pollev.com/kakiryan

COMP311: *COMPUTER ORGANIZATION!*

Lecture 18: Pipelining pt 2, Intro to Assembly

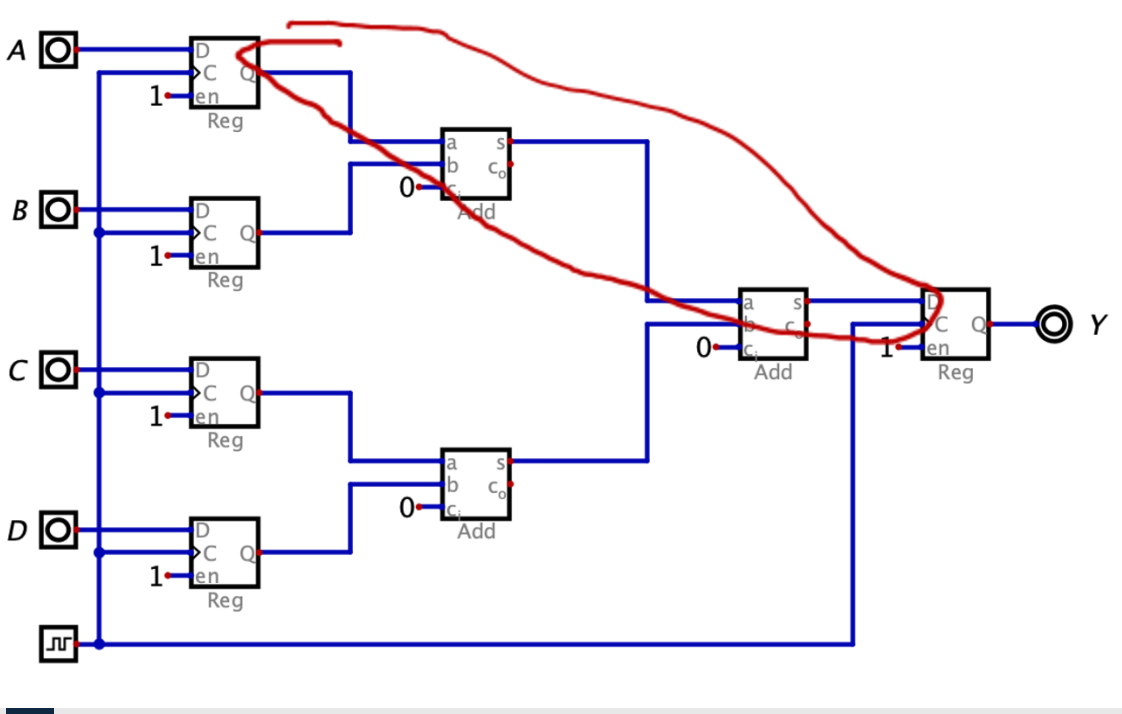
tinyurl.com/comp311-fa25

PIPELINING

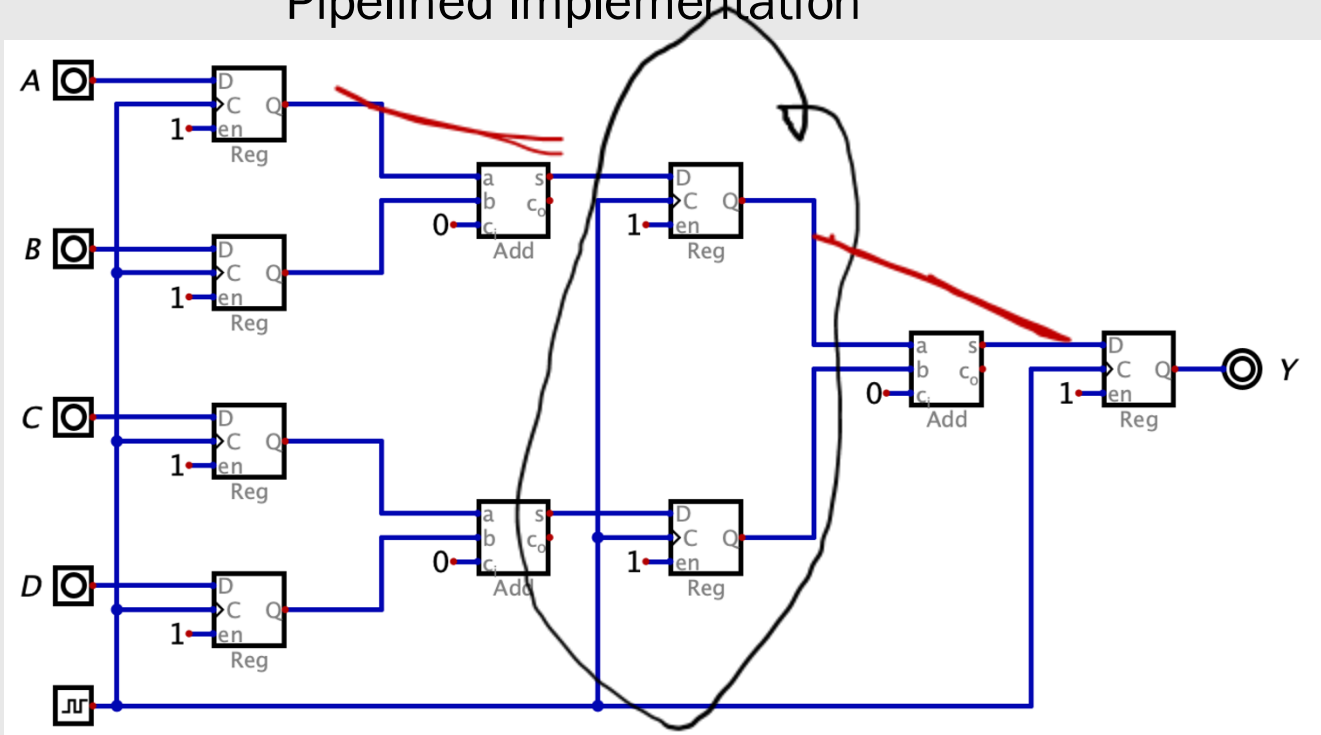
Single Cycle vs Pipelined

Clock-to-q delay = 20ps
Adder propagation delay = 480ps

Single-Cycle Implementation



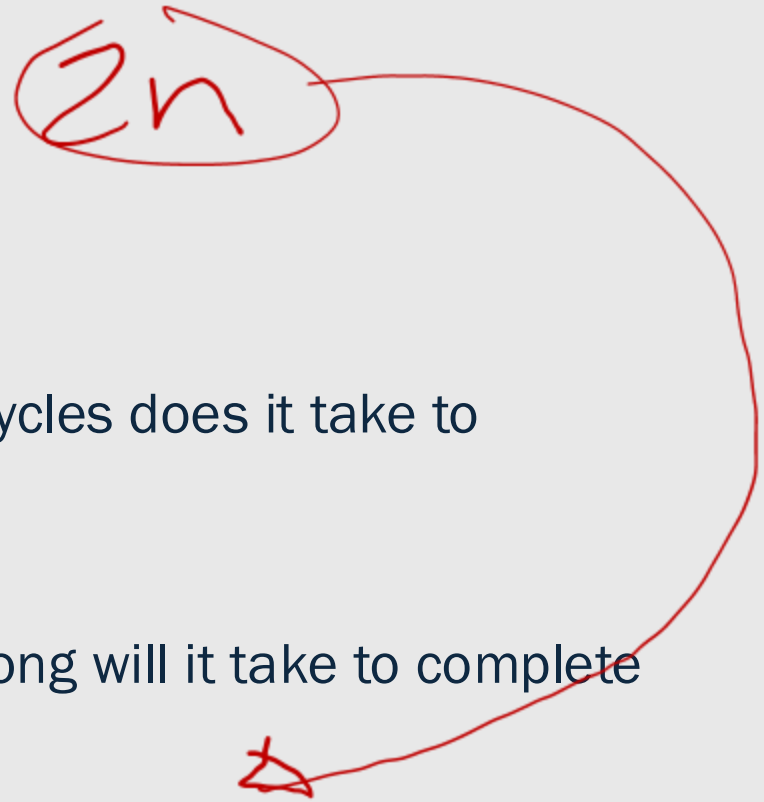
Pipelined Implementation



	Single-Cycle	Pipelined
Longest combinational path	960ps	480ps
Min clock period	980ps	500ps
# cycles to complete one operation	1	2
Time to complete 10 ops at min clock speed	9800ps	5500ps

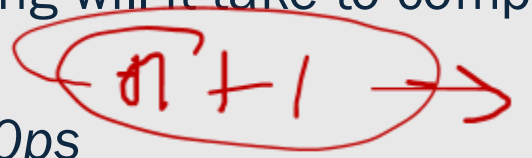
Single Cycle

1. What is the longest combinational path in this circuit?
   - *The delay through the two adders*
   - *480ps + 480ps = 960ps*
2. What is the min clock period?
   - *Clock-to-q delay + longest combinational delay*
   - *20ps + 960ps = 980ps*
3. If we set the clock to the min period, how many clock cycles does it take to perform one operation?
   - *1 cycle*
4. If we run this circuit at the minimum clock period, how long will it take to complete 10 operations?
   - *Final result is available at t = (980ps)(10) = 9800ps*
   - *Technically, the result is not available on Y until 9800ps + 20ps = 9820ps, but when we are talking about metrics, we keep it simple and just say 9800ps*



Pipelined

1. What is the longest combinational path in this circuit?
   - *The longest combinational delay between any two registers is through one adder*
   - *480ps*
2. What is the min clock period?
   - *Clock-to-q delay + longest combinational delay*
   - *20ps + 480 = 500ps*
3. If we set the clock to the min period, how many clock cycles does it take to perform one operation?
   - *2*
4. If we run this circuit at the minimum clock period, how long will it take to complete 10 operations?
   - *Final result is available at t = (500ps)(10+1) = 5500ps*
   - *Technically, the result is not available on Y until 5500ps + 20ps = 5520ps, but when we are talking about metrics, we keep it simple and just say 5500ps*



Single-Cycle vs Pipelined Execution Times

- To perform n operations
  - *Single-cycle: (n)(clock period)*
  - *Pipelined (n + 1)(clock period)*

(10 + 1)

but takes 20 cycles!

Metrics

- Latency
  - *The amount of time that it takes to complete a single operation*
- Throughput
  - *The number of operations that can be completed in a given amount of time*
- Speedup
  - *The measure of how much faster the pipelined implementation is in comparison to the single-cycle implementation.*

$$speedup = \frac{Time\ it\ takes\ to\ complete\ a\ given\ set\ of\ operations\ on\ the\ single\ cycle\ implementation}{Time\ it\ takes\ to\ complete\ the\ same\ set\ of\ operations\ on\ the\ pipelined\ implementation}$$

Performance Metrics

- What is the latency of an operation in the single-cycle implementation?
  - *One clock cycle = 980ps*
- What is the latency of an operation in the pipelined implementation?
  - *Two clock cycles = 2(500ps) = 1000ps*
- Is the throughput higher in the single cycle example or the pipelined example?
  - *Pipelined*
- Speedup
  - *9800ps/5500ps = 1.78*
- If we had performed 20 operations, would the speedup be higher?
  - *Yes*

Performance Metrics

- What is the latency of an operation in the single-cycle implementation?
  - *One clock cycle = 980ps*
- Wh
- Is
  ex
- Sp
- If

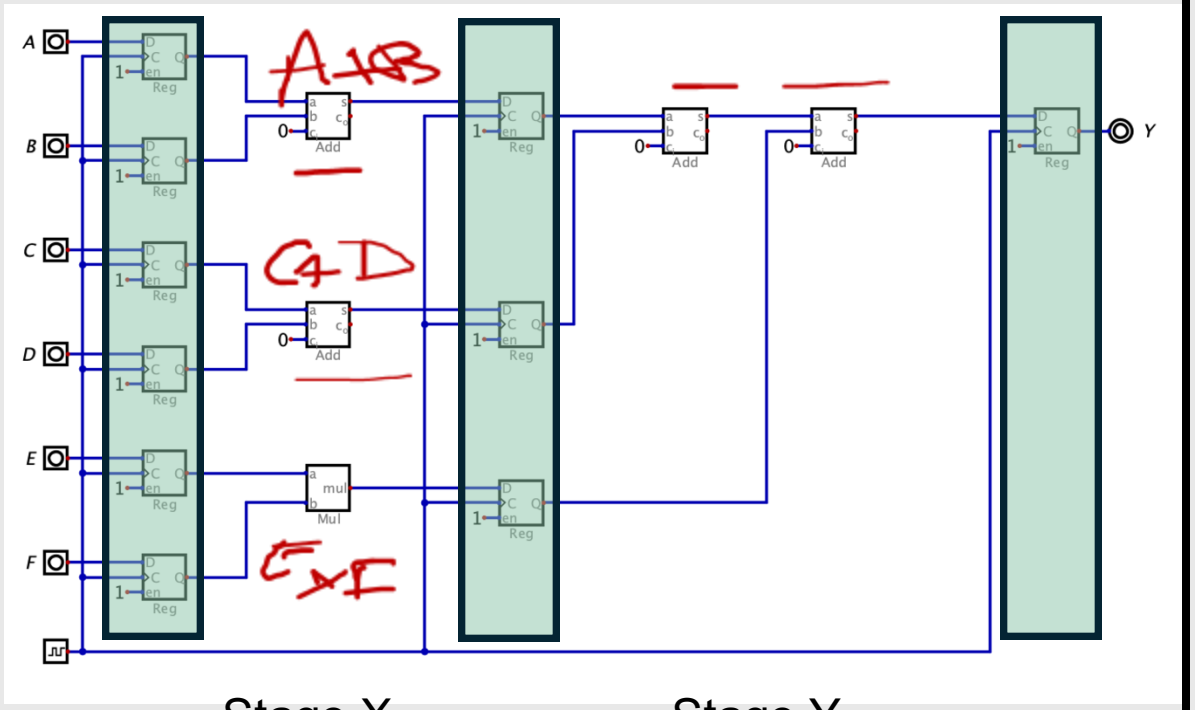
With a pipelined operation, the latency of a single operation may be higher since we have broken things into stages

Why? We have to wait for extra clk-to-q delays for things to pass through the pipeline registers!

But! overtime we get higher throughput once we are to overlap operations and keep the CPU/hardware busy. ☺

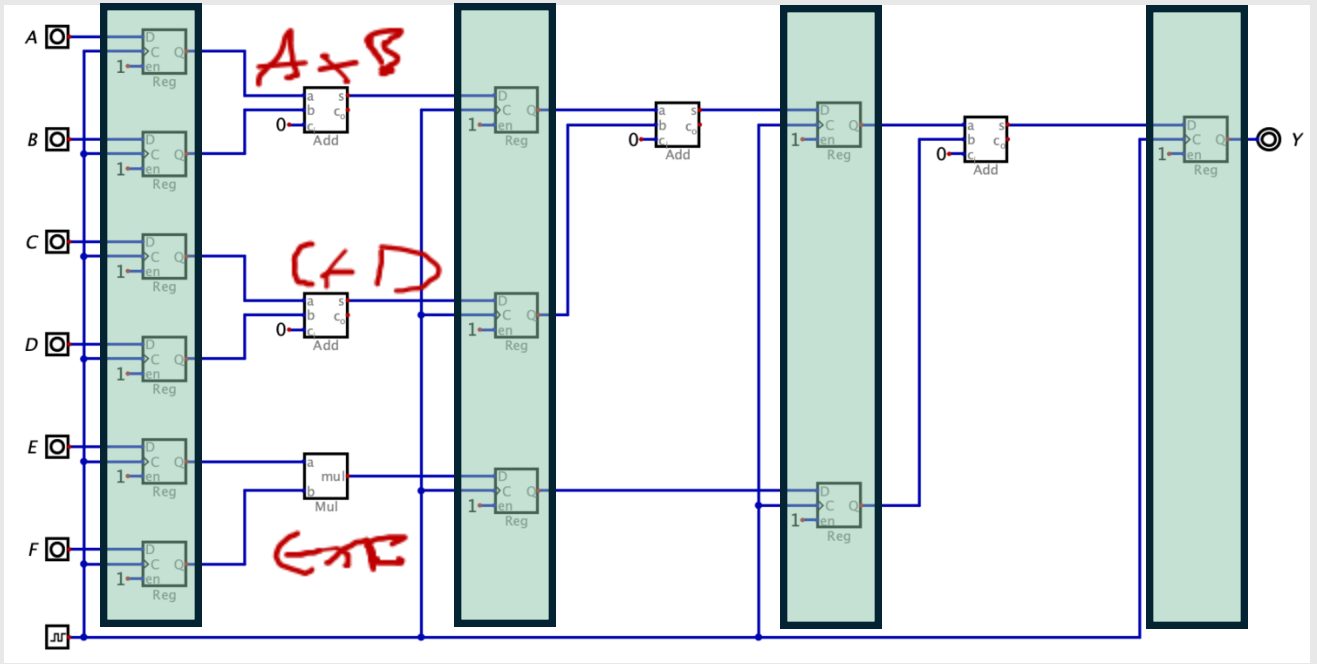
Y = A + B + C + D + EF

- Let's compare two circuits that perform the operation Y = A + B + C + D + EF



Stage X Stage Y

2-stage Pipeline



Stage X Stage Y Stage Z

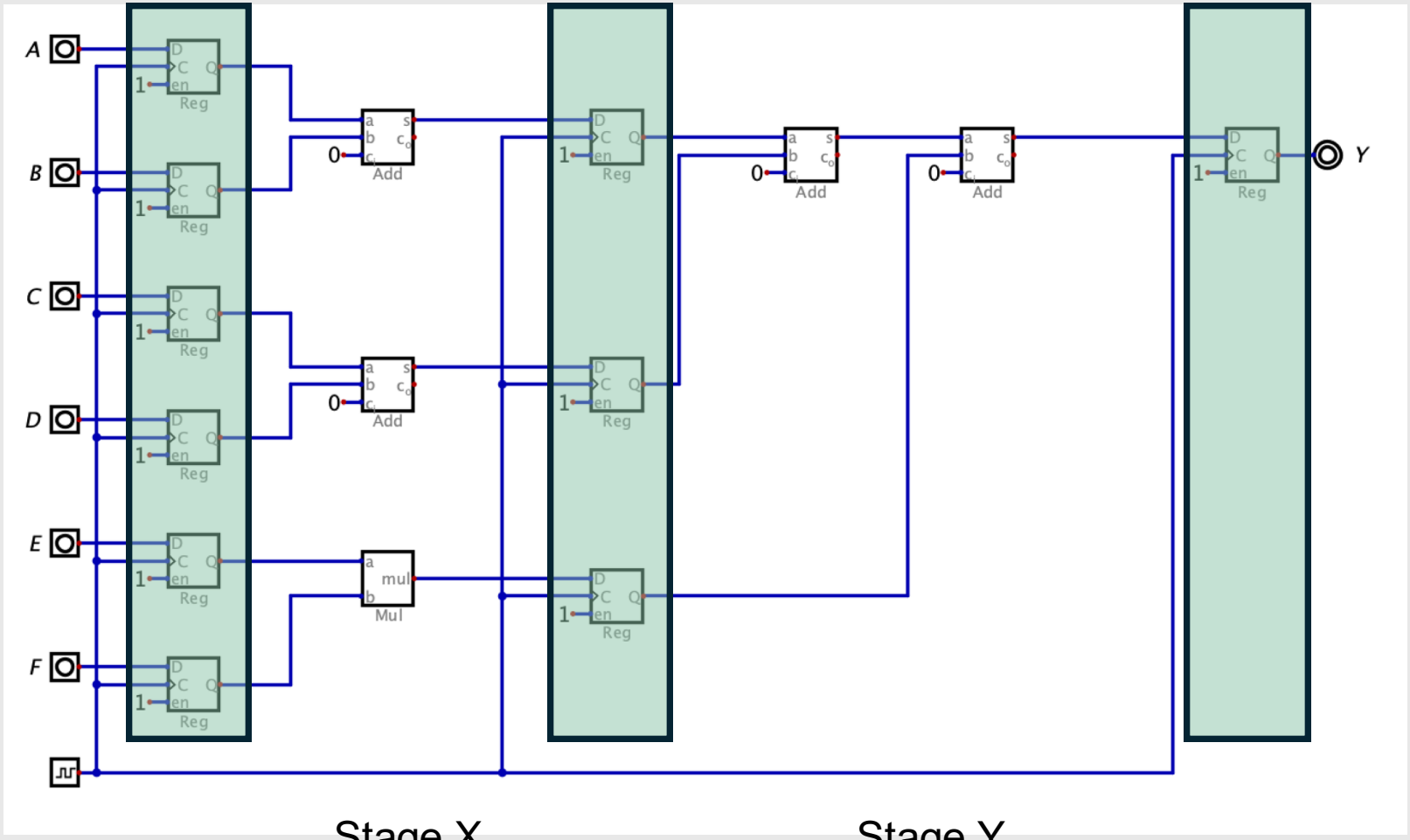
3-stage Pipeline

2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we want to perform the following operation:
Y = 1 + 2 + 3 + 4 + (5)(6)

- In Cycle 0, we will compute

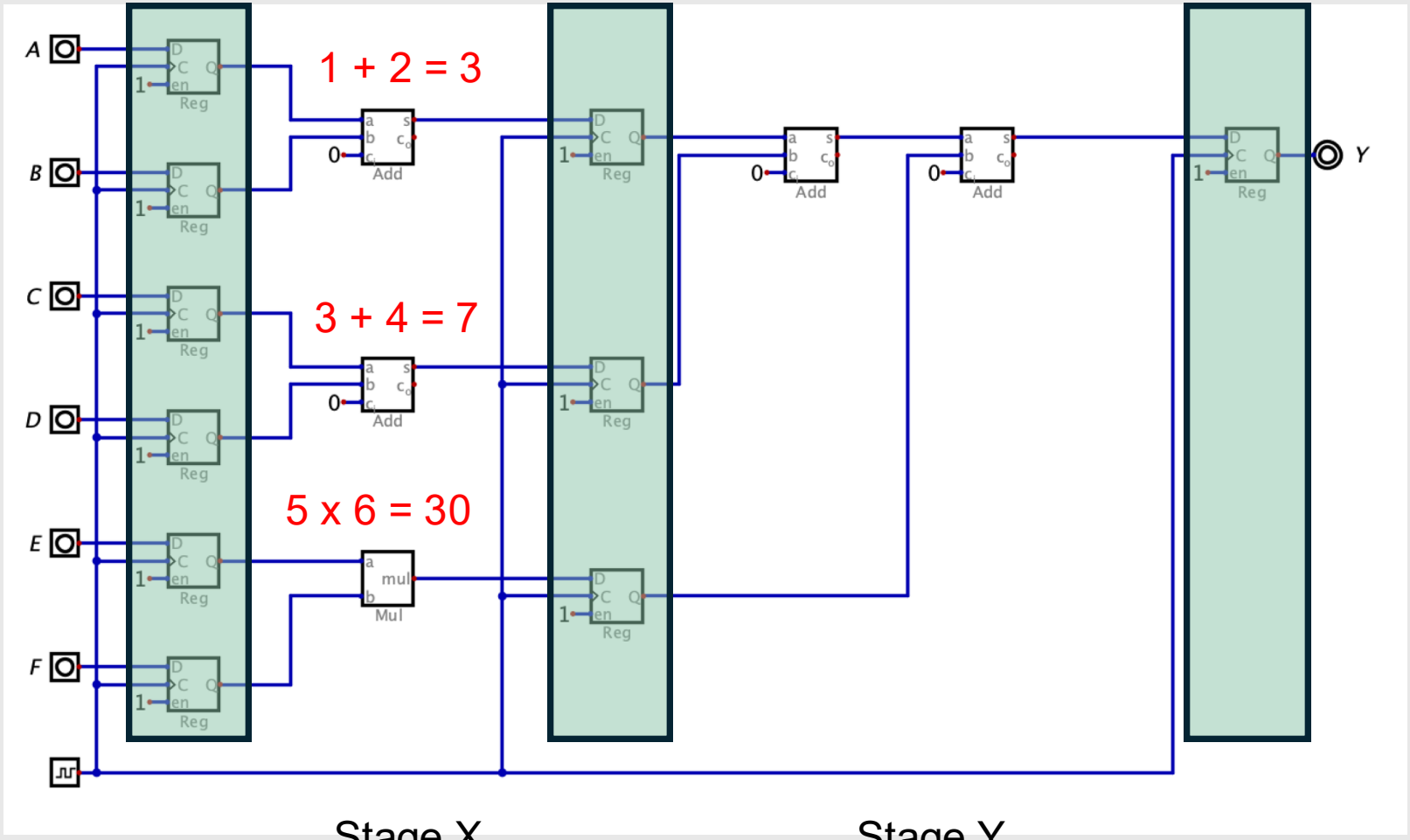


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we want to perform the following operation:
Y = 1 + 2 + 3 + 4 + (5)(6)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30

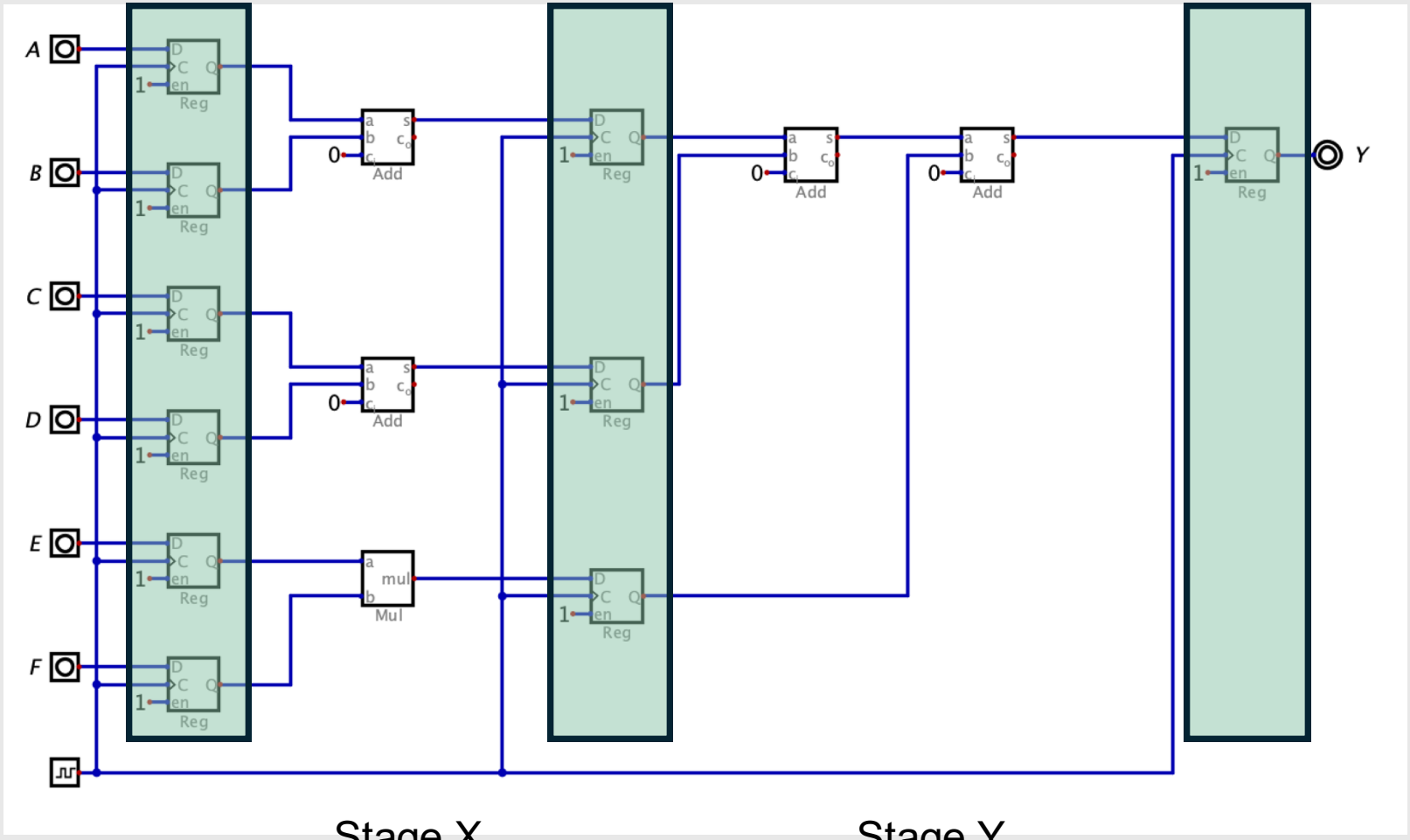


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we want to perform the following operation:
Y = 1 + 2 + 3 + 4 + (5)(6)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30
- In Cycle 1, we will compute

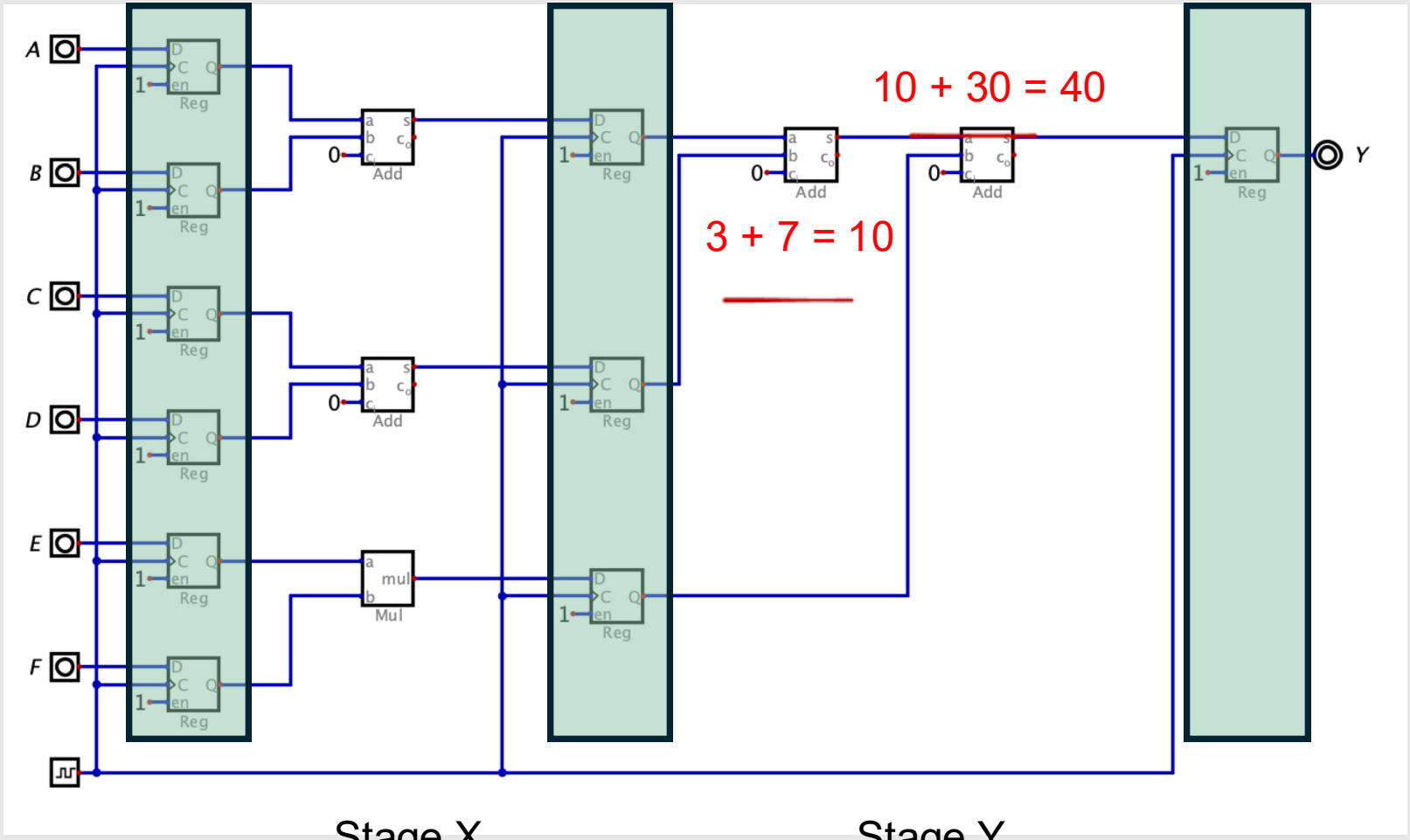


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we want to perform the following operation:
Y = 1 + 2 + 3 + 4 + (5)(6)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30
- In Cycle 1, we will compute
  - 3 + 7 + 30 = 40

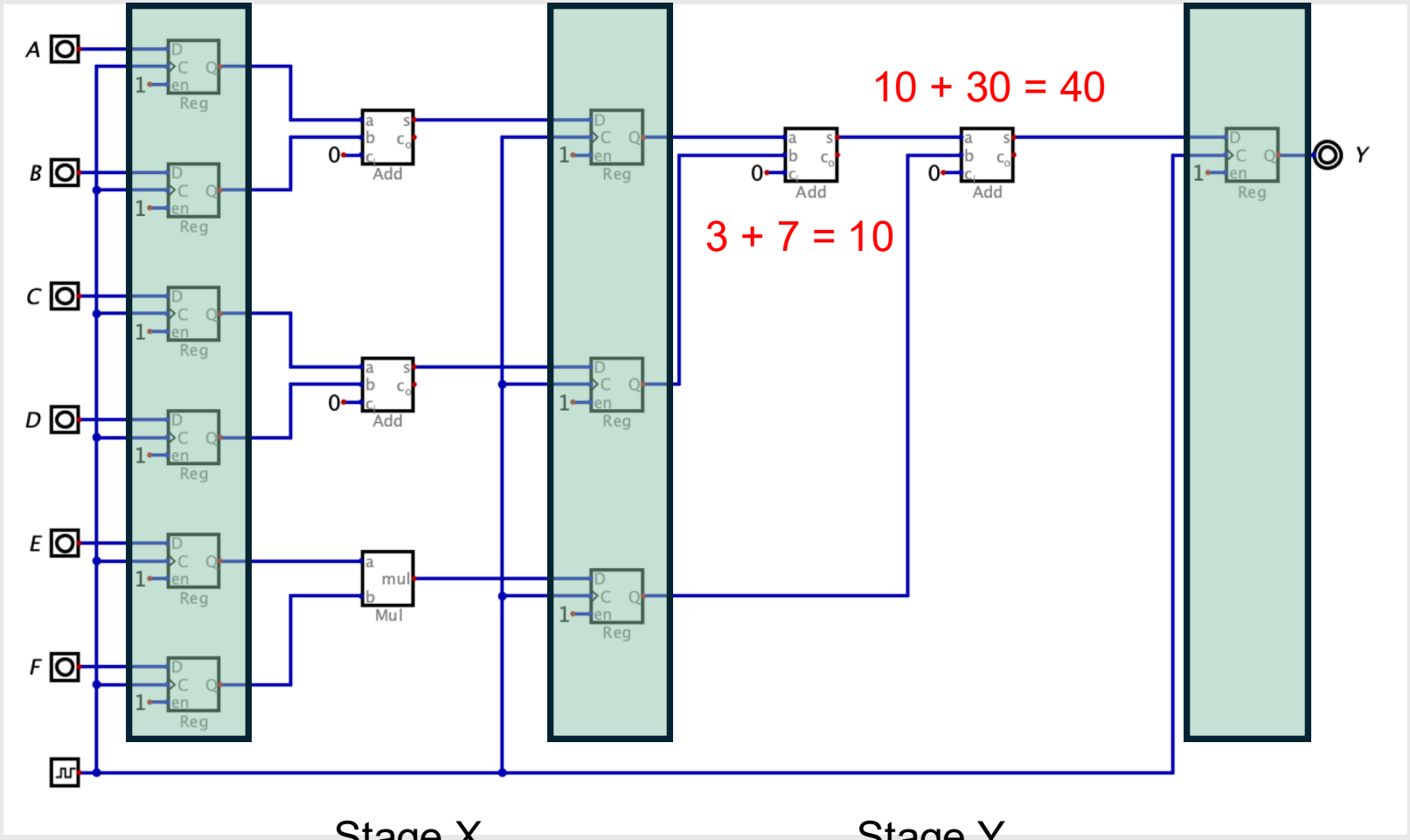


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

In Cycle 1, Stage X is empty. We can make better use of our resources and start another operation while the first operation is in Stage Y

Let's say we also want to compute
Y = 4 + 2 + 7 + 10 + (9)(8)

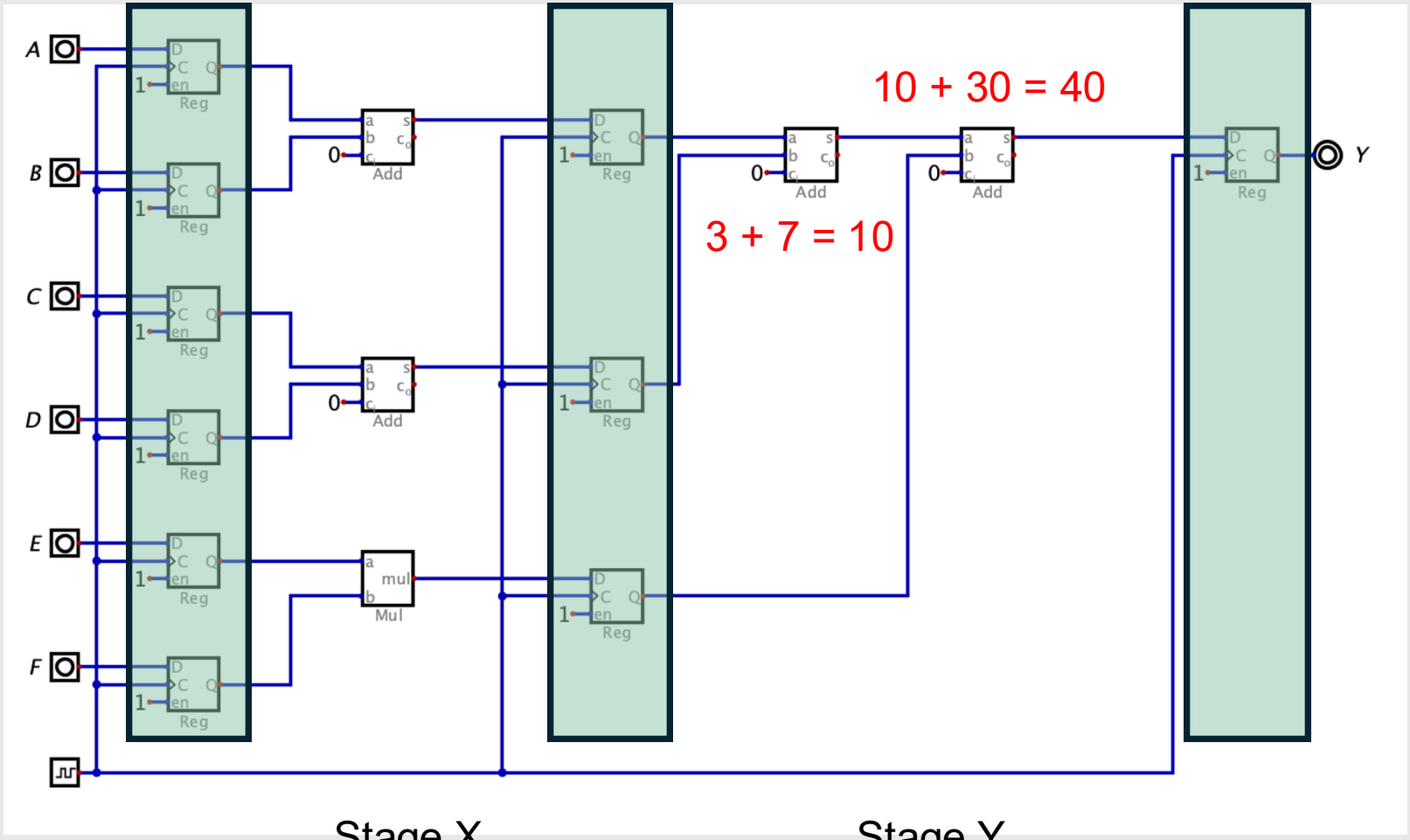


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we also want to perform the following operation:
Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 1, we will compute

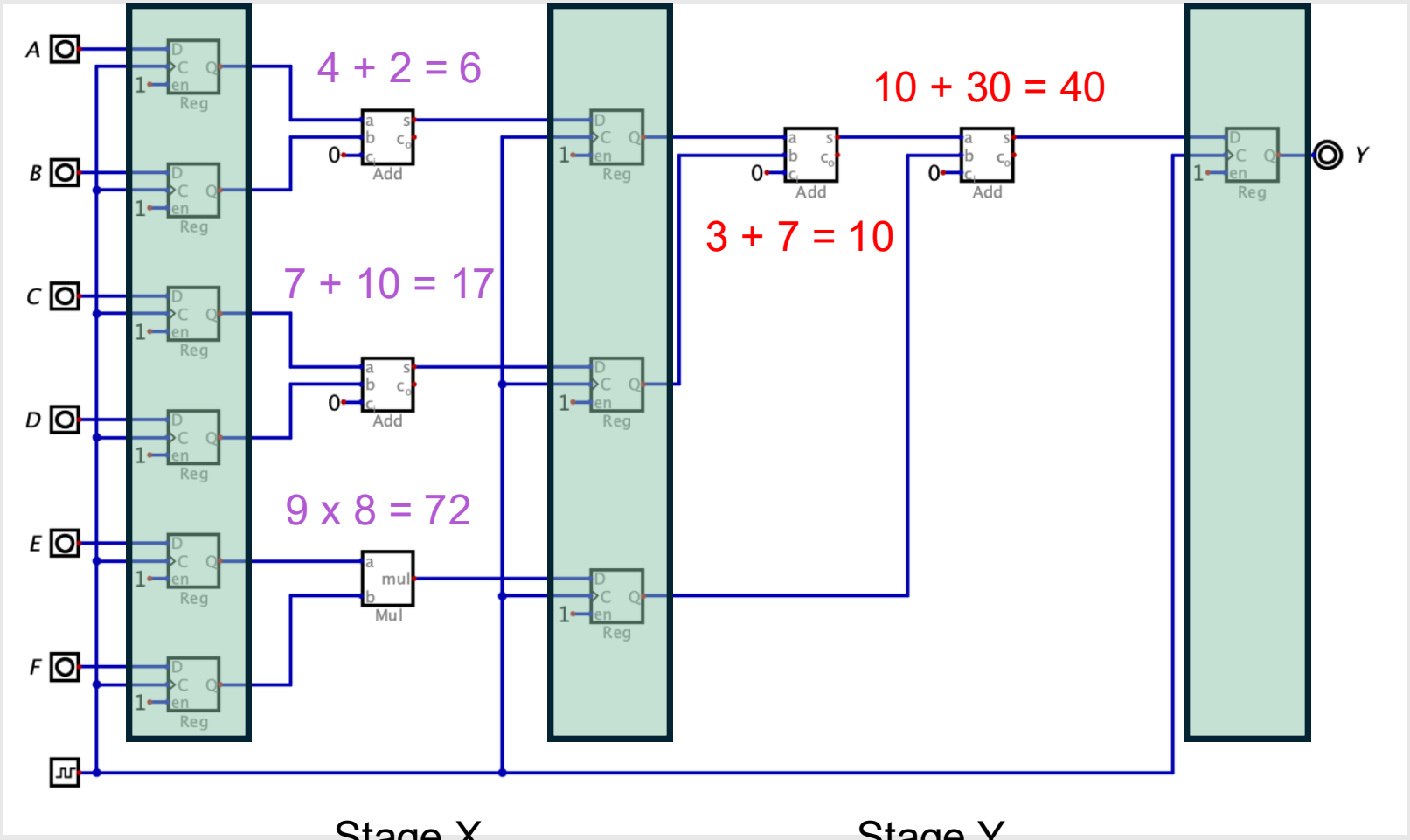


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we also want to perform the following operation:
Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 1, we will compute
  - 4 + 2 = 6
  - 7 + 10 = 17
  - 9 x 8 = 72

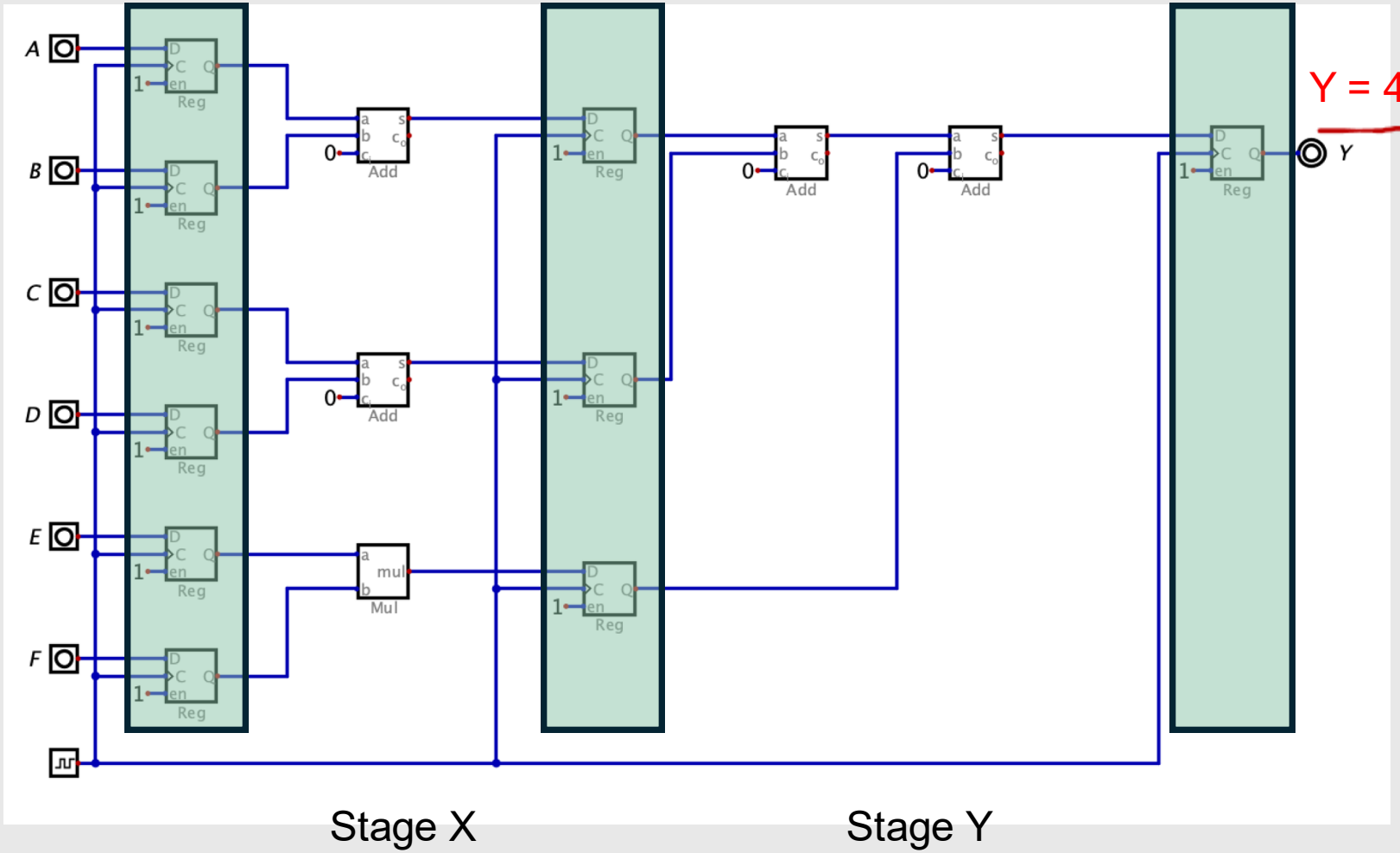


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we also want to perform the following operation:
Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 1, we will compute
  - 4 + 2 = 6
  - 7 + 10 = 17
  - 9 x 8 = 72
- In Cycle 2, we will compute

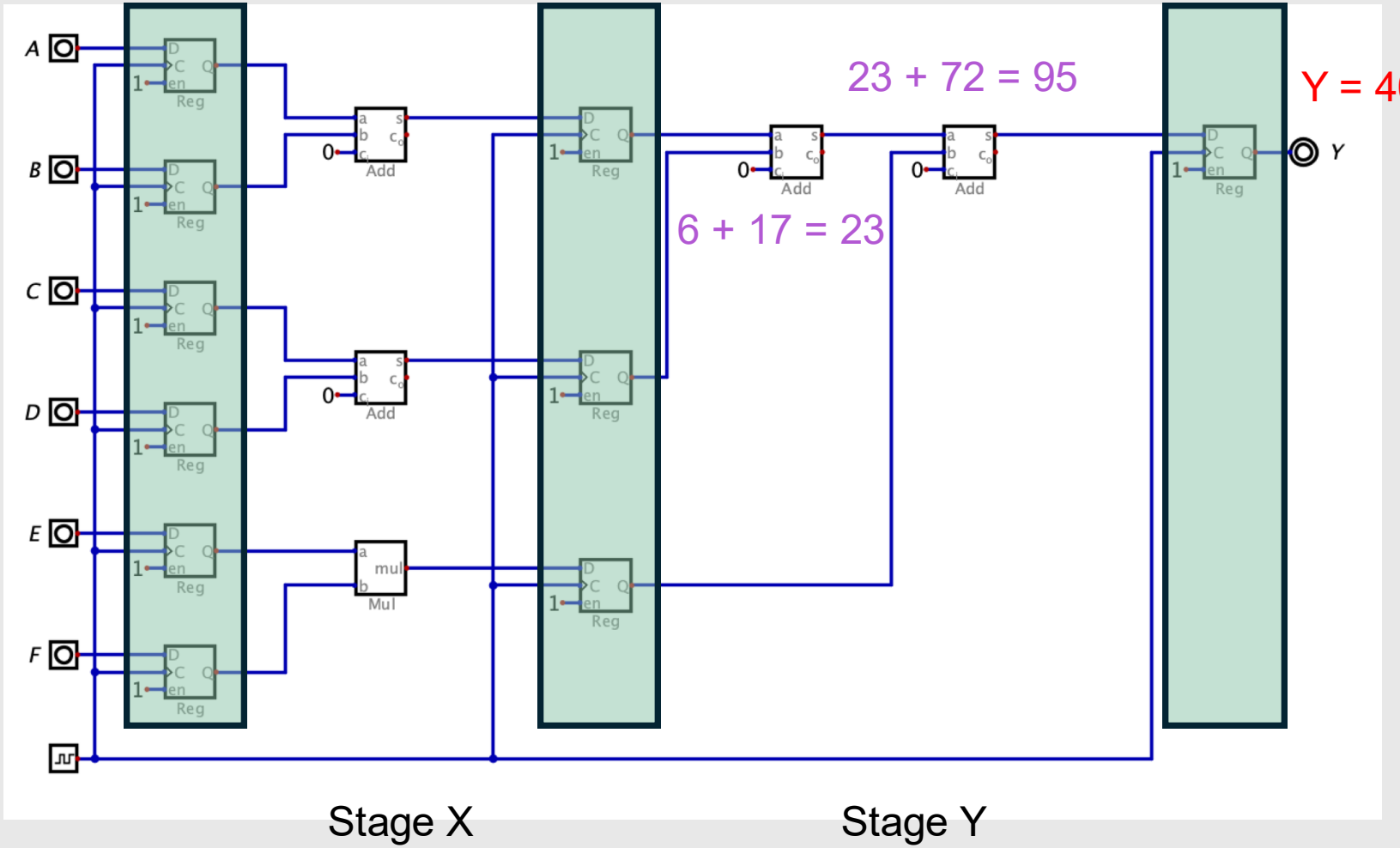


2-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's say we also want to perform the following operation:
Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 1, we will compute
  - 4 + 2 = 6
  - 7 + 10 = 17
  - 9 x 8 = 72
- In Cycle 2, we will compute
  - 6 + 17 + 72 = 95

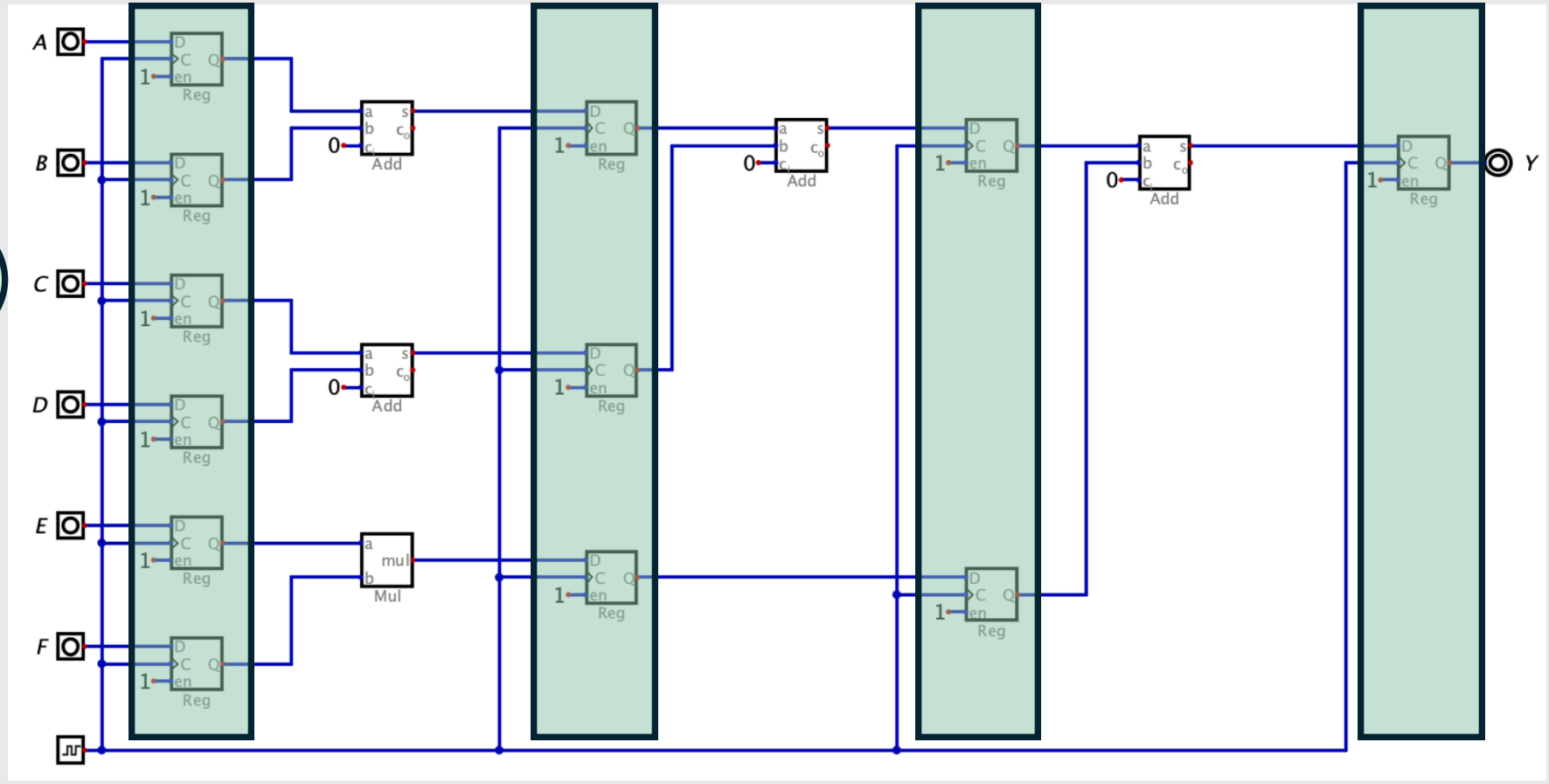


3-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Let's look at the same set of operations in the 3-state pipeline

$Y = 1 + 2 + 3 + 4 + (5)(6)$
$Y = 4 + 2 + 7 + 10 + (9)(8)$



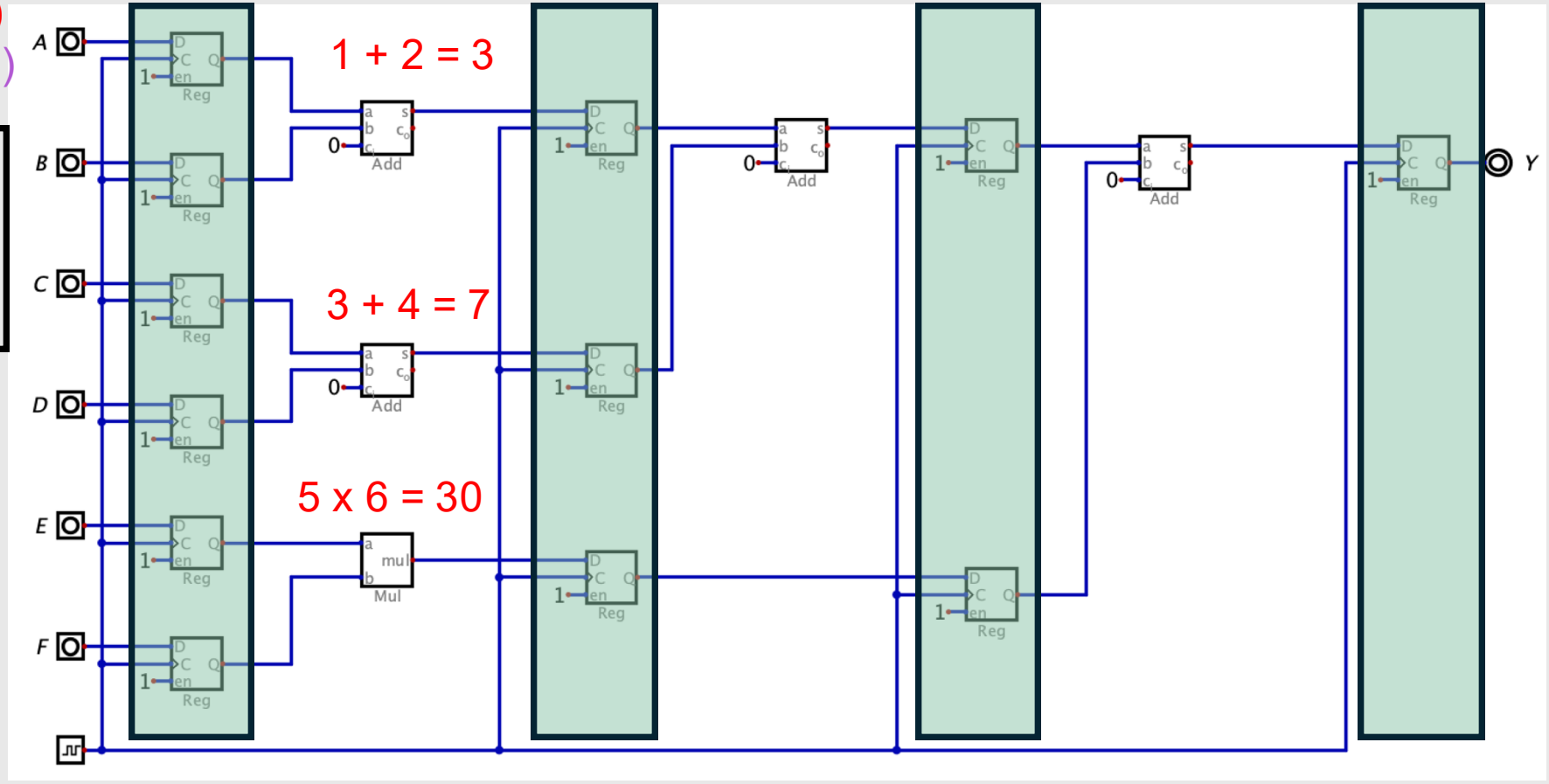
Stage X Stage Y Stage Z

3-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Op0: Y = 1 + 2 + 3 + 4 + (5)(6)
Op1: Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30

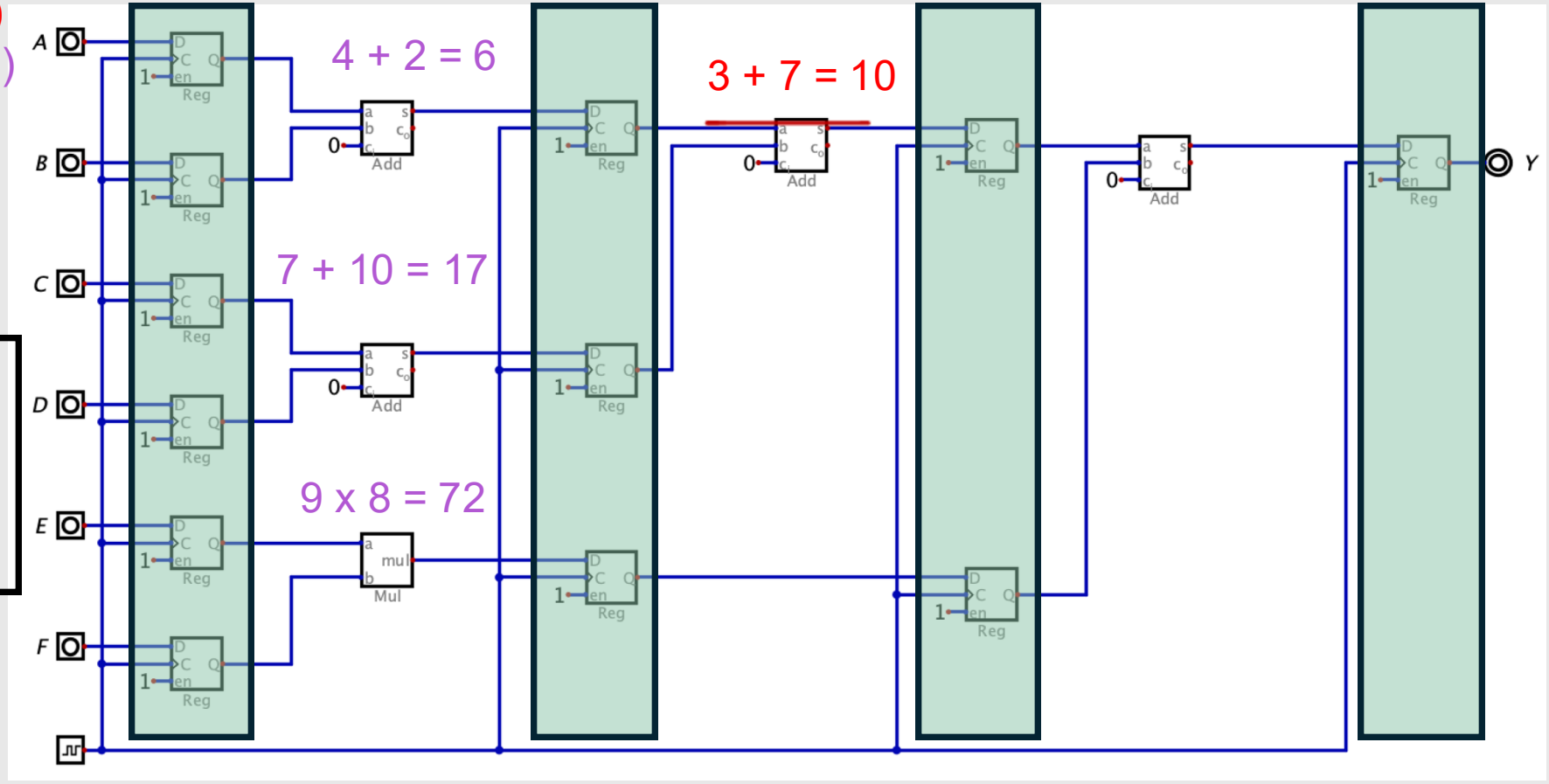


3-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Op0: Y = 1 + 2 + 3 + 4 + (5)(6)
Op1: Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30
- In Cycle 1, we will compute
  - 3 + 7 = 10
  - 4 + 2 = 6
  - 7 + 10 = 17
  - 9 x 8 = 72

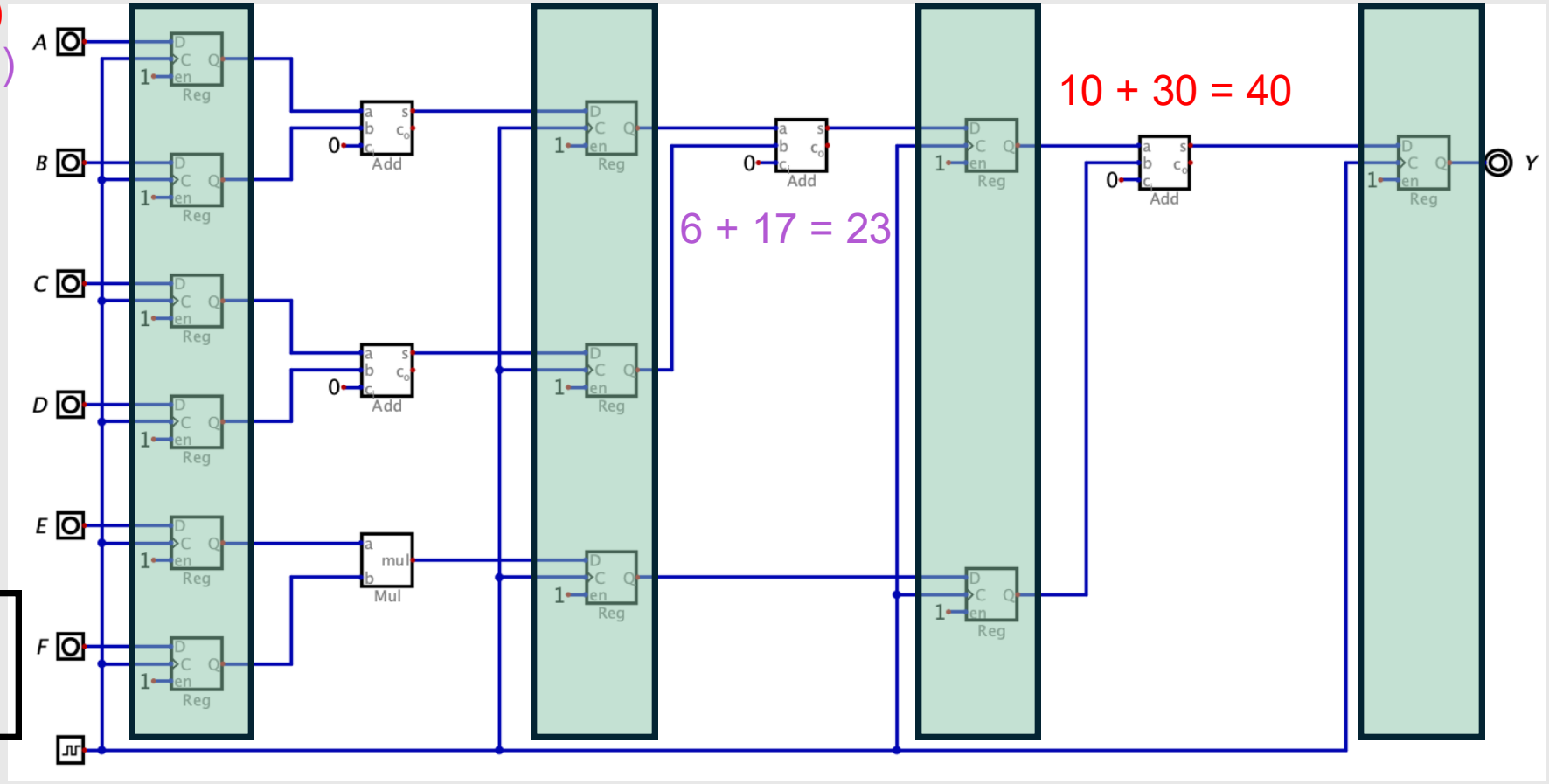


3-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Op0: Y = 1 + 2 + 3 + 4 + (5)(6)
Op1: Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30
- In Cycle 1, we will compute
  - 3 + 7 = 10
  - 4 + 2 = 6
  - 7 + 10 = 17
  - 9 x 8 = 72
- In Cycle 2, we will compute
  - 10 + 30 = 40
  - 6 + 17 = 23

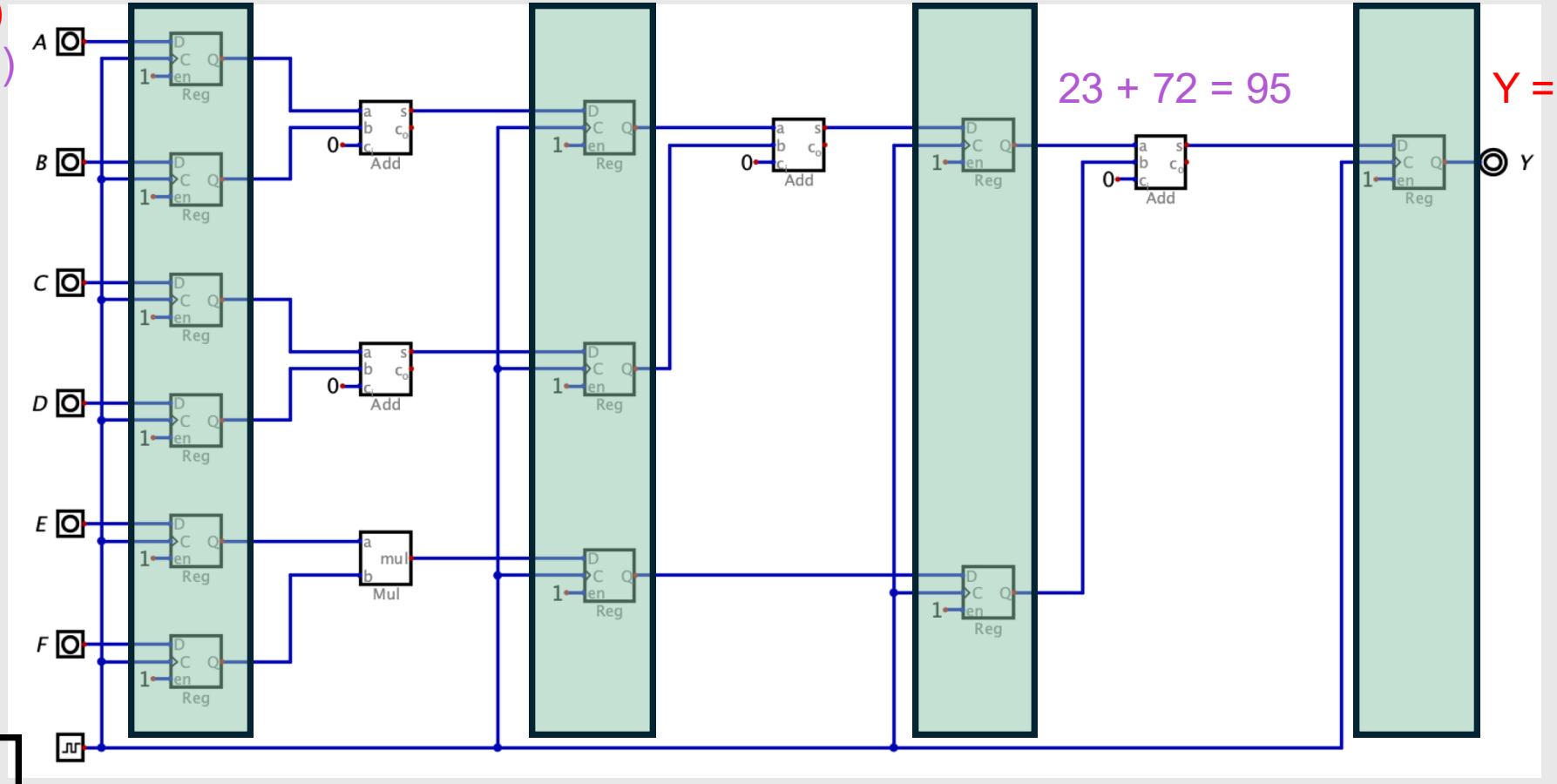


3-Stage Pipeline

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Op0: Y = 1 + 2 + 3 + 4 + (5)(6)
Op1: Y = 4 + 2 + 7 + 10 + (9)(8)

- In Cycle 0, we will compute
  - 1 + 2 = 3
  - 3 + 4 = 7
  - 5 x 6 = 30
- In Cycle 1, we will compute
  - 3 + 7 = 10
  - 4 + 2 = 6
  - 7 + 10 = 17
  - 9 x 8 = 72
- In Cycle 2, we will compute
  - 10 + 30 = 40
  - 6 + 17 = 23
- In Cycle 3, we will compute
  - 23 + 72 = 95



2-stage Pipeline Diagram

Example Operations

Operation 0: 1 + 2 + 3 + 4 + (5)(6)
Operation 1: 4 + 2 + 7 + 10 + (9)(8)

Cycle #	0	1	2	3	4	5	6	7	8	9	10	11	12
Op0	X	Y											
Op1		X	Y										
Op2			X	Y									
Op3				X	Y								
Op4													
Op5													
Op6													
Op7													
Op8													
Op9													

2-stage Pipeline Diagram

Example Operations

Operation 0: 1 + 2 + 3 + 4 + (5)(6)

Operation 1: 4 + 2 + 7 + 10 + (9)(8)

X: ty: stages

Cycle #	0	1	2	3	4	5	6	7	8	9	10	11	12
Op0	X	Y											
Op1		X	Y										
Op2			X	Y									
Op3				X	Y								
Op4					X	Y							
Op5						X	Y						
Op6							X	Y					
Op7								X	Y				
Op8									X	Y			
Op9										X	Y		

3-stage Pipeline Diagram

Example Operations

Operation 0: 1 + 2 + 3 + 4 + (5)(6)
Operation 1: 4 + 2 + 7 + 10 + (9)(8)

Cycle #	0	1	2	3	4	5	6	7	8	9	10	11	12
Op0	X	Y	Z										
Op1		X	Y	Z									
Op2													
Op3													
Op4													
Op5													
Op6													
Op7													
Op8													
Op9													

3-stage Pipeline Diagram

Example Operations

Operation 0: 1 + 2 + 3 + 4 + (5)(6)
Operation 1: 4 + 2 + 7 + 10 + (9)(8)

Cycle #

	0	1	2	3	4	5	6	7	8	9	10	11	12
Op0	X	Y	Z										
Op1		X	Y	Z									
Op2			X	Y	Z								
Op3				X	Y	Z							
Op4					X	Y	Z						
Op5						X	Y	Z					
Op6							X	Y	Z				
Op7								X	Y	Z			
Op8									X	Y	Z		
Op9										X	Y	Z	

For each circuit...

Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

1. What is the longest combinational path?

1200 ps

2. What is the min clock period (critical path)?

1220

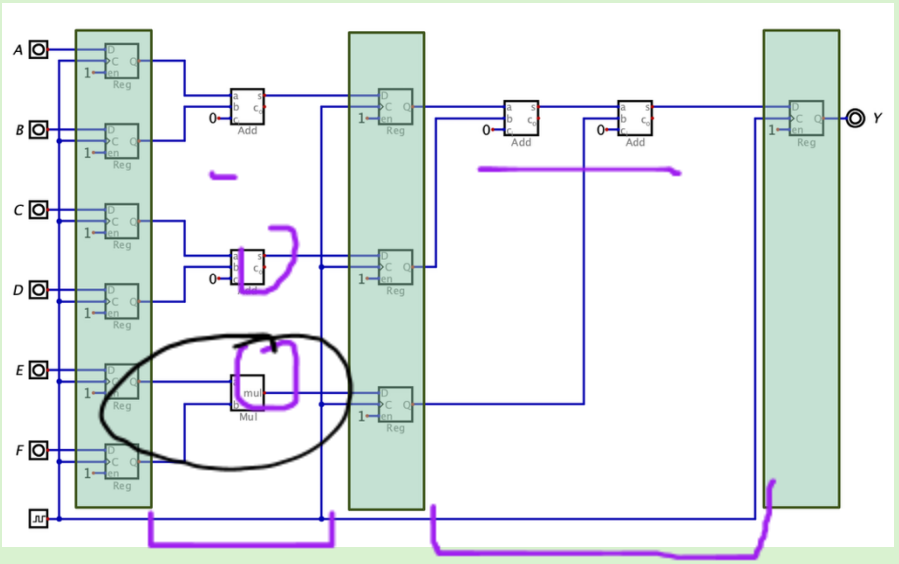
3. If the clock period is at least the min period, how many clock cycles does it take to perform one operation?

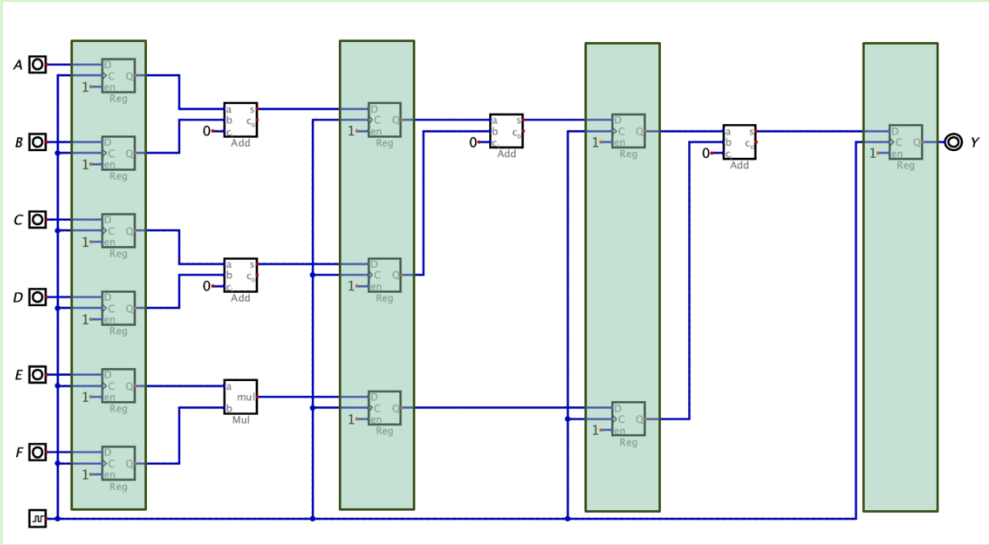
2

4. If we run this circuit at the minimum clock period, how long will it take to complete 10 operations?

(n + 1) (min clk)
PQ

ops extra stages





Two Stage Pipeline

1. What is the longest combinational path?
   - *Multiplier*
   - *1200ps*
2. What is the min clock period?
   - *Clock-to-q delay + longest combinational delay*
   - *20ps + 1200ps = 1220ps*
3. If the clock period is at least the min period, how many clock cycles does it take to perform one operation?
   - *two*
4. If we run this circuit at the minimum clock period, how long will it take to complete 10 operations?
   - *Final result is available at t = (1220ps)(10+1) = 13420ps*

Three Stage Pipeline

1. What is the longest combinational path?
   - *Multiplier*
   - *1200ps*
2. What is the min clock period?
   - *Clock-to-q delay + longest combinational delay*
   - *20ps + 1200ps = 1220ps*
3. If the clock period is at least the min period, how many clock cycles does it take to perform one operation?
   - *three*
4. If we run this circuit at the minimum clock period, how long will it take to complete 10 operations?
   - *Final result is available at t = (1220ps)(10+2) = 14640ps*

Performance Metrics

- What is the latency of an operation in the 2-stage pipeline?
- What is the latency of an operation in the 3-stage pipeline?
- Is the throughput higher in the 2-stage or 3-stage pipeline?
- What is the speedup of completing 10 operations on the 2-stage pipeline vs 2-stage pipeline?

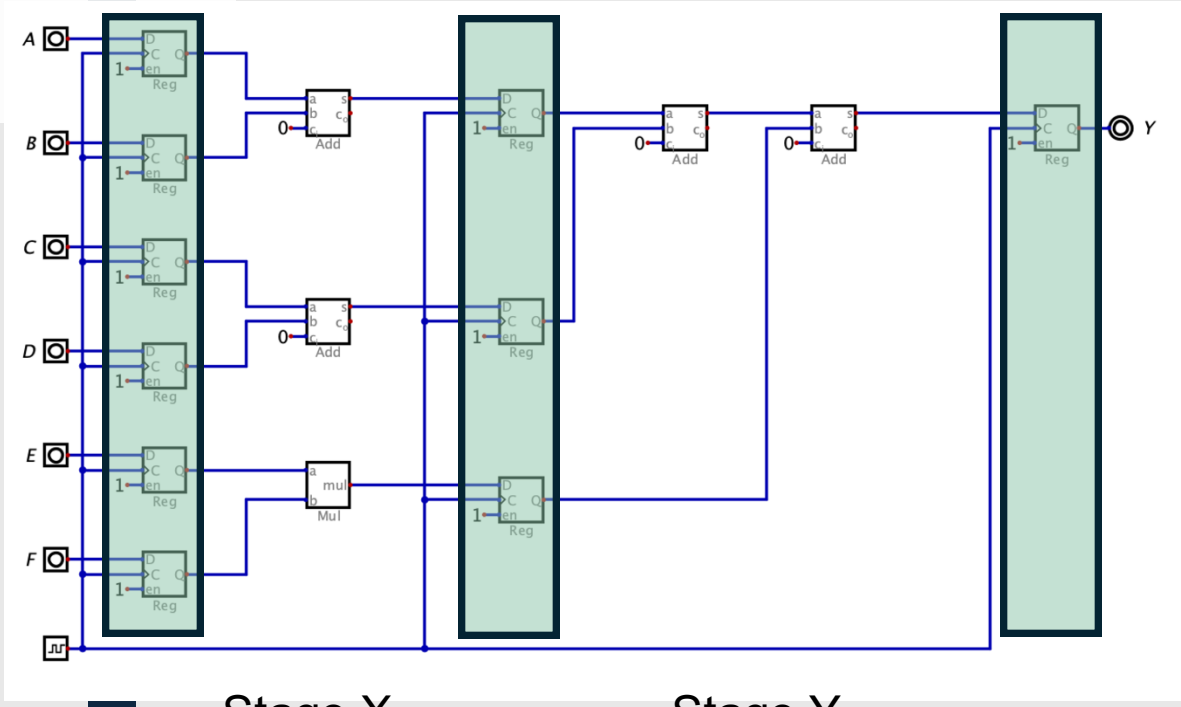
Performance Metrics

- What is the latency of an operation in the 2-stage pipeline?
  - *(1220ps)(2) = 2440ps*
- What is the latency of an operation in the 3-stage pipeline?
  - *(1220ps)(3) = 3660ps*
- Is the throughput higher in the 2-stage or 3-stage pipeline?
  - *2 stage pipeline*
- What is the speedup of completing 10 operations on the 2-stage pipeline vs 3-stage pipeline?
  - *13420ps/14640ps = .9167*
    - The two-stage pipeline is faster

$Y = A + B + C + D + EF$

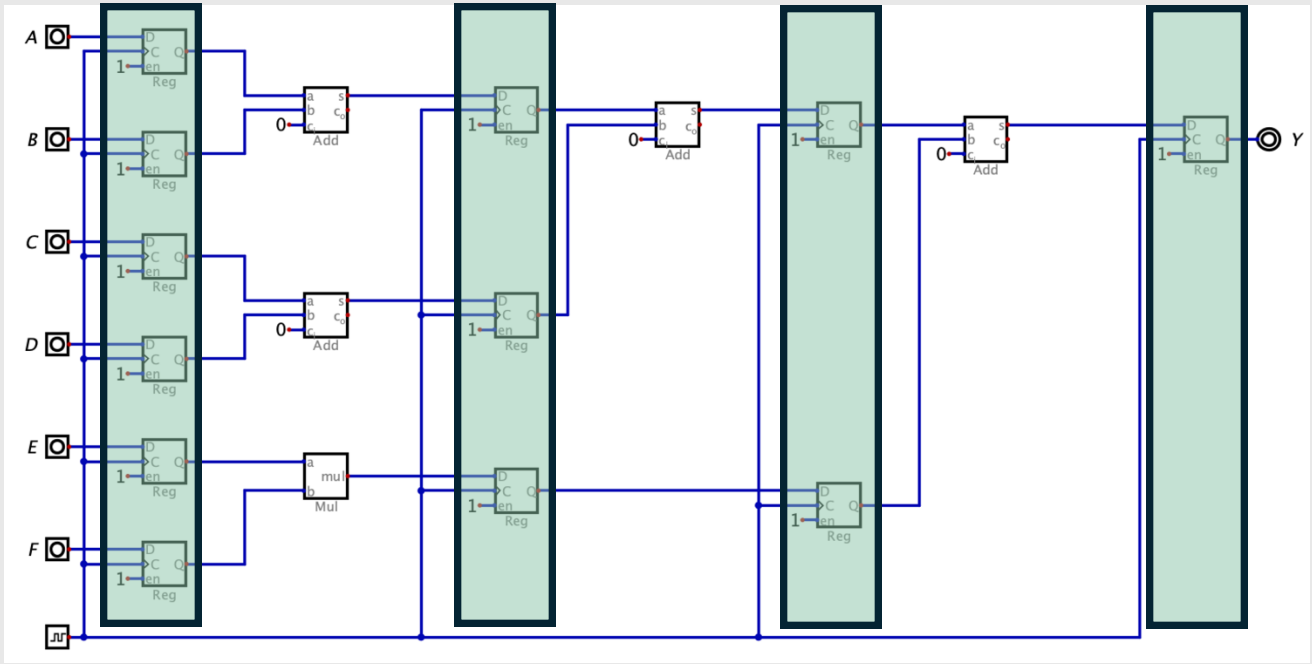
Clock-to-q delay = 20ps
Adder propagation delay = 480ps
Multiplier propagation delay = 1200ps

Clock period = 1220ps



Stage X Stage Y

2-stage Pipeline



Stage X Stage Y Stage Z

3-stage Pipeline

INTRO TO RISC-V ASSEMBLY! ☺ YAY YAY YAY FINALLY

What is the course about?

High-level programming languages

```
// High-level (C)
int add3(int a, int b)
{
    return a + b + 3;
}
```

Easy for humans to read/write

Assembly
Low-level languages

```
# Matching RISC-V assembly
# a0 = a, a1 = b;  return in a0

add   a0, a0, a1   # a0 = a0 + a1
addi  a0, a0, 3    # a0 = a0 + 3
ret                # return
```

Human readable

Machine code

```
0000000 01011 01010
000 01010 0110011

000000000011 01010
000 01010 0010011

000000000000 00001
000 00000 1100111
```

Machine-readable

Square in different assembly languages

MIPS

```
1   square(int):
2       addiu   $sp,$sp,-8
3       sw      $fp,4($sp)
4       move    $fp,$sp
5       sw      $4,8($fp)
6       lw      $2,8($fp)
7       nop
8       mult    $2,$2
9       mflo    $2
10      move    $sp,$fp
11      lw      $fp,4($sp)
12      addiu   $sp,$sp,8
13      jr      $31
14      nop
```

x86

```
1   square(int):
2       push    rbp
3       mov     rbp, rsp
4       mov     DWORD PTR [rbp-4], edi
5       mov     eax, DWORD PTR [rbp-4]
6       imul    eax, eax
7       pop     rbp
8       ret
```

ARM

```
1   square(int):
2       push    {r7}
3       sub     sp, sp, #12
4       add     r7, sp, #0
5       str     r0, [r7, #4]
6       ldr     r3, [r7, #4]
7       mul     r3, r3, r3
8       mov     r0, r3
9       adds    r7, r7, #12
10      mov     sp, r7
11      ldr     r7, [sp], #4
12      bx      lr
```

Square in different assembly languages

MIPS

x86

ARM

RISC-V

```
square(int):
    addi    sp, sp, -16       # make stack frame
    sw      ra, 12(sp)        # save return address
    sw      s0, 8(sp)         # save frame pointer
    addi    s0, sp, 16        # set up new frame pointer
    sw      a0, -4(s0)        # save argument x

    lw      t0, -4(s0)        # t0 = x
    mul     t0, t0, t0        # t0 = x * x
    mv      a0, t0            # move result to a0

    lw      ra, 12(sp)        # restore return address
    lw      s0, 8(sp)         # restore frame pointer
    addi    sp, sp, 16        # pop stack frame
    jr      ra                # return
```

```
square(int):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

Instruction Set Architecture (ISA)

- A set of rules that define the interface between the hardware and software.

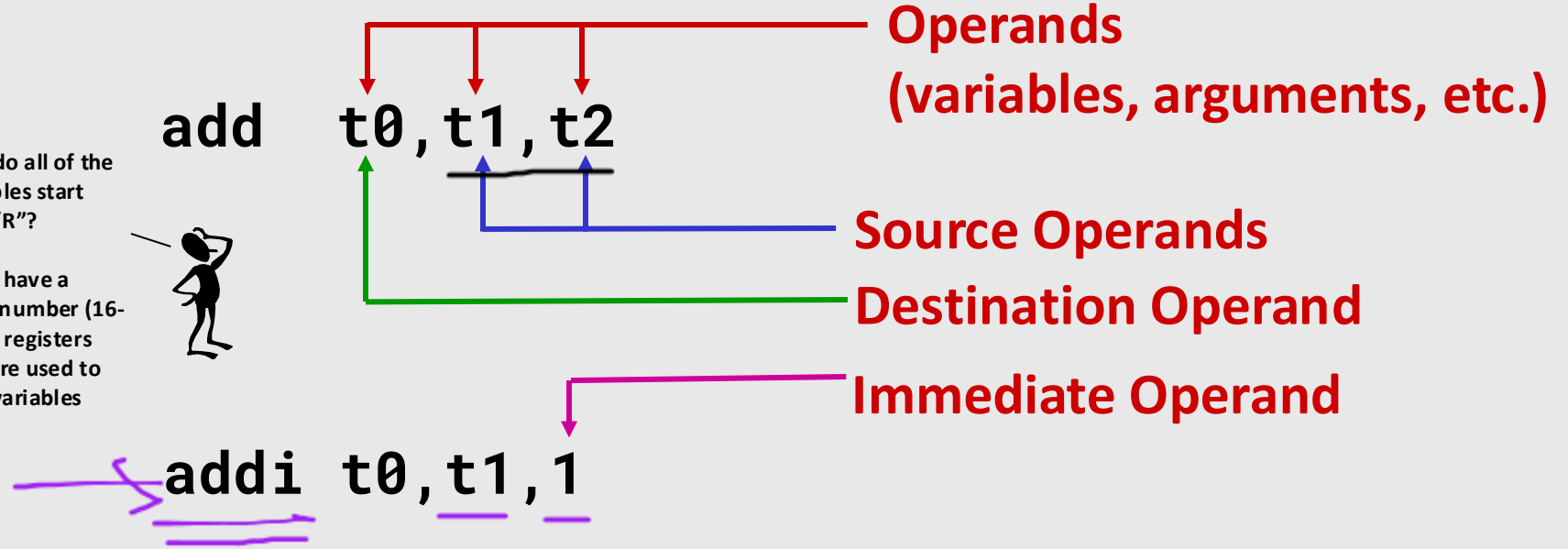
Anatomy of an Instruction

addi x0, x1, 2

- Computers execute a set of primitive operations called **instructions**
- Instructions specify an **operation** and its **operands** (arguments of the operation)
- Types of operands: **destination**, **source**, and **immediate**

x0 = x1 + 2

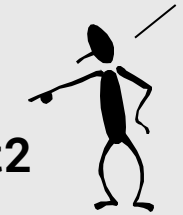
↳ constant



Meaning of an Instruction

- Operations are abbreviated into **opcodes** (1-4 letters)
- Instructions are specified with a very regular syntax
  - *Opcodes are followed by arguments*
  - *Usually the destination is next, then one or more source arguments (This is not strictly the case, but it is generally true)*
- Why this order?
  Analogy to high-level language like Java or C

```
add  t0,t1,t2
```



The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

```
int r0, r1, r2;
r0 = r1 + r2;
```

Instead of:

```
r1 + r2 = r0;
```

A Series of Instructions

- Generally…
  - *Instructions are retrieved sequentially from memory*
  - *An instruction executes to completion before the next instruction is started*
  - *But, there are exceptions to these rules*

Instructions

```
add t0, t1, t1
add t0, t0, t0
add t0, t0, t0
sub t1, t0, t1
```

What does this program do?

Variables

t0:1 2 4 48
t1:1 42
t2:8
t3:10

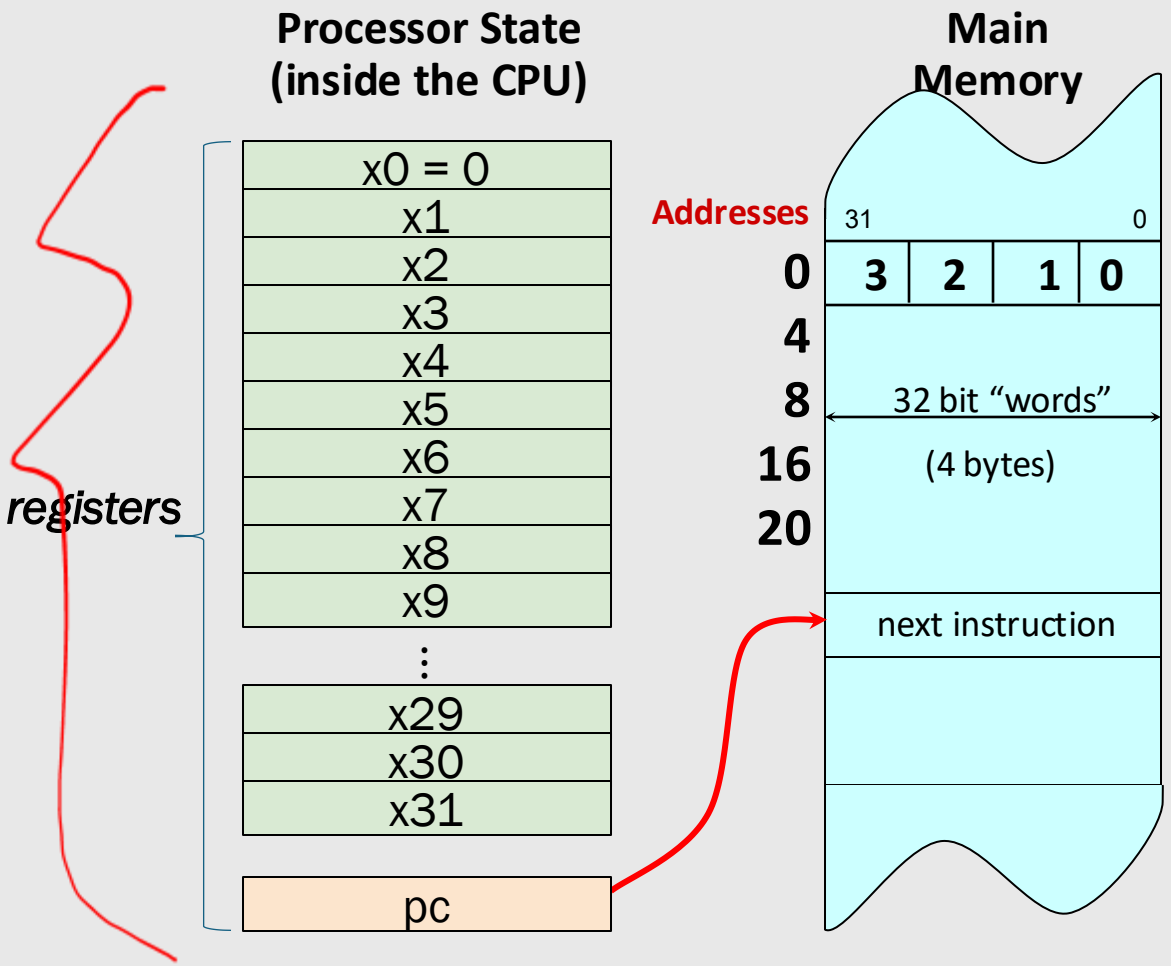
Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices…

- Trade Offs: performance, compactness, programmability
- Uniformity. Should different instructions
  - *Be the same size (number of bits)?*
  - *Take the same amount of time to execute?*
  - *Trend: Uniformity. Affords simplicity, speed, pipelining.*
- Complexity. How many different instructions? What level operations?
  - *Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc*
  - ***"Reduced Instruction Set Computer"*** *(RISC) philosophy: simple instructions, optimized for speed*
- Mix of Engineering & Art…

RISC-V Programming Model

A representative RISC machine



In Comp 311 we'll use a subset of the RISC-V core Instruction set as an example ISA (RV32I).

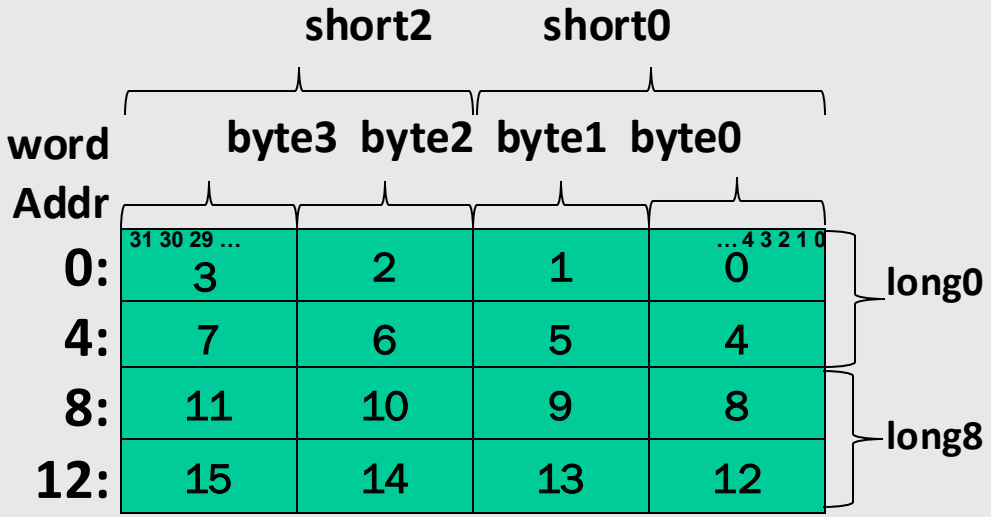
Fetch/Execute loop:

- **fetch Mem[PC]**
- **execute fetched instruction (may change PC!)**
- **PC = PC + 4**
- **repeat!**

RISC-V uses BYTE memory addresses. However, each instruction is 32-bits wide.. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

RISC-V Memory Details

- Memory locations are addressable in different sized chunks
  - *8-bit chunks (bytes)*
  - *16-bit chunks (shorts)*
  - *32-bit chunks (words)*
  - *64-bit chunks (longs/doubles)*
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (RISC-V is a byte-addressable machine)
- Most instructions are one word



RISC-V Register Details

- There are 32 named registers [x0, x1, …. x31]
- x0 is special. It always contains "0" and, when used as a destination, the result is ignored
- The operands of most instructions are registers
- This means to operate on a variables in memory you must:
  - *Load the value/values from memory into a register*
  - *Perform the instruction*
  - *Store the result back into memory*
- Going to and from memory can be expensive (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- By convention, most registers are dedicated to specific tasks

A.K.A a "Load-Store Architecture"