```
//-------------------------------------------------------------------------------
--------------
//File:   Branch.cs
//Desc:   This file defines a class Branch that contains logic for the Branch in
structions.
//-------------------------------------------------------------------------------
---------------

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    public class Branch : Instruction
    {
        public List<string> strInst = new List<string>(){"b", "bl", "bx" };

        public const int TYPE = 0b100;
        public uint cond, typ, L, X, Rm = 18;
        public int PC_SOffset, indx = 0;

        //like will be a store of
        public override void DecodeInst() {
            cond = Memory.ExtractBits(Inst, 0, 3);
            typ = Memory.ExtractBits(Inst, 4, 6);

            L = Memory.ExtractBits(Inst, 7, 7);

            if(typ == 0)
            {
                Rm = Memory.ExtractBits(Inst, 28, 31);
                indx = 2;
            }
            else
            {
                PC_SOffset = (((int)(Memory.ExtractBits(Inst, 8, 31) << 8)) >> 8
) << 2;
                indx = (int)L;
            }
        }

        public virtual void Execute(int Rm) { }
        public override void Execute() {
            List<Branch> binst = new List<Branch>() { new B(), new BX() };

            DecodeInst();

            Branch br = binst[indx > 1 ? 1: 0];
            br.L = L;
            br.I_Reg = I_Reg;
            br.I_RAM = I_RAM;
            br.InstAddr = InstAddr;

            if(indx > 1){ br.Execute((int)Rm); }
            else { br.Execute(PC_SOffset); }
        }
        public override string ToString()
        {
            DecodeInst();

            ASMRepr = strInst[indx] + (Cond < 0b1110 ? CondSufx[Cond] : "") + "\t
" + (indx > 1 ? CPU.GetStrRegr(I_Reg, (int)Rm) :
                (InstAddr + 8 + PC_SOffset).ToString());

            return ASMRepr;
        }
    }
```



```
    // branches/branch & Link to offset
    class B: Branch
    {
        public override void Execute(int offset)
        {
            int pc = CPU.GetRegr(I_Reg, 15); //returns pc + 8;
            pc -= 4;    //this might be a problem
            if (L == 1)
                CPU.SetReg(I_Reg, 14, pc);
            CPU.SetReg(I_Reg, 15, ((int)InstAddr + 8 + offset));
        }
    }

    //branches to RM address pointed to.
    class BX: Branch
    {
        public override void Execute(int Rm)
        {
            uint rm0 = (uint)CPU.GetRegr(I_Reg, Rm) & 0xFFFFFFFE;
            CPU.SetReg(I_Reg, 15, (int)(rm0));
        }
    }

}
```

```
//-------------------------------------------------------------------------------
--------------
//File:   Computer.cs
//Desc:   This file defines a class Computer which contains all the logic and co
mponents of the
//        ARM simulator.
//-------------------------------------------------------------------------------
---------------


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.ComponentModel;
using System.Security.Cryptography;
using System.Windows.Media;
using System.IO;
using System.Runtime.InteropServices;
using System.CodeDom.Compiler;

namespace armsim
{
    //Class that puts it all together
    public class Computer : INotifyPropertyChanged
    {
        /// <summary>
        /// Define logic for Computer simulation
        /// </summary>
        ///

        public const int MB_Size = 32768;
        public event PropertyChangedEventHandler PropertyChanged;

        private List<int> Breakpoints = new List<int>();

        //-----------------Tracing-------------------------
        public static StreamWriter Tracelog = null;
        public string[] modes = { "SVC", "SYS", "IRQ" };
        //-------------------------------------------------

        //--------------------RAM, Registers, CPU-----------------------------
------
        int reg_num = 23 * 4; //16 norms + 1 CPSR(16), 1 SP_irq(17), 1 LR_irq(18
), 1 SPSR_irq(19), 1 SP_svc(20), 1 LR_svc(21), 1 SPSR_svc(22)

        private Memory ram;
        private Memory registers;
        private CPU cpu;

        public Memory CompRAM { get { return ram; } set { ram = value; } }
        public Memory Registers { get { return registers; } set { registers = va
lue; } }
        public CPU CompCPU { get { return cpu; } set { cpu = value; } }
        //-------------------------------------------------------------------
-------


        //-------------------Status Bar: Filename, Program Status, CheckSum of R
AM----------------------------
        private string progName;
        private string progStatus;
        private int sum;

        public string ProgName { get { return progName; } set { progName = value
; SetProperty("ProgName"); } }
        public string ProgStatus { get { return progStatus; } set { progStatus =
 value; SetProperty("ProgStatus"); } }
        public int SumRAM { get { return sum; } set { sum = value; SetProperty("
SumRAM"); } }
```



```
        //-------------------------------------------------------------

        //--------------------Bindings for CPSR State-----------------
        public Dictionary<uint, string> proc_modes = new Dictionary<uint, string
>{
            { 0b10011, "Supervisor" }, { 0b11111, "System" }, { 0b10010, "IRQ" } };
        private uint Curr_Mode;
        public string Proc_Mode { get { return proc_modes[Curr_Mode]; } set { Se
tProperty("Proc_Mode"); } }

        public bool N_Flag { get { return registers.TestFlag(16 * 4, 0); } set {
 SetProperty("N_Flag"); } }
        public bool Z_Flag { get { return registers.TestFlag(16 * 4, 1); } set {
 SetProperty("Z_Flag"); } }
        public bool C_Flag { get { return registers.TestFlag(16 * 4, 2); } set {
 SetProperty("C_Flag"); } }
        public bool V_Flag { get { return registers.TestFlag(16 * 4, 3); } set {
 SetProperty("V_Flag"); } }
        public bool I_Flag { get { return registers.TestFlag(16 * 4, 24); } set
{ SetProperty("I_Flag"); } }

        //For toolbar and trace.
        public bool running = false, trace_closed = false, trace = true;
        public bool Running { get { return !running; } set { SetProperty("Running
"); } }

        public bool Enable_Trace { get { return trace; } set { trace = value; Se
tProperty("Enable_Trace"); } }
        public bool CompTraceall { get; set; }
        public int Step_Cnt = 1;

        //-------------Console-----------------------------
        public List<char> Input_Buffer = new List<char>();
        public StringBuilder ConsoleBuilder = new StringBuilder();
        public string Comp_Console { get { return ConsoleBuilder.ToString(); } s
et { SetProperty("Comp_Console"); } }

        // -------------------------------------------------------------------
---------------

        public Computer(int size, string filename){
            ram = new Memory(size);
            registers = new Memory(reg_num);
            cpu = new CPU(ram, registers);

            cpu.CPU_Console_Ref = ConsoleBuilder;
            cpu.CPU_Input_Buff = Input_Buffer;

            ProgName = filename == null ? "(None)" : filename ;

            Enable_Trace = true;
            Tracelog = File.CreateText(Directory.GetCurrentDirectory() + "\\trace.lo
g");
        }

        //-------------------------------------------------------------------
--------------------
        // perform fde cycle until fetch or breakpoint encountered returns 0
        public void Run() {
            uint val;

            try
            {
                Running = true;
                do
                {
                    if (Breakpoints.Contains(CompCPU.PC)) { break; }
                    val = Step();

                } while (val != 0);
```

```
            }
            catch (Exception) { }
            finally { ; }
            running = false;
            Running = true;
        }

        //For updating Flags and Console
        void Update_FlagsMode()
        {
            N_Flag = true;
            Z_Flag = true;
            C_Flag = true;
            V_Flag = true;
            I_Flag = true;

            Curr_Mode = Memory.ExtractBits((uint)CompCPU.CPSR, 27, 31);
            Proc_Mode = Curr_Mode.ToString();
            Comp_Console = "";
        }

        // 1 fde cycle
        public uint Step() {

            uint val = (uint)CompCPU.fetch();
            int pc = CompCPU.PC;

            Instruction inst = CompCPU.decode(val);
            try
            {
                CompCPU.execute(inst);     //Should Execute be in Task.Run()? be
cause of loop
            } catch (OperationCanceledException) {
                val = 0;
            }
            finally {; }

            if (val != 0 && CompCPU.IRQ && !Registers.TestFlag(16 * 4, 24))
            {
                CompCPU.Do_IRQProcessing();
                CompCPU.IRQ = false;

            }
            Update_FlagsMode();
            Trace(pc);
            if(val == 0) { Enable_Trace = false; }
            return val;
        }

        //Zeroes out Registers and Memory
        public void Reset()
        {
            CompRAM = new Memory(MB_Size);
            Registers = new Memory(reg_num);

            CompCPU = new CPU(ram, registers);
            CompCPU.CPU_Console_Ref = ConsoleBuilder;
            Input_Buffer.Clear();

            if(ConsoleBuilder.Length != 0)
            {
                ConsoleBuilder.Append("\n");
            }

            CompCPU.CPU_Input_Buff = Input_Buffer;

            CompCPU.SP = 0x7000;
            CompCPU.CPSR = 0x13;

            Step_Cnt = 1;
```



```
            if (!trace_closed)
            {
                Tracelog.Flush(); Tracelog.Close();
                trace_closed = true;
            }


            if (Enable_Trace && trace_closed) {
                Tracelog = File.CreateText(Directory.GetCurrentDirectory() + "\\tr
ace.log");
                trace_closed = false;
            }

        }

        //Adjusts if no OS loaded
        public void Adjust_Reset()
        {
            if(CompRAM.ReadByte(0) != 0)
            {
                CompCPU.PC = 0;
            }
            else
            {
                CompCPU.CPSR = 0x1F;
            }

            Update_FlagsMode();
        }

        //Add a breakpoint address to a list of breakpoints if not already conta
ined.
        public void AddBreakP(int addr)
        {
            if (!Breakpoints.Contains(addr)) { Breakpoints.Add(addr); }
        }

        //-------------------------------------------------------------------------
---------------------------------
        public void Trace(int pc)
        {
            string format = "{0:000000} {1:X8} {2:X8} {3} {19} 0={4:X8} 1={5:X8} 2={6:X8} 3={7:
X8} 4={8:X8} 5={9:X8} 6={10:X8}" +
                    " 7={11:X8} 8={12:X8} 9={13:X8} 10={14:X8} 11={15:X8} 12={16:X8} 13={17:X8} 14
={18:X8} ";

            int[] regs = new int[15];
            for(int i = 0; i < 15; ++i) { regs[i] = CPU.GetRegr(Registers, i);
}

            if (Enable_Trace)
            {
                if (trace_closed) {
                    Tracelog = File.CreateText(Directory.GetCurrentDirectory() +
 "\\trace.log");
                    trace_closed = false;
                }
                uint intcpsr = Memory.ExtractBits((uint)CompCPU.CPSR, 0, 3);
                uint mode = Memory.ExtractBits((uint)CompCPU.CPSR, 27, 31);
                string cpsr = Convert.ToString(intcpsr, 2).PadLeft(4, '0');

                //Check for traceall flag
                if(CompTraceall || mode == 0x1F)
                {
                    int indxMode = (mode == 0x1F) ? 1 : (mode == 0x12 ? 2 : 0);
                    Tracelog.WriteLine(format, Step_Cnt, pc - 4, CompRAM.CheckSu
m(CompRAM.Cells), cpsr, regs[0], regs[1],
                                regs[2], regs[3], regs[4], regs[5], regs[6], regs[7], re
gs[8], regs[9], regs[10], regs[11],
```

```
                        regs[12], regs[13], regs[14], modes[indxMode]);
                    Tracelog.Flush();
                    ++Step_Cnt;
                }

            }
            else
            {
                if (!trace_closed)
                {
                    Tracelog.Flush();
                    Tracelog.Close();
                    trace_closed = true;
                }
            }
        }

        //Event handler for the ProperyChanged event Notifying the object bound
of the change in the source in its parameters.
        protected void SetProperty(string source)
        {
            PropertyChangedEventHandler handle = PropertyChanged;
            if(handle != null)
            {
                PropertyChanged(this, new PropertyChangedEventArgs(source));
            }
        }
    }
}
```



```
//-------------------------------------------------------------------------------
--------------
//File:   CPU.cs
//Desc:   This file defines a class CPU which defines all the logic for the CPU
actions and
//          components.
//-------------------------------------------------------------------------------
---------------


using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace armsim
{
    public class CPU
    {
        /// <summary>
        /// contains logic for CPU class.
        /// </summary>
        public Memory CPU_Registers;//{ get; set; }
        public Memory CPU_RAM; //{ get; set; }

        public StringBuilder CPU_Console_Ref;
        public List<char> CPU_Input_Buff;

        public bool IRQ { get; set; }
     // need to change the return true statement

        public CPU(Memory ram, Memory reg)
        {
            CPU_RAM = ram;
            CPU_Registers = reg;
        }

        //Read word from RAM by val in PC reg
        public uint fetch() {
            uint val = (uint)CPU_RAM.ReadWord(PC);
            PC += 4;
            return val;
        }

        // Create an instruction
        public Instruction decode(uint instr) {
            Instruction inst = Instruction.CreateInstr(instr, CPU_Registers, CPU
_RAM);

            inst.InstAddr = (uint)(PC - 4);
            inst.I_Console_Ref = CPU_Console_Ref;
            inst.I_Input_Buff = CPU_Input_Buff;

            return inst;
        }

        // Pause 1 quarter second
        public void execute(Instruction instr) {
            if(CompCond(instr.Cond, (uint)CPSR))
                instr.Execute();                    //will need to test Flag ins
tead.
        }

        //Test flags for Conditional execution
        bool CompCond(uint cond, uint flags) //will need to come fix this to use
 TestFlag instead
        {
```

```
            bool[] compared = {
                (Memory.ExtractBits(flags, 1, 1) == 1), (Memory.ExtractBits(flag
s, 1, 1) == 0), //eq, ne
                (Memory.ExtractBits(flags, 2, 2) == 1), (Memory.ExtractBits(flag
s, 2, 2) == 0), //cs, cc
                (Memory.ExtractBits(flags, 0, 0) == 1), (Memory.ExtractBits(flag
s, 0, 0) == 0), //mi, pl
                (Memory.ExtractBits(flags, 3, 3) == 1), (Memory.ExtractBits(flag
s, 3, 3) == 0), //vs, vc

                (Memory.ExtractBits(flags, 2, 2) == 1) && (Memory.ExtractBits(fl
ags, 1, 1) == 0), //hi
                (Memory.ExtractBits(flags, 2, 2) == 0) || (Memory.ExtractBits(fl
ags, 1, 1) == 1), //ls

                (Memory.ExtractBits(flags, 0, 0) == Memory.ExtractBits(flags, 3,
 3)), //ge
                (Memory.ExtractBits(flags, 0, 0) != Memory.ExtractBits(flags, 3,
 3)), //lt

                ((Memory.ExtractBits(flags, 1, 1) == 0) && (Memory.ExtractBits(f
lags, 0, 0) == Memory.ExtractBits(flags, 3, 3))), //gt
                ((Memory.ExtractBits(flags, 1, 1) == 1) || (Memory.ExtractBits(f
lags, 0, 0) != Memory.ExtractBits(flags, 3, 3)))  //le
            };

            if (cond < 14)
                return compared[cond];

            return true;
        }

        //Calls Exception Processing with the correct mode.
        public void Do_IRQProcessing()
        {
            Exception_Process(CPU_Registers, 0b10010);
        }

        //Processes all exceptions
        public static void Exception_Process(Memory reg, uint mode)
        {
            uint cpsr = (uint)GetRegr(reg, 16);
            int pc = GetRegr(reg, 15);

            pc -= 4;

            int modespsr = mode == 0b10010 ? 19 : 22;
            int vector_addr = mode == 0b10010 ? 0x18 : 0x8;

            SetReg(reg, modespsr, (int)cpsr); //save to spsr_mode;
            cpsr = ((cpsr >> 5) << 5) | mode; //Change mode bits to mode

            SetReg(reg, 16, (int)cpsr); //changed the order to do updating mode
bits first
            SetReg(reg, 14, pc); //then update lr_mode

            reg.SetFlag(16 * 4, 24, true);  //set I-bit in CPSR
            SetReg(reg, 15, vector_addr); //set PC to address in table
        }

        //finds the offset in the registers array based on the current mode.
        static int getoffset(uint mode, int num)
        {
            int[] banked = { 13, 14 };
            //checking for mode requested and asking which one
            if (mode == 0b10011)
            {
                num = (banked.Contains(num)) ? num + 7 : num;

            }else if (mode == 0b10010)
```



```
            {
                num = (banked.Contains(num)) ? num + 4 : num;
            }
            return num;
        }

        //for getting register values;
        public static int GetRegr(Memory reg, int num){
            uint mode = Memory.ExtractBits((uint)reg.ReadWord(16 * 4), 27, 31);
            num = getoffset(mode, num);

            return num == 15 ? reg.ReadWord((num * 4)) + 4 :  reg.ReadWord((num
* 4));
        }

        //for setting registers
        public static void SetReg(Memory reg, int nreg, int val){
            uint mode = Memory.ExtractBits((uint)reg.ReadWord(16 * 4), 27, 31);
            nreg = getoffset(mode, nreg);

            reg.WriteWord(val, (nreg * 4));
        }

        //--------------------For string processing of registers----------------
---------
        static List<string> regs = new List<string>()
        {
            "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8",
            "r9", "r10", "r11", "ip", "sp", "lr", "pc", "CSPR",
            "sp_irq", "lr_irq", "SPSR_irq", "sp_svc", "lr_svc", "SPSR_svc"
        };

        public static string GetStrRegr(Memory reg, int ind)
        {
            uint mode = Memory.ExtractBits((uint)reg.ReadWord(16 * 4), 27, 31);
            ind = getoffset(mode, ind);

            return regs[ind];
        }

        public static string GetStrRegr(int ind)
        {
            return regs[ind];
        }

        //Register Properties
        public int IP { get { return CPU_Registers.ReadWord(0x30); } set { CPU_R
egisters.WriteWord(value, 0x30); } }
        public int SP { get { return CPU_Registers.ReadWord(0x34); } set { CPU_R
egisters.WriteWord(value, 0x34); } }
        public int R14 { get { return CPU_Registers.ReadWord(0x38); } set { CPU_
Registers.WriteWord(value, 0x38); } }
        public int PC { get { return CPU_Registers.ReadWord(0x3C); } set { CPU_R
egisters.WriteWord(value, 0x3C); } }
        public int CPSR { get { return CPU_Registers.ReadWord(0x40); } set { CPU
_Registers.WriteWord(value, 0x40); } }

    }
}
```

```
//-------------------------------------------------------------------------------
--------------
//File:   DataProcess.cs
//Desc:   This file defines a class DataPrcess with subclasses that contains log
ic for the
//           dataprocessing instructions.
//-------------------------------------------------------------------------------
--------------

using NUnit.Framework.Constraints;
using System;
using System.CodeDom;
using System.Collections.Generic;
using System.IO.Ports;
using System.Linq;
using System.Runtime.InteropServices;
using System.Security.Cryptography;
using System.Security.Policy;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    //INSTRUCTIONS: MOV(), MVN(), ADD(), SUB(), RSB(),
    // AND(), ORR(), EOR(), BIC(), MUL()

    //Offsets: cond = 0 - 3, type = 4-6(I=6), Opcode = 7-10, S = 11-11, Rn=12-15
, Rd=16-19, oper2 = 20-31,

    //all child class will have their own TYPE so DATA; In comments shift_operan
d refers to the Operand2 type
    public class DataProccess : Instruction
    {
        public List<string> strInst = new List<string>()
        {
            "and", "eor", "sub", "rsb", "add",
            null, null, null, null, null, "cmp", null,
            "orr", "mov", "bic", "mvn", "mul"
        };

        public const int TYPE = 0b0;

        public uint cond, typ, opcode, Rn, Rm, Rs, bit7, bit4;
        public bool regimm, sbit;
        public Operand2 Oper2;

        //See general definition in parent(Instruction)
        public override void DecodeInst() {
            cond    =   Memory.ExtractBits(Inst, 0, 3);
            typ     =   Memory.ExtractBits(Inst, 4, 6);
            bit7    =   Memory.ExtractBits(Inst, 24, 24);
            bit4    =   Memory.ExtractBits(Inst, 27, 27);

            if (typ == 0 &&  bit7 == 1 && bit4 == 1)
            {
                opcode = 0x10;                                  //for convenience, I
suppose.
                Rd = Memory.ExtractBits(Inst, 12, 15);
                Rs = Memory.ExtractBits(Inst, 20, 23);
                Rm = Memory.ExtractBits(Inst, 28, 31);
            }
            else
            {
                regimm = Convert.ToBoolean(Memory.ExtractBits(typ, 31, 31)); //t
rue == 1; false = 0;
                opcode = Memory.ExtractBits(Inst, 7, 10);
                sbit = Convert.ToBoolean(Memory.ExtractBits(Inst, 11, 11)); //sa
me as regimm;
                Rn = Memory.ExtractBits(Inst, 12, 15);
                Rd = Memory.ExtractBits(Inst, 16, 19);
```



```
                Oper2 = Operand2.GetOper2(regimm, Memory.ExtractBits(Inst, 20, 3
1));
                Oper2.Oper2Regs = I_Reg;
            }
        }

        //override for MUL execute.
        public virtual void Execute(uint Rd, uint Rm, uint Rs) {; }

        //override for execute which all subclasses, except for MUL, use.
        public virtual void Execute(uint Rn, uint Rd, Operand2 oper2) {; }

        //calls subclasses execute method that takes an operand2, registers
        public override void Execute() {
            List<DataProccess> dpinst = new List<DataProccess>()
            {
                new AND(), new EOR(), new SUB(), new RSB(), new ADD(),
                null, null, null, null, null, new CMP(), null,
                new ORR(), new MOV(), new BIC(), new MVN(), new MUL()
            };

            DecodeInst();
            DataProccess instR = dpinst[(int)opcode];
            instR.I_Reg = I_Reg;
            instR.sbit = sbit;

            if (opcode == 0x10){
                instR.Execute(Rd, Rm, Rs);
            } else {
                instR.Execute(Rn, Rd, Oper2);
            }
        }

        public override string ToString()
        {
            DecodeInst();

            ASMRepr = strInst[(int)opcode] + (sbit ? ((opcode != 10)? "s": ""):
"")+ (Cond < 0b1110 ? CondSufx[Cond] : "") +
                "\t" + CPU.GetStrRegr(I_Reg, (opcode == 10 ? (int)Rn : (int)Rd));

            if (strInst[(int)opcode] == "mul")
            {
                ASMRepr += "," + CPU.GetStrRegr(I_Reg, (int)Rm) + "," + CPU.GetS
trRegr(I_Reg, (int)Rs);//check on that

            } else if(new List<string>() { "mov", "mvn", "cmp" }.Contains(strIns
t[(int)opcode]))
            {
                ASMRepr += "," + Oper2.ToString();
            }
            else
            {
                ASMRepr += "," + CPU.GetStrRegr(I_Reg, (int)Rn) + "," + Oper2.To
String();
            }

            return ASMRepr;
        }
    }

    class CMP : DataProccess
    {
        public new const int TYPE = 0b1010;
        //Update flags after Rn - shifter_operand
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            uint rnval = (uint)CPU.GetRegr(I_Reg, (int)Rn), op2val = (uint)oper2
.GetValue();
            int val = (int)(rnval - op2val);
```

```
            uint N = Memory.ExtractBits((uint)val, 0, 0);
            int Z = val == 0 ? 1 : 0;
            int C = op2val <= rnval ? 1 : 0;  //May not be correctly handling C
flag will need to modify

            int rnv = (int)rnval, op2v = (int)op2val;
            val =  rnv - op2v;
            int V = (rnv > 0 && op2v < 0 && val < 0) || (rnv < 0 && op2v > 0 &&
val > 0) ? 1 : 0;

            bool[] flags = { Convert.ToBoolean(N), Convert.ToBoolean(Z), Convert
.ToBoolean(C), Convert.ToBoolean(V) };
            for(int i = 0; i < 4; ++i)
                I_Reg.SetFlag(4 * 16, i, flags[i]);
        }

    }

    class MOV : DataProccess
    {

        public new const int TYPE = 0b1101;
        //Rd := shifter_operand (no Rn)
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            CPU.SetReg(I_Reg, (int)Rd, oper2.GetValue());

            int mode = (int)Memory.ExtractBits((uint)CPU.GetRegr(I_Reg, 16), 27,
 31);
            if(sbit && Rd == 15)
            {
                int spsr = CPU.GetRegr(I_Reg, 16);
                if(mode == 0b10010)
                {
                    spsr = CPU.GetRegr(I_Reg, 19); //because I had to place them
 weirdly.
                    //CPU.SetReg(I_Reg, (int)Rd, oper2.GetValue() - 4); //adjust
 the LR for the PC when IRQ
                }
                else if (mode == 0b10011)
                {
                    spsr = CPU.GetRegr(I_Reg, 22);
                }

                CPU.SetReg(I_Reg, 16, spsr);
            }
        }
    }

    class MVN : DataProccess
    {
        public new const int TYPE = 0b1111;

        //Rd := NOT shifter_operand (no Rn)
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            CPU.SetReg(I_Reg, (int)Rd, ~oper2.GetValue());
        }

    }

    class ADD: DataProccess
    {
        public new const int TYPE = 0b0100;

        // Rd := Rn + shifter_operand
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            int val =  CPU.GetRegr(I_Reg, (int)Rn) + oper2.GetValue();
            CPU.SetReg(I_Reg, (int)Rd, val);
```



```
        }
    }

    class SUB: DataProccess
    {
        public new const int TYPE = 0b0010;

        //Rd := Rn - shifter_operand
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            int val = CPU.GetRegr(I_Reg, (int)Rn) - oper2.GetValue();
            CPU.SetReg(I_Reg, (int)Rd, val);
        }
    }

    class RSB: DataProccess
    {
        public new const int TYPE = 0b0011;

        //Rd := shifter_operand - Rn
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            int val =  oper2.GetValue() - CPU.GetRegr(I_Reg, (int)Rn);
            CPU.SetReg(I_Reg, (int)Rd, val);
        }
    }

    class MUL: DataProccess
    {
        //Rd := Rm * Rs;
        public override void Execute(uint Rd, uint Rm, uint Rs){
            int first = CPU.GetRegr(I_Reg, (int)Rm);
            int second = CPU.GetRegr(I_Reg, (int)Rs);
            long res = (first * second) & 0xFFFFFFFF;
            CPU.SetReg(I_Reg, (int)Rd, (int)res);
        }
    }

    class AND: DataProccess
    {
        public new const int TYPE = 0b0000;

         // Rd := Rn AND shifter_operand(oper2)
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
          int val = CPU.GetRegr(I_Reg, (int)Rn) & oper2.GetValue();
          CPU.SetReg(I_Reg, (int)Rd, val);
        }
    }

    class ORR: DataProccess
    {
        public new const int TYPE = 0b1100;

        // Rd := Rn OR shifter_operand
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            int val = CPU.GetRegr(I_Reg, (int)Rn) | oper2.GetValue();
            CPU.SetReg(I_Reg, (int)Rd, val);
        }
    }

    class EOR: DataProccess
    {
        public new const int TYPE = 0b0001;

        //Rd := Rn EOR shifter_operand
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            int val = CPU.GetRegr(I_Reg, (int)Rn) ^ oper2.GetValue();
```

```
            CPU.SetReg(I_Reg, (int)Rd, val);
        }
    }

    class BIC: DataProccess
    {
        public new const int TYPE = 0b1110;

        //Rd := Rn AND NOT(shifter_operand)
        public override void Execute(uint Rn, uint Rd, Operand2 oper2)
        {
            int val = CPU.GetRegr(I_Reg, (int)Rn) & (~oper2.GetValue());
            CPU.SetReg(I_Reg, (int)Rd, val);
        }
    }

    //-------------------Passed my IS A test----------------
    public class Operand2
    {
        public uint OperBits { get; set; }
        public Memory Oper2Regs { get; set; }
        public string Oper2_Repr{ get; set; }
        //public List<string> Oper2_Str_Regs { get; set; }

        private string[] repr_shifts = new string[] { "lsl", "lsr", "asr", "ror" };
        public string[] Repr_shifts { get { return repr_shifts; } }

        //creates an Operand2 object for the DP instruction based on the operand
 type bits to use.
        public static Operand2 GetOper2(bool regimm, uint bits)
        {
            Operand2 oper2;
            if (regimm)
            {
                oper2 = new Oper2_RORImm();
            }
            else if (Convert.ToBoolean(Memory.ExtractBits(bits, 27, 27)))
            {
                oper2 = new Oper2_RegSReg();
            }else
            {
                oper2 = new Oper2_RegSImm();
            }

            oper2.OperBits = bits;
            return oper2;
        }

        //Uses BarrelShift to get value for Operand2
        public virtual int GetValue() { return 0; }
        public override string ToString() { return ""; }
    }

    public class Oper2_RegSReg : Operand2
    {
        public int reg, reg2;
        public uint shift;

        public override int GetValue()
        {
            reg = (int)Memory.ExtractBits(OperBits, 28, 31);
            shift = Memory.ExtractBits(OperBits, 25, 26);
            reg2 = (int)Memory.ExtractBits(OperBits, 20, 23);

            return (int)BarrelShift.Compute(shift, (uint)CPU.GetRegr(Oper2Regs,
reg), (uint)CPU.GetRegr(Oper2Regs, reg2));
        }

        public override string ToString()
        {
```



```
            GetValue();
            return CPU.GetStrRegr(reg) + "," + Repr_shifts[shift] + " " + CPU.Ge
tStrRegr(reg2);
        }
    }

    public class Oper2_RegSImm : Operand2
    {
        public uint reg, imm, shift;
        public override int GetValue()
        {
            reg = Memory.ExtractBits(OperBits, 28, 31);
            shift = Memory.ExtractBits(OperBits, 25, 26);
            imm = Memory.ExtractBits(OperBits, 20, 24);

            return (int)BarrelShift.Compute(shift, (uint)CPU.GetRegr(Oper2Regs,
(int)reg), imm);
        }

        public override string ToString()
        {
            GetValue();
            if(imm > 0)
            {
                return CPU.GetStrRegr((int)reg) + "," + Repr_shifts[shift] + "#
" + imm.ToString();
            }
            else
            {
                return CPU.GetStrRegr((int)reg);
            }
        }

    }

    public class Oper2_RORImm : Operand2
    {
        uint rot, num;

        public override int GetValue()
        {
            rot = Memory.ExtractBits(OperBits, 20, 23) * 2;
            num = Memory.ExtractBits(OperBits, 24, 31);

            return (int)BarrelShift.Compute(0x11, num, rot);
        }

        public override string ToString()
        {
            return "#" + GetValue().ToString();
        }
    }

    //---------------------Does not pass IS A test, therefore the LSL, LSR, ASL,
 ASR are functions of BarrelShift------------------------------
    public class BarrelShift
    {
        //based on the bitpattern of code, do bitwise operations and return the
results.
        public static uint Compute(uint code, uint toShift, uint displcmnt)
        {
            switch (code)
            {
                case 0:  // lsl
                    return (toShift << (int)displcmnt);
                case 1:  // lsr
                    return (toShift >> (int)displcmnt);
                case 2: // asr
                    return (uint)((int)(toShift) >> (int)displcmnt);
                default: // ror
```

```
                    uint high = Memory.ExtractBits(toShift, (31 - displcmnt), 31
) << (int)(32 - displcmnt);
                    uint low = toShift >> (int)displcmnt;

                    //used in ROR_IMM Oper2
                    return (high | low);
                }
            }
        }
}
```



```
//-------------------------------------------------------------------------------
--------------
//File:   Instruction.cs
//Desc:   This file defines a class Instruction which is the base class for all
the instructions
//         implemented.
//-------------------------------------------------------------------------------
---------------


using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Windows.Input;

namespace armsim
{
    public abstract class Instruction
    {

        /// <summary>
        /// Defines base class for all Instructions
        /// </summary>

        public string[] CondSufx = {
             "eq", "ne", "cs", "cc", "mi", "pl", "vs",
             "vc", "hi", "ls", "ge", "lt", "gt", "le"
        };


        public uint Rd, Cond;                           //Destination Register com
mon for all instructions
        public string ASMRepr { get; set; }     //String represention of the ins
truction
        public uint Inst { get; set; }          //Holds the actual numerical val
ue of the instruction
        public uint InstAddr { get; set; }

        public Memory I_Reg { get; set; }                   //Memory reference f
or registers
        public Memory I_RAM { get; set; }                   //Memory reference f
or RAM
        public StringBuilder I_Console_Ref { get; set; }
        public List<char> I_Input_Buff { get; set; }
        public char I_Last_Char { get; set; }


        //Logic for which type of instruction to create and returns the Instruct
ion
        public static Instruction CreateInstr(uint instr, Memory reg, Memory ram
)
        {
            uint typebits = Memory.ExtractBits(instr, 4, 6);
            uint[] bx = { Memory.ExtractBits(instr, 7, 11), Memory.ExtractBits(i
nstr, 24, 27), Memory.ExtractBits(instr, 12, 23) };

            Instruction I_Instr;
            if(typebits == 0b111 && Memory.ExtractBits(instr, 7, 7 ) == 1){
                I_Instr = new SWI();

            } else if((typebits == 0b101) || (typebits == 0 && bx[0] == 18 && bx
[1] == 1 && bx[2] == 0xFFF)){
                I_Instr = new Branch();

            }else if(typebits > 0b1){
                I_Instr = new LoadStore();

            }else{
```

```
                uint sbit = Memory.ExtractBits(instr, 11, 11);
                uint opcode = Memory.ExtractBits(instr, 7, 10);

                if(sbit == 0 && opcode > 0b111 && opcode < 0b1100)
                {
                    uint bit10 = Memory.ExtractBits(instr, 10, 10);
                    I_Instr = (bit10 == 0b1) ? new MSR() as Instruction: new MRS
();
                }
                else
                {
                    I_Instr = new DataProccess();
                }

            }

            I_Instr.I_RAM = ram;
            I_Instr.I_Reg = reg;
            I_Instr.Inst = instr;
            I_Instr.Cond = Memory.ExtractBits(instr, 0, 3);

            return I_Instr;

        }

        //General defintion: Extracts the bits Executes needs to run and stores
them in variables.
        public abstract void DecodeInst();

        //General definition: Uses bits extracted by DecodeInstr to execute the
sub classes intructions
        public abstract void Execute();

        //General definition: returns ASMRepr - will be adjusting later to remov
e duplicates
        public override string ToString() {
            return ASMRepr;
        }

    }

    public class SWI: Instruction
    {
        /// <summary>
        /// Implements the SWI Arm Instruction Logic
        /// </summary>

        public uint cond, typ, swinum;  //condition bits, type bits, and swinumb
er
        public bool I_Reading { get; set; }
        //See general definition
        public override void DecodeInst() {
            typ = Memory.ExtractBits(Inst, 4, 7);
            swinum = Memory.ExtractBits(Inst, 8, 31);
        }

        //Simply throws an exception for the Step function to know SWI has execu
ted.
        public override void Execute() { //
            DecodeInst();

            if (swinum == 0x11)
            {
                throw new OperationCanceledException(); //halt
            }

            CPU.Exception_Process(I_Reg, 0b10011);
        }

        //See general definition
```



```
        public override string ToString() {
            DecodeInst();

            ASMRepr = "svc"+ (Cond < 0b1110 ? CondSufx[Cond] : "") + "\t0x" + swin
um.ToString("X8");
            return ASMRepr;
        }
    }

    //for saving status register
    class MRS : Instruction
    {
        //uint Rd;
        bool Rbit;
        public override void DecodeInst()
        {
            Rd = Memory.ExtractBits(Inst, 16, 19);
            Rbit = Convert.ToBoolean(Memory.ExtractBits(Inst, 9, 9));
        }

        public override void Execute()
        {
            DecodeInst();

            uint mode = Memory.ExtractBits((uint)CPU.GetRegr(I_Reg, 16), 27, 31)
;
            int which = 16;

            if (mode == 0b10010)
            {
                which = 19;
            }
            else if (mode == 0b10011)
            {
                which = 22;
            }

            CPU.SetReg(I_Reg, (int)Rd, CPU.GetRegr(I_Reg, (Rbit ? which : 16)));
        }

        public override string ToString()
        {
            DecodeInst();
            //Update StrReg to reflect the state for the
            ASMRepr = "mrs" + (Cond < 0b1110 ? CondSufx[Cond] : "") + "\t" + CPU.
GetStrRegr(I_Reg, (int)Rd) + "," + (Rbit ? "SPSR," : "CPSR,");
            return base.ToString();
        }
    }

    //For restoring status register
    class MSR: Instruction
    {
        public uint cond, Rm;
        public bool regimm, Rbit;  //Rbit tells me whether or not I am dealing w
ith SPSR or CPSR
        public uint imm_val;

        public override void DecodeInst()
        {
            cond = Memory.ExtractBits(Inst, 0, 3);
            Rbit = Convert.ToBoolean(Memory.ExtractBits(Inst, 9, 9));
            regimm = Convert.ToBoolean(Memory.ExtractBits(Inst, 6, 6));

            Rm = Memory.ExtractBits(Inst, 28, 31);

            imm_val = Memory.ExtractBits(Inst, 20, 31);
            Operand2 temp = new Oper2_RORImm() { OperBits = imm_val };
            imm_val = (uint)temp.GetValue();
        }
```

```
        public override void Execute()
        {
            DecodeInst();

            uint operand = regimm ? imm_val : (uint)CPU.GetRegr(I_Reg, (int)Rm);

            if (Rbit == false)
            {
                CPU.SetReg(I_Reg, 16, (int)operand);
            }
            else
            {
                uint mode = Memory.ExtractBits((uint)CPU.GetRegr(I_Reg, 16), 27,
 31);
                if(mode == 0b10010 || mode == 0b10011)
                {
                    int indx = mode == 0b10010 ? 19 : 22;
                    CPU.SetReg(I_Reg, indx, (int)operand);
                }
            }
        }

        public override string ToString()
        {
            DecodeInst();
            ASMRepr = "msr" + (Cond < 0b1110 ? CondSufx[Cond] : "") + "\t" + (Rbi
t ? "SPSR, " : "CPSR, ") +
                (regimm ? "#" + imm_val.ToString(): CPU.GetStrRegr(I_Reg, (int)R
m));
            return ASMRepr;
        }
    }


}
```



```
//-------------------------------------------------------------------------------
--------------
//File:   LoadStore.cs
//Desc:   This file defines a class LoadStore that contains logic for the LoadSt
ore instructions.
//-------------------------------------------------------------------------------
---------------


using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Runtime.InteropServices;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace armsim
{
    public class LoadStore : Instruction
    {
        /// <summary>
        /// Defines attributes and methods for LoadStore
        /// </summary>
        ///

        string[] StrInstr = new string[] { "str", "ldr", "strb", "ldrb", "stm", "ldm",
 "push", "pop" };

        public const int TYPE = 0b010;
        public uint typ, P, U, B, W, L, Rn; //Rd inherited
        public bool I;

        public Offset LsOffset;
        public List<int> Reglist;

        public int indx = 0;

       // Extracts the bits Executes needs to run and stores them in variables.
        public override void DecodeInst() {
            typ = Memory.ExtractBits(Inst, 4, 6);
            I = Convert.ToBoolean(Memory.ExtractBits(Inst, 6, 6));
            P = Memory.ExtractBits(Inst, 7, 7);
            U = Memory.ExtractBits(Inst, 8, 8);
            B = Memory.ExtractBits(Inst, 9, 9);
            W = Memory.ExtractBits(Inst, 10, 10);
            L = Memory.ExtractBits(Inst, 11, 11);
            Rn = Memory.ExtractBits(Inst, 12, 15);

            if(typ != 4)
            {
                Rd = Memory.ExtractBits(Inst, 16, 19);
                LsOffset = Offset.GetOffset(I, Memory.ExtractBits(Inst, 20, 31))
;
                LsOffset.Offset_Regs = I_Reg;
                LsOffset.U = (int)U;
            }
            else
            {
                Reglist = GetList(Inst);
                L += 4;          //----------------------------See comment below
about index
            }
        }

        public virtual void Execute(uint Rn, List<int> reglist) {; }
        public virtual void Execute(uint Rn, uint Rd, Offset offst) {; }

        public override void Execute() {
```

```
            DecodeInst();

            List<LoadStore> instr = new List<LoadStore>() {  new STR(), new LDR(
),  new STM(), new LDM() };

            LoadStore ls = instr[(int)(L < 4 ? L : L - 2)]; //------------------
--using L to index may not be the best idea-
            ls.I_RAM = I_RAM;
            ls.I_Reg = I_Reg;

            ls.I_Console_Ref = I_Console_Ref;
            ls.I_Last_Char = I_Last_Char;
            ls.I_Input_Buff = I_Input_Buff;

            ls.P = P; ls.U = U; ls.B = B; ls.W = W;

            if (typ != 4)
            {
                ls.Execute(Rn, Rd, LsOffset);
            }
            else
            {
                ls.Execute(Rn, Reglist);
            }
        }

        public override string ToString()
        {
            DecodeInst();                   //------FD(IA)---------------   ------
----FD(DB)----------
            bool pushpop = (Rn == 13) && ((P == 0 && U == 1 && W == 1) || (P ==
1 && U == 0 && W == 1));
            int indx = (int)(L < 4 ? (L + (B == 1 ? 2 : B)) : (L + (!pushpop ? 0
 : 2)));
            ASMRepr = StrInstr[indx] + (Cond < 0b1110 ? CondSufx[Cond] : "") + "
\t";

            if (typ != 4)
            {
                ASMRepr += CPU.GetStrRegr(I_Reg, (int)Rd) + ",[" + CPU.GetStrReg
r(I_Reg, (int)Rn) + LsOffset.ToString() + "]" + (W == 1 ? "!" : "");
            }
            else
            {
                string strReglist = "";
                for (int i = 0; i < Reglist.Count; ++i)
                    strReglist += (i == 0 ? "" : ",") + CPU.GetStrRegr(I_Reg, Re
glist[i]);

                ASMRepr += (pushpop ? "" : CPU.GetStrRegr(I_Reg, (int)Rn) + (W =
= 1 ? "!" : "") + ",") + "{" + strReglist + "}";
            }
            return ASMRepr;
        }

        public int GetEffAddr(uint rn, Offset offst)
        {
            long EA = (CPU.GetRegr(I_Reg, (int)rn) + (U == 1 ? offst.GetValue()
: -offst.GetValue())) & 0xFFFFFFFF;
            return (int)EA;
        }

        List<int> GetList(uint list)
        {
            List<int> reglst = new List<int>();

            for (int cnt = 0; cnt < 16; ++cnt)
            {
                if (Memory.ExtractBits(list, (uint)(31 - cnt), (uint)(31 - cnt))
 == 1)
```



```
                {
                    reglst.Add(cnt);
                }
            }

            return reglst;
        }
    }

    public class LDR: LoadStore
    {
        //Loads word from memory address:
        public override void Execute(uint Rn, uint Rd, Offset offst) {
            //LDR   <Rd>, <addressing_mode>
            int EA = GetEffAddr(Rn, offst);          //((int)Rn + (U == 1 ? offst
.GetValue(): -offst.GetValue())) & 0xFFFFFFFF;

            if(EA == 0x100001)
            {
                //check after check if input buff is empty
                I_Last_Char = I_Input_Buff.Count != 0 ? I_Input_Buff[I_Input_Buf
f.Count - 1] : (char)0;
                CPU.SetReg(I_Reg, (int)Rd, Convert.ToInt32(I_Last_Char));
            }
            else
            {
                if (B == 1)
                {
                    CPU.SetReg(I_Reg, (int)Rd, I_RAM.ReadByte((uint)EA));
                }
                else
                {
                    CPU.SetReg(I_Reg, (int)Rd, I_RAM.ReadWord(EA)); //Remember t
o check what memreads return if invalid EA
                }

                if (W == 1)
                    CPU.SetReg(I_Reg, (int)Rn, EA);
            }
        }
    }

    public class STR: LoadStore
    {
        //stores word to memory address:
        public override void Execute(uint Rn, uint Rd, Offset offst)
        {
            int EA = GetEffAddr(Rn, offst);         //((int)Rn + (U == 1 ? offst.
GetValue() : -offst.GetValue())) & 0xFFFFFFFF;

            if(EA == 0x100000)
            {
                I_Console_Ref.Append((char)CPU.GetRegr(I_Reg, (int)Rd));
            }
            else
            {
                if (B == 1)
                {
                    byte b = Convert.ToByte(CPU.GetRegr(I_Reg, (int)Rd) & 0xFF);
                    I_RAM.WriteByte(b, EA);
                }
                else {
                    I_RAM.WriteWord(CPU.GetRegr(I_Reg, (int)Rd), EA);
                }

                if (W == 1)
                    CPU.SetReg(I_Reg, (int)Rn, EA);
            }
        }
    }
```

```
    public class STM : LoadStore
    {
        //pushes register list onto stack
        public override void Execute(uint Rn, List<int> Reglist)
        {
            int EA = CPU.GetRegr(I_Reg, (int)Rn);
            EA -= (4 * Reglist.Count);

            if (W == 1)
                CPU.SetReg(I_Reg, (int)Rn, EA);

            foreach (int i in Reglist)
            {
                int val = CPU.GetRegr(I_Reg, i);
                I_RAM.WriteWord(val, EA);
                EA += 4;
            }
        }
    }

    //pops values from stack to registers in list
    public class LDM : LoadStore
    {
        public override void Execute(uint Rn, List<int> Reglist) {
            int EA = CPU.GetRegr(I_Reg, (int)Rn);

            foreach (int i in Reglist)
            {
                int val = I_RAM.ReadWord(EA);
                CPU.SetReg(I_Reg, i, val);
                EA += 4;
            }

            if (W == 1)
                CPU.SetReg(I_Reg, (int)Rn, EA);
        }
    }

    //Defines logic for getting Offset-------------------------------------
    public class Offset
    {
        public uint OffBits;
        public Memory Offset_Regs { get; set; }
        public string Offset_Repr { get; set; }
        public int U = 0;
        private string[] repr_shifts = new string[] { "lsl", "lsr", "asr", "ror" };
        public string[] Repr_shifts { get { return repr_shifts; } }

        //creates an Operand2 object for the DP instruction based on the operand
type bits to use.
        public static Offset GetOffset(bool regimm, uint bits)
        {
            Offset offst;
            if (regimm)
            {
                offst = new Offset_Reg();
            }
            else
            {
                offst = new Offset_Imm();
            }
            offst.OffBits = bits;
            return offst;
        }

        //Uses BarrelShift to get value for OffSet
        public virtual int GetValue() { return 0; }

        public override string ToString() { return ""; }
```



```
    }

    public class Offset_Imm : Offset
    {
        public override int GetValue()
        {
            return (int)OffBits;
        }

        public override string ToString()
        {
            int offbits = GetValue();
            return offbits > 0 ? ",#" + (U == 1 ? "" : "-") + offbits.ToString()
 : "";
        }
    }

    public class Offset_Reg : Offset
    {
        public uint reg, imm, shift;
        public override int GetValue()
        {
            reg = Memory.ExtractBits(OffBits, 28, 31);
            shift = Memory.ExtractBits(OffBits, 25, 26);
            imm = Memory.ExtractBits(OffBits, 20, 24);

            return (int)BarrelShift.Compute(shift, (uint)CPU.GetRegr(Offset_Regs
, (int)reg), imm);
        }

        public override string ToString()
        {
            GetValue();
            return "," + (U == 1 ? "" : "-") + CPU.GetStrRegr((int)reg) + (imm >
 0 ? "," + Repr_shifts[shift] + "#" + imm.ToString() : "");

        }
    }
}
```

```
//-----------------------------------------------------------------------------
--------------
//File:   Memory.cs
//Desc:   This file defines a class Memory that contains logic for representing
Computer Memory.
//-----------------------------------------------------------------------------
---------------


using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Diagnostics;
using System.Reflection.Emit;

namespace armsim
{
    public class Memory
    {
        //TraceSwitch memTrace = new TraceSwitch("MemoryTrace", "Switch for the
memory class");
        private byte[] ram;

        public byte[] Cells { get { return ram; } }  //property for the class to
 use.

        //Constructor for RAM Simulation
        public Memory(int size)
        {
            ram = new byte[size];
        }

        //Validates that address is appropriate; type: 0 =short(16), 1=word(32)
        bool IsVldAddr(int addr, int type)
        {
            if (type == 0) { return addr % 2 == 0; }
            return addr % 4 == 0;
        }

        // All 3 receive a 32-bit address and returns the number of bits request
ed that are currently in the address, or -1 for incorrect address
        public int ReadWord(int addr)
        { //Question: what happens when I read the end of the file? Should I val
idate that the address in the memory location(?)
            if (IsVldAddr(addr, 1))
            {
                return (int)((Cells[addr + 3] << 24) + (Cells[addr + 2] << 16) +
 (Cells[addr + 1] << 8) + Cells[addr]);
            }
            return -1;
        }
        public short ReadHalfWord(int addr)
        {
            if (IsVldAddr(addr, 0))
            {
                return (short)((Cells[addr + 1] << 8) + Cells[addr]);
            }
            return 0;
        }
        public byte ReadByte(uint addr) { return ram[addr]; } //Occured to me th
at this could be used to read bytes ;)

        // All 3 receive a 32-bit address and  the number of bits requested that
 are currently in the address, or -1 for incorrect address
        //as long as there is space, this ok, but that is not always so. Must fi
x when writing unit tests. -Sone
        public void WriteWord(int val, int addr)
        {
            if (IsVldAddr(addr, 1))
            {
                int op = 0x000000FF;
                int byt;
                for (int i = 0; i < 4; ++i)
                {
                    byt = (val >> (8 * i)) & op;
                    WriteByte(Convert.ToByte(byt), addr + i);
                }
            }
        }
        public void WriteHalfWord(short val, int addr)
        {
            if (IsVldAddr(addr, 0))
            {
                int op = 0x000000FF;
                int byt2 = val & op;
                int byt1 = (val >> 8) & op;
                WriteByte(Convert.ToByte(byt2), addr);
                WriteByte(Convert.ToByte(byt1), addr + 1);
            }
        }
        public void WriteByte(byte val, int addr) { ram[addr] = val; }


        //Flags dealings
        public bool TestFlag(int addr, int bit)
        {
            int num = ReadWord(addr);
            return ((num >> (31 - bit)) & 0x00000001) == 1;
        }

        public void SetFlag(int addr, int bit, bool flag)
        {
            int num = ReadWord(addr);
            int flagged = flag ? num | (0x0000001 << (31 - bit)) : num & (~(0x00
00001 << (31 - bit)));

            WriteWord(flagged, addr);

        }

        //Extracts bits from number
        public static uint ExtractBits(uint word, uint startBit, uint endBit)
        {
            word = word << (int)startBit;
            word = word >> (int)((31 - endBit) + startBit);

            return word;
        }

        //Computes Checksum of memory cells
        public int CheckSum(byte[] mem)
        {
            int cksum = 0;

            for (int i = 0; i < mem.Length; ++i)
            {
                cksum += mem[i] ^ i;
            }
            return cksum;
        }
    }

}
```