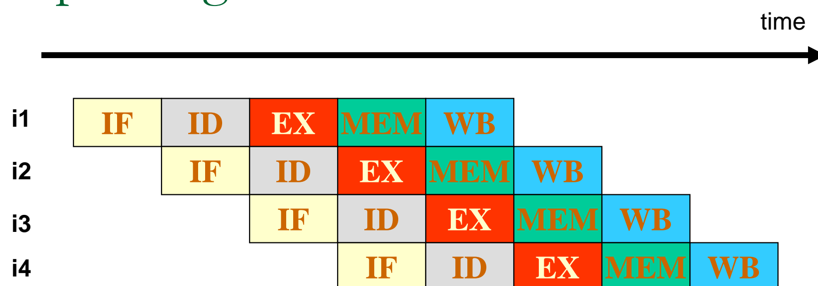
SE-292 High Performance Computing
Pipelining

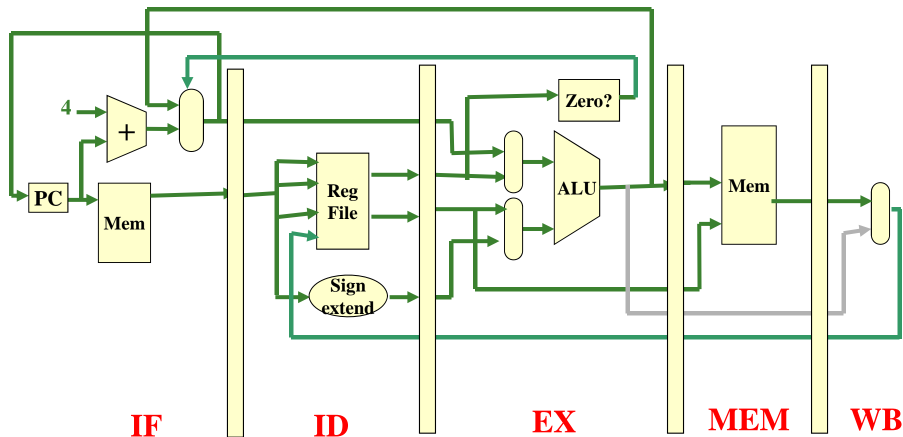
R. Govindarajan
govind@serc

Pipelining



- Execution time of instruction is still 5 cycles, but throughput is now 1 instruction per cycle
- Initial pipeline fill time (4 cycles), after which 1 instruction completes every cycle

Pipelined Processor Datapath

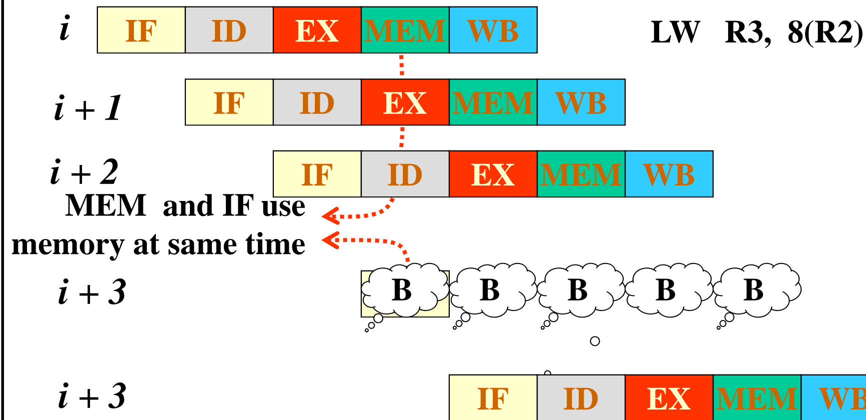


Problem: Pipeline Hazards

Situation where an instruction cannot proceed through the pipeline as it should

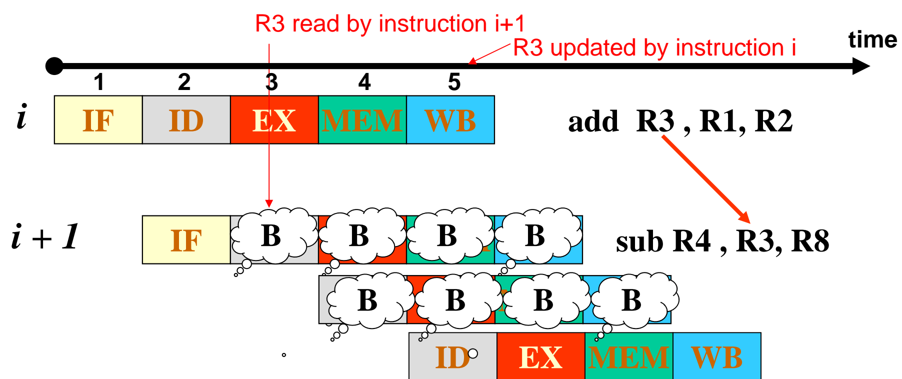
1. Structural hazard: When 2 or more instructions in the pipeline need to use the same resource at the same time
2. Data hazard: When an instruction depends on the data result of a prior instruction that is still in the pipeline
3. Control hazard: A hazard that arises due to control transfer instructions

Structural Hazard



5

Data Hazard



6

Data Hazard Solutions

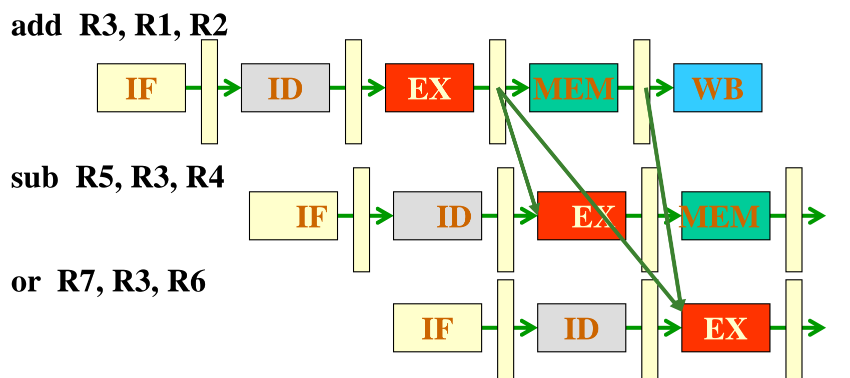
- Interlock: Hardware that detects data dependency and stalls dependent instructions

time →

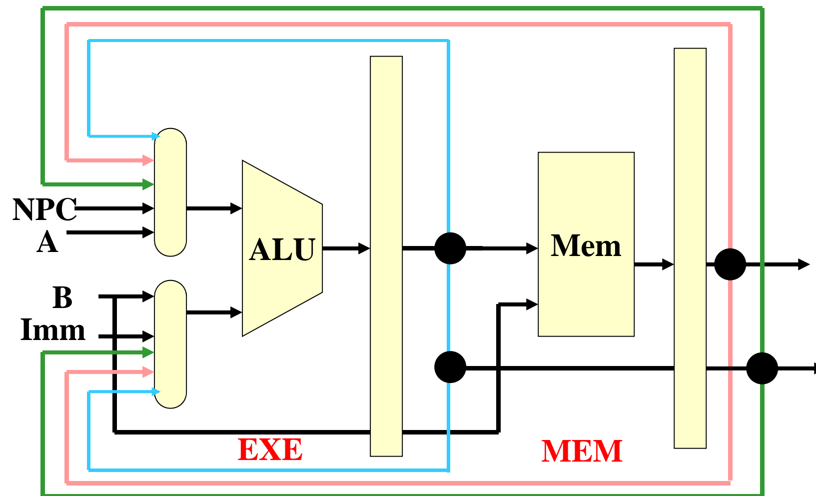
inst	0	1	2	3	4	5	6
Add	IF	ID	EX	MEM	WB		
Sub		IF	stall	stall	ID	EX	MEM
Or			stall	stall	IF	ID	EX

Data Hazard Solutions…

Forwarding or Bypassing: forward the result to EX as soon as available anywhere in pipeline



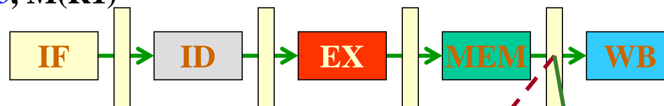
Modified Processor Datapath



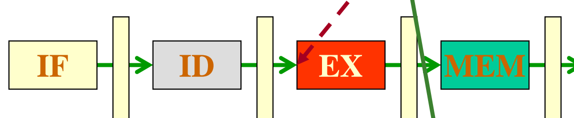
9

Forwarding is Not Always Possible

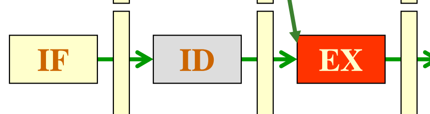
lw R3, M(R1)



sub R5, R8 R4



or R7, R3, R6



10

Other Data Hazard Solutions…

- Load delay slot
  - The hardware requires that an instruction that uses load value be separated from the load instruction
- Instruction Scheduling
  - Reorder the instructions so that dependent instructions are far enough apart
  - Compile time vs run time instruction scheduling

Static Instruction Scheduling

- Reorder instructions to reduce the execution time of the program
- Reordering must be safe – should not change the meaning of the program
- Two instructions can be reordered if they are independent of each other
- Static: done before the program is executed

Example

Program fragment:

```
lw   R3,  0(R1)
addi  R5, R3, 1      ⇨ 1 stall
add  R2, R2, R3
lw   R13, 0(R11)
add  R12,  R13,  R3  ⇨ 1 stall
```

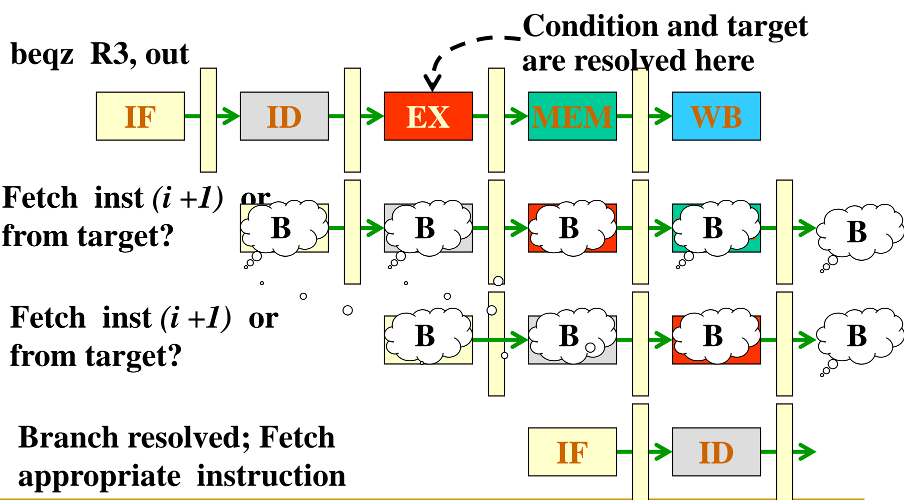
2 stalls

Scheduling:

```
lw R3, 0(R1)
addi R5, R3, 1
add R2, R2, R3
lw R13, 0(R11)
add R12, R13, R3
```

0 stalls

Control Hazards



Reducing Impact of Branch Stall

- Conditional branches involve 2 things
  1. resolving the branch (determine whether it is to be taken or not-taken)
  2. computing the branch target address
- To reduce branch stall effect we could
  - evaluate the condition earlier (in ID stage)
  - compute the target address earlier (in ID stage)
- Stall is then reduced to 1 cycle

15

Control Hazard Solutions

Static Branch Prediction

Example: Static Not-Taken policy

- Fetch instruction from PC + 4 even for a branch instruction
- After the ID phase, if it is found that the branch is not to be taken, continue with the fetched instruction (from PC + 4) -- results in 0 stalls
- Else, squash the fetched instruction and re-fetch from branch target address -- 1 stall cycle
- Average branch penalty is < 1 cycle

16

Control Hazard Solutions…

Delayed Branching

- Design hardware so that control transfer takes place ***after*** a few of the following instructions

```
add   R3, R2, R3
beq   R1, out
```

- Delay slots: following instructions that are executed whether or not the branch is taken
- Stall cycles avoided when delay slots are filled with useful instructions

Delayed Branching: Filling Delay Slots

- Instructions that do not affect the branching condition can be used in the delay slot
- Where to get instructions to fill delay slots?
  - From the branch target address
    - only valuable when branch is taken
  - From the fall through
    - only valuable when branch is not taken
  - From before the branch
    - useful whether branch is taken or not

Delayed Branching…Compiler's Role

- When filled from target or fall-through, patch-up code may be needed
- May still be beneficial, depending on branching frequency
- Difficulty in filling more than one delay slot
- Some ISAs include canceling delayed branches, which squash the instructions in the delay slot when the branch is taken (or not taken) – to facilitate more delay slots being filled

19

Pipeline and Programming

- Consider a simple pipeline with the following warnings in the ISA manual
  1. One load delay slot
  2. One branch delay slot
  3. 2 instructions after FP arithmetic operation can't use the value computed by that instruction
- We will think about a specific program, say vector addition

```
double A[1024], B[1024];
for (i=0; i<1024; i++)
    A[i] = A[i] + B[i];
```

20

Vector Addition Loop

/ R1: address(A[0]), R2: address(B[0])

```
Loop:  FLOAD  F0, 0(R1)     / F0 = A[i]        Loop:  FLOAD  F0, 0(R1)
       FLOAD  F2, 0(R2)     / F2 = B[i]               FLOAD  F2, 0(R2)
       FADD   F4, F0, F2    / F4 = F0 + F2            ADDI   R1, R1, 8
       FSTORE 0(R1), F4     / A[i] = F4               FADD   F4, F0, F2
       ADDI   R1, R1, 8     / R1 increment            ADDI   R2, R2, 8
       ADDI   R2, R2, 8     / R2 increment            BLE    R1, R3, Loop
       BLE    R1, R3, Loop                            FSTORE -8(R1), F4
                            / R3: address(A[1023])
```

11 cycles per iteration

7 cycles per iteration

An even faster loop? Loop Unrolling

- Idea: Each time through the loop, do the work of more than one iteration
  - More instructions to use in reordering
  - Less instructions executed for loop control
  - … but program increases in size

Unrolled loop

```
Loop:  FLOAD  F0, 0(R1)
       FLOAD  F2, 0(R2)
       FADD   F4, F0, F2
       FSTORE 0(R1), F4
       ADDI   R1, R1, 8      (struck out)
       ADDI   R2, R2, 8      (struck out)

       FLOAD  F0, 8(R1)
       FLOAD  F2, 8(R2)
       FADD   F4, F0, F2
       FSTORE 8(R1), F4
       ADDI   R1, R1, 8 16
       ADDI   R2, R2, 8 16

       BLE    R1, R3, Loop
```

18 cycles for 2 iterations
(9 cycles/iteration)

```
Loop:  FLOAD  F0, 0(R1)
       FLOAD  F2, 0(R2)
       FLOAD  F10, 8(R1)
       FLOAD  F12, 8(R2)
       FADD   F4, F0, F2
       FADD   F14, F10, F12
       ADDI   R1, R1, 16
       FSTORE -0 (R1), F4
       ADDI   R2, R2, 16
       BLE    R1, R3, Loop
       FSTORE -8 (R1), F4
```

Reorder to reduce to
5.5 cycles per iteration

Some Homework on Loop Unrolling

- Learn about the loop unrolling that gcc can do for you.
- We have unrolled the SAXPY loop 2 times to perform the computation of 2 elements in each loop iteration. Study the effects of increasing the degree of loop unrolling.

SAXPY Loop:

```
double a, X[16384], Y[16384], Z[16384];
for (i = 0 ; i < 16384; i++)
    Z[i] = a * X[i] + Y[i] ;
```

Assignment #1 (Due Date: Sept. 25, 2014)

1. Write a C program to check whether the machine uses the IEEE 754 standards for double precision FP representation; Find the largest and smallest positive numbers that it can represent, with and without the normalized representation. Check the representation for NaN, and infinity; Also find the machine epsilon.
2. Write a C program to identify in which region the following types of variables are stored:

   (a) global (b) local; (c) static, and (d) dynamically allocated .
3. Write a simple C program and generate the corpg. assembly language program for two different processors (e.g., x86 and MIPS). Explain how the function call mechanism works, including how the parameters are passed, how the stack/function frames are allocated, how the return address is retrieved, and how the stack and frame pointers are restored.
4. Consider the Vector add program (Z[i] = X[i] + Y[i]) and compile it with and without optimization for your machine. See the differences in the assembly code generated with and without the opt. Study the loop unrolling gcc can do. Find the optimal unrolling factor for an array size of 16384.