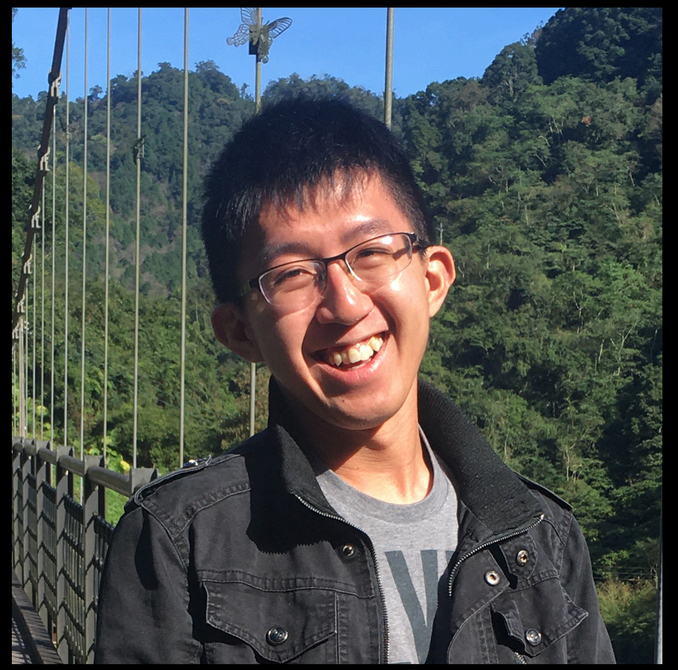
Handling inline assembly in Clang and LLVM

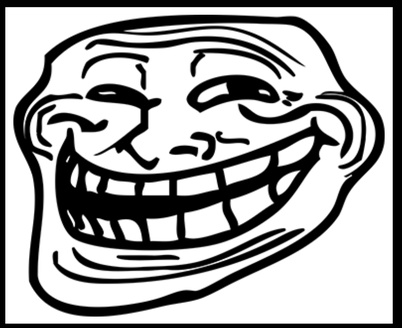
Min-Yih “Min” Hsu @ LLVM Dev Meeting 2021

about:me

“Min” Hsu

- Computer Science PhD Candidate in University of California, Irvine
- Code owner of M68k LLVM backend
- Author of book “*LLVM Techniques, Tips and Best Practices*” (2021)





How Inline Assembly is Processed in Clang & LLVM

Inline Assembly

Introduction to inline assembly

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)


  else
    sum += 87;
}
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

```
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    testb   $1, -9(%rbp)
    je  LBB0_2

    ## InlineAsm Start
    movl    %eax, %ebx
    addl    %ebx, %esi
    ## InlineAsm End

    jmp LBB0_3
LBB0_2:
    movl    -8(%rbp), %eax
    addl    $87, %eax
    movl    %eax, -8(%rbp)
LBB0_3:
    popq    %rbp
    retq
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

```
foo:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    testb   $1, -9(%rbp)
    je  LBB0_2

    ## InlineAsm Start
    movl    %eax, %ebx
    addl    %ebx, %esi
    ## InlineAsm End

    jmp LBB0_3
LBB0_2:
    movl    -8(%rbp), %eax
    addl    $87, %eax
    movl    %eax, -8(%rbp)
LBB0_3:
    popq    %rbp
    retq
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Why use inline assembly?

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Why use inline assembly?

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

Performance critical code

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Why use inline assembly?

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

Performance critical code

Low-level code

e.g. Kernel, firmware

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Why use inline assembly?

```
void foo(...) {
  int sum = 0;
  bool flag = ...;
  if (flag)
    asm ("movl   %%eax, %%ebx\n"
         "addl   %%ebx, %%esi" :::);
  else
    sum += 87;
}
```

Performance critical code

Low-level code

  e.g. Kernel, firmware

Compiler optimizations “barrier”

* x86_64 assembly w/ AT&T syntax

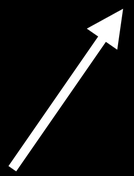
Introduction to inline assembly

```
asm (“movl   %%eax, %%ebx\n"
     "addl   %%ebx, %%esi" :::);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

```
asm (“movl   %%eax, %%ebx\n"
     "addl   %%ebx, %%esi" :::);
```

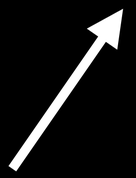


Assembly code (template)

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

```
asm (“movl   %%eax, %%ebx\n"
     "addl   %%ebx, %%esi" :::);
```



Assembly code (template)

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Output operands

```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Output operands

```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Output operands

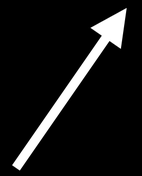
```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Output operands

```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```



Operand constraints

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Operands constraints

```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Operands constraints

```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```

```
## InlineAsm Start
    movl    %eax, %ebx
    addl    %ebx, %esi
## InlineAsm End
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Operands constraints

```
int out_var;
asm ("movl   %%eax, %%ebx\n"
     "addl   %%ebx, %0"
     : "=r"(out_var) ::);
```

```
## InlineAsm Start
    movl    %eax, %ebx
    addl    %ebx, %esi
## InlineAsm End
```

```
## InlineAsm Start
    movl    %eax, %ebx
    addl    %ebx, -8(%ebp)
## InlineAsm End
```

* x86_64 assembly w/ AT&T syntax

Operand constraints

Operand constraints

Target-independent Constraints

- ‘r’ : General-purpose register operand
- ‘i’ : Immediate integer operand
- ‘m’ : Memory operand w/ arbitrary addressing mode

Operand constraints

Target-independent Constraints

- ‘r’ : General-purpose register operand
- ‘i’ : Immediate integer operand
- ‘m’ : Memory operand w/ arbitrary addressing mode

X86 Constraints

- ‘a’ : AL / AH / EAX / RAX
- ‘I’ : Integer constant in the range of [0, 31]
- ‘N’ : Unsigned 8-bit integer constant

Operand constraints

Target-independent Constraints

- ‘r’ : General-purpose register operand
- ‘i’ : Immediate integer operand
- ‘m’ : Memory operand w/ arbitrary addressing mode

X86 Constraints

- ‘a’ : AL / AH / EAX / RAX
- ‘I’ : Integer constant in the range of [0, 31]
- ‘N’ : Unsigned 8-bit integer constant

M68k Constraints

- ‘J’ : 16-bit signed integer constant
- “Ci” : Constant integers
- “Cj” : Constant signed integers that do NOT fit in 16 bits

Operand constraints

Target-independent Constraints

- ‘r’ : General-purpose register operand
- ‘i’ : Immediate integer operand
- ‘m’ : Memory operand w/ arbitrary addressing mode

X86 Constraints

- ‘a’ : AL / AH / EAX / RAX
- ‘I’ : Integer constant in the range of [0, 31]
- ‘N’ : Unsigned 8-bit integer constant

M68k Constraints

- ‘J’ : 16-bit signed integer constant
- “Ci” : Constant integers
- “Cj” : Constant signed integers that do NOT fit in 16 bits

Constraint Modifers

- ‘=‘ : This is an output operand
- ‘+’ : This is an input / output operand

Introduction to inline assembly

Input operands

```
int out_var, in_var;
asm ("movl   %1, %%ebx\n"
     "addl   %%ebx, %0"
     : “=r"(out_var)
     : “r”(in_var) :);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Input operands

```
int out_var, in_var;
asm ("movl   %1, %%ebx\n"
     "addl   %%ebx, %0"
     : “=r"(out_var)
     : “r”(in_var) :);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Input operands

```
int out_var, in_var;
asm ("movl   %1, %%ebx\n"
     "addl   %%ebx, %0"
     : “=r"(out_var)
     : “r”(in_var) :);
```

Operand constraints

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Clobber operands

```
int out_var, in_var;
asm ("movl   %1, %%ebx\n"
     "addl   %%ebx, %0"
     : “=r"(out_var)
     : “r”(in_var)
     : “ebx”);
```

* x86_64 assembly w/ AT&T syntax

Introduction to inline assembly

Clobber operands

```
int out_var, in_var;
asm ("movl   %1, %%ebx\n"
     "addl   %%ebx, %0"
     : “=r"(out_var)
     : “r”(in_var)
     : “ebx”);
```

* x86_64 assembly w/ AT&T syntax

For more inline assembly syntax…

https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

Handling inline assembly in Clang & LLVM

Background

Handling inline assembly in Clang & LLVM
Background

- Most parts of an inline assembly string are simply copied into the final assembly file

Handling inline assembly in Clang & LLVM
Background

- Most parts of an inline assembly string are simply copied into the final assembly file
- LLVM needs to “glue” inline assembly operands with the surrounding code

Handling inline assembly in Clang & LLVM
Background

- Most parts of an inline assembly string are simply copied into the final assembly file
- LLVM needs to “glue” inline assembly operands with the surrounding code
- Lots of target-specific logics
  - In both Clang and the backend

Handling inline assembly in Clang & LLVM
Background

- Most parts of an inline assembly string are simply copied into the final assembly file
- LLVM needs to “glue” inline assembly operands with the surrounding code
- Lots of target-specific logics
  - In both Clang and the backend
- Target-specific callbacks are scattered in the codebase
  - Documentation for this part is a little…shy

Goals

Goals

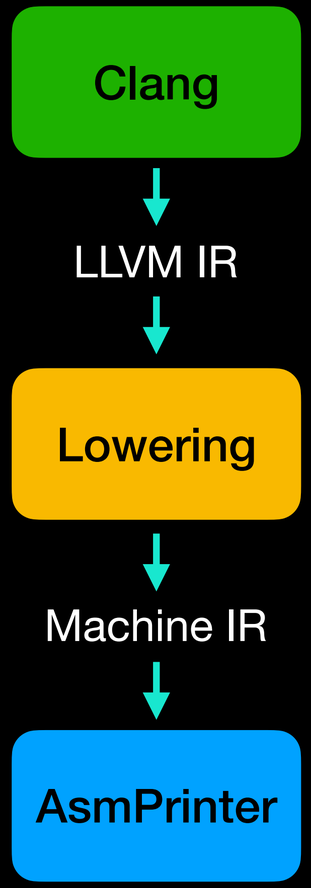
Learning inline assembly workflow in Clang / LLVM

Goals

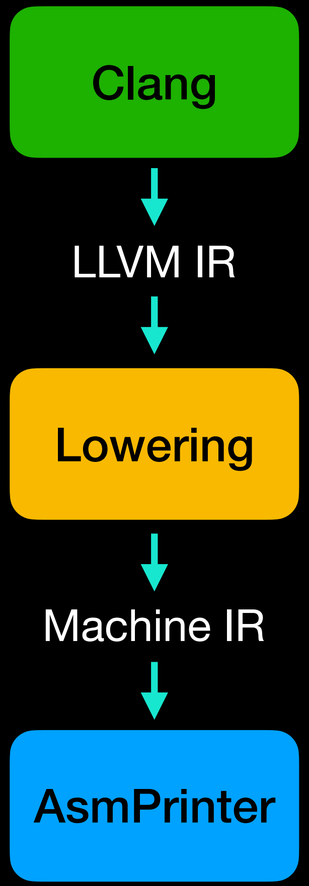
Learning inline assembly workflow in Clang / LLVM

A simple guide for backend developers to add inline assembly support

Outline of target-specific logics in each stage

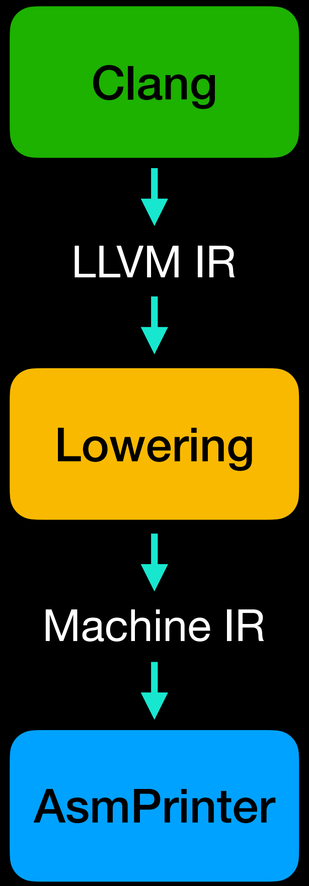


Outline of target-specific logics in each stage



- Simple validation on operand constraints
- Converting constraints

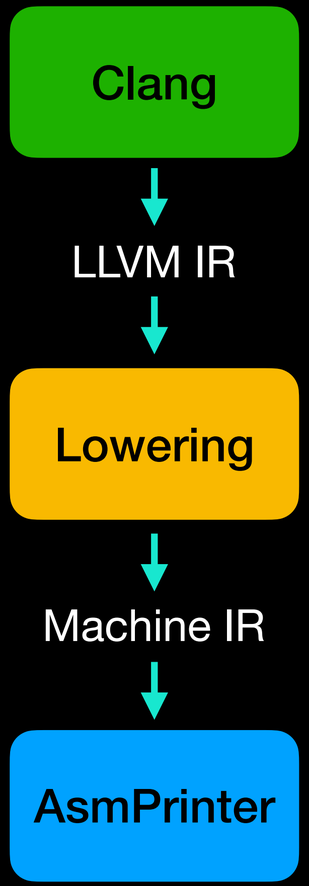
Outline of target-specific logics in each stage



- Simple validation on operand constraints
- Converting constraints

- Classifying constraints
- Constraint validations
- Lowering the operands

Outline of target-specific logics in each stage

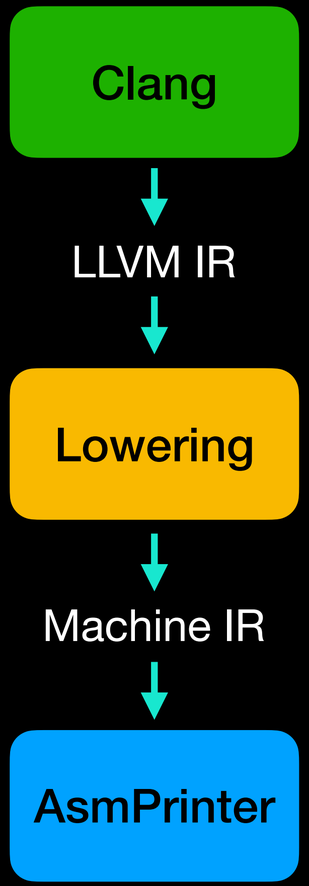


- Simple validation on operand constraints
- Converting constraints

- Classifying constraints
- Constraint validations
- Lowering the operands

Print out different types of operands

Outline of target-specific logics in each stage



- Simple validation on operand constraints
- Converting constraints

- Classifying constraints
- Constraint validations
- Lowering the operands

Print out different types of operands

Operand constraint validations in Clang

```
bool TargetInfo::validateAsmConstraint(const char *&, ConstraintInfo &) const;
```

Operand constraint validations in Clang

```
bool TargetInfo::validateAsmConstraint(const char *&, ConstraintInfo &) const;

bool M68kTargetInfo::validateAsmConstraint(const char *&Name, ConstraintInfo &info) const {
  switch (*Name) {
  case 'a': // address register
    info.setAllowsRegister();
    return true;



  }
  …
}
```

Operand constraint validations in Clang

```
bool TargetInfo::validateAsmConstraint(const char *&, ConstraintInfo &) const;

bool M68kTargetInfo::validateAsmConstraint(const char *&Name, ConstraintInfo &info) const {
  switch (*Name) {
  case 'a': // address register
    info.setAllowsRegister();
    return true;
  case 'J': // constant signed 16-bit integer
    info.setRequiresImmediate(std::numeric_limits<int16_t>::min(),
                              std::numeric_limits<int16_t>::max());
    return true;
  }
  …
}
```

Operand constraint validations in Clang

Limitation on immediate value validations

* M68k assembly w/ Motorola syntax

```
void foo() {
  int32_t x;
  asm ("move.l %0, %%d1" : : "J" (x));
}
```

Operand constraint validations in Clang

Limitation on immediate value validations

* M68k assembly w/ Motorola syntax


```
void foo() {
  int32_t x;
  asm ("move.l %0, %%d1" : : "J" (x));
}
```

Operand constraint validations in Clang

Limitation on immediate value validations

* M68k assembly w/ Motorola syntax


```
void foo() {
  int32_t x;
  asm ("move.l %0, %%d1" : : "J" (x));
}
```

```
$ clang -target m68k -fsyntax-only foo.c
# No error
$ clang -target m68k -emit-llvm foo.c
# No error
```

Operand constraint validations in Clang

Limitation on immediate value validations

* M68k assembly w/ Motorola syntax


```
void foo() {
  int32_t x;
  asm ("move.l %0, %%d1" : : "J" (x));
}
```

```
$ clang -target m68k -fsyntax-only foo.c
# No error
$ clang -target m68k -emit-llvm foo.c
# No error

$ clang -target m68k -S foo.c
error: constraint 'J' expects an integer constant expression
```

Inline assembly in LLVM IR

C/C++

```
void foo() {
  const int x = 87;
  asm ("move.l %0, %%d1" : : "Ci" (x) : "d1");
}
```

* M68k assembly w/ Motorola syntax

Inline assembly in LLVM IR

C/C++

```
void foo() {
  const int x = 87;
  asm ("move.l %0, %%d1" : : "Ci" (x) : "d1");
}
```

* M68k assembly w/ Motorola syntax

LLVM IR

```
call void asm sideeffect                                    (      )
```

Inline assembly in LLVM IR

C/C++

```
void foo() {
  const int x = 87;
  asm ("move.l %0, %%d1" : : "Ci" (x) : "d1");
}
```

* M68k assembly w/ Motorola syntax

LLVM IR

```
call void asm sideeffect "move.l $0, %d1"                    (        )
```

Inline assembly in LLVM IR

C/C++

```
void foo() {
  const int x = 87;
  asm ("move.l %0, %%d1" : : "Ci" (x) : "d1");
}
```

* M68k assembly w/ Motorola syntax

LLVM IR

```
call void asm sideeffect "move.l $0, %d1"                (i32 87)
```

Inline assembly in LLVM IR

C/C++

```
void foo() {
  const int x = 87;
  asm ("move.l %0, %%d1" : : "Ci" (x) : "d1");
}
```

* M68k assembly w/ Motorola syntax

LLVM IR

```
call void asm sideeffect "move.l $0, %d1"                (i32 87)
```

Inline assembly in LLVM IR

C/C++

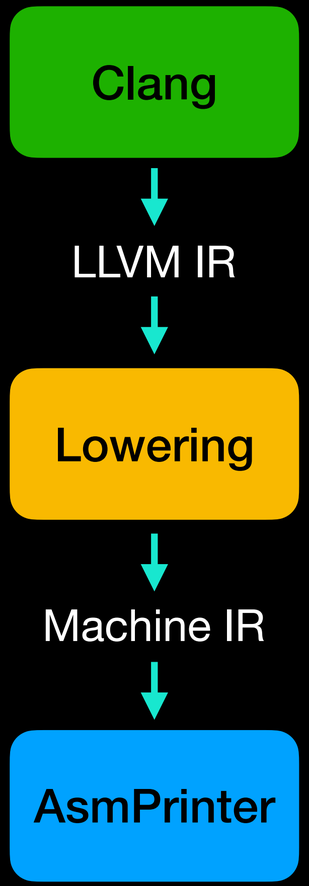
```
void foo() {
  const int x = 87;
  asm ("move.l %0, %%d1" : : "Ci" (x) : "d1");
}
```

* M68k assembly w/ Motorola syntax

LLVM IR

```
call void asm sideeffect "move.l $0, %d1", “^Ci,~{d1}“(i32 87)
```

Outline of target-specific logics in each stage



- Simple validation on operand constraints
- Converting constraints

- Classifying constraints
- Constraint validations
- Lowering the operands

Print out different types of operands

LLVM IR

```
call void asm sideeffect "move.l $0, %d1", “^Ci,~{d1}“(i32 87)
```

LLVM IR

SelectionDAG

```
call void asm sideeffect "move.l $0, %d1", “^Ci,~{d1}“(i32 87)
```

```
t2: ch,glue = inlineasm ... "move.l $0, %d1", ...,
                  TargetConstant:i32<87>, Register:i16 $d1
```

LLVM IR

```
call void asm sideeffect "move.l $0, %d1", “^Ci,~{d1}“(i32 87)
```

SelectionDAG

```
t2: ch,glue = inlineasm ... "move.l $0, %d1", ...,
                    TargetConstant:i32<87>, Register:i16 $d1
```

Machine IR

```
INLINEASM &"move.l $0, %d1", ..., /* imm */ 87, /* clobber */ $d1
```

LLVM IR

```
call void asm sideeffect "move.l $0, %d1", “^Ci,~{d1}“(i32 87)
```

SelectionDAG

```
t2: ch,glue = inlineasm ... "move.l $0, %d1", ...,
                  TargetConstant:i32<87>, Register:i16 $d1
```

Machine IR

```
INLINEASM &"move.l $0, %d1", ..., /* imm */ 87, /* clobber */ $d1
```

Constraint classification

```
TargetLowering::ConstraintType
M68kTargetLowering::getConstraintType(StringRef Constraint) const;
```

Constraint classification

```
TargetLowering::ConstraintType
M68kTargetLowering::getConstraintType(StringRef Constraint) const;
```

Return:

Constraint classification

```
TargetLowering::ConstraintType
M68kTargetLowering::getConstraintType(StringRef Constraint) const;
```

Return:

- C_RegisterClass Ex: ’*r*’

Constraint classification

```
TargetLowering::ConstraintType
M68kTargetLowering::getConstraintType(StringRef Constraint) const;
```

Return:

- C_RegisterClass Ex: ’*r*’
- C_Immediate Ex: ‘*i*’

Constraint classification

```
TargetLowering::ConstraintType
M68kTargetLowering::getConstraintType(StringRef Constraint) const;
```

Return:

- C_RegisterClass Ex: ’*r*’
- C_Immediate Ex: ‘*i*’
- C_Memory Ex: ‘*m*’, ‘*Q*’ (AArch64)

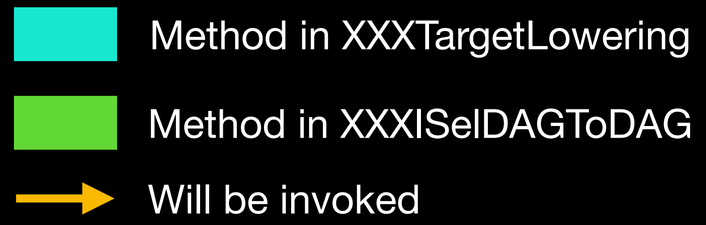
Constraint classification

```
TargetLowering::ConstraintType
M68kTargetLowering::getConstraintType(StringRef Constraint) const;
```

Return:

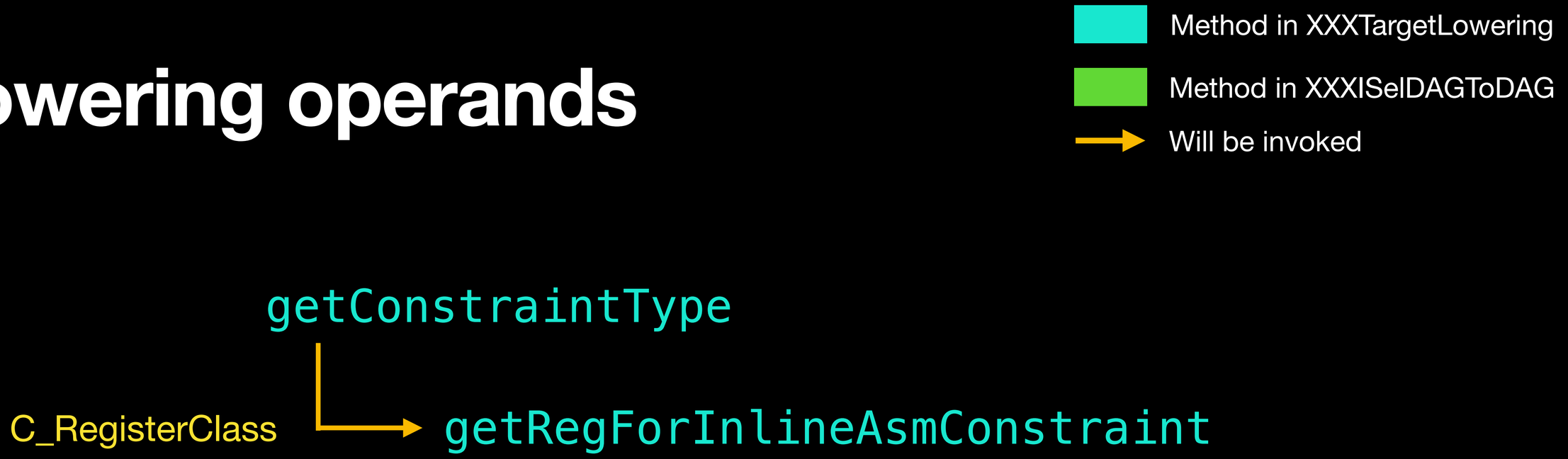
- C_RegisterClass — Ex: ’*r*’
- C_Immediate — Ex: ‘*i*’
- C_Memory — Ex: ‘*m*’, ‘*Q*’ (AArch64)
- C_Other

Lowering operands

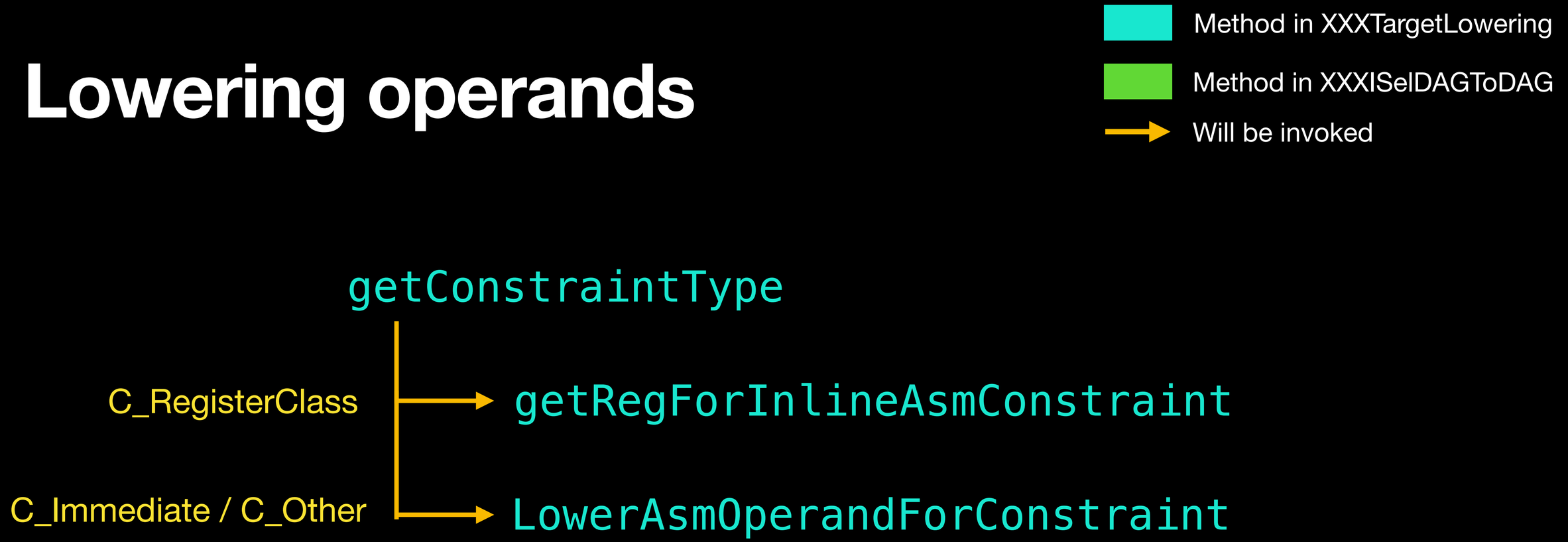



```
getConstraintType
```

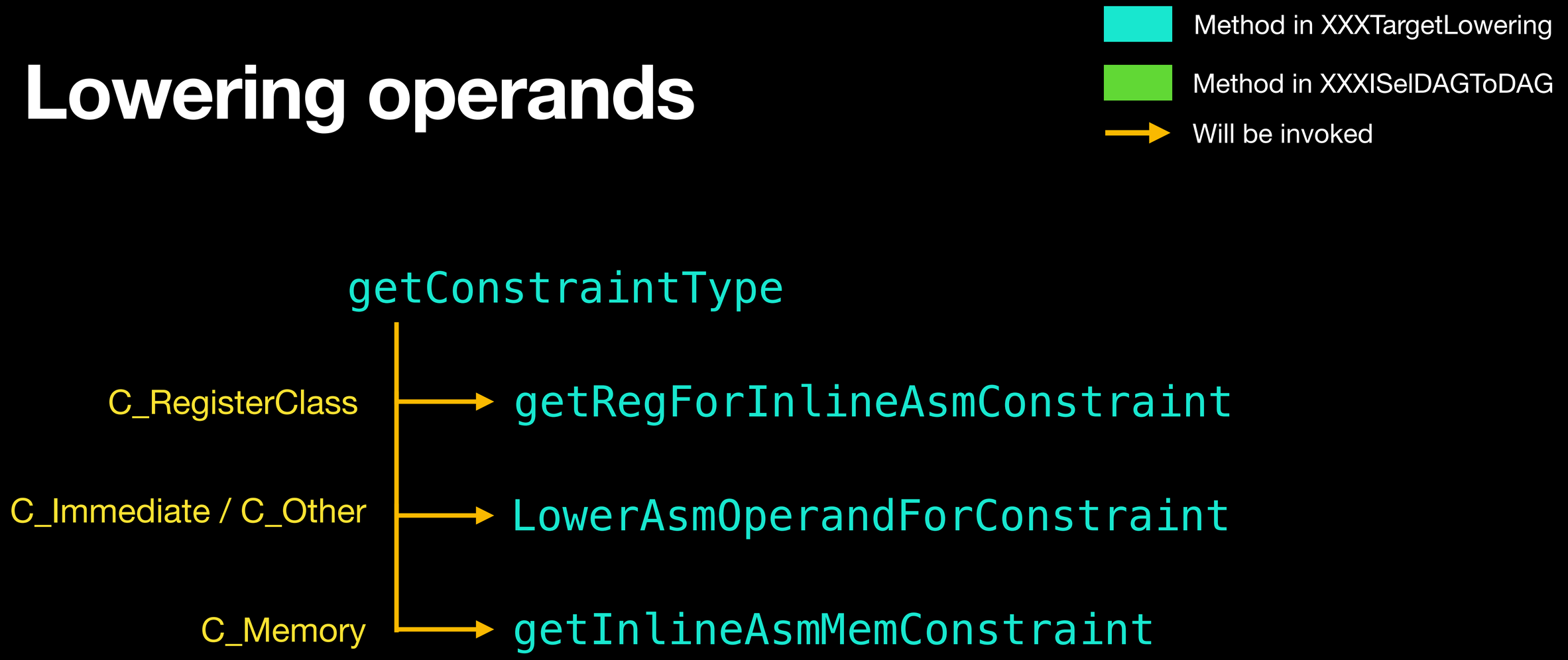
Lowering operands



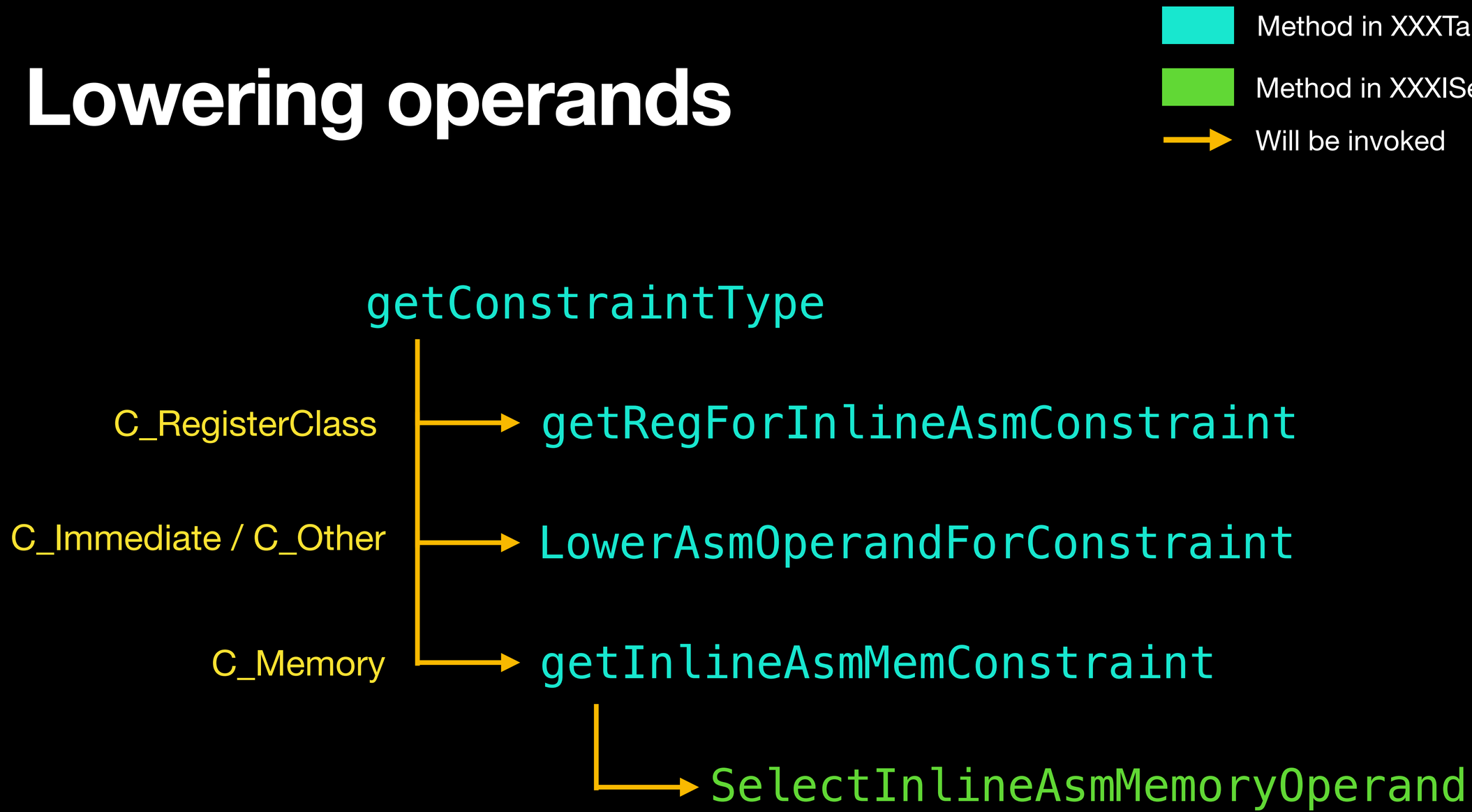
Lowering operands



Lowering operands



Lowering operands

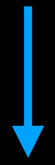


Lowering register operands

```
std::pair<unsigned, const TargetRegisterClass *>
TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                                             StringRef Constraint,
                                             MVT VT) const;
```

Lowering register operands

A specific register or 0 if not applicable

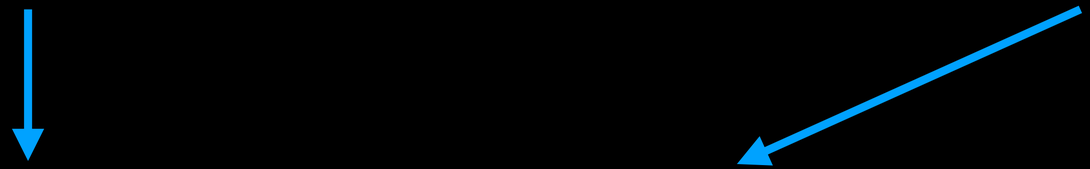


```
std::pair<unsigned, const TargetRegisterClass *>
TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                                             StringRef Constraint,
                                             MVT VT) const;
```

Lowering register operands

A specific register or 0 if not applicable

Valid register class to select from



```
std::pair<unsigned, const TargetRegisterClass *>
TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *TRI,
                                             StringRef Constraint,
                                             MVT VT) const;
```

Lowering register operands

M68k Example

```
std::pair<unsigned, const TargetRegisterClass *>
M68kTargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *,
                                                 StringRef Constraint,
                                                 MVT VT) const {

  switch (Constraint[0]) {
    case 'a':
    switch (VT.SimpleTy) {
      case MVT::i16:
        return std::make_pair(0U, &M68k::AR16RegClass);
    }
  }
}
```

Lowering register operands

M68k Example

```
std::pair<unsigned, const TargetRegisterClass *>
M68kTargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *,
                                                 StringRef Constraint,
                                                 MVT VT) const {

  switch (Constraint[0]) {
    case 'a':
    switch (VT.SimpleTy) {
      case MVT::i16:
        return std::make_pair(0U, &M68k::AR16RegClass);
    }
  }
}
```

Lowering register operands

X86 Example

```
std::pair<unsigned, const TargetRegisterClass *>
X86TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *,
                                                StringRef Constraint,
                                                MVT VT) const {

  ...
  if (Constraint == "Yz") {
    // First SSE register (%xmm0).
    switch (VT.SimpleTy) {
    case MVT::f32:
    case MVT::i32:
      return std::make_pair(X86::XMM0, &X86::FR32RegClass);
    }
  }
}
```

Lowering register operands

X86 Example

```
std::pair<unsigned, const TargetRegisterClass *>
X86TargetLowering::getRegForInlineAsmConstraint(const TargetRegisterInfo *,
                                                StringRef Constraint,
                                                MVT VT) const {

  ...
  if (Constraint == "Yz") {
    // First SSE register (%xmm0).
    switch (VT.SimpleTy) {
    case MVT::f32:
    case MVT::i32:
      return std::make_pair(X86::XMM0, &X86::FR32RegClass);
    }
  }
}
```

Lowering immediate / other operands

```
void TargetLowering::
LowerAsmOperandForConstraint(SDValue Op,
                             std::string &Constraint,
                             std::vector<SDValue> &Ops,
                             SelectionDAG &DAG) const;
```

Lowering immediate / other operands

M68k Example

```
void M68kTargetLowering::LowerAsmOperandForConstraint(SDValue Op,
                                                      std::string &Constraint,
                                                      std::vector<SDValue> &Ops,
                                                      SelectionDAG &DAG) const {
  switch (Constraint[0]) {
  case 'J': { // constant signed 16-bit integer









  }
  }
}
```

Lowering immediate / other operands

M68k Example

```
void M68kTargetLowering::LowerAsmOperandForConstraint(SDValue Op,
                                                      std::string &Constraint,
                                                      std::vector<SDValue> &Ops,
                                                      SelectionDAG &DAG) const {

  switch (Constraint[0]) {
  case 'J': { // constant signed 16-bit integer
    if (auto *C = dyn_cast<ConstantSDNode>(Op)) {
      int64_t Val = C->getSExtValue();




    }
  }
  }
}
```

Lowering immediate / other operands

M68k Example

```
void M68kTargetLowering::LowerAsmOperandForConstraint(SDValue Op,
                                                      std::string &Constraint,
                                                      std::vector<SDValue> &Ops,
                                                      SelectionDAG &DAG) const {

  switch (Constraint[0]) {
  case 'J': { // constant signed 16-bit integer
    if (auto *C = dyn_cast<ConstantSDNode>(Op)) {
      int64_t Val = C->getSExtValue();
      if (isInt<16>(Val)) {
        Ops.push_back(Op);
        return;
      }
    }
  }
  }
}
```

Lowering memory operands

Memory constraint classification

```
unsigned TargetLowering::getInlineAsmMemConstraint(StringRef ConstraintCode) const;
```

Lowering memory operands

Memory constraint classification

```
unsigned TargetLowering::getInlineAsmMemConstraint(StringRef ConstraintCode) const;
```

Return:

Lowering memory operands

Memory constraint classification

```
unsigned TargetLowering::getInlineAsmMemConstraint(StringRef ConstraintCode) const;
```

Return:

- `InlineAsm::Constraint_m`
- `InlineAsm::Constraint_o`
- `InlineAsm::Constraint_v`

Generic memory constraints

Lowering memory operands

Memory constraint classification

```
unsigned TargetLowering::getInlineAsmMemConstraint(StringRef ConstraintCode) const;
```

Return:

- `InlineAsm::Constraint_m`
- `InlineAsm::Constraint_o`
- `InlineAsm::Constraint_v`

Generic memory constraints

- `InlineAsm::Constraint_A`
- `InlineAsm::Constraint_Q`

Target-specific constraints!

Lowering memory operands

Memory constraint classification — RISCV Example

```
unsigned RISCVTargetLowering::
getInlineAsmMemConstraint(StringRef ConstraintCode) const {
  switch (ConstraintCode[0]) {
  case 'A':
    return InlineAsm::Constraint_A;
  ...
  }
  ...
}
```

Lowering memory operands

```
bool
SelectionDAGISel::SelectInlineAsmMemoryOperand(const SDValue &Op,
                                               unsigned ConstraintID,
                                               std::vector<SDValue> &OutOps);
```

Lowering memory operands
RISCV Example

```
bool RISCVDAGToDAGISel::SelectInlineAsmMemoryOperand(
    const SDValue &Op, unsigned ConstraintID,
    std::vector<SDValue> &OutOps) {

  switch (ConstraintID) {
  case InlineAsm::Constraint_A:
    OutOps.push_back(Op);
    return false;
  default:
    break;
  }

  return true;
}
```

Lowering memory operands

Complex addressing mode — X86 Example

```
bool X86DAGToDAGISel::
SelectInlineAsmMemoryOperand(const SDValue &Op, unsigned ConstraintID,
                             std::vector<SDValue> &OutOps) {
  SDValue Op0, Op1, Op2, Op3, Op4;

  switch (ConstraintID) {
  case InlineAsm::Constraint_m: // memory
    if (!selectAddr(nullptr, Op, Op0, Op1, Op2, Op3, Op4))
      return true;
    break;
  }

  OutOps.insert(OutOps.end(), {Op0, Op1, Op2, Op3, Op4});
  return false;
}
```

Lowering memory operands

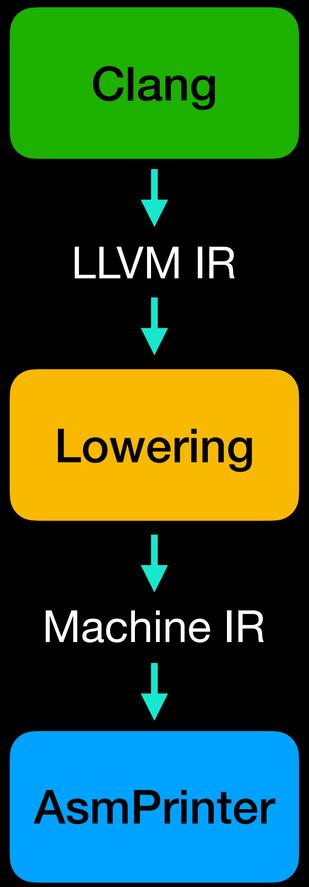
Complex addressing mode — X86 Example

```
bool X86DAGToDAGISel::
SelectInlineAsmMemoryOperand(const SDValue &Op, unsigned ConstraintID,
                             std::vector<SDValue> &OutOps) {
  SDValue Op0, Op1, Op2, Op3, Op4;

  switch (ConstraintID) {
  case InlineAsm::Constraint_m: // memory
    if (!selectAddr(nullptr, Op, Op0, Op1, Op2, Op3, Op4))
      return true;
    break;
  }

  OutOps.insert(OutOps.end(), {Op0, Op1, Op2, Op3, Op4});
  return false;
}
```

Outline of target-specific logics in each stage



- Simple validation on operand constraints
- Converting constraints

- Classifying constraints
- Constraint validations
- Lowering the operands

Print out different types of operands

Machine IR

```
INLINEASM &"move.l $0, %d1", ..., /* imm */ 87, /* clobber */ $d1
```

Machine IR

```
INLINEASM &"move.l $0, %d1", ..., /* imm */ 87, /* clobber */ $d1
```

```
move.l  #87, %d1
```

Machine IR

```
INLINEASM &"move.l $0, %d1", ..., /* imm */ 87, /* clobber */ $d1
```

```
move.l  #87, %d1
```

```
move.l  87, %d1
```

Machine IR

```
INLINEASM &"move.l $0, %d1", ..., /* imm */ 87, /* clobber */ $d1
```

Printing asm operands

```
bool AsmPrinter::PrintAsmOperand(const MachineInstr *MI, unsigned OpNo,
                                 const char *, raw_ostream &O);

bool AsmPrinter::PrintAsmMemoryOperand(const MachineInstr *MI, unsigned OpNo,
                                       const char *, raw_ostream &O);
```

Printing non-memory asm operands

M68k Example

```
void M68kAsmPrinter::printOperand(const MachineInstr *MI, int OpNum,
                                  raw_ostream &OS) {
  const MachineOperand &MO = MI->getOperand(OpNum);
  switch (MO.getType()) {
  ...
  }
}
```

Printing non-memory asm operands

M68k Example

```
void M68kAsmPrinter::printOperand(const MachineInstr *MI, int OpNum,
                                  raw_ostream &OS) {
  const MachineOperand &MO = MI->getOperand(OpNum);
  switch (MO.getType()) {
  case MachineOperand::MO_Register:
    OS << "%" << M68kInstPrinter::getRegisterName(MO.getReg());
    break;




  ...
  }
}
```

Printing non-memory asm operands

M68k Example

```
void M68kAsmPrinter::printOperand(const MachineInstr *MI, int OpNum,
                                  raw_ostream &OS) {
  const MachineOperand &MO = MI->getOperand(OpNum);
  switch (MO.getType()) {
  case MachineOperand::MO_Register:
    OS << "%" << M68kInstPrinter::getRegisterName(MO.getReg());
    break;

  case MachineOperand::MO_Immediate:
    OS << '#' << MO.getImm();
    break;
  ...
  }
}
```

Epilogue

Advanced topics

Converting constraints in Clang

Asm dialects

Operand modifiers

Multi-alternative constraints

Constraint weights

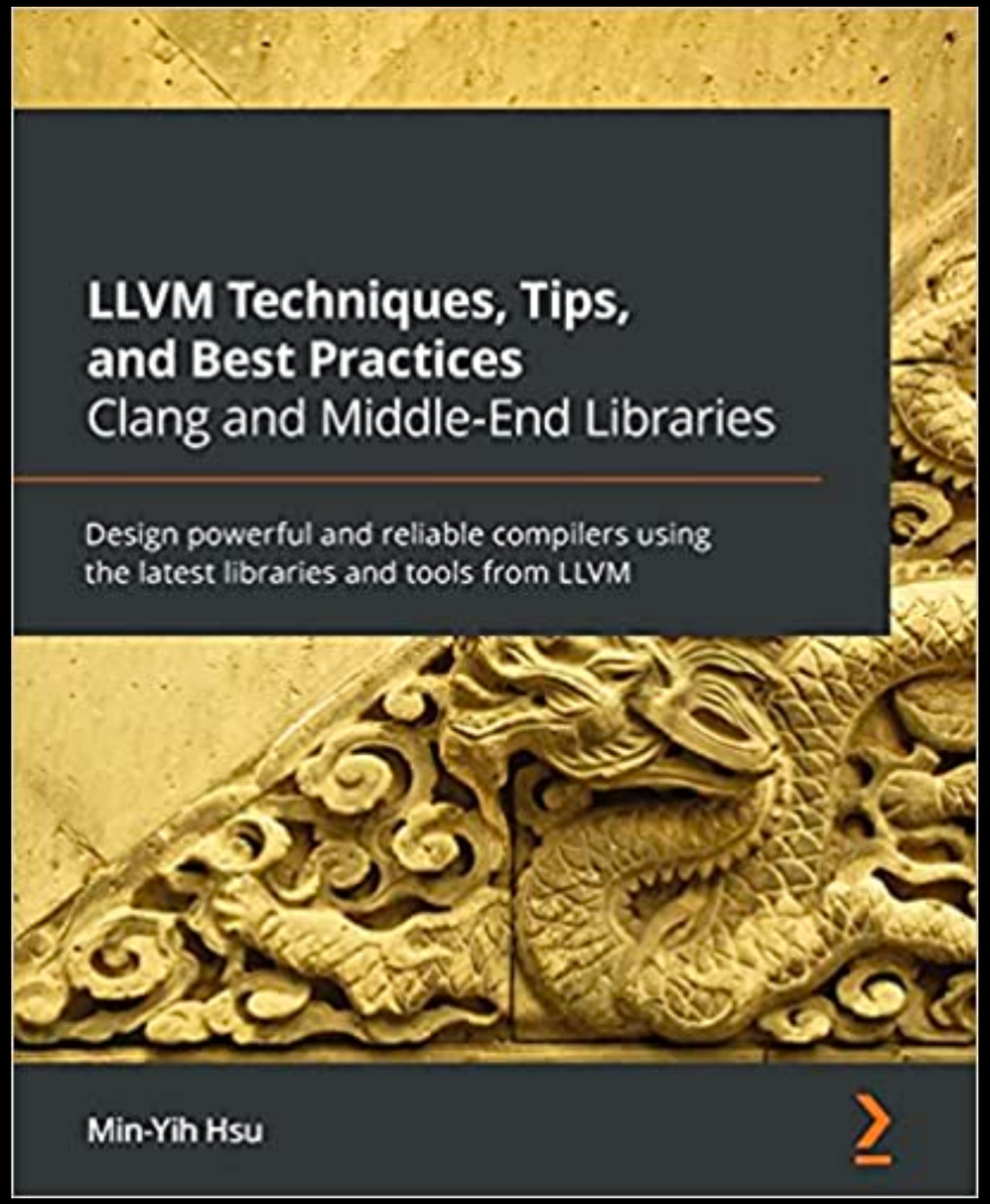
Turn (simple) inline asm into LLVM code

Q&A

GitHub: mshockwave

Email: minyihh@uci.edu

Book URL: https://tinyurl.com/3xnc5r3t



Appendix

Converting constraints in Clang

An operand constraint is assumed to have only *a single character* by default

Converting constraints in Clang

An operand constraint is assumed to have only *a single character* by default

Example: “Ci”

Converting constraints in Clang

An operand constraint is assumed to have only *a single character* by default

Example: “Ci” -> “^Ci”

Converting constraints in Clang

An operand constraint is assumed to have only *a single character* by default

Example: “Ci” -> “^Ci”

```
std::string
M68kTargetInfo::convertConstraint(const char *&Constraint) const override {
  if (*Constraint == 'C')
    // Two-character constraint; add "^" hint for later parsing
    return std::string("^") + std::string(Constraint++, 2);

  return std::string(1, *Constraint);
}
```

Operand constraint validations in Clang

Limitation on immediate value validations (cont’d)

```
bool M68kTargetInfo::validateAsmConstraint(const char *&Name, ConstraintInfo &info) const {
  switch (*Name) {
  …
  case 'C':
    ++Name;
    switch (*Name) {
    case 'i': // constant integer
    case 'j': // integer constant that doesn't fit in 16 bits
      info.setRequiresImmediate();
      return true;
    }
    break;
  }
  …
}
```