Week 10 - T1 2020

Computer Design Fundamentals

ELEC2141: Digital Circuit Design

UNSW SYDNEY

Overview

Registers

Shift registers

Computer design fundamentals

Arithmetic logic units

Datapath representation

Reading: Mano - Chapter 6, 6.1-6.6

Chapter 8, 8.1-8.5

UNSW SYDNEY

Registers

An *n-bit register* consists of n flip-flops and is capable of storing n-bits of binary information.

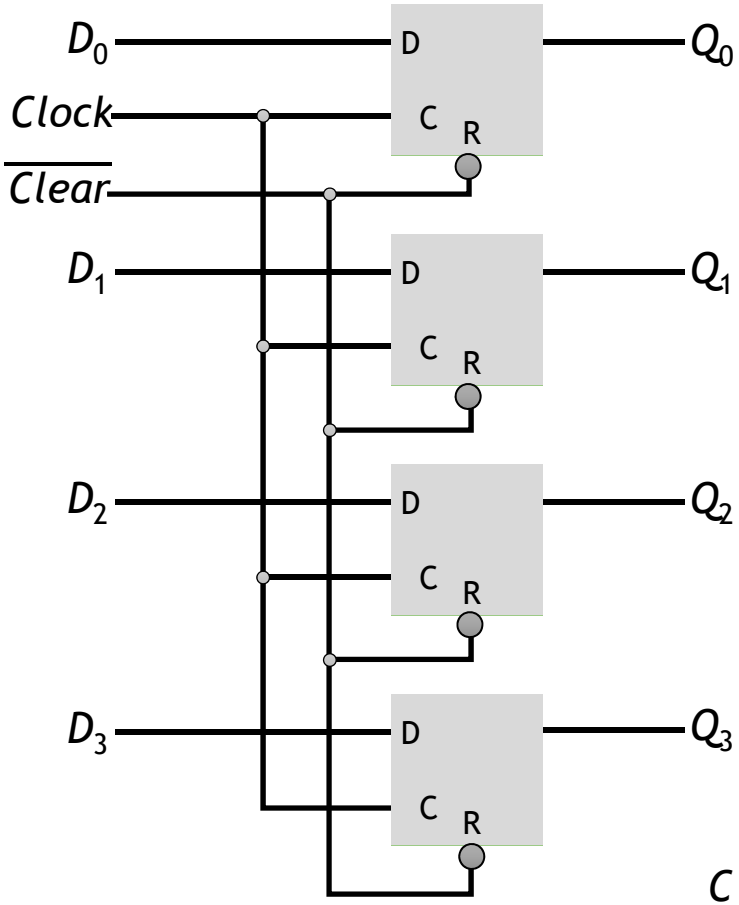
The flip-flops are connected to a common clock

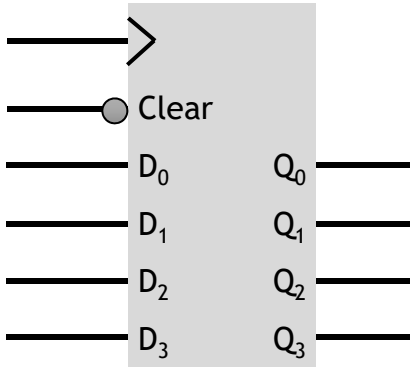
May also have a combinational circuit that implements state transitions of the flip-flops.

The flip-flops hold data and the combinational circuit determine the new or transformed data that will transfer to the flip-flops.

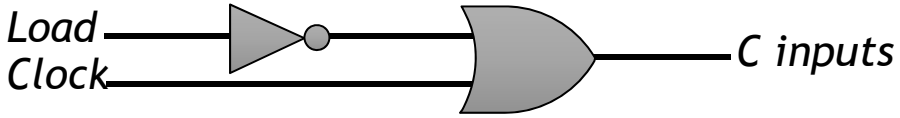
It is useful for storing and manipulating information in digital computers.

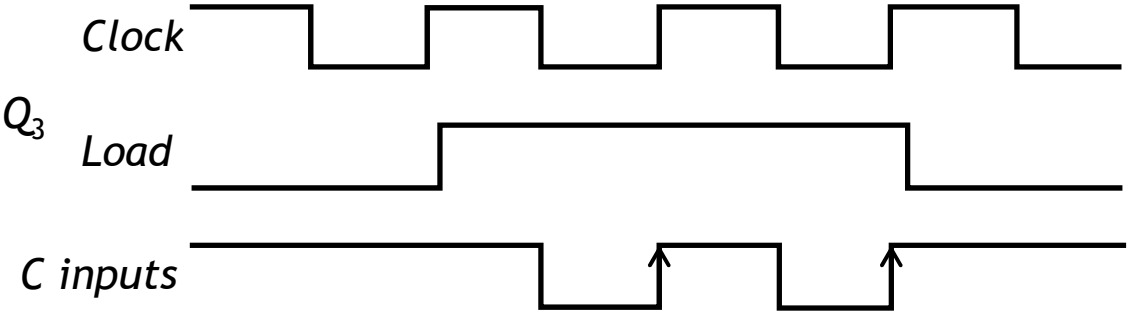
4-bit register





Clock gating





Clock skew

Inserting additional logic in the clock path introduces a delay

Clock signals arrive at different flip-flops and registers at different times depending on the logic in their clock path

This is known as *clock skew*

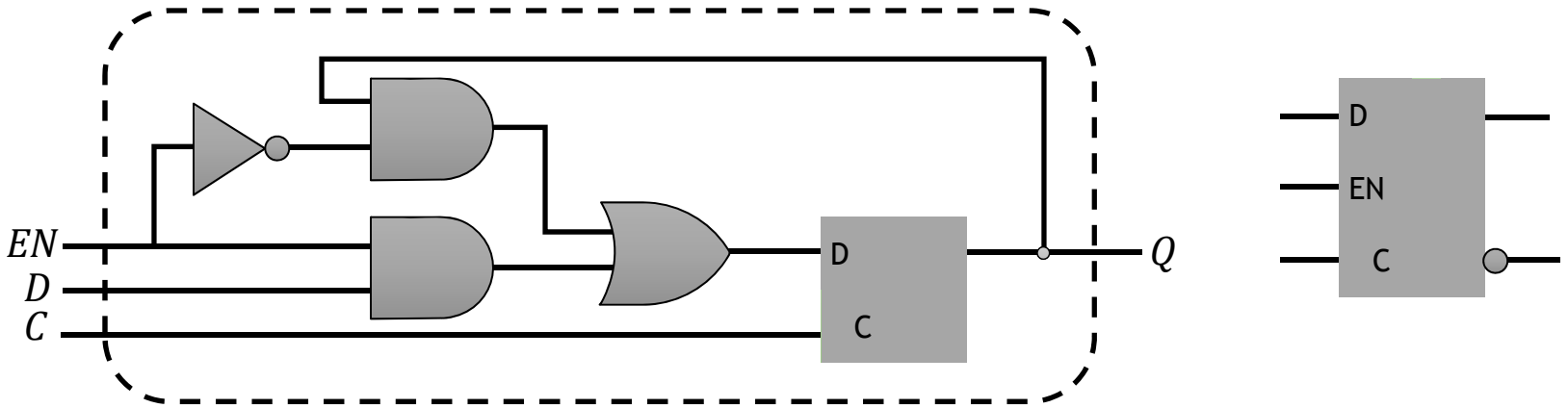
In truly synchronous system, we must insure that all clock pulses arrive simultaneously throughout the system so that all flip-flops trigger at the same time

Therefore, control operation without clock gating is advisable

D Flip-flops with enable

A D flip-flop with enable, with no clock gate, is a better option

It consists of a 2-to-1 multiplexer and a D flip-flop



Feedback connection necessary at D flip-flop does not have a ‘no change’ input condition

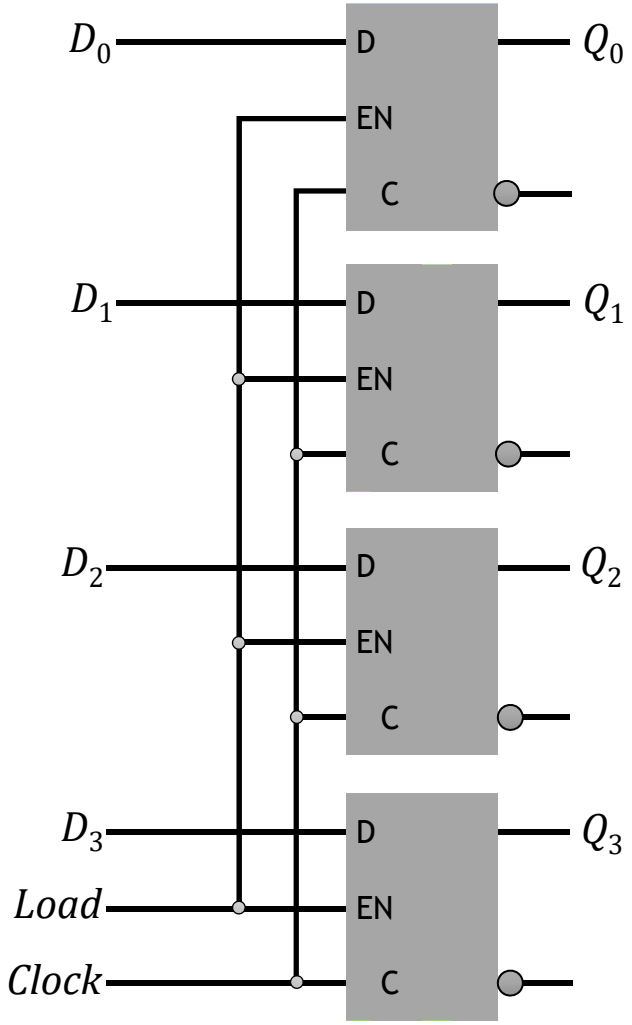
UNSW SYDNEY

4 bit Register: D flip-flop with enable

Here the D flip-flop with enable is used as the building block for the register

A common *load* signal is connected to the EN inputs of the flip-flops

When load is 1, data from the inputs is transferred into the register with the next positive clock edge; otherwise, the current value in the register remains the same



Shift register

Shift registers are capable of moving information upon the occurrence of a clock-signal either uni-directionally or bi-directionally

Information can be entered into registers either in

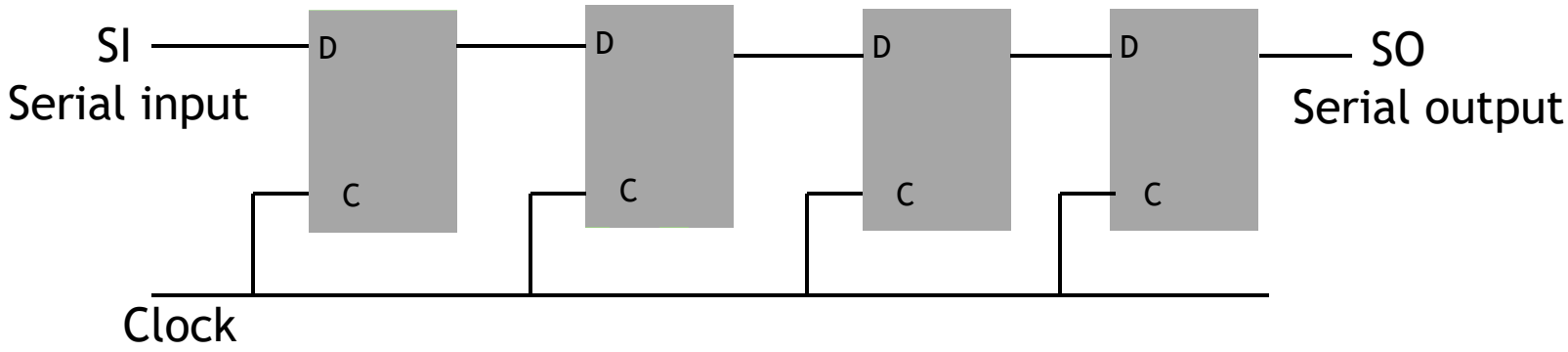
- Parallel: All 0/1 bits are handled simultaneously; requires as many lines as symbols being transferred
- Serial: Involves the bit-by-bit transfer of information in a time sequence

Four possible ways registers can transfer information:

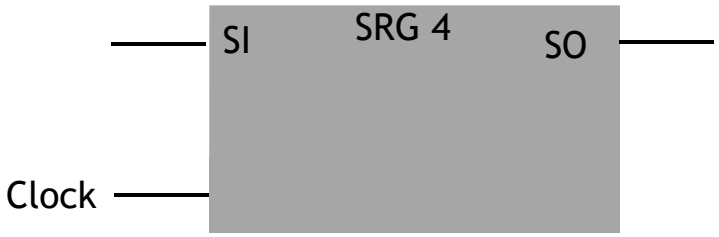
- Serial-in/serial-out
- Serial-in/parallel-out
- Parallel-in/parallel-out
- Parallel-in/serial-out

Shift register

Simplest possible shift register is a serial in – serial out shift register



4-bit Shift register



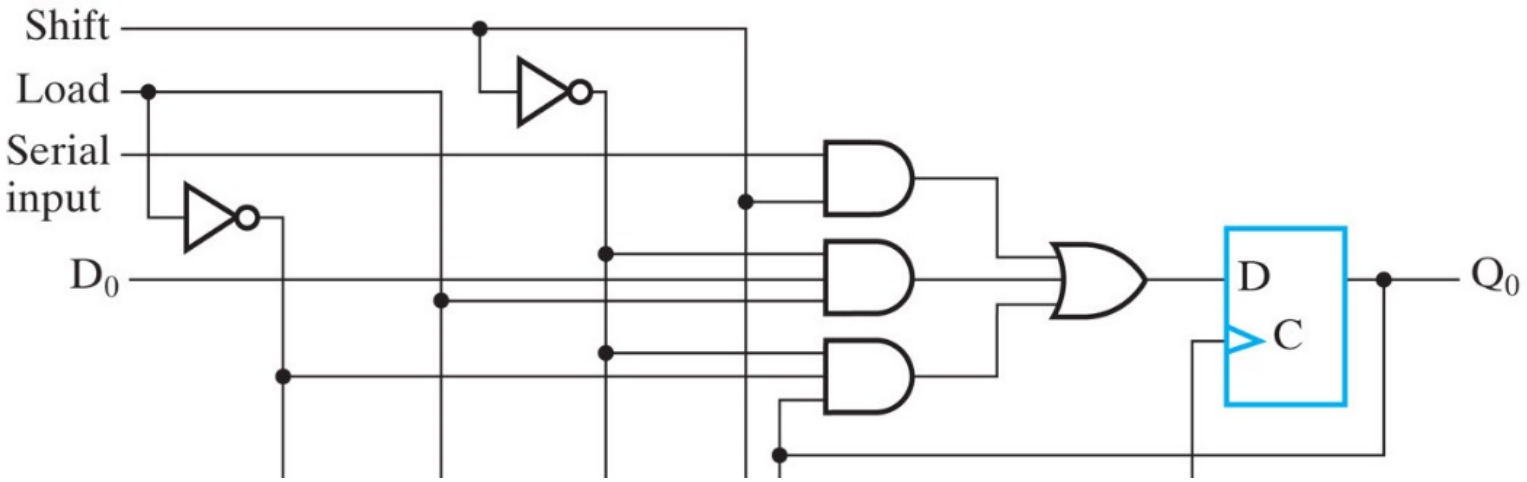
Bidirectional shift register with parallel loading provides versatile operation

UNSW SYDNEY

Shift register with parallel load

If all flip-flop outputs are accessible, output bits can be in parallel

An *n*-bit register can be created by interconnecting *n* of these units

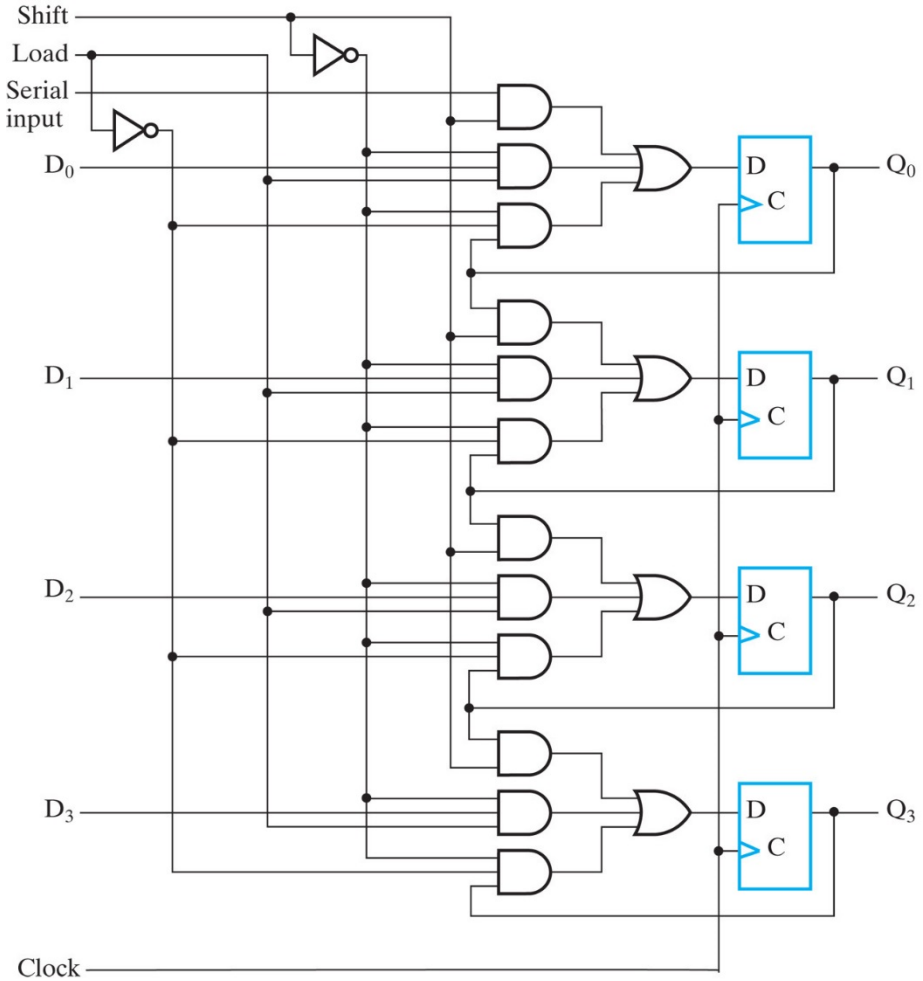


Shift	Load	Register Operation
0	0	No change (Hold)
0	1	Parallel load
1	x	Serial input

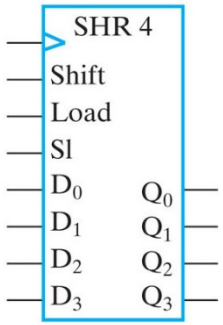
Shift register with parallel load

Adding a parallel load, allows parallel to serial shifting and vice-versa

Shift registers are used to interface digital system far from each other



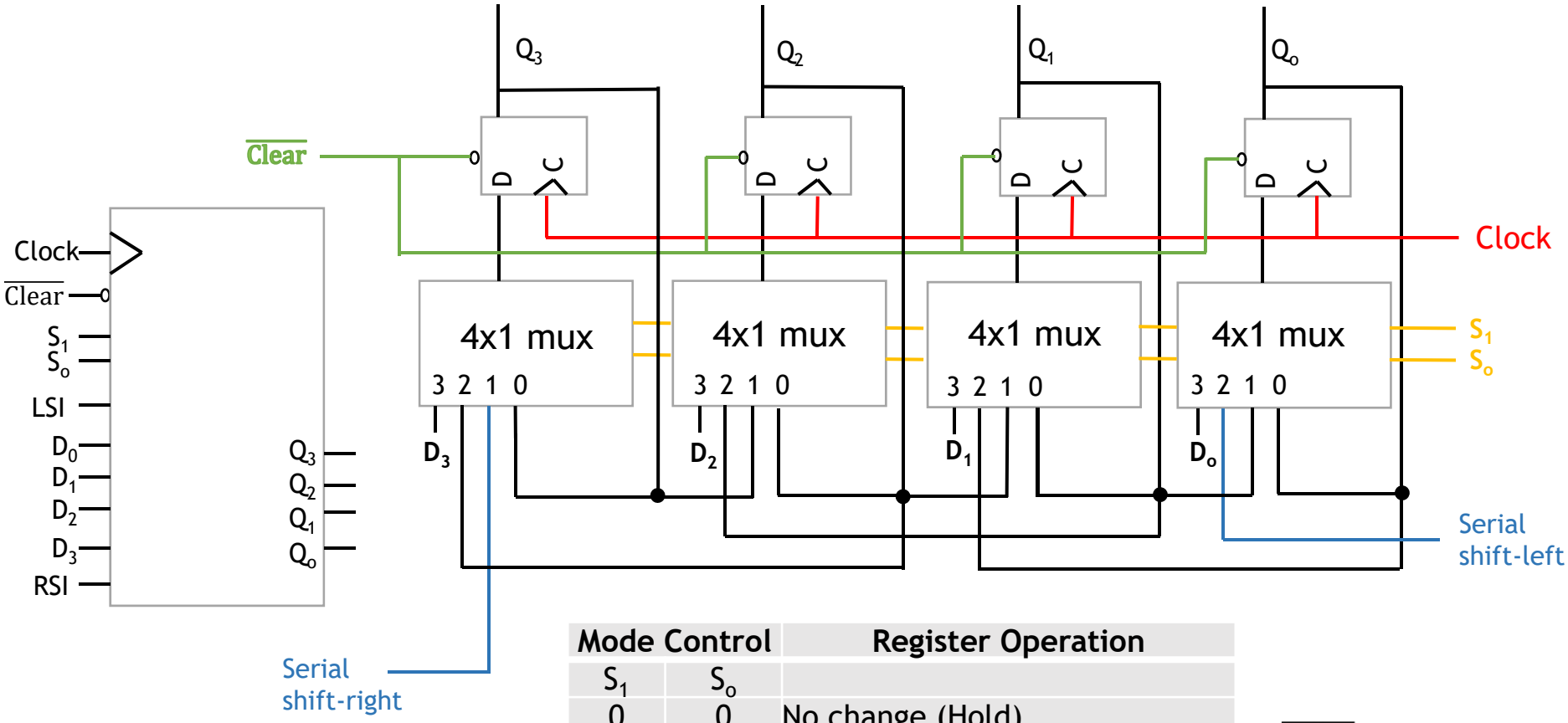
Shift	Load	Register Operation
0	0	No change (Hold)
0	1	Parallel load
1	x	Shift left (Q_o to Q_3)



UNSW SYDNEY

Bidirectional shift register

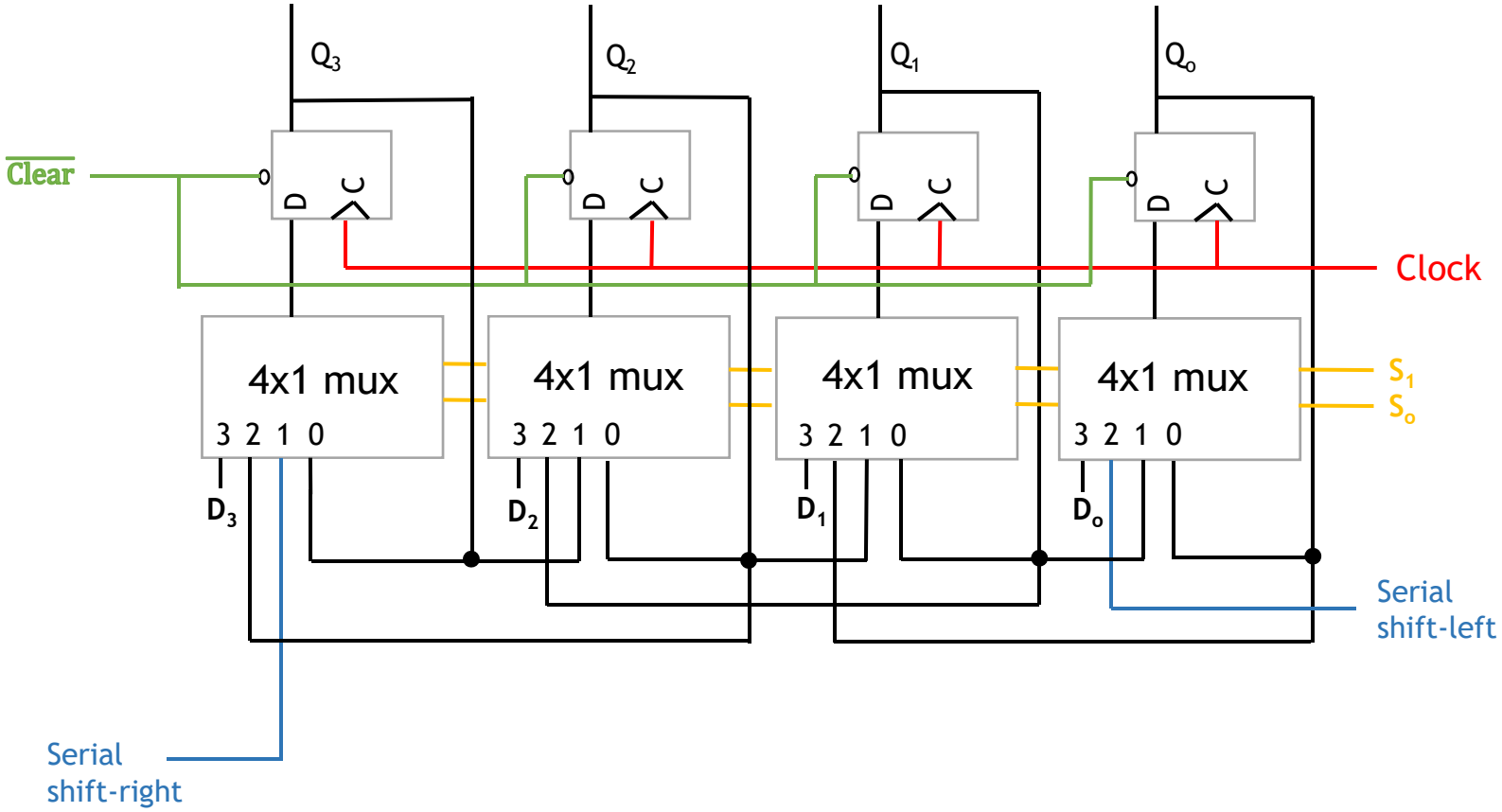
A bidirectional shift register is capable of left and right serial shifts, parallel loading, and holding data



Mode Control		Register Operation
S_1	S_o	
0	0	No change (Hold)
0	1	Shift right (RSI)
1	0	Serial Shift left (LSI)
1	1	Parallel load

UNSW SYDNEY

Bidirectional shift register in Verilog



Mode Control		Register Operation
S_1	S_o	
0	0	No change (Hold)
0	1	Shift right (RSI)
1	0	Serial Shift left (LSI)
1	1	Parallel load

UNSW SYDNEY

Verilog HDL for shift register

```
// Behavioral description of a 4-bit universal shift register

module  Shift_Register_4_beh(
  output reg    [3:0]  Q;              // Register output
  input         [3:0]  D;              //Parallel input
  input         [1:0]  S;              //Select inputs
  input         LSI, RSI,              //Serial inputs
                CLK, Clear_b );        //Clock and Clear
always @ (posedge CLK, negedge Clear_b)
   if (Clear_b == 0)   Q <= 4'b0000;
   else
     case (S)
        2'b00:  Q <= Q;                // no change
        2'b01:  Q <= {RSI, Q[3:1]};    // Shift right
        2'b10:  Q <= {Q[2:0], LSI};    // Shift left
        2'b11:  Q <= D;                //Parallel load of input
     endcase

endmodule
```

HDL for the top level module

```
// Structural description of a 4-bit universal shift register

module  Shift_Register_4_str(
  output     [3:0]  Q;            // Register output
  input      [3:0]  D;            //Parallel input
  input      [1:0]  S;            //Select inputs
  input      LSI, RSI,            //Serial inputs
             CLK, Clear_b );      //Clock and Clear

// instantiate the four stages

stage ST0 (Q[0], Q[1], LSI, D[0], Q[0], S, CLK, Clear_b);
stage ST1 (Q[1], Q[2], Q[0], D[1], Q[1], S, CLK, Clear_b);
stage ST2 (Q[2], Q[3], Q[1], D[2], Q[2], S, CLK, Clear_b);
stage ST3 (Q[3], RSI, Q[2], D[3], Q[3], S, CLK, Clear_b);

endmodule
```

HDL for the next top level module

```
// Declaring one stage of shift register

module  stage(i0,i1,i2,i3,Q,select, CLK, Clr_b);

     input      i0, i1, i2, i3;              // Register output
     output       Q;              //Parallel output
     input [1:0]  select;                 //Select inputs
     input       CLK,Clr_b;       //Serial inputs
     wire         mux_out;       //Clock and Clear

// instantiate mux and flip-flop

mux_4_x_1 M0 (mux_out, i0, i1, i2, i3, select);
DFF   M1 (Q,mux_out,CLK, Clr_b);

endmodule
```

Declaring the low level modules

```
// Declaring  4x1 multiplexer

module
Mux_4_x_1(mux_out,i0,i1,i2,i3,select);

  input      i0, i1, i2, i3;
  output reg       mux_out;
  input [1:0]  select;

Always @ (select, i0,i1,i2,i3)
   case (select)
       2'b00: mux_out = i0;
       2'b01: mux_out = i1;
       2'b10: mux_out = i2;
       2'b11: mux_out = i3;
  endcase

endmodule
```

```
// Declaring  DFF

module  DFF(Q,D,CLK, Clr_b);

  output reg   Q;
  input      D, CLK, Clr_b;

   always @ (posedge CLK, negedge Clr_b)
         if (Clr_b == 0)  Q<= 1'b0;
          else
          Q <= D;
endmodule
```

Counters

Counters are a specific kind of register that goes through a prescribed sequence of distinct states

Also called a pattern generator

Each stored 0/1 combination is called the state of the counter

The total number of states is called its *modulus*

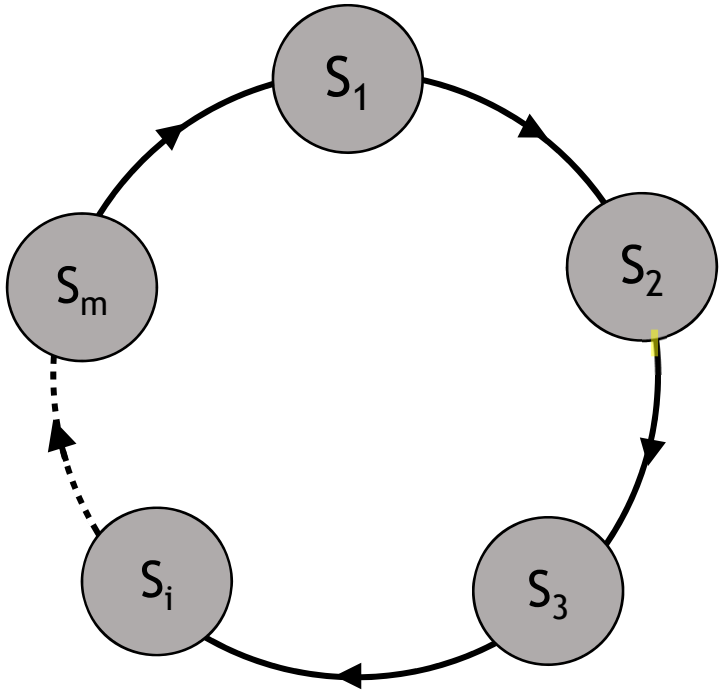
- If a counter has *m* distinct states then it is called a *mod-m* counter

Two categories: *ripple* and *synchronous* counters

State diagram of a counter

The order in which states appear is referred to as its *counting sequence* and depicted with a state diagram

S_i denotes one of the states of the counter



UNSW SYDNEY

Ripple counters

In *ripple counters*, the flip-flop output transitions cause changes in other flip-flops, i.e. the effect of a count pulse ripples through the counter

Also referred to as asynchronous counters

In *binary ripple counters*, the counting sequence corresponds to that of the binary numbers

Modulus is 2^n, where n is the number of flip-flops in the counter

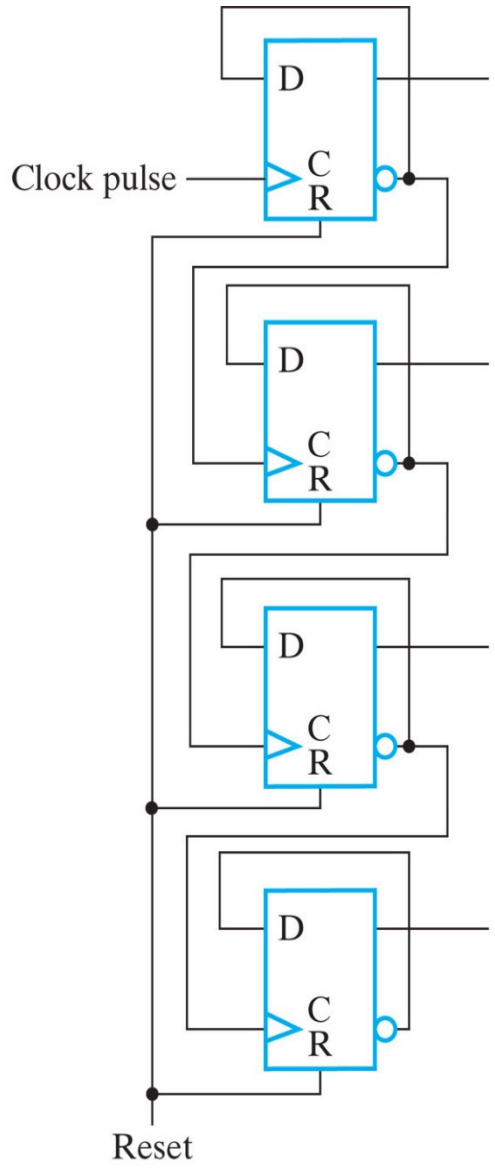
Can have a *binary up-counter*, *binary down-counter*

4-bit binary ripple counter

The least significant flip-flop receives the incoming clock pulse

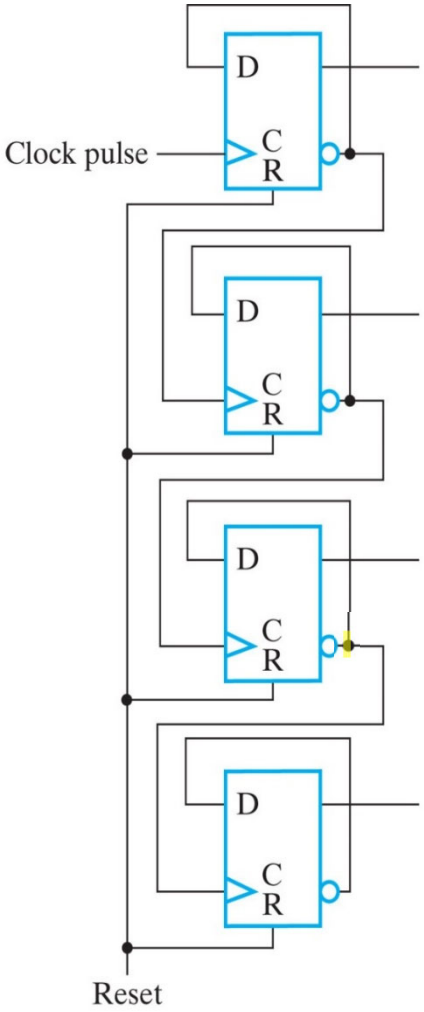
The complemented output of each flip-flop is connected to the clock of the next most significant flip-flop

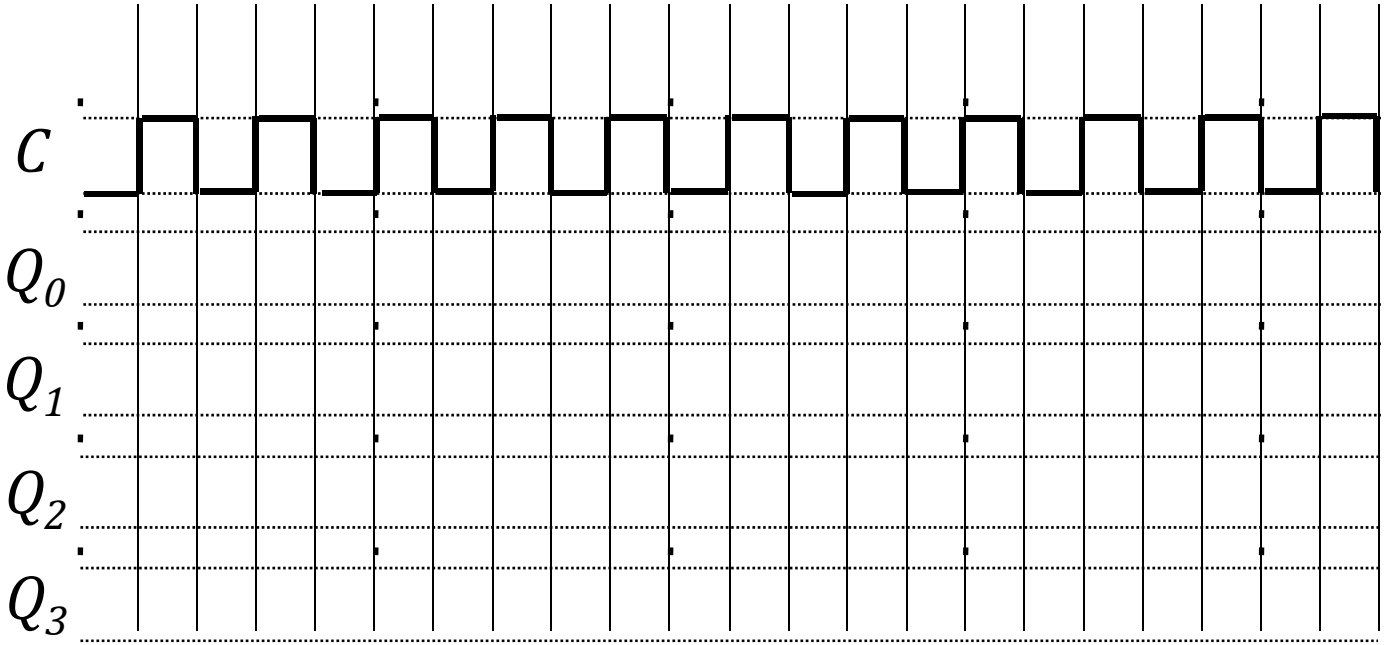
Each positive transition of the clock pulse causes each flip-flop to toggle



UNSW SYDNEY

4-bit binary ripple counter

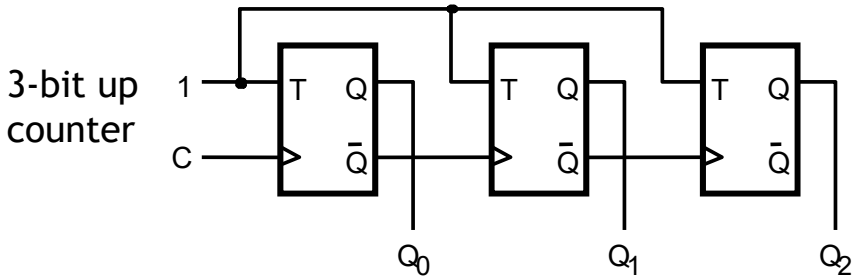


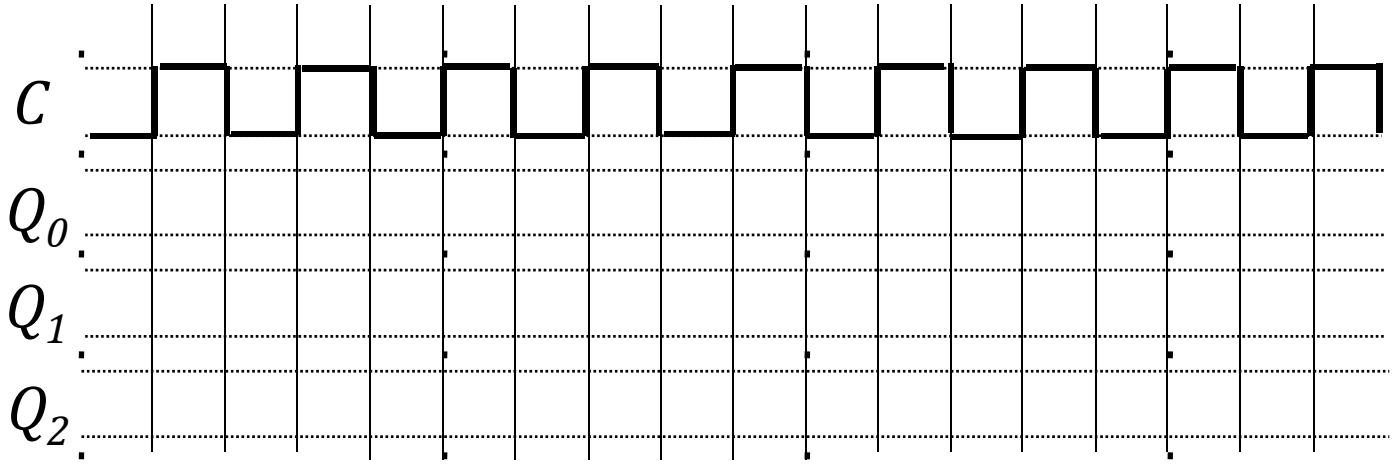


UNSW SYDNEY

Binary ripple counter

Counters built with T flip-flops are the simplest as the toggle feature is naturally suited to the counting operation

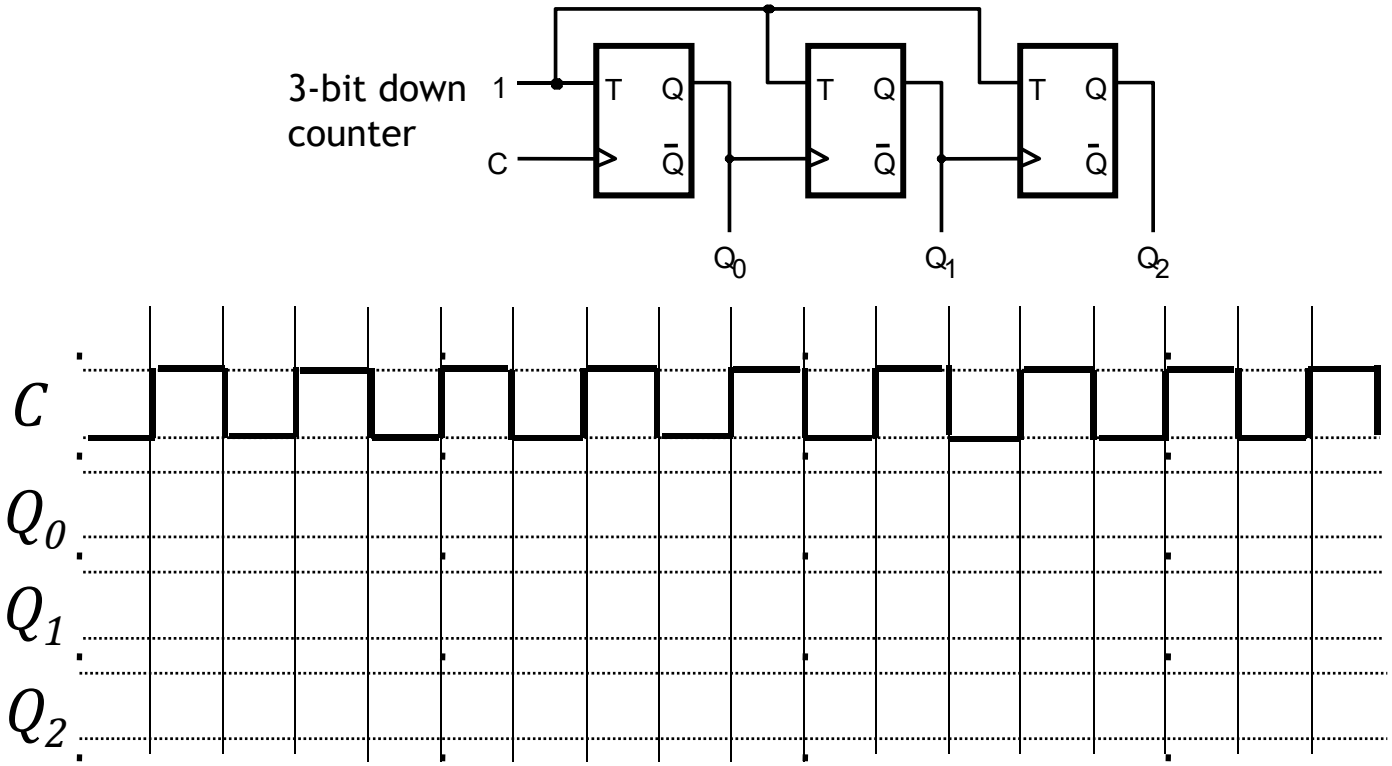




UNSW SYDNEY

Binary ripple counter

Counters built with T flip-flops are the simplest as the toggle feature is naturally suited to the counting operation



UNSW SYDNEY

Delays in ripple counters

There is a propagation delay between the input and output of a flip-flop

Rippling behavior affects the overall time delay between the occurrence of a count pulse and when the stabilized count appears at the output

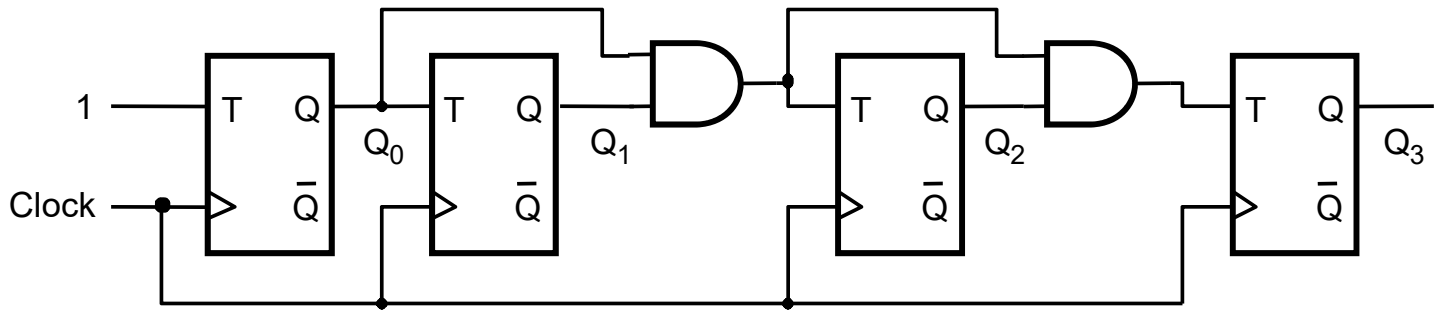
When going from $111 \cdots 111$ to $000 \cdots 000$, toggle signals must propagate through the entire length of the counter

For *n*-stage binary ripple counter, the worst case time is $n \cdot t_{pd}$, where t_{pd} is the propagation delay time associated with each flip-flop

Synchronous binary counters

In *synchronous counters*, all flip-flops change simultaneously after the propagation delay associated with a single flip-flop

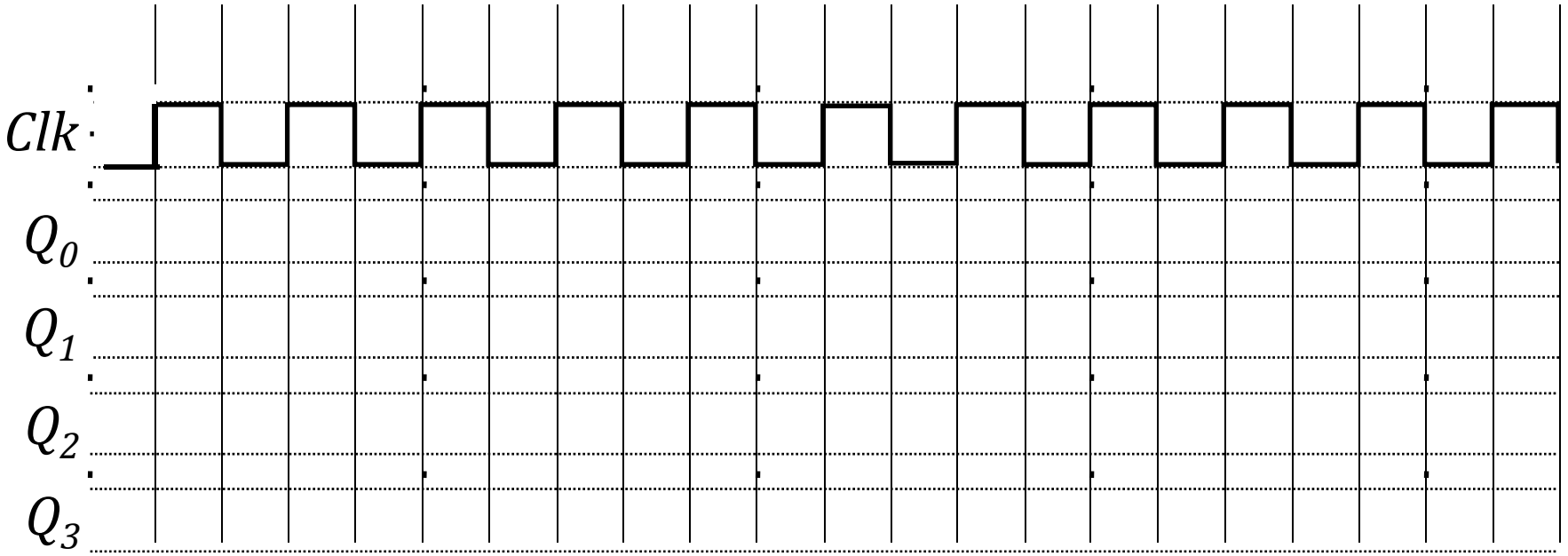
Count pulses are applied directly to the control inputs, C, of all the clocked flip-flops

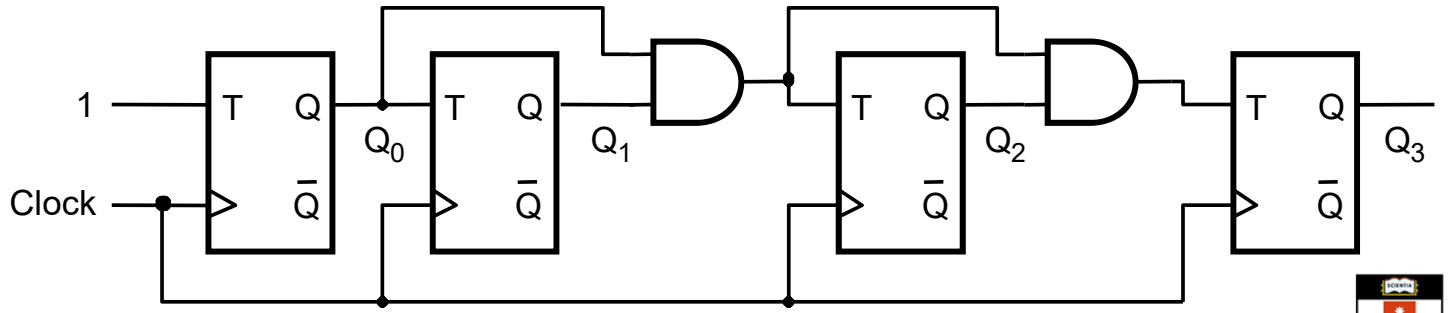


4-bit synchronous up counter with T flip-flops

UNSW SYDNEY

Synchronous binary counters



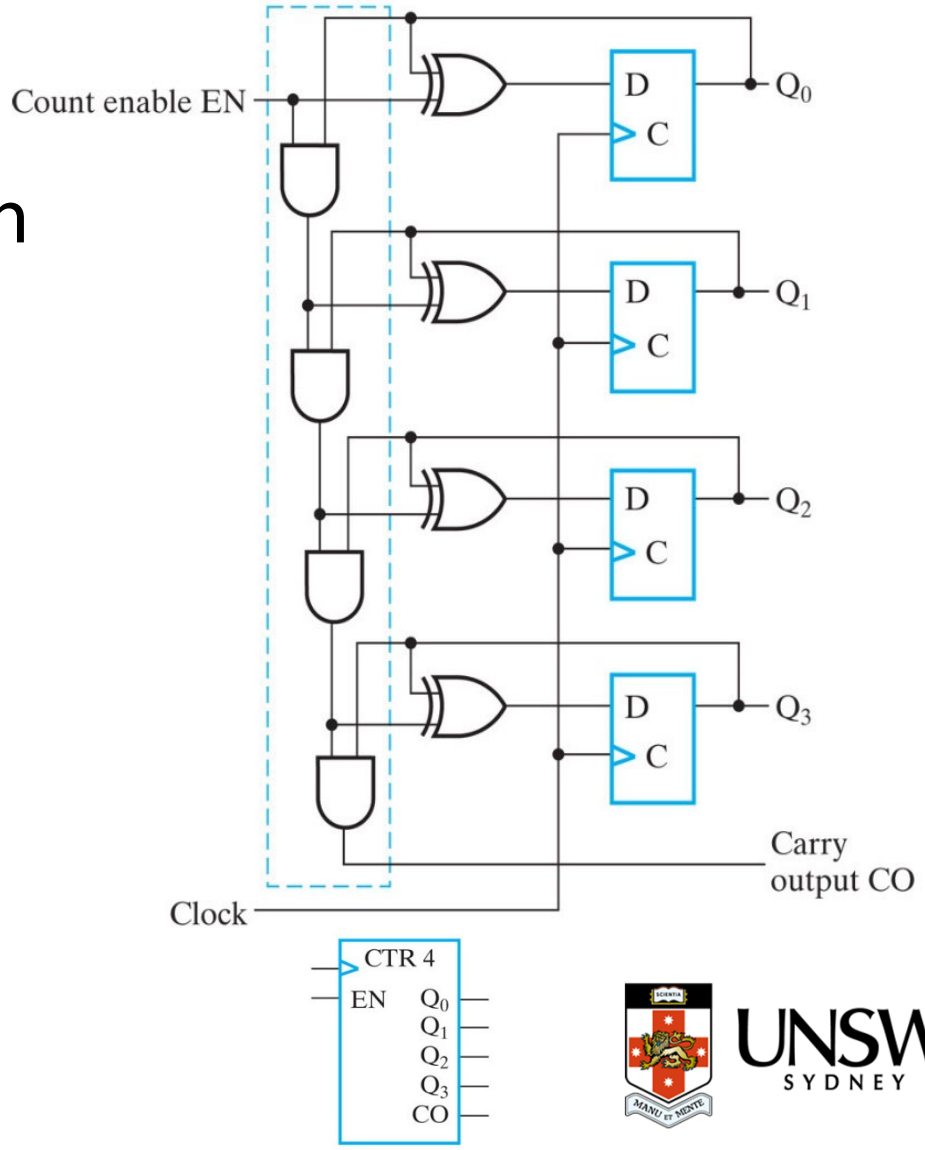


Synchronous binary counters

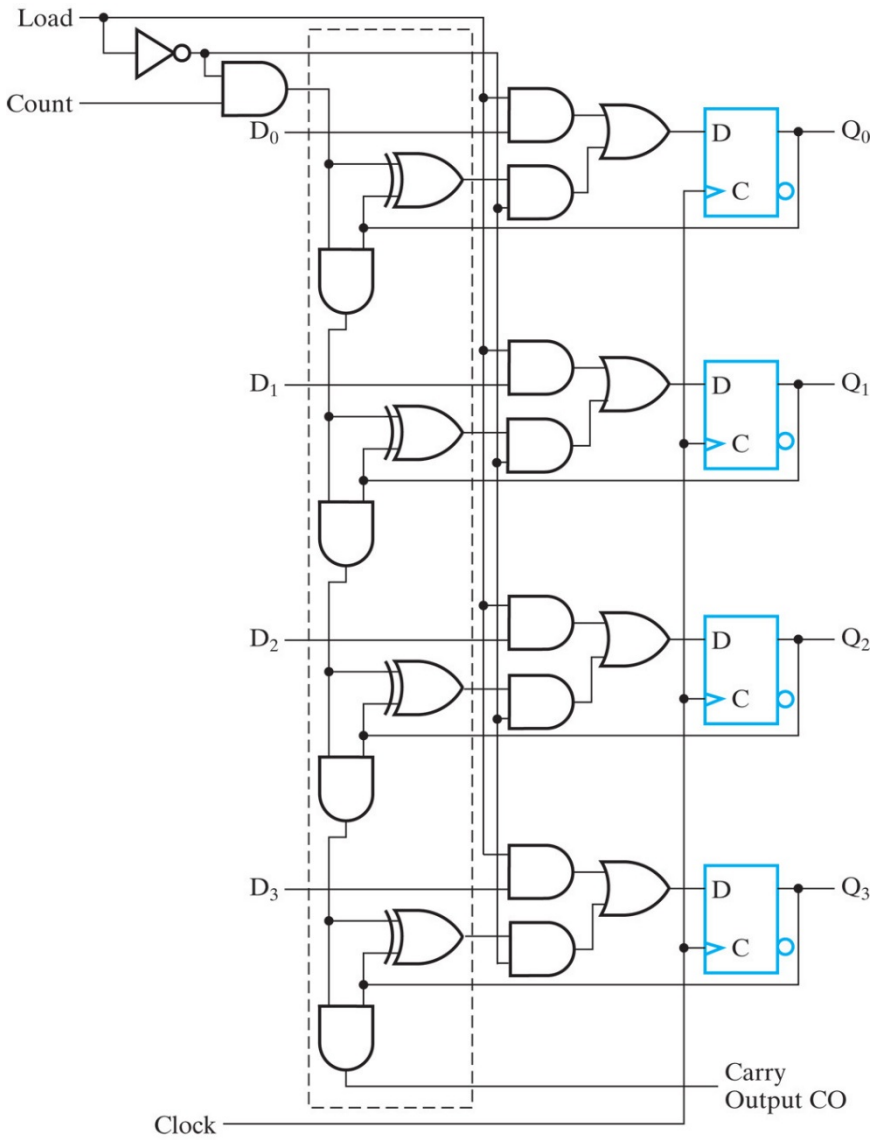
A *synchronous counter* can also be made by combining an adder with D flip-flop

All the flip-flops trigger on the positive edge of the clock (can also use negative edge)

The carry output CO is used to extend the counter to more stages



Binary counter with parallel load



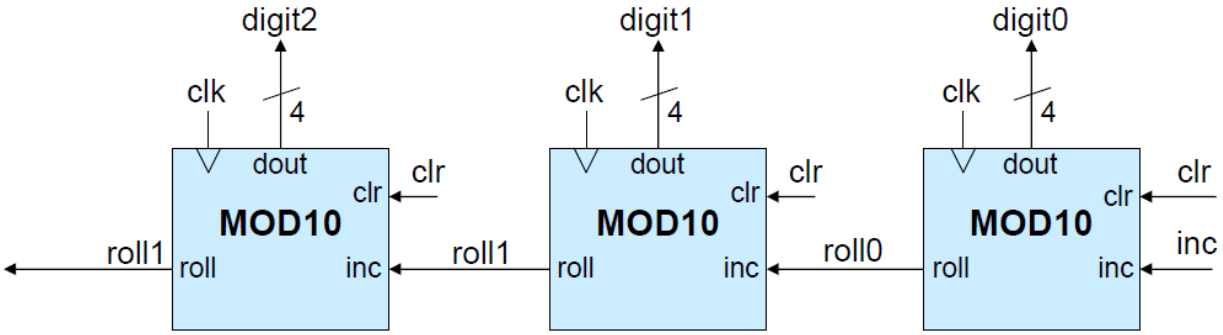
Parallel load added to input an initial binary number into counter prior to count operation

Load	Count	Operation
1	0	Parallel load
0	1	Count enabled
0	0	Hold

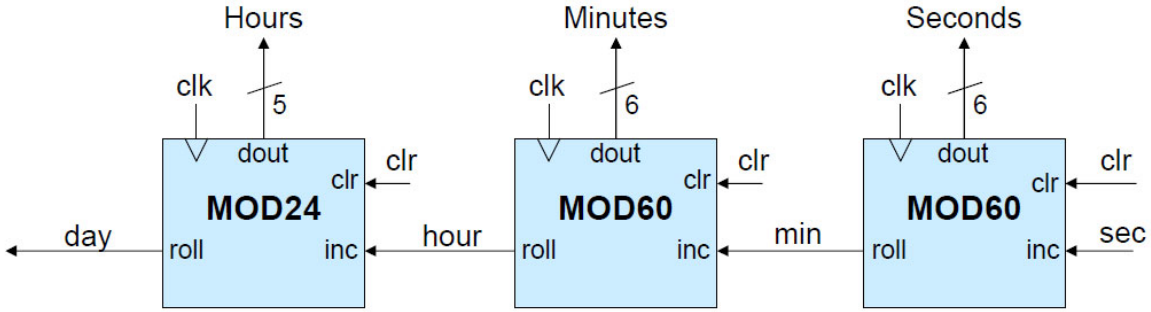
UNSW SYDNEY

Cascaded counters

Larger counters can be built by combining smaller counters together



3 digit BCD counter



Hour:minute:second clock

UNSW SYDNEY

Computer design fundamentals

A simple computer architecture can be divided into

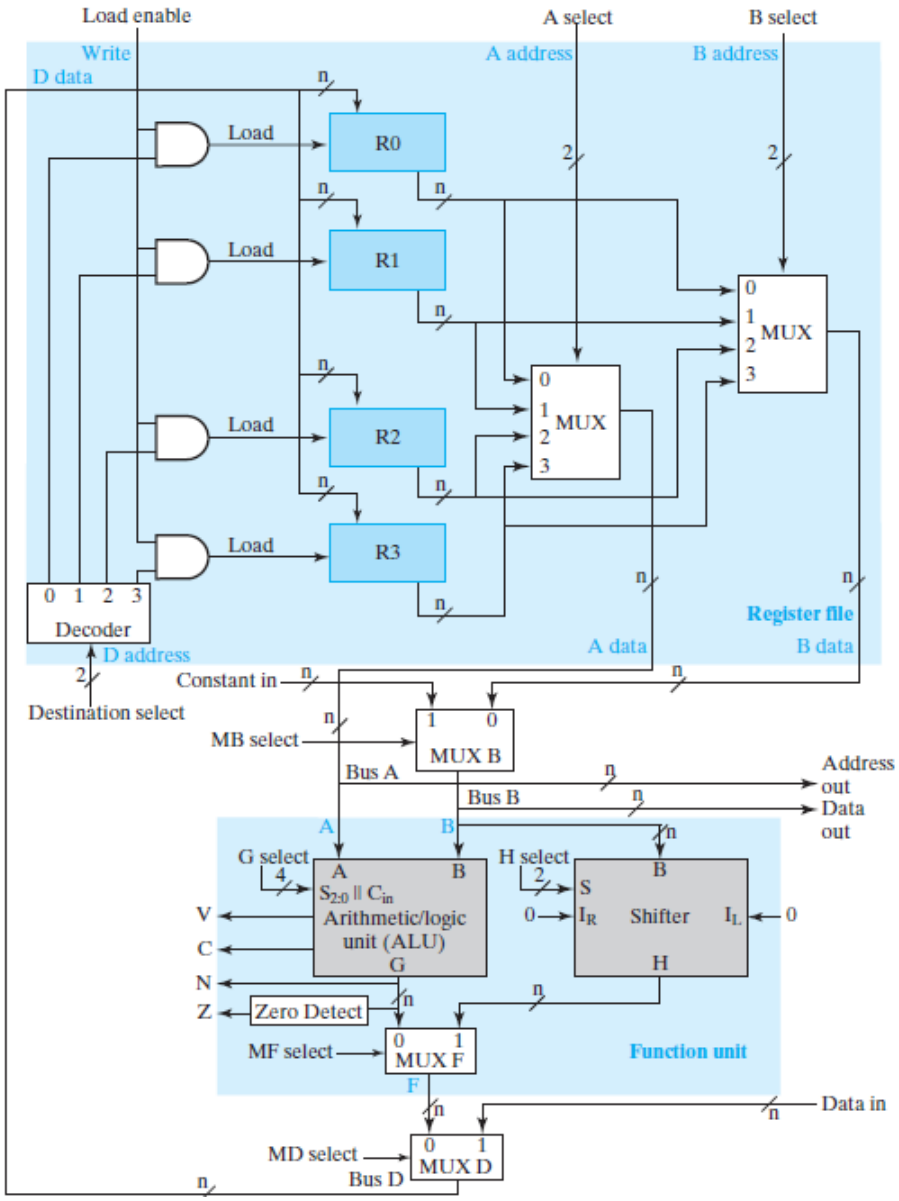
1. *Datapath*, which consists of:
   - A set of registers
   - The micro-operations performed on data stored in the registers
   - The control interface
2. *Control Unit*, which provides signals that control micro-operations performed in the datapath and other components

Micro-operations are elementary operations performed on data stored in registers (arithmetic/logic/shift/transfer)

In computer systems, a shared *arithmetic/logic unit (ALU)* is used to perform micro-operations

The contents of registers are transferred to the ALU, which performs the operation and then transfers them to the destination

Datapath



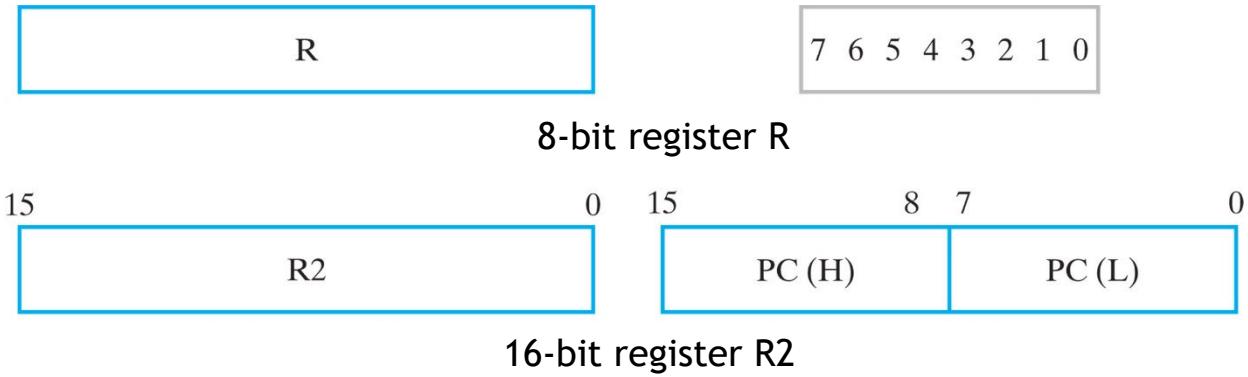
UNSW SYDNEY

Datapath

Registers are denoted by uppercase letters (followed by numbers)

AR – address register, *PC* – program counter, *R2* – register 2

The individual flip-flops in a *n* bit register are numbered from 0 to *n*-1 with 0 the LSB



Datapath

Data transfer from one register to another is denoted using the ← operator

$$R2 \leftarrow R1$$

$R1$ is the source of transfer and $R2$ the destination

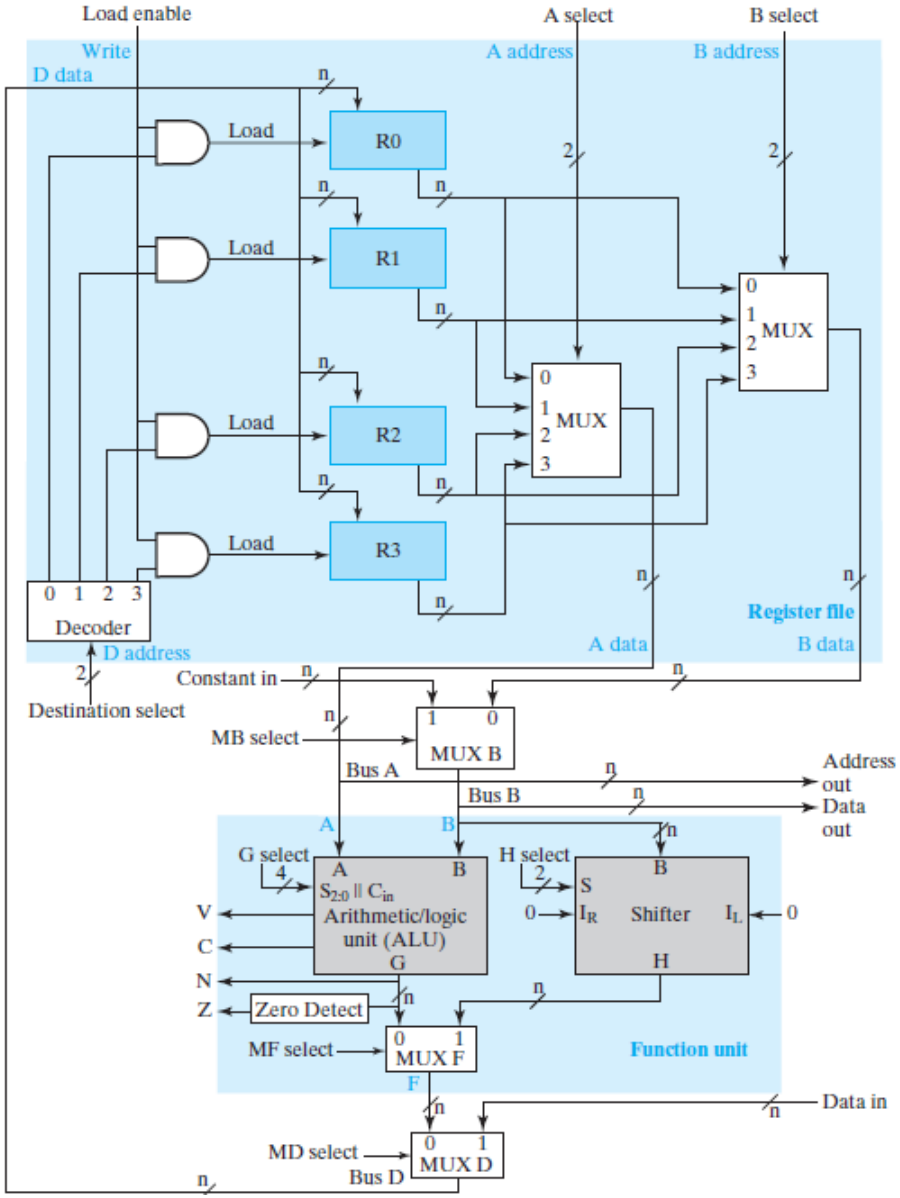
Conditional statements used when data transfer occurs only on receiving a control signal

$$K_1: R2 \leftarrow R1$$

K_1 can be any Boolean function or variable that evaluates to 0 or 1

$$K_3: R2 \leftarrow R1,\ R1 \leftarrow R2$$

Datapath



To perform *R1* ← *R2* + *R3*

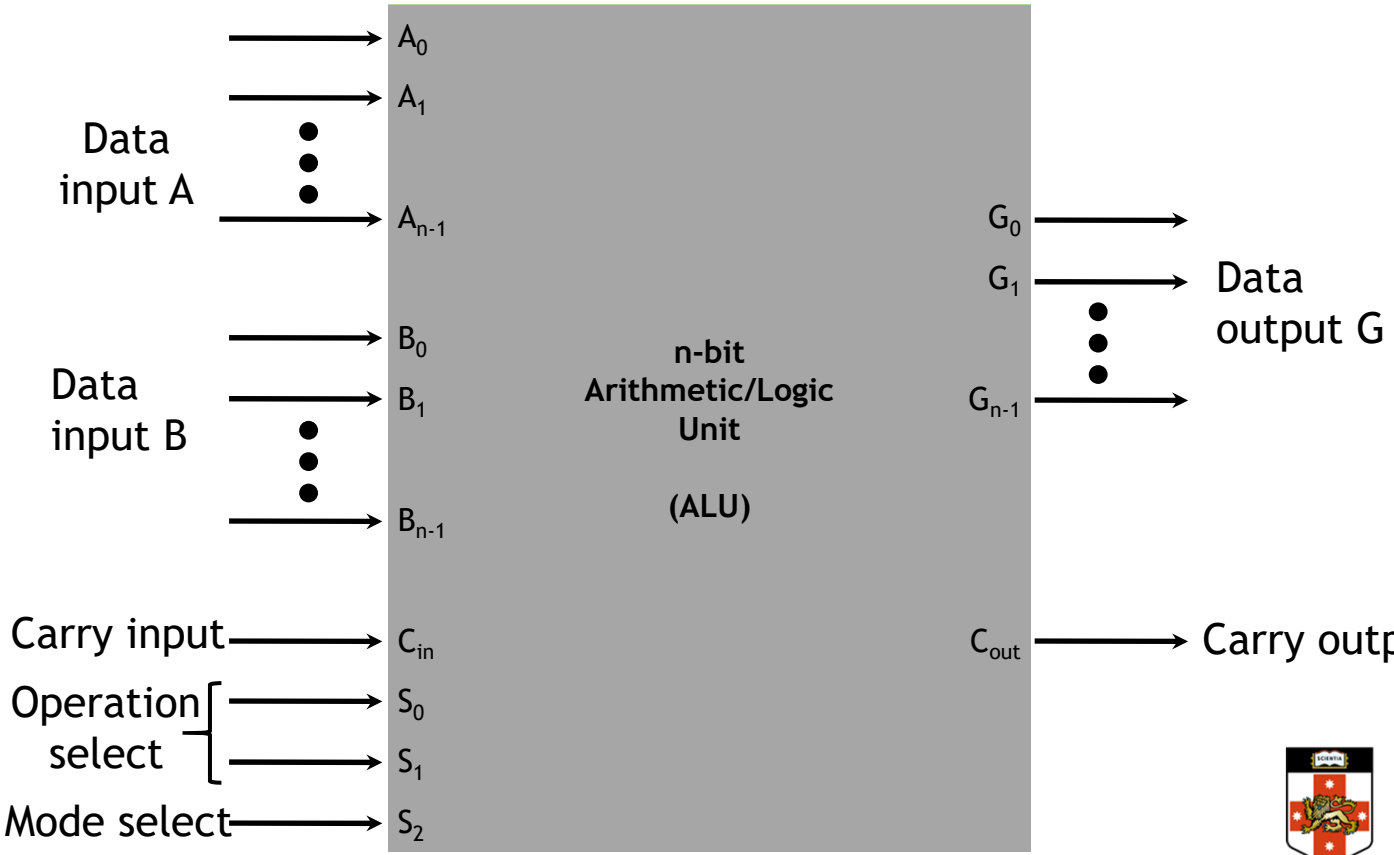
The control unit must provide the following control inputs:

1. *A select*, to place contents of *R2* onto A *data* and Bus A
2. *B select* to place contents of *R3* onto 0 input of MUX B; and *MB select* to put 0 input of MUX B onto Bus *B*
3. *G select*, to complete A+B
4. *MF select*, to place the ALU output on the MUX F output
5. *MD select*, to place the MUX F output onto Bus *D*
6. *Destination select*, to select *R1* as the destination of the data on Bus *D*
7. *Load enable*, to enable *R1* to be loaded

UNSW SYDNEY

Arithmetic/logic unit

An *arithmetic/logic unit (ALU)* is a combinational circuit that performs basic arithmetic and logic operations



UNSW SYDNEY

Arithmetic/logic unit

Performs a set of basic arithmetic and logic micro-operations

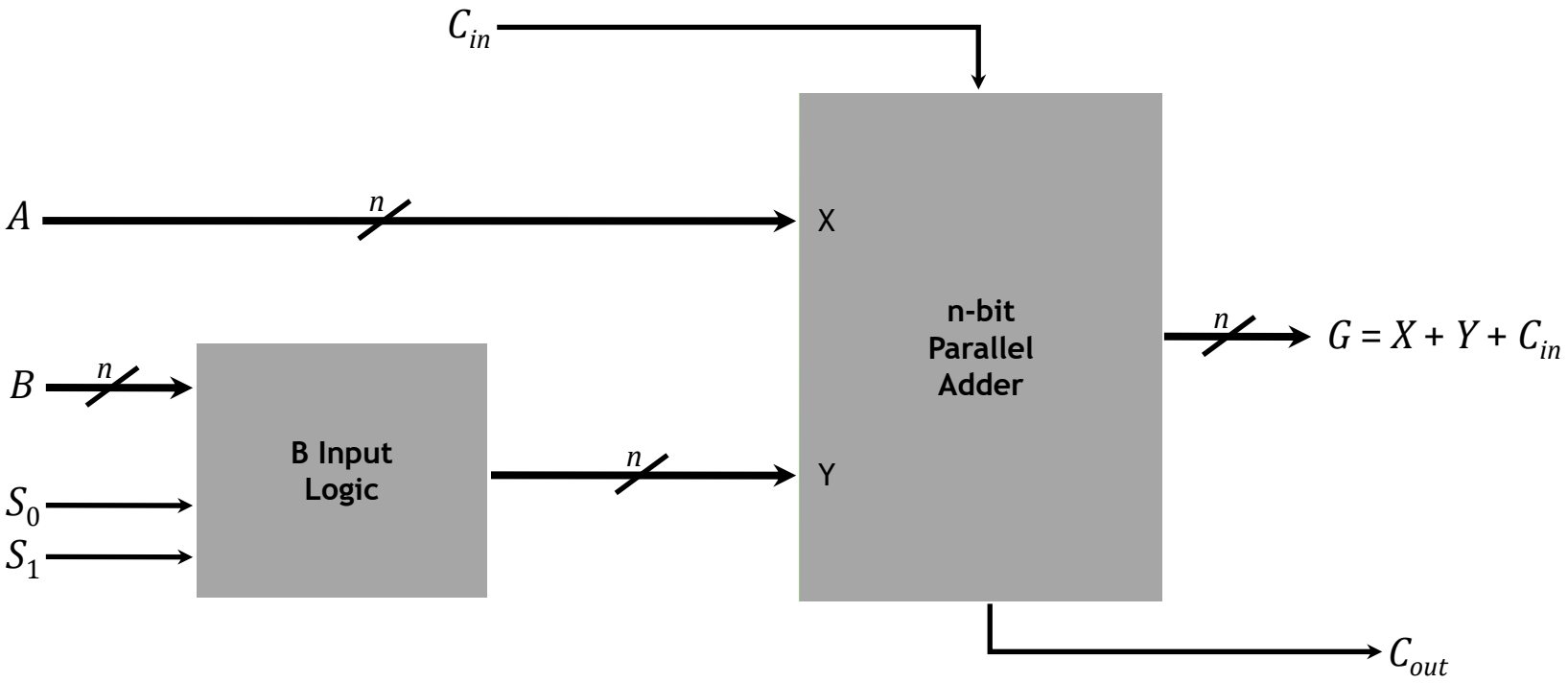
The n-bit result is output on vector G with carry-out on C_{out}

S_2 selects between arithmetic or logic operations

S_1 and S_0 select one of 4 operations within the arithmetic or logic operations

Can be extended to 8 operations by using the C_{in} (carry-in) input

Arithmetic/logic unit



UNSW SYDNEY

Arithmetic unit

The arithmetic circuit is based on the *n*-bit parallel adder

By controlling the inputs to the adder, different types of arithmetic operations can be obtained

Connect the *X* input of the adder directly to operand *A*

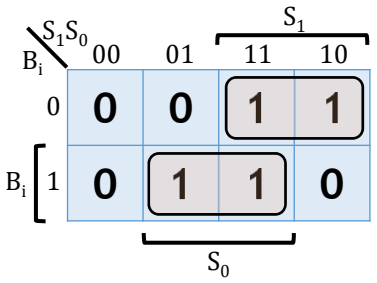
Make the input *Y* some function of operand *B*

Select		Input	$G = A + Y + C_{in}$	
S_1	S_0	Y	$C_{in} = 0$	$C_{in} = 1$
0	0	All 0s	$G = A$ (transfer)	$G = A + 1$ (increment)
0	1	B	$G = A + B$ (add)	$G = A + B + 1$
1	0	$\overline{B}$	$G = A + \overline{B}$	$G = A + \overline{B} + 1$ (subtract)
1	1	All 1s	$G = A - 1$ (decrement)	$G = A$ (transfer)

Arithmetic unit

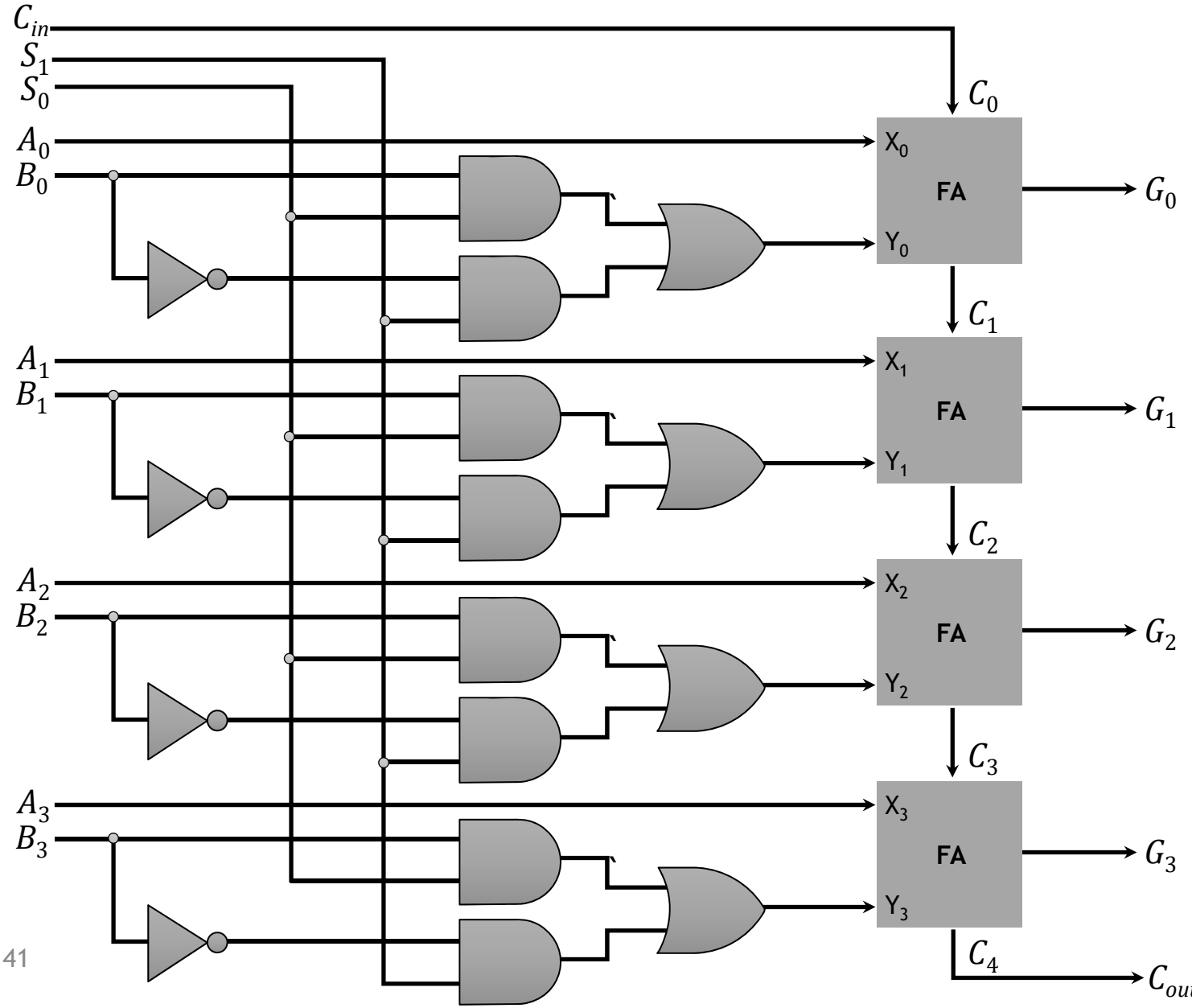
Use a truth table and k-maps to implement the select logic on input *B*

Inputs			Output	Function
S_1	S_0	B_i	Y_i	
0	0	0	0	$Y_i = 0$
0	0	1	0	
0	1	0	0	$Y_i = B_i$
0	1	1	1	
1	0	0	1	$Y_i = \overline{B}_i$
1	0	1	0	
1	1	0	1	$Y_i = 1$
1	1	1	1	



$$Y_i = B_i S_0 + \overline{B}_i S_1$$

Arithmetic unit



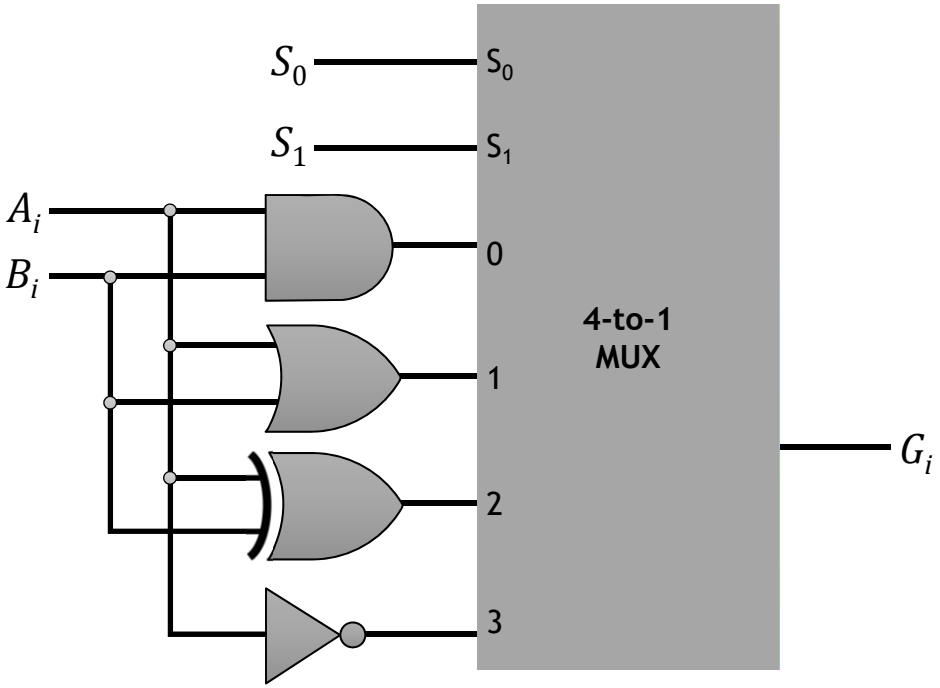
UNSW SYDNEY

Logic unit

Performs bitwise logical operations on the bits of the operands

There are four commonly used logic operations- AND, OR, XOR, and NOT

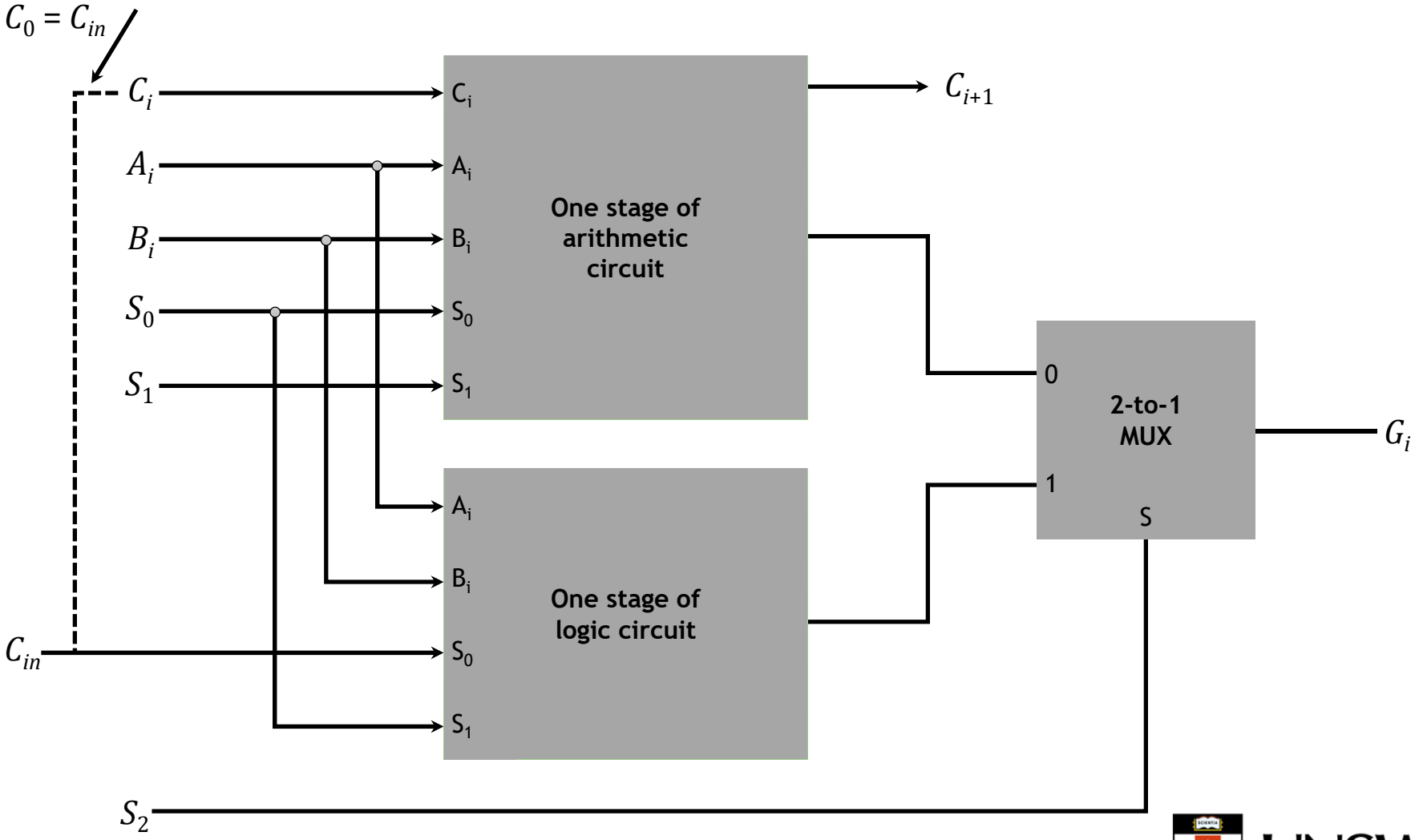
One stage logic unit: (as many as n-bit are required)



S_1	S_0	Output	Operation
0	0	$G = A \,\&\, B$	AND
0	1	$G = A \mid B$	OR
1	0	$G = A \oplus B$	XOR
1	1	$G = \overline{A}$	NOT

UNSW SYDNEY

Arithmetic/logic unit



Arithmetic/logic unit

Operation Select				Operation	Function
S_2	S_1	S_0	C_{in}		
0	0	0	0	$G = A$	Transfer A
0	0	0	1	$G = A + 1$	Increment A
0	0	1	0	$G = A + B$	Addition
0	0	1	1	$G = A + B + 1$	Add with carry input of 1
0	1	0	0	$G = A + \overline{B}$	A plus 1's complement of B
0	1	0	1	$G = A + \overline{B} + 1$	Subtraction
0	1	1	0	$G = A - 1$	Decrement A
0	1	1	1	$G = A$	Transfer A
1	X	0	0	$G = A \,\&\, B$	AND
1	X	0	1	$G = A \mid B$	OR
1	X	1	0	$G = A \oplus B$	XOR
1	X	1	1	$G = \overline{A}$	NOT (1's complement)

The Shifter

The *shifter* shifts the value on Bus *B* by one bit to the left or by one bit to the right

Constructed using a combinational circuit so not dependent on the system clock

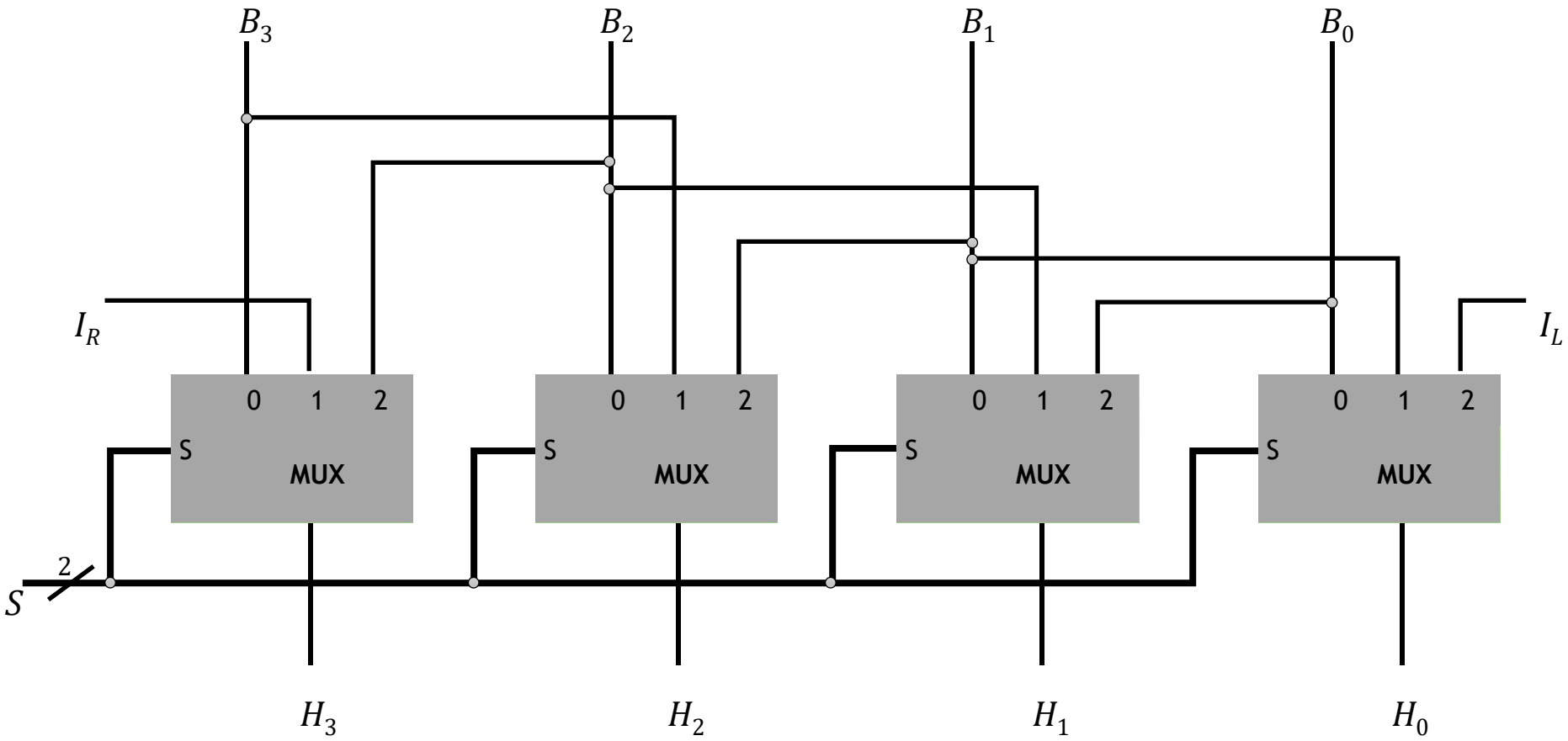
The shifter can be designed using *n* multiplexers with two select lines:

S = 00 – no shift

S = 01 – right shift

S = 10 – left shift

4-bit shifter



Datapath representation

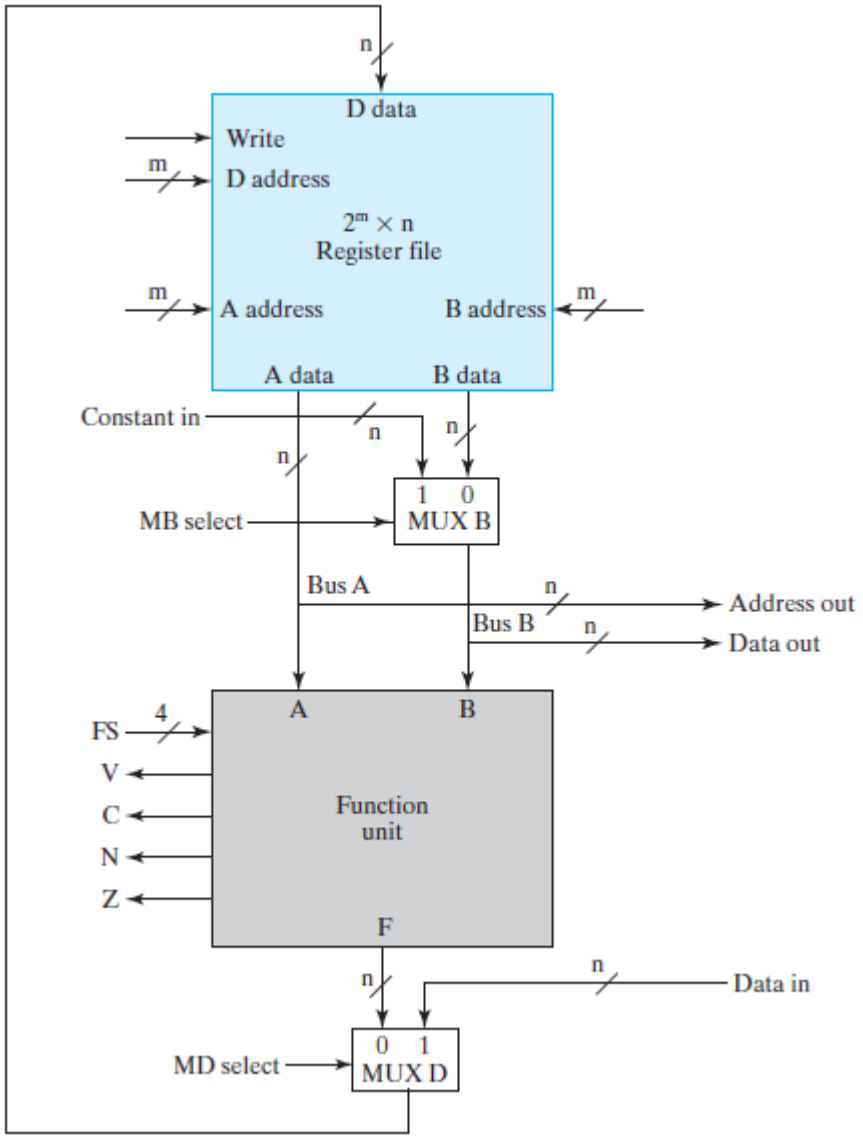
We can reduce the apparent complexity of the datapath by enclosing different functions into modules

Different implementations of the modules can then be interchanged without redesigning the entire datapath

All registers with common micro-operations are organised into a *register file*

Since the ALU and shifter are shared processing units, they are grouped together with the MUX into a shared *function unit*

Datapath representation



The size of the *register file* is $2^m \times n$ where m is the number of register address bits and n is the number of bits per register

The operation of the *function unit* is selected using a 4-bit *FS* (function select)

UNSW SYDNEY

Datapath representation

FS[3:0]	MF Select	G[3:0] Select	H[1:0] Select	Operation
0000	0	0000	XX	$F = A$
0001	0	0001	XX	$F = A + 1$
0010	0	0010	XX	$F = A + B$
0011	0	0011	XX	$F = A + B + 1$
0100	0	0100	XX	$F = A + \overline{B}$
0101	0	0101	XX	$F = A + \overline{B} + 1$
0110	0	0110	XX	$F = A - 1$
0111	0	0111	XX	$F = A$
1000	0	1X00	XX	$F = A \,\&\, B$
1001	0	1X01	XX	$F = A \mid B$
1010	0	1X10	XX	$F = A \oplus B$
1011	0	1X11	XX	$F = \overline{A}$
1100	1	XXXX	00	$F = B$
1101	1	XXXX	01	$F = B >> 1$
1110	1	XXXX	10	$F = B << 1$

Datapath representation

To fully specify the function unit, all the codes for *MF select*, *G select* and *H select* must be defined in terms of *FS*.

From the function table, can derive the internal inputs in the function unit in terms of the *FS* external input:

MF = FS[3]FS[2]

G[3:0] = FS[3:0]

H[1:0] = FS[1:0]

Final exam format

2 hours

4 questions – 25 marks each

Similar to mid semester exam

Topics covered

- Combinational circuits
- Sequential circuits
- Verilog
- Arithmetic circuits
- Digital integrated circuits
- Computer design fundamentals

Combinational circuits

$$\overline{W}\overline{X}Y + \overline{W}\overline{Y} + \overline{X}YZ + \overline{W}\overline{X}Y$$

Algebraic simplification

Simplification via K-maps

SOP, POS form

NAND, NOR implementation

Implementation using decoders/MUXs

GIC

Sequential circuits

Serial sequence detector, e.g. three consecutive bits

State diagram

Mealy, Moore state machines

State table

State minimisation

Implementation via flip-flops (JK/D/T)

Verilog

```
module mux_case(out,cntrl,in1,in2);
input cntrl,in1,in2;
output out;
reg out;

always @ *
case (cntrl)
1'b0:
out = in1;
1'b1 :
out = in2;
endcase
endmodule
```

- Describe what code does
- Explain specific parts of code
- Identify errors in code
- Add/modify lines of code

Arithmetic circuits

67 + 23 -57+110 23-49

Half and full adder

Binary ripple carry and look carry-ahead adder

1's and 2's complement

Unsigned and signed binary addition and subtraction

Overflow and Status flags

Digital integrated circuits

$$F(w,x,y,z) = \sum m(0,1,2,4,5)$$

BJT logic families – understanding how they operate

Fan-out, power dissipation, propagation delay, noise margin, cost

CMOS implementation of logic functions

Computer design fundamentals

Registers

Shifters – serial, parallel, bidirectional

Datapaths

Arithmetic Unit

Logic Unit

Shifter Unit

Datapath representation