Compiler Construction

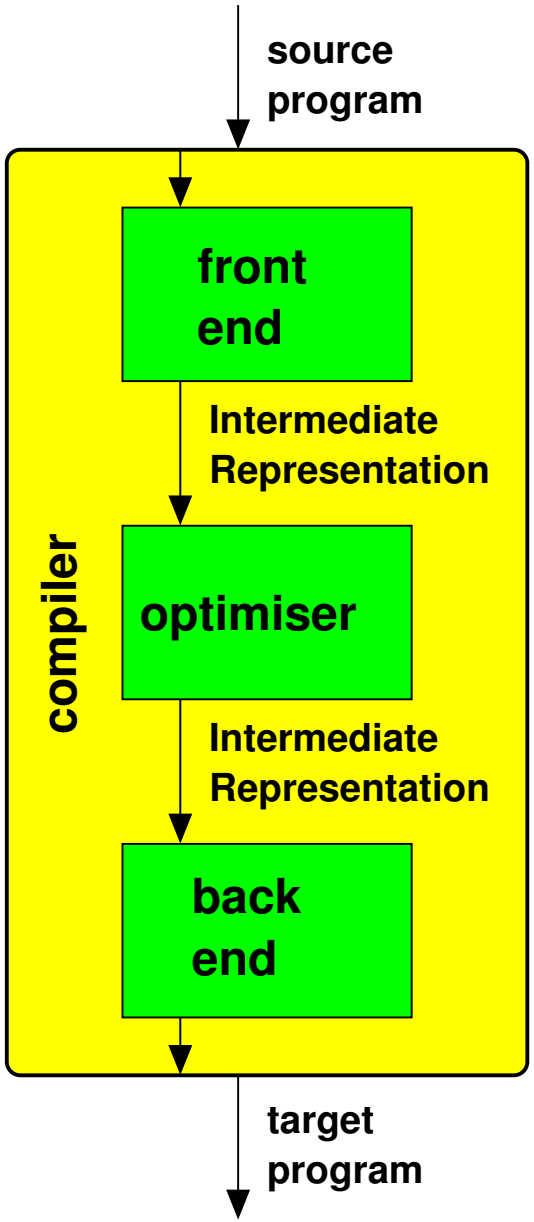
Chapter 7

Code Generation

Clemens Grelck

System and Network Engineering Lab

Advanced Compiler Structure



Front end:

- Analysis of source program

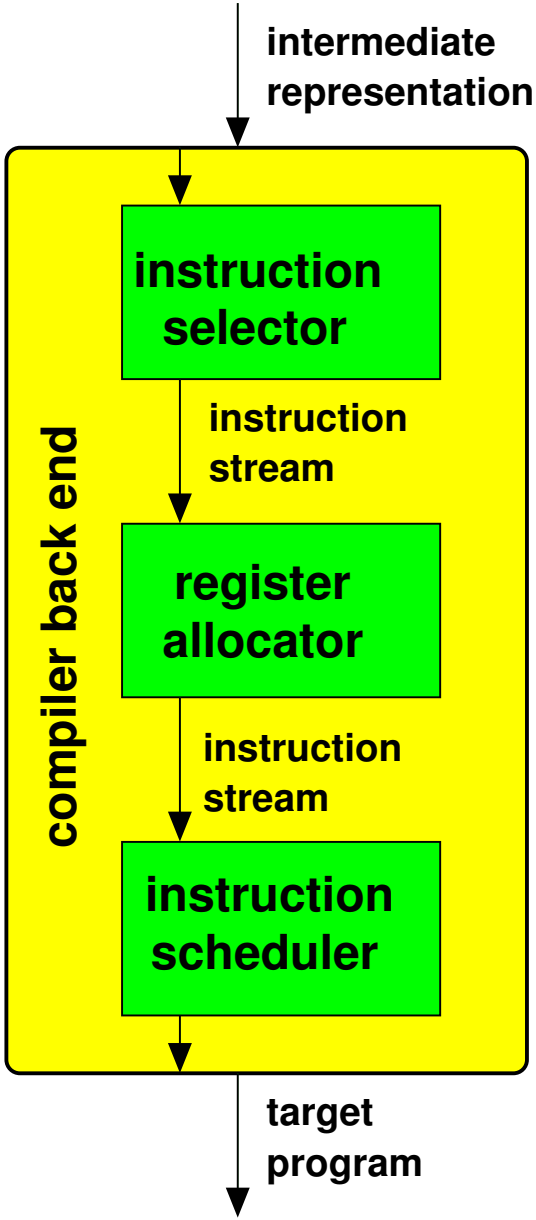
Optimiser (Middle end):

- Code transformation within intermediate representation
- Motivated by some non-functional requirements / desires:
- Speed, power, size, ...

Back end:

- Synthesis of target program

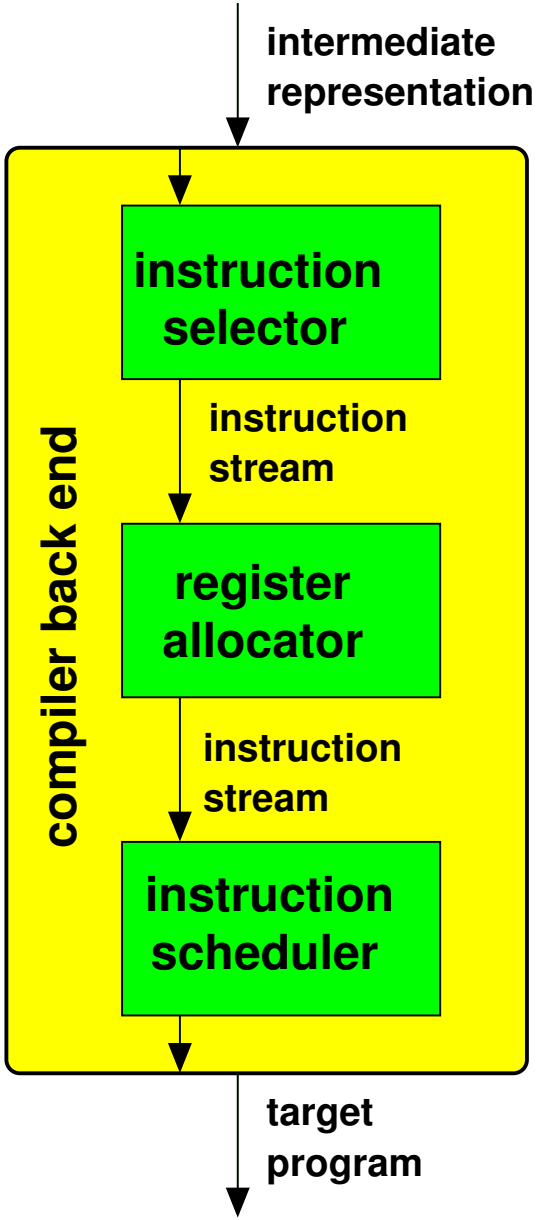
Compiler Backend



Instruction selector:

- Input: abstract syntax tree
- Output: finite stream of assembly instructions
- Unbounded number of registers used

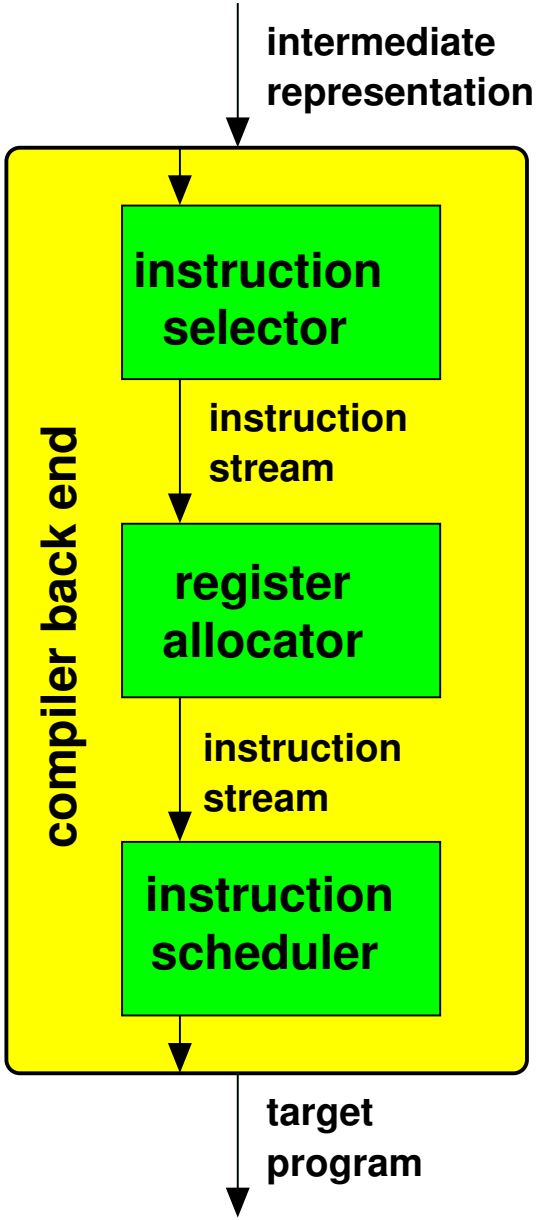
Compiler Backend



Register allocator:

- Input: finite stream of assembly instructions
- Output: finite stream of assembly instructions
- Bounded number of registers used
- Spilling to runtime stack

Compiler Backend



Instruction scheduler:

- Input: finite stream of assembly instructions
- Output: finite stream of assembly instructions
- Instruction reordering
- For superscalar architectures
- Reduce required number of cycles

Taxonomy of Target Languages

Machine assembly:

- Linear sequence of elementary instructions
- Instructions implemented in hardware
- Program runs on “real hardware”

Taxonomy of Target Languages

Machine assembly:

- Linear sequence of elementary instructions
- Instructions implemented in hardware
- Program runs on "real hardware"

Virtual machine assembly:

- Linear sequence of elementary instructions
- Instructions implemented in software
- Programs run on "virtual hardware" / interpreter

Taxonomy of Target Languages

Machine assembly:

- Linear sequence of elementary instructions
- Instructions implemented in hardware
- Program runs on "real hardware"

Virtual machine assembly:

- Linear sequence of elementary instructions
- Instructions implemented in software
- Programs run on "virtual hardware" / interpreter

Higher level language:

- Structured control flow constructs
- Procedure / function abstractions
- Program compiled by additional backend compiler

Taxonomy of Target Languages: Machine Assembly

Advantages:

- Execution speed
- Adaptation to specific machine characteristics
- Someone has to do the dirty job

Taxonomy of Target Languages: Machine Assembly

Advantages:

- Execution speed
- Adaptation to specific machine characteristics
- Someone has to do the dirty job

Disadvantages:

- No portability among different architectures
- One code generator per supported architecture
- Real hardware often has weird side conditions
- Design of Instructions motivated by hardware requirements/opportunities

Taxonomy of Target Languages: Machine Assembly

Advantages:

- Execution speed
- Adaptation to specific machine characteristics
- Someone has to do the dirty job

Disadvantages:

- No portability among different architectures
- One code generator per supported architecture
- Real hardware often has weird side conditions
- Design of Instructions motivated by hardware requirements/opportunities

Good to know:

- Modern compilers often use code generator generators
- Like with parser generators the code generator for some architecture is generated from an abstract description.

Taxonomy of Target Languages: Virtual Machine Assembly

Advantages:

- Portability among large range of architectures
- Instruction set design can take source language into account
- Additional virtualisation layer facilitates code generation
- Byte code can be more compact than register transfer code
- Byte code interpretation has interesting security aspects

Taxonomy of Target Languages: Virtual Machine Assembly

Advantages:

- Portability among large range of architectures
- Instruction set design can take source language into account
- Additional virtualisation layer facilitates code generation
- Byte code can be more compact than register transfer code
- Byte code interpretation has interesting security aspects

Disadvantages:

- Lower runtime performance due to interpretation overhead !!

Taxonomy of Target Languages: Virtual Machine Assembly

Advantages:

- Portability among large range of architectures
- Instruction set design can take source language into account
- Additional virtualisation layer facilitates code generation
- Byte code can be more compact than register transfer code
- Byte code interpretation has interesting security aspects

Disadvantages:

- Lower runtime performance due to interpretation overhead !!

Good to know:

- Speed penalty mitigated through just-in-time compilation
- Today portability is often more valued than execution speed
- Unless you work in high performance computing etc
- Many modern languages follow this approach:
  Java, C#, Python, Perl, Pascal, Lua, etc

Taxonomy of Target Languages: Higher Level Language

Advantages:

- Feature rich compilation target
- Good portability if target language is wide-spread
- Leaves "uninteresting" features as "someone else's problem"
- Good performance if compiled to native code

Taxonomy of Target Languages: Higher Level Language

Advantages:

- Feature rich compilation target
- Good portability if target language is wide-spread
- Leaves "uninteresting" features as "someone else's problem"
- Good performance if compiled to native code

Disadvantages:

- Only makes sense if source language is of considerably higher abstraction level
- Deficiencies in backend compilation are beyond your control

Taxonomy of Target Languages: Higher Level Language

Advantages:

- Feature rich compilation target
- Good portability if target language is wide-spread
- Leaves "uninteresting" features as "someone else's problem"
- Good performance if compiled to native code

Disadvantages:

- Only makes sense if source language is of considerably higher abstraction level
- Deficiencies in backend compilation are beyond your control

Good to know:

- Can also be used if source and target language are of similar abstraction level:
  - Target language is more wide-spread than source language ($\Rightarrow$ portability)
  - You have plenty of code in source language that you do not want to rewrite

Virtual Machines

Register-based virtual machines:

- Mimick RISC machines
- Example: `ADD R0 R1 R2 // R0 = R1 + R2`
- 3-address register transfer code
- Large number of virtual registers
- Registers mapped to stack
- $\rightarrow$ Lua

Virtual Machines

Register-based virtual machines:

- Mimick RISC machines
- Example: `ADD R0 R1 R2 // R0 = R1 + R2`
- 3-address register transfer code
- Large number of virtual registers
- Registers mapped to stack
- $\rightarrow$ Lua

Stack-based virtual machines:

- Almost all operations use a stack as implicit parameter
- Example: `ADD // ...;op1;op2 -> ...;op1+op2`
- No registers, just stack locations
- Very compact "byte" code: instruction takes literally 1 byte
- $\rightarrow$ Java, C#, ...

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

Instead of a block diagram:

- Instruction memory

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

Instead of a block diagram:

- Instruction memory
- Stack memory

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

Instead of a block diagram:

- Instruction memory
- Stack memory
- Heap memory

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

Instead of a block diagram:

- Instruction memory
- Stack memory
- Heap memory
- 3 Special purpose registers:
  - program counter (PC)
  - stack pointer (SP)
  - frame pointer (FP)

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

Instead of a block diagram:

- Instruction memory
- Stack memory
- Heap memory
- 3 Special purpose registers:
  - program counter (PC)
  - stack pointer (SP)
  - frame pointer (FP)
- Runtime constant pool

The CiviC Virtual Machine

General Characteristics:

- Many ideas and concepts adopted from Java Virtual Machine
- Simplified in several aspects:
  no objects, inheritance, threads, ...
- Stack-based design
- Byte code interpretation

Instead of a block diagram:

- Instruction memory
- Stack memory
- Heap memory
- 3 Special purpose registers:
  - program counter (PC)
  - stack pointer (SP)
  - frame pointer (FP)
- Runtime constant pool
- Global variable frame

Arithmetic Operations in CiviC-VM

General characteristics:

- CiviC-VM features virtual machine instructions for
  - addition: `iadd`, `fadd`
  - subtraction: `isub`, `fsub`
  - multiplication: `imul`, `fmul`
  - division: `idiv`, `fdiv`
  - remainder: `irem`
- Operations are typed:
  - $int \times int \rightarrow int$
  - $float \times float \rightarrow float$
- Operations implicitly operate on the stack:
  - ... ; lop ; rop $\Rightarrow$ ... ; lop $\oplus$ rop

Relational Operations in CiviC-VM

General characteristics:

- CiviC-VM features virtual machine instructions for
  - equality: `ieq`, `feq`
  - inequality: `ine`, `fne`
  - less than: `ilt`, `flt`
  - greater than: `igt`, `fgt`
  - less than or equal: `ile`, `fle`
  - greater than or equal: `ige`, `fge`
- Operations are typed:
  - $int \times int \rightarrow bool$
  - $float \times float \rightarrow bool$
- Operations implicitly operate on the stack:
  - ... ; lop ; rop $\Rightarrow$ ... ; lop $\oplus$ rop

Logical Operations in CiviC-VM

General characteristics:

- CiviC-VM features virtual machine support for
  - equality: `beq`
  - inequality: `bne`
  - negation: `bnot`
- Operations are typed:
  - $bool \times bool \rightarrow bool$
  - $bool \rightarrow bool$
- Operations implicitly operate on the stack:
  - ... ; lop ; rop $\Rightarrow$ ... ; lop $\oplus$ rop
  - ... ; op $\Rightarrow$ ... ; $\neg$ op

Logical Operations in CiviC-VM

Logical disjunction and conjunction:

- Not supported by CiviC-VM !!
- Not supported by other physical or virtual assemblies either !!

Logical Operations in CiviC-VM

Logical disjunction and conjunction:

- Not supported by CiviC-VM !!
- Not supported by other physical or virtual assemblies either !!

Motivation:

- Short circuit evaluation of disjunction and conjunction
- Once the values of both operand expressions are on the stack (in registers), it is too late to implement
- We need to treat these operations different from all other binary operations

Type Conversion in CiviC-VM

General characteristics:

- CiviC-VM features 3 basic types: int, float, bool
- But only int and float values can be converted into each other:
  - Conversion int to float: `i2f`
  - Conversion float to int: `f2i`
- Operations are typed:
  - $int \rightarrow float$
  - $float \rightarrow int$
- Operations implicitly operate on the stack:
  - ... ; op $\Rightarrow$ ... ; $\oplus$ op

Type Conversion in CiviC-VM

General characteristics:

- CiviC-VM features 3 basic types: int, float, bool
- But only int and float values can be converted into each other:
  - Conversion int to float: `i2f`
  - Conversion float to int: `f2i`
- Operations are typed:
  - $int \rightarrow float$
  - $float \rightarrow int$
- Operations implicitly operate on the stack:
  - ... ; op $\Rightarrow$ ... ; $\oplus$ op

Other type conversions:

- CiviC supports further conversions
- They require explicit implementation, be creative !

Constants in CiviC-VM

General characteristics:

- Constants are stored in the runtime constant pool

Constants in CiviC-VM

General characteristics:

- Constants are stored in the runtime constant pool
- Table of all constants used within the program

Constants in CiviC-VM

General characteristics:

- Constants are stored in the runtime constant pool
- Table of all constants used within the program
- Example:

0	int	0
1	int	42
2	float	3.141
3	int	-1

Constants in CiviC-VM

General characteristics:

- Constants are stored in the runtime constant pool
- Table of all constants used within the program
- Example:

0	int	0
1	int	42
2	float	3.141
3	int	-1

- Instructions to load values from constant pool onto stack:
  `iloadc` *index*, `floadc` *index*

Variables in CiviC-VM

General characteristics:

- Variables are stored in current frame:
  - function parameters
  - local variables

Variables in CiviC-VM

General characteristics:

- Variables are stored in current frame:
  - function parameters
  - local variables
- Variables in the current frame are enumerated starting with zero for the first parameter
- Example:

```
int foo( int a, int b)
{
  float x;
  bool y;
  ...
}
```

yields

0	int	a
1	int	b
2	float	x
3	bool	y

Variables in CiviC-VM

General characteristics:

- Variables are stored in current frame:
  - function parameters
  - local variables
- Variables in the current frame are enumerated starting with zero for the first parameter
- Example:

```
int foo( int a, int b)
{
  float x;
  bool y;
  ...
}
```

yields

0	int	a
1	int	b
2	float	x
3	bool	y

- Load values from current frame onto stack:
  `iload` *index*, `fload` *index*, `bload` *index*
- Store values from stack into current frame:
  `istore` *index*, `fstore` *index*, `bstore` *index*

Code Generation for Basic Blocks

Example:

```
z = (2 * x) + (y / 42);
```

Code Generation for Basic Blocks

Example:

```
z = (2 * x) + (y / 42);
```

Following context analysis:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

- Indices into current frame for local variables and parameters
- Indices into runtime constant pool for constants

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable
3. Traverse into right hand side expression

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable
3. Traverse into right hand side expression
   - binary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into right operand
     - 3.3 traverse into left operand

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable
3. Traverse into right hand side expression
   - binary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into right operand
     - 3.3 traverse into left operand
   - unary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into operand

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable
3. Traverse into right hand side expression
   - binary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into right operand
     - 3.3 traverse into left operand
   - unary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into operand
   - local variable:
     - 3.1 push load instruction with index from symbol table

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable
3. Traverse into right hand side expression
   - binary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into right operand
     - 3.3 traverse into left operand
   - unary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into operand
   - local variable:
     - 3.1 push load instruction with index from symbol table
   - constant
     - 3.1 push loadc instruction with index from constant table

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Strategy:

1. Traverse down to last statement
2. Push store instruction for left hand side variable
3. Traverse into right hand side expression
   - binary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into right operand
     - 3.3 traverse into left operand
   - unary operator:
     - 3.1 push instruction symbol
     - 3.2 traverse into operand
   - local variable:
     - 3.1 push load instruction with index from symbol table
   - constant
     - 3.1 push loadc instruction with index from constant table
4. Traverse back up to previous statement

Code Generation for Basic Blocks

Example:

$z^{(2)}$ = ($2^{(0)}$ * $x^{(0)}$) + ($y^{(1)}$ / $42^{(1)}$);

Blackboard !!

Code Generation for Basic Blocks

Example:

$z^{(2)} = (2^{(0)} * x^{(0)}) + (y^{(1)} / 42^{(1)});$

Resulting code:

```
iloadc 0   // loading constant 2
iload 0    // loading variable x
imul       // multiplication leaves 2*x on stack
iload 1    // loading variable y
iloadc 1   // loading constant 42
idiv       // division leaves y/42 on stack
iadd       // addition leaves (2 * x) + (y / 42) on stack
istore 2   // storing top of stack in variable z
```

Byte Code Optimisation

Purpose of optimisation:

- Speed up interpretation of byte code
- Make byte code more compact

Byte Code Optimisation

Purpose of optimisation:

- Speed up interpretation of byte code
- Make byte code more compact

Opportunity for optimisation:

- Byte code allows for 256 different opcodes
- Vital instructions do not occupy available range

Byte Code Optimisation

Purpose of optimisation:

- Speed up interpretation of byte code
- Make byte code more compact

Opportunity for optimisation:

- Byte code allows for 256 different opcodes
- Vital instructions do not occupy available range

Idea for optimisation:

- Add special instructions for common code patterns

Byte Code Optimisation

Loading constants:

- Loading boolean constants: `bloadc_t`, `bloadc_f`
- Loading integer constants: `iloadc_0`, `iloadc_1`, `iloadc_m1`
- Loading float constants: `floadc_0`, `floadc_1`

Byte Code Optimisation

Loading constants:

- Loading boolean constants: `bloadc_t`, `bloadc_f`
- Loading integer constants: `iloadc_0`, `iloadc_1`, `iloadc_m1`
- Loading float constants: `floadc_0`, `floadc_1`

Loading variables:

- Loading first frame elements: `iload_0`, ..., `iload_3`
- Loading first frame elements: `fload_0`, ..., `fload_3`

Byte Code Optimisation

Loading constants:

- Loading boolean constants: `bloadc_t`, `bloadc_f`
- Loading integer constants: `iloadc_0`, `iloadc_1`, `iloadc_m1`
- Loading float constants: `floadc_0`, `floadc_1`

Loading variables:

- Loading first frame elements: `iload_0`, ..., `iload_3`
- Loading first frame elements: `fload_0`, ..., `fload_3`

Immediate arithmetic on current frame:

- Incrementing integer: `iinc` *index const*, `iinc_1` *index*
- Decrementing integer: `idec` *index const*, `idec_1` *index*

Code Generation for Simple Expressions

Example expression:

```
i = i + 1;
```

Code Generation for Simple Expressions

Example expression:

```
i = i + 1;
```

Naive code generation: 8 bytes:

```
iload 0          // 1 + 1 Byte (index into frame)
iloadc 0         // 1 + 2 Byte (index into constant pool)
iadd             // 1     Byte
istore 0         // 1 + 1 Byte (index into frame)
```

Code Generation for Simple Expressions

Example expression:

```
i = i + 1;
```

Naive code generation: 8 bytes:

```
iload 0          // 1 + 1 Byte (index into frame)
iloadc 0         // 1 + 2 Byte (index into constant pool)
iadd             // 1     Byte
istore 0         // 1 + 1 Byte (index into frame)
```

Less naive code generation: 4 bytes:

```
iload_0          // 1 Byte
iloadc_1         // 1 Byte
iadd             // 1 Byte
istore_0         // 1 Byte
```

Code Generation for Simple Expressions

Example expression:

```
i = i + 1;
```

Naive code generation: 8 bytes:

```
iload 0          // 1 + 1 Byte (index into frame)
iloadc 0         // 1 + 2 Byte (index into constant pool)
iadd             // 1     Byte
istore 0         // 1 + 1 Byte (index into frame)
```

Less naive code generation: 4 bytes:

```
iload_0          // 1 Byte
iloadc_1         // 1 Byte
iadd             // 1 Byte
istore_0         // 1 Byte
```

Clever code generation: 2 bytes:

```
iinc_1 0         // 1 + 1 Byte
```

Code Generation for Control Flow Constructs

CiviC-VM instructions:

- Needed: Instructions that manipulate the PC other than incrementation

Code Generation for Control Flow Constructs

CiviC-VM instructions:

- Needed: Instructions that manipulate the PC other than incrementation
- Unconditional jump: `jump` *offset*
  - continues program execution at PC plus *offset*

Code Generation for Control Flow Constructs

CiviC-VM instructions:

- Needed: Instructions that manipulate the PC other than incrementation
- Unconditional jump: `jump` *offset*
  - continues program execution at PC plus *offset*
- Conditional branch: `branch_t` *offset*
  - continues program execution at PC plus *offset* **if** the top of the operand stack is **true**
  - otherwise, continues with next instruction
  - boolean value popped from operand stack

Code Generation for Control Flow Constructs

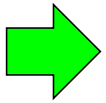
CiviC-VM instructions:

- Needed: Instructions that manipulate the PC other than incrementation
- Unconditional jump: `jump` *offset*
  - continues program execution at PC plus *offset*
- Conditional branch: `branch_t` *offset*
  - continues program execution at PC plus *offset* **if** the top of the operand stack is **true**
  - otherwise, continues with next instruction
  - boolean value popped from operand stack
- Conditional branch: `branch_f` *offset*,
  - continues program execution at PC plus *offset* **if** the top of the operand stack is **false**
  - otherwise, continues with next instruction
  - boolean value popped from operand stack

Code Generation for Conditionals

Example:

```
if (a < 0) {
  a = a + 1;
}
else {
  a = a * 5;
}
b = a;
```



```
iload 0
iloadc_0
ilt
branch_f ??
iinc_1 0
jump ??
iload 0
iloadc 0
imul
istore 0
iload 0
istore 1
```

Offsets for Jump/Branch Instructions

Direct approach:

- When recursively generating code for branches, count instructions on byte level
- Compute offsets directly
- Difficult !

Offsets for Jump/Branch Instructions

Direct approach:

- When recursively generating code for branches, count instructions on byte level
- Compute offsets directly
- Difficult !

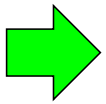
Indirect approach:

- First, generate code with symbolic labels
- When done, counts bytes of instructions and operands
- At last, replace labels by numbers as counted

Code Generation for Conditionals

Example with labels:

```
if (a < 0) {
  a = a + 1;
}
else {
  a = a * 5;
}
b = a;
```

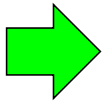


```
  iload 0
  iloadc_0
  ilt
  branch_f L1
  iinc_1 0
  jump L2
L1:
  iload 0
  iloadc 0
  imul
  istore 0
L2:
  iload 0
  istore 1
```

Code Generation for Conditionals

Example with exact byte counts:

```
if (a < 0) {
  a = a + 1;
}
else {
  a = a * 5;
}
b = a;
```

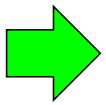


```
iload 0
iloadc_0
ilt
branch_f 8
iinc_1 0
jump 11
iload 0
iloadc 0
imul
istore 0
iload 0
istore 1
```

Code Generation for Do-While-Loops

Example with labels:

```
do {
  a = a + 1;
}
while (a <= 10);
b = a;
```

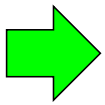


```
L1:
  iinc_1 0
  iload 0
  iloadc 0
  ile
  branch_t L1
  iload 0
  istore 1
```

Code Generation for Do-While-Loops

Example with exact byte counts:

```
do {
  a = a + 1;
}
while (a <= 10);
b = a;
```



```
iinc_1 0
iload 0
iloadc 0
ile
branch_t -8
iload 0
istore 1
```

Code Generation for While-Loops

Direct approach:

- Add code generator in analogy to do-while loop

Code Generation for While-Loops

Direct approach:

- Add code generator in analogy to do-while loop

Transformational approach:

- Transform all while-loops into do-while-loops before code generation
- Re-use existing code generation scheme
- See Assignments

Global Variables in CiviC-VM

General characteristics:

- Global variables are stored in the global frame
- Static stack frame similar to runtime constant pool

Global Variables in CiviC-VM

General characteristics:

- Global variables are stored in the global frame
- Static stack frame similar to runtime constant pool
- Example:

0	int	*value*
1	int	*value*
2	float	*value*
3	int	*value*

Global Variables in CiviC-VM

General characteristics:

- Global variables are stored in the global frame
- Static stack frame similar to runtime constant pool
- Example:

0	int	*value*
1	int	*value*
2	float	*value*
3	int	*value*

- Load values from global frame onto operand stack:
  `iloadg` *index*, `floadg` *index*, `bloadg` *index*
- Store values from operand stack into global frame:
  `istoreg` *index*, `fstoreg` *index*, `bstoreg` *index*

Initialisation of Global Variables

Language recap:

- Global variables can be initialised by arbitrary expression
- Corresponding code must be executed at load time

Initialisation of Global Variables

Language recap:

- Global variables can be initialised by arbitrary expression
- Corresponding code must be executed at load time

Solution:

- All global initialisation code needs to be transferred into a module initialisation function `__init` prior to code generation

Initialisation of Global Variables

Language recap:

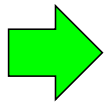
- Global variables can be initialised by arbitrary expression
- Corresponding code must be executed at load time

Solution:

- All global initialisation code needs to be transferred into a module initialisation function `__init` prior to code generation

Example:

```
int foo = 42;
float bar = foobar( foo);
```



```
int foo;
float bar;

void __init( )
{
  foo = 42;
  bar = foobar( foo);
}
```

Code Generation for Extern Declarations

Problem:

- Functions declared `extern` come from a different module
- Requires linking at load time of virtual machine
- Linking means locating functions in other modules

Code Generation for Extern Declarations

Problem:

- Functions declared `extern` come from a different module
- Requires linking at load time of virtual machine
- Linking means locating functions in other modules

Solution:

- An extern declaration leads to an entry in the runtime constant pool

Code Generation for Extern Declarations

Problem:

- Functions declared `extern` come from a different module
- Requires linking at load time of virtual machine
- Linking means locating functions in other modules

Solution:

- An extern declaration leads to an entry in the runtime constant pool

Example: `extern void foo( int a, float b):`

0	int	0
1	int	42
2	float	3.141
3	void int float	“foo”

External linkage information

Problem:

- To locate extern functions in other modules we need a mechanism to publish exported functions as well

External linkage information

Problem:

- To locate extern functions in other modules we need a mechanism to publish exported functions as well

Solution:

- Each exported function definition incurs an entry in the runtime constant pool

External linkage information

Problem:

- To locate extern functions in other modules we need a mechanism to publish exported functions as well

Solution:

- Each exported function definition incurs an entry in the runtime constant pool

Example: `export int bar( bool a) {...}`:

0	c	int	0	
1	c	int	42	
2	c	float	3.141	
3	i	void int float	“foo”	offset
4	e	int bool	“bar”	offset

Defining the Runtime Constant Pool

Example:

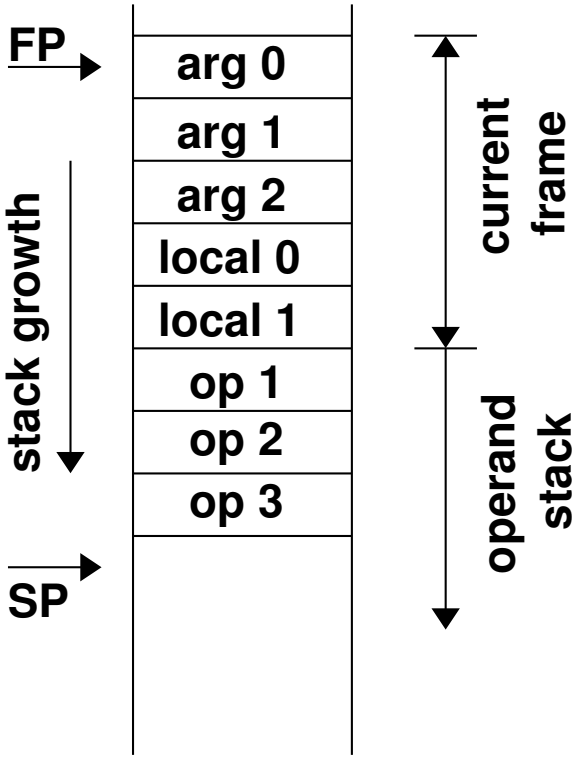
0	c	int	0	
1	c	int	42	
2	c	float	3.141	
3	i	void int float	“foo”	offset
4	e	int bool	“bar”	offset

Pseudo instructions:

```
.const int 0                        // integer constant with value 0
.const int 42                       // integer constant with value 42
.const float 3.141                  // float constant with value 3.141
.import "foo" void int float        // extern function with types
.export "bar" int bool 256          // export function with types and
                                    // address offset in local module
```

Organisation of Runtime Stack

Procedure view:



Calling Convention

Characteristics:

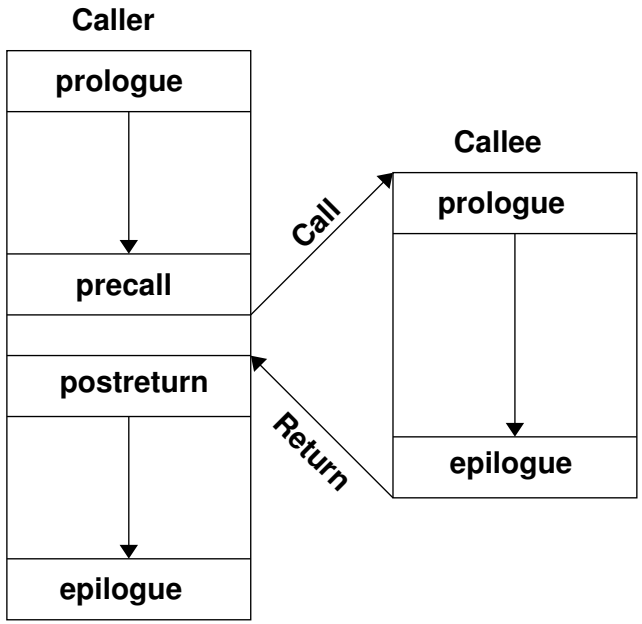
- The calling convention relates a procedure call to a procedure definition
- Call and definition may be in different modules
- Separate compilation requires standard procedure

Calling Convention

Characteristics:

- The calling convention relates a procedure call to a procedure definition
- Call and definition may be in different modules
- Separate compilation requires standard procedure

Illustration:



Calling Convention

Precall sequence:

- Starts constructing the callee's environment:
- Reserve space to hold return value (if necessary)
- Reserve space for activation record
- Evaluate arguments
- Jump to callee's start address

Calling Convention

Precall sequence:

- Starts constructing the callee's environment:
- Reserve space to hold return value (if necessary)
- Reserve space for activation record
- Evaluate arguments
- Jump to callee's start address

Prologue sequence:

- Completes construction of callee's environment
- Reserve space for locals

Calling Convention

Epilogue sequence:

- Evaluate return value
- Store return value in caller's environment
- Start dismantling callee's environment
- Release space for locals
- Restores caller's environment
- Jump to return address

Calling Convention

Epilogue sequence:

- Evaluate return value
- Store return value in caller's environment
- Start dismantling callee's environment
- Release space for locals
- Restores caller's environment
- Jump to return address

Postreturn sequence:

- Reverts the precall sequence actions
- Release callee's activation record

CiviC Procedure Call Convention by Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition:
```
int foo( int x, bool y)
{ int i=2*x; return i-x;}
```

CiviC Procedure Call Convention by Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

Generated code:

```
isr                 // initiate subroutine call
iload 0             // assuming a is at index 0 of current frame
iloadg 2            // assuming b is at index 2 of global frame
iadd                // a+b
bloadc_t            // load true
jsr 2 123           // jump to subroutine, #args, #offset
          esr 1     // enter subroutine, #locals
          iloadc 0  // load constant at index 0 of constant pool
          iload 0   // load x from current frame
          imul      // 2*x
          istore 2  // store to i in current frame
          iload 2   // load i from current frame
          iload 0   // load x from current frame
          isub      // i-x
          ireturn   // save integer result
                    // proceed at store address
istore 0            // store result
```

CiviC Procedure Call Convention

Instructions:

- isr "initiate subroutine call"
  allocate new activation block, save FP

CiviC Procedure Call Convention

Instructions:

- isr "initiate subroutine call"
  allocate new activation block, save FP
- jsr *#args offset* "jump to subroutine"
  save PC, set new FP, jump to PC+offset

CiviC Procedure Call Convention

Instructions:

- isr “initiate subroutine call”
  allocate new activation block, save FP
- jsr *#args offset* “jump to subroutine”
  save PC, set new FP, jump to PC+offset
- esr *#locals* “enter subroutine”
  get space for locals

CiviC Procedure Call Convention

Instructions:

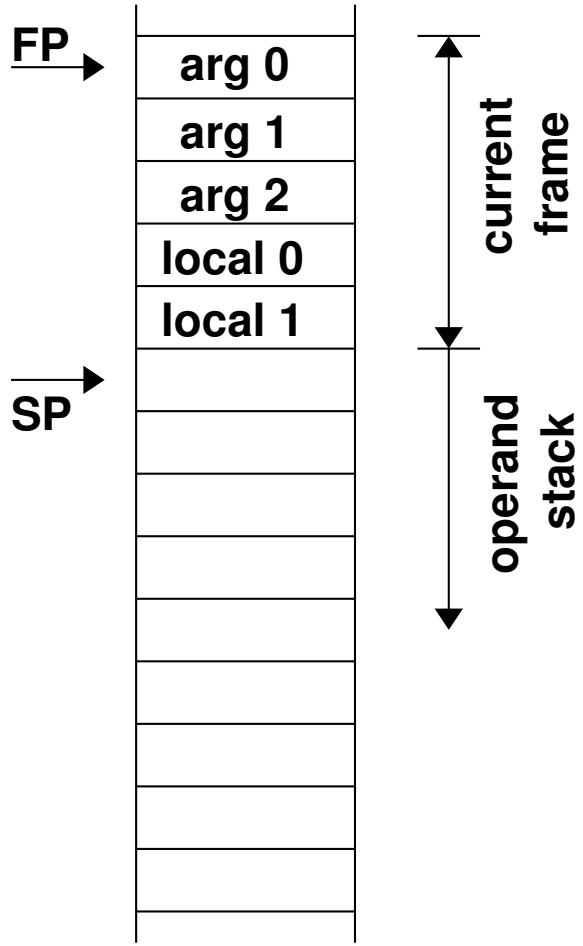
- isr “initiate subroutine call”
  allocate new activation block, save FP
- jsr *#args offset* “jump to subroutine”
  save PC, set new FP, jump to PC+offset
- esr *#locals* “enter subroutine”
  get space for locals
- ireturn, freturn, breturn, return
  save result, reset FP, jump to return address

CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```

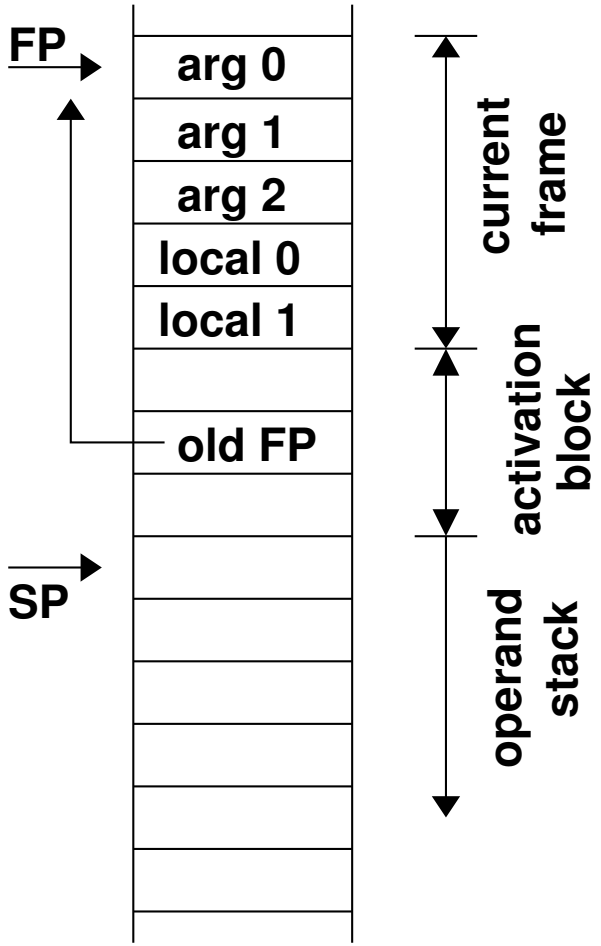


CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```

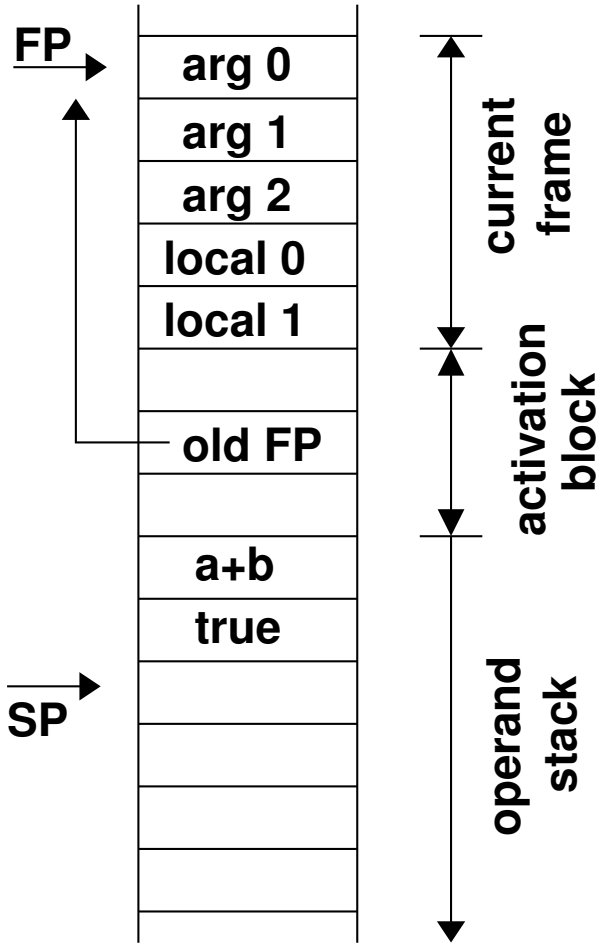


CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```

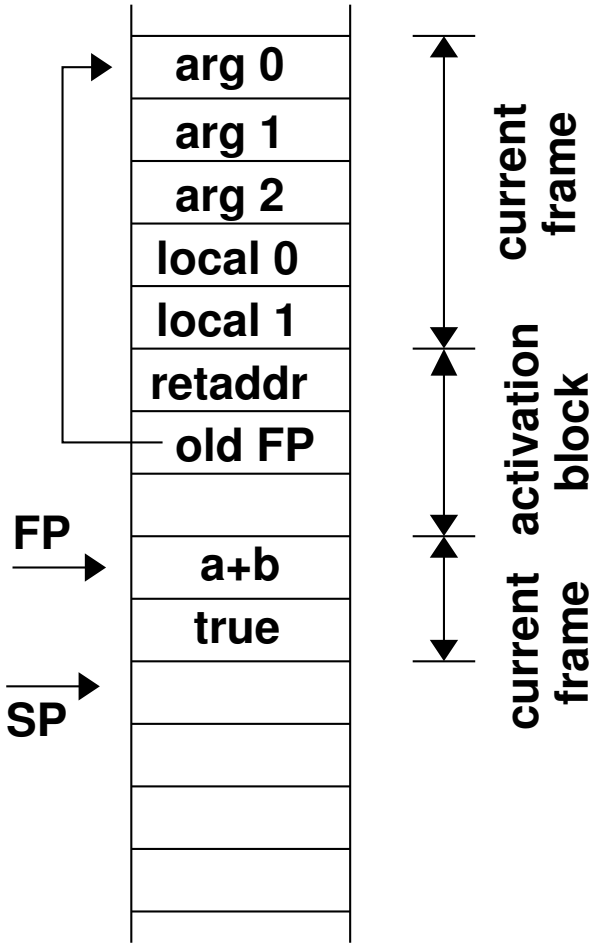


CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```

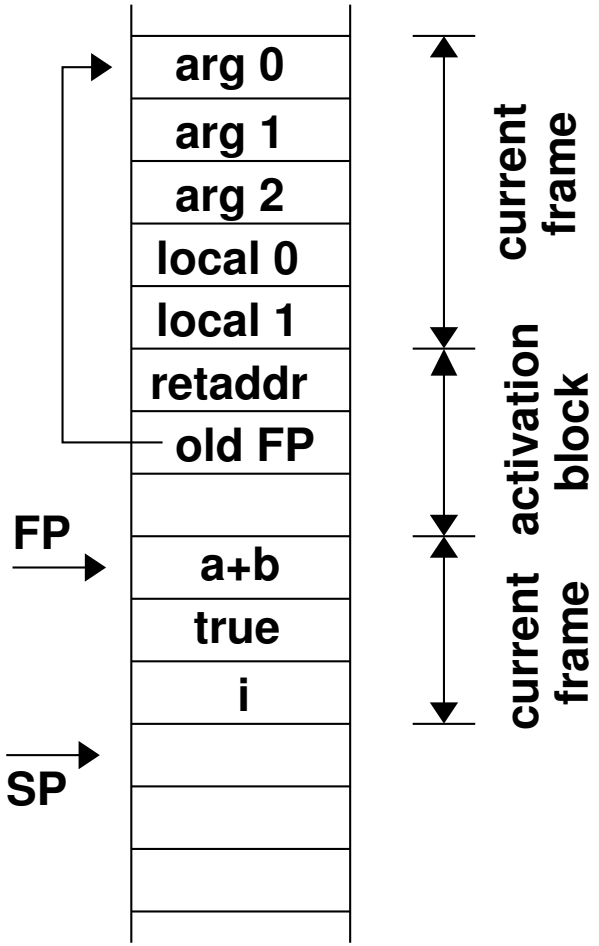


CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```

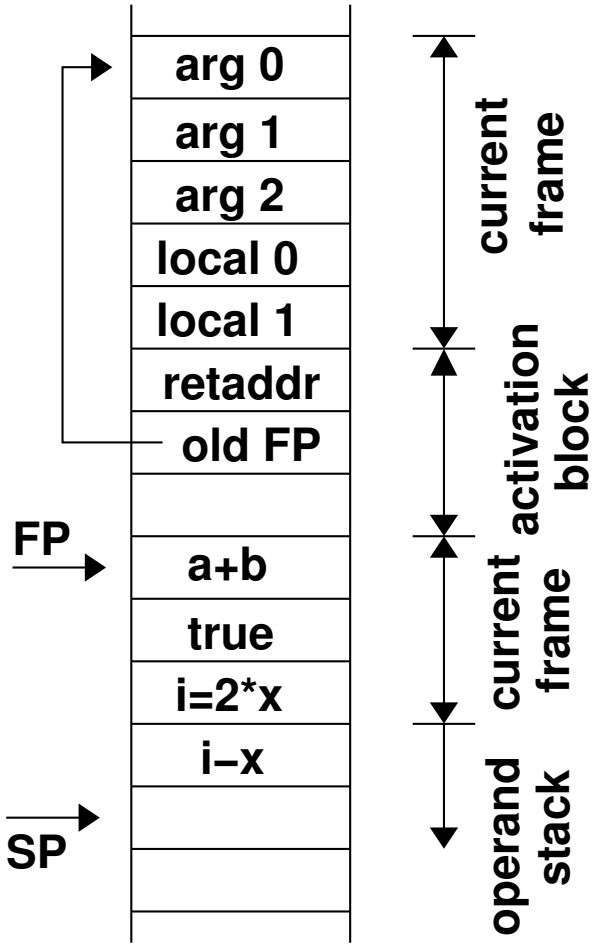


CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```

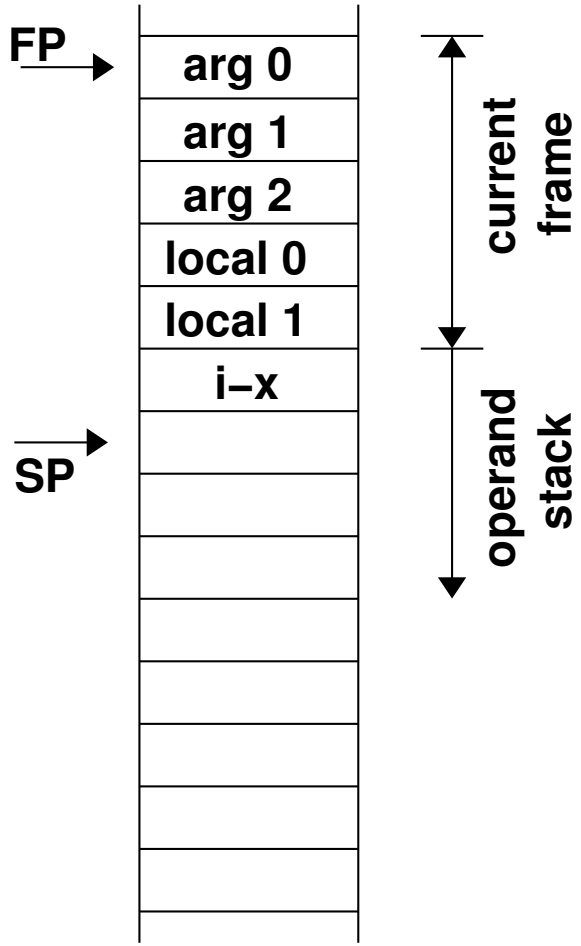


CiviC Procedure Call Convention Example

Example:

- Call site: `a = foo( a+b, true);`
- Definition: `int foo( int x, bool y)`
  `{ int i=2*x; return i-x;}`

```
isr
iload 0
iloadg 2
iadd
bloadc_t
jsr 2 123
      esr 1
      iloadc 0
      iload 0
      imul
      istore 2
      iload 2
      iload 0
      isub
      ireturn
istore 0
```



Nested Functions

Example:

```
int foo( int a)
{
  int b = 1;
  int f( int x) {
    return x + b;
  }
  int g( int x) {
    return f( x - b);
  }
  return g( b);
}
```

Nested Functions

Example:

```
int foo( int a)
{
  int b = 1;
  int f( int x) {
    return x + b;
  }
  int g( int x) {
    return f( x - b);
  }
  return g( b);
}
```

Insight:

- It's not good enough to access the current frame only
- We need access to all visible frames at any time
- Lexical scoping defines what is visible and what not

Nested Functions

Example:

```
int foo( int a)
{
  int b = 1;
  int f( int x) {
    return x + b;
  }
  int g( int x) {
    return f( x - b);
  }
  return g( b);
}
```

Insight:

- It's not good enough to access the current frame only
- We need access to all visible frames at any time
- Lexical scoping defines what is visible and what not

Question:

- Impact on code generation ?

Extended Context Analysis

```
int foo( int a)
{
  int b = 1;
  int f( int x) {
    return x + b;
  }
  int g( int x) {
    return f( x - b);
  }
  return g( b);
}
```

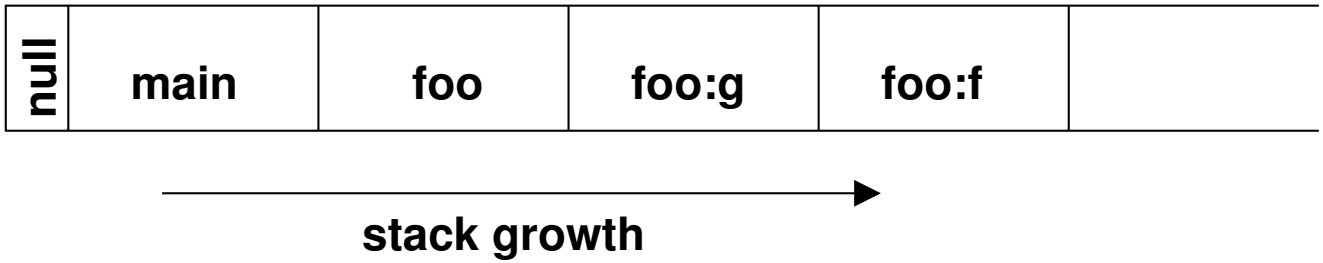
Context Analysis

```
int foo( int a<0>)
{
  int b<1> = 1;
  int f( int x<0>) {
    return x<0,0> + b<1,1>;
  }
  int g( int x<0>) {
    return f<0>( x<0,0> - b<1,1>);
  }
  return g<-1>( b<0,1>);
}
```

- Parameters/local declarations: current frame index
- Variables:
  - relative frame identifier
  - index in that frame
- Function applications: relative nesting level of definition
  - 0: same level
  - -1: one level down
  - n: n levels up

Static vs Dynamic Link

Extended view on stack:

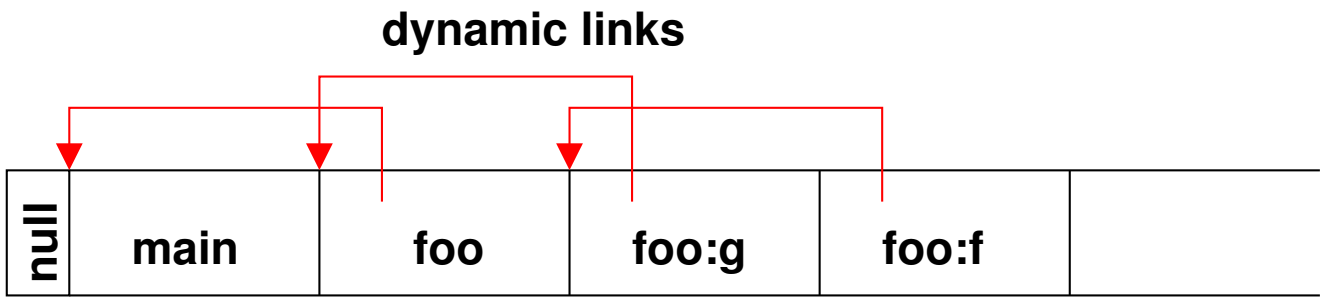


Stack structure:

- Stack frames piled one on top of the other

Static vs Dynamic Link

Extended view on stack:

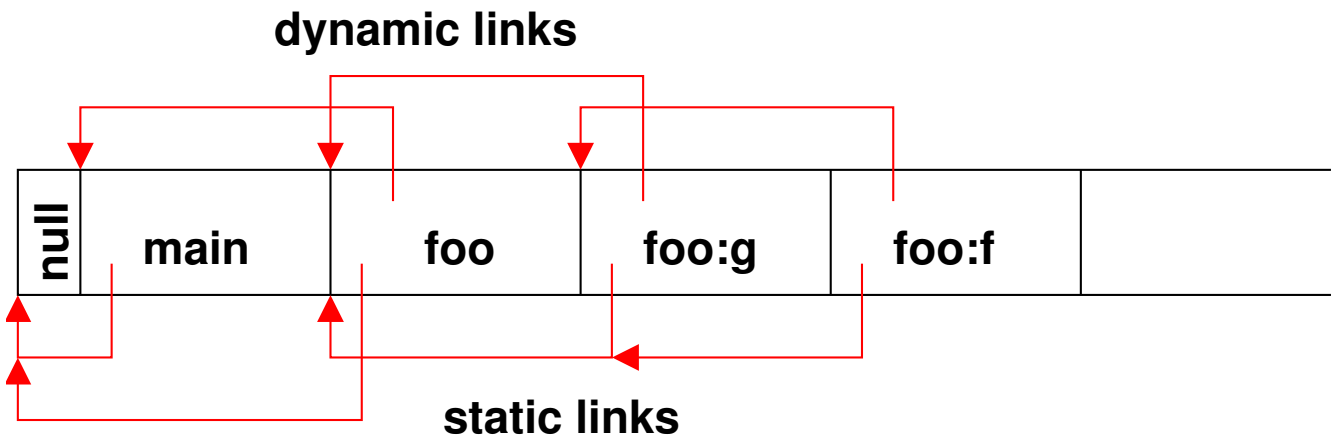


Stack structure:

- Stack frames piled one on top of the other
- Dynamic link identifies previous stack frame

Static vs Dynamic Link

Extended view on stack:

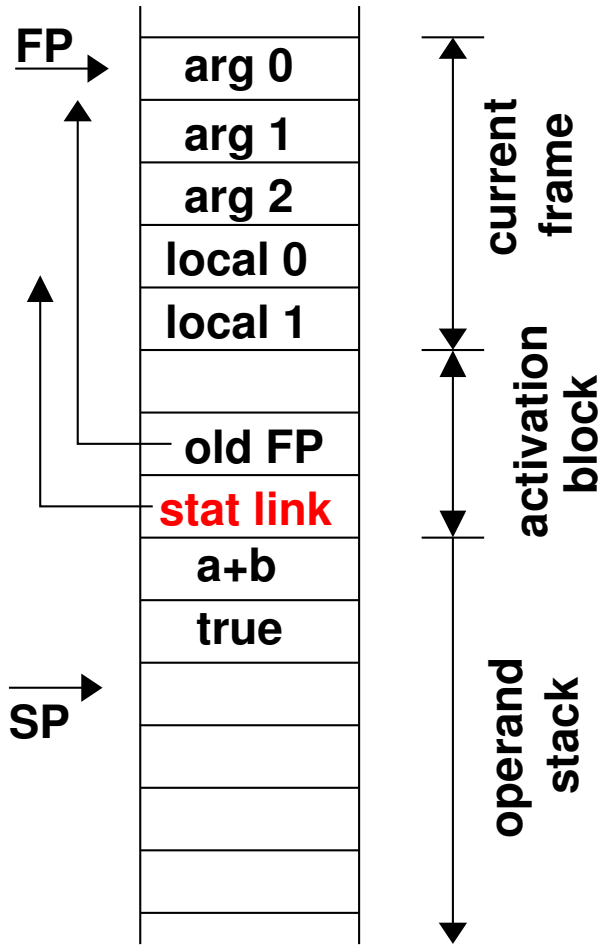


Stack structure:

- Stack frames piled one on top of the other
- Dynamic link identifies previous stack frame
- Static link identifies stack frame of lexically enclosing context

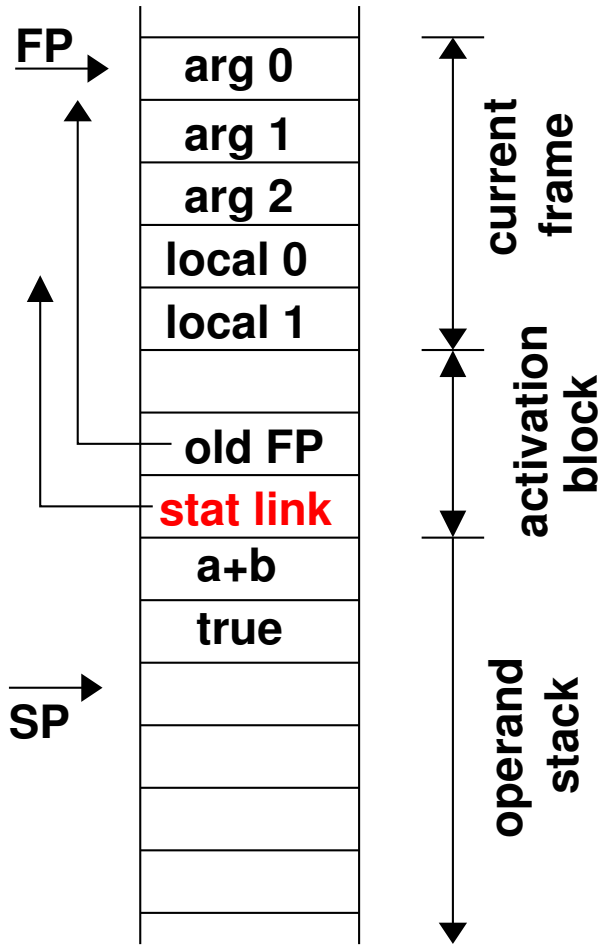
Creating the Static Link: 4 Cases

- Callee is on same level as caller
  - execution environment remains
  - callee's static link is caller's static link
  - `isr` "init subroutine call"



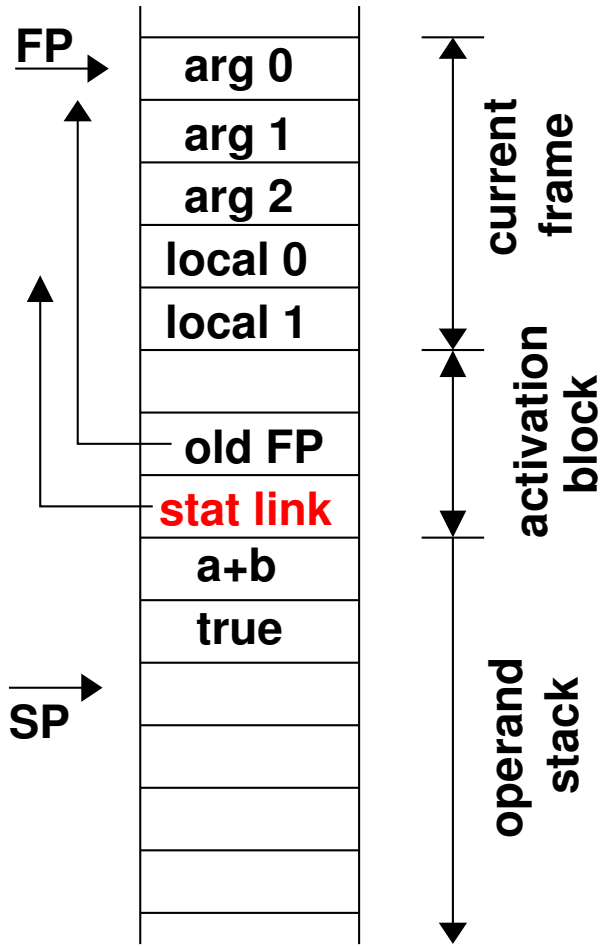
Creating the Static Link: 4 Cases

- Callee is on same level as caller
  - execution environment remains
  - callee's static link is caller's static link
  - `isr` "init subroutine call"
- Callee local to caller
  - dive into local execution environment
  - callee's static link is caller's frame
  - `isrl` "init subroutine call local"



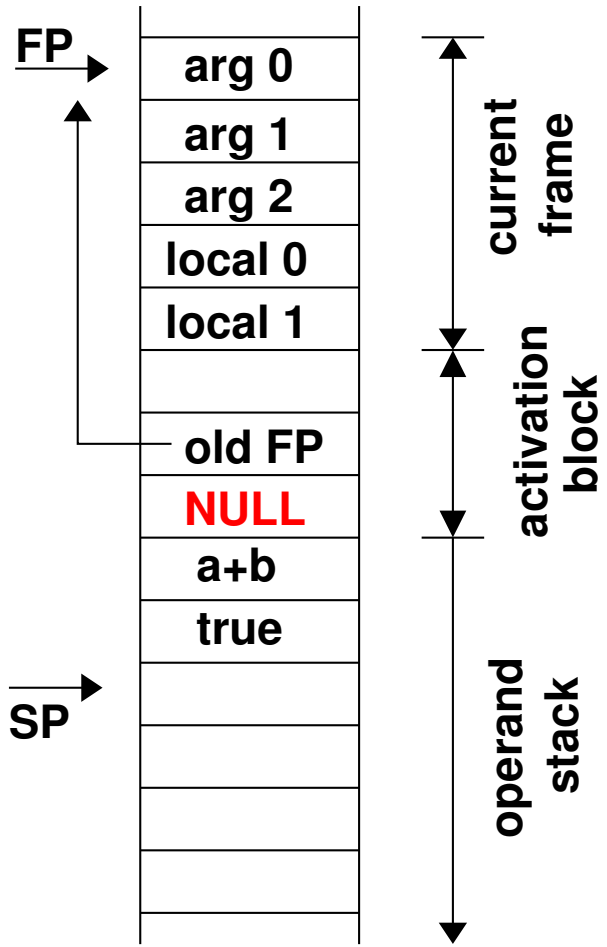
Creating the Static Link: 4 Cases

- Callee is on same level as caller
  - execution environment remains
  - callee's static link is caller's static link
  - `isr` "init subroutine call"
- Callee local to caller
  - dive into local execution environment
  - callee's static link is caller's frame
  - `isrl` "init subroutine call local"
- Callee above caller
  - exit caller's execution environment
  - follow static link n times
  - `isrn` n "init subroutine call n"



Creating the Static Link: 4 Cases

- Callee is on same level as caller
  - execution environment remains
  - callee's static link is caller's static link
  - `isr` "init subroutine call"
- Callee local to caller
  - dive into local execution environment
  - callee's static link is caller's frame
  - `isrl` "init subroutine call local"
- Callee above caller
  - exit caller's execution environment
  - follow static link n times
  - `isrn` n "init subroutine call n"
- Callee top-level
  - no local environment
  - set static link to null
  - `isrg` "init subroutine call global"



Using the Static Link: 2 Cases

Variable defined locally:

- Use offset from current FP
- `iload` *`offset`* , ...

Using the Static Link: 2 Cases

Variable defined locally:

- Use offset from current FP
- `iload` *`offset`* , ...

Variable defined in frame above:

- Follow static link given number of times
- Then use offset from into appropriate frame
- `iloadn` *`levels offset`* , ...

Calling External Functions

Very similar to internal functions:

- Set up activation block like for top-level function
- Evaluate argument expressions
- Use instruction `jsre` *index*
  "jump to external subroutine"
- Index points to runtime constant pool
- Constant pool holds absolute address of subroutine

Instruction Selection

Observation:

- Code generation is a non-unique mapping from IR to target language
- Different choices have different impacts on "success metrics"
  - code size
  - execution speed
- How do we organise code generation?

  $\rightarrow$ instruction selection

Instruction Selection

Observation:

- Code generation is a non-unique mapping from IR to target language
- Different choices have different impacts on "success metrics"
  - code size
  - execution speed
- How do we organise code generation?
  $\rightarrow$ instruction selection

Two fundamental approaches:

- Naive instruction selection + peephole optimisation
- Cost-model based sophisticated instruction selection
- In practice: combinations thereof

End of Compilerbouw Course

The End: Questions ?