

Design Tradeoffs in Sequential Circuits

Software vs. Hardware Design

Timing is the key difference

1. Software interfaces (even instructions) are **timing-independent**

- Specify *what* should happen, not *when*

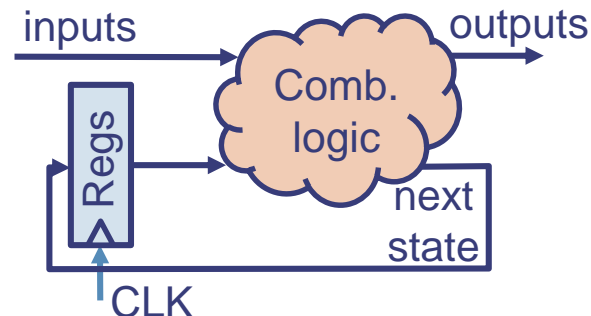
```
while (b != 0) {  
    a = a * b;  
    b = b - 1;  
}
```

```
loop: mv a1, s0  
      call mul  
      addi s0, s0, -1  
      beqz s0, loop
```

2. Hardware design is **all about timing**

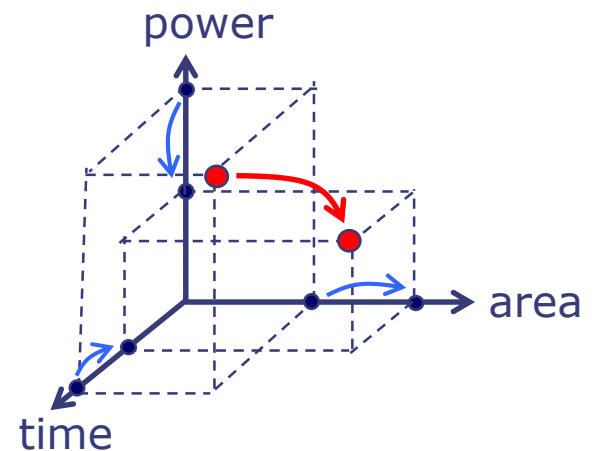
- Specify what happens on every clock cycle...
- ...which itself determines the length of the clock cycle

```
module Factorial;  
    Reg#(Word) a(0);  
    Reg#(Word) b(0);  
    rule step;  
    ...
```



Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
 - Throughput
 - Latency
 - Area of the design
 - Power consumption
 - Energy of executing a task
 -



©Advanced Micro Devices (with permission)

VS.



Justin14 (CC BY-SA 4.0)

Lecture Outline

- Examine design tradeoffs in digital logic: throughput, latency, and area
 - Power & energy are important, but out of scope for 6.004
- Extend pipelined designs to integrate them with other circuits
 - Valid bits, stall logic, queues
- Study how to generalize an FSM to solve multiple problems
 - First step towards building a general-purpose processor!

Throughput, Latency, and Area Tradeoffs

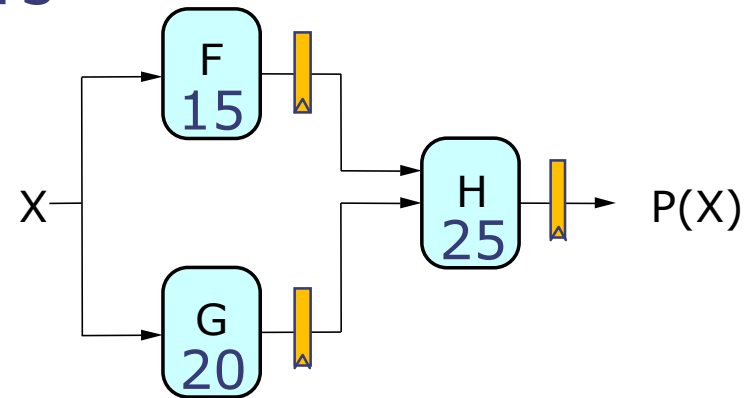
Recap: Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
 - Variable amount of input and/or output
 - Variable number of steps
- Even when combinational circuits suffice, sequential circuits allow more design tradeoffs
 - **Pipelined circuits** *improve throughput* by increasing frequency and overlapping multiple computations
 - **Folded circuits** *reduce area* by reusing a small amount of combinational logic over multiple cycles

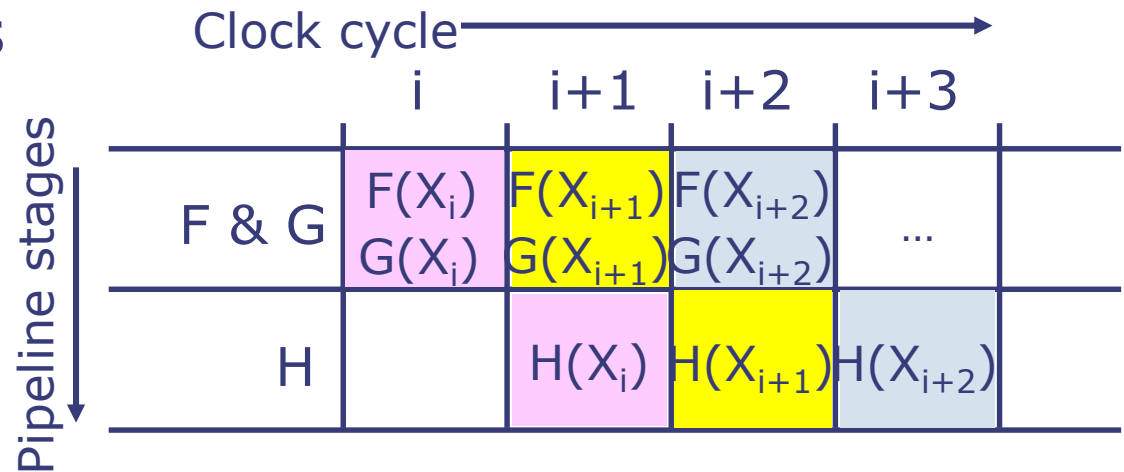
Reminder: Pipelined Circuits

- Pipelining breaks a combinational circuit over multiple **stages** using registers

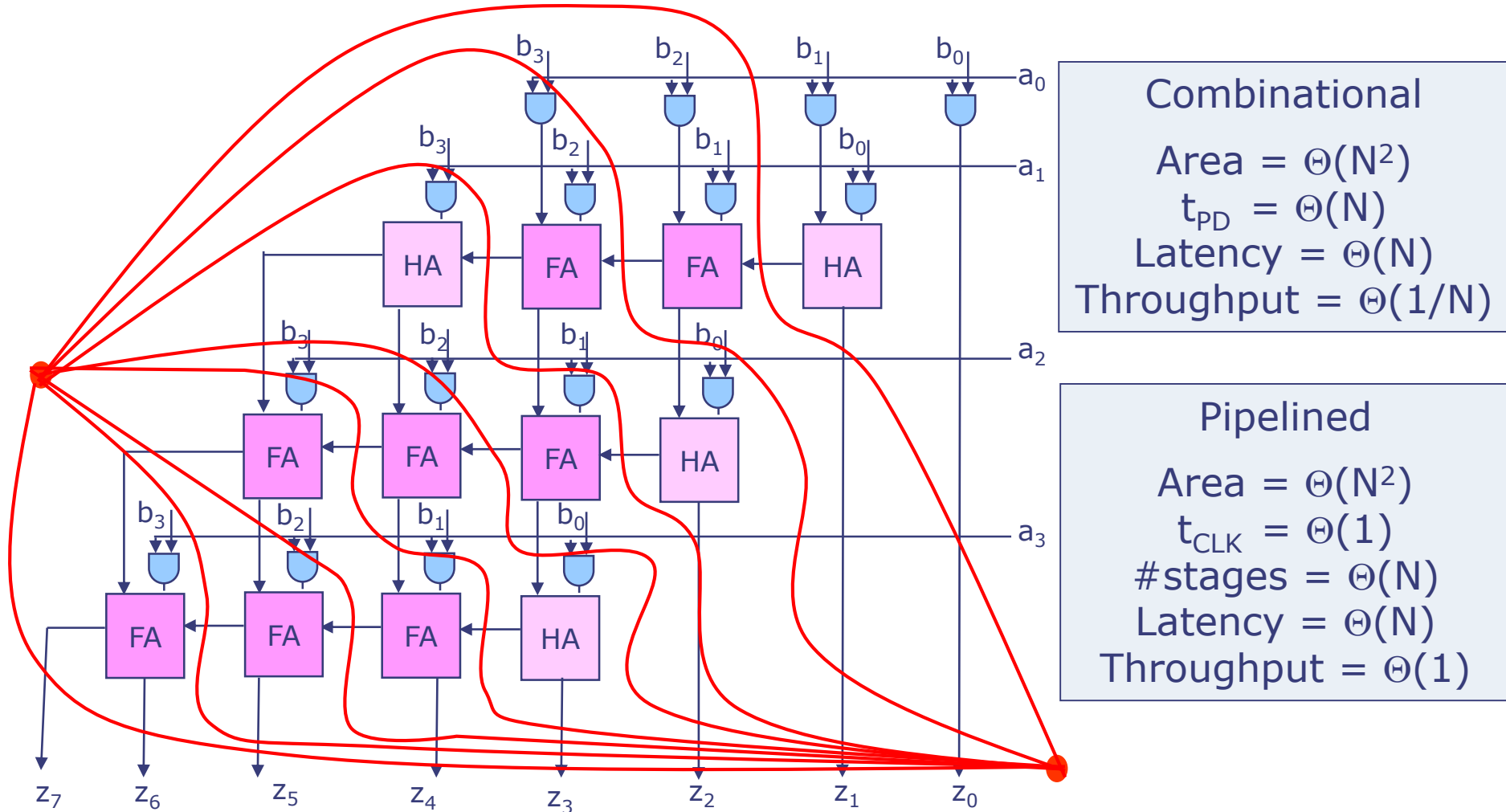
- Each computation takes multiple cycles
- On each cycle, each stage processes a different value
- $t_{\text{CLK}} \downarrow \rightarrow \text{Throughput} \uparrow$



- Pipeline diagrams

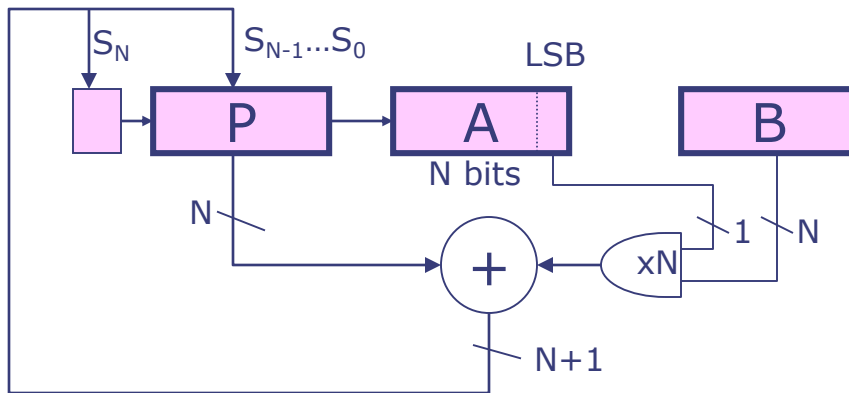


Reminder: Pipelined Multiplier



Reminder: Folded Circuits

- Combinational circuits often have repetitive logic
 - Example: N-bit multiplier has N-1 adders
- Folded circuits** use less combinational logic, reuse it over multiple cycles
 - Example: Implement multiplication with one adder, taking $\sim N$ cycles to perform the additions



Init: $P \leftarrow 0$, load A&B

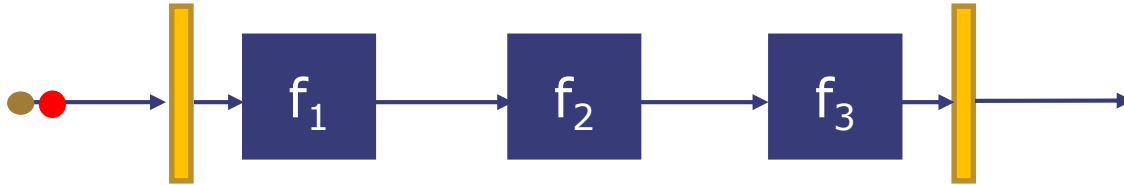
```
Repeat N times {  
   $P \leftarrow P + (A_{\text{LSB}} == 1 ? B : 0)$   
  shift  $S_N, P, A$  right one bit  
}
```

Done: 2N-bit result in P,A

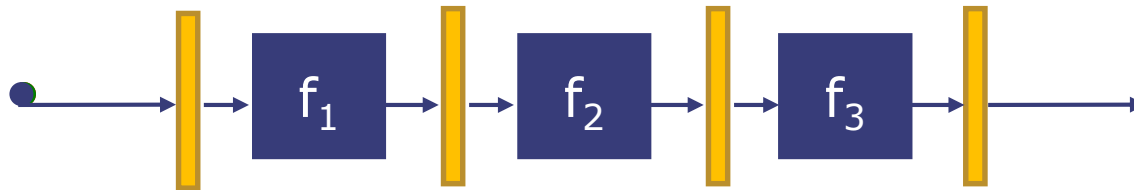
Tradeoff: reduced area, but lower throughput

Summary: Design Alternatives

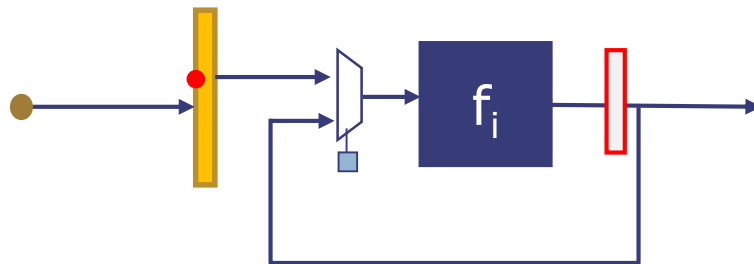
Several combinational blocks in one pipeline stage (A)



One block per pipeline stage (B)



Folded: Reuse a single block, multicycle (C)



Clock: $B \approx C < A$

Area: $C < A < B$

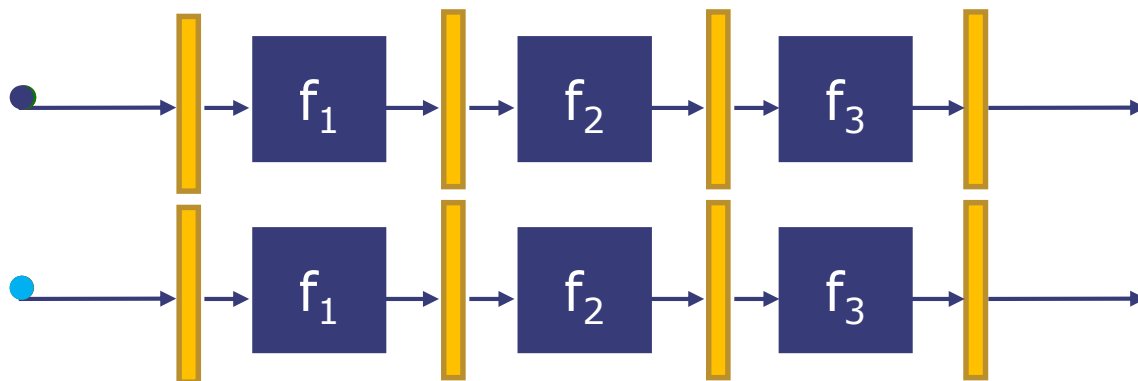
Throughput: $C < A < B$

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
 - So lower $t_{\text{PD}} \rightarrow$ lower $t_{\text{CLK}} \rightarrow$ lower latency & higher throughput
- In practice, other constraints may set t_{CLK}
 - Propagation delay of other circuits
 - Limits on power consumption
- When our own circuit is not limiting t_{CLK} , throughput and latency tradeoffs change
 - Example: 4-stage vs. 2-stage pipeline
 - If $t_{\text{CLK},4\text{stage}} = t_{\text{CLK},2\text{stage}}/2$? Throughput: 2x, Latency: 1x
 - If $t_{\text{CLK},4\text{stage}} = t_{\text{CLK},2\text{stage}}$? Throughput: 1x, Latency: 2x

Increasing Throughput with Replication

- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



- Processes two values each cycle
- Metrics vs a single pipeline:

<i>Clock?</i>	<i>Same</i>
<i>Throughput?</i>	<i>2x</i>
<i>Area?</i>	<i>2x</i>

Example: Pipeline or Replicate?

- Consider the following two multipliers

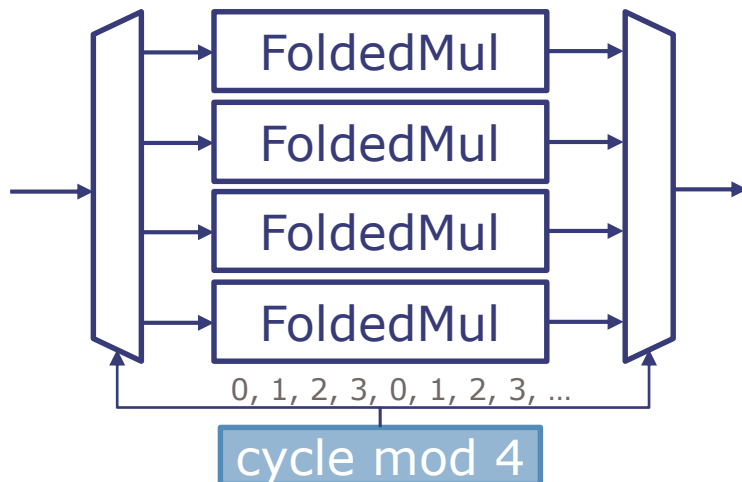


4-stage pipelined multiplier
Throughput = $1/t_{\text{CLK}}$



Folded multiplier that takes 4 cycles per output -> Throughput = $1/(4t_{\text{CLK}})$
Similar t_{CLK} vs. PipedMul, lower area

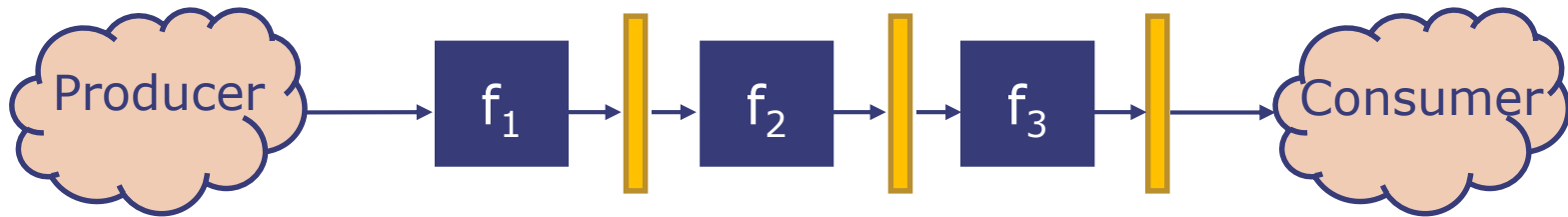
- Can you design a circuit that uses FoldedMul to achieve the same throughput as PipedMul?



Replicate FoldedMul 4 times
Each FoldedMul produces an output and takes a new input every 4 cycles
Throughput = $4 * 1/(4t_{\text{CLK}}) = 1/t_{\text{CLK}}$

Pipeline Extensions

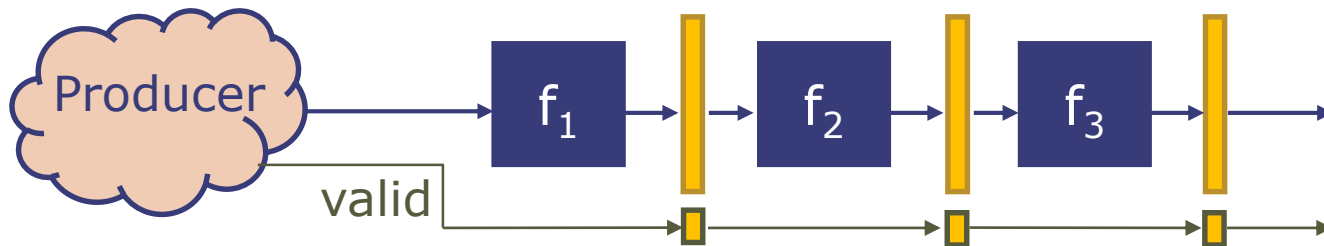
Pipeline Extensions



- Producer may not have input every cycle
→ **Valid bits**
- Consumer may not be able to accept output every cycle → **Stall logic** to freeze/pause the pipeline
- With large pipelines, may need to decouple stages further → Use **queues** instead of registers

Pipelines with Valid Bits

- If the producer won't give an input every cycle, tag each stage with a valid bit
 - In Minispec, use Maybe types

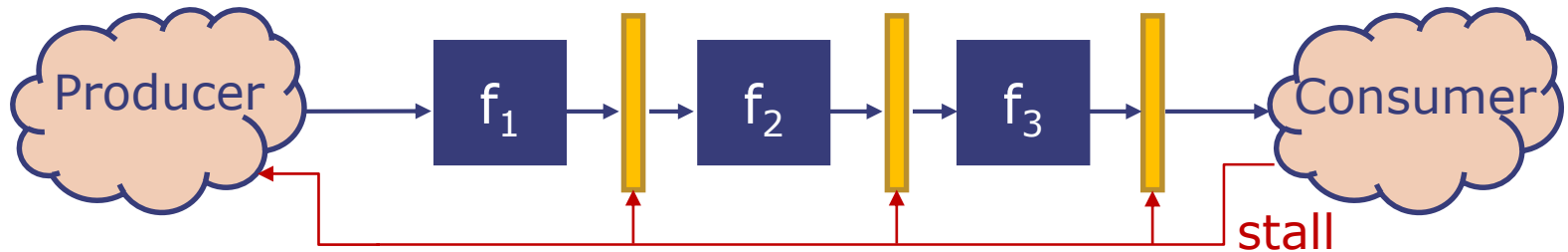


- Invalid inputs propagate through the pipeline, produce invalid outputs:

<i>Cycle</i>	1	2	3	4	5	6	7	8
Stage 1	V1	V2	Inv	V3	Inv	Inv	V4	V5
Stage 2		V1	V2	Inv	V3	Inv	Inv	V4
Stage 3			V1	V2	Inv	V3	Inv	Inv

Pipelines with Stall Logic

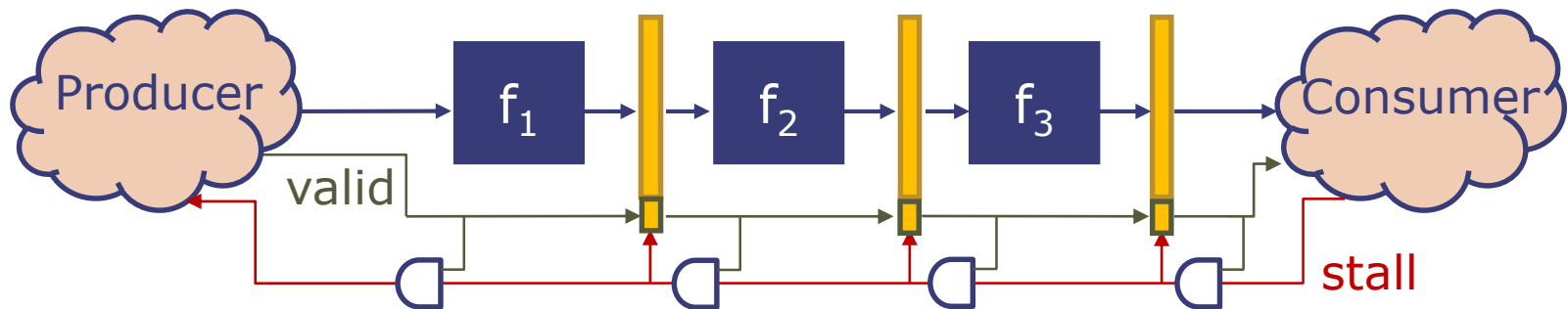
- If the consumer can't accept an output every cycle, we need to freeze the pipeline (and the producer!)
- Solution: Stall signal + registers with enable circuit
 - If stall is True, all pipeline registers retain their values



Cycle	1	2	3	4	5	6	7	8
Stage 1	V1	V2	V3	V4	V4	V5	V5	V5
Stage 2		V1	V2	V3	V3	V4	V4	V4
Stage 3			V1	V2	V2	V3	V3	V3
Stall	False	False	False	True	False	True	True	False

Combining Valid Bits + Stall Logic

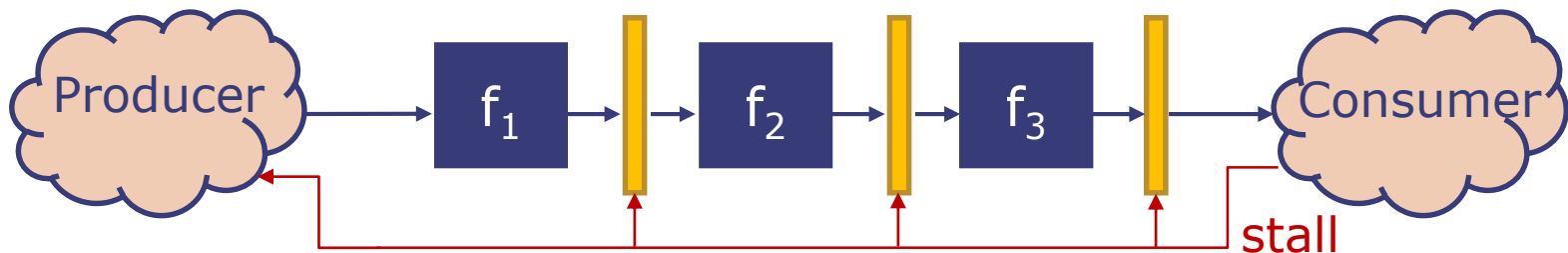
- If the consumer stalls, we can still let the pipeline make progress if a stage has an invalid value:



Cycle	1	2	3	4	5	6	7	8
Stage 1	V1	V2	Inv	V3	V4	Inv	V5	V5
Stage 2	Inv	V1	V2	Inv	V3	V4	V4	V4
Stage 3	Inv	Inv	V1	V2	V2	V3	V3	V3
Output	Inv	Inv	Inv	V1	V1	V2	V2	V2
Stall	False	False	False	True	False	True	True	False

Stalling Delay in Large Pipelines

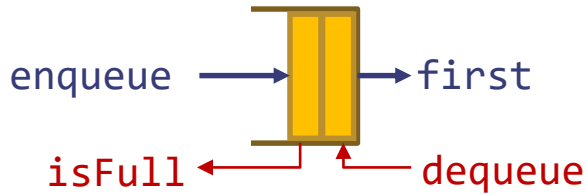
- We can't stall large pipelines immediately
 - Stall signal drives a huge number of register enables → excessive fan-out causes delay
 - Stall delay eventually sets t_{PD} , and limits t_{CLK} !



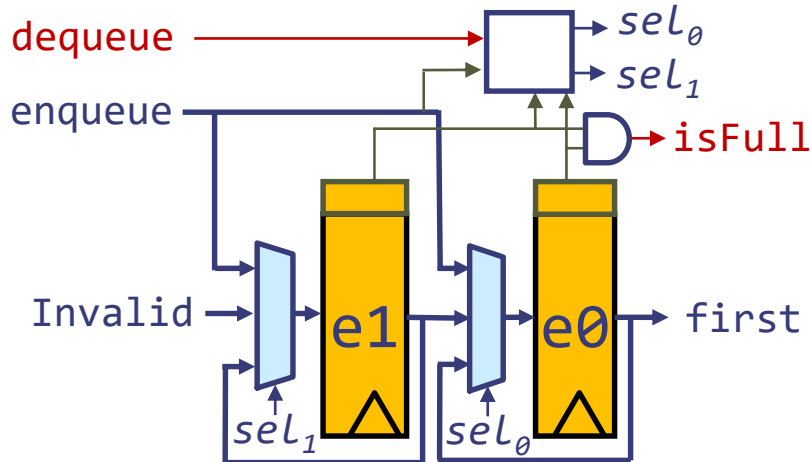
- Solution: Use queues with >1 element instead of registers to separate pipeline stages
 - Stages don't stall unless queue is full
 - Allows making stall decisions local

Example: 2-Element FIFO Queue

First-In, First-Out



- Holds up to two values
- Outputs first enqueued value
- dequeue input controls whether to advance queue
- Possible implementation:



```

module FIFO2#(type T);
  Reg#(Maybe#(T)) e0(Invalid);
  Reg#(Maybe#(T)) e1(Invalid);

  method Maybe#(T) first = e0;
  method Bool isFull =
    isValid(e0) && isValid(e1);

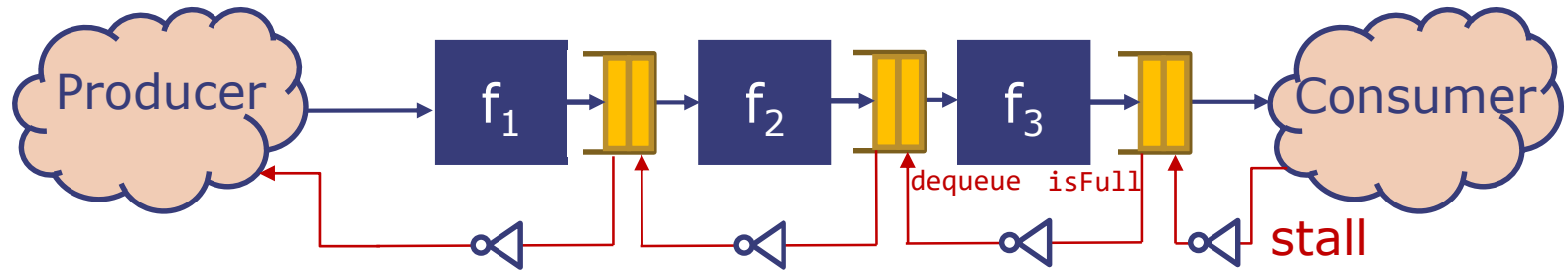
  input Bool dequeue default = False;
  input Maybe#(T) enqueue default = Invalid;

  rule tick;
    let next0 = dequeue? e1 : e0;
    let next1 = dequeue? Invalid : e1;
    if (isValid(enqueue)) begin
      if (!isValid(next0))
        next0 = enqueue;
      else if (!isValid(next1))
        next1 = enqueue;
    end
    e0 <= next0;
    e1 <= next1;
  endrule
endmodule

```

Using Queues to Decouple Stages

- Queues allow decoupling stall decisions:



- A stage stalls only if its output queue is full
- Tradeoff: Can't enqueue to full queue, even if an element is being dequeued on the same cycle
 - We could build a queue that allowed this, but this would add a combinational path from dequeue to isFull (so we'd still have the problem of high stall t_{pD} !)
- Queues also provide tolerance to variable latencies
 - Buffer multiple results without stalling producer when consumer takes variable number of cycles

From Special-Purpose FSMs to General-Purpose Processors

6.004 So Far

Finite State
Machines

Sequential
Elements

Combinational
Logic

CMOS Gates

Transistors

- What can you do with these?
 - Take a (solvable) problem
 - Design a procedure (recipe) to solve the problem
 - Design a finite state machine that implements the procedure and solves the problem
- What you'll be able to do after this week:
 - Design a machine that can solve any solvable problem, given enough time and memory (a **general-purpose computer**)

Example: Factorial FSM

Let's design a circuit to compute factorial(N)

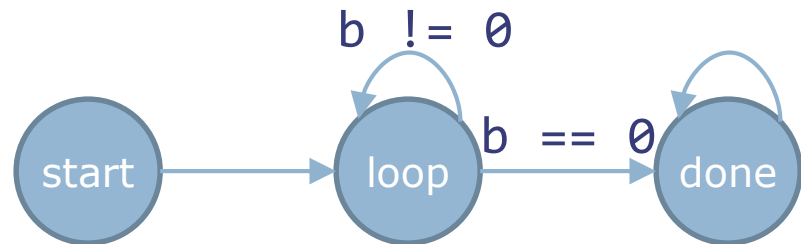
Python:

```
a = 1
b = N
while b != 0:
    a = a * b
    b = b - 1
```

C:

```
int a = 1;
int b = N;
while (b != 0) {
    a = a * b;
    b = b - 1;
}
```

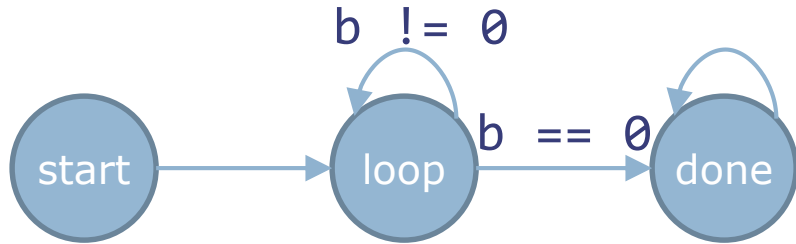
High-level FSM:



$a \leftarrow 1$	$a \leftarrow a * b$	$a \leftarrow a$
$b \leftarrow N$	$b \leftarrow b - 1$	$b \leftarrow b$

- Describes cycle-by-cycle behavior
- **Registers** (a, b)
- States (start, loop, done)
- Boolean transitions ($b=0$, $b \neq 0$)
- **Register assignments** in states (e.g., $a \leftarrow a * b$)

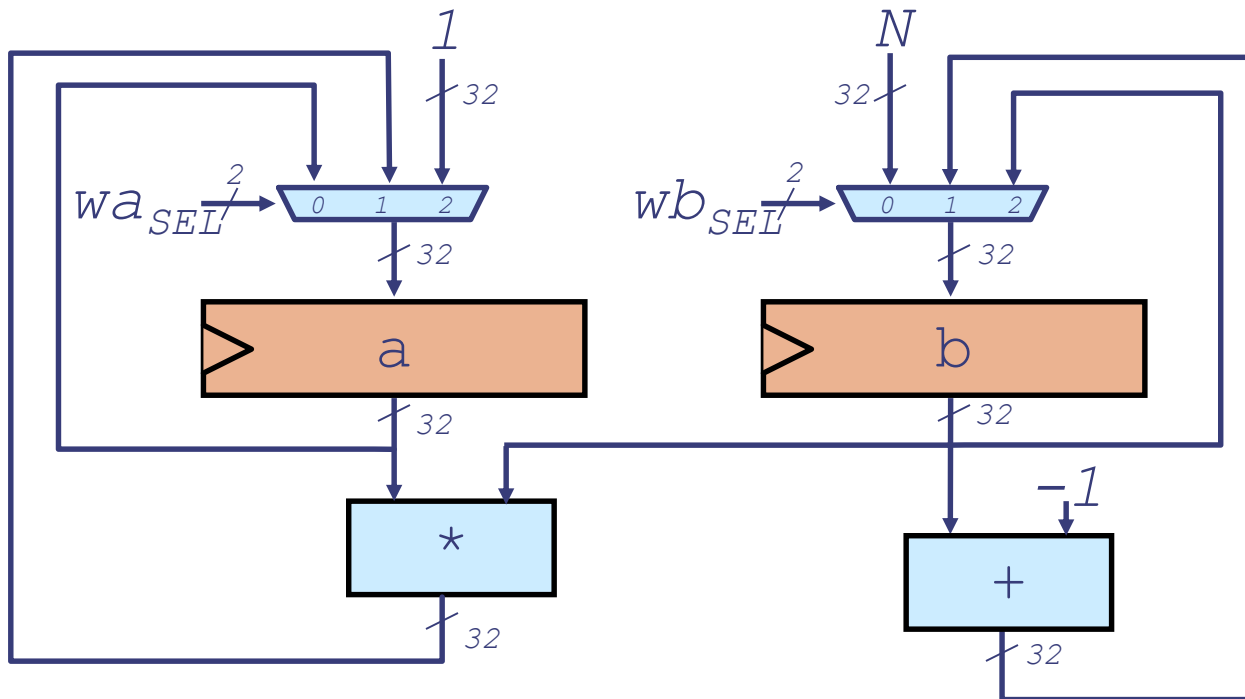
Datapath for Factorial



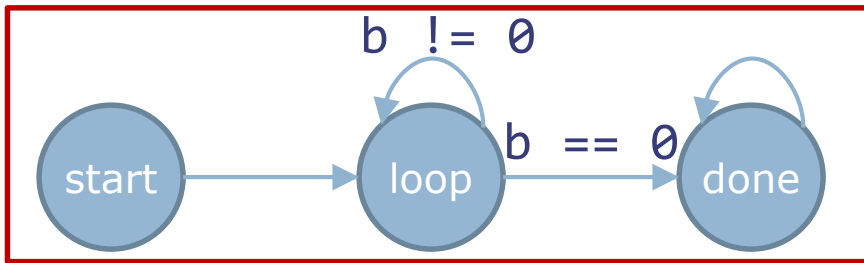
```

a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
    
```

- Implement registers
- Implement combinational circuit for each assignment
- Connect to input muxes

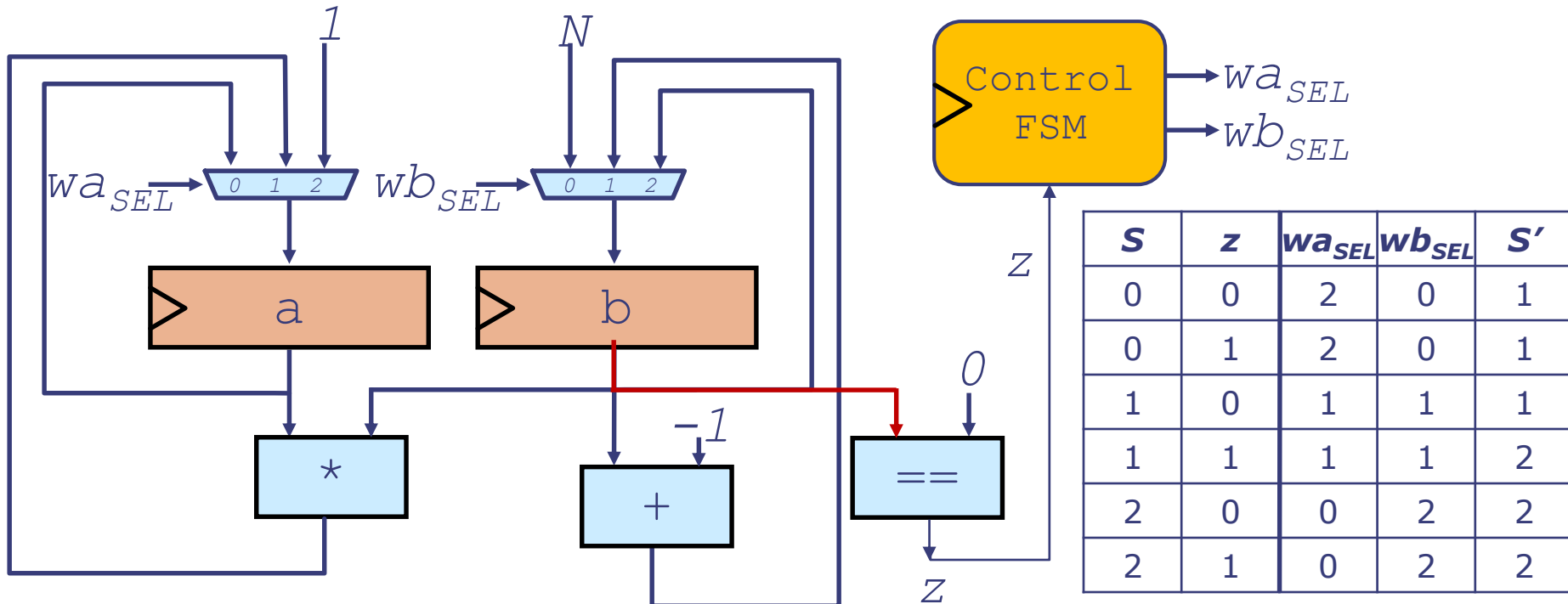


Control FSM for Factorial



$a \leq 1$ $a \leq a * b$ $a \leq a$
 $b \leq N$ $b \leq b - 1$ $b \leq b$

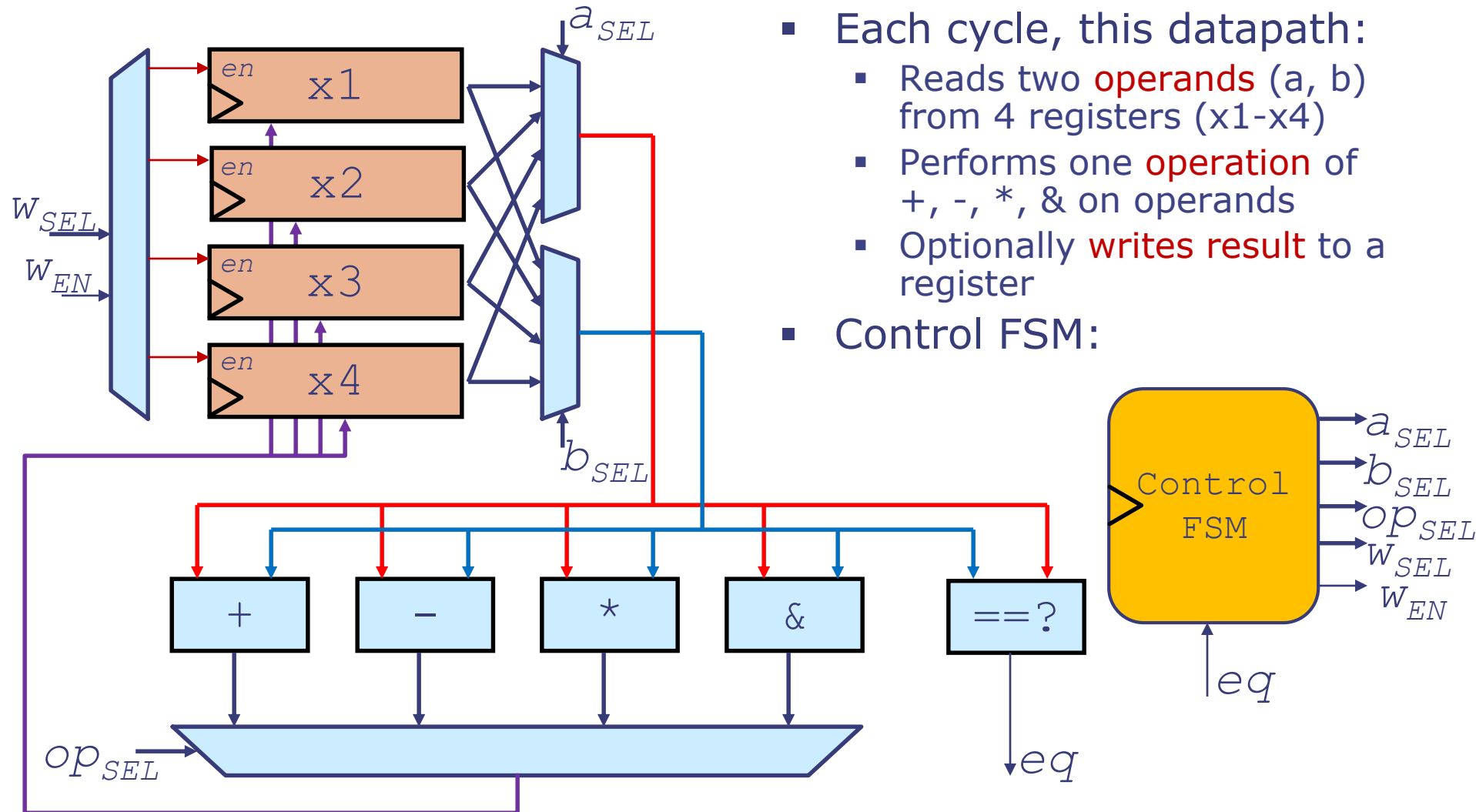
- Implement combinational logic for transition conditions
- Implement control FSM:
 - States: High-level FSM states
 - Inputs: Transition conditions
 - Outputs: Mux select signals



Programming the Datapath

- We can use our factorial datapath and change the control FSM to solve other problems! Examples:
 - Multiplication
 - Squaring
- But very limited problems. Reasons:
 - Limited storage (only two registers!)
 - Limited set of operations, and inputs to those operations
 - Limited inputs to the control FSM

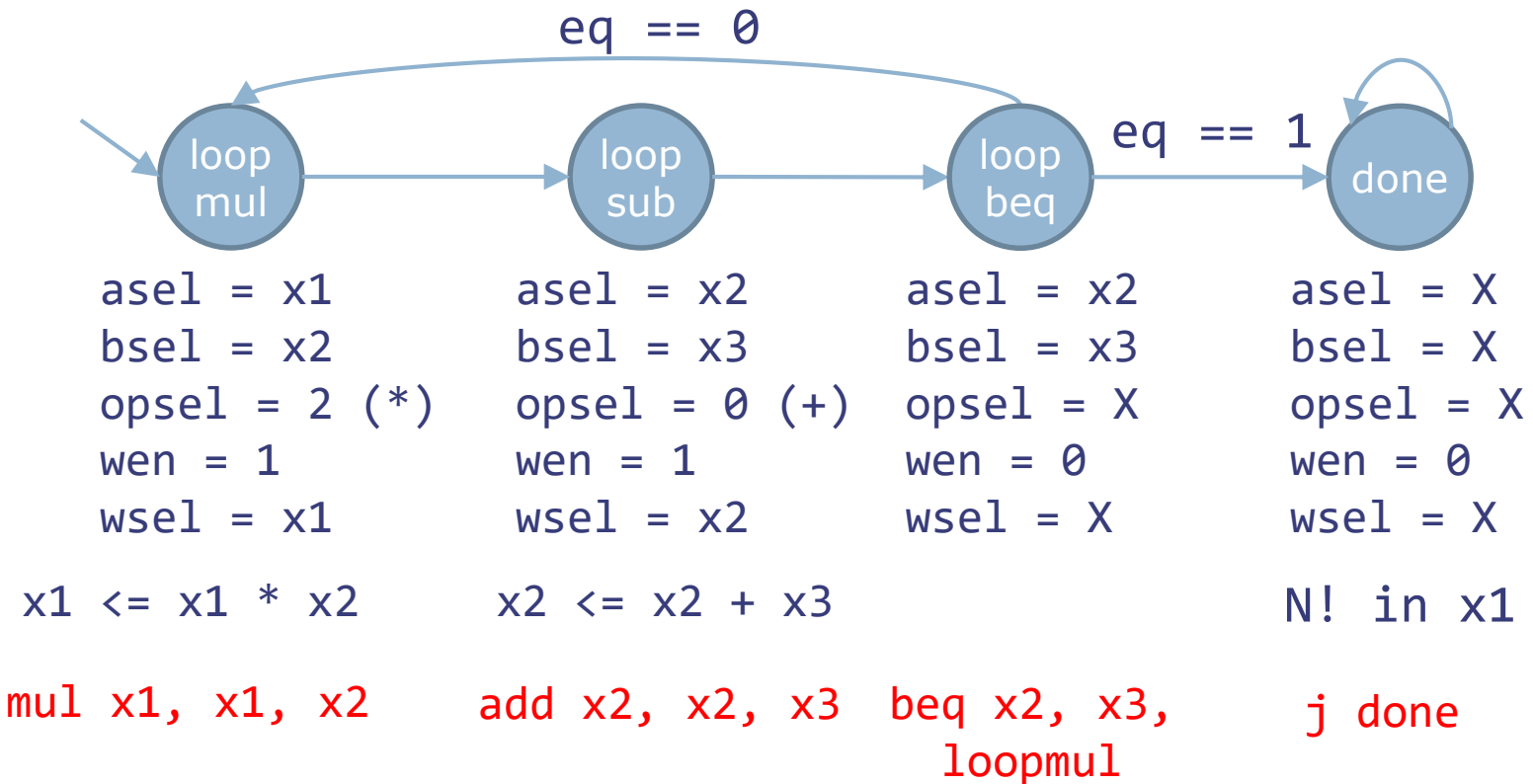
A Simple Programmable Datapath



- Each cycle, this datapath:
 - Reads two **operands** (a, b) from 4 registers (x1-x4)
 - Performs one **operation** of +, -, *, & on operands
 - Optionally **writes result** to a register
- Control FSM:

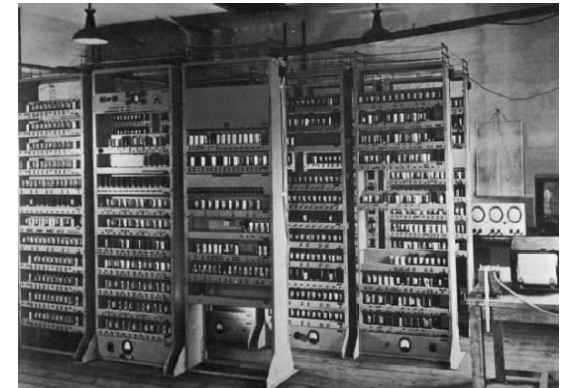
A Control FSM for Factorial

- Assume initial register contents:
 - x1 value = 1
 - x2 value = N
 - x3 value = -1
 - x4 value = 0
- Control FSM:



New Problem → New Control FSM

- You can solve many problems with this datapath!
 - GCD, Fibonacci, exponentiation, division, square root, ...
 - But nothing that requires more than four registers
- By designing a control FSM, we are **programming the datapath**
- Early digital computers were programmed this way!
 - ENIAC (1943):
 - First general-purpose digital computer
 - Programmed by setting huge array of dials and switches
 - Reprogramming it took about 3 weeks
- Modern computers instead store programs in memory, coded as a sequence of instructions



more on next lecture...

Thank you!

Next lecture: Building a RISC-V processor