0001: //ltloop - link training loop test

0002:

0003: #include <stdio.h>

0004: #include <stdlib.h>

0005: #include <stdint.h>

0006: #include <sys/time.h>

0007: #include <dlfcn.h>

0008: #include <string.h>

0009: #include <unistd.h>

0010: #include <itpdriver/itp\_driver.h>

0011: #include <itpdriver/itp\_driver1.h>

0012: #include <itpdriver/itp\_vcu.h>

0013: #include <itpdriver/defines.h>

0014: #include <itpdriver/itp\_iosf.h>

0015: #include <time.h>

0016:

0017:

0018: //information for port2bdf routine

0019: #define MAXPORT 17

0020: //port 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

0021: uint32\_t pbus[] = {0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4 }; // Bus numbers: DMI=0

0022: uint32\_t pdev[] = {3, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5, 2, 3, 4, 5 };

0023: uint32\_t pfun[] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

0024: uint32\_t root[] = {0, 1, 1, 1, 1, 5, 5, 5, 5, 9, 9, 9, 9,13,13,13,13 }; //root port number

0025: #define PCIEBASE 0x80000000

0026:

0027: //Register offsets

0028: #define REG\_LNKSTS 0x52

0029: #define REG\_DMI\_LNKSTS 0x1B2

0030: #define REG\_LNKCON 0x50

0031: #define REG\_DMI\_LNKCON 0x1B0

0032: #define REG\_LNKCON2 0x70

0033: #define REG\_DMI\_LNKCON2 0x1C0

0034: #define REG\_LNKCAP 0x4C

0035:

0036: #define REG\_LTRCON 0x11C

0037: #define REG\_CORERRSTS 0x110 //"ERRCORSTS" correctable error status

0038: #define REG\_UNCERRSTS 0x104 //"ERRUNCSTS" uncorrectable error status

0039:

0040: #define REG\_SECBUS 0x19

0041:

0042: #define REG\_AERCAPHDR 0x100 //Advanced Error Reporting Extended Capability Header

0043: #define REG\_CPUBUSNO 0x104

0044: #define REG\_CPUBUSNO1 0x108

0045: #define REG\_CPUBUSNO2 0x10A

0046: #define REG\_CPUBUSNO\_VALID 0x110

0047: #define REG\_SOCKET\_BUS\_RANGE 0x114

0048:

0049:

0050: //option values

0051: int m\_port = 0;

0052: int m\_loops = 1;

0053: int m\_type = 1;

0054: int m\_socket = 1;

0055: bool m\_readDSC = false;

0056: bool m\_overview = false;

0057: bool m\_pciScan = false;

0058: int m\_forceSpeed = 0;

0059:

0060: //Notes on m\_socket and m\_peciCPU

0061: //m\_socket is always the option chosen by the user (or the default)

0062: //m\_peciCPU is used by the read/write register routines to know which

0063: //CPU to access. Note that it is used by the "Mem" routines to

0064: //get the correct bus number (per cpu)

0065: //We need both because we need to always enumerate busses on all CPUs

0066: //And for TOPO, which always does all CPUs

0067: //Also note that m\_socket is the "user", number; 1 is the first CPU

0068: //m\_peciCPU is internal, and 0 is the first CPU

0069: int m\_peciCPU = 1;

0070:

0071: //Information from CPUBUSNO registers

0072: #define MAX\_BUSSES 12

0073: #define MAX\_CPUS 4

0074: bool m\_socket\_bus\_range\_valid = false;

0075: int m\_socket\_segment;

0076: int m\_socket\_start\_bus;

0077: int m\_socket\_end\_bus;

0078: int m\_socket\_busses[MAX\_CPUS][MAX\_BUSSES];

0079: int m\_bus0; //Bus to start with when finding busses per socket

0080:

0081: //Given a port number, we find the proper bus, device, and function to use

0082: //This will be a PECI bus [5..0]

0083: void port2bdf(uint32\_t port, uint32\_t \*bus, uint32\_t \*dev, uint32\_t \*fun)

0084: {

0085: //Translates a port number(input) to bus/device/function (outputs)

0086:

0087: if ((port >= MAXPORT) || (port < 0))

0088: {

0089: return;

0090: }

0091:

0092: \*bus = pbus[port];

0093: \*dev = pdev[port];

0094: \*fun = pfun[port];

0095:

0096: } // port2bdf

0097:

0098:

0099: //Returns the bus number to use with memmory based register reads given a PECI bus

0100: //Requires that getBusNumbers routine has setup the data

0101: bool getMemBus(uint32\_t bus, uint32\_t \*membus)

0102: {

0103: //Not a fool proof check, but at least let's take a look:

0104: if (!m\_socket\_bus\_range\_valid)

0105: {

0106: printf("Bus range has not been acquired from CPU prior to accessing memory based bus numbers.\n");

0107: return false;

0108: }

0109:

0110: if (bus >= MAX\_BUSSES)

0111: {

0112: printf("Bus %d is out of range", bus);

0113: return false;

0114: }

0115:

0116: if (m\_peciCPU >= MAX\_CPUS)

0117: {

0118: printf("PECI CPU %d is out of range.\n");

0119: return false;

0120: }

0121:

0122: \*membus = m\_socket\_busses[m\_peciCPU][bus];

0123: return true;

0124: }

0125:

0126: //For PECI access, the higher busses are remapped to fit in the smaller bus number allowed.

0127: //Bus 31 is mapped to bus 14, and bus 30 is mapped to bus 13.

0128: uint32\_t remapBus(uint32\_t bus)

0129: {

0130: if (bus == 31)

0131: {

0132: return 14;

0133: }

0134: if (bus == 30)

0135: {

0136: return 13;

0137: }

0138:

0139: return bus;

0140: }

0141:

0142: uint32\_t readRegisterPeci(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t offset, AI\_buswidth busWidth)

0143: {

0144: bool ret;

0145: uint32\_t regval = 0;

0146: uint8\_t pcireg[4];

0147: uint8\_t readLen;

0148: uint8\_t cc;

0149: int i;

0150:

0151: for (i=0; i<4; i++)

0152: {

0153: pcireg[i] = 0;

0154: }

0155: switch (busWidth)

0156: {

0157: case AI\_bw8: readLen = 1; break;

0158: case AI\_bw16: readLen = 2; break;

0159: case AI\_bw32: readLen = 4; break;

0160: default: readLen = 4; break;

0161: }

0162:

0163: bus = remapBus(bus);

0164:

0165: ret = ai\_mIOSFreadPCIConfigLocal(mHandle, m\_peciCPU, bus, dev, fun, offset,

0166: readLen, pcireg, &cc);

0167: if (!ret)

0168: {

0169: printf ("ERROR with readPCIConfigLocal\n");

0170: return 0xFFFFFFFF;

0171: }

0172: if (cc != 0x40)

0173: {

0174: // printf("Bad CC 0x%02hhx for readPCIConfigLocal, b:0x%04x d::0x%04x f::0x%04x o::0x%04x\n",

0175: // cc, bus, dev, fun, offset);

0176: return 0xFFFFFFFF;

0177: }

0178:

0179: regval = (pcireg[3] << 24) | (pcireg[2] << 16) | (pcireg[1] << 8) | pcireg[0];

0180: return regval;

0181:

0182: } // readRegisterPeci

0183:

0184:

0185: void writeRegisterPeci(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t offset, AI\_buswidth busWidth, uint32\_t newValue)

0186: {

0187: uint8\_t writeLen;

0188: uint8\_t cc;

0189: bool ret;

0190:

0191: switch (busWidth)

0192: {

0193: case AI\_bw8: writeLen = 1; break;

0194: case AI\_bw16: writeLen = 2; break;

0195: case AI\_bw32: writeLen = 4; break;

0196: default: writeLen = 4; break;

0197: }

0198:

0199: bus = remapBus(bus);

0200:

0201: ret = ai\_mIOSFwritePCIConfigLocal(mHandle, m\_peciCPU, bus, dev, fun, offset,

0202: writeLen, newValue, &cc);

0203: if (!ret)

0204: {

0205: printf ("ERROR with writePCIConfigLocal\n");

0206: return;

0207: }

0208: if (cc != 0x40)

0209: {

0210: // printf("Bad CC 0x%02hhx for writePCIConfigLocal, b:0x%04x d::0x%04x f::0x%04x o::0x%04x\n",

0211: // cc, bus, dev, fun, offset);

0212: return;

0213: }

0214:

0215: return;

0216:

0217: } // writeRegisterPeci

0218:

0219:

0220: //reads a register on a DownStream component (PECI)

0221: uint32\_t readRegisterDS(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t offset, AI\_buswidth busWidth)

0222: {

0223: int iError = 0;

0224: uint32\_t regval = 0;

0225: uint8\_t pcireg[4];

0226: uint8\_t cc;

0227: int i;

0228:

0229: for (i=0; i<4; i++)

0230: {

0231: pcireg[i] = 0;

0232: }

0233:

0234: bus = remapBus(bus);

0235:

0236: //On downstream components, read length is always 4 (32 bits)

0237: iError = ai\_mIOSFreadPCIConfig(mHandle, (m\_socket - CPU\_ZERO\_POS), bus, dev, fun, offset,

0238: pcireg, &cc);

0239: if (iError != AI\_SUCCESS)

0240: {

0241: printf ("ERROR with readPCIConfig\n");

0242: return 0xFFFFFFFF;

0243: }

0244: if (cc != 0x40)

0245: {

0246: // printf("Bad CC 0x%02hhx for readPCIConfig, b:0x%04x d::0x%04x f::0x%04x o::0x%04x\n",

0247: // cc, bus, dev, fun, offset);

0248: return 0xFFFFFFFF;

0249: }

0250:

0251: regval = (pcireg[3] << 24) | (pcireg[2] << 16) | (pcireg[1] << 8) | pcireg[0];

0252: return regval;

0253:

0254: } // readRegisterDS

0255:

0256:

0257: //Writes a register on a DownStream component (PECI)

0258: void writeRegisterDS(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t offset, AI\_buswidth busWidth, uint32\_t newValue)

0259: {

0260: uint8\_t writeLen;

0261: uint8\_t cc;

0262: int iError = 0;

0263:

0264: switch (busWidth)

0265: {

0266: case AI\_bw8: writeLen = 1; break;

0267: case AI\_bw16: writeLen = 2; break;

0268: case AI\_bw32: writeLen = 4; break;

0269: default: writeLen = 4; break;

0270: }

0271:

0272: bus = remapBus(bus);

0273:

0274: iError = ai\_mIOSFwritePCIConfig(mHandle, (m\_socket - CPU\_ZERO\_POS), bus, dev, fun, offset,

0275: writeLen, newValue, &cc);

0276: if (iError != AI\_SUCCESS)

0277: {

0278: printf ("ERROR with writePCIConfig\n");

0279: return;

0280: }

0281: if (cc != 0x40)

0282: {

0283: // printf("Bad CC 0x%02hhx for writePCIConfig, b:0x%04x d::0x%04x f::0x%04x o::0x%04x\n",

0284: // cc, bus, dev, fun, offset);

0285: return;

0286: }

0287:

0288: return;

0289:

0290: } // writeRegisterDS

0291:

0292:

0293: uint32\_t readRegisterMem(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t offset, AI\_buswidth busWidth)

0294: {

0295: uint32\_t addr;

0296: int iError = 0;

0297: uint32\_t regval = 0;

0298:

0299:

0300: addr = PCIEBASE + (bus << 20) + (dev << 15) + (fun << 12) + offset;

0301:

0302: iError = ai\_mReadMemory(mHandle, addr, &regval, busWidth);

0303: if (iError != AI\_SUCCESS)

0304: {

0305: printf ("Read Memory ERROR: %s\n" , ai\_ErrorToString(iError));

0306: return 0;

0307: }

0308:

0309: return regval;

0310:

0311: }

0312:

0313: void writeRegisterMem(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t offset, AI\_buswidth busWidth, uint32\_t newValue)

0314: {

0315:

0316: uint32\_t addr;

0317: int iError = 0;

0318:

0319:

0320: addr = PCIEBASE + (bus << 20) + (dev << 15) + (fun << 12) + offset;

0321:

0322: iError = ai\_mWriteMemory(mHandle, addr, &newValue, busWidth);

0323: if (iError != AI\_SUCCESS)

0324: {

0325: printf ("Write Memory ERROR: %s\n" , ai\_ErrorToString(iError));

0326: return;

0327: }

0328:

0329: return;

0330:

0331: } // writeRegister

0332:

0333:

0334: //Read the registers and display (and save) the bus numbers that the CPU has allocated

0335: void getBusNumbers(int mHandle, int CPU)

0336: {

0337: uint32\_t reg;

0338: uint32\_t valid\_busses;

0339: int outpos;

0340: int inpos;

0341: int shifty;

0342:

0343: //PECI can't read bus 0, dev 2; so use memory reads instead

0344:

0345: reg = readRegisterMem(mHandle, m\_bus0, 2, 0, REG\_SOCKET\_BUS\_RANGE, AI\_bw32);

0346:

0347: if ((reg & 0x80000000) != 0)

0348: {

0349: //Bit 31 set indicates data is valid

0350: m\_socket\_bus\_range\_valid = true;

0351: m\_socket\_segment = (reg >> 16) & 0xFF;

0352: m\_socket\_end\_bus = (reg >> 8) & 0xFF;

0353: m\_socket\_start\_bus = reg & 0xFF;

0354: printf("Socket %d info: segment: 0x%02x start bus: 0x%02x end bus: 0x%02x\n",

0355: CPU + CPU\_ZERO\_POS, m\_socket\_segment, m\_socket\_start\_bus, m\_socket\_end\_bus);

0356: }

0357: else

0358: {

0359: printf("SOCKET\_BUS\_RANGE indicates socket bus range data is NOT valid.\n");

0360: }

0361:

0362: //Read the registers for bus numbers.

0363: //First read is bit array of valid busses

0364: //Actual numbers of busses come from register reads below

0365: //Note that input is not necessarily contiguous, it may have "holes" in the bus numbers used.

0366: valid\_busses = readRegisterMem(mHandle, m\_bus0, 2, 0, REG\_CPUBUSNO\_VALID, AI\_bw32);

0367: //printf("Valid bus flags: 0x%08x\n", valid\_busses);

0368: outpos = 0; //The bus index for the saved array of bus numbers (output)

0369: inpos = 0; //bus index for the valid bits we just read (input)

0370: shifty = 0;

0371: printf("Valid busses: ");

0372: for (inpos = 0; inpos < MAX\_BUSSES; inpos++)

0373: {

0374: switch (inpos)

0375: {

0376: case 0:

0377: reg = readRegisterMem(mHandle, m\_bus0, 2, 0, REG\_CPUBUSNO, AI\_bw32);

0378: //printf("CPU BUS NO 0 0x%08x\n", reg);

0379: shifty = 0;

0380: break;

0381: case 4:

0382: reg = readRegisterMem(mHandle, m\_bus0, 2, 0, REG\_CPUBUSNO1, AI\_bw32);

0383: //printf("CPU BUS NO 1 0x%08x\n", reg);

0384: shifty = 0;

0385: break;

0386: case 8:

0387: reg = readRegisterMem(mHandle, m\_bus0, 2, 0, REG\_CPUBUSNO2, AI\_bw32);

0388: //printf("CPU BUS NO 2 0x%08x\n", reg);

0389: shifty = 0;

0390: break;

0391: }

0392: if (((valid\_busses >> inpos) & 1) != 0)

0393: {

0394: //The valid bit is set for this bus, save it and print it

0395: m\_socket\_busses[CPU][outpos] = (reg >> shifty) & 0xFF;

0396: printf("0x%02x ", m\_socket\_busses[CPU][outpos]);

0397: outpos++;

0398: }

0399: shifty += 8; //Advance to the next input byte position, whether we used this one or not

0400: }

0401:

0402: printf("\n");

0403:

0404: m\_bus0 = m\_socket\_end\_bus + 1; //Next socket, will start with bus just past this socket

0405: }

0406:

0407:

0408:

0409: void devUnhide(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun, uint32\_t reg)

0410: {

0411: uint32\_t devhide;

0412:

0413: devhide = readRegisterPeci(mHandle, bus, dev, fun, reg, AI\_bw32);

0414: if ((devhide & 0xc0) != 0)

0415: {

0416: writeRegisterPeci(mHandle, bus, dev, fun, reg, AI\_bw32, (devhide & 0xffffFF3F));

0417: }

0418: } // devUnhide

0419:

0420:

0421: void prepTarget(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun)

0422: {

0423:

0424: uint32\_t lnksel;

0425: uint32\_t sublink;

0426: uint32\_t logctrl0;

0427: uint32\_t logctrl1;

0428:

0429: int rootport;

0430:

0431: //Emulate what the cscript does to prepare the target for the test.

0432: //DEVHIDE[0-7]

0433: //ICX: bus moved from bus 0 to bus 30; and device moved from 8 to 0

0434: devUnhide(mHandle, 30, 0, 2, 0x144); //ICX: devhide0 moved from 0x104 to 0x144

0435: devUnhide(mHandle, 30, 0, 2, 0x108);

0436: devUnhide(mHandle, 30, 0, 2, 0x10C);

0437: devUnhide(mHandle, 30, 0, 2, 0x110);

0438: devUnhide(mHandle, 30, 0, 2, 0x114);

0439: devUnhide(mHandle, 30, 0, 2, 0x118);

0440: devUnhide(mHandle, 30, 0, 2, 0x11C);

0441: devUnhide(mHandle, 30, 0, 2, 0x120);

0442: } // prepTarget

0443:

0444:

0445: void usage(void)

0446: {

0447: printf("\nUsage:\n");

0448: printf("-l<n> loop count\n");

0449: printf("-p<n> port to test, n=0 to %d\n", MAXPORT);

0450: printf("-t<n> loop type:\n");

0451: printf(" n=1 basic retrain loop\n");

0452: printf(" n=6 retrain with speed change\n");

0453: printf("-s<n> socket n=1|2\n");

0454: printf("-d check for errors on downstream component, if available\n");

0455: printf("-o print overview of ports (PCIe topology) - preempts other options.\n");

0456: printf("-f<n> attempt to force speed to n\n");

0457: printf("-c scan PCI\n");

0458: } // usage

0459:

0460:

0461: int parseArgs(int argc, char \*\*argv)

0462: {

0463: int c;

0464: int retval = 0;

0465: while ((c = getopt (argc, argv, "l:p:t:s:f:cdbo?h")) != -1)

0466: {

0467: switch(c)

0468: {

0469: case 'p':

0470: //printf("Option p: %s\n", optarg);

0471: m\_port = atoi(optarg);

0472: if ((m\_port < 0) || (m\_port >= MAXPORT))

0473: {

0474: printf("Invalid port value: %d \n", m\_port);

0475: retval = -1;

0476: m\_port = 0;

0477: }

0478: break;

0479:

0480: case 'l':

0481: m\_loops = atoi(optarg);

0482: if (m\_loops < 1)

0483: {

0484: printf("Invalid number of loops, must be > 0.\n");

0485: retval = -1;

0486: m\_loops = 1;

0487: }

0488: break;

0489:

0490: case 's':

0491: m\_socket = atoi(optarg);

0492: //Will check socket # in main after we get system topology and learn # of CPUs

0493: break;

0494:

0495: case 't':

0496: m\_type = atoi(optarg);

0497: if ((m\_type != 1) && (m\_type != 6))

0498: {

0499: printf("Invalid test type, only type 1 and type 6 are supported.\n");

0500: m\_type = 1;

0501: retval = -1;

0502: }

0503: break;

0504:

0505: case 'd':

0506: m\_readDSC = true;

0507: break;

0508:

0509: case 'b':

0510: ai\_mIOSFdebugPrint(true);

0511: printf("Debug printouts turned on\n");

0512: break;

0513:

0514: case 'o':

0515: m\_overview = true;

0516: break;

0517:

0518: case 'f':

0519: m\_forceSpeed = atoi(optarg);

0520: if ((m\_forceSpeed > 4) || (m\_forceSpeed < 1))

0521: {

0522: printf("Invalid speed, must be between 1 and 4\n");

0523: retval = 1;

0524: m\_forceSpeed = 0;

0525: }

0526: break;

0527: case 'c':

0528: m\_pciScan = true;

0529: break;

0530: case '?':

0531: case 'h':

0532: retval = 1; //Note: caller will treat as error and print usage()

0533: break;

0534:

0535: } // switch...

0536: } //while...

0537:

0538: //Further checks on params

0539: //Can't check downstream component on DMI (port 0)

0540: if ((m\_port == 0) && (m\_readDSC))

0541: {

0542: m\_readDSC = false;

0543: printf("Unable to check DSC errors on DMI.\n");

0544: //We allow test to continue in this case, as it is probably what user would do if we issued an error.

0545: }

0546:

0547: return retval;

0548:

0549: } // parseArgs

0550:

0551:

0552: void lt\_loop(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun)

0553: {

0554: //major test; loop on link training.

0555:

0556: int i;

0557: int j;

0558: int widthIndex;

0559: uint32\_t lnksts;

0560: uint32\_t lnkcon;

0561: uint32\_t lnkcon2;

0562: uint32\_t dbg0;

0563: uint32\_t startWidth;

0564: uint32\_t startSpeed;

0565: uint32\_t currentWidth;

0566: uint32\_t currentSpeed;

0567: uint32\_t expectedSpeed;

0568: uint32\_t speed1;

0569: uint32\_t speed2;

0570: uint32\_t errs;

0571: uint32\_t secbus;

0572: uint32\_t caphdr;

0573: uint32\_t nextcap;

0574: uint32\_t reg\_dscuncerr;

0575: uint32\_t reg\_dsccorerr;

0576: uint32\_t reg\_lnksts;

0577: uint32\_t reg\_lnkcon;

0578: uint32\_t reg\_lnkcon2;

0579: uint32\_t membus;

0580:

0581: bool foundit;

0582: bool readDSC;

0583: int speedErrors = 0; //Count of link is at different speed after retrain

0584: int widthErrors = 0; //Count of link is different width after retrain

0585: int noTrains = 0; //Count of link failed to (re)train

0586: int uncorErrors = 0; //Count of UNcorrectable errors

0587: int corErrors = 0; //Count of correctable errors

0588: int dscUncorErrors = 0; //Count of UNcorrectable errors at down stream component

0589: int dscCorErrors = 0; //Count of UNcorrectable errors at down stream component

0590:

0591: //Counters for each speed/width found when testing; used to display results at end.

0592: int widthSpeedCount[5][5]; //[width][speed]

0593: for (i=0; i<5; i++)

0594: for (j=0; j<5; j++)

0595: widthSpeedCount[i][j] = 0;

0596:

0597: //Correct for register offsets because DMI doesn't follow pattern of all other ports.

0598: if (m\_port == 0)

0599: {

0600: reg\_lnksts = REG\_DMI\_LNKSTS;

0601: reg\_lnkcon = REG\_DMI\_LNKCON;

0602: reg\_lnkcon2 = REG\_DMI\_LNKCON2;

0603: }

0604: else

0605: {

0606: reg\_lnksts = REG\_LNKSTS;

0607: reg\_lnkcon = REG\_LNKCON;

0608: reg\_lnkcon2 = REG\_LNKCON2;

0609: }

0610:

0611: if (!getMemBus(bus, &membus))

0612: {

0613: printf("Unable to find proper memory bus for PECI bus %d", bus);

0614: return;

0615: }

0616:

0617: printf("\nPreparing target to test port %d\n", m\_port);

0618: prepTarget(mHandle, bus, dev, fun);

0619:

0620: //read LNKSTS and get link active, speed, width

0621: lnksts = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnksts, AI\_bw16);

0622: //Check for all ones; indicates we are not reading anything valid.

0623: if (lnksts == 0xFFFF)

0624: {

0625: printf("Link status is all ones; aborting test.\n");

0626: return;

0627: }

0628: //check to make sure the link is active before we begin. May be inactive due to not connected on target.

0629: if ((lnksts & 0x2000) == 0)

0630: {

0631: printf("Link is not active on startup, aborting test.\n");

0632: return;

0633: }

0634:

0635: startWidth = (lnksts >> 4) & 0x3F;

0636: startSpeed = lnksts & 0xF;

0637:

0638: printf ("Initial conditions for port %d speed = %d, width = %d\n", m\_port, startSpeed, startWidth);

0639:

0640: if (m\_type == 6)

0641: {

0642: //Type 6 is speed change every loop (initial speed <==> gen 1)

0643: speed1 = startSpeed; //speed for odd iterations of loop

0644: speed2 = 1; //speed for even iterations of loop

0645: currentSpeed = startSpeed;

0646: }

0647: else

0648: {

0649: speed1 = startSpeed;

0650: speed2 = startSpeed;

0651: currentSpeed = startSpeed;

0652: }

0653:

0654: readDSC = false;

0655: if (m\_readDSC)

0656: {

0657: //Get the secondary bus number. We'll need it to read the DSC (down stream component) errors.

0658: secbus = readRegisterPeci(mHandle, bus, dev, fun, REG\_SECBUS, AI\_bw8);

0659: // printf("secbus: 0x%08x\n", secbus);

0660:

0661: nextcap = REG\_AERCAPHDR;

0662: //Search for the extended capability with id = 0x0001

0663: foundit = false;

0664: do

0665: {

0666: caphdr = readRegisterDS(mHandle, secbus, 0, 0, nextcap, AI\_bw32);

0667: // printf("caphdr at 0x%08x is 0x%08x\n", nextcap, caphdr);

0668: if ((caphdr & 0xFFFF) == 0x0001)

0669: {

0670: foundit = true;

0671: break;

0672: }

0673: nextcap = (caphdr >> 20) & 0xFFF; //Yes, this is supposed to be one behind.

0674: } while ( (nextcap != 0x0000) && (nextcap != 0xFFF) ); // 0x000 = end of list; 0xFFF = we're off in the weeds

0675: if (foundit)

0676: {

0677: //We've found the "Advanced Error Reported Extended Capability Header"

0678: //Offsets from that header indentify the regs we want to read.

0679: reg\_dscuncerr = nextcap + 0x4; //offset from found capability header to the uncorrectable error register for DSC

0680: reg\_dsccorerr = nextcap + 0x10; //offset to correctable error register for DSC

0681: readDSC = true;

0682: printf( "Monitoring down stream component errors at offsets 0x%03x and 0x%03x\n", reg\_dscuncerr, reg\_dsccorerr);

0683: //clear any errors before test starts

0684: writeRegisterDS(mHandle, secbus, 0, 0, reg\_dscuncerr, AI\_bw32, 0xffffFFFF);

0685: writeRegisterDS(mHandle, secbus, 0, 0, reg\_dsccorerr, AI\_bw32, 0xffffFFFF);

0686: }

0687: else

0688: {

0689: printf( "Unable to find error registers for down stream component.\n" );

0690: }

0691: }

0692:

0693: //Get the initial value of these registers so we don't repeatedly read them during the loop:

0694: lnkcon = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnkcon, AI\_bw16);

0695: lnkcon = lnkcon | 0x20; //set the retrain\_link bit

0696: lnkcon2 = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnkcon2, AI\_bw16);

0697:

0698:

0699: printf ("Starting test\n");

0700:

0701: for (i=0; i<m\_loops; i++)

0702: {

0703: //Clear the correctable and UNcorrectable errors //Will not work as PECI

0704: writeRegisterMem(mHandle, membus, dev, fun, REG\_CORERRSTS, AI\_bw32, 0xFFFFFFFF);

0705: writeRegisterMem(mHandle, membus, dev, fun, REG\_UNCERRSTS, AI\_bw32, 0xFFFFFFFF);

0706: //Clear framing errors

0707: // writeRegister(mHandle, bus, 7, 0, REG\_FRMERRSTS, AI\_bw32, 0xFFFFFFFF);

0708: //Clear errors in undocumented register

0709: //writeRegister(mHandle, bus, 7, 0, REG\_REUTERRRCV, AI\_bw32, 0xFFFFFFFF); //If we don't use it, don't clear it....

0710:

0711: //Check to see if we need to change speed (for type 6 loop)

0712: if ((i & 1) == 0)

0713: {

0714: //Even loop - use speed2

0715: expectedSpeed = speed2;

0716: }

0717: else

0718: {

0719: //Odd loop - use speed1

0720: expectedSpeed = speed1;

0721: }

0722:

0723: if (currentSpeed != expectedSpeed)

0724: {

0725: //First register is from cscripts; not certain; set bit "prvntinitspdch"

0726: //dbg0 = readRegister(mHandle, bus, 7, 2, REG\_PHYLTSSMDBG0, AI\_bw32);

0727: //writeRegister(mHandle, bus, 7, 7, REG\_PHYLTSSMDBG0, AI\_bw32, (dbg0 | 0x4)); //save dbg0 value so we can restore it later

0728: lnkcon2 = (lnkcon2 & 0xFFF0) | expectedSpeed; //Request new link speed

0729: writeRegisterMem(mHandle, membus, dev, fun, reg\_lnkcon2, AI\_bw16, lnkcon2); //Will not work as PECI

0730: }

0731: //Write the LNKCON register to cause the link to retrain

0732: writeRegisterMem(mHandle, membus, dev, fun, reg\_lnkcon, AI\_bw16, lnkcon); //PECI

0733:

0734: //TEST: extra check - second part - restore the reg:

0735: //writeRegister(mHandle, bus, 7, 7, REG\_PHYLTSSMDBG0, AI\_bw32, dbg0); //restore dbg0 value

0736:

0737: //loop checking the link status until the link has finished retraining (or we give up)

0738: for (j=0; j<100; j++) //max number of times to try

0739: {

0740: lnksts = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnksts, AI\_bw16);

0741: //printf ("LNKSTS:%08x\n", lnksts);

0742: if ((lnksts & 0x800) == 0)

0743: {

0744: //printf("Exit link retrain status loop after: %d tries.\n", j);

0745: break; //Normal exit for this loop after 1 or 2 tries

0746: }

0747:

0748: }

0749:

0750: if ((lnksts & 0x800) != 0)

0751: {

0752: printf("Link failed to finish re-training.\n");

0753: noTrains++;

0754: break;

0755: }

0756:

0757: if ((lnksts & 0x2000) == 0)

0758: {

0759: printf("Link is no longer active after retrain.\n");

0760: noTrains++;

0761: break;

0762: }

0763: currentWidth = (lnksts >> 4) & 0x3F;

0764: if (currentWidth != startWidth)

0765: {

0766: printf("Width: %d after retrain does not equal start width: %d\n", currentWidth, startWidth);

0767: widthErrors++;

0768: }

0769: currentSpeed = lnksts & 0xF;

0770: if (currentSpeed != expectedSpeed)

0771: {

0772: printf("Speed: %d after retrain does not equal expected speed: %d\n", currentSpeed, expectedSpeed);

0773: speedErrors++;

0774: }

0775: //Width returned is 1, 2, 4, 8 or 16. Convert to an array index.

0776: switch(currentWidth)

0777: {

0778: case 1: widthIndex = 0; break;

0779: case 2: widthIndex = 1; break;

0780: case 4: widthIndex = 2; break;

0781: case 8: widthIndex = 3; break;

0782: case 16: widthIndex = 4; break;

0783: default: widthIndex = 0; break;

0784: }

0785: //Speed is 1, 2, or 3 or 4; so we \*shouldn't\* need to convert it.

0786: //But we do need to watch in case we got all 1's back

0787: if (currentSpeed < 5)

0788: {

0789: widthSpeedCount[widthIndex][currentSpeed]++; //Count this loop's results for printout

0790: }

0791: //Check for errors that occured during retrain

0792: errs = readRegisterPeci(mHandle, bus, dev, fun, REG\_UNCERRSTS, AI\_bw32);

0793: if (errs != 0)

0794: {

0795: uncorErrors++;

0796: printf("Uncorrectable error: 0x%08X on port: %d loop: %d\n", errs, m\_port, i);

0797: }

0798: //Check for errors that occured during retrain

0799: errs = readRegisterPeci(mHandle, bus, dev, fun, REG\_CORERRSTS, AI\_bw32);

0800: if (errs != 0)

0801: {

0802: corErrors++;

0803: printf("Correctable error: 0x%08X on port: %d loop: %d\n", errs, m\_port, i);

0804: }

0805:

0806: if (readDSC)

0807: {

0808: errs = readRegisterDS(mHandle, secbus, 0, 0, reg\_dscuncerr, AI\_bw32);

0809: if (errs != 0)

0810: {

0811: dscUncorErrors++; //NEED: printfs, or up all errors for printout?

0812: writeRegisterDS(mHandle, secbus, 0, 0, reg\_dscuncerr, AI\_bw32, 0xffffFFFF);

0813: }

0814: errs = readRegisterDS(mHandle, secbus, 0, 0, reg\_dsccorerr, AI\_bw32);

0815: if (errs != 0)

0816: {

0817: dscCorErrors++;

0818: writeRegisterDS(mHandle, secbus, 0, 0, reg\_dsccorerr, AI\_bw32, 0xffffFFFF);

0819: }

0820: }

0821: }

0822:

0823: printf("Test finished on port %d\n", m\_port);

0824:

0825: //Print out the nice box with counts of each speed and width

0826:

0827: printf("+--------+-----------+-----------+-----------+-----------+-----------+\n");

0828: printf("| | x1 | x2 | x4 | x8 | x16 |\n");

0829: printf("+--------+-----------+-----------+-----------+-----------+-----------+\n");

0830: printf("| Gen1 | %8d | %8d | %8d | %8d | %8d |\n", widthSpeedCount[0][1], widthSpeedCount[1][1], widthSpeedCount[2][1], widthSpeedCount[3][1], widthSpeedCount[4][1]);

0831: printf("+--------+-----------+-----------+-----------+-----------+-----------+\n");

0832: printf("| Gen2 | %8d | %8d | %8d | %8d | %8d |\n", widthSpeedCount[0][2], widthSpeedCount[1][2], widthSpeedCount[2][2], widthSpeedCount[3][2], widthSpeedCount[4][2]);

0833: printf("+--------+-----------+-----------+-----------+-----------+-----------+\n");

0834: printf("| Gen3 | %8d | %8d | %8d | %8d | %8d |\n", widthSpeedCount[0][3], widthSpeedCount[1][3], widthSpeedCount[2][3], widthSpeedCount[3][3], widthSpeedCount[4][3]);

0835: printf("+--------+-----------+-----------+-----------+-----------+-----------+\n");

0836: printf("| Gen4 | %8d | %8d | %8d | %8d | %8d |\n", widthSpeedCount[0][4], widthSpeedCount[1][4], widthSpeedCount[2][4], widthSpeedCount[3][4], widthSpeedCount[4][4]);

0837: printf("+--------+-----------+-----------+-----------+-----------+-----------+\n");

0838:

0839: printf("\nSpeed errors %d\n", speedErrors);

0840: printf( "Width errors %d\n", widthErrors);

0841: printf( "No train %d\n", noTrains);

0842: printf( "Uncorrectable errors %d\n", uncorErrors);

0843: printf( "Correctable errors %d\n", corErrors);

0844: if (readDSC)

0845: {

0846: printf( "DSC Uncorrectable errors %d\n", dscUncorErrors);

0847: printf( "DSC Correctable errors %d\n", dscCorErrors);

0848: }

0849: } // lt\_loop

0850:

0851:

0852: //Used with command line arguments -f<n> to (attempt to) force a speed on a given link

0853: void changeSpeed(int mHandle, uint32\_t bus, uint32\_t dev, uint32\_t fun)

0854: {

0855: int j;

0856: uint32\_t lnksts;

0857: uint32\_t lnkcon;

0858: uint32\_t lnkcon2;

0859: uint32\_t reg\_lnksts;

0860: uint32\_t reg\_lnkcon;

0861: uint32\_t reg\_lnkcon2;

0862: uint32\_t startWidth;

0863: uint32\_t startSpeed;

0864: uint32\_t currentWidth;

0865: uint32\_t currentSpeed;

0866: uint32\_t membus;

0867: const char \*activ;

0868:

0869: //Correct for register offsets because DMI doesn't follow pattern of all other ports.

0870: if (m\_port == 0)

0871: {

0872: reg\_lnksts = REG\_DMI\_LNKSTS;

0873: reg\_lnkcon = REG\_DMI\_LNKCON;

0874: reg\_lnkcon2 = REG\_DMI\_LNKCON2;

0875: }

0876: else

0877: {

0878: reg\_lnksts = REG\_LNKSTS;

0879: reg\_lnkcon = REG\_LNKCON;

0880: reg\_lnkcon2 = REG\_LNKCON2;

0881: }

0882: if (!getMemBus(bus, &membus))

0883: {

0884: printf("Unable to find proper memory bus for PECI bus %d", bus);

0885: return;

0886: }

0887:

0888: prepTarget(mHandle, bus, dev, fun);

0889:

0890: //read LNKSTS and get link active, speed, width

0891: lnksts = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnksts, AI\_bw16);

0892: //Check for all ones; indicates we are not reading anything valid.

0893: if (lnksts == 0xFFFF)

0894: {

0895: printf("Link status is all ones; aborting test.\n");

0896: return;

0897: }

0898: //check to make sure the link is active before we begin. May be inactive due to not connected on target.

0899: if ((lnksts & 0x2000) == 0)

0900: {

0901: printf("Link is not active on startup, this is not likely to work.\n");

0902: activ = "NOT active";

0903: //return; let's try it anyway.

0904: }

0905: else

0906: {

0907: activ = "active";

0908: }

0909:

0910: startWidth = (lnksts >> 4) & 0x3F;

0911: startSpeed = lnksts & 0xF;

0912:

0913: printf ("Initial conditions for port %d %s, speed = %d, width = %d\n", m\_port, activ, startSpeed, startWidth);

0914:

0915: lnkcon2 = (lnkcon2 & 0xFFF0) | m\_forceSpeed; //Request new link speed

0916: writeRegisterMem(mHandle, membus, dev, fun, reg\_lnkcon2, AI\_bw16, lnkcon2);

0917:

0918: lnkcon = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnkcon, AI\_bw16);

0919: lnkcon = lnkcon | 0x20; //set the retrain\_link bit

0920:

0921: //Write the LNKCON register to cause the link to retrain

0922: writeRegisterMem(mHandle, membus, dev, fun, reg\_lnkcon, AI\_bw16, lnkcon);

0923:

0924: //loop checking the link status until the link has finished retraining (or we give up)

0925: for (j=0; j<100; j++) //max number of times to try

0926: {

0927: lnksts = readRegisterPeci(mHandle, bus, dev, fun, reg\_lnksts, AI\_bw16);

0928: //printf ("LNKSTS:%08x\n", lnksts);

0929: if ((lnksts & 0x800) == 0)

0930: {

0931: //printf("Exit link retrain status loop after: %d tries.\n", j);

0932: break; //Normal exit for this loop after 1 or 2 tries

0933: }

0934:

0935: }

0936:

0937: if ((lnksts & 0x800) != 0)

0938: {

0939: printf("Link failed to finish re-training.\n");

0940: }

0941:

0942: if ((lnksts & 0x2000) == 0)

0943: {

0944: activ = "NOT active";

0945: }

0946: else

0947: {

0948: activ = "active";

0949: }

0950:

0951: currentWidth = (lnksts >> 4) & 0x3F;

0952: if (currentWidth != startWidth)

0953: {

0954: printf("Width: %d after retrain does not equal start width: %d\n", currentWidth, startWidth);

0955: }

0956: currentSpeed = lnksts & 0xF;

0957: if (currentSpeed != m\_forceSpeed)

0958: {

0959: printf("Speed: %d after retrain does not equal expected speed: %d\n", currentSpeed, m\_forceSpeed);

0960: }

0961:

0962: printf ("Final port conditions for port %d %s, speed = %d, width = %d\n", m\_port, activ, currentSpeed, currentWidth);

0963:

0964: } // changeSpeed

0965:

0966:

0967: //Utility for displayOverview

0968: //prints a line based on number of ports to separate display elements

0969: void printSeparator(void)

0970: {

0971: int i;

0972:

0973: printf("+-------+");

0974: for (i=0; i<MAXPORT; i++)

0975: {

0976: printf("----+");

0977: }

0978: printf("\n");

0979: } // printSeparator

0980:

0981: void displayOverview(int mHandle, int numcpus)

0982: //Displays an overview of the PCIe links. (Known as topo in cscripts, but I already used -t for type)

0983: {

0984: int portNegotiatedWidth[MAXPORT];

0985: int portMaxWidth[MAXPORT];

0986: uint32\_t portActive[MAXPORT];

0987: uint32\_t portWidth[MAXPORT];

0988: uint32\_t portSpeed[MAXPORT];

0989: uint32\_t portSecBus[MAXPORT];

0990: uint32\_t deviceVendorID[MAXPORT];

0991: int i;

0992: int base;

0993: uint32\_t bus;

0994: uint32\_t dev;

0995: uint32\_t fun;

0996: uint32\_t bif;

0997: uint32\_t lnkcap;

0998: uint16\_t lnksts;

0999: uint32\_t msm\_pci\_peci\_bios;

1000: int curcpu;

1001: int iError = 0;

1002:

1003: for (curcpu = CPU\_ZERO\_POS; curcpu < (CPU\_ZERO\_POS + numcpus); curcpu++)

1004: {

1005: m\_peciCPU = curcpu - CPU\_ZERO\_POS; //required for PECI access to work properly

1006: printf("\n\nPCIe topology for socket %d\n", curcpu);

1007:

1008: if ((iError = ai\_mSetActiveCPU(mHandle, curcpu)) != AI\_SUCCESS)

1009: {

1010: printf ("\nSetActiveCPU: ERROR: %s Socket %hu\n" , ai\_ErrorToString(iError), curcpu);

1011: return;

1012: }

1013: ai\_mSetActiveCore(mHandle, CORE\_ZERO\_POS);

1014: ai\_mSetActiveThread(mHandle, THREAD\_ZERO\_POS);

1015:

1016: for (i=0; i<MAXPORT; i++)

1017: {

1018: portMaxWidth[i] = 0;

1019: portActive[i] = 0;

1020: portWidth[i] = 0; //Negotiated width

1021: portSpeed[i] = 0;

1022: portSecBus[i] = 0;

1023: }

1024:

1025: for (i=0; i<MAXPORT; i++)

1026: {

1027: port2bdf(i, &bus, &dev, &fun);

1028: deviceVendorID[i] = readRegisterPeci(mHandle, bus, dev, fun, 0, AI\_bw32);

1029: if (i == 0)

1030: {

1031: lnksts = readRegisterPeci(mHandle, bus, dev, fun, REG\_DMI\_LNKSTS, AI\_bw16);

1032: }

1033: else

1034: {

1035: lnksts = readRegisterPeci(mHandle, bus, dev, fun, REG\_LNKSTS, AI\_bw16);

1036: }

1037: if (lnksts != 0xFFFF)

1038: {

1039: portActive[i] = (lnksts & 0x2000) >> 13;

1040: portWidth[i] = (lnksts & 0x3F0) >> 4;

1041: portSpeed[i] = lnksts & 0xF;

1042: }

1043: lnkcap = readRegisterPeci(mHandle, bus, dev, fun, REG\_LNKCAP, AI\_bw32);

1044: if (lnkcap != 0xFFFFFFFF)

1045: {

1046: portMaxWidth[i] = (lnkcap & 0x3f0) >> 4;

1047: }

1048: portSecBus[i] = readRegisterPeci(mHandle, bus, dev, fun, REG\_SECBUS, AI\_bw8);

1049: }

1050:

1051:

1052: //Time to print everything out

1053: printSeparator();

1054: printf("| port |");

1055: for (i=0; i<MAXPORT; i++)

1056: {

1057: printf(" %2d |", i);

1058: }

1059: printf("\n");

1060: printSeparator();

1061:

1062: printf("|DEV ID |");

1063: for (i=0; i<MAXPORT; i++)

1064: {

1065: printf("%04x|", ((deviceVendorID[i] >> 16) & 0xFFFF));

1066: }

1067: printf("\n");

1068: printSeparator();

1069:

1070: printf("|VEN ID |");

1071: for (i=0; i<MAXPORT; i++)

1072: {

1073: printf("%04x|", (deviceVendorID[i] & 0xFFFF));

1074: }

1075: printf("\n");

1076: printSeparator();

1077:

1078: printf("|max wd |");

1079: for (i=0; i<MAXPORT; i++)

1080: {

1081: switch(portMaxWidth[i])

1082: {

1083: case 4: printf( " x4 |"); break;

1084: case 8: printf( " x8 |"); break;

1085: case 16: printf("x16 |"); break;

1086: default: printf(" |"); break;

1087: }

1088: }

1089: printf("\n");

1090: printSeparator();

1091: printf("|Active |");

1092: for (i=0; i<MAXPORT; i++)

1093: {

1094: if (portActive[i] != 0)

1095: {

1096: printf(" Y |");

1097: }

1098: else

1099: {

1100: printf(" |");

1101: }

1102: }

1103: printf("\n");

1104: printSeparator();

1105: printf("| width |");

1106: for (i=0; i<MAXPORT; i++)

1107: {

1108: switch(portWidth[i])

1109: {

1110: case 1: printf( " x1 |"); break;

1111: case 2: printf( " x2 |"); break;

1112: case 4: printf( " x4 |"); break;

1113: case 8: printf( " x8 |"); break;

1114: case 16: printf("x16 |"); break;

1115: default: printf(" |"); break;

1116: }

1117:

1118: }

1119: printf("\n");

1120: printSeparator();

1121:

1122: printf("| speed |");

1123: for (i=0; i<MAXPORT; i++)

1124: {

1125: if (portActive[i] != 0)

1126: {

1127: printf(" %d |", portSpeed[i]);

1128: }

1129: else

1130: {

1131: printf(" |");

1132: }

1133: }

1134: printf("\n");

1135: printSeparator();

1136:

1137:

1138: printf("| secbus|");

1139: for (i=0; i<MAXPORT; i++)

1140: {

1141: if (portActive[i] != 0)

1142: {

1143: printf(" %02x |", portSecBus[i]);

1144: }

1145: else

1146: {

1147: printf(" |");

1148: }

1149: }

1150: printf("\n");

1151: printSeparator();

1152: printf("\n");

1153:

1154:

1155:

1156: } //for each cpu

1157:

1158: } // displayOverview

1159:

1160:

1161: void pciScan(int mHandle)

1162: {

1163: uint32\_t bus;

1164: uint32\_t dev;

1165: uint32\_t fun;

1166: uint32\_t devend;

1167: int count;

1168:

1169: printf("Bus Dev Fun Device/Vendor ID\n");

1170: count = 0;

1171:

1172: for (bus = 0; bus < 5; bus++)

1173: {

1174: for (dev = 0; dev < 5; dev++)

1175: {

1176: for (fun = 0; fun < 8; fun++)

1177: {

1178: devend = readRegisterPeci(mHandle, bus, dev, fun, 0, AI\_bw32);

1179: if (devend != 0xFFFFffff)

1180: {

1181: printf("%3d %3d %3d 0x%08x\n", bus, dev, fun, devend);

1182: count++;

1183: }

1184: }

1185: }

1186: }

1187:

1188: printf("\nFound %d PCI devices.\n\n", count);

1189:

1190: } // pciScan

1191:

1192: void do\_test(int mHandle, int numcpus, uint32\_t bus, uint32\_t dev, uint32\_t fun)

1193: {

1194: if (m\_overview)

1195: {

1196: displayOverview(mHandle, numcpus);

1197: return;

1198: }

1199:

1200: if (m\_pciScan)

1201: {

1202: pciScan(mHandle);

1203: return;

1204: }

1205:

1206: if (m\_forceSpeed > 0)

1207: {

1208: changeSpeed(mHandle, bus, dev, fun);

1209: return;

1210: }

1211:

1212: lt\_loop(mHandle, bus, dev, fun);

1213:

1214: } // do\_test

1215:

1216: int main (int argc, char \*\*argv)

1217: {

1218: int numcores;

1219: int curcore;

1220: int numcpus;

1221: int curcpu;

1222: int iError = 0;

1223: bool pwrchk = true, scnsetup = true, savemodarch = true;

1224: int mHandle;

1225: FILE \*UUTDiagsHexFile;

1226: char ver[200];

1227: uint64\_t msr;

1228: uint64\_t regdata;

1229: int i;

1230: struct timespec start\_time;

1231: struct timespec end\_time;

1232: double secs;

1233: uint32\_t bus;

1234: uint32\_t dev;

1235: uint32\_t fun;

1236: ai\_ITPtopology\_t topo;

1237: uint16\_t curCPU;

1238:

1239: UUTDiagsHexFile = NULL;

1240:

1241: printf("\n\nLink Training Loop test\n");

1242:

1243: iError = parseArgs(argc, argv);

1244: if (iError != 0)

1245: {

1246: usage();

1247: return iError;

1248: }

1249:

1250: ai\_GetLibraryVersion(ver);

1251: printf("Library version = %s\n", ver);

1252:

1253: AI\_pdcselector pdctarget = AI\_pdc\_0;

1254:

1255: if ((iError = ai\_mOpen(pdctarget, 1, &mHandle)) != AI\_SUCCESS)

1256: {

1257: printf ("\nOpen ERROR: %s Channel %i\n" , ai\_ErrorToString(iError), pdctarget);

1258: return 1;

1259: }

1260:

1261: if ((iError = ai\_mSetTargetCPUType(mHandle, AI\_sandybridge)) != AI\_SUCCESS)

1262: {

1263: printf ("\nSetTargetCPUType: ERROR: %s Channel %i\n" , ai\_ErrorToString(iError), pdctarget);

1264: return 1;

1265: }

1266:

1267: ai\_mConfig (mHandle, 100, UUTDiagsHexFile, 0x10000LL, pwrchk, scnsetup, savemodarch);

1268:

1269: iError = ai\_mGetITPScanChainTopology(mHandle, &topo, true);

1270: if (iError != AI\_SUCCESS)

1271: {

1272: printf ("\nERROR getting target topology: %s\n" , ai\_ErrorToString(iError));

1273: return iError;

1274: }

1275:

1276: numcpus = topo.tck[TCK\_ZERO\_POS].numcpus;

1277: if ((m\_socket < CPU\_ZERO\_POS) || (m\_socket > numcpus))

1278: {

1279: printf("Invalid socket number, must be between %hu and %hu\n", CPU\_ZERO\_POS, numcpus);

1280: return -1;

1281: }

1282:

1283: //Halt all cores in all CPUs

1284: for (curCPU=CPU\_ZERO\_POS; curCPU < (numcpus + CPU\_ZERO\_POS); curCPU++)

1285: {

1286: if ((iError = ai\_mSetActiveCPU(mHandle, curCPU)) != AI\_SUCCESS)

1287: {

1288: printf ("\nSetActiveCPU: ERROR: %s Socket %hu\n" , ai\_ErrorToString(iError), curCPU);

1289: return 1;

1290: }

1291:

1292: ai\_mSetActiveCore(mHandle, CORE\_ZERO\_POS);

1293: ai\_mSetActiveThread(mHandle, THREAD\_ZERO\_POS);

1294: if ((iError = ai\_mEnterDebugMode(mHandle)) != AI\_SUCCESS)

1295: {

1296: printf ("\n EnterDebugMode: ERROR: %s Socket %hu\n" , ai\_ErrorToString(iError), curCPU);

1297: return 1;

1298: }

1299: }

1300:

1301: //TODO

1302: //Check return values:

1303: ai\_mIOSFTAPinit(mHandle);

1304:

1305: m\_bus0 = 0; //Start with bus 0 for bus discovery

1306:

1307: //Get TAP ownership for all TAPs, overview will use all TAPs (CPUs)

1308: for (curCPU=0; curCPU < numcpus; curCPU++)

1309: {

1310: m\_peciCPU = curCPU;

1311: ai\_mIOSFTAPownership(mHandle, true, curCPU);

1312: }

1313:

1314: //Prepare the target and get the bus numbers for each socket

1315: //We need the bus min/max for first socket in order for second socket to work

1316: for (curCPU=0; curCPU < numcpus; curCPU++)

1317: {

1318: m\_peciCPU = curCPU;

1319: prepTarget(mHandle, 0, 0, 0); //attempt to "unhide" devices; b/d/f is ignored

1320: getBusNumbers(mHandle, curCPU);

1321: }

1322:

1323: printf("Selecting socket %hu\n", m\_socket);

1324:

1325: if ((iError = ai\_mSetActiveCPU(mHandle, m\_socket)) != AI\_SUCCESS)

1326: {

1327: printf ("\nSetActiveCPU: ERROR: %s Socket %hu\n" , ai\_ErrorToString(iError), m\_socket);

1328: return 1;

1329: }

1330: m\_peciCPU = m\_socket - CPU\_ZERO\_POS;

1331: ai\_mSetActiveCore(mHandle, CORE\_ZERO\_POS);

1332: ai\_mSetActiveThread(mHandle, THREAD\_ZERO\_POS);

1333:

1334:

1335: clock\_gettime(CLOCK\_MONOTONIC, &start\_time);

1336:

1337: port2bdf(m\_port, &bus, &dev, &fun); //Convert the command line option -p<n> to bus, device, function

1338:

1339: do\_test(mHandle, numcpus, bus, dev, fun);

1340:

1341: clock\_gettime(CLOCK\_MONOTONIC, &end\_time);

1342: secs = (double)(end\_time.tv\_sec - start\_time.tv\_sec) + (double)(end\_time.tv\_nsec - start\_time.tv\_nsec) / 1000000000.0;

1343: printf("Time for test: %7.2f seconds.\n\n", secs);

1344:

1345: ai\_mClose(mHandle);

1346:

1347: return iError;

1348: }

1349:

1350: //copyright 2021 ASSET InterTech, Inc.