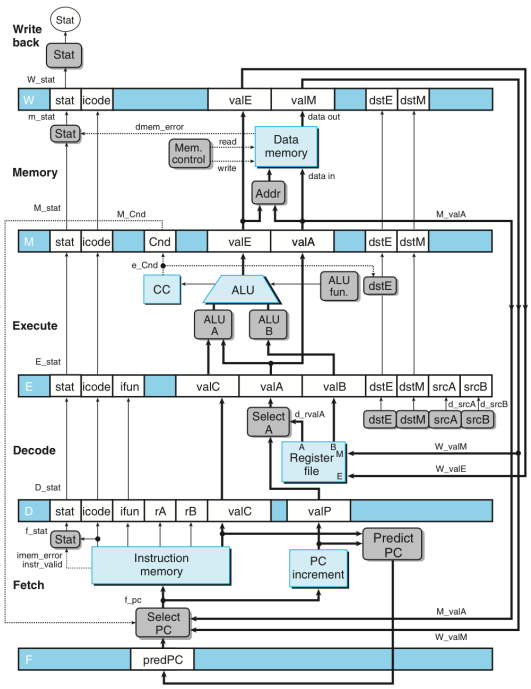


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

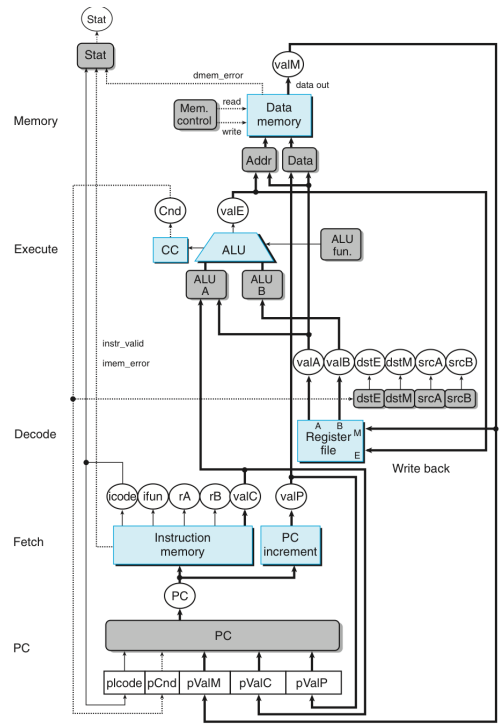


Figure 4.40 **SEQ+ hardware structure.** Shifting the PC computation from the end of the clock cycle to the beginning makes it more suitable for pipelining.

F holds a *predicted* value of the program counter, as will be discussed shortly.

D sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.

E sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

M sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about branch conditions and branch targets for processing conditional jumps.

W sits between the memory stage and the 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.

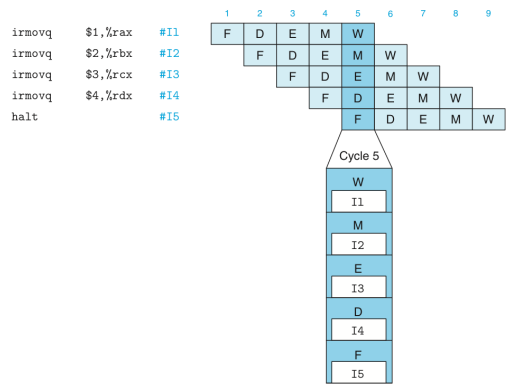
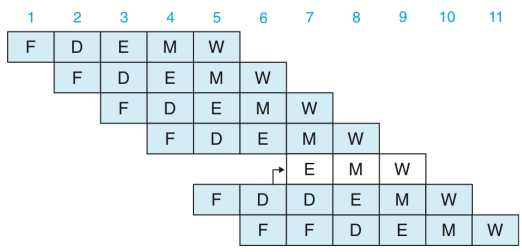


Figure 4.42 **Example of instruction flow through pipeline.**

plete our design. These dependencies can take two forms: (1) *data* dependencies, where the results computed by one instruction are used as the data for a following instruction, and (2) *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 com-

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addlq %rdx,%rax
0x018: halt
```



```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

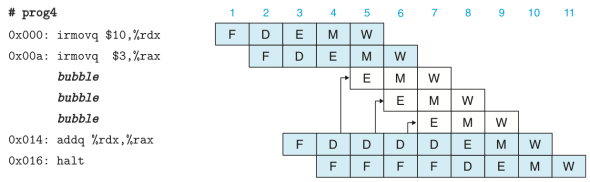


Figure 4.48 Pipelined execution of `prog4` using stalls. After decoding the `addq` instruction in cycle 4, the stall control logic detects data hazards for both source registers. It injects a bubble into the execute stage and repeats the decoding of the `addq` instruction on cycle 5. It again detects hazards for both source registers, injects a bubble into the execute stage, and repeats the decoding of the `addq` instruction on cycle 6. Still, it detects a hazard for source register `%rax`, injects a bubble into the execute stage, and repeats the decoding of the `addq` instruction on cycle 7. In effect, the machine has dynamically inserted three `nop` instructions, giving a flow similar to that shown for `prog1` (Figure 4.43).

In holding back the `addq` instruction in the decode stage, we must also hold back the `halt` instruction following it in the fetch stage. We can do this by keeping the program counter at a fixed value, so that the `halt` instruction will be fetched repeatedly until the stall has completed.

We handle these by injecting a *bubble* into the execute stage each time we hold an instruction back in the decode stage. A bubble is like a dynamically generated `nop` instruction—it does not cause any changes to the registers, the memory, the condition codes, or the program status. These are shown as white boxes in the pipeline diagrams of Figures 4.47 and 4.48. In these figures the arrow between

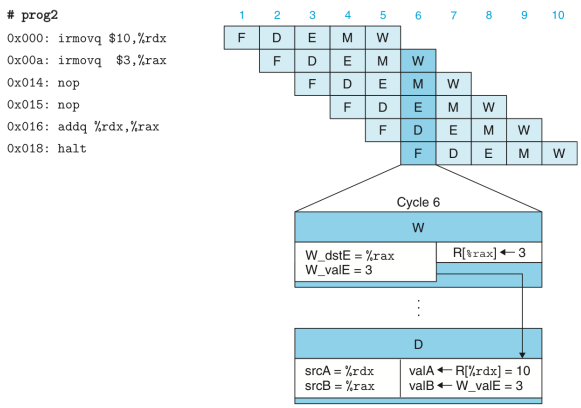


Figure 4.49 Pipelined execution of `prog2` using forwarding. In cycle 6, the decode-stage logic detects the presence of a pending write to register `%rax` in the write-back stage. It uses this value for source operand valB rather than the value read from the register file.

the write-back stage. Rather than stalling until the write has completed, it can simply pass the value that is about to be written to pipeline register E as the source operand. Figure 4.49 shows this strategy with an expanded view of the

Data forwarding (bypassing)

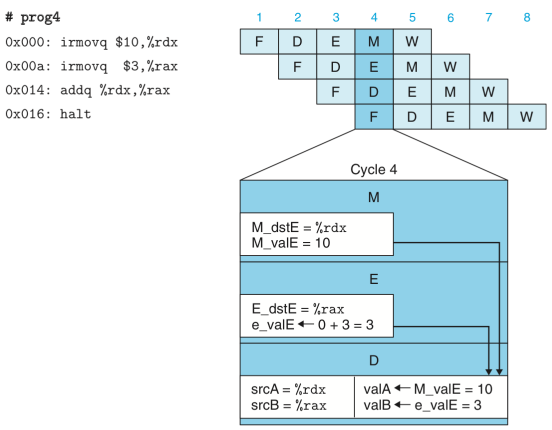


Figure 4.51 Pipelined execution of `prog4` using forwarding. In cycle 4, the decode-stage logic detects a pending write to register `%rdx` in the memory stage. It also detects that a new value is being computed for register `%rax` in the execute stage. It uses these as the values for valA and valB rather than the values read from the register file.

register `%rax`. It can use the value in the memory stage (signal M_valE) for operand valA. It can also use the ALU output (signal e_valE) for operand valB. Note that using the ALU output does not introduce any timing problems. The decode stage

Load/Use hazards

One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline. Figure 4.53 illustrates an example of a *load/use hazard*, where one instruction (the `mrmovq` at address `0x028`) reads a value from memory for register `%rax` while the next instruction (the `addq` at address `0x032`) needs this value as a source operand. Expanded views of cycles 7 and 8 are shown

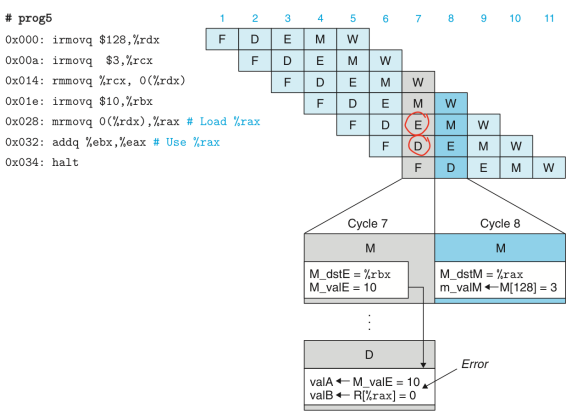


Figure 4.53 Example of load/use data hazard. The `addq` instruction requires the value of register `%rax` during the decode stage in cycle 7. The preceding `mrmovq` reads a new value for this register during the memory stage in cycle 8, which is too late for the `addq` instruction.

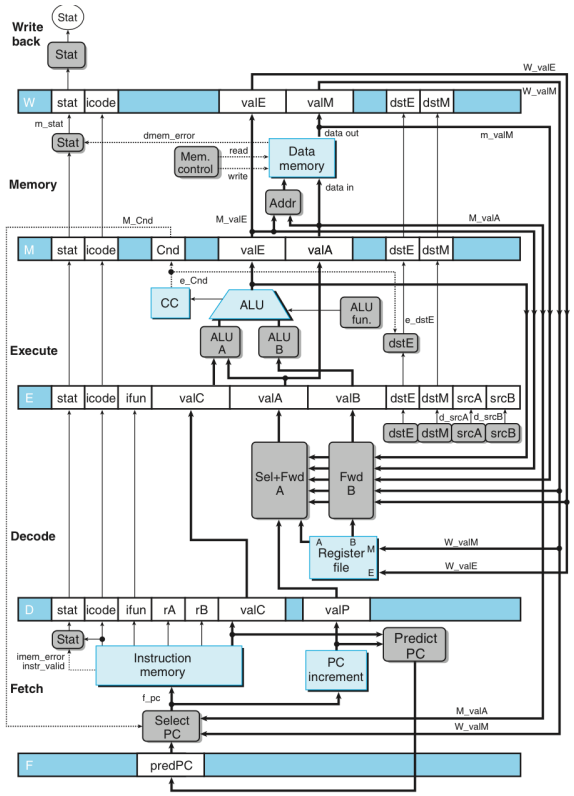


Figure 4.52 Hardware structure of PIPE, our final pipelined implementation. The additional bypassing paths enable forwarding the results from the three preceding instructions. This allows us to handle most forms of data hazards without stalling the pipeline.

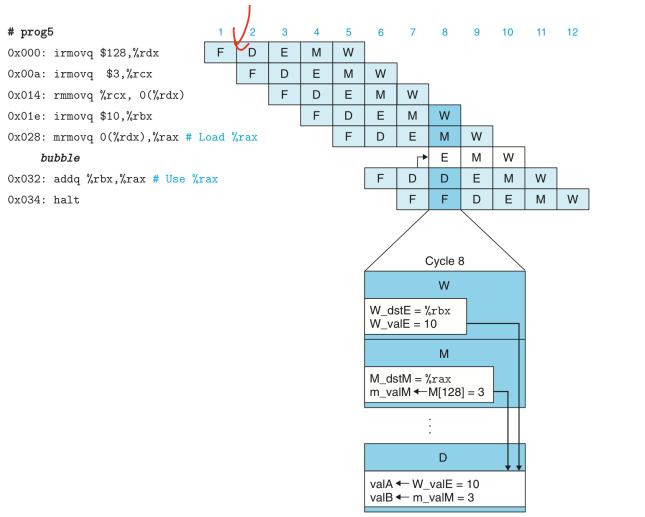


Figure 4.54 Handling a load/use hazard by stalling. By stalling the `addq` instruction for one cycle in the decode stage, the value for valB can be forwarded from the `mrmovq` instruction in the memory stage to the `addq` instruction in the decode stage.

mispredicted branch

```
0x000:    xorq %rax,%rax
0x002:    jne  target        # Not taken
0x00b:    irmovq $1, %rax    # Fall through
0x015:    halt
0x016: target:
0x016:    irmovq $2, %rdx    # Target
0x020:    irmovq $3, %rbx    # Target+1
0x02a:    halt
```

tions are listed in the order they enter the pipeline, rather than the order they occur in the program. Since the jump instruction is predicted as being taken, the instruction at the jump target will be fetched in cycle 3, and the instruction following this one will be fetched in cycle 4. By the time the branch logic detects that the jump should not be taken during cycle 4, two instructions have been fetched that should not continue being executed. Fortunately, neither of these instructions has caused

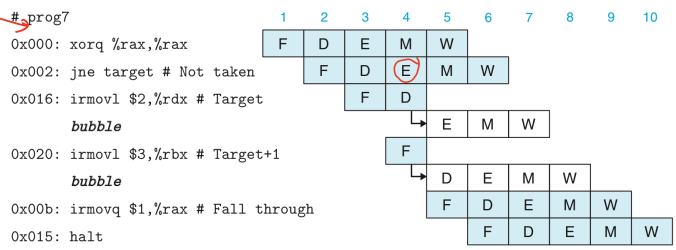


Figure 4.56 Processing mispredicted branch instructions. The pipeline predicts branches will be taken and so starts fetching instructions at the jump target. Two instructions are fetched before the misprediction is detected in cycle 4 when the jump instruction flows through the execute stage. In cycle 5, the pipeline *cancels* the two target instructions by injecting bubbles into the decode and execute stages, and it also fetches the instruction following the jump.

broken. Exceptions can be generated either *internally*, by the executing program, or *externally*, by some outside signal. Our instruction set architecture includes three different internally generated exceptions, caused by (1) a `halt` instruction, (2) an instruction with an invalid combination of instruction and function code, and (3) an attempt to access an invalid address, either for instruction fetch or data read or write. A more complete processor design would also handle external

Exception Handling

possible to have exceptions triggered by multiple instructions simultaneously. For example, during one cycle of pipeline operation, we could have a `halt` instruction in the fetch stage, and the data memory could report an out-of-bounds data address for the instruction in the memory stage. We must determine which of these exceptions the processor should report to the operating system. The basic rule is to put priority on the exception triggered by the instruction that is furthest along the pipeline. In the example above, this would be the out-of-bounds address attempted

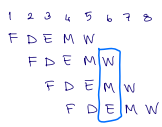
system.

A second subtlety occurs when an instruction is first fetched and begins execution, causes an exception, and later is canceled due to a mispredicted branch. The following is an example of such a program in its object-code form:

```
0x000: 6300                 |    xorq %rax,%rax
0x002: 741600000000000000   |    jne  target      # Not taken
0x00b: 30f00100000000000000 |    irmovq $1, %rax  # Fall through
0x015: 00                   |    halt
0x016:                      | target:
0x016: ff                   |    .byte 0xFF       # Invalid instruction code
```

In this program, the pipeline will predict that the branch should be taken, and so it will fetch and attempt to use a byte with value 0xFF as an instruction (generated in the assembly code using the `.byte` directive). The decode stage will therefore detect an invalid instruction exception. Later, the pipeline will discover that the branch should not be taken, and so the instruction at address 0x016 should never even have been fetched. The pipeline control logic will cancel this instruction, but we want to avoid raising an exception.

A third subtlety arises because a pipelined processor updates different parts of the system state in different stages. It is possible for an instruction following one causing an exception to alter some part of the state before the excepting instruction completes. For example, consider the following code sequence, in which we assume that user programs are not allowed to access addresses at the upper end of the 64-bit range:



```
1    irmovq $1,%rax
2    xorq %rsp,%rsp   # Set stack pointer to 0 and CC to 100
3    pushq %rax       # Attempt to write to 0xfffffffffffffff8
4    addq  %rax,%rax  # (Should not be executed) Would set CC to 000
```

The `pushq` instruction causes an address exception, because decrementing the stack pointer causes it to wrap around to 0xfffffffffffffff8. This exception is detected in the memory stage. On the same cycle, the `addq` instruction is in the execute stage, and it will cause the condition codes to be set to new values. This would violate our requirement that none of the instructions following the

is the motivation for us to include a status code stat in each of our pipeline registers (Figures 4.41 and 4.52). If an instruction generates an exception at some stage in its processing, the status field is set to indicate the nature of the exception. The exception status propagates through the pipeline with the rest of the information for that instruction, until it reaches the write-back stage. At this point, the pipeline control logic detects the occurrence of the exception and stops execution.

To avoid having any updating of the programmer-visible state by instructions

the memory or write-back stages has caused an exception. In the example program above, the control logic will detect that the `pushq` in the memory stage has caused an exception, and therefore the updating of the condition code register by the `addq` instruction in the execute stage will be disabled.

in the same order as they would be executed in a nonpipelined processor, we are guaranteed that the first instruction encountering an exception will arrive first in the write-back stage, at which point program execution can stop and the status code in pipeline register W can be recorded as the program status. If some in-

decode valA

```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```

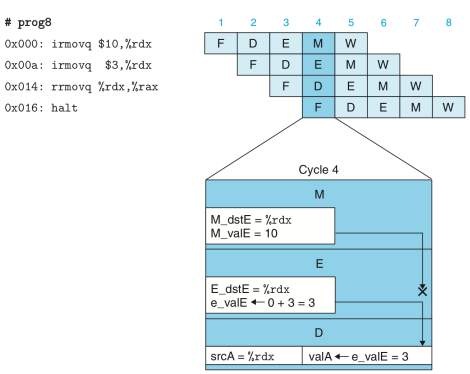


Figure 4.59 **Demonstration of forwarding priority.** In cycle 4, values for `%rdx` are available from both the execute and memory stages. The forwarding logic should choose the one in the execute stage, since it represents the most recently generated value for this register.

```
0x000:    irmovq stack,%rsp  #   Initialize stack pointer
0x00a:    call proc          #   Procedure call
0x013:    irmovq $10,%rdx    #   Return point
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                 # proc:
0x020:    ret                #   Return immediately
0x021:    rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                # stack: Stack pointer
```

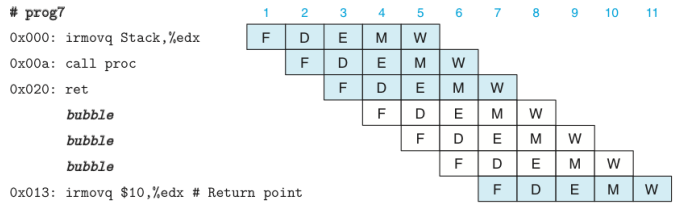
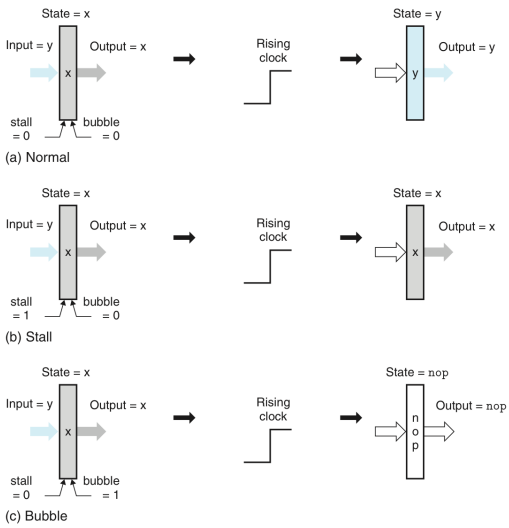


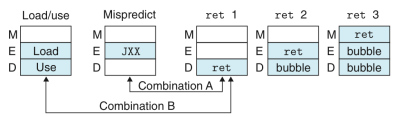
Figure 4.55 Simplified view of `ret` instruction processing. The pipeline should stall while the `ret` passes through the decode, execute, and memory stages, injecting three bubbles in the process. The PC selection logic will choose the return address as the instruction fetch address once the `ret` reaches the write-back stage (cycle 7).



Condition	Pipeline register F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Figure 4.66 Actions for pipeline control logic. The different conditions require altering the pipeline flow by either stalling the pipeline or canceling partially executed instructions.

Figure 4.67 Pipeline states for special control conditions. The two pairs indicated can arise simultaneously.



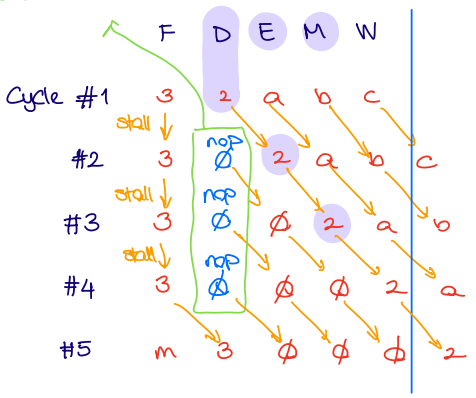
Condition	Pipeline register F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

PIPE
- data hazards
  - data forwarding
  - load/use hazard → data forwarding may not work here → insert a bubble
    - ↳ mrmovq (%rax), %rbx
    - addq %rbx, %rcx
- mispredicted branch
  - ↳ return
  - ↳ jmp

Condition	Pipeline register F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Figure 4.66 **Actions for pipeline control logic.** The different conditions require altering the pipeline flow by either stalling the pipeline or canceling partially executed instructions.

bubble ≡ insert nop



① irmovq 7, %rax

② ret

③

- Detection

Condition	Trigger
Processing `ret`	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Cnd

trigger will start on D stage → insert bubble

lose 3 cycles / lose 1 cycle / lose 2 cycles

- Action (on next cycle)

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

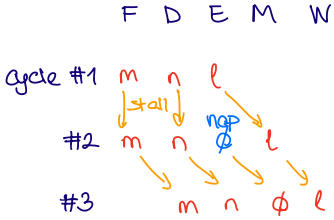
↳ remove the previous 2 instructions while we bubble

as long as those conditions triggered, the stall/bubble process will be active for ret (for the next cycle)

l popq %rcx ≡ mrmovq (%rsp), %rcx

n addq #8, %rsp

m



F D E M W

cycle #1 3 2 1

cycle #2 next ∅ ∅ 1

① Jle target # default→jmp is taken

④ . . . . → if condition does not satisfy

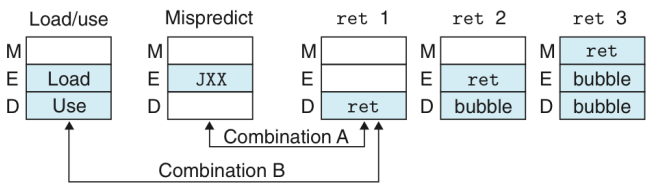
. target

② irmovq . . .

③ rrmovq . . .

What if they come one after the other

Figure 4.67
Pipeline states for special control conditions. The two pairs indicated can arise simultaneously.



Combination A →

M
E JXX) they both trigger the system) should handle as mispredicted branch
D ret

Condition	Pipeline register F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

Combination B

M
E load) load /use should get priority
D ret

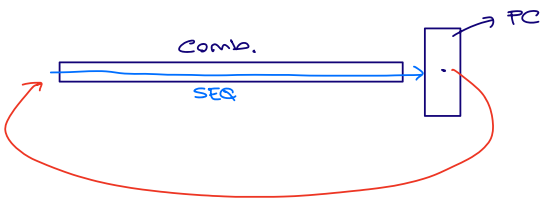
Condition	Pipeline register F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

→ load /use prioritized

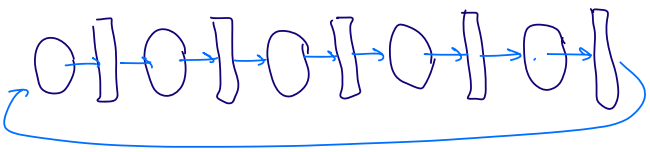
M E D F load ret

F D E M W

PERFORMANCE METRICS



ret	S	B	N	N	N
load	S	S	B		
	S	S	B		



clock rate is much smaller

SEQ → CPI = 1

PIPE →

PIPE — 5 pipe stages

Executable

1 inst.	5 cycles
2 inst.	6 cycles
n inst.	n + 4

$$CPI_{pipe} = \frac{\#C}{\#I} \Rightarrow \frac{\#I + \#B}{\#I} = 1 + \frac{\#B}{\#I} \geq 1$$

#I: number of instructions; #B: number of bubbles inserted

ret → 3
load → 1
jmp → 2