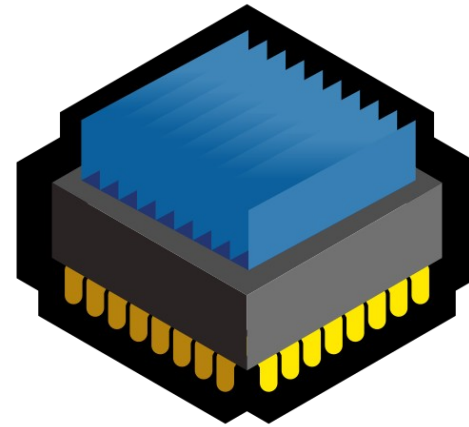


CS 261

Fall 2022

Mike Lam, Professor



CPU Architecture

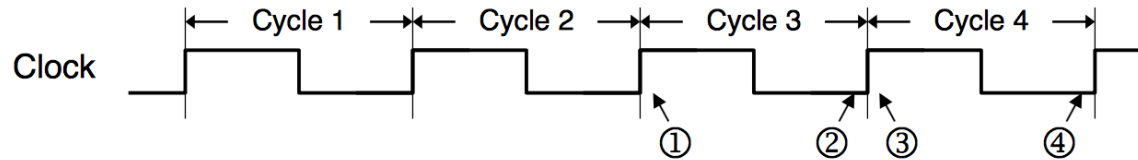
Topics

- CPU stages and design
- Pipelining

CPU overview

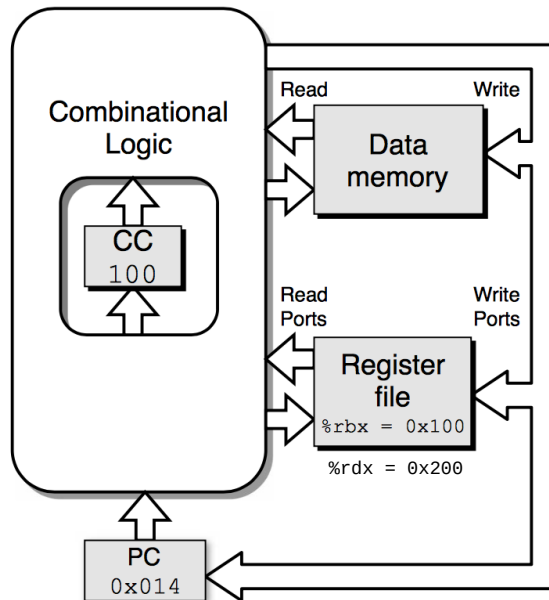
- A CPU consists of
 - Combinational circuits for computation
 - Sequential circuits for memory
 - Wires/buses for connectivity and intermediate results
 - A clocked register PC for synchronization

Example

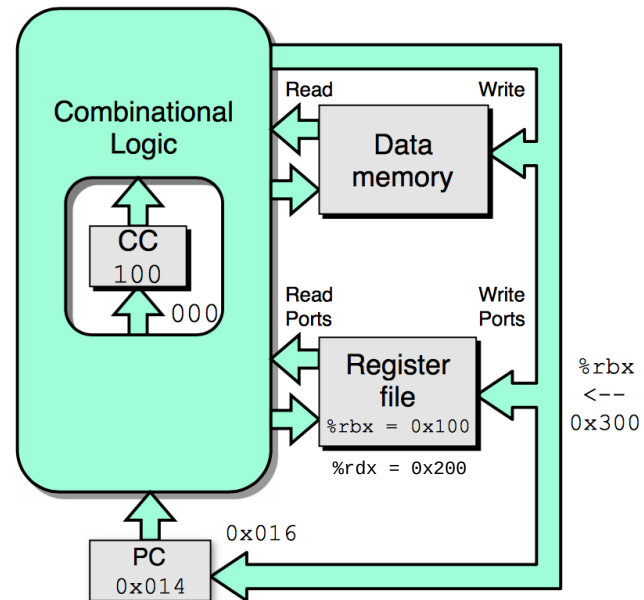


Cycle 1:	0x000:	<code>irmovq \$0x100,%rbx</code>	# %rbx <-- 0x100
Cycle 2:	0x00a:	<code>irmovq \$0x200,%rdx</code>	# %rdx <-- 0x200
Cycle 3:	0x014:	<code>addq %rdx,%rbx</code>	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	<code>je dest</code>	# Not taken
Cycle 5:	0x01f:	<code>rmmovq %rbx,0(%rdx)</code>	# M[0x200] <-- 0x300

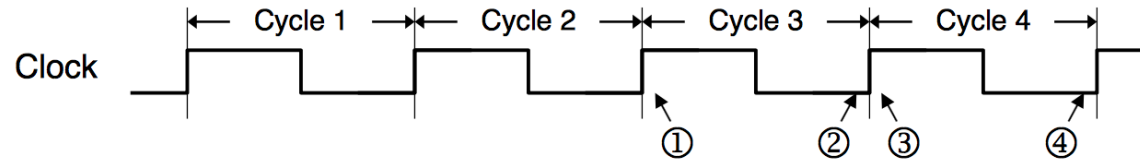
① Beginning of cycle 3



② End of cycle 3

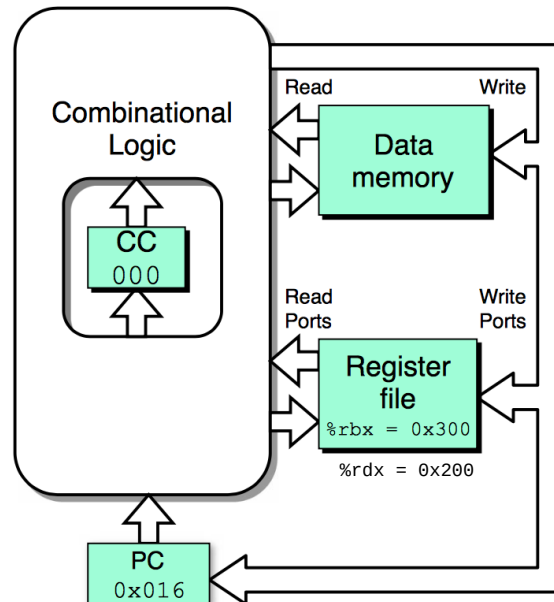


Example

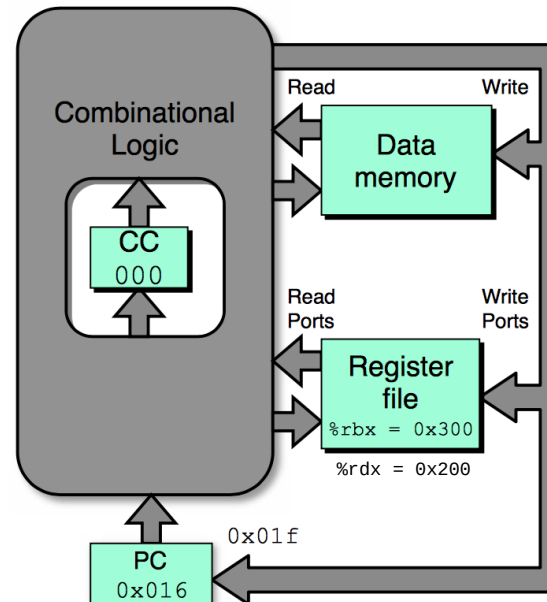


Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

③ Beginning of cycle 4

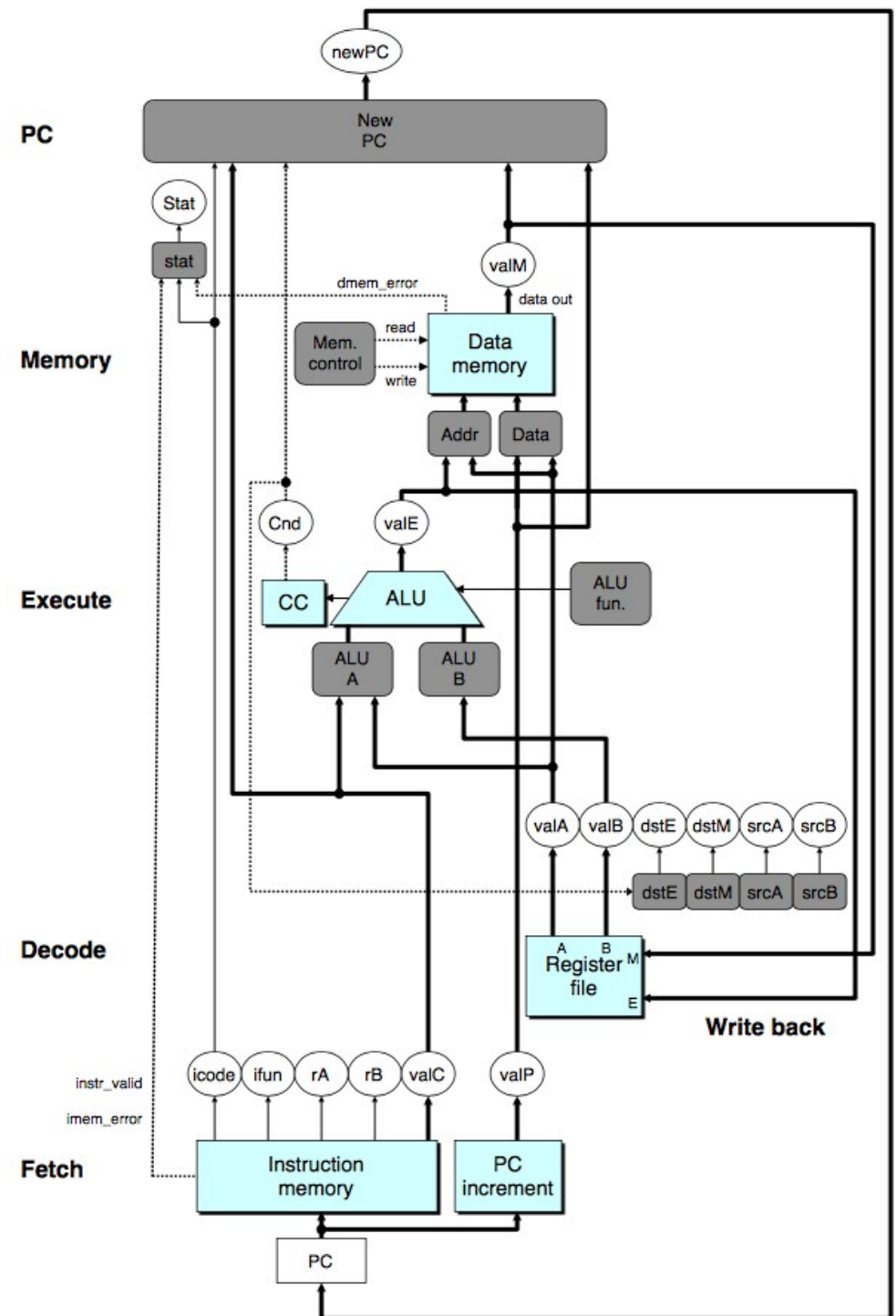


④ End of cycle 4



CPU design

- **SEQ**: sequential Y86 CPU
 - Runs one instruction at a time
 - ysim: simulator
- **Components**:
 - Clocked register (PC)
 - Hardware units (blue boxes)
 - Combinational/sequential circuits
 - ALU, register file, memory
 - Control logic (grey rectangles)
 - Combinational circuits
 - Details in textbook
 - Wires (white circles)
 - Word (thick lines)
 - Byte (thin lines)
 - Bit (dotted lines)

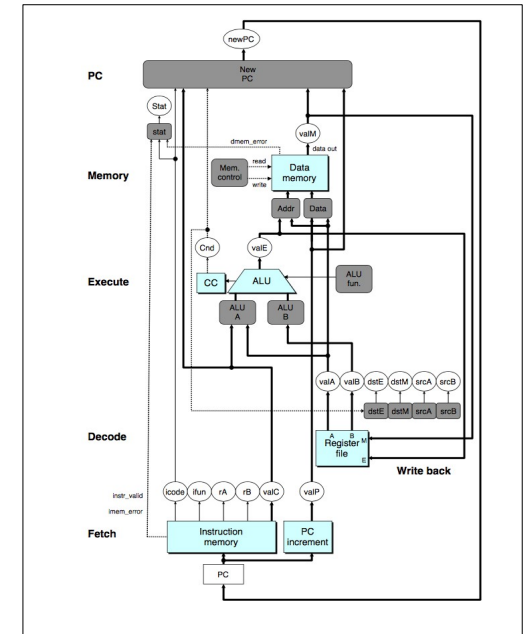


System design

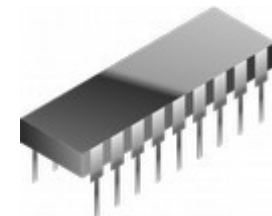
- CPU measurement
 - **Throughput**: instructions executed per second
 - GIPS: billions of (“giga-”) instructions per second
 - 1 GIPS → each instruction takes 1 nanosecond (a billionth of a second)
 - **Latency**: time required per instruction (or sequence)
 - Picosecond: 10^{-12} seconds Nanosecond: 10^{-9} seconds
 - 1,000 ps = 1 nanosecond
 - Relationship: *throughput* = # instructions / latency
 - Example: $1 / 320\text{ps} * (1000\text{ps/ns}) = 0.003125 * 1000 \approx 3.1$ GIPS

System design

- Current CPU design is serial
 - One instruction executes at a time
 - Only way to improve is to run faster!
 - Limited by speed of light / electricity
- One approach: make it smaller
 - Shorter circuit = faster circuit
 - Limited by manufacturing technology

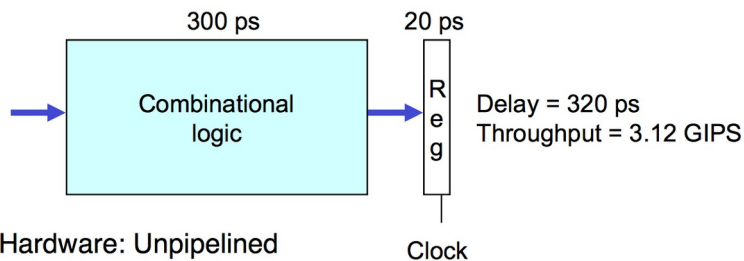


What else could we do?

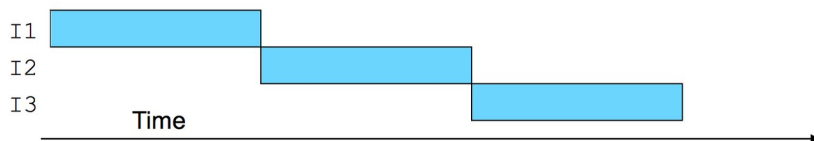


System design

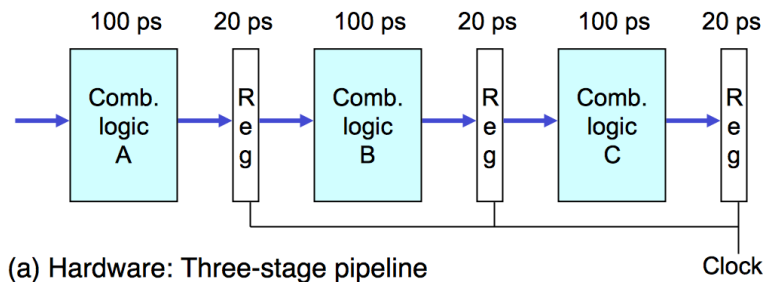
- Idea: **pipelined** design
 - Multiple instructions execute simultaneously (“**instruction-level parallelism**”)
 - Similar to cafeteria line or car wash
 - Split logic into stages and connect stages with clocked registers



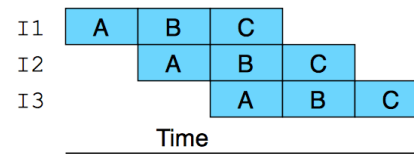
(a) Hardware: Unpipelined



(b) Pipeline diagram



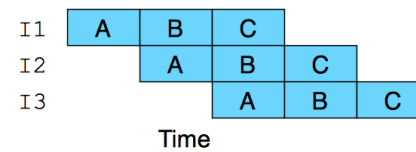
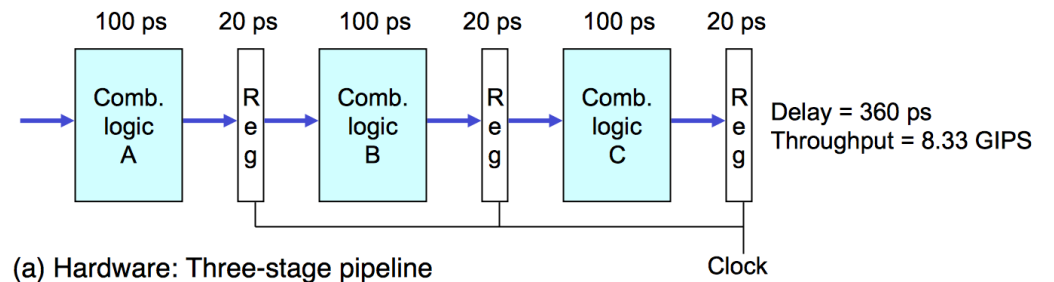
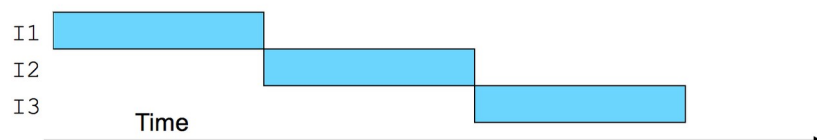
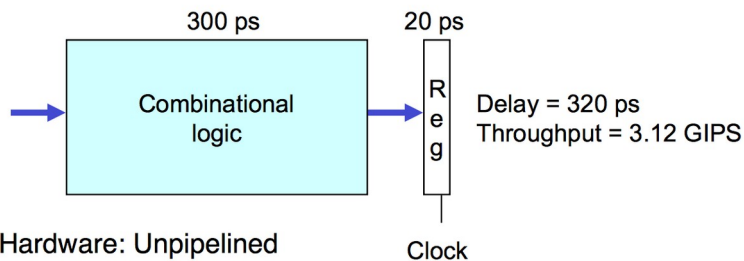
(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

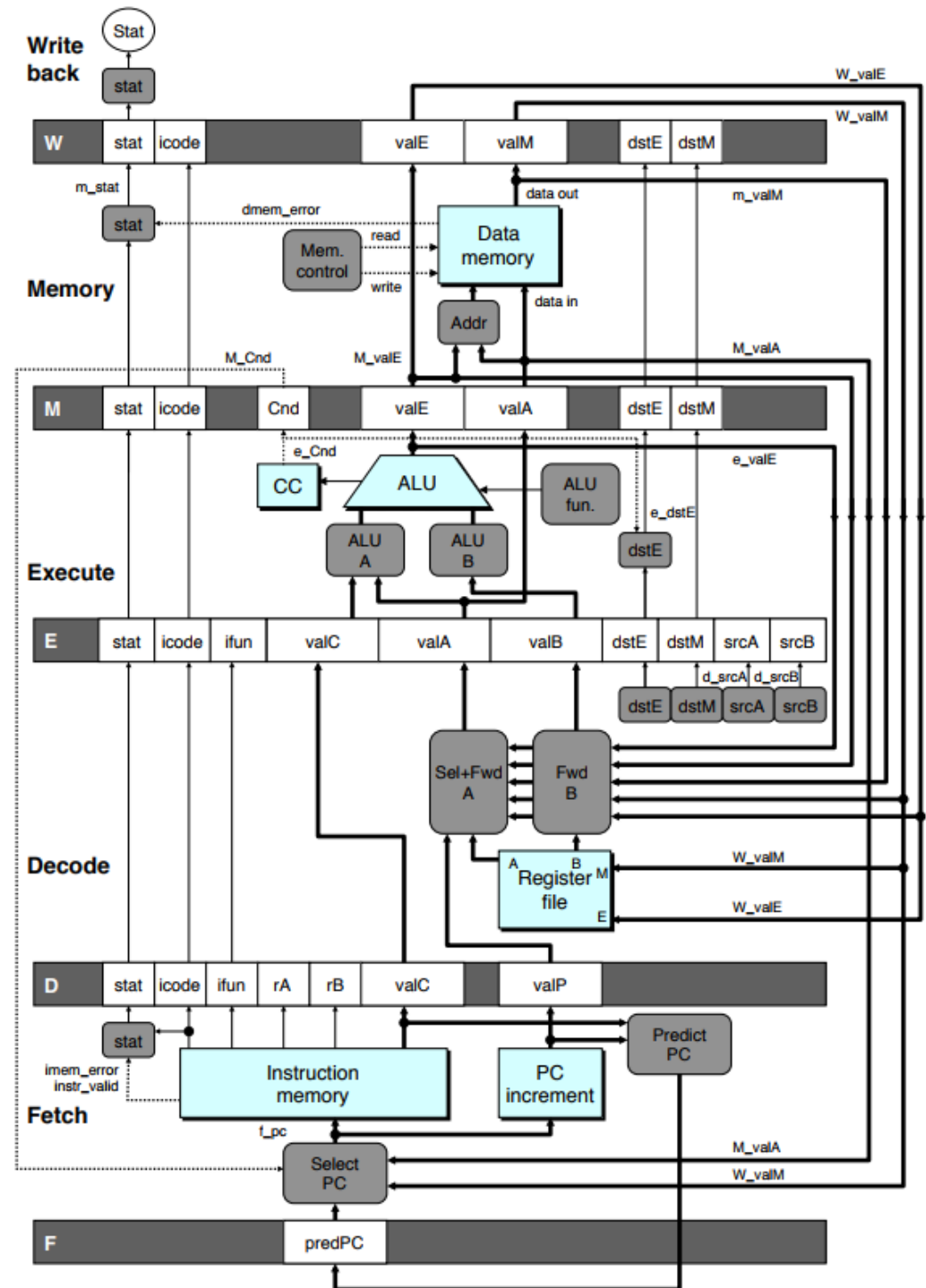
System design

- Idea: **pipelined** design
 - Multiple instructions execute simultaneously (“**instruction-level parallelism**”)
 - Similar to cafeteria line or car wash
 - Split logic into stages and connect stages with clocked registers
 - System design tradeoff: **throughput vs. latency**



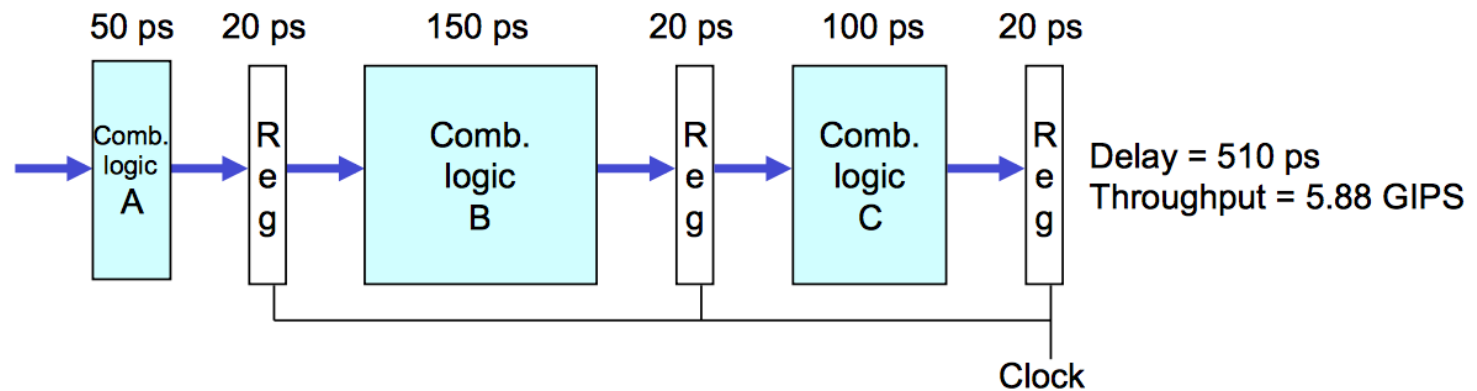
Y86 pipelining

- It's complicated!
 - Split up the stages and add more clocked registers for intermediate results

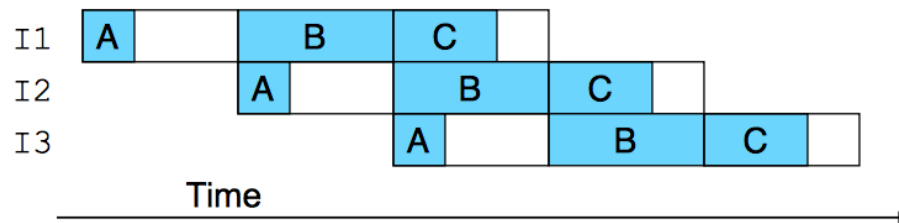


Pipelining

- Limitation: **non-uniform partitioning**
 - Logic segments may have significantly different lengths



(a) Hardware: Three-stage pipeline, nonuniform stage delays



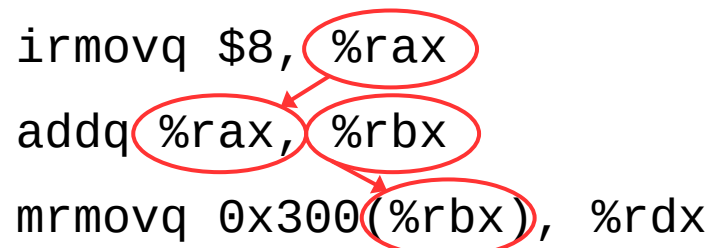
(b) Pipeline diagram

Pipelining

- Limitation: **dependencies**
 - The effect of one instruction depends on the result of another
 - Both **data** and **control** dependencies
 - Sometimes referred to as **hazards**

Data dependency:

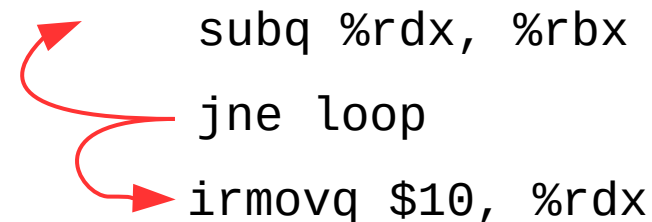
```
irmovq $8, %rax
addq %rax, %rbx
mrmovq 0x300(%rbx), %rdx
```



Control dependency:

loop:

```
    subq %rdx, %rbx
    jne loop
    irmovq $10, %rdx
```



Pipelining

- Approaches to avoiding hazards
 - **Halt** execution (or throw an exception)
 - **Stalling**: “hold back” an instruction temporarily
 - **Data forwarding**: allow latter stages to feed into earlier stages, bypassing memory or registers
 - (for data dependencies)
 - **Branch prediction**: guess address of next instruction
 - (for control dependencies)
 - For more info, read CS:APP section 4.5

Conditional moves

- Similar to conditional jumps, but they move data if certain condition codes are set
 - Benefit: no **branch prediction** penalty
 - Improved performance in the presence of pipelining

```
if (a > b) c = d;
```

```
subq  %rbx, %rax  
jle skip  
rrmovq %rdx, %rcx  
skip:
```



```
subq  %rbx, %rax  
cmovg %rdx, %rcx
```

Data (CCs) and control dependencies

No control dependency (only data)

Amdahl's Law

T_s = serial time

$$S = \text{speedup} = \frac{T_s}{T_p}$$

*should
increase
as p grows*

T_p = parallel time

p = # of parallel stages

r = % of logic not amenable to pipelining

$$T_p = \frac{(1-r)T_s}{p} + rT_s$$

$$S = \text{speedup} = \frac{T_s}{\frac{(1-r)T_s}{p} + rT_s}$$

Amdahl's Law: $S \leq \frac{1}{r}$ as p increases

Amdahl's Law

p = # of parallel stages

r = % of logic not amenable to pipelining

Amdahl's Law:

$$S \leq \frac{1}{r} \text{ as } p \text{ increases}$$

$r = 50\%$ → speedup limited to 2x

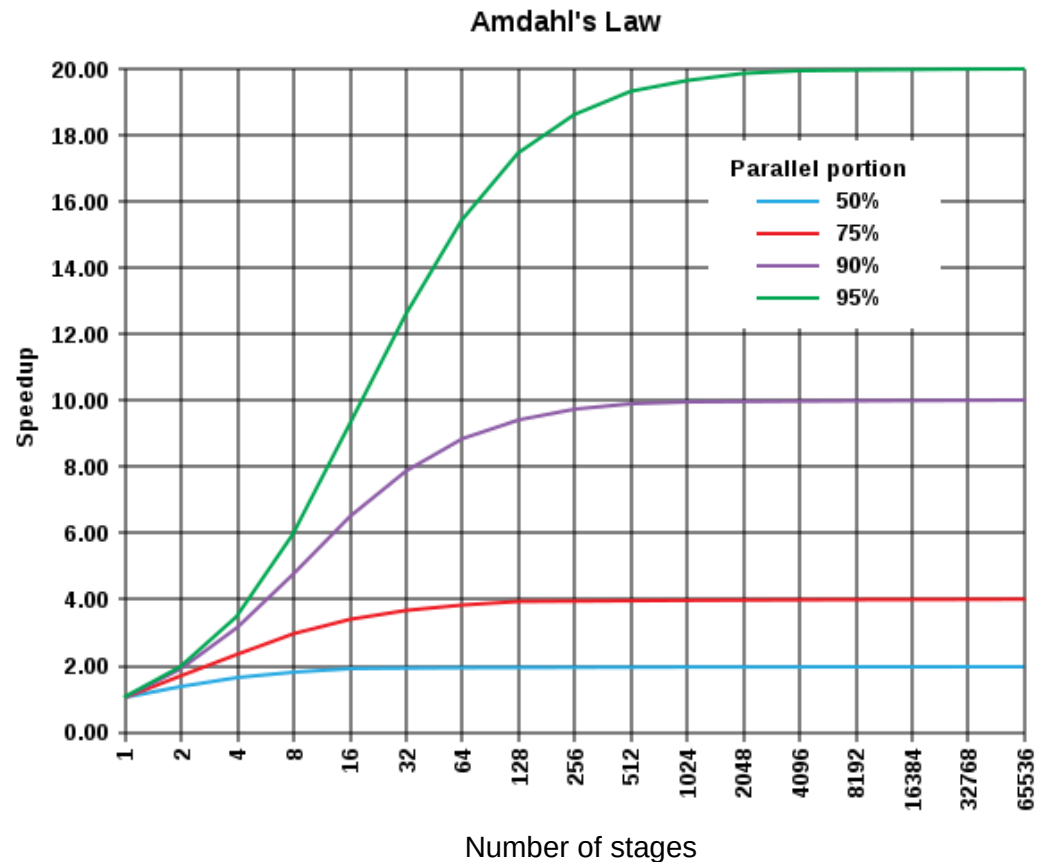
$r = 25\%$ → speedup limited to 4x

$r = 10\%$ → speedup limited to 10x

$r = 5\%$ → speedup limited to 20x

Speedup limited inversely proportionally by serial %

$$S = \text{speedup} = \frac{T_s}{\frac{(1-r)T_s}{p} + rT_s}$$



Summary

- We've now learned how a CPU is constructed
 - Transistors → logic gates → circuits → CPU
 - Pipelining provides instruction-level parallelism
 - Although there are some limitations
- This is not a CPU architecture class
 - We won't be closely studying the specifics of SEQ
 - If you're interested, the details are in section 4.3
 - Same for PIPE (the pipelined version), in section 4.5
 - If you're REALLY interested, plan to take CS 456

CS 456: Architecture

- Course objectives:
 - **Summarize the construction of a pipelined processor from low-level building blocks**
 - Describe and categorize hardware techniques for parallel implementation at the instruction, data, and thread levels
 - Summarize storage and I/O interfacing techniques
 - Apply address decoding and memory hierarchy strategies
 - Evaluate the performance impact of various hardware designs, including caches
 - Describe how hardware implementations can improve overall system performance
 - Justify the use of hardware-based optimizations that fail occasionally
 - Compare and contrast the actual execution of code with software designs
 - Analyze how a person's logical flow of thinking (sequential) differs from the processor implementation
 - Demonstrate the ability to communicate hardware and software design trade-offs to both professional colleagues and laypeople

Lessons learned

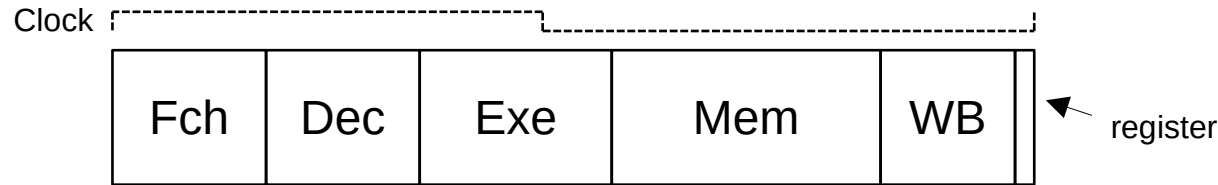
- **Computers are not human**; they're complex machines
 - Machines require extremely precise inputs
 - Machine output can be difficult to interpret
- **Abstraction helps to manage complexity**
 - Use simpler components to build more complex ones
- **System design involves tradeoffs**
 - Simpler ISA vs. ease of coding
 - Throughput vs. latency
- **The details matter (A LOT!)**
 - There are many ways to fail
 - Skill and dedication are required to succeed

Next up

- Y86 architecture and semantics
- Memory architecture and caching
- Final module: operating systems

Lab Diagram

Sequential CPU:



Reminders:

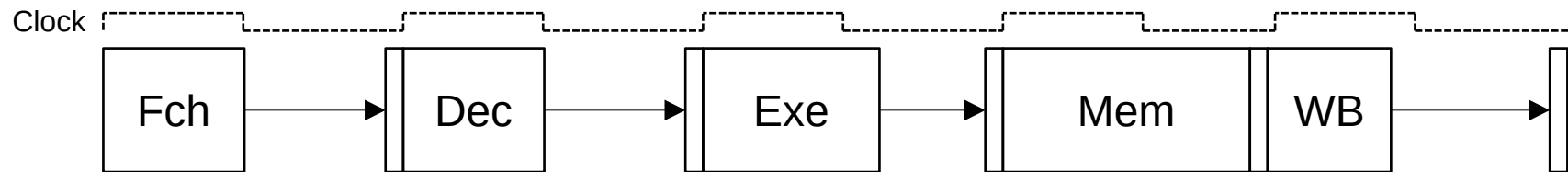
$$\text{Throughput} = \frac{\text{Instruction count}}{\text{Latency}}$$

0.001 instructions/ps = 1 GIPS

mrmovq (%rdi), %rbx	Fch-Dec-Exe-Mem-WB				
addq %rbx, %rax		Fch-Dec-Exe-Mem-WB			
addq %rcx, %rdi			Fch-Dec-Exe-Mem-WB		
subq %rdx, %rsi				Fch-Dec-Exe-Mem-WB	
jne loop					Fch-Dec-Exe-Mem-WB

Time →

Pipelined CPU:



mrmovq (%rdi), %rbx	Fch	Dec							
addq %rbx, %rax		Fch							
addq %rcx, %rdi									
subq %rdx, %rsi									
jne loop									

Time →

(UNFINISHED - #3 on lab)