

CS 261

Fall 2016

Mike Lam, Professor

CPU architecture

Topics

- CPU stages
- Y86 CPU design
- Pipelining

CPU stages

1) Fetch ← P3!

- Splits instruction at PC into pieces

2) Decode (register file)

- Reads registers
- P4: Sets **valA**

3) Execute (ALU)

- Arithmetic/logic operation, effective address calculation, or stack pointer increment/decrement
- P4: Sets **valE** and **Cnd**

4) Memory (RAM)

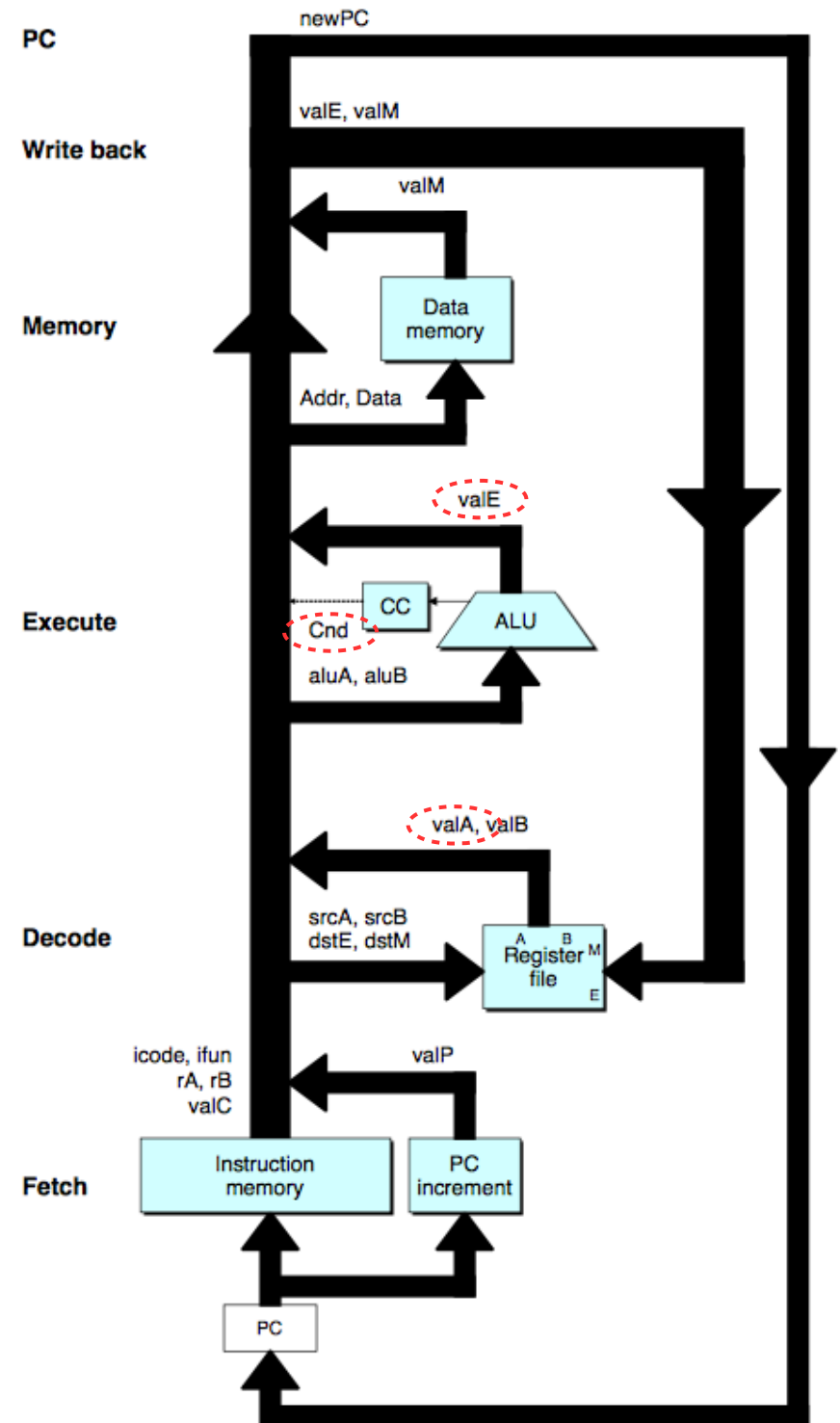
- Reads/writes memory

5) Write back (register file)

- Sets registers

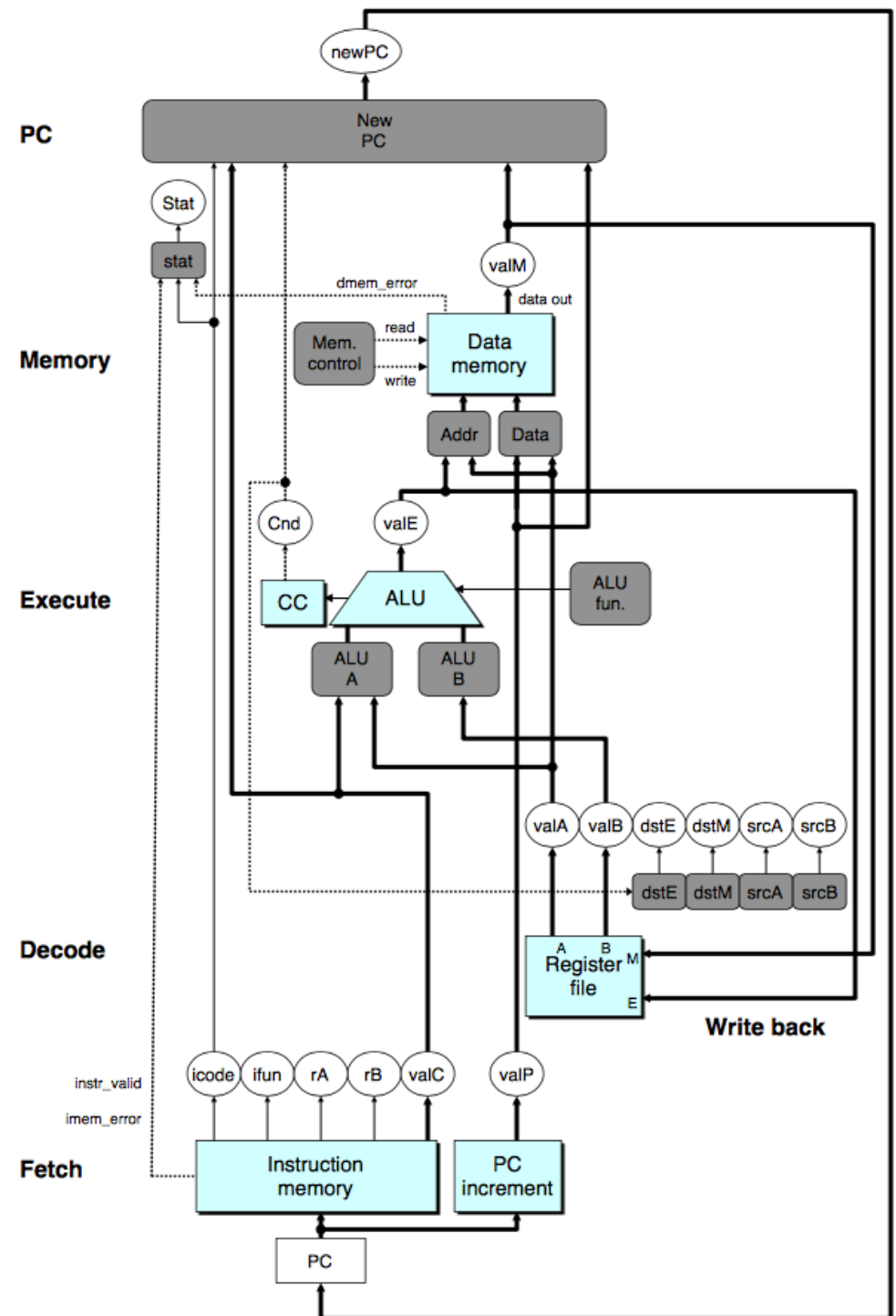
6) PC update

- Sets new PC

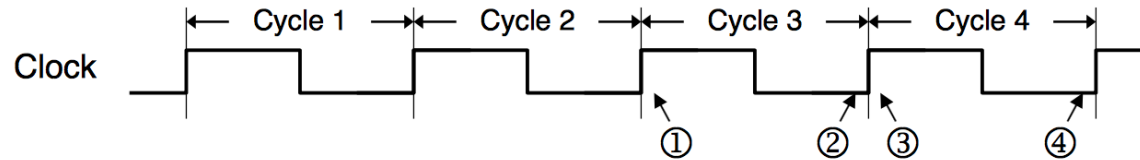


CPU design

- **SEQ**: sequential Y86 CPU
 - Runs one instruction at a time
 - `ssim`: 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)
- Principle: no reading back
 - Stages run simultaneously
 - Effects remain internally consistent

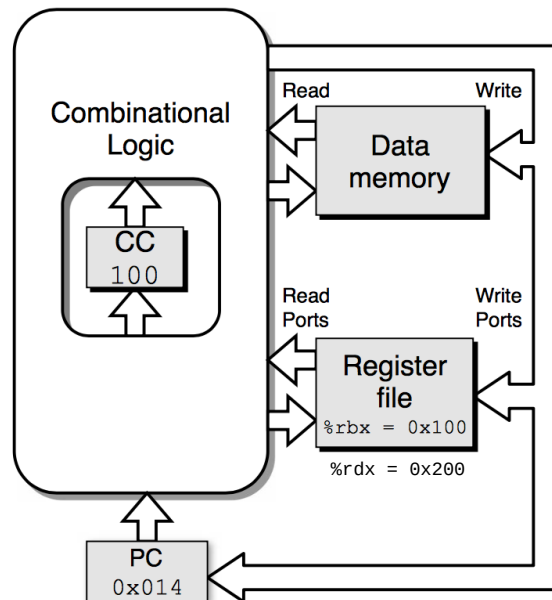


Example

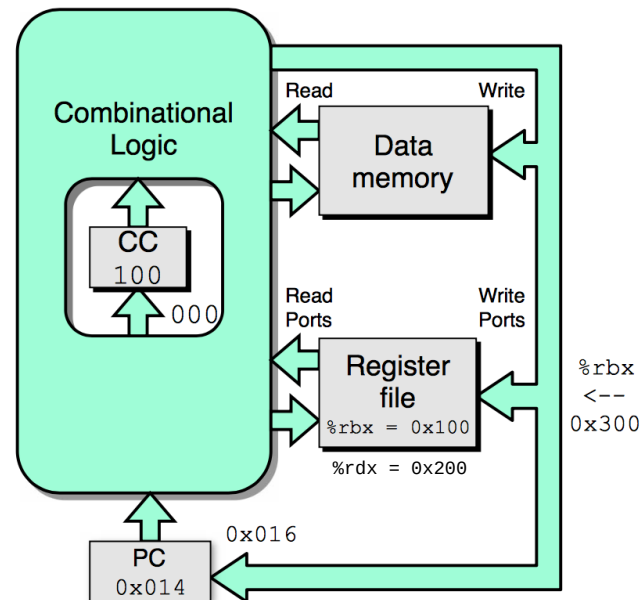


Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

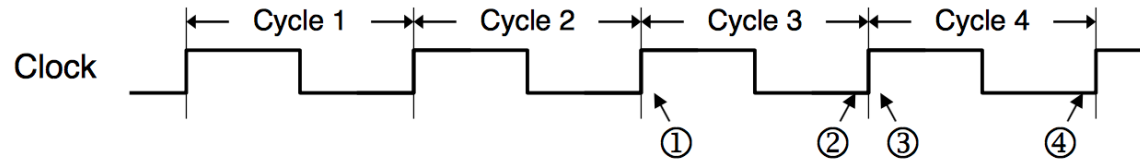
① Beginning of cycle 3



② End of cycle 3

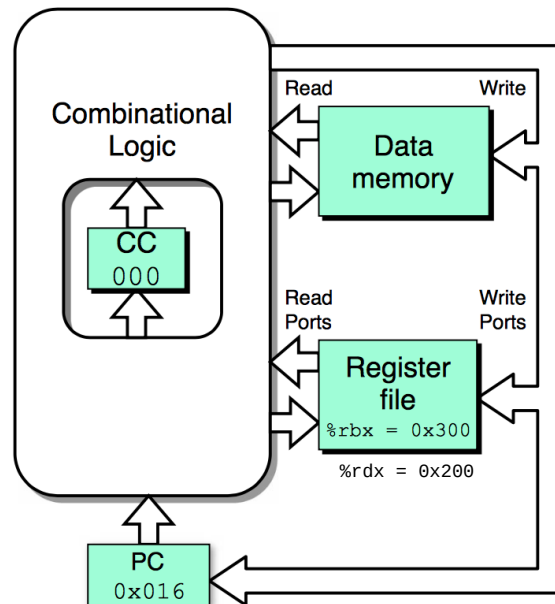


Example

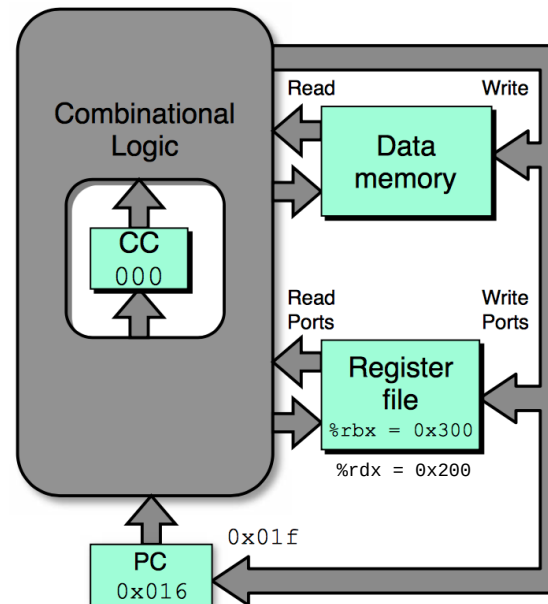


Cycle 1:	0x000:	irmovq \$0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq \$0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

③ Beginning of cycle 4



④ End of cycle 4

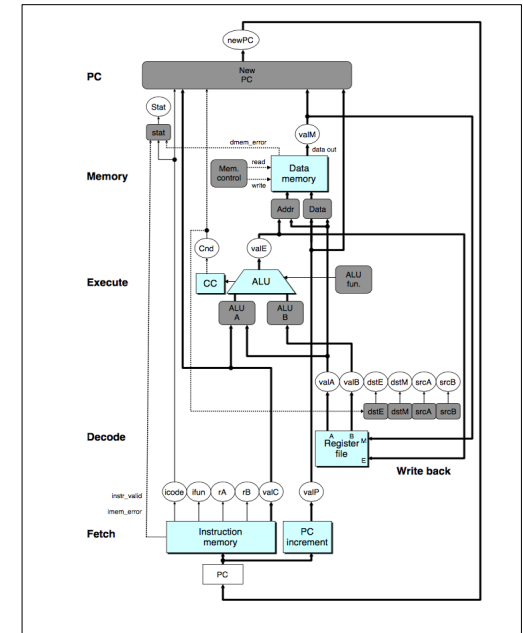


System design

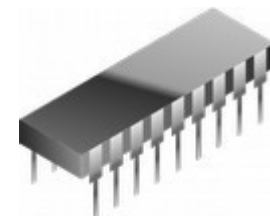
- CPU measurement
 - **Throughput**: instructions executed per second
 - GIPS: billions of (“giga-”) instructions per second
 - 1 GIPS → each instruction takes 1 nanosecond (a billionth of a second)
 - **Latency / delay**: time required per instruction
 - Picosecond: 10^{-12} seconds Nanosecond: 10^{-9} 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
- One approach: make it smaller
 - Shorter circuit = faster circuit
 - Limited by manufacturing technology

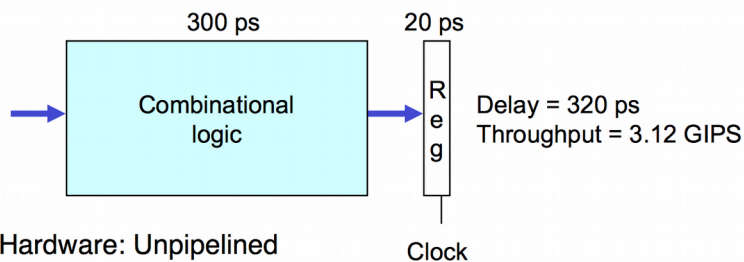


What else could we do?



System design

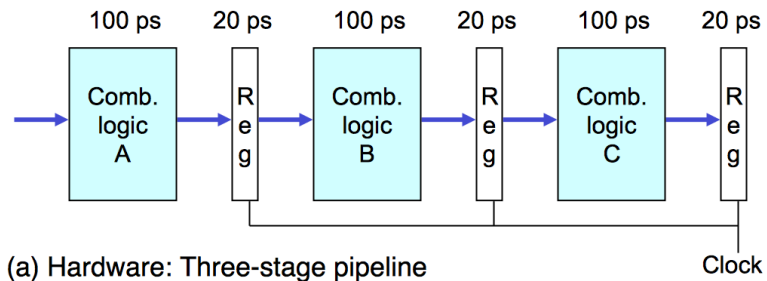
- Idea: **pipelined** design
 - Multiple instructions execute simultaneously (“**instruction-level parallelism**”)
 - Similar to cafeteria line or car wash
 - Split logic into stages and connect stages with clocked registers
 - System design tradeoff: **throughput vs. latency**



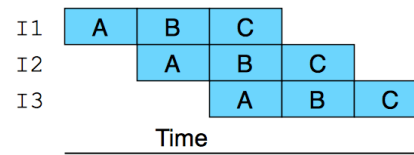
(a) Hardware: Unpipelined



(b) Pipeline diagram



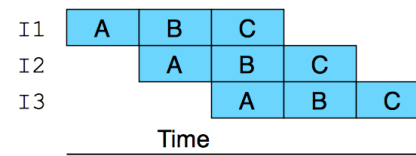
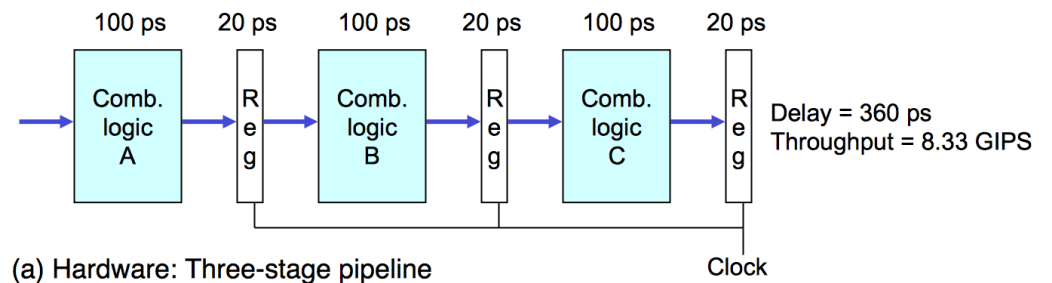
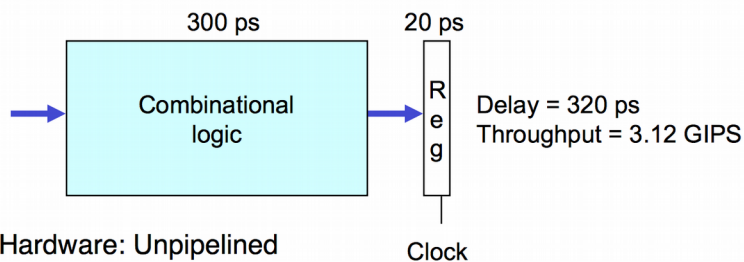
(a) Hardware: Three-stage pipeline



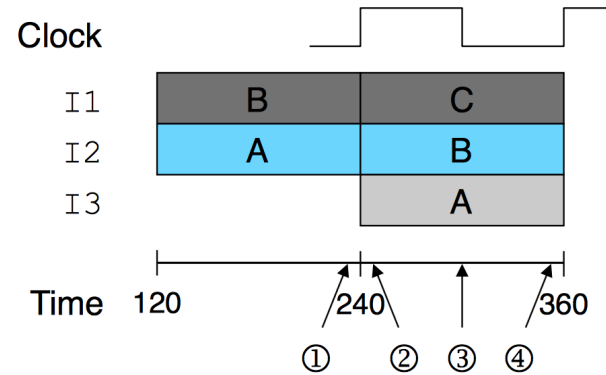
(b) Pipeline diagram

System design

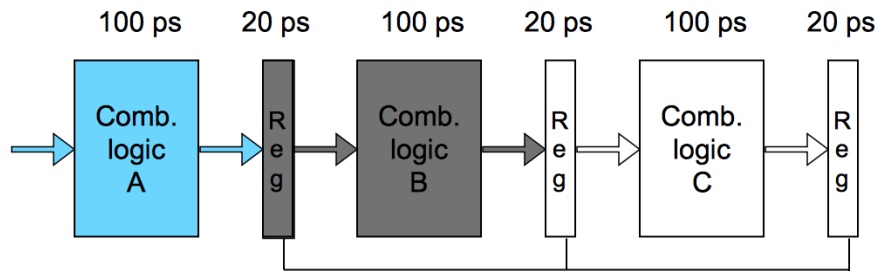
- Idea: **pipelined** design
 - Multiple instructions execute simultaneously (“**instruction-level parallelism**”)
 - Similar to cafeteria line or car wash
 - Split logic into stages and connect stages with clocked registers
 - System design tradeoff: **throughput vs. latency**



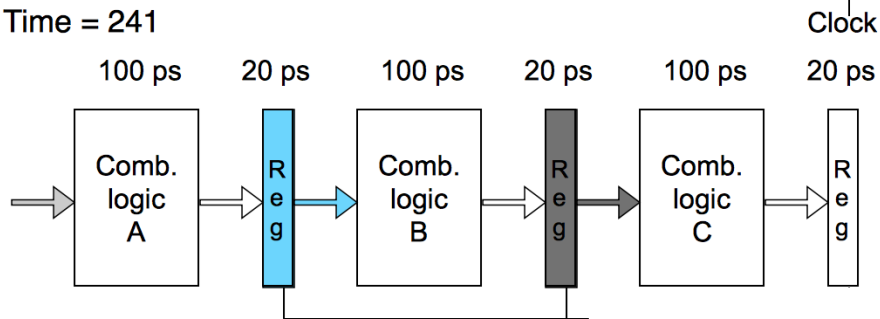
Pipelining example



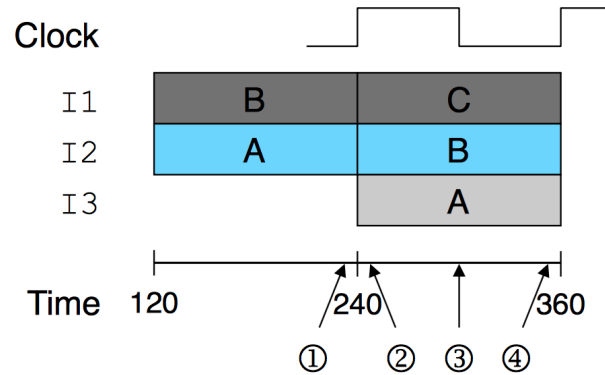
① Time = 239



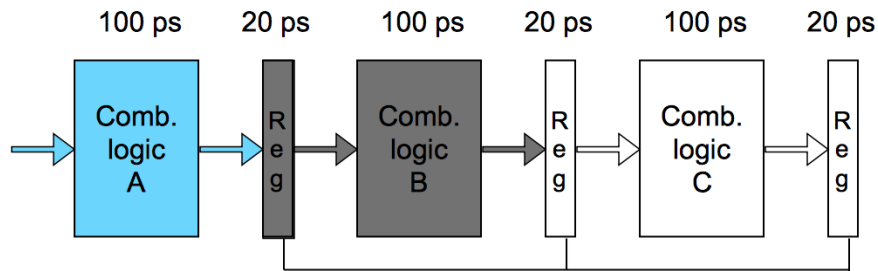
② Time = 241



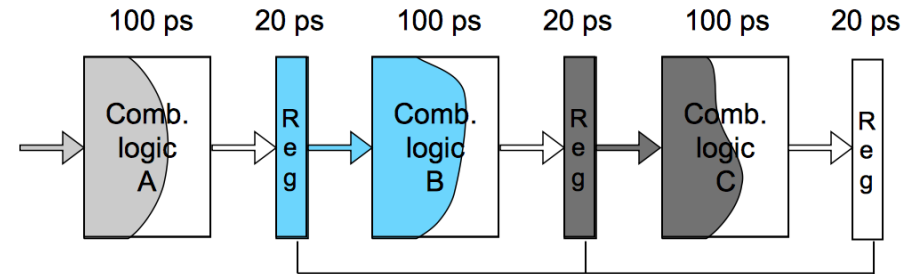
Pipelining example



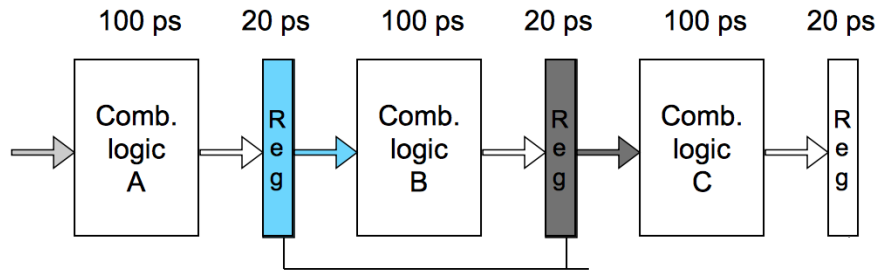
① Time = 239



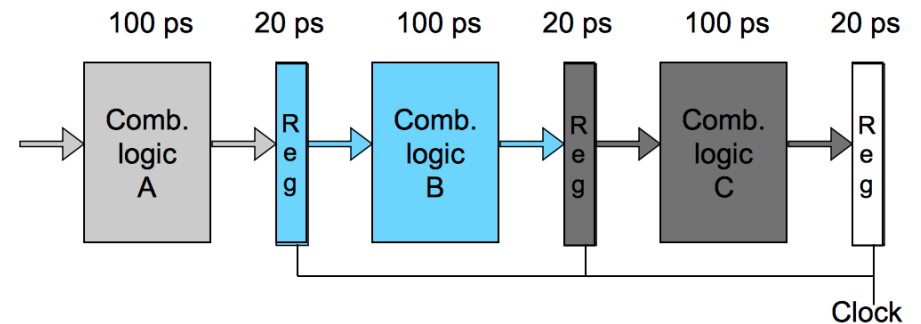
③ Time = 300



② Time = 241

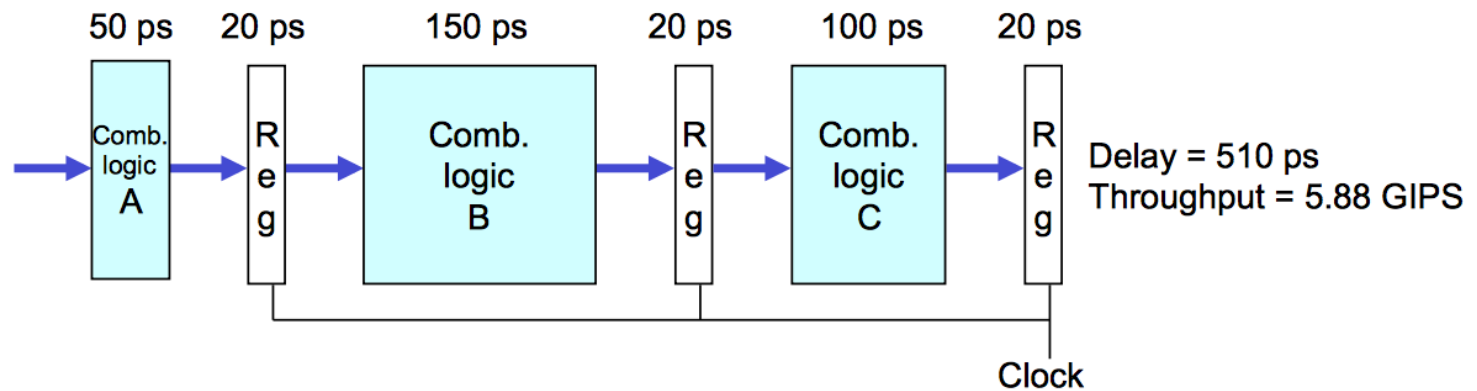


④ Time = 359

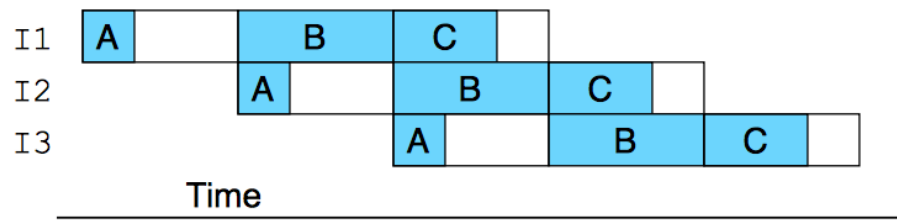


Pipelining

- 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:

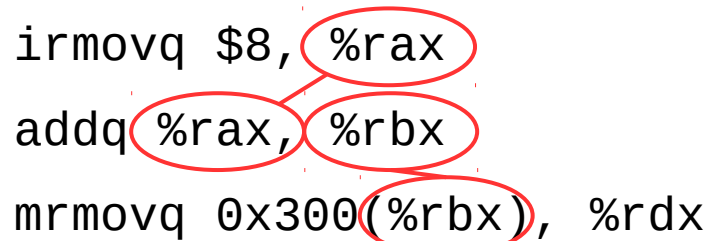
```
loop:
    subq %rdx, %rbx
    jne loop
    irmovq $10, %rdx
```

Pipelining

- 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:

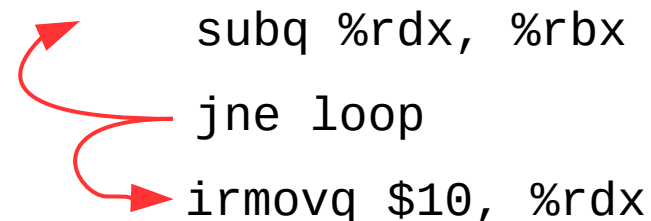
```
irmovq $8, %rax
addq %rax, %rbx
mrmovq 0x300(%rbx), %rdx
```



Control dependency:

```
loop:
```

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



Pipelining

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

Summary

- We've now learned how a CPU is constructed
 - Transistors → logic gates → circuits → CPU
 - Pipelining provides instruction-level parallelism
- 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, lobby for CS 456

CS 456: Architecture

- Course objectives:
 - Describe the construction of a pipelined CPU from low-level components
 - Describe hardware techniques for parallelism at various levels
 - Summarize storage and I/O interfacing techniques
 - Apply address decoding and memory hierarchy strategies
 - Evaluate the performance impact of cache designs
 - Implement custom hardware designs in an FPGA
 - Justify the use of hardware-based optimization that fails occasionally
 - Develop a sense for the challenges of hardware debugging

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 time: Y86 semantics

Stage	HALT	NOP	CMOV	IRMOVQ
Fch	$\text{icode} \leftarrow M_1[\text{PC}]$	$\text{icode} \leftarrow M_1[\text{PC}]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC} + 10$
Dec		$\text{valP} \leftarrow \text{PC} + 1$	$\text{valP} \leftarrow \text{PC} + 2$ $\text{valA} \leftarrow R[\text{rA}]$	
Exe	$\text{cpu.stat} = \text{HLT}$		$\text{valE} \leftarrow \text{valA}$ $\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$	$\text{valE} \leftarrow \text{valC}$
Mem				
WB			$\text{Cnd} ? R[\text{rB}] \leftarrow \text{valE}$	$R[\text{rB}] \leftarrow \text{valE}$
PC	$\text{PC} \leftarrow 0$	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$
Stage	RMMOVQ	MRMOVQ	OPq	JUMP
Fch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$ $\text{valC} \leftarrow M_8[\text{PC}+2]$ $\text{valP} \leftarrow \text{PC} + 10$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 9$
Dec	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	$\text{valB} \leftarrow R[\text{rB}]$	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{rB}]$	
Exe	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB} + \text{valC}$	$\text{valE} \leftarrow \text{valB OP valA}$	$\text{Cnd} \leftarrow \text{Cond}(\text{CC}, \text{ifun})$
Mem	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valE}]$		
WB		$R[\text{rA}] \leftarrow \text{valM}$	$R[\text{rB}] \leftarrow \text{valE}$	
PC	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{Cnd?valC:valP}$
Stage	CALL	RET	PUSHQ	POPQ
Fch	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$	$\text{icode:ifun} \leftarrow M_1[\text{PC}]$ $\text{rA:rB} \leftarrow M_1[\text{PC}+1]$
Dec	$\text{valC} \leftarrow M_8[\text{PC}+1]$ $\text{valP} \leftarrow \text{PC} + 9$	$\text{valP} \leftarrow \text{PC} + 1$	$\text{valP} \leftarrow \text{PC} + 2$	$\text{valP} \leftarrow \text{PC} + 2$
Exe	$\text{valB} \leftarrow R[\text{RSP}]$ $\text{valE} \leftarrow \text{valB} - 8$	$\text{valA} \leftarrow R[\text{RSP}]$ $\text{valB} \leftarrow R[\text{RSP}]$ $\text{valE} \leftarrow \text{valB} + 8$	$\text{valA} \leftarrow R[\text{rA}]$ $\text{valB} \leftarrow R[\text{RSP}]$ $\text{valE} \leftarrow \text{valB} - 8$	$\text{valA} \leftarrow R[\text{RSP}]$ $\text{valB} \leftarrow R[\text{RSP}]$ $\text{valE} \leftarrow \text{valB} + 8$
Mem	$M_8[\text{valE}] \leftarrow \text{valP}$	$\text{valM} \leftarrow M_8[\text{valA}]$	$M_8[\text{valE}] \leftarrow \text{valA}$	$\text{valM} \leftarrow M_8[\text{valA}]$
WB	$R[\text{RSP}] \leftarrow \text{valE}$	$R[\text{RSP}] \leftarrow \text{valE}$	$R[\text{RSP}] \leftarrow \text{valE}$	$R[\text{RSP}] \leftarrow \text{valE}$ $R[\text{rA}] \leftarrow \text{valM}$
PC	$\text{PC} \leftarrow \text{valC}$	$\text{PC} \leftarrow \text{valM}$	$\text{PC} \leftarrow \text{valP}$	$\text{PC} \leftarrow \text{valP}$