CPU Design and its Effects on Performance

Sarala Arunagiri

24 September 2013

Rearrangement of topics covered in the next four classes and

- Von Nuemann architecture, modern floating point units, pipelining, ILP – first lecture
- Memory hierarchies – fourth lecture
- Multicore architectures, locality and data reuse, roofline model – second lecture
- Programming strategies for performance, Power consumption – third lecture

Two Dominant Computer Architectures

1. Von Neumann Architecture and
2. Harvard Architecture

Look up the video for the differences between the two architectures

http://www.pictutorials.com/Harvard_vs_Von_Nuemann_Architecture.htm

Features of Von Neumann Architecture

Refer to the image
http://en.wikipedia.org/wiki/Von_Neumann_architecture

- Program and data share the same bus to the memory leading to Von Neuman bottleneck
- Program and data share the same address space in memory - this makes things such as just in time (JIT) compiling possible; program modifications can be quite harmful, either by accident or design; needs memory protection and access control.

Features of Harvard Architecture

Image and content from Wikipedia- http://en.wikipedia.org/wiki/Harvard_architecture

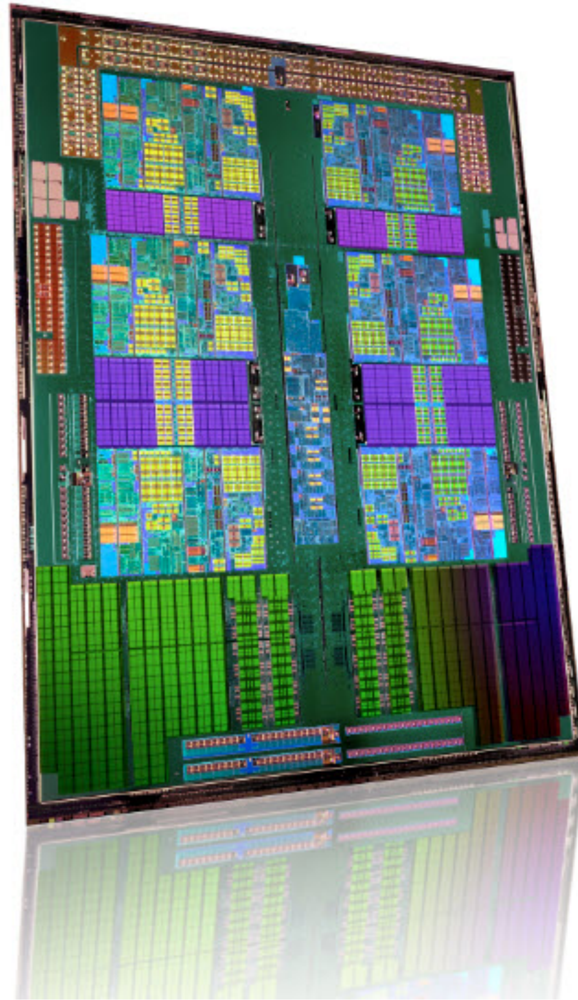
- Separate busses for program and data
- Separate address spaces for program and data

Relatively pure Harvard architecture machines are used mostly in applications where tradeoffs, like the cost and power savings from omitting caches, outweigh the programming penalties from featuring distinct code and data address spaces. Examples are embedded systems.

Modern Computer Architecture

- Has a single address space for programs and data - like Von Neumann
- Has caches and therefore access to different locations may take different times; concept of locality – unlike Von Neumann
- Processors have a separate instruction cache and data cache – unlike Von Neumann

What does a CPU look like?



Contemporary Architecture

- Multiple operations simultaneously "in flight"
- Operands can be in memory, cache, register
- Results may need to be coordinated with other processing elements
- Operations can be performed speculatively

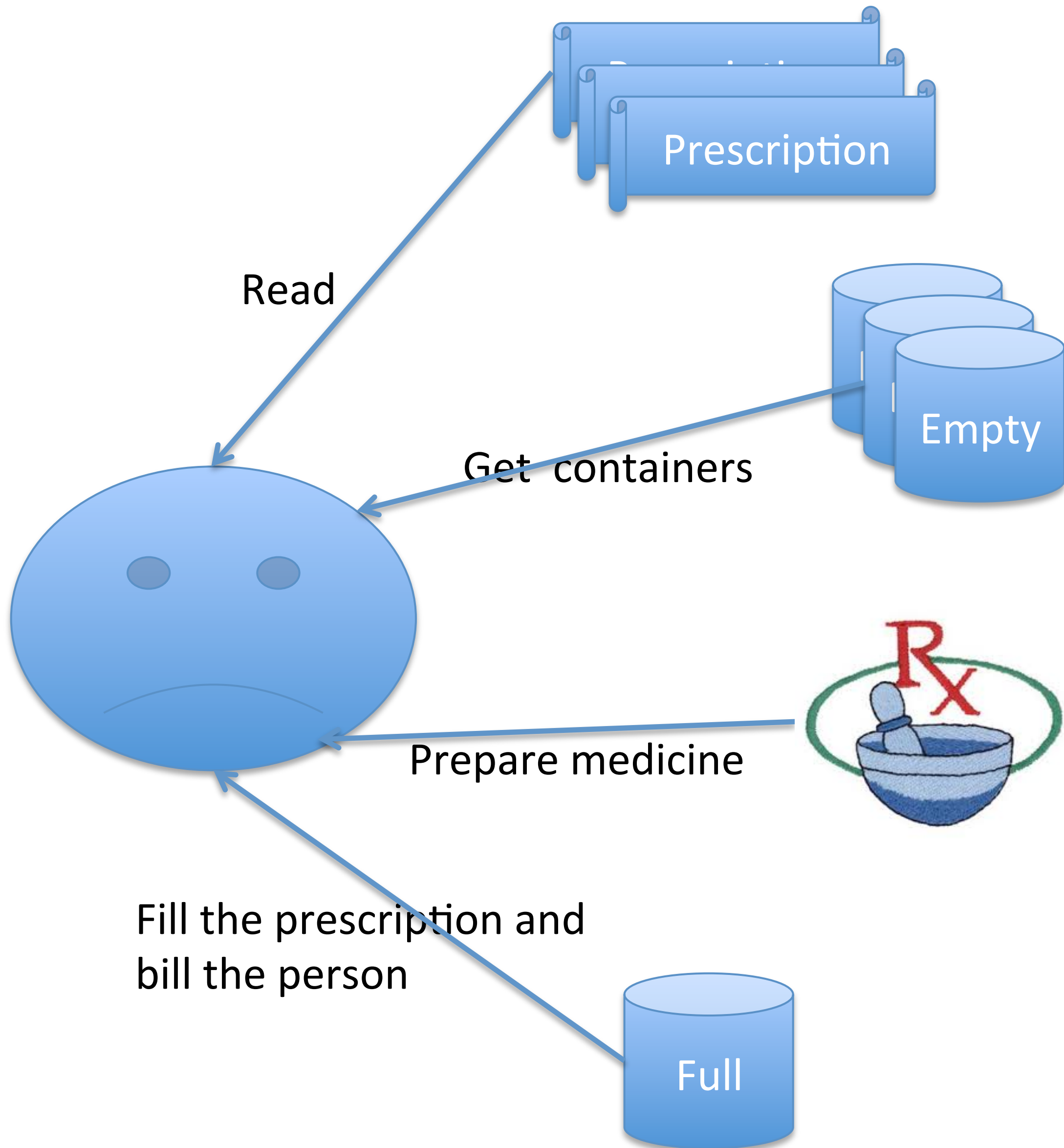
Instruction Cycle

An instruction cycle is the basic operation cycle of a computer and details of cycle depend on the instruction set. Generally consists of the following main parts:

1. Fetch,
2. Decode, and
3. Data fetched from main memory (if necessary)
4. Execute – includes writing back to registers.

This cycle is repeated continuously by the central processing unit (CPU), from boot-up to when the computer is shut down.

Analogy: Pharmacy



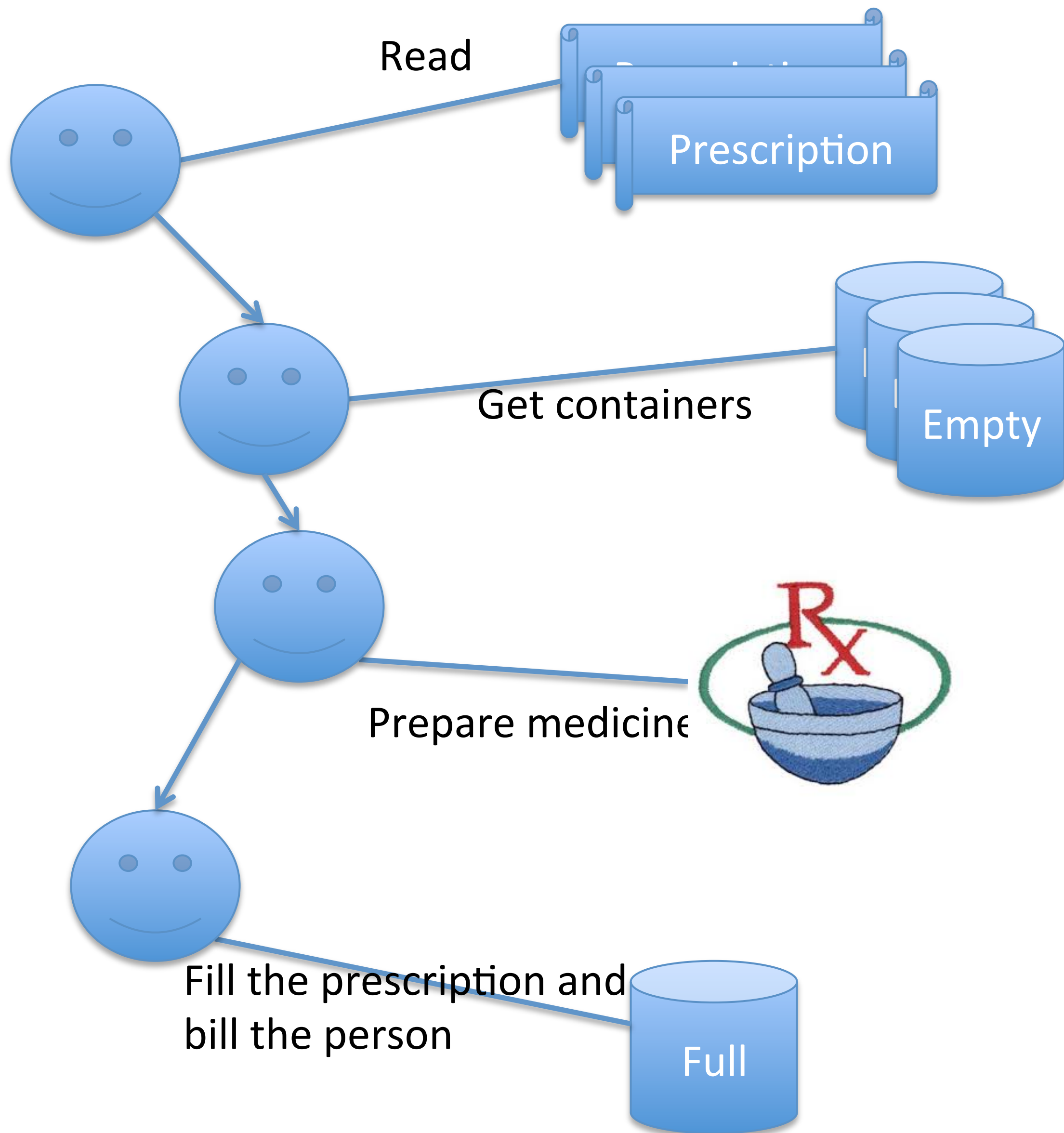
Pipelining

What is Instruction pipelining?
An instruction pipeline is a technique used in the design of computers to increase their instruction throughput.

- Each instruction is split into a sequence of dependent steps.
- Seeks to let the processor work on as many instructions as there are dependent steps and to keep every portion of the processor busy with some instruction.
- Pipelining lets the computer's cycle time be the time of the slowest step, and ideally lets one instruction complete in every cycle.

Image of five stage pipeline in wikipedia at
http://en.wikipedia.org/wiki/Instruction_pipeline

Analogy: Pharmacy Pipelining

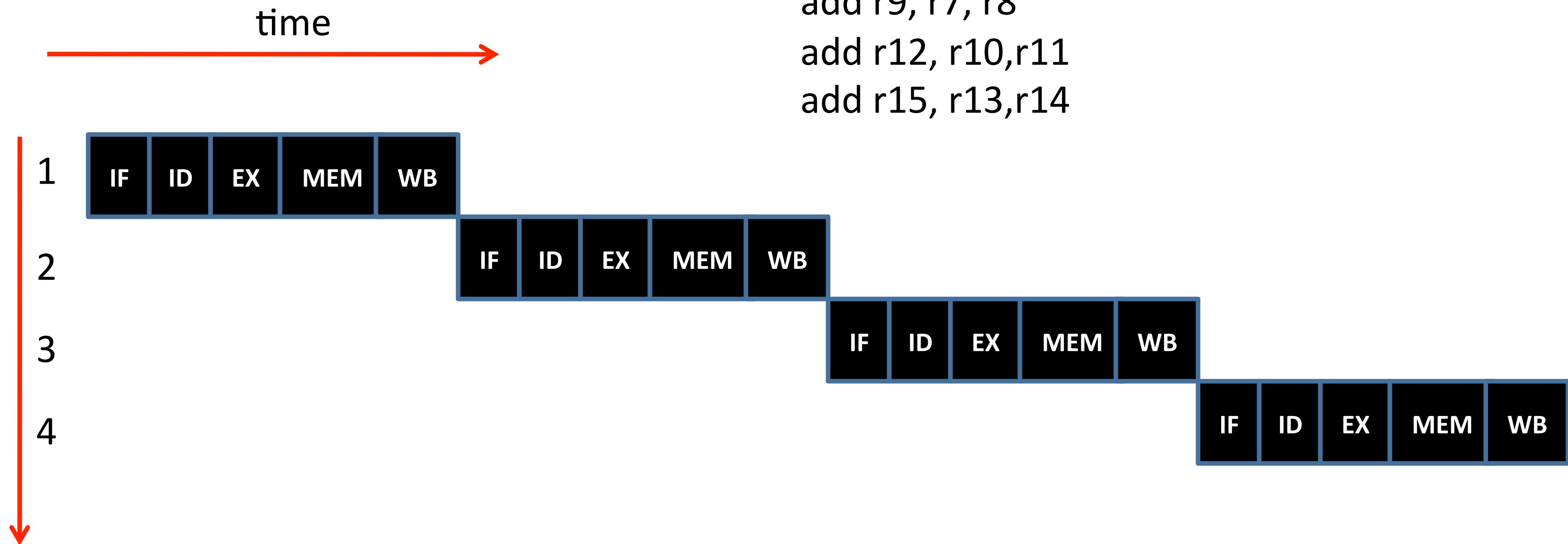


Assumptions and Notations in Timing Calculations presented in the rest of this presentation

- All illustrations use the basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back). It also computes time for 5 operations.
- All general expressions are derived assuming full pipelines and the time taken per pipeline stage is one clock cycle.
- Notations for general expressions: n= number of operations; l = number of pipeline stages; w = width of multiple issue processors (superscalar); and v = length of vector registers in SIMD or vector processors.

No Pipeline

Example sequence of five instructions:
add r3, r1, r2
add r6,r4, r5
add r9, r7, r8
add r12, r10,r11
add r15, r13,r14



instructions

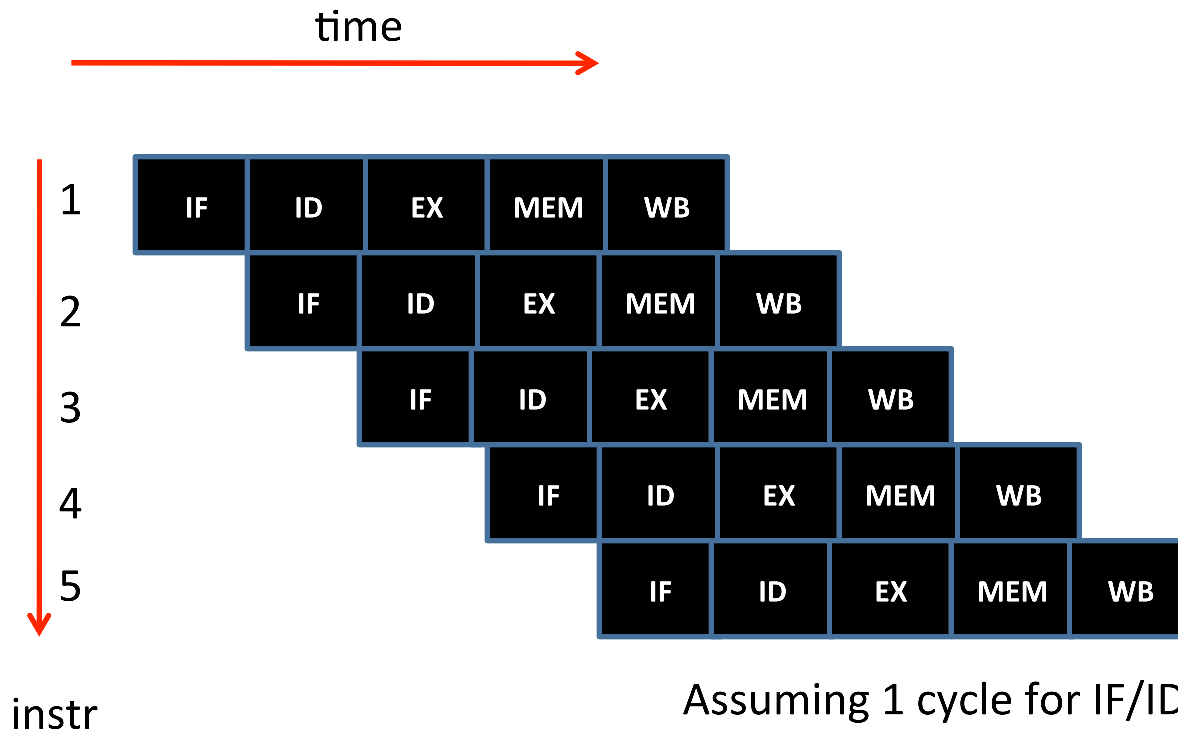
Assuming 1 cycle for IF/ID/EX/MEM/WB,

Total # cycles for n instructions: n*5
For n = 5 => (25 cycles)

In general, for a processor having l non-pipelined stages for processing each instruction and one clock cycle per stage, time taken to complete n operations is (n*l).

5-stage Pipeline

add r3, r1, r2
add r6,r4, r5
add r9, r7, r8
add r12, r10,r11
add r15, r13,r14



Assuming 1 cycle for IF/ID/EX/MEM/WB,

Total # cycles for n instructions = 5 + (n-1) = n + 4
For (n = 5) => (9 cycles)

In general, for a processor with a pipeline having l stages and one clock cycle per stage, time taken to complete n operations is $(l+n-1)$.

For pipelined Floating Point Units with l stages, there is an additional set-up time q resulting in the total time to compute n operations of $(l+q+n-1)$.

Pipeline analysis: result rate r and $n_{1/2}$

- With l segments and n operations, the time without pipelining is ln
- With pipelining it becomes
  - $l+n-1+q$ for functional units, where q is setup time.
  - l+n-1 for instruction pipelines
- With n operations, actual result rate for an instruction pipeline is $n/(l+n)$
- Result rate depends on how full the pipeline is.
- Asymptotic rate r = 1 result per clock cycle
- $n_{1/2}$ is the value of n at which half the asymptotic rate is achieved and from the expression for the result rate, $n_{1/2}$ =l

Pipelining - Hazards

Hazards are problems with the instruction pipeline in central processing unit (CPU) microarchitectures when the next instruction cannot execute in the following clock cycle, and doing so can potentially lead to incorrect computation results. There are typically three types of hazards:

- data hazards (can occur in FPU pipelines too)
- structural hazards
- control hazards (branching hazards)

Illustration: Pipelining Hazards

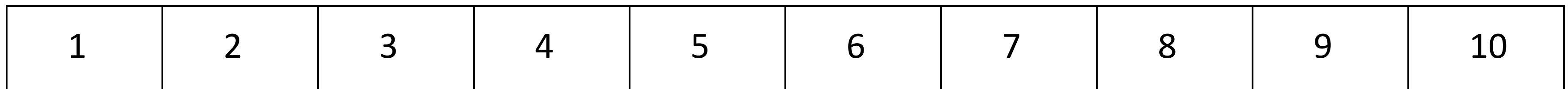
Refer to the material on wikipedia at:

http://en.wikipedia.org/wiki/Instruction_pipeline#Hazards

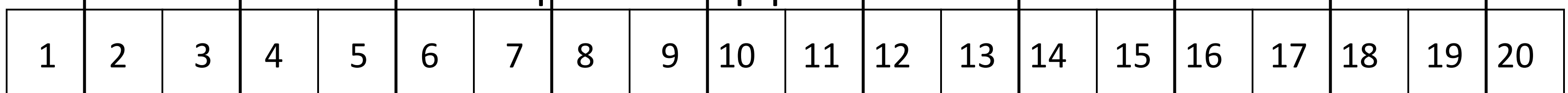
Branch Prediction

- The “instruction pipeline” is all of the processing steps (also called segments) that an instruction must pass through to be “executed”.
- Higher frequency machines have a larger number of segments.
- Branches are points in the instruction stream where the execution may jump to another location, instead of executing the next instruction.
- For repeated branch points (within loops), instead of waiting for the loop to branch route outcome, it is predicted.

Pentium III processor pipeline



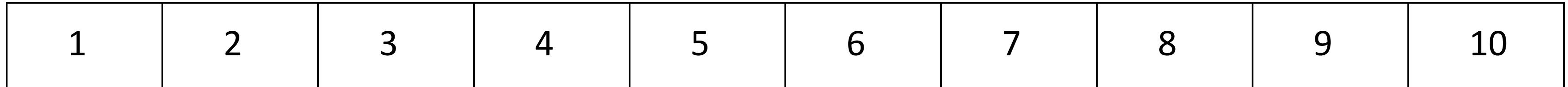
Pentium 4 processor pipeline



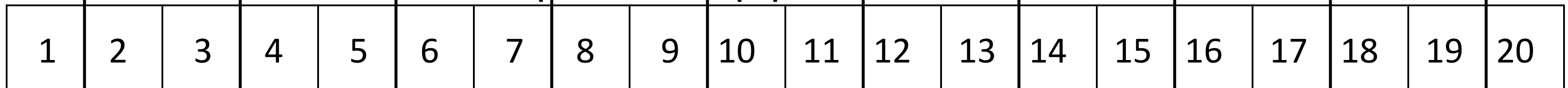
Misprediction is more “expensive” on Pentium 4’s.

Implications of Deeper Pipelines

Pentium III processor pipeline



Pentium 4 processor pipeline

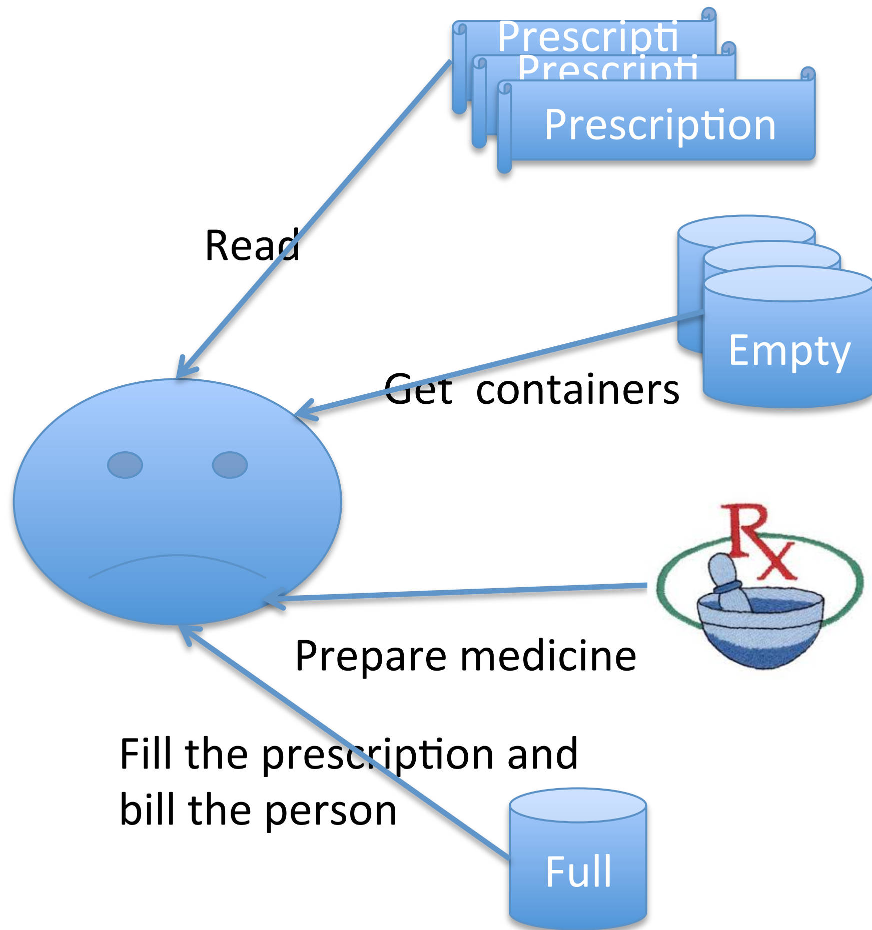


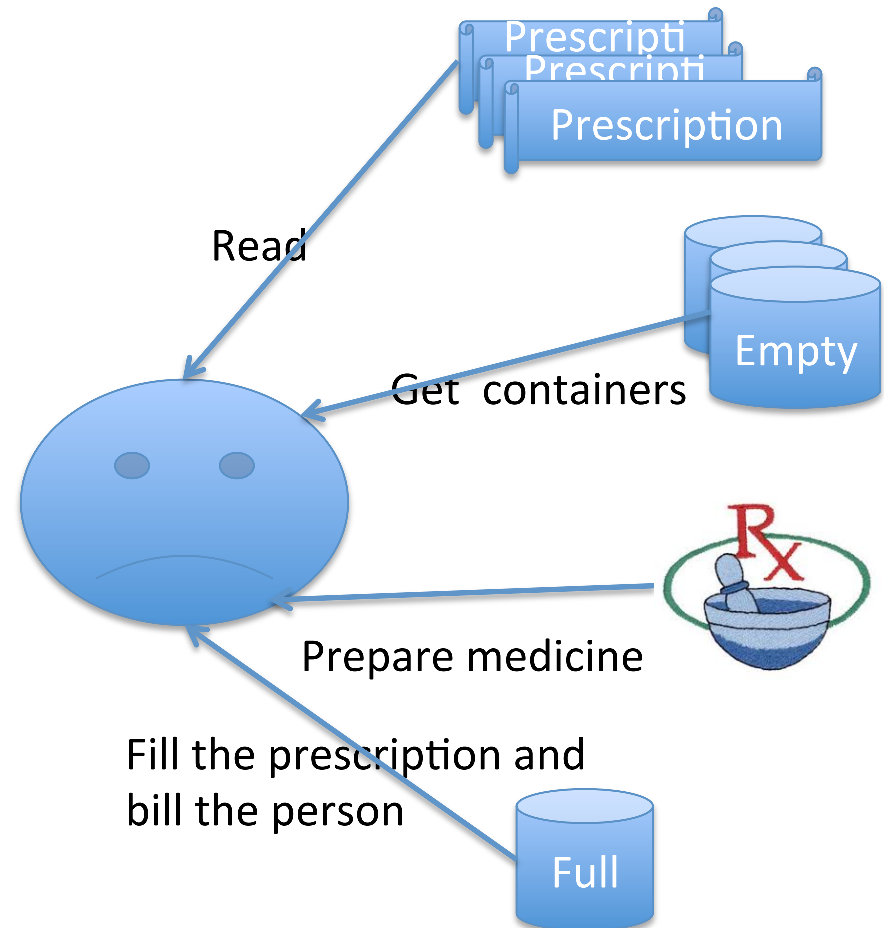
Note that the theoretical peak performance of a computer in terms of FLOPS is computed Assuming the asymptotic result rate, r.

Higher frequency => Deeper pipelines => larger the $n_{1/2}$, the number of instructions with no hazards to achieve half the asymptotic result rate.

If the compiler cannot discover sufficient ILP, we will need to shift to explicit parallelism in order to achieve high performance and the target flop rate.

Analogy: Pharmacy
Superscalar without pipelining

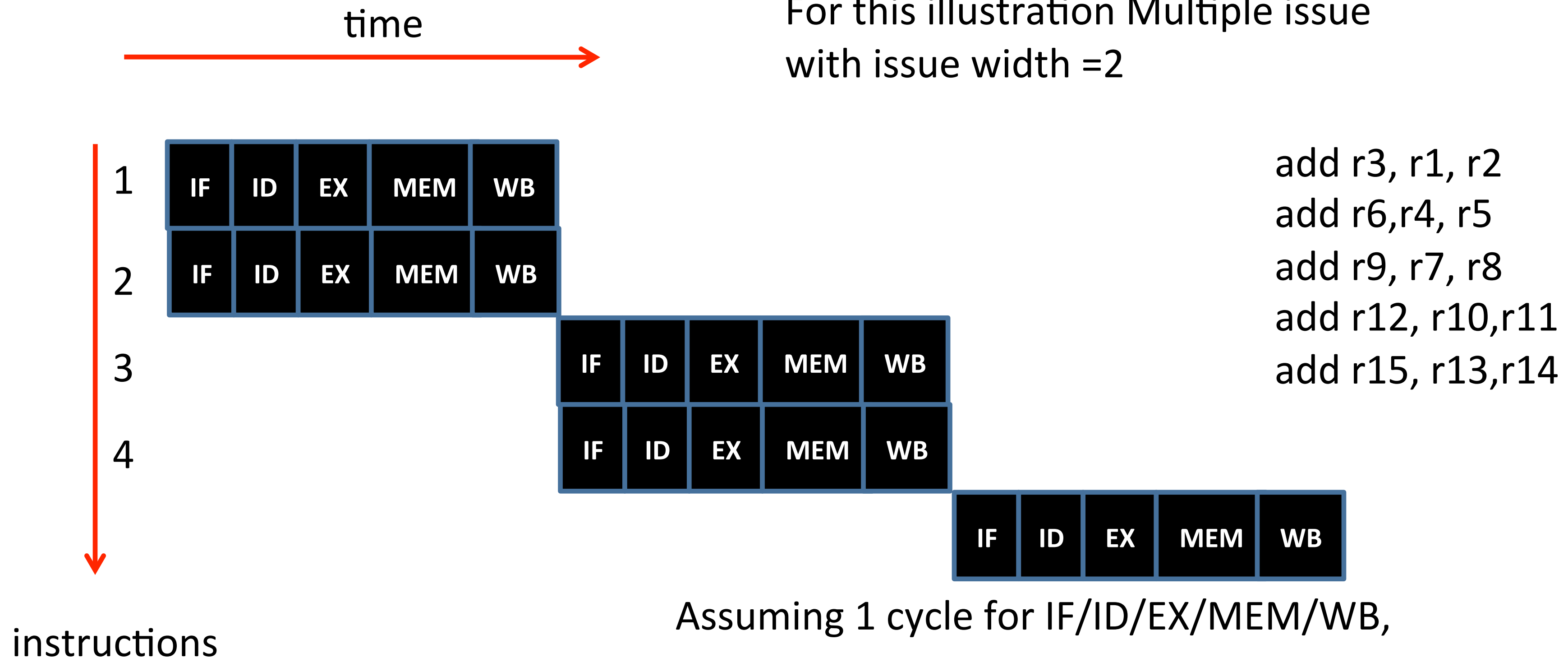




Superscalar with No Pipeline

Superscalar with no pipeline
might just be theoretical concept.

For this illustration Multiple issue
with issue width =2



add r3, r1, r2
add r6,r4, r5
add r9, r7, r8
add r12, r10,r11
add r15, r13,r14

Assuming 1 cycle for IF/ID/EX/MEM/WB,

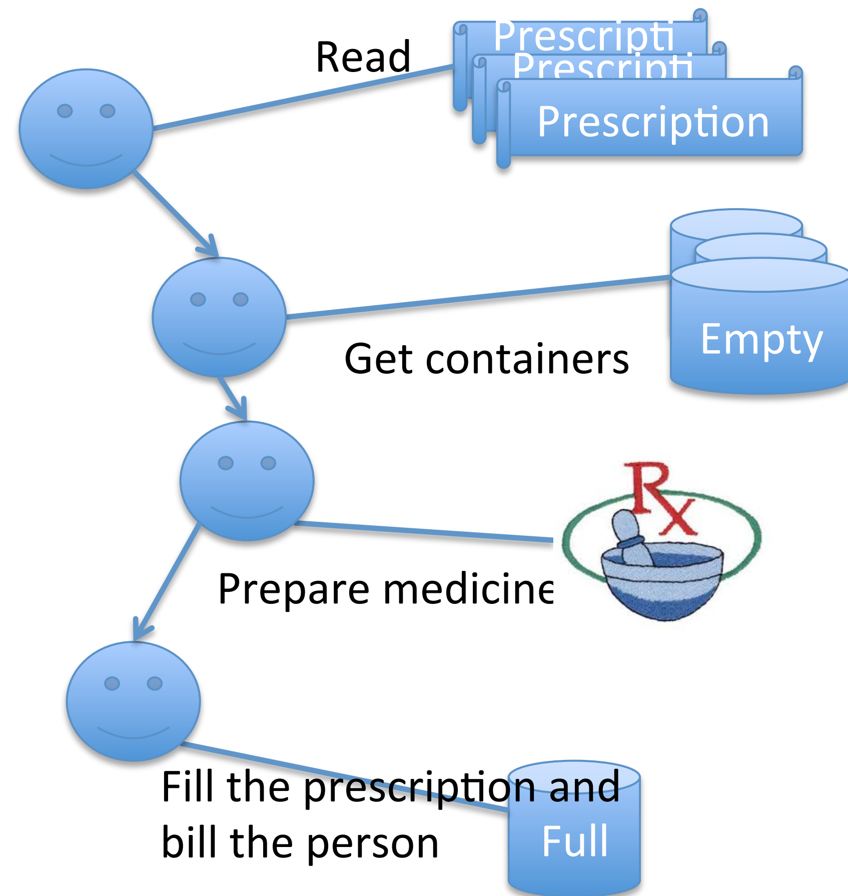
Total # cycles for n instructions: ceil(n/2)*5
For n = 5 => (15 cycles)

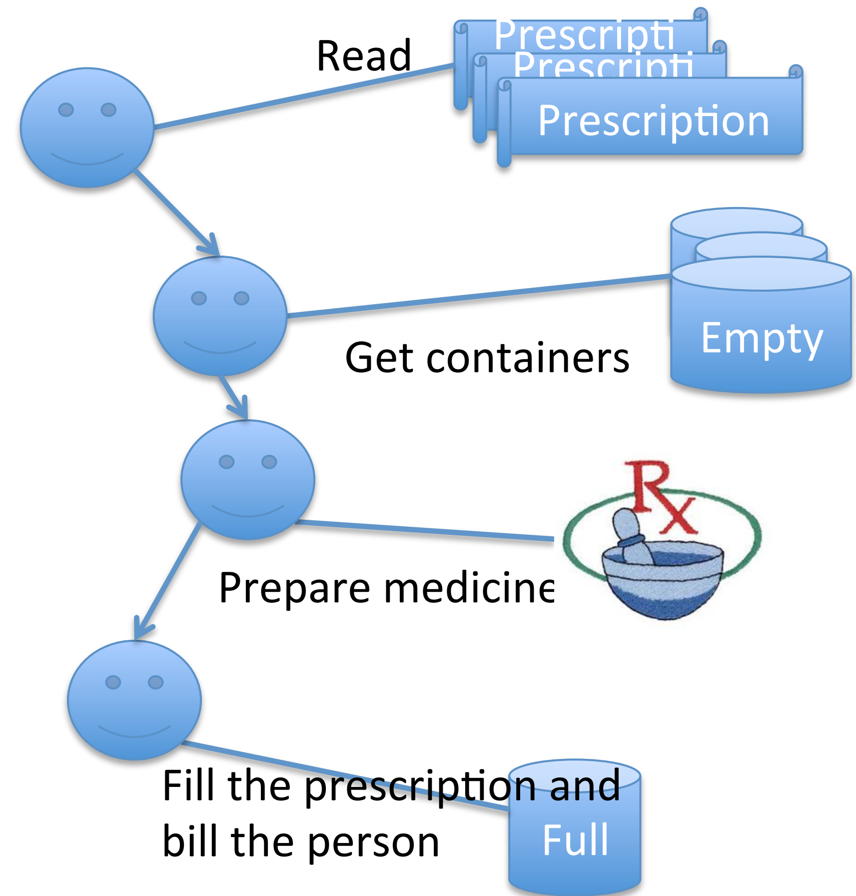
In general, for a superscalar processor with multiple issue width of w and l non-pipelined stages for processing each instruction; assuming one clock cycle per stage, time taken to complete n operations is (l*ceil(n/w)).

Superscalar Architecture

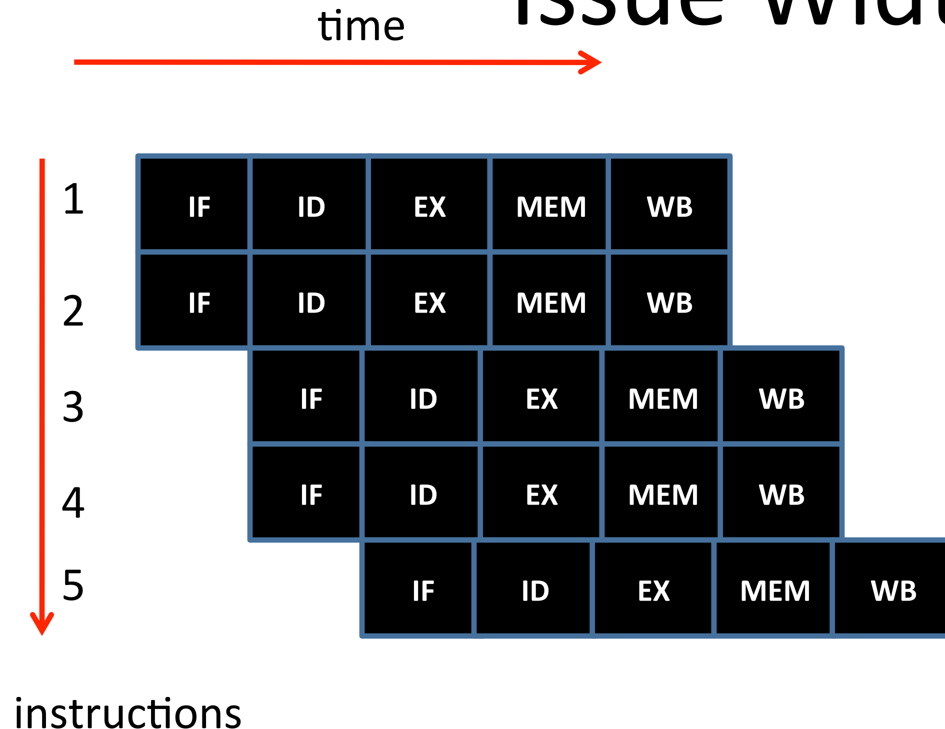
- A superscalar CPU architecture implements ILP within a single processor to achieve a higher throughput.
- A superscalar processor executes more than one instruction during a clock cycle by simultaneously dispatching multiple instructions to redundant functional units on the processor.
- Each functional unit is not a separate CPU core but an execution resource within a single CPU such as an arithmetic logic unit, a bit shifter, or a multiplier.

Analogy: Pharmacy
Superscalar with pipelining





Superscalar with Pipeline
Issue Width = 2

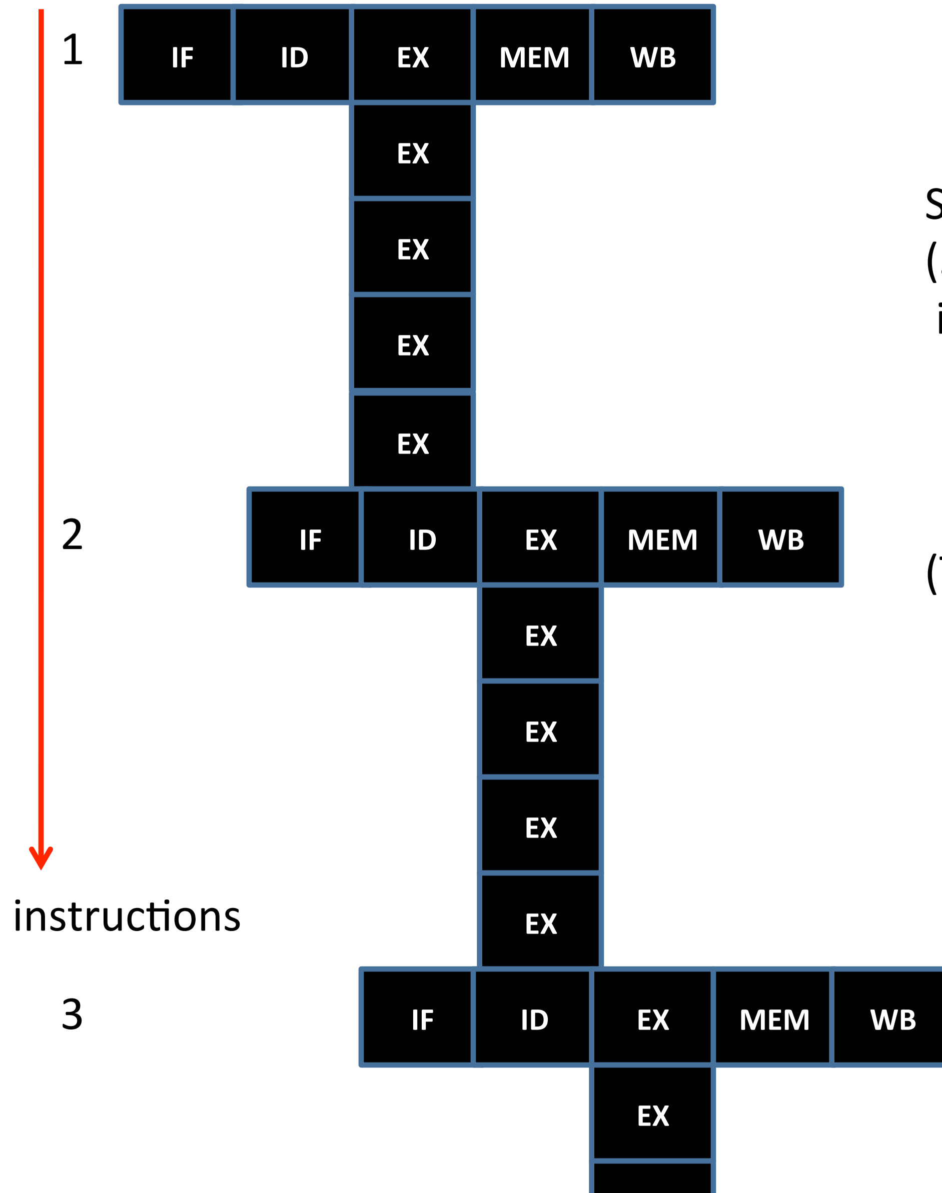


add r3, r1, r2
add r6,r4, r5
add r9, r7, r8
add r12, r10,r11
add r15, r13,r14

Assuming 1 cycle for IF/ID/EX/MEM/WB,

Total # cycles for n instructions = 5 + ceil[(n-2)/2]
For (n = 5) => (7 cycles)

SIMD Pipeline



Assume 5-element vector
v1 = [r1, r4, r7,r10,r13]
v2 = [r2, r5, r8, r11, r14]

addv v3, v1, v2

Single vector instr takes 5 cycles
(5 scalar insts
in 7 cycles on superscalar)

addv v6, v5, v4
addv v9, v8, v7

(Total of 15 scalar adds in 7 cycles) vs
(4 scalar adds in 7 cycles)

Fewer instructions operate on more elements in parallel

If the size of the vector register is v, i.e., it can hold v data; n vector operations can be performed in (l+n-1) clock cycles and each operation operates on v data elements of the vector. So in (l+n-1) time v*n data is operated on. This is true for SIMD (Single Instruction Multiple Data) processing as well.

Instruction Level Parallelism (ILP)

- Instruction-level parallelism (ILP) is a measure of how many of the operations in a computer program can be performed simultaneously.
- Level of ILP in a program is application dependent.
- ILP is discovered by compiler and utilized by processors designed to do so. Not explicitly managed by programmer.

Examples of Exploiting Instruction Level Parallelism (ILP)

- Pipelining – performing parts of the instruction cycle of different instructions concurrently.
- Superscalar – using more than one functional units and thereby executing arithmetic instructions concurrently.
- Out of order execution – Instructions execute in any order without violating data dependencies.
- Speculative Execution - which allow the execution of complete instructions or parts of instructions before being certain whether this execution should take place.