SPEAR Modular Arithmetic Format Specification Version 1.0

Domagoj Babić
babic <at> cs.ubc.ca

Date: 2007/12/21 17:48:12 Revision: 1.7

1 Legal Matters

2 Introduction

This document is a specification of a simple quantifier-free modular arithmetic input format. The format currently supports bit-vectors up to 64 bits, and all standard bit-vector operators. As the format specified in this document is a native input format of SPEAR automated theorem prover, it will be called the SPEAR format (SF).

The benchmarks in this format can be bit-blasted to boolean satisfiability instances (SPEAR can do that conversion for you). However, it is convenient to have a higher-level modular arithmetic format to avoid duplication of encoding effort and to facilitate interpretation of results. In addition, the higher level structure available in modular arithmetic constraints can be exploited for more efficient solving.

SPEAR format is designed to be:

- Compact — Benchmarks can get very large, even hundreds of megabytes, so every byte counts. The format strives to achieve maximal compactness without going to binary, or sacrificing readability.

- Trivial to parse — In my humble opinion, one of the main reasons why the boolean satisfiability (SAT) competition[1] has been so successful is an exceptionally low-effort entry into SAT solving. The input format is so simple that anyone can get a simple SAT solver up and running in a day. The format presented in this document strives for the same level of simplicity.

- Expressive — Any modular arithmetic format worth its while needs to support multiplication, unsigned and signed division, as well as other frequently used operators.

- Simple to generate — Each constraint can be seen as a small circuit with a named output. The constraints are all in the format

  $$output\ operator\ operand_1\ operand_2\ \cdots\ operand_n$$

  while the predicates are in the format

  $$operator\ operand_1 \cdots operand_n$$

  Chances are that your internal representation of the problem you want to encode to modular arithmetic is either a set of expressions that are already in this form, or a circuit-like graph, from which such constraints can be trivially generated.

- Precise — Precise semantics is crucial in operator specification. Theorem provers are useless if each produces a different answer.

- Readable — Facilitates debugging of the theorem provers.

- Suitable for random benchmark generation — Decision procedures are very complex pieces of software that require very rigorous testing (buggy decision procedures are useless for formal verification). Format that enables simple random test-case generation makes it possible to design simple automatic test environments, which, in my experience, are very effective in finding bugs in decision procedures.

- Simple output format — Facilitates building of test environments, competition environments (like SAT competition scripts), as well as the usage of the results. Variables in this format are required to have unique names, which means that it is trivial to map solutions to variables. In contrast, the formats that allow context-dependent name overloading complicate the usage of the results.

The limitation to 64 bits is for practical reasons. If only 64-bit types are allowed, all constants are representable in machine integers, and all arithmetic can be done in 128 bits, which means that theorem provers do not need big-number libraries. This is important for several reasons:

- Efficiency — Big num libraries are often much slower than regular machine arithmetic.

- Portability — Fewer library dependencies.

[1] http://www.satcompetition.org/

- Simplicity — Simpler design of the theorem prover.

The SF file ending is `.sf`. According to my web search, that seems to be a rarely used ending. The file itself is a regular ASCII text file. In this document, the content of an The SF file will be always in the `verbatim` style.

This format specifies the queries as satisfiability queries. The query is equal to a conjunction of all constraints and predicates in the input file. To prove validity, invert your query and check for unsatisfiability.

3 Types

The basic type is boolean, denoted as `i1` (one-bit integral type). All other types are composed of booleans. For instance, 64-bit vector is `i64`.

Declarations of variables and constants are immediately coupled with their types by using a colon as a separator. For instance:
(`variable:type` or `constant:type`).

There is no distinction between signed and unsigned bit-vectors. All arithmetic is two's complement arithmetic, and the operators that actually depend on the sign, like division, will have both signed and unsigned versions.

4 Constants

Constants are represented in the form `value:type`, and can be used directly as operands. `value` is an unsigned integer. Since all the arithmetic is in two's complement, the constants can be read as unsigned 64-bit integers and then truncated to the bit-width of the given type. Unsigned values that do not fit in the given type are allowed, but will be truncated.

5 Variables

The result of every operation has to be assigned to a variable. Every circuit can be easily transformed into this form. Variable names must start with a lower- or uppercase letter, and can be followed by a zero or more letters, digits, or underscores. Other characters are disallowed. Every variable has to be declared on the declaration line in the header before used.

Variables must have unique names for several reasons:

- Simpler and faster parsing.
- If the instance is satisfiable, unique names simplify mapping of the solutions to the subexpressions of the original problem. Without unique names, deciphering the solutions would be unnecessarily complicated.
- If the instance is converted into the CNF form, a simple map file can be easily generated that maps modular arithmetic variables to boolean variables.

Operator names (like *extr*, *zext*, etc.) cannot be used as variable names.

6 High-Level Structure

This section specifies the high-level structure of the SF files. The file has to begin with a header, and continue with constraints and predicates.

Each header must contain a version line, which shows the version of the format specification according to which the file was generated. The version line starts with `v`, followed by a white space (one or more spaces or tabs), and ends with two integers separated by a dot. For instance:

```
v 1.2
```

In addition to the version line, the header may contain:

- Comment line(s) — Anything starting with `#` is considered a comment.
- Expect line — Expected result, `0` for unsatisfiable, and `1` for satisfiable. Starts with `e`, followed by a white space. Expect line can be used for adjusting the heuristics to the expectations.
- Declaration line(s) — Declares variables used in the instance. Starts with `d`, followed by a white space.

For instance, a header could look like:

```
# Header example
v 1.0
e 0
d a:i64 b:i64 res:i64
```

Constraints are given in the Polish notation in the form:
`output operator operand1 operand2 ... operandn`,
one constraint per line. Intuitively, `output` can be seen as the name of the wire that the circuit represented by the `operator` drives. Constraint lines have to start with `c`, followed by a white space. The rest of the line contains a variable representing the output (or a constant), the operator, and the operands in the natural order (left-to-right). The order is fixed and each two adjacent tokens on the line must be separated by a white space. In the following example, the result of addition of `a` and `b` is stored into `res`:

```
c res + a b
```

The following constraint requires the sum of `a` and `b` to be zero:

```
c 0:i32 + a b
```

Unlike in real circuits, you can provide multiple definitions of the same variable. For instance:

```
d a:i8 b:i8
c a + 1:i8 b
c a - 1:i8 b
```

This instance happens to be satisfiable (having two solutions: `a=0,b=1` and `a=129,b=128`, binary: 00000001+100000000 = 10000001, 00000001−10000000 = 00000001 + 01111111 + 000000001 = 10000001), but in general, multiple definitions of a variable can easily cause inconsistency (instance becomes unsatisfiable).

Finally, the predicate lines start with `p`, followed by a white space. The predicate line contains a single predicate (e.g. `predicate op1 op2 ... opn`). Predicates are considered to be:

- All operators presented in Sec. 7
- Operators *and*, *xor*, *or*, *if-then-else*, and *not* if and only if all their operands are of boolean type.

The following example illustrates a possible use of predicate lines. The validity of $a+b = b+a$ can be done by checking that $a+b \neq b+a$ is unsatisfiable:

```
# Commutativity of addition
v 1.0
e 0
d a:i64 b:i64 c:i64 d:i64
c c + a b
c d + b a
p /= c d
```

Predicates produce a result of the boolean type. If the types are properly matched, predicates can be also used as constraints. For instance, these are all valid usages of predicates:

```
d res:i1
p /= a b
p /= a 384:i64
c 1:i1 /= a b
c res /= 24:i64 b
```

Main points:

- Every line starts with a letter followed by a white space, except for the comments. Comments start with `#`, followed by anything.
- Order of lines must be: the header, and then constraints and predicates.
- Variables can have multiple definitions, but must have unique names.

7 Operators

This section specifies all supported operators. Operators that handle signed and unsigned bit-vectors differently will be distinguished by the operator symbol.

7.1 Bitwise Operators

Operators presented in this section take one or two operands. If the operator takes two operands, both operands must be of the same type. The result is always of the same type as the operands.

Def. 1 (&) *Bitwise AND. Operands: 2.*

Def. 2 (|) *Bitwise OR. Operands: 2.*

Def. 3 (ˆ) *Bitwise XOR. Operands: 2.*

Def. 4 (∼) *Bitwise negation (NOT). Operands: 1.*

There is no distinction between logical and bitwise operators. Bitwise AND is logical AND when its operators are of the boolean type.

7.2 Predicates

Bitwise operators in Sec. 7.1 are considered predicates if their operands are of the boolean type. If the operands are booleans, so is the result.

The operators presented in this section always produce boolean results and always require two operands. Both operands must be of the same type.

Most of the predicates in this section will be defined in the terms of flags set up by subtraction of the operands of the predicate. Here we define those flags:

- (Z)ero flag — true if and only if all bits of the subtraction result are zero.
- (C)arry flag — The carry flag is true if and only if subtraction produces a carry out of the most significant bit (also known as borrow).
- (N)egative flag — true if and only if the most significant bit of the subtraction result is true.
- (O)verflow flag — Overflow flag is defined as XOR of the carry bits from the two most significant bits produced by subtraction.

Def. 5 (=>) *Implication.* $a \Rightarrow b$ *is a shorthand for* $\neg a \vee b$.

Def. 6 (=) *Equal. Returns* true *(`1:i1`) if operands are equal, or* false *(`0:i1`) otherwise. I.e.,* $a = b$ *iff* $a - b$ *sets* Z *flag.*

Def. 7 (/ =) *Not equal. Returns* true *if at least one bit of the first operand does not match the corresponding bit of the second operand,* false *otherwise. I.e.,* $a/ = b$ *iff* $a - b$ *does not set* Z *flag.*

Def. 8 (*ule*) *Unsigned less or equal.* a *ule* b *is* true *iff* $a - b$ *sets flags so that* $\neg C \vee Z$ *is* true.

Def. 9 (*uge*) *Unsigned greater or equal.* a *uge* b *is* true *iff* $a - b$ *sets flags so that* C *is* true.

Def. 10 (*ult*) *Unsigned less than.* a *ult* b *is* true *iff* $a - b$ *sets flags so that* $\neg C$ *is* true.

Def. 11 (*ugt*) *Unsigned greater than. a ugt b is* true *iff* $a - b$ *sets flags so that* $C \wedge \neg Z$ *is* true.

Def. 12 (*sle*) *Signed less or equal. a sle b is* true *iff* $a - b$ *sets flags so that* $Z \vee (N \hat{} V)$ *is* true, *where* $\hat{}$ *is XOR.*

Def. 13 (*sge*) *Signed greater or equal. a sge b is* true *iff* $a - b$ *sets flags so that* $N = V$ *is* true, *where* $=$ *is equality.*

Def. 14 (*slt*) *Signed less than. a slt b is* true *iff* $a - b$ *sets flags so that* $N \hat{} V$ *is* true.

Def. 15 (*sgt*) *Signed greater than. a sgt b is* true *iff* $a - b$ *sets flags so that* $\neg Z \wedge (N = V)$ *is* true.

7.3 If-then-else operator

Def. 16 (*ite*) *If-then-else. Operands: 3. If the first operand is* true, *returns the second operand, otherwise returns the third operand. First operand must be of type* `i1`. *The other two operands must be of an equal type. The returned result is of the type of the last two operands.*

7.4 Arithmetic Operators

Operators in this section take two operands. Both operands must be of the same type. The result is of the same type as the operands.

For precise definition of division and remainder in this section we need to introduce *ceiling* and *floor* operators. These operators are not supported by the SF, and serve only as helper definitions:

- Ceiling — $\lceil a \rceil$, rounds a real number a to the nearest integer by rounding towards $+\infty$ if a is positive, or towards $-\infty$ if a is negative.
- Floor — $\lfloor a \rfloor$, rounds a real number a to the nearest integer by rounding towards zero.

In addition to ceiling and floor, we also need to introduce a special helper operator for division of reals. Real division operator will be denoted as $\div$. If operands of $\div$ are integers, it will be assumed that they are casted to reals before the division is performed.

Def. 17 ($+$) *Standard two's complement addition.*

Def. 18 ($-$) *Standard two's complement subtraction,* $a - b = a + (\sim b) + 1$.

Note that `-` cannot be used as two's complement. To compute a two's complement of a variable `a`, use `0 - a`.

Def. 19 ($*$) *Standard two's complement multiplication.*

Def. 20 ($/u$) *Unsigned division. a /u b is defined as* $\lfloor a \div b \rfloor$ *if* $b \neq 0$. *For* $b = 0$, *the result is undefined.*

sign of a	sign of b	sign of q
-	-	+
-	+	-
+	-	-
+	+	+

Table 1: Sign table for signed division.

Def. 21 ($/s$) *Signed division. The operator is formally defined as:*

$$a\ /s\ b = \begin{cases} \lfloor a \div b \rfloor & \text{if} \quad b \neq 0 \wedge a * b \geq 0 \\ \lceil a \div b \rceil & \text{if} \quad b \neq 0 \wedge a * b < 0 \end{cases}$$

The most significant bit of the quotient q is given in Table 1. For $b = 0$, the result is undefined.

Def. 22 ($\%u$) *Unsigned remainder. $a\ \%u\ b$ is defined as $a - (a\ /u\ b) * b$. For $b = 0$, the result is undefined.*

Def. 23 ($\%s$) *Signed remainder. $a\ \%s\ b$ is defined as $a - (a\ /s\ b) * b$. The most significant bit (sign) of the result is equal to the most significant bit of a. For $b = 0$, the result is undefined.*

If division (resp. remainder) by zero happens while the instance is being solved, the result is undefined. This is the only allowed source of possible mismatches of the satisfiability results produced by different theorem provers.

Since all operators require either variables or constants as operands, it is easy to detect runtime division or remainder by zero. Automatic solution checkers should disregard such solutions (and not flag them as incorrect).

7.5 Shift Operators

Operators in this section take two operands. The first operand can be of an arbitrary integral type, while the second must be of `i8` type. The return type is always of the same type as the first operand.

Def. 24 ($<<$) *Left shift. If the second operand is larger than the bit-width of the first operand, the result is zero.*

Def. 25 ($>> a$) *Arithmetic shift right. The most significant bit of the first operand is shifted in from the left. If the second operand is larger than the bit-width of the first operand, the result is zero if the most significant bit of the first operand is zero, or a bit-vector of ones otherwise.*

Def. 26 ($>> l$) *Logical shift right. Zero is shifted in from the left. If the second operand is larger than the bit-width of the first operand, the result is zero.*

7.6 Cast Operators

This section specifies supported casting operators.

Def. 27 (*trun*) *Truncation. Operands: 1. Operand is truncated to the bit-width of the result, keeping the least significant bits. The bit-width of the operand must be strictly larger than the bit-width of the result. For instance:*

```
d a:i32 b:i1 c:i2
c b trun a
c c trun 43:i32
```

The first constraint says that `b` *is equal to the least significant bit of* `a`*, while the second says that* `c` *is equal to* `3:i2`*.*

Def. 28 (*sext*) *Sign extend. Operands: 1. Sign extends the operand to the bit-width of the result. The bit-width of the operand must be strictly smaller than the bit-width of the result.*

Def. 29 (*zext*) *Zero extend. Operands: 1. Zero extends the operand to the bit-width of the result. The bit-width of the operand must be strictly smaller than the bit-width of the result.*

Def. 30 (*conc*) *Concatenate. Operands: 2. Given two operands of types* `ix` *and* `iy`*, returns result of type* `i(x+y)`*, such that the bits of the second operand are copied into the least significant part of the result, and the bits of the first operand are put into the remaining places.*

For instance, concatenation of `1:i8` *and* `0:i8` *would produce* `256:i16`*. Similarly,* 0110 *concatenated with* 1001 *produces* 01101001*.*

Def. 31 (*extr*) *Extract. Operands: 3. First operand is a variable (resp. constant) as specified in Sec. 5 (resp. 4). The other two operands are constants 0–64 of type* `i8`*. Zero represents the least significant bit. The operator extracts bits in range from the second operand (inclusive) to the third operand (exclusive). The second operand must be always strictly smaller than the third operand.*

If the first operand is x, the second n, and the third m, the operator extracts bits $[n–m)$ from x. The result is of type `i(m-n)`*.*

For instance, `extr 22:i8 0:i8 3:i8` *returns* `6:i3`*, while* `extr 22:i8 3:i8 4:i8` *returns* `0:i1`*.*

8 Output Format

The expected output is very similar to the SAT competition output format.

Every output that a theorem prover produces should start either with `c`, `s`, or `v`, followed by a white space. The letter `c` stands for a comment, while the letter `s` stands for solution. Solution must be: `SATISFIABLE`, `UNSATISFIABLE`, or `UNKNOWN`. The last one is used if the theorem prover detected internal inconsistency, unexpected behaviour, or timed out. The satisfying assignment must be printed on a line starting with `v` in the form `variable=value`, where the value is an unsigned integer. Solutions on the `v`-line have to be separated by one or more white spaces. Two examples of the valid output:

```
c Speedy Gonzo Theorem Prover, v 0.0001
c
c <get your coffee, this is going to take some time...>
```

```
c
c TIMEOUT
s UNKNOWN

c Speedy Gonzo Theorem Prover, v 0.0001
c
c
s SATISFIABLE
v a=1 b=2 c=3
```

Given a variable of type `ik`, the solution must fit into `k` bits.

9 Notes on Efficiency

This section provides several suggestions how to make your instances more compact and easier for the theorem provers to prove:

- Simplify expressions — Simplify expressions as much as possible before encoding them into the SF. I suggest using the following simple heuristic: Try to minimize the total number of bits required for representing all the variables.
- Slice your query — Before you pass the query to a theorem prover, remove all redundant expressions. This can make a drastic difference in both the size of the instance and the time required to solve it.
- Experiment with the encoding order — Changing the order of your constraints in the instance can make 10–20X difference in the runtimes. Try encoding your internal representation both forward and backward, and see what works better for you.
- Propagate constants — Theorem provers are usually extremely efficient at propagating constants, so constant propagation will not have much impact on the solving runtimes, but will make the instances smaller. This technique is especially effective if you have a large number of relatively simple queries. Although the speedup on each individual query is not likely to be large, when you have thousands of queries, the savings become much more significant.
- Encode constants immediately as operands — Write `a + 6:i8 b` instead of `= c 6:i8, a + c b`. This will improve performance a bit, at least if you use Spear.

Strength reduction is another possibility to consider[2]. If you use Spear, that will not make much difference (except for the noise caused by the impact of this change on the heuristics), because Spear performs strength reduction internally. Other theorem provers might not have this feature, so keep this in mind as a possibility for improving performance.

[2]For instance, replacing `a*8` with `a<<3`.

10 Examples

Example 1 (Checking a simple assertion) *Assume that you want to generate an instance that corresponds to the following C-like sequential code:*

```
int f(int a, int b) {
    int a1;
    if (a%2) { a1 = a + 1; }
    else { a1 = a; }
    int c = a1 * a1;
    assert(c % 4 == 0);
}
```

You could check the validity of the assertion by checking the unsatisfiability of the negated formula (corresponds to checking that the assertion can never be false*):*

```
# Checking a simple assertion
v 1.0
d a:i32 a1:i32 c:i32 inca:i32 tmp1:i32 tmp2:i1 tmp3:i32 assert:i1
c inca + a 1:i32
c tmp1 %s a 2:i32
c tmp2 trun tmp1
c a1 ite tmp2 inca a
c c * a1 a1
c tmp3 %s c 4:i32
c assert = tmp3 0:i32
p = assert 0:i1
```

SPEAR *proves this query to be unsatisfiable in 0.02 sec on AMD 64 X2 4600+ for 32-bit integers, and in 0.08 sec for 64-bit integers with the default heuristics.*

Example 2 (Last Fermat's theorem) *The last Fermat's theorem says that for integer $n > 2$ the equation $a^n + b^n = c^n$ has no solutions for non-zero integers a, b, and c.*

However, if $a, b, c \in \mathbb{Z}_m$[3], the equation can have non-trivial (non-zero) solutions. Let's try to find one such example:

```
# Finding a solution of a^4 + b^4 = c^4
v 1.0
d a:i64 b:i64 c:i64 a2:i64 b2:i64 c2:i64 a4:i64 b4:i64 c4:i64 sum:i64
c a2 * a a
c b2 * b b
c c2 * c c
c a4 * a2 a2
c b4 * b2 b2
c c4 * c2 c2
c sum + a4 b4
p = sum c4
```

[3]Ring of integers modulo m.

SPEAR *finds a non-trivial solution of this query in 24.08 sec on AMD 64 X2 4600+ with the default heuristics and in 3.18 sec with the* `fh_1_1` *heuristic*[4].

Example 3 (Checking the equivalence of two circuits) *In hardware verification, it is frequently required to prove that some circuit transformation preserves the properties of the circuit. For instance, HDL synthesis might do various optimizations, and it is important that it preserves the properties of the original design. The process of checking the equivalence of two representations is called equivalence checking.*

In this example we shall check that a trivially simple shift-and-add synthetized multiplier circuit actually performs single-precision multiplication. The output of the synthetized circuit is denoted as `SYNTH`*, while the output of the golden model, to which we are comparing our synthetized circuit, is denoted as* `GOLD`*. There should exist no input vectors for which the results differ. Hence, we simply check that* `/= SYNTH GOLD` *is unsatisfiable:*

```
v 1.0
# Basic inputs and outputs
d A:i4 B:i4 GOLD:i4 SYNTH:i4

# Bits of B
d b0:i1 b1:i1 b2:i1 b3:i1

# Sign extended bits of B
d seb0:i8 seb1:i8 seb2:i8 seb3:i8

# Zero extended A
d zea:i8

# Partial products
d pp0:i8 pp1:i8 pp2:i8 pp3:i8

# Shifted partial products
d spp1:i8 spp2:i8 spp3:i8

# Partial sums
d ps1:i8 ps2:i8 ps3:i8

# ---------- Constraints ----------

# Golden model of a multiplier, we
# check whether the below synthetized
# multiplier gives equivalent result
# to the GOLD model.
c GOLD * A B

# ---- Synthetized multiplier ----
```

[4]Run SPEAR `--help` for more info on available parameter sets, and SPEAR `--hidden` to see all adjustable parameters

```
# Compute gating vectors by sign
# extending individual bits of B
c b0 extr B 0:i8 1:i8
c b1 extr B 1:i8 2:i8
c b2 extr B 2:i8 3:i8
c b3 extr B 3:i8 4:i8

# Sign extend bits of B to
# get gating vectors.
c seb0 sext b0
c seb1 sext b1
c seb2 sext b2
c seb3 sext b3

# Zero extend A
c zea zext A

# Compute partial products
c pp0 & zea seb0
c pp1 & zea seb1
c pp2 & zea seb2
c pp3 & zea seb3

# Shift partial products
c spp1 << pp1 1:i8
c spp2 << pp2 2:i8
c spp3 << pp3 3:i8

# Compute partial sums
c ps1 + pp0 spp1
c ps2 + ps1 spp2
c ps3 + ps2 spp3

# The lower half of ps3 is the
# result of single-precision
# multiplication performed by the
# above synthetized circuit.
c SYNTH trun ps3

# There should exist no vectors for
# which the results differ (i.e.
# this instance should be UNSAT).
p /= SYNTH GOLD
```

SPEAR *proves this query to be UNSAT in 0.00 sec on AMD 64 X2 4600+ with the default heuristics. Hence, the simple multiplier synthetized above always computes the same result as* SPEAR *multiplication (golden model).*