

Parallel Programming

Architectures – Pt.1

Prof. Paolo Bientinesi

HPAC, RWTH Aachen
`pauldj@aices.rwth-aachen.de`

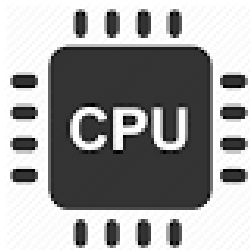
WS16/17



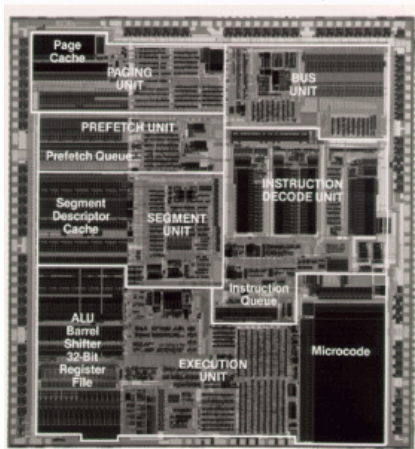
Outline

- 1 Uniprocessor Architecture Review
- 2 Data Dependencies

Quick architecture review

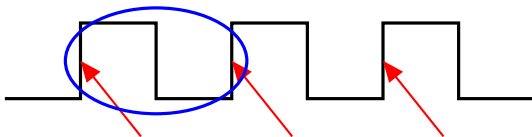


From Computer Desktop Encyclopedia
Reproduced with permission.
© 2001 Intel Corporation



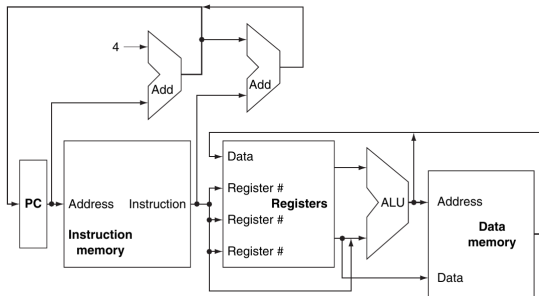
Clock, cycle, frequency

- Clock determines when events take place in the hardware



- Frequency (or clock rate): # of cycles per second.
For instance: 2GHz $\rightarrow 2 \times 10^9$ cycles per second

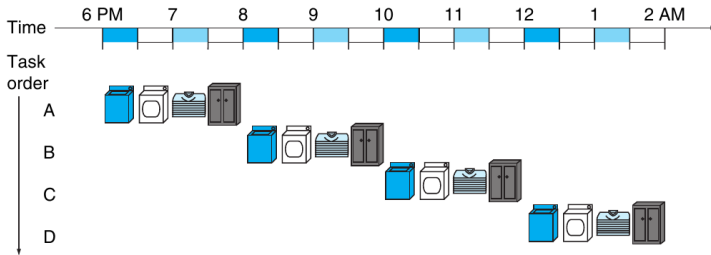
Basic processor architecture



Source: *Computer organization and design.* Patterson, Hennessy.

- **Instruction Fetch (IF):** read instruction from cache
- **Instruction Decode (ID):** read register data
- **Execute (EX):** execute arithmetic/logic operation
- **Store (ST):** store the result

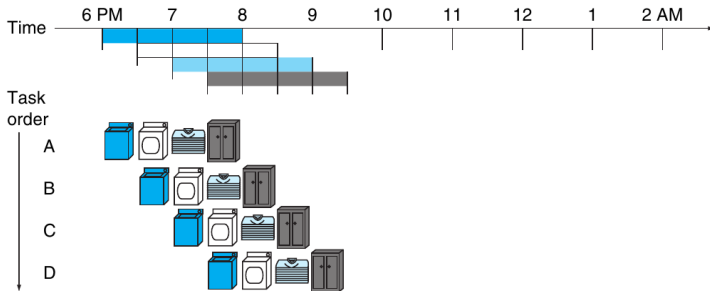
The laundry analogy



Source: *Computer organization and design*. Patterson, Hennessy.

- Latency: 1 load takes 2 hours
- Throughput: 4 loads take 8 hours, $\frac{1}{2}$ load per hour
- How can we improve the throughput? **Pipelining**

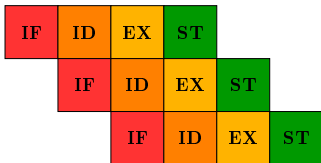
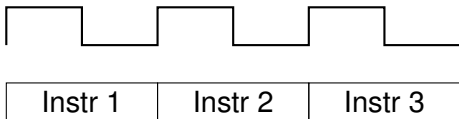
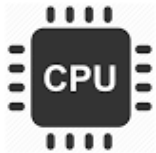
Pipelining



Source: *Computer organization and design*. Patterson, Hennessy.

- Latency: still 2 hours
- Throughput: $(\frac{n}{2+(n-1)*0.5})$ loads per hour ($\frac{4}{3.5} \approx 1.14$ loads/hour)
- $\lim_{n \rightarrow \infty} \text{Throughput} = 2$ (vs original $\frac{1}{2}$)

Pipelined processors



- Each step is known as *stage*
- 4-stage pipeline

Throughput

- Program \mathcal{P} : n instructions
- s -stage pipeline
- $\text{latency}(\text{stage}) = k$ secs

Single resource
 \Rightarrow **serial execution**

Latency(\mathcal{P}) = nsk secs

Throughput:
 $\frac{n}{nsk} = \frac{1}{sk}$ instr/sec

Multiple resources
 \Rightarrow **pipelined execution**

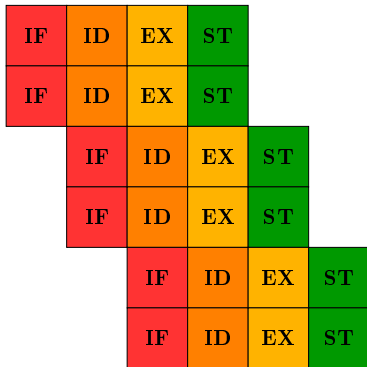
Latency(\mathcal{P}) = $sk + (n - 1)k$ secs

Throughput:
 $\lim_{n \rightarrow \infty} \frac{n}{sk + (n - 1)k} = \frac{1}{k}$ instr/sec

Morale: The throughput for the pipelined execution is s times as large as the one for the serial execution. Also, the more stages (the larger s), the smaller k , and the higher the throughput.

Multiple-issue processors

- Replicate internal components to launch multiple instructions per cycle
- Allows instruction execution rate $>$ clock rate
- That is, allows to complete the execution of more than one IPC



Towards the multi-core era

Limitations in ILP

Trends in multiple-issue processors.

	486	Pentium	Pentium II	Pentium 4	Itanium	Itanium 2	Core2
Year	1989	1993	1998	2001	2002	2004	2006
Width	1	2	3	3	3	6	4

- High-performance processors:
 - Issue width has stabilized at 4-6
 - Alpha 21464 (8-way) was canceled (2001).
 - Need hardware/compiler scheduling to exploit the width
- Embedded/Low-power processors:
 - Typical width of 2
 - Simpler architectures, no advanced scheduling

Towards the multi-core era

Limitations in ILP

Microarchitecture	Pipeline stages
i486	3
P5 (Pentium)	5
P6 (Pentium Pro/II)	14
P6 (Pentium 3)	8
P6 (Pentium M)	10
NetBurst (Northwood)	20
NetBurst (Prescott)	31
Core	12
Nehalem	20
Sandy Bridge	14
Haswell	14

Table: Evolution of the pipeline depth for a sample of Intel microarchitectures.

Source: [wikipedia.org](https://en.wikipedia.org/wiki/Intel_Pipeline)

Outline

- 1 Uniprocessor Architecture Review
- 2 Data Dependencies

On dependencies and parallel execution

- In parallel execution there is no time ordering:
If two instructions are executed in parallel, one cannot expect that one finishes before the other
- Two independent instructions can be executed in parallel (by different resources)
- If there are dependencies, then an ordering is necessary to preserve the semantics of the program
- Pipelining is still possible, even with dependencies
- Some dependencies can be removed by duplicating data

True / Flow dependency

{x = 1, y = 2, a = 3}	
...	...
y := a * x + y	w := 3 * y
w := 3 * y	y := a * x + y
...	...
{y = 5, w = 15}	{y = 5, w = 6}

- Calculation of value w depends on the updated value of y
- Executing the two statements in different order changes the semantics of the program

Anti dependency

$\{x = 1, y = 2, a = 3\}$

...	...
$w := 3 * y$	$y := a * x + y$
$y := a * x + y$	$w := 3 * y$
...	...
$\{y = 5, w = 6\}$	$\{y = 5, w = 15\}$

- Calculation of value w needs the initial value of y
- Executing the two statements in different order changes the semantics of the program

Output dependency

$\{x = 1, y = 2, a = 3\}$

...		...
$w := 3 * y$		$w := a * x$
$w := a * x$		$w := 3 * y$
...		...
$\{w = 3\}$		$\{w = 6\}$

- Final value of w depends on the order in which the statements are executed
- Executing the two statements in different order changes the semantics of the program