elfconv: AOT compiler that translates Linux/AArch64 ELF binary to WebAssembly

repo: https://github.com/yomaytk/elfconv

2024/02/04
Masashi Yoshimura, NTT
Twitter : @ming_rrr

Features of WASM

- ✅ portable
  - enables to run apps on **both browsers and servers** without modification
- ✅ secure
  - **highly isolated from the host kernel** on the server by **WASI**.
  - memory isolation with harvard architecture
    - architecture that separates codes and data in the memory.
- ❌ limitation in the capability of apps
  - can jump to only the instructions that are determinable at compile time
    - cannot indirectly jump to the instructions generated in the data memory at runtime
  - WASI implementation doesn't cover all POSIX APIs (e.g. fork, exec)

challenging in building WASM

Many programming languages support WASM (e.g. C, C++, Rust, Go, …).

However, it isn’t easy to build WASM in some cases as follows.

1. The programming language that you want to use doesn’t completely support WASM
   - The support of some languages is insufficient
   - ref: https://github.com/appcypher/awesome-wasm-langs
2. binaries are available, but the source codes of the binaries are not available
   - e.g.) The source code is not available under lisence
3. difficult to build the source code
   - cannot use the dependent libraries

→ **Run binaries on WASM environment**

Existing projects that run Linux binaries on WASM

- **TinyEMU**: https://bellard.org/tinyemu/
  - Author: Fabrice Bellard
  - x86 and RISC-V emulator available on the browser
  - Linux kernel can run on the browser
- **container2wasm**: https://github.com/ktock/container2wasm
  - Author: Kohei Tokunaga, NTT
  - enables to run Linux kernel and container runtimes with emulators compiled to WASM (e.g. TinyEMU)
  - can run containers without modification on the browser and WASI runtimes

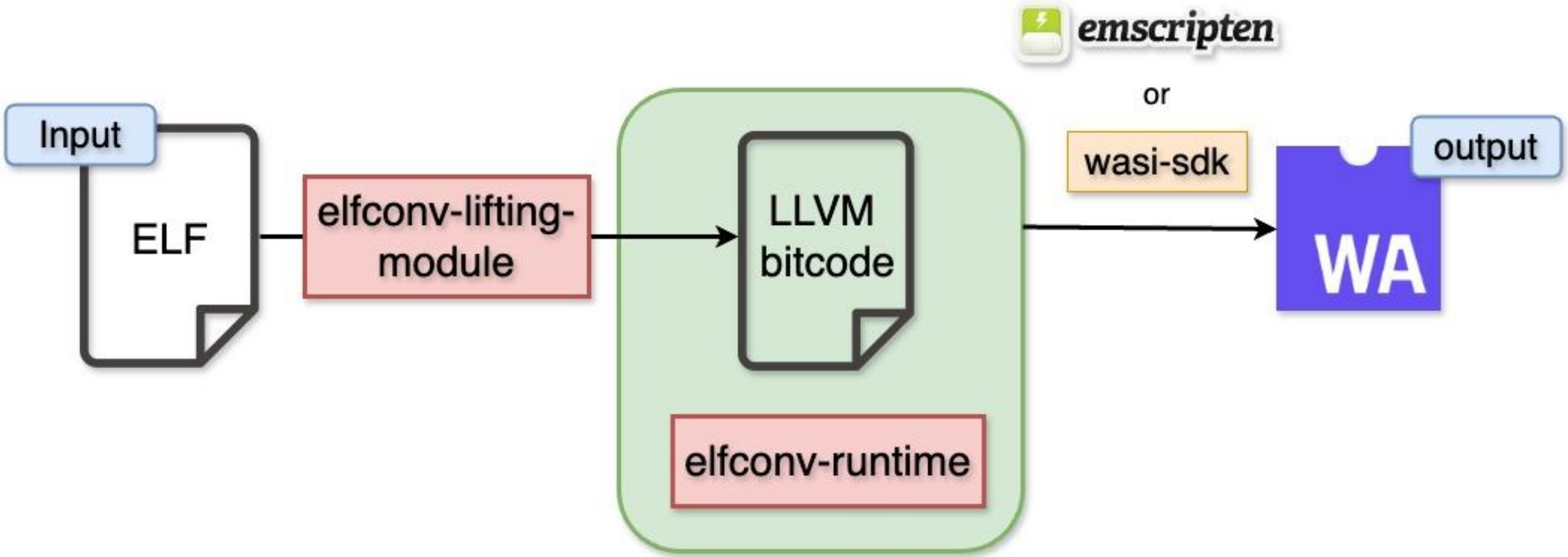
But, emulators possibly incur large performance overheads…

➡ **AOT compile Linux binaries to WASM!**

elfconv: AOT compiler from Linux/ELF to WASM

NTT

- *elfconv-lifting-module* compiles Linux ELF binary to LLVM bitcode
- compile LLVM bitcode and *elfconv-runtime* to WASM
    - elfconv-runtime includes Linux syscalls emulation etc…



Demo

- Demo Program : Neural Network for training MNIST database
  - MNIST database: large database of handwritten digits for training
  - repo : https://github.com/AndrewCarterUK/mnist-neural-network-plain-c
  - keep outputting ***Average Loss*** and ***Accuracy***

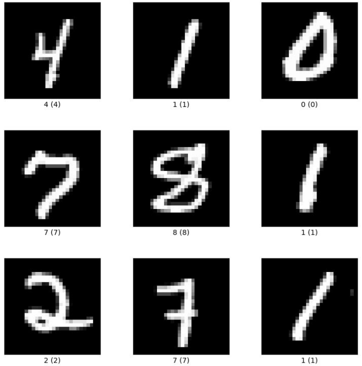
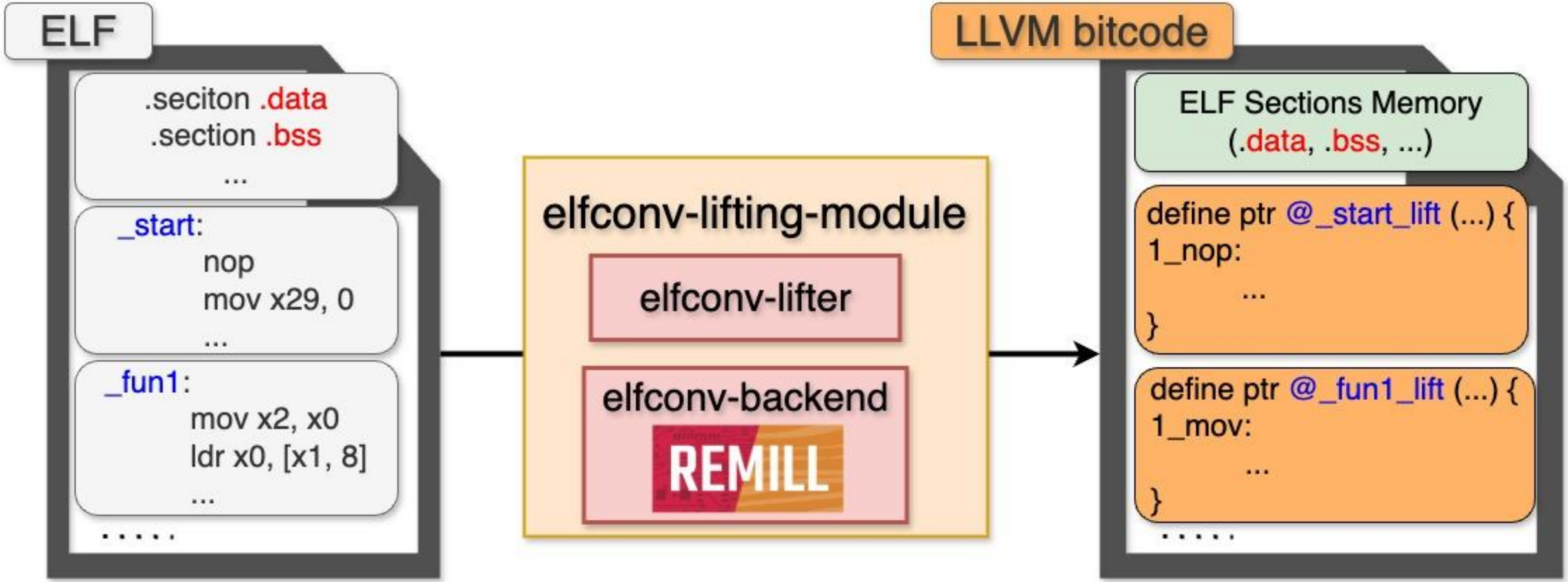


Fig. MNIST database

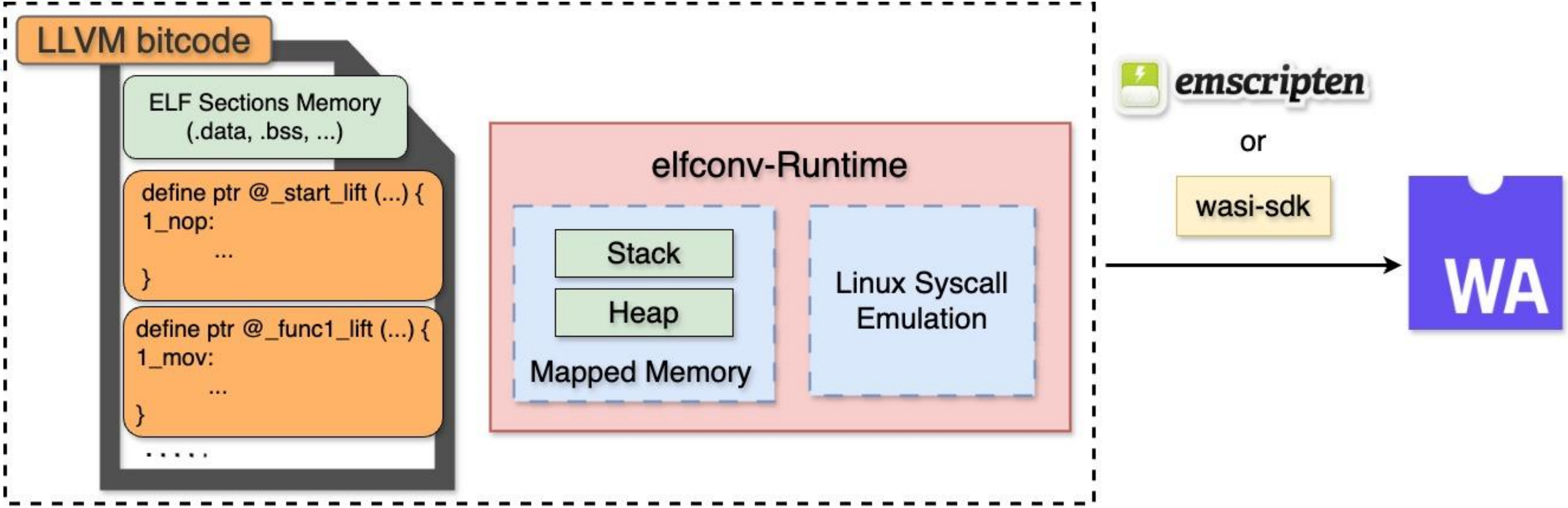
How it works? (ELF -> LLVM bitcode)

- elfconv-lifter
  - parse ELF binary, map every ELF section, etc…
- remill (elfconv-backend) : https://github.com/lifting-bits/remill
  - library for lifting machine code to LLVM IR



How it works? (LLVM bitcode -> WASM)

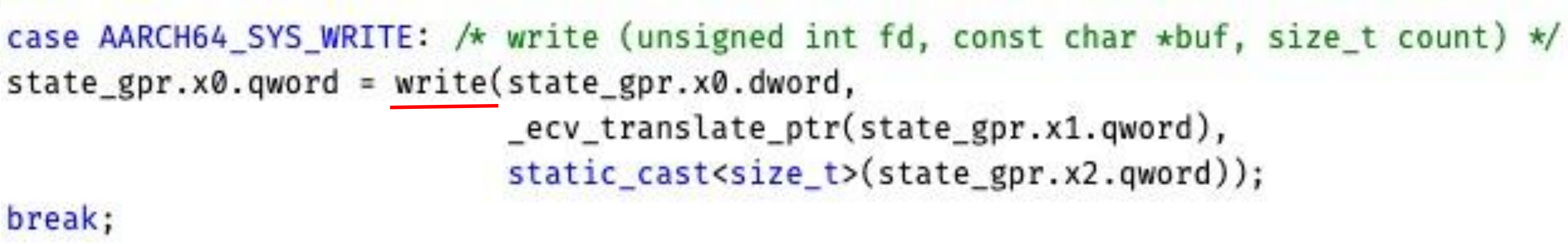
- statically link LLVM bitcode and elfconv-Runtime
- elfconv-Runtime
  - mapped memory (stack, heap), Linux system calls emulation



How it works? (Linux syscalls emulation)

- libc implementation: emscripten, wasi-libc, etc…

Case 1. use libc function if it exists (e.g. write)

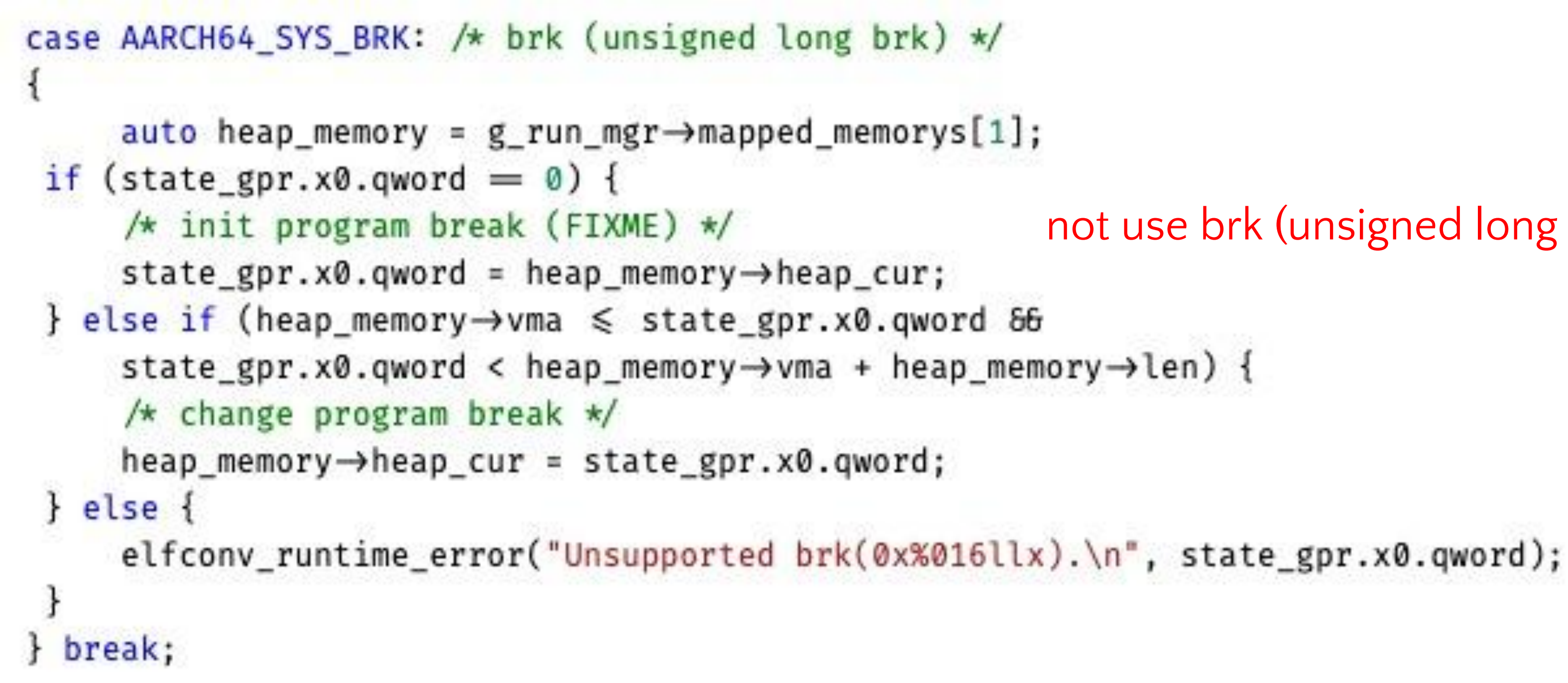


```
case AARCH64_SYS_WRITE: /* write (unsigned int fd, const char *buf, size_t count) */
state_gpr.x0.qword = write(state_gpr.x0.dword,
                           _ecv_translate_ptr(state_gpr.x1.qword),
                           static_cast<size_t>(state_gpr.x2.qword));
break;
```

How it works? (Linux syscalls emulation)

- libc implementation: emscripten, wasi-libc, etc…

Case 2. pseudo-implement the syscall if it doesn't exist (e.g. brk)

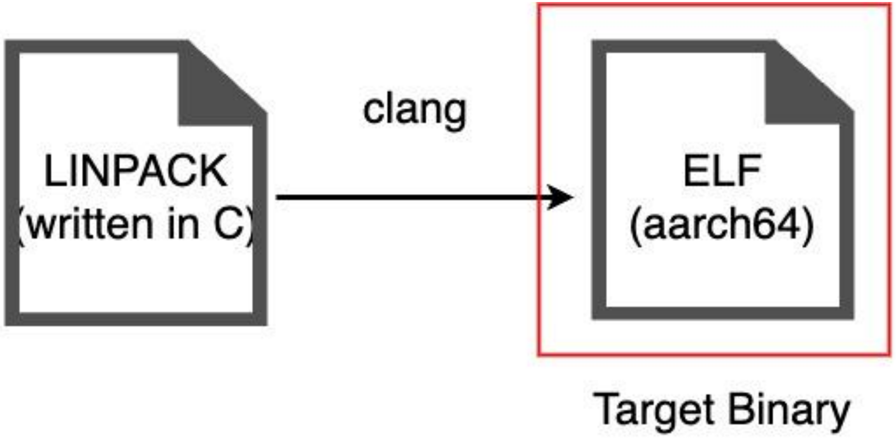


```
case AARCH64_SYS_BRK: /* brk (unsigned long brk) */
{
     auto heap_memory = g_run_mgr->mapped_memorys[1];
 if (state_gpr.x0.qword == 0) {
     /* init program break (FIXME) */
     state_gpr.x0.qword = heap_memory->heap_cur;
 } else if (heap_memory->vma <= state_gpr.x0.qword &&
     state_gpr.x0.qword < heap_memory->vma + heap_memory->len) {
     /* change program break */
     heap_memory->heap_cur = state_gpr.x0.qword;
 } else {
     elfconv_runtime_error("Unsupported brk(0x%016llx).\n", state_gpr.x0.qword);
 }
} break;
```

not use brk (unsigned long brk)

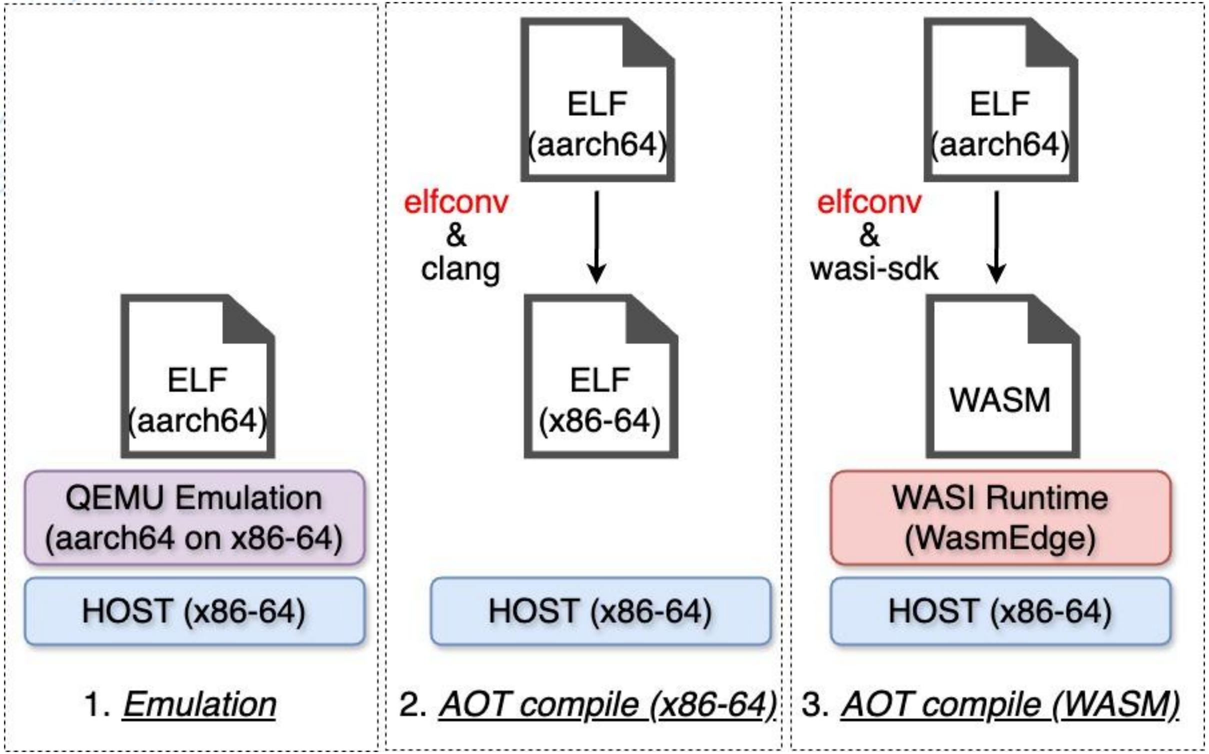
Performance

- Benchmark : LINPACK Benchmark (https://netlib.org/benchmark/hpl/)
  - program to evaluate 64-bit floating-point operations per second (FLOPS).
  - source code : https://www.netlib.org/benchmark/linpackc.new



Performance Measure Method

- compare three methods
  - 1. *Emulation*   2. *AOT compile (x86-64)*   3. *AOT compile (WASM)*



Performance

NTT

1. *Emulation*

   195.115 (MFLOPS)

2. *AOT compile (x86-64)*

   200.177 (MFLOPS)

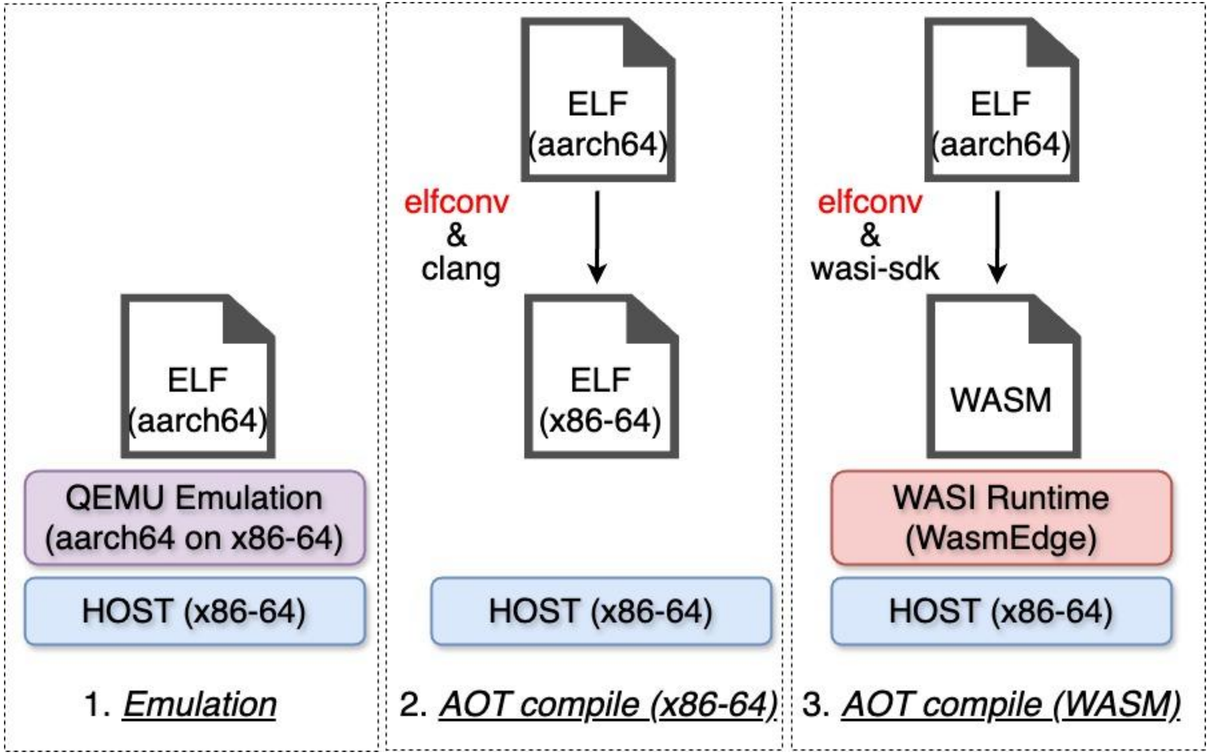
3. *AOT compile (WASM)*

   68.958 (MFLOPS)

ref.) compile from source code:

x86-64: 5527.070 (MFLOPS)

WASM: 2850.599 (MFLOPS)



Performance

1. *Emulation*

   195.115 (MFLOPS)

2. *AOT compile (x86-64)*

   200.177 (MFLOPS)

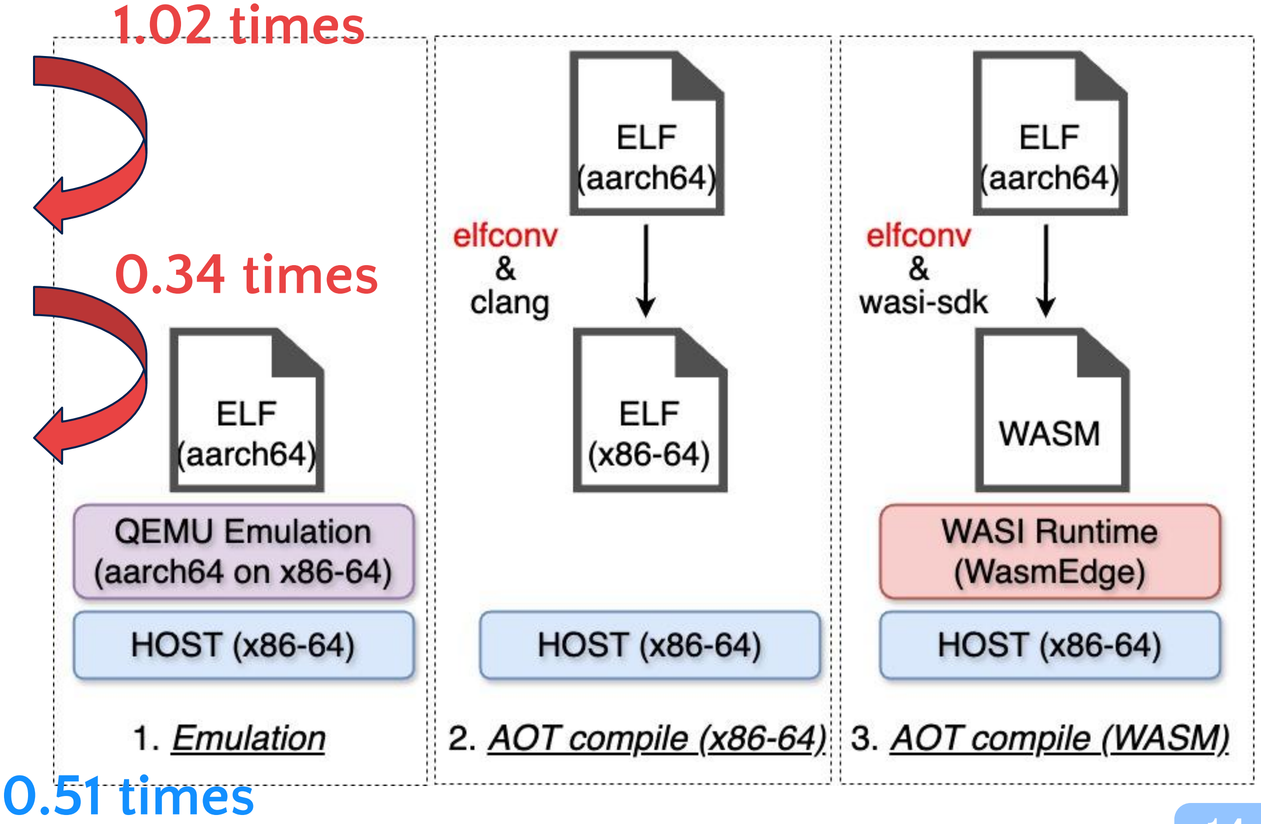
3. *AOT compile (WASM)*

   68.958 (MFLOPS)

ref.) compile from source code:

x86-64: 5527.070 (MFLOPS)

WASM: 2850.599 (MFLOPS)

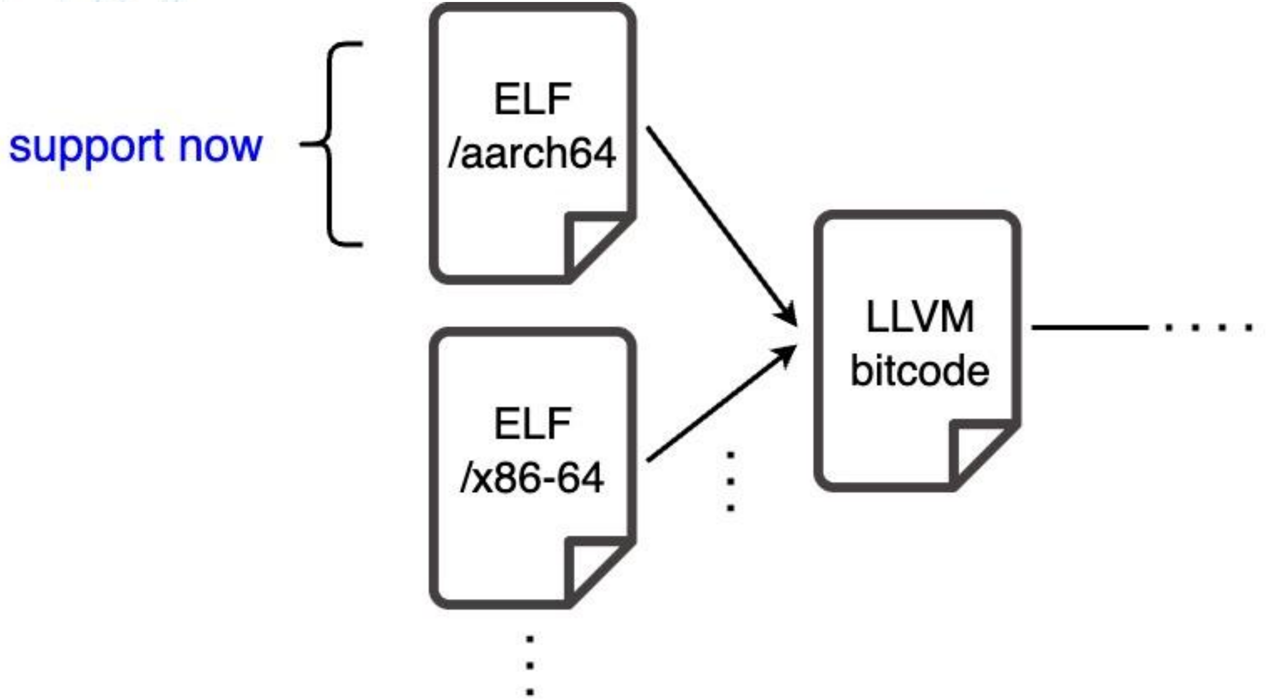


Future works

- append system calls emulation
  - a part of Linux system calls are implemented in the current version
  - Some system calls (e.g. fork, exec) are difficult to implement when targeting WASM
- support dynamic linking
  - statically linked ELF binary is suppored in the current version
- make the generated binary and LLVM bitcode more efficient
  - want to generate LLVM bitcode that runs faster than QEMU
  - want to make the translated WASM faster

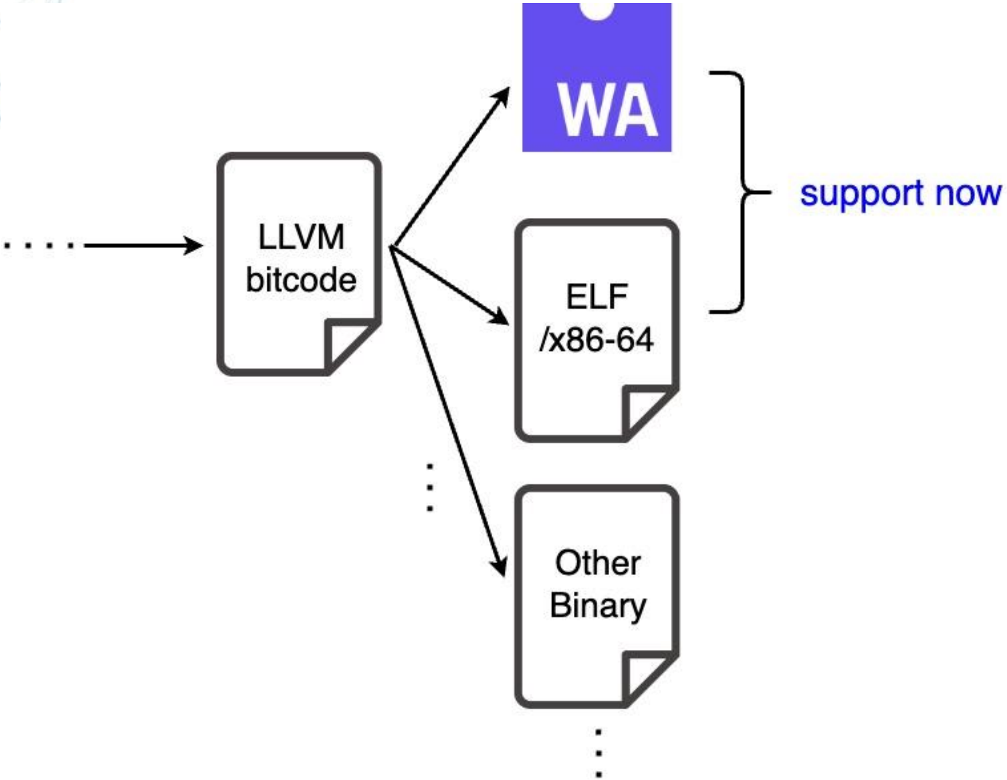
Future works

- translate ELF of other CPU architectures
  - only aarch64 is supported in the current vesion



Future works

- output other binary formats
  - WASM, ELF/x86-64 are supported in the current version



Future works

- Spread elfconv and integrate into existing ecosystem
  - Please hesitate to throw an issue or make PRs!

Related works

- elfconv is successor to **myAOT:** https://github.com/AkihiroSuda/myaot
  - Author: Akihiro Suda, NTT
  - An experimental AOT-ish compiler (Linux/riscv32 ELF → Linux/x86_64 ELF, Mach-O, WASM, ...)

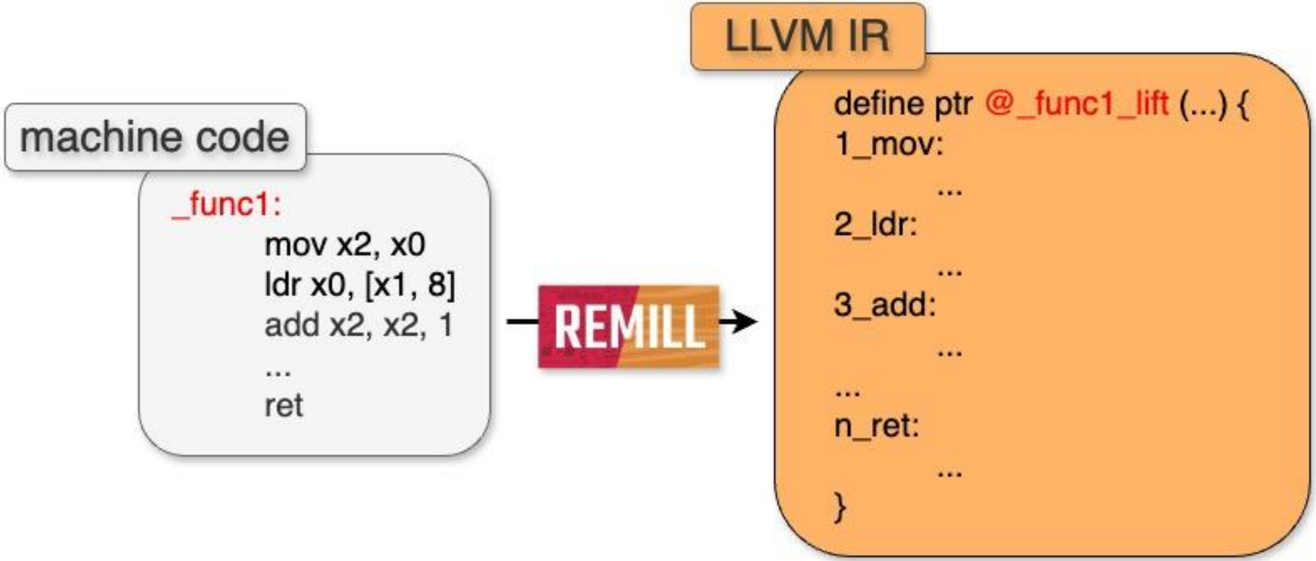
NTT

Questions?

repo: https://github.com/yomaytk/elfconv

How it works? (remill)

- A LLVM IR function consists of many *basic blocks*.
  - *basic block* is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit (e.g. 1_mov, 2_ldr, 3_add, …).
- convert a function to a LLVM IR function (e.g. _func1 -> @_func1_lift)
  - But, *elfconv-lifter* needs to detect every function from ELF



machine code

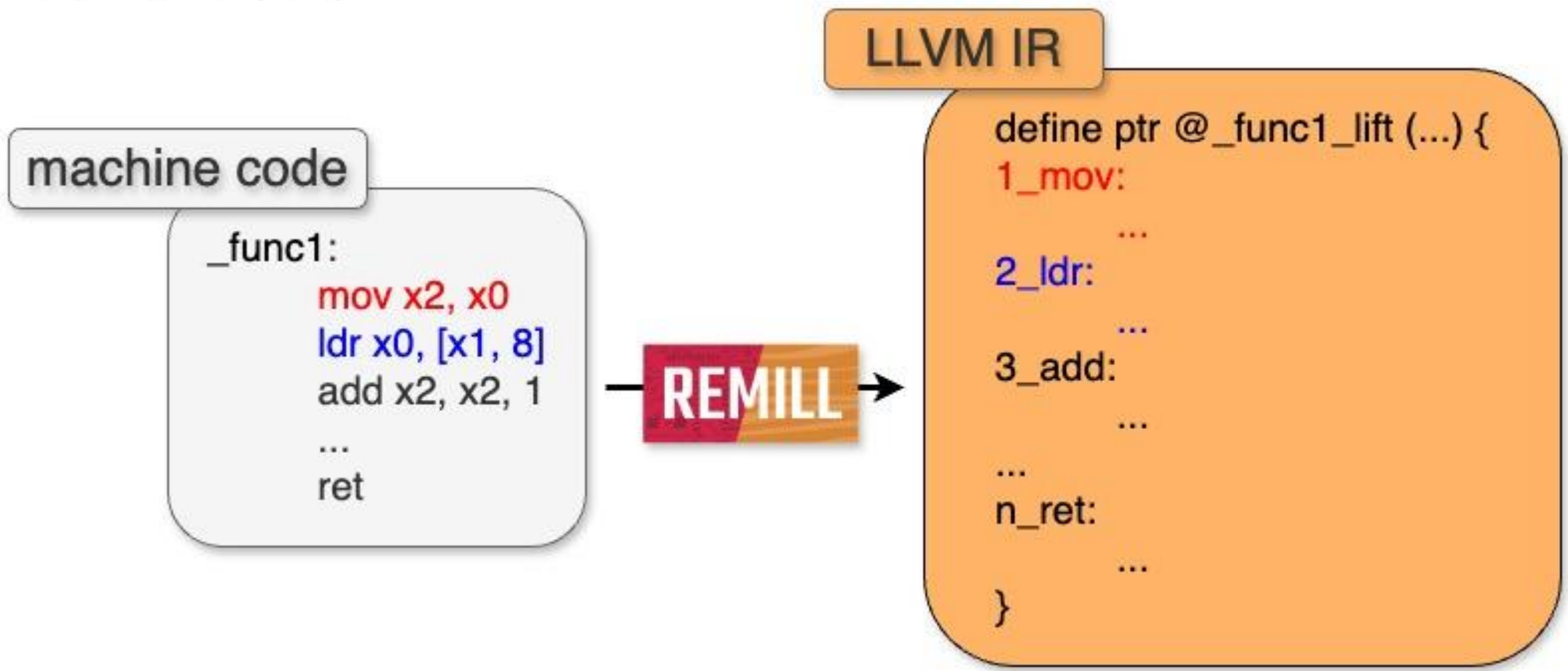
```
_func1:
        mov x2, x0
        ldr x0, [x1, 8]
        add x2, x2, 1
        ...
        ret
```

LLVM IR

```
define ptr @_func1_lift (...) {
1_mov:
        ...
2_ldr:
        ...
3_add:
        ...
...
n_ret:
        ...
}
```

How it works? (remill)

- convert a CPU instruction to a LLVM IR block (e.g. mov x2, x0 -> 1_mov)

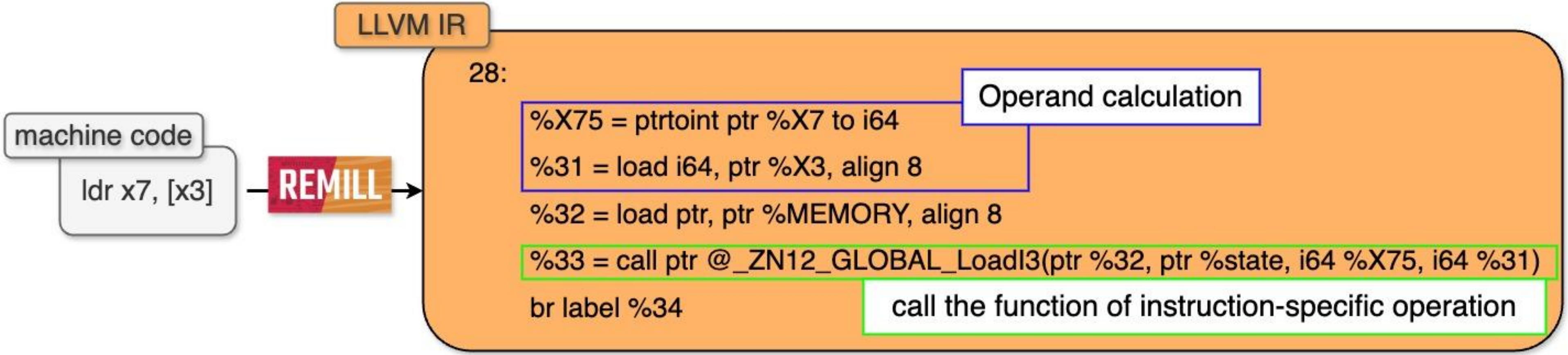


```
_func1:
        mov x2, x0
        ldr x0, [x1, 8]
        add x2, x2, 1
        ...
        ret
```

```
define ptr @_func1_lift (...) {
1_mov:
        ...
2_ldr:
        ...
3_add:
        ...
...
n_ret:
        ...
}
```

How it works? (remill)

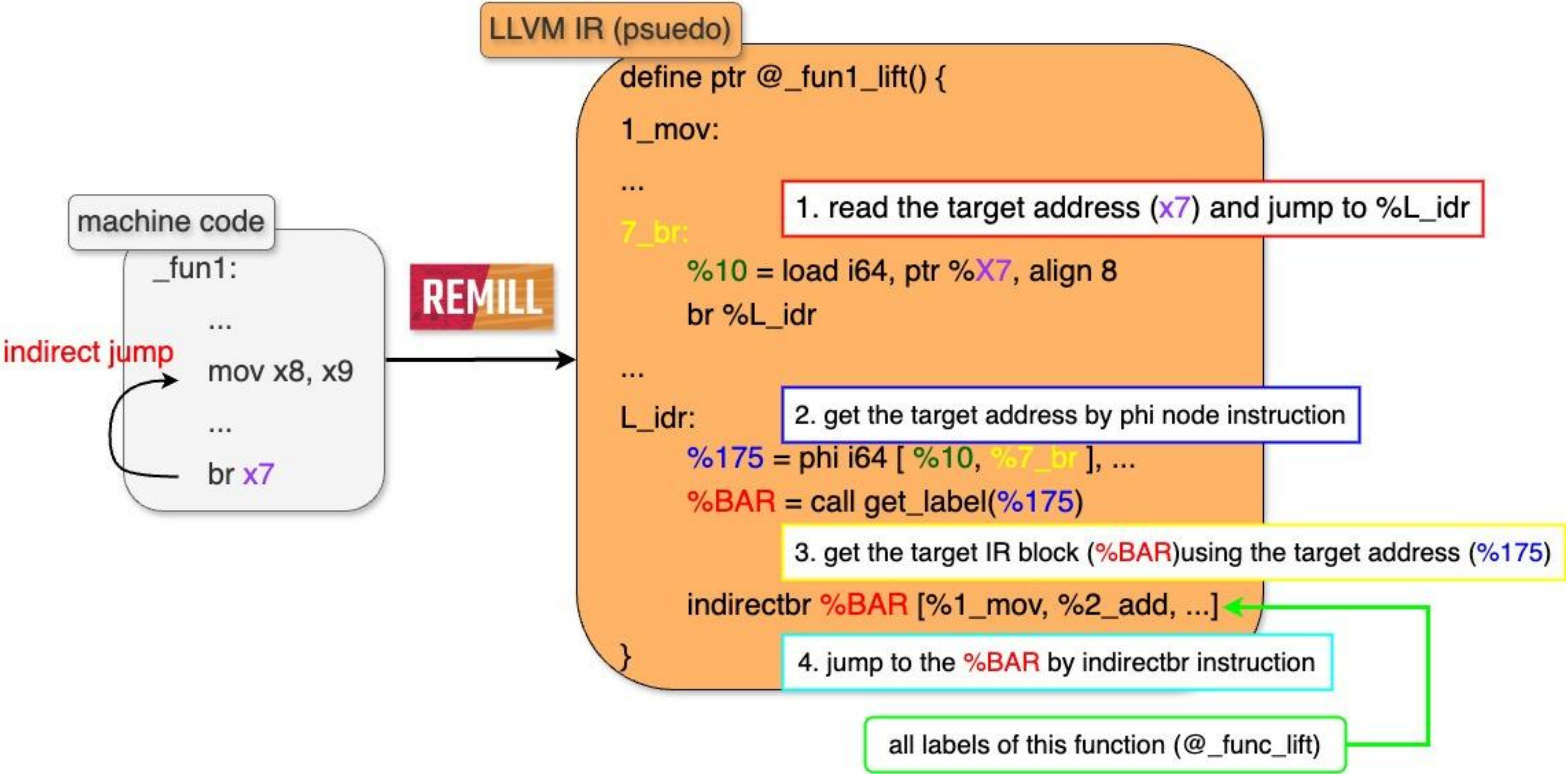
- convert a CPU instruction to a LLVM IR block
  - Operand calculation
  - call the function of the instruction-specific operation



```
ldr x7, [x3]
```

```
28:
    %X75 = ptrtoint ptr %X7 to i64
    %31 = load i64, ptr %X3, align 8
    %32 = load ptr, ptr %MEMORY, align 8
    %33 = call ptr @_ZN12_GLOBAL_LoadI3(ptr %32, ptr %state, i64 %X75, i64 %31)
    br label %34
```

How it works? (indirect jump)



machine code

```
_fun1:
    ...
    mov x8, x9
    ...
    br x7
```

indirect jump

LLVM IR (psuedo)

```
define ptr @_fun1_lift() {
1_mov:
...
7_br:
        %10 = load i64, ptr %X7, align 8
        br %L_idr
...
L_idr:
        %175 = phi i64 [ %10, %7_br ], ...
        %BAR = call get_label(%175)

        indirectbr %BAR [%1_mov, %2_add, ...]
}
```

1. read the target address (x7) and jump to %L_idr

2. get the target address by phi node instruction

3. get the target IR block (%BAR)using the target address (%175)

4. jump to the %BAR by indirectbr instruction

all labels of this function (@_func_lift)