Pipeline Review

Review

- Covered in EECS2021 (was CSE2021)
- Just a reminder of pipeline and hazards
- If you need more details, review 2021 materials

The basic MIPS Processor

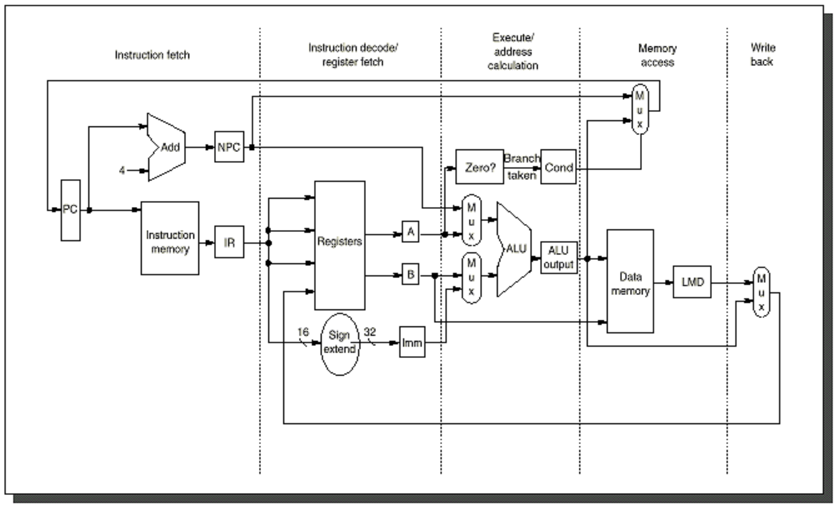


FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles.

Pipeline

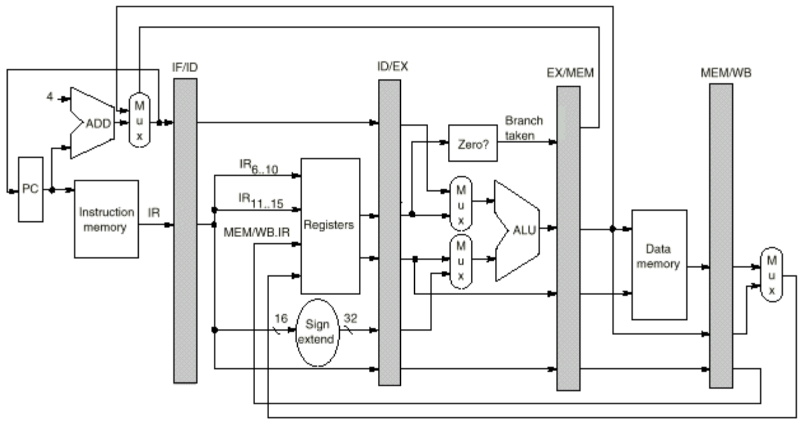


FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages.

Performance of pipelining

- Pipelining does not reduce the time to execute the instruction (it actually increases it). It increases the throughput.
- We can not skip stages anymore, and the pipeline cycle = time for longest cycle longest cycle
- Example: a machine with 10-ns cycle, it takes 4 cycles for ALU and branches, and 5 for memory (40,20,40%), what is the effect of the pipelining
- Without execution time = 10*(0.6*4+0.4*5)= 44 ns
- With pipelining 10+1 (overhead)=11
- Speedup = 4

Performance of Pipelining

- The length of a machine clock cycle is determined by the time required for the slowest pipe stage.
- An important pipeline design consideration is to balance the length of each pipeline stage.
- If all stages are perfectly balanced, then the time per instruction on a pipelined machine (assuming ideal conditions with no stalls):

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

- Under these ideal conditions:
  - Speedup from pipelining equals the number of pipeline stages: n,
  - One instruction is completed every cycle, CPI = 1 .

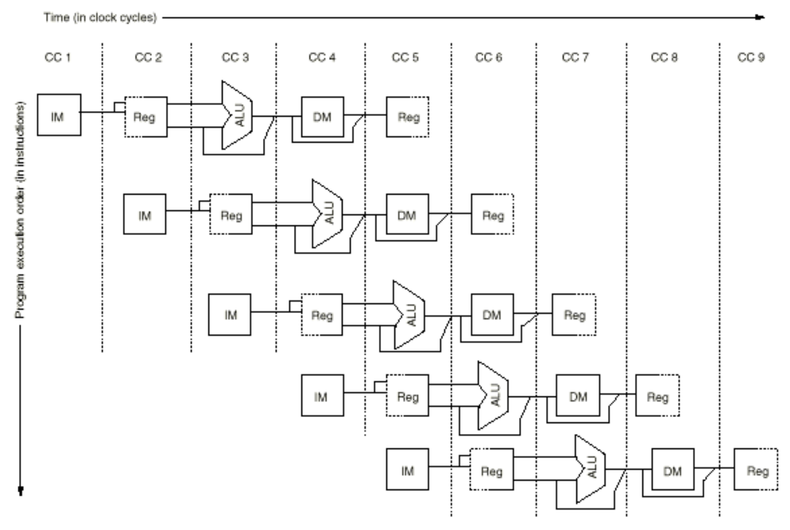


FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time.

Hazards

- There are situation called *hazards* that prevents the continuous flow of instructions in the pipe
- Structural hazards: resource conflicts
- Data hazards: instruction depends on the results from a previous instruction that is not ready yet.
- Control hazards: branches (we don't know the address of the next instruction)

Performance

- Hazards in pipelines may make it necessary to stall the pipeline by one or more cycles and thus degrading performance from the ideal CPI of 1.

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction

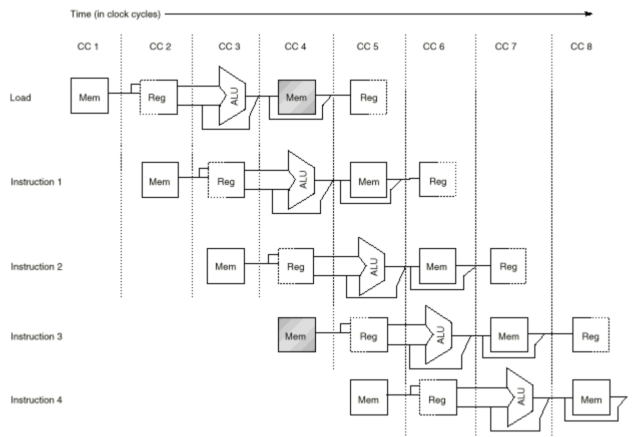
- When all instructions take the same number of cycles and is equal to the number of pipeline stages then:

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)

Structural Hazards

- When we pipeline a machine, the overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:
- One example is when we have a single memory for both instructions and data.

Structural Hazards



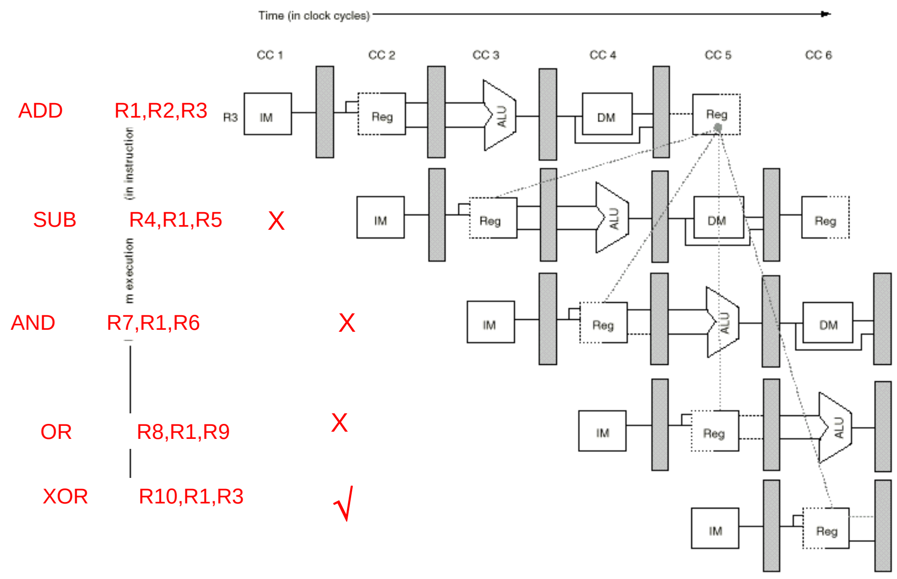
3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs.

Data Hazards

- Pipelining changes the relative timing of the instructions by overlapping their execution.
- If the timing of read/write accesses to the operands is changed, that might result in incorrect execution
- Example:

```
ADD   R1, R2, R3
SUB   R4, R1, R5
AND   R7, R1, R6
OR    R8, R1, R9
XOR   R10,R1,R11
```

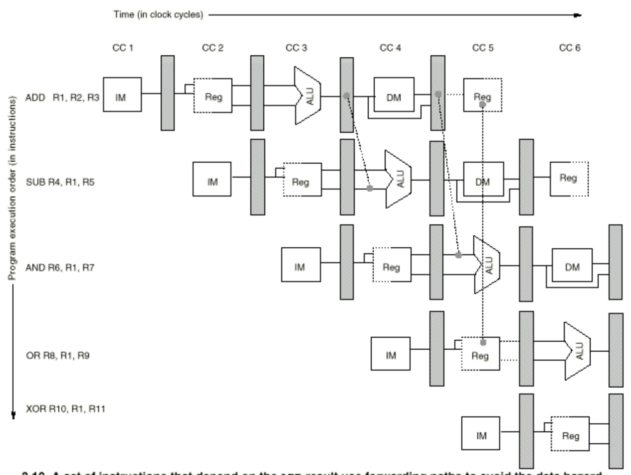
  – All the instructions after ADD use the result of the ADD instruction (ready in WB stage)
  – Without proper precautions, SUB will read the old value in R1
  – SUB, AND, and OR instructions need to be stalled for correct execution.



Forwarding

- It is one thing to read operand before it is written, and between requesting an operand before it is produced.
- Results of ADD is written in CC 5, read by SUB in CC 3
- BUT, the results of ADD is produced in CC 3, requested by SUB in CC 4
- We can use Forwarding
  1. The ALU result from EX/MEM register is always fed back to the input of the ALU
  2. If the forwarding hardware detects that if the previous ALU op writes to a register that is the source of the current ALU op, use a MUX to choose the fed back value instead of source register
  – We need to forward results not only from previous instruction, but from an instruction that started 3 cycles earlier

Forwarding



3.10 A set of instructions that depend on the ADD result use forwarding paths to avoid the data hazard.

Forwarding

- Consider the following sequence

```
ADD    R1, R2, R3
LW     R4, 0(R1)
SW     12(R1), R4
```

To prevent stalls, we need to forward the values in R1 and R4 from the pipeline registers to the inputs of the ALU and data memory.

- We may require a forwarding path from any pipeline register to the input of any functional unit

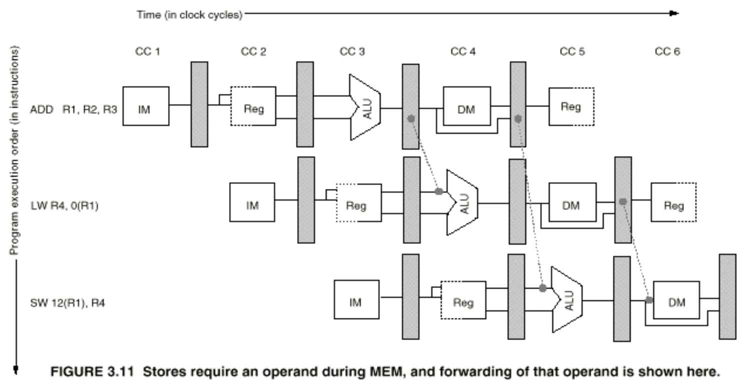
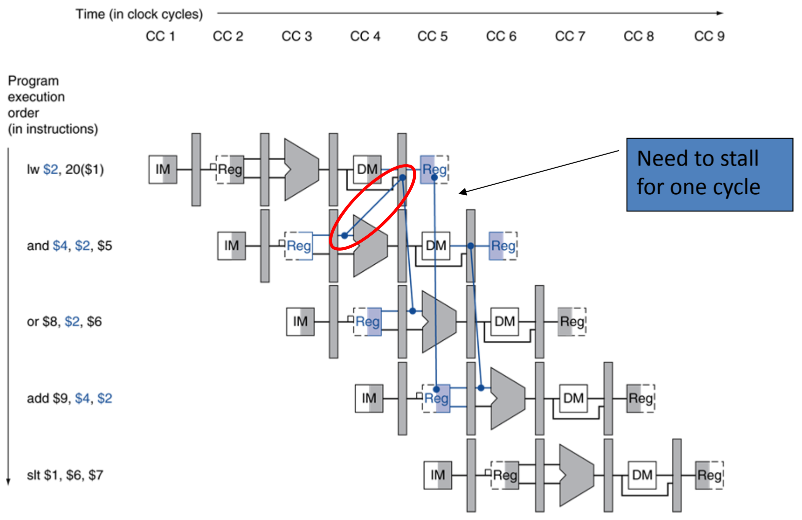
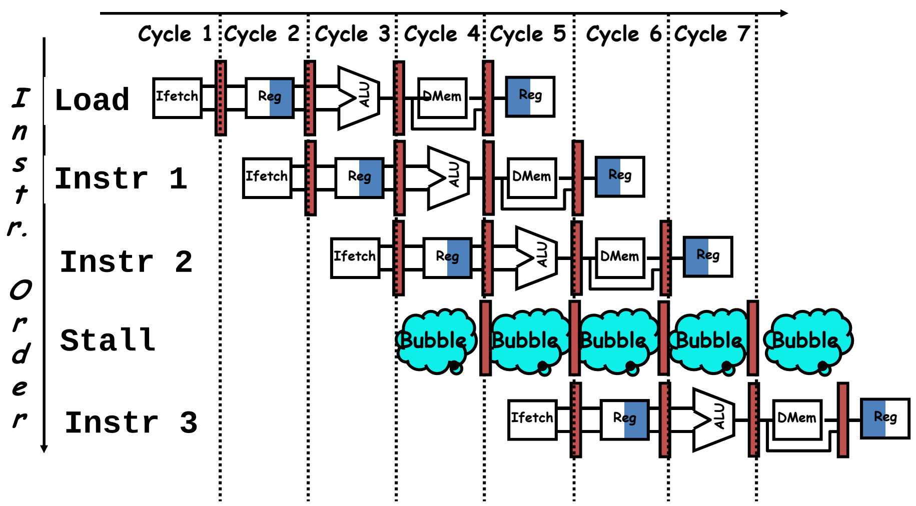


FIGURE 3.11 Stores require an operand during MEM, and forwarding of that operand is shown here.

Load-Use Data Hazard



Chapter 4 — The Processor — 18



Compiler Scheduling for Data Hazards

- Compiler may try to rearrange the code in order to avoid stalls.
- For example, avoid generating a code where there is a load followed by the immediate use of the loaded value
- Example, consider the code segment

```
A=b+c
D=e-f
Here is two way of generating
  code
```

EX

```
LW    Rb,b                    LW    Rb,b
LW    Rc,c                    LW    Rc,c
ADD   Ra,Rb,Rc     Stall      LW    Rf,f
SW    a,Ra                    ADD   Ra,Rb,Rc
LW    Rf,f                    LW    Re,e
LW    Re,e                    SW    a,Ra
SUB   Rd,Re,Rf                SUB   Rd,Re,Rf
SW    d,Rd         Stall      SW    d,Rd
```

Control Hazards

Branch instruction	**IF**	**ID**	**EX**	**MEM**	**WB**					
Branch successor		**IF**	**stall**	**stall**	**IF**	**ID**	**EX**	**MEM**	**WB**	
Branch successor + 1						**IF**	**ID**	**EX**	**MEM**	**WB**
Branch successor + 2							**IF**	**ID**	**EX**	**MEM**
Branch successor + 3								**IF**	**ID**	**EX**
Branch successor + 4									**IF**	**ID**
Branch successor + 5										**IF**

° In order to reduce the branch penalty, we must do two things

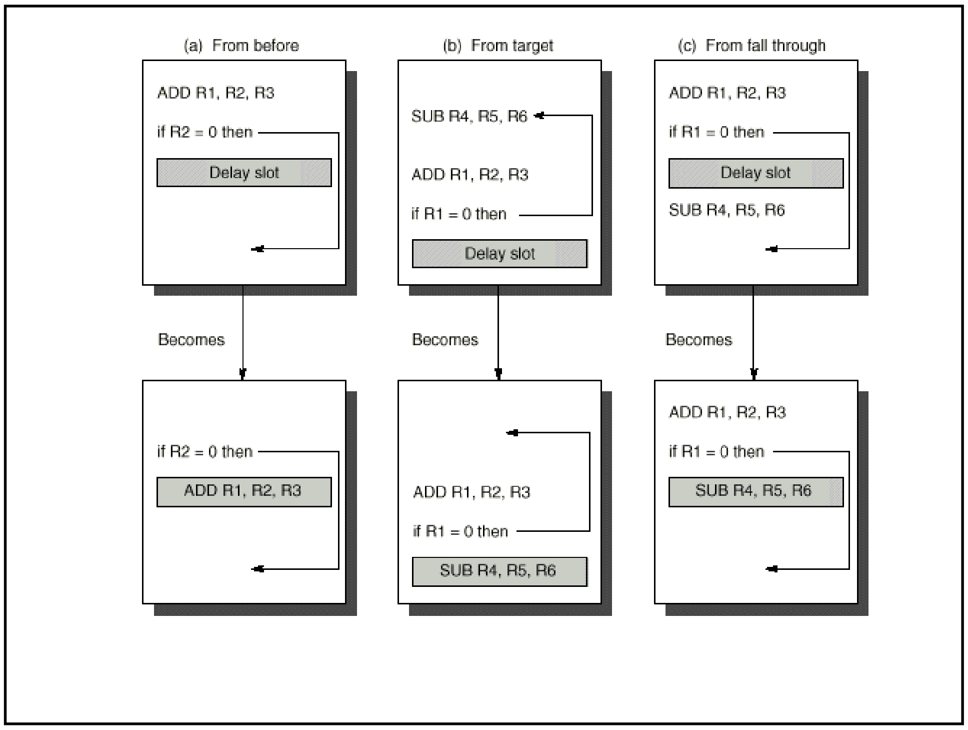
- Find out if the branch is taken or not as soon as possible
- Compute the taken PC ASAP

Compile Time Solutions

- Compiler may decide to predicts during compilation if the branch is taken or not
- Easiest way is to *freeze* or *flush* the pipe (simple)
- Predict branch is not taken, proceed as usual but be careful either not to change the state of the machine (or *back out* if you do) until you know if the prediction was correct or not
- Predict taken Useful only if we know the target address before we know the result of the comparison (condition)

Scheduling the Branch Delay Slot

- In this case, the job of the compiler is to make the successor instruction valid and useful (Fig. 3.28)
- In (a) the scheduled instruction should be done anyway, no harm at all
- IN (b) and (c) the use of R1 in the branch condition prevents moving the instruction to after the branch
- In both cases, it must be O.K. to execute the SUB instruction when the branch will go the opposite direction
- In (b) useful when the branch is taken with a high probability, the reverse in (c) .


(a) From before
ADD R1, R2, R3
if R2 = 0 then
Delay slot
Becomes
if R2 = 0 then
ADD R1, R2, R3
(b) From target
SUB R4, R5, R6
ADD R1, R2, R3
if R1 = 0 then
Delay slot
Becomes
ADD R1, R2, R3
if R1 = 0 then
SUB R4, R5, R6
(c) From fall through
ADD R1, R2, R3
if R1 = 0 then
Delay slot
SUB R4, R5, R6
Becomes
ADD R1, R2, R3
if R1 = 0 then
SUB R4, R5, R6