RISC-V Binary Translation Prototyping & Evaluation

Yucheng Chen*, Xi Wang#
RISC-V International Opensource Lab
Tsinghua University

2021-9-27

RIOS

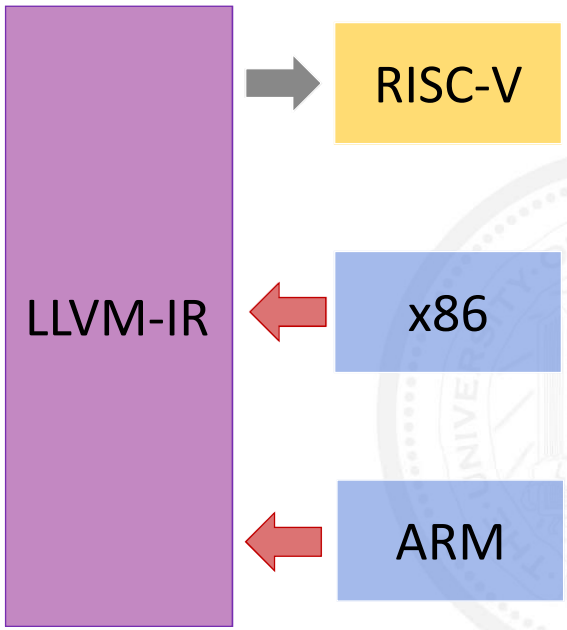
CONTENTS

RIOS

CONTENTS

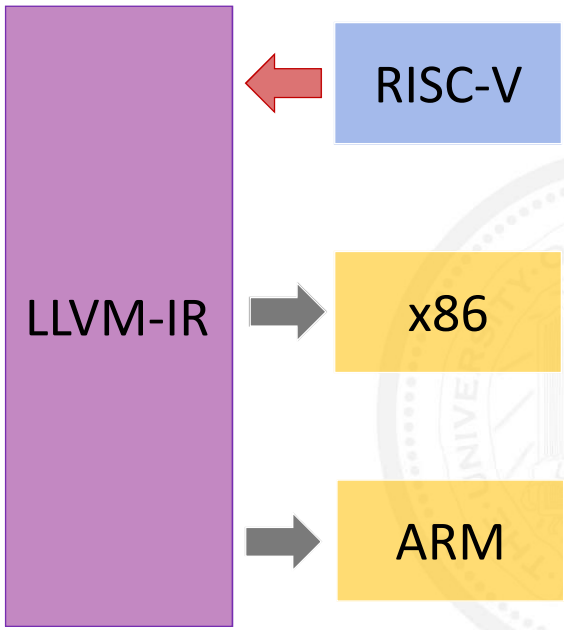
A High Level View of Our Work

- **Ultimate Objective:**
  - Translate x86 /ARM binary to IR that can be recompiled to RISC-V binary



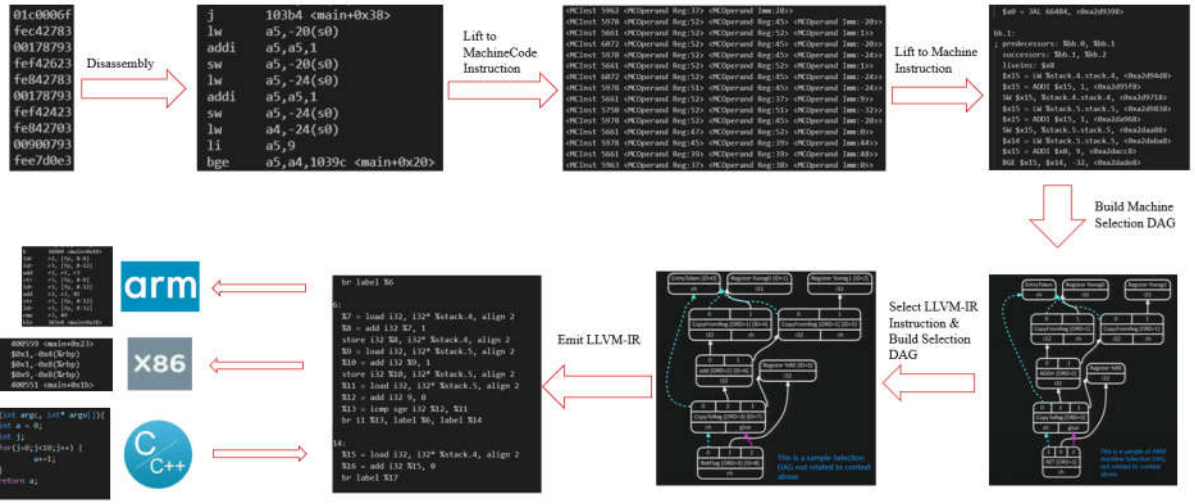
A High Level View of Our Work

- **Ultimate Objective:**
  - Translate x86/ARM binary to IR that can be recompiled to RISC-V binary
- **Motivation of Lifting RISC-V binary to IR:**
  - Technique of translating different ISA binary to IR is similar
  - Using x86 or ARM as a startup is unfriendly
    - A large number of instructions
    - Complicate instruction mechanism (e.g. side effect)
  - Choose RISC-V to prototype
    - RISC-V has reduced instructions and neat mechanism, which make it a good choice



A High Level View of Our Work

- **Workflow**
  - **RISC-V binary >> MCInst >> Machine Instruction >> Selection DAG >> LLVM-IR >> x86/ARM binary**

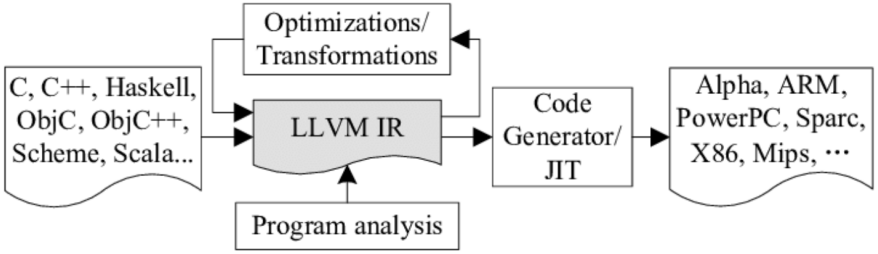


RIOS

CONTENTS

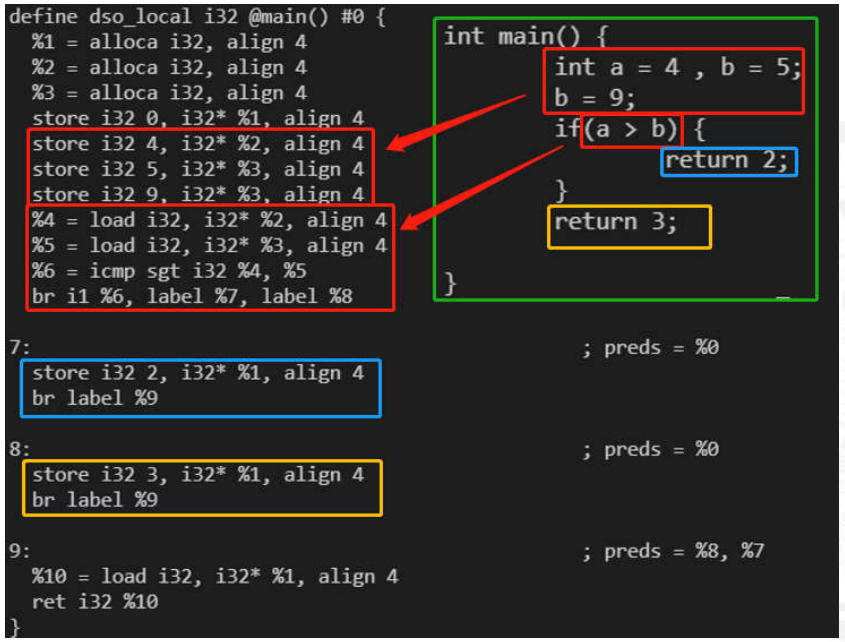
Introduction to LLVM-IR

- About LLVM
  - A compiler infrastructure
  - Frontend: translate source code to IR
  - LLVM IR: a intermediate representation independent of source code and target machine
  - Optimization: optimization pass can be performed on IR
  - backend(codegen): input LLVM-IR output target machine code



Introduction to LLVM-IR

- **LLVM-IR**
  - similar to assembly
  - strongly typed RISC instruction set
  - abstracts away most details of the target.
  - static single assignment

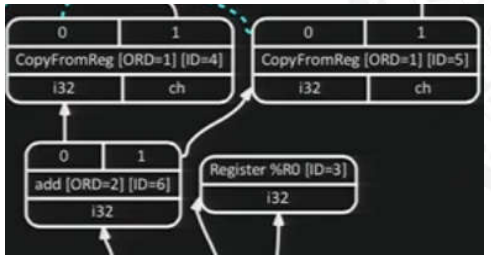


Introduction to LLVM-IR

- Other backend IR of LLVM
  - **MCInst**
    - A low level IR of LLVM
    - close to machine assembly format
    - Target independent, opcode isa-specific
  - **Machine Instruction**
    - More abstract than MCInst
    - Only keeps track of opcode number, a set of operands.
    - Organized by linkist, collected by Machine Basic Block
  - **SelectionDAG**
    - Higher abstraction than Machine Instruction
    - Provide topology info for instruction selection

```
0x00000000000000dc: <MCInst 5661
<MCOperand Reg:39> <MCOperand Reg:39>
<MCOperand Imm:-48>>
0x00000000000000e0: <MCInst 6072
<MCOperand Reg:45> <MCOperand Reg:39>
<MCOperand Imm:44>>
```

```
bb.0:
  liveins: $x1, $x8, $x10, $x11
  $x10 = ADDI %stack.1, 0
  $x11 = ADDI %stack.2, 0
  SW $x10, %stack.7.stack.7, <0x915d678>
  SW $x11, %stack.8.stack.8, <0x915d6f8>
```

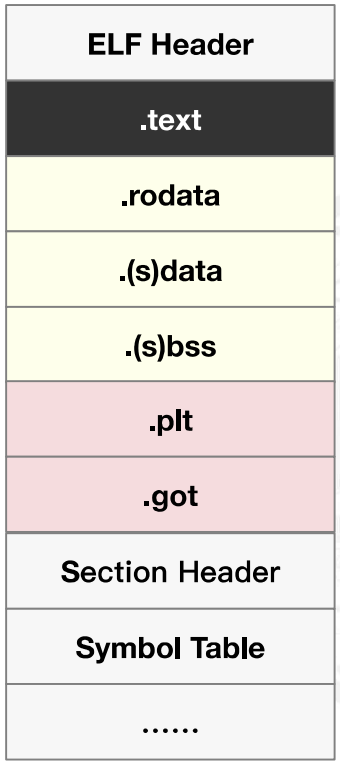


RIOS

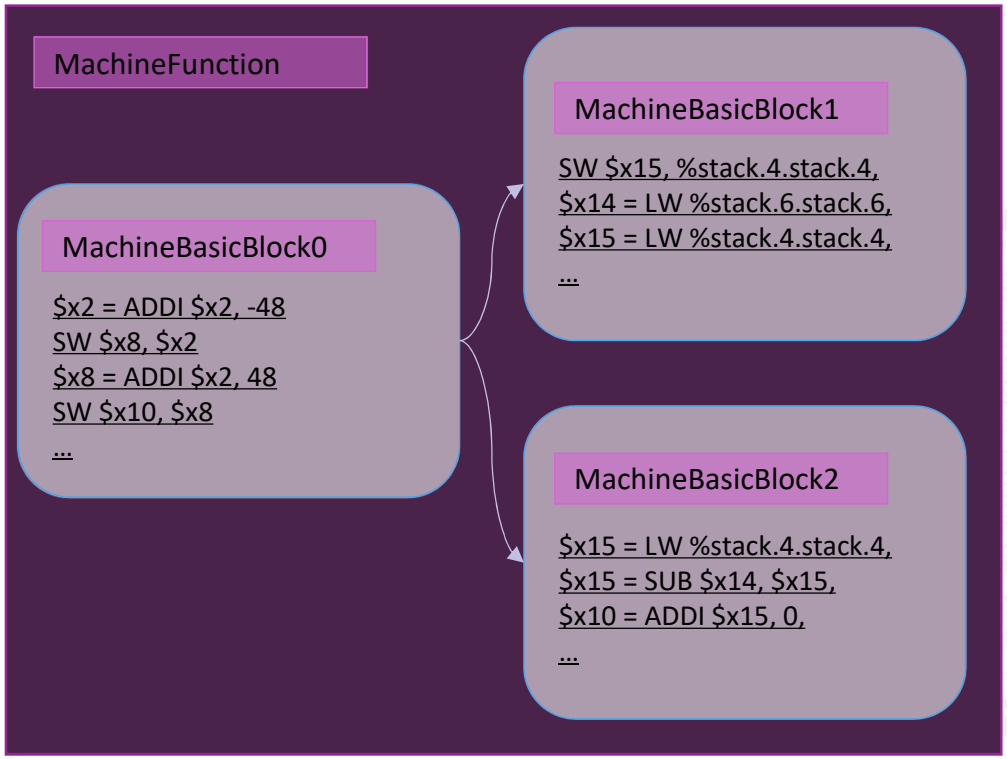
CONTENTS

Disassembly: Raise Binary to MCInst

- Read byte array of ELF, parse it into ObjectFile
- Read symbol table
- Use MCDisassembler.getInstruction to decode raw code
- Filter unnecessary section in elf file
  - e.g. .init,.fini,.line,.note
- Filter unnecessary code in elf file
  - e.g. _start, deregister_tm_clones, frame_dummy
  - Most of unnecessary code is linker-produced code.
- llvm-mctoll infra is used

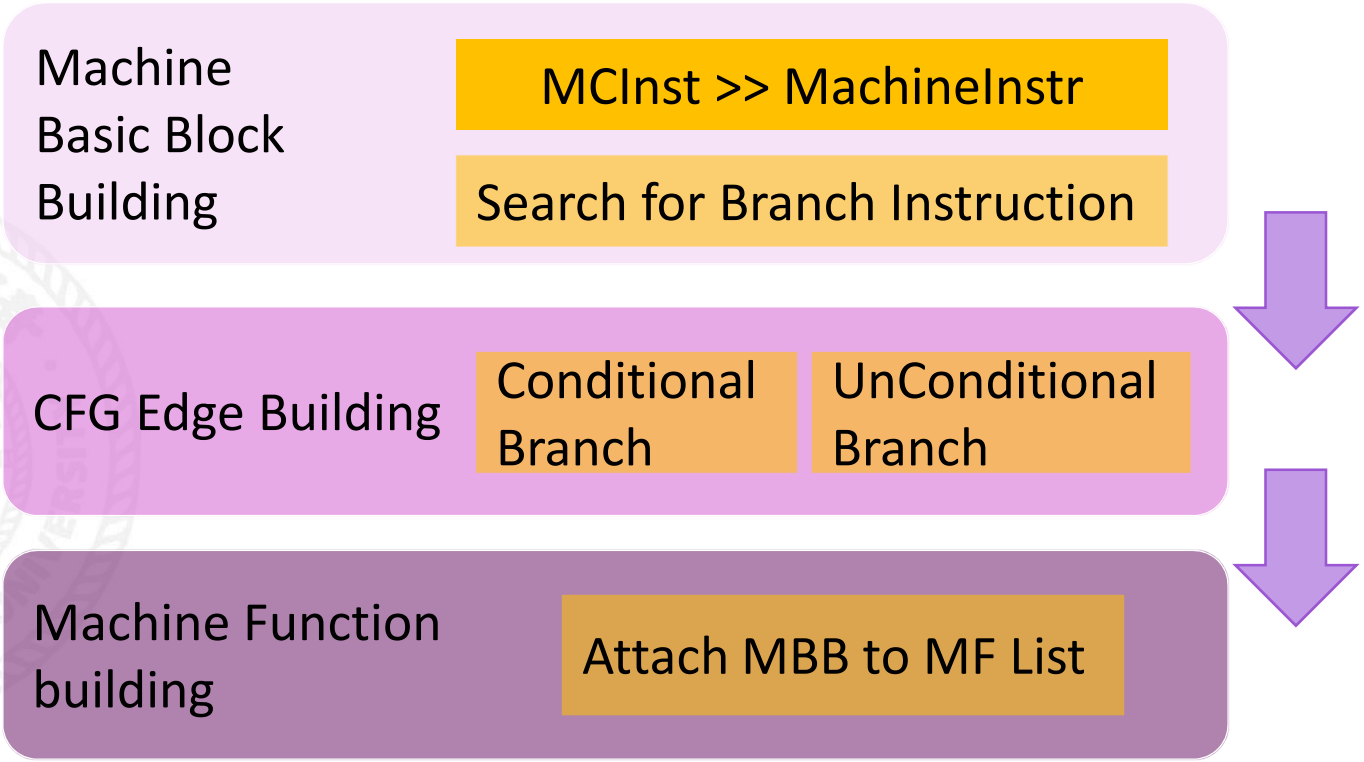


BuildCFG: Raise MCInst to MachineInstr



- MCInst
  - Organized by a list
  - Only contain opcode&operands
- Machine Instr
  - Organized by MBB and MF
  - Connect each other by linkist

BuildCFG: Raise MCInst to MachineInstr



RIOS

CONTENTS

Preprocessing on MachineInstr

```
bb.0:
  liveins: $x1, $x8
  $x2 = ADDI $x2, -32, <0x9882b48>
  SW $x1, $x2, 28, <0x9882c68>
  SW $x8, $x2, 24, <0x9882d88>
  $x8 = ADDI $x2, 32, <0x9882ea8>
  $x14 = LW $x3, -2020, <0x9882fc8>
  $x15 = LW $x3, -2024, <0x98830e8>
  $x15 = ADD $x14, $x15, <0x9883208>
  SW $x15, $x8, -20, <0x9883328>
  $x10 = LW $x8, -20, <0x9883448>
  $x1 = JAL -72, <0x9883568>
  SW $x10, $x8, -24, <0x9883688>
  $x14 = LW $x3, -2024, <0x98837a8>
  $x15 = LW $x8, -24, <0x98838c8>
  $x15 = SUB $x14, $x15, <0x98849f8>
  $x10 = ADDI $x15, 0, <0x9884b18>
  $x1 = LW $x2, 28, <0x9884c38>
  $x8 = LW $x2, 24, <0x9884d58>
  $x2 = ADDI $x2, 32, <0x9884e78>
  $x0 = JALR $x1, 0, <0x9884f98>
```

MachineInstr Revise

Eliminate Prolog&Epilog

Raise Arguments

Building Frame

```
bb.0:
  liveins: $x1, $x8
  $x14 = LW &f, <0x9882fc8>
  $x15 = LW &i, <0x98830e8>
  $x15 = ADD $x14, $x15, <0x9883208>
  SW $x15, %stack.3.stack.3, <0x9883328>
  $x10 = LW %stack.3.stack.3, <0x9883448>
  $x1 = JAL 66428, <0x9883568>
  SW $x10, %stack.4.stack.4, <0x9883688>
  $x14 = LW &i, <0x98837a8>
  $x15 = LW %stack.4.stack.4, <0x98838c8>
  $x15 = SUB $x14, $x15, <0x98849f8>
  $x10 = ADDI $x15, 0, <0x9884b18>
  $x0 = JALR $x1, 0, <0x9884f98>
```

Preprocessing on MachineInstr
MachineInstr Revise

- MachineInstr Revise will **examine every instruction**, to find some pattern that might **hinder** subsequent lifting work and do some **modification or preparation**
- Some example:
  - Global variable indirect access handling
  - Jump address replacement
  - PLT(Procedure Linkage Table) entry pre-calc, symbol extraction
  - …

Preprocessing on MachineInstr

MachineInstr Revise

```
bb.0:
  liveins: $x1, $x8
  $x2 = ADDI $x2, -32, <0x9882b48>
  SW $x1, $x2, 28, <0x9882c68>
  SW $x8, $x2, 24, <0x9882d88>
  $x8 = ADDI $x2, 32, <0x9882ea8>
  $x14 = LW $x3, -2020, <0x9882fc8>
  $x15 = LW $x3, -2024, <0x98830e8>
  $x15 = ADD $x14, $x15, <0x9883208>
  SW $x15, $x8, -20, <0x9883328>
  $x10 = LW $x8, -20, <0x9883448>
  $x1 = JAL -72, <0x9883568>
  SW $x10, $x8, -24, <0x9883688>
  $x14 = LW $x3, -2024, <0x98837a8>
  $x15 = LW $x8, -24, <0x98838c8>
  $x15 = SUB $x14, $x15, <0x98849f8>
  $x10 = ADDI $x15, 0, <0x9884b18>
  $x1 = LW $x2, 28, <0x9884c38>
  $x8 = LW $x2, 24, <0x9884d58>
  $x2 = ADDI $x2, 32, <0x9884e78>
  $x0 = JALR $x1, 0, <0x9884f98>
```

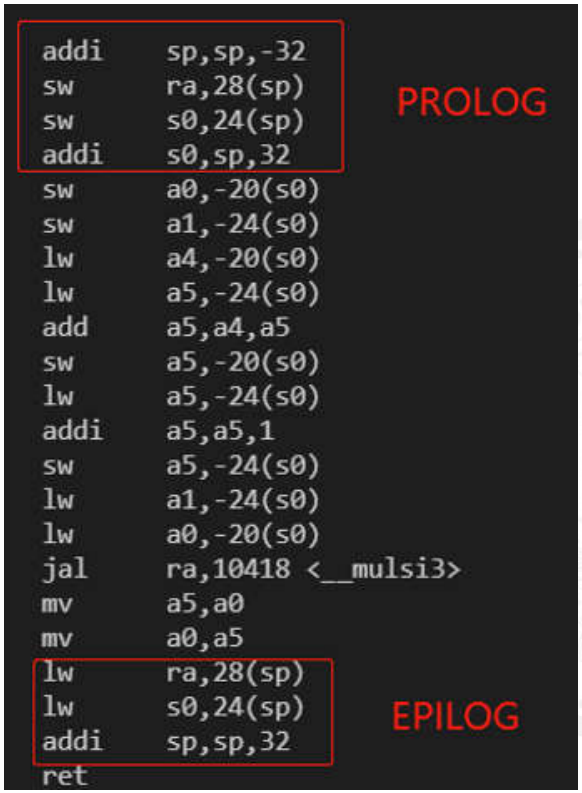
```
bb.0:
  liveins: $x1, $x8
  $x2 = ADDI $x2, -32, <0x9882b48>
  SW $x1, $x2, 28, <0x9882c68>
  SW $x8, $x2, 24, <0x9882d88>
  $x8 = ADDI $x2, 32, <0x9882ea8>
  $x14 = LW &f, <0x9882fc8>
  $x15 = LW &i, <0x98830e8>
  $x15 = ADD $x14, $x15, <0x9883208>
  SW $x15, $x8, -20, <0x9883328>
  $x10 = LW $x8, -20, <0x9883448>
  $x1 = JAL 66428, <0x9883568>
  SW $x10, $x8, -24, <0x9883688>
  $x14 = LW &i, <0x98837a8>
  $x15 = LW $x8, -24, <0x98838c8>
  $x15 = SUB $x14, $x15, <0x98849f8>
  $x10 = ADDI $x15, 0, <0x9884b18>
  $x1 = LW $x2, 28, <0x9884c38>
  $x8 = LW $x2, 24, <0x9884d58>
  $x2 = ADDI $x2, 32, <0x9884e78>
  $x0 = JALR $x1, 0, <0x9884f98>
```

Observation:

1. Indirect access of reg gp(x3) is raised to a global symbol
   e.g. &f, &i
2. Original relative jump target address is replaced by an absolute address
   e.g. JAL 66428

Preprocessing on MachineInstr
Eliminate Prolog & Epilog

- Prolog:
  - Head of the calling convention
    - Adjusting StackPointer(SP)
    - Saving old & Assigning new FramePointer(FP)
- Epilog:
  - Tail of the calling convention
    - Loading ReturnAddress(RA)
    - Restoring FP & SP
- Why Eliminate them?
    - Machine assembly use prolog and epilog to construct a **physics stack**
    - LLVM stack is **abstract stack**, so prolog and epilog are no longer required



Preprocessing on MachineInstr
Eliminate Prolog & Epilog

```
bb.0:
  liveins: $x1, $x8
  $x2 = ADDI $x2, -32, <0x9882b48>
  SW $x1, $x2, 28, <0x9882c68>
  SW $x8, $x2, 24, <0x9882d88>
  $x8 = ADDI $x2, 32, <0x9882ea8>
  $x14 = LW &f, <0x9882fc8>
  $x15 = LW &i, <0x98830e8>
  $x15 = ADD $x14, $x15, <0x9883208>
  SW $x15, $x8, -20, <0x9883328>
  $x10 = LW $x8, -20, <0x9883448>
  $x1 = JAL 66428, <0x9883568>
  SW $x10, $x8, -24, <0x9883688>
  $x14 = LW &i, <0x98837a8>
  $x15 = LW $x8, -24, <0x98838c8>
  $x15 = SUB $x14, $x15, <0x98849f8>
  $x10 = ADDI $x15, 0, <0x9884b18>
  $x1 = LW $x2, 28, <0x9884c38>
  $x8 = LW $x2, 24, <0x9884d58>
  $x2 = ADDI $x2, 32, <0x9884e78>
  $x0 = JALR $x1, 0, <0x9884f98>
```

```
bb.0:
  liveins: $x1, $x8
  $x14 = LW &f, <0x9882fc8>
  $x15 = LW &i, <0x98830e8>
  $x15 = ADD $x14, $x15, <0x9883208>
  SW $x15, $x8, -20, <0x9883328>
  $x10 = LW $x8, -20, <0x9883448>
  $x1 = JAL 66428, <0x9883568>
  SW $x10, $x8, -24, <0x9883688>
  $x14 = LW &i, <0x98837a8>
  $x15 = LW $x8, -24, <0x98838c8>
  $x15 = SUB $x14, $x15, <0x98849f8>
  $x10 = ADDI $x15, 0, <0x9884b18>
  $x0 = JALR $x1, 0, <0x9884f98>
```

- Observation:
  - Head & Tail of the calling convention has been removed

Preprocessing on MachineInstr
Raising Arguments

- Why Raise Arguments
  - In order to change
    register passing though function
    into
    parameter passing through function

RISC-V Assembly Calling Convention

Caller Load Args

```
int main() {
        foo(1,2,3);
}
```

```
li      a2,3
li      a1,2
li      a0,1
jal     ra,1037c <foo>
```

Register a0,a1,a2 Live Through

Callee Store Args

```
int foo(int x, int y, int z) {
        x = x + 1;
        z = z + 3;
        return y+z;
}
```

```
addi    sp,sp,-32
sw      s0,28(sp)
addi    s0,sp,32
sw      a0,-20(s0)
sw      a1,-24(s0)
sw      a2,-28(s0)
```

LLVM-IR Calling Mechanism

Caller Fill Parameter into foo(…)

```
define i32 @main() {
EntryBlock:
  ...
  %7 = call i32 @foo(i32 %6, i32 %5, i32 %4)
  ...
}
```

C A L L

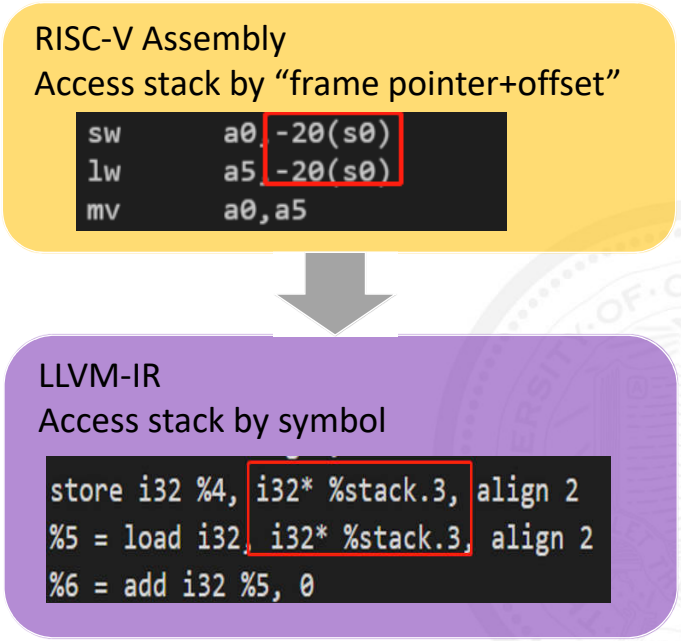
Callee receive Parameter from foo(..)

```
define i32 @foo(i32 %arg.1, i32 %arg.2, i32 %arg.3) {
EntryBlock:
  %4 = add i32 %arg.1, 0
  %5 = add i32 %arg.2, 0
  %6 = add i32 %arg.3, 0
  store i32 %4, i32* %stack.5, align 2
  store i32 %5, i32* %stack.6, align 2
  store i32 %6, i32* %stack.7, align 2
  %7 = load i32, i32* %stack.5, align 2
```

Parameter filling based argument passing

Preprocessing on MachineInstr
Building Frame

- Why Build Frame?
  - RISC-V assembly use “fp(frame pointer)+offset” to access stack object by VMA (physic stack)
  - LLVM-IR use alloca symbol to access stack object (abstract stack)
- How to build frame?
  - we will **extract stack information** from MachineInstr raw pattern and build LLVM abstract stack frame.
  - we will also **replace** machine stack access pattern by LLVM FrameIndex symbol



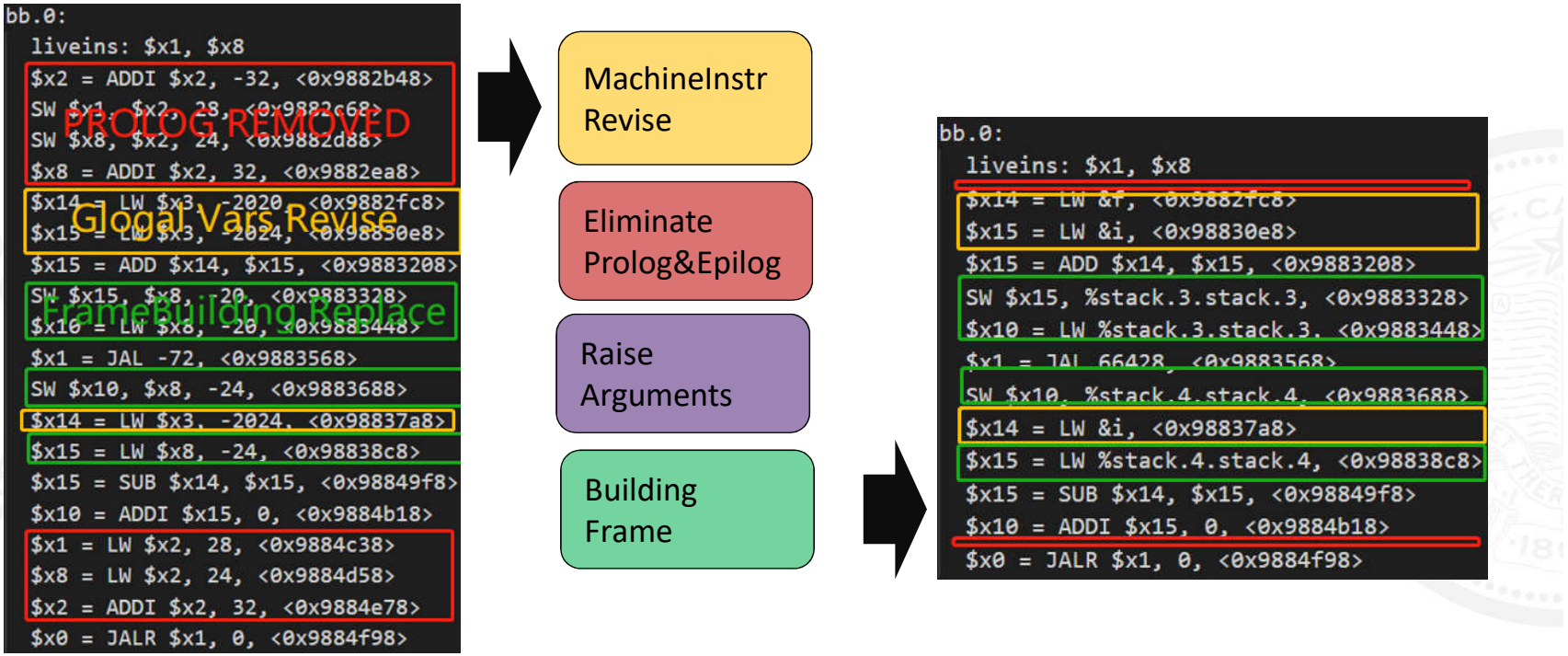
Preprocessing on MachineInstr
Building Frame

```
bb.0:
  liveins: $x1, $x8
  $x14 = LW &f, <0x982afc8>
  $x15 = LW &i, <0x982b0e8>
  $x15 = ADD $x14, $x15, <0x982b208>
  SW $x15, $x8, -20, <0x982b328>
  $x10 = LW $x8, -20, <0x982b448>
  $x1 = JAL 66428, <0x982b568>
  SW $x10, $x8, -24, <0x982b688>
  $x14 = LW &i, <0x982b7a8>
  $x15 = LW $x8, -24, <0x982b8c8>
  $x15 = SUB $x14, $x15, <0x982c9f8>
  $x10 = ADDI $x15, 0, <0x982cb18>
  $x0 = JALR $x1, 0, <0x982cf98>
```

```
bb.0:
  liveins: $x1, $x8
  $x14 = LW &f, <0x982afc8>
  $x15 = LW &i, <0x982b0e8>
  $x15 = ADD $x14, $x15, <0x982b208>
  SW $x15, %stack.3.stack.3, <0x982b328>
  $x10 = LW %stack.3.stack.3, <0x982b448>
  $x1 = JAL 66428, <0x982b568>
  SW $x10, %stack.4.stack.4, <0x982b688>
  $x14 = LW &i, <0x982b7a8>
  $x15 = LW %stack.4.stack.4, <0x982b8c8>
  $x15 = SUB $x14, $x15, <0x982c9f8>
  $x10 = ADDI $x15, 0, <0x982cb18>
  $x0 = JALR $x1, 0, <0x982cf98>
```

- Observation
  - frame pointer access pattern is replace by LLVM FrameIndex

Preprocessing on MachineInstr

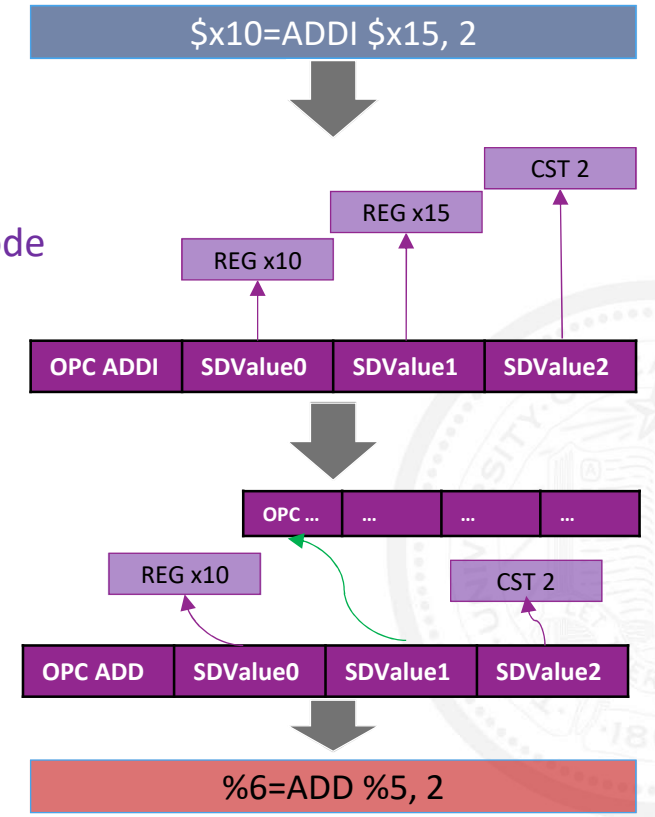


RIOS

CONTENTS

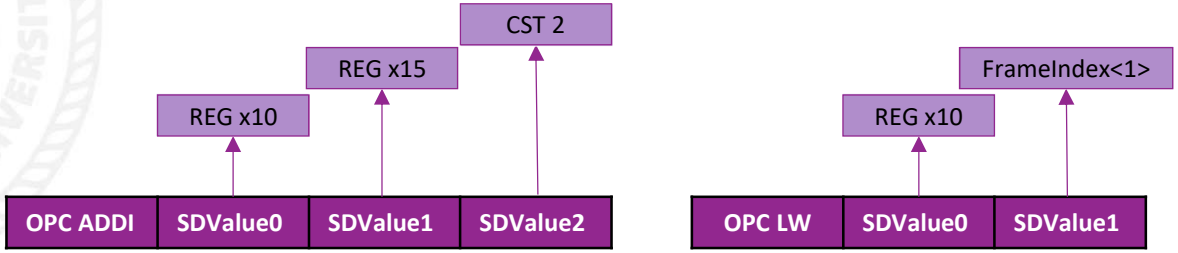
SelectionDAG & Emitting IR

- **Build Machine SelectionDAG Node(SDNode)**
  - One Machine Instruction=>1 Machine Instruction SDNode + multiple Machine Operand SDNode
- **Select Machine SDNode to IR SDNode**
  - Opcode Semantic Lift up
  - **Connect** IR Instruction SDNode
- **Emitting IR Through IRBuilder**
  - Get IR value at each SDNode
  - Use IRBuilder.Create to construct IR Instruction



SelectionDAG & Emitting IR
Build Machine SelectionDAG Node

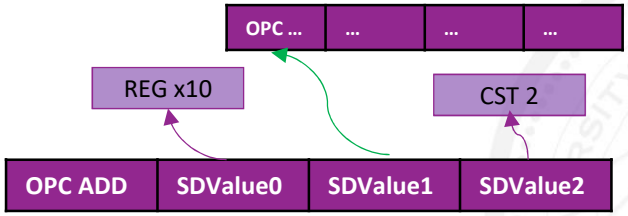
- Why Translate MachineInstr to Machine SDNode?
  - MachineInstr is independent of each other **(ISOLATION)**
  - Machine Oprands is contained in one MachinInstr, no connection to other MachineInstr **(ISOLATION)**
  - In DAG, MachineInstr, MachineOperand will present as SDNode
  - SDNode can connect each other, describe the dataflow



SelectionDAG & Emitting IR
Select MachineInstr SDNode to IR SDNode

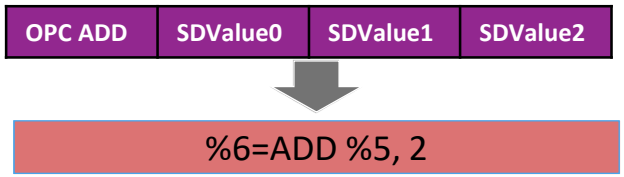
- So…what's the different?
  - MachineInstr SDNode's Opcode **is same as** MachineInstr
  - IR SDNode's Opcode **is same as** LLVM-IR
  - IR SDNode has High level of abstraction than MachineInstr SDNode
    - e.g. ADD, ADDI => ADD

- How to raise it?
  - **Define** new SDNode of IR opcode with corresponding operands
  - Use new SDNode **replace** the old one
  - **Redefine** register SDNode, to connect between SDNode



SelectionDAG & Emitting IR
Emitting IR Through IRBuilder

- Final Step!
- Extract LLVM value from IR SDNode.
- Invoke IRBuilder.Create to emit LLVM IR
- llvm-llc backend to generate target binary

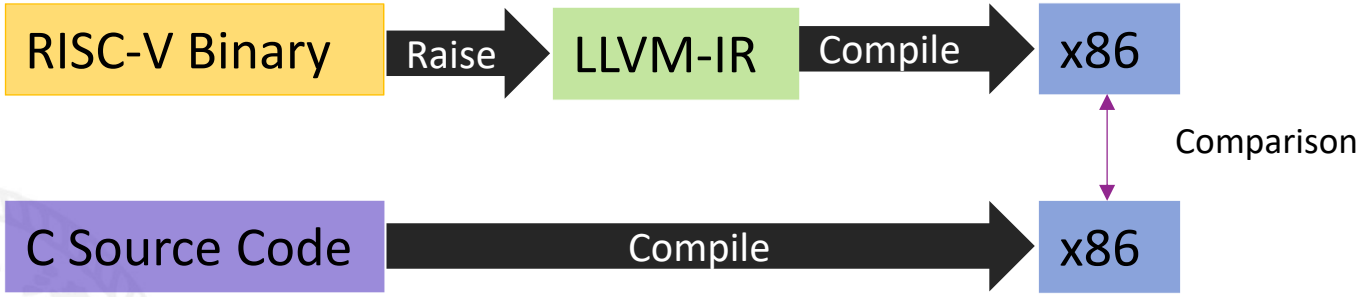


RIOS

CONTENTS

Evaluation

Compare with Native compiled code



Example: Fibonacci Kernel N=10^9. Translated binary is 1.7x faster than native binary

```
j       103d8 <main+0x5c>
lw      a4,-24(s0)
lw      a5,-28(s0)
add     a5,a4,a5
sw      a5,-32(s0)
lw      a5,-28(s0)
sw      a5,-24(s0)
lw      a5,-32(s0)
sw      a5,-28(s0)
lw      a5,-20(s0)
addi    a5,a5,1
sw      a5,-20(s0)
lw      a4,-20(s0)
lui     a5,0x3b9ad
addi    a5,a5,-1536 # 3b9aca00
bge     a5,a4,103ac <main+0x30>
```

RISC-V Binary

```
mov     0x80(%rsp),%eax
mov     0x84(%rsp),%ecx
add     %eax,%ecx
mov     %ecx,0x7c(%rsp)
mov     %eax,0x84(%rsp)
mov     %ecx,0x80(%rsp)
mov     0x88(%rsp),%eax
inc     %eax
mov     %eax,0x88(%rsp)
cmp     $0x3b9aca01,%eax
jl      4005a0 <main+0x60>
```

Translated x86 binary From RISC-V

```
real    0m2.008s
user    0m2.001s
sys     0m0.002s
```

Translated x86 binary from RISC-V

VS

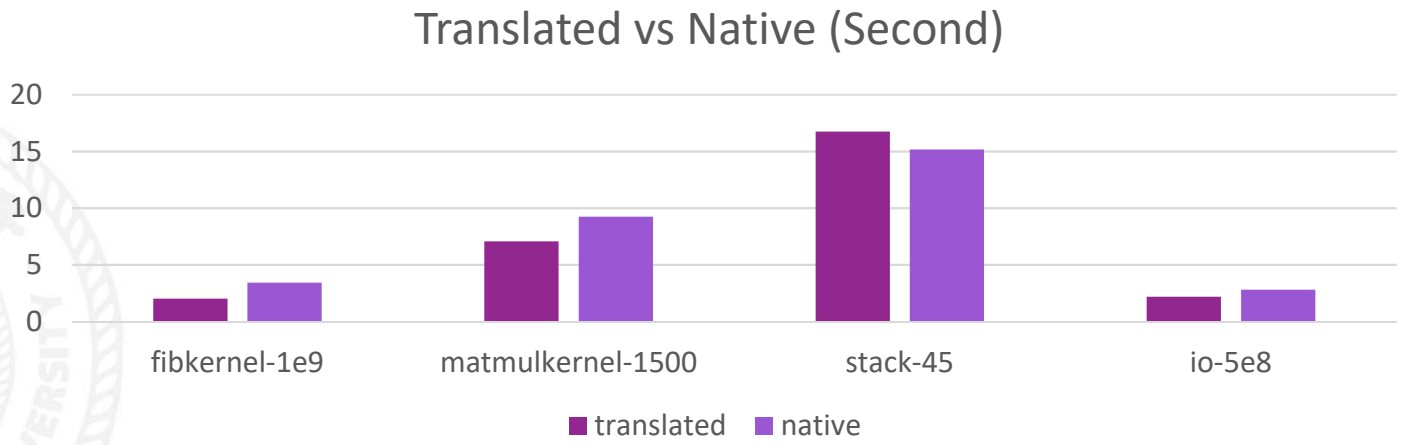
```
real    0m3.423s
user    0m3.413s
sys     0m0.001s
```

```
jmp     40057a <main+0x44>
mov     -0x8(%rbp),%edx
mov     -0xc(%rbp),%eax
add     %edx,%eax
mov     %eax,-0x10(%rbp)
mov     -0xc(%rbp),%eax
mov     %eax,-0x8(%rbp)
mov     -0x10(%rbp),%eax
mov     %eax,-0xc(%rbp)
addl    $0x1,-0x4(%rbp)
cmpl    $0x3b9aca00,-0x4(%rbp)
jle     40055f <main+0x29>
```

Native Compiled x86 binary

Native GCC output x86 binary

Evaluation
Compare with Native compiled code



- Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz
- Compiler flags are disabled in both RISC-V and x86 compilation

RIOS

CONTENTS

Conclusion

- Our prototype is an inverse procedure of compiler backend
- Statically recompiled binaries achieved competitive performance against native binaries
- TODOs
  - Cover more use case
  - Raise x86/ARM binaries to IR. Build the “Rosetta” of RISC-V.