

Computer Organization and Design Project 2

implementation of a MIPS-like CPU

Ma Yesheng*
5140209064

December 31, 2015

1 introduction

In this lab, we are going to simulate a MIPS-like CPU using C++ language. We will implement the function of every component and bind them together to get a single-cycle data path.

2 lab environment

In this lab, we will implement the project on Microsoft Visual Studio 2012, using Visual C++ v11 compiler.

3 lab overview

3.1 logic structure

In a single cycle CPU, the essential components are program counter, register file, memory, controller, ALU and control path.

3.2 code framework

3.2.1 projects in solution

1. libalu: behaviour of CPU on change.
2. libalucontrol: behaviour of CPU when given different ALUctrl signals.
3. libcomponent: base class for components' in CPU
4. libcpu: bind data path and control path together.

*mayesheng@sjtu.edu.cn

5. libctr: determine instruction type on opcode field.
6. libmemory: memory behaviour on change.
7. libother: sign-extension and other components.
8. libpc: do $PC + 4$ or do branch.
9. libreg: simulate the behaviour of registers given RegWrite and RegIn.

3.2.2 base components in libcomponent

1. LineData: data transferred in data path.
2. EventListener: trigger function.
3. name: names for each component.
4. In/Out: map the data to I/O interfaces.
5. InputEvent/OutputEvent: map trigger functions to component names.
6. input: assign the value of the input of a component.
7. output: get the output of a component.
8. bind: bind the output of one component and the input of another component.
9. getListener: get the trigger function for a component.
10. setOutPut: set up the output of given component.
11. onChange/onClock: they are triggered when data are modified or when the rising edge comes.

4 code implementation

4.1 libCtr implementation

The main control is given opcode field of the instruction. Ctr decodes the instruction and send control signals to corresponding logic components.

input: opCode(6 bit)

output: regDst(1 bit), aluSrc(2 bit), memtoReg(1 bit), regWrite(1 bit), memRead(1 bit), memWrite(1 bit), branch(1 bit), aluOp(2 bit), jump(1 bit).

The decode table is already given in the reference book.

For example, when opcode is 101011, the code is as follows:

```

} else if (in[opCode] == B("101011")) {
    setOutput(jump, 0);
    setOutput(regDst, 0);
    setOutput(ALUSrc, 1);
    setOutput(memToReg, 0);
    setOutput(regWrite, 0);
    setOutput(memRead, 0);
    setOutput(memWrite, 1);
    setOutput(branch, 0);
    setOutput(aluOp, B("00") );
}

```

4.2 ALU Ctrl implementation

ALU Ctrl decides which ALU operation to do by ALUOp and funct field.

input: ALUOp(2 bit), funct(6 bit)

output: ALU Ctrl(4 bit)

The decode table is also listed in the reference book.

The code is as follows:

```

void ALUControl::onChange() {
    if (in[aluOp] == 0)
        setOutput(aluCtrl, B("0010"));
    else if (in[aluOp] == B("01"))
        setOutput(aluCtrl, B("0110"));
    else if (in[aluOp] == B("10"))
        if (in[funct] == B("0000"))
            setOutput(aluCtrl, B("0010"));
        else if (in[funct] == B("0010"))
            setOutput(aluCtrl, B("0110"));
        else if (in[funct] == B("0100"))
            setOutput(aluCtrl, B("0000"));
        else if (in[funct] == B("0101"))
            setOutput(aluCtrl, B("0001"));
        else if (in[funct] == B("1010"))
            setOutput(aluCtrl, B("0111"));
}

```

4.3 ALU implementation

Given the ALU control signal, implement the function on two operands.

input: input1(32 bit), input2(32 bit), aluCtrl(4 bit)

output: zero(1 bit), aluRes(32 bit)

The code is as follows:

```

void ALU::onChange() {
    LineData res = 0;
    if (in[aluCtr] == 0) {
        res = in[input1] & in[input2];
    } else if (in[aluCtr] == B("0000")) {
        res = in[input1] & in[input2];
    } else if (in[aluCtr] == B("0001")) {
        res = in[input1] | in[input2];
    } else if (in[aluCtr] == B("0010")) {
        res = in[input1] + in[input2];
    } else if (in[aluCtr] == B("0110")) {
        res = in[input1] - in[input2];
    } else if (in[aluCtr] == B("0111")) {
        res = (in[input1] < in[input2]) ? 1 : 0;
    } else if (in[aluCtr] == B("1100")) {
        res = ~(in[input1] | in[input2]);
    }
    setOutput(aluRes, res);
    if (res == 0) setOutput(zero, 1);
    else setOutput(zero, 0);
}

```

4.4 register implementation

simulate the register file in MIPS. Register data is stored in reg::memory and we have to implement the function of onClock and onChange.

input: readReg1(5 bit), readReg2(5 bit), writeReg(5 bit), writeData(32 bit), regWrite(1 bit)

output: readData1(32 bit), readData2(32 bit)

The code is as follows:

```

void Reg::onChange() {
    LineData regA = in[readReg1], regB = in[readReg2];
    setOutput(readData1, memory[regA]);
    setOutput(readData2, memory[regB]);
}
void Reg::onClock() {
    onChange();
    if (in[regWrite] == 1) {
        memory[in[writeReg]] = in[writeData];
    }
}

```

4.5 program counter implementation

In the PC section, we first do $PC + 4$, and then determine whether branch is required. Then, we update PC and prepare the program to receive the next instruction.

input: branch(1 bit), zero(1 bit), immData(28 bit)

output: newPC(32 bit), memRead(1 bit)

The code is as followed:

```
void PC::onClock() {
    m_pc += 4;
    if (in[branch] == 1 && in[zero] == 1) {
        m_pc += in[immData] << 2;
    }
    setOutput(newPC, m_pc);
    setOutput(memRead, 1);
}
```

5 integration of the whole single-cycle CPU

In this section, we are going to bind every component together to build a CPU that is able to run.

The components needed to be bound is as follows.

1. IMem with readReg1 and readReg2.
2. Imem with opcode.
3. Imem with immediate data.
4. Imem with funct field.
5. Imem with input1 and input2 of muxRegDst.
6. ctr with RegDst, jump, branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite.
7. muxRegDes with WriteReg in Reg.
8. muxALU with ALU input2
9. muxMentoReg with WriteData in Reg.
10. reg's ReadData1 with ALU input1, reg's ReadData2 with muxALU input2 and dataMem WriteData.
11. ALU's res with dataMem's address and muxMemtoReg input2.
12. ALU's zero output with PC's zero.

13. sign-extend with muxALU's input2 and PC.
14. ALUCtr with ALU's ctr signal input.
15. dataMem's ReadData with muxMemtoReg's input1.
16. PC with IMem's address and ReadData.

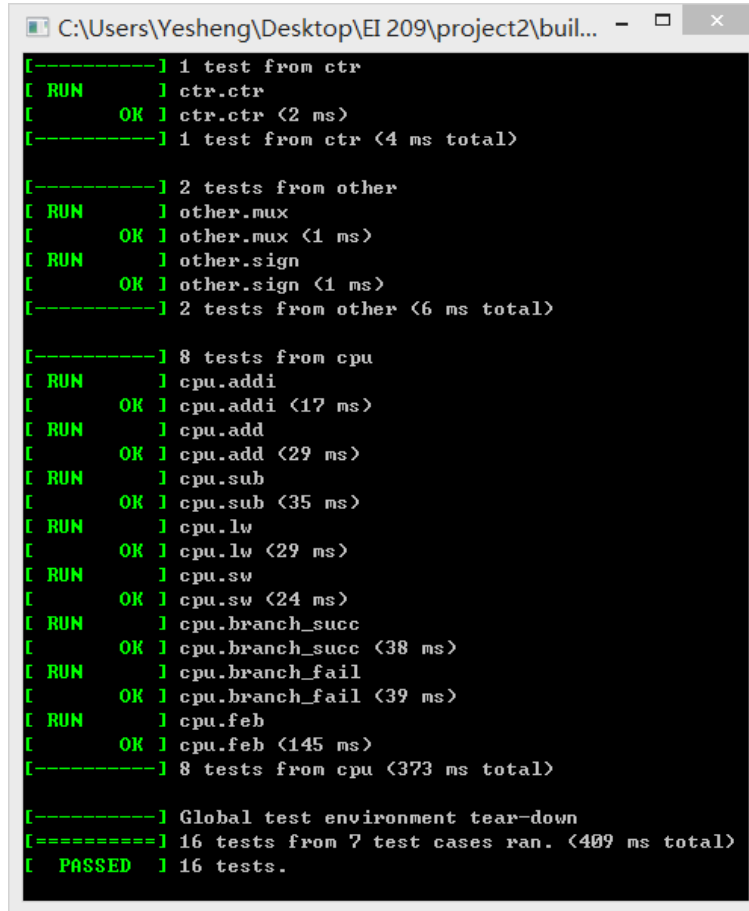
The code is shown is follows:

```
CPU::CPU() : pc(-4), instMem("Instruction Memory"), dataMem("Data Memory"),
    muxMem2Reg("muxMem2Reg"), muxAlu("muxAlu"), muxRegDes("muxRegDes") {
    instMem.bind(readData, partialListener(26, 31, ctr, opCode));
    instMem.bind(readData, partialListener(21, 25, reg, readReg1));
    instMem.bind(readData, partialListener(16, 20, reg, readReg2));
    instMem.bind(readData, partialListener(16, 20, muxRegDes, input1));
    instMem.bind(readData, partialListener(11, 15, muxRegDes, input2));
    instMem.bind(readData, partialListener(0, 15, signExtend, immInput));
    instMem.bind(readData, partialListener(0, 3, aluControl, funct));
    BIND(ctr, jump, pc, branch);
    BIND(ctr, regDst, muxRegDes, muxSel);
    BIND(ctr, branch, pc, branch);
    BIND(ctr, memRead, dataMem, memRead );
    BIND(ctr, memToReg, muxMem2Reg, muxSel);
    BIND(ctr, aluOp, aluControl, aluOp);
    BIND(ctr, ALUSrc, muxAlu, muxSel);
    BIND(ctr, regWrite, reg, regWrite);
    BIND(ctr, memWrite, dataMem, memWrite);
    BIND(aluControl, aluCtr, alu, aluCtr);
    BIND(muxRegDes, muxOut, reg, writeReg);
    BIND(muxAlu, muxOut, alu, input2);
    BIND(muxMem2Reg, muxOut, reg, writeData);
    BIND(reg, readData1, alu, input1);
    BIND(reg, readData2, muxAlu, input1);
    BIND(reg, readData2, dataMem, writeData);
    BIND(alu, aluRes, dataMem, address);
    BIND(alu, aluRes, muxMem2Reg, input1);
    BIND(alu, zero, pc, zero);
    BIND(signExtend, immData, muxAlu, input2);
    BIND(signExtend, immData, pc, immData);
    BIND(dataMem, readData, muxMem2Reg, input2);
    BIND(pc, newPC, instMem, address);
    BIND(pc, memRead, instMem, memRead);
```

Hence, the whole data path is completed.

6 test on the 16 test cases

As is shown in the picture bellow, all the 16 test cases run successfully.



```
C:\Users\Yesheng\Desktop\EI 209\project2\buil... - □ ×
[-----] 1 test from ctr
[ RUN      ] ctr.ctr
[      OK ] ctr.ctr <2 ms>
[-----] 1 test from ctr <4 ms total>

[-----] 2 tests from other
[ RUN      ] other.mux
[      OK ] other.mux <1 ms>
[ RUN      ] other.sign
[      OK ] other.sign <1 ms>
[-----] 2 tests from other <6 ms total>

[-----] 8 tests from cpu
[ RUN      ] cpu.addi
[      OK ] cpu.addi <17 ms>
[ RUN      ] cpu.add
[      OK ] cpu.add <29 ms>
[ RUN      ] cpu.sub
[      OK ] cpu.sub <35 ms>
[ RUN      ] cpu.lw
[      OK ] cpu.lw <29 ms>
[ RUN      ] cpu.sw
[      OK ] cpu.sw <24 ms>
[ RUN      ] cpu.branch_succ
[      OK ] cpu.branch_succ <38 ms>
[ RUN      ] cpu.branch_fail
[      OK ] cpu.branch_fail <39 ms>
[ RUN      ] cpu.feb
[      OK ] cpu.feb <145 ms>
[-----] 8 tests from cpu <373 ms total>

[-----] Global test environment tear-down
[=====] 16 tests from 7 test cases ran. <409 ms total>
[ PASSED ] 16 tests.
```

7 conclusion

This lab is the implementation of a MIPS-like single-cycle CPU. We use C++ to simulate the whole process. This is not only a challenge for us, but also prepare us for the upcoming course of computer organization lab. In this project, we simulate the behaviour of instruction memory, main control, several multiplexers, sign-extension unit, register file, ALU, data memory, ALU control and program counter. We have a better understanding of these hardware.

By the way, I think it is worth noting that there is a mistake in the reference book that the place of muxRegDst and muxALU are marked wrong, and is somewhat misleading. Also, the implementation of PC component should not be left over. Moreover, it seems that instructions such as addi is not mentioned

in the reference book and I waste much time repairing this bug.
In general, this project really practices us a lot. Anyway, thanks a lot to prof.
Zhu and TA's effort. It is really a fantastic class.