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

namespace armsim
{
    public class Branch : Instruction
    {
        public const int TYPE = 0b100;
        //public new uint Inst { get; set; }

       /* public Branch()
        {
            //
        }*/

        //like will be a store of
        public override void DecodeInst() { ; }
        public override void Execute() { }
        public override string ToString()
        {
            return ASMRepr;
        }
    }
}
```



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

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

    //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 const int TYPE = 0b0;
        //public new uint Inst { get; set; }

        uint cond, typ, opcode, Rn, Rd, Rm, Rs;
        bool regimm, sbit;
        Operand2 Oper2;

        //See general definition in parent(Instruction)
        public override void DecodeInst() {
            cond    =   Memory.ExtractBits(Inst, 0, 3);
            typ     =   Memory.ExtractBits(Inst, 4, 6);
            regimm  =   Convert.ToBoolean(Memory.ExtractBits(typ, 31, 31)); //tr
ue == 1; false = 0;
            opcode  =   Memory.ExtractBits(Inst, 7, 10);
            sbit    =   Convert.ToBoolean(Memory.ExtractBits(Inst, 11, 11)); //s
ame as regimm;
            Rn      =   Memory.ExtractBits(Inst, 12, 15);
            Rd      =   Memory.ExtractBits(Inst, 16, 19);
            Oper2   =   Operand2.GetOper2(regimm, Memory.ExtractBits(Inst, 20, 3
1));
        }

        public void DecodeInst_MUL()
        {
            /*
             * Offsets: Rd = 8-11, Rs = 16-19, Rm = 28-31
             Rd = Extractbits
             Rm = Extractbits
             Rs = Extractbits
             */
        }

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

        //calls subclasses execute method that takes an operand2, registers
        public override void Execute() {
            DecodeInst();
            /*if MUL then{
                   DecodeInst_MUL();
                   new MUL().Execute(Rd, Rm, Rs);
            } else do { DecodeInst(); everything below}*/


            List<DataProccess> inst = new List<DataProccess>()
            {
                new AND(), new EOR(), new SUB(), new RSB(), new ADD(),
```

```
                null, null, null, null, null, null, null,
                new ORR(), new MOV(){ Reg = Reg, StrReg = StrReg}, new BIC(), ne
w MVN()
            };

            inst[(int)opcode].Execute(Rn, Rd, Oper2);
            ASMRepr = inst[(int)opcode].ToString();

            /*switch (opcode)
            {
                case MOV.TYPE:
                    MOV.Execute(Rn, Rd, Oper2);
                    break;
                case MVN.TYPE:
                    MVN.Execute(Rn, Rd, Oper2);
                    break;
                case ADD.TYPE:
                    ADD.Execute(Rn, Rd, Oper2);
                    break;
                case SUB.TYPE:
                    SUB.Execute()

            }*/
        }

        public override string ToString()
        {
            return ASMRepr;
        }
    }

    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(base.Reg, (int)Rd, oper2.GetValue());

            ASMRepr = "mov " + StrReg[(int)Rd] + "," + oper2.ToString();
        }

        /*public override string ToString() //already implemented in parent, so
remove duplicate
        {
            return ASMRepr;
        }*/
    }

    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(base.Reg, (int)Rd, ~(oper2.GetValue()));

            ASMRepr = "mvn " + StrReg[(int)Rd] + ", " + oper2.ToString();
        }
         */
    }

    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 = oper2.GetValue() + getreg(rn);
            CPU.SetReg(base.Reg, (int)Rd, val);

            ASMRepr = "add " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToStrin
g();
        }
         */
    }

    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 = getreg(rn) - oper2.GetValue();
            CPU.SetReg(base.Reg, (int)Rd, val);

            ASMRepr = "sub " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToString
();
        }
         */
    }

    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() - getreg(rn);
            CPU.SetReg(base.Reg, (int)Rd, val);

            ASMRepr = "rsb " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToString
();
        }
         */

    }

    class MUL: DataProccess
    {
        //Special
        /*
         public Execute(Rd, Rm, Rs){
            first = getreg(rm)
            second = getreg(rd)

            setreg(regs, Rd, (first * second))
        }
         */

    }

    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 =  oper2.GetValue() & getreg(rn);
          CPU.SetReg(base.Reg, (int)Rd, val);

          ASMRepr = "and " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToString(
);
      }
```

```
        */
    }

    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 =  oper2.GetValue() | getreg(rn);
          CPU.SetReg(base.Reg, (int)Rd, val);

          ASMRepr = "orr " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToString(
);
      }
       */
    }

    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 =  oper2.GetValue() ^ getreg(rn);
          CPU.SetReg(base.Reg, (int)Rd, val);

          ASMRepr = "eor " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToString(
);
      }
       */
    }

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

        /*
         public override void Execute(uint Rn, uint Rd, Operand2 oper2)
      {
          int val =  (~oper2.GetValue()) & getreg(rn);
          CPU.SetReg(base.Reg, (int)Rd, val);

          ASMRepr = "bic " + StrReg[(int)Rd] + ", " StrReg[rn] + oper2.ToString(
);
      }
       */
    }

    //-------------------Passed my IS A test----------------
    public class Operand2
    {
        public uint OperBits { get; set; }

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



```
                return new Oper2_RegSImm() { OperBits = bits };
            }
        }

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

    public class Oper2_RegSReg : Operand2
    {
        /*public override int GetValue()
        {
            reg = extractbits
            shift = extractbits
            reg2 = getreg(extractbits)

            return (int)BarrelShifter.Compute(shift, reg, reg2)
        }*/
    }

    public class Oper2_RegSImm : Operand2
    {
        /*public override int GetValue()
        {
            reg = extractbits
            shift = extractbits
            imm = extractbits

            return (int)BarrelShifter.Compute(shift, reg, imm)
        }*/
    }

    public class Oper2_RORImm : Operand2
    {
        uint rot, num, high, low;
        int final;

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

            high = Memory.ExtractBits(OperBits, (31 - rot), 31) << (int)rot;
            low = OperBits >> (int)rot;

            return final = (int)(high | low);
        }

        public override string ToString()
        {
            return "#"+final.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:
                    return (toShift << (int)displcmnt);
                case 1:
                    return (toShift >> (int)displcmnt);
                case 2:
```

```
                    return (uint)((int)(toShift) >> (int)displcmnt);
                default:
                    //same code as Oper2_RORImm
                    return 0; //there is no need to worry about right rotate rig
ht now.
            }
        }
    }
}
```



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

namespace armsim
{
    public class CPU
    {
        public Memory CPU_Registers;
        public Memory CPU_RAM;

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

        //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
        public Instruction decode(uint instr) {
            return Instruction.CreateInstr(instr, ref CPU_RAM, ref CPU_Registers
);
        }

        // Pause 1 quarter second
        public void execute(Instruction instr) {
            instr.Execute();
        }
        //for getting register values;
        public static int GetRegr(Memory reg, int num){
            return reg.ReadWord((num * 4));
        }

        public static void SetReg(Memory reg, int nreg, int val){
            reg.WriteWord(val, (nreg * 4));
        }

        //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); } }

    }
}
```

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;

namespace armsim
{

    public abstract class Instruction
    {
        private static List<string> regs = new List<string>()
        {
            "r0", "r1", "r2", "r3", "r4", "r5", "r7", "r8",
            "r9", "r10", "r11", "r12", "r13", "r14", "r15", "CSPR"
        };

        public List<string> StrReg { get; set; }

        public string ASMRepr { get; set; }
        public uint Inst { get; set; }

        public Memory Reg { get; set; }
        public Memory RAM { get; set; }

        //byte[] condflags, type, rn, rd;

        //Logic for which type of instruction to create and returns the Instruct
ion
        public static Instruction CreateInstr(uint instr, ref Memory reg, ref Me
mory ram)
        {

            uint typebits = Memory.ExtractBits(instr, 4, 6);
            //Trace.WriteLine("Typebits: " + Convert.ToString(typebits, 2));

            if(typebits == 0b1111)
            {
                return new SWI();
            } else if(typebits > 0b11)
            {
                return new Branch() { Inst = instr, Reg = reg, RAM = ram};
            }else if(typebits > 0b1)
            {
                return new LoadStore() { Inst = instr, Reg = reg, RAM = ram };
            }else
            {
                return new DataProccess() { Inst = instr, Reg = reg, RAM = ram,
StrReg = Instruction.regs };
            }

/*            switch (typebits)   will delete once confirmed unnecessary
            {
                case DataProccess.TYPE:
                    return new DataProccess(instr);
                case LoadStore.TYPE:
                    return new LoadStore(instr);
                case Branch.TYPE:
                    return new Branch(instr);
                default:
                    break;
            }
            return null;*/
        }

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

    public class SWI: Instruction
    {
        public override string ToString() {return ASMRepr; } //might be doing re
dundant work in children classes.
        public override void DecodeInst() {; }
        public override void Execute() { //Execute(ref cpu);
        }

        //Not quite sure that SWI is Instruction as much as it is just a number(
?)
        public void Execute(ref CPU cp) {
            //cp.Running = false
        }
    }
}
```

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

namespace armsim
{
    public class LoadStore : Instruction
    {
        public const int TYPE = 0b010;
        //public new uint Inst { get; set; }
        byte[] pubwl;
        uint rn, rd, oper2;

       /* public LoadStore(uint instr)
        {

        }*/
       // Extracts the bits Executes needs to run and stores them in variables.
        public override void DecodeInst() {
            /*
             pubwl = extractBits
             rn = extractbits
             rd = extractbits
             oper2 = extractbits
             */

            ;
        }

        // public void Execute(Rn, Rd, Oper2)
        // poublic void Execute(Rn, Rd, Oper2)
        public override void Execute() {
            DecodeInst();
            /*List<LoadStore> loadStores = new List<LoadStore>() {  new Store, n
ew Load()}

            LoadStores[pubwl[4]].Execute(Rn, Rd, Oper2); //indexes into array to
 run the correct command
             */
        }
        public override string ToString()
        {
            return ASMRepr;
        }
    }

    public class LDR: LoadStore
    {
        /*
         *
         * public void Execute(Rn, Rd, Oper2){
         *      setreg(Rd, RAM.readword(getreg(Rn) + oper2.getvalue()))
         * }
         *
         */
    }

    public class STR: LoadStore
    {
        /*
         * public void Execute(Rn, Rd, Oper2){
         *      RAM.WriteWord(getreg(Rd), getreg(Rn) + Oper2)
         * }
         */
    }

    public class STM : LoadStore
    {
```



```
        /*
         * public void Execute(Rn, reglist){
         *      for i in reglist{
         *          RAM.WriteWord(getreg(i), getreg(Rn))
         *      }
         *
         * }
         *
         */
    }

    public class LDM : LoadStore
    {
        /* public void Execute(Rn, reglist){
         *  for i in reglist{
         *       setreg(i, RAM.readword(getreg(rn)))
         *       }
         *  }
         */


    }
}
```