How to Decompile a DSP Architecture

Benedikt (Alkalem)

About Me

- Benedikt (Alkalem)
- Computer science student at KIT
- Member of local CTF team KITCTF
- Working student at aramido

Introduction

```
$ file chall.elf
ELF 32-bit LSB executable, *unknown arch 0xffff9c60* version 1 (SYSV),
             statically linked, stripped
```

- Machine value `0x9c60` not assigned, actually `0x8c` for `EM_TI_C6000`
- Tiny C compiler for TMS320C6x

Introduction

- Disassembly for binary:

```
0000005ac <.text>:
5ac:       003c0058        add .L1 0,a15,a0
5b0:       07bc1058        add .L1X 0,b15,a15
5b4:       023c54f4        stw .D2T1 a4,*b15--(8)
5b8:       023c54f6        stw .D2T2 b4,*b15--(8)
5bc:       033c54f4        stw .D2T1 a6,*b15--(8)
5c0:       07fffc52        addk .S2 -8,b15
5c4:       003c54f4        stw .D2T1 a0,*b15--(8)
5c8:       01bc54f6        stw .D2T2 b3,*b15--(8)
5cc:       00000028        mvk .S1 0,a0
5d0:       00000068        mvkh .S1 0,a0
```

- How to analyze further?

Decompilers

- Recover higher level structures
- Results close to original source code, lossy process
- IDE: rename functions, create types, track changes
- High level analysis independent of platform
- Architecture Plugin: extend using API, support for TMS320C6x family

The original binary

```
$ strings chall.elf | grep "GPNCTF"
GPNCTF{you_really_know_your_instruments_now_PR_it_to_ghidra}
```

Decompiler Plugin Basics

- Exclaimer: Binary Ninja is a commercial product by Vector35, opinions presented are my own
- API in C++, **Python** or Rust (experimental)
- Register every plugin part with Binary Ninja on startup

```python
from binaryninja.architecture import Architecture

class TMS320C6x(Architecture):
    name = 'TMS320C6x'

TMS320C6x.register()
```

Architecture

- Core of our plugin
- General information: name, address size, instruction length, registers etc.
- Core Tasks: control flow, disassembly, lifting
- Disassembler is required for all tasks
- Architecture plugin is intended to process instructions separately

```
class TMS320C6x(Architecture):
    name = 'TMS320C6x'
    address_size = 4     # 32-bit addresses
    max_instr_length = 4 # 32-bit instructions

    regs = {
        'A0': RegisterInfo('A0', 4),
        'A0H' = RegisterInfo('A0', 2, 2), # ...
    }
    stack_pointer = 'B15'
```

Recovering Control Flow

```
def get_instruction_info(self, data: bytes, addr: int) -> Optional[InstructionInfo]
```

- Provide instruction size and branch information
- Branch: type, (known) target, architecture, delay
- Binary Ninja recovers control flow and functions using default algorithm

Disassembly Text

```
def get_instruction_text(self, data: bytes, addr: int)
            -> tuple[list[InstructionTextToken], int]
```

- Used for disassembly view
- Translate disassembly to Binary Ninja's disassembly tokens
- Tokens may improve highlighting, search, and navigation

```
    return ([
        InstructionTextToken(
            InstructionTextTokenType.InstructionToken,
            instr.opcode)
    ], 4)
```

Lifting

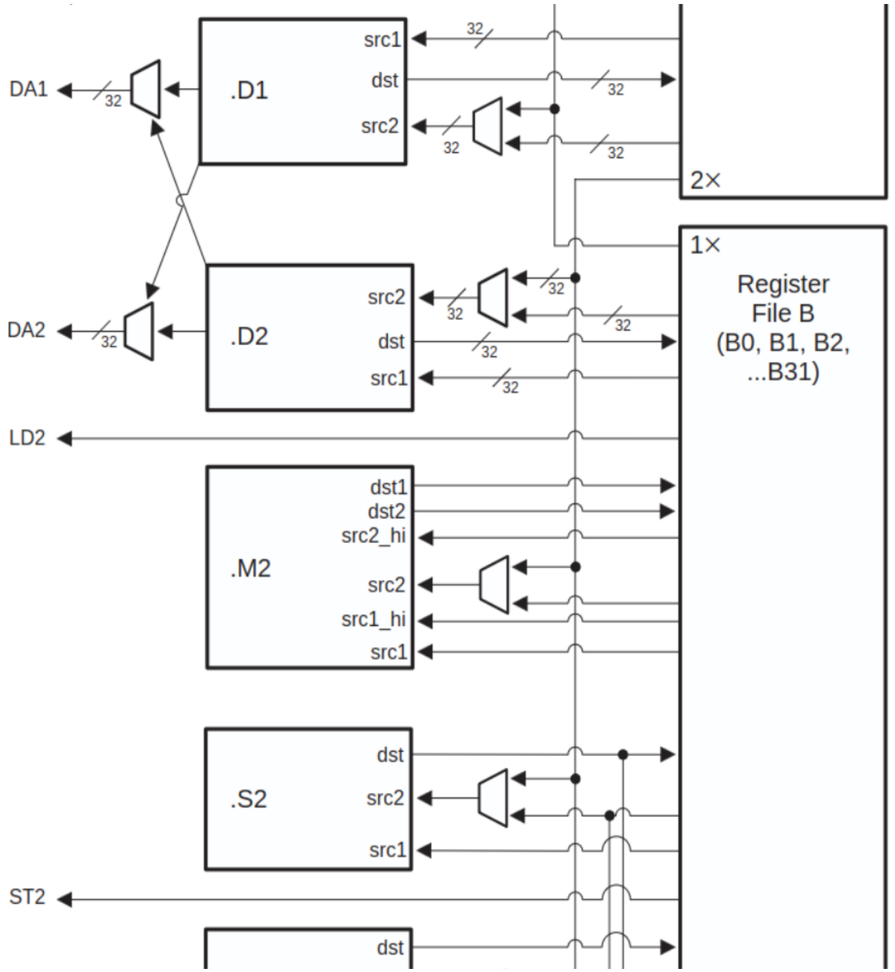
```
def get_instruction_low_level_il(self, data: bytes, addr: int, il: LowLevelILFunction) -> int
```

- Translate meaning of an instruction to intermediate language
- Multiple levels of IL, provide initial level
- Tree-based custom IL language, simple low-level instruction (e.g. set register)

```
    il.append(
        il.set_reg(4, "A0",
            il.add(4, il.const(4, -1), il.reg(4, "A0")))
    )
```

TMS320C6x Architecture

- Architecture Family by Texas Instruments
- Digital Signal Processor (DSP), for audio or radar applications, multiply-and-accumulate
- 32 bit Adresses, Registers and Instructions, Load/Store RISC-Architecture
- 2 register files (A and B), 16 or 32 registers each

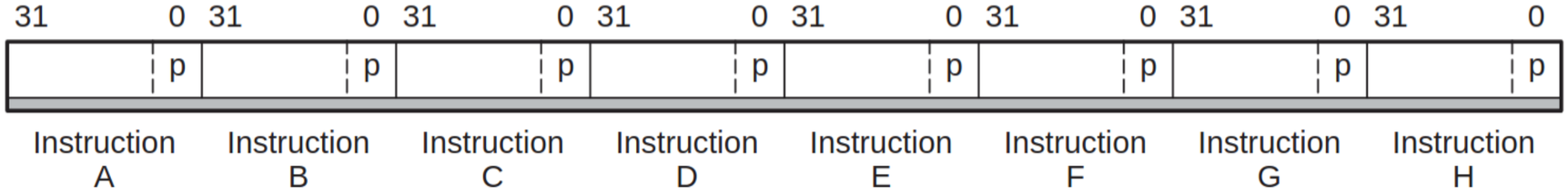


Very Long Instruction Words (VLIW)

- 8 functional units (FU), 2 of each type
- Instruction-Level Parallelism (ILP)
- Fetch Packet (FP): aligned block of 8 instructions fetched, 256 bit

Parallelism

- Instructions in one FP can be parallel, not necessary
- Execution Packet (EP): actual parallel execution
- Visible to instruction set architecture (ISA)




```
        instruction A
||      instruction B
```

Instruction Delay

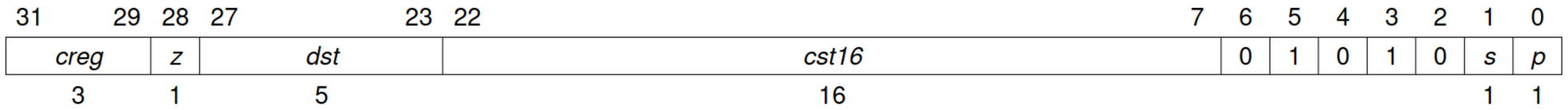
- Result may take multiple cycles to compute or fetch

Instruction type	Delay Slots	Functional Unit Latency	Read Cycles[1]	Write Cycles	Branch Taken
NOP	0	1			
Store	0	1	i	i	
Load	4	1	i	i, i+4 [2]	
Branch	5	1	i [3]		i+5
Single cycle	0	1	i	i	
2-cycle	1	1	i	i+1	
3-cycle	2	1	i	i+2	
4-cycle	3	1	i	i+3	
DP compare	1	2	i, i+1	i+1	
2-cycle DP	1	1	i	i+1	
INTDP	4	1	i	i+3,i+4	
MPYSP2DP	4	2	i	i+3,i+4	
ADDDP/SUBDP	6	2	i, i+1	i+5,i+6	
MPYSPDP	6	3	i, i+1	i+5,i+6	
MPYI	8	4	i, i+1, i+2, i+3	i+8	
MPYID	9	4	i, i+1, i+2, i+3	i+8, i+9	
MPYD	9	4	i, i+1, i+2, i+3	i+8, i+9	

Conditional Execution

- Supported by majority of instructions
- Condition registers B0-B2, A1-A2; A0 in C64x

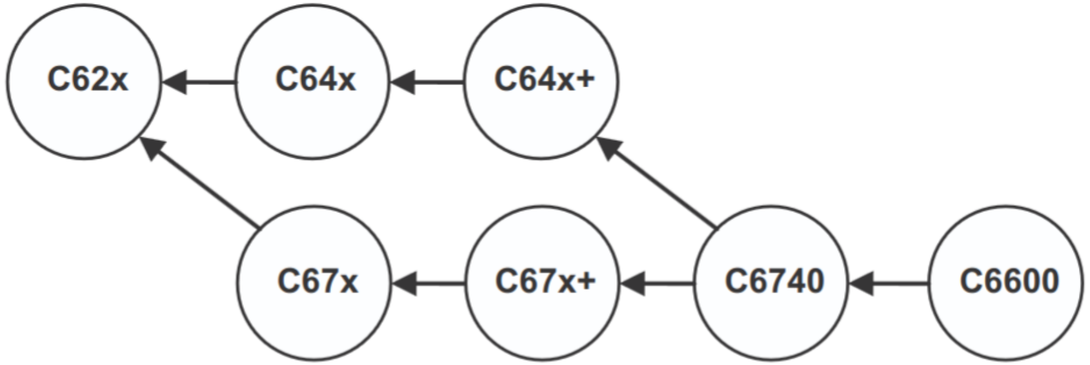
```
|| [!A0] mvkh.S1 0xffff0000, A3
```



Specified Conditional Register		creg			z
	Bit:	31	30	29	28
Unconditional		0	0	0	0
Reserved		0	0	0	1
B0		0	0	1	z
B1		0	1	0	z
B2		0	1	1	z
A1		1	0	0	z
A2		1	0	1	z
A0		1	1	0	z
Reserved		1	1	1	x(1)

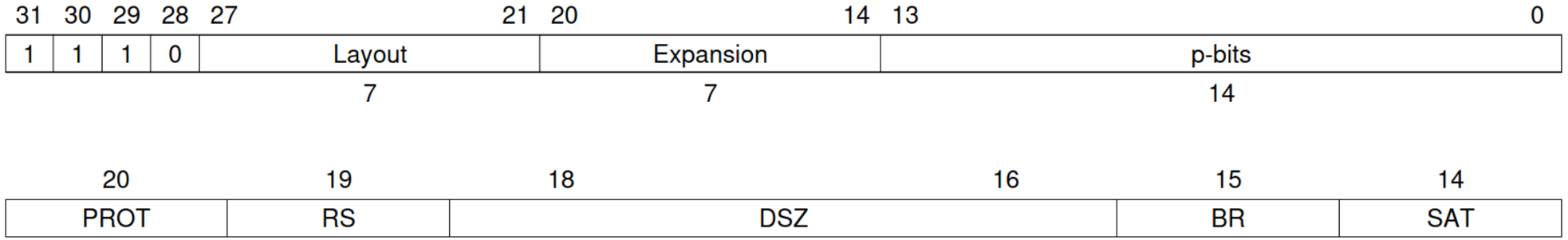
Architecture Family

- C62x: fixed-point DSP
- C64x: 8-bit and 16-bit arithmetic, nonaligned load/store
- C64x+: architectural upgrade (compact instructions, SPLOOPs)
- C67x: floating-point instructions
- C67x+: 64 registers, execute packets can span fetch packets
- C674x: merger of C64x+ and C67x+
- C66x: vector processing (SIMD), floating point improvements



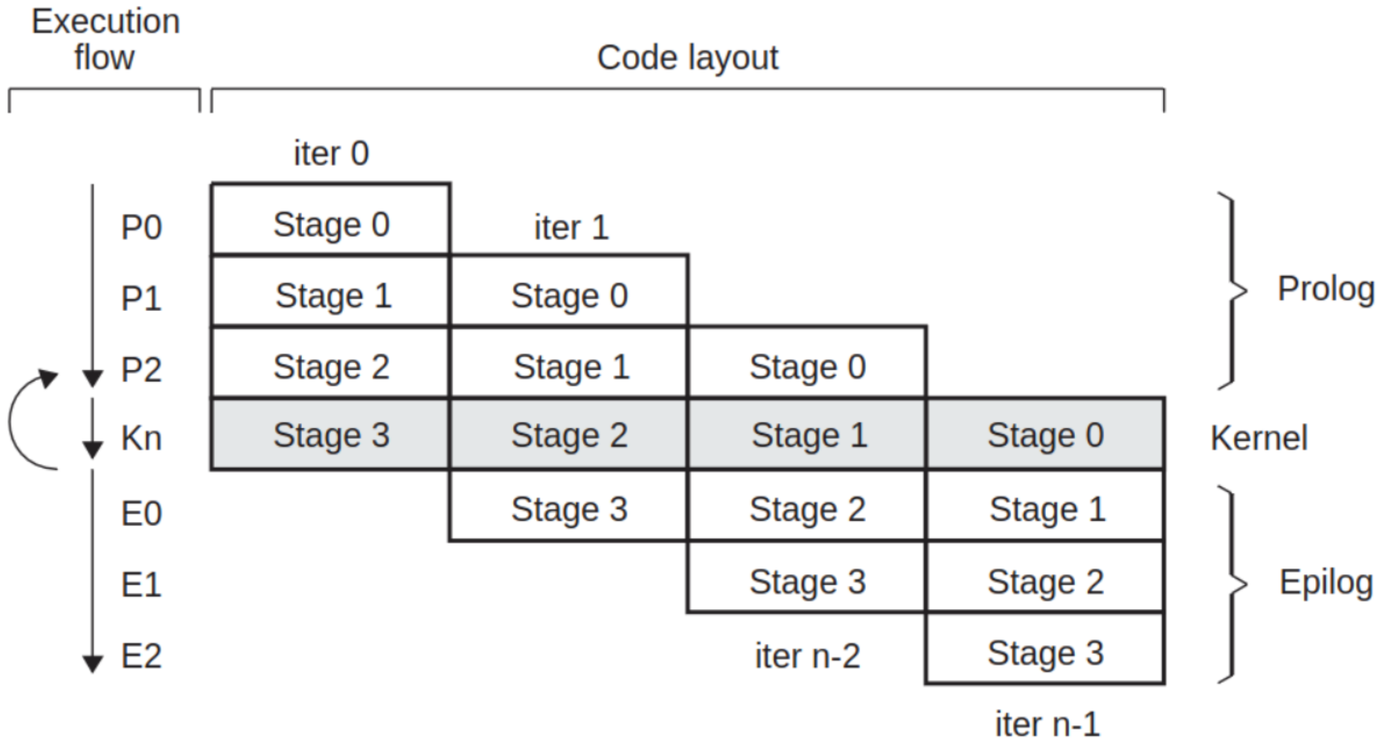
Compact Instructions

- Encoding of frequent operations in 16 bit
- FP header: last instruction in FP, stores packing information



Software Pipelined Loop (SPLOOP)

- Hardware support for modulo loops
- Operations loaded into loop buffer, executed each iteration
- Instructions: SPLOOP(D/W), SPKERNEL(R), SPMASK(R)



Example

Multiply-and-Accumulate Loop:

```
         ldw.D1 *A4++, A5
||       ldw.D2 *B4++, B5
|| [B0]  sub.S2 B0, 1, B0
|| [B0]  b.S1 loop
||       mpy.M1 A5, B5, A6
||       mpyh.M2 A5, B5, B6
||       add.L1 A7, A6, A7
||       add.L2 B7, B6, B7
```

Example

Multiply-and-Accumulate SPLOOP:

```
         mvc.S2      B0, ILC
         nop 3
         sploop 1
         ldw.D1 *A4++, A5
||       ldw.D2 *B4++, B5
         nop 4
||       mpy.M1 A5, B5, A6
||       mpyh.M2 A5, B5, B6
         nop 1
         spkernel 2, 0
||       add.L1 A7, A6, A7
||       add.L2 B7, B6, B7
```

Problems

- Control Flow: cycle branch delay, multiple queued branches
- Disassembly: parallel instructions
- Lifting: parallel or delayed execution, basic block boundaries
- Limited to handling single instructions or at most basic blocks at once

Custom Basic Block Analysis

```
def analyze_basic_blocks(self, func: Function, context: BasicBlockAnalysisContext)
```

- *From Binary Ninja 5.1 (July 2025)*
- Replaces default algorithm for block analysis
- Return blocks and branches between blocks
- Instruction info is not necessary, custom implementation can skip it

Custom Basic Block Analysis

Handling Basic Blocks:

```
block = context.create_basic_block(location.arch, location.addr)
block.add_pending_outgoing_edge(branch_type, location.addr, arch, falltrough)
block.add_instruction_data(instruction_data)
block.can_exit = True # not read-only
block.end = end_address # important!
context.add_basic_block(block)
```

Other important parts:

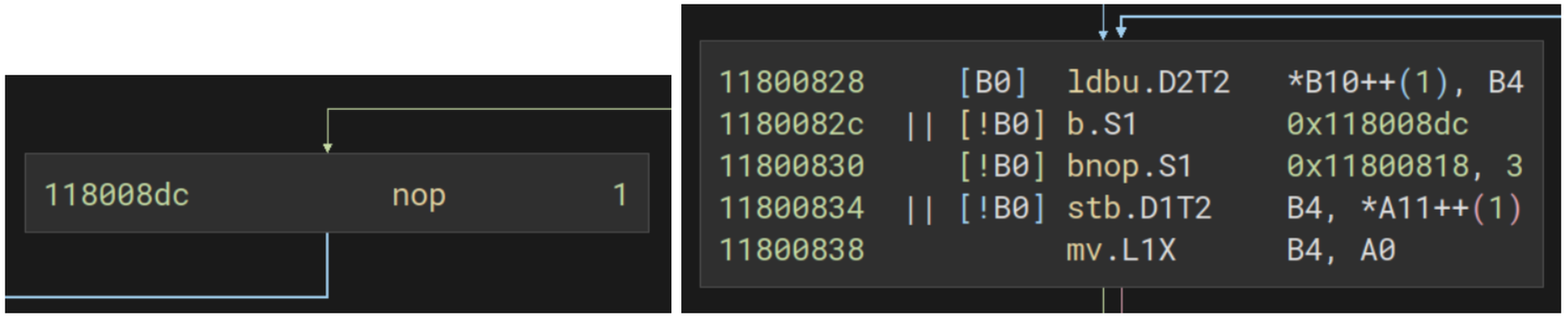
```
view = func.view
view.read(location.addr, arch.max_instr_length)
view.analysis_is_aborted
context.max_function_size
context.finalize()
```

TMS320C6x Basic Block Analysis

Loop for each discovered basic block:

1. Read instructions grouped into cycles (EP)
2. Enqueue delayed branches
3. Apply branches for current cycle
4. Transfer pending branches for target blocks

TMS320C6x Basic Block Analysis Results



Better Instruction Text

- *From Binary Ninja 5.2 (November 2025)*
- NewlineToken to translate multiple disassembly lines

Before:

```
11800828  97362822              [B0]  ldbu.D2T2   *B10++(1), B4; ||
1180082c  90170030              [!B0] b.S1        $C$L5
11800830  2161fc3f              [!B0] bnop.S1     $C$L1, 3; ||
11800834  36362c32              [!B0] stb.D1T2    B4, *A11++(1)
11800838  4612                        mv.L1X      B4, A0
```

After:

```
11800828  97362822                 [B0]  ldbu.D2T2   *B10++(1), B4
1180082c  90170030              || [!B0] b.S1        $C$L5
11800830  2161fc3f                 [!B0] bnop.S1     $C$L1, 3
11800834  36362c32              || [!B0] stb.D1T2    B4, *A11++(1)
11800838  4612                        mv.L1X      B4, A0
```

Better Instruction Text

First attempt:

```
11800828  9736282290170030      [B0]  ldbu.D2T2   *B10++(1), B4
11800828  9736282290170030   || [!B0] b.S1        0x118008dc
11800830  2161fc3f36362c32      [!B0] bnop.S1     $C$L1, 3
11800830  2161fc3f36362c32   || [!B0] stb.D1T2    B4, *A11++(1)
11800838  4612                       mv.L1X       B4, A0
```

```python
class InstructionTextToken:
    def __init__(self, type: InstructionTextTokenType, text: str,
            value: int = 0, size: int = 0, operand: int = 0xffffffff,
            #...
        )
```

Finding Documentation

- Blog: posts include custom architecture tutorial and basic block analysis
- API Reference: stable and dev build
- Developer Documentation: especially the cookbook examples
- Examples from the public API repository
- Slack forum with developers and users
- Or compare with other API (here Rust):

```
    NewLine {
        // Offset into instruction that this new line is associated with
        value: u64,
    },
    // ...
    BNInstructionTextTokenType::NewLineToken => Self::NewLine { value: value.value },
```

Function Context

- *From Binary Ninja 5.3 (April 2026)*
- Context information passed between block analysis and instruction text or lifting
- Instruction text: FP headers and SPLOOP ii in context
- Lifting: information on branches and other instructions spanning basic blocks

```
# prepare context:
def analyze_basic_blocks(self, func: Function,
        context: BasicBlockAnalysisContext) -> None:
    context.function_arch_context = FunctionContext() # any type possible

# later used for:
def get_instruction_text_with_context(self, data: bytes, addr: int, context: Any)
def lift_function(self, func: LowLevelILFunction,
        context: FunctionLifterContext) -> bool:
    function_context = context.function_arch_context
```

Function Context Example

Without function context:

```
11800a54  01800300                sploop      1
11800a58  db1f2402          ||    or.L2X      0, A9, B4
11800a5c  00000000          ||    nop         1

11800a60  e62c453d                <invalid opcode>
11800a64  45570c02                stdw.D1T1   A5:A4, *A3++(0x10)
11800a68  faf0c403          ||    sub.L2X     B7, A17, B7
11800a6c  00400308                spkernel    0x20
```

With function context:

```
11800a54  01800300                sploop      1
11800a58  db1f2402          ||    or.L2X      0, A9, B4
11800a5c  00000000          ||    nop         1

11800a60  e62c                    spmask      L2
11800a62  453d              ||    stdw.D2T2   B5:B4, *B6++(0x10)
11800a64  45570c02          ||    stdw.D1T1   A5:A4, *A3++(0x10)
11800a68  faf0c403          ||    sub.L2X     B7, A17, B7
11800a6c  00400308                spkernel    1, 0
```

Function Lifting

- *From Binary Ninja 5.3 (April 2026)*
- Replace default lifting algorithm for functions
- Function context and binary view available during lifting
- Lift each basic block, possibly with shared context

```
for block in context.blocks:
    function.set_current_source_block(block)
    context.prepare_block_translation(function, arch, block.start)
    addr = block.start
    while addr < block.end:
        opcode = block.get_instruction_data(addr)
        stream = arch.disasm.disasm(opcode, addr)
        lifted_bytes = lift_basic_block(stream, shared_context)
        addr += lifted_bytes
```

Lifting TMS320C6x

- DSP executes parallel, decompiler expects serial LLIL
- Temporary registers store values for later use
- Translate instructions grouped by cycle (EP):
  1. Split instructions in parts and enqueue
  2. Read condition registers and inputs
  3. Lift opcodes completing in current cycle
  4. Write results back
  5. Optionally trigger branches

Accuracy of Specification

- Disassembly includes access type and cycles for operands
- Lifting depends on correctness to enqueue parts correctly

Documentation for instruction ADDK:

Execution

```
if (cond) cst16 + dst → dst
else nop
```

Pipeline

Pipeline Stage	E1
Read	*cst16*
Written	*dst*
Unit in use	.S

Lifting Across Basic Blocks

- Example: delayed branch, calculates target directly, branches later
- Pass values through blocks in temporary registers

```
144 @ 118008b0  temp101.d = memset
145 @ 118008c0  goto 149
```

```
270 @ 118008a4  B4 = temp8.d
271 @ 11800894  temp101.d = memset
272 @ 11800884  jump($C$L4 => 149 @ $C$L4)
```

```
149 @ 118008c0  B3 = $C$RL0
150 @ 118008c4  temp8.d = A11
151 @ 118008c8  temp9.d = A12
152 @ 118008c4  temp4.d = temp8.d
153 @ 118008c8  temp8.d = temp9.d
154 @ 118008c4  A4 = temp4.d
155 @ 118008c8  A6 = temp8.d
156 @ 11800894  call(temp101.d)
```

```
int32_t A11_1;
int A12;
char* B10_3;
entry_A3 = memset(outbuf_1, fill, len);
```

Other Topics

- Adding calling convention to recognize function parameters
- Undo loop unrolling, lifting SPLOOPs
- Simplify IL in analysis workflows
- Practical and advanced examples

Our example binary:

```
11800e40    int main() __pure

11800e40    {
11800e40        return 0;
11800e40    }
```

Time for Questions

Email: benedikt.waibel@student.kit.edu

Fediverse: @alkalem@infosec.exchange

Plugin: https://github.com/Alkalem/binja-tms320c6x

Slides: https://kitctf.de/learning/gpn24-decompile-dsp

Thank you for listening!