Computer Architecture

Pipelining and Instruction Level Parallelism–An Introduction

Outline of This Lecture

Introduction to the Concept of Pipelined Processor

- Pipelined Datapath and Pipelined Control
- Pipeline Example: Instructions Interaction

Pipeline Hazards

- Forwarding
- Stalls

Introduction to Instruction Level Parallelism

- Superscalar, VLIW
- Out-of-order execution
- Branch Prediction
- Future

The Five Stages of Load

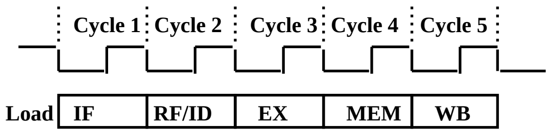
IF: Instruction Fetch
- Fetch the instruction from the Instruction Memory

RF/ID: Registers Fetch and Instruction Decode

EX: Calculate the memory address

MEM: Read the data from the Data Memory

WB: Write the data back to the register file



Key Ideas Behind Pipelining

Analogy–Grading the mid term exams:
- 6 problems, six people grading the exam
- Each person grades ONE problem
- Pass exam to next person as soon as one finishes her part
- Assume each problem takes 0.15 hour to grade
  - Each individual exam still takes 0.9 hours to grade
  - But with 6 people, all exams can be graded much quicker:
    - 100 exams: 90 hours, vs. 90 hrs x 6 = 540 hours

The load instruction has 5 stages:
- Five independent functional units to work on each stage
  - Each functional unit is used only once
- Another load can start as soon as 1st finishes its IF stage
- Each load still takes five cycles to complete
- The throughput, however, is much higher

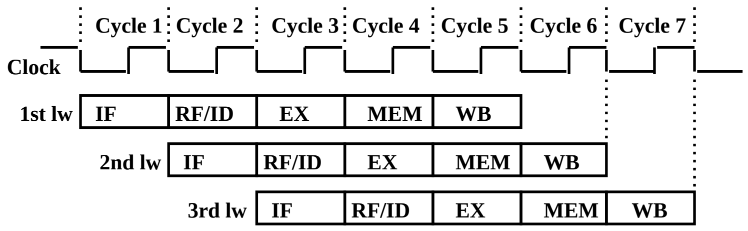
Pipelining the Load Instruction

Five independent functional units in pipeline are:

- Instruction Memory for the IF stage
- Register file's read ports for the RF/ID stage
- ALU for the EX stage
- Data Memory for the MEM stage
- Register File's Write port (bus W) for the WB stage

1 instruction enters the pipeline every cycle

- 1 instruction comes out of pipeline (completes) every cycle
- "Effective" Cycles per Instruction (CPI) is 1



Four Stages of R-type

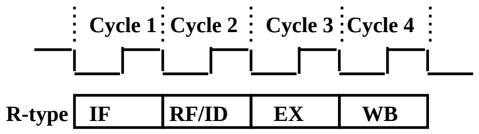
IF: Instruction Fetch

- Fetch the instruction from the Instruction Memory

RF/ID: Registers Fetch and Instruction Decode

EX: ALU operates on the two register operands

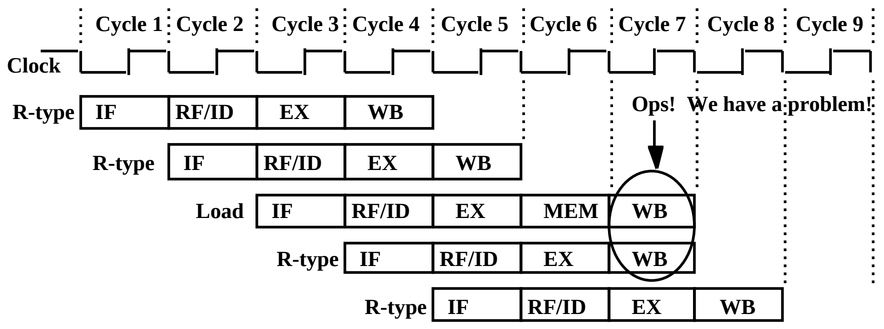
WB: Write the ALU output back to the register file



Pipelining R-type + Load

We have a problem:

– Two instructions try to write to register file at same time!



Important Observation

A functional unit can be used *once* per instruction

Each functional unit must be used at same stage for all instructions:

– Load uses Register File's Write Port during its 5th stage

–

•

–

–

–

	1	2	3	4	5
Load	IF	RF/ID	EX	MEM	WB

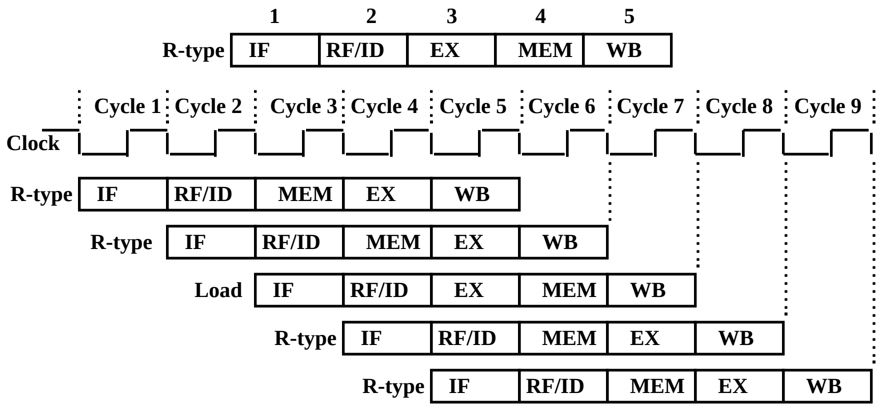
– R-type uses Register File's Write Port during its 4th stage

	1	2	3	4
R-type	IF	RF/ID	EX	WB

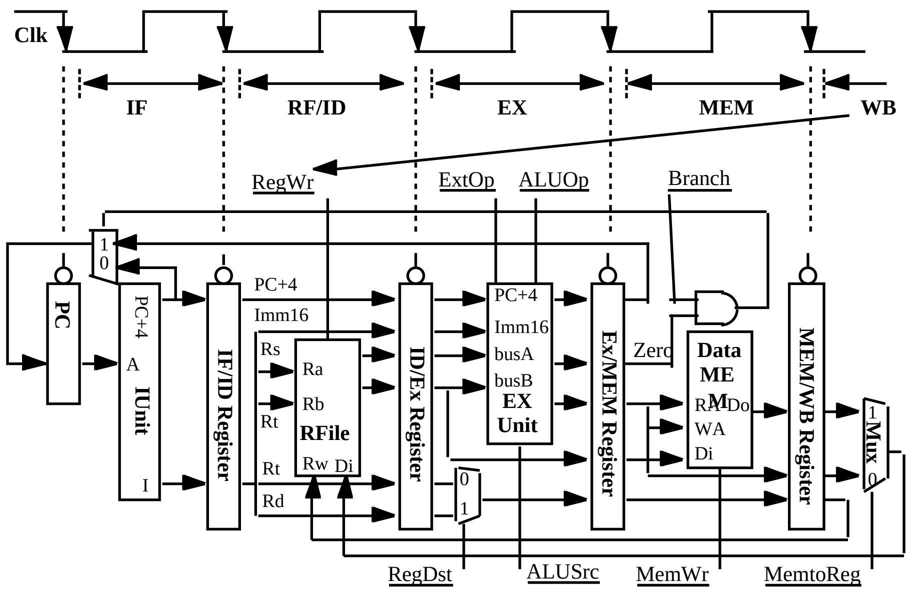
Solution: Delay R-type WB a Cycle

Delay R-type's register write by one cycle:

- **R-type instructions also use Reg File's write port at Stage 5**
- **MEM stage is a NOOP stage: nothing is being done**

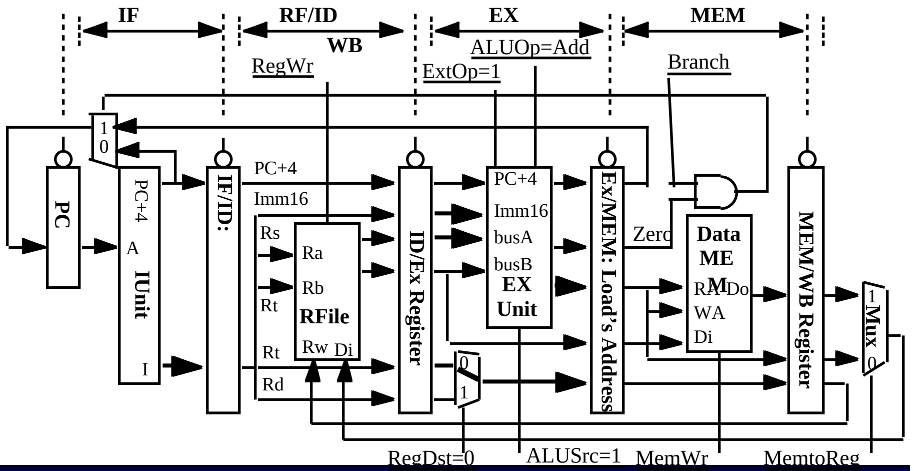


A Pipelined Datapath



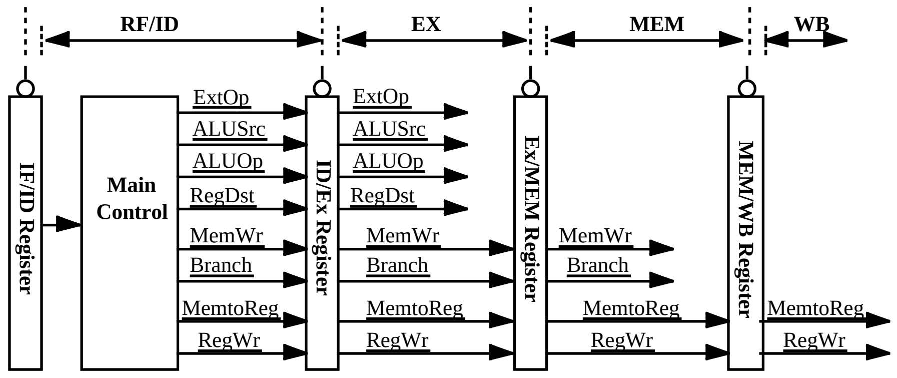
How About Control Signals?

- Control Signals at Stage N = Func (Instr. at Stage N)
  - N = EX, MEM, or WB
- Example: Controls Signals at EX Stage
  - Func(Load's EX)

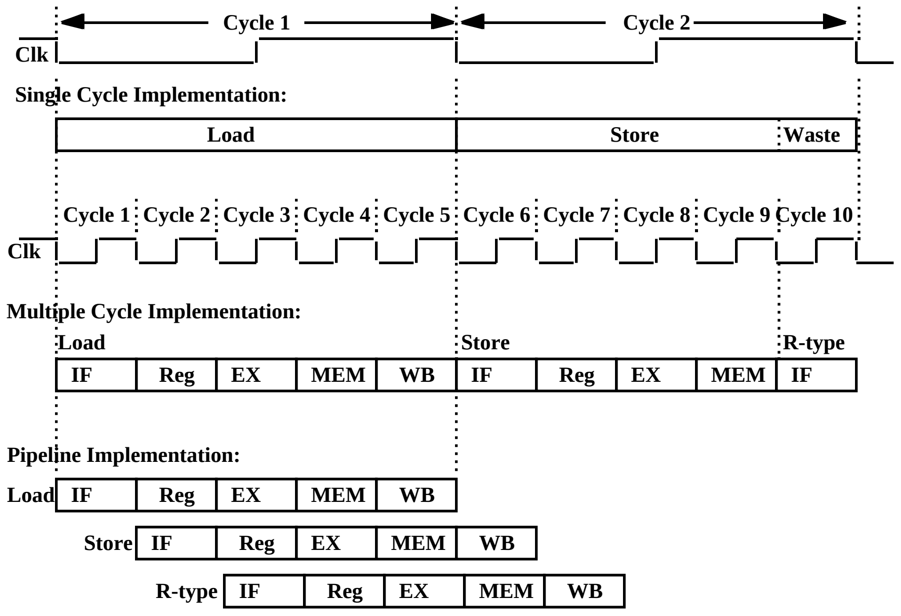


Pipeline Control

- The Main Control generates the control signals during RF/ID
  - Control signals for EX (ExtOp, ALUSrc, ...) used 1 cycle later
  - Control signals for MEM (MemWr, Branch) used 2 cycles later
  - Control signals for WB (MemtoReg MemWr) used 3 cycles later



Single Cycle, Multi-Cycle, Pipelined



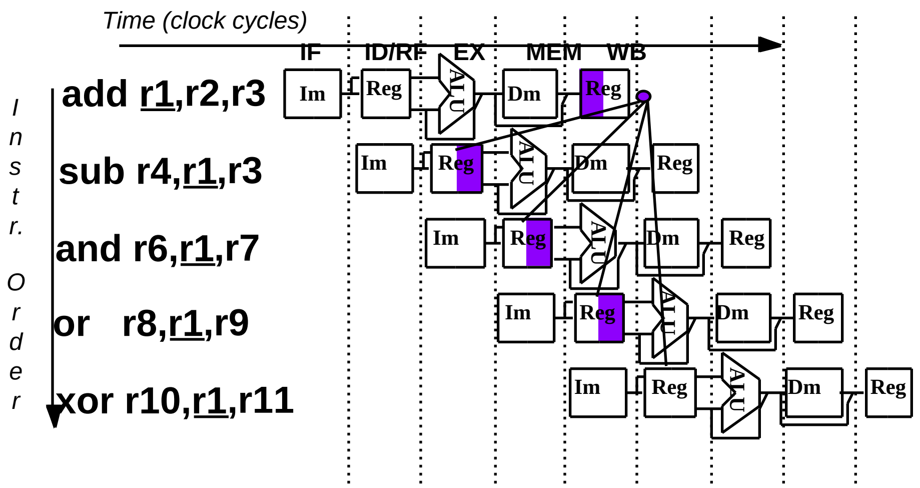
Hazards–Challenge to Pipelining

Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle

- structural hazards: HW cannot support this combination of instructions
  - earlier case of load and R-typ like a structural hazard, but normally cannot fix by retiming instruction.
- data hazards: instruction depends on result of prior instruction still in the pipeline
- control hazards: pipelining of branches & other instructionsCommon solution is to stall the later part of the pipeline until the hazard pipeline

Data Hazard on r1

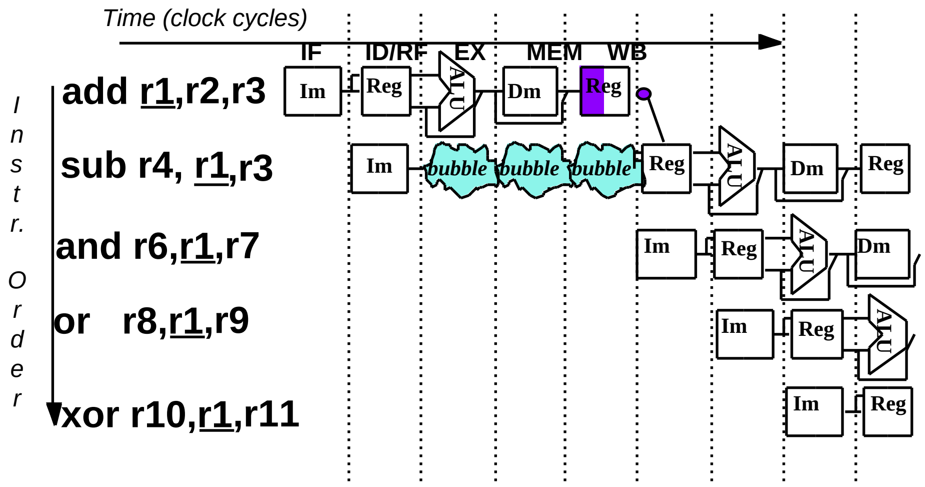
Dependencies backwards in time are hazards



HW Stalls to Resolve Hazard

Dependencies backwards in time are hazards

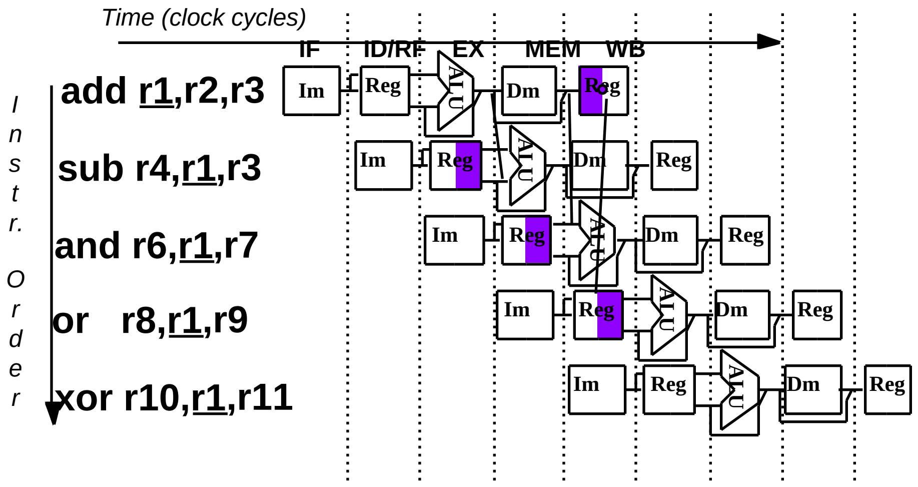
- **eliminate "reverse time" by a stall**



Insight: Data is available!

Pipeline registers already contain needed data

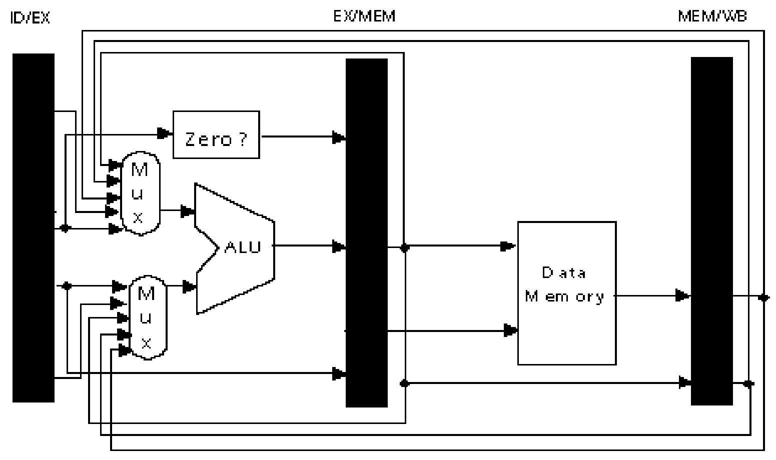
- **"Forward" the data to the appropriate unit**



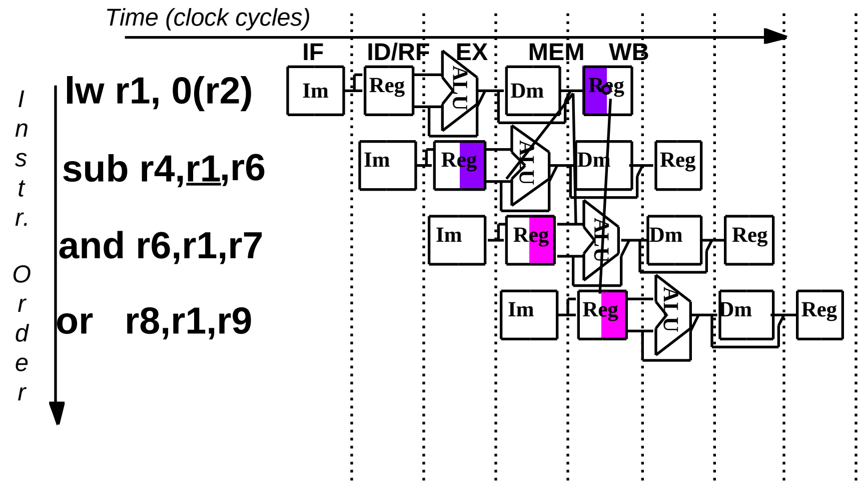
HW for "Forwarding" (Bypassing)

Increase multiplexors to add paths from registers

- **Assumes register read during write gets new value (otherwise more results to be forwarded)**

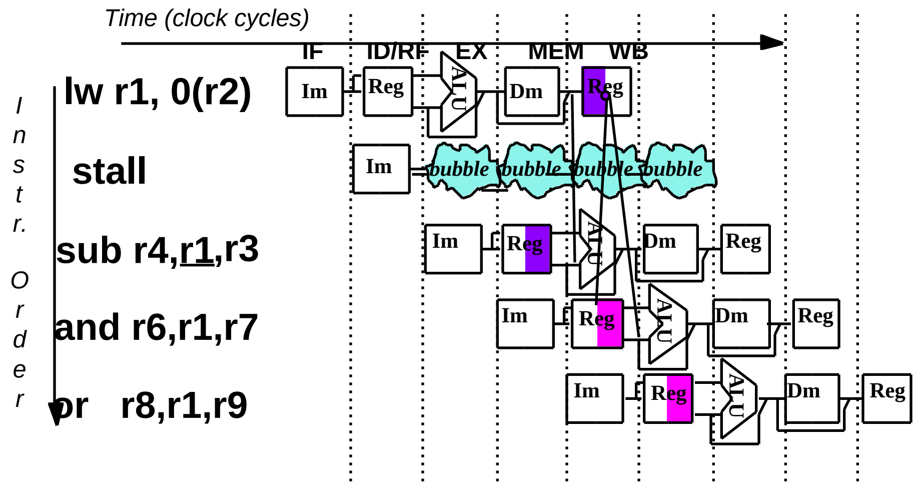


Forwarding Cannot Hide All Hazards



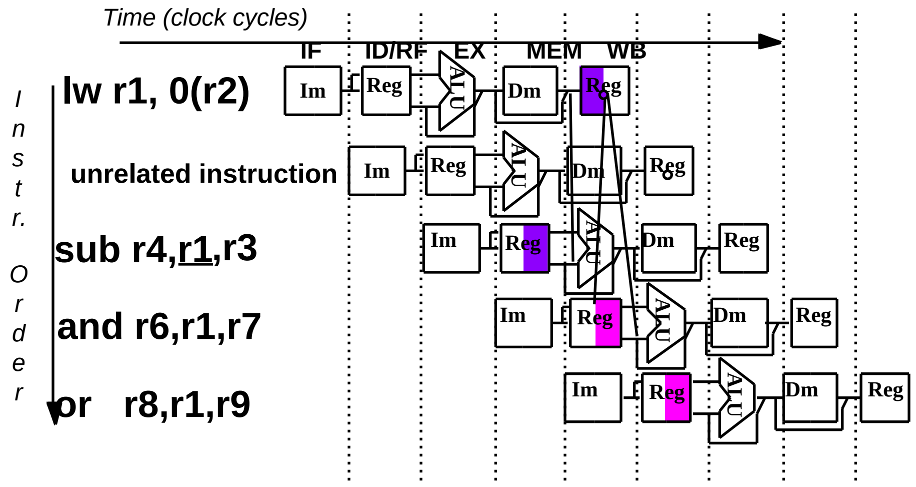
Option: HW Stalls to Resolve Hazard

"Interlock": checks for hazard & stalls

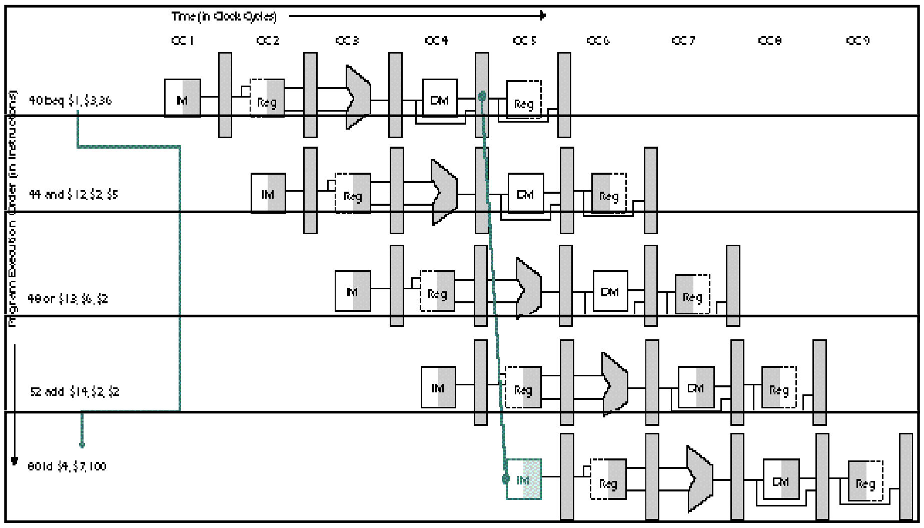


Option: SW resolves hazard

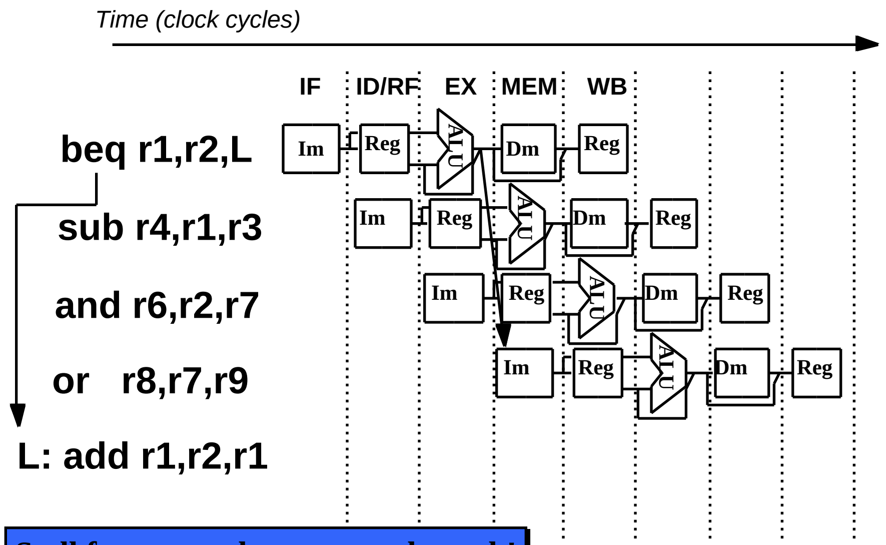
- **SW inserts independent instuctions**
  - **Worst case: performance no better/worse**



Control Hazard on Branches



Hazards on Branches



Stall for two cycles on every branch!

Branch Stall Impact

CPI Impact:

- If CPI = 1, 30% branch, Stall 2 cycles => new CPI = 1.6!

Reducing the branch penalty

- MIPS branch already more aggressive than most
- limited eq/neq allows us to determine branch condition early (after EX), rather than later (e.g., after MEM)
- doing better
  - use separate comparator rather than ALU and move branch decision to RF (hard!!!)
  - reduces penalty to 1 cycle

Going further

- Variety of techniques:
  - separating branch and destination
  - separating branch condition and branch decision
  - hardware prediction of branche

When is pipelining hard?

Interrupts: 5 instructions executing in 5 stage pipeline

- How to stop the pipeline?
- Restrart?
- Who caused the interrupt?

Stage	*Problem interrupts occurring*
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic interrupt
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation

Load with data page fault, Add with instruction page fault?

Solution 1: interrupt vector/instruction, restart everything incomplete

First Generation RISC Pipelines

All instructions: 1 pipeline order (“static schedule”).

Register write in last stage + reads performed in first stage after issue.

- Simpliy/eliminate hazards

Memory access in stage 4

- Avoid all memory hazards

Control hazards use delayed branch (with fast path)

RAW hazards use bypass, except on load results

- Load resolved by delayed load or stall

Good pipeline performance at little cost/complexity.

Summary of Pipelining Basics

Speed Up = Pipeline Depth

Hazards limit performance on computers:

- structural: need more HW resources
- data: need forwarding, compiler scheduling
- control: early evaluation & PC, delayed branch, prediction

Increasing length of pipe increases hazards

- since pipelining helps instruction bandwidth, not latency

Compilers can reduce cost of data & control hazards

- load delay slots
- branch delay slots

Exceptions (also FP, ISA) make pipelining harder

Advanced Pipelining

Pipelining exploits parallelism among instructions by overlapping them

- Called Instruction Level Parallelism (ILP)
- Limited by a variety of things:
  - parallelism in the program
  - compiler technology in exposing parallelism
  - functional unit capability: how many ovrlapping instructions
  - ability of hardware to find instructions to run in parallel

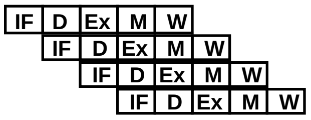
Exploiting ILP is “hot topic” in processor design:

- Lots of different approaches
  - Multiple instuctions/cycle
    - compiler vs. HW for scheduling instructions
  - Both architecture approaches and compiler approaches

Exploiting Available ILP

Technique

Pipelining

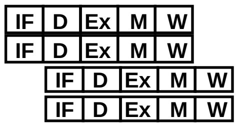


HW Limitation

Issue rate,
FU stalls,
FU depth

Super-scalar

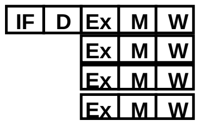
Issue multiple scalar instructions per cycle



Hazard resolution

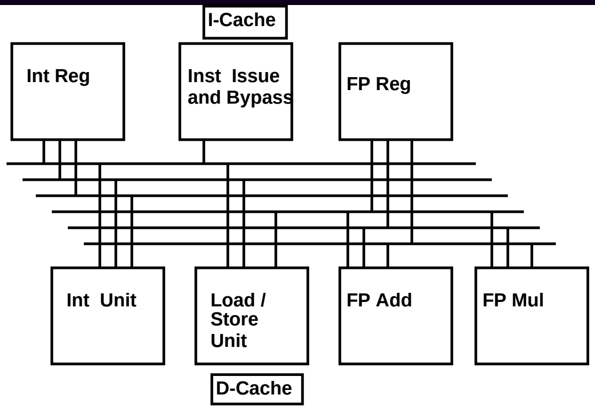
VLIW

Each instruction specifies multiple scalar operations



Packing

Easy Superscalar



Issue integer and FP operations in parallel!

- potential hazards?
- expected speedup?
- what combinations of instructions make sense?

Issuing Multiple Instruction/ Cycle

Superscalar: 2 instructions, 1 FP & 1 anything else

- Fetch 64-bits/clock cycle; Int on left, FP on right
- Can only issue 2nd instruction if 1st instruction issues
- More ports for FP registers to do FP load & FP op in a pair

Type	***Pipe***	***Stages***				
Int. instruction	IF	ID	EX	MEM	W	
FP instruction	IF	ID	EX	MEM	WB	
Int. instruction		IF	ID	EX	MEM	WB
FP instruction		IF	ID	EX	MEM	WB
Int. instruction			IF	ID	EX	MEM	WB
FP instruction			IF	ID	EX	MEM	WB

1 cycle load delay expands to 3 instruction in SS

- instruction in right half can't use result, nor can either instruction in next slot

Dynamic Branch Prediction

Predict direction of branches on past behavior

- keep a cache of branch behavior, look up prediction

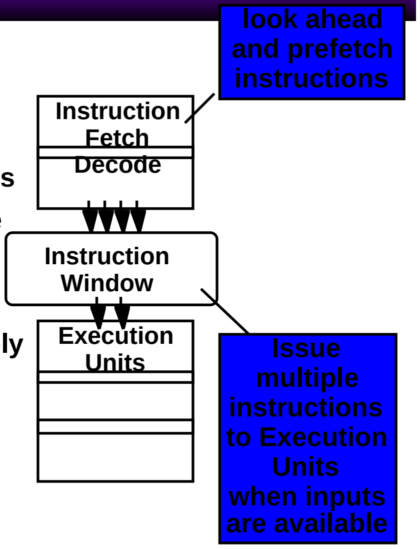
Performance = f(accuracy, cost of misprediction)

Branch prediction buffer:

- lower bits of PC address index table of 1-bit values
- says whether or not branch taken last time
- evaluate actual banch condition, if prediction incorrect:
  - recover by flushing pipeline, restarting fetch
  - reset prediction

Speculative Superscalar Execution

- Get all available parallelism
  - across branches
  - in face cache misses
  - limited only by data dependences
- Goal: resources and available bandwidth are only HW limit
- Branch prediction
  - execute instructions speculatively
- Hazard detection and aggressive resolution
  - out-of-order execution (dynamic scheduling)
  - in-order completion
    - Exception handling easier
    - handles incorrect speculation


look ahead and prefetch instructions
Instruction Fetch
Decode
Instruction Window
Execution Units
Issue multiple instructions to Execution Units when inputs are available

Variety of Modern Microprocessor

Processor	Instruction Completion Rate	Scheduling of pipeline	Branch prediction
PowerPC 604	4	Dynamic, nonspeculative	HW
MIPS R10000	4	Dynamic, speculative	HW
Pentium II	4	Dynamic, nonspeculative	HW
UltraSPARC	4	Static	HW
Merced	?	Static?	Static?

Limits to Multi-Issue Machines

Inherent limitations of ILP

- 1 branch in 5 => 5-way VLIW busy?
- Latencies of units => many operations must be scheduled
- Need about Pipeline Depth x No. Functional Units of independentDifficulties in building HW
- Duplicate FUs to get parallel execution
- Increase ports to Register File (3 x integer/FP rate)
- Increase ports to memory
- Decoding challenge and impact on clock rate, pipeline depth

Limitations specific to either SS or VLIW implementation

- Decode issue in SS
- VLIW code size: unroll loops + wasted fields in VLIW
- VLIW lock step => 1 hazard & all instructions stall
- VLIW & binary compatibility

Summary

Instruction Level Parallelism in SW or HW

Loop level parallelism is easiest to see

SW dependencies/Compiler sophistication determine if compiler can unroll loops

SW Scheduling

HW scheduling

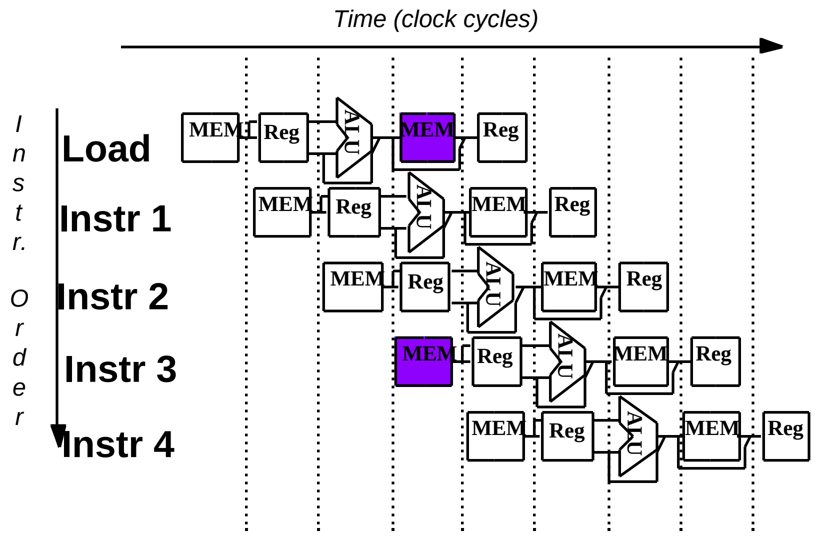
Branch Prediction

SuperScalar and VLIW

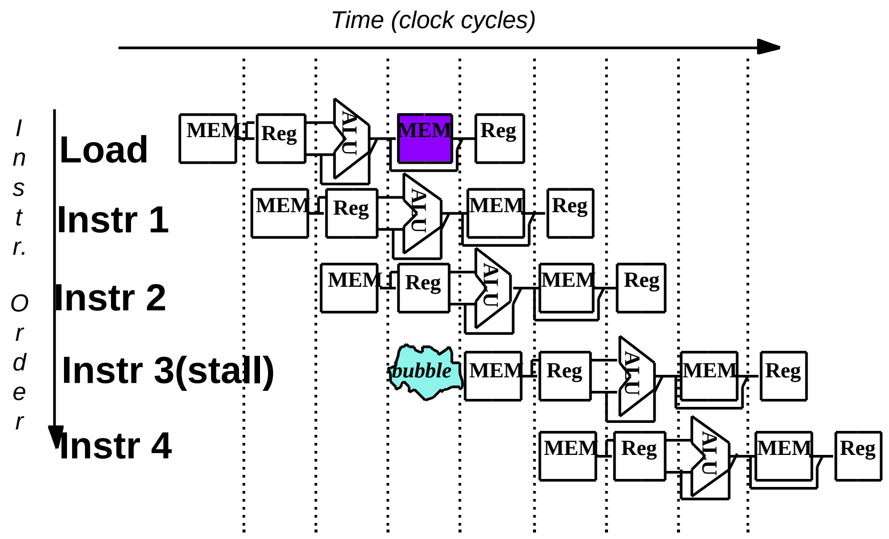
- CPI < 1
- Dynamic issue vs. Static issue
- More instructions issue/clock, larger penalty of hazards

Future? Stay tuned…

Single Memory=>Structural Hazard

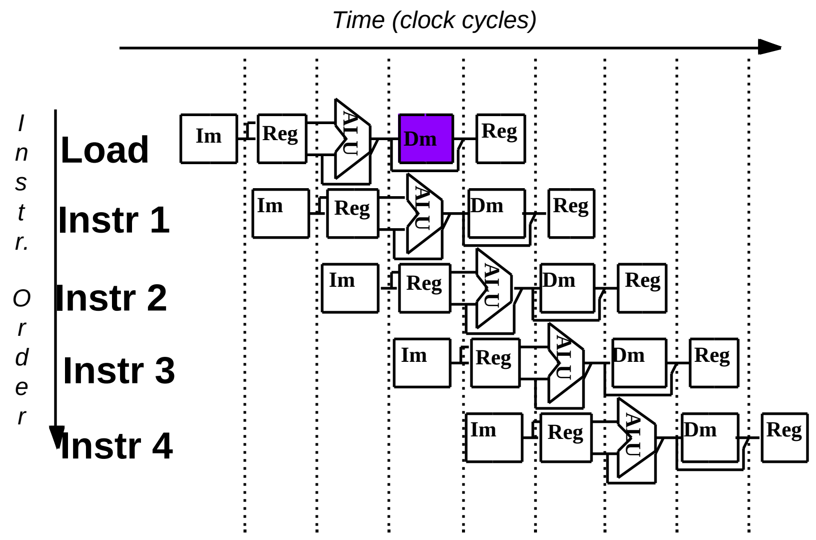


Stall to resolve Structural Hazard



Duplicate to Resolve Hazard

Separate Instruction Cache (Im) & Data Cache (Dm)



Time (clock cycles)
I n s t r. O r d e r
Load
Instr 1
Instr 2
Instr 3
Instr 4
Im
Reg
ALU
Dm
Reg