CSC 252: Computer Organization Spring 2018: Lecture 12

Instructor: Yuhao Zhu

Department of Computer Science
University of Rochester

Action Items:

- **Assignment 3 is due March 2, midnight**
- **Mid-term: March 8**

Announcement

- Programming Assignment 3 is due on March 2, midnight
- Mid-term exam: March 8; in class
- Prof. Scott has some past exams posted: https://www.cs.rochester.edu/courses/252/spring2014/resources.shtml

Sun	Mon	Tue	Wed	Thu	Fri	Sat
25	26	27	28	Mar 1	2	3
		Lecture		**Lecture**	**A3 due**	
4	5	6	7	8	9	10
		Lecture		**Midterm**		

Announcement

- Another faculty candidate talk
- Tomorrow, noon, Goergen 109, with food

Wednesday, February 28, 2018
12:00 PM
Goergen 109

Michelle Ichinco
Washington University in St. Louis

Supporting novices in learning programming on-the-fly using examples

Many people, including children, begin learning programming independently in open-ended contexts. This prevents them from receiving feedback that would introduce them to new skills. In this talk, I will present a system called the Example Guru, which suggests new skills to novice programmers using example code. Both in lab studies and the wild, novices chose to access suggestions significantly more often than common forms of support, like documentation or tutorials. Accessing suggestions often led to more use of new code. I will also discuss my approach for semi-automatically generating suggestions and examples. This approach generated both a set of suggestions similar to an expert hand-authored set, as well as an additional set of original suggestions. This type of support for independent novice programmers has the potential to significantly help the large population of non-

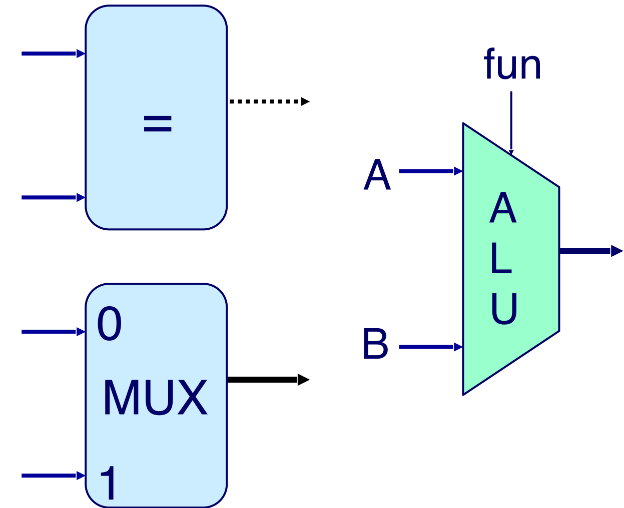
Today: Processor Microarchitecture

- Sequential, single-cycle microarchitecture implementation
  - Basic idea
  - Hardware implementation
- Pipelined microarchitecture implementation
  - Basic Principles
  - Difficulties: Control Dependency
  - Difficulties: Data Dependency

Building Blocks

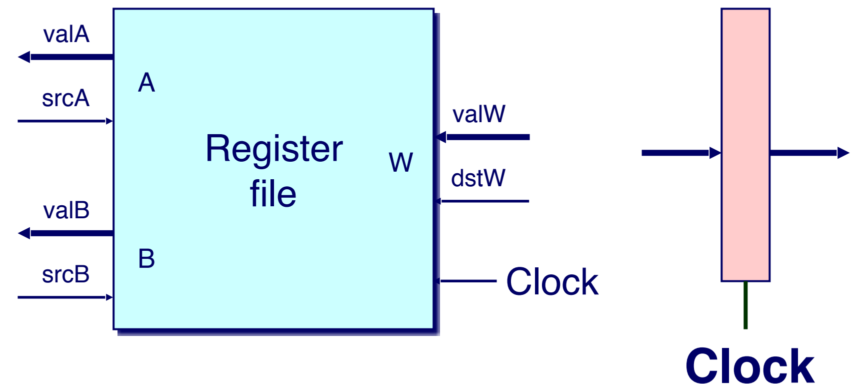
Combinational Logic

- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control



Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises



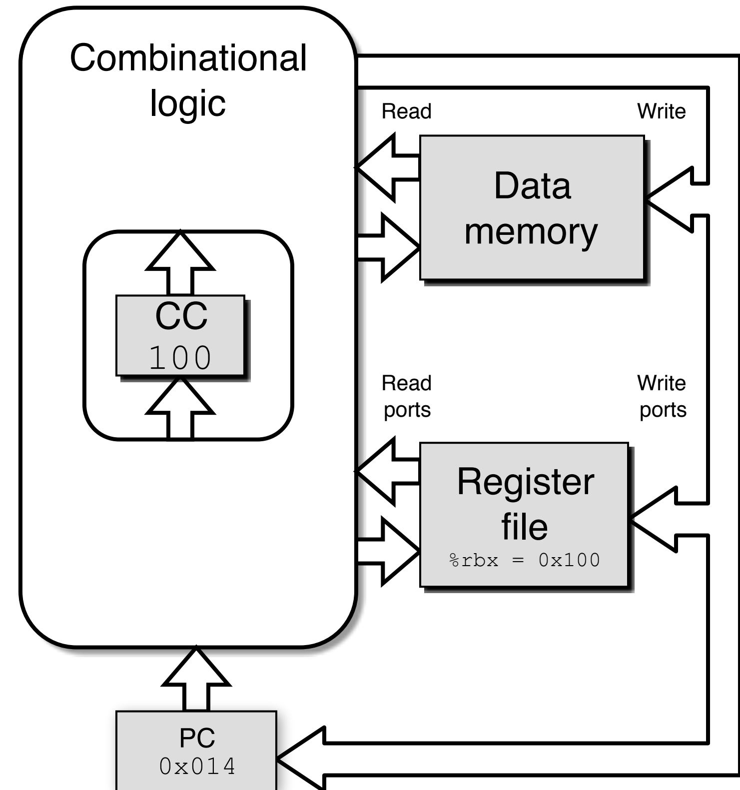
Microarchitecture Overview

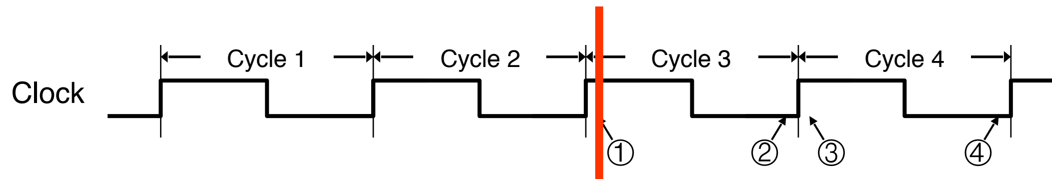
Combinational Logic

- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

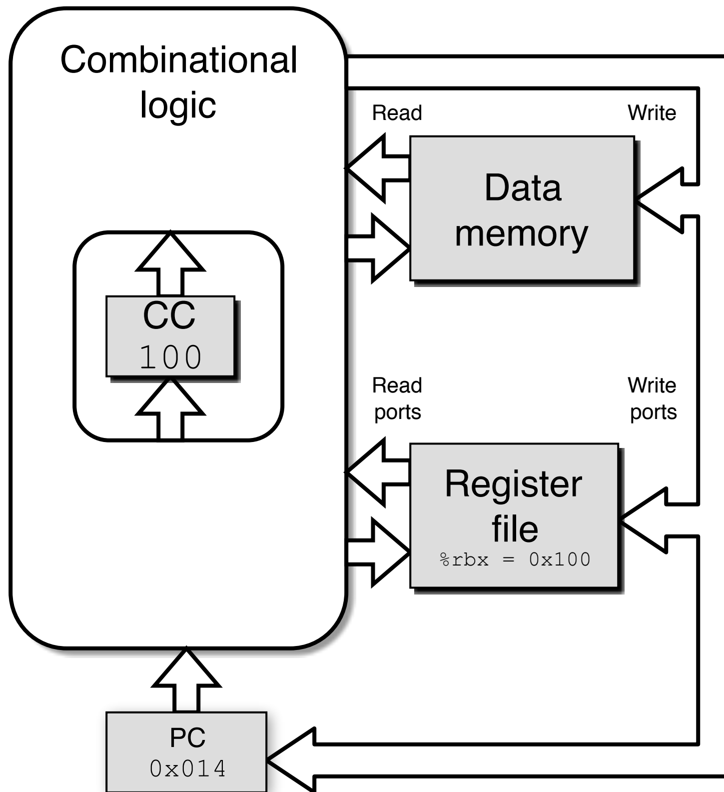
Storage (All updated as clock rises)

- PC register
- Cond. Code register
- Data memory
- Register file

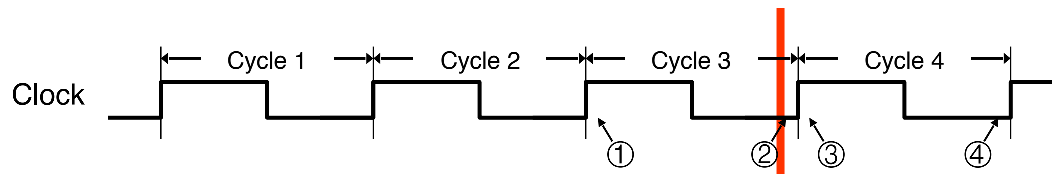




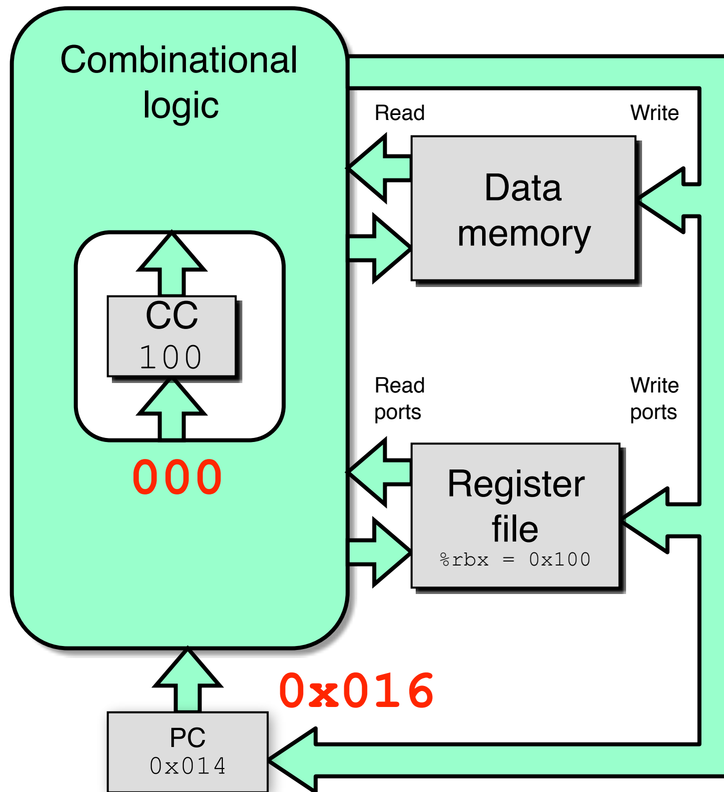
Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

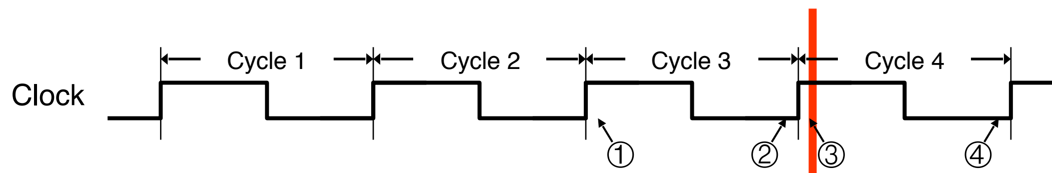


Cycle 1:	0x000: irmovq $0x100,%rbx # %rbx <-- 0x100
Cycle 2:	0x00a: irmovq $0x200,%rdx # %rdx <-- 0x200
Cycle 3:	0x014: addq %rdx,%rbx # %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016: je dest # Not taken
Cycle 5:	0x01f: rmmovq %rbx,0(%rdx) # M[0x200] <-- 0x300

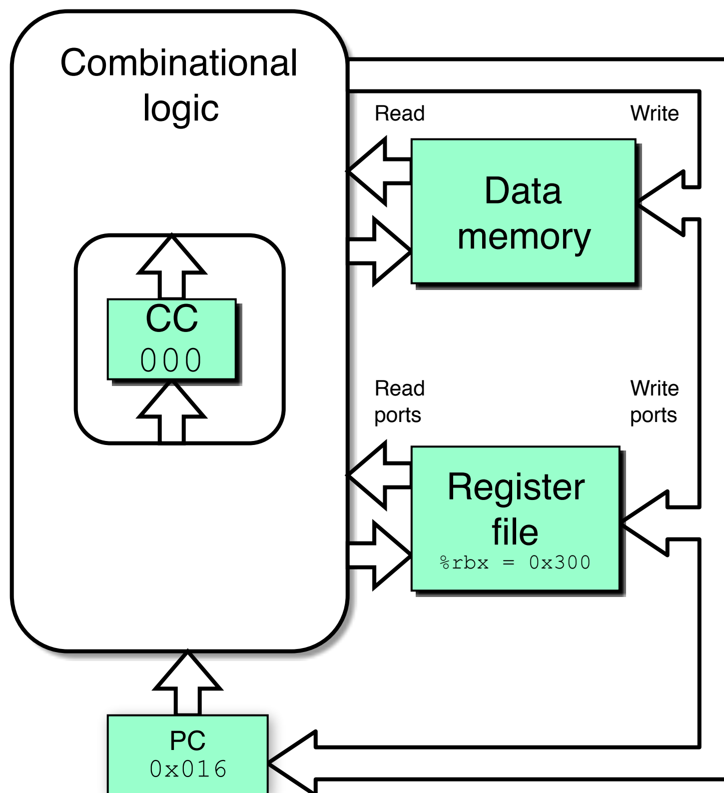


%rbx <-- 0x300

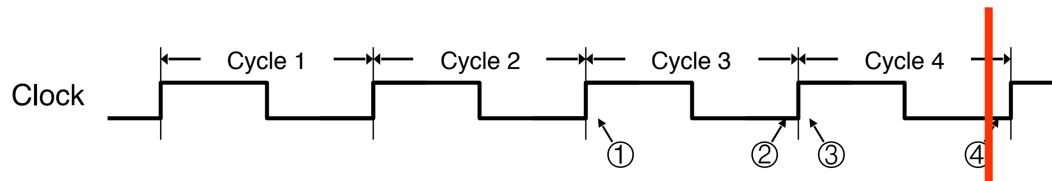
- state set according to second `irmovq` instruction
- combinational logic generates results for addq instruction



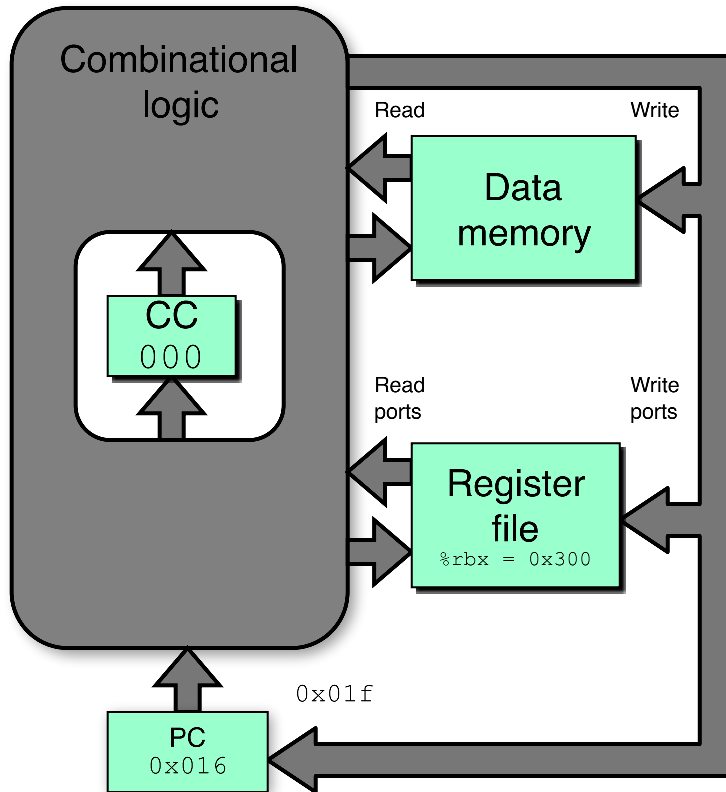
Cycle	Address	Instruction	Comment
Cycle 1:	`0x000:`	`irmovq $0x100,%rbx`	`# %rbx <-- 0x100`
Cycle 2:	`0x00a:`	`irmovq $0x200,%rdx`	`# %rdx <-- 0x200`
Cycle 3:	`0x014:`	`addq %rdx,%rbx`	`# %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:`	`je dest`	`# Not taken`
Cycle 5:	`0x01f:`	`rmmovq %rbx,0(%rdx)`	`# M[0x200]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



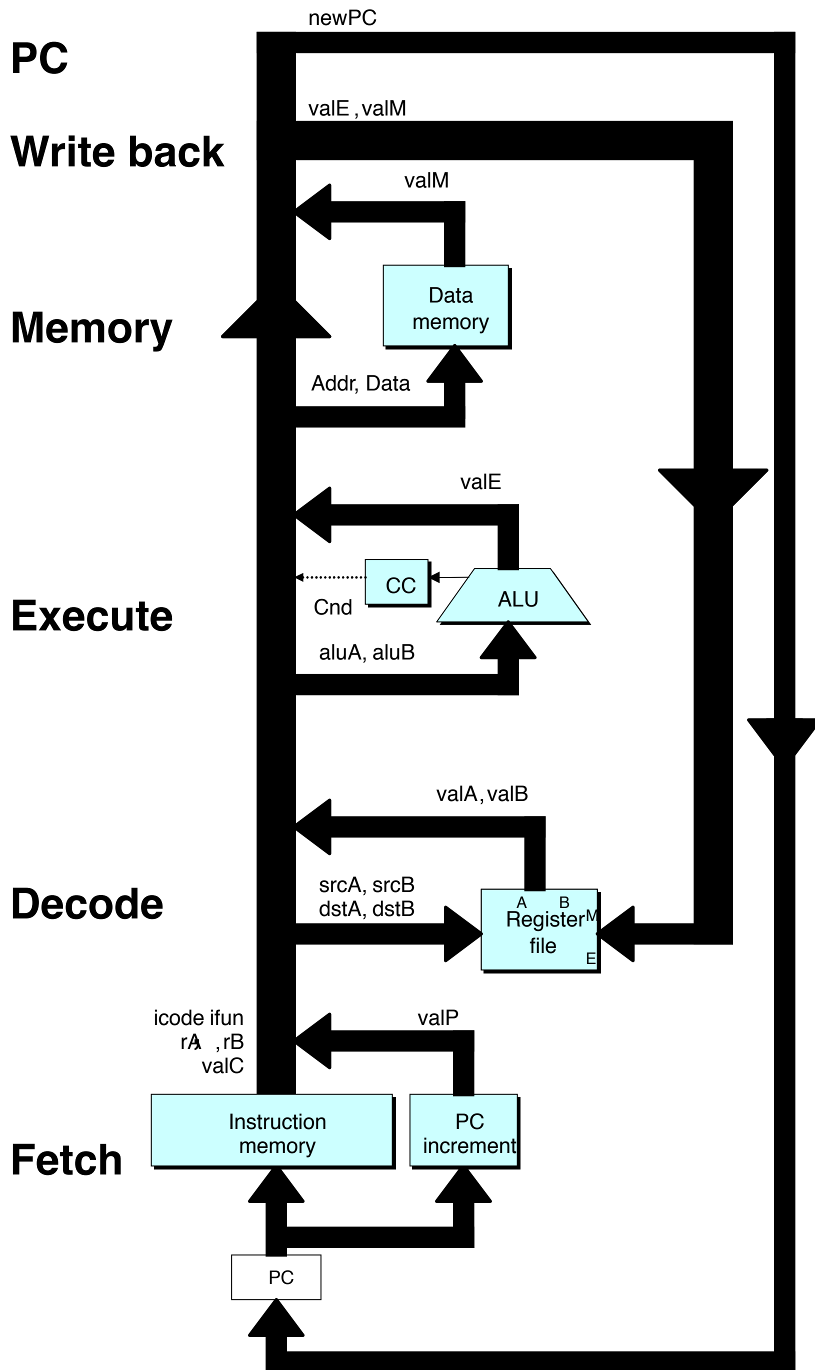
- state set according to `addq` instruction
- combinational logic starting to react to state changes



Cycle	Address	Instruction	Comment
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



- state set according to `addq` instruction
- combinational logic generates results for `je` instruction



Fetch

- Read instruction from instruction memory

Decode

- Read program registers

Execute

- Compute value or address

Memory

- Read or write data

Write Back

- Write program registers

PC

- Update program counter

PC

Write back

Memory

Execute

Decode

Fetch

newPC

valE , valM

valM

Data memory

Addr, Data

valE

CC

ALU

Cnd

aluA, aluB

valA, valB

srcA, srcB
dstA, dstB

A B Register file M E

icode ifun
rA , rB
valC

valP

Instruction memory

PC increment

PC

PC

Memory

Execute

Decode

Fetch

Write back

newPC

New PC

Stat

stat

valM

dmem_error

data out

Mem. control

read

write

Data memory

Addr

Data

Cnd

valE

CC

ALU

ALU fun.

ALU A

ALU B

valA valB dstE dstM srcA srcB

dstE dstM srcA srcB

A B Register file M E

icode ifun rA rB valC valP

instr_valid

imem_error

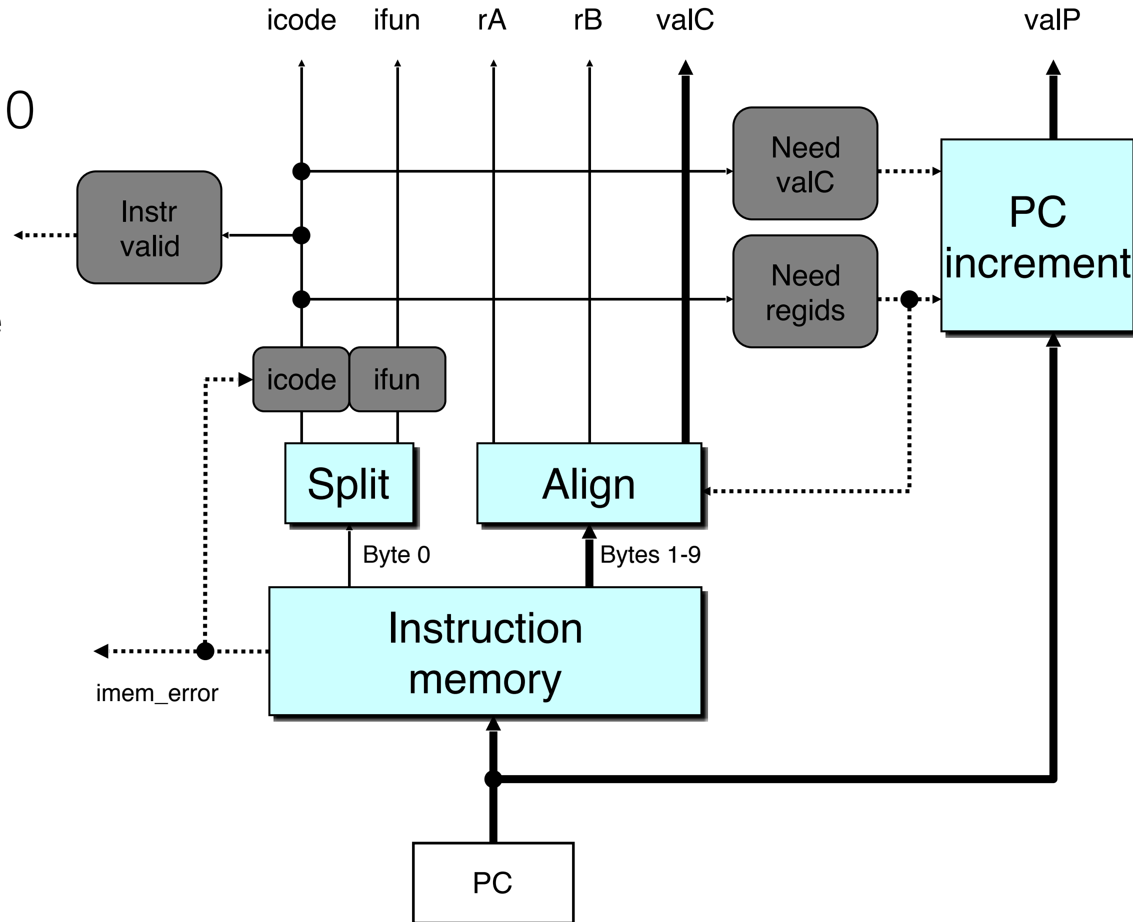
Instruction memory

PC increment

PC

Fetch Logic

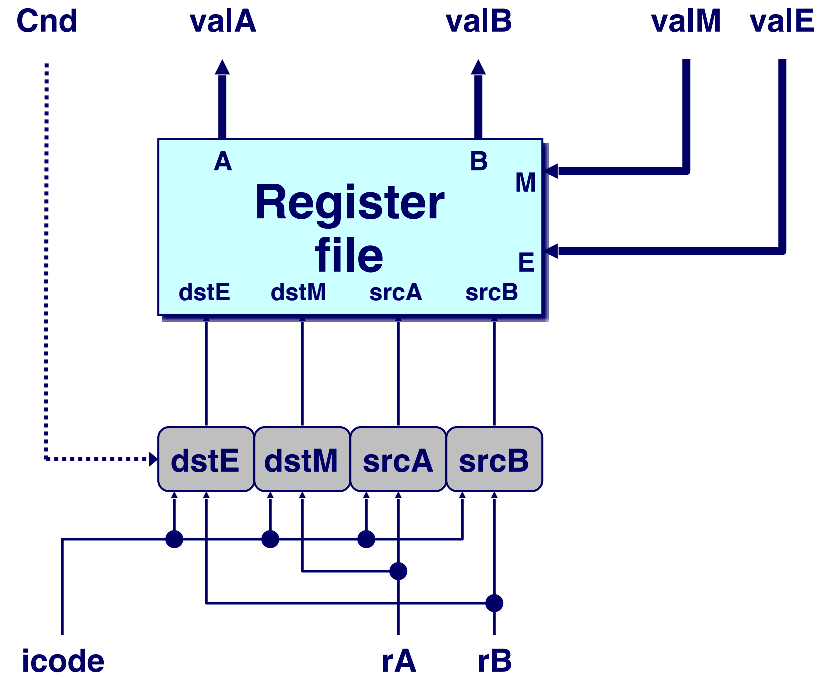
- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
  - Signal invalid address
- Split: Divide instruction byte into icode and ifun
- Align: Get fields for rA, rB, and valC



Decode Logic

Register File

- Read ports A, B
- Write ports E, M
- Addresses are register IDs (0~15)



Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses
- icode tells us whether the register file needs to be accessed

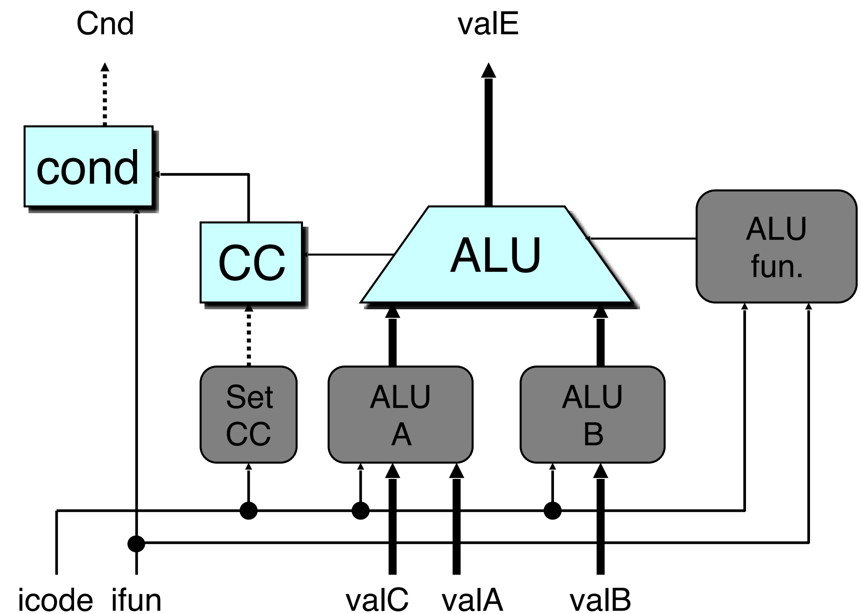
Execute Logic

Units

- ALU
  - Implements 4 required functions
  - Generates condition code values
- CC
  - Register with 3 condition code bits
- cond
  - Computes conditional jump/move flag

Control Logic

- Set CC: Should condition code register be loaded?
- ALU A: Input A to ALU
- ALU B: Input B to ALU
- ALU fun: What function should ALU compute?



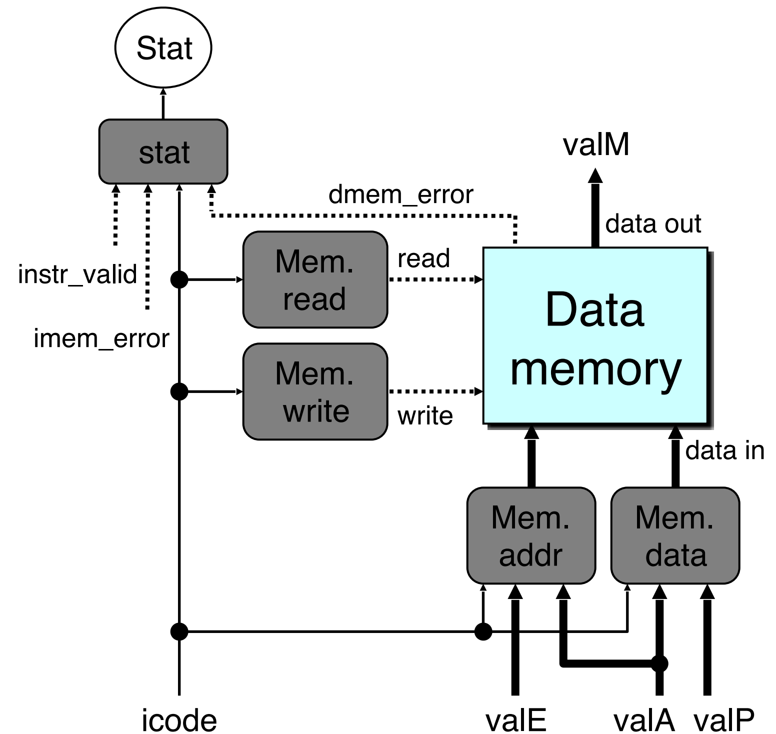
Memory Logic

Memory

- Reads or writes memory word

Control Logic

- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



PC Update Logic: Select Next PC

	OPq rA, rB
PC update	PC ← valP

Update PC

	`rmmovq` rA, D(rB)
PC update	PC ← valP

Update PC

	`popq` rA
PC update	PC ← valP

Update PC

	jXX Dest
PC update	PC ← Cnd ? valC : valP

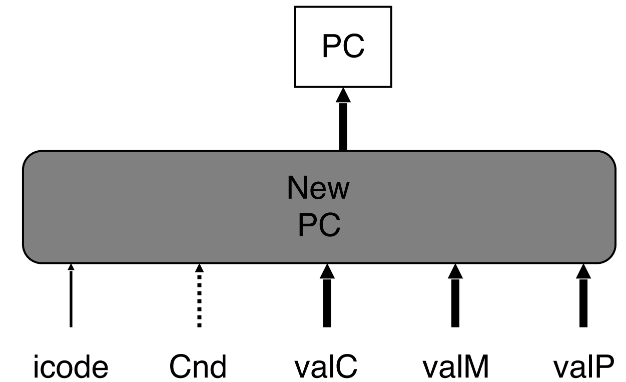
Update PC

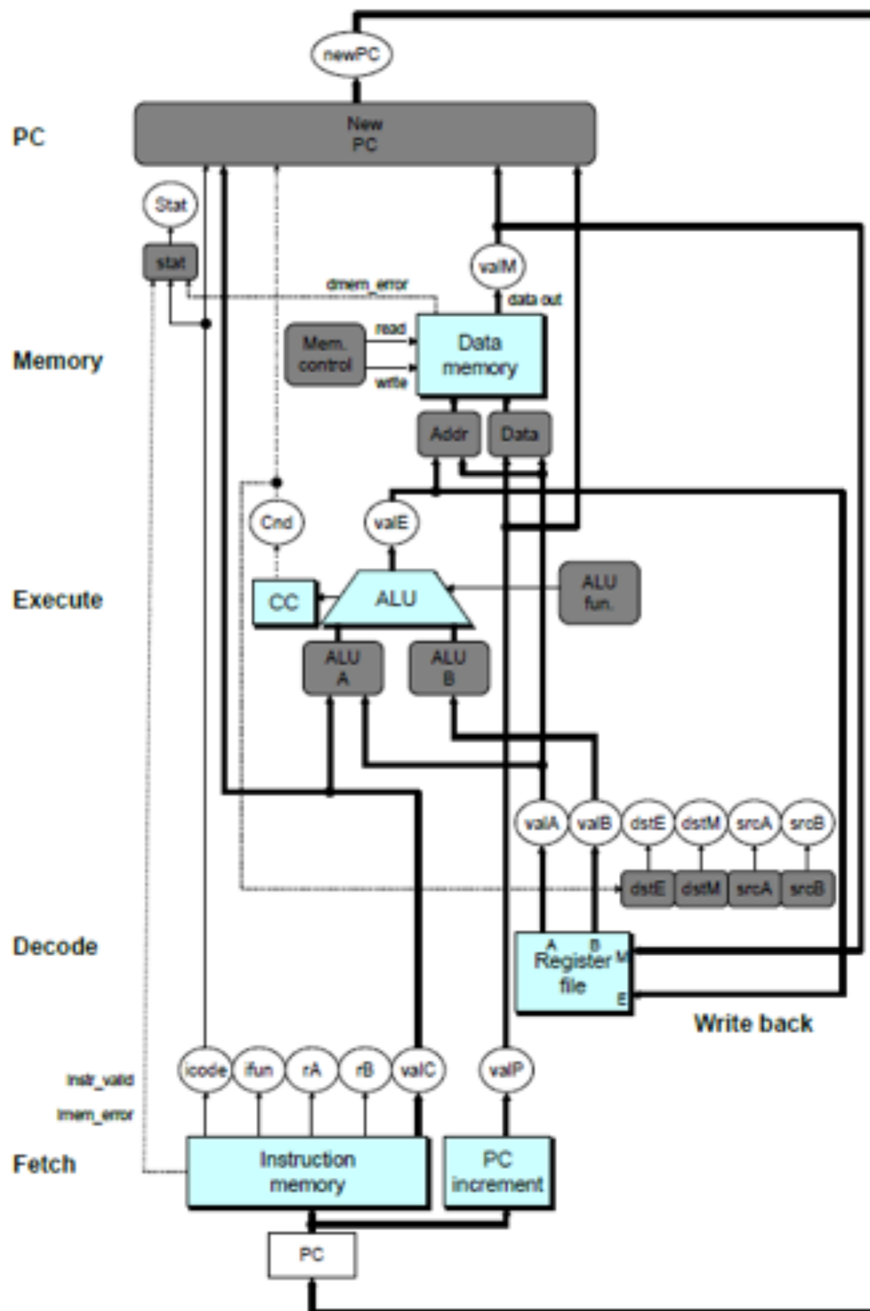
	`call` Dest
PC update	PC ← valC

Set PC to destination

	`ret`
PC update	PC ← valM

Set PC to return address




newPC
PC
New PC
Stat
stat
dmem_error
valM
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
icode
ifun
rA
rB
valC
valP
instr_valid
imem_error
Fetch
Instruction memory
PC increment
PC

Today: Processor Microarchitecture

- Sequential, single-cycle microarchitecture implementation
  - Basic idea
  - Hardware implementation
- Pipelined microarchitecture implementation
  - Basic Principles
  - Difficulties: Control Dependency
  - Difficulties: Data Dependency

Real-World Pipelines: Car Washes

Real-World Pipelines: Car Washes

Sequential

Real-World Pipelines: Car Washes

Sequential

Pipelined

Real-World Pipelines: Car Washes

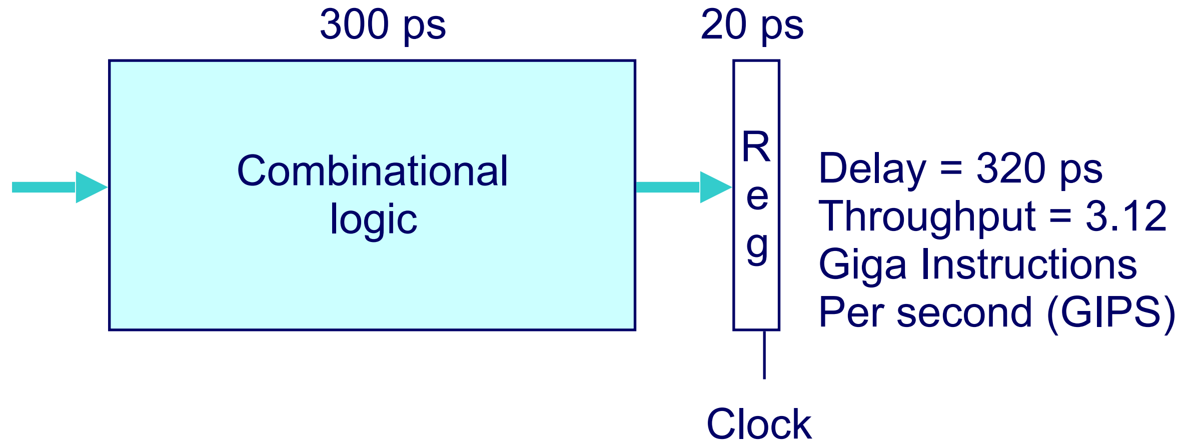
Sequential

Pipelined

Idea

- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

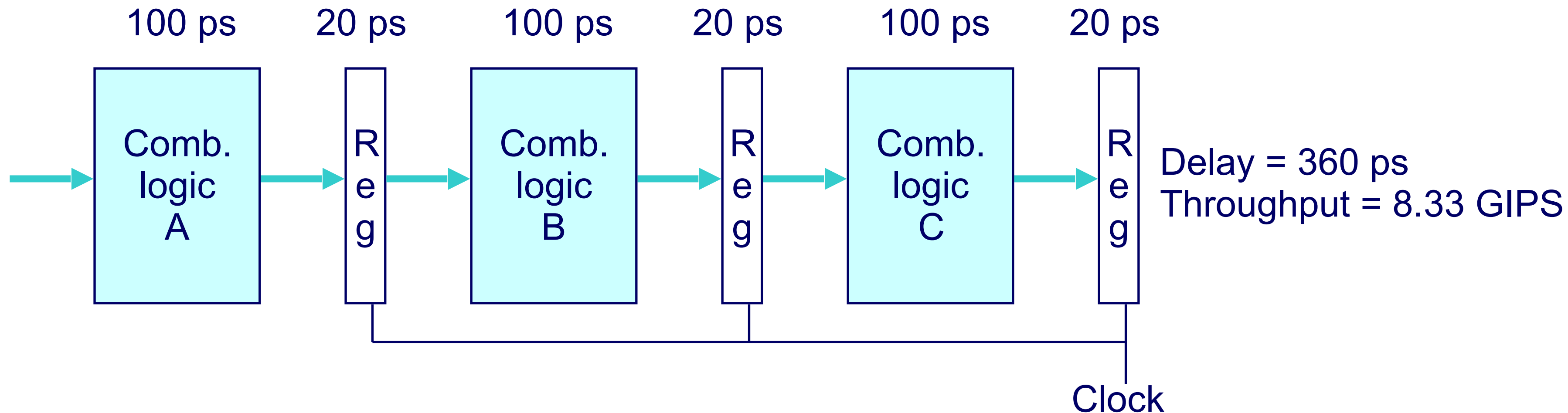
Computational Example



System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle time of at least 320 ps

3-Stage Pipelined Version



System

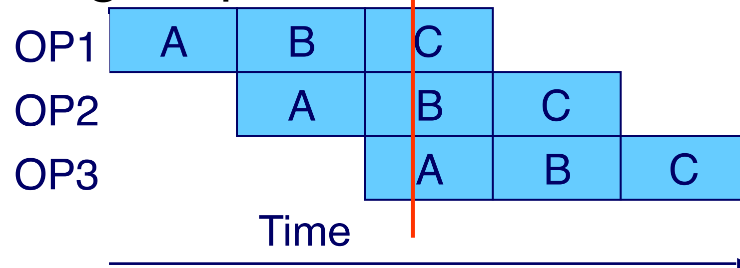
- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish

Pipeline Diagrams

Unpipelined

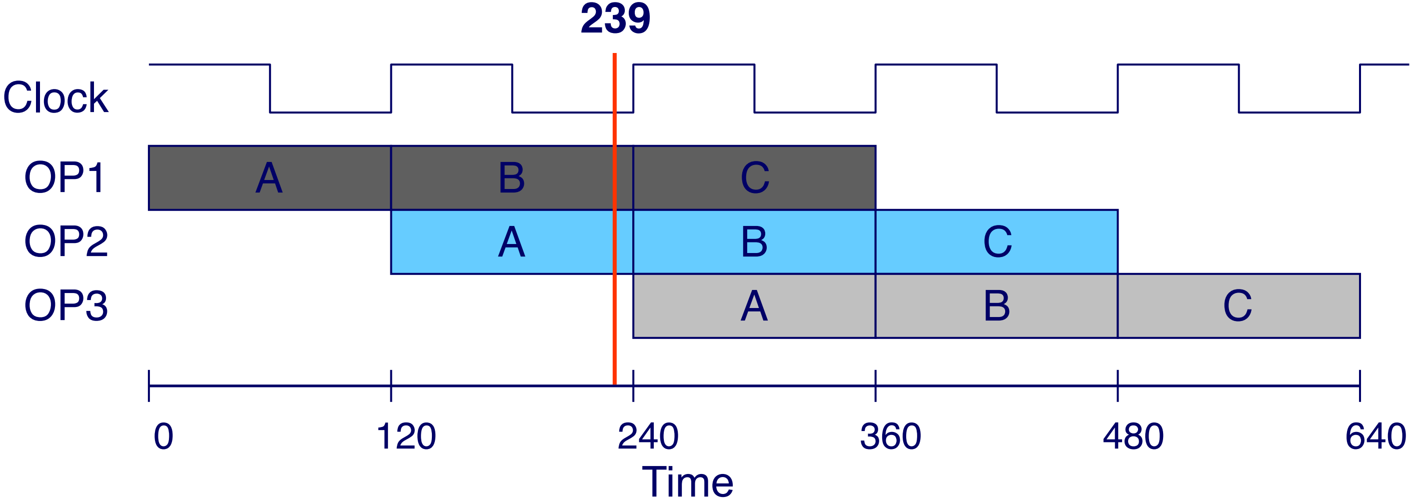
- Cannot start new operation until previous one completes

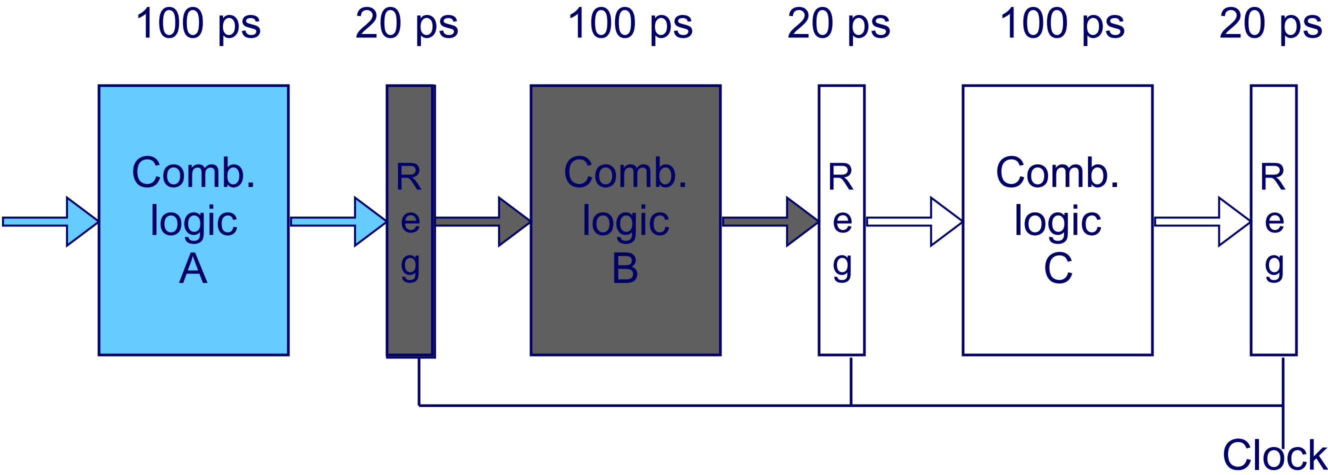
3-Stage Pipelined



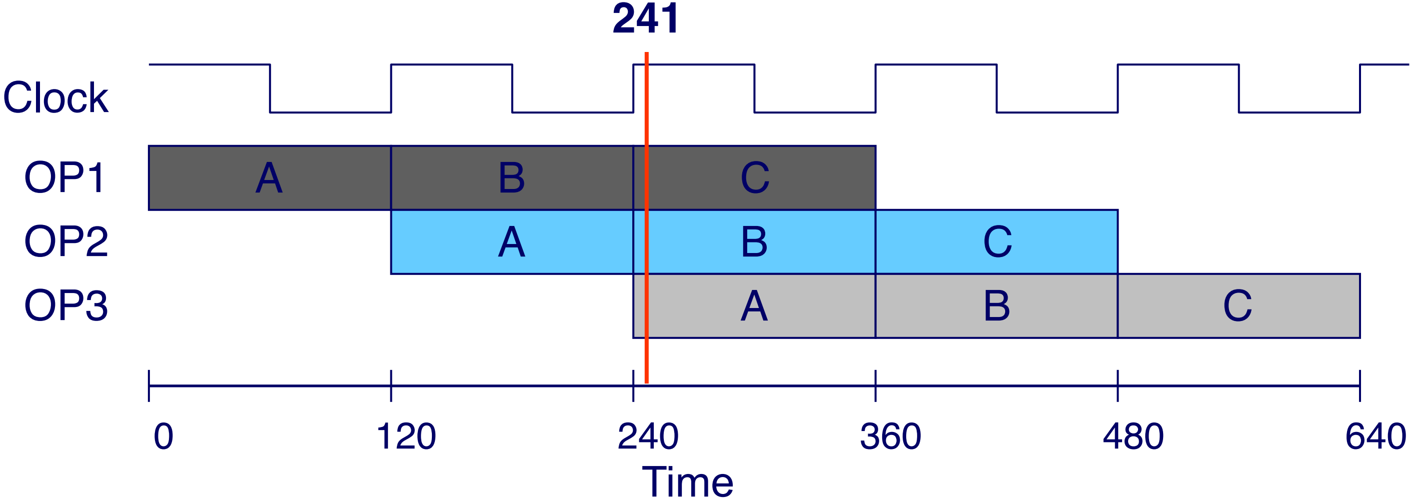
- Up to 3 operations in process simultaneously

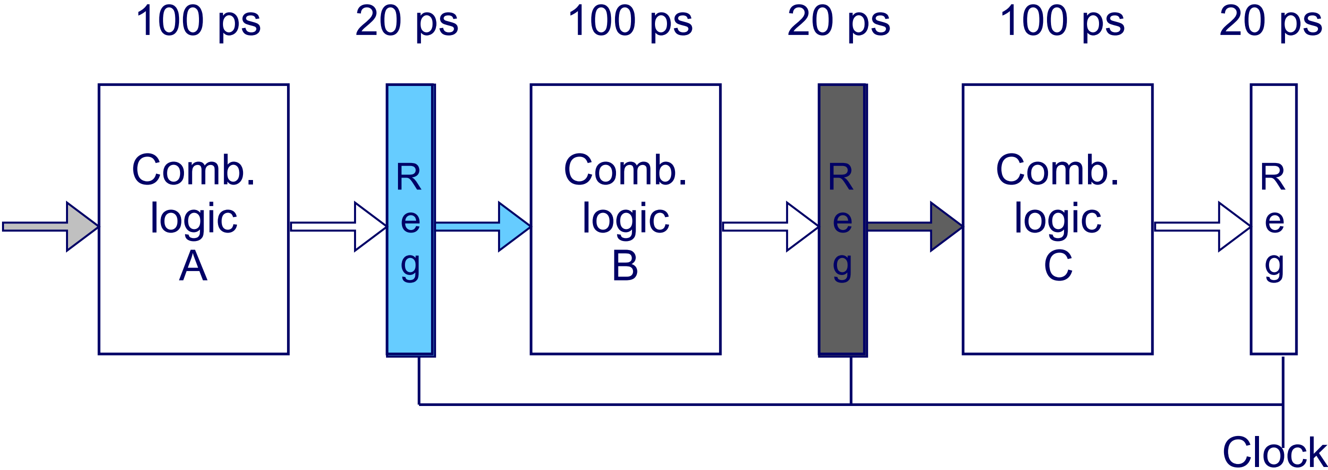
Operating a Pipeline



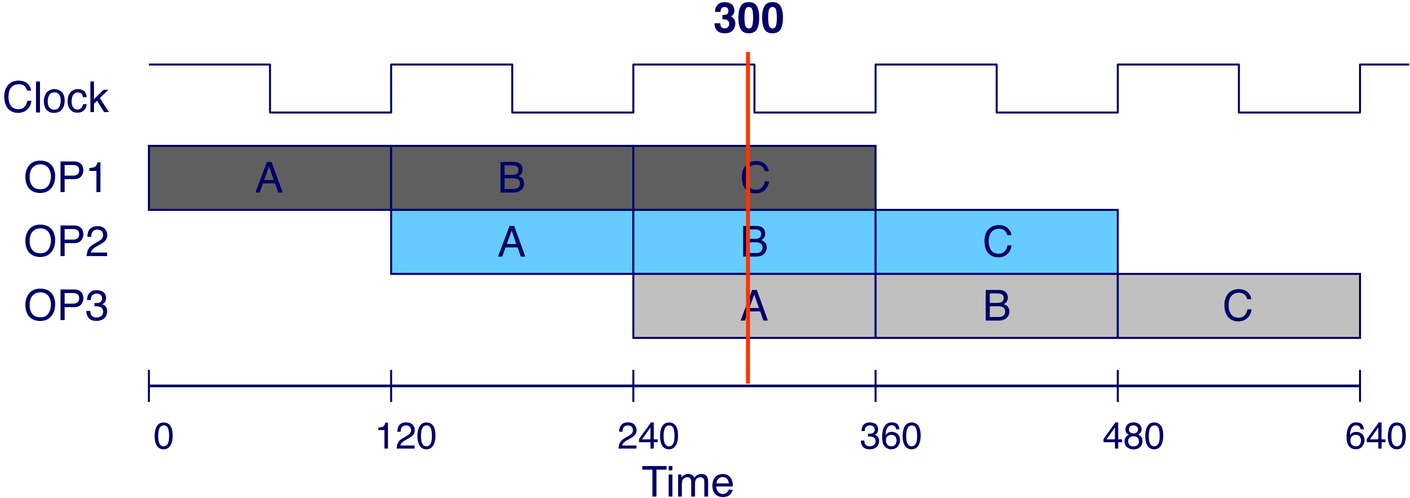


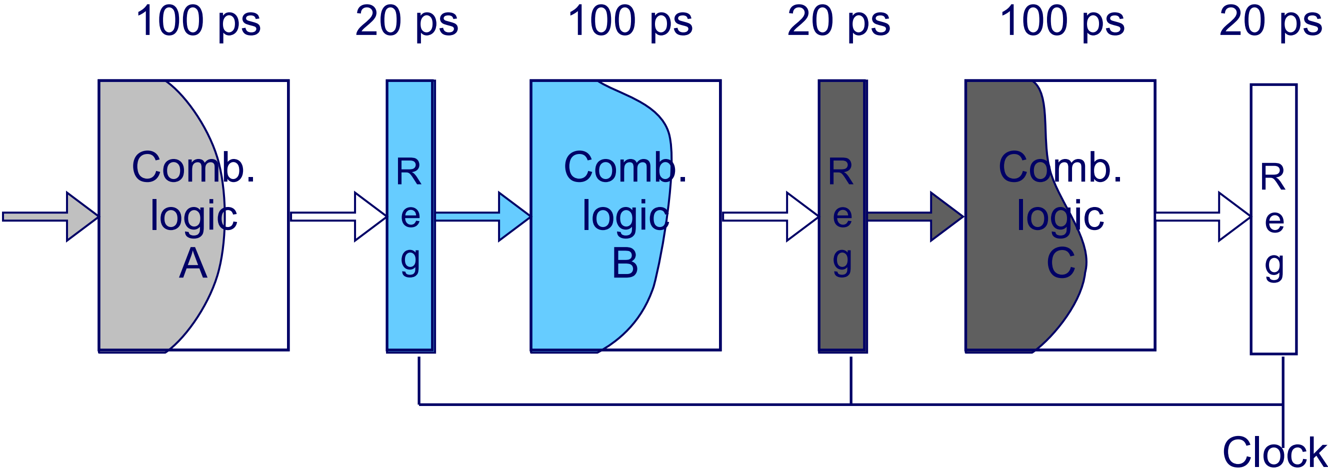
Operating a Pipeline



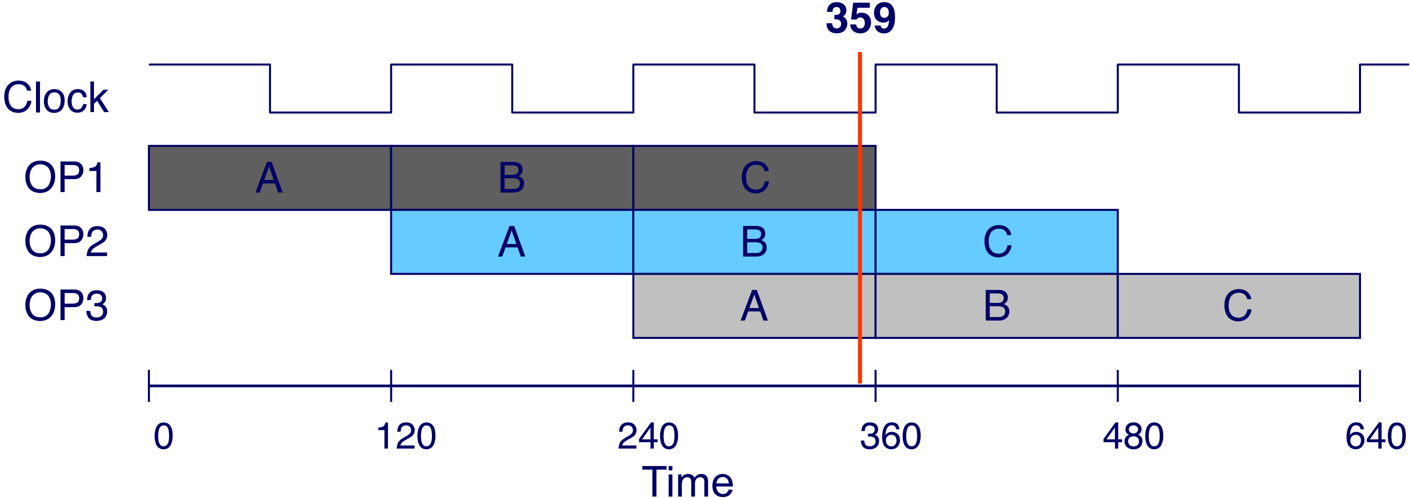


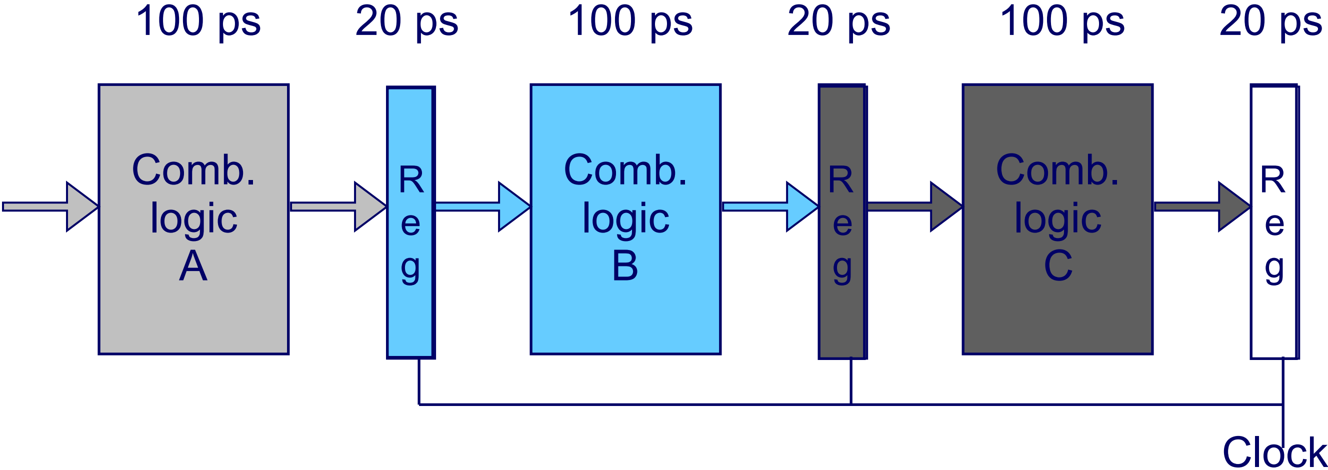
Operating a Pipeline





Operating a Pipeline





Pipeline Demonstration

```
irmovq   $1,%rax  #I1
irmovq   $2,%rcx  #I2
irmovq   $3,%rdx  #I3
irmovq   $4,%rbx  #I4
halt              #I5
```

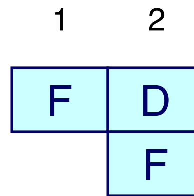
Pipeline Demonstration

```
irmovq   $1,%rax  #I1
irmovq   $2,%rcx  #I2
irmovq   $3,%rdx  #I3
irmovq   $4,%rbx  #I4
halt              #I5
```

	1
#I1	F

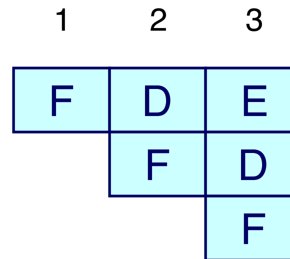
Pipeline Demonstration

```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



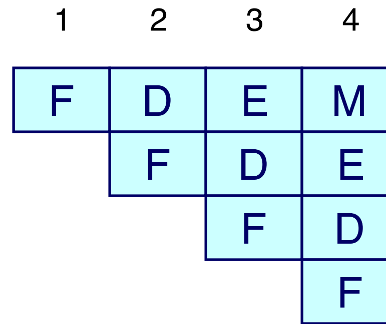
Pipeline Demonstration

```
irmovq   $1,%rax  #I1
irmovq   $2,%rcx  #I2
irmovq   $3,%rdx  #I3
irmovq   $4,%rbx  #I4
halt              #I5
```



Pipeline Demonstration

```
irmovq   $1,%rax  #I1
irmovq   $2,%rcx  #I2
irmovq   $3,%rdx  #I3
irmovq   $4,%rbx  #I4
halt              #I5
```



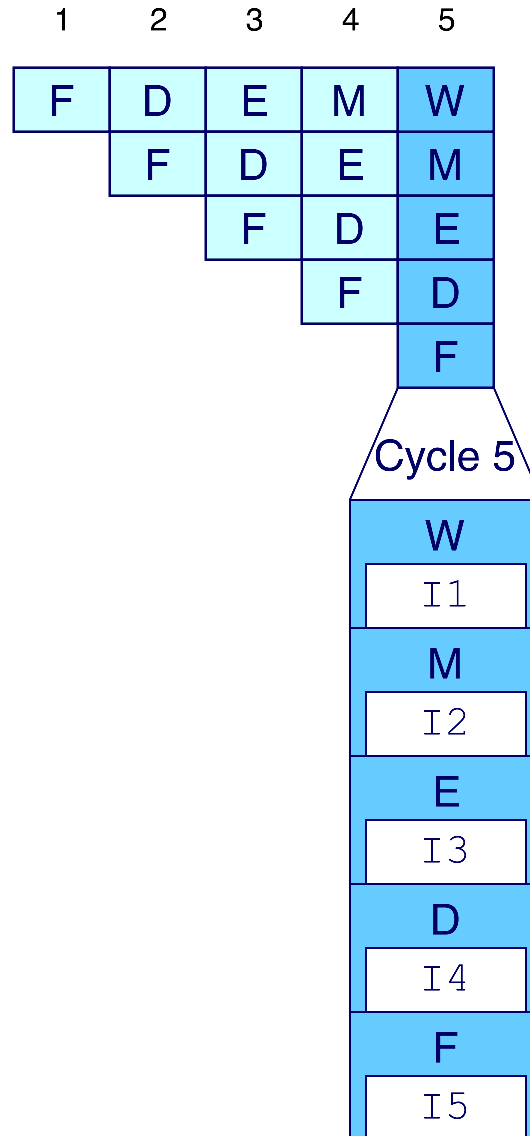
Pipeline Demonstration

```
irmovq   $1,%rax  #I1
irmovq   $2,%rcx  #I2
irmovq   $3,%rdx  #I3
irmovq   $4,%rbx  #I4
halt              #I5
```

	1	2	3	4	5
#I1	F	D	E	M	W
#I2		F	D	E	M
#I3			F	D	E
#I4				F	D
#I5					F

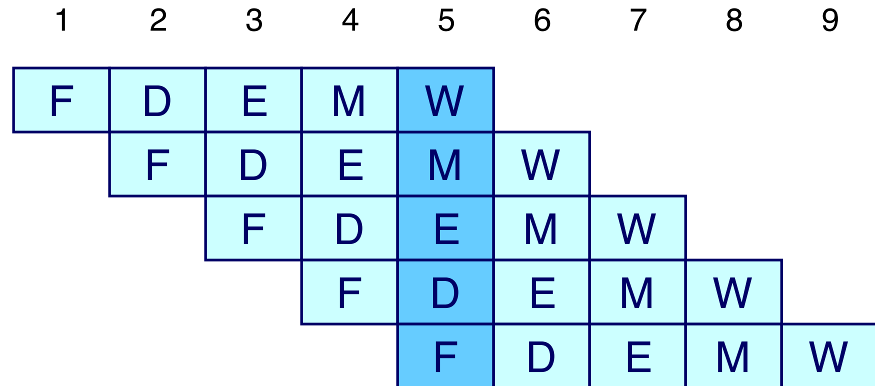
Pipeline Demonstration

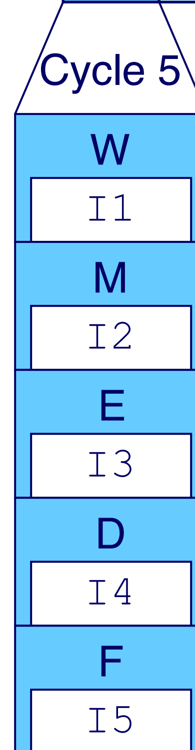
```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



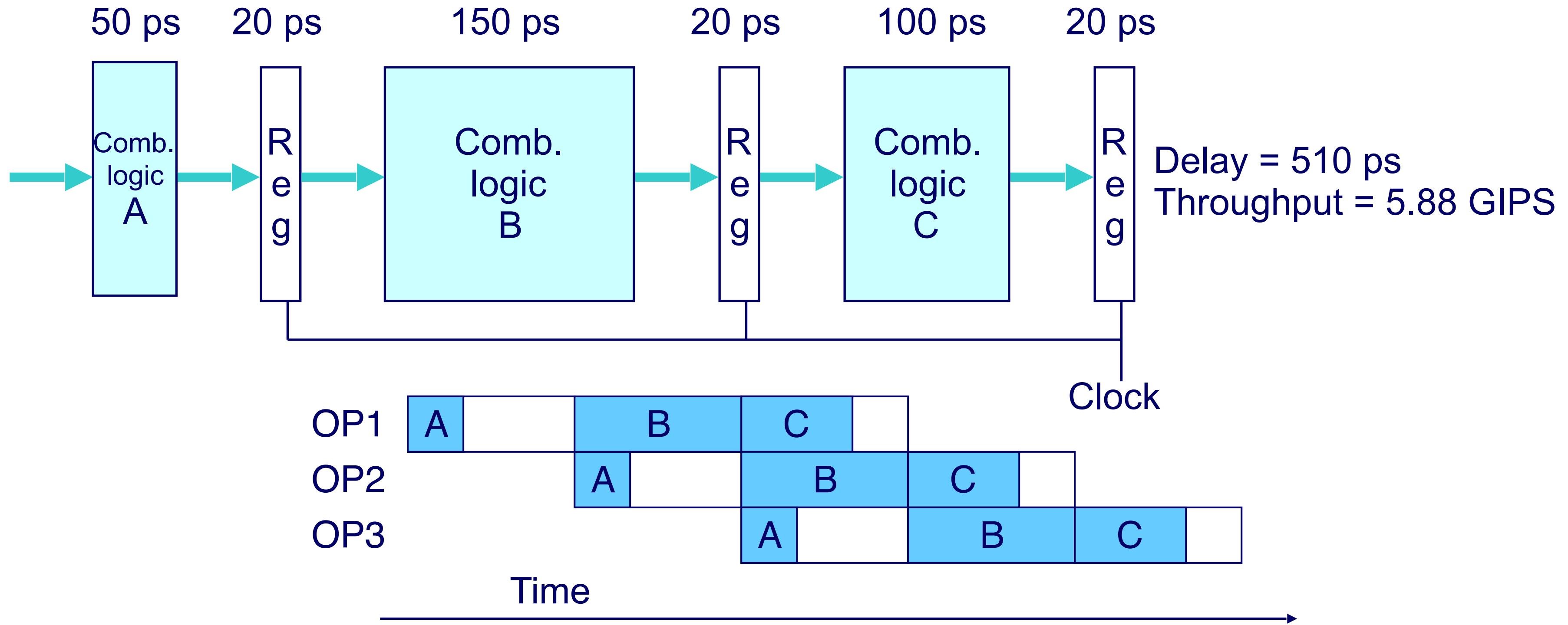
Pipeline Demonstration

```
irmovq   $1,%rax   #I1
irmovq   $2,%rcx   #I2
irmovq   $3,%rdx   #I3
irmovq   $4,%rbx   #I4
halt               #I5
```



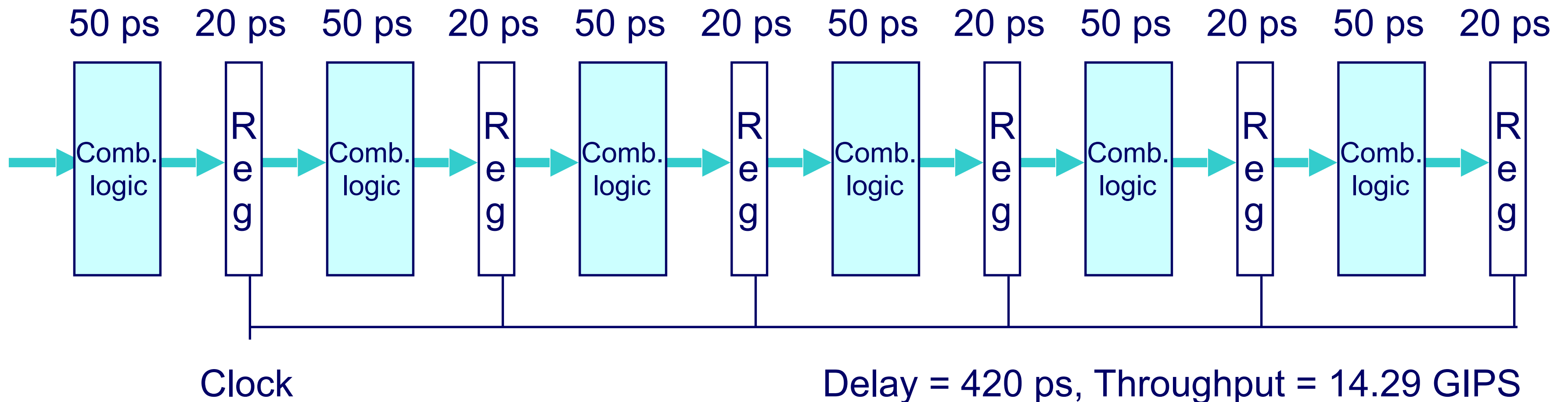


Nonuniform Delays

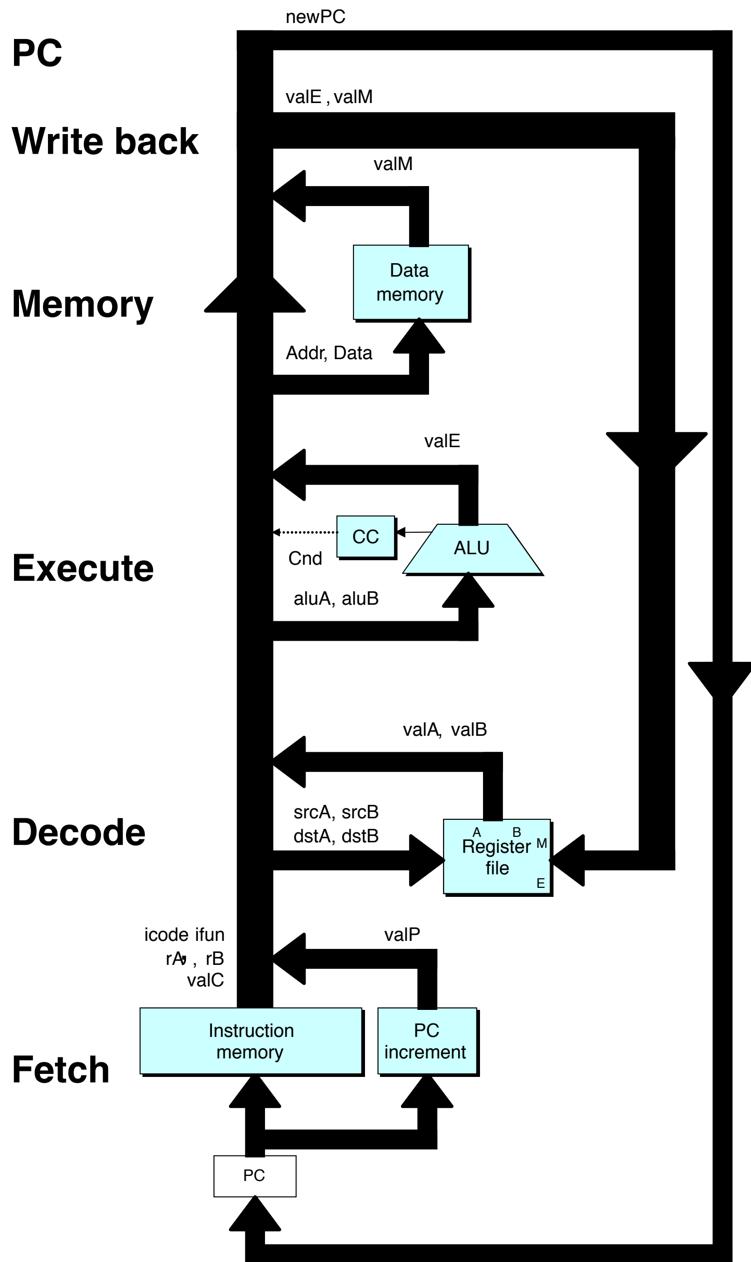


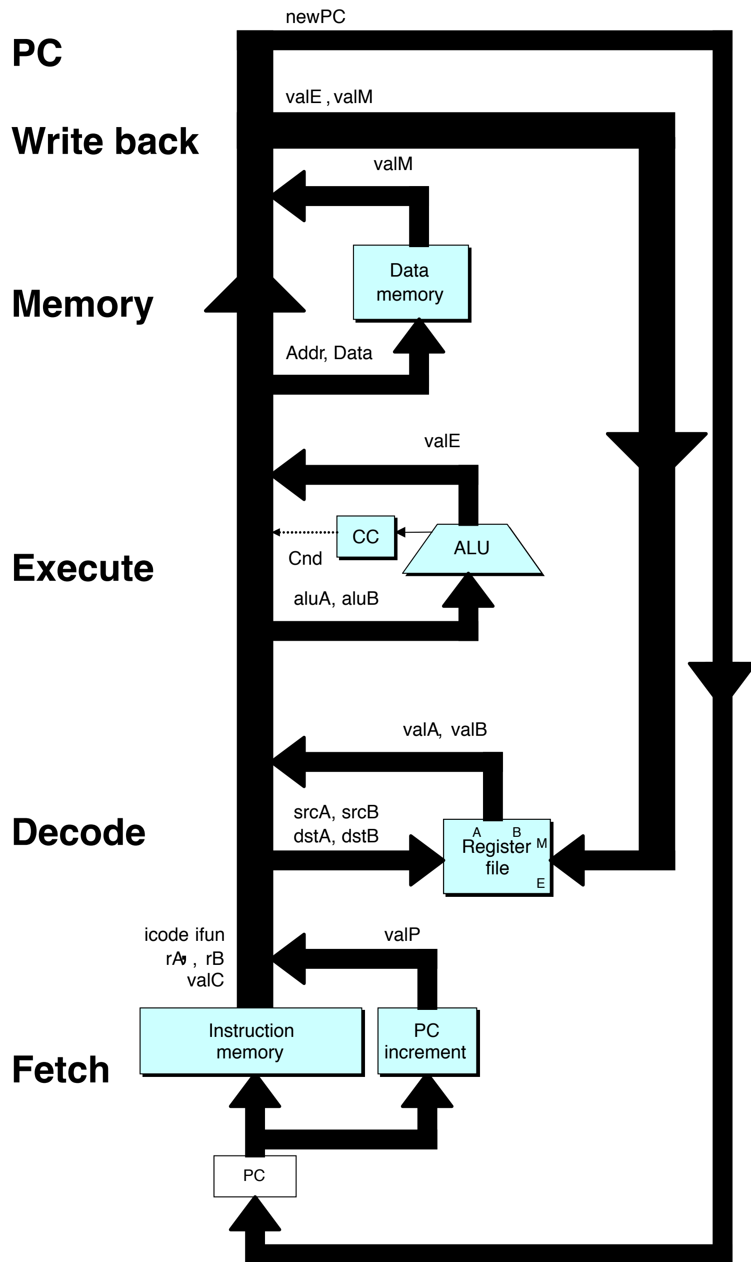
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Register Overhead



- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%


PC
Write back
Memory
Execute
Decode
Fetch
newPC
valE , valM
valM
Data
memory
Addr, Data
valE
CC
ALU
Cnd
aluA, aluB
valA, valB
srcA, srcB
dstA, dstB
A
B
Register
file
M
E
icode ifun
rA , rB
valC
valP
Instruction
memory
PC
increment
PC



W_icode, W_valM
W_valE, W_valM, W_dstE, W_dstM
W
Memory
valM
Data memory
M_icode,
M_Cnd,
M_valA
Addr, Data
M
Cnd
valE
CC
ALU
Execute
aluA, aluB
E
valA, valB
d_srcA,
d_srcB
Decode
Register file
Write back
D
icode, ifun,
rA, rB, valC
valP
Fetch
Instruction memory
PC increment
predPC
PC
f_pc
F

Today: Processor Microarchitecture

- Sequential, single-cycle microarchitecture implementation
  - Basic idea
  - Hardware implementation
- Pipelined microarchitecture implementation
  - Basic Principles
  - Difficulties: Control Dependency
  - Difficulties: Data Dependency

Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     xorg %rax, %rax
     jne L1                # Not taken
     irmovq $1, %rax       # Fall Through
L1   irmovq $4, %rcx       # Target
     irmovq $3, %rax       # Target + 1
```

Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     xorg %rax, %rax
     jne L1                # Not taken
     irmovq $1, %rax       # Fall Through
L1   irmovq $4, %rcx       # Target
     irmovq $3, %rax       # Target + 1
```

1
F

Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     xorg %rax, %rax
     jne L1              # Not taken
     irmovq $1, %rax     # Fall Through
L1   irmovq $4, %rcx     # Target
     irmovq $3, %rax     # Target + 1
```

	1	2
xorg %rax, %rax	F	D
jne L1		F

Control Dependency

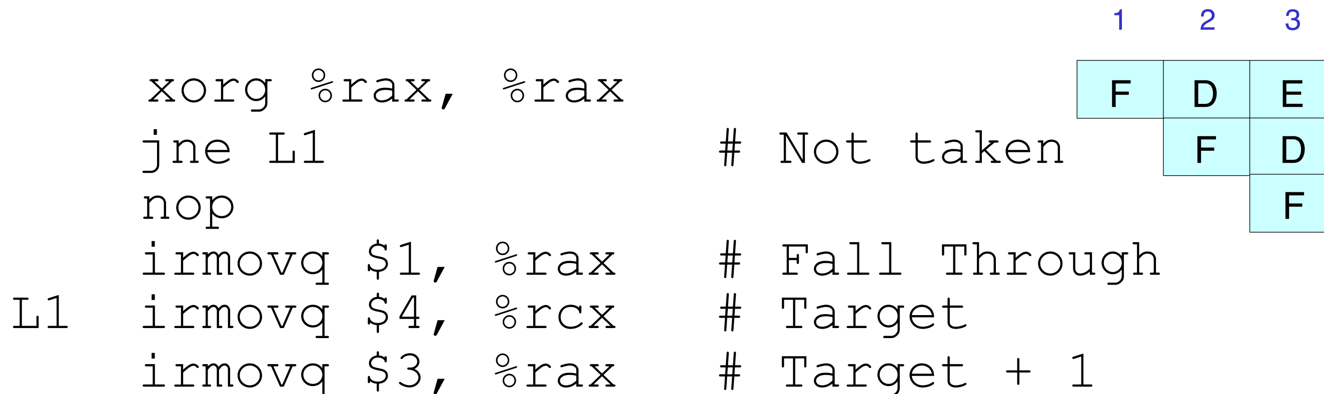
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     xorg %rax, %rax
     jne L1                # Not taken
     irmovq $1, %rax       # Fall Through
L1   irmovq $4, %rcx       # Target
     irmovq $3, %rax       # Target + 1
```

	1	2	3
xorg %rax, %rax	F	D	E
jne L1		F	D

Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
    - `jne L1` determines whether `irmovq $1, %rax` should be executed
    - But `jne` doesn't know its outcome until after its Execute stage



Control Dependency

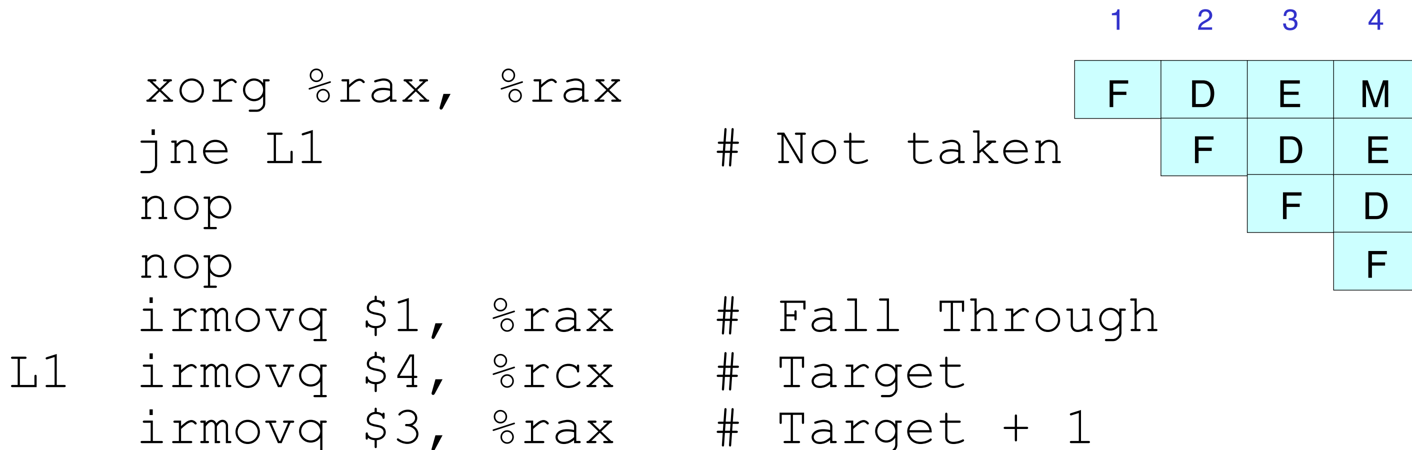
- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

```
     xorg %rax, %rax
     jne L1                # Not taken
     nop
     irmovq $1, %rax       # Fall Through
L1   irmovq $4, %rcx       # Target
     irmovq $3, %rax       # Target + 1
```

1	2	3	4
F	D	E	M
	F	D	E
		F	D

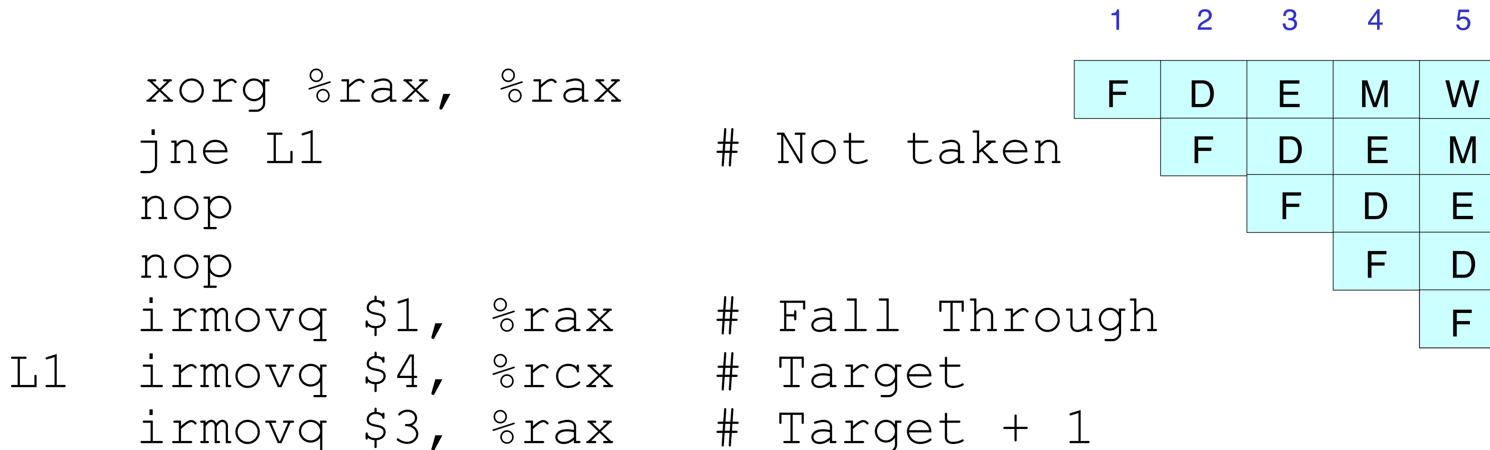
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



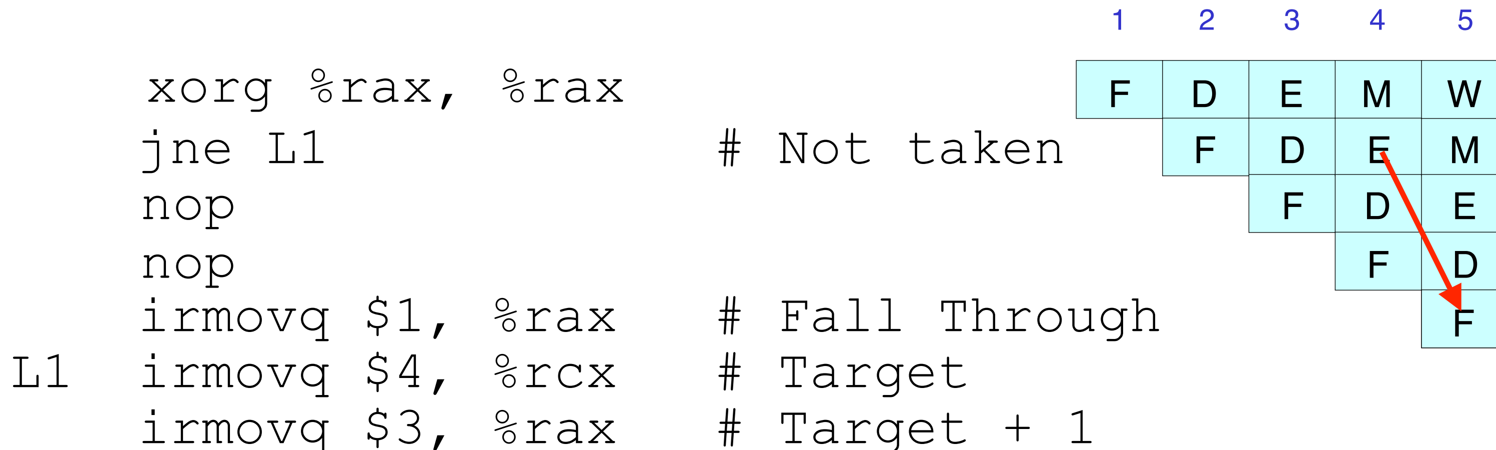
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



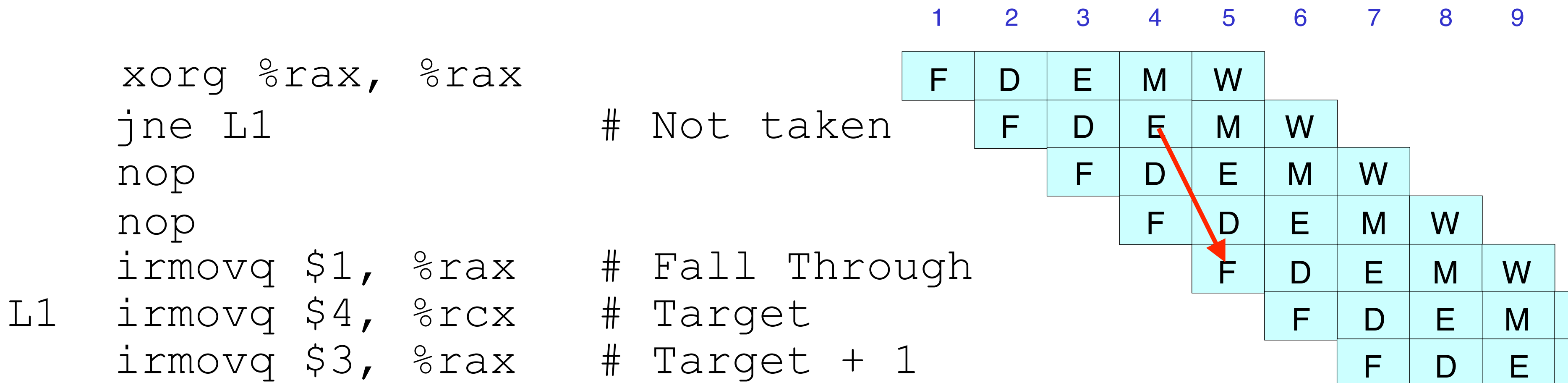
Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage



Control Dependency

- **Definition**: Outcome of instruction A determines whether or not instruction B should be executed or not.
- Jump instruction example below:
  - `jne L1` determines whether `irmovq $1, %rax` should be executed
  - But `jne` doesn't know its outcome until after its Execute stage

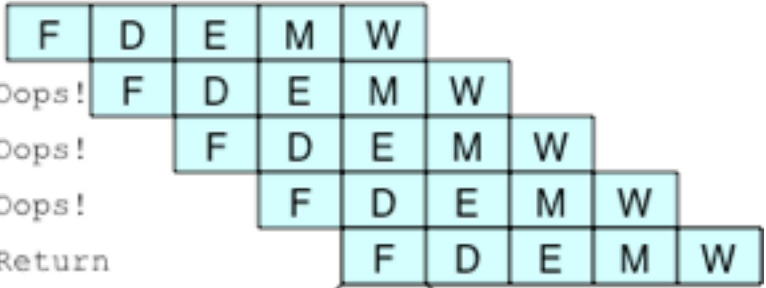


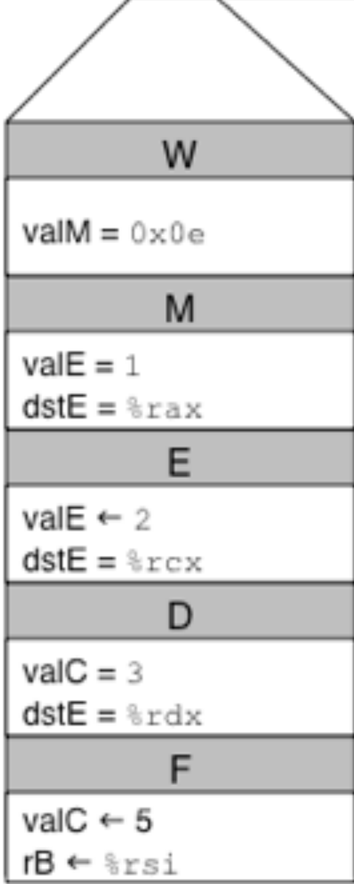
Control Dependency: Return Example

```
0x000:     irmovq Stack,%rsp   # Intialize stack pointer
0x00a:     call p              # Procedure call
0x013:     irmovq $5,%rsi      # Return point
0x01d:     halt
0x020: p:  irmovq $-1,%rdi     # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax      # Should not be executed
0x035:     irmovq $2,%rcx      # Should not be executed
0x03f:     irmovq $3,%rdx      # Should not be executed
0x049:     irmovq $4,%rbx      # Should not be executed
0x100: Stack:                  # Stack: Stack pointer
```

Control Dependency: Incorrect Return

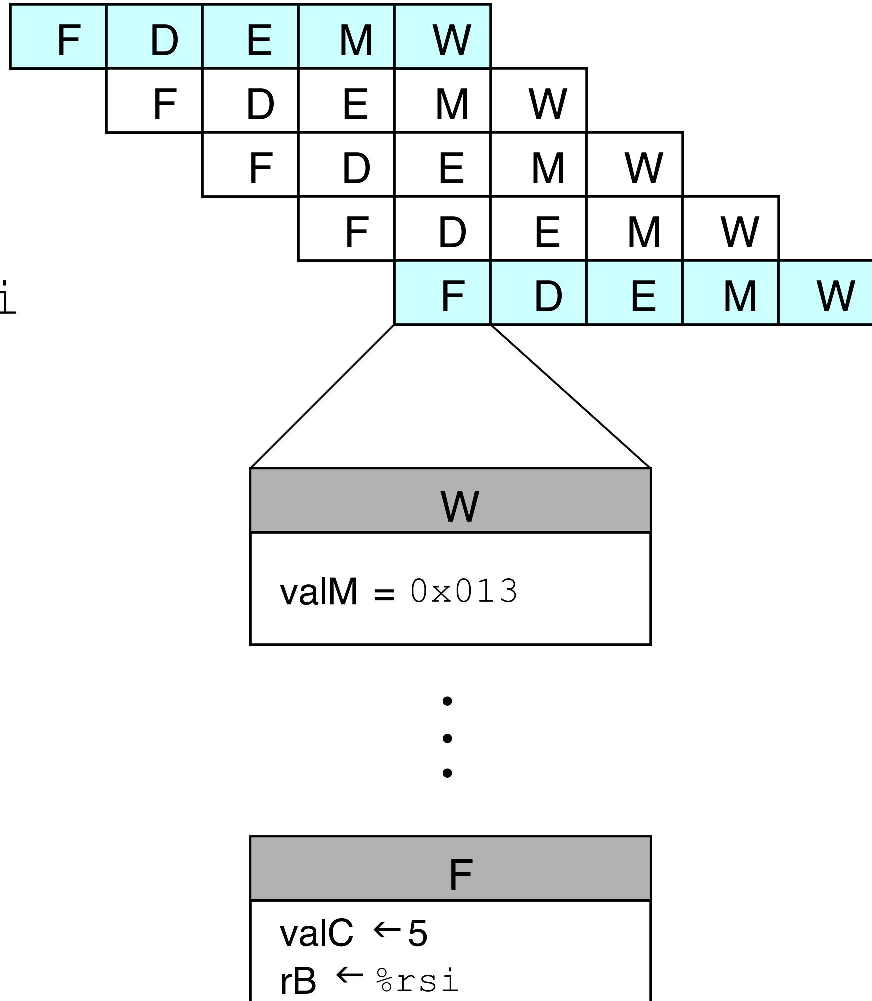
```
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```





Control Dependency: Correct Return

```
0x026:    ret
          nop
          nop
          nop
0x013:    irmovq $5,%rsi
```



Resolving Control Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
  - Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach
- Hardware mechanisms
  - Stalling
  - Branch Prediction
  - Return Address Stack
- We will discuss them more next lecture

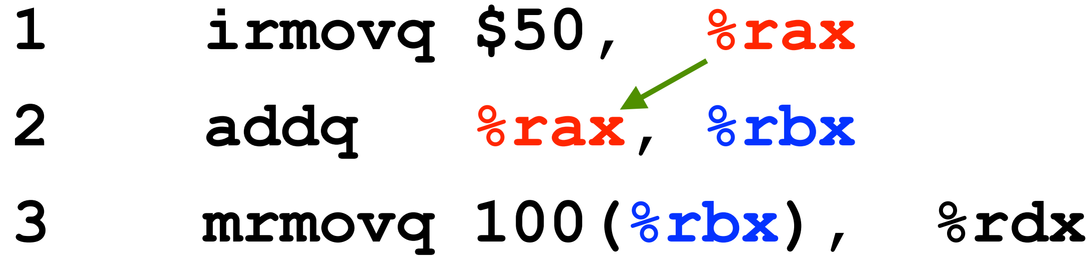
Today: Processor Microarchitecture

- Sequential, single-cycle microarchitecture implementation
  - Basic idea
  - Hardware implementation
- Pipelined microarchitecture implementation
  - Basic Principles
  - Difficulties: Control Dependency
  - Difficulties: Data Dependency

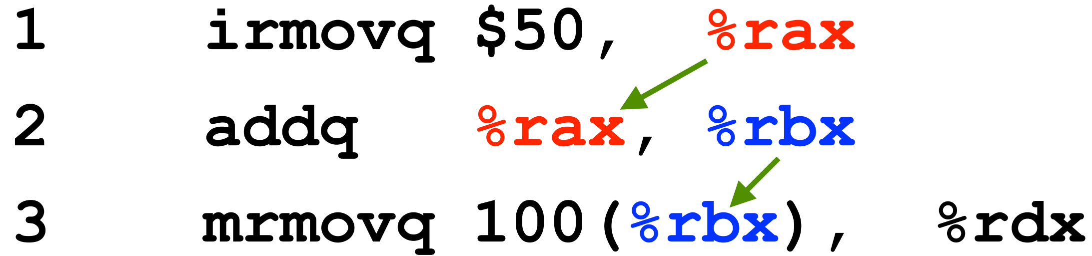
Data Dependencies

```
1    irmovq $50,  %rax
2    addq   %rax, %rbx
3    mrmovq 100(%rbx),  %rdx
```

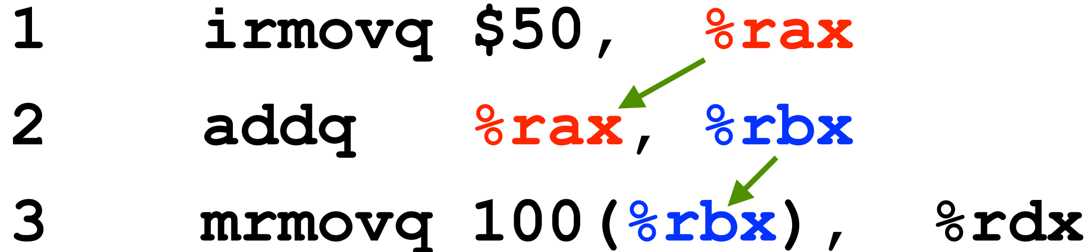
Data Dependencies



Data Dependencies

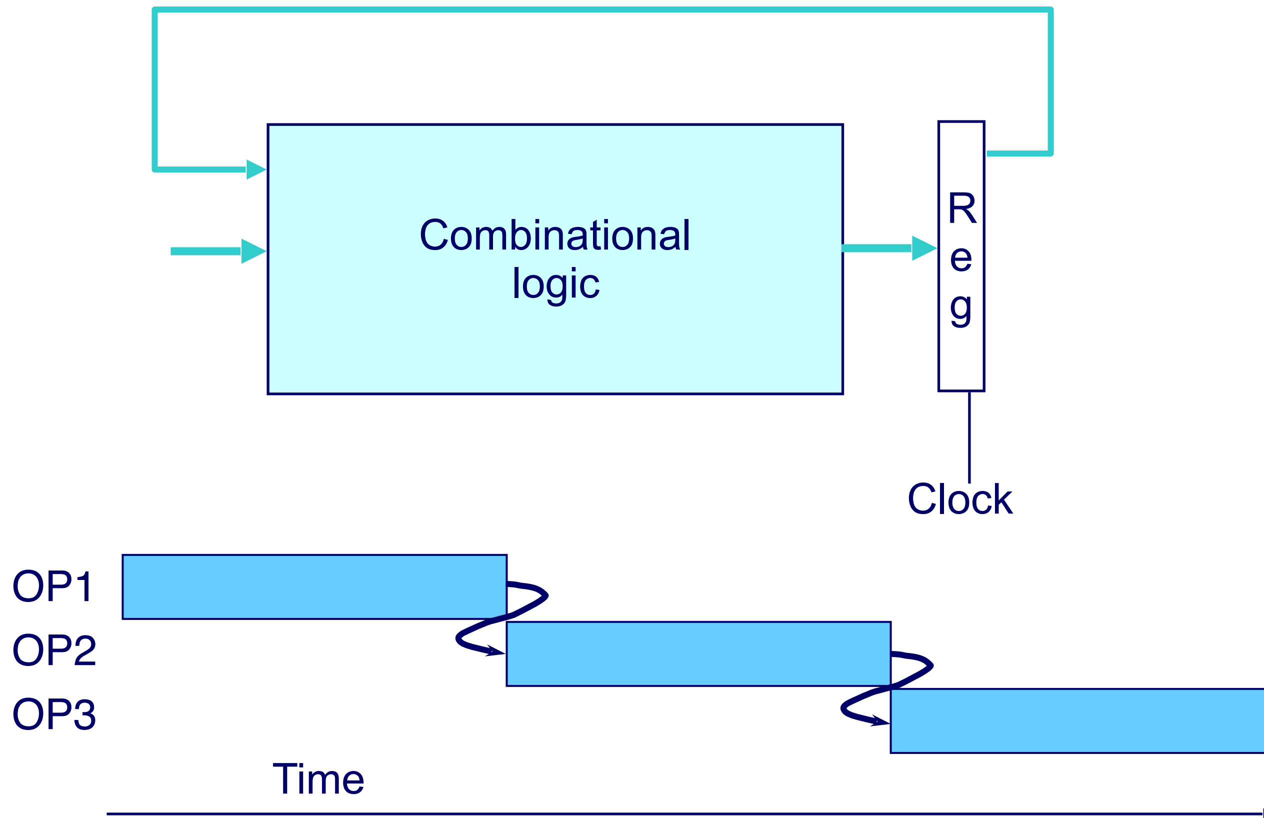


Data Dependencies



- Result from one instruction used as operand for another
  - **Read-after-write (RAW)** dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

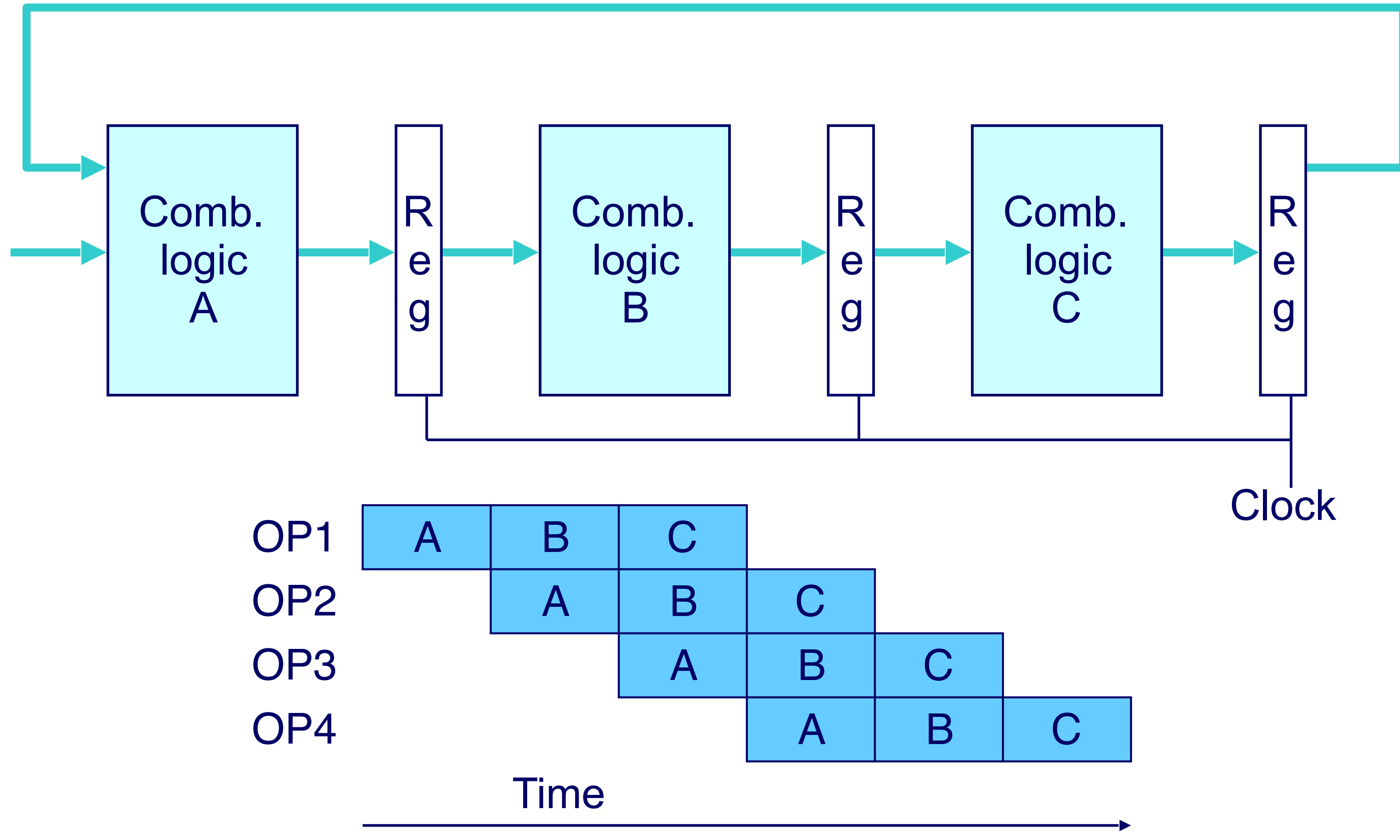
Data Dependencies in Single-Cycle Machines



In Single-Cycle Implementation:

- Each operation starts only after the previous operation finishes. Dependency always satisfied.

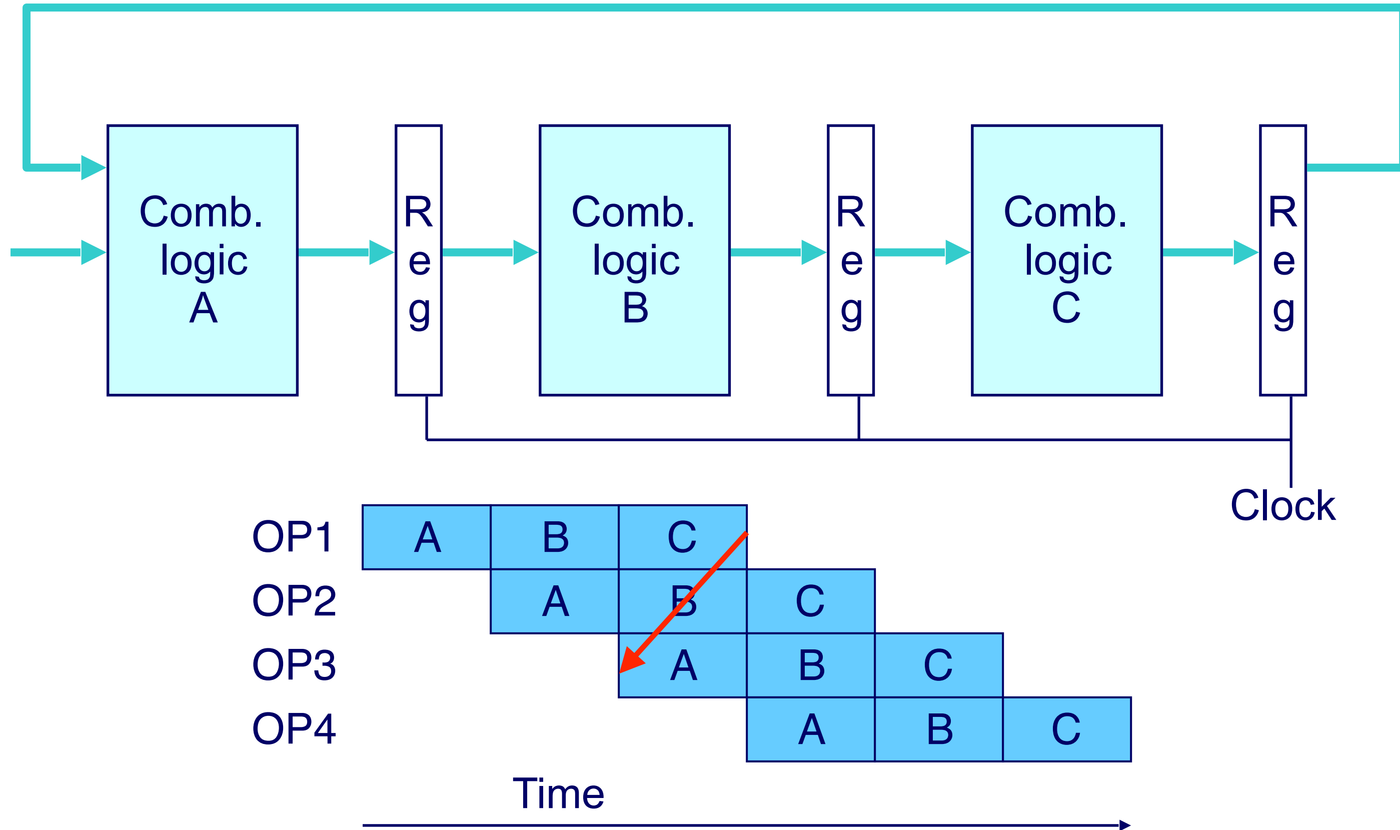
Data Dependencies in Pipeline Machines



Data Hazards happen when:

- Result does not feed back around in time for next operation

Data Dependencies in Pipeline Machines

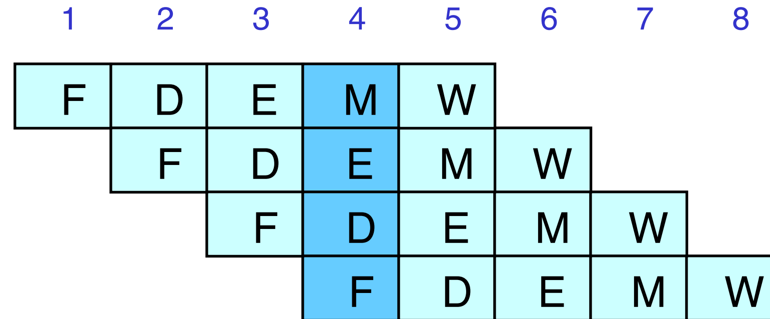


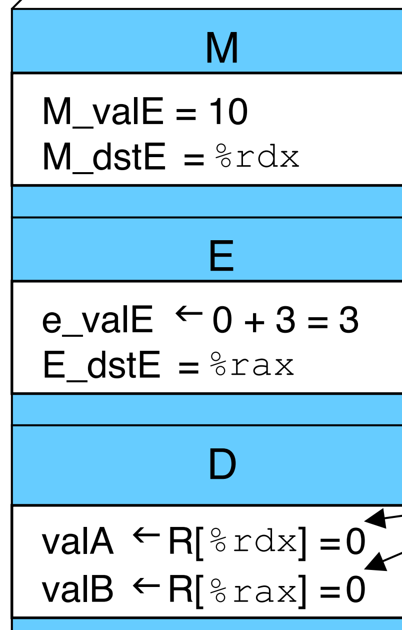
Data Hazards happen when:

- Result does not feed back around in time for next operation

Data Dependencies: No Nop

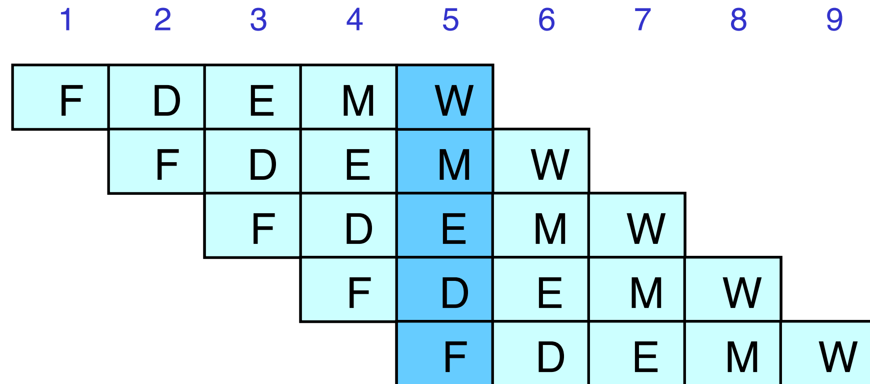
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

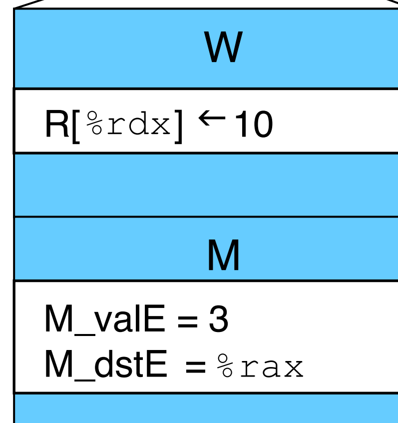


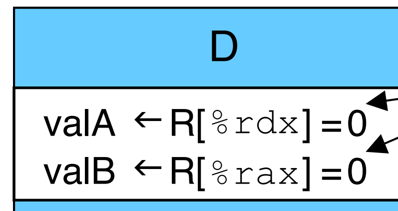


Data Dependencies: 1 Nop

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```

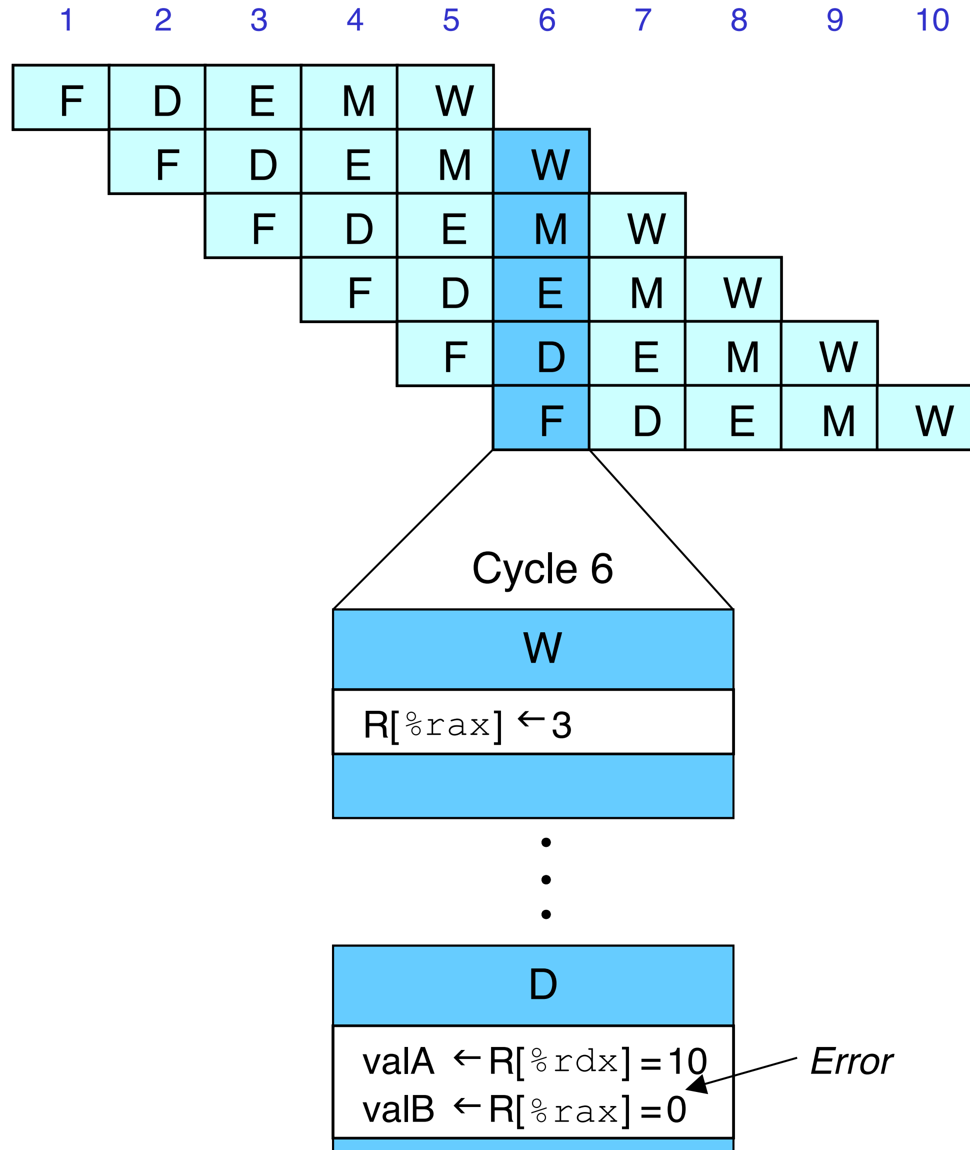






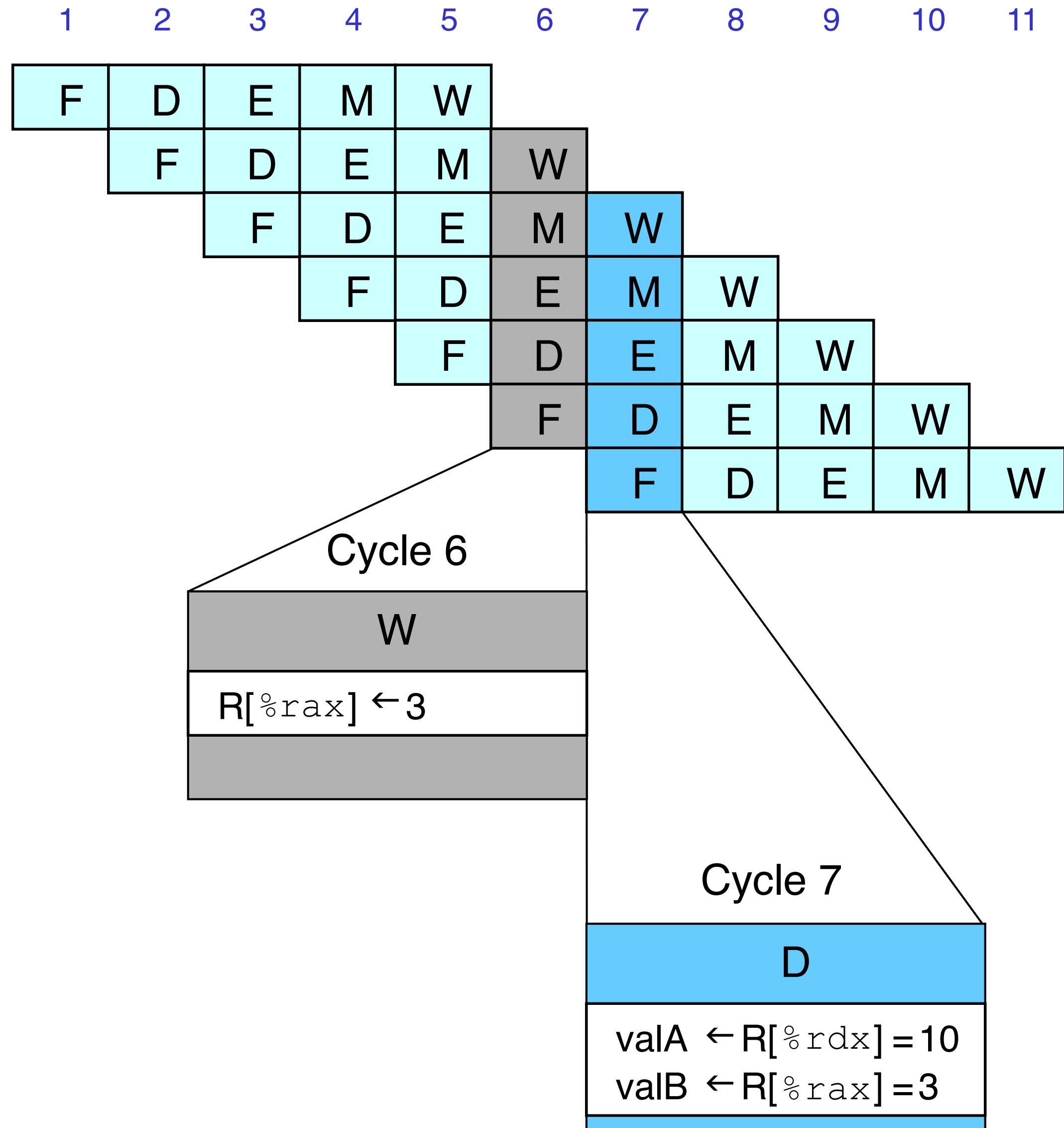
Data Dependencies: 2 Nop's

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



Data Dependencies: 3 Nop's

```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```



Resolving Data Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
- Hardware mechanisms
  - Stalling
  - Forwarding
  - Out-of-order execution
- We will discuss them more next lecture