Instruction Execution
&
Pipelining

Instruction Execution

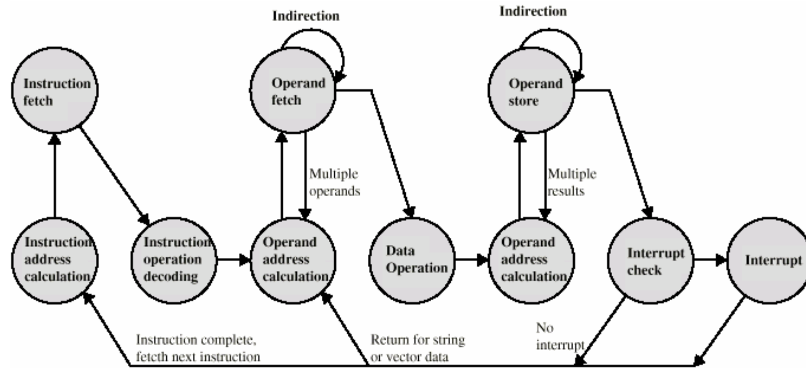
Instruction Execution

- Simple fetch-decode-execute cycle:
  1. Get address of next instruction from PC
  2. Fetch next instruction into IR
  3. Change PC
  4. Determine instruction type (add, shift, …)
  4. If instruction has operand in memory, fetch it into a register
  5. Execute instruction storing result
  6. Go to step 1

Indirect Cycle

- Instruction execution may require memory access to fetch operands
- Memory fetch may use indirect addressing
  - Actual address to be fetched is in memory
  - Requires more memory accesses
  - Can be thought of as additional instruction subcycle

Basic Instruction Cycle States

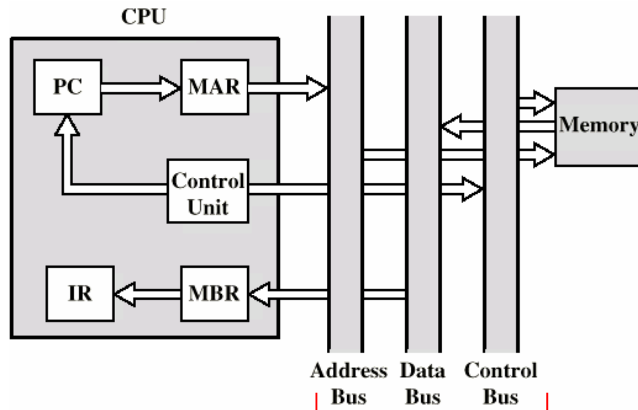


Data Flow (Instruction Fetch)

- Depends on CPU design; below typical
- Fetch (Step 2)
  - PC contains address of next instruction
  - Address moved to MAR
  - Address placed on address bus
  - Control unit requests memory read
  - Result placed on data bus, copied to MBR, then to IR
  - Meanwhile PC incremented by 1 instruction word
    - Typically byte addressable, so PC incremented by 4 for 32-bit address ISA

Data Flow (Instruction Fetch)

3-Bus Architecture



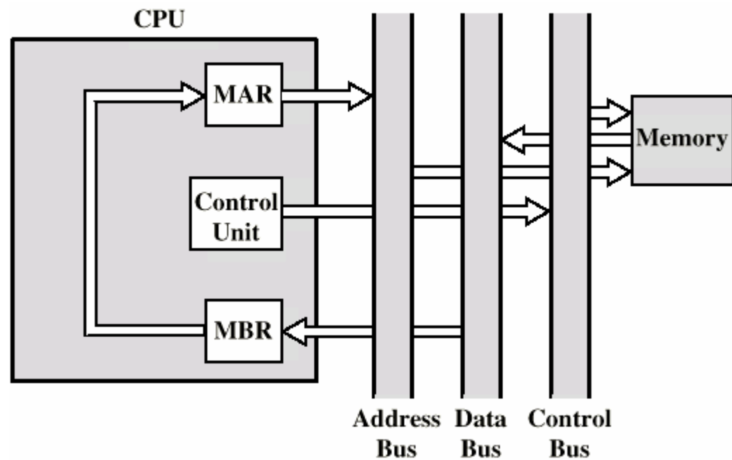
MBR = Memory buffer register
MAR = Memory address register
IR = Instruction register
PC = Program counter

Combined in 1-Bus Architecture

Data Flow (Data Fetch)

- IR is examined
- If indirect addressing, indirect cycle is performed
  - Right most bits of MBR transferred to MAR
  - Control unit requests memory read
  - Result (address of operand) moved to MBR

Data Flow (Indirect Diagram)



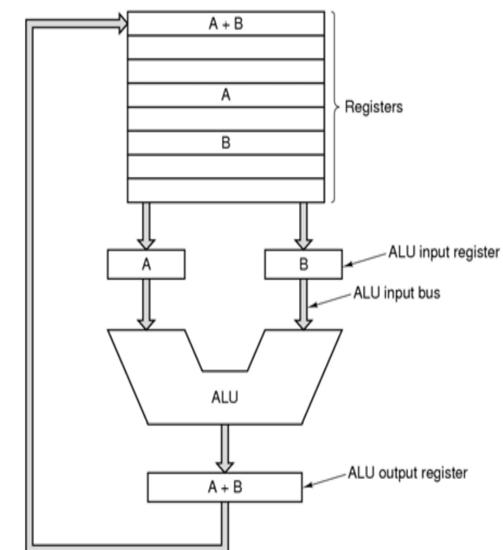
Data Flow (Execute)

- May take many forms
- Depends on instruction being executed
- May include
  - Memory read/write
  - Input/Output
  - Register transfers
  - ALU operations

Data Path [1]

- Data path cycle: two operands going from registers, through ALU and results stored (the faster the better).
- Data path cycle time: time to accomplish above
- Width of data path is a major determining factor in performance

Data Path [2]



RISC and CISC Based Processors

- RISC (Reduced Instruction Set Computer)
  vs
  CISC (Complex Instruction Set Computer)
  - "War" started in the late 1970's
  - RISC:
    - Simple interpret step so each instruction faster
    - Issues instructions more quickly (faster)
  - Basic concept:
    If RISC needs 5 instructions to do what 1 CISC instruction does but 1 CISC instruction takes 10 times longer than 1 RISC, then RISC is faster

RISC vs. CISC Debate

- CISC & RISC both improved during 1980's
  - CISC driven primarily by technology
  - RISC driven by technology & architecture
- RISC has basically **won**
  - Improved by over 50% / year
  - Took advantage of new technology
  - Incorporated new architectural features (e.g., Instruction Level Parallelism (ISL), superscalar concepts)
- Compilers have become **critical** component in performance

More on RISC & CISC

- RISC-CISC debate largely irrelevant now
  - Concepts are similar
    - Heavy reliance on compiler technology
  - Modern RISC has "complicated" ISA
  - CISC leader (Intel x86 family) now has large RISC component

Prefetching

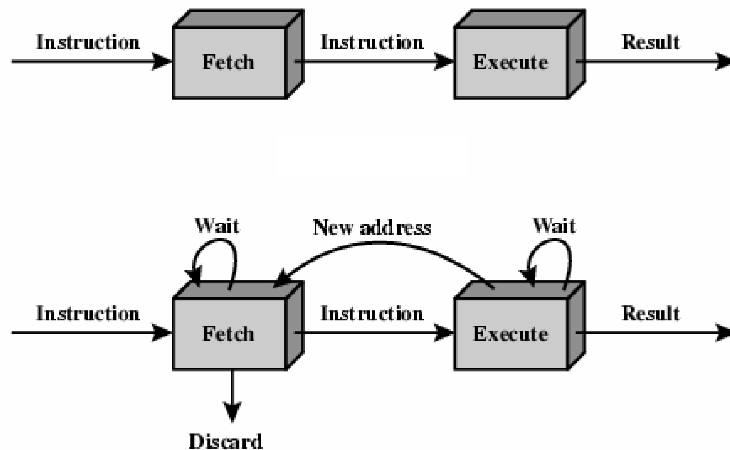
Prefetch

- Fetch accesses main memory
- Execution usually does not access main memory
- Can fetch next instruction during execution of current instruction
- Called instruction prefetch

Improved Performance

- But not doubled:
  - Fetch usually shorter than complete execution of one instruction
    - Prefetch more than one instruction?
  - If fetch takes longer than execute, then cache (will discuss later) saves the day!
  - Any jump or branch means that prefetched instructions are not the required instructions

Prefetching, Waiting & Branching



Pipelining

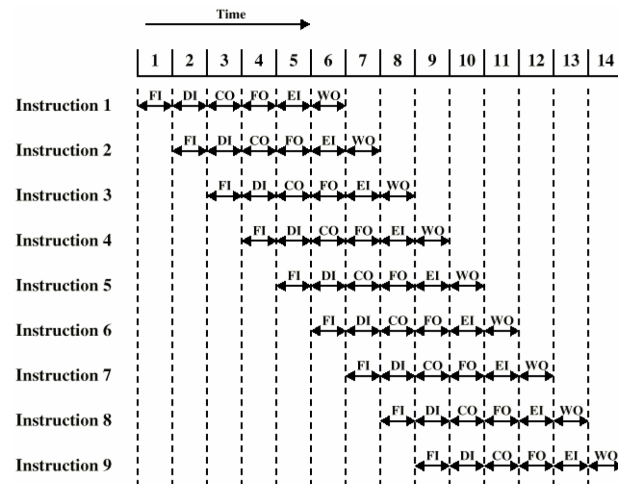
Pipelining

- Add more stages to improve performance
- For example, 6 stages:
  - Fetch instruction (FI) — Memory access
  - Decode instruction (DI)
  - Calculate operands (CO)
  - Fetch operands (FO) — ? Memory access ?
  - Execute instruction (EI)
  - Write operand (result) (WO) — ? Memory access ?

  Overlap these stages

Again, cache saves the day!

Timing of Pipeline



Important points

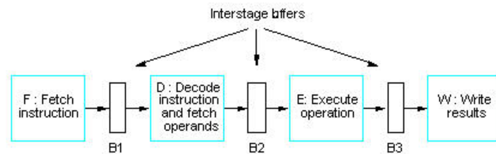
- An instruction pipeline passes instructions through a series of stages with each performing some part of the instruction.
- If the speed of two processors, one with a pipeline and one without, are the same, the pipeline architecture will not improve the overall time required to execute one instruction.
- If the speed of two processors, one with a pipeline and one without, are the same, the pipelined architecture has a higher throughput (number of instructions processed per second).

Execution Time

- Assume that a pipelined instruction processor has 4 stages, and the maximum time required in the stages are 10, 11, 10 and 12 nanoseconds, respectively.
- How long will the processor take to perform each stage? **12 nanoseconds**
- How long does it take for one instruction to complete execution? **48 nanoseconds**
- How long will it take for ten instruction to complete execution? **(48 + 9*12) nanoseconds**
  156 nanoseconds
  15.6 nanoseconds/instru

4-Stage Pipeline

- Another example (with interstage buffers):
  - Fetch instruction (F) — Memory access
  - Decode instruction and fetch operands (D)
  - Execute instruction (E) — ? Memory access ?
  - Write operand (result) (W)

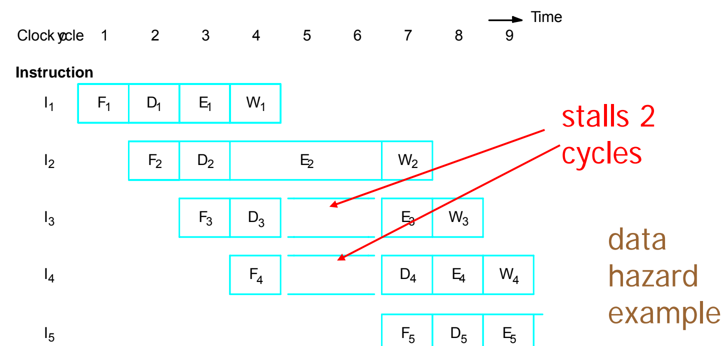


Some Potential Causes of Pipeline "Stalls"

- What might cause the pipeline to "stall"?
  - One stage takes too long (examples)
    - Different execution times (data hazard)
    - Cache miss (control hazard)
    - Memory unit busy (structural hazard)
  - Branch to a new location (control hazard)
  - Call a subroutine (control hazard)

Long Stages – Execution [1]

- For a variety of reasons, some instructions take longer to execute than others (e.g., add vs divide) $\Rightarrow$ stall



stalls 2 cycles

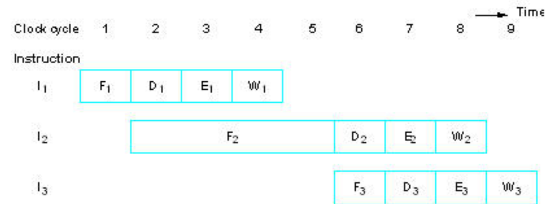
data hazard example

Long Stages – Execution [2]

- Solution:
  - Sub-divide the long execute stage into multiple stages
  - Provide multiple execution hardware so that multiple instructions can be in the execute stage at the same time
  - Produces one instruction completion every cycle after the pipeline is full

Long Stages – Cache Miss

- Delay in getting next instruction due to a cache miss (will discuss later)



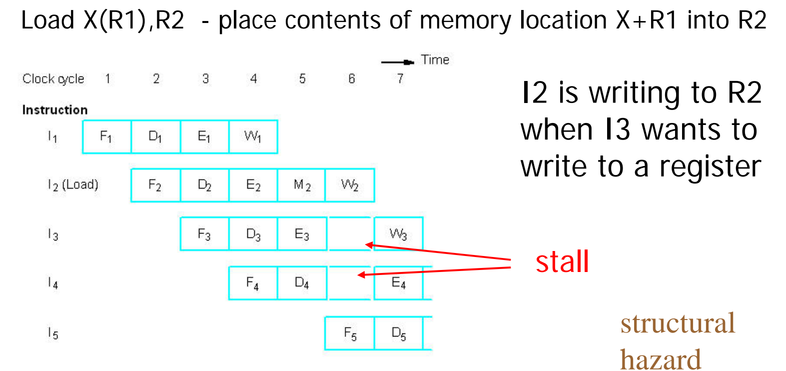
control hazard example

Decode, Execute & Write stages become idle – stall.

Long Stages – Memory Access Conflict

- Two different instructions require access to memory at the same time

Load X(R1),R2 - place contents of memory location X+R1 into R2



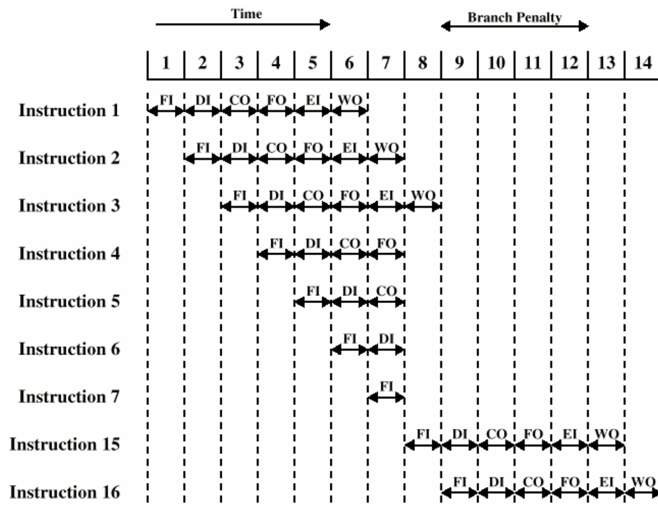
I2 is writing to R2 when I3 wants to write to a register

stall

structural hazard

Conditional Branch in a Pipeline

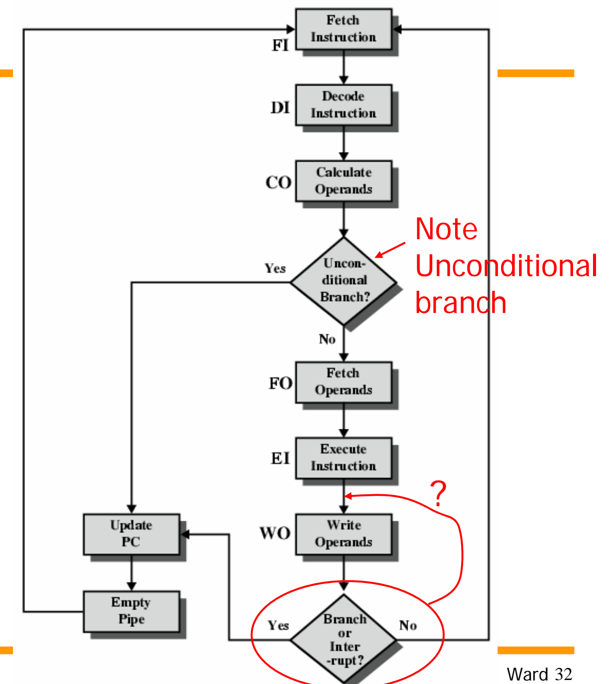
Instruction 3 is a conditional branch to Instruction 15



Important Architectural Point: When is the PC changed?

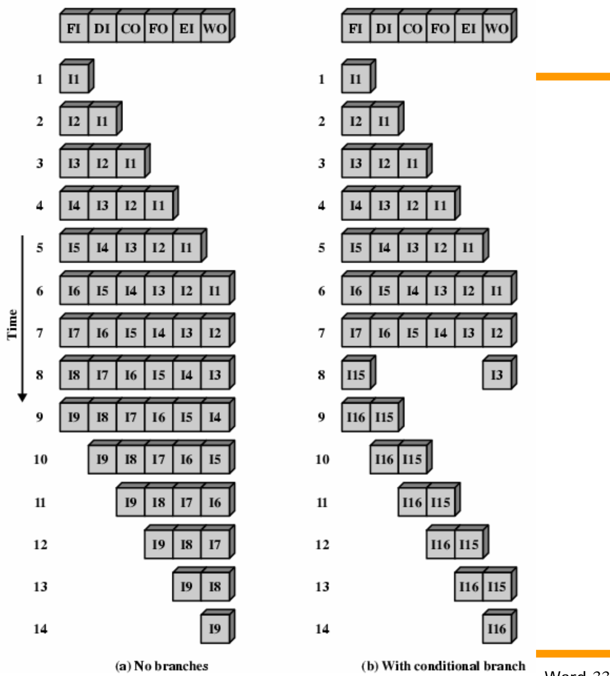
control hazard

Six Stage Instruction Pipeline Logic

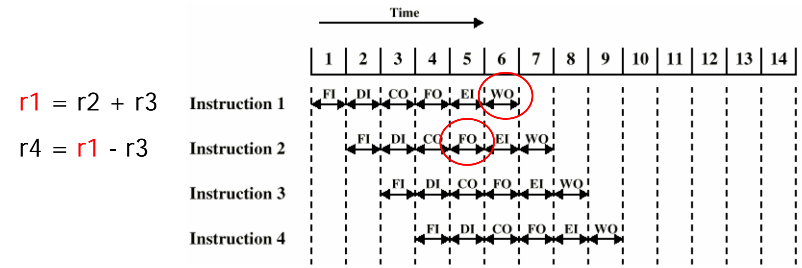


Note Unconditional branch

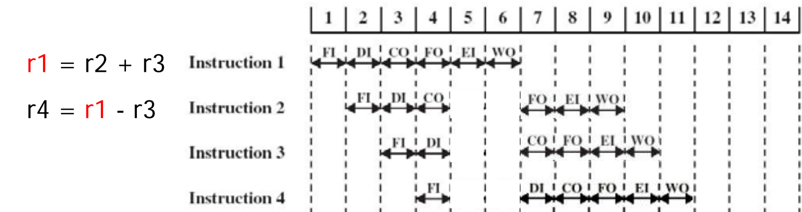
Alternative Pipeline Depiction



Data Dependencies in a Pipeline



Pipeline stalls until r1 has been written. (RAW hazard)



Example of Avoiding Stalls

(a)	(b)
C ← add A B	C ← add A B
D ← subtract E C	F ← add G H
F ← add G H	M ← add K L
J ← subtract I F	D ← subtract E C
M ← add K L	J ← subtract I F
P ← subtract M N	P ← subtract M N

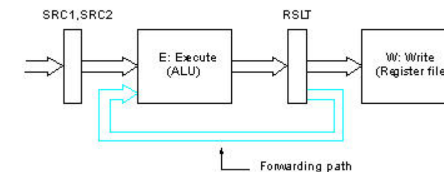
- Stalls eliminated by rearranging (a) to (b)

RAW Hazard: Forwarding

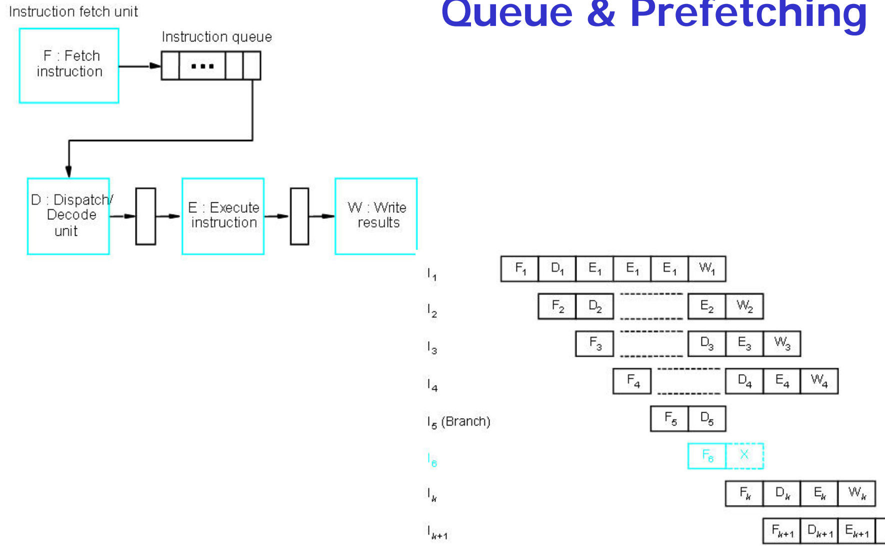
- Hardware optimization to avoid stall
- Allows ALU to reference result in next instruction
- Example:

  Instruction K: C ← add A B

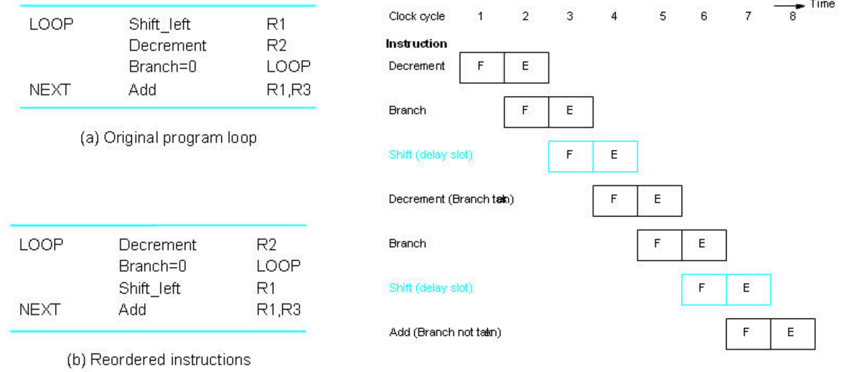
  Instruction K+1: D ← subtract E C



Unconditional Branches: Instruction Queue & Prefetching

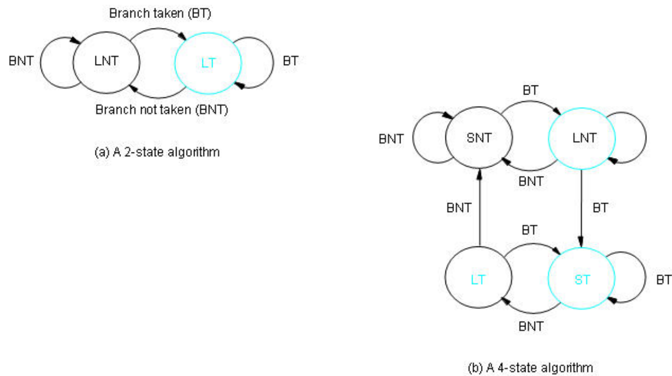


Conditional Branches: Delayed Branch

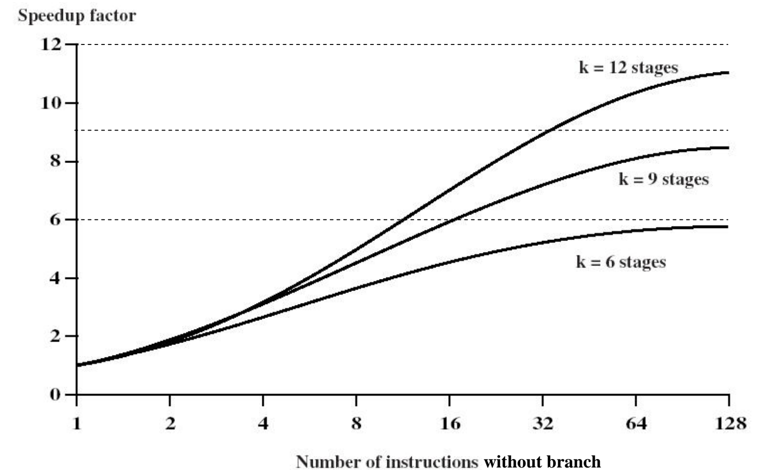


With more stages, need to reorder more instructions.

Conditional Branches: Dynamic Prediction

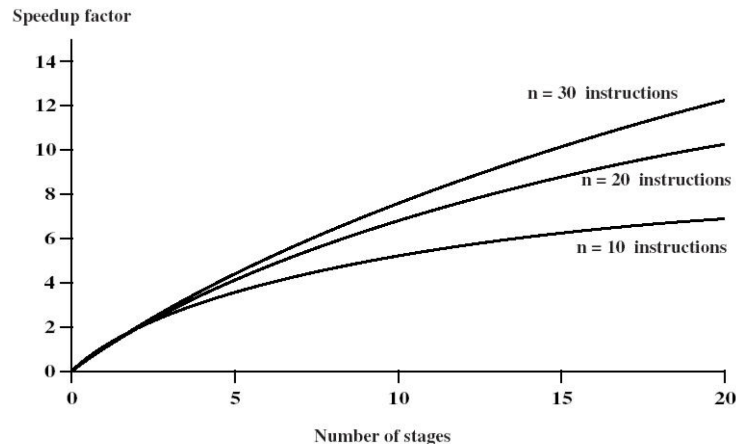


Pipelining Speedup Factors [1]



Never get speedup better than the number of stages in a pipeline

Pipelining Speedup Factors [2]



Never get speedup better than the number of instructions without a branch.

Classic 5-Stage RISC Pipeline

- Instruction Fetch (IF) — Memory access
- Instruction Decode & Register Fetch (ID)
- Execute / Effective Address (load-store) (EX)
- Memory Access (MEM) — ? Memory access ?
- Write-back (to register) (WB)

Another 5-Stage Pipeline

- Instruction Fetch (IF) — Memory access
- Instruction Decode (ID)
- Operand Fetch (OF) — ? Memory access ?
- Execute Instruction (EX)
- Write-back (WB) — ? Memory access ?

Achieving Maximum Speed

- Program must be written to accommodate instruction pipeline
- To minimize stalls, for example:
  - Avoid introducing unnecessary branches
  - Delay references to result register(s)

More About Pipelines

- Although hardware that uses an instruction pipeline will not run at full speed unless programs are written to accommodate the pipeline, a programmer can choose to ignore pipelining and assume the hardware will automatically increase speed whenever possible.
- Modern compilers are typically able to move instructions around to optimize pipeline execution for RISC architectures.

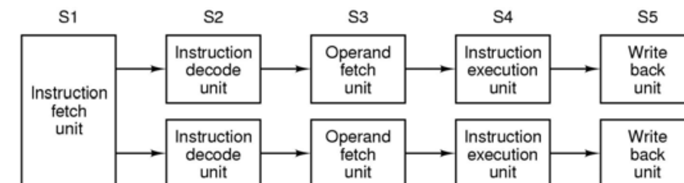
Advanced Pipeline Topics

- Structural hazards (e.g., resource conflicts)
- Data hazards (e.g., RAW, WAR, WAW)
- Control hazards (e.g., branch prediction, cache miss)
- Dynamic scheduling
- Out-of-order issue
- Out-of-order execution
- Speculative execution
- Register renaming
- . . . .

Parallelizations

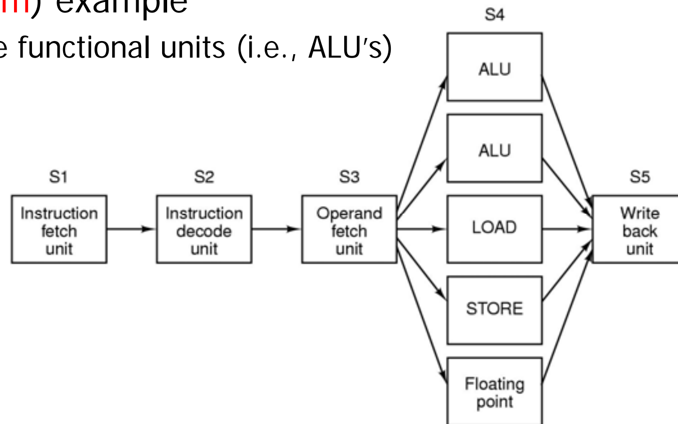
Superscalar Architecture [1]

- Superscalar architecture (parallelism inside single processor – Instruction-Level Parallelism) example
  - 2 pipelines with 2 ALU's
  - 2 instructions must not conflict over resource (e.g., register, memory access unit) and must not depend on other result



Superscalar Architecture [2]

- Superscalar architecture (parallelism inside single processor – Instruction-Level Parallelism) example
  - Multiple functional units (i.e., ALU's)



Processor-Level Parallelism

- Array processor: single control unit, large array of identical processors that perform same instructions on different data
- Vector processor: heavily pipelined ALU efficient at executing instructions on data pairs (like matrices) and vector register (set of registers that can be loaded simultaneously)
- Multiprocessors: more than one independent CPU sharing same memory
- Multicomputers: large number of interconnected computers that don't share memory but pass data on buses to share it