Basic Pipelining

B.Ramamurthy

CS506

Introduction

- In a typical system speedup is achieved through parallelism at all levels: Multi-user, multi-tasking, multi-processing, multi-programming, multi-threading, compiler optimizations.
- **Pipelining** : is a technique for overlapping operations during execution. Today this is a key feature that makes fast CPUs.
- Different types of pipeline: instruction pipeline, operation pipeline, multi-issue pipelines.

Topics to be discussed

- **What is a pipeline?**
- **A simple implementation of DLX**
- **Basic pipeline of DLX**
- **Performance issues**
- **Structural hazards**
- **Data hazards**
- **Control hazards**
- **Implementation issues**
- **Handling multi-cycle operations**
- **Instruction set design and pipelining**
- **Example: MIPS pipeline**
- **Summary**

What is a pipeline?

- Pipeline is like an automobile assembly line.
- A pipeline has many steps or **stage**s or **segment**s.
- Each stage carries out a different part of instruction or operation.
- The stages are connected to form a pipe.
- An inst or operation enters through one end and progresses thru' the stages and exit thru' the other end.
- Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream.

Pipeline characteristics

- **Throughput:** Number of items (cars, instructions, operations) that exit the pipeline per unit time. Ex: 1 inst / clock cycle, 10 cars/ hour, 10 fp operations /cycle.
- **Stage time**: The pipeline designer's goal is to balance the length of each pipeline stage. Balanced pipeline. In general,

stage time = Time per instruction on non-pipelined machine / number of stages.

In many instances, stage time = max (times for all stages).

- **CPI** : Pipeline yields a reduction in cycles per instruction. CPI approx = stage time.

Implementation of DLX's ISA

- DLX instruction can be implemented in at most five cycles:
- **Instruction fetch** (IF):

  IR <== Mem[PC]

  NPC <== PC + 4

- **Instruction decode** (ID)

A <== Regs[IR 6..10]

B <== Regs[IR 11..15]

Imm <== IR 16..31 with sign

Implementation of DLX's ISA

- **Execution/Effective address (EX)**: Four alternatives:
- Mem. Reference :

ALUoutput <== A + Imm;

- Register-Register ALU inst:

ALUoutput <== A op B;

- Register-Immediate :

ALUoutput <== A op Imm;

- Branch:

ALUoutput <== NPC + Imm; Cond <== (A op 0)

Implementation … (contd.)

- **Memory access /branch completion (MEM):**
- Memory access:

LMD <== Mem[ALUoutput] or

Mem[ALUoutput] <== B

- Branch:

if (cond) PC <== ALUoutput else PC<== NPC

Implementation … (contd.)

- **Write Back cycle (WB):**
- Register-register ALU inst:

Regs[IR $_{16..20}$] <== ALUoutput

- Register-Immediate ALU inst. :

Regs[IR $_{11..15}$] <== ALUoutput

- Load Instruction:

Regs[IR $_{11..15}$] <== LMD

Hardware diagram

- Fig. 3.1 Study and understand thoroughly the various components.

Timing and control (The missing links)

- What's missing in the RTL description of DLX given above is the timing and control information:
- For example: (Add R1,R2,R3)

Add.t0: IR <== Mem[PC], NPC <== PC + 4

Add.t1: A <== Regs[IR $_{6..10}$], B <== Regs[IR $_{11..15}$]

Add.t2: ALUoutput <== A op B;

Add.t3: do nothing (idling)

Add.t4: Regs[IR $_{16..20}$] <== ALUoutput

Timing and control - Branch

Br.t0 : IR <== Mem[PC], NPC <== PC + 4

Br.t1 : A <== Regs[IR $_{6..10}$], Imm <== IR $_{16..31}$ with sign

Br.t2 : ALUoutput <== NPC + Imm; Cond <== (A op 0)

Br.t3 : if (cond) PC <== ALUoutput else PC<== NPC

Basic pipeline of DLX

- Five stages: IF, ID, EX, MEM, WB
- On each clock cycle an instruction is fetched and begins its five cycle execution.
- Performance is up to five times that of a machine that is non-pipelined.
- What do we need in the implementation of the data path to support pipelining?

Pipelining the DLX datapath

1) Separate instruction and data caches eliminating a conflict that would arise between instruction fetch and data memory access. This is shown in the data path we studied earlier. This design avoids resource conflict.
2) We need to avoid register file access conflict: it is accessed once during ID and another time during WB stage.
3) Update PC every cycle. So mux from memory access stage is to be moved to IF stage.
4) All operations in one stage should complete within a clock cycle.
5) Values passed from one stage to the next must be placed in buffers/latches (I use buffers instead of registers to avoid confusion with regular registers).

Pipelining the DLX datapath

- How do arrive at the above list of requirements? Examine what happens in each pipeline stage depending on the instruction type. Make a list of all the possibilities.
- RTL statements of the events on every stage of the DLX pipeline is given in Fig.3.5.
- To control this pipeline, we only need to determine how to set the control on the four multiplexers (mux)
  - The first one inputs to PC. Lets call it MUX1.
  - The next two the input to ALU: MUX2, MUX3
  - The fourth one input to register file: MUX4

Controlling the pipeline

- Lets refer to interface between stages IF and ID, **IF/ID** and the other interfaces between stages **ID/EX**, **EX/MEM**, and **MEM/WB**.
- MUX1: is controlled by the condition checking done at EX/MEM. Based on this condition **EX/MEM.cond**, the MUX1 selects the current PC or the branch target as the instruction address.

Controlling the pipeline (contd.)

- MUX2 and MUX3 are controlled by the type of instruction. MUX2 is set by whether the instruction is a branch or not. MUX3 is set by whether the instruction is Register-Register ALU operation or any other operation.
- MUX4: is controlled by whether the instruction in the WB stage is a load or an ALU operation.
- In addition there is one MUX which chooses the correct portion of the IR in the MEM/WB buffer to specify the register destination field.

Pipeline performance - Example1

- General: 40% ALU, 20% branch, 40% memory.
- Design1: Non- pipelined. 10ns clock cycles. ALU operations and branches take 4 cycles, memory operations take 5 cycles..In other words, ALU operations and branches take 4*10 = 40 ns time.
- Design 2: Pipelined. Clock skew and setup add 1 ns overhead to clock cycle.
- What is the speedup?

Pipeline performance (contd.)

- Design1:

Average instruction execution time = clock cycletime * CPI

= 10ns * (4 *0.4 + 4 *0.2+ 5*0.4) = 10 *(1.6+0.8+2.0)

= 44ns

- Design 2:

Average instruction time st steady state is clock cycle time:

= 10ns + 1ns (for setup and clock skew) = 11ns

- Speed up = 44/11 = 4

Pipeline performance - Example2

- Assume times for each functional unit of a pipeline to be: 10ns, 8ns, 10ns, 10ns and 7ns. Overhead 1ns per stage. Compute the speed of the data path.
- Pipelined: Stage time = MAX(10,8,10,10,10,7) + overhead

= 10 + 1 = 11ns.

This is the average instruction execution time at steady state.

- Non-pipelined: 10+8+10+10+7 = 45ns
- Speedup = 45/11= 4.1 times

Pipeline hazards

- Hazards reduce the performance from the ideal speedup gained by pipelines:
- **Structural hazard**: Resource conflict. Hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
- **Data hazard**: When an instruction depends on the results of the previous instruction.
- **Control hazard**: Due to branches and other instructions that affect the PC.

Pipeline stalls

- A stall is the delay in cycles caused due to any of the hazards mentioned above.
- Speedup :

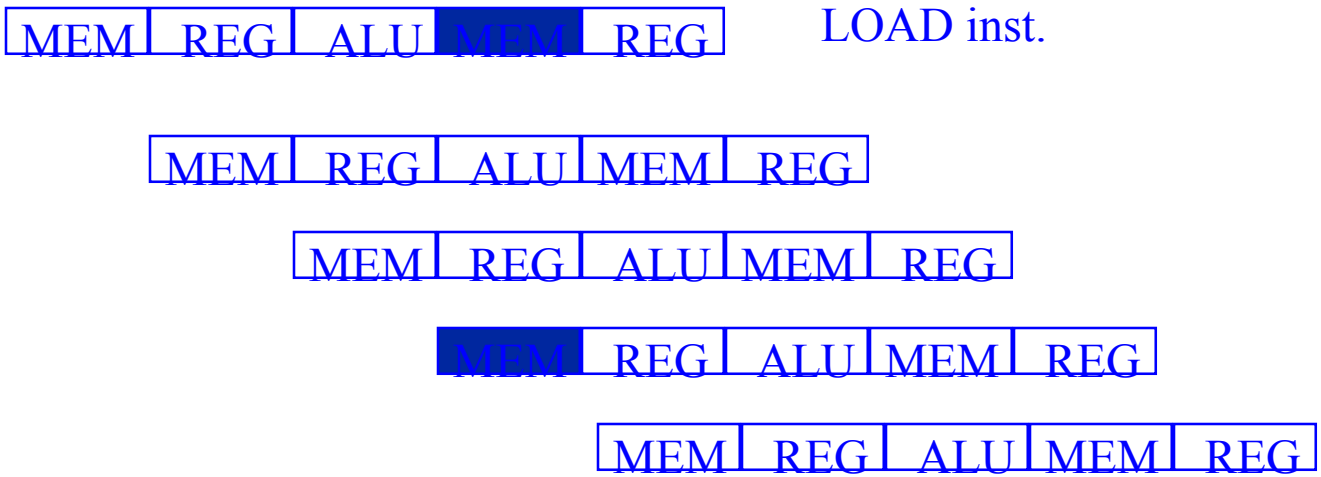
1/(1+pipeline stall per instruction)* Number of stages

- So what is the speed up for an ideal pipeline with no stalls?
- Number of cycles needed to initially fill up the pipeline could be included in computation of average stall per instruction

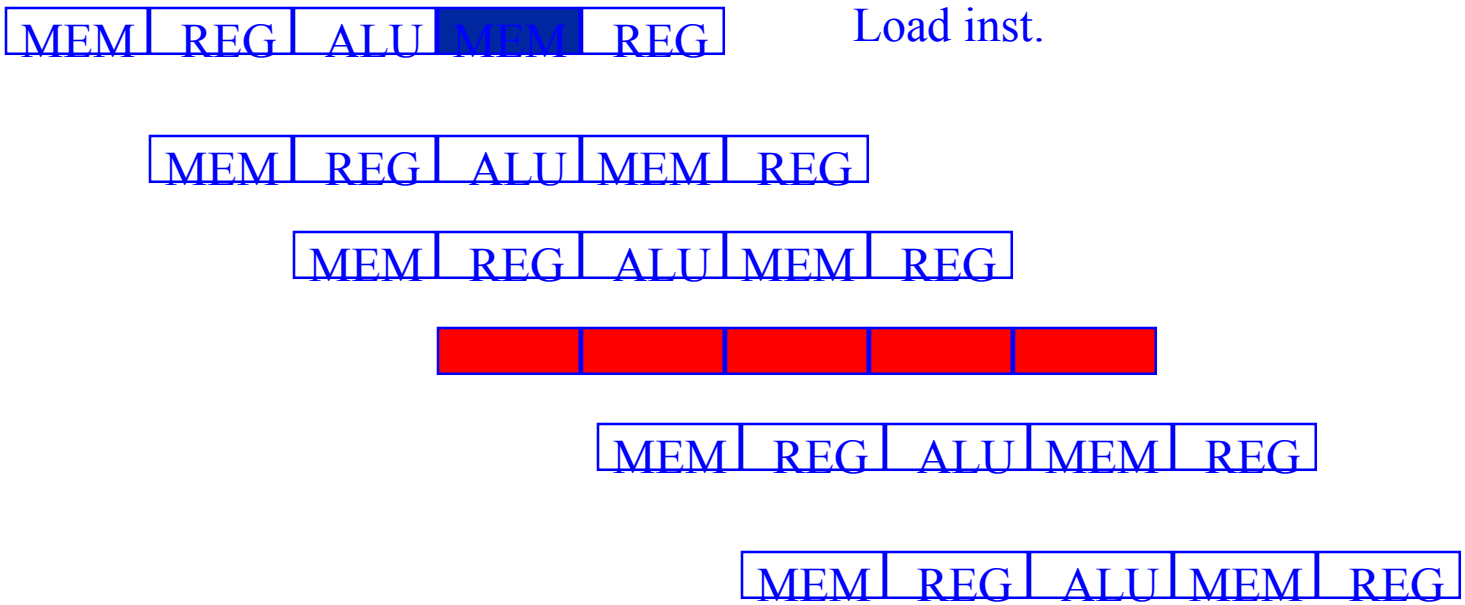
Structural hazards

- When more than one instruction in the pipeline needs to access a resource, the datapath is said to have a **structural hazard**.
- Examples of resources: register file, memory, ALU.
- Solution: Stall the pipeline for one clock cycle when the conflict is detected. This results in a pipeline bubble.
- See Fig.3.6, 3.7 that illustrate the memory access conflict and how it is resolved by stalling an instruction. Problem: one memory port.

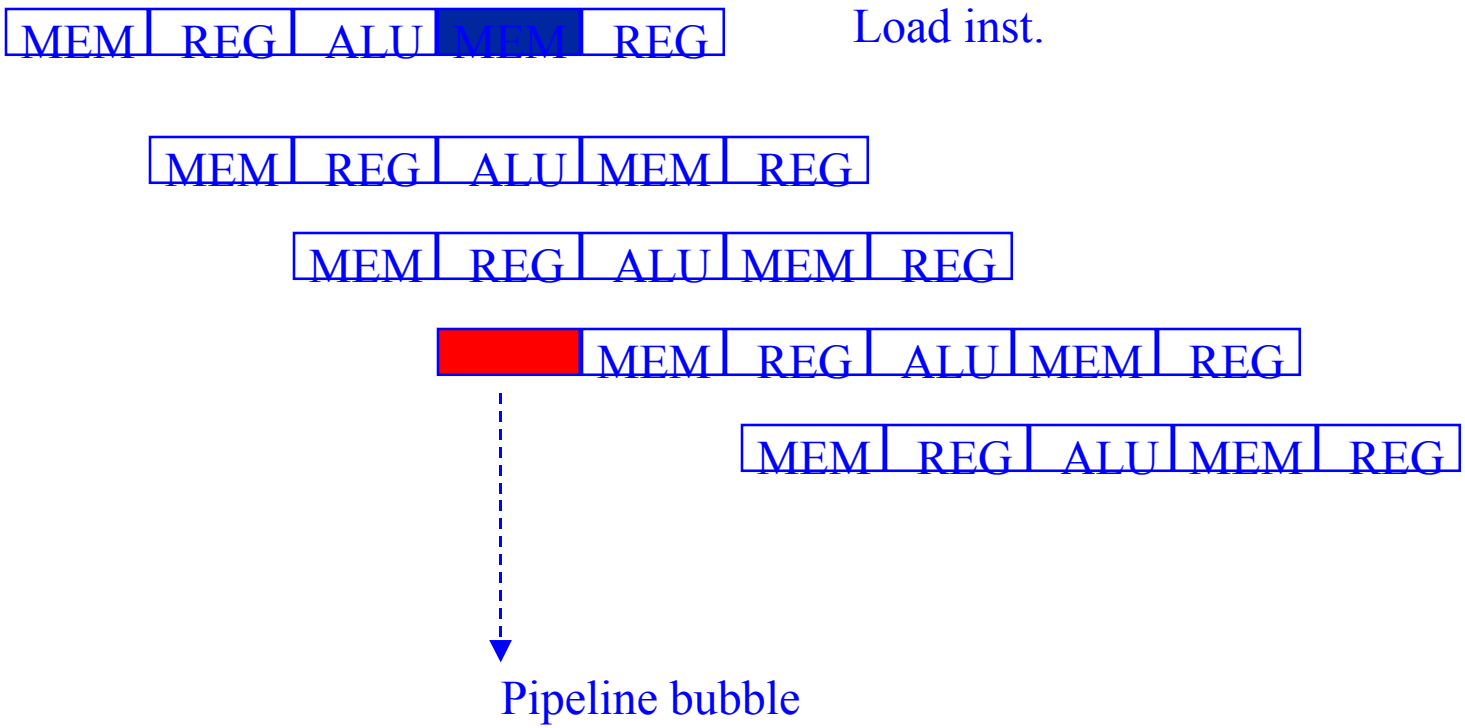
Structural Hazard and Stalls - Conflict



Structural Hazard and Stalls - Solution



Structural Hazard and Stalls - Bubble



Structural hazard: Example3

- Machine with load hazard: Data references constitute 40% of the mix. Ideal CPI is 1. Clock rate is 1.05 of the machine without hazard. Which machine is faster, the one with hazard (machine A) or without the hazard (machine B)? Prove.
- Solution: Hazard affects 40% of the B's inst.
- Average inst time for **machine A**: CPI * clock cycle time = 1 * x = **1.0x**

Example 3 - page 144 (contd.)

- Average inst time for machine B:

1) CPI has been extended.

= 40% of the times 1 more cycle

2) Clock rate is faster: 1.05 times: less than machine A. By how much?

Avg instruction time for **machine B**: (1 + 40/100*1) * (clock cycle time /1.05)

= 1.4 * x/1..05 = **1.3x**

Proved that A is faster.

Data hazard

- Consider the inst sequence:

ADD R1,R2,R3 ; result is in R1

SUB R4,R5,R1

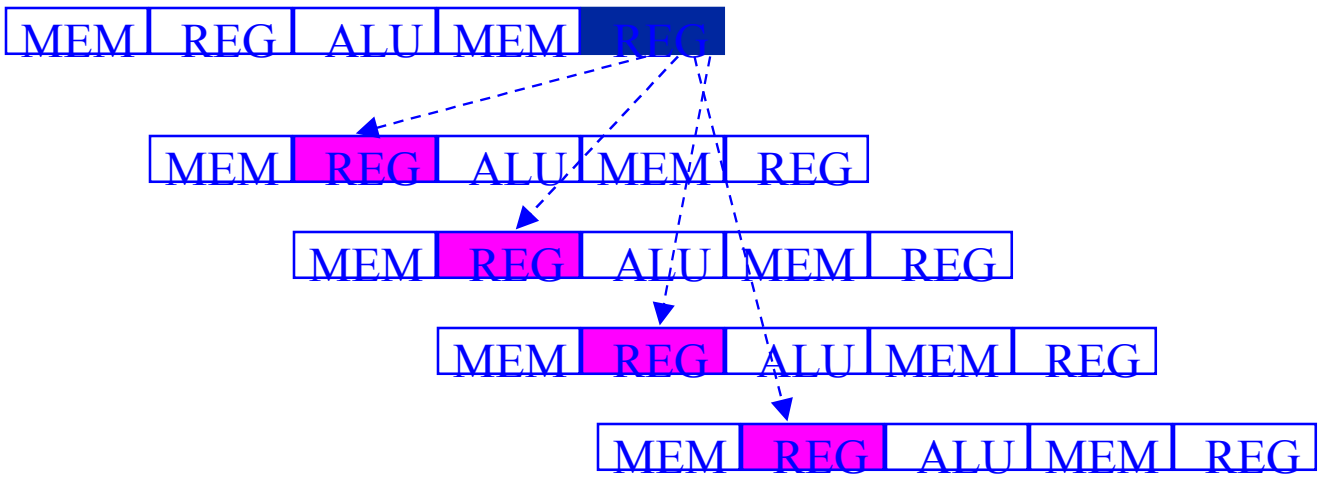
AND R6,R1,R7

OR R8,R1,R9

XOR R10,R1,R11

All instructions use R1 after the first inst.

Data hazard - Time-stage diagram



Data hazard - solution

- Usually solved by data or register forwarding (bypassing or short-circuiting).
- How? The data selected is not really used in ID but in the next stage: ALU.
- Forwarding works as follows:
- ALU result from EX/MEM buffer is always fed back to ALU input latches.
- If the forwarding hardware detects that its source operand has a new value, the logic selects the newer result than the value read from the register file.

Data hazard - solution (contd.)

- The results need to forwarded not only from the immediately previous instruction but also from any instruction that started upto three cycles before.
- The result from EX/MEM (1 cycle before) and MEM/WB (2 cycles before) are forwarded to the both ALU inputs.
- Writing into the register file is done in the first half of the cycle and read is done in the second half.(3 cycles before)

Data hazard classification

- **RAW - Read After Write. Most common: solved by data forwarding.**
- **WAW - Write After Write : Inst i (load) before inst j (add). Both write to same register. But inst i does it before inst j. DLX avoids this by waiting for WB to write to registers. So no WAW hazard in DLX.**
- **WAR - Write after Read: inst j tries to write a destination before it is read by I, so I incorrectly gets its value. This cannot happen in DLX since all inst read early (ID) but write late (WB). But WAW happens in complex instruction sets that have auto-increment mode and require operands to be read late cycle experience WAW.**

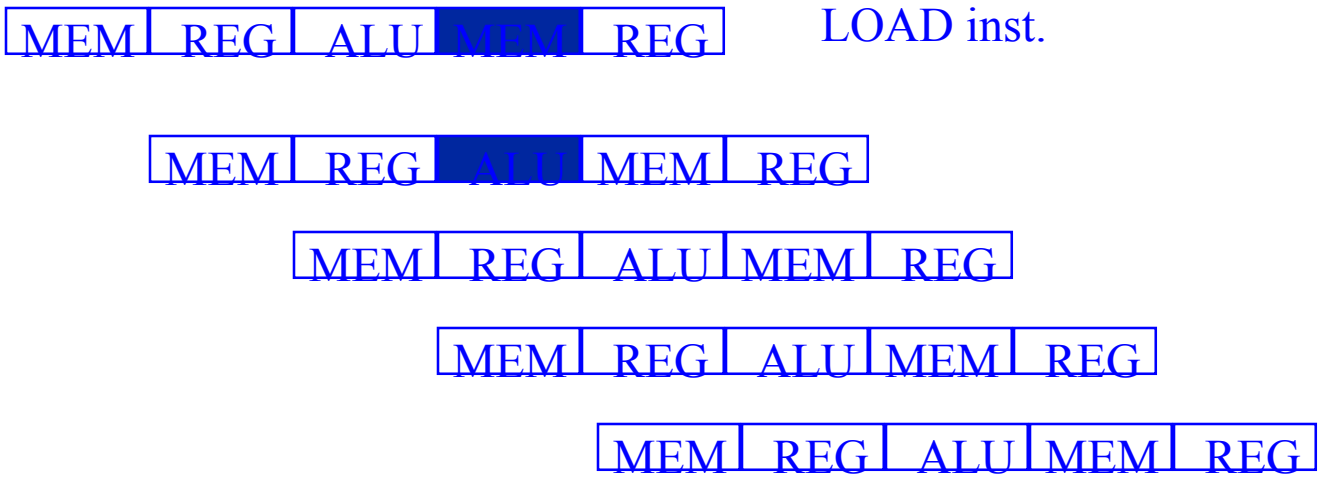
Data hazard - stalls

- All data hazards cannot be solved by forwarding:

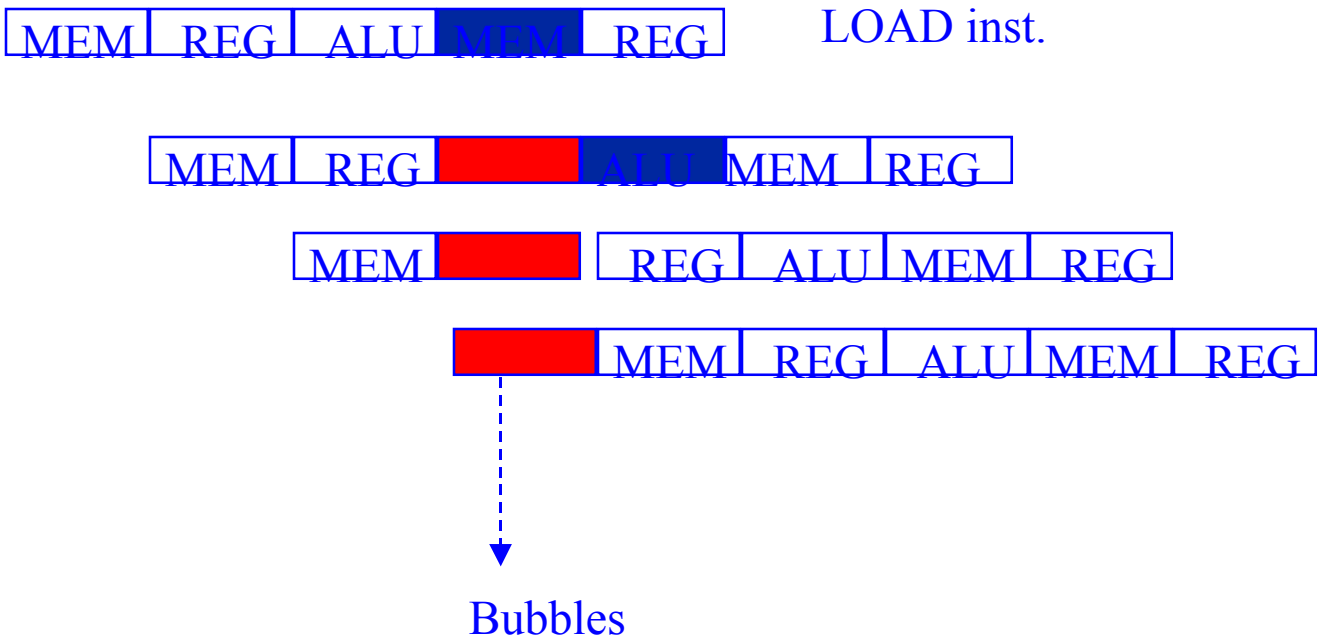
```
LW   R1,0(R2)
SUB  R4,R1, R5
AND  R6,R1,R7
OR   R8,R1,R9
```

- Unlike the previous example, data is available until MEM/WB. So subtract ALU cycle has to be stalled introducing a (vertical) bubble.

Data Hazard and Stalls



Data Hazard and Stalls



Summary

- Concepts in basic pipelining were studied in details.
- Data hazards and control hazards and methods for resolving these were also discussed.