ECSE 425
Topic 3: Pipelining

Appendix A

1

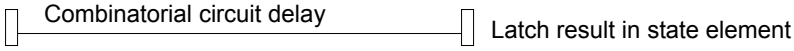
Introduction: What is pipelining?

- Organization technique whereby multiple instructions are overlapped in execution.
- Takes advantage of parallelism that exists among the actions needed to execute instructions.
- Today, pipelining is the key implementation technique used to make fast CPUs.
- Like car assembly line:
  - Many steps – each working on some part of car
  - Each step works in parallel to others but on a different car
- Computer: each step (*pipe stage, pipe segment*) in pipeline completes part of instruction.
- Different steps are completing different parts of different instructions in parallel.

2

What is Pipelining?

- The essential idea (not unique to CPU design): Consider the time needed to execute an instruction (gate delays), i.e. time between two clock edges

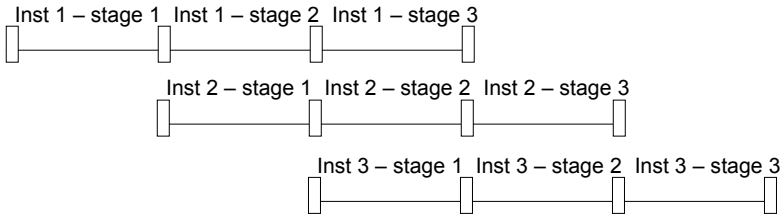


- The early gates are busy switching, while the later ones are idle: inefficient. Divide the instruction into stages and add a register between each stage:

3

Pipelining

Now each stage of the pipeline can work on a different instruction:



***Pipeline depth**:* number of stages in pipeline (here 3).
Processor cycle time: time required to move instruction one step down pipeline.
Throughput of instruction pipeline = number instructions completed per unit time, determined by how often an instruction exits the pipeline.

Here, throughput been increased by a factor three if we neglect the overhead due to pipe registers and other difficulties which are the topic of this chapter.

Notice at CC 3, each section of the hardware works on different instruction.

4

Pipelining

- Multiple instructions running in parallel!
  - Processor cycle determined by the longest stage.
  - Ideal condition achieved when all stages have the same delay (balanced)
  - *Ideally* (if balanced), throughput increases by number of pipe stages:
  - Pipelining reduces the average execution time per instruction
  - Many tradeoffs due to overheads incurred when adding more stages

5

Pipelining

- Reduction in average execution time reduces either CPI or Clock Cycle Time:
  - Time per instruction = Time per instruction unpipelined / # stages
- Throughput increases by a factor equal to the # stages
- Memory traffic increased (will demand the use of caches, Topic 5).
- Design challenges in hardware and for the compiler.
- A major driving factor for designing CPUs with simple ISA's (a.k.a. RISC design).

6

Pipelining

- First, consider implementation without pipelining for a simple machine that implements a subset of a RISC architecture: load-store word, branch, integer ALU ops.
- Every instruction can be implemented in at most 5 clock cycles.
  - Instruction Fetch (IF)
    - Send PC to memory and fetch the current instruction
    - Update the PC

7

Pipelining

  - Instruction Decode / Register Fetch (ID)
    - Decode instruction and read registers
    - Equality test on registers for a possible branch
      - Sigh-extend the offset field if needed and calculate the possible branch target address
    - Sigh-extend immediate if needed
  - Execution / Effective Address cycle (EX)
    - Memory reference: adds base register and offset to form effective address
    - Reg-reg ALU instruction
    - Reg-Imm ALU instruction

8

Pipelining

– Memory Access (MEM)
  - Load instruction: memory does a read
  - Store instruction: memory writes the data from register
– Write-back cycle (WB)
  - Register-Register ALU instruction, or Load instruction: write the result into register file

Branches: 2 cycles

Stores: 4 cycles

Others: 5 cycles

- Typical instruction mix: branches 12%, stores 10% → CPI = 4.54
- Not the most optimal design, can do better…but good basis for pipelining.

9

Pipelining

Classic Five-Stage RISC Pipeline

Starts a new instruction each clock cycle

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

10

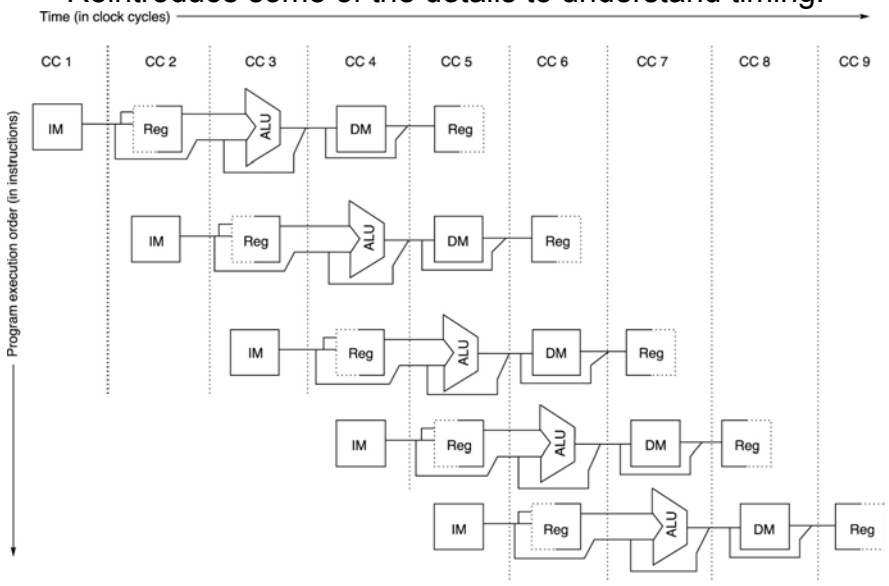
Pipelining

- Treat memory as two separate memories for now
- Clear definition of main functional units and high usage efficiency.
  - To reason about it, we now hide the details.

11

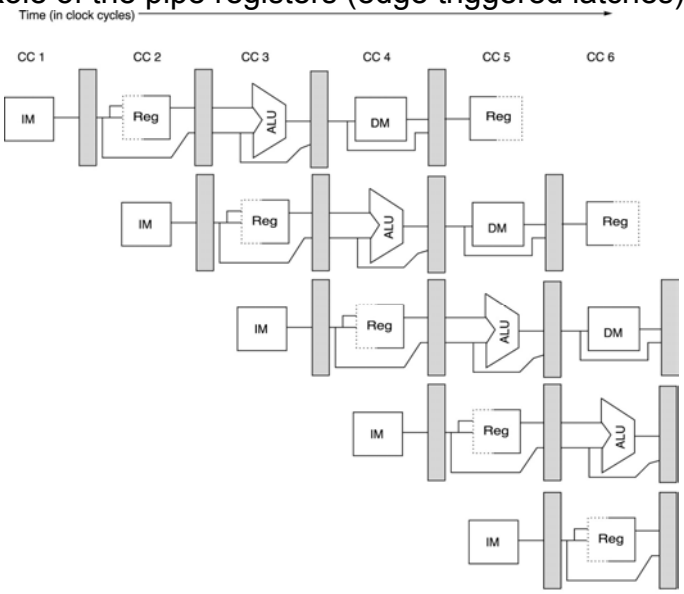
Pipelining

- Reintroduce some of the details to understand timing.



Pipelining

- Role of the pipe registers (edge triggered latches)



13

Pipelining

- ***Performance optimization***
- This is where digital system design meets organization design.
- Goal: increase the fetch rate, although instructions can take a long time to complete. Several limits on the optimization:
- Imbalance among different stages (clock can run no faster than slowest stage)
- Introduction of overhead due to extra control and latches.
- Sensitivity to clock skew (max delay between when clock arrives at 2 registers).

14

Pipelining

- ***Performance optimization***
- But there are other issues. We assumed so far that each instruction behaves like a sequence of stages with each stage consuming data produces by the previous stage so they can run in parallel.
- Unfortunately, this is clearly not the case in a program: one instruction can consume in one stage data produced by another instruction in the same or in another stage. There are also cases where two instructions would require the same functional unit simultaneously.
- These problems are collectively called *hazards*. We need some systematic method to reason about these.

15

Pipelining

- ***Pipeline Hazards***: If not treated, the CPU is *stalled* for 1 or more cycles.
- *Structural hazards*: Resource conflicts when hardware cannot support all possible combinations of overlapping instructions.
- *Data hazards*: Instruction needs data from a previous instruction (admittedly a common case) in a way such that overlapped execution causes problem.
- *Control hazards*: Caused by instructions (e.g. branches) which change the PC

16

Pipelining

- ***Performance of Pipelines with Stalls***
  - If we ignore the latches overhead, take the case of a perfectly balanced design and instructions with identical numbers of stages, performance can be estimated as a function of the statistical occurrence of hazards:

$$\text{SpeedUpPipe} = \frac{\text{AverageInstrTimeNoPipe}}{\text{AverageInstrTImePipe}} = \frac{\text{CPI\_NoPipe}}{\text{CPI\_Pipe}} \times \frac{\text{CC\_NoPipe}}{\text{CC\_Pipe}}.$$

But, $\text{CPI\_Pipe} = \text{CPI\_Ideal} + \text{StallsPerInstr} = 1 + \text{StallsPerInstr}$, and $\text{CPI\_NoPipe} = 1$.

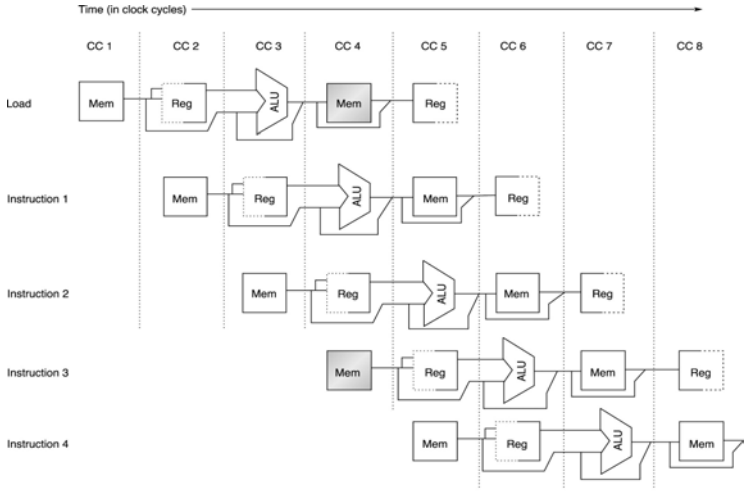
Also, $\frac{\text{CC\_NoPipe}}{\text{CC\_Pipe}} = \text{PipeDepth},$

so $\text{SpeedUpPipe} = \frac{\text{PipeDepth}}{1 + \text{StallsPerInstr}}$

17

Pipelining

- *Structural Hazards*
  - Typical occurrences: a functional unit is not pipelined (this happens commonly for FP Units), insufficient duplication of resources: Take the example of a CPU with a single memory port.



18

Pipelining

- This is more concisely seen using a timing diagram. The cure is of course a more capable memory sub-system if we decide the extra cost is worth the gain in performance. In this case the single memory port has a huge impact since it causes one stall per load (the most common instruction).

	Clock cycle number									
Instruction	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Pipeline Bubble “floats” through the pipeline.

19

Pipelining

- ***Cost of load structural hazard – example***:
  - Data reference: 40% of mix
  - CPI_ideal of pipelined processor (no structural hazard) = 1
  - Machine 1: processor with no structural hazard
  - Machine 2: processor with structural hazard
  - Machine 2 has clock rate 1.05 x faster than machine 1.
- Disregarding any other performance losses, which machine is faster?

Average instruction time = CPI x ClockCycleTime

Machine 1: Average instruction time = ClockCycleTime1

Machine 2:

Average instruction time =

(CPI_ideal + Clocks for hazard) x ClockCycleTime2 =

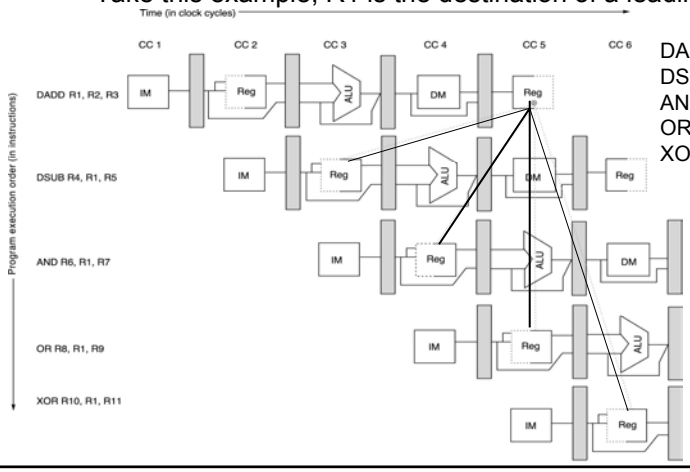
(1 + 0.4 x 1) x ClockCycleTime1/1.05 = 1.3 x ClockCycleTime1

- Machine 1 is 1.3 x faster!

20

Pipelining

- *Data Hazards*
  - Created when dependencies between instructions that overlap in pipelining that changes order of access to operands.
  - Take this example, R1 is the destination of a leading instruction

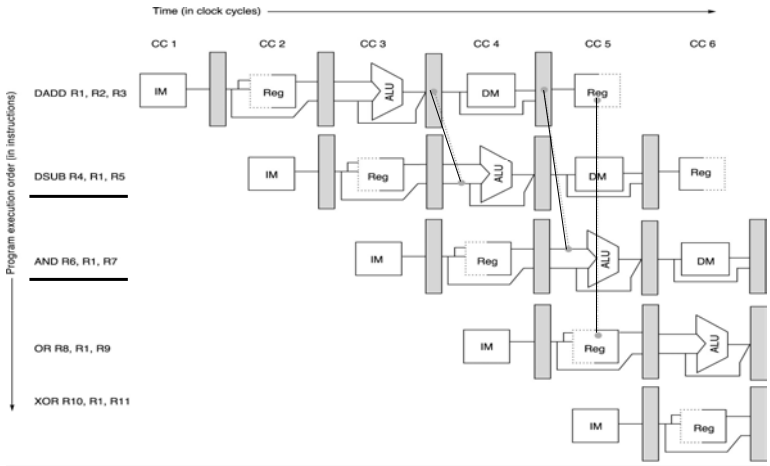


DADD	**R1**, R2, R3
DSUB	R4, **R1**, R5
AND	R6, **R1**, R7
OR	R8, **R1**, R9
XOR	R10, **R1**,R11

21

Pipelining

- Notice that the result of DADD is actually ready at the end of CC3 so with additional circuitry and control, the sequence could execute without stalls. This is called *forwarding* (or *bypassing*). It is implemented by adding multiplexers to the data path.

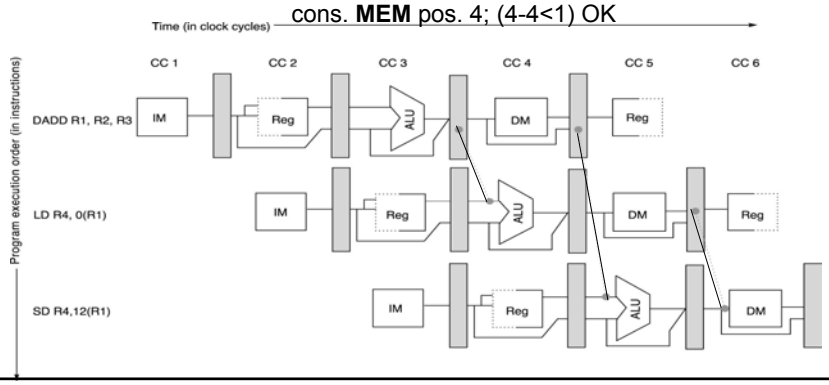


22

Pipelining

- Forwarding can be used whenever the difference between the positions of the producer stage and the consumer stage is smaller than the difference of fetch clock cycles. This is illustrated by looking at the interaction between the EX stage and the MEM stage (a general concept).

DADD **R1**,R2,R3 // *i*, prod. **EX** pos. 3;
LD **R4**,0(**R1**) // *i*+1, cons. **EX** pos. 3, (3–3<1) OK; prod. **MEM** pos. 4
SD **R4**,12(**R1**) // *i*+2, cons. **EX** pos. 3, (3–3<2) OK;
cons. **MEM** pos. 4; (4-4<1) OK

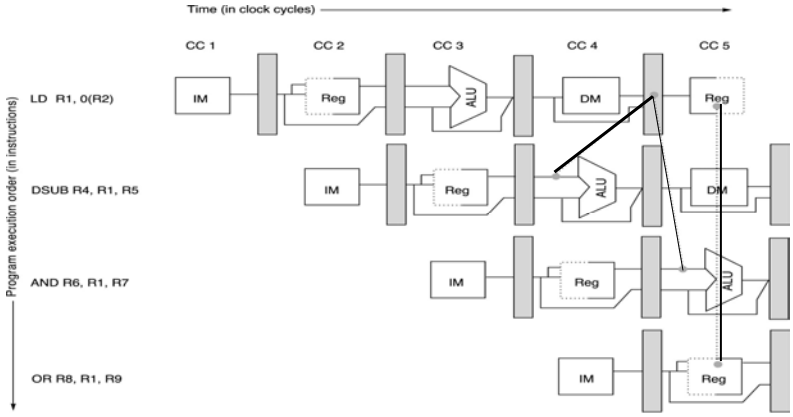


23

Pipelining

- *Required stalls* must occur when the relationship is broken

LD R1,0(R2) // *i*, prod. MEM pos. 4
DSUB R4,R1,R5 // *i*+1, cons. EX pos. 3; ¬(4-3<1) STALL needed
AND R6,R1,R7 // *i*+2, cons. EX pos. 3; (4-3<2) OK
OR R8,R1,R9 // *i*+3, cons. EX pos. 3; (4-3<3) OK



24

Pipelining

- Again, this is more concisely seen in the form of a timing diagram.

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

Notice the **column** of stalls in the bottom timing diagram that executes correctly:
a stalled instruction occupies a stage and prevents all following instructions from proceeding to the next stage.

25

Pipelining

- *Code Scheduling*
  - In the situations when the CPU either is not or partially bypassed or stalls are required as in the case of load stalls for 5 stage MIPS pipeline, the stalls can be reduced or even totally eliminated by re-ordering independent instructions This known as *static scheduling* (one of the target-specific back-end compiler optimizations). For example consider this:

a = b + c
d = e - f

Raw compilation

```
LW      Rb,…
LW      Rc,…
Stall
DADD    Ra,Rb,Rc
SW      Ra,…
LW      Re,…
LW      Rf,…
Stall
DSUB    Rd,Re,Rf
SW      Rd,…
```

Scheduled code

```
LW      Rb,…
LW      Rc,…
LW      Re,…
DADD    Ra,Rb,Rc
LW      Rf,…
SW      Ra,…
DSUB    Rd,Re,Rf
SW      Rd,…
```

26

Pipelining

- *Control Hazards*
  - It's the same idea, except now the consuming stage is always IF. The value of the PC is determined later in the pipeline if branch taken. In meantime, if nothing is done in hardware, the CPU will fetch possibly *n* incorrect *fall-through* instructions, where *n* is the position difference between the fetch stage and the stage the branch outcome is determined.

27

Pipelining

- *Control Hazards*
  - The simplest approach is to re-fetch once the branch is decoded in ID, resulting in always wasting a clock cycle for each branch when in fact the fetched instruction could have been the right one if the branch is not taken.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

- Successors incorrectly fetched if branch is taken.

28

Pipelining

- *Control Hazards*
  - Designers have invented all kind of schemes to squeeze more performance for branches. We look at four (among others) applicable to single pipelines.
  - (1) *Flush* pipeline (easily done by clearing pipe registers), once the branch is known (solution above) until branch destination known.

29

Pipelining

- *Control Hazards*
  - (2) *Predict-not-taken* scheme is only slightly more complicated. Proceed as above, but re-fetch instruction only if the branch is taken.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

30

Pipelining

- *Control Hazards*
- (3) For some processors, the branch target is known before the branch outcome, so it makes sense to organize the pipeline in a *predict-taken* fashion, i.e. fetch first instruction at the target address and restart the fetch sequence at the fall-through sequence if the prediction is incorrect.
- In both predict-not-taken and predict-taken, the compiler has a role to play when translating if and for statements which have lopsided statistics accounting for the target CPU.
- (4) A popular scheme is the *delayed branch*. The idea, in its simplest expression, is to do nothing at all in hardware and leave it to the compiler to deal with the problem: one or several instructions following the branch will execute **even if** the branch is taken. It is actually not as silly as it seems.

31

Pipelining

- *Delayed branch*

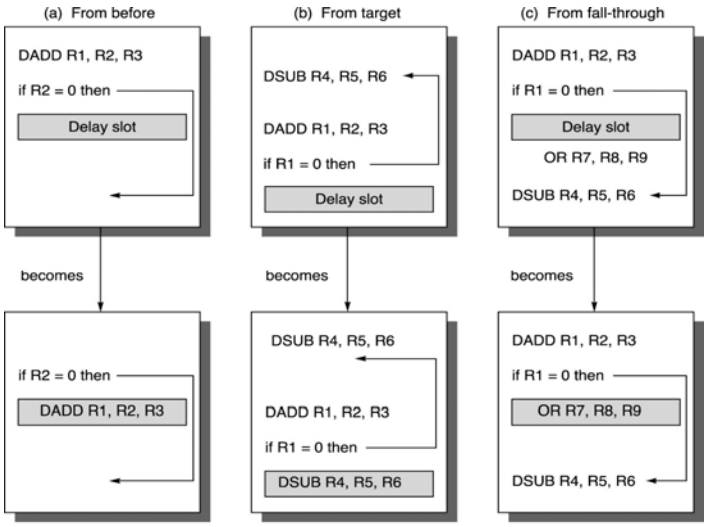
Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

32

Pipelining

- Illustrated in the next figure are three options for the compiler to move instruction(s) into the *delay slot*(s).



33

Pipelining

- Limitations on delayed-branch scheduling:
  - Restrictions on the instructions that are scheduled into delay slots
  - Our ability to predict at compile time whether branch is likely to be taken or not
- Finally, the delayed branch scheme can be advantageously combined with pipeline flush, this is called *canceling*.
- *Performance of branch schemes.*
  - Again, if we can statistically figure the frequency of occurrence of mis-predictions as well as the cost incurred. We can write:

$$\text{SpeedUpPipe} = \frac{\text{PipeDepth}}{1 + \text{BranchFrequency} \times \text{BranchPenalty}}$$

34

Pipelining

- Example: MIPS R4000
  - Deeper pipeline – takes 3 pipeline stages before branch-target address known + additional cycle before branch condition evaluated (assuming no stalls on register in conditional comparison)
  - Delay leads to branch penalties for 3 simplest prediction schemes:

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

  - Find effective addition to CPI from branches in pipeline assuming frequencies:
    - Unconditional branch 4%
    - Conditional branch, untaken 6%
    - Conditional branch, taken 10%

Branch scheme	unconditional	untaken	taken	All branches
Frequency	4%	6%	10%	20%
Flush pipeline	0.08	0.18	0.3	0.56
Predicted taken	0.08	0.18	0.2	0.46
Predicted untaken	0.08	0	0.3	0.38

35

Simple implementation of MIPS

- **<u>How is Pipelining Implemented?</u>**
  - Simple, unpipelined MIPS: load-store word, branch equal zero, integer ALU ops (not aggressive implementation of a branch instruction)
- IF:

  IR ← Mem[PC];

  NPC ←PC + 4;
- ID:

  A ←Regs[rs];

  B ← Regs[rt];

  Imm ←sign-extended immediate field of IR;

36

Simple implementation of MIPS

- **How is Pipelining Implemented?**
- EX:
  - Mem ref:
    - ALUOutput ← A + Imm;
  - Reg-Reg ALU op:
    - ALUOutput ← A func B;
  - Reg-Imm ALU op:
    - ALUOutput ←A op Imm;
  - Branch:
    - ALUOutput ← NPC + (Imm << 2);
    - Cond ← (A == 0)

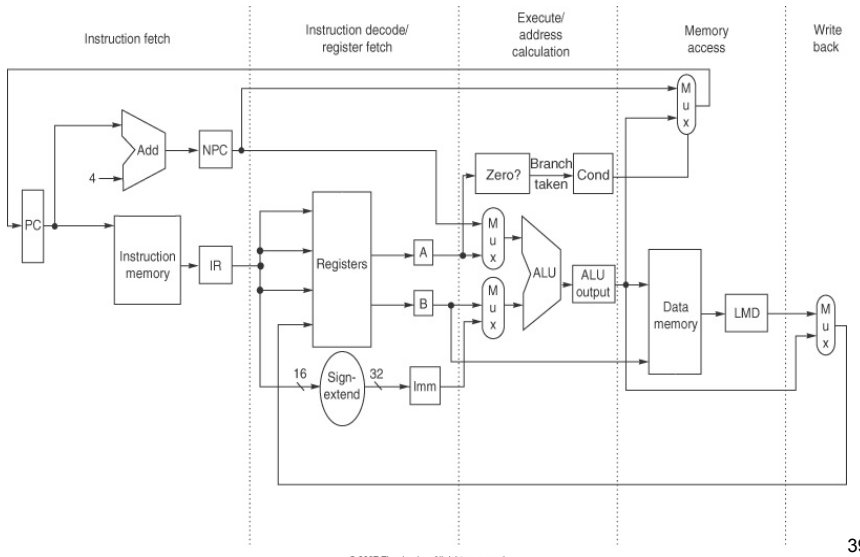
37

Simple implementation of MIPS

- **How is Pipelining Implemented?**
- MEM:
  - PC ← NPC;
  - Mem ref:
    - LMD ←Mem[ALUOutput] or Mem[ALUOutput] ← B;
  - Branch:
    - If (cond) PC ← ALUOutput;
- WB:
  - Reg-Reg ALU:
    - Regs[rd] ←ALUOutput;
  - Reg-Imm ALU:
    - Regs[rt] ← ALUOutput;
  - Load:
    - Regs[rt] ← LMD;

38

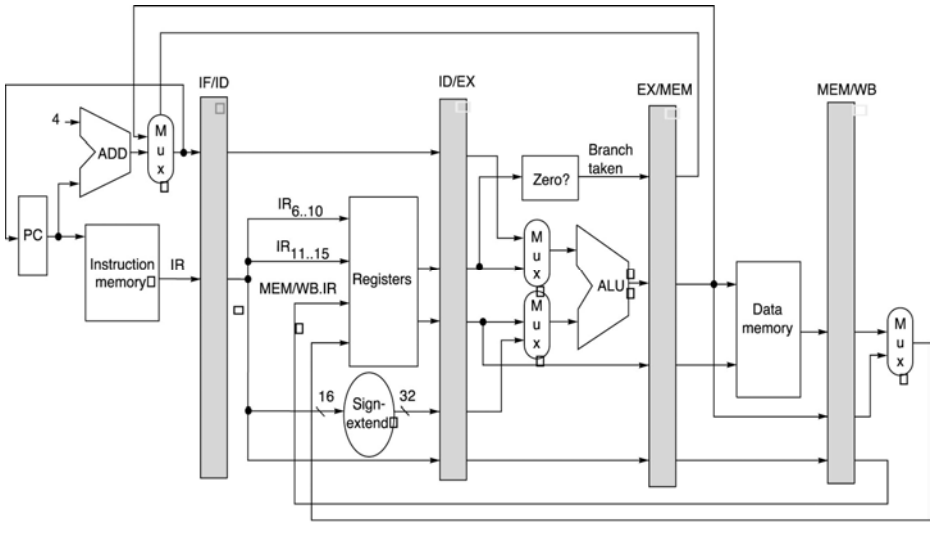
Simple implementation of MIPS



39

Basic MIPS Pipeline

- 1 ALU, 2 Delay Slots



Basic MIPS pipeline

Stage	Any instruction		
IF	IF/ID.IR←Mem[PC]; IF/ID.NPC,PC ←(if ((EX/MEM.opcode==branch) & EX/MEM.cond) {EX.MEM.ALUOutput} else {PC+4});		
ID	ID.EX.A ←Regs[IF/ID.IR[rs]]; ID/EX.B ←Regs[IF/ID.IR[rt]]; ID/EX.NPC ←IF/ID.NPC; ID/EX.IR ←IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);		
	ALU instruction	**Load or store instruction**	**Branch instruction**
EX	EX/MEM.IR ←ID/EX.IR; EX/MEM.ALUOutput ← ID.EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ←ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm<<2); EX/MEM.cond ← (ID/EX.A ==0);
MEM	MEM/WB.IR ←EX/MEM.IR; MEM/WB.ALUOutput ←EX/MEM.ALUOutput;	MEM/WB.IR ←EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ←EX/MEM.B;	
WB	Regs[MEM/WB.IR[rd]] ←MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ←MEM/WB.ALUOutput;	For load only; Regs[MEM/WB.IR[rt]] ←MEM/WB.LMD;	

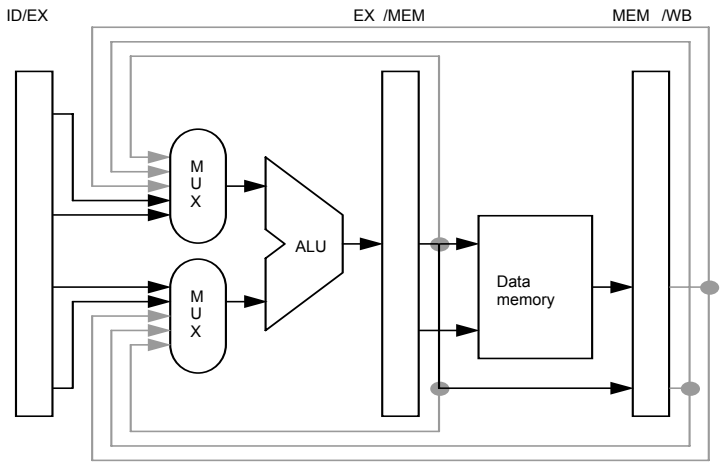
41

Implementing Control for the MIPS pipeline

- All data hazards can be checked during ID phase. If exists – stalled before *issued* (ID – EX). Can determine what forwarding will be needed during ID and set controls then.
- Detect interlocks early!
- Pipeline hazard detection hardware – compare destination and sources of adjacent instructions (only need to compare on 2 instructions following instruction that wrote destination)
- Once hazard detected, insert pipeline stall (change instruction to no-op)
- Forwarding from: ALU output, data memory output to
  - ALU input, data memory input, zero detection
  - This implies additional multiplexer inputs + add connections from pipeline registers

42

Forwarding



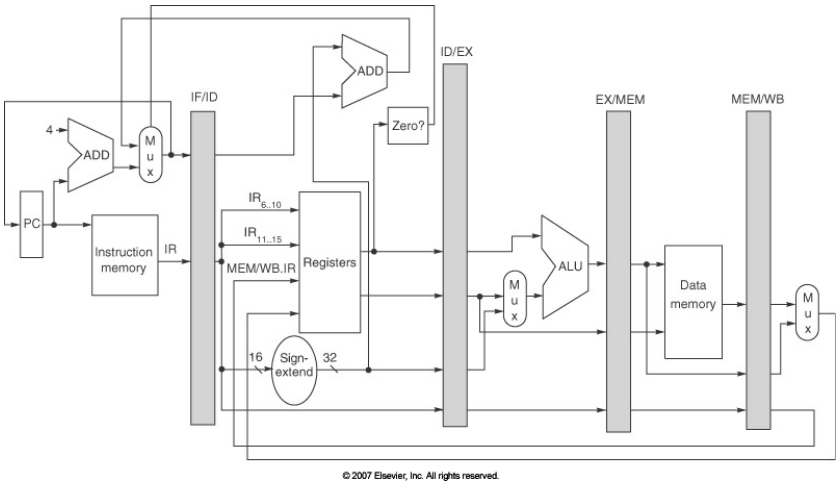
43

Dealing with branches in the pipeline

- **Dealing with branches in the pipeline: aggressive implementation**
- **Consider the cases BEQZ and BNEZ**
  - **Moving the zero test into ID cycle**
  - **Compute the branch-target address during ID**
    - **Additional adder**
- **1 clock-cycle stall on branches**
- **(Figure A.25: self study)**

44

Dealing with branches in the pipeline



45

Exceptions

- So far, we have assumed that the CPU executes a single program and that there was no external events that could disrupt the instruction sequence. Obviously, it is not the case, save for the simplest applications. Major cases:
  - I/O device request
  - Invoking OS service from user program
  - Tracing instructions
  - Breakpoints
  - Integer arithmetic overflow
  - FP arithmetic anomaly
  - Page fault (related to virtual memory, Chap 5)
  - Misaligned memory access
  - Memory protection violation
  - Undefined instructions
  - Hardware malfunctions of various kinds
  - Power failure

46

Exceptions

- *Properties of exceptions*
  - *Synchronous / asynchronous*:
    - occurs always the same place/data or randomly
  - *User requested / coerced*:
    - has been programmed by the user or external event
  - *Maskable / nonmaskable*:
    - may be programmed to be ignored by hardware
  - *Within / between instructions*:
    - prevents instruction completion (asynchronous: terminal) or not
  - *Resume / terminate*:
    - CPU can restart where it was interrupted after service

47

Exceptions

I/O	Async.	Coerced	Nonmaskable	Between	Resume
OS Invoke	Sync.	Requested	Nonmaskable	Between	Resume
Trace Instr. ex.	Sync.	Requested	Maskable	Between	Resume
Breakpoint	Sync.	Coerced	Maskable	Between	Resume
Int. Overflow	Sync.	Coerced	Maskable	Within	Resume
FP Exception	Sync.	Coerced	Maskable	Within	Resume
Page Fault	Sync.	Coerced	Nonmaskable	Within	Resume
Misalig. Acc.	Sync.	Coerced	Maskable	Within	Resume
Mem. Protection	Sync.	Coerced	Nonmaskable	Within	Resume
Undef. Instruction	Sync.	Coerced	Nonmaskable	Within	Terminate
Hardw. Fault	Async.	Coerced	Nonmaskable	Within	Terminate
Power Failure	Async.	Coerced	Nonmaskable	Within	Terminate

48

Difficulties with exceptions

- Hardest case: within instruction and restartable (Page fault). Hardware must:
  - Shut down the pipeline with instructions at various stages of completion.
  - Restart the pipeline in the same state when it was when interrupted.
- In more detail, steps to save the pipeline safely:
  - Force a trap instruction into the pipeline on the next IF. This will cause the CPU to switch to an *exception handling routine*.
  - Turn off all the writes for the faulty instruction and those that follow but not for those that precede. This will make it possible to restart since preceding instructions can complete as if nothing happened, and those after it can start from scratch. This is called *precise exception* handling.
  - Once control is given to the exception handling routine, save PC of faulty instruction to restart with it.
- With floating point operations (see later) that can take unpredictable number of clock cycles. Certain designs have a unprecise mode for floating point to boost performance (fewer things to do).

49

Exceptions in MIPS

- Some of the exceptions that can happen:

IF	Page Fault, misalignment, memory protection
ID	Undefined instruction
EX	Arithmetic exception
MEM	Page Fault, misalignment, memory protection
WB	None

Example

LD	IF	ID	EX	**MEM**	WB	
DADD		IF	ID	**EX**	MEM	WB

Multiple exceptions can occur in the same clock cycle, e.g. a page fault and an arithmetic exception. There can also be a data page fault in the MEM stage of the LD and an instruction page fault in the IF of the DADD which occurs earlier. Exceptions raised by LD must be serviced first and then those of the DADD.
Solution: Post all exceptions in a status vector associated with each instruction in the pipe which is checked when the instruction completes. If exceptions are raised they can be processed in order.

50

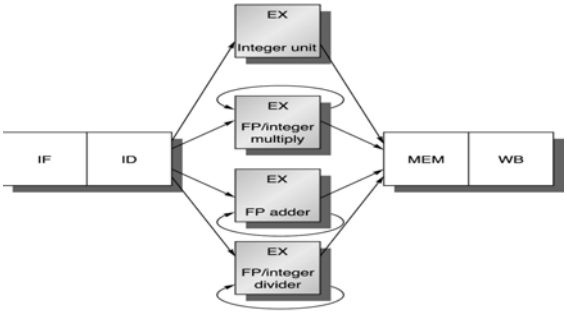
Instruction Set Complications

- *More complications*
- With CPU having instructions that write only once, precise exceptions are relatively straightforward since there is a one-to-one correspondence between a fetch and the eventual state change, e.g. MIPS integer pipeline writes at the end of the MEM state only. If an instruction is guaranteed to complete, it is said to be *committed* and the write can be done anytime later.
- If there can be a change of state in the middle of the instruction execution, e.g. multiple writes (e.g. autoincrement addressing mode, condition codes, and so-on), in order to maintain a precise exception model, processor must be able to back out of changes made. This can involve undoing the writes!
- Again the problem of exceptions enters into the CISC/RISC argument.

51

Extending MIPS to handle Multicycle Operations

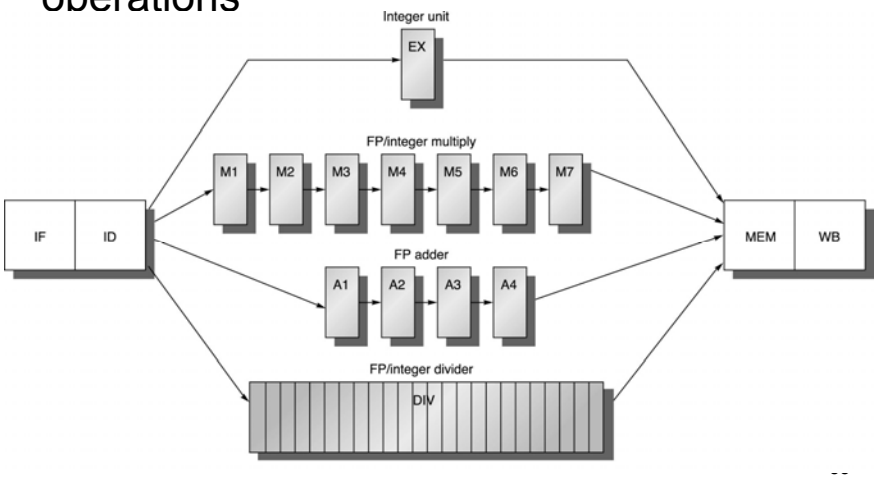
- This concerns FP ops which require many more (and even unpredictable) clock cycles than integer units. The solution is to provide distinct functional units:
  - Integer (regular ALU considered so-far),
  - FP and integer multiplier,
  - FP adder (handles FP add, subtract, convert),
  - FP and integer divider. Not pipelined.



52

Extending MIPS to handle Multicycle Operations

- Pipeline that supports multiple FP operations



Extending MIPS to handle Multicycle Operations

- *Systematic Description*
- *Initiation interval*: number of clock cycles between issuing 2 operations of a given type.
- *Latency*: number of intervening clock cycles between producing unit (providing result) and consuming unit (using result). When they are at the same position in a bypassed pipeline it also applies to instructions.

54

Extending MIPS to handle Multicycle Operations

Functional Unit	Latency	Initiation Interval
Integer ALU (EX)	0	1
Data Memory (integer and FP)	1	1
FP add	3	1
FP and int. multiply	6	1
FP and int. divide	23	24

Comments:

•Possibility for structural hazards, *e.g.* divide unit not pipelined, several instructions could compete to write to register.

•New kinds of hazards: an instruction fetched after another instruction could write the register before the earliest one!

•More and longer stalls.

55

Classification of data hazards

- We need a way to reason more systematically about data hazards. In the next chapter we will generalize this even further. For now, consider pairs of instructions which *depend* on each other because they share a register written by a producer and read by a consumer instruction. This holds for bypassed data paths since forwarding can only reduce latency but not make it negative! With this in mind, look at all cases:
- RAW (read after write). A pair of instructions has a consumer trying to read before the producer writes.
  - The most common type: e.g. a = 0; …; a = 1; b = a + 1, then, we expect b to have the value 2, not 1.
- WAW (write after write). A pair of instructions are producers but the instruction that writes first appears after in the program:
  - e.g. a = 1+2; …;a = 2*3, then, we expect a to have the value 6, not 3.
- WAR (write after read). A producing instruction occurs after a consuming instruction but writes before the consumer reads:
  - e.g. a = 1; b = a + 1; a = 2, then, we expect b to have the value 2, not 3.

56

Hazards and Forwarding in Longer Latency Pipelines

- Now we can examine examples of code sequences for a multicyle pipeline involving such hazards; Yellow marks: RAW, red marks: structural hazards.
- In the following, they are treated by stalling the pipeline at appropriate places.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	ME	W	B											
MUL.D F0,F4,F6		IF	ID	stall	M	M2 1	M3	M4	M5	M6	M7	ME	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	ME	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	ME

57

Hazards and Forwarding in Longer Latency Pipelines

- In this example, hazards are not treated. Note at cycle 10, only the L.D uses the memory but at 11, three instructions compete for the register file. Here, delaying the ADD.D rather than the L.D would create a WAW hazard.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	ME	WB					
…		IF	ID	EX	ME	WB										
…			IF	ID	EX	ME	WB									
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	ME	WB					
…					IF	ID	EX	ME	WB							
…						IF	ID	EX	ME	WB						
L.D F2,0(r2)							IF	ID	EX	ME	WB					

58

Hazards and Forwarding in Longer Latency Pipelines

- There is a number of structural hazards. The way they are treated is dependent on the details of hardware design. For example, most machines would not allow two instructions in the same stage to share the same pipe register.

59

Hazards and Forwarding in Longer Latency Pipelines

- In summary, multicycle operation introduce new kinds of hazards which need to be detected and treated. In general, three checks must be performed before an instruction can *issue*, that is transit from decode to execute. Assuming that all hazard detection is done in the ID stage:
- Check for structural hazards, delay until the required unit is available (e.g. divide unit here).
- Check for RAW hazard. Delay until the source registers are not listed as pending destinations.
- Check for WAW hazard. Delay the instruction ID.

60

Maintaining Precise Exceptions

- Finally, there are further complications with exceptions of course.

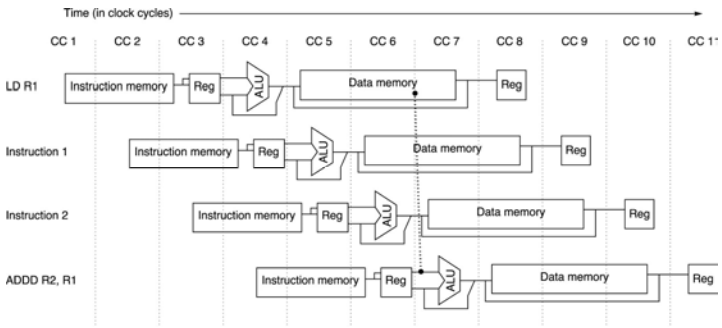
```
DIVD.D      F0,F2,F4
ADD.D       F10,F10,F8
SUB.D       F12,F12,F14
```

- The SUB.D could cause an imprecise exception when the DIVD.D is not yet completed but the ADD.D has (*out-of-order completion*). In this case the ADD.D would have destroyed one operand, so the exception cannot be precise.

61

Superpipelining

- This refers to subdividing the clock cycles even further, yielding major cycles in a deeper pipeline. This way a stage can span a variable number of minor clock cycles. For example, the MIPS R4000 superpipeline.



Everything we have seen generalizes with more hazards, longer delays, more difficult exceptions, but hopefully higher throughput.

62

Fallacies and Pitfalls

- **Section A.6: Self Study**
- **Section A.7: OMIT**
- **Pitfall**: *Unexpected execution sequences may cause unexpected hazards*.
- WAW hazard to occur in normal code because of branches.

```
        BNEZ   R1,foo
        DIV.D  F0,F2,F4        // delay slot fill-in from fall-through
        …
foo:    L.D    F0,something
```

- Predicted untaken and the branch is, in fact, taken. L.D will reach WB before DIV.D can complete. Optimize for the best but always expect the worse!
- **Pitfall**: *Extensive pipelining can impact other aspect of a design, leading to overall worse cost-performance.*
- Industrial examples (see text). Can end up with same performance with less hardware.

63