Part II: LLVM Intermediate Representation

陳韋任 (Chen Wei-Ren)
chenwj@iis.sinica.edu.tw

Computer Systems Lab, Institute of Information Science,
Academia Sinica, Taiwan (R.O.C.)

April 10, 2013

Outline

- LLVM IR and IR Builder
- From LLVM IR to Machine Code
- Machine Code Layer

C Source Code

```c
int fib(int x) {
  if (x <= 2)
    return 1;

  return fib(x - 1) + fib(x - 2);
}
```

```
$ clang -emit-llvm -O0 -S fib.c -o fib.ll
```

C → LLVM IR

```
define i32 @fib(i32 %x) nounwind uwtable {
entry:
  ...
  br i1 %cmp, label %if.then, label %if.end

if.then:      ; preds = %entry
  store i32 1, i32* %retval
  br label %return

if.end:       ; preds = %entry
  ...
  store i32 %add, i32* %retval
  br label %return

return:       ; preds = %if.end, %if.then
  %3 = load i32* %retval
  ret i32 %3
}
```

LLVM IR 1/2

LLVM defines its own IR to represent code in the compiler. LLVM IR has following characteristics:

- SSA form: Every definition creates a new variable.
- Infinite virtual registers
- Phi node: Determine the value of the variable comes from which control flow.

LLVM IR has three different forms:

- in-memory compiler IR
- on-disk bitcode representation (*.bc)
- on-disk human readable assembly (*.ll)

LLVM IR 2/2

```
define i32 @bar(i32 %x) nounwind {
  %1 = alloca i32, align 4
  store i32 %x, i32* %1, align 4
  %2 = load i32* %1, align 4
  %3 = icmp ne i32 %2, 0
  br i1 %3, label %4, label %7

; <label>:4                                      ; preds = %0
  %5 = call i32 @foo()
  %6 = add nsw i32 %5, 1
  br label %9

; <label>:7                                      ; preds = %0
  %8 = call i32 @baz()
  br label %9

; <label>:9                                      ; preds = %7, %4
  %10 = phi i32 [ %6, %4 ], [ %8, %7 ]
  ret i32 %10
}
```

SSA Form & Infinite Virtual Register

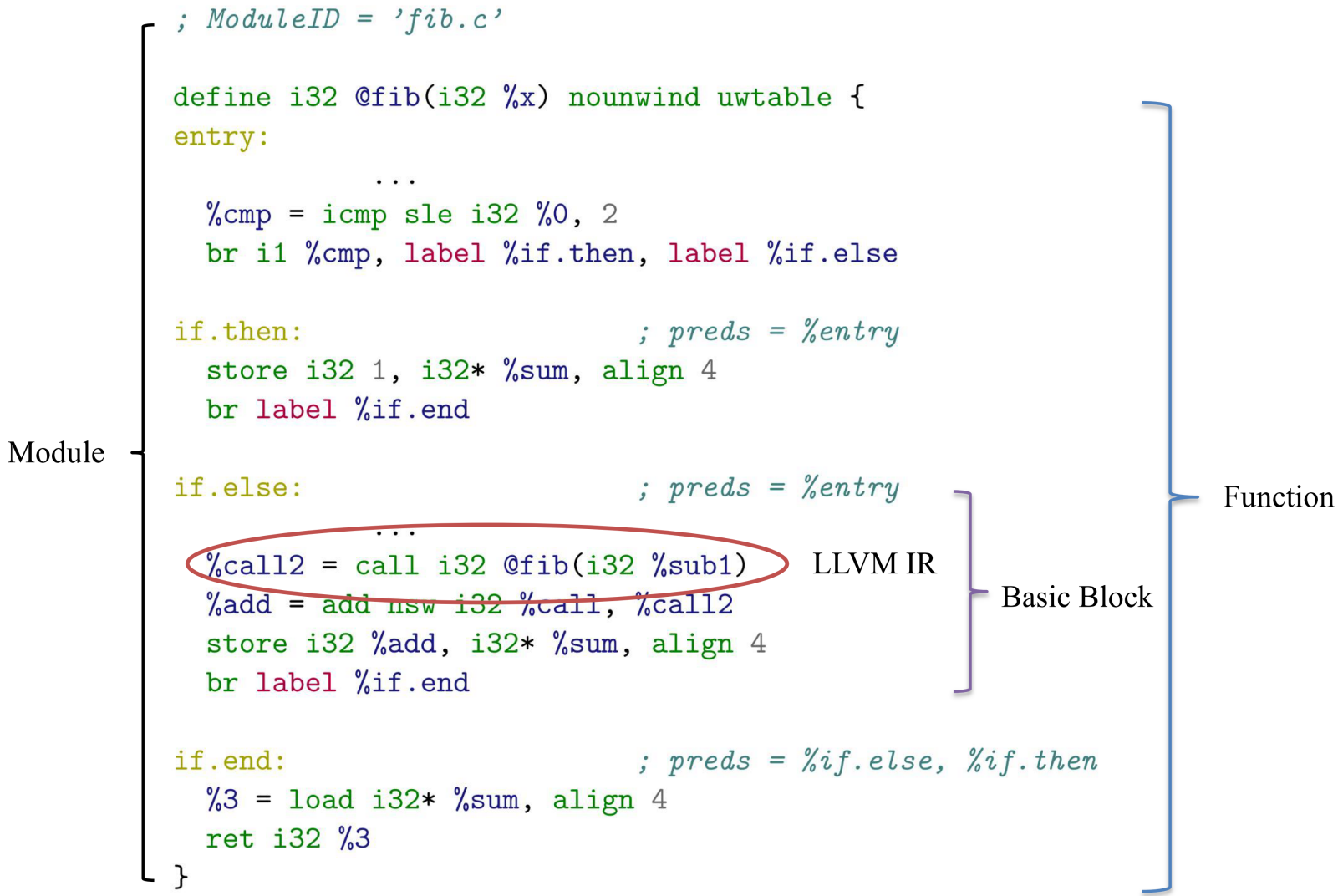
Phi Node

IR Builder 1/3

LLVM IRBuilder is a class template that can be used as a convenient way to create LLVM instructions.

- Module: The top level container of all other LLVM IR. It can contains global variables, functions, etc...
- Function: A function consists of a list of basic blocks and a list of arguments.
- Basic block: A container of instructions that can be executed sequentially.
- LLVMContext: A container of *global* state in LLVM.

IR Builder 2/3



IR Builder 3/3

```
// Create LLVMContext for latter use.
LLVMContext Context;

// Create some module to put our function into it.
Module *M = new Module("test", Context);

// Create function inside the module
Function *Add1F =
  cast<Function>(M->getOrInsertFunction("add1", Type::getInt32Ty(Context),
                                        Type::getInt32Ty(Context),
                                        (Type *)0));

// Add a basic block to the function.
BasicBlock *BB = BasicBlock::Create(Context, "EntryBlock", Add1F);

// Create a basic block builder. The builder will append instructions
// to the basic block 'BB'.
IRBuilder<> builder(BB);

// ... prepare operands for Add instrcution. ...

// Create the add instruction, inserting it into the end of BB.
Value *Add = builder.CreateAdd(One, ArgX);
```

Let's start with a simple HelloWorld!

We will stick with LLVM 3.2 Release!

Step 1. Create Module

```
#include "llvm/LLVMContext.h"
#include "llvm/Module.h"
#include "llvm/IRBuilder.h"

int main()
{
  llvm::LLVMContext& context = llvm::getGlobalContext();
  llvm::Module* module = new llvm::Module("top", context);
  llvm::IRBuilder<> builder(context);

  module->dump( );
}
```

```
; ModuleID = 'top'
```

Step 2. Create Function

```
            ...

  llvm::FunctionType *funcType =
      llvm::FunctionType::get(builder.getInt32Ty(), false);
  llvm::Function *mainFunc =
      llvm::Function::Create(funcType,
                             llvm::Function::ExternalLinkage,
                             "main", module);

  module->dump( );
}
```

```
; ModuleID = 'top'

declare i32 @main()
```

Step 3. Create Basic Block

```
            ...

  llvm::BasicBlock *entry =
      llvm::BasicBlock::Create(context, "entrypoint", mainFunc);
  builder.SetInsertPoint(entry);

  module->dump( );
}
```

```
; ModuleID = 'top'

define i32 @main() {
entrypoint:
}
```

Step 4. Add "hello world" into the Module

```
            ...

  llvm::Value *helloWorld
      = builder.CreateGlobalStringPtr("hello world!\n");

  module->dump( );
}
```

```
; ModuleID = 'top'

@0 = private unnamed_addr constant [14 x i8] c"hello world!\0A\00"

define void @main() {
entrypoint:
}
```

Step 5. Declare "puts" method

```
          ...

  std::vector<llvm::Type *> putsArgs;
  putsArgs.push_back(builder.getInt8Ty()->getPointerTo());
  llvm::ArrayRef<llvm::Type*>  argsRef(putsArgs);

  llvm::FunctionType *putsType =
    llvm::FunctionType::get(builder.getInt32Ty(), argsRef, false);
  llvm::Constant *putsFunc
    = module->getOrInsertFunction("puts", putsType);

  module->dump();
}
```

```
; ModuleID = 'top'

@0 = private unnamed_addr constant [14 x i8] c"hello world!\0A\00"

define void @main() {
entrypoint:
}

declare i32 @puts(i8*)
```

Step 6. Call "puts" with "hello world"

```
            ...

  builder.CreateCall(putsFunc, helloWorld);
  builder.CreateRetVoid();

  module->dump();
}
```

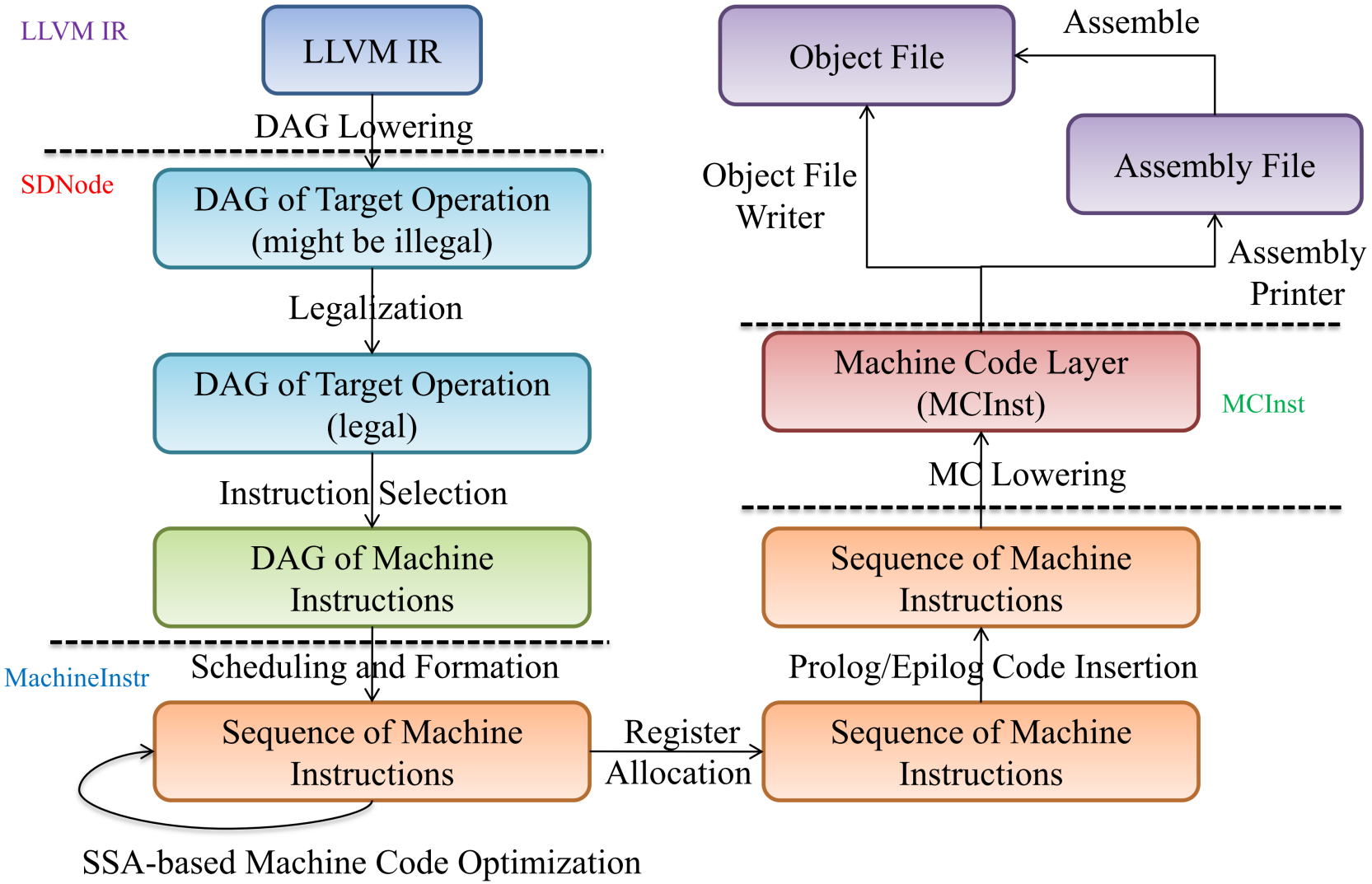
```
; ModuleID = 'top'

@0 = private unnamed_addr constant [14 x i8] c"hello world!\0A\00"

define void @main() {
entrypoint:
  %0 = call i32 @puts(i8* getelementptr inbounds ([14 x i8]* @0, i32 0,
  ret void
}

declare i32 @puts(i8*)
```

Lowering Flow Overview



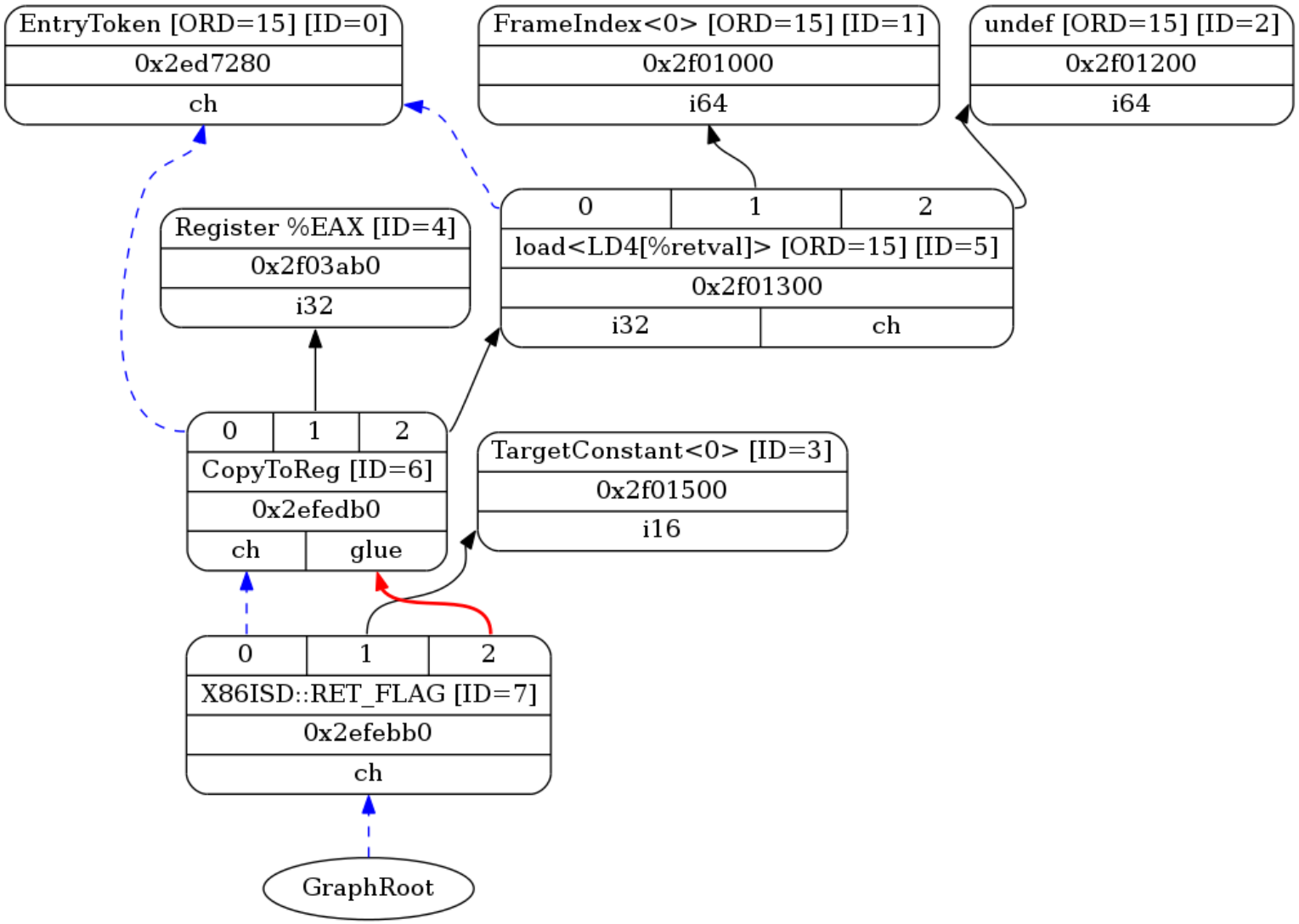
LLVM → DAG

We first translate a sequence of LLVM IR into a SelectionDAG to make instruction selection easier. DAG also make optimization efficiently, CSE (Common Subexpression Elimination), for example.

For each use and def in the LLVM IR, an SDNode (SelectionDAG Node) is created. The SDNode could be intrinsic SDNode or target defined SDNode.

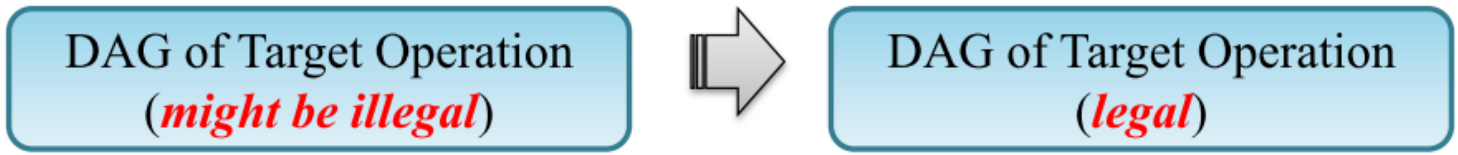
```
$ llc -view-dag-combine1-dags fib.ll
$ dot -Tpng dag.fib.dot -o dag.fib.png
```

LLVM → DAG



legalize input for fib:return

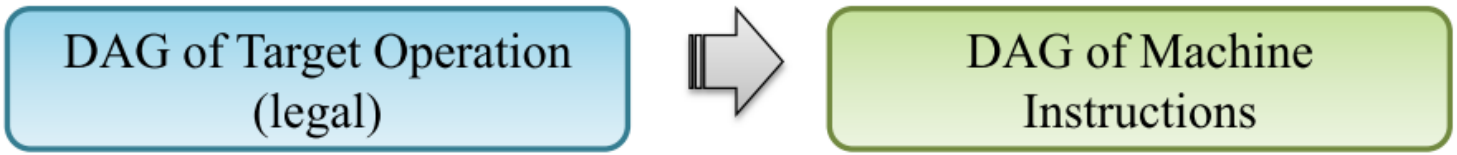
Legalize



Target machine might not support some of LLVM types or operations, we need to legalize previous DAG.

```
$ llc -view-dag-combine2-dags fib.ll
```

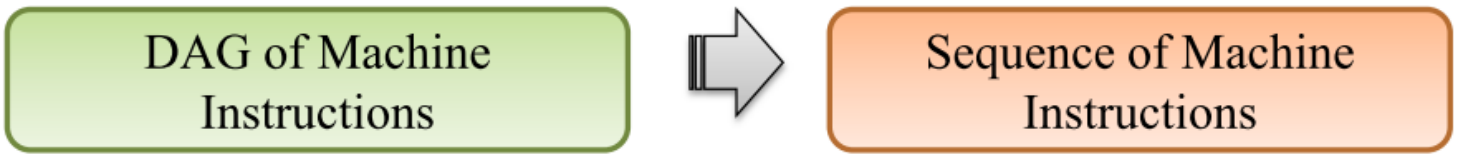
IR DAG → Machine DAG



The instruction selector match IR SDNode into MachineSDNode. MachineSDNode represents everything that will be needed to construct a MachineInstr (MI).

```
# displays the DAG before instruction selection
$ llc -view-isel-dags fib.ll
# displays the DAG before instruction scheduling
$ llc -view-sched-dags fib.ll
```

Scheduling and Formation



Machine DAG are deconstructed and their instruction are turned into list of MachineInstr (MI). The scheduler has to decide in what order the instructions (MI) are emitted.

```
$ llc -print-machineinstrs fib.ll
```

SSA-based Machine Code Optimization

Ths list of MachineInstr is still in SSA form. We can perform target-specific low-level optimization before doing register allocation. For example, LICM (Loop Invariant Code Motion).

- addMachineSSAOptimization (lib/CodeGen/Passes.cpp)

Register Allocation

All virtual registers are eliminated at this stage. The register allocator assigns physical register to each virtual register if possible. After register allocation, the list is not SSA form anymore.

Prologue/Epilogue Code Insertion

We insert prologue/epilogue code sequence into function beginning/end respectively. All abstract stack frame references are turned into memory address relative to the stack/frame pointer.

What We Have Now?

$$\text{C} \rightarrow \text{LLVM IR} \rightarrow \text{IR DAG} \rightarrow \text{Machine DAG} \rightarrow \text{MachineInstr (MI)}$$

Code Emission

This stage lowers MachineInstr (MI) to MCInst (MC).

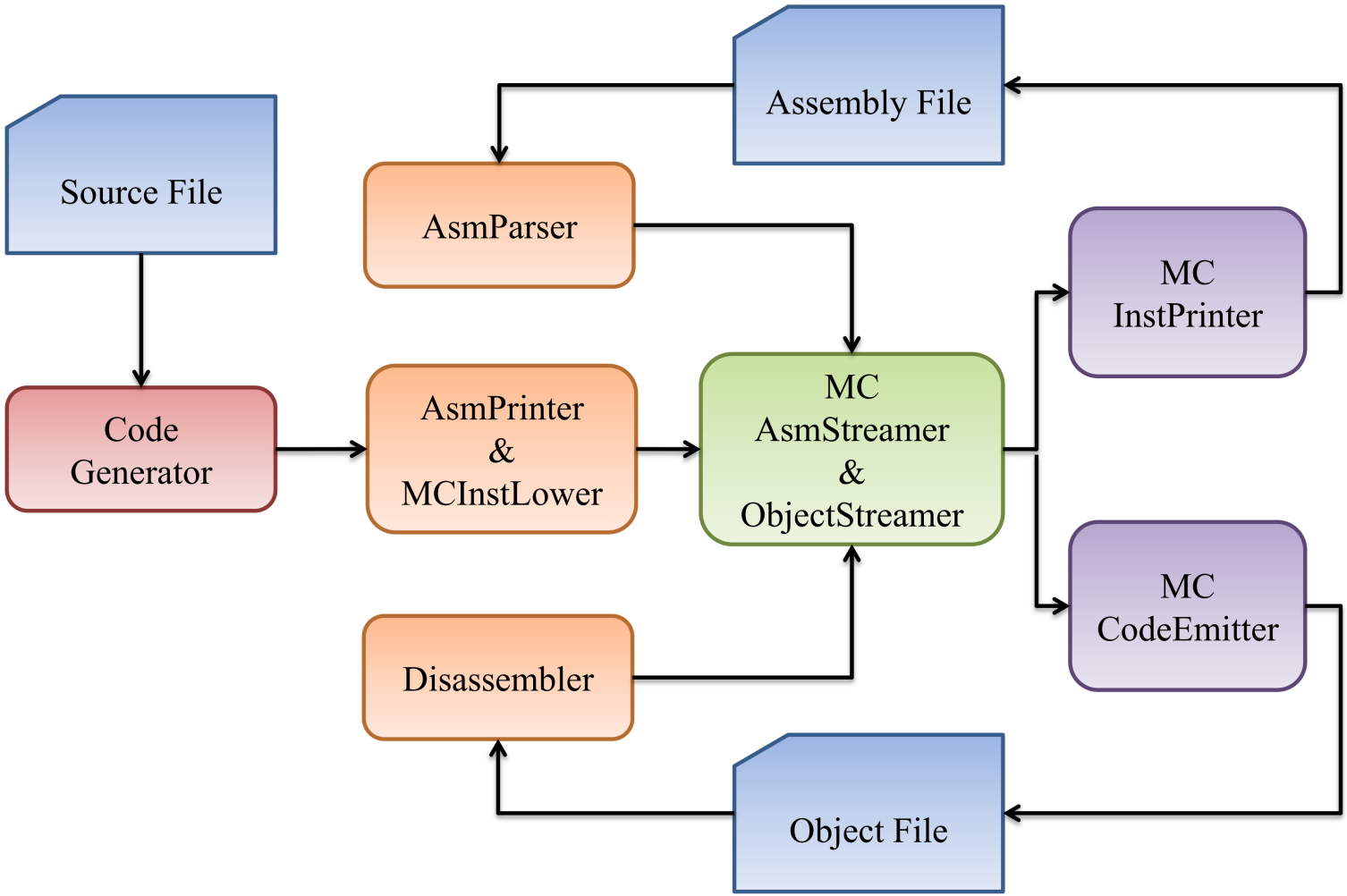
- MachineInstr (MI): Abstraction of target instruction, consists of opcode and operand only. We have MachineFunction and MachineBasicBlock as containers for MI.
- MCInst (MC): Abstraction of object file. No function, global variable ...etc, but only label, directive and instruction.

Machine Code (MC) Layer

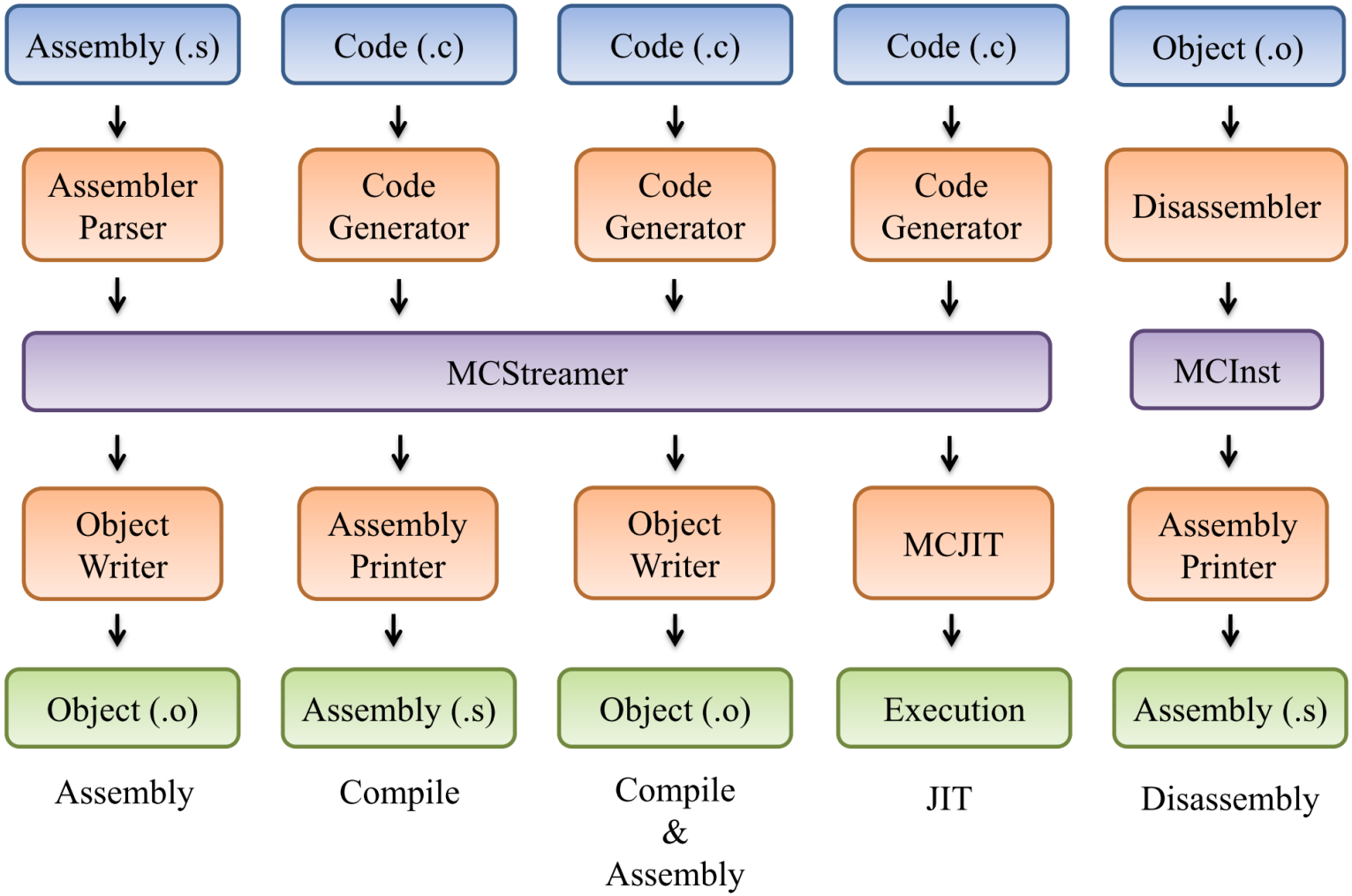
MC layer works at the level of abstraction of object file.

```
$ llc -show-mc-inst fib.ll
$ llc -show-mc-encoding fib.ll
```

MC Layer Flow



What Is The Role of MC?



Assembler Relaxation

Assmbler can adjust the instructions for two purposes:

- Correctness: If the operand of instruction is out of bound, either expand the instruction or replace it by a longer one.
- Optimization: Replace expensive instruction with a cheaper one.

This process is called *relaxation*.

Assembler Relaxation Example 1/4

```
extern int bar(int);

int foo(int num) {
  int t1 = num * 17;
  if (t1 > bar(num))
    return t1 + bar(num);
  return num * bar(num);
}
```

```
$ clang -S -O2 relax.c
$ clang -c relax.s
$ objdump -d relax.o
```

Assembler Relaxation Example 2/4

```
0000000000000000 <foo>:
  1a:        e8 00 00 00 00        callq  1f <foo+0x1f>
  1f:        45 39 f7              cmp    %r14d,%r15d
  22:        7e 05                 jle    29 <foo+0x29>
  24:        44 01 f8              add    %r15d,%eax
  27:        eb 03                 jmp    2c <foo+0x2c>
  29:        0f af c3              imul   %ebx,%eax
  2c:        48 83 c4 08           add    $0x8,%rsp
  30:        5b                    pop    %rbx
  31:        41 5e                 pop    %r14
  33:        41 5f                 pop    %r15
  35:        5d                    pop    %rbp
  36:        c3                    retq
```

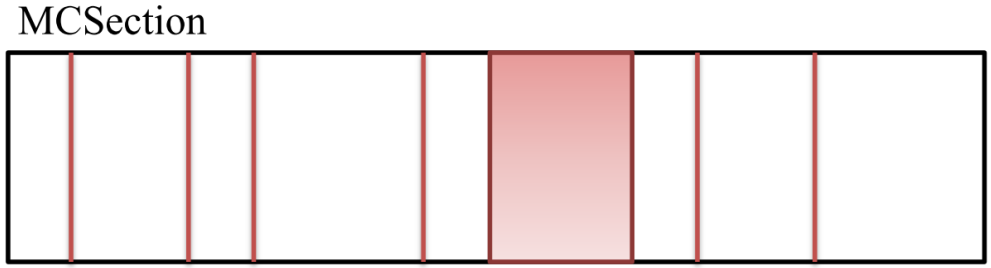
Assembler Relaxation Example 3/4

```
        callq    bar
        cmpl     %r14d, %r15d
        jle      .LBB0_2
        .fill 124, 1, 0x90 # nop
# BB#1:
        addl     %r15d, %eax
        jmp      .LBB0_3
.LBB0_2:
        imull    %ebx, %eax
.LBB0_3:
        addq     $8, %rsp
        popq     %rbx
        popq     %r14
        popq     %r15
        popq     %rbp
        ret
```

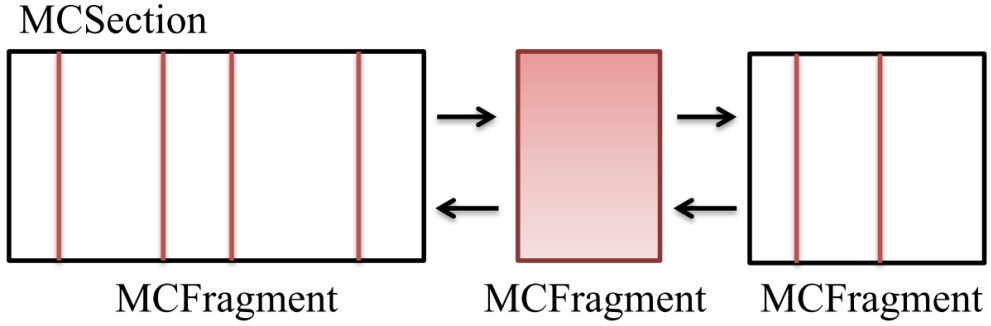
Assembler Relaxation Example 4/4

```
0000000000000000 <foo>:
  1a:        e8 00 00 00 00          callq  1f <foo+0x1f>
  1f:        45 39 f7                cmp    %r14d,%r15d
  22:        0f 8e 81 00 00 00       jle    a9 <foo+0xa9>
  28:        90                      nop
  a3:        90                      nop
  a4:        44 01 f8                add    %r15d,%eax
  a7:        eb 03                   jmp    ac <foo+0xac>
  a9:        0f af c3                imul   %ebx,%eax
  ac:        48 83 c4 08             add    $0x8,%rsp
  b0:        5b                      pop    %rbx
  b1:        41 5e                   pop    %r14
  b3:        41 5f                   pop    %r15
  b5:        5d                      pop    %rbp
  b6:        c3                      retq
```

Section and Fragment



In order to make relaxation process easier, LLVM keeps instructions as a linked list of *Fragment* not a byte array.



- MCAssembler::layoutSectionOnce (lib/MC/MCAssembler.cpp)

Reference

- LLVM - Another Toolchain Platform
- Design and Implementation of a TriCore Backend for the LLVM Compiler Framework
- Life of an instruction in LLVM
- Assembler relaxation
- Instruction Relaxation
- Intro to the LLVM MC Project
- The LLVM Assembler & Machine Code Infrastructure
- Howto: Implementing LLVM Integrated Assembler
- The Architecture of Open Source Applications - Chapter LLVM
- The LLVM Target-Independent Code Generator

The End

Q & A

You can download the manuscript and related material from here.