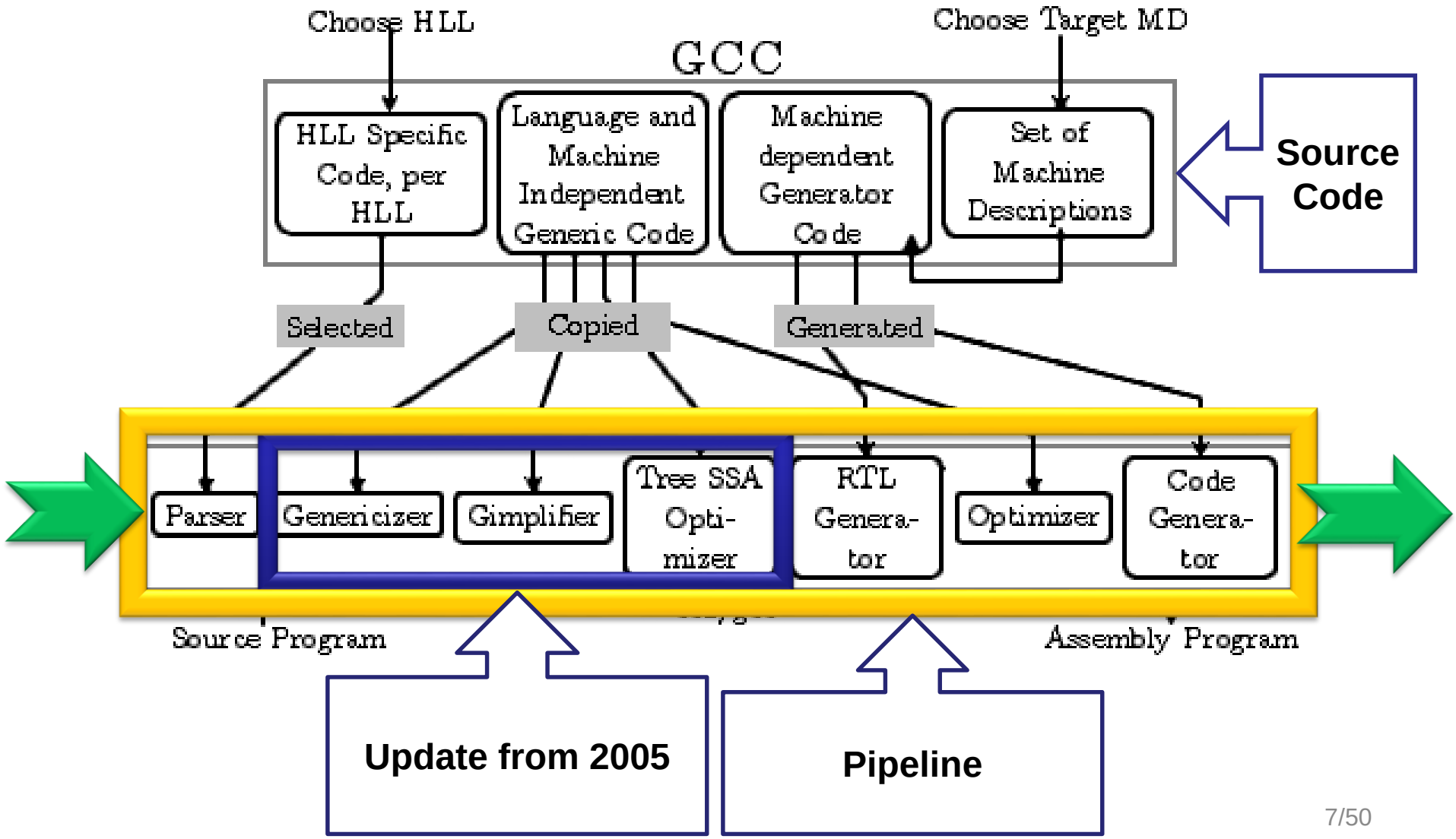
Why change Architecture?

- FE close to BE
- common representation
- structural complexity

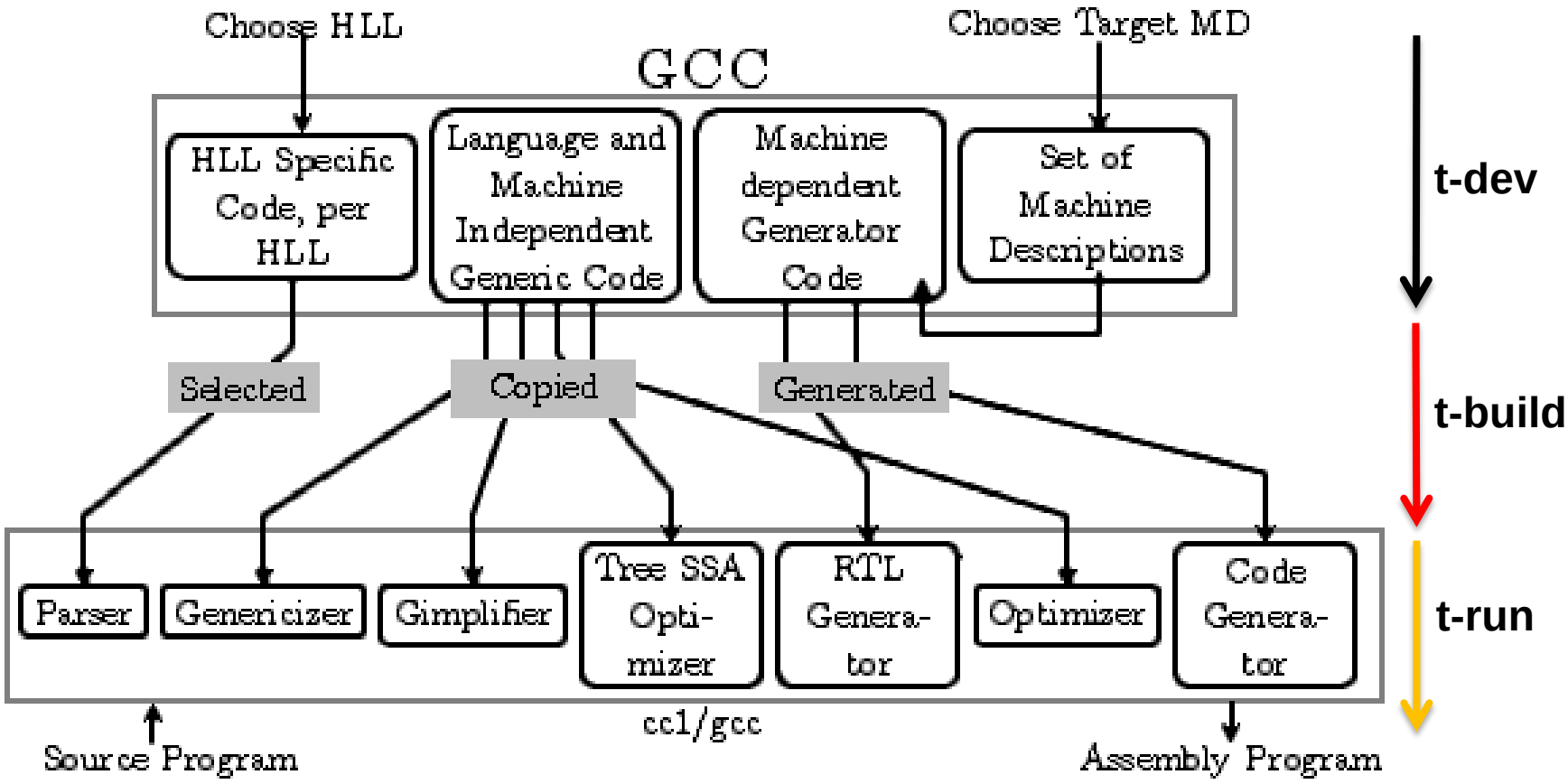
GCC Generation Architecture



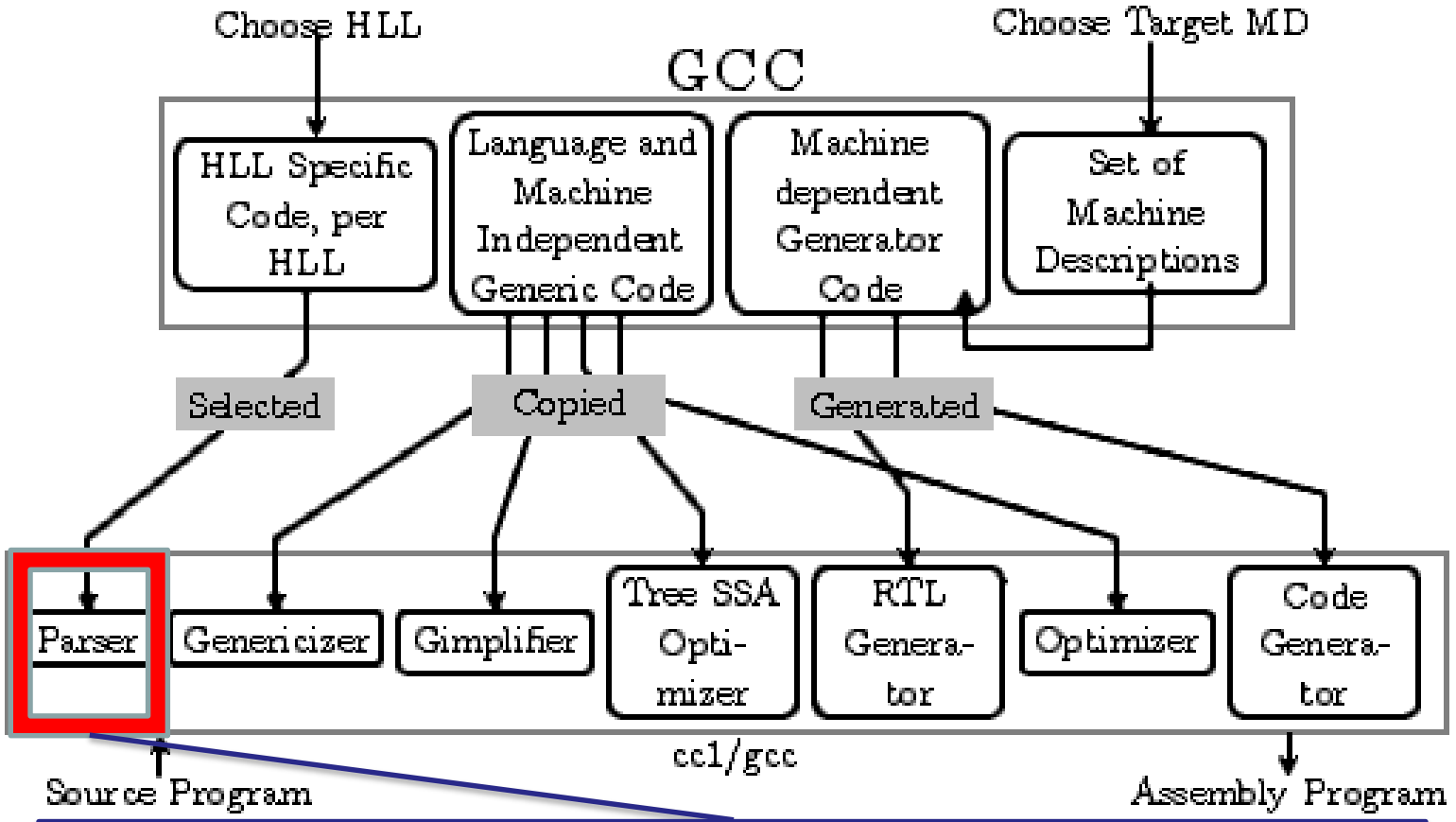
GCC Generation Architecture



GCC Generation Architecture



GCC Generation Architecture

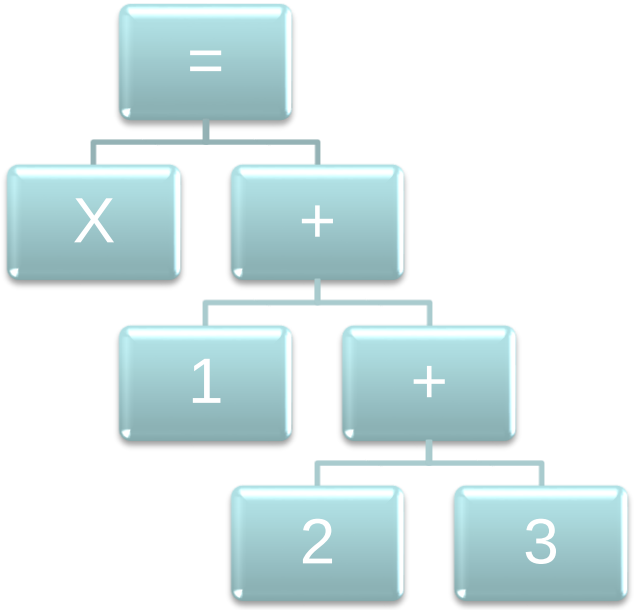


- **syntax analyses**
- **procedural abstractions**
- **returns Abstract Syntax Tree(AST)**

GCC Generation Architecture

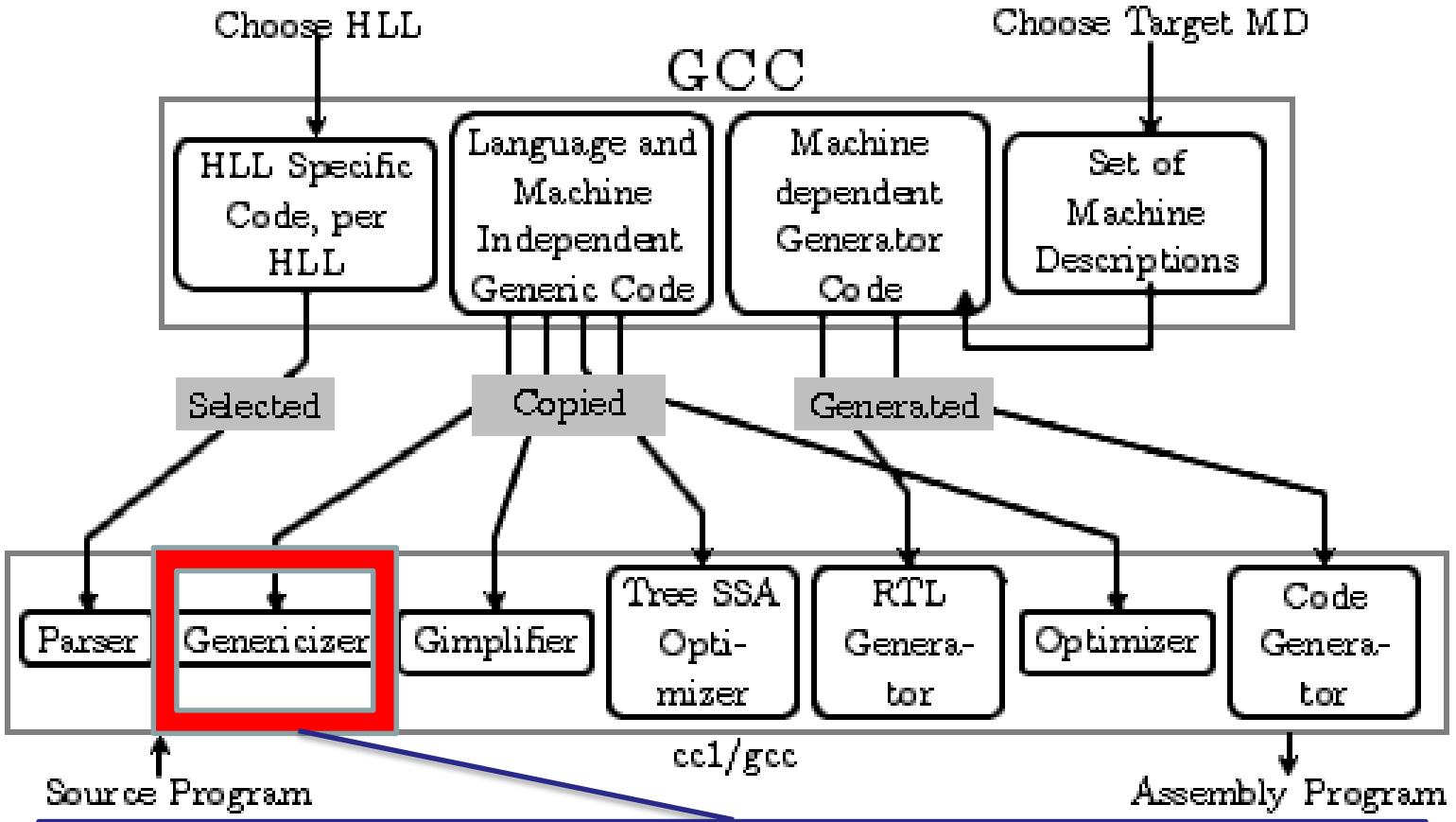
Example:

x = 1 + 2 + 3



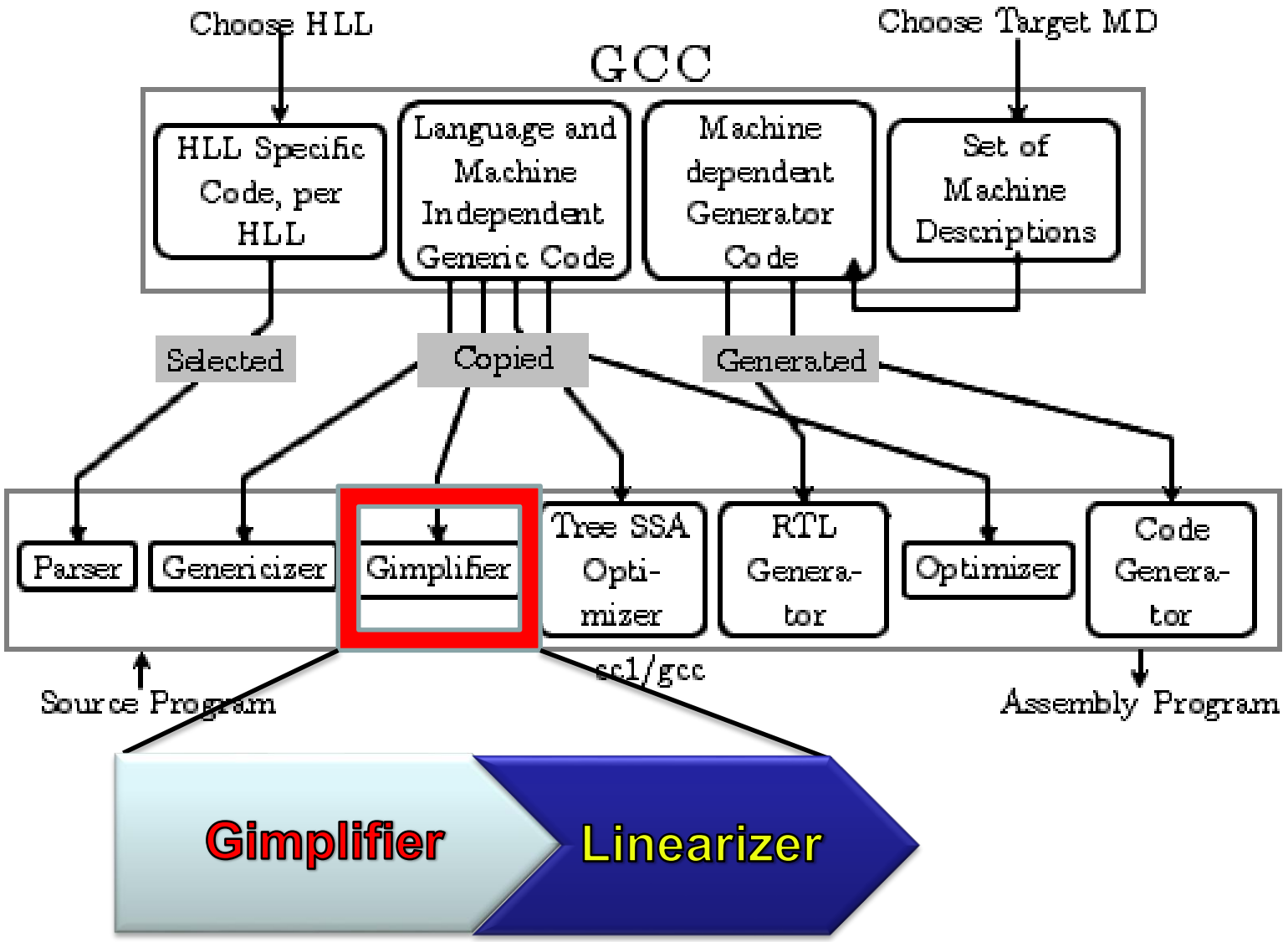
tree var_decl = build_decl (UNKNOWN_LOCATION, VAR_DECL,get_identifer("x"),…);
tree expr_tmp = build2 (PLUS_EXPR, …, (…, 2), …(integer_type_node, 3));
tree expr = build2(PLUS_EXPR, integer_type_node, build_int_cst(integer_type_node, 1),);
tree eval = build2 (MODIFY_EXPR, integer_type_node, var_decl, expr);

GCC Generation Architecture



- **removing HLL dependencies**
- **returns a common Tree representation**
- **name space abstractions**

GCC Generation Architecture



GIMPLE

Why?

- Before: only common intermediate representation (IR) was RTL
- Drawbacks of RTL:
  - Low-level IR, which works for optimizations close to machine (e.g. register allocation)
  - Some high level information is difficult to extract from RTL (e.g. data types)
  - Introduces stack too soon, even if later optimizations do not demand it

GIMPLE

Goals:

- Lower control flow
  - Only sequence statements and unrestricted jumps
- Simplify expressions
  - Typically two operand assignments
- Simplify scope
  - Move local scope to block begin (including temporaries)

Result:

closer to register machines & easier SSA!

GIMPLE

Constraints:

- Max 1 load & store operation per statement
- 3-address representation
- Only single-level pointer
- Real(virtual) operands
- Loops with if/goto

GIMPLE

Other facts:

- Derived from SIMPLE(IR of McCAT compiler)
- control flow abstractions
- Represent alias information explicit
- No hidden side-effects
- Extension to OpenMP

GIMPLE

A little example:

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

GIMPLE

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

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

GIMPLE

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

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

GIMPLE

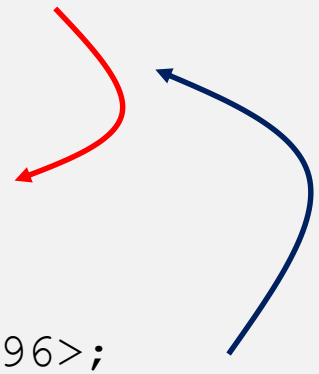
```
int main(void)
{
        int a=2, b=3, c=4;
        while (a<=7)
        {
                a = a+1;
        }
        if (a<=12)
                a = a+b+c;
}
```

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

GIMPLE

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

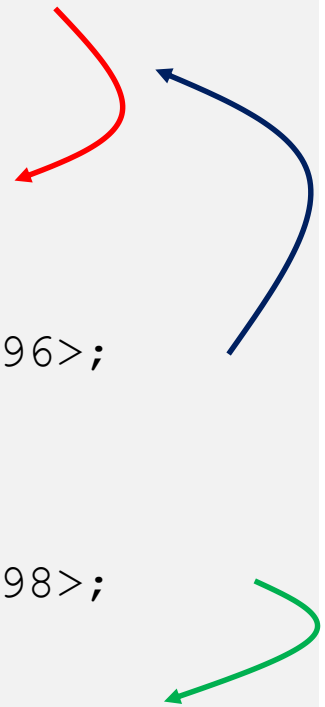
```
goto <D.1197>;
<D.1196>:;
a = a+1;
<D.1197>:;
if (a<=7)
{
        goto <D.1196>;
}
else
{
        goto <D.1198>;
}
<D.1198>:;
```



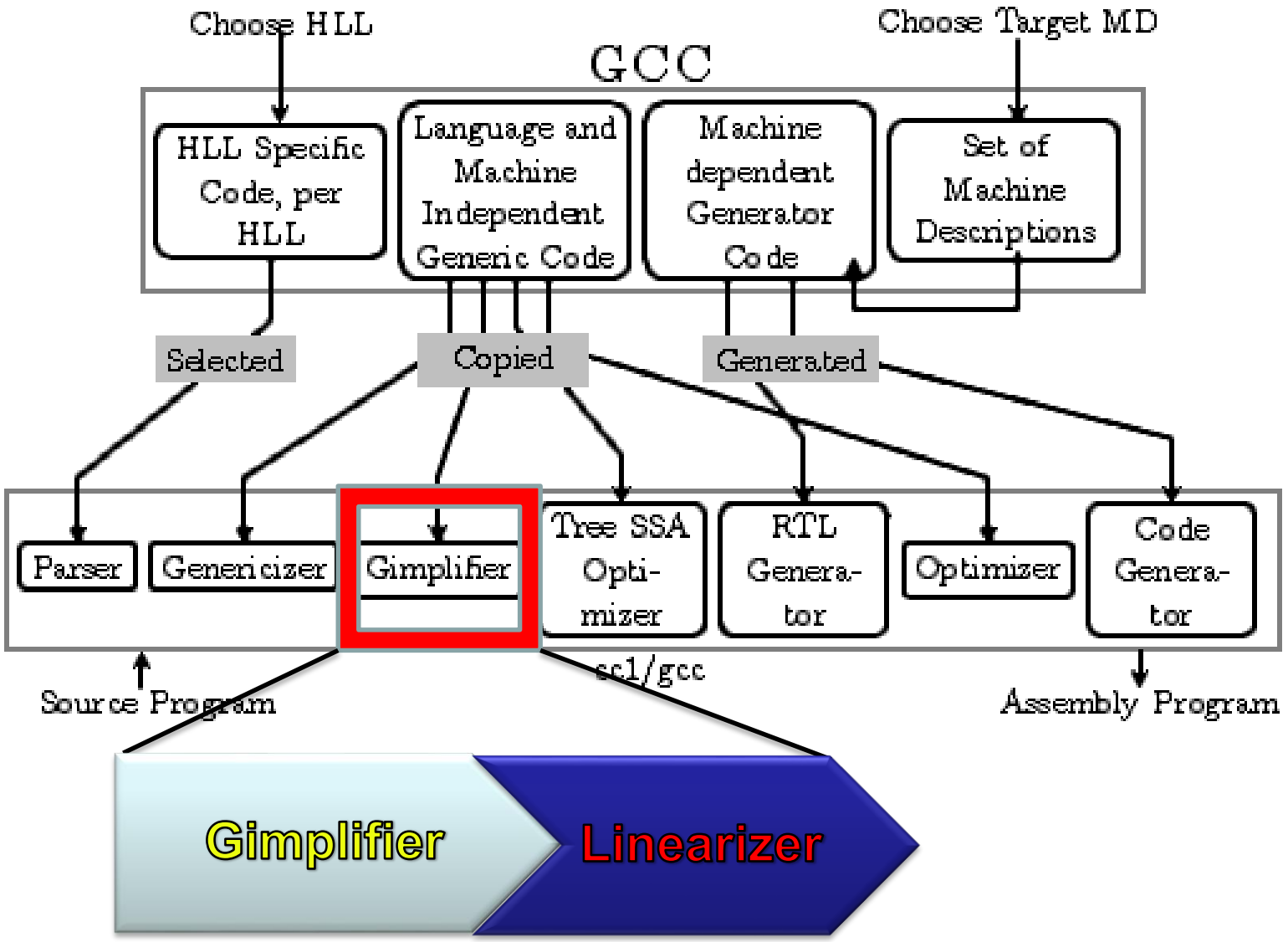
GIMPLE

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

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



GCC Generation Architecture



Lowering GIMPLE

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

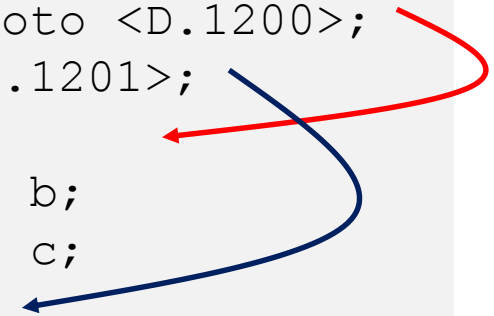
```
if (a<=12) goto <D.1200>;
else goto <D.1201>;
<D.1200>:;
D.1199 = a + b;
a = D.1199 + c;
<D.1201>:;
return;
```

if translated in terms of conditional and unconditional gotos

Lowering GIMPLE

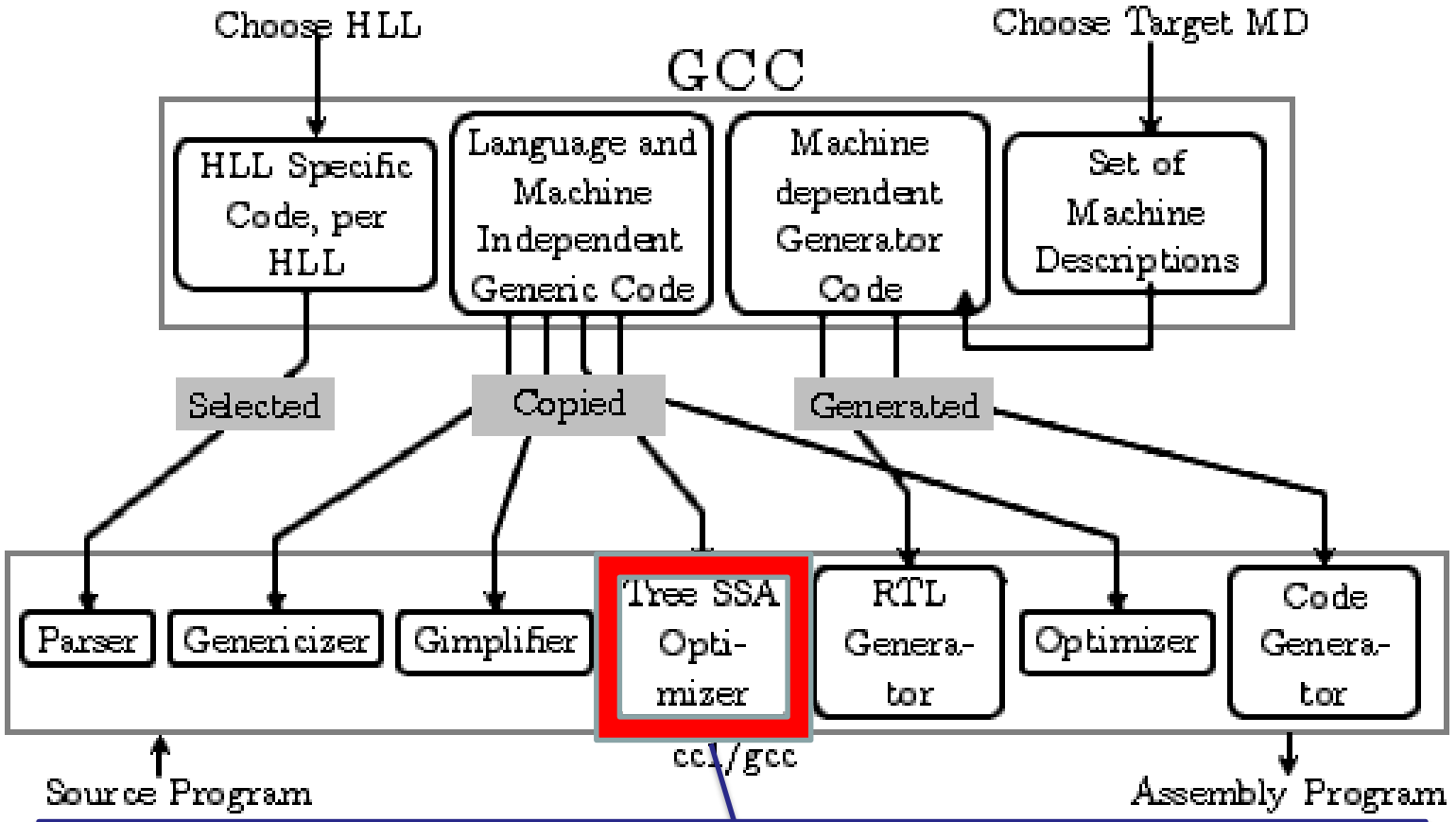
```
if (a<=12)
{
        D.1199 = a+b;
        a = D.1199 + c;
}
else
{
}
```

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



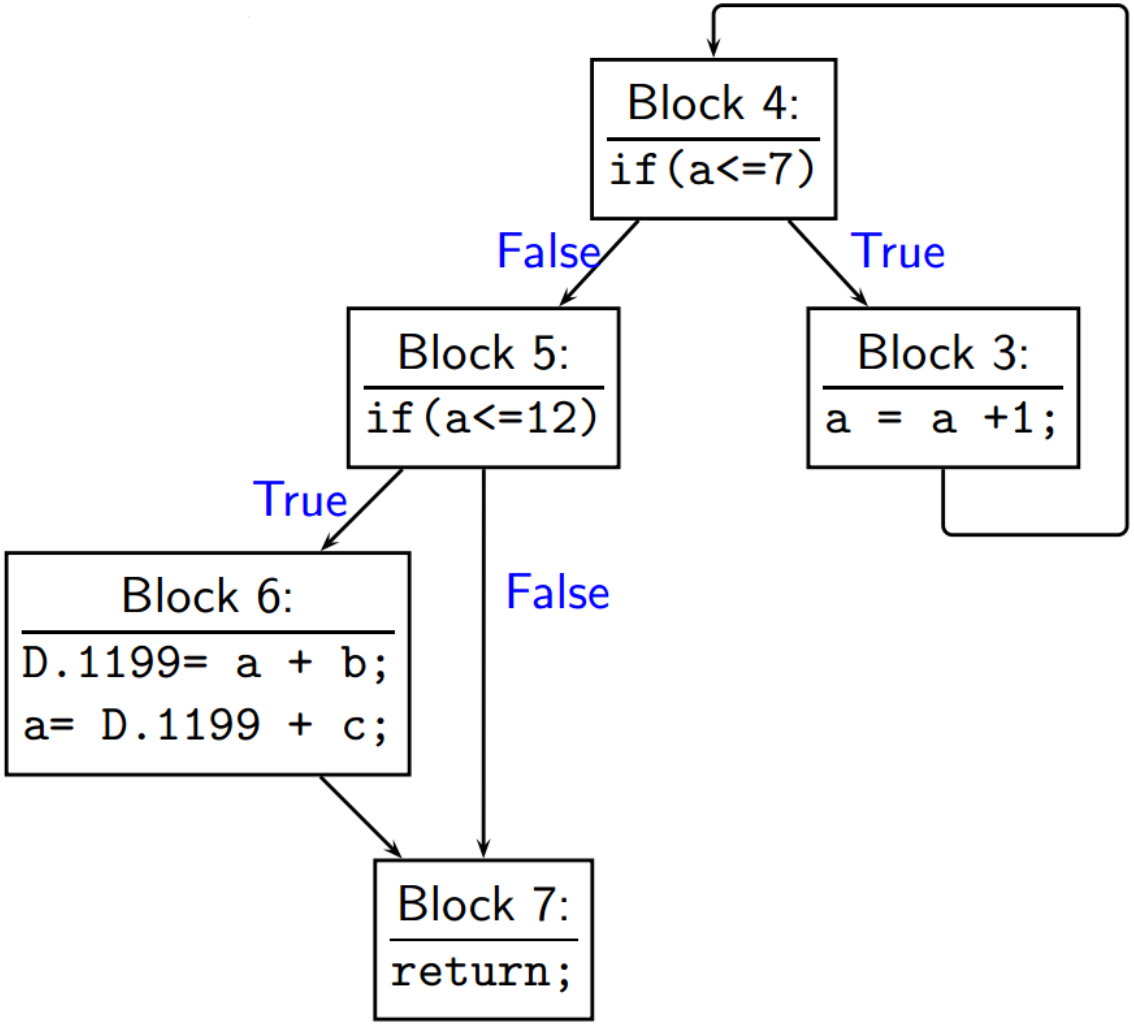
if translated in terms of conditional and unconditional gotos

GCC Generation Architecture

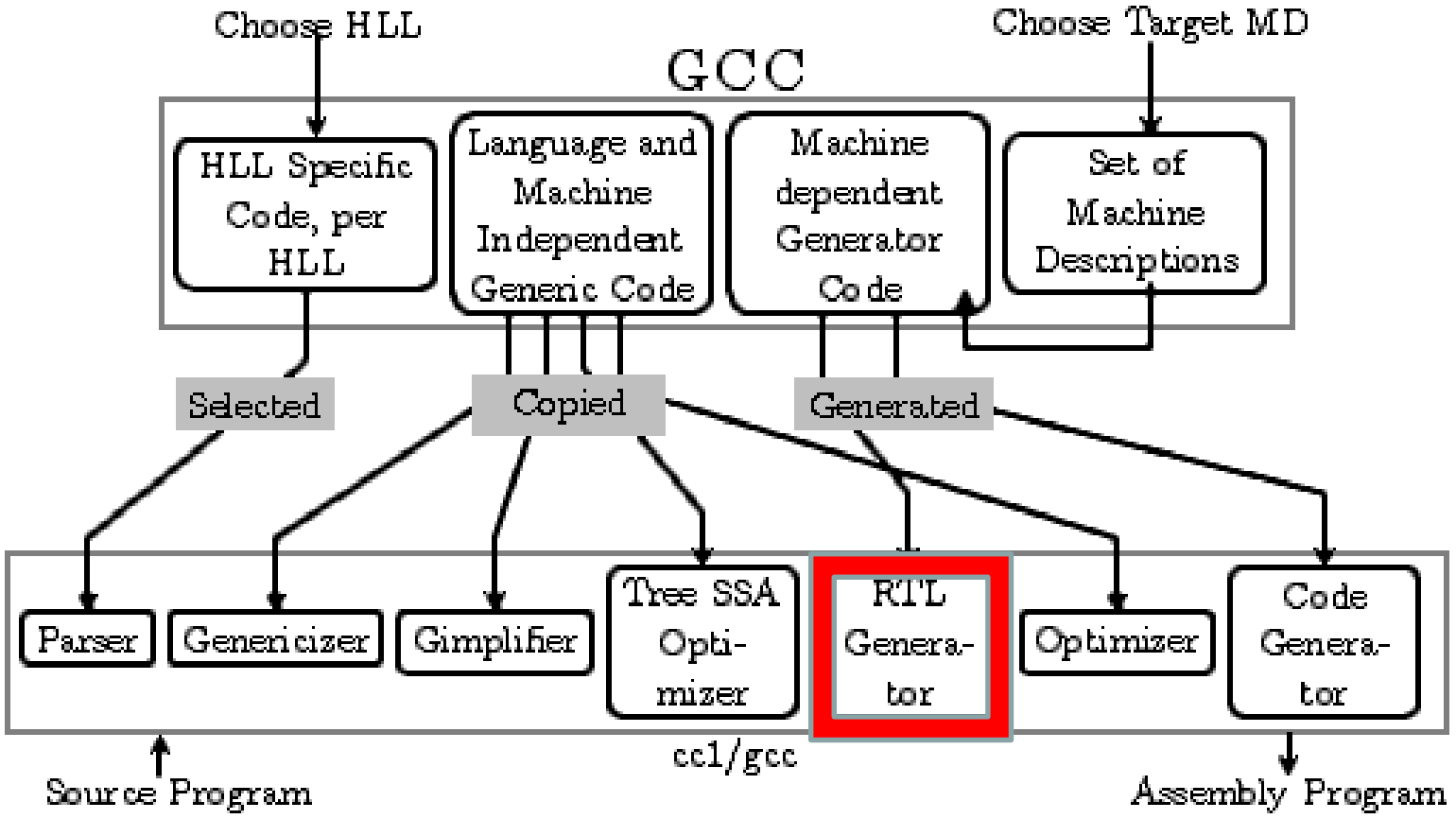


- **Static Single Assignment (SSA)**
- **Explicit shows flow of data**
- **GIMPLE → Control Flow Graph → RTL**

Control Flow Graph



GCC Generation Architecture



RTL

- Register Transfer Language
  - Assembly language for an abstract machine with infinite registers
- Abstract out essential characteristics of typical hardware
- Two purposes:
  - Express target properties
  - Represent compilation of program

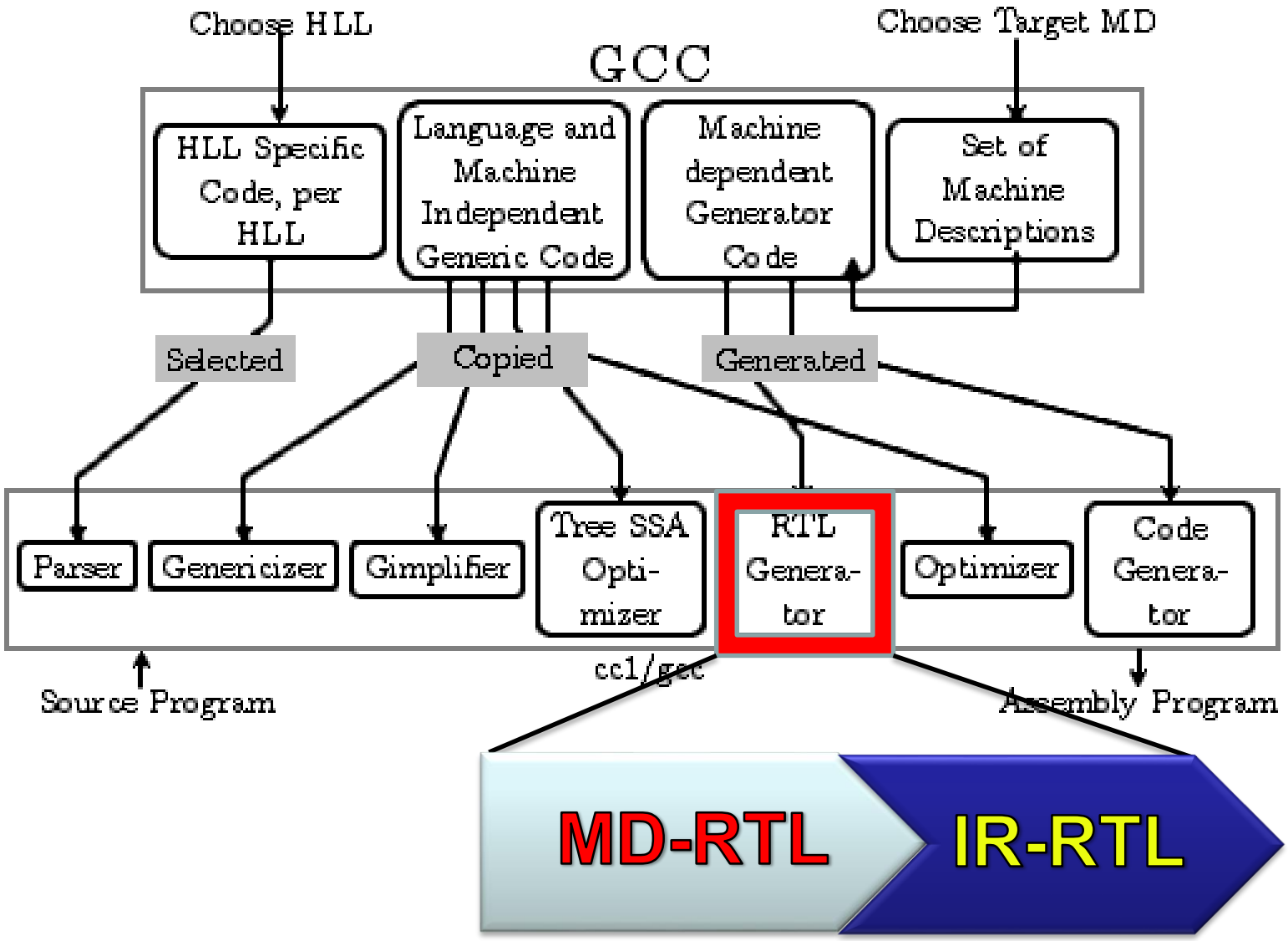
RTL

Two languages:

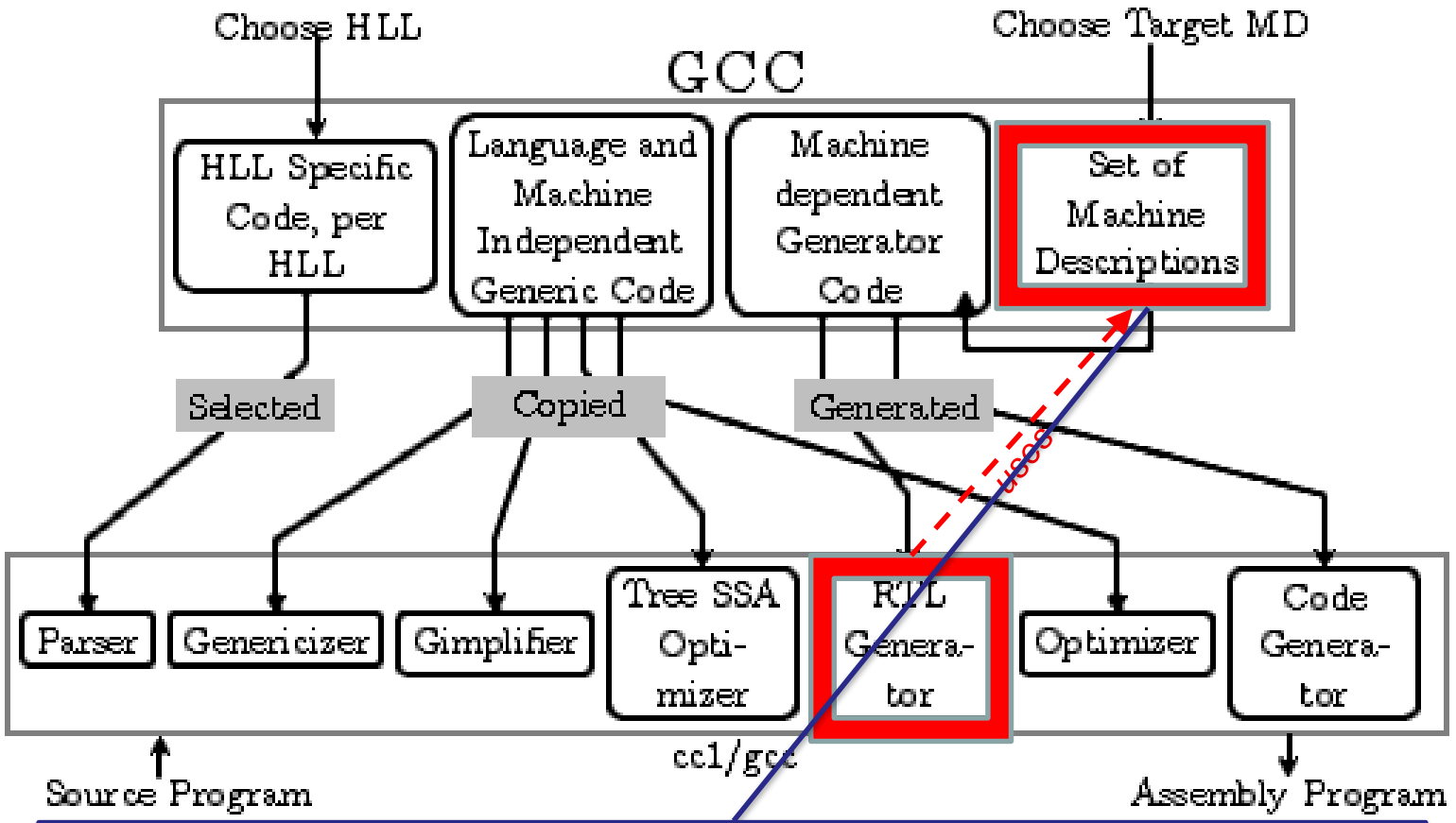
- MD-RTL
  - Using machine descriptions (MD)
  - To specify the semantics of target instructions
  - Human readable
  - Generated from the Control Flow Graph

- IR-RTL
  - Intermediate representation (IR) very close to assembly language
  - To express a program being compiled
  - Machine readable at runtime
  - Result of compiling a C representation that is generated from the MD-RTL specifications

GCC Generation Architecture



GCC Generation Architecture



- **Instructions for generation of assembly code from RTL**
- **Depend only on target machine**

A Target Instruction in Machine Descriptions

```
(define_insn
        "movsi"
        (set
                (match_operand 0 "register_operand" "r")
                (match_operand 1 "const_int_operand" "k")
        )
        "" /* C Boolean expression, if required */
        "li %0, %1"
)
```

A Target Instruction in Machine Descriptions

Define instruction pattern

```
(define_insn
        "movsi"
        (set
                (match_operand 0 "register_operand" "r")
                (match_operand 1 "const_int_operand" "k")
        )
        "" /* C Boolean expression, if required */
        "li %0, %1"
)
```

A Target Instruction in Machine Descriptions

Define instruction pattern

Standard Pattern Name (optional)

```
(define_insn
        "movsi"
        (set
                (match_operand 0 "register_operand" "r")
                (match_operand 1 "const_int_operand" "k")
        )
        "" /* C Boolean expression, if required */
        "li %0, %1"
)
```

A Target Instruction in Machine Descriptions

Define instruction pattern

Standard Pattern Name (optional)

```
(define_insn
        "movsi"
        (set
                (match_operand 0 "register_operand" "r")
                (match_operand 1 "const_int_operand" "k")
        )
        "" /* C Boolean expression, if required */
        "li %0, %1"
)
```

RTL Expression (RTX):
Semantics of target instruction

A Target Instruction in Machine Descriptions

Define instruction pattern

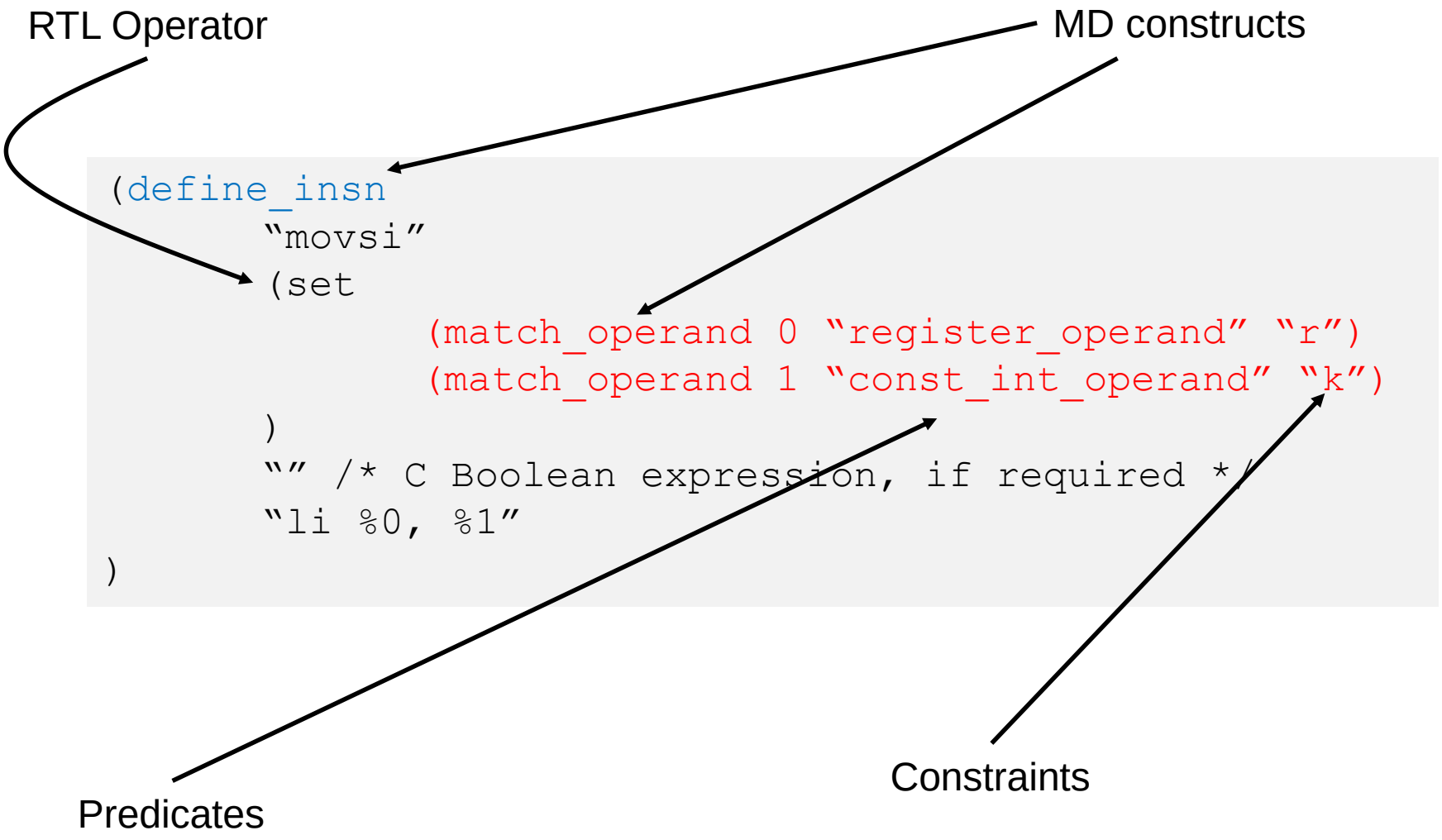
Standard Pattern Name (optional)

```
(define_insn
        "movsi"
        (set
                (match_operand 0 "register_operand" "r")
                (match_operand 1 "const_int_operand" "k")
        )
        "" /* C Boolean expression, if required */
        "li %0, %1"
)
```

Target asm instruction:
Concrete syntax for RTX

RTL Expression (RTX):
Semantics of target instruction

A Target Instruction in Machine Descriptions

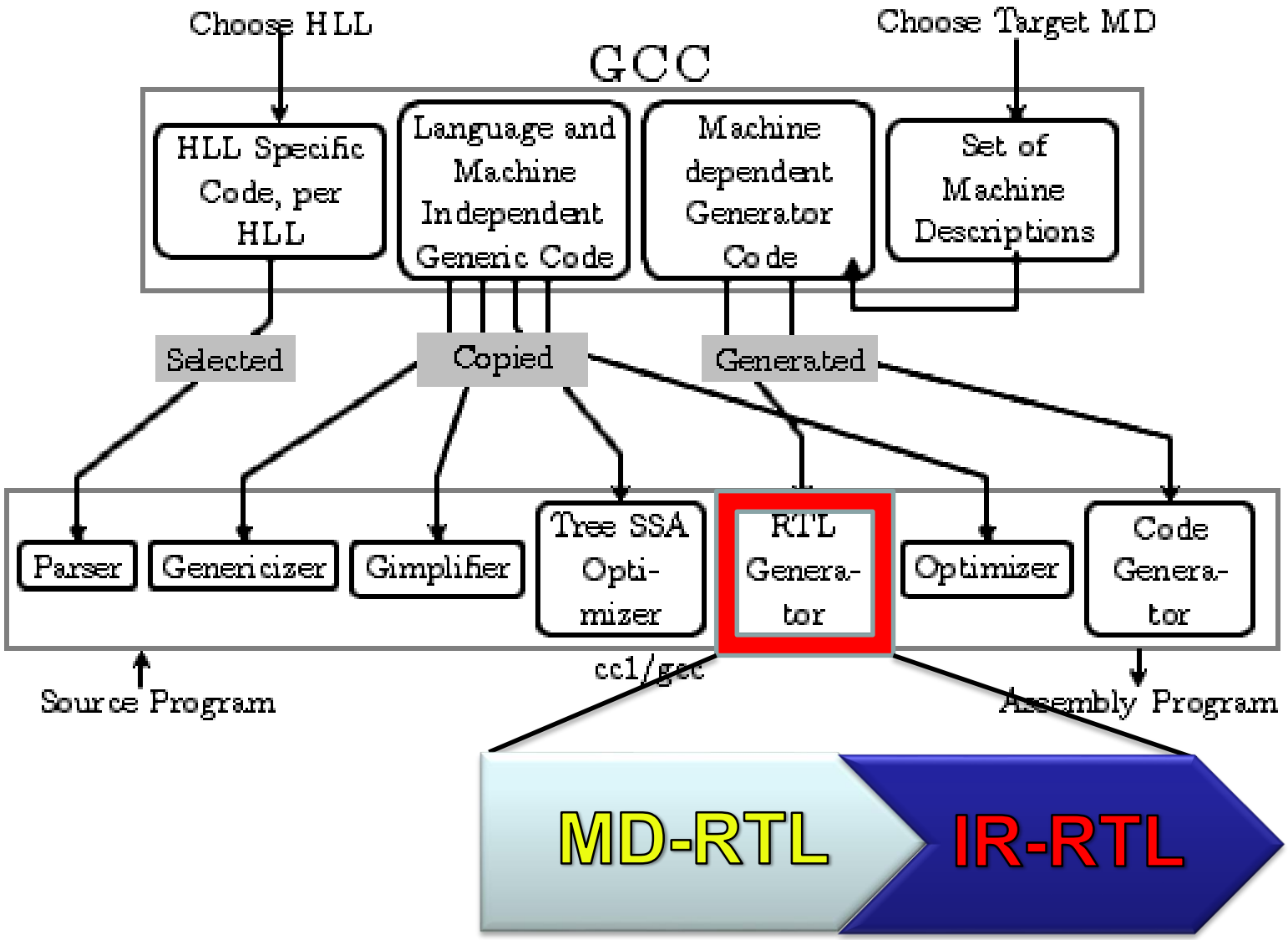


A Target Instruction in Machine Descriptions

```
(define_insn
        “movsi”
        (set
                (match_operand 0 “register_operand” “r”)
                (match_operand 1 “const_int_operand” “k”)
        )
        “” /* C Boolean expression, if required */
        “li %0, %1”
)
```

```
D.1713 = 42;
```
→
```
(set
        (reg:SI 58 [D.1713])
        (const_int 42: [0xa])
)
```
→
```
li $t0, 42
```

GCC Generation Architecture



IR-RTL for i386

Translation of a = a + 1: `stack($fp – 4) = stack($fp – 4) + 1 || flags=?`

IR-RTL for i386

Translation of a = a + 1: stack($fp – 4) = stack($fp – 4) + 1 || flags=?

```
(insn 12 11 10 (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
                                (ref/f:SI 54 virtual-stack-vars)
                                (const int -4 […])) […])
                        (const int 1 […])))
        (clobber (ref:CC 17 flags))
]) -1 (nil))
```

IR-RTL for i386

Translation of a = a + 1: stack($fp – 4) = stack($fp – 4) + 1 || flags=?

```
(insn 12 11 10 (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
                                (ref/f:SI 54 virtual-stack-vars)
                                (const int -4 […])) […])
                        (const int 1 […])))
        (clobber (ref:CC 17 flags))
]) -1 (nil))
```

IR-RTL for i386

Translation of a = a + 1: stack($fp – 4) = stack($fp – 4) + 1 || flags=?

```
(insn 12 11 10 (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
                                (ref/f:SI 54 virtual-stack-vars)
                                (const int -4 […])) […])
                        (const int 1 […])))
        (clobber (ref:CC 17 flags))
]) -1 (nil))
```

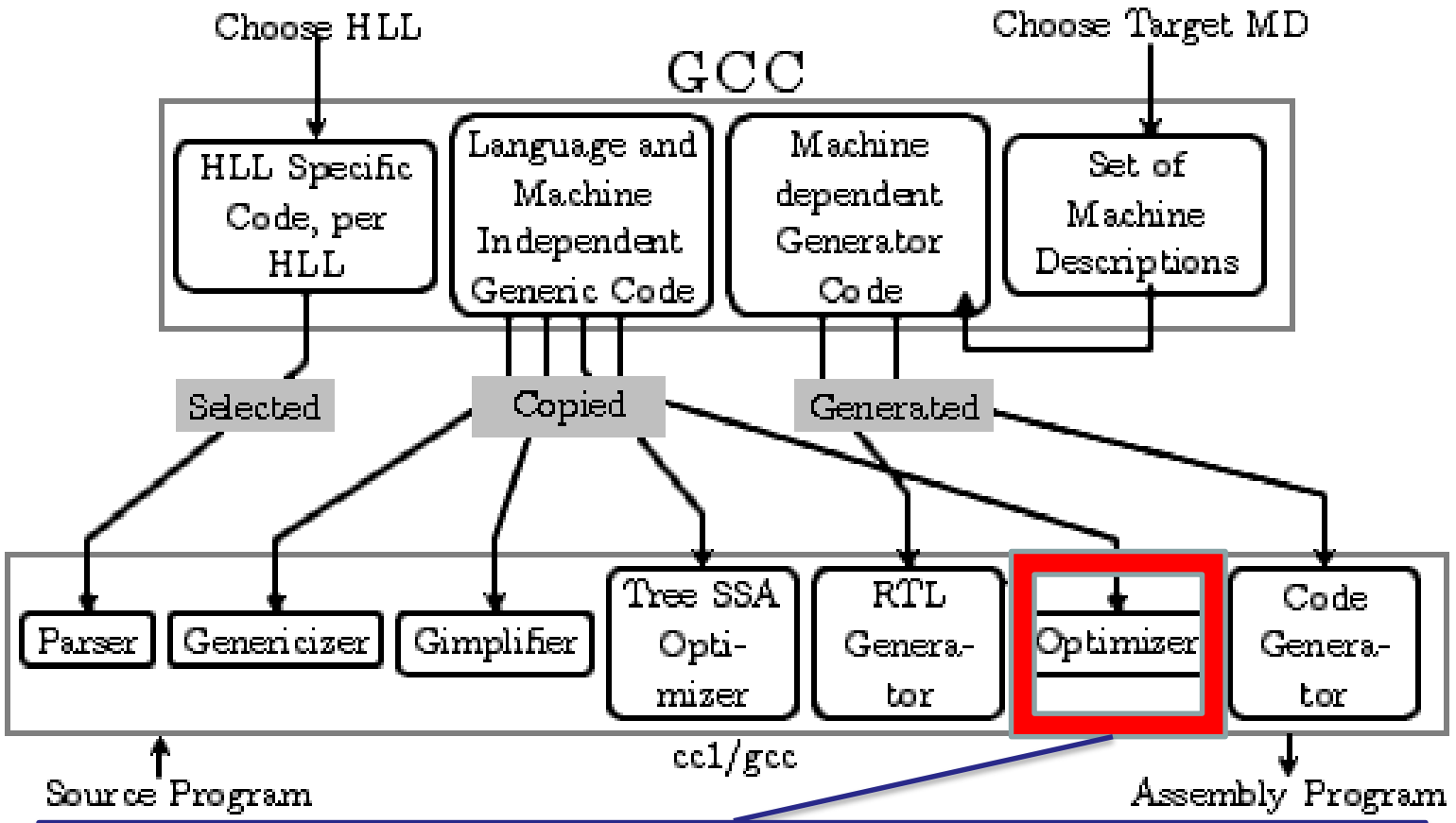
IR-RTL for i386

Translation of a = a + 1: stack($fp – 4) = stack($fp – 4) + 1 || flags=?

```
(insn 12 11 10 (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
                                (ref/f:SI 54 virtual-stack-vars)
                                (const int -4 […])) […])
                        (const int 1 […])))
        (clobber (ref:CC 17 flags))
]) -1 (nil))
```

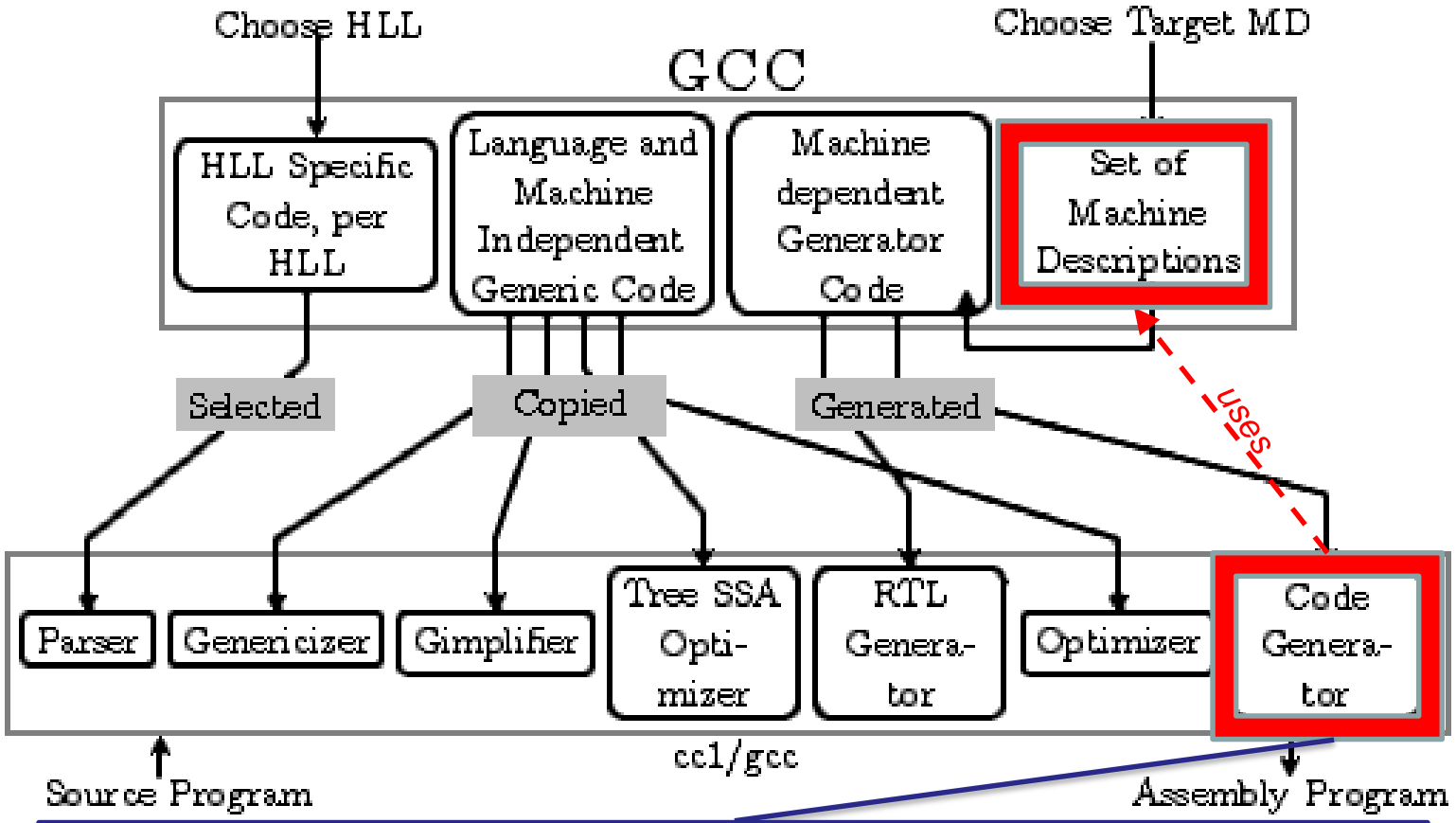
Condition Code register is clobbered to record possible side effect of plus

GCC Generation Architecture



- **Target machine specific optimizations**
- **Low-level transformations**
- **Peephole optimizations**

GCC Generation Architecture

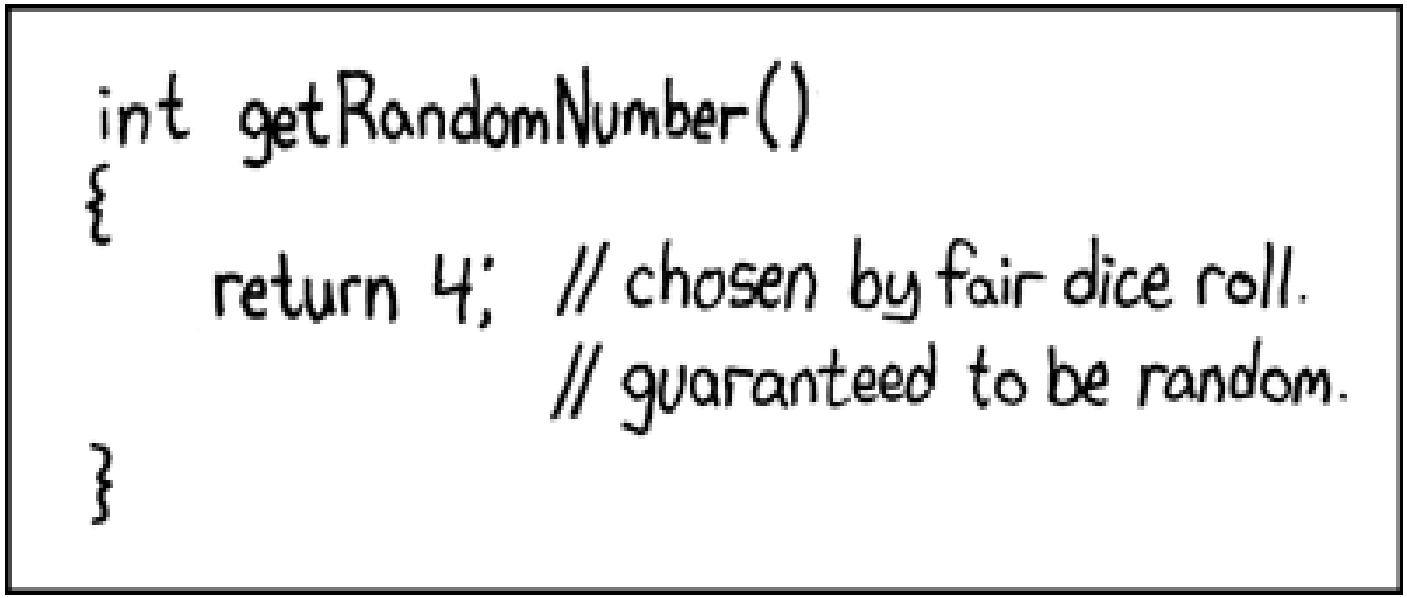


- **Use MDs for generation of assembly code from RTL**

Summary

- The GCC Generation Architecture is huge!
  - Machine Descriptions: ~400,000 LOC
  - Distributed front ends: ~350,000 LOC
  - Base compiler: ~ 325,000 LOC
  - Build GCC: ~ 1,100,000 LOC
- GIMPLE very important
  - Reduces code complexity (3-address representation)
  - Simplifies code analysis and optimizations like Vectorization (not even possible before)
  - Target *independent*!
- RTL cannot be replaced
  - Needed for low level optimizations like Peephole
  - Target *dependent*!
- Thank god it is generated!
  - Very small examples show high complexity

Thank You for your Attention



Reference

- Publications by Diego Novillo
  - http://www.airs.com/dnovillo/Papers/hipeac2007.pdf
- Redhat – Magazine (Issue #2 December 2004)
  - http://www.redhat.com/magazine/002dec04/features/gcc
- Department of Computer Science and Engineering. Indian Institute of Technology Bombay(CSE)
  - http://www.cse.iitb.ac.in/grc/intdocs/gcc-conceptual-structure.html
  - http://www.cse.iitb.ac.in/~uday/courses/cs715-09/gcc-gimple.pdf
  - http://www.cse.iitb.ac.in/~uday/courses/cs715-09/gcc-rtl.pdf

(all references last accessed on 7/20/2014)

- GNU Tools Cauldron Conference
  - https://gcc.gnu.org/wiki/cauldron2014