PinaVM: a SystemC Front-End Based on an Executable Intermediate Representation

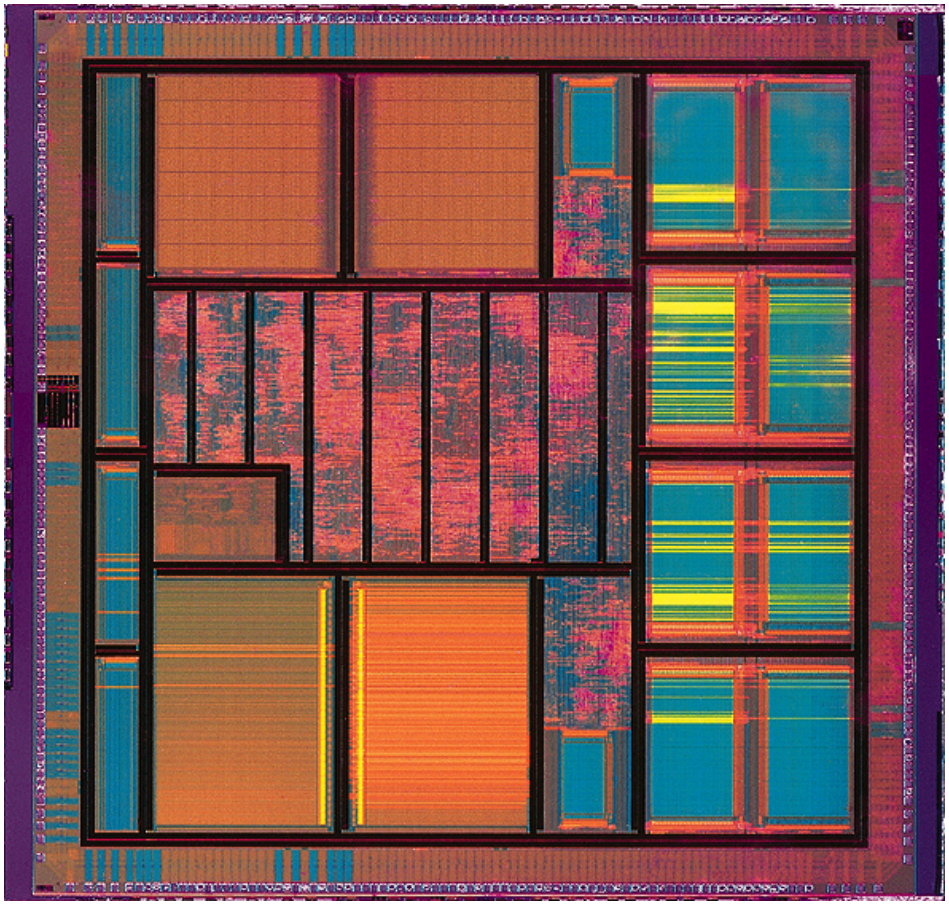
Kevin Marquet Matthieu Moy

Verimag (Grenoble INP)
Grenoble
France

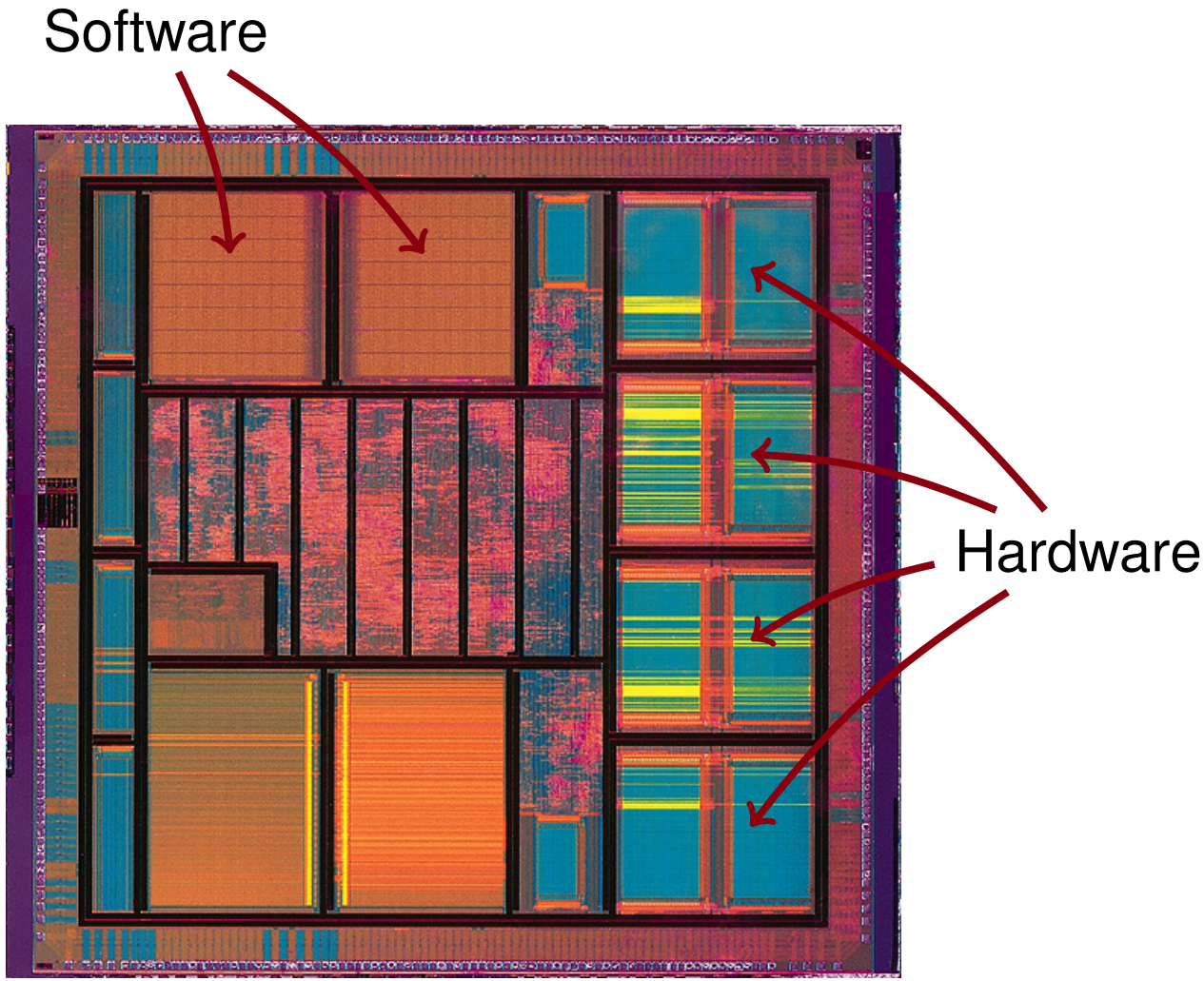
Emsoft, October 25th 2010

Summary

Modern Systems-on-a-Chip



Modern Systems-on-a-Chip

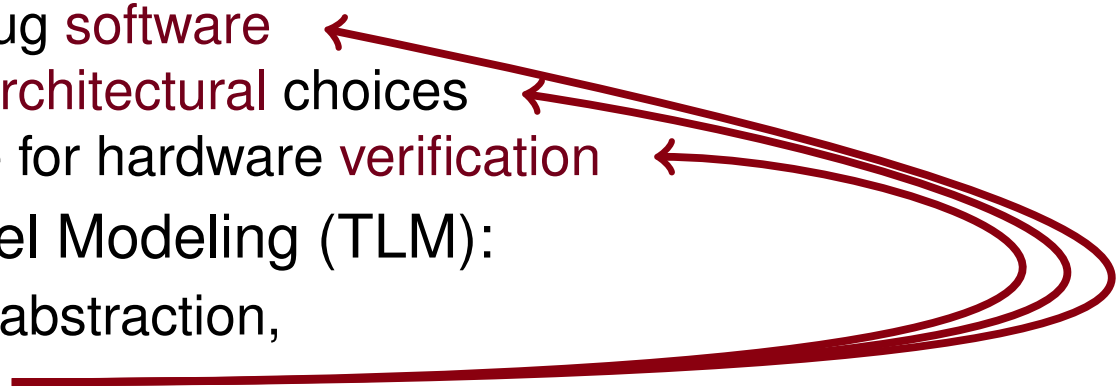


SystemC and Transaction-Level Modeling

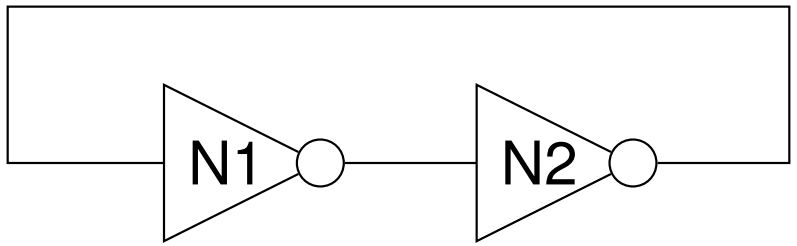
- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification

SystemC and Transaction-Level Modeling

- (Fast) simulation essential in the design-flow
  - To write/debug software
  - To validate architectural choices
  - As reference for hardware verification
- Transaction-Level Modeling (TLM):
  - high level of abstraction,
  - suitable for
- SystemC :
  - Industry-standard for high-level modeling (TLM, ...) of Systems-on-a-Chip,
  - Library for C++ (compile with `g++ -lsystemc...`)



SystemC: Simple Example



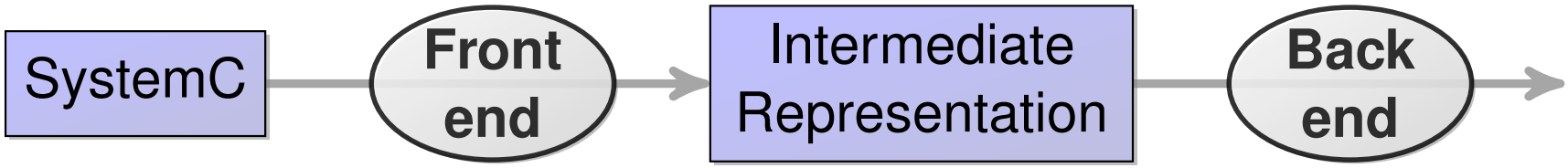

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;

    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    // Instantiate modules ...
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // ... and bind them together
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS);
    return 0;
}
```

Summary

SystemC Front-End

- In this talk: Front-end = “Compiler front-end” (AKA “Parser”)



Intermediate Representation = Architecture + Behavior

Applications of SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

Applications of SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

  $\Rightarrow$ No reference front-end available on `http://systemc.org/`

Applications of SystemC Front-Ends

- When you *don't* need a front-end:
  - Main application of SystemC: Simulation
  - Testing, run-time verification, monitoring...

  $\Rightarrow$ No reference front-end available on `http://systemc.org/`
- When you *do* need a front-end:
  - Symbolic formal verification, High-level synthesis
  - Visualization
  - Introspection
  - SystemC-specific Compiler Optimizations
  - Advanced debugging features

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)

2. Architecture is built at runtime, with C++ code

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   ⇝ Write a C++ front-end or reuse one (g++, clang, edg, . . .)
2. Architecture is built at runtime, with C++ code
   ⇝ Analyze elaboration phase or execute it

```
SC_MODULE(not_gate) {
    sc_in<bool> in;
    sc_out<bool> out;
    void compute (void) {
        // Behavior
        bool val = in.read();
        out.write(!val);
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

```
int sc_main(int argc, char **argv) {
    // Elaboration phase (Architecture)
    not_gate n1("N1");
    not_gate n2("N2");
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Challenges and Solutions with SystemC Front-Ends

1. C++ is complex (e.g. `clang` ≈ 200,000 LOC)
   ⇝ Write a C++ front-end or reuse one (g++, clang, edg, . . .)
2. Architecture is built at runtime, with C++ code
   ⇝ Analyze elaboration phase or execute it

```
SC_MODULE(no _gate) {
    sc_in<bo l> in;
    sc_out<b ol> out;
    void co ute (void) {
        //  havior
        in.read();
        );
    }

    SC_CTOR(not_gate) {
        SC_METHOD(compute);
        sensitive << in;
    }
};
```

Static Approaches

```
int sc_main(in  argc, char **argv) {
    // Elaborat  phase (Architecture)
    not_gate n1("
    not_g
    sc_signal<bool> s1, s2;
    // Binding
    n1.out.bind(s1);
    n2.out.bind(s2);
    n1.in.bind(s2);
    n2.in.bind(s1);

    // Start simulation
    sc_start(100, SC_NS); return 0;
}
```

Dynamic Approaches

Why Elaboration Phase Can Be Complex

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

Why Elaboration Phase Can Be Complex

- Static approach: cannot deal with such code
- Dynamic approach: can extract the architecture for individual instances of the system

```
int sc_main(int argc, char **argv) {
    int n = atoi(argv[1]);
    int m = atoi(argv[2]);
    Node array[n][m];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            array[i][j]
                = new Node(...);
            ...
        }
    }

    sc_start(100, SC_NS);
    return 0;
}
```

Existing SystemC front-ends

An attempt at a classification

	Static	Dynamic
Home-made parser	KaSCPar, sc2v, ParSyC, Scoot, SystemPerl. . .	
Existing parser	SystemCXML	DATE09, Pinapa, PinaVM

- Hard to classify: Quiny (purely dynamic approach)
- Commercial tools (closed, not detailed here): Synopsys, Semantic Design, NC-SystemC (Cadence)

Self-advertisement

FDL 2010 paper:
“A Theoretical and Experimental Review of SystemC Front-ends”

Summary

Before it started: Pinapa [Emsoft05]

- Pinapa's principle:
  - Use GCC's C++ front-end
  - Compile, dynamically load and execute the elaboration (`sc_main`)
- Pinapa's drawbacks:
  - Uses GCC's internals (hard to port to newer versions)
  - Hard to install
  - No separate compilation
  - Based on complex Abstract Syntax Tree (AST)
    (e.g. one construct for `for`, one for `while` and one for `do ... while`)
  - Ad-hoc match of SystemC constructs in AST

Static Single Assignment

- Non-SSA program

```
x = 42;
x = x + 1;
y = x;
```

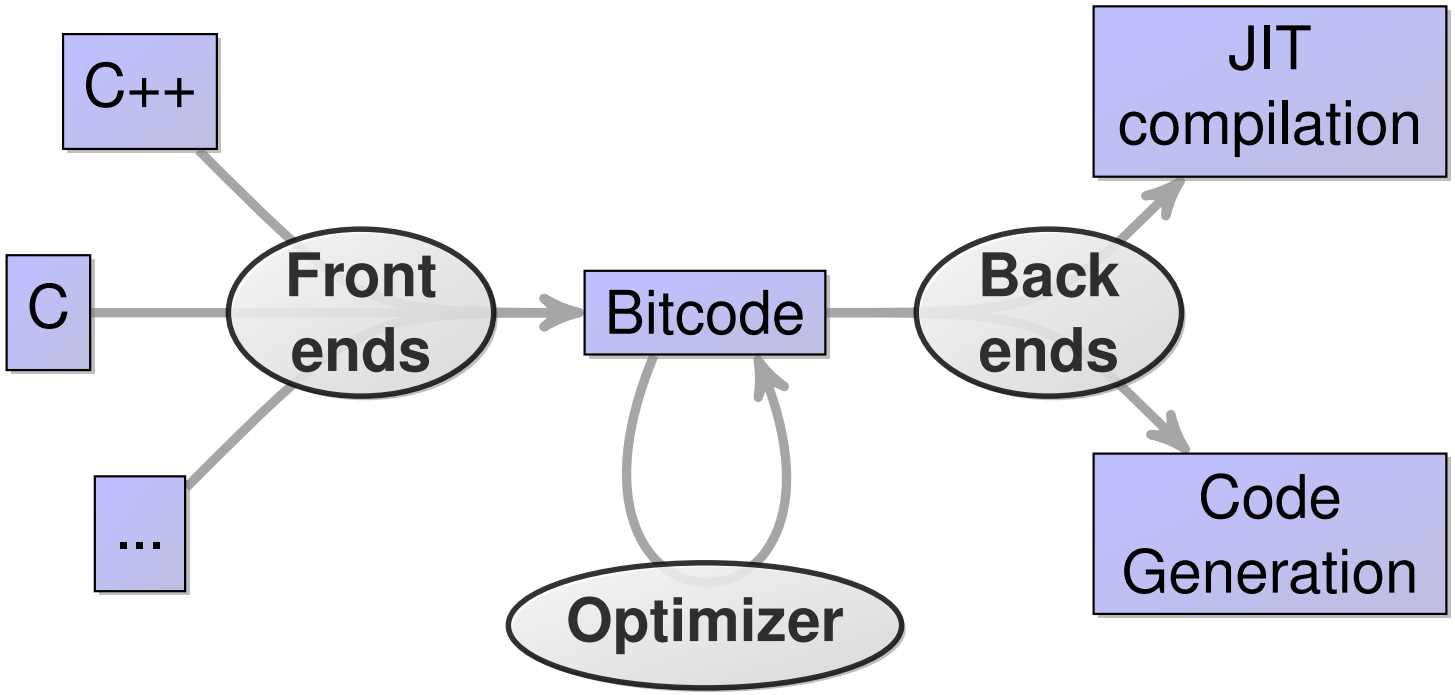
- SSA program

```
x1 = 42;
x2 = x1 + 1;
y = x2;
```

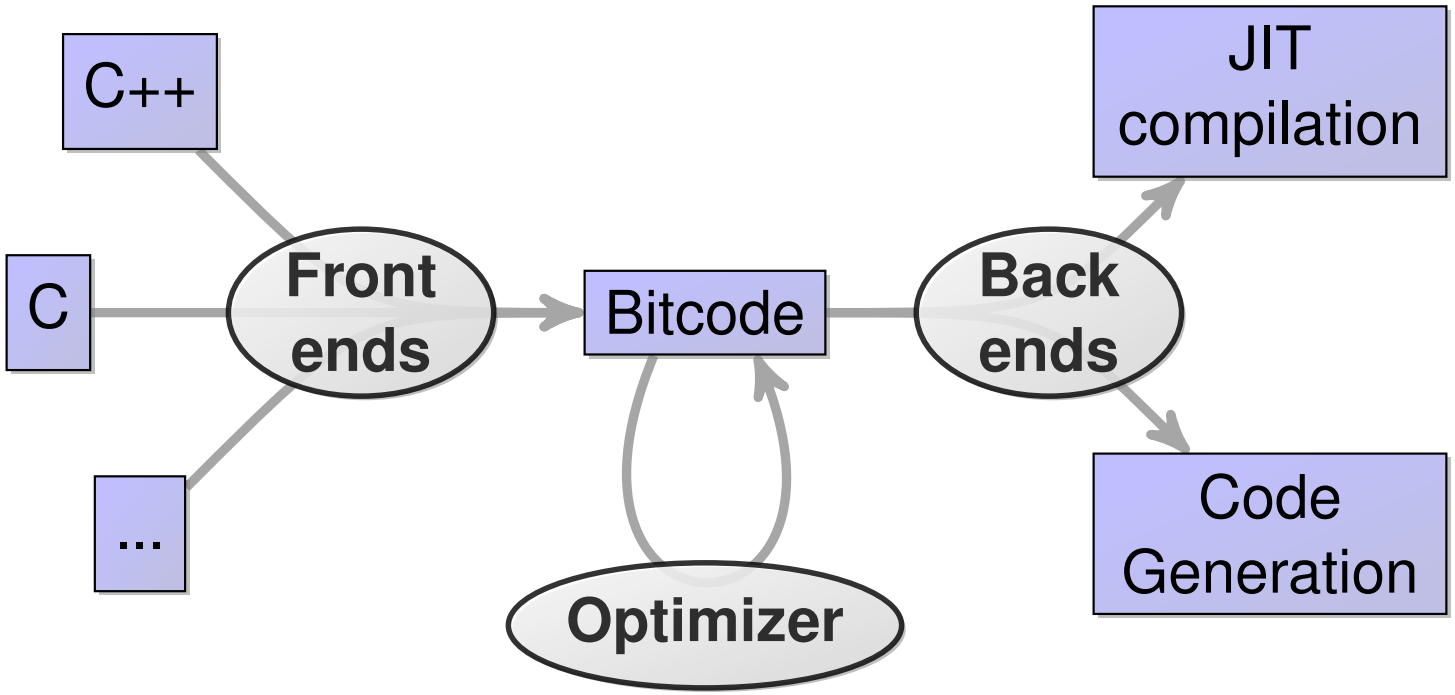
- SSA form widely used by modern compilers...
- ...and by some formal verification tools[1]
- Candidates C++ front-end providing SSA form:
  - GCC $\geq$ 4.0
  - LLVM

[1] Automatic translation of C/C++ parallel code into synchronous formalism using an SSA intermediate form, AVOCS 2010

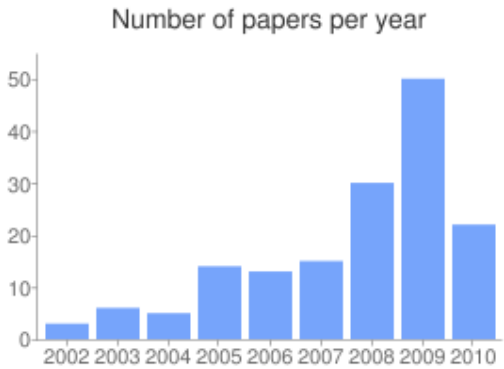
LLVM: Low Level Virtual Machine



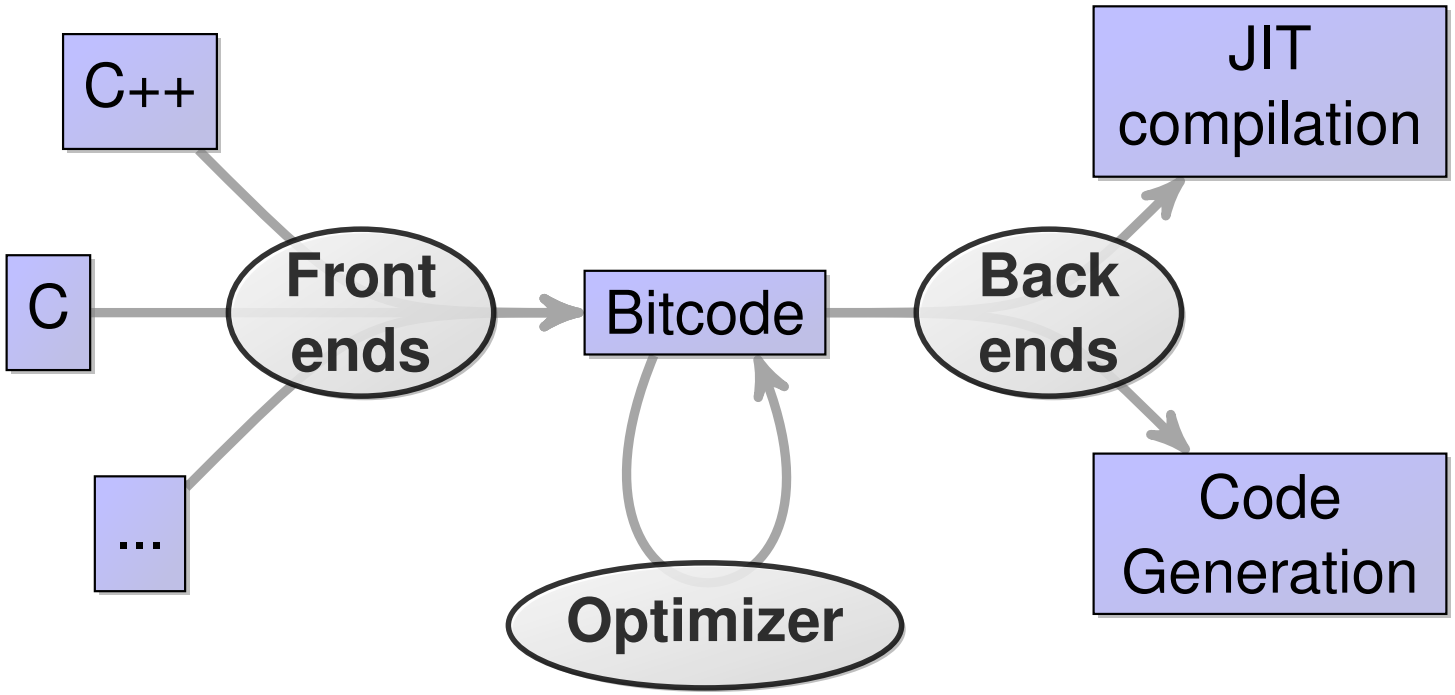
LLVM: Low Level Virtual Machine



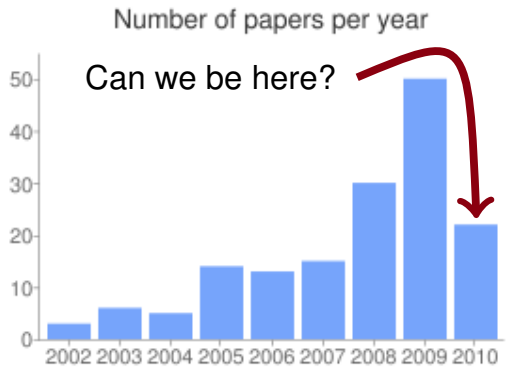
- Clean API
- Clean SSA intermediate representation
- Many tools available



LLVM: Low Level Virtual Machine

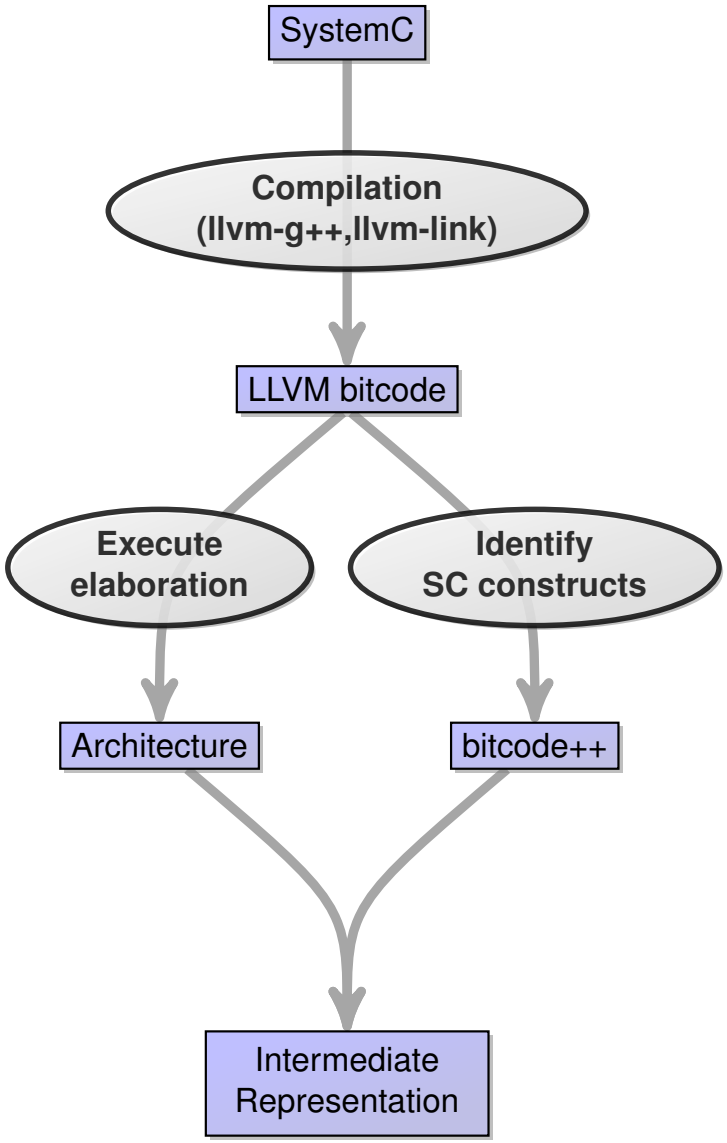


- Clean API
- Clean SSA intermediate representation
- Many tools available

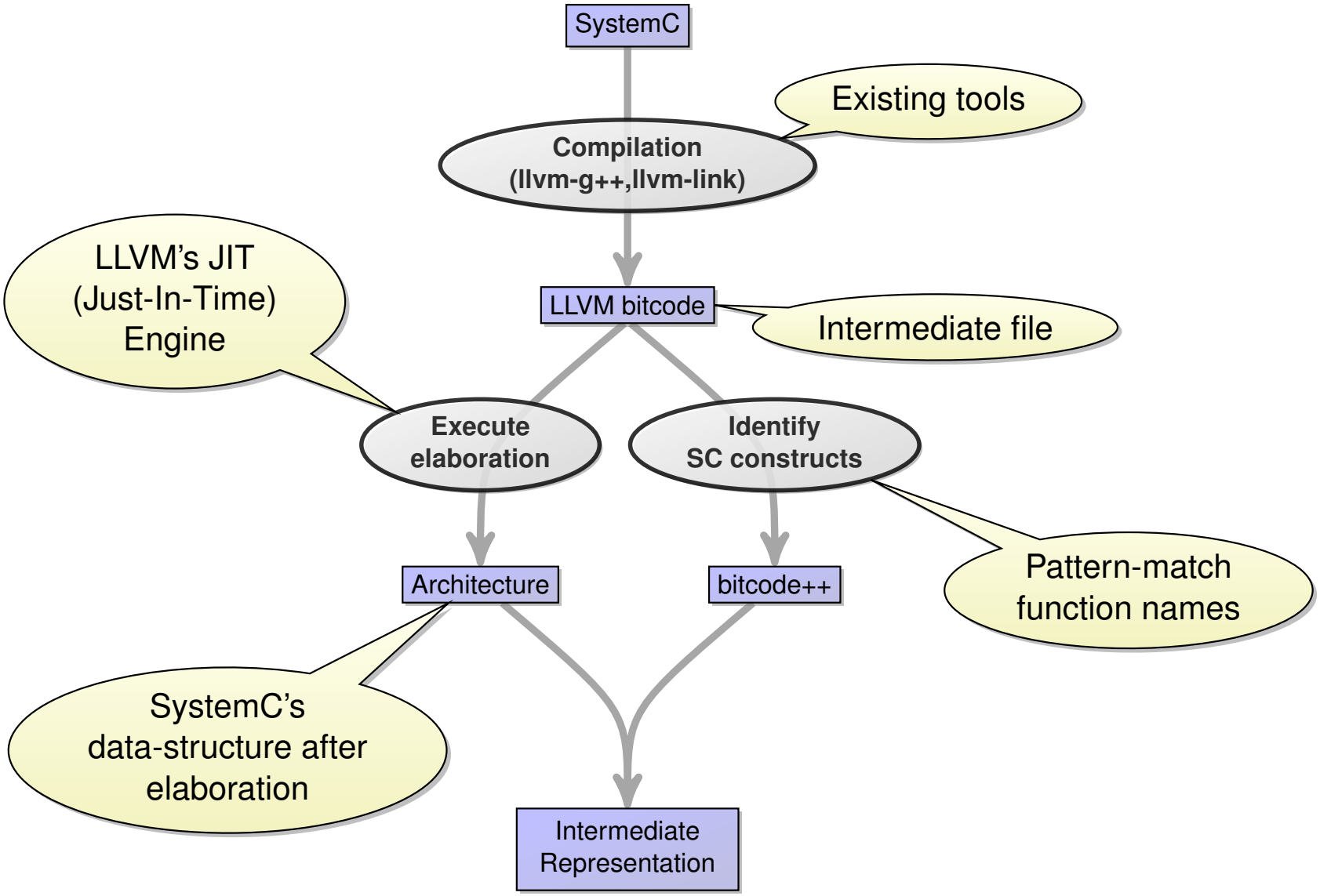


Summary

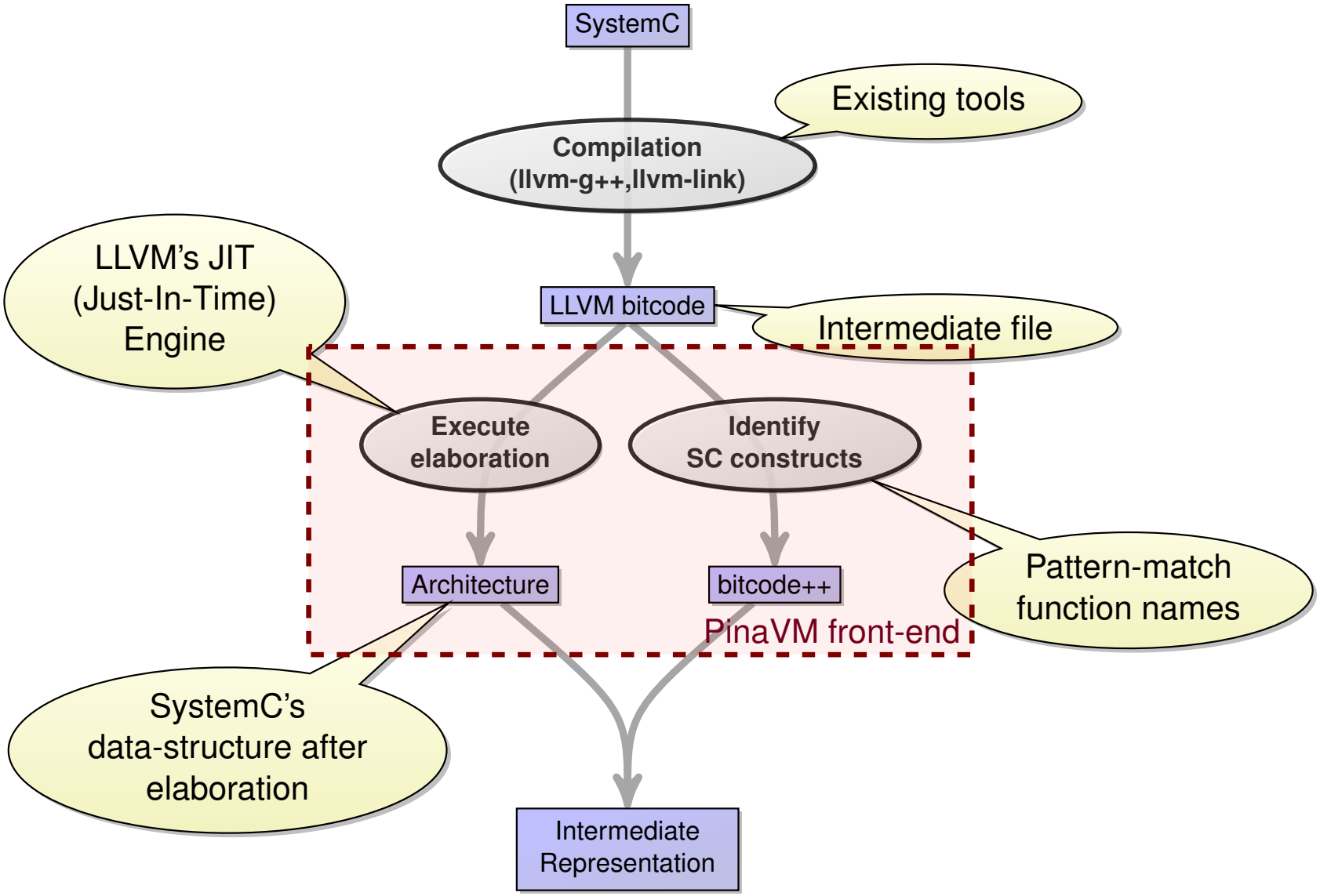
PinaVM : Architecture



PinaVM : Architecture



PinaVM : Architecture



SystemC constructs in LLVM's bitcode

- SystemC (C++) source code:

```
void compute() {
    out.write(true);
}
```

- Bitcode after complation with `llvm-g++`:

```
define linkonce_odr void @_ZN6Source7computeEv(%struct.Source* %this) {
entry:
  %0 = alloca i8
  %"alloca point" = bitcast i32 0 to i32
  store i8 1, i8* %0, align 1
  %1 = getelementptr inbounds %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0
  call void @_ZN7sc_core8sc_inoutIbE5writeERKb
                                  (%"struct.sc_core::sc_inout<bool>"* %2, i8* %0)
  br label %return

return:
  ret void
}
```

SystemC constructs in LLVM's bitcode

- SystemC (C++) source code:

```
void compute() {
    out.write(true);
}
```

- Bitcode after complation with `llvm-g++`:

```
define linkonce_odr void @_ZN6Source7computeEv(%struct.Source* %this) {
entry:
  %0 = alloca i8
  %"alloca point" = bitcast i32 0 to i32
  store i8 1, i8* %0, align 1
  %1 = getelementptr inbounds %struct.Source* %this, i32 0, i32 1
  %2 = getelementptr inbounds %"struct.sc_core::sc_out<bool>"* %1, i32 0, i32 0
  call void @_ZN7sc_core8sc_inoutIbE5writeERKb
                                  (%"struct.sc_core::sc_inout<bool>"* %2, i8* %0)
  br label %return

return:
  ret void
}
```

SystemC constructs in LLVM's bitcode

- Simplified Bitcode:

```
define void Source::compute(%this) {
  ; "out.write(true)" compiled into:
    ; piece of code computing %data = true = 1

    ; piece of code computing %port as
    ; a function of %this

    call sc_core::sc_inout::write(%port, %data)
    ret void
}
```

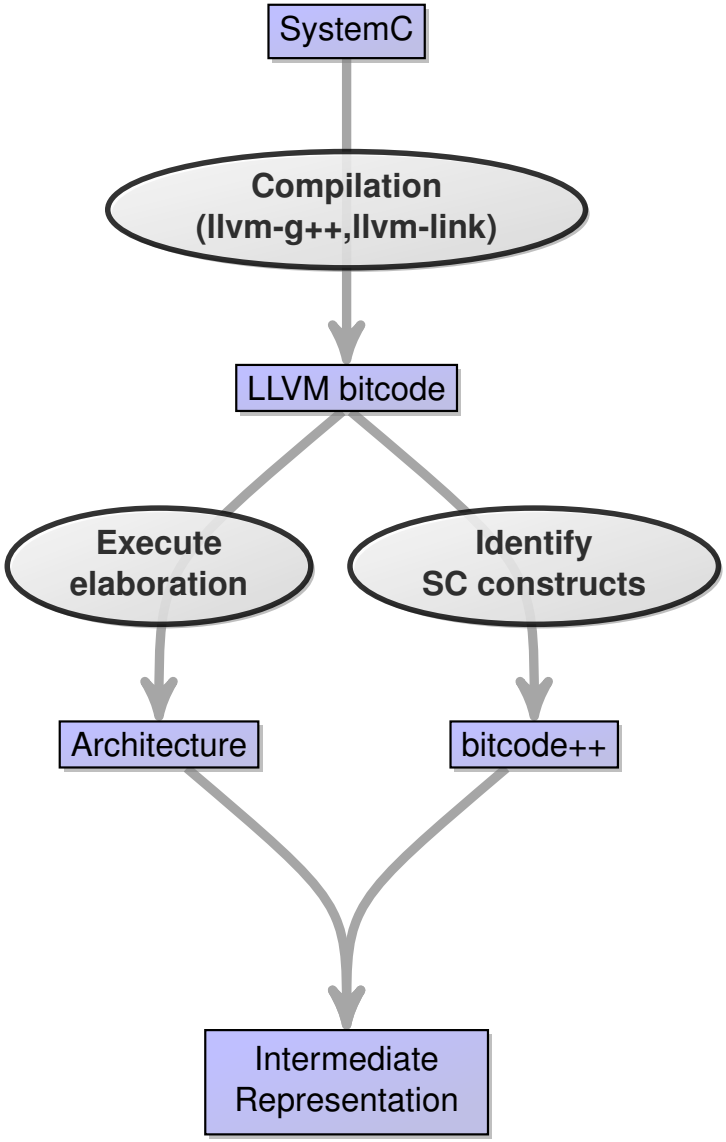
SystemC constructs in LLVM's bitcode

- Simplified Bitcode:

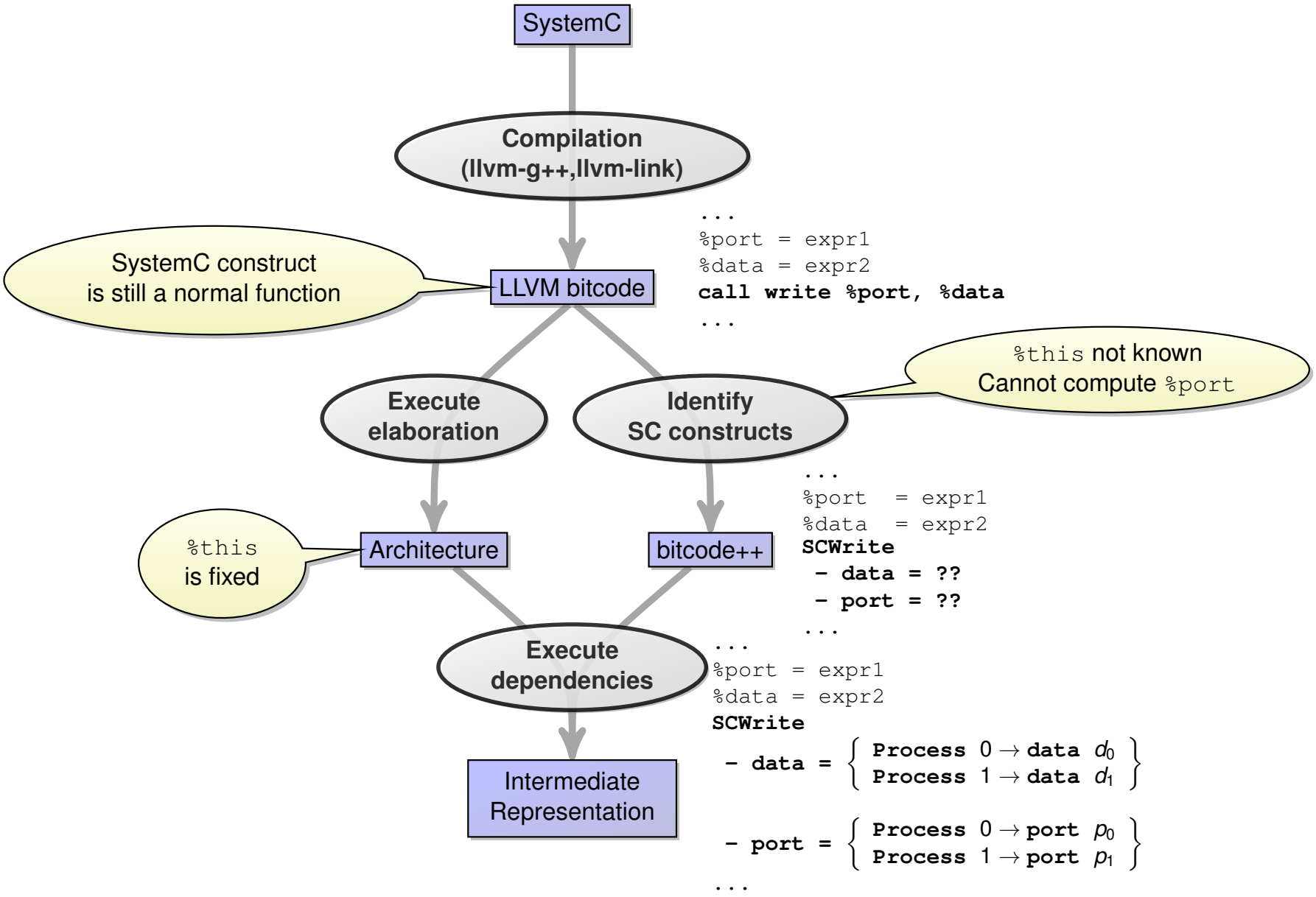
```
define void Source::compute(%this) {
  ; "out.write(true)" compiled into:
    ; piece of code computing %data = true = 1

    ; piece of code computing %port as
    ; a function of %this

    call sc_core::sc_inout::write(%port, %data)
    ret void
}
```

- Computation %port and %data (argument of SystemC primitive)
  - Unknown statically (depend on `this`)
  - Computable for each module once we know `this`!
- What PinaVM does:
  - Extract (slice) pieces of code computing %data and %port
  - JIT-compile and execute them after fixing `%this`

PinaVM: Enriching the bitcode?



PinaVM: Enriching the bitcode?



Summary

Summary

- PinaVM relies on executability (JIT Compiler) for:
  - Execution of elaboration phase ($\approx$ like Pinapa)
  - Execution of sliced pieces of code
- Using a virtual machine to write a SystemC front-end is a *really* good idea!
- Could have benefited from some higher-level constructs in bytecode (builtin object and method calls?)

PinaVM

- Open Source : `http://gitorious.org/pinavm/pages/Home`
- Still a prototype, but very few fundamental limitations
- $\approx$ 3000 lines of C++ code on top of LLVM
- Experimental back-ends for model-checking (SPIN)

Thank you

Questions?

`http://gitorious.org/pinavm/pages/Home`