CSE471—Computer Design and Organization

Prof. Steven Burns *Autumn 1994*

Lecture Notes: Ninteenth Day

Hazard Detection Unit

- If we want to maintain a sequential semantics for the machine, we must resolve the data hazards is some manner.
- It is safe to stall in any situation, but fewer stalls means higher performance.
- Let's try to determine exactly the conditions where a stall is necessary.
- (We won't look at the contents of the registers.)

Data Hazard Conditions

```
   ID/EX.RegWrite &
   ((ID/EX.RegDst=0 & ID/EX.WriteRegRt=IF/ID.ReadReg1) |
    (ID/EX.RegDst=1 & ID/EX.WriteRegRd=IF/ID.ReadReg1) |
    (ID/EX.RegDst=0 & ID/EX.WriteRegRt=IF/ID.ReadReg2) |
    (ID/EX.RegDst=1 & ID/EX.WriteRegRd=IF/ID.ReadReg2))
|  EX/MEM.RegWrite &
   ((EX/MEM.WriteReg=IF/ID.ReadReg1) |
    (EX/MEM.WriteReg=IF/ID.ReadReg2))
|  MEM/WB.RegWrite &
   ((MEM/WB.WriteReg=IF/ID.ReadReg1) |
    (MEM/WB.WriteReg=IF/ID.ReadReg2))
```

More Refined Hazard Conditions

- For `lw`, the *rt* field is the destination register. A false stall occurs in the case:

```
    add  $8, $1, $2
    lw   $8, 1200($5)
```

- How do we fix this?

```
    ID/EX.RegWrite &
    ((ID/EX.RegDst=0 & ID/EX.WriteRegRt=IF/ID.ReadReg1) |
     (ID/EX.RegDst=1 & ID/EX.WriteRegRd=IF/ID.ReadReg1) |
     (ID/EX.RegDst=0 & ID/EX.WriteRegRt=IF/ID.ReadReg2 &
                       IF/ID.OpCode!='lw'              ) |
     (ID/EX.RegDst=1 & ID/EX.WriteRegRd=IF/ID.ReadReg2 &
                       IF/ID.OpCode!='lw'              ))
| ...
```

Register Zero

- There is no need to stall on a `WriteReg` index of zero because the instruction will not change the contents of the register file.
- We can do this by not asserting the `RegWrite` when this control signal is generated in the ID stage.

Forwarding

- We can use data values from later stages of the pipeline instead of stalling.
- Consider only the case of a sequence of R-type instructions.
- We introduce multiplexors with more inputs into the EX stage.
- The extra inputs come from the ALUResult stored in the EX/MEM pipeline register and the ReadData or delayed ALUResult stored in the MEM/WB pipeline register.

Forwarding Control

- If there is a data hazard between `EX/MEM.WriteReg` and `ID/EX.ReadReg1` then we want to use the stored ALUResult as the A input to the ALU.
- If there is a data hazard between `EX/MEM.WriteReg` and `ID/EX.ReadReg2` then we want to use the stored ALUResult as the B input to the ALU.

Forwarding Control (cont.)

- If there is a data hazard between `MEM/WB.WriteReg` and `ID/EX.ReadReg1` (and there is not a `EX/MEM.WR` to `ID/EX.RR1` hazard), then we want to use the stored data in the MEM/WB pipeline register as the A input to the ALU.
- If there is a data hazard between `MEM/WB.WriteReg` and `ID/EX.ReadReg2` (and there is not a `EX/MEM.WR` to `ID/EX.RR2` hazard), then we want to use the stored data in the MEM/WB pipeline register as the B input to the ALU.
- Why does the EX/MEM register have higher priority than the MEM/WB register?

Other Complications

- Remember, every use of register zero must supply a zero value.
- We don't want to forward if the register index is zero.
- We must change the forwarding control to achieve this.
- How do we do this?

Forwarding with Loads and Stores

- Forwarding doesn't solve all data hazards.
- An R-type register use after an `lw` cannot be corrected by forwarding.
- We can handle this by reintroducing a Hazard Detection Unit or leaving out the hardware and making the compiler deal with it (delayed loads).
- What is the condition for a stall?

Other Complexities

- We must put the signed-immediate input to the ALU back into the datapath and control the multiplexor correctly.
- We can use the existing forwarding to handle data hazard with `sw`.
- We can introduce more forwarding hardware to improve store after load sequences (memory to memory copying).

Branch Hazards

- Why do we need to stall?
- Which branch do we take without a stall?
- How do we flush the pipeline (squash the instruction)?

Branch Prediction

- We can proceed with execution without introducing as many bubbles if we guess correctly whether or not the branch should be taken.
- This can be done statically or dynamically:
  - A simple static strategy is to predict as taken all backward branches and predict as not taken all forward branches.
  - Dynamic schemes are also possible. A good way is to use two bits of state for each instruction in the I-cache (or a special small cache called a branch-prediction buffer.) You use the two bits so that two mispredictions in row cause the prediction to change.

Branch Target Buffer

- To really speed things up, we need to know the predicted branch target (address to the next instruction) before the instruction is decoded.
- This may seem impossible, but caching techniques work well.
- We access into this BTB every instruction. If there is a hit, we know the instruction is a branch and we should set the next PC accordingly.

Reducing the Number of Branch Delays

- With the current hardware, there is a three cycle penalty.
- We can move the branch decision hardware to earlier stages in the pipeline by rearranging and duplicating circuitry.
- Pushing the target mux from the MEM stage to the EX stage saves one cycle. (It may extend the cycle period.)
- We can push the branch target calculation and comparison into the ID stage if we can design a fast enough equality checker (it must run in the same cycle as the register access.)

Exceptions

- These make pipelining difficult.
- *Precise* and *imprecise* exceptions.
- Exceptions are detected in different stages based on their type:
  - IF — Instruction Memory Page Fault
  - ID — Illegal Opcode
  - EX — Overflow
  - MEM — Data Memory Page Fault

General Exception Strategies

- Post each exception in a status register that is carried along with each instruction as it goes down the pipeline. In the WB stage, check if an exception has occurred. This ensures exceptions are seen in order.
- Alternatively, we can handle the exception as soon as it occurs. We must stop immediately without completing any instructions that have yet to change state. This could be $i-2$, $i-1$, i, and $i+1$ if the exception is an IMEM fault. We must restart this case with $i-2$. (Why?)

Delayed Branches

- We can always execute some number of instructions following a branch.
- Delayed branches introduce special problems for handling exceptions.
- Think about the case of an exception caused by the instruction in the branch delay slot.
- We need to save and restore a number of PCs (one more than the number of branch delay slots.)

Advanced Pipelining

- How do we control multiple execution units with differing latencies?
- Structural Hazards — we might need to stall because all the appropriate execution units are busy.
- New data hazards: WAR and WAW
- Out of order completion, inorder issue might have WAW hazards.
- Out of order issue might have WAR hazards as well.