Pipelining Review and Its Limitations

Yuri Baida
yuri.baida@gmail.com
yuriy.v.baida@intel.com

October 16, 2010

MDSP Project | Intel Lab
Moscow Institute of Physics and Technology

Agenda

- Review
  - Instruction set architecture
  - Basic tools for computer architects
- Amdahl's law
- Pipelining
  - Ideal pipeline
  - Cost-performance trade-off
  - Dependencies
  - Hazards, interlocks & stalls
  - Forwarding
  - Limits of simple pipeline

Review

Instruction Set Architecture

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines instruction format (bit encoding) and instruction semantics
  - Examples: MIPS, x86, IBM 360, JVM
- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), z10 (c. 2008)
  - x86 implementations: 8086, 286, 386, 486, Pentium, Pentium Pro, Pentium 4, Core 2, Core i7, AMD Athlon, Transmeta Crusoe
  - MIPS implementations: R2000, R4000, R10000, R18K, …
  - JVM: HotSpot, PicoJava, ARM Jazelle, …

Basic Tools for Computer Architects

- Amdahl's law
- Pipelining
- Out-of-order execution
- Critical path
- Speculation
- Locality
- Important metrics
  - Performance
  - Power/energy
  - Cost
  - Complexity

Amdahl's Law

Amdahl's Law: Definition

- Speedup = $\text{Time}_{\text{without enhancement}}$ / $\text{Time}_{\text{with enhancement}}$
- Suppose an enhancement speeds up a fraction f of a task by a factor of S

$$\text{Time}_{\text{new}} = \text{Time}_{\text{old}} \times \left((1 - f) + \frac{f}{S} \right) \Rightarrow \quad S_{\text{overall}} = \frac{1}{(1 - f) + \frac{f}{S}}$$

- We should concentrate efforts on improving frequently occurring events or frequently used mechanisms

Amdahl's Law: Example

- New processor 10 times faster
- Input-output is a bottleneck
  - 60% of time we wait

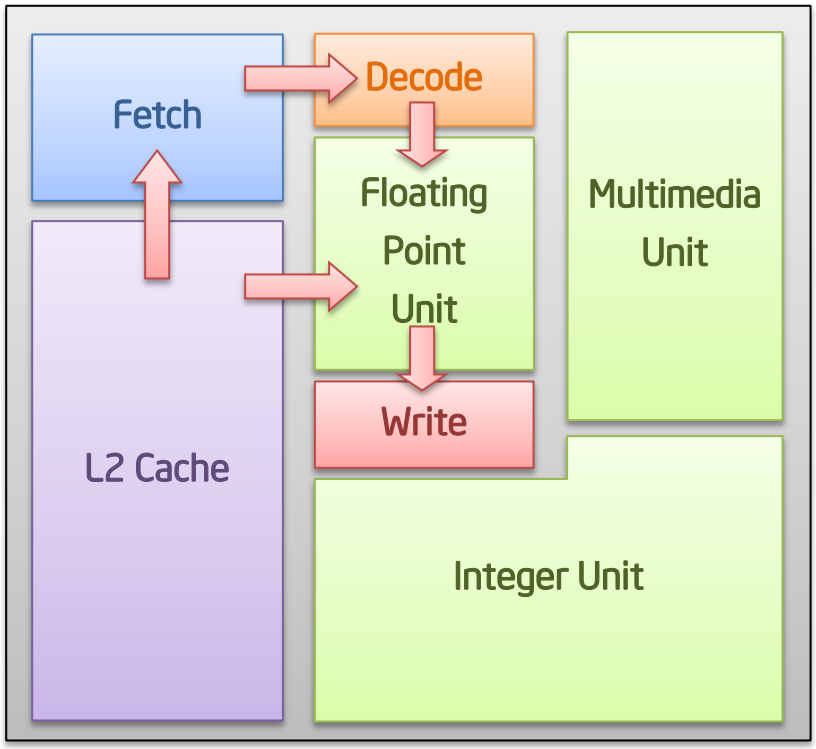
- Let's calculate

$$S_{\text{overall}} = \frac{1}{(1-f) + \frac{f}{S}} = \frac{1}{(1-0.4) + \frac{0.4}{10}} = \frac{1}{0.64} = 1.56$$

- Its human nature to be attracted by 10× faster
  - Keeping in perspective its just 1.6× faster

Pipelining

A Generic 4-stage Processor Pipeline

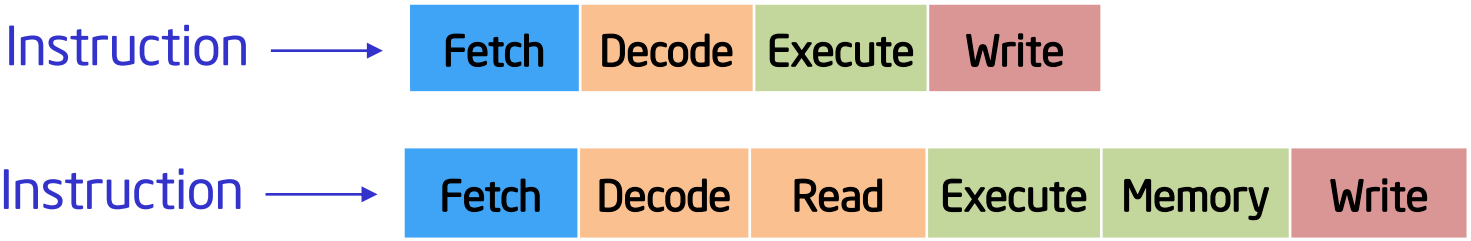


Fetch Unit gets the next instruction from the cache.

Decode Unit determines type of instruction.

Instruction and data sent to Execution Unit.

Write Unit stores result.

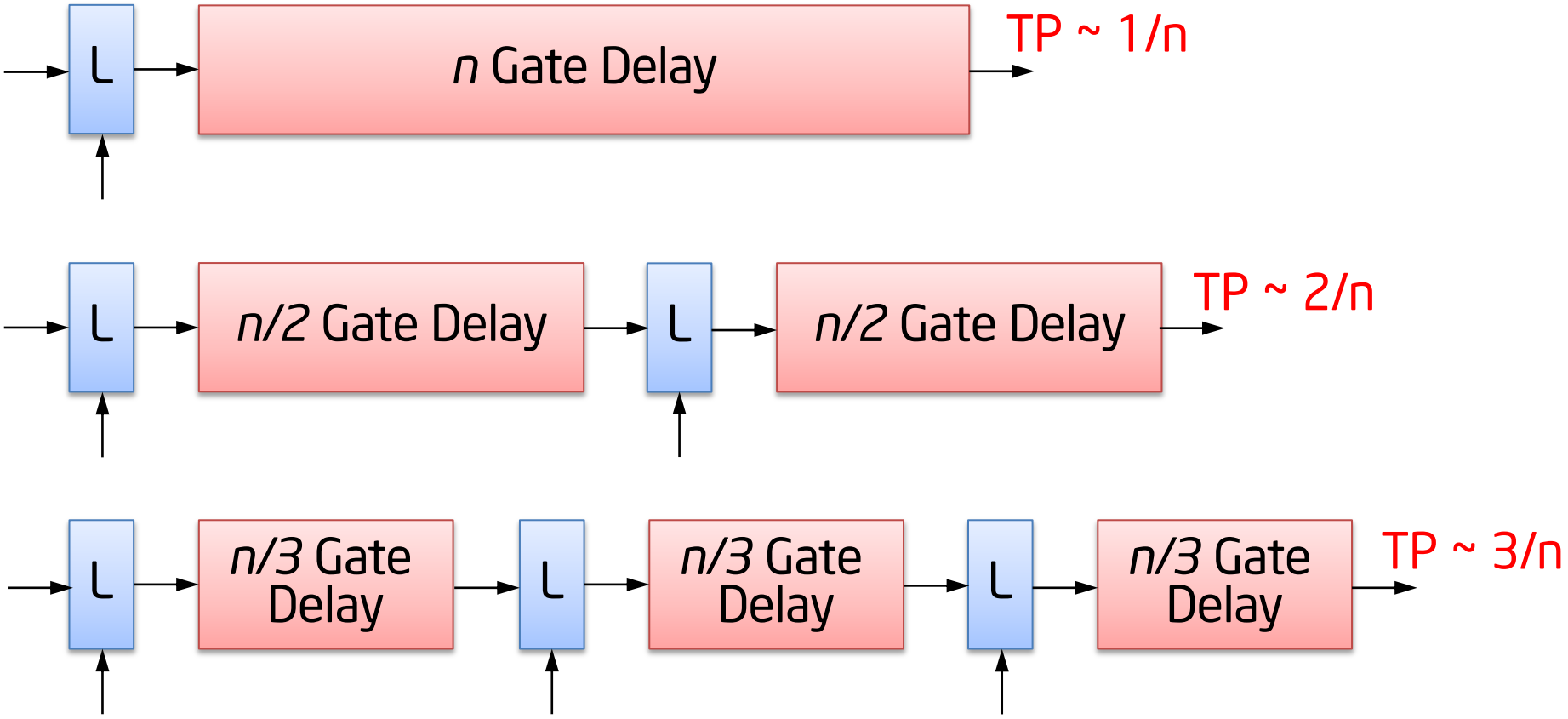


Pipeline: Steady State

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Read	Execute	Memory	Write			
$Instr_2$		Fetch	Decode	Read	Execute	Memory	Write		
$Instr_3$			Fetch	Decode	Read	Execute	Memory	Write	
$Instr_4$				Fetch	Decode	Read	Execute	Memory	Write
$Instr_5$					Fetch	Decode	Read	Execute	Memory
$Instr_6$						Fetch	Decode	Read	Execute

- Latency — elapsed time from start to completion of a particular task
- Throughput — how many tasks can be completed per unit of time
- Pipelining only improves throughput
  - Each job still takes 4 cycles to complete
- Real life analogy: Henry Ford's automobile assembly line

Pipelining Illustrated



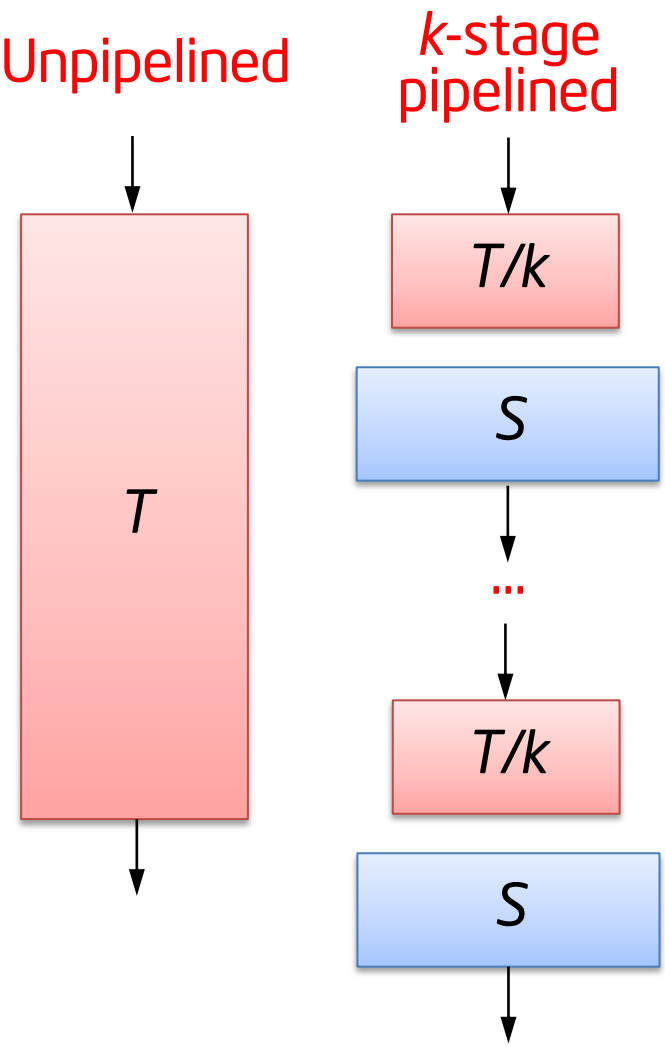
Pipelining Performance Model

- Starting from an unpipelined version with propagation delay *T* and TP = 1/*T*

$$P_{\text{pipelined}} \sim \text{TP}_{\text{pipelined}} = \frac{1}{\frac{T}{k} + S}$$

where

- *S* — delay through latch and overhead



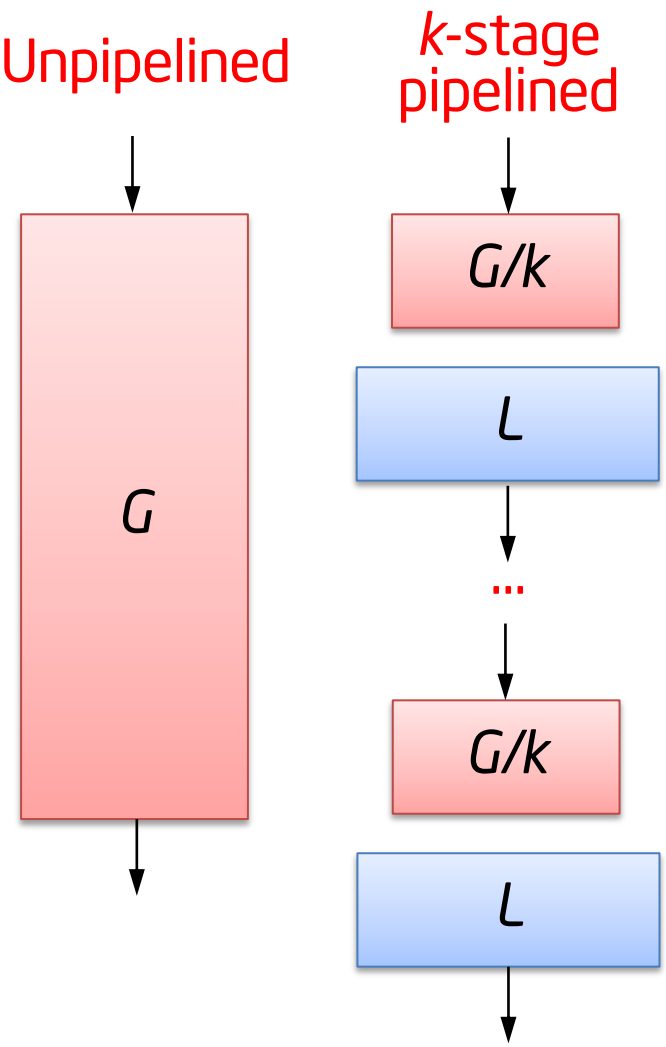
Hardware Cost Model

- Starting from an unpipelined version with hardware cost *G*

$$\mathrm{Cost}_{\mathrm{pipelined}} = kL + G$$

where

- *L* — cost of adding each latch



Cost/Performance Trade-off
[Peter M. Kogge, 1981]

$$\frac{\text{Cost}}{\text{Perf}} = \frac{kL + G}{\frac{1}{\frac{T}{k}+S}} = (kL+G)\left(\frac{T}{k} + S\right) =$$

$$LT + GS + LSk + \frac{GT}{k}$$

$$\frac{\mathrm{d}}{\mathrm{d}k}\left(\frac{\text{Cost}}{\text{Perf}}\right) = 0 + 0 + LS - \frac{GT}{k^2}$$

$$LS - \frac{GT}{k^2} = 0 \Rightarrow k_{\text{opt}} = \sqrt{\frac{GT}{LS}}$$

Pipelining Idealism

- Uniform suboperations
  - The operation to be pipelined can be evenly partitioned into uniformlatency suboperations
- Repetition of identical operations
  - The same operations are to be performed repeatedly on a large number of different inputs
- Repetition of independent operations
  - All the repetitions of the same operation are mutually independent
  - Good example: automobile assembly line

Pipelining Reality

- Uniform suboperations... NOT! ⇒ Balance pipeline stages
  - Stage quantization to yield balanced stages
  - Minimize internal fragmentation (some waiting stages)
- Identical operations... NOT! ⇒ Unify instruction types
  - Coalescing instruction types into one "multi-function" pipe
  - Minimize external fragmentation (some idling stages)
- Independent operations... NOT! ⇒ Resolve dependencies
  - Inter-instruction dependency detection and resolution
  - Minimize performance lose

Pipelining Reality

- Uniform suboperations… NOT! ⇒ Balance pipeline stages
  - Stage quantization to yield balanced stages
  - Minimize internal fragmentation (some waiting stages)
- Identical operations… NOT! ⇒ Unify instruction types
  - Coalescing instruction types into one "multi-function" pipe
  - Minimize external fragmentation (some idling stages)
- Independent operations… NOT! ⇒ Resolve dependencies
  - Inter-instruction dependency detection and resolution
  - Minimize performance lose

Pipelining Reality

- Uniform suboperations… NOT! ⇒ Balance pipeline stages
  - Stage quantization to yield balanced stages
  - Minimize internal fragmentation (some waiting stages)
- Identical operations… NOT! ⇒ Unify instruction types
  - Coalescing instruction types into one "multi-function" pipe
  - Minimize external fragmentation (some idling stages)
- Independent operations… NOT! ⇒ Resolve dependencies
  - Inter-instruction dependency detection and resolution
  - Minimize performance lose

Pipelining Reality

- Uniform suboperations… NOT! ⇒ Balance pipeline stages
  - Stage quantization to yield balanced stages
  - Minimize internal fragmentation (some waiting stages)
- Identical operations… NOT! ⇒ Unify instruction types
  - Coalescing instruction types into one "multi-function" pipe
  - Minimize external fragmentation (some idling stages)
- Independent operations… NOT! ⇒ Resolve dependencies
  - Inter-instruction dependency detection and resolution
  - Minimize performance lose

Pipelining Reality

- Uniform suboperations… NOT! ⇒ Balance pipeline stages
  - Stage quantization to yield balanced stages
  - Minimize internal fragmentation (some waiting stages)
- Identical operations… NOT! ⇒ Unify instruction types
  - Coalescing instruction types into one "multi-function" pipe
  - Minimize external fragmentation (some idling stages)
- Independent operations… NOT! ⇒ Resolve dependencies
  - Inter-instruction dependency detection and resolution
  - Minimize performance lose

Pipelining Reality

- Uniform suboperations… NOT! ⇒ Balance pipeline stages
  - Stage quantization to yield balanced stages
  - Minimize internal fragmentation (some waiting stages)
- Identical operations… NOT! ⇒ Unify instruction types
  - Coalescing instruction types into one “multi-function” pipe
  - Minimize external fragmentation (some idling stages)
- Independent operations… NOT! ⇒ Resolve dependencies
  - Inter-instruction dependency detection and resolution
  - Minimize performance lose

Hazards, Interlocks, and Stalls

- Pipeline hazards
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard resolution
  - Static Method: performed at compiled time in software
  - Dynamic Method: performed at run time using hardware
    - Stall
    - Flush
    - Forward
- Pipeline interlock
  - Hardware mechanisms for dynamic hazard resolution
  - Must detect and enforce dependences at run time

Dependencies & Pipeline Hazards

- Data dependence (register or memory)
  - True dependence (RAW)
    - Instruction must wait for all required input operands
  - Anti-dependence (WAR)
    - Later write must not clobber a still-pending earlier read
  - Output dependence (WAW)
    - Earlier write must not clobber an already-finished later write
- Control dependence
  - A “data dependency” on the instruction pointer
  - Conditional branches cause uncertainty to instruction sequencing
- Resource conflicts
  - Two instructions need the same device

Example: Quick Sort for MIPS

```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low
```

Example: Quick Sort for MIPS

```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low

        bge     $10,    $9,     L2
        mul     $15,    $10,    4
        addu    $24,    $6,     $15
        lw      $25,    0($24)
        mul     $13,    $8,     4
        addu    $14,    $6,     $13
        lw      $15,    0($14)
        bge     $25,    $15,    L2
L1:
        addu    $10,    $10,    1
        ...
L2:
        addu    $11,    $11,    -1
```

Example: Quick Sort for MIPS

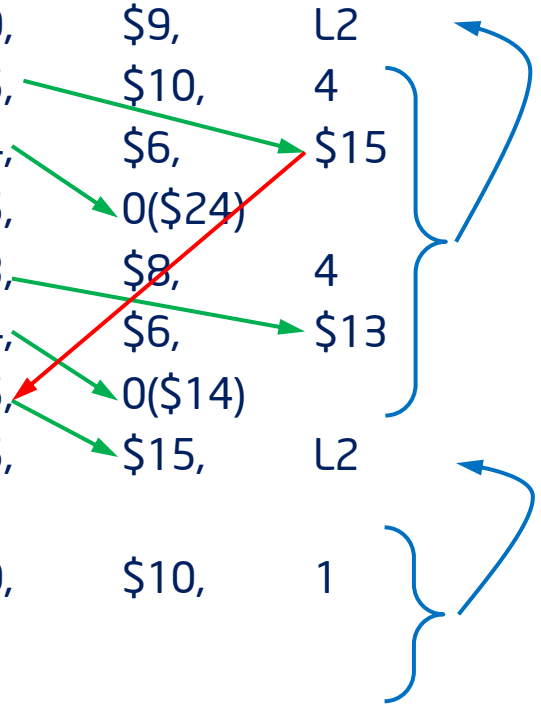
```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low

        bge     $10,    $9,     L2
        mul     $15,    $10,    4
        addu    $24,    $6,     $15
        lw      $25,    0($24)
        mul     $13,    $8,     4
        addu    $14,    $6,     $13
        lw      $15,    0($14)
        bge     $25,    $15,    L2
L1:
        addu    $10,    $10,    1
        ...
L2:
        addu    $11,    $11,    -1
```

Example: Quick Sort for MIPS

```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low

        bge     $10,    $9,     L2
        mul     $15,    $10,    4
        addu    $24,    $6,     $15
        lw      $25,    0($24)
        mul     $13,    $8,     4
        addu    $14,    $6,     $13
        lw      $15,    0($14)
        bge     $25,    $15,    L2
L1:
        addu    $10,    $10,    1
        ...
L2:
        addu    $11,    $11,    -1
```

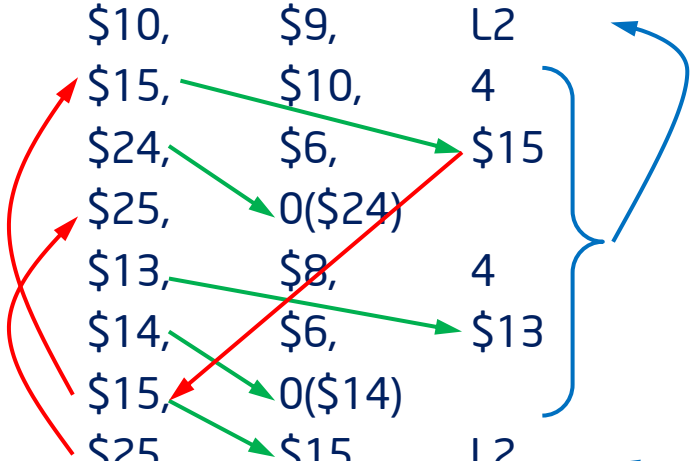
Example: Quick Sort for MIPS

```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low

        bge     $10,    $9,     L2
        mul     $15,    $10,    4
        addu    $24,    $6,     $15
        lw      $25,    0($24)
        mul     $13,    $8,     4
        addu    $14,    $6,     $13
        lw      $15,    0($14)
        bge     $25,    $15,    L2
L1:
        addu    $10,    $10,    1
        ...
L2:
        addu    $11,    $11,    -1
```



Example: Quick Sort for MIPS

```
# for (;(j<high)&&(array[j]<array[low]);++j);
# $10 = j; $9 = high; $6 = array; $8 = low

        bge     $10,    $9,     L2
        mul     $15,    $10,    4
        addu    $24,    $6,     $15
        lw      $25,    0($24)
        mul     $13,    $8,     4
        addu    $14,    $6,     $13
        lw      $15,    0($14)
        bge     $25,    $15,    L2
L1:
        addu    $10,    $10,    1
        ...
L2:
        addu    $11,    $11,    -1
```



Pipeline: Data Hazards

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Read	Execute	Memory	Write			
$Instr_2$		Fetch	Decode	Read	Execute	Memory	Write		
$Instr_3$			Fetch	Decode	Read	Execute	Memory	Write	
$Instr_4$				Fetch	Decode	Read	Execute	Memory	Write
$Instr_5$					Fetch	Decode	Read	Execute	Memory
$Instr_6$						Fetch	Decode	Read	Execute

- $Instr_2$: _ → r_k
- $Instr_3$: r_k → _

- How long should we stall for?

Pipeline: Stall on Data Hazard

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Read	Execute	Memory	Write			
$Instr_2$		Fetch	Decode	Read	Execute	Memory	Write		
$Instr_3$			Fetch	Decode	Stalled			Read	Execute
$Instr_4$				Fetch	Stalled			Decode	Read
$Instr_5$					Stalled			Fetch	Decode
$Instr_6$									

- $Instr_2$: _ → r_k
- Bubble
- Bubble
- Bubble
- $Instr_3$: r_k → _

- Make the younger instruction wait until the hazard has passed:
  - Stop all up-stream stages
  - Drain all down-stream stages

Pipeline: Forwarding

Cycle	1	2	3	4	5	6	7	8	9
$Instr_1$	Fetch	Decode	Read	Execute	Memory	Write			
$Instr_2$		Fetch	Decode	Read	Execute	Memory	Write		
$Instr_3$			Fetch	Decode	Read	Execute	Memory	Write	
$Instr_4$				Fetch	Decode	Read	Execute	Memory	Write
$Instr_5$					Fetch	Decode	Read	Execute	Memory
$Instr_6$						Fetch	Decode	Read	Execute

Limitations of Simple Pipelined Processors
(aka Scalar Processors)

- Upper bound on scalar pipeline throughput
  - Limited by IPC = 1
- Inefficiencies of very deep pipelines
  - Clocking overheads
  - Longer hazards and stalls
- Performance lost due to in-order pipeline
  - Unnecessary stalls
- Inefficient unification Into single pipeline
  - Long latency for each instruction
  - Hazards and associated stalls

Limitations of Deeper Pipelines

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}} =$$

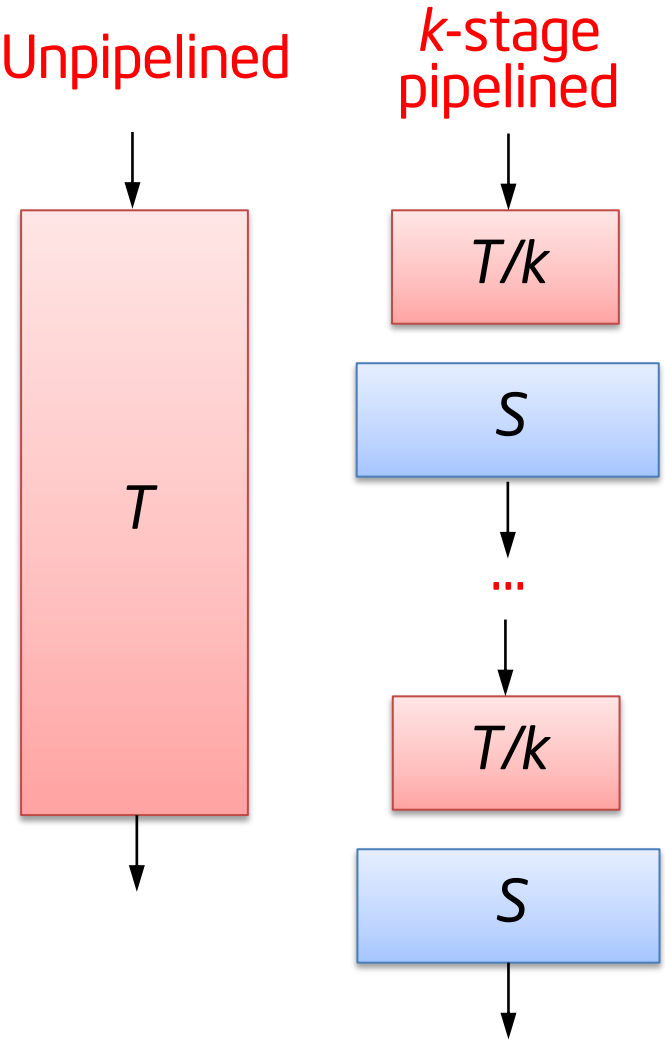
$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(Code size) (CPI) (Cycle)

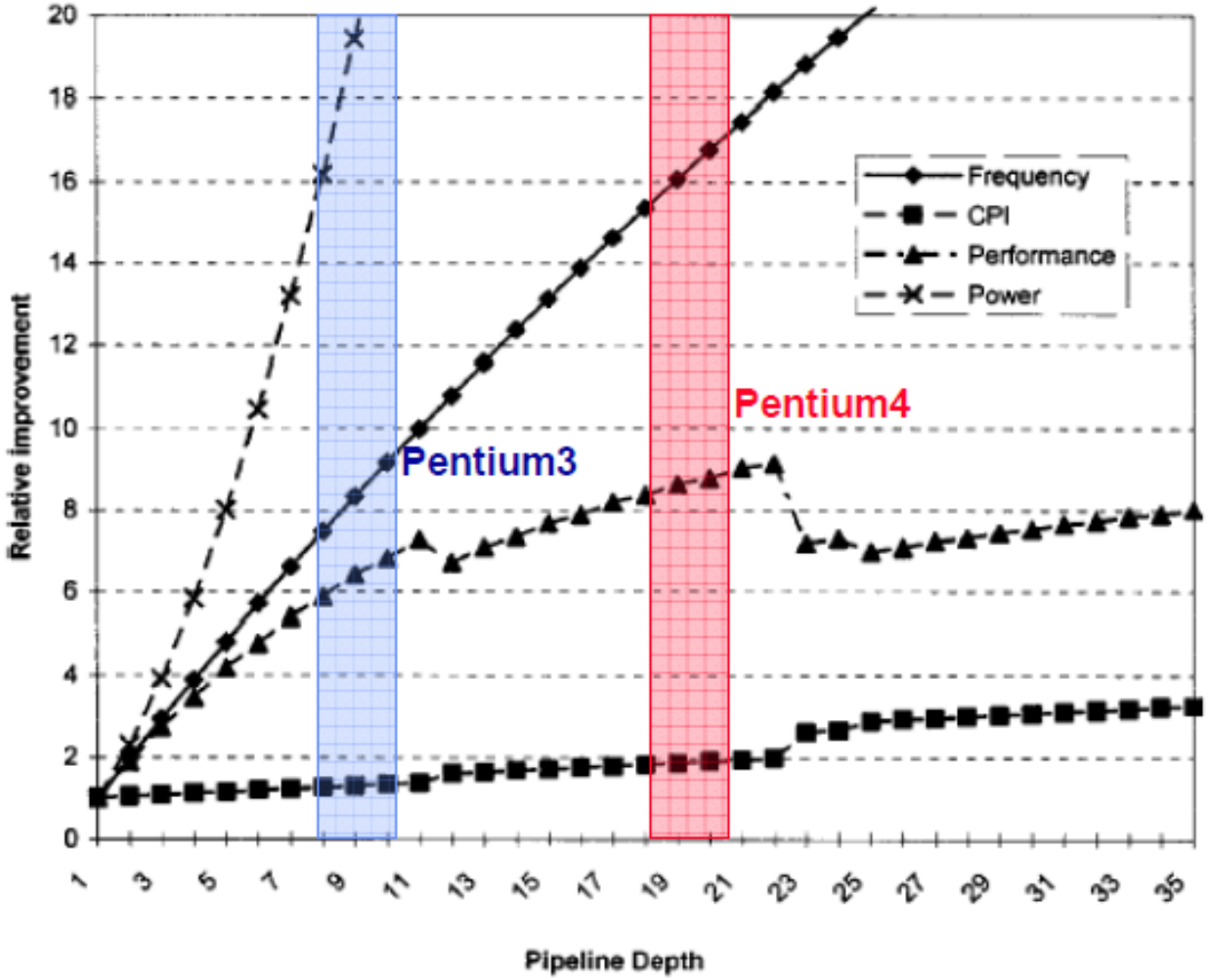
?

$\frac{T}{k} + S$

Eventually limited by S



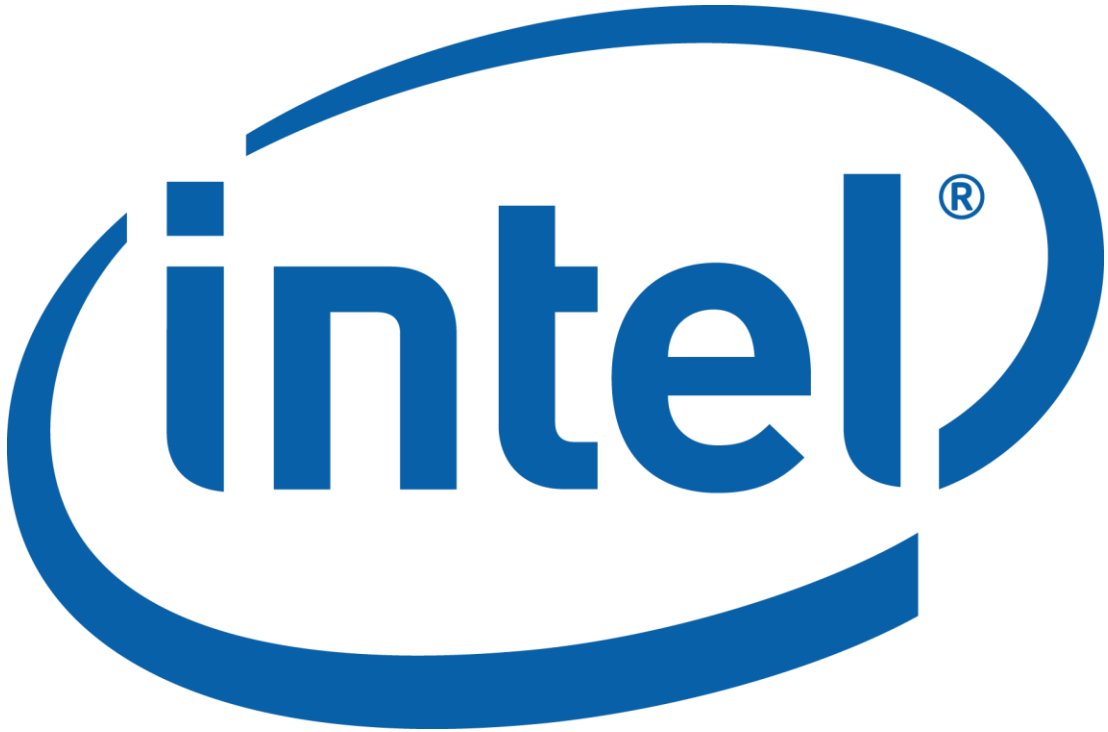
Put it All Together: Limits to Deeper Pipelines



Source: Ed Grochowski, 1997

Acknowledgements

- These slides contain material developed and copyright by:
  - Grant McFarland (Intel)
  - Christos Kozyrakis (Stanford University)
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)


intel