Architecture Lab1 Report--Part 2

姜海天 19307110022

Stage 1: Figure out `loop.s`

Only by figuring out what `loop.s` is doing and what the value of `16($r30)` is can we test the correctness of out pipeline CPU. Reference: [MIPS Instruction Reference]()

Here is the C code I reversed from `loop.s`.

```c
// pseudo-code.c
int main(){
    int result = 0;  // 16($fp)
    int tmp = 0;  // r6

    /* detailed version like the assembly code
    for (int r2 = result; r2 < 10; ){  // r3 = (r2 < 10)
        int r4 = r2;
        int r5 = r4 << 2;
        r4 = r5 - 13;
        tmp += r4;
        int r3 = result;
        r2 = r3 + 1;
        result = r2;
    }  */

    // human-readable version
    for (; result < 10; ++result){
        tmp += (result << 2) - 13;
    }
    result = tmp & 0xffff;
    return 0;
}
```

Running the `pseudo-code.c`, we can get the final result of `16($r30)` is 50.

```
result      r4      tmp
0           -13     -13
1           -9      -22
2           -5      -27
3           -1      -28
4           3       -25
5           7       -18
6           11      -7
7           15      8
8           19      27
9           23      50
10          /       /
```

Stage 2: Reading source code

Useful source code fragments for coding, stored here for check.

```c
// part of file: host.h
typedef int bool_t;          /* generic boolean type */
typedef unsigned char byte_t;       /* byte - 8 bits */
typedef signed char sbyte_t;
typedef unsigned short half_t;      /* half - 16 bits */
typedef signed short shalf_t;
typedef unsigned int word_t;        /* word - 32 bits */
typedef signed int sword_t;
```

```c
// part of file: machine.h
typedef word_t md_addr_t;
typedef struct {
  word_t a;     /* simplescalar opcode (must be unsigned) */
  word_t b;     /* simplescalar unsigned immediate fields */
} md_inst_t;

/* integer register specifiers */
#define RS      (inst.b >> 24)          /* reg source #1 */
#define RT      ((inst.b >> 16) & 0xff)     /* reg source #2 */
#define RD      ((inst.b >> 8) & 0xff)      /* reg dest */
/* returns shift amount field value */
#define SHAMT       (inst.b & 0xff)
/* returns 16-bit signed immediate field value */
#define IMM     ((int)((/* signed */short)(inst.b & 0xffff)))
#define UIMM        (inst.b & 0xffff)
/* returns 26-bit unsigned absolute jump target field value */
#define TARG        (inst.b & 0x3ffffff)

void
md_print_insn(md_inst_t inst,       /* instruction to disassemble */
          md_addr_t pc,     /* addr of inst, used for PC-rels */
          FILE *stream);        /* output stream */
```

```c
// part of file: sim-pipe.h
#define MD_FETCH_INSTI(INST, MEM, PC)                    \
  { INST.a = MEM_READ_WORD(mem, (PC));                   \
    INST.b = MEM_READ_WORD(mem, (PC) + sizeof(word_t)); }
#define SET_OPCODE(OP, INST) ((OP) = ((INST).a & 0xff))
```

```c
// part of file: sim-pipe.c
#define SET_NPC(EXPR)       (regs.regs_NPC = (EXPR))
/* current program counter */
#define CPC             (regs.regs_PC)
/* general purpose registers */
#define GPR(N)              (regs.regs_R[N])
#define SET_GPR(N,EXPR)     (regs.regs_R[N] = (EXPR))
#define INC_INSN_CTR()  sim_num_insn++
```

Stage 3: Sequential+jump, pipeline, no hazard guard

In this stage, I implemented a naïve pipelined CPU that cannot handle branch, jump or any hazard.

3.1 Data structure of buffers

Every step of the execution process in the CPU is required to be printed. `machine.h` provided a function `md_print_insn(md_inst_t inst, md_addr_t pc, FILE *stream)` for printing the instruction being executed. The function takes the instruction and the address of the instruction as parameters, so we have to record these values in the buffers. Therefore, `md_inst_t inst;` and `md_addr_t pc;` appear in all buffer struct.

1. For `ifid_buf`, only the two above are sufficient, because `NPC` can is not cared by the latter pipelines.
2. For `idex_buf`, `rs` and `rt` are needed to store the number of register for hazard checking; `rs_val=GPR(rs)` and `rt_val=GPR(rt)` are needed to pass the value read from register file; `opcode` is needed for transferring the opcode decoded during instruction decoding pipeline.
3. For `exmem_buf`, we can get the information about whether the instruction need `load_mem` or `write_rt` or `write_rd` from the execution process, and `write_enable` signal is also generated in execution. Also, we need to store `ALU_out`. If it is a `sw` instruction, we need `data_in` to store the value we want to write into memory. We also have to store the number of register in `rs` and `rt` for future hazard checking.
4. For `memwb_buf`, we need `from_mem` and `ALU_out` to store the data we may write, and `write_rt`, `write_rd`, `load_mem` to know where and which to write. We also have to store the number of register in `rs` and `rt` for future hazard checking.
5. For `wb_buf`, nothing more is needed, `inst` and `pc` are used for `print_trace`.

3.2 Main loop

Now that the buffers are set, every part of the pipeline can take the value it need from the former buffer and write the answers to the latter buffer. So in each cycle, the 5 pipelines can do their jobs separately. But the order is crucial. A correct order should be like this.

```c
while (TRUE){
    INC_INSN_CTR();
    do_wb();
    do_mem();
    do_ex();
    do_id();
    do_if();
    print_trace();
}
```

Basically, the order of the pipelines is the reverse order. This is because the buffers should be used before updated. What's more, write register before read register(instruction decoding) can naturally avoid structural hazard.

In physical word, this "reverse order" is implemented because the buffers' value will not be updated until the rising edge of clock.

3.3 Bug I met

Because the naïve CPU only supports sequential and jump instructions, the code for register buffer transferring is rather simple. With the data structure of buffers and the source code, I think it is quite easy to understand, so I won't tell the details about the implementation. I will only show the bug I have met.

```c
void do_if(){
    CPC = regs.regs_NPC;
    md_inst_t inst;
    MD_FETCH_INSTI(inst, mem, CPC)
    int opcode;
    MD_SET_OPCODE(opcode, inst);
    if (opcode == JUMP){  // no other work needed for JUMP instruction
        SET_NPC((CPC & 0xf0000000) | (TARG << 2));
    }else{
        SET_NPC(CPC + sizeof(md_inst_t));
    }
    fd.inst = inst;
    fd.pc = CPC;
}
```

The naïve `do_if` is implemented like this. After running the `sim-pipe-naive`, I found that the first instruction of my test assembly code cannot be executed. So I checked about the `CPC` and found that the given `simpipe.c` did the job in a way different from me. The original `regs.regs_NPC = regs.regs_PC + sizeof(md_inst_t);` before the while-true loop should be changed into `regs.regs_NPC = regs.regs_PC;`. After that, the `sim-pipe-naive` can work as desired, namely a processor that cannot deal with branch or data hazard.

The test assembly code:

```
    .global __start
__start:
    nop
    li   $fp,0x7fff0000
    nop
    nop
    sw   $fp,16($fp)
    j    $L1
    li   $6,0
$L1:
    nop
    lw   $2,16($fp)
    nop
    nop
    slt $3,$2,10
    syscall
```

3.4 Currently supported instructions

`ADD`, `ADDU`, `SUBU`, `ADDUI`, `ADDi`, `LUI`, `LW`, `SW`, `SLL`, `SLTI`, `JUMP`.

The source code of `sim-pipe-naive.c` will be enclosed.

Stage 4: Modify naïve CPU to be pipeline with stall

Here, I will implement a workable pipeline using bubbles.

4.1 Solve data hazard

To solve data hazard using bubbles, we shall first detect data hazard and then set the control signals(buffer registers) so that the CPU acts like executing a `NOP` instruction. So we need a `check_and_set` before executing the five stage in every cycle.

If the register to be read in instruction decoding stage (that is `de.rs` and `de.rt`) is the same as the register to be written in latter stage(former instruction), then there is data hazard. So we can simply check registers to be written(`de.rs`, `de.rt`) with the register to be read(`em.rd`, `em.rt`, `mw.rd`, `mw.rt`).

After detecting the data hazard, we will take the instruction in ID stage back to IF stage to start over again, and set the instruction in ID stage as `NOP` to pass the bubble.

So here is the `check_and_set`

```c
void check_and_set(){
    bool_t em_rt_hazard=em.write_rt&&((de.rs==em.rt)||(de.rt==em.rt));
    bool_t em_rd_hazard=em.write_rd&&((de.rs==em.rd)||(de.rt==em.rd));
    bool_t mw_rt_hazard=mw.write_rt&&((de.rs==mw.rt)||(de.rt==mw.rt));
    bool_t mw_rd_hazard=mw.write_rd&&((de.rs==mw.rd)||(de.rt==mw.rd));
    /* explanation for checking inst.a != NOP:
    if inst.a == NOP, the register still can be some meaningful register
    number because the NOP can be set by CPU after detecting data hazard
    but our CPU doesn't change other field. So we should make sure that
    the data hazard we detect is not from a NOP instruction set by CPU*/
    bool_t hazard_in_em=(em.inst.a!=NOP)&&(em_rt_hazard||em_rd_hazard);
    bool_t hazard_in_mw=(mw.inst.a!=NOP)&&(mw_rt_hazard||mw_rd_hazard);
    if (hazard_in_em){
        printf("em\n");
        fd.inst = de.inst;
        fd.pc = de.pc;
        de.inst.a = NOP;
        de.opcode = 0; // opcode for NOP
        em.write_enable = 0;
        SET_NPC(CPC);
    }
    if (hazard_in_mw){
        printf("mw\n");
        fd.inst = de.inst;
        fd.pc = de.pc;
        de.inst.a = NOP;
        de.opcode = 0;
        SET_NPC(CPC);
    }
}
```

4.2 Bug met here

A first silly bug is that in the line 11 and line 12, what I first wrote was:

```
bool_t hazard_in_em=(em.inst.a!=NOP)&&(em_rd_hazard||em_rd_hazard);
bool_t hazard_in_mw=(mw.inst.a!=NOP)&&(mw_rd_hazard||mw_rd_hazard);
```

It was a typo, so the hazard in register specified by rt will not be detected.

The second bug is a tricky one. What I wrote first was:

```
void check_and_set(){
    ...
    if (hazard_in_em){
        fd.inst = de.inst;
        fd.pc = de.pc;
        de.inst.a = NOP;
        de.opcode = 0;
        em.write_rd = 0;  // buggy line
        em.write_rt = 0;  // buggy line
        em.write_enable = 0;
        SET_NPC(CPC);
    }
}
```

These 2 lines are redundant and harmful. If they were executed, the data hazard in memory stage will not be detected because `mw.write_rd` and `mw.write_rt` are inherited from `em.write_rd` and `em.write_rt` in `do_mem()`. But we need them to be the original value so that we can detect data hazard. From another perspective, what `em.write_rd` and `em.write_rt` does is to pass the value to `mw`, so they should remain unchanged.

After finding these two bugs, the CPU works as desired. We now only have branch and the consequent control hazard to do.

The test assembly code for the CPU at present:

```
    .global __start
__start:
    li   $sp,0x7fff0000
    move    $fp,$sp         # data hazard
    sw   $fp,16($fp)     # data hazard
    lw   $5,16($fp)
    j    $L1
    li   $6,0
$L1:
    lw   $2,16($fp)
    slt $3,$2,10        # data hazard
    syscall
```

4.3 Add support for branch (`bne`)

If we don't use data forwarding, we will find the control hazard two cycles later after the instruction passed execute pipeline.

Firstly, I add `em.go_branch` and `em.branch_addr` to record the information after ALU do the arithmetic for `bne`. And I added some judgements in `do_if()`, `do_id()` and `do_exe()` so that they can fetch the correct instruction desired by `bne` instruction and insert bubbles.

```
void do_if(){
    if (em.go_branch){  // do the branch
        CPC = em.branch_addr;
    }else{
        CPC = regs.regs_NPC;
    }...
}
void do_id(){
    if (em.go_branch) {  // insert bubble
        de.inst.a = NOP;
        de.pc = fd.pc;
        return;
    }...
}
void do_ex(){
    if (em.go_branch) {  // insert bubble
        em.inst.a = NOP;
        em.pc = de.pc;
        return;
    }...
    switch (de.opcode){
        ...
        case BNE:
            if (de.rs_val != de.rt_val){
                em.go_branch = 1;
                em.branch_addr = em.pc + sizeof(md_inst_t) + (OFS << 2);
            }
            break;
    }
}
```

But the CPU cannot work correctly. To be more exact, that is the CPU will not halt and the target instruction of `bne` will be fetched one cycle earlier.

After pondering on this bug before sleep, the next day, I figured out that since in one cycle, my pipelines are doing reversely (`wb` -> `mem` -> `exe` -> `id` -> `if`), what I have done in `exe` will immediately affect the `id` and `if` in the same cycle, which is not what the CPU should do. In physical world, these five parts do work together, so there is no worry like this, and that is also why the bug is here, because I figured out how to avoid control hazard based on real CPUs. If the reason why the bug appear(the discussion before) is clear, it is quite easy to solve the bug. We just need to know whether it is the same cycle after `do_exe` or the next cycle. Correct fetch and insert bubbles should be done in next cycle. The solution is to set `em.go_branch=1` if `bne` goes to branch, and set `em.go_branch=2` in the next cycle if `em.go_branch` is already 1. And all the fetching and inserting bubbles will be done only when `em.go_branch` is 2, which ensures they are done in the next cycle.

The bug-free version of dealing with control hazard:

```
void do_if(){
    if (em.go_branch == 2){
        CPC = em.branch_addr;
        em.go_branch = 0;
    }else if (em.go_branch == 1){
        em.go_branch = 2;  // set the signal so that go to branch in next cycle
        CPC = regs.regs_NPC;
    }else{
        CPC = regs.regs_NPC;
    }
}
void do_id(){
    if (em.go_branch == 2) {
        de.inst.a = NOP;
        de.pc = fd.pc;
        return;
    }
}
void do_ex(){
    if (em.go_branch == 2) {
        em.inst.a = NOP;
        em.pc = de.pc;
        return;
    }
}
```

Now, I have finished the design of the CPU `sim-pipe-withstall`, the source code is enclosed in `sim-pipe-withstall.c`. Also, it worked well for `loop.s`, the trace file `trace-withstall.txt` will also be enclosed.

Hooray!

4.4 Currently supported instructions

`ADD`, `ADDU`, `SUBU`, `ADDUI`, `ADDi`, `LUI`, `LW`, `SW`, `SLL`, `SLTI`, `JUMP`, `BNE`.

Stage 5: Pipeline with data forwarding

This part is relatively easy, because I only need to modify the `check_and_set()` function. There are two kinds of data hazard. The data hazard regarding read the register in use can be fully solved without stalling by data forwarding. But the use after load can not be solved in this way, because the usable data is got after memory stage, not execute stage. So I split the two of them into two functions.

```c
void load_use_stall(){
    // only load-use hazard needs stalling
    if(em.inst.a != NOP && em.load_mem && (em.rt == de.rs || em.rt == de.rt)) {
        fd.inst = de.inst;
        fd.pc = de.pc;
        de.inst = MD_NOP_INST;
        de.opcode = 0;
        SET_NPC(CPC);
    }
}

void forward(){
    bool_t em_rt_hazard = em.write_rt && ((de.rs == em.rt) || (de.rt == em.rt));
    bool_t em_rd_hazard = em.write_rd && ((de.rs == em.rd) || (de.rt == em.rd));
    bool_t mw_rt_hazard = mw.write_rt && ((de.rs == mw.rt) || (de.rt == mw.rt));
    bool_t mw_rd_hazard = mw.write_rd && ((de.rs == mw.rd) || (de.rt == mw.rd));
    bool_t hazard_in_em = (em.inst.a != NOP) && (em_rt_hazard || em_rd_hazard);
    bool_t hazard_in_mw = (mw.inst.a != NOP) && (mw_rt_hazard || mw_rd_hazard);

    if (hazard_in_em){
        if ((em.write_rt && em.rt == de.rs) || (em.write_rd && em.rd == de.rs)){
            de.rs_val = em.ALUout;
        }else if ((em.write_rt && em.rt == de.rt) || (em.write_rd && em.rd == de.rt)) {
            de.rt_val = em.ALUout;
        }
    }else if(hazard_in_mw) {
        if ((mw.write_rt && mw.rt == de.rs) || (mw.write_rd && mw.rd == de.rs)) {
            de.rs_val = mw.load_mem?mw.from_mem:mw.ALUout;
        }else if ((mw.write_rt && mw.rt == de.rt) || (mw.write_rd && mw.rd == de.rt)) {
            de.rt_val = mw.load_mem?mw.from_mem:mw.ALUout;
        }
    }
}
```

This code can go through the test file `loop.s` correctly. The complete source code will be enclosed.