Design and Implementation of a Binary Translator from AArch64 to a Custom Intermediate Representation

Konstantin Garbers
Advisor: Martin Fink
Chair of Computer Systems
https://dse.in.tum.de/

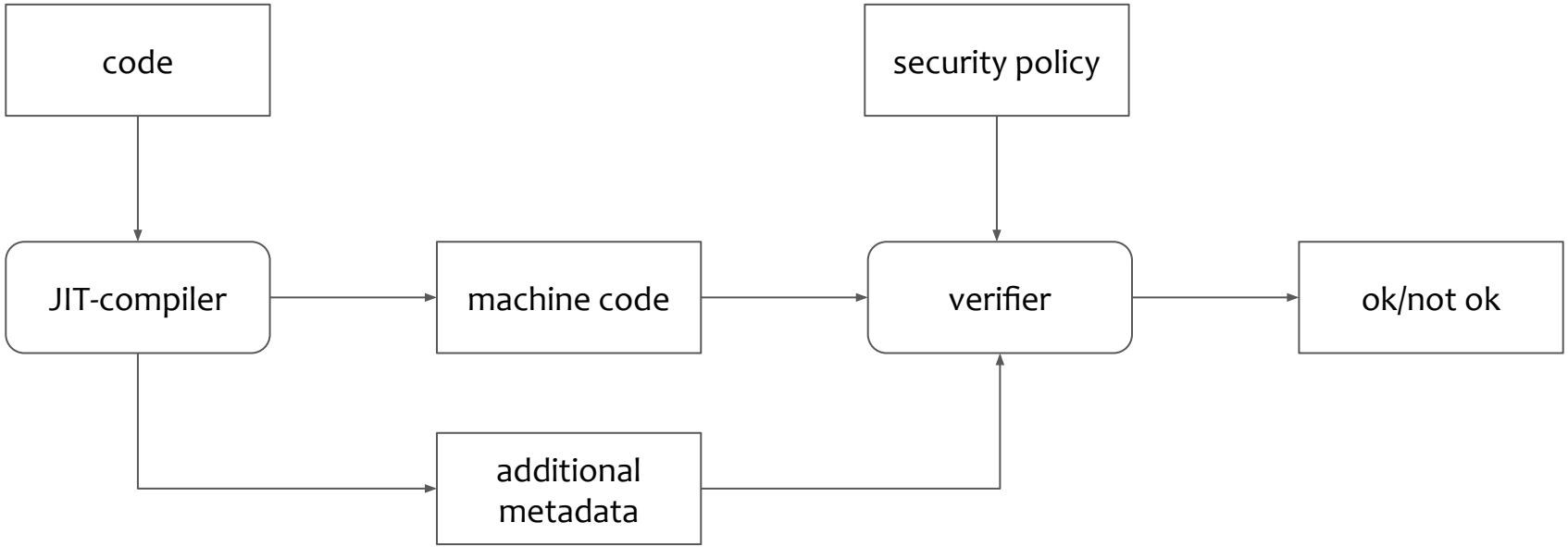
29.10.2024 – 28.02.2025

Introduction: JIT-Compilers

- Code execution tools used in the JVM or in browsers
- Interpretes code and compiles certain parts of code to machine code
- Compiled machine code is source of security issues

Introduction: TrustNoJit (TNJ)

Library that uses the **proof-carrying-code** framework to verify JIT-compiled code



Motivation

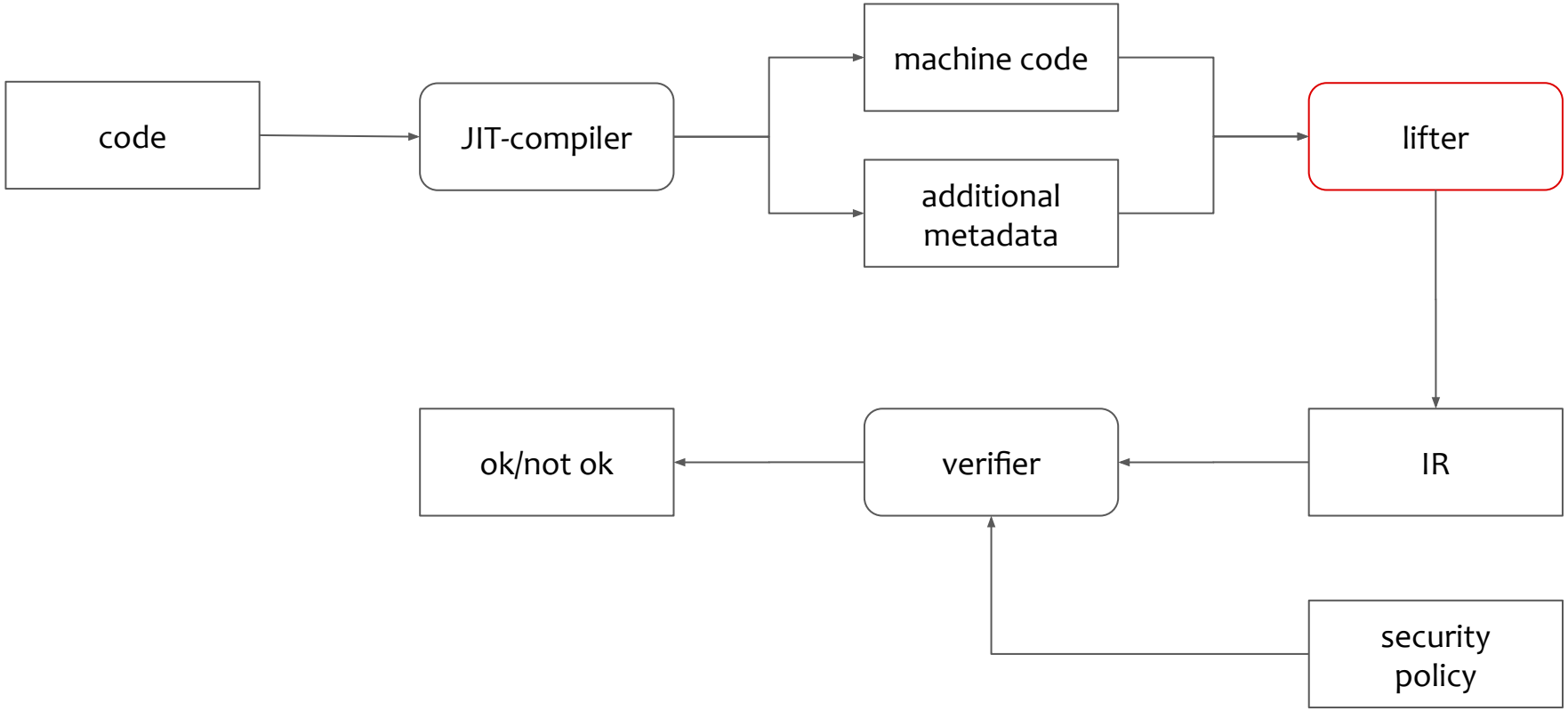
Problem: Machine code is hard to verify!

- **Difficult control flow** tracking
- Verification requires intricate knowledge of **side effects** of assembly instructions
- Verifier needs to be rewritten for every language

Solution:

- Transform assembly code into an Intermediate Representation (IR)

Solution: Integration of a lifter into TNJ



Outline

- ~~Motivation~~
- Background
- Design
- Implementation
- Evaluation

Background: AArch64

64-bit execution state of of ARMv8-A

- Every instruction is 4 bytes

Challenges:

- 5 addressing modes
- 32 general-purpose registers
- 4 condition flags
- More than 300 base instructions and multiple extensions

Background: Assembly IR (AIR)

```
0x00: ldr x0, [x1, #0x80]
```

BasicBlock (BB)

```
block_0:
v0 = i64.read_reg "x1"
v1 = i64.add v0, 0x80
v2 = i64.load v1
i64.write_reg v2, "x0"
```

AArch64 Code

AIR

Outline

- ~~Motivation~~
- ~~Background~~
- Design
- Implementation
- Evaluation

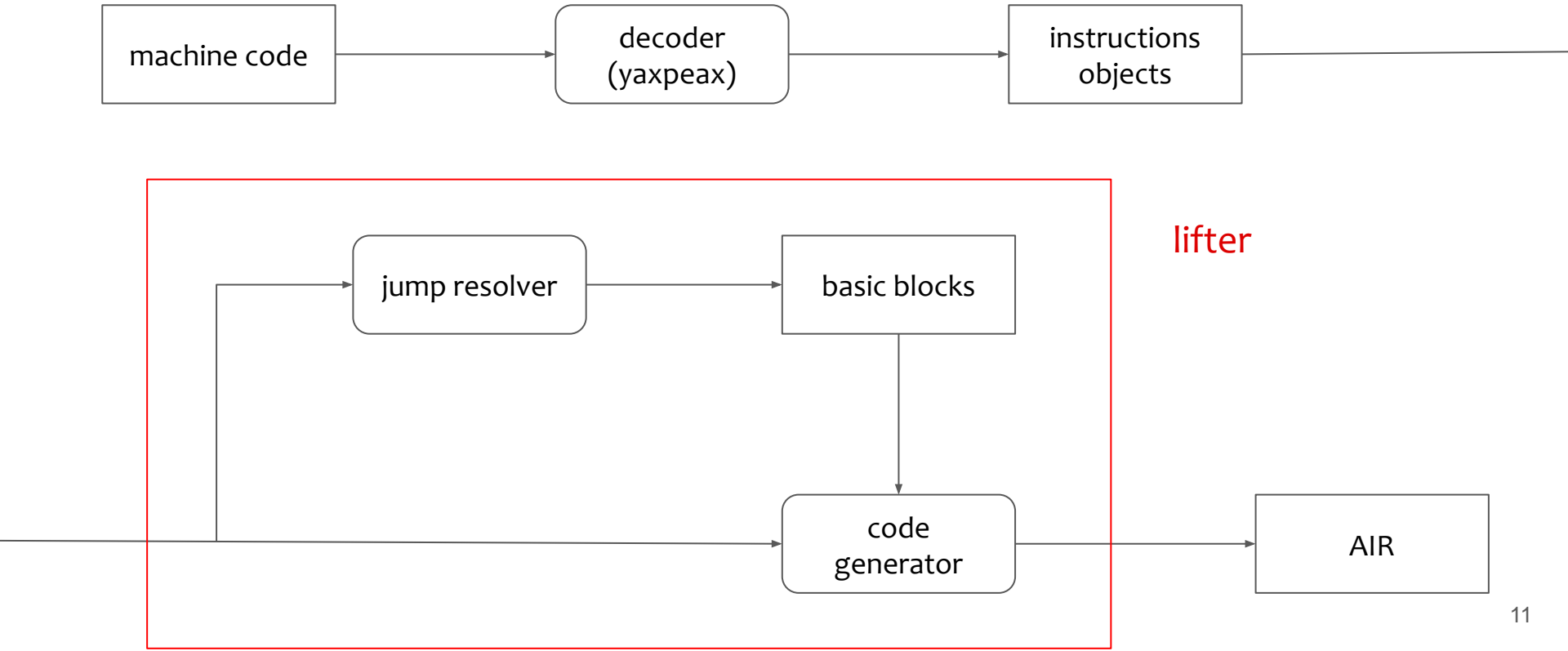
Problem: Lifter Can Not Translate Backward Jumps

AArch64 Code

```
0x00: add x1, x2, x3
0x04: b 0x08
0x08: b 0x04
```

- AIR consists of BasicBlocks (BB)
- Jumps **jump to the beginning** of BBs
- Jumps may only be **at the end** of a BB

Lifter Workflow



Solution: Create BBs in A Separate Translation Pass

AArch64 Code

```
0x00: add x1, x2, x3
0x04: b 0x08
0x08: b 0x04
```

AIR

```
block_0:
 v0 = i64.read_reg "x2"
 v1 = i64.read_reg "x3"
 v2 = i64.add v1, v0
 i64.write_reg v2, "x1"
 jump block_4
```

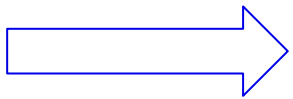
```
block_4:
 jump block_8
```

```
block_8:
 jump block_4
```

Unsupported Instructions

Mark destination registers of unsupported instructions as opaque

```
umov x0, v1.s[0]
```



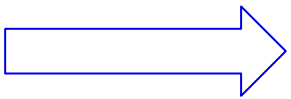
```
v0 = i64.opaque
i64.write_reg v1, "x0"
```

Represent unknown value using opaque value

Untrackable Instructions

System calls, hypervisor calls or function calls modify registers untraceably

```
0x00: blr x0
```



```
v0 = i64.read_reg "x0"
dynamic_jump v0
invalidate_regs
```

Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- Implementation
- Evaluation

Implementation

- Developed in **rustc 1.84.0-nightly**
- Uses the **yaxpeax**[1] library to decode machine code
- Supports **129 out of 301** base variants

[1]yaxpeax: https://github.com/iximeow/yaxpeax-x86

Outline

- ~~Motivation~~
- ~~Background~~
- ~~Design~~
- ~~Implementation~~
- Evaluation

Evaluation: Setup

Hardware:

- AMD Ryzen 5 5500U CPU
- 16 GiB DRAM
- Ubuntu 22.04.05 LTS

Test Binaries:

- WASM-Applications from the Sightglass[2]-benchmarking suite
- Compilation to AArch64 binaries using standard compilation settings

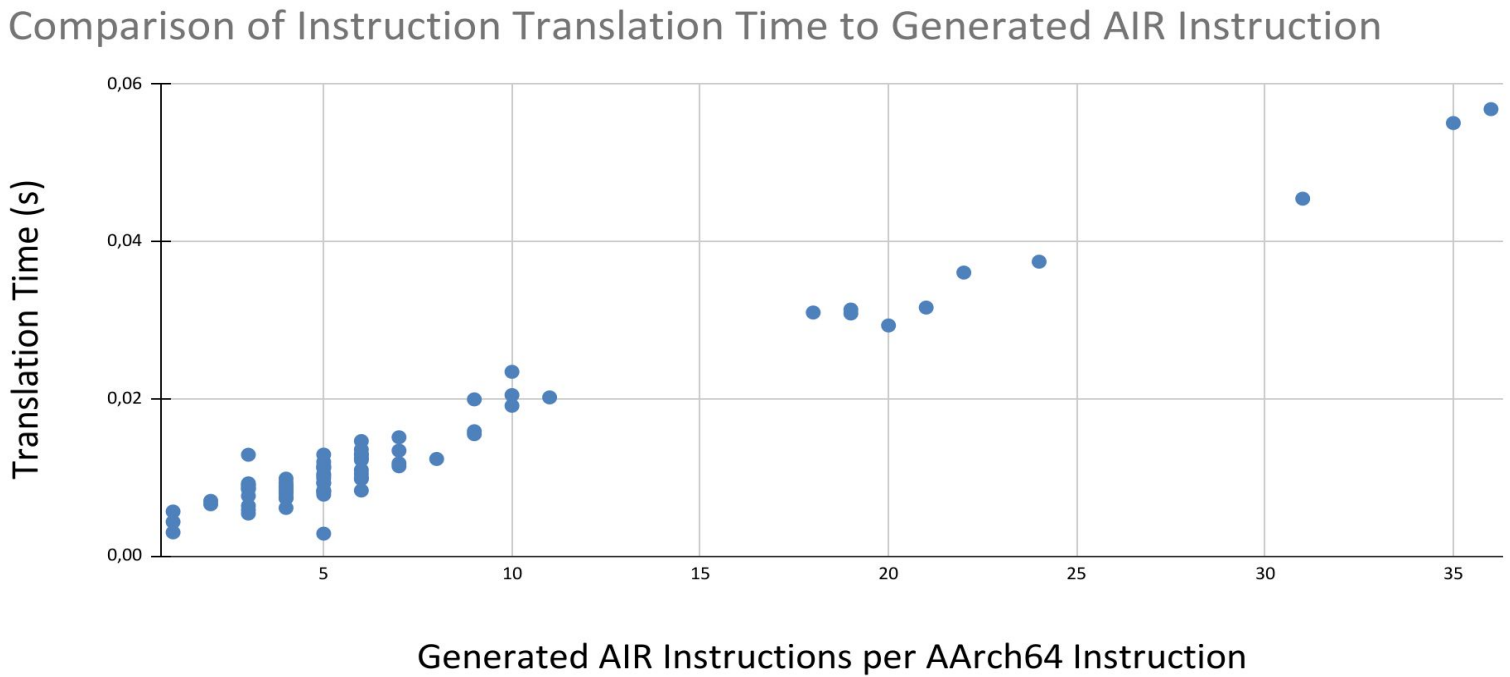
[2]Sightglass: https://github.com/bytecodealliance/sightglass

Evaluation: Research Questions

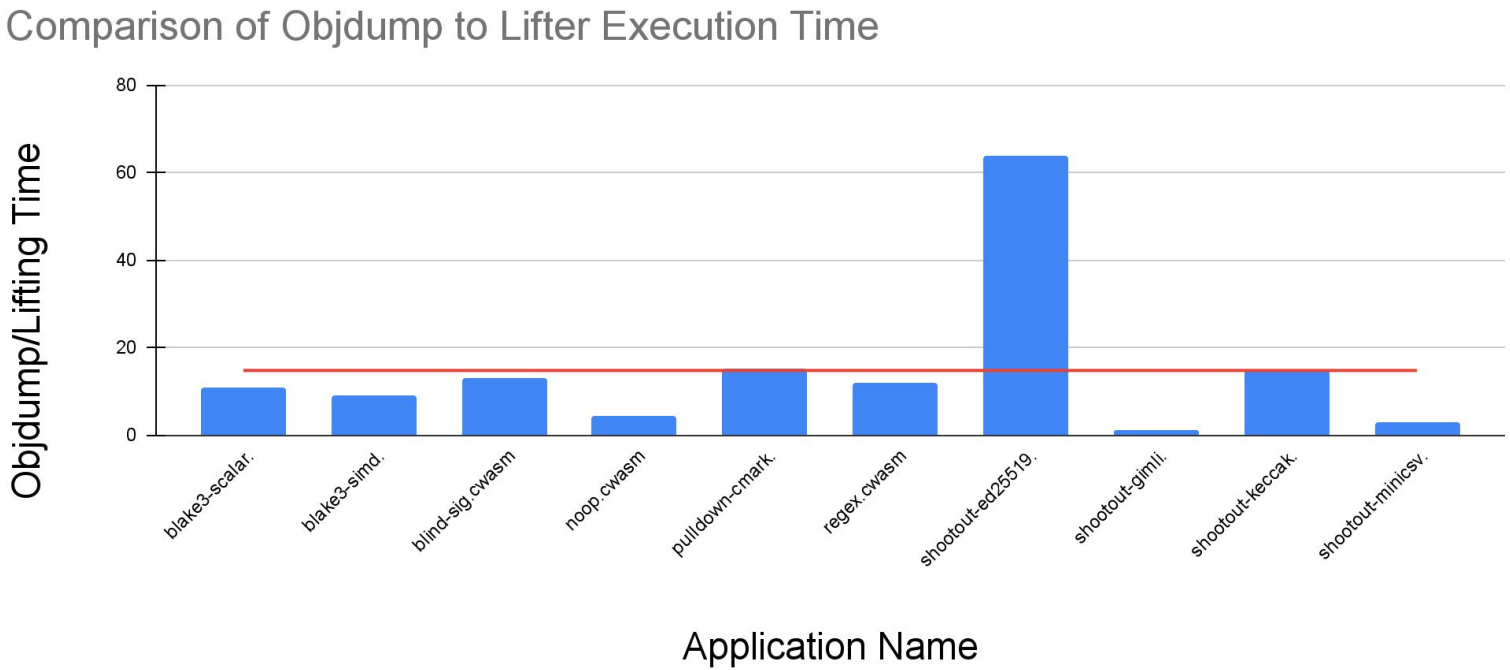
1. Is lifter **fast** enough for real-word-settings?

2. Is the lifter **usable** in real-world settings?

Insight: Translation Speed Correlates To Generated AIR-Instructions



Insight: Lifting Time Varies a Lot Across Different Applications



Lower Values -> Better

Evaluation: Research Questions

1. Is lifter **fast** enough for real-word-settings?
   - A single instruction takes **20µs-80µs** to lift
   - Average function will take **2ms-11ms** to lift

2. Is the lifter **usable** in real-world settings?
   - Machine decoding dependency **yaxpeax is a current limitation**
   - Supports on average more than **99.7%** of all tested instructions
   - Supports more than **99%** of instructions in every tested application

Related Works: State-of-the-art

Verifier Frameworks that lift AArch64 into an IR:

- angr[1]: Python framework that lifts AArch64 into a VEX IR
- Miasm[2]: Python framework that lifts AArch64 into a custom IR
- BAP[3]: OCaml framework that lifts AArch64 into a custom IR

[1] angr: https://github.com/angr/angr
[2] Miasm: https://github.com/cea-sec/miasm
[3] BAP: https://github.com/BinaryAnalysisPlatform/bap

Related Works: A Custom Lifter is Advantageous

Integration with TNJ

- **Smooth data flow** between the verifier and lifter
- **Direct access** to TNJ's built-in functionalities

Execution Speed

- **Single-purpose** application
- **Lower-level programming language** for faster execution time

Tailored Intermediate Representation (IR)

- IR is specifically designed for **TNJ's requirements**

Conclusion

Contributions

- Design of a custom intermediate language AIR
- Implementation of an AArch64 to AIR lifter

Evaluation

- Lifter supports more than **99.7%** of all encountered instructions
- Decoding dependency yaxpeax is a **bottleneck**
- Future aim is support of **function calls, pointer authentication and memory ordering operations**

Backup

Implementing Flag Registers in AIR

Represent flags using **separate** registers

Reading Flags:

```
v0 = i1.read_reg "z"
```

Writing Flags:

```
i1.write_reg v0, "z"
```

Represent flags using a **single** registers

Writing immediates:

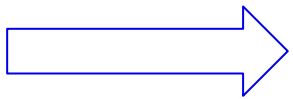
```
i8.write_reg 0xf, "flags"
```

Implementing Register Sizes in AIR

Supported Sizes: **BOOL, I8, I16, I32, I64, I128**

Edge case requires support for I128:

```
smulh x1, x2, x3
```



```
v0 = i64.read_reg "x2"
v1 = i64.read_reg "x3"
v2 = i64.imul v0, v1
v3 = i128.ashr v2, 0x40
i64.write_reg v3, "x1"
```

Implementing Flags in AIR

Represent flags using **separate** registers

Reading Flags:

```
v0 = i1.read_reg "z"
```

Writing Flags:

```
i1.write_reg v0, "z"
```

Generated instructions per application