CG2028 Lecture 6: Pipelining Basics

Rajesh Panicker, NUS

CG2028

Acknowledgement:

- Text by Harris and Harris and companion materials
- Text by Patterson & Hennessy and companion materials by Mary Jane Irwin, PSU



Instruction Critical Paths

- What is the clock cycle time assuming negligible delays for muxes, control unit, extend, PC read/write, wires, register setup times & propagation delays except:
 - Instruction and Data Memory (200 ps)
 - ALU (120 ps)
 - Adders (75 ps)
 - Register File access (reads 100 ps, writes 60 ps)

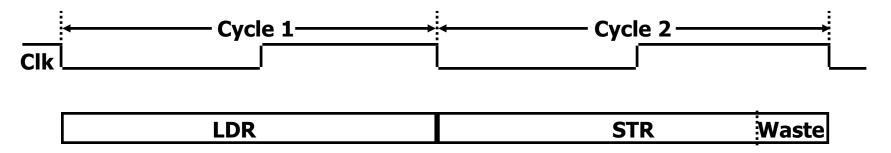
Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
DP	200	100	120		60	480
LDR	200	100	120	200	60	680
STR	200	100	120	200		620
В	200		120			320

Note: Addition for PC+4 is done in parallel with Imem access and takes even less time. So adders are not in the critical path.



Single Cycle Disadvantages

- Uses the clock cycle inefficiently the clock cycle must be timed to accommodate the slowest instruction
 - especially problematic for more complex instructions like floating point multiply



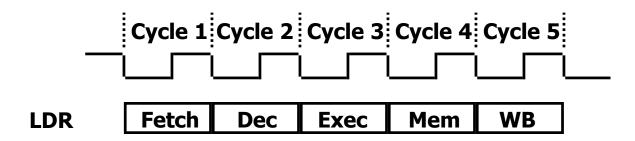
 May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle



How Can We Make It Faster?

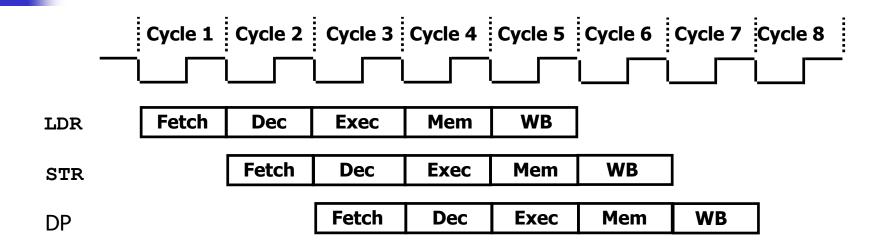
- Start the next instruction before the current one has completed
 - Pipelining most modern processors are pipelined for performance
- Under *ideal* conditions and with a large number of instructions, the speedup from pipelining is approximately equal to the number of pipe stages
 - A five stage pipeline is nearly five times faster because the clock period is nearly one fifth
 - Improves throughput total amount of work done in a given time
 - Does NOT reduce instruction latency time from the start of an instruction to its completion

The Five Stages of Load Instruction



- Fetch: Instruction Fetch and Update PC
- Decode: Registers Fetch and Instruction Decode
- Execute: Execute DP-type; calculate memory address
- Memory: Read/write the data from/to the Data Memory
- WriteBack: Write the result data into the register file

A Pipelined Processor



- Pipeline rate limited by the slowest pipeline stage
 - Some stages don't need the whole clock cycle (e.g., WB)
 - Unbalanced pipe stages makes for inefficiencies
- The time to "fill" pipeline and time to "drain" can impact speedup for deep pipelines and short code runs
- For some instructions, some stages are wasted cycles (i.e., nothing is done during that cycle for that instruction)



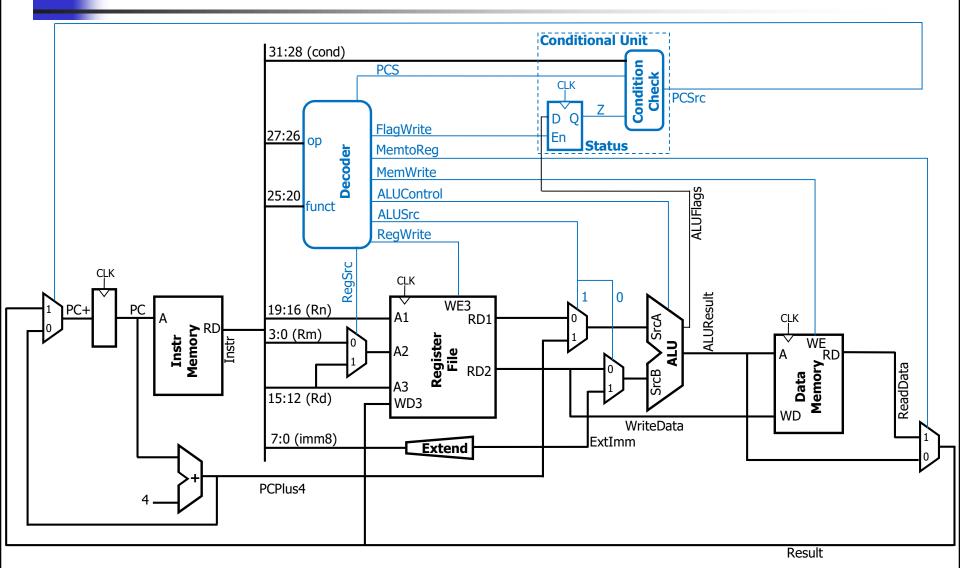
Instruction Critical Paths

- What is the clock cycle time assuming negligible delays for muxes, control unit, extend, PC read/write, wires, register setup times & propagation delays except:
 - Instruction and Data Memory (200 ps)
 - ALU (120 ps)
 - Adders (75 ps)
 - Register File access (reads 100 ps, writes 60 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
DP	200	100	120		60	480
LDR	200	100	120	200	60	680
STR	200	100	120	200		620
В	200		120			395

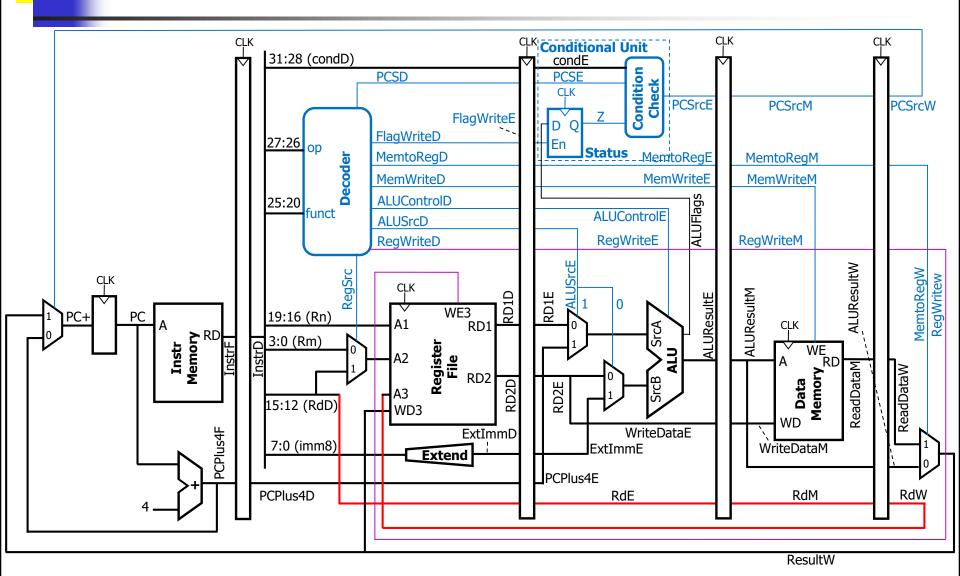
Note: Addition for PC+4 is done in parallel with Imem access and takes even less time.

Single-Cycle Processor



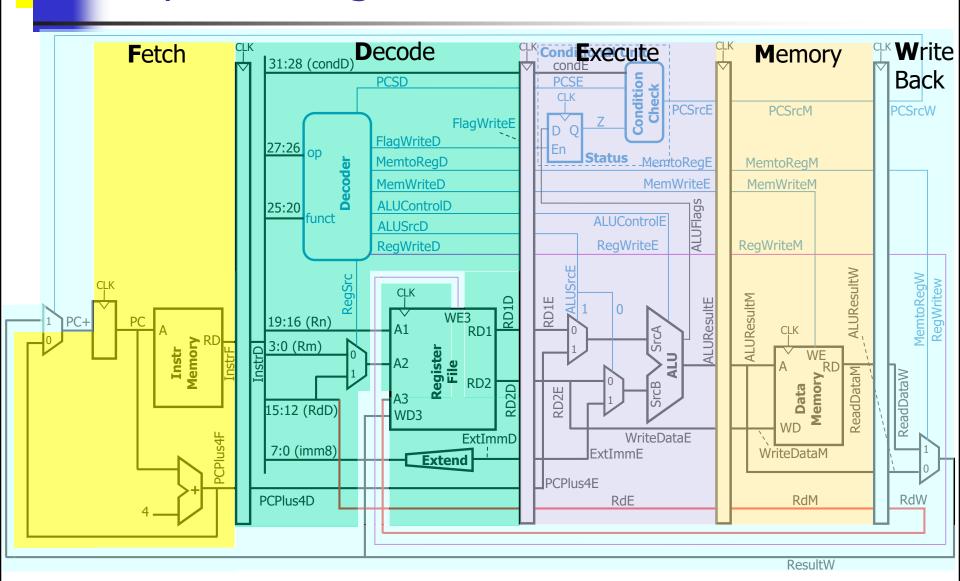
 T_C (and hence, performance) limited by critical path (LDR)

Pipelined Processor



Registers between each pipeline stage to isolate them and reduce the critical path

Pipeline Stages



Pipelining

- Any info that is needed in the next stage must be passed to that stage via a pipeline register (i.e., whenever data crosses a clock edge, it needs to be stored in a register)
- All the information related to an instruction should move together, i.e., should be delayed equally

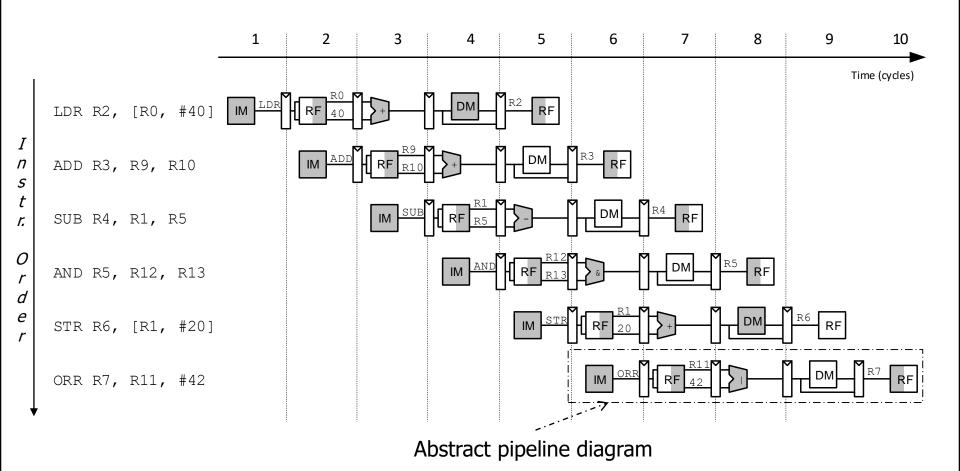
A3 (RdW) must arrive at same time as Result
(ResultW)

- SUB R1, R2, R3 in D
- ADD R4, R5, R6 in W will be ADD R1, R5, R6 in W effectively if Rd is not delayed!
- RegWrite should also be delayed
- PCSrcW must arrive at same time as ResultW (which is the BTA for a branch instruction)
- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage

PC	Instruction	Stage
0x00	ADD R4, R5, R6	W
0x04	Instruction 2	М
80x0	Instruction 3	Е
0x0C	SUB R1, R2, R3	D
0x10	Instruction 5	F

BTA = Branch Target Address

Why Pipeline? For Performance!



Once the pipeline is full, (ideally) one instruction is completed every cycle

Can Pipelining Get Us Into Trouble?

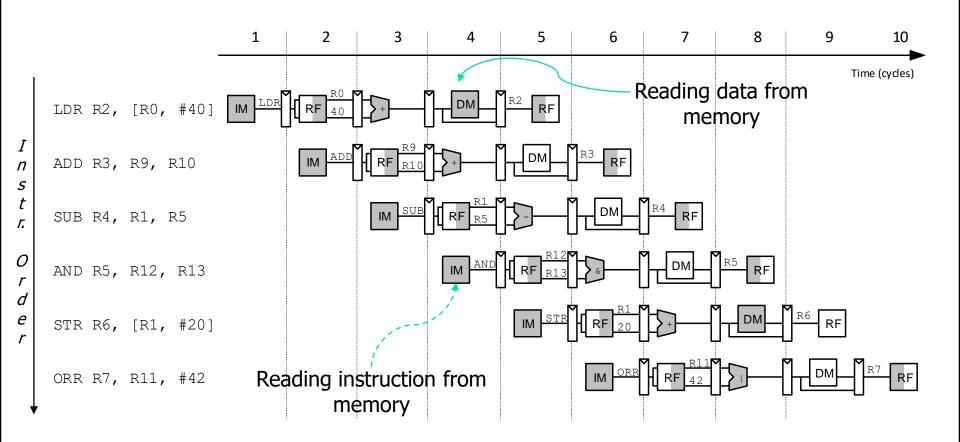
Yes: Pipeline Hazards

- structural hazards: attempt to use the same resource by two different instructions at the same time
- data hazards: attempt to use data before it is ready
 - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
- control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
 - Branch instructions, writes to R15, interrupts/exceptions

Can usually resolve hazards by waiting

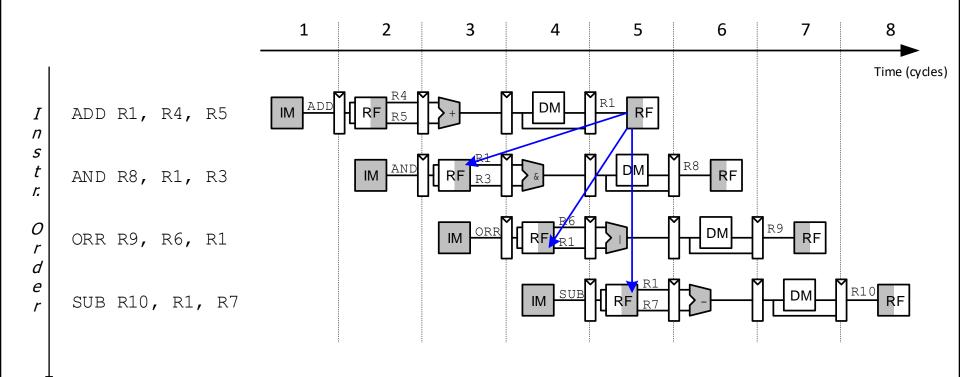
- Compiler (software) or pipeline control (hardware) must detect the hazard
- and take action to resolve hazards

A Single Memory Would Be a Structural Hazard



- Fix with separate instr and data memories (IM and DM)
 - Or at least separate caches

Data Hazard



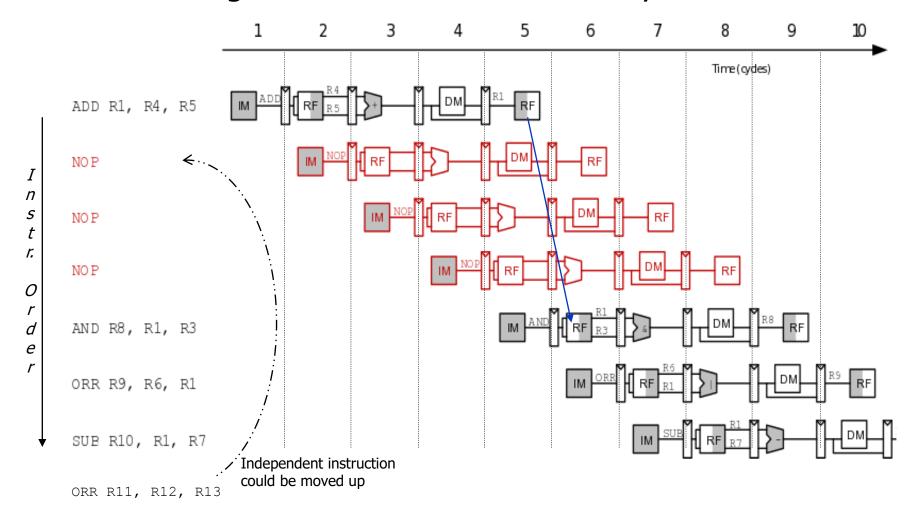
- Also known as RAW (read after write) hazard or true data dependency
- Occurs very frequently in practice -> represents the flow of information in the program

Handling Data Hazards

- Insert NOPs (MOV RO, RO) in code at compile time
 - Insert enough NOPs for result to be ready
 - NOPs waste time => No useful work is done
 - NOPs waste code memory => makes the code bulkier / bloated
 - Compiler needs to know the microarchitecture to know the number of NOPs => code is not very portable
- Rearrange code at compile time
 - Move independent useful instructions forward
 - Might not be possible all the time
- Stall the processor at run time
 - Impacts performance, needs additional hardware
- Forward data at run time
 - Good performance, needs even more hardware

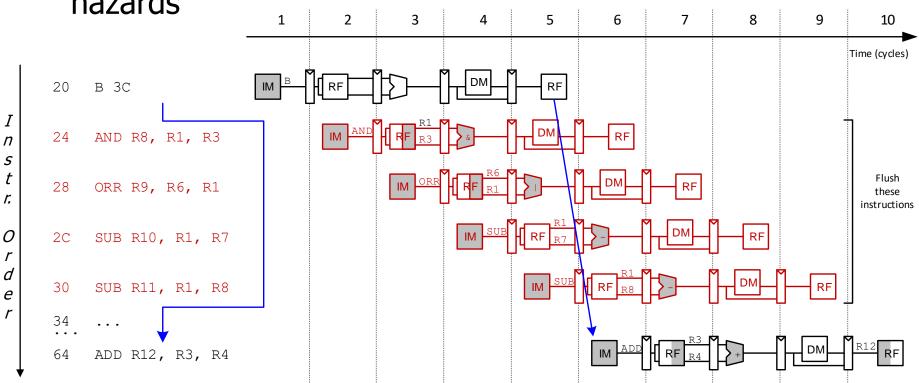
Data Hazard: Inserting NOPs

Insert enough NOPs for result to be ready



Control Hazards

Branch instructions, writes to R15 can cause control hazards



- Compile time NOPs (4 NOPs after the branch in this case) can help, but
 - NOPs reduce performance, even when the branch is not taken
 - Increases the code size and makes the code microarchitecture dependent

Handling Control Hazards

- Possible approaches (all need additional hardware)
 - Stall until branch decision and BTA are available (performance ↓)
 - Early BTA move decision point as early in the pipeline as possible (rather than in WB stage), thereby reducing the number of stall cycles
 - Branch prediction (a form of speculative execution)
 - Guess the branch outcome as well as the BTA
 - Start execution from BTA if branch is predicted to be taken, else next instr.
 - If prediction is correct, no penalty associated with branches!
 - If prediction is wrong, need to restart the pipeline and discard the results from the speculatively executed instructions
 - Predicated/conditional execution (IT statements can help eliminate some branches)
 - Fine grained multithreading,...,...
- Control hazards occur less frequently than data hazards, but are (much) harder to deal with than data hazards

How Can We Make It *Even* Faster?

- Key to performance enhancement : Exploit Parallelism
- One option : start (issue) more than one instruction per clock
 multiple issue processors
 - Needs duplication of many / most functional units such as ALU
 - Need to ensure that the instructions are independent done by the compiler at compile time or hardware at runtime
 - Exploits Instruction Level Parallelism (ILP)
- Static multiple issue processors a.k.a VLIW (Very Long Instruction Word)
 - Compiler creates bundles of independent instructions
 - Eg: Many Digital Signal Processors (DSPs), Intel Itanium
- Dynamic multiple issue processors a.k.a Superscalar
 - Independence check done by the hardware at runtime
 - Eg : Most modern processors

^{*}Note: a simple pipelined processor issues only 1 instruction per clock



Summary

- Almost all modern day processors use pipelining
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Potential speedup: once the pipeline is full, one instruction completes every cycle, at a higher clock frequency
- Pipeline rate limited by slowest pipeline stage
- Must detect and resolve hazards at compile time (by the compiler) or runtime (by the hardware)
- More on pipelining, speculative execution, other types of parallelisms, multiple issue processors, multicore / multiprocessor systems etc. in CG3207