Bitinstruction

**case** OpCode.***AND***:

//AND

instructionParts = parseLogic(opCode); // use whatever Kegan calls his opCode parser

**break**;

**case** OpCode.***ORR***:

//AND

instructionParts = parseLogic(opCode); // use whatever Kegan calls his opCode parser

**break**;

**case** OpCode.***NOT***:

//AND

instructionParts = parseLogic(opCode); // use whatever Kegan calls his opCode parser

**break**;

**case** OpCode.***SRC***:

//Shift

instructionParts = parseShiftRotate(opCode);

**break**;

**case** OpCode.***RRC***:

//Rotate

instructionParts = parseShiftRotate(opCode);

**break**;

minicomputer

**case** OpCode.***SRC***:

isTransferInstruction = **true**;

register = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

arithmeticOrLogic = instructionParse.get(BitInstruction.***KEY\_ARITHMETIC\_OR\_LOGIC***).getValue();

leftOrRight = instructionParse.get(BitInstruction.***KEY\_LEFT\_OR\_RIGHT***).getValue();

shiftCount = instructionParse.get(BitInstruction.***KEY\_SHIFT\_COUNT***).getValue();

src(register, arithmeticOrLogic, leftOrRight, shiftCount);

**break**;

**case** OpCode.***RRC***:

isTransferInstruction = **true**;

register = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

arithmeticOrLogic = instructionParse.get(BitInstruction.***KEY\_ARITHMETIC\_OR\_LOGIC***).getValue();

leftOrRight = instructionParse.get(BitInstruction.***KEY\_LEFT\_OR\_RIGHT***).getValue();

shiftCount = instructionParse.get(BitInstruction.***KEY\_SHIFT\_COUNT***).getValue();

rrc(register, arithmeticOrLogic, leftOrRight, shiftCount);

**break**;

**case** OpCode.***AND***:

isTransferInstruction = **true**;

//Use Kegan's Parsing nomenclature

register = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

register2 = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

and(register, register2);

**break**;

**case** OpCode.***ORR***:

isTransferInstruction = **true**;

//Use Kegan's Parsing nomenclature

register = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

register2 = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

orr(register, register2);

**break**;

**case** OpCode.***NOT***:

isTransferInstruction = **true**;

//Use Kegan's Parsing nomenclature

register = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

register2 = Integer.*parseInt*(instructionParse.get(BitInstruction.***KEY\_REGISTER***).getValue(), 2);

not(register, register2);

**break**;

mini computer functions

**public** **void** src(**int** register, String arithmeticOrLogic, String leftOrRight, String shiftCount)

{

// Retrieve the specified register

Register registerSelect1 = getR(register);

// Move the register contents into the Internal Result Register (IRR)?

IRR[0].setBitValue(registerSelect1.getBitValue());

// If IRR contents is >= 0, move the EA to the Internal Address Register (IAR)

// Should I be calling the TRR instruction or setting the EQUALORNOT CC register bit when testing if zero??

**int** irr = Integer.*parseInt*(IRR[0].getBitValue().getValue());

//if(irr >= 0) {

//IAR.setBitValue(ea);

//} else {

// Else set IAR value to Shift Register

// src(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

IAR.setBitValue(ArithmeticLogicUnit.*src*(String.*valueOf*(irr), arithmeticOrLogic, leftOrRight, shiftCount));

//}

// **TODO**: Check that address specified by IAR is valid (not reserved, not larger than max)

// Store IAR contents into the PC

// PC can only hold 12 bits so chop off the leading zeros

//String pc = IAR.getBitValue().getValue().substring(4, 16);

//PC.setBitValue(pc);

}

/\*\*

\* Rotate Register by Count

\* **@param** register

\* **@param** arithmeticOrLogic

\* **@param** leftOrRight

\* **@param** shiftCount

\*/

**public** **void** rrc(**int** register, String arithmeticOrLogic, String leftOrRight, String shiftCount)

{

// Retrieve the specified register

Register registerSelect1 = getR(register);

// Move the register contents into the Internal Result Register (IRR)?

IRR[0].setBitValue(registerSelect1.getBitValue());

// If IRR contents is >= 0, move the EA to the Internal Address Register (IAR)

// Should I be calling the TRR instruction or setting the EQUALORNOT CC register bit when testing if zero??

**int** irr = Integer.*parseInt*(IRR[0].getBitValue().getValue());

//if(irr >= 0) {

//IAR.setBitValue(ea);

//} else {

// Else set IAR value to Shift Register

// src(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

IAR.setBitValue(ArithmeticLogicUnit.*rrc*(String.*valueOf*(irr), arithmeticOrLogic, leftOrRight, shiftCount));

//}

// **TODO**: Check that address specified by IAR is valid (not reserved, not larger than max)

// Store IAR contents into the PC

// PC can only hold 12 bits so chop off the leading zeros

//String pc = IAR.getBitValue().getValue().substring(4, 16);

//PC.setBitValue(pc);

}

**public** **void** and(**int** register, **int** register2)

{

// Retrieve the specified register

Register registerSelect1 = getR(register);

Register registerSelect2 = getR(register2);

// Move the register contents into the Internal Result Register (IRR)?

IRR[0].setBitValue(registerSelect1.getBitValue());

IRR[1].setBitValue(registerSelect1.getBitValue());

// If IRR contents is >= 0, move the EA to the Internal Address Register (IAR)

// Should I be calling the TRR instruction or setting the EQUALORNOT CC register bit when testing if zero??

**int** irr = Integer.*parseInt*(IRR[0].getBitValue().getValue());

**int** irr2 = Integer.*parseInt*(IRR[1].getBitValue().getValue());

//if(irr >= 0) {

//IAR.setBitValue(ea);

//} else {

// Else set IAR value to Shift Register

// src(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

IAR.setBitValue(ArithmeticLogicUnit.*and*(String.*valueOf*(irr), String.*valueOf*(irr2)));

//}

// **TODO**: Check that address specified by IAR is valid (not reserved, not larger than max)

// Store IAR contents into the PC

// PC can only hold 12 bits so chop off the leading zeros

//String pc = IAR.getBitValue().getValue().substring(4, 16);

//PC.setBitValue(pc);

}

**public** **void** orr(**int** register, **int** register2)

{

// Retrieve the specified register

Register registerSelect1 = getR(register);

Register registerSelect2 = getR(register2);

// Move the register contents into the Internal Result Register (IRR)?

IRR[0].setBitValue(registerSelect1.getBitValue());

IRR[1].setBitValue(registerSelect1.getBitValue());

// If IRR contents is >= 0, move the EA to the Internal Address Register (IAR)

// Should I be calling the TRR instruction or setting the EQUALORNOT CC register bit when testing if zero??

**int** irr = Integer.*parseInt*(IRR[0].getBitValue().getValue());

**int** irr2 = Integer.*parseInt*(IRR[1].getBitValue().getValue());

//if(irr >= 0) {

//IAR.setBitValue(ea);

//} else {

// Else set IAR value to Shift Register

// src(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

IAR.setBitValue(ArithmeticLogicUnit.*and*(String.*valueOf*(irr), String.*valueOf*(irr2)));

//}

// **TODO**: Check that address specified by IAR is valid (not reserved, not larger than max)

// Store IAR contents into the PC

// PC can only hold 12 bits so chop off the leading zeros

//String pc = IAR.getBitValue().getValue().substring(4, 16);

//PC.setBitValue(pc);

}

**public** **void** not(**int** register)

{

// Retrieve the specified register

Register registerSelect1 = getR(register);

// Move the register contents into the Internal Result Register (IRR)?

IRR[0].setBitValue(registerSelect1.getBitValue());

// If IRR contents is >= 0, move the EA to the Internal Address Register (IAR)

// Should I be calling the TRR instruction or setting the EQUALORNOT CC register bit when testing if zero??

**int** irr = Integer.*parseInt*(IRR[0].getBitValue().getValue());

//if(irr >= 0) {

//IAR.setBitValue(ea);

//} else {

// Else set IAR value to Shift Register

// src(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

IAR.setBitValue(ArithmeticLogicUnit.*and*(String.*valueOf*(irr)));

//}

// **TODO**: Check that address specified by IAR is valid (not reserved, not larger than max)

// Store IAR contents into the PC

// PC can only hold 12 bits so chop off the leading zeros

//String pc = IAR.getBitValue().getValue().substring(4, 16);

//PC.setBitValue(pc);

}

opcode.java

/\*\*

\* AND

\*/

**public** **static** **final** String ***AND*** = "010011"; //23

/\*\*

\* OR

\*/

**public** **static** **final** String ***ORR*** = "010100"; //24

/\*\*

\* NOT

\*/

**public** **static** **final** String ***NOT*** = "010101"; //25

/\*\*

\* Shift Register by Count

\*/

**public** **static** **final** String ***SRC*** = "011111"; //31

/\*\*

\* Rotate Register by Count

\*/

**public** **static** **final** String ***RRC*** = "100000"; //32

ALU

**public** **static** **void** and(String p, String q)

{

//register will be of length 16 bits

**int** n = 16;

**for**(**int** i = 0;i<n;i++)

{

String pbit = p.substring(i, i+1);

String qbit = q.substring(i, i+1);

String r = pbit + qbit;

**if** (i==0)

{

//edge case: first bit

**if** (r == "00" || r == "01" || r == "10")

{

p= "0" + p.substring(i+1, i+2);

}

**else** **if** (r == "11")

{

p= "1" + p.substring(i+1, i+2);

}

}

**else** **if** (i==n)

{// edge case: last bit

**if** (r == "00" || r == "01" || r == "10")

{

p= p.substring(i-1, i)+"0";

}

**else** **if** (r == "11")

{

p= p.substring(i-1, i)+"1";

}

}

**else**

{//general case

**if** (r == "00" || r == "01" || r == "10")

{

p= p.substring(i-1, i)+"0"+ p.substring(i+1, i+2);

}

**else** **if** (r == "11")

{

p= p.substring(i-1, i)+"1"+ p.substring(i+1, i+2);

}

}

}

}

/\*\*

\* Logical OR of RegisterP and RegisterQ

\* **@param** p : which is a bitString

\* **@param** q : which is a bitString

\* **@return** c(p) <- c(p) OR c(q)

\*/

**public** **static** **void** or(String p, String q)

{

//register will be of length 16 bits

**int** n = 16;

**for**(**int** i = 0;i<n;i++)

{

String pbit = p.substring(i, i+1);

String qbit = q.substring(i, i+1);

String r = pbit + qbit;

**if** (i==0)

{

//edge case: first bit

**if** (r == "11" || r == "01" || r == "10")

{

p= "1" + p.substring(i+1, n);

}

**else** **if** (r == "00")

{

p= "0" + p.substring(i+1, n);

}

}

**else** **if** (i==n)

{// edge case: last bit

**if** (r == "11" || r == "01" || r == "10")

{

p= p.substring(0, i)+"1";

}

**else** **if** (r == "00")

{

p= p.substring(0, i)+"0";

}

}

**else**

{//general case

**if** (r == "11" || r == "01" || r == "10")

{

p= p.substring(0, i)+"1"+ p.substring(i+1, n);

}

**else** **if** (r == "00")

{

p= p.substring(0, i)+"0"+ p.substring(i+1, n);

}

}

}

}

/\*\*

\* Logical NOT of RegisterP ; i.e. switch "1's & 0's"

\* **@param** p : which is a bitString

\* **@return** c(p) <- NOT c(p)

\*/

**public** **static** **void** not(String p)

{

//register will be of length 16 bits

**int** n = 16;

**for**(**int** i = 0;i<n;i++)

{

String r = p.substring(i, i+1);

**if** (i==0)

{

//edge case: first bit

**if** (r == "1")

{

p= "0" + p.substring(i+1, n); //switch the first bit, save the rest

}

**else** **if** (r == "0")

{

p= "1" + p.substring(i+1, n);

}

}

**else** **if** (i==n)

{// edge case: last bit

**if** (r == "0")

{

p= p.substring(0, i)+"1"; //save everything, but switch the last bit

}

**else** **if** (r == "1")

{

p= p.substring(0, i)+"0";

}

}

**else**

{//general case

**if** (r == "0" )

{

p= p.substring(0, i)+"1"+ p.substring(i+1, n); //change the ith bit

}

**else** **if** (r == "1")

{

p= p.substring(0, i)+"0"+ p.substring(i+1, n);

}

}

}

}

/\*\*

\* SHIFT Register Command

\* **@param** bitword : which is a bitString

\* **@return** The register will shift left/right, logic/arithmetic, 1-15 units

\*/

**public** **static** String src(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

{

//Parse bitword

// this parse may have to occur in the main program and the Register passed in??

///bitInstruction Class - ParseInstruction --- call from minicomputer in SingleStep

//String Opcode = bitword.substring(0, 6);

//String r = bitword.substring(6, 8); //register

//String ArithmeticOrLogic = bitword.substring(8, 9); //this is a flag to adjust for a sign bit; 0 = arithmetic and 1 = logic;

//String LeftOrRight = bitword.substring(9, 10); // left = 1; right = 0;

//String sCount = bitword.substring(12, 16);

//convert sCount from string to number, this will be the loop counter

**int** n = Integer.*parseInt*(sCount);

String buffer;

String keeper;

String shifted;

// please note that the mechanics of the simple machine would in fact shift one bit at a time.

// and then loop through the ALU again to perform additional shifts to keep the real estate on

// the chip small. i.e. a 'two shift' isn't build into the hardware.

/////we can use this same code for rotation by setting the Buffer = substring(0,1) or substring(15,16)////

**for**(**int** i = 0;i<n;i++)

{

//shift values left

**if**(LeftOrRight == "1")

{

keeper = Registervalue.substring(1, 16);

buffer = "0";

shifted = keeper + buffer; ///shifted to the left

**if**(ArithmeticOrLogic == "0") //i.e. arithmetic shift

{

String overflow = Registervalue.substring(1, 2);

**if** (overflow == "1" ) {String SetOverflow = "1";} /////////////do we have an ALU overflow flag set yet?????????????????????????

}

}

**else**

{

//shift values right

keeper = Registervalue.substring(0, 15);

**if**(ArithmeticOrLogic == "0") //i.e. arithmetic shift

{//if you are arithmetic shifting to the right, then you insert the sign bit

buffer = Registervalue.substring(0, 1);

}

**else**

{

buffer = "0"; //if you are logic shifting you insert a zero

}

shifted = buffer + keeper; ///shifted to the right

}

Registervalue = shifted; // this is to get ready to loop through one more time

}

// this is to exit with final answer

**return** Registervalue;

}

/\*\*

\* ROTATE Register Command

\* **@param** bitword : which is a bitString

\* **@return** The register will rotate left/right, 1-15 units

\*/

**public** **static** String rrc(String Registervalue, String ArithmeticOrLogic, String LeftOrRight, String sCount)

{

//Parse bitword

// this parse may have to occur in the main program and the Register passed in??

//String Opcode = bitword.substring(0, 6);

//Register r = bitword.substring(6, 8); //register

//String ArithmeticOrLogic = bitword.substring(8, 9); //this is a flag to adjust for a sign bit; 0 = arithmetic and 1 = logic;

//String LeftOrRight = bitword.substring(9, 10); // left = 1; right = 0;

//String sCount = bitword.substring(12, 16);

//convert sCount from string to number, this will be the loop counter

**int** n = Integer.*parseInt*(sCount);

String buffer;

String keeper;

String shifted;

// please note that the mechanics of the simple machine would in fact shift one bit at a time.

// and then loop through the ALU again to perform additional shifts to keep the real estate on

// the chip small. i.e. a 'two shift' isn't build into the hardware.

**for**(**int** i = 0;i<n;i++)

{

//shift values left

**if**(LeftOrRight == "1")

{

keeper = Registervalue.substring(1, 16);

buffer = Registervalue.substring(0, 1);

shifted = keeper + buffer; ///shifted to the left

}

**else**

{

//shift values right

keeper = Registervalue.substring(0, 15);

buffer = Registervalue.substring(15, 16);

shifted = buffer + keeper; ///shifted to the right

}

Registervalue = shifted; // this is to get ready to loop through one more time

}

**return** Registervalue; // this is to exit with final answer

}