

AMD Programming Contest

Hosted by the IEEE Computer Society

March 22, 2008

1 Contest Information

1.1 Rules

The AMD Programming Contest is open to all current UT ECE and Computer Science majors. Contestants will compete in teams of two and will not be allowed to discuss or collaborate with other teams in the contest.

There will be five programming puzzles that each team will have four days to complete. The contest will formally start at 9:00 P.M. March 19th and end at 11:59 P.M. March 23rd. Each problem is worth 20 points with 100 possible points in the entire contest. The results for the contest will be presented at the AMD Keynote Lecture on March 26, 2008 in ACES 1.104.

1.2 Prizes

- 1st Place - AMD Turion X2 Laptops
- 2nd Place - Nintendo Wiis
- 3rd Place - 22" Acer Widescreen Monitors

1.3 Requirements

The contest will be open to five languages to accommodate the participants. The languages are as follows:

- C/C++ (GCC 4.1.2)
- Java (Sun SDK 1.5.0)
- Ruby 1.8.5
- Python 2.4.3
- Perl 5.8

Your submitted solutions will be invoked on the command line on the ECE or CS Linux machines. This means you cannot submit a Visual Studio or Eclipse project or a Java application with a GUI. You may use either a 32bit or 64bit machine but you must specify which platform to build on. Along with your submission, include a README file that provides instructions for compiling and running your code.

Additionally, you are **NOT** allowed to use source files or libraries from external sources.

1.4 Judging

Each programming puzzle will be graded against test cases that will not be available to participants. It is your responsibility to test your code for corner cases in addition to verifying your solution returns the correct result for the given sample test cases. Any submissions that error during run time or do not compile will not be debugged (or graded)

Each puzzle is worth 20 points of the your total grade, but the puzzles are not the same level of difficulty. Contestants are responsible for determining the difficulty of each programming puzzle and should allocate their time so that they can achieve the most points possible.

Excluding the ISA problem, 5 out of the 20 points for each puzzle grade will be determined based on a criteria unique to each problem. For example, in Reiser's Maze, 5 points will be given based on the execution time of your solution compared to the solution of other contestants. If your solution executes our test cases faster than all other contestants you can expect to earn five points. If not, you will be awarded 1-4 points.

1.5 Submission

Teams will submit their solutions to the IEEE Computer Society's Challenge website. All submission files **MUST** be a zip, gzipped or bzip2 tarball of your solution. In addition to submitting source code solutions, you will be required to provide a solution for a given test case on the website.

Each team will only be allowed to submit ONE solution per problem. The submission website will not accept your source code until the correct answer has been given for the problem. We advise that you do not wait till the last minute to submit because if you don't have the correct answer to the given test case, your source file will not be accepted. After you have answered a problem correctly and submitted the corresponding source code, you will not be able to update or resubmit new code for this problem.

Tip: You can submit your solution for the given test case on the website **WITHOUT** submitting source files. If you do this, your answer will be confirmed as correct or incorrect and you will subsequently see a message about submitting a source file. Use this policy to your team's advantage.

1.6 Clarifications

If you have questions related to the contest problems or submission policy, send an email to amdprogrammingcontest@gmail.com with your team name and question. We will post clarifications and answers to these questions on the IEEE Computer Society's website under the forum section.

2 BitBin

2.1 Problem Statement

Mathematician Dimitri Bozo has written a solution for a very difficult math problem proposed to him by an aspiring student.

The problem is as follows:

Given $f(p,z)$ is true if the number of '1's in the binary representation of p is equal to any prime factor of z ,

Compute how many numbers in the range (x,y) make $f(p,z)$ true i.e. $\text{bitbin}(x,y,z)$.

example:

$\text{bitbin}(1,8,6) = 4$ (0b11, 0b101, 0b110, 0b111)

Dimitri's solution has been proven correct but it does not scale for large ranges of (x,y) . Your job is to help Dimitri by providing him with a solution that works for all values of x and y inside the unsigned 64 bit space. Also, z and its largest prime factor is guaranteed to be less than $2^{64} - 1$ and 64, respectively.

Your program must take x , y , and z from stdin and output the result of $\text{bitbin}(x,y,z)$.

Example:

```
./bitbin
```

```
314
```

```
31415926535897
```

```
6469693230
```

```
7637371712902
```

2.2 Grading Criteria

Pts	Metric
15	Correctness (i.e. "does it work?")
5	Source code file size (Smaller is better)

3 Break This Protocol

3.1 Problem Statement

Professors Amber and Bart at the University of Tejas have developed a new key exchange protocol that allows a server and its clients to exchange a symmetric key. They've decided to use this protocol to enable the two parties to communicate securely by encrypting their conversation using a shared key with AES encryption. The protocol is described below. After the protocol has been invoked, the server A and client B will communicate by encrypting their messages using AES with the shared key K_{ab} .

Description: Server A tries to exchange a shared key K_{ab} with with client B.

A: server B: client

K_a, K_b : A and B's public key (RSA Cipher)

K_{ab} : Key shared by A and B

$\{r\}_{K_b}, \{q\}_{K_b}$: r and q encrypted using RSA with B's public key

$\{K_{ab}\}_r$: Client B and server A's shared key encrypted using AES with symmetric key r

0. B->A : B // B starts communication with A
1. A->B : $\{r\}_{K_b}, \{K_{ab}\}_r$ // Server sends shared key K_{ab} encrypted with nonce r to client
// Server also sends the nonce encrypted with B's public key
2. B->A : $\{Z\}_{K_a}$ // B tries to verify A by sending a challenge nonce Z
3. A->B : $\{Z\}_{K_b}$ // A responds to B's challenge by decrypting $\{Z\}_{K_a}$ and encrypting
// Z with B's public key
4. B->A : $\{q\}_{K_a}, \{K_{ab}\}_q$ // B verifies to A that he received K_{ab} (See above)

After analyzing this protocol, Dr. Goodcard has concluded it is broken and that an attacker can recover shared keys. You've been hired by Dr. Goodcard to design a proof of concept that supports his belief. He has provided you with the output of a previous key exchange conversation between the server and Dr. Bart along with an account on the server named 'charlie'. Your job is to decrypt the secret message in the SECRET file by obtaining $K_{a,bart}$. Dr. Goodcard has provided you with Java source files he obtained from Dr. Amber and recommends you use these files to extract the shared key.

3.2 Build

The source files provided by Dr. Amber use Apache ANT as its build system. To build the project using ant, run *ant server* or *ant client*.

3.3 Files

- SECRET This file contains a message encrypted with symmetric key $K_{a,bart}$. It is your job to decrypt this message and submit it as your solution
- SAMPLE_EXECUTION This file contains a previous run executed between Dr. Bart and the server.

3.4 Grading Criteria

Pts	Metric
15	Correctness
5	Submission time (sooner is better)

4 Maze

4.1 Problem Statement

Hans Reiser was looking for his car seat when he stumbled into a four-dimensional maze. Being the computer genius that he is, he made a map of the maze and encoded it into a file using his iPhone. However, he forgot to install the SDK onto it before he got stuck in the maze, so he can't write a program to solve it himself. Instead, he sent it to you to tell him how to get out. Thankfully, he also included a description of the encoding he used:

“OK, I know the maze is a 4-dimensional hypercube (every side is the same length). Now, each byte in the file represents one point in the maze. I am at the point represented by the first byte in the file, and I think the last byte represents the point I need to get to to get out of here (I could only see it from other points). Being a four dimensional maze, every point in the file has eight possible movements away from it (two per dimension). Each bit in the byte represents whether I can move in that direction from that point. For example, we will enumerate the directions as follows:

North, South,
East, West,
Up, Down,
Right, Left

A point will be encoded as 0b10110010 if, from that point, I can move North, but not South, East or West, and Right, but not Up, Down, or Left. Now, I wrote them out in the file as though they were generated using the following code:

```
for(a=0; a < maze.side_length; a++)
  for(b=0; b < maze.side_length; b++)
    for(c=0; c < maze.side_length; c++)
      for(d=0; d < maze.side_length; d++)
        fprintf(mazefile, "%c", maze[a][b][c][d]);
```

Please help me out!”

That Hans Reiser! .. Does it every time. Ok, well using his encoding above, help him out of the maze. Have your solver tell you the directions he needs to escape; use only the first letter of the directions (N for North, D for Down, etc.) A sample maze and solution have been provided.

4.2 Grading Criteria

Pts	Metric
15	Correctness
5	Execution time

5 Zombie Administrators!

5.1 Problem Statement

On December 20, 2012 a malicious worm known as AmILegend? began to spread across a network of computers located in Boston. The malicious worm only affected systems on the top floor of buildings and unexplainably converted the system administrators of these systems into zombies. These zombie admins eventually became passive members in society as they lacked the skills necessary to operate the elevators in the buildings they managed.

When the zombie admins came to work, they always started on the ground floor and rode the elevator to the top floor, never going in the down direction; however, the zombies could randomly stop at floors between their entry and exit levels. You've been hired by the Umbrella Firm to investigate this strange behavior and have been asked to write a program that outputs the number of unique combinations of stops a zombie can make given the number of floors in the building.

Your program should take in the number of floors in a building from stdin and output the number of trips to stdout. The program should exit once a q is passed into stdin.

For example:

```
test_data.txt
```

```
-----
```

```
59
```

```
31
```

```
91
```

```
q
```

Sample Execution

```
-----
```

```
./zombie < test_data.txt
```

```
xxxxxxxxxxxxxxxxxxxxxx
```

```
xxxxxxxxxx
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxx
```

The Umbrella Firm has decided it will only provide you with buildings that have more than five but less than 112 floors.

5.2 Grading Criteria

Pts	Metric
15	Correctness
5	Source code file size (Smaller is better)

6 CPU Simulator

6.1 Problem Statement

For this problem, you are given an ISA specification and a few test programs both in machine code and assembly language. Your task is to write an instruction-accurate simulator for a processor implementing this ISA. The details about how you go about solving this problem are up to you, so long as you completely implement the ISA. Our supplied test cases will *not* be comprehensive, especially compared to what we'll actually be grading your submission with. You will not be supplied with an assembler.

You must submit your source code for the simulator. Simply hand-executing the code will be tedious, error-prone, and won't get you any points. Your simulator must accept any number of arguments on the command line which will all be programs to be loaded into memory. This programs will be specified as follows :

```
0x2000 // the starting memory location
0x1830 // instruction #1
0x2900 // instruction #2
0xB202 // instruction #3
...
```

and so on and so forth. No comments will be placed in the .obj file; they are simply in the above example for explanatory purposes. The PC will be initialized to the starting memory address of the first .obj file specified. You might want to use a language with decent bit-banging abilities built in. C and C++ both come to mind.

See the task.obj and task.asm files. **The “answer” to this problem, aside from a working simulator, is the output of the simulator when the accompanying test1.obj is executed.** Although an assembly file will be distributed, your simulator **must** accept object files as a command line argument(s). The .asm file is only provided for your convenience.

6.1.1 Binary-Coded Decimal

Binary-Coded Decimal (BCD) is a once-popular means of inefficiently representing values in a format convenient to be displayed to users, especially via things like seven-segment displays. The basic idea is that rather than representing a given decimal number (e.g. 123_{10}) in binary the “normal” way (e.g. $0b0000\ 0000\ 0111\ 1011$, or $0x007B$) we instead give each decimal digit its own bit field. Four bits is sufficient for each decimal digit, so we can pack four decimal digits into one 16b register (e.g. $0b0000\ 0001\ 0010\ 0011$, or $0x0123$).

To do BCD arithmetic, each arithmetic operation must be applied to corresponding BCD digits (in 4-bit chunks) modulo 10 with the corresponding base-10 carry.

example 1:

```

          1
    42      0100 0010
+ 23      + 0010 0011
-----
    65      0110 0101
```

example 2: (with BCD carry in parentheses)

```

          11  11
    11 <---lol win--> (1) (1)
    99      0000 1001 1001
+ 23      + 0000 0010 0011
-----
    122      0001 0010 0010
```

Traditionally, BCD implementations have ways to represent signed quantities. For the purposes of this exercise, assume all BCD values are positive/unsigned. You do not have to worry about negative quantities.

6.2 The ISA

6.2.1 The Architecture

Some good information to have about the machine in question:

- 64k memory space (that is, 16b address space)
- Big Endian
- 16b instructions
- 2-address machine
- condition flags: negative (N), zero (Z), positive (P), carry (C), signed overflow (V), half-carry (H)
- The condition flags are initialized to zero, except for Z equals 1.
- 8 registers (R0-R7)

6.2.2 The Instructions

Cat.		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	BR	0	0	0	0	N	Z	P	C	V	H	Offset 6					
1 #	ADD	0	0	0	1	0	Dest Reg			0	0	0	0	0	Src Reg		
1 #	ADD	0	0	0	1	1	Dest Reg			Imm 8							
1 #	AND	0	0	1	0	0	Dest Reg			0	0	0	0	0	Src Reg		
1 #	AND	0	0	1	0	1	Dest Reg			Imm 8							
1 #	XOR	0	0	1	1	0	Dest Reg			0	0	0	0	0	Src Reg		
1 #	XOR	0	0	1	1	1	Dest Reg			Imm 8							
1	KBRD	1	1	1	1	0	Dest reg			0	0	0	0	0	0	0	0
1	DISP	1	1	1	1	0	Src reg			0	0	0	0	0	0	0	1
1	HALT	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	JMP	0	1	1	1	0	Addr Reg			Off 8							
1	RET	0	1	1	1	0	1	1	1	0	0	0	0	0	0	0	0
1 #	SHF	0	1	1	0	A	Dest Reg			D	Src Reg			Unsigned 4			
1	BSR	1	0	0	0	Offset 12											
2 #	LD	0	1	0	1	0	Dest reg			S*	Imm 7						
2	ST	0	1	0	1	1	Src Reg			S*	Imm 7						
2 #	LDR	1	1	0	0	S*	Dest reg			Addr Reg			Imm 5				
2	STR	1	1	0	1	S*	Src reg			Addr Reg			Imm 5				
3	PSH	0	1	0	0	0	Src Reg			0	0	0	0	0	Stack Ptr		
3 #	POP	0	1	0	0	1	Dest Reg			0	0	0	0	0	Stack Ptr		
3	CSWP	1	0	1	1	N	Z	P	C	V	Reg 1			H	Reg 2		
4	BCDTB	1	1	1	0	1	Reg			0	0	0	0	0	0	0	0
4	BCDTB	1	1	1	0	1	Reg 1			1	Reg 2			0	0	0	0
4	BCDTB	1	1	1	0	1	Reg 1			1	Reg 2			1	Reg 3		
4 #	BCDA	1	0	0	1	0	Dest Reg			0	0	0	0	0	Src Reg		
4 #	BCDA	1	0	0	1	1	Dest Reg			Imm 8 (decimal)							
4	BTBCD	1	0	1	0	1	Reg			0	0	0	0	0	0	0	0
4	BTBCD	1	0	1	0	1	Reg 1			1	Reg 2			0	0	0	0
4	BTBCD	1	0	1	0	1	Reg 1			1	Reg 2			1	Reg 3		

```
# Modifies condition flags (N, Z, P, H, C, and/or V)
* S = Data Size; 1 = Word, 0 = Byte.
```

6.2.3 Instruction Behavior

BR

```
IF ((N && CURRENT_LATCHES.N) ||
    (Z && CURRENT_LATCHES.Z) ||
    (P && CURRENT_LATCHES.P) ||
    (C && CURRENT_LATCHES.C) ||
    (V && CURRENT_LATCHES.V) ||
    (H && CURRENT_LATCHES.H)) {
    Next_PC := PC* + sext(imm6 << 1);
}
```

ADD

```
Dest_Reg := Dest_Reg + Src_Reg;
setCC();
//(or)
Dest_Reg := Dest_Reg + sext(imm8);
setCC();
```

AND

```
Dest_Reg := Dest_Reg & Src_Reg;
setCC();
//(or)
Dest_Reg := Dest_Reg & sext(imm8);
setCC();
```

XOR

```
Dest_Reg := Dest_Reg ^ Src_Reg;
setCC();
//(or)
Dest_Reg := Dest_Reg ^ sext(imm8);
setCC();
```

HALT/KBRD/DISP

```
// halt
// PROTIP: just stop executing... hammertime/hammerzeit jokes may lead to disqualification.

// kbrd
scanf("%c", &Dest_Reg);

// disp
printf("%c", Src_Reg);
```

JMP/RET

```
//ret is just jmp with Src_Reg = R7, Off8 = 0
PC := Src_Reg + (sext(Off8)<<1); // Updated 3/21 1:30pm
```

SHF

```
if (D == 0)
    Dest_Reg = Src_Reg << imm4;
else {
    if (A == 0)
        Dest_Reg = Src_Reg >> imm4,0;
    else
        Dest_Reg = Src_Reg >> imm4,SR[15];
}
setCC();
```

BSR

```
R7 := PC*
PC := PC* + (sext(Off12) << 1);
```

LD/ST

```
//ld
Dest_Reg := sext(mem[PC* + sext(imm7)]); // byte
Dest_Reg := mem[PC* + (sext(imm7) << 1)]; // word
setCC();
```

```
//st (updated 3/22 10:45pm)
mem[PC* + sext(imm7)] := LOW8(Src_Reg); // byte
mem[PC* + (sext(imm7) << 1)] := Src_Reg; // word
```

LDR

```
Dest_Reg := sext(mem[Addr_Reg + sext(imm5)]); //byte, updated 3/21 1:48PM
Dest_Reg := mem[Addr_Reg + (sext(imm5) << 1)]; //word
setCC();
```

STR

```
mem[Addr_Reg + sext(imm5)] := LOW8(Src_Reg); // byte
mem[Addr_Reg + (sext(imm5) << 1)] := Src_Reg; // word
// updated 3/22, 10:45PM
```

PSH/POP

```
//push (updated 3/21 9:43PM)
Stack_Ptr := Stack_Ptr - 2;
mem[Stack_Ptr] := Src_Reg;
```

```
//pop
Dest_Reg := mem[Stack_Ptr];
Stack_Ptr := Stack_Ptr + 2;
setCC();
```

CSWP

```
if ((N && CURRENT_LATCHES.N) ||
    (Z && CURRENT_LATCHES.Z) ||
    (P && CURRENT_LATCHES.P) ||
    (C && CURRENT_LATCHES.C) ||
    (V && CURRENT_LATCHES.V) ||
    (H && CURRENT_LATCHES.H)) {
    swap(Reg1, Reg2);
}
```

BCDTB

```
// see BCD primer above... RegN should hold 4 BCD digits
RegN := Binary(RegN); // N can be 1, 2, and/or 3...
```

BCDA

```
// same as ADD, except register operands are interpreted as containing
// binary-coded decimal values (4 digits per 16 bits). The imm8 should
// be treated as a regular _unsigned_ binary value. The result of the
// operation should be 4 digits encoded in 16b worth of BCD-goodness.
// Set the H (half-carry) flag accordingly.
```

```
Dest_Reg := Dest_Reg + Src_Reg;
setCC();
//(or)
Dest_Reg := Dest_Reg + imm8;
setCC();
```

BTBCD

```
// see BCD primer above... RegN should be an unsigned binary number
RegN := BinaryCodedDecimal(RegN); // N can be 1, 2, and/or 3...
```

6.3 Grading Criteria

Pts	Metric
4	Implement All Category 1 instructions
4	Implement All Category 2 instructions
4	Implement All Category 3 instructions
4	Implement All Category 4 instructions
4	Style and Documentation