1 Case Study: GVML u16 Adder

1 Overview

GVML has a sophisticated adder for unsigned, 16-bit integers. It packs 26 APL commands into 12 instructions that execute in 12 clocks. It is a perfect test case for the compatibility theory above and for both of the safe-but-incompatible combinations of Section XXX. It is also a worthy target for BELEX.

Literal blocks

There are two ways to go about transcribing this algorithm into BELEX: literal and figurative. The default is figurative: the compiler performs vectorization (loop rerolling), section parallelism, instruction selection with automatic creation of temporaries, register allocation, automatic derivation of temporaries, and instruction scheduling (lane parallelism), amongst standard optimizations like dead-code removal and peephole optimizations. Code generated by an optimizing compiler will be better than naive APL, but not as good as hand-optimized APL. It is unlikely that a compiler could generate as good as this GVML adder without excessive amount of hinting, so much hinting that the programmer might just as well write the sophisticated code literally, similarly to the #asm keyword in C.

A literal transcription is a more friendly syntax for BLECCI. We already have done all the back-end work. In a BELEX literal block, the user furnishes all commands, specifies instruction lanes, names all temporary registers, and accesses machine resources like RL, GL, GGL, and RSP16 through thin abstractions. The compiler suspends vectorization and most other optimizations in a literal block.

2 Section Numbers & Section Masks

Section numbers are 1D arrays of exponents of 2. For instance, section mask 0x1111 corresponds to bit numbers {0, 4, 8, 12}, or, more conveniently in hex, {0, 4, 8, *C*} because $0\,x1111 == 2^0 + 2^4 + 2^7 + 2^{12}$. We pack ordered lists of bit numbers into strings for convenience.

In[ ]:=
```
ClearAll[bitNumbersFromMask, maskFromBitNumbers];
bitNumbersFromMask[mask_] :=
  Flatten[Position[Reverse[IntegerDigits[mask, 2]], 1] - 1];
maskFromBitNumbers[bitNumbers_] := Plus @@ (2^#1 & /@ bitNumbers);
```

In[ ]:=
```
If[unitTesting,
 Module[{i}, For[i = 0; i < 2^16, i++,
   Assert[i === maskFromBitNumbers@bitNumbersFromMask@i]]]]
```

Section mask 16^^1111 corresponds to hex bit numbers {0, 4, 8, *C*} = `"048C"`,
`16^^1111 << 1` to `16^^2222 = {1,5,9,D} = "159D"`,
`16^^1111 << 2` to `16^^4444 = {2,6,A,E} = "26AE"`, and

`16^^1111 << 3` to `16^^8888 = {3,7,B,F} = "37BF"`.

The section mask `16^^3333` corresponds to this list of bit-numbers: `{0,1,4,5,8,9,C,D}="014589CD"`

Here are some convenience functions for splicing arguments from frequently used masks in the examples below. It will become obvious how to use them.

In[ ]:=
```
ClearAll[
  s048C, s159D, s26AE, s37BF, s4567, s89AB, sCDEF, (* lists of bit numbers *)
  sFFFF, sFFFE, s7FFF, s3333, (* exceptions: actual bit masks *)
  m048C, m159D, m26AE, m37BF, m4567, m89AB, mCDEF, (* lists of bit numbers *)
  mFFFF, mFFFE, m7FFF, m3333] (* exceptions: actual bit masks *);
m048C = 16^^1111; s048C[x_] := Sequence[m048C, x];
m159D = 16^^2222; s159D[x_] := Sequence[m159D, x];
m26AE = 16^^4444; s26AE[x_] := Sequence[m26AE, x];
m37BF = 16^^8888; s37BF[x_] := Sequence[m37BF, x];
m4567 = maskFromBitNumbers[{4, 5, 6, 7}];
s4567[x_] := Sequence[m4567, x];
m89AB = maskFromBitNumbers[{8, 9, 10, 11}];
s89AB[x_] := Sequence[m89AB, x];
mCDEF = maskFromBitNumbers[{12, 13, 14, 15}];
sCDEF[x_] := Sequence[mCDEF, x];

mFFFE = 16^^FFFE; sFFFE[x_] := Sequence[mFFFE, x];
m7FFF = 16^^7FFF; s7FFF[x_] := Sequence[m7FFF, x];
mFFFF = 16^^FFFF; sFFFF[x_] := Sequence[mFFFF, x];
m3333 = 16^^3333; s3333[x_] := Sequence[m3333, x];
(* mandatory blank line *)
```

3 Notation

Section masks are ordered: 16^^1111 means $\{0, 4, 8, C\}$ in that order.

To save space, write section-number lists in ordered hex strings without commas and without a leading `"0x"`: $c_{\text{"159D"}}$ means $c_{\{1,5,9,D=13\}}$ and not $c_{\text{"0x159D"}}$, which would mean $c_{\text{"023478AC"}}$. Lack of a subscript means “all sections,” as in $c == c_{\text{"0123456789ABCDEF"}}$. A numerical subscript, as in $c_{16\wedge\wedge 169D}$, is a section mask, not a section-number list.

Section Lists are Ordered

Section lists in strings are ordered: $c_{\text{"048C"}} <= c_{\text{"4C08"}}$ specifies a permutation in parallel, whereby section 4 of c is moved to section 0, section C is moved to section 4, and so on.

Repeated Indices

Repeated indices are permitted; $c_{\text{"048C"}} <= c_{\text{"4405"}}$ means copy section 4 to section 0, section 4 to section 4, section 0 (the old section 0, not the new one) to section 8, and section 5 to section C. The copy is done in parallel; that’s why the old section 0 is copied to section 8.

Naming Conventions

To save space, write (*x* ^ *y*) as x▴y, bound more tightly than & or |. It is Plus, without carry, in GF(2), the field of Booleans. We might write it as *x* ⊻ *y*, but that's longer and uglier.

Machine variables and functions are in all caps, like `RL`, `GL`, `GGL`, `BGGL()`, `RSP16`. User-visible BELEX variables and functions, are in lower case, like add_u16 *x*, *y*, and x▴y. Notice that the *x* and *y* in x▴y are not italicized, even though italics are standard in mathematical notation. The omission of italics is not significant. Mathematica removes italics from names of more than one character.

It is already natural to write (*x* & *y*) as xy. Notice, again, it’s not italicized.

Multiple, comma-separated VRs, (up to 3), in an SB expression, as in `SB[x,y]`, are implicitly ANDed.

TODO: when needed, refactor BELOPS below to honor multiple VR/SBs.

4 Background: Ripple-Carry Adder (UNDONE)

5 Analysis (UNDONE)

Objective: prove that the effects an unsigned, 16-bit addition.

```
\libs-gvml\src\include\arith_cmds_inst_apl.h
```

An edited copy appears below, with line-by-line analysis and BELEX design in the following sections.

```
/*
 * addsub_frag _add _u16 (RN_REG x, RN_REG y, RN_REG res, RN_REG x_xor _y, RN_REG cout1)
 * cout1[0] = X[0]^Y[0];
 * cout1[1] = (X[0]^Y[0]) & (X[1]^Y[1]);
 * cout1[2] = (X[0]^Y[0]) & (X[1]^Y[1]) & (X[2]^Y[2]);
 * cout1[3] = (X[0]^Y[0]) & (X[1]^Y[1]) & (X[2]^Y[2]) & (X[3]^Y[3]);
 * cout0[0] = X[0] & Y[0];
 * cout0[1] = X[1] & Y[1] | (COUT0[0] & X[1]^Y[1]);
 * cout0[2] = X[2] & Y[2] | (COUT0[1] & X[2]^Y[2]);
 * cout0[3] = X[3] & Y[3] | (COUT0[2] & X[3]^Y[3]);
*/
  #define ARITH_ADD _U16 _GL _CO _FLAG _CO _T0 _INST (x, y, res, x_xor _y, cout1)
```

6 The Adder in BLECCI

7 Literal BELEX

Suppose we could write BELEX like this:

```
@belex_literal_block(build_examples=add_u16_gvml_examples)
def add_u16_gvml(out: VR, x: VR, y: VR):
    # can compiler automatically deduce the need for this temporary from
    # subexpressions like x() ^ y(), or must the user supply it explicitly?
    x▴y = VR(0)  # same as IR.var(0)
    cout1 = VR(0)
```

Instructions 1-3

```
   1.1  FFFF    :   RL = SB[x];
}{ 2.1  FFFF    :   RL ^= SB[y];
   2.2  3333    :   GGL = RL;      (* Safe 9.2 -- uses updated RL *)
}{ 3.1  FFFF    :   SB[x⊻y] = RL;  (* separate instruction, new RL *)
```

Let “RL” be a system-supplied function or symbol with the shape of a VR.

Lack of a section mask or section list on the left-hand, as in `RL() = ...`, side implies "all sections."

BELEX statements in a list (surrounded by square brackets) can be laned together, in order, into an instruction.

```
  # literal BELEX      # equivalent BLECCI (from nanosim-apu/test/test_BLECCI_add_u16.py
  #                    #                     ::bb_add_u16_24_gvml)
  # ================== # ===============================================================
  [ RL() <= x() ^ y()  # lane_start()
                       # rl_from_sb(fs, x)
                       # lane_end()    # compiler deduces the lane boundary
                       # lane_start()  # compiler splits expression into 2 commands
                       # rl_xor_equals_sb(fs, y)
```

Let “BGGL” be a system-supplied function or symbol with the shape of a VR and with special semantics, as described in Subsection XXX, Section XXX.

Inside round brackets, a string specifies a list of section numbers in hexadecimal. Thus “048C” means [0, 4, 8, 12]. Order does not matter, so “048C” is the same as “80C4” and all 23 other permutations of those digits.

A single integer inside round brackets is a section mask. The corresponding section list is the list of exponents in an expansion of the mask in binary. For instance, the section list corresponding to mask 0xFEED is It is the sumeither an exact power of two or not. If it’s an exact power of two, it specifies a single section. The number

A section mask or section list on the left hand side automatically specifies the section mask or section list for any empty lists on the right-hand side. The following means `GGL("014589CD") = RL("014589CD")`.

```
  # literal BELEX                 # equivalent BLECCI
  # =========================== # ================================
  , BGGL("014589CD") <= RL()      # lane_start(); ggl_from_rl(threes)
  ]                               # lane_end()
  [ x▴y() <= RL()                 # lane_start(); sb_fromSrc(fs, x_xor_y, 'RL')
  ]                               # land_end()
```

Instruction 4

```
}{ 4.1  1111     :   SB[cout1] = RL;       (* change c"048C" *)
   4.2  1111<<1  :   SB[cout1] = GGL;      (* change c"159D" *)
   4.3  1111<<2 :   RL = SB[x⊻y] & GGL;      /* (CIN & x⊻y) */
   4.4  3333    :   RL = SB[x,y];            (* safe 9.1 write into 4.1 SB before read into RL
*)
                                             /* CALC COUT0  */
```

```
  # literal BELEX                    # equivalent BLECCI
  # ================================= # ============================================
  [ cout1("048C")  <= RL()           # lane_start(); sb_from_src(ones, cout1, 'RL')
  , cout1("159D")  <= BGGL()         # sb_from_src(ones<<1, cout1, 'GGL')
  , RL("26AE")     <= x▴y() & BGGL() # rl.from_sb_and_src(ones<<2, x_xor_y, 'GGL')
  , RL("014589CD") <= x() & y()      # rl_from_sb(threes, [x, y])
  ]                                  # lane_end(); # compiler deduces SB[x, y]
```

Instruction 5

```
}{ 5.1  1111<<2 :   SB[cout1] = RL;          (* 26AE: Safe 9.1: write SB b4 updating RL in 5.4*
)
   5.2  1111<<3 :   RL = SB[x⊻y] & NRL;      /* 37BF: (CIN & x⊻y) */
   5.3  1111<<1 :   RL |= SB[x⊻y] & NRL;     /* 159D: (CIN & x⊻y) */
   5.4  1111<<2 :   RL = SB[x,y];            /* 26AE: CALC COUT0 */
```

```
  # literal BELEX                    # equivalent BLECCI
  # ================================= # ==================================================
  [ cout1("26AE") <= RL()            # lane_start(); sb_from_src(ones<<2, cout1, 'RL')
  , RL("37BF")    <= x▴y() & NRL()   # rl_from_sb_and_src(ones<<3, x_xor_y, 'NRL')
  , RL("159D")    |= x▴y() & NRL()   # rl_or_equals_sb_and_src(ones<<1, x_xor_y, 'NRL')
  , RL("26AE")    <= x() & y()       # rl_from_sb(ones<<2, [x, y])
  ]                                  # lane_end(); # compiler deduces SB[x, y]
```

Instruction 6

```
}{ 6.1  1111<<3 :   SB[cout1] = RL;          (* 37BF: Safe 9.1: Write SB before updating RL *)
   6.2  1111<<3 :   RL = SB[x,y];            (* 37BF *)
   6.3  1111<<2 :   RL |= SB[x⊻y] & NRL;     /* 26AE: (CIN & x⊻y) = COUT? */
   6.4  0001    :   GGL = RL;
```

A single integer in list brackets is a section list. `BGGL([0])` is the same as `BGGL("0")`, `BGGL(1)`, `BGGL(0x1)`, etc.

```
 # literal BELEX                    # equivalent BLECCI
 # ================================= # ==================================================
 [ cout1("37BF") <= RL()            # lane_start(); sb_from_src(ones<<3, cout1, 'RL')
 , RL("37BF")    <= x() & y()       # rl_from_sb(ones<<3, [x, y])
 , RL("26AE")    |= x▴y() & NRL()   # rl_or_equals_sb_and_src(ones<<2, x_xor_y, 'NRL')
 , BGGL([0])     <= RL()            # ggl_from_rl(os)
 ]                                  # lane_end()
```

Instruction 7

```
}{ 7.1  1111<<3 :   RL |= SB[x⊻y] & NRL;     /* (CIN & x⊻y) */
   7.2  0001<<3 :   GL = RL;             (* Safe 9.2, update RL first *)
   7.3  0001    :   RL = SB[cout1];
```

```
 # literal BELEX                    # equivalent BLECCI
 # ================================= # ==================================================
[ RL("37BF")    |= x▴y() & NRL()    # lane_start()
                                    # rl_or_equals_sb_and_src(ones<<2, x_xor_y, 'NRL')
, GL()          <= RL("3")          # # also GL("3") <= RL() or GL("3") <= RL("3")
                                    # gl_from_rl(os<<3)
, RL("0")       <= cout1()          # rl_from_sb(os, cout1)
]                                   # lane_end()
```

Instruction 8

```
}{ 8.1  000F<<4 :   RL |= SB[cout1] & GL;   (* Safe 9.2 Read into RL before broadcast *)
   8.2  0001<<7 :   GL = RL;            (* Safe 9.2 --- uses new RL *)
   8.3  0001    :   SB[res] = RL;
```

```
 # literal BELEX                     # equivalent BLECCI
 # ================================= # ==================================================
 [ RL("4567")    |= cout1() & GL()   # lane_start()
                                     # rl_or_equals_sb_and_src(one_f<<4, cout, 'GL')
 , GL()          <= RL("7")          # gl_from_rl(os<<7)
 , out("0")      <= RL()             # sb_from_src(os, out, 'RL')
 ]                                   # lane_end()
```

Instruction 9

```
}{ 9.1  000F<<8     :   RL |= SB[cout1] & GL;
   9.2  0001<<11:   GL = RL;
   9.3  0001        :   RL = GGL;
```

```
 # literal BELEX                     # equivalent BLECCI
 # ================================= # ==================================================
 [ RL("89AB")    |= cout1() & GL()   # lane_start()
                                     # rl_or_equals_sb_and_src(one_f<<8, cout1, 'GL')
 , GL()          <= RL("B")          # gl_from_rl(os<<11)
 , RL("0")       <= BGGL()           # rl_from_src(os, 'GGL')
 ]                                   # lane_end()
```

Instruction 10

```
}{ 10.1 000F<<12:   RL |= SB[cout1] & GL;
   10.2 0001<<15:   GL = RL;
```

```
 # literal BELEX                     # equivalent BLECCI
 # ================================= # ==================================================
 [ RL("CDEF")    |= cout1() & GL()   # lane_start()
                                     # rl_or_equals_sb_and_src(one_f<<12, cout1, 'GL')
 , GL()          <= RL("F")          # gl_from_rl(os<<15)
 ]                                   # lane_end()
```

Instruction 11

```
}{ 11.2 0001 << C_FLAG  :   SB[RN_REG_FLAGS] = GL;
   11.3 ~0001           :   RL = SB[x_xor_y] ^ NRL;
```

```
 # literal BELEX                     # equivalent BLECCI
 # ================================= # ==================================================
 [ RN_REG_FLAGS(C_FLAG) <= GL()      # lane_start()
                                     # sb_from_src(os<<C_FLAG, flags, 'GL')
 , RL(0xFFFE) <= x▴y() ^ NRL()       # rl_from_sb_xor_src(~os, x_xor_y, 'NRL')
 # same as RL(0xFFFE) <= x▴y() ^ RL("7FFF")
 ]                                   lane_end()
```

Instruction 12

```
}{ 12.4 ~0001   :   SB[res] = RL;
```

```
# literal BELEX                     # equivalent BLECCI
# ================================= # ==================================================
[ out(0xFFFE) <= RL()               # lane_start(); sb_from_src(~os, out, 'RL')
]                                   # lane_end()
```