Reverse Engineering DSP Code

Pierre Bourdon

delroth@lse.epita.fr
http://lse.epita.fr

February 12, 2013

Context

- Core developer of the Dolphin Emulator (GameCube/Wii)
- Recently working mainly on sound processing emulation
- Had to understand how it worked and reverse engineer the code running on it to reimplement it

What is a DSP?

- Digital Signal Processor
- Highly specialized CPUs with several ways to make signal processing fast
- Applications: sound mixing, sound effects processing, signal demodulation, etc.

Sound processing

- Mixing sounds together: $s = a + b$
- Setting a volume: $s = v \times i$ $(0 <= v <= 1)$
- You can only mix together sounds at the same sample rate, so resampling might be needed (linear, cubic, FIR)
- Sound delaying to simulate precise 3D positioning
- Filters: LPF, FIR, etc.

Tricks

- Unless you need to cover a large range of values, floating point numbers are bad compared to fixed point numbers
- Sound samples are in $[-1.0, 1.0]$
- Volume is in $[0.0, 1.0]$
- Each sound sample can be represented as a 16 bit number in $[-32768, 32767]$
- Volume can be represented as a value in $[0, 32767]$
- Big optimization: ALU computations are a lot faster than FPU
- Need to be careful with overflows in intermediate computations

Communication with other components

- The DSP also needs to communicate with several external components: CPU, RAM, hardware decoder, ...
- Often has interrupts and in/out ports support to get events from the CPU
- Data from RAM is fetched and written using DMA to an internal, smaller RAM

Specs

- Custom Macronix DSP design
- Runs at 81MHz (fast!)
- Hardware 32 bit multiplier with overflow handling
- 4K IRAM, 4K DRAM
- 4K IROM, 8K DROM
- DMA access to the GameCube RAM and ARAM
- Hardware PCM8, PCM16 and ADPCM decoding from ARAM

Registers

- 4 Address Registers: $AR0, $AR1, $AR2, $AR3
- 4 Index Registers: $IX0, $IX1, $IX2, $IX3
- 4 Wrapping Registers: $WR0, $WR1, $WR2, $WR3
- 2 32 bit "general" registers: $AX0, $AX1
- 2 40 bit accumulators: $ACC0, $ACC1
- 1 40 bit multiplication result register: $PROD

Subregisters

- $AX0.H, $AX0.L (16 bit)
- $ACC0.H (8 bit), $ACC0.M, $ACC0.L (16 bit)
- Access to $ACC0.H can be either zero-extended or sign-extended

More peculiarities

- 16 bit bytes: addresses index 16 bit values
- Instructions can be either 16 or 32 bits long (usually with a 16 bit immediate)
- Some instructions can be merged with an "extended operation" to perform 2 operations at once
- Strange control flow instructions using an internal loop register stack: LOOP, BLOOP, IFC, ...

```
CLR $ACC0              // ACC0 = 0;
LOOP $ACC1.M           // while (ACC1.M--)
SRRI @$AR0, $ACC0.M //      *AR0++ = ACC0.M;
```

Extended operations

- Explicit parallelization of some operations that can be performed at the same time
- For example, "load from memory" and "multiply two numbers"
- Used a lot to make loops faster: load and store data at the same time you perform operations
- More than memory access: moving data from a register to another, adding an index register to an address register, etc.
- Uses parts of the CPU not used by the main instruction

Example

```
opcode: bcf0
disasm:
    MULAX'LD $AX0.H, $AX1.H, $ACC0 : $AX0.H, $AX1.H, @$AR0

pseudocode:
    ACC0 += PROD;
    PROD = AX0.H * AX1.H;
    AX0.H = *AR0++;
    AX1.H = *AR3++;
```

Example 2

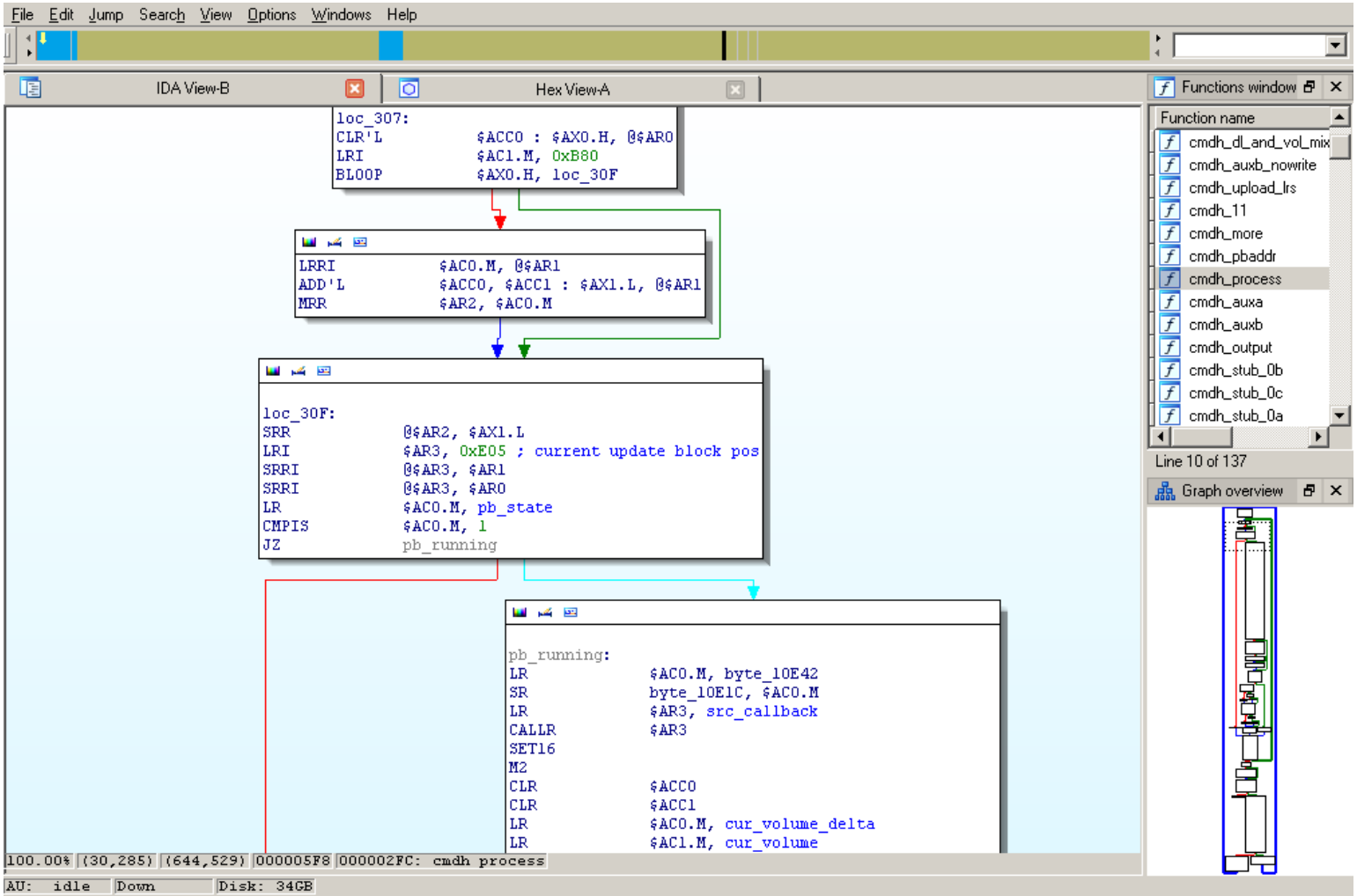
```
opcode: f2e7
disasm:
    MADD'LDN $AX0.L, $AX0.H : $AX0.H, $AX1.L, @$AR3

pseudocode:
    $PROD += AX0.L * AX0.H;
    AX0.H = *AR0++;
    AX1.H = *AR3;
    AR3 += IX3;
```

Tools

- Only one disassembler available, no real static analysis tool
- Wrote an IDA plugin for the GCN DSP in November 2011
- IDA handles surprisingly well most of the strange features of this DSP (including 16 bit bytes)
- Made it a lot easier to do cross-references, renaming symbols, etc.
- Writing IDA plugins will make you hate it, but it's worth the trouble in the end

IDA


File Edit Jump Search View Options Windows Help
IDA View-B
Hex View-A
loc_307:
CLR'L $ACC0 : $AX0.H, @$AR0
LRI $AC1.M, 0xB80
BLOOP $AX0.H, loc_30F
LRRI $AC0.M, @$AR1
ADD'L $ACC0, $ACC1 : $AX1.L, @$AR1
MRR $AR2, $AC0.M
loc_30F:
SRR @$AR2, $AX1.L
LRI $AR3, 0xE05 ; current update block pos
SRRI @$AR3, $AR1
SRRI @$AR3, $AR0
LR $AC0.M, pb_state
CMPIS $AC0.M, 1
JZ pb_running
pb_running:
LR $AC0.M, byte_10E42
SR byte_10E1C, $AC0.M
LR $AR3, src_callback
CALLR $AR3
SET16
M2
CLR $ACC0
CLR $ACC1
LR $AC0.M, cur_volume_delta
LR $AC1.M, cur_volume
Functions window
Function name
cmdh_dl_and_vol_mix
cmdh_auxb_nowrite
cmdh_upload_lrs
cmdh_11
cmdh_more
cmdh_pbaddr
cmdh_process
cmdh_auxa
cmdh_auxb
cmdh_output
cmdh_stub_0b
cmdh_stub_0c
cmdh_stub_0a
Line 10 of 137
Graph overview
100.00% (30,285) (644,529) 000005F8 000002FC: cmdh_process
AU: idle Down Disk: 34GB

Confusing idioms

- All of the code is written directly in assembly, without respect for any kind of calling convention
- Branching has an impact on speed, so loops are sometimes manually unrolled
- Wrapping registers used to implement circular buffers
- Automatic multiply by 2 for volume mixing

Pipelining

Used to increase the throughput of a loop by taking advantage of the explicit parallelization.

```
LRRI            $AX0.H, @$AR3
LRRI            $AX0.L, @$AR3
MULX            $AX0.L, $AX1.L
MULXMV          $AX0.H, $AX1.L, $ACC0
BLOOPI          0x30, 0x0655
    ASR16'L         $ACC0 : $AC1.M, @$AR1
    ADDP'LN         $ACC0 : $AC1.L, @$AR1
    LRRI            $AX0.H, @$AR3
    ADD'L           $ACC1, $ACC0 : $AX0.L, @$AR3
    MULX'S          $AX0.L, $AX1.L : @$AR1, $AC1.M
    MULXMV'S        $AX0.H, $AX1.L, $ACC0 : @$AR1, $AC1.L
```

Questions?

- @delroth_
- http://dolphin-emu.org/