Processors and Assembly Languages

Operating Systems and Security

Viktor Iakovlev and Ksenia Lavrentieva

Stages of Sorce Code Processing

- Text preprocessing
  You can see the output by `gcc -E` option
- Context-Free grammar parsing
  To build an Abstract Syntax Tree (AST)
- Create an object binary code
  Implicitly creates *an assembly code*

Why to Know an Assembly Code?

- Not to code using Assembly
- Not to make preliminary optimizations

No!

- To understand the things Under The Hood
- To understand the Key Concepts of Operating Systems interaction

What is an Assembly Code

- The text to be matched by Regular Expressions, but not Context-Free Grammar
- There is no calculable expressions and program structures. Just lines of instructions

- But the modern compilers allows some syntax sugar like constant-based expressions

The Target

- Each line of code (if not a comment or directive) is an Instruction
- All instructions are processor-specific

- In general, there are two types of instructions:
  - Arithmetic
  - Control

RISC v.s. CISC

- Simple hardware logic: Reduced Instruction Set Compiting (RISC)
  - ARM and AArch64
  - MIPS, PowerPC, SPARC
  - AVR
- Complex hardware logic (in practice implemented as a microcode): Complex Instruction Set Computing (CISC)
  - Intel x86 and AMD64
  - Motorola M68K and successors

RISC v.s. CISC

- RISC:
  - each instruction has a Word Size length
  - instructions are closely related to hardware
  - instructions can operate only to registers
- CISC:
  - each instruction has an arbitary length
  - multiple addressing modes
  - looks like high-level programming

An Example: The **`loop`** in x86

`loop` Address

- Increases the `ecx` register value by 1
- If the value of `ecx` is 0 then `eip`+=sizeof(loop)
- Else `eip`:=Address

The Levels of Abstraction

- The Abstract Syntax Tree and Symbol Tables
- Intermediate Language
  - Internal representation (gcc compilers family)
  - Microsoft Intermediate Language
  - Low-Level Virtual Machine (used by clang)
  - Bytecode to be JIT-compliled: Java, PyPy
- The Binary Code to be executed by Processor

How The Program Structure
can be coded by simple instructions

```
while (true)
 {
    <statement 1>
    . . . .
    <statement N>
 }
```

```
Loop_Start:
    <statement 1>
    . . . .
    <statement N>
    br %Loop_Start
```

How The Program Structure
can be coded by simple instructions

```
while (<cond>)
 {
    <statement 1>
    . . . .
    <statement N>
 }
```

```
Loop_Start:
   %cond = . . .
   br i1 %cond,
     label %Loop_Body,
     label %Loop_End
 Loop_Body:
   <statement 1>
   . . . .
   <statement N>
   br %Loop_Start
 Loop_End:
   . . . .
```

How The Program Structure
can be coded by simple instructions

```
for (<init>;<cond>;<incr>)
{
    <statement 1>
    . . . .
    <statement N>
}
```

```
  <init statements>
Loop_Start:
  %cond = . . .
  br i1 %cond,
    label %Loop_Body,
    label %Loop_End
Loop_Body:
  <statement 1>
  . . . .
  <statement N>
  <increment statement>
  br %Loop_Start
Loop_End:
  <cleanup statements>
  . . . .
```

Intermediate Language

- Has an unlimited “registers” set
- The translator might use the same number of hardware registers as of exists on target processor
- The rest registers are emulated by storing data in memory

What is a Register

- A small amount of memory inside the CPU
- Implemented in hardware using SRAM

- In opposite to cache memory,
  can be directly accessed by instruction

- Cons: the number of available Registers is too restricted

Instructions Encoding

An example from AVR datasheet

ADD – Add without Carry

Description

Adds two registers without the C Flag and places the result in the destination register Rd.

	Operation:		
(i)	(i) Rd ← Rd + Rr		
	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

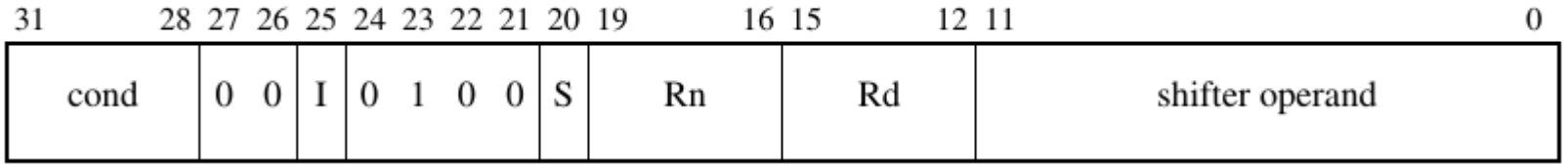
0000	11rd	dddd	rrrr

What to encode

- The Operational Code
- The register(s) used by instruction
- The constant values used by instruction:
  - values to be used in arithmetics
  - address offsets

ARM-32 Instruction Encoding

ADD



- **cond** - the condition the instruction to be processed
- **I** - the bit indicating on **shifter_operand** presence
- **S** - the bit to update Status Register
- **Rn**, **Rd** - resigter numbers for Source and Destination

Examples:

ADD **r0**, **r1**, **r2** // r0 ← r1 + r2
ADD**S** **r0**, **r1**, **r2** // r0 ← r1 + r2, update flags Z,V,C,N
ADD**EQ** **r0**, **r1**, **r2** // If Z is set then r0 ← r1 + r2
ADD **r0**, **r1**, **r2, lsl #3** // r0 ← r1 + (r2 << 3) [5bit shift; 2bit type; 0; 4bit reg]
ADD **r0**, **r1**, **#0xFF** // r0 ← r1 + 0b0000'1111'1111
ADD **r0**, **r1**, **#0x200** // r0 ← r1 + (0b0000'0010 ROR 0b1100)

The Key Problem on Coding

- The instruction size is 32-bit length
- Some of bits are occupied by operation number and register number

There is no space to encode any value:

- constants: not all values can be encoded
- addresses: only the limited range

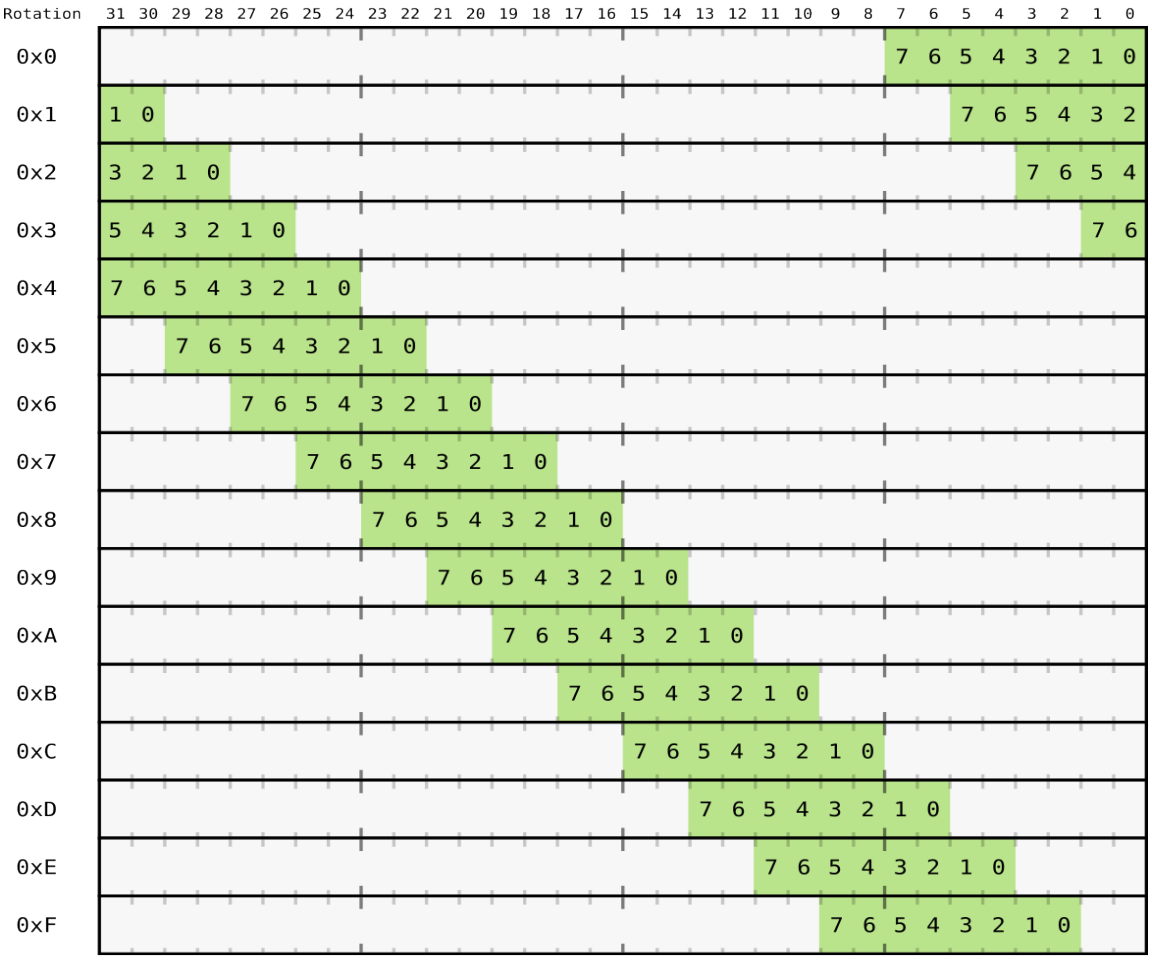
The Constants Problem

Solution for ARM-32

- The length for constant value is 12 bits
- 8 of them is for base value
- The rest 4 bits codes ROT value

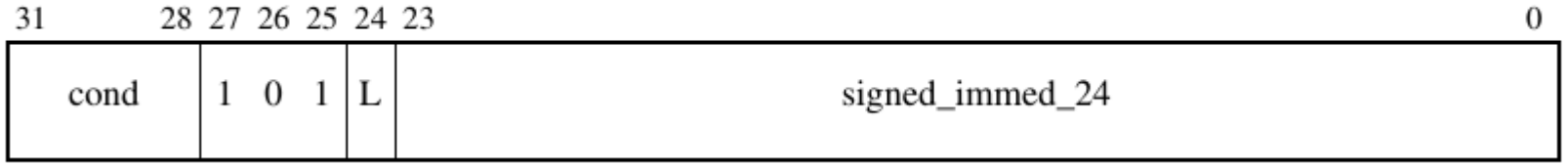
ROtate-and-Translate

```
  255 = 0xFF ROR 0
  256 = 0x01 ROR 24
  512 = 0x02 ROR 24
1024 = 0x03 ROR 24
```



Address Encoding Problem

B, BL



$+/- 2^{23}$ - is a +/- 8MiB

Address Encoding Problem

- Not a Problem while addressing +/- 8MiB function address offset
- Use "trampoline" to access far addresses:
  - the one of problems to be solved by PLT

Procedure Linkage Table

```
function@plt:
    add  ip, pc, #0
    add  ip, ip, #OFFSET_TO_TABLE_BEGIN
    ldr  pc, [ip, #OFFSET_TO_FUNCTION_INDEX]
```

Procedure Linkage Table

- The way to solve long-jump problem
- Allows to place any code at arbitary location to be known at run time