Design Tradeoffs in Sequential Circuits

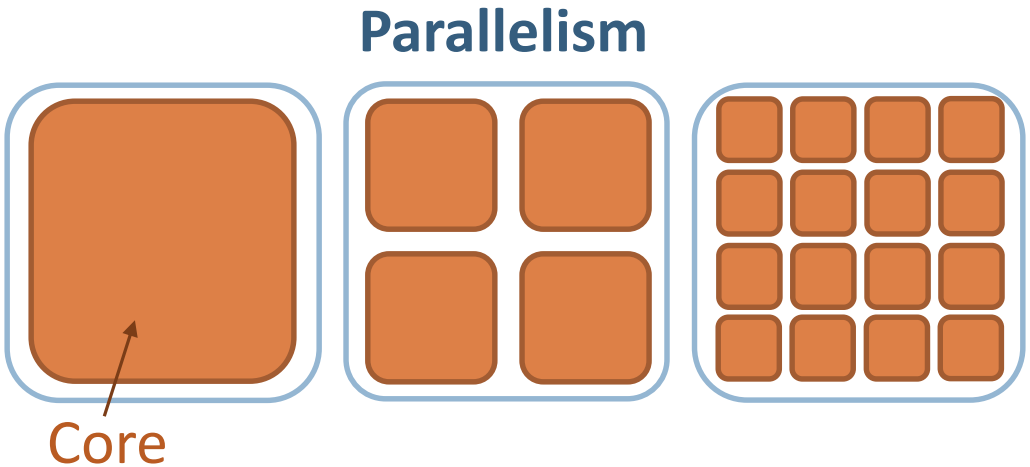
Why Study Hardware Design?

Why Study Hardware Design?

- The end of Moore's Law is transforming computing
  - Past 50+ years: Exponential growth in transistors per chip (~2x/2 years)
  - Now/soon: How to improve performance without more or faster transistors?

Why Study Hardware Design?

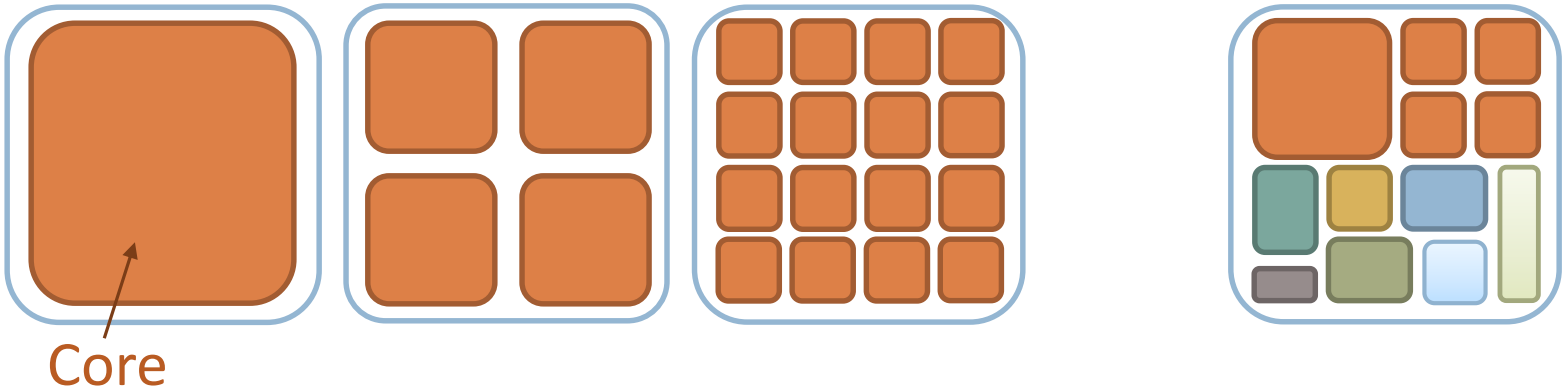
- The end of Moore's Law is transforming computing
  - Past 50+ years: Exponential growth in transistors per chip (~2x/2 years)
  - Now/soon: How to improve performance without more or faster transistors?



Why Study Hardware Design?

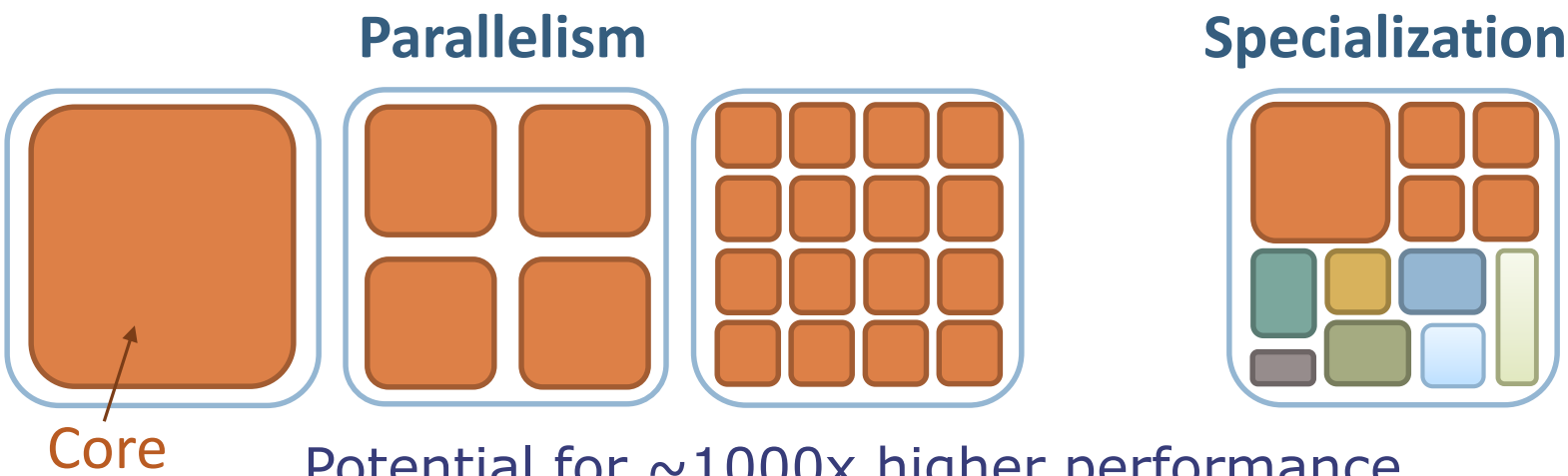
- The end of Moore's Law is transforming computing
  - Past 50+ years: Exponential growth in transistors per chip (~2x/2 years)
  - Now/soon: How to improve performance without more or faster transistors?

Parallelism **Specialization**



Why Study Hardware Design?

- The end of Moore's Law is transforming computing
  - Past 50+ years: Exponential growth in transistors per chip (~2x/2 years)
  - Now/soon: How to improve performance without more or faster transistors?

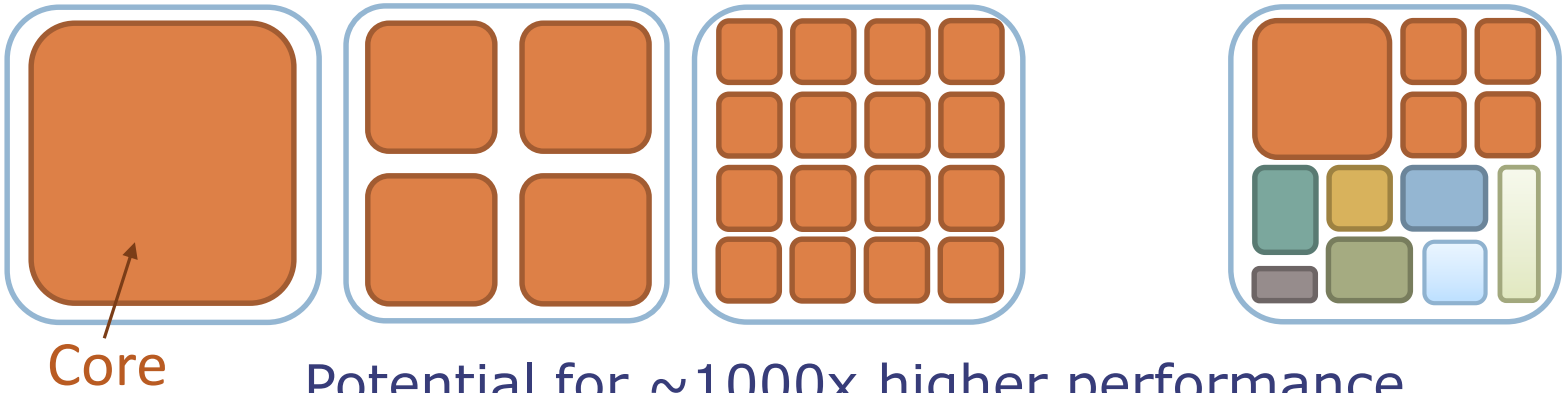


Potential for ~1000x higher performance

Why Study Hardware Design?

- The end of Moore's Law is transforming computing
  - Past 50+ years: Exponential growth in transistors per chip (~2x/2 years)
  - Now/soon: How to improve performance without more or faster transistors?

Parallelism **Specialization**



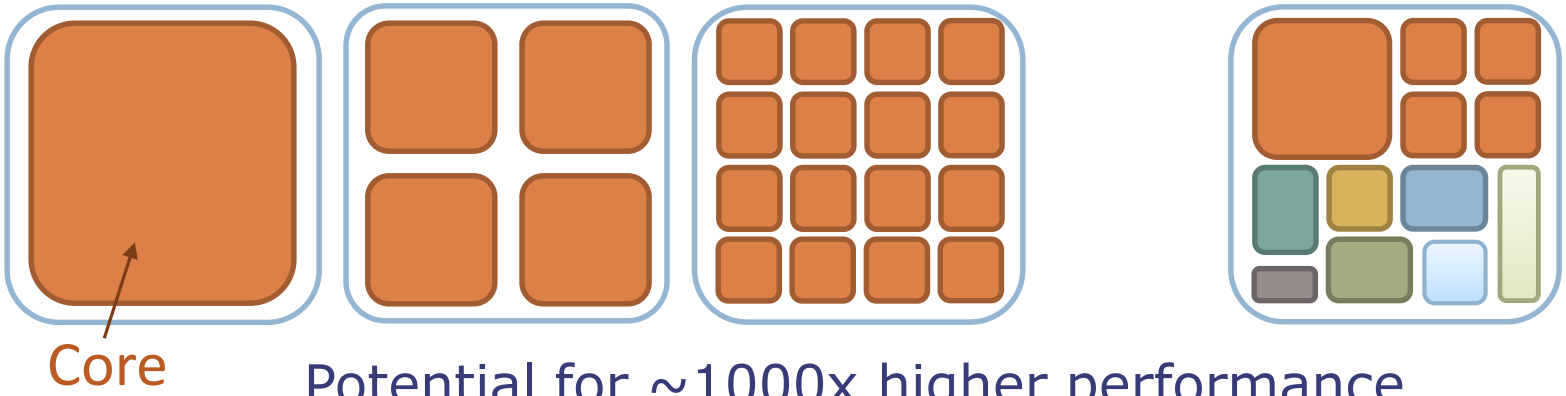
Potential for ~1000x higher performance

Many companies now building custom hardware

Why Study Hardware Design?

- The end of Moore's Law is transforming computing
  - Past 50+ years: Exponential growth in transistors per chip (~2x/2 years)
  - Now/soon: How to improve performance without more or faster transistors?

Parallelism **Specialization**



Potential for ~1000x higher performance

Many companies now building custom hardware

You are likely to build or interact closely with specialized hardware in your career!

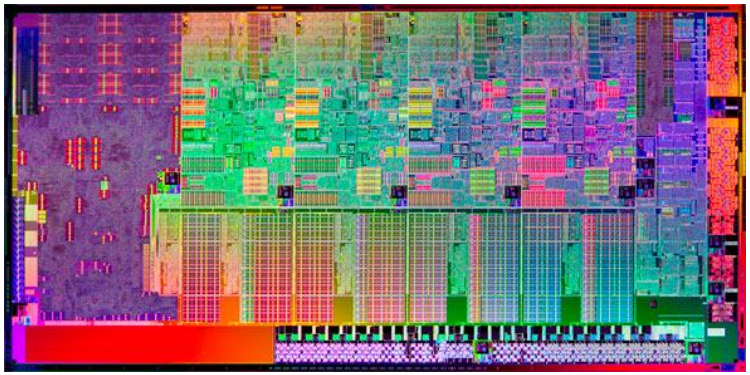
The Trend Towards Specialization

- Most of the area in modern processors is spent on specialized units, not general-purpose cores

The Trend Towards Specialization

- Most of the area in modern processors is spent on specialized units, not general-purpose cores

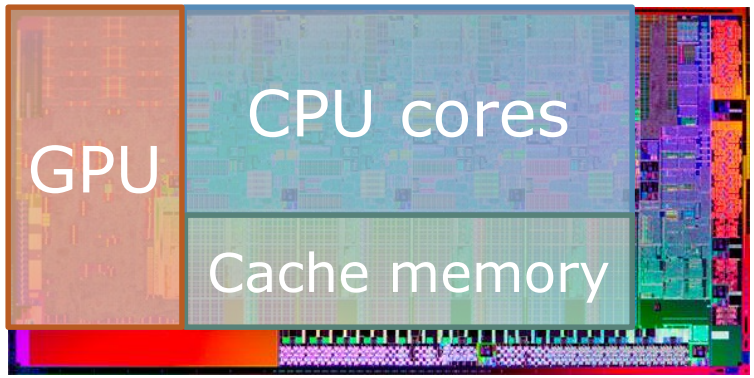
Intel Sandy Bridge (2010)
216mm^2, 1.1B transistors, 32nm



The Trend Towards Specialization

- Most of the area in modern processors is spent on specialized units, not general-purpose cores

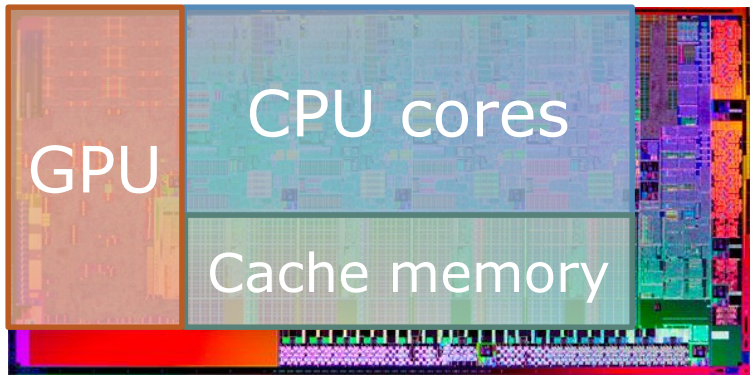
Intel Sandy Bridge (2010)
216mm^2, 1.1B transistors, 32nm



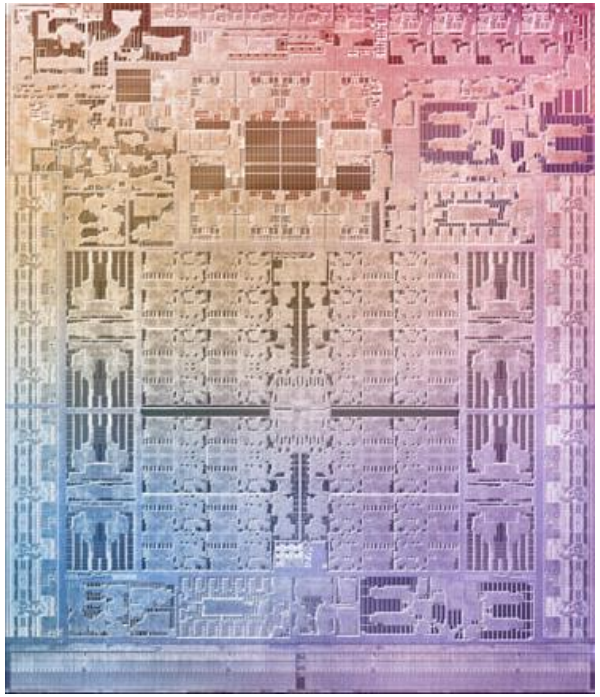
The Trend Towards Specialization

- Most of the area in modern processors is spent on specialized units, not general-purpose cores

Intel Sandy Bridge (2010)
216mm^2, 1.1B transistors, 32nm



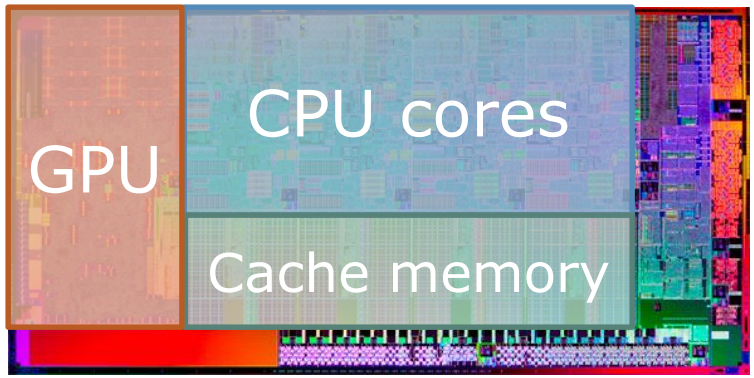
Apple M1 Max (2021)
420mm^2, 57B transistors, 5nm



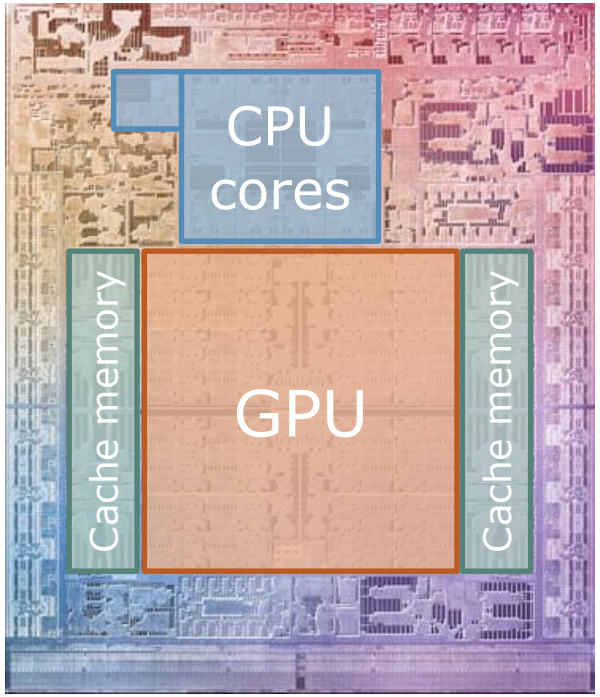
The Trend Towards Specialization

- Most of the area in modern processors is spent on specialized units, not general-purpose cores

Intel Sandy Bridge (2010)
216mm^2, 1.1B transistors, 32nm



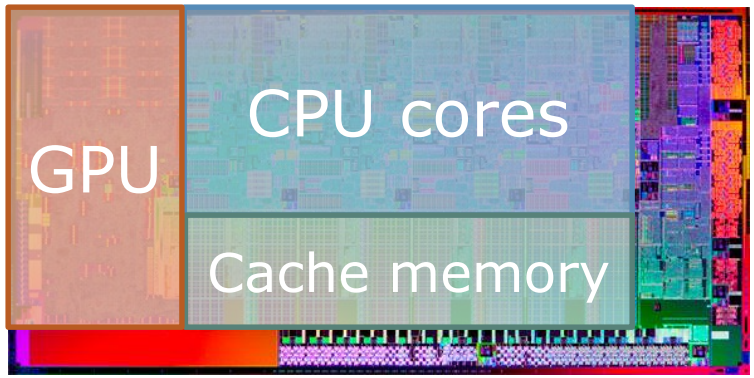
Apple M1 Max (2021)
420mm^2, 57B transistors, 5nm



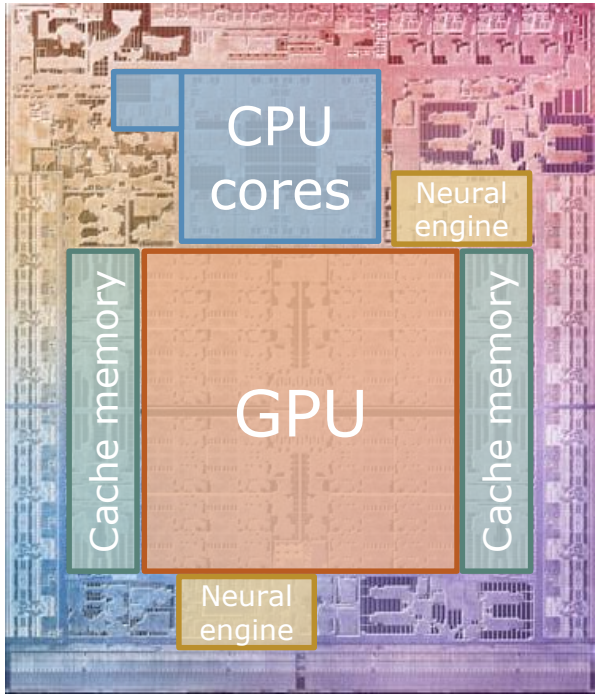
The Trend Towards Specialization

- Most of the area in modern processors is spent on specialized units, not general-purpose cores

Intel Sandy Bridge (2010)
216mm^2, 1.1B transistors, 32nm



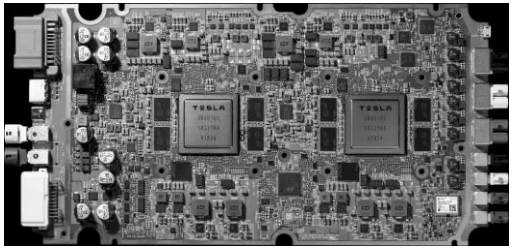
Apple M1 Max (2021)
420mm^2, 57B transistors, 5nm



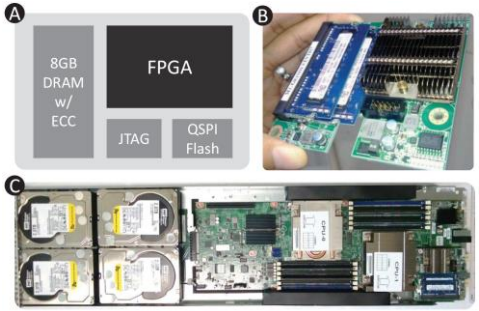
The Trend Towards Specialization

- Google TPU (2016+) is a specialized chip and system for deep learning
- Tesla builds custom chips for autonomous driving (FSD) and DNN training (Dojo)
- Microsoft and Amazon use FPGA accelerators in their datacenters

Google, 2017



Tesla, 2019

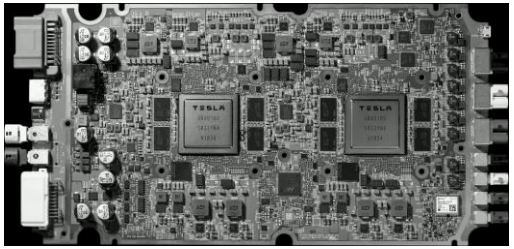


Microsoft, 2014
[Catapult ISCA 2014]

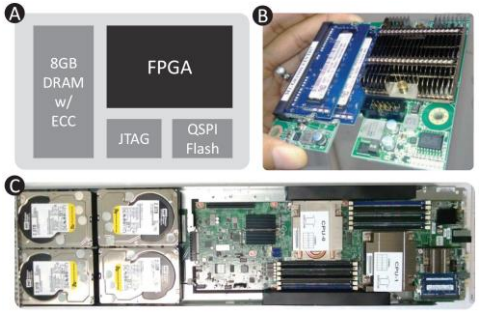
The Trend Towards Specialization

- Google TPU (2016+) is a specialized chip and system for deep learning
- Tesla builds custom chips for autonomous driving (FSD) and DNN training (Dojo)
- Microsoft and Amazon use FPGA accelerators in their datacenters
- None of these companies built chips ~10 years ago!

Google, 2017



Tesla, 2019

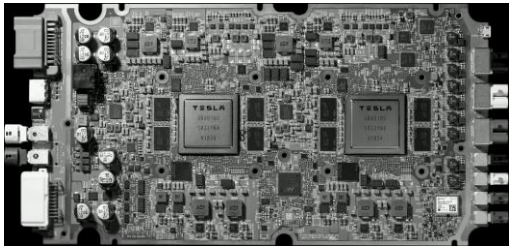


Microsoft, 2014
[Catapult ISCA 2014]

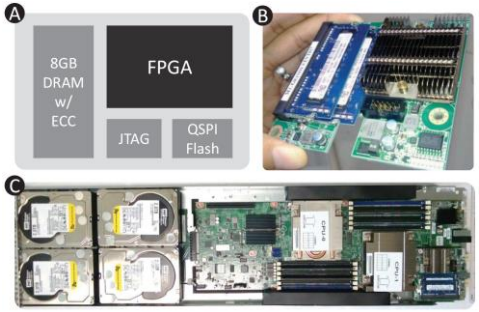
The Trend Towards Specialization

- Google TPU (2016+) is a specialized chip and system for deep learning
- Tesla builds custom chips for autonomous driving (FSD) and DNN training (Dojo)
- Microsoft and Amazon use FPGA accelerators in their datacenters
- None of these companies built chips ~10 years ago!
- It is likely that you will build or closely interact with specialized hardware
  - Limited abstraction
  - Using these systems well requires understanding hardware design

Google, 2017



Tesla, 2019



Microsoft, 2014
[Catapult ISCA 2014]

Lecture Outline

- Examine design tradeoffs in digital logic: throughput, latency, and area
  - Power & energy are important, but out of scope for 6.004
  - Case study: Multiplier

- Study how to generalize an FSM to solve multiple problems
  - First step towards building a general-purpose processor!

Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs

Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency

Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency
  3. Area of the design

Optimizing Your Hardware Design

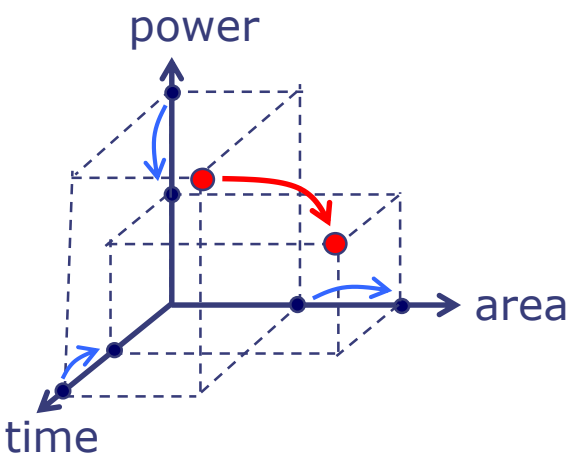
- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency
  3. Area of the design
  4. Power consumption
  5. Energy of executing a task

Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency
  3. Area of the design
  4. Power consumption
  5. Energy of executing a task
  6. …

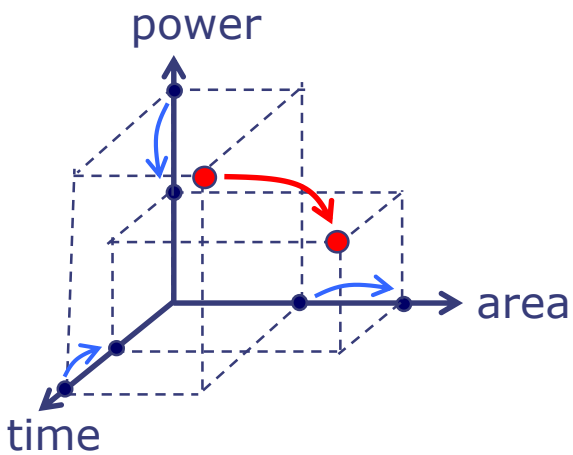
Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency
  3. Area of the design
  4. Power consumption
  5. Energy of executing a task
  6. …



Optimizing Your Hardware Design

- There are many possible implementations of the same functionality, with different area-time-power tradeoffs
- Optimization metrics:
  1. Throughput
  2. Latency
  3. Area of the design
  4. Power consumption
  5. Energy of executing a task
  6. …



©Advanced Micro Devices (with permission)

vs.

Justin14 (CC BY-SA 4.0)

Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

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

Benefits of Sequential Logic

- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input and/or output
  - Variable number of steps

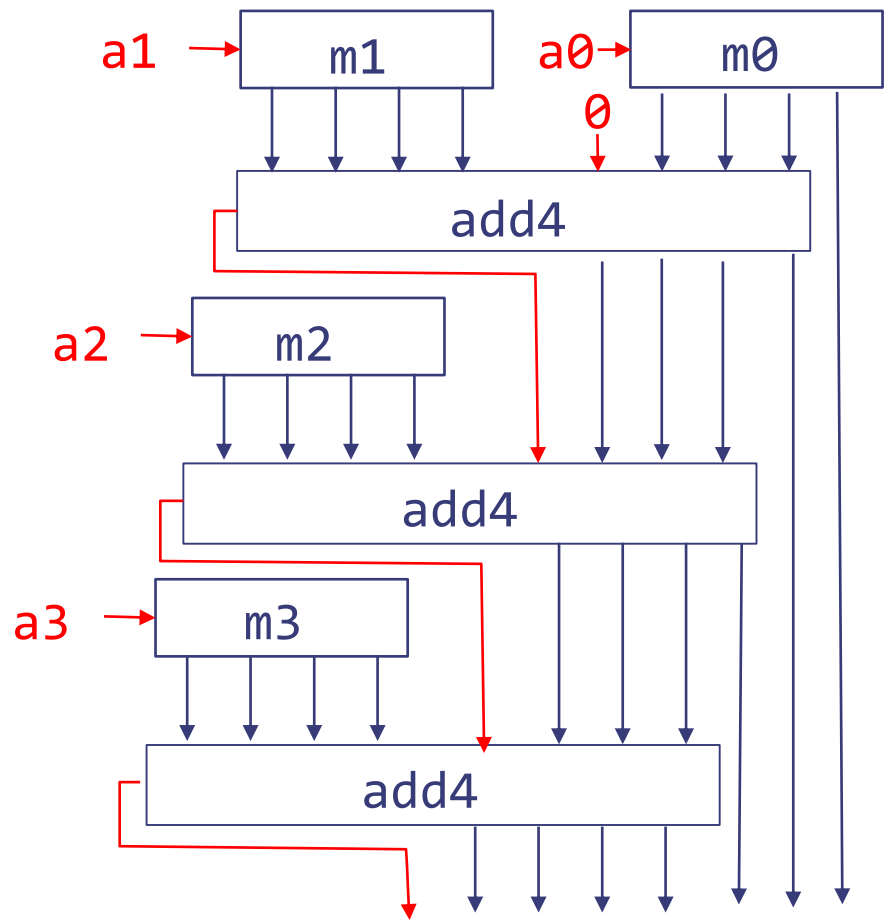
- Even when combinational circuits suffice, sequential circuits allow more design tradeoffs
  - Pipelined circuits *improve throughput* by decreasing clock period and overlapping multiple computations
  - Multi-cycle / folded circuits *reduce area* by reusing a small amount of combinational logic over multiple cycles

Reminder: Multiplication by repeated addition

```
b Multiplicand   1101   (13)
a Multiplier   * 1011   (11)

tp                0000
m0          +     1101
            ------------
tp               01101
m1          +    1101
            ------------
tp              100111
m2          +   0000
            ------------
tp             0100111
m3          + 1101
            ------------
tp            10001111   (143)
```

```
mi = (a[i]==0)? 0 : b;
```

Implementation: Cascade of N-1 N-bit adders



Implementation of mi

Implementation of mi

```
mi = (a[i]==0)? 0 : b;
```

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

```
mi = (a[i]==0)? 0 : b;
```

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

= AND

```
mi = (a[i]==0)? 0 : b;
```

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

= AND

```
mi = (a[i]==0)? 0 : b;
```

$$
\begin{array}{rrrrrrr}
 & & & & B_3 & B_2 & B_1 & B_0 \\
 & & & & \times\ A_3 & A_2 & A_1 & A_0 \\
\hline
 & & & & B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 & \text{m0} \\
 & & & B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 & & \text{m1} \\
 & & B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 & & & \text{m2} \\
+ & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 & & & & \text{m3} \\
\hline
\end{array}
$$

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

= AND

```
mi = (a[i]==0)? 0 : b;
```

$$
\begin{array}{rrrrrrrr}
 & & & & & B_3 & B_2 & B_1 & B_0 \\
 & & & & \times & A_3 & A_2 & A_1 & A_0 \\
\hline
 & & & & & B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
 & & & & B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 & \\
 & & & B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 & & \\
 & + & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 & & & \\
\hline
\end{array}
$$

BA_i called a "partial product" → B_3A_0 B_2A_0 B_1A_0 B_0A_0

m0
m1
m2
m3

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

= AND

```
mi = (a[i]==0)? 0 : b;
```

$$
\begin{array}{rrrrrrrr}
 & & & & & B_3 & B_2 & B_1 & B_0 \\
 & & & & \times & A_3 & A_2 & A_1 & A_0 \\
\hline
 & & & & & B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 \\
 & & & & B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 & \\
 & & & B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 & & \\
 & + & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 & & & \\
\hline
\end{array}
$$

BA_i called a "partial product" → B_3A_0 B_2A_0 B_1A_0 B_0A_0 m0

B_3A_1 B_2A_1 B_1A_1 B_0A_1 m1

B_3A_2 B_2A_2 B_1A_2 B_0A_2 m2

$+ B_3A_3$ B_2A_3 B_1A_3 B_0A_3 m3

Multiplying N-digit number by M-digit number gives (N+M)-digit result

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

= AND

```
mi = (a[i]==0)? 0 : b;
```

$$
\begin{array}{rrrrrrrr}
 & & & & & B_3 & B_2 & B_1 & B_0 \\
 & & & & \times & A_3 & A_2 & A_1 & A_0 \\
\hline
 & & & & & B_3A_0 & B_2A_0 & B_1A_0 & B_0A_0 & \text{m0} \\
 & & & & B_3A_1 & B_2A_1 & B_1A_1 & B_0A_1 & & \text{m1} \\
 & & & B_3A_2 & B_2A_2 & B_1A_2 & B_0A_2 & & & \text{m2} \\
 & + & B_3A_3 & B_2A_3 & B_1A_3 & B_0A_3 & & & & \text{m3} \\
\hline
\end{array}
$$

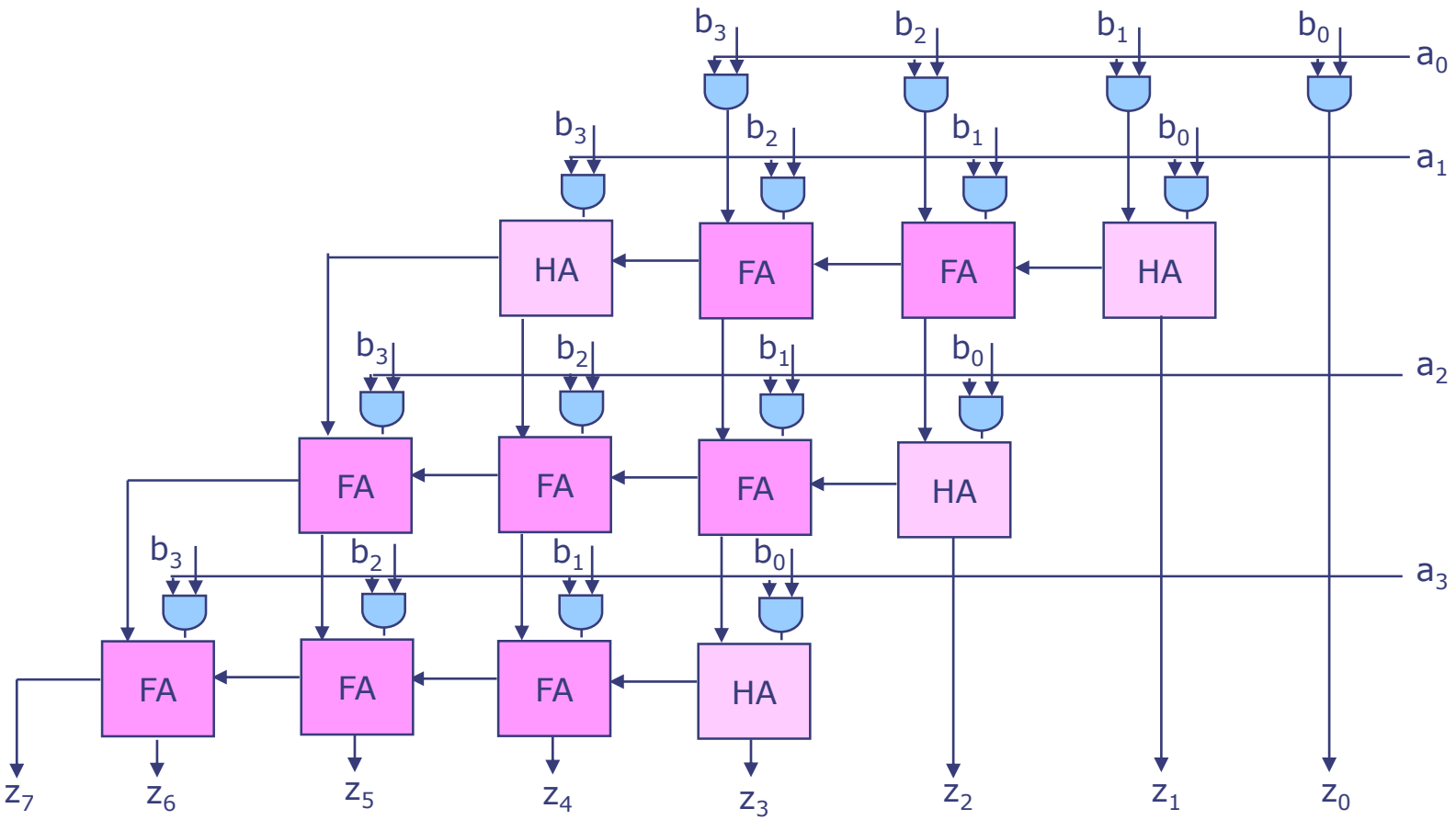
BA_i called a "partial product"

Multiplying N-digit number by M-digit number gives (N+M)-digit result

Easy part: forming partial products (bunch of AND gates)
Hard part: adding M N-bit partial products

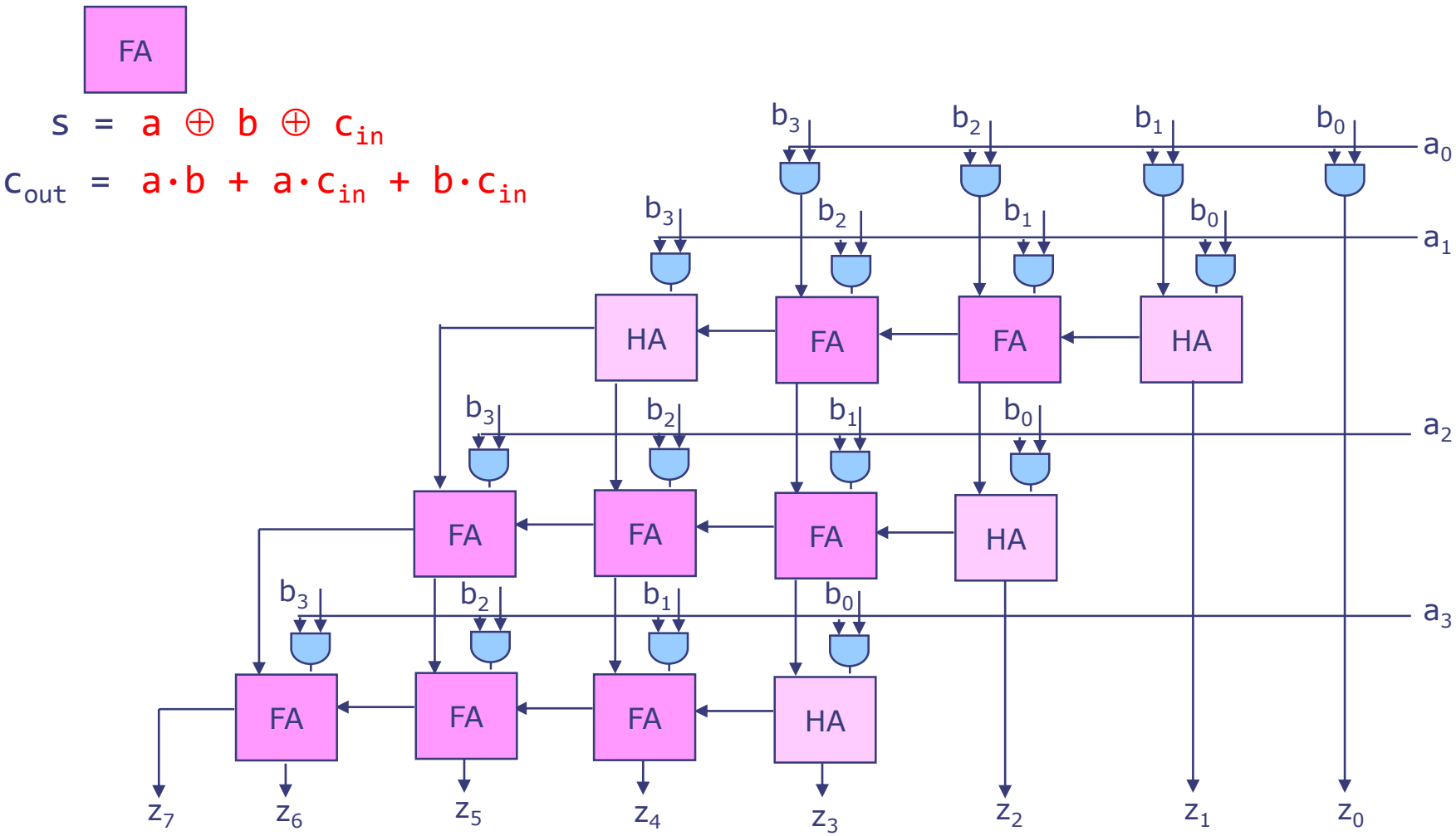
Combinational Multiplier Redrawn

Using ripple-carry adders



Combinational Multiplier Redrawn

Using ripple-carry adders



Combinational Multiplier Redrawn

Using ripple-carry adders

FA

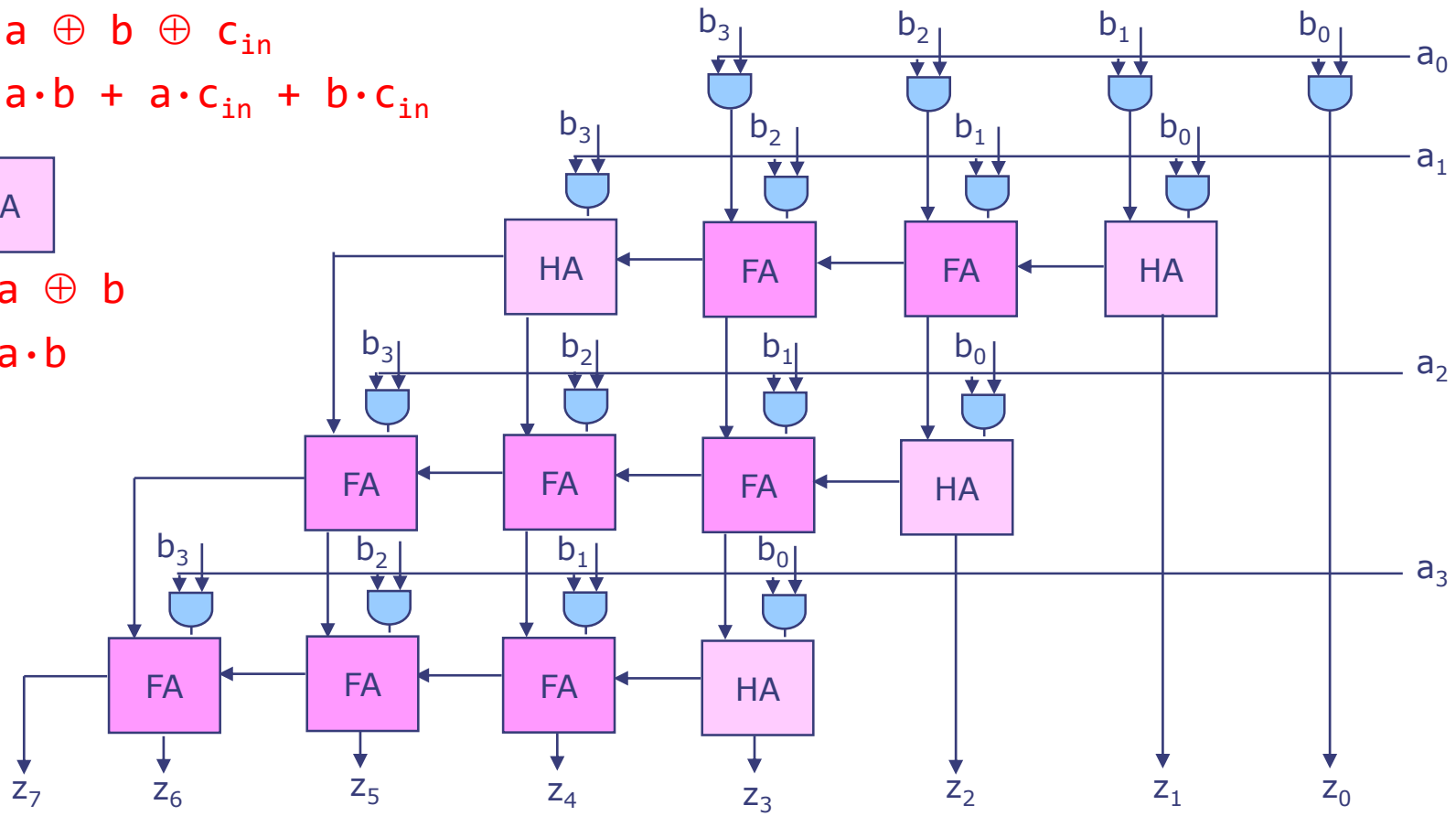
$s = a \oplus b \oplus c_{in}$

$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$

HA

$s = a \oplus b$

$c_{out} = a \cdot b$



Combinational Multiplier Redrawn

Using ripple-carry adders

FA

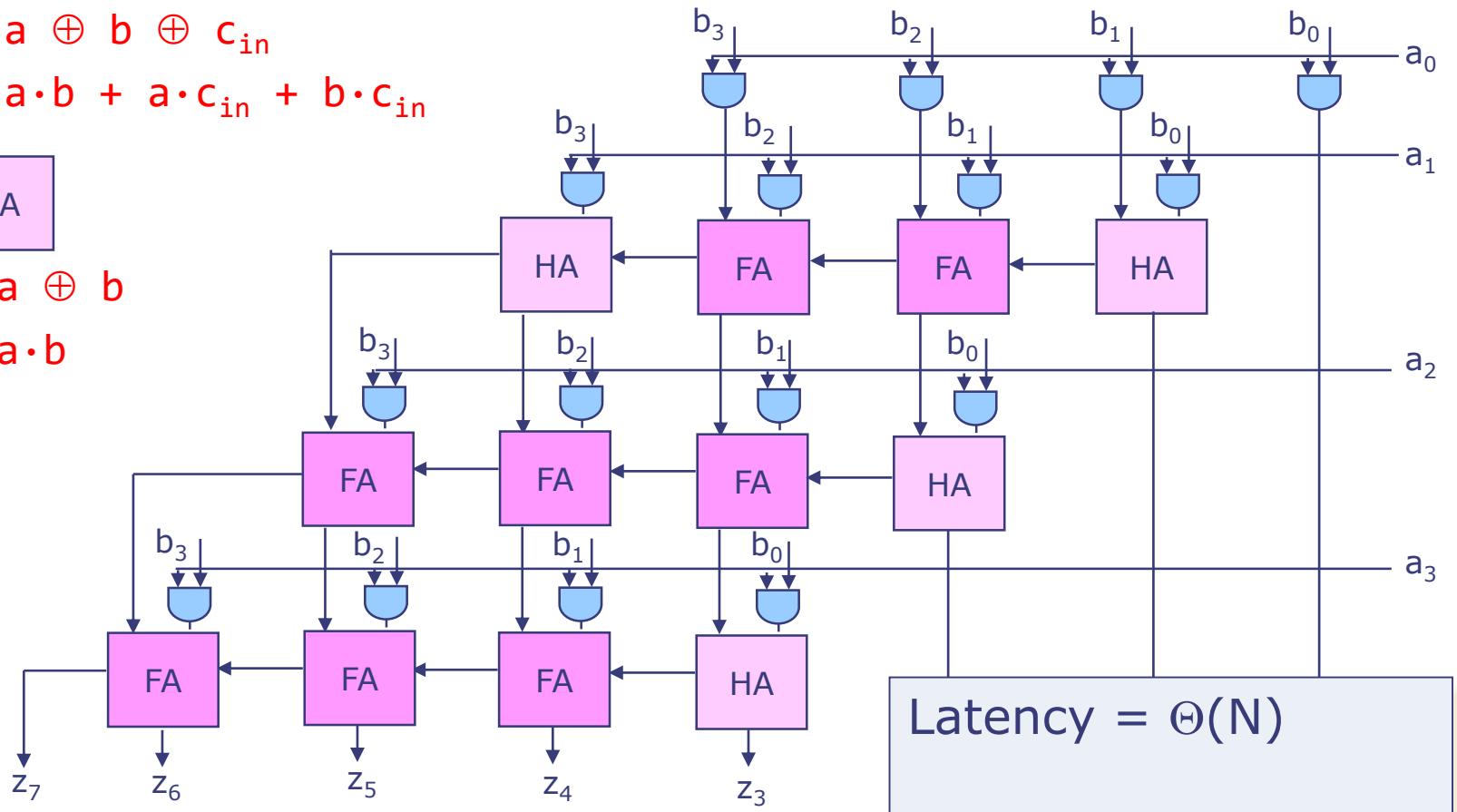
$s = a \oplus b \oplus c_{in}$

$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$

HA

$s = a \oplus b$

$c_{out} = a \cdot b$



Latency = $\Theta(N)$

Combinational Multiplier Redrawn

Using ripple-carry adders

FA

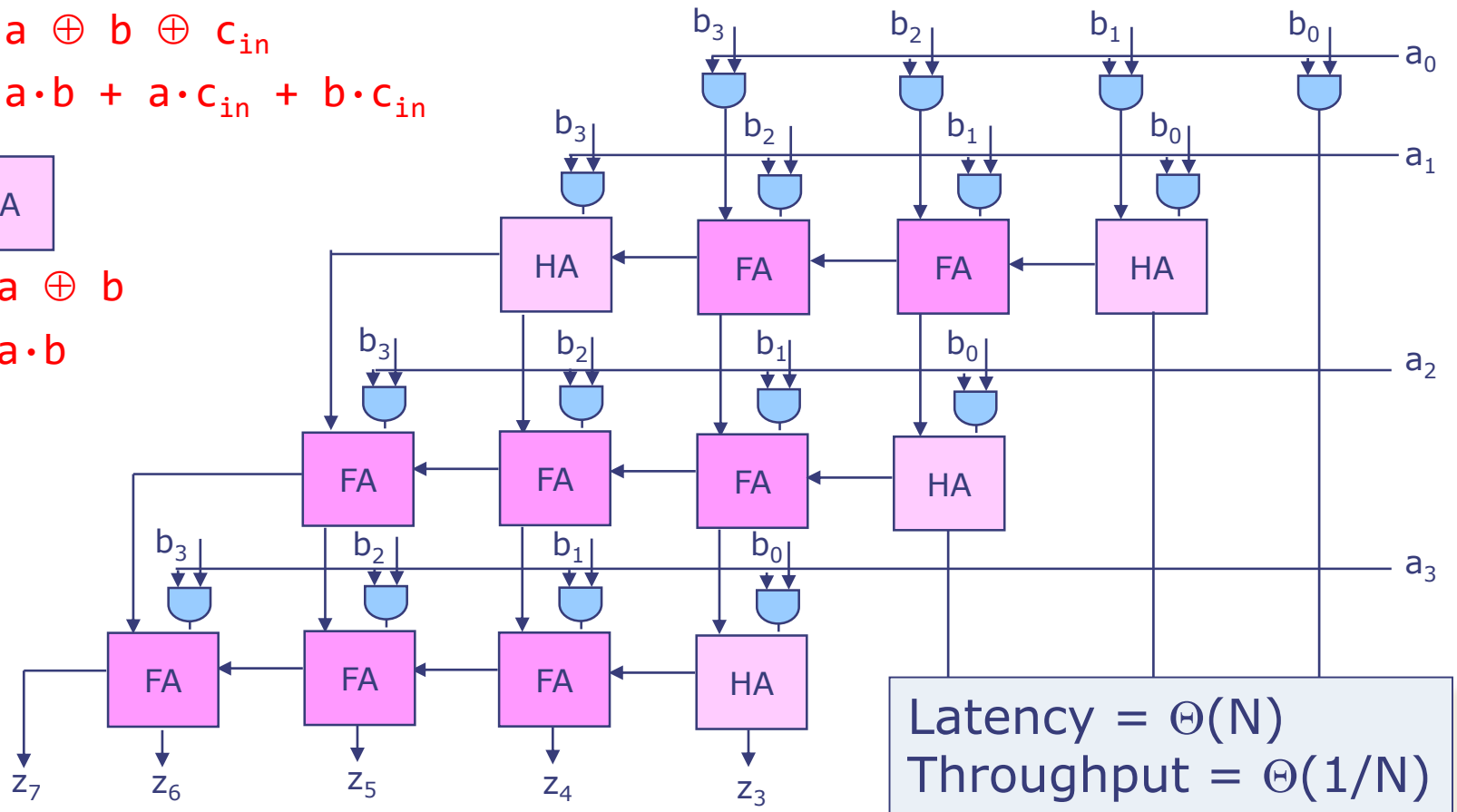
$s = a \oplus b \oplus c_{in}$

$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$

HA

$s = a \oplus b$

$c_{out} = a \cdot b$



Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$

Combinational Multiplier Redrawn

Using ripple-carry adders

FA

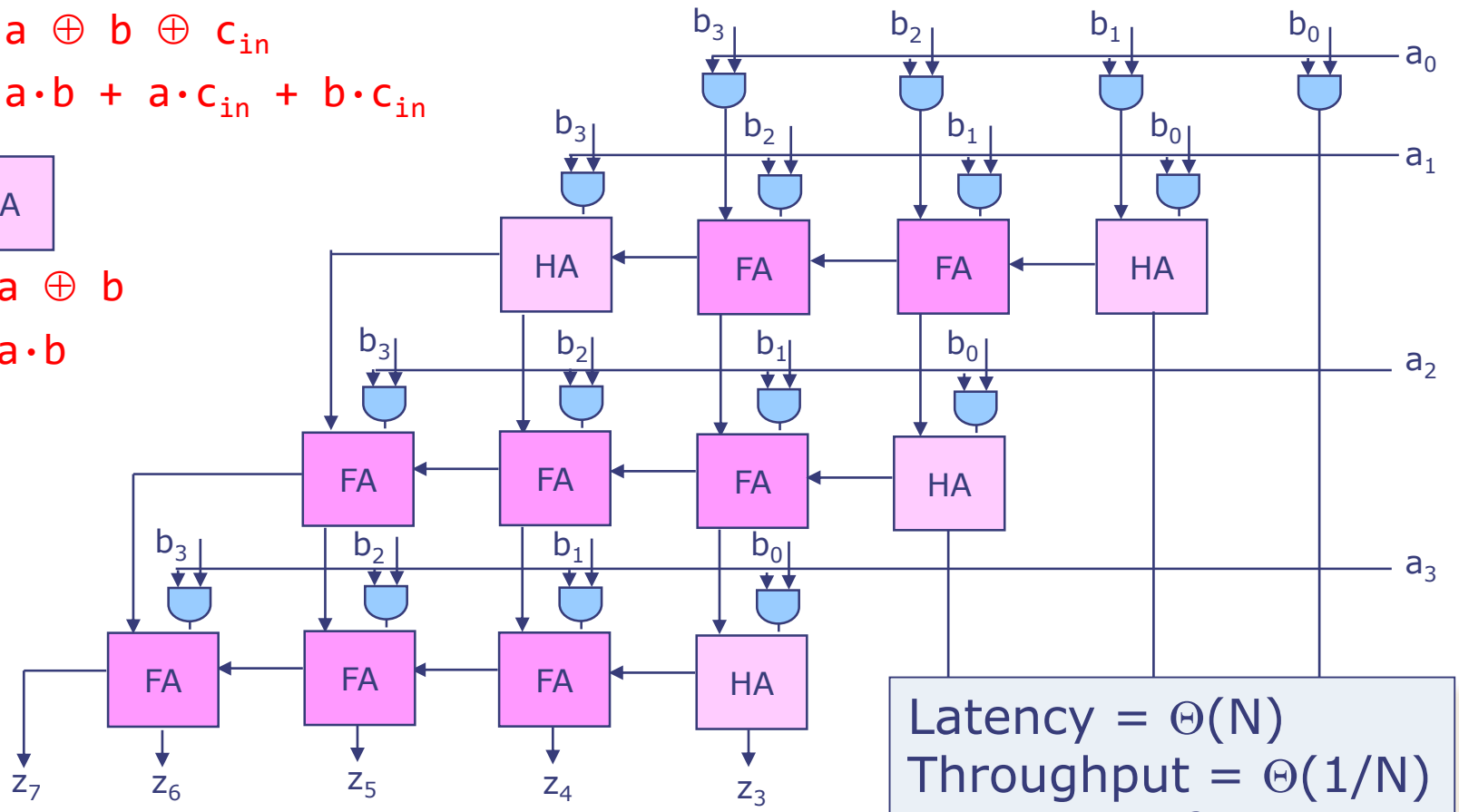
$s = a \oplus b \oplus c_{in}$

$c_{out} = a \cdot b + a \cdot c_{in} + b \cdot c_{in}$

HA

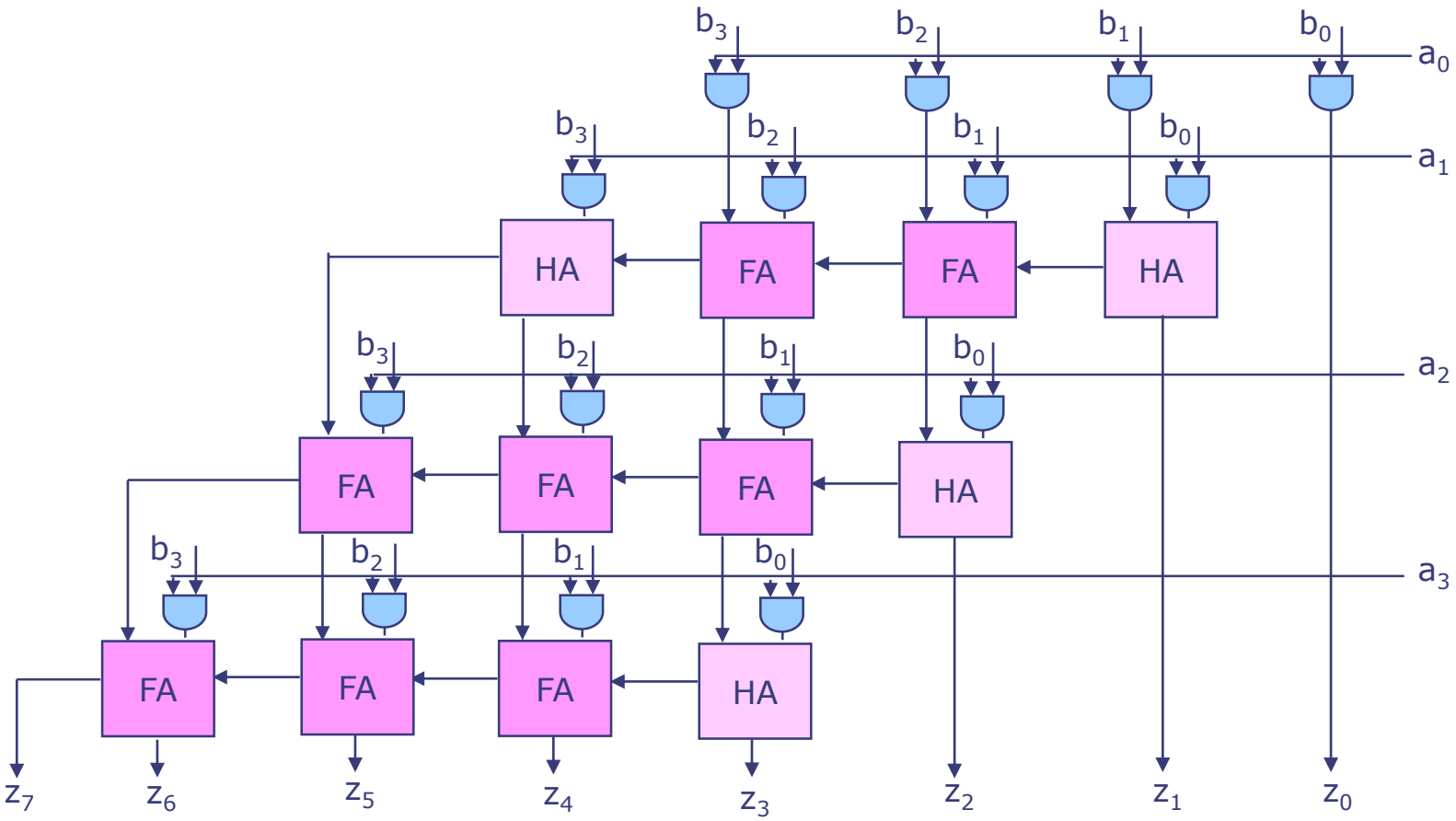
$s = a \oplus b$

$c_{out} = a \cdot b$

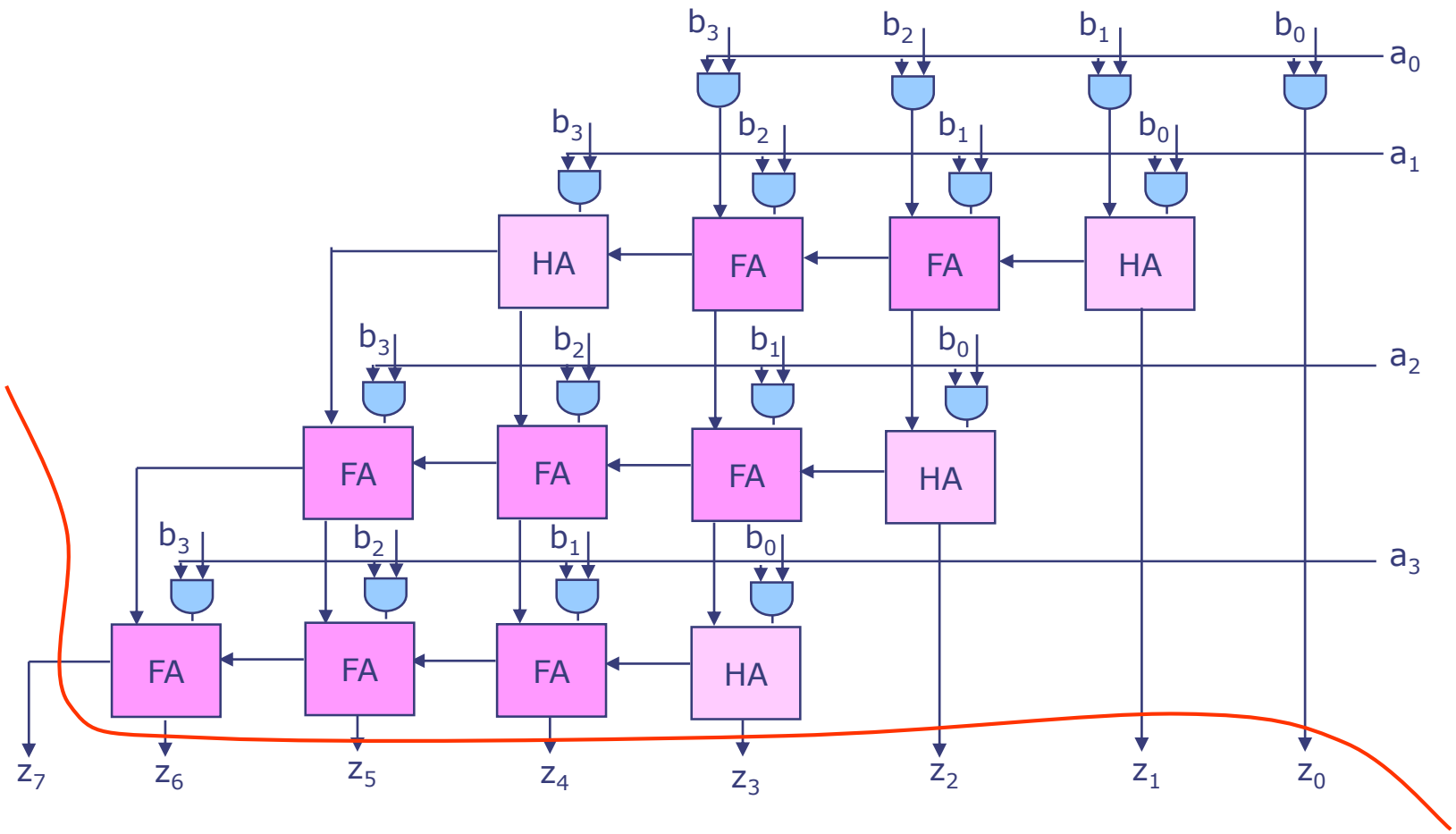


Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Area = $\Theta(N^2)$

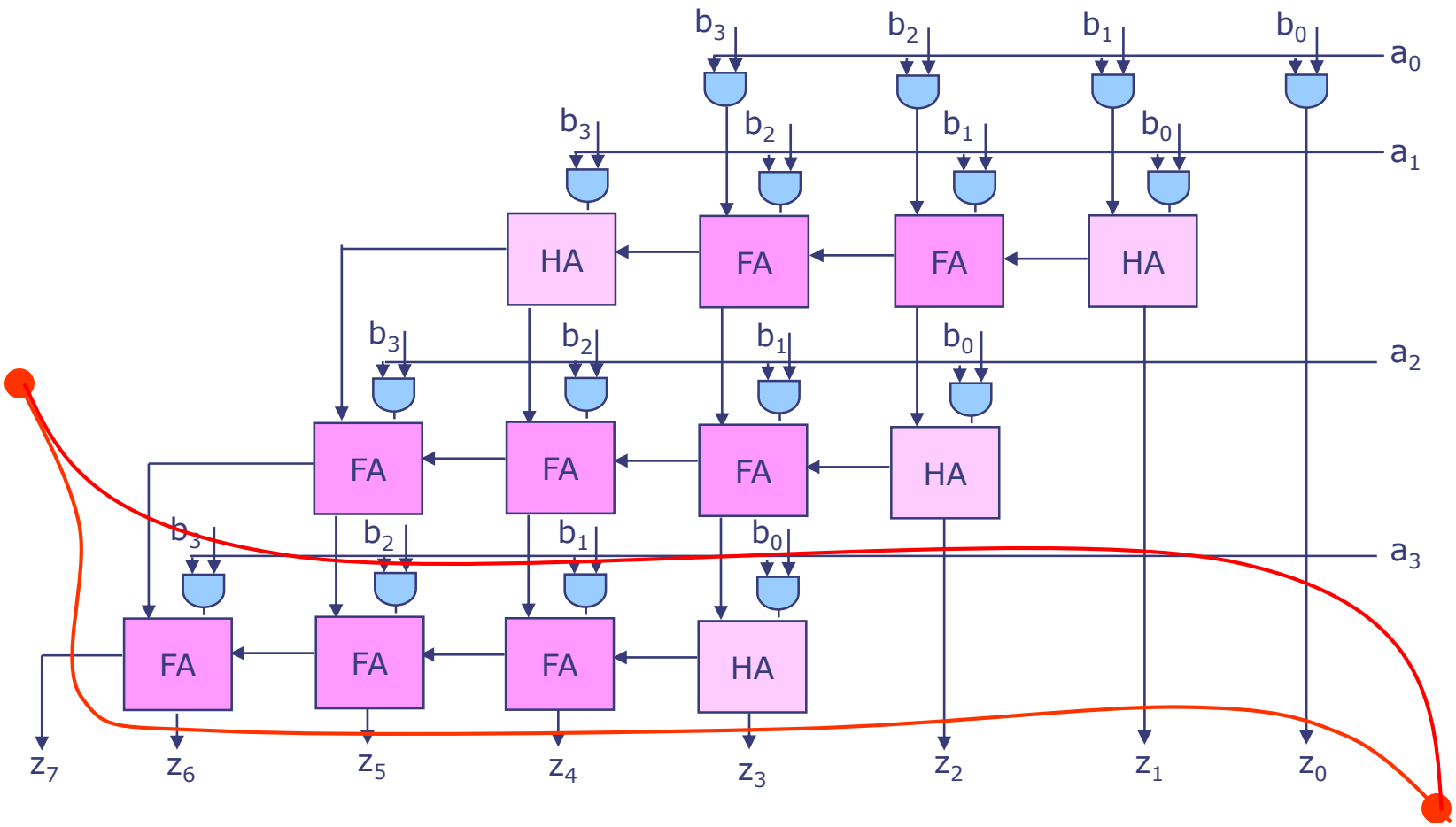
Pipelining to Increase Throughput
First Attempt



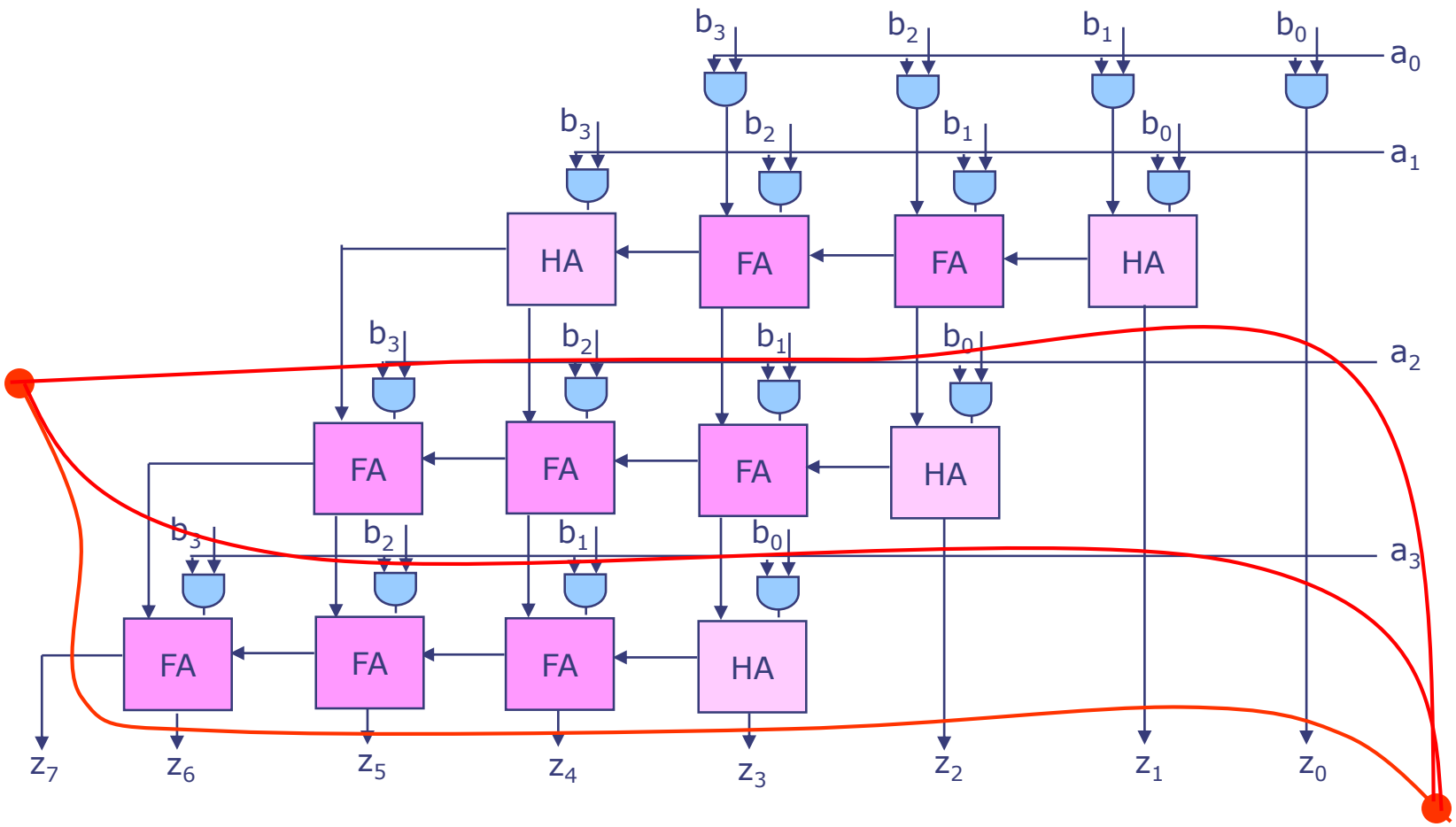
Pipelining to Increase Throughput
First Attempt



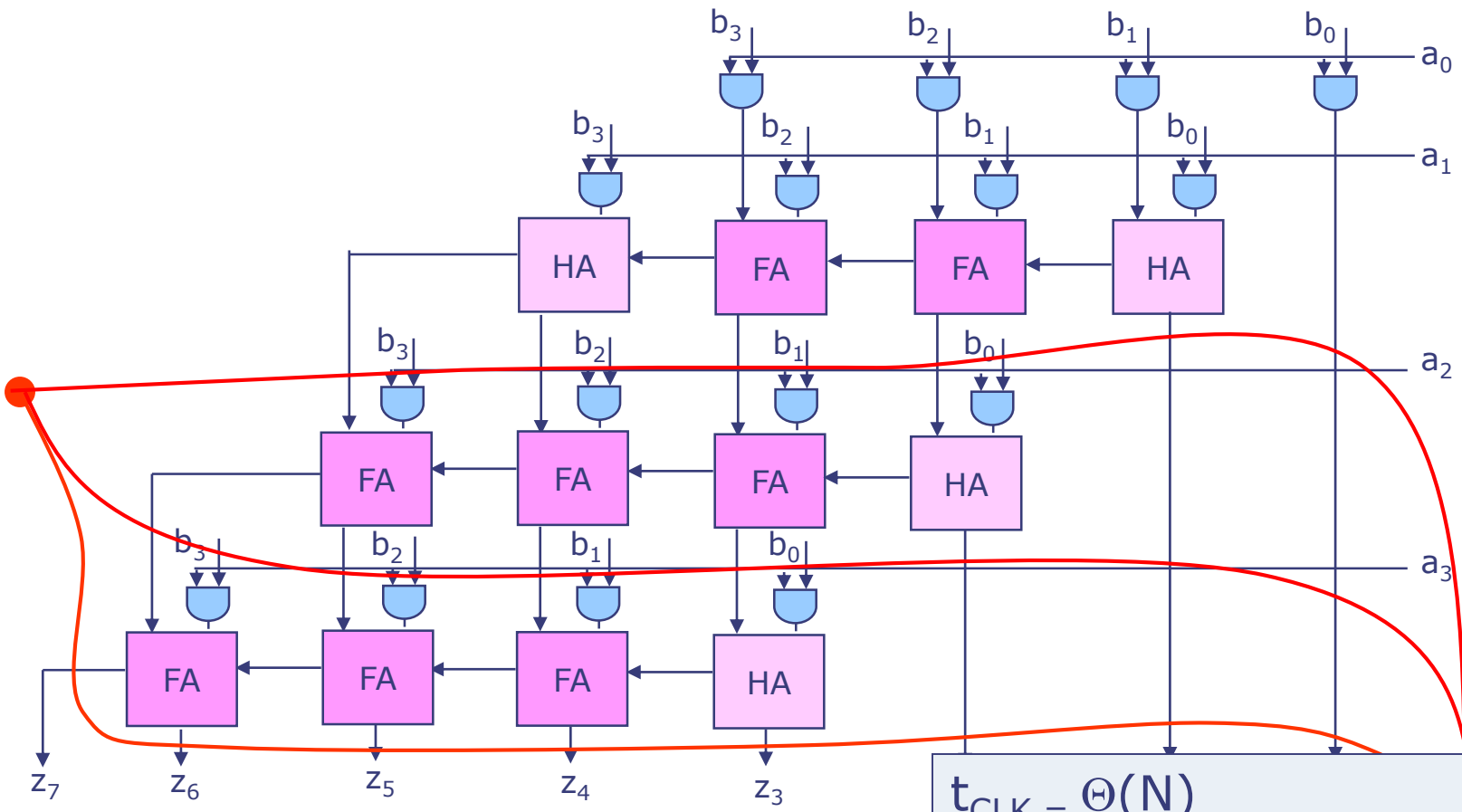
Pipelining to Increase Throughput
First Attempt



Pipelining to Increase Throughput
First Attempt

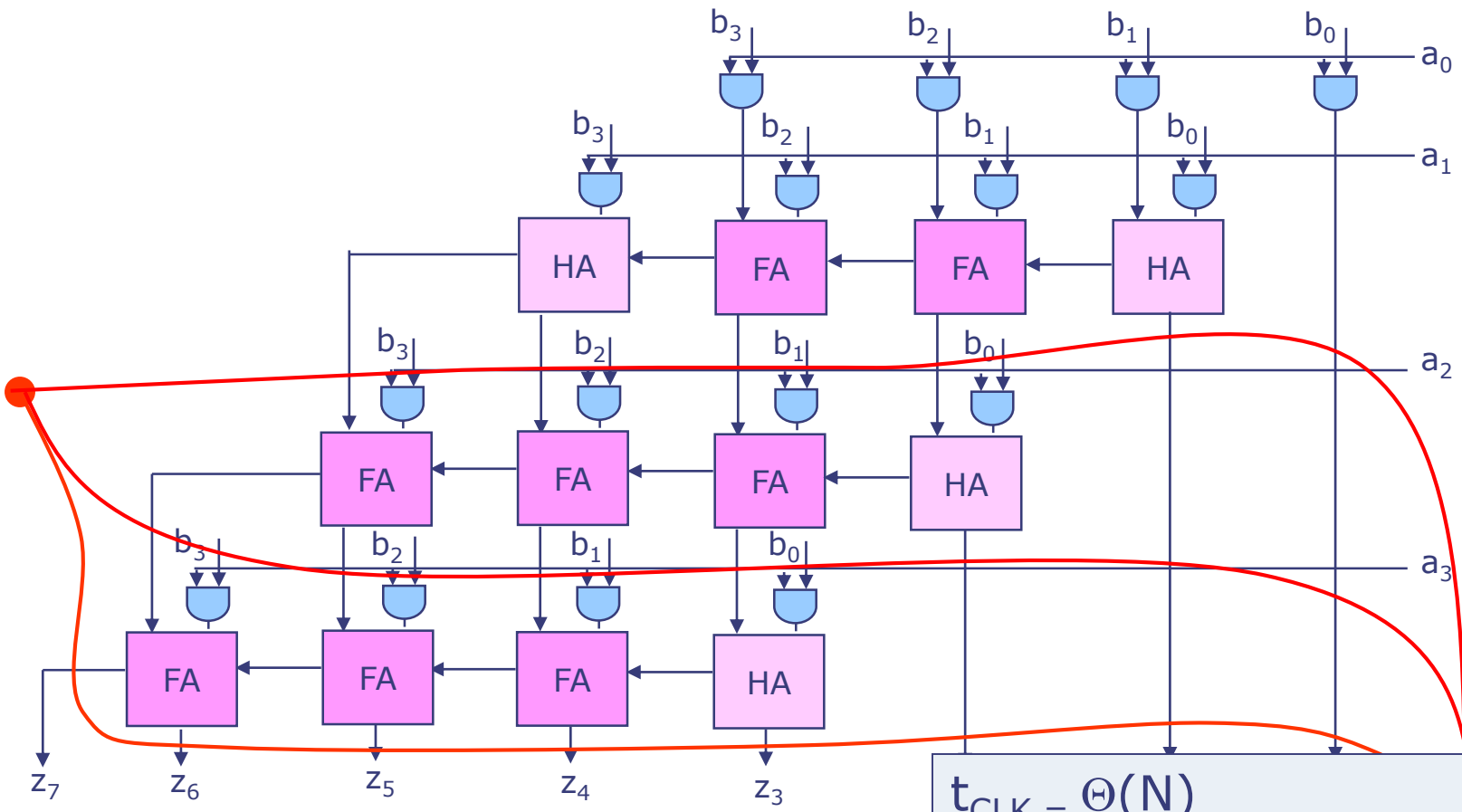


Pipelining to Increase Throughput
First Attempt



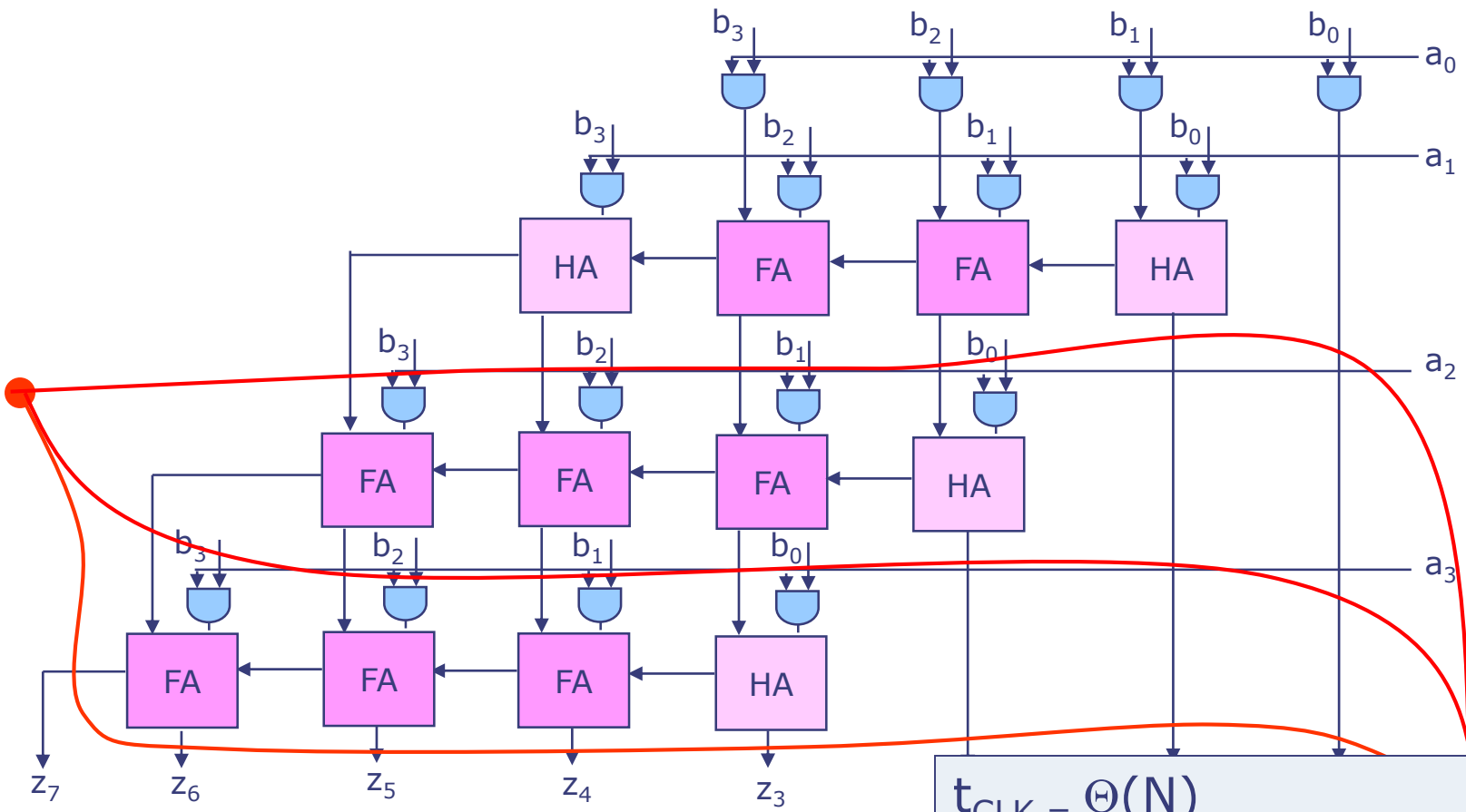
$t_{CLK} = \Theta(N)$

Pipelining to Increase Throughput
First Attempt



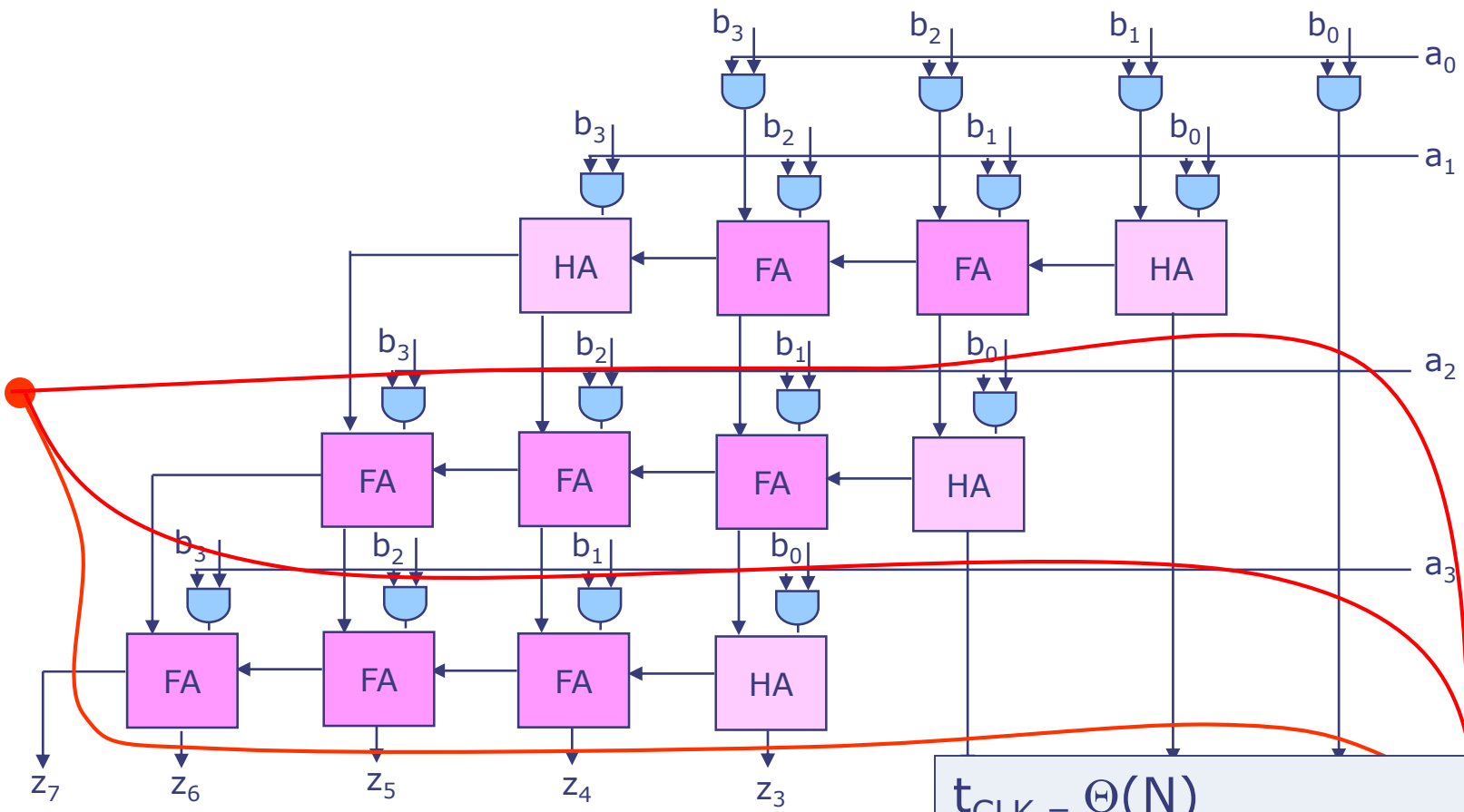
$t_{CLK} = \Theta(N)$
stages = $\Theta(N)$

Pipelining to Increase Throughput
First Attempt



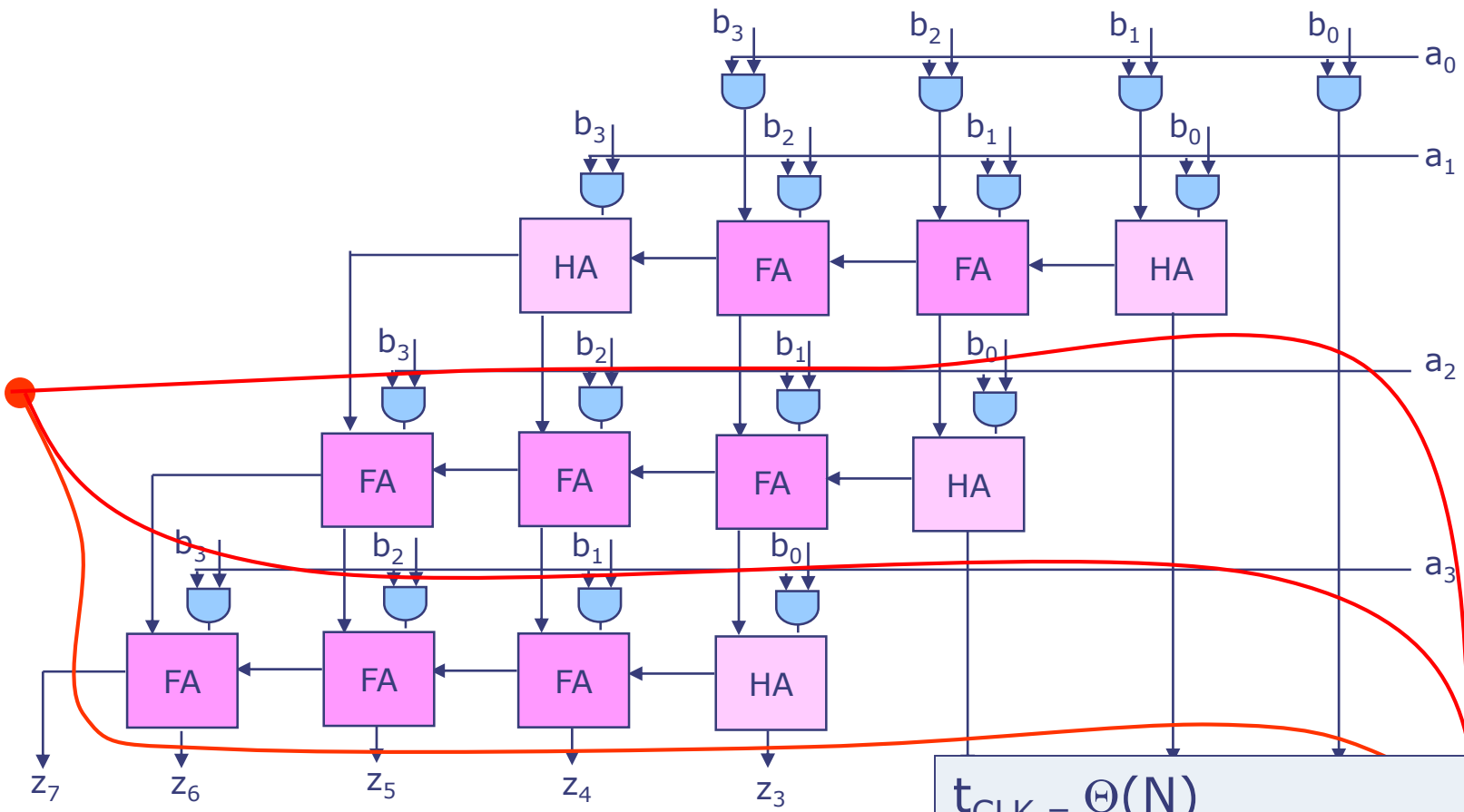
$t_{CLK} = \Theta(N)$
# stages = $\Theta(N)$
Latency = $\Theta(N^2)$

Pipelining to Increase Throughput
First Attempt



$t_{CLK} = \Theta(N)$
# stages = $\Theta(N)$
Latency = $\Theta(N^2)$
Throughput = $\Theta(1/N)$

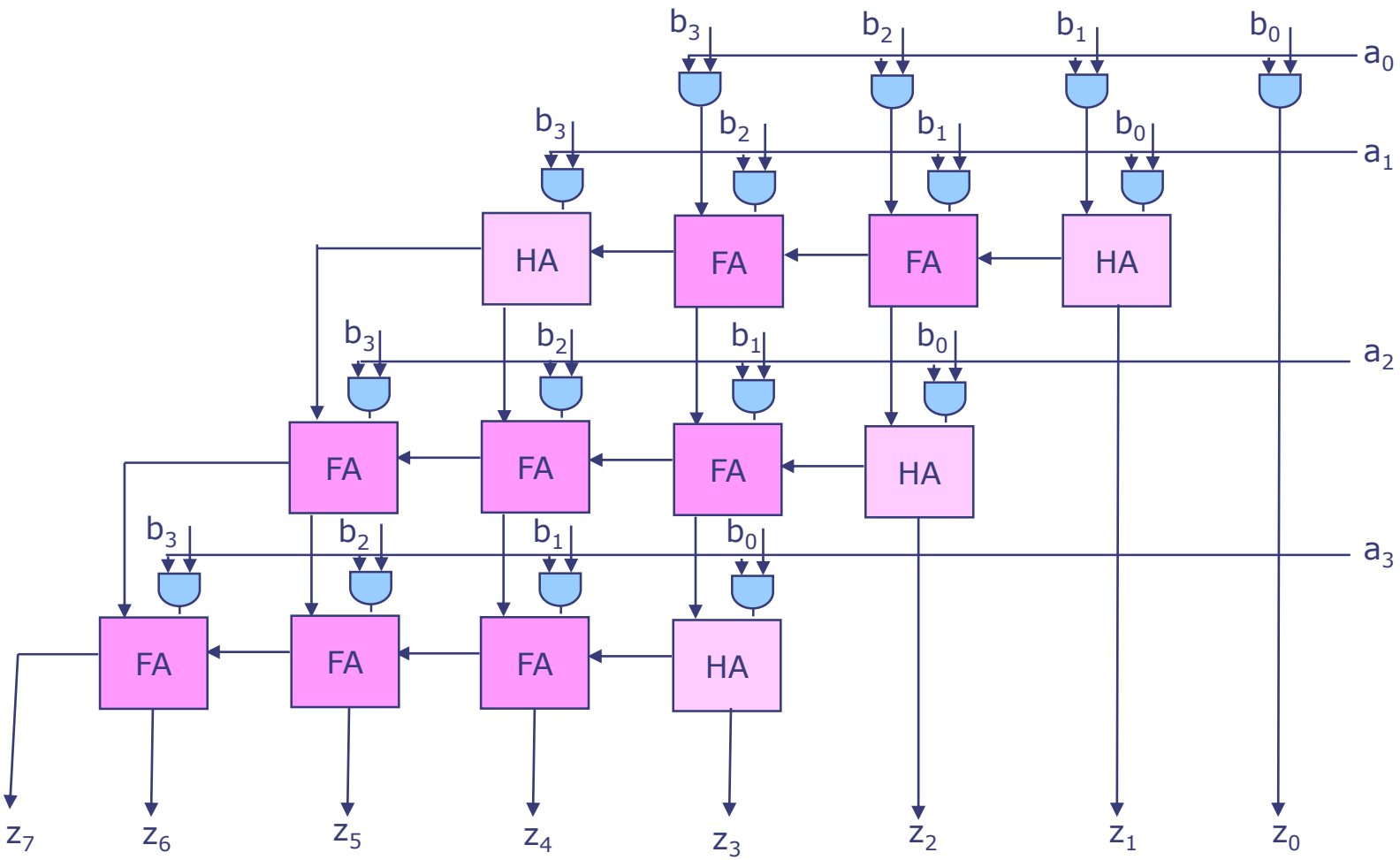
Pipelining to Increase Throughput
First Attempt



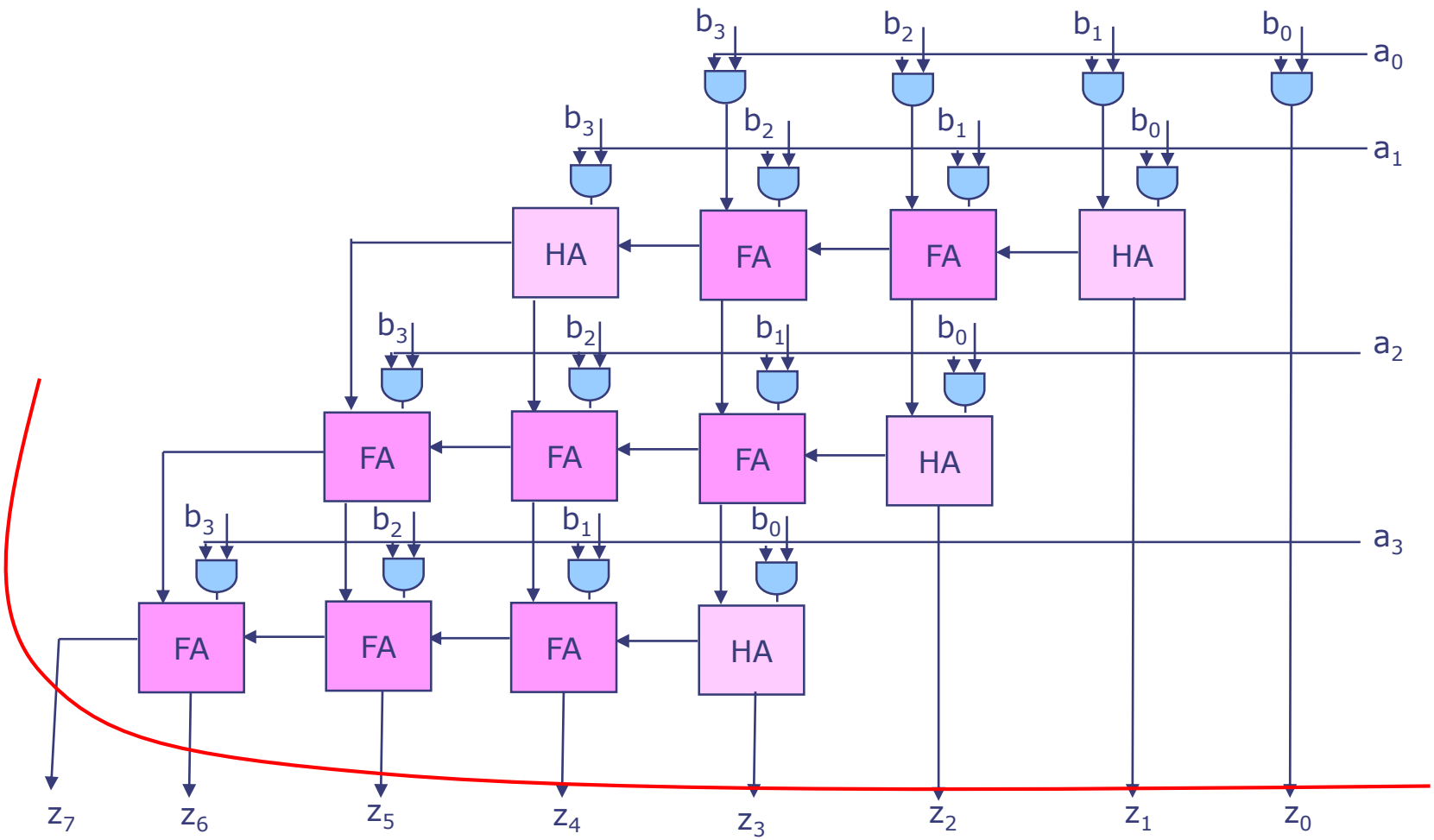
$t_{CLK} = \Theta(N)$
# stages = $\Theta(N)$
Latency = $\Theta(N^2)$
Throughput = $\Theta(1/N)$

Need to break carry chain

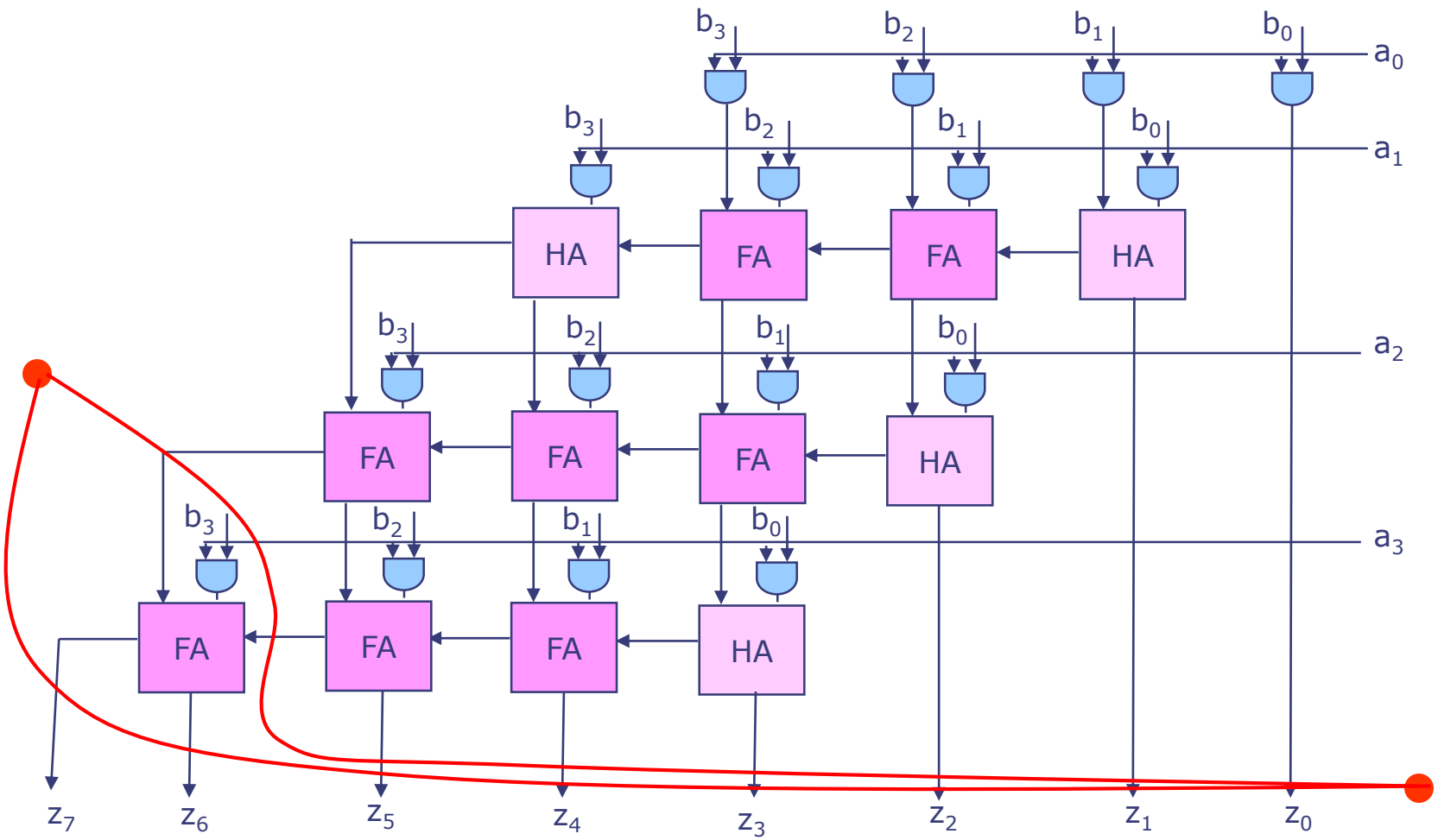
Pipelining to Increase Throughput



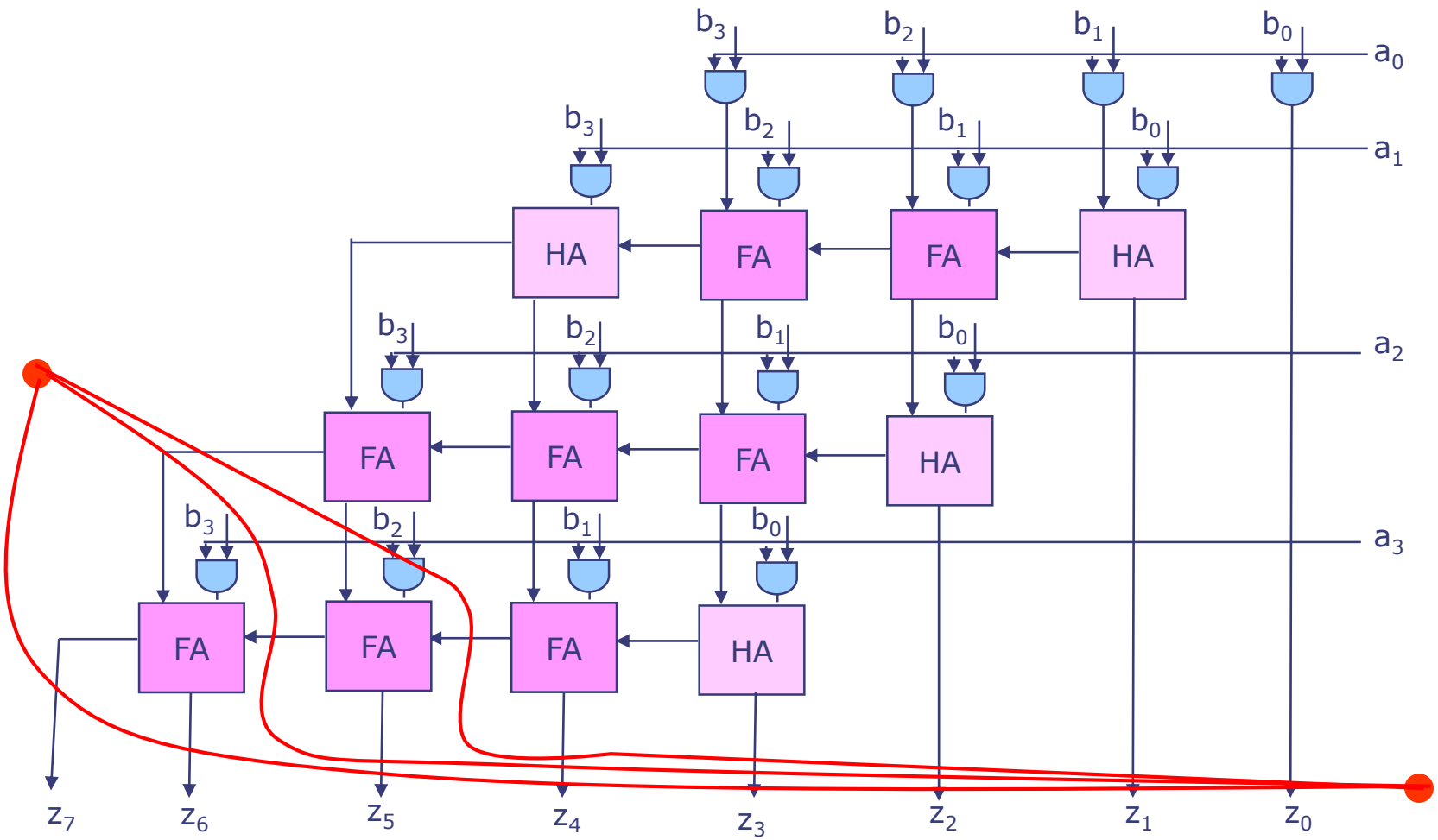
Pipelining to Increase Throughput



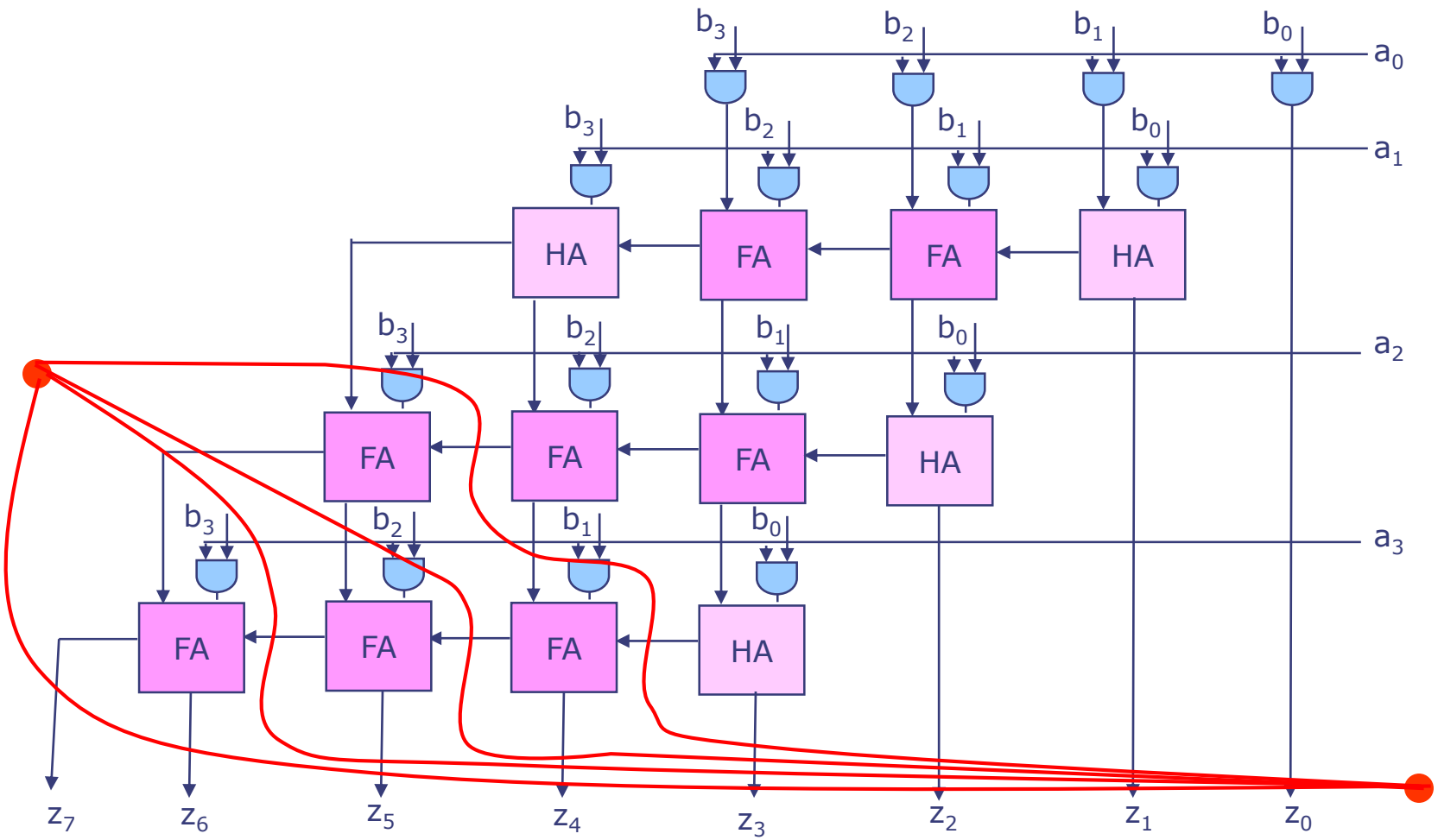
Pipelining to Increase Throughput



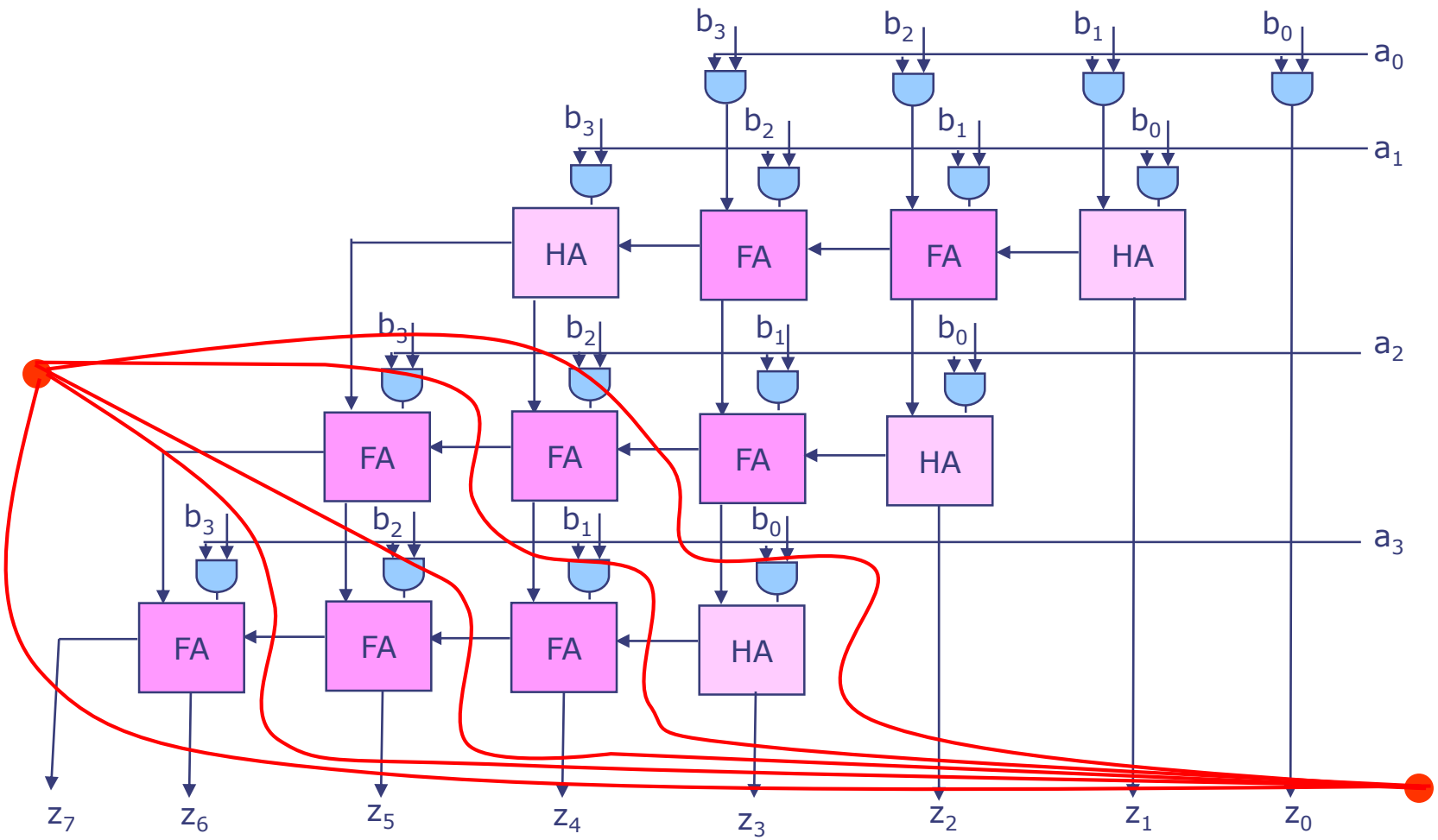
Pipelining to Increase Throughput



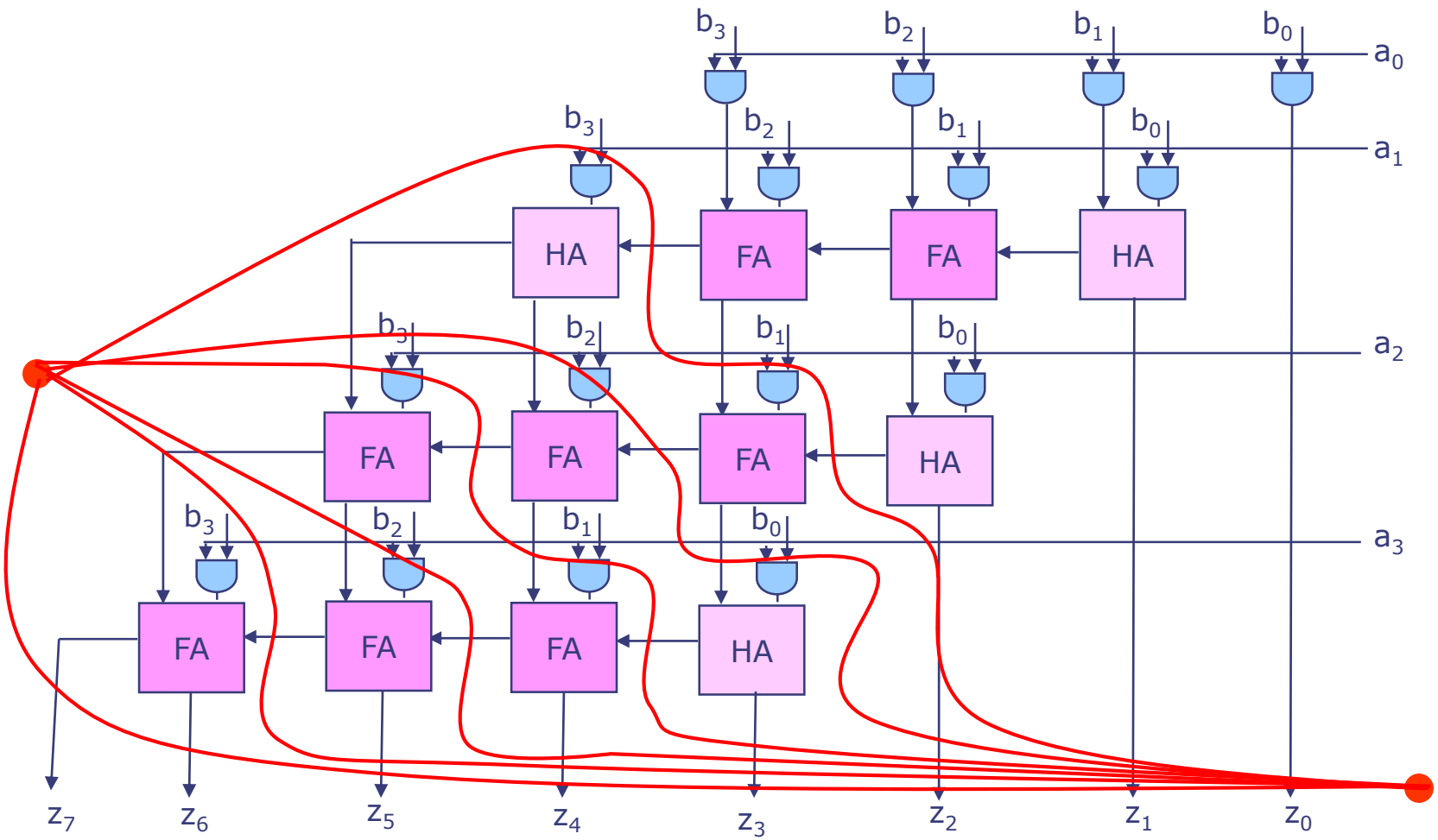
Pipelining to Increase Throughput



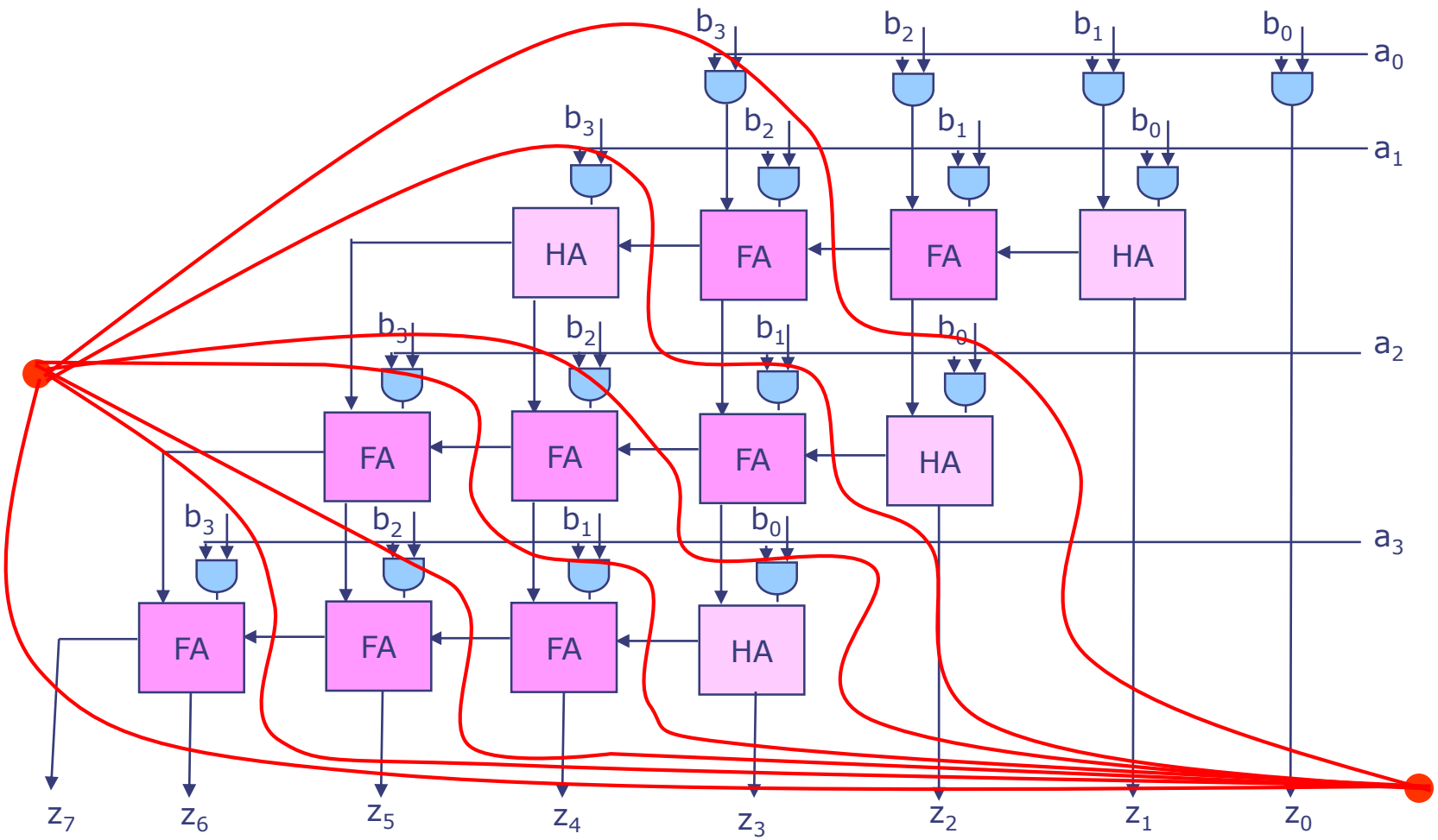
Pipelining to Increase Throughput



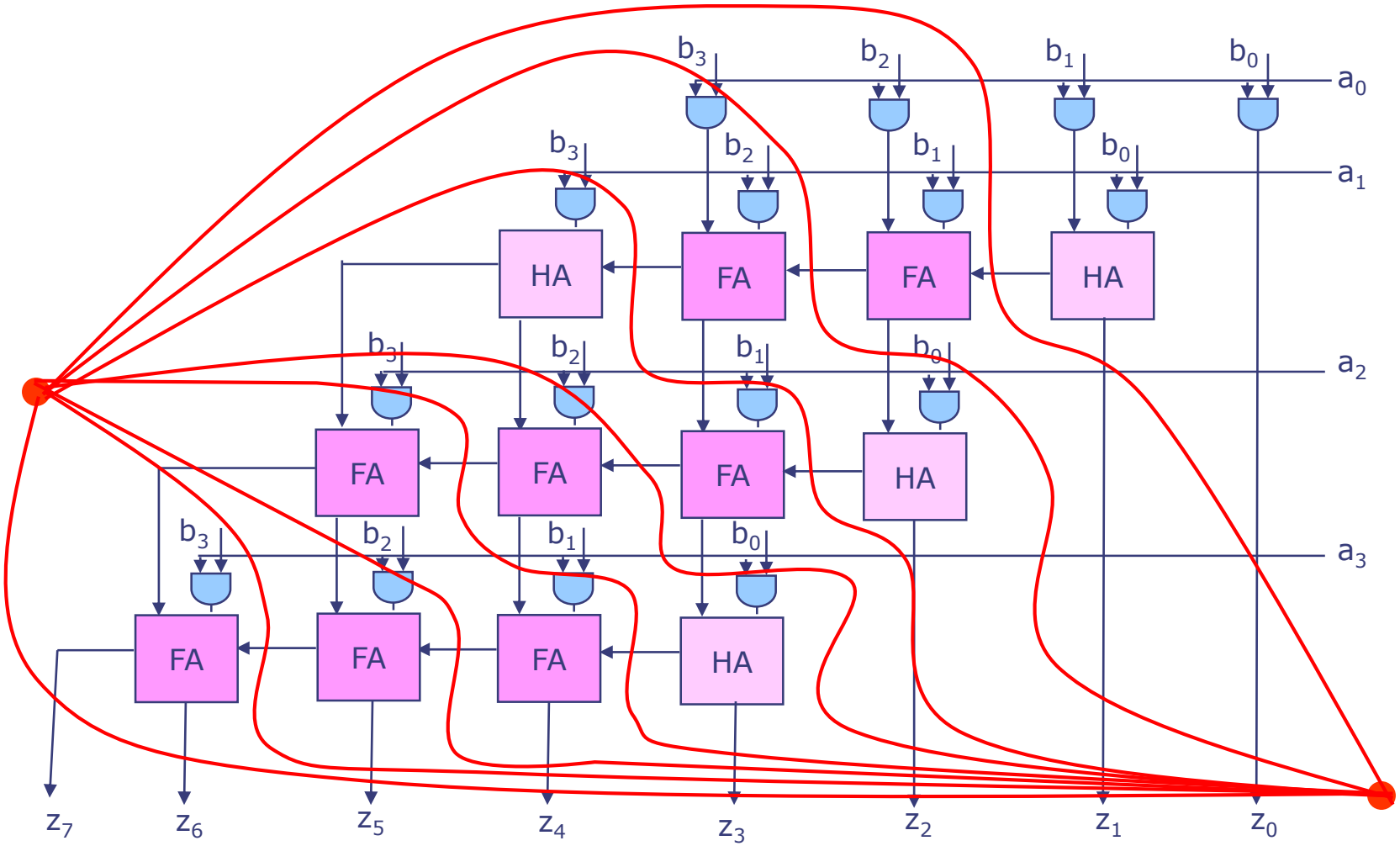
Pipelining to Increase Throughput



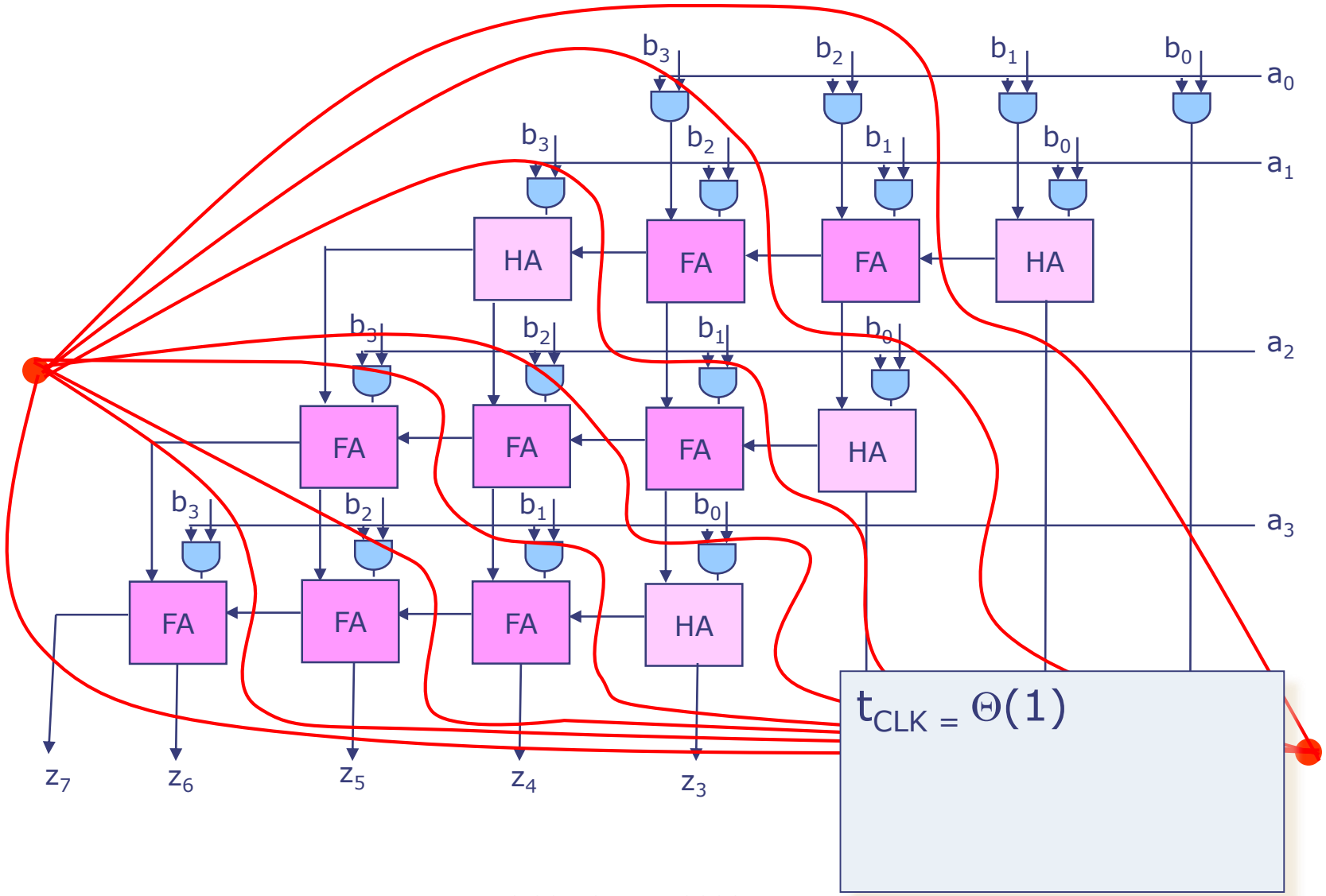
Pipelining to Increase Throughput



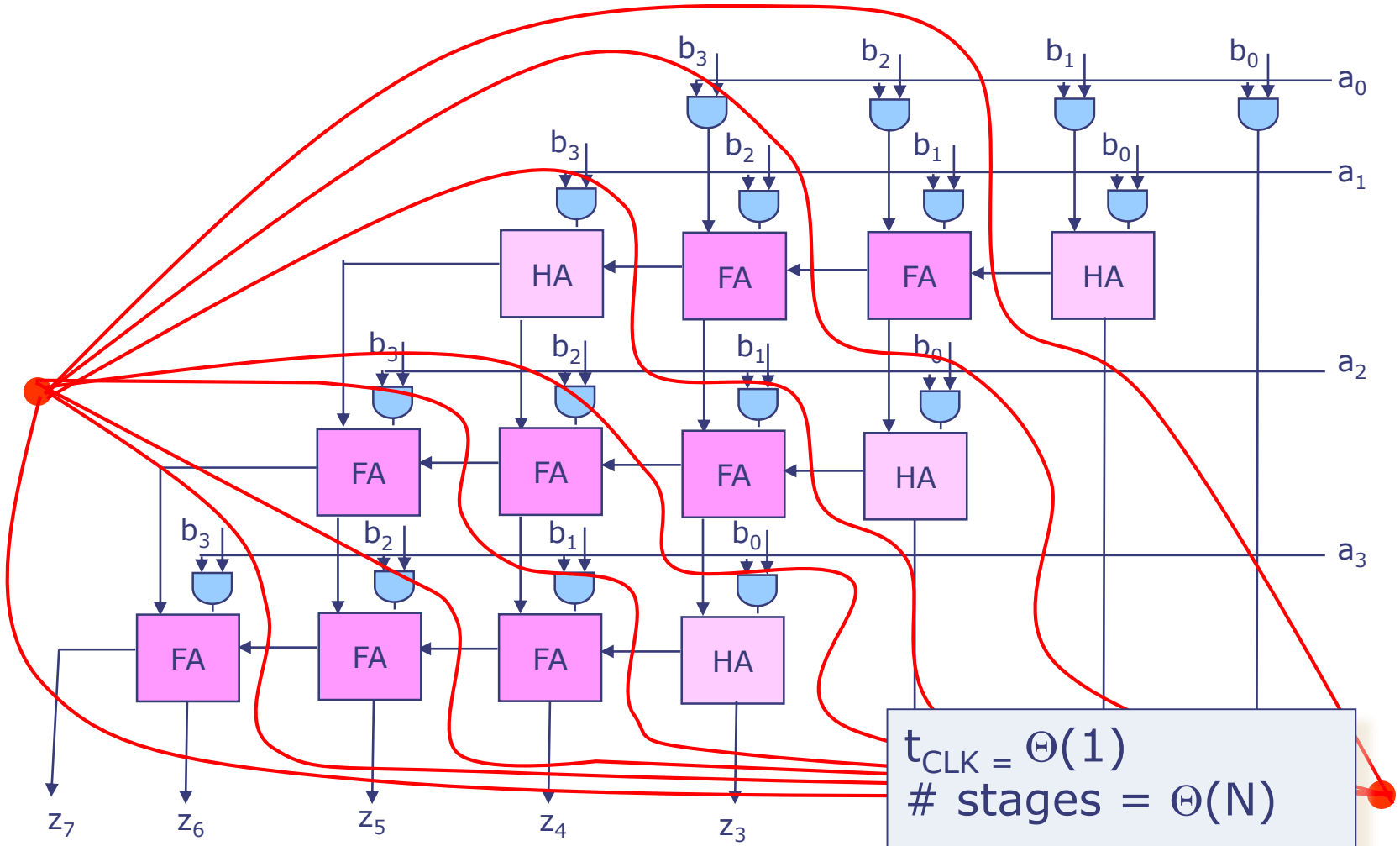
Pipelining to Increase Throughput



Pipelining to Increase Throughput



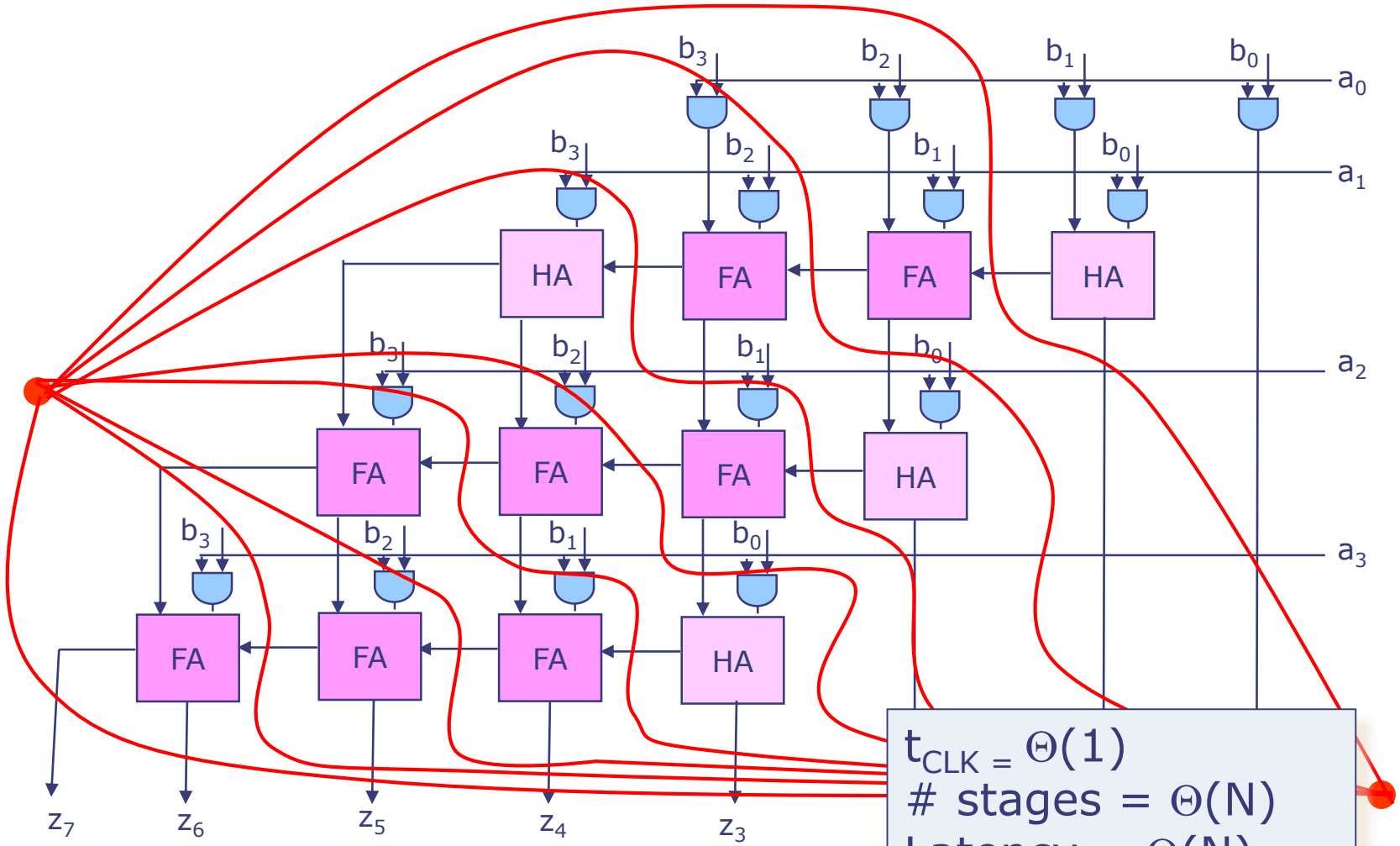
Pipelining to Increase Throughput



$t_{CLK} = \Theta(1)$

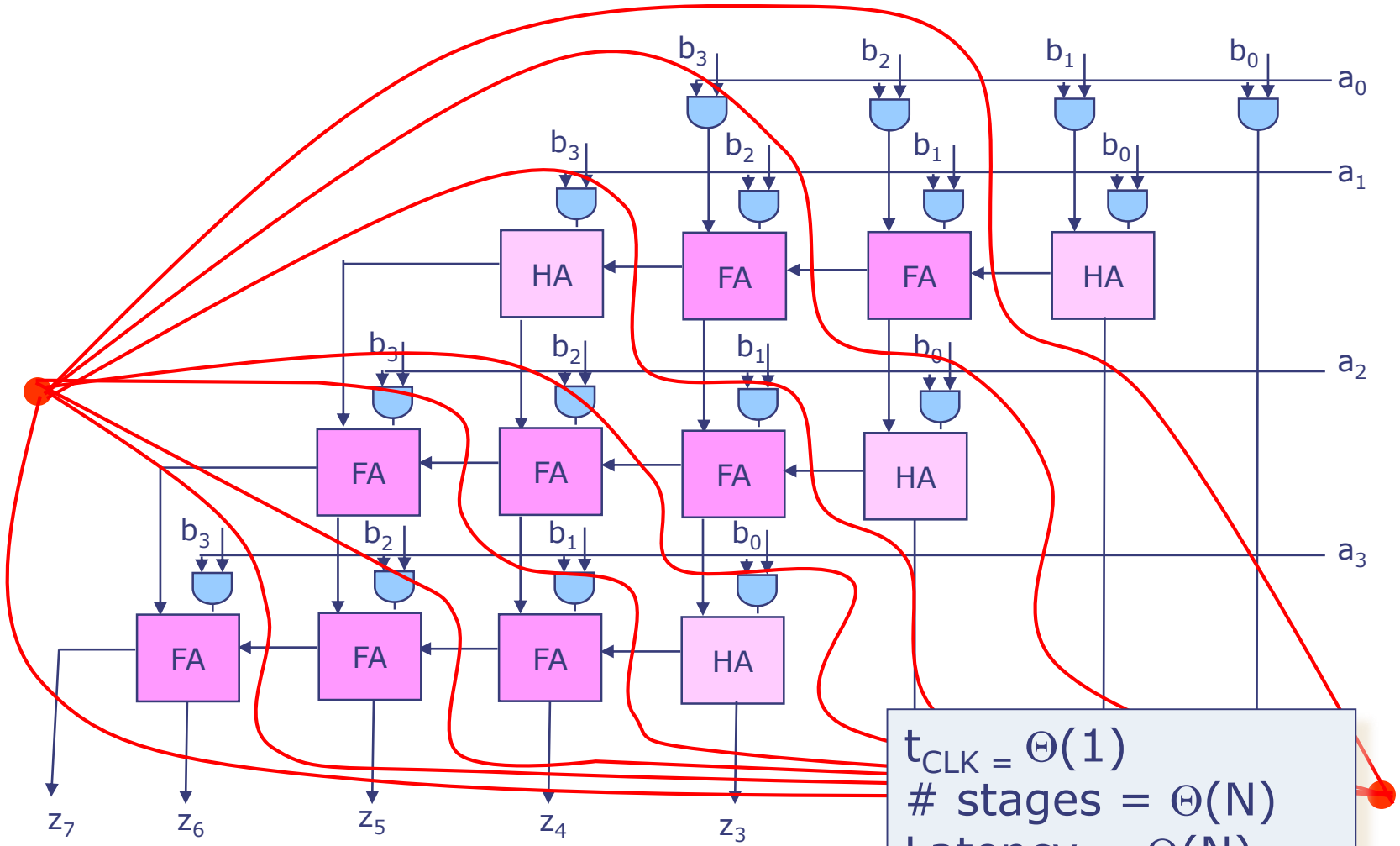
stages = $\Theta(N)$

Pipelining to Increase Throughput



$t_{CLK} = \Theta(1)$
# stages = $\Theta(N)$
Latency = $\Theta(N)$

Pipelining to Increase Throughput



$t_{CLK} = \Theta(1)$
# stages = $\Theta(N)$
Latency = $\Theta(N)$
Throughput = $\Theta(1)$

Folded (Multi-Cycle) Multiplier

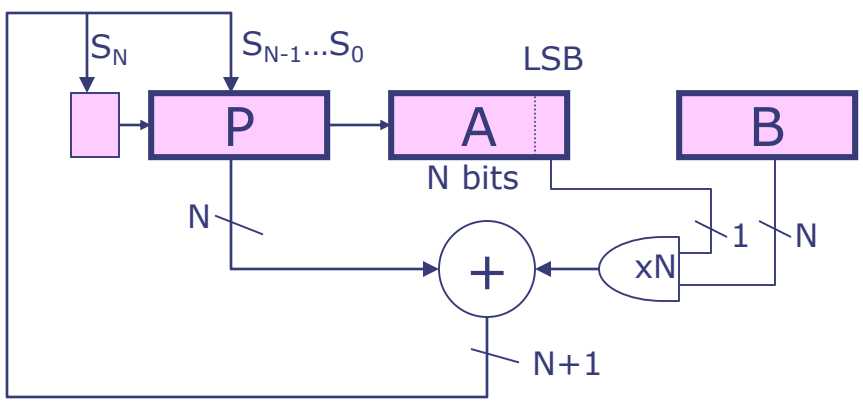
- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders

Folded (Multi-Cycle) Multiplier

- 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 ~N cycles to perform the additions

Folded (Multi-Cycle) Multiplier

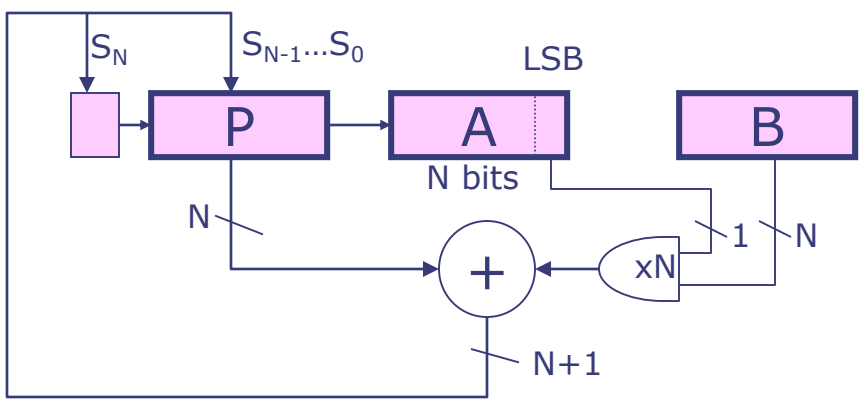
- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
- Folded circuits use less combinational logic, reuse it over multiple cycles
  - Example: Implement multiplication with one adder, taking ~N cycles to perform the additions




```
Init: P←0, load A&B

Repeat N times {
   P ← P + (A_LSB==1 ? B : 0)
   shift S_N,P,A right one bit
}

Done: 2N-bit result in P,A
```

Folded (Multi-Cycle) Multiplier

- 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 ~N cycles to perform the additions

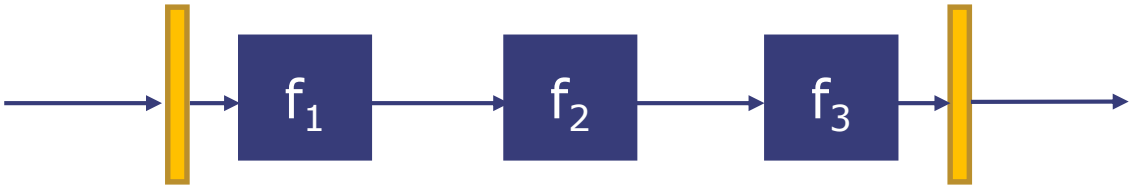



```
Init: P←0, load A&B

Repeat N times {
   P ← P + (A_LSB==1 ? B : 0)
   shift S_N,P,A right one bit
}

Done: 2N-bit result in P,A
```

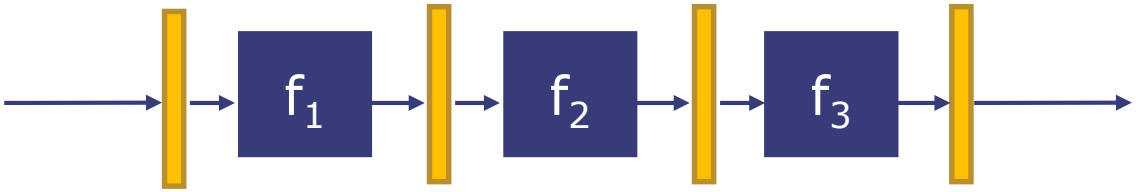
Tradeoff: reduced area, but lower throughput

Summary: Design Alternatives

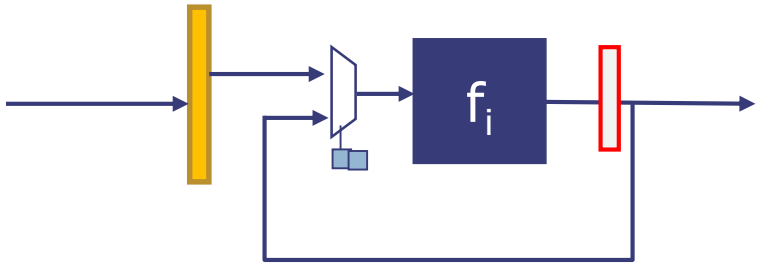
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

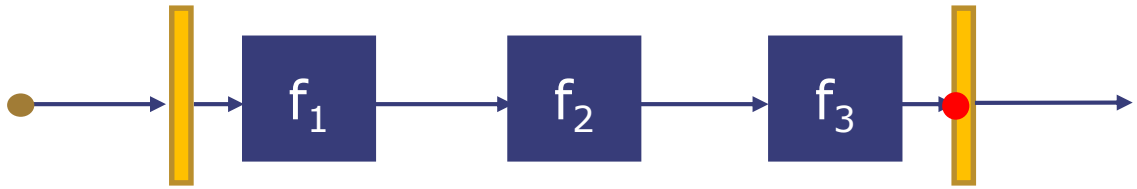


Folded reuse a block, multi-cycle (C)

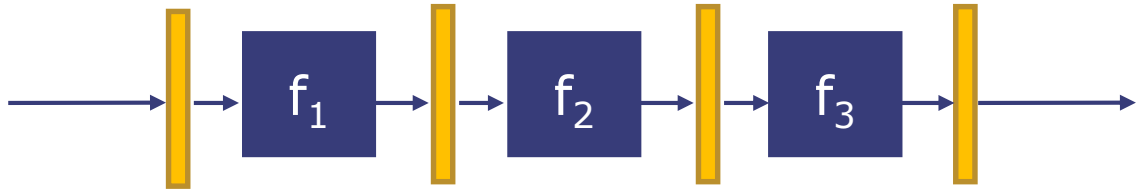


Summary: Design Alternatives

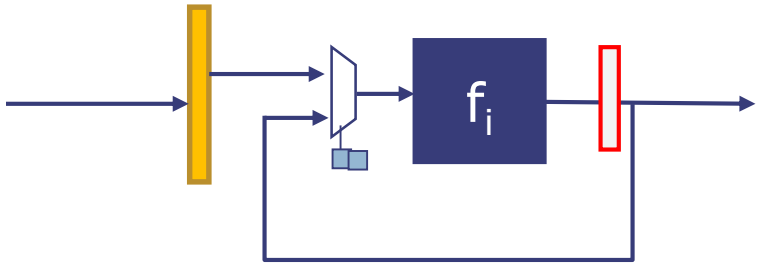
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

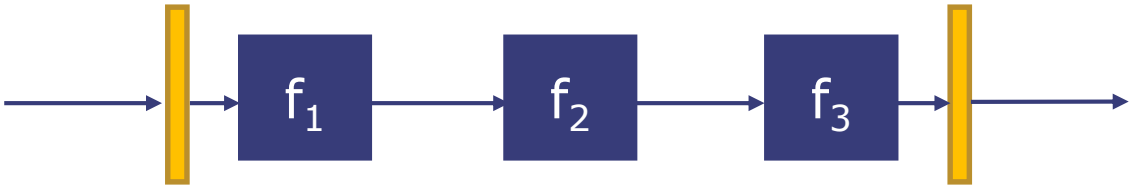


Folded reuse a block, multi-cycle (C)

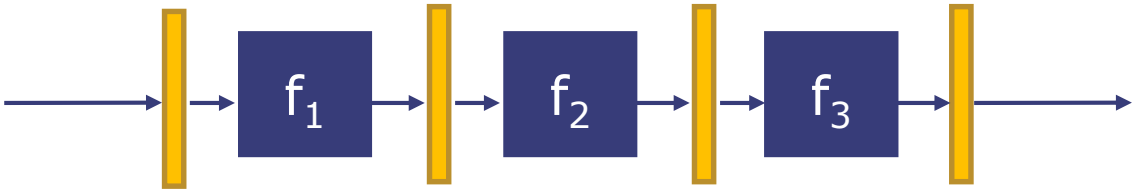


Summary: Design Alternatives

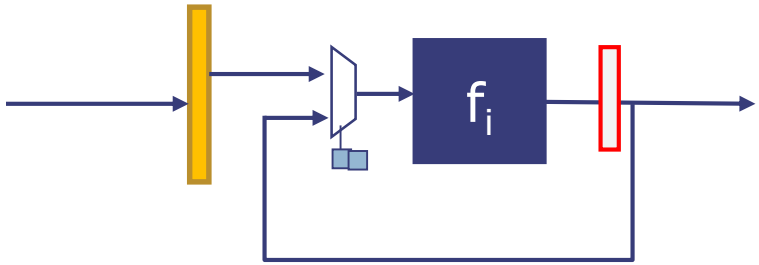
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

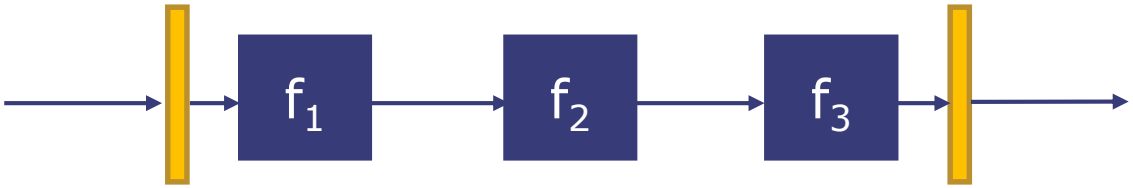


Folded reuse a block, multi-cycle (C)

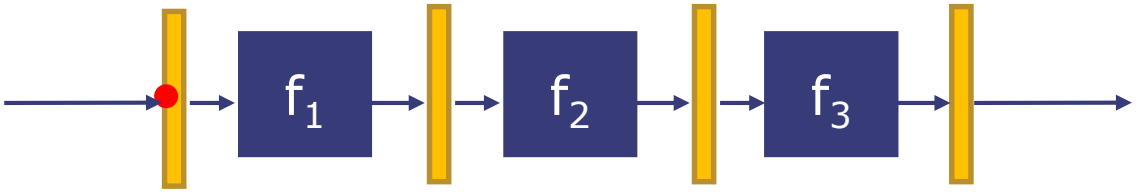


Summary: Design Alternatives

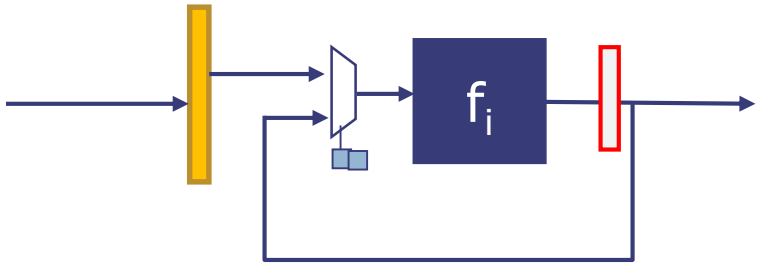
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

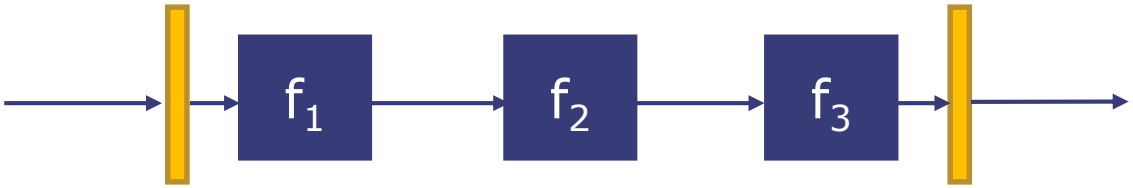


Folded reuse a block, multi-cycle (C)

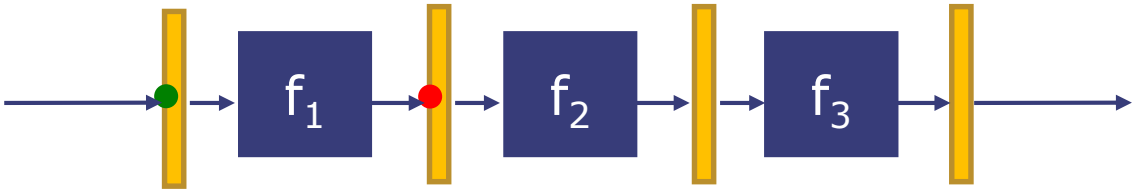


Summary: Design Alternatives

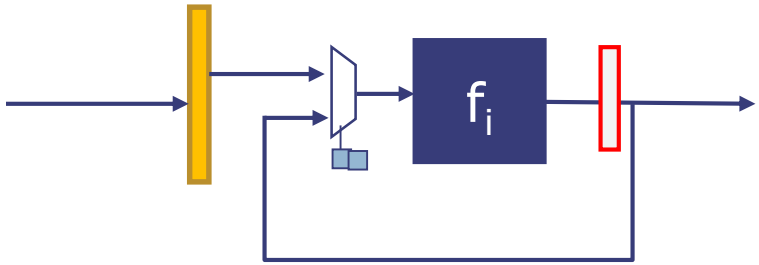
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

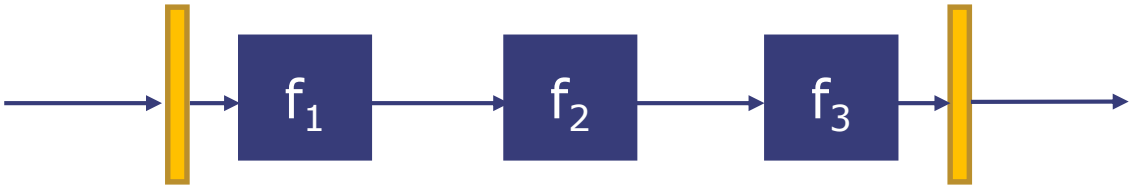


Folded reuse a block, multi-cycle (C)

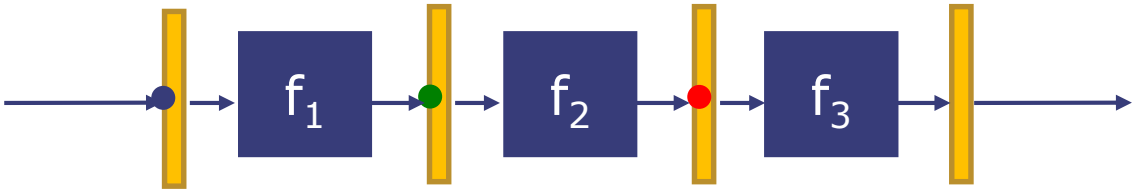


Summary: Design Alternatives

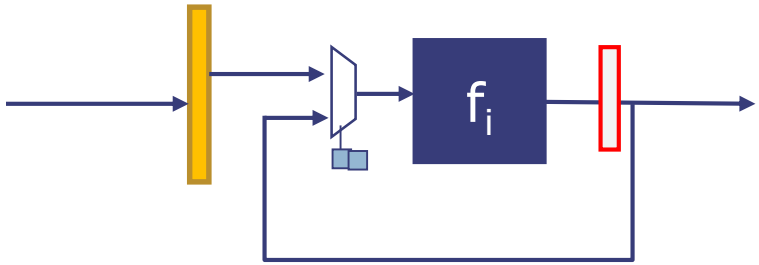
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

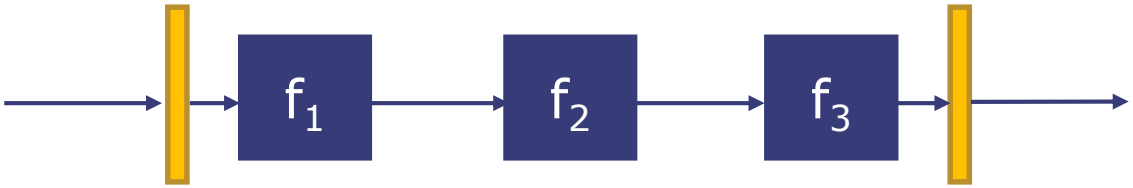


Folded reuse a block, multi-cycle (C)

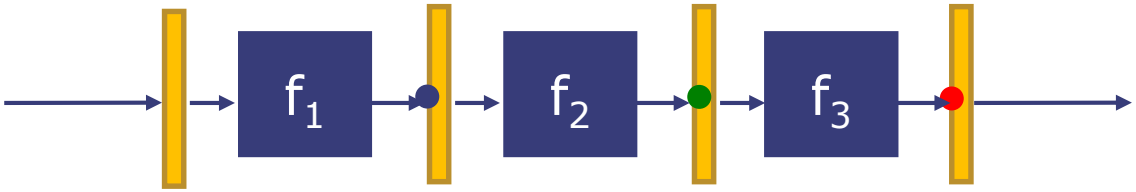


Summary: Design Alternatives

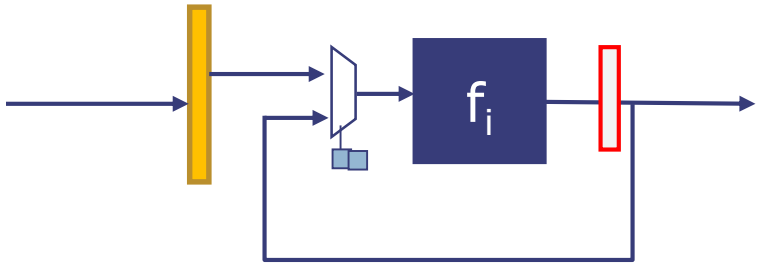
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

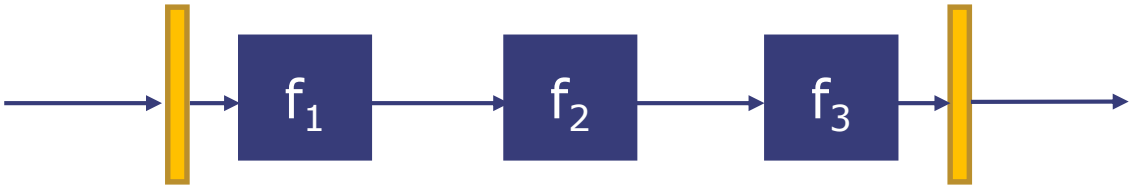


Folded reuse a block, multi-cycle (C)

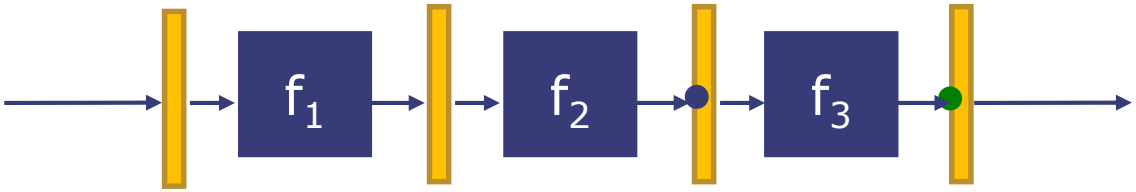


Summary: Design Alternatives

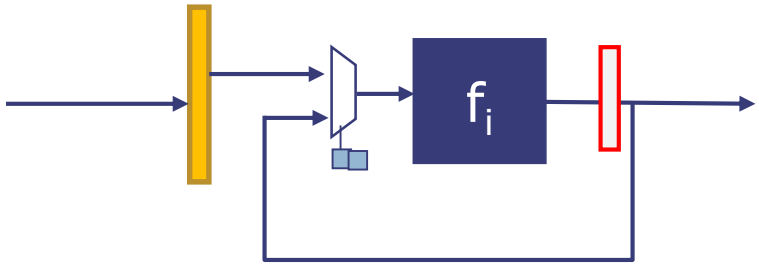
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

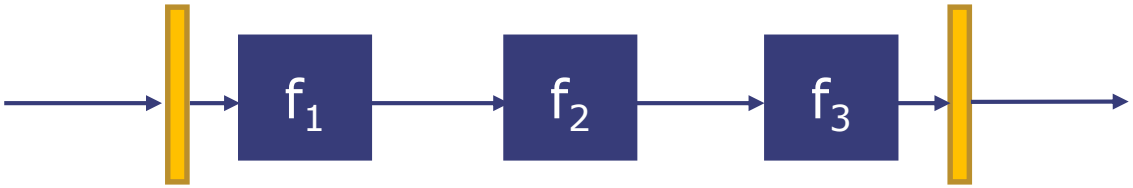


Folded reuse a block, multi-cycle (C)

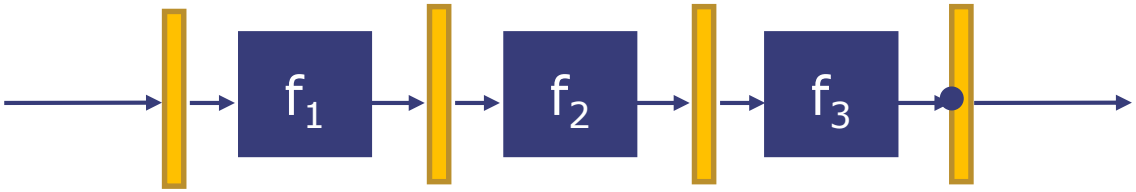


Summary: Design Alternatives

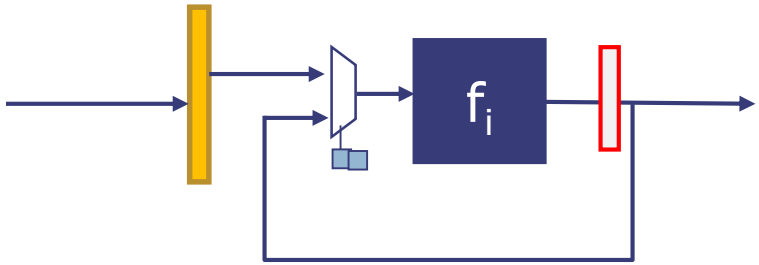
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

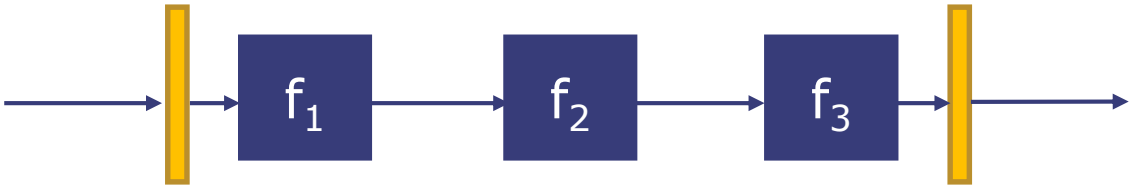


Folded reuse a block, multi-cycle (C)

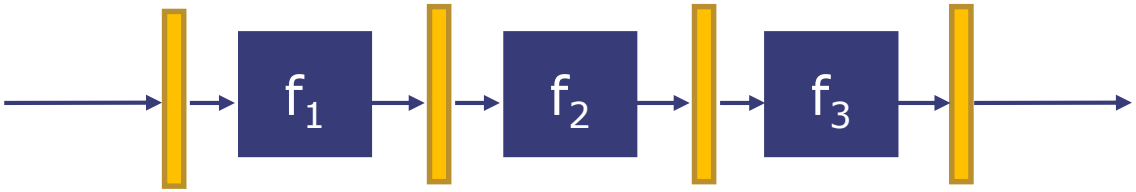


Summary: Design Alternatives

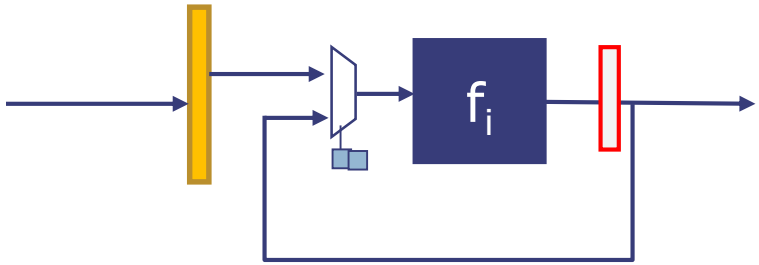
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

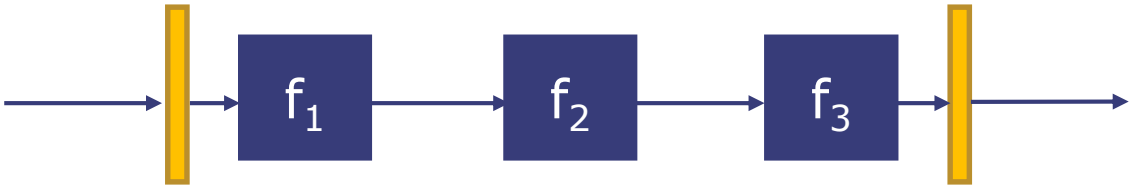


Folded reuse a block, multi-cycle (C)

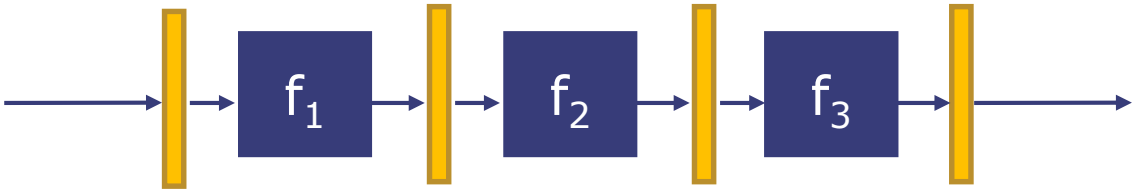


Summary: Design Alternatives

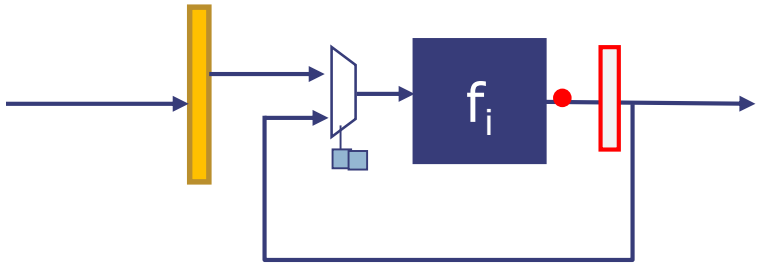
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

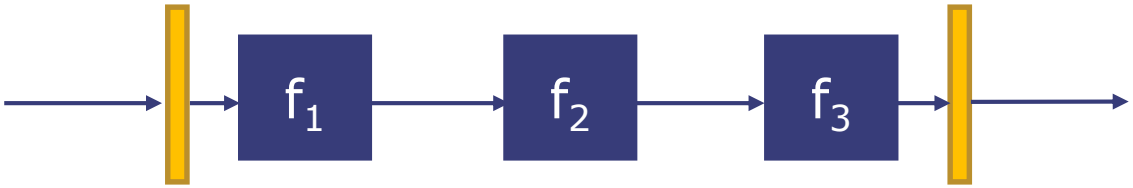


Folded reuse a block, multi-cycle (C)

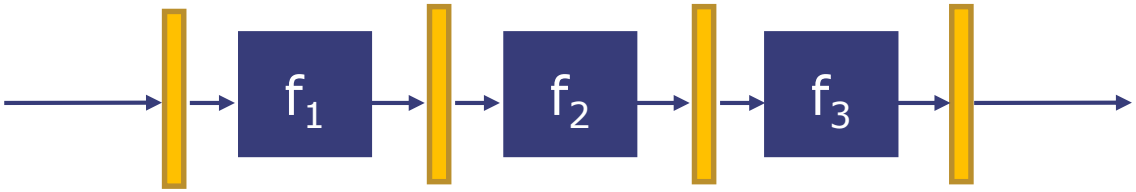


Summary: Design Alternatives

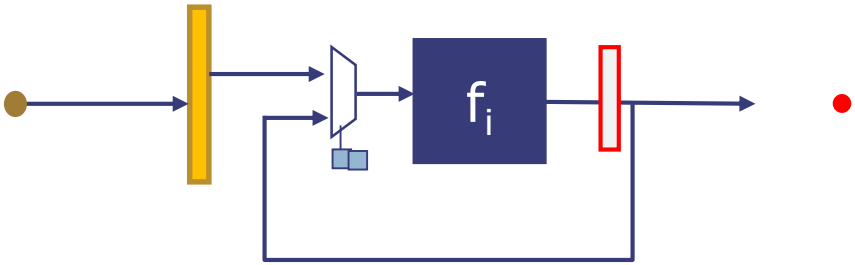
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

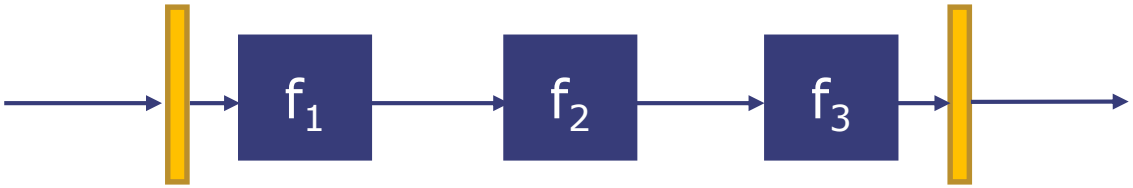


Folded reuse a block, multi-cycle (C)

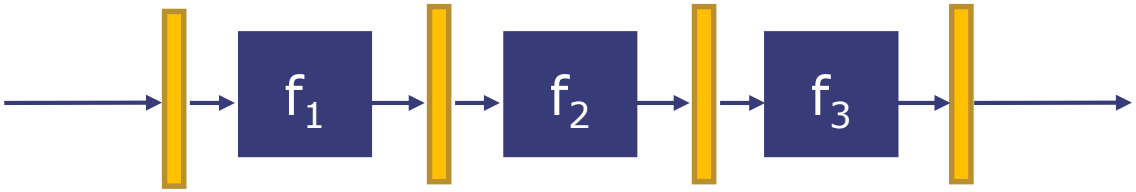


Summary: Design Alternatives

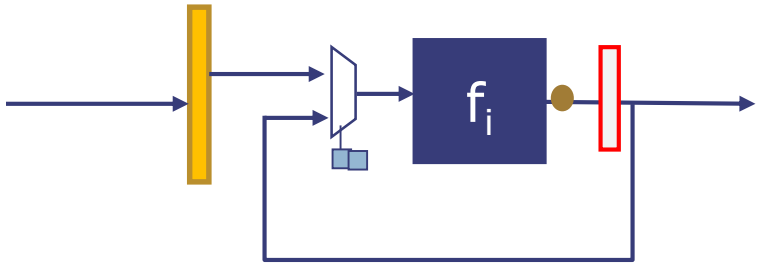
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

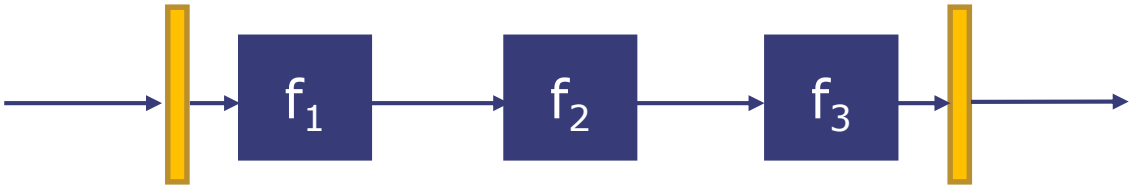


Folded reuse a block, multi-cycle (C)

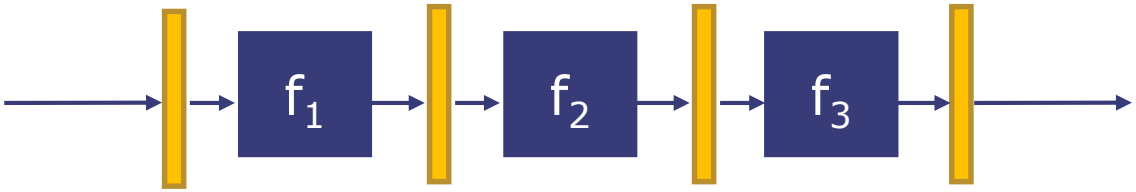


Summary: Design Alternatives

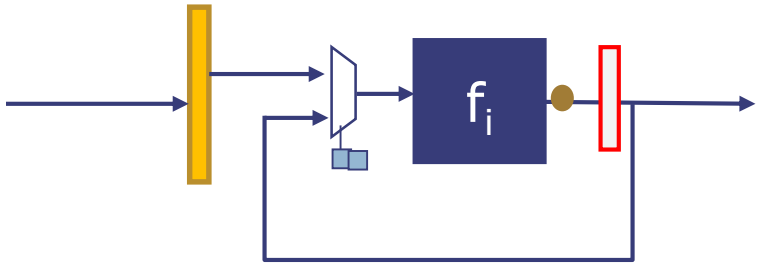
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock?

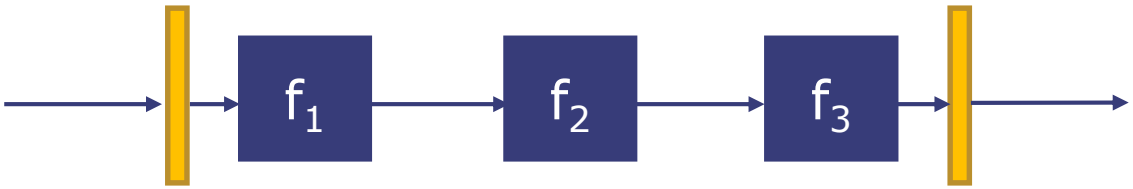
Latency?

Area?

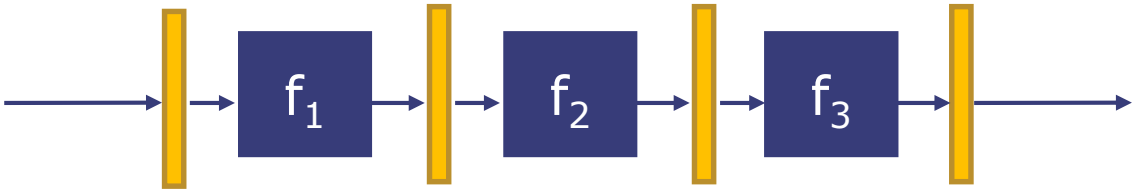
Throughput?

Summary: Design Alternatives

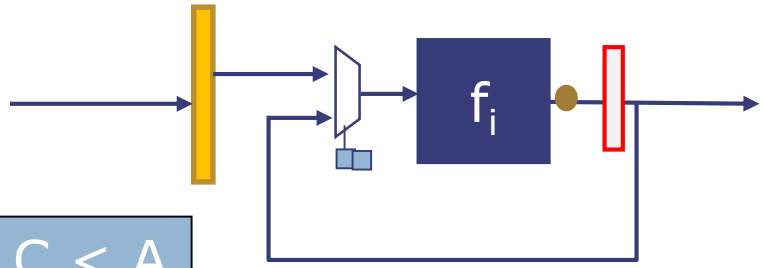
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

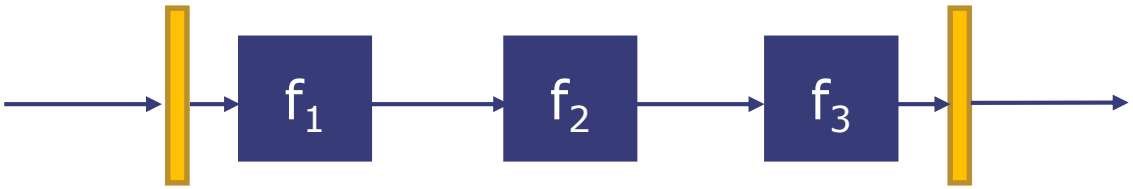
Latency?

Area?

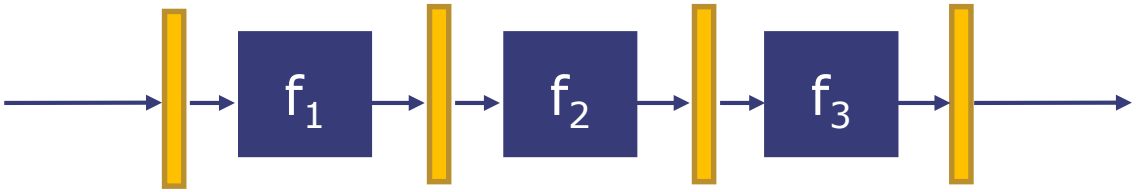
Throughput?

Summary: Design Alternatives

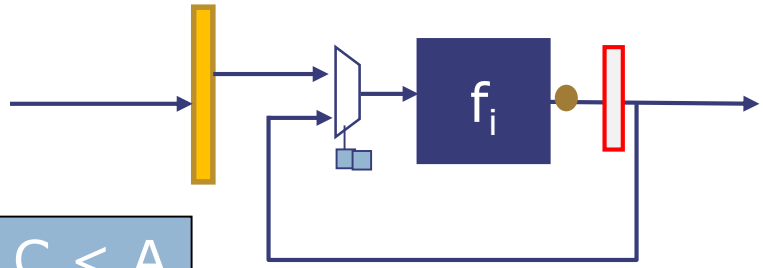
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

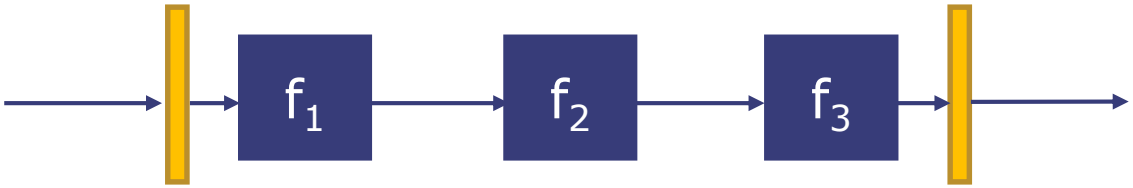
Area: C < A < B

Latency?

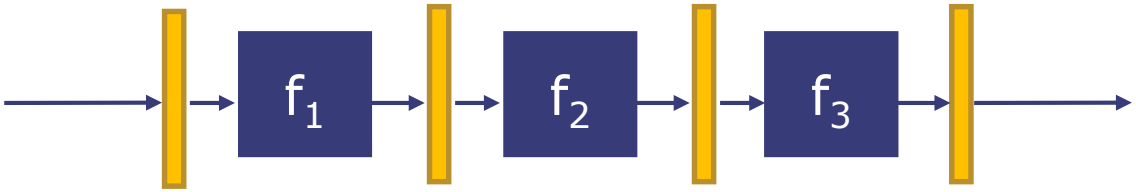
Throughput?

Summary: Design Alternatives

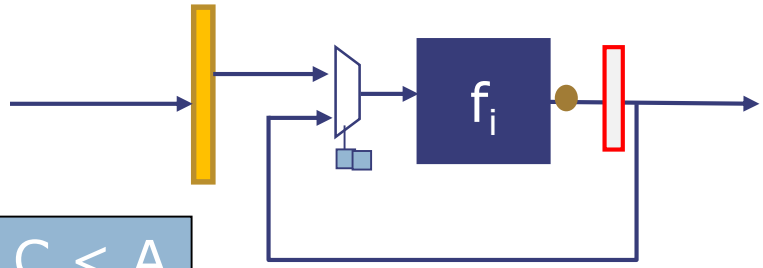
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

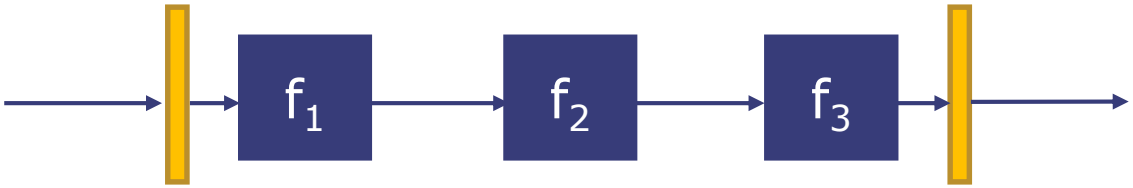
Latency: A < B < C

Area: C < A < B

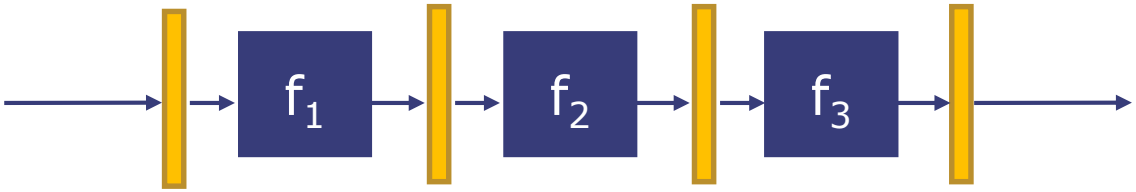
Throughput?

Summary: Design Alternatives

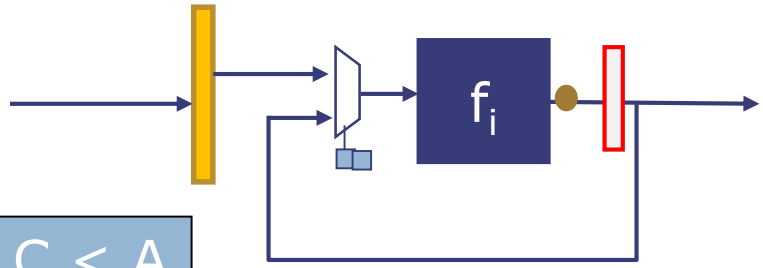
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

Latency: A < B < C

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_{PD} → lower t_{CLK} →
    lower latency & higher throughput

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    lower latency & higher throughput
- In practice, other constraints may set t_{CLK}
  - Propagation delay of other circuits
  - Limits on power consumption

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    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

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    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*

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    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* $t_{CLK,4stage} = t_{CLK,2stage}/2$ *?*

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    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* $t_{CLK,4stage} = t_{CLK,2stage}/2$? Throughput: 2x, Latency: 1x

Clock Frequency Constraints

- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    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* $t_{CLK,4stage} = t_{CLK,2stage}/2$? Throughput: 2x, Latency: 1x
    - *If* $t_{CLK,4stage} = t_{CLK,2stage}$?

Clock Frequency Constraints

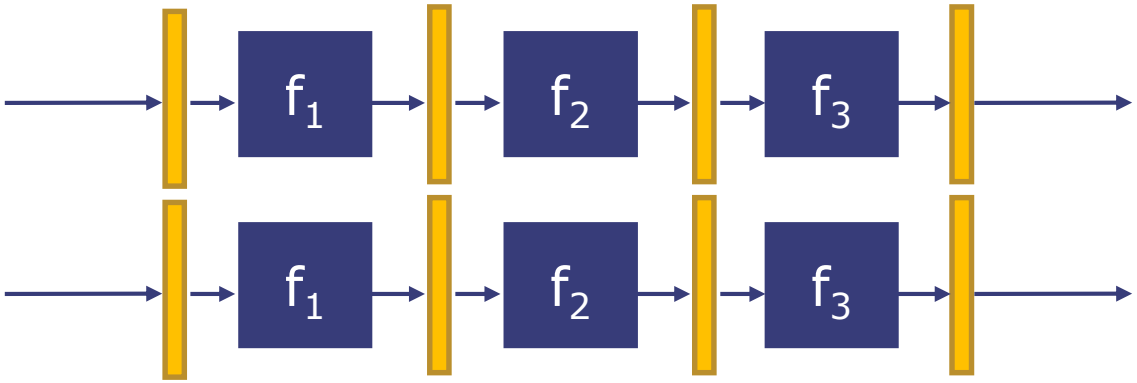
- To analyze latency and throughput, so far we've assumed t_{CLK} depends only on our circuit
  - So lower t_{PD} → lower t_{CLK} →
    lower latency & higher throughput
- In practice, other constraints may set t_{CLK}
  - Propagation delay of other circuits
  - Limits on power consumption
- When our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

Increasing Throughput with Replication

- We can increase throughput by replicating a circuit and using the copies in parallel

Increasing Throughput with Replication

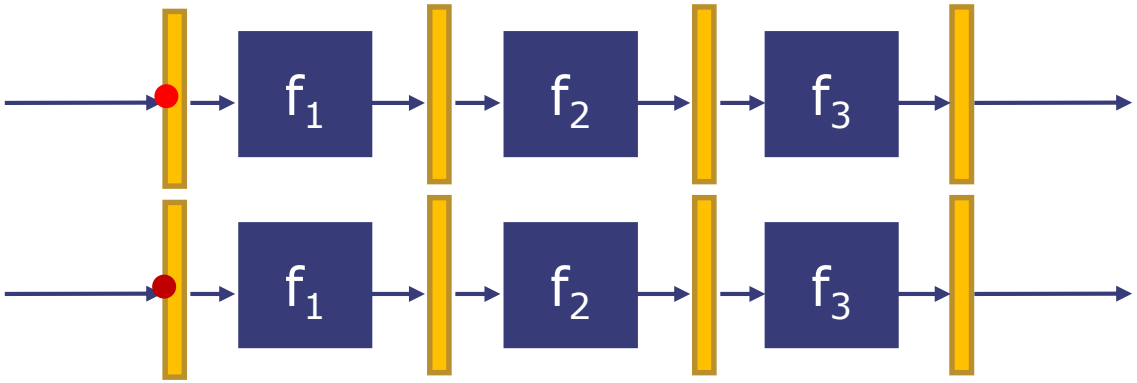
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

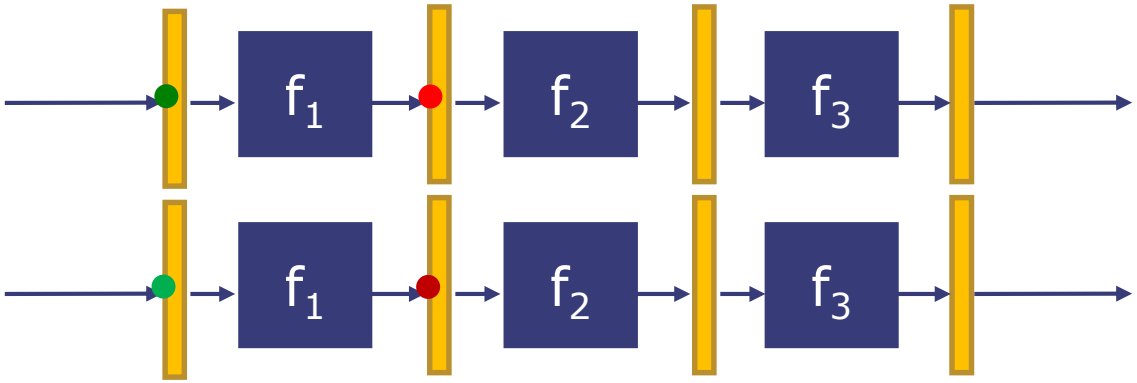
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

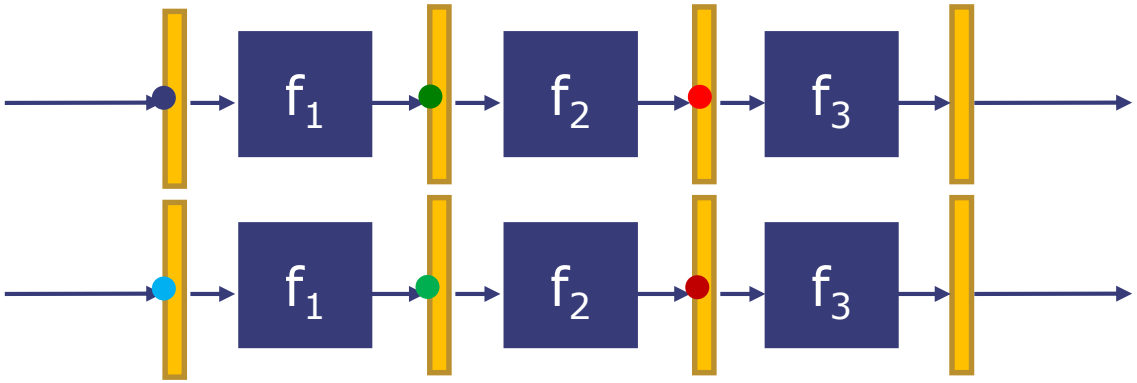
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

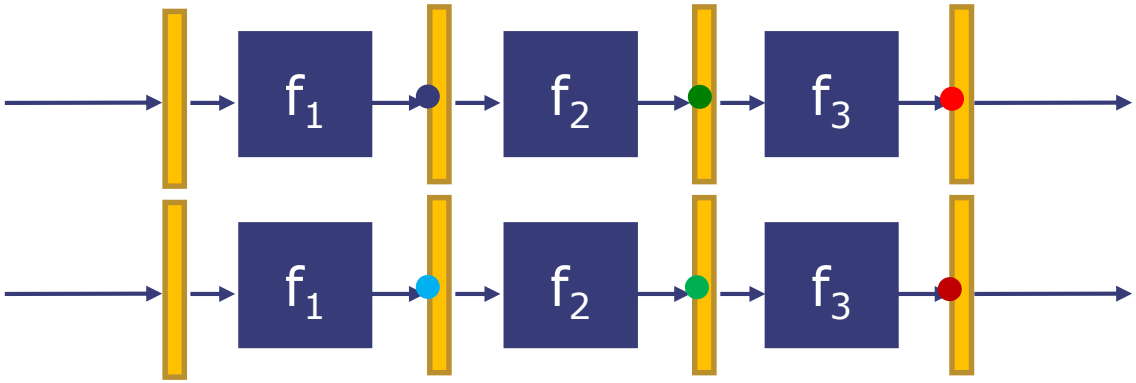
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

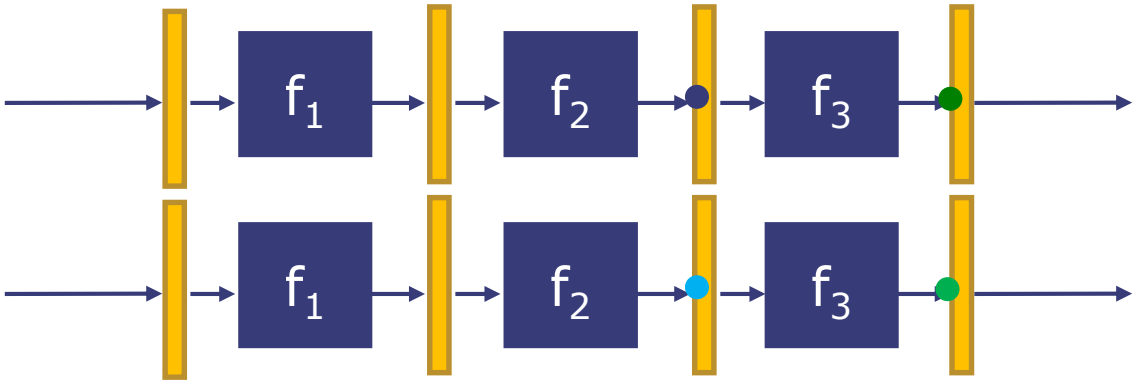
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

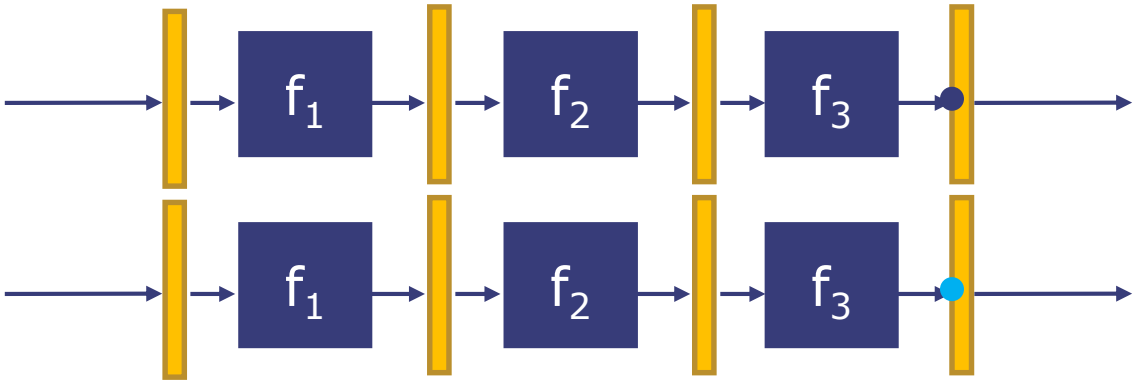
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

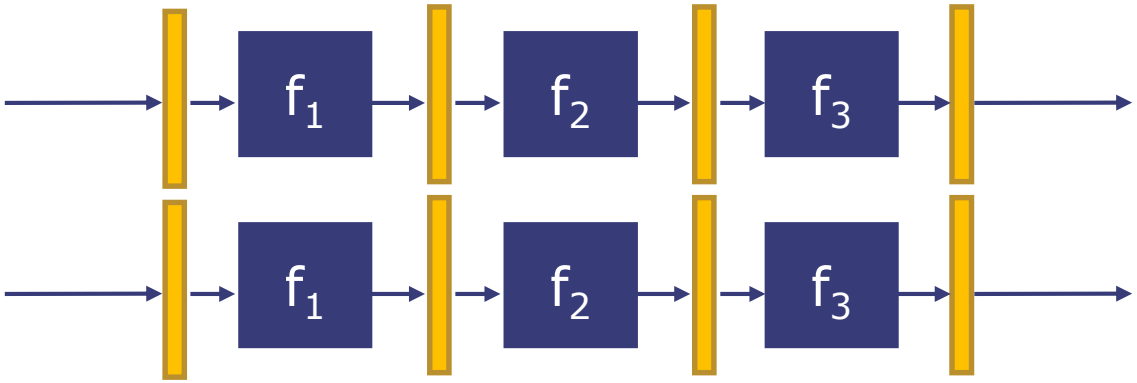
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle

Increasing Throughput with Replication

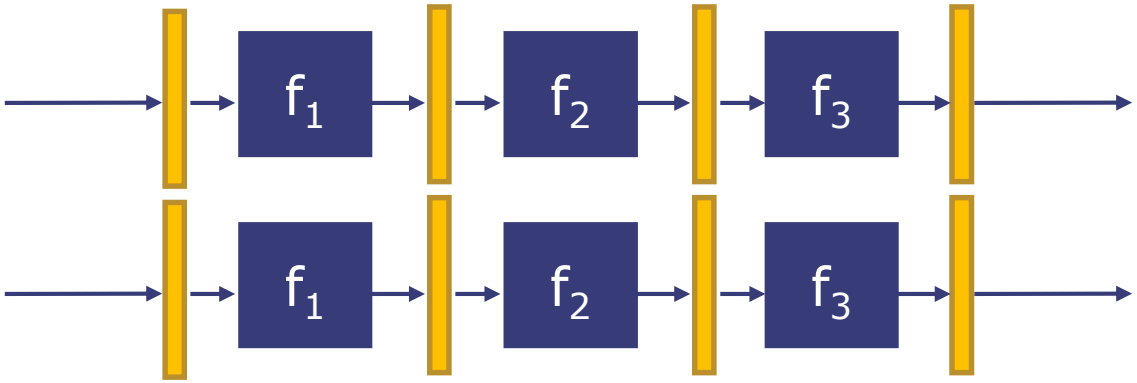
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle
  - Metrics vs a single pipeline: *Clock?*
    Latency?
    Throughput?
    Area?

Increasing Throughput with Replication

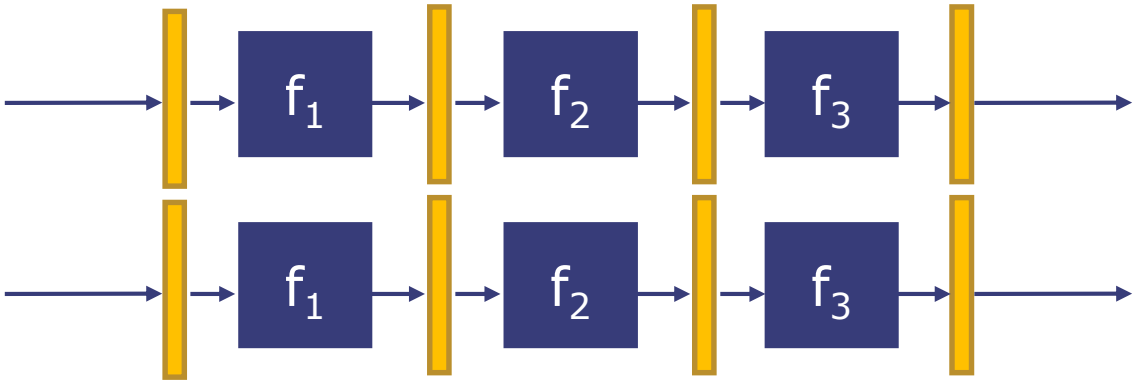
- We can increase throughput by replicating a circuit and using the copies in parallel
- Example: Using two pipelined circuits in parallel



  - Processes two values each cycle
  - Metrics vs a single pipeline: *Clock?* *Same*
    Latency?
    Throughput?
    Area?

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

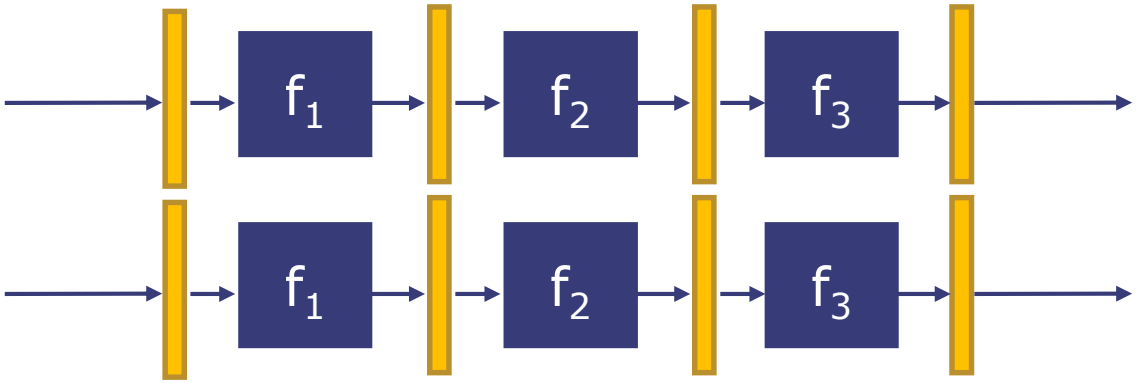


  - Processes two values each cycle
  - Metrics vs a single pipeline:

Clock?	*Same*
Latency?	*Same*
Throughput?	
Area?	

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

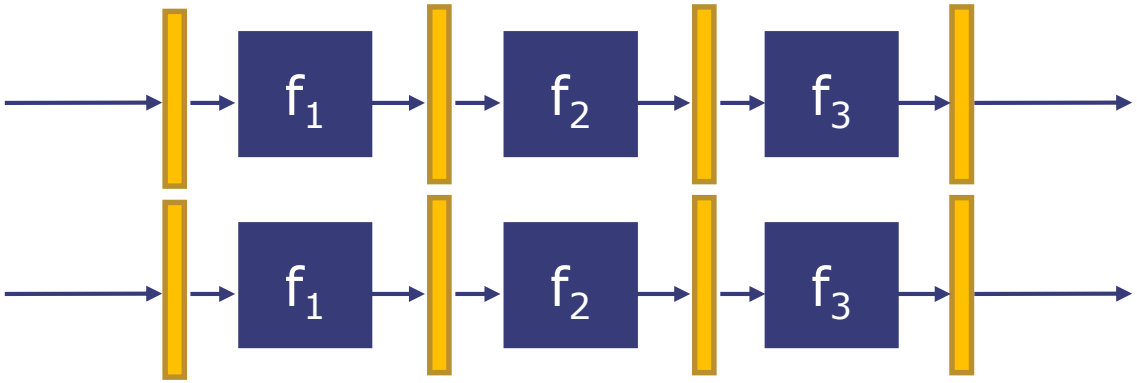


  - Processes two values each cycle
  - Metrics vs a single pipeline:

Clock?	*Same*
Latency?	*Same*
Throughput?	*2x*
Area?	

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:

Clock?	*Same*
Latency?	*Same*
Throughput?	*2x*
Area?	*2x*

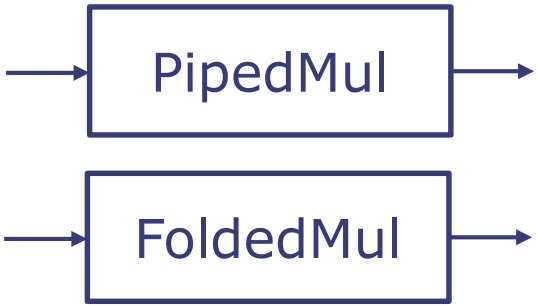
Example: Pipeline or Replicate?

- Consider the following two multipliers

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

Example: Pipeline or Replicate?

- Consider the following two multipliers

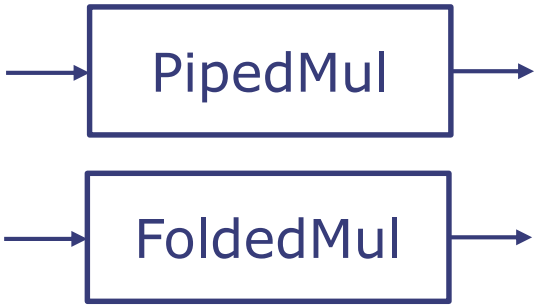


4-stage pipelined multiplier
Throughput = $1/t_{CLK}$

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

Example: Pipeline or Replicate?

- Consider the following two multipliers



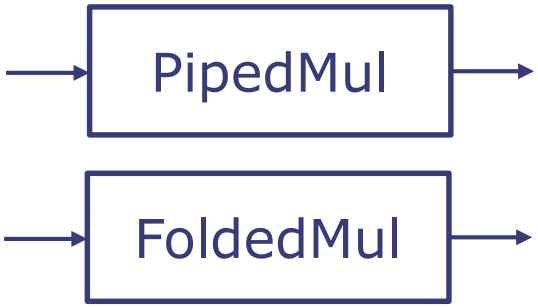
4-stage pipelined multiplier
Throughput = $1/t_{CLK}$

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

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

Example: Pipeline or Replicate?

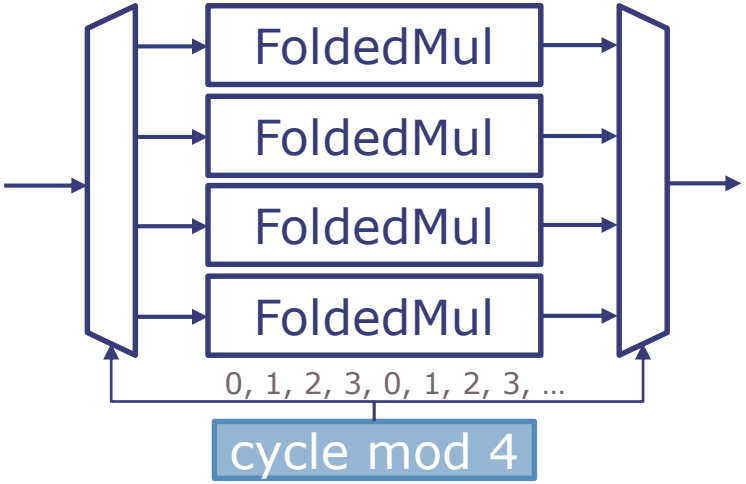
- Consider the following two multipliers



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

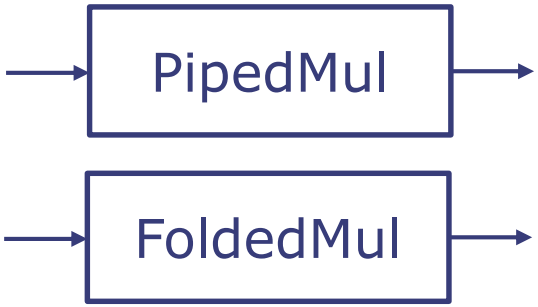
Folded multiplier that takes 4 cycles per output -> Throughput = $1/(4t_{CLK})$
Similar t_{CLK} vs. PipedMul, lower area

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



Example: Pipeline or Replicate?

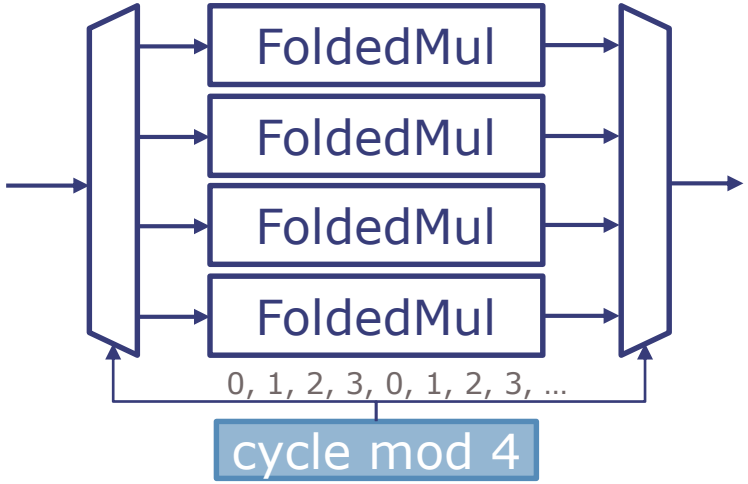
- Consider the following two multipliers



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

Folded multiplier that takes 4 cycles per output -> Throughput = $1/(4t_{CLK})$
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 = $4*1/(4t_{CLK}) = 1/t_{CLK}$

Software vs. Hardware Design
Timing is the key difference

1. Software interfaces (even instructions) are timing-independent
   - Specify *what* should happen, not *when*

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
```

Software vs. Hardware Design
Timing is the key difference

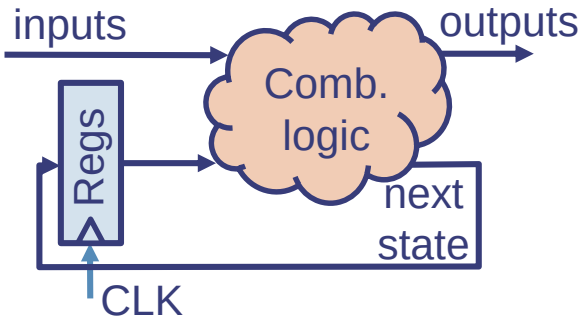
1. Software interfaces (even instructions) are timing-independent
   - Specify *what* should happen, not *when*

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

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

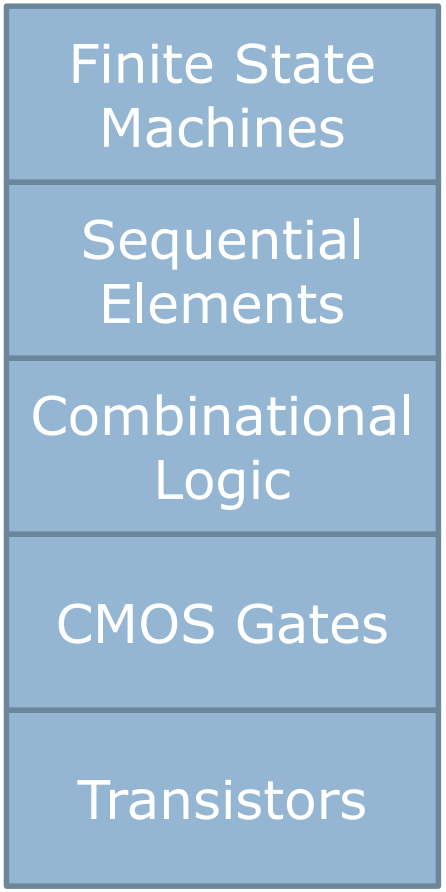
2. Hardware design is all about timing
   - Specify what happens on every clock cycle…
   - …which itself determines the length of the clock cycle

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

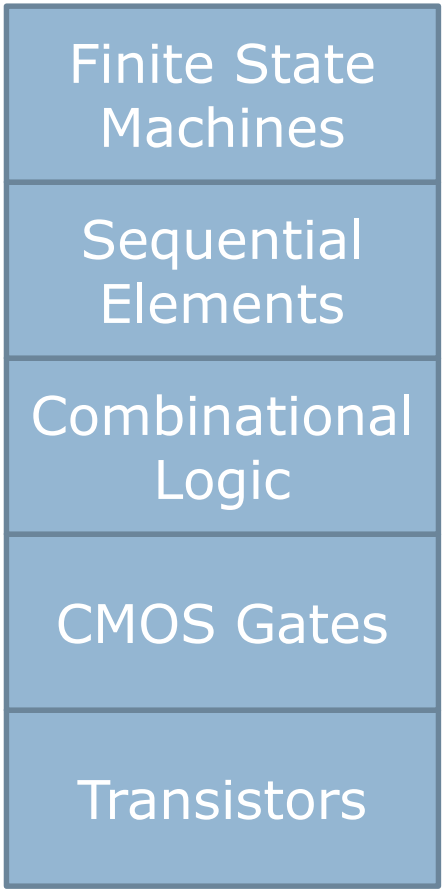


From Special-Purpose FSMs to General-Purpose Processors

6.004 So Far


Finite State Machines
Sequential Elements
Combinational Logic
CMOS Gates
Transistors

6.004 So Far



- 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

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

Example: Factorial FSM

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

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;
}
```

Example: Factorial FSM

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

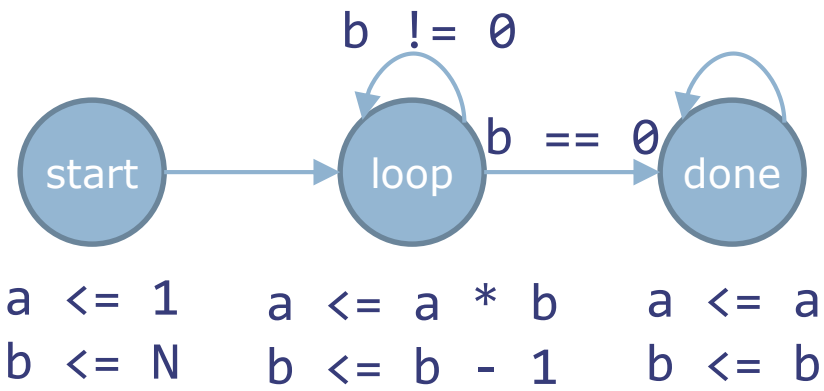
Python:

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

C:

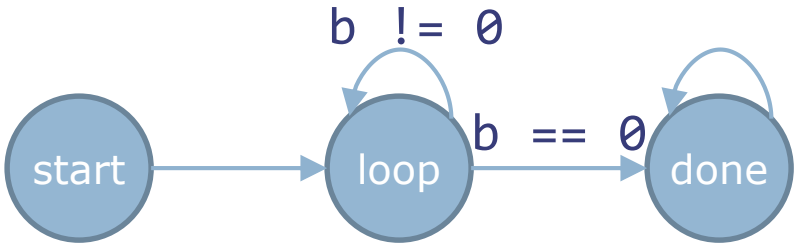
```
int a = 1;
int b = N;
while (b != 0) {
      a = a * b;
      b = b - 1;
}
```

High-level FSM:



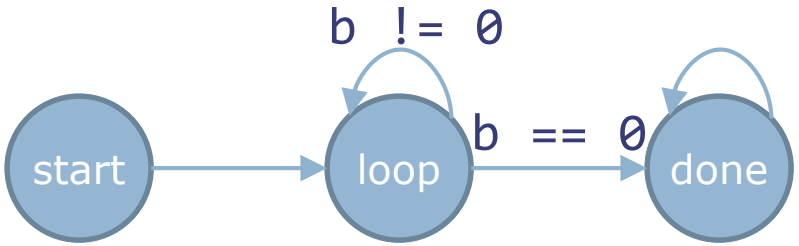
- Describes cycle-by-cycle behavior
- Registers (a, b)
- States (start, loop, done)
- Boolean transitions (b==0, b!=0)
- Register assignments in states (e.g., a ← a * b)

Datapath for Factorial




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

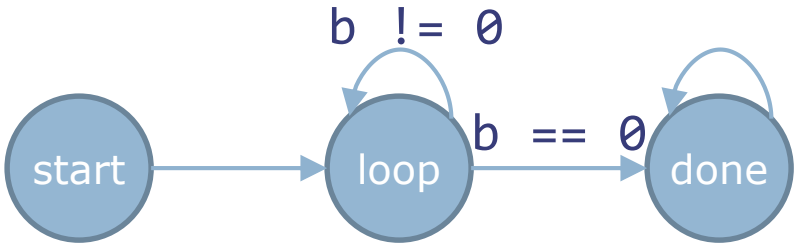
Datapath for Factorial




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

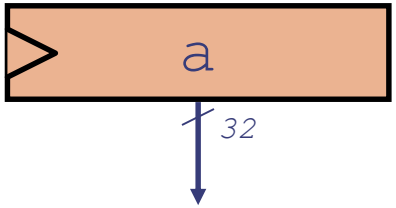
- Implement registers

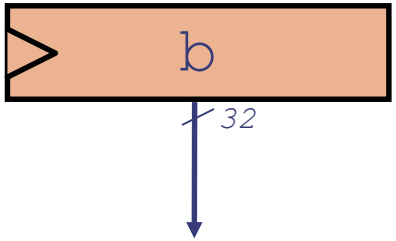
Datapath for Factorial



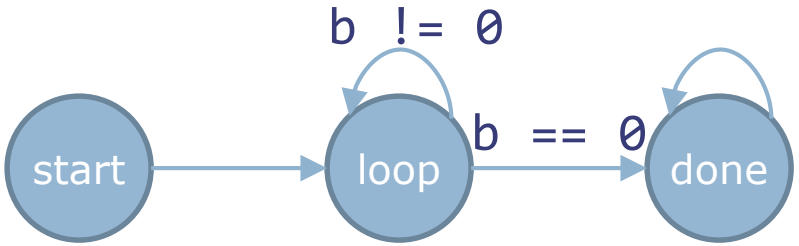

```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers



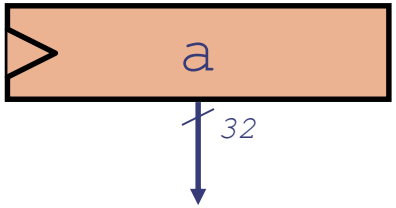


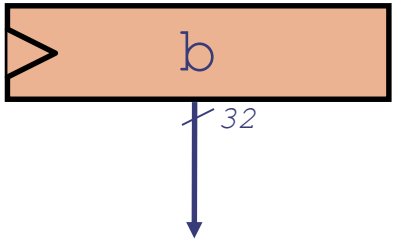
Datapath for Factorial



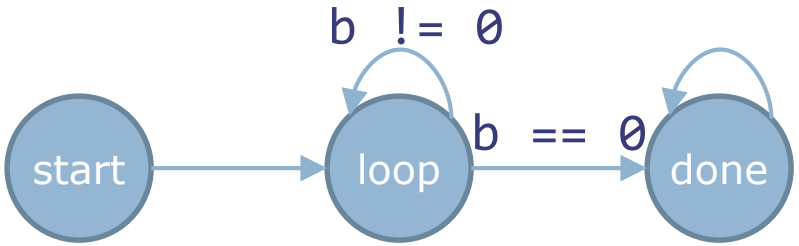

```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment



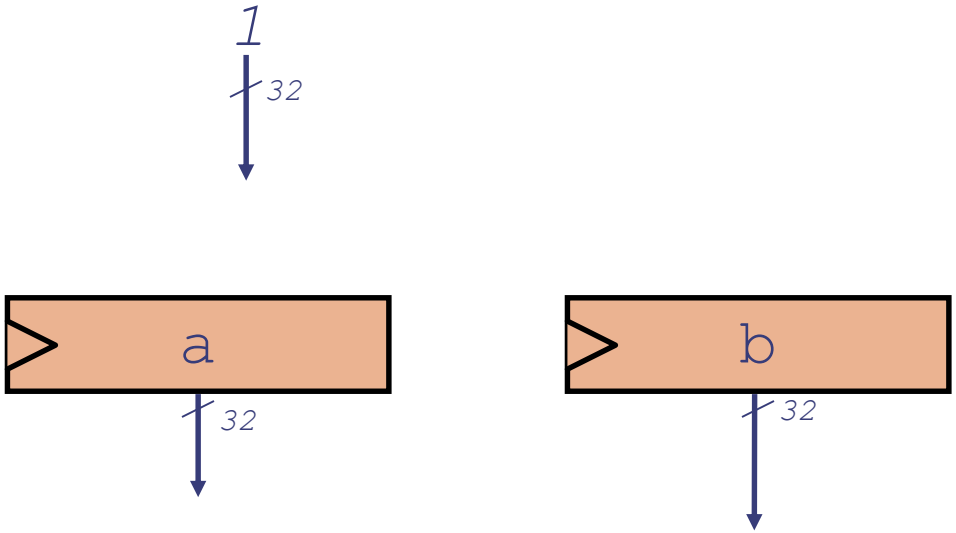


Datapath for Factorial

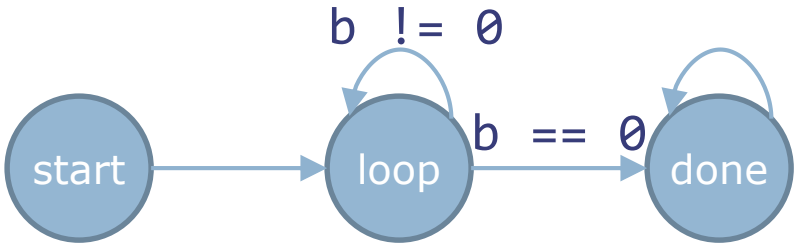



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment

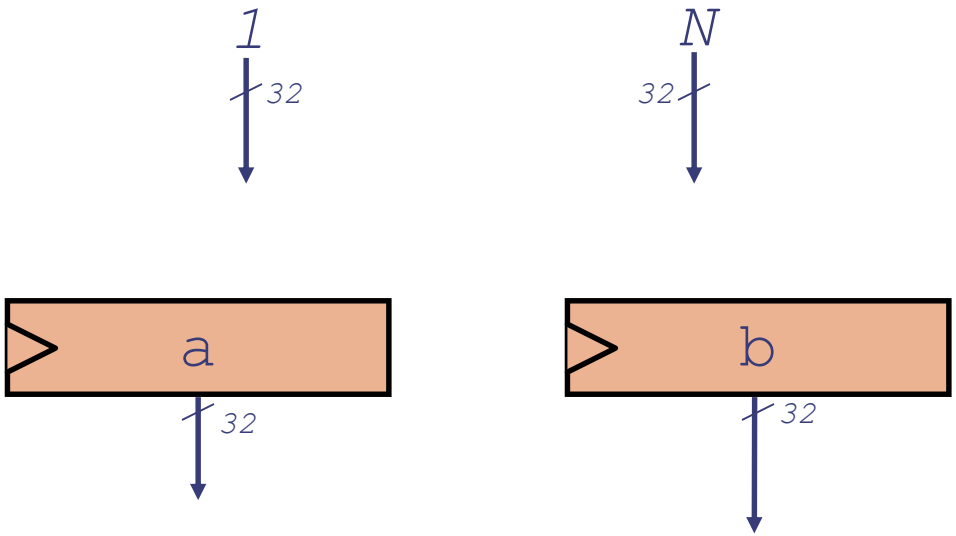


Datapath for Factorial

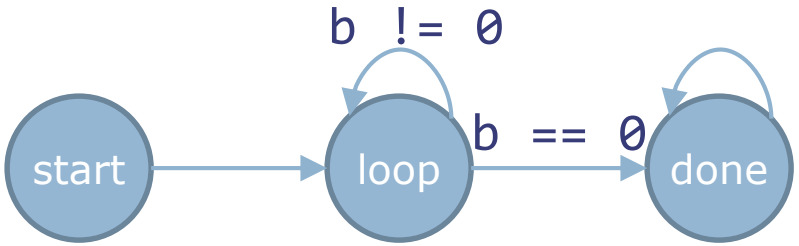



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment

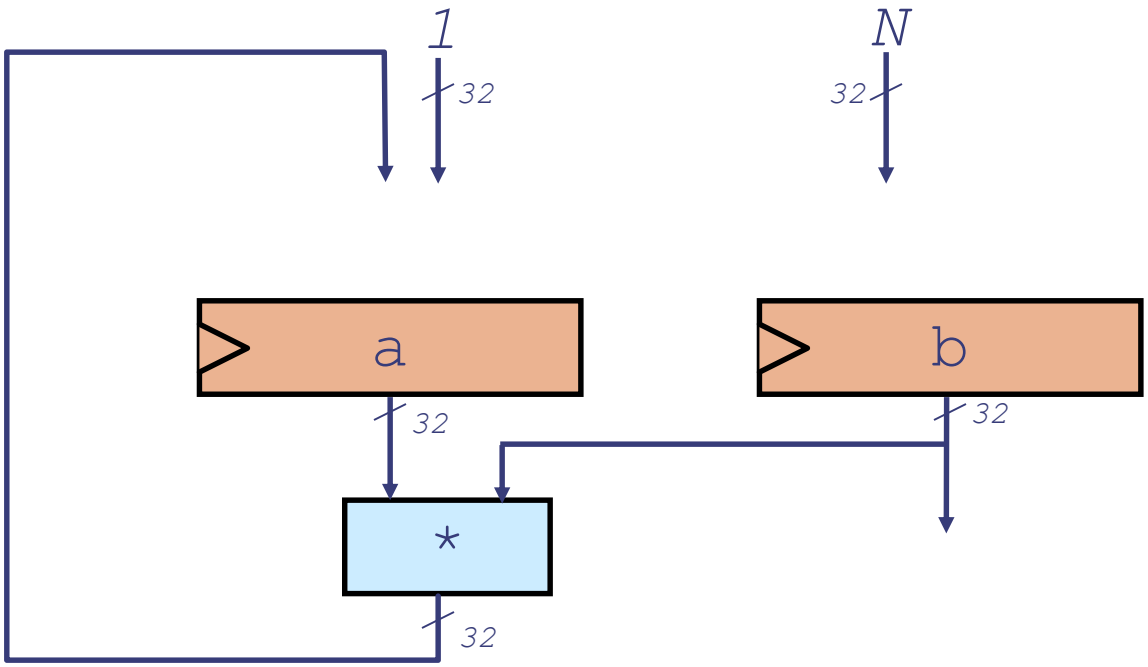


Datapath for Factorial

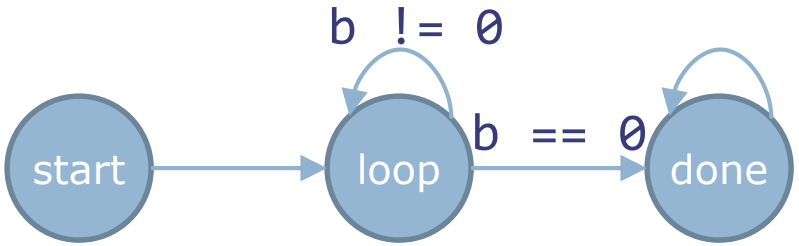



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment

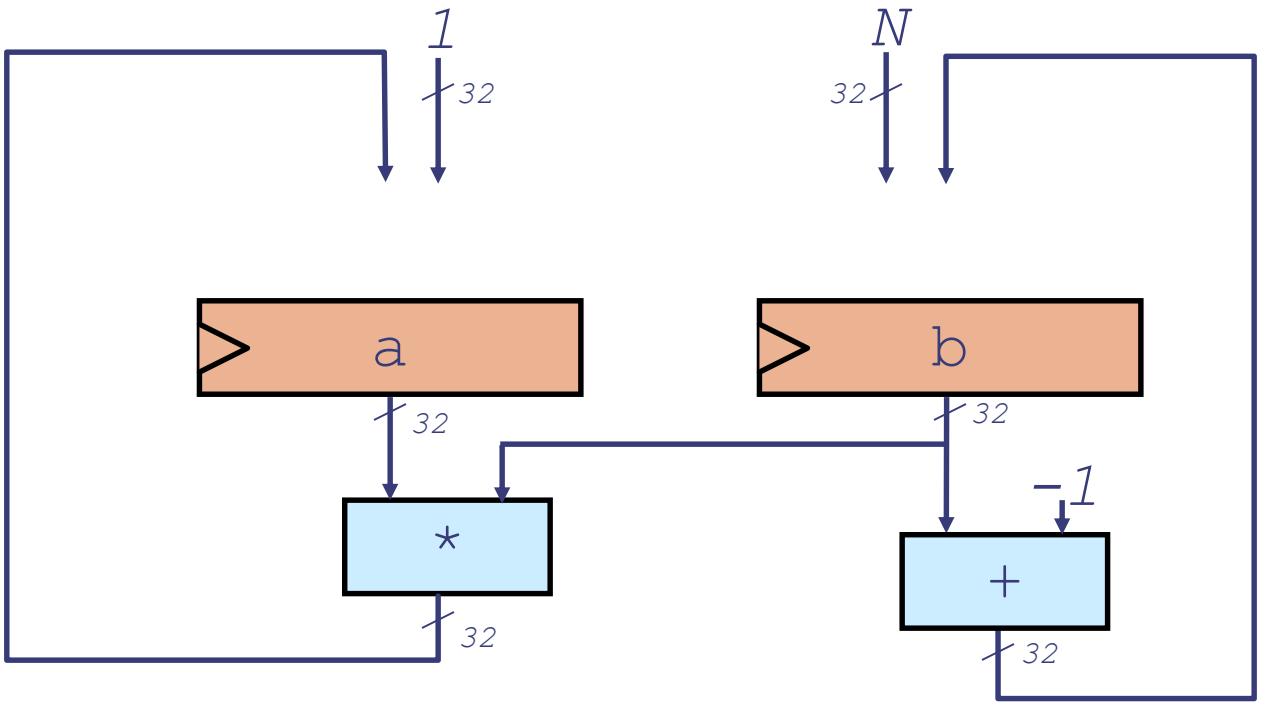


Datapath for Factorial

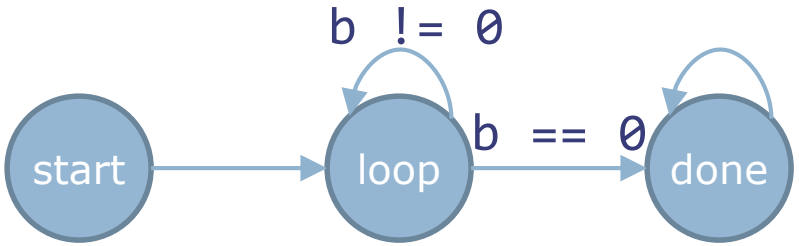



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment

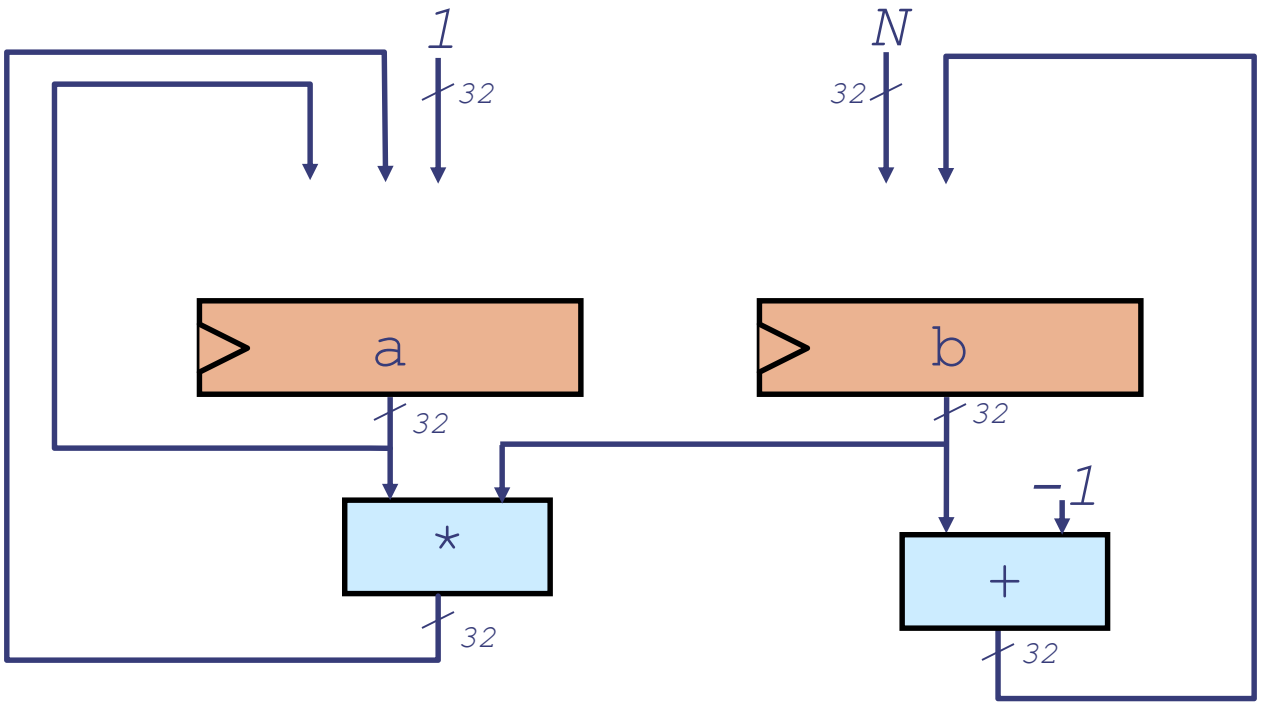


Datapath for Factorial

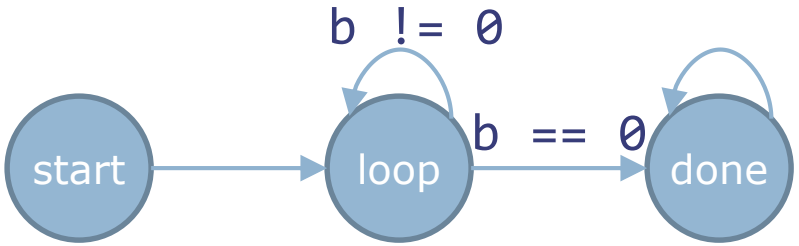



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment

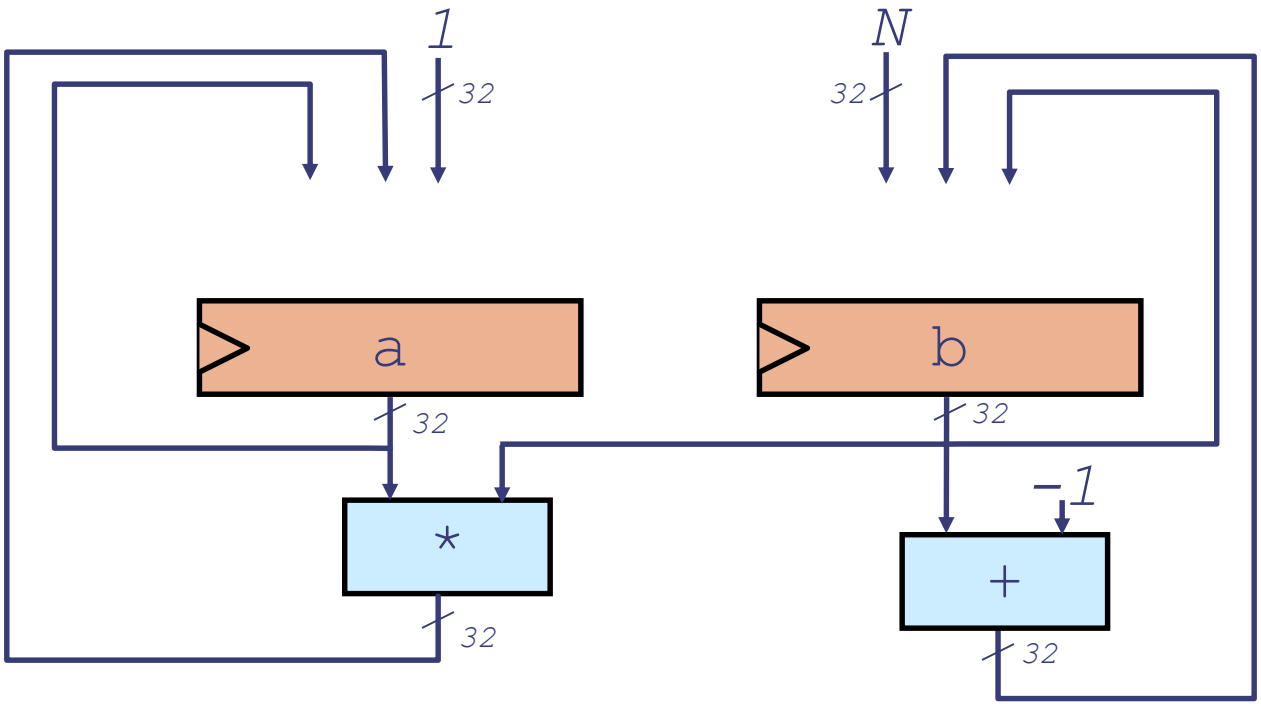


Datapath for Factorial

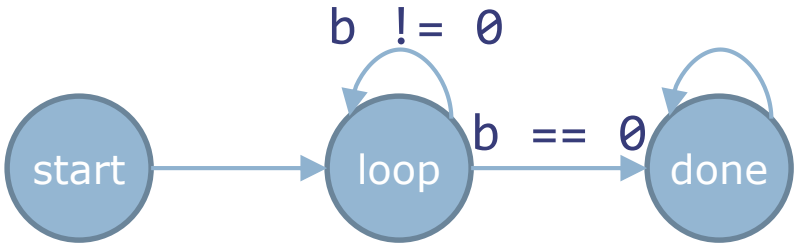



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement registers
- Implement combinational circuit for each assignment

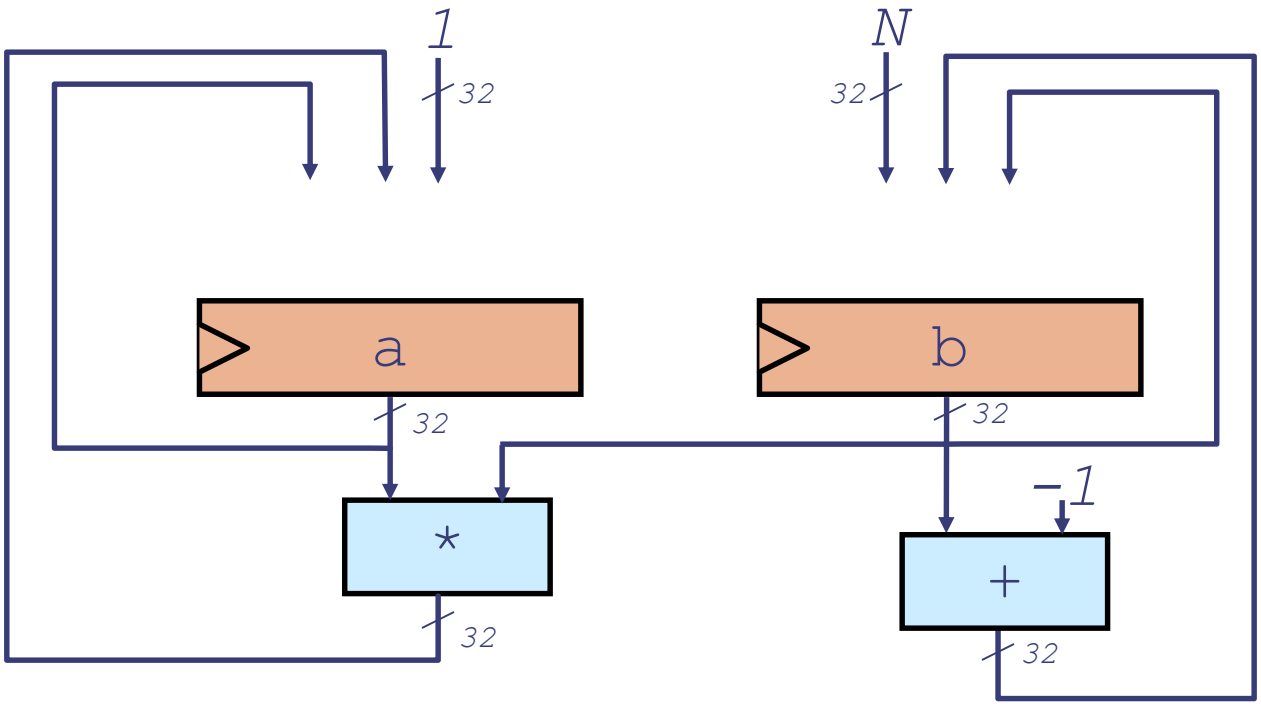


Datapath for Factorial

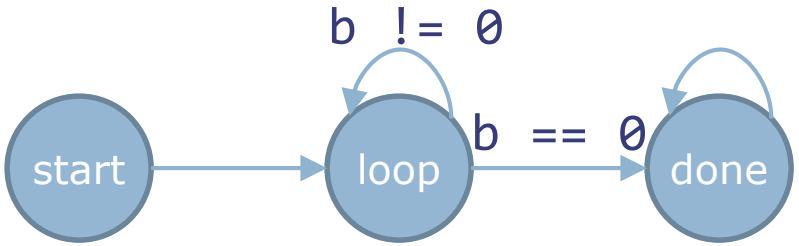



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

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

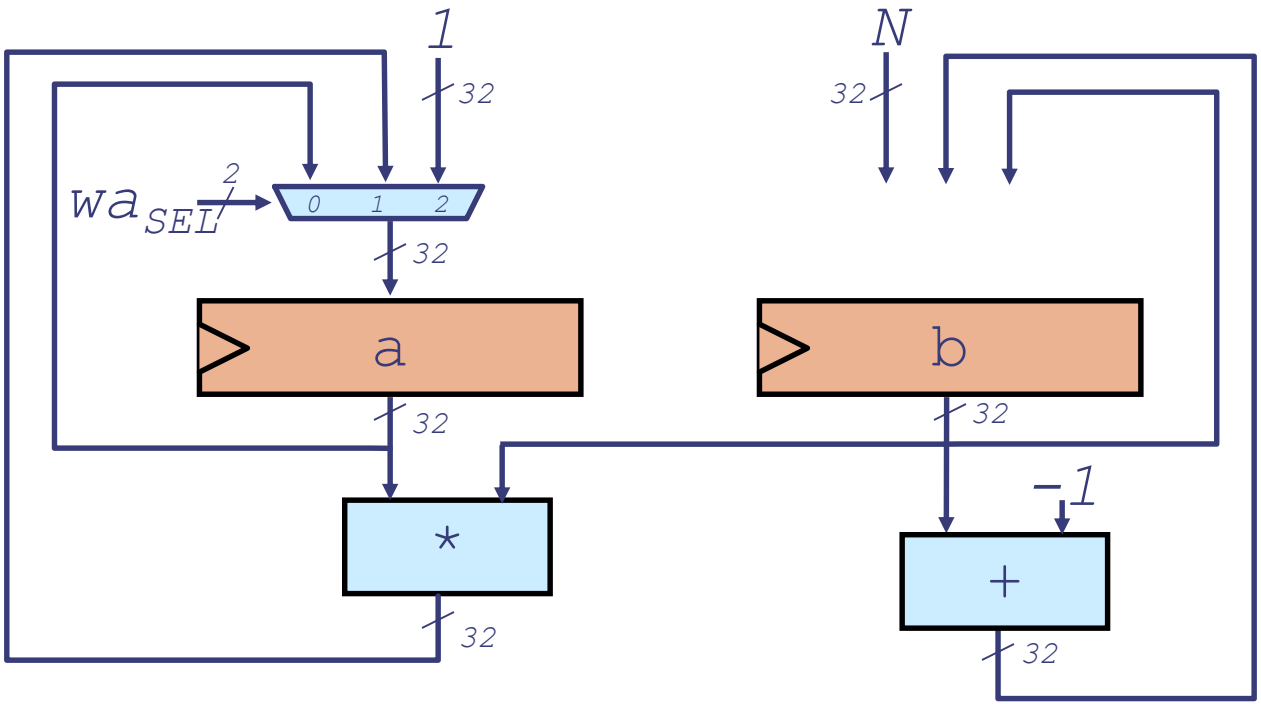


Datapath for Factorial

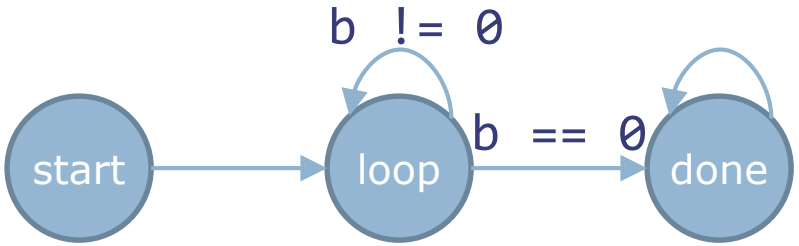



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

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

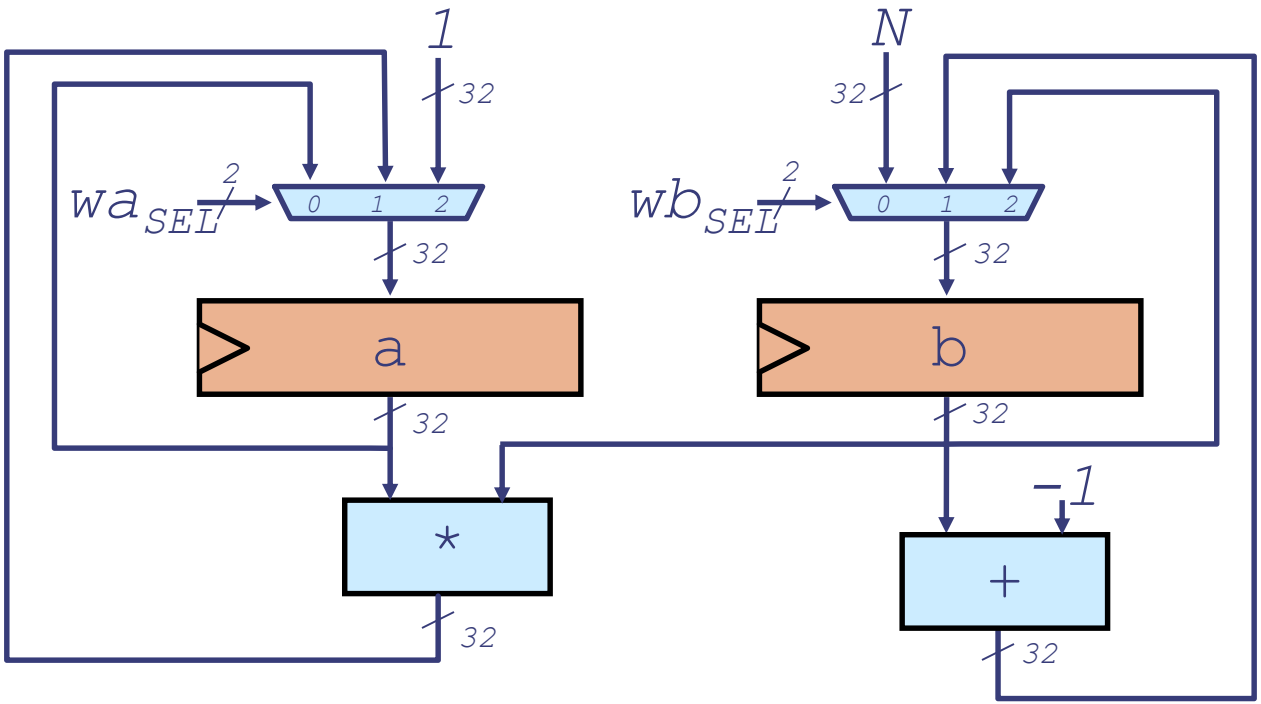


Datapath for Factorial

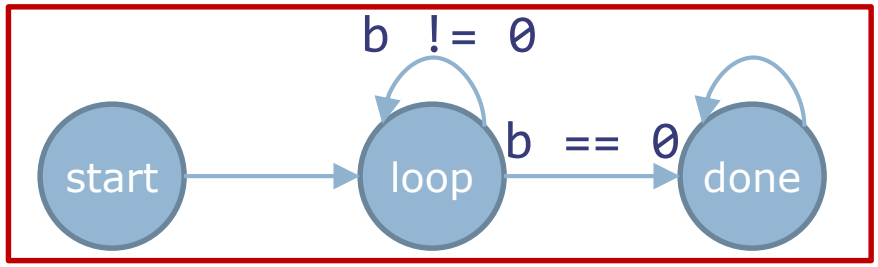



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

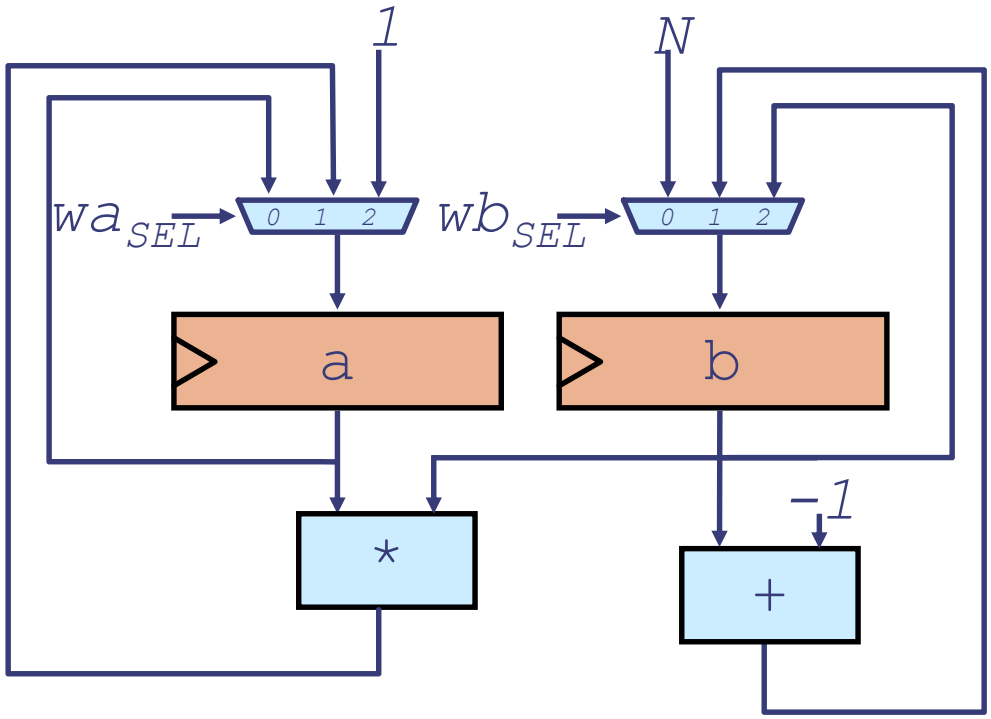
- Implement registers
- Implement combinational circuit for each assignment
- Connect to input muxes



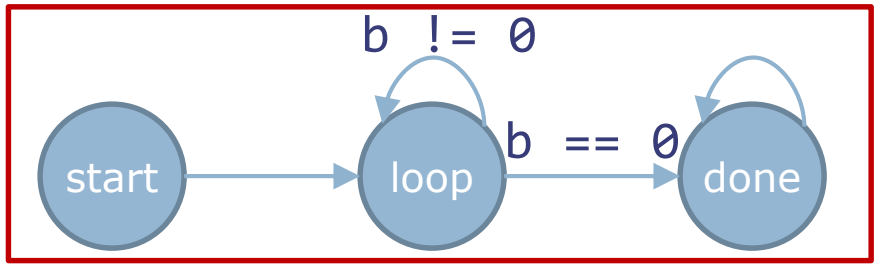
Control FSM for Factorial




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

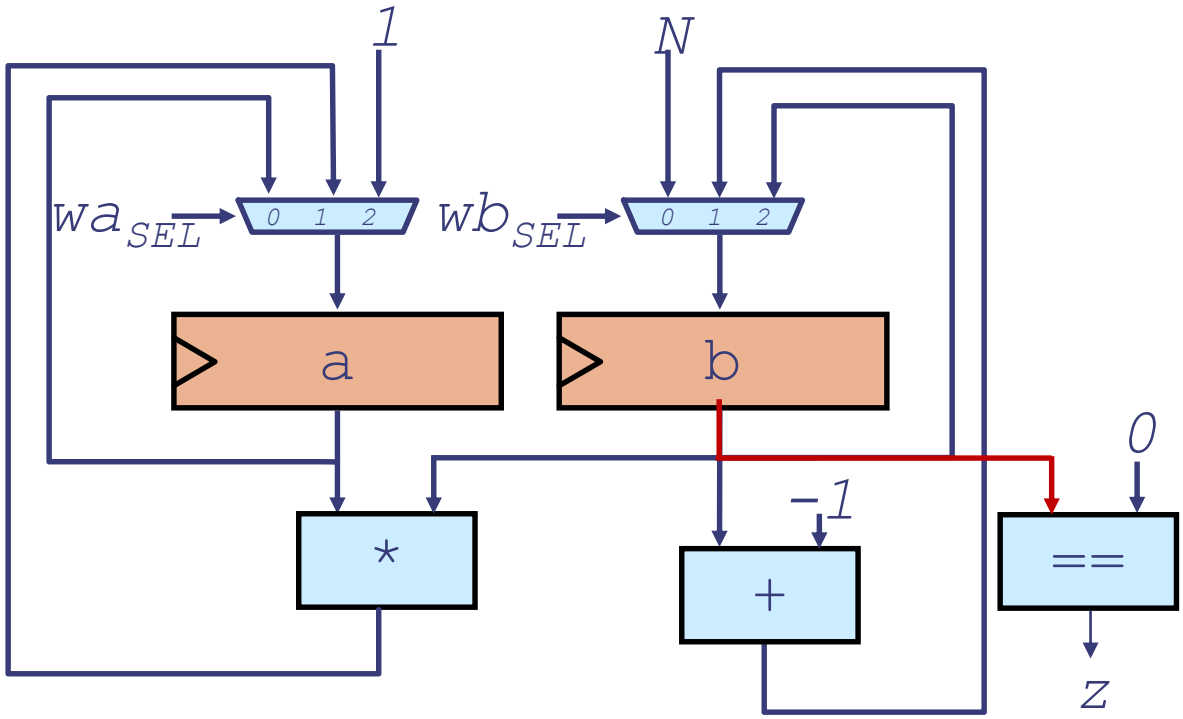


Control FSM for Factorial

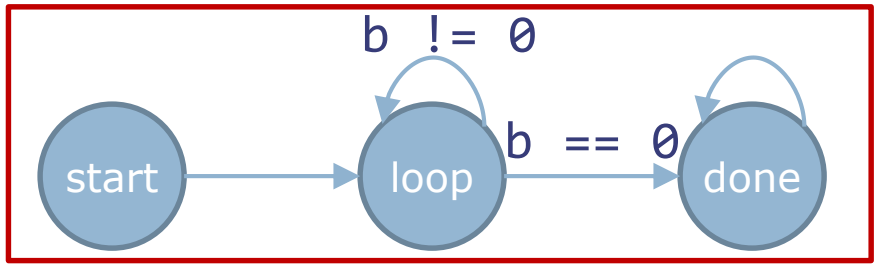



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

- Implement combinational logic for transition conditions

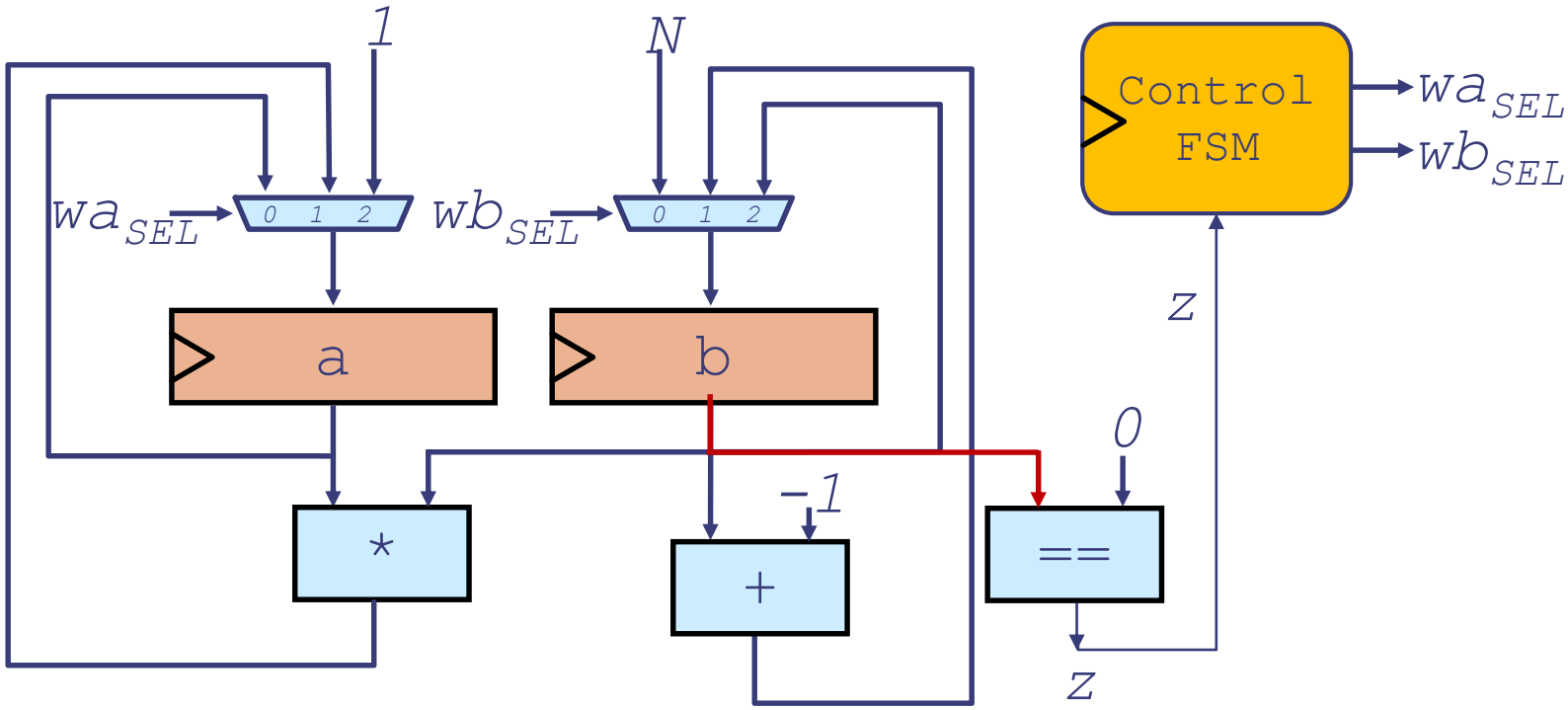


Control FSM for Factorial

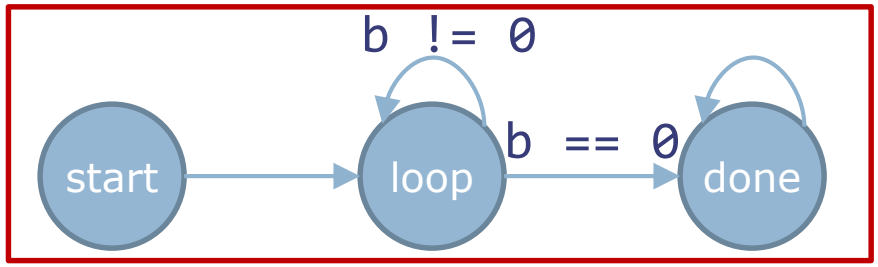



```
a <= 1      a <= a * b      a <= a
b <= N      b <= b - 1      b <= b
```

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

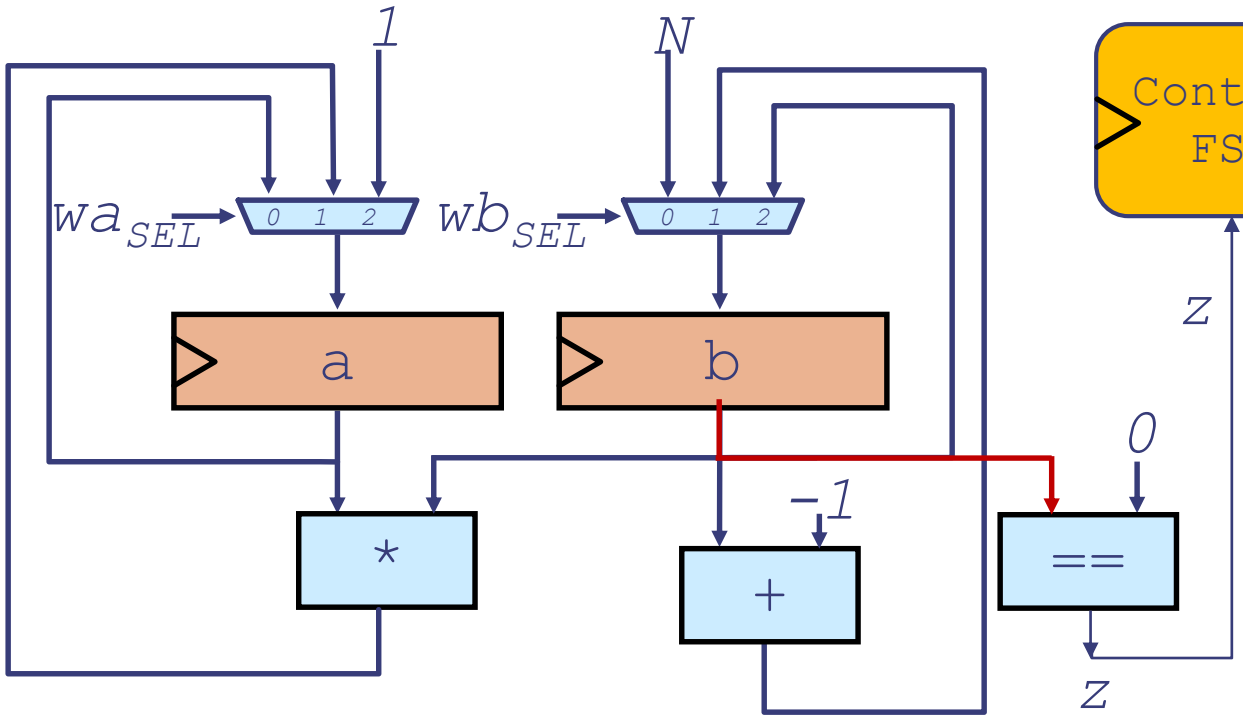


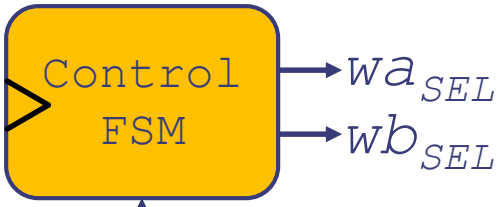
Control FSM for Factorial




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

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





S	z	wa_{SEL}	wb_{SEL}	S′
0	0	2	0	1
0	1	2	0	1
1	0	1	1	1
1	1	1	1	2
2	0	0	2	2
2	1	0	2	2

Programming the Datapath

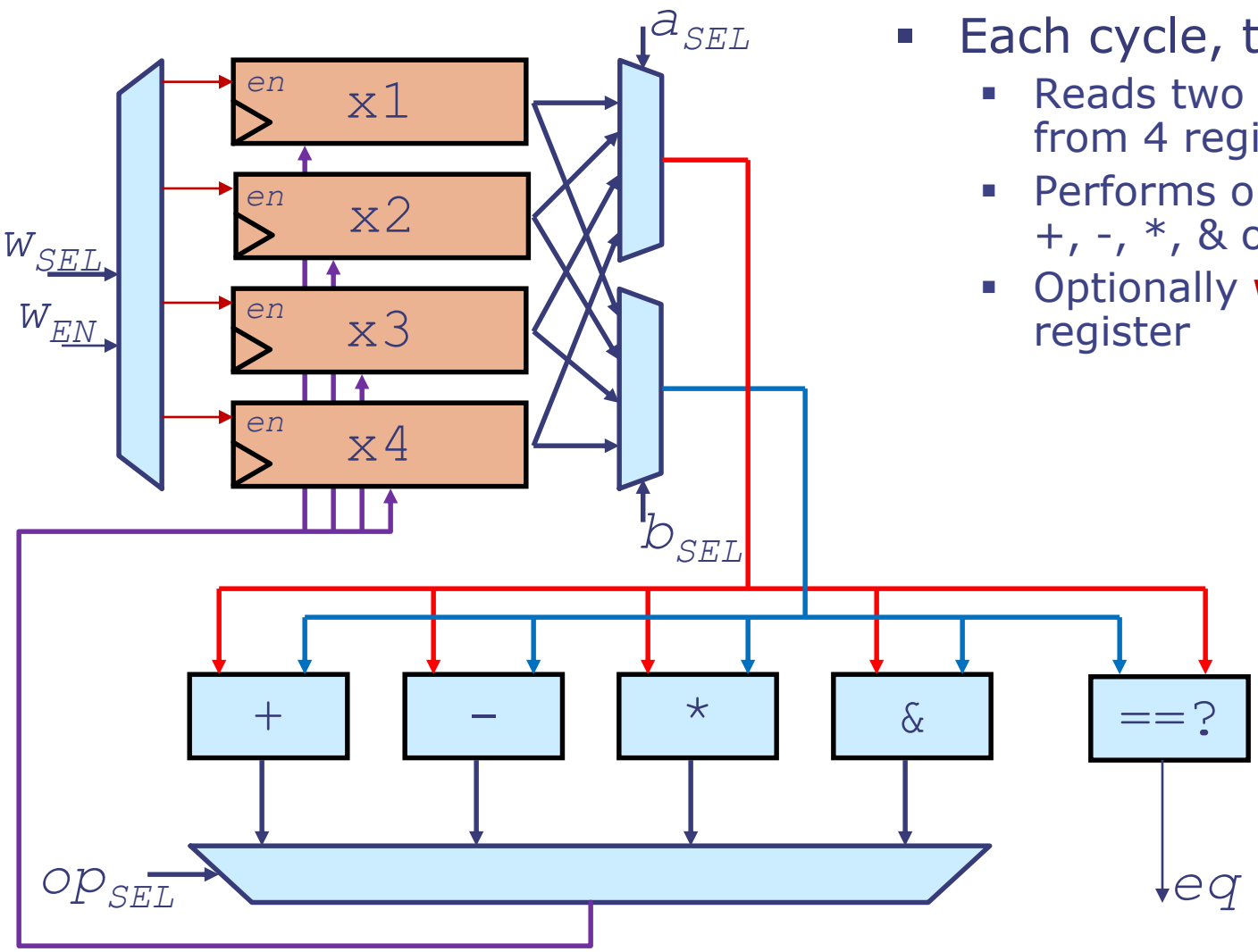
- We can use our factorial datapath and change the control FSM to solve other problems! Examples:
  - Multiplication
  - Squaring

Programming the Datapath

- We can use our factorial datapath and change the control FSM to solve other problems! Examples:
  - Multiplication
  - Squaring

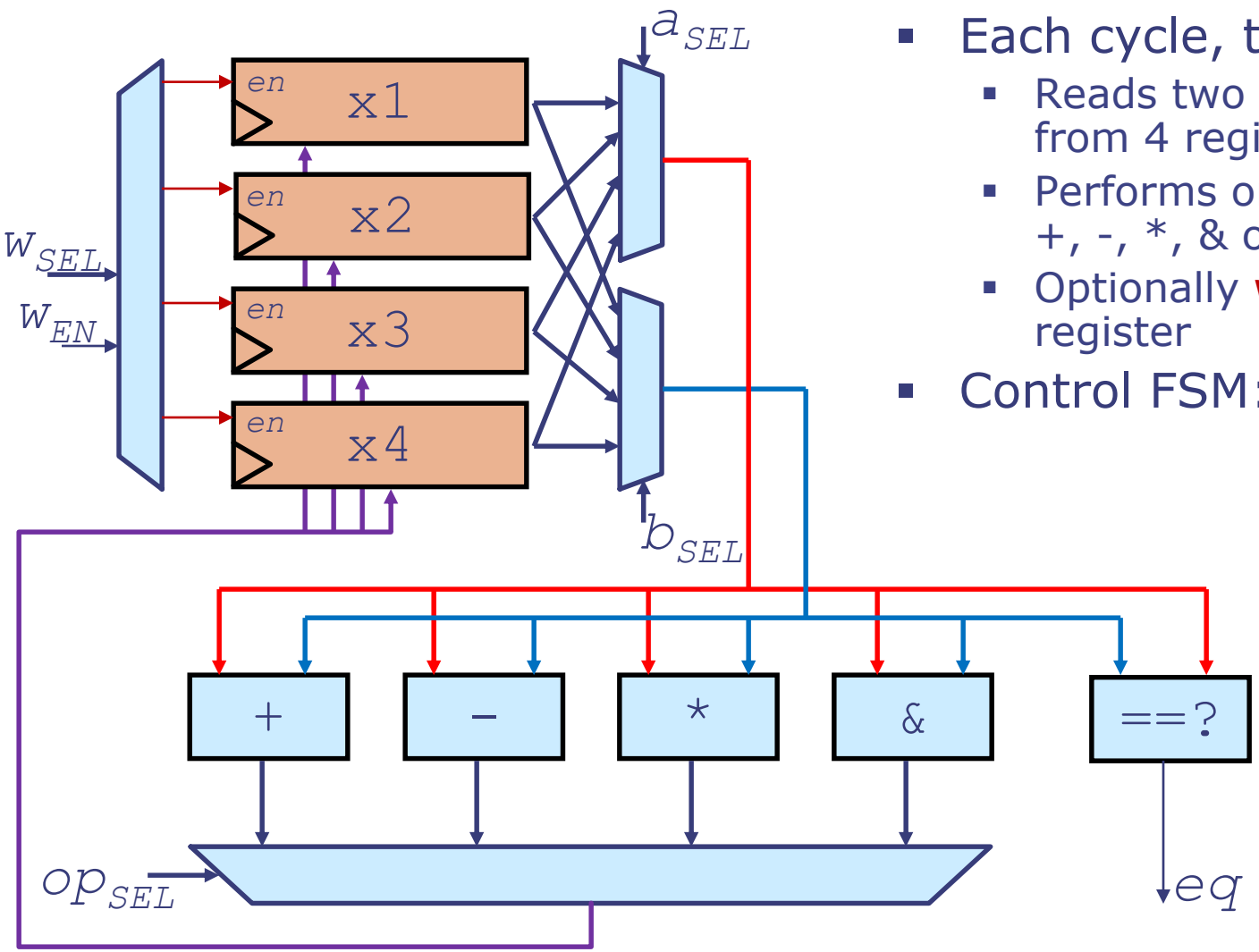
- But very limited problems. Reasons:
  - Limited storage (only two registers!)
  - Limited set of operations, and inputs to those operations
  - Limited inputs to the control FSM

A Simple Programmable Datapath

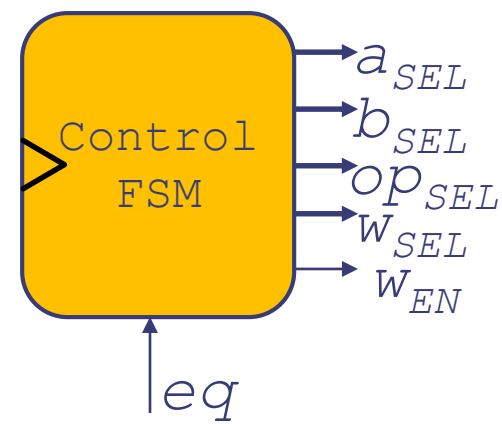


- Each cycle, this datapath:
  - Reads two operands (a, b) from 4 registers (x1-x4)
  - Performs one operation of +, -, *, & on operands
  - Optionally writes result to a register

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:

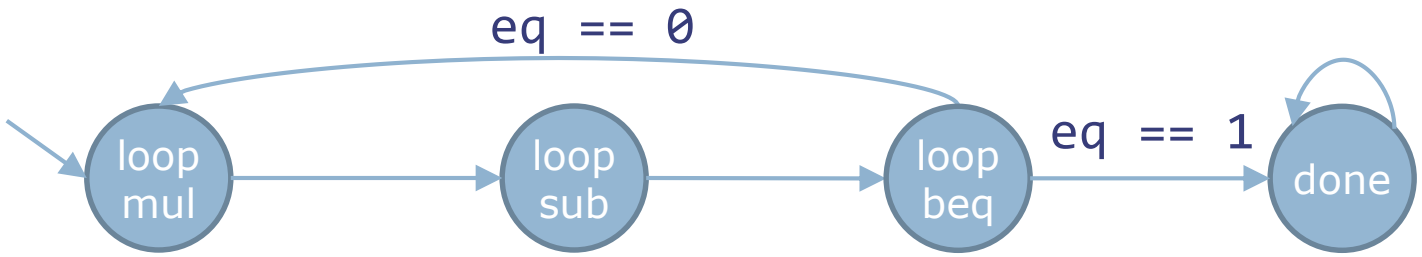


A Control FSM for Factorial

- Assume initial register contents:

```
x1 value = 1
x2 value = N
x3 value = -1
x4 value = 0
```

- Control FSM:



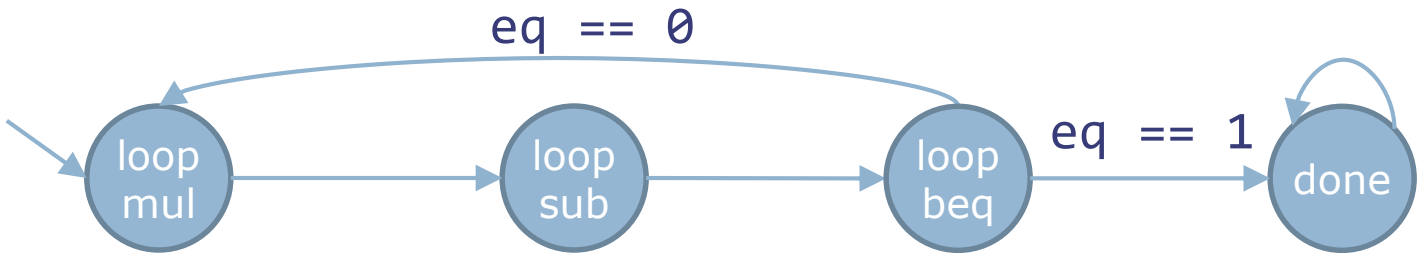
loop mul	loop sub	loop beq	done
asel = x1	asel = x2	asel = x2	asel = X
bsel = x2	bsel = x3	bsel = x4	bsel = X
opsel = 2 (*)	opsel = 0 (+)	opsel = X	opsel = X
wen = 1	wen = 1	wen = 0	wen = 0
wsel = x1	wsel = x2	wsel = X	wsel = X

A Control FSM for Factorial

- Assume initial register contents:

```
x1 value = 1
x2 value = N
x3 value = -1
x4 value = 0
```

- Control FSM:



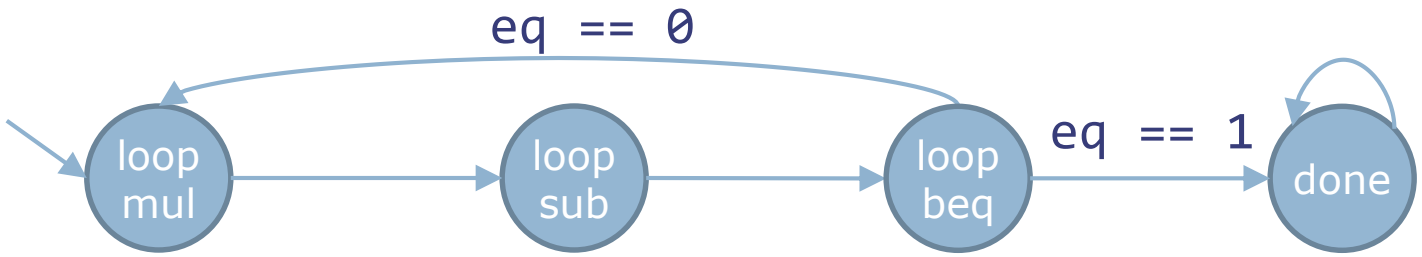
loop mul	loop sub	loop beq	done
asel = x1	asel = x2	asel = x2	asel = X
bsel = x2	bsel = x3	bsel = x4	bsel = X
opsel = 2 (*)	opsel = 0 (+)	opsel = X	opsel = X
wen = 1	wen = 1	wen = 0	wen = 0
wsel = x1	wsel = x2	wsel = X	wsel = X
x1 <= x1 * x2	x2 <= x2 + x3		N! in x1

A Control FSM for Factorial

- Assume initial register contents:

```
x1 value = 1
x2 value = N
x3 value = -1
x4 value = 0
```

- Control FSM:



loop mul	loop sub	loop beq	done
asel = x1	asel = x2	asel = x2	asel = X
bsel = x2	bsel = x3	bsel = x4	bsel = X
opsel = 2 (*)	opsel = 0 (+)	opsel = X	opsel = X
wen = 1	wen = 1	wen = 0	wen = 0
wsel = x1	wsel = x2	wsel = X	wsel = X
x1 <= x1 * x2	x2 <= x2 + x3		N! in x1

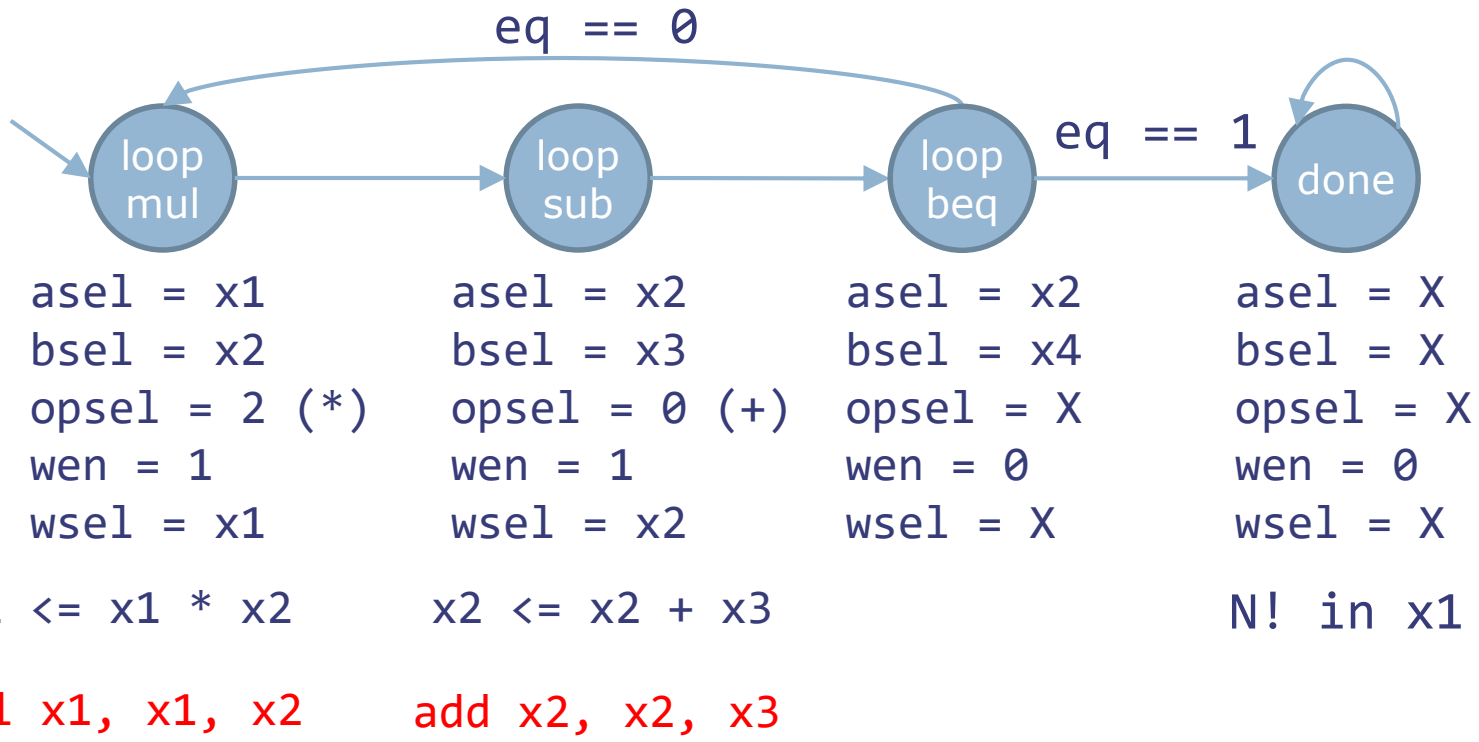
```
mul x1, x1, x2
```

A Control FSM for Factorial

- Assume initial register contents:

```
x1 value = 1
x2 value = N
x3 value = -1
x4 value = 0
```

- Control FSM:

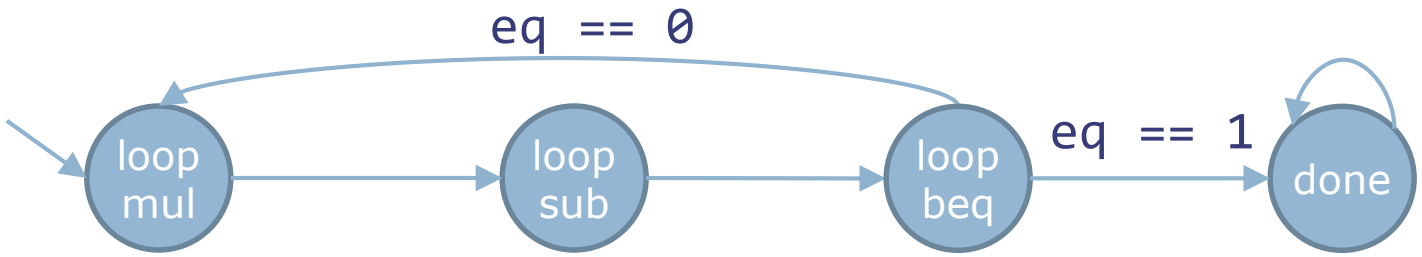


A Control FSM for Factorial

- Assume initial register contents:

```
x1 value = 1
x2 value = N
x3 value = -1
x4 value = 0
```

- Control FSM:



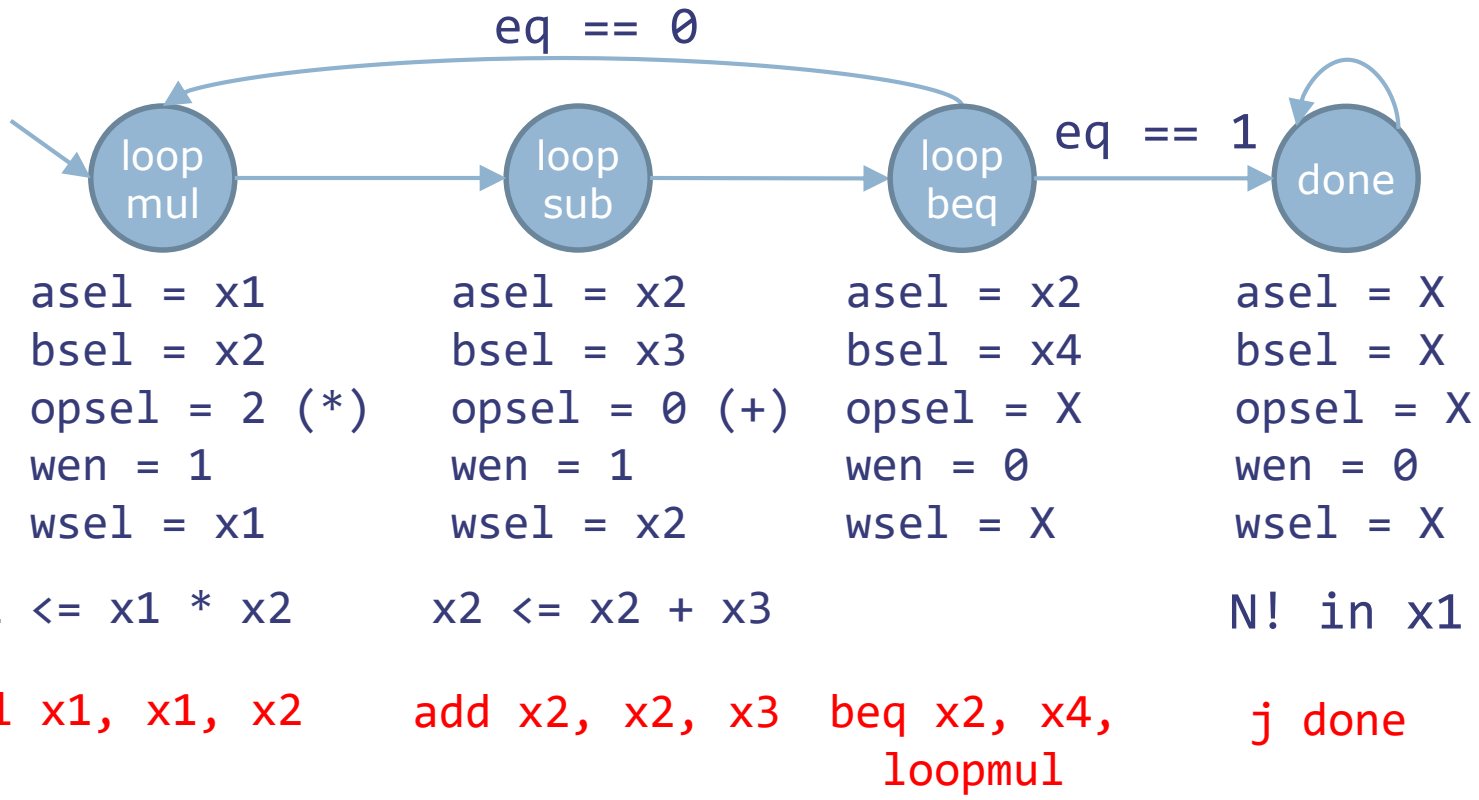
loop mul	loop sub	loop beq	done
asel = x1	asel = x2	asel = x2	asel = X
bsel = x2	bsel = x3	bsel = x4	bsel = X
opsel = 2 (*)	opsel = 0 (+)	opsel = X	opsel = X
wen = 1	wen = 1	wen = 0	wen = 0
wsel = x1	wsel = x2	wsel = X	wsel = X
x1 <= x1 * x2	x2 <= x2 + x3		N! in x1
mul x1, x1, x2	add x2, x2, x3	beq x2, x4, loopmul	

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

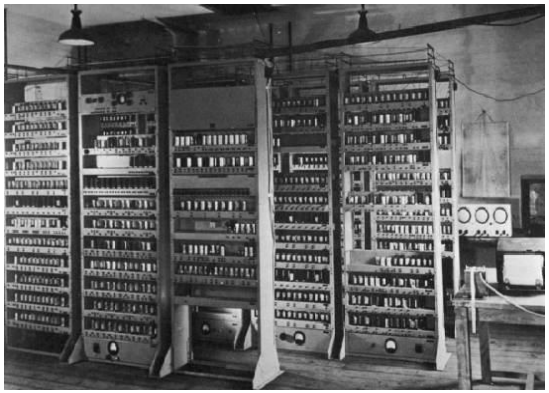
- You can solve many problems with this datapath!
  - GCD, Fibonacci, exponentiation, division, square root, …
  - But nothing that requires more than four registers

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

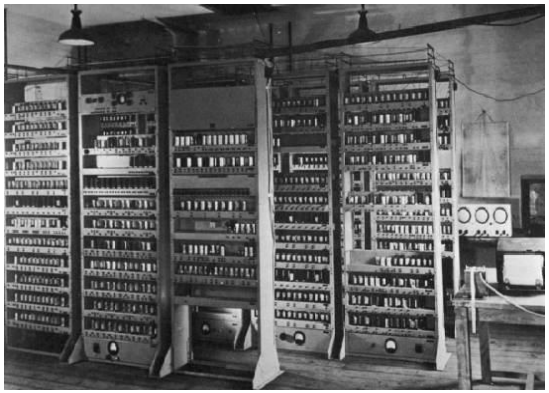
New Problem → New Control FSM

- 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



New Problem → New Control FSM

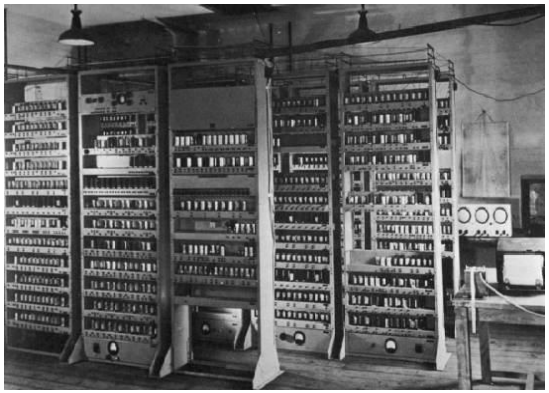
- You can solve many problems with this datapath!
  - GCD, Fibonacci, exponentiation, division, square root, …
  - But nothing that requires more than four registers
- By designing a control FSM, we are programming the datapath
- Early digital computers were programmed this way!
  - ENIAC (1943):
    - 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

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