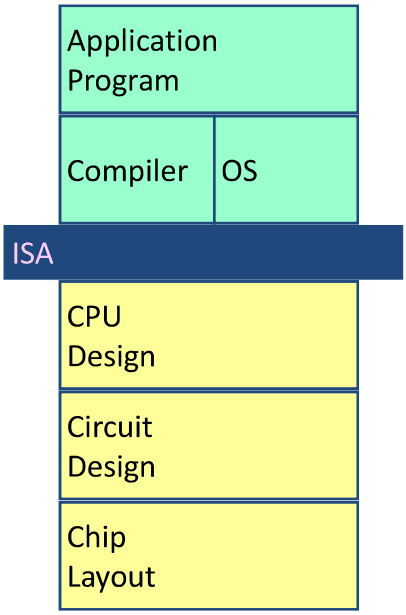
Computer Systems Organization
CS2.201

Topic 4

Based on chapter 4 from Computer Systems by
Randal E. Bryant and David R. O'Hallaron

Instruction Set Architecture

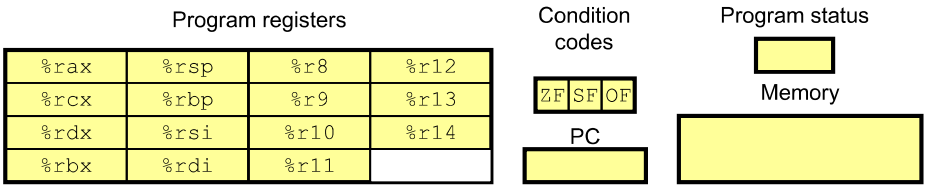
- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addq`, `pushq`, `ret`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



What is Y86-64 ?

- Y86 is a “toy” machine that is similar to the x86 but much simpler and acts as a gentler introduction to assembly level programming.
  - Just a few instructions as opposed to hundreds for the x86;
  - Uses absolute addressing
  - Everything you learn about the Y86 will apply to the x86 with very little modification
  - Chapter introduces pipelining in that context of Y86

Y86-64 Processor State



- Program Registers
  - 15 registers (omit `%r15`). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF:Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Encoding Registers

- Each register has 4-bit ID

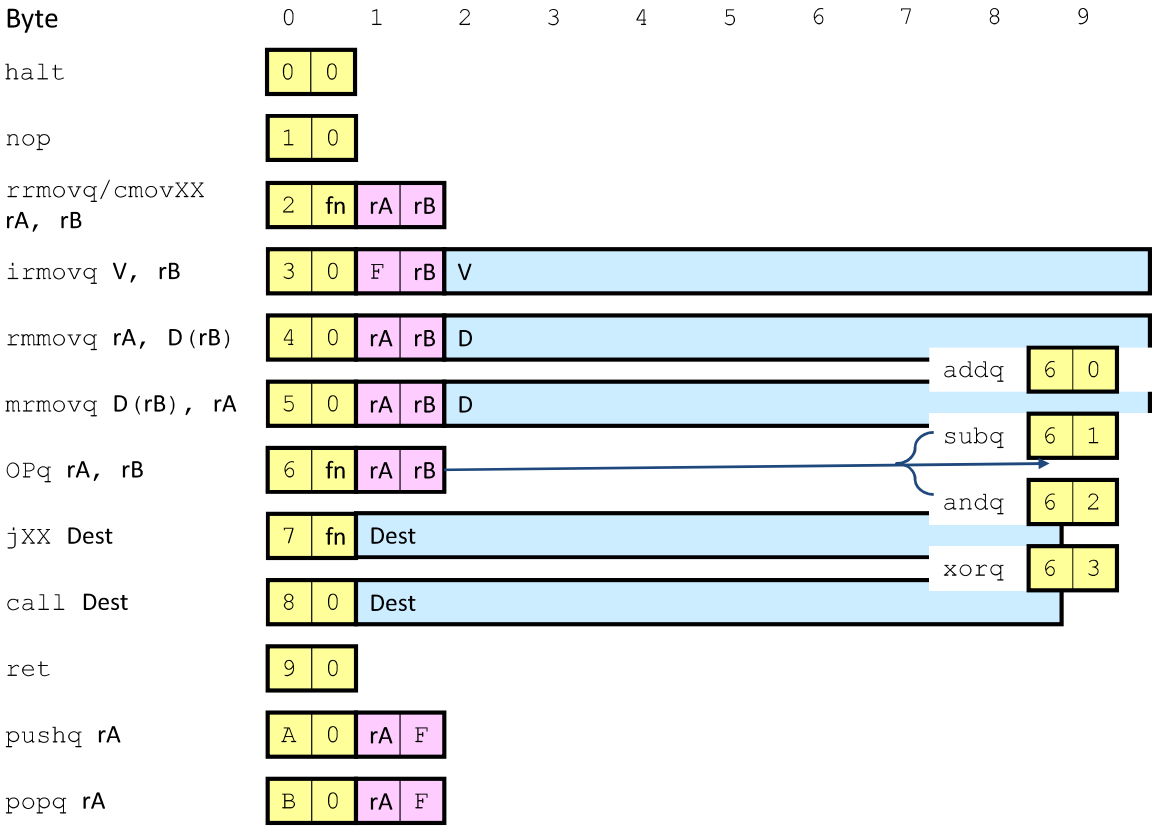
%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

  - Same encoding as in x86-64
- Register ID 15 (`0xF`) indicates "no register"
  - Useful in hardware design in multiple places

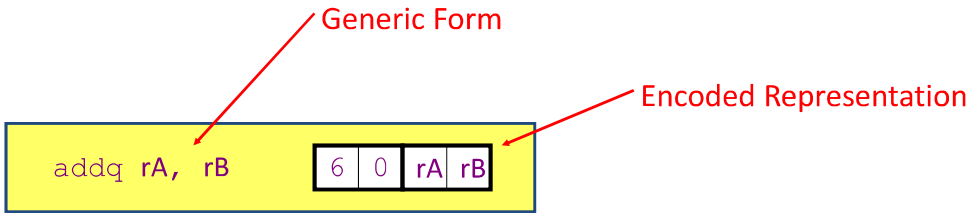
Y86-64 Instructions

- Format
  - 1–10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with x86-64
  - Each accesses and modifies some part(s) of the program state

Y86-64 Instruction Set



Instruction Example



- Addition Instruction
  - Add value in register rA to that in register rB
    - Store result in register rB
    - Y86-64 only allows addition to be applied to register data
  - Set condition codes based on result
  - e.g., `addq %rax,%rsi` Encoding: `60 06`
  - Two-byte encoding
    - First indicates instruction type
    - Second gives source and destination registers

Move Operations

Register ➔ Register

Immediate ➔ Register

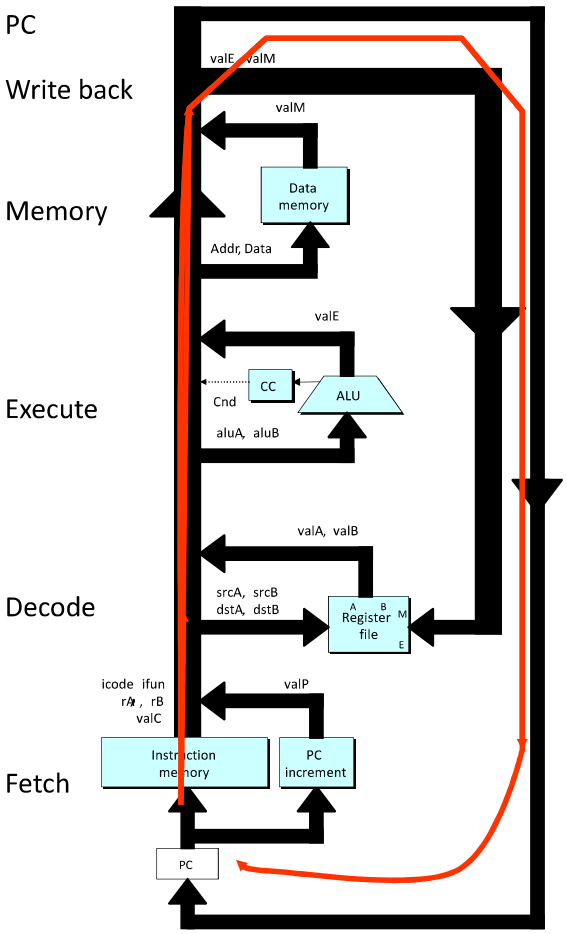
Register ➔ Memory

Memory ➔ Register

- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Different names to keep them distinct

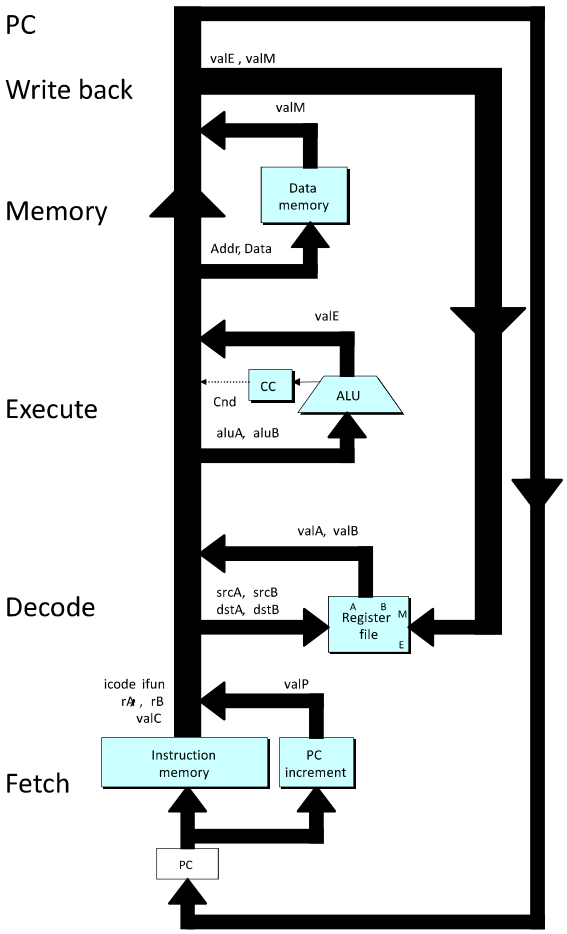
Sequential (SEQ) Hardware Structure

- State
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter



SEQ Stages

- 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



Executing Arith./Logical Operation

- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register
- PC Update
  - Increment PC by 2

Stage Computation: Arith/Log. Ops

	OPq rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Executing `rmmovq`

- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10

Stage Computation: `rmmovq`

	`rmmovq` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

– Use ALU for address computation

SEQ Summary

- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assemble registers, memories, predesigned combinational blocks
  - Connect with control logic
- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle

Real-World Pipelines: Car Washes

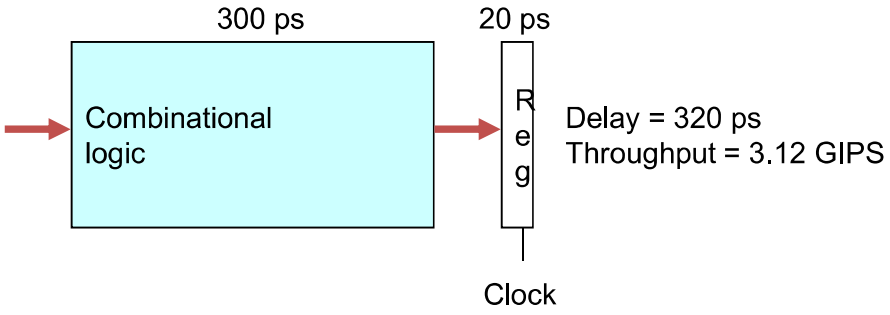
Sequential

Parallel

Pipelined

- Idea
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any given times, multiple objects being processed
  - Spraying water and soap, scrubbing, applying wax, drying etc.

Computational Example



- System
  - Computation requires total of 300 picoseconds
  - Additional 20 picoseconds to save result in register
  - Must have clock cycle of at least 320 ps and the clock signal controls loading of register at regular time interval

Pipelining

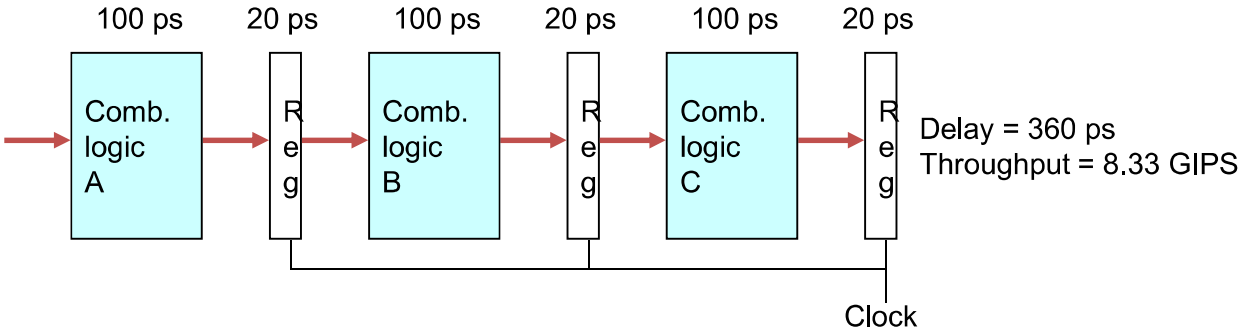
- [Car example] Key feature of pipelining is it increases throughput (i.e., number of cars serviced per unit time), but it may increase latency to some extent (i.e., time required to service an individual customer).
  - E.g., if a car needs one of the task, it may still need to pass through the pipeline

Throughput

- Throughput = 1 instruction/(300+20) picosecs
  = 1/320ps * 1000ps/1 nanosec
- ~ 3.12 GIPS (giga or billion instructions per sec)
- 1ps = 10^-12 secs = 10^-9 ns
- 1 billion = 1000 million = 1000 *10^6 = 10^9

- Total time required to perform a single instruction is called latency which is 320 ps here.

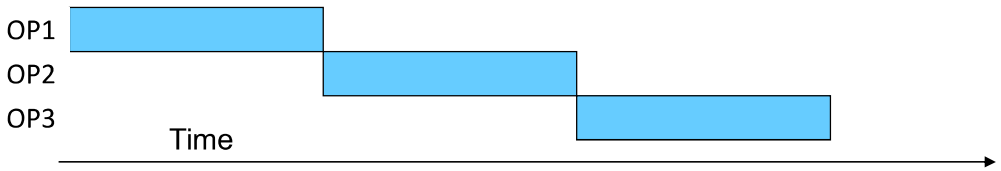
3-Way Pipelined Version



- System
  - Divide combinational logic into 3 blocks of 100 ps each
  - Can begin new operation as soon as previous one gets done
    - Begin new operation every 120 ps
  - Overall latency increases
    - 360 ps from start to finish

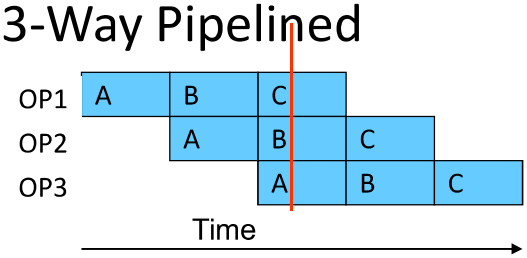
Pipeline Diagrams

- Unpipelined



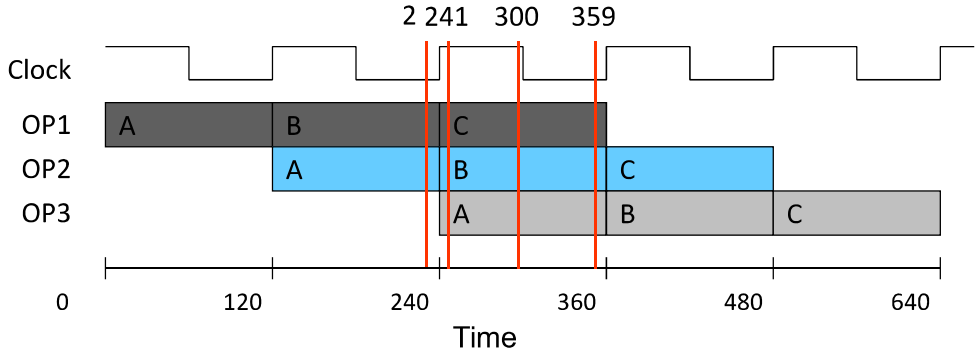
  - Cannot start new operation until previous one completes

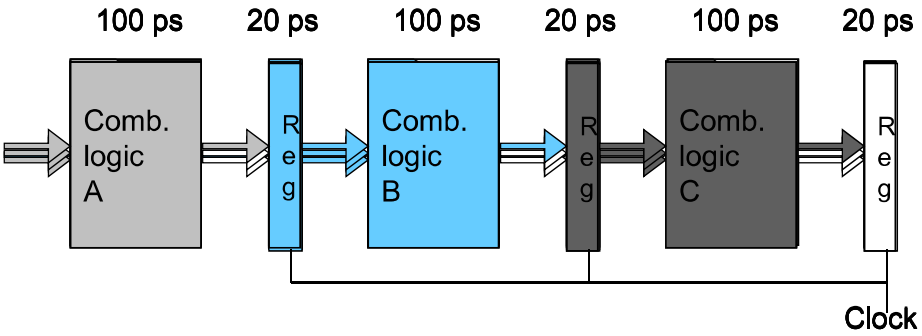
- 3-Way Pipelined



  - Up to 3 operations in process simultaneously

Operating a Pipeline

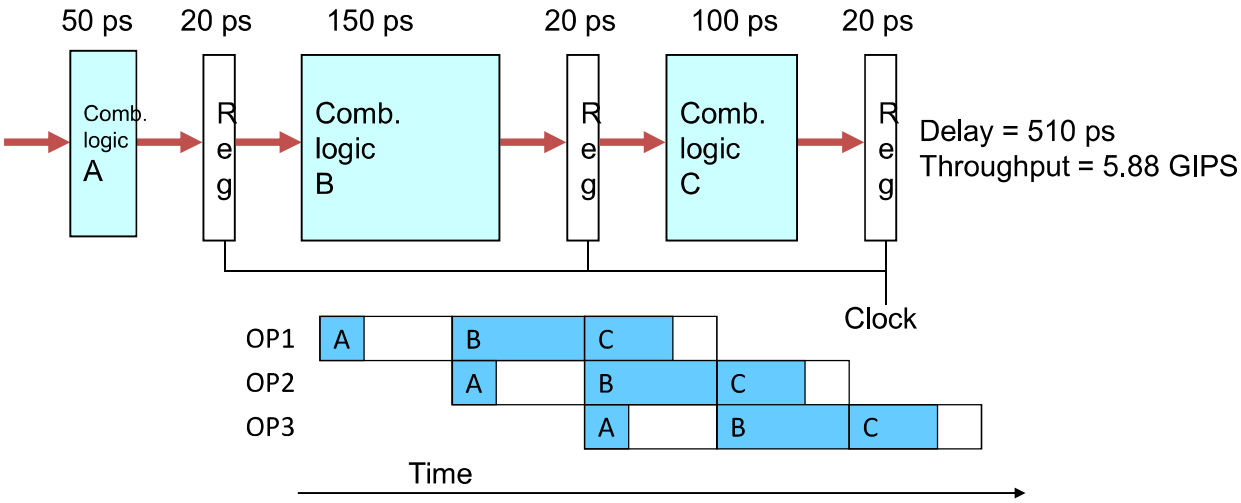




Operating a pipeline

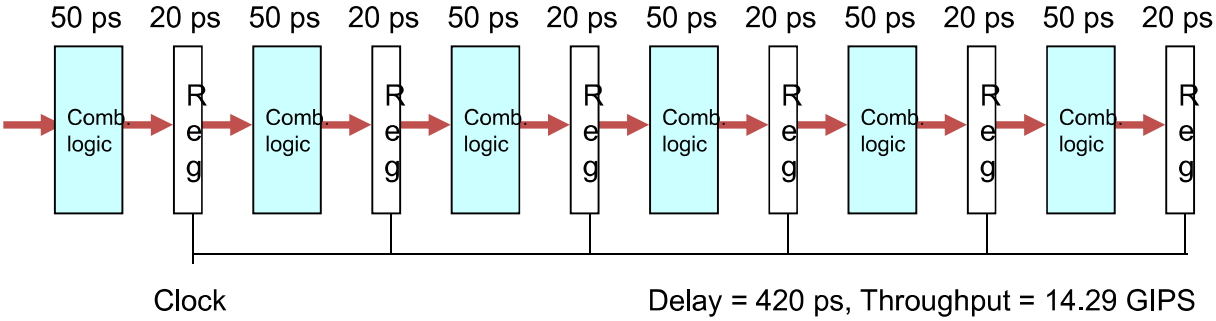
- Just before rising clock at 240, values in A for OP2 reached input of first pipeline register but the register state and output remain set to those computed during A for OP1
- As clock rises, these inputs are loaded into the pipeline registers becoming the register outputs
- Signals propagate to the pipeline register inputs, but no change in the register states will occur until the clock rises
- If clock runs too fast, values would not have time to propagate through the combinatorial logic, so register inputs would not yet be valid when the clock rises.

Limitations: Nonuniform Delays



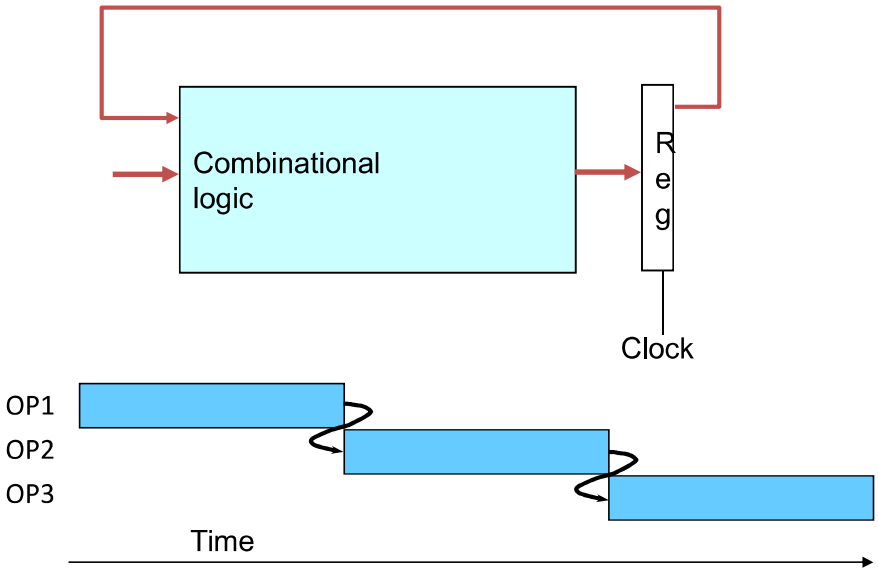
- Throughput limited by slowest stage
- Other stages sit idle for much of the time
- Challenging to partition system into balanced stages

Limitations: Register Overhead



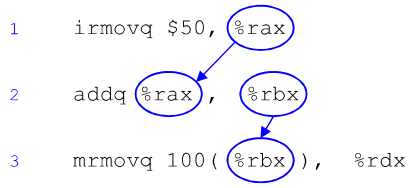
- As you try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining (15 or more stages)

Data Dependencies



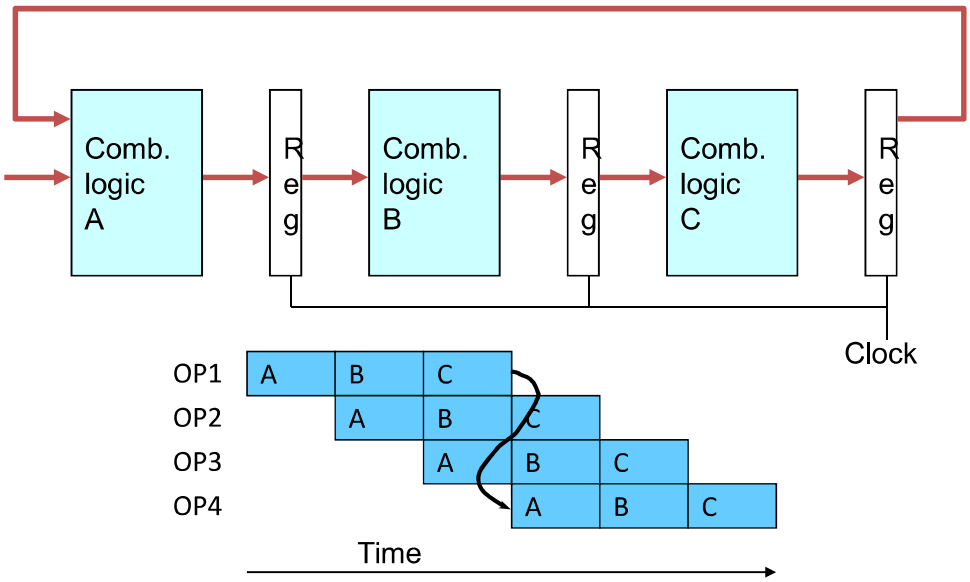
- System
  - Each operation depends on result from preceding one

Data Dependencies in Processors



- 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

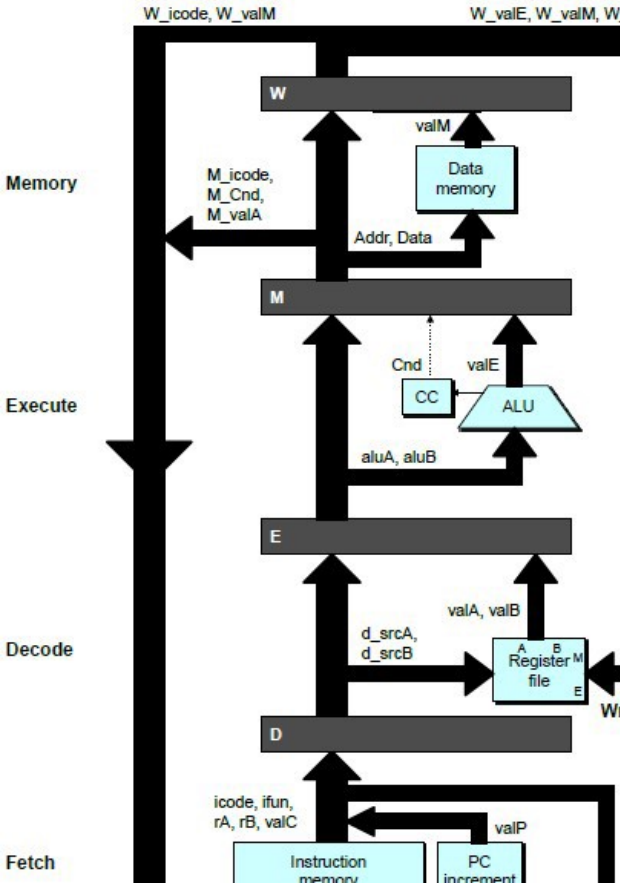
Data Hazards



- Result does not feed back around in time for next operation
- Pipelining has changed behavior of system

Pipeline Stages

- Fetch
  - Select current PC
  - Read instruction
  - Compute incremented PC
- Decode
  - Read program registers
- Execute
  - Operate ALU
- Memory
  - Read or write data memory
- Write Back
  - Update register file
  - Pipeline registers hold intermediate values from instruction execution
- Forward (Upward) Paths
  - Values passed from one stage to next
  - Cannot jump past stages



Predicting the PC

- One goal of pipelined design is to issue a new instruction on every clock cycle
  - Ensures throughput of one instruction per cycle
  - Start fetch of new instruction after current one has completed fetch stage
    - Not enough time to reliably determine next instruction in all cases e.g., for conditional jump and ret instruction
    - May need to guess which instruction will follow and recover if prediction was incorrect
  - Apart from conditional jump and return, can determine address of next instruction based on information computed during the fetch stage
    - For call and jmp (unconditional) it will be valC, the constant word in the instruction while for all others it will be valP
- Conditional Jumps
  - Would not know whether jump would be taken so new PC value would be valC or will not be taken so valP
  - Either case will need to deal with issue that the prediction was incorrect i.e., wrong instruction may have been fetched (and even partially executed)

Predicting the PC

  - Technique of guessing branch direction and initiating the fetching of instructions based on guess is known as branch prediction (used in some form by virtually all processors)
  - Simple prediction strategy used in Y86 implementation – conditional branches are always taken i.e., predict PC to be valC
    - Studies show typically right 60% of time
- Return Instruction
  - Cannot determine the return location until the instruction has retrieved the location to return to
  - Set of possible results is not bounded
  - Don't try to predict and hold off processing until top of stack retrieved
- Instructions that Don't Transfer Control
  - Predict next PC to be valP - always reliable
- Call and Unconditional Jumps
  - Predict next PC to be valC (destination) - always reliable

Pipeline Registers

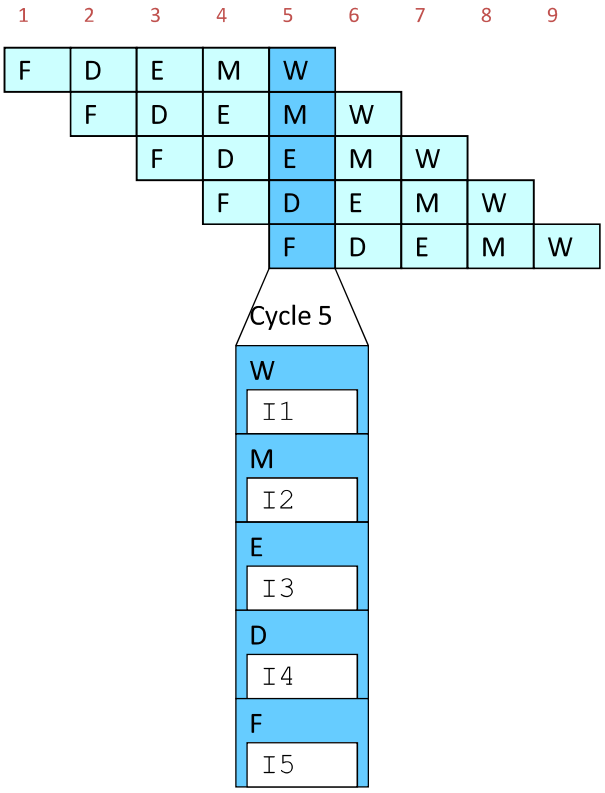
- **F** holds predicted value of PC
- **D** sits between fetch and decode stages. Holds information about the most recently fetched instruction for processing by decode stage
- **E** sits between the decode and execute stages. Holds information about the most recently decoded instruction and the values read from the register file for execute stage
- **M** sits between execute and memory stages. Holds results of the most recently executed instruction for processing by memory stage. Also holds information about branch conditions and branch targets for processing conditional jumps.

Pipeline Registers

- **W** sits between memory stage and feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.

Pipeline Demonstration

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

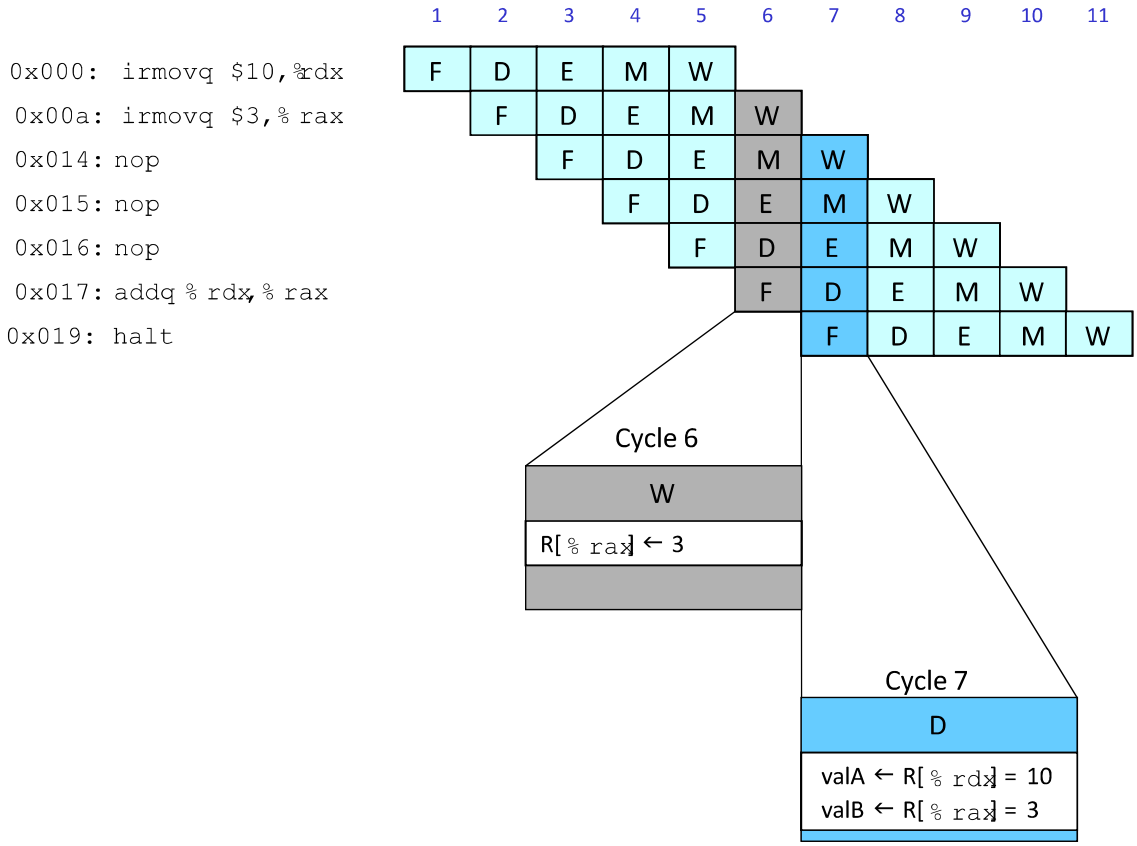


Pipeline Hazards

- Introducing pipelining into a system with feedback can lead to problems when there are dependencies between successive instructions which can take two forms:
  - Data dependencies where the results computed by one instruction are used as the data for a following instruction
  - Control dependencies where one instruction determines the location of the following instruction, such as when executing a jump, call, or return.
- When such dependencies have the potential to cause an erroneous computation by the pipeline, they are called hazards. Hazards are therefore classified as data hazards or control hazards.

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
```

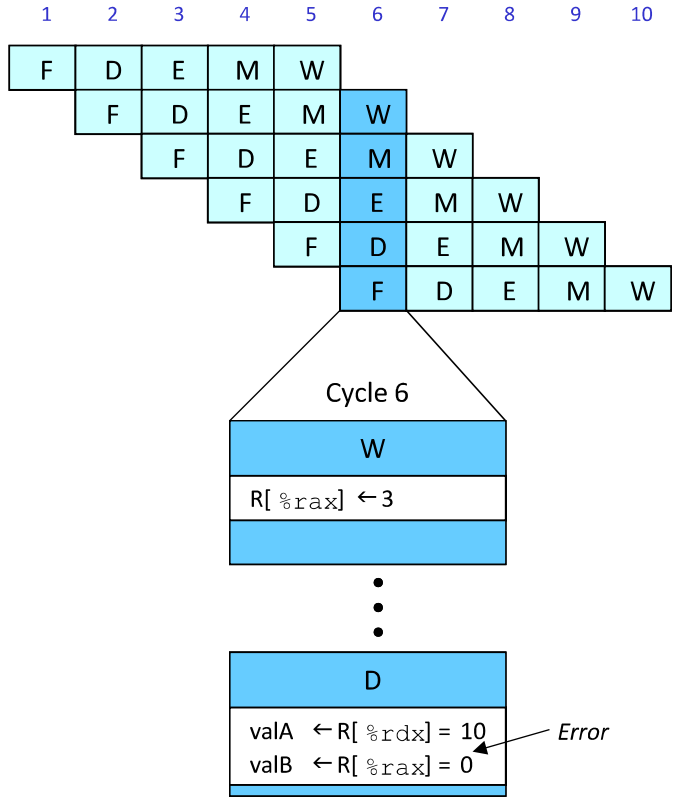


Data Dependencies: 3 Nop's

- Assume program registers initially all have value 0
- Code loads values 10 and 3 into the program registers %rdx and %rax, executes 3 nop's and then adds %rdx to %rax
- Pipeline stages for cycles 6 and 7 are shown
- Expanded view provides write back activity in cycle 6 and decode activity in cycle 7
- After start of cycle 7, registers %rdx and %rax hold updated values, hence addq instruction reads correct values
- Data dependencies between the two irmovq's and addq have not created data hazards

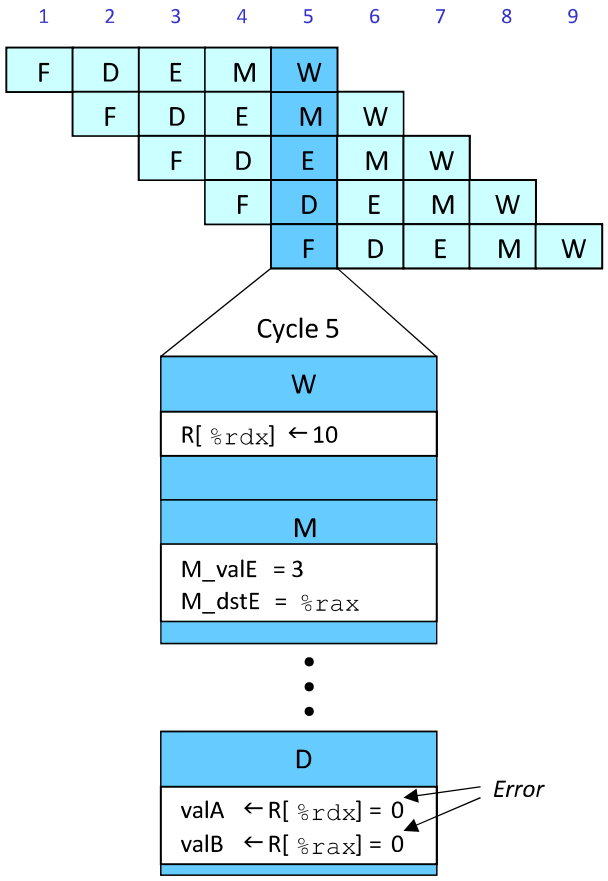
Data Dependencies: 2 Nop's

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



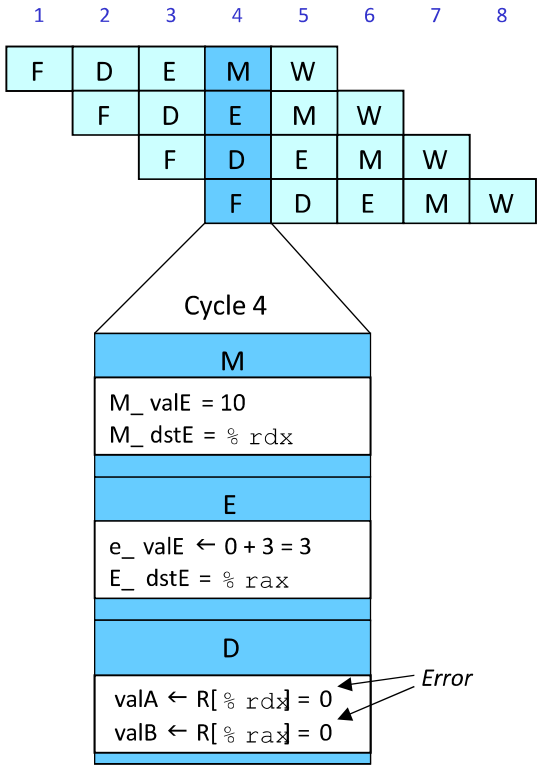
Data Dependencies: 1 Nop

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



Data Dependencies: No Nop

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

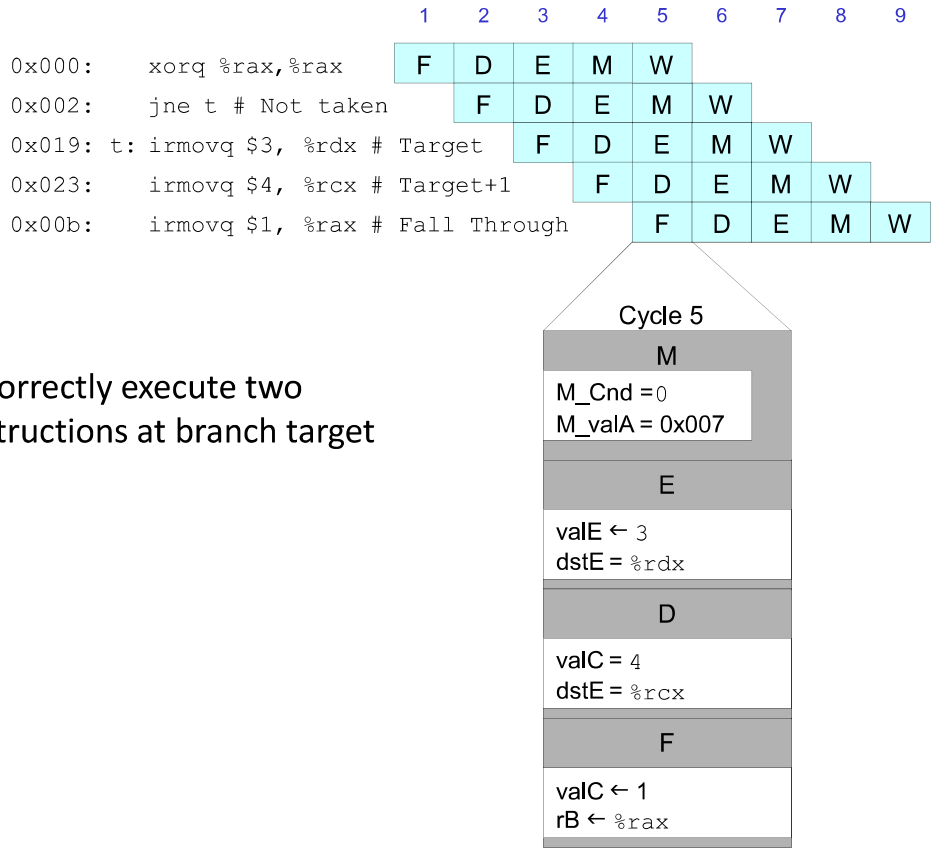


Branch Misprediction Example

```
0x000:     xorq %rax,%rax
0x002:     jne  t              # Not taken
0x00b:     irmovq $1, %rax     # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx     # Target (Should not execute)
0x023:     irmovq $4, %rcx     # Should not execute
0x02d:     irmovq $5, %rdx     # Should not execute
```

– Should only execute first 7 instructions

Branch Misprediction Trace



- Incorrectly execute two instructions at branch target

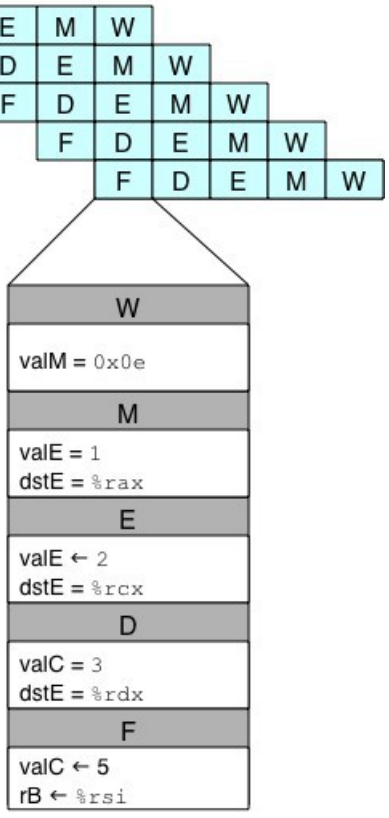
Return Example

```
0x000:    irmovq Stack,%rsp    # Intialize stack pointer
0x00a:    nop                  # Avoid hazard on %rsp
0x00b:    nop
0x00c:    nop
0x00d:    call p               # Procedure call
0x016:    irmovq $5,%rsi       # Return point
0x020:    halt
0x020: .pos 0x20
0x020: p: nop                   # procedure
0x021:    nop
0x022:    nop
0x023:    ret
0x024:    irmovq $1,%rax        # Should not be executed
0x02e:    irmovq $2,%rcx        # Should not be executed
0x038:    irmovq $3,%rdx        # Should not be executed
0x042:    irmovq $4,%rbx        # Should not be executed
0x100: .pos 0x100
0x100: Stack:                   # Initial stack pointer
```

- Require lots of nops to avoid data hazards

Incorrect Return Example

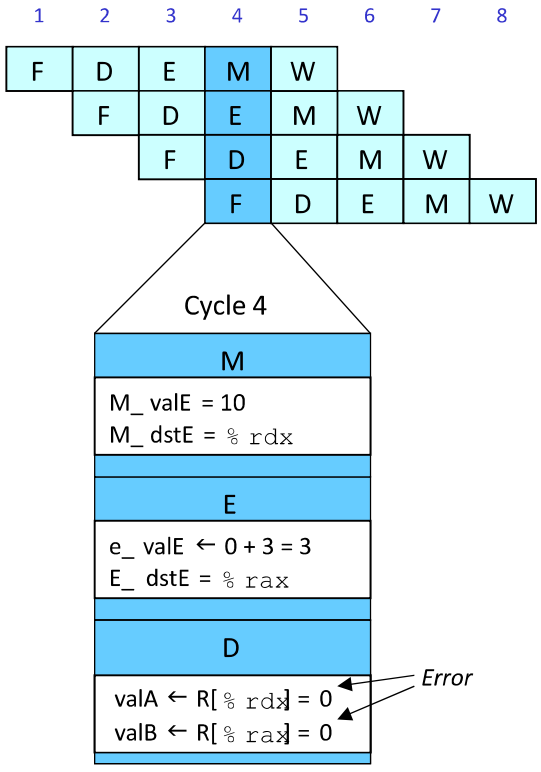
```
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



- Incorrectly execute 3 instructions following `ret`

Data Dependencies: No Nop

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



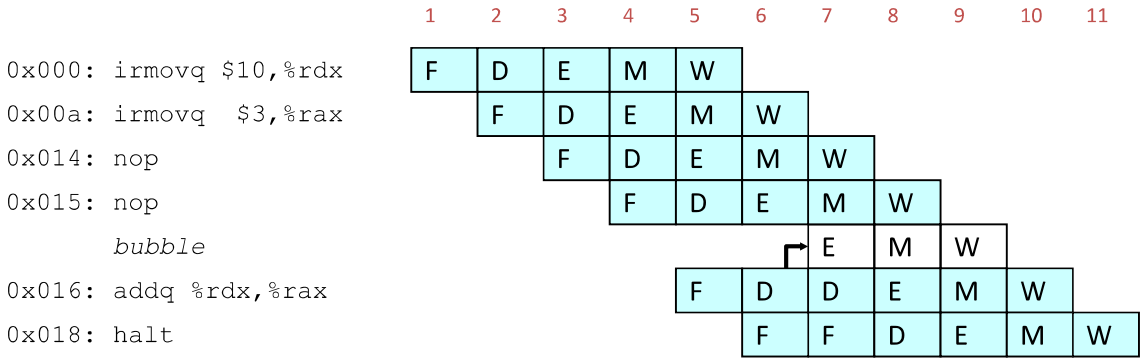
Avoiding data hazards by stalling

- Processor holds back one or more instructions in pipeline until the hazard condition no longer holds – called stalling.
- Rather than letting addq instruction pass through decode stage with incorrect results, processor stalls the instruction
- The next instruction(s) would also need to be stalled – can be done by keeping the program counter at a fixed value (halt instruction will be fetched repeatedly until stall has completed)
- Stalling involves holding back one group of instructions in their stages while allowing other instructions to continue flowing through the pipeline

Avoiding data hazards by stalling

- Handled by injecting a bubble into the execute stage each time we hold back an instruction in the decode stage
- Bubble is like a dynamically generated nop instruction – does not cause any changes to registers, memory, condition codes or program status

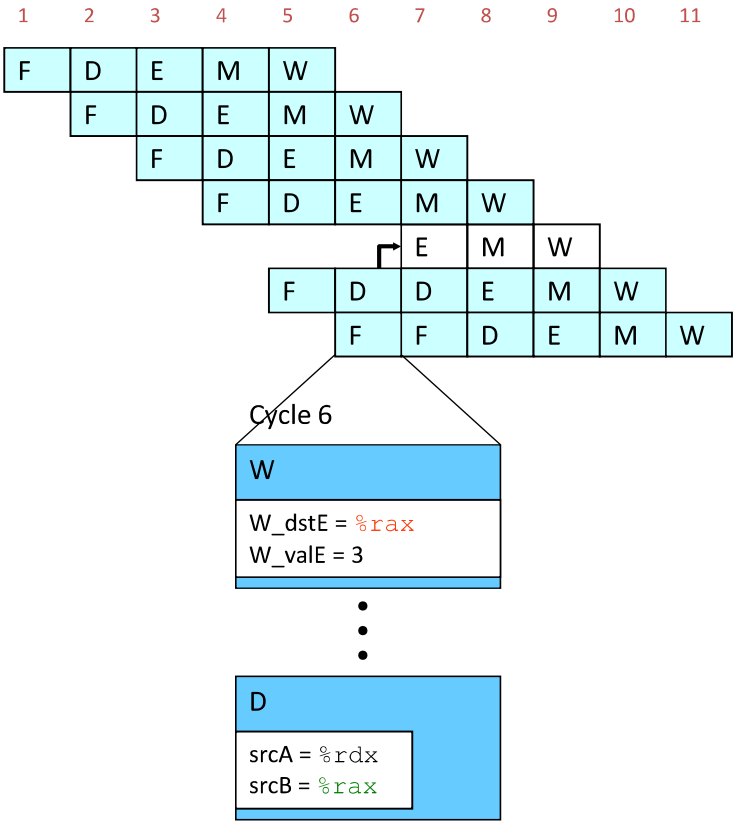
Stalling for Data Dependencies



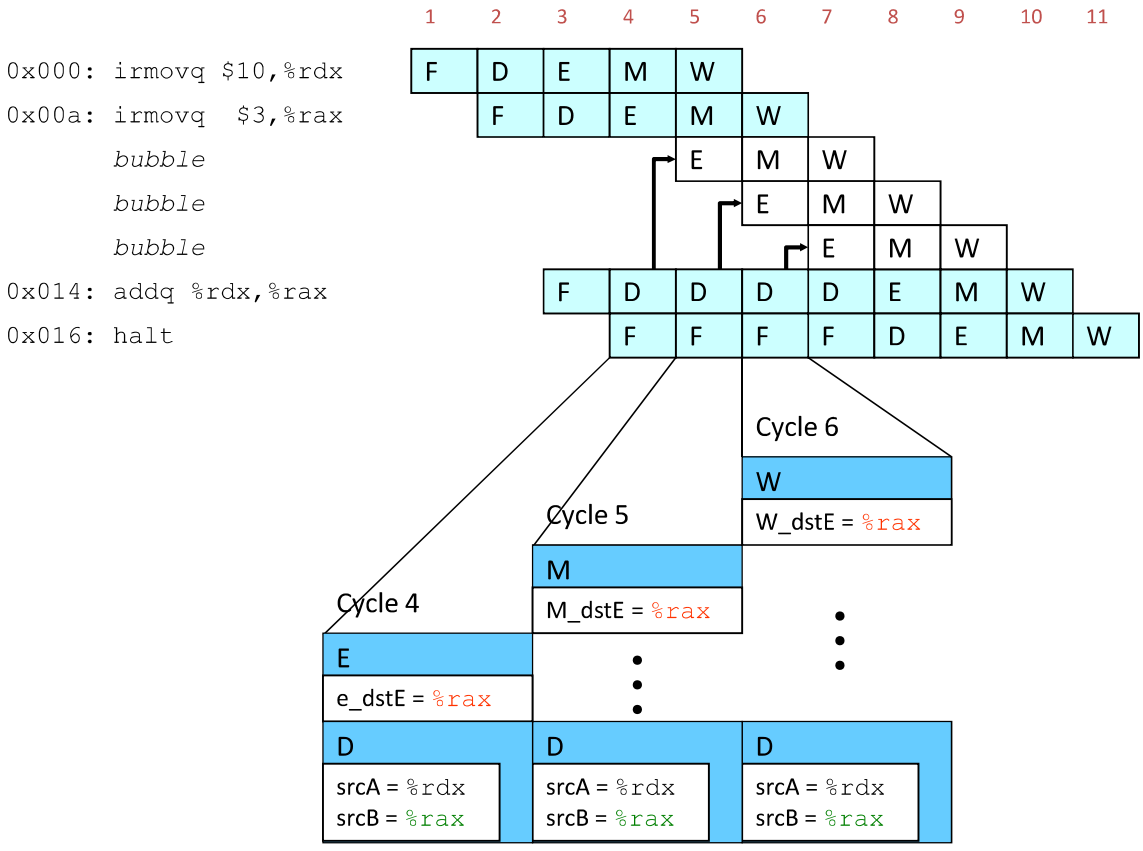
– Injects bubble for execute i.e., repeat decoding of addq instruction
– Has similar effect as nop instructions

Detecting Stall Condition

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



Stalling


0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
bubble
bubble
bubble
0x014: addq %rdx,%rax
0x016: halt
1 2 3 4 5 6 7 8 9 10 11
F D E M W
F D E M W
E M W
E M W
E M W
F D D D D E M W
F F F F D E M W
Cycle 6
W
W_dstE = %rax
Cycle 5
M
M_dstE = %rax
Cycle 4
E
e_dstE = %rax
D
srcA = %rdx
srcB = %rax
D
srcA = %rdx
srcB = %rax
D
srcA = %rdx
srcB = %rax

What Happens When Stalling?

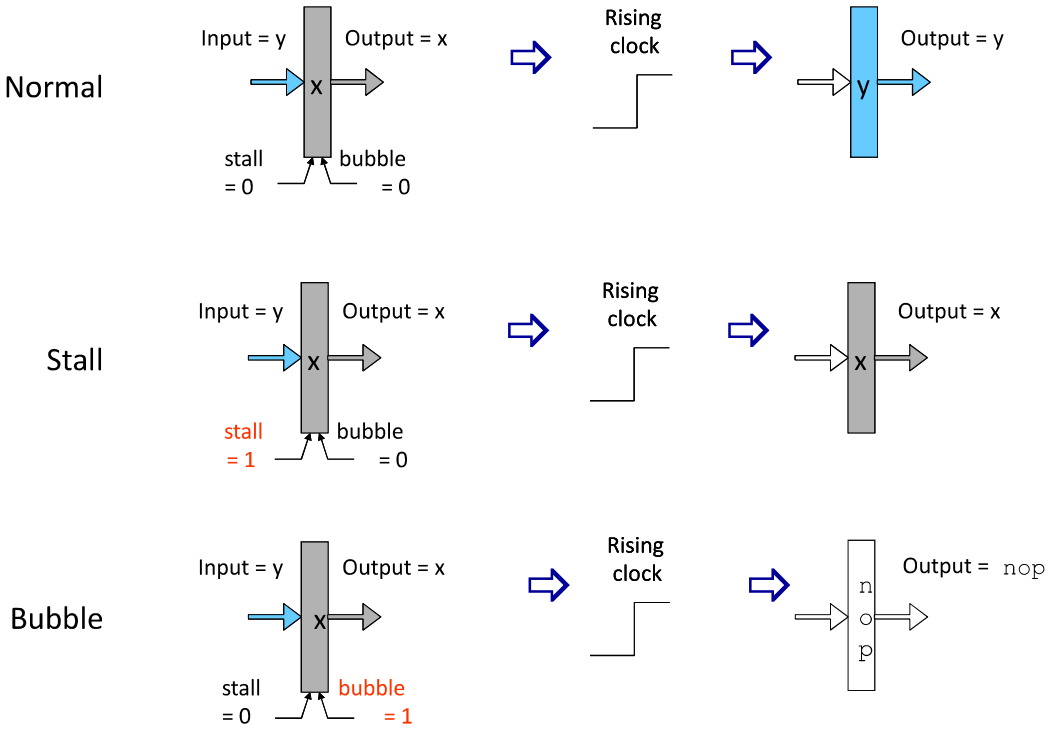
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8

Write Back	*bubble*
Memory	*bubble*
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

- Stalling instruction held back in decode stage
- Following instruction stays in fetch stage
- Bubbles injected into execute stage
  - Like dynamically generated nop's
  - Move through later stages

Pipeline Register Modes

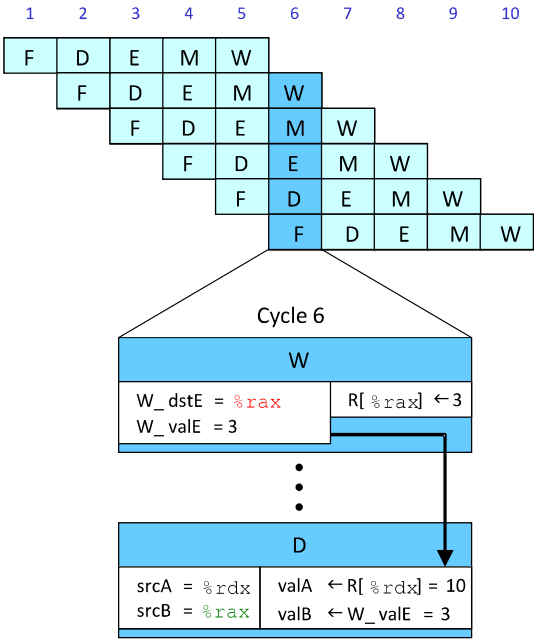


Data Forwarding

- Naive Pipeline
  - Register isn't written until completion of write-back stage
  - Source operands read from register file in decode stage
    - Needs to be in register file at start of stage
- Observation
  - Value generated in execute or memory stage
- Trick
  - Pass value directly from generating instruction to decode stage
  - Needs to be available at end of decode stage

Data Forwarding Example

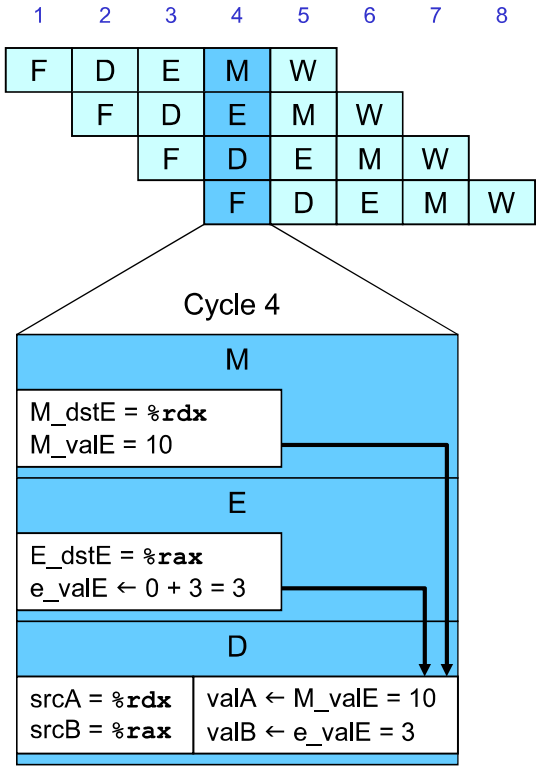
```
0x000:   irmovq $10, %rdx
0x00a:  irmovq $3, %rax
0x014: nop
0x015: nop
0x016: addq %rdx, %rax
0x018: halt
```



- `irmovq` in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

Data Forwarding Example #2

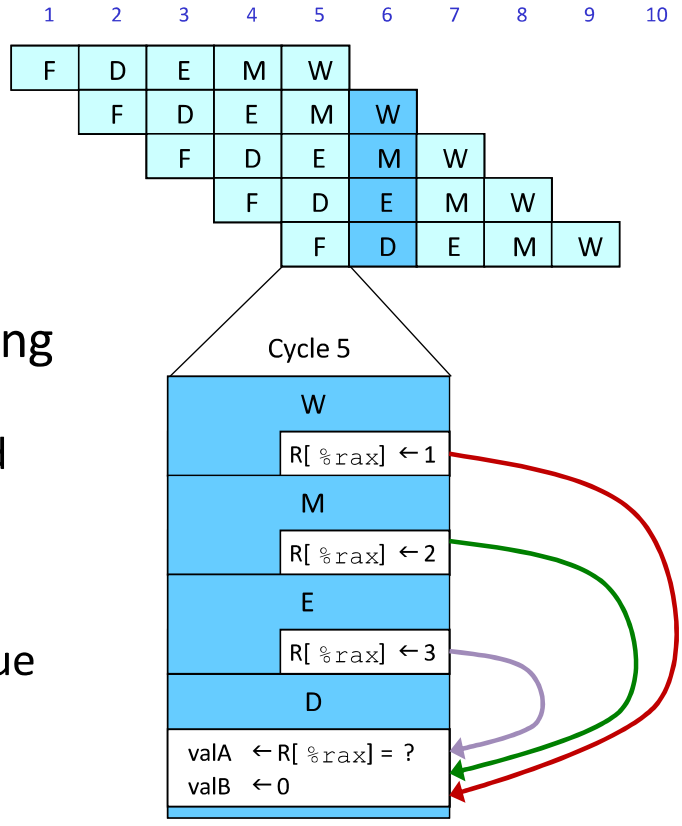
```
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



- Register `%rdx`
  - Generated by ALU during previous cycle
  - Forward from memory as valA
- Register `%rax`
  - Value just generated by ALU
  - Forward from execute as valB

Forwarding Priority

```
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```



- Multiple Forwarding Choices
  - Which one should have priority
  - Match serial semantics
  - Use matching value from earliest pipeline stage