Homework #3, CS6904

Yasaman Bahrami Samani

This tracing program is written with Javascript and Bootstrap. It is classified into stages and interstage buffers. Each stage and buffer will be explained in the following report. To run this program, you simply need to open the "index.html" file using a browser. The inputs are opcode (add, sub, lw, sw or beq), operands (operand1, operand2, operand3) , the initial values of Registers, Data memory and pc. The test inputs for Registers and Data memory are in the testInputs folder. This program is accessible at:

http://www.cs.mun.ca/~ybs572/MIPS/

Code Structure:

In this part, we will talk about different parts of the code.

Index:

This html file contains view of the program which includes input and output section. Also, Javascript functions of different stages and buffers are called here.

```
function trace() {
    var pc = document.getElementById("pc").value;
    var readAddressIM = pc;
    var opcode = document.getElementById("opcode").value;
    var operand1 = document.getElementById("operand1").value;
    var operand2 = document.getElementById("operand2").value;
    var operand3 = document.getElementById("operand3").value;
    var controllers = setControllers(opcode);
    /* ------ IF Stage ------*/
    pc = pcAdder(pc);
    document.getElementById("semantics").innerHTML = showSemantics(opcode, operand1, operand2, operand3);
    /* ------ IF/ID Buffer ------*/
    var IFIDBuffer = setIFIDBuffer(opcode, operand1, operand2, operand3,pc);
    setTableHeader(document.getElementById("IFIDBufferHead"),opcode);
    insertTableRow(document.getElementById("IFIDBufferBody"), IFIDBuffer);
    /* ------ ID Stage ------*/
    var register = registers(IFIDBuffer, (document.getElementById("registers").value));
    /* ------ ID/EX Buffer ------*/
```

```
var IDEXBuffer = setIDEXBuffer(opcode, register, IFIDBuffer[2], IFIDBuffer[3], pc);

insertTableRow(document.getElementById("IDEXBufferBody"), IDEXBuffer);

/* ------ EX Stage ------*/

var MUX12Result = MUX12(IDEXBuffer[0], IDEXBuffer[1], controllers[0]);

var ALUControlValueResult = ALUControlValue(IDEXBuffer[2], controllers[1]);

var MUX11Result = MUX11(IDEXBuffer[2], IDEXBuffer[3], controllers[2]);

var ALUResult = ALU(IDEXBuffer[4], MUX11Result, ALUControlValueResult);

var shiftLeft2Result = shiftLeft2(IDEXBuffer[2]);

var addResult = add(shiftLeft2Result, IDEXBuffer[5]);

var EXControllers = [controllers[0], controllers[1], controllers[2]];

insertTableRow(document.getElementById("EXControllers"), EXControllers);

/* ------ EX/MEM Buffer ------*/

var EXMEMBuffer = setEXMEMBuffer(MUX12Result, IDEXBuffer[3], ALUResult[1], ALUResult[0], addResult, controllers[3]);

insertTableRow(document.getElementById("EXMEMBufferBody"), EXMEMBuffer);

/* ------ MEM Stage ------*/

var readData = DataMemory(EXMEMBuffer[1], EXMEMBuffer[2], controllers[4], controllers[5], document.getElementById("dataMemory").value);

var branchResult = branch(EXMEMBuffer[3], controllers[3]);

var MEMControllers = [controllers[3], controllers[4], controllers[5]];

insertTableRow(document.getElementById("MEMControllers"), MEMControllers);

/* ------ MEM/WB Buffer ------*/

var MEMWBBuffer = setMEMWBBuffer(EXMEMBuffer[0] ,EXMEMBuffer[2], readData);

insertTableRow(document.getElementById("MEMWBBufferBody"), MEMWBBuffer);

/* ------ WB Stage ------*/

var MUX14Result = MUX14(MEMWBBuffer[2], MEMWBBuffer[1], controllers[7]);

var WBControllers = [controllers[6], controllers[7]];

insertTableRow(document.getElementById("WBControllers"), WBControllers);

var registerResult = registerWrite(EXMEMBuffer[0], MUX14Result, controllers[6], (document.getElementById("registers").value));

if(registerOutput != ""){
```

```
            document.getElementById("registers").value="";
            document.getElementById("registers").value = registerOutput;
        }
        if(memoryOutput != ""){
            document.getElementById("dataMemory").value="";
            document.getElementById("dataMemory").value = memoryOutput;
        }
        var IFStage = [readAddressIM, EXMEMBuffer[4], pc];
        insertTableRow(document.getElementById("IFStageBody"), IFStage);
        var IDStage = [IDEXBuffer[0], IDEXBuffer[1], IDEXBuffer[2], MUX14Result, EXMEMBuffer[0], IFIDBuffer[2], IFIDBuffer[1], register[1], register[0]];
        insertTableRow(document.getElementById("IDStageBody"), IDStage);
        var EXStage = [IDEXBuffer[0], IDEXBuffer[1], IDEXBuffer[2], IDEXBuffer[2], IDEXBuffer[3], MUX11Result, IDEXBuffer[4], EXMEMBuffer[2], EXMEMBuffer[3], shiftLeft2Result, IDEXBuffer[5], EXMEMBuffer[4]];
        insertTableRow(document.getElementById("EXStageBody"), EXStage);
        var MEMStage = [EXMEMBuffer[1], EXMEMBuffer[2], EXMEMBuffer[2], MEMWBBuffer[2]];
        insertTableRow(document.getElementById("MEMStageBody"), MEMStage);
        var WBStage = [MEMWBBuffer[1], MEMWBBuffer[2], MUX14Result];
        insertTableRow(document.getElementById("WBStageBody"), WBStage);
        pc= MUX3(pc, EXMEMBuffer[4], branchResult);
        document.getElementById("pc").value = pc;
}
function showSemantics(opcode, operand1, operand2, operand3){
        var semantics;
        if(opcode == "add")
            semantics = "$"+operand1+" <-- $"+operand2+"+ $"+operand3;
        else if(opcode == "sub")
            semantics = "$"+operand1+" <-- $"+operand2+"- $"+operand3;
        else if(opcode == "lw")
            semantics = "$"+operand1+" <-- Memory[ $"+operand3+"+ "+operand2+"]";
```

```
        else if(opcode == "sw")
                semantics = "Memory[ $"+operand3+"+ "+operand2+"] <-- $"+operand1;
        else if(opcode == "beq")
                if(operand1 == operand2)
                        semantics = "pc <-- pc+4+4*"+operand3;
                else
                        semantics = operand1 + " is not equal to "+operand2;
        return(semantics);
}
function insertTableRow(element, cells){
        element.innerHTML = "";
        var row = element.insertRow(0);
        for (var i=0; i<cells.length; i++)
                row.insertCell(i).innerHTML = cells[i];
}
```

Controllers:

In this part of code, in order to the entry opcode, proper control vectors will be set:

```
function setControllers(opcode){
        var regWrite = setRegWrite(opcode);
        var regDst = setRegDst(opcode);
        var ALUOpt = setALUOpt(opcode);
        var ALUSrc = setALUSrc(opcode);
        var branchCS = setBranchCS(opcode);
        var memRead = setMemRead(opcode);
        var memWrite = setMemWrite(opcode);
        var memtoReg = setMemtoReg(opcode);
        var controllers = [regDst, ALUOpt, ALUSrc, branchCS, memRead, memWrite, regWrite,
memtoReg];
        return controllers;
}
```

```
function setRegWrite(opcode){
	var regWrite;
	if(opcode=="sub" || opcode=="add" || opcode=="lw")
		regWrite = 1;
	else
		regWrite = 0;
	return regWrite;
}
function setRegDst(opcode){
	var regDst;
	if(opcode == "sub" || opcode == "add")
		regDst = 1;
	else if(opcode == "lw")
		regDst = 0;
	else
		regDst = "-";
	return regDst;
}
function setALUOpt(opcode){
	var ALUOpt;
	if(opcode == "sub" || opcode == "add")
		ALUOpt = "10";
	else if(opcode == "lw" || opcode == "sw")
		ALUOpt = "00";
	else
		ALUOpt = "01";
	return ALUOpt;
}
function setALUSrc(opcode){
	var ALUSrc;
```

```
        if(opcode == "sub" || opcode == "add" || opcode == "beq")
                ALUSrc = 0;
        else if(opcode == "lw" || opcode == "sw")
                ALUSrc = 1;
        return ALUSrc;
}
function setBranchCS(opcode){
        var branchCS;
        if(opcode == "sub" || opcode == "add" || opcode == "lw" || opcode == "sw")
                branchCS = 0;
        else if(opcode == "beq")
                branchCS = 1;
        return branchCS;
}
function setMemRead(opcode){
        var memRead;
        if(opcode == "sub" || opcode == "add" || opcode == "beq" || opcode == "sw")
                memRead = 0;
        else if(opcode == "lw")
                memRead = 1;
        return memRead;
}
function setMemWrite(opcode){
        var memWrite;
        if(opcode == "sub" || opcode == "add" || opcode == "beq" || opcode == "lw")
                memWrite = 0;
        else if(opcode == "sw")
                memWrite = 1;
        return memWrite;
}
```

```
function setMemtoReg(opcode){
        var memtoReg;
        if(opcode == "sub" || opcode == "add")
                memtoReg = 0;
        else if(opcode == "lw")
                memtoReg = 1;
        else
                memtoReg = "-";
        return memtoReg;
}
```

IF Stage:

This stage has just one function (pcAdder) which adds 4 to the program counter:

```
function pcAdder(pc){
        pc = parseInt(pc)+4;
        return pc;
}.
```

IF/ID Buffer:

This part reveals the IF/ID Buffer which has different values according to the opcode.

```
function setIFIDBuffer(opcode, operand1, operand2, operand3,pc){
        var     IFIDBuffer;
        if(opcode == "add")
                IFIDBuffer = [0, operand2, operand3, operand1, 0, 32, pc];
        else if(opcode == "sub")
                IFIDBuffer = [0, operand2, operand3, operand1, 0, 34, pc];
        else if(opcode == "lw")
                IFIDBuffer = [35, operand3, operand1, operand2, pc];
        else if(opcode == "sw")
                IFIDBuffer = [43, operand3, operand1, operand2, pc];
        else
                IFIDBuffer = [4, operand1, operand2, operand3, pc];
```

```
    return IFIDBuffer;
}.
```

ID Stage:

This stage has two main functionalities. First one is Registers which gets the addresses of relevant sources and sets the values of them as outputs:

```
function registers(IFIDBuffer, registers){
    var readRegister1;
    var readRegister2
    var readData1;
    var readData2;
    var registersResult = "-";
    var registerObject = getRegistersObject(registers);
    readRegister1 = IFIDBuffer[1];
    readData1 = getRegisterValue(readRegister1, registerObject);
    readRegister2 = IFIDBuffer[2];
    readData2 = getRegisterValue(readRegister2, registerObject);
    registersResult = [readData1, readData2];
    return registersResult;
}
```

The second one is RegisterWrite which gets the result of MUX14 in the WB stage and writes it in the relevant write register address:

```
function registerWrite(writeRegister, writeData, regWrite, registers){
    if(regWrite == 1){
        registerOutput = "";
        var registerObject = getRegistersObject(registers);
        for(var i=0; i<registerObject.length; i++){
            if(registerObject[i].number == writeRegister && writeData!= "-")
                registerObject[i].value = writeData;
        }
        return registerObject;
    }
```

```
}
```

ID/EX Buffer:

This buffer has the values of register function outputs (Read data1 and Read data2), pc and also three other data paths (instruction[15..0], instruction[20..16] and instruction [15..11]):

```
function setIDEXBuffer(opcode,registers, IFIDBuffer2, IFIDBuffer3, pc){
    var     IDEXBuffer = [];
    if(opcode == "add")
        IDEXBuffer = [IFIDBuffer3,"-",32, registers[1], registers[0], pc];
    else if(opcode == "sub")
        IDEXBuffer = [IFIDBuffer3,"-",34, registers[1], registers[0], pc];
    else
        IDEXBuffer = ["-",IFIDBuffer2,IFIDBuffer3, registers[1], registers[0], pc];
    return IDEXBuffer;
}
```

EX Stage:

The outputs of the ID/EX Buffer are inputs of different functions of Ex stage. We will talk about them and depict the codes of the most important ones.

MUX12:

In order to the value of the RegDst control signal, the result of this function would be IDEXBuffer[0] (Rd field) or IDEXBuffer[1].

```
function MUX12(IDEXBuffer0, IDEXBuffer1, regDst){
    if (regDst == 0)
        MUX12Result = IDEXBuffer1;
    else if (regDst == 1)
        MUX12Result = IDEXBuffer0;
    else
        MUX12Result = "-";
    return MUX12Result;
}
```

ALU control:

This function holds the values of the ALU Op control vector and IDEXBuffer[2] (instruction[15..0]). It sends proper ALU Control to the ALU to perform the relevant operation (add, sub, lw, sw or beq).

```
function ALUControlValue(IDEXBuffer2, ALUOP){
    var ALUInput1 = IDEXBuffer2;
    var ALUControlResult = [ALUInput1, ALUOP];
    return ALUControlResult;
}
```

MUX11:

The result of this function is determined by ALU Src control vector. If it is zero (r-type and beq), the output will set by Read data2; else if is one (Lw and Sw), the result will be IDEXBuffer[2] (instruction[15..0]).

```
function MUX11(IDEXBuffer2, IDEXBuffer3, ALUSrc){
    var MUX11Result;
    if (ALUSrc == 0)
        MUX11Result = IDEXBuffer3;
    else
        MUX11Result = IDEXBuffer2;
    return MUX11Result;
}
```

Shift left2:

This function gets its input from IDEXBuffer[2] and shift it to the left two times. In decimal, the output will be:

IDEXBuffer[2]*4

```
function shiftLeft2(IDEXBuffer2){
    var shiftLeft2;
    if(IDEXBuffer2 != "-")
        shiftLeft2 = parseInt(IDEXBuffer2)*4;
    else shiftLeft2 = "-"
    return shiftLeft2;
}
```

ALU:

This is one of the most important functionalities of this stage. It gets its input values from the result of MUX11 and Read data1. In order to ALU Src, it calculates the proper outputs. For different opcode:

```
function ALU(readData1, MUX11Result, ALUControlValueResult){
    var zero = 0;
```

```
    var ALUResult1;
    if(ALUControlValueResult[1] == "10"){
        if(ALUControlValueResult[0]==32)
            ALUResult1 = parseInt(readData1)+parseInt(MUX11Result);
        else if(ALUControlValueResult[0]==34)
            ALUResult1 = parseInt(readData1)-parseInt(MUX11Result);
    }else if(ALUControlValueResult[1] == "01"){
        if(readData1 == MUX11Result)
            zero = 1;
        ALUResult1 = "-";
    }else if(ALUControlValueResult[1] == "00"){
        ALUResult1 = parseInt(ALUControlValueResult[0])+ parseInt(readData1);
    }
    var ALUResult = [zero, ALUResult1];
    return ALUResult;
}
```

Add:

Regarding to the inputs of this function, we can say that it is actually calculating the result of beq opcode:

Pc+4+ ShiftLeft2Result = pc+4+ IDEXBuffer[2]*4.

```
function add(shiftLeft2Result, IDEXBuffer5){
    var input1 = IDEXBuffer5;
    var input2 = shiftLeft2Result;
    var addResult;
    if(input2 != "-")
        addResult = parseInt(input1)+parseInt(input2);
    else
        addResult = "-";
    return addResult;
}
```

EX/MEM Buffer:

This buffer contains the result of MUX12, Read data2, ALU result, Zero and Add result. Obviously, the Add result is just relevant to the beq opcode.

```
function setEXMEMBuffer(MUX12Result, readData2, ALUResult1, ALUResult0, addResult, branchCS){
    var EXMEMBuffer;
    if(branchCS == 1)
        EXMEMBuffer = [MUX12Result, readData2, ALUResult1, ALUResult0, addResult];
    else
        EXMEMBuffer = [MUX12Result, readData2, ALUResult1, ALUResult0, "-"];
    return EXMEMBuffer;
}
```

MEM Stage:

The DataMemory function in this stage is controlled by MemRead and MemWrite control vectors. If MemRead is 1, data will load from memory. On the other hand, if MemWrite is 1, data will store to the memory. This function gets the read and write address from ALU result and the write data value from Read data2 from Registers.

```
function DataMemory(EXMEMBuffer1, EXMEMBuffer2, memRead, memWrite, dataMemory){
    memoryOutput = "";
    var memory = getDataMemory(dataMemory);
    var readAddress = EXMEMBuffer2;
    var writeAddress = EXMEMBuffer2;
    var writeData = EXMEMBuffer1;
    var readData = "-";
    if(memRead == 1){
        for(var i=0; i<memory.length; i++){
            if(memory[i].address == readAddress)
                readData = memory[i].value;
        }
    }else if(memWrite == 1){
        for(var i=0; i<memory.length; i++){
            if(memory[i].address == writeAddress)
                memory[i].value = writeData;
```

```
        }
    }
}
```

This stage has another function which determines if the conditions of beq are established or not. Bench function is an ‘And’ function with the inputs Zero and Branch control vector. If the result of this function is true, a control signal will go to the MUX3 which needs to select between two inputs: pc+4 or pc+4+pc*offset:

```
function branch(ALUResult0, branchCS){
    var branchResult = 0;
    if (ALUResult0 == 1 && branchCS == 1)
        branchResult = 1;
    else
        branchResult = 0;
    return branchResult;
}
```

MEM/WB Buffer:

This buffer keeps the value of WBMEMBuffer[0], ALU result and Read data:

```
function setMEMWBBuffer(MUX12Result ,ALUResult1, readData){
    var MEMWBBuffer = [MUX12Result ,ALUResult1, readData];
    return MEMWBBuffer;
}
```

WB Stage:

MUX14 in this stage gets value of Read data and Read address. According to the Memto Reg control signal, the proper output is sent to Registers.

```
function MUX14(MEMWBBuffer2, MEMWBBuffer1, memtoReg){
    var MUX14Result;
    if(memtoReg == 0)
        MUX14Result = MEMWBBuffer1;
    else if (memtoReg == 1)
        MUX14Result = MEMWBBuffer2;
    else
        MUX14Result = "-";
```

return MUX14Result;

}

Testing:

In this part, we will trace MIPS machine program for every opcode that we considered (add, sub, lw, sw and beq).

Add:

Figure 1, shows how we considered inputs. As you can see in this figure, Registers that are related to the input operands are specified.

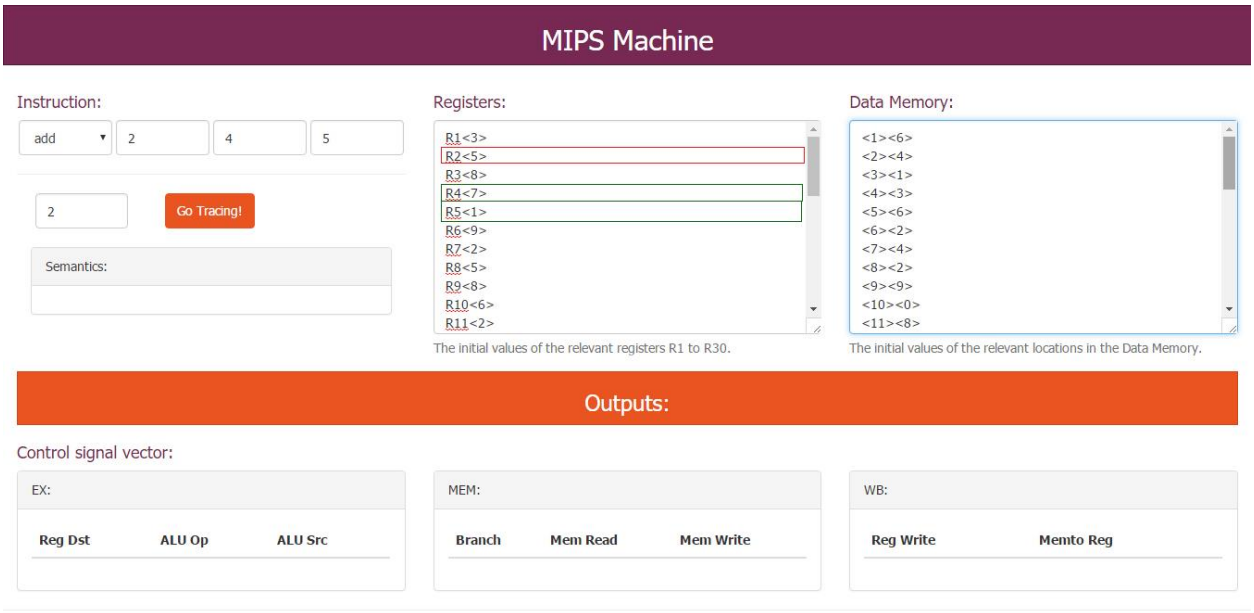

Figure 1- Initializing input for Add opcode

After we run the program by pushing the “Go Tracing!” button, the values of pc, Registers and Control signals will change. Figure 2 shows them:

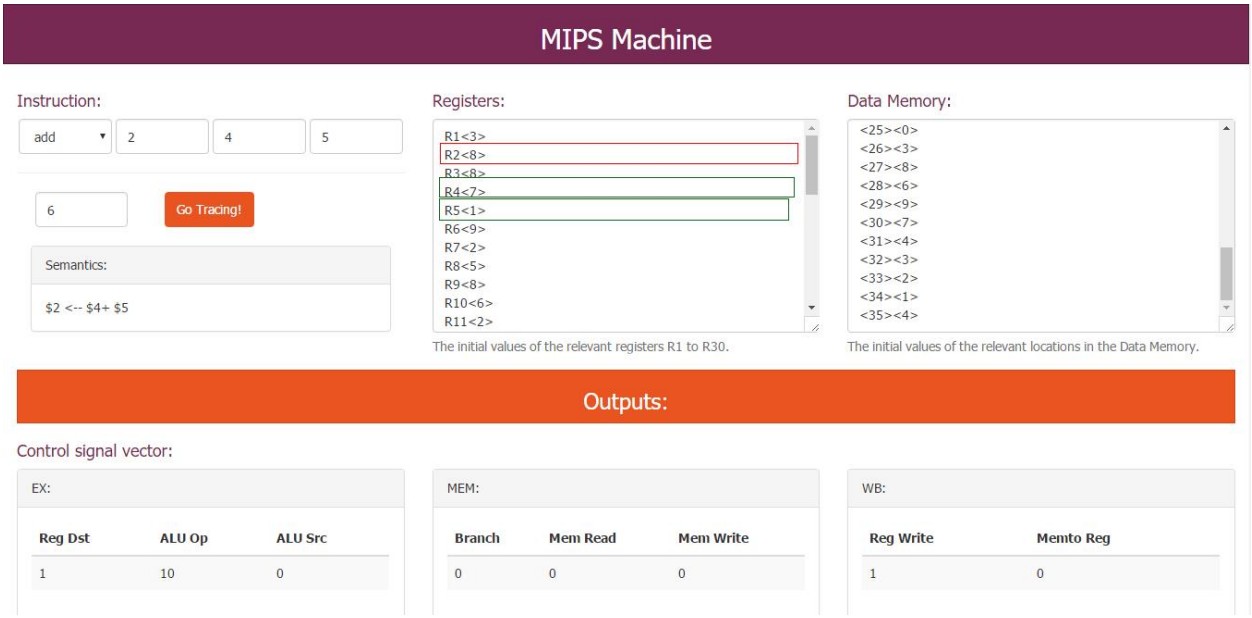

Figure 2- Results of control signal, pc, data memeory and register after tracing the 'Add' opcode

Figure 3 shows all buffers, data paths and data ports value during tracing the MIPS pipeline:

Data ports, Data paths and the interstage buffers:

IF Stage:

Read Address	MUX3-1	MUX3-0
2	-	6

IF/ID BUFFER:

op	rs	rt	rd	shamt	funct	pc
0	4	5	2	0	32	6

ID Stage:

[15..11]	[20..16]	[15..0]	Write Data	Write Register	Read Register2	Read Register1	Read Data2	Read Data1
2	-	32	8	2	5	4	1	7

ID/EX BUFFER:

Rd	[20..16]	[15..0]	Read Data1	Read Data2	pc
2	-	32	1	7	6

EX Stage:

MUX12-1	MUX12-0	ALU Control	MUX11-1	MUX11-0	ALU Input2	ALU Input1	ALU Result	Zero	Add Input2	Add Input1	Add Result
2	-	32	32	1	1	7	8	0	128	6	-

EX/MEM BUFFER:

MUX12	ReadData2	ALU Result	Zero	Add Result
2	1	8	0	-

MEM Stage:

Write Data	Write Address	Read Address	Read Data
1	8	8	-

MEM/WB BUFFER:

Rd	ALU Result	Read Data
2	8	-

WB Stage:

MUX14-0	MUX14-1	MUX14Result
8	-	8

Figure 3- Results of data ports, data paths and the interstage buffers after tracing the 'Add' opcode

The following sections include the same figures for sub, lw, sw and beq opcodes. Each section starts with an initialization figure, the next figure will represent the results of control signal, pc, data memeory and register and the last figure will depict the values for data ports, data paths and the interstage buffers after tracing each opcode.

SUB:

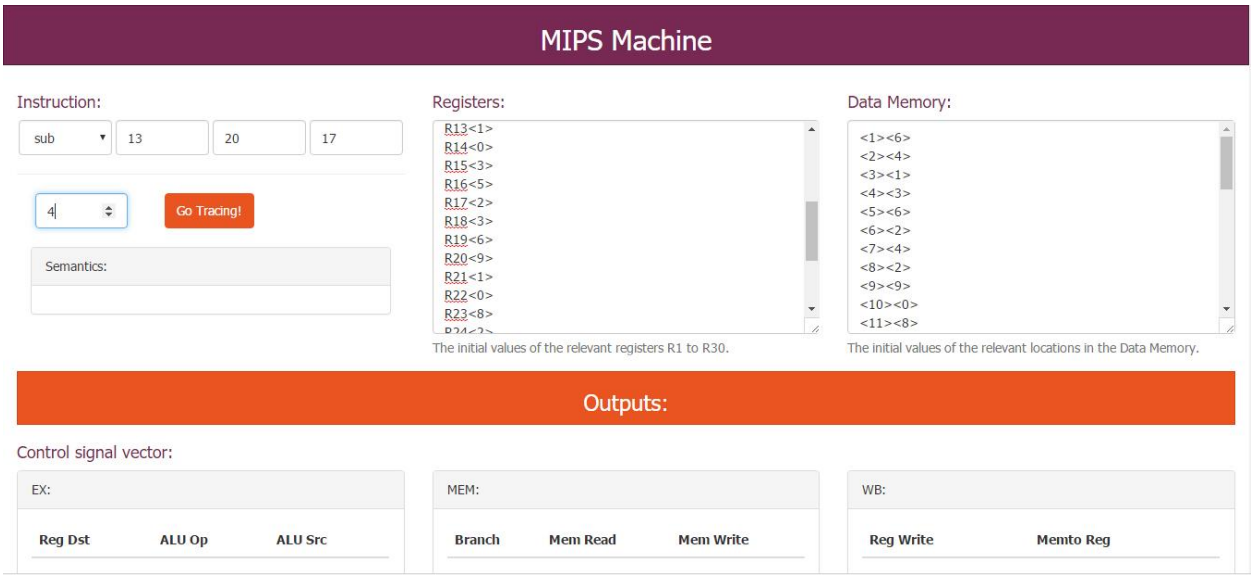

Figure 4- Initializing input for Sub opcode

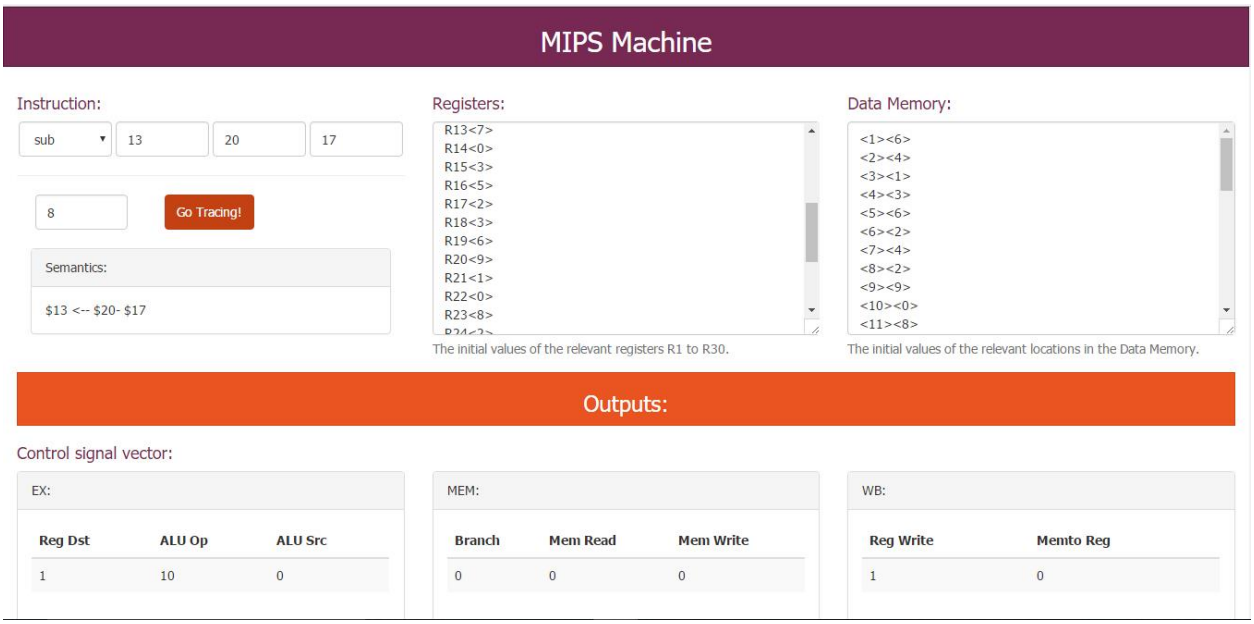

Figure 5- Results of control signal, pc, data memeory and register after tracing the 'Sub' opcode

Data ports, Data paths and the interstage buffers:

IF Stage:

Read Address	MUX3-1	MUX3-0
4	-	8

IF/ID BUFFER:

op	rs	rt	rd	shamt	funct	pc
0	20	17	13	0	34	8

ID Stage:

[15..11]	[20..16]	[15..0]	Write Data	Write Register	Read Register2	Read Register1	Read Data2	Read Data1
13	-	34	7	13	17	20	2	9

ID/EX BUFFER:

Rd	[20..16]	[15..0]	Read Data1	Read Data2	pc
13	-	34	2	9	8

EX Stage:

MUX12-1	MUX12-0	ALU Control	MUX11-1	MUX11-0	ALU Input2	ALU Input1	ALU Result	Zero	Add Input2	Add Input1	Add Result
13	-	34	34	2	2	9	7	0	136	8	-

EX/MEM BUFFER:

MUX12	ReadData2	ALU Result	Zero	Add Result
13	2	7	0	-

MEM Stage:

Write Data	Write Address	Read Address	Read Data
2	7	7	-

MEM/WB BUFFER:

Rd	ALU Result	Read Data
13	7	-

WB Stage:

MUX14-0	MUX14-1	MUX14Result
7	-	7

Figure 6- Results of data ports, data paths and the interstage buffers after tracing the 'Sub' opcode

Lw:

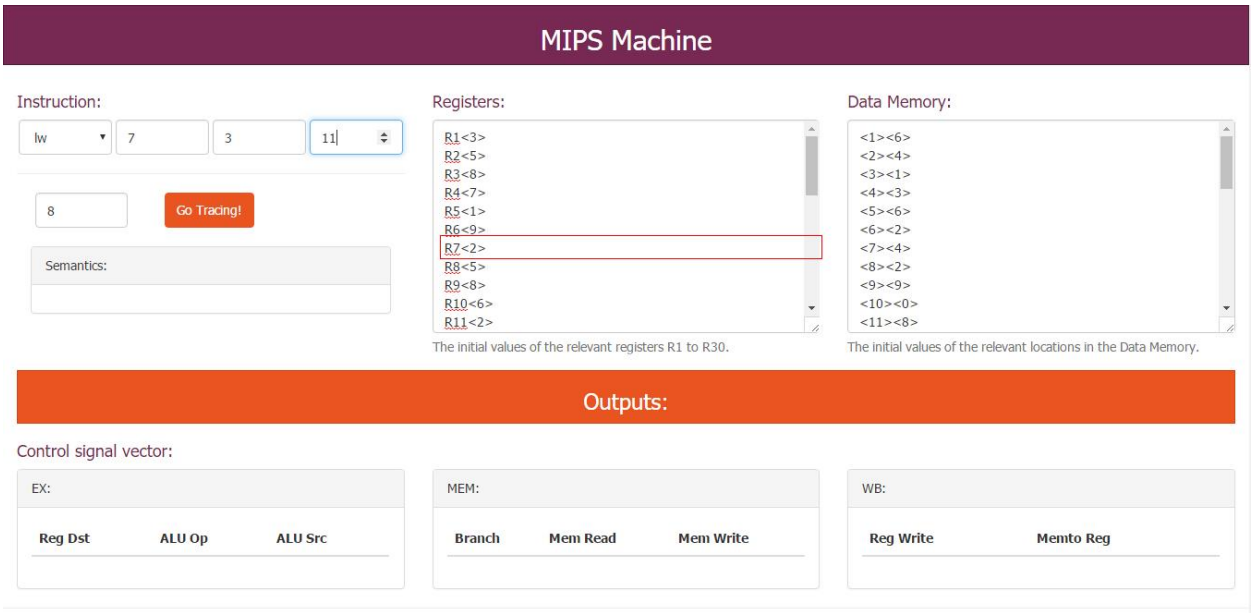

Figure 7- Initializing input for Lw opcode

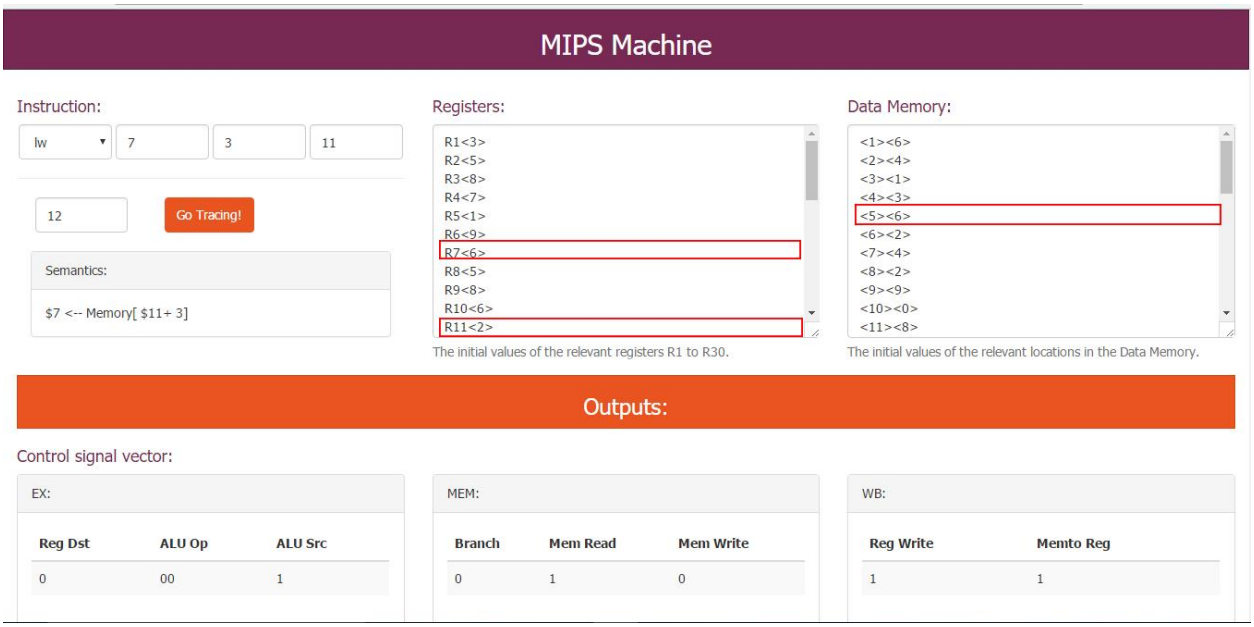

Figure 8- Results of control signal, pc, data memeory and register after tracing the 'Lw' opcode

Data ports, Data paths and the interstage buffers:

IF Stage:

Read Address	MUX3-1	MUX3-0
8	-	12

IF/ID BUFFER:

op	rs	rt	offset	pc
35	11	7	3	12

ID Stage:

[15..11]	[20..16]	[15..0]	Write Data	Write Register	Read Register2	Read Register1	Read Data2	Read Data1
-	7	3	6	7	7	11	2	2

ID/EX BUFFER:

Rd	[20..16]	[15..0]	Read Data1	Read Data2	pc
-	7	3	2	2	12

EX Stage:

MUX12-1	MUX12-0	ALU Control	MUX11-1	MUX11-0	ALU Input2	ALU Input1	ALU Result	Zero	Add Input2	Add Input1	Add Result
-	7	3	3	2	3	2	5	0	12	12	-

EX/MEM BUFFER:

MUX12	ReadData2	ALU Result	Zero	Add Result
7	2	5	0	-

MEM Stage:

Write Data	Write Address	Read Address	Read Data
2	5	5	6

MEM/WB BUFFER:

Rd	ALU Result	Read Data
7	5	6

WB Stage:

MUX14-0	MUX14-1	MUX14Result
5	6	6

Figure 9- Results of data ports, data paths and the interstage buffers after tracing the 'Lw' opcode

Sw:

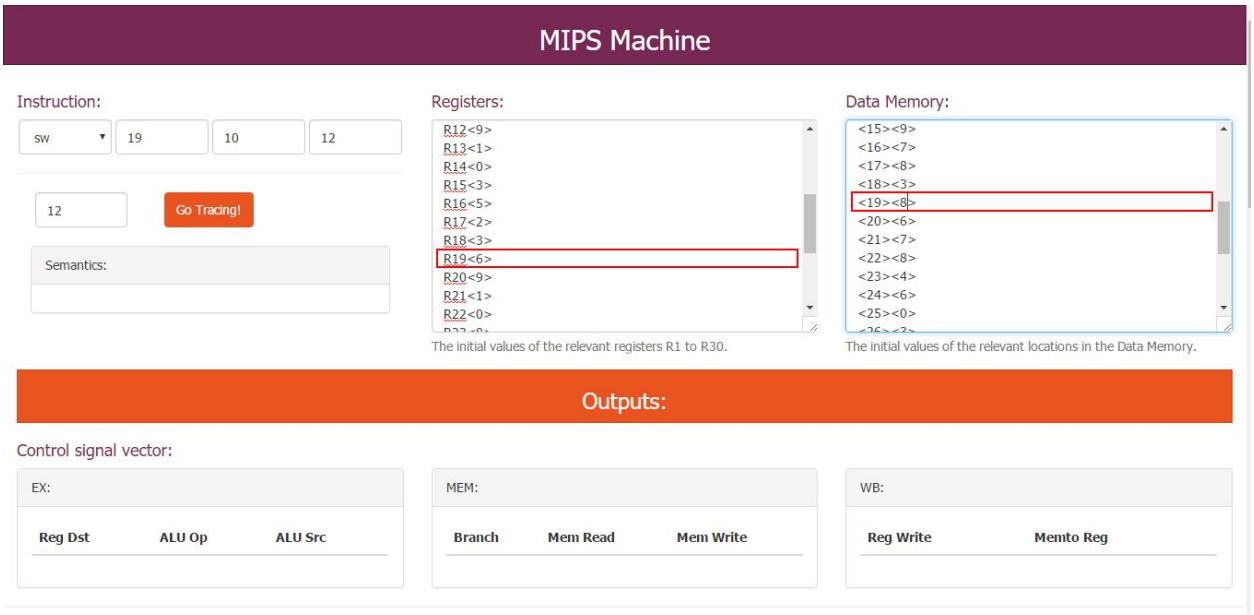

Figure 10- Initializing input for Sw opcode

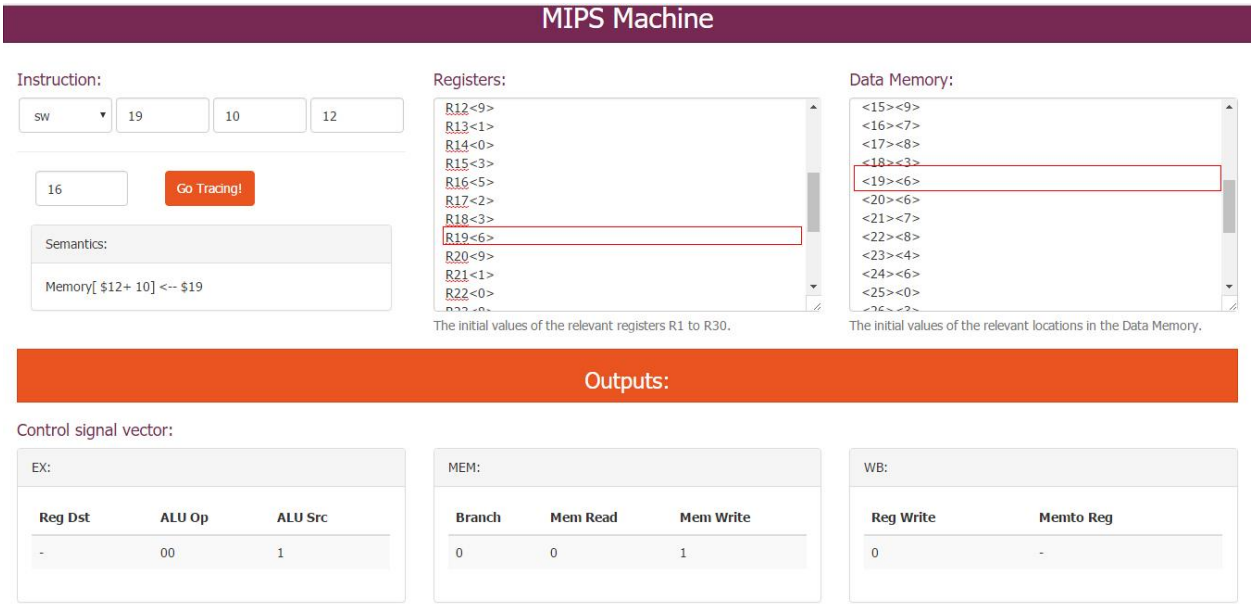

Figure 11- Results of control signal, pc, data memeory and register after tracing the 'Sw' opcode

Data ports, Data paths and the interstage buffers:

IF Stage:

Read Address	MUX3-1	MUX3-0
12	-	16

IF/ID BUFFER:

op	rs	rt	offset	pc
43	12	19	10	16

ID Stage:

[15..11]	[20..16]	[15..0]	Write Data	Write Register	Read Register2	Read Register1	Read Data2	Read Data1
-	19	10	-	-	19	12	6	9

ID/EX BUFFER:

Rd	[20..16]	[15..0]	Read Data1	Read Data2	pc
-	19	10	6	9	16

EX Stage:

MUX12-1	MUX12-0	ALU Control	MUX11-1	MUX11-0	ALU Input2	ALU Input1	ALU Result	Zero	Add Input2	Add Input1	Add Result
-	19	10	10	6	10	9	19	0	40	16	-

EX/MEM BUFFER:

MUX12	ReadData2	ALU Result	Zero	Add Result
-	6	19	0	-

MEM Stage:

Write Data	Write Address	Read Address	Read Data
6	19	19	-

MEM/WB BUFFER:

Rd	ALU Result	Read Data
-	19	-

WB Stage:

MUX14-0	MUX14-1	MUX14Result
19	-	-

Figure 12- Results of data ports, data paths and the interstage buffers after tracing the 'Sw' opcode

Beq:

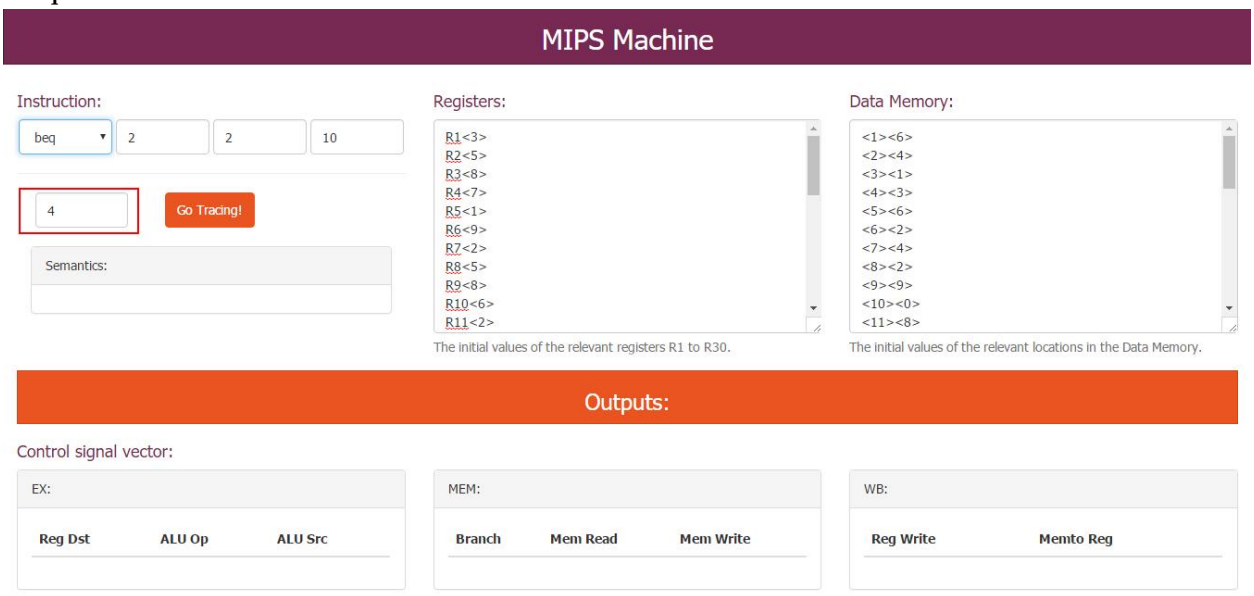

Figure 13- Initializing input for Beq opcode

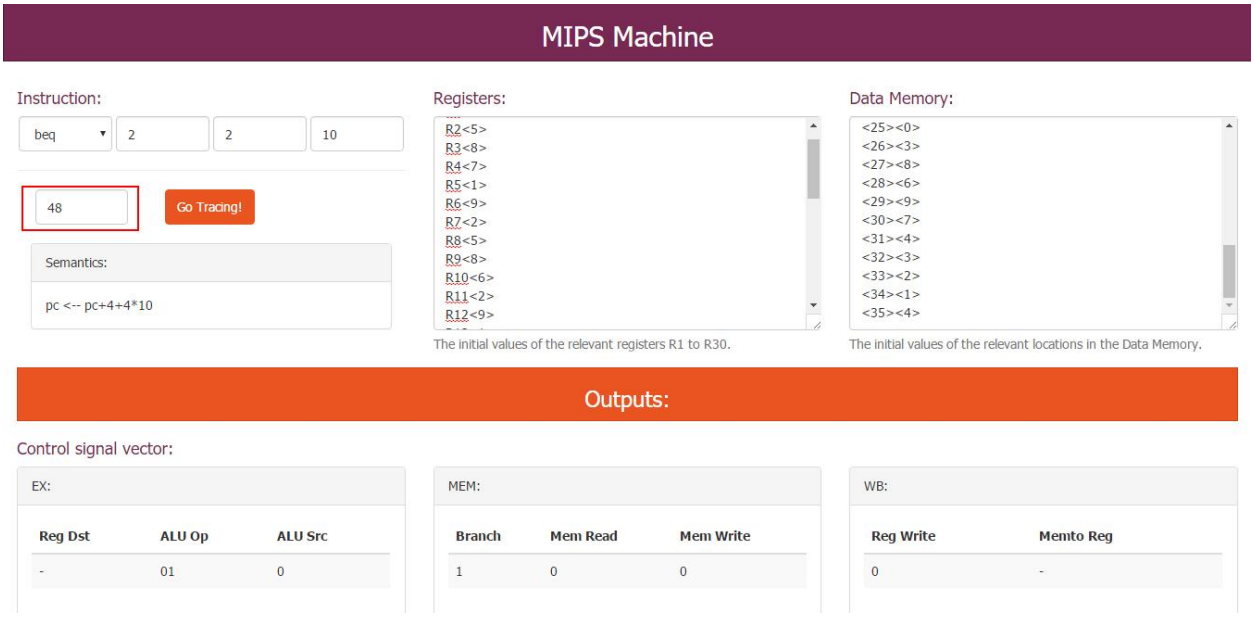

Figure 14- Results of control signal, pc, data memeory and register after tracing the 'Beq' opcode

Data ports, Data paths and the interstage buffers:

IF Stage:

Read Address	MUX3-1	MUX3-0
4	48	8

IF/ID BUFFER:

op	rs	rt	offset	pc
4	2	2	10	8

ID Stage:

[15..11]	[20..16]	[15..0]	Write Data	Write Register	Read Register2	Read Register1	Read Data2	Read Data1
-	2	10	-	-	2	2	5	5

ID/EX BUFFER:

Rd	[20..16]	[15..0]	Read Data1	Read Data2	pc
-	2	10	5	5	8

EX Stage:

MUX12-1	MUX12-0	ALU Control	MUX11-1	MUX11-0	ALU Input2	ALU Input1	ALU Result	Zero	Add Input2	Add Input1	Add Result
-	2	10	10	5	5	5	-	1	40	8	48

EX/MEM BUFFER:

MUX12	ReadData2	ALU Result	Zero	Add Result
-	5	-	1	48

MEM Stage:

Write Data	Write Address	Read Address	Read Data
5	-	-	-

MEM/WB BUFFER:

Rd	ALU Result	Read Data
-	-	-

WB Stage:

MUX14-0	MUX14-1	MUX14Result
-	-	-

Figure 15- Results of data ports, data paths and the interstage buffers after tracing the 'Beq' opcode