CS356: Discussion #14

Processor Architecture

Marco Paolieri (paolieri@usc.edu)

Illustrations from CS:APP3e textbook

USC University of Southern California

Processor Families

Instruction Set Architecture (ISA)

Instructions supported by a processor (and their byte-level encoding).

- Examples: x86-64, IA32, ARMv8.

Processor Family

Different processors implementing the same ISA.

- Examples: Intel i5 and i7 (x86-64).

The ISA is the shared interface / level of abstraction for:

- Compiler writers (translate C to assembly of an ISA).
- Processor designers (design logic to execute ISA assembly instructions).

Very clever optimizations are adopted by processor designers:

- Pipeline
- Out-of-order execution
- Branch prediction

Recently responsible of security attacks (Meltdown and Spectre).

Main Idea: Parallelism

Take sequential ISA instructions and run them in parallel.

- The result must be the same as sequential execution.

Parallelism at many levels

- At sub-instruction level: pipeline.
- At instruction level: superscalar execution (e.g., two pipelines).
- At thread level: run multiple threads on separate cores.
- At data level: single-instruction multiple-data (SIMD).

Problems

- Data dependencies: the next instruction needs (at some point) the result of the previous one. Cannot run them in parallel!

Clever strategies to deal with data dependencies:

- Out-of-order execution
- Static and dynamic scheduling
- Loop unrolling and renaming

Instruction Sets: RISC and CISC

CISC Processors

- Large number of instructions
- Instructions with long execution time (e.g., memory to memory)
- Complex, variable-size instruction encodings (e.g., 1-15 bytes for x86-64)
- Complex addressing formats, e.g., `movq %rds,2(%rax,%rdx,8)`
- ALU operations applicable to memory and registers: `addq %rcx,(%rax)`
- Stack intensive: use stack for return addresses and arguments (e.g., IA32)

RISC Processors

- Many fewer instructions (less than 100)
- Instructions only for quick, primitive operations
- Fixed-length instruction encoding (typically, 4 bytes)
- Simple addressing formats, e.g., just base and displacement: `2(%rax)`
- ALU operations applicable only to registers: `addq %rcx,%rax`
- Register intensive: use registers for return addresses and arguments.

Today: x86-64 CISC instructions translated by CPU to RISC-like instructions.

Example: Translating to RISC-like assembly

```
// CISC instruction
movq 0x40(%rdi, %rsi, 4), %rax

// RISC equivalent
mov   %rsi, %rbx   // use %rbx as a temp
shl      2, %rbx   // %rsi * 4
add   %rdi, %rbx   // %rdi + (%rsi*4)
add  $0x40, %rbx   // 0x40 + %rdi + (%rsi*4)
mov (%rbx), %rax   // %rax = *%rbx
```

General Principles

- Replace complex addressing with sequence of arithmetic operations
- Replace memory-to-register ALU operations with register-to-register operations and load/store.

RISC: Classroom Instructions

- Load from memory into register:
  - `ld 0x40(%rdi), %rax`
- Store register into memory:
  - `st %rax, 0x40(%rdi)`
- Arithmetic and logic instructions on registers:
  - `add %rdi, %rax`
  - `sub %rdi, %rax`
  - `xor %rdi, %rax`
  - …
- Moves between registers
  - `mov %rdi, %rax`
- Jumps
  - `je 0x123`
  - `jg 0x123`

Example: Translation

```
// example #1
mov (%rdi), %rax
mov 0x40(%rdi), %rax
mov 0x40(%rdi,%rsi), %rax
```

```
ld 0x0(%rdi), %rax
ld 0x40(%rdi), %rax
mov %rsi, %rbx
add %rdi, %rbx
ld  0x40(%rbx), %rax
```

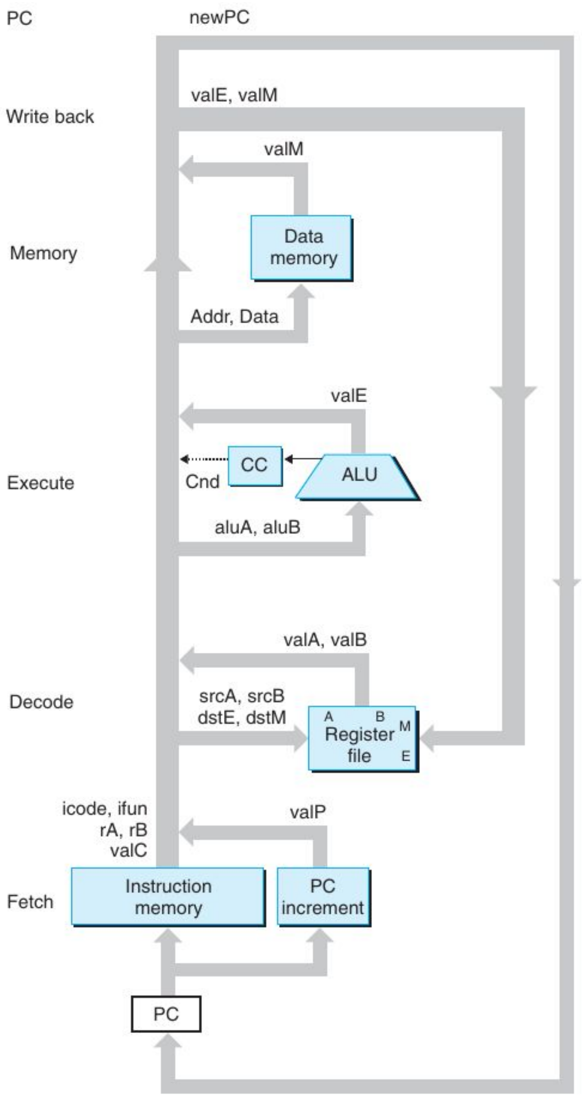
```
// example #2
mov %rax, (%rdi)
mov %rax, 0x40(%rdi)
mov %rax, 0x40(%rdi,%rsi)
```

```
st %rax, 0x0(%rdi)
st %rax, 0x40(%rdi)
mov %rsi, %rbx
add %rdi, %rbx
st  %rax, 0x40(%rbx)
```

```
// example #3
add %rax, (%rsp)
```

```
ld   0(%rsp), %rbx
add  %rax, %rbx
st   %rbx, 0(%rsp)
```

Sequential Processor



On each clock cycle, perform all the steps to run an instruction (so, clock cycle will be large!).

Fetch. Read instruction from memory and extract `icode`, registers `rA`/`rB`, constant `valC`.

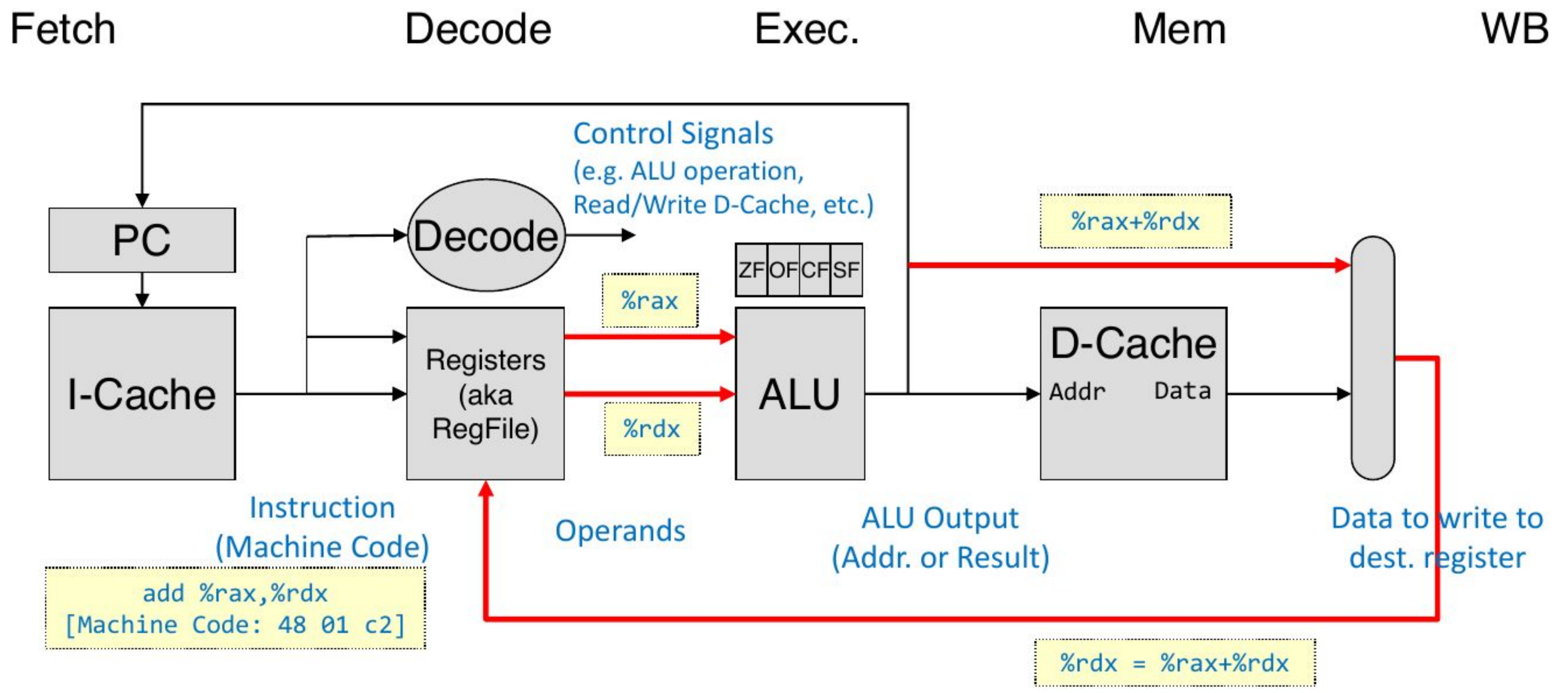
Decode. Read up to 2 operands from register file, obtaining `valA` and `valB` (for ALU operations).

Execute. ALU operation on registers, effective address computation (for **`ld`** and **`st`**). Produces an output value and a condition code.

Memory. Read data from memory to `valM` (for **`ld`**), or write data to memory (for **`st`**). Uses the address computed during Execute.

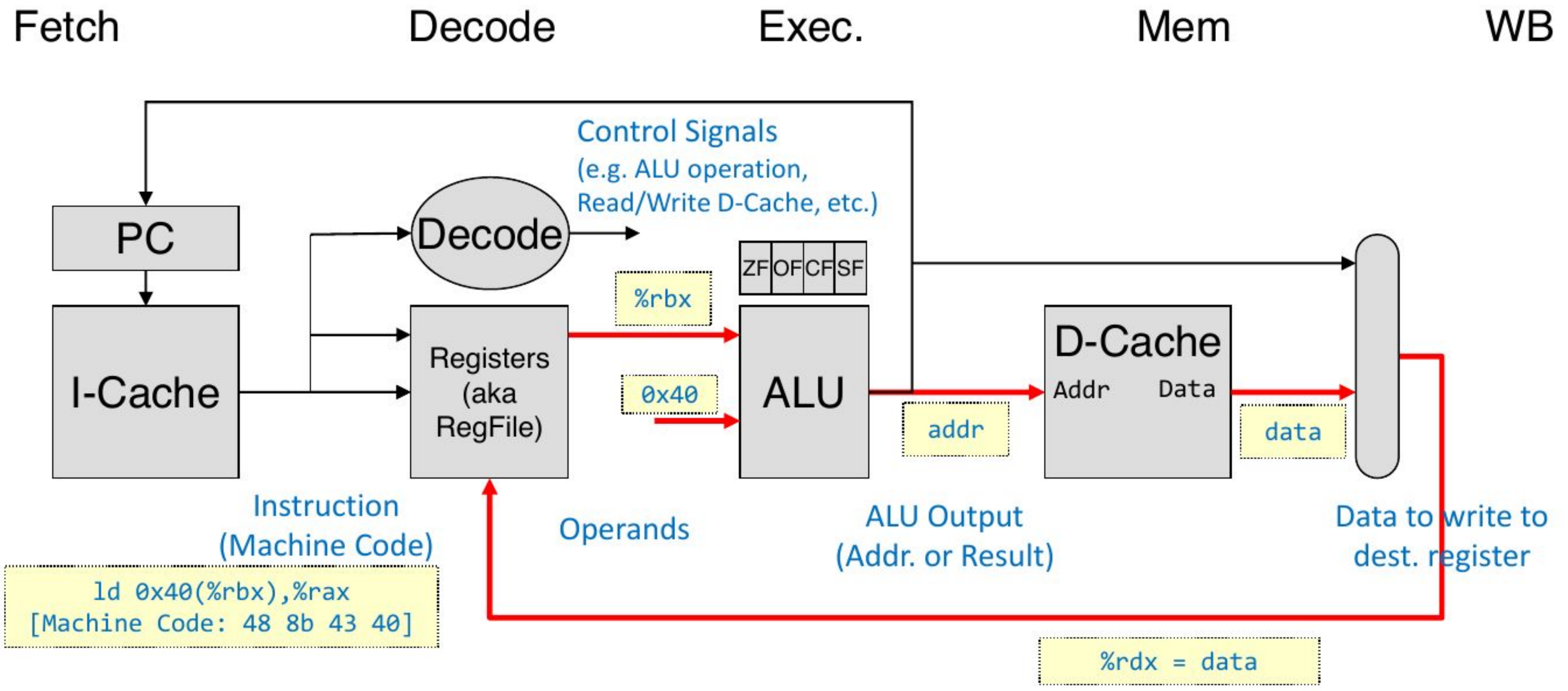
Write Back. Save Ex/Mem output to registers.

Sequential Processor: Add



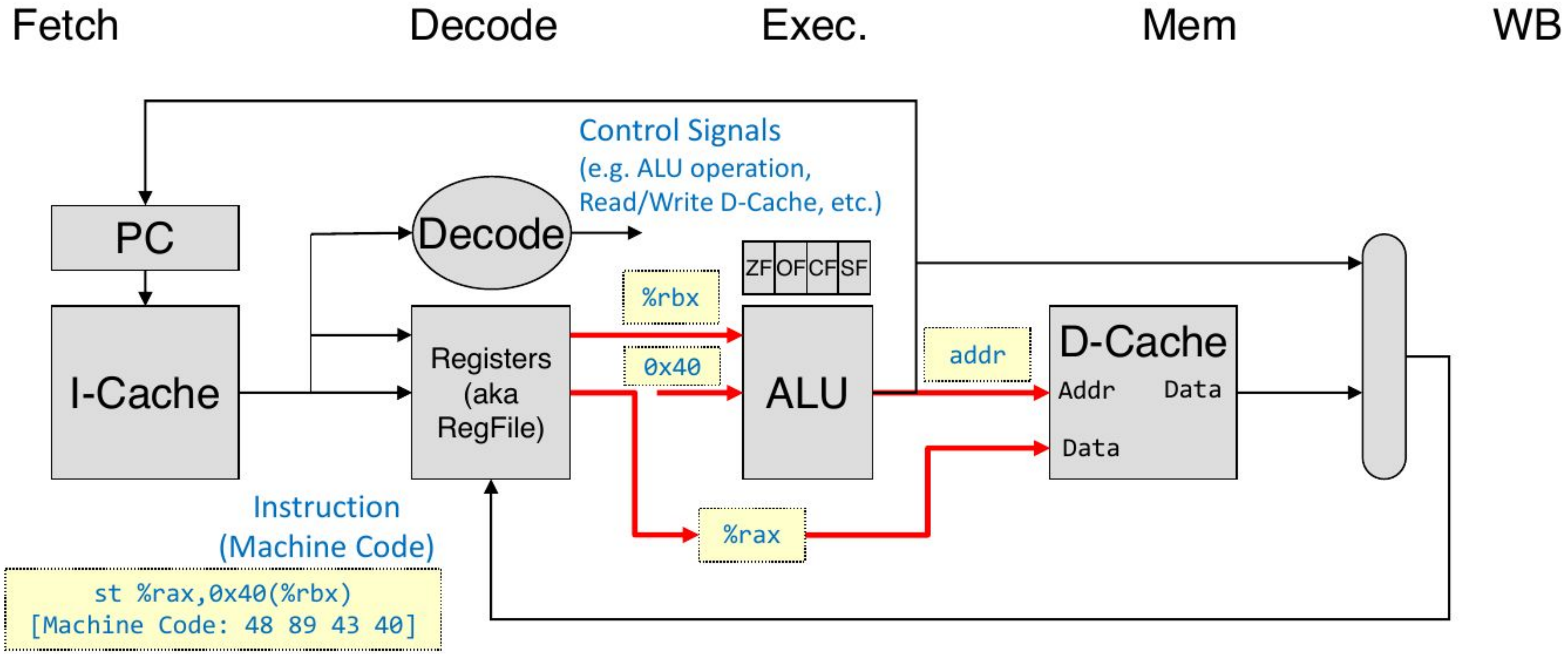
add does not need to access the data cache, **no memory access**.

Sequential Processor: Load



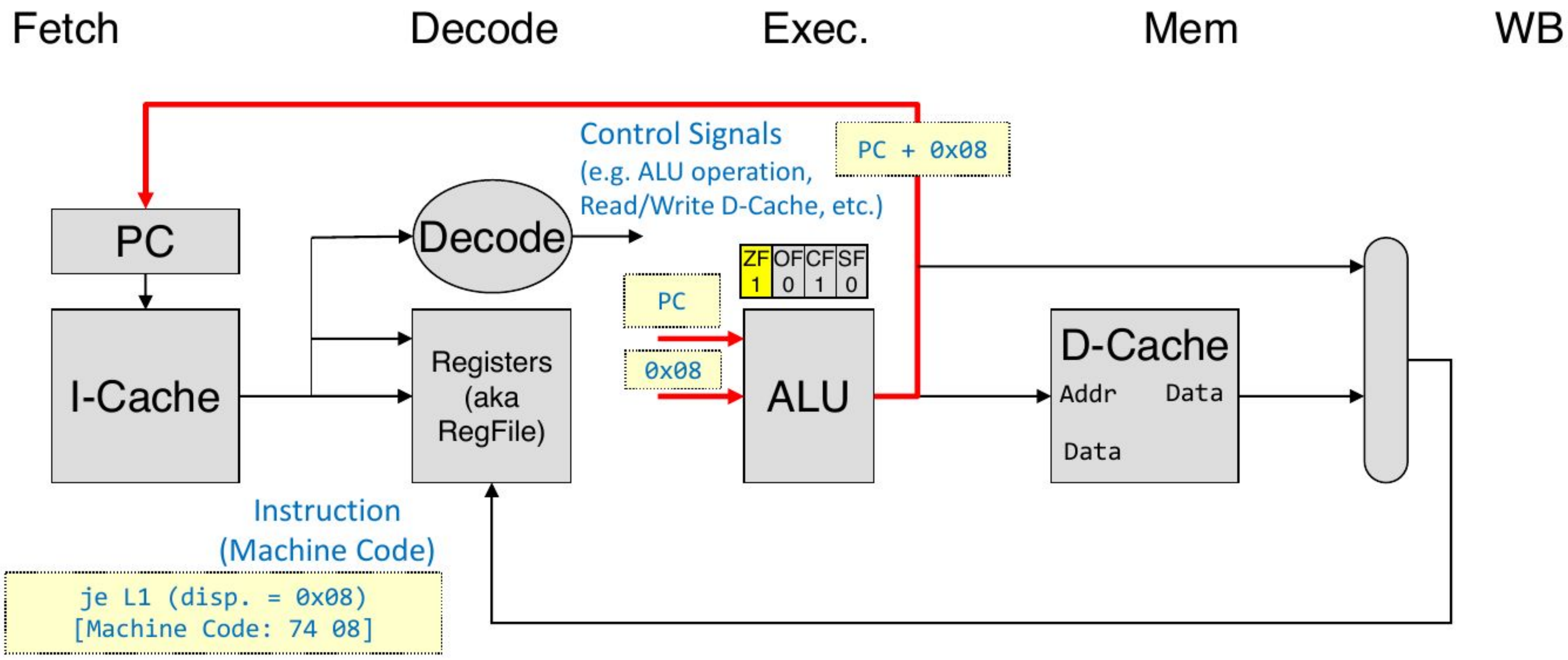
`ld` uses the ALU operation to compute the affective address.

Sequential Processor: Store



`st` uses the ALU operation to compute the affective address, **no write-back**.

Sequential Processor: Jump

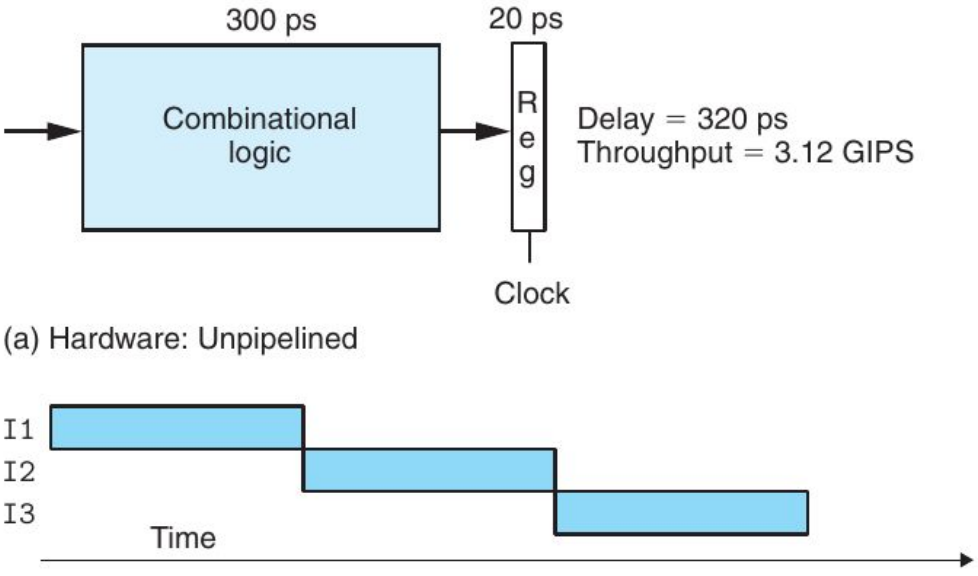


`je` uses condition code and ALU to increment PC,
no memory access, no write-back.

Pipeline: Motivation

The sequential processor executes one instruction at a time.

While one unit (Fetch, Decode, Execute, Memory, Write-Back) is computing, the others are waiting.



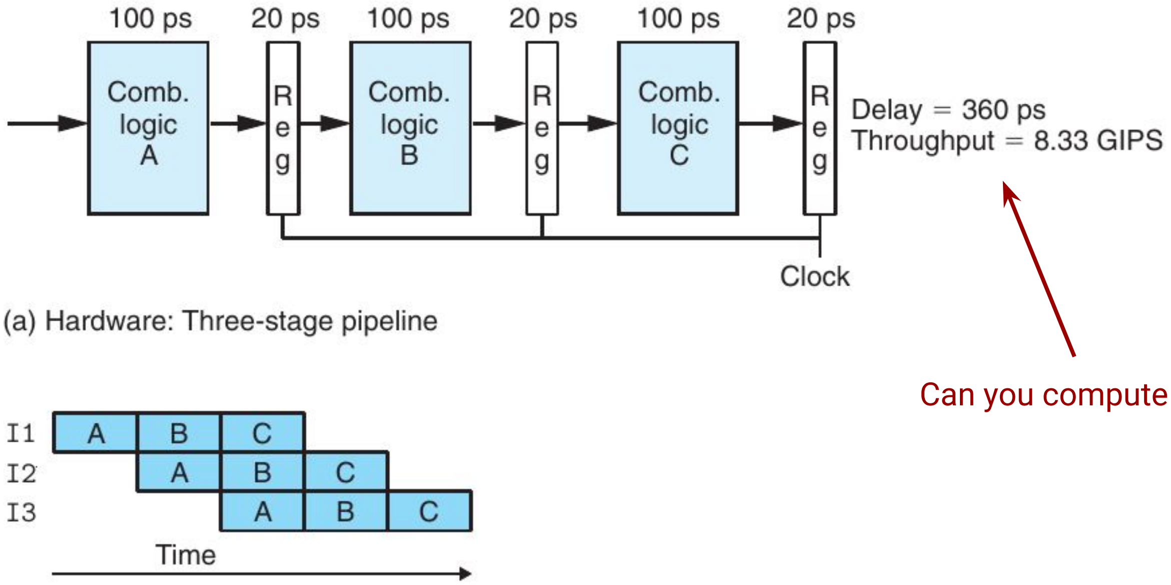
(a) Hardware: Unpipelined

(b) Pipeline diagram

Pipeline: Idea

Add intermediate buffers, process multiple instructions at the same time.

- Increases throughput (instructions processed / second)
- Slightly increases latency (time from start to end of an instruction)

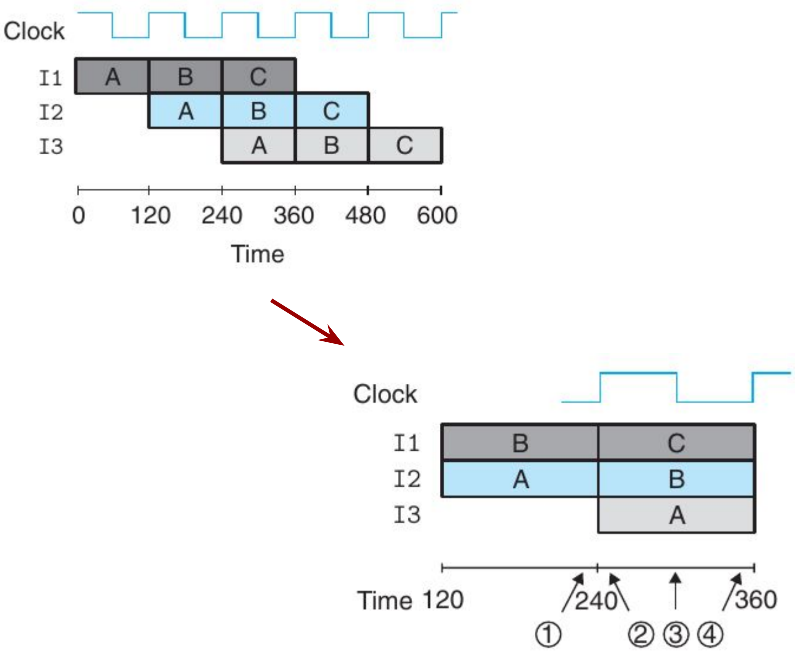


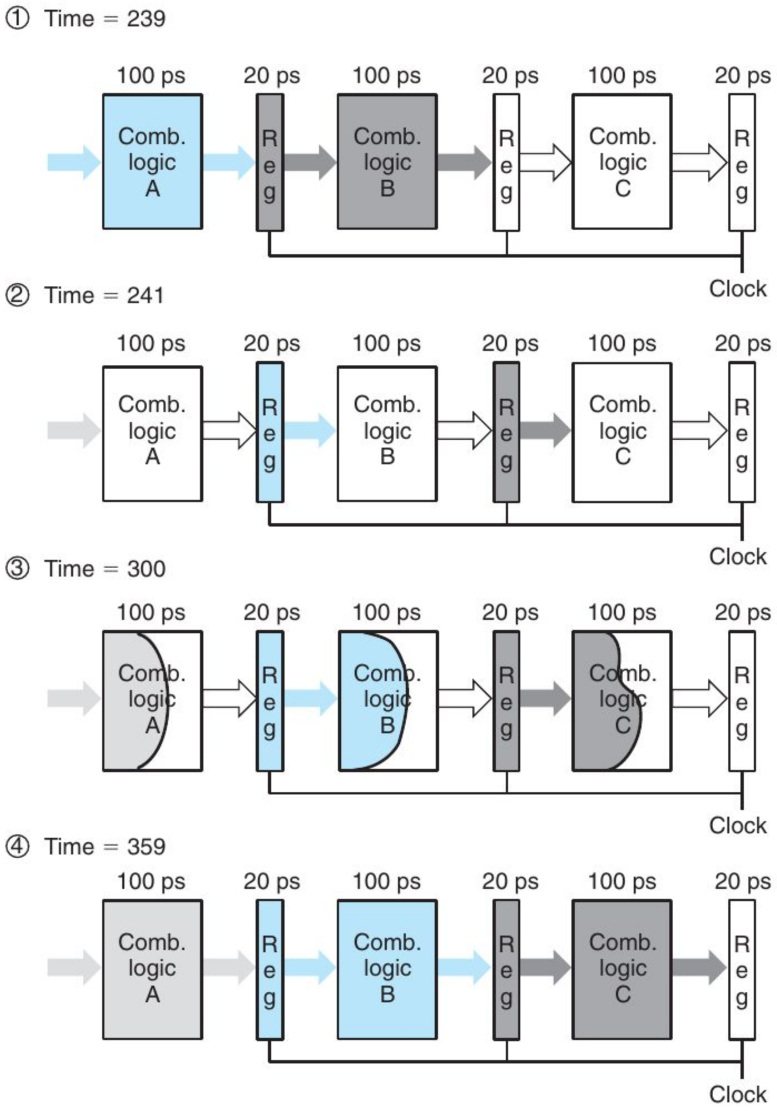
(a) Hardware: Three-stage pipeline

(b) Pipeline diagram

Can you compute these values?

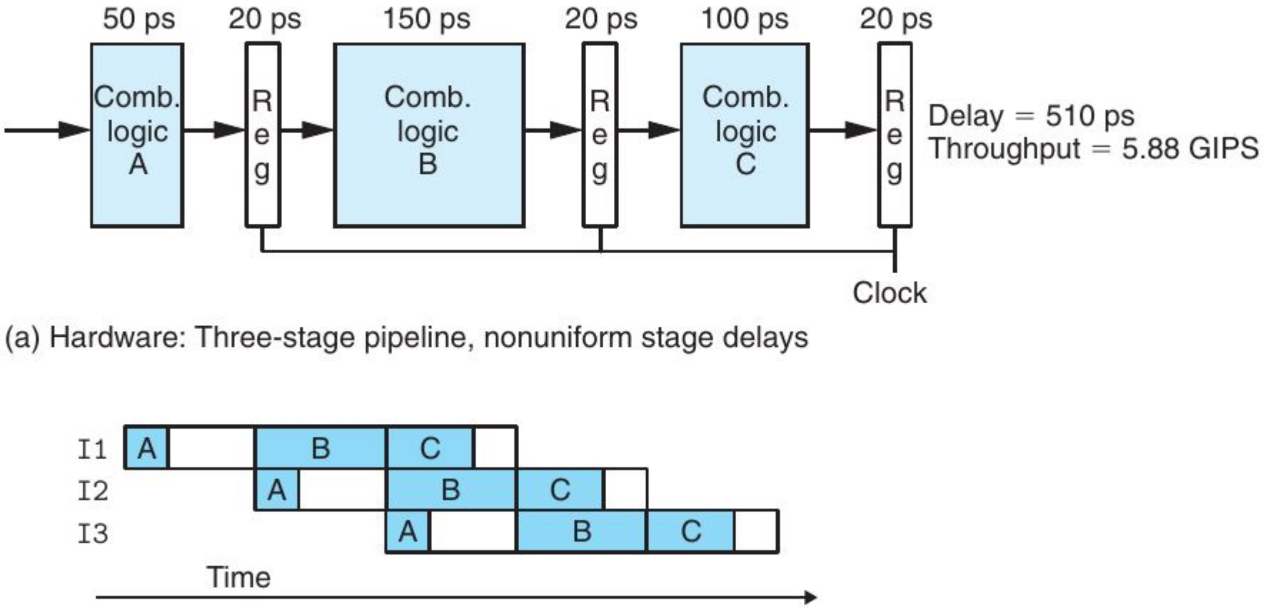
Pipeline: Operation





During each clock cycle, the combinatorial logic of a stage computes the next intermediate result of an instruction.

Pipeline: Non-Uniform Stage Delays



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

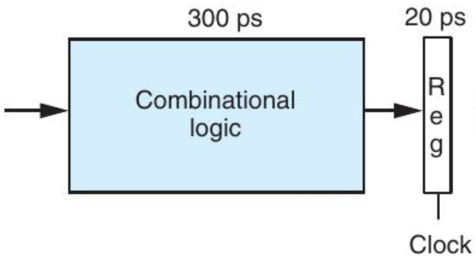
(b) Pipeline diagram

The clock cycle must be greater or equal to the maximum stage delay.

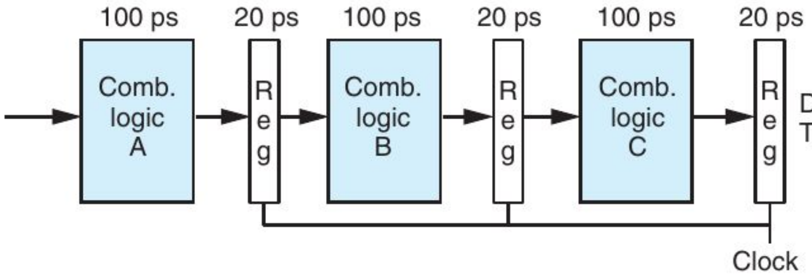
In the example: max(70, 170, 120) = 170 ps, so:

- Delay is 170 × 3 = 510 ps
- Throughput is 1/.17 GIPS

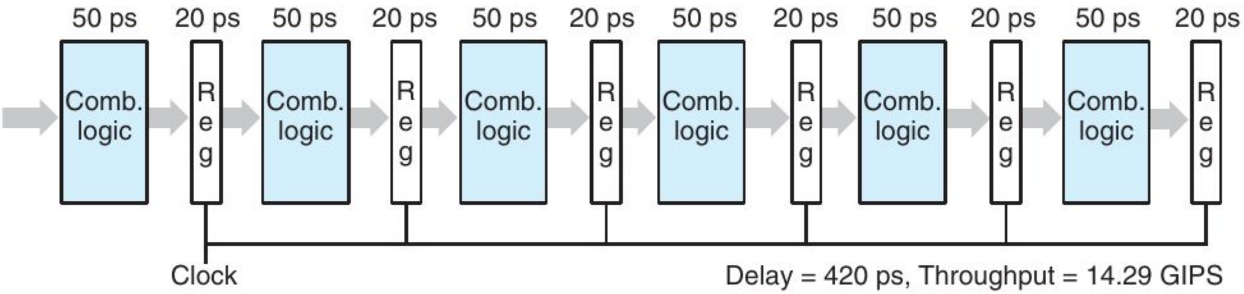
Pipeline: Diminishing Returns of Deep Lines



Delay = 320 ps
Throughput = 3.12 GIPS



Delay = 360 ps
Throughput = 8.33 GIPS

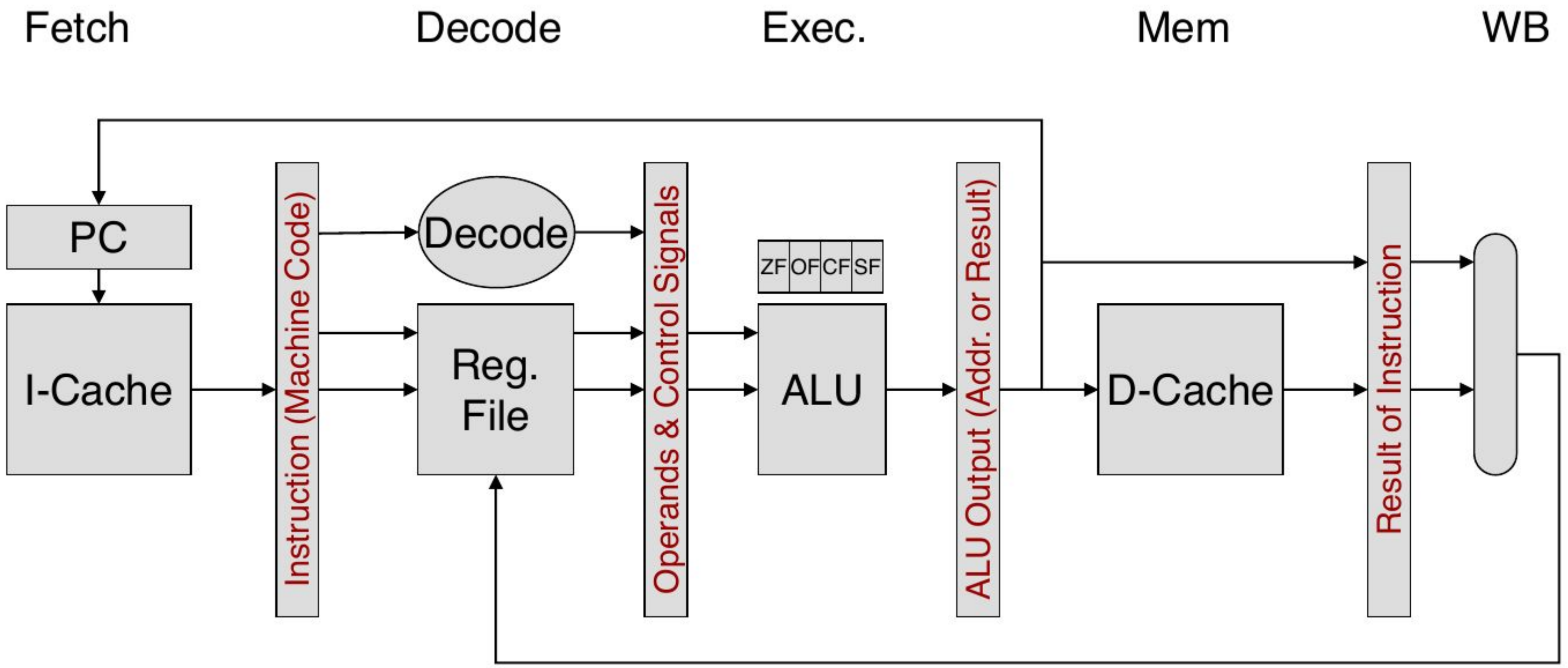


Delay = 420 ps, Throughput = 14.29 GIPS

n	clock (ps)	tput (GIPS)
1	320	3.125
2	170	5.882
3	120	8.333
4	95	10.526
5	80	12.500
6	70	14.286

```
clock = 300/n + 20
tput  = 1/clock
delay = n*clock
```

Pipelined Processor



Note that there can be a pending write to the register file during decode/execute of following instructions.

Pipeline: Hazards

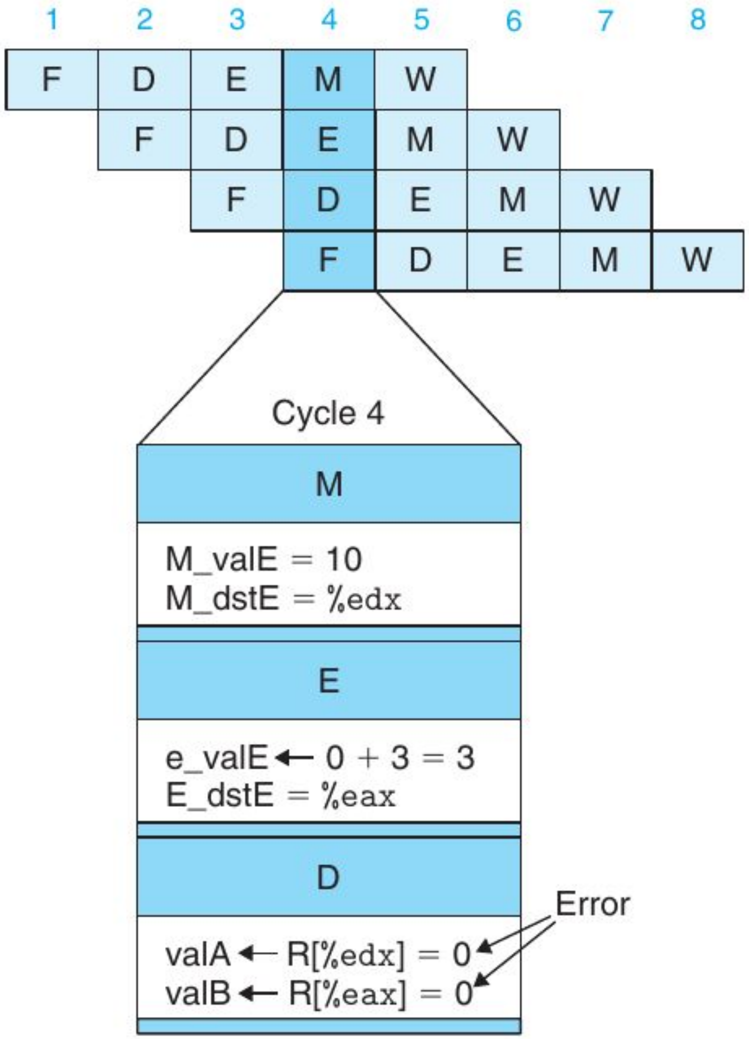
Data Dependencies
The results computed by an instruction are used by the following one.

Control Dependencies
One instruction determines the location of the next one (e.g., jumps).

Sequential dependencies can create **pipeline hazards**.

- Careless pipelining can produce different program behavior!

```
mov $10, %edx
mov $3, %eax
add %edx, %eax
```



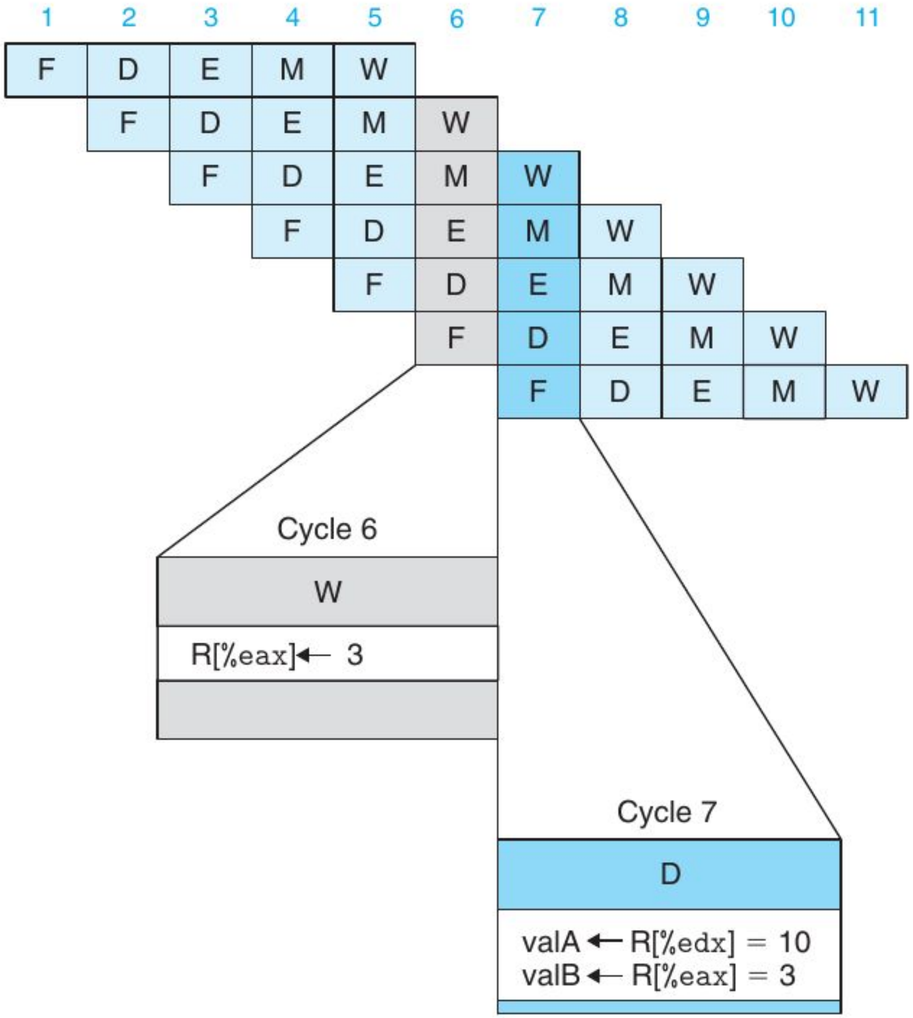
Pipeline: Avoiding Hazards

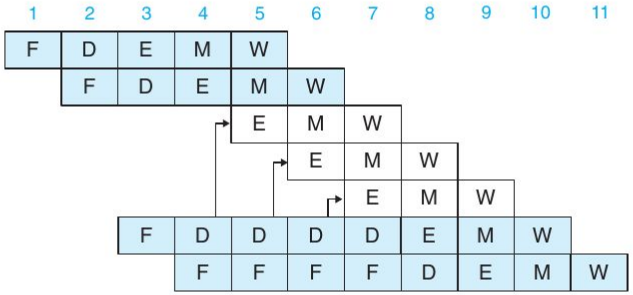
Stalling

Insert no-op and wait for results

```
mov $10, %edx
mov $3, %eax
nop
nop
nop
add %edx, %eax
```

When add is decoding, moves have completed write-back.





Pipeline: Avoiding Hazards

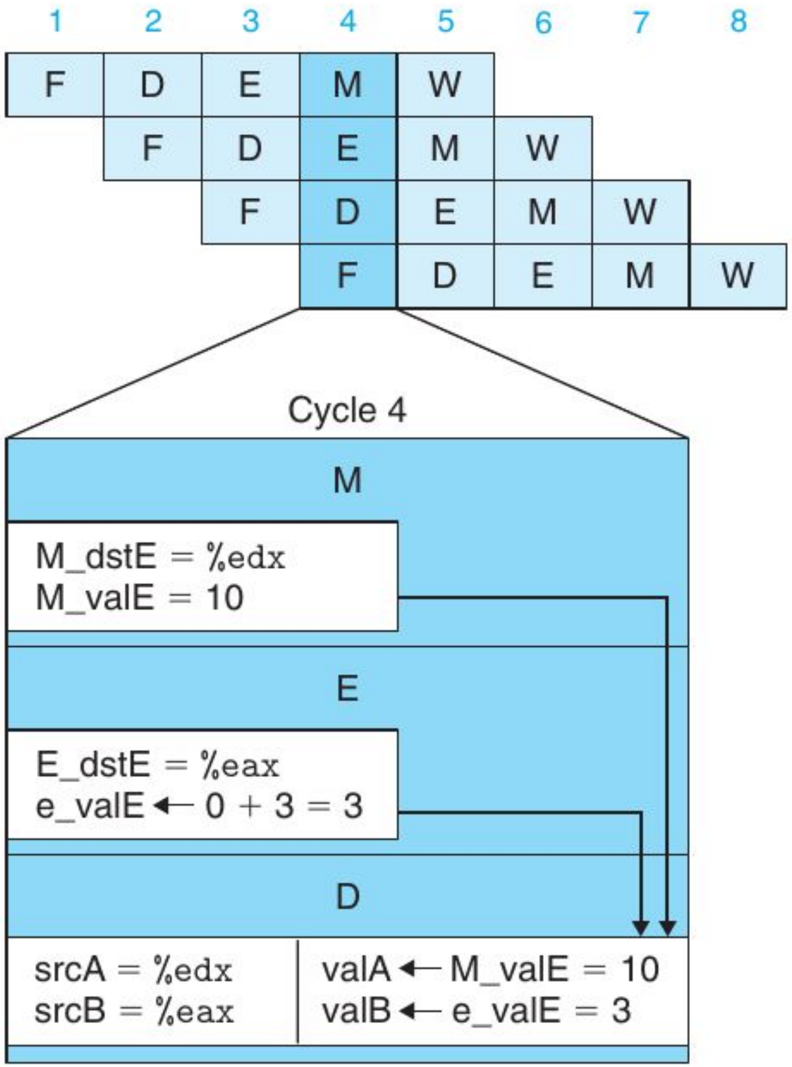
Forwarding

Pass new values to previous stages

```
mov $10, %edx
mov $3, %eax
add %edx, %eax
```

In cycle 4, both mov operations have their output value ready: if forwarding logic is added to the processor, add can read those values during its decode stage.

This is effectively by-passing reads from registers.



1	2	3	4	5	6	7	8
F	D	E	M	W			
	F	D	E	M	W		
		F	D	E	M	W	
			F	D	E	M	W

Cycle 4

M

M_dstE = %edx
M_valE = 10

E

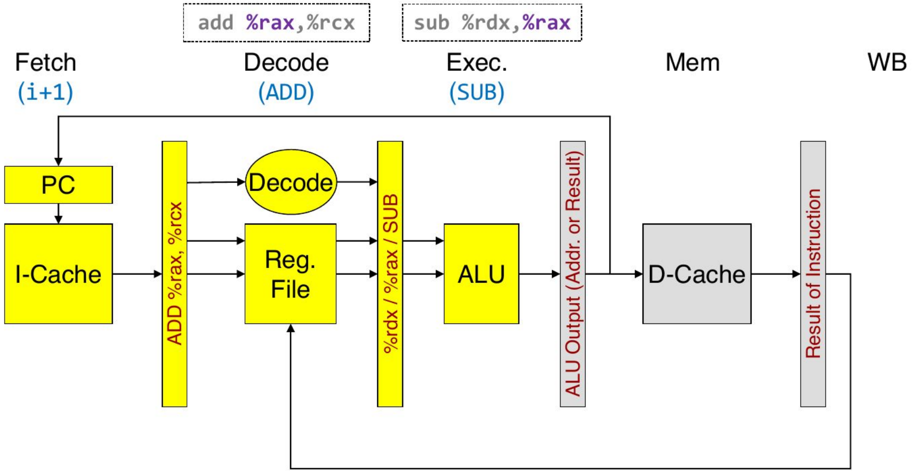
E_dstE = %eax
e_valE ← 0 + 3 = 3

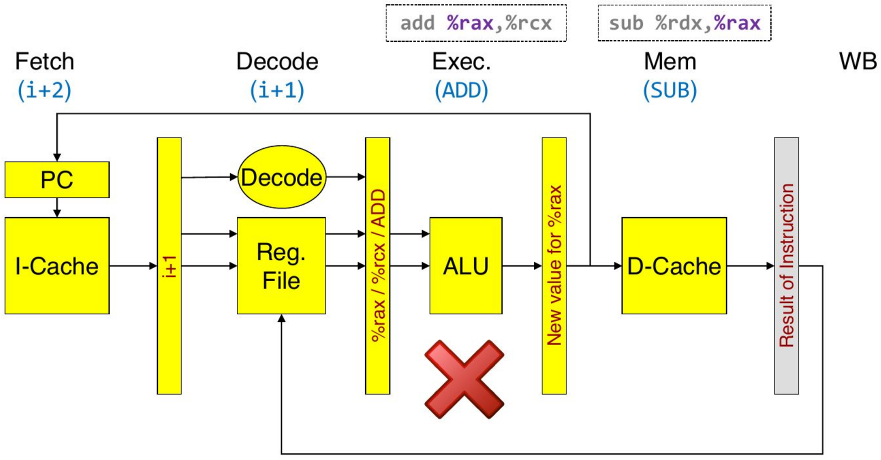
D

srcA = %edx
srcB = %eax

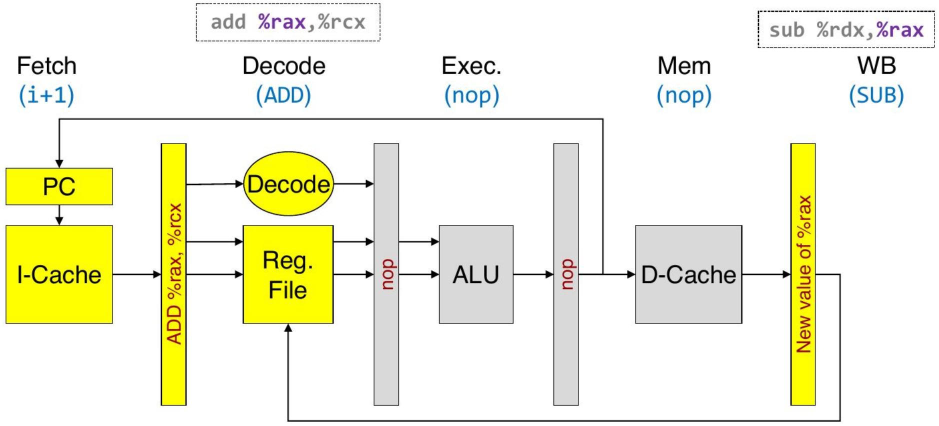
valA ← M_valE = 10
valB ← e_valE = 3

Example from class

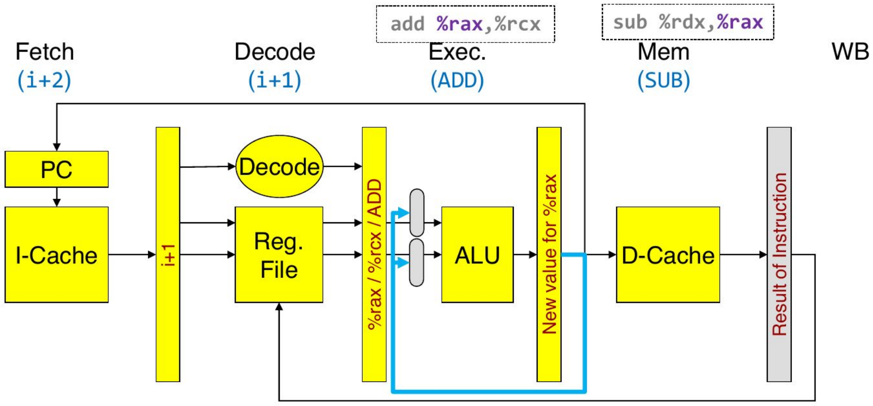




Stalling



Forwarding

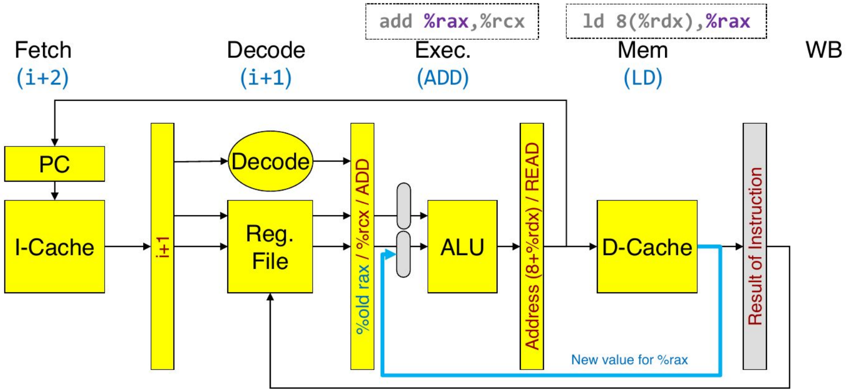


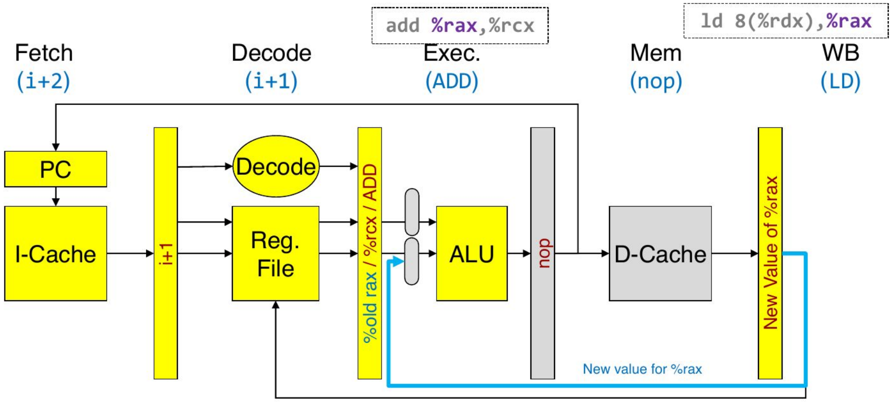
Structural Hazard: Load for next instruction

```
ld 8(%rdx), %rax
add %rax, %rcx
```

While `ld` is saving `%rdx` into a register (phase M), `add` is already using its input to compute a result in phase E.

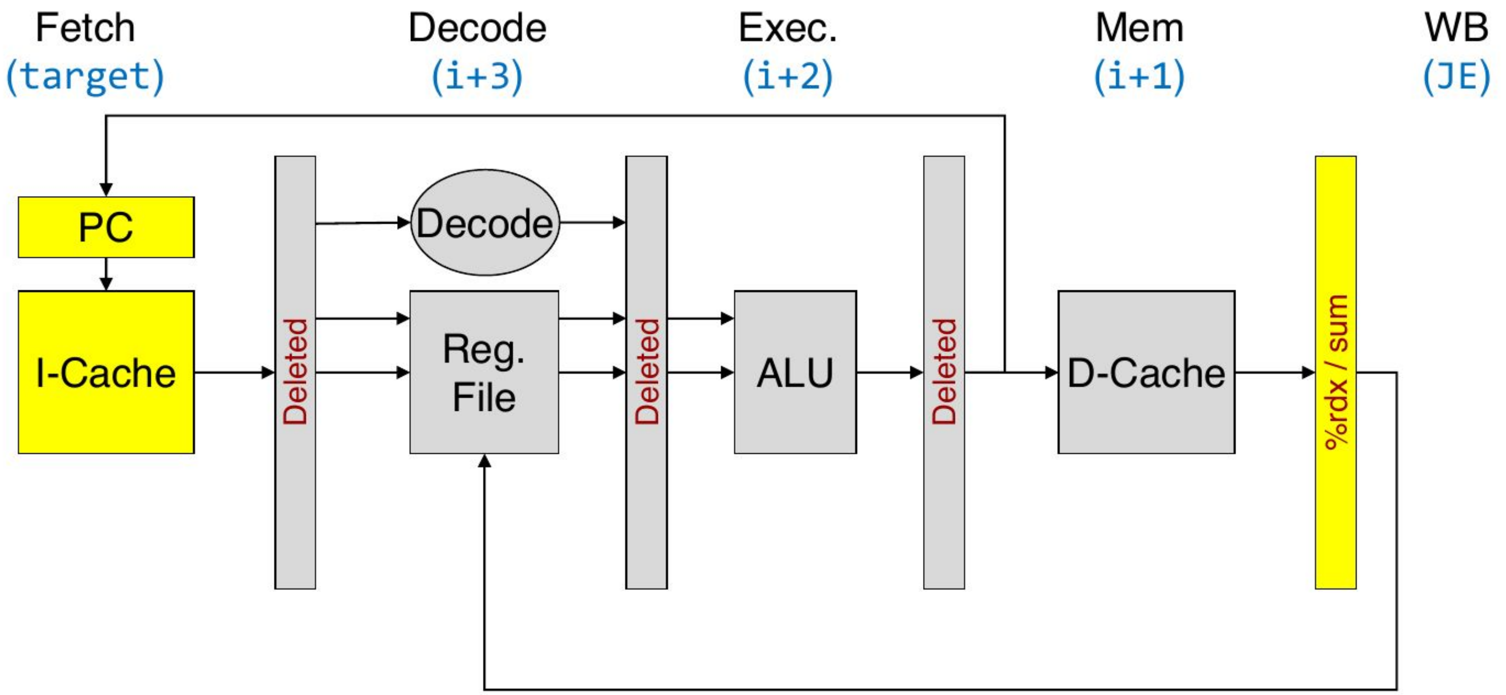
- Forwarding is not enough! We need the output of D-Cache, not the input...
- Use **stalling and forwarding together**.
  - `add` is stalled by 1 phase
  - `ld` passes back the new value of `%rdx` during phase WB





Control Hazard

When a branch is mispredicted, the pipeline (and its effects) must be flushed.



Fetch next instruc i+3

Delete i+3

Delete i+2

Delete i+1

Do nothing

Need to "flush" wrongly fetched instructions

Code Reordering

```c
void increment(int *a, int n, int x) {
    for (int i = 0; i < n; i++) {
        a[i] += x;
    }
}
```

```
increment:
        mov     $0, %ecx   // i
.L1:
        cmp     %esi, %ecx
        jge     .L2
        ld      0(%rdi), %eax
        // nop added here
        add     %edx, %eax
        st      %eax, 0(%rdi)
        add     $4, %rdi
        add     $1, %ecx
        j       .L1
.L2:
        ret
```

```
increment:
        mov     $0, %ecx   // i
.L1:
        cmp     %esi, %ecx
        jge     .L2
        ld      0(%rdi), %eax
        add     $1, %ecx
        add     %edx, %eax
        st      %eax, 0(%rdi)
        add     $4, %rdi

        j       .L1
.L2:
        ret
```

Instead of stalling after the “load for next instruction,” we can move up the counter increment (since it doesn’t affect other instruction until the jump to L1).

Similarly, branch delay slots: move always-executed instructions after the jump.

Superscalar Execution

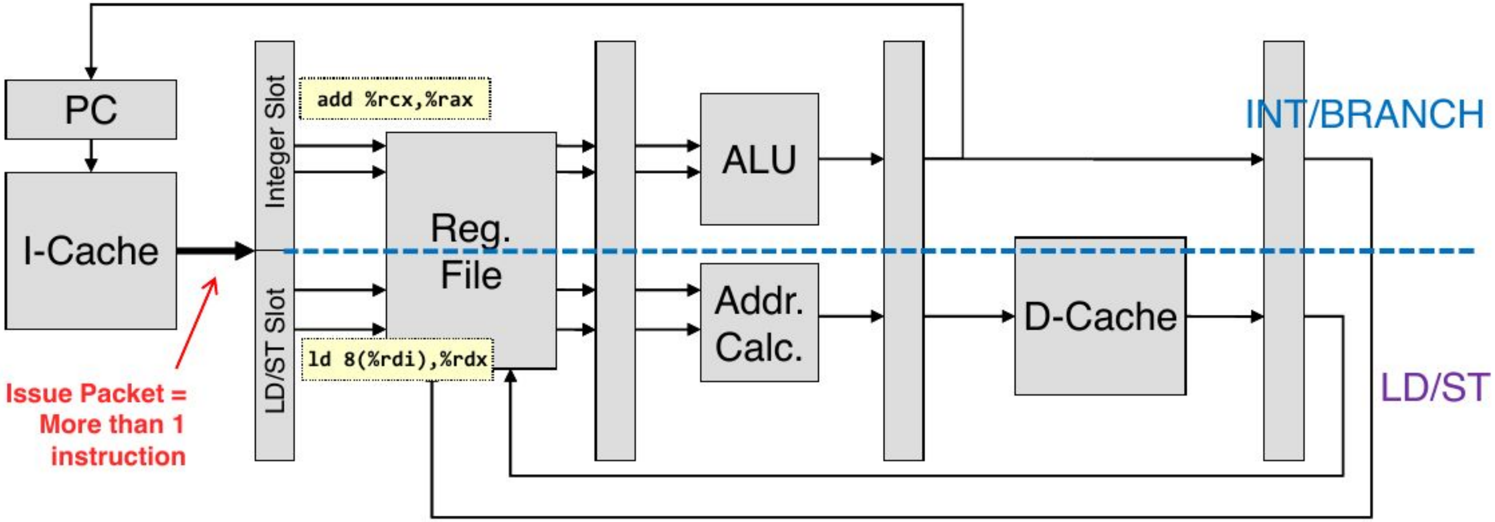
With a pipeline, the throughput is at most 1 / (clock cycle). Can we do better?

- Idea: use instruction-level parallelism.
- Multiple pipelines, each running different instructions in parallel.
- Problems:
    - Data dependencies, or RAW (read-after-write) hazards.
    - Control hazards (jumps).

Approaches

- Static scheduling: compiler packs instructions to be executed in parallel.
- Dynamic scheduling: hardware assigns instructions to parallel queues.

2-way Very Large Instruction Word Machine



- No forwarding between instructions of an “issue packet”
- Full forwarding to instructions behind in the pipeline
- Stall 1 cycle at “load for next instruction”

2-way VLIW Machine: Scheduling Example

```c
void incr5(int *a, int n) {
    for (; n != 0; n--, a++)
        *a += 5;
}
```

```
incr5:
.L1:
      ld    0(%rdi), %r9
      // nop required here
      add   $5, %r9
      st    %r9, 0(%rdi)
      add   $4, %rdi
      add   $-1, %esi
      jne   $0, %esi, .L1
```

Unoptimized Schedule (no gain wrt single pipeline)

```
=== INTEGER SLOT ===          ===  LD/ST SLOT  ===
                              ld 0(%rdi), %r9
add $-1, %esi
add $5, %r9
                              st %r9, 0(%rdi)
add $4, %rdi
jne $0, %esi, .L1
```

Optimized Schedule (move up increase of `si`/`di`)

```
=== INTEGER SLOT ===          ===  LD/ST SLOT  ===
add $-1, %esi                 ld 0(%rdi), %r9
add $4, %rdi
add $5, %r9
jne $0, %esi, .L1             st %r9, -4(%rdi)
```

From 6/6 = 1 instructions per cycle to 6/4 = 1.5

Loop Unrolling

Sometimes we don’t have enough instruction for parallel pipelines.

Idea: copy body *k* times and iterate only *n*/*k* times (assume *n* multiple of *k*)

- Different copies of body can run in parallel.

```
void incr5(int *a, int n) {
    for (; n != 0; n-= 4, a+=4) {
        *a += 5;
        *(a+1) += 5;
        *(a+2) += 5;
        *(a+3) += 5;
    }
}
```

Still can’t run in parallel: all copies use the register **%r9**
⇒ **Read-After-Write (RAW)**
⇒ **Register renaming**

```
incr5:
.L1:
0     ld     0(%rdi), %r9
0     add    $5, %r9
0     st     %r9, 0(%rdi)
1     ld     4(%rdi), %r9
1     add    $5, %r9
1     st     %r9, 4(%rdi)
2     ld     8(%rdi), %r9
2     add    $5, %r9
2     st     %r9, 8(%rdi)
3     ld     12(%rdi), %r9
3     add    $5, %r9
3     st     %r9, 12(%rdi)
      add    $16, %rdi
      add    $-4, %esi
      jne    $0, %esi, .L1
```

```
old-incr5:
.L1:
0     ld     0(%rdi), %r9
0     add    $5, %r9
0     st     %r9, 0(%rdi)
      add    $4, %rdi
      add    $-1, %esi
      jne    $0, %esi, .L1
```

Loop Unrolling and Register Renaming

```
incr5:
.L1:
0     ld     0(%rdi), %r9
0     add    $5, %r9
0     st     %r9, 0(%rdi)
1     ld     4(%rdi), %r10
1     add    $5, %r10
1     st     %r10, 4(%rdi)
2     ld     8(%rdi), %r11
2     add    $5, %r11
2     st     %r11, 8(%rdi)
3     ld     12(%rdi), %r12
3     add    $5, %r12
3     st     %r12, 12(%rdi)
      add    $16, %rdi
      add    $-4, %esi
      jne    $0, %esi, .L1
```

Optimized Schedule

```
=== INTEGER SLOT ===          ===  LD/ST SLOT  ===
                              ld 0(%rdi), %r9
add $-4, %esi                 ld 4(%rdi), %r10
add $5, %r9                   ld 8(%rdi), %r11
add $5, %r10                  ld 12(%rdi), %r12
add $5, %r11                  st %r9, 0(%rdi)
add $5, %r12                  st %r10, 4(%rdi)
add $16, %rdi                 st %r11, 8(%rdi)
jne $0, %esi, .L1             st %r12, -4(%rdi)
```

IPC = 15/8

Notice: We exploit independence of loop bodies.