Introduction to Pipelining

Lecture Goals

- Conclude the discussion on multi-cycle computations from last lecture

Lecture Goals

- Conclude the discussion on multi-cycle computations from last lecture
- Introduce pipelining, a technique that uses registers to improve the throughput of a circuit

Lecture Goals

- Conclude the discussion on multi-cycle computations from last lecture
- Introduce pipelining, a technique that uses registers to improve the throughput of a circuit
- Examine the tradeoffs of different design styles (combinational, pipelined, multi-cycle) through a multiplier case study

Reminder: Multi-Cycle Computations

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

Reminder: Multi-Cycle Computations

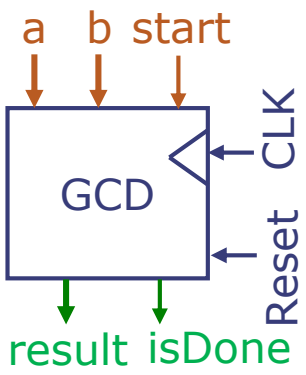
- Sequential circuits can implement more computations than combinational circuits
  - Variable amount of input/output
  - Variable number of steps

- Multi-cycle circuits implement a computation over multiple cycles
  - New computation is started by setting inputs at a particular cycle
  - Circuit takes several cycles (possibly a variable number) to finish computation, then makes output available
  - Circuit performs only one computation at a time; a new computation cannot begin until previous one has finished

Reminder: GCD Circuit

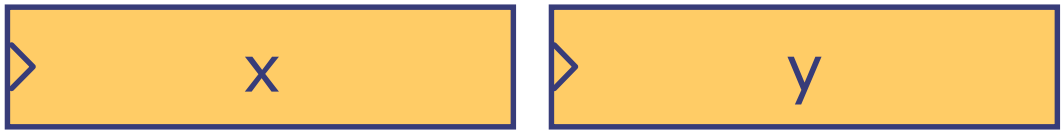
```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

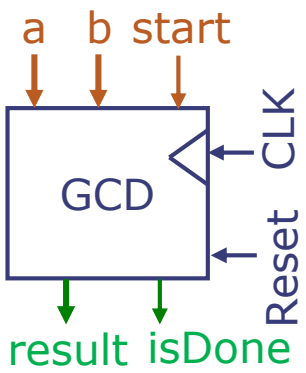
Reminder: GCD Circuit




```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

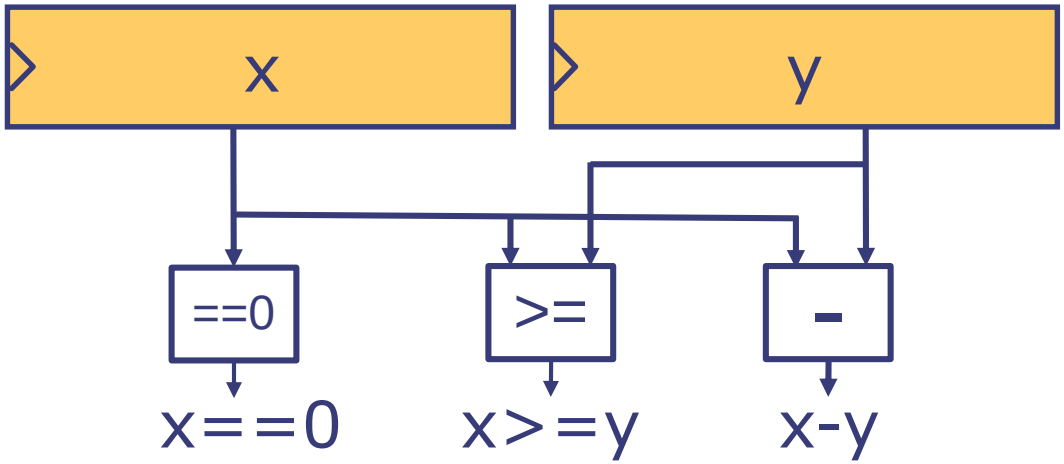
Reminder: GCD Circuit

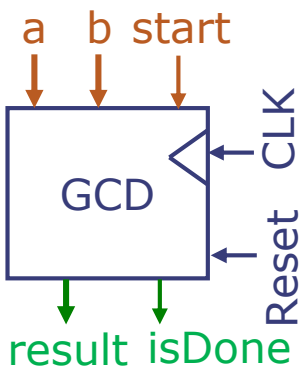





```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

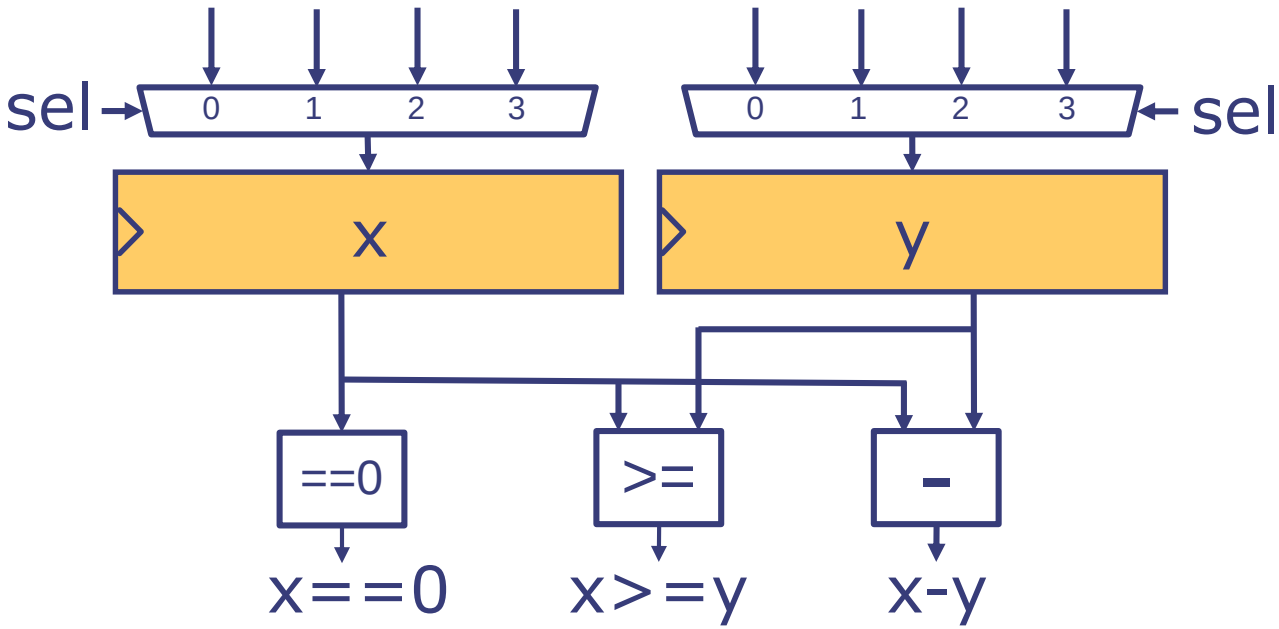
Reminder: GCD Circuit

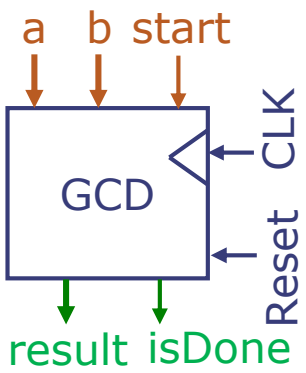





```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

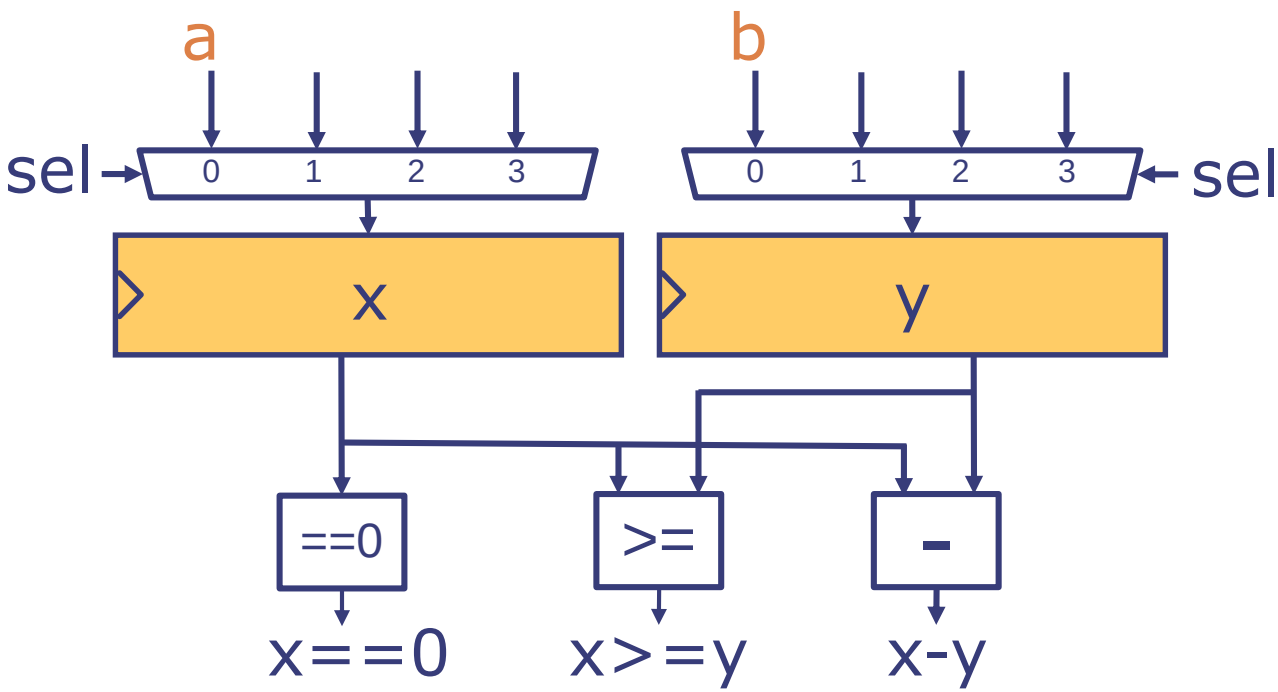
Reminder: GCD Circuit

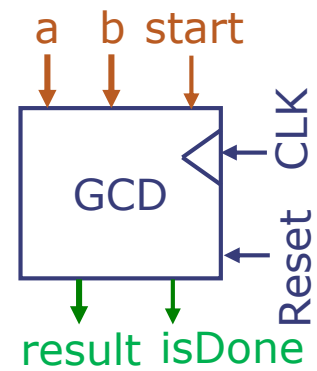





```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

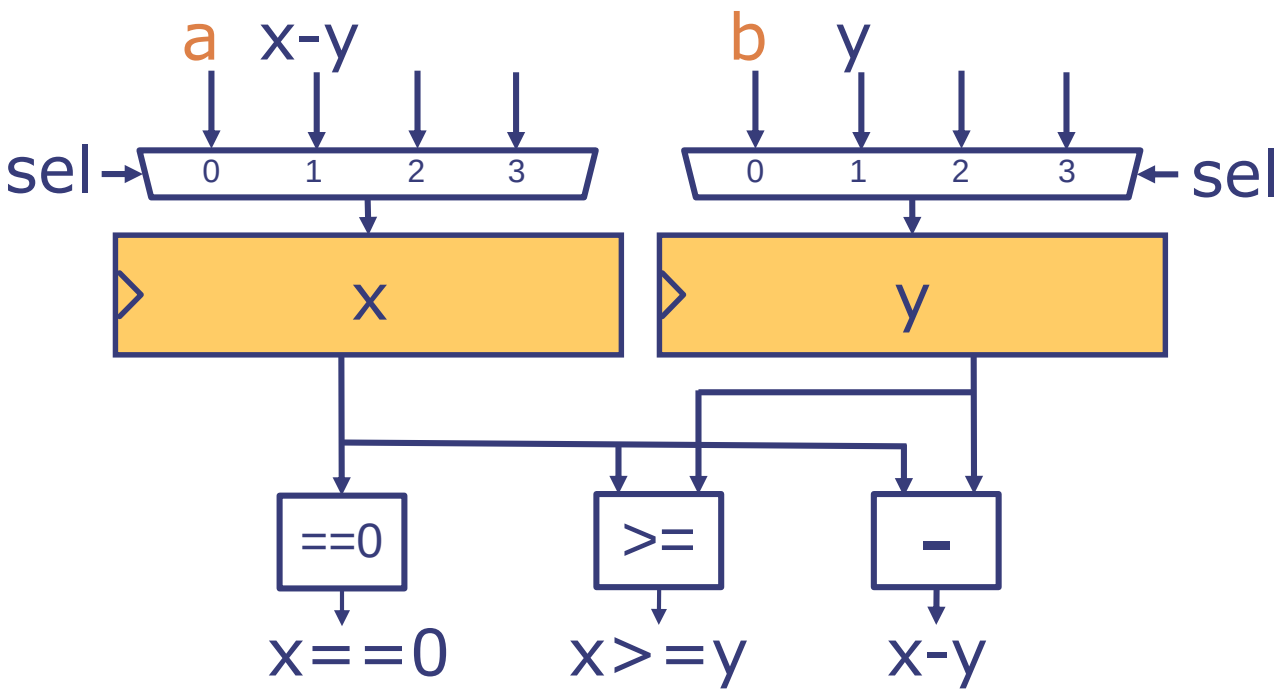
Reminder: GCD Circuit

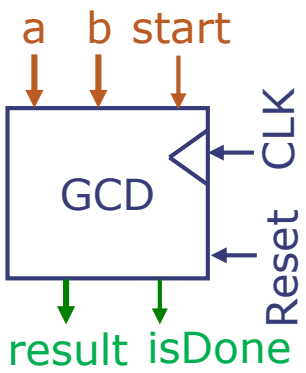





```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

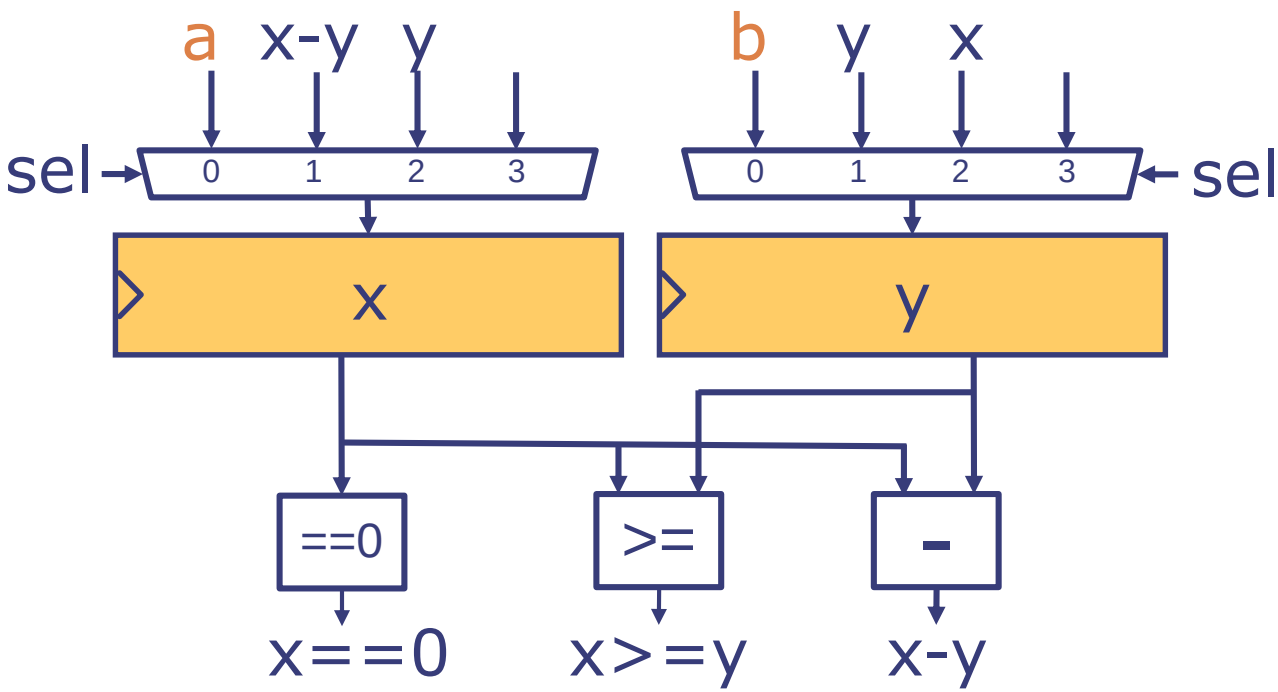
Reminder: GCD Circuit

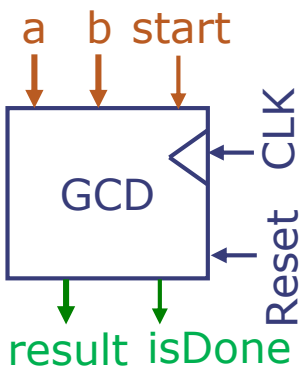





```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

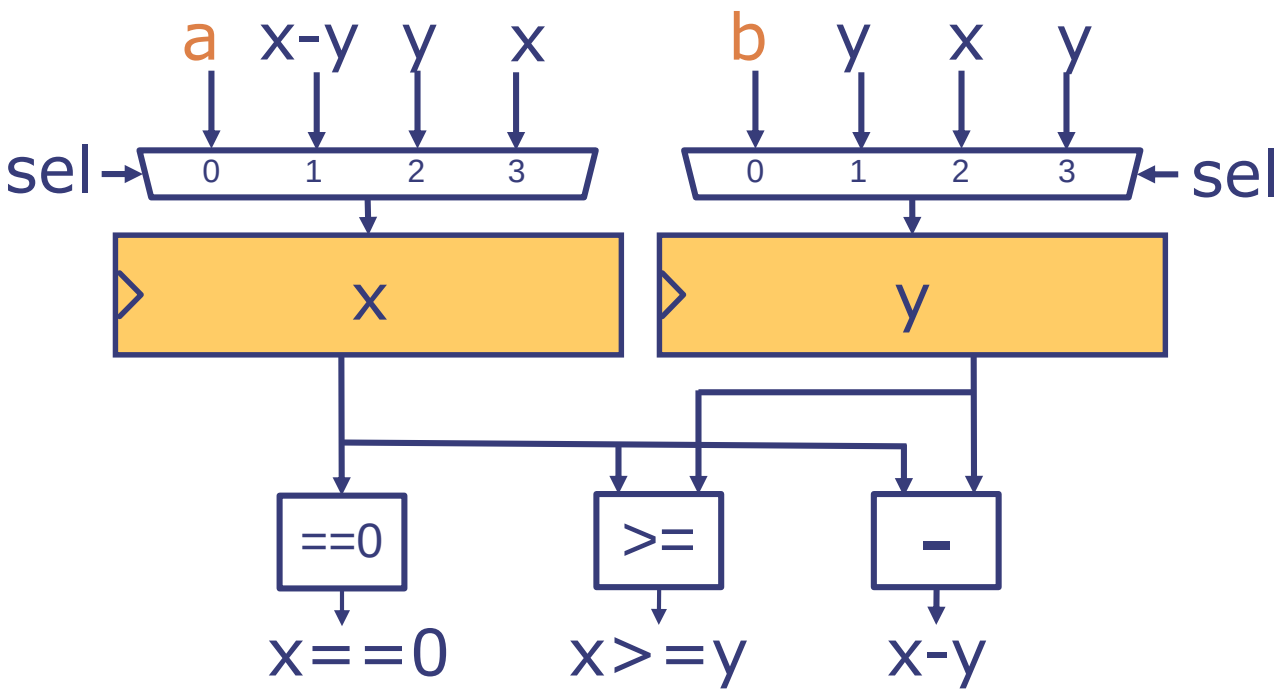
Reminder: GCD Circuit

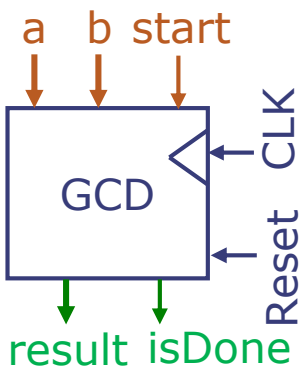





```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

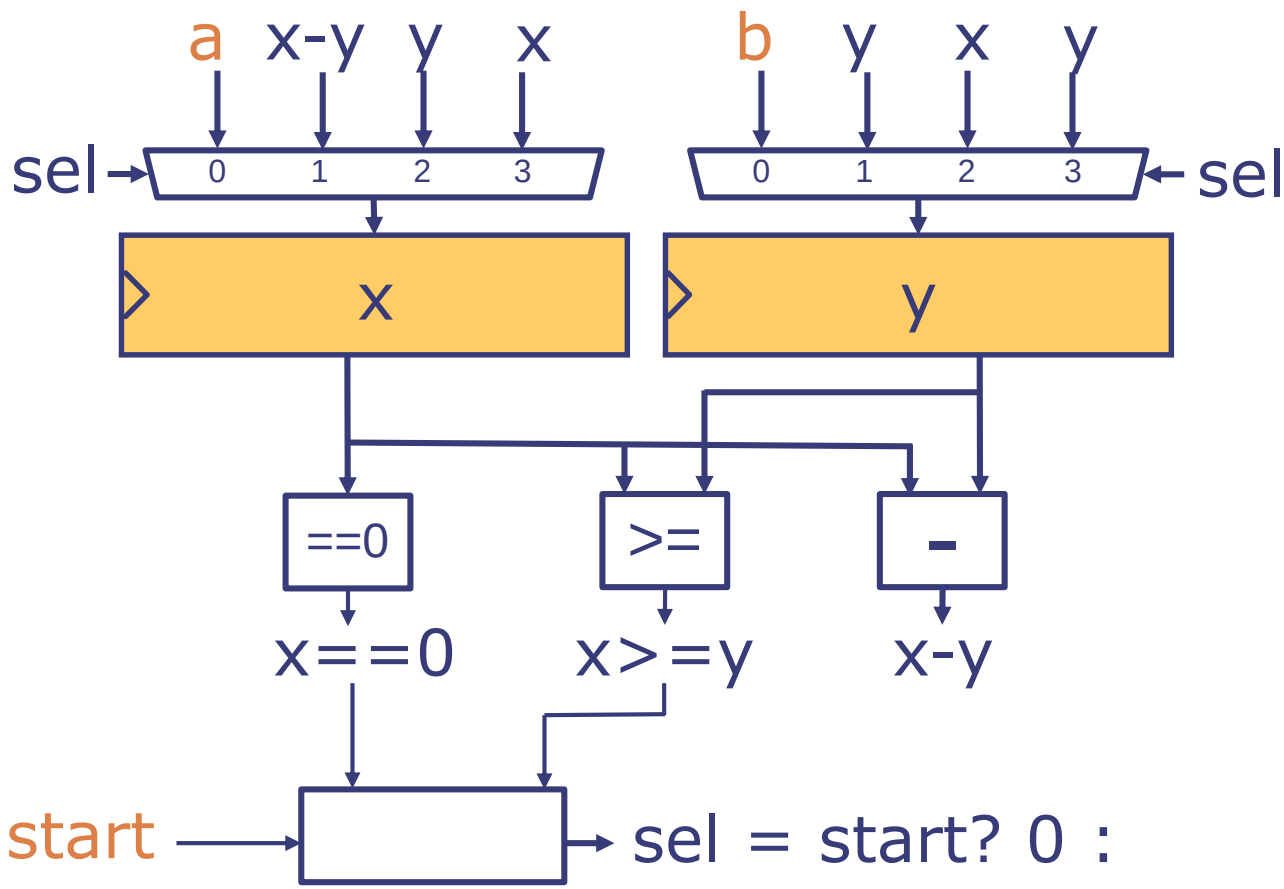
Reminder: GCD Circuit



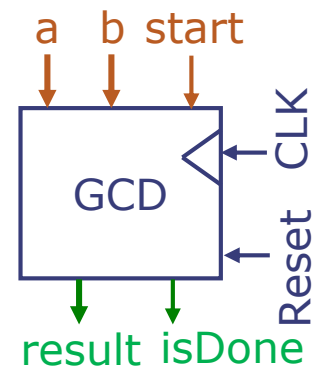



```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

Reminder: GCD Circuit

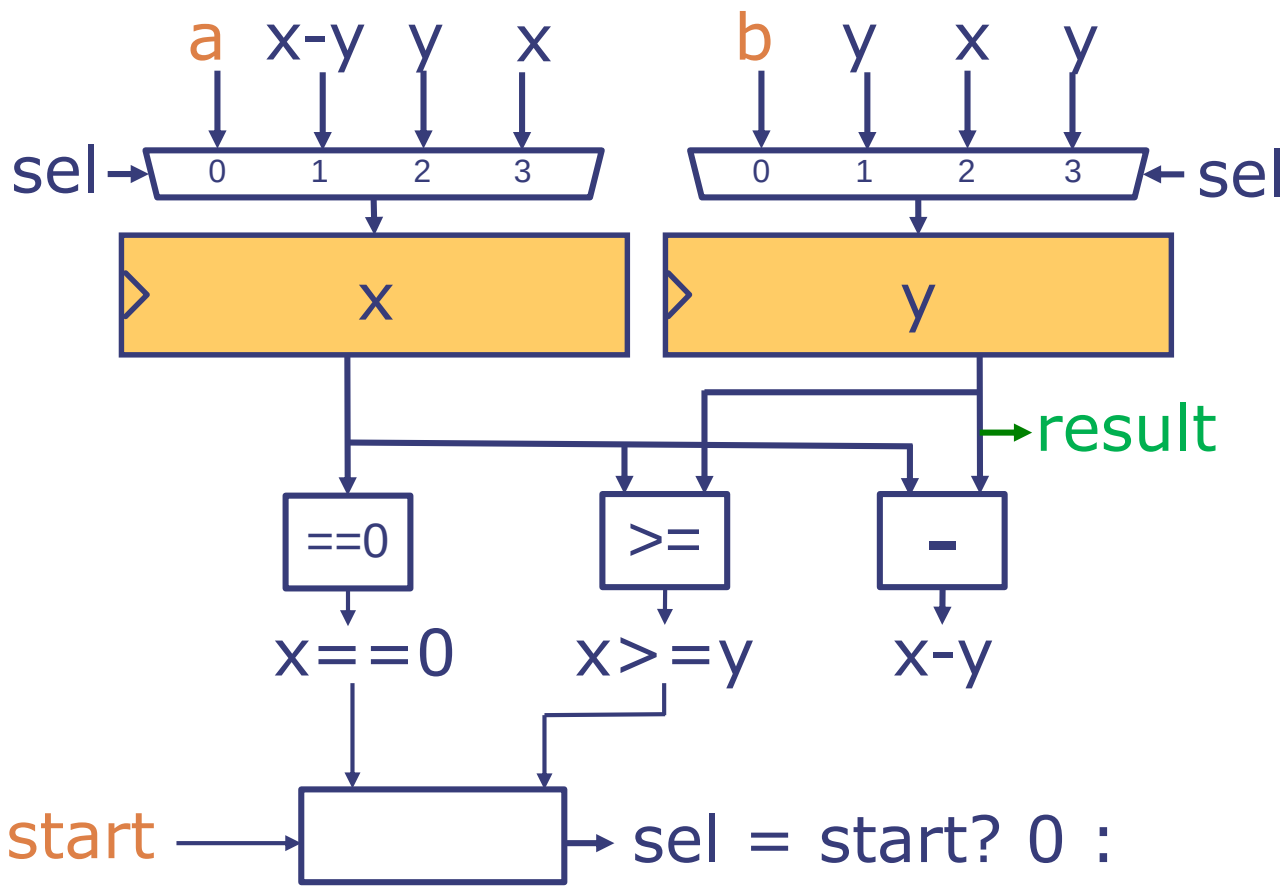


sel = start? 0 :
(x==0)? 3 :
(x>=y)? 1 : 2;

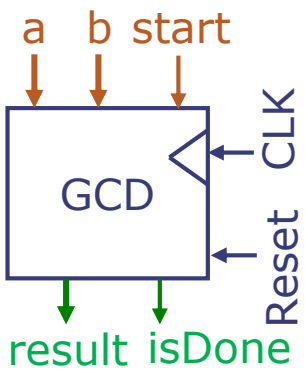



```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

Reminder: GCD Circuit

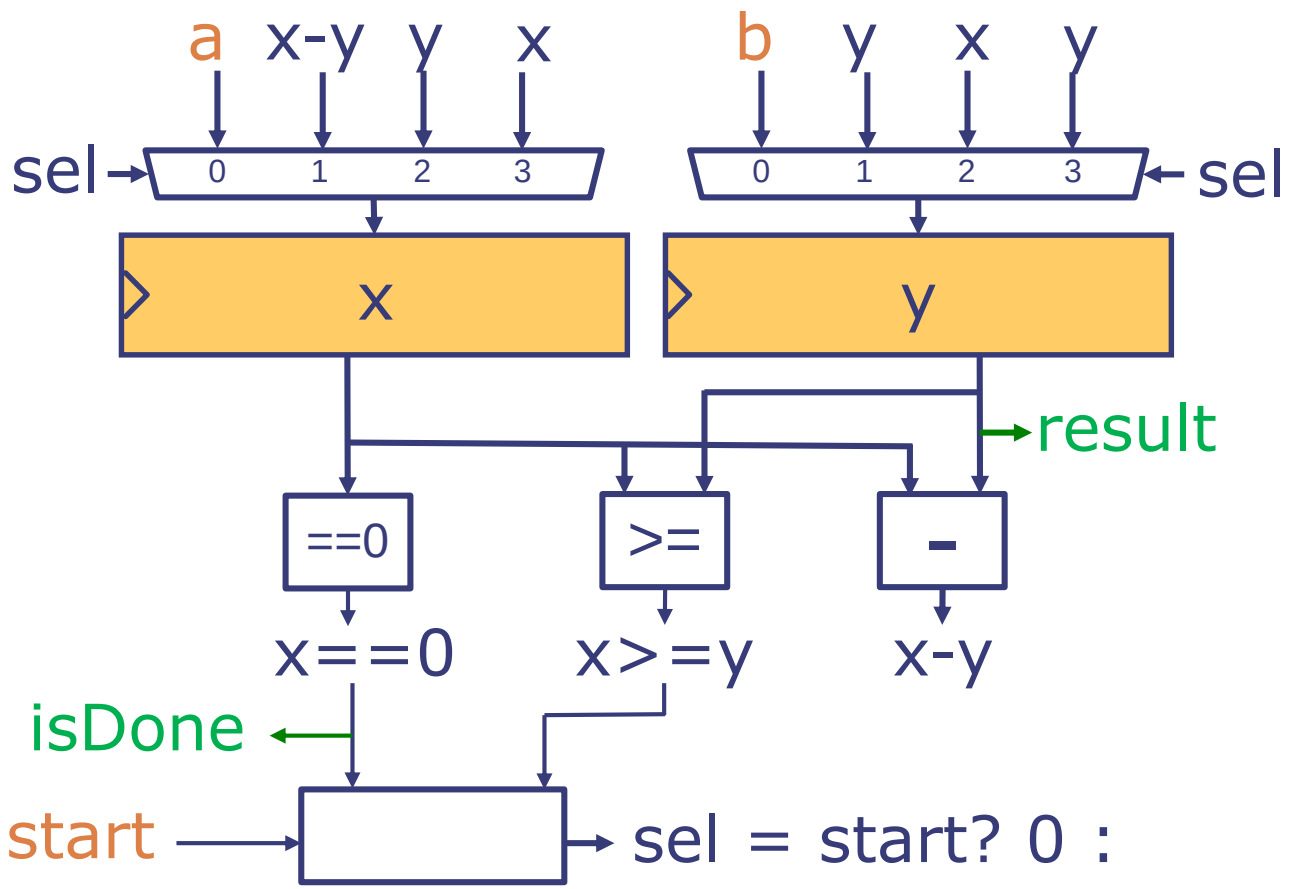


sel = start? 0 :
(x==0)? 3 :
(x>=y)? 1 : 2;

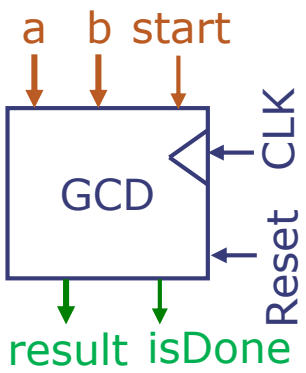



```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

Reminder: GCD Circuit



sel = start? 0 :
(x==0)? 3 :
(x>=y)? 1 : 2;




```
def gcd(a, b):
  x = a
  y = b
  while x != 0:
    if x >= y:
      x = x - y
    else:
      (x, y) = (y, x)
  return y
```

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;

endmodule
```

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);
```

```
endmodule
```

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

  input Bool start;
  input Word a;
  input Word b;

endmodule
```

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

  input Bool start;
  input Word a;
  input Word b;

  rule gcd;
    if (start) begin
      x <= a;  y <= b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule



endmodule
```

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

  input Bool start;
  input Word a;
  input Word b;

  rule gcd;
    if (start) begin
      x <= a;  y <= b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule

  method Word result = y;
  method Bool isDone = (x == 0);
endmodule
```

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

  input Bool start;
  input Word a;
  input Word b;

  rule gcd;
    if (start) begin
      x <= a;  y <= b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule

  method Word result = y;
  method Bool isDone = (x == 0);
endmodule
```

Poor interface: Several inputs and outputs are closely coupled

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

  input Bool start;
  input Word a;
  input Word b;

  rule gcd;
    if (start) begin
      x <= a;  y <= b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule

  method Word result = y;
  method Bool isDone = (x == 0);
endmodule
```

Poor interface: Several inputs and outputs are closely coupled

- New GCD computation is started by setting *start* input to True and passing arguments through inputs *a* and *b*

GCD in Minispec
First version

```
typedef Bit#(32) Word;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

  input Bool start;
  input Word a;
  input Word b;

  rule gcd;
    if (start) begin
      x <= a;  y <= b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule

  method Word result = y;
  method Bool isDone = (x == 0);
endmodule
```

Poor interface: Several inputs and outputs are closely coupled

- New GCD computation is started by setting *start* input to True and passing arguments through inputs *a* and *b*
- Several cycles later, the module signals that it has finished by having *isDone* return True; only then, the *result* method returns the correct result for gcd(a,b)

Designing Good Module Interfaces

- The previous GCD module has a poor interface

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isDone and read wrong result!*

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isDone and read wrong result!*
  - Tedious to use. *Why?*

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isDone and read wrong result!*
  - Tedious to use. *Why?*
    - *e.g., if start is False, we still have to set the a and b inputs, even though they are not used!*

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isDone and read wrong result!*
  - Tedious to use. *Why?*
    - *e.g., if start is False, we still have to set the a and b inputs, even though they are not used!*

- To design good interfaces,
  group related inputs and outputs

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isDone and read wrong result!*
  - Tedious to use. *Why?*
    - *e.g., if start is False, we still have to set the a and b inputs, even though they are not used!*

- To design good interfaces,
  group related inputs and outputs
  - In our case, GCD should have:
    - A single output that is either invalid or a valid result
    - A single input that is either no arguments or arguments

Designing Good Module Interfaces

- The previous GCD module has a poor interface
  - Easy to misuse. *Why?*
    - *e.g., may forget to check isDone and read wrong result!*
  - Tedious to use. *Why?*
    - *e.g., if start is False, we still have to set the a and b inputs, even though they are not used!*

- To design good interfaces,
  group related inputs and outputs
  - In our case, GCD should have:
    - A single output that is either invalid or a valid result
    - A single input that is either no arguments or arguments
  - This requires we learn about one last type…

The Maybe Type

- Maybe#(T) represents an optional value of type T
  - Either Invalid and no value, or Valid and a value

The Maybe Type

- Maybe#(T) represents an optional value of type T
  - Either Invalid and no value, or Valid and a value

- Possible implementation: A value + a valid bit

```
typedef struct { Bool valid; T value; } Maybe#(type T);
```

The Maybe Type

- Maybe#(T) represents an optional value of type T
  - Either Invalid and no value, or Valid and a value

- Possible implementation: A value + a valid bit

```
typedef struct { Bool valid; T value; } Maybe#(type T);
```

  - Although we could implement our own, optional values are so common that Maybe#(T) has a few built-in operations

The Maybe Type

- Maybe#(T) represents an optional value of type T
  - Either Invalid and no value, or Valid and a value

- Possible implementation: A value + a valid bit

```
typedef struct { Bool valid; T value; } Maybe#(type T);
```

  - Although we could implement our own, optional values are so common that Maybe#(T) has a few built-in operations

```
Maybe#(Word) x = Invalid;    // no need to give value!
Maybe#(Word) y = Valid(42);  // must specify a value
```

The Maybe Type

- Maybe#(T) represents an optional value of type T
  - Either Invalid and no value, or Valid and a value

- Possible implementation: A value + a valid bit

```
typedef struct { Bool valid; T value; } Maybe#(type T);
```

  - Although we could implement our own, optional values are so common that Maybe#(T) has a few built-in operations

```
Maybe#(Word) x = Invalid;     // no need to give value!
Maybe#(Word) y = Valid(42);   // must specify a value

if (isValid(y))               // check validity
  Word z = fromMaybe(?, y);   // extract valid value
```

Improved GCD Module
Using Maybe Types

```
typedef struct {Word a; Word b;} GCDArgs;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);

endmodule
```

Improved GCD Module
Using Maybe Types

```
typedef struct {Word a; Word b;} GCDArgs;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);
  input Maybe#(GCDArgs) in;

endmodule
```

Improved GCD Module
Using Maybe Types

```
typedef struct {Word a; Word b;} GCDArgs;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);
  input Maybe#(GCDArgs) in;
  rule gcd;
    if (isValid(in)) begin
      let args = fromMaybe(?, in);
      x <= args.a;  y <= args.b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule


endmodule
```

Improved GCD Module
Using Maybe Types

```
typedef struct {Word a; Word b;} GCDArgs;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);
  input Maybe#(GCDArgs) in;
  rule gcd;
    if (isValid(in)) begin
      let args = fromMaybe(?, in);
      x <= args.a;  y <= args.b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule
  method Maybe#(Word) result =
    (x == 0)? Valid(y) : Invalid;
endmodule
```

Improved GCD Module
Using Maybe Types

```
typedef struct {Word a; Word b;} GCDArgs;
module GCD;
  Reg#(Word) x(1);
  Reg#(Word) y(0);
  input Maybe#(GCDArgs) in;
  rule gcd;
    if (isValid(in)) begin
      let args = fromMaybe(?, in);
      x <= args.a;  y <= args.b;
    end else if (x != 0) begin
      if (x >= y) begin // subtract
        x <= x - y;
      end else begin    // swap
        x <= y;  y <= x;
      end
    end
  endrule
  method Maybe#(Word) result =
    (x == 0)? Valid(y) : Invalid;
endmodule
```

Single input and output:

- New GCD computation is started by setting a Valid input *in* (which always includes a and b)
- When GCD computation finishes, *result* becomes a Valid output

Pipelined Circuits

Performance Measures

- Two timing metrics of interest when designing a system:

Performance Measures

- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

Performance Measures

- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

Performance Measures

- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

- The metric to prioritize depends on the application

Performance Measures

- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

- The metric to prioritize depends on the application
  - *Airbag deployment system?*

Performance Measures

- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

- The metric to prioritize depends on the application
  - *Airbag deployment system?* Latency

Performance Measures

- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

- The metric to prioritize depends on the application
  - *Airbag deployment system?* Latency
  - *General-purpose processor?*

Performance Measures

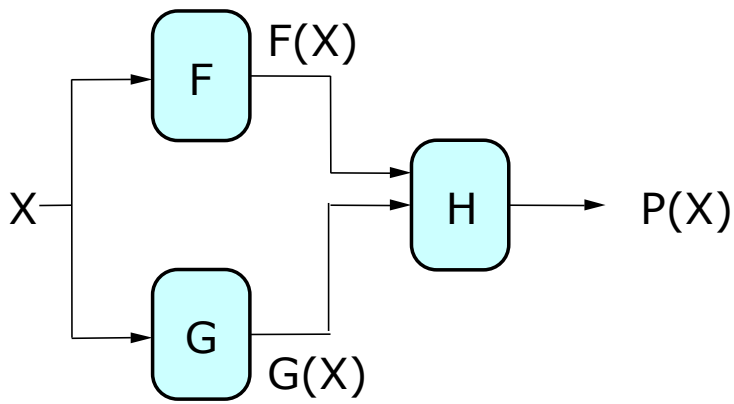
- Two timing metrics of interest when designing a system:

1. Latency: The *delay* from when an input enters the system until its associated output is produced

2. Throughput: The *rate* at which inputs or outputs are processed

- The metric to prioritize depends on the application
  - *Airbag deployment system?* Latency
  - *General-purpose processor?* Throughput (maximize instructions/second)

Performance of Combinational Logic

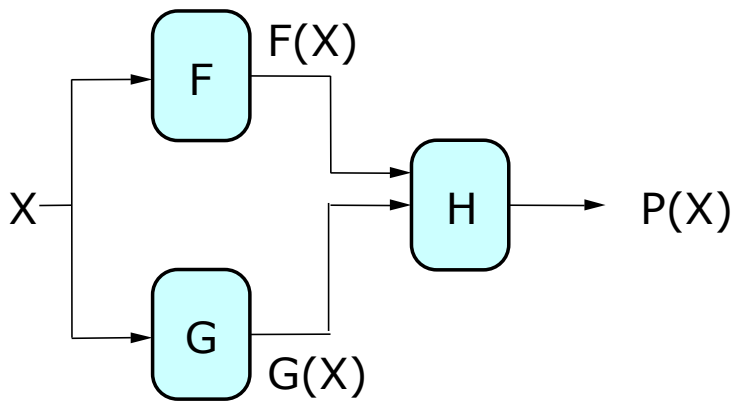


For combinational logic:

latency = t_{PD}

throughput = $1/t_{PD}$

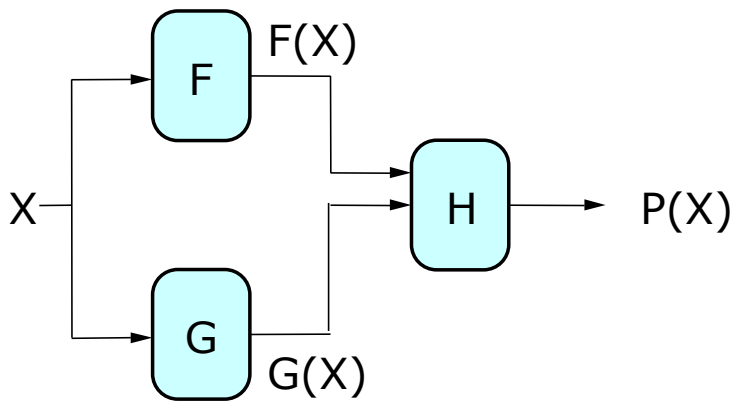
Performance of Combinational Logic



For combinational logic:
- latency = t_{PD}
- throughput = $1/t_{PD}$

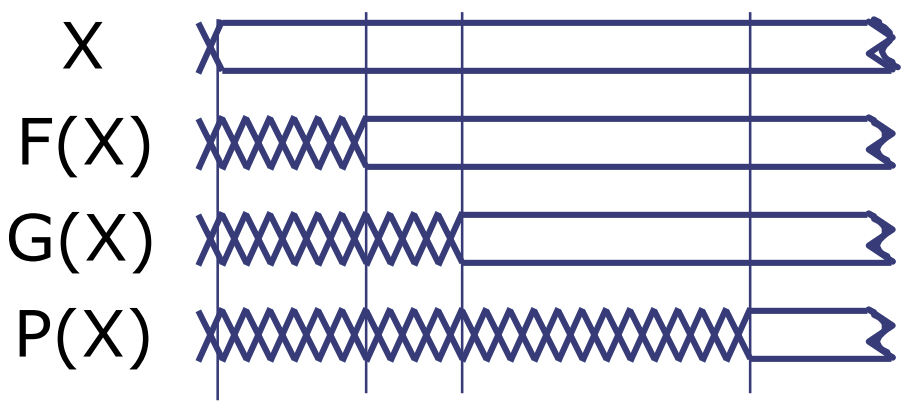
We can't get the answer any faster, but are we making effective use of our hardware at all times?

Performance of Combinational Logic

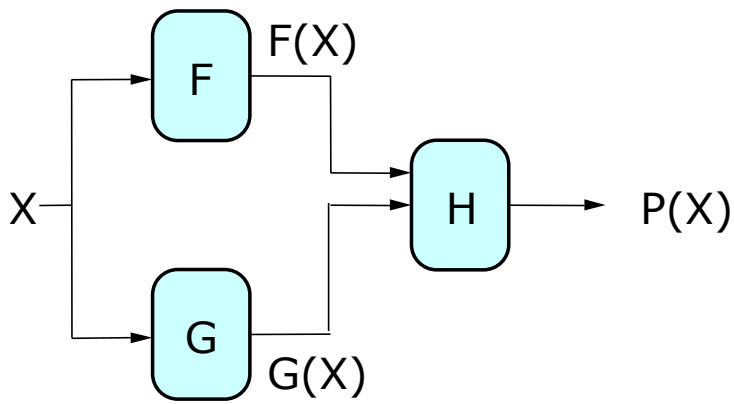


For combinational logic:
latency = t_{PD}
throughput = $1/t_{PD}$

We can't get the answer any faster, but are we making effective use of our hardware at all times?

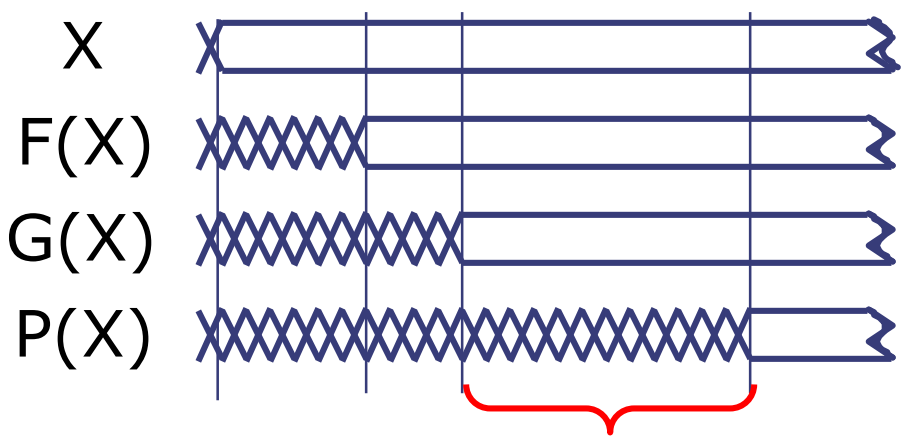


Performance of Combinational Logic



For combinational logic:
latency = t_{PD}
throughput = $1/t_{PD}$

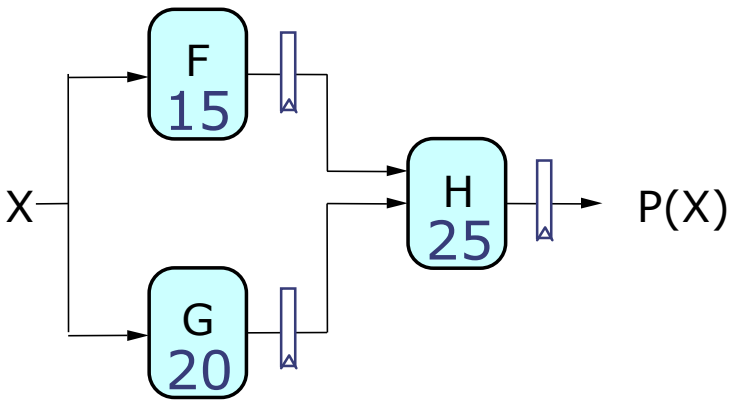
We can't get the answer any faster, but are we making effective use of our hardware at all times?



F & G are "idle", just holding their outputs stable while H performs its computation

Pipelined Circuits

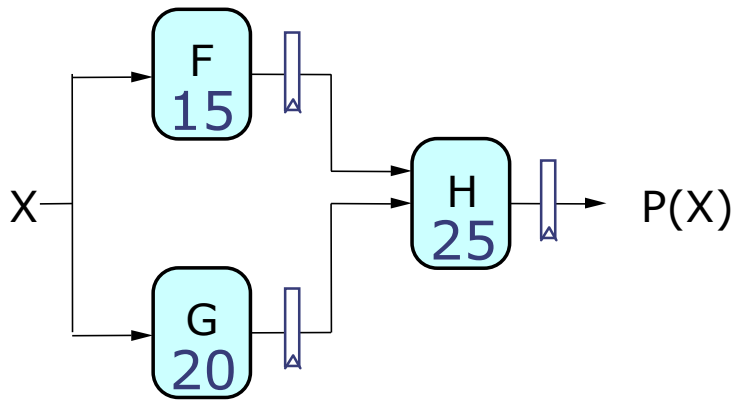
Use registers to hold H's input stable!



Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Pipelined Circuits

Use registers to hold H's input stable!



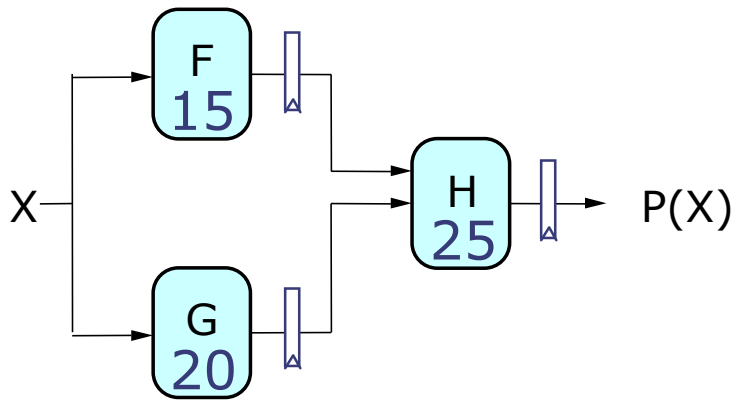
Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers (t_{PD} = 0, t_{SETUP} = 0):

	latency	throughput
unpipelined	45 ns	1/(45 ns)
2-stage pipeline	_____	_____

Pipelined Circuits

Use registers to hold H's input stable!



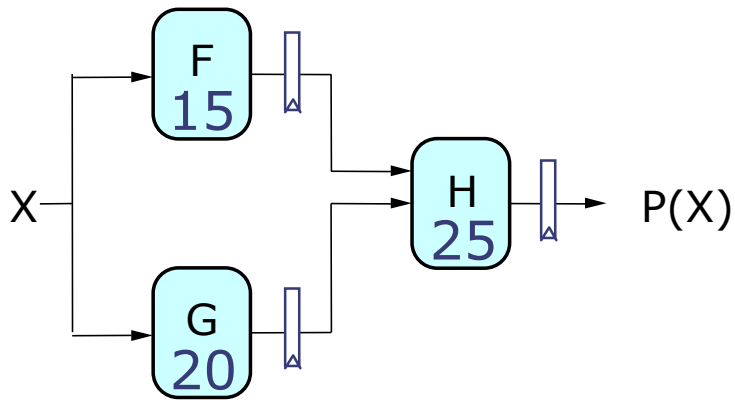
Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers (t_{PD} = 0, t_{SETUP} = 0):

	latency	throughput
unpipelined	45 ns	1/(45 ns)
2-stage pipeline	50 ns	_____

Pipelined Circuits

Use registers to hold H's input stable!



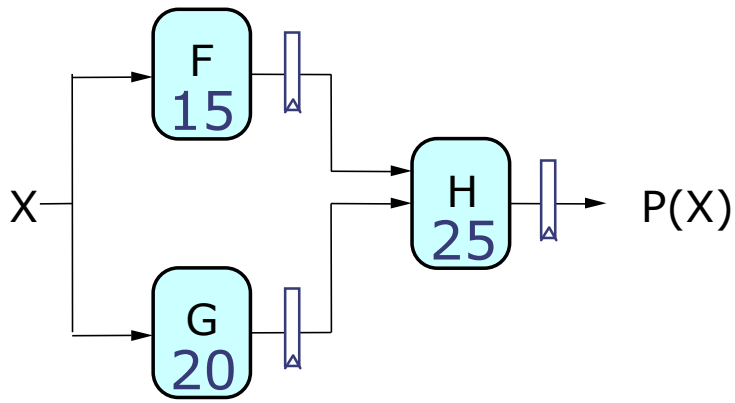
Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers (t_{PD} = 0, t_{SETUP} = 0):

	latency	throughput
unpipelined	45 ns	1/(45 ns)
2-stage pipeline	50 ns	_____
	worse	

Pipelined Circuits

Use registers to hold H's input stable!



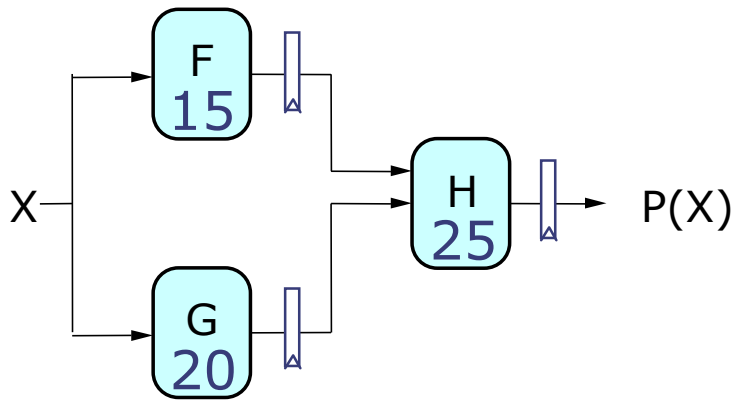
Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers (t_{PD} = 0, t_{SETUP} = 0):

	latency	throughput
unpipelined	45 ns	1/(45 ns)
2-stage pipeline	50 ns	1/(25 ns)
	worse	

Pipelined Circuits

Use registers to hold H's input stable!

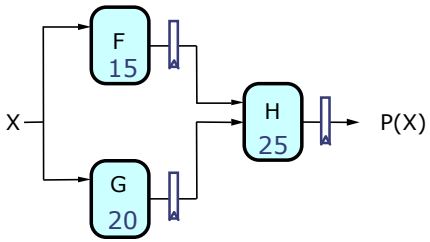


Now F & G can be working on input X_{i+1} while H is performing its computation on X_i. We've created a 2-stage *pipeline*: if we have a valid input X during clock cycle j, P(X) is valid during clock j+2.

Suppose F, G, H have propagation delays of 15, 20, 25 ns and we are using ideal registers (t_{PD} = 0, t_{SETUP} = 0):

	latency	throughput
unpipelined	45 ns	1/(45 ns)
2-stage pipeline	50 ns	1/(25 ns)
	worse	better!

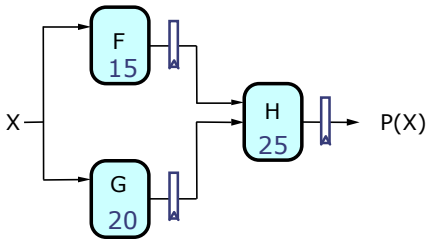
Pipeline Diagrams



Clock cycle →

Pipeline stages ↓	i	i+1	i+2	i+3	
F & G					
H					

Pipeline Diagrams

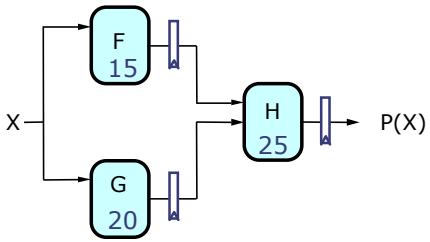


Clock cycle →

Pipeline stages ↓	i	i+1	i+2	i+3
F & G				
H				

Xi stable here

Pipeline Diagrams

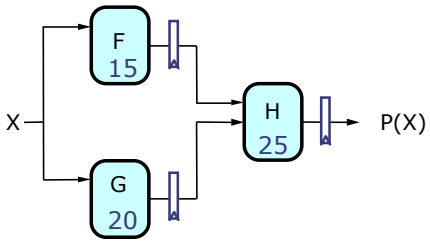


Clock cycle →

Pipeline stages ↓	i	i+1	i+2	i+3
F & G	$F(X_i)$ $G(X_i)$			
H				

Xi stable here

Pipeline Diagrams



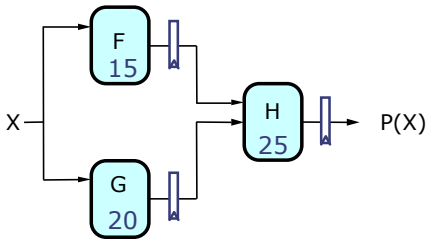
Clock cycle →

Pipeline stages ↓

	i	i+1	i+2	i+3
F & G	$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$		
H		$H(X_i)$		

Xi stable here

Pipeline Diagrams



Clock cycle →

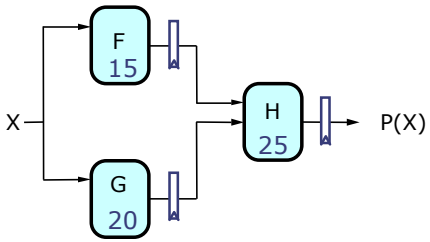
Pipeline stages ↓

	i	i+1	i+2	i+3
F & G	$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$		
H		$H(X_i)$		

Xi stable here

P(Xi) available here

Pipeline Diagrams



Clock cycle →

Pipeline stages ↓

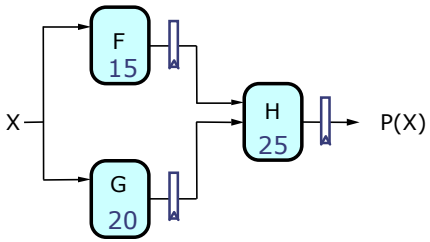
	i	i+1	i+2	i+3	
F & G	$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$			
H		$H(X_i)$			

Xi stable here

P(Xi) available here

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Diagrams



Clock cycle →

Pipeline stages ↓

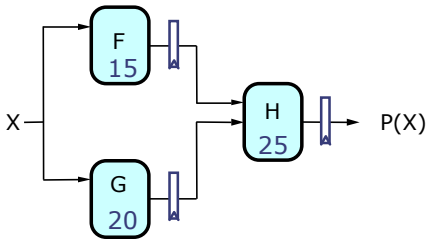
	i	i+1	i+2	i+3	
F & G	$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$	$F(X_{i+2})$ $G(X_{i+2})$		
H		$H(X_i)$	$H(X_{i+1})$		

Xi stable here

P(Xi) available here

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Diagrams



Clock cycle →

Pipeline stages ↓

	i	i+1	i+2	i+3
F & G	$F(X_i)$ $G(X_i)$	$F(X_{i+1})$ $G(X_{i+1})$	$F(X_{i+2})$ $G(X_{i+2})$	…
H		$H(X_i)$	$H(X_{i+1})$	$H(X_{i+2})$

Xi stable here

P(Xi) available here

The results associated with a particular set of input data moves *diagonally* through the diagram, progressing through one pipeline stage each clock cycle.

Pipeline Conventions

Definition:

A well-formed *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Pipeline Conventions

Definition:

A well-formed *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Composition convention:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *output* (not on its input).

Pipeline Conventions

Definition:

A well-formed *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Composition convention:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *output* (not on its input).

Clock period:

The clock must have a period t_{CLK} sufficient to cover the longest register to register propagation delay plus setup time.

Pipeline Conventions

Definition:

A well-formed *K-Stage Pipeline* ("K-pipeline") is an acyclic circuit having exactly K registers on *every* path from an input to an output.

A combinational circuit is thus a 0-stage pipeline.

Composition convention:

Every pipeline stage, hence every K-Stage pipeline, has a register on its *output* (not on its input).

Clock period:

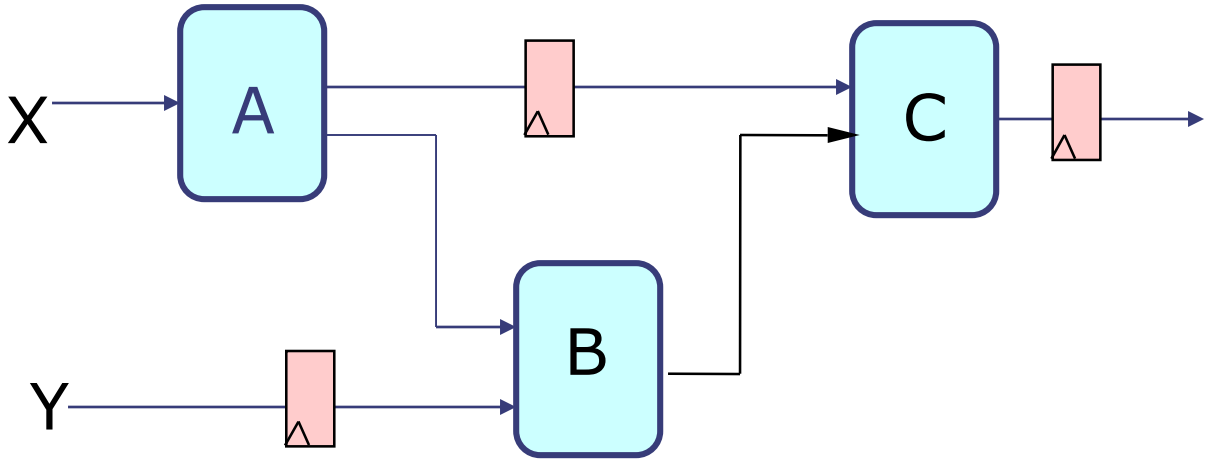
The clock must have a period t_{CLK} sufficient to cover the longest register to register propagation delay plus setup time.

K-pipeline latency $L = K * t_{CLK}$

K-pipeline throughput $T = 1 / t_{CLK}$

Ill-Formed Pipelines

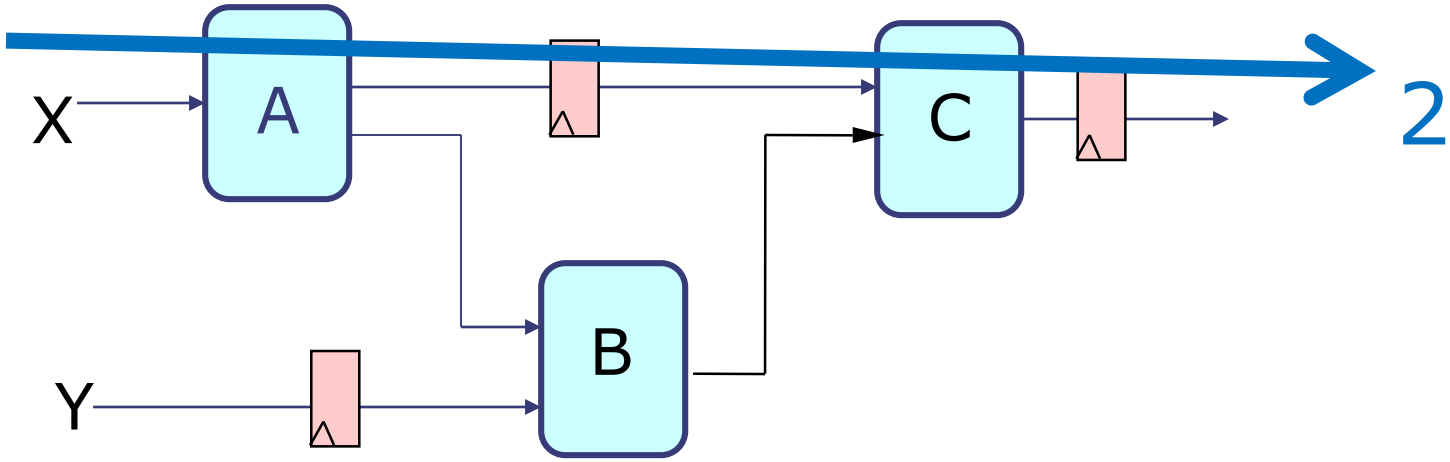
Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

Ill-Formed Pipelines

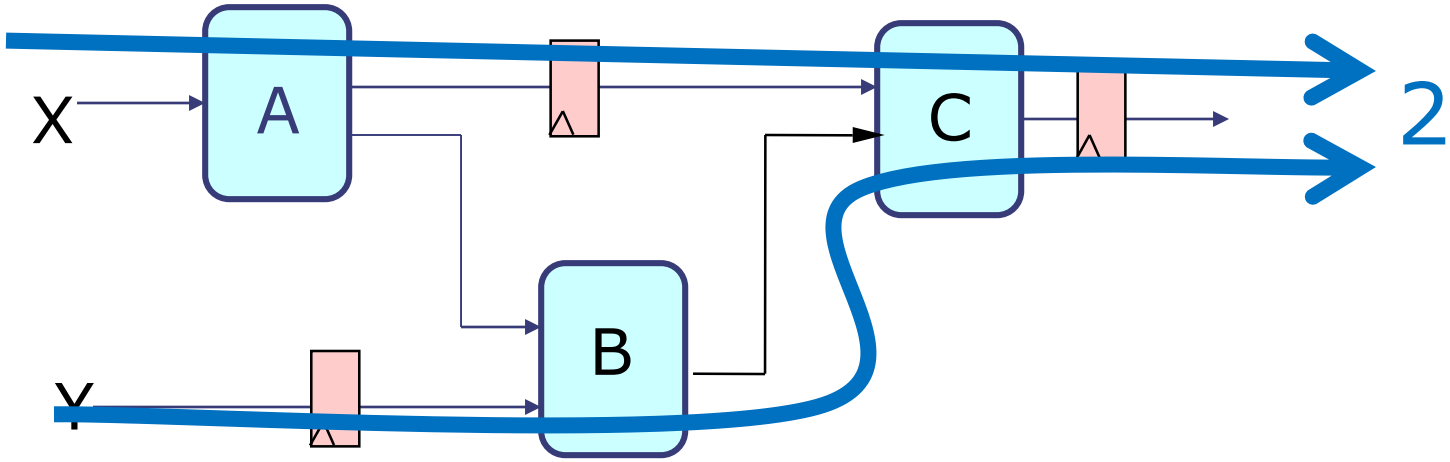
Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

Ill-Formed Pipelines

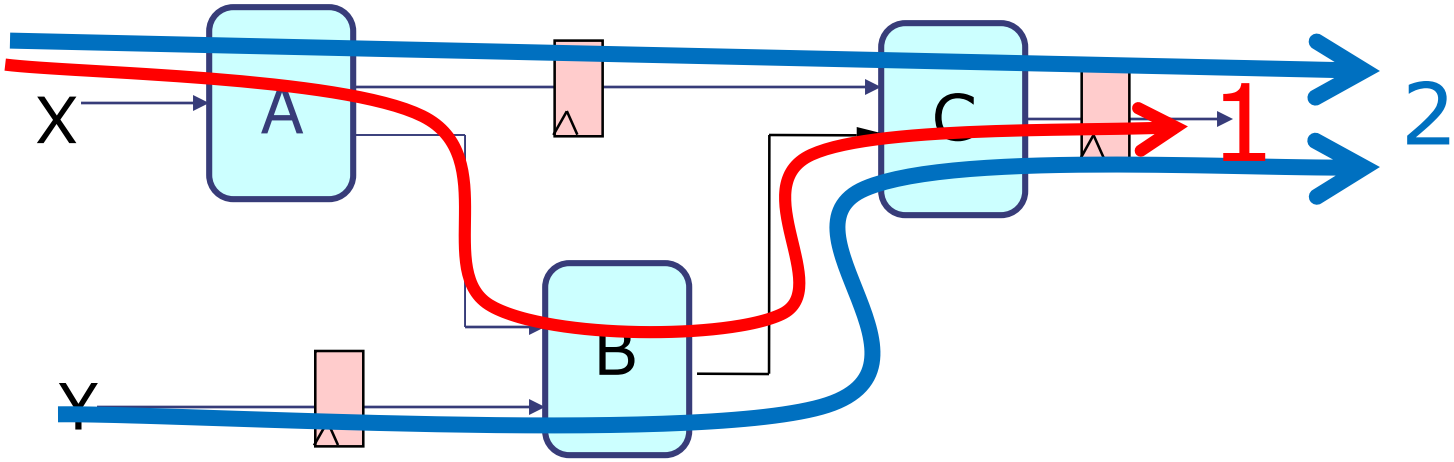
Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

Ill-Formed Pipelines

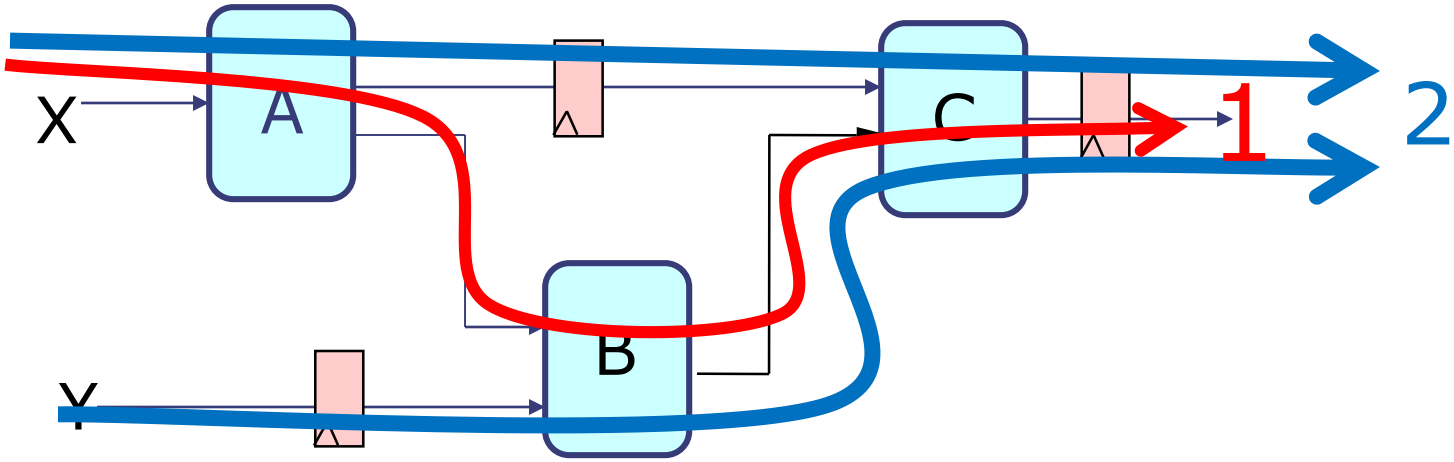
Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline?

Ill-Formed Pipelines

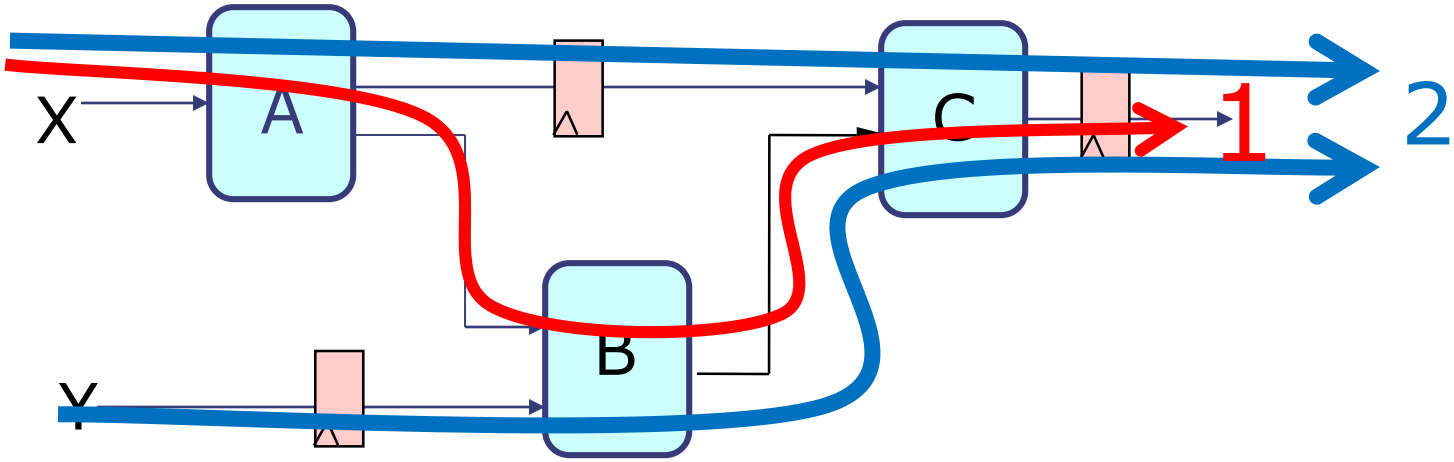
Consider a BAD job of pipelining:



For what value of K is the following circuit a K-Pipeline? *none*

Ill-Formed Pipelines

Consider a BAD job of pipelining:



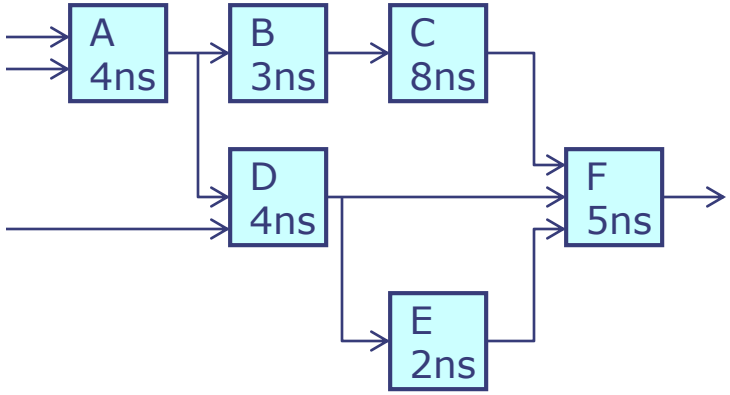
For what value of K is the following circuit a K-Pipeline? *none*

Problem:

Successive inputs get mixed: e.g., $B(A(X_{i+1}), Y_i)$. This happens because some paths from inputs to outputs have 2 registers, and some have only 1!

This can't happen in a well-formed K pipeline!

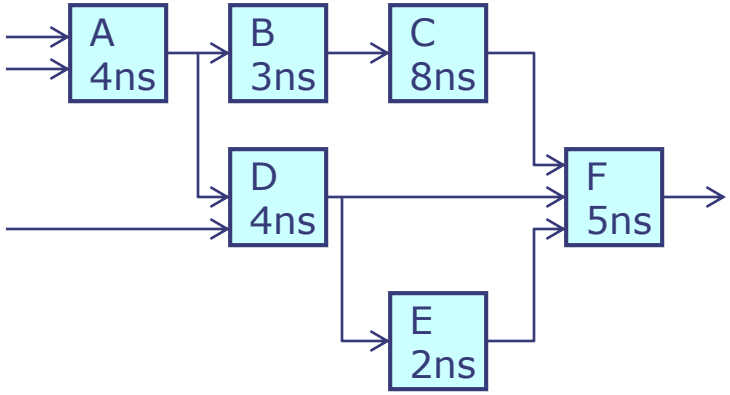
A Pipelining Methodology



A Pipelining Methodology

Step 1:

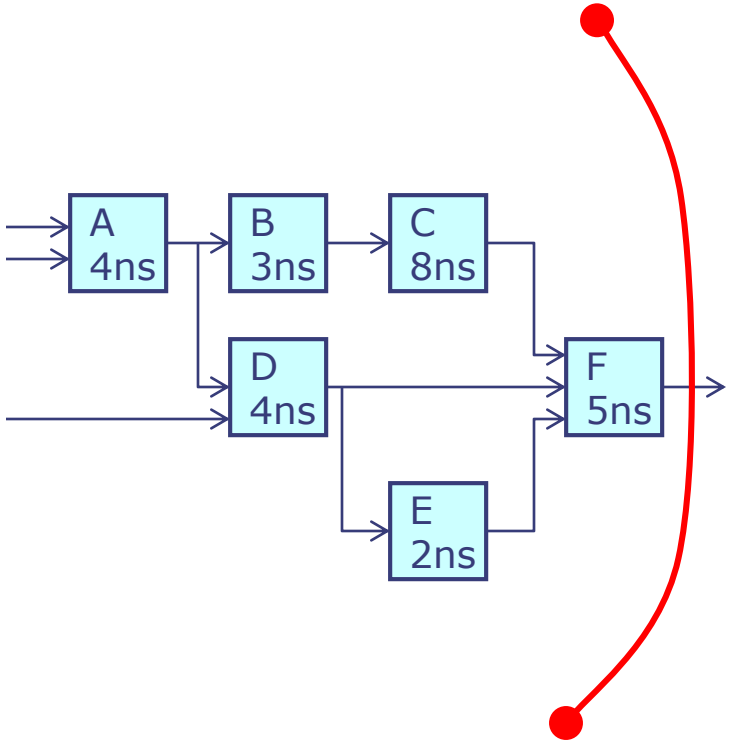
Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.



A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.



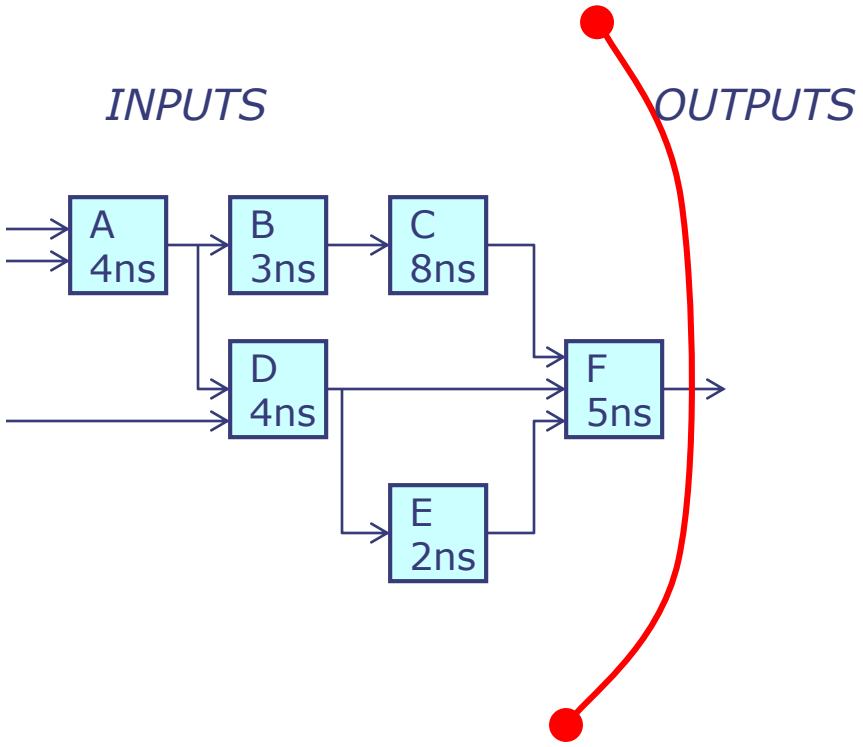
A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.



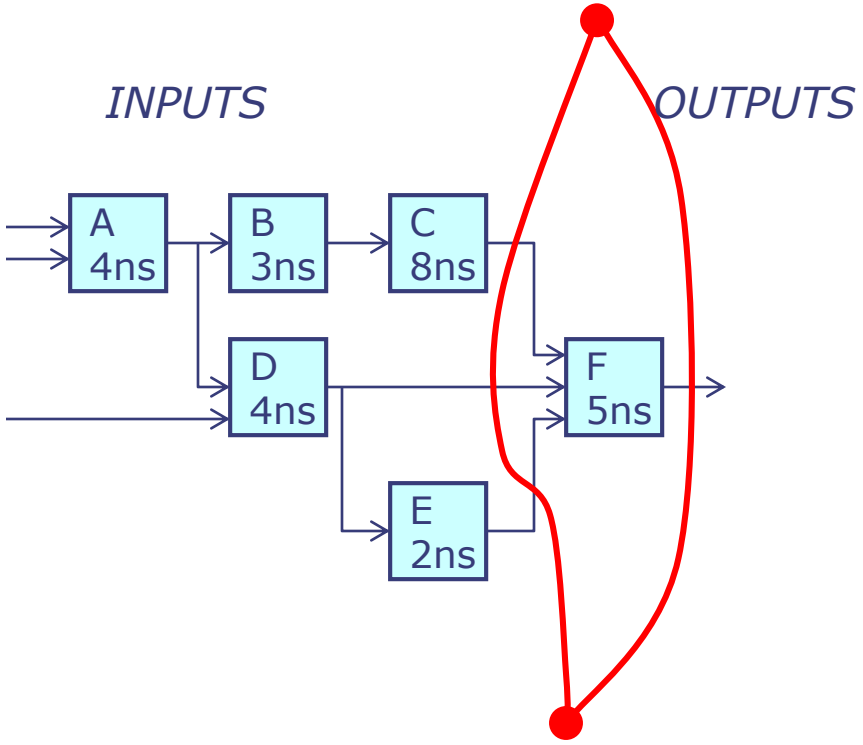
A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.



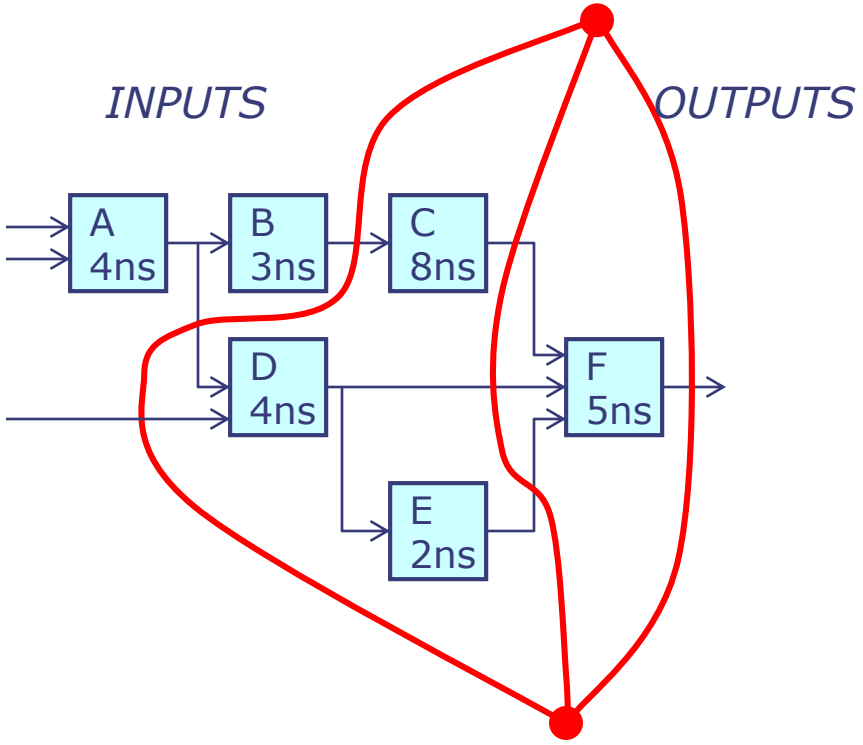
A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.



A Pipelining Methodology

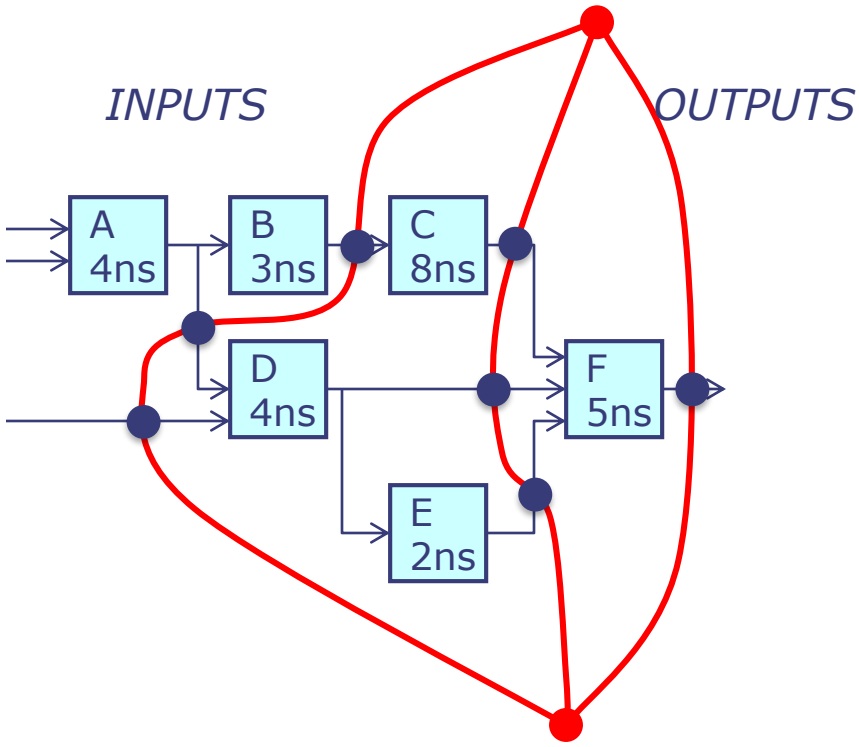
Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.



A Pipelining Methodology

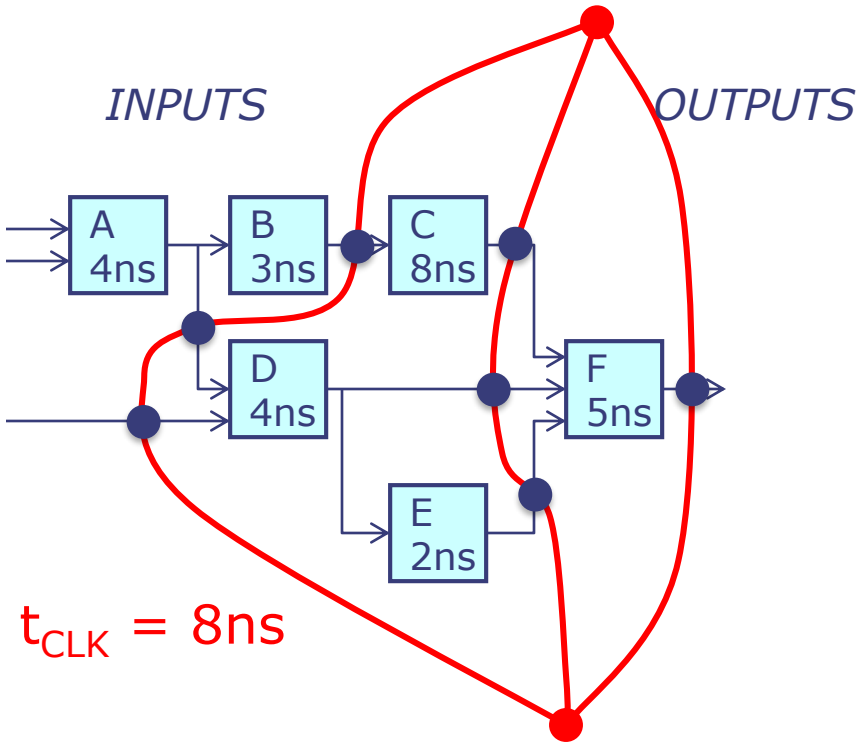
Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.



A Pipelining Methodology

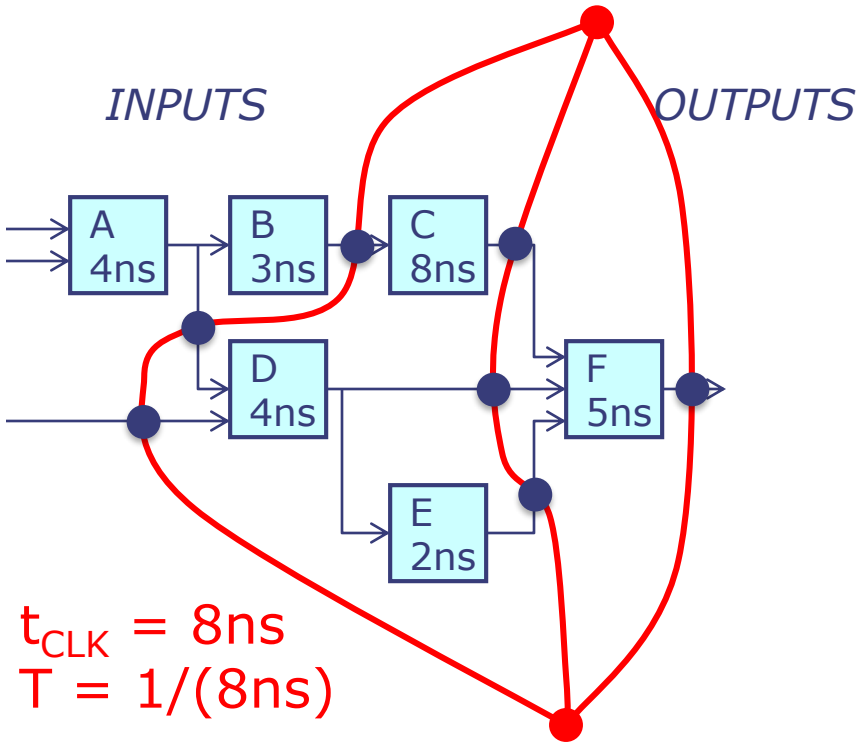
Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.



A Pipelining Methodology

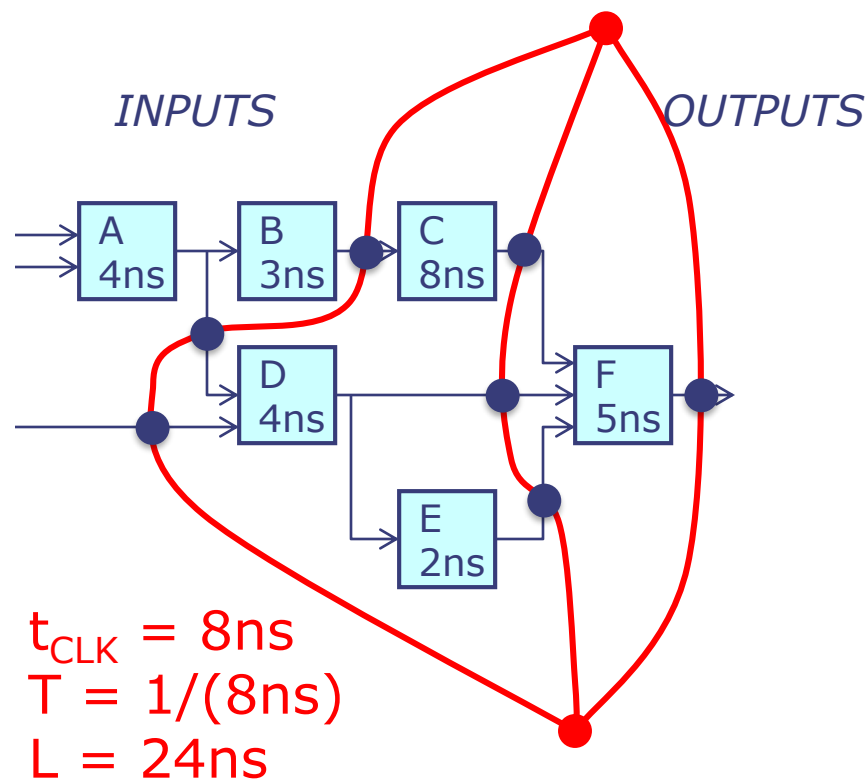
Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.



A Pipelining Methodology

Step 1:

Draw a line that crosses every output in the circuit, and mark the endpoints as terminal points.

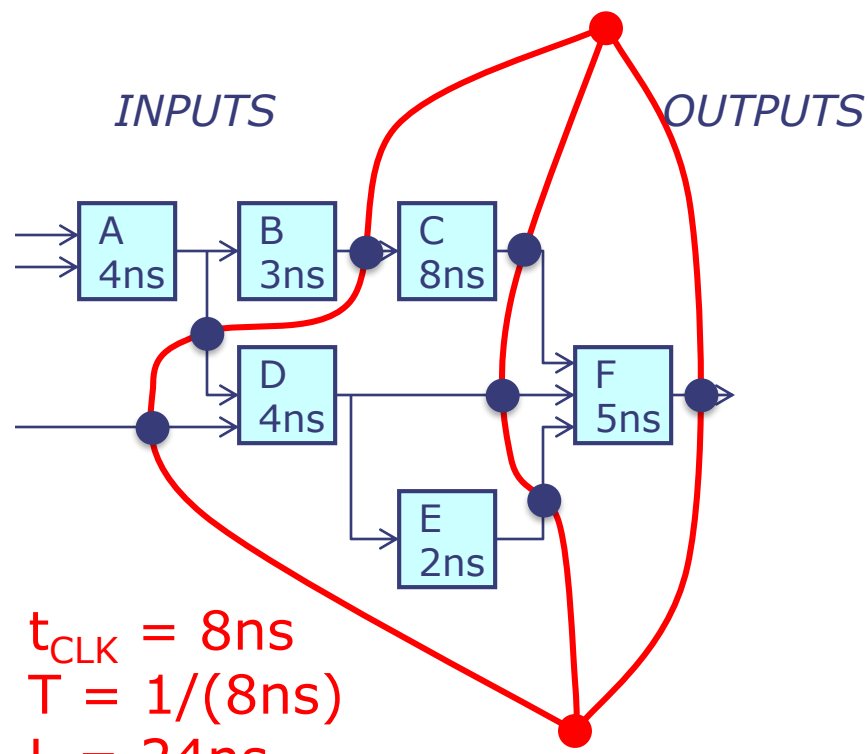
Step 2:

Continue to draw new lines between the terminal points across various circuit connections, ensuring that every connection crosses each line in the same direction. These lines demarcate *pipeline stages*.

Adding a pipeline register at every point where a separating line crosses a connection will always generate a valid pipeline.

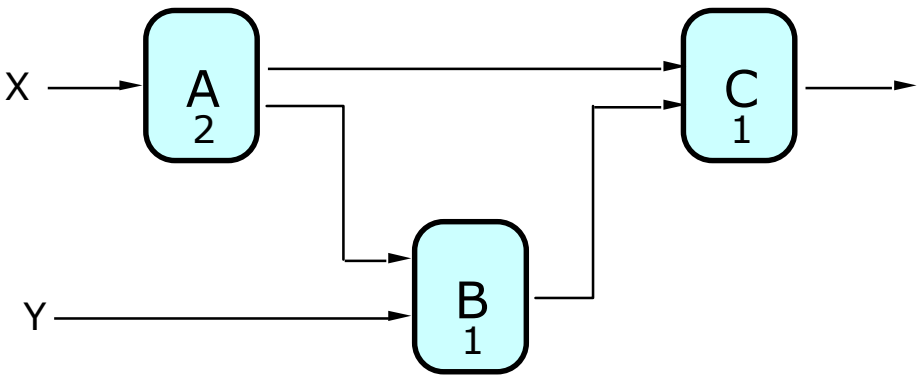
Strategy:

Focus your attention on placing pipelining registers around the slowest circuit elements (*bottlenecks*).



t_{CLK} = 8ns
T = 1/(8ns)
L = 24ns

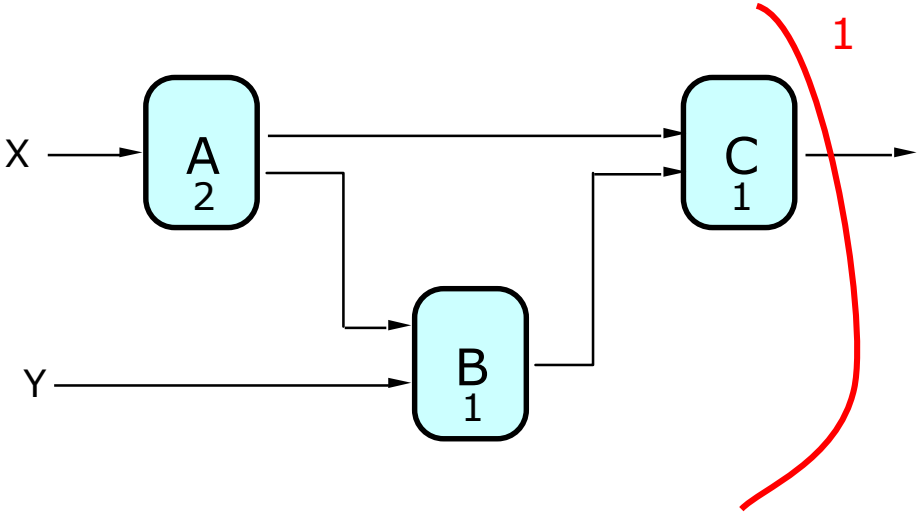
Pipeline Example



OBSERVATIONS:

	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:		
2-pipe:		
3-pipe:		

Pipeline Example

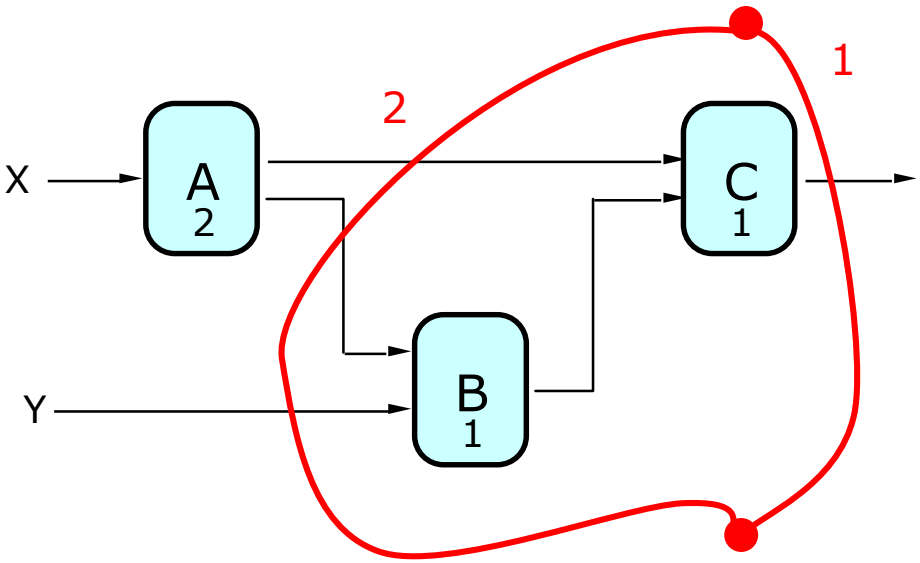


	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:		
3-pipe:		

OBSERVATIONS:

- 1-pipeline improves neither L nor T.

Pipeline Example

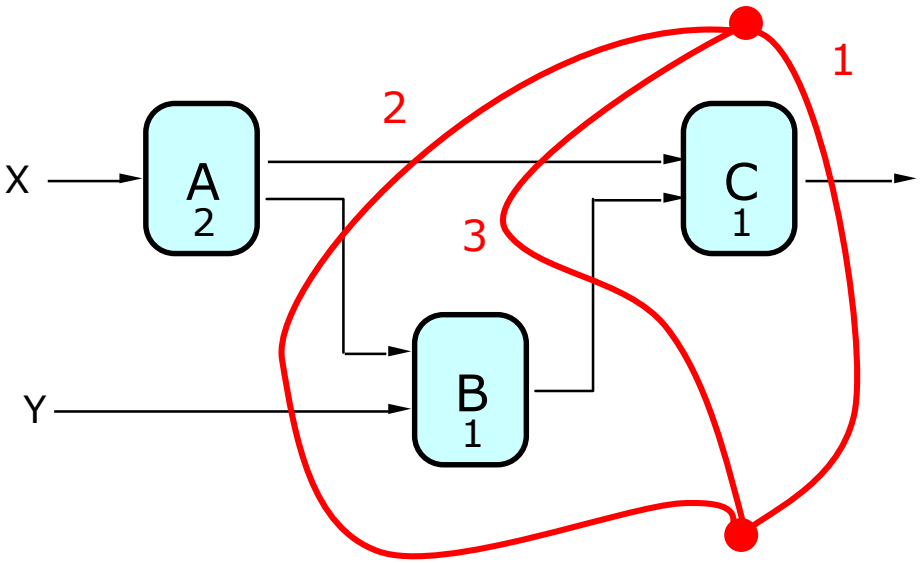


	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:		

OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.

Pipeline Example

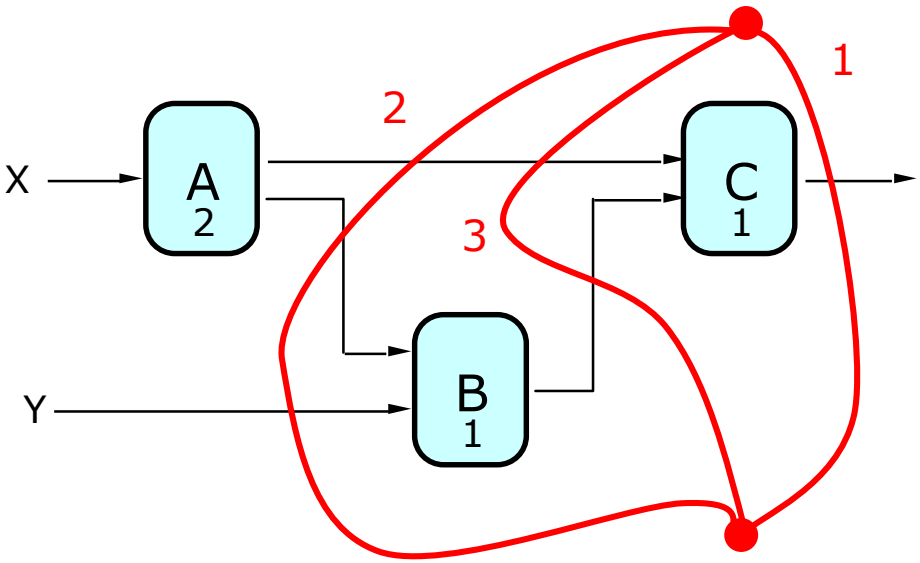


	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.

Pipeline Example

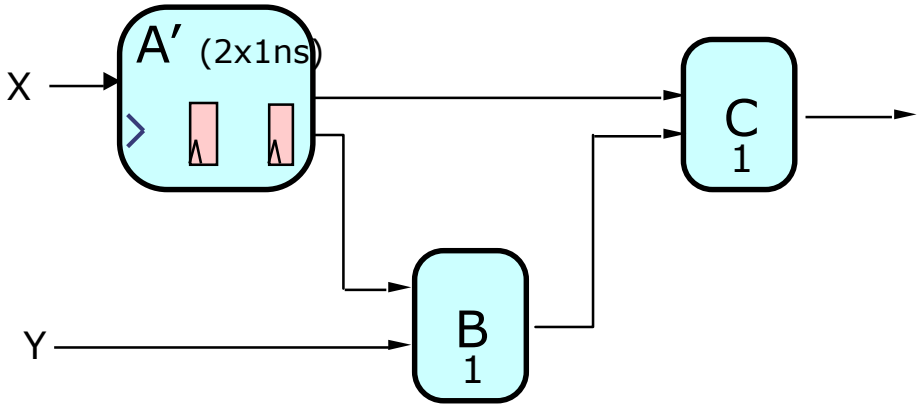


	LATENCY	THROUGHPUT
0-pipe:	4	1/4
1-pipe:	4	1/4
2-pipe:	4	1/2
3-pipe:	6	1/2

OBSERVATIONS:

- 1-pipeline improves neither L nor T.
- T improved by breaking long combinational paths, allowing faster clock.
- Too many stages cost L, don't improve T.
- Back-to-back registers are sometimes needed to keep pipeline well-formed.

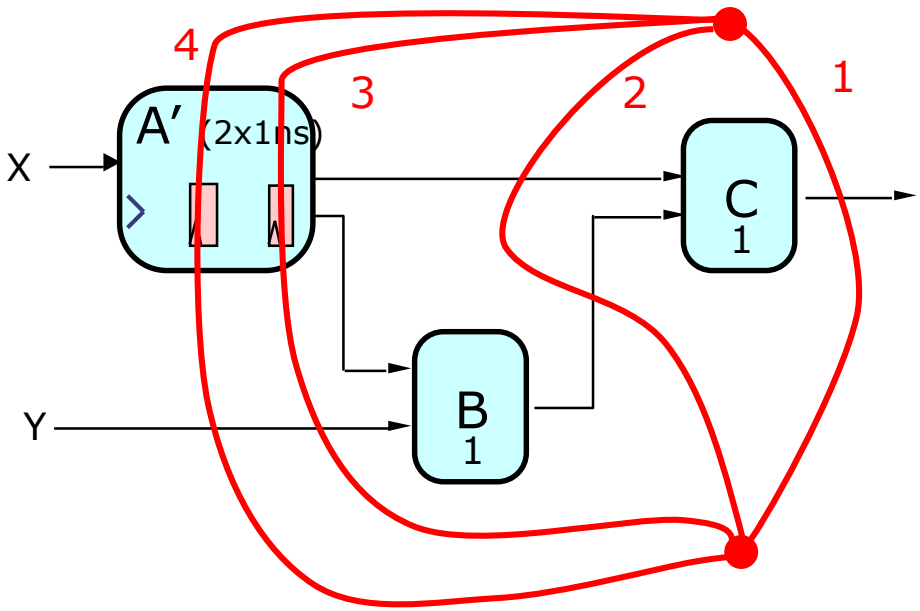
Pipelined Components



Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period

Pipelined Components



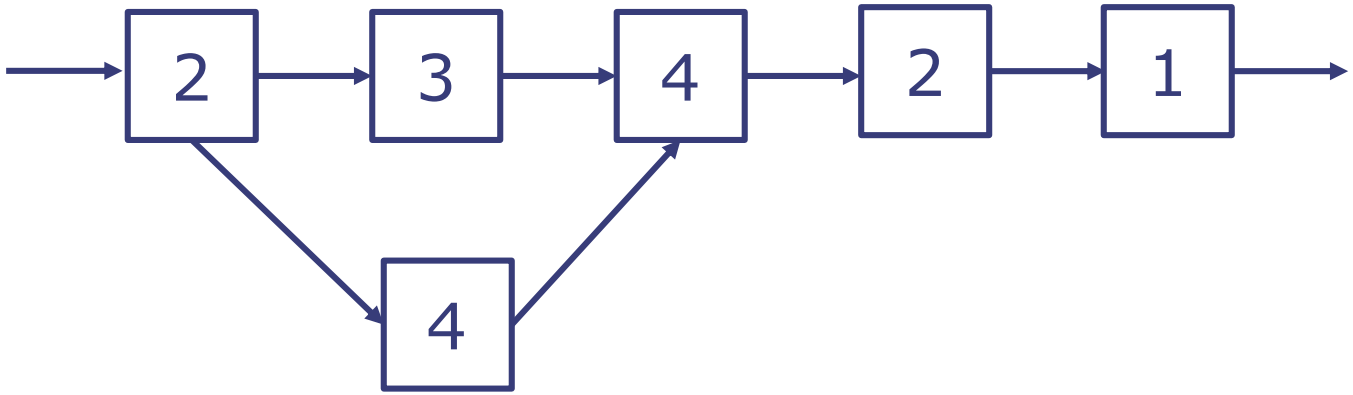
4-stage pipeline, throughput=1

Pipelined systems can be hierarchical:

- Replacing a slow combinational component with a k-pipe version may let us decrease the clock period
- Must account for new pipeline stages in our plan

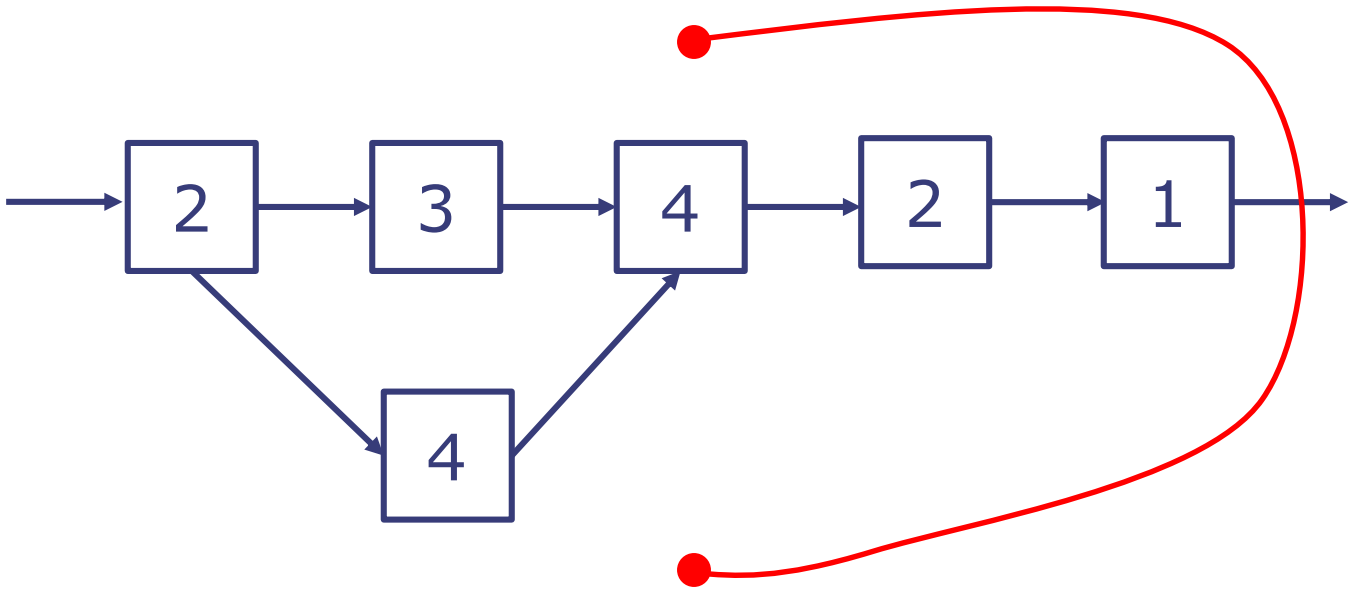
Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



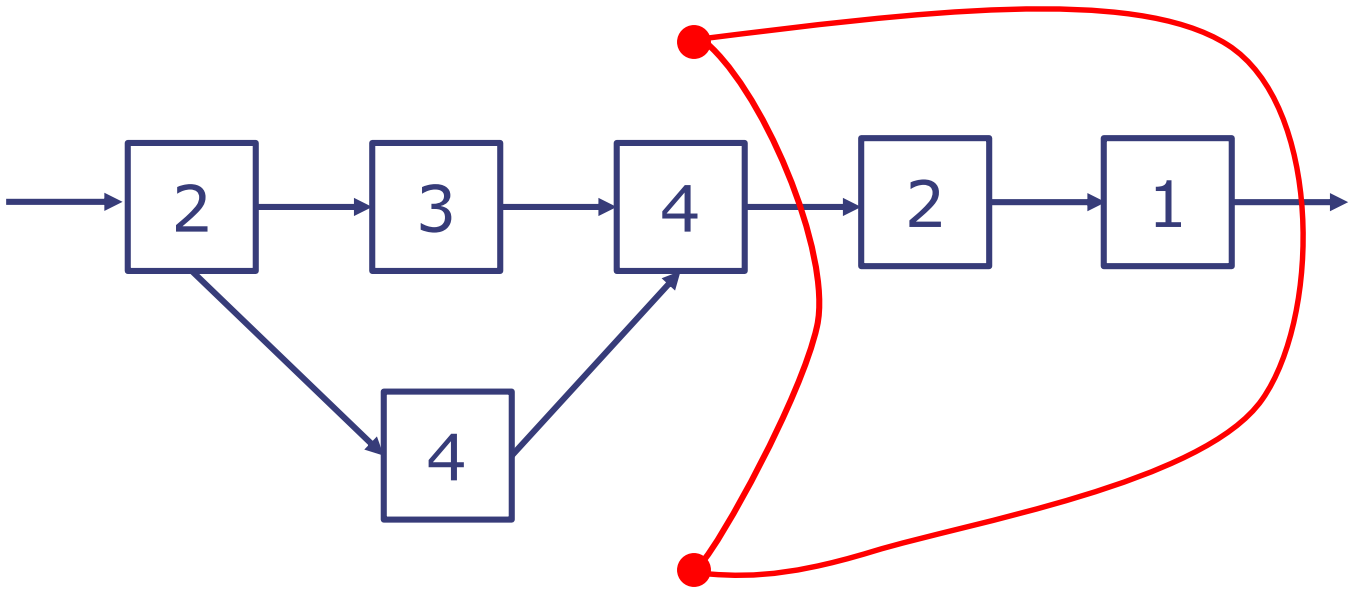
Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



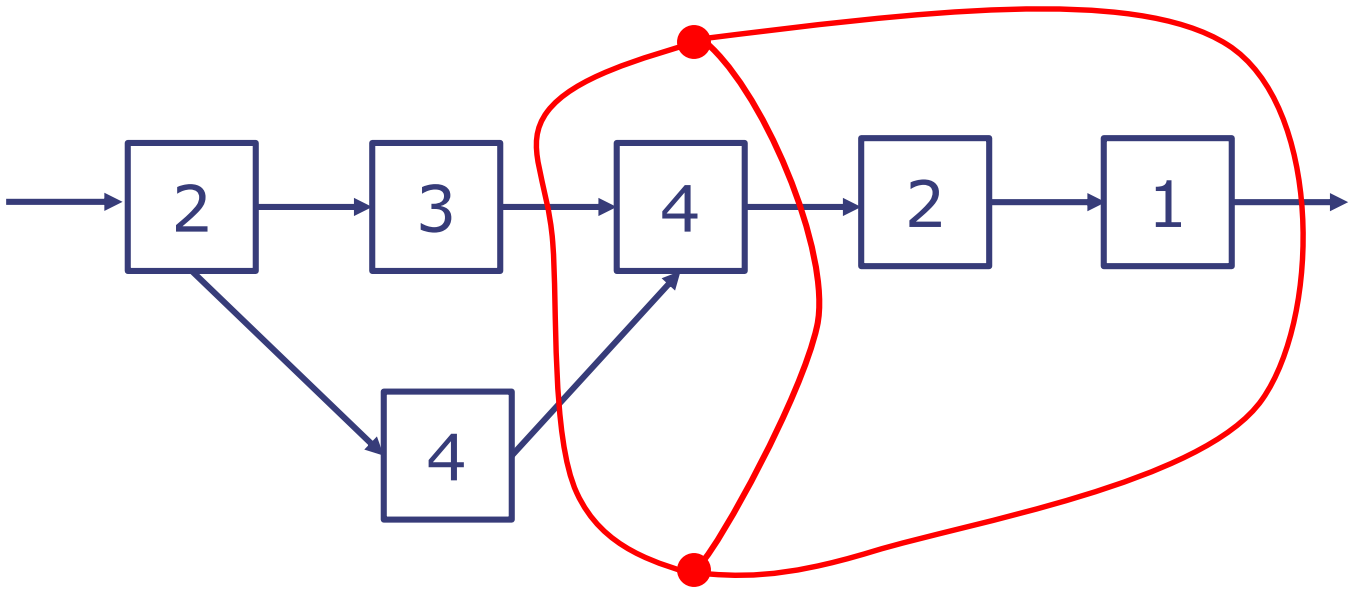
Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



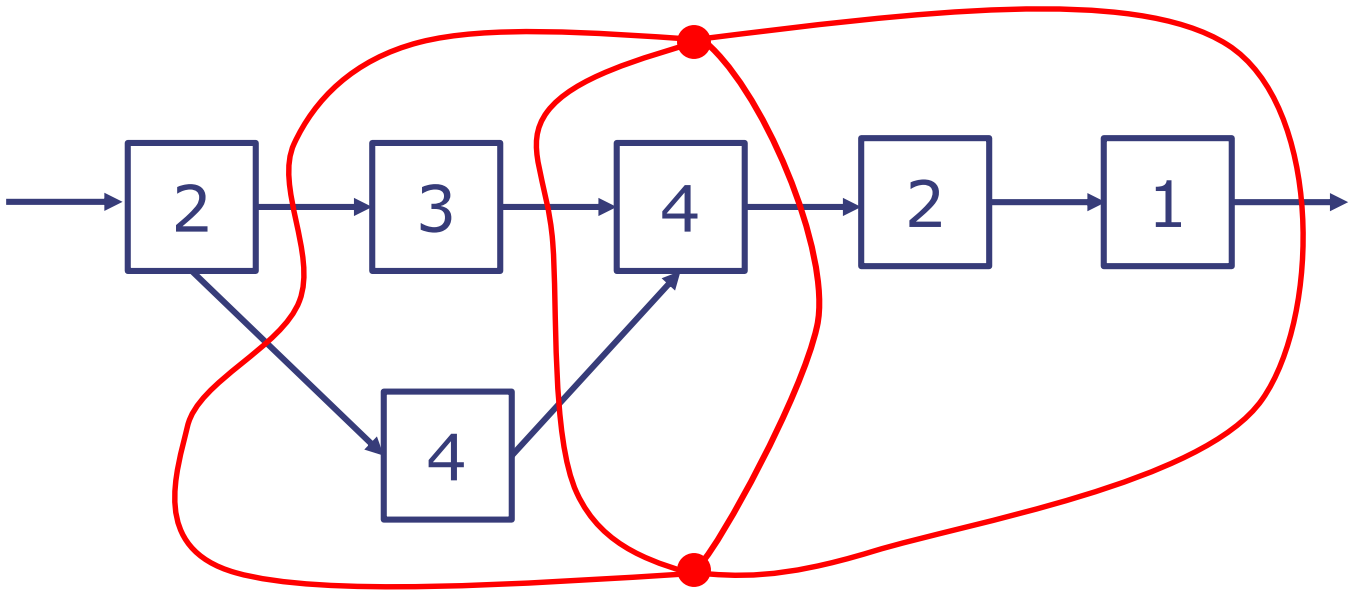
Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



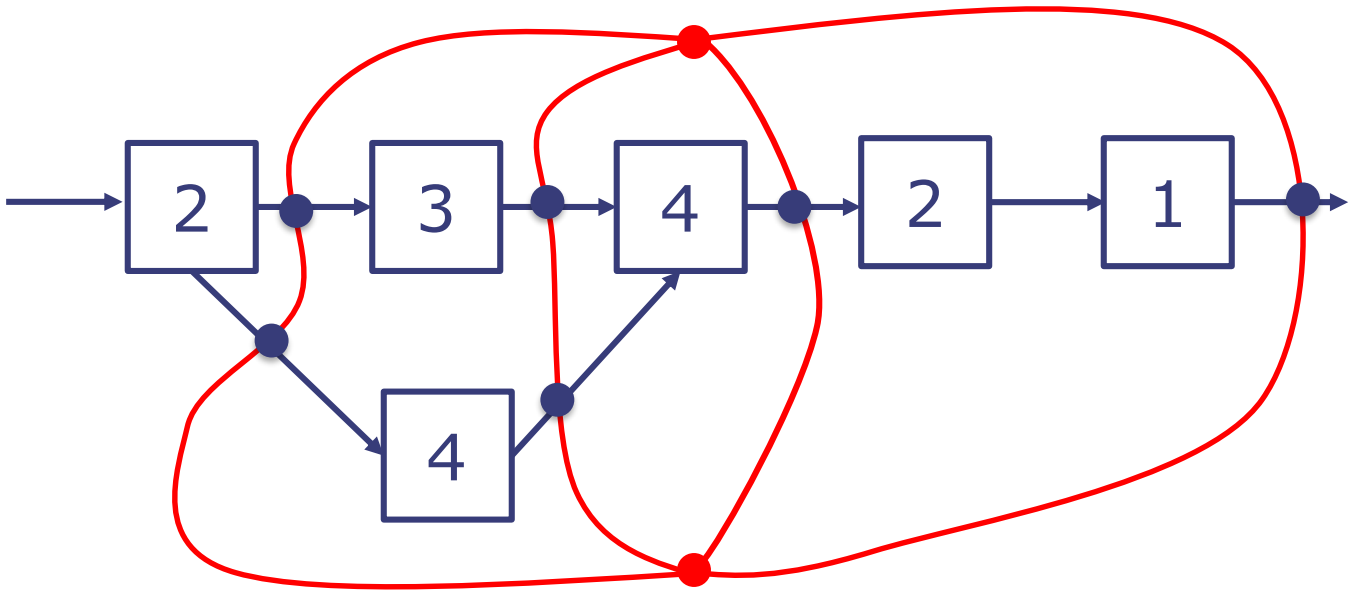
Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



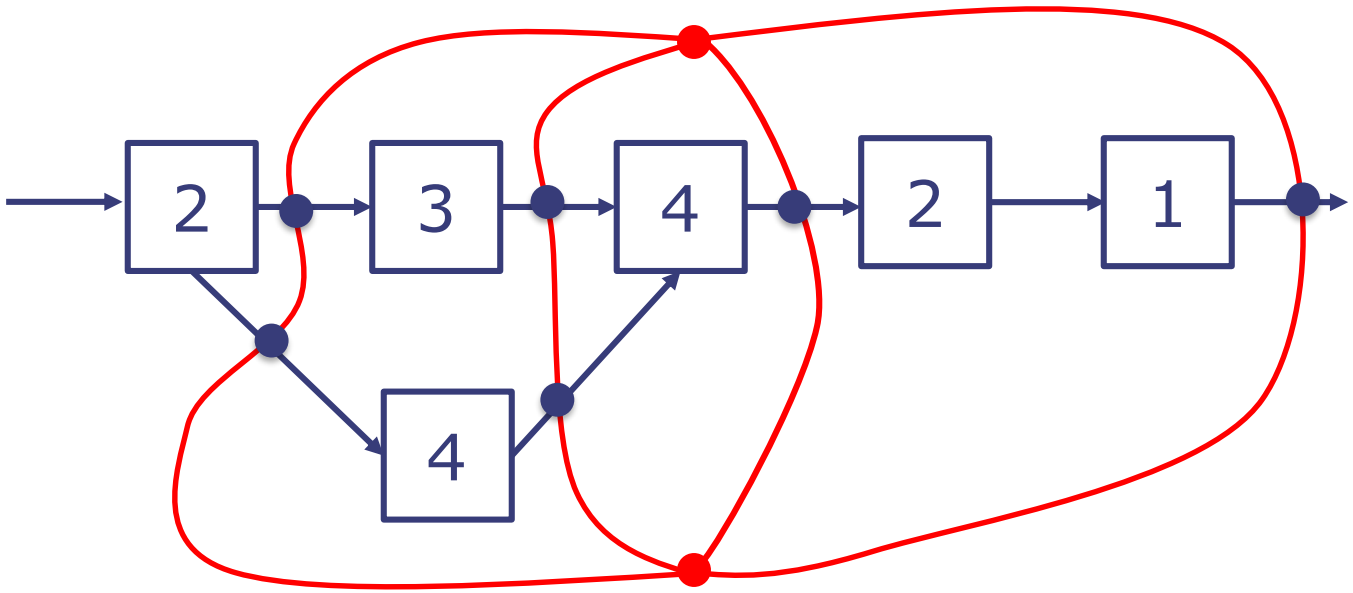
Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



Sample Pipelining Problem

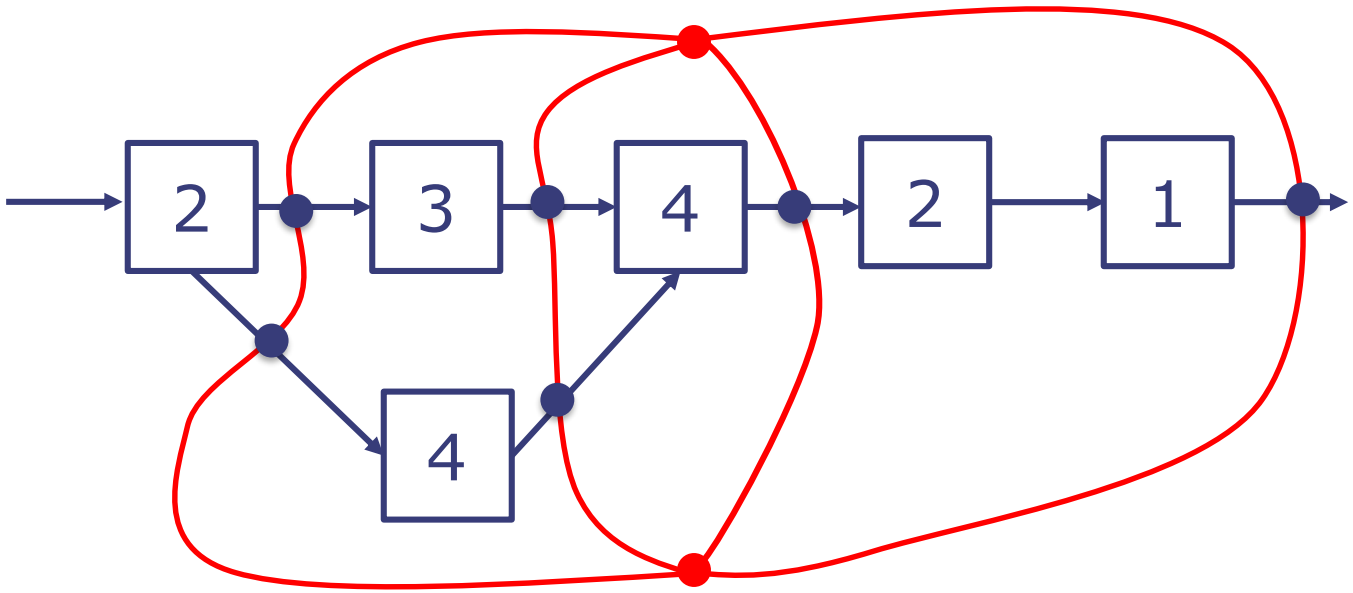
- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



- What is the latency and throughput of your pipelined circuit?

Sample Pipelining Problem

- Pipeline the following circuit for maximum throughput while minimizing latency. The number in each module is the module's latency.



- What is the latency and throughput of your pipelined circuit?

$t_{CLK} = 4$
$T = 1/(4)$
$L = 4*4 = 16$

Design Tradeoffs Introduction:

Multiplier Case Study

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

We also shift the result by one position at every step

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

We also shift the result by one position at every step

Note the first addition is unnecessary because it simply yields m0

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)
```

At each step we add either b (1101) or 0 to the result depending upon a bit in the multiplier

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

We also shift the result by one position at every step

Note the first addition is unnecessary because it simply yields m0

Also note that these are unsigned binary numbers.

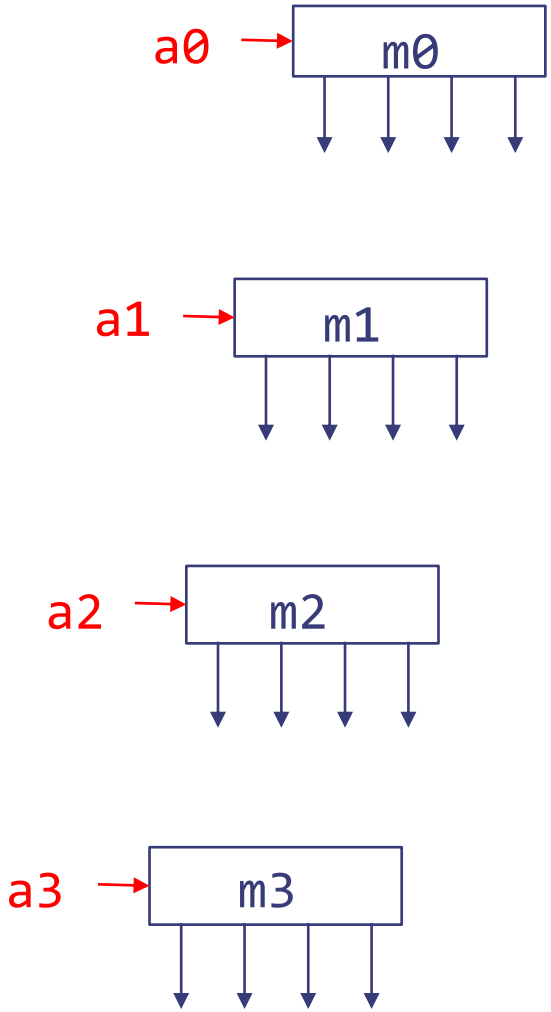
Multiplication by repeated addition
Combinational circuit

```
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;
```

Multiplication by repeated addition
Combinational circuit

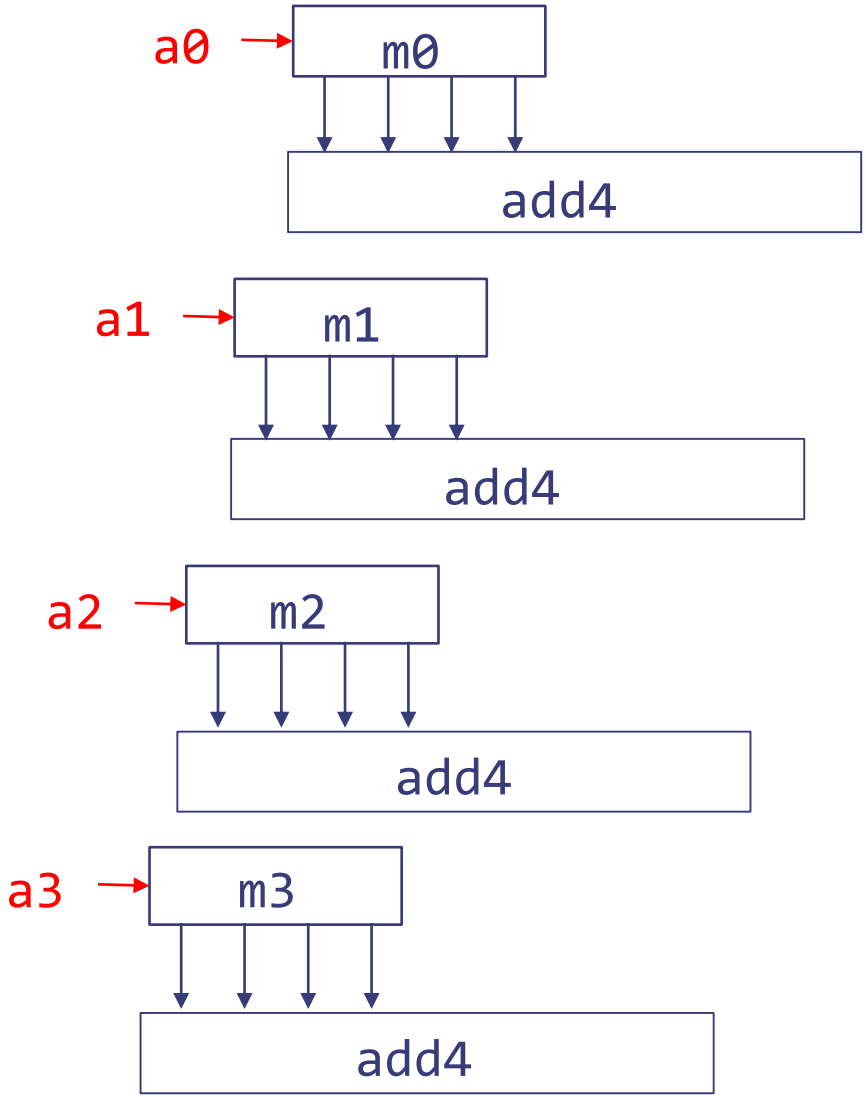
```
b Multiplicand   1101   (13)
a Multiplier   * 1011   (11)

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;
```

Multiplication by repeated addition
Combinational circuit

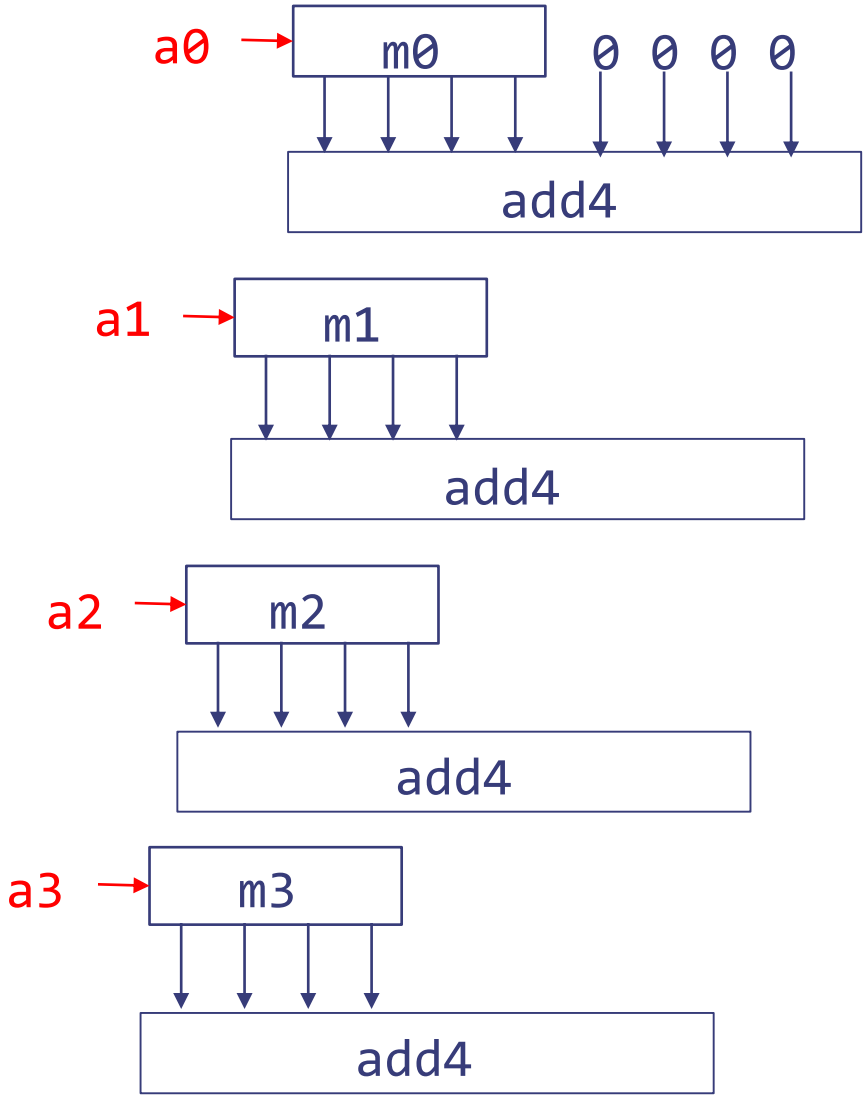
```
b Multiplicand   1101   (13)
a Multiplier   * 1011   (11)

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;
```

Multiplication by repeated addition
Combinational circuit

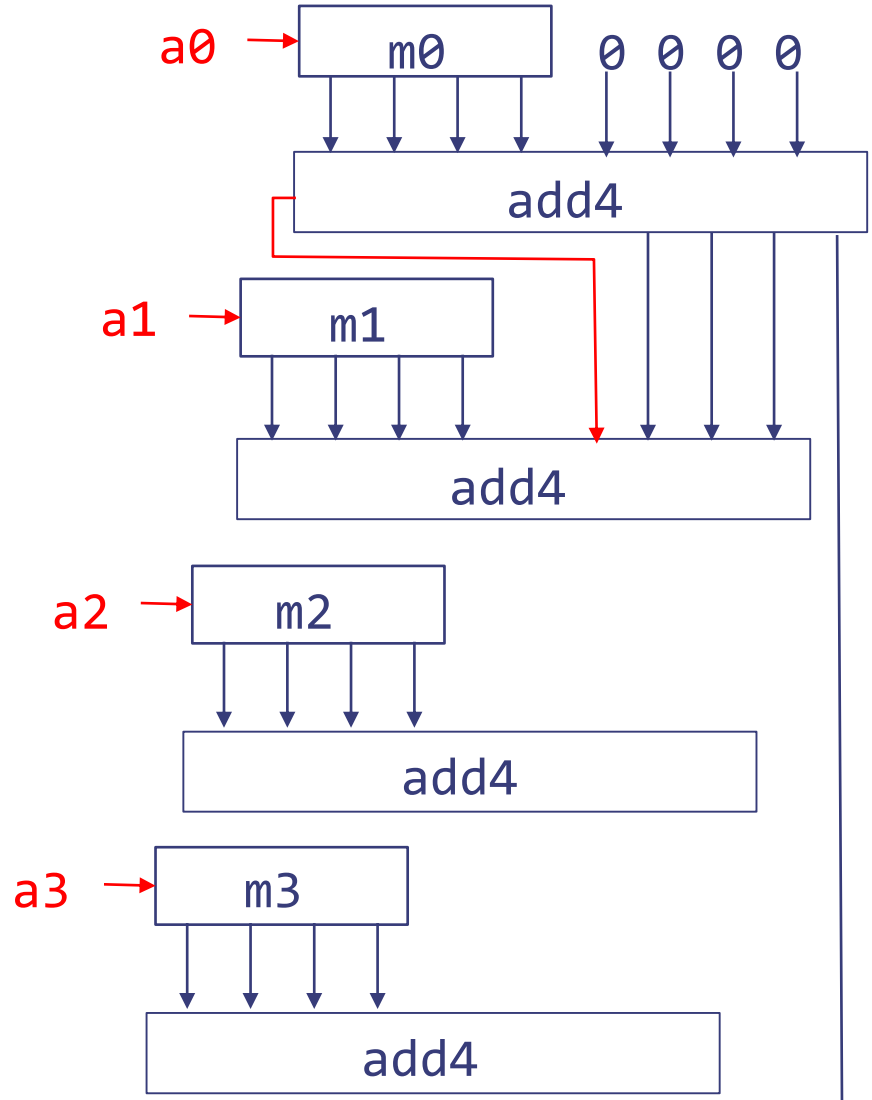
```
b Multiplicand   1101   (13)
a Multiplier   * 1011   (11)

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;
```

Multiplication by repeated addition
Combinational circuit

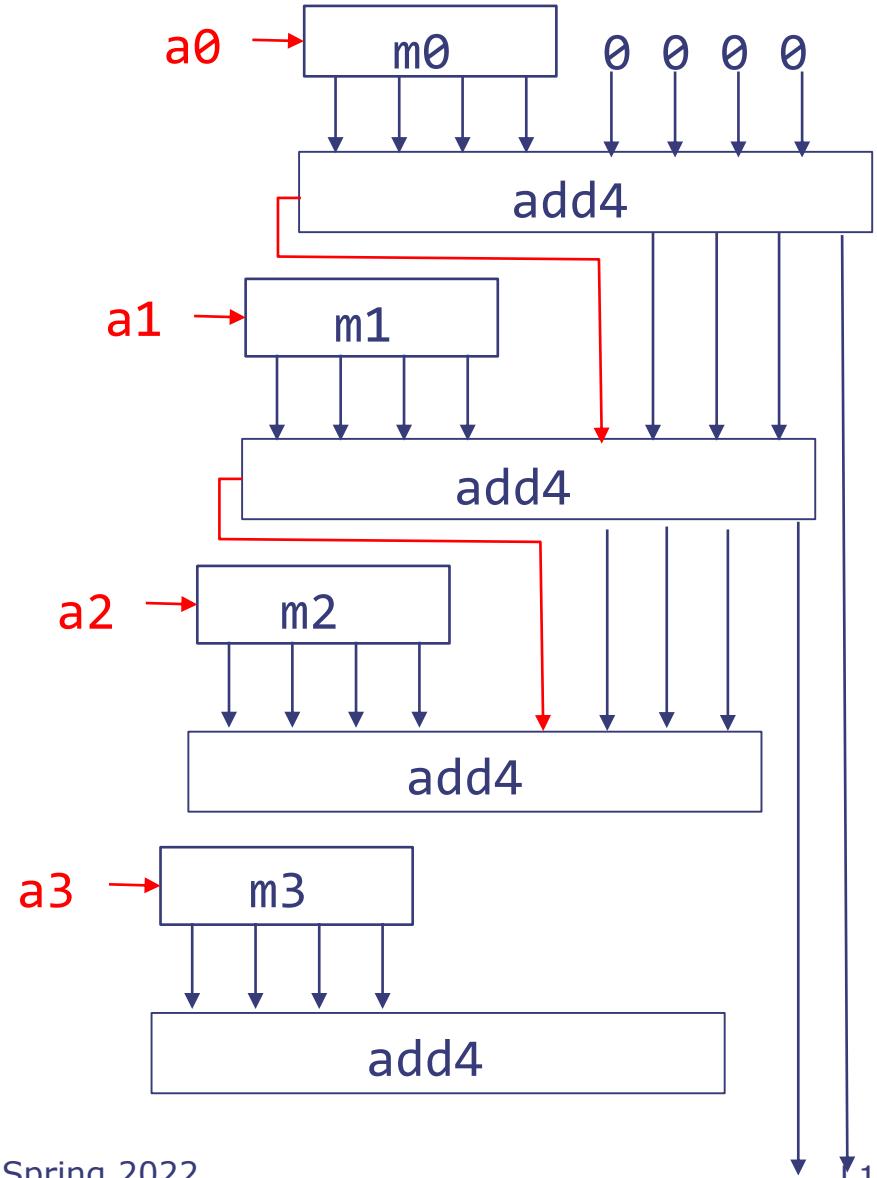
```
b Multiplicand   1101    (13)
a Multiplier   * 1011    (11)

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;
```

Multiplication by repeated addition
Combinational circuit

```
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)
```

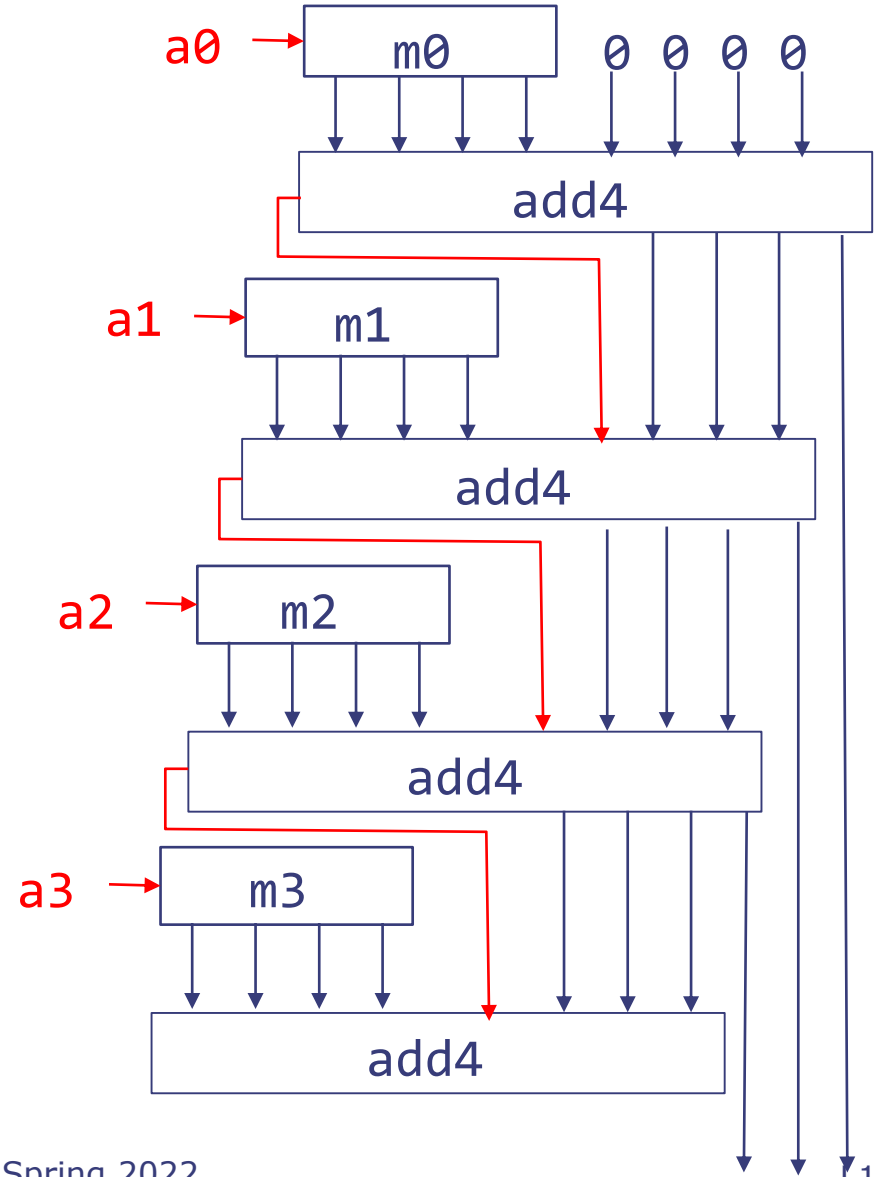



```
mi = (a[i]==0)? 0 : b;
```

Multiplication by repeated addition
Combinational circuit

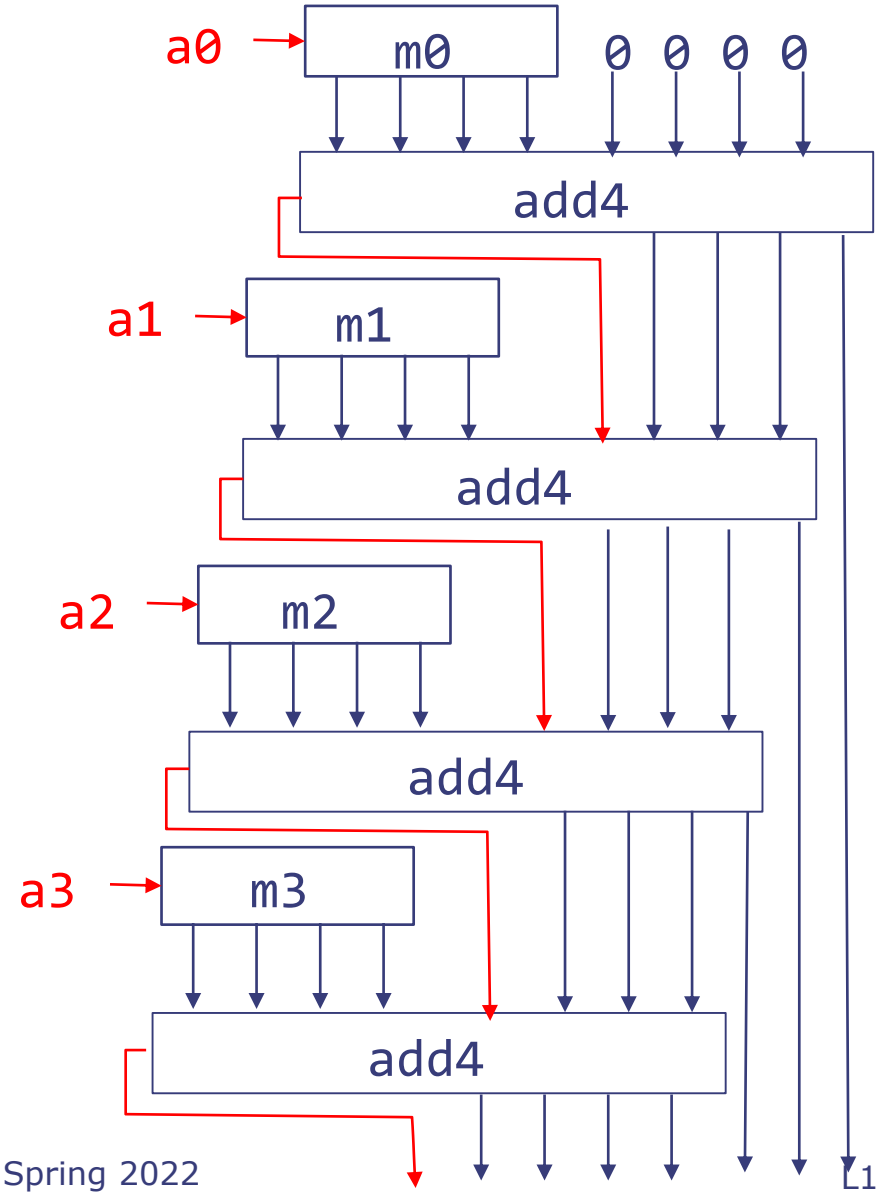
```
b Multiplicand   1101    (13)
a Multiplier   * 1011    (11)

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;
```



Multiplication by repeated addition
Combinational circuit

```
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)
```



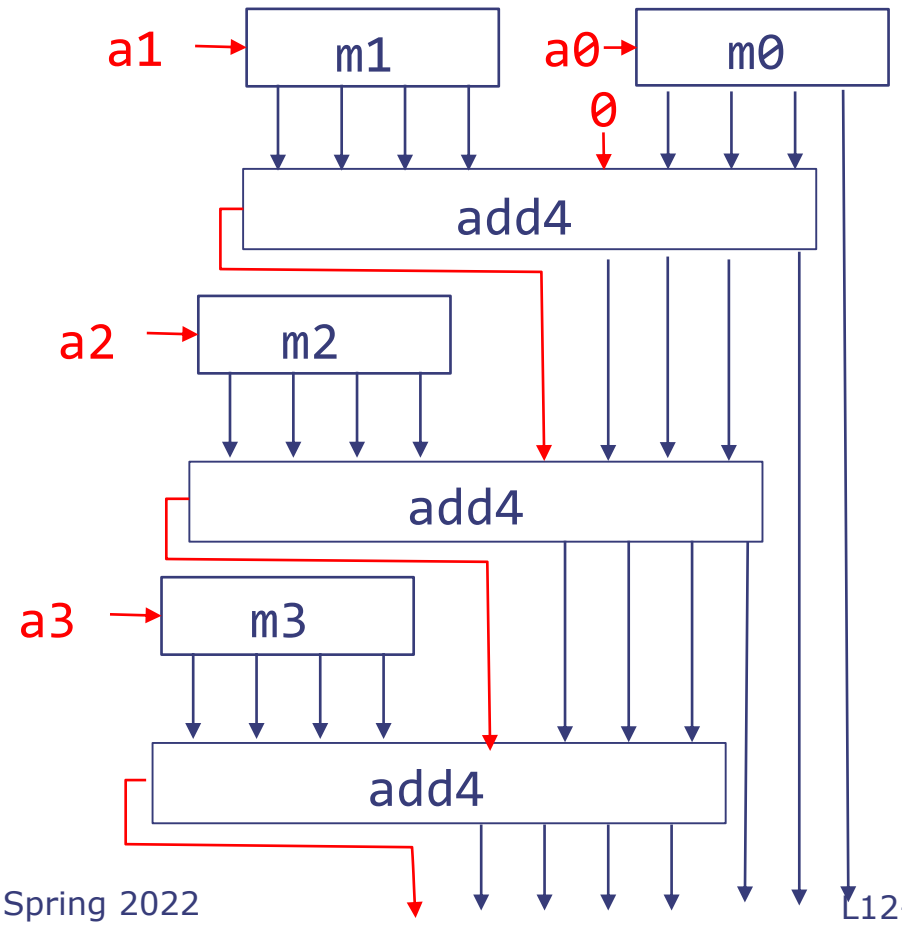

```
mi = (a[i]==0)? 0 : b;
```

Multiplication by repeated addition
Combinational circuit

```
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;
```

First adder is unnecessary (always yields m0)



Implementation of mi

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

Implementation of mi

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

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

m0
m1
m2
m3

Implementation of mi

The "Binary" Multiplication Table

*	0	1
0	0	0
1	0	1

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

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

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;
```

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

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
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
m1
m2
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;
```

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

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

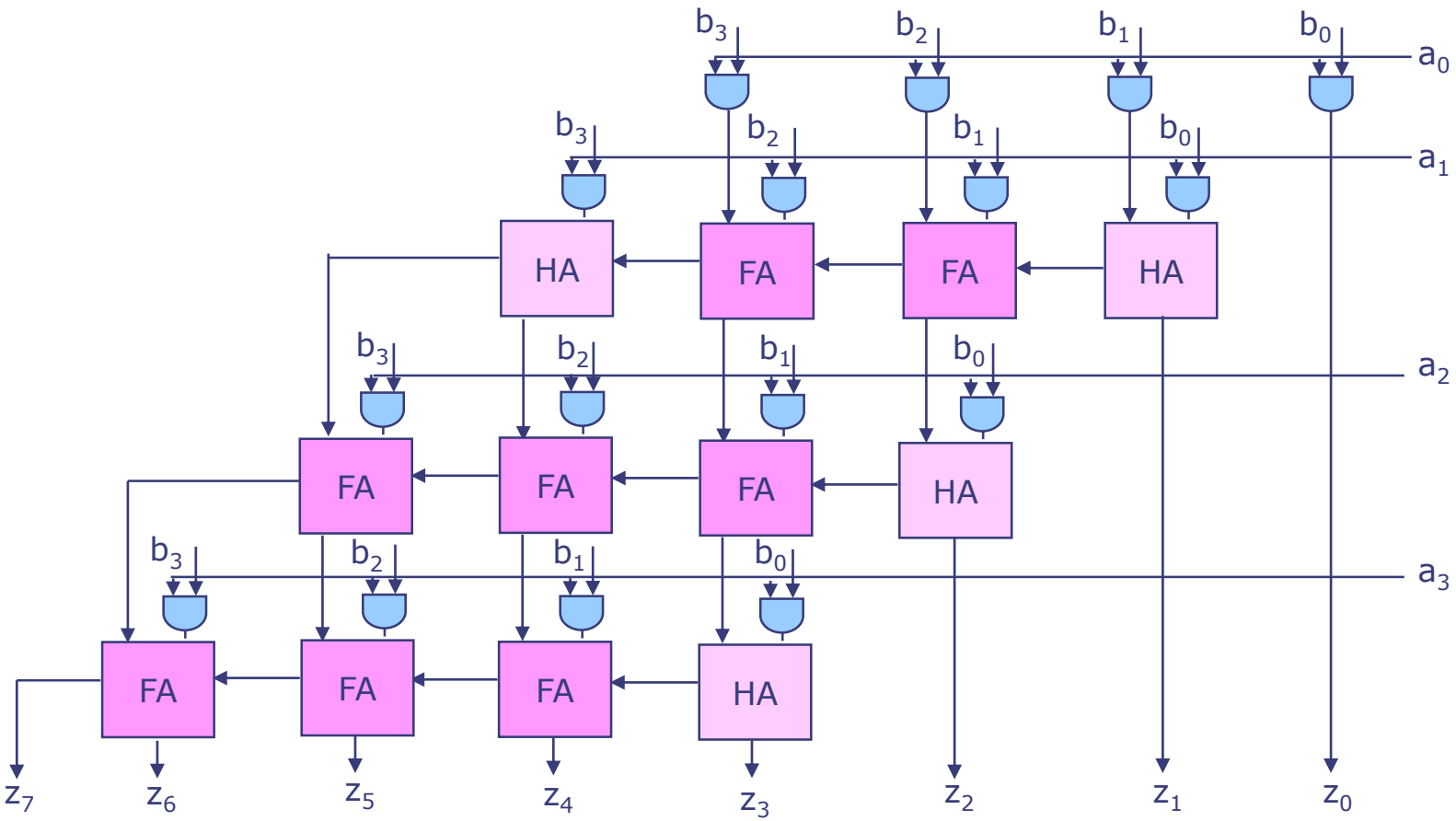
B_3A_2 B_2A_2 B_1A_2 B_0A_2 m2

$+ B_3A_3$ B_2A_3 B_1A_3 B_0A_3 m3

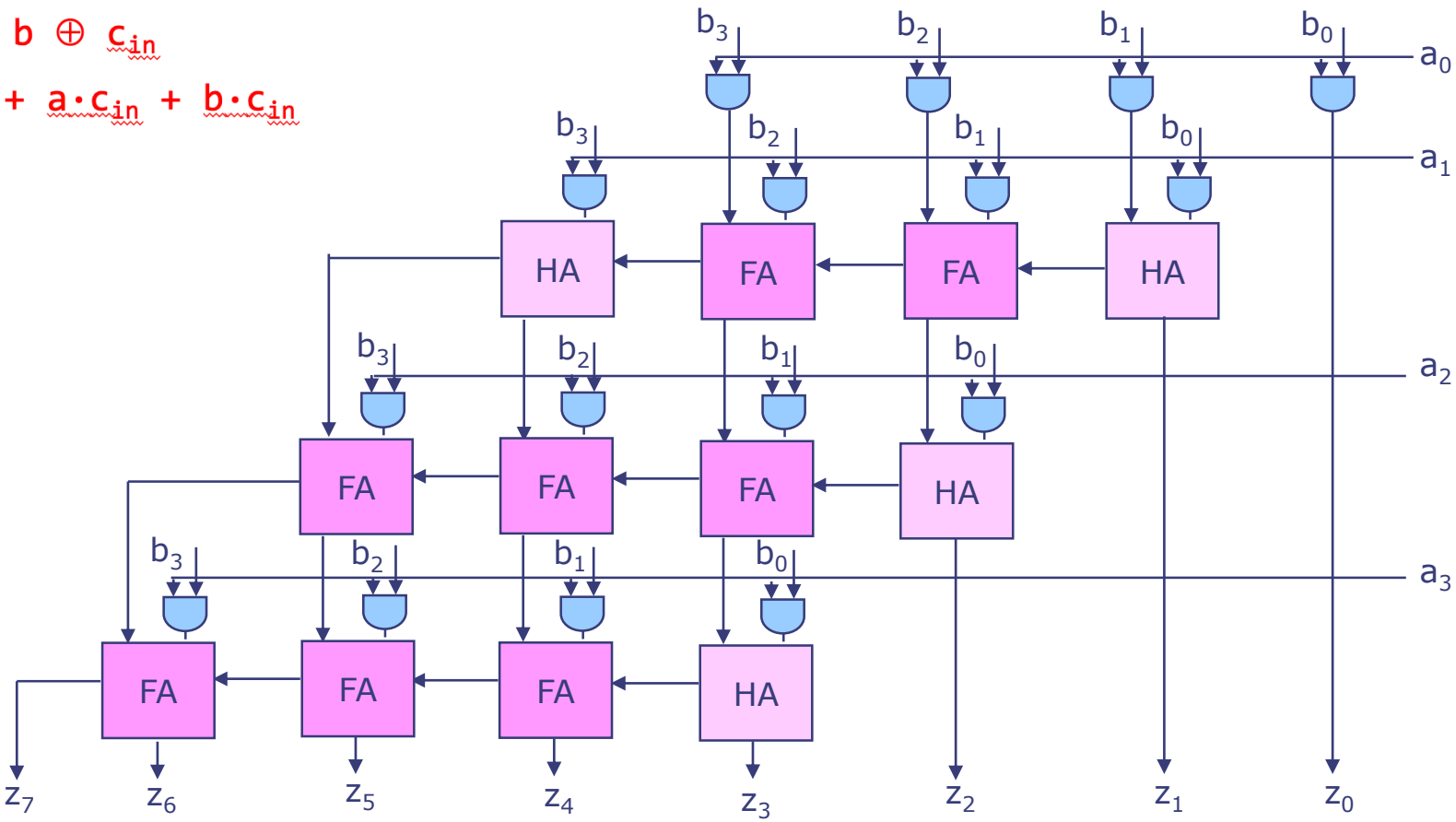
Multiplying N-digit number by M-digit number gives (N+M)-digit result

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

Combinational Multiplier Redrawn



Combinational Multiplier Redrawn



Combinational Multiplier Redrawn

FA

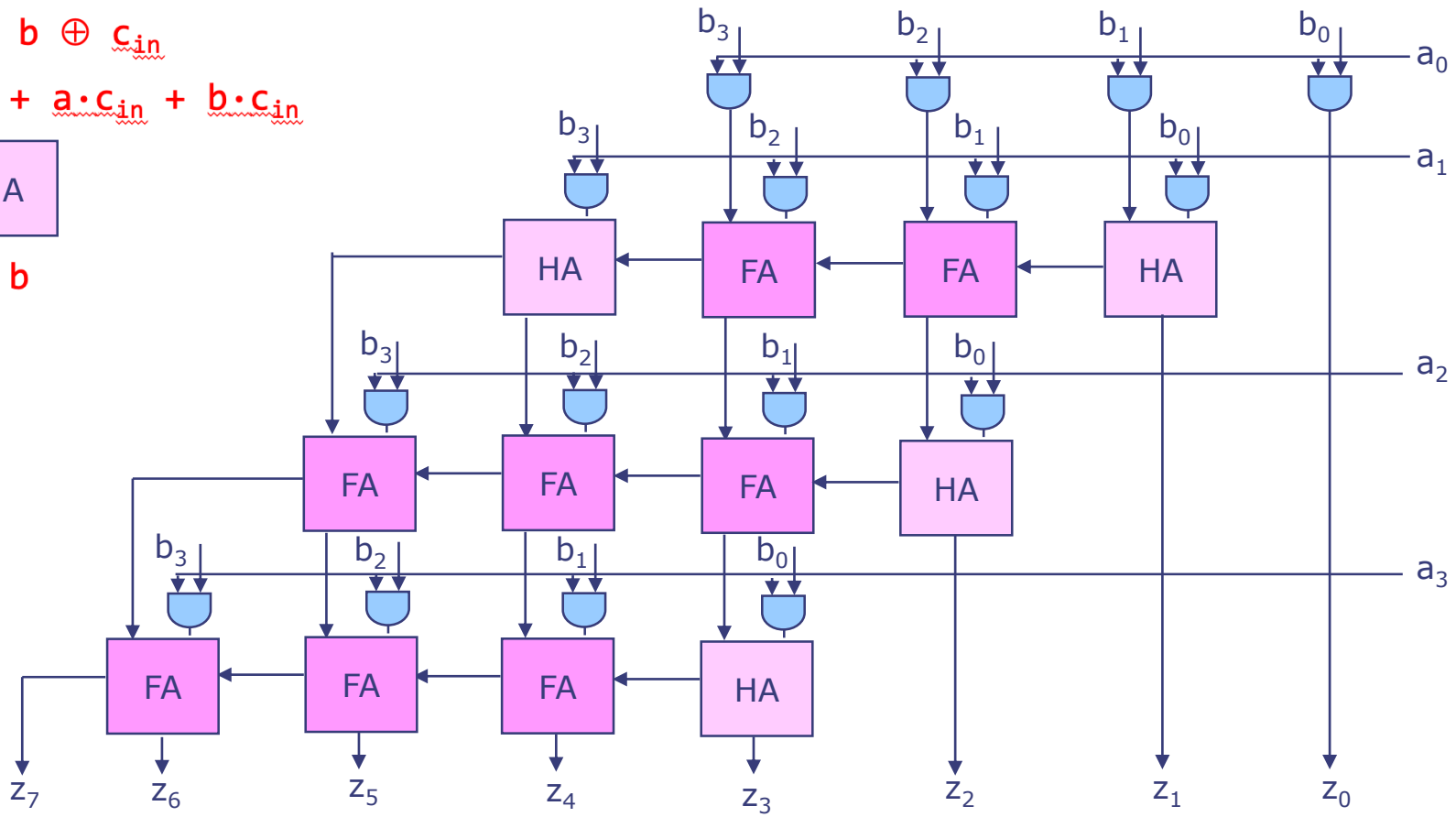
$s = a \oplus b \oplus c_{in}$

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

HA

$s = a \oplus b$

$c_{out} = a \cdot b$



Combinational Multiplier Redrawn

FA

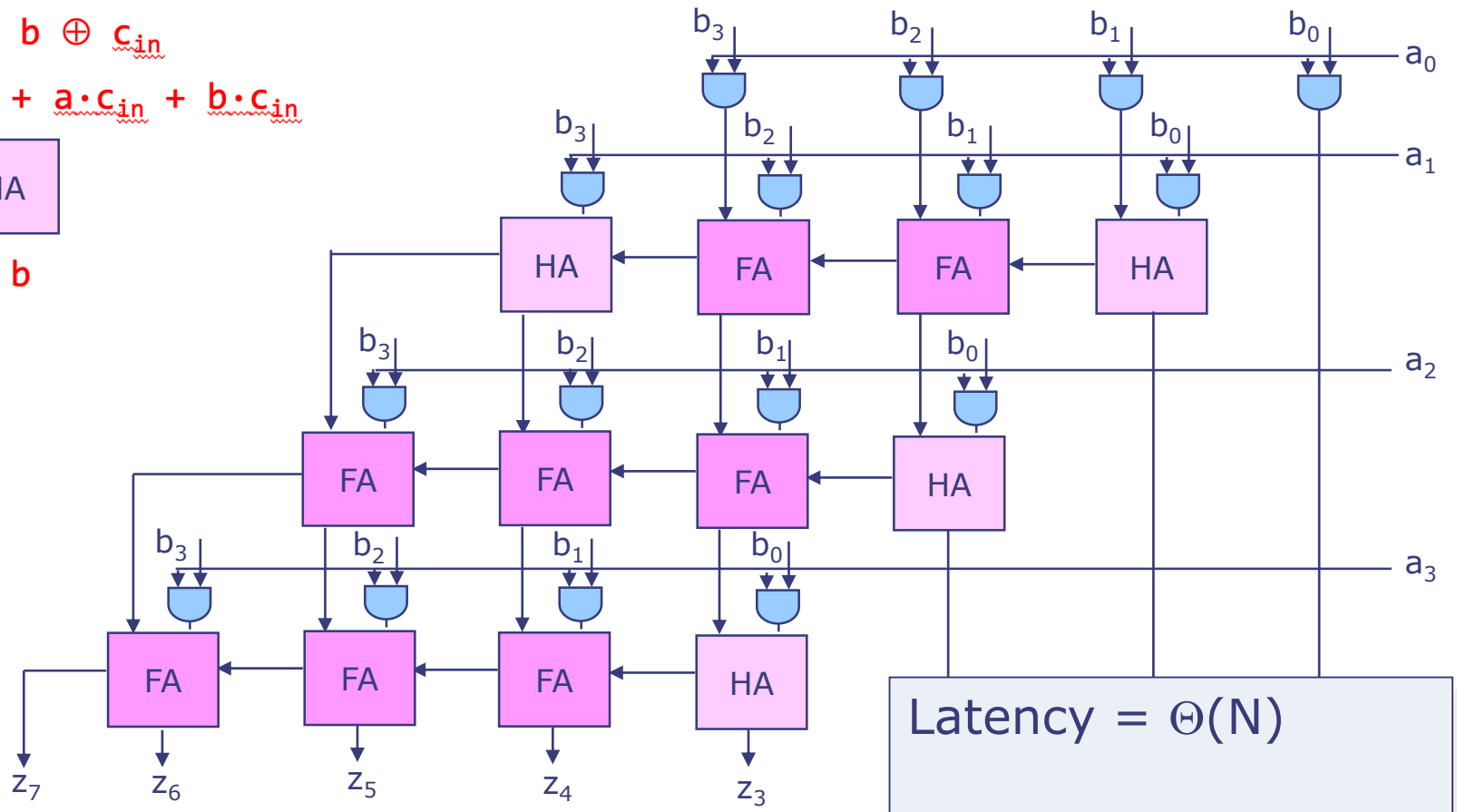
$s = a \oplus b \oplus c_{in}$

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

HA

$s = a \oplus b$

$c_{out} = a \cdot b$



Latency = $\Theta(N)$

Combinational Multiplier Redrawn

FA

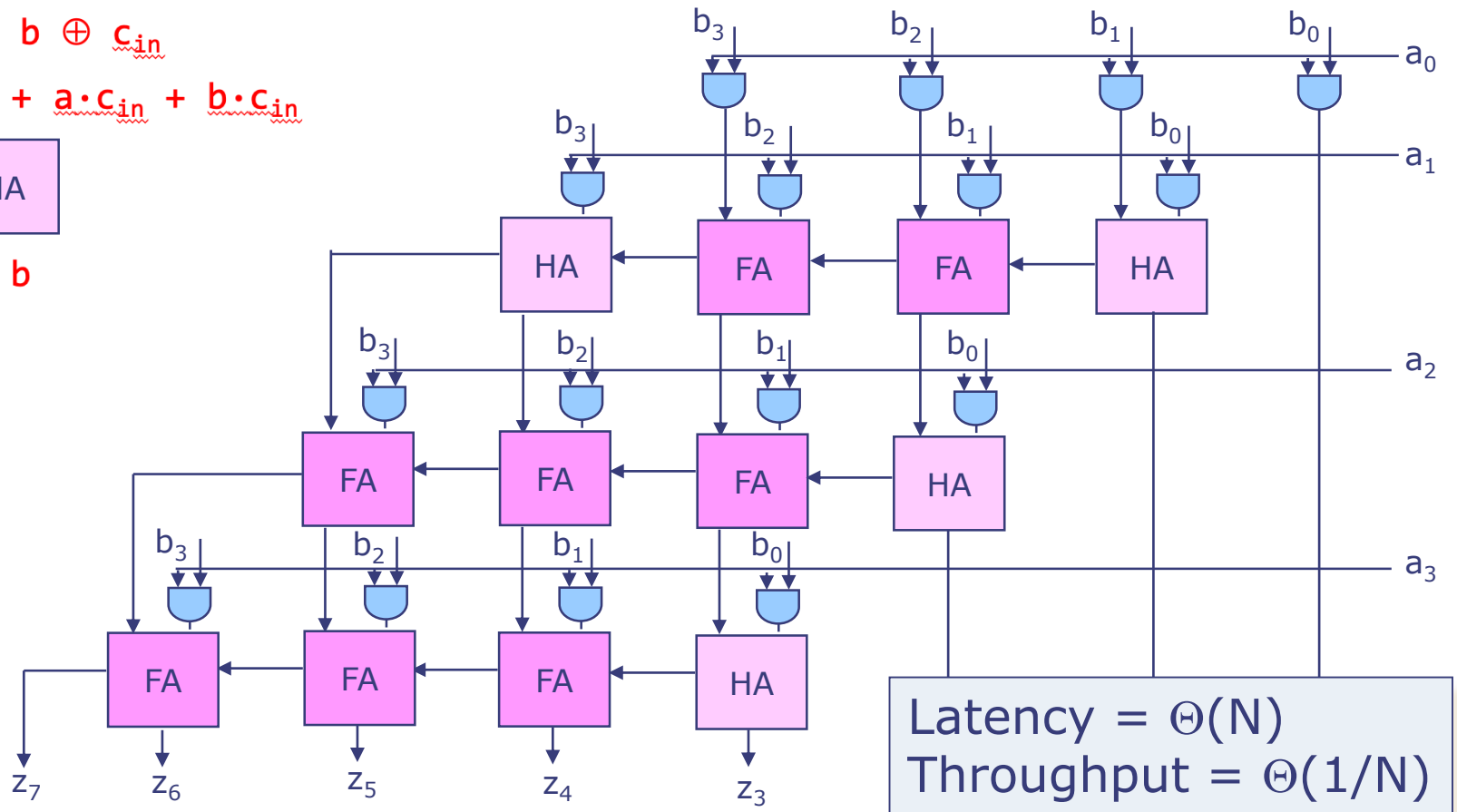
$s = a \oplus b \oplus c_{in}$

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

HA

$s = a \oplus b$

$c_{out} = a \cdot b$



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

Combinational Multiplier Redrawn

FA

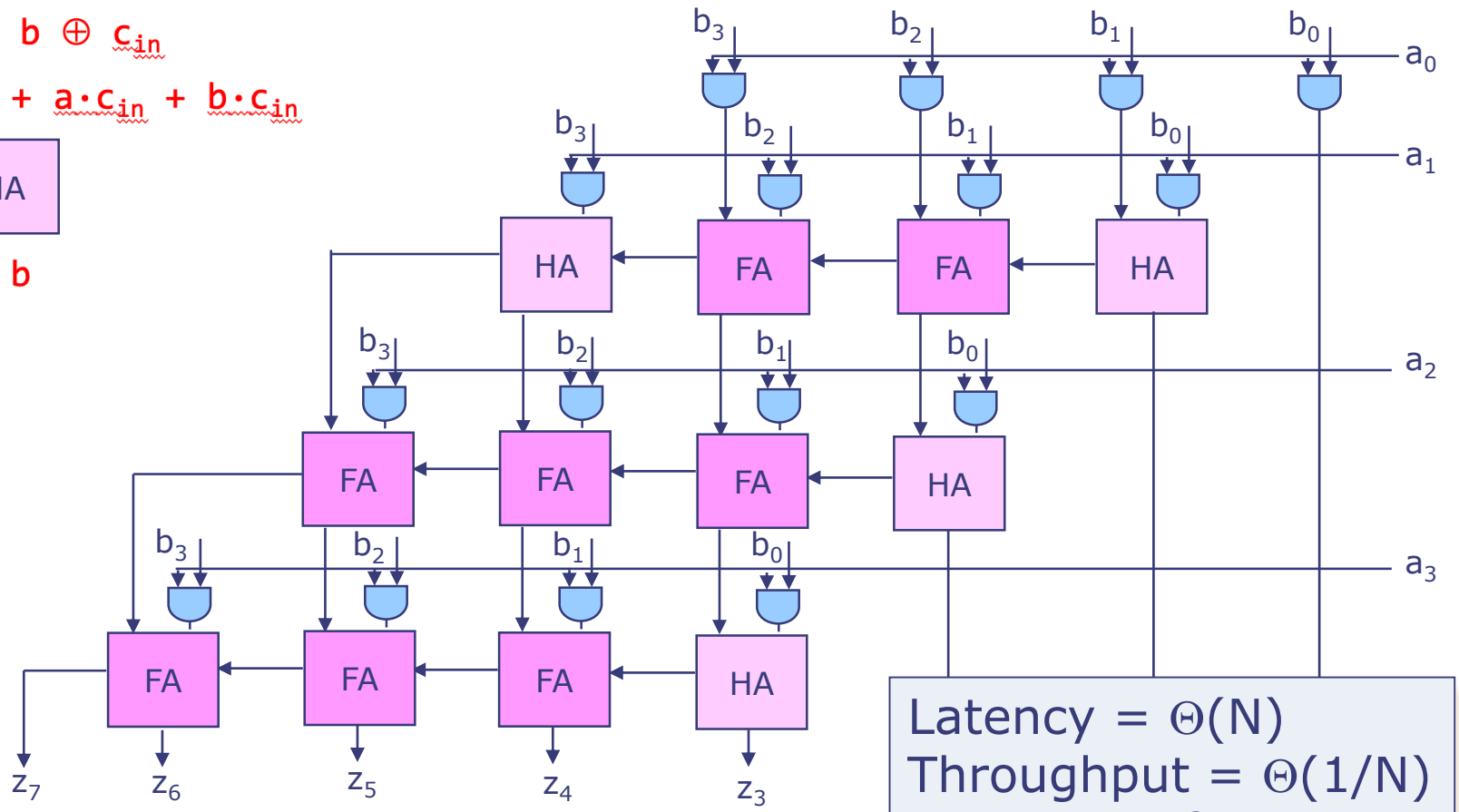
$s = a \oplus b \oplus c_{in}$

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

HA

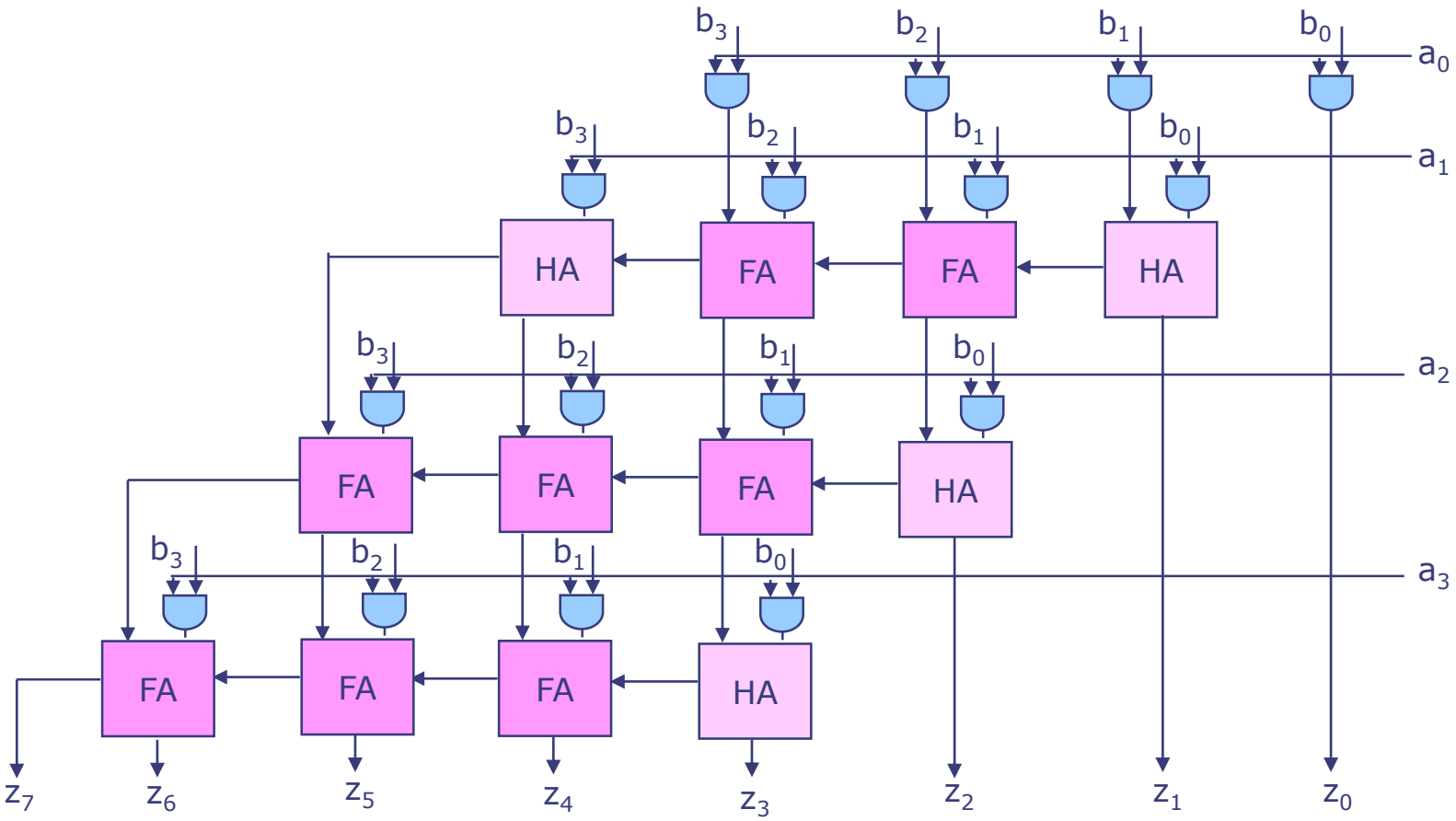
$s = a \oplus b$

$c_{out} = a \cdot b$

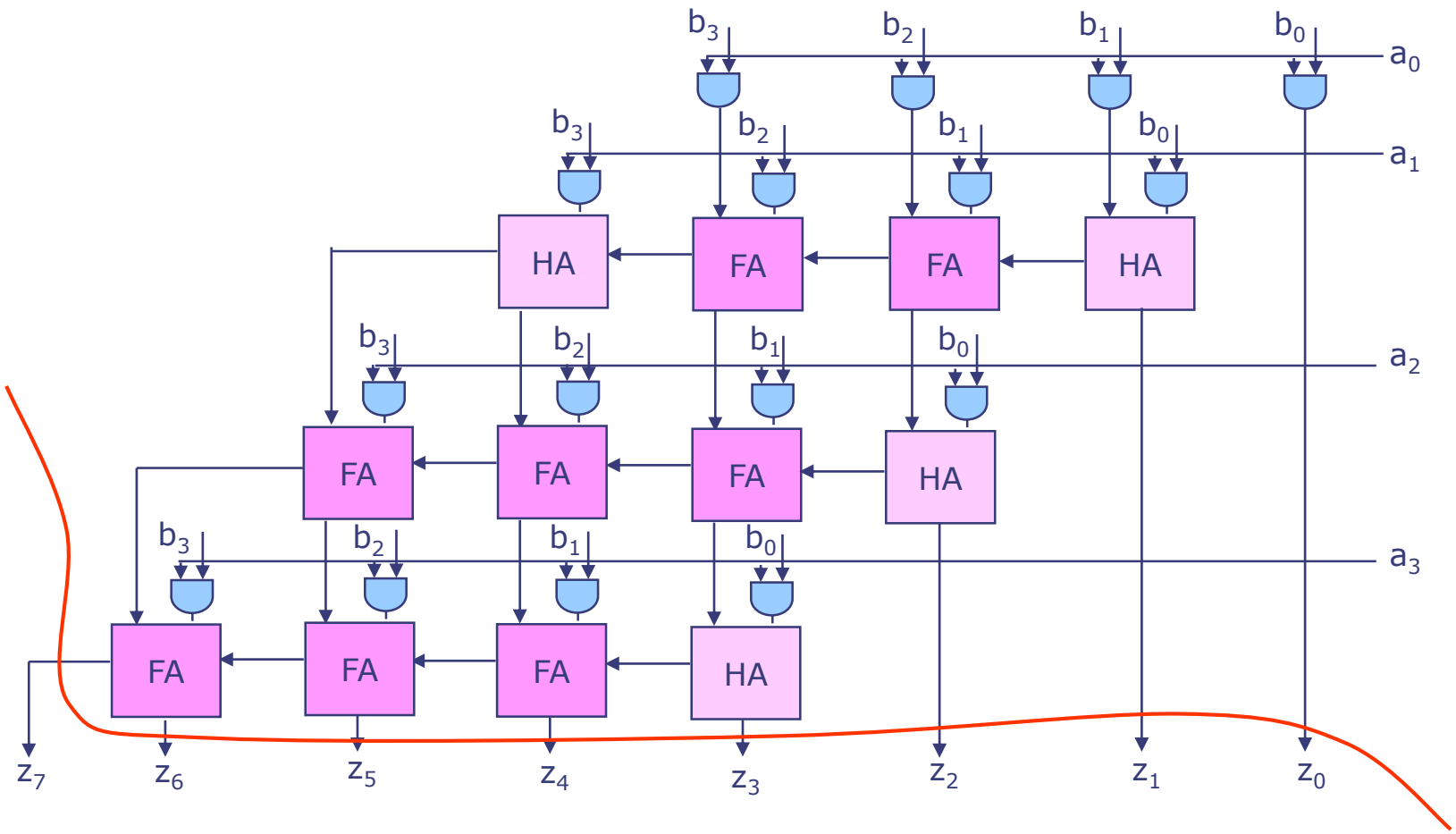


Latency = $\Theta(N)$
Throughput = $\Theta(1/N)$
Area = $\Theta(N^2)$

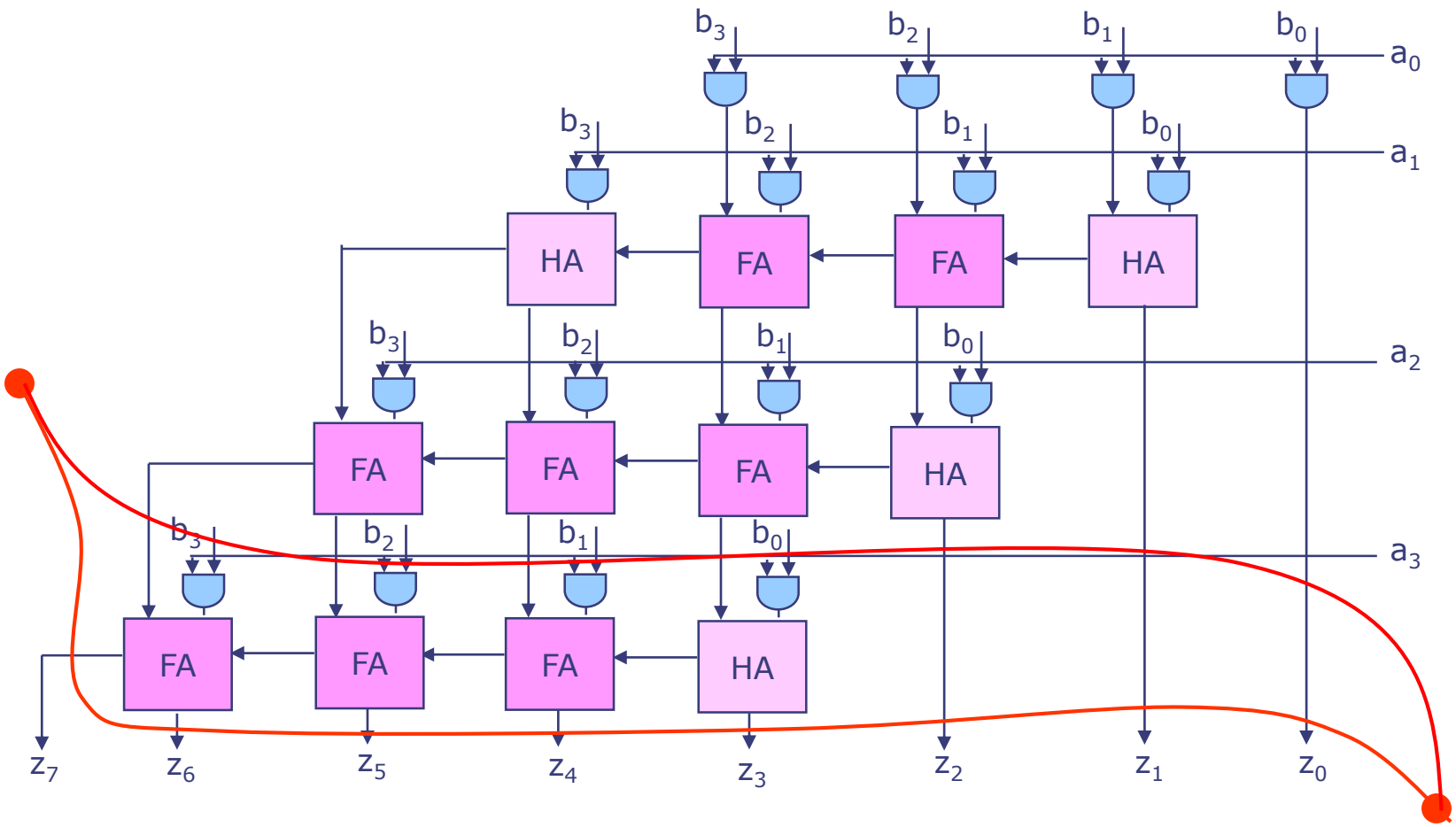
Pipelining to Increase Throughput
First Attempt



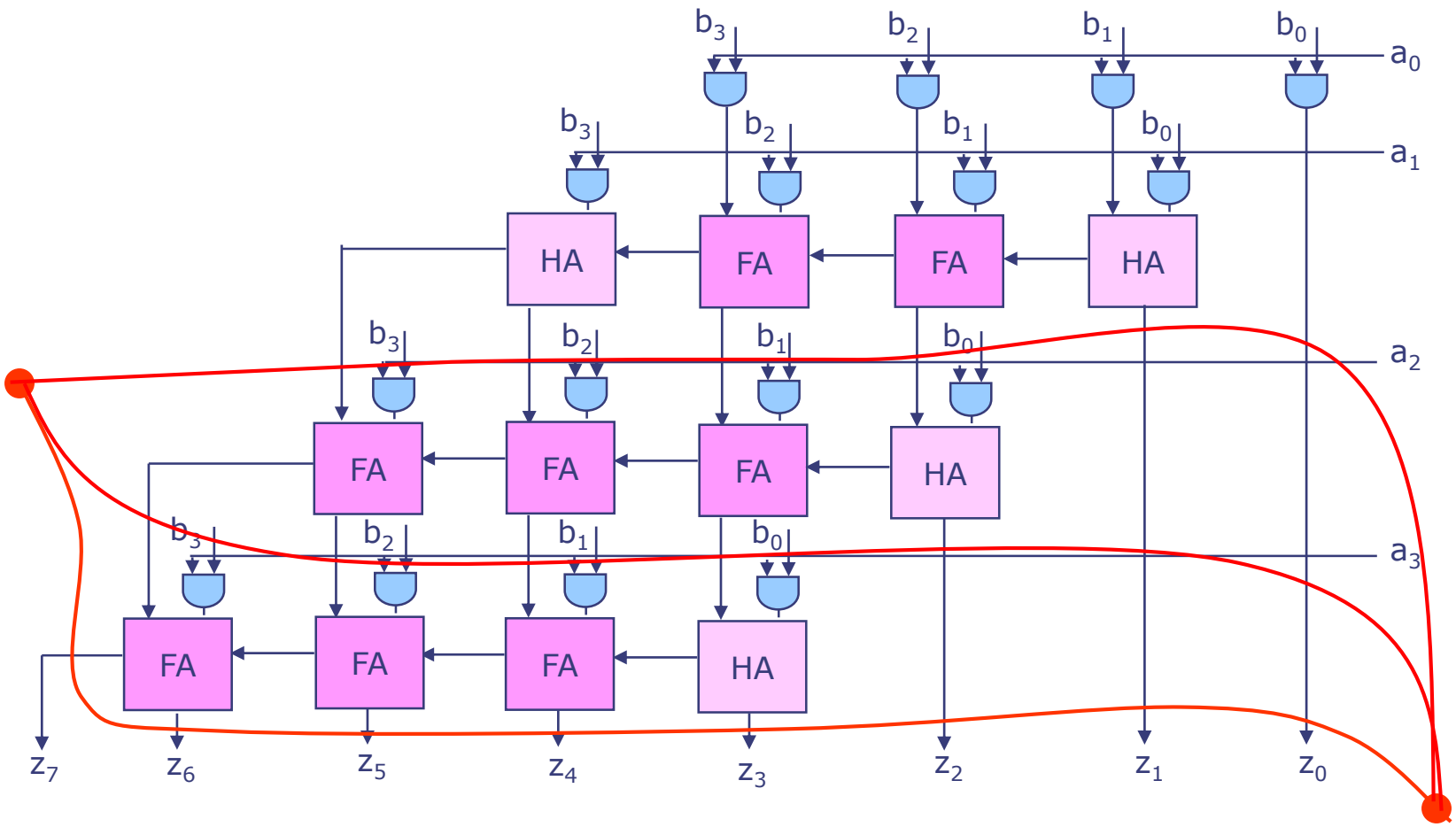
Pipelining to Increase Throughput
First Attempt



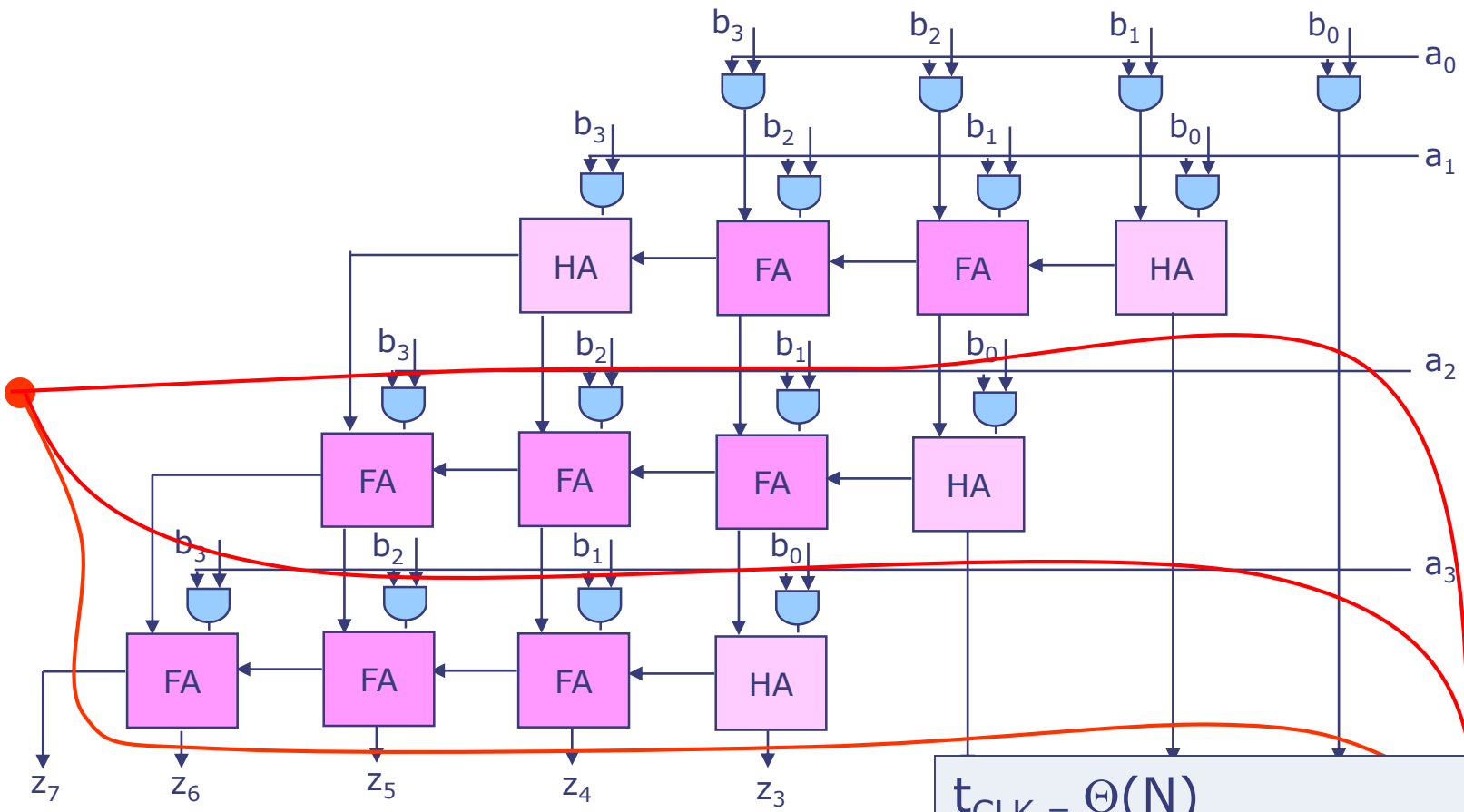
Pipelining to Increase Throughput
First Attempt



Pipelining to Increase Throughput
First Attempt

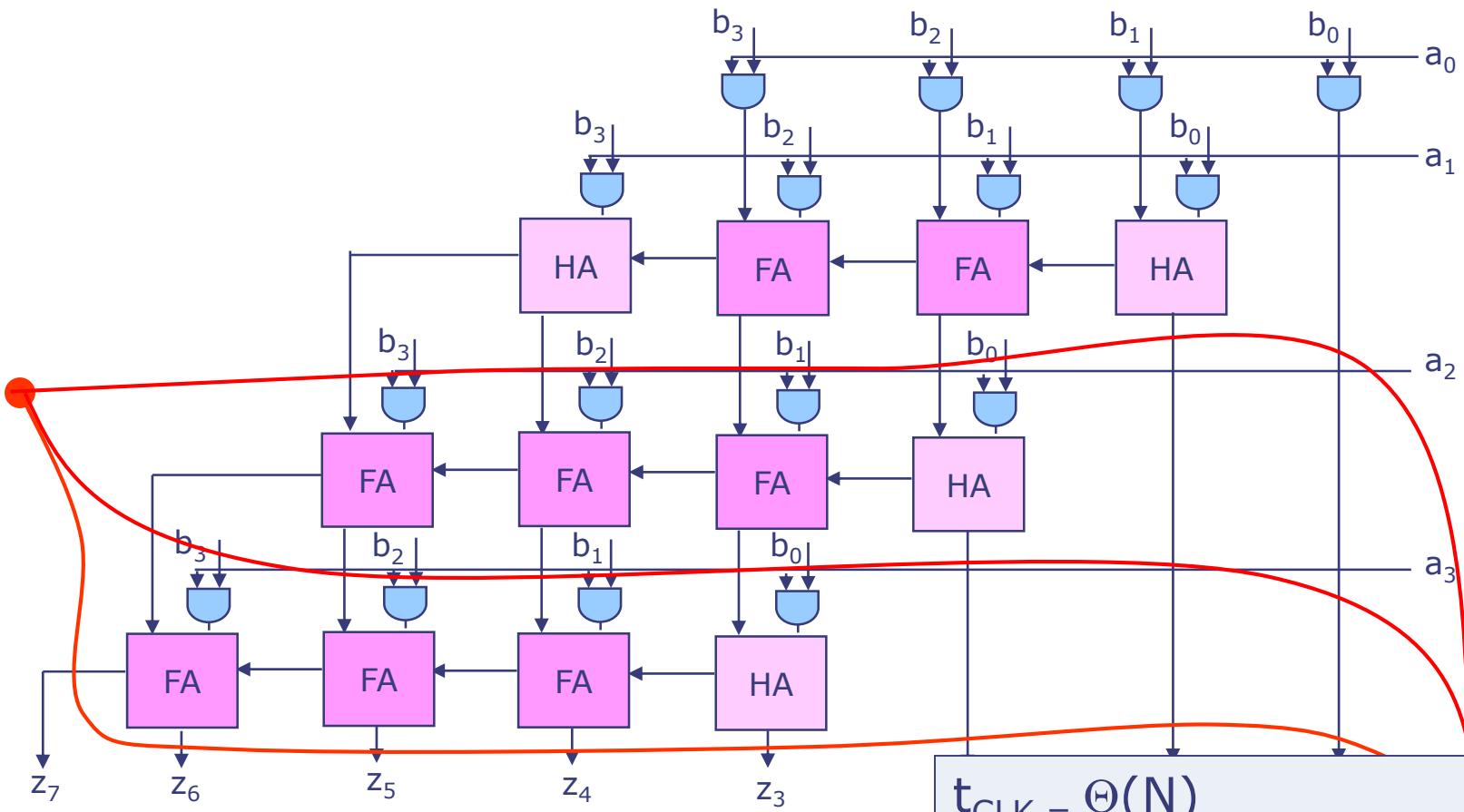


Pipelining to Increase Throughput
First Attempt



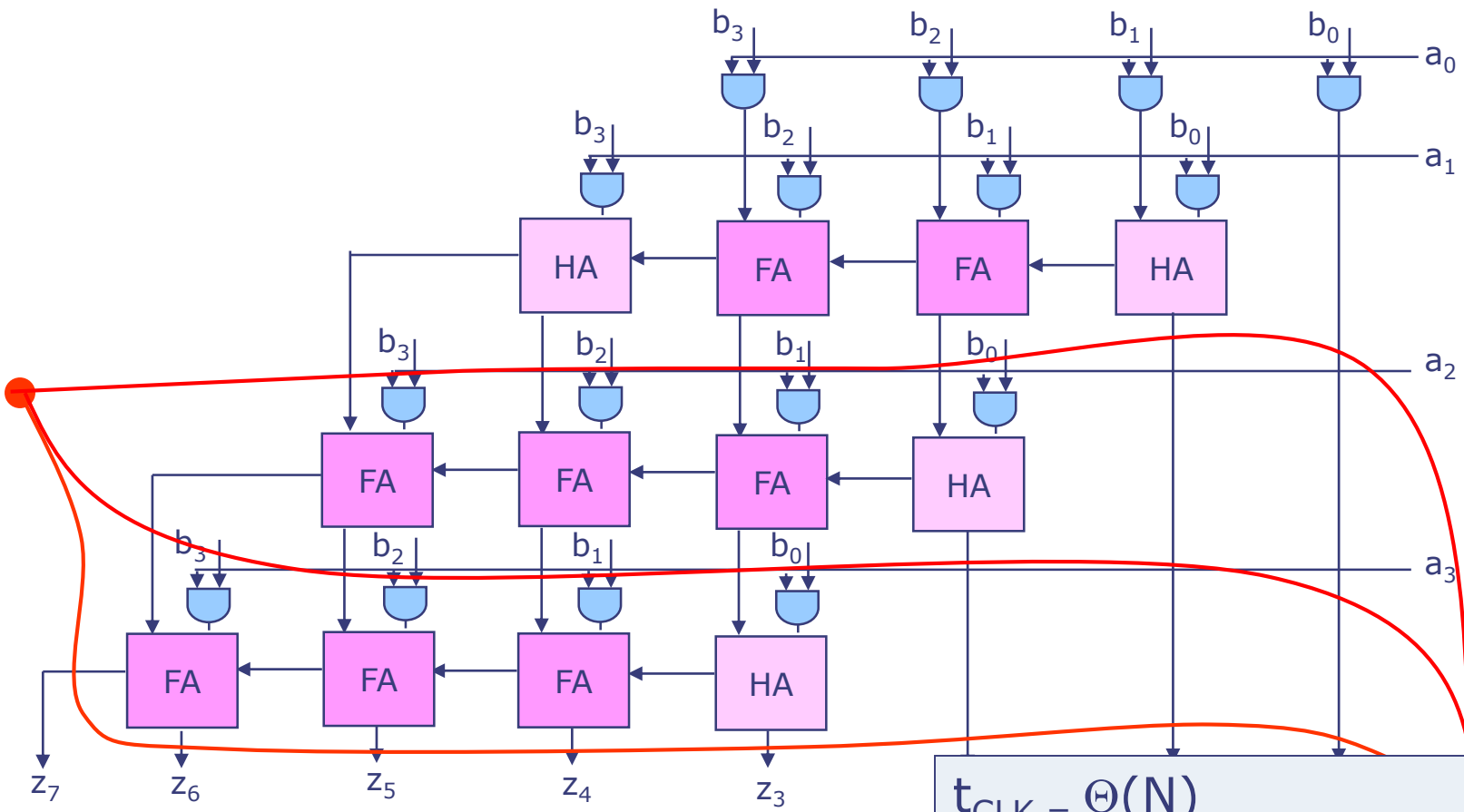
$t_{CLK} = \Theta(N)$

Pipelining to Increase Throughput
First Attempt



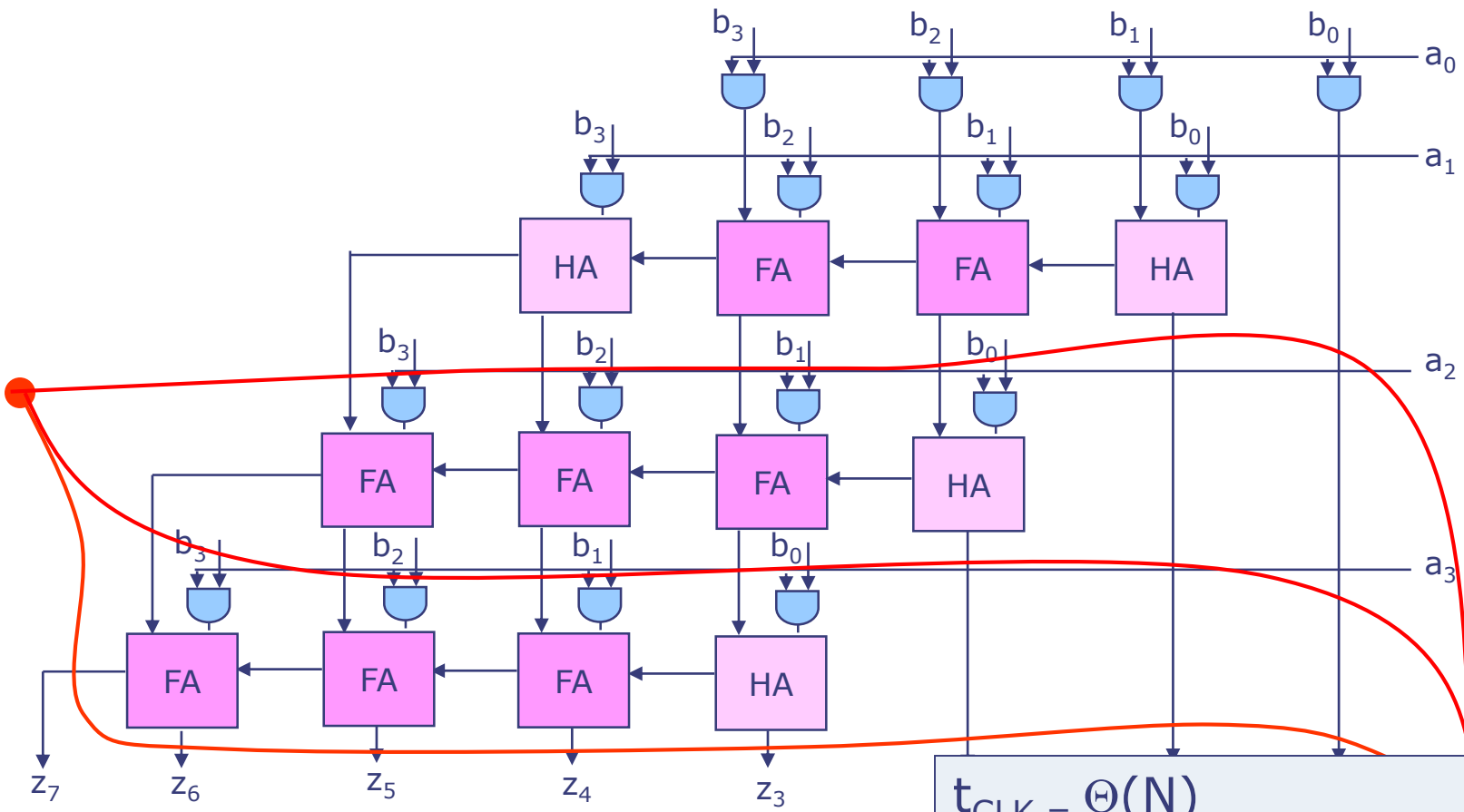
$t_{CLK} = \Theta(N)$
stages = $\Theta(N)$

Pipelining to Increase Throughput
First Attempt



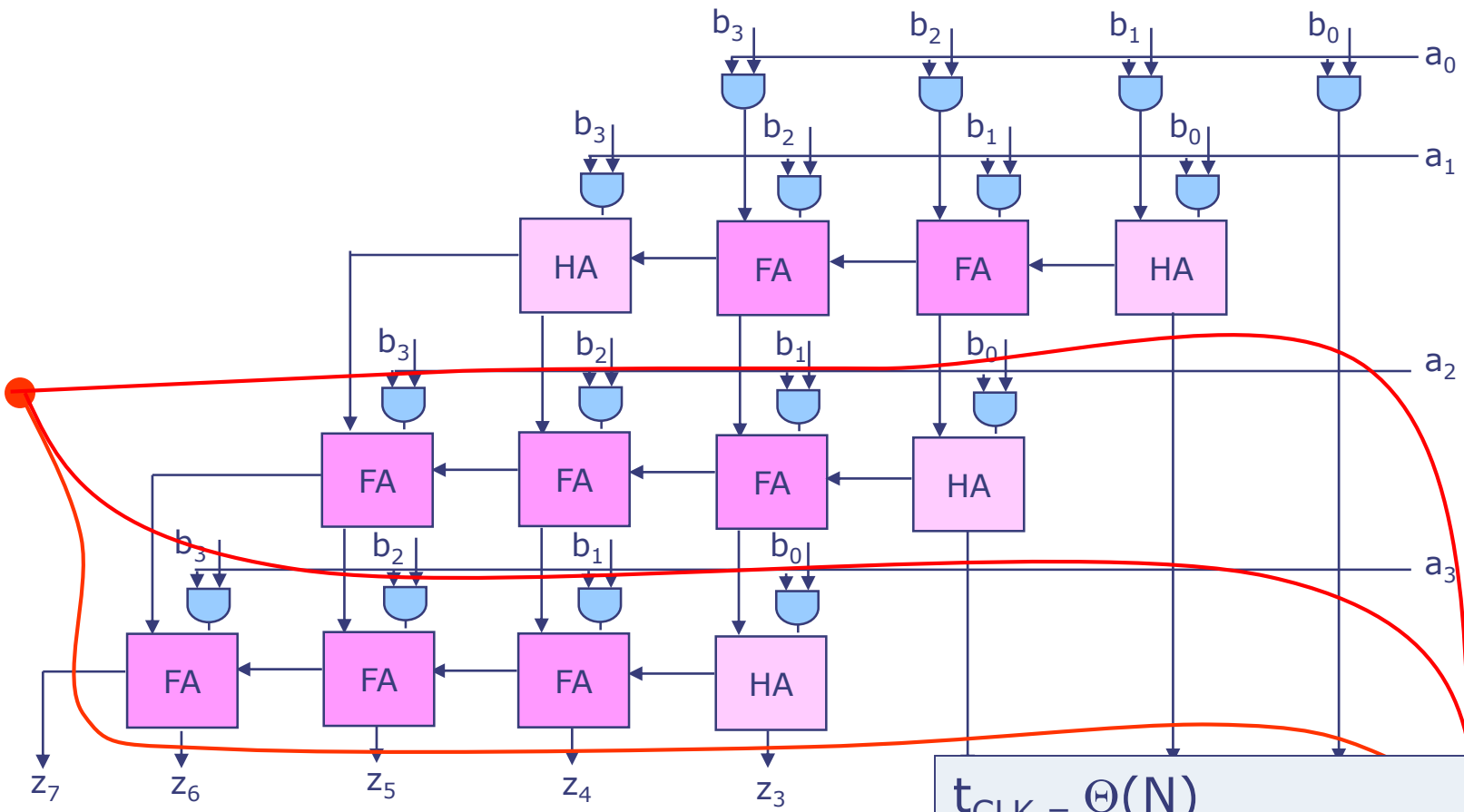
$t_{CLK} = \Theta(N)$
# stages = $\Theta(N)$
Latency = $\Theta(N^2)$

Pipelining to Increase Throughput
First Attempt



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

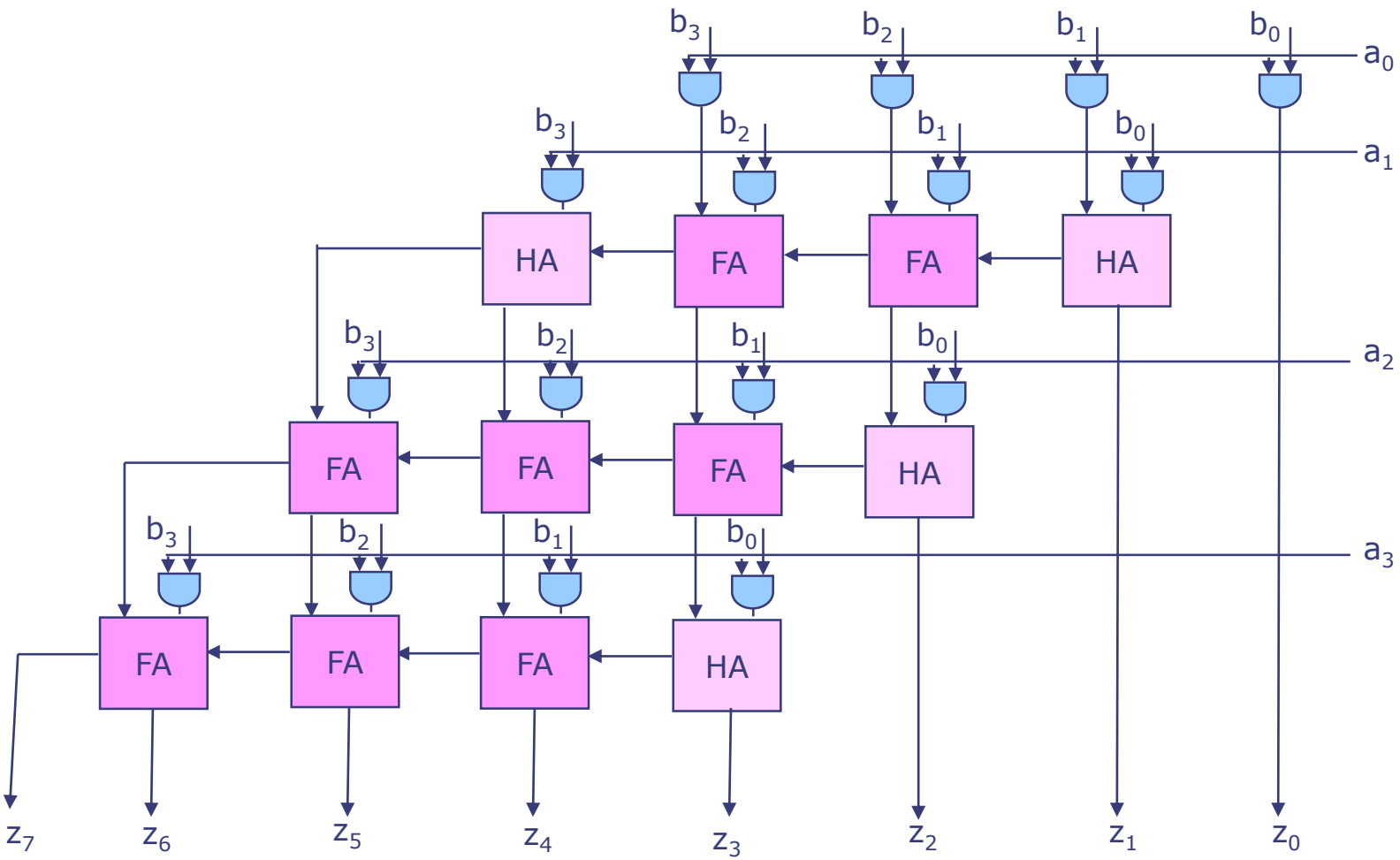
Pipelining to Increase Throughput
First Attempt



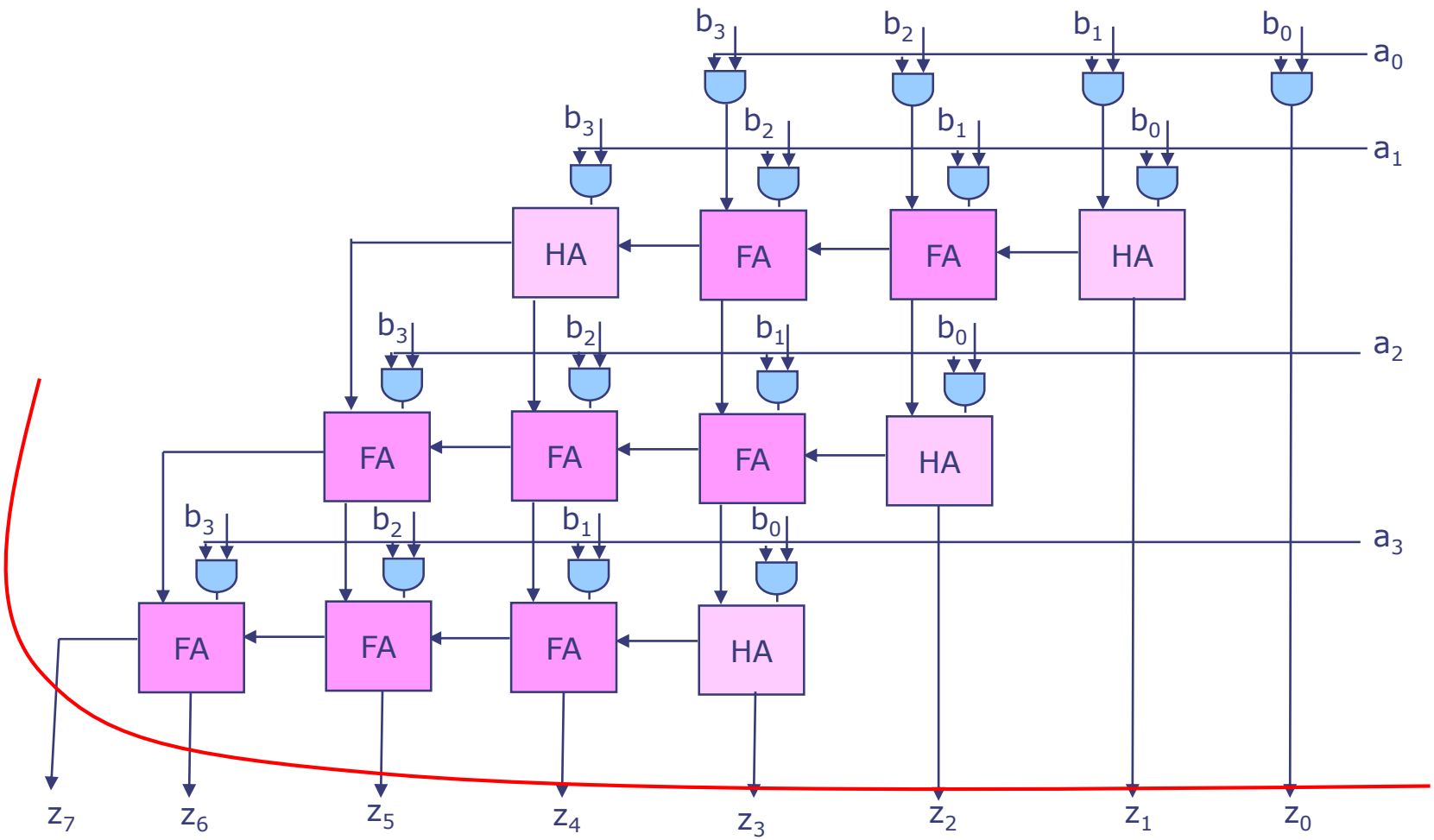
$t_{CLK} = \Theta(N)$
# stages = $\Theta(N)$
Latency = $\Theta(N^2)$
Throughput = $\Theta(1/N)$

Need to break carry chain

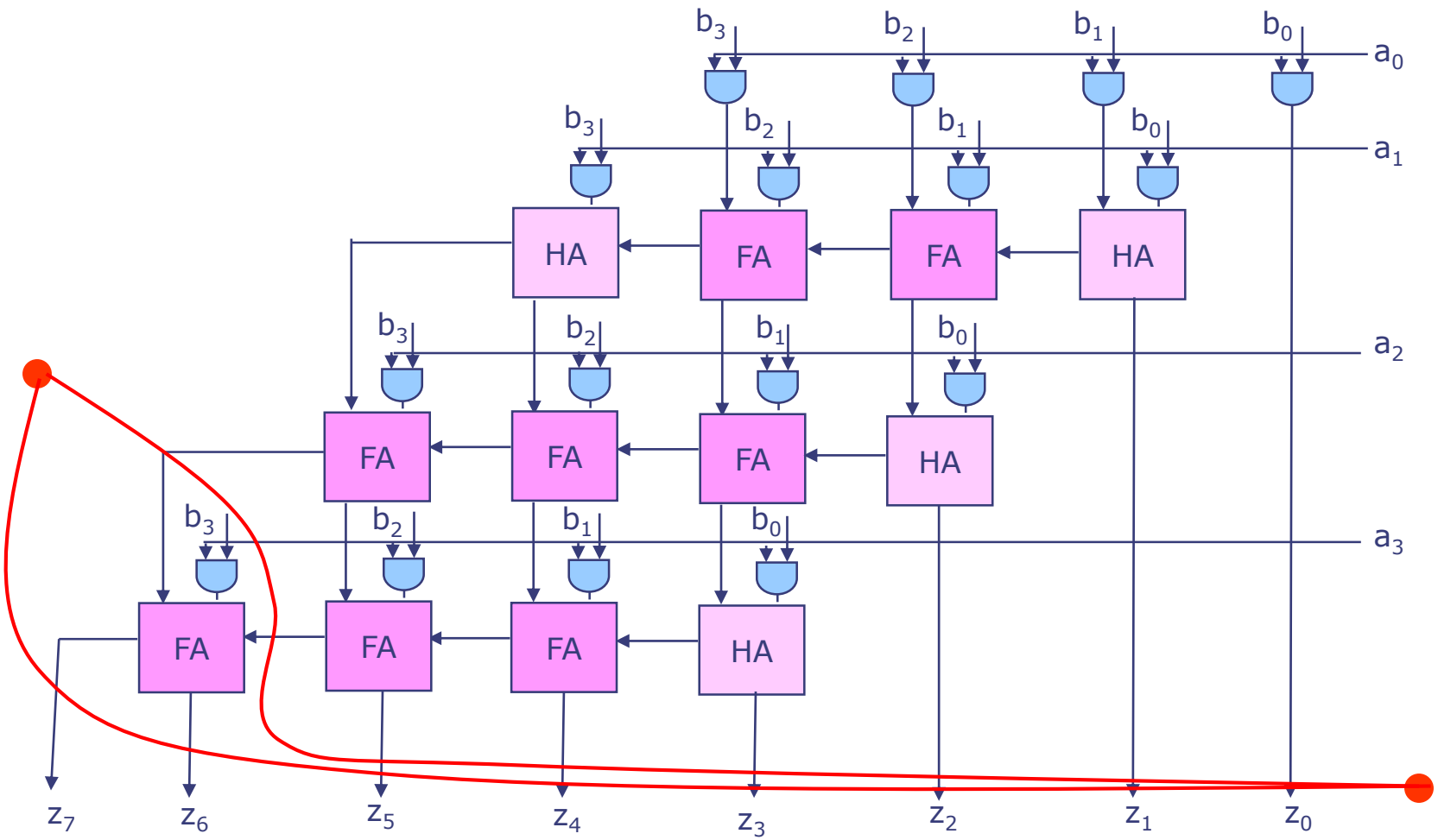
Pipelining to Increase Throughput



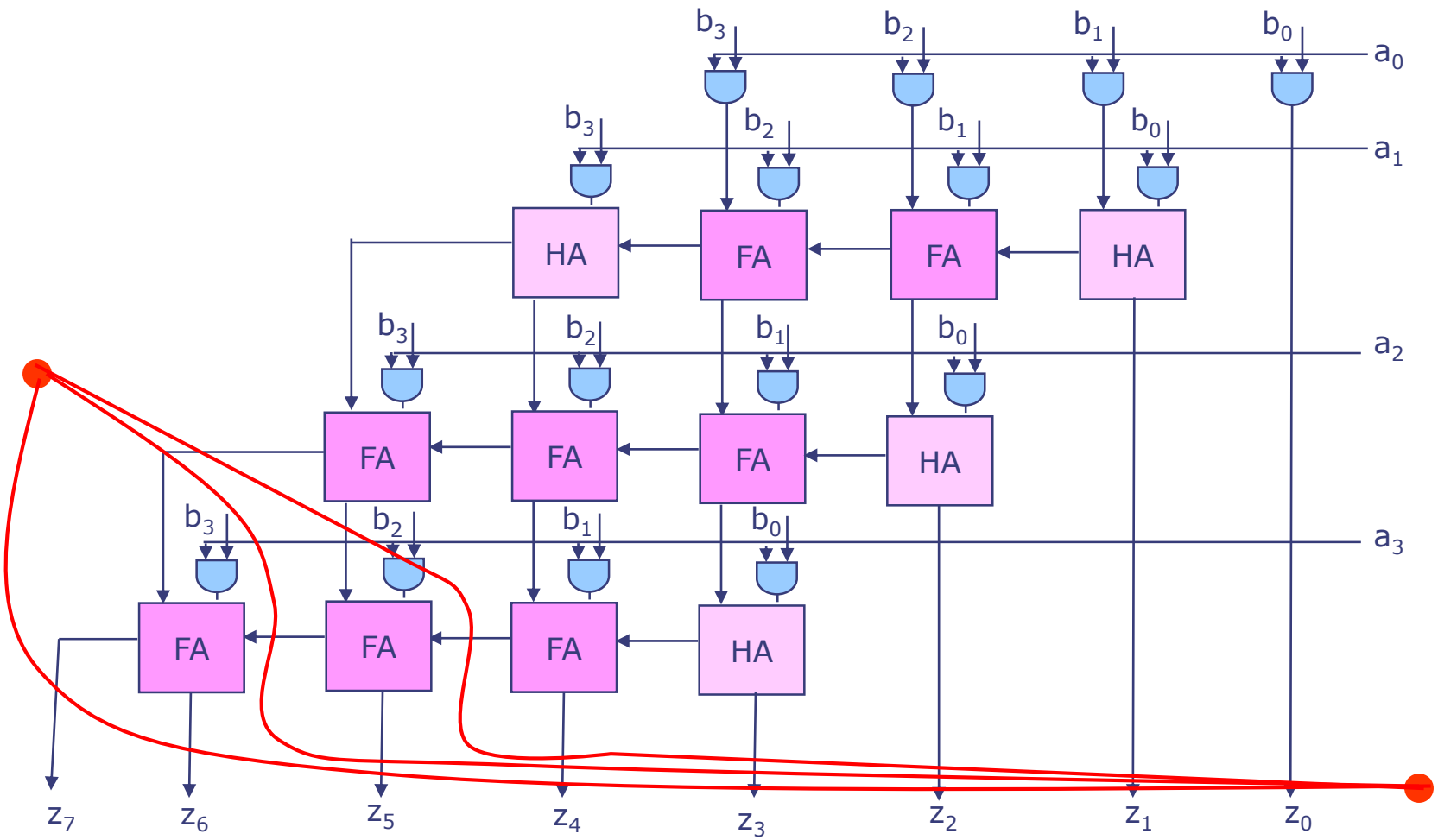
Pipelining to Increase Throughput



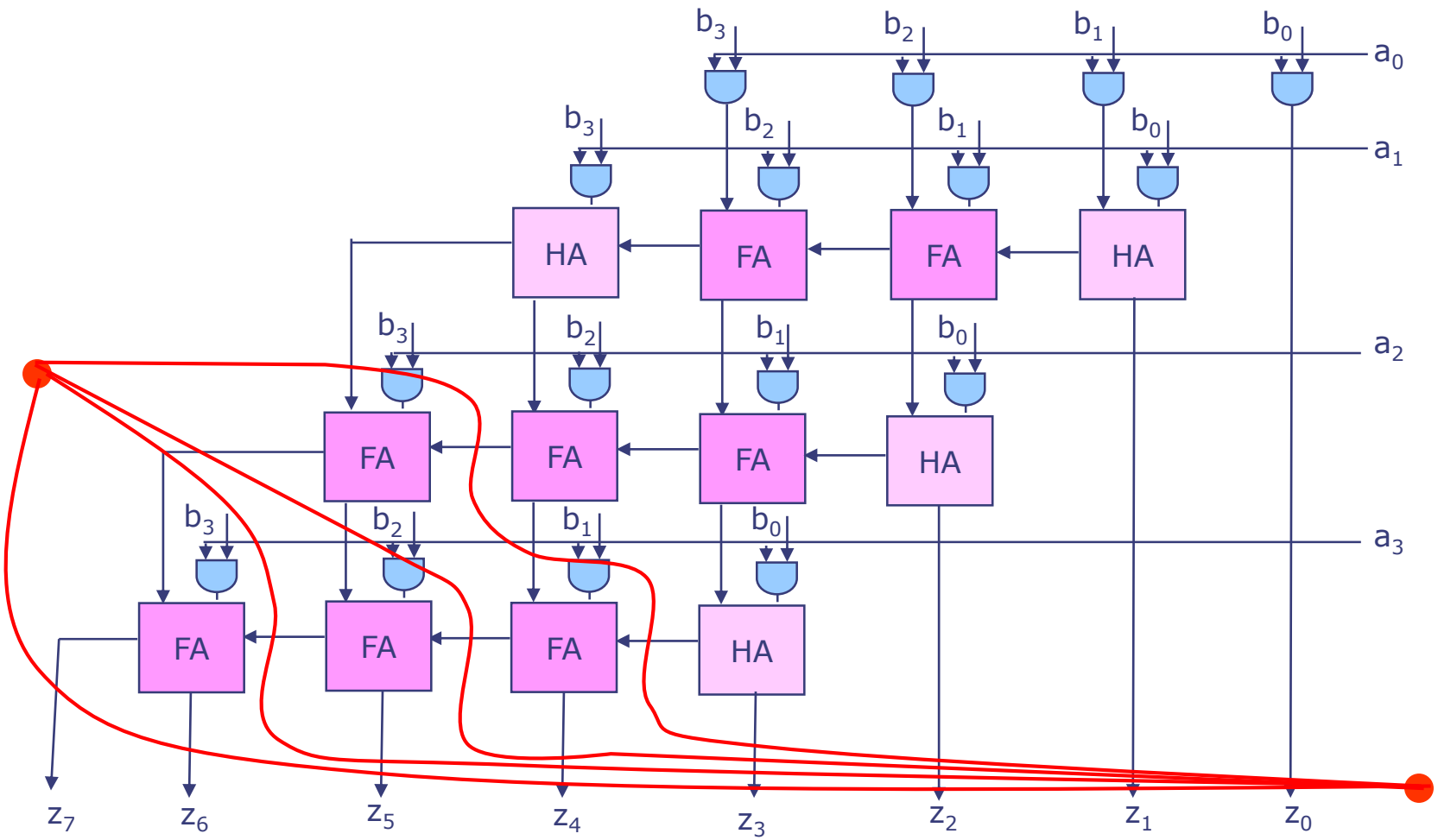
Pipelining to Increase Throughput



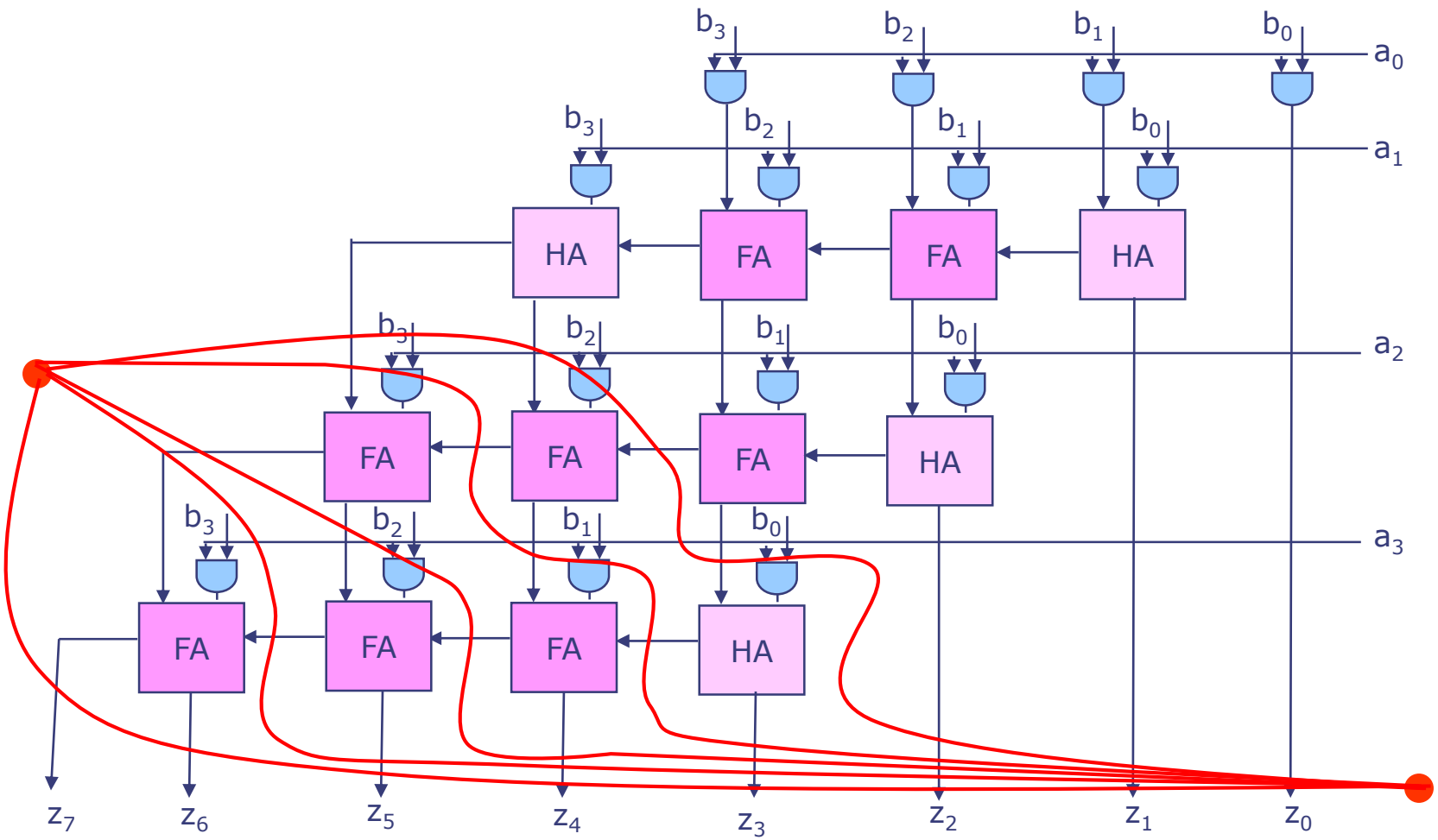
Pipelining to Increase Throughput



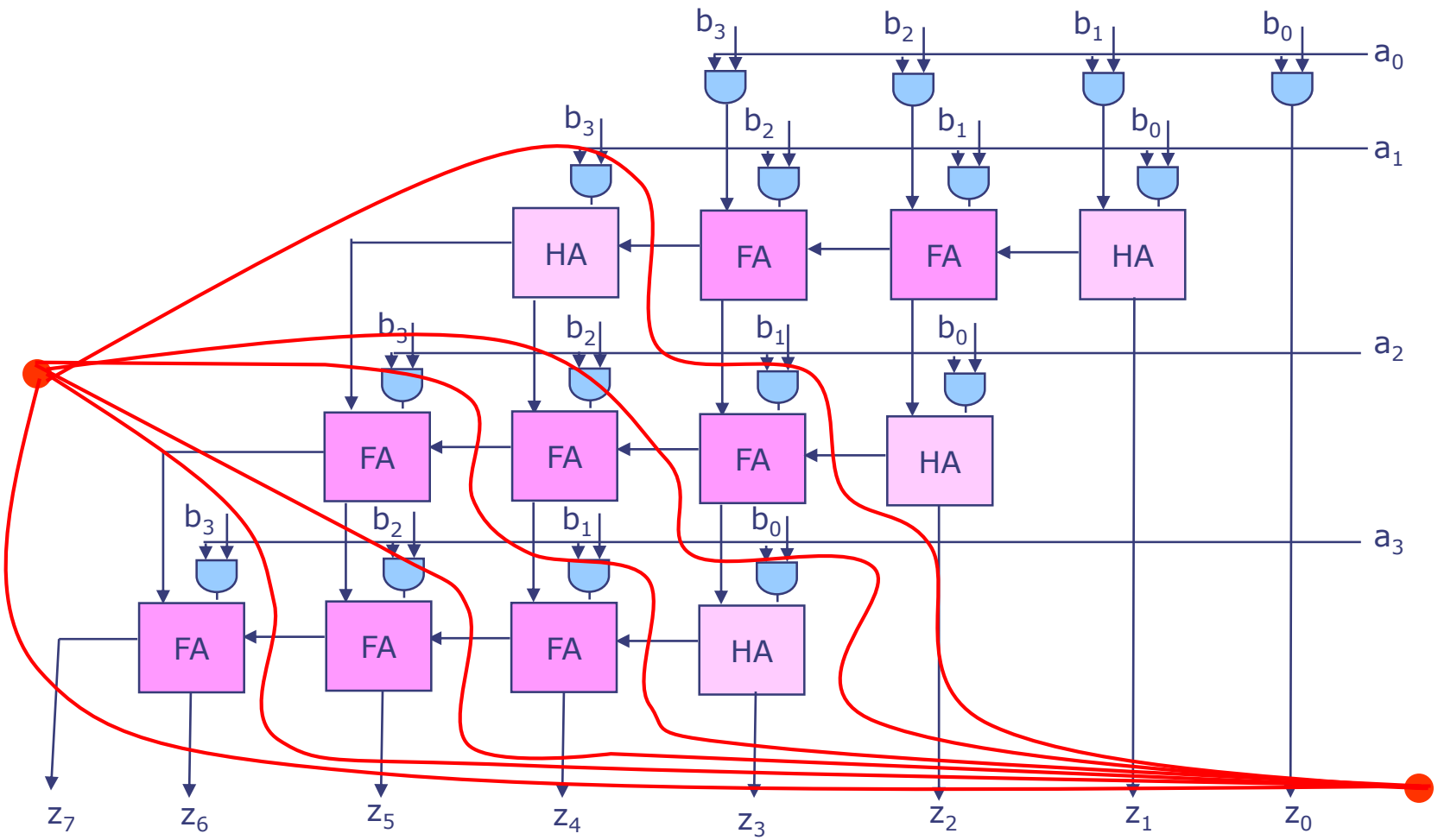
Pipelining to Increase Throughput



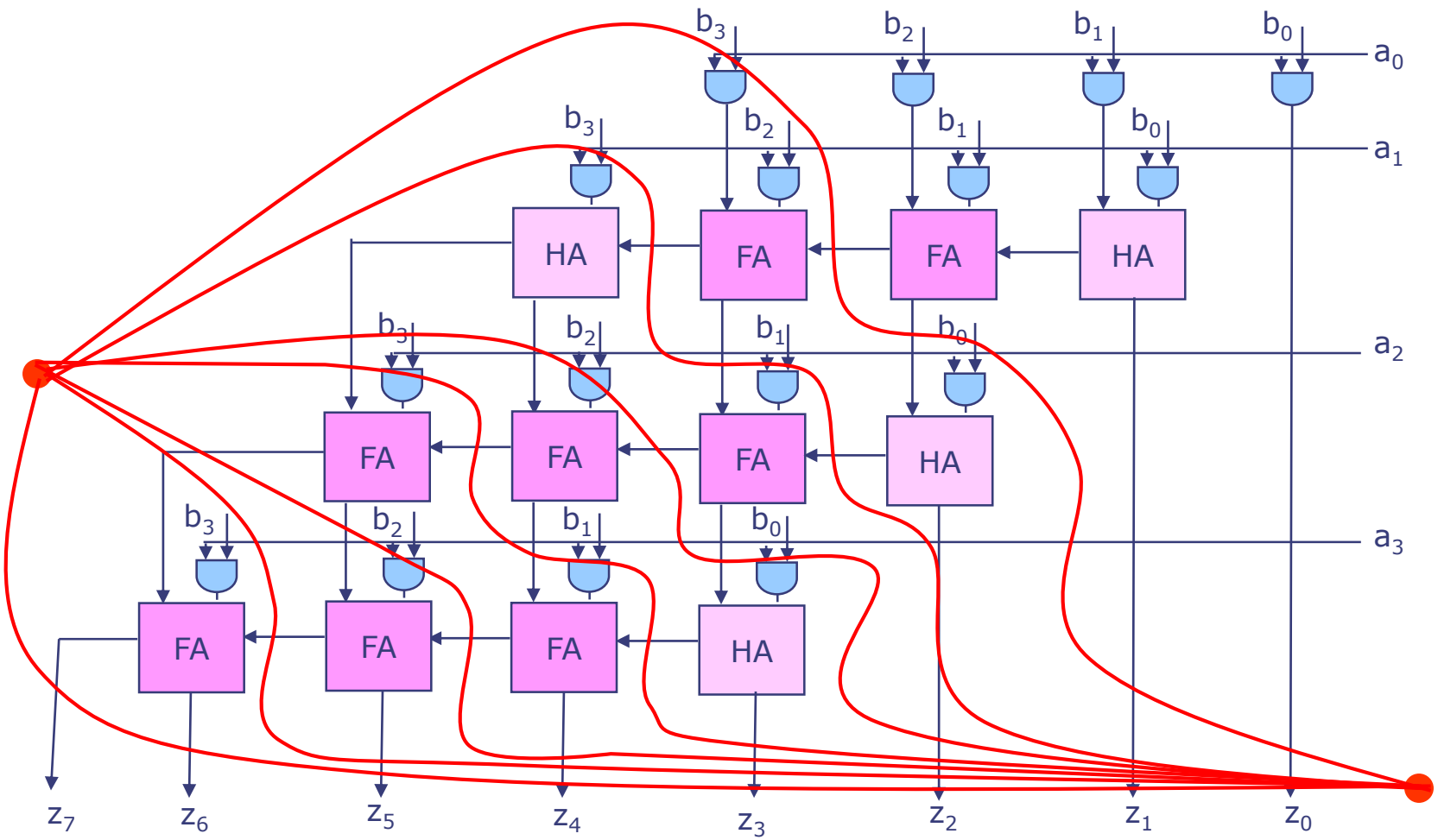
Pipelining to Increase Throughput



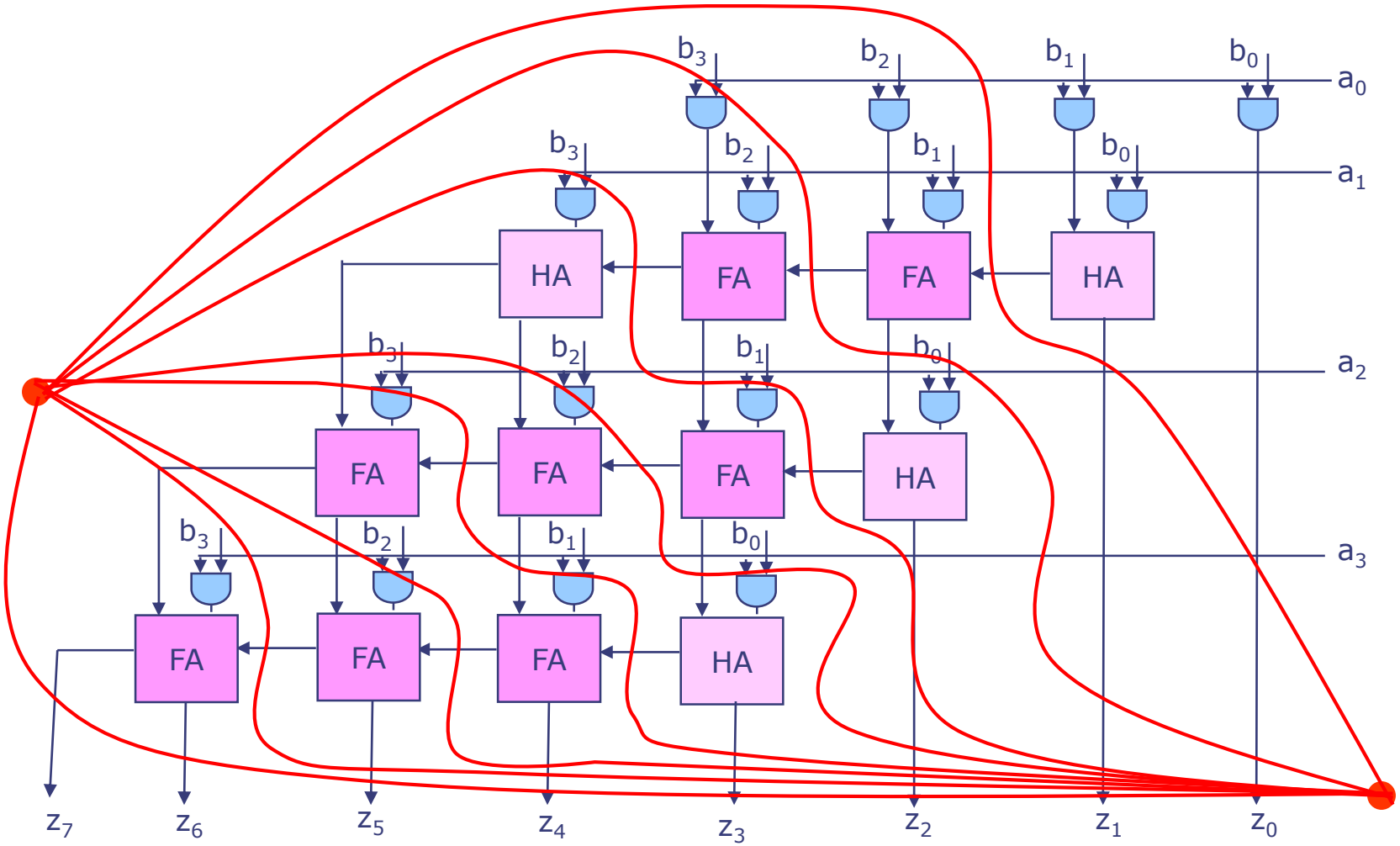
Pipelining to Increase Throughput



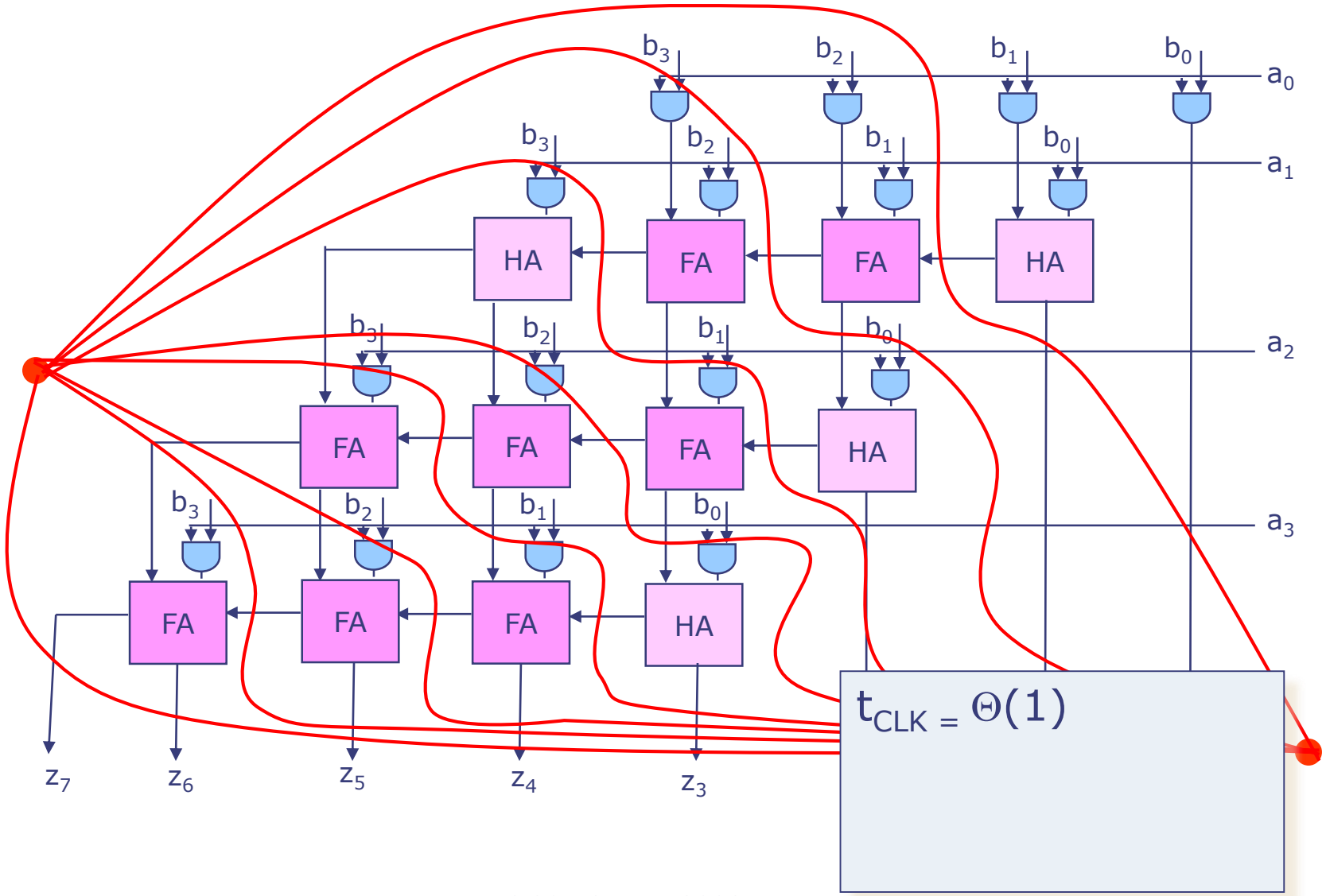
Pipelining to Increase Throughput



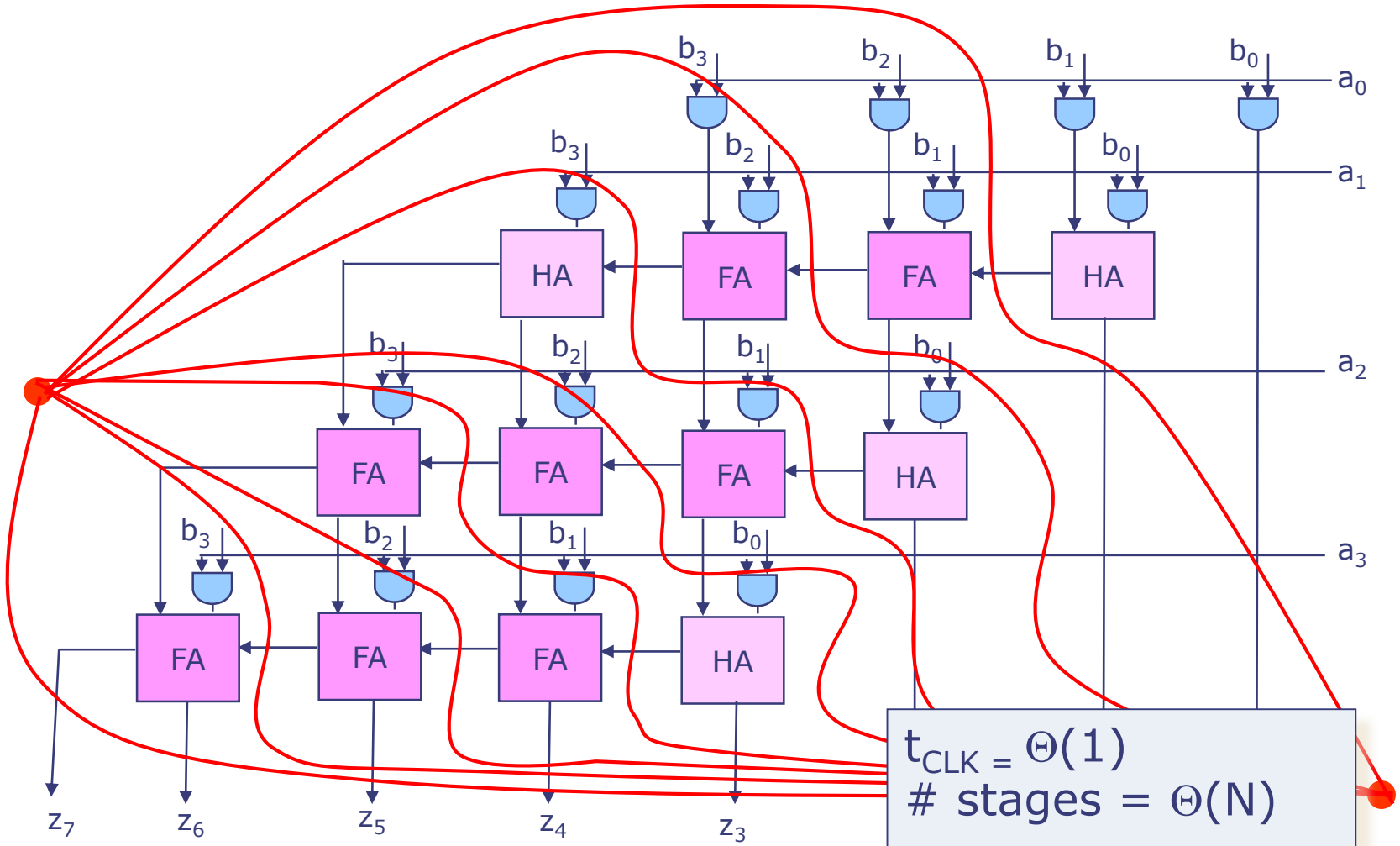
Pipelining to Increase Throughput



Pipelining to Increase Throughput



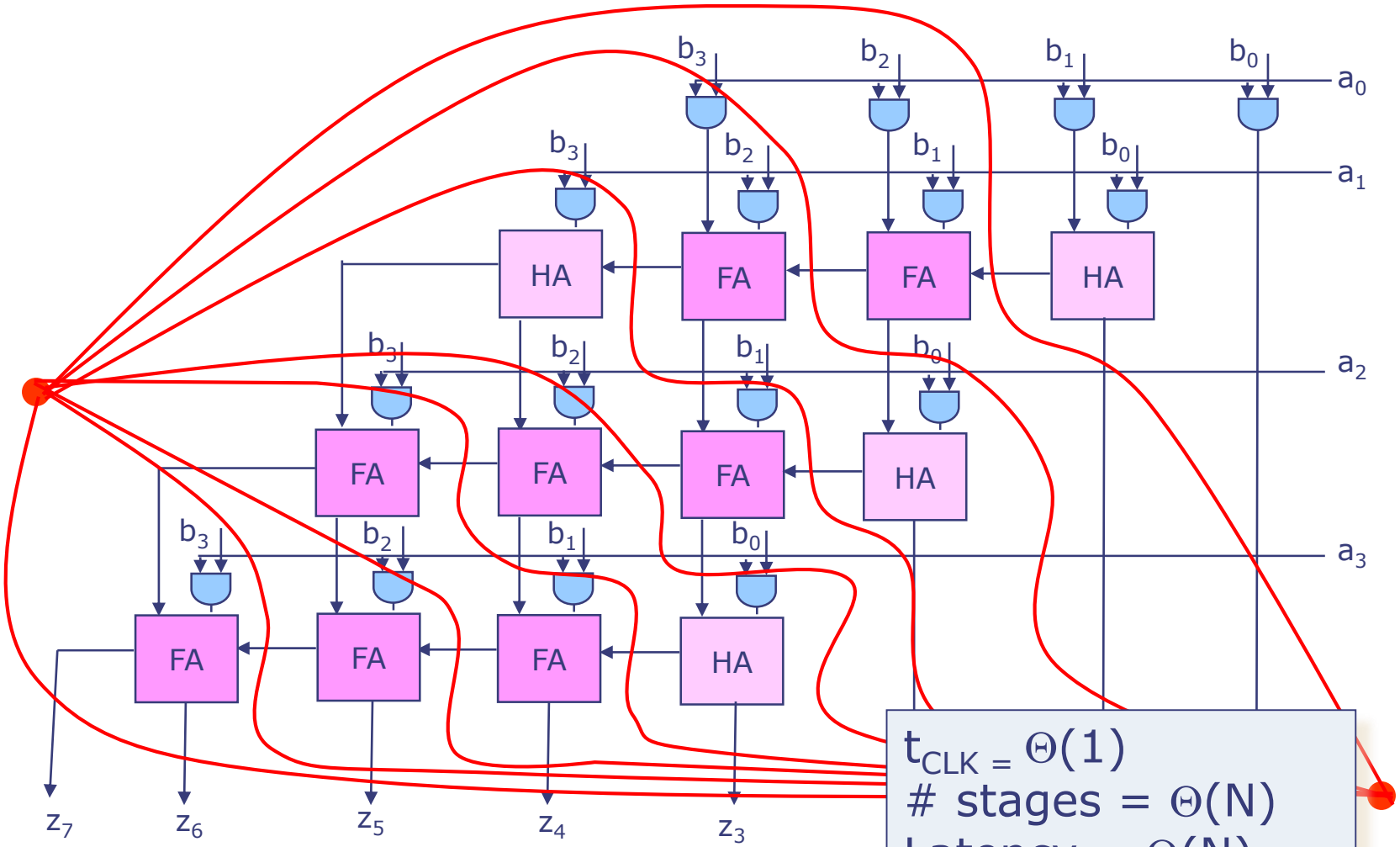
Pipelining to Increase Throughput



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

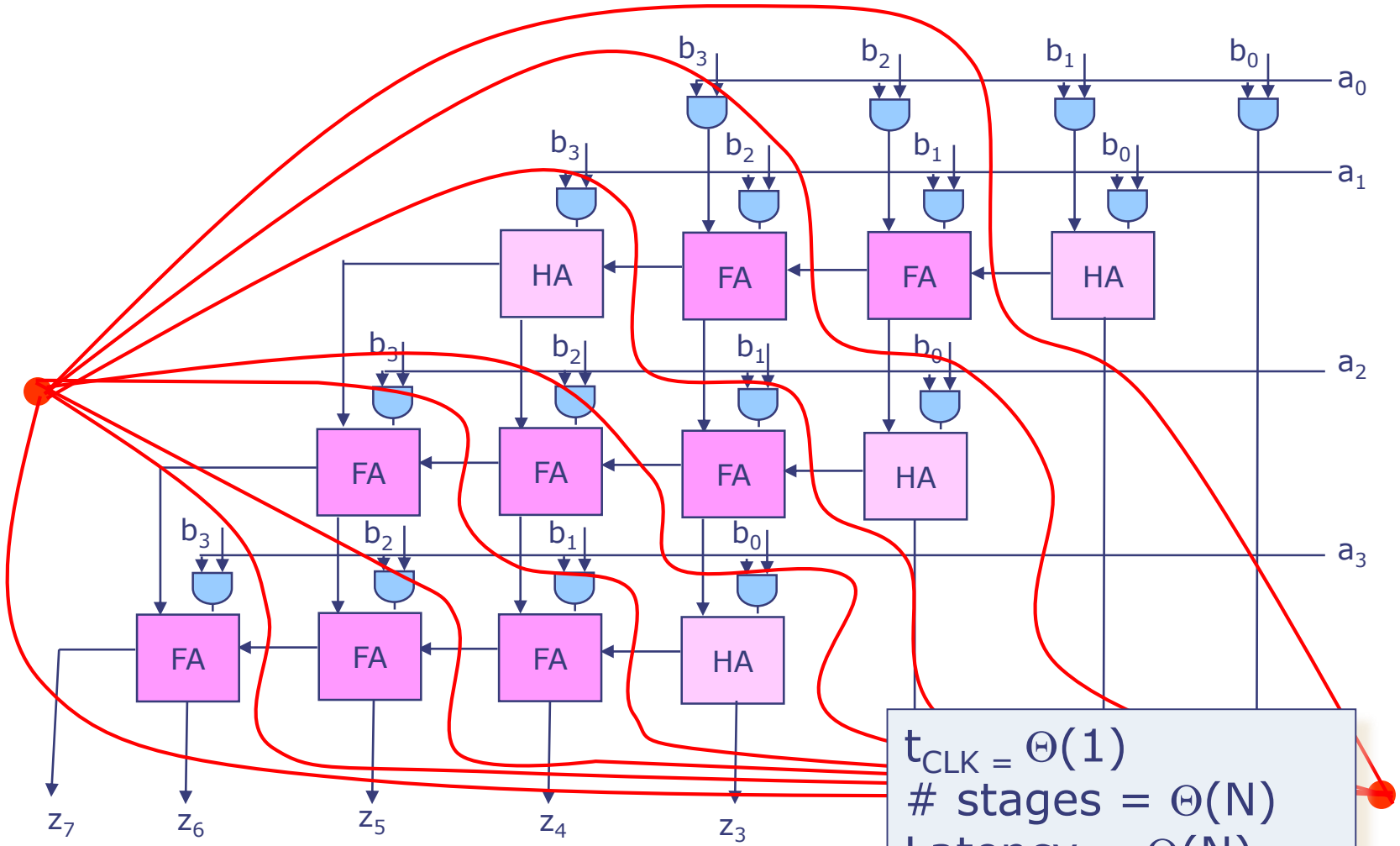
\# stages = $\Theta(N)$

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

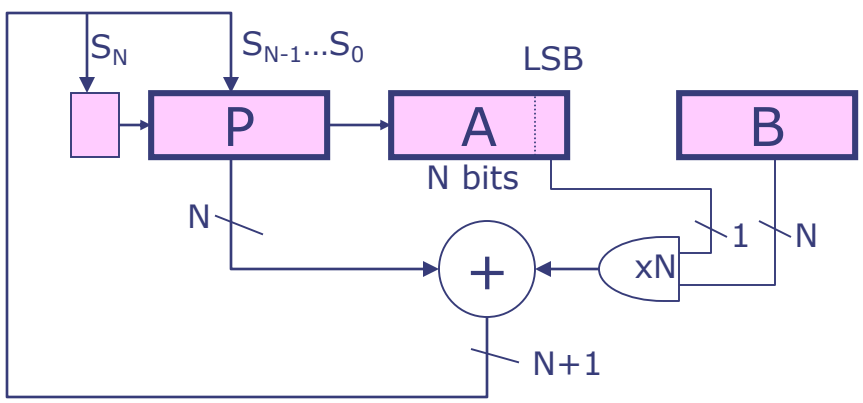
- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders

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

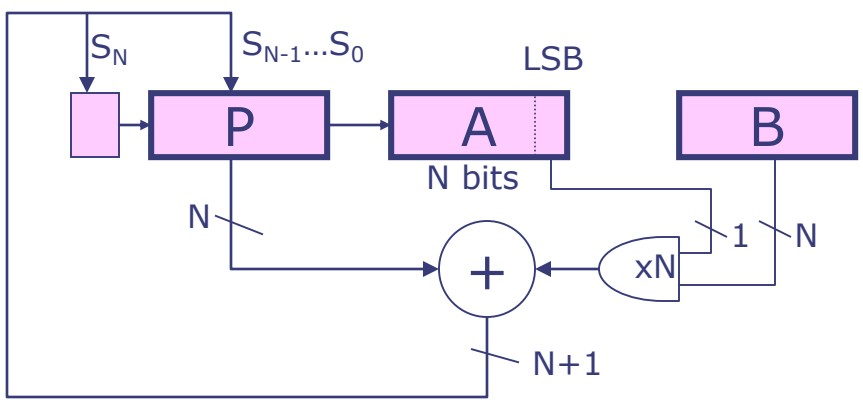
- Combinational circuits often have repetitive logic
  - Example: N-bit multiplier has N-1 adders
- Folded circuits use less combinational logic, reuse it over multiple cycles
  - Example: Implement multiplication with one adder, taking ~N cycles to perform the additions




```
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

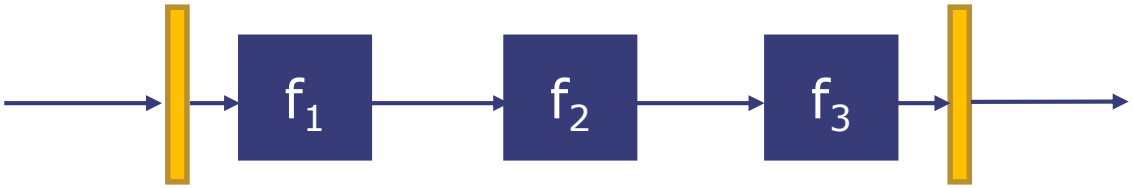



```
Init: P←0, load A&B

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
```

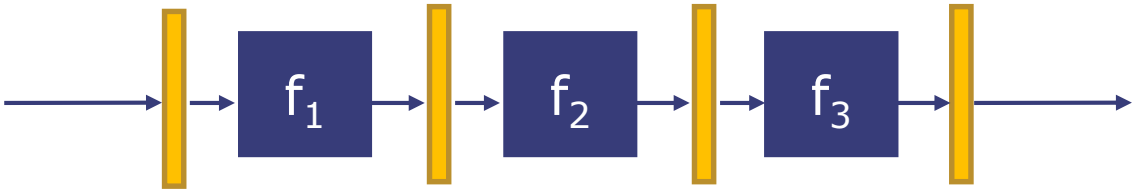
Tradeoff: reduced area, but lower throughput

Summary: Design Alternatives

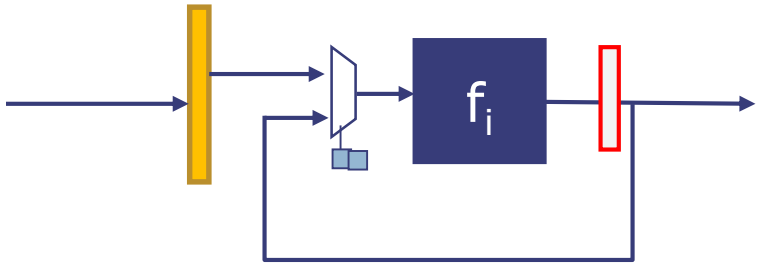
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

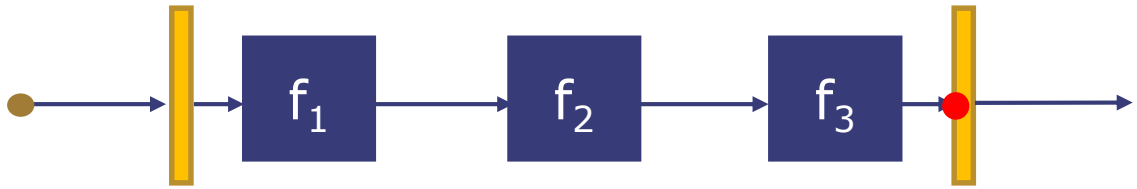


Folded reuse a block, multi-cycle (C)

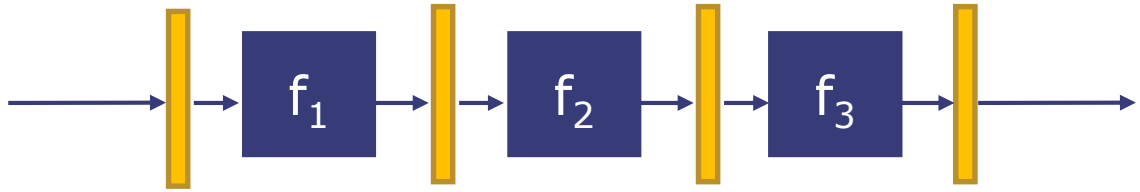


Summary: Design Alternatives

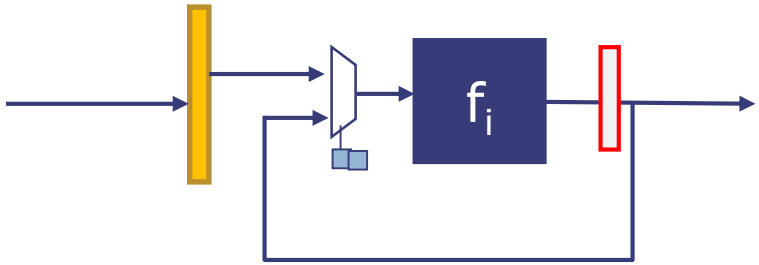
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

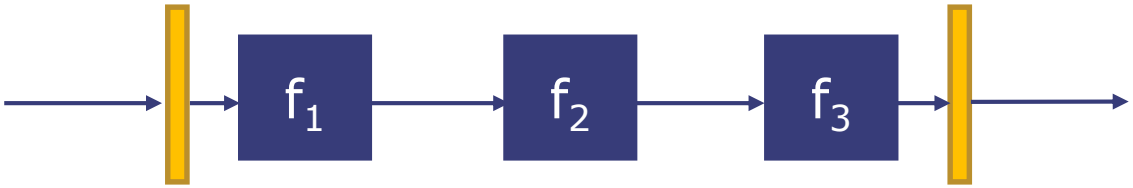


Folded reuse a block, multi-cycle (C)

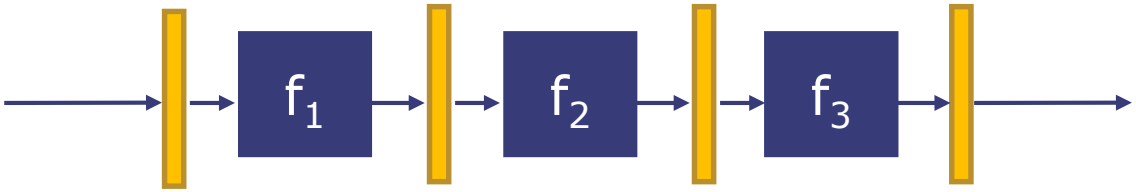


Summary: Design Alternatives

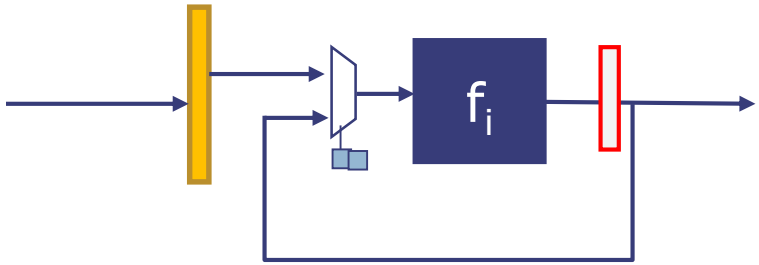
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

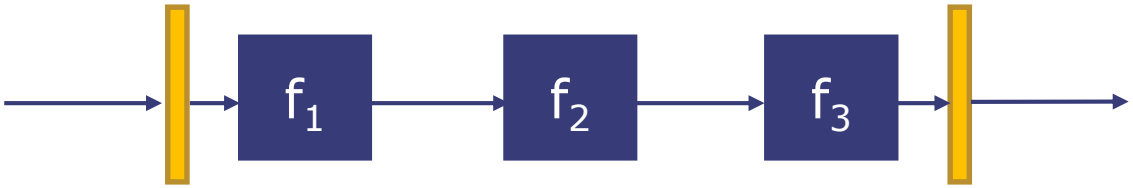


Folded reuse a block, multi-cycle (C)

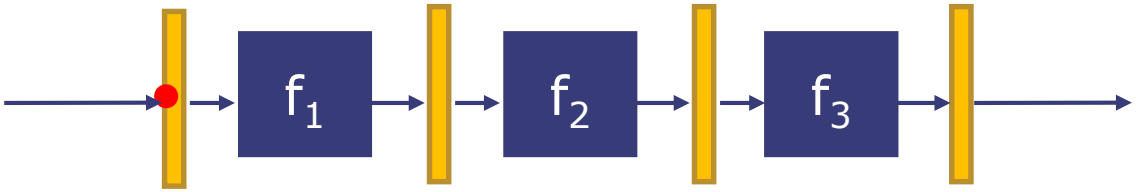


Summary: Design Alternatives

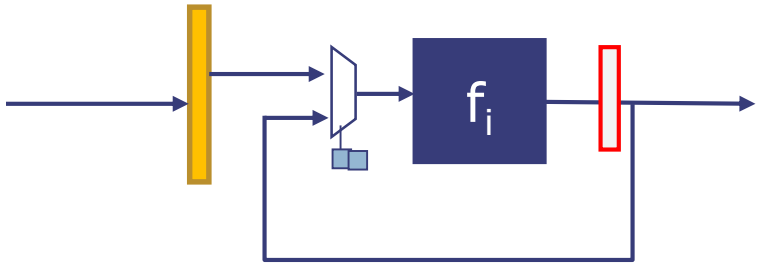
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

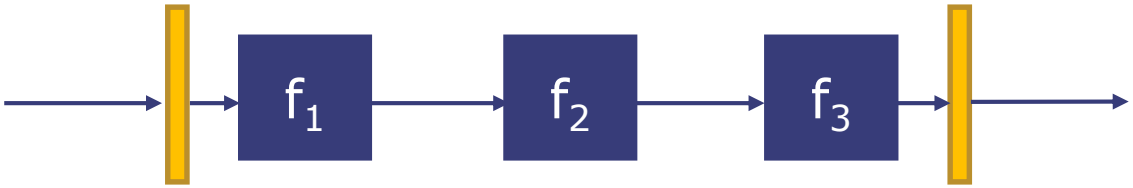


Folded reuse a block, multi-cycle (C)

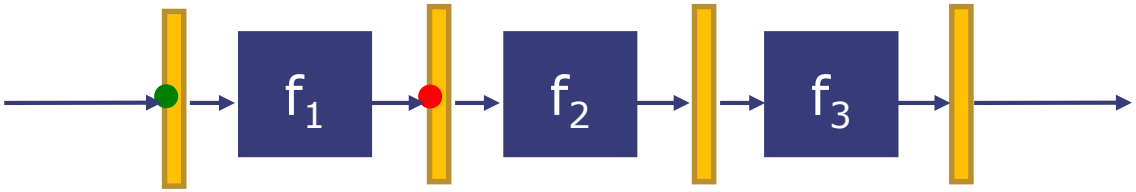


Summary: Design Alternatives

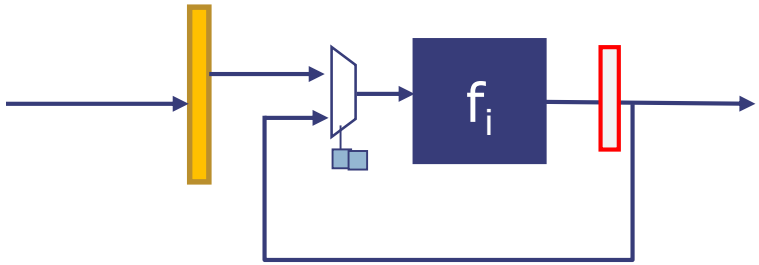
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

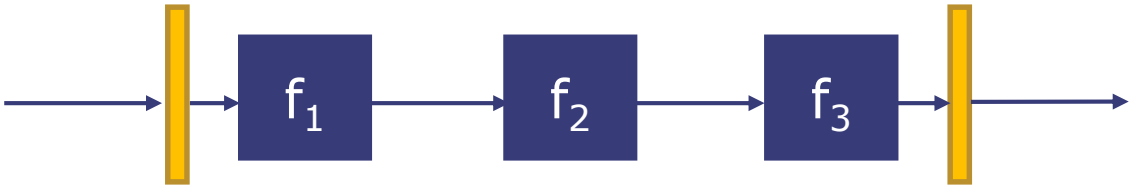


Folded reuse a block, multi-cycle (C)

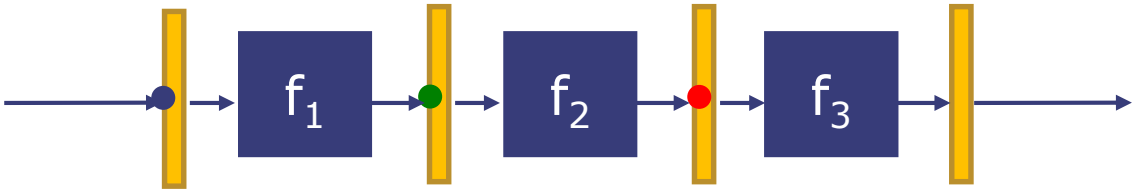


Summary: Design Alternatives

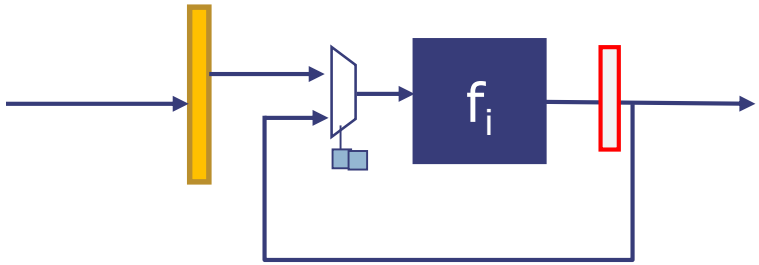
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

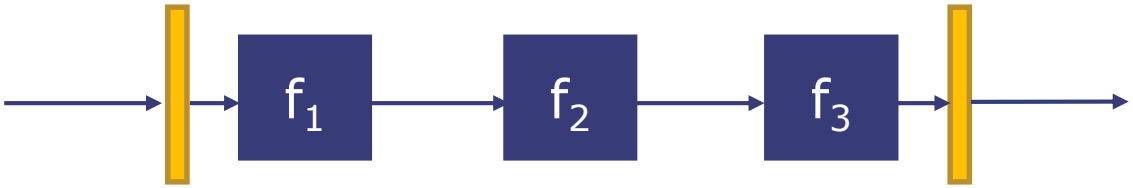


Folded reuse a block, multi-cycle (C)

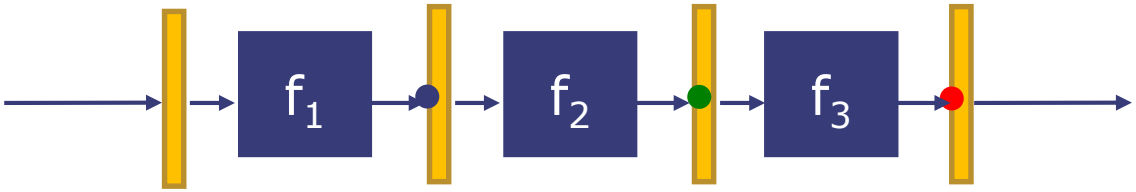


Summary: Design Alternatives

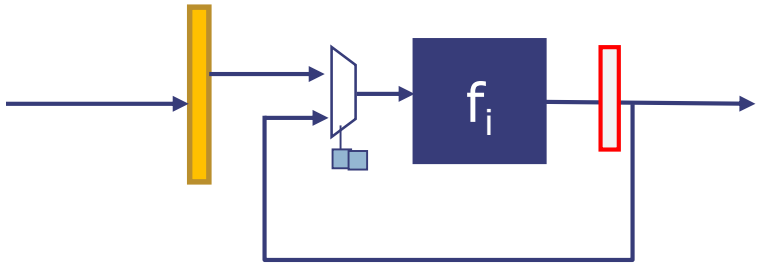
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

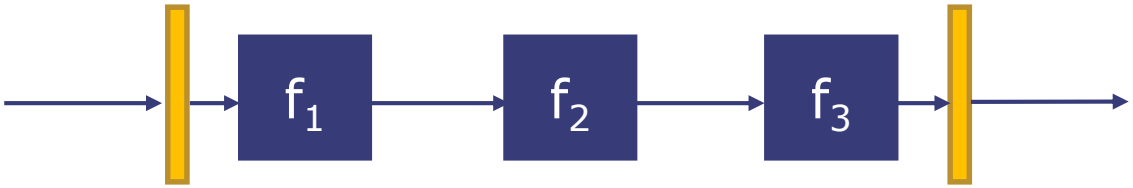


Folded reuse a block, multi-cycle (C)

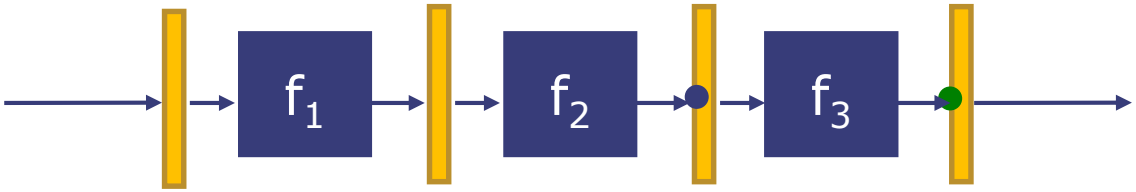


Summary: Design Alternatives

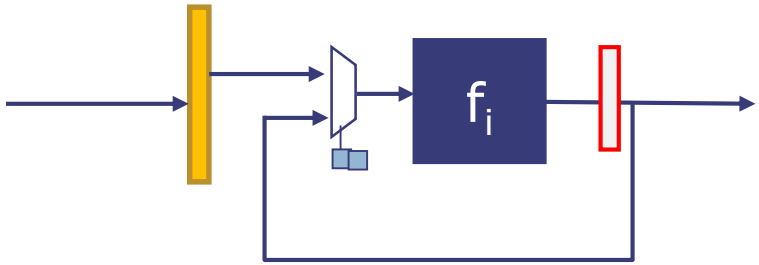
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

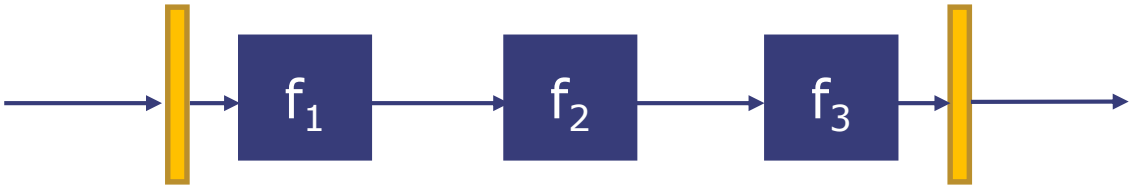


Folded reuse a block, multi-cycle (C)

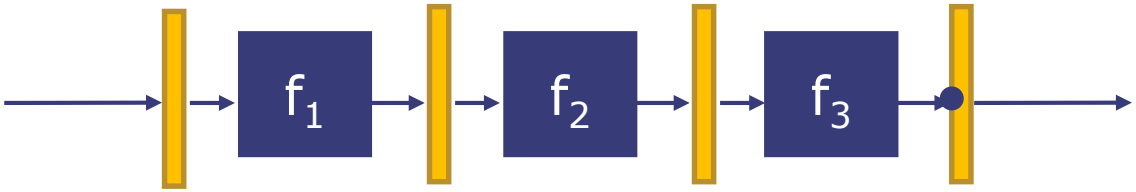


Summary: Design Alternatives

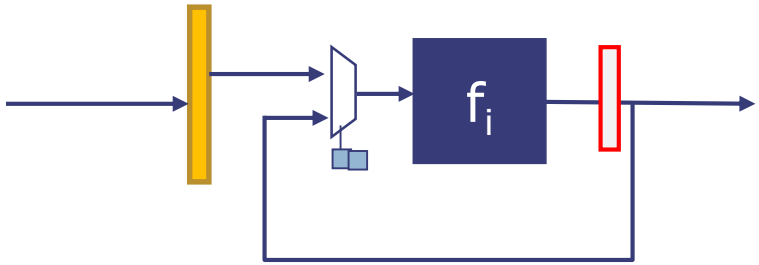
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

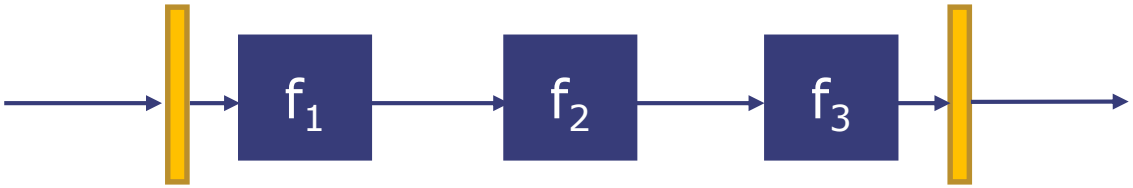


Folded reuse a block, multi-cycle (C)

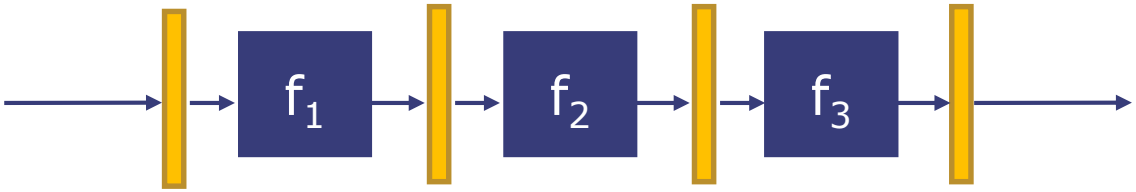


Summary: Design Alternatives

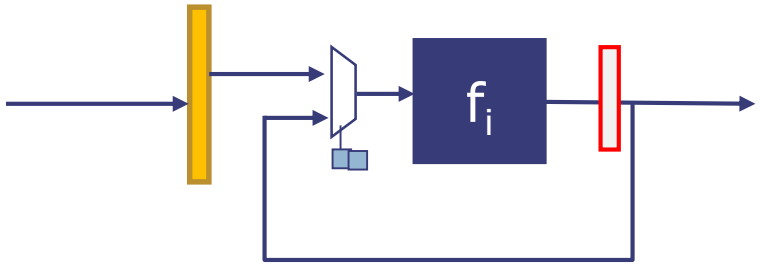
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

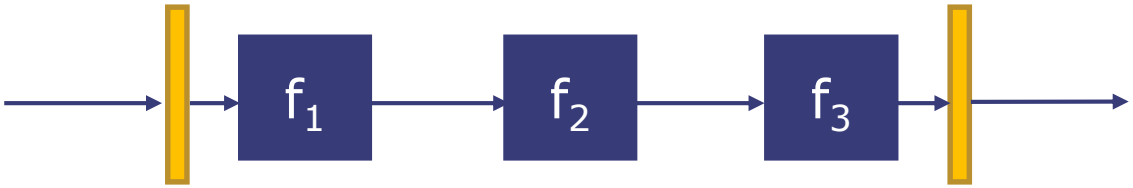


Folded reuse a block, multi-cycle (C)

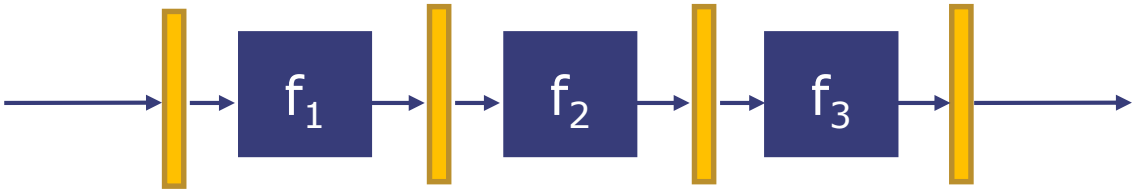


Summary: Design Alternatives

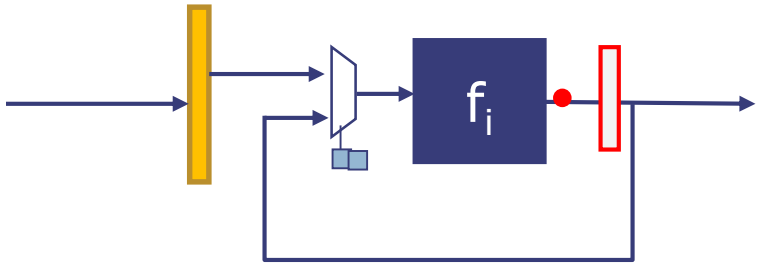
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

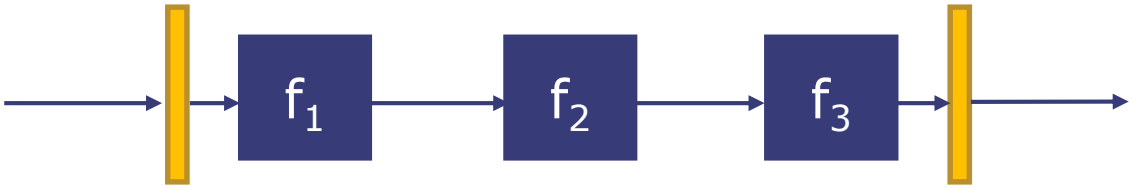


Folded reuse a block, multi-cycle (C)

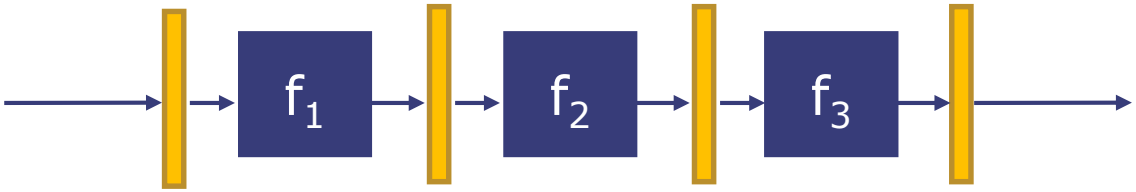


Summary: Design Alternatives

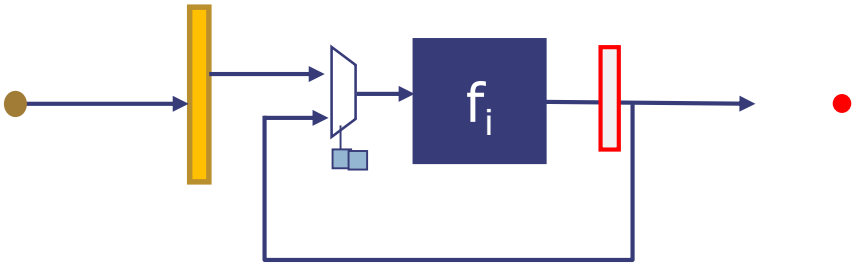
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

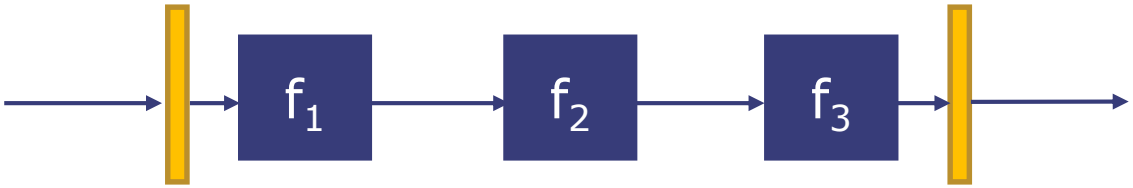


Folded reuse a block, multi-cycle (C)

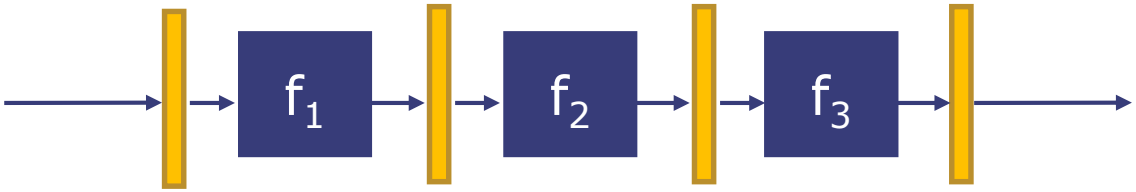


Summary: Design Alternatives

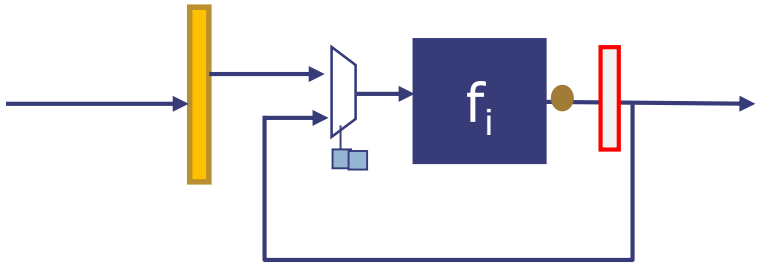
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)

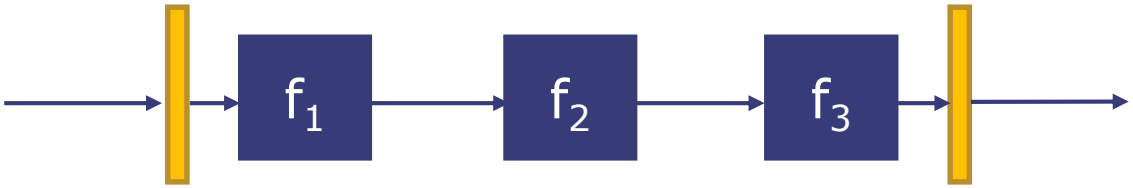


Folded reuse a block, multi-cycle (C)

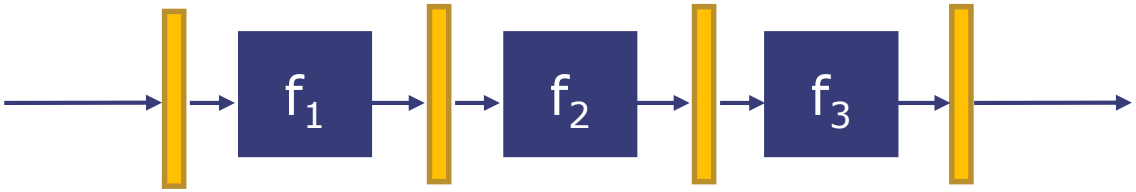


Summary: Design Alternatives

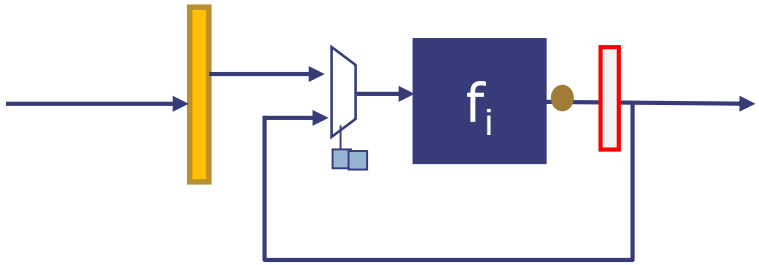
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock?

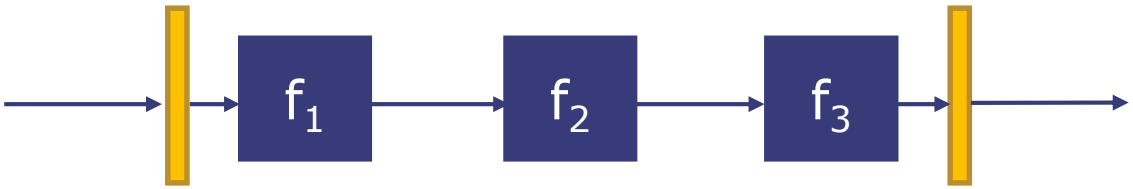
Latency?

Area?

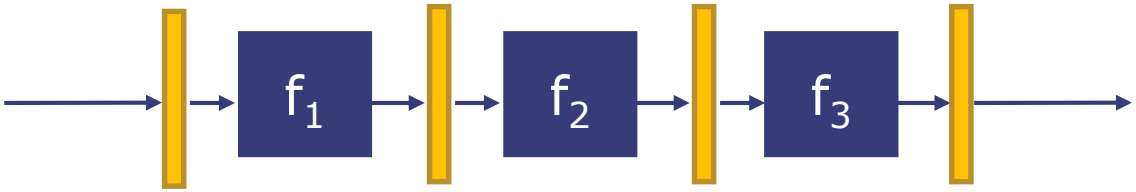
Throughput?

Summary: Design Alternatives

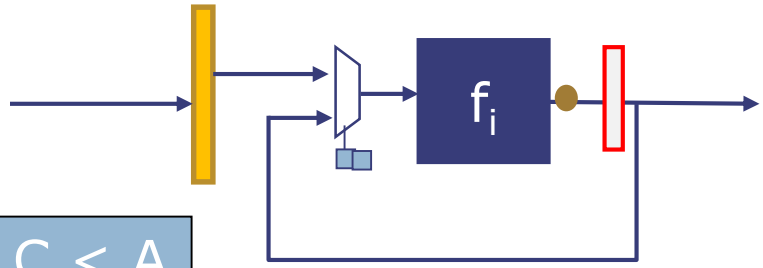
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

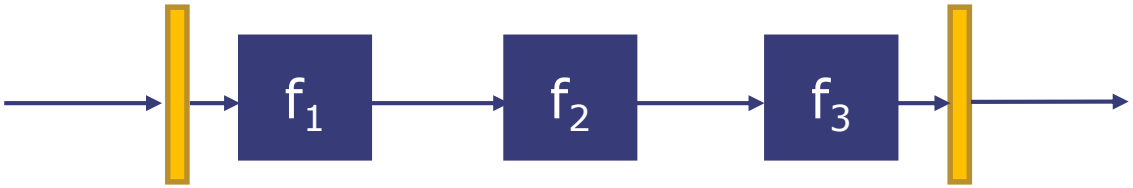
Latency?

Area?

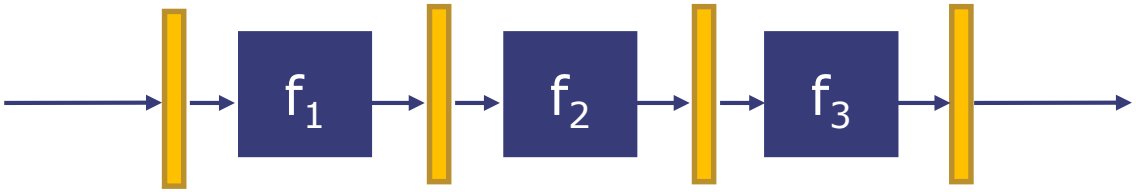
Throughput?

Summary: Design Alternatives

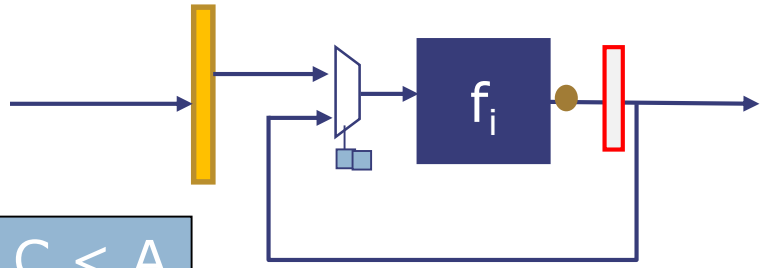
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

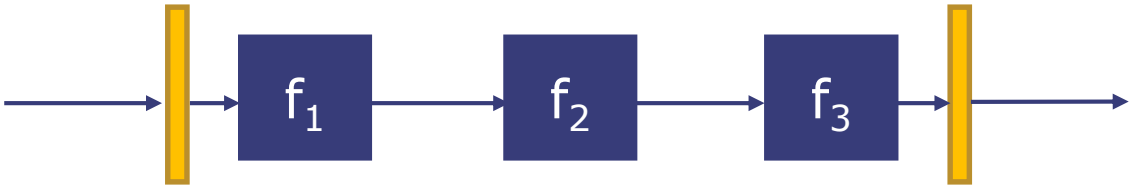
Area: C < A < B

Latency?

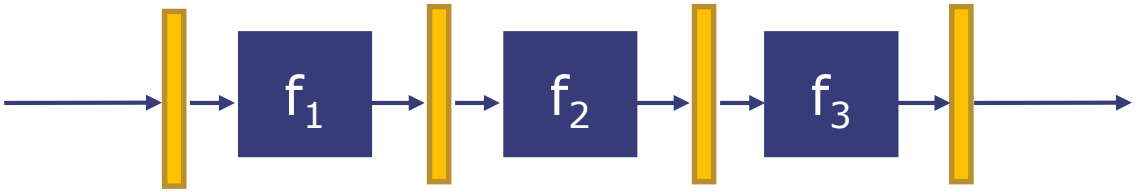
Throughput?

Summary: Design Alternatives

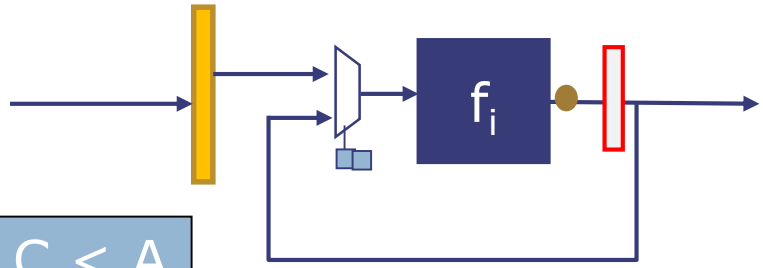
Several combinational modules in one pipeline stage (A)



One module per pipeline stage (B)



Folded reuse a block, multi-cycle (C)



Clock: B ≈ C < A

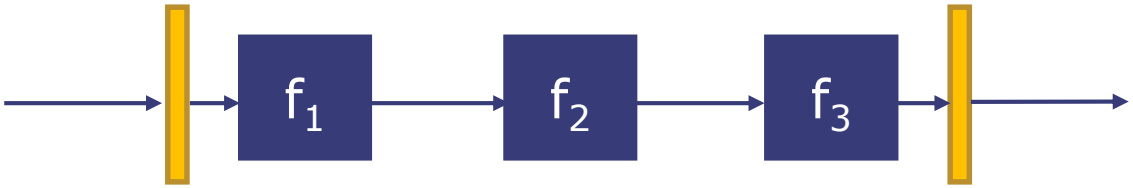
Latency: A < B < C

Area: C < A < B

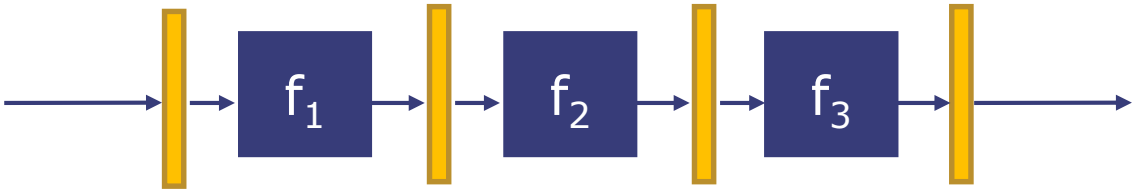
Throughput?

Summary: Design Alternatives

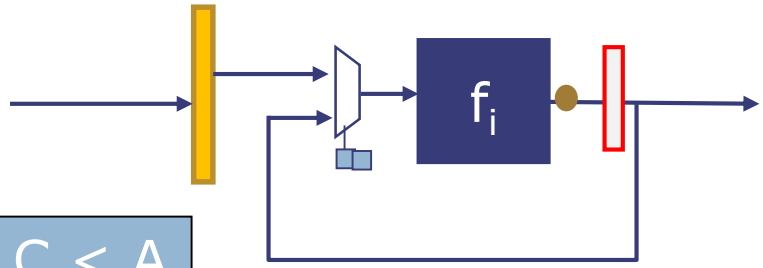
Several combinational modules in one pipeline stage (A)



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

Thank You!

Next Lecture:
Design Tradeoffs in Sequential Circuits