

CPU实验报告

罗璇, 5112409040
李青林, 5110309074

PS1:由于本报告是在开发过程中逐渐完成, 所以呈现中英混杂的结果, 请谅解

PS2:访问[这里](#)得到更好的排版效果

综述

我们写的是带reorder buffer的tomasulo

doc目录下, 除了5stage, isa和dependency graph, 其他的图片是我们的原件设计与连接图

Tomasulo With Reorder Buffer部分介绍了整个CPU的设计

Work Division部分介绍了分工

因为我们设计了一个代码重排的算法, code scheduling部分介绍了该算法。

doc目录下, code schedule.xlsx介绍了该算法的一个例子。dependency graph则是该例子中代码的依赖关系图。

isa部分介绍了我们使用的isa。我们的isa与mips类似, 但增加了双寄存器寻址的lwrr与swrr, 改变了寻址方式, 使得compiler也做了大范围的相应修改。

Tomasulo With Reorder Buffer

Unit List

1. reorder buffer (RB)
2. alu_rs (rs stands for reservation station)
 - can be used as mul or add,sub. But one alu_rs can be used for one purpose only, either mul,muli or add,addi,sub,subi
 - 3 alu_rs for mul,muli and 3 alu_rs for add,addi, sub, subi
3. 3 load_rs
4. 1 store_rs
5. 1 branch_rs
6. reg_status
7. reg_file
8. 1 data_cache
9. data_memory
10. 1 inst_cache
11. inst_memory
12. 1 CDB_data bus
 - it consists of 3 groups of wires:
 - wire[WORD_SIZE*RB_SIZE-1:0] CDB_data_data
 - data of each rs is written to it
 - wire[RB_SIZE-1:0] CDB_data_valid

- to show whether the data on CDB_data_data and CDB_data_addr is valid or not
- wire[WORD_SIZE*RB_SIZE-1:0] CDB_data_addr
 - store_rs put the address on CDB_data_addr, and the write data on CDB_data_data

13. 1 CDB_data_controller

- Since each rs can write to CDB_data bus, which will easily cause conflicts, we use CDB_data_controller to deal with all writes to the CDB_data bus.

14. 1 CDB_inst

- for RB to issue instruction to the corresponding function unit(fu)
- it consists of 3 groups of wires
 - wire[FU_INDEX-1:0] CDB_inst_fu (to which fu)
 - wire[WORD_SIZE-1:0] CDB_inst_inst (inst to issue)
 - wire[RB_INDEX-1:0] CDB_inst_RBindex (write result to this RB entry)

Plan for Each Crucial Stage

All writes occur at negedge while the command of write issue before this

1 IF

Reorder Buffer

```
@posedge:
    check if RB not full
        new PC put into PC
        get Instr from instr cache
@negedge:
    if (instr miss)
        wait until the 99th cycle's negedge

    if (j)
        pc = jump target
    else
        get the instr and add to RB's back
        Pc = Pc+1
```

Note: Pc can be affected by branch at write back cycle.

2 ISSUE

Reorder Buffer

```
@posedge
    if (inc(tail)) has instr
        // if (RB_valid[inc(tail)]), inc(tail) = (tail+1) %RB_SIZE
        issue if can
        tail = inc(tail)
```

issue if can

RB

```
@posedge
    if there's a corresponding FU not busy
        <FU, RB entry index, inst> put onto CDB_inst
        issue the command of updating reg_status
```

Note: When "write-back" wants to update the same reg status, do not write back.

Reg_status

```
@negedge
    update Register status
```

RS

```
@posedge

    if (!busy)
        #0.1 if see fu on CDB_inst_fu == RS's fuindex
            update
                busy, op,
                invalld(set the correspodng CDB_data_valid to invalid
                dest(the RB entry that issued the command)
            check corresponding register status to update Qj, Qk, Vj
            for i, j, k
                if (Q ready)
                    put the data onto V
                else check corresponding reorder buffer entry's CDB
                    if (ready)
                        put the data on V
                    else set wait for index

            for branch_rs
                it will execute and set CDB_data_data (jump? 1:0)
```

3 execute

each RS (eg. a mul has 3 RS, each work independently)

```
@posedge
    // since busy set at #0.1 after posedge
    // for a newly issued op, it'll wait until the next cycle to
    // put... onto CDB_data... is a command issued to CDB_data_
    if (busy)
        if(Qj and Qk are both ready) {
            if (add or sub or branch)
                #0.1 set CDB_data_valid valid
                put the result onto CDB_data_data
            else if (mul)
                #3.1 set CDB_data_valid valid
```

```

        put the result onto CDB_data_data
    else if (load buffer) {
        #0.5 if (hit)
            read cache data
        else
            #99/*at the 99th cycle's negedge*/ read cache da

        set CDB_data_valid valid
        put the data onto CDB_data_data
    } else if (store buffer) {
        // mem[Qj+Qk] = Qi
        if (Qi ready)
            #0.1/*at negedge*/ put Qj+Qk onto the CDB_data_a
            put Qi onto CDB_data_addr
    }
}

```

4 write back

Reorder Buffer

@negedge

```

#0.1 check each entry's corresponding CDB
    if (there's a branch which wants to jump) {
        pc = jump target // since it update pc after the IF
    }

```

@posedge

```

if (RB_data_valid[head])
    if (write to reg && the reg is still waiting for its data)
        write it to the register file
        set corresponding register status to empty
    else if (write to cache) {
        if (cnt_enable && cnt < MEM_STALL)
            #(MEM_STALL-cnt) begin end
        cnt = 0;
        cnt_enable = 1'b0;

        we_mem = 1'b1;
        wd_mem = RB_data[head];
        ws_mem = RB_addr[head];

        #0.6 if (!mem_hit) {
            cnt = 1;
            cnt_enable = 1'b1;
        }
        // for the purpose of having the cnt and cnt_enable, see
    }
}

```

Hardware Optimization

Reorder Buffer

Using 3 pointers--head, tail and back--instead of the only 2 pointers -- head and tail-- normally used in RB.

Back is to load instruction in advance to reduce stall that may encounter if an instruction is only loaded when tail is about to issue it.

Branch

The branch_rs will check its data availability at issue cycle. So if the data is available at the issue cycle, there'll be only 1 stall.

Write back

When writing back to data cache and a write miss occurs, reorder buffer will not just wait until the write is done. It will still write to register file and will only stall at the next write to the memory.

Data cache

The data cache has one write port and 3 read ports corresponding to the one store_rs and the 3 load_rs, so that the 3 load_rs can load data simultaneously.

ISA

instructions

add sub mul lw sw addi subi muli lwrr swrr li j jr bge

add

calculate $op1 + op2$ and save the result into dst

usage

```
add dst, op1, op2
```

```
dst: register
op1: register
op2: register
```

binary

```
0000_dst(5bits)_op1(5bits)_op2(5bits)_0(13bits)
```

sub

calculate $op1 - op2$ and save the result into dst

usage

```
add dst, op1, op2
```

```
dst: register
```

```
op1: register
```

```
op2: register
```

binary

```
0001_dst(5bits)_op1(5bits)_op2(5bits)_0(13bits)
```

mul

calculate $op1 * op2$ and save the result into dst

usage

```
mul dst, op1, op2
```

```
dst: register
```

```
op1: register
```

```
op2: register
```

binary

```
0010_dst(5bits)_op1(5bits)_op2(5bits)_0(13bits)
```

lwrr

load form address $base + offset$ into dst

usage

```
add dst, base, offset
```

```
dst: register
```

```
base: register
```

```
offset: register
```

binary

```
0011_dst(5bits)_op1(5bits)_op2(5bits)_0(13bits)
```

swrr

save dst to address $base + offset$

usage

```
add dst, base, offset
```

```
dst: register
```

```
base: register
```

```
offset: register
```

binary

0100_dst(5bits)_op1(5bits)_op2(5bits)_0(13bits)

addi

calculate $op1 + op2$ and save the result into dst

usage

```
add dst, op1, op2
```

dst: register

op1: register

op2: immediate

binary

0101_dst(5bits)_op1(5bits)_op2(18bits)

subi

calculate $op1 - op2$ and save the result into dst

usage

```
add dst, op1, op2
```

dst: register

op1: register

op2: immediate

binary

0110_dst(5bits)_op1(5bits)_op2(18bits)

mul

calculate $op1 * op2$ and save the result into dst

usage

```
mul dst, op1, op2
```

dst: register

op1: register

op2: immediate

binary

0111_dst(5bits)_op1(5bits)_op2(18bits)

lw

load from address `base+offset` into `dst`

usage

```
add dst, base, offset
```

```
dst: register
```

```
base: register
```

```
offset: immediate
```

binary

```
1000_dst(5bits)_op1(5bits)_op2(18bits)
```

sw

save `dst` to address `base+offset`

usage

```
add dst, base, offset
```

```
dst: register
```

```
base: register
```

```
offset: immediate
```

binary

```
1001_dst(5bits)_op1(5bits)_op2(18bits)
```

li

load an immediate into `dst`

usage

```
li dst, imm
```

```
dst: register
```

```
imm: immediate
```

binary

```
1010_dst(5bits)_imm(23bits)
```

j

jump to a label

usage

```
j label
```


label: label

binary

1011_pc-offset (28bits)

jr

jump to address of dst

usage

jr dst

dst: register

binary

1100_dst (5bits) _0 (23bits)

bge

branch if $op1 \geq op2$

usage

bge op1, op2, label

op1: register

op2: immediate

label: label

binary

1101_op1 (5bits) _op2 (10bits) _pc-offset (13bits)

halt

halt

usage

halt

binary

1110_00...0 (28bits)

Work Division

Luo Xuan

Compiler

1. design a code scheduling algorithm(see code scheduling.mkd for an example and short description)
2. dead code elimination
3. strength reduction

eg.

```
i = i*2
```

would be compiled as ``i = i+i``,

```
j = i+0
```

would be compiled as ``move j, i``

(although this will not be used in the target code of this proj)

CPU

most of the initial designing of the protocols between the components

1. reorder_buffer.v
2. ALU_RS.v
3. store_RS.v
4. CDB_data_controller.v
5. CPU.v
6. def_param.v
7. parameters.v
8. reg_file_RX.v
9. reg_status.v
10. timescale.v

Li Qinlin

most of the test work - test matrix of different data and different sizes

CPU

1. reorder_buffer.v
2. branch_RS.v
3. load_RS.v
4. CPU.v
5. data_cache.v
6. data_memory.v
7. inst_cache.v
8. inst_memory.v
9. def_param.v
10. parameters.v

Assembler

1. translate code.asm into binary code

Test

1. rakefile

- to simplify the compilation of verilog codes