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

// 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\_COMP\_H

#define AESL\_COMP\_H

#include <systemc>

//#include "systemc.h"

#define DEBUG(x)

///CFU: combinational FU, no clock, reset and ce

//template<int NUM\_STAGE,

// int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

// int DOUT\_WIDTH=DIN0\_WIDTH>

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH> class AESLFUComp;

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE( AESLCFUComp )

{

public:

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

SC\_CTOR( AESLCFUComp );

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH> \*super;

virtual void sanity\_check() {}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{ if (super) return super->compute(in0, in1);

else return sc\_lv<DOUT\_WIDTH>(); }

void thread\_compute() {

DEBUG(

{

if (DIN0\_WIDTH == 64)

printf("[ACMP\_ADD] input: %llx %llx\n",

din0.read().to\_uint64(),

din1.read().to\_uint64());

else

printf("[ACMP\_ADD] input: %x %x\n",

din0.read().to\_uint(),

din1.read().to\_uint());

});

sc\_lv<DOUT\_WIDTH> result = compute(din0.read(), din1.read());

dout.write(result);

DEBUG(

{

if (DIN0\_WIDTH == 64)

printf("[ACMP\_ADD] comb output: %llx\n", result.to\_uint64());

else

printf("[ACMP\_ADD] comb output: %x\n", result.to\_uint());

});

}

};

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>::

AESLCFUComp(const sc\_module\_name name): sc\_module( name ), super(0) {

//sanity\_check();

SC\_METHOD( thread\_compute );

sensitive << din0;

sensitive << din1;

}

///FU: sequential FU, with clock, reset and ce

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE(AESLFUComp)

{

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH> core;

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

/// Dont use stage\_regvec[0].

sc\_signal< sc\_lv<DOUT\_WIDTH> > stage\_regvec[NUM\_STAGE];

SC\_CTOR( AESLFUComp );

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{ return sc\_lv<DOUT\_WIDTH>(); }

void thread\_stage() {

if (ce.read() == SC\_LOGIC\_0) return;

dout.write(stage\_regvec[NUM\_STAGE-1]);

for (unsigned i = NUM\_STAGE-1; i > 1; --i) {

stage\_regvec[i].write(stage\_regvec[i-1].read());

}

DEBUG(

{

if (DIN0\_WIDTH == 64)

printf("[ACMP\_ADD] reg output %llx\n",

stage\_regvec[NUM\_STAGE-1].read().to\_uint64());

else

printf("[ACMP\_ADD] reg output %x\n",

stage\_regvec[NUM\_STAGE-1].read().to\_uint());

});

}

};

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>::

AESLFUComp(const sc\_module\_name name): sc\_module( name ), core("CFU\_U") {

//sanity\_check();

core.din0(din0);

core.din1(din1);

core.dout(stage\_regvec[1]);

core.super = this;

if (NUM\_STAGE > 1)

{

SC\_METHOD( thread\_stage );

sensitive << clk.pos();

}

}

//Need refine here//////////////////////////////////////////////////////////////

///CFU: combinational FU, no clock, reset and ce

//template<int NUM\_STAGE,

// int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

// int DOUT\_WIDTH=DIN0\_WIDTH>

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH> class AESLFUComp\_seq;

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE( AESLCFUComp\_seq )

{

public:

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

SC\_CTOR( AESLCFUComp\_seq );

AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH> \*super;

virtual void sanity\_check() {}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{ if (super) return super->compute(in0, in1);

else return sc\_lv<DOUT\_WIDTH>(); }

void thread\_compute() {

DEBUG(

{

if (DIN0\_WIDTH == 64)

printf("[ACMP\_ADD] input: %llx %llx\n",

din0.read().to\_uint64(),

din1.read().to\_uint64());

else

printf("[ACMP\_ADD] input: %x %x\n",

din0.read().to\_uint(),

din1.read().to\_uint());

});

sc\_lv<DOUT\_WIDTH> result = compute(din0.read(), din1.read());

dout.write(result);

DEBUG(

{

if (DIN0\_WIDTH == 64)

printf("[ACMP\_ADD] comb output: %llx\n", result.to\_uint64());

else

printf("[ACMP\_ADD] comb output: %x\n", result.to\_uint());

});

}

};

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

AESLCFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>::

AESLCFUComp\_seq(const sc\_module\_name name): sc\_module( name ), super(0) {

//sanity\_check();

SC\_METHOD( thread\_compute );

sensitive << din0;

sensitive << din1;

}

///FU: sequential FU, with clock, reset, ce and start

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE(AESLFUComp\_seq)

{

AESLCFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH> core;

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_logic > start;

sc\_in< sc\_logic > done;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

/// Dont use stage\_regvec[0].

sc\_signal< sc\_lv<DOUT\_WIDTH> > stage\_regvec[NUM\_STAGE];

SC\_CTOR( AESLFUComp\_seq );

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{ return sc\_lv<DOUT\_WIDTH>(); }

void thread\_stage() {

if (ce.read() == SC\_LOGIC\_0) return;

dout.write(stage\_regvec[NUM\_STAGE-1]);

for (unsigned i = NUM\_STAGE-1; i > 1; --i) {

stage\_regvec[i].write(stage\_regvec[i-1].read());

}

DEBUG(

{

if (DIN0\_WIDTH == 64)

printf("[ACMP\_ADD] reg output %llx\n",

stage\_regvec[NUM\_STAGE-1].read().to\_uint64());

else

printf("[ACMP\_ADD] reg output %x\n",

stage\_regvec[NUM\_STAGE-1].read().to\_uint());

});

}

};

template<int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH,

int DOUT\_WIDTH>

AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>::

AESLFUComp\_seq(const sc\_module\_name name): sc\_module( name ), core("CFU\_U") {

//sanity\_check();

core.din0(din0);

core.din1(din1);

core.dout(stage\_regvec[1]);

core.super = this;

if (NUM\_STAGE > 1)

{

SC\_METHOD( thread\_stage );

sensitive << clk.pos();

}

}

//end///////////////////////////////////////////////////////////////////////////

template<int OPC\_WIDTH, int DIN\_WIDTH, int DOUT\_WIDTH>

struct FUComputeCore

{

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<OPC\_WIDTH>& opcode, const sc\_lv<DIN\_WIDTH>& in)=0;

};

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

// Abstract Multi-op combinational Functional Units.

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

template<int NUM\_STAGE, int OPC\_WIDTH, int DIN\_WIDTH, int DOUT\_WIDTH>

SC\_MODULE( AESLCFUMOComp )

{

public:

sc\_in< sc\_lv<OPC\_WIDTH> > opcode;

sc\_in< sc\_lv<DIN\_WIDTH> > din;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

public:

SC\_CTOR( AESLCFUMOComp ) {

assert(OPC\_WIDTH > 0 && DIN\_WIDTH > 0 && DOUT\_WIDTH > 0

&& NUM\_STAGE == 1);

SC\_METHOD( thread\_compute );

sensitive << opcode << din;

}

//virtual sc\_lv<DOUT\_WIDTH>

//compute(const sc\_lv<OPC\_WIDTH>& opcode, const sc\_lv<DIN\_WIDTH>& in) {

FUComputeCore<OPC\_WIDTH, DIN\_WIDTH, DOUT\_WIDTH>\* mCore;

void setComputeCore(FUComputeCore<OPC\_WIDTH, DIN\_WIDTH, DOUT\_WIDTH>\* core) {

mCore = core;

}

FUComputeCore<OPC\_WIDTH, DIN\_WIDTH, DOUT\_WIDTH>\* getComputeCore() {

return mCore;

}

public:

void thread\_compute() {

DEBUG( cerr << "[ACMP] " << name() << " Input: "

<< din.read() << endl; );

assert(mCore);

sc\_lv<DOUT\_WIDTH> result;

if (din.read().is\_01()) {

result = mCore->compute(opcode.read(), din.read());

}

else {

result = sc\_lv<DOUT\_WIDTH>();

}

dout.write(result);

DEBUG( cerr << "[ACMP] " << name() << " C-out: "

<< result << endl;);

}

};

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

// Abstract Pipelined Functional Units.

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

template<int NUM\_STAGE, int OPC\_WIDTH, int DIN\_WIDTH, int DOUT\_WIDTH>

SC\_MODULE( AESLFUMultiCycle )

{

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_lv<OPC\_WIDTH> > opcode;

sc\_in< sc\_lv<DIN\_WIDTH> > din;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

public:

SC\_CTOR( AESLFUMultiCycle ) {

assert(OPC\_WIDTH > 0 && DIN\_WIDTH > 0 && DOUT\_WIDTH > 0

&& NUM\_STAGE > 0);

SC\_METHOD( thread\_compute );

sensitive << opcode << din;

sensitive << clk;

if (NUM\_STAGE > 1)

{

SC\_METHOD( thread\_stage );

sensitive << (clk.pos());

}

}

//virtual sc\_lv<DOUT\_WIDTH>

//compute(const sc\_lv<OPC\_WIDTH>& opcode, const sc\_lv<DIN\_WIDTH>& in) {

FUComputeCore<OPC\_WIDTH, DIN\_WIDTH, DOUT\_WIDTH>\* mCore;

void setComputeCore(FUComputeCore<OPC\_WIDTH, DIN\_WIDTH, DOUT\_WIDTH>\* core) {

mCore = core;

}

FUComputeCore<OPC\_WIDTH, DIN\_WIDTH, DOUT\_WIDTH>\* getComputeCore() {

return mCore;

}

public:

sc\_signal< sc\_lv<DOUT\_WIDTH> > stage\_regvec[NUM\_STAGE];

void thread\_compute() {

DEBUG( cerr << "[ACMP] " << name() << " Input: "

<< din.read() << endl; );

assert(mCore);

sc\_lv<DOUT\_WIDTH> result;

if (din.read().is\_01() && opcode.read().is\_01()) {

result = mCore->compute(opcode.read(), din.read());

}

else {

result = sc\_lv<DOUT\_WIDTH>();

}

if (NUM\_STAGE > 1)

stage\_regvec[1].write(result);

else

dout.write(result);

DEBUG( cerr << "[ACMP] " << name() << " C-out: "

<< result << endl;);

}

void thread\_stage() {

if (ce.read() == SC\_LOGIC\_0) return;

dout.write(stage\_regvec[NUM\_STAGE-1]);

for (unsigned i = NUM\_STAGE-1; i > 1; --i) {

stage\_regvec[i].write(stage\_regvec[i-1].read());

}

DEBUG(

cerr << "[ACMP] " << name() << " R-out: "

<< stage\_regvec[NUM\_STAGE-1].read() << endl;);

}

};

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

/// Unary operator

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

template< int NUM\_STAGE,

int DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

SC\_MODULE( CFUUnaryOp )

{

public:

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

SC\_CTOR(CFUUnaryOp) {

mMCModule =

new AESLFUMultiCycle

<NUM\_STAGE, 1, DIN0\_WIDTH, DOUT\_WIDTH>("u\_U");

mMCModule->opcode(sigone);

mMCModule->din(din0);

mMCModule->dout(dout);

mMCModule->clk(clk);

mMCModule->reset(reset);

mMCModule->ce(ce);

SC\_METHOD(IOConnection);

}

AESLFUMultiCycle<NUM\_STAGE, 1, DIN0\_WIDTH, DOUT\_WIDTH>\* mMCModule;

void setComputeCore(FUComputeCore<1, DIN0\_WIDTH, DOUT\_WIDTH>\* core) {

mMCModule->setComputeCore(core);

}

private:

sc\_signal< sc\_lv<1> > sigone;

sc\_signal< bool > clk;

sc\_signal< sc\_logic > reset;

sc\_signal< sc\_logic > ce;

void IOConnection() {

sigone.write(sc\_lv<1>("1"));

}

};

template< int NUM\_STAGE,

int DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

SC\_MODULE( FUUnaryOp )

{

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

SC\_CTOR(FUUnaryOp) {

mMCModule =

new AESLFUMultiCycle

<NUM\_STAGE, 1, DIN0\_WIDTH, DOUT\_WIDTH>("u\_U");

mMCModule->opcode(sigone);

mMCModule->din(din0);

mMCModule->dout(dout);

mMCModule->clk(clk);

mMCModule->reset(reset);

mMCModule->ce(ce);

SC\_METHOD(IOConnection);

sensitive << clk;

}

AESLFUMultiCycle<NUM\_STAGE, 1, DIN0\_WIDTH, DOUT\_WIDTH>\* mMCModule;

void setComputeCore(FUComputeCore<1, DIN0\_WIDTH, DOUT\_WIDTH>\* core) {

mMCModule->setComputeCore(core);

}

private:

sc\_signal< sc\_lv<1> > sigone;

void IOConnection() {

sigone.write(sc\_lv<1>("1"));

}

};

template< int NUM\_STAGE,

int OPC\_WIDTH,

int DIN0\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE( FUMOUnaryOp )

{

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_lv<OPC\_WIDTH> > opcode;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

SC\_CTOR(FUMOUnaryOp) {

mMCModule =

new AESLFUMultiCycle

<NUM\_STAGE, OPC\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>("mu\_U");

mMCModule->opcode(opcode);

mMCModule->din(din0);

mMCModule->dout(dout);

mMCModule->clk(clk);

mMCModule->reset(reset);

mMCModule->ce(ce);

}

AESLFUMultiCycle<NUM\_STAGE, OPC\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>\* mMCModule;

void setComputeCore(FUComputeCore<OPC\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>\* core) {

mMCModule->setComputeCore(core);

}

};

template< int NUM\_STAGE,

int DIN0\_WIDTH,

int DIN1\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE (CFUBinaryOp)

{

public:

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

AESLFUMultiCycle<NUM\_STAGE, 1, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\*

mMCModule;

SC\_CTOR( CFUBinaryOp ) {

mMCModule =

new AESLFUMultiCycle

<NUM\_STAGE, 1, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>("b\_u");

//sigone.write(sc\_lv<1>("1"));

mMCModule->opcode(sigone);

mMCModule->din(din\_sig);

mMCModule->dout(dout);

mMCModule->clk(clk);

mMCModule->reset(reset);

mMCModule->ce(ce);

SC\_METHOD(IOConnection);

sensitive << din0 << din1;

}

void setComputeCore(FUComputeCore<1, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\*

core) {

mMCModule->setComputeCore(core);

}

sc\_signal< sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> > din\_sig;

sc\_signal< sc\_lv<1> > sigone;

sc\_signal< bool > clk;

sc\_signal< sc\_logic > reset;

sc\_signal< sc\_logic > ce;

void IOConnection() {

sigone.write(sc\_lv<1>("1"));

sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> din\_tmp;

din\_tmp.range(0, DIN0\_WIDTH-1) =

din0.read().range(0, DIN0\_WIDTH-1);

din\_tmp.range(DIN0\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH-1) =

din1.read().range(0, DIN1\_WIDTH-1);

din\_sig.write(din\_tmp);

}

};

template< int NUM\_STAGE,

int DIN0\_WIDTH,

int DIN1\_WIDTH,

int DOUT\_WIDTH>

SC\_MODULE (FUBinaryOp)

{

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

AESLFUMultiCycle<NUM\_STAGE, 1, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\*

mMCModule;

SC\_CTOR( FUBinaryOp ) {

mMCModule =

new AESLFUMultiCycle

<NUM\_STAGE, 1, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>("b\_u");

//sigone.write(sc\_lv<1>("1"));

mMCModule->opcode(sigone);

mMCModule->din(din\_sig);

mMCModule->dout(dout);

mMCModule->clk(clk);

mMCModule->reset(reset);

mMCModule->ce(ce);

SC\_METHOD(IOConnection);

sensitive << din0 << din1 << clk;

}

void setComputeCore(FUComputeCore<1, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\*

core) {

mMCModule->setComputeCore(core);

}

sc\_signal< sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> > din\_sig;

sc\_signal< sc\_lv<1> > sigone;

void IOConnection() {

sigone.write(sc\_lv<1>("1"));

sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> din\_tmp;

din\_tmp.range(0, DIN0\_WIDTH-1) =

din0.read().range(0, DIN0\_WIDTH-1);

din\_tmp.range(DIN0\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH-1) =

din1.read().range(0, DIN1\_WIDTH-1);

din\_sig.write(din\_tmp);

}

};

template< int NUM\_STAGE,

int OPC\_WIDTH,

int DIN0\_WIDTH,

int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

SC\_MODULE (CFUMOBinaryOp)

{

public:

sc\_in< sc\_lv<OPC\_WIDTH> > opcode;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

AESLCFUMOComp<NUM\_STAGE, OPC\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\*

mMCModule;

SC\_CTOR( CFUMOBinaryOp ) {

mMCModule =

new AESLCFUMOComp

<NUM\_STAGE, OPC\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>("mb\_U");

mMCModule->opcode(opcode);

mMCModule->din(din\_sig);

mMCModule->dout(dout);

SC\_METHOD(IOConnection);

sensitive << din0 << din1 << opcode ;

}

void setComputeCore(FUComputeCore<OPC\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\* core) {

mMCModule->setComputeCore(core);

}

sc\_signal< sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> > din\_sig;

void IOConnection() {

sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> din\_tmp;

din\_tmp.range(0, DIN0\_WIDTH-1) =

din0.read().range(0, DIN0\_WIDTH-1);

din\_tmp.range(DIN0\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH-1) =

din1.read().range(0, DIN1\_WIDTH-1);

din\_sig.write(din\_tmp);

}

};

template< int NUM\_STAGE,

int OPC\_WIDTH,

int DIN0\_WIDTH,

int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

SC\_MODULE (FUMOBinaryOp)

{

public:

sc\_in< bool > clk;

sc\_in< sc\_logic > reset;

sc\_in< sc\_logic > ce;

sc\_in< sc\_lv<OPC\_WIDTH> > opcode;

sc\_in< sc\_lv<DIN0\_WIDTH> > din0;

sc\_in< sc\_lv<DIN1\_WIDTH> > din1;

sc\_out< sc\_lv<DOUT\_WIDTH> > dout;

AESLFUMultiCycle<NUM\_STAGE, OPC\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\*

mMCModule;

SC\_CTOR( FUMOBinaryOp ) {

mMCModule =

new AESLFUMultiCycle

<NUM\_STAGE, OPC\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>("mb\_U");

mMCModule->opcode(opcode);

mMCModule->din(din\_sig);

mMCModule->dout(dout);

mMCModule->clk(clk);

mMCModule->reset(reset);

mMCModule->ce(ce);

SC\_METHOD(IOConnection);

sensitive << din0 << din1 << opcode << clk;

}

void setComputeCore(FUComputeCore<OPC\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH, DOUT\_WIDTH>\* core) {

mMCModule->setComputeCore(core);

}

sc\_signal< sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> > din\_sig;

void IOConnection() {

sc\_lv<DIN0\_WIDTH+DIN1\_WIDTH> din\_tmp;

din\_tmp.range(0, DIN0\_WIDTH-1) =

din0.read().range(0, DIN0\_WIDTH-1);

din\_tmp.range(DIN0\_WIDTH, DIN0\_WIDTH+DIN1\_WIDTH-1) =

din1.read().range(0, DIN1\_WIDTH-1);

din\_sig.write(din\_tmp);

}

};

template< bool SIGNED,

int DIN0\_W,

int DIN1\_W=DIN0\_W,

int DOUT\_W=DIN0\_W>

struct IntArithCoreDivRem :

public FUComputeCore<2, DIN0\_W+DIN1\_W, DOUT\_W>

{

virtual sc\_lv<DOUT\_W>

compute(const sc\_lv<2>& opcode, const sc\_lv<DIN0\_W+DIN1\_W>& in) {

sc\_lv<DIN0\_W> in0;

sc\_lv<DIN1\_W> in1;

(in1, in0) = in;

sc\_lv<DOUT\_W> undef;

if (!opcode.is\_01() || !in.is\_01()) return undef;

bool opc = opcode[0].to\_bool();

if (!SIGNED) {

unsigned long long val0 = in0.to\_uint64();

unsigned long long val1 = in1.to\_uint64();

if (val1 == 0) return undef;

sc\_biguint<DIN0\_W> bval0 = sc\_biguint<DIN0\_W>(in0);

sc\_biguint<DIN1\_W> bval1 = sc\_biguint<DIN1\_W>(in1);

sc\_biguint<DOUT\_W> boutv = opc ? (bval0 % bval1) : (bval0 / bval1);

return sc\_lv<DOUT\_W>(boutv);

} else {

long long val0 = in0.to\_int64();

long long val1 = in1.to\_int64();

if (val1 == 0) return undef;

sc\_bigint<DIN0\_W> bval0 = sc\_bigint<DIN0\_W>(in0);

sc\_bigint<DIN1\_W> bval1 = sc\_bigint<DIN1\_W>(in1);

sc\_bigint<DOUT\_W> boutv = opc ? (bval0 % bval1) : (bval0 / bval1);

return sc\_lv<DOUT\_W>(boutv);

}

}

};

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

// Integer UDIV/UREM component.

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

template<int ID,

int NUM\_STAGE,

int DIN0\_W,

int DIN1\_W,

int DOUT\_W>

struct ACMP\_udivurem\_comb :

public CFUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>

{

ACMP\_udivurem\_comb(const char\* mname ) :

CFUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>(mname) {

setComputeCore(

new

IntArithCoreDivRem<false, DIN0\_W, DIN1\_W, DOUT\_W>());

}

};

template<int ID,

int NUM\_STAGE,

int DIN0\_W,

int DIN1\_W,

int DOUT\_W>

struct ACMP\_udivurem :

public FUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>

{

ACMP\_udivurem(const char\* mname ) :

FUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>(mname) {

setComputeCore(

new

IntArithCoreDivRem<false, DIN0\_W, DIN1\_W, DOUT\_W>());

}

};

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

// Integer SDIV/SREM component.

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

template<int ID,

int NUM\_STAGE,

int DIN0\_W,

int DIN1\_W,

int DOUT\_W>

struct ACMP\_sdivsrem\_comb :

public CFUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>

{

ACMP\_sdivsrem\_comb(const char\* mname ) :

CFUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>(mname) {

setComputeCore(

new IntArithCoreDivRem<true, DIN0\_W, DIN1\_W, DOUT\_W>());

}

};

template<int ID,

int NUM\_STAGE,

int DIN0\_W,

int DIN1\_W,

int DOUT\_W>

struct ACMP\_sdivsrem :

public FUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>

{

ACMP\_sdivsrem(const char\* mname ) :

FUMOBinaryOp<NUM\_STAGE, 2, DIN0\_W, DIN1\_W, DOUT\_W>(mname) {

setComputeCore(

new IntArithCoreDivRem<true, DIN0\_W, DIN1\_W, DOUT\_W>());

}

};

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

// Integer UDIV component.

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

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_udiv\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_udiv\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

unsigned long long ival1 = (in1).to\_uint64();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DIN0\_WIDTH> bval0 = sc\_biguint<DIN0\_WIDTH>(in0);

sc\_biguint<DIN1\_WIDTH> bval1 = sc\_biguint<DIN1\_WIDTH>(in1);

sc\_biguint<DOUT\_WIDTH> boutv = bval0 / bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_udiv :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_udiv(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

// assert(DIN0\_WIDTH == DOUT\_WIDTH);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

unsigned long long ival1 = (in1).to\_uint64();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DIN0\_WIDTH> bval0 = sc\_biguint<DIN0\_WIDTH>(in0);

sc\_biguint<DIN1\_WIDTH> bval1 = sc\_biguint<DIN1\_WIDTH>(in1);

sc\_biguint<DOUT\_WIDTH> boutv = bval0 / bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_udiv\_seq :

public AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_udiv\_seq(const char\* mname) :

AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

// assert(DIN0\_WIDTH == DOUT\_WIDTH);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

unsigned long long ival1 = (in1).to\_uint64();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DIN0\_WIDTH> bval0 = sc\_biguint<DIN0\_WIDTH>(in0);

sc\_biguint<DIN1\_WIDTH> bval1 = sc\_biguint<DIN1\_WIDTH>(in1);

sc\_biguint<DOUT\_WIDTH> boutv = bval0 / bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_sdiv\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_sdiv\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

int ival1 = (in1).to\_int();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DIN0\_WIDTH> bval0 = sc\_bigint<DIN0\_WIDTH>(in0);

sc\_bigint<DIN1\_WIDTH> bval1 = sc\_bigint<DIN1\_WIDTH>(in1);

sc\_bigint<DOUT\_WIDTH> boutv = bval0 / bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_sdiv :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_sdiv(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

//assert(DIN0\_WIDTH == DOUT\_WIDTH);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

int ival1 = (in1).to\_int();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DIN0\_WIDTH> bval0 = sc\_bigint<DIN0\_WIDTH>(in0);

sc\_bigint<DIN1\_WIDTH> bval1 = sc\_bigint<DIN1\_WIDTH>(in1);

sc\_bigint<DOUT\_WIDTH> boutv = bval0 / bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_sdiv\_seq :

public AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_sdiv\_seq(const char\* mname) :

AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

//assert(DIN0\_WIDTH == DOUT\_WIDTH);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

int ival1 = (in1).to\_int();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DIN0\_WIDTH> bval0 = sc\_bigint<DIN0\_WIDTH>(in0);

sc\_bigint<DIN1\_WIDTH> bval1 = sc\_bigint<DIN1\_WIDTH>(in1);

sc\_bigint<DOUT\_WIDTH> boutv = bval0 / bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int DIN0\_WIDTH, int DIN1\_WIDTH, int DOUT\_WIDTH>

sc\_lv<DOUT\_WIDTH> compute\_mul\_ss(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DOUT\_WIDTH> outi = sc\_bigint<DIN0\_WIDTH>(in0) \* sc\_bigint<DIN1\_WIDTH>(in1);

return sc\_lv<DOUT\_WIDTH>(outi);

}

template<int DIN0\_WIDTH, int DIN1\_WIDTH, int DOUT\_WIDTH>

sc\_lv<DOUT\_WIDTH> compute\_mul\_su(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DOUT\_WIDTH> outi = sc\_bigint<DIN0\_WIDTH>(in0) \* sc\_biguint<DIN1\_WIDTH>(in1);

return sc\_lv<DOUT\_WIDTH>(outi);

}

template<int DIN0\_WIDTH, int DIN1\_WIDTH, int DOUT\_WIDTH>

sc\_lv<DOUT\_WIDTH> compute\_mul\_us(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DOUT\_WIDTH> outi = sc\_biguint<DIN0\_WIDTH>(in0) \* sc\_bigint<DIN1\_WIDTH>(in1);

return sc\_lv<DOUT\_WIDTH>(outi);

}

template<int DIN0\_WIDTH, int DIN1\_WIDTH, int DOUT\_WIDTH>

sc\_lv<DOUT\_WIDTH> compute\_mul\_uu(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1)

{

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DOUT\_WIDTH> outi = sc\_biguint<DIN0\_WIDTH>(in0) \* sc\_biguint<DIN1\_WIDTH>(in1);

return sc\_lv<DOUT\_WIDTH>(outi);

}

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_mul\_ss :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_mul\_ss(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_ss<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_mul\_su :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_mul\_su(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_su<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_mul\_us :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_mul\_us(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_us<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_mul\_uu :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_mul\_uu(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_uu<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_smul\_ss :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_smul\_ss(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_ss<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_smul\_su :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_smul\_su(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_su<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_smul\_us :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_smul\_us(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_us<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_smul\_uu :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_smul\_uu(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

return compute\_mul\_uu<DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(in0, in1);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_add\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_bigint<DIN0\_WIDTH> OpDataTypeT;

ACMP\_add\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT ival0 = OpDataTypeT(in0);

OpDataTypeT ival1 = OpDataTypeT(in1);

OpDataTypeT outi = ival0 + ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_add :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_bigint<DIN0\_WIDTH> OpDataTypeT;

ACMP\_add(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT ival0 = OpDataTypeT(in0);

OpDataTypeT ival1 = OpDataTypeT(in1);

OpDataTypeT outi = ival0 + ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_sub\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_bigint<DIN0\_WIDTH> OpDataTypeT;

ACMP\_sub\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT ival0 = OpDataTypeT(in0);

OpDataTypeT ival1 = OpDataTypeT(in1);

OpDataTypeT outi = ival0 - ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_sub :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_bigint<DIN0\_WIDTH> OpDataTypeT;

ACMP\_sub(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT ival0 = OpDataTypeT(in0);

OpDataTypeT ival1 = OpDataTypeT(in1);

OpDataTypeT outi = ival0 - ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_urem\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_urem\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 32);

// assert(DIN1\_WIDTH == DOUT\_WIDTH);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

unsigned int ival0 = (in0).to\_uint();

unsigned int ival1 = (in1).to\_uint();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DIN0\_WIDTH> bval0 = sc\_biguint<DIN0\_WIDTH>(in0);

sc\_biguint<DIN1\_WIDTH> bval1 = sc\_biguint<DIN1\_WIDTH>(in1);

sc\_biguint<DOUT\_WIDTH> boutv = bval0 % bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_urem :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_urem(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

unsigned int ival0 = (in0).to\_uint();

unsigned int ival1 = (in1).to\_uint();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DIN0\_WIDTH> bval0 = sc\_biguint<DIN0\_WIDTH>(in0);

sc\_biguint<DIN1\_WIDTH> bval1 = sc\_biguint<DIN1\_WIDTH>(in1);

sc\_biguint<DOUT\_WIDTH> boutv = bval0 % bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_urem\_seq :

public AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_urem\_seq(const char\* mname) :

AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

unsigned int ival0 = (in0).to\_uint();

unsigned int ival1 = (in1).to\_uint();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_biguint<DIN0\_WIDTH> bval0 = sc\_biguint<DIN0\_WIDTH>(in0);

sc\_biguint<DIN1\_WIDTH> bval1 = sc\_biguint<DIN1\_WIDTH>(in1);

sc\_biguint<DOUT\_WIDTH> boutv = bval0 % bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_srem\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_srem\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

int ival1 = (in1).to\_int();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DIN0\_WIDTH> bval0 = sc\_bigint<DIN0\_WIDTH>(in0);

sc\_bigint<DIN1\_WIDTH> bval1 = sc\_bigint<DIN1\_WIDTH>(in1);

sc\_bigint<DOUT\_WIDTH> boutv = bval0 % bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_srem :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_srem(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

int ival1 = (in1).to\_int();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DIN0\_WIDTH> bval0 = sc\_bigint<DIN0\_WIDTH>(in0);

sc\_bigint<DIN1\_WIDTH> bval1 = sc\_bigint<DIN1\_WIDTH>(in1);

sc\_bigint<DOUT\_WIDTH> boutv = bval0 % bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_srem\_seq :

public AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>

{

public:

ACMP\_srem\_seq(const char\* mname) :

AESLFUComp\_seq<NUM\_STAGE, DIN0\_WIDTH, DIN1\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

//assert(DIN0\_WIDTH <= 64);

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN1\_WIDTH>& in1) {

if (!in0.is\_01() || !in1.is\_01()) return sc\_lv<DOUT\_WIDTH>();

int ival1 = (in1).to\_int();

if (ival1 == 0)

return sc\_lv<DOUT\_WIDTH>();

sc\_bigint<DIN0\_WIDTH> bval0 = sc\_bigint<DIN0\_WIDTH>(in0);

sc\_bigint<DIN1\_WIDTH> bval1 = sc\_bigint<DIN1\_WIDTH>(in1);

sc\_bigint<DOUT\_WIDTH> boutv = bval0 % bval1;

return sc\_lv<DOUT\_WIDTH>(boutv);

}

};

// shl

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_shl\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_biguint<DIN0\_WIDTH> OpDataTypeT1;

typedef sc\_biguint<DIN1\_WIDTH> OpDataTypeT2;

ACMP\_shl\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN0\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT1 ival0 = OpDataTypeT1(in0);

OpDataTypeT2 ival1 = sc\_bv<DIN1\_WIDTH>(in1.range(DIN1\_WIDTH-1, 0));

if(ival1>DIN0\_WIDTH) return sc\_lv<DOUT\_WIDTH>(0);

OpDataTypeT1 outi = ival0 << ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_shl :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_biguint<DIN0\_WIDTH> OpDataTypeT1;

typedef sc\_biguint<DIN1\_WIDTH> OpDataTypeT2;

ACMP\_shl(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN0\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT1 ival0 = OpDataTypeT1(in0);

OpDataTypeT2 ival1 = sc\_bv<DIN1\_WIDTH>(in1.range(DIN1\_WIDTH-1, 0));

if(ival1>DIN0\_WIDTH) return sc\_lv<DOUT\_WIDTH>(0);

OpDataTypeT1 outi = ival0 << ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

// lshr

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_lshr\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_biguint<DIN0\_WIDTH> OpDataTypeT1;

typedef sc\_biguint<DIN1\_WIDTH> OpDataTypeT2;

ACMP\_lshr\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN0\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT1 ival0 = OpDataTypeT1(in0);

OpDataTypeT2 ival1 = sc\_bv<DIN1\_WIDTH>(in1.range(DIN1\_WIDTH-1, 0));

OpDataTypeT1 outi = ival0 >> ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_lshr :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_biguint<DIN0\_WIDTH> OpDataTypeT1;

typedef sc\_biguint<DIN1\_WIDTH> OpDataTypeT2;

ACMP\_lshr(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN0\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT1 ival0 = OpDataTypeT1(in0);

OpDataTypeT2 ival1 = sc\_bv<DIN1\_WIDTH>(in1.range(DIN1\_WIDTH-1, 0));

OpDataTypeT1 outi = ival0 >> ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

// ashr

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_ashr\_comb :

public AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_bigint<DIN0\_WIDTH> OpDataTypeT1;

typedef sc\_biguint<DIN1\_WIDTH> OpDataTypeT2;

ACMP\_ashr\_comb(const char\* mname) :

AESLCFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN0\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT1 ival0 = OpDataTypeT1(in0);

OpDataTypeT2 ival1 = sc\_bv<DIN1\_WIDTH>(in1.range(DIN1\_WIDTH-1, 0));

OpDataTypeT1 outi = ival0 >> ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

template<int ID, int NUM\_STAGE,

int DIN0\_WIDTH, int DIN1\_WIDTH=DIN0\_WIDTH,

int DOUT\_WIDTH=DIN0\_WIDTH>

struct ACMP\_ashr :

public AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>

{

public:

typedef sc\_bigint<DIN0\_WIDTH> OpDataTypeT1;

typedef sc\_biguint<DIN1\_WIDTH> OpDataTypeT2;

ACMP\_ashr(const char\* mname) :

AESLFUComp<NUM\_STAGE, DIN0\_WIDTH, DIN0\_WIDTH, DOUT\_WIDTH>(mname) {

sanity\_check();

}

virtual void sanity\_check() {

}

virtual sc\_lv<DOUT\_WIDTH>

compute(const sc\_lv<DIN0\_WIDTH>& in0, const sc\_lv<DIN0\_WIDTH>& in1) {

if(in0.is\_01() && in1.is\_01()) {

OpDataTypeT1 ival0 = OpDataTypeT1(in0);

OpDataTypeT2 ival1 = sc\_bv<DIN1\_WIDTH>(in1.range(DIN1\_WIDTH-1, 0));

OpDataTypeT1 outi = ival0 >> ival1;

return sc\_lv<DOUT\_WIDTH>(outi);

}

else

return sc\_lv<DOUT\_WIDTH>(SC\_LOGIC\_X);

}

};

#endif

// 67d7842dbbe25473c3c32b93c0da8047785f30d78e8a024de1b57352245f9689