Building a GenSim Model

Harry Wagstaff, Tom Spink, Bruno Bodin, Bjoern Franke

Institute for Computing Systems Architecture
University of Edinburgh

October 2018

Introduction

In this talk:

- Slightly more detailed look at existing models
- Overview of components of a GenSim description
- Tools and techniques for producing a model

Existing Models

Four models currently actively supported:

- ARMv7
- ARMv8
- RISC-V
- x86-64

Existing Models

Four models currently actively supported:

- ARMv7
- ARMv8
- RISC-V
- x86-64

But several other models have been worked on as side/student projects:

- PowerPC (MSc project)
- MIPS (Side project)
- TI C6x DSP (Undergraduate internship)
- GPU (PhD project)

ARMv7 Model

- Full ARM Core Instruction Set
- Thumb and Thumb-2 Support
- Some NEON and VFP Support
- User Mode and Full System (via Archsim)

ARMv8 Model

- Full AArch64 Instruction Set
- Some FP and Vector Support
- Full-System Support (via Captive)

RISC-V Model

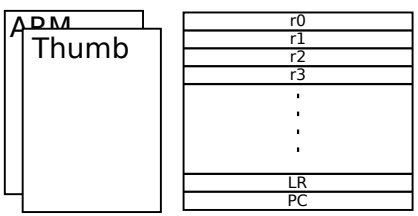
- Full Core Instruction Set
- Some FP Support
- User Mode Only

x86-64 Model

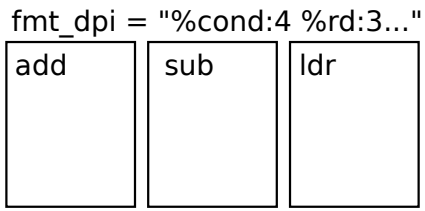
- User mode only
- Uses Intel XED for decoding
- Under development, runs SPEC

Model Components

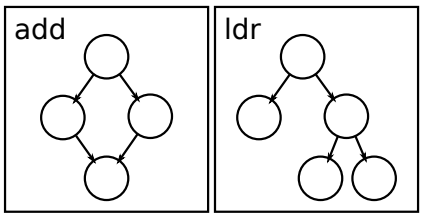
System



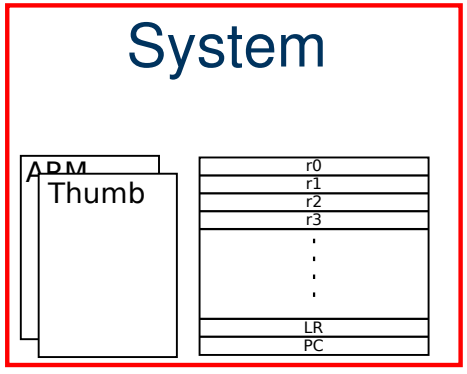
Syntax

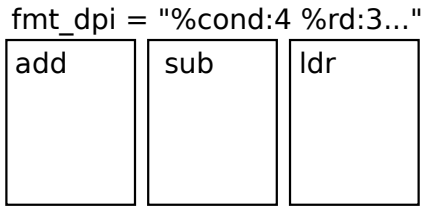


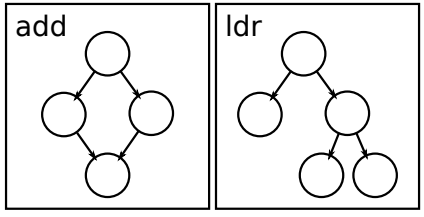
Semantics



System Description







System Description

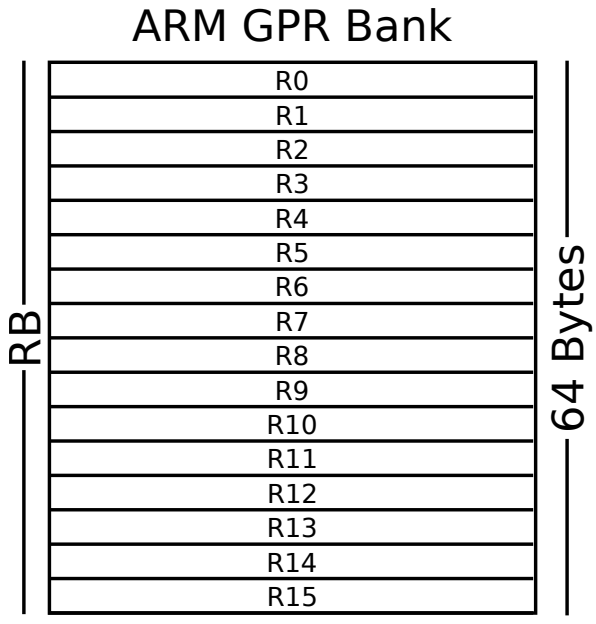
Three main components to system description:

- Register Files
- Features
- Instruction Sets

Register File

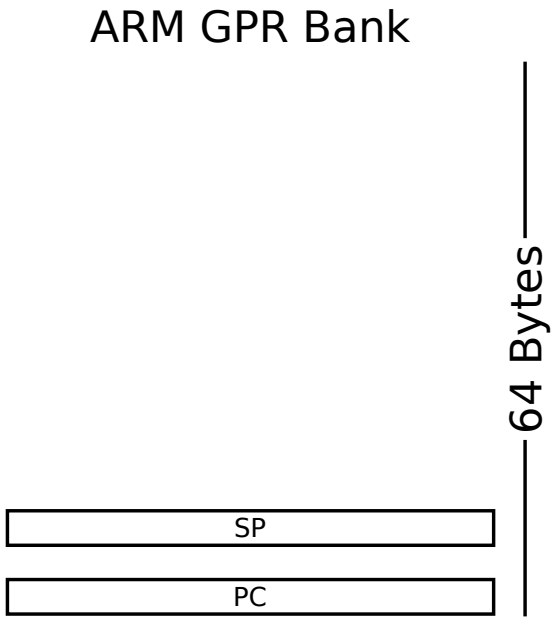
GenSim treats the register file as a set of flat memory regions (spaces) with aliased ‘views’.

Register File



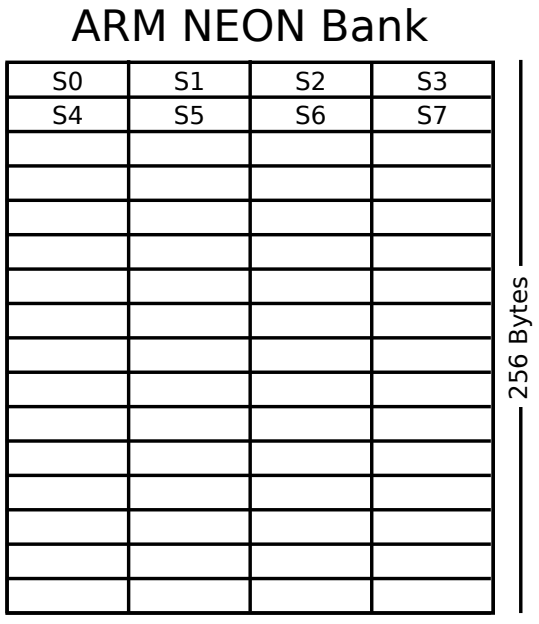

```
ac_regspace(64) {
        bank RB (uint32, 0, 16, 4, 1, 4, 4);
        slot PC (uint32, 4, 60) PC;
        slot SP (uint32, 4, 52) SP;
}
```

Register File



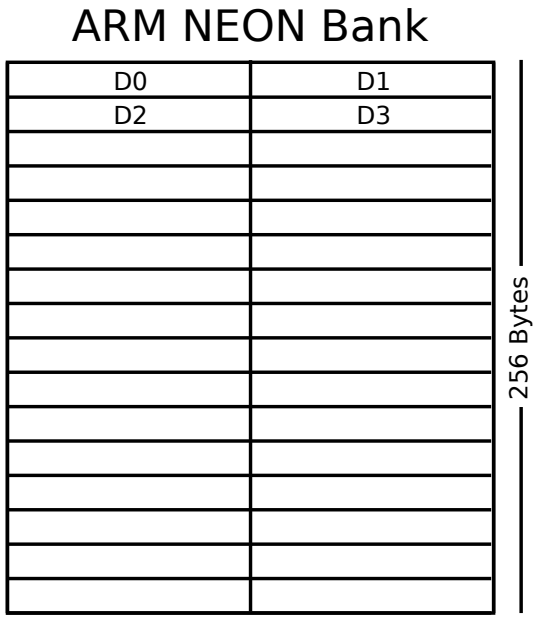

```
ac_regspace(64) {
        bank RB (uint32, 0, 16, 4, 1, 4, 4);
        slot PC (uint32, 4, 60) PC;
        slot SP (uint32, 4, 52) SP;
}
```

Register File - Vectors



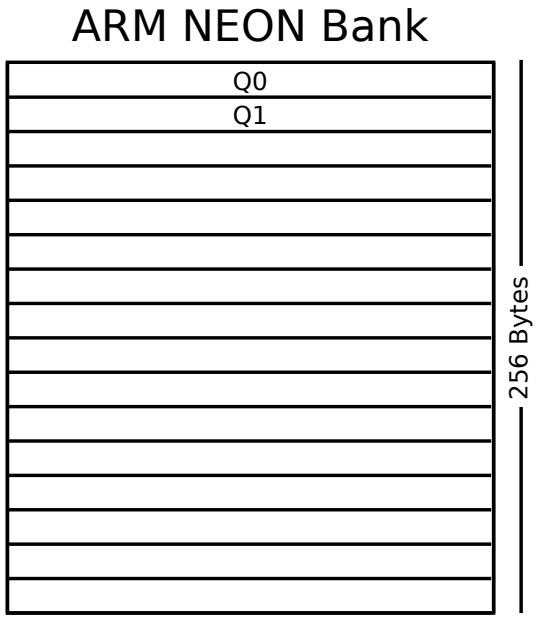

```
ac_regspace(256) {
        bank FP_SP (float,  0, 32, 4, 1, 4, 4);
        bank FP_DP (double, 0, 32, 8, 1, 8, 8);
        bank FP_Q (float, 0, 16, 16, 4, 4, 4);
}
```

Register File - Vectors




```
ac_regspace(256) {
        bank FP_SP (float,  0, 32, 4, 1, 4, 4);
        bank FP_DP (double, 0, 32, 8, 1, 8, 8);
        bank FP_Q (float, 0, 16, 16, 4, 4, 4);
}
```

Register File - Vectors




```
ac_regspace(256) {
        bank FP_SP (float,  0, 32, 4, 1, 4, 4);
        bank FP_DP (double, 0, 32, 8, 1, 8, 8);
        bank FP_Q (float, 0, 16, 16, 4, 4, 4);
}
```

Features

Primarily performance feature - give hints to simulator about things which change infrequently.

- Actual processor features
- FPU Enablement
- Privilege mode
- Configuration info

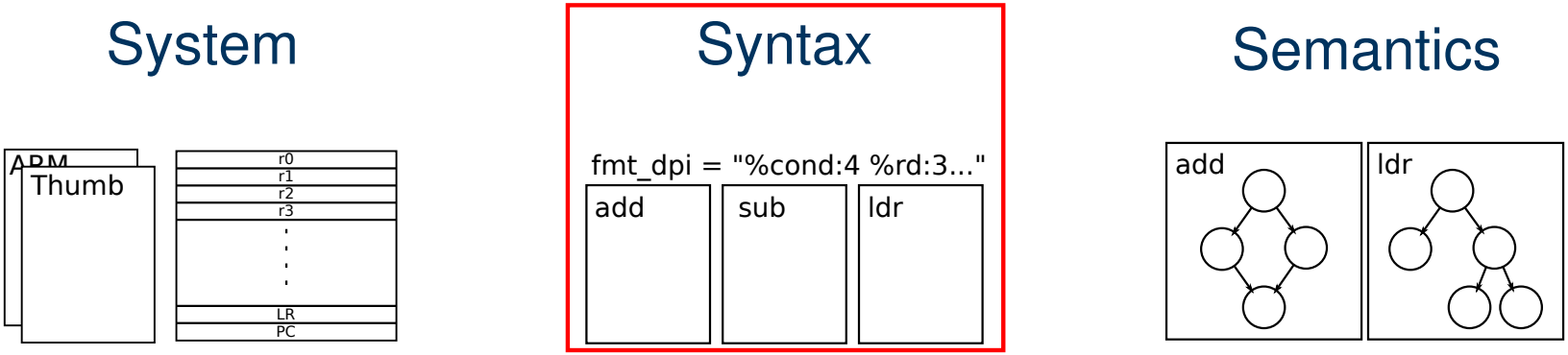
Instruction Sets

Which instruction sets are available to this architecture?

- Allows for instruction set switching
- e.g. ARM $\Leftrightarrow$ Thumb[1]

[1] SAMOS'15: https://doi.org/10.1109/SAMOS.2015.7363665

Syntax Description



Overview

How are instructions encoded?
How do we decode them?

GenSim Assumptions

GenSim currently makes a few assumptions about instruction encoding:

- Instruction decoding is stateless
- Instructions are standalone
- Instructions execute in PC order

GenSim Assumptions

However, GenSim does have native support for the following features:

- Instruction Predication
- Variable Length Instructions

Formats

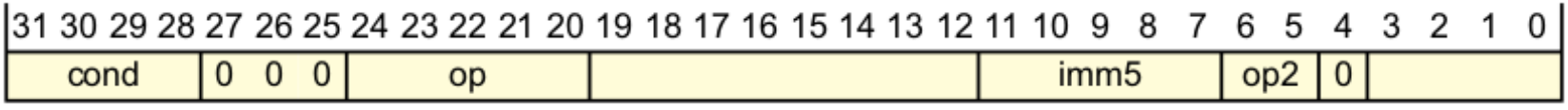
Formats described using printf-like string

- Inspired by ArchC[2]
- Specify constant and variable fields

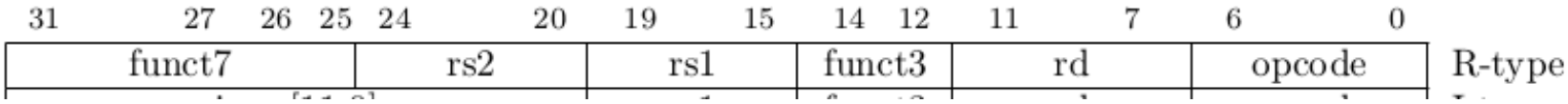
[2] Developed by UNICAMP, `www.archc.org`

Formats

ARM Data-Processing (Register) Format

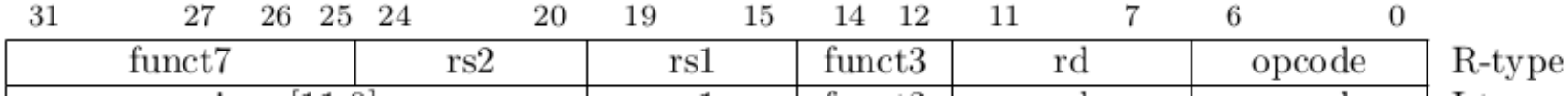


RISC-V R-Type



Formats

RISC-V R-Type




```
"%funct7:7 %rs2:5 %rs1:5 %funct3:3 %rd:5 %opcode:7";
```

Formats

```
AC_ISA(riscv)
{

ac_format rtype = "%funct7:7 %rs2:5 %rs1:5 %funct3:3 %rd:5 %opcode:7";

}
```

Instructions

Instructions represent ‘individually decodable’ units

- Somewhat flexible...

Instructions

Instructions represent ‘individually decodable’ units

- Somewhat flexible...
- But decode as much as possible at decode time!

Instructions

Instructions represent ‘individually decodable’ units

- Somewhat flexible...
- But decode as much as possible at decode time!
- i.e., avoid general ‘ALU’ instructions

Instructions

```
AC_ISA(riscv)
{

ac_format rtype = "%funct7:7 %rs2:5 %rs1:5 %funct3:3 %rd:5 %opcode:7";

ac_instr<rtype> add;

}
```

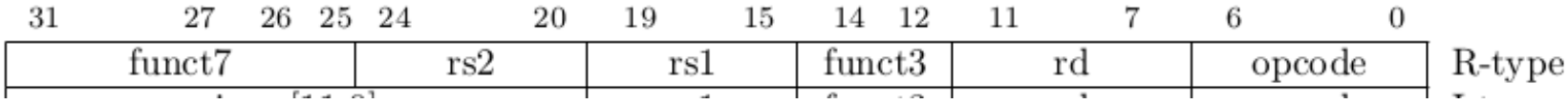
Decoding

Let's consider an example from RISC-V:

Decoding

Let's consider an example from RISC-V:

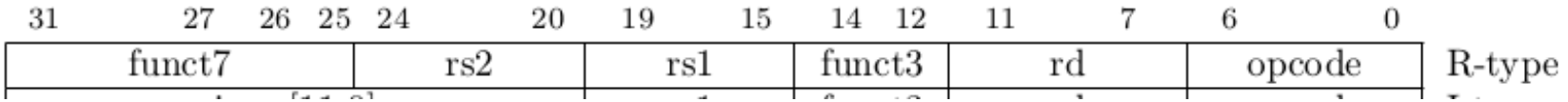
RISC-V R-Type



Decoding

Let's consider an example from RISC-V:

RISC-V R-Type

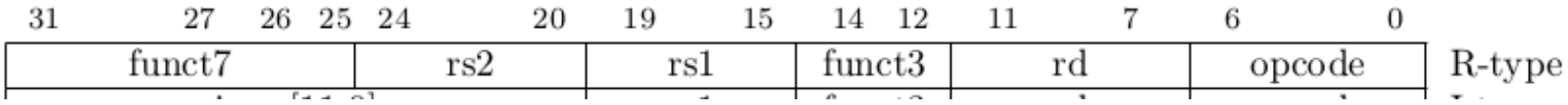


RISC-V ADD

Decoding

Let's consider an example from RISC-V:

RISC-V R-Type



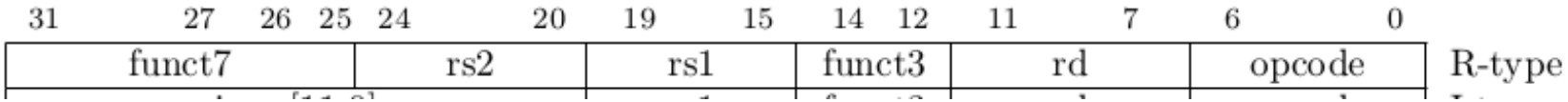
RISC-V ADD


```
funct7   0b0000000
funct3   0b000
opcode   0b0110011
```

Decoding

Let's consider an example from RISC-V:

RISC-V R-Type



RISC-V ADD


```
funct7   0b0000000
funct3   0b000
opcode   0b0110011
```

```
add.set_decoder(funct7=0x0, funct3=0x0, opcode=0x33);
```

Decoding

```
AC_ISA(riscv)
{

ac_format rtype = "%funct7:7 %rs2:5 %rs1:5 %funct3:3 %rd:5 %opcode:7";

ac_instr<rtype> add;

isa_ctor(riscv)
{

add.set_decoder(funct7=0x0, funct=0x0, opcode=0x33);

}

}
```

Branch Metadata

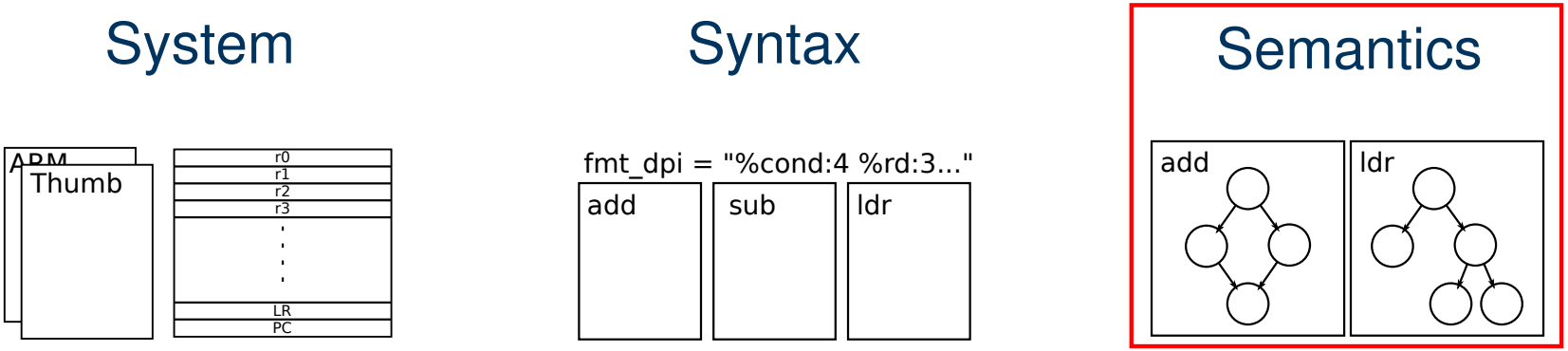
Used to indicate which instructions can change the PC

- Is an instruction an end-of-block?
- Is it a direct jump? What is it's target?
- Is it an indirect jump?

Can also be used for various optimisations.[3]

[3] LCTES'14, http://doi.acm.org/10.1145/2666357.2597810

Semantics Description



GenC

C-like language used to implement instruction behaviours

- C-like syntax
- Only basic types supported
- Vector types and operations supported natively
- Reference parameters are supported

Architectural Manipulation

Processor state is manipulated using built-in functions

- Register accesses
- Memory access
- Privilege/ISA modes
- Floating Point Environment

Register Access

Built-in functions available:

```
slot_type read_register(slot_name)
reg_type read_register_bank(bank_name, index)

void write_register(slot_name, slot_type)
void write_register_bank(bank_name, index, reg_type)
```

slot_type depends on the type specified in the System Description

Memory Access

```
void mem_read_[8,16,32,64](address_type, uint[8,16,32,64]&)
void mem_write_[8,16,32,64](address_type, uint[8,16,32,64])

void mem_read_[8,16,32,64]_user(address_type, uint[8,16,32,64]&)
void mem_write_[8,16,32,64]_user(address_type, uint[8,16,32,64])
```

‘user’ variants are used in full-system simulation, where certain kernel-mode memory instructions (ldrt/strt in ARM) access memory with user privileges

Vector Types

GenC has support for vector types and operations on values of those types

- Vectors are declared using array notation
- Vectors can be read and written to register banks
- Arithmetic operations can be performed on vectors

Vector Types

GenC has support for vector types and operations on values of those types

- **Vectors are declared using array notation**
- Vectors can be read and written to register banks
- Arithmetic operations can be performed on vectors

```
float[2] fp_vector;
uint32[4] u32_vector;
```

Vector Types

GenC has support for vector types and operations on values of those types

- Vectors are declared using array notation
- **Vectors can be read and written to register banks**
- Arithmetic operations can be performed on vectors

```
fp_vector = read_register_bank(FP_SP2, inst.vm);
write_register_bank(U32_V, inst.vm, u32_vector);
```

Vector Types

GenC has support for vector types and operations on values of those types

- Vectors are declared using array notation
- Vectors can be read and written to register banks
- **Arithmetic operations can be performed on vectors**

```
// pairwise addition
result_vector = fp_vector + fp_vector;

// operation by scalar
result_vector = fp_vector + 10.0;
```

Intrinsics

There are a large number of intrinsics available for either manipulating the architectural state or performing other operations. For example:

- `take_exception` - trigger an exception
- `__builtin_clz` - count leading zeros
- `bitcast_float_u32` - 'bitcast' a float to a uint32

Developing a Model

So how do we build a model from the ground up?

Developing a Model

Model development is not too difficult but can take a long time

- ARMv7 and RISC-V models developed by students with no prior experience
- Mostly a task of transcribing instruction pseudocode
- Can be easy to introduce subtle errors or edge cases

Our Workflow

We generally start small and work our way up:

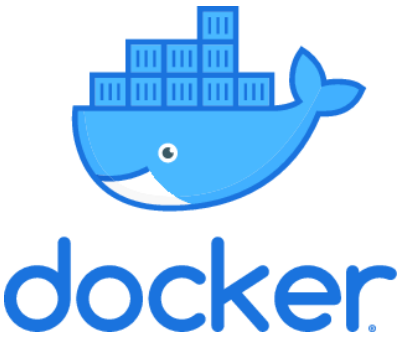
- Implement enough instructions for a Hello World program
  - (this can be quite a lot due to startup/shutdown code)
- Then try small benchmarks (e.g., EEMBC, MyBench)
- Work up to larger benchmarks (e.g., SPEC)

The model should be tested at each step (using fuzzing/small unit tests)

Continuous Build/Test

We use Jenkins for continuous build.

- Commits to GenSim repo are built and tested
- All supported repos tested via Docker



Jenkins: http://jenkins.io, Docker: http://www.docker.com/

Instruction Fuzzing

Fuzzing is a randomized testing method

Instruction Fuzzing

Fuzzing is a randomized testing method

- Test instructions with many inputs against a ground truth

Instruction Fuzzing

Fuzzing is a randomized testing method

- Test instructions with many inputs against a ground truth
- We use QEMU as a ground truth

Instruction Fuzzing

Fuzzing is a randomized testing method

- Test instructions with many inputs against a ground truth
- We use QEMU as a ground truth
- **Still problems with unspecified/undefined behaviour**

Instruction Fuzzing

Fuzzing is a randomized testing method

- Test instructions with many inputs against a ground truth
- We use QEMU as a ground truth
- **Still problems with unspecified/undefined behaviour**

We have our own tools for instruction fuzzing (which are linked from GenSim website)

Debugging with Tracing

Once we're confident that our model works, we can run some larger programs

Debugging with Tracing

Once we're confident that our model works, we can run some larger programs

But they might still go wrong! How do we debug this?

Debugging with Tracing

We can carefully inspect a trace of the simulation and try and see what went wrong.

- This can be quite a tedious and error-prone process
- It is usually necessary to narrow down the issue beforehand
- Key idea is to spot errors in execution manually
  - Need to be very familiar with the architecture!

Model Optimisation

Certain code structures can lead to poor performance:

- Very complex instructions
- Non-fixed loop bounds
- ‘Hand written’ predicates
- Enablement checks (can be solved using Features)

Model Optimisation

```
if(read_register(FP_ENABLED)) {
        // do something
}

if(__builtin_get_feature(FP_ENABLED)) {
        // do something
}
```

The x86-64 Model

- The newest supported GenSim model
- Runs Linux User Mode applications
- Some SSE and x87 Support

X86 Instruction Decoding

- x86 has a stateful instruction encoding
- This can't be easily handled by GenSim
- We handle this by using the Intel XED library
- Instruction semantics still use GenSim

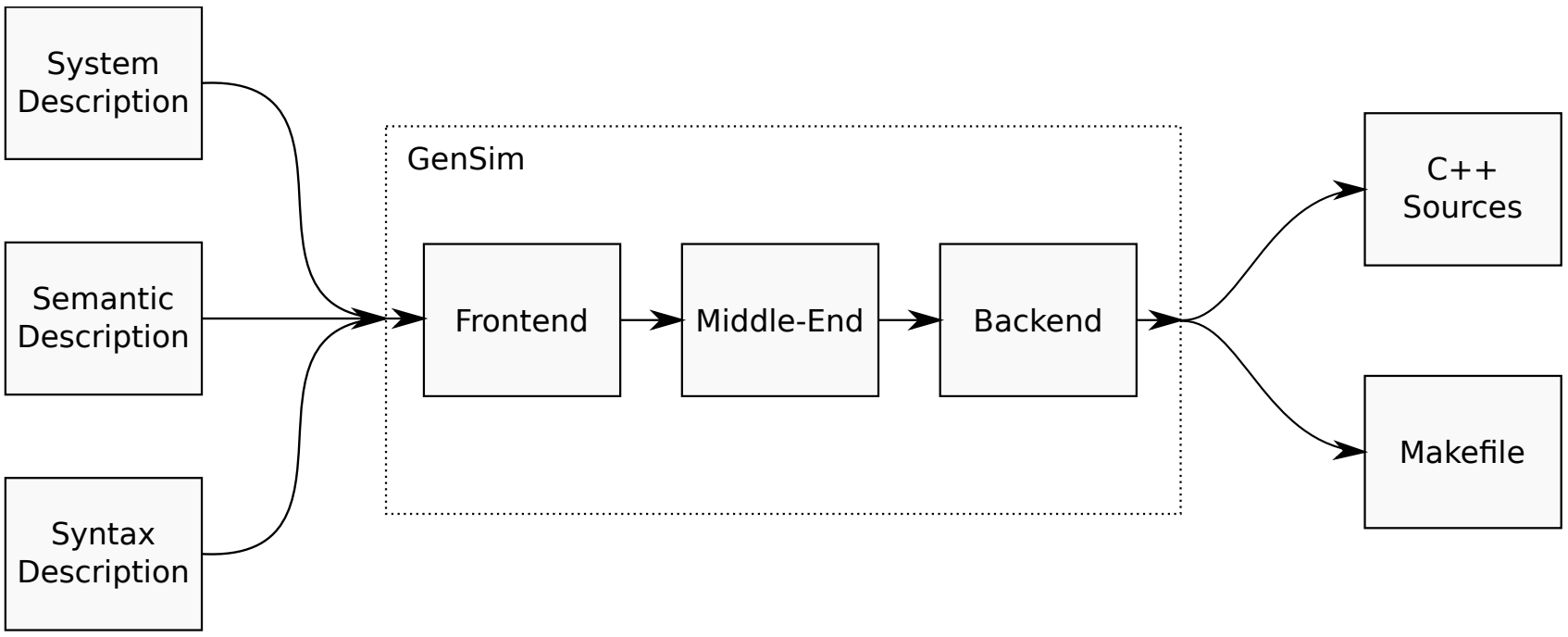
Experiences with x86

- Biggest issue is inconsistency
- Lots of 'defined undefinedness' makes testing tricky
- Vector performance is critical
- Otherwise handled fairly well in GenSim

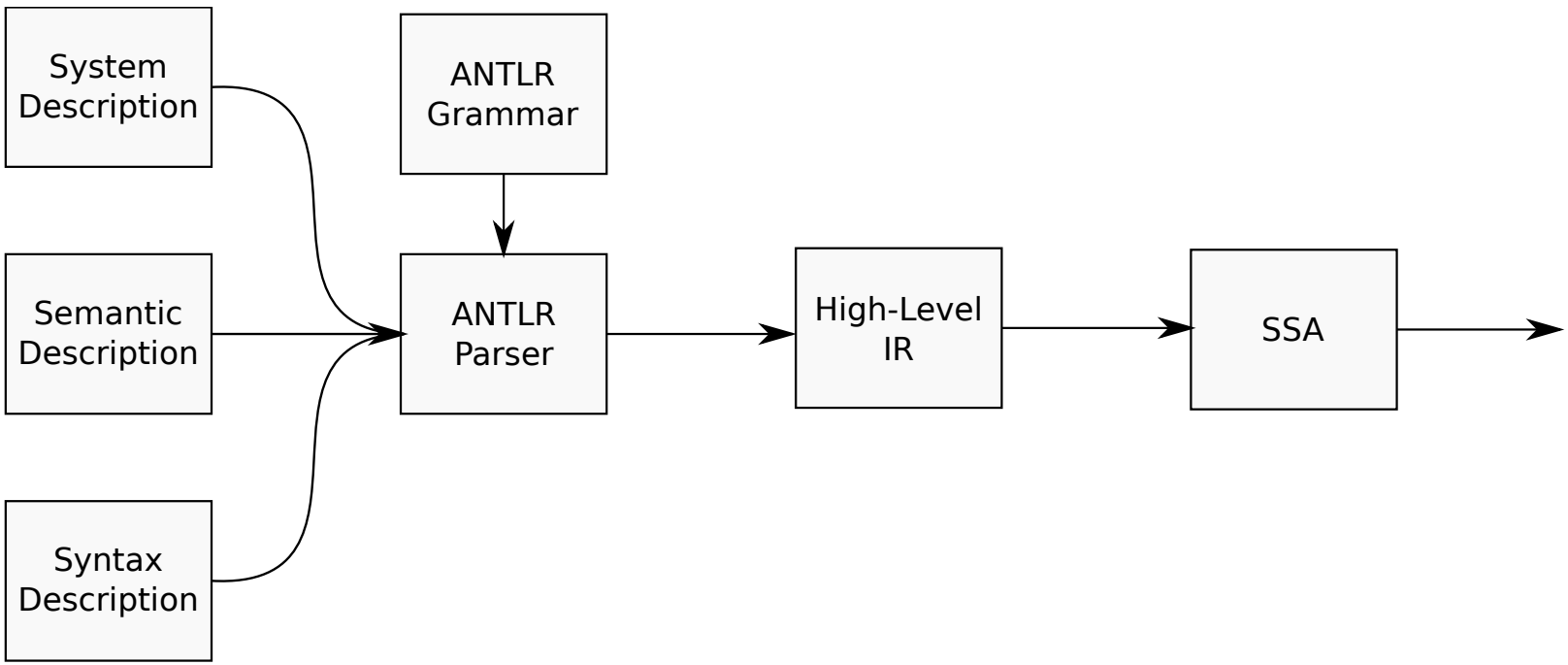
GenSim Internals

General Flow

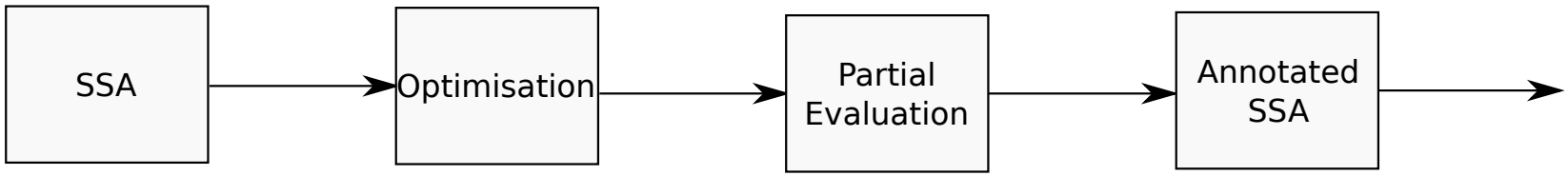
General Flow



Frontend

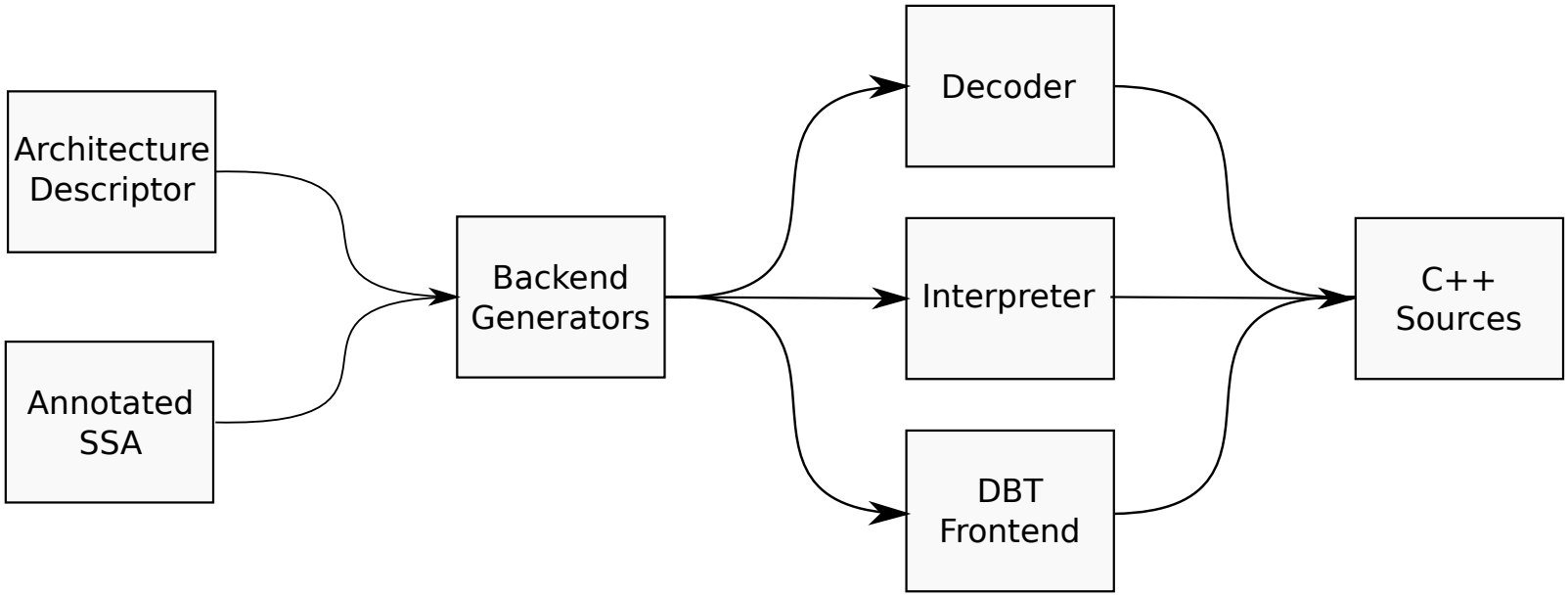

System Description
ANTLR Grammar
Semantic Description
ANTLR Parser
High-Level IR
SSA
Syntax Description

Middle-End



DAC'13: https://doi.org/10.1145/2463209.2488760

Back-Ends


Architecture Descriptor
Annotated SSA
Backend Generators
Decoder
Interpreter
DBT Frontend
C++ Sources

Recap

To conclude:

- We've gone over the available models
- Looked at the components of a model
- I've shared some experiences building models
- Briefly covered the internal flow of GenSim

Conclusion

Thanks for coming!

Any Questions?

harry.wagstaff@gmail.com
general@gensim.org