Arbitrary and Variable Precision Floating-Point Arithmetic Support in Dynamic Binary Translation

Marie Badaroux & Frédéric Pétrot
Univ. Grenoble Alpes, CNRS,
Grenoble INP, TIMA 🏠 tima.imag.fr/sls
✉️ marie.badaroux@univ-grenoble-alpes.fr
Who

Marie Badaroux, PhD Student
Engineering degree in CS, Grenoble INP/Ensimag, 2020,
Master of Science in CS, Univ. Grenoble Alpes, 2020

Frédéric Pétrot, MSC in EECS, 1990
PhD in CS, 1994
Université Paris VI
Prof. at Grenoble INP/Ensimag, 2004
Deputy director of TIMA Lab
Introduction

The big picture
- Fast processor centric digital system simulators
  - Instruction Set Architecture design choices evaluation
  - Playground for compiler backends for ISA extensions

The goal
- Arbitrary precision floating-point arithmetic support in DBT
  - Demonstrate feasibility
  - Propose a design and an implementation

Why?
- Limits of IEEE Standard

Marie Badaroux & Frédéric Pétrot (UGA-TIMA)

ASP-DAC 2021

January 20, 2021
Introduction

The big picture
- Fast processor centric digital system simulators
  - Instruction Set Architecture design choices evaluation
  - Playground for compiler backends for ISA extensions

The goal
- Arbitrary precision floating-point arithmetic support in DBT
  - Demonstrate feasibility
  - Propose a design and an implementation
Introduction

The big picture
- Fast processor centric digital system simulators
  - Instruction Set Architecture design choices evaluation
  - Playground for compiler backends for ISA extensions

The goal
- Arbitrary precision floating-point arithmetic support in DBT
  - Demonstrate feasibility
  - Propose a design and an implementation

Why?
- Limits of IEEE 754 Standard
Introduction

The big picture
- Fast processor centric digital system simulators
  - Instruction Set Architecture design choices evaluation
  - Playground for compiler backends for ISA extensions

The goal
- Arbitrary precision floating-point arithmetic support in DBT
  - Demonstrate feasibility
  - Propose a design and an implementation

Why?
- Limits of IEEE 754 Standard
Introduction

The big picture
- Fast processor centric digital system simulators
  - Instruction Set Architecture design choices evaluation
  - Playground for compiler backends for ISA extensions

The goal
- Arbitrary precision floating-point arithmetic support in DBT
  - Demonstrate feasibility
  - Propose a design and an implementation

Why?
- Limits of IEEE 754 Standard

How?
- Propose an ISA extension
- Propose a simulation solution for dynamic binary translation
- Technical choices
  - ISA: **RISC-V**, DBT engine: **QEMU**, AP library: **MPFR**
Table of Contents

1. Background: Floating-point numbers & DBT mechanism

2. State-of-the-art of floating-point numbers emulation

3. Arbitrary precision support in DBT

4. Experiments

5. Conclusion
# Table of Contents

1. **Background: Floating-point numbers & DBT mechanism**

2. State-of-the-art of floating-point numbers emulation

3. Arbitrary precision support in DBT

4. Experiments

5. Conclusion
Floating-point numbers
IEEE 754 Standard

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 bit</td>
<td>e bits</td>
<td>m bits</td>
</tr>
</tbody>
</table>

\[ \pi \approx 3.1415927410125732421875 \]

0 10000000 10010010000111111011011
**DBT mechanism**

**QEMU emulation process**

```
Yes
PC already seen?

No
Fetch

Decode

Branch?

No

Execute

Translation Cache
(host binary code)

Yes

Tiny code generator

Code Generation

Micro-operations built-in

Instruction

Target binary code (.elf)

Binary Translation

micro-ops buffer
```

Marie Badaroux & Frédéric Pétrot (UGA-TIMA)

ASP-DAC 2021

January 20, 2021
DBT mechanism

**QEMU emulation process**

- **PC already seen?**
  - Yes → Target binary code (.elf)
  - No → Fetch

- **Binary Translation**
  - Micro-operations built-in
  - Tiny code generator

- **Decode**
  - No → Branch?
  - Yes → Execute

- **Target to Host code generation example**

  18  instrX_target
DBT mechanism

QEMU emulation process

Target to Host code generation example

18 instrX_target micro-op1_instrX

micro-op2_instrX
DBT mechanism

QEMU emulation process

Target to Host code generation example

18 instrX_target micro-op1_instrX host_instr1_micro-op1_instrX
    host_instr2_micro-op1_instrX
    host_instr3_micro-op1_instrX
    micro-op2_instrX host_instr1_micro-op2_instrX
DBT mechanism

QEMU emulation process

Target to Host code generation example

18 instrX_target micro-op1_instrX host_instr1_micro-op1_instrX
host_instr2_micro-op1_instrX
host_instr3_micro-op1_instrX
micro-op2_instrX host_instr1_micro-op2_instrX
DBT mechanism

Target code \(\rightarrow\) TCG Intermediate Representation code \(\rightarrow\) Host code

1 IN: [Target Insn] OUT: [size=68] [Host Insns]
2 0x000800001ac: add t1,t1,a3 0x7fb464008700: movl -0x10(%rbp), %ebx 0x7fb464008703: testl %ebx, %ebx
3 OP: 0x7fb464008705: jl 0x7fb464008738 0x7fb46400870b: movq 0x30(%rbp), %rbx
4 ... 0x7fb464008713: addq %r12, %rbx
5 ... 0x7fb46400871a: nop
6 ... 0x7fb46400871b: jmp 0x7fb464008720
7 ---- 000000080001ac [µOps] 0x7fb464008720: movl $0x800001b0, %ebx 0x7fb464008725: movq %r12, 0x600(%rbp)
8 mov_i64 tmp2,t1 0x7fb46400872c: leaq -0xf3(%rip), %rax
9 mov_i64 tmp3,a3 0x7fb464008733: jmp 0x7fb464000018 0x7fb46400873f: jmp 0x7fb464000018
10 add_i64 tmp2,tmp2,tmp3 0x7fb464008738: leaq -0xfc(%rip), %rax
11 mov_i64 t1,tmp2 0x7fb464008740: movq %rbx, 0x600(%rbp)
12 goto_tb $0x0 0x7fb464008743: jmp 0x7fb464000018
13 movi_i64 pc,$0x800001b0 0x7fb464008746: movq %rbx, 0x600(%rbp)
14 exit_tb $0x7fb464008640 0x7fb464008749: movl $0x800001b0, %ebx
15 set_label $L0 0x7fb464008750: movq %r12, 0x600(%rbp)
16 exit_tb $0x7fb464008643

Figure: Translation process of a single add instruction
Table of Contents

1. Background: Floating-point numbers & DBT mechanism

2. State-of-the-art of floating-point numbers emulation

3. Arbitrary precision support in DBT

4. Experiments

5. Conclusion
State-of-the-art

Reminder

Our goal is to simulate a functional AP floating-point arithmetic

ISA floating-point zoo

- Classical IEEE 754, F/D extensions (Berkeley’11)
- Half Precision, Transprecision, SmallFloat (VDAT’19, SBAC-PAD’19, DATE’19)
- Type I, II, III Unum coprocessors (CF’19, VLSI’19, ICCD’18)
State-of-the-art

Reminder
Our goal is to simulate a functional AP floating-point arithmetic

ISA floating-point zoo
- Classical IEEE 754, F/D extensions (Berkeley’11)
- Half Precision, Transprecision, SmallFloat (VDAT’19, SBAC-PAD’19, DATE’19)
- Type I, II, III Unum coprocessors (CF’19, VLSI’19, ICCD’18)

Emulating instructions
- Softfloat libraries (MASCOTS’10, Berkeley’18)
- Partial use of host hardware (RAPIDO’16, VEE’19)
# Table of Contents

1. Background: Floating-point numbers & DBT mechanism

2. State-of-the-art of floating-point numbers emulation

3. Arbitrary precision support in DBT

4. Experiments

5. Conclusion
Arbitrary Precision support in DBT

Our Extended Arbitrary Precision ISA Proposal:

- 32 AP registers of virtually unlimited size
- 2 CSRs of 64-bit holding exponent size and precision
- 31 instructions largely inspired from F/D extensions

### Table: Arbitrary Precision Instruction Encodings

<table>
<thead>
<tr>
<th>imm[11:0]</th>
<th>31</th>
<th>27</th>
<th>26</th>
<th>25</th>
<th>24</th>
<th>20</th>
<th>19</th>
<th>15</th>
<th>14</th>
<th>12</th>
<th>11</th>
<th>7</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs3</td>
<td>10</td>
<td>rs2</td>
<td>rs2</td>
<td>rs2</td>
<td>rs1</td>
<td>rm</td>
<td>rd</td>
<td>0000111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rs2: Source Register 2
- rs1: Source Register 1
- rm: Register Mask
- rd: Destination Register
- FLP: Floating Point Load
- FSP: Floating Point Store
- FMADD.P: Floating Point Multiply Add
- FADD.P: Floating Point Add
- FMSUB.P: Floating Point Multiply Subtract
- FNMSUB.P: Floating Point Normalized Multiply Subtract
- FMUL.P: Floating Point Multiply
- FDIV.P: Floating Point Divide
- FSQRT.P: Floating Point Square Root
- FSGNJ.P: Floating Point Select Greater
- FSGNJN.P: Floating Point Select Greater
- FSGNJX.P: Floating Point Select Greater
- FMIN.P: Floating Point Min
- FMAX.P: Floating Point Max
- FEQ.P: Floating Point Equal
- FLT.P: Floating Point Less Than
- FLE.P: Floating Point Less Than or Equal
- FCVT.x.y: Floating Point Convert
Arbitrary Precision support in DBT

Our registers:

Types:

typedef struct {
  mpfr_prec_t _mpfr_prec;
  mpfr_sign_t _mpfr_sign;
  mpfr_exp_t _mpfr_exp;
  mp_limb_t *_mpfr_d;
} __mpfr_struct;

typedef __mpfr_struct mpfr_t[1];

Fields:

struct CPURISCVState {
  target_ulong gpr[32];
  uint64_t fpr[32];
  /* vector coprocessor state. */
  uint64_t vreg[32 * RV_VLEN_MAX / 64];
  /* arbitrary precision registers */
  mpfr_t apr[32];
  ...
  target_ulong pc;
  ...
  /* arbitrary precision mantissa size csr */
  target_ulong fprec;
  /* arbitrary precision exponent size csr */
  target_ulong fexp;
  ...
};
Principle:
Translate all AP instructions as *helpers*
Arbitrary Precision support in DBT

Target code $\rightarrow$ TCG Intermediate Representation code $\rightarrow$ Host code
Arbitrary Precision support in DBT

Decodetree
- at QEMU compile time: understands instructions through patterns
- at QEMU runtime: calls translator function for matched pattern

Example (IR generated when matching \texttt{fadd.p})

```c
static bool trans_fadd_p(DisasContext *ctx, arg_fadd_p *a)
{
    TCGv dest = tcg_temp_new();
    ...
    TCGv rm   = tcg_temp_new();

    tcg_gen_movi_tl(dest, a->rd);
    ...
    tcg_gen_movi_tl(rm, a->rm);

    gen_helper_fadd_p(cpu_env, dest, src1, src2, rm);

    tcg_temp_free(dest);
    ...
    tcg_temp_free(rm);
    return true;
}
```
Arbitrary Precision support in DBT

Target code $\Rightarrow$ TCG Intermediate Representation code $\Rightarrow$ Host code
Arbitrary Precision support in DBT

Helpers

Example (Function being called within a TB containing \texttt{fadd.p})

```c
void helper_fadd_p(CPURISCVState *env,
                 target_ulong dest, target_ulong src1, target_ulong src2,
                 target_ulong rm)
{
    if (mpfr_get_prec(env->apr[dest]) != env->fprec) {
        MPFR_DECL_INIT(x, env->fprec);
        mpfr_add(x, env->apr[src1], env->apr[src2],
                  rm == 7 ? env->frm : rm);
        mpfr_set_prec(env->apr[dest], env->fprec);
        mpfr_set(env->apr[dest], x, rm == 7 ? env->frm : rm);
    } else {
        mpfr_add(env->apr[dest], env->apr[src1], env->apr[src2],
                 rm == 7 ? env->frm : rm);
    }
}
```
Table of Contents

1. Background: Floating-point numbers & DBT mechanism
2. State-of-the-art of floating-point numbers emulation
3. Arbitrary precision support in DBT
4. Experiments
5. Conclusion
Experiments

Benchmark:

- **unit-tests**
  
  Test all AP instructions with random values

- **e**
  
  Compute $e$ using Taylor expansion: $e = \sum_{k=0}^{n} \frac{1}{k!}$. 

- **cholesky**
  
  Compute Cholesky decomposition of lower triangular matrices.
Experiments

Figure: x86-64 and QEMU execution times for e
Figure: Execution times of e for double and AP
Experiments

Figure: Execution times of e with dynamic AP changes
Experiments

cholesky

Figure: Execution times of cholesky on $n \times n$ matrices
Table of Contents

1. Background: Floating-point numbers & DBT mechanism
2. State-of-the-art of floating-point numbers emulation
3. Arbitrary precision support in DBT
4. Experiments
5. Conclusion
Conclusion

- Operational support of arbitrary precision floating-point arithmetic in dynamic binary translation
  - Demonstrated feasibility
  - Achieved Design, implementation and evaluation
- Some points are left to be addressed
  - Compiler does not target our AP extension
  - Use other libraries than MPFR
  - Generalize to other architectures
- MPFR data structure proposal for "small" precision
  Mantissa external pointer $\rightarrow$ Flexible array
THANK YOU

https://gricad-gitlab.univ-grenoble-alpes.fr/tima/sls/projects/qemu-vp