İhsan Doğramacı Bilkent University

Computer Science

Computer Organization

CS 224

Design Report

Lab 5

Section 1

Ömer Oktay Gültekin

21901413

18/04/2022

b-c) 3 types of hazards would occur in this pipeline:

Structural:

1) Instruction Fetch after load-store instruction

   - Write Back and Instruction Fetch stages affected

   Sol:

   - Separating data and instruction memories (which we use)

   - Adding one more read port to the memory

Data:

1) Read After Write (RAW-Compute Use) with R type instructions

```
- add $t0, $t1, $t2
  sub $t3, $t0, $t1
  sub $t3, $t0, $t1
  add $t4, $t0, $t1 # solved by using posedge to write the
                    # new value and negedge of clock to
                    # read the register (Register File Access
                    # Hazard).
```

If precautions are not taken, hazard cause to use old $t0 value in calculations. Thus, Execute and Write Back stages are affected.

- for the first and second sub-operations, we need to take the $t0 values from different positions

Sol:

- Waiting for 2 clock cycles (2 stalling)

- Code Reordering (may be done by smart compilers)

- Forwarding (Bypassing) the new \$t0 value from the start of DM Stage to the second instruction Execute Stage or from WB Stage to the third instruction Execute Stage (which we use).

- In our datapath, forwarding will be implemented as AluOutM and ResultW and (ForwardAE or ForwardBE control signal depending on its position as operand).

2) Load-Use

- lw \$t0, 0(\$t1)

  add \$t2, \$t0, \$t3

- we can't use the earlier forwarding since the new value is determined not after execute stage but memory stage.

- hazard is the same as before; we may use the old value for \$t0.

- Write Back and Execute stages affected.

Sol: (From now on, just the solution we use will be shown)

- 1 stall and forwarding (We need to forward data from the memory stage to execute stage, which can not happen if we do not do one stall since we cannot do forwarding backward, as soon as we can do forwarding is if the execute stage happens one clock cycle after the memory stage).

- In our datapath, forwarding will use ResultW as operands of the execute stage, which is controlled by ForwardAE and ForwardBE signals; stalling will use StallF, StallD, and FlushE: the first two are for repeating the next instruction; whereas the last one for noop.

3) Load-Store

- lw $t0, 0($t1)

  sw $t0, 0($t2)

- Hazard is $t2 will not have the new value of $t0

- Write Back and Execute stages affected

Sol:

- A feedback system from WB stage to the Memory stage, but our datapath does not implement it. We need to mux ResultW and WriteDataM to determine WD of data memory. Thus, the solution is forwarding.

Control:

1) Branch

- Branch is normally determined in execute stage.

- It affects Instruction Fetch stage; therefore, we lose 3 clock cycles. Therefore, we need to flush 3 instructions, normally.

Sol:

- Determine branch as soon as register values are known in Instruction Decode stage.

- Normally, we need to wait for one clock cycle, which is one instruction flush (1 stall)

- However, we may need forwarding as well, if we use r-type instruction just before branch instruction which affects the operand registers of the branch

- In our datapath, PCSrcD, EqualD, and comparator were added in ID stage to implement early branch resolution; ForwardAD and ForwardBD were added for forwarding the new register values in case needed.

StallF, StallD, and FlushE will be high for one clock cycle when stalling.

d)

1) Forwarding Logic for ForwardAE:

```
if ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM)
        then ForwardAE = 10
else
   if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW)
        then ForwardAE = 01
   else      ForwardAE = 00
```

2) Forwarding Logic for ForwardBE:

```
if ((rtE != 0) AND (rtE == WriteRegM) AND RegWriteM)
       then ForwardBE = 10
else
   if ((rtE != 0) AND (rtE == WriteRegW) AND RegWriteW)
       then ForwardBE = 01
   else      ForwardBE = 00
```

3) Load Stalling Logic:

```
lwstall = ((rsD==rtE) OR (rtD==rtE)) AND MemtoRegE
StallF = StallD = FlushE = lwstall
```

Load Forwarding Logic uses 1 and 2 above.

4) Forwarding Logic for ForwardAD:

```
ForwardAD = (rsD !=0) AND (rsD == WriteRegM) AND
RegWriteM
```

5) Forwarding Logic for ForwardBD:

```
ForwardBD = (rtD !=0) AND (rtD == WriteRegM) AND
RegWriteM
```

6) Stalling Logic for Branch

```
branchstall = BranchD AND RegWriteE AND
                    (WriteRegE == rsD OR WriteRegE == rtD)
```

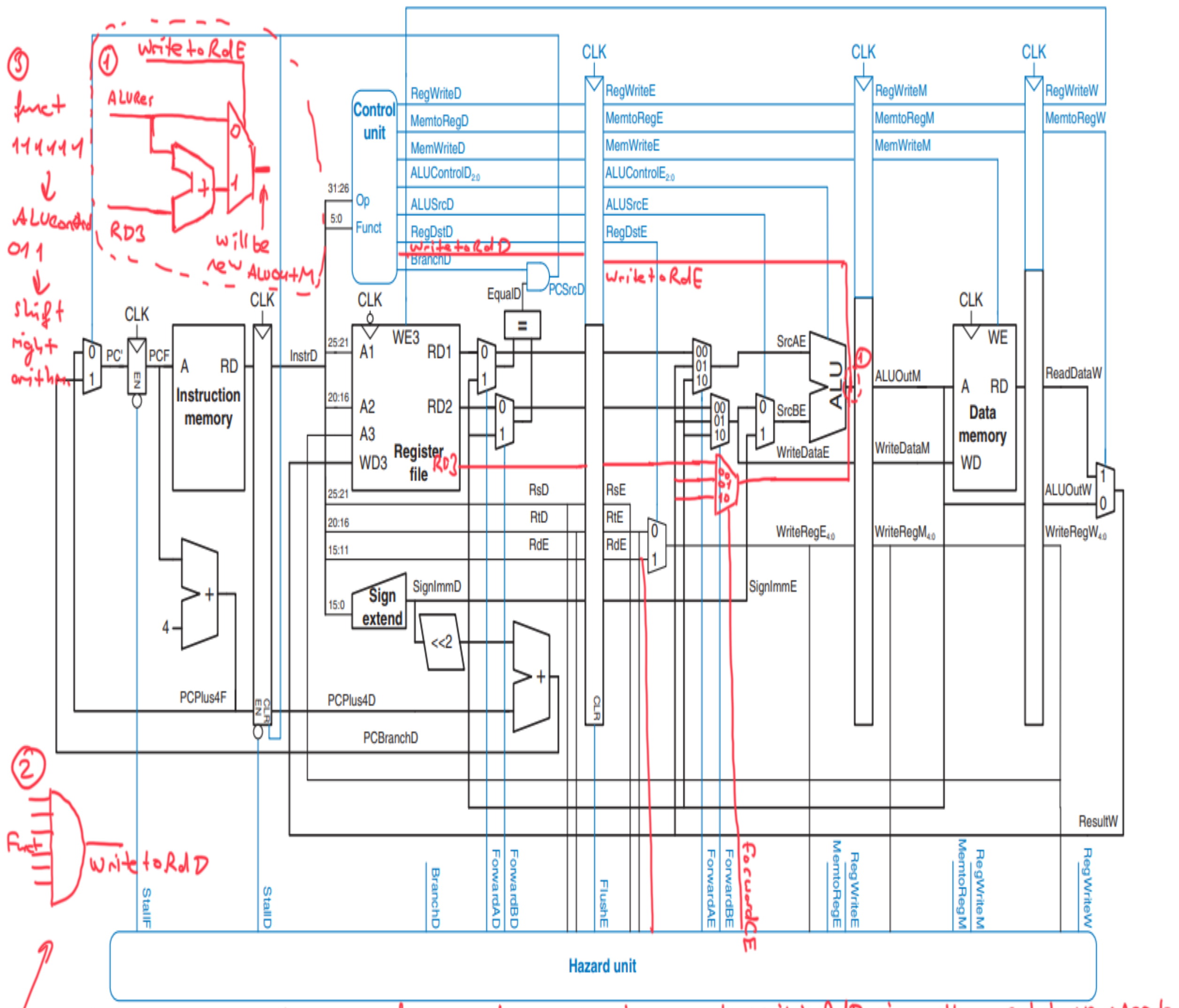
OR

BranchD AND MemtoRegM AND

(WriteRegM == rsD OR WriteRegM == rtD)

StallF = StallD = FlushE = (lwstall OR branchstall)

e)



since it is r-type, opcode is 0 and we need new way to generate write to RdD, since others not have operate on rd features. Assume fun code 111111. It not affect clock since it simultaneous with main decoder.

Figure 7.58 Pipelined processor with full hazard handling

sracc does not cause any new hazard since it is an r-type instruction, and we just add one new adder and mux to the system. According to the slowest step, new hardware shown with 1 in the figure can be added to the end of the execute stage or to the beginning of the memory stage, but the memory stage is usually slower. 2 and 3 show other implementation details, but what is important is it doesn't add new hazard just RAW (compute use) hazard. However, we use the rd register as an operand. Therefore, as with other operands, it may be changed in previous instructions, but our calculations may not see the new value in the rd register. Therefore, we need to add a new control signal to the hazard unit. We give RdE in the hazard unit to check if it is changed in previous instructions, aka later stages and if it is, we make ForwardCE 01 or 10 according to where the new value changed.

```
# This test program only tests sracc since others are already
# given in SampleTestsForHazards.txt.
addi $t0, $zero, 4
addi $t1, $zero, 2
addi $t2, $t0, 5
sracc $t2, $t0, $t1 # Hazard! $t2 needs new value as it is operand now. (
                    # from memory stage)
sracc $t2, $t0, $t1 # Hazard! $t2 has different values in memory and
                     # Write Back stage, hardware must take the latest
                     # (from memory stage)
```

```
addi $t0, $t0, 4

srcacc $t2, $t0, $t1 # Hazard! $t2 needs to take new value from WB
                     # stage and $t0 needs to be taken from Memory
                     # stage.

and $s0, $t0, $t1

add $s1, $t0, $s0

sracc $t2, $s0, $s1 # No new hazard, but earlier compute use ($s0 and
                    # $s1 should be taken from appropriate stages)

sracc $s0, $t0, $t1 # No hazard
```