// ==============================================================

// File generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC

// Version: 2017.2

// Copyright (C) 1986-2017 Xilinx, Inc. All Rights Reserved.

//

// ==============================================================

#ifndef AESL\_PKG\_HH

#define AESL\_PKG\_HH

#include "systemc.h"

namespace ap\_rtl {

////////////////////////////////////////////////////////////////

// Comparisons

////////////////////////////////////////////////////////////////

template <int T, int W0, int W1>

bool esl\_seteq(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0>=1 && W1>=1);

if (W0 > W1)

return (i0.range(W1-1, 0) == i1);

else

return (i1.range(W0-1, 0) == i0);

assert(W0 == W1);

return (i0 == i1);

}

template <int T, int W0, int W1>

bool esl\_setne(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0>=1 && W1>=1);

if (W0 > W1)

return (i0.range(W1-1, 0) != i1);

else

return (i1.range(W0-1, 0) != i0);

assert(W0 == W1);

return (i0 != i1);

}

template <int T, int W0, int W1>

bool esl\_seteq(const sc\_logic& i0, const sc\_logic& i1) {

assert(W0 == W1);

return (i0 == i1);

}

template <int T, int W0, int W1>

bool esl\_setne(const sc\_logic& i0, const sc\_logic& i1) {

assert(W0 == W1);

return (i0 != i1);

}

template <int T, int W0>

bool esl\_not(bool i0) {

return (!i0);

}

template <int T, int W0>

sc\_logic esl\_not(const sc\_logic& i0) {

return (~i0);

}

template <int T, int W0, int W1>

bool esl\_setle(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return false;

return (sc\_biguint<W0>(i0) <= sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

bool esl\_setge(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return false;

return (sc\_biguint<W0>(i0) >= sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

bool esl\_setlt(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return false;

return (sc\_biguint<W0>(i0) < sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

bool esl\_setgt(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return false;

return (sc\_biguint<W0>(i0) > sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_eq(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0 == W1);

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return sc\_lv<1>(i0 == i1);

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_ne(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0 == W1);

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return sc\_lv<1>(i0 != i1);

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_ugt(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_biguint<W0>(i0) > sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_uge(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_biguint<W0>(i0) >= sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_ult(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_biguint<W0>(i0) < sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_ule(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_biguint<W0>(i0) <= sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_sgt(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_bigint<W0>(i0) > sc\_bigint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_sge(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_bigint<W0>(i0) >= sc\_bigint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_slt(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_bigint<W0>(i0) < sc\_bigint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<1> esl\_icmp\_sle(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<1>();

return (sc\_bigint<W0>(i0) <= sc\_bigint<W1>(i1));

}

template <int T, int W0>

sc\_lv<T> esl\_trunc(const sc\_lv<W0>& i0) {

assert(T <= W0);

return (i0.range(T-1, 0));

}

template <int T, int W0>

sc\_lv<T> esl\_sext(const sc\_lv<W0>& i0) {

assert(T >= W0);

if (!i0.is\_01()) return sc\_lv<T>();

return ((sc\_lv<T>)(sc\_bigint<W0>(i0)));

}

template <int T, int W0>

sc\_lv<T> esl\_zext(const sc\_lv<W0>& i0) {

assert(T >= W0);

if (!i0.is\_01()) return sc\_lv<T>();

return ((sc\_lv<T>)(sc\_biguint<W0>(i0)));

}

template <int T, int W0>

sc\_lv<T> esl\_bitcast(const sc\_lv<W0>& i0) {

assert(T == W0);

return i0;

}

////////////////////////////////////////////////////////////////

// Conversions

////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////

// Logic/Arithmatic operations

////////////////////////////////////////////////////////////////

template <int T, int W0, int W1>

sc\_lv<T> esl\_and(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0 == W1);

return (i0 & i1);

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_or(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0 == W1);

return (i0 | i1);

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_xor(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(W0 == W1);

return (i0 ^ i1);

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_shl(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

//assert(W1 <= 32);

if (!i1.is\_01()) return sc\_lv<T>();

return (i0 << (unsigned short)i1.to\_uint());

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_lshr(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

//assert(W1 <= 32);

if (!i1.is\_01()) return sc\_lv<T>();

return (i0 >> (unsigned short)i1.to\_uint());

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_ashr(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

//assert(W1 <= 32);

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) >> (unsigned short)i1.to\_uint());

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_add(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) + sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_sub(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) - sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_mul\_UU(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_biguint<W0>(i0) \* sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_mul\_SU(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) \* sc\_biguint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_mul\_US(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_biguint<W0>(i0) \* sc\_bigint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_mul\_SS(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) \* sc\_bigint<W1>(i1));

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_udiv(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

sc\_biguint<W1> v1(i1);

if (v1.to\_uint() == 0) return sc\_lv<T>();

return (sc\_biguint<W0>(i0) / v1);

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_sdiv(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

sc\_bigint<W1> v1(i1);

if (v1.to\_uint() == 0) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) / v1);

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_urem(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

sc\_biguint<W1> v1(i1);

if (v1.to\_uint() == 0) return sc\_lv<T>();

return (sc\_biguint<W0>(i0) % v1);

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_srem(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

if (!i0.is\_01() || !i1.is\_01()) return sc\_lv<T>();

sc\_bigint<W1> v1(i1);

if (v1.to\_uint() == 0) return sc\_lv<T>();

return (sc\_bigint<W0>(i0) % v1);

}

template <int T, int W0>

sc\_lv<T> esl\_ctlz(const sc\_lv<W0>& i0) {

int count = 0;

for (unsigned i = W0-1; i >= 0; --i) {

if (i0[i] == SC\_LOGIC\_0) count ++;

else break;

}

return count;

}

template <int T, int W0>

sc\_lv<T> esl\_cttz(const sc\_lv<W0>& i0) {

int count = 0;

for (unsigned i = 0; i < W0; ++i) {

if (i0[i] == SC\_LOGIC\_0) count ++;

else break;

}

return count;

}

////////////////////////////////////////////////////////////////

// Other operations

////////////////////////////////////////////////////////////////

template <int T, int W0, int W1, int W2>

sc\_lv<T> esl\_select(const sc\_lv<W0>& i0,

const sc\_lv<W1>& i1, const sc\_lv<W2>& i2) {

assert((W0 == 1) && (W1 == W2));

if (!i0[0].is\_01()) return sc\_lv<T>();

bool flag = (i0[0].to\_bool());

return (flag ? i1 : i2);

}

template <int W0, int W1>

inline sc\_lv<W0+W1> esl\_concat(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

return (i0, i1);

}

template <int T, int W0, int W1>

inline sc\_lv<T> esl\_bitselect(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

assert(T == 1);

if (!i1.is\_01()) return sc\_lv<T>();

unsigned int loc = (sc\_biguint<W1>(i1)).to\_uint();

//assert(W0 >= loc);

if (W0 <= loc)

return sc\_lv<T>();

return i0.range(loc, loc);

}

template <int T, int W0, int W1, int W2>

inline sc\_lv<T> esl\_bitset(const sc\_lv<W0>& i0,

const sc\_lv<W1>& i1, const sc\_lv<W2>& i2) {

assert(T == W0);

if (!i1.is\_01()) return sc\_lv<T>();

unsigned int loc = (sc\_biguint<W1>(i1)).to\_uint();

//assert(W0 >= loc);

if (W0 <= loc)

return sc\_lv<T>();

sc\_lv<W0> res = i0;

res[loc] = i2.or\_reduce();

return res;

}

template <int T, int W0, int W1, int W2>

inline sc\_lv<T> esl\_partselect(const sc\_lv<W0>& i0,

const sc\_lv<W1>& iLo, const sc\_lv<W2>& iHi) {

if (!iLo.is\_01() || !iHi.is\_01()) return sc\_lv<T>();

unsigned int Lo = (sc\_biguint<W1>(iLo)).to\_uint();

unsigned int Hi = (sc\_biguint<W2>(iHi)).to\_uint();

unsigned int rsize = abs((int)(Hi-Lo)) + 1;

// Warning out the dont-care situation.

if (Lo >= W0 || Hi >= W0) {

// std::cout << "Warning: partselect out of range!\n";

return sc\_lv<T>();

}

assert(W0 >= rsize);

return i0.range(Hi, Lo);

}

template <int T, int W0, int W1, int WLo, int WHi>

inline sc\_lv<T> esl\_partset(const sc\_lv<W0>& i0,

const sc\_lv<W1>& i1,

const sc\_lv<WLo>& iLo, const sc\_lv<WHi>& iHi) {

if (!iLo.is\_01() || !iHi.is\_01()) return sc\_lv<T>();

assert(W0 == T);

unsigned int Lo = (sc\_biguint<WLo>(iLo)).to\_uint();

unsigned int Hi = (sc\_biguint<WHi>(iHi)).to\_uint();

unsigned int rsize = abs((int)(Lo-Hi)) + 1;

// Warning out the dont-care situation.

if (Hi >= T || Lo >= T) {

// std::cout << "Warning: partset out of range!\n";

return sc\_lv<T>();

}

assert(W0 >= rsize);

sc\_lv<T> res = i0;

res.range(Hi, Lo) = ((sc\_lv<T>)i1).range(rsize-1, 0);

return res;

}

template <int T, int W0>

sc\_lv<T> esl\_orreduce(const sc\_lv<W0>& i0) {

assert(T == 1);

sc\_lv<1> res;

res[0] = i0.or\_reduce();

return res;

}

template <int T, int W0>

inline sc\_lv<T> esl\_andreduce(const sc\_lv<W0>& i0) {

assert(T == 1);

sc\_lv<1> res;

res[0] = i0.and\_reduce();

return res;

}

template <int T, int W0>

inline sc\_lv<T> esl\_xorreduce(const sc\_lv<W0>& i0) {

assert(T == 1);

sc\_lv<1> res;

res[0] = i0.xor\_reduce();

return res;

}

template <int T, int W0>

inline sc\_lv<T> esl\_nandreduce(const sc\_lv<W0>& i0) {

assert(T == 1);

sc\_lv<1> res;

res[0] = i0.nand\_reduce();

return res;

}

template <int T, int W0>

inline sc\_lv<T> esl\_xnorreduce(const sc\_lv<W0>& i0) {

assert(T == 1);

sc\_lv<1> res;

res[0] = i0.xnor\_reduce();

return res;

}

template <int T, int W0>

inline sc\_lv<T> esl\_norreduce(const sc\_lv<W0>& i0) {

assert(T == 1);

sc\_lv<1> res;

res[0] = i0.nor\_reduce();

return res;

}

////////////////////////////////////////////////////////////////

// Floating point operations

////////////////////////////////////////////////////////////////

struct esl\_FP {

static float esl\_INTSP(unsigned int x) { return (\*(float\*)(&(x))); }

static double esl\_INTDP(unsigned long long x) { return (\*(double\*)(&(x))); }

static unsigned int esl\_SPINT(float x) { return (\*(unsigned int\*)(&(x))); }

static unsigned long long esl\_DPINT(double x) { return (\*(unsigned long long\*)(&(x))); }

};

#define esl\_LVSP(x) esl\_FP::esl\_INTSP((x).to\_uint())

#define esl\_LVDP(x) esl\_FP::esl\_INTDP((x).to\_uint64())

#define esl\_sitodp(lv) esl\_FP::esl\_DPINT(double((lv).to\_int()))

#define esl\_dptosi(lv) (int(esl\_LVDP(lv)))

#define esl\_dadd(x, y) \

esl\_FP::esl\_DPINT(esl\_LVDP(x) + esl\_LVDP(y))

#define esl\_fadd(x, y) \

esl\_FP::esl\_SPINT(esl\_LVSP(x) + esl\_LVSP(y))

#define esl\_dsub(x, y) \

esl\_FP::esl\_DPINT(esl\_LVDP(x) - esl\_LVDP(y))

#define esl\_fsub(x, y) \

esl\_FP::esl\_SPINT(esl\_LVSP(x) - esl\_LVSP(y))

#define esl\_dmul(x, y) \

esl\_FP::esl\_DPINT(esl\_LVDP(x) \* esl\_LVDP(y))

#define esl\_fmul(x, y) \

esl\_FP::esl\_SPINT(esl\_LVSP(x) \* esl\_LVSP(y))

#define esl\_ddiv(x, y) \

esl\_FP::esl\_DPINT(esl\_LVDP(x) / esl\_LVDP(y))

#define esl\_fdiv(x, y) \

esl\_FP::esl\_SPINT(esl\_LVSP(x) / esl\_LVSP(y))

#define esl\_drem(x, y) \

esl\_FP::esl\_DPINT(esl\_LVDP(x) % esl\_LVDP(y))

#define esl\_frem(x, y) \

esl\_FP::esl\_SPINT(esl\_LVSP(x) % esl\_LVSP(y))

//#define esl\_dsqrt(x) esl\_FP::esl\_DPINT(sqrt(esl\_LVDP(x)))

//#define esl\_fsqrt(x) esl\_FP::esl\_SPINT(sqrt(esl\_LVSP(x)))

#define esl\_DFCMP\_FALSE(x, y) \

(false)

#define esl\_DFCMP\_ORD(x, y) \

(!isnan(esl\_LVDP(x)) & !isnan(esl\_LVDP(y)))

#define esl\_DFCMP\_OEQ(x, y) \

(esl\_DFCMP\_ORD(x, y) & (esl\_LVDP(x) == esl\_LVDP(y)))

#define esl\_DFCMP\_OGT(x, y) \

(esl\_DFCMP\_ORD(x, y) & (esl\_LVDP(x) > esl\_LVDP(y)))

#define esl\_DFCMP\_OGE(x, y) \

(esl\_DFCMP\_ORD(x, y) & (esl\_LVDP(x) >= esl\_LVDP(y)))

#define esl\_DFCMP\_OLT(x, y) \

(esl\_DFCMP\_ORD(x, y) & (esl\_LVDP(x) < esl\_LVDP(y)))

#define esl\_DFCMP\_OLE(x, y) \

(esl\_DFCMP\_ORD(x, y) & (esl\_LVDP(x) <= esl\_LVDP(y)))

#define esl\_DFCMP\_ONE(x, y) \

(esl\_DFCMP\_UNO(x, y) & (esl\_LVDP(x) != esl\_LVDP(y)))

#define esl\_DFCMP\_UNO(x, y) \

(isnan(esl\_LVDP(x)) | isnan(esl\_LVDP(y)))

#define esl\_DFCMP\_UEQ(x, y) \

(esl\_DFCMP\_UNO(x, y) | (esl\_LVDP(x) == esl\_LVDP(y)))

#define esl\_DFCMP\_UGT(x, y) \

(esl\_DFCMP\_UNO(x, y) | (esl\_LVDP(x) > esl\_LVDP(y)))

#define esl\_DFCMP\_UGE(x, y) \

(esl\_DFCMP\_UNO(x, y) | (esl\_LVDP(x) >= esl\_LVDP(y)))

#define esl\_DFCMP\_ULT(x, y) \

(esl\_DFCMP\_UNO(x, y) | (esl\_LVDP(x) < esl\_LVDP(y)))

#define esl\_DFCMP\_ULE(x, y) \

(esl\_DFCMP\_UNO(x, y) | (esl\_LVDP(x) <= esl\_LVDP(y)))

#define esl\_DFCMP\_UNE(x, y) \

(esl\_DFCMP\_UNO(x, y) | (esl\_LVDP(x) != esl\_LVDP(y)))

#define esl\_DFCMP\_TRUE(x, y) \

(true)

template <int T, int W0>

sc\_lv<T> esl\_dsqrt(const sc\_lv<W0>& x) {

sc\_lv<T> ret = esl\_FP::esl\_DPINT(sqrt(esl\_LVDP(x)));

return ret;

}

template <int T, int W0>

sc\_lv<T> esl\_fsqrt(const sc\_lv<W0>& x) {

sc\_lv<T> ret = esl\_FP::esl\_SPINT(sqrt(esl\_LVSP(x)));

return ret;

}

template <int T, int W0, int W1>

sc\_lv<T> esl\_getelementptr(const sc\_lv<W0>& i0, const sc\_lv<W1>& i1) {

return esl\_add<T, W0, W1>(i0, i1);

}

}

#endif

// 67d7842dbbe25473c3c32b93c0da8047785f30d78e8a024de1b57352245f9689