Implementing an ISA in Ghidra

Presented by Mike J. Bell & Tracy Mosley
June 19, 2020 at Fortego U
Implementing an ISA in Ghidra

A Visual Guide
Who Are We?

Mike J. Bell
Vulnerability Researcher and former Ghidra Team Member

Tracy Mosley
Embedded Developer and Reverse Engineer

Why This Talk?

Submitted for a related talk
For REcon 2020...whoops

Perhaps some other time...
But we figured we’d lay out the basics here!
So you need to model a processor, eh?

- From inception, ISA-independent
- Processor “languages”, like most of Ghidra, are extensions
- Myriad components that come together to describe a processor ISA
- Comprehensive, holistic approach to modeling
- Feeds almost all aspects of the software analysis, from loading to disassembly to decompilation
- Must write using some Domain Specific Language (DSL), some XML, even custom Java classes
YAAAAS Sleigh!

- Sleigh is a DSL that has three specific types of information
  - Architectural definitions
  - Pattern matching and display components
  - Semantic sections
- Located under data/languages/X.slaspec (also could include .sinc files)
- Full documentation available Ghidra/docs/languages/html/sleigh.html
Sleigh: Architecture information

- Specify
  - Endianness
  - Alignment
  - Address spaces
  - Registers
    - Context
  - Instruction parsing tokens with their fields
**Example: Sleigh Architecture**

```plaintext
define endian=little;
define alignment=2;

define space ram type=ram_space wordsize=2 size=4 default;
define space register type=register_space wordsize=2 size=4;

define register offset=0 size=4
[ R0  R1  R2  R3  R4  R5  R6  R7
  R8  R9  R10 R11 R12 R13 R14 R15
  R16 R17 R18 R19 R20 R21 R22 R23
  R24 R25 R26 R27 R28  ST  RA  PC ];

define register offset=0 size=8
[ R00_01 R02_03 R04_05 R06_07
  R08_09 R10_11 R12_13 R14_15
  R16_17 R18_19 R20_21 R22_23
  R24_25 R26_27 R28_ST RA_PC ];

define register offset=4 size=8
[ R01_02 R03_04 R05_06 R07_08
  R09_10 R11_12 R13_14 R15_16
  R17_18 R19_20 R21_22 R23_24
  R25_26 R27_28 ST_RA PC_R0 ];
```
Sleigh: Tokens form the basis for pattern matching

- Token has a size in bits
- Tokens are comprised of contiguous-range bit fields (named “Fields”, obvi)
- Fields are used in pattern matching, and must be from the same token
- Multiple different tokens with the same or different sizes can be specified
- Variable length instructions are supported with token concatenation
Example: Sleigh Tokens and Fields

```c
define token instr(16)
    op1215 = (12, 15)
    op0811 = (8, 11)
    op0303 = (3, 3)
    rs    = (4, 7)
    cc0002 = (0, 2)
;
```

Token name “instr”

16 bit wide token

Inclusive contiguous bit ranges

Field names
Sleigh: Constructors for instructions

- Constructors are the top level pattern matching device for decoding
- They’re not named
- Use subconstructors to handle common patterns

Display in Listing: \texttt{mnemonic \textless print pieces\textgreater \textit{is} \textless pattern\textgreater \textless context block\textgreater \textless semantic block\textgreater}
Sleigh: Subconstructors for support

- Subconstructors abstract reusable patterns and concepts
  - Addressing modes (anchor scalars into absolute addresses, registers, indirect offsets, etc.)
  - Branch/call condition codes
  - Extract immediate values from fields
- Unlike constructors, they are named

```
name: <print pieces>
block>
```

* Keyword
  - Name!
  - Displayed in Listing
  - How to match
  - Performs match-time calculations
  - Pcode
Sleigh: Semantics (pcode) for the decompiler!

- **P-code**
  - Machine independent Intermediate Representation (IR)
  - Static Single Assignment (SSA)
  - 50+ expression operators that implement typical processor functions
  - Can create custom placeholder operations for unmodelable instructions

- See [Ghidra/docs/languages/html/pcoderef.html](https://Ghidra/docs/languages/html/pcoderef.html)
Example: pcode

...{
    local tmp:4 = 0;
    <loopstart>
        *[ram]:2 (RegA_3n_0711 + tmp) = *[ram]:2 (RegB_3n_1216 + tmp);
        tmp = tmp + 1;
        if (tmp < RegC_3n_1721) goto <loopstart>,
    }

Initialize a local varnode of size 4 bytes
Create a label for looping or branching
"Cast" expression to a 2-byte pointer in RAM
Looks almost like C!
Let’s see a complete example in action!

- **Modeling a conditional branch instruction**
  - Indirect register (branch to the address stored in the register)
  - Only do so if the condition code is met at this instruction
  - Disassembly should show the correct text
  - Branch should not appear in “always branch” (unconditional) case
Example: conditional branching (architecture)

```c
define token instr(16)
    op1215   = (12, 15)
    op0811   = (8, 11)
    op0303   = (3, 3)
    rs       = (4, 7)
    cc0002   = (0, 2)
;
attach variables [ rs ] [
    r0  r1  r2  r3  r4  r5  r6  r7
    r8  r9  r10 r11 r12  sp  lr  pc
];
```

- **Opcode fields:** to select which instruction mnemonic
- **Field rs:** will attach to registers (see below)
- **Condition code field:** selects which conditional execution
- **Field rs:** is a selector into the register file
Example: conditional branching (subconstructors)

RS: rs is rs { export rs; }
Example: conditional branching (subconstructors)

```plaintext
RS: rs is rs { export rs; }

CC: "eq" is cc0002=0x0 { export Z; }
CC: "ne" is cc0002=0x1 { tmp = !Z; export tmp; }
CC: "lt" is cc0002=0x2 { tmp = N != V; export tmp; }
CC: "le" is cc0002=0x3 { tmp = Z || (N != V); export tmp; }
CC: "lo" is cc0002=0x4 { export C; }
CC: "mi" is cc0002=0x5 { export N; }
CC: "vs" is cc0002=0x6 { export V; }
CC: "" is cc0002=0x7 { export 1:1; }

COND: CC is CC { if (!CC) goto inst_next; }
COND: CC is CC & cc0002=0x7 { } # unconditional
```

Idempotent register subconstructor
Table of condition codes
Special case subconstructor
Example: conditional branching (constructor)

```pascal
:br^COND RS is COND & RS & op1215=0xf & op0811=0x0 & op0303=0x0
{
  build COND;
go to [RS];
}
```

- Constructor has no name
- "br" mnemonic concatenated with COND code space RS
- Keyword
- Pattern matching
- Pcode

A CONSTRUCTOR HAS NO NAME
Didn’t even talk about…!

- Macros
- Preprocessor
- Context Blocks
- Delay slots
- Prefix code parsing restarts (think x86)
- ...
Write .pspec files for processor specifications

- Any number of pspec files to describe processor variants
- Customize things that don’t require changing Sleigh
  - Sets Program Counter
  - Renames or hides registers
  - Specifies segmented memory
  - Lays down default fixed symbols and memory blocks
  - Sets custom context and properties
- XML with schema - see
  Ghidra/Framework/SoftwareModeling/data/languages/processor_spec.rxg
- Located under data/languages/X.pspec
Example: .pspec

```xml
<processor_spec>
  <properties>
    <property key="addressesDoNotAppearDirectlyInCode" value="true"/>
    <property key="emulateInstructionStateModifierClass"
              value="ghidra.program.emulation.MIPSEmulateInstructionStateModifier"/>
    <property key="assemblyRating:MIPS:BE:32:default" value="PLATINUM"/>
  </properties>
  <programcounter register="pc"/>
  <context_data>
    <context_set space="ram">
      <set name="PAIR_INSTRUCTION_FLAG" val="0" description="1 if LWL/LWR instr. is a pair"/>
      <set name="RELP" val="1" description="1 if mips16e, 0 if micromips"/>
    </context_set>
  </context_data>
  <register_data>
    <register name="contextreg" hidden="true"/>
    <register name="ext_isjal" hidden="true"/>
    <register name="ext_value" hidden="true"/>
  </register_data>
</processor_spec>
```

- Set program property
- Define program counter
- Hide register
Write `.cspec` files for compiler specifications

- One `.cspec` file per significant compiler version
- Sets Stack Pointer
- Specify Compiler structure packing and alignment
- Describes calling conventions used
  - Input/output parameter entries (pentry)
  - Unaffected registers
  - Extra stack space, etc.
- XML with schema - see Ghidra/Framework/SoftwareModeling/data/languages/compiler_spec.rxg
- Located under data/languages/X.cspec
Example: .cspec

```xml
<compiler_spec>
  <data_organization>
    <pointer_size value="4"/>
    <float_size value="4"/>
    <double_size value="8"/>
    <long_double_size value="8"/>
    <size_alignment_map>
      <entry size="1" alignment="1"/>
      <entry size="2" alignment="2"/>
      <entry size="4" alignment="4"/>
      <entry size="8" alignment="8"/>
    </size_alignment_map>
  </data_organization>

  <stackpointer register="sp" space="ram"/>
  <funcptr align="2"/>
  ...
</compiler_spec>
```

Structure packing rules

Set stack pointer
Example: .cspec cont’d

```xml
<default_proto>
  <prototype name="__stdcall" extrapop="0" stackshift="0">
    <input>
      <pentry minsize="1" maxsize="4">
        <register name="a0"/>
      </pentry>
      ...
      <pentry minsize="1" maxsize="500" align="4">
        <addr offset="16" space="stack"/>
      </pentry>
    </input>
    <output>
      <pentry minsize="1" maxsize="4">
        <register name="v0"/>
      </pentry>
      ...
    </output>
  </prototype>
</default_proto>
```

- Define calling convention
- Set register parameter
- Set stack parameters
- Set output register
Create an .ldefs file to tie a “language” all together

- Correlate the Sleigh, pspec, and cspec files together as a unit
- Metadata in the file give Ghidra and the analyst information about the ISA
- XML with schema - see
  Ghidra/Framework/SoftwareModeling/data/languages/language_definitions.rxg
- Located under data/languages/X.ldefs
Example: .ldefs

```xml
<language_definitions>
  <language
    processor="MIPS"
    endian="big"
    size="32"
    variant="default"
    version="1.5"
    slafile="mips32be.sla"
    processorspec="mips32.pspec"
    manualindexfile="../manuals/mipsM16.idx"
    id="MIPS:BE:32:default">
    <description>MIPS32 32-bit addresses, big endian, with mips16e</description>
    <compiler name="default" spec="mips32.cspec" id="default"/>
    <compiler name="Visual Studio" spec="mips32.cspec" id="windows"/>
    <external_name tool="gnu" name="mips:4000"/>
    <external_name tool="IDA-PRO" name="mipsb"/>
    <external_name tool="DWARF.register.mapping.file" name="mips.dwarf"/>
  </language>
</language_definitions>
```
Some extras you’ll probably need

- Custom Analyzers (Java)
  - Performs work required to handle quirks of your processor that aren’t expressed elsewhere
- Function Start/End Patterns (XML)
  - Feed the Function Start Search analyzer
- Opinion Files (XML)
  - Guide the loaders (PE, COFF, ELF, Mach-O) to select the correct architecture for a binary
- Elf Relocation Handler (Java)
  - Handle your ISA relocation types per the ABI
Wrapping up...

- Hope you’ve got some insight into processor modeling now
- Our next talk has a lot more detail about a specific ISA
- Thanks!

- Oh, and...kitteh: