May 19, 2018 at 02:31

1*. Intro. Given m and n, where $n \ge m \ge 2$, together with a nonnegative integer $z < 2^{m+n}$, this program generates clauses that are satisfiable if and only if z can be factored into an m-bit integer times an n-bit integer.

It uses Luigi's Dadda's scheme [*Alta Frequenza* **34** (1964), 349–356], choosing bits to add in a first-in-first-out manner. Change files will readily adapt this algorithm to other queuing disciplines.

The integers being multiplied are denoted by $(x_m \ldots x_1)_2$ and $(y_n \ldots y_1)_2$, and the product is $(z_{m+n} \ldots z_1)_2$. Intermediate variables of weight 2^k are named `A`k`.`l, `P`k`.`l, `Q`k`.`l, `S`k`.`l. The `A` variables are input bits, while `P`, `Q`, and `S` are intermediate results in the calcalation of a full adder for (a, a', a''):

$$s \leftarrow a \oplus a', \quad p \leftarrow a \wedge a', \quad r \leftarrow s \oplus a'', \quad q \leftarrow s \wedge a'', \quad c \leftarrow p \vee q.$$

(Here r goes into the current bin, and becomes `A` or `Z`; c is a carry that becomes an `A` in the next bin.)

This variant of the program actually omits z. It forms two copies of the multiplier circuitry, and states that they produce different results. (These clauses are clearly unsatisfiable, but the solver will have to figure that out.)

```
#define nmax 1000
#include <stdio.h>
#include <stdlib.h>
  int bin[nmax + nmax][nmax];   /* what items l are in bin k? */
  int count[nmax + nmax];   /* how many items have we ever put in bin k? */
  int size[nmax + nmax];   /* how many items currently in bin k? */
  int adders[nmax + nmax];   /* how many full adders have we used in bin k? */
  int m, n;   /* the given parameters */
  int addend[3];   /* three inputs to a full adder */
  main(int argc, char *argv[])
  {
    register int i, j, k, l;
    ⟨Process the command line 2*⟩;
    printf("~␣sat-dadda-miter␣%d␣%d\n", m, n);
    ⟨Generate the difference clauses for z 3*⟩;
    ⟨Generate the main clauses 4⟩;
  }
```

2*. ⟨Process the command line 2*⟩ ≡

```
  if (argc ≠ 3 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &n) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣m␣n\n", argv[0]);
    exit(−1);
  }
  if (n > nmax) {
    fprintf(stderr, "Sorry,␣n␣must␣be␣at␣most␣%d!\n", nmax);
    exit(−2);
  }
  if (m < 2 ∨ m > n) {
    fprintf(stderr, "Sorry,␣m␣can't␣be␣%d␣(it␣should␣lie␣between␣2␣and␣%d)!\n", m, n);
    exit(−3);
  }
```

This code is used in section 1*.

3* Variable `@`k will be true only if there's a solution with `z`$k = 0$ and `Z`$k = 1$.

⟨ Generate the difference clauses for z 3* ⟩ ≡

```
for (j = 0; j < m + n; j++) {
  printf("~@%d␣~z%d\n", j + 1, j + 1);
  printf("~@%d␣Z%d\n", j + 1, j + 1);
}
for (j = 0; j < m + n; j++) printf("␣@%d", j + 1);
printf("\n");
```

This code is used in section 1*.

4. ⟨ Generate the main clauses 4 ⟩ ≡

```
⟨ Generate the original one-bit products 6* ⟩;
for (k = 3; k ≤ m + n; k++) ⟨ Generate the clauses for bin k 5 ⟩;
```

This code is used in section 1*.

5. ⟨ Generate the clauses for bin k 5 ⟩ ≡

```
{
  while (size[k] > 2) ⟨ Do a full add 8* ⟩;
  if (size[k] > 1) ⟨ Do a half add 7* ⟩;
}
```

This code is used in section 4.

6* **#define** *make_and*(a, ka, la, b, kb, lb, c, kc, lc)

```
{
  if (ka) printf("~%c%d.%d␣", a, ka, la);
  else printf("~%c%d␣", a, la);
  if (kb) printf("%c%d.%d\n", b, kb, lb);
  else printf("%c%d\n", b, lb);
  if (ka) printf("~%c%d.%d␣", a, ka, la);
  else printf("~%c%d␣", a, la);
  if (kc) printf("%c%d.%d\n", c, kc, lc);
  else printf("%c%d\n", c, lc);
  if (ka) printf("%c%d.%d␣", a, ka, la);
  else printf("%c%d␣", a, la);
  if (kb) printf("~%c%d.%d␣", b, kb, lb);
  else printf("~%c%d␣", b, lb);
  if (kc) printf("~%c%d.%d\n", c, kc, lc);
  else printf("~%c%d\n", c, lc);
}
```

⟨ Generate the original one-bit products 6* ⟩ ≡

```
for (i = 1; i ≤ m; i++)
  for (j = 1; j ≤ n; j++) {
    k = i + j;
    if (k ≡ 2) {
      make_and('z', 0, 1, 'X', 0, i, 'Y', 0, j);
      make_and('Z', 0, 1, 'X', 0, i, 'Y', 0, j);
    } else {
      l = count[k] = ++size[k];
      bin[k][l − 1] = l;
      make_and('a', k, l, 'X', 0, i, 'Y', 0, j);
      make_and('A', k, l, 'X', 0, i, 'Y', 0, j);
    }
  }
```

This code is used in section 4.

7* **#define** *make_xor*(a, ka, la, b, kb, lb, c, kc, lc)

```
{
  if (ka) printf("%c%d.%d␣", a, ka, la);
  else printf("%c%d␣", a, la);
  if (kb) printf("~%c%d.%d␣", b, kb, lb);
  else printf("~%c%d␣", b, lb);
  if (kc) printf("%c%d.%d\n", c, kc, lc);
  else printf("%c%d\n", c, lc);
  if (ka) printf("%c%d.%d␣", a, ka, la);
  else printf("%c%d␣", a, la);
  if (kb) printf("%c%d.%d␣", b, kb, lb);
  else printf("%c%d␣", b, lb);
  if (kc) printf("~%c%d.%d\n", c, kc, lc);
  else printf("~%c%d\n", c, lc);
  if (ka) printf("~%c%d.%d␣", a, ka, la);
  else printf("~%c%d␣", a, la);
  if (kb) printf("%c%d.%d␣", b, kb, lb);
  else printf("%c%d␣", b, lb);
  if (kc) printf("%c%d.%d\n", c, kc, lc);
  else printf("%c%d\n", c, lc);
  if (ka) printf("~%c%d.%d␣", a, ka, la);
  else printf("~%c%d␣", a, la);
  if (kb) printf("~%c%d.%d␣", b, kb, lb);
  else printf("~%c%d␣", b, lb);
  if (kc) printf("~%c%d.%d\n", c, kc, lc);
  else printf("~%c%d\n", c, lc);
}
```

⟨ Do a half add 7* ⟩ ≡

```
{
  make_xor('z', 0, k - 1, 'a', k, bin[k][0], 'a', k, bin[k][1]);
  make_xor('Z', 0, k - 1, 'A', k, bin[k][0], 'A', k, bin[k][1]);
  if (k ≡ m + n) {
    make_and('z', 0, k, 'a', k, bin[k][0], 'a', k, bin[k][1]);
    make_and('Z', 0, k, 'A', k, bin[k][0], 'A', k, bin[k][1]);
  } else {
    l = count[k + 1] = ++size[k + 1], bin[k + 1][l - 1] = l;
    make_and('a', k + 1, l, 'a', k, bin[k][0], 'a', k, bin[k][1]);
    make_and('A', k + 1, l, 'A', k, bin[k][0], 'A', k, bin[k][1]);
  }
}
```

This code is used in section 5.

8* **#define** *make_or*(*a*, *ka*, *la*, *b*, *kb*, *lb*, *c*, *kc*, *lc*)

```
{
  if (ka) printf("%c%d.%d␣", a, ka, la);
  else printf("%c%d␣", a, la);
  if (kb) printf("~%c%d.%d\n", b, kb, lb);
  else printf("~%c%d\n", b, lb);
  if (ka) printf("%c%d.%d␣", a, ka, la);
  else printf("%c%d␣", a, la);
  if (kc) printf("~%c%d.%d\n", c, kc, lc);
  else printf("~%c%d\n", c, lc);
  if (ka) printf("~%c%d.%d␣", a, ka, la);
  else printf("~%c%d␣", a, la);
  if (kb) printf("%c%d.%d␣", b, kb, lb);
  else printf("%c%d␣", b, lb);
  if (kc) printf("%c%d.%d\n", c, kc, lc);
  else printf("%c%d\n", c, lc);
}
```

⟨ Do a full add 8* ⟩ ≡

```
{
  for (i = 0; i < 3; i++) ⟨ Choose addend[i] 9 ⟩;
  i = ++adders[k];
  make_xor('s', k, i, 'a', k, addend[0], 'a', k, addend[1]);
  make_xor('S', k, i, 'A', k, addend[0], 'A', k, addend[1]);
  make_and('p', k, i, 'a', k, addend[0], 'a', k, addend[1]);
  make_and('P', k, i, 'A', k, addend[0], 'A', k, addend[1]);
  l = ++count[k], bin[k][size[k]++] = l;
  if (size[k] ≡ 1) {
    make_xor('z', 0, k − 1, 's', k, i, 'a', k, addend[2])
    make_xor('Z', 0, k − 1, 'S', k, i, 'A', k, addend[2])
  } else {
    make_xor('a', k, l, 's', k, i, 'a', k, addend[2]);
    make_xor('A', k, l, 'S', k, i, 'A', k, addend[2]);
  }
  make_and('q', k, i, 's', k, i, 'a', k, addend[2]);
  make_and('Q', k, i, 'S', k, i, 'A', k, addend[2]);
  if (k ≡ m + n) {
    make_or('z', 0, k, 'p', k, i, 'q', k, i);
    make_or('Z', 0, k, 'P', k, i, 'Q', k, i);
  } else {
    l = count[k + 1] = ++size[k + 1], bin[k + 1][l − 1] = l;
    make_or('a', k + 1, l, 'p', k, i, 'q', k, i);
    make_or('A', k + 1, l, 'P', k, i, 'Q', k, i);
  }
}
```

This code is used in section 5.

9. Finally, here's where I use the first-in-first-out queuing discipline. (Clumsily.)

$\langle$ Choose *addend*[*i*] 9 $\rangle \equiv$

```
{
  addend[i] = bin[k][0];
  for (l = 1; l < size[k]; l++) bin[k][l - 1] = bin[k][l];
  size[k] = l - 1;
}
```

This code is used in section 8*.

10.* Index.

The following sections were changed by the change file: 1, 2, 3, 6, 7, 8, 10.

SAT-DADDA-MITER