Compsys scripts

Emil Møller Hansen

Contents

1 Address table

```
//Use this if you want to translate multiple addresses at once, use cache.js if
//you only want to translate one address.
//The hope is that this makes cache exercises easier as you can look at all
//the address translations at once


//Inputs:
//B: Size of each block in bytes
//E: Lines per set
//C: Size of cache in bytes
//m: number of bits in address
//addresses: array of addresses needed to be translated
//base: base to print, set index, tag and offset in (number between 2 and 36)
//Prints a table showing translation of each address
function translateAddresses(B, E, C, m, addresses, base) {
    var b = log2(B)
    var S = C / (B * E);
    var s = log2(S);
    var t = m - (s + b);
    var headLine = "Address (HEX)\tBits (BIN)"
    //insert correct amount of tabs to make room for all bits
    for(var i = 0; i < parseInt(m / 4) - 2; i++) {
        headLine += "\t";
    }
    headLine += "Set\t\tTag\t\tOffset";
    console.log(headLine);


    for(var i = 0; i < addresses.length; i++) {
        var str = "";
        //Shift away the block offset and use bit mask to isolate index
        var setIndex = (addresses[i] >>> b) & bitMask(s)
        //Shift away block offset and index, the only thing left is the tag
        var tag = (addresses[i] >>> (b+s));
        //Use bit mask to remove bits more significant than those used to the
        //block offset
        var blockOffset = addresses[i] & bitMask(b);

        str += addresses[i].toString(16) + "\t\t";
        str += toBin(addresses[i], m) + "\t";
        //Insert extra tab in case of many bits
        if(m >= 24) {
            str += "\t";
```

```
        }
        str += setIndex.toString(base) + "\t\t";
        str += tag.toString(base) + "\t\t";
        str += blockOffset.toString(base);
        console.log(str);
    }
}

//Converts number to string of len bits, I use this to get correct length
function toBin(num, len) {
    var res = "";
    var temp = 0;
    for (var i = len - 1; i >= 0; i--) {
        temp = num & parseInt(Math.pow(2, i));
        if (temp == 0) {
            res += "0";
        } else {
            res += "1";
        }
        if (i % 4 == 0) {
            res += " ";
        }
    }
    return res;
}

//Returns a number with p ones as least significant bits, rest is zeros
function bitMask(p) {

    var mask = 0;
    for (var i = 0; i < p; i++) {
        mask += Math.pow(2, i);
    }
    return parseInt(mask);
}

//Since IE does not support Math.log2 use this
function log2(x){
    return Math.log(x)/Math.log(2);
}

//Examples below:

//Prints information in binary
// translateAddresses(64, 2, 2048, 20,
```

```
//[0xA000, 0xF020, 0xFF00, 0xFF0C, 0x0018,0xF0A4, 0xF004], 2);

//Prints information in hex
//translateAddresses(64, 2, 2048, 20,
//[0xA000, 0xF020, 0xFF00, 0xFF0C, 0x0018,0xF0A4, 0xF004], 16);
```

2 Cache address

```
//Script to give information on an address when using specific cache
//I believe that the names comply with the book as seen on page 653
var address = 0x0004; //Change this to fit exercise
//Size of each block in bytes
var B = 64; //Change this to fit exercise
//Number of lines pr. set
var E = 2; //Change this to fit exercise
//Size of cache in bytes
var C = 2048; //Change this to fit exercise

//number of bits in address
var m = 16; //Change this to fit exercise

//Prints info on a given address with given cache attributes
function translateAddress(B, E, C, m, address) {
    var b = log2(B);
    var S = C / (B * E);
    var s = log2(S);
    var t = m - (s + b);
    //Dont know if this is needed, but here you go you spoiled brat
    var bits = toBin(address, m);
    console.log("Bits in address: " + bits);

    //Use bit mask to remove bits more significant than those used to the block offset
    var blockOffset = address & bitMask(b);
    console.log("Block offset (HEX): " + blockOffset.toString(16));

    //Shift away the block offset and use bit mask to isolate index
    var setIndex = (address >>> b) & bitMask(s)
    console.log("Set index (HEX): " + setIndex.toString(16));

    //Shift away block offset and index, the only thing left is the tag
    var tag = (address >>> (b+s));
    console.log("Cache tag (HEX): " + tag.toString(16));
}

//Converts number to string of len bits, I use this to get correct length
function toBin(num, len) {
    var res = "";
    var temp = 0;
    for (var i = len - 1; i >= 0; i--) {
        temp = num & parseInt(Math.pow(2, i));
        if (temp == 0) {
```

```
            res += "0";
        } else {
            res += "1";
        }
        if (i % 4 == 0) {
            res += " ";
        }
    }
    return res;
}

//Returns a number with p ones as least significant bits, rest is zeros
function bitMask(p) {

    var mask = 0;
    for (var i = 0; i < p; i++) {
        mask += Math.pow(2, i);
    }
    return parseInt(mask);
}

//Since IE does not support Math.log2 use this
function log2(x){
    return Math.log(x)/Math.log(2);
}
```

3 Cache info

```
//Script to give information on an address when using specific cache
//I believe that the names comply with the book as seen on page 653
var address = 0x0004; //Change this to fit exercise
//Size of each block in bytes
var B = 64; //Change this to fit exercise
//Number of lines pr. set
var E = 2; //Change this to fit exercise
//Size of cache in bytes
var C = 2048; //Change this to fit exercise

//Number of bits in address
var m = 16; //Change this to fit exercise

//Prints how many bits are used for different things for cache with given attributes
//B: Block size
//E: Number of lines pr. set
//C: Capacity of cache
//m: number of bits in address
function cacheInfo(B, E, C, m) {
    var b = log2(B);
    var S = C / (B * E);
    var s = log2(S);
    var t = m - (s + b);

    console.log("Number of sets: " + S);
    console.log("Number of bits in block offset: " + b);
    console.log("Number of bits in set index: " + s);
    console.log("Number of bits in tag: " + t);
}

//Since IE does not support Math.log2 use this
function log2(x){
    return Math.log(x)/Math.log(2);
}
```

4 Chapter 3 p51 (networking)

```
//Script to complete exercises like the P51 in chapter 3 network book

//prints the senders cwnd for each RTT. Assuming that if there are sent more
//than the link can contain both senders get data segment loss and assuming
//that senders receive triple dub ACK then that happens. Therefor we dont need
//the thresholds

//Segments sent by each sender pr. round
var cwnd_c1 = 15;
var cwnd_c2 = 10;
//Round trip time for each sender in ms
var rtt_c1 = 100;
var rtt_c2 = 100;
//speed of the link c1 and c2 uses in segments / sec
var link_speed = 30;
var maxTime = 2200;
var timeInc;
//Set incrementation of timer so we can simulate each round trip for each sender
if (rtt_c1 < rtt_c2) {
    timeInc = rtt_c1;
} else {
    timeInc = rtt_c2;
}
console.log("Time: " + 0 + "\t\tcwnd for c1: " + cwnd_c1 + "\tcwnd for c2: " + cwnd_c2);
//Notice that 3 isn't added when loss is detected, sicne the exercise doesn't
// say there should
for (var i = timeInc; i <= maxTime; i += timeInc) {
    //save old cwnd for c1 to use for computations later
    old_cwnd_c1 = cwnd_c1;
    if (i % rtt_c1 == 0) {
        if (cwnd_c1 * 1000 / rtt_c1 + cwnd_c2 * 1000 / rtt_c2 > link_speed) {
            //Both experience loss, notice that there will not be added 3 as
            //normaly, because the exercise I made this for didn't
            cwnd_c1 = parseInt(cwnd_c1 / 2);
            //Cannot go bellow 1
            if(cwnd_c1 < 1) {
                cwnd_c1 = 1;
            }
        } else {
            cwnd_c1++;
        }
    }
    if (i % rtt_c2 == 0) {
```

```
        if (old_cwnd_c1 * 1000 / rtt_c1 + cwnd_c2 * 1000 / rtt_c2 > link_speed) {
            //Both experience loss, notice that there will not be added 3 as
            //normaly, because the exercise I made this for didn't
            cwnd_c2 = parseInt(cwnd_c2 / 2);
            //Cannot go bellow 1
            if(cwnd_c2 < 1) {
                cwnd_c2 = 1;
            }
        } else {
            cwnd_c2++;
        }
    }
    console.log("Time: " + i + "\tcwnd for c1: " + cwnd_c1 +
         "\tcwnd for c2: " + cwnd_c2);
}
```

5 Cyclic redundancy check

```
//JavaScript-C24.2.0 (SpiderMonkey)
```

```
function crc(generator, message) {
    var message_original = message; //retain old M

    var i = 0;
    var offset = generator.length; //get G length

    for (n = 0; n < offset-1; n++) { //Append G length -1 zeros to M
        //(G = length of generator)
        message += "0";
    }
    console.log("M  : "+message);
    while (i <= message.length - offset) {
        //Continue until we find a one in the message
        if (message[i] == "1") {
            var temp = "";
            for (g = 0; g < generator.length; g++) {
                //Some char xor-ing
                if (message[i+g] == "0" && generator[g] == "0") {
                    temp += '0';
                }
                else if (message[i+g] == "0" && generator[g] == "1") {
                    temp += '1';
                }
                else if (message[i+g] == "1" && generator[g] == "0") {
                    temp += '1';
                }
                else if (message[i+g] == "1" && generator[g] == "1") {
                    temp += '0';
                }
            }
            //Fill string with some amount of spaces
            var spaces = "";
            for (var z = 0; z < i; z++) { spaces += ' '; }

            console.log("G:   "+spaces + generator);
            console.log("temp:"+spaces + temp);
            console.log("_________________");
```

```
            //Add intermediate result to message
            message = message.substring(0, i) + temp +
                message.substring(i+offset, message.length);
            console.log("M  : "+message);
        }
        i++;
    }
    console.log("\n\ntemp: "+temp);

    //gets the last (offset - 1) chars of message = R
    var R = message.substring(message.length - (offset - 1));
    console.log("R: " + R);
    //message concated with R
    console.log("M&R: " + message_original +" "+ R);
}

var generator = "1001";
var message = "101110";
//Example:
//crc(generator, message); //Should give R = 011 see page 478
```

6 internet checksum

```
//The checksum is computed with carry and wraparound.


//Splits string into 16-bit integers then sums it up and prints 1s compliment
//of that, meaning the negated version
//If string does not have even length, the last byte is 0
function checkSumString(str) {
    var sum = 0;
    for(var i = 0; i < str.length; i+=2) {
        var n1 = str.charCodeAt(i);

        //Set n2 to zero in case string is not any longer
        var n2 = 0;
        if(i + 1 < str.length) {
            n2 = str.charCodeAt(i+1);
        }
        //Shift n1 8 bits since this is most significant byte
        var n = (n1 << 8) + n2;
        if(i + 1 < str.length) {
            console.log(str[i] + str[i+1] + ":\t\t" + toBin(n,16));
        } else {
            //No more chars so just write null-char
            console.log(str[i] + "\\0:\t\t" + toBin(n,16));
        }

        sum += n;
        //Wrap around if there is overflow
        if(sum > Math.pow(2, 16)) {
            //Subtract 2^16 to remove the 17th bit, then add the carry in the
            // least significant bit
            sum = sum - Math.pow(2,16) + 1;
        }
    }
    console.log("-----------------------------------")
    console.log("Sum:\t\t" + toBin(sum, 16));
    //This is the checksum
    console.log("1s compliment:\t" + toBin(~sum, 16) +
        " (this is the checksum)");

}

//Same as above, but takes array of bytes instead of a string
//Expects that all entries are between 0 and 255 (both inclusive)
```

```
function checkSumBytes(bytes) {
    var sum = 0;
    for(var i = 0; i < bytes.length; i+=2) {
        var n1 = bytes[i];

        //Set n2 to zero in case string is not any longer
        var n2 = 0;
        if(i + 1 < bytes.length) {
            n2 = bytes[i + 1];
        }
        //Shift n1 8 bits since this is most significant byte
        var n = (n1 << 8) + n2;
        if(i + 1 < bytes.length) {
            console.log(bytes[i] + "" + bytes[i+1] + ":\t\t" + toBin(n,16));
        } else {
            //No more chars so just write null-char
            console.log(bytes[i] + "\\0:\t\t" + toBin(n,16));
        }

        sum += n;
        //Wrap around if there is overflow
        if(sum > Math.pow(2, 16)) {
            //Subtract 2^16 to remove the 17th bit, then add the carry in the
            //least significant bit
            sum = sum - Math.pow(2,16) + 1;
        }
    }
    console.log("___________________________________")
    console.log("Sum:\t\t" + toBin(sum, 16));
    //This is the checksum
    console.log("1s compliment:\t" + toBin(~sum, 16) + " (this is the checksum)");
}

//Example from textbook page 233
//the last two characters might not be displayed correctly
//but the computation agrees with the book
//var str = "f`UU" + String.fromCharCode(143) + String.fromCharCode(12);
//checkSumString(str);
// var bytes = [parseInt("01100110", 2), parseInt("01100000", 2),
//  parseInt("01010101", 2), parseInt("01010101", 2),
//  parseInt("10001111", 2), parseInt("00001100", 2)];
// checkSumBytes(bytes);


//Converts number to string of len bits, I use this to get correct
```

```
//amount of bits
function toBin(num, len) {
    var res = "";
    var temp = 0;
    for (var i = len - 1; i >= 0; i--) {
        temp = num & parseInt(Math.pow(2, i));
        if (temp == 0) {
            res += "0";
        } else {
            res += "1";
        }
        if (i % 4 == 0) {
            res += " ";
        }
    }
    return res;
}
```

7 Hash function page 640

```
//Couldn't find a good description of this, so I assume two things:
//1) the message is padded with zeros if length is not divisible by 4
//2) That there won't be exercises with this in the exam since we haven't used
// it much in the course

//checkSumString('CHKSUM');

function checkSumString(str) {
    var sums = [];
    for (var i = 0; i < 4; i++) {
        sums[i] = 0;
    }
    var res = "";
    for (var i = 0; i < str.length; i++) {
        if (i % 4 == 0) {
            console.log(res);
            res = "";
        }
        res += str.charCodeAt(i).toString(16) + "   ";

        sums[i % 4] = (sums[i % 4] + str.charCodeAt(i)) % 0xff;
    }
    //Since the last line won't be printed in the loop, print it here
    console.log(res);
    console.log("-------------------")
    res = "";
    for (var i = 0; i < 4; i++) {
        res += sums[i].toString(16) + "   ";

    }
    console.log(res);

}


//Returns the n'th byte from num, 0th being the least significant byte
function extractByte(n, num) {
    var mask = 0;
    for (var i = n * 8; i < (n + 1) * 8; i++) {
        //Add 8 bits set to one in correct position
        mask += parseInt(Math.pow(2, i));
    }
    //bitwise and to set all other bytes to 0
```

```
    //then shift so we only have one relevant byte
    return (mask & num) >> (n * 8);
}
```

8 To bin

```
//Converts number to string of len bits, I use this to get correct length
function toBin(num, len) {
    var res = "";
    var temp = 0;
    for (var i = len - 1; i >= 0; i--) {
        temp = num & parseInt(Math.pow(2, i));
        if (temp == 0) {
            res += "0";
        } else {
            res += "1";
        }
        if (i % 4 == 0) {
            res += " ";
        }
    }
    console.log(res);
}
```

9 Virtual address translation

```
//Script to give information on a specific address in virtual mem,
//virtual address to be used
var vAddress = 0x03d4; //Change this to fit exercise
var pageSize = 64; //Change this to fit exercise
//Number of sets in TLB
var tlbNrOfSets = 4; //Change this to fit exercise
//number of bits in virtual address
var vAddressLen = 14; //Change this to fit exercise


function translateAddress(vAddress, pageSize, tlbNrOfSets, vAddressLen) {
    var p = log2(pageSize);

    var t = log2(tlbNrOfSets);

    var bits = toBin(vAddress, vAddressLen);
    console.log("Bits in virtual address: " + bits);

    //Shift in order to only keep the bits needed to vpn
    var vpn = vAddress >>> p;
    console.log("VPN (HEX): " + vpn.toString(16));
    console.log("VPN (BIN): " + toBin(vpn, vAddressLen - p));

    var tlbI = (vAddress >>> p) & bitMask(t);
    console.log("TLB Index (HEX): " + tlbI.toString(16));
    console.log("TLB Index (BIN): " + toBin(tlbI, t));

    var tlbT = vAddress >>> (p + t);
    console.log("TLB Tag (HEX): " + tlbT.toString(16));
    console.log("TLB Tag (BIN): " + toBin(tlbT, p + t));

    var vpo = vAddress & bitMask(p);
    console.log("VPO (HEX): " + vpo.toString(16));
    console.log("VPO (BIN): " + toBin(vpo, p));

}

//Converts number to string of len bits, I use this to get correct length
function toBin(num, len) {
    var res = "";
    var temp = 0;
    for (var i = len - 1; i >= 0; i--) {
        temp = num & parseInt(Math.pow(2, i));
```

```
        if (temp == 0) {
            res += "0";
        } else {
            res += "1";
        }
        if (i % 4 == 0) {
            res += " ";
        }
    }
    return res;
}

//Returns a number with p ones as least significant bits, rest is zeros
function bitMask(p) {

    var mask = 0;
    for (var i = 0; i < p; i++) {
        mask += Math.pow(2, i);
    }
    return parseInt(mask);
}

//Since IE does not support Math.log2 use this
function log2(x){
    return Math.log(x)/Math.log(2);
}

//translateAddress(0x0825, 32, 4, 13);
```