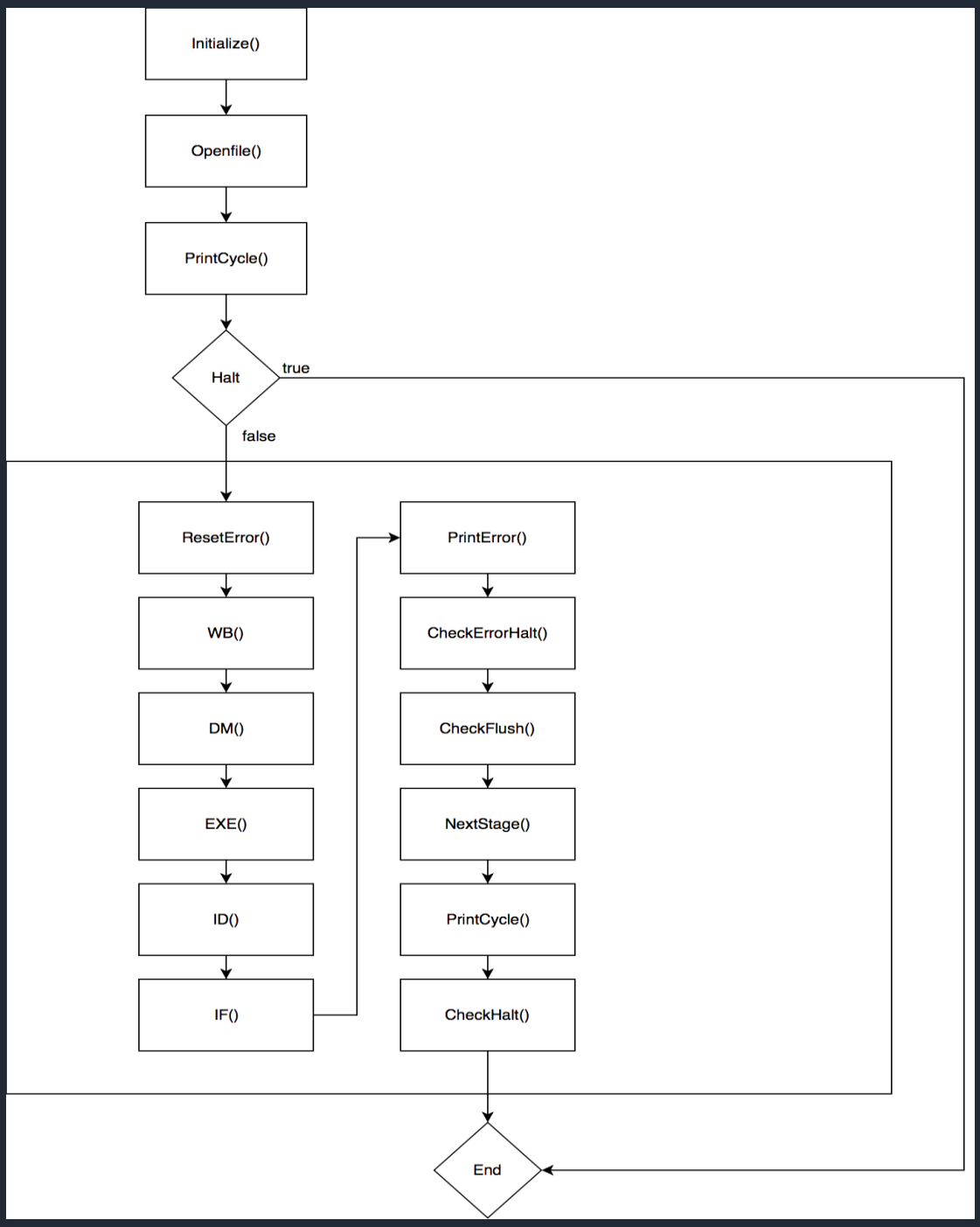
Archi_2017_Project_2

104062206 蔡鎮宇

Flow chart:



IF:

Decode current line of instruction.Including inst.name,inst.rs,inst.rd,inst.rt……
The store the result into **Simulator::IF_ID.inst.**

ID:

1. Set **inst** to **Simulator::IF_ID.inst.**
2. If **STALLED ,** set **Simulator::IF_ID.inst.** to **NOP.**
3. Forwarding (The process of check forwarding is done in previous cycle so here we only need to use the result) , If **inst.fwdrs or inst.fwdrt** is **true , then Data.Rs(t) = Simulator::MEM_WB.ALU_result** otherwise it's **Simulator::reg[inst.rs(t)].**
4. Set **Simulator::ID_EX.RegWrite , Simulator::ID_EX.MemRead Simulator::ID_EX.WriteDes** to corresponding values.
   EX: for **LW : the sequence is (true,true,inst.rt).**
5. If the inst involves **jump** or **branch** , it sets
   Simulator::Branch_taken ,Simulator::Branch_PC to corresponding value.
   It also checks if the Branch_PC is numovf.
   The Branch and Flush are done in main.
6. Set Simulator::ID_EX().inst as the inst now and set the
   Simulator::ID_EX.RegRs = DataRs;
   Simulator::ID_EX.RegRt = DataRt;

EXE():

1. Set **inst** to **Simulator::ID_EX.inst.**
2. Forwarding , again we use the result from previous cycle.

```
if (!Simulator::isBranch(inst)) {
    if (inst.fwdrs)
    {
        DataRs = Simulator::MEM_WB.Data;
        if (inst.fwdrs_EX_DM_from)
            DataRs = Simulator::WB_AFTER.ALU_result;
    }
    else
        DataRs = Simulator::ID_EX.RegRs;
    if (inst.fwdrt)
    {
        DataRt = Simulator::MEM_WB.Data;
        if (inst.fwdrt_EX_DM_from)
            DataRt = Simulator::WB_AFTER.ALU_result;
    }
    else
        DataRt = Simulator::ID_EX.RegRt;
}
```

inst.fwdrs(t)_EX_DM_from means if we need to forward from MEM_WB , if true then we pass the value in DataRs(t) = Simulator::WB_AFTER.ALU_result; if not we forward from EX_MEM (DataRs(t) = Simulator::MEM_WB.Data)

3. Then we do the execution steps(ALU calculate).
4. Then we pass the result and inst to **Simulator::EX_MEM.**

DM()

1. Set **inst** to **Simulator::EX_MEM.inst.**
2. This stage is to load/write the memory just as Project1.
3. Then we pass inst to **Simulator:MEM_WB.inst.**

WB()

1. Set **inst** to **Simulator::MEM_WB.inst.**
2. Write back to the register that we changed.
3. **Pass ALU_result and Data to Simulator::WB_AFTER.ALU_result. This is used to forward MEM/WB to EX.**

```
if (Simulator::MEM_WB.RegWrite) {
    if ((Simulator::MEM_WB.WriteDes == 0) && !Simulator::isBranch(inst) && !Simulator::notS(inst.name)) {
        Simulator::error_toggle[0] = true;      //Write 0
        return;
    }
    if (inst.type == 'R' || inst.type == 'J')
    {
        Simulator::reg[Simulator::MEM_WB.WriteDes] = Simulator::MEM_WB.ALU_result;
        Simulator::WB_AFTER.ALU_result = Simulator::MEM_WB.ALU_result;
    }
    else if (inst.type == 'I')
    {
        Simulator::reg[Simulator::MEM_WB.WriteDes] = Simulator::MEM_WB.Data;
        Simulator::WB_AFTER.ALU_result = Simulator::MEM_WB.Data;
    }
}
```

NextStage()

We call **checkfwdinID(Simulator::IF_ID.inst)** **checkfwdinEX(Simulator::ID_EX.inst)** **checkstall(Simulator::IF_ID.inst),** they actually play an important role in PrintCycle().

checkfwdinID()

1. This is only used when Branch.
2. If **Simulator::IF_ID.inst** is a Branching inst then we do the check otherwise we wont.
3. If EX_MEM **writes,not reading** and the **destination** = **Simulator::IF_ID.inst.rs** then **RsData = Simulator::EX_MEM.ALU_result;**
   Simulator::IF_ID.inst.fwdrs = true;
4. If not, **Simulator::IF_ID.inst.fwdrs = false;** and we also need to check **Simulator::MEM_WB** if it **writes** and **destination** = **Simulator::IF_ID.inst.rs** We need to set **RsData = Simulator::MEM_WB.ALU_result** depending on its type (This is to solve loading issue).
5. Same for **rt** expect inst can't be a jump inst.
6. Then we check RsData and RtData for branching condition, then we set **Simulator::Flush = (Branching) ? true : false;**
7. **Keep in mind this only check fwd in next stage and the Flush detect is used in PrintCycle , the actual branching and flushing is handled in main.**

checkfwdinEX()

1. This checks if we forward from MEM/WB or EX/MEM.
2. If **Simulator::ID_EX.inst isn't branching or MFLO(HI) or HALT**
   Then we check for forward in EX.
3. We check If EX_MEM **writes,not reading** and the **destination** = **Simulator::ID_EX.inst.rs** and **Simulator::ID_EX.inst.name** isn't **"SLL" "SRL" "SRA" "LUI" (˙.˙ their rs is don't care)** then **Simulator::ID_EX.inst.fwdrs = true; Simulator::ID_EX.inst.fwdrs_EX_DM_from = false;**

4. For rt it's pretty much the same except if **type is I TYPE ,** we only care about "SW" "SB" "SH" (˙.˙ **for other I type inst we don't depend on rt),**otherwise it works the same as rs.
5. Then we consider forward from MEM/WB , it works pretty much as same as the previous two checks except it checks **MEM_WB** instead of **EX_MEM** and **if the previous two checks have already forward a thing then it won't forward in order to prevent double harzard.**

checkstall()

1. This is to check whether it will stall on next stage.
2. If **Simulator::IF_ID.inst isn't MFLO(HI) ,** then we check if it's branch , if false then we check if **Simulator::ID_EX.MemRead and the destination is Simulator::IF_ID.inst.rs or rt.**
3. If **Simulator::IF_ID.inst.rs** = **Simulator::ID_EX.WriteDes and Simulator::IF_ID.inst.name** isn't **"SLL" "SRL" "SRA" "LUI" (˙.˙ their rs is don't care)** then **Simulator::Stall = true;** (looks familiar with previous one)
4. If **Simulator::IF_ID.inst.rt** = **Simulator::ID_EX.WriteDes** it's pretty much the same except if **type is I TYPE ,** we only care about "SW" "SB" "SH" (˙.˙ **for other I type inst we don't depend on rt),**otherwise it works the same as rs. (looks familiar with previous one)
5. If it's branch,then we check **Simulator::ID_EX.RegWrite if it writes and the destination is Simulator::IF_ID.inst.rs and Simulator::IF_ID.inst depends on it (not type J),then Stall = true;**
   For rt,pretty much the same as above except it considers bgtz,jal,jr,j(˙.˙ rt is don't care)
6. If **Simulator::EX_MEM.MemRead=true(load/save) and its write destination is Simulator::IF_ID.inst.rs(rt) , Stall must be TRUE.**

My testcase:

```
|   PC = 0x00000008
|   lw $1,0x0000($0)
|   lw $2,0x0008($0)
|   nop
|   nop
|   nop
|   mult $1,$2          //mult test
|   nop
|   add $3,$1,$2
|   sub $3,$3,$1        //0x00611822
|   multu $3,$3         //double harzard
|   mflo $3             //0x00001812
|   bne $3,$3,0x0001    //0x14630001
|   sub $3,$3,$1        //double harzard
|   sub $3,$3,$1
|   sub $3,$3,$1
|   add $0,$3,$3
|   beq $3,$3,0x0000
|   nop
|   nop
|   nop
|   sub $3,$3,$3        //0x00631822
|   nop
|   bne $3,$3,0x0000    //0x14630000
|   bne $3,$3,0x0000    //0x14630000
|#test addi largest ovf
|   lw $1,0x0000($0)
|   addi $1,$1,0x7FFF
|#test addi smallest ovf
|   lw $1,0x0008($0)
|   addi $1,$1,0x8000
|#test largest positive sub smallest negative overflow
|   lw $1,0x0000($0)
|   lw $2,0x0008($0)
|   sub $1,$1,$2
|#test largest positive sub overflow
|   lw $1,0x0000($0)
|   lw $2,0x0000($0)
|   sub $1,$1,$2
|#test smallest negative sub overflow
|   lw $1,0x0008($0)
|   lw $2,0x0008($0)
|   sub $1,$1,$2
|#test smallest negative add overflow
|   lw $1,0x0008($0)
```

```
|   #jump to boundary then jump back to execute next instruction
|
|   jr $4
|   halt
|   halt
|   #test every writeto0
|   add $0,$0,$0
|   addu $0,$0,$0
|   sub $0,$0,$0
|   and $0,$0,$0
|   or $0,$0,$0
|   xor $0,$0,$0
|   nor $0,$0,$0
|   nand $0,$0,$0
|   slt $0,$0,$0
|   srl $0,$0,0X0000
|   sra $0,$0,0X0000
|   mult $1,$1
|   multu $1,$1
|   mfhi $0
|   mult $1,$1
|   addi  $0,$1,0xffff
|   addiu  $0,$1,0xffff
|   lw   $0,0x0000($0)
|   lh    $0,0x0000($0)
|   lhu   $0,0x0000($0)
|   lb    $0,0x0000($0)
|   lbu   $0,0x0000($0)
|   lui   $0,0x0000
|   andi   $0,$1,0xffff
|   ori    $0,$1,0xffff
|   nori   $0,$1,0xffff
|   slti    $0,$1,0xffff
|   #test branch
|   addi $2,$0,0x0000
||<-beq  $2,$0,0x0001
||   halt
||<->bne  $1,$0,0x0001
||  halt
||->bgtz $1,0xFFFF
||<-bgtz $4,0X0009
||  halt
|||>lw $4,0x8($0)
||| lh $3,0x(-1)($4) #####test misalign,numovf,addovf END#####
||| halt
```

```
||| halt
||<->bgtz $4,0Xfffb
||| halt
||<->bne $1,$0,0xfffd
||| halt
||<->beq $2,$0,0xfffd
|------ j 0x200  #pc=0x3fc command = 0x08000080 jump to PC=0x200
```